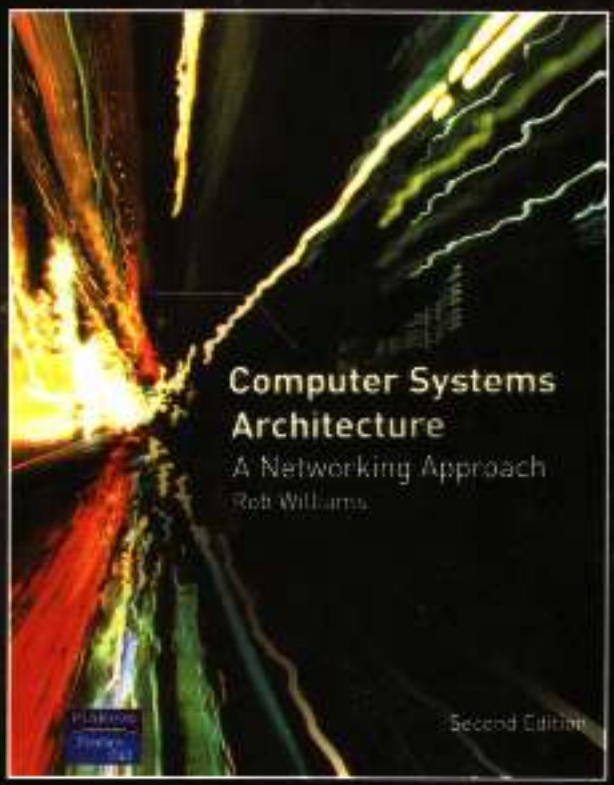
计 算 机 科 学 丛 书

原书第2版

计算机系统结构

(英) Rob Williams 著　赵学良 等译　丁艳 审校



Computer Systems Architecture
A Networking Approach
Second Edition

机械工业出版社
China Machine Press

计算机系统结构（原书第2版）

“计算机系统结构”是计算科学与工程及相关专业大学第一学年的基础性课程，历时两个学期，内容依次涉及数字逻辑电路、硬件外设、软件层、网络通信和操作系统。

本书紧密联系实际，注重动手实践，利用学生感兴趣和亲身体验过的技术来提高学习的积极性。同时强调了现代计算机与网络环境中其他设备的协作依赖关系，增加了对ARM和安腾（Itanium）处理器的介绍，以及数据通信延伸领域的最新内容。

本书特点

- 使用实际的处理器（奔腾处理器），学生在家中使用自己的机器就能够完成绝大部分的练习作业。
- 内容组织合理，深入浅出，材料取自于作者自己从事教学和实验工作的真实需求。
- 介绍数据传输和通信相关的思想和概念，为联网和网络通信相关的课程打下基础。
- 每章结束后的练习均经过精心挑选，本书的练习答案请登录华章网站www.hzbook.com下载。
- 书中用到许多现代的、商业化的实例，能够有效地激发读者学习的兴趣，并将理论与实际结合起来。

作者简介

Rob Williams 是位于英国布里斯托的西英格兰大学计算机系统技术学院院长。他在实时系统领域造诣颇深，同时还是GWE/GNE、Marconi Avionics和Nexos Office System的微处理器系统工程师。



ISBN 7-111-20417-4（第1版）
定价：69.00 元

上架指导：计算机/体系结构

ISBN 978-7-111-22356-6

9 787111 223566

ISBN 978-7-111-22356-6
定价：49.00 元

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

华章网站 http://www.hzbook.com

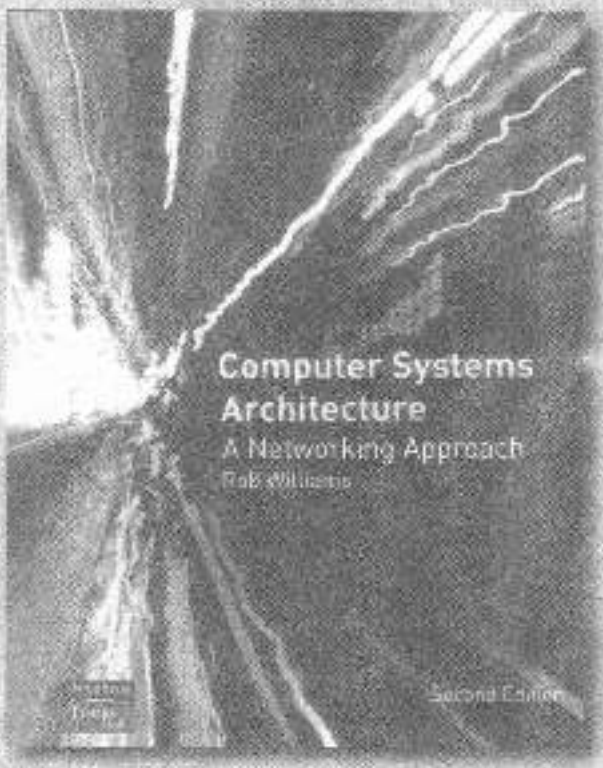
网上购书：www.china-pub.com

计 算 机 科 学 丛 书

原书第2版

计算机系统结构

(英) Rob Williams 著　赵学良 等译　丁艳 审校



Computer Systems Architecture

A Networking Approach

Second Edition

机械工业出版社
China Machine Press

本教材以与时俱进、自底向上的方式，依次介绍计算机系统结构的基本概念和基本内容，首先是数字逻辑电路和计算机硬件，接下来是运行于硬件之上的软件层，最后讲述通信和操作系统领域的基础知识。本书紧密联系实际，注重动手实践，利用学生感兴趣和亲身体验过的技术（如因特网、图形用户界面、移动通信等）来提高学生的学习积极性。贯穿全书，在分析系统的性能时注意将软件硬件结合起来讨论，练习题充分展示出硬件和软件之间这种相互影响、相互依赖的基本关系。

本书适合作为高等院校计算机及相关专业计算机系统结构课程的导论性教材。

Rob Williams: *Computer Systems Architecture: A Networking Approach*, Second Edition (ISBN 10: 0-321-34079-5, ISBN 13: 978-0-321-34079-5).

Copyright © 2006, 2001 by Pearson Education Limited.

This translation of *Computer Systems Architecture: A Networking Approach*, Second Edition (ISBN 10: 0-321-34079-5, ISBN 13: 978-0-321-34079-5) is published by arrangement with Pearson Education Limited.

All rights reserved.

本书中文简体字版由英国Pearson Education培生教育出版集团授权出版。

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：01-2006-6517

图书在版编目（CIP）数据

计算机系统结构：原书第2版 /（英）威廉斯（Williams, R.）著；赵学良等译. －北京：机械工业出版社，2008.1

（计算机科学丛书）

书名原文：Computer Systems Architecture: A Networking Approach, Second Edition

ISBN 978-7-111-22356-6

Ⅰ.计… Ⅱ.①威… ②赵… Ⅲ.计算机体系结构 Ⅳ.TP303

中国版本图书馆CIP数据核字（2007）第143545号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：刘立卿

北京瑞德印刷有限公司印刷·新华书店北京发行所发行

2008年1月第1版第1次印刷

184mm×260mm · 25.75印张

定价：49.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

机械工业出版社 华章公司
Huazhang Graphics & Information Co., Ltd

教师服务登记表

尊敬的老师：

您好！感谢您购买我们出版的_____教材。

机械工业出版社华章公司本着为服务高等教育的出版原则，为进一步加强与高校教师的联系与沟通，更好地为高校教师服务，特制此表，请您填妥后发回给我们，我们将定期向您寄送华章公司最新的图书出版信息，为您的教材、论著或译著的出版提供可能的帮助。欢迎您对我们的教材和服务提出宝贵的意见，感谢您的大力支持与帮助！

个人资料（请用正楷完整填写）

教师姓名		□先生 □女士	出生年月		职务		职称：□教授 □副教授 □讲师 □助教 □其他
学校			学院			系别	
联系电话	办公： 宅电： 移动：			联系地址及邮编			
				E-mail			
学历		毕业院校			国外进修及讲学经历		
研究领域							

主讲课程	现用教材名	作者及出版社	共同授课教师	教材满意度
课程： □专 □本 □研 □MBA 人数： 学期：□春□秋				□满意 □一般 □不满意 □希望更换
课程： □专 □本 □研 □MBA 人数： 学期：□春□秋				□满意 □一般 □不满意 □希望更换
课程： □专 □本 □研 □MBA 人数： 学期：□春□秋				□满意 □一般 □不满意 □希望更换

备注				
	已出版著作		译著	
	著书计划	方向一		
		方向二		
	是否愿意从事翻译工作 □是 □否		翻译方向	
	意见和建议			

填妥后请选择以下任何一种方式将此表返回：（如方便请赐名片）
地 址：北京市西城区百万庄南街1号 华章公司营销中心 邮编：100037
电 话：(010) 68353079 88378995 传真：(010) 88379578
E-mail：hzedu@hzbook.com marketing@hzbook.com 图书详情可登录http://www.hzbook.com网站查询

译 者 序

作为译者，大部分时间都是字斟句酌、小心翼翼、瞻前顾后、殚精竭虑，翻过书的最后一页后，才有机会舒口气，放松一下，考虑有什么东西需要告诉读者。编写序言是个很难得的机会，人蹲久了就要站起来伸伸手脚，翻译和做人莫不如此。

读书者，直奔主题者有之，细细品味者亦有之。想来，也会有读者与这篇序擦肩而过。过宝山而空回？我不敢说，别人也未必同意。凡译者序，皆会介绍原著，所不同者程度而已，或言过其实，或洋洋洒洒，或浅尝辄止，不一而足。

我曾为自己的第一本译作写了很长的序言，之后，逐次递减，一度不再为译作写序。后来有读者在留言中提到说会仔细地阅读序言，我才再做冯妇，又写下这篇不短的序。生活就好像一个圆，转了一圈后你会发现又来到起点。抽刀断水，逝者如斯，时过境迁，感慨良多呀！

科学的殿堂不存在拥有后就能睥睨天下、统制一切的魔戒，但却存在许多能够打开其大门的钥匙。本书无疑就是其中之一。钥匙当然可以分为很多类，但最根本的两类无疑是能够开锁的钥匙和不能够开锁的钥匙。我们需要学习书籍，但目的却不是书籍，而是书籍所讲述的知识领域。它就如同一扇窗、一幅地图、一本游记或是留有前人足迹的路，赋予读者一双能够透过迷雾看得更远的眼睛。

我有幸翻译了本书的第1版和第2版，由于第1版并未出版，所以读者也不用去寻找第1版的中译本。计算机技术发展的速度如此之快，以至于第1版时使用的一些数据（比如处理器的主频、存储器的速度和容量）在今天看来都有些遥远；第1版中的一些预测和想象，今天都变成了现实，甚至走得更远。第2版中在这些方面做了大量的改动。第1版出版后的这段时间内，计算机和网络对于人们工作、生活和娱乐的影响更加深入，64位桌面计算成为现实，移动设备（手机、PDA、MP3等）大量走入人们的日常生活，因特网对于社会的影响无孔不入，……。本书的第2版针对技术的新发展和出现的新事物也做了相应的调整和改动。

总体来看，本书具有以下三个特点：

- 内容新且全面，深浅适当；
- 注重实践，提倡学习与动手结合；
- 给出大量Internet链接，方便进一步的学习。

本书从第1版到第2版，仅仅几年内，就不得不做出大量的调整以适应最新的变化，可以预见，当读者将散发着墨香的本书捧在手中时，又会有许多新的事物涌现，计算机的发展可能又出现新的方向（比如多核等）。只有主动地适应这种变化才能跟上这个时代的步伐。

“计算机系统结构”这门课程是高校计算机及相关专业的一门重要的专业基础课，旨在从计算机组织和结构的角度出发，让读者学习和领会计算机系统。本书是作者多年讲授这门课程的经验集成，已在英国西英格兰大学等多所高校使用多年。此时，将本书引进到国内，也会丰富计算机体系结构教材的市场，使高校任课教师选用教材时又多了一个好的选择。

其实，越是基础性、入门性的课程，对学生的影响越深远，有很多人都有这样的体会：当离开学校十年以后再次聚首时，依旧觉得还是大学一年级时入门性课程对自己的影响最大，在以后的工作和学习中，这些基础知识会随读者接触不同的领域而浮现。“纸上学来终觉浅，绝知此事要

躬行”，在学习过程中要充分利用书后的练习、实践作业和课外读物，尽最大可能地缩小学习和工作之间的落差。

在这里，我要感谢机械工业出版社，以及所有让这本中译本出版成为可能的人。参与本书翻译和审校的人还有董健、靳友英、乔艳、乔颖等，也感谢他们为本书的出版做出的努力。

本书涉及面广、内容丰富，术语量大，但由于译者水平有限，译文中不当之处在所难免，诚请读者批评指正并不吝赐教。

赵学良

2007年4月

第2版前言

为变动如此之快的学科（如计算科学）编写教科书，是一份十分具有挑战性的体验。对于本书的第2版，一种极大的诱惑是，简单地创建一个执行查找替换任务的宏，由它根据情况插入或删除额外的零，以完成所有数值型参数的更新！尽管根本性的技术和原理没有太大的变化，但对当前处理器的速度及内存的大小有个清晰的认识，还是相当重要的。了解宽带连接能够提供的下载速度也比较有用。当我们对事物做出估计时，能够立即说出结果，并从常识的角度看看是否合理或者是否可行，是一种很有价值的技能。所有的科学家和工程师都在练习这项能力，程序员也不例外。我还清晰地记得，有一次我估计文件传输需要花费6个小时的时间，并因此失掉了一份商业合约。直到后来，我才意识到我忘记将这个数字除以10，因为我误将字节/秒当做位/秒。

在新版中，我专门为Intel 64位安腾处理器辟出一章，提供了更多的篇幅对它进行更详细的介绍，这是因为安腾处理器代表了后RISC时代的新型重要架构。引入术语RISC和后RISC，是为了更清楚地区分最初的简单但快速的处理器和最近更为复杂的继任者。本书采用ARM CPU作为微控制器的代表性架构，使用Intel StrongARM/XScale作为实际的例子。为了反映USB与程序员和计算机用户日益紧密的关系，本书对USB通信做了更多、更详细的说明。调制解调器技术已经做了更新，将56 k拨号设备和ADSL宽带设备包括了进来。便携式设备的广泛流行依赖于音频和视频压缩技术的飞速发展，本书专门为此增加一节内容，介绍了MPEG算法。介绍万维网和搜索引擎的部分已经过重新修订和更新，以Google为例进行讲述。

本书新增一章介绍并行技术、集群技术和网格计算，其中介绍了新型的IBM Cell处理器，这种处理器会催生游戏机产业的新生，或许最终会引发桌面计算的革命。

在准备新版的过程中，我有机会改正前一版中所犯的一些错误，但也有可能会引入一些新的错误。书中出现这些错误也有好的一面，就是我能够收到来自于全世界的电子邮件，指出书中的问题。再次感谢那些和我联系并指出书中错误的人。

在此，我还要感谢对本书做出主要贡献的两位编辑，第1版的编辑Keith Mansfield和第2版的编辑Simon Plumtree。没有他们的鼓励和努力，这些文字现在可能依旧以troff/pic文件的形式保存在我的硬盘上。

rob.williams@uwe.ac.uk

第1版前言

本书的由来

本书是根据第一学年的学位课程——计算机系统结构（Computer Systems Architecture，uqc104s1）而编写的，我在英国布里斯托尔的西英格兰大学（University of the West of England，UWE）讲授这门课程多年。这门课程历经扩展、压缩和再次扩展，所以它已经历过好几次彻底的整顿与重组。我们培养的许多毕业生都投入到不断发展壮大的电信产业中，人们对这一领域的兴趣日益高涨。因此，这门课程以及这本书反映出了这种倾向性，人们期望这门课程不仅适用于计算机科学，同时还要适应数据通信和联网的需要。

学生

十年前，当学生开始学习第一学年的课程时，他们在计算学科方面的资质和经历都十分有限；如今，超过半数的新生早已成功地完成计算领域的两学年课程（A等或BTEC）。他们可能注册参加了计算机科学（Computer Science）、软件工程（Software Engineering）或实时系统计算（Computing for Real-Time Systems）方面的学士学位课程。尤其是最后一项，吸引了那些对新的领域和事物（比如联网、Unix、嵌入式系统或实时程序设计）十分敏锐的技术狂热者。考虑到这些要求更高的听众，我尝试在课程中注入新的思想，并紧跟最新的技术动向。如果他们认为这门课程“太容易”或者“过时”，那么他们就会变得不耐烦并撤消该课程。本书尝试利用他们已有的经验，在此基础上进行教学。

另一项常常为学生所提及的顾虑，是关于课程中数学的位置。本课程不需要任何高级的数学知识，只是偶尔会用到简单的代数和算术知识。常识是最重要的资源！同样，我们也不会涉及到焊接。这并不是电子技术课程。

技术的进步

跟上技术前进的步伐，是所有计算机课程和书籍所面临的一个问题。所以，尽管奔腾处理器十分复杂，我还是选择基于奔腾处理器来编写本书，以应对这种挑战。那些看似能够在市场上保持领先地位好几年的系统，现在可能在发布后数月内被取代。许多软件工具，在大学等研究机构有机会获得拷贝之前，已经被从事商业开发的程序员研究并采用了很长时间。因此，计算领域内的课程需要被不断地评估审查，以在更广泛的范围内保持其时效性。我的学生们可能会回家去使用他们自己的计算机系统，因为他们自己的计算机系统比大学实验室中的系统要先进得多，这并不奇怪。但是，使我忧虑的是，计算领域变得越来越赶时髦，这有可能会导致在课堂上讨论过的思想和例子被认为是过时的东西，尽管它们在学术上和理论上完全正确。基于这种考虑，我希望在本书中引入了足够多的、新的、现代的材料，从而能够保持对学生的吸引力。

本书的使用

我们的学习方法虽然各不相同，但有效使用文字的能力，很久以来一直是现代文明的核心。我们都从记录在纸张上的别人的经验中受益很多。忽略这个巨大的资源等同于自残手足。万维网（World Wide Web，WWW）的引入对每个人的读写能力提出了更高的要求。大多数网页依旧严重依赖于文字来表达有效的信息。尽管一幅图可能胜过千言万语，但最初的认知常常来自于伴随的文

字信息。

在每一章的开头，都用一幅图来表达该章的内容，模仿我们熟悉的Windows Explorer文件浏览器。通过它，我们可以清晰地了解每一章在学科中的上下文（前后关系）。它不是一份内容清单，相反，它更像是一个表意符号。每一章后面都给出一系列问题，答案在本书最后提供[⊖]。读者并非一定要认认真真地自己解答每个问题，然后再去看答案！在本书结尾处，还提供一份完整的术语表，提供它的意图是帮助读者应对大量的术语。我们都对这种情况感到遗憾，然而，这是成为全面的计算机专业技术人员所必须面对和克服的困难之一，不能简单地加以忽略。

操作系统和编程语言

在讲授一门课程或在计算机上撰写文字时，选择使用哪种语言或操作系统，可能会是一件具有较强感情色彩的事情。传统上，大学课程一直尽量避免与商业相关的问题，坚持使用被学术界广泛接受的事物。我曾经试图在履行一份商业合同时使用Pascal，接下来遇到了各种各样完全能够预料的问题，但是，这些问题我们从未真正和我们的学生论及过，这的确让人备感挫折。有过这种经历以后，我就一直力图在我的教学中使用在商业上可行的工具。全世界到处都可以见到的两种操作系统是Unix和Windows NT（包括Windows 2000）。我希望读者最好能够接触到所有的系统，以便于更好地完成实习作业。Linux是Unix的绝佳例子，尽管我在这本教材中使用Sun Microsystem的Solaris进行讲解。对于语言，我假定读者在学习计算机系统课程的同时，正在学习一门编程语言，因而能够很快具备C语言的基本技能，能够理解我给出的代码片段。但我并不期望每个人都是C++专家。

为了帮助读者测试书中给出的例子，并试着解决每章末尾给出的问题，本书包括了Microsoft Developer Studio Visual C++的学生版本。附录给出一些指导性的注解，以帮助读者安装软件包并快速地学会如何使用它。尽管我们只用它来编写一些十分基本的程序示例，但是如果读者有足够的精力，也可以继续学习C++、面向对象编程，以及使用MFC（Microsoft Foundation Classes，微软基本类库）等。同时，为了支持读者的学习，本书还提供了源代码的示例，以及一个十分有效的在线学习系统。

面向实践的课程定位

读者正在学习的课程与我们的课程可能会在许多方面存在不同。我规划了24周的课程，拆分成两个学期。这样，每一章服务于一周的课程。课程的次序以及课程的扩展，由每周实验课中所进行的实习作业而定。我相信，基本上是实习作业充分地巩固了对技术的理解。我永远不会相信未经实际工作检验的软件设计！减少实验课，或者将它们与授课过程中进行的理论性论述隔离开来，都会弱化我们理解和吸收新观点的能力。我们都有各自不同的学习方式，有些人喜欢听，有些人喜欢读，等等。就个人而言，我一直觉得实践活动能够使人对新概念的理解更牢固和持久。

“听过会忘记，看过才会牢记，参与过才能理解个中含义。”

致指导老师

学生读者

在写这本书时，我所考虑的典型读者是计算科学（computing）或相关领域第一学年的学生。他们至少会参加某些您每周讲授的课程，完成练习题，阅读教材并与他们的朋友讨论作业。他们还会经常访问因特网，可能在家中，也可能是在学校。但是，依据我的经验，要想通过这门学位课程，仅仅依靠上述活动中的某一项是不够的。另外，您的学生也不会只学习计算机系统和联网，很可能他们还会同时参加程序设计、系统设计、数学方法以及电子学等课程。一门成功的课程应该帮助学

⊖ 为缩减中文版的篇幅并适应国内高校的要求，答案部分不会放在纸版的中文版中，而是会放在网站www.hzbook.com上，供读者自由下载。——编者注

生将这些经验都融汇贯通，并且鼓励不同学科间的相互对照。这正是本书能够帮助他们做到的事情，他们在本书中能够找到指向他们正在学习的其他领域的链接及路标。

和许多其他课程一样，我们的学位课程，被集成到一个可以灵活组合的方案中，其中还为学生提供第二学年的一些选择。我发现，许多学生在第一学年中需要这些可选课程的简要介绍，以便做出正确的决定。尤其是系统管理、操作系统和联网，可能是学生完全陌生的领域。本书的目的就是在给予学生计算系统实践和原理方面基础知识的同时，也起到“基层性课程”的作用。

实习作业单

我在本书的配套网站（www.pearsoned.co.uk./williams）上，提供我授课时使用的每周作业单样本。它们注重于在刚讲授的课程中所涉及的理论性知识的各个方面。作业单常常也是后续课外作业的引导。如您所见，我们依旧将学生的实验室作业与他们的课堂知识紧密地结合起来，尽管这对于组合式课程常常是难以做到的。

评估考核

由于本课程的实践面向性，我设置了两套经过评估的程序设计作业。现在，为第一学期准备的是一份汇编程序的练习，使用Visual C Developer Studio。它的内容涉及使用PC的COM端口进行串行通信。第二个作业是一组练习，构建在第一个作业的基础之上。它要求编制软件来支持面向分组的环形网络，同样使用PC的COM端口。第二个程序设计作业完全使用C语言，涉及到协议协商和协同工作。这份作业的目的是为第二学年与联网相关的课程以及最后学年的分布式系统方面的课程打基础。作为课程评测的一部分，我要求学生演示他们的代码，并回答一系列与结构和功能相关的问题。虽然这样做需要许多时间，但就我的经验而言，这是一项值得做的投资。

相对于仅在期末进行一次考试，我更倾向于设置几个课堂测试。因为这样做能够定期给学生有用的反馈信息，同时允许教师及时给予纠正。每一章后面列出的问题，能够帮助学生为测试进行准备，同时在本书最后对答案进行了汇总。测试的样本可以从出版商的网站得到，相关的内容受到保护。

致谢

感谢Phil Naylor提供第13章中管理脚本的例子，以及为我们的由Sun工作站组成的网络提供出色的维护。我还要真挚地感谢我的同事Bob Lang和Craig Duffy，他们耐心地阅读了本书的早期草稿，并给出了十分有用的意见。在此，我还要提及参加过实时系统计算学士学位课程的许多学生，不管是以前的学生还是现在的学生，他们的幽默感和决心常常使耗时很长的调试过程变得轻松愉快。对于任何教师来说，看到学生进步，看到他们在技术上更加自信，成功地毕业并开始有益的事业，就是最好的回报。

随着本书的成长，我妻子由最初的怀疑态度也变为了温和的容许，接下来，随着文字铺满起居室的地板，她又变得惊慌和不相信。尽管她也对本书做了校对及编辑上的努力，但所有的错误都是我自己的，我希望您能够将您的意见和观点反馈给我（rob.williams@uwe.ac.uk）。

我还得赞扬Brian Kernighan，因为他的pic语言真是太棒了，我使用它设计了文中所有的线路图。最初的正文是用emacs编辑的，格式编排工作都是使用Richard Stallman GNU程序组中的groff完成的。正是在本书的编写过程中，我体会到了使用pic编制图表的乐趣。

Rob Williams

西英格兰大学计算科学系，英国布里斯托尔

2000年7月

本书的出版商要向下面的学者表示感谢，他们为本书提供了极有价值的建议和鼓励：

Hernk Corporaal

荷兰Delft大学
Peter Hood
英国Huddersfield大学
Prasant Mohaptra
美国密歇根州大学
Henk Neefs
比利时Gent大学
Andy Pimentel
荷兰阿姆斯特丹大学
Mike Scott
爱尔兰都柏林城市大学
Bernard Weinberg
美国前密歇根州立大学

出版商致谢

我们要感谢下面的机构允许我们使用相关的受版权保护的资料：

图9-2由Intersil公司授权使用，Intersil Corporation版权所有；图12-3来自于Hatfield，D.J.和Gerald，J.（1971）“Program restructuring for virtual memory”IBM系统期刊，第10卷，No.3，189页，国际商业机器公司（IBM）版权所有（1971），由IBM系统期刊授权复印；图15-14和15-15中Netscape Communicator浏览窗口由Netscape Communications公司版权所有（2005），使用得到许可。Netscape Communications公司并未批准、赞助、支持或赞成本书的出版，并且也不对本书的内容负责；图15-20和表15-6来自于The Anatomy of a Largescale Hypertextual Web Search Engine（Brin S. and Page L, 2000），获得Google的许可；图21-19得到英国IPAQ Repair and Parts，Ratby的许可；屏幕截图的出版获得微软公司的许可。

在某些情况下，我们无法找到受版权保护的资料的所有者，如果有人能够给予我们任何相关的信息，帮助我们找到版权的所有者，我们将会十分感激。

目　录

译者序
第2版前言
第1版前言

第一部分　计算机的基本功能及其构成

第1章　导论：软硬件接口 ……1
1.1　计算机系统及网络通信的重要性 ……1
1.2　硬件和软件的互相依赖 ……2
1.3　硬件编程：VHDL ……3
1.4　人人都应了解的系统管理问题 ……4
1.5　语音、图像和数据：技术的趋同现象 …5
1.6　窗口界面（WIMP） ……5
1.7　因特网：连接所有的网络 ……7
1.8　使用PC：学习CSA的更多理由 ……9
1.9　小结 ……10
实习作业 ……10
练习 ……10
课外读物 ……11
第2章　冯·诺依曼体系结构的特征 ……13
2.1　以2为基：二进制的优点 ……13
2.2　程序控制存储：通用机器 ……13
2.3　指令代码：控制机器动作的指令系统 ……14
2.4　转换：编译器和汇编器 ……15
2.5　链接：将程序组合到一起 ……16
2.6　解释器：执行高级命令 ……16
2.7　代码共享和重用：不要总是从头做起 ……17
2.8　数据编码：数值和字符 ……18
2.9　操作系统：Unix和Windows ……20
2.10　客户机服务器计算：网络时代的方式 ……22
2.11　可重配置硬件：读取－执行的另一种替代方式 ……23
2.12　小结 ……23
实习作业 ……23
练习 ……24
课外读物 ……24
附录：以11为基的计数 ……25
第3章　功能部件和读取－执行周期 ……26
3.1　各部分的命名：CPU、存储器、IO单元 ……26
3.2　CPU的读取－执行周期：高速且单调 …29
3.3　系统总线：同步或异步 ……31
3.4　系统时钟：指令周期时序 ……32
3.5　预取：前期工作以使速度得到提高 …34
3.6　存储器长度：寻址宽度 ……35
3.7　字节次序：微软与Unix，以及Intel与Motorola ……36
3.8　简单的输入输出：并行端口 ……38
3.9　小结 ……38
实习作业 ……38
练习 ……39
课外读物 ……39
第4章　构成计算机的逻辑电路：控制单元 ……40
4.1　电子积木和逻辑电路：模块化器件的优点 ……40
4.2　基本逻辑门 ……40
4.3　真值表和多路复用器：简单但有效的设计工具 ……42
4.4　可编程逻辑器件：可重新配置的逻辑芯片 ……44
4.5　交通灯控制器：无法避免 ……46
4.6　根据真值表实现电路：一些实用提示 …47
4.7　译码器逻辑：控制单元及存储器的根本所在 ……48
4.8　CPU控制单元：“核心“ ……49
4.9　洗衣机控制器：简单的CU ……49
4.10　RISC与CISC译码：使计算机的处理速度更快 ……52
4.11　小结 ……53
实习作业 ……53
练习 ……54
课外读物 ……54

第5章 构成计算机的逻辑电路：算术逻辑单元 ……55
5.1 德·摩根等价定律：逻辑互换性 ……55
5.2 二进制加法：半加器、全加器、并行加法器 ……55
5.3 二进制减法：2的补码的整数格式 ……57
5.4 二进制移位：桶形移位器 ……58
5.5 整数乘法：移位和相加 ……60
5.6 浮点数：从极大到极小 ……63
5.7 小结 ……65
实习作业 ……66
练习 ……66
课外读物 ……66
第6章 计算机的逻辑构成：存储器 ……67
6.1 数据存储 ……67
6.2 存储设备 ……68
6.3 静态存储器 ……69
6.4 动态存储器 ……70
6.5 DRAM刷新 ……71
6.6 分页访问存储器：EDO和SDRAM ……71
6.7 存储器映射：寻址和译码 ……73
6.8 IO端口映射 ……75
6.9 小结 ……77
实习作业 ……77
练习 ……77
课外读物 ……78
第7章 Intel奔腾CPU ……79
7.1 奔腾：高性能的微处理器 ……79
7.2 CPU寄存器：数据和地址变量的临时存储区 ……82
7.3 指令集：基本奔腾指令集简介 ……85
7.4 指令的结构：CU如何理解指令 ……86
7.5 CPU状态寄存器：十分短期的存储空间 ……87
7.6 寻址方式：构建有效地址 ……89
7.7 执行流水线：RISC加速技术 ……90
7.8 奔腾4：扩展 ……91
7.9 Microsoft Developer Studio：调试器的使用 ……92
7.10 小结 ……94
实习作业 ……95
练习 ……95
课外读物 ……96
第8章 子例程 ……97
8.1 子例程的目的：节省空间和精力 ……97
8.2 返回地址：堆栈的引入 ……98
8.3 使用子例程：HLL程序设计 ……99
8.4 堆栈：大多数操作的基本要素 ……100
8.5 参数传递：将子例程具体化 ……100
8.6 堆栈框架：所有局部变量 ……102
8.7 对HLL的支持：CPU针对子例程处理的特性 ……104
8.8 中断服务例程：由硬件调用的子例程 ……104
8.9 访问操作系统例程：后期绑定 ……105
8.10 小结 ……106
实习作业 ……106
练习 ……106
课外读物 ……106
第9章 简单的输入输出 ……107
9.1 基本IO方法：轮询、中断和DMA ……107
9.2 外设接口寄存器：程序员的角度 ……108
9.3 轮询：单字符IO ……111
9.4 中断处理 ……114
9.5 关键数据的保护：如何与中断通信 ……119
9.6 缓冲IO：驱动中断设备的驱动程序 ……122
9.7 直接内存访问：自治的硬件 ……123
9.8 单字符IO：屏幕和键盘例程 ……124
9.9 小结 ……125
实习作业 ……126
练习 ……126
课外读物 ……126
第10章 串行通信 ……127
10.1 串行传输：数据、信号和时序 ……127
10.2 数据的格式：编码技术 ……127
10.3 时序同步：频率和相位 ……129
10.4 数据编码和错误控制：奇偶校验、检验和、汉明码和CRC ……131
10.5 流量控制：硬件和软件方法 ……137
10.6 16550 UART：RS232 ……138
10.7 串行鼠标：机械或光学 ……142
10.8 串行端口 ……144
10.9 USB：通用串行总线 ……144
10.10 调制解调器：载波调制 ……147
10.11 小结 ……151
实习作业 ……152

练习……152
课外读物……152
第11章 并行连接……153
11.1 并行接口……153
11.2 Centronics：大于打印端口但小于总线……153
11.3 SCSI：小型计算机系统接口……155
11.4 IDE：智能驱动电路……158
11.5 AT/ISA：计算机标准的成功案例…158
11.6 PCI：外设部件的互连……160
11.7 即插即用：自动配置……162
11.8 PCMCIA：个人计算机存储卡国际联盟……163
11.9 小结……164
实习作业……165
练习……165
课外读物……165
第12章 存储体系……166
12.1 系统的性能……166
12.2 访问局部化：利用重复……167
12.3 指令及数据的高速缓存：匹配内存和CPU的速度……171
12.4 高速缓存映射……172
12.5 虚拟内存：分段和按需页面调度…174
12.6 地址公式化：时间、地点和数量…178
12.7 硬盘使用：参数、访问调度和数据安排……179
12.8 性能提高：块、高速缓存、碎片整理、调度、RAM磁盘……181
12.9 光盘：CD-DA、CD-ROM、CD-RW和DVD……182
12.10 DVD：数字通用光盘……184
12.11 MPEG：视频和音频压缩……185
12.12 闪存：新型软盘……190
12.13 小结……190
实习作业……190
练习……191
课外读物……191

第二部分 网络通信及复杂性的增加

第13章 程序员的观点……193
13.1 不同的观点与不同的需求……193
13.2 应用程序用户及办公软件包……193
13.3 系统管理：软件安装和维护……195
13.4 HLL程序员：Java、C++和BASIC…198
13.5 系统编程：汇编和C……200
13.6 硬件工程师：硬件的设计和维护…202
13.7 分层虚拟机：体系结构简介……202
13.8 汇编器：简单的转换器……203
13.9 编译器：转换及其他诸多工作……204
13.10 小结……205
实习作业……205
练习……205
课外读物……206
第14章 局域网……207
14.1 用户之间的纽带：电子邮件、打印机和数据库……207
14.2 PC网络接口：布线和接口卡……210
14.3 以太网：带冲突检测的载波侦听、多路访问……213
14.4 局域网的寻址：逻辑和物理方案…215
14.5 主机名：另外一个转换层……217
14.6 分层和封装：TCP/IP软件堆栈……217
14.7 网络文件系统：跨网络共享文件…218
14.8 网络的互连：网关……219
14.9 socket编程：WinSock简介……220
14.10 小结……222
实习作业……223
练习……223
课外读物……224
第15章 广域网……225
15.1 Internet的起源……225
15.2 TCP/IP基本协议……226
15.3 TCP错误处理和流量控制……229
15.4 IP路由：数据包如何找到正确的路径…230
15.5 DNS：分布式域名数据库……234
15.6 万维网的起源……236
15.7 浏览Web：Netscape Navigator……236
15.8 HTTP……239
15.9 搜索引擎Google……241
15.10 操作系统互连：一种理想的方案…243
15.11 小结……245
实习作业……245
练习……245
课外读物……246
第16章 其他网络……247
16.1 PSTN：电话网络……247

16.2 Cellnet：移动通信提供商 ……251
16.3 ATM：异步传输模式 ……257
16.4 消息传递：无线寻呼和分组无线网络 ……260
16.5 ISDN：全数字 ……261
16.6 DSL：数字用户线路 ……264
16.7 有线电视：数据传输设施 ……264
16.8 小结 ……266
实习作业 ……267
练习 ……267
课外读物 ……267
第17章 操作系统 ……269
17.1 历史渊源：基本功能的发展 ……269
17.2 Unix：操作系统的里程碑 ……271
17.3 概要结构：模块化 ……273
17.4 进程管理：初始化和调度 ……273
17.5 调度决策：时间片划分、抢先和协作 ……277
17.6 任务通信：管道和重定向 ……278
17.7 排斥和同步：信号量和信号 ……279
17.8 内存分配：malloc()和free() ……283
17.9 用户界面：GUI和外壳 ……284
17.10 输入输出管理：设备处理程序 ……285
17.11 小结 ……287
实习作业 ……287
练习 ……287
课外读物 ……288
第18章 Windows XP ……289
18.1 Windows GUI：满足用户的需求 …289
18.2 Win32：推荐的用户API ……290
18.3 进程和线程：多任务 ……290
18.4 内存管理：虚拟内存的实现 ……291
18.5 Windows注册表：集中化的管理数据库 ……291
18.6 NTFS：Windows NT文件系统 ……293
18.7 文件访问：ACL、权限和安全 ……293
18.8 共享软件组件：OLE、DDE和COM …295
18.9 Windows XP主机：Winframe终端服务器 ……295
18.10 小结 ……296
实习作业 ……296
练习 ……296
课外读物 ……296
第19章 档案管理系统 ……298
19.1 数据存储：文件系统和数据库 ……298
19.2 PC文件分配表：FAT ……303
19.3 Unix索引节点：不同的方式 ……305
19.4 Microsoft NTFS ……308
19.5 RAID：更安全的磁盘子系统 ……309
19.6 文件安全：访问控制 ……311
19.7 CD可移植文件系统：多个区段内容清单 ……312
19.8 小结 ……313
实习作业 ……313
练习 ……313
课外读物 ……314
第20章 图形输出 ……315
20.1 计算机和图形：捕获、存储、处理和重现 ……315
20.2 PC图形接口卡：图形协处理器 ……320
20.3 激光打印机：机电一体化 ……323
20.4 Adobe PostScript：页面描述语言 …325
20.5 WIMP：重塑计算机的形象 ……327
20.6 Win32：图形API及其他 ……328
20.7 X窗口系统：分布式处理 ……329
20.8 MMX技术：辅助图形计算 ……329
20.9 小结 ……330
实习作业 ……330
练习 ……331
课外读物 ……331
第21章 RISC处理器：ARM和SPARC …332
21.1 RISC的优点：更高的指令吞吐量 …332
21.2 流水线技术：更多的并行操作 ……335
21.3 超标量方法：并行的并行 ……336
21.4 寄存器存储：更多的CPU寄存器 …336
21.5 分支预测方法：流水线的维护 ……338
21.6 编译器支持：RISC的重要组成部分…339
21.7 ARM 32位CPU的起源 ……339
21.8 StrongARM处理器：32位微控制器…345
21.9 HP iPAQ：StrongARM PDA ……347
21.10 Puppeteer：StrongARM SBC ……348
21.11 Sun SPARC：RISC架构的标量处理器 ……350
21.12 嵌入式系统：交叉开发技术 ……351
21.13 小结 ……352
实习作业 ……352

练习 ……352
课外读物 ……353
第22章 VLIW处理器：EPIC安腾 ……354
22.1 安腾64位处理器简介 ……354
22.2 安腾汇编语言：对CPU控制更多 …359
22.3 运行时调试：gvd/gdb ……363
22.4 未来的处理器设计 ……364
22.5 小结 ……364
实习作业 ……365
练习 ……365
课外读物 ……365
第23章 并行处理 ……366
23.1 并行处理基础 ……366
23.2 指令级并行：流水线化 ……368
23.3 超标量：多执行单元 ……368
23.4 未来的对称、共享内存并行处理 …368
23.5 单芯片多处理器：IBM Cell ……370
23.6 集群和网格：应用级并行 ……372
23.7 小结 ……373
实习作业 ……374
练习 ……374
课外读物 ……374
附录 Microsoft Visual Studio 8 Express版 …375
术语表 ……383
参考文献 ……396
习题答案[⊖]

⊖ 答案部分已放到网站www.hzbook.com上，供读者下载。——编者注

第一部分 计算机的基本功能及其构成

第1章 导论：软硬件接口

在大学中开始学习新课程时，即使不考虑应付全新学科所要做的工作，开学第一周也会非常忙乱。因此，作为引导性的章节，本章只论述计算机系统这门学科涵盖了哪些领域，与联网的相关性，以及为什么要学习它。本章的重点是硬件和软件的相互影响，它是影响计算机整体性能的重要因素。这可能不是一个为我们所熟悉或容易论述的话题，因此，下面给出一些理由，说明为什么要对这门学科有相当深度的了解。这些理由包括：掌握了它，才能更好地维护自己的计算机；才能跟上技术的最新进展，包括互连互通性和Internet；才能对我们使用的窗口界面有更深刻的认识；以及为将来不可避免的技术上更为深远和广泛的变革做好准备。硬件和软件开发人员之间建立起来的这种十分成功的协作关系，还将一直进行下去。

1.1 计算机系统及网络通信的重要性

在大学课程中，经常会有一门介绍性的课程，称为"计算机系统结构"（Computer System Architecture，CSA）。但是，由于对CSA尚不存在统一的定义，因而各种各样的课程以及不同的解读之间，存在相当大的差距（见图1-1）。加之商业词汇的大量产生，使得学生在认识上的混淆日益增长。有时，CSA出现在面向硬件的数字电子技术中。有时，它又会被披上某个系列计算机的统一软件规格说明的外衣。不管是计算机设计人员还是最终用户，都对网络设施的决定性作用缺乏足够的认识，尽管我们都认识到它在社会活动中的重要性日益增长。毫无疑问，越来越多的计算机科学专业毕业的学生涉足到数据通信行业中，并且得益于在计算机领域内打下的基础。因此，本书的目标之一，就是将网络紧密地揉合到CSA中。

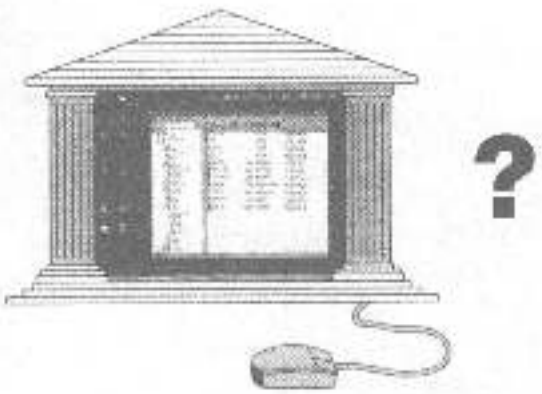


图1-1 计算机系统体系结构

很明显，计算机和网络都要求硬件和软件共同参与才能完成工作。但是，由于以往学术界对这两者的分割，造成现今在讲述这类课程时，面临一个困难的平衡问题。两者都同样重要，并且都受到各自热心支持者的强有力拥护。硬件和软件之间的差别，可以比做是球队的正式合影（在家门前快速摆好姿式）和世界杯决赛那激动人心的不可预知性之间的遥远联系。球员的静态照片只是含糊地暗示出对抗性赛事无限的可能性。随着计算机硬件的复杂度不断增长，分解和钻研旧的计算机已经不再受到鼓励。目前，电子部件的价格成倍下降，加上新款游戏对最新设备的需求，成就了销售人员的梦想。令人想不到的是，这种趋势尽管吸引更多的人使用计算机，但对于刚进入大学的计算科学（computing）专业的学生学习和掌握基础知识，却会造成负面影响。尽管我们不能让时间回退到那个由业余爱好者自己组装计算机的年代。然而，对于那些想要专业地使用计算机的人，有关硬件（hardware）和软件（software）交互作用的知识依旧有用，甚至必不可少。

对周围的计算机系统、Internet以及我们日益依赖的移动电话网络的好奇心和求知欲，促使我们探究和思考在软件和电子技术领域到底发生了什么。未来几年内我们可能使用到的设施，极大地依赖于当前在微电子和软件设计方法论方面的进展，通过它们，我们能够看到未来。

贯穿全书，我们都把CSA看做是研究硬件和软件交互作用的学科，这种交互作用决定了联网计算机系统的表现。我们还会力图展示，我们总是可以将计算机看做是分层组织的系统，为了更完整地理解它们的运作，我们还可以进一步将它们分解成更简单的组成部分（硬件或软件，如图1-3所示）。不同于其他研究领域，如物理和化学，在计算机领域，复杂的思想总是能够拆分成更简单的容易理解的概念。这种渐进式分解的方式，不仅仅在研究计算机时有效，而且在设计和构建新系统时，同样具有非常宝贵的价值。

1.2　硬件和软件的互相依赖

尽管对于计算机系统涉及到硬件和软件已经达成了广泛的共识，但是大学的计算机课程依旧很少要求学生对这两个领域都要有相当的了解。或许拿吃半个熟鸡蛋来做比喻比较恰当——我们需要冒吃不到蛋黄的风险（见图1-2）。这种划分，或者说是专业化，带来了一系列的负面后果。当我们分别按照硬件工程师和程序员招募开发团队时，这两个阵营间互相敌对的裂缝有可能会越来越大。由于硬件工程师和软件工程师所采用的方式和词汇的不同而造成的简单误解，可能造成专业上的对抗。当出现问题时（在这种情况下几乎是不可避免的），一个阵营可能会责备另外的阵营，进而使得问题需要更长的时间才能得到解决。程序员有时会发现在未经磋商的情况下，选择了不合适的设备；与此同时，当软件不能利用硬件设计人员革命性的新电路所提供的性能优势时，程序员可能根本就无能为力。

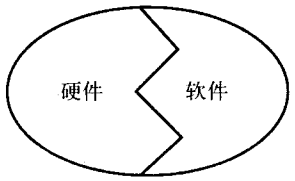


图1-2　硬件和软件都不可或缺

一些商业分析师声称，硬件制造已经没有太大的商业意义，利润的来源在于程序设计：它将世界带入到系统软件的开发中。但是，现今很明显的是，在这样一个快速变动的世界中，对新硬件设计的及时了解，将有助于在软件行业中抢占市场先机。第一个利用新硬件提供的某种功能的软件产品，无疑会在市场上占据领导地位。从长期来讲，忽视计算产业中的硬件方面，不会有任何好处。理解基本的原理，领会现代技术在一系列最近产品中的应用，是本书的主要目的。如果程序员忽视硬件方面的发展，将会承担风险。相反的情形是，如果忽视软件，也会导致类似的失败。个人计算机（Personal Computer，PC）运行Windows操作系统之后，在商业上取得了巨大成功，最近人们看到，自Netscape Navigator浏览器发布以来，Internet的应用取得了爆炸式的发展。与之相对应的是，许多优异的机器因为它们非标准的软件，在商业上获得惨败。除了一些公开的广为流传的案例外，成百上千强调硬件和软件应该共同开发的尝试也成为灾难。我们现在认识到，尽管它们在技术上存在优越性，但是计算机系统可能会由于许多原因而不被人们接受，比如设计不完善的用户界面、应用软件的缺乏、操作系统的不恰当选择等。最近的许多进展是由于硬件和软件的同时进步而取得的：窗口界面只有通过复杂精致的软件和强大的图形卡才能实现；网络连接要由独立的协处理器与复杂的驱动程序例程一同提供支持；正是由于PostScript解释程序支持静电印刷打印引擎后，激光打印机才变得到处流行。还有许多这样的例子，可以说明保持硬件和软件开发并排进行的价值。在探询硬件和软件的交互作用时，获得对相关设施的访问正变得越来越困难。大型的、多用户的计算机，为保护其他用户，不允许普通程序员访问其硬件和关键的软件，这样做完全可以理解。然而，随着Windows NT的引入，这类安全约束已被纳入到单用户的个人工作站，从而使得直接访问硬件变得不可能。只有操作系统的代码（见图1-3）拥有这种特权，普通的程序必须调用“受信任的”操作系统例程，来完成对任何硬件的读写。



图1-3 硬件之上的软件层

1.3 硬件编程：VHDL

戈登·摩尔是Intel的创始人之一，他提出一个著名的经验性公式来描述硅技术的快速发展。这个公式就是以他名字命名的定律——摩尔定律，声称可以集成在给定大小芯片上的电路数量（晶体管数量）大约每两年增长一倍。图1-4给出了前十年间一些具有代表性的电路的数据。注意，纵坐标轴不是线性的，因此，点都沿着一条直线排列，24个月前设计的电路，现在可以缩小到原来一半的面积之内。Intel最初的4004处理器集成了2300个晶体管，而奔腾4处理器大约拥有4200万数量级的晶体管。但芯片的面积并没有因此而增长近20000倍！渐进地缩小电子电路大小的能力，使得芯片的成本不断下降，因为在单片硅圆上可以加工更多的电路，但是，技术上的进步更多地被用来增强芯片的功能。Intel在将8086和8087合并，以及后来引入一级（Level 1，L1）缓存和之后的二级缓存时，都采用这种方式。

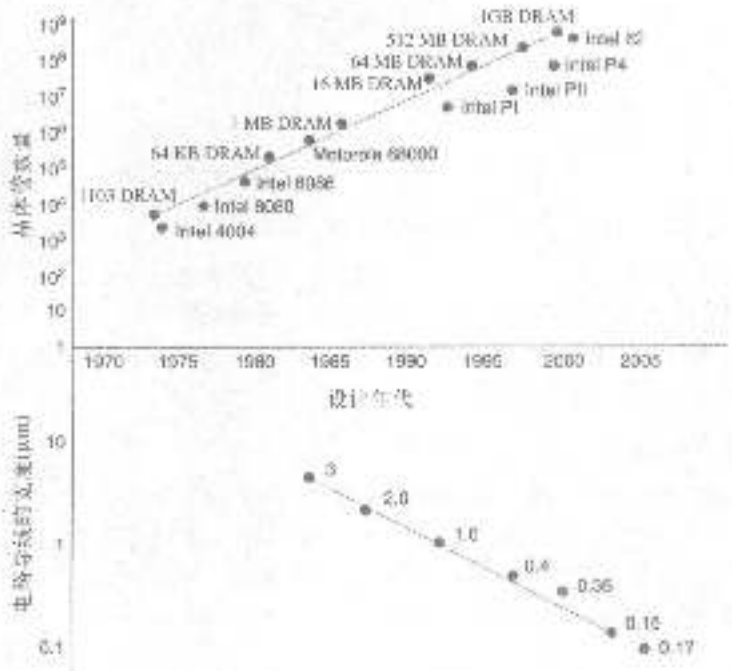


图1-4 技术发展的摩尔定律

令人吃惊的是，从20世纪70年代早期至今，这条定律一直有效，并且有可能会持续到21世纪20年代，直到电路元件的大小变得太小，以至于量子物理的“海森伯格测不准原理”介入为止。尽管

芯片上电路之间的连接已经只有0.25μm宽，绝缘层可能仅有几十个分子那么薄，但是摩尔定律依旧神奇地保持了下来。造成这种发展趋势的底层因素十分复杂，包括维护超清洁制造环境的能力、国际贸易壁垒的降低、更高分辨率光蚀刻法的发展，以及游戏机平台在营销上的成功！

尽管与硬件相关的事物和与软件相关的事物之间的分裂已经根深蒂固，但是一些新的进展可能会扭转这种趋势。程序设计语言**VHDL**（Very high speed integrated circuits Hardware Description Language，超高速集成电路硬件设计描述语言）和Verilog已经开始用来描述硬件电路的结构及功能。图1-5给出了一段用VHDL语言编写的程序，它描述的是一种译码器电路的动作（我们将在第4章中再来论述各种译码器）。制造工艺的进步使得元件的尺寸不断缩小，与此同时，硬件工程师们发现，布设试验（或原型）电路板越来越困难，因为他们所要构建的电路已经变得太过复杂。硬件工程师使用传统手段能够轻松处理的大尺寸元器件的表现，无论如何不会与最终生产的集成电路上所使用的缩小的等价电路完全相同。采用VHDL设计电路时，还可以使用计算机进行模拟跟踪，并且能够向排列最终电路的自动规划工具包提供相应的输入。设计计算机的工程师的工作越来越面向语言：每件事都通过编写程序来指定，即使是对于硬件也如此。或许在不远的将来，即使在只接受过软件技术培训的情况下，计算机科学家也能够编写出可以用来生产新芯片的代码。

```
ENTITY decoder8 IS
   PORT  (sel: IN  std_logic_vector (2 DOWNTO 0); -- select i/p signals
          sig: out std_logic_vector (7 downto 0)); -- eight o/p signals
END decoder8;

ARCHITECTURE rtl OF decoder8 IS
BEGIN
   s <=  "0000_0001" WHEN (sel = X"0") ELSE
         "0000_0010" WHEN (sel = X"1") ELSE
         "0000_0100" WHEN (sel = X"2") ELSE
         "0000_1000" WHEN (sel = X"3") ELSE
         "0001_0000" WHEN (sel = X"4") ELSE
         "0010_0000" WHEN (sel = X"5") ELSE
         "0100_0000" WHEN (sel = X"6") ELSE
         "1000_0000";
END rtl;
```

图1-5　VHDL代码片段：译码器电路

过去，经过培训的电子工程师有向软件迁移的趋势，他们学习程序设计技能，并开始涉足系统程序设计。这种趋势现在逆转了吗？程序员和软件工程师所接受的培训都是如何处理大型系统和复杂的规格说明，也可以通过学习VHDL或Verilog参与硬件设计。这又是一个具体的例子，展现出硬件和软件如何通过工具和系统开发人员所需的技能揉合到了一起。

1.4　人人都应了解的系统管理问题

IBM PC及其后续兼容产品的流行意味着，现在更多的普通计算机用户在办公室和家中都不得不自力更生，担负起系统管理员的角色，执行修复、升级和软件安装等工作。这和早期的大型机和小型机——一般都由经过培训的技术人员团队来负责管理各项必须的例行工作——形成鲜明的对比。这么奢侈的事情可能已经成为过去。不管程度如何，掌握本课程讲述的内容，肯定会对从事上述这些活动有所裨益。在理解技术手册上艰涩难懂的叙述时，能够清楚地了解到底要做什么肯定会带来巨大的好处。这就如同阅读地图时，知道要去哪里肯定很重要。没有经验的用户常常被劝说购买很贵的硬件，以提高他们使用的系统的性能，而事实上，仅仅改变一下软件甚至是数据的组织方式，就有可能在不付出任何成本的情况下，使系统的性能有较大的提升。对于一些简单的问题，比如打印机不工作，甚至只需拥有一些CSA方面的基础知识，就能够诊断并解决问题。

1.5 语音、图像和数据：技术的趋同现象

除传统的商业数据处理以外，现代计算技术还涵盖十分广泛的领域。公众对计算技术的感知已经从燃气账单、发票打印及雇员的工资单替换为Internet、空中交通管制和移动电话。作为一些新的应用领域，语音通信、视频动画、出版、广播和音乐已经完全离不开计算机技术。电话通信与数据处理的分隔（许多公司通过分别设立通信经理和数据处理经理将之制度化）正在迅速消失。现代的数字电话交换机，也称为专用自动分支交换机（Private Automatic Branch Exchanges，PABX），如今可以看做是拥有某种专门的输入－输出能力，可以处理数字化语音信号流的计算机。

一些大型的电话交换系统已经使用Unix的特定版本作为其操作系统，因此，电信与数据处理之间的传统边界早就不那么明显了。如果大家知道数字电话交换设备将所有的语音信号转换成数字形式，然后按照处理任何二进制数据的方式对它们进行处理之后，对这种趋同（convergence）现象会更有感触。图1.6中给出的电话交换示意图，说明了控制计算机的核心作用。所有新的电话功能，回叫、预占、电话转接和电话会议等，都直接由交换软件来实现，而不需对硬件进行任何特别的改动。这绝对是营销上梦寐以求的东西！

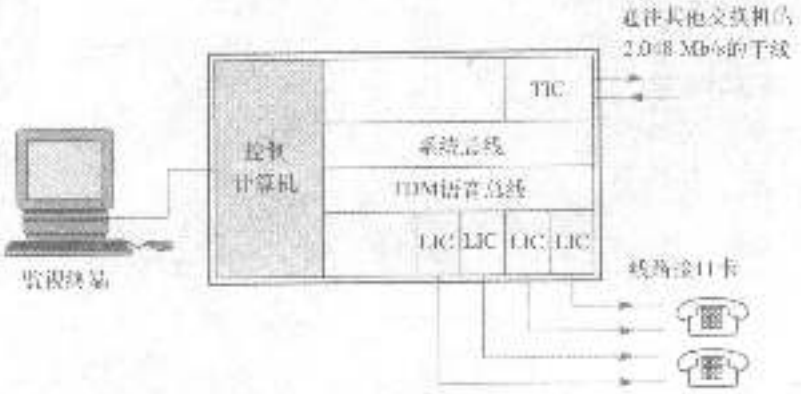


图1-6 电话交换示意图：嵌入式计算机的应用

Microsoft在Windows平台上提供了一套专为开发电话应用的程序员而准备的接口——TAPI（Telephony Application Programming Interfaces，电话应用系统编程接口），它能够与许多电信卡（可以通过多种渠道获得）一起工作。这些卡都遵守一套统一的接口标准，称做多厂商集成协议（Multi-Vendor Integration Protocol，MVIP），从而能够独立于PC总线交换语音数据流。计算机与电话的集成（Computer Telephony Integration，CTI）这一领域正在高速扩张。不同的机构组织都抓住机遇，利用这些硬件和软件组件构建自己的新系统，呼叫中心、电话应答服务和信息银行——所有这些都依赖于这项新技术。最近，电信运营商开始提供VOIP（Voice Over IP，即通过IP网络传输语音）服务，这项服务使用Internet来取代传统的电话网络。

我们日常生活中更熟悉的另一个声音和数据处理融合的例子是声霸卡（Sound Blaster），它广受PC游戏生产者的欢迎。多媒体（Multimedia）设备已成为近期硬件发展的主要受益者，使用DSP（Digital Signal Processor，数字信号处理器）单元，先启和分级DAC（Digital to Analogue Convertor，数字模拟转换器）可以重放高品质的声音和立体声音乐。我们还可以连接到某个网站，下载声音文件，然后由声霸卡重建声音，通过我们的立体声系统聆听音乐。

1.6 窗口界面（WIMP）

新千年中的计算机科学系学生可能只接触过“用户友好”的窗口界面——窗口、图标、菜单和指针（Windows，Icon，Menus and Pointers，WIMP），见图1-7。它们使用屏幕和绘图硬件，用矩形的窗口来显

示文本和图像，运行Windows的PC，装有X的Unix，以及Apple Macintosh都提供这类界面。然而，最初人们与计算机之间的对话却是完全基于文本的，依赖于命令行解释进程或通过外壳（shell）来解码那些通过ASCII键盘输入的“神秘”的指令，如图1-8所示。我们常常将这种情形称为运行“C>”、“DOS提示符”或“纯Unix”。现代的WIMP界面是微处理器革命的一个副产品，正是由于微处理器的革命性发展，才允许所有的用户都能够在桌面上拥有快速的图形屏幕，因而能够感受色彩和图像所带来的好处。



图1-7　用Windows资源管理器浏览目录

```
% cat .cshrc
umask 077
limit core 0
setenv TERM vt100
setenv PRINTER lw
set prompt = "`hostname` -> "
set history = 25
biff y
mesg n
alias tr99 (setenv DISPLAY 2799:0)
set path = (./usr/ucb /usr/bin/X11 /bin /usr/bin /usr/local
set path = ($path /etc /usr/etc /usr/lang /usr/local
            $home/bin)
```

图1-8　Unix命令行界面：显示.cshrc文件

很明显，Microsoft Windows操作系统已经全方位地渗透到我们的日常生活中，但遗憾的是，这种渗透是我在上班时路过附近的银行自动柜员机（Automatic Teller Machine，ATM）时感受到的。在那个屏幕上，是一段不常见但却很烦人的错误消息：

```
DLL initialization failure
C:\WINNT\System32\KERNEL32.DLL
The process is terminating abnormally
```

尽管WIMP界面取得了很大的成功，但是，相比于早期的窗口系统，老式的命令行（command line）方式依旧有许多优势。它们鼓励用户建立批处理（batch）或命令文件，存储在磁盘上，从而

避免重复键入相同的命令。早期的窗口界面不提供这种能力，但是微软在Windows NT中恢复了这一功能。当用户登录到Unix时，操作系统会自动运行用户的.login和.cshrc外壳初始化脚本，如图1-8所示。它们是小型的批处理文件，其中含有设置用户工作环境的命令。所有这些命令都可以直接在命令行输入，但存储在一个文件中要更容易、更安全。

用户需求以及技术的提高和改进，为硬件的发展提供了足够的活力和财务驱动力，使得硬件性能从1974年的1 MHz 8位的8080 CPU，提升到当前的1 GHz 32位奔腾处理器。在20年中，计算成本显著地下降，与此同时，处理能力和用户的期望却在上升。CPU能力的增加非常快就被新的计算要求耗尽，WIMP图形界面的引入抵消了16位处理器所提供的大部分优势，类似地，向多任务Windows系统的迁移，吸收了大量由32位处理器所提供的增强性能。有可能从未来的软硬件突破中受益的应用领域有：

- 实用的语音输入。
- 对能源资源的更好管理。
- 更丰富多彩的AI游戏。
- 图形娱乐。
- 视频通信。
- 互动教育。

最近，出现了一些通过现有的电话网络进行在线视频放送的商业计划。这就需要用到快速的压缩技术、庞大的数据库服务器，以及高速调制解调器。这又是一个硬件和软件紧密互动，提供新的产品和服务的实例。

1.7 因特网：连接所有的网络

由于Netscape Navigator（以及其他WWW浏览器）的广泛应用，公众的注意力已经转移到计算机、通信和软件上。它们使用户能够容易地访问到Internet上的海量数据。Web用户的数目快速增长，重现了当年电话拨号替代手工交换，从而使得能够连接的呼叫者达到前所未有的水平时所发生的变化。

新的应用处于不断的开发中，比如网络广播、Internet电话、在线购物、远程监视。新的企业不断涌现，他们发扬和促进了由于计算技术和通信技术创造性地融合而产生的服务。就如同铁路起源于蒸汽动力的发动机和运输工程的融合，通过使每个人都能够便宜地旅行从而变革了整个社会一样，Internet同样会通过降低对旅行的需求，尤其是上班和下班，从而改变我们的生活和工作习惯。

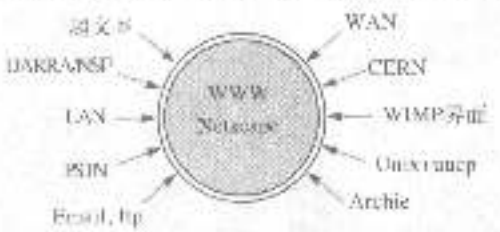


图1-9 影响万维网的各种因素

我们的生活已经开始受其影响，尽管你可能不欢迎，商家们已经开始利用这种新渠道发送个性化的宣传资料。万维网（World Wide Web，WWW）是现有功能的综合（参见图1-9），它们包括FTP（远程文件访问）、Archie（国际性的主题数据库）、超文本（基于关键词的文档间引用方案）、WIMP界面和网络访问能力（调制解调器或网卡）。位于瑞士日内瓦的CERN核研究所中，大型计算机与局域网（Local Area Network，LAN）连成一体，Tim Berners-Lee据此产生一种用更集成的方式来进行信息存储和访问的想法。后来，这种想法超越CERN的范围，成为了WWW，来自于全世界的站点共同连接到这个分布式的信息网中。我们将在第15章中更详细地论述WWW。或许，将来人们会认为，这是这个世纪（20世纪）最后十年中最杰出的技术成就。只要拥有PC和调制解调器，通过购买公众Internet服务提供商（Internet Service Provider，ISP），比如CompuServe、Demon或Virgin的服务，就能够使用本地的电话网络拨入它们提供的端口，很容易地连接到Internet。1998年，免费Internet服务（由Dixons的Freeserve和Tesco的TESCOnet发起）在英国的引入，大大促进了电子邮件和Web浏览器的家庭应用。然而，当时网上购物尚没有任何明显的迹象表明可能取得成功或者失败。

并非所有的商业机构都直接连接到Internet。出于安全方面的考虑，许多机构依旧倾向于仅在必要的时候接入到Internet服务提供商那里，或者直接使用调制解调器连接到客户的计算机。大学的员工和学生一般能够享受政府为鼓励高等教育领域中的研究协作和学术发展而提供的快速Internet访问。许多学校除了电话账单有可能会大幅增长，依旧未连接到Internet。

如图1-10所示，Internet由数以千计相互连接的局域网形成。每个局域网又会连接许多计算机，类似于国际公路系统可以被简单地看做是城市和城镇的互连一样。通过Internet，数百万的计算机得以交换电子邮件、共享数据文件及处理能力。每个局域网在布局和复杂度上都有所不同，图1-11给出了一个典型的配置，从中我们可以看出网络中有可能采用的各种技术。

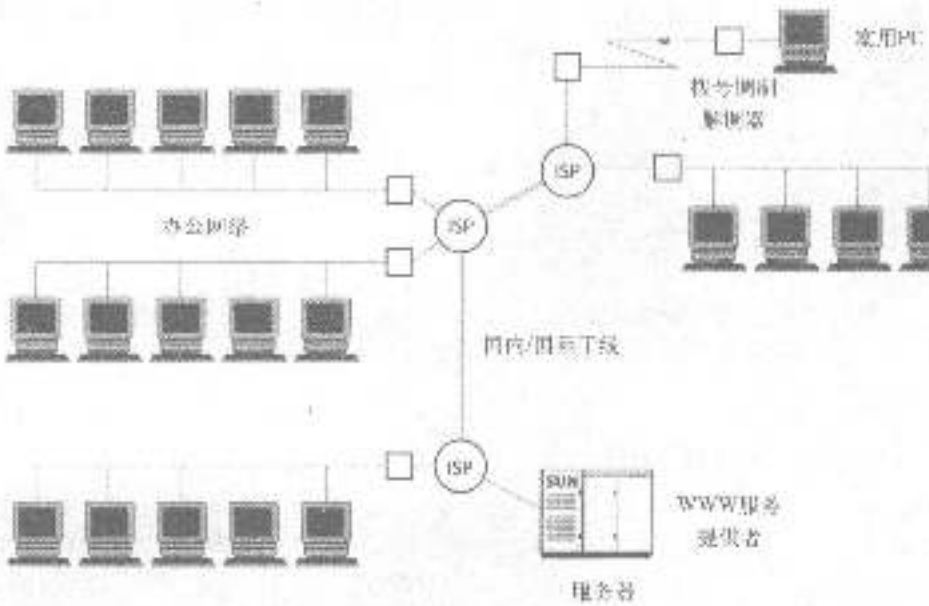


图1-10　将网络接入Internet

近年来最伟大的技术成就之一，就是Internet的建立。这是国家级国际层面，为了使数据能够跨越世界快速和无缝地流通，而进行的复杂协商和合作所造成的结果。Internet中的数据传输常常最初由局域网发起，然后通过ISP的网关传输到Internet上（见图1-11）。为了适合传输包（packet）的大小，消息必须进行拆分。每个数据包都含有重要的路由信息，当然还有需要传输的数据。

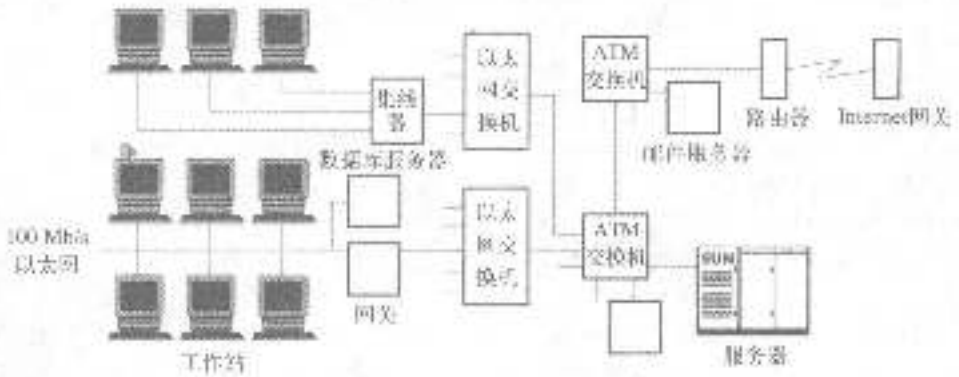


图1-11　典型的局域网

图1-12给出的经过简化的数据包基于以太网（Ethernet）标准，以太网能够以100 Mb/s的速度发

送数据。就像信件和包裹的前面必须写上目的地址，邮局工作人员才能知道要送往何处一样。每个以太网数据包都含有一个48位的目的地址，从而使数据能够被网络中正确的机器读取，而被其余的机器忽略。14.4节将更详细地讲解如何执行这种方案，但有几个细节需要提一下。我使用的Sun工作站的以太网地址是42.49.4F.00.20.54，尽管这个48位地址全球惟一，而且分配方案由硬件制造厂商负责安排，但是，提供一个基于它们的全世界范围的路由目录是不可能的。可以想像下面这种情况，如果我们必须接受在工厂中预先分配给电话机的制造代码作为家庭电话号码，那会怎样呢？因此，我们没有使用静态的以太网编号来完成消息路由，而是采用另一种可配置的数值系列。这些数值被称为IP（Internet Protocol，Internet协议）编号，我们必须为每个连接到Internet的计算机分配一个惟一的IP标识。例如，我的工作站被赋予2752186723这个数字，我们日常见到的并不是这种格式，它通常表示为以256为基的4组数字的形式：164.11.9.99，通过下面的公式，可以得出它们实际上是相同的值：

(((164×256)+11)×256+9)×256+99)

为了对用户更友好，我们还为这个IP编号分配一个惟一的名字，我的Sun工作站被称为"olvesun"，它是英国布里斯托尔北部的一个小乡村。

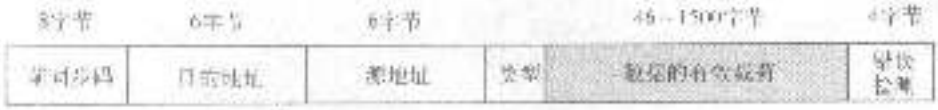


图1-12 LAN数据包内部结构中的地址字段

我们使用10个数字的电话号码已经超过50年的时间，也许我们也能够应付12位数字的IP代码！但是，在实际应用中，拥有容易识别的Internet访问名称，不管其是电子邮件或WWW URL，这种营销上的优点很快就被每个人接受。

1.8 使用PC：学习CSA的更多理由

如果读者对于CSA尚未完全了解，还没有认识到CSA的意义和功用，那么下面的说明或许会有所帮助。

如果你决定将家里的PC通过电话网络连接到办公室中的电脑或者连接到商业服务提供商，则需购买和插入调制解调卡，然后安装该卡并注册到PC的操作系统中，并载入恰当的设备驱动程序。其间，有可能需要了解系统中已经安装了哪些硬件，以及分配不同的IRQ值和端口号。随之而来的一些问题是：

- 你了解PC IRQ的角色和功能吗？两个设备能共享同一IRQ吗？
- 与主机建立拨入终端会话，需要设置各种线路传输参数：字的位数、奇偶校验、握手CTS/RTS、缓冲区大小和线路的速度，如果某项设置不正确的话，你能找出症节所在吗？
- 建议你建立一个PPP或SLIP会话，使得Netscape Navigator能够在PC的Windows环境下运行，这两种操作模式的真正区别是什么？为什么它们比基于字符的终端更好？
- 你肯定使用过WWW搜索引擎，比如Yahoo!或AltaVista，以找出某些有相关数据的网站。但你了解它们如何收集所需的信息吗？
- 为什么在载入Netscape Navigator之后，字处理程序运行得很慢？疯狂的硬盘活动是不是与之有关系？什么是交换空间？它是虚拟内存的替代物吗？
- 你可能会觉得有必要升级一下内存。但是，用更多的DIMM DRAM模组将PC的内存扩大到512 MB，可能需要慎重对待。如果你正在使用SIMM，为什么要同时购买两个部件呢？为什么要指定60ns的访问时间呢？更便宜的100ns SIMM是否同样能够和旧的200 MHz奔腾处理器一同工作呢？

- 你是否了解Windows操作系统如何“知道”鼠标正指向哪个图标？几个程序如何共享单个计算机，而且看起来像在同时运行？
- 你是否为声卡不能为多个应用程序所用而感到困惑和灰心丧气呢？
- 即插即用（Plug-and-Play）是如何工作的呢？

所有这些问题都属于CSA课程的内容，我们将在本书的课程中研究所有这些问题。在计算机系统领域，一个特别的挑战正在等待我们。这要归因于技术变革那令人难以置信的步伐。最初的PC在其20年的商业生涯中，见证了速度和内存大小数百倍的增长。考虑到这一点，任何计算领域的课程，都必须面对这一职业在技术和应用上不断发生巨大变革这一事实，将满足这个职业潜在的知识要求作为首要目标。通过研究大量的系统，按照基本的概念分析它们的动作，并考虑行业内的历史性趋势，我们就能够为未知但激动人心的将来做最好的准备。

你可能喜欢阅读一些关于早期电子计算机（20世纪50年代和60年代建造）的历史资料或视频资料。这是一个有待于考古学家去发掘的新领域，但是，软件工程师和系统开发人员也不能完全忽视已有的经验。如果在开发新计算机系统的过程中，某本书使你免于犯下严重的错误，就不会有人质疑这种投入是否值得。

1.9 小结

- 计算机和网络已经成为我们生活中不可或缺的组成部分。因此，我们应该学习并掌握它们。
- 在计算机内部，硬件（电子设备）和软件（程序）协同工作，完成处理工作。
- 硬件常常被看做是多层结构的系统中最下面的一层，而应用程序则在这个堆栈的最顶层。中间部分是操作系统。
- 硬件的演化正如摩尔定律所预测的那样，在尺寸、速度和成本上不断改善。
- 现在，用来开发软件和硬件的方法十分类似。
- 广播中心、电话交换和计算机单元电路中所使用的各项技术，都趋向于将所有的事物看做是需要收集、处理和发送的数据。
- Windows图形界面迅速地使更多的人能够接触并使用计算机（而不仅仅限于计算机学科毕业的少数专业技术人员）。这是一个硬件和软件紧密合作、共同发展的实例。
- Internet最初是几台运行Unix的相互连接的计算机。WWW的引入使它得到了长足的发展。它正在成为商业和社会变革的核心。
- 我们现在都是系统管理员，所以我们需要对硬件和软件之间的相互作用有更好的理解。

实习作业

我们推荐的实习作业包括：熟悉计算机设备、现有的打印设备、网络访问和操作系统。图书馆还可能提供在线目录和预订图书的功能。

首先要保证你能够收发电子邮件。然后，使用Web搜索引擎（AltaVista），搜索POP和IMAP电子邮件协议之间的差异。找出你可以使用其中哪一种。

看看能不能找到汇总介绍计算机发展历史的视频资料，或其他类似的档案资料。

开始着手建立一个个人书签文件，其中记录有用的Web站点的URL，以便使用Web浏览器。你可以将它以文本文件的形式复制到闪存或软盘上，供下次上网时使用。

练习

1. 对CSA课程的学习，如何帮助你购买新的PC？
2. 什么环境使得IBM PC如此成功？
3. 数字电话交换机是否能够区分出语音和数据？
4. 声霸卡是用来做什么的？

5. 窗口界面是否提供C> DOS外壳提供的全部功能?
6. 你使用什么路由设备连接到Internet?
7. Tim Berners-Lee是谁? CERN是什么?
8. 造一个句子，用到下面这些词：订购、20、月、花费、Internet、英镑。
9. 为什么Unix更适合用做Internet主机的操作系统?
   A. 以太网地址有多长? Internet地址有多长（IP编号）?
   B. 说出四个商业ISP的名字。
   C. 摩尔定律是什么? 谁是戈登·摩尔? 摩尔定律与程序员有关吗?
   D. 你对“技术趋同”怎么理解?
   E. 试着勾勒出一幅你所知道的局域网的布局。
   F. 一秒有多少毫秒（ms）?
   一秒有多少微秒（μs）?
   一秒有多少纳秒（ns）?
   1ns的千分之一称做什么?

课外读物

- 在图书馆中找出与计算科学相关的区域。了解长期、中期和短期借出之间的差异。检查相关的费用。
- 在图书馆的计算科学区，找出下列杂志《*Dr. Dobb's Journal*》、《*Byte*》和《*Personal Computer World*》；同样也要找出录像区。查看计算区、工程区和商业区的相关书籍。找出是否有在线计算机目录和帮助程序，有可能在图书馆以外也能够访问它们。
- 观看“BBC地平线”视频节目：The Dream Machine - Giant Brains。
- 看看附近的书店推荐哪些教材。选择几本书（可以从本书参考书目中挑选），熟悉它们的结构和内容清单，这样你就能够知道到哪里去找特定的主题。
- 随时注意一下公告牌，看有没有感兴趣的二手教科书出售。
- 使用Web浏览器（如Internet Explorer、Netscape、Mozilla、Opera、Konqueror）访问下述网站：
  - 提供许多学科引导性信息的好网站：
    http://webopedia.internet.com/
  - PC Guide也常常很有用：
    http://www.pcguide.com/
  - 基础性知识的在线教程：
    http://www.karbosguide.com
  - Wikipedia（维基百科）是Internet上由自愿者编写并维护的不断增长的信息源，任何人，只要有Web浏览器，就能够更改其中的绝大部分文章。它自2001年1月投入运行，有时其中的某些观点会引发一些论战。使用下面给出的URL，将最后的单词设为你想要查询的目标。试试“sausages”，阅读“Health Concerns”一节。
    http://en.wikipedia.org/wiki/sausages
  - 微处理器的背景知识：
    http://www.ibm.com/developerworks/library/pa-microhist.html
  - 构建自己的PC：
    http://www.buildyourown.org.uk
  - 现在及过去的重要的CPU和微处理器（这确实值得一读！）：
    http://www.sasktelwebsite.net/jbayko/cpu.html
  - Intel的新闻及产品：

http://www.developer.intel.com/

- 计算机的创造者及他们的发明创造：
  http://inventors.about.com/library/blcindex.htm
- Google出色的搜索引擎：
  http://www.google.com/
- 计算技术与电话技术日渐融合的征兆，参见：
  http://www.mvip.org/overview.htm
- 一些有关硬件设备的技术性汇总：
  http://www.basichardware.com
- 一些历史性的内容：
  http://www.codesandciphers.org.uk
- 如果想更深入地了解许多与计算相关的主题，参见：
  http://www.programmersheaven.com/

向他人推荐网站是一个棘手的问题，因为信息有可能会被移到了不同的位置，甚至撤消。但是，并不需要将这些信息归档，使得别人能够在晚些时候访问它们，就如同书籍和期刊一样。URL（Uniform Resource Locator）的任何更动都需要我们重新进行搜索。有时，我们能够使用搜索引擎（Search Engine），比如Google，快速地发现这些网站，因为我们现在拥有好的“关键词”。我们甚至可以通过已经掌握的信息猜出类似的URL。但是，如果得到类似于下面的错误消息：“Requested URL could not be retrieved”，不要立即放弃。这就如同在图书馆中查找特定的书籍一样。如果它不在书架上相应的位置，我们就到处看看，检查它是否被移动到了其他地方；或者试着使用目录找出它来；最后，还可以从他人那里寻求帮助。类似的文章可能会提供有效的替代，坚持就会获得回报！找到好的网站，要及时设置一个书签，这样以后就能够容易地重新返回这里。这要比写下这个URL容易得多。你可以将书签文件拷贝到闪存或软盘上，带着它以备下次访问。许多科目都和“计算机系统结构”一样，变动非常快，有时，Web是惟一能够被广泛访问的好的技术信息来源。在这个领域内，有几个广为人知的大学教材，我将会在每章结尾提示相关的、需要关注或能够提供帮助的章节。这些书籍的详细情况在参考文献中给出。

- Buchanan（1998）
- Hamacher等（2002）
- Heuring和Jordan（2004）
- Patterson和Hennessy（2004）
- Tanenbaum（2000）

这些网站可以通过本书的配套网站（http://www.pearsoned.co.uk/williams）访问。

第2章　冯·诺依曼体系结构的特征

计算机是由程序控制其动作的机器。程序不过是精心准备的指令清单。在这一点上，所有类型的数字计算机在基本原理上都是类似的。本章介绍读取-执行周期（fetch-execute cycle）的基本技术原理，数字计算机就是通过读取-执行周期来执行程序的。指令只有组织成一种专门的二进制格式，才能为计算机所理解。编译器的作用就是从以用户容易接受的方式表达的那些程序中产生这种二进制格式。数据同样也必须以二进制格式存储和操作。从用户的角度看，计算机的个性更多地由所运行的操作系统软件，而非底层的硬件而定。现今的软件系统，常常会以客户机-服务器的结构分布在网络中的几台计算机上。

2.1　以2为基：二进制的优点

当代数字计算机的基本结构，包括软件和硬件，要归功于约翰·冯·诺依曼（John von Neumann）。有些历史学家还坚持强调查尔斯·巴贝奇（Charles Babbage）也在1830年为此做出了相当的贡献。毫无疑问的是，最终还是冯·诺依曼的清晰描述和说明，在20世纪40年代晚期吸引了工程师和科学家们的注意力。其基本工作流程（见图2-1）为输入-处理-输出，这是特殊的定制程序所要控制的全部内容。



图2-1　程序控制的全部内容

早在20世纪30年代晚期，康拉德·楚泽（Konrad Zuse）就在德国建造了第一台能够工作的电子数字计算机。他提议将二进制（以2为基）用于数字的表示。对于使用电子开关电路实现来讲，这要比常规的以10为基的方式更合适（参见本章结尾的注释）。然而，希特勒的军事野心以及乐观的预期，限制了他有限的技术视野。同时，为了火箭助推的飞行器项目，这个项目被取消了。故而，德国在这方面的工作没有取得任何成果。在英国，战时进行的另一项重要的计算方面的开发工作是Colossus。这个电子—机械相结合的计算机器用来帮助对加密传输的无线电信号进行解密。但它是专为特殊的任务进行构建和设计的，没有内部存储设施来保存指令。由于军事解密活动的机密性，在1975年之前，很少有这方面的信息为外界所知。在美国，爱荷华州大学的John Atanasoff和贝尔实验室的George Stibitz也在进行开拓性的研究工作。这些项目都展示出远远超前于他们所在时代的技术性革新，Atanasoff用到了电容性的存储单元——当代DRAM芯片的前身，使用二进制对指令和数据进行编码的应用，在1950年得到清晰的实现。

2.2　程序控制存储：通用机器

冯·诺依曼最初的想法是，由可执行程序控制通用机器的动作。此时，程序是一系列用来指导任务的指令，乐谱和编织图案就是绝好的例子，它们由人来解读，但存储在外部；而数字计算机使用内部存储器来存程序和数据。乍看起来，这样可能会导致灾难性的后果，因为计算机无法轻易地区分数据和指令。指令和数据都用二进制编码来表示。在不同情况下，一个16位的指令代码可以表示一个数字或两个字符。如果发生某些异常（比如语法），计算机有可能会去执行数据，或对指令进行加减运算。它不可能立即意识到这种混乱。大部分情况下，计算机会很快崩溃，因为其后的一系列动作是随机的（如果合法的话）。

只有通过硬件和软件的紧密交互，计算机的各种复杂活动才能够完成。软件存储在存储器中（见图2-2），**中央处理单元**（Central Processing Unit，CPU）实际上就有执行程序指令的硬件。

大部分计算机都采用通用的硬件，依赖于不同的程序（而非硬件）来产生所要求的特定动作。为了在存储器中存储程序，以及以计算机的CPU可以理解的方式提交程序，指令和数据必须按照某种特定的编码形式进行组织。现代计算机在发展上的巨大进步，是实现了程序可以存储在能够快速访问的存储器中，而非最初的纸带上。

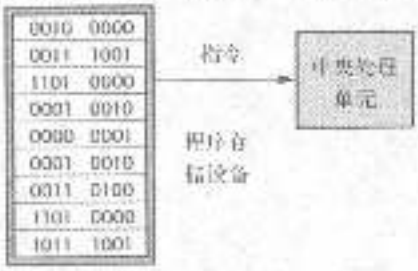


图2-2　程序的存储

表2-1　各类数字计算机及其典型应用

计算机	应用
智能卡	电话卡或信用卡
微控制器	洗衣机控制器
游戏机	互动式娱乐
家用电脑	Web信息浏览
工作站	设计电路板的布局
办公服务器	局域网内文件的集中网络
大型机	企业数据库
超级计算机	飞行模拟研究

因此，我们可以这样认为，之所以**主存储器**（main memory）是必需的，是因为磁盘和磁带存储介质太过于缓慢。后面，在第12章，我们将会更为详细地介绍计算机的各种存储器所构成的复杂的体系结构。因为现实和经济上的原因，PC、工作站或大型计算机中都没有采用单片式的主存储器。现今，或许只能在洗衣机上才能看到这么简单直接的模式，当然，前提是洗衣机必须要有电子微控制器，而非简单的发条装置。

现今，计算机的类型多种多样（见表2-1），我们在日常生活中常常会遇到其中的许多类型，它们的处理能力和制造成本可能会相差10^7之多。但是，即使在最极端的例子中，我们依旧可以找出这些计算机的十分相似之处，这一点值得我们注意。如我们所见，冯·诺依曼最初的设想之一是单一模型，即通用的计算机，通过各种各样的程序库完成所有的功能，这与当代的商业观点并不吻合。

但在冯·诺依曼的描述中，计算机含有可替换的程序，它保存在统一的存储器中，控制着计算机的运作，这一描述保持了50年。一种变化细微的变体，称为“哈佛结构”（Harvard architecture），最近发展起来。它将数据和程序分隔开来，需要用到不同的存储器和访问总线。其意图是增加传输速率，提高吞吐量。在第21章中论述RISC CPU时，我们将会清晰地看到这种改动的必要性。另外，哈佛设计的另一个应用已扩展到数字信号处理（Digital Signal Processing，DSP）芯片中，因为这些芯片中的程序经常需要以极快的速率处理大量的数据。

2.3　指令代码：控制机器动作的指令系统

计算机有一套小型的固定指令系统，称为**机器指令集**（machine instruction set）。每个制造商，比如IBM、Intel或Sun，设计制造的计算机中所使用的CPU都有自己的一套本地（native）指令集。这个指令集一般包括100到200条指令。与ASCII和Unicode数据编码已经被普遍采用形成鲜明对比的是，行业中尚不存在公认的标准指令集。

不同的CPU支持的指令数存在很大的差异，从十几个到数百个不等。因此，人们热心于开发标准化的**高级语言**（High Level Language，HLL）。许多人更喜欢使用高级语言进行编程，比如BASIC或C，因为使用这些语言编程更容易。但是，HLL指令代码依旧必须转换成低级的机器代码，才能使计算机运行该程序。这种转换一般会将每个HLL指令扩展成5～10条机器代码。图2-3中给出一行C语言代码的例子。对于那些先驱者而言，在测试第一台数字计算机时所面临的一个重大问题就是，如何编写程序，以及之后如何将编写好的程序传入到机器的存储器。实际上，当时的程序是用二进制手工编写的，使用一排触发开关（toggle switch）手动输入到存储器中。这种折磨人的费力的过程，突出了人类语言表达思想和意图的方式与计算机指令表示数据处理活动的方式之间的巨大差异。这种差异被称为是**语义鸿沟**（semantic gap）（见图2-3）。语言学的学者们可能会反对这种说法，他们或许更喜欢说“词汇—语法的割裂”。但对于人类来说，使用HLL进行编码的优势是毋庸置疑的。

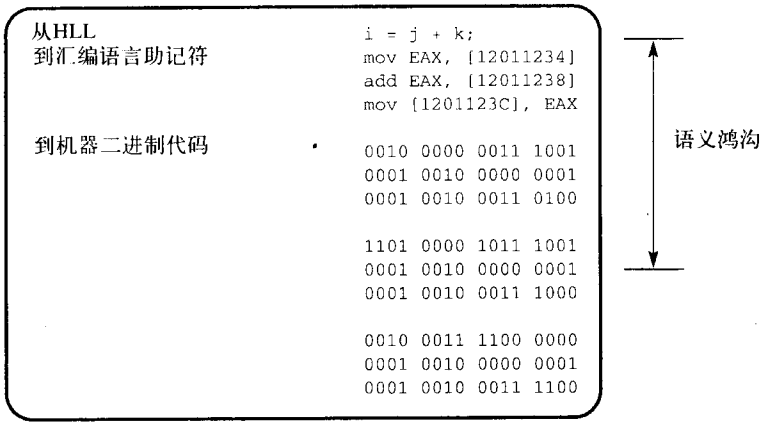


图2-3 HLL、汇编语言和机器代码的对照

如表2-2所示，所有的机器指令都可以归结成很少的几组类别，牢记这一点，对以后的学习将会很有帮助。

表2-2 机器指令类别

1. 数据传输和操纵
2. 输入-输出
3. 程序控制的转移
4. 机器控制

我们将在后面的第7章到第10章再做进一步的论述，现在，只需对这些术语有大概的了解即可。数据传输和操纵指令组包括诸如下面的指令：MOV、ADD、MUL、AND、OR和循环/移位等。这些是最常用到的机器指令。输入-输出指令负责与外部世界交换数据。奇怪的是，某些类型的CPU中并不存在这组指令，在这类CPU中，所有通往外部世界的端口，都必须按照与内部存储器一样的方式寻址和访问。但是，Intel x86和奔腾处理器都有单独的IO空间，并提供专门的IO指令：IN和OUT，来执行输入输出操作。第三组，程序控制转移指令组包括分支、跳转和子例程调用指令：BRA、JMP、BSR和RTS。读者可能已经知道，学院派回避使用GOTO指令，因为GOTO指令的使用被看做是坏编程习惯的根源之一。可怜的BRA和JMP，在他们看来都是GOTO的难兄难弟！最后一组指令是所谓的“危险”指令组，它们被归入机器控制类别，系统常常禁止普通用户使用它们。它们能够停止进程，更换中断掩码或重置硬件。您或许已经受够了机器代码的折磨，现在正准备仔细考虑一下使用HLL所带来的明显优势了。

2.4 转换：编译器和汇编器

令人感到欣慰的是，大部分程序现在是使用HLL（如C、C++、Java或BASIC）编写的。由专门的程序，即**编译器**（compiler），将HLL指令转换成能够直接在计算机上执行的二进制机器代码。编译器已经成为计算机系统至关重要的部分，它们常常决定着程序员利用硬件的效率。在HLL编译器被广泛采用之前，程序员们使用**汇编器**（assembler）。它同样拥有不需直接用二进制机器代码编写程序的优点。在汇编语言中，每个机器代码指令被赋予一个**助记符**（mnemonic），如ADD、SUB或MOVE。因而，编写的程序实际上是一系列的助记符，这些助记符能够容易地被转换成它们所对应的二进制机器代码。由于大部分情况下由汇编器所做的转换都是简单的一对一操作，因此可以编写程序执行这种功能（这种转换很容易完成）。尽管我们偶尔还是能够看到汇编器的应用，但大多数情况下，HLL编译器已经占据主导地位。

HLL指令到机器代码的转换，现在由图2-4所概括的过程自动完成。这项转换既可以由**解释器**（interpreter）来实时地完成，也可以由**编译器**来预先完成。BASIC和Java都是由解释器开始，之后推出编译器。Pascal和C几乎总是必须进行编译，但解释器也是存在的。在现今的处理器越来越快，以及人们越来越倾向于将大规模的程序拆分成半自治的各个组成部分的情况下，编译-链接需要时间来完成这一缺点，已经基本上没有什么影响。解释器的主要优点已经由减少开发延迟，变成提供跨多种不同类型计算机的、安全的、统一的执行环境。

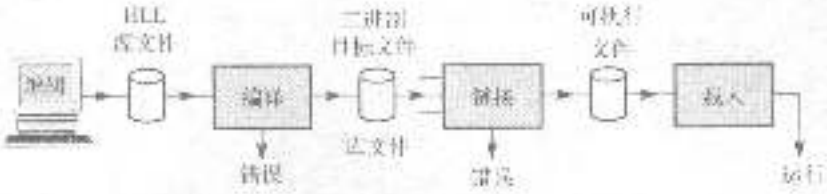


图2-4　使用编译器生成执行代码的过程

2.5　链接：将程序组合到一起

由于程序的规模越来越大，以至于需要许多年的工作才能开发出来，因此，现在人们常常在项目开始的时候，就将它们划分成几个单独的部分，或称为**模块**（modules）。为了生成一个能够运行的程序，每个模块都必须设计、编码并编译，然后由**链接器**（linker）将产生的各个部分连接到一起。该连接过程并非简单的将各个部件粘合起来，还涉及到解析**外部引用**（external references），在将大型的程序分成多个模块的时候，常常会出现某个模块内的代码需要引用其他模块中的数据和子例程的情况（参见图2-5），编译器每次只转换一个模块。因此，当编译器试图为这些符号引用确定一个数值等价物时，它会发现在当前的模块中找不到相应的值。未解决的符号称为外部引用，并保持为符号（而非数值），直到链接器工作，将引用替换为其他模块中的数据项（item）为止。

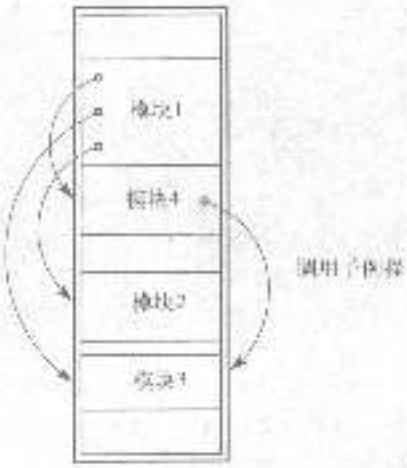


图2-5　含有外部引用的模块

链接阶段常常会被程序员忽略，因为编译器可能会自动地在没有用户参与的情况下，将工作传递给链接器。以Unix下编译和链接代码的cc命令为例，我们很容易将其理解为"compile C"，而忽略掉"and link, too, if possible"。对库子例程的使用，一定会导致模块间的外部引用，需要链接器来解决。

库文件，如Unix中/lib和/usr/lib目录下的那些文件，保存了许多函数转换后的目标代码，但是，仅当把它们链接到我们的代码中时才能使用。通过设置恰当的链接器选项，我们可以请求保留库函数的私有（静态）副本，或者同意与其他程序共享同一副本（动态）。Microsoft平台下的程序员常常将这样的动态库模块（将在8.9节中更进一步地论述）称为DLL（Dynamic Linked Library，动态链接库）。在购买编译器时，交易的重要部分之一就是获得精心设计组织的库例程，用到自己的代码中。

将程序所有的目标模块链接到一起，并指定相关函数所在的函数库，所有外部引用都已解决之后，程序就已就绪，可以载入到指定的存储器地址执行。但是，根据给予链接器的标志和参数值的不同，程序有可能依旧需要做进一步的地址调整才可以执行。如果想要在支持多任务、虚拟内存的系统（且提供进程保护和动态库绑定）上运行程序，如Windows或Unix，还需要在代码载入后创建供内存管理单元使用的系统表，以及安装合适的段描述符。更多关于这方面的内容，在本书后面的第17章论述。

2.6　解释器：执行高级命令

解释器，如同BASIC和Java中使用的那样，提供另一种运行HLL程序的方式。使用解释器时，不再需要将所有的HLL指令都转换成机器代码并创建可执行文件，而是由**解释器**（interpreter）每次读入一条HLL指令，然后使用自己的例程库执行指令的命令。采用这种方式时，可执行代码不是由源代码生成，而是包含在解释器内。解释器以HLL源代码为输入数据，对其进行分析，并执行它要

求的处理工作。采用解释模式处理HLL程序的优点是快速启动，并且显然去除了编译和链接的复杂性。遗憾的是，解释器的缺点是它们动作较慢，编译完成后的程序总是比解释执行的软件要快。

在决定应该采取什么动作之前，解释器一般会将输入的指令转换成中间形式（由多种标记符构成）。在图2-6中，每条HLL指令都从源文件中读出，然后检查错误，分析关键字和语法结构。之后，简化后的标记符被传送给解码器，解码器负责选择恰当的执行例程。我们有时将解释器称做一种"虚拟机"（virtual machine），因为它的行为从某种角度上来讲，类似于计算机硬件——它每次读取一条指令并执行它们。虚拟机解释器是缩小2.3节中提到的语义鸿沟的一种方法，它能够提高执行环境的级别，使之接近问题所在。Java对编译和解释做了一个很有意思的安排，如图2-7所示。

Java源文件经过编译器的编译，产生的目标文件既可以作为传统的解释性语言程序运行，也可以载入到Web浏览器中，比如Netscape Navigator或Internet Explorer，以便为Web页面提供额外的功能。

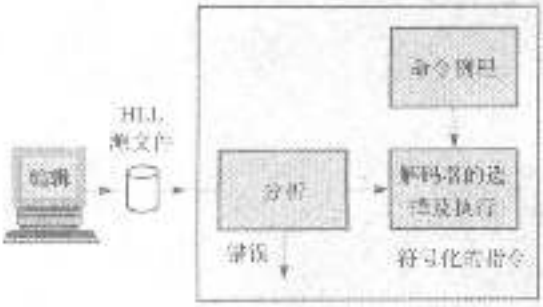


图2-6 使用解释器转换和执行程序

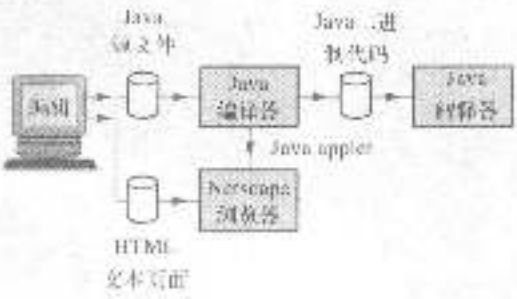


图2-7 Java使用编译器和解释器

2.7 代码共享和重用：不要总是从头做起

在开发新系统时，重用已有的经过验证的软件，能够节省大量的时间和金钱，人们早就认识到这一点。但是，它依旧是一只难以捉摸的圣杯，时至今日，我们尚未完全拥有。软件开发方面的每种新方式都声称获得了突破，完成了"可重用软件组件"的目标。经过长时间认真的考虑，我认为就重用而言，判断是否成功的准则应该取决于软件行业的分销商们，比如英国的RS Components和美国的Avnet，看看他们是否引入了专门的软件模块目录。这显然还没有发生！

达成高效代码"重用"（或代码共享）的许多提案，可以映射到图2-8中所示的软件开发流程上。在汇编语言程序设计的最早期，就实现了源代码级的子例程和宏库。意图是取出库例程中的副本，将它们加入到新的代码中，然后整体运行转换。许多C程序中常见的#include头文件，就是使用这种方式。源代码流通起来以后，问题很快浮现：谁拥有该代码呢，它被修改成多少份副本呢，谁应该维护它呢？

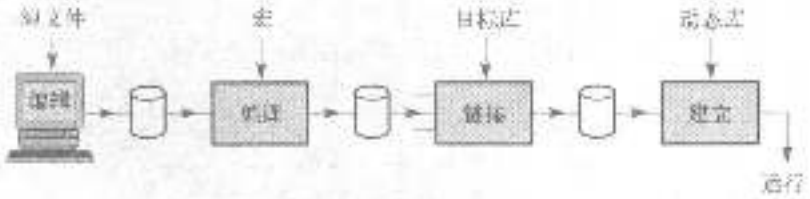


图2-8 代码如何共享

为了解决这些问题，预先完成转换、可重定位的二进制库被引入进来，它们可以不加固读或更改地链接到新的程序中。这样做的意图是，实现类似于电子硬件的软件组件市场。尽管链接到可重定位目标库的技术已经成熟，并且对于现今所有的软件开发，这依旧是最基本的，但它需要用到技

术性的开发工具，并且需要对这种方法有较好的理解。明显的缺点是，每个程序都需要有子例程的私有副本，在多任务系统中，会浪费宝贵的存储器空间及交换时间。

Microsoft推广**动态链接**（Dynamic Linking）的方法。它允许用户载入的程序使用已经载入到内存的“公共”例程。驻留内存的库在构建阶段完成映射，通过内存管理系统控制对代码的访问，并避免生成多份代码副本。透过Microsoft ActiveX标准的成功，我们或许已经看到了**软件组件**市场的真正春天！

2.8　数据编码：数值和字符

早在20世纪30、40年代，从早期的试验阶段起，使用二进制表达计算机内存储的数字的优点，就已经得到广泛的认同。这要归结于计算机所采用的二态、开/关技术。因此，在计算机可以对数字进行任何算术运算甚至存储之前，都必须将数字从以10为基（十进制）转换成以2为基（二进制）。或许有些出乎人们的意料，这项转换可以由计算机自己使用转换例程来完成。有时，这就像听的是法语，但却用英语思考。如同十进制数一样，二进制数，只使用有限数目的符号，通过使用“位置权重”（positional significance）来扩展表达的范围。因此，“1”可以表示1，2，4，8，16，32，64，或128……这要依它在数字中的位置而定。

4096	2048	1024	512	256	128	64	32	16	8	4	2	1	权重
0	1	0	0	1	0	1	0	1	1	1	0	1	

从二进制格式到等价的十进制数的转换，通过用权重值乘以对应的数字（4096×0，2048×1等），然后将结果累加完成。由于实现二进制乘法极为容易，因此，我们只写出结果数字：

2048 + 256 + 64 + 16 + 8 + 4 + 1 = 2397

从十进制数到等价的二进制数的转换一般并不是用同样的方式来完成，最明显的原因是由于它更困难。

1111101000	0001100100	0000001010	0000000001	权重
2	3	9	7	

0010 × 1111101000 + 0011 × 0001100100 +
1001 × 0000001010 + 0111 × 0000000001 = 100101011101

计算机可以很容易地完成这类求和运算。对用户更友好的将十进制转换成二进制的方法，是重复用2去除十进制数，每次都写下余数，不是0就是1，从右至左。可以从简单的数开始尝试这种方法，如图2-9所示。

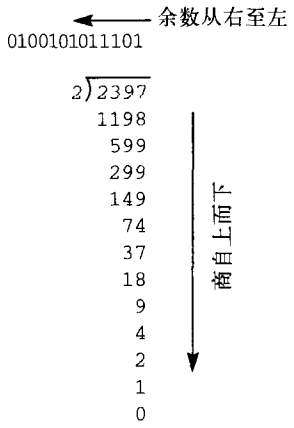


图2-9　通过不断做除法进行的十进制到二进制的转换

本书稍后，还会出现**十六进制**（hex）格式的数字，尤其是地址。这种数字基只是用来简化数字的表示，使人们能够容易地阅读较长的二进制数。我们一般不会用到十六进制的算术运算。在表2-3中可以看到，十六进制数的序列从9之后，使用字母A到F。在本书中，可以简单地将一个十六进制数看做是四个二进制位的一种表示方法。

人们频繁地使用键盘向计算机输入数据。现在，键盘上的按键可能超过100个，加上SHIFT和CTRL修饰键的使用，我们需要超过256个标识代码。因此，键的代码从8个二进制位扩展到16位。同时，以类似的方式，曾经精心构建的**8位ASCII**代码也因为对现代的基于Windows的字处理软件的显示需要太过局限而被放弃。同时，在由Microsoft这样的公

司所建立起来的全球软件市场中，需要处理国际字符集的问题。所有这些努力的结果，就是**16位Unicode**字符代码集被引入，7位的ASCII是它的一个子集。

由于采用16位代码，Unicode的字符空间中可以包含65 536个字符。这个空间被划分成256个块，每个块中可以包含最多256个字符，所包含的语言包括：阿拉伯语、亚美尼亚语、孟加拉语、梵文、希腊语、古吉拉特语、果鲁穆奇语、希伯来语、奥里雅语、俄语、泰米尔语，以及众多其他语言。如果希望查看全部的语言，可以查看http://www.unicode.org/提供的表格。

表2-3 十六进制数字对应的二进制数

0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

对于许多应用程序而言，基本ASCII字符集（见表2-4）的8位扩展依旧绰绰有余。但是ASCII助记符仍然显得有些神秘。参见表2-4，请先查出BEL（代码编为07）。当二进制模式0000111到达终端时，铃声（或者嘟嘟声）就应该响起，但屏幕上不会打印出任何东西。

表2-4 ASCII字符码表

位 4321	十六进制	位765 十进制	000	001	010	011	100	101	110	111
			0	16	32	48	64	80	96	112
			0	10	20	30	40	50	60	70
0000	0	0	NUL	DLE	SP	0	@	P	`	p
0001	1	1	SOH	DC1	!	1	A	Q	a	q
0010	2	2	STX	DC2	"	2	B	R	b	r
0011	3	3	ETX	DC3	#	3	C	S	c	s
0100	4	4	EOT	DC4	$	4	D	T	d	t
0101	5	5	ENQ	NAK	%	5	E	U	e	u
0110	6	6	ACK	SYN	&	6	F	V	f	v
0111	7	7	BEL	ETB	'	7	G	W	g	w
1000	8	8	BS	CAN	(	8	H	X	h	x
1001	9	9	TAB	EM	)	9	I	Y	i	y
1010	10	A	LF	SUB	*	:	J	Z	j	z
1011	11	B	VT	ESC	+	;	K	[	k	{
1100	12	C	FF	FS	,	<	L	\	l	|
1101	13	D	CR	GS	−	=	M	]	m	}
1110	14	E	SO	HOME	.	>	N	^	n	~
1111	15	F	SI	NL	/	?	O	−	o	DEL

NUL	Null（空）	DLE	Data Link Escape（数据通信换码）
SOH	Start of Heading（标题开始）	DC1	Device Control 1, ^Q（设备控制1, ^Q）
STX	Start of Text（文本开始）	DC2	Device Control 2, ^R（设备控制2, ^R）
ETX	End of Text（文本结束）	DC3	Device Control 3, ^S（设备控制3, ^S）
EOT	End of Transmission（传送结束）	DC4	Device Control 4, ^T（设备控制4, ^T）
ENQ	Enquiry（询问）	NAK	Negative Acknowledge（拒绝应答）
ACK	Acknowledge（应答）	SYN	Synchronization character（同步字符）
BEL	Bell（响铃）	ETB	End of Transmitted Block（传输块结束）
BS	Back Space（退格）	CAN	Cancel（取消）
HT	Horizontal Tab（水平制表符）	EM	End of Medium（介质结束）
LF	Line Feed（换行）	SUB	Substitute（替换）
VT	Vertical Tab（垂直制表符）	ESC	Escape（退出）
FF	Form Feed（换页）	FS	File Separator（文件分隔符）
CR	Carriage Return（回车）	GS	Group Separator（组分隔符）
SO	Shift Out（移出）	RS	Record Separator（记录分隔符）
SI	Shift In（移入）	US	Unit Separator（单元分隔符）
SP	Space（空格）	DEL	Delete（删除）

请试着运行一下图2-10给出的C程序，它向屏幕发送二进制0000111。可以试着插入个for(;;)循环来打扰一下旁边的邻居。

任何能够支持算术运算的数值表示法，都必须处理整数和实数值、正数和负数的问题。**整数**是完整的数字（-1，47，747），没有小数部分，而**实数**则扩展到小数点以后的部分（59.5，0.101，-2.303）。数值的表示也已经渐渐形成标准，就如同字符代码获得国际公认一样。整数一般32位长。以**二进制补码**表示和操作负整数的方法，由于硬件制造商如Intel（它所提供的处理器中内置了这种方式）的强制推行而最终得以推广。类似地，浮点数的**IEEE 754**标准，最初由浮点处理器的制造商引入，他们也造就了32和64位格式事实上的行业标准。我将在第5章中再次提及这些内容。

```
#include <stdio.h>

void main() {
     putchar (7);
}
```

图2-10　让终端响铃

在函数或程序的顶部声明变量时，就是在告诉编译器为保存该变量预留恰当大小的存储空间。请考虑图2-11中的C代码。

变量letter的大小为1个字节，存储ASCII字符代码。count大小为2个字节，类型为short int。uk_population作为4字节正整数存储，world_population也是4个字节长。实数变量body_weight、building_weight和world_weight，分别是4、8和16个字节长。知道变量的物理大小之后，可以帮助我们解释某些类型的问题，甚至是错误——当它们出现时。将C程序从一台机器移植到硬件不同的机器上时，int变量的宽度常常是许多莫名其妙的错误的起因。不过，随着商业上可以选择的CPU已经合理化到少数几个，它们均采用32位宽作为标准，这种问题已经不存在了。新的Intel 64位安腾处理器可能会使这个问题再度复活。

```
char letter;

short count;
unsigned int uk_population;
long world_population;

float body_weight;
double building_weight;
long double world_weight;
```

图2-11　HLL程序中的变量声明

2.9　操作系统：Unix和Windows

尽管有时候用户会注意到硬件上的差异，但是，真正赋予计算机“个性”的，还是**操作系统**（Operating System）。从一开始，用户必须熟悉的就是操作系统的怪脾气。因为当计算机硬件到达用户手中时，往往磁盘上已经预装了操作系统，许多用户难以认识到操作系统不过是一个程序，而非硬件部件。尽管现在的读者可能只知道少数几个名字，但是，在过去的40年中，所开发出来的操作系统的确为数不少。我能够立即想起来的都列在表2-5中。但这并不是全部。

表2-5　一些人们熟悉的操作系统

AIX	OS/2	CDOS	Pick
CICS	PRIMOS	CMS	PSTOS
CP/M	RSX/11	MS-DOS	RTL/11
George	TDS	IDRIS	THE
ISIS	Unix	LYNXOS	Ultrix
MINIX	VERSADOS	MOP	VM
VMS	MVS	Windows NT和95	BeOS
Multics	XENIX	OS-9	Linux

某些值得尊敬的创造可能只有您的祖父母才能记得，而且他们恰好在20世纪60年代和70年代从事与计算机相关的工作。但令人吃惊的是，技术核心并没有变动。只有窗口界面的引入，才使现代操作系统有别于许多开拓性的前辈。一般地，程序员有两种方式可以访问操作系统的功能：命令行和系统调用。

对于Unix，操作系统的功能常常由表2-6列出的某种方法来调用。常常出现在C语言程序设计课程中的一个例子是键盘输入。如果你在编写一个基于字符界面的菜单程序，你可能希望关闭键盘输入缓冲区，直接从键盘接受“Y/N”输入。正常情况下，你必须在每次敲击Y/N之后敲ENTER键，但这样做会产生一个问题：CR字符将会保持在缓冲区内，直到下次读取键盘为止。在测试代码时，这可能会带来一些混乱！Unix实用程序stty（见图2-12）采用的解决办法是，在等待按键时关闭字符缓冲并禁止编辑输入。

表2-6 访问操作系统功能的方法

1. 命令行解释器（CLI），外壳脚本或在桌面上选择
2. 用户程序中的函数调用（API）

```
rob [50]: stty -icanon min 1 time 0; menu_prog
  Are you ready to proceed? (Y/N)
```

图2-12 设置“原始”输入的Unix命令行指令

```
#include <errno.h>
#include <stdio.h>
#include <sys/termios.h>
#include <unistd.h>
#define TIMEOUT -1

extern int errno;
int sys_nerr;
extern char * sys_errlist[];

void setterm(void) {
  struct termios tty;
  int status;
    status = ioctl(0,TCGETS, &tty);
      tty.c_iflag &= ~ICANON;
      tty.c_cc[VTIME] = 0;
      tty.c_cc[VMIN] = 1;
    status = ioctl(0,TCSETS, &tty);
    if ( status == -1 ) {
        printf("ioctl error \n");
        perror(sys_errlist[errno]);
        exit();
    }
}
```

图2-13 设置“原始”输入的Unix系统调用

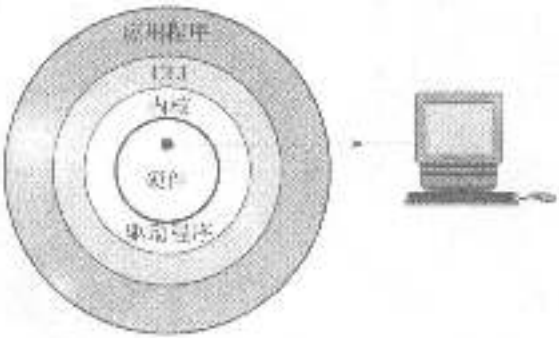


图2-14 包围硬件的软件层

第二种访问操作系统资源的方式，如表2-6中所述，是直接调用库函数，如图2-13中的C代码片段所示。之后，这项请求通过库代码传递给操作系统，从而，负责对键盘的输入进行处理的程序能够如此用户请求的更改。如果要达到和前面提到过的菜单程序相同的功能，现在需要插入Unix ioctl()系统调用，以告诉操作系统接受“原始”输入。

可以说，操作系统的主要目的是使计算机资源，即硬件和软件，更容易为用户所用。软件的多层排列常常被描绘成像洋葱一样的同心圆环（见图2-14），表明硬件的复杂性已经或最大可能地对用户隐藏，敏感的核心硬件已经被保护起来，免于受到无知或恶意用户的伤害。软件的最内层直接与硬件打交道的部分，是设备驱动程序、内存分配程序及进程调度。用户通过CLI或桌面软件访问计算机。

从前，为了给操作系统一个命令，曾经必须使用工作控制或命令语言（Job Control or Command Language，JCL）。Unix提供外壳脚本（Shell Script）语言，VAX有DCL，甚至MS-DOS也提供好几个命令。随着窗口界面的引入，这项活动变得越来越用户友好，原始的命令被对话框和下拉式菜单所隐藏。但底层的功能依旧是相同的。用箭头指向小幅的图画（图标）看起来要比摩涅的助记符，比如dir或ls，更容易理解和记忆。但是，为了节省时间、减少错误以及为将来提供文档性的证据，对认真的专业人员来说，将一系列指令存储成一个“命令文件”或“脚本”的能力是必不可少的。由于早期的WIMP界面并没有提供这种能力，因此这被认为是一项重大的不足。Unix提供几种命令行解释器，或称外壳，并且鼓励用户选择他们自己所喜欢的方式。表2-7列出了一些最常见的选择。

表2-7　各种Unix外壳（命令行解释器）

sh	最初的Bourne外壳，管理员依旧常用它来执行脚本
csh	C外壳提供要类似于C的语法，较适合于交互式人机会话
tcsh	Tenex外壳，表面是最常用的交互式外壳，类emacs的按键
ksh	Korn外壳，最常安装在Hewlett-Packard工作站上
bash	同样是Bourne外壳，结合其他外壳的特点而新写成，为自由软件

尽管得等到第17、18章我们才会详细介绍操作系统的内容，但在此还是要提醒读者，要了解操作系统的目的就是使得计算机的资源，即硬件和软件，更好地为用户所用。当计算机由许多任务或用户共享时，它的角色还会包括当争用发生时进行仲裁和决定，同样还会提供系统级的实用工具程序。计算机系统的整体效能往往并不是受CPU的处理能力所支配，而是由负责管理硬件的操作系统例程的精巧程度而定。

2.10　客户机服务器计算：网络时代的方式

自冯·诺依曼最初的蓝图发布后，众多的技术进步对他最初的想像进行了很大的变革。最深远的变革或许就是跨计算机网络（computer networks）的快速信息交换的引入。现今，几乎所有的商务PC、工作站和大型机都通过局域网（Local Area Network，LAN）互相连接起来。对网络设施的访问同样一般由操作系统提供。Novell为PC开发了远程文件服务器系统，Sun为它的Unix工作站开发了NFS（Networked Filing System，网络文件系统）。所有这些努力都是为了让对本地和远程资源的访问不为用户所察觉，不需要用户的干预。

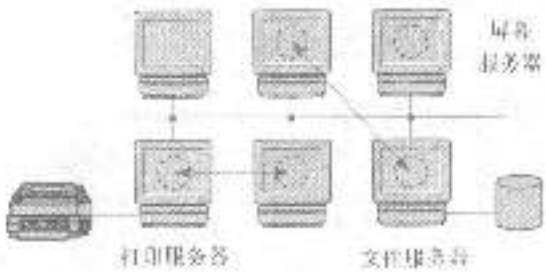


图2-15　在网络上运行的客户机-服务器应用

引入这种联网的构架（见图2-15）是为了提供打印机共享和文件的集中归档，但是，这种构架却被更充分地用来实现跨多台（一组）机器分配处理负载。现在，程序在多台地理上并不接近的计算机上运行，紧密协作完成处理和显示信息等任务的情况已经很普遍。请求的发起者被称为“客户机”，服务的提供者被指定为“服务器”。本地的屏幕依旧负责显示编排方面的工作，而由更为强劲的中心主机提供数据库功能、电子邮件分发或Web服务。这是对多任务程序的极大扩展，此时，协调的任务由不同的机器共同分担。这种安排被称为客户机-服务器（client-server）计算，它是由MIT的X Window系统率先开发的，这种设计策略的最终影响现在还不清楚。

客户机进程通过向合适的服务器发送一个请求消息，开始交互过程（参见图2-16）。响应消息有可能包含所请求的数据，或仅仅是一个确认，说明已经采取了动作。随后，进一步的请求就会从客户端到达服务器端。图2-17中，一台过时的200 MHz PC（名为pong）运行着Linux和X Window屏幕服务器软件。它已经使用了强大的文件服务器（名为kenny），它还远程登录到另一台联网的主机（名为milly）。这台PC实际上相当于X终端，显示那些通过网络发送给它的文本和图形。实际的工作由另两台计算机执行，采用这种方式时，本地计算机在处理能力上的不足不是那么明显。

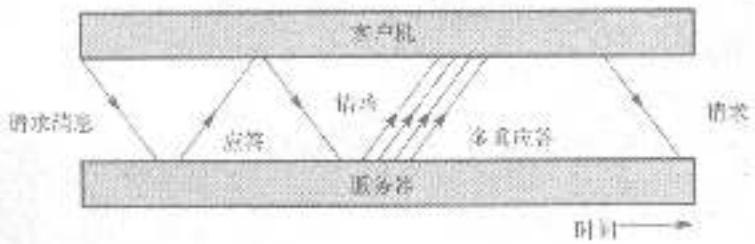


图2-16　客户机-服务器交互的时序



图2-17 跨网络的客户机-服务器操作

2.11 可重配置硬件：读取-执行的另一种替代方式

冯·诺依曼在20世纪40年代有关可编程性（programmability）的思想，必须放在当时能够使用的技术的环境中加以考虑。当时，如果需要改变一台机器的动作（功能），唯一的办法就是使用数以百计的插头和电缆，重新配置大量的一排排开关或庞大的接线板。他的中心思想是由易于替换的程序来控制机器的活动。人们甚至从来没有想像过使用“可载入”（loadable）模式来装配硬件，使之完成特定的任务。

但是现在，这恰恰出现在可编程逻辑器件（Erasable Programmable Logic Devices，EPLD）中。通过下载初始化数据，可以快速地重新配置这类设备，甚至都不需要将芯片从设备上取下来。通过这种方式，我们能够创造出截然不同的逻辑电路，执行不同的功能，而不需改动任何硬件。如果这种技术更早出现，那么，我们也许现在就生活在没有指令集和读取-执行周期的世界中。

2.12 小结

- 数字计算机遵循由冯·诺依曼为我们勾划的轮廓，即机器（计算机）载入指令清单（程序），然后一条条指令地执行下去（读取-执行周期）。
- 采用非专用硬件，能够执行任何由指令构成的程序的优点，已经得到了很好的利用——同一PC既可以用来计算工资单，也可以用来监测水的净化。
- 计算机中使用二进制，是因为使用二进制能够方便地用开关器件（晶体管）的阵列表示数字。
- 每台计算机都能够执行一系列基本的操作。所有的程序都由这些操作组成。每个操作都有唯一的二进制代码与之相关联，以触发特定的操作。
- 使用二进制指令代码直接编写程序既长又乏味且极易出错。高级语言允许程序员使用更为适合的方式来编写程序。此后，需要编译器将HLL程序转换为二进制机器代码，以便计算机能够理解。
- 编译器检查并转换完HLL源代码后，需要链接器将各种模块，包括库函数，绑定到一起。所有这些工作最终产生一个可执行程序。
- 在计算机内，二进制代码还用来表示数据，比如ASCII字符代码集和二进制整数。
- 操作系统，比如Linux或Windows XP，是塑造计算机“个性和印象”的软件，同时还提供许多方便的功能。
- 软件实用工具和其他资源现在能够通过网络容易地得到，这催生了客户机-服务器程序设计范例的产生，这种方法可以将处理工作分配在几个计算机之间。

实习作业

我们推荐的实习作业要求了解本地操作系统（Unix或Windows）的更多细节，涉及到管理个人目录空间和检查设置文件的访问权限。

首先要确定在本课程后面的程序设计练习中使用哪个编辑器。试着写一个“hello world”程序来测试编译器和链接器的各项设置。

及时修正自己对二进制计数系统的理解，避免以后的学习障碍。

练习

0001. 说“计算机理解一个程序”，其意思是什么？

0010. 列出五种CPU的名字和编号，并给出它们的制造厂商名称。

0011. 比较一下将法语翻译成英语及将C转换成机器代码。

0100. 下面这些词的含义是什么：汇编语言、二进制可执行程序、编译、登录、崩溃。

0101. 选择一个编织图样，对它进行分解，用简单的语言解释它的操作。为什么HLL编码要优于汇编语言级别的编码？

0110. 下面机器上常见的应用程序是什么？IBM AS/400、Sun工作站、Cray处理器、Psion Organiser（一种PDA）、PC兼容机。

0111. 什么是程序？冯·诺依曼结架的特别之处是什么？哈佛结构是指什么？

1000. 用下面的单词造一个句子：以太网、微、位、远程、窗口、局域网、每秒、100。

1001. 什么是十六进制（hexadecimal）?它和二进制有什么区别？

1010. 分析表2-4中的ASCII字符，可以看到我们可以将‘A’读做：65、41H或1000001。那么‘a’呢？分别用十六进制、二进制和十进制写下你的名字来。$0A $0D这对ASCII对文本窗口（或VDU）意味着什么呢？使用图2-10给出的小程序发送不同的ASCII码到屏幕上。将7改成‘7’，看看会有什么情况发生。

1011. Unicode是什么？试着使用http://www.google.com搜索Internet上的内容。

1100. 奔腾处理器能识别多少种指令代码？回去查看图2-3给出的例子。比较机器代码的二进制地址和汇编语言的十六进制数。它们一致吗？

1101. GNU是什么？（这次不要使用Google，试试Unix的apropos gnu命令。）

1110. 链接器是什么？什么时候我们需要它？

1111. 如果下面列出的这些程序运行太慢，你会怎么办？

(a) 流行的应用程序，比如Word。

(b) 你的数学计算作业。

(c) 用SQL写的复杂的数据库查询。

(d) 电子表格的宏。

课外读物

- 在你使用的图书馆中，计算科学区的代码是什么？
- 你能远程登录到图书馆的目录系统，查找书名，并预订它吗？你能在终端上续借一本书吗？这样可以避免罚金。

```
if (today == thursday)
     { 阅读《卫报》在线 }
else
     { 浏览最新的《Dr. Dobbs Journal》 }
```

- 查找并检查Tanenbaum（2000）的经典教科书。
- Unicode的主页：

http://www.unicode.org/

- 传说中的GNU项目的方方面面：

http://www.gnu.org/

- 计算和电信科学的在线术语汇总：

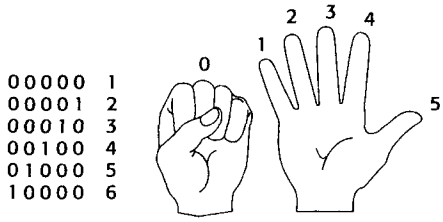
http://www.infotech.siu.edu/csc/help/glossary.html

• 一份免费编译器的清单：
http://www.idiom.com/free-compilers/
• Patterson和Hennessy（2004）
• Heuring和Jordan（2004）
历史性的综述和注解，参见：
Tanenbaum（2000），第1章，是对本章主题的宽泛介绍。
• 这些网站可以通过本书的配套网站访问：
http://www.pearsoned.co.uk/williams

附录：以11为基的计数

人们已经接受了以10计数的习惯，也就是以10为基，因为我们有10个手指，但这并不是完全理性的。以促使我们采用以2为基进行计算的电子开关为例，它只有两个状态：开和关。现在，只用一只手，如果我们一次只允许一个手指的话，我们能够保持6种状态：无、拇指、第一、第二、第三以及第四根手指。试着做一下。然后，使用两只手，我们实际上有11种状态，所以人类应该以11为基进行计数才是合理的！

当然，十个手指的不同伸屈组合有2^{10}（1024）种状态。但是，最起码我不知道怎么做以1024为基的算术运算。



第3章　功能部件和读取-执行周期

根据数字计算机系统级别的结构图，我们可以看到的基本功能模块有：CPU、存储器、输入输出以及将它们连接起来的总线。它们都参与到读取—执行周期（fetch-execute cycle）中，它是计算机活动的基本单位，通过它，计算机能够“理解并执行”用正确语言编写的任何程序。这种按照指令流单执行相应动作的能力，是计算机和其他类型机器的根本区别。保持这些模块之间的同步至关重要。因而，在计算机中，系统时钟信号被分发给所有的组成部分。为了访问存储器中的存储单元，人们制定了一套严格的寻址方案。并行端口是最简单的输入-输出装置。

3.1　各部分的命名：CPU、存储器、IO单元

尽管从理论上讲，数字计算机可以看做仅由三个主要的硬件单元组成（见图3-1），但打开PC的机箱后，我们会发现PC的主板常常要复杂得多。本节的目的是区分每个部分的功能与用途，以及它们与基本功能的联系。图3-2给出了一个现代PC主板的示意性布局，可见的部分都已标注出来。

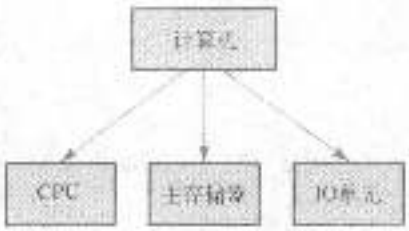


图3-1　计算机的基本部件

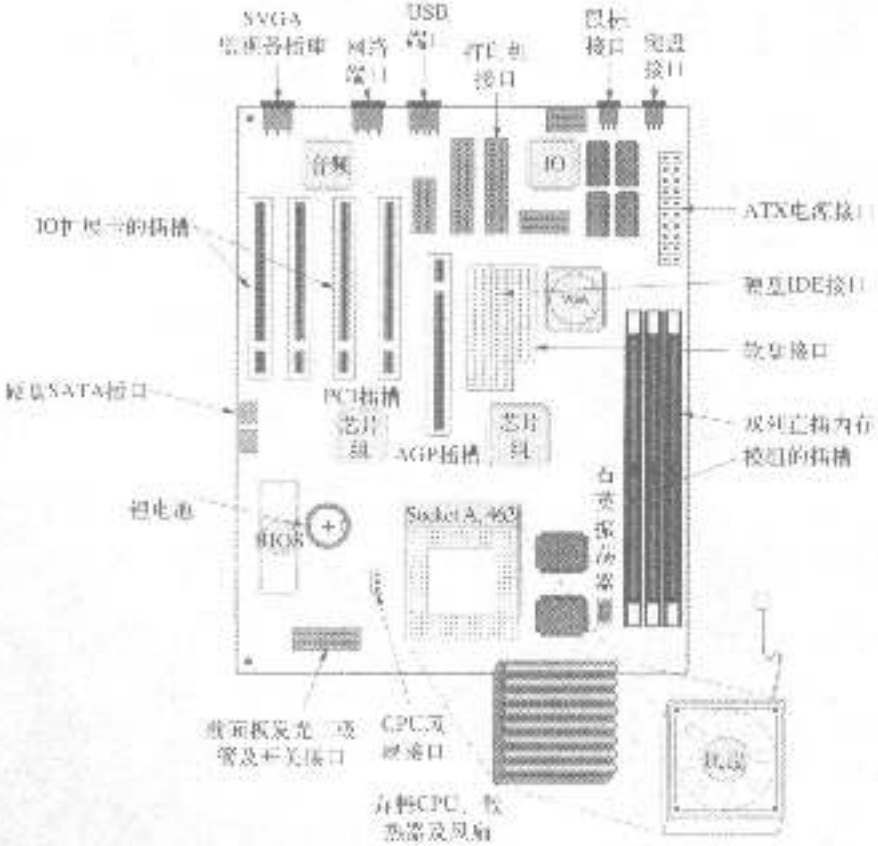


图3-2　PC-AT主板上CPU、存储器和IO插槽的位置

打开PC机箱观察其内部时，首先要识别出主板上的三个部件：**中央处理单元**（Central Processing Unit，CPU）、**主存储器**（Main Memory）和**输入输出**设备。这三者是维持数字计算机运行的最基本部件，所有其他部件都可以归类为附加性的奢侈品。后面，在第12章中，我们将会研究磁盘驱动器，它们是在线数据存储设备中最重要的部分，但现在我们暂时忽略它们。

当前的CPU是**奔腾**（Pentium）系列，它们都装有大型的金属散热片以散发过剩的功率，保持硅芯片不至于过热。不同类型的处理器产生的热量也不同，奔腾处理器可能会产生65W热量，而ARM处理器可能只有1W。根据需要，散热片上有时会安装小型风扇，协助散热（人的大脑的额定功率为12W左右）。现在，Intel在其处理器内核中内置温度探测器，当温度过高时，自动降低系统时钟，减少热量的产生，避免芯片烧毁。图3-2中给出了CPU直接安装到Socket A中的示意图，Socket A是AMD Athlon/Duron处理器在主板上的插座。将所有462个针脚（这种类型的处理器的规格）插入到（承插方式的）插座中并非易事，因此，主板一般会提供零插拔力（zero insertion force，ZIF）单元，只需将CPU（小心地）轻放到正确的位置，然后用侧面的杠杆进行紧固即可。更新型的Socket 939（提供更多的针脚，Athlon 64需要这种插座）操作方式完全相同，稍老的Pentium II和III使用另外一种方式，这种CPU先被固定到次级电路板上，然后插到主板上的专门插槽中。表3-1列出了现有的一些处理器插座。一种好的趋势是，新型的主板上现在提供固定点，以承载CPU风扇和散热片的重量。

表3-1 插座大小和类型的变化

插座	针脚数	最大时钟（MHz）	处理器
486	168	33	486DX
插座1	169	33	486SX/DX
插座2	238	50	486SX
插座3	237	50	486SX/DX
插座4	273	66	奔腾60
插座5	320	66	奔腾75
插座7	321	124	AMD K6
Slot1	242	133	赛扬
Slot2	330	133	奔腾II
插座370	370	133	赛扬(1.4 GHz)
插座A	462	200	Athlon XP(2GHz)
插座423	423	100	奔腾IV(2 GHz)
插座478	478	200	赛扬(2.8 GHz)
插座603/4	603/4	200	至强(3 GHz)
PAC611	611	200	安腾2(1 GHz)
插座754	754	200	Athlon64(2.8 GHz)
插座940	940	200	Opteron
插座939	939	200	Athlon64(3.8MHz)

主板有几种不同的尺寸，上面的芯片的组织方式也稍有不同。它们分别为：AT、Baby-AT、ATX、ATX-mini等。遗憾的是，如果尝试替换使用这些不同尺寸的主板，可能会产生意想不到的问题，比如：CPU的散热风扇可能和软盘部分在空间上冲突。有时，惟一的解决方案是重新购买一套附带主板的新型机箱。最常见的存储器是**DIMM**形式的小型电路板。有时，这些电路板两面均安装有存储芯片。有大量的IO设备需要连接：键盘、鼠标、屏幕、打印机和磁盘驱动器，同时不要忘记前面板的各个发光二极管以及重置开关！在第11章中，我们将会论述如何利用**扩展槽**进行更多的IO连接。但是现在我们必须从设备大体上的功能入手，来了解和掌握这些设备。认识计算机能够运作所必需的基本功能，从长远来讲，要比能够辨别GTi型号的豪华版重要得多。

如图3-1所示，计算机可以划分为三个主要的子系统：CPU、主存储器和输入输出单元。每个

子系统常常又由许多部件组成，在主板上，所有的部件都通过信号高速公路，或称为总线，互连起来。总线（bus）是一束导线，可以是电线或电路板上的通道。Intel 8086拥有20条共享的地址/数据线，以及另外的17条用于控制的线路。在Intel奔腾处理器中，数据总线扩展到64条，地址总线扩展到32条，下一代安腾处理器地址线将会达到64条。每个硬件设备都连接到地址、数据和控制总线上，通过这种简单的方式，我们可以方便地构建出每个单元能够与所有其他单元通信的复杂系统。新单元的加入对现有系统几乎没有影响，故障单元可以替换出来加以测试。如果你打开PC，把看看构成总线的导线时，你可能会很失望。承载众多芯片的电路板（或称主板）可能会由多个层构成，并且一般涂有绿色的不透明漆以保护总线的精巧细节。总线互连示意图中在表示总线时一般并不画出单独的线路，而是将其表示为宽的通道（如图3-3所示），我们假定总线所连接的设备都能够访问它们需要的任何信号线。

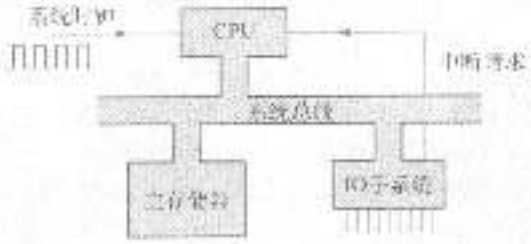


图3-1　系统互连示意图

总线并非惟一的选择。也曾有人尝试过使用点对点的互连方案，但是，当涉及的部件超过二个时，线路就会纠缠不清，甚至根本不可实现。n个部件，如果实现完全的两两互连，所需的通道数量可以用数学的方式表示出来：

$$\text{通道的数量} = n(n-1)/2$$

要知道，每个通道依旧需要一条全宽的数据高速公路，可能由32条线以及几条（约6条）控制线构成。但是，这种方式也有优点，因为所有的通道都点对点通信，因此，并不需要全宽的地址总线。地址有两种用途：在广播式高速公路上指定目的设备，以及选择设备内的位置。对于点对点通道来讲，不再需要前者。下面这个例子中，需要连接的单元数量（n）为30：

$$\text{不同通道的数量} = \frac{30 \times 29}{2} = 435$$

$$\text{导线的数量} = 435 \times (32+6) = 16\,530$$

这种互连方案所需的导线数无法控制，计算一下图3-4给出的简单例子中的连接路径数，就能够验证前面的计算方法。同时，为了对比，图中还给出了一个总线方案，含6个单元，从中可以看出总线方案要简单得多。

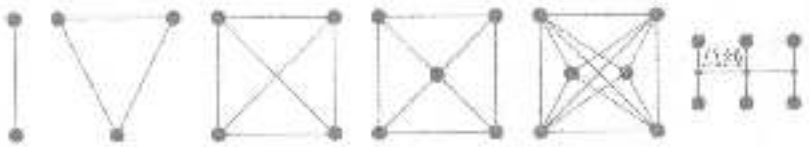


图3-4　与简单的总线互连相比，点对点方案复杂度剧增

因此，考虑到计算机内需要互相间进行通信的设备数目众多，总线互连方案很快就受到欢迎。但是，总线架构有一个严重的缺陷。因为电子总线和铁路线一样，同一时间只能传送一项数据，即使采用更快的处理器，它所能够达到的性能也会受到限制。这个约束术语称为总线瓶颈。因此，信号在总线上的传输速度，很快成为考虑增加系统吞吐量时一项至关重要的参数。要想达到更快的传输速度，仅仅增加时钟频率，降低脉冲的宽度是做不到的。总线通道的电容和电感为时钟速度设定了上限。为了避开这个问题，微处理器正在采用更宽的数据总线。自20世纪80年代早期以来，数据总线已经增长了8倍，从8到64条线。另外，功能更为强大的计算机采用几条独立的总线，从而允许计算机同时传送几项数据。

我们在第14章中介绍联网计算机时，还会遇到由数据传输的最大速率造成的限制。为了更进一步地对硬件进行研究，我们一般将CPU划分成两个功能性子单元：**控制单元**（Control Unit，CU）和**算术逻辑单元**（Arithmetic and Logic Unit，ALU）。将图3-5加到图3-1中，整个结构就会展现在我们面前。

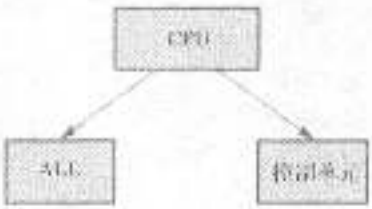


图3-5 进一步的分解

3.2 CPU的读取-执行周期：高速且单调

计算机必须读取并执行每个程序，包括操作系统本身，每次一条指令。乍看起来，与有生命的生物体能够同时执行多个动作相比，这对性能的提高好像是一种严重的阻碍。基本的操作——我们称之为**读取-执行周期**（fetch-execute cycle），是将程序内的每条指令从内存读入到CPU中，然后解码并执行。不过，由于电子硬件速度非常快，从而使得这种冗长的重复循环具备实际价值。在计算机发展的历史上，任何时候，读取-执行周期所涉及的三个部件中（存储器、总线和CPU），总会有一个成为限制因素。这既会影响到计算机工程师们在工作时必须遵守的设计参数，也会影响到解决问题时算法的选择。例如，有时问题既存在"存储器敏感"的方法，又存在"计算敏感"的方法。如果存储器速度很快且价格低廉，则前者更适合，否则则应选择后者。目前，**主存储器**所使用的DRAM芯片还不能和CPU一样快。虽然存在更快的存储设备（SRAM），但成本极高，所以它们只用在小型的快速缓冲器中，称为**高速缓存**（memory cache）。它们通过保持当前指令和数据的副本，能够在某种程度上降低主存储器的访问延时。第12章将会更深入地探讨这个主题。为了进一步降低冯·诺依曼"单流水线"瓶颈所带来的负面效应，新一代RISC（Reduced Instruction Set Computer，精简指令集计算机）依靠完成流水线，通过同时执行几条（约5条）指令来加速读取-执行的速率。第12章将会更完整地解释这些内容。

尽管我们已经熟悉了计算机，但是，对于我们来说，真正理解它们那种难以置信的运作速度，还是有些不太可能。为了试着弥补我们理解上的这种差距，表3-2给出了各种活动的速度，以进行对比。

表3-2 各种活动的时间对比

ns	μs	ms
$\frac{1}{1000000000}$	$\frac{1}{1000000}$	$\frac{1}{1000}$
读取-执行周期 10 ns	光线 300 mμs	人类的反应 300 ms
逻辑门延迟 5 ns	TV行扫描 60 μs	TV读 20 ms
SRAM访问 15 ns	中断 2～20 μs	硬盘访问 10 ms
	引擎火花 10 μs	汽车中轴（3000 r/min）20 ms

从表3-2，可以很容易地看出计算机操作和现实世界在速度上的差异。即使对于我们来说已经非常快的事物，比如单个电视扫描行，都要比CPU的读取-执行周期慢上千倍。

正如我们所见，计算机程序由二进制编码的指令构成，例如：

```
1011 1000 0000 0000 0000 0001
```

是一条奔腾指令，用汇编语言助记符可以表达为：

```
MOV AX, 0x100
```

或者以C语言的形式（更易认）表达为：

```
nooont = 256;
```

通过这些，我们能够很容易地看出使用编程语言的优点（甚至是低级的汇编语言！）。这条指令将CPU的累加寄存器（AX）设为数字256。

读取-执行周期是这样一个过程：CPU从内存中取出程序下一条指令，对它进行译码，执行它请求的动作。完整地介绍奔腾CPU的读取-执行周期，远远超出本章的内容，甚至本书的内容。但是，我们能够给出一个概括性介绍，帮助读者认识读取-执行周期实际上由几个独立的活动阶段构成。在图3-6中可以看到CPU、**指令指针**（Instruction Pointer，IP）、**累加器**（Accumulator，AX）和**指令寄存器**（Instruction Register，IR）。主存储器保存程序，其中可以看到二进制形式的MOV AX，256指令。存储地址寄存器（Memory Address Register，MAR）也画了出来，因为在后面的叙述中将会提到它。图3-7展示了读取阶段，图3-8是MOV AX，256指令的执行阶段。现在，请试试看，能不能描绘出构成读取-执行周期的动作序列。

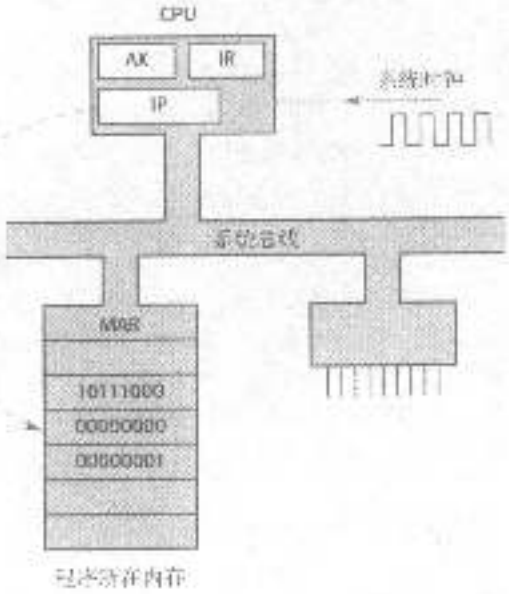


图3-6　指令指针寄存器（IP）指向内存中的下一条指令

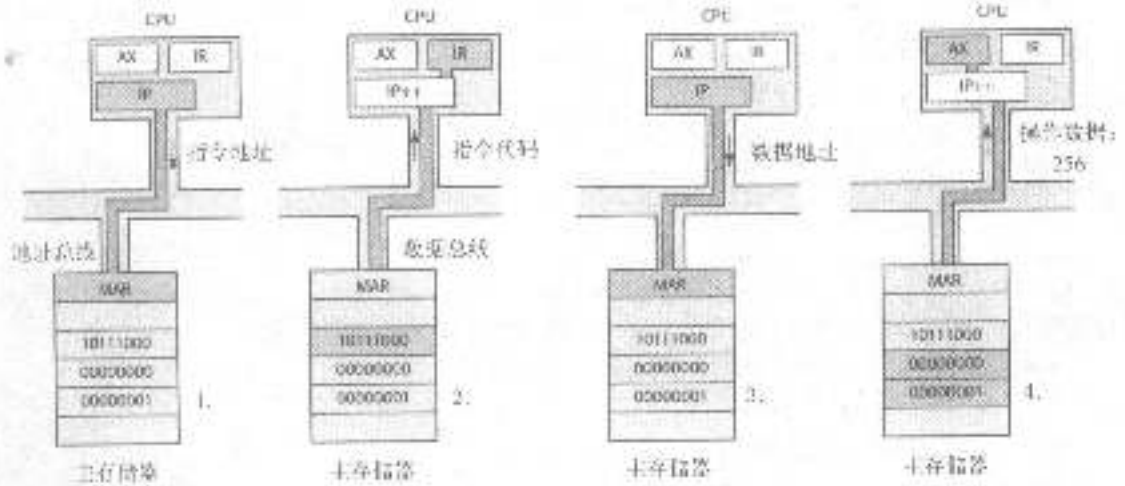


图3-7　读取-执行周期的读取阶段

图3-8　MOV AX，256指令读取-执行周期的执行阶段

相比于现代处理器实际的读取-执行周期序列，这个示意性的例子相对比较简单直接。有些指令需要一个额外的执行周期，先从内存中读入一个地址值，然后使用这个地址值从内存中读出期望

的值进行操作。在此没有必要复述每个奔腾指令的读取-执行周期。只需理解基本原理，并能够用到简单的指令上就够了。程序员只有在使用未经验证的原型硬件时（最好能够避免这种经历），才会碰到这种级别的问题。

总结一下：读取-执行周期序列中，程序的每条指令从内存中读出，译码并执行。为了一项操作能够执行，这个过程有可能需要从内存中读入更多的数据项，最后的结果写回到内存。

Unix和Windows都提供工具来实时地监视各种活动。Sun工作站上的Unix工具称为**perfmeter**；在Linux上，它是**xsysinfo**；在Windows上，它是**Performance Monitor**（性能监视器）。Sun的perfmeter由命令行启动，显示当前CPU的负载。

```
perfmeter -t cpu &
```

图3-9中，活动的波峰是由于为本图示截取和处理屏幕图像所致。

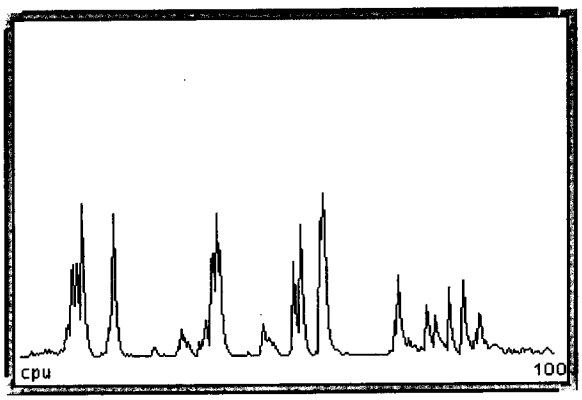


图3-9 使用Sun perfmeter显示当前CPU的读取-执行负载

3.3 系统总线：同步或异步

由CPU通过系统总线发出的信号分为以下三组：

- **数据总线**：典型地为32位宽，但会很快扩展到64位。
- **地址总线**：32位宽，很快将会需要更多。
- **控制总线**：约15根，负责启动和停止各种动作

控制总线中，有一条线路为**系统时钟**，它由高频的石英振荡器（在主板上为一个小的银白色圆柱体，常常位于靠近CPU的地方）产生。大多数情况下，是由CPU通过总线向其他单元发送信号，开始一项操作。然后，这些单元通过总线回送信号，做出响应。有时，动作可能是由非CPU的单元触发，它可能会暂时取代CPU，获得总线的控制权。总线信号的序列必须遵循一个十分精确的时序模式。如果时序同步完全和系统时钟信号吻合，则称总线是“同步的”。

同步总线（synchronous bus）要求所有连接其上的部件、内存和IO芯片以相同的速度运转。从硬件的角度看，这种方案十分容易实现，但缺乏灵活性，不易接纳大量以不同速度运转的设备。通过图3-10中的信号轨迹，我们可以看到同步总线上的时序关系。要注意，64条地址线和32条数据线均表示为单个总线线路。所传送的二进制模式是什么并不重要，我们只需知道什么时候总线上有合法的数字。这幅图表示的是一幅理想化的四轨迹示波器的屏幕。在实际的计算机中，实际的电压轨迹要杂乱得多，因而也很难理清。以100 MHz运行的主板——该频率常常称为前端总线（Front Side Bus，FSB）速度，单个时钟是10 ns，200 MHz时钟的周期为5 ns。试着跟踪图3-10中的读取-执行周期。首先，存储在IP寄存器中的下一条指令的地址（地址1）被复制到地址总线，而后通过读/写控制线，告诉存储设备从这个地址单元执行“读”操作。指令代码从内存（指令）复制到数据总线，

再转回到CPU，存储在IR寄存器中。读取周期现在结束。执行周期需要另外两个总线周期，第一个读入用于处理的数据项，第二个写出结果。

异步总线（asynchronous bus）更为复杂，但也更为灵活。它们允许CPU根据正在通信的单元的速度，调整其循环周期。这两种系统之间重要的不同在于，集中化的系统时钟不再存在，并且引入ALE和DTA控制线（参见图3-11）。**地址锁存允许**（Address Latch Enable，ALE）信号由CPU控制，用来向总线上的部件表明什么时候地址信号合法。实际上，它部分地充当了系统时钟的角色。**数据传送确认**（Data Transfer Acknowledge，DTA）信号并非由CPU发出，而是由连接在总线上的部件生成。它告诉CPU什么时候写数据的操作成功完成，或者什么时候数据合法，可以读取。如果内存需要更多的时间，它可以延迟激活DTA信号，因而延迟总线周期的完成。ALE和DTA是数据传输**握手系统**（handshake system）的一个实例。

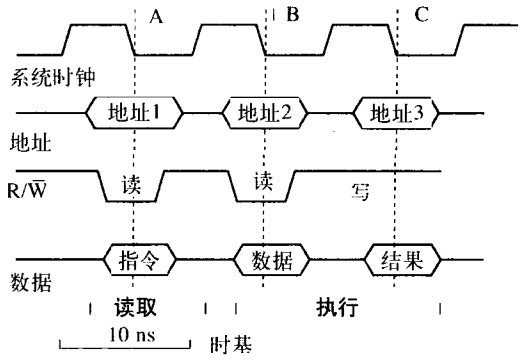


图3-10　同步总线读取-执行周期时序图（100 MHz）

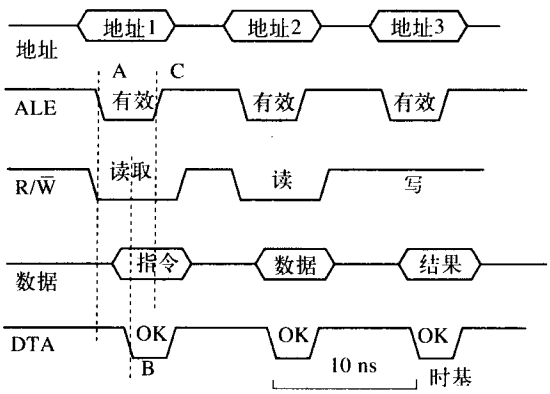


图3-11　异步总线周期的时序图

考虑图3-11，我们可以看到，该序列开始时，CPU将地址值放在地址总线上，而后激活ALE控制线。所寻址的部件将寻址的存储单元复制到数据总线上，并激活DTA控制线。当CPU识别出DTA信号后，它会读数据总线，然后撤消ALE，清除地址。这个序列沿着虚线A、B和C进行。

同步工作方式和异步工作方式之间的差异和所有的通信信道相关，包括网络，因而我们需要对它做更全面的阐述。同步系统中，从CPU到所有连接在总线上的部件都共享同一时钟信号，它们均由信号的下降沿控制，如图3-10中的虚线所示。边A告诉内存将指定位置的指令复制到数据总线上，因为读/写信号被置为读。接下来，边B从内存中获得一些数据，而边C将指令执行的结果写回到另外的内存单元。使用这种方式，内存芯片只响应CPU的控制线，但是，它的动作每次都必须足够快，因为CPU没有办法分辨出数据总线上的数据是有效数据，还是随机振荡。如果内存不能足够快地得到一个数据项，CPU就不会注意到这些，而是会继续执行下去，直到错误的数据使它崩溃为止！现在，你能够明白为什么CPU的运行速率只能和总线上最慢的部件一样快了。当系统需要使用几种前一代的芯片时（这常常会发生），这就是一项严重的约束。

3.4　系统时钟：指令周期时序

经过3.2节和3.3节的介绍，你现在应该很清楚了，读取-执行周期不是一个简单的事件，而是由许多不同的阶段，或**微周期**（microcycle）组成。这些都由系统时钟产生，对于200 MHz时钟，微周期为10ns：

$$\text{周期}=\frac{1}{\text{频率}},\quad t_{\text{micro}}=\frac{1}{200\times10^{6}}\text{s}=\frac{1}{200}\mu\text{s}=\frac{1000}{200}\text{ns}=5\text{ns}$$

图3-12给出的例子，在整个读取-执行指令周期内有5个微周期。每个微周期负责完成一种特定

的操作。现在CISC和RISC计算机在如何安排和实现微周期上存在重大的不同。在后续的章节中，我们将进一步研究这种区别。

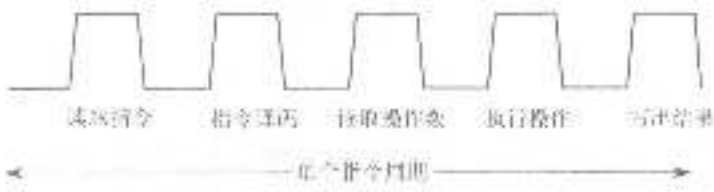


图3-12 多阶段的指令周期

为了提高处理器的吞吐量，时钟的速度不断上升，以至于内存很难在单个周期内做出反应。同时，由于VHF FM无线电通信的广播波段正好位于90～100 MHz（见图3-13），由此引发了更多的问题。这个频率恰好是下一代处理器的时钟所要达到的速度。更糟糕的是，FM广播的天线长度恰好和主板布线的尺度大致相当，从而使得两者成为绝佳的发射器和接收器：

$$\lambda_{m} = \frac{c}{f} = \frac{3\times10^{8}}{100\times10^{6}} = 3\text{m}$$

故而，全波长的四分之一是3/4m，或75cm。

微计算机的先驱们在测试最初的MITS 8080微计算机时，利用了这种效应。它只装备了极少的输入输出设施，只提供单行的八个LED和八个切换开关。如何利用这么简单的设施输出大量的结果，是一项十分需要技巧的任务。之后，人们注意到附近的一部晶体管收音机不经意地成为了一种输出设备。因此，利用收音机的扬声器播放不同曲调的程序，很快被写了出来。无线电波辐射的问题通过使用时钟倍频暂时得以缓解。这种技术允许CPU从主板接受较低频的时钟，然后合成一种更高速率的信号在芯片内使用。150 MHz的奔腾处理器将66.6 MHz的系统时钟加倍。相对于较大的主板布线，CPU的封装要小得多，从而能够降低VHF无线电的辐射量。长期的解决方案是将计算机隐藏在金属制成的箱体内，同时将箱子良好接地，以屏蔽内部的任何无线电辐射。以现今PC主板的时钟为例，它一般运行在200 MHz。假定采用14倍倍频，则奔腾处理器能够运行在2.8 GHz。主板的时钟也称为**前端总线**（Front Side Bus，FSB）时钟，这种叫法来源于以前Slot 1和Slot 2 CPU接口卡的构造。“前端”连接主板上的主存储器，而“后端”连接到子板上的缓存。由于CPU的运行速度取决于主系统总线和内存，因此，对于在更快的芯片内集成高速缓存的需要突然变得十分强烈，从CISC到RISC架构的转移，也变得越来越急迫。第21章将详细地讨论RISC的基本原理和优点。

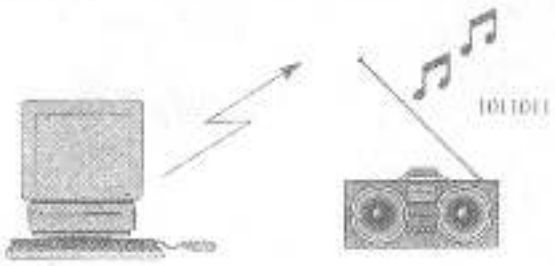


图3-13 计算机发射出的无线电波会对FM波段造成强烈的干扰

计算机内信号的最大速率受到基本电子效应的限制：电阻和电容。尽管我们用垂直的边将信号描绘成尖锐的脉冲，但导线和布线的物理电阻和电容都会产生让边变圆的效应。当脉冲的“压”逐渐与下一个脉冲的上升沿重叠时，就会产生问题，参见图3-14。

为了增加脉冲的速率，即时钟的频率，计算机硬件工程师需要降低互连总线布线的电阻和电容。现已通过将越来越多的电路集成到单晶片上，成功地做到了这一点。信号线路的电容越低，将它充电到需要的伏特（逻辑1）所要注到的电子就越少。类似地，它也能够更快地放电到低电平（逻辑0）。

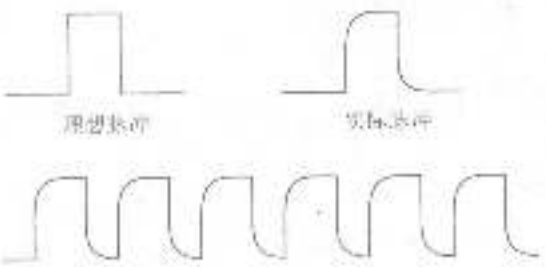


图3-14　时钟频率的影响

许多年来，表示逻辑1和逻辑0的电平，一般都是按照德州仪器公司在1970年为他们的TTL元件所制订的标准。逻辑0用低于0.8V的电压表示，逻辑1用大于2.5V的电压表示。如果电源为5V，这种方案工作得很好。但是，现在为了提高时钟速度，几家芯片制造厂商采用较低的3.5V电源。这项改动成功地降低了传输线路充放电的延迟，但却要求更清晰的信号边沿和更小的电源噪声。快速芯片最新的电压级别甚至更低（1.8V）。另一项制约CPU时钟速度的重要因素，是芯片内产生的热量。硅晶体管在过热的情况下不能正常工作，时钟越快，所产生的热量越多。奔腾处理器会由于过热而停止工作，或者需要额外的散热设备，比如风扇，以使它们保持运转。也有人曾经用冰块为CPU降温，甚至可以买到一种商业的冰箱部件，可以将芯片的温度降到0度以下。它们常常用在CPU的开发或超频试验中。

3.5　预取：前期工作以使速度得到提高

由数据副总线的读取-执行序列（见图3-12）可以运行得更快，但要受到当前技术水平的限制。人们早就意识到，在读取-执行周期中，许多时候CPU被挂起，这主要是因为主存储器DRAM依旧比CPU慢很多。如果能够在前一条指令完全结束之前，就开始读取下一条指令，将每个指令周期与下一个重叠，则能够使性能得到很大的提高。于是，预取队列（pre-fetch queue）被发明出来（见图3-15），应用这项技术后，指令持续不断地从主存储器读出，保存在CPU内部的队列中，准备好译码和执行。CPU的构架也被重新设计安排，以支持这种功能的分离，并且最终会借鉴现代RISC处理器的流水线方案。

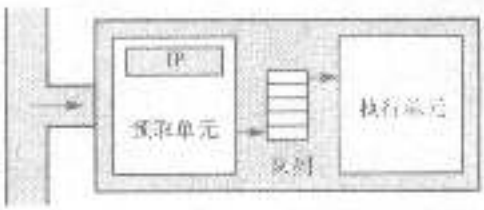


图3-15　带预取队列缓冲区的CPU控制单元

预取缓冲区由预取单元（pre-fetch unit）填充，预取单元在执行单元（execution unit）真正需要指令之前，从内存中读入它们。这种活动的重叠为CPU的运行引入一定程度的并行机制（parallelism）。它之所以能够工作，是因为指令译码和执行所需的大部分时间内不涉及任何总线活动，从而使得预取传输可以进行（现实世界中这种做法也很常见。人们为了应付突发的需求，会在周末大采购到来之前，事先在商店内储备各种杂货）。活动的重叠已经成为计算机内为加速运行而常常采用的技术（见图3-16）。但是，当程序出现条件分支或选择时，比如由if-else或switch-case语句所形成的结构时，就会产生问题。预取缓冲区只能跟踪条件分支中某个分支的指令流，另一条分支依旧在主存储器中，未被读入。如果IF语句决定沿另一条‘else’例程执行，整个预取缓冲区都得清空，有时还会丢失再次读入的程序中其他例程的指令。这就如同你在买了鲑鱼排后，发现这个周末的客人都是严格的素食主义者。所以还需再次到超级市场去买些豆腐来。在第21章中，我们将会详细讨论相关的技术（预测或猜测选择分支的结果，从而能够预取正确的例程），但如果猜错，时间上的损失依旧存在。



图3-16 读取和执行操作重叠带来的速度优势

仙童（Fairchild）公司曾在F8微处理器家族上尝试过旨标在于加速读取-执行周期的另一种有趣的架构，这种架构将IP寄存器从CPU中移出，放在存储程序的内存芯片中。采用这种方式，每次读取周期中将指令地址发送到内存的总线传输延迟得以消除。这显然会带来其他问题。比如，在需要扩展内存时，应该怎么办呢？

3.6 存储器长度：寻址宽度

主存储器（在不引起混淆的情况下，一般称为内存。译者注）是计算机正常运行的核心。程序就存储在主存储器中，指令代码要从这里读取并执行。同时，为了方便和速度，相关联的数据一般也载入到主存储器中。鉴于存储器作用重大，选择存储器的类型极为重要。因为访问存储器的速度是决定指令执行速率的重要参数，进而决定程序能够多快完成。理论上，任何技术都能够胜任：纸带、磁盘、声延迟线、DRAM、SDRAM、等等。但是，当前最普遍的选择还是DRAM，它能够大批量制造，在单个计晶片上可以提供达1000百万位的读写存储，访问时间最低可达15 ns。

主存储器的可用大小是一个重要参数，但对系统的设计者来说，更加重要的信是内存的最大长度。这是由CPU地址寄存器（如IP）的宽度（位数）所控制的。很明显，地址总线也必须足够宽，才能传送最大的地址值。然而，摩托罗拉MC68000最初有32位宽的CPU寄存器，地址总线却只有24位宽。这种需抉决定的原因，是为了通过限制引脚的数目降低封装的成本！同样是由于成本问题，Intel在8086处理器上将地址和数据总线信号共用相同系列的引脚，这种解决方案（沿相同的总线线路，先发送地址，后跟数据）确实会带来时序上的损失，并且一般会使主板的电路更加复杂，因为不可避免地需要在使用前将信号分离出来（解复用）。

看看图3-17，了解一下地址宽度和内存长度之间的关系。我们可以将它想像成邮政地址。如果住宅的号码只允许使用两位数字，则每条路上的住宅数都要少于100个。重要的是，永远不要将内存宽度（数据）限制与内存长度（地址）约束混淆。先了解与内存相关的四对数字：16位=64 KB，20位=1 MB，24位=16 MB和32位=4 GB，64位=16EB。它们表示了计算机的五代。有了它，在快速地估计位数时，就不必进行计算了。大多数计算机系统的内存从来没有满过，除非接近产品生命期的结束。因此，基于X86的计算机被在产品生命周期结束时，均装备最大限度的64KB，Atari的热爱者购买额外的RAM卡，使之接近16 M的物理限制。当前，装有3GB DRAM的PC正在接近奔腾处理器的最高限制4 GB。这也是引入安腾处理器的主要原因之一。安腾处理器地址宽度为64位，能够直接访问16 EB（Exabyte，2的64次方）的数据。

早期的微处理器以一个字节为单位访问内存，但这并非普遍的标准。老的PDP-8从内存中读取12位的字，它的后继者，PDP-10，访问36位的字。奔腾处理器一次读入64位，到达CPU后，再将它们分解成8位的字节、16位的字和32位的长字。但要注意，外部数据总线的宽度已经不再是衡量机器基本能力的可靠标准。

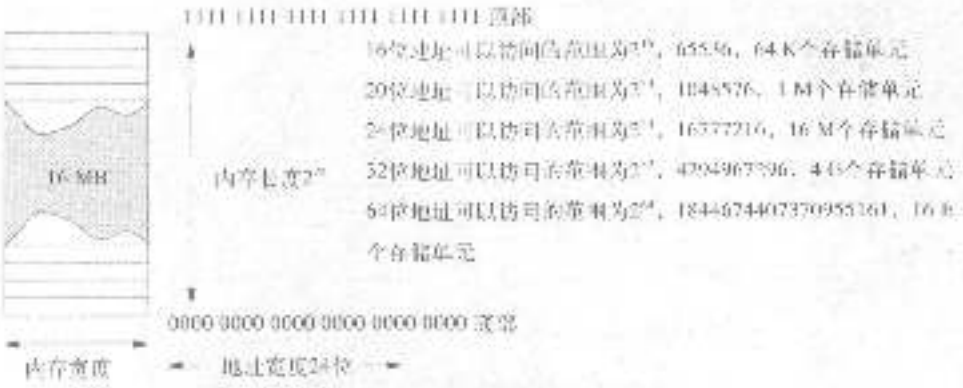


图3-17　地址宽度和内存长度

在处理存储地址时，常常使用十六进制的记法。因此，32位的地址，比如：

1010 0011 1101 0010 0101 1111 1100 0001

可以简化为：A3 D2 5F C1。

十六进制的编号系统列在表3-2中十进制和二进制等同的符号下面。但需要注意的是，十六进制是二进制的简写。

表3-3　不同的编号系统

十进制	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
二进制	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
十六进制	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

值得一提的是，现代操作系统支持“虚拟内存”（virtual memory），这项技术允许主存储器溢出到磁盘上的一段区域，奔腾处理器不但能够通过它的32位地址宽度处理4 GB的物理寻址，还可以通过虚拟内存管理单元支持64 TB的磁盘虚拟地址空间。尽管这明显会影响到性能，因为访问磁盘相比于访问主存储器要慢得多。但是，在多任务系统中，使用虚拟内存带有其他方面的优势。有关虚拟内存的主题，将会在第18章详细介绍。

3.7　字节次序：微软与Unix，以及Intel与Motorola

多字节数据项在内存中的排列，要遵循两种分别由Intel和Motorola独立设立的约定中的一种。开始时，Intel采用下面的策略存储16位数据：将它拆分成两个字节，首先存储低位（least significant，LS）字节，接下来是高位（most significant，MS）字节。这种方式称为**little-endian**（小端在前）。这样做的优点是直观。Motorola避开了常规的逻辑，它选择了先存储高位字节，然后是低位字节，这种方式称为**big-endian**（高端在前）。这样做在实践上的确有一定的优势，即能以“打印次序”表示数据。当跟踪内存中的数据将其显示到屏幕上时，数据的次序完全适合于打印输出。它以从左到右的书写顺序出现：高位一低位，高位一低位，高位一低位。

图3-18清楚地显示出这两种不同的字节次序约定。图中的内存显示窗口分别由两种不同的调试器（Intel奔腾和Motorola 68030）生成。在我们所见到的内存区域内，含有三个整数数组，分别保存字节、字和长型。图中，每个字节由一对十六进制数字表示。因而，01表示00000001，FF表示11111111。

三个数组的C语言声明如下：

```
unsigned char b1[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 251, 252,
                       253, 254, 255};
unsigned short s2[ ] = {1, 2, 3, 4, 5, 256, 255, 256, 257, 65532,
                        65533, 65534, 65535};
```

```
unsigned int b4[ ] = {1, 2, 3, 4, 5, 254, 255, 256, 4095, 4096,
                      4097, 4294657295};
```

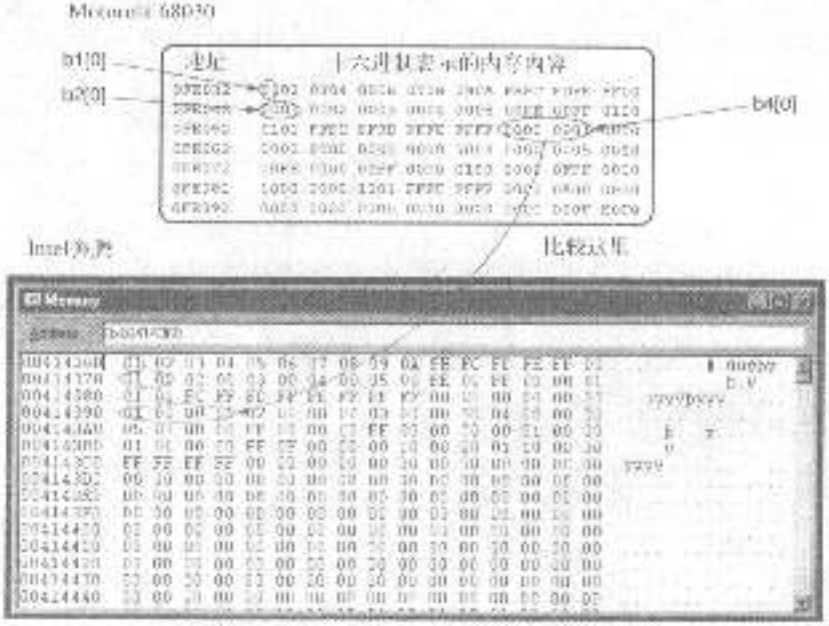


图3-18 多字节整数一字节次序的差异

通过将这些声明加入到C程序中，并分别在MC68030和Intel奔腾机器上编译该代码，就有可能使用调试器显示出含有这一个数组的内存区域。从图3-18可以看出，8位整数（unsigned char）的存储是相同的，16位整数（short）的高位字节和低位字节存储次序正好相反，32位整数（int）也是如此。使用特殊的值有助于辨别这些数据。

这里要说明的一点是，如果存储的所有数据项都是字节（char），则没有次序的问题。如果文件在写入时遵循某种约定，然后被传递到一台执行不同数字体系的计算机上，那么从文件中读取数据时也会发生问题。如果仅存在一种类型的数据，比如说32位整数，那么，从big-endian到little-endian的转换比较直接，字节的位置可以在访问前迅速地调整完毕。但是，如果整数和char数据混在一起，则转换将会极为复杂。整数需要将MS和LS字节交换，而char字符串数据则可以顺序访问，不需要做位置上的调整。

在将数值型的数据文件从PC传输到Unix时，可能会需要调换奇字节和偶字节。Unix平台的dd工具就具有交换字节的功能，因为这项主要的不同，如果将数据文件从PC传输到Sun，必须对它们进行转换。Sun由于在早期的工作站中使用MC68000处理器，因此遵循Motorola的约定，Microsoft则依赖于Intel处理器，因为IBM最初选择Intel 8088作为PC的处理器。在跨网络通信时，这种不一致也会带来麻烦，TCP/IP数据包默认的排列是高位字节在前（big-endian），如果发送者知道接收方的字节排列次序，则可以忽略这个问题，由于大多数据情况下这一点并不能确定，因此，一般均以big-endian格式发送所有的数据包，即使传输发生在两台遵循little-endian格式的主机之间。

在将文本文件从Unix传输到PC时，常常会遇到另一项差异，反之亦然。这涉及到用来标记行结束的字符。PC应用程序常常使用^M（CR，Carriage Return，回车），而Unix则使用^J（LF，Line Feed，换行）。在考虑这两个ASCII控制字符想要表达的意思时，你将会看到，两者实际上都要求将光标或打印头移动到左边界，然后向下走一行。实际上，为了节省存储空间，常常会在文本文件中只包括两者之一，而依靠屏幕驱动程序来重新插入配对的代码。遗憾的是，这个简单的选择为以后的不一致埋下祸根了！

3.8　简单的输入输出：并行端口

从硬件的角度来说，提供字节宽度（byte wide）的输入端口十分简单，如图3-19所示。对程序员来说，输入操作和内存读操作看起来相同：CPU简单地读取指定端口地址，获得数据的一份拷贝。但是，在现代多任务操作系统，比如Windows XP和Unix中，对硬件端口的访问受到严格控制，普通用户需要请求操作系统替他们执行端口读写。我们将在第10章再次介绍这一主题。

输出端口，比如PC机的打印机端口（见图3-20），在硬件方面要稍微复杂一些，因为在CPU写端口结束后，必须采取某种方式记住输出数据最近的值。实际上，在提供输出端口时，必须提供单字节的内存或锁存器（latch），这样，在CPU写出某个数据项之后，它就会持续有效，直到另外的值改写它为止。通过这种方式，连接在该端口上的外部设备在任何时候都有合法的值。对于需要通过这类接口控制设备的程序员来说，重要的是不可能从输出端口读回输出的值。某个值在写入到输出端口后，就不能再访问到。因此，需要保存一份本地副本或镜像，以备进一步使用和修改。这一点至关重要。

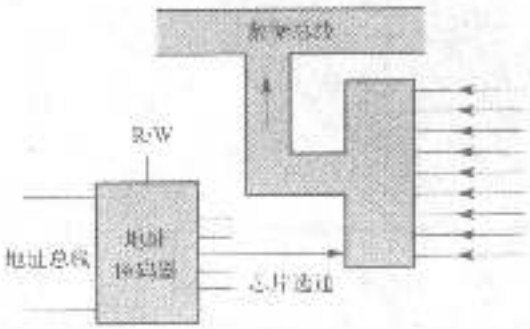


图3-19　输入端口示意图

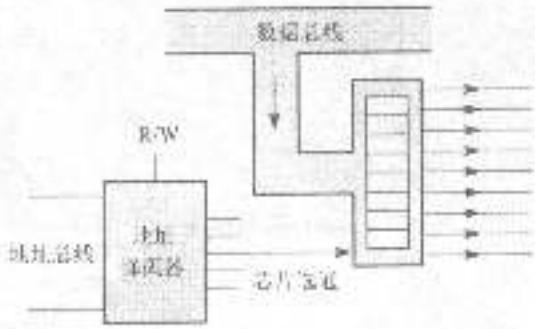


图3-20　输出端口示意图

3.9　小结

- 数字计算机由三个基本的部分组成：中央处理单元（CPU）、主存储设备和输入输出单元。它们对于读取-执行周期的执行是必需的。读取-执行周期是程序存储型机器的特点。
- 所有这些单元都通过并行的总线（PC主板上的导线）互相连接起来。
- CPU可以进一步划分成两部分：算术逻辑单元（ALU）及控制单元（CU）。
- 读取-执行周期是最基本的活动，计算机每秒钟重复上百万次。在读取阶段，当前程序的下一条指令被送入到CPU中。之后在执行阶段进行译码和执行。
- 由于用做互连的总线每次只能传送单一数据项，故而形成性能的瓶颈。
- 计算机内所有单元的运作由中央振荡器（或称时钟）来同步。
- 同时执行几件事可以提高吞吐量。可以通过预测对指令的需求，将它们预取到内存中来加快执行的速度。
- 内存的最大长度由地址的宽度决定。
- 在处理多字节数据时，需要弄清楚构成数据的字节在内存中的次序。
- 对于程序员而言，简单的输入输出端口可以看做是地址空间内的单字节内存。

实习作业

我们推荐的实习作业包括揭开PC的盖子，看看其内部，识别其重要的构成部分，并给出其名称。讨论它们在系统内的功能。给出可更换部件及其要供应的名称。

在这个阶段，要注意多使用Internet获取更多、更深入的技术信息，但要注意避开那些营销性质

的内容和网站，因为这样的网站常常会浪费掉你许多时间来阅读，但却学不到什么有价值的东西。

练习

1. 列举出数字计算机三个基本的组成模块。勾画一幅它们互相连接的草图。试着再插入一个IO单元。
2. 什么是读取-执行周期，为什么它这么重要？读取-执行周期可以拆分成五个子周期，它们是什么？
3. 奔腾2 GHz能够在电视扫描一行的时间内完成多少条指令？2 GHz奔腾处理器的一个时钟周期为多长？假定每个时钟周期能完成一条指令，估计一下，2 GHz的奔腾PC在2 MB的数组中查找一个字，需要花多长时间？
4. 数据和地址总线的宽度如何影响计算机的性能？
5. 术语“总线瓶颈”的含义是什么？提出一些避开总线瓶颈的方案。
6. 系统时钟的作用是什么？停止时钟会不会发生问题？
7. 异步总线的基本特征是什么？为什么它被认为要比同步方式更好一些？
8. Intel如何减少8086封装上的引脚数？这种策略有什么缺点吗？
9. 如果电信号的速度是光在真空中速度的1/3，那么复位脉冲传输PC主板这段距离要花多长时间？放一个VHF/FM收音机在PC的顶部，并将其打开。
10. 写一个制造蛋糕的指令清单。现在告诉一个朋友如何制作蛋糕，通过电话，一次一个操作。也许他们的电话听筒不在厨房中！
11. 为什么提供预取指令缓冲区能够加速处理器的执行速度？使用预取缓冲区的缺点是什么？
12. CPU提供芯片内（Level 1）高速缓存的好处是什么？为什么片外（Level 2）高速缓存效率要低一些？
13. IO端口与内存存储单元之间的区别是什么？相比于输入端口，输出端口需要什么额外的硬件？
14. 在Sun与PC上的文件中，“Hello World”的代码有什么不同？现在，使用含有0～9的文件重复这项练习。什么时候需要使用Unix命令：dd -conv=swab？
15. 从8位转换到16位和从16位转换到32位时，对于微计算机软件应用程序来说，最重要的改变有哪些？
16. 使用Windows性能监视器查看CPU的负载：开始－程序－管理工具－性能监视器。可能需要启动CPU显示：编辑－加入到表－处理器。启动Netscape Navigator或Microsoft Word，检查负载的变化。

课外读物

- Tanenbaum（2000）第2.1节介绍读取-执行周期。
- Heuring和Jordan（2004）详细描述了读取-执行周期。
- 在Internet上有许多有关CPU超频的文章：
  http://www.hardwarecentral.com/hardwarecentral/tutorials/134/5/
  http://www.tomshardware.com/guides/overclocking/
- KryoTech CPU散热器的细节，参见：
  http://www.tomshardware.com/1999/10/20/kryoten/
- 请尝试通读Build your own PC所描述的各个步骤：
  http://www.pcmech.com/
- 这些网址可以通过本书的配套网站访问：
  http://www.pearsoned.co.uk/williams

第4章　构成计算机的逻辑电路：控制单元

控制单元几乎要占CPU一半的面积，是任何计算机中最重要的部分。它虽然是由逻辑门构建而成，控制单元的基本活动是对指令译码。我们可以通过研究较简单的译码器电路来了解它的活动。在日常生活中，时常能够看到它的身影，如交通灯和洗衣机。因而，译码器和编码器电路具有很广的适用性。本章还根据对控制单元的影响，讨论了新的RISC与老式的CISC CPU之间的差异。在解释逻辑动作时，我们使用真值表。在设计软件时，如果输入条件比较复杂，也可以采用真值表。

4.1　电子积木和逻辑电路：模块化器件的优点

这一节不是想让您变成电子工程师！但是，学习一些基本的组合逻辑（combinatorial logic），对理解计算机的运作以及如何编程控制它们会有所帮助。现今，所有的计算机都是由VLSI（Very Large-Scale Integration，超大规模集成电路）逻辑电路构成，这种技术能够将数以千计的简单逻辑门集成到面积大约25平方毫米那么小的硅晶片上。

通过使用照相缩版技术，现在，晶片表面的导线轨迹已小于0.1微米宽，已经超出光学分辨率的极限，可见蓝光的波长为0.45微米，用于生产的掩模必须使用波长更短的X射线或电子束来制造。使用VLSI技术制造电路提高了它们的可靠性，加快了电路的运作，并且降低了单位生产成本。第一个成功的数字**集成电路**（integrated circuit）产品系列是德州仪器公司生产的7400系列**TTL**（晶体管—晶体管逻辑电路）。参见图4-1中的示意图。

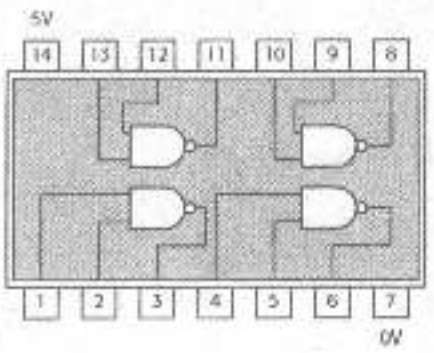


图4-1　四元组二输入端NAND SN7400，TTL

由于创立伊始TTL就拥有十分清晰的设计规则，就使得其他制造厂商能够遵循这些规则进行开发。因此，多年来，不断扩大的TTL芯片产品线一直是大部分计算机开发工作的基础。相比之下，软件提供商却未能达到类似水平的标准化，常让人抱腕叹息。拥有一系列可信赖部件的优点是，硬件开发人员可以快速地切入到更为复杂的设计工作中。由于日益猖獗的软件盗版问题所带来的影响越来越严重，以至于有时人们更倾向于采用硬件设计，而非采用方便得多的软件实现，以保护产品的版权。由于VHLSL（Very High Level Specification Language，高级描述语言）变得越来越流行，将来在硬件和软件间切换设计将会变得更为常见。系统设计人员将会使用VHLSL将他们的想法陈述出来，然后决定是通过硬件还是软件来实现。描述该系统的"程序"可以转换成常规的HLL，比如C，执行传统的软件编译，也可以转换成VHDL（参见1.3节），生成VLSI产品的掩模。

4.2　基本逻辑门

虽然有许多更为正式的逻辑电路设计方法，如卡诺图，但仅仅使用常识和类似于玩积木时所使用的建造方法，也能够成功构建简单的数字电路，此时，只需了解简单的真值表（Truth Table）就够了。不需要复杂的设计方法论，软件工程师或计算机程序员就可以熟悉基本的原理，学习实用的技能。

参见图4-2，首先从一行行地阅读真值表做起。左边是所有可能的输入组合，右边为对应的输出值。这样，对于二输入端的**AND**门（与门），需要四行来罗列可能的组合：00、01、10、11。大

多数情况下，按照二进制顺序依次写下去比较好，因为这样无疑有助于检查是否将所有可能的位模式都列了出来。

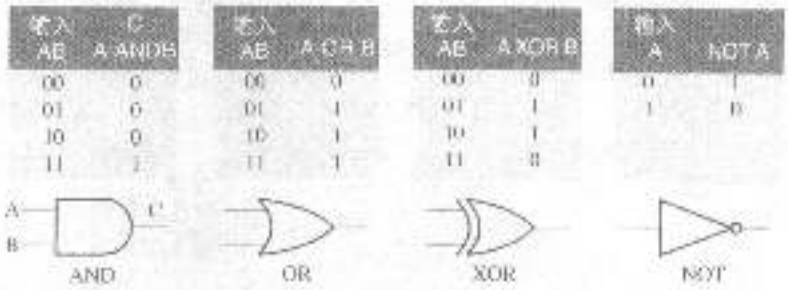


图4-2 基本数字逻辑门电路

通过简单的内部转换电路，就可以使AND和OR逻辑电路具有相同的功能。将台灯的电源线插入到墙上的插座时，就是AND逻辑。所有的开关必须都闭合，灯才能亮。**OR**功能用在汽车门开关的提示灯上：如果一扇或多扇门处于打开状态，则灯点亮。而顶部和底部都有开关的楼梯灯则类似于**XOR**（异或）：要想让灯亮，开关必须处于不同的状态。如果楼梯顶底之间的接线发生交错，其动作则类似**XNOR**门（同或门）。我们可以将XOR门（异或门）看做差异检测器，将AND门看做是全1检测器，将OR门看做全0检测器。通过NOT门（反相门）的合理运用，我们可以检测任何输入模式，见图4-3。

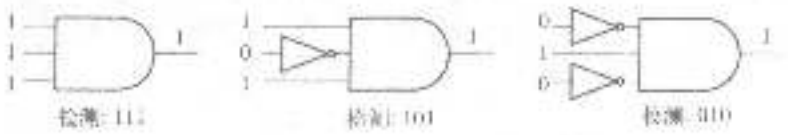


图4-3 使用AND门检测指定的位模式

尽管我们所画的AND门都只有两个或三个输入，但从理论上讲，任意数目的输入都是可能的。八端输入的AND门常常被用来构建地址译码电路。而且，通过将两个AND门连接到第三个AND门，您能够合成更宽版本的AND门。单个双端输入AND门最简单的实际应用就是控制数据流。在图4-4中，**数据阀门**（data valve）允许一个电路控制另一个电路的运作，数据输入的真值可以是0或1。逻辑与的关系也可以写成：O = D AND X，或者O = D · X。

DX	输出
00	0
01	0
10	0
11	1



图4-4 数据流控制电路

这种情况下，真值表为另一种形式，如下所示：

D	X	输出
0	0	0
0	1	0

在上面的表中，其中输入的逻辑值并不重要，我们用符号“d”来表示，它既可以为1也可以是0。

随后，它可能会出现在输出栏、表达式中，或者单独出现（如本例所示）。

另一种构建数字电路的方法越来越普遍，这种方法使用存储在存储芯片上的**查找表**（look-up table，LUT）。对于可编程逻辑单元来说，这样会非常方便，因为不同的逻辑功能，可以简单地通过在表中存储不同的模式来实现。

3.1节已经突出了连接各个部件的总线对于现代计算机系统的核心重要性。如果仅仅引入信号总线高速公路，则会使简单、易于扩展的数字系统复杂化，而且，这种互连方案连接前面刚提到过的逻辑门时，存在一个严重的问题。简单地说，它们会逐渐混合，这个问题产生的根源在于用来控制它们输出的晶体管电路。如果将多个简单门电路的输出连接到总线中某根导线上，当有些门电路输出逻辑0，而其他的门电路力图将同一总线导线的电平变成逻辑1时，灾难就会发生。这场输出之间的斗争实际上最终会以所有输出逻辑0的门电路获胜而告终，因为它们的晶体管更有力。这个严重问题可以通过引入特殊的**三态门**（见图4-5）驱动总线来解决。三态驱动电路提供三种输出：0、1和关闭；要做到这一点，我们需要提供额外的控制线，即**输出允许**（Output Enable，OE，或称使能），使输出在关闭和两种开启状态之间切换，这样就解决了总线问题。

输入	输出允许	输出
d	0	OFF
d	1	d

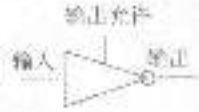


图4-5　三态驱动电路

4.3　真值表和多路复用器：简单但有效的设计工具

真值表是强大的图解工具，它可以用来描述任何逻辑系统或逻辑变换的输入输出关系。不管是软件例程还是逻辑电路，都可以根据真值表进行设计，根据给定的输入模式产生正确的输出值。

“数据阀门”的一种增强型应用是数据选择器，或称**多路复用器**（multiplexer）。我们在后面还会用到它，在此给出这个电路（见图4-6），仅为介绍真值表之用。

控制 WXYZ	数据 ABCD	输出
0001	abcd	a
0010	abcd	b
0100	abcd	c
1000	abcd	d

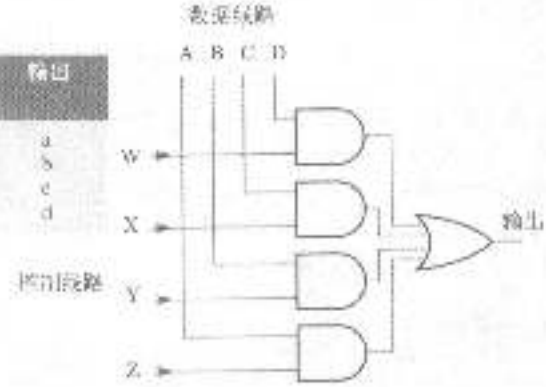


图4-6　数据选择器电路

注意真值表的书写，此处并没有指定数据输入（A、B、C、D）为1或为0，因为在该应用中，实际的值并不重要，我们所需要知道的，只是哪个数据输入出现在输出中。如果以方程式的形式来表述，该逻辑关系看起来很复杂：

O = (A AND Z) OR (B AND Y) OR (C AND X) OR (D AND W)

数据选择器电路允许控制线路（W、X、Y、X）将与之相关联的数据输入传送到输出，但遗憾的是，如果同一时间多条控制线试图选择数据，则这个电路会失败。如果可以分别将四个输入编做0、1、2、3，然后使用数字选择其个数据流，则要方便得多。这要用到译码电路。

所需译码器的真值表在图4-7中给出。要注意，在输出的每一行和列中只有一个1，这样实现起

来要容易一些。我们可以用这个电路作为前述数据选择器的一个部件，以制成更好的数据多路复用器。如图4-8所示，在X、Y控制输入上给出的由两个二进制位构成的数字，可以选择将四条数据线路（A、B、C、D）中的任一条传送到输出。通过这个例子，我们还将引入用逻辑门实现真值表的一种方法——暴力法（brute force）。现在，我们不再仅仅是检查输入中活动的1，我们还检查不活动的0。改进后的数据选择器可以写成下面的逻辑方程式：

选择控制 YX	线选输出 d c b a
00	0001
01	0010
10	0100
11	1000

图4-7 二线译码器

O = (A AND (X AND Y) OR (B AND (X AND $\overline{Y}$)) OR
(C AND ($\overline{X}$ AND Y)) OR (D AND ($\overline{X}$ AND $\overline{Y}$))

符号上面的短线表示NOT，一种逻辑求反运算。尽管这个逻辑表达式看起来复杂，但实际上，它是由四个相似的项组成。每一项负责数据选择器真值表中的一行。通过这种方式，在任何时候，A、B、C或D数据线路中只会有一个被选中。不可能发生不正确的选择。

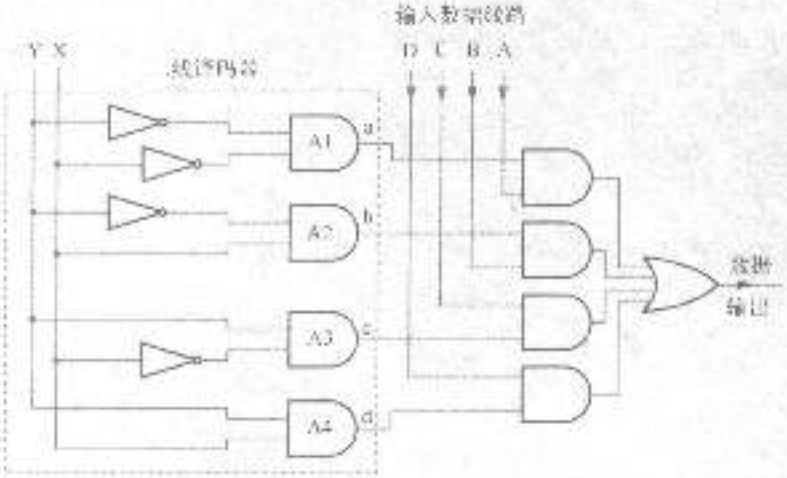


图4-8 使用二线译码器的数据多路复用器

真值表的一种替代方式是**信号时序图**（Signal Timing Diagram）。在第3章中显示总线信号时，我们已经使用过它。一些硬件调试工具以这种格式显示它们的数据。因此，最好能够理解这种显示方式。通过图4-9中的输出轨迹，可以知道任何时候它正在传送的是哪个输入数据流。从中我们可以得出多路复用器的行为方式：由来自译码器的控制线路a～d选取输入流A～D之一，同一时间只有一条输入流被选中。

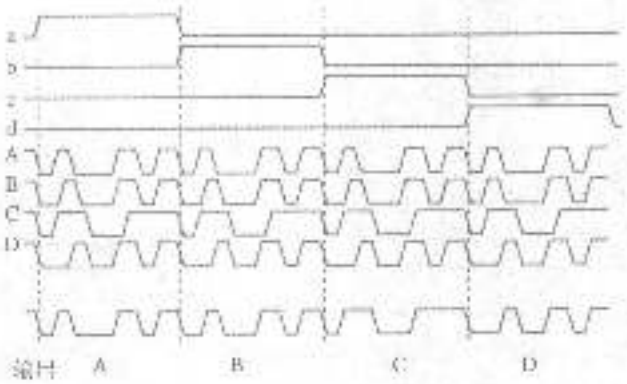


图4-9 四线数据选择器的信号时序图

在设计真值表的逻辑实现时，有时可以忽略那些对正在考虑的输出列不产生任何输出的行，以降低最终的复杂度。不需要专门去抑制那些不产生输出的情况。

4.4　可编程逻辑器件：可重新配置的逻辑芯片

PLD（Programmable Logic Device，可编程逻辑器件）可以帮助我们实现逻辑电路。它提供逻辑门资源，我们可以根据真值表的要求，使用它构建硬件逻辑电路。一个PLD可以代替许多分立的逻辑器件。PLD有几种不同类型的应用，但就基本功能来说十分类似。PLD是逻辑芯片，用户可以对它重新配置。制造厂商在芯片的内部提供很多通路，用户可以通过熔断内部的熔断器，清除不需要的通路，只将需要的通路保留下来。另一种技术使用存储在RAM或EEPROM（Electrically Erasable Programmable Read-Only Memory，电擦写可编程只读存储器）存储器的位组合，启用所需的路径。通过这种方案，用户可以多次重新配置同一芯片，我们称之为EPLD（Erasable Programmable Logic Device，可擦除型可编程逻辑器件）。

可编程逻辑阵列（Programmable Logic Array，PLA）提供多组AND-OR逻辑门，非常适合于用来实现真值表。PLA的输入（参见图4-12），以及相关的反向对，都通过可熔断链接连接到AND门上。同样，到OR门的连接同样可熔断。因而我们可以通过“熔解”不需要的线路，只保留那些需要的线路，来实现所需的电路。由于有这些可熔断链接，使得我们可以将AND门连接到任何输入，可以将OR门连接到任何AND门。通过这种方式，我们可以为每个输出准备一个电路（见图4-10），来表达“乘积之和”项：

$$O1 = (\bar{i}_3 \text{ AND } i_2 \text{ AND } i_1) \text{ OR } (\bar{i}_2 \text{ AND } i_1 \text{ AND } \bar{i}_0) \text{ OR } (i_3 \text{ AND } \bar{i}_2 \text{ AND } i_0 \text{ AND } \bar{i}_1)$$

“乘积之和”一词来源于表达式所采用的数学逻辑符号：

$$O1 = (\bar{i}_3 \cdot i_2 \cdot i_1) + (i_1 \cdot i_0 \cdot \bar{i}_0) + (i_3 \cdot \bar{i}_2 \cdot i_0 \cdot \bar{i}_1)$$

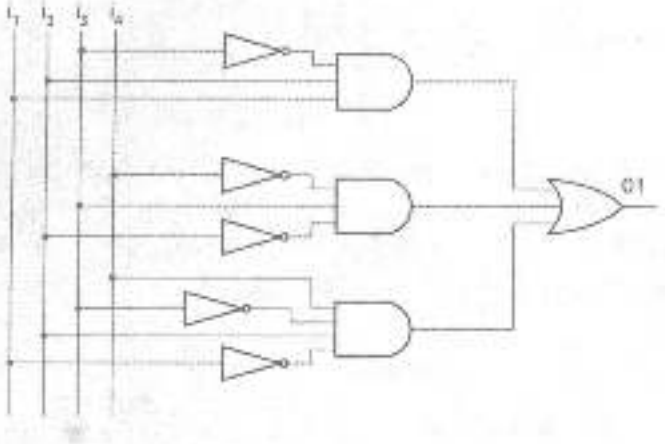


图4-10　“乘积之和”逻辑公式的实现

任何组合逻辑功能都能使用AND、OR和NOT门来实现。实际上，如果需要，只使用NAND就能够完成所有的事情。这个门（见图4-11）的行为就如同AND门后立即跟上一个NOT门。重要的是要记住，反相发生在AND之后，或许称之为ANDN门更好一些。发音上的混淆不可避免地会导致设计上的错误。第5.1节将会就逻辑门的互换性进行论述。

X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0



图4-11　二输入的NAND门及其真值表

在设计数字逻辑电路时，还有可能会用到门最小化技

术。但在实际开发中鲜有应用，其他的约束，比如不同类型门电路的成本、电力消耗、信号通过延迟（signal-through-delay）以及是否能够买到、供应情况等，更为紧迫。对于我们的目的来说，不需要对电路进行合理化或最小化。

PLA可以作为译码器。图4-12给出的设备即为一例。此处，AND门组成的矩阵直接连接到输入上，或连接到经过反相后的输入上。从真值表的角度，可以将其描述为“行检测器”。之后，这些AND门的输出连接到由OR门组成的第二个矩阵（它们的行为类似于“列生成器”）。所有的连接都可以由程序员来更改，这使得这个设备十分灵活，但设计起来却十分困难。它适合于实现所有类型的真值表。

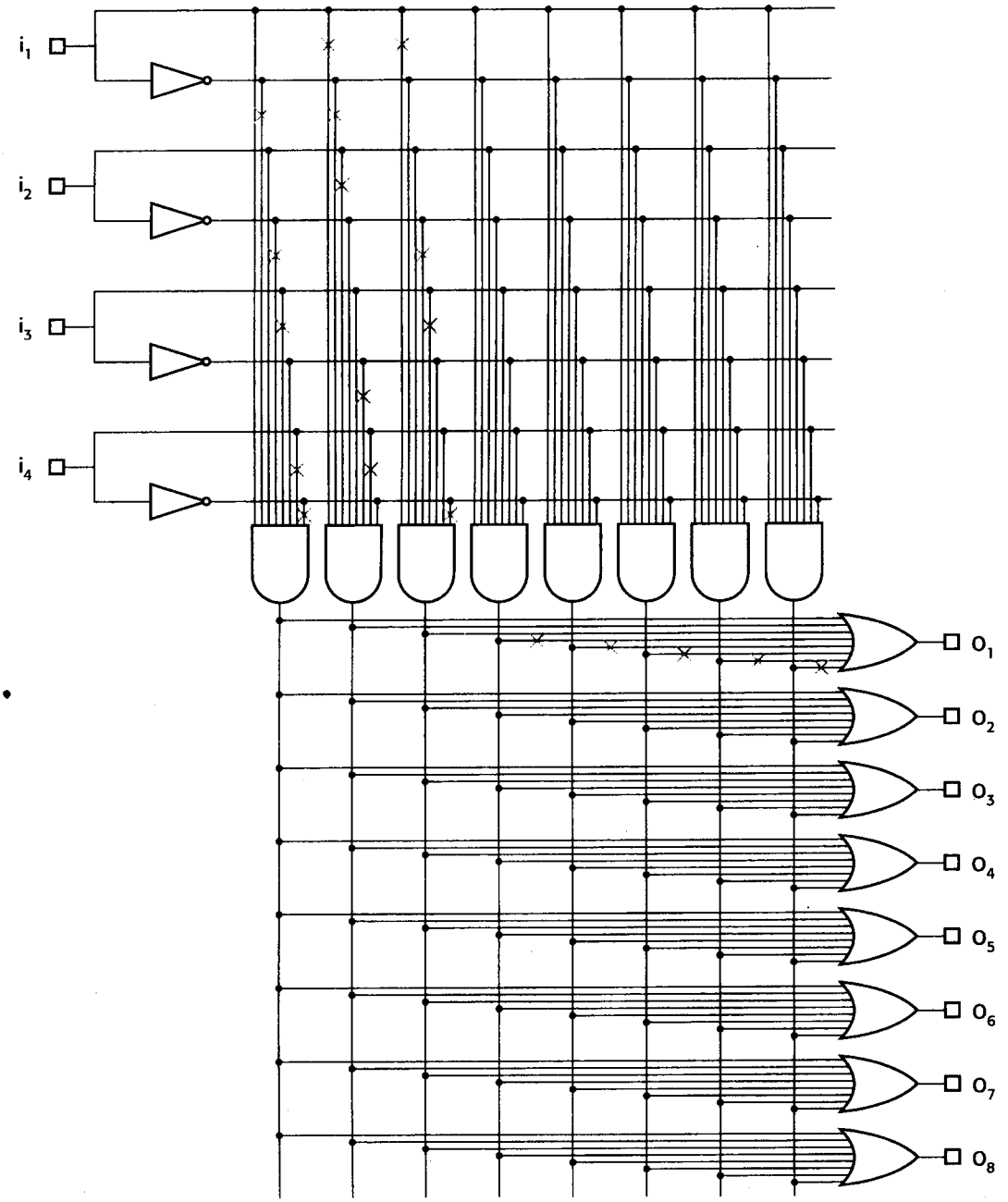


图4-12　实现图4-10中电路的可编程逻辑阵列

另一种实现“译码”的方法采用**PROM**（Programmable Read-Only Memory，可编程只读存储设备）。这种器件内部保存有供查找之用的表（LUT）。您可能会觉得惊奇，因为它与存储器存储程序和数据的常规角色有很大差别。不管怎样，我们也可以将PROM看做是可编程逻辑器件。类似于PLA中输入按照固定模式连接到AND门，PROM将输入线路传入的地址送到译码电路，由译码电路选择单独的存储单元。这一功能可以通过使用AND门检测每种二进制地址来做到。PROM的输出由事先存入存储单元内的值来提供。

另一种不同的逻辑设备是**PAL**（Programmable Array Logic，可编程阵列逻辑），通过对AND阵列的编程，可提供到OR阵列的固定连接。使用这样的设备以及相关的少数附加电路，就可以建立一个有限状态控制器。

4.5　交通灯控制器：无法避免

在向学生介绍特定的科目或理论时，有几个例子和系统常常为教师所用。逻辑课程总是会介绍交通灯控制器，这已经司空见惯。下面我们就来看一下这个例子。考虑图4-13中的真值表，其中列出了英国交叉路口显示交通灯光的顺序：红、红/黄、绿、黄、红。

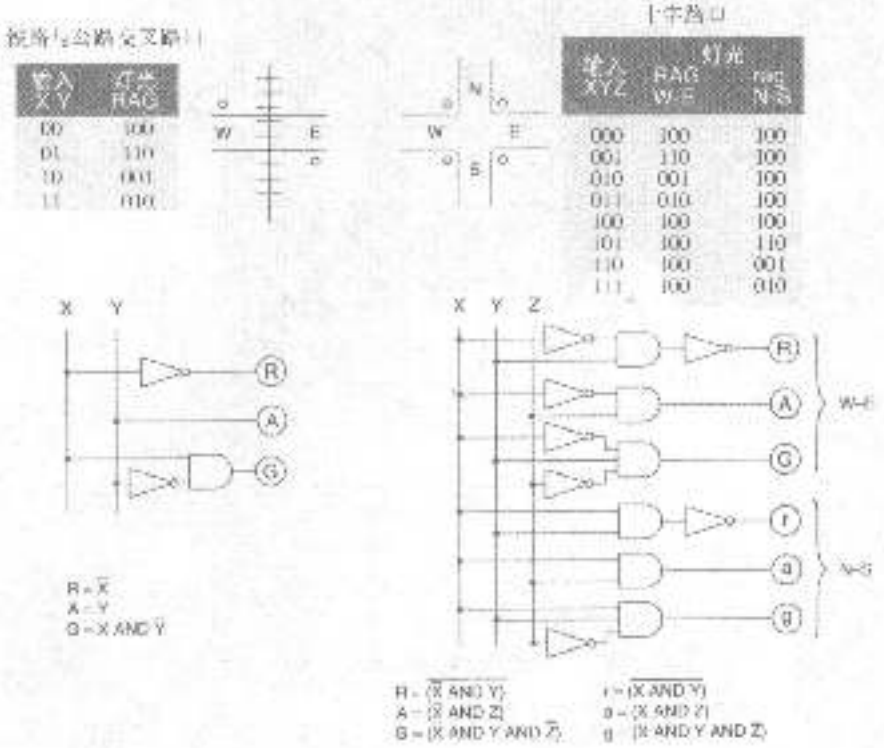


图4-13　交通灯控制电路

在阅读完本章后，你就会发现，交通灯控制器不过是译码器电路（与CPU的控制单元紧密相关）的另一个应用实例而已。很长时间里，商品化的交通灯控制器都使用电子-机械继电器来构造，因为人们认为它们比电子部件更可靠。不过，微处理器以及在其上运行的错误检查软件，已经得到了道路规划者的热情接受，因为它们能够提供更为复杂的控制系列。越来越多的城市中心安装了综合城市交通管理系统，它的工作依赖于交通灯控制器和中央控制计算机之间的通信链接。简单的数字逻辑电路根本不可能提供这种功能。

4.6 根据真值表实现电路：一些实用提示

在填写真值表时，必须首先将所有可能的输入组合列出来，并输入到左边的栏中。按照数字顺序常常会方便些。之后，每一行右边的栏必须填上正确的输出模式。在检查完真值表的每一行之后，就可以开始设计逻辑电路。具体使用的技术依具体情况而定，此处列出了几种捷径。

1. 完全相同的列

依次检查每一栏，当输出栏与输入栏等同时，我们可以采取一种简单的实现方式。只需直接将输入连接到输出：不需要任何逻辑电路。交叉道口的黄灯输出和Y输入即属此类。使用代数公式可以表示为：

A = Y

2.差不多相同的栏

输出可以由某些输入构成的简单逻辑函数生成。交叉道口的红灯是输入X的简单反相：

$R = \overline{X}$

3. 单独的列

如果输出栏只有一个“1”，可以使用AND门来检测输入行的模式。所有的绿灯输出皆属此类：

$G = X \text{ AND } \overline{Y}$

十字路口为：

$g = X \text{ AND } Y \text{ AND } \overline{Z}$

4. 反相的单独列

当一个输出栏只有单个“0”时，使用AND门来检测这种输入的行模式，后接NOT反相器。交叉路口的红灯很接近这种情况：只有两行需要加以考虑，X = 0和Y = 1。同样，Z可以忽略：

$R = \overline{(\overline{X} \text{ AND } Y)}$

5. 标准模式

有时，可以使用现成的逻辑电路，仅仅做出稍许的改动。有些情况下，XOR可用做“差异检测器”，在本章后面洗衣机中的马达控制输出中，我们能够看到这种应用。

马达 = (X XOR Y) AND Z

- 6. 消去

在考虑十字路口的真值表时，可以采取一些稍微简捷的方法。有两行包含X = 1和Z = 1。Y可以是1或0，并且看起来当X和Z都是1时，Y值是什么对输出没有任何影响。这种情况下就可以将它忽略：

a = X AND Z

$A = \overline{X} \text{ AND } Z$

7. 乘积之和（Sum-of-products）

在这些捷径无效时，我们就只好使用暴力法了。在4.3节中曾用到过这种方法，依次处理每个输出栏，将每个提供“1”的行标记出来。然后，为每个标记出来的行建立一个AND门模式检测器，如果要检测“0”，则不要忘记在输入上使用NOT门。现在，为每个输出栏分配一个OR门。仅当AND门的输出为1时，它们才能成为输入。下面是针对交叉道口问题的解决方法：

$R = (\overline{X} \text{ AND } \overline{Y}) \text{ OR } (\overline{X} \text{ AND } Y)$

$A = (\overline{X} \text{ AND } Y) \text{ OR } (\overline{X} \text{ AND } Y)$

$G = X \text{ AND } \overline{Y}$

值得注意的是，我们为交通灯设计的控制电路，只是在计算机系统中应用十分广泛的译码器的一种特殊的经过修改的版本。

4.7　译码器逻辑：控制单元及存储器的根本所在

译码器的根本目的是识别编码，并激活对应的动作。这个动作可能看起来功能有限，起初只能改变特定线路的逻辑电平，但这个简单的功能的确很实用。实际上，不管对于存储器还是对于CPU的控制单元，它都是基本的。译码器的示意图是一个方框，输出多于输入。重要的是要理解，严格意义上的译码器（如图4-14中的74F138）在一个时间只选择单个输出线。所选择的输出线路依赖于输入的编号。

输入		输出
使能 G_1G_2	选择 CBA	Y 01234567
X1	XXX	11111111
0X	XXX	11111111
10	000	01111111
10	001	10111111
10	010	11011111
10	011	11101111
10	100	11110111
10	101	11111011
10	110	11111101
10	111	11111110

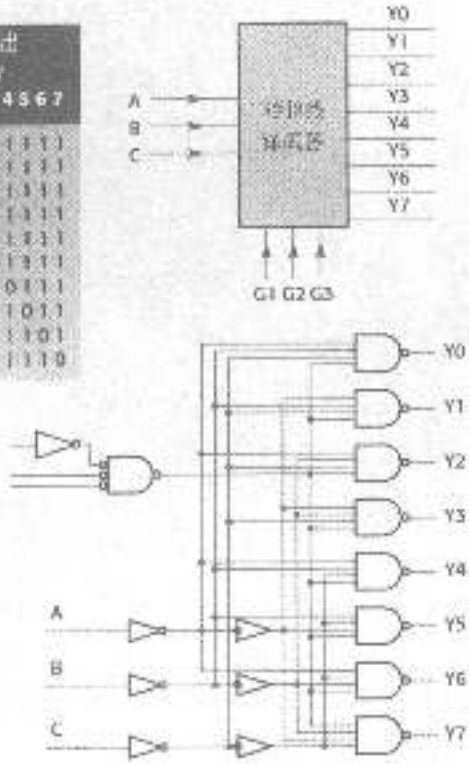


图4-14　SN74F138 3到8线译码器：真值表、示意图和逻辑电路

使用编号进行“寻址”或“选择”的能力，是信息理论的重要概念。尽管它看起来不过是一种降低传输线路数量的小技巧，但是，它却和一个严肃得多的思想有关——符号表示（symbolic representation）。打印机接收7位的ASCII码，识别出它，并选择点阵模式（也就是我们后来看到的字母）。我们的大脑又进一步地将它解释为单词的一部分，表示某种思想。也许思考不过是从一系列可能的选项中选择，那译码是不是意识的第一步？也可能不是——更多可能是无意识的。

如前一章所述，真值表中每个输出栏的逻辑项都可以直接使用暴力法实现，我们接下来就会阐述这种方法。图4-14中的译码器有三个输入和八个输出，能够加以修改实现二进制到八段式显示的转换器。真值表在4-15中给出。这类装置常常用来控制廉价设备前置面板的显示。该单元由7个发光二极管（Light-Emitting Diode，LED）构建而成。需要注意的是，在输出栏中0的数目小于1的数目时，我们如何转而采用取反（OFF）条件。这样做降低了逻辑表达式的长度，因为涉及的行会减少。特别地，指定c比指定c容易得多！

所有LED段对应的结果是：

$$\bar{a} = (W \text{ AND } X \text{ AND } Y \text{ AND } Z) \text{ OR } (\overline{W} \text{ AND } X \text{ AND } \overline{Y} \text{ AND } \overline{Z}) \text{ OR } (\overline{W} \text{ AND } X \text{ AND } Y \text{ AND } Z)$$

$\overline{5}$ = (W AND X AND $\overline{Y}$ AND Z) OR ($\overline{W}$ AND X AND Y AND Z)

c = $\overline{W}$ AND $\overline{X}$ AND Y AND $\overline{Z}$

d = (W AND X AND Y AND Z) OR ($\overline{W}$ AND X AND $\overline{Y}$ AND $\overline{Z}$) OR (W AND X AND Y AND Z)

$\overline{e}$ = ($\overline{W}$ AND $\overline{X}$ AND $\overline{Y}$ AND $\overline{Z}$) OR (W AND X AND Y AND Z) OR ($\overline{W}$ AND X AND Y AND Z) OR (W AND X AND Y AND Z)

f = (W AND X AND Y AND Z) OR (W AND X AND Y AND Z) OR (W AND X AND Y AND Z) OR (W AND X AND Y AND Z)

g = (W AND $\overline{X}$ AND $\overline{Y}$ AND $\overline{Z}$) OR ($\overline{W}$ AND $\overline{X}$ AND Y AND Z) OR (W AND X AND Y AND Z)

使用这些逻辑表达式，我们能够设计一个7段式显示驱动电路，它接受4位的数字，通过点亮相应的一系列发光二极管，来显示等价的十进数。图4-16给出"a"段的驱动线路。其他六个电路的设计，留待下雨天再慢慢解决！

输入	二极管
W X Y Z	a b c d e f g
0 0 0 0	1 1 1 1 1 1 0
0 0 0 1	0 1 1 0 0 0 0
0 0 1 0	1 1 0 1 1 0 1
0 0 1 1	1 1 1 1 0 0 1
0 1 0 0	0 1 1 0 0 1 1
0 1 0 1	1 0 1 1 0 1 1
0 1 1 0	0 0 1 1 1 1 1
0 1 1 1	1 1 1 0 0 0 0
1 0 0 0	1 1 1 1 1 1 1
1 0 0 1	1 1 1 1 0 1 1

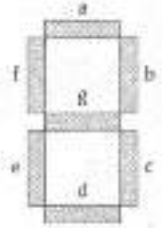


图4-15 7段式转换器

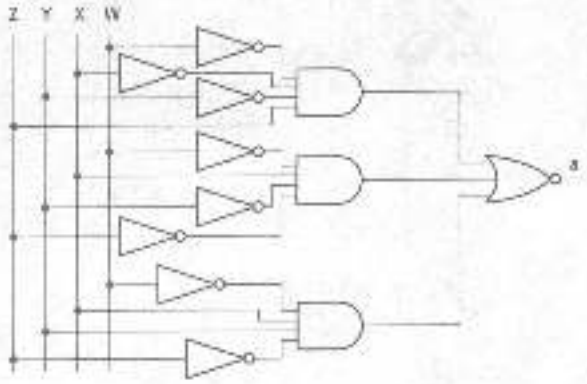


图4-16 7段LED驱动电路中"a"段的逻辑电路

4.8 CPU控制单元："核心"

控制单元（CU）是计算机最核心的部件，它担当着乐队指挥的角色：设置时序，在需要时发起各种活动，监视系统的全面运作，以及裁决冲突。它的首要职责还包括从主存储器中取指令，每次一条，并执行译码后的指令。它负责控制和协调计算机的各种活动，从CU引出启动和停止各种功能的控制线路，读存储器、写存储器、数据输入和输出等。除了这些外部活动以外，控制单元还协调CPU自身内部的复杂时序。实际上，基本的读取-执行周期就是由控制单元发起和控制的。

控制单元中被称做**译码器**（decoder）的部分负责选取由指令寄存器（IR）中保存的当前指令所要求的特定活动（ADD、SUB、MOV）。CU译码器必须从硬件设计者提供的机器指令系统中选择一个活动。计算机的指令常常被再次划分成几个（5的倍数）微周期，以简化设计过程。因此，任何指令都被再次划分成微指令，微指令依次执行，即可以完成完整的机器指令。在构建译码器时，一般有两种技术可供计算机设计人员选择——硬连线式的译码或微程序译码。

4.9 洗衣机控制器：简单的CU

为了更好地了解CU的运作，下面我们来考查一下简单的洗衣机控制器是如何工作的。我们的洗衣机只遵照一套简单的程序动作：就绪、注水、加热、洗、排水、漂洗、排水、旋转（脱水），回到就绪，参见图4-17。

洗衣周期中的八个阶段之间的先后次序，是由一个小型的、慢速转动的马达来执行的。它驱动固定在马达转动轴上的一组三个凸轮旋转。该马达必须转动得十分缓慢，因为完整的洗衣周期在马达的一周360度旋转中完成，实际上将会花大约30分钟的时间。每个凸轮的侧面由一个微动开关来检测（或"触踪"）。微动开关只能表示1和0。因而，三个微动开关共同提供一个三位数（或指令），可以进行译码和执行。实际上，我们是在构建一个3位CU来控制洗衣机的动作。在此，我们不考虑温度传感器、门锁、水位开关和其他附加的安全特性。

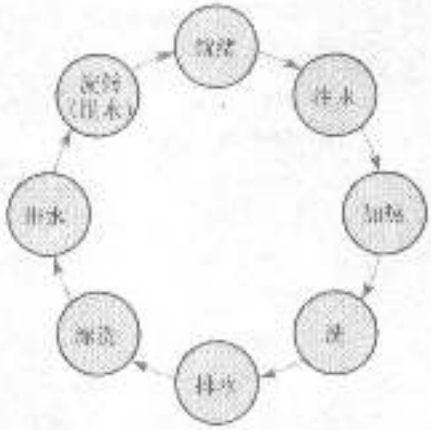


图4-17　洗衣机的状态转换图

如果洗衣周期需要30分钟，且定序马达必须以0.55 mHz旋转（毫赫兹、每秒千分之一转），如下面的算法所示：

$$1\text{转每}\frac{1}{2}\text{小时}\rightarrow 2\text{转/小时}\rightarrow\frac{2}{3600}\text{转/秒}\rightarrow\frac{1}{1800}\text{Hz}\rightarrow\frac{1000}{1800}\text{mHz}\rightarrow 0.55\text{ mHz}$$

洗衣机中含有一些必须加以控制的设备：水阀门、加热器、排水泵、马达，马达有两种转速，以便进行洗涤和旋转。我们的目标是设计一些逻辑电路，接收追随凸轮的微动开关产生的三位指令，将它译码为分别控制这四个设备的控制信号。图4-18中给出了这种机制运作的真值表。

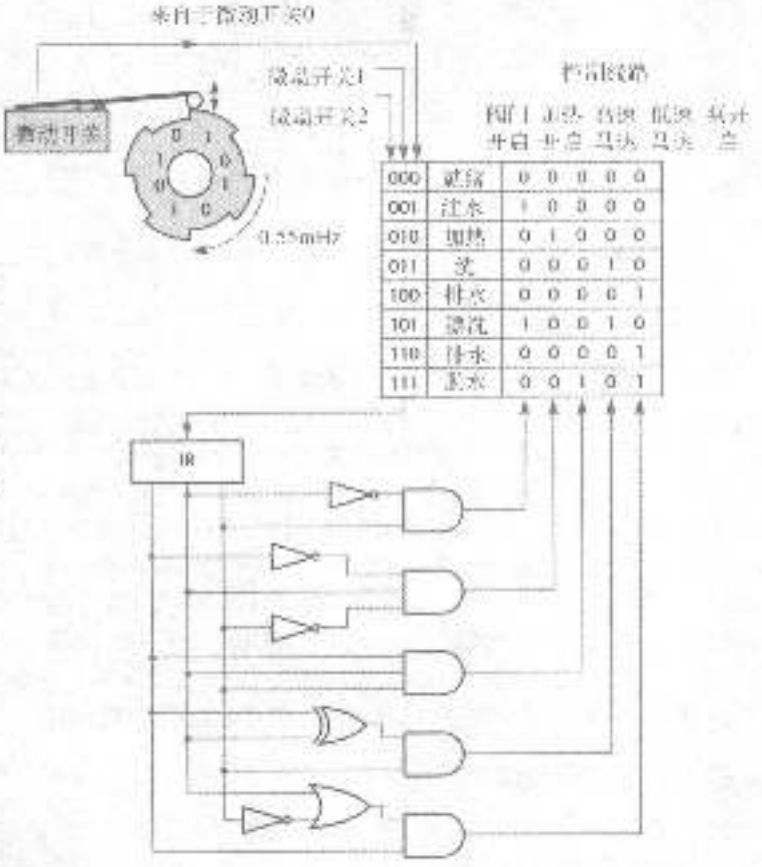


图4-18　洗衣机控制器的示意图

3位指令的另一种译码方式，是使用存储器和地址寄存器或程序计数器（Program Counter，PC）。地址寄存器中的数字用来访问存储器中保存着控制位模式的存储单元。这基本上是微码CPU所使用的方法。实际上，这个指令用来形成一个地址，用以访问存储控制字的存储单元。洗衣机需要8个字长、5位宽的内存。在洗衣周期中六个阶段中的每个阶段，来自于声轮微动开关的3位指令会发送到该存储器，从那里得出5位控制字来选取对应的操作。图4-19说明了这种方案。

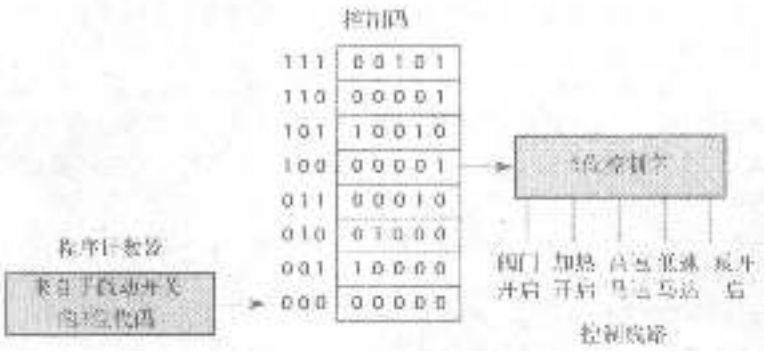


图4-19 使用存储器译码

这种使用存储器的译码方法比离散逻辑更易于实现，而且它的功能还能够进一步增强，支持条件跳转和循环。通过将控制字加宽，使之能够保存可以直接载入到PC中的地址字段（注意，此处的PC不是指个人计算机，而是Program Counter的缩写，译者注），就可以实现跳转，转到其他存储单元。图4-20给出了这种方案。为了能检查条件，实现条件跳转功能，我们需要加入其他字段来容纳状态标志允许位。根据这些标志位，我们能够有选择地检查指定的微动开关，看它是表示正确水位的输入还是表示洗涤温度。在我们的模型中，译码器只能选择一个开关，如果该开关是TRUE，则新的地址将被载入到程序计数器中。这就是条件跳转。

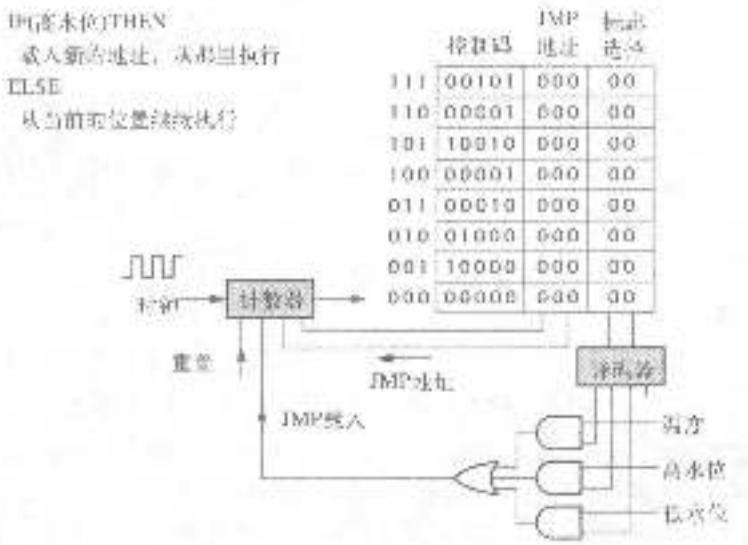


图4-20 具有条件跳转功能的控制单元示意图

遗憾的是，本例未提供处理子例程的功能。因为我们尚无法存储和重新载入返回地址。这是我们在第8章中将会处理的问题。但是，毕竟我们只是在分析一个洗衣机！这种类推或许已经逐渐失去控制，因此，我们最好不要再继续深入下去。然而，它的确可以让我们更好地了解数字机器是如何"理解程序"的。

4.10　RISC与CISC译码：使计算机的处理速度更快

最近，计算机设计人员团体内存在一些分歧：一些人提倡继续遵循VAX和68000构架，尽最大可能提供更丰富、更强大的指令集（复杂指令集系统，CISC）。另外一些人则制造RISC（reduced instruction set computer，精简指令集计算机）机器，比如Sun的SPARC和Power PC。它们的指令较少，相对简单，但运行速度要比复杂的CISC快得多。通过CU译码程序指令的方式，就能够很清楚地了解这种差异。第21章中将更完整地展示RISC的理念，但在介绍译码技术的章节中，只介绍差异就够了，因为这两种方法差异实在太大。

硬件实现的译码。如图4-18洗衣机中使用的译码方法，即为RISC处理器使用的技术，见图4-21。尽管仅仅从示意图看，它并不比前述的交通灯控制器复杂多少，但它涉及到使用十分复杂的译码器电路执行“多选一”活动。这常常被称为使用“随机逻辑”，当然，这些都经过十分精心的设计和测试。当前的指令代码从存储器中读出，载入到指令寄存器（IR）中。在那里，两个部分（操作码字段和操作数字段）被分别进行处理。操作码值——指定需要执行的动作（ADD、SUB或MOV），用做译码器阵列的部分输入。这个阵列的其他输入包括ALU状态标志（参见第5章）和当前机器周期编号。由于指令可以有多种不同的执行长度，机器周期计数器在每条指令的最后一个机器周期中被重置为0，为下一次读取－执行周期准备就绪。这种行为高度流水线化且十分高效。但不能处理十分复杂的指令，因为这种原因，程序会更长，同时执行所需的动作需要更多的指令。一般地，这对于HLL程序员来说没有什么不方便，因为一般是编译器来处理机器级别的指令。

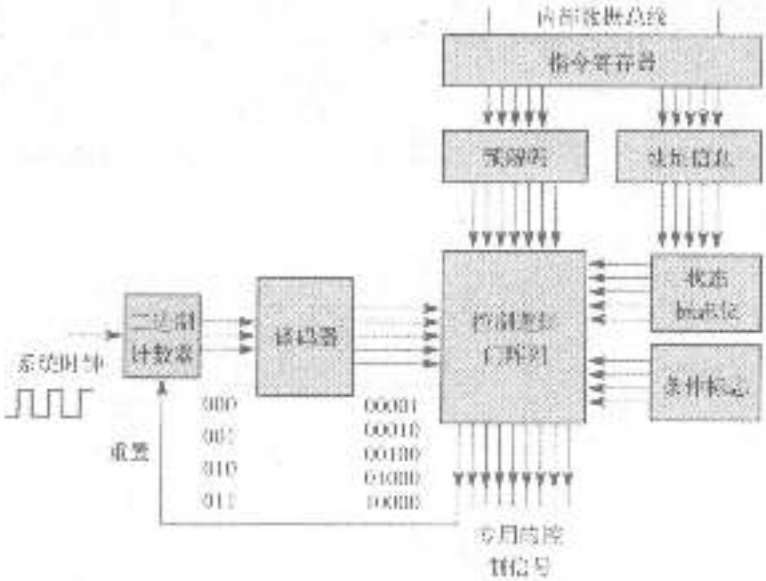


图4-21　硬件实现的（RISC）控制单元的示意图

另一种译码策略称为微码化，见图4-22，它用在CISC CPU中，它使用快速的“内部处理器”或超微型的计算机，来执行机器级别的指令。图4-19中是使用这种方法的简单实例。在这个例子中，3位洗衣机代码从存储的控制信息中选取出下一个控制字，和硬件实现的方法一样，读取－执行周期被进一步划分成几个机器周期，每个周期完成动作的一部分。基本上通过访问快速微码存储器中由控制字构成的大型矩阵来完成。微码字较宽，常常为64～128位，一般分成几个字段，分别包含信号控制位和地址指示器。后者将接下来的访问定位到存储微码的地方。因此，我们能够在微码中实现循环和子例程调用。但实现起来可能需要较高的技巧。再强调一下，处理器在处理变长指令时会将机器周期计数器置0，为启动下一指令周期做好准备。

接下来就是在正确的时间向正确的线路发送一个信号，使某些事情发生。在每个微周期内，都会发出某个预置的控制信号，以实现微指令。

在微码构架下，控制信号的这些模式存储在能够快速访问的存储器中，我们称之为“控制存储器”或“微码存储器”。这个存储器虽然很短，但却很宽：常常64位宽、4 k位长。为计算机编写微码很难，而且很耗时（参见Kidder，1981）。开发中没有什么辅助手段，而且尚未形成广泛接受的标准。

Motocola 6800 16/32位微处理器就是微码驱动的处理器，IBM通过启用新的微码方案来执行高级工作站上使用的IBM 370指令集，也从中受益。PDP LSI-11也是重新微码化以直接执行Pascal P代码的。解释型语言Forth拥有一个专为它编写的微码化CPU，即RTX2000。许多PostScript激光打印机中都有一个微码化的AMD29000。经过一段时间的研发，Sun推出针对Java的高速硬件：microJava 701 CPU。但就现在看来，它的商业化还有待时日。

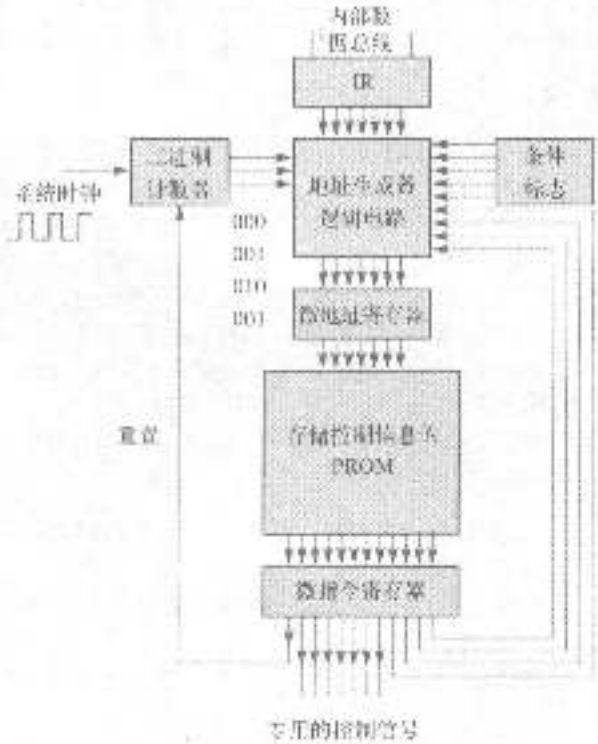


图4.22 微码（CISC）控制单元

一些计算机提供用户可以访问的RAM控制存储器，这样，用户就能够动态地更改指令集。对于常人来讲，这是一件危险的活动。使用这项功能，我们能够让IBM微计算机执行Sun SPARC程序，而不需要重新编译代码！

4.11 小结

- 计算机由数字逻辑电路构成。它们模块化的结构、精确的互连规则，有助于工程师设计复杂的电路。
- 早期的逻辑单元（称为TTL）已经被高度集成的、复杂的电路（VLSI）替代，这些电路中常常集成了数以百万计的开关晶体管。
- 基本的逻辑门是：AND、OR和NOT，对于许多应用来说，XOR也非常有用，另外，我们可以用NAND来合成基本的功能。
- 通过显示输入—输出之间的关系，真值表能够清楚地表达出逻辑电路的功能。
- 信号时序图还用在监控设备和逻辑模拟软件中，显示电路的输入和输出之间的关系。
- 现在已经存在直接提供电路重新配置功能的逻辑设备，这是原型制作和缺陷修复的福音。
- 译码器电路，如交通灯控制器所示，通过使用一个输入二进制数，来选择一个或几条输出线路。
- 逻辑电路可以直接从真值表信息来实现，有一些捷径可以加速这个过程。在构建电路时，一般不先考虑逻辑最小化。
- 译码器广泛地应用在CPU控制单元中。控制单元从存储器中逐条读入二进制指令，按照规定的时序，使用译码器来激活恰当的控制线，执行要求的动作。
- 除硬件实现的逻辑译码外（RISC），另一种译码方式使用CPU中的快速存储器来保存控制线路活动的位模式（CISC）。

实习作业

我们推荐的实习作业包括调查与控制单元相关的数字逻辑电路及译码器的主要示例。这项作业

可以使用试验电路板或计算机模拟程序来完成，但最好两种方式都试一下。

在开始实习之前，必须熟悉如何画真值表，以及如何使用这类表来设计逻辑电路。在这个阶段，还可以试验使用基本的逻辑门制造XOR元件，动手检查德·摩根定理（参见第5章）。

练习

01. 使用简单的门电路为安全锁设计一个电路。它有十个输入开关，因此，需要用户记住一个10位二进制数。为了存储“密码”，在安全箱内还需要另外的10个开关。
02. 写出2 → 4线译码器的真值表，然后，只使用简单门设计该电路。
03. 什么是数据解多路复用器？画出一个真值表，然后为四线解多路复用器设计一个逻辑电路。
10. PC键盘上有多少个键？它可能使用什么类型的电路来检测按键，并与CPU进行通信呢？
11. 如果你对数字逻辑感兴趣，请阅读卡诺图方法，并将它们与真值表进行比较。卡诺图有什么优势呢？
12. 如何使用256位的ROM存储器测试字节的奇偶值？
13. 哪一个有微码，是RISC还是CISC？
20. 如果A = 1，B = 0，C = 1，下面的C语句会得到什么值呢？

    $((\overline{A} \| \overline{B}) \&\& (A \| B) \&\& C$
21. C语言没有逻辑XOR运算符，即相对于&&和||的XOR等价物（应该是^^）并不存在。那么，如何在C语言中表达IF (A XOR B）呢？
22. 为什么有些集成电路封装会变得很热，而其他的则保持凉爽，即使在工作正常的时候？奔腾处理器要消耗多少瓦电能？
23. 相同的C语言程序，为CISC CPU重新编译时，最终生成的程序的大小比为RISC CPU编译出来的程序要长些还是短些？
30. 一个译码器电路有32条输出。它应该有多少输入？
31. 在对机器指令译码时，CPU的指令寄存器起到什么样的作用？
32. MC68000的指令代码为16位长。你是否对只有100条左右的操作码感到吃惊？
33. 一种光学扫描仪使用4 × 5的光学传感器矩阵。请问如何使用EEPROM（用做译码器）来区分数字0～9的图像？
100. C语言中，与硬件译码器等同的语句是什么？

课外读物

- Thewlis和Foxon（1983），逻辑电路的基本介绍。
- Heuring和Jordan（2004），RISC和CISC构架的简明对比；微程序控制的CU的介绍。
- Patterson和Hennessy（2004），在附录中介绍数字逻辑。
- Tanenbaum（2000），第3章，数字逻辑。
- 德州仪器公司（1984）：The TTL Data Book，也称为橙皮圣经。
- Altera-Xilinx和Lattice网站，可供逻辑电路设计者参考：

http://www.altera.com/html/mktg/isp.html

http://www.xilinx.com/

http://www.latticesemi.com/

- Digital Works是D.J. Barker在Teesside大学读书时写的电路模拟软件，在Windows上运行。这是一个免费软件，尽管最新版已经开始收费。试试下面的网址：

http://www.electronics-tutorials.com/downloads/downloads.htm

或者：

http://www.cems.uwe.ac.uk/~rwilliam/CSA_cw/digital_works_2.exe

- 这些网址可以通过本书的配套网站访问：

http://www.pearsoned.co.uk/williams

第5章　构成计算机的逻辑电路：算术逻辑单元

CPU的ALU部分负责执行当前运行的程序中指令所请求的整数运算及逻辑操作。如果将整数以2的补码的形式表示，则加法器电路可以同时处理加法和减法。整数乘法是重复的“移位-相加”运算。本章还介绍了IEEE浮点数格式，从中我们可以得出实数的优点。

5.1　德·摩根等价定律：逻辑互换性

画完真值表并仔细选择好AND、OR和NOT门的正确组合，构建好逻辑电路后，我们会发现，可以将AND替换成OR，OR替换成AND，这虽然会有些出人意料，但却恰好是**德·摩根等价定律**允许我们做的事情（见图5-1）。由于这看起来好像不太可能，所以，我建议大家最好做些实验，或者亲自推导一下，以获得直观的认识。科学上的一些观念确实和直觉相反。

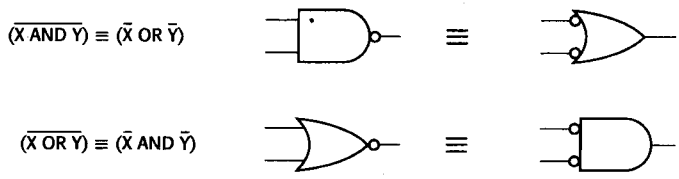


图5-1　德·摩根等价关系

很明显，程序员也会遇到类似的问题。比如，C语言代码中的逻辑条件：

```
while ( ! dog && ! cat )
{ plant_flowers( ) ;}
```

可以转换成：

```
while (! (dog || cat ))
{ plant_flowers( ) ;}
```

这样可以使循环的退出条件更清楚些，至少对部分人来说如此。

5.2　二进制加法：半加器、全加器、并行加法器

如第3章所述，**CPU**有两个主要的部件：控制单元（Control Unit，**CU**）和算术逻辑运算单元（Arithmetic and Logic Unit，**ALU**）。后者为计算机提供算术和逻辑处理能力，通过研究构成加法器的逻辑电路，我们能够对ALU有粗略的了解。简单的并行加法器和ALU中使用的加法器类似，可以用相对较少的基本逻辑门来构成。本章中，我们就以它为例，来探讨现实生活中真实ALU的工作方式。遗憾的是，由于各种各样的原因（当然，我们在这里并不需要考虑这些因素），实际的电路显然要复杂得多。

画出2位二进制加法的真值表后（如图5-2所示），我们就会发现，电路需要提供两个输出：和（Sum，S）以及进位（Carry，C）。单独查看进位输出与输入之间的关系，我们会发现它和AND门的模式相同。我们这样做是遵照4.6节给出的建议，在采用“暴力法”之前，先看看是否存在标准的逻辑模式。在处理逻辑问题时，最好将AND、OR和XOR的真值表放在手边（或牢记在心）。另外，和（SUM）的输出近乎等同于OR门——仅仅是输入均为1的那一行不正确。但通过在开始处使用OR门，在OR门的输出上插入第二个AND门，就能够容易地消除这个错误。该AND门仅在输入为两个1

的时候将和（SUM）输出置成0。这种方式与数据流控制电路（见图4.4）中控制线路的操作模式相同。至此，我们利用前面提及的多项技巧，设计完成了1位加法器电路的逻辑。加法器的真值表提醒我们，输出有两项——和及进位。

	加法
X Y	C S
0 0	0 0
0 1	0 1
1 0	0 1
1 1	1 0

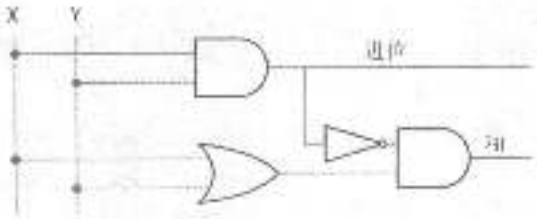


图5-2　加法器第1版

实现一种电路常常有多种方式。图5-3给出了另一种更为常见的加法器电路。它是用暴力法（见第4.6节）导出的：选取一个输出栏，找出输出为1的那些行，分别针对每一行设计AND门。给出这种输入模式，然后，将所有AND门的输出连接到一个OR门，OR门的输出即为栏的结果。

图5-4给出了另一种电路，它使用的门电路和前述电路不同，而且更少。

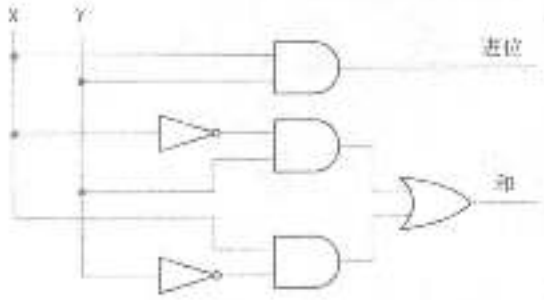


图5-3　加法器第2版

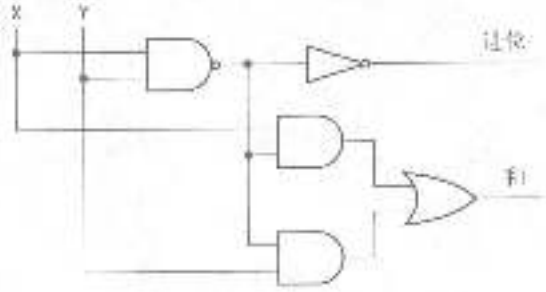


图5-4　加法器第3版

如果允许使用更复杂的XOR门，加法器甚至还可以更简单，如图5-5所示。

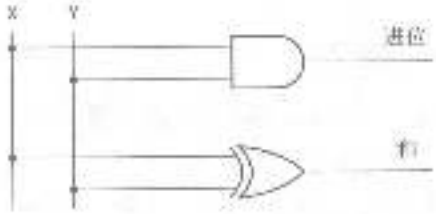


图5-5　加法器第4版（使用XOR门求和）

在考虑多位二进制数的加法时，这些优秀的加法器电路所存在的问题就显现出来。这同时也说明了为什么人们常常将它们称做半加法器。

$$\begin{array}{r} 1011 \\ 0111+ \\ \hline 10010 \end{array}$$

我们注意到，右边第一栏的两位相加时，"进1"会发生。这个进位需要加到第二栏中。很明显，我们需要处理来自于前一栏的进位值。因而，我们需要一个有三个输出端的加法器电路：3位全加法器，而不是2位半加法器。全加法器必须处理来自前一阶段的进位输入，同时还要处理自己的两

位；使用两个半加法器单元，就能够成功地组合出全加法器来，如图5-6所示。

X Y Z	C S
000	00
001	01
010	01
011	10
100	01
101	10
110	10
111	11

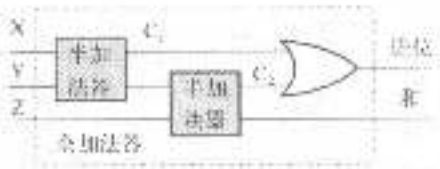


图5-6 全加法器

在图5-7中，四个全加法器依次排成一行。它们每个分别负责处理来自4位数中的两个位的加法。注意，图中的行波传送进位线路（ripple-through carry line）将溢出的位送到下一阶段。或许有人认为右边可以换成半加器，以节省成本。因为C_0没有什么用处，但事实并非如此。我们很快就会看到它有非常大的优点。

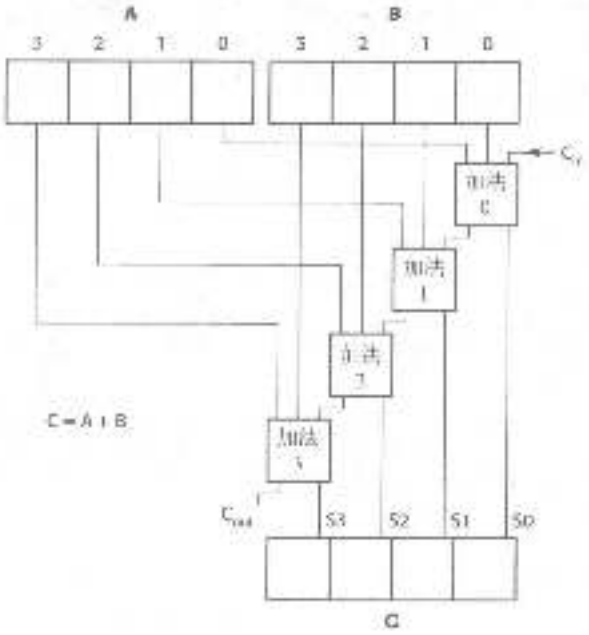


图5-7 4位数的并行加法

5.3 二进制减法：2的补码的整数格式

2的补码是一种表示整数的约定，这种方法支持以二进制存储负数。另一种表示整数的方式是符号加数量的方案：每个整数专门留出一位来标明正（0）或负（1）。尽管后面一种方法好像更简单，但是，在设计能够同时处理整数加减的电路时，对于硬件工程师来说，使用2的补码表示负整数和正整数所带来的好处是显而易见的。只需将减法中的第二个数切换成其负值，减法就可以按加法来考虑。从而，我们可以用加法器来完成减法运算。将使用2的补码表示的二进制整数转换成其负值时，只需将所有的位取反，之后再加1，就可以方便地完成转换。

这里的问题是：7－3＝?

考虑二进制的等价形式：0111 － 0011＝?

将二进制的3转换成相应的以2的补码表示的－3：0011 → 1100 + 1 → 1101

将两个值相加：0111 + 1101 = 0001，结果和我们的预期相同。

注意，我们忽略了进位输出的上溢！

这种处理方法（如果有些怀疑，可以使用纸和铅笔亲身试验一下）意味着，如果要将我们的并行加法器（见图5-7）改造成并行减法器，惟一需要做的改动，就是在B端的输入上插入反相器（NOT门），同时保持最初的进位（C_{in}）为逻辑1。这正是我们在最低有效位使用全加法器的原因。



图5-8　以2的补码的形式表示的4位带数

分析图5-8中2的补码表，粗略地观察时，人们往往注意不到其中负数的数量要比正数的数量多一个。在此，有人可能会产生疑问：你如何知道1100表示－4呢？我喜爱的答案是：将它加到+4上去，然后判断结果是否为0。警觉的观察者会注意到加法所产生的上溢进位，而我们好像忽略了这一点。数学的实际应用就是这样的。

现在，如果我们还希望继续使用ALU来完成加法，则插入反相器的做法只能满足一时之需，图5-9给出的示意性电路通过将XOR门用做可切换反相器（switchable inverter）来做到这一点。如果对XOR门的这种应用不了解，可以检查XOR门的真值表。除了完成数值数据的算术操作以外，计算机还需要加法器单元完成其他功能。如3.2节所述，IP寄存器在每个指令周期后需要增加1、2或4。并且，计算机常常需要根据基址寄存器（base register）和变址寄存器（index register）中的值来构造地址。

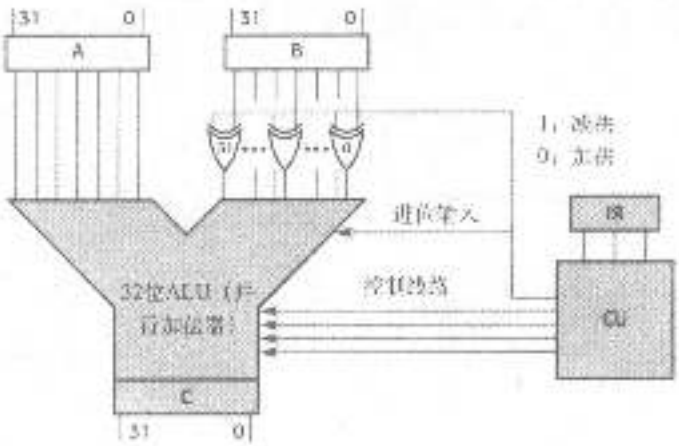


图5-9　示意性的ALU及其控制线路

现代的CPU提供多个ALU，分别用于整数、浮点和地址运算。整数ALU专为执行所有正、负二进制整数的常规算术运算而设计。它们包括ADD、SUB、MUL、DIV、CMP、AND和OR，还有一些更为特殊的运算，比如横向移位和位测试。在CPU的旁边是一小块快速内存，我们称之为CPU**寄存器**（CPU register），它们保存频繁使用的系统变量，或当前运行的程序中一些经常使用的变量。该ALU只负责整数运算、逻辑操作和位处理。CU读入指令并译码后，会通过控制线路触发ALU中所需的动作。

在这个ALU中，含有远比我们所设计的并行加法器电路更先进的版本。它常常表示为大写的Y。相加（或其他任何运算）的两个数在顶部输入，结果从底部放出。

5.4　二进制移位：桶形移位器

ALU常常需要将二进制数或位模式横向移位，以进行更为复杂的运算。在下一节中，我们会看

到数的乘法如何通过重复相加和移位来完成。由于程序员有时需要显式地移位，因此，C语言提供<<和>>运算符来执行左移和右移。这两个运算符调用机器的移位指令，可以处理8、16和32位数值。尽管左移也可以通过简单地将一个数加上其自身来完成，并且在需要的时候也确实常常这样做，但这是一项速度较慢的技术。同时，减法并不能完成反向的移位！交叉点开关器件可以用来实现**桶形移位器**（barrel shifter），图5-10给出了针对字节的实现。交叉点开关器件是一个开关阵列，可以由控制线路动态地操控。最初，设计它们是用于电话中的路由选择（参见16.1），当然也有其他一些应用。通过连通对角线上的所有开关，输入二进制模式可以在完成所需的横向位移后，传递到输出总线上。移位和循环都能够完成。

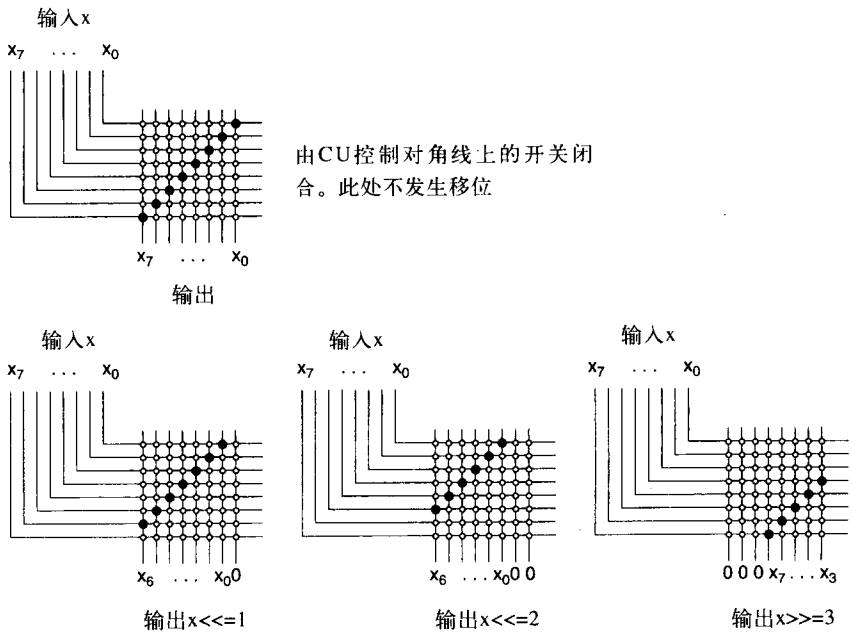


图5-10 用做移位器的交叉点矩阵

• C语言中的<<和>>运算符使用移位指令，如SHL和SHR。这些运算符在空出的位上填0，而如果使用循环移位运算符，如ROR或ROL，末端被推出的位会回绕，重新插入到相反的一端，如图5-11所示。ALU硬件必须能够根据指令译码器的选择，提供回绕循环移位和插0移位。单周期的桶形移位器对于加速CPU运算速度十分有益。而在采用单流水线译码的RISC计算机中，这是一项最基本的操作。

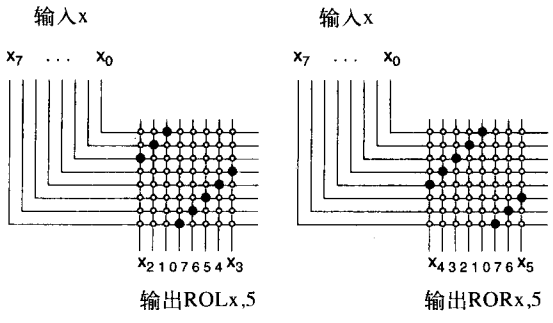


图5-11 用做字循环移位器的交叉点矩阵

5.5 整数乘法：移位和相加

二进制乘法的过程和十进制长乘法相同。此处保留了临时性的进位，以便读者更方便地观察运算的执行。

```
 173                10101101
  57 ×              00111001 ×
1211                10101101
8650               00000000
9861              00000000
                 10101101
                10101101
               10101101
              00000000
              ----------------
               0100110100000101
```

这个8位乘法（输出结果为16位）可以用并行加法器和桶形移位器电路来构造。图5-12给出了乘法的软件实现——使用加法和移位运算。

```
/* function to multiply two 16 bit positive integers
returning a 32 bit result, using only integer addition
and shift operators */

int multiply(int a, int c)
{
   int i;
       c = c << 16;

       for (i=0; i<16; i++)
       {
         if (a & 1) { a += c };
         a = a >>1;
       }
       return a;
```

图5-12　完成整数乘法的C函数

幸运的是，在十进制长乘法中使用的“乘、移位和相加”，在二进制运算中简化为“测试、相加和移位”。尽管我们觉得最后再来完成所有的加法更容易些，但计算机更倾向于随着运算的进行完成相加。要理解图5-12中的multiply函数，必须首先将加法运算符+及两个横向移位运算符<<和>>找出来。注意，它们每次将数值移一个二进制位，而非每次一个十进制数。因此，>>8向右移完整的一个字节。使用这些运算符，整数乘法可以写成一个测试、相加和移位的循环。以8位二进制数为例，这样我们就可以将运算过程放在页面上，一步一步地跟踪整数乘法函数的运算过程（见图5-13）。两个寄存器分别为累加器（A，保存结果和乘数）和被乘数（C），它们共同完成乘法运算：$45 \times 57 = 2565$。

现在检查一下，2565真的是0000 1010 0000 0101！将2565转换成二进制的方法见2.8节。

上面在描述算术运算的同时，也对CPU的寄存器做了一些介绍。5.3中在说明ALU的运作时（作为加法器和减法器），用到了寄存器A、B和C。大多数CPU的数据寄存器远不止三个，尽管我们也可以说奔腾处理器只有四个寄存器。图5-14中所示的ALU拥有另外的八个数据寄存器，它们都被安置在极靠近ALU的位置，这样在进行算术或逻辑运算时，它们就能够快速地提供操作数。

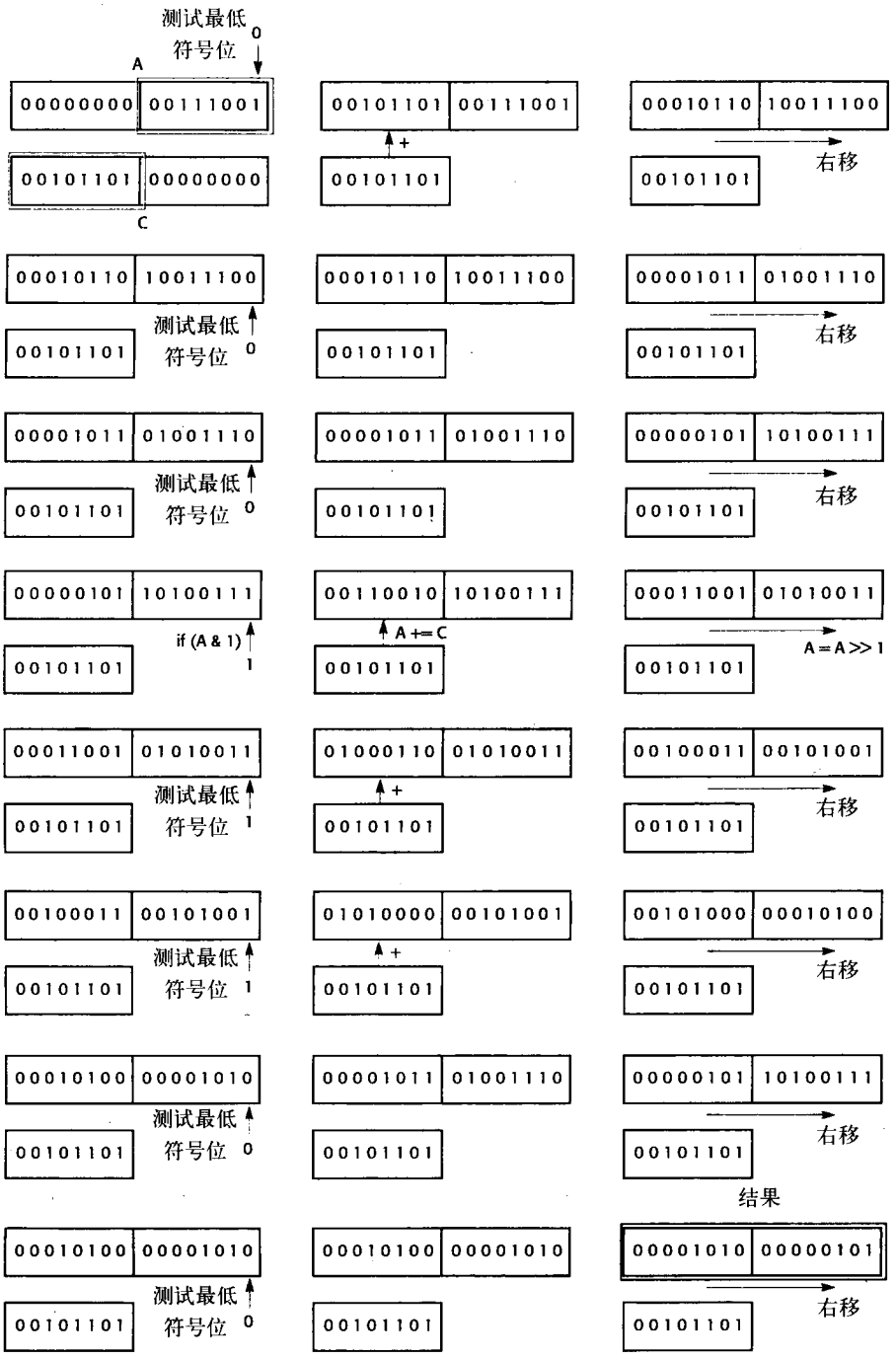


图5-13　8位二进制数乘法

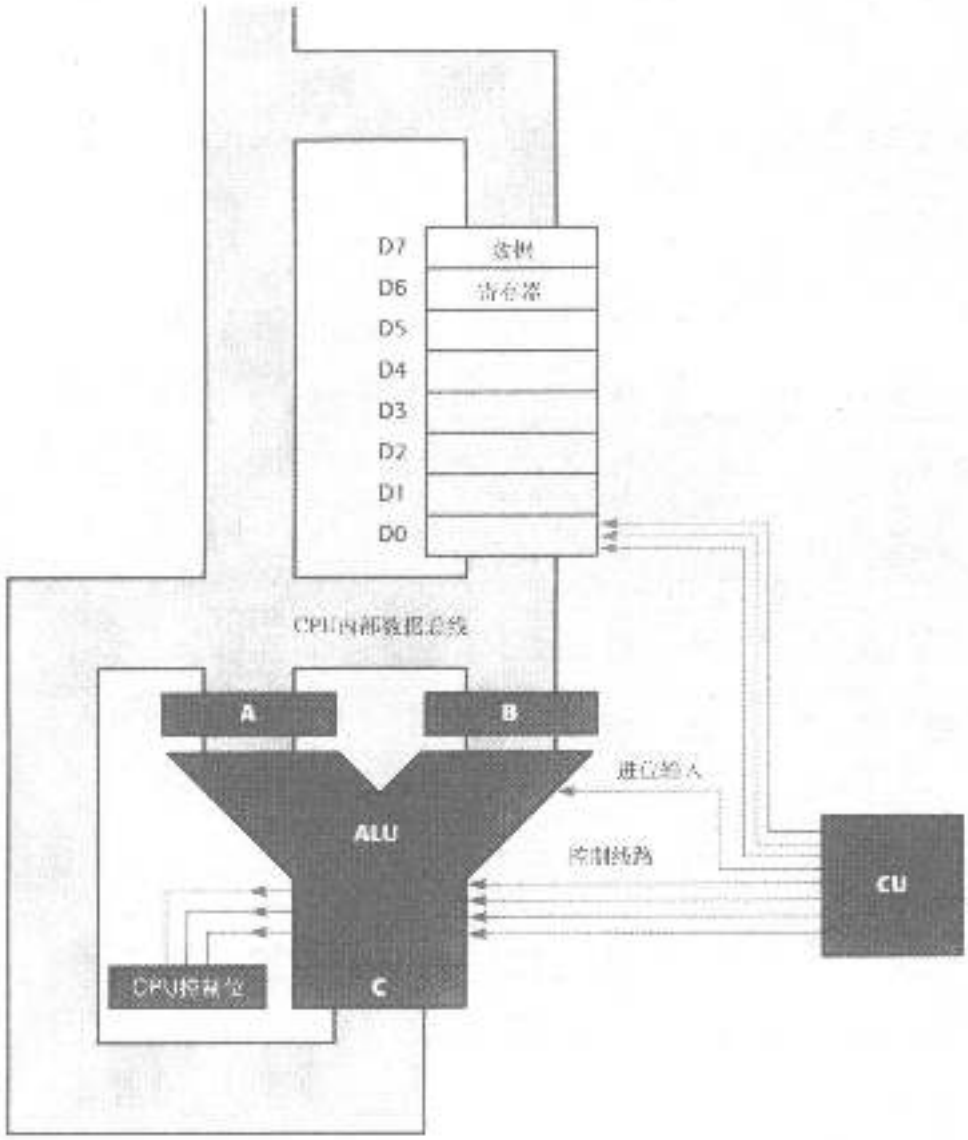


图5-14　ALU和CU的示意图以及CPU的数据寄存器

图5-15给出了市场上ALU 74xx181部件的简图，尽管从技术的角度看，它已经有些过时了，但却可以用来当做真实设备的例子。从中，我们可以清楚地看到八条输入线路——两个输入数A和B即通过它们进入ALU中，以及来自前一阶段的进位输入——在级联该设备以处理大于4位的数时，这是必需的。结果从F（F是function的缩写）和C_{out}送出。同时，它还提供另一个有用的特性：条件A == B会通过一个输出线路向外部发信号。注意，总共有五条控制线，用来从该ALU能够执行的48个操作中选取一个。您可能会担心由于2^5 = 32小于48，所以从48项中选取一个是不可能的，但要注意，此时，进位输入线被当做另外的一条控制线。CU可能会向这些控制线发送10010来选取ADD操作，或10111选取逻辑AND操作。尽管计算机工程师现在已经不在最新的计算机板上使用74xx181芯片，但是，他们依旧会从规格库中选取提供相同功能的等效VHDL模块，以使用VLSI内部的电路提供所需的功能。

S3–S0	M = 1 逻辑	输出 F M = 0 算术 C_{in} = 0	C_{in} = 1
0000	$F = \overline{A}$	F = A	F = A + 1
0001	$F = \overline{A \text{ OR } B}$	F = A OR B	F = (A OR B) + 1
0010	$F = \overline{A}$ AND B	$F = \overline{B}$ OR A	$F = (A \text{ OR } \overline{B}) + 1$
0011	F = 0	F = –1	F = 0
0100	$F = \overline{A \text{ AND } B}$	F = A + ($\overline{B}$ AND A)	F = A + ($\overline{B}$ AND A) + 1
0101	$F = \overline{B}$	F = (A OR B) + ($\overline{B}$ AND A)	F = (A OR B) + ($\overline{B}$ AND A) + 1
0110	F = A XOR B	F = A – B – 1	F = A – B
0111	F = $\overline{B}$ AND A	F = $\overline{B}$ AND A – 1	F = ($\overline{B}$ – A)
1000	F = $\overline{A}$ OR B	F = A + (B AND B)	F = A + (A AND B) + 1
1001	$F = \overline{A \text{ XOR } B}$	F = A + B	F = A + B + 1
1010	F = B	F = ($\overline{B}$ OR A) + (A AND B)	F = ($\overline{B}$ OR A) + (A AND B) + 1
1011	F = A AND B	F = (A AND B) – 1	F = (A AND B)
1100	F = 1	F = A << 1	F = (A << 1) + 1
1101	F = $\overline{B}$ OR A	F = (A OR B) + A	F = (A OR B) + A + 1
1110	F = A OR B	F = A + ($\overline{B}$ OR A) + A	F = ($\overline{B}$ OR A) + A + 1
1111	F = A	F = A – 1	F = A

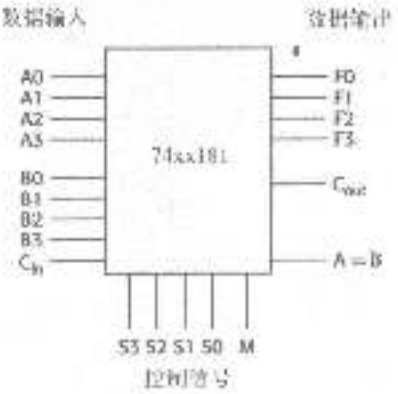


图5-15 74xx181 ALU的控制码

5.6 浮点数：从极大到极小

到目前为止，我们都假定所有二进制数均表示整数：没有小数部分的值，小数点后面没有数字。如果仅仅使用这类数字，会使程序员的工作很为困难。例如，存储极小的数，如一个电子的静态质量：9.1×10^{-31} kg，或0.000 000 000 000 000 000 000 000 000 000 000 91 kg，用标准的整数格式根本不可能实现。类似地，极大的数，如宇宙的年龄，也为程序员提出一个难题。观察这些数，我们可以看出，它们大部分由零组成，只在某一端有几个数字。用来表示数的浮点格式就利用了这一特点，它计算出零的个数，将之存储为“指数”，而将重要的数字部分单独保存为“尾数”。

浮点运算可以由软件例程（依靠整数机器指令，使用整数ALU）来完成，但是，现今越来越普遍的做法是，提供专门的硬件来执行浮点运算。这也就是平常所说的**FPU**（Floating-Point Unit，浮点单元）。Intel 80486和MC68030处理器拥有独立的FPU，和CPU在同一芯片上。奔腾处理器已经将所有这些算术运算都完全集成到主CPU中，提供专门的ALU运行浮点运算。

最初，IBM PC机内提供一个专门的插座，用以安装可选的浮点单元（Intel 8087）。后来Intel 8087被合并到CPU芯片中。现今，浮点数的常见格式是IEEE 754中32、64和128位数的标准。有趣的是，由于Intel 8087的批量生产，才实现了浮点数及运算的全行业单一标准。在这之前，每家计算机生产厂商好像都以为每个新处理器引入新的方案为傲。

尽管32位整数运算是ALU的主要动作，并且是计算机运作的核心，但是，应用程序的开发人员

更常用到的还是有小数部分的“实数”。这种浮点数分成二部分存储，表示该数为正或负的部分，小数点的位置和数字本身。从图5-16的代码片段中，我们能够看到C代码中浮点数的应用。

```
float net_cost, tot_cost, price;
float vat = 0.175;
int items;
    tot_cost = price * items;
    tot_cost = net_cost - net_cost * vat;
```

图5-16　在C程序中使用浮点数

在介绍浮点数运算的细节之前，先来回顾一下大多数软件和硬件使用的存储方法，这对后面的讲述会有所帮助，这涉及到将常见的格式转换成科学或指数形式。图5-17给出了二进制的情况。

常规表达	指数表达
1234.5625	1.2345625×10^3
-3.3125	-3.3125×10^0
0.065625	6.5625×10^{-2}



1234.5625	10011010010.1001	$1.0011010010101 \times 2^{1010}$
	未经规格化的格式	规格化后的格式
-3.3125	-11.0101	-1.10101×2^1
0.065625	0.00011	1.1×2^{-4}

IEEE 754 32位格式的浮点数存为存中的表示：

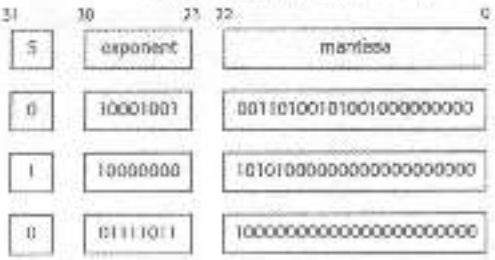


图5-17　IEEE 754浮点格式

浮点数被“规格化”为一种标准的布局，或称格式，以便任何例程或硬件都知道数的表达方式（见图5-17）。规格化需要向移动二进制数点，不断调整指数以保证值始终正确，直到只有唯一的前导数1剩下为止。前面的符号位清楚地表明数的正负；接下来是指数，后面是尾数。如果想让浮点数既能表示极小的数，如2^{-127}，也能表示极大的数，2^{127}，则指数必须既可以为正也可以为负。使用2的补码表示负指数值，可能是一种较为合适的方案；但是，IEEE却采用了另一种方法——偏移127，也就是在存储真正的指数值之前加上127。因此，指数0在内存中为127，指数－127在内存中为0。由于2^{-127}实在是太小太小，实际上很接近于0，因此，我们常常将它看做是指数为00000000，尾数为1.00000000000000000000000的数。但是，在处理算术运算时，就不能使用精简的规格化形式了。

手动地将十进制float转换成IEEE二进制float比较繁琐，但并不难：

1）将整数部分转换成二进制形式。

2）将小数部分转换成二进制形式，注意1/2，1/4，1/8，1/16所对应的模式。

... 128 64 32 16 8 4 2 1 · 0.5 0.25 0.125 0.0625 0.03125

3）通过将二进制的数点移动到形如1.xxxx的格式，对其进行规格化，得出或正或负的偏移值。

4）删除前导的1（略去，多出一位存储更多的有用值），将右面空余的位填0，得出一个23位的尾数。

5）将127加到偏移数上，得出8位的指数。

我们可以编写程序来执行这样的转换，并将结果存储到文件中，供后来查看。图5-18中的C程序将浮点数231.125写到文件float_data中。我们可以使用文件查看工具来检查它。图5-19给出这个过

程及结果，其中使用Unix od命令（octal dump）来提供文件的信息。实际上，od远比这个名字所暗示出来的用途更有用，但是，Unix的早期出身从某些方面吓住了当代的用户！面向字节的微处理器革命，已经将3位八进制变成历史。检查Unix的手册页（man od），就会得到有关如何让od以多种格式显示文件内容的介绍，包括ASCII、整数、原始十六进制数和科学指数。

```
/* floatit.c - to write a real number into a file for
   viewing */

#include <stdio.h>

int main ( )
{
        FILE *fp;
        float f = 231.125;
          if (fp = fopen ("float_data", "w"))
          {
              fwrite(&f, 4, 1, fp);
          };
        return 0;
}
```

图5-18 将浮点数存入文件的程序

```
rwilliam@olveston [78] cc floatit.c -o floatit
rwilliam@olveston [79] floatit
rwilliam@olveston [80] od -x float_data
  0000000 4367   2000
  0000004
rwilliam@olveston [129] od -f float_data
  00000002.3112500e+02
  0000004
rwilliam@olveston [130]
```

图5-19 使用Unix od实用程序检查数据文件

十六进制值43 67 20 00是32位浮点数：

0 100 0011 0110 0111 0010 0000 0000 0000

| | | |

符 8位指数 24位尾数中的23位

号 以偏移127

位 的格式表示

各种浮点格式的范围和精度都列在表5-1中。

或许您会注意到尾数字段是23位，而我们却声称24位精度，这也许出乎意料。为了节省一位的存储空间，尾数的前导1被取消，没有显式地存储起来。由于浮点数在存储之前一定经过了规格化，故而尾数总有一个前导1，才使得这种过时的极度节俭成为可能。

表5-1 浮点格式的范围和精度

	范围	精度	
32位	8位	24位	(1 in 16×10^6)
64位	11位	53位	(1 in 8×10^{15})
128位	15位	64位	(1 in 16×10^{18})

5.7 小结

- ALU是CPU的一部分，它负责执行算术和逻辑运算。
- 根据德·摩根等价定律，数字电路可以使用AND或OR逻辑电路重新拟定。
- CPU可以进一步划分成两部分：算术逻辑运算单元（ALU）和控制单元（CU）。

- 最简单的半加器电路不能处理进位输入位，所以在构造并行加法器时，一定要使用全加法器。
- 以2的补码的形式来表示负数，能够让加法器电路执行减法运算。
- ALU中含有一个复杂的并行加法器。它的操作由控制单元根据当前指令译码的结果决定。
- ALU能够执行的其他操作，比如字的横向移位，能够帮助完成乘法和除法运算。
- 奔腾处理器有单独的浮点处理单元来执行实数（有小数部分的数）的运算。
- 浮点数存入存储器时，使用IEEE 754规格化格式。

实习作业

我们推荐的实习作业依旧是研究数字逻辑电路，但是，现在转到与ALU相关的电路上。基本的例子是加法器。同样，这份作业可以使用试验电路板或计算机模拟程序来完成，但最好两种方式都试一下。

在设计电路之前，不管是半加法器、全加法器还是其他，都要先写出其真值表。

如果手头有多个可以工作的器件，可以将它们装配成一个并行加法器。之后，可以用它来演示使用2的补码进行的减法运算。

练习

1. 将本书中使用的逻辑符号（AND、OR、NOT）与数学课及C语言中使用的那些符号对应起来。
2. 使用简单的AND、OR和NOT门，设计XOR（异或）门的逻辑。记住，要从绘制真值表做起。
3. 使用一个7400（四元组，二端输入NAND门）构造一个XOR门。
4. 将反相器放在门电路之前和之后有区别吗？换句话说，“NOT AND”和“AND NOT”一样吗？
5. 5.2节中的并行加法器的缺点是什么？
6. 电子汽车锁使用32位代码，如果一次尝试花费3秒钟的时间，那么使用系统化的搜索方法，电子“黑盒”要花多长的时间才能找出正确的代码来？
7. 为防盗警报器设计一个硬件逻辑电路，要求能够接收四个来自于传感器开关的输入，如果其中超过三个激活，则打开报警器。
8. 除Speak & Spell玩具以外，德州仪器公司以什么而闻名？
9. 电话通话时的比特率是多少？
10. 下面这些C语言中的类型为多长：char、long、int、float、double、unsigned char？
11. 将图5-13给出的8位整数乘法过程改成整数除法。
12. 什么是2的补码？列出一系列由5位的以2的补码表示的数。将下面的数表示成8位的2的补码：0、1、−1、127、−128、15、−23。
13. 将图5-6中的4位并行加法器扩展到8位数，包括计算“A减去B”的能力。
14. 在C语言中，int和unsigned int的区别是什么？什么情况下这种区别才重要？
15. 使用5.6节中介绍的floatit.c程序，以及文件转储工具od，检查将−123.625转换成IEEE 754 32位浮点格式后的结果。接下来试试0.00123625和−0.0123625。

课外读物

- Thewlis和Foxon（1983），对逻辑电路的非常基本的介绍。
- 德州仪器公司（1984）：The TTL Data Book，也称做橙色圣经。
- Heuring和Jordan（2004），介绍有关运算电路的更高级细节的章节。
- Patterson和Hennessy（2004），第4章中有更多有关计算机运算的理论探讨。
- Hamacher等著（1996），比较多的硬件细节。
- 有关硬件描述语言VHDL的更多细节，参见：
  http://www.doulos.co.uk/。
- William Kahan讲述自己参与IEEE 754标准设计的过程：
  http://www.eecs.berkeley.edu/~wkahan/ieee754status/754story.html
- 这些网址可以通过本书的配套网站访问：
  http://www.pearsoned.co.uk/williams

第6章　计算机的逻辑构成：存储器

存储数据和程序指令和快速存储器是数字计算机的基本部件。当前，它的基本构成是经电路，但并非从来都如此。每个位都有对应的存储单元，其构造方式可能有多种。我们将要论述的是SRAM和DRAM的基本存储单元。现代PC中，必须将一定数量的DRAM芯片组织起来，以向CPU提供足够快的服务。由于存在多个存储芯片，因此必须精心设计，保证在存储访问周期内正确的存储芯片被激活。从程序员的角度看，IO端口与存储器中的存储单元在很多方面均相同。（本章中，“存储器”和“内存”的英文名称都是memory。一般文中在讲述硬件时，多采用“存储”一词，而在讲述软件和程序时，多采用“内存”一词，以与惯用的说法相一致。同时也是为了区别磁带、磁盘等外部存储介质。译者注）

6.1　数据存储

即使简单的基本存储单元，至少也需要两个门电路。虽然并非绝对如此，但由单个门电路构成的基本存储单元（见图6-1）确实存在严重的缺陷。

这个由单个AND门构成的基本存储单元，在输入变回1之后，能够记住前面的0。这个电路利用了AND门的特性，即AND门的任一输入为0，AND门的输出就为0。实际上AND门就是一个“0检测器”。如果我们将输出信号0反馈到某个输入端，那么，即使其他所有的输入都变为1，输出依旧为0。信号的“通过”延迟很短，在4ns的数量级。遗憾的是，这种方案有一个严重的缺点：不能应付短暂断电，数据不能保持。同时，它是一次性的存储。虽然可以用另一个门电路将反馈连接断开，但更常用的还是由两个NOR门构成的电路，即基本锁存器（latch），如图6-2所示。

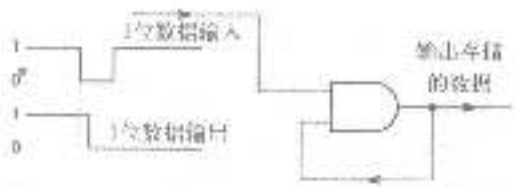


图6-1　单个位的存储

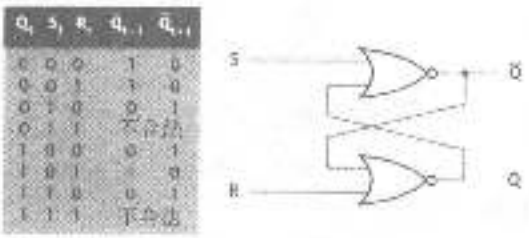


图6-2　基本锁存器电路

在这个电路中，使用NOR门是因为它是“1检测器”。只要任何输入提供1，输出就会变为0，和前面一样。我们的计划是，在S输入端输入1，使输出$\bar{Q}$产生0，这个值又会被回馈到另一输入端，最终在Q输出端产生1。当最初的输入S变回到0时，电路的输出状态保持不变。从而，正脉冲，不管来自于R或S，都被锁存器记下来，直到配对的输入收到删除脉冲为止。

锁存器，或称为触发器（flip-plop），是一种能够记忆的逻辑电路。它通过使用反馈连接巩固接收到的信号，而后使用它来取代输入信号，从而达到记忆的效果。NOR或NAND门都可以用来构造这一电路，因为NOR门当输入部分或全部为1时，输出为0，NAND门则当部分或全部输入为0时，输出为1。因此，我们能够撤去原始的触发脉冲，而不会影响到输出，因为通过来自于互补电路的反馈连接，使得配对的输入端依旧保持相同的值。这个电路之所以能够工作，是因为输入信号通过逻辑门电路并改变输出的状态，需要一小段时间的延迟。图6-3按照时间关系给出事件序列。

和所有的逻辑电路一样，市场上存在许多替代物和变种：R-S、J-K、D型和T型均属于不同的触发器设备，每一种都适合于某种特定的应用。锁存器在任何时候都响应输入的变动，而触发器则与控制时钟脉冲同步运行。但是，它们都使用交叉耦合的门电路，通过反馈连接互相作用。两个NAND也可以连接起来作为一个R-S锁存器（如图6-4所示）。这个电路可以用来制成猫洞监视器，提示您的宠物是安全地呆在家里，还是在外面给邻居制造麻烦。依照开关安装方式的不同，LED（light-emitting diode，发光二极管）可以在猫外出的时候发光，或安全地呆在家里时发光。档板的回弹对开关会有一定的影响，需要认真加以考虑；上拉的10 kΩ电阻确保在档板既没有接触IN触点也未接触OUT触点时，开关的输入保持5V（逻辑1）。

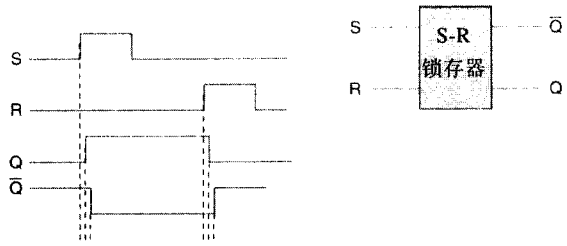


图6-3　S-R锁存器信号时序

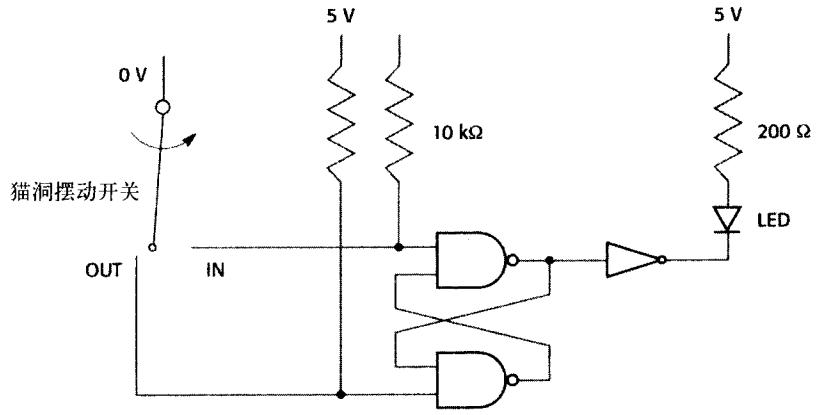


图6-4　猫洞指示器

6.2　存储设备

一般地，**RAM**表示“随机访问存储器”（Random Access Memory），**PROM**表示“可编程只读存储器”（Programmable Read-Only Memory）。但是，根据这些术语并不能区分RAM和PROM的功能，因为这两种设备都是可以随机访问的存储器。这就使得RAM单纯就名字而言没有什么特别的特征。实际上，RAM应该为**RWM**（Read-Write Memory，可读可写存储器），而PROM应为**WORMM**（Write Once Read Many Memory，一次写入多次读取存储器）。

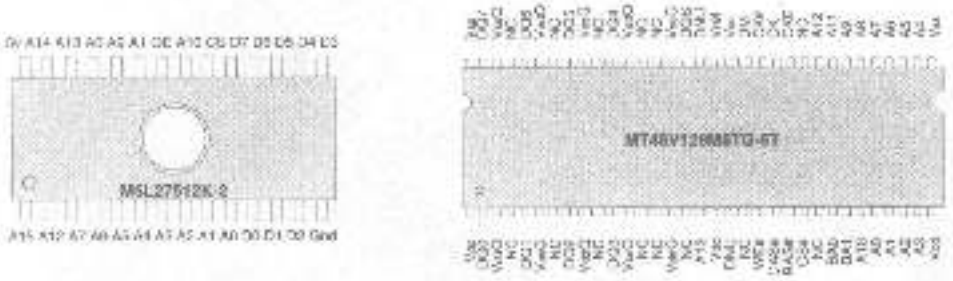
存储芯片多种多样，用途各不相同，但均支持随机访问。这意味着，不管内容存储在芯片中的什么地方，所有数据项的访问时间是一样的。串行磁带存储设备有很大的不同，显然读取磁带开始处的数据块要快于读取结束处的数据块。

表6-1列出了一些计算机内使用的不同类型的存储设备，同时也列出了它们对应的访问速度。

图6-5中的EPROM中间有一个小的石英玻璃窗口，需要对EPROM重新编程时，只需用紫外线透过这个窗口进行照射，就能够擦除现有的内容。这种重新编程的方法（需要将存储芯片从电路板上移走）现已大部分被在电路板上就能够重新编程的EEPROM所替代。我们可以小心地为EEPROM加一个比正常操作要高的电压来完成擦除。

表6-1 存储器的类型

RAM	50ns	DRAM，动态随机访问存储器，可读可写，随机访问
	10ns	SRAM，静态随机访问存储器
ROM		只读存储器，二元可写，随机访问
PROM		可编程ROM，可写，但只能写一次
EPROM	150ns	UV可擦除PROM，在芯片的封装上留有窗口，但需外充遇上
EEPROM		电可擦除PROM，适用于参未久性的程
FLASH		类似于EEPROM，可重新编程，非易失性ROM



三菱，M5L27512K-2
64 KB EPROM
200 ns访问时间

三菱，MT46V128M8TG-6T
128 MB(1 Gbit) DRAM，DDR 333
167 MHz工作频率，6ns访问时间

图6-5 存储器封装的实例

6.3 静态存储器

SRAM（Static RAM，静态随机访问存储器）由数以千计的触发器电路构成。它是计算机内使用的存储器中最快的，也是最贵的，一般仅用于高速缓存。有些SRAM电路能够在10ns内响应访问请求，这使得它们适用于最快（L1）的芯片内高速缓存。SRAM的另一个优点是电力消耗低，这使得它们成为电池供电的便携式设备以及电池支持的CMOS RAM的理想选择。SRAM常常制成"单字节宽"，即每次读写访问只能处理8位数据。封装和针脚的布局常常依照JEDEC标准，从而能够保证与EPROM的兼容性。故而，用EPROM来替换SRAM是可能的，只要其中含有的是正确的程序。

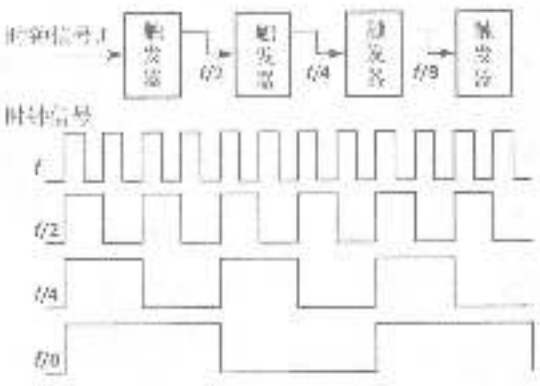


图6-6 触发器用做频率除法器（分频器）

奇妙的是，除了可以作为存储器的基本存储单元以外，触发器还能够完成除以2的运算，参见图6-6。只有在极为特别的情况下，它才有机会发挥这种不寻常的能力。当脉冲流以确定的频率通过触发器时，输出流将会是原始频率的一半。尽管在建造ALU除法器时很难用上这一特性，但在合成CU和计算机内其他地方所需的各种系统时钟频率时，这一特性十分有用。在3.4节中，我们介绍了重要的系统时钟信号、计算机中，用来生成所有相对较低频率信号的电路，就依赖于触发器。而采用时钟倍频技术的奔腾处理器所需的高频，也来自于一种被称为锁相环（Phase-Locked Loop，PLL）的电路技术，但是该技术超出了本书范围。

6.4　动态存储器

DRAM（Dynamic RAM，动态随机访问存储器）采用完全不同的方法来存储二进制位，并且，它所占用的面积仅为等效SRAM触发器存储器的1/4。DRAM利用了场效应晶体管（Field Effect Transistor，FET）栅极连接的微小电容。图6-7给出一个储能电容器，我们可以使用字线（word line）选择它，通过位线（bit line）读（或改写）。现代的512 M位DRAM必须利用仅15 fF（飞微微法，femtofarads，femto是丹麦语中的15），即10×10^{-15}法拉左右的电容来工作。这是一项了不起的成就。电容能够将电荷保持一段时间，就如同水桶可以存水一样。遗憾的是，就DRAM这种情况，这个水桶是漏水的，每秒钟大约要重新注满100次！这一频率正好和荧光灯管的频率相同。另外涉及的一个问题是随机错误，有可能由宇宙射线或自然辐射引发的质子微粒造成。64 MB的DIMM在正常使用的情况下，统计计算得出可能每20年发生一次这类错误。尽管DRAM有许多小的不方便之处，它还是成为当前几乎所有计算机系统的基本组成部分，原因就在于它价格低廉。DRAM按照存储能力（256 M位～1 G位）和访问速度（50ns～5ns）进行分类。存储器访问时间指芯片花多长时间才能响应对某个基本存储单元的访问。我们将在下一节中论述刷新周期。大多数情况下，存储器的容量越大越好，而对于访问时间，则是越小越好，100 MHz系统时钟的PC需要能够在10ns内做出响应的快速反应设备。DRAM的另一项缺点是在读/写事件之间需要"休息"。虽然DRAM的标称访问时间可能为10ns，但并不表示每秒钟可以读取该设备1亿次。最小周期时间可能会大于100ns，因而每秒钟实际的最大访问频率只能达到一千万次左右。周期时间延迟是因为在下次读操作发生之前，我们必须对位线进行预充电，使之达到恒定的电压。DRAM封装提供1位、4位和8位宽度，但是，更常见的是直接购买已经安装到SIMM或DIMM上的DRAM，它们提供64位字长的读/写能力。64位奔腾处理器的数据总线需要两个SIMM并排，以构成64位的字。

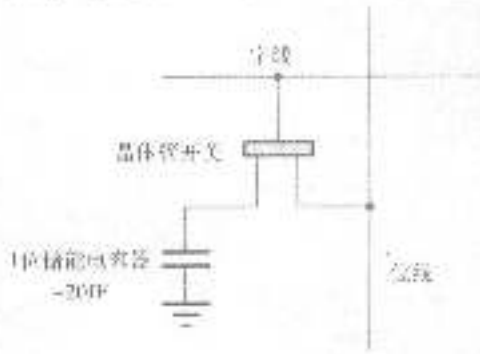


图6-7　单个DRAM存储单元

主存储器现在都由DRAM的阵列构成。在过去的20年间，制造这些设备的制造工艺取得了惊人的改进，芯片的尺寸减小，每兆字节的价格暴跌，与此同时，访问速度不断提高。

DRAM存储芯片被组织成电容性基本存储单元的两维阵列（见图6-8）。行由字线构成，列由位线构成。这样，该阵列能够提供对任何基本存储单元的随机访问。行访问称为**RAS**（Row Address Select，行地址选择），列访问称为**CAS**（Column Address Select）。在写周期内，表示位数据的少量电荷沿位线，经闭合的晶体管开关电路，传送到电容器上，开关电路断开后，它们就保存在那里。类似地，读周期则通过激活某条字线选择正确的行，之后读取相应的位线，这样就可以访问到所请求的基本存储单元。将地址划分成高、低两部分复用DRAM

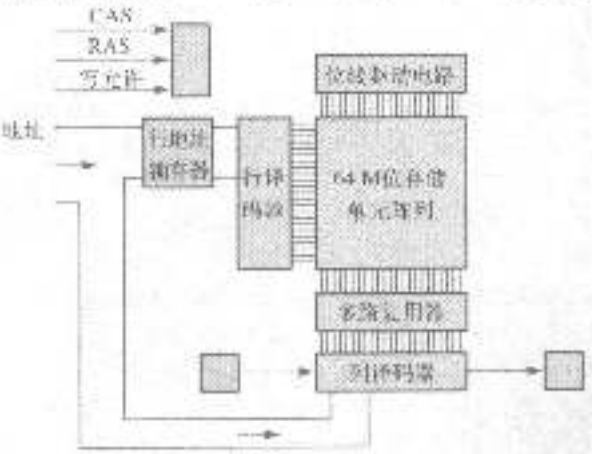


图6-8　DRAM存储芯片阵列

地址的做法很常见：行编号和列编号，参见图6-9。这降低了所需译码电路的大小，并提高了它们的速度。对于支持突发式访问的DRAM，这种方案也是必须的（在6.6节论述）。要访问二维阵列中的基本存储单元，首先要通过RAS信号，发出行编号，写入行锁存器中。然后，使用同一地址总线，发送列编号，从正确的列中选择输出（CAS）。将32位地址分成两个16位数字，然后依次经由复用的总线发送的工作，由DRAM控制器来完成。

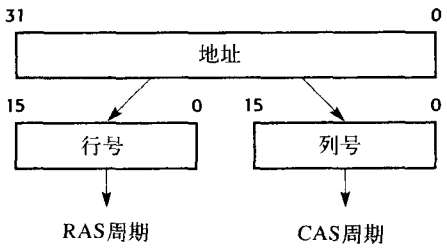


图6-9 DRAM地址排列

6.5 DRAM刷新

DRAM的刷新需要耗费时间，并且需要加入额外的电路。其中包括用以保持地址值的硬件计数器，在执行专门的读写刷新周期时需要用到。刷新时CPU并不亲自访问存储器，而是由刷新电路在恰当的时间对下一行执行RAS周期，将上半部分地址发送给RAM。行的刷新是依次进行的，这样可以避免由于遗漏造成数据丢失。所有列的位线上的电压被放大，并重新应用到储能电容器上。这样，整行的基本存储单元就同时被刷新。对于256 M位的DRAM，这意味着将每10 ms所需的操作从256 M降低到16 K位。

存储模组在设计时就已经包括刷新电路，从而增加了硬件的复杂程度和成本。最初IBM PC的4个DMA通道中，就有一个专门负责刷新DRAM，现在看来，这样做是对资源的一种浪费。新一代PC的存储系统并不依赖于这类机制。

有意思的是，在SuperBrain和Sinclair计算机中使用的Z80 CPU，将DRAM刷新逻辑作为存储器接口的一部分提供。这种做法很快在系统设计人员中得以推广，以前他们不得不自己去小心翼翼地处理构建DRAM刷新电路所面临的各种困难。下一代DRAM芯片被设计成自动刷新，不需要外部的干涉。

6.6 分页访问存储器：EDO和SDRAM

近年来，加速内存访问的需求日益强烈。CPU越来越快，特别是Intel 80486和奔腾处理器采用时钟倍频、三倍频和四倍频来加速CPU的运算之后。采用时钟倍频的2.8 GHz奔腾处理器依旧被由200 MHz系统时钟驱动的内存和支持芯片包围。主存仍以很低的主板时钟速率运行。Intel认识到了这种瓶颈，它在CPU芯片中集成了16 KB的快速SRAM高速缓存。奔腾II还在紧靠CPU的位置提供256 KB的芯片外二级高速缓存，缓存和CPU均封装在一种专门的盒式包装中。这种方案未能在奔腾4处理器上继续实施，奔腾4处理器在芯片内提供32 KB的L1高速缓存和2 MB的L2高速缓存。

当前，主板的时钟速度一般为100、133、166或200 MHz，由于采用双倍数据速率（Double Data Rate，DDR）的DRAM存储芯片，这些主板实际的FSB速率是200、266、333或400 MHz。就奔腾4的FSB而言，总线的实际运行频率可以为200 MHz，由于处理器支持四倍数据速率（Quadruple Data Rate，QDR），实际的FSB应该是800 MHz。因此，不只CPU以高于系统总线的时钟速率运行，主存储设备也是如此。每个时钟周期内，地址能够发送两次，同时，通过提供双通道构架，可以访问的数据也加倍，这就是所谓的四倍频数据传输。

除了需要定期刷新这一缺点以外，DRAM还存在恢复问题。每次访问后，列线路，或称位线路，需要时间来充电，之后才能进行下一个周期的访问。这种延迟被称做**周期时间**（cycle time）。如果能够将对内存芯片的访问**交错**（interleave）起来，那么这种延迟就可以避免。想像一下，两组存储器，偶数内存地址在右，奇数内存地址在左。随着CPU内存读取指令的不断执行，对内存的访问将会是左、右、左。通过这种方式，在下一次访问到来之前，芯片一般有两个内存周期的时间来进行恢复。

另一项技术，如图6-10所示，叫做页模式或**EDO**（Extended Data Output，扩展数据输出），采用这项技术时，针对每次读写请求，都突发性地执行四个访问周期。序列中第一次访问是常规的，同时行地址被锁存起来，正确的列被选定，但接下来的三个列会被依次快速地读取出来，只改变列

的地址。这样就可以消除三次访问的RAS周期，对四个字的突发访问能够节省约40%的时间，从而降低周期时间，就总线周期而言，这种四阶段的突发访问需要5-2-2-2，而非5-5-5-5。也许需要提醒一下，这种速度上的收益只有在牺牲真正随机访问能力的情况下才有可能实现；75%的数据是顺序访问的！同样，严格地讲，我们永远不能确定CPU会直的使用后续三个存储单元中的指令或数据，它们在读入后可能被简单地忽略，并遗弃。

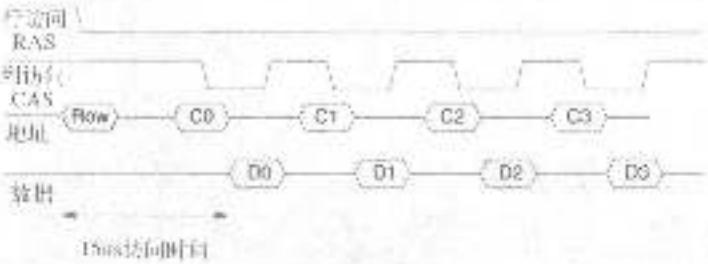


图6-10　EDO DRAM时序图

当代的同步突发式DRAM，**SDRAM**进一步提高了访问速度，只需将起始存储单元的地址发送到芯片，芯片会使用集成的CAS计数器自动循环处理其后的存储单元。这种突发（bursting）方法允许SDRAM提供5-1-1-1的访问性能，在某些设备上甚至能够达到3-1-1-1。EDO和SDRAM最大的不同在于时序。EDO由RAS和CAS信号线控制和同步，而SDRAM和系统时钟同步。为了强调同步和异步操作之间的差异，我们可以看到，EDO的性能以访问响应延迟来计算，而SDRAM由所支持的最大时钟速度来评定。SDRAM还提供内部自刷新机制，突发访问的长度可选择，单次请求可以访问1、2、4、8，甚至整行数据。现在，在主板时钟频率为200 MHz的情况下，DDR DRAM芯片能够支持超过3.2 GB/s的突发访问速率。

存储器的突发式数据访问在高速缓存应用中得到了充分的利用。我们将在第17章更全面地介绍这一主题。在此需要提一下的是，典型处理器的高速缓存以32字节为单位，突发式地载入数据。这需要用到四个总线周期，每个周期执行8个字节，这和图6-10中列出来的EDO DRAM完全匹配。我们所面临的挑战是在不增加成本或错误率的前提下，提高主板的时钟速度，使之能够与新一代的DRAM芯片相匹配。

PC主板现在一般都接受存储模组，见图6-11。它们是小型的130 mm×30 mm的电路板，其上装有8个或9个DRAM芯片。第9个芯片提供奇偶错误校验。根据所使用的接插件的类型，这些模组被称为SIMM（72针单边接触内存模组）或**DIMM**（184针双边接触内存模组）。市场上存储模组的大小各不相同，可供选购的存储模板包括256 MB、512 MB和1 GB的DIMM。也许将它们描述为32 M×64位、64 M×64位和128 M×64位模组更清楚些。对于DDR SDRAM DIMM，突发访问时间现在已经达到2.5ns，和200 MHz主板时钟速度匹配。

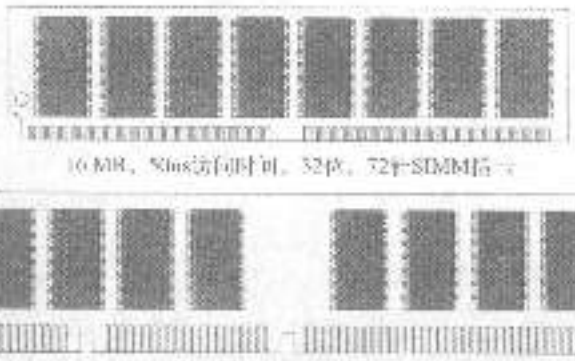


图6-11　72针SIMM和168针DIMM DRAM模组

牢记：$2^{20}=1$ M，因此：$2^{22}=4$ M

在256MB DIMM上的每个8×32MB EDO DRAM芯片，都在其存储单元内存储8位数据。从而使得单次读-写操作能够处理64位数据。它们额定的访问时间是10ns，和100 MHz主板相配套。由于奔腾处理器64位的数据总线能够同时传送4个16位长的指令，这意味着每条指令只需2.5ns（10/4=2.5ns）。此外，通过EDO DRAM的访问机制，随后三个总线周期只有5ns的延迟，从而能够达到大约每1.6ns（25ns/4×4指令=1.6ns/指令）一条指令。这种估算有些乐观，在估算过程中我们忽略了其他一些重要的时间延迟，因为它们涉及电子工程领域内的知识，在此暂不考虑。

为了使访问更快，DDR存储芯片允许单个时钟周期内完成两个读（或写）操作。

值得一提的是，DRAM芯片的运行电压持续下降，从5V到3.5V，直至2.5V。因此，在为你的PC选择存储模组时，要考虑这个因素，避免不匹配问题。

安装SIMM卡时，需要以偏离垂直方向30°的角度将其插入到插槽内，然后向垂直方向推到直立的位置即可。要拆下它，首先要松开两边的金属卡簧，让SIMM能够倾斜下来。DIMM的安装和卸下更方便，只需垂直向下压入插槽即可装上，而两端的卡簧可以将其弹出。

6.7 存储器映射：寻址和译码

如3.6节所述，从CPU中引出的地址线的数量，决定了能够相对容易地访问的内存的最大长度：20线对应1 M内存，24线对应16 M内存，32线对应4 G内存。随着存储器的更新换代，主存（RAM）已经变得越来越便宜，但内存的最大寻址空间依旧很少完全用尽，对于大部分用户，这样做都太昂贵了。图6-12所描绘的情况并不具有普遍性。

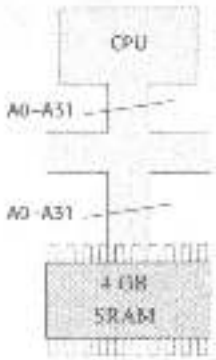


图6-12 内存的理想化配置

读或写周期内，需要额外的译码器逻辑电路来选择正确的存储芯片。CPU发出的地址中，低端地址位在RAM内用于选择基本存储单元，高端地址位由系统内存译码器逻辑电路来选择正确的芯片。如果不这样做，所有的RAM芯片将会同时响应来自于CPU的所有读-写请求。这就如同拨打0117 987 654，区号0117被忽略，全英国所有号码为987 654的电话都同时应答。

系统设计人员绘出**存储器映射图**（memory map）来描绘存储芯片在地址空间中的位置。为避免访问内存产生问题，最好是将存储芯片按照次序依次排列。如果两个芯片在内存读周期内同时响应，则最终在数据总线上产生的数据将会无效，因为它将会是来自于不同存储单元数据的混合。这会为CPU制造混乱，可能会导致系统崩溃。如果不在开始时充分规划好内存的译码，则会在寻址空间产生“空洞”，从而要求可执行程序必须分段。如果CPU试图在空的存储单元上执行读取-执行周期，则会迅速导致灾难的发生！图6-13给出了存储映射图的一个具体例子，图6-14给出了对应的电路示意图。

器件	大小	引脚	32位地址总线	地址范围
PROM1	1MB	20	0000 0000 xxxx ++++ ++++ ++++ ++++ ++++	0000 0000～000F FFFF
RAM1	16MB	24	0000 0001 ++++ ++++ ++++ ++++ ++++ ++++	0100 0000～01FF FFFF
RAM2	16MB	24	0000 0010 ++++ ++++ ++++ ++++ ++++ ++++	0200 0000～02FF FFFF
RAM3	16MB	24	0000 0011 ++++ ++++ ++++ ++++ ++++ ++++	0300 0000～03FF FFFF
RAM4	16MB	24	0000 0100 ++++ ++++ ++++ ++++ ++++ ++++	0400 0000～04FF FFFF

+ 直接用做内部选择的地址线
x 被忽略的行，表示部分（退化）寻址
0 选定芯片时必须为0
1 选定芯片时必须为1

图6-13 小型计算机系统的存储器映射

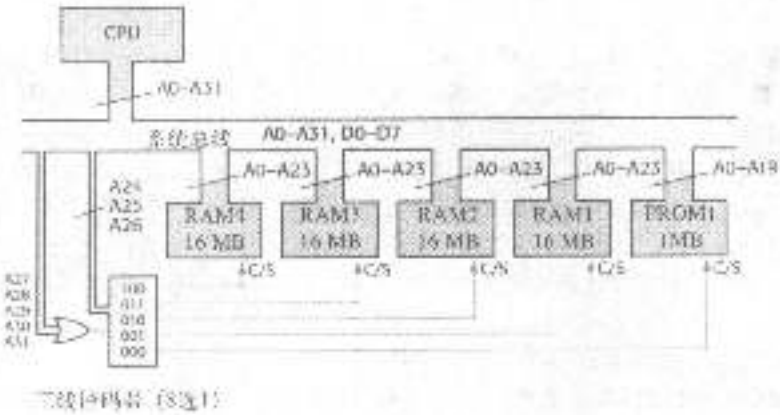


图6-14　存储系统中的译码电路

图6-15给出另一种不同的PC存储器映射方式，在该方案中，这种方案更易于扩展。采用这种方案，可以很容易地将PC的内存从常见的512 MB扩充到更大，直至可能的4 GB。第一次接触这么大的内存空间，肯定会感觉出乎意料。

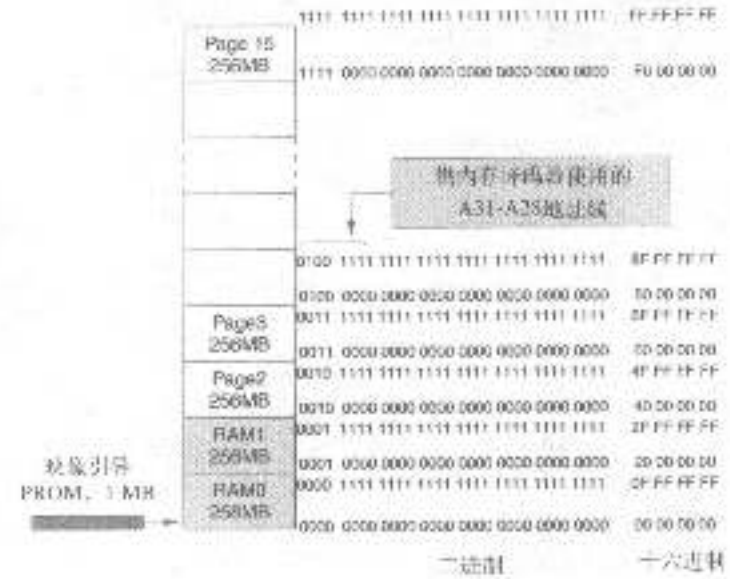


图6-15　4 GB（32根地址线）内存的组织

设计计算机的存储系统，使之为用户提供尽可能快的访问时间，同时为管理员提供最大限度的灵活性，使它们可以方便地增加模块，是一项复杂而且需要很高技巧的工作。译码电路、电压供给、刷新率和时钟脉冲时序都需要配置。这些都使得硬件设计人员的工作更难重重。微控制器，以及一些CPU，在芯片上提供内存译码和脉冲生成电路，就是为了协助计算机系统设计中这方面的工作。

存储芯片有好几种宽度。DRAM芯片在制造时采用1、4或8位的基本存储单元，而SRAM和EPROM一般为1字节宽。如我们在图6-11中所见，8字节宽的DRAM可以合并到一起，形成64位的DIMM插入到奔腾的总线中。对于数据总线仅为一个字节宽的简单计算机系统，每个存储芯片封装都直接连接到所有8条数据线上。但控制线并非如此，只有需要的控制线才连接到存储芯片上。就地址线来说，从最低位的A0开始，只有那些对从内部阵列中选择基本存储单元所必需的线路才会连

接。所有的封装还拥有一个芯片选择（CS）输入，它可以激活芯片，常用来将芯片映射到为它们分配的地址。

RAM或ROM存储芯片，一般并不与地址总线的宽度匹配。由于不同计算机上的地址宽度也各种各样，实际上也做不到这一点。由于存储芯片的地址引脚要少于CPU地址总线的条数，故而CPU可能发送32位的地址，而RAM封装只能直接接受20位的地址。这就要求设计人员必须仔细规划，设计出**内存地址译码**（Memory Address Decoding）电路的规格说明。该电路使用空余的高位地址线，将它们送给单独的译码器，由这个译码器控制存储芯片封装上的芯片选择（CS）引脚，来完成寻址。任何未用的高位地址线可以留待将来扩展之用。

不对地址空间进行完全译码的做法十分常见。这就导致内存封装会在地址空间内有多个映像。一般地，如果每个人都了解这种情况，并且不去在“已经占用”的位置插入其他器件，一般不会产生任何问题。

一些计算机系统将IO端口安排到主存的映射表中。图6-16说明了这种情形。这会给程序员的地址空间带来不方便的分段。同时，内存和端口芯片访问时间的差异，有可能会使译码电路变得极为复杂，而且对总线周期时序的要求也更为严格。

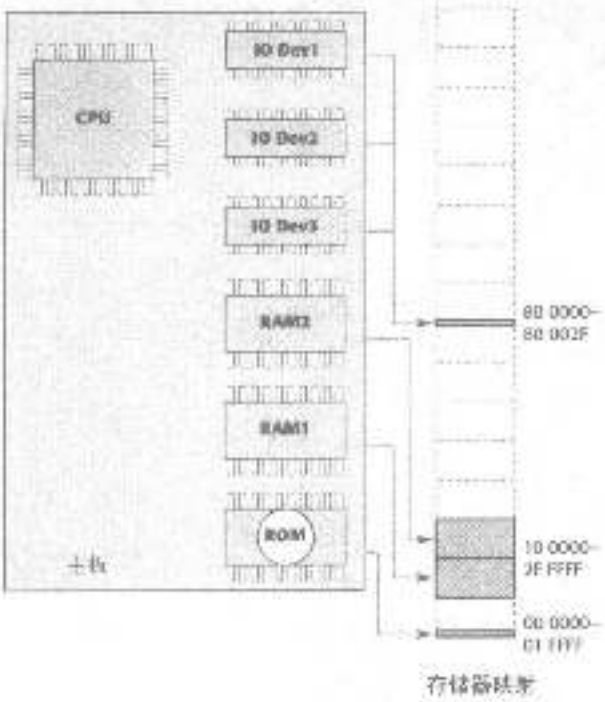


图6-16 IO存储器映射的布局

在纸上画出内存区域的分布，就可以清楚地看出采用各种不同译码方案时，地址空间的使用效率！

6.8 IO端口映射

至此，我们已经论及将IO端口看做存储芯片映射到同一寻址空间内的情况。这种方法称为**存储器映射IO**（memory-mapped IO）。其他计算机，最明显的是Intel奔腾系列，将寻址空间分为两个区，分别服务于内存和IO，见图6-17。为了实现这种方案，CPU必须提供专门的指令，如IN和OUT，来完成CPU和IO端口之间的数据传送。完成IO传送还需要引入不同的总线信号。由于端口出现在不同于内存的地址空间内，故而我们称其为**IO映射**。奔腾计算机中，内存空间要远大于IO空间——4 GB

对64 KB，因而，内存地址为32位长，而IO地址仅16位长。这两种不同的方案各有优劣，需要进一步加以论证。由于IO端口和内存保持分离，因此能够保持主存为单一的连续块，这是程序员所乐于见到的。然而，由于IO指令和与之兼容的寻址方式要远少于常规的面向内存的组，这使得程序可能会稍长一些。不过，使用面向内存的指令（范围更广）处理IO操作所带来潜在好处，并不如预想的那么大。许多设置和清除内存中某个字节单个位的机器指令，看起来适用，但实际上不能用于操作外设芯片的寄存器。这是由于这类寄存器常常为“只写”，使得“读-处理-写”指令序列不适用。下面这组MC68000指令就受到这类限制。

```
ori.b #bmask,OP_reg ; logical OR a mask to set a port bit
andi.b #$f7,OP_reg  ; logical AND a mask to clear a port bit
asl.b (a5)          ; shift port bits left for display purposes
not.b OP_reg    ; shift port bits right for display purposes
bclr #1,OP_reg ; test a port bit and leave it 0
bset #2, (a2)   ; test a port bit and leave it 1
```

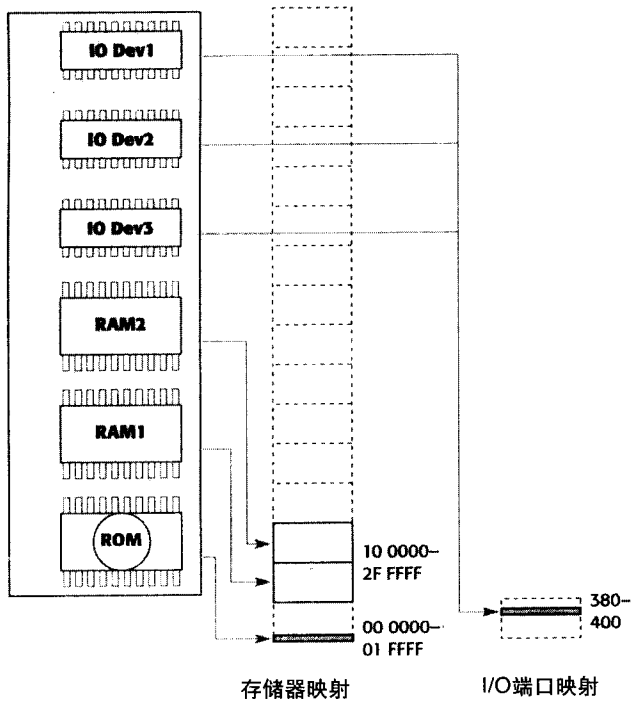


图 6-17

所有这些指令都是从指定的位置将数据读入CPU，按照要求进行修改，然后将它们写回到原来的存储单元。对于只能写入的硬件寄存器，这是不可能的。在处理常规的数据变量时，编译器将会使用这类指令，但遗憾的是，汇编器和编译器都不能将对存储器映射中外设芯片的访问区分出来，因而，程序在运行时可能会产生问题。

一般地，IO总线周期要比等价的内存周期更长一些，这可能要归因于IO芯片的制造工艺相对落后。使用不同的IO总线信号，使得硬件工程师在实现相应的电路时要容易一些。早期的8位微处理器刚刚出现时，人们常说程序员喜欢Motorola，而电子工程师则喜欢Intel！有时在差异性的系统上，低级调试更容易进行，因为可以更全面地设置断点和错误陷阱。之后，对输出端口的任何访问都可以立即发现，并在需要时加以纠正。最终，在第21章讨论完转向RISC CPU构架所要做的一些技术调整之后，我们就能够很容易地认识到，任何不必要的、额外的指令都会加重CU的负担，对性能

造成很大的影响。通过将IO端口映射到内存中，所有其他指令的执行都能够加快。

6.9 小结

- 可行的最简单的1位基本存储单元可以用两个交叉耦合（cross-coupled）的NOR门构成。我们称之为S-R锁存器。它和用来构建SRAM（常用做高速缓存）的电路相关。
- 人们使用各种技术制造具有不同特性的存储器件：DRAM、SRAM、EPROM、闪存、EEPROM。当前，数字计算机的主存由SDRAM（同步DRAM）芯片组成。
- 动态RAM使用小型的集成电容储存电荷来表示二进制1。遗憾的是，电容会漏电，因此必须定期刷新。
- DRAM一般以DIMM的形式出现，我们将它插在主板的插槽中。其访问时间和周期时间需要与系统时钟速度相匹配。为了提高内存的整体访问效率，现在，数据一般以突发的方式进行读写，而不是以单个字节为单位进行。
- 当整个地址空间（由地址宽度决定）没有完全被占用时，就要用到内存映射。那些未使用的高位地址线用来选择正确的存储模块。
- IO端口既可以映射到内存中，也可以安装在独立的IO地址空间中，如果为后者，则需要用专门的指令访问。
- 同时执行几件事能够增加吞吐量。一种做法是，预测对指令的需求，将它们从内存中预先读取出来。
- 最大内存长度由地址宽度决定。
- 在处理多字节数据时，首先需要确定字节的次序。
- 从程序的角度看，简单的输入输出端口可能作为内存地址空间内的单个字节出现。

实习作业

我们推荐的实习作业包括了解和设计新的存储器映射。如果对微计算机电子技术方面的内容感兴趣，可以试着设计译码器电路。

练习

1. CPU数据寄存器和内存中的存储单元之间有什么不同？
2. 在写周期内，如果两个RAM芯片同时响应该地址，会发生什么情况？
3. 奔腾CPU支持的最大物理内存是多少？奔腾处理器支持的最大虚拟内存是多少？
4. 访问单字节宽的24 MB内存，需要多少条地址线？
5. 访问200 MB的磁盘，需要使用多少条线路？
6. 1 M位PROM封装的最小引脚数是多少？
7. 使用上述芯片设计一个2 MB的PROM插卡。
8. Atari插卡端口拥有两组ROM接口，提供地址线A0-A15。那么，插入式游戏程序的最大尺寸是多少呢？
9. 我的Sun UltraSPARC工作站有64 MB的RAM。服务于该内存空间的地址总线的最小宽度是多少呢？64位SPARC CPU能够处理多少额外的64 MB的DRAM呢？
10. 处理器所支持的物理内存如何能超过由程序计数器的宽度所决定的最大范围？
11. DRAM中，访问时间和周期时间之间的关系是什么？为了避免周期时间带来的延迟，人们常常采用什么技术？
12. 地址总线为32位的CPU，填满其内存空间需要花费多少？——通过网络查找DIMM的价格。
13. 叙述使用同步或异步内存访问周期的优缺点。

14. DRAM为什么需要刷新？多长时间刷新一次？在标准PC上如何完成？估算单个DRAM基本存储单元上存储了多少电子（$Q = V \times C$；$q_e = 1.6 \times 10^{-19}$C；假定$V = 1.2$V）。
15. 在32位地址空间内，可以装入多少64 MB大小的页？
16. 64 MB的内存可以划分成4个16 MB的页。选择单独的页需要用到哪些地址线？
17. 在奔腾系统上，IO空间有多大？提供独立IO寻址空间的CPU有什么优点？

课外读物

- Tanenbaum（2000），3.3节：introduction to memory components。
- Heuring和Jordan（2004），RAM的结构，SRAM和DRAM。
- Patterson和Hennessy（2004），关于存储单元的附录。
- 有关存储芯片制造的更多技术信息，参见：

http://www.micron.com/K12/semiconductors/hrw.html

- Tom硬件指南也给出一些有用的细节：

http://www7.tomshardware.com/guides/ram.html

- 这些网址可以通过本书的配套网站访问：

http://www.pearsoned.co.uk/williams

第7章　Intel奔腾CPU

奔腾微处理器为大多数PC设备提供处理能力。作为程序员，了解一些有关CPU内部结构的知识十分有益。本章对CPU寄存器、基本指令集、状态标志位和寻址方式都做了一定程度的论述，因为它们都对HLL程序的执行效率有所影响。同时还给出使用Microsoft Developer Studio编写汇编例程的一些必要的指导，因为进行汇编级别的编程，是了解计算机系统运作的最好方式。

7.1　奔腾：高性能的微处理器

在介绍新的事物时，最好从简单的例子开始，但对于微处理器来说，这越来越难以做到，因为在对提高性能的不断探寻中，微处理器电路的设计越来越复杂。本章中我们将不使用简化的"理想模型"，而是尝试论述奔腾处理器本身（尽管这样会引入额外的复杂性），讲述那些对程序员最重要的方面。由于所有新的PC几乎都使用微处理器，Intel，IDT，AMD和Cyrix等都设计制造出各种变体和具体应用，从而使它已经成为到目前为止最常见的处理器。实际上，Intel常常在电视的高峰时段大张旗鼓地为它最新的奔腾处理器做宣传。从历史的角度看，奔腾（Pentium）不过是Intel 80x86系列中最新的CPU，就它的寄存器结构来看，依旧与Intel最初在1975发布的8080微处理器有一定程度的类似性。基于这一点，它被认为是最快的8位处理器——尽管这有些不太公平！

这些年来一个巨大的变化，就是微处理器的封装和散热。当时，8080被封装到40针的双列直插式（dual-in-line，DIL）塑料封装内。如今的奔腾4芯片被封装到大得多的方形针状阵列封装中，从底部引出多达478根左右的针脚（见图7-1）。它需要散发约50到80W的热能，相对于只有25平方厘米的表面区域，它甚至高于我的家用取暖器，取暖器单位面积散发的热量是750W/m^2，而以60W计，奔腾处理器达到60W/25 × 10^{-4}m^2 = 24kW/m^2。为了驱散这些过多的热量，就型散热风扇取得了令人瞩目的发展。赛扬和Athlon处理器的风扇-散热器组合是由钢制的弹性夹片固定在ZIF插座上。有时，当用来固定的小塑料在突起盖压下所裂后，风扇-散热器组合可能会脱落，从而造成处理器温度急剧上升，到达危险的程度。由于奔腾4要求使用更重的风扇-散热器组合，因此它提供四个螺丝固定孔，通过主板固定到金属机箱上。这种方法明显优于之前的方案。

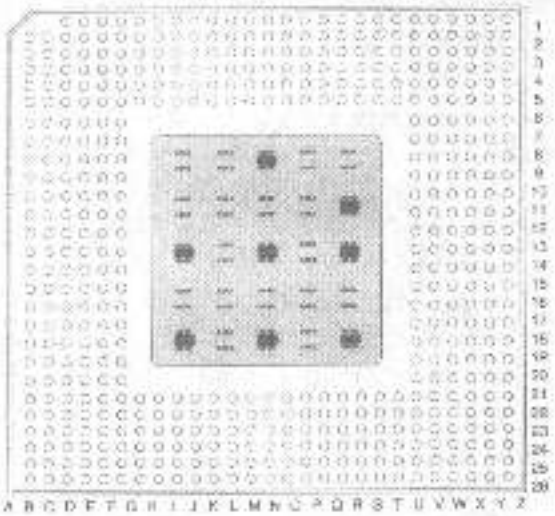


图7-1　Socket 478针型栅格阵列封装，Intel 2.5 GHz奔腾4

1975年以后的这些年间，Intel奔行微处理器产品保持二进制后向兼容的策略，这意味着现有的程序应该能够不需重新编译源代码，就可以运行在新的处理器上。最开始，这种功能好像无关紧要，因为在微处理器的早期应用中，每个用户都编写自己的软件，也就是说，他们对源代码拥有控制权，从而能够为市场上出现的任何新处理器重新编译现有的软件。但是，后来的情况发生了翻天覆地的变化。现在，许多PC用户拥有大量二进制格式的应用软件库，但拿不到源代码，从而，在决定是否升级处理器时，重复更换这样的库文件所需的成本，成为重要的考虑因素，从而使后向二进制兼容

有了重要的市场意思。对CPU的制造商而言，硬件升级方面的任何潜在阻力，都不利于其营收状况。

从CISC 8086逐渐进化到奔腾的过程中，Intel不断尝试将一些当代RISC技术好的方面引入进来，并且依旧保持了与x86系列早期成员的二进制兼容。例如，相对简单的指令现在均由硬件逻辑电路进行译码，这要快于x86 CPU中使用的微码技术。Intel能够在处理器中除包括微码引擎之外，还另外包括一对硬件逻辑译码器（见图7-4），通过这些，我们能够感受到近年来在VLSI制造技术上的进步。这使得奔腾的CU依旧使用微码处理相对比较复杂的i486指令。通过这种方式，Intel力图得益于RISC技术，同时依旧保持CISC的灵活性。

Intel最初的8088处理器——用在早期1980年的IBM PC中，仅工作在4.77 MHz的时钟速度。同时代的主存DRAM完全能够在四个时钟周期内做出响应（839 ns=4 × 1000/4.77），能够满足总线控制器的需求。但是，从那时起到现在，处理器时钟速度已经提高了500倍，但DRAM的访问速度只提高了10倍左右。这意味着如果现代的微处理器直接从主存中读取数据或指令，它可能要等待25个时钟周期，才能接收到从内存中请求的数据。这种延迟显然不符合我们的需要！一种可能的解决方案是，将DRAM换成快速的SRAM，但这样会使价格提升很多，同时由于SRAM基本存储单元大于DRAM基本存储单元，因而所需的电路板面积也要增加。当前广为采用的解决方案是，使用相对较少数量的快速SRAM作为CPU的本地高速缓存。

大多数高性能的现代计算机，包括PC/奔腾这一组合在内，都是成本和效能之间的折衷。从第6章我们知道，DRAM更廉价，但访问速度明显慢于SRAM。通过提供相对较少数量的SRAM作为高速缓存，可以部分弥补大容量DRAM主存缓慢的访问时间。我们在第17章会彻底全面地讲述计算机内存的组织，在此给出奔腾系统的内存布局，仅供参考。

奔腾处理器的地址总线依旧是32位宽，但主数据总线扩展到64位，内部的CPU总线更是达到128位或256位的宽度。拓宽数据的通道能够让慢速的内存传送足够的数据，不至于让CPU等待。但是，宽而慢的通道需要在接收端拥有足够的数据缓冲区。这就类似于大容量、低速度的运输模式，比如海轮——提供商依靠仓库来平滑货运的高峰和低谷。奔腾CPU提供一个高速的芯片内缓存，称为一级高速缓存，在奔腾Ⅲ上它被组织成两个独立的16 KB的缓冲区。一个负责数据，另一个负责指令。缓存还有一种特殊的“突发式访问”功能，以加速数据的传递。使用四个连续的DRAM总线周期，可以将32字节的数据移入或移出高速缓存。这样的一块数据称为“缓存行”（Cache Line）。

正是有了保持在高速SRAM（和CPU制造在同一片硅晶圆上）中的8 KB指令，才使得CPU的性能得以稳步地提高，但在奔腾4上面，设计人员对这种高速缓存结构做了改进，L1数据高速缓存的大小被削减为8 KB，另外提供更大的执行跟踪高速缓存（Execution Trace Cache，ETC），可能是96KB，取代之前的L1代码高速缓存。L1和ETC之间主要的差别在于它们保存的内容：L1高速缓存保存x86机器码，而ETC保存的是部分译码的微操作指令，可以直接送入执行流水线中。8 KB的数据高速缓存依旧可以加速基于堆栈的局部变量的访问，提高系统的整体效能。

早期的i486处理器只有单一的高速缓存。通过引入RISC风格的流水线译码方案，流水线的不同阶段肯定会对高速缓存SRAM有并发访问需求。通过将高速缓存分成两部分，奔腾降低了这类访问冲突发生的数量，从而也就避免了相关的延迟。遗憾的是，大部分程序在编译成代码后，都远大于一级高速缓存提供的8 KB，因此，所有近期的CPU系统，现在都包括二级（Level 2，L2）SRAM高速缓存，称为二级高速缓存，它位于L1 CPU高速缓存和主存之间。需要注意的是，高速缓存的访问时间会随其大小的增长而增长，因此，总的说来，最好将高速缓存组织成分层次的独立单元。奔腾Ⅱ为了获得更好的性能表现，将这个L2高速缓存和CPU一同隐藏在Slot 1封装内。CPU和高速缓存的紧密集成，在奔腾III和奔腾4处理器上继续发展。与此同时，通过DRAM突发式访问技术，主存的访问时间在不断加快，慢速主板从L2高速缓存获得的性能优势越来越不明显。

Intel内部使用的一些项目名现在经常出现在一些营销性质的文字中（见表7-1）。赛扬就是其中之一，这个名字最初是指处理器电路板没有L2高速缓存的奔腾II CPU Slot 1处理器（见图7-2）。这一产品是一款面向家用市场的廉价产品，一直以来，这一市场一直为AMD的K6-2所主宰。后来，

Intel引入了Mendocino版本的凌扬CPU，时钟频率达到333 MHz，并且在CPU芯片上加入128 KB的L2高速缓存。由于CPU和L2高速缓存之间的总线能够以处理器的时钟速度全速运行，因此它的性能远远超过预期。由于Slot 1处理器电路板仅支持这种CPU，因此Intel决定重新回到插座，放弃了更为昂贵的奔腾II插槽。更大的478插座（见图7-3）现在已成为Willamette和Northwood版本奔腾4的接口。实际上，超过200根针脚只是为处理器芯片输送电力，所以处理器变热也就不奇怪了！其余的针脚是用于系统控制和中断线路。

表7-1 奔腾处理器的开发项目名

名 称	处理器
P24T	486Pentium OverDrive, 63 or 83 MHz, Socket 3
P54C	Classic Pentium 75-200 MHz,Socket 5/7, 3.3 V
P55C	Pentium MMX 166-266 MHz, Socket 7,2.8 V
P54CTB	Pentium MMX OverDrive 125+, Socket 5/7, 3.3 V
Tillamook	Mobile Pentium MMX 0.25 µm, 166-266 MHz, 1.8 V
P6	Pentium Pro, Socket 8
Klamath	Original Pentium II, 0.35µm, Slot 1
Deschutes	Pentium II, 0.25 µm, Slot 1, 256 kbyte L2 cache
Covington	Celeron PII, Slot 1, no L2 cache
Mendocino	Celeron, PII with 28 kbyte L2 cache on die
Dixon	Mobile Pentium II PE, 256 kbyte L2 cache on die
Katmai	Pentium III, PII with SSE instructions
Willamette	Pentium 4, L2 cache on die
Tanner	Pentium III Xeon
Cascades	PIII, 0.18 µm, L2 cache on die
Coppermine	PIII, 0.18 µm, 256 kbyte L2 cache on die
Merced	P7, First IA-64 processor, L2 cache on die, 0.18 µm
McKinley	1 GHz, improved Merced, IA-64, 0.18 µm, recipe improvements
Foster	Improved PIII, IA-32
Madison	Improved McKinley, IA-64, 0.13 µm
Prescott	P4,90 nm
Northwood	P4, 90 nm, 12K microOP
Tualatin	P4, 130 nm, Pentium III, Coppermine core
Itanium2	32 kbyte L1, 256 kbyte L2, 3 Mbyte L3 cache

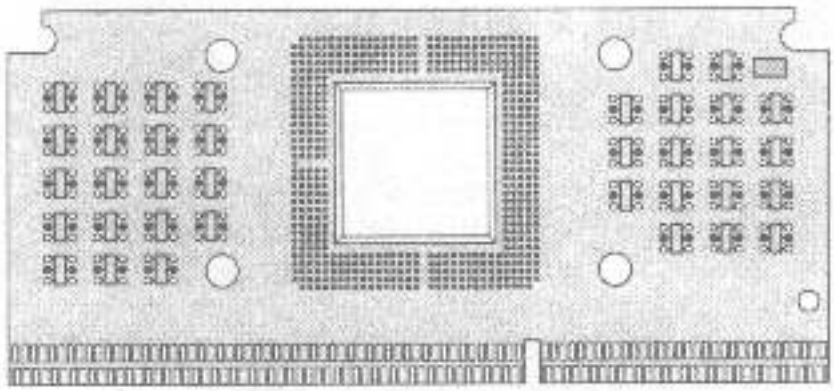


图7-2 Intel奔扬系列奔腾II处理器模块

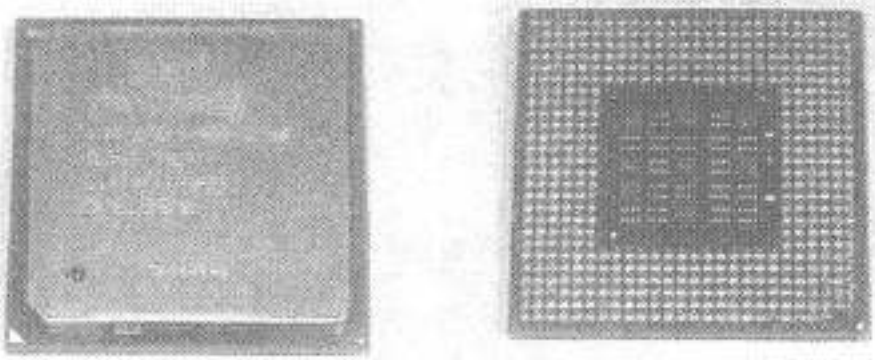


图7-3　478针封装的Intel奔腾4

AMD曾成功地在桌面处理器市场引入K6-3处理器，它有64 KB的L1高速缓存，以及256 KB的芯片内L2高速缓存。自此以后，AMD似乎走到Intel前面，它推出了Athlon、Duron，以及高速缓存达到服务级的Opteron-64处理器。

更让人印象深刻的是，奔腾处理器在芯片上提供分段和分页内存管理的电路，有关的细节将会在第12章提供。但是，最好记牢（当我们查看虚拟内存管理方案时），用来访问高速缓存的地址是实际的32位物理地址。

奔腾还支持分支预测逻辑（Branch Prediction Logic，**BPL**），以及与之相关的查询表，参见图7-4。在第21章中将会进一步讨论这个话题。就现在来讲，可以将其理解为在指令进入译码器流水线时，检测任何条件性指令，如IF...ELSE或CASE语句，并试图猜测程序最后会选择哪条路径执行。由于预取单元（Pre-fetch Unit）预先读取指令，因此读到恰当的指令至关重要。



图7-4　奔腾CPU的框图

7.2　CPU寄存器：数据和地址变量的临时存储区

如前所示，奔腾处理器巧妙地结合新一代RISC构架在速度上的优点，以及CISC电路的后向兼容性。复杂的专用寄存器明显来源于其前身——CISC构架的8086处理器。在第21章我们会看到，纯RISC的设计提供更多的寄存器，但很少指定专门的用途。奔腾CPU的基本寄存器系列从80386以来就没有改变过。尽管经过这么多年的发展，从奔腾处理器的寄存器设置中，依旧可以看出最初8080寄存器的影子！诸如CPU寄存器集，以及如何使用它们的细节知识，对于HLL编程来说，当然不是必需的。但对于计算机体系结构的学生，这些知识必不可少，它们有助于我们理解各种计算机性能上的差异，进而对底层的活动有更清晰的认识。

图7-5中的CPU寄存器不包括浮点寄存器，以及MMX（Multi-Media Extension，多媒体扩展）指令，这些将在后面第20章的第8节论述。尽管我们尚未全面地介绍奔腾的指令集，但为了展示CPU寄存器的使用，这里还是列出了一些以汇编语言助记符表示的指令示例。要注意，Intel汇编语言助记符按照HLL赋值规则来表示数据传送：从右到左（有可能会发生混淆，因为Motorola和Sun

汇编语言助记符是以相反的方向来编写的。）

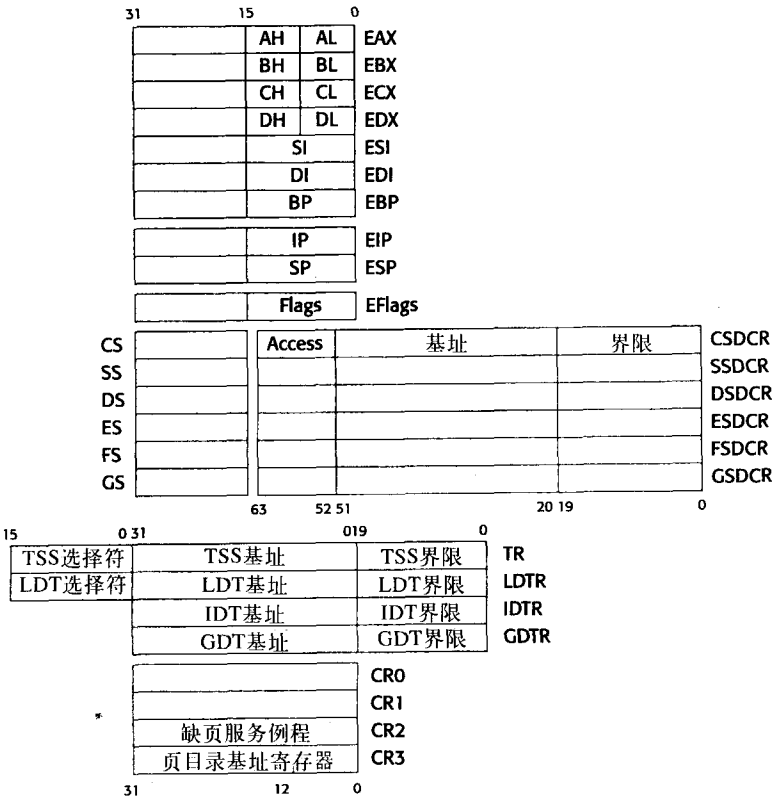


图7-5 Intel 80x86/奔腾CPU寄存器集

EAX：现在有时仍会称之为累加器（Accumulator），在执行算术和逻辑运算时，用做通用数据寄存器。一些指令针对A寄存器做了优化，使用该寄存器时执行会更快。另外，所有与输入-输出端口相关的数据传递都要用到A。它还可以用不同的方式来访问，比如8位、16位或32位，分别用AL、AH、AX或EAX来表示。对于乘法和除法指令，它是一个“隐含”的角色，即虽然在运算中大量使用，但并不在指令中专门提及。这样的限制实际上只有汇编级别的程序员或编译器的编写人员才会关心。

```
MOV EAX,1234H        ; 将常量4660装入32位累加器中
INC EAX              ; 将累加器中的值加1
CMP AL,'Q'           ; 比较ASCII Q与EAX最低字节的值
MOV maxval,EAX       ; 将累加器的值存入内存变量maxval
DIV DX               ; 用16位D寄存器的值除累加器
```

EBX：基址寄存器，可以保存指向数据结构（如内存中的数组）基地址的地址。

```
LEA EBX,marks        ; 用变量marks的地址初始化EBX
MOV AL,[EBX]         ; 使用EBX作为内存指针，将字节值取到AL中
ADD EAX,EBX          ; 将EBX与累加器相加，结果存入累加器
MOV EAX,table[BX]    ; 使用BX的值作为数组索引，从table数组取32位的值
```

ECX：计数寄存器，它有一个特殊的作用，就是在循环或移位操作中作为计数器。

```
    MOV ECX,100      ; 初始化ECX，作为循环的索引
    ...
for1:                ; 符号地址标签
```

```
...
LOOP for1          ; 递减ECX，检测是否为零，如果非零则跳回继续执行
```

EDX：数据寄存器，在执行输入-输出数据传递或执行整数乘法和除法时，可能会用到它。其他时候可以用它来保存变量。

```
IN AL,DX           ; 从端口输入字节值，DX中存储的是16位的端口地址
MUL DX             ; A乘以D中的值
```

ESI：源变址寄存器（Source Index），用做数据段内字符串或数组操作的指针。

```
LEA ESI,dtable     ; 用变量dtable的内存地址初始化ESI
MOV AX,[EBX+ESI]   ; 使用基址寄存器和变址寄存器取一个16位的字到AX中
```

EDI：目的变址寄存器（Destination Index），用做数据段内字符串或数组操作的指针。

```
MOV [EDI],[ESI]    ; 从内存中的源存储单元向目的存储单元传送一个32位的字
```

EBP：堆栈结构基址指针寄存器（Stack Frame Base Pointer），用做堆栈框架的指针，为HLL过程操作提供支持。它一般作为堆栈段内的偏移来使用。

```
ENTER 16           ; 将EBP保存在堆栈上，复制ESP到EBP中，然后从ESP中减16
```

EIP：指令指针（Instruction Pointer，也写作Program Counter，即程序计数器），保存当前代码段中下一条指令的偏移地址。

```
JMP errors         ; 强行将EIP设为一个新地址
```

ESP：堆栈指针寄存器（Stack Pointer），保存当前堆栈段栈顶的偏移量。

```
CALL subdo         ; 调用一个子例程（subdo），将返回地址存储在堆栈上
PUSH EAX           ; 将累加器中的32位值存到堆栈上
```

EFLAG：标志寄存器，保存CPU的状态标志，和所有条件性指令有关。

```
JGE back1          ; 测试符号标志，条件跳转
LOOP backagin      ; 测试零标志，循环退出条件
```

CS-GS：引入这些16位的段选择寄存器，最初是为了扩展8086处理器的寻址范围，同时依旧保持16位的IP。即通过为IP增加一个段寄存器，扩展其寻址范围。由于新的段寄存器也只有16位，因此，在使用之前首先要将它们左移4位，以得到20位的寻址范围（2^{20} = 1 MB）。但是，每个单独的段——代码段、堆栈段、数据段以及其他三个附加数据段——依旧受限于16位IP的限制，最大只能64 KB。这种讨厌的限制（它也被错误地引入到MS-DOS操作系统中），强迫程序员多年来一直得使用一种曲折的寻址方式，在全32位硬件引入之后，还持续了很长一段时间。现在，对于奔腾处理器，段寄存器可以看做是扩展内存管理的一部分，当奔腾处理器在保护模式运行时，利用它们可以将主存储空间划分成段。段选择寄存器不再是段基地址的指针，因而也就不再直接参与有效地址的生成。它们现在分别指向段描述符表（GDT和LDT）中的相关项。之后，这些选定的项被读入到相应的64位高速缓存寄存器（CSDCR），从那里可以读入基址指针，和EIP一起形成有效地址。

CSDCR-GSDCR：这些寄存器对程序员不可见，但它们对MMU的运作至关重要。在常规的保护模式下运行时，64位代码段描述符缓冲寄存器（Code Segment Descriptor Cache Register，CSDCR）保存当前的代码段描述符，包括基地址、大小限制和访问权限。段描述符通过内存中的全局或局部描述符表获取。由于每次内存访问都需要用到这些值，因而提供快速、容易的访问很重要。CU发出的每个有效地址都要加上相应的段基地址，同时进行界限和访问权限的检查。

TR：任务寄存器（Task Register，TR），保存当前任务的16位段选择符、32位基地址、16位的大小限制以及描述符属性。它指向全局描述符表（Global Descriptor Table，GDT）中的一个TSS描述符。在任务切换时，任务寄存器会自动重新载入。

IDTR：中断描述符表寄存器（Interrupt Descriptor Table Register，IDTR），保存当前中断向量表

(Interrupt Vector Table，IVT）的基地址和大小限制。它是48位的寄存器，低16位专用于大小限制的值，高32位保存IVT的基地址。这允许奔腾处理器重定位默认起始位置为0000 0000的1 KB的IVT。这种机制在开发过程中十分有用，同时，它使得Windows NT能够同时提供多个“虚拟机”环境。

GDTR：全局描述符表寄存器（Global Descriptor Table Register，GDTR），保存段描述符，它们指向全局可用的段，以及保存局部描述符的表。

LDTR：除全局描述符表以外，每个任务还可以使用一个局部描述符表（Local Descriptor Table Register，LDTR）。这个寄存器表示使用局部段描述符表中的哪一项。

CR3：控制寄存器（Control Register，CR），指向分页单元（Paging Unit）的目录表。

CR2：这个控制寄存器指向处理页错误的例程，当CPU试图访问的地址属于不在内存中的页时，就会发生页错误。服务例程将会激发磁盘操作，将页从磁盘调回到主存中。

MM0-MM7（未在图7-5中标出，参见图20-20）：这些是由MMX（Multi-Media Extension）指令使用的8个扩展数据寄存器。奔腾处理器有大约50条左右的MMX指令，它们的目标是通过同时执行几条指令加速算术运算。MMX数据寄存器为64位宽，因此，单条指令能够载入和处理4个16位整数。这有助于处理与图形相关的矩阵计算。

FP0-FP7：这些64位寄存器没有在图7-5中给出。它们专供FPU ALU使用，从而使浮点（实数）运算能够更快地执行。

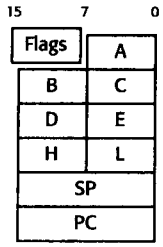


图7-6 Intel 8080 CPU的寄存器集

相比较而言，1975年Intel 8080的寄存器集（见图7-6）现在看起来真是微不足道。

在8080中能够看到奔腾通用寄存器的起源吗？比较图7-6和图7-5，您可能会感到吃惊。尽管这些寄存器的32位扩展已使得奔腾处理器更加强大和用途广泛，但很明显，在比较这两个寄存器集时，我们会发现，绝大多数发展都集中在内存管理系统上。在第12章中将讨论更多这方面的内容。

7.3 指令集：基本奔腾指令集简介

所有计算机都有一套机器指令来执行各种各样的动作，以操作数据及控制硬件的运作。前面的小节中，在说明CPU寄存器的应用时，已经用到奔腾指令集中的几条指令。实际上，奔腾处理器拥有超过200条的不同指令，同时，几种不同的寻址方式进一步扩展了它们所能够执行的操作。表示这些指令的二进制代码不是定长的，其长度从1到15个字节不等，这有助于降低程序的长度，因为“常用”的指令都被赋予比较短的代码。随CPU构架向RISC转变，以及存储器容量的增长，变长指令代码已经失宠。

表7-2 机器指令分类

1. 数据传送（复制）
2. 数据输入输出操作
3. 数据操作
4. 控制转移
5. 机器管理

每条机器指令都可以归类到表7-2中给出的五种类别中的一种。

数据传送指令组有时也可以用来执行输入输出操作，Motorola 68000即如此，它惟一专门处理数据IO的指令就是MOVEP。提供它只是为了协助单字节宽度的外设端口，并非十分有用。Intel选择为它的处理器（包括当前的奔腾处理器）提供专门的IO指令，即IN和OUT。数据操作指令，除普通的ADD、SUB、MUL和DIV以外，还包括位逻辑操作AND、OR和NOT。控制转移指那些能够改变IP值稳步递增（CPU在执行读取-执行周期时IP执行的动作）的指令。IP的不连续变化表示程序发生跳转，原因多种多样，可能是由于过程调用，甚至有可能是不常用的GOTO语句。诸如JP和CALL等指令都能够造成这类动作。在尝试使用各种指令时，机器管理指令常常是最危险的指令。HLT指令可能需要硬件系统重置，才能恢复回来，而中断阈值（interrupt threshold）的改变可能会使系统得不到所需的输入。即使不恰当地改变了内存中的全局描述符表，也可能会让整个系统崩溃。

在学习新语言时，未知词汇的数量可能会令人生畏。表7-3简要地列出了一部分基本的奔腾指令，它有助于我们理解本书的例子，并开始熟悉汇编语言程序设计。如果觉得18条太多，甚至可以将它缩减到14条。

表7-3　奔腾处理器的基本指令

MOV	在存储单元、寄存器和内存间复制数据
LEA	载入有效地址
CALL	调用子例程
RET	从子例程返回
PUSH	将一项内容压到堆栈上，可能是作为子例程的参数
POP	从堆栈上弹出一项内容
INC/DEC	递增或递减
ADD	算术整数加法
SUB	以2的补码表示的整数的算术减法
CMP	比较两个值，同时更新标志位。该指令进行一次减法，但不存储结果，只根据结果更改条件标志位
AND/OR/XOR	逻辑操作
TEST	位测试
JZ	条件跳转
LOOP	通过递减CX寄存器，实现FOR循环
ENTER	建立子例程（过程）堆栈框架
LEAVE	从子例程退出时清除堆栈框架
JMP	可怕的跳转指令
INT	触发软件中断，执行操作系统例程

7.4　指令的结构：CU如何理解指令

指令从内存中读入并插入到某个指令流水线之后，CU必须对指令的二进制模式进行译码，并开始每个阶段所必需的活动。这并非由第4章中介绍的单个译码器电路就能够完成，而是要求使用更为复杂的方案，因为这个过程需要5个步骤，需要分阶段顺序执行。大部分指令需要包含表7-4中列出的三块信息。

表7-4　机器指令的功能构成

1. 指令的动作或操作
2. 涉及的操作数
3. 结果的存放

为指定这三个完全不同的部分，机器指令被规划成不同的位字段，以包含操作必需的信息、操作数的位置和结果、操作数的数据类型。奔腾指令可以是1到15字节之间的任何长度，依所需的操作数及所采用的寻址方式而定。前导字段一个重要作用就是，确定当前指令的长度。但不管长度为多少，所有指令都必须指定动作和操作数。奔腾指令格式在图7-7中给出。

操作码字节的前6位标识基本的操作（ADD、AND、MOV等），D位表示后面的REG字段代表的是源操作数还是目的操作数。W位区别字节和字操作数，而将32位的字与16位的字区别开来，则需要引入前导字节，设置前导字节中相关段描述符中的DB位。通过提供“可选的”前导字节，80386维护了与其前代简单产品的后向二进制兼容性。MOD字段标识操作数是否有一个在内存中，或者全部在寄存器内。后一种情况中，R/M字段保存第二个寄存器的标识符，否则R/M是内存访问方式字段。

图7-8中给出了一些简单指令的编码结构。我们可以使用Developer Studio的调试器来检查任何指令的二进制值。

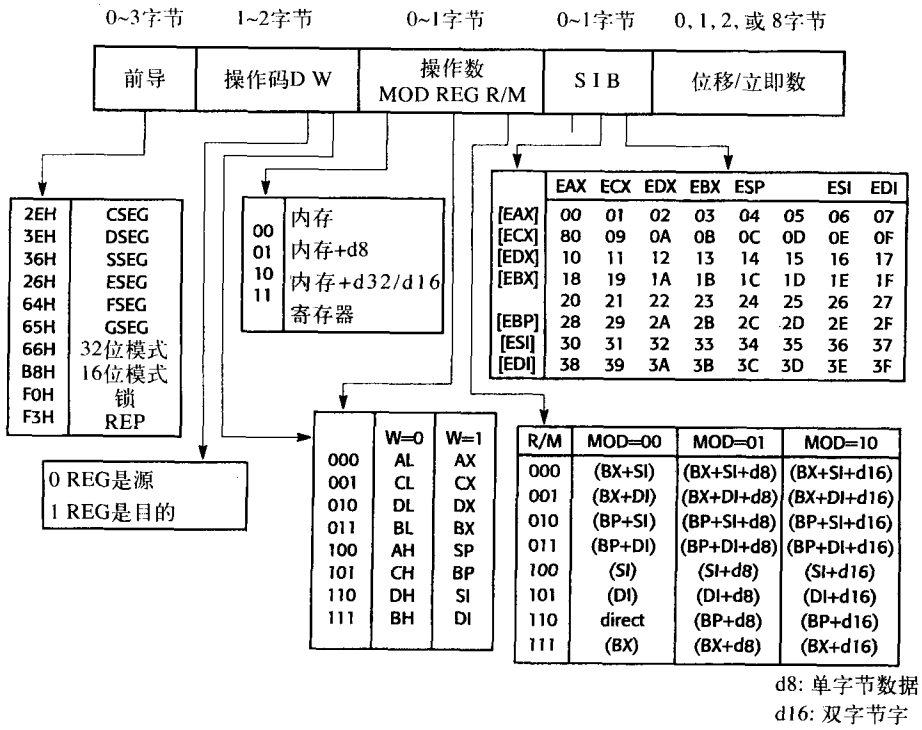


图7-7 奔腾指令中的5个字段

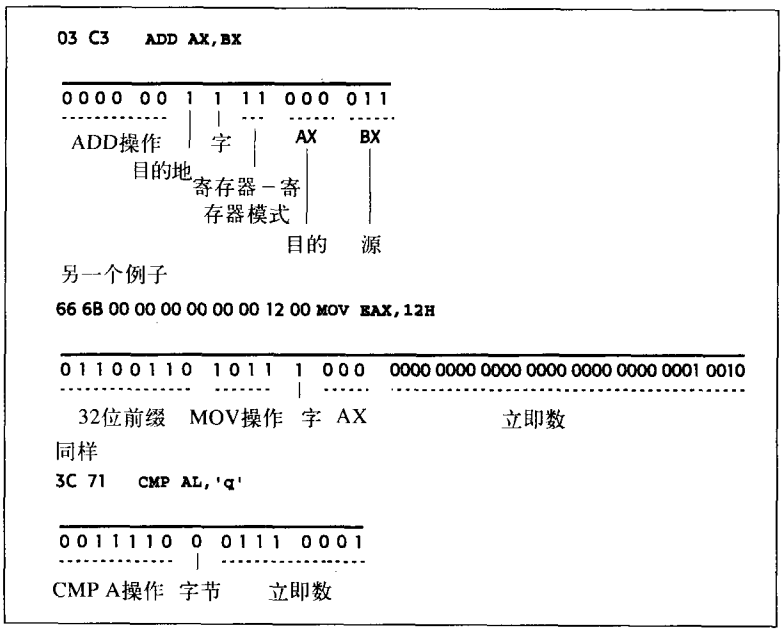


图7-8 奔腾指令的位字段

7.5 CPU状态寄存器：十分短期的存储空间

奔腾系统中，与所有数字计算机内一样，有一系列的状态标志位，它们被组织在一个CPU寄存器中，记录各种操作和事件的输出。一些CPU只在算术或逻辑运算后才会改变标志位。Intel的安排

是数据传送（MOV指令）不会影响任何标志位。状态寄存器有时被称为Eflags，因为随着奔腾的引入，它已经从16位扩展到32位。

如图7-9所示，CPU状态标志位，对所有数字计算机的运作起着根本性的作用。由于从根本上讲，冯·诺依曼体系是序列化的，一条指令接一条指令逐执行，因而存在将指令的结果与后续指令进行沟通的隐性需求。如果结果是一个数据项，则没有任何问题，因为数值要么放在寄存器中，要么存放在内存存储单元内。但如果指令的结果只是测试一个数，看它是否为零，那么将测试的结果写到另外的寄存器中就没有意义，因为随后还得对这个值进行测试，依次类推，测试永远没有完成的时候！

31																																0
												ID	VIP	VI	AC	VM	R		N	IOP	O	D	I	T	S	Z		A		P		C

ID	标识标志或CPUID的可用性
VIP	虚拟中断待决（Virtual Interrupt Pending）
VI	虚拟中断激活（Virtual Interrupt Active）
AC	对齐检查（Alignment Check）
VM	虚拟8086模式激活
RFR	在断点中断后重新开始任务
NT	嵌套任务（Nested Task）
IOPL	IO特权级别（IO Privilege Level）
O	算术溢出错误（Overflow）
D	访问字符串数组的方向（Direction（方向））
IE	外部中断允许（External Interrupt Enable）
T	Trap（解析），单步调试，执行每次指令后产生INT #1
S	Sign（符号），MS位的值
Z	Zero，表示结果为0
A	Auxiliary carry（辅助进位），BCD运算的低4位需要使用它
P	Parity（奇偶），操作数先状态
C	Carry（进位），表示算术进位或借位结果

图7-9　CPU状态标志位

*　在图7-10给出的例子中，CMP指令用来检查A寄存器中保存的字母变量是否为“q”。CMP指令就如同减法，不过它忽略所产生的结果，因而只有设置和清除CPU标志位的（次级）副效应在执行后保留下来。在种情况下，如果AL中的变量正好为“q”，则CMP指令执行后Z（zero）标志位会被设为1，因为71H－71H=0（“q”的ASCII码的十六进制表示为71H，二进制为0111 0001）。当下一个测试Z标志位的条件指令执行时，它所执行的操作由前面的CMP指令而定。这里，我们假定在CMP和JZ之间没有其他指令修改Z标志位的状态，从而破坏两者之间的连接。汇编级别的程序员可能花许多小时的幸福时光来调整这种敏感的交互作用，以获得速度或大小上的优势。

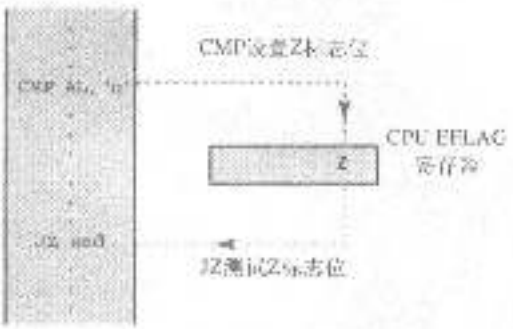


图7-10　CPU状态标志位如何工作

因此，对于可编程的机器，CPU状态标志位是其运作的中枢。它们将简单的机器指令结合起来，允许指令留下1位信息，引导后续指令的动作。我们还可以将CPU标志位看做是ALU和CU之间通信的一种方式。ALU设置标志位，CU测试它们。

7.6 寻址方式：构建有效地址

为了扩展机器指令的应用，使指令集更加灵活，更易于使用，CPU一般支持几种寻址方式。它们代表了形成地址操作数的不同方式，这样就能够对HLL操作大型数据结构时的需要提供更好的支持。MOD字段对指定CPU访问操作数时使用的寻址方式起到十分核心的作用。注意，如果指令有两个操作数，它还有可能混合使用两种寻址方式，两种寻址方式不必相同。

1. 数据寄存器直接寻址

```
MOV EAX,EBX
```

这种寻址方式执行最快，也最易于理解，用于将数据项从CPU寄存器读出或写入。这个例子中有两个寄存器直接寻址的操作数，机器指令通过3位的操作数字段标识是哪个寄存器（见图7-7）。在执行时，依据寄存器是源操作数还是目的操作数，数据项会被传递到该寄存器，或者由寄存器读出数据项。

2. 立即数寻址（IP间接寻址）

```
MOV EAX,1234
```

这种情况下，常量1234直接作为程序的一部分存储在指令之后。这种方式下，由于指令指针（EIP）在读取-执行周期间内是递增的，因而它会便利地指向数据，从而能够容易地从内存中读入进来。因此，立即数寻址方式也称为IP间接寻址方式。

3. 内存直接寻址

```
MOV EAX,[var1]
```

我们常常需要访问存储在内存中的变量。这种方式构建在立即数寻址方式之上，不过是在指令后存储变量的地址。IP同样用来指向保存该地址的存储单元，接下来，该地址会被读入CPU内临时的寄存器中。从那里，它被送回内存，以选择所要求的数据项，然后将数据读入到CPU内。简单地读取一个内存变量，就是一个十分复杂且极为耗时的操作！从内存中读数据的操作，其术语称为载入（loading），相反的操作——将数据写回内存中，称为存储（storing）。

注意，汇编程序能够将1234与[1234]区分开来，你也一定能够区分它们！

4. 地址寄存器直接寻址

```
LEA EBX,var1
```

载入有效地址指令使用立即数寻址方式将内存地址载入到寄存器中，常用于初始化指针，之后，就可以用该寄存器引用数据的数组。

5. 寄存器间接寻址

```
MOV EAX,[EBX]
```

在此，寄存器保存变量的内存地址。我们称之为“指向数据”。这个指令只需将这个地址从寄存器传给内存，将数据取回来。这是一种快速且高效的过程。这种寻址方式的一种重要扩展是自动将寄存器中的地址值递增或递减，将指针指向的位置在内存中向上或向下移动。奔腾处理器在每条指令的读取-执行操作中使用这种方式，每次读取完指令后，EIP寄存器都会自动递增。同样，PUSH指令自动递减ESP，而与之对应的指令POP，使用自动递增。对于奔腾处理器，程序员一般没有办法直接控制寄存器的自动递增/递减功能。

6. 带偏移的变址寄存器间接寻址

```
MOV EAX,[table+EBP+ESI]
```

并非一定要采用两个寄存器。例子中的符号“table”是数组的基地址。在处理一维数组时，这种方式也可以写成另一种对熟悉HLL的程序员来说更易于解读的形式。

```
MOV EAX,table[ESI]
```

7.7　执行流水线：RISC加速技术

在3.5节中，我们已经解释说明了在连续的处理序列中，将操作重叠对性能的提高立竿见影。RISC构架的CPU就利用了这项技术，它将预取缓冲区组织成一条生产线，换句话说，预取来的指令不再仅仅在队列中等待轮到自己译码并执行，而是随着在队列中的前进，不断部分译码并执行。这就称为流水线（pipeline），它对性能的提高作用显著，出人意料。图7-11给出了流水线结构的示意图。新的指令由预取器从存储器中读入，在进入流水线之前保存在缓冲区内。指令在通过流水线的过程中，就得以分阶段处理。对于给定的时钟速率，假定100 MHz，CISC控制单元可能使用5个周期完成每条指令，实际的有效效能不过25MHz。而采用流水线的RISC，每个时钟周期都可以完成一条指令，使得它具有$n \times 5$的速度优势。

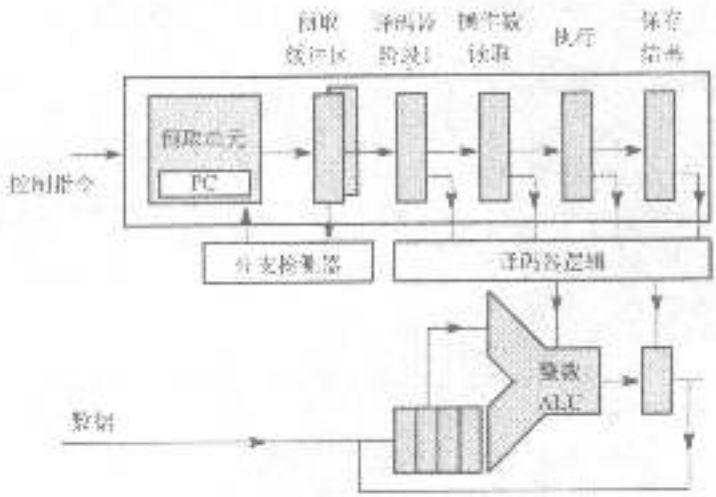


图7-11　采用流水线译码的CPU控制单元

5级的流水线应该拥有分别负责不同译码任务的阶段，如图7-12所示。在任何时钟周期内，流水线应该完成一条指令的执行，同时在前端读入另一条指令。为了加快时钟速率，我们需要简化每个时钟周期执行的操作，这样做的结果是，完成整个读取-执行周期所需的时钟周期数可能要增加。奔腾4为此采用了20级的流水线。

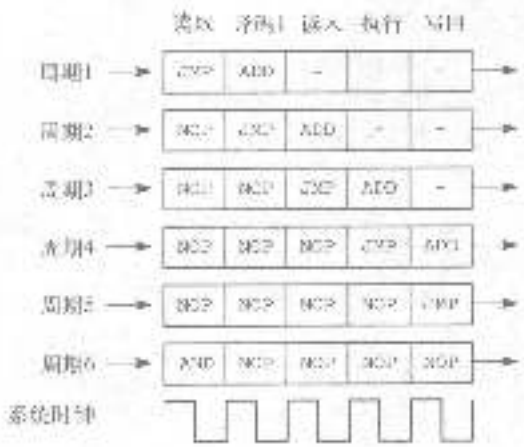


图7-12　指令译码的5级流水线

流水线可以看做是多道处理的一种形式。几个动作同时发生。为了能够管理这类并行机制，CPU硬件必需更为复杂，同时可能要提供一些重复的电路。但性能上的提高能够很容易抵消在这方面的损失。

至此，我们只规划单个指令流，由单个流水线化的译码器来处理。但为了更进一步地提高奔腾芯片的性能，Intel的设计工程师实现了多流水线方案（见图7-13），这样，理论上，每个时钟周期可以完成几条指令。奔腾4处理器有6条并行的流水线译码器。在计算领域，这项技术称为“超标量”（superscalar）处理，以区别于多条指令的“向量处理器”，以及基本的、单指令的“标量”单元。最初的奔腾处理器提供两个5阶段指令流水线，分别为U和V，每个时钟周期能够完成两个指令的译码过程，但是，只有非浮点数指令适用于这种并行处

理。浮点计算指令使用专为浮点运算准备的第三条流水线！当前，流水线V在硬件支持上不如流水线U；同时，8级浮点流水线并不能完全与整数运算并行运行。因而，硬件依旧有提高的空间！

在适当的条件下，两个整数指令能够在同一时钟周期完成。浮点单元的性能已经远高于早期的Intel 8087，但很遗憾，它共享了U流水线的一些逻辑电路，因而也就从某种程度上降低了并行运算的可能性。

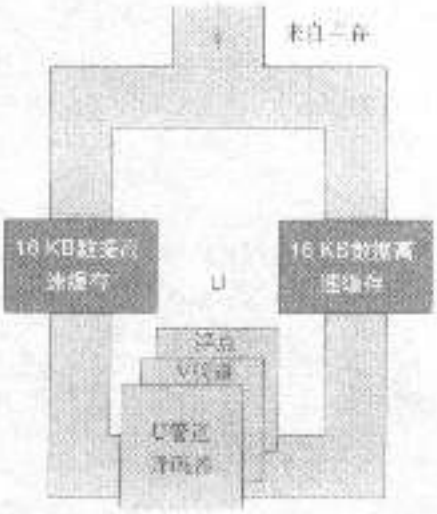


图7-13 奔腾处理器中L1高速缓存为其二条译码流水线提供数据和指令

7.8 奔腾4：扩展

Intel在奔腾4处理器中引入了多项重大的设计创新，我们称之为Netburst架构。

如前一节所述，分支预测恢复流水线被扩展到20级。这样的深度是奔腾III处理器的2倍。通过简化每一级发生的操作，它允许工作时钟速率大幅增长。但是，采用这么深的流水线，当程序在执行时跳转到未预料到的代码块时，所带来的负面影响也会非常大，流水线越长，再次装满它所需时间也越长！为了降低这种流水线再装的发生频率，设计人员为分支预测单元设计了更好的算法，并且装备更大（4 K）的高速缓存以提供分支预测的成功率。据称，这样能够将预测条件分支结果的性能提高33%。

奔腾4处理器还装备有双倍时钟速率的整数ALU，性能是之前电路的2倍。在1.5 GHz奔腾4处理器上，它们的实际效能为3 GHz，双倍于CPU的时钟速度。

或许最吸引人的变革是L1高速缓存。它已经被移到初级译码器后面的位置，并被改名为执行跟踪高速缓存（Execution Trace Cache，ETC）。传统上，奔腾和Athlon将x86指令（CISC指令）译码成更小的微操作，以得益于更快的RISC风格的流水线结构。奔腾III将x86指令保存在L1中，而奔腾4则将转换后的微操作保存在ETC中。这样，ETC取代了传统的L1指令高速缓存，它缓存微操作，而非x86指令。奔腾4处理器中跟踪高速缓存的大小尚且不得而知，但它能够存储约12000条微操作，微操作可能为64位宽。同时，奔腾4数据高速缓存的大小已经被精简到只有8 KB。但是，L2高速缓存已经扩充到256KB，它与CPU的接口也更快，宽度扩展为256位，并且运行速度达到CPU时钟速度。另外，100 MHz、四倍数据访问前端总线，提供高达3.2GB/s的超高数据速率。

Intel还在奔腾4中添加了144条新指令。它们是单指令多数据流扩展（Streaming SIMD Extension，SSE2）。即使用单条指令处理多个数据目标。这也是单指令多数据（Single Instruction Multiple Data，SIMD）名称的由来。SSE2指令处理128位、双精度浮点数的运算，能够加速多媒体、工程或科学等应用。但是，对于某些运算，奔腾4实际的FPU的运算能力比不上奔腾III中安装的FPU。

在奔腾4中，Intel还引入了在两个不同的代码线程间共享执行流水线的能力，即所谓的超线程技术。其目标是让流水线尽可能地忙，尤其是在访问高速缓存失败，可能会延迟其他活动的情况下。当

这样的事件发生时，超线程机制允许CPU将流水线中当前指令交换出去，替换成另外一组指令立即执行。之所以能够这样，是因为大部分电路均可以重用，只有主要的寄存器和控制状态需要保存，以备将来之用。这样，流水线的易变环境可以设置为两个版本中的一个，其效果立即显现。Intel声称这样的增强只用了不到5%的芯片面积，但却提供快速上下文切换的能力，能够带来约15%的性能增长。

7.9　Microsoft Developer Studio：调试器的使用

调试器工具一般会和编译器和连接程序一同提供，有些程序员很少使用调试器，他们更喜欢在其程序中插入临时性的打印语句来发现问题，指示变量什么时候被破坏。同样的工作如果使用调试器来完成会更优雅。造成这种情况的原因，或许是由于人们认为学习调试器命令很困难。如果已经苦于熟悉新的工作平台、编辑器命令、语法和应用程序的逻辑，另外的学习任务当然不会受到欢迎。但调试器对计算机体系结构的学生很有用。使用它可以方便地查看CPU内软件的运作。

Developer Studio调试器允许程序员监视程序的执行（见图7-14）。系统会同时运行调试器和目标程序，在两者之间进行切换，以显示CPU的寄存器和所需的内存存储单元。对于我们来说，最有用的功能就是单步调试（single stepping）。单步调试允许我们每次执行一条指令，从而有大量的机会检查分析每条指令的结果。按下F10键，调试器会执行一条指令。我们可以看到指令所导致的CPU标志位、寄存器和内存存储单元的变化。



图7-14　Microsoft Developer Studio调试器

在使用调试器之前，在大多数系统上，一般要执行特殊的编译和链接操作，以提供调试器所需的额外信息。在Microsoft Developer Studio中，这个调试选项通过【Build】→【Set Active

Configurations】→【Debug】设置。切记，每次编辑源文件后重新编译可执行代码。

在开始时，最好调试直接用汇编语言编写的例程。对初学者甚至专家来说，HLL编译器的输出都太难理解。为了省去购买和安装独立汇编器的需要，我使用VC++编译器提供的内嵌汇编语句__asm功能。其他编译器也提供类似的功能。图7-14中使用的汇编代码的例子，在图7-15中更清楚地列出。

```
/* demonstration of the use of asm instructions within C prog*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
char format[] = "Hello World\n"  //declare variables in C

__asm {                  ;switch to inline assembler
      mov ecx,10         ;initialize loop counter
Lj:   push ecx           ; loop count index saved on stack
      lea eax,format
      push eax           ;address of string, stack parameter
      call printf        ;use library code subroutine
      add esp,4          ;clean 4 byte parameter off stack
      pop ecx            ;restore loop counter ready for test
      loop Lj            ;dec ECX, jmp back IF NZ
}                        ;back to C
return 0;
}
```

图7-15 __asm指令的应用实例

HLL控制结构（SEQ、IT和SEL）由机器指令集里的专门指令直接支持。例如，FOR循环由LOOP指令组实现。这些递减分支指令在决定继续循环还是退出时，可以测试2个条件。在将ECX减1后，LOOPNZ检查计数值是否到达0，或者CPU的Z标志位（ZF）是否设置。任何一个条件都会终止循环，否则JMP回到循环的顶部。有意思的是，ECX的递减本身不会影响Z标志，这样，循环内的其他指令能够使用该标志位作为退出标志位。

调试器可能会十分杂乱，因而应该将显示窗口的数目降到最低。我建议只使用4个窗口：编辑器、CPU寄存器、数据输出以及（有时）内存窗口。可以使用调试工具栏打开或关闭这些显示窗口。

• 在Microsoft Developer Studio中，有一些有用的快捷键可以减轻鼠标在垫子上的移动。表7-5将它们列了出来，在长期的工作中，使用它们可以节省时间。

调试器工具栏有一系列的按钮，可以用来打开各种调试器视图窗口，图7-16对它们进行了汇总。如果调试器工具栏没有显示，可以点击Developer Studio窗口右上侧最右端的灰色背景，得到一个下拉式菜单。调试器提供5个窗口（列在表7-6中），参见图7-14的快照。

表7-5 Developer Studio调试器中一些有用的键盘快捷键

F1	帮助
F4	跳到下一个错误
^F5	运行程序
F7	生成可执行代码
^F7	仅编译
F9	设置断点
F10	单步执行（跳过函数）
F11	单步执行，进入函数
ALT+TAB	切换窗口

表7-6 Microsoft Developer Studio调试器显示的窗口

1. CPU寄存器
2. 程序内存及标签和反汇编后的助记符
3. 数据内存及ASCII解码表
4. 正在调试的程序的输出窗口
5. 堆栈，仅返回地址

在寄存器和反汇编窗口打开时，可以使用鼠标右键调出下拉式菜单，对显示的内容进行配置。一般不需要显示浮点寄存器，因此，我们将它关闭，以释放出屏幕空间。

在熟悉了之后，可以试着使用反汇编程序，显示由编译器为C/C++程序生成的汇编语言助记符！

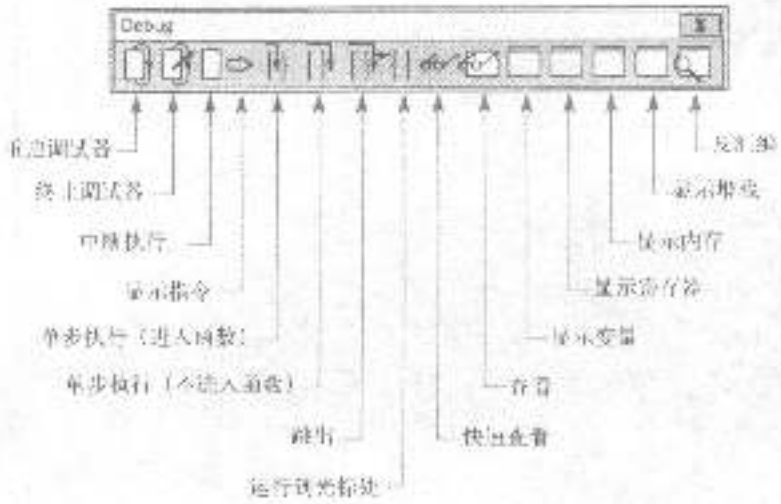


图7-16　Microsoft Developer Studio调试器工具栏

最后，表7-7中列出的按键对所有的Windows应用程序都是通用的，在开发程序时，它们会比较有用。

表7-7　Windows的键盘快捷方式

^ESC	打开任务栏上"开始"菜单，打开后用箭头进行
Tab	在桌面上、资源管理器、任务栏和"开始"菜单间切换。在打开"开始"菜单的情况下，它在应用程序中切换
Alt+F4	关闭当前的应用程序。在桌面上使用可以关闭Windows
Alt+Tab	切换到下一个窗口
Shift+Alt+Tab	切换到前一个窗口
ESC	在对话框中取消当前的动作
F1	显示应用程序的在线帮助
Shift+F1	上下文敏感型的帮助信息
F2	如果一个图标处于选择状态，使用这个键可以改变它的名字
F3	打开查找窗口

7.10　小结

- 最著名的CPU是奔腾微处理器，几乎所有现代PC中都使用这种处理器，Intel、AMD和Cyrix制造了许多不同的版本。
- 为了提高性能，处理器芯片上提供高速缓存（L1）。现在，赛扬370A还在芯片上集成L2高速缓存。
- 尽管奔腾是CISC CPU，但是它采用了RISC设计中的流水线译码，这使得CPU可以同时对几条指令译码，如同生产线那样。
- 奔腾CPU装备有两条译码流水线，使得它能够同时处理两条指令流——只要它们不互相影响。
- CPU寄存器保留了一些固定的专用功能，比如ECX处理循环计数。这一点正是CISC的特性，甚至与最初的Intel 8位微处理器有关。

- 为了理解译码过程，我们可以将基本指令代码分成字段，这也是CU必须要做的事。
- 在程序序列中，CPU状态标志位起到指令间消息传递者的作用。
- 不管数据是在内存中还是在CPU寄存器中，可以使用许多不同的方式来完成数据访问。
- 流水线译码器通过重叠操作加速CPU的运行。

实习作业

我们推荐的实习作业包括熟悉Microsoft Developer Studio集成开发环境（Integrated Development Environment，IDE），以及在C语言编写的HLL程序中，用__asm指令开发和运行小型的汇编语言例程。在本书后面的附录中，附有安装指南以及一些建议，帮助你开始编写自己的第一个程序。

练习

1. 奔腾CPU有多少个数据寄存器？这些寄存器为程序员提供多少字节的存储空间呢？
2. 列出机器指令的5种类型，将下面的奔腾指令分配到正确的种类：

```
MOV AX,count
OUT port1,AL
ADD EAX,20
CMP AL, 'Q'
JMP DONE
CALL doit
INT 21H
HLT
```

3. 说明下面这条指令中用到的寻址方式：ADD EAX,[ESI]。在什么情况下编译器会使用这条指令？
4. 给出三种不同的将AX清零的方法。
5. 寄存器间接寻址的主要优点是什么？
6. 你认为哪种寻址方式是最基本的方式？为什么？
7. 在执行单条奔腾指令时，最多会用到多少寻址方式？解释术语“有效地址”。
8. 下面这6条指令中，使用了哪些寻址方式：

```
mov DX,[countx]
lea ESI,array1
add EAX,[SI]
cmp EAX,target
call doneit
mov result,EAX
```

9. 说明奔腾LOOPZ指令的使用。如果希望循环10次，那么计数器的初始值和结束值分别应该是什么呢？

A. 说明奔腾CMP指令的动作。

B. JZ和JNZ指令之间的不同是什么？

C. 在程序执行时，CPU状态标志位的作用是什么？为什么它们可以看做是ALU和CU之间通信的一种手段？

D. 与可怕的HLL goto指令等价的奔腾指令是什么？为什么你更倾向于使用这些指令而不使用goto呢？

E. 考虑图7-15中的汇编代码。printf()的参数列表是什么？为什么ESP加4就能让它消失？将add esp,4替换成另外一行代码。

F. 下面这6条奔腾指令分别完成什么工作？

```
LOOP strout, MOV AL, 123H
```

```
MOV DX, [count], PUSH AX
MOV CX, 0AH, LEA EBX, NAME1
```

课外读物

- Messmer（2001）。
- Anderson和Shanley（1995）。
- Intel（1997）。
- 下面的网站有详细的信息，但需要花一些时间：

http://developer.intel.com/design/pentium/manuals/

- Intel Technology Journal的在线版，以前发行的内容可以参见：

http://developer.intel.com/technology/itj/index.htm

- 现代CPU的总结性论述：

http://www.pugetsystems.com/cat_info.php?cat=m&show=CPU

- 汇编语言的介绍及说明：

http://maven.smith.edu/~thiebaut/ArtOfAssembly/artofasm.html

- 编程爱好者不能错过下面这个网站：

http://www.programmersheaven.com/

- Microsoft Developer Studio的在线帮助，提供许多不错的解释和技术说明。
- Heuring和Jordan（2004），指令集。
- Tanenbaum（2000），5.5.8节：Pentium II instructions。
- Intel和AMD处理器的不同插座：

http://users.erols.com/chare/sockets.htm

- 这些网址可以通过本书的配套网站访问：

http://www.pearsoned.co.uk/williams

第8章 子例程

子例程，过程，函数或方法均指相同的事物。它们是一个个的代码块，可供许多不同程序在不同的地方多次重复访问。对子例程化（subroutining）的有效支持，成为CPU功能的重要方面，向子例程传递参数、管理局部变量以及管理子例程的返回等等，都由CPU内部的专门硬件设施来协助完成。

8.1 子例程的目的：节省空间和精力

现在，程序员一般使用高级语言（High-Level Language，HLL）来编写他们的程序。使用HLL语言编写程序能够提高效率，减少错误，同时隐藏底层大部分的复杂细节。子例程（subroutine）——在Pascal中称为过程（procedure），在C中称为函数，在Java中称为方法，是程序执行时被重复使用的代码段。起先，人们最初只是将使用了例程看做是共享代码，以降低程序大小的一种方法。人们为此专门提供了特殊的机器指令，如BSR或CALL，这些指令能够改写IP寄存器的地址值，使读取执行周期切换到子例程的代码执行。这种改变IP寄存器的方法相当于“jmp”或“goto”操作，不同点是，CALL指令还会执行一些额外的动作（我们将在随后的小节中论述），从而使控制权能够返回到调用发生的地方。

单个程序中使用了例程时，需要解决子例程结束后“返回到哪里？”或“我来自哪里？”的问题。图8-1用具体的例子说明了这一问题。主程序中，在两个不同的地方、不同的时间两次调用了average()子例程，每次子例程执行完毕后，都需要解决“返回到哪里”的问题。

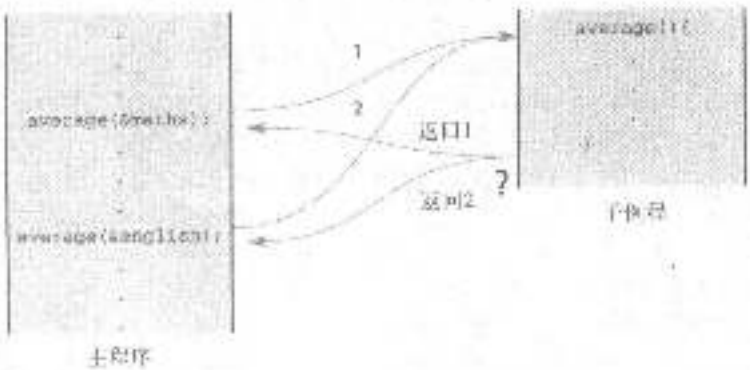


图8-1 调用来自何处

当子例程代码由几个处于多任务操作系统（如Unix或Windows）控制之下、有可能并发运行的程序共享时，更多的问题会浮现出来。这种情况下，子例程在公共的动态链接库（Dynamic Link Library，DLL）中，如图8-2所示。它们就客户程序一同载入内存。“返回到哪里”这个问题变得更为复杂，因为多个程序可能会同时执行同一子例程的代码！另外，还必须做到不能让用户程序有机会修改共享的代码，或干扰由其他用户设置的变量。

使用子例程的确会降低程序的执行速度，但对于HLL程序员来说，这已经不是什么大问题，在本章后面，我们会介绍造成这种延迟的原因。现在，程序员的确期望编译器能够在优化阶段将他们代码的运行时间最小化，但如果优化后依旧不能接受，为什么不购买一个更快的CPU呢？

有关子例程的更重要的一个方面，就是它们封装（或包装，encapsulate）特定活动的方式。在设计和编写了例程时，一种好的风格是让子例程执行相对独立的特定动作或处理工作。这会使得了

例程（在经过充分的测试检验之后）适合于放入库中供其他用户使用。子例程库的来源包括HLL编译器、图形环境、操作系统接口、Internet支持包等等。如我们在2.5节所述，链接器将编译后的文件连接到一起，制成可执行程序。为了加快链接过程，子例程常常编译成目标格式后放入到库中。链接器在试图找到正确的子例程来使用时，一般要扫描好几个这样的库文件。因而，尽管程序员最初使用子例程的目的是降低代码的大小，但现在它们更重要的作用是，让程序员能够更方便地访问到那些经过测试和实践检验的代码。这样能够降低成本，帮助程序员更快地开发出更可靠的软件系统。

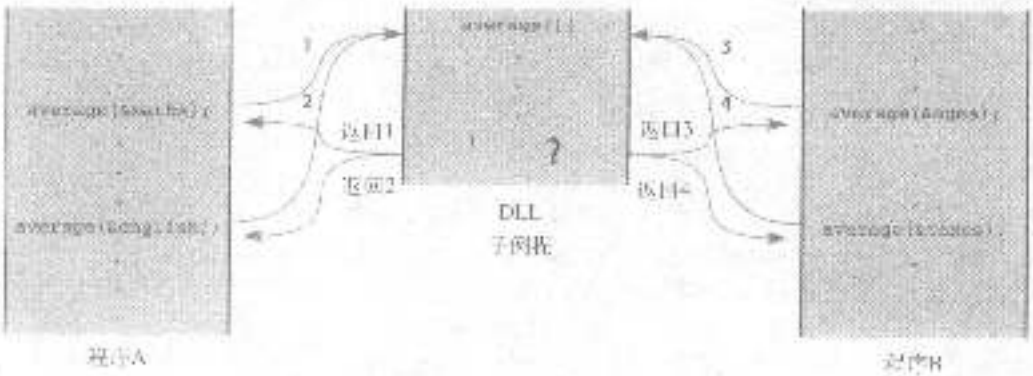


图8-2　调用来自何处

8.2　返回地址：堆栈的引入

所有数字计算机都会面临的一个基本问题，就是共享代码或子例程的使用，在将控制权从程序的一处转到另一处时（在CALL子例程期间），如果想返回，如何记住程序是从哪里转来的呢？有人可能会建议将返回地址（return address）存储在特殊的专用内存存储单元中，甚至是空闲的CPU寄存器中。刚开始，这或许能够工作。但如果子例程本身又需要CALL另外的子例程，这个问题又会出现，因为特殊的内存存储单元或寄存器已经被占用，因而需要更多记录返回地址的寄存器。但是，问题在于在程序实际运行之前，并不知道具体需要多少才能够满足要求。

这个复杂的问题现在已经有了完美的解决方案：在跳转到子例程之前，将返回地址存储到内存中的堆栈（stack）上。在子例程结束时再恢复这个地址。图8-3给出了这种方案的工作流程。CALL指令现在有两件工作要做，首先将地址从EIP复制到堆栈上（PUSH），接下来将所请求的子例程的地址存入EIP（JMP），在结束子例程时，RET指令只是简单地将最后32位数据项退栈，并将它作为返回地址，将其填回EIP。如果从堆栈中取出然后存入EIP的32位数据项不表示合法地址，则会发生严重的崩溃事故。为了便于理解，可以将CALL指令想像成PUSH EIP和JMP SUBR的组合。

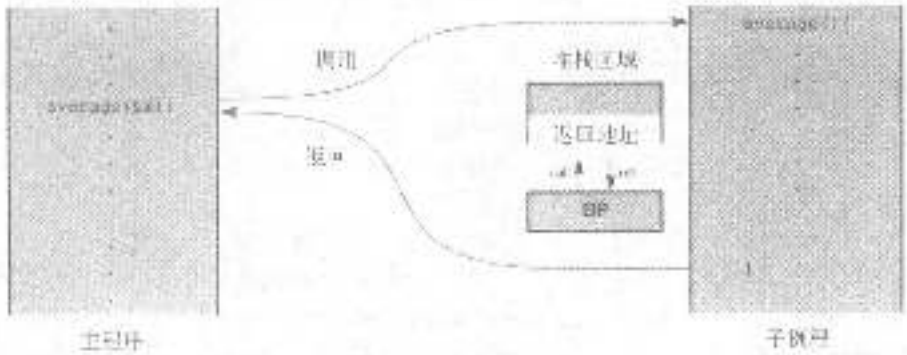


图8-3　使用堆栈保存子例程的返回地址

由于这类应用全关重要并且频繁使用，因而，CPU内专门提供一个特殊的寄存器——堆栈指针寄存器（Stack Pointer Register，ESP），专门指向主存中这一指定作为堆栈的区域。理解这点非常重要。即：堆栈是内存中的一段数据区域，它的位置由CPU的ESP寄存器中的地址来确定。

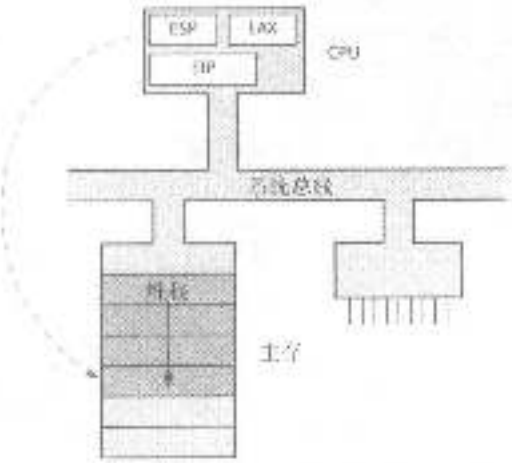


图8-4 堆栈指针寄存器（ESP）指向栈顶

如图8-4所示，系统堆栈常常从内存的顶端开始，向下增长，但这只是一个惯例。重要的是，不要混淆“队列”和“堆栈”这两种数据结构。队列是以一种民主的，先进先出（FIFO）的方式进行操作。而堆栈提供的是另一种十分不同的后进先出（LIFO）的方式。要注意，堆栈上的数据仅当其处于堆栈指针以上时才合法有效。如果处于堆栈指针以下，虽然使用调试器它依旧是可读的，但从程序的角度看，它已经无效，且是不可读的。

8.3 使用子例程：HLL程序设计

所有处理器（比如英特尔或ARM）的指令集中，都有一组专门的指令支持子例程的使用。编译器在转换HLL程序时，如果遇到过程或函数，则会适用这些指令。图8-5中给出的C语言代码片段中，用到了库函数printf()和用户自定义函数average()。

```
#include <stdio.h>
#define MAXNUM 10

        int marks_scores[MAXNUM];
        int tech_scores[MAXNUM];

float average(int x, int*y){
  int i;
  float av;
  for (i=0; i<x; i++){
     av += *y++;
  }
  return av/x;
}

void main (void) {
  int i;
  .....

printf("Average of marks=%7.1f\n", average(marks,MAXNUM)); /*FOR MORE*/
  .....
printf("Average of technology=%3.1f\n", average(tech_scores,MAXNUM));
  .....
}
```

图8-5 函数（或子例程）在C程序average.c中的使用

我们注意到，在程序中的不同地方，函数average()被用到不止一次。编译器会将main()和average()函数转换成机器代码。同样，多次使用的函数printf()是由C函数库提供的，因而不需要再次编译，只需在用户的程序载入之前链接进来就可以了。子例程的另一个重要特性是局部变量的声明，这些局部变量在进入函数时自动出现，函数结束后自动消失。函数average()中的变量i和av就是局部变量的具体例子。汇编后的程序展示出，在底层子例程是如何使用的。还有对操作系统子例程的直接访问，用来输出内容到屏幕。在8.5节中，我们将会给出编译器为average.c程序生成的汇编代码。

8.4　堆栈：大多数操作的基本要素

堆栈也用来存储临时变量。如果两个变量需要使用同一CPU寄存器，比如ECX，它们可以采用堆栈作为临时性的存储区域，依次使用该寄存器。机器指令PUSH和POP将数据项复制到堆栈中，之后在需要时从中移走。参见图8-6。这些显式的堆栈操作使用堆栈指针寄存器ESP来保存当前位于堆栈栈顶的数据项的地址。



图8-6　使用PUSH和POP堆栈指令

PUSH指令首先将ESP中的地址减掉数据项所占字节数的多少，然后将数据项写入到ESP所指定的内存单元中。这是“寄存器间接预减量”寻址方式的例子。POP指令的工作方式正好相反，将ESP作为指向内存的指针读出数据项，然后按照从堆栈上移出的数据项的大小，递增ESP。这种方式即为“寄存器间接后增量”寻址方式。有时，程序员可能会需要显式地将堆栈指针向上或向下调整。

```
ADD ESP, 4                ; 将4字节移出堆栈
SUB ESP, 256              ; 在堆栈上保留256字节的空间
```

和许多其他计算机一样，奔腾不能PUSH或POP单个字节。对于单字节的情况，它会将其作为完整的字来处理。

有些计算机为了提高速度，在CPU中内建堆栈区域。还有一些计算机仅有少数几个CPU寄存器，它们依靠堆栈来保持所有的算术操作数。计算机能够只使用单个堆栈工作，但是如果这样，就不能将用户程序与享有更多特权的系统代码分开。为了达到这种级别的安全性，计算机必须在用户堆栈和系统堆栈之间切换。多任务系统可以将这个特性扩展成每个任务都有自己的私有堆栈。任何时候只有一个处于活动状态，ESP指向其顶部的数据项。实际上，ARM处理器提供6个版本的堆栈指针（一般为R13寄存器），以协助不同运行上下文间的切换。奔腾处理器提供进一步的安全特性，它将内存分成不同的段（segment）来保护内存中的数据项。通过这种方式，内存被划分成代码段、数据段和堆栈段。为了保证只有合法的动作才能在堆栈上执行，可以在独立的堆栈段初始化堆栈。采用这种方案时，读取/执行访问只允许在代码段内进行。PUSH和POP只能在堆栈段工作。Modula-2、C、Pascal和ADA的编译器都使用内存中的堆栈来处理过程的CALL/RETURN序列。存储临时变量以及其他一些活动。表8-1列出了可以用堆栈完成的四项基本工作。

表8-1　系统堆栈的常见应用

1. 在过程调用期间保存返回地址
2. 向过程传递参数
3. 为局部变量分配存储空间（堆栈框架）
4. 保存寄存器值的临时高速缓存存储器

如果对堆栈的操作不平衡，则会发生一些有意思的和细微的错误。程序可能在测试时一切正常，甚至能够长时间运行，直到堆栈超出内存的顶端，或写代码或文件不合法的返回地址，引起系统崩溃。堆栈错误才会显现出来。在进行HLL编程时，编译器会确保所有的参数、局部变量和返回地址都正确地入堆栈和出堆栈。

8.5　参数传递：将子例程具体化

大多数子例程都需要一些参数来告诉它们每次调用具体应该做什么。尤其是print()。如果没有参数，则毫无用处！如果子例程每次调用都做完全相同的事，那么它们的用途也不会太大。参数可

以将子例程的行为具体化。对于我们来说，重要的是要了解调用程序如何传递参数值，以及子例程如何将结果送回来。向子例程传递数据最简单的方法是，在将要跳转到子例程之前，将数值复制到空闲的CPU寄存器中。进入了例程之后，值依旧还在寄存器中。同样的技术也可以用来返回一个数值。实际上，C函数通常的确是在数据寄存器中返回它们的返回值，奔腾处理器使用EAX。

只使用CPU的寄存器向子例程传递参数，其局限性很大。因而，堆栈也被引入进来，完成参数的传递。因此，编译器大量依赖于堆栈来完成子例程的CALL-PASS RETURN序列。在调用子例程时，首先参数被压入堆栈中，然后是返回地址，最后是为子例程中声明的任何局部变量（见下一节）分配空间。总之，堆栈就如同信箱。向子例程传递**值参数**（value parameter）类似于报章将报纸留着花园栅栏扔进来。它是受到欢迎甚至期待的，但其来源可能不好确定！如果您需要与报刊经销商取得联系，那最好还是有一个电话号码，或者一个具体的地址。这也就是**引用参数**（reference parameter，或称地址，address）的由来。此时，压入堆栈的不是参数的值，而是参数的地址。在C函数中，变量名添加&修饰符，即为引用参数。使用引用参数，子例程可以修改原来的变量。这种传递方式存在一定的危险性，使用时要加倍小心。

图8-7中Microsoft Developer Studio窗口是反汇编（disassembly）画面。其中既有多行C源代码，也有奔腾处理器的汇编助记符。调试器读入可执行的机器代码，将二进制代码逆转换成汇编语言的助记符。同时，它还在相应的位置叠加原来的C源代码。这样，我们就能清楚地看到编译器为每条HLL指令生成的代码。对于那些热衷于技术的人，能够看到这种类型的输出内容，肯定欣喜若狂。



图8-7 Microsoft Developer Studio调试器反汇编窗口（average.c）

左边空白处的小箭头表示当前读取-执行的位置，或者说是EIP指向的位置。这幅快照是在调用

average()函数的CALL指令执行完毕后立即堵获的。查看画面的底部，可以看到这个函数调用 "n = average(NCLASS, maths_scores);"。

注意其后的两条指令（PUSH maths_scores和PUSH 0AH），它们是为调用函数average()声明的两个变量（0A是10的十六进制表示）。在调用之前压入堆栈中，传递给子例程。CALL后的指令是ADD ESP, 8。这条指令将堆栈指针向上移8个字节，通过实际两个参数失效，形式上从堆栈中将它们移除。这种动作称为堆栈清理，这种动作是必需的，否则随程序的运行，堆栈会变得越来越大。这有可能是内存泄漏的一种原因，发生内存泄露（memory leakage）时，PC的可用RAM会越来越小。实际上，它是由于不断分配内存资源，但用完后未能正确释放它们的程序引起的。



图8-8　Microsoft Developer Studio调试器的堆栈和寄存器窗口（average.c）

这样的反汇编清单给出许多有意义的信息，因而，我们将在下一节处理局部变量和堆栈框架时，再次回到图8-7。

图8-8中的寄存器窗口和内存窗口的实际和图8-7取自同一时刻。我已将内存窗口显示的内容定位到内存中存储堆栈的区域，这样我们就能够看到函数的CALL/RETURN活动。同时，我还将它调整为只显示4个字节的宽度，从而能够更容易地区分32位的数据项。堆栈指针（ESP）标明当前堆栈顶的位置，注意，在这幅画面中，堆栈向上增长。堆栈中已有的数据项处于箭头之下高一些的地址。我们能够看到两个子例程参数以及返回地址。

回到图8-7，记下紧跟函数调用call @ILT+0(_average)后面的指令的地址。我们会发现，这一地址和存储在堆栈上的地址（00401073）是同一地址。在内存窗口中，字节都以与数值次序相反的方式显示，这会带有不便。作为了获得查看存储处理器堆栈如何工作的特权，这种小的不方便也是值得的。

参数入堆栈的次序是Pascal和C编译器的不同之一。C入堆栈从列表右端的参数开始，而Pascal选择首先将最左边的参数入堆栈。如果使用库例程时没有注意入堆栈顺序的不同，可能会造成混乱。

8.6　堆栈框架：所有局部变量

过程使用的堆栈区域称为**堆栈框架**（Stack Frame）。其中保有返回地址、过程的参数和局部变量。由于过程调用有可能嵌套，因此需要一个标志，标明当前堆栈框架的顶部以及前一个框架的底部。为此，CPU中的一个寄存器专门用做**堆栈框架基址指针**（Stack Frame Base Pointer）。在微处理器中，EBP常常保留做这种用途，在过程执行期间，ESP可能或增或减，但赋予EBP的值保持不变。但是，CPU中只有一个EBP，因而，当分配另外的堆栈框架时——可能由对其他过程的调用引发，EBP中的值要压入堆栈中，释放出EBP来指向新的堆栈框架。CPU在运行编译后的代码时，在调用过程期间，堆栈有可能会类似于图8-9所示。

有时，我们将保存在堆栈上的EBP的值称为**环境指针**，这是由于它指向前一个堆栈框架。现在，在诸如Pascal或ADA等语言中，前面的堆栈框架称为"in-scope"，其中的局部变量依旧能够通过环境指针来访问。值得一提的是，C语言中不允许这种做法，因为C语言不允许在函数内声明函数。Pascal编译器能够通过检查源代码内过程的嵌套声明，预测运行期间堆栈上框架的排列。

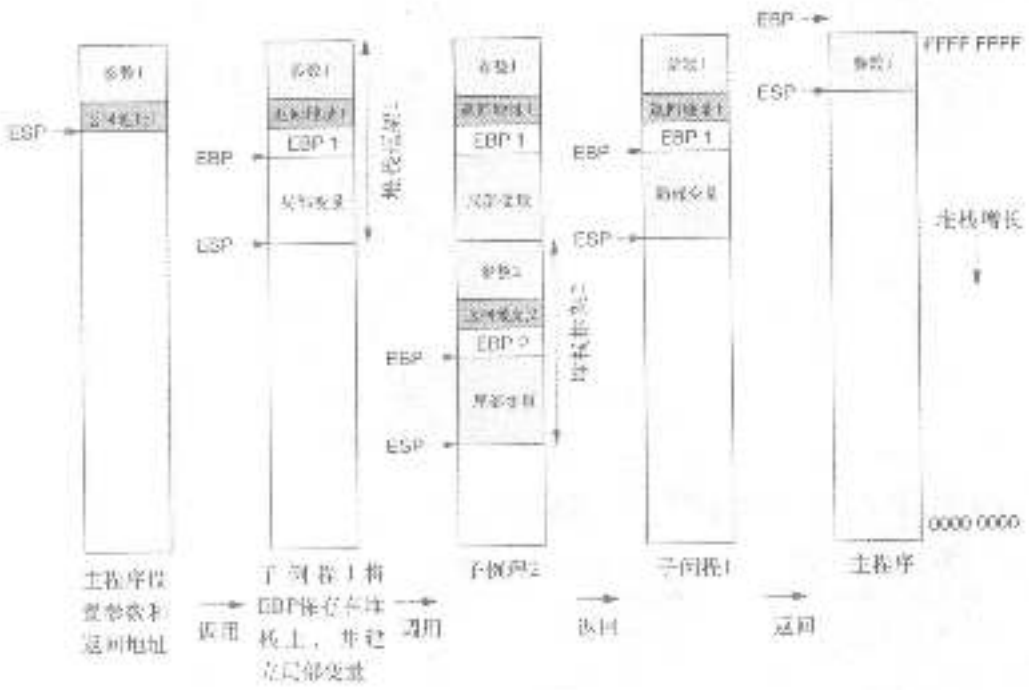


图8-9 使用堆栈存储局部变量

局部变量和参数都存储在堆栈上，而且都通过框架指针（EBP）以相同的方式访问，它们是紧密相关的。实际上，我们可以将过程参数简单地看做是预先初始化有效值的局部变量。

参见图8-7，我们能够看到参数和局部变量都建立在堆栈框架上。用来调用average()的参数在8.5节已经讨论过。现在我们检查图顶部average()内的前三条指令。在这里，可以看到框架基址指针的值被保存到堆栈上（PUSH EBP），新的框架指针被初始化（MOV EBP,ESP），同时为两个局部变量建立了存储空间（SUB ESP,8）。接下来查看图8-8，我们可以在堆栈上看到下面这些数据项：

```
02 00 00 00      av的存储空间      当前的堆栈头
80 11 41 00      i的存储空间                  ↑ 堆栈增长
80 FF 12 00      EBP之前的值
```

子例程到达结尾返回后，堆栈框架就无效了，也就不能再引用这些局部变量。但是，使用调试器依旧可以检查用做堆栈的这段内存区域，在新数据项压入堆栈中将它们改写之前，都能看到旧数据。如果堆栈的内存被无意中改写，或堆栈增长得过大，所产生的运行时错误可能很难预料，也难以追踪。

汇编语言级别的程序员保有对所有堆栈处理的控制。如果忽视了一些错误，则可能在运行时导致一些严重的混乱。汇编器并不提供这个级别的错误检查，由HLL编译器生成的代码得益于某些语法检查。但是，即使如此，指针操作在使用不当时，也常常会将程序弄得混乱。我们应该以图8-10中列出的例子为戒。它能够通过编译器的检查，但有时会产生严重的运行时故障。

图8-10中的函数getname()使用gets()输入一个字符串，将它存储在局部变量nstring[]中，并将字符串的起始地址作为指针返回给调用者。但是，子例程执行完毕返回后，其中的局部变量也就不再有效。遗憾的是，当调用该函数的程序试图使用指针myname访问数据时，错误处理程序不能有效地阻止这种行为。尽管现在数据字符串在堆栈指针之下，表明这些数据已经失效了，但有可能它们依旧可读。尽管存在这种严重的缺陷，但程序极有可能有时能够正常工作。因为仅当有新的数据项压入堆栈中时，含有字符串数据的内存区域才会被改写。试着运行这个程序并检查其结果，然后在getnam()和printf()间插入另一个函数调用，将含有失效局部变量nstring[]的堆栈区域覆盖。

```
#include <stdio.h>

char* getname(void) {
  char nstring[25];
      printf("Please type your name: ");
      gets( nstring);
      putchar( '\n');
  return nstring; //SERIOUS ERROR IN THIS PROGRAM
}

int main(void) {
  char* myname;
  myname = getname();
  printf("%s\n", myname);
return 0;
}
```

图8-10　使用堆栈上的C指针时常犯的一种错误

8.7　对HLL的支持：CPU针对子例程处理的特性

所有CPU中，过程调用和返回都是由专门的CALL/RET指令对来执行。从根本上说，它们和跳转指令（插入新地址值到程序计数器EIP内）相同。如8.2节所述，CALL指令还将当前EIP的值压入堆栈中，作为返回地址，然后将子例程的起始地址载入进来，这样就能完成向子例程代码的跳转。为了加速从子例程中返回的速度，奔腾处理器提供一个增强型的“RET n”指令。这条指令在从堆栈中删除所有的参数后，将控制权返回给主调程序。这样的动作符合Pascal编译器的约定，但C语言不是在子例程中，而是紧跟在CALL指令之后插入堆栈清理代码，将参数删除。这样，参数在同一块代码中插入和删除。十分奇怪的是，Windows坚持采用Pascal约定，包括采用与C语言相反的次序将参数压入堆栈中。Pascal编译器从左向右将参数列表入堆栈，而C则从右到左！

奔腾芯片提供另一项有用的指令对，即ENTER和LEAVE，它们用来打开和关闭局部变量的堆栈框架。类Pascal HLL语言比较特殊，它需要访问保存在前一框架中的作用域内变量，即堆栈的更高端，这种情况下，ENTER指令能够将环境指针从堆栈上的前一框架中复制下来。这些指针提供对作用域内变量的快速访问。一组环境指针称为显示框架（Display Frame）。如果没有显示框架，访问作用域内堆栈变量的惟一方式是使用环境指针一个框架一个框架地在堆栈中向上查找，直到到达正确的框架为止—这不是一种方便的操作。

8.8　中断服务例程：由硬件调用的子例程

尽管**中断服务例程**（Interrupt Service Routine，ISR）类似于普通的子例程——它们都含有代表系统执行某项操作的代码，但它们被程序员赋予完全不同的状态。人们常常会对中断服务例程十分敬畏，一般的编程课程常常完全忽略这部分内容。或许这要归因于它们执行的是十分关键的功能，传统上常常用汇编语言编写！中断服务例程和常规程序之间主要的区别在于，ISR从来不会用常规的CALL指令调用，它们仅在硬件中断信号（见图8-11）、特殊的陷阱指令（有时称为**异常**）发生时才执行。中断可能来源于磁盘驱动器，希望告诉CPU它已经完成数据传输，或者鼠标提示自身的移动，或者内存发出信号，表示发生致命错误。相应地，陷阱指令可以用来请求由操作系统控制的资源或动作。中断是完全无法预测的。这会存在什么问题呢？在中断发生时，CPU快速地切换到ISR的代码去执行，但是，执行这些代码需要使用几个CPU寄存器，而CPU中并没有专供ISR使用的寄存器，也就是说，这些寄存器中可能已经存有先前正在运行的程序中的重要数据——这就是问题之所在：在中断发生时，我们如何保护现有的程序和数据不被破坏呢？

解决方案是：将ISR可能用到的任何寄存器的内容都PUSH到堆栈上——执行ISR——然后在返回主程序执行之前，将所有的寄存器都POP回来。但是，这个优秀的解决方案对于不能直接使用PUSH和POP指令的HLL程序员没有任何用处。标准的HLL不提供对应PUSH和POP的指令。因此另一项差异明显的特性也就浮现出来：ISR需要用汇编语言编写。

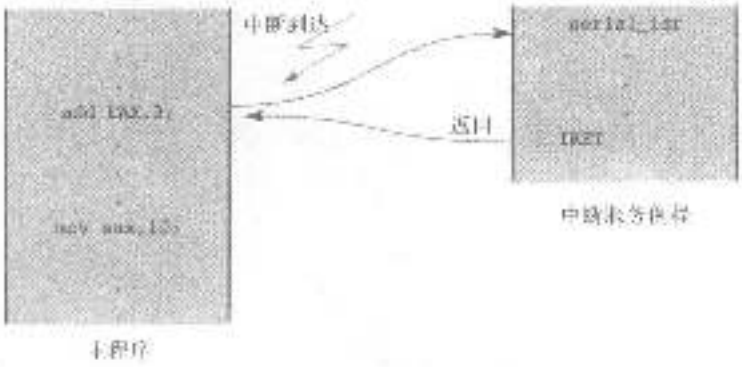


图8-11 中断——由硬件调用的例程

ISR使用相同的返回机制（使用堆栈保存返回地址），如8.3节所述。但开发ISR代码和开发子例程存在诸多重要的差异。

由于硬件中断可能在任何时候到达，没有任何先兆，程序员必须考虑到各种可能性。中断服务例程和常规过程之间另一项重大差异是，它们完全没有任何参数。由于中断可能在任何时候发生，因此，不可能准备一个参数列表。返回结果时也存在类似的问题，这也就意味着，中断例程的惟一工作就是更新内存中的一些全局变量，这还会产生其他一些问题，我们将在9.5节论述。

8.9 访问操作系统例程：后期绑定

在计算领域，关于地址绑定的正确时机有许多设想：我们是否应该延迟绑定呢？此处，绑定指的是确定某件事物的物理地址，从而使我们能够访问它。在考虑如何获得操作系统的功能，或者访问已经载入内存中的共享库模块时，这种讨论变得具有重大的实际意义。如果编译器不知道入口点的精确位置，那么它就不能将正确的地址插入到CALL指令中。考虑下面的代码片段，其中使用常规的CALL指令。虽然编译器或链接器能够计算出filter1的正确地址，并将其插入到可执行程序中：

```
call filter1
```

此处没有任何疑问：编译器确定所引用的子例程，链接器解决其修饰地址。这就是早期绑定。但是，下面的代码片段：

```
lea esi,filter1
...
call [esi]
```

实际被调用的子例程只能在运行时由载入到ESI寄存器中的地址决定。通过将不同的地址载入到ESI中，依前面代码的不同，可能有几个可选的子例程被调用。我们将其称为后期绑定（late binding）。寄存器间接寻址使这种动态引用成为可能。Windows在将应用程序中的CALL链接到由DLL（Dynamic Link Library，动态链接库）提供的函数时，就是在做后期绑定。Windows操作系统需要将DLL函数的入口地址传递给调用程序，这样调用程序才能获得对函数的访问。

MS-DOS和其他一些操作系统采用了另外一种需要使用特定指令的方式：软中断INT。软件激活中断的概念或许会有些奇怪。中断意味着不可预测的外部事件，但此处我们有一条指令在顺序执行的程序中请求发生中断。实际上，它只用来替代CALL指令，临时性地将控制权传递给服务例程。使用INT的方式进行调用所具有的重要优势，是服务例程的位置在中断向量表（Interrupt Vector Table，IVT）中，它位于内存中的固定位置。从而，如果操作系统必须将入口点移动到不同的处理例程时，只需更新IVT中的地址项，应用程序可以保持不变。

另外，值得注意的是，中断会改变CPU的优先级别，因为所有的ISR一般都由操作系统所拥有。通过将所有对操作系统的调用都经由中断执行，操作系统可以控制对关键资源的访问。

8.10 小结

- 子例程、函数和方法是一段段的代码，可以在程序内多次调用。它们能够节省内存空间及程序员的工作。
- 由于它们能够在多个地方调用，因此，必须有某种机制来记录子例程结束后返回到哪里。
- 和大多数处理器一样，奔腾处理器使用堆栈来保存子例程的返回地址。这是一种方便的技术，因为它还可以处理嵌套的子例程调用。
- 数据的读写只能在堆栈的一端（头部或顶部）进行。这些操作称为PUSH和POP。堆栈一般在内存中向下增长。
- 参数通过堆栈传入子例程（在转到被调用代码执行前，将参数存入堆栈中）。
- 在子例程中声明的局部变量占据参数之后的堆栈空间，这段区域称为堆栈框架。
- 参数可以看做是预先初始化好的局部变量。
- 在将指针作为参数传递给子例程时，常常会发生几类常见但严重的错误。
- 编译器提供子例程库，它们由链接器绑定到用户的代码中。

实习作业

我们推荐的实习作业包括使用子例程CALL/RET指令，使用调试器的内存窗口查看内存中的堆栈。还可以尝试两种不同的将参数传递给子例程的方法：CPU寄存器和堆栈。分别使用HLL和asm级别的代码调用C语言的库函数。

练习

1. 在计算过程中，“我来自哪里”这个问题的含义是什么？给出一些可行的解决方案。
2. 解释术语“局部变量”和“函数参数”之间的不同。
3. 为什么堆栈对于HLL程序员那么重要？奔腾指令RET n的作用是什么？
4. 为什么堆栈指针在CPU中，而堆栈自身却在内存中？
5. C函数如何将它们的值返回给调用程序？函数如何返回数组变量？
6. 什么时候编译器会选择使用PUSH/POP指令？
7. 用汇编语言的词汇，解释值参数和引用参数之间的不同。
8. HLL全局变量保存在哪里？
9. 链接器实际上做的是什么事情？

A. 检查各种HLL变量声明的汇编语言等价形式：char、int及char的数组。

B. 在C函数调用过程中，堆栈的作用是什么？使用调试器单步调试图8-11的程序，监视堆栈，看返回的指针如何变为“无效”。

C. 没有系统堆栈的计算机如何管理过程的调用和返回活动？参数又如何处理呢？

D. 为什么要通过框架指针，而非简单地使用堆栈指针，来访问局部变量呢？

E. 一些子例程，如printf()，如何拥有可变数目的参数呢？

F. 在调用子例程时，相对于使用参数传递，依赖于全局性数据的缺点是什么？

课外读物

- Heuring和Jordan（2004）。
- Tanenbaum（2000），5.5.5节：procedure call instructions；5.6.3节：comparison of subroutines to coroutines。
- Patterson和Hennessy（2004），有关硬件对过程调用的支持。
- 这些网站可以通过本书的配套网站访问：

http://www.pearsoned.co.uk/williams

第9章　简单的输入输出

本章介绍计算机从外界接收或向外部传送数据的三项技术（轮询、中断和DMA），并给出其体的应用。同时，本章阐述了“阻塞式”和“专注式”轮询之间的重要区别。从程序员的角度，讲述了IO芯片中的三类寄存器（状态寄存器、命令寄存器、数据寄存器）。本章揭示出中断服务的某些方面，同时讨论了中断的各种应用，这涉及关键数据的问题，或者说由于并发访问造成的数据破坏。本章论述了如何恰当地使用DMA的功能，对中断在支持多任务的操作系统中的重要作用也做了介绍。

9.1　基本IO方法：轮询、中断和DMA

3.8节中介绍过的最简单的字节宽度的并行端口，现在依旧广泛地用于计算机的数据输入输出。Centronics打印机端口基本上就是这类字节宽度的并行端口的一种，从计算机发展的初期至今，所有的PC都提供并行口。串行口也同样有用，人们常常称之为COM端口或调制解调器端口。还要注意的是，从根本上讲，局域网接口是快速的串行端口，它使得计算机可以访问高带宽的网络。为了处理这么快的突发数据，需要硬件提供更大的支持。新的USB（Universal Serial Bus，通用串行总线）标准就是为了在无需投资完整联网设备的情况下，提供另一种将设备连接到计算机的方式。串行通信链接的更多细节在第10章中论述，第11章专门讲述并行总线和接口。

不管处理串行端口还是并行端口，用来通过计算机IO端口传送数据的软件技术主要有三种，如表9-1所列。

表9-1　不同的输入输出技术

1. 专注式和间歇式轮询
2. 中断驱动
3. 直接内存访问（DMA）

本章中，我们将会详细地讨论每种方法。每种方法都需要软件驱动程序（driver）例程与它们所对应的IO硬件部件紧密协作才能完成。这些例程一般是操作系统的一部分，很少有不用汇编语言编写的情况。在PC市场上，扩展卡供应商在销售硬件时，会在软盘或CD-ROM上提供这类驱动程序例程，这样用户在需要的时候可以安装它们。现在通过Internet获得驱动程序例程库的情况也越来越普遍。从Unix开始，现代的操作系统都尽可能地用HLL编写，比如C语言。通过这种方式，将操作系统移植到新的处理器会更快也更可靠——只要有个好的C编译器就行！Windows为软件定义了一套特殊的硬件接口，即HAL（Hardware Abstraction Layer，硬件抽象层）。它就如同虚拟机层，有助于将操作系统移植到新处理器的工作。传统上，人们将软件看成相互通信的多层结构，如图9-1所示，相邻的层互通消息。每个层都有特定的数据处理任务。



图9-1　软件对硬件的访问

HAL隐藏了类似Alpha和MIPS处理器许多具体的硬件差异，使得操作系统代码的主要部分不需要考虑这些硬件差异，从而让移植和维护系统代码更为容易。尽管对IO硬件的直接访问在Windows 98上依旧可行，但Unix和Windows XP出于安全上的考虑，严格地禁止这种操作。大部分应用程序程序员不需要考虑这项限制，他们只需调用HLL编译器提供的库过程，比如gets()和puts()，就能够完成IO功能。这些过程接下来也可能会调用操作系统代码——有时存储在PROM或闪存内，以获得对实际硬件的访问。

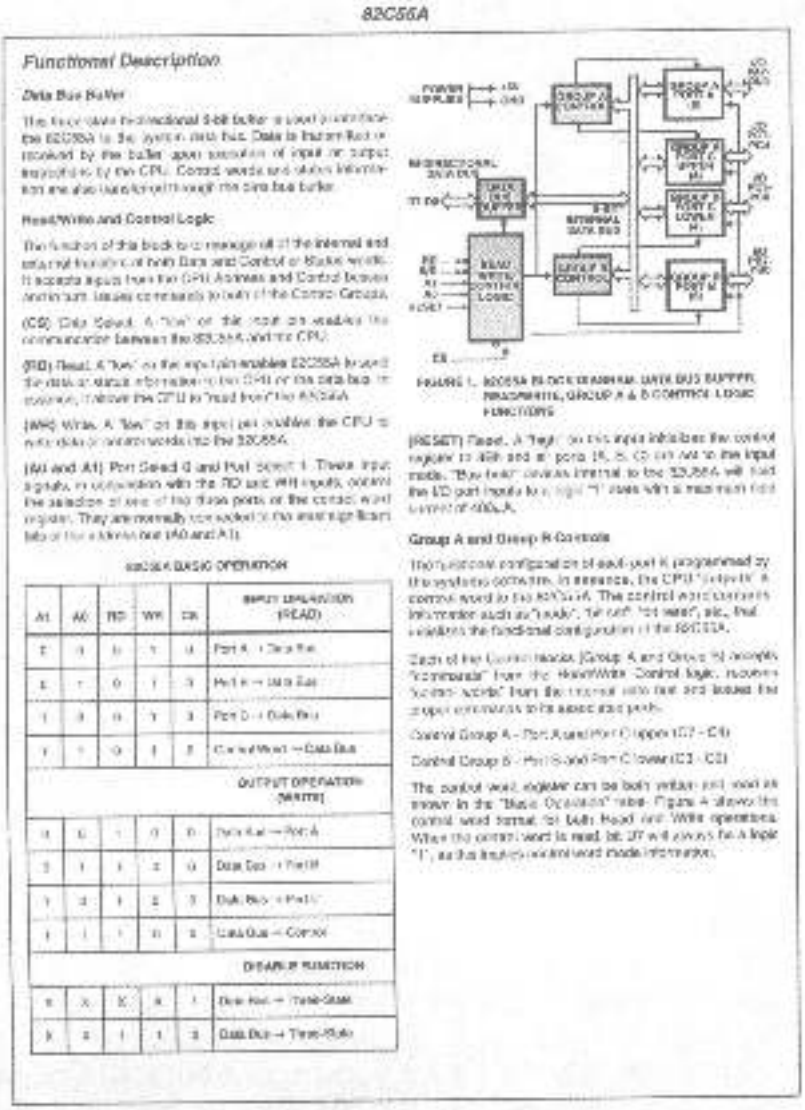
9.2　外设接口寄存器：程序员的角度

如果需要直接寻址IO端口——假定操作系统的安全控制允许这种行为，那么我们必须知道IO芯片在存储器映射中的基地址，以及其内部寄存器的布局和功能。

这种情况下，来自于芯片制造商的技术数据表以及一些有帮助的使用说明是非常珍贵的，即使不算必需的话。在面对这些资料时，作为程序员而非电子工程师，必须快速学会如何有选择地阅读，只将注意力放在那些与编程有关的方面。图9-2给出一个具体的数据表（完整版在http://www.intersil.com/data/FN/FN2969.pdf），程序员当然需要有关内部寄存器的端口地址以及它们的功能的信息。阅读外设接口芯片的硬件用户指南，尽管乍看起来很复杂，但实际上并不比阅读编程的书籍困难多少。IO芯片主要含有三类寄存器，在表9-2中列出，每种类型可能没有，也可能有多个，IO寄存器常常作为字节而非字来访问，有时，每个位对应一项独立的特性或功能。在阅读IO芯片的数据表时，最好从区分这三种寄存器的类型开始。

表9-2　外设芯片寄存器分类

1. 命令寄存器
2. 状态寄存器
3. 数据寄存器



82C55A

Functional Description

Data Bus Buffer

The three-state bi-directional 8-bit buffer is used to interface the 82C55A to the system data bus. Data is transmitted or received by the buffer upon execution of input or output instructions by the CPU. Control words and status information are also transferred through the data bus buffer.

Read/Write and Control Logic

The function of this block is to manage all of the internal and external transfers of both Data and Control or Status words. It accepts inputs from the CPU Address and Control buses and in turn issues commands to both of the Control Groups.

(CS) Chip Select. A "low" on this input pin enables the communication between the 82C55A and the CPU.

(RD) Read. A "low" on this input pin enables 82C55A to send the data or status information to the CPU on the data bus. In essence, it allows the CPU to read from the 82C55A.

(WR) Write. A "low" on this input pin enables the CPU to write data or control words into the 82C55A.

(A0 and A1) Port Select 0 and Port Select 1. These input signals, in conjunction with the RD and WR inputs, control the selection of one of the three ports or the control word register. They are normally connected to the least significant bits of the address bus (A0 and A1).

82C55A BASIC OPERATION

A1	A0	RD	WR	CS	INPUT OPERATION (READ)
0	0	0	1	0	Port A → Data Bus
0	1	0	1	0	Port B → Data Bus
1	0	0	1	0	Port C → Data Bus
1	1	0	1	0	Control Word → Data Bus
					OUTPUT OPERATION (WRITE)
0	0	1	0	0	Data Bus → Port A
0	1	1	0	0	Data Bus → Port B
1	0	1	0	0	Data Bus → Port C
1	1	1	0	0	Data Bus → Control
					DISABLE FUNCTION
X	X	X	X	1	Data Bus → Three-State
0	0	1	1	0	Data Bus → Three-State

FIGURE 1. 82C55A BLOCK DIAGRAM: DATA BUS BUFFER, READ/WRITE, GROUP A & B CONTROL LOGIC FUNCTIONS

(RESET) Reset. A "high" on this input initializes the control register to 9Bh and all ports (A, B, C) are set to the input mode. "Bus hold" circuits internal to the 82C55A will hold the I/O port inputs to their logic "1" state with a maximum hold current of 400µA.

Group A and Group B Controls

The functional configuration of each port is programmed by the system software. In essence, the CPU "outputs" a control word to the 82C55A. The control word contains information such as "mode", "bit set", "bit reset", etc., that establishes the functional configuration of the 82C55A.

Each of the two Control blocks (Group A and Group B) accepts "commands" from the Read/Write Control logic, receives "control words" from the internal data bus and issues the proper commands to its associated ports.

Control Group A - Port A and Port C (Upper) (C7 - C4)

Control Group B - Port B and Port C (Lower) (C3 - C0)

The control word register can be both written and read as shown in the "Basic Operation" table. Figure 4 shows the control word format for both Read and Write operations. When the control word is read, bit D7 will always be a logic "1", as this is the control word mode information.

图9-2　Intersil 82C55A并行端口IO芯片的数据表（第3页）。Intersil公司版本所有，经许可后复制

命令寄存器（Command Register）的目的是允许程序员为IO芯片指定更为精确的动作。芯片可能提供许多选项，比如传输速度、缓冲区大小和错误处理技术，这些都是为了适合于不同的环境和各种不同的应用。这允许芯片的制造商提供单一产品，然后使用命令寄存器进行设置，使之适用于大量不同的应用。在系统初始化时，通过将相应的位模式写入到命令寄存器中，就可以做出特定的选择。有时，硬件将命令寄存器置为“只写”，这种情况下。如果在后面程序中想要更新寄存器中的某些位时，就存在一定的难度。

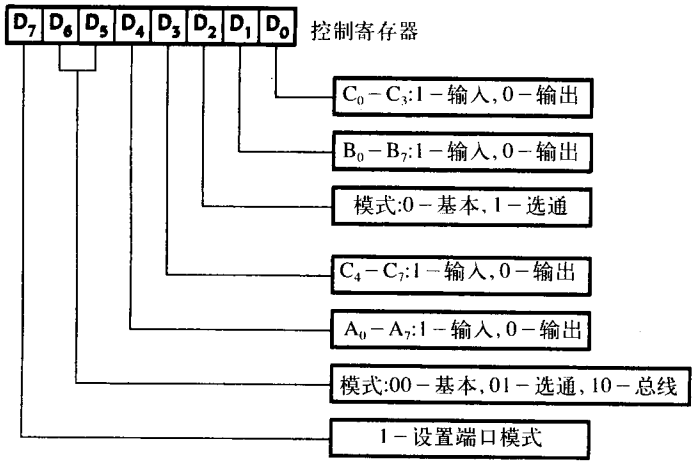
另一方面，提供**状态寄存器**（Status Register）的目的是为了让软件能够通过读取和测试IO芯片提供的状态标志来监视它。这些寄存器可以为“只读”，并不会造成什么太大的不便。

数据寄存器是数据进出计算机的“信箱”。一般地，我们希望输出到输出端口，从输入端口读入数据，但有时候，相反的方向也是允许的。

可编程外设接口（8255）最初是由Intel为8080微处理器生产的IO支持芯片系列的一员。尽管它早期的起源可以追溯到1970年，但是，如今在一些PC IO卡上——提供三个并行端口连接设备到计算机，依旧可以看到它的身影。图9-2中电路的示意性资料展示出三个双向端口，分别为A、B和C。端口C有各种不同的功能，在特定的条件下，可以拆分成两个4字节段。图9-2中的寻址表中，两条地址线A0和A1，可以看做是选择某个端口，而READ和WRITE控制线决定当前的访问是输入活动还是输出活动。使用可编程IO设备的好处，来自于它能够提供很大的灵活性。譬如程序可以控制许多主要的功能，包括它是作为输入端口还是输出端口。控制寄存器（或命令寄存器）必须在数据传输之前正确地设置好。有三种动作模式：

- 模式0：基本的字节宽度的输入端口和输出端口。
- 模式1：通过选通（异步）握手进行的字节传递。
- 模式2：三态总线动作。

A、B和C，这三个字节宽度的端口能够同时运行在不同的活动模式下。图9-3给出了控制寄存器每个位的功能，并提供一段C代码初始化8255，为例行IO操作做好准备。




```
// win-98。初始化0x1F3处的8255：端口A IN；端口B OUT；端口C OUT
// 初始化8255的命令寄存器
```

图9-3 8255 PPI控制寄存器的功能

PC机使用奔腾处理器提供的单独的64 KB端口地址空间，用IN和OUT机器指令来访问。C语言提供inp()和outp()扩展，来支持这些代码。其他处理器没有这种明确的地址空间，所有IO端口必须和RAM和ROM一样，映射到内存地址空间中。

图9-4给出的系统中的IO芯片是经过存储器映射的。这意味着可以将它看做内存来访问——不需

要使用专门的“端口入”和“端口出”指令。机器指令的MOV分组能够像处理内存存储单元一样，处理存储器映射后的端口，在这种情况下，寄存器位于下面的地址：

```
FC000003H:命令寄存器
FC000001H:状态寄存器
FC000002H:接收数据
FC000003H :发送数据
```

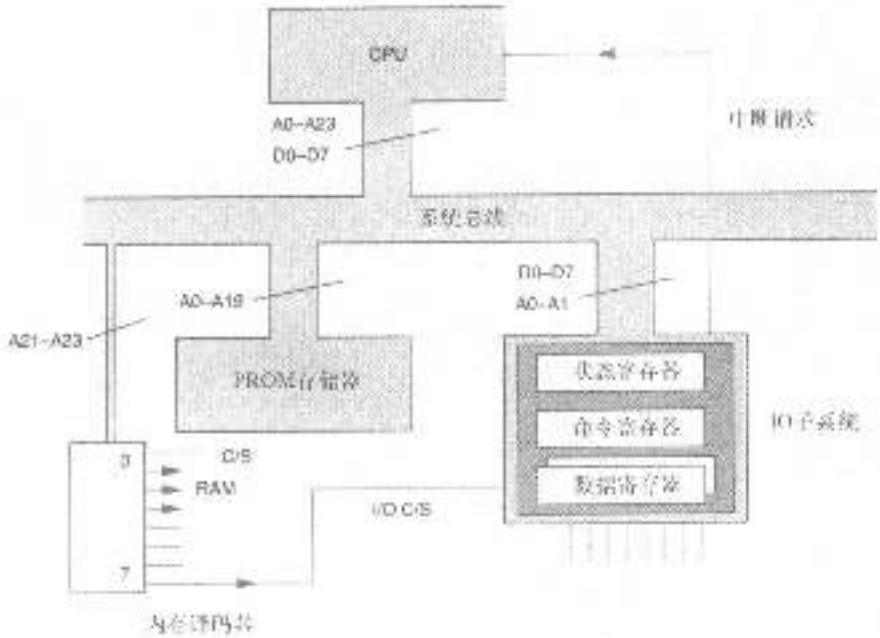


图9-4　访问存储器映射I/O中的寄存器

在分析图9-4中的内存译码电路时，我们会看到，总共16 MB地址空间被地址的三个最高符号位等分成8个2MB的页，高端的内存页为输入—输出设备保留，低端的7个页由RAM或ROM使用。正式的存储器映射表在图9-5中给出。注意，由于我们没有采用完全译码，IO芯片的行为会比较杂乱，并会对上面页中每个由四个地址构成的块做出响应。同一芯片有524 288个等同的地址，这种对次要的高信号的影响，使得我们如果不修改内存译码电路，在顶端的2MB页内根本就不可能安装任何其他设备。

设备	大小	地址引脚	主机总线	地址范围
PROM1	1 MB	20	000x xxxx xxxx xxxx xxxx xxxx	00 0000–0F FFFF
RAM1	2 MB	21	001x xxxx xxxx xxxx xxxx xxxx	20 0000–3F FFFF
RAM2	2 MB	21	010x xxxx xxxx xxxx xxxx xxxx	40 0000–5F FFFF
RAM3	2 MB	21	011x xxxx xxxx xxxx xxxx xxxx	60 0000–7F FFFF
IO	4 B	2	111x xxxx xxxx xxxx xxxx xx11	E0 0000–E0 0003
			别名	E0 0004–E0 0007
				E0 0008–E0 000B
				E0 000C–E0 000F
				...

图9-5　图9-4中所示系统的存储器映射

总之，命令寄存器含有可写的标志位，通过它们，程序员可以控制芯片的活动，一般通过在初始化阶段把特定的位模式写到寄存器中。而状态寄存器含有可读的标志位，提示芯片的活动和错误。数据寄存器，一般指定为输入或输出，是数据通往外部设备的窗口。

9.3 轮询：单字符IO

对于Motorola MC680x0，从输入端口读入数据到CPU寄存器的基本方法，就是简单地执行MOVE指令。和奔腾的MOV指令一样，它也需要源参数和目的参数来正确地传递数据。MC68000要求所有的IO端口安装后必须类似于内存存储单元，如6.8节所述，而在Intel奔腾处理器中，由于为端口提供独立的IO映射，所以必须显式地使用IN和OUT指令。

图9-6中的奔腾汇编语言代码，就是通过程序控制的**轮询**（polling）进行数据输入的例子。这种方式只能在运行MS-DOS或Windows 98的系统上工作，因为Windows XP和Unix出于安全上的原因，明确拒绝这种对硬件直接访问的方式。Windows XP中，操作系统的代码负责处理所有的IO操作，因此，IN和OUT指令被隐藏在XP或Unix设备驱动程序代码之内。在紧凑的轮询循环中，程序会不断重复检查状态寄存器内的接收就绪标志（RXRDY），直到它被硬件设为1（表示新的数据项到达数据接收寄存器中）为止。之后，循环停止，字节被从数据接收寄存器内读出，并进行零值检查。值为零表示当前数据传送结束。如果值为非零，则使用指针将数据项存储在数据数组内，然后继续之前的循环。

```
LOOP:    IN AX,STATUS            ;read status port
         TEST AL,RXRDY           ;test device status
         JZ LOOP                 ;if no data go back again

DATAIN: IN AX,RXDATA             ;get data from Rx port & clear RXRDY flag
         OR AL,AL                ;test for end marker
         JZ COMPLETE             ;jmp out if finished
         MOV [DI],AL             ;save character in data buffer
         INC DI
         JMP LOOP                ;back for more input
COMPLETE: ....                   ;character string input complete
```



在内存映射的I/O系统中，这段代码的C语言版本可能会是下面这样：

```
do {
  while (!(*(BYTE*)RXSTATUS & RXRDY)) { } ;/* wait until data arrives */
} while (*pch++ = *(BYTE*)RXDATA) ;         /* check data for a NULL */
```

图9-6 输入轮询的实例（汇编和C代码）

还要注意的一点是，IO端口硬件会检查对数据寄存器（RXDATA）的读取行为，清除RXRDY标志，以便为下个数据项的到达做好准备。

至此，如果对C代码不感兴趣，可以直接跳到GASP，以免下面有关C代码的内容令人头疼。

判断图9-6中哪段代码更清楚些可能会很微妙，更多是依据教条式的法则而非客观的证据。C代码明显要短些，或许我们需要对这段奇怪的C代码进行一些解释或调整，以便它更容易理解。使用C语言时，在引用如RXSTATUS和RXDATA等常量定义时，有时会产生一个问题。#define定义的常量默认为整数类型，但编译器不知道定义者期望什么样的类型。因而，需要使用(BYTE*)将类型从整型强制转换成指向BYTE的指针（BYTE不过是unsigned char的另一个名字而已）。其次，前导*号获得指针指向的值，从存储器映射的端口寄存器中提取所期望的字节值。如果不完成这些工作，好的编译器会抛出一条警告消息，我们当中爱追根究底的人可能会受到它的干扰。为了避免这种情况发生，我们必须采用这种古怪的*(BYTE*)类型转换结构。

需要强调的另外一点和C编译器相关——程序员可能会对此更感兴趣，在编写程序时，人们一般不会想到在声明状态寄存器和数据寄存器时，应用类型限定符volatile。但这有可能是必需的，因为编译器在注意到变量RXDATA和RXSTATUS只供读取，从不写入后，可能会对最终的代码进行优化。它们对编译器来说是常量，编译器有可能会对生成的机器代码进行调整，希望使之运行得更

有效率，因此，我们必须告诉编译器尽管RXDATA和RXSTATUS好像是常量，但实际上它们会被某些软件不可见的其他操作所更改：因此符合“volatile”资格。

GASP：

互补的版本，其输出数据的过程几乎完全相同，不过轮询的标志变为状态寄存器中的TXDATA标志，程序会一直检查该标志，直至其变为1，表示数据传输寄存器可以使用为止。接下来，下一个数据项被从内存移至TXDATA（Transmit Register，发送寄存器）中。此时，循环又重新开始。

由于CPU运行的速度比外设以及使用它们的人类要快得多，如果计算机完全依靠这种方法（称为循环轮询）与外设进行通信，那么计算机绝大部分时间将会花在无用的轮询循环中：依次测试每个设备，检查它们是否就绪和请求服务。这就如同振铃坏了的电话一样：打电话完全没有问题，但确定有没有电话的惟一方法，就是不断地拿起听筒，通过听来检查是否正好有人在打电话。在决定使用哪种IO技术之前，必须考虑到事件可能的发生频率，以及错过事件所带来的损失的严重性。表9-3给出常见活动的不同频率以供参考。

表9-3　各种操作的相对速度

系统总线的操作	500 MB/s
字符块的传送	100 MB/s
以太网数据传输	10 MB/s
电话通话	8 KB/s
串行线的常见速率	1 KB/s
爱普生打印机	100 B/s
键盘发送字符	4B/s

我们将定期对设备进行查询的方式称为**间歇式轮询**（intermittent polling）。在许多家用的嵌入式系统、环境监控设备或交通计数系统中，常常会使用这种技术。轮询的间隔根据应用的需要选定，可能每秒一次（1Hz），或者更快。有时，周期的时序由比较具有附属性的需求决定，如将前面板的闪烁降到最低。通过间歇式地轮询设备，循环轮询过程中时间的浪费得以避免。这种方法工作得很好，并且避免了中断处理的复杂性。这个建议稍微有些不坦白，因为要达到25ms的间歇式轮询频率，依旧需要中断驱动的时钟的协助，因此，这样不过是以一种中断来替代多个中断！但在论述不同技术的优点和价值时，最好清楚地区分**专注式**（循环）和**间歇式**（定时）轮询（见图9-7）。

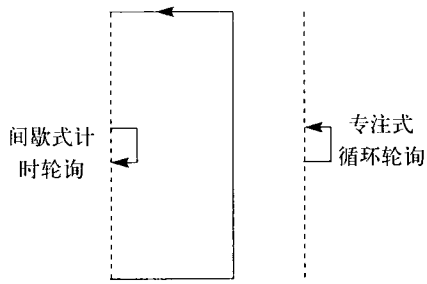


图9-7　专注式和间歇式轮询的对比

一些设备，比如雷达回声处理系统，需要对接收到的数据做出快速的反应，甚至于中断响应都太慢。在这种情况下，只有快速的专注式循环能够满足要求。

试图从设备读入数据时，在开始时确定是否采用**阻塞**（block）方式很重要。阻塞是指我们命令程序等待，直到数据到来为止。HLL程序中的一些调用即采用阻塞方式，比如scanf()，在没有得到所要求的参数之前不会返回。在多任务的系统中，宿主操作系统处理IO的复杂细节，但是，对于程序员，感觉起来依旧如同专注式的轮询循环。如果程序有许多事情要做，那么这种一心一意的做法可能就不适合。一种简单的解决方案是，确保在进入阻塞读（如scanf()）之前有数据等待被读取。输出（如调用putc()）也会发生相同的情况。如果输出设备没有准备好分发数据——或许由于它依旧忙于传送之前的数据，它将会阻塞写请求，不会返回，直到它空闲下来提供服务为止。为了避免程序在每个输入输出函数上阻塞，操作系统可以执行可用性检查。有好几个函数可供检查之用，比如kbhit()或GetInputState()。遗憾的是，这些状态检查函数不是标准C库调用的一部分。每个操作系统都有它自己的状态检查函数。

从图9-7中，可以看出间歇式轮询和专注式轮询有清晰的差异。尽管在外设比CPU慢得多的情况下，专注式轮询浪费了许多CPU时间，但中断驱动的IO和自治的DMA对硬件和软件的支持要求更多。由于轮询容易实现，而且容易测试和调试，因而一般都将它作为首先的选择。图9-8和9-9给出一些直接通过轮询硬件状态标志来处理数据IO的C代码。

图9-8和9-9中列出的代码片段，说明了直接访问硬件的轮询式IO的使用。在IO设备没有就绪时，代码中操作有可能被阻塞的两处已经用注释标出。一般地，这些代码都隐藏在操作系统中。直接与

裸硬件一同工作的情况，如图9-9所示，仅在处理微控制器开发时才会用到，这种应用中，应用程序的大小决定了它不会提供全部的操作系统功能，比如文件系统或用户密码管理。

```
/* io.h    68k header file with h/w definitions */

/* messages */
#define  PAPER_OUT       -1
#define  DE_SELECT       -2
#define  YES             0
#define  NO              -1
#define  OK              0

/* address, offsets and setting  for M68681 DUART */
#define  DUART           0XFFFF80 /*base address */
#define  ACR             9        /*aux control reg */
#define  CRA             5        /*command reg A */
#define  MRA             1        /*mode reg A */
#define  CSRA            3        /*clock select A */
#define  SRA             3        /*status reg A */
#define  RBA             7        /*rx reg A */
#define  TBA             7        /*tx reg A */
#define  RXRDY           1        /*bit mask for rx ready bit */
#define  TXRDY           4        /*bit mask for tx ready bit */

/*Settings for the Motorola M68230 Parallel Interface Timer
  These only deal with mode 0.0, and for ports B and C
  No details about the timer.
*/

/* PI/T offsets and adresses, PIT registers are all on odd addresses */
#define  PIT             0Xffff40 /*address of PI/T */
#define  BCR             0Xf      /*offset for port B cntrl Reg*/
#define  BDDR            7        /*offset for B data direction*/
#define  BDR             0X13     /*offset port B data reg */
#define  CDR             0X19     /*offset port C data reg */

/* Parallel port settings masks and modes */
#define  MODE0           0X20     /* mode 0.0, 2X buff i/p, single buff o/p */
#define  MODE01X         0X80     /* mode 0.1X, unlatch i/p, 1X buff o/p */
#define  OUT             0XFF     /* all bits output: 0 - i/p, 1 - o/p */
#define  STROBE_MINUS    0X28     /* strobe printer -ve */
#define  STROBE_PLUS     0x20     /* strobe printer +ve */
#define  PRINT_ST        1        /* paper out pin 00000001 */
#define  PAPER_ST        2        /* paper out pin 00000010 */
#define  SELECT_ST       4        /* selected pin  00000100 */
```

图9-8 68k单板机的头文件

```
/*Initialization and data transfer routines for 68k SBC */

#include "io.h"

/* set up Mc68681 DUART serial port A only */
void dinit() {
   register char *p;
   register int i;

        p = (char *)DUART;
        *(p+ACR) = 128;          /* set baud rate */
        *(p+CRA) = 16;           /* reset Rx */
        *(p+MRA) = 19;           /* no modem, no PARITY,8 bits */
        *(p+MRA) = 7;            /* no ECHO, no modem cntrl, 1 STOP */
        *(p+CRA) = 5;            /* enable Rx & Tx */
        *(p+CSRA)= 187;          /* Rx & Tx at 9600 */

        p = ( char *) PIT;       /* set to base address of PI/T */
        *(p + BCR ) = MODE0;     /* mode 0.0 */
        *(p + BDDR ) = OUT;
```

图9-9 C程序示例（使用轮询IO，直接访问硬件）

```
        for(i=0; i != 1000;i++) ;/* init delay*/
}

/* set up 68230 PIT for print out on port B */
void pinit() {
   char *p;
        p = ( char *) PIT;        /* set to base address of PI/T */
        *(p + BCR ) = MODE0;      /* mode 0.0 */
        *(p + BDDR ) = OUT;
}

/* get char from serial port A  returns character */
char get() {
   register char *p;

        p = (char *)DUART;
        while ( !( *(p+SRA) & RXRDY )) {};  /* block here */
        return *(p+RBA);
}

/* put character c to serial port A */
void put( char c) {
   register char *p;

        p = (char *)DUART;
        while ( !( *(p+SRA) & TXRDY )) {};  /* block here */
        *(p+TBA) = c;
}

/* put string to serial port A using put routine */
void puts(char* p) {
        while( *p )
                put(*p++);
        put('\n');
}
/*put character to parallel port */
int print (int c) {
  register char * p ;

      p = (char *) PIT;
      while ( *(p + CDR) & PAPER_ST) )
      {
          if ( !( *(p + CDR) & PAPER_ST) )
                return (PAPER_OUT) ;
          if ( !( *(p + CDR) & SELECT_ST) )
                return ( DE_SELECT);
      }
      *(p + BDR) = c;              /*send data */
      *(p + BCR) = STROBE_MINUS;/* strobe positive */
      *(p + BCR) = STROBE_PLUS; /* strobe negative */
      return OK ;
}
```

图9-9　(续)

9.4　中断处理

如我们在8.8节所述，我们可以安排一个专门编写的子例程，当有外部触发的**中断**信号时，调用它提供服务。大部分CPU都提供一条或多条中断请求线，它们在保持系统的运行、维护面向所有客户的服务中担当重要功能。我们可以将中断想像成电话铃（见图9-10）。正当您全神贯注地洗盘子时，电话铃响了，您放下抹布，跑到电话旁和朋友聊上一会，结束后，您将电话听筒放在托架上，然后继续洗盘子。盘子完全没有受到这次中断的影响。或者，操作系统可能会介入，不将控制权交还给最初被中断的任务。如果是您妈妈提醒您送张生日卡片给您的姐姐，您可能决定接完电话后切换任务，将盘子推迟到晚些时候优

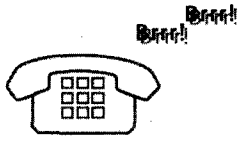


图9-10　电话干扰（中断）

先级允许时，再继续执行。

中断信号到达CPU时，会强制它停止执行当前的程序，切换到另外的ISR（Interrupt Service Routine，中断服务例程）。ISR完成后，CPU会继续执行被中断的程序。中断工作的方式十分简单，尽管在启动时需要执行的初始化工作可能会很为复杂。

系统常常有多种中断源，如图9-11所示。这使得在中断发生时确定是由哪个设备请求中断比较困难。同时，多个中断请求同时发生的情况下，如何处理也是一个难点。此时，系统必须做出决定，哪个立即服务，哪个向后推迟。

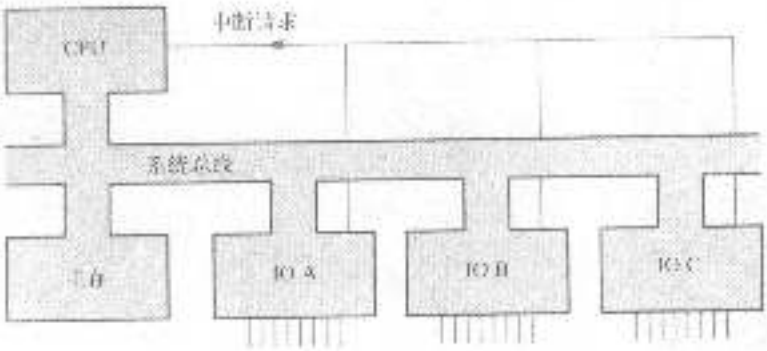


图9-11 中断设备到CPU的连接

我们可以使用硬件来划分输入中断信号的**优先级别**，参见图9-12。海报**PIC**（Programmable Interrupt Controller，可编程中断控制器）芯片只允许单个来自于最紧急设备的**IRQ**通过它传送到CPU（参见图9-13）。即使如此，如果CPU已经禁止了所有中断处理，它依旧不会接受请求，如图7.9所示。每种处理器有一个标志，表明CPU是否准备好响应中断。用专门的一对指令STI和CLI来控制这个标志位，它们分别设置和清除中断标志。仅只有拥有特权的用户才能使用这些指令。

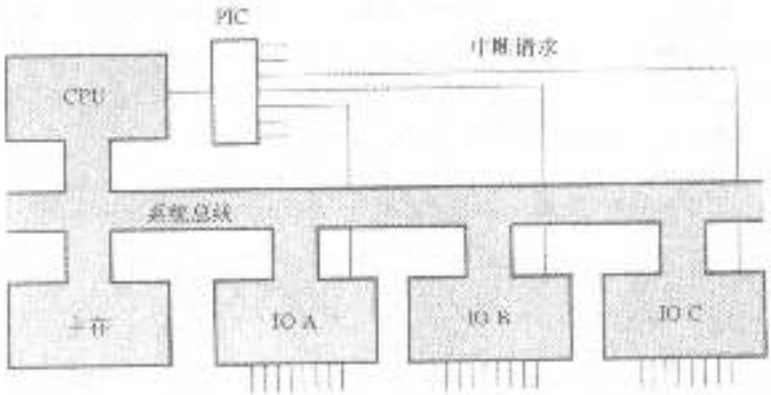


图9-12 中断优先级复分硬件

中断被CPU接受以后，还存在确定请求来源以及在内存中定位正确ISR的问题。PIC芯片会给出哪个IRQ对本次中断负责，然而，在CPU询问标识符时，它并不返回IRQ编号，而是返回一个8位数字，称为向量。CPU就使用它来访问内存中的中断向量表（Interrupt Vector Table，**IVT**）中正确的入口。IVT数据表保存着所有ISR入口点的地址。每种不同的中断源在IVT中都有惟一的入口。

在向PC添加更多设备时，PC提供的中断线过少这个问题就会显现。但到目前为止，人们一般都能克服这些问题——常常通过禁用现有的设备来实现，甚至拔下电路板以共享一个IRQ。在安装新卡时，IRQ冲突是一个常见的问题。

整数编号	功能	来源
77	硬盘2	IRQ15
76	硬盘1	IRQ14
75	8087	IRQ13
74	PS/2鼠标	IRQ12
73	声卡	IRQ11
72	网络	IRQ10
71	重定向	IRQ9
70	RTC	IRQ8
...		
18	BIOS/打印	INT
17	BIOS/软引导	INT
16	BIOS/打印	INT
15	BIOS/键盘	INT
14	BIOS/通信	INT
13	BIOS/磁盘	INT
12	BIOS/内存大小	INT
11	BIOS/检查	INT
10	BIOS/视频	INT
0F	LPT1:	IRQ7
0E	软盘控制器	IRQ6
0D	声卡	IRQ5
0C	COM1:	IRQ4
0B	COM2:	IRQ3
0A	...	IRQ2
09	键盘	IRQ1
08	系统定时器	IRQ0
07		
06		
05	将屏幕传输到打印机	
04	数字溢出	
03	断点	
02	NMI电源故障	
01	单步跟踪	
00	整数除法错误	

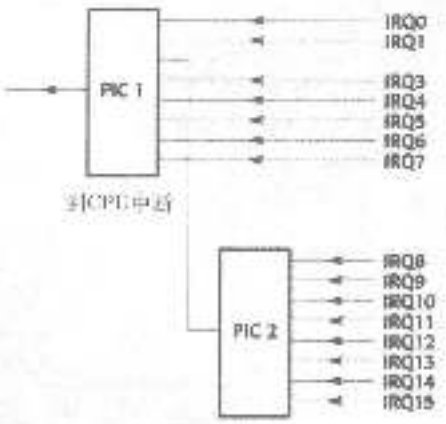


图9-13　PC中断向量表（部分）和相应的PIC连接

IRQ1拥有最高的优先级，IRQ15优先级最低。不是所有的IRQ线都映射到ISA和PCI扩展总线。

IRQ0：指定用于系统的主定时器，它生成的时钟信号使操作系统（Unix、Windows 95/98/NT/XP）能够在进程用完分配的时间片后，重新获得控制权。

IRQ1：指定用于键盘控制器。

IRQ2：指定用于串联次级PIC（提供IRQ8-IRQ15）。IRQ9承担了IRQ2输入的角色，但为了避免冲突的发生，并不常用。

IRQ3：指定用于COM2串行端口，但更广泛用于调制解调器，外置或内置，因而发生冲突的机率也更大。可以将调制解调器重新分配到未用的IRQ上，或在BIOS设置参数中禁止COM2，以避免这类问题。有时，声卡会试图占用这个IRQ，这令问题更加复杂。另外的问题是，很遗憾，COM4也同样被指定使用这个IRQ。

IRQ4：指定用于COM1/鼠标，它是一个RS232端口，常用于串行鼠标。和COM2/COM4的情况类似，这个中断与COM3共享，由于调制解调器常常预先配置为使用COM3，因而肯定会和串行鼠标发生冲突。解决方案是使用PS/2鼠标或USB鼠标。

IRQ5：指定用于LPT2——最初为打印机准备的第二个并行口，但现在更多用做声卡。没有多少PC机需要两台打印机，但是，并行口也可以作为其他接入式设备的接口，因而依旧有可能发生冲突。

IRQ6：指定用于软盘控制器（Floppy Disc Controller，FDC）。

IRQ7：指定用于LPT1，为并行打印机端口。

IRQ8：指定用于硬件计时器——为实时系统的程序员准备。

IRQ9：常用于网卡以及其他基于PCI的设备。

IRQ10：可以并且常用于网卡、声卡或SCSI适配器。也可以供PCI总线使用。

IRQ11：同IRQ10。

IRQ12：如果有的话，指定用于PS/2鼠标，否则类似于IRQ10和IRQ11。

IRQ13：指定用于数学协处理器（8087）。

IRQ14：指定用于第一个IDE磁盘控制器。

IRQ15：指定用于第二个IDE控制器。

Windows提供一个非常有用的管理工具，它允许我们检查IRQ的分配情况，以及其他一些有意义的参数。要运行该管理工具，执行下面的命令：

Start→Programs→Administrative Tools (Common)→Windows Diagnostics→Diagnostics→Windows Resources→IRQ

图9-14列出了一台Windows NT机器的IRQ清单。如果需要更多细节，可以点击Properties（属性）按钮。在本章后面，我们还将返回这个视图，查看DMA画面。

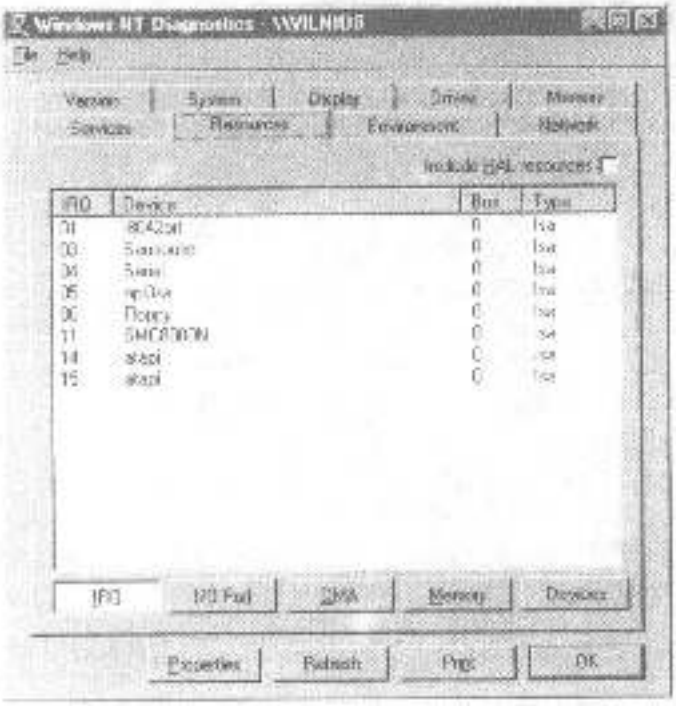


图9-14 使用Windows NT显示PC的IRQ

对于外部IRQ中断，每当CPU识别那条中断线被激活，如果同时有几个设备连到同一IRQ线上，那么仅靠这些信息难以识别发出请求的是哪个设备。每个设备都能引发这个优先级别的中断。一种解决方案是在ISR内查询每个可疑设备，直到找到引发中断的设备为止。由于时间比较宝贵，这不是一个理想的方案。更好的方案（MC68000即使用这种方案）是每个设备都保存它们自己的中断向量（Interrupt Vector），而不是将责任都集中到PIC中。采用这种方案时，在中断确认周期内，CPU会请求设备验证它自己。CPU广播一个中断确认信号，询问该优先级别的任何设备中，哪个设备有一个活动的中断请求未决，由设备自己标识出来。设备的响应为8位的中断向量，它是IVT偏移值，表示相应的ISR入口地址。为了支持8位的中断向量，态与向量化中断的游戏，外设芯片必须提供一个向量寄存器。并非所有IO芯片都能够做到这一点。

图9-15给出奔腾处理器、PIC、IVT和外设之间的关系。请求中断号的过程——检索地址表并跳转到某个条目——显得有些费力。为什么PIC不简单地返回ISR的32位完整地址给CPU呢？这里可能存在一些我所不了解的历史上的原因。追溯到20世纪70年代，当时的PIC将完整的含ISR地址的CALL指令放到总线上，以响应中断确认信号，这会强制8080 CPU去执行相应的ISR。由于采用这种方法时，CPU需要从PIC中（而非主存）读取3个字节的中断信息，使得这种方法十分不灵活，因而被弃之不用。但是，一般看来，当前使用的PIC中断方案依旧很受限制。

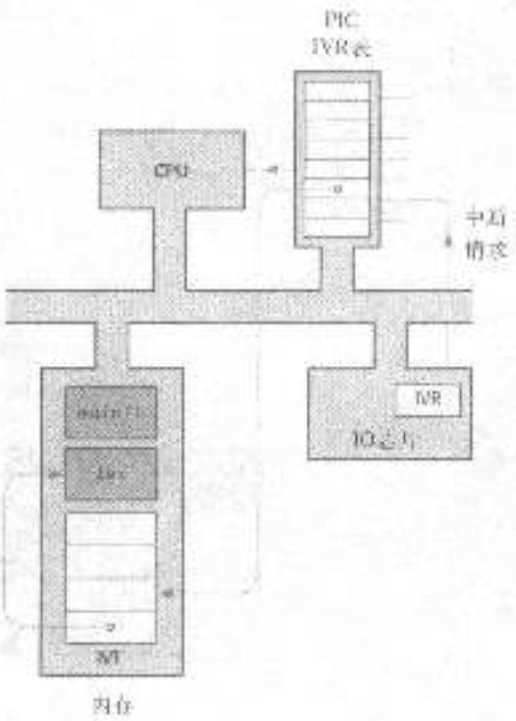


图9-15　定位中断服务例程

进入ISR后，第一件需要完成的任务，就是保护那些可能在ISR中使用的CPU寄存器不被破坏（corruption）。要完成这个任务，只需简单地将这些内容压入堆栈中，并在中断处理完成后从中恢复出来即可。接下来需要验证中断的来源，这项任务可以通过检测外设的标志位以及被禁除中断的起源来确定。在完成这一步之前，始终存在受同一来源的重复中断影响的危险，即使所有的处理过程完成后也是如此。接下来，可能必须重新初始化（reinitialize）外设，以使之能够处理另外的中断请求，尽管有时这并非必需。ISR结束的标志是将保存的寄存器压出堆栈，执行rte指令恢复PC的值。

8086要求IVT从0000开始。但是，80386通过提供中断描述符表寄存器（Interrupt Descriptor Table Register，IDTR），引入将IVT重定位到主存的任何地方的机制。如果要移动IVT，只需将内容复制到新的位置，做出任何必需的修改，并将IDTR指向新表的基地址。

如果一个设备只需偶尔关心一下，那么使用中断完全能够胜任：外设仅在需要CPU时，才要求CPU的帮助。使用中断的确需要安装专门的硬件，从主程序到ISR的来回切换也是一项费时的活动，每次常常都要花费几微秒的时间才能完成。

。许多外设都能被配置为使用CPU的中断机制，但如果有太多的设备触发中断请求，就会出现新的问题。当几个请求同时到达时，必须划定设备的优先级别，并据此对请求进行排队，等待服务。判定中断来源时，如果硬件不能返回中断向量，则要显式地查询设备。一般情况下，直接使用VBR和IVT定位出内存中恰当的中断服务例程。

表9-4　可能的中断源

1. IO数据传输请求
2. 软件TRAP（SVC）
3. 页故障
4. 实时时钟
5. 运行时软件错误
6. 系统重置或定时器

中断的各种来源可以划分成六类，在表9-4中列出。

一些小型的微控制器不允许中断处理过程自身再被中断，能够在任何时候接受和处理中断的能力，称为中断嵌套。在这种情况下，高优先级的设备，比如硬盘，能够打断低优先级别的服务例程，不需等待。

现代中断机制的另一项重要应用是访问模式的实现。通过CPU提供的功能，结合内存段的“用户”或“特权”状态，系统能够保护它自己的核心功能不受普通用户的影响，将风险最大的功能保留给“root”（或超级用户）。图9-16展示出这种应用，优先级别用同心圆表示。这种访问控制是现代多用户操作系统的基石，它的实现依赖于所有能够将CPU切换到特权模式的中断。

中断为应用软件访问系统功能提供一个入口，我们可以通过中断服务例程来管理对操作系统功能的访问。中断服务例程现在已经成为操作系统的一部分，这可以看做是软件中断或TRAP指令的最重要应用。

另外的问题是系统中断例程的调试。对于常规的子例程，程序员应该能够确切地知道每个子例程什么时候运行，尽管IF-THEN-ELSE语句会根据变化的环境条件选择不同的路径执行。实际上，有些例程可能从来不会执行，原因多种多样，可能由于恰好是闰年，或者瑞典不接受欧洲旅行护照。不一而足，中断服务例程（Interrupt Service Routine，ISR）的任务是在CPU中断信号被激活时执行，这种情况有时超出了程序员的控制范围。因此，我们说子例程是可预测而且是同步执行的，而中断服务例程则是异步的、不可预测的。这使得调试几个中断可能在任何时候激活的软件系统需要很高的技巧，尤其当调试器软件为其自身的目的而使用跟踪中断时，情况会变得更混乱。

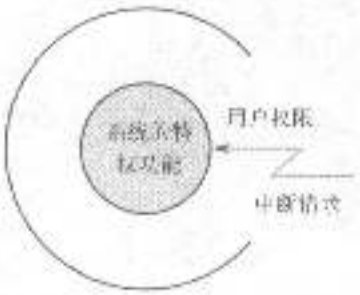


图9-16 中断如何让操作系统更安全

由于中断发生的不可预测性，以及给全面测试系统带来的困难，数字计算机上中断的应用已经被认为很危险。因而，程序员在为性命攸关的应用构建嵌入式计算机系统时，常常方图避免依靠中断机制。但是，在风险不大的领域中，硬件中断是一种有效的技术，能够用来实现反应迅速的多任务系统。Windows和Unix提供一些查看工具，我们可以用它们来查看本机上中断处理的速率。Unix有perfmeter，在3.2节曾经介绍过它，Windows提供性能监视程序，如图9-17所示。

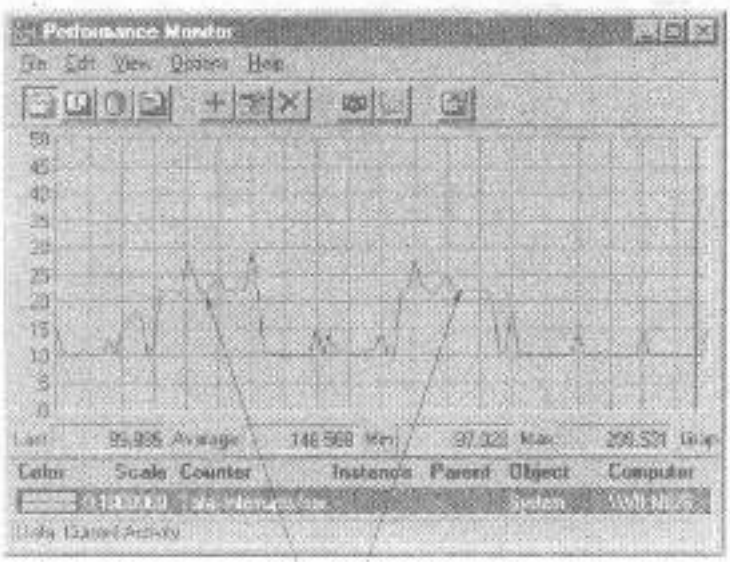


图9-17 Windows NT上的中断活动

这个管理工具允许我们监视IRQ的分配，以及其他一些有意义的参数。使用下面的序列启动它：

Start→Programs→Administrative Tools (Common)→Performance Monitor→Edit→Add to Chart→Interrupts/s

在图9-17中，中断活动以0.1s为间歇显示运行中的平均值。在晃动鼠标时，它会产生一连串的中断，平均值也会随之上升。在使用键盘或任何其他与PC中断系统紧密结合的设备时，也同样会造成平均值增加。

9.5 关键数据的保护：如何与中断通信

从8.8节我们已经开始提及，从ISR返回数据时存在特定的问题。考虑图9-18中的情形，这是一个

显示一天内的时间的程序，它由两部分组成。一部分为实时时钟（Real-Time Clock，RTC）的中断服务例程，它每隔10ms被触发一次，增加秒、分钟和小时的值；另一部分为显示例程，它负责刷新数字面板上显示当天时间的一系列数字，微秒、秒、分钟和小时数据都保存在内存中。刷新显示内容的例程定期运行，大约每秒钟两次，属于低优先级的任务，或许在主处理循环中执行。它读取当前的秒数，将它转换成7段的格式，并将它输出到显示屏。然后，它会依次对分钟和小时做同样的事情。

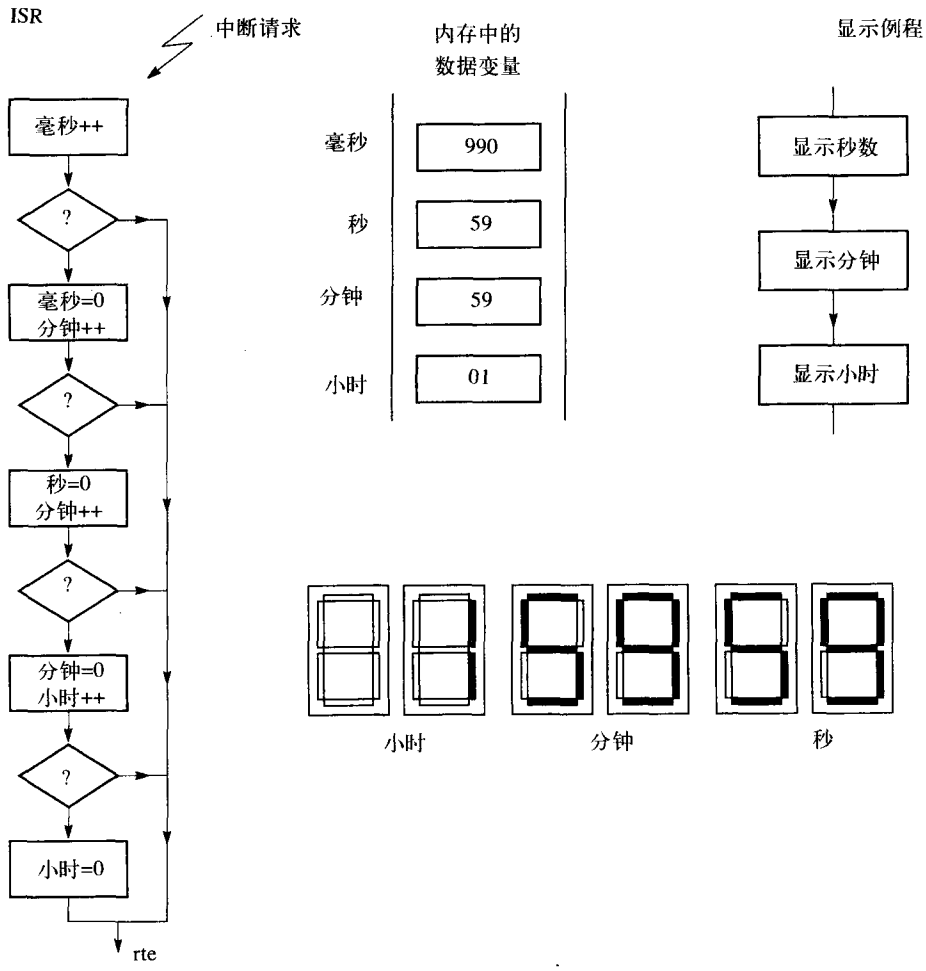


图9-18　实时时钟和时间显示

RTC ISR更紧要，它会打断任何正在运行的处理过程，更新主存中的时钟数据。现在，如果RTC中断发生时显示屏正在刷新，会发生什么事情呢？考虑时间恰好为01：59：59的情况，显示屏刚刚更新完新的秒数和分钟数，正准备从内存中读取小时数——正在此时，RTC ISR发生了。它依次将各个时间值递增，毫秒数变为00，然后秒数变为00，然后分钟数变为00，最后小时数变为02。完成这些工作后，它会将控制权返回给主程序。而此时恰好是刷新小时显示的时候。这会造成人们看到不正确的值02:59:59。这个错误的值会保持在那里，直到下一次显示刷新纠正它为止。在这个例子中，内存中的时钟值没有被破坏，所以没有造成持续性损坏（除非您因此而错过了火车）。如果显示更新例程还服务于网络连接，那么将会产生大范围的灾难。

上述问题的产生要归因于所处理的数据的复杂性，以及两个或更多进程并发非同步访问的需求。推而广之，在为Unix或Windows编写多任务应用程序时，也会遇到类似的问题。我们将这类问题称

为**临界资源**（critical resource）问题。值得注意的是，在这种情形下，如果将时间简单地以32位整数的形式存储，那么不会发生任何问题，因为值永远是合法的：重叠的更新和显示操作不会导致数据的破坏，除非计算机拥有多个处理器！

表9-5 保护临界资源的解决方案

1. 关闭中断
2. 将访问串行化
3. 使用信号量

为了解决这个难题，有几种解决方案可以尝试，如表9-5所列。

如果在显示屏更新例程的入口处将中断禁止，从而阻塞RTC中断请求，刷新完成后再重新恢复中断，则显示的内容会保持一致，但会慢慢地走向错误。当系统接收到RTC中断时，如果中断恰好被禁止，那么系统对它的响应会被推迟，因而TOD就有可能变得不正确。在时钟的计时信号快于100Hz时，这个问题更加严重。实际上，关闭所有的中断太过激烈，因为这样将会影响到其他中断驱动的活动，比如IO传送。这种方案仅仅适用于小型的微控制器系统，此时中断的推迟不会产生什么影响。

对于这个例子，更好的解决方案是将对临界数据区域的访问串行化，如图9-19所示。这意味着

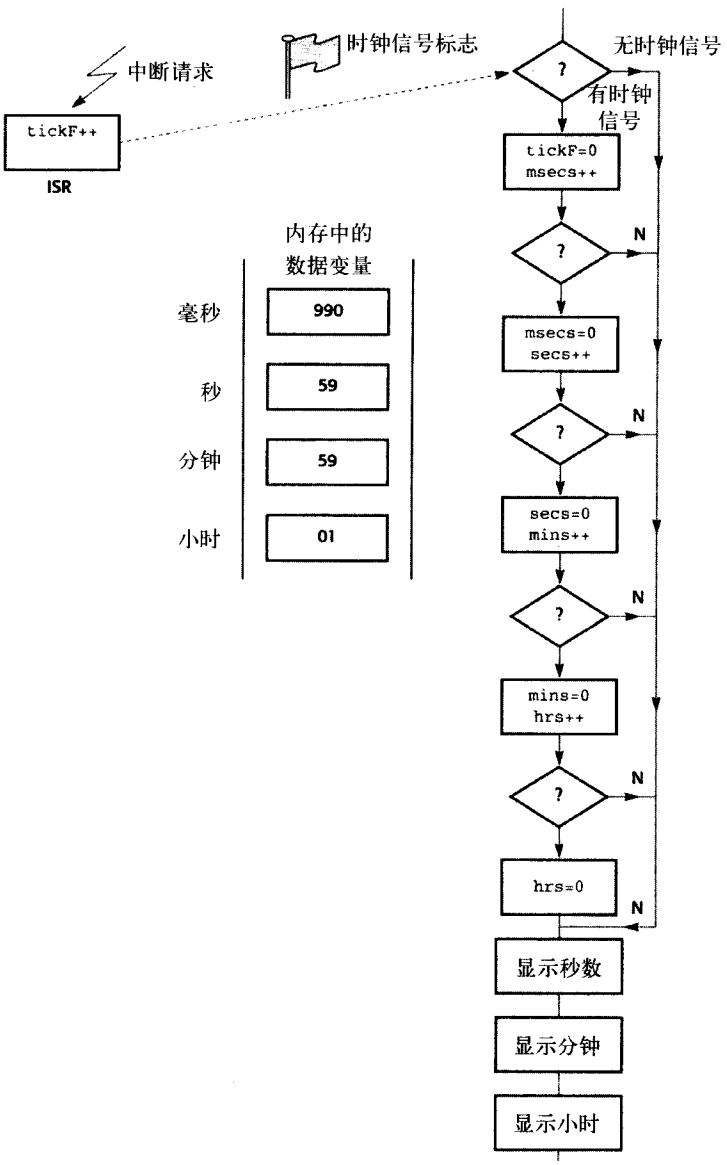


图9-19 使用时钟信号标志

将所有代码从RTC ISR移到主程序流程中，放在显示刷新例程之前。这样就不可能读到部分更新的数据。现在，RTC ISR必须用一个标志或简单的整数，来标识自上一次显示更新以来，是否曾接收到时钟信号。通过检查RTC标志的值，就可以确定是否需要对时钟递增。

我们要认识到，时钟信号标志本身就是一个临界资源，也有可能会遭受破坏。为了协助完成这项任务，奔腾处理器为指令集中的一些指令提供一种特殊的不可中断模式，这些指令包括BTS（Bit-Test-and-Set）、BTR（Bit-Test-and-Reset）和BTC（Bit-Test-and-Complement）。它们对于测试和重置单个标志位很有用，并且当前面有LOCK指令时，即使执行期间接收到中断或DMA请求，也能保证运行完成。

9.6　缓冲IO：驱动中断设备的驱动程序

在计算机安装有复杂的多任务操作系统时，不管是简单的硬件轮询IO，还是中断例程，应用程序的程序员都不能直接访问。前者将会很浪费，而后者是被阻止的，因为允许常规用户控制中断系统会带来巨大的风险。所有进出计算机的数据传输工作，都必须由可以信任的操作系统例程（即**设备驱动程序**）来处理。在Unix和Windows中，这些例程都运行在特权级别，负责直接访问硬件，为上层提供接口，其中包括流量控制，以及在必要的情况下进行错误捕获。它们还常常要提供缓冲数据缓冲区，以平滑数据流中的断续问题。在第17章，我们将会知道为什么IO中断和设备驱动程序服务例程对于进程调度方案至关重要。应用程序的程序员要么通过HLL库函数，如printf()或putch()，要么通过专门提供的需要显式链接到代码中的库函数，来获得对设备驱动程序例程的访问。图9-20给出设备中断驱动程序大致的结构和行为。我们可以看到，设备驱动程序的代码分成两部分：前端接口和中断服务例程。在这之间是数据缓冲区——数据被送出前的临时存储区。

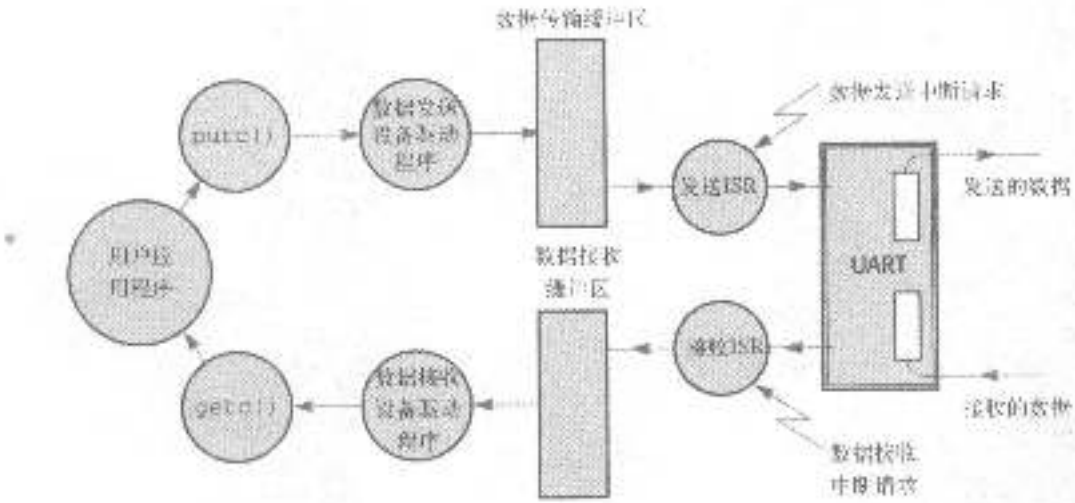


图9-20　带缓冲的中断驱动IO

以图9-20所示的数据输出操作为例，应用程序以数据项为参数调用输出函数（putch()）。之后，控制权传递给库例程，库例程会调用操作系统例程。操作系统的相关例程会检查与指定设备相关联的传输缓冲区是否有足够的空间。如果有空间，则将数据项加入到队列的后面，更新队列的指针后，控制权返回到最初的程序。如果缓冲区内恰好没有空间（或许是链接失败造成的），则返回一个错误状态码。数据会呆在传输缓冲区内，直至传输ISR将其传送到输出端口为止。传输ISR由端口硬件在准备好接收后续数据时触发。因而，应用程序可能满意地将一些数据输出到某个设备，但数据实际上可能只是呆在传输缓冲区内。

接收通道的操作次序类似于发送。数据到达接收端口后，会触发一个中断，调用接收ISR，ISR将数据移到接收缓冲区，然后应用程序就可以读取它。

9.7 直接内存访问：自治的硬件

对于大批量的数据，或高速的传送，直接内存访问（Direct Memory Access，DMA）方法比较适用。使用这种方法时，数据不经过CPU，直接经总线从源传送到目的地。CPU甚至不负责执行一个个的数据传送周期，但系统中必须有独立的DMA控制器，负责管理这样的数据传送。总体的操作依旧由CPU发起。这种自治活动极大地降低了CPU的处理负担。DMA控制器能够独立地管理读一写总线周期，生成地址以及控制总线传输——将数据从源移到目的地所需要完成的全部事务。为完成这个任务，它需要使用总线请求机制从CPU那里"借用"整个总线。DMA活动有可能延迟正常的CPU读取-执行周期，但它也可以智能地仅在CPU不需要总线时占用总线。PC机一直受累于十分过时的由两个古老的8237控制器提供的8位DMA。更快的16位Ultra DMA现在已经取代了原始的8位设备。分立的部件已经连同APIC以及其他一些系统支持设备一同集成到VLSI系统芯片内。每个PC DMA设备（见表9-6）提供4个独立的通道，但由于两个设备是层叠的，因而只有7个可用。通道4用来连接第二个设备。

表9-6 PC DMA通道的分配

通道	功能	宽度
0	DRAM刷新	8位
1	声卡	8位
2	软盘驱动器	
3		
4	连接到第一个DMA控制器	
5	声卡	16位
6		
7		

CPU依旧负责启动数据传输。它会载入DMA控制器内的几个寄存器，发动这项行为。

数据传输常常在IO部件和内存之间发生，见图9-21。DMA部件直接通过握手信号线控制IO部件，但它需要知道目的数据在内存中的地址，以及需要传送的数据的数量。由于DMA数据传送能够和主CPU并行进行，因此CPU并不知道什么时候传送结束。因而，DMA控制器使用中断通知CPU数据传送任务完成。前面的两种方法中，所有的输入输出数据必须通过CPU的寄存器。但DMA直接沿总线从源到目的地传送数据，省去了通过CPU数据寄存器这段耗时的典型历程。这样做看起来是一个好的思想，因为它将CPU解放出来从事其他活动，但在支持缓存以及虚地址的系统中，还有一些根本性的问题需要解决。这些机制将在第12章详细介绍。

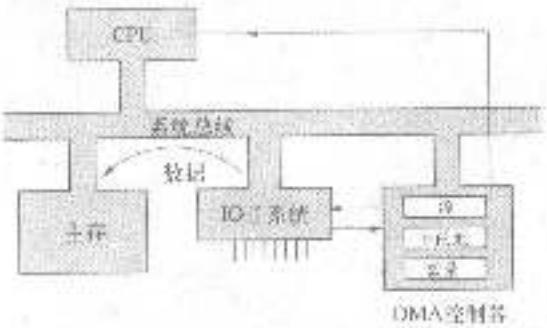


图9-21 使用DMA传输数据

商业的DMA控制芯片一般提供多重通道，允许数据传送并发进行。令人好奇的是，IBM PC将一个DMA通道保留用做DRAM刷新。DMA操作需要总线时间，这就会与CPU以及其他DMA控制器产生竞争。时间的分配可以以若干块为单位进行，我们称之为脉冲串，在此期间，CPU被完全排除在外，不能访问主存。时间的分配也可以以周期为单位进行，这样可以将DMA对存储器的访问安排在CPU活动的间隙中进行。从而避免对主处理过程产生严重影响。对于奔腾处理器，总线争夺问题由于L1高速缓存的引入已经有所改善，L1高速缓存允许CPU继续读取指令并执行，只要高速缓存内保存的本地指令满足CPU执行的需要。

为了进一步将CPU从慢速的IO中解放出来，人们开发出更智能的通道处理器。它们具有自治能力，包括设备轮询、通道程序的执行（包括代码转换）、中断激活以及数据和指令的DMA。将输入时间、处理时间和输出时间画在条形图中，见图9-22。重叠操作的优势显而易见。

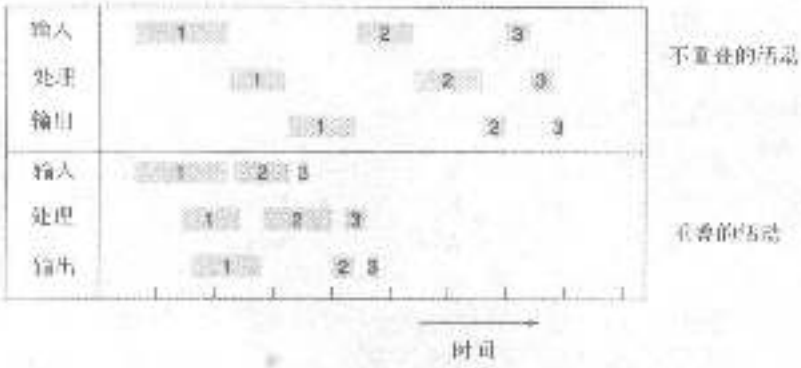


图9.22　处理过程与输入输出同时进行的优势

通道处理器是半独立的处理器，仅仅偶尔与主处理器通信。现在，我们可以将它们看做是类型不断增加的专用协处理器（浮点处理器，LAN接口处理器，图形处理器等）中的一种，两类最基本的通道处理器是：选择器通道和多路复用通道。对于选择器通道，每个通道可以服务于一系列的IO设备，这些IO设备都有自己相关的控制器。一般认为它们和快速的数据流有关，如硬盘驱动器。在数据传送过程中，通道专门服务于选定的设备，排斥所有其他设备，直到传送完成为止。在考虑慢速的设备时，多路复用通道更适合一些，因为通过位交错或数据流的多路复用，它能够支持几个设备同时进行传送，不会带来效率上的损失。

这种多个设备共享单个通道处理器的方式，导致了层次总线结构的产生，在这种结构中，主总线通过双端口通道处理器连接到IO总线。

9.8　单字符IO：屏幕和键盘例程

在编写通过菜单输入的程序时，需要用到来自键盘的单字符序列，这就涉及操作系统软件如何处理程序IO请求的问题。图9-23列出的代码片段，开始时会按照我们的预期运行。但在请求输入的循环进入第二次迭代后，会发生奇怪的现象：不需要任何键盘输入，菜单就会第二次打印出来——见图9-24。这并不是客户所要求的行为！

```
#include <stdio.h>

int main(void) {
    int answer;
do {
    printf("please enter a single letter: ");
    answer = getchar();
    putchar('\n');
    printf("%c\n",answer);
    } while (answer != 'C');

return 0;
}
```

图9-23　键盘输入

在该代码运行时，用户会发现程序阻塞在getchar()调用上，直到键入回车键后才会继续运行。对第一个字符"A"的处理是正确的。遗憾的是，在第二个字符"B"之前，有一个奇怪事件，即存在某种虚影按键。产生这种现象的原因，可以追溯到系统对键盘输入数据的处理方式。键盘的输入数据在传送到用户程序之前，临时性地存储在底层的缓冲区内。采用这种方案时，系统能够提供行编辑功能，并能捕获控制字符。程序正在等待时，输入^C（control-C）将会终止该进程。实际上，程序在用户键入回车键之前不会收到任何字符。然后整个缓冲区内的内容，包括回车符（CR），会一同发送到程序进行

```
please enter a single letter: A
A

please enter a single letter:  ←——— ?

please enter a single letter: B
B

please enter a single letter:  ←——— ?

please enter a single letter:
```

图9-24　输入缓冲问题的演示

处理。对于图9-23中的代码，这意味着“A”后面紧跟一个CR代码：0AH。通过使用调试器，或简单地将printf()语句括号中的%c换为%x，都可以进行核实，参见图9-25中的输出结果。这样我们就能够得出字符的数字代码，即使它是不可见字符。

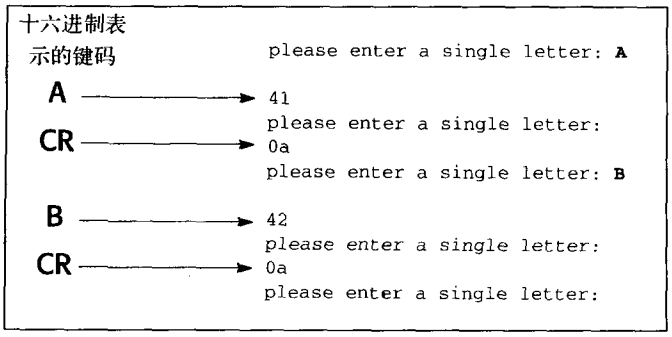


图9-25 显示隐藏的字符

在2.9节已经给出在Unix平台下该问题的解决方案。另一种解决方案（也适用于PC机）是，让用户程序在CR字符产生任何危害之前将其“消耗”掉，如图9-26所示。额外的getchar()只是为了从缓冲区内读入随后的CR，然后将它忽略。另一种方式是使用scanf()来替代getchar()。scanf()可以一次读入多个字符，也可以设置格式字符串，使之忽略字符。我们可以使用下面的语句替代两个getchar()：

```
scanf("%c%*c", &answer);
```

%*c读入单个字符，但并不赋予任何变量。这是解决CR字符的最快的方式。

```
#include <stdio.h>

int main(void) {
      int answer;
do {
      printf("please enter a single letter: ");
      answer = getchar();
      getchar( );
      printf("%c\n",answer);
      } while (answer != 'E');

return 0;
}
```

图9-26 键盘单字符输入

9.9 小结

- 三种基本的IO方法是：轮询、中断和DMA。
- 外部设备提供寄存器，供执行IO操作之用。有些操作系统禁止常规的应用程序代码访问外部设备的寄存器。这种情况下，我们可以使用设备驱动程序例程的系统调用来处理IO。
- IO设备内一般有状态寄存器、命令寄存器和数据寄存器。在采用存储器映射的系统内，这些寄存器看起来和内存中的存储单元是一样的。但是，如果系统为IO映射系统，那么除常规的HLL指令集以外，还必需提供专门的指令。
- 轮询让CPU等待事件，因而很浪费处理器资源。间歇式轮询要好于专注式（循环）轮询。
- 设备可以通过中断线请求CPU的注意。CPU会停下它当前的处理任务，跳转到专门指定的例程（ISR）来处理中断。如果有多个中断源，则必须提供验证和优先排序机制。在处理完中断后，CPU可以返回到被中断的代码继续执行。

- PC只有15条中断线（IRQ），在安装新硬件时会引发一些困难。共享中断线需要专门编写的ISR。
- 中断服务例程通过IVT（中断向量表，其中保存着所有已安装的ISR的起始地址）在内存中定位。
- 由于中断可能在任何时候到来，因此存在破坏数据的风险。这叫做临界数据问题。
- DMA能够在一定程度上将CPU从IO活动中解放出来，一般用于磁盘或LAN接口。DMA控制器硬件可以完成计算机内数据块的传送工作。

实习作业

我们推荐的实习作业包括尝试使用C函数库提供的输入输出例程。数据可以使用fputc()发送到COM1或COM2串行端口，用fgetc()从端口读入进来。使用终端仿真器，比如Hyperterminal，发送和接收文本消息。

练习

1. 在外设IO芯片内有哪三种寄存器？它们的用途是什么？它们只能使用汇编例程访问吗？
2. 给出每种IO方法在PC中的具体应用：轮询、中断、DMA。
3. 术语“阻塞”对程序员的含义是什么？在什么情况下putch()会阻塞？如何阻止PC的C语言程序在getchar()调用时阻塞？查看Microsoft Developer Studio在线帮助中有关kbhit()的内容。
4. 试着估计下面这些事件最低的轮询速率：
   汽车加速器踏板的位置以及汽车引擎进气管的压力，都服务于引擎控制部件；
   洗衣机前面板按钮；
   起居室调温装置；
   超市条形码扫描器（12位数字，每个4根条码）；
   用于交通灯控制的公路车辆检测器。
5. 对于下面这些应用，你会选择哪种IO技术：
   咖啡自动贩卖机前面板的键盘；
   LAN接口卡；
   火警烟雾检测器；
   信用卡读卡器；
   PC鼠标。
6. 计算机如何跟踪每天的时间，即使在关机状态？
7. 实时时钟信号有什么重要功能？RTC时钟信号来自何处？一般地，系统时钟信号的频率范围是多少？在Y2K问题中，RTC芯片是个什么角色？
8. 作为IO的一种手段，专注式标志轮询受到的主要批评是什么？如果轮询式IO这么差，为什么它还被如此广泛地采用？什么情况下需要使用专注式轮询？
9. 如何在HLL程序中使用中断？
10. CPU如何鉴别中断的来源？在中断发生后，中断的阈值会有什么变化？什么地方会用到rts和rte指令？
11. 列出ISR内需要执行的操作。
12. 使用中断时，什么情况下会出现临界区域？有什么编程方法能够解决这个问题？
13. DMA指什么？什么地方、什么时候PC机需要使用DMA？如果用DMA来定期更新大量的数据，程序员需要预先采取什么措施？
14. 如果不经意间输入一个制表键或空格字符，图9-23中的程序会发生什么问题？你能解释引起这个问题的原因吗？
15. 为IO通道提供数据缓冲区的目的是什么？使用缓冲有什么缺点吗？

课外读物

- Heuring和Jordan（2004），输入输出处理。
- Buchanan（1998），第5章：interfacing；第8章：interrupt action。
- Tanenbaum（2000），5.6.5节：using interrupts。

第10章　串行通信

设备间的数据交换常常使用1位宽度的通道来进行。本书后面介绍联网时遇到的许多技术问题可以仅通过两台机器间互连就可以学习和研究。这些问题包括接收器同步、错误检测和数据流量控制。本章还介绍各种不同类型的调制解调器，以及它们在沿模拟电话线路进行数据传输中的应用。

10.1　串行传输：数据、信号和时序

数字数据的传输已经从计算机与外设间的直接连接，发展到计算机之间复杂的国际性网络。但是，从简单的点对点链接中（常常指电子工业协会制订的**RS232**标准，见图10-1），我们依旧能够学到许多有用的东西。要注意，串行通信面临的问题不同于与计算机内部并行总线相关的问题。



图10-1　使用RS232电缆的点对点串行通信

尽管相比之下并行传送速度更快，但是为了降低电缆和连接器的成本，计算机之间大部分的数据传输工作都是采用串行的方式来完成。而且，并行总线能够工作的距离也存在物理上的限制。不管采用哪种技术，降低成本和更好地利用有限的资源都是必需的。串行通信时，数据一位一位地发送，尽管这种方案看起来似乎效率低下，但整个Internet都依赖于这样的通道进行通信，这些通道的传送速率一般在10 Mb/s和150 Mb/s之间。尽管RS232串行通信技术已经非常成熟，但是，如果将PC的COM1端口插到另一台计算机上，想迅速完成文件的传送，依旧会面临许多不可预见的困难，可能要几天的时间才能处理好，期间可能会遇到从未想像过的软件复杂性。和计算科学的许多其他领域一样，渐渐掌握实际的应用程序后，我们可能就会完全忘记那些复杂的硬件。几乎在所有的情况下，复杂的难题往往是因为对软件的不理解造成的。

在某种程度上，从学术的角度讲，所有的通信活动都会涉及三个方面，如表10-1所列。

表10-1　通信的基本构成

数据（data）	标准和编码方案，数量
时序（timing）	发送方和接收方之间的同步，频率和相位
信号传送（signaling）	错误处理，流量控制和路由

我们在分析点对点的串行传输时，实际上就已经开始在研究网络。尽管RS232很简单，但它依旧展现出许多我们将会在第14、15和16章遇到的网络问题。Internet和PC的COM端口都需要处理接收器与发送器之间的同步问题、偶然错误的检测问题，以及数据流的速率控制问题。但是，点对点串行通信方式与联网通信方式在其中一点上存在重大的差异，就是数据的路由。由于点对点串行通信只有一个地方可以去，因而不会涉及路由选择的问题！如果表10-1中列出的任一领域发生不匹配，数据传送最后都将失败。通信工程师的工作就是力图预料到这类失败，并采取有效的恢复措施。

10.2　数据的格式：编码技术

遗憾的是，使用串行通道传输数字型数据存在一个历史问题。最初设计串行通道的意图是，用来传输7位的ASCII码（见表2-4），但ASCII字符集中控制字符部分的不可打印字符可能会为设备驱动程序制造一些麻烦。例如，XOFF控制字符（^S）会暂停所有的传输，直到XON控制码（^Q）到

达，消除通道的阻塞为止。这是因为最初这些串行通道是用在控制台终端上，用户要求这些附加的编辑和控制功能。但在传输一幅图像文件时，当二进制模式0001_0011到达串行端口时，驱动程序可能被设置为识别XOFF控制，并会停止下所有后续的数据传输工作。因此，一般不在RS232串行链路上传输未加修饰的8位二进制数据。此外，大部分UART电路也有局限性，它们只能处理7位数据，加上一位错误检测奇偶位，构成8位。如果传输原始的8位值，那么从概率上讲，50%的字节的奇偶标志会不匹配。

有几种技巧可以用来避免这类问题：base64、uuencoding（Unix to Unix encoding）、Intel Hex和Motorola S记录（见图10-15）等格式的文件，均将8位二进制文件转换成7位ASCII码，以使它们能够安全地传输。多用途Internet邮件扩展（Multipurpose Internet Mail Extension，MIME）是为邮件附件设计的一种现代标准，它允许邮件的附件包括原始的二进制数据，可能是图像或声音文件，均可以安全地通过串行线路传输。

uuencoding和base64在开始部分相同，均从原始的二进制数据中取出3个字节，将它们转换成24位的整数，然后重新将它每6位一组划分成4组，每组所表示的整数从0到63。接下来，在将6位二进制数转换成可打印的ASCII时，两种编码方法出现分歧。uuencoding简单地加上32，将0～63平移到32～95。看看ASCII表（见表2-4）就会知道这些数字表示从“SP”到“－”的可打印字符。base64使用另一种转换方式，它将6位二进制数作为索引，在一个保存可打印ASCII代码的数组（见表10-2）中，查找对应的字符。

表10-2　为了使8位二进制数据能够串行传输而生成的uuencoding和base64格式

6位二进制数	uuencoding	base64
0	'SP'	'A'
.	.	.
.	.	.
25	'9'	'Z'
26	';'	'a'
.	.	.
.	.	.
50	'R'	'z'
57	'S'	'0'
.	.	.
.	.	.
61	'L'	'9'
62	'^'	'+'
63	'_'	'/'
	'='	

在采用uuencoding编码的文件中，至少每隔45个字符，换行符就会被插入，每行的开始会插入行的长度——表示为二进制数加上32。采用uuencoding编码的数据首行以单词begin开始，末行以单词end结束。文件名和Unix访问权限被放在begin后面，见图10-2。在发送电子邮件附件时，二进制格式的文件，图像、声音或者Word文档，在附加到文本消息后面之前，必须先完成base64转换。图10-3中是我的一封邮件，在邮件中，我以gzip压缩的PDF格式发送全书，因此，图10-3中的附件是缩简后的内容。

```
begin 664 test.bin
data here.....
$.............
end
```

图10-2　uuencoding编码的文件

Linux提供uuencode和uudecode工具，将文件转换成ASCII格式的编码文件，或将文件转换回来。类似地，我们也可以下载和安装叫做base64的小工具。这种转换过程会增加文件的大小，熟悉文件压缩的学生可能会感到惊奇。

```
From - Wed Feb 23 19:02:55 2005
X-Mozilla-Status: 0008
X-Mozilla-Status2: 00000000
Message-ID: <41ED1406.6070905@uwe.ac.uk>
Date: Tue, 18 Jan 2005 13:49:58 +0000
From: Rob Williams CEMS Staff <rob.williams@uwe.ac.uk>
Reply-To:  rob.williams@uwe.ac.uk
Organization: uwe, Bristol
User-Agent: Mozilla/5.0 (X11; U; SunOS sun4u; en-US; rv:1.0.1) Gecko/20020920 Netscape/7.0
X-Accept-Language: en-us, en
MIME-Version: 1.0
To: Simon Plumtree <simon.plumtree@pearsoned-ema.com>
Subject: Evidence of good intent
References: <004501c4fd4c$6e9951a0$bcd487d9@ariespcsystem> <41ECEB66.3060701@uwe.ac.uk>
Content-Type: multipart/mixed;
 boundary="------------040703090908080304070400"

This is a multi-part message in MIME format.
--------------040703090908080304070400
Content-Type: text/plain; charset=UTF-8; format=flowed
Content-Transfer-Encoding: 7bit

Hi Simon,

Here is the complete thing, minus a picture or two, and essential
checks on two chapters. Have a look and tell me if it is OK so far. The
good news here is the twin 64 bit Opticon server which has just been fired
up, this cuts the book compile time by a factor of 100. Now a complete
pdf book is assembled from the troff cuneiform and pic cave-wall
scratchings in about 5 secs!

Regards
Rob
=========================================
CRTS 20 year Reunion/Celebration May 13/14
=========================================
Dr Rob Williams :-)
CEMS, UWE
Bristol, BS16 1QY

--------------040703090908080304070400
Content-Type: application/gzip;
 name="book.pdf.gz"
Content-Transfer-Encoding: base64
Content-Disposition: inline;
 filename="book.pdf.gz"

H4sICAIQ7UEAA2Jvb2sucGRmAIz8Y5gtQbMtCre52rZt27Zt27Ztu3u1bdu2bdvWXe8+Z98P
+8/+U09mzKqozMiMihEjahaJvLAoDQMtExTJ6FVcARQrPj2+naElFDc3nbSJrZmzOT7bP4ki
naiFtbOJIz6dqLWBs4mwiZGdsQkvL5STs6OJgQ2Ue3aKsozTJity16Nal9LgVDBEv9z9kgNx
 .....
 .....
 .....
uJ3hwLzVb6GFZ9Yz0F4yG9/l7Dc84Pw2/RAWltcr+ev5PaM4Bm31S7qo+xPer39/+uO33//w
3R/f/uN//OJPv/n+//ruC79Z6Is//frHH3/6wobHF19/8adf/uGff/wi+d/f/uzP3v71p2//
+NN/++N3//yWfW8r9rd/9+/+6ne/efu/AT/hr8rGsS8A
--------------040703090908080304070400--
```

图10-3 MIME编码的电子邮件及base64编码的附件

10.3 时序同步：频率和相位

一般地，只有在学习外语时，收听者与说话者之间才会出现同步上的困难。接下来，识别单词以及辨别重要短语的问题，对学习者也是一种考验。我们常常会从比较低级的策略做起，将我们贫乏的词汇与听到的声音流进行匹配；而且，我们会转而采用复杂得多的会话的方式进行交流，在这种交流方式中，说话者的含意和意图成为更重要的部分。同时，我们还会发现，如果不断问问题，中断说话者的讲述，更多地控制会话，会使交谈变得更容易。

为了使接收方能够可靠地输入数据位的中心位置采样，肯定需要采取某种机制。接收方必须知道位的宽度，以及它们的起始点（参见图10-4），也可以这样说，接收方需要知道数据流的频率和相位。如果发送方和接收方的基本时钟脉冲能够互相校合（精确匹配），则能够保证完全同步。但这种方式并非一定可以实现。在图10-5中，发送方在时钟脉冲的上升沿写入1位数据，而接收方在半个时钟周期后，使用下降沿对线路采样，在数据位的中间。但是，在图示中，接收方时钟脉冲的频率比发送方低1%。这种情形下，特定的时间以后（本例中为75个数据位之后），当采样点跳过一个状态的变迁时，渐进性漂移造成的不同步就会产生。PC机中用来产生时钟频率的石英振荡器的精确度一般在百万分之20，或0.002%。如我们所见，如果传送方的时钟脉冲与接收方相差半个时钟周期，错误就会发生。传输速率为9600b/s（位/每秒），1位大约0.1毫秒宽，因此，在传输25 000个数据位的持续时间段（也就是2.6秒）之后，晶体漂移就会导致错误。为了避免发生这种情况时所引发的问题，接收方必须定期地在位级别重新同步。由于其他一些原因，字符、包和消息的起始点也必须定期同步。

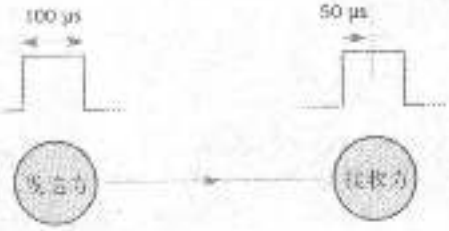


图10-4　接收方为位的中间采样

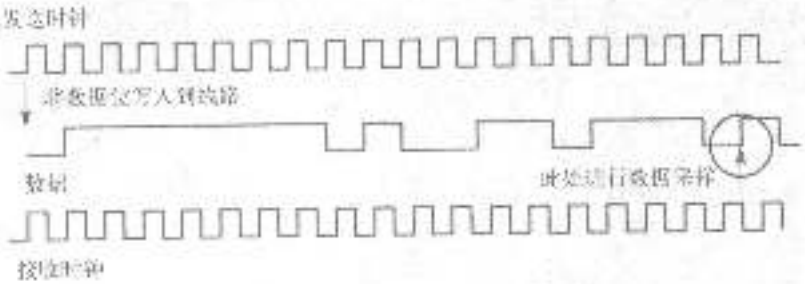


图10-5　接收设备的同步问题

每秒钟传输的位数一般记做“比特率”（b/s），旧差的术语“波特率”（baud）依们在使用。严格来讲，波特率是指基本时钟频率。对于简单的传输，波特率和比特率是同步的，后面我们会遇到将几个位压入单个时钟周期内的情况，所以比特率有可能大于相应的波特率。

为了双方能够达成一种相同的速率，或称位宽度，我们可能必须在初始化菜单中手动地设定“线路速度”。或者采用另一种在登录时进行的半自动过程，它要求用户敲几次回车键，从而使接收方能够猜测正确的线路速度——因为接收方知道如果不能将接收到的位模式识别为回车字符，则应该尝试不同的线路速度。还有一些系统在发送数据时，专门将时钟信号也一同发送出去，这种方式很类似于整点时的GMT广播报时信号，全美国的人都可以根据它来校正时钟。如果接收方和发送方使用公共的系统时钟，我们称它们工作在**同步模式**（Synchronous Mode），而如果它们有各自独立的时钟（相同的频率），那么它们就一定工作在所谓的**异步模式**（Asynchronous Mode）。在使用PC的COM1和COM2时，后者十分普遍。

由于现在的晶体谐振环振荡器能够做到十分精确，因此，提供良好且稳定的频率源不再是一件困难的事。但接收方依旧需要确定每个输入位的中点，以便更可靠地读取数据，在异步模式下，发送方不将时钟信号和数据一同发送。取而代之，它在每个传输字节之前插入一个额外时钟脉冲，我们称之为起始位。每个ASCII字符都作为一次单独的传送来处理，都附加有起始位、结束位和奇偶校验位（见图10-20），线路速度在会话开始前手动设置，接收方必须检测起始位的下降沿以确定相位信息。重要的是，为了使这种方案能够工作，字符之间必须有一段时间的“寂静”，这由逻辑的停止位来完成。如果愿意，可以将起始位看做是发送方给接收方的同步脉冲。编程控制微控制器，使它

作为串行通信设备，在每个起始位到来时测量它的宽度，在技术上是可行的。采用这种方式，接收方能够从发送方获得频率和相位信息，以一种准同步的方式工作。我们可以将8位、异步的传输方法看做是8B/10B编码技术，采用这种方法时，8位的数据在传输时被编进10位的字中。

在同步模式下，字符快速地连续传送，没有起始位和结束位。为了给接收方机会，对数据进行同步，消息块需要用接收方的硬件能够检测出来的特殊同步字符作为前导。在没有数据可供传送时，发送方也会连续地发送这些同步字符。同步模式的传输可以使用智能调制方案（依赖于额外的硬件），在同一对线路上传送数据和时钟信息。以太网就使用这种方式，我们称之为曼彻斯特编码，但是，对于短距离的传输，使用单独的导线传送时钟和数据更容易也更廉价。输入的时钟信号可以向接收方提供精确的位频率和相位信息，接收方因而也就知道什么时候可以安全地读取输入的位。

还有一种同步方法，它主要针对快速的串行传输，用于非字符、面向位的数据。这项技术允许更快的数据传输速率，是以太网的先驱。它的全称为“高级数据链路控制”（High-level Data Link Control，HDLC），它也使用特殊的位模式（01111110）来使接收方能够锁定消息的开始。如果这个位序列出现在数据流中，则插入0形成011111010。插入的0在接收端会被删除。

在描述这些方法时要注意一个小问题，术语“异步”的含义在计算科学的其他领域并不一致。例如，“异步总线”是指控制数据传输的一种互锁握手。对于Centronics并行端口和SCSI总线，异步也是这种含义。哪一种正确呢？“异步”仅仅是指“没有共享的时钟”吗？或者暗示更为复杂的“签收式的”移交吗？我认为后者更合理一些。两种模式的真正区别在于，接收方是否能够（通过信号线）向发送方反馈信息，告诉发送方传送已经成功完成，它现在可以结束当前的周期，处理下面数据。因而，异步链路允许通信设备以不同的处理速率运行，而不会丢失数据，也不需要在会话中降低较快一方的时钟频率。

在软件层面也存在类似的同步问题。收听收音机时，在我们喜爱的节目开始时，我们如何能够判别出来，进而更加专心地听呢？我们能够理解收音机播出的每个单词，但它们是否属于我们喜爱的喜剧评论节目前面的一场政治辩论呢？或者我们是否错过了开始，现在已经进入分组讨论阶段？为了帮助收听者解决这些困难，电台需要在切换节目时传送清晰的标志。查看2.8节中的ASCII表，现在我们能够理解为什么有一些奇怪的代码：SYN、SOH、STX、ETX、EOT和ENQ。它们是信号代码，用来在数据帧和消息级别对接收方进行同步。IBM在早期的BiSync协议中使用它们，这个协议将8位字符代码和一位时钟信号一同传送。这是一个同步协议，发送方和接收方共享同一位时钟，因而在位级别没有同步困难。这些代码在字节、帧和消息级别处理频率/相位问题。具体地，这些代码分别表示：

SYN：特殊的标志位，用来帮助接收方进行字节级别的同步。仅在通道以同步模式运行时使用。

SOH：消息报头的开始。

STX：消息正文的开始。

ETX：消息正文的结束。消息可以拆分成多个文本块。

EOT：消息传输的结束。

发送方需要在输出数据流的恰当位置插入这些代码，以使接收方能够知道应该如何处理输入的数据。

我们已经提到过，异步的链路在没有数据发送时是平静的。但BiSync通信并非如此，BiSync协议在数据传输之间会插入许多SYN字符，使线路保持忙碌。这是为了在任何时候都维护与接收方的同步。它允许沿数据线发送时钟信号，在接收端由专门的硬件电路复原，因而可以省去一条单独的线路。

10.4 数据编码和错误控制：奇偶校验、检验和、汉明码和CRC

如果传输的线路受到电子干扰，就有可能发生错误。荧光灯点火、大型电动机开启和关闭以及振荡恒温器，是这类噪音的典型来源，见图10-6。可恶的感应脉冲尖波会向四周放射能量，附近的线路会像电视的天线那样接收这种能量。为了提高速度和集成密度，计算机系统内使用的操作电压已经被

降低，噪音拾取造成的影响加大。计算机的机箱就是为了排除这类电子干扰而造的。甚至主电力连接也得仔细地过滤，以保护计算机的电路。来到外面的世界之后，错误发生的机率可能会成百万倍地增长。这听起来很恐怖，但实际上计算机内偶发性错误的可能性非常低，大约为10^{-18}。即使以10^9b/s的速度处理数据，也就是奔腾芯片的处理速度，也要约10^9秒才有可能发生一次错误，大约为30年，或者是玩一生的《毁灭战士》（Doom）或《古墓丽影》（Tomb Raider）游戏才会遇到一次。

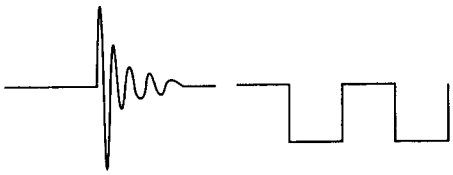


图10-6　电子噪声脉冲

尽管现代通信信道越来越可靠，尤其是光纤类通道，但是，CD-ROM和DVD的引入，使错误检测和纠正技术的应用获得了新生。要是没有错误检测/纠正技术，表面的损坏很快会使它们成为一堆废物。有几种方法值得研究（见表10-3）。所有的错误检测方法都有成本，在传输不重要的数据时，需要注意。明显的例子是接收方将输入的数据回送给发送方。这样，使用的带宽是原来带宽的两倍，要注意由此带来的花费！另外，也可以在最初的数据之后再第二次传送数据的拷贝，这样接收方就可以进行对比。前一种（回送）方法同时提供流量控制机制，我们可能会用到。我们将在下一节中再论述这个主题。

表10-3　错误检测和纠正技术

奇偶校验位	易于使用，不太安全
块检验和	易于使用，不能提供太多的帮助
多项式除法	计算较复杂，安全性好

所有的错误捕获系统都依赖于冗余（不是必需的）数据。数据常常在传输之前进行编码。在第2章中，我们已经遇到过ASCII和Unicode字符代码集。如果某些位模式根本不属于选定的代码集，那么显然它是错误的，这有助于错误的检测。接收方在检测到不合法的代码时，或者针对具体情况做一些处理工作，或者简单地将被破坏的数据丢弃。

ASCII编码字符的串行传输大多采用**奇偶校验**（parity）作为错误检测的方法。我们要以下面这种方式看待单个的奇偶位：它将代码集大小翻倍，但让半数可能的二进制模式（binary pattern）不合法。合法编码数字之间的“距离”称为Hamming距离。因此，含单个奇偶位的数字的Hamming距离为2——改变单个位就会使最终的值无效（位于有效数字之间的“鸿沟”中），而改变两个位必定会得到邻近的有效数字。这是错误检测的根本所在：在每个有效代码周围挖一道宽的Hamming沟（距离），错误将会落入其中。

表10-3中列出的三项技术在数据存储应用领域以及串行数据传输中依旧广泛使用。这类技术的共性是：发送方/写入方（见图10-7）取一些数据，计算出某种特征值（签名），然后将数据和签名一同发送出去。在消息到达时，接收方/读取方同样使用数据计算一些东西，然后将新的签名与刚接收到的进行比较。如果签名不吻合，则有错误发生。奇偶校验位技术适用于任何长度的二进制数据。每个字都会附加额外的一位作为签名，将整个字的奇偶状态调整为奇或偶。奇字含有奇数个1，偶字含有偶数个1。计算奇偶的工作常常使用XOR功能来完成，如图10-7所示。

所有的UART设备都提供在传输前插入一个奇偶校验位、在接收时进行奇偶校验的硬件电路（参见10.6节）。但是，如图10-8所示，单个奇偶位只能检测出单个位发生的错误。任何两个位出错都会互相抵消，检测不出——在噪音较大的环境中，电子尖波脉冲极有可能破坏字中的多个位，因此，在这种情况下，这种技术不太安全。是否安全完全依赖于传输的速度（位的宽度）和噪音脉冲的频繁程度而定。

检测到错误后并不一定采取什么措施。也许忽略它并不会产生什么问题。比如，如果受到破坏的数据不过是实时视频流中构成某一帧的上百万像素中的某个像素。这种情况下，由于15毫秒左右下一帧数据就会到来，因而等待下一帧几乎是最好的处理方法。或者，如果协商建立了某种重传方案，可以向发送方请求全新的副本。理想的解决方案应该是接收方将任务自己承担起来，找出被破坏的位。这种方式称为“错误检测和纠正”，或**前向错误控制**（Forward Error Control，FEC）。如我们在图10-8所见，单个奇偶位肯定不能提供太多的安全性，或找出字中错误发生的精确位置。它只能表明在一些地方发生了奇数个错误。

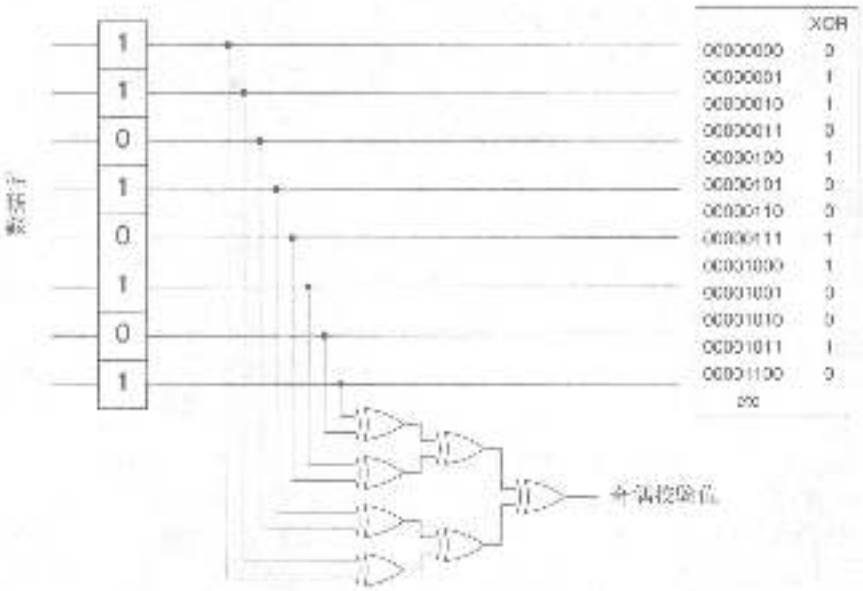


图10-7 使用XOR门电路计算奇偶位

待发送的数据	计算奇偶位并附加到数据中，形成偶数个1	传输	计算奇偶校验位并比较	
0110 0111	0110 0111 1	无错误	0110 0111 1	没有检测到错误
0111 0110	0111 0110 1	0111 1110 1 有一个错误	0111 1110 0	检测到错误
0111 0100	0111 0100 0	0111 1101 0 多个错误	0110 0101 0	没有检测到错误

图10-8 使用单个奇偶校验位检测错误

通过添加多个奇偶位，我们可以将错误的位置鉴定出来，我们将这种编码称为汉明码。图10-9给出如何在只加入3个附加奇偶位的情况下，对半字节（4个二进制位）数据进行错误检测和纠正。每个奇偶位负责三个数据位。这种方案是一种实用的廉价方案。如果将4个位组成的半字节发送两次，接收方也能验证它们的合法有效性，但在两个版本不符时，没有办法对位的数值进行纠正。如果要在发生错误时能够纠正，那么，我们必须发送三份拷贝（12位），才能在发生错误时，通过表决确定正确的值。

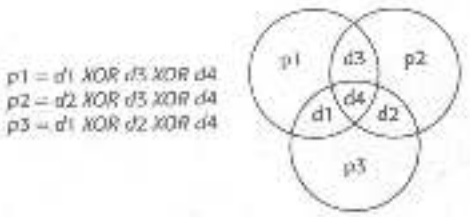


图10-9 三奇偶位方案

但是，图10-9说明的方案中，如果d1或d2或d3中的任一个发生错误，那么会有两个奇偶位受到影响。从而能够找出哪个地方发生错误，d4发生错误时，会影响到所有三个奇偶位，因而能够追踪出来。类似地，如果单个奇偶位发生错误，由于缺乏来自于其他奇偶位的验证，所以也能够识别出来。在找出受到破坏的位之后，只需将其反转即可！这种方案可以正式化，并扩展到更长的数据字。

但是，奇偶位的恰当摆位是实现优雅方案的关键。

要了解如何计算汉明码，重要的是要理解奇偶位插入到数据字中的方法。图10-10给出4位数据中奇偶位的安排，但是，很显然我们感兴趣的是更长的字，此处的技巧是将所有2的幂的位置分配给奇偶位，1、2、4、8、16，依次类推。其他位置用于存储数据，现在我们检查每个奇偶位分别对哪些数值做出贡献：

8	7	6	5	4	3	2	1
p4				p2		p0	p1
P	d4	d3	d2	P	d1	P	P

图10-10　将奇偶位分配给更长的字

位1表示所有位置为奇数的位：1、3、5、7等。

位2表示2、3、6、7等

位4表示4、5、6、7等

位8表示8、9、10、11等

位16表示16、17、18等

因此，第3、5、7、9等位由p1（位置1）校验，第3、6、7、10、11等位由p2（位置2）校验，而第5、6、7、9、10等位由p3（位置4）校验。

我们可以将数据位的数量和对应的完成单个位错误纠正所需的奇偶校验位的数量之间的关系列成表格。图10-11给出这种方法以及成对的数据位数量和奇偶校验位数量。

奇偶校验位的数量	p	1	2	3	4	5	6	7	8
	2^p	2	4	8	16	32	64	128	256
数据位的数量	d	0	1	4	11	26	57	120	247

$d = 2^p - (p + 1)$　　4d−3p

图10-11　数据位和完成单个位错误纠正的奇偶校验位之间的数量关系

每个奇偶位的值都是使用指定的数据位计算得出，从图10-12我们可以看出，有可能开发一种选择矩阵，使用矩阵操作来完成这项运算，尽管它看起来十分复杂，但它对于计算机执行来说十分简单，而且能够产生十分优雅的方法。

考虑图10-12中的数据项1 0 1 1，我们需要对它进行编码，插入正确的奇偶值：1 0 1 p3 1 p2 p1，通过使用选择器运算符，我们可以计算出三个奇偶位的值（初始值为0）。

矩阵中着重显示出来的位是奇偶位选择器。奇偶值插入到数据字中指定的位置之后，重复矩阵乘法，结果将为（0 0 0）　　表示数据正确。

下面我们来看看，当人为地将一个错误引入到数据字中，而后重新运行这项运算时，会发生什么。计算的结果称做检验子（syndrome）。下面我们将数据字中第5位上的数据置为错误值0（见图10-13）。

```
[1010100]  ×  | 1 1 1 |   =[001]
 7654321      | 1 1 0 |
 dddpdpp      | 1 0 1 |  结果为p3=0，
              | 1 0 0 |  p2=0，p1=1，应
              | 0 1 1 |  该发送的数据是[1
              | 0 1 0 |  0 1 0 1 0 1]
              | 0 0 1 |
```

图10-12　计算4d-2p的检验子（发送方）

```
错误在这里
1000101   ×   1 1 1   = 101
              1 1 0
              1 0 1
                0 0
              0 1 1
              0   0
              0 0
```

图10-13　在有错误的情况下计算4d-3p的检验子（接收方）

我们看到，检验子是5（当然以二进制表示），这真是令人惊奇。它不但检测出这个错误，而且还指出错误在数据字中的位置。我们所需要做的就是找出指定的位，将它取反，就纠正了这个传输错误。这是一个数学技巧吗？或许是这样。但它的确给我留下很深的印象！遗憾的是，这个魔术不能处理双位错误，因此，现在常见的做法是，提供一个总体奇偶校验位p0，负责帮助检测两个位出

错的情况。这种情况下，检验子不是0，而是比较混乱的值，如表10-4所示。这种方案称为“双位错误检测，单个错误纠错”。

表10-4 使用多个奇偶校验的单个位错误纠正，双位错误检测

	没有错误	单个位错误	双位错误
p0	一致	错误	一致
检验子	0	非零一错误	非零一混乱

对于奔腾处理器使用的64位内存字，需要引入7个奇偶位：p0～p6，分别插入到第0、1、2、4、8、16、32位。这样做能够达成双重检测级别的保护，对于某些系统，比如航空交通管理计算机，这是一种必需的措施。

如果尚未学习过上面用到的矩阵代数，可以向数学导师抱怨。计算机十分擅长于矩阵代数的计算，而且矩阵代数在图形程序设计中也十分有用。

另一种经常用来检测传输错误的方法是**块检验和**（block checksum）。首先需要将数据切分成数据块，这样就能够计算其检验和，并在传输前将检验和附加上去。生成检验和的方法，是简单地将一个数据块内的所有数据相加，截取产生的和。这个和数常常被限制为单字节值，仅取小于256的值。可以将这个过程认为是将数据累加到8位的累加器，忽略溢出，或执行取模（256）除法。不管采用哪种方式，最高符号位都会丢弃！之后，将检验和取反，附加在数据之后发送出去。接收方的任务是将不断接收到的所有数据项累加起来，包括最后的检验和。如果没有错误发生，产生的和应该是0。如果它不是0，则一些数据可能遭到破坏，数据块必须重传。检验和没有纠错功能。最广为人知的应用检验和的例子是Motorola S记录格式，或Intel Hex记录格式。它们是在微处理器开发的早期设计的，那时纸带阅读器还是标准的设备，但不太可靠。在试图读入长的纸带时，纸张的细微破损都会引发错误。此时，可以停下阅读器，将纸带回退一小段长度，重新读取有问题的数据块，这样就能够避免一些问题的产生。

Motorola S记录和Intel Hex文件中存储的数据都是ASCII格式。在传输之前，每个字节都得拆分成两个半字节，然后将其转换成十六进制数字（0～F）的ASCII表示。这也就说明了为什么图10-14中单字节检验和会以两个字节出现。

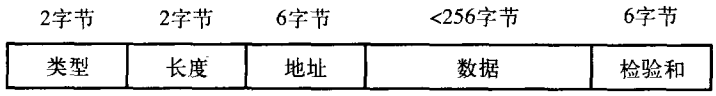


图10-14 带检验和的Motorola S记录格式

图10-15是Motorola S记录文件的一个片段，为了能够更方便地识别字段，它已经被人为地隔开。现在，我们应该验证检验和的算法。选择一个较短的数据记录（S2类型）——倒数第二个比较方便，将长度、地址和数据字段加起来（用十六进制）。一些计算器提供十六进制模式，如果没有的话就去借一个：

```
S0 03 0000 FC
S2 24 010400 46FC26002E7C000808006100005E6100008261000033C46FC270023FC00010678 6B
S2 24 010420 000C011023FC00010678000C011423FC00010678000C011823FC00010678000C 6D
S2 24 010440 011C610003A4303C271053406600FFFC46FC21006100057A4E4B000000004E75 3B
. . .
S2 24 012200 0968584F4878004C4EB900010928584F206EFFFC524810BC0004602248790001 7D
S2 24 012220 21CA4EB900010968584F487800484EB900010928584F206EFFFC524842104E5E 84
S2 08 012240 4E750000 D1
S8 04 000000 FB
```

图10-15 一段采用Motorola S记录格式的文件

08 + 01 + 22 + 40 +4B + 75 + 00 + 00 = 12E

将溢出的1去掉，得到2E（0010 1110）。

逐位取反：1101 0001（D1）——检验和！

一种更复杂的计算检验和的方法是**Bose-Chaudhuri-Hocquenghem**（BCH）方法。它能够检测出多个位发生错误的情况，同时还能够确定出它们的位置。我们应该珍视这类技术，不只是因为它曾被成功地用来保护汽车停车票内的时间和日期信息。在自动售票设备首次使用微处理器控制时（见图10-16），打孔代码在实际操作过程中太容易受到损坏，造成拒票率不可接受的高。通过引入BCH错误检测和纠错码，只通过票的大部分就能够读取出有效的数字。勇敢的销售人员，为演示BCH编码的有效性，将票撕成两半，随便地将两个半片输入读卡器内，依旧完成了有效的交易。这是让数学家骄傲的事！



图10-16　使用BCH纠错编码的停车票

另一种常用的错误检测方法是**循环冗余校验**（Cyclic Redundancy Check，CRC）。这是一种强大的错误检测方法，但不能确定发生错误的位的具体位置。和其他错误控制思想类似，CRC方法计算一个检验和，但这次是使用算术除法。根据数字与适当的多项式之间的关系，专门选取一个数字，用这个数字去除要发送出去的数据。本质上，它是一种将数据位进行分组，并与一个冗余位关联起来的方式。除法使用模2运算（禁止进位和借位，实际上就相当于XOR运算！）执行。图10-17使用数据值11001除以101，得出一个余数。之后这个余数作为签名，跟在数据的后面发送出去。



图10-17　发送方与接收方对数据项11001的CRC的计算

我们可以将除数中1的位置看做是实现了和前面方法中的汉明运算符选到的相同的选择过程。两种方法都将二进制掩码应用到数据上来选择位元组。CRC进行重复的移位，对全长的数据应用相同的短掩码，而汉明方法采用几个全宽的掩码。数据和余数同时传送。商（除法的结果）被丢弃。接收方，如我们所预期，重新计算除法，并对余数进行对比，看它们是否相同。如果不同，就只有请求重新传送。这种方法的效果令人印象深刻。生成16位余数的CRC可以检测下面这些错误：

1）所有的16位或小于16位的突发错误。

2）所有奇数位的错误。

3）99.9985%任意长度的所有突发错误。

用硬件来计算CRC相对于使用长除法的软件方式要容易些。图10-18中给出了针对16位数据字计算CRC的移位寄存器及XOR门反馈电路的排列。

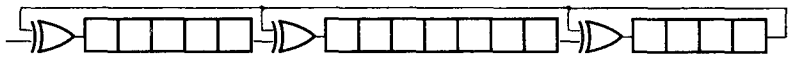


图10-18 使用移位寄存器和XOR门生成CRC

10.5 流量控制：硬件和软件方法

对于控制数据的传输来说，协议是必需的，尤其是复杂的网络或大型的计算机系统之间的数据传输。一般情况下，**流量控制**（flow control）是必需的，它可以防止由于接收方不能足够快地处理输入的数据而造成的溢出错误。如果前一批数据接收正常，但未能及时处理，后续到达的数据就会覆盖接收缓冲区内的较早数据。当发送方提供数据的速度快于接收方的处理速度时，就一定要采取流量控制措施，以防止接收方受到数据溢出的困扰。

要注意，这与波特率不兼容没有任何关系。最初，接收方能够正常地处理数据，没有错误发生。仅当输入缓冲区完全被填满，后面输入的数据开始覆盖早期到达的数据时，溢出错误才会出现。正如我们所见，输入缓冲区的操作模式为浴盆型溢出模式。只要输入数据到达的速率低于浴盆流出的速率，就不会发生溢出。但是，如果大批的数据突然到达时，就极有可能将缓冲区填满，然后覆盖前面的数据项——除非接收方能够请求发送方停止发送数据，直到缓冲区被清空为止。这种机制就叫做流量控制，它几乎对于所有的通信链接都是必不可少的。

表10-5中列出的技术一般并不属于应用程序员需要关心的内容，而是在比较低级别的操作系统设备驱动程序和管理例程中实现。在10.3节中，我们已经遇到过使用“回声”来检测传输错误的例子。这种技术也可以用来完成流量控制。当我们向编辑器键入字符时，字符出现在屏幕上的文本缓冲窗口内。这就是由接收方（计算机）发送给我们的“回声”。如果我们输入过快，超过系统的处理速度（就我来说不太可能），计算机会停止回显，屏幕就会不再显示我们正在输入的内容。停止响应后，大部分用户会停止打字，给系统一些时间来清空输入缓冲区，重新跟上用户的输入。之后，系统会重新开始回显用户的击键，我们就可以继续打字。我们敲击的键，在系统没有回显时存储在键盘缓冲区内，并不会丢失。如果长期的平均数据输入速度属于接收方能够控制的范围，那么提供更大的存储缓冲区会有帮助。问题在于传输所固有的突发特性，一段段的高发期与长时间平静的间隔交织在一起。缓冲当然是对付这种情形的标准解决方案。硬件控制和下一节中将介绍的RS232串行链接相关。软件控制不需要任何额外的硬件就能够进行。流量控制可以用软件实现，仅仅偶尔通过数据通道发送一些特殊的代码值就可以做到。它们的工作方式和人类的电话会话极为相似。一般参与者依次讲话——尽管同步的交流是可以的！如果在会话中错过某些信息，我们可以说“对不起，请再说一遍”，希望重新传送一次。如果对方讲话太快，我们可以说“等一下”。我们拥有很多单词和短语，能够作为会话的控制代码。计算机在这方面相对有限：Control-S（^S）对应停止，Control-Q(^Q)对应继续。这些代码常被称为Xon和Xoff。如果有机会使用Unix，可以通过键盘试试它们。新手常常会不小心按下^S，将屏幕冻结，然后一筹莫展，不知道到底发生了什么事。

表10-5 流量控制技术

回声
硬件控制线路：RTS/CTS
软件控制代码：^S/^Q
基于帧的握手代码：ACK/NAK

程序员必须理解图10-19中阐述的所有这些基本的流量控制方法，有时也许还会用到。

在快速的同步线路和网络上，常常采用一种更复杂的流量控制方法，这种方法允许发送方即使在没有接收到确认信号的情况下，在一定的时间内，也继续发送数据。这样做的意图是，保持通道处于完全占用状态，而不无端地阻塞发送方。我们将这种控制技术称做**窗口流量控制**（Window Flow Control）。发送方维护一个缓冲区来保存已发送但未接收到确认的数据项。这样，在接收到滞后的重传请求或发生超时的情况下，能够立即重新传送。只有得到接收方的充分确认之后，才将这些数据项从发送缓冲区内删除。

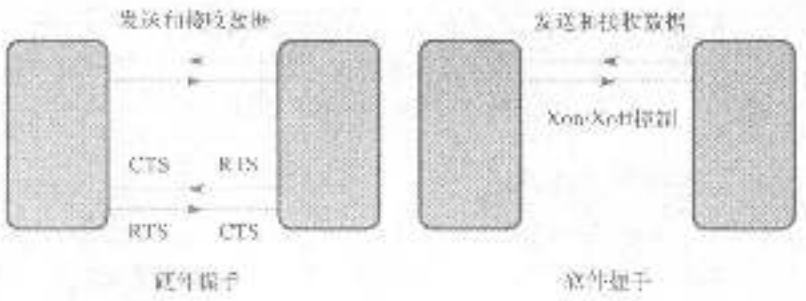


图10-19　RS232流量控制技术

如表10-6所列，任何超出简单点到点连接的方案，还需要传送路由信息（源地址和目的地址）。较长的消息一般会切成一段段的子消息，高带宽的通道常常共享，以降低成本，在解多路复用以及重新组装时，需要区分来自不同来源的数据。我们会逐渐认识到信号发送协议可能会变得极为复杂。

表10-6　串行通信的数据路由方法

串行链路	专用路径，不需要寻址
LAN	广播传输，由接收方完成识别
WAN	通过转发地址信息来选择路由，可以动态改变

信号发送这一主题还包括跨大型网络传输数据时对路由数据的需求。在点对点串行链接中，常常不需要选择路由，如果发生的话，一般是在处理设备控制问题。例如，我们可能需要发送特殊的转义码到交换机，来选择特定的出口端口。在表10-6中，我们罗列出路由的三种方式。在随后的两章我们还会论述这一主题。

10.6　16550 UART，RS232

最底层的用于连接设备的通用标准是我们称之为RS232或V24的串行数据链路。它使用25针或9针D形插头和插座，但实际互相连接的线路常常只有三条。在传送数据时，二进制的1被表示为约−9V，+9V代表二进制的0，图10-20勾画出RS232传输1（字符）的ASCII码（31H或00110001B）时的电压图。

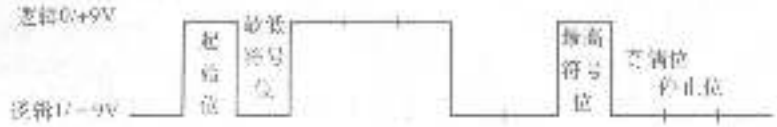


图10-20　表示ASCII "1"的RS232电压

RS232 D型接口的针脚有两种定位方式，分别针对DCE（数据通信设备，现在可以理解为计算机）、DTE（数据终端设备，现在可以理解为终端或打印机）。由于最初的25线D型连接头往往只使用其中的三个针脚（为了节省成本，可以购买只安装三个针脚的25路D型插头），因此，一种新的9路D型标准被引入进来（见图10-21）。



图10-21　RS232 9路D型针脚的功能（COM1和COM2）

在过去，CTS控制输入不过是一个硬件门电路，它不受程序员的影响。要使计算机能够传输数据，除非安装合适的曲别针，将CTS强行置为正确的状态。使用前述的曲别针将RTS输出直接连接到CTS输入时，有可能破坏常规的硬件流量控制系统！在Windows上运行超级终端（Hyperterminal）程序时，会弹出串行端口设置画面（见图10-22），它允许我们重置一些基本的参数。



图10-22 在Hyperterminal中设置COM1端口参数

线路速度（位宽度）、奇偶模式、流量控制方法和其他几项参数都在我们控制之下。现在，试着将一台PC的COM1端口连到另一台PC的COM1端口中。要在两台计算机之间传输数据，我们需要正确的交叉连线。而将终端设备连接到计算机上时，直接连线就可以，但在连接两台计算机时，则需要交叉连线。有时我们也将其称做“空调制解调器”。D型插头有两种：插头和插座。在购买这类导线之前，先要仔细看看COM2插座。

图10-23给出的程序展示出使用为访问文件而设计的函数如何直接访问PC机的COM端口。有两个端口的单个PC也可以用来运行这个程序，我们可以向COM1写入数据，从COM2读出。要注意，stdio.h中将常量EOF定义为 -1。在试验这个程序之前，最好在两台机器上都运行Hyperterminal（超级终端），或类似的终端仿真程序，以检测连接是否正常，数据是否能够可靠地发送。尽管程序中将数据项作为32位int来处理，但在通过COM2传输时，它将会降为低字节，高字节不会被传递。在由键盘输入数据时，Control-Z可以提供EOF，供接收方检测。

```
/* Transmitter.c */
#include <stdio.h>

int main(void)
{
FILE *dp;
int c;
      if ((dp = fopen("COM2", "w")) == NULL)
      {
          printf("Fail to open COM port\n");
          return 1;
      }
       while ((c=getch()) != EOF)
       {
        fputc( c, dp);
        fflush(dp);
       }
       return 0;
}
=============================================================

/* Receiver.c */
#include <stdio.h>

int main(void)
{
FILE *dp;
int c;
      if ((dp = fopen("COM2", "r")) == NULL)
      {
          printf("Fail to open COM port\n");
          return 1;
      }
      while ((c= fgetc(dp)) != EOF)
      {
       putch( c);
      }
       return 0;
}
```

图10-23 在PC机上通过RS232链接交换信息

任何新型的通信互连在开始时都会让人受到许多挫折，注意：每一步都要小心且合理。在接触新的硬件时，惟一获得所需信息的方式，常常是勇敢地打开硬件的电路图，找到描述串行接口的部分。这需要钢铁般坚强的神经或好朋友的帮忙，如果使用试错法来跟踪问题，则不要忘记做笔记并将测试配置写下来，以备将来之用。

用来将内部的并行数据转换成外部RS232串行格式的接口芯片，叫做通用异步接收和发送设备(Universal Asynchronous Receive and Transmit device，**UART**)。它和其他任何设备一样，连接在总线上。所有为调制解调器或终端连接提供串行端口的计算机都有UART。

UART是数据协议转换器，它将内部的并行总线与外部的串行线路连接在一起(如图10-24所示)。因此，它也常被称为SIPO/PISO，意思是串行入并行出(Serial In，Parallel Out)或并行入串行出(Parallel In，Serial Out)。UART还可以执行一些低级的错误捕获任务，一般地，它能够检测出输入字符的长度错误或奇偶错误，以及到达太快，改写了前面数据项的情况。

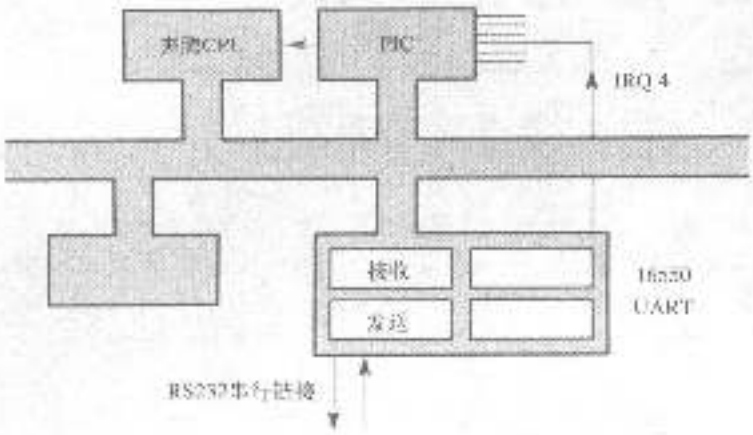


图10-24　安装UART串行线路接口

早期PC使用的8025 UART，由于硬件上的缺陷，程序员对它的了解并不多。它的替代产品是Intel 8251，它在正确地响应CTS线路上也存在一些问题！有意思的是，国家半导体提供的性能更强的16550A UART对发送和接收通道都提供16字节的FIFO缓冲区。这样就降低了对CPU时间的限制，因为在CPU读取数据并将数据安全地存储在主存之前，它允许至多接收16个字符。在PC机上，当数据接收需要服务时，会生成IRQ4中断，警示CPU。通过16字节的输入缓冲区，中断频率最高可以降低16倍，这一点十分重要，因为如果CPU未能及时做出响应，后续的输入数据会溢出，会破坏接收缓冲区内已有的数据。在发送器拥有16字节的缓冲区的情况下，甚至值得考虑是否使用DMA控制器完全消除CPU的直接参与。

图10-25给出的程序，通过两台PC机的COM2串行端口连接，将一台PC机上的文件传送到另一台PC机中。但是，如果使用Windows操作系统，在将这个COM端口的例子扩展到更有意义的应用时，会产生一个问题，图10-25列出的代码在fgetc()函数调用上阻塞。这意味着如果在端口上没有输入数据，程序就会停下来等待数据到达。如果还有其他端口需要检查，或屏幕需要更新，这可能就不是一种我们期望的表现。Windows中有一个常用的状态检查函数kbhit()，它允许我们在提交阻塞读操作之前检查键盘设备，但对于COM端口却没有这类函数。一种可行的办法是使用SetCommTimeouts()函数，这个函数可以将阻塞的时间限制到1毫秒。这意味着，在没有数据的情况下，也可以使用ReadFile()从COM获取字符，而不会承担太多的阻塞延迟。图10-26列出一些有用的函数。更复杂的方式是使用并行运行的多线程机制，来处理多重操作和阻塞问题。

```
/* Filetrans.c */
#include <stdio.h>
#include <conio.h>
#define CNTRLZ 0x1A

int main(void)
{
FILE * fp
FILE * dp;
int c;
if ((fp = fopen("C:\\TEMP\\text.dat", "rt")) == NULL)
{
        printf("fail to open data file\n");
        return 1;
}
if ((dp = fopen("COM2", "wt")) == NULL)
{
        printf("fail to open COM port\n");
        return 1;
}
while ((c = fgetc(fp )) != EOF)
{
 fputc( c, dp);
}
fputc(CNTRLZ, dp);
fflush(dp);}
fclose(fp);
return 0;
}


/* Filereceive.c */
#include <stdio.h>
#include <conio.h>
#define CNTRLZ 0x1A

int main(void) {
FILE *fp;
FILE *dp;
int c;
       if ((fp = fopen("C:\\TEMP\\text.dat", "w")) == NULL)
       {
           printf("fail to open data file\n");
           return 1;
       }
       if ((dp = fopen("COM2", "r")) == NULL)
       {
           printf("fail to open COM port\n");
           return 1;
       }
       while ((c= fgetc(dp)) != CNTRLZ)
       {
        fputc( c, fp);
       }
       fflush(fp);
       fclose(fp);
       return 0;
}
```

图10-25 访问COM2：PC间慢速的文件传送

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
#include <winbase.h>
 HANDLE hCom;
 char inpacket[16], outpacket[16];
 BOOL fSuccess;
////////////////////////////////////////////////////////
// Initializes PC COM2 port to non-blocking mode
//
void initcomm(void)
{
 COMMTIMEOUTS noblock;
 DCB dcb;
```

图10-26 以非阻塞方式使用COM2

```
  hCom=CreateFile("COM2", GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING, 0, NULL );
  if (hCom == INVALID_HANDLE_VALUE) {
     dwError = GetLastError();
     printf("INVALID_HANDLE_VALUE()");
  }
  fSuccess = GetCommTimeouts(hCom, &noblock);
     noblock.ReadTotalTimeoutConstant = 1;
     noblock.ReadTotalTimeoutMultiplier = MAXDWORD;
     noblock.ReadIntervalTimeout = MAXDWORD;
  fSuccess = SetCommTimeouts(hCom, &noblock);

  fSuccess = GetCommState(hCom, &dcb);
     if(!fSuccess) printf("GetCommState Error!");
        dcb.BaudRate = 9600;
        dcb.ByteSize = 7;
        dcb.fParity = TRUE;
        dcb.Parity = EVENPARITY;
        dcb.StopBits = TWOSTOPBITS;
        dcb.fRtsControl = RTS_CONTROL_HANDSHAKE;
        dcb.fOutxCtsFlow = TRUE;
  fSuccess = SetCommState(hCom, &dcb);
     if(!fSuccess) printf("SetCommState Error!");
     else  printf("Comm port set OK!\n");
}
////////////////////////////////////////////////
// Reads COM2, single character
// IF no char at COM2 it returns 0, ELSE it returns ASCII char
//
char readcomm()
{
       char item;
       int ni;
fSuccess = ReadFile( hCom,
                &item
                1,
                &ni,
                NULL

  );
  if (ni >0 ) return item;
  else return 0;
}
////////////////////////////////////////////////
// tests and reads keyboard
// IF no char at kbd it returns 0, ELSE it returns ASCII char
//
char readkbd()
{
 if (kbhit() ) return _getch();
 else return 0;
}
```

图10-26　(续)

10.7　串行鼠标：机械或光学

现在的鼠标采用两种技术：电子机械结合和光学图像处理。前者应用较早，它依赖于机械器件记录鼠标在桌面上的物理移动，其内部有一个实心的橡胶球，它在桌面上滚动，带动两个边缘开槽的滚轮，见图10-27。滚轮转动时会中断红外线束，从而触发信号，记录鼠标在X和Y方向上的移动。

乍看起来，我们可以使用中断来处理按键，以及来自于检测两个滚轮在X和Y方向移动的四个光学传感器的脉冲输入。周长为80ms的橡胶球，每转动一周，滚轮的边缘会触发100次脉冲。以40毫米/秒的速度移动鼠标，产生的脉冲大约为50个/秒。尽管传感器有四个，但由于它们是成对工作，因此系统需要处理100个脉冲/秒，以及用户偶尔的按键事件。每个边缘开槽的滚轮都需要一对光束和传感器组，以提供方向和速度信息。

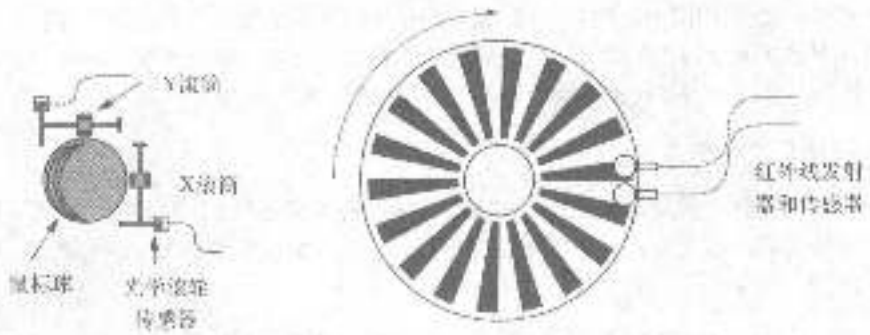


图10-27 滚轮鼠标及用来检测方向和速度的光学闸盘

这类事件的负载使用中断能够容易地管理，但公平地讲，轮询也同样能够处理。定期地轮询鼠标所带来的CPU处理负载，大约在100×50 = 5000条指令/秒。现代的微处理器每秒钟能够处理数亿条指令。鼠标轮询给CPU带来的负担不过是其处理能力的0.0005%。

PS/2串口鼠标拥有自己专用的RS232端口，它装备有专门的UART来处理与PC间的串行通信。它还会对数据进行预处理，根据鼠标在X和Y方向移动时，来自于红外线光束接收器的脉冲，递增或递减内部的寄存器。之后，它将得到的有符号整数转换成串行的字节，以1200比特率的速率传输到PC。图10-28中给出了更详细的示意性电路。

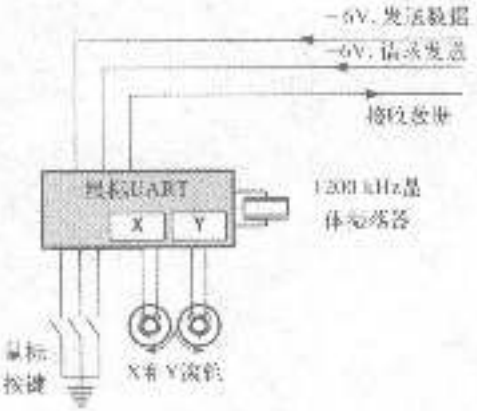


图10-28 含UART的机械式PS/2鼠标

最近发展起来的光学鼠标（见图10-29）使用完全不同的方式来监视水平方向的移动。它使用小型的数字照相机（朝向鼠标滑动的表面）和专门的数字信号处理器（Digital Signal Processor，DSP）协同工作，通过检测鼠标水平移动造成的投影图像的变化，来判断鼠标的移动。

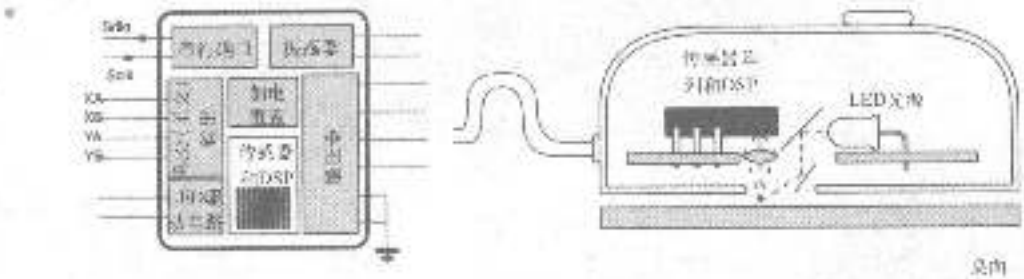


图10-29 光学鼠标的图像传感器DSP

光学鼠标的基础性技术，由安捷伦科技在2000年引入，相关专利依旧受到严格的保护。这种鼠标装有一个发射红光的发光二极管，它照向桌子表面，形成一幅供16×16传感器阵列检测的图像。256个像素中每个都被转换成6位的值，之后，传感器将256×6位的数据传递给相关的DSP进行分析。这种分析每秒进行约2000次。运行在DSP中的算法检测图像间的差异，并确定什么样的水平移动造成这样的变化。这是一项沉重的计算任务，需要大约每秒一千五百万条指令的处理能力。类似的算法还用在数码摄像机中，一般是为了提供“数字图像稳定”，或者用在MPEG编码器中用来压缩视频数据，但在这些应用中，由于处理的是现实世界的影像，复杂度要更大。在鼠标中，我们完全可以

安全地假定桌子表面在使用期间保持一致，图像的任何变化均是因为照相机的移动造成的。与PC间的通信一般不再使用更基本的RS232，而是通过USB链接来进行，这里需要一个专门的微控制器负责处理USB协议。无线链路也越来越流行，尽管为鼠标的电池充电稍有不便。

10.8 串行端口

首次将设备连接到一起时，很多原因都可能造成设备不能正常工作。整理出最明显的情形，对发现细微和不确定的问题会有所帮助。对于串行线路接口，请参阅表10-7给出的建议，以解决实际可能遇到的困难。

表10-7　处理串行连接失败的一些忠告和提示

硬件问题
插头和插座不匹配：25针与9针，插座与插头
发送和接收针脚不正确——交叉或非交叉连接
不同的插头配置
硬件流量控制（CTS/RTS）连线不正确
使用了线缆内部保留的连线
线缆内部线路组装错误
接口卡存在问题（IRQ、DMA、端口号）
串行端口硬件未初始化
不兼容的传输格式
ASCII与EBCDIC或Unicode
线路速率设置：1200、2400、9600 b/s
错误检验：奇/偶/无奇偶校验
ASCII字符长度：7位与8位
停止位的数量
用户定义的包长度
不同的文件中，CR-LF行结束符也不同
制表符与多个空格之间的不同
字处理软件控制字符
EOF问题
流量控制失败
接收方无法控制CTS输入
RTS/CTS与Xon/Xoff进行会话
端到端流量控制中存在中间缓冲区
串行线路上有未读取的回声字符
RAM缓冲区最大容量问题
软件问题
通过错误的通道发送/接收数据
安装了不正确的设备驱动程序
卸载了设备驱动程序

10.9 USB：通用串行总线

20世纪90年代早期，由计算机制造商组成的一个组织聚在一起，讨论供应商和用户在将新设备连接到PC上时遇到的问题。会上决定应该为串行接口开发新的工业标准，这个标准主要面向慢速的外设，其设计目标是提供从用户的角度看十分简单直接的接口。因而，新的标准不能要求用户手动地配置设备或重启计算机，每种设备不能要求额外的PCI卡，具有单一风格的统一接口和连线，不需要外部电源供给。最初，USB端口的速度低于1.5 MB/s，但在USB 2.0中，这一速度已增加到12 MB/s。这个带宽由所有连接到USB总线的设备共享，由于地址宽度为7位，所以最多能够连接127个

设备。其电缆内有四根导线，两条线路用做数据传输，另一对用做电力供给（5V和0V）。USB的当前实现允许同时连接127个设备。总的通信带宽被限制在12 Mb/s。但是，由于低速设备的使用，USB中断的管理以及其他开销，使得理想条件下能够达到的吞吐量也不高于8.5 Mb/s。一般的情况下大约是2 Mb/s。USB标准的最新修订版——USB 2.0中，所定义的传输速度达到48 Mb/s，可以为音频和视频应用提供高速的通道。USB使用主-从协议控制共享总线上的通信，这种方案一指定主控设备，由主控设备来查询从属设备，简洁地解决了访问裁定问题。还有一些技术方案可以避免在总线上产生并行传输冲突，有些甚至更好，但其代价是引入更大的复杂性。所有的传输都由主控设备发起，从属设备不能直接与其他从属设备建立连接。这个缺点现在已经通过USB 2.0协议的On-The-Go（OTG）扩展得以解决。



图10-30 USB插座

为了帮助用户正确地连接设备，在图10-30中给出了不同的主控插头（mini-USB A型）和从属插头（mini-USB B型）。连接的布局为树形的拓扑结构，节点处为集线路由器，见图10-31。为了适应这种分层的方案，USB设备常常提供额外的插座来连接更多的USB设备。USB标准定义了三种类型的单元：USB主机，USB集线器和连接在USB上的功能单元。主机一般为PC机。集线器可以是独立的设备，或者与外部设备结合成一体，同时提供所需的功能。

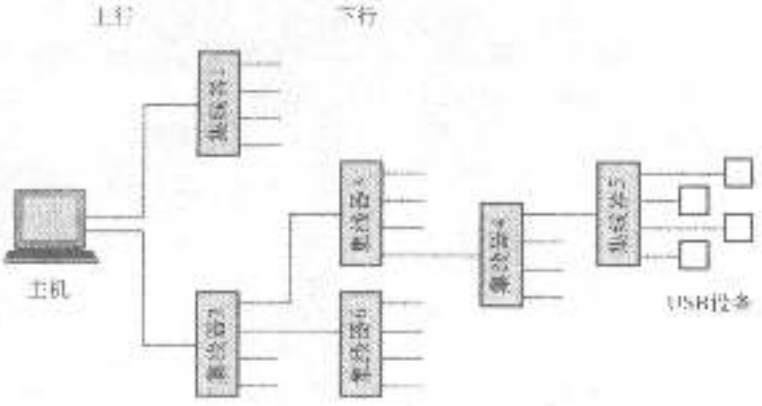


图10-31 通用串行总线的连接

前面提到过，USB采用主-从协议，使用分层星形（tiered star）拓扑结构（见图10-31），主机在顶部，集线器在中部，延展下来的终结点是一个个的设备。USB总线最多能够连接127台设备，整个树的深度不能大于6级。每集线器每毫秒（1 kHz）都用同步帧寻址所有连接的设备。这就产生了通信时间片（slot）的概念，在时间片内，控制器可以向下发送数据给设备，集线器可以请求数据，设备也可以返回数据给集线器。帧的类型有四种，列在表10-8中。

表10-8 不同类型的USB帧

控制（Control）	由根集线器使用，向设备传递配置指令和数据，尤其用在初始化阶段
等时（Isochronous）	定时的时间传送，针对能够产生实时数据流的设备
批处（Bulk）	简单的对时间不敏感的传输
中断（Interrupt）	USB不是一个中断驱动的系统，它靠来自于主集线器的周期性轮询来获取数据，比如键盘输入

一帧可以封装几个数据包，既可以是输入也可以为输出。每个帧以帧头（Start-of-Frame，SOF）包起始，数据包最多可以含有64个字节的数据，每个包以同步字节开始，以错误检测CRC字节结束。

数据串行传输的速率最大能够达到12 Mb/s。USB标准的目的是解决PC和许多外设的连接问题。消除了传统的插入新的ISA和PCI卡所要处理的种种难题。在由集线器构成的树形拓扑结构中，最多可寻址127个设备。对于处理所有的“慢速”设备来说，应该说提供了足够的扩展性和灵活度。主机软件在设计时会考虑到动态检测，因而可以在PC正在运行时安装和移除设备，所需的软件模块将会自动载入和卸下。地址分配和系统资源的分配也自动完成，表面最终用户不需处理神秘的IO地址分配或IRQ分配问题。适用的设备包括鼠标、键盘、游戏操纵杆、移动电话和数码相机。

USB线缆最长只能是5米，因而它适用于外设，但不适用于通信网络。另外，小于500毫安的电力需求可以从USB的4芯电缆中获得，因而大多数情况下，不再需要电池或额外的主电力供给。用来将设备连接到主机或集线器的电缆，每端都有不同接头——连接到PC的为A型插头，连接到外部设备端的为B型插头。USB设备也可以提供另外的插座，供连接其他设备之用。

PC系统一般都有一个内建的集线器，提供两个A型插座，因而可以直接连接一对外设。我们还可以通过安装集线器扩展插座的数量。要注意，就当前的情况来说，如果PC有两个端口，它们均属同一逻辑总线。

设备供应商和制造商对于USB标准的热忱迅速增长。他们又分成两大阵营，分别由康柏（已被惠普并购）和Intel领导。前者拥护开放式主机控制器接口标准（Open Host Controller Interface，OHCI），而后者支持通用主机控制器接口（Universal Host Controller Interface，UHCI）。一般说来，UHCI的硬件相对要简单一些，从而也更廉价，但结果是它需要更多的软件支持，从而需要更多的CPU负载，在开发小型低客户端设备时，这可能会是个问题。Intel为此专门制造了特殊版本的i8251微控制器，其中含有高速的UART用于USB的UHCI接口。

图10-32中的i8x931使用PLL（Phase-Locked Loop，锁相环）时钟同步器来同步本地的位时钟和输入的数据信号。每一帧都由特殊的SOF（Start-of-Frame，帧起始）字节作为前导，提供字节级别的同步。输入通道和输出通道都有16字节的FIFO缓冲区，以降低设备产生的中断数以及数据溢出的风险。该硬件还提供将帧路由到下行设备的支持。

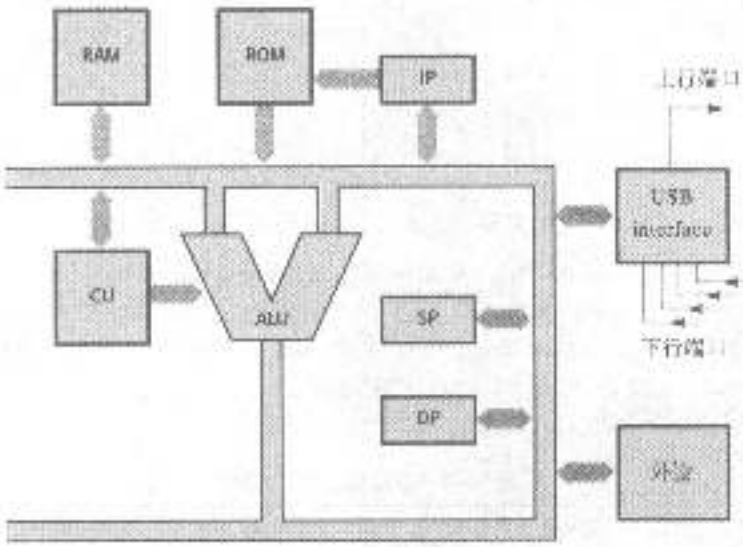


图10.32　Intel 8x931 USB外设微控制器

端点（endpoint）是USB中引入的连接概念。在最底层，每个USB设备都实现一个或多个端点。我们可以将端点看做是主机和客户设备之间进行通信的虚拟端口。端点可以是数据的发送者，也可以是数据的接收者，它具有方向性。USB 2.0最大支持15个端点。端点0是必不可少的，因为在配置阶段需要它来处理一些事务。

在通信开始之前，USB主机必须执行所谓的**枚举**过程，以检测所有连接到USB总线的设备。在这个阶段，主机会查询所有连接的设备，让它们介绍自己，并协商性能参数，比如电力消耗、传输协议和轮询速率。枚举过程由主机发起，当主机检测到有新的设备连接到总线上时，就会启动枚举过程。这些完全在后台进行，不会干扰应用程序。

在软件方面，USB使用数据结构，或称**设备描述符**，来保存诸如制造商标识、产品编号、序列号、USB设备类别以及设备支持不同配置的数量。对于任何给定的应用，只有一个设备描述符。USB定义了五种不同的描述符，分别用于设备、配置、接口、端点和字符串信息。

配置描述符提供设备对电力的需求信息，以及该配置支持多少种不同的接口。一台设备可以拥有的配置描述符可以多于一个。

接口描述符列举接口使用的端点数，以及所使用的驱动程序的类别。每种配置只能有一个接口描述符。

端点描述符保存给定功能所使用的寄存器的详情。这些信息包括数据传输的方向（输入或输出）、支持的数据类型、所需的带宽和轮询间隔。设备可以有多个端点，不同的接口可以共享端点。

这些描述符可能会引用或指向以字符串格式存储的各种信息。字符串描述符采用Unicode格式，以用户友好的方式提供与供应商或应用相关的信息。

操作系统必须提供USB驱动程序，它们根据设备类型（称为类）的共通性将功能归类。可能的设备类别包括：存储、音频、通信和人机接口。对于大多数应用来说，针对某一类设备的类驱动程序就能够满足它们大半的功能需求。

由于USB连接和数据传输方案最初是为PC市场而生的，因而在设计时就考虑到打印机、键盘、鼠标及其他传统的外设，但是，现在许多新型的手持设备、PDA、数字照相机、移动电话等，都装有USB端口。为了让照机机能够直接接到打印机，或者让PDA直接接到投影机，我们需要重新审视主控和从属设备之间的严格区分，USB 2.0中引入了新的增补内容——On-The-Go。在这种实现中，设备可以是主机、客户或者扮演双重角色。新型的微型插座和连接头被引入，它们更适合于手持式设备，它们能够接mini A型和mini B型插头。然后确定互连的设备中，哪个充当通信主控方的角色，哪个依旧是从属方。我们也可以这样说，USB-OTG已经进化得可以直接与火线技术（Firewire，即所谓的1394）竞争。

10.10 调制解调器：载波调制

电话和数据通信的结合已经有多年的历史。拨号连接（使用调制解调器——modem，见图10-30）允许计算机跨越长距离进行通信。典型的电话调制解调器能够提供的通信速率从300、1200、2400、9600、14 400、19 200、28 800、33 600，直至56 000 b/s。最初，电话公司只接受音频波段（300～3000 Hz）的数据传输。这限制了早期的频移键控（Frequency Shift Keying，FSK）调制解调器，使之只能提供300 b/s的缓慢传输速率。采用这种技术时，二进制的1和0通过发送不同的音频来发送，类似于使用两种不同音调的哨声。但很快，人们就意识到，通过同时变化相位和振幅，我们可以在每个哨声的周期内传输相当多的数据，但传输的载波依旧局限于音频的“基带”范围以内。经过一段时间以后，电话公司才允许更高频率的高频载波经由它们的线路传输。一旦证实这种方式能够胜任最长达5公里的距离，“宽带”DSL调制解调器的市场就发展起来，由于它们突破了3 kHz的限制，从而能够以很高的速率传输数据。

“modem”一词是“modulator-demodulator”的缩写。由于电话系统已完全转向数字技术，曾经有一段时间，人们认为调制解调器已经过时。但现在看起来并非如此，用于小范围电缆网络及DSL标准（参见16.6节）的新一代的调制解调器已经发展起来。人们已经充分意识到在专为传送模拟信号而设计的电话线上传送二进制计算机数据的难点，常规的解决方案是在计算机和电话插座之间安装调制解调器。图10-33即为这种方式。调制解调器将计算机的逻辑电平1和0转换成不同的音调，或啸叫。如果曾经拨打过装有调制解调器或传真机的电话号码，就会听到过它们。音调的频率必须

在正常的语音范围之内（300～3500Hz），这样才能通过电话通道传送。图10-34给出两种可供使用的方案。第一种发送一种音调来表示1，但没有表示0的音调。另外一种叫做FSK的方案更可靠一些。它使用不同的音调表示1和0。由于连接用户的链路一般为二线，能够完成双向的通信，因此现实生活中，存在使之工作在全双工模式的需求——同时传送和接收数据。为了满足这项重要的需求，此处需要使用四种音调，每个方向两个。

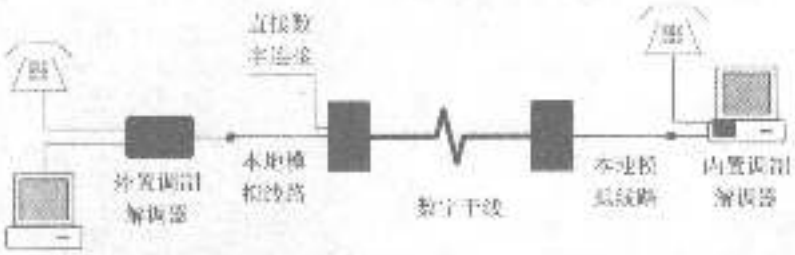


图10-33　使用调制解调器跨电话网络传送数据

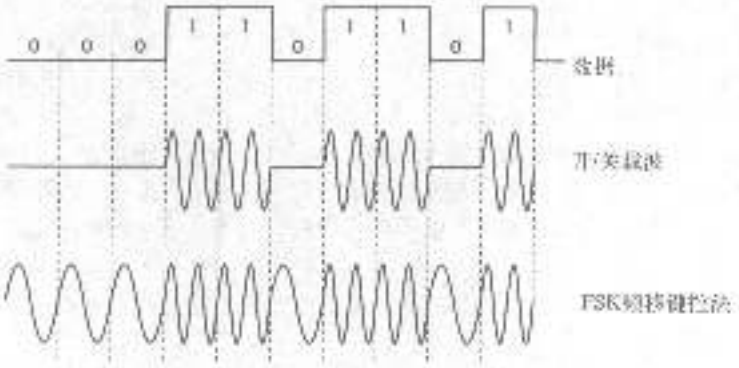


图10-34　频率调制技术

其他难点包括电压标准、噪声级别和带宽限制。RS232 COM端口使用+9/－9V电压，而正常的电话用户线路使用0/－50V。另外，电话语音信道所能够传送的最大频率是3.5 kHz，而计算机COM端口常见的传输速率是9.6 kb/s，就表面上看，要求使用19.2 kHz的信道。另一个需要解决的问题，是两线的用户线路接口到终端设备要求的四线接口之间的转换。通过同一端口，在同一线路上发送和接收数据需要一些技巧。传统的电话技术还能容忍电子干扰，不过会在我们的会话中产生爆裂声和噼啪声。新的数字时代，尽管能够提供更大程度的“噪音免疫性”，但需要复杂的错误检测和纠正机制。调制解调器在解决这些问题上起到了重要的作用。

调制解调器有多种国际标准。表10-9列出其中的一些。现代的调制解调器一般能够使用取自于贺氏AT指令集（见表10-10）的命令进行初始化。最常见的配置字符串是ATZ，它是硬件重置命令。初始化命令由PC发送给调制解调器，但调制解调器也在内部存储配置信息，可以使用ATF1命令访问。另外，现在调制解调器内建处理器的情况也很常见。这样，调制解调器就可以处理初始化对话，并能提供可选的数据压缩机制，以提高数据传输的速度和能力。最常见的是由Microcom公司开发的MNP-5，ITU以它为范本，在做出一些改进的基础上制订了V.42bis标准。它使用Lempel-Ziv-Welch（LZW）压缩算法，对纯文本可以达到3:1的压缩比。更先进的调制解调器，在常规的4 kHz有限带宽的电话线路上，提供38.4 kb/s或56 kb/s的传输速率。它们依赖于有效的数据编码和压缩来达到这么高的数据传输率，如果数据已经压缩过，那么有效的数据传输率就会相应地下降。

表10-9 调制解调器标准和编码方案

ITU分类	能 力	类 型
V.21	300/600 b/s	频率偏移
V.22	1200 b/s	相位偏移
V.22bis	2400 b/s	振幅和相位偏移
V.29	9600 b/s	相位偏移
V.32	9600 b/s	振幅和相位偏移
V.32bis	14.4 kb/s	振幅和相位偏移
V.17	14.4 kb/s	传真
V.34	33.6 kb/s	QAM，振幅和相位偏移
V.90/2	56.0 kb/s	PAM（Pulse Amplitude Modulation，脉冲幅度调制）
	< 10 Mb/s	Cable/DSL调制解调器

表10-10 贺氏调制解调器AT指令集中的一些命令

命 令	功 能
ATA	应答呼叫
ATDnnn-nnnn	语音拨号nnn-nnnn
ATL	重拨
ATPDnnn-nnnn	脉冲拨号nnn-nnnn
ATW	等待拨号音
ATH0	挂机
ATM0	关闭扬声器
ATM1	扬声器保持打开，直到检测到载波为止
ATM2	扬声器持续打开
ATO0	将调制解调器置于数据模式
ATO1	将调制解调器置于非数据模式
ATY0	禁止暂停时断开
ATY1	允许暂停时断开

图10-35给出了如何通过将几个比特编入单个周期内，从而降低以恒定传输速率传送数据所需的载波频率。但这项技术常常用来增加比特率，而非降低载波频率。采用这种技术，2400 Hz的调制解调器载波能够用来以2400 b/s、9600 b/s、19200 b/s或者更高的速率传输数据，实际速率依所选的编码方案的不同而定。

能够快速区分出同一频率的不同相位的能力依赖于**DSP**（Digital Signal Processor，数字信号处理器）技术的应用。DSP芯片提供足够快的算术处理速度，能够跟踪载波并检测相位内的急剧变化——表示数据的值。较早的技术能够区分出π/2（90°）的相位变化，新技术能够可靠地识别出小至π/8（22.5°）的相位差异。DSP芯片还能够完成线路均衡，包括补偿信号的衰减效应以及执行回声消除。这样，调制解调器就能够在一对线路上以全双工模式工作。这种模式要求DSP芯片能够从叠加后的输入信号内减去强的输出信号，将发送和接收的数据有效地分离。

图10-36的振幅相位示意图——有时也叫做星空图，给出了一些相位调制技术在现代调制解调器中的应用实例。QAM调制解调器使用单频载波，其相位和振幅均变化。

正交幅度调制（Quadrature Amplitude Modulation，QAM）结合振幅调制和相移键控，在每个载波周期内编入多个数据位。在图10-36中，第一个图表示简单的相位移位器，它通过按照既定的周期传送一个突发频率来发送二进制1信号，发送二进制0时将相位翻转π。第二幅图阐述相位和振幅同时变化，使发送方每个时隙能够发送4个比特。它通过使用载波的四种可辨别的变化来完成这种功能：高振幅+0相位，低振幅+相位，高振幅+180°相位偏移，低振幅+180°相位偏移。第三种方

案叫做QAM，载波的频率保持恒定，振幅有两种高度，相位有四种不同的值：45°、135°、225°、315°。最后，最复杂的方法使用四个振幅和八个相位设置，提供32种不同的载波变化。

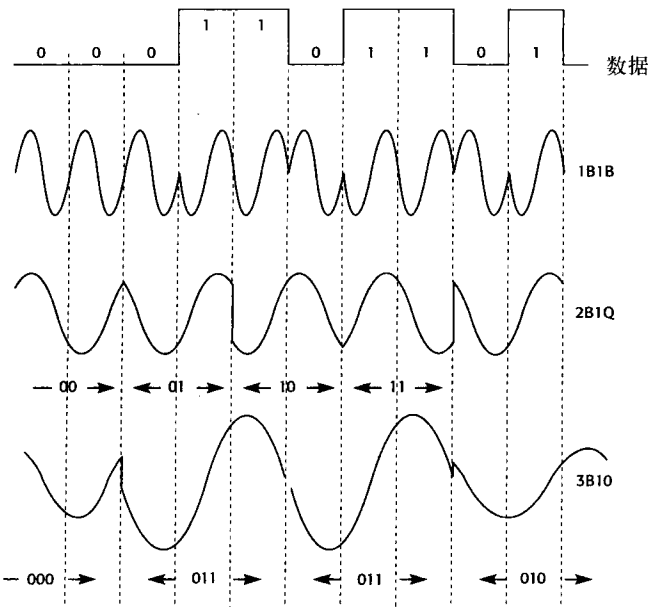


图10-35　相位调制增加了位信号发送的速率

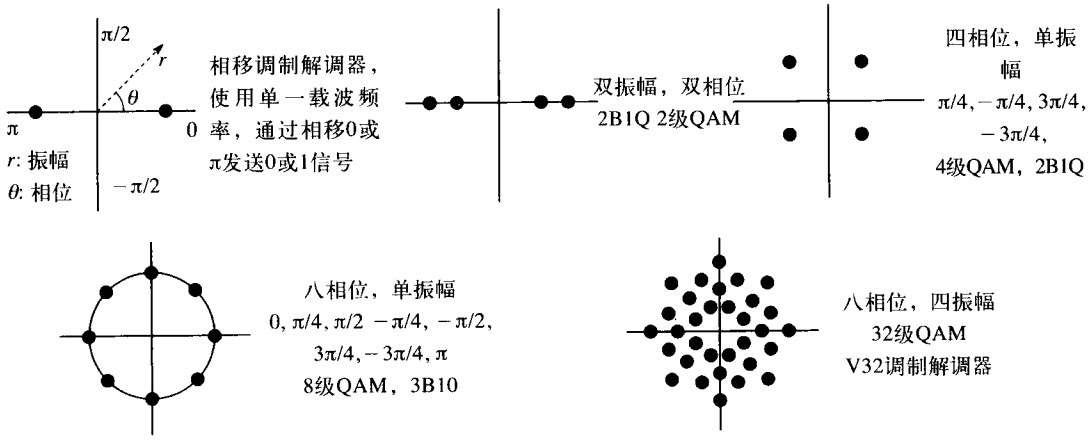


图10-36　一些调制方案的振幅−相位示意图

在表10-11的例子中，传输的信号每周期有3位，可以表示8个数值。其中用到两种不同振幅和四种相对相移，即8级QAM。

以传输下面这些位为例：

001010100011101000011110

首先，我们需要将它们每三个分成一组，就像转换成八进制表达方式一样，之后才能选择适当的振幅和相位：

001_010_100_011_101_000_011_110

现在我们需要做的，就是确定最后的信号应该如何。

表10-11　8级QAM的编码

比特值	振幅	相位偏移
000	1	None
001	2	None
010	1	+π /4
011	2	+π /4
100	1	+π /2
101	2	+π /2
110	1	−π /4
111	2	−π /4

需要注意的是，对波形偏移是相对于前面的波形进行。

V.90（56 kb/s调制解调器）采用完全不同的方式传输数据。有了这种设备，我们基本上可以将电话系统看做是数字网络，仅从本地电话局的交换机到消费者之间是一段短的模拟连接。另外，这种连接是不对称的，服务器端得益于与本地交换机的直接数字连接。见图10-33。这种方案的优点是能够降低由于对数据执行模-数和数-模转换而引入的噪音。V.90调制方案不将数据转换成语音振幅，而是直接使用8位的值来表示数据，能够利用干线的8位PCM编码。故而，服务器直接将数字值传递给数字电话网络进行传输。这也是数字56 k的由来。由于数字化语音传输的速率为8 kHz，数字化的数据被限制在7位，以改善错误率，从而：7位 × 8 kHz = 56 kb/s。客户端的调制解调器必须能够区分128种不同的电平，每种电平分别表示0到127间的某个离散二进制数。要做到这一点，调制解调器必须小心地与8 kHz的传输时钟保持同步，以保证采样时间的准确。一般地，只有下行方向使用这种方法，而上行链接只运行在33.6 kb/s，使用V.34，QAM标准。这意味着两个客户端，即V.90调制解调器，不能跨电话网络以56.6 kb/s互连。

DSL和有线调制解调器使用更高的载波频率，其频率位于VHF高频波段。有线网络还使用光纤将数据从中心办公室传输到社区，最后通过同轴电缆将数据分发到最终用户那里。

广泛使用的高频载波调制解调器有四个功能。正交幅度（64-256QAM）解调器接收载有信息的高频载波信号，从载波中将数据分离出来。它采用的方法是检测由发送调制解器施加到载波上的幅度和相位的变化。检测的结果为模拟数据信号，该信号之后由快速的模－数转换器（Analogue-to-Digital Converter，ADC）转换成数字形式。而后，错误纠正模块检查接收到的数据，查看是否有错误发生，如果需要，则进行修复。许多有线公司使用MPEG帧格式来传输数字电视信号及Internet数据。**有线网络数据服务接口规范**（Data Over Cable Service Interface Specification，DOCSIS）能够保证不同的提供商和网络运营商能够在一起有效地工作。

10.11 小结

- Internet建立在计算机之间的串行通信的基础之上。两台PC机可以使用RS232标准通过COM端口建立直接的点对点连接。
- 通信包括三个重要的方面：数据、时序和信号传送。
- 接收方需要识别出每个输入位的中点，这样才能精确读取。这就需要知道位的宽度（比特率、波特率）以及位的起始点（相位信息）。
- 异步传输需要发送方在每个字节前插入一个脉冲信号，用来提示和同步接收方。同步传输用在更快速的数据传送中。此时，发送方会伴随数据发送连续的时钟信号，使接收方与之完全同步。
- 在传输过程中可能会发生错误。在有额外的冗余信息与数据一同传送的情况下，错误检测甚至纠正都是可能的。单个奇偶校验位是最常见的错误检测方法。
- 采用多个奇偶校验位的汉明码能够检测和纠正错误。
- 块检验和能够检测存储和传输中的错误，且易于实现。但CRC更有效率，应用也更广泛。
- 为防止快速的发送方超过慢速接收方的处理极限，我们需要采用流量控制系统，否则接收缓冲区内会发生溢出错误。对于RS232，可以使用硬件（CTS/RTS）和软件（XON/XOFF）两种方式来完成流量控制。
- 简单的点对点串行通信不存在路由选择的问题。
- UART是一种IO接口芯片，它将数据从并行格式转换成串行格式。它还能执行基于奇偶校验的错误检查，以及RS232流量控制。
- 使用COM端口的鼠标内部，有一个特殊的UART。
- USB是一种新型的高速串行传输标准，用于连接不同的外设。
- 通过传统的模拟电话线传输串行数据时，需要使用调制解调器。它们对音频载波进行调制，来表示需要传输的数据。

实习作业

我们推荐的实习作业包括使用几台PC机的COM端口，实现一个初级的包传递环形网络。除非仅仅限于两台PC机，否则就要制作一根特殊的电缆。初始化、输入和输出例程都使用Win32函数。相比于前一章中的点对点方案，这会使我们面临由网络广播带来的许多问题。要注意避免阻塞系统调用。

练习

1. 异步通信和同步通信之间的区别是什么？下面这些方式是同步还是异步？
   • 电视广播；• 发送传真；• 以太网；• CD-ROM。
   解释为什么。
2. 勾画通过RS232传送字母A时的位模式，假定传输时使用8位、偶校验、1个停止位。
3. 当前传真调制解调器的比特率是多少？Group III传真扫描设备传送一幅未压缩的A4纸要花多长时间（Group III共分1142行，每行1728个黑/白像素）？假定以9600 b/s传输。
4. 数字化数据传输的基本优势是什么？
5. 解释CTS和RTS信号。在与调制解调器通信时，它们最初的功能是什么？在计算机之间通过串行线路直接传送数据时，又是如何使用它们的呢？
6. 为什么数据通信中一定要采用某种形式的流量控制？提供输入缓冲区是不是就不再需要流量控制了呢？
7. 有线电视信号是模拟信号还是数字信号？
8. 通过电话线传送2500个字符要花多长时间？
9. 以19 200 b/s的速率接收数据，在启用接收缓冲区的情况下，系统处理中断请求的频繁程度如何？
10. 试着计算图10-15中S记录文件的某个检验和。
11. 试着将10.4节中汉明码示例中的另外一位置为错误的值。重新进行计算，看看检验子是否正确。
12. 查看图4-9，判断四个输入流上承载的数据。

课外读物

- Heuring和Jordan（2004）中有关错误控制方法和串行通信的内容。
- Hyde（2002）针对程序员从技术的角度介绍了USB。
- Tanenbaum（2000），2.2.4节，错误纠正码（纠错码）。
- Anderson（1997）。
  http://users.tkk.fi/~then/mytexts/mouse.html
- 有关USB接口的更多技术信息和应用的例子，试试访问下面的网址：
  http://www.intel.com/design/usb/
- 调制解调器的相关术语：
  http://www.physics.udel.edu/wwwusers/watson/student_projects/scen167/thosguys/
- ADNS-2051光学鼠标传感器的资料表：
  http://www.agilent.com
- 标准家用调制解调器的资料：
  http://broadband.motorola.com/consumer/products/sb5120

第11章 并行连接

计算机内总线上的数据传输是以并行模式执行的，即多个位沿一组线路同时发送。在数据传输中通常使用的异步握手技术，是为Centronics打印机端口而引入的。SCSI和IDE互连总线是一种十分重要的标准，它们能够将其他设备可靠地连接到PC机上。

11.1 并行接口

在计算机内，数据沿着称为**总线**（bus）的并行导线传送。数据、地址和控制总线将计算机维系成一个整体。如3.6节所述，在设计新的计算机系统时，地址总线的宽度（导线的数量）是一项至关重要的参数。从20世纪70年代起，每一代微处理器不断加宽地址总线和数据总线，以获得更高的效能。现在，奔腾处理器使用64位的外部数据总线，以缓解“总线瓶颈”的束缚。使用64位的总线，可以同时将两个长字传送到内存，或从内存中取出。而在CPU自身内部，为进一步提高传送速率，数据总线甚至会达到128位宽。

一般地，使用并行连接的设备，诸如打印机或屏幕等，并不直接安装到系统总线上，而是需要**接口**电路来管理连接。电压、时序和控制信号上的差异，使得直接连接存在问题。前面3.8节已给出一种最简单的并行接口，但在实践中，要将控制总线转换成与设备控制线路兼容的形式，仅仅靠8位**锁存器**是远远不够的。要求连接能够双向工作的需求，即支持输入和输出操作，使得这个问题变得更加复杂。

总体上，让用户和程序员感到最棘手的领域之一，依旧是外部设备和计算机之间的连接。尽管现在我们已经完成了许多标准化的工作，引入了即插即用，并且在大、中、小型计算机方面拥有多年经验，但这个问题依旧难以解决。

11.2 Centronics：大于打印端口但小于总线

- 多条导线的并行链接，比如Centronics打印机接口，可以一次性地传输8位ASCII码数据。短距离内（10米），传输速率最大可达100 KB/秒，但这种接口只提供点到点的主-从服务。这意味着，将它看做总线是不恰当的，因为总线互连的基本特性之一，是能够向多个设备（>2）提供服务。Centronics接口现有三种标准，能力各不相同。详情参见表11-1的汇总。

表11-1 PC并行端口（Centronics）标准

SPP	标准并行端口 (Standard Parallel Port)	100 KB/秒	输出	软件控制
EPP	增强型并行端口 (Enhanced Parallel Port)	1 MB/秒	输入-输出	硬件握手电路
ECP	功能扩展型端口 (Extended Capability Port)	5 MB/秒	输入-输出	带FIFO的DMA

标准并行端口（SPP）专门针对数据输出，它只提供执行异步握手所必需的少数几条输入线路。由于对控制引脚和状态引脚的检验和设置工作都由软件来完成，因而传输速度十分有限。硬件实际上不过是一个简单的字节宽度的并行端口，传输的最大限制约100 KB/秒。

向打印机发送数据时，计算机通过向端口的数据寄存器输出数据来设置8条数据线。如图11-1所示，等待一段时间，待信号稳定后（50ns），将选通信号置为低电压至少需要0.1μs。打印机以这

个选通信号作为“数据出现”的指示。在情况允许时，会立即读取数据端口。完成数据的读取后，打印机在ACK线路上发送4μs的脉冲答复计算机。如果打印机跟不上计算机的处理速度，它会使用一条单独的线路（BUSY）阻塞来自于计算机的后续传送。但是，流量控制的实现往往要靠ACK信号来完成。图11-2给出了另一种描述方式，它使用时序图，程序员对它可能会更熟悉些。

D25引脚	SPP	主机打印机
1	选通信号	→
2	数据位0	→
3	数据位1	→
4	数据位2	→
5	数据位3	→
6	数据位4	→
7	数据位5	→
8	数据位6	→
9	数据位7	→
10	ACK	←
11	BUSY	←
12	缺纸	←
13	SLCT	←
14	自动进纸	→
15	出错	←
16	INIT	→
17	SLCT IN	→
18-25	地	

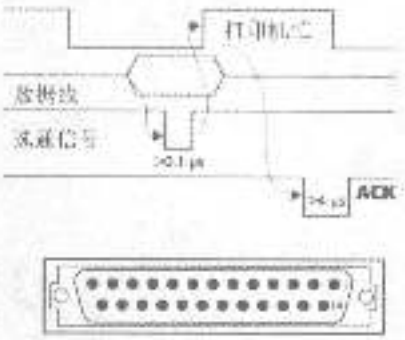


图11-1　Centronics标准接口（SPP）

图11-2描述的方案就是异步握手方案的一个实例。使用这种方案时，仅当接收方就绪后才发送数据，因此发送方和接收方设备所使用的时钟频率可以相差很大，内部的运行速度也可以不同。

并行连接看起来好像比串行链接具有较大的带宽优势。那么，既然串行链接要八倍，为什么我们还要使用串行链接呢？答案和电子工程技术密切相关。全部8位数据同时沿并行线路传送时，它们要受到电子信号衰减的影响。分立导线不同的电阻和电容会引入不同的传输延迟，从而使得脉冲不能对齐。工程师已经意识到这类问题，并行总线上最终的信号可能会发生严重的对齐问题，甚至会完全破坏最后接收到的数据值，以至于程序员也不能不考虑这类问题。

Centronics标准经过扩展，包括双向EPP和ECP模式。在这两种模式下，数据线不但能用来输出数据，同样能用来输入数据。另外，4条状态线被重新部署用于其他用途。见表11-2。双向数据传输对于支持即插即用设备是必需的，因为宿主计算机在启动阶段需要对设备进行询问，由设备来标识自己。



图11-2　Centronics数据传输中的事件序列

表11-2　Centronics增强接口（EPP/ECP）

D25引脚	EPP	主计算机	打印机
1	写	→	
2	数据位0	↔	
3	数据位1	↔	
4	数据位2	↔	
5	数据位3	↔	
6	数据位4	↔	
7	数据位5	↔	
8	数据位6	↔	

（续）

D25引脚	EPP	计算机	打印机
9	数据位7		↔
10	中断		←
11	等待		←
12	用户定义		←
13	用户定义		←
14	选通信号		←
15	用户定义		←
16	重置		→
17	地址选通		↔
18-25	地		

在进行异步数据传输时，握手机制支持各种不同的数据传输速率，以匹配通信双方中最慢的一方——不过，等待打印结束可能不是一件愉快的事，但有许多方法可以避免这种不便，如表11-3所列。

表11-3 如何防止由打印引起的延迟

1. 在打印机内安置较大的存储缓冲区
2. 在计算机上安装一个后台运行的假脱机程序
3. 使用完全多任务的操作系统（如Unix、Windows NT、Windows 98）

大多数计算机工作站现在运行多任务操作系统，它们能够在不影响前台交互活动的情况下操控打印机。简单一些的计算机系统可以采用将打印驱动程序加载为中断例程，提供类似的机制。这些驱动程序读取需要打印的数据，将这些字符传递给打印机，有时这个过程可能会附加到键盘扫描或实时时钟更新过程中。采用这种方式，能够在不影响前台的主要活动的情况下，为慢速的打印机提供充足的数据，使之保持忙碌。

在过去，将Centronics打印机端口用做其他用途，比如访问安全**软件狗**（dongles），或作为可擦写CD驱动器的接口时，会存在一个问题。即使实际使用的芯片常常有能力处理双向活动，但SPP模式限制硬件只支持数据输出。现在这种情况已经得到改变，处于EPP或ECP模式时，“打印端口”能够服务于许多其他功能。这些快速的模式由额外的硬件提供支持，这些硬件提供流量控制握手机制，减轻了软件在检测和同步方面的负担。程序只需将一个字节的数据输出到输出端口，端口硬件就会检查打印机是否忙，并发送选通脉冲。这些模式一般不检查ACK线路。

Windows NT引入以后，人们突然对这种机制又开始感兴趣。对于DOS到NT的巨大变化，接口卡的制造厂商普遍没有准备，他们发现他们的DOS软件在更安全的NT环境下不能工作。快捷的解决方案是将他们的特殊设备移出计算机，转而使用PC上普遍装备的并行端口。但现在，USB接口已经成为连接外部设备更普遍的选择。

11.3 SCSI：小型计算机系统接口

SCSI并行接口提供快速的异步、单字节宽的总线（5 MB/s），它最多可以连接8个设备。主机适配器（在PC机中应该是ISA或PCI扩展卡）被分配为SCSI地址“0”，SCSI地址通过电路板上的微型开关来设置。由于主机适配器也是SCSI单元，因而将几台PC连接到同一SCSI总线是可行的。这是实现磁盘共享的一种方式。SCSI总线上安装的其他设备可以使用任何其他地址。不同寻常的是，地址是直接设置某个恰当的地址线，以译码后的形式在数据总线上传递。因此，地址0对应地址线0，地址2对应地址线2，依次类推。使用这种方式可以同时寻址几个目标设备。在将新的设备连接到SCSI总线上时，切记要使用不同的地址。SCSI是为硬盘驱动器、CD-ROM或磁带设备设计的。

SCSI是计算机内的次级总线，它可以降低主总线上的通信量，从而提高计算机的性能。从最初5 MB/s的标准开始，后来发展出来的各种标准的速度成倍增加，同步传输模式（Fast SCSI）为10 MB/s，32位数据总线的变体（Wide SCSI）达到40 MB/s。

SCSI通信事件有三个基本阶段：总线抽获、设备选择和信息传送。如图11-3所示，为了更好地理解SCSI传送发生的方式，最好将不同的阶段在状态图中表示出来。启动通信的发起者和负责应答的目标设备互换数据包。由于每个设备都被分配了惟一的总线地址（0—7），因而几个会话可以同时重叠进行。PC主机适配器卡并不一定会是发起者。任何设备都可以启动与指定目标设备的通信。如前所述，这种多总线管理的能力，使得SCSI总线可以用做多处理器计算机中跨处理器通信的干线。

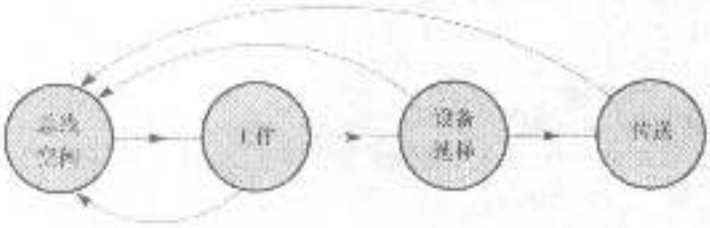


图11-3　SCSI传输中的相位序列

发起通信会话的主控设备首先通过查看BSY线检查总线是否可用，如果可用，则取得其控制权，通信会话开始。主控设备会将它的8位ID复制到数据总线上，同时使用BSY控制线声明数据总线忙，如果没有拥有更高优先级的设备响应（0为低优先级，7为高优先级），则成功获得总线的使用权，可以进行数据传输。新的主控设备将含有一个代码（见表11-4）的查询包发送给希望建立通信的目标设备，目标设备必须用状态信息来应答。数据传输才能够继续进行。

从图11-4中，我们注意到宽度为1个字节的数据总线由单个奇偶位保护，因而10.3节与此处的内容有关系！SCSI控制线（下面列出）能够执行异步握手，将数据通过总线传送。

表11-4　SCSI信息代码

第1组			
00	Test unit ready	13	Verify
01	Rezero unit	14	Recover buffer
03	Request sense	15	Mode select
04	Format unit	16	Reserved unit
05	Read block limits	17	Release unit
07	Reassign blocks	18	Copy
08	Read	19	Erase
0A	Write	1A	Mode sense
0B	Seek	1B	Start/stop
0F	Read reverse	1C	Receive diagnostic
10	Write file mark	1D	Send diagnostic
11	Space	1E	Lock media
12	Inquiry		
第2组			
25	Read capacity	30	Search data high
28	Extend addr rd	31	Search data equal
2A	Extend addr wr	32	Search data low
2F	Write 7 verify	33	Set limits
2F	Verify	39	Compare
		3A	Copy and verify

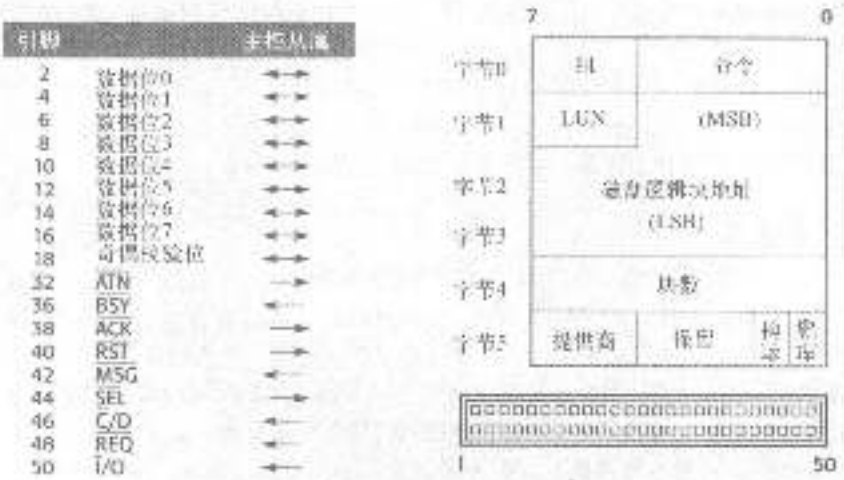


图11-4 小型计算机系统接口（SCSI）及其命令包

$\overline{\text{BSY}}$：Busy（忙），表示当前有设备使用总线。

SEL：Select（选择），由发起者用来选择一个设备，或由目标设备用来重新开始被中断的会话。

C/D：Control/Data（控制/数据），由目标设备控制，用于指明当前数据总线上传输的信息是控制信息还是数据信息。

I/O：Input/Output（输入/输出），允许目标设备定义数据传输的方向。

$\overline{\text{ACK}}$：Acknowledge（应答），用来确认请求。

$\overline{\text{ATN}}$：Attention（注意），主控设备用它通知从属设备，总线上有数据。

$\overline{\text{MSG}}$：Message（消息），在消息传输阶段由目标设备激活。

REQ：Request（请求），目标设备用它来发信号给主控设备，通知数据可以传输。它是REQ/ACK握手对的组成部分。

$\overline{\text{ACK}}$：Acknowledge（应答），由发起者控制，用来确认传输。

$\overline{\text{RST}}$：Reset bus（重置总线），强制所有安装的设备停止活动，重置硬件。

命令包的结构在图11-4中给出。由于命令包有几种不同的大小，因而每个数据包的开始都记录包的大小。Group 0使用6个字节，Group 1和2使用10个字节。

50路IDC扁平线以菊花链的方式连接在一起，卡到卡的距离最长可达6米。它的每一端都必须正确地终结，以避免产生有害的电子反射。所有的信号线都有与之配对的地线，使得传输更干净。在连接新设备时，一定要为它分配惟一的总线地址（1～7）。不管是通过DIL开关还是PCB跳线，都要正确地设置总线地址。两个设备响应同一地址会造成混乱。SCSI总线的确提供更高的性能、速度和能力，但成本也比IDE要高。它在PC机上的应用很少，但在大型系统中十分常见。它的一大优势就是只需单个中断（IRQ）。就能够为最多7个磁盘驱动器服务。而使用IDE连接7个磁盘驱动器，则要使用4个IRQ，这是不可能实现的。主板很少将SCSI控制器作为标准设备提供。因此，我们还需购置SCSI控制卡，这使得成本进一步增加。它们还可以集成快速的DMA控制器和RAM缓冲器，以提供远快于标准PC中DMA控制器的数据传输速度。

烧录CD-R盘片时，使用SCSI磁盘还有一项特殊的优点。因为在烧录CD-R时——我们在12.9节中将会看到，必须为刻录机提供不间断的输入数据流（1倍速刻录需要300 KB/s），否则会发生"欠载（underrun）错误"。在依赖于IDE磁盘驱动器时，这种问题比较常见，但SCSI控制器支持请求排队，因而能够降低两次数据块读取之间的时间延迟。

由于4速和8速CD-R/CD-RW越来越普及，从而要求输入数据流越来越快，SCSI总线的优势再次为人们所赏识。

SCSI总线的另一项常见应用是RAID磁盘阵列，RAID磁盘阵列并行部署多个磁盘，提供一共指

施保护数据不受硬件故障的影响。通过使用错误纠正码，或简单的数据镜像，RAID已经成为PC机上可靠数据存储的普遍方案。在19.5节中，我们将更详细地探讨这一主题。SCSI的另一种不那么常见的应用是，作为多处理器超级计算机中处理器间的数据高速公路。在这些系统中，每个处理器除连接到通常用于访问本地存储器和IO设备的系统总线以外，还提供单独的端口连接到SCSI总线，通过SCSI总线，CPU能够快速地交换与进程调度和资源分配相关的信息。

11.4　IDE：智能驱动电路

IDE接口是ISA总线的简化版，它是为应对PC硬盘驱动器日益增长的大小和速度而引入的。早期IBM为ST412/506磁盘驱动器设计的接口需要单独电路板，其上遍布电子元器件，以将数据从时钟信号中分离出来，并组织磁头在磁碟上的移动。磁盘驱动器内只安置马达控制设备。这种情况下，跟上市场上流行的各种不同的驱动器和控制卡（它们有时还不兼容）就比较困难，而且经由互连扁平线的数据传输速度也是个问题。来自磁盘驱动器的信号几乎完全未经处理，极易受到干扰和衰减。通过在磁盘驱动器中提供更多的电路，可以在传输到计算机总线之前，将IDE信号进行部分处理。通过在驱动器中集成控制电路和驱动机制，传输和兼容性问题都得以解决。通用访问方法（Common Access Method，CAM）团体（由多家制造商组成）在1989年制订的ATA（AT attachment standard）标准定义了用来与IDE设备通信的命令集，其中包括约30条指令，计算机与IDE设备之间的通信就用这些指令来完成。

智能（或集成）驱动电路的引入，使得PC机用户能够单独升级他们的硬盘驱动器，不用同时升级昂贵而复杂的接口卡。由于相关的磁盘驱动器价格低廉，而且十分丰富，这项标准很快被采纳。基本的控制电路包括在驱动器上，40路IDC扁平线插头直接连接到AT总线。每个电缆只能服务于两个驱动器，其中之一为主控驱动器（Master），另一个为从属驱动器（Slave）。一般会使用跳线来强制这种区分。如果同一电缆上的所有驱动器都被设为主控，则它们都不能工作！尽管不同的驱动器在磁道和扇区规格方面存在物理差异，但该接口使得它们都按照传统的ST506驱动器的响应方式工作。因此，40个扇区的磁道会被转变成好像17个扇区的磁道。最初的IDE标准设了1024磁道、8个磁碟（16磁头）和63个扇区的上限，合计起来最大磁盘大小为504 M。奇怪的是，磁头的限制不是由于驱动器寄存器中相应字段的宽度限制而造成的，而是来自于低级的驱动程序例程（BIOS）。因而，增强型BIOS通过将4位磁头数字段改成8位（和Y2K问题并无二致），就将这个大小增加到7.88 G。新型的**EIDE**（Extended Integrated Drive Electronics，增强型集成驱动电路）标准能够访问65 536磁道、256个磁碟和63个扇区，合计最大磁盘大小为128 G。

格式化也存在两种不同的级别。磁道和扇区的低级格式化在工厂内完成，重新格式化不容易。只有次级格式化（设置分区和目录结构）可以由用户执行。

访问IDE磁盘上数据的例程，现在由操作系统提供。一般不需为个别制造商生产的部件提供专门的设备驱动程序软件。与SCSI的描述非常类似，IDE总线也有三种阶段（或周期）。首先，CPU将参数写入到IDE命令寄存器。接下来，载入柱面、磁头和扇区信息。之后就可以进行数据块的传输。

在过去的某些时候，CPU被安排来处理所有实际的数据传输，同时还要负责生成初始的命令。由于当时CPU的设计和制造已经应用了最新的技术，但外设部件却没有，因而DMA控制器处理相同的工作时要慢得多。因此，所有数据都以轮询的方式访问。但是，使用新的DMA-33，情况已经有所改观，IDE驱动器现在使用DMA机制传输它们的数据。

最初的IDE接口标准已经增强为EIDE。驱动器的控制通过向命令寄存器写入一系列的控制代码来完成。这是设备驱动程序例程的责任，我们可以使用HLL函数调用，如open()和fprintf()，来访问它们。

11.5　AT/ISA：计算机标准的成功案例

最初，IBM PC的62路扩展总线中，包括8位数据总线和20位地址总线。但它很快被扩展到能够处理16位数据和24位地址，并逐渐脱离人们的视野，现在仅在一些老的接口卡（见图11-5）中才能

看到它的身影。大部分新主板板卡不提供ISA插槽，它们更倾向于采用PCI和USB。由于Intel 8088处理器运行在4.77 MHz，因而ISA扩展总线也以这个频率运行。因为ISA总线可以看做是系统主总线的延伸，同时提供接口控制信号。在提供ISA总线的现代PC中，输入的数据在到达主板的系统总线时，需要通过两个入口控制设备（见图11-6）。尽管处理器和主板的运行速度日新月异，但为了维护与较慢的接口芯片的兼容性，ISA总线时钟依旧保持8.33 MHz。由于该总线与系统的主时钟同步运行，因此，CPU需要插入额外的WAIT状态，以匹配慢速的外设，这会对系统的性能产生很大的负面影响，但通过引入总线入口控制芯片，主板和ISA总线实际上已经解除了这种耦合关系。在11.6节中，还会对此做进一步的介绍。

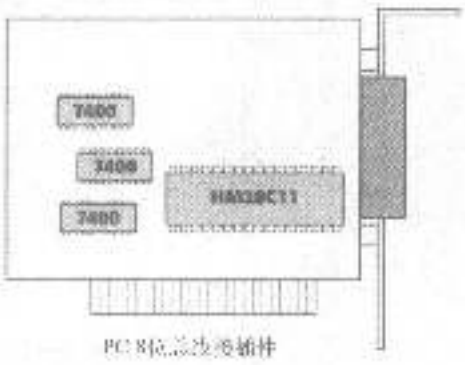


图11-5 8位PC/ISA总线打印机接口卡

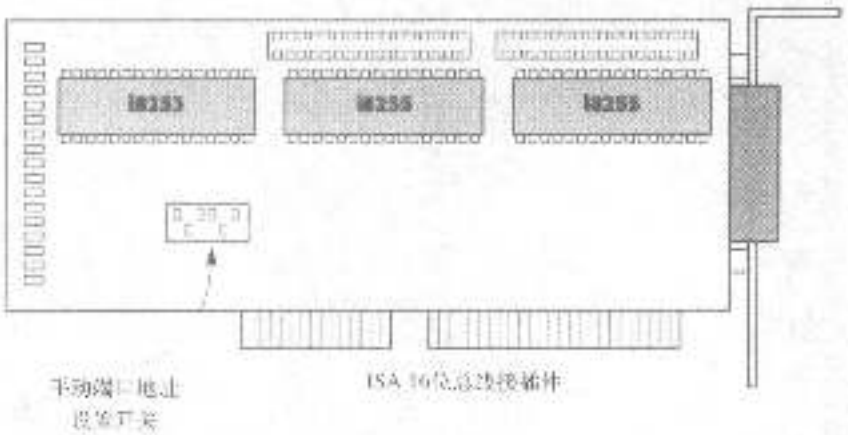


图11-6 16位扩展PC/ISA总线并行IO卡

* ISA总线在经过改造后，已经为PC/104堆叠卡所采用。图11-7给出了16位版本的扩展ISA（EISA）接口及PC/104插槽供我们参考。数字104来源于用来连接接口卡的总线引脚数。它们比常规的ATX主板要小得多，只有96毫米×96毫米，还提供额外的36根针脚，可以传输16位的数据，以及进行24位的寻址。大小的减小和堆叠方案带来的方便性，使得PC/104标准在工业和娱乐领域的开发人员中很受欢迎。因为这些领域在选择处理器和支持芯片时，常常会垂青于那些运行时功耗不大、不需要风扇的器件，以提高设备的可靠性，从源头上控制深入的噪音。

由于历史原因造成的8.33 MHz速度上限和16 MB的寻址限制，很快成为ISA标准的严重缺陷。当更快的奔腾CPU（64位总线）出现时，ISA标准对性能的影响越来越明显。如果我们考虑主总线（800 MB/s）和ISA总线（16 MB/s）的峰值数据传输速率，通过ISA总线传送大批数据的代价就会变得很明显。总线宽度不匹配的技术问题（8字节的奔腾总线必须与2甚至1字节的ISA总线通信）由总线控制器负责处理。四个BE（Byte Enable）CPU总线控制线路接适当的数据线，依次让各个字节通过，很像火车的调度室。

请检查图11-7中每个针脚的标签，找出构成地址、数据和控制总线的导线组。

新一代总线标准是EISA（Extended ISA），它允许33 MHz的传输速率。但新的PCI总线很快成为针对高性能奔腾系统设计的IO卡的最普遍选择。

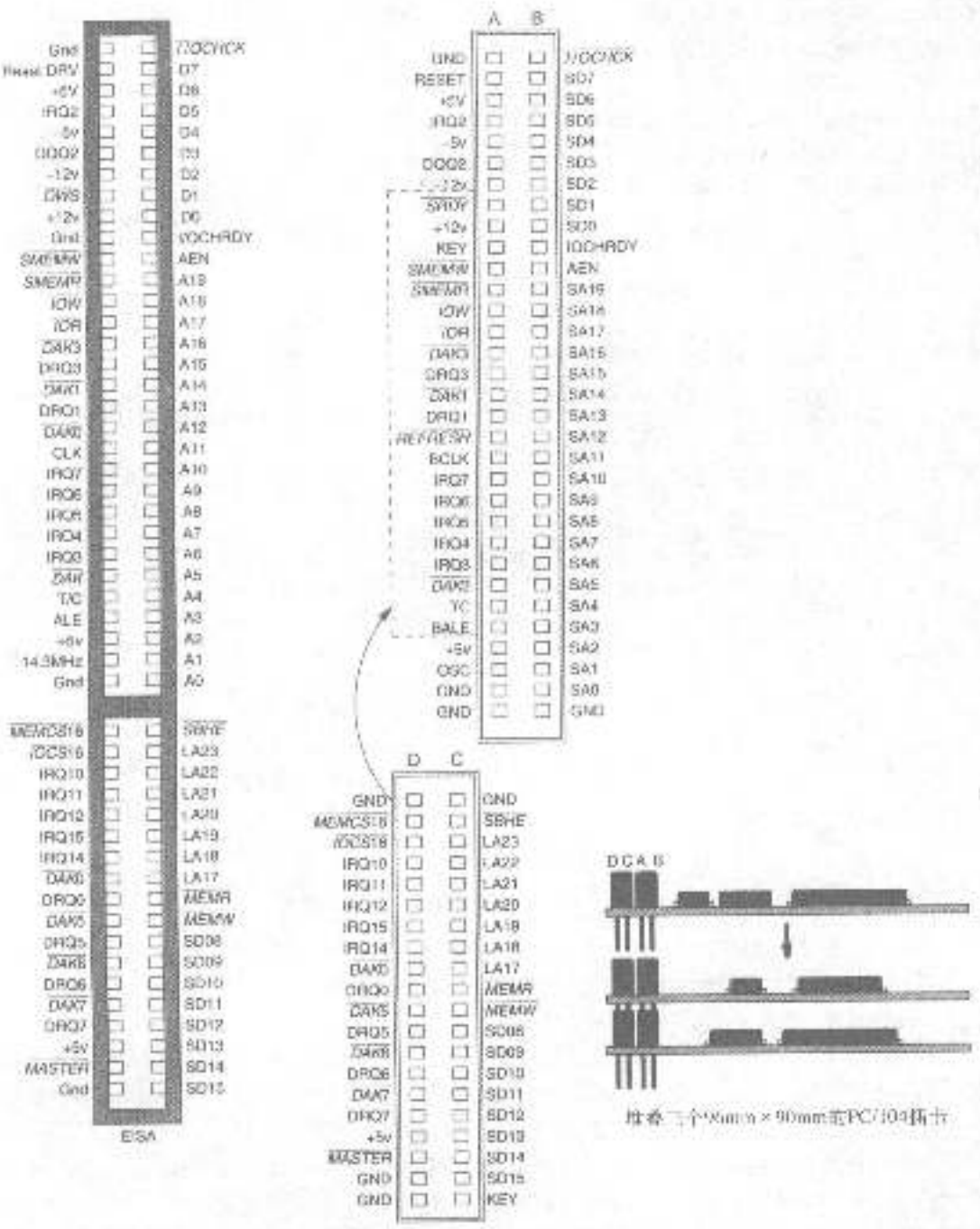


图11.7　ISA总线接插件的布局

11.6　PCI：外设部件的互连

尽管ISA总线经过增强，已经能够承载16位的数据，但是根本性的速度限制以及维护后向兼容的需要，基本上决定了它最终的命运。PCI总线最近在夺取图形卡和内置式调制解调器市场中取得了成功，加速了ISA的死亡。ISA卡现在已经是遗留产品。如图11-8所示，PCI总线通过桥接器（比如82443）从主总线分离出来，我们一般将其称为“北桥”，以将它与其他PCI到ISA桥接设备，比

如82371/PIX4（也就是我们常说的南桥），区分开来。由于PCI总线和ISA总线传输速率的不同，因而将PCI总线和ISA总线都与主系统总线隔离开来是必需的。主板使用66 MHz的系统时钟，而PCI以其一半的频率运行，但由于它的数据宽度为32位，因而它能够达到132 MB/s的传输速率，64位的版本可以达到264 MB/s。在PCI设计者加宽总线以提高存储器的同时，主板的时钟频率也从66 MHz增加到133 MHz，接下来会增长到200 MHz。和它的前辈ISA总线当时的情况一样，PCI的新标准很快就会提上议事日程。将PCI看做是"遗产"还有些为时过早。人们为便携式计算机开发出mini-PCI（微型PCI）用于附加的设备，比如无线和网络适配器。另外，PCI总线也被一些装备有复杂背板的大型计算机系统采用，以取代较老的VME系统。



图11-8 PCI桥接设备和主系统总线的关系

PCI总线有两种模式：

- **多路复用模式**：单个32位总线由地址和数据信息共享。这种模式能够增加数据的有效宽度，但会降低数据传输速率。
- **突发模式**：这也是EDO DRAM采用的技术，在地址被发送之后，几个数据项将会很快地跟进。桥接器能够对数据包进行重组，准备就绪后一次性地通过PCI总线发送出去。

如图11-9所示，PCI桥接器需要两个内部总线和4个数据缓冲区，来处理它需要同时执行的载入和卸载操作。它将PCI总线从系统总线上分离出来，对双方都有很大的好处。PCI桥接器内的两个内部总线使得系统端的活动与PCI端的活动可以分离开来。每侧的双重缓冲区允许当两边都在处理外部数据传输时，数据也可以在桥接器内部传输。

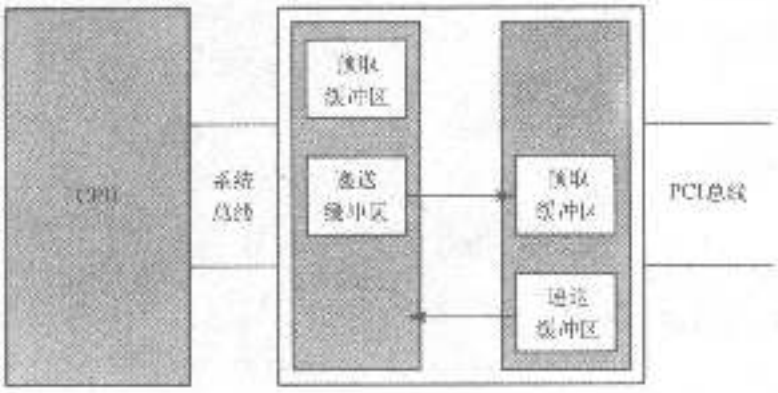


图11-9 PCI桥接器

图11-10给出了一个PCI插槽。

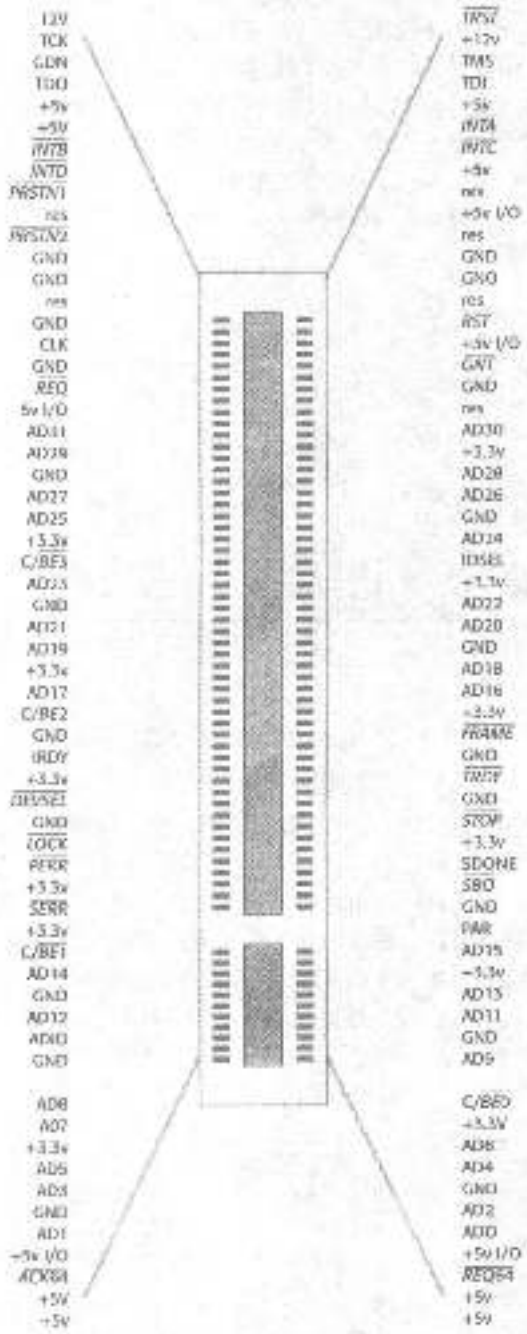


图11-10　PCI插槽

11.7　即插即用：自动配置

即插即用（Plug-and-Play）是Microsoft为简化新设备的安装和配置而引入的。它允许操作系统能够识别出新装入的设备，为其选择正确的配置参数，并安装恰当的设备驱动程序软件。另外，为

了执行启动时的查询会话，还需要专门的BIOS例程。如果采用手动配置，对于经验不太丰富的用户则是困难重重，极易使系统崩溃。即插即用方案最初和Windows 95一同发布，但Windows NT与之并不完全兼容，直到Windows 2000和XP才提供全面的支持。

对于操作系统来说，根本的困难是找出安装了哪些硬件，并获得合适的软件驱动程序来正确地访问它。遗憾的是，没有实际的规范强加在硬件提供商身上。他们可以自由地选择使用哪个端口地址和IRQ线。不同制造厂商生产的卡不兼容已司空见惯。针对这种情况，在建立供操作系统在启动时可以采用的查询协议时，面临巨大的问题。即插即用需要扩展槽上所有的插卡都以正确的方式响应一个经过编码的、写入到特定端口的初始化键。之后，这些卡开始竞争，获得配置程序的关注。每个卡都以位序列的方式输出它惟一的70位序列号。这个序列号由制造商代码（MID）和个别产品的代码（PID）组成。表11-5列出一些具体的例子。由于所有的插卡都同时争用单个总线线路，因而必定会产生混乱的竞争。一个设备输出的1可能会被邻近插卡输出的0所覆盖。如果某个设备将1输出到线路上，但读取线路状态得到0，则必须撤出竞争。如果某个设备的序列号全部由0组成（不合法），那么它将总是能够坚持到最后。如果每个卡能够顺利地通过每个清除阶段，操作系统就会询问它的相关信息，并对它进行远程配置，同时告诉它撤出即插即用的识别竞争。最终，所有的设备都被操作系统识别出来，并完成配置且正确地安装。这个过程包括为每个设备分配必需的资源，消除任何冲突。基地址、IRQ号和DMA通道必须组织得让每个卡满意——这是对计算机的一项苛刻要求！表11-6中给出一个成功的序列。我们可以将序列号的各个位想像成以从A到H的反向二进制次序依次走出，0总是战胜1。胜利者退出，之前的失败者留下来继续竞争。任何阶段的胜者都是数字中0最长的那个。在第一个1之后，又是0最长的那个获胜。当所有的参与者都输出1时，就没有设备被排除在处。

表11-6显示了15个卡按照顺序输出它们的8位序列号时的情况。A到D轮没有完成撤消，因为卡的序列号都是0，接下来E轮消去了卡8～15，因为它们对应的位为1。F轮只剩下1、2和3在争用。最后一轮G将2和3消去，1留下来做为惟一的卡，下一轮没有对手，因此它的序列数字得以正确地传输。第二轮竞争不会包括早期的胜者，结果会是卡2和卡3在G和H轮互相测验。G轮将会是平局，因为它们都提供1，但H轮卡3会被撤消。

表11-5 即插即用特征编码的例子

制造商	MID	PID
Adaptec	9004	36868
康柏	1032	4146
创新	10F6	4342
Cyrix	1078	4216
爱普生	1008	4104
惠普	103c	4156
Intel	8086	32902
Matsushita	10F7	4343
三菱	1067	4199
摩托罗拉	1057	4183
NCR	1000	4096
东芝	102F	4143
Tseng Labs	100C	4108

表11-6 即插即用序列

插卡	序列号	插卡	序列号
	ABCDEFGH		ABCDEFGH
1	00000001	9	00001001
2	00000010	10	00001010
3	00000011	11	00001011
4	00000100	12	00001100
5	00000101	13	00001101
6	00000110	14	00001011
7	00000111	15	00001111
8	00001000		

11.8 PCMCIA：个人计算机存储卡国际联盟

如果您曾经参加过涉及PCMCIA卡的会谈，一定会高兴地知道，这个名字已经缩写成PC卡。

这个标准描述的是PC接口、总线和卡大小，最初它的目的只是为便携式个人计算机提供扩展存储模组。这些卡大小和信用卡相同（85.6mm×54mm），一侧是微型的68针连接器。现在，该格式已经被

制造厂商采用，提供各种各样的IO功能。在相对的一侧是另一个连接器。现在我们可以买到PCMCIA调制解调器、并行端口、以太网接口、ISDN端口、视频会议系统、硬盘驱动器，等等。为了满足多样化的需求，PC卡有三种不同的标准，其厚度不断增加，从3.3、5.5到10.5mm，其中所含的电路也越来越多，但长和宽保持相同。由于连接器采用保护相同，因此薄的卡可以插入到厚的插座中，但相反则不行。这个接口允许"热交换"：移去或插入卡无需关闭电源，甚至无需重启计算机。

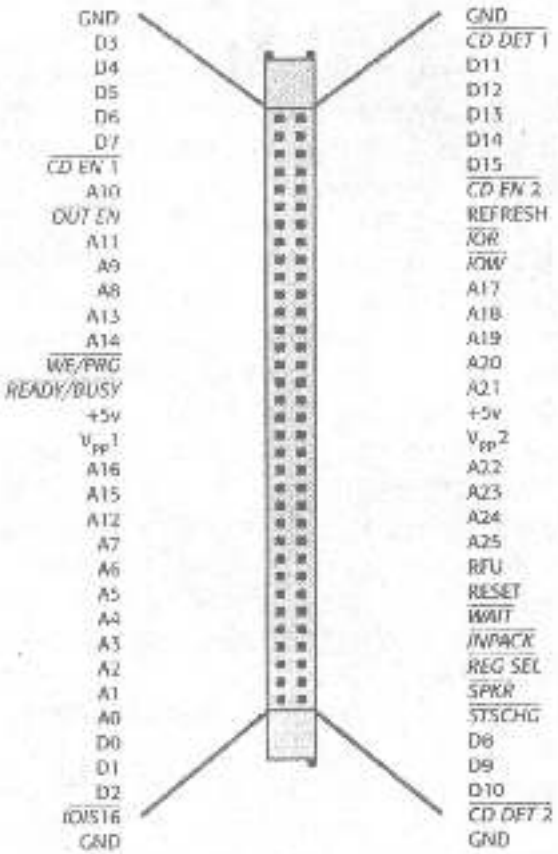


图11-11　PCMCIA接口

卡上的地址总线是26位宽，因此寻址范围为64 MB。地址总线为16位，对存储器和IO端口的寻址不能直接完成。所有卡内的寻址都由PCIC控制器（如i82365SL）处理。它使用PCI总线，负责将PC卡提供的地址范围映射到主存或IO地址中。这是一种有限的虚拟内存管理。被映射的区域称为"窗口"，映射窗口的起始地址和结束地址由PCMCIA控制寄存器保存。

在使用时，PCMCIA卡插入一个特殊的插座内，这个插座就如同一个IO单元，不管插入的是PCMCIA还是其他硬件，卡插入后，IO地址、IRQ、内存映射地址和DMA（如果需要的话）都会自动完成配置。

在PCMCIA卡插入插座时（见图11-11），PCIC控制器使用CD DET输入线，能够检测出它的存在。Vpp线用于对闪存卡编程。如果PCMCIA卡有写保护开关，那么它的状态通过IOIS16输入读取。令人惊奇的是，SPKR引脚允许卡直接连接到音频系统！

11.9　小结

- 计算机内的硬件使用并行总线通信。总线是一束束的导线，数据总线常见的宽度为8位、16位、32位、64位，也有可能为其他数字。总线越宽，吞吐量（或带宽）越大。
- 外部设备需要接口电路将它们的信号转换成系统总线上的信号。
- Centronics端口是含有异步握手控制电路的8位并行接口。Centronics打印机端口现有三种标准。尽管将它称为总线，但它只适合于点对点通信。
- SCSI总线最多可以连接8个设备。PC机需要安装SCSI PCI/ISA接口卡。SCSI既可以工作在异步模式，也可以工作在同步模式。它是为磁盘或磁带设备设计的，但也可以用于任何装备有SCSI接口的设备。它的优势是运转快，有内部数据缓冲，且能够通过重叠事务同时活动。可以对事务排队，提供专门的DMA控制器等。
- IDE磁盘驱动器内部包括控制电路。PC一般最多可以安装4个IDE驱动器，它们成对地连接到主板的两个插座上。
- 16位ISA总线由原来PC机中的扩展卡插槽发展而来。它以同步模式运行在8.33 MHz时钟频率下，在处理慢速设备时，需要插入等待状态。
- 32位PCI运行的频率是主板频率的一半，带缓冲的入口芯片将它与主总线隔离开来。现在，它是PC机的首选扩展总线。
- 即插即用是管理自配置硬件的一种方案。要做到即插即用，需要在启动时执行专门的BIOS例

程。操作系统可以在启动时自动识别出新设备，并安装正确的软件驱动程序。

- 扩展卡的PCMCIA（PC卡）标准是为便携式计算机设计的，但在移动设备领域有广泛的应用。

实习作业

由实验教师在课堂上分配课程作业，并演示代码。

练习

1. 将下面的Centronics打印机端口信号按照起源的不同分成两组，一组来源于计算机，一组来源于打印机：STROBE、ERROR、SELECT、AUTO_FEED、OUT_OF_PAPER、BUSY、SELECT_INPUT、INT、ACK。
   其中，哪些是基本的？描述字节传送时的握手信息交换。
2. Centronics接口在哪些方面优于串行COM端口？
3. Ultra-SCSI标准提供16位宽的数据总线，最大能够达到20 MHz。它可以连接多少个设备呢？SCSI子系统如何同时执行多个事务呢？
4. Centronics提供传输时的错误检测吗？一个Centronics端口可以连接多少设备？Centronics接口能够和16位的Unicode一起工作吗？
5. SCSI总线可以连接多种不同的设备。您认为SCSI总线是同步的还是异步的？使用SCSI能够达到什么样的传输速率？
6. IDE和SCSI之间的区别是什么？为什么在共享同一主板总线时，我们依旧认为SCSI驱动器要快些？
7. 业界对AT/ISA总线的最大时钟频率已经达成共识，是多少？将一幅SVGA屏幕图像通过ISA总线传送，要花多长时间？
8. PCI桥接器是什么？它们通过哪种方式加速数据的传输？
9. 声霸卡以44.1 KHz的速率交付一对对的16位字。慢速的ISA总线能够满足要求吗？
10. 解释ISA和PCI总线的中断结构。
11. 人们不断将光纤应用到之前并行总线的应用领域，是不是有迹象表明，宽而笨重的总线的好时光已经过去了呢？
12. 在选定正确的驱动程序例程之前，即插即用如何确定新硬件的存在呢？

课外读物

- 对本章主题的简要介绍以及到其他网站的一些链接：
  http://www.webopedia.com/
- 嵌入式系统中的PC104格式：
  http://www.pc104.com/whatisit
- 下面这份资料介绍PC/104，它很不错，可以下载下来阅读：
  http://www.bluechiptechnology.co.uk/products/Single_Board_Computers/PC104_Spec_v2.5.pdf
- Centronics操作模式的简要汇总：
  http://whatis.com/eppecp.htm
- Tom硬件指南提供很多不错的基本和最新的市场资讯：
  http://www7.tomshardware.com/storage/97q3/970728/hdd-06.html
- 一些包括SCSI和IDE资料的指导性材料：
  http://www.hardwarecentral.com/hardwarecentral/tutorials/
- 从下面的网址，可以找到对PCMCIA标准更全面的讨论：
  http://www.pc-card.com/pccardstandard.htm
- 这些网站可以通过本书的配套网站访问：
  http://www.pearsoned.co.uk/williams

第12章 存储体系

各种不同的存储设备——CD-ROM、硬盘、DRAM主存、SRAM高速缓存、CPU高速缓存和CPU寄存器的均衡配比，有助于提高系统的吞吐量。任何时候，数据和指令都是在这个存储体系中向上或向下移动，以满足上层的需求，最终为CPU的运算提供必要的支持。这种方案所基于的基本情况是，程序一般都顺序执行，所访问的数据具有局部性（locality），由于这种特征，我们可以将指令和数据成块地在存储体系中向上或向下移动，以提高效率。

12.1 系统的性能

我们可以为CPU提供足够的主存，完全占满它全部的地址空间（由地址总线的宽度决定），我们也可以选取执行读写操作的速度与系统时钟相匹配的内存。这就意味着2 GHz的奔腾个人计算机将会安装4 GB（2^{32}字节）的DRAM芯片，并且这些芯片的访问延迟都达到0.5ns的级别（见图12-1），这当然还不太现实，并且它也不是一种高效的解决方案。我们最好记住下面这个事实：当一种CPU系列达到它商业生命的终点时，主存才会完全占满，并且这样对系统的性能不会有什么真正的提高。软件的需求常常早就超过现有的内存，必须采用其他方法以满足对内存的需求。此时，为了优化性能，同时将成本压缩到合理的水平，许多计算机都采用存储设备的分层体系，力图跟上更快速的CPU的需求，同时应对日益庞大的软件。

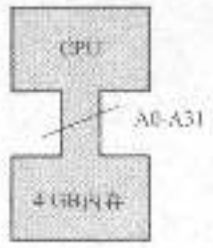


图12-1 完全占满的主存

存储体系（见图12-2）的顶部是最快速的CPU寄存器，底部是最慢的磁带驱动器，中间为磁盘、DRAM主存和SRAM高速缓存。主存提供比CPU寄存器多得多的存储空间，因为相对于DRAM，在CPU芯片中提供存储空间的代价极为高昂。每种类型存储设备的相对价格，事实上决定了它们实际提供的数量。因而，我们可以将这个体系看做是一个金字塔，顶部窄，底部宽。

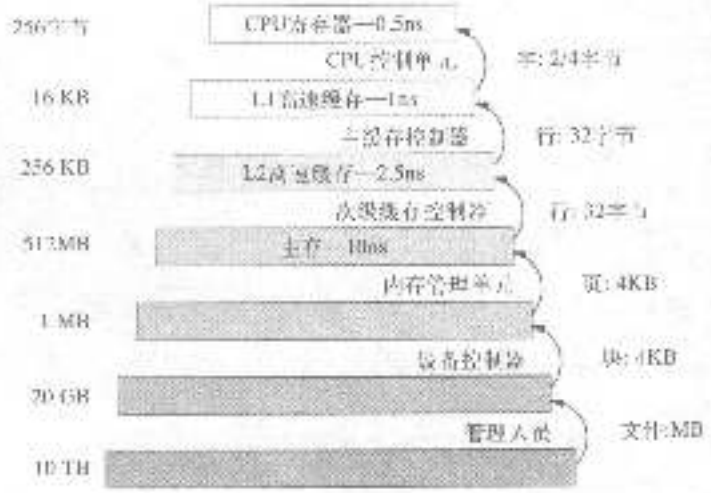


图12-2 存储器性能和存储体系

但是，比较在各种类型的存储设备上存储数据的实际成本，是一个复杂的问题，因为还需要考

虑许多隐性成本。电力供给、散热风扇、控制芯片、驱动软件、维护合同甚至所占用的桌面空间，都会对成本造成影响。放弃维护老旧的空调设备所节省下的费用，或许能够完成许多次磁盘升级！类似地，在市政中心的办公室内，移走不再需要的橱柜节省出空间后，会计师才会考虑投资文档扫描器和CD-RW设备。

各种存储设备的选择和使用依赖于好几种因素。成本和执行的速度显然是至关重要的，但电力耗费、存储的稳定性、物理大小、使用的方便性、操作的安全性，以及其他非重要因素，比如风扇噪声，也很重要。表12-1列出了各种设备的大致价格。最后一栏列出了我们常常最感兴趣的数字：每字节成本。

表12-1 存储设备价格表

类型	大小	设备	单位成本（英镑）	每兆成本（英镑）
DRAM	512 M	SDRAM 168针DIMM	25	0.05
SRAM	512 K/5 ns	SRAM芯片	2	64
SCSI	PCI卡		180	
硬盘	120 G	IDE	80	6×10^{-4}
	10 G	SCSI	250	0.025
CD-ROM	650 M	32 × IDE	15	0.03^{R}
CD-RW	650 M	2 × IDE	60	0.12^{R}
DVD	8.5 G	24 × IDE	90	1×10^{-8}
	WORM	磁盘	0.75	
	RW	磁盘	2.50	
Jaz	1 G	驱动器	200	
	1 G	磁盘	65	0.25^{R}
Zip	100 M	驱动器	60	
	100 M	磁盘	15	0.75^{R}
磁带机	4 G	SCSI驱动器	350	
	4 G	磁带	2.5	0.09^{R}
软盘	1.4 M	驱动器	15	
	1.4 M	磁盘	0.5	11^{R}
USB	512 M	闪存	35	0.07

注：表中容量均为字节。
R＝可移动介质。

如果只考虑**访问速度**，我们可以将所有的存储设备组织到一个**性能体系**中。在需要时，数据在相邻的层间传输。有趣的是，层间数据传输的单位大小并非完全一致（参见图12-2），这也反映出开始数据传输所需的启动时间的差异。同样，各个层之间的数据传输并非集中控制，而是处于三类独立单元的监控之下，它们以完全不同的体制运行。这又是一个例子，说明硬件部件和系统软件的紧密配合，能够在性能和商业可用性上取得革命性的突破。存储体系如同供给流水线，将数据和代码块排成队列向CPU传输，直至将它们载入到L1缓存内，可以供CU访问为止。管理这些传输活动的根本目的，是尽可能快地响应对数据的请求。所传输数据的大小常常大于请求的大小，这是基于附近的存储单元随后常常会被访问这种预期而做出的。这是**预取**（pre-fetching）的一个实例，它的意图是预测对数据的需求，并提前做好准备。

12.2 访问局部化：利用重复

我们可以想像得到，计算机的一种重要特性是，在一段时间内对内存的访问往往局限在相对有限的范围之内，即局部化（locality）。这来源于编程语言提供的基本结构：SEQ、IT和SEL（参见第13章），人们已经认识到程序执行的这个非常有用的特性，并利用它来增强计算机的性能。计算机

常常集中处理特定区域的数据并会多次访问相同代码段，这一特性部分归因于程序员将相关的数据项存储在数组或记录中，部分归因于编译器试图以更高效的方式组织代码。不管原因如何，在构建存储体系时，可以利用内存访问的局部化这一特征。代码和数据中实际为CPU所用的部分应该载入到最靠近CPU、同时也最快的存储设备中。程序和数据的其他部分，可以保存在存储体系中访问速度较慢的地方。

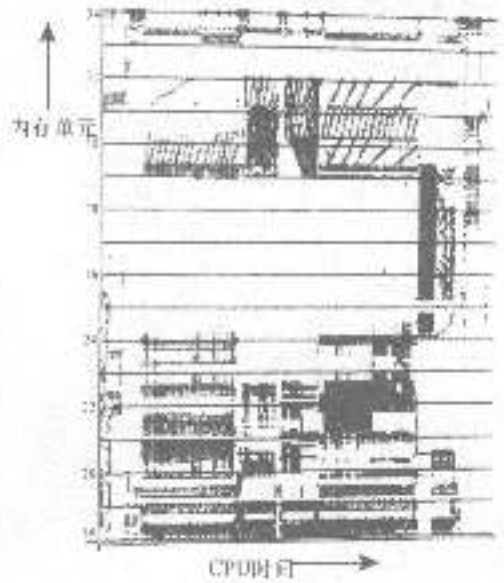


图12-3　内存访问图（展示局部性效应，来自于Hatfield D.和Gerald J.（1971），IBM Systems Journal，10（3））

图12-3是一幅实际的图，它表示在一段时间内，任一时刻CPU访问哪个内存地址。它显然不是随机分布，而是体现出访问的局部性，其中不连贯的变化可能来源于需要特定服务时对子例程的调用，比如输入或输出。如果知道特定子例程载入到内存中的什么地方，就能够通过研究这种图来分析它们的活动。

程序员如果希望避免性能上的波动，则要考虑存储体系方面的问题。查看存储介质的相对访问时间，很显然，任何对磁盘的访问都应该是尽量避免的。在繁忙的周一早晨，让负责计算机的工作人员安装特殊的数据磁带，有可能得不到快速的响应！总之，在使用有可能突发地访问文件的数据处理应用程序时，程序员会试图在程序开始时，将完整的输入文件都读入到数组中，以提高运行时的性能。类似地，输出数据也可以暂时组织到数组中，最后在程序结束时写出到磁盘上。这种明确的策略很不错，但底层操作系统的内存管理操作依旧有可能会影响甚至破坏我们的计划。如果数据文件很大，那么磁盘管理器肯定会将它移出到磁盘上的交换区域。尽管这是一种透明的操作，能为程序员提供方便，但它无疑降低了精心选择、基于数组的算法所能够带来的速度优势。

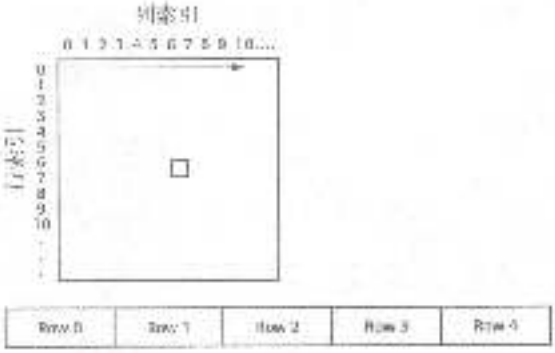


图12-4　数组检索及数组数据在内存中的布局

程序员还可以以内存管理活动最小化为目标来安排数据和设计代码。为了演示这种做法，请编译并运行图12-5给出的C程序，这段程序不过是将1000×1000二维整型数组的所有元素累加起来。分析两部分所花费的运行时间，我们会发现，第一部分（一行行地访问数组）要比第二部分（一列列地访问数组，其他不变）快一倍。图12-4说明了数组或矩阵在内存中的存储方式。

我的Sun工作站有256 MB主存，256 KB高速缓存和4 GB本地磁盘。测试程序的运行结果在图12-6中给出。我还以相反的次序重复这两种情形，以检查次序对结果的影响。结果发现没有什么差异。函数times()给出处理过程运行多少个系统时钟单位。我的系统每秒产生100个时钟信号。

图12-6给出的两个测试循环的运行时间分别为150和310；数据访问的小的变动就引起相当大的差异。这150ms的时间来自于哪里呢？两个嵌套的for(;;)循环执行1百万次，这意味着每次工作站都要花费140ns。通过获得这段代码对应的汇编语言代码，我们可以检查循环内实际含有的机器指令数。检查图12-7中列出的代码，我们可以看到在内层循环有21条指令，外层有另外13条指令。假

定我们使用200 MHz的CPU，则每个指令周期为5ns。

$21 \times 5\text{ns} \times 10^6 + 13 \times 5\text{ns} \times 10^3 = 105\text{ms}$

```
//Linux version of cache.c to run on a 1 GHz cpu
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>   // or <limits.h>
#define MAX 10000

clock_t times(struct tms* b);

main () {
int i, j;
int big[MAX][MAX];
int sum, start, middle, end, hz;
struct tms tbuff;

  hz = sysconf(_SC_CLK_TCK);
  printf("system ticks per sec = %d\n", hz);

  times( &tbuff); start = tbuff.tms_utime;
  for(i=0; i<MAX; i++) {
        for(j=0; j<MAX; j++) {
          sum += big[i][j];  /* <------------ i, j  here */
        };
   };
  times( &tbuff); middle = tbuff.tms_utime;
  for(i=0; i<MAX; i++) {
        for(j=0; j<MAX; j++) {
          sum += big[j][i];  /* <------------ j, i  here */
        };
   };
  times( &tbuff); end = tbuff.tms_utime;
  printf("First run time is %d \n",(middle - start)/hz);
  printf("Second run time is %d \n",(end - middle)/hz);
}
```

图12-5 高速缓存动作的演示

```
rwilliam@olveston [80] cc cache.c -o cache
rwilliam@olveston [81] cache
First run time is 150
Second run time is 310
rwilliam@olveston [82]
```

图12-6 高速缓存测试的结果

这与实际的结果尚有45ms的差距，可能是由于调度活动引起的（进程的暂停与恢复）。但是，同样数量的指令每次以不同的次序执行时，310ms和150ms之间的巨大差异仍然存在。要注意到的是，执行代码文件的大小为23340字节，这个大小会占据6个虚拟内存页面或730个缓存行。

这种100%的差异来自于对4 MB的数据数组的访问方式的差异。尽管64 MB的主存足够保存这个数量的数据，因而不会发生磁盘交换，但256 KB的缓存显然不够存下这么多数据。在顺序访问数组中的数据时，如cache.c中的第一部分，存储管理系统在组织数据缓存时没有困难。然而，第二个版本在访问内存时则需要采取较大的动作，会导致数据在存储体系中向上或向下移动。参见图12-8中给出的示意图。

```
! 19   for(i=0; i<MAX; i++) {
              mov 0,%10
              st %10,[%fp-8]
              ld [%fp-8],%10
              cmp %10,1000
              bge .L118
              nop
              ! block 2
  .L119:
  .L116
! 20   for(j=0; j<MAX; j++) {
              mov 0,%10
              st %10,[%fp-12]
              ld [%fp-12],%10
              cmp %10,1000
              bge .L122
              nop
              ! block 3
  .L123:
  .L120
! 21   sum += big[i][j];
              sethi %hi(-4000016),%10
              or %10,%lo(-4000016),%10
              ld [%fp+%10],%14
              sethi %hi(-4000012),%10
              or %10,%lo(-4000012),%10
              add %fp,%10,%13
              ld [%fp-8],%10
              sll %10,12,%12
              sll %10,5,%11
              sub %12,%11,%12
              sll %10,%6,%11
```

```
! 19   for(i=0; i<MAX; i++) {
              sub %12,%11,%11
              add %13,%11,%12
              ld [%fp-12],%10
              sll %10,2,%11
              add %12,%11,%10
              ld [%10+0],%11
              add %14,%11,%11
              sethi %hi(-4000016),%10
              or %10,%lo(-4000016),%10
              st %11,[%fp+%10]
! 22   };
              ld [%fp-12],%10
              add %10,1,%10
              st %10,[%fp-12]
              ld [%fp-12],%10
              cmp %10,1000
              bl .L120
              nop
              ! block 4
  .L124:
  .L122:
! 23   };
              ld [%fp-8],%10
              add %10,1,%10
              st %10,[%fp-8]
              ld [%fp-8],%10
              cmp %10,1000
              bl .L116
              nop
              ! block 5
  .L125;
  .L118;
```

图12-7　SPARC编译器编译图12-5中的cache.c产生的（部分）汇编代码

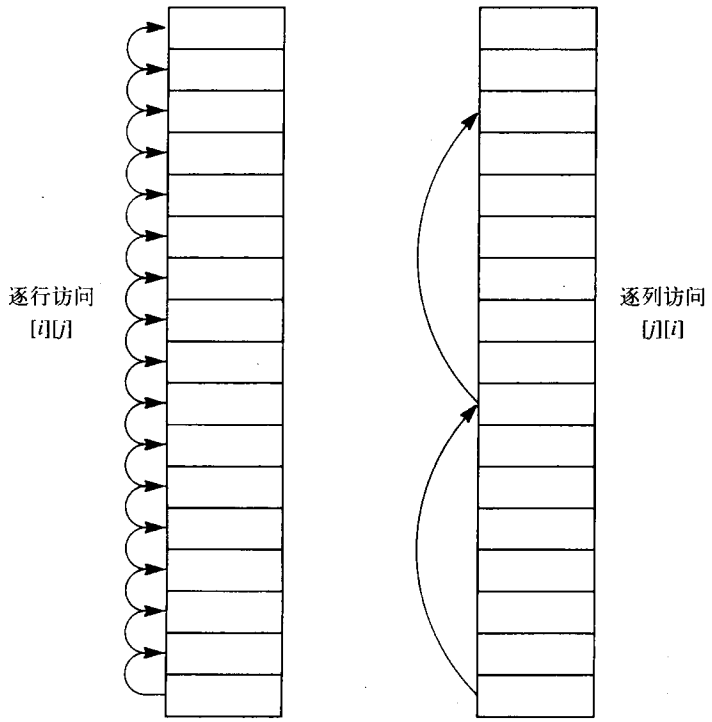


图12-8　对二维数组的不同访问模式

12.3 指令及数据的高速缓存：匹配内存和CPU的速度

考虑时钟频率为2 GHz的奔腾CPU。它平均每0.5ns就需要访问程序所在的内存，读取指令，如果将数据变量的访问需要考虑在内，则还会更频繁。DDO DRAM只能在10ns时间内做出响应。因而这里存在一个严重的速度不均衡问题，需要我们解决！如果CPU只用DRAM来存储程序代码，那么整个系统的运行速度将会慢20倍。部件性能的不匹配是现代计算机中许多复杂问题的根源。

在过去的50年间，CPU和存储设备的相对速度早跷跷板式地发展。在特定的时间内，存储器能够应对更快的CPU，其他时候CPU走在存储器的前面。当前，奔腾处理器大约运行在3 GHz或更高，远远走在了内存的前面。这使得我们不得不采用一些巧妙的设计，避免将CPU的大部分时间浪费在等待内存响应它的请求上。这也是存储体系中引入高速缓存后的原因。由于高速缓存（cache）是由更昂贵的SRAM芯片构建而成，SRAM芯片的处理速度比DRAM要快。因而高速缓存能够提供更快的响应速度。理想情况下，缓存的访问速度应该和系统时钟速度一致。这样CPU在访问数据和指令时，就不用额外地等待。但实际上大多数CPU依旧需要等待慢速设备完成一项操作，因此CPU的总线控制器能够在任何读-写总线周期内插入等待状态（Wait State），以处理这种情况。

高速缓存和它的CCU（Cache Controller Unit，高速缓存控制单元），被安排在CPU和主存之间（见图12-9），这样它就能够截获CPU发出的任何内存访问请求。这样做的目标是，在快速的高速缓存中维护当前活动的代码和数据块。所有来自于CPU的内存读写请求，都被重定向到高速缓存中，以期能够获得快速的响应。CCU检查内存地址，查看所请求的数据是否当前就在缓存中。如果在，CPU就能够立即得到响应，否则控制器就会从主存中读入所请求的数据项，这会造成一定的时间延迟。在程序执行较少指令的循环时，所有的指令都能够装入高速缓存中。这样做显然会很好地加速程序的运行。但是，如果高速缓存控制器能够预读，并能预测CPU的需求，及时更新高速缓存中的数据时，对于顺序访问的线性代码，它也可以工作得很好。一些编译器通过插入特殊的指令，激发CCU完成这项功能。

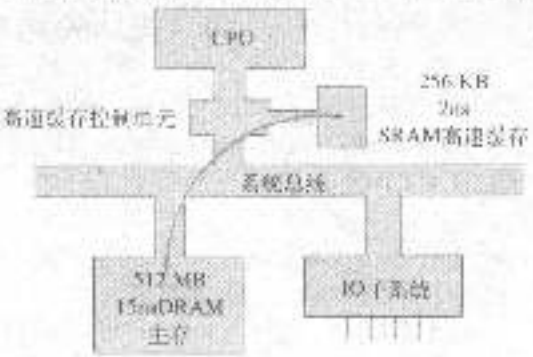


图12-9 高速缓存存储器和控制单元

在需要将新的数据项引入到高速缓存中时，常常必须决定应该牺牲哪些现有的数据项，为新的数据提供空间。如果驻留高速缓存中的数据被更改过，则还要将更改过的数据写回到主存中，这会导致进一步的延迟。

L1和L2高速缓存之间的差别，主要在于它们的位置。最快的高速缓存最接近CPU。现在它已经被集成到CPU芯片上。这部分高速缓存称做主高速缓存或L1高速缓存。在奔腾4中，也称做执行跟踪高速缓存（Execution Trace Cache）。由于其位置特殊，所以可以获得极大的速度优势，但由于制造工艺上的限制，其大小受到局限。奔腾II处理器共有16 KB的L1高速缓存，Intel将它一分为二，分别用于数据和指令：高速缓存D和高速缓存I。这种存储器异化的方法不适合冯·诺依曼的理论，但它允许计算机在预取指令的同时访问变量。使用流水线译码器时，如7.7节所述，系统需要提供双总线，以防止两个或更多流水线阶段需要在同一时刻传输数据项时总线争用的发生。这种做法在技术上仅对芯片内的高速缓存可行，芯片外高速缓存不能采用这种做法。

如果L2高速缓存不在芯片内，而是位于主板上，则它的大小完全根据需要而定。但由于距离CPU较远，它的运行速度要慢一些，因为芯片发出信号和信号通过系统总线的传输都会引入延迟。为了改善这种情况，Intel引入奔腾II处理器，它将L2高速缓存分离出来，和CPU一同构建在Slot 1/Slot 2封装内。在CPU和L2高速缓存间有一条专用的局部总线，运行在200 MHz，比主板上的100 MHz系统总线要快许多。

如7.1节所述，Intel现已成功地在CPU芯片上集成128 KB的L2高速缓存，和L1高速缓存放在一起，以同样的速度运行。当前的奔腾芯片继续提供这样的集成L2高速缓存。也许我们应该考虑一下，是不是应该将L1和L2集成为统一的缓冲器，但是，随着高速缓存存储器大小的增长，访问机制的效率会下降，表现为访问速度减慢，复杂度上升，因此，分层次的高速缓存体系是一种更佳的技术解决方案。

值得注意的是，图12-2中，高速缓存和主存之间进行数据传输的单元称为**行**（line），其长度可以达到256字节，尽管奔腾处理器使用32字节的行（等于四个总线周期，每周期8字节）。这和EDO和DDR DRAM配合得很好，因为它们能够快速地访问连续的四个存储单元（见6.6节）。行在存储体系中的移动速度对于提高系统的性能十分重要。采用宽总线以及以行为单位进行**总线突发传输**（bus burst），可以使系统跟上CPU的步伐。我们可以将数据在各种存储部件之间的传送理解为公路咖啡馆风格的服务。一组组的观光者乘坐大客车以15分钟为间隔不断到达，排队等待咖啡或卫生间。尽管观光者每隔15分钟来一次，但这些设施的利用率均为100%。这类系统的正常运转依赖于让顾客排队等待，并让处理速度足够快以避免拥塞。对于CPU，缓冲数据和指令不过是提供必需的寄存器，处理速度的问题则通过提供16个RAM芯片的存储器（可以同步地访问64位总线上指定的位）来解决。

12.4　高速缓存映射

SRAM高速缓存的大小相对于主DRAM内存要小得多，在使用时就会出现这样一个问题（在计算机系统的其他部分也存在类似问题）：我们如何将较大的内存空间映射到较小的工作空间呢？尽管理论上，我们需要将4 G的地址空间映射到512 KB的L2高速缓存，进而映射到16 KB的L1高速缓存，但实际上，一般PC内仅配置256 MBDRAM。下一代采用64位CPU的计算机中，高速缓存与主存在大小上的差异还会进一步拉大。解决这种“多对少”的地址映射问题有多种方案，表12-2列出了其中的三种。

表12-2　主存到高速缓存的映射

1. 折叠地址空间，也称为直接映射
2. 关联（内容可寻址）内存
3. 散列映射

列出散列法仅仅出于完整性的考虑。由于时间延迟问题，高速缓存地址映射中并不使用这种方法，但在虚拟内存页面映射中，它被用做类似的用途。我们将在12.5节中讨论这一主题。

管理缓存的**直接映射**（Direct Mapping）技术将主存划分成高速缓存大小的页面。每个页面进一步划分成**行**（line）。以奔腾处理器的L2高速缓存为例，其地址为32位，大小为512 K，每行32字节，地址可以划分成三段。低端的5位（A0～A4）定位行（32字节）内的字节。接下来的14位（A5～A18）定位高速缓存内的行，余下的13位（A19～A31）指定行来自于主存中哪个页面。图12-10给出一个地址宽度为10位，8行高速缓存的简单例子。

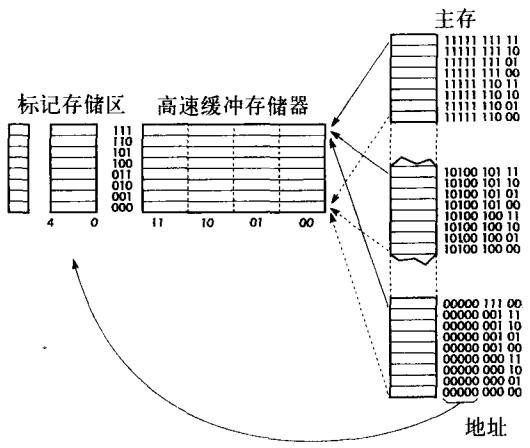


图12-10　高速缓存直接映射中的地址折叠

每一行通过地址位A5～A18映射到高速缓存中的**区段**（slot）。将新的行复制到高速缓存中对应的区段时，地址的高5位被写入到**标记**（Tag）存储区中相关联的位置。这样，当高速缓存控制器访问高速缓存中的区段时，它还必须检查标记存储区中的相关页面编号，以确定行来自于主存的哪个页面。

在CPU发出地址时，高速缓存控制器使用地址的中间部分确定应该访问高速缓存中的哪个区段，它比较地址的高位部分与标记存储区内的相关条目，使用最低符号位端的位选出请求的数据字。

只要当前程序不访问映射到相同区段的来自于其他页面的行内的数据，这种方案就能够工作。但区段冲突（两个活动的行映射到同一区段）的风险是不可避免的，一旦发生，会导致性能上的巨大下降。

在更为实际的系统中，每0.5 MB内存页面含有16 384个32字节的行。64 M内存则会有128个行映射到高速缓存中同一区段。区段冲突的可能性依赖于程序的大小和数据的结构。

另一种方案是**关联映射**（Associative Mapping），见图12-11。这种方案中，主存中的数据行可以复制到高速缓存中的任意区段，主存也不再划分成页，只有行。还是以前面例子中的10位地址和32字节高速缓存行为例，标记存储区此时必须为8位宽。

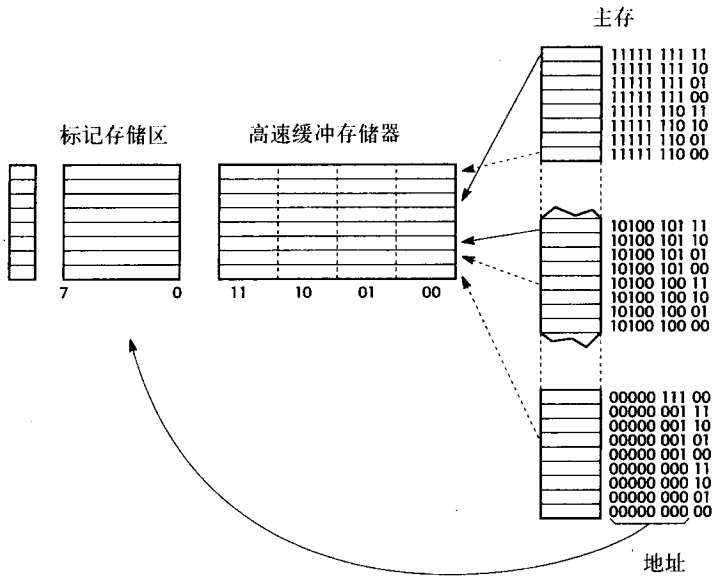


图12-11 关联高速缓存

同样，在新的行进入高速缓存中时，行的主存地址的高位被写入到标记存储区中。但现在高速缓存控制器必须检查标记存储区内所有的数据项（8192），以找出正确的区段。这样的搜索，如果顺序执行，肯定会极为缓慢，因而关联高速缓存系统提供许多额外的硬件来完成这种地址检查。参见图12-12中的XOR门。

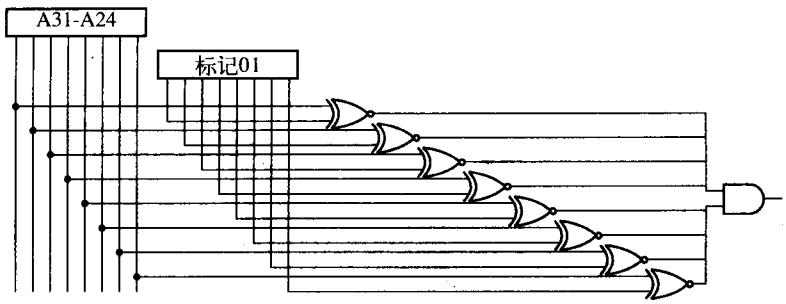


图12-12 检查关联存储器中的地址

8个XOR门组成的阵列决定输入地址的高端位是否与标记寄存器中的值相匹配。每个高速缓存区段都有自己的比较电路，从而，所有这些检查工作可以同时进行。尽管这样做快速且简洁，但电路开销很大。实践中经常使用的是这两项技术的结合——**块集关联映射**（Block-Set-Associative），奔腾处理器即使用这种方式。块集关联映射允许多行数据占有同一采用直接映射的高速缓存区段，它为高速缓存控制器提供两个或多个高速缓存设备。对高区段冲突的机会得到显著的降低。高端地址位将应用程序导向"扩展区段"，在那里进行十分简单的比较，确定两个标记寄存器中哪个与输入的值匹配。

所有高速缓存方案都需要解决的一个共同问题是，在使用数据的副本时，如何在写操作后维护**数据的一致性**（data consistency）。一种策略是**高速缓存写穿透**（Cache Write-Through）。每次改写高速缓存中的数据项时，同时更新主存中的初始版本。这会导致一定的时间延迟。另一种方式是**高速缓存写返回**（Cache Write-Back）。在需要将改写过的行交换出高速缓存时，才将其写入到主存。这种回写仅当标志位标示该行在载入到高速缓存后被修改过才会执行。因此，必须为高速缓存中的每一行提供一个**脏标志位**（dirty bit flag）。在数据写入后置位，在新行载入时清除。

另一项影响系统性能的策略性决定是，在高速缓存已满的情况下，高速缓存控制器发出没有命中的信号时应该怎么办。**高速缓存没有命中**（cache miss）发生在所请求的地址当前不在高速缓存中的情况下。这种情况下，显然必须牺牲一些东西。随机的选择就拥有效率，但LRU（Least Recently Used，近期最少使用）方法（在前一段时间内访问次数最少的行被替换）听像更为合理！一般地，缓存未命中在三种情况下发生：程序第一次运行，高速缓存中当前没有所请求的行；以及发生直接的映射冲突（见表12-3）。

表12-3　高速缓存访问未命中的原因

1. 启动新的程序
2. 高速缓存大小，不足以保存当前实执行指令
3. 直接映射式高速缓存内，高速缓存行之间发生冲突

确定高速缓存的合适大小是一项复杂的事情，依赖于所运行的应用程序以及存储体系中其他存储部件的相对性能而定。使用在线性能监视程序获取一些有关什么地方什么时候发生高速缓存未命中的数据，常常会比直接预测所要简单一些。

您可能会觉得，要进一步提高高速缓存的命中率，可行的技术会是在实际的请求发生之前，将数据预先载入到高速缓存中，这样可以使CPU运行得更平稳。我们已经在实现这项技术上取得了一些进展，一般是简单地将当前指令所在行后面的行载入到高速缓存中。这项技术所基于的预测是代码序列的执行点不断向前。遗憾的是，情况并非总是如此，所以接下来所请求的行被替换的情况总是会发生！

12.5　虚拟内存：分段和按需页面调度

虚拟内存（virtual memory）系统是在主存相当昂贵、稀少且速度不快的情况下发展起来的。通过将数据和程序代码交换到磁盘上，系统的主存得以扩展（见图12-13）。磁盘上为这一目的专门保留的区域称为**交换空间**（Swap Space），由操作系统负责管理。应用程序的程序员一般不需要关心具体的实现，只是简单地将它全部交由虚拟内存管理服务。

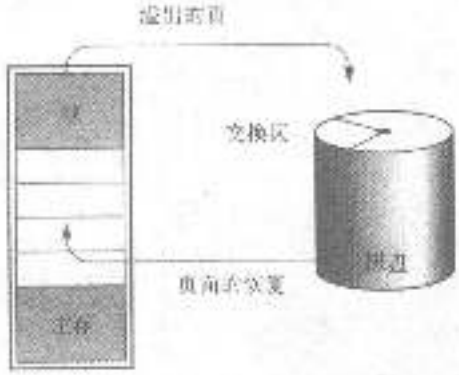


图12-13　主存不够时采用的虚拟内存管理方案

在这项技术被广泛采用之前，当代码增长得过大，不能装入主存中时，惟一的选择就是将它分成几段，链接起来。首先将第一段从磁盘载入到主存中，并开始执行。在它结束后，它会载入第二段，并将控制权转交给第二段程序。因此，每段代码都需要在结束后载入下一

段代码。应用程序的程序员负责组织这项基本的内存管理方案。

磁盘存储部件现在已经成为商业数据处理应用程序的最基本的外部设备。另外，许多工业和个人计算机系统还使用硬盘作为辅助性的存储设备，因为它具有良好的性价比。尽管大型半导体存储设备的性能取得了快速的发展，但磁盘制造厂商依旧在每字节价格上保持领先。

现代具有虚拟内存管理功能的多用户操作系统为每个用户提供独立的空间，每个用户在编写程序时感觉自己是系统的惟一用户，独享整个系统。最初，多用户操作系统简单地在需要时将整个程序交换到内存中，或交换出去。这种资源共享的方式十分耗时而且效率低下。随后出现的方案是将程序组织到**段**（segment）中，类似于依旧在Intel x86系列中出现的代码段、数据段和堆栈段。将段（而非整个程序）交换到内存中或交换出内存的可行性，降低开销，同时维护单用户的假象。段可以位于内存中的任何位置，因为它们是由专门的**段基地址寄存器**（Segment Base Register）来定位的。段载入到内存中后，必须将这些地址寄存器设置为指向段的首地址。由于对不同类型的段的访问可由硬件来控制，因而这种方式具有灵活及能够做到运行时安全的优点。

但是，由于段的大小不固定，内存管理器在高效地重用内存上存在相当大的困难。有时，在计算机运行较长时间后，由于**碎片**（fragmentation）问题会使得主存的一大部分变得不可访问。我们大学停车场的地面上没有画停车线。随各种不同宽度的汽车（以及驾驶员技能的不同）不断来和走，停车的布局变得无序和低效率，车与车之间的空隙存在浪费。大型车的离开和小型轿车的到达都占用相同的空间。这不过是资源碎片的另一个实例而已。专门的服务员可以走来走去组织各种汽车的停放，按照需要分配合适大小的空间，达成更有效率的停车安排。因此，通过给予大型的汽车较大的空间，将小的空间组合起来，就可以释放更多的可用停车空间。实际上，一些计算操作环境确实提供这类功能——称为**垃圾回收器**（Garbage Collector）。直接使用大小可变的段，最终必然会导致大量的内存空间被切分成微小的部分，不能再使用。为了避免这种情况，成功的虚拟内存管理方案将程序段切成等同大小的页，以页做为交换单位。由于所有的页大小相同（一般为4 KB），因而也就不存在严重的碎片问题。只是这种方式的复杂性成为硬件和系统设计人员必须面对的挑战。

微处理器诞生以来，RAM变得越来越便宜，个人工作站的到来预示着复杂虚拟内存系统的死亡，但或许令人吃惊的是，它们依旧是几乎所有现代计算机的一项重要特性。这种情况部分是由于这类操作系统提供的**保护**和**安全**特性。除管理页面交换以外，进程与它们数据的分离也是由内存分段方案提供保证。由于虚拟内存允许程序员在开发程序时无须关心程序是否能够装入现有的主存中，许多更大型的系统不断涌现，尤其是在图形和通信领域。类似地，在声明庞大的数据矩阵时，如果不需考虑RAM大小的限制，对于开发大型的数据库系统会有很大的帮助。

现代的虚拟内存系统，比如Unix和Windows XP，将主存划分成**帧**（frame），其大小常常为4 K。执行程序也被类似地划分成帧大小的块，我们称之为**页**（page）。启动程序时，这些系统并不将所有的页面都载入到主存中，载入主存中的只是那些启动程序必须的页。其余的部分被复制到称为交换文件的专为支持虚拟内存而提供的磁盘区域内。随程序的运行，读取执行周期总会请求到尚未载入到主存中的页面中的指令。这种情况称为**缺页**（page fault）。缺页中断会被触发，操作系统介入从磁盘载入后续的页。如果没有帧空闲，则必须决定释放哪个最少使用的帧，以允许新的页载入。这称做**交换**（swapping）。

用户程序中对变量的引用是32位的逻辑地址。如果所使用的系统中页的大小为4 KB，那么，我们可以将低12位看做是页内地址，高20位用做页编号。通过**页表**（page table）可以查出逻辑页编号所对应的物理帧的编号。在操作系统将程序的全部或部分载入到物理内存中时，它还必须更新页表中对应的项。

如果希望将源程序编译成可以在多任务虚拟内存的机器上运行，则代码要从地址0000开始。所有对函数和变量的内部地址引用，都建立在这个基础之上。很显然，在现实中这种情况并不会发生。不是每个程序都可以载入到相同的位置。逻辑到物理地址的映射使之可以实现，这种映射依赖于操作系统提供的页面重定位功能。程序在编译链接阶段无法指定自己载入到内存的什么位置，以及以

什么样的次序载入。因此，我们需要地址转换器来理顺由于这些不确定性而造成的混乱。MMU（Memory Management Unit，内存管理单元）处于CPU和内存之间（包括高速缓存），它将编译器设定的对变量和函数的引用，映射成计算机硬件所需的由数编号和偏移地址组成的物理引用。每次对存储设备的读写访问，MMU都会实时地执行这种从虚拟到物理地址的转换。逻辑地址和物理地址之间的关系见图12-14。MMU使用数组（页表）将逻辑页面引用映射到物理帧的地址。实践中，页表太大，常常有上百万项，不能存储在MMU中。因而只有最近使用的页面的信息保存在MMU中。如果引用的数据项不在MMU表中列出的页面中，则计算机需要读取内存，根据完整的页表更新MMU中的数组。图12-15给出了虚拟内存管理单元的不同位置。

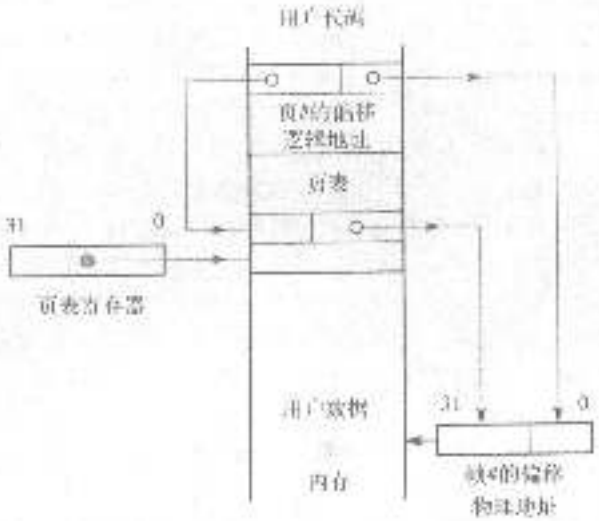


图12-14　从虚拟内存逻辑页到物理帧地址的转换

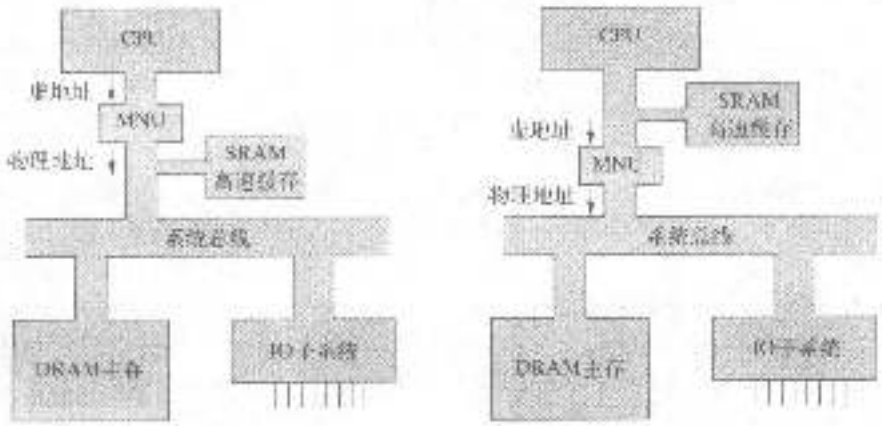


图12-15　虚拟内存管理单元的位置

由于这种转换过程必然会带来时间上的延迟，因此，我们一般会为MMU提供快速的由高速缓存构成的缓冲区，以保存来自于页表的活动项。这个高速缓存一般称为TLB（Translation Lookaside Buffer，转换后备缓冲区），它避免了每次内存访问时，都要去访问主存中的整个页表以完成虚地址转换的需要。尽管奔腾处理器和其他处理器一样，依旧保留了早期的分段机制，但操作系统已经完全转向分页技术。分段机制已经不再出现。图12-16概括了虚拟内存的页面调度和缓存，该图基于

奔腾系统。要注意，到中使用物理地址访问高速缓存，当发生虚地址到物理地址的转换时，会有延迟。然而，由于只需处理地址的前20位，因此TLB高速缓存对于加速这种活动非常高效。这样，256 KB虚拟地址空间的地址转换能够快速完成。另一种加速L1高速缓存查找速度的方法是重叠操作。尽管完整的高速缓存查询需要完整的物理地址，但由于虚地址和物理地址的低12位完全相同，所以只要MMU发出的物理地址中前20位就绪，就能够选取对应的行，将它的四个标记值读出。

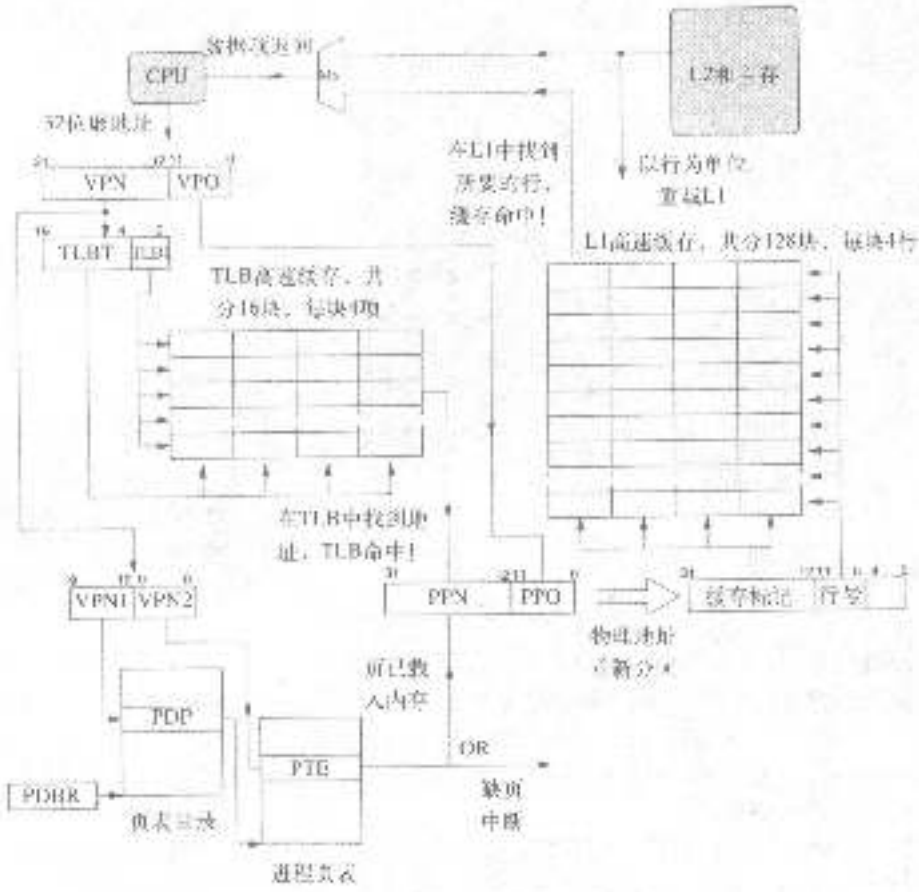


图12-16 奔腾内存地址转换次序

尽管段最初是由Intel为扩展CPU的寻址范围而引入的，但它们的作用也包括了内存访问控制。这种功能依旧可以为操作系统所用，保护和控制多任务环境，但现在这项功能实际上与分页机制部分重合，Windows使用后者。每次内存访问都得修正段偏移，与段边界寄存器进行对比，并传递给分页单元以获得等价的物理地址，这一过程看起来太缓慢；按需调页虚拟内存之所以能够工作，完全是由于快速的TLB高速缓存所致。

这项复杂的按需调页（demand paged）虚拟内存方案，现已完全替代了早期的链接和交换方案。但这种方法在使用主存方面，依旧存在微小的内存可用性损失。每个程序的最后一页可能不会填满所分配的帧。因而每个进程都有一个未完全使用的帧。这就是这种方案的损失。页面在磁盘和内存间的移动所耗费的时间依旧不确定。此外，如果存在对内存帧的过度要求，一种称为系统颠簸（thrashing）的病态情况依旧有可能发生。来自于PRIMOS系统的错误消息：

```
sorry, too busy to run logout
```

就是这种情况的强有力的证据。这种情况下，低优先级的任务永远不能获得足够的资源来完成工作。

12.6　地址公式化：时间、地点和数量

磁盘和主存间、主存和高速缓存间、高速缓存和CPU间之间数据传输的单位是不同的。这类问题常常用术语——**粒度**（granularity）来表述。磁盘设备将4 KB的页传输到主存。高速缓存要求32字节的行，CPU每次从高速缓存中读入几条指令。最终的传输单位由数据总线的宽度决定，图面典型能够处理8个字节，最多一次可以读入4条指令。

在阅读介绍寻址的课本时，常常会遇到下面几个术语：逻辑、虚拟、有效和物理。它们的含义常常重叠。图12-17对此做了总结性的解释说明。

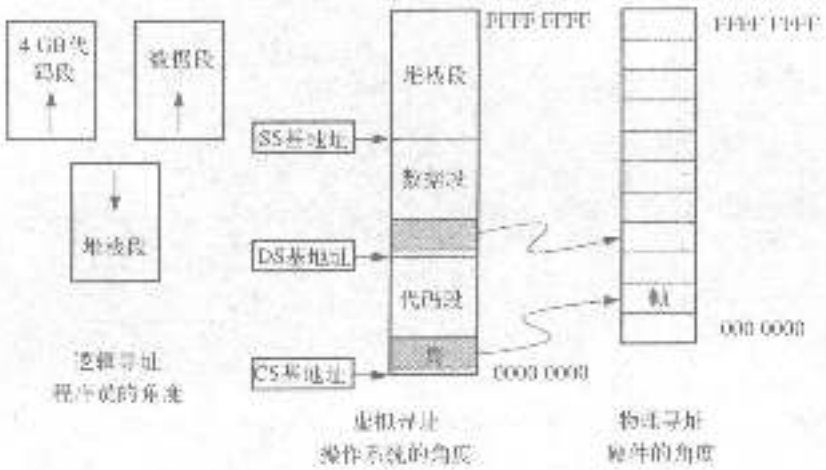


图12-17　不同地址之间的关系

逻辑地址（Logical Address）是程序员的用语，用来指代程序员在其中编写程序的理想化空间。它常常涉及好几个独立的段，每个都从地址0开始，并常常假定可以扩展到无限！哪个段是活动的，要依访问的类型而定。所有的读取-执行周期都被定向到代码段，变量的读写则是针对数据段。堆栈操作针对堆栈段。逻辑地址由段标识符和段内偏移组成。程序员还假定只有一个程序在系统上运行。即使是在多任务环境中。

虚拟地址（Virtual Address）的思想是为将独立的段映射到单一线性地址空间的需求而生的。段在虚拟空间中依次安放。每个段都被分配一个基地址寄存器以保存它的起始地址。每个用户都有一个独立的虚地址空间。载入程序负责为应用程序分配虚地址空间。我们可以将虚地址看做是为逻辑对应物的实现。

有效地址（Effective Address）在指令执行时在CU内构建。例如，程序员可能会指定应该使用基地址（数组名）和偏移量（索引）来访问一个数据结构。但是CPU需要在读取内存存储单元之前，构建一个单一的地址。这个地址就是有效地址。有效地址空间的范围由CPU地址寄存器（或地址总线，如果它更窄的话）的宽度决定。

物理地址（physical address）是分派给高速缓存控制器（进而到达主存）的实际的32位或64位地址，用来选择接下来的数据项。如果虚拟内存管理系统处于激活状态，则这个物理地址由有效地址经过转换过程获得。这个过程可能是直接加上偏移，将逻辑空间从0向上移，也可能涉及更为复杂的页表查找过程。不管采用哪种方法，物理地址空间的最大值都由地址总线的宽度决定。当然，它也受限于实际安装的DRAM数量。向空的插槽上发送32位地址没有任何意义！采用虚拟内存后，虚地址可以超出物理RAM的大小，扩展到磁盘区域。因而，虚地址的最大区间由磁盘区域（交换文件）的大小决定。

12.7 硬盘使用：参数、访问调度和数据安排

温切斯特驱动器（3340）由IBM引入，它采用气动悬浮的磁头，可以提高DASD（Direct Access Storage Device，直接访问存储设备）部件的性能。磁头和悬臂都被封入磁盘所在的外壳内，以减少灰尘的问题。如我们在图12-18所见，一粒烟尘微粒的影响都会极其威胁，而一根人类的触毛则会引发一场灾难！磁头仅在磁盘停止旋转时才会落在磁盘的表面上。磁盘上有专为磁头停靠而保留的区域。磁头对磁盘表面的刮擦将会永久性地损坏存储的数据，并使所涉及的区域不能再使用。这就是我们所知道的磁头碰撞事故。这些高容量的硬盘驱动器对现代计算技术的许多发展都做出了十分重要的贡献，大型数据库、虚拟内存操作系统、WWW服务器和电话语音消息，任何需要快速、在线访问大量数据的应用，没有千兆字节的磁盘是不可能实现的。

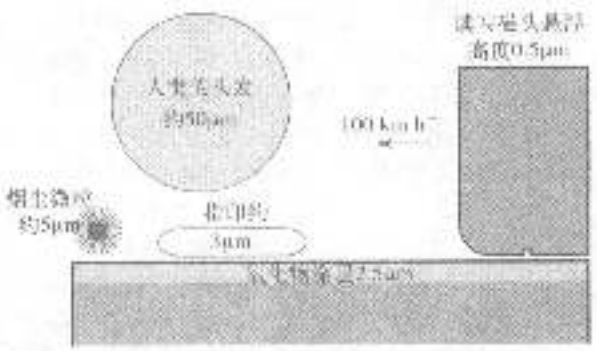


图12-18 各种环境干扰与读写磁头的相对大小

回顾一下，第一台PC机只装备了软盘驱动器。后来的PC-AT机的流行，部分是由于它将ST506硬盘作为标准配置提供。随后，Compaq提供的IDE（Intelligent Drive Electronics，智能驱动电路）磁盘设备拥有更大的存储能力和更好的性能。这种发展同时也取代了对单独AT接口卡的需求。与此同时，SCSI（Small Computer System Interface，小型计算机系统接口）已经更为广泛地为小型机用户所接受。这种接口已经被磁盘设备以外的其他设备所采用，但它要比IDE接口更复杂和昂贵。

从图12-19我们可以看出，读写磁头安装在摆动的臂上，从而可以悬停在任何磁道上。在磁盘掉电时，悬臂快速地摆回，将磁头放置在磁盘上的安全区域。用于推拉悬臂的机械装置类似于扬声器的线圈，位于较大磁场内的小型电动线圈在有电流通过它时，会因为两个磁场相互作用而产生物理位移。这是一种方便快捷的效应，但不会提供精确的定位信息。为了克服这个缺点，驱动器中的一个表面永久性地写入磁道信息，不用做存储数据，它可以提供对齐信号，作为定位回馈信号输入给伺服系统，为定位线圈提供电流。这种专门的伺服循环能够自动补偿任何温度效应，我们还可以采用一种智能的“微步进”技术，以更好地找到丢失的磁道。

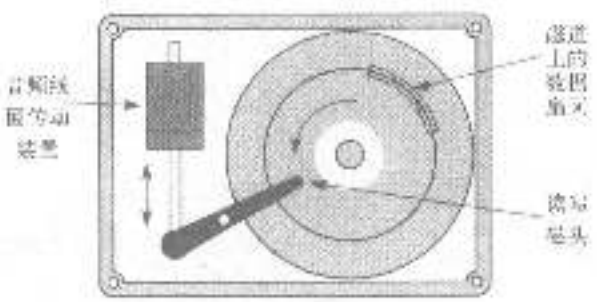


图12-19 硬盘的示意图

典型地，硬盘每磁道17个扇区，每面有980个磁道，最大存储容量可以由下面的公式得出：

磁盘容量＝表面数×磁道数×扇区数×扇区大小

磁道有时也称做柱面，因为当几个磁盘安装在同一主轴上一同旋转时，不同表面上的相同磁道表现为一个单一的扩展磁道，即柱面。

为了让计算机能够有更充分的时间来处理来自于磁盘读取磁头的快速数据流，现代磁盘常常使用扇区交错技术。扇区1、2、3、4、5等不是简单地依次放置，而是类似于1、6、2、7、3、8、4、9、5。这样就给予系统一个扇区的间隙，在数字次序的下一个扇区到达读取磁头之前，对已读到的数据进行处理。

访问磁盘数据通过告诉磁盘控制器访问哪个柱面、磁头和扇区来执行。这称为CHS（Cylinder, Head, Sector）访问，属于最底层的物理磁盘控制。将磁头移到某个柱面需要时间，等待正确的扇区转

动到磁头下的临界位置需要更多的时间。最后，数据流从磁盘读出，经过磁头，传入到数据缓冲区也需要时间。磁盘驱动器制造商需要考虑所有这些因素。通过提供商的广告或数据表，我们可以得出几个相关的技术参数，从而可以对性能进行估计。最先需要考虑的参数是硬盘驱动器设备的全部存储容量。当前，适用于PC的小型IDE驱动器容量都超过20 GB。价格也好像呈对称地增长：较小和较大的驱动器都要花费更多。这是一个大规模生产的市场，价格也反映出这一特点。我们可以得到不那么流行的产品，但要付出更多的金钱！下一个参数是磁头的移动速度，或称**寻道时间**。在此要涉及到另外两个参数：磁道到磁道和随机寻道。随机寻道是统计平均值，等于将磁头移动最大距离的三分之一。需要注意的是，从文件中读取数据存在巨大的差异，数据以整齐有序的方式依次存储时，访问速度很快，但如果数据存储在使用较长时间、充满碎片的磁盘上时，数据文件有可能会被分开存储，分散在各个地方，遇到这种情况，效率会十分低下。旋转的速度常常以**旋转等待**时间图的形式给出。在计算时，我们采用旋转周期的一半作为延迟，因为磁头平均需要半个旋转周期才能等到数据的到来。知道单个磁道有多少数据就会得出数据传输率：数据必须直接从磁头传输到内存缓冲区——此处不能有中断，否则读操作会失败。

考虑图12-20列出的性能估计数据。最新的驱动器参数在表12-4中给出，但由于这个领域依旧进步飞快，因而应该从网上下载最新的值。

```
A computer system having 2 Mbyte of RAM has hard disks with the
following characteristics:

Rotational speed:                          3600 r.p.m.
Track capacity:                            16384 byte
Heads/cylinder:                            10
Head movement time, track to track:        20 ms
Average seek time:                         50 ms

How long does it take to dump memory onto disk? Assume the disk is empty.

3600 r.p.m. = 60 r.p.s.
Rotational period = 1/60 s = 1000/60 ms ≈ 16.66 ms = 17 ms
Latency = 8.5 ms

Data rate = 16 kbyte/17 ms = 1 Mbyte/s
Flow time for 2 Mbyte = 2000 ms = 2 s

2 Mbyte needs 128 tracks or 12.8 cylinders, i.e. 13 head movements

Total time = head movement (seeks)+ rotational delays (latencies) + data flow time
           = 1 × 50 + 12 × 20      + 13 × 8.5                     + 2000
           = 2314 ms
```

图12-20　硬盘数据读取时间的估算

由于受到软件的影响，这类估计并非十分可靠，影响因素包括与其他进程的争用，或操作系统的磁盘调度等。这涉及将磁盘访问请求进行排队，对它们进行重新组织以提高性能。这种方案常常会充满争议：我们应该力图为每个进程提供公平的机会吗？或是优待其中几个？将所有对数据的访问请求按照特定柱面进行归组好像比较合理。采用这种方式，磁头只需要移动一次。但是故意延迟公共汽车的开动时间，直到所有的坐位都坐满乘客的做法，或许并不流行！聪明的算法对一部分客户产生意想不到的损害，他们可能会不理解为什么他们得到的服务变得这么差。

表12-4　Deskstar硬盘的规格表

磁盘性能规格	
磁道到磁道寻道时间	1ms
平均寻道时间	<9ms
最大寻道时间	20ms
平均延迟	5.77ms
转速	5400（r/min）
控制器开销	<0.3ms

（续）

磁盘性能规格					
	启动时间			7.3s	
	计算机接口速率			<66.7 MB/s	
	介质读写速率			<27.8 MB/s	
	每磁道扇区数			266-462	
	柱面（每表面的轨道数）			17 549	
	每扇区字节数			512	
	每表面数据区数			16	
	集成缓冲区的大小			2MB	
	存储器类型			SDRAM	
型号	7/40	7/80	7/250	7/500	
容量	40	80	250	500	GB
磁头数	1	2	6	10	
磁盘数	1	1	3	5	
寻道时间	8.8	8.8	8.5	8.5	ms
磁道到磁道	1.1	1.1	1.1	1.1	ms
转速	7200	7200	7200	7200	r/min
扇区	512	512	512	512	B
缓冲区	150	150	150	150	MB/s
GB	133	133	133	133	MB/s
缓冲区(IDE)	2	2	8	8	
(SATA)	2	8	8	16	MB

12.8 性能提高：块、高速缓存、碎片整理、调度、RAM磁盘

数据并不是以字节为单位传入或传出磁盘。几个字节的数据几乎不值得移动磁头！相反，在磁盘与RAM缓冲区之间移动的是数据块，一般为512字节。所有的磁盘访问都通过这个由磁盘控制器管理的传输缓冲区进行。**成块传输**的性能优势，仅当以顺序的方式访问数据的时候才能体现。我们知道，程序代码的运行就类似于这种方式，因而没有问题。如果数据不是连续的，显然性能会下降，因为有可能在只需要1个字节的情况下，却要读入512个字节的数据。有时，可能会通过在主存中声明更大的**磁盘缓存**（Disc Cache）来扩展这项技术。对于操作文件的程序员，最基本的需求是在结束程序前清空缓冲区：因为可能有重要的数据保存在磁盘缓存中，尚未写入到磁盘。现代计算机中装备数以千兆的主存后，磁盘缓存的问题变得更为重要。如果磁盘缓存大于活动的磁盘，这是可能的，那么整个文件系统就会在RAM缓存中运行。这种情况可能会造成即使磁盘已拔去，对磁盘的访问依旧有可能成功，这会产生破坏性的结果！使用Unix时，我们推荐在系统关机前执行sync命令，这样会执行清空整个磁盘缓存的动作。

小型的系统或许不提供本地的硬盘设备，这种情况下有可能需要使用RAM磁盘。提供**RAM磁盘**的程序保留一定量的主存，将它们模拟成磁盘安装到操作系统中，响应对设备的访问请求。早期的Amiga和Atari家用计算机和装备软盘的PC机，需要使用这种能够快速访问的伪磁盘，以使HLL编译成为一项可以容忍的活动。要注意的问题是，在关闭电源或重启系统之前，需要传输所有保存在RAM磁盘上的重要数据文件。

当一项任务对磁盘上的数据发出请求时，操作系统的磁盘处理例程会简单地以**“先到先服务”**（First Come，First Served，FCFS）的方式予以响应。数据请求会被排队，等待执行。这样会使悬臂的移动十分低效，读写磁头可能会匆忙地在磁盘表面上来回移动，以响应即时的请求。想像一下，这就如同邮递员随机地从邮包中取出一封信，然后立即出发去递送它！在有许多工作要做时，最好对请求进行整理，先计划一下路线，以减少无谓的往返。现代的多处理器计算机系统中，可能会装

备几十个大型的磁盘设备，提供数兆兆字节（10^{12}字节）的在线存储，同时服务于许多用户。请求队列不可避免地会跨多个磁盘组。在并发处理有大量请求的请求队列时，需要考虑重新调度请求的可行性。在小型的系统上，默认的策略FCFS可能就足够了。但对庞大的公司数据库，即"磁盘农场"(disk farms)，则应采用更为完备的方式。可能需要将请求记录下来，让执行最快的请求先发送出去。可能优先处理那些需要最短移动距离最小的请求——最短寻道时间优先。这种方式的风险是那些数据存储在较远的，不太通用的区域的用户，可能会受到不公平的对待，比其他用户等待更长的时间。

另一种方式取材于邮件的递送。磁头执行固定的扫描路径，一般是顺序地访问磁盘的各个磁道，对所到达的读-写请求提供服务。如果磁头恰好在新的请求需要访问的位置，则新的请求被插入到队列中，可以得到快速的服务。这有些类似于将邮件传送给正在半路的邮递员。这样他就不需要返回分送办公室。这种方法称为扫描（SCAN）策略。在安装电梯的公司中，这种算法应用十分普遍。采用扫描方法时，请求时间更容易预测，但在所有的请求都局限于几个邻近的磁道时，需要做出改动。因为如果没有邮件，就没必要走遍村子的各个角落，访问每个家庭。查看（LOOK）方法就是应此项需求而生的。采用这种方法时，对磁盘的访问请求首先在局部进行分类，然后按照磁道顺序进行服务。图12-21给出读-写磁头在磁盘上不同区域移动的图示。这些移动均服务于同一个初始队列。这种情况下，性能上的差距是很明显的（见图12-22），扫描策略能够很容易地胜出。

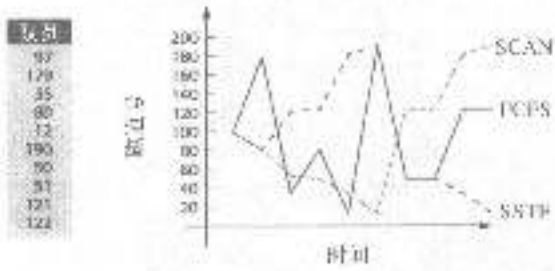


图12-21　磁盘访问调度技术

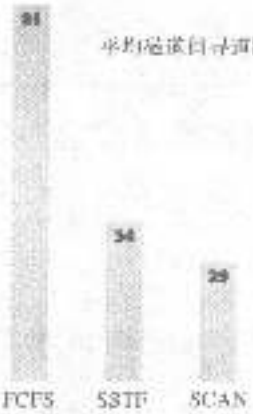


图12-22　磁盘调度技术的对比

所有可读写存储设备的特有问题，就是可用空间的碎片问题。12.5节中已经从另一个角度介绍过碎片问题。但磁盘存在的问题有些不同。在磁盘中，不是空闲空间的基本单元不可用——它们的大小均相同，因此没有问题。相反，问题在于如果存储文件的各个块不以连续的方式存储，而是散布在整个磁盘组，则访问时间会变长。要访问这样一个文件，针对每个块磁头都需要移动到新的磁道，这种情况并不好。如果延迟变得不可接受，可以采用"碎片修理"实用工具重新安排数据块。基本上，它会重写所有的文件，将数据整理成线性的连续排列。碎片整理是一项十分耗时的工作。

12.9　光盘：CD-DA、CD-ROM、CD-RW和DVD

光盘存储起源于音乐行业中使用CD作为塑料LP的数字替代物。它们是由金属板片机械挤压出来的聚碳酸酯片。约5公里长的单螺旋轨道（见图12-23）能够保存74分钟的立体声音乐，双声道，16位编码，44.1 kHz采样率。数据位表示为唱片上的物理凹陷，它会使激光束发散，降低反射光的亮度。来自于CD驱动器的高品质音乐数据流的速率为150KB/s，大约每22.7μs就有一个新的值。基于某种原因，压制在一面完成，而读取在相反的一面执行。激光必须穿过

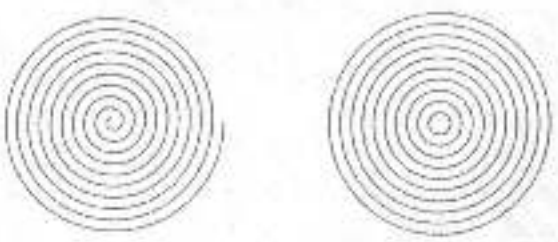


图12-23　CD数据螺旋和磁盘的同心圆轨道

整个聚碳酸脂盘基（1.2mm），以检查是否有凹陷存在（见图12-24）。对顶部涂层的划擦较之激光进入的一面更为严重，更容易造成表面意外损坏，从而使得盘片不可读取。由于顶部的涂层只有约10μm厚，凹陷的深度大概在0.1μm的数量级，因而顶面划伤的可能性很大。让激光从下向上穿过盘基底面的原因，是为了降低表面污点对光束的影响。这可能是因为光速在进入盘片时，并不聚焦在一片小的区域，因而划伤显得不那么极具破坏性。

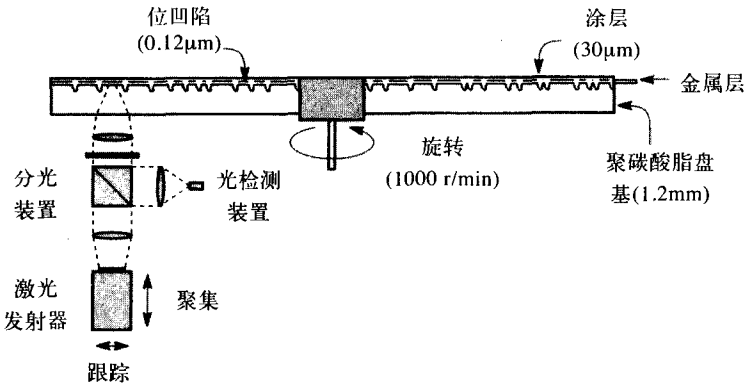


图12-24 光盘读取头

数据在CD-DA（Digital Audio）上的组织是考虑到长时间的连续访问而规划的。螺旋轨道被划分成270 000个块，每个保存2 KB的数据。每个块有12个字节的引导头和索引号。索引号是三对BCD（Binary-Coded Decimal）数字编码，表示已经逝去的播放时间：分钟、秒、扇区。每秒钟有75个扇区，每分钟60s，每张盘60（实际为74）分钟。额外的14分钟播放时间要归因于ECC（Error-Correcting Code，错误纠正编码）占据的空间。ECC编码采用的是一种十分强大的Hamming类型的编码——Reed-Solomon编码。每个2 KB大小的块还可能附加288个字节的检验和。一分钟未压缩的CD音乐需要约10 MB的存储空间。

将CD-ROM制造技术和回放设备用于存储计算机数据时，惟一需要做出的重大变动是，提高错误的检测率和纠正率。音乐CD只采用基本的CRC（参见10.3节），由于音乐信号的漏码错误可以简单地通过内插来解决，并没有什么太大的困难，因此基本的CRC就足够了，但对于计算机数据来说，单个错误可能就会十分严重，甚至致命。为提高对错误的捕获能力，数据被重新分块，如同在磁性介质中那样，对每个块应用辅助CRC。块的大小一般为2 KB，附加的ECC为280个字节，大约为存储容量的10%，可用空间降低到650 M。但这样，CD就适合于安全的备份存储和长时间的存档。需要记住的是，它们以螺旋的方式编排，大约有三英里那么长，光盘的读取也是从中心向外，和乙烯基的LP相反。

工厂预制的光学CD-ROM，在游戏软件的发行中已经得到广泛的采用，但它们的影响直到光盘刻录设备出现后，才得到完全的发挥。此后，如果需要的副本不多，用户可以方便地直接刻录，无需工业的压制设施。软件发行和长期的归档，开始使用称为CD-WORM（Write Once，Read Many）的一次写入多次读取光盘，它们开始逐渐侵占磁带技术的市场。与磁性的硬盘相比，它们具有可移动的优点，完全兼容不同的机器。但相比于磁盘，它们的访问速度要慢得多。CD-WORM驱动器使用高功率的激光束，在夹在聚碳酸脂盘片内的精密金属层上烧蚀微小的孔洞。这种不可逆的过程之所以可行，是因为空白光盘价格比较低廉，发生错误是可以容忍的。另一种方案使用化学染料层，使得较低功率的激光束就能够完成写入。但是，擦除数据的能力和光盘重用，依旧是我们想要的。有两种方案可以完成这两项功能：磁光学盘片和相变CD-RW。

磁光学盘片，也称为MiniDisc，同样使用高功率的激光加热嵌入在盘片内的金属层。这个金属薄膜是根据金属的磁学属性选取的，但如同所有的磁性物质一样，在加热到特定的温度后（居里点），它会失去自身的磁性排列。但在冷却下来后，它极易受到所处的任何磁场的影响。因而，CD-MO驱动器中还有电子线圈，用于在磁盘上激光聚焦的点附近建立磁场。通过反转磁场的方向并发射激光，可以逆转该点处

的磁区，从而保存1或0。读取技术更为复杂。首先要对来自于低功率激光的光进行极化。这样当光束被磁性薄膜反射时，由于光学和电磁学领域间复杂的交互作用（Kerr效应）的影响，它的偏振面会受到影响。

另外一种可重复使用的光盘是CD-RW，它提供完全的重写能力，但不依赖于磁学。它使用可以以两种状态存在的化学物质：有序的结晶态或混乱的非晶体态。高功率激光束产生的热量能够融化晶体，依冷却过程时间长短的不同，这种化学物质可以冷却为均一的结晶态，或无序的非晶体态。减慢关闭激光的时间可以制造出较慢的变化。为了读回存储的数据，必须能够根据这两种状态对光的反射来区分出它们。这项工作的达成要感谢晶体和非晶体薄膜所表现出来的不同的反射率。如图12-25所示，在制造时，就在聚碳酸脂盘片的顶部物理性地压入螺旋形的轨道。盘基上还要沉积其他的层，以产生可转换的反射率来提供写入读取能力。压出的轨道在访问数据时，可以有效地引导读取头。

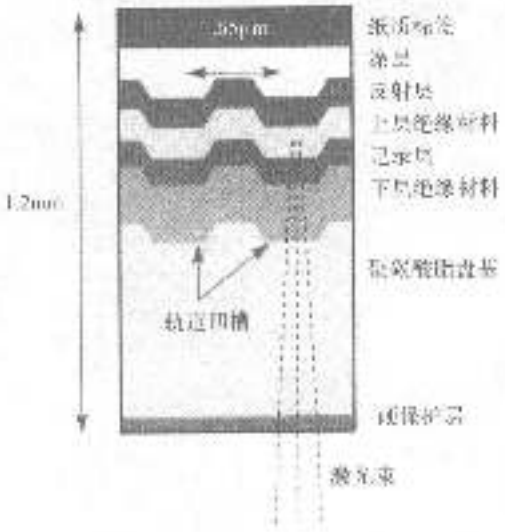


图12-25　CD-RW盘片结构

不同于磁性盘片，CD-ROM内圈轨道上位的大小与外圈轨道上的大小相同。这是由于磁性盘片的位都由恒定长度的脉冲写入，而CD-ROM是机械压制而成的。因而磁性盘片在任何轨道上保存相同数量的位，而CD-ROM在螺旋的外圈轨道拥有3倍的位。依据这种情况，为了维护音乐CD的恒定数据速率，旋转的速度依读取头在螺旋上的位置的不同而不同。随读取头向外缘移动，光盘的旋转逐渐慢下来。对1倍速的驱动器，内圈的轨道每55ms一圈，要维护恒定的读取速度，外圈则要耗费150ms。这种方式称为CLV（Constant Linear Velocity，恒定线速度）。Apple最初在Macintosh机的软驱中就实用了这种技术。它允许外缘轨道可以存储比内圈轨道更多的数据。对于更高性能的驱动器，控制旋转的速度变得十分困难，它们放弃CLV，转而采用更简单的CAV（Constant Angular Velocity，恒定角速度）技术，常规的硬盘中也是采用这项技术。

在PC上使用CD-R驱动器时，还存在一个连续性的问题。在将数据文件写到光盘上的时候，数据流必须连续，否则操作将会不可挽回地失败。在将数据块写入到螺旋形轨道上时，一分一秒都不能停下，否则光盘会被写坏。对于单倍速的驱动器，这种问题一般不会发生，但随着技术的进步，2倍速、4倍速和8倍速版本不断开发出来，数据欠载的风险也越来越大。为了应对这种威胁，CD-R驱动器的制造厂商在驱动器内提供2 MB的RAM缓冲区，以平滑数据提供过程中的任何异常现象。2倍速的驱动器，每秒需要300KB的数据，如果提供这样一个缓冲区的话，可以应对6秒的间隔。

12.10　DVD：数字通用光盘

数字通用光盘，有时也称做数字视频光盘，正在微机存储领域取代CD-ROM和CD-RW的角色。和CD-ROM一样，DVD既能作为一种娱乐媒介，也可以用做计算机海量存储。DVD的只读版本首先出现，最初的意图是提供全长影片的回放，以及服务于大型的计算机图形游戏。后来，可重写DVD-RW盘片在音频应用领域得到广泛的应用。

DVD对CD-RW技术进行了扩展，它允许多层记录或双面光盘。单面单层DVD提供4.7 GB的存储容量，而CD只能达到650 MB。双面双层DVD能够提供高达17 GB的容量。

DVD和CD-ROM的基本制造流程十分相似。DVD的螺旋形轨道约12km长，轨道间的间隔只有740nm。生产时，首先将注射成形的聚碳酸脂盘片冲压，随后是加反射涂层。DVD需要两个这样的盘片粘合在一起。双层盘需要沉积一个半反射层，以便激光束能够聚集在任一个层上，进行读取和写入。DVD中的激光器件提供比早期的CD更高的分辨率，能够可靠地处理更小的位凹坑。单层DVD的最小凹坑长度是0.4μm，而CD为0.83μm。另外，DVD轨道间距降低到0.75μm，是CD的一半。

由于将透镜从一个反射层聚焦到另一个反射层速度很快，因此，数据在DVD上的布局可以模拟多磁碟硬盘的柱面模型。同时，数据还可以以任一方向读出：由外向内，或由内向外，我们可以为此引入一些新的调度策略。

DVD技术在计算机存储领域的应用从DVD-ROM开始，它提供3.9 GB的容量，DVD-RW提供4.7 GB（单面）和8.5 GB（双面）的容量。读写速度当前与CD-ROM和CD-RW相同。

12.11 MPEG：视频和音频压缩

在存储和传输高分辨率的图像、音乐和视频文件时，高效的数据压缩技术十分重要。其中最流行的技术是**MPEG**算法，它由运动图像专家组（Moving Picture Experts Group）制定，它所代表的是一系列为音频-视觉信息（如电影、视频、音乐）的数字压缩而开发的算法。MP1、MP2、MP3和MP4均为相关的压缩标准。

尽管MPEG算法属于有损压缩，解压缩后的文件和最初的版本并不完全相同，但与其他视频和音频编码格式相比，在同等质量的情况下，MPEG格式的文件更小。这是因为MPEG使用了十分复杂的技术去识别并消除原始数据流中存在的空间和时间冗余。

运动图像专家组由国际标准化组织（International Organization for Standardization，ISO）和国际电工技术委员会（International Electrotechnical Commission，IEC）提供赞助。它由几个小组构成，目标是制订多媒体文件（包括图像、视频、音频以及类似数据）的编码标准。MPEG-1关注运动视频和音频。MPEG-2是MPEG-1的扩展，其图像分辨率更高，支持TV上常用的隔行扫描，提供更佳的错误恢复过程，能够处理更多的色度格式，并通过采用非线性图像量化获得更高的压缩效能。MPEG-3处理高度压缩的音频流，但由于受Napster官司的影响而名声不佳。

MPEG视频压缩使用多项技术，在不影响视频感官体验的前提下，达到更高的压缩比。这个过程的核心是**离散余弦变换**（Discrete Cosine Transformation，DCT），采用这种变换，我们可以将任何图像拆分成由空间分量构成的图案。它的基础是**傅立叶变换**——任何复杂的波形均可以通过许多简单的正弦波形分量叠加而成。图12-28和12-30给出了一些DCT的基函数。

一幅数字图像首先被划分成许多8×8的像素块。然后使用DCT，将这些由64个像素构成的块转换成空间分量图。这样，每个像素块就被表示为64个波形系数，见图12-27。至此，所有的处理过程完全可逆，我们完全可以重建该图像，不会有任何损失或失真。这也是空间（spatial）压缩发生的地方。利用人类视觉系统的缺陷，我们在表示低频分量时采用相对较高的精度，表示高频分量时则采用较低的精度，即相比较而言，采用更多的位来表达低频数据，见图12-29。这是因为在查看物体时，大的物体比小的物体更容易引起我们的注意。我们的视觉系统对于空间域的高频分量的分辨能力较弱。因此，MPEG视频压缩实际上丢弃了图形在空间域中不容易为人类所感知的高频信息。如果让鹰和海鸥来观看我们的MPEG视频，它们可能会觉得不可接受。

到目前为止，图像被划分成8×8的像素块。每个像素块经由DCT算法转换成64个分量系数。我们可以将系数放在方格内，如图12-27所示，[0,0]中的数字表示最低低频分量的振幅——我们可以将它认为是8×8宏块的平均亮度。[0,1]中的值表示低频在水平方向上的变化，[1,0]中的值表示低频在垂直方向上的变化。类似地，[0,7]和[7,0]分别表示高频变化。[7,7]中的值描述高频在对角线方向的变化。

图12-26和12-27中的系数均用4个二进制位来表示。考虑到人眼有限的视觉分辨能力，64个分量均采用4位来表示有些浪费。图12-29给出一种更节约地分配存储空间的方案。原来的矩阵有64个格，每个格用4位，总共需要256位。精简后的矩阵所需的位数为：

$(12 \times 1) + (24 \times 2) + (19 \times 3) + (9 \times 4) = 12 + 48 + 57 + 36 = 153$位

这就节省了大约40%的存储空间。实际的MPEG编码器中，每一帧上DCT矩阵使用的位分配都有可能不同。需要注意：4位分量能够表示分量的16级振幅，而单个位只能够表示该分量是否存在。如果你正在将不同的颜色混合在一起，这样的限制会有什么影响呢？你或许可以每次6%地调整红色，但蓝色则要么全部，要么没有——这种限制还挺麻烦的。

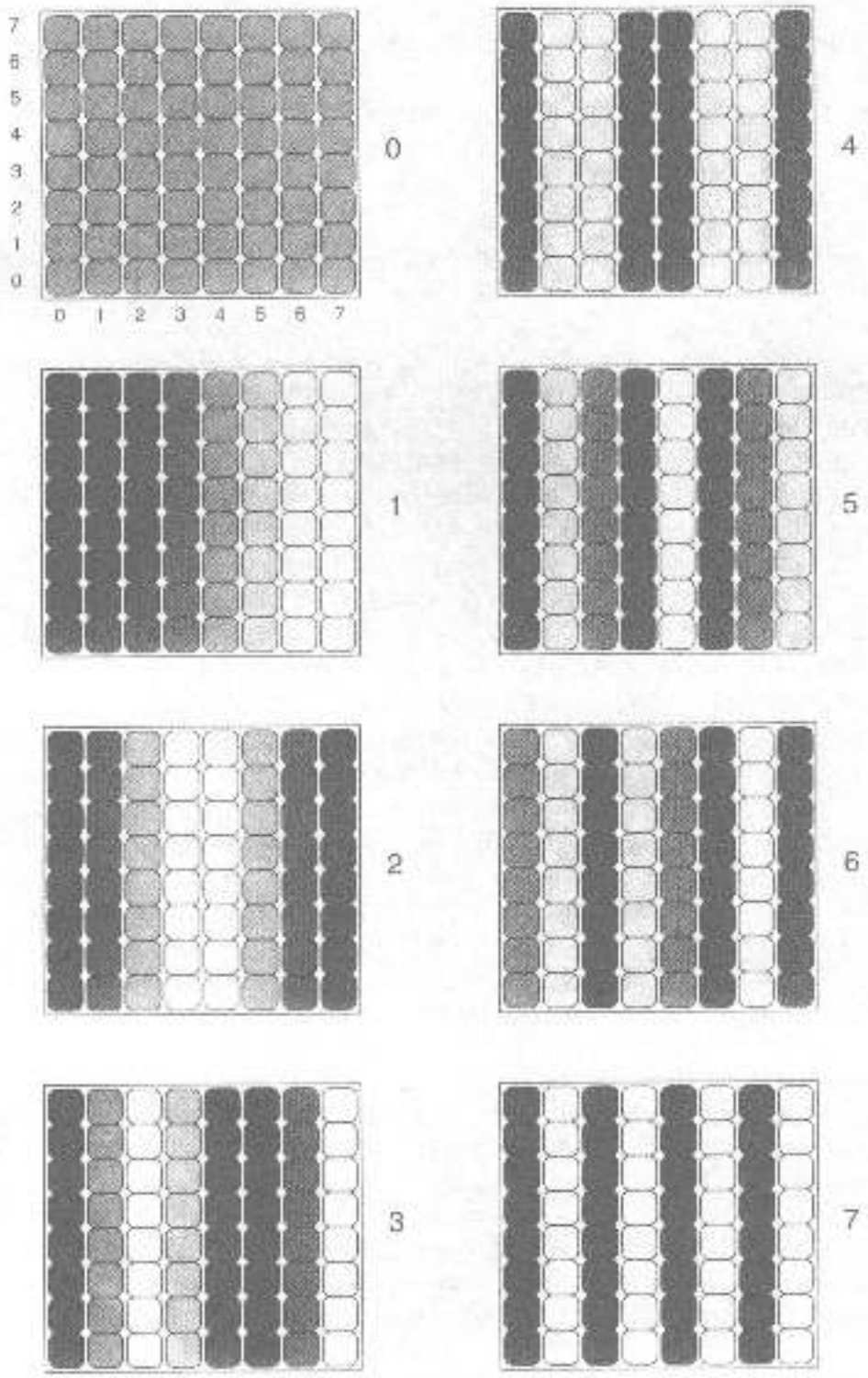


图12-36　8×8像素组的水平空间分量

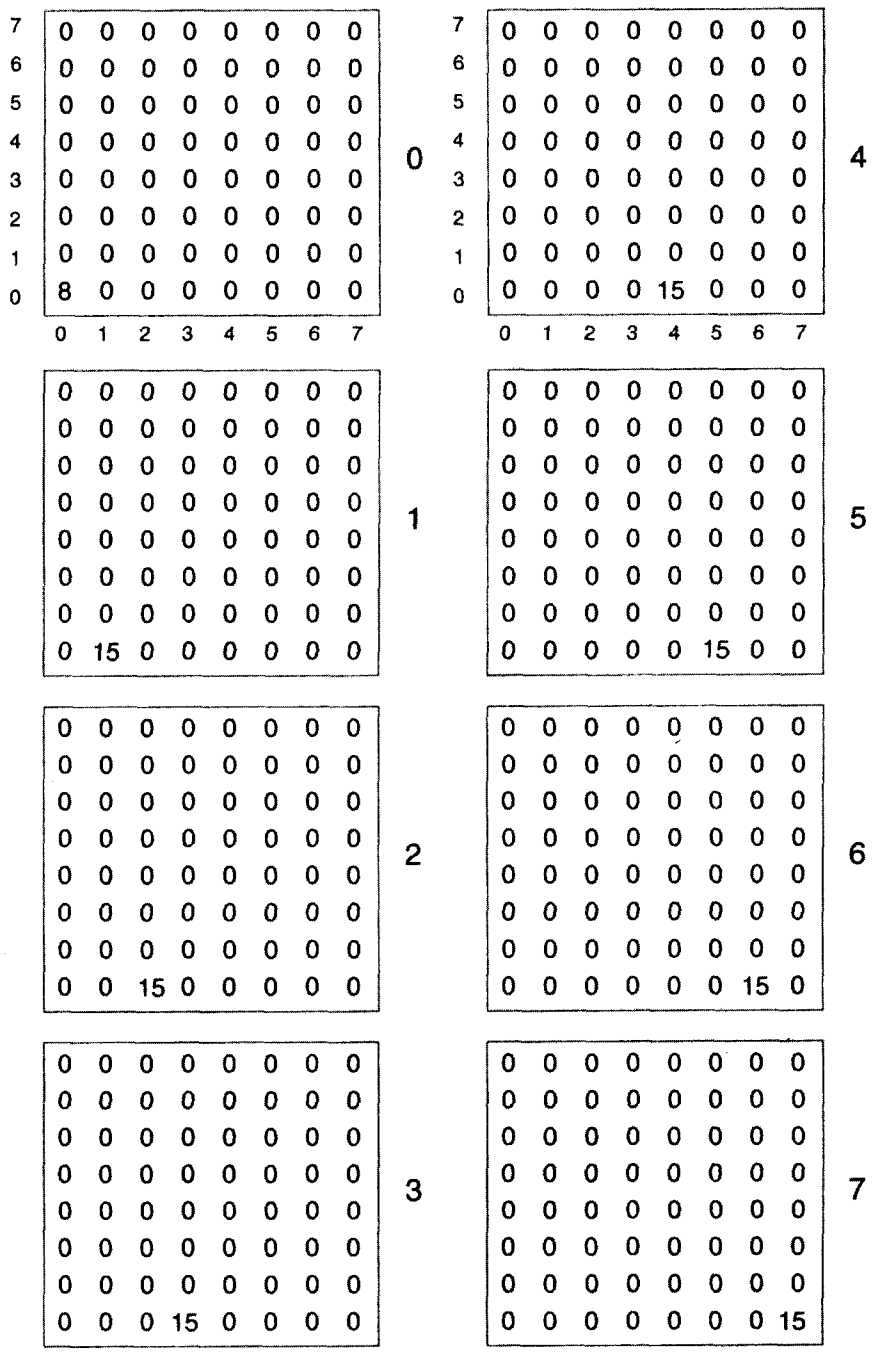


图12-27 对应图12-26中各种分量的4位水平空间频率系数块，每个64个系数构成的矩阵代表一个8×8像素块的空间分量

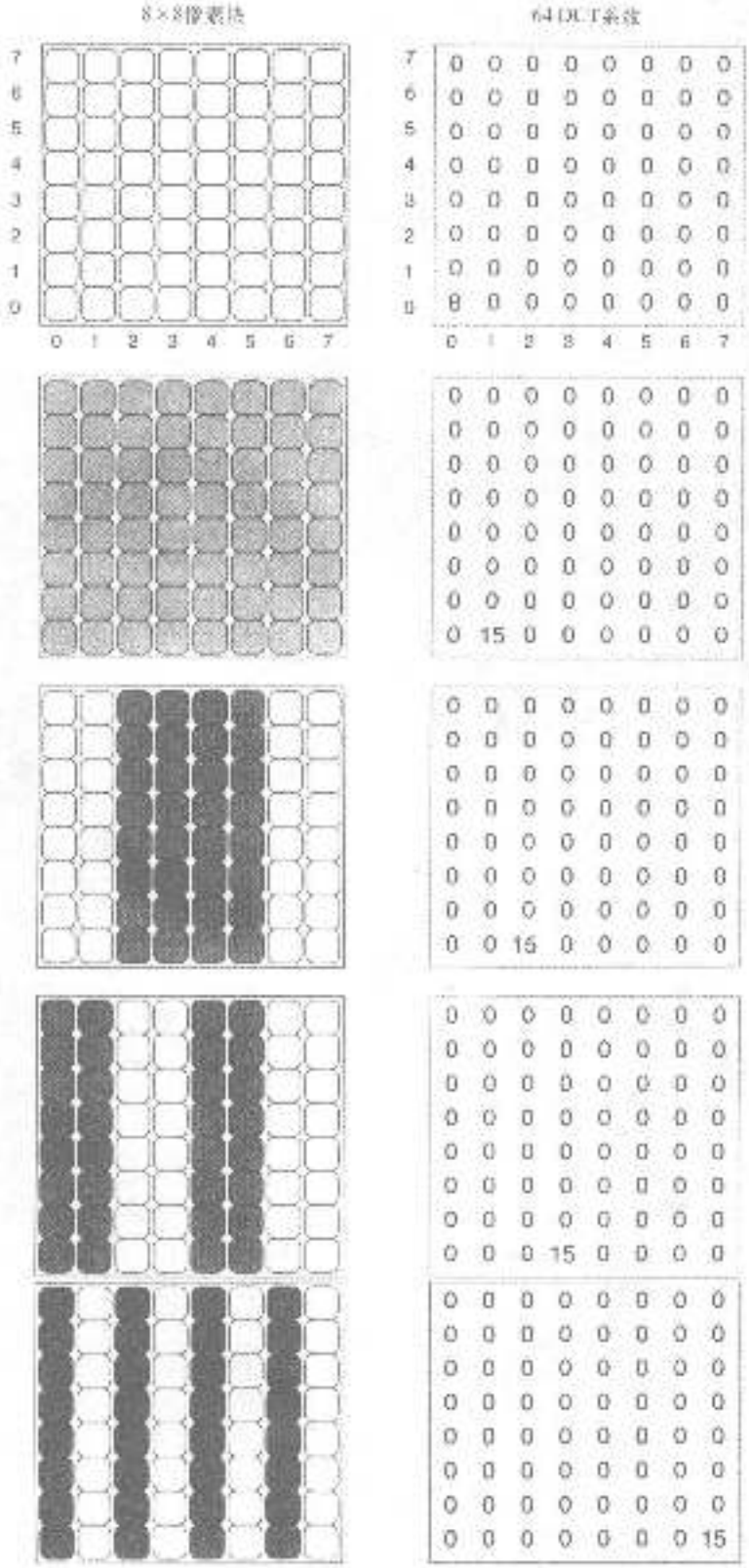


图12-28　DCT 64个系数的分配矩阵

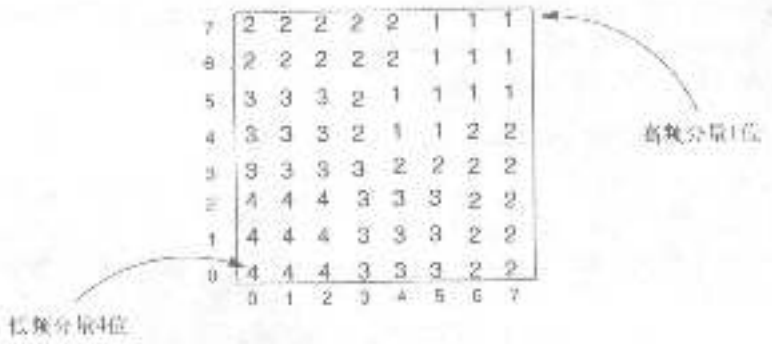


图12-29 精简后的DCT系数位分配（由256位精简到155位）

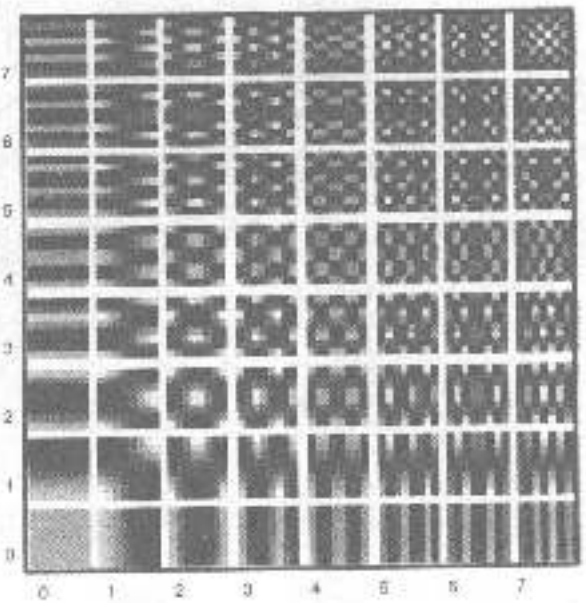


图12-30 针对8×8像素块的DCT基函数

对于单幅图像，DCT块的系数采用哈夫曼频率编码。在这种编码算法中，常用的符号以更短的代码值来表示，而不常使用的符号使用较长的编码值，这样就使得最终生成的文件较小。

对于视频图像序列，64个值组成的系数块构成宏块，表示256个相邻像素。之后，这些宏块被用来确定原图中那些在帧和帧之间只是简单地改变位置的部分，如果能够完成这一步，视频中仅需要提供完整的第一幅图像（I帧）。后面只需发送向量信息更新元素在图像中的移动即可，不需要重复传送不改变的背景细节。接收端的MPEG解码器，只需简单地将未改变的宏块从前面的参考帧中复制过来即可。对于那些相对于参考帧仅发生细微改变的区域，压缩算法首先计算像素的差值，然后使用DCT编码后传输。如果编码器能够检测出物体从一个宏块移动到另一个宏块的区域，它会发送运动向量和差异信息。运动向量告诉解码器宏块移动的距离和方向。如果编码器在参考帧中找不到类似的宏块，则直接对其进行编码并传输，就像在I帧里那样。MPEG文件的编码较之解码要复杂得多，同时也更耗时间。

通过使用来自于之前和之后图像的信息，传输视频流所需的带宽能够得到节省。一般地，每12帧有一个I帧，其他的帧所需的位只有I帧的40%。如果前向和后向相关均采用，帧的数据甚至可能降低到I帧的15%。

MPEG音频压缩依赖于人类听觉感知的心理声学模型，对不重要的频率使用较低的精度，以降

低编码后数据流所需的带宽。音频数据以预先确定的速率进行采样，固定电话系统以8 kHz对语音进行采样，音乐采样速率一般更快，达到32 kHz、44.1 kHz或48 kHz（MPEG-1音频）。另外，音乐数据采样一般16位宽，而电信业内仅使用8位。

12.12　闪存：新型软盘

当微处理器初次在市场上出现时，对于开发出价格能够让普通消费者承受的个人计算机系统来说，软盘起着十分重要的作用。最初，IBM引入128 KB的8英寸软盘，是为了将微码载入到大型微机中，但它很快被应用到微机行业，作为新型微处理器CPU理想的次级存储。很快，更小的3.5英寸软盘系统取代了价格昂贵、可靠性差的5.25英寸和8英寸驱动器。典型地，3.5英寸软盘有两面，每面有80个磁道，每磁道有18个容量为512字节的扇区，总容量为1.4 MB。不同于硬盘，软盘驱动器的读取头在读取时直接与软盘的磁性表面接触，即使旋转速度很低，只有300r/min，也会存在磨损问题。苹果公司的软盘旋转速度可变。软盘在访问外圈磁道时，以300r/min的速度旋转，在访问内圈的磁道时，旋转速度为600r/min，这样能够保证轨道上的位密度恒定，避免了外圈磁道数据位较长的问题。变速驱动技术已经重新在CD-ROM设备上焕发青春。

USB闪存盘可以提供256 MB以上的存储容量，现已完全取代软盘，成为一种便携式介质。闪存盘一般预格式化为MS-DOS FAT-32文件系统——将在第19章中介绍。尽管这种文件系统不是尖端科技，但它不需要任何调整，就可适用于Linux和Windows操作系统。闪存芯片需要专门的驱动来处理读操作和写操作之间的不同。删除和改写操作比读取操作要花费更多的时间，并且必须以阻塞的方式完成，因而更像是磁盘，而非RAM。同时，将面向文件的数据块请求转化成一系列的随机访问操作，也需要专门的设备驱动程序来完成。

12.13　小结

- 由于费用的缘故，计算机的地址空间很少被完全占满。为提高性能，我们提供一个存储体系，力图匹配处理器的需求。
- 性能特征不同的存储设备划分成不同的层，在需要时，操作系统会在层间传输数据。
- 这种分层体系能够高效工作的能力，依赖于系统对存储设备的访问的局部化，即在特定的时段内，访问集中在一段区域内进行。
- 程序员应该理解存储体系的结构和功能，这样才能据此优化他们所编写的代码。
- 高速缓存位于主存和CPU之间。它比主存小，但比主存快。要得到最佳的性能，高速缓存最好和CPU在同一芯片内。
- 主存的溢出区，或称交换区，在磁盘上分配。这是虚拟内存的基础，在虚拟内存中，所有的程序都被拆分成固定大小的页，映射到内存中的空闲空间。MMU负责地址转换过程。
- 通过将硬件设施（MMU）和操作系统功能结合起来，我们可以将虚拟内存划分成段，对指定区域的内存提供访问控制。
- 磁盘的性能是由访问时间（磁头移动的速度）、延迟（旋转速度）和数据传输速率（磁道位密度）等参数决定的。碎片文件的读取会花费较长的时间，因为需要更频繁地移动磁头。
- 硬盘部件在洁净室内封装，以避免灰尘的污染。磁头悬浮在由旋转的磁盘形成的一个空气薄层之上。
- 操作系统可能会对硬盘读写操作的请求排队，并重新调度，以将磁头移动的开销最小化。有多种不同的算法可供采用。
- 光盘可以稳定地存储许多数据，但就目前而言，其访问速度较慢。

实习作业

我们推荐的实习作业包括调查内存性能的差异。可以设计会导致（或不会导致）虚拟内存交换的程序。最好设计出能够充分利用高速缓存的程序。

练习

1. 从Internet查找内存模组和IDE磁盘的价格。分别计算SRAM、DRAM和磁盘存储设备的每位成本（cost per bit）。然后计算每种存储介质访问带宽的花费（单位为bits/s，即位/秒）。
2. 为什么Intel要在PC领域引入两级SRAM高速缓存？
3. 访问的局部性如何使PC的性能受益？
4. DRAM代表什么？它由什么而得名？DRAM最近采用什么方式减少随机访问？SDRAM DIMM如何接入PC的L2缓存？
5. 将12.2节中的程序改为将和写回到数组中。

```
Sum += big[i][j];
Big[i][j] = sum;  /* 加入这一行 */
Sum += big[j][i];
Big[j][i] = sum;  /* 加入这一行 */
```

   这样的改动对高速缓存有什么影响？
6. Intel将L1高速缓存分成两部分，分别处理数据和指令。采用统一高速缓存会存在什么问题呢？
7. 在什么情况下会发生高速缓存未命中呢？程序员能够降低高速缓存未命中的频率吗？
8. 从表12.4中选出一个磁盘，重复图12-20中的计算，但将主存设为64 MB。
9. DVD和CD-ROM的区别是什么？
10. 计算74分钟播放时间的音乐CD上，记载了多少未压缩的数据。这种方案所能够精确回放的最高频率是多少？你能在PC上播放音乐CD吗？
11. CD-ROM驱动器的访问速度比硬盘慢得多。为什么？你能说出怎样改善这种情况吗？

课外读物

- 磁盘和CD格式的技术信息，可以参见PC Guide：
  http://www.pcguide.com/ref/cd/
- Heuring和Jordan（2004），存储器设计和磁盘系统。
- Patterson和Hennessy（2004），存储体系。
- 下面这些资料也挺不错：
  “How do CD-Rs behave when microwaved?”：
  http://www.cdrfaq.org/faq07.html S7-8
- MPEG编码技术简介：
  http://www.techonline.com/community/ed_resource/feature_article/37054
  其他的一些问答集也值得一读。
- 这些网站可以通过本书的配套网站访问：
  http://www.pearsoned.co.uk/williams

第二部分 网络通信及复杂性的增加

第13章 程序员的观点

我们可以从许多不同的角度对计算机进行分析。硬件工程师和程序员对同一设备的表述肯定会各不相同，即使主要从软件的角度来考虑，表述计算机的各种功能也有许多不同的方式。本章介绍不同用户的视点，以帮助读者理解其他用户的需求。通过对计算机各个方面的调查，我们引入分层抽象技术，以帮助读者更好地划分计算机的复杂功能。本章还介绍虚拟机的概念，以及了解一些为网络应用提供支持的基础性功能的必要性，因为网络应用越来越流行。

13.1 不同的观点与不同的需求

在处理与计算机相关的事物时，能够了解一些有关底层硬件和操作系统软件的技术细节，将会很有帮助。但现实生活中，许多计算机用户在完成他们各自的工作时，常常并不需要用到这个层面的知识。实际上，他们使用更为简化的计算机**抽象模型**，这些模型只保留与特定任务相关的结构和信息。经验会告诉我们，哪些信息重要，而哪些仅仅具有理论价值。如果同事或客户有不同的观点，那么合作起来将会颇具挑战性（见图13-1）。有时，你会发现不同的人使用不同的词来指代同一常见事物。他们可能会将程序叫做应用，把数据文件称为文档，将目录称做文件夹。这些词汇上的变动常常并没有将事物简化，反而使之神秘化。遗憾的是，当前，这是快速变动、高度竞争的商业活动的共性。



图13-1 不同的工作，导致不同的观点

我们以一个运行速度过慢的电子表格程序，来说明审视和处理问题时的各种不同的角度及方式。如果某个大电子表格需要40分钟才能完成更新，硬件工程师会建议升级处理器以加速基本的机器周期。在最好的情况下，这样做或许能够将运行时间降低。应用程序的用户则被告知效能差是因为过时的软件产品所致，采用新的全功能电子表格软件，包会令结果大有改善。如果真的采用新的软件，有可能更新要花费60分钟的时间。系统管理员可能会采用Unix awk脚本来执行同样的计算任务，可能只需10分钟即可完成。但它只能完成指定类型的计算。HLL程序员可能会首先阅读用户手册，然后发现通过禁止每次计算完后都刷新屏幕，处理时间可以降低到4分钟。系统程序员会试使用指针取代单元格索引访问存储数据的数组，可能会成功地将更新时间缩短到40秒以下。

软件日益影响我们生活的一个方面，是其越来越庞大的开发费用。每年数十亿的金钱花到新软件的开发上，但大多数都未能成功交付使用。软件依旧是极耗费人力的手工制品（craft），成本因素已经使得软件系统有时对社会产生意想不到的宇宙影响。由于完成转换工作需要投入极为高昂的代价。Y2K问题大部分是由于分隔旧的程序造成的，这些程序在设计之初，从未考虑过能够使用十多年以上，但事实上30年之后程序依旧在维护。没有程序员会想像到，他们在20世纪60年代早期写的大型机COBOL程序，会在1999年还在运行。

13.2 应用程序用户及办公软件包

绝大部分的计算机只用于日常的字处理、电子表格和电子邮件应用程序。这种现实可能会让许

多计算机科学家感到失望。Microsoft Excel、Word和各种电子邮件程序为大多数人提供熟悉的日常工作环境，他们对软件如何工作并不十分感兴趣，对于硬件安装一般并不在行。这种技术上的脱离对于个人或组织来说并不好，对硬件和软件功能的更透彻理解，将有助于日常工作。

使用字处理软件包的用户，他们的视野可能仅仅局限于图表式的文件系统（见图13-2）、屏幕、打印机、键盘和鼠标，**WIMP**（Windows，Icons，Menus，Pointer）界面在20世纪80年代中期引入，引入它是为了使那些不是专业从事计算工作的人，更容易地访问计算资源。就这一点，它非常成功。Windows GUI（Graphical User Interface，图形用户界面）中的许多功能，都可以通过简单的拖放来完成，对大多数用户来说，这种界面要比之前的基于字符的界面（见图13-3）友好得多。

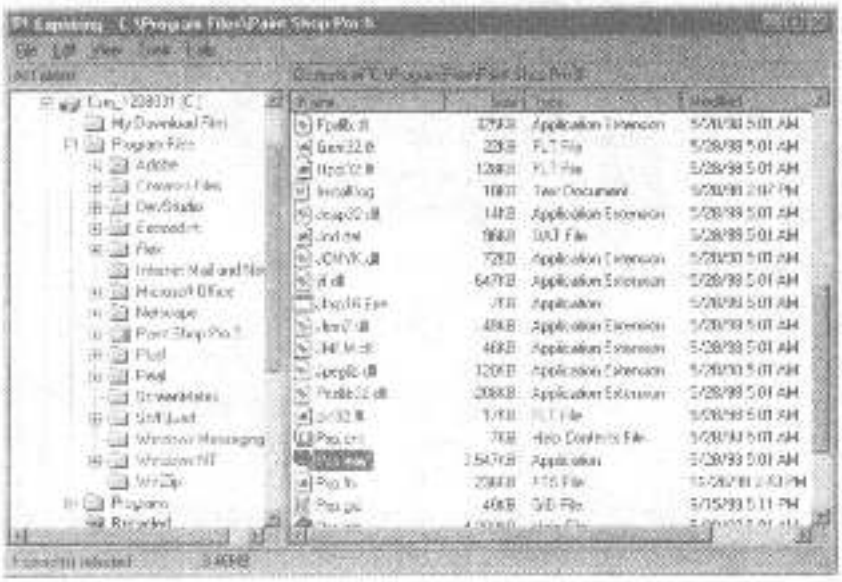


图13-2　资源管理器对PC机目录的图形化表达

```
comhilly [50] ls -alt
total 4765
drwx--x--x  76 root   system     4160 Aug  7  7:57 .
drwxr-xr-x   3 root   staff      3072 Aug  7 15:55 ..
-rwx------   1 root   system    48546 Aug  6  4:57 .checkpoint.c4
-rwx         1 root   staff     12345 Aug  5 14:55 cont.c
-rwx------   1 root   system     9635 Aug  5 14:35 jeffam.c
 rwx         1 root   system     9525 Aug  5 14:31 jeffam.o
-rwx------   1 root   staff      5144 Aug  5 14:32 a.out
-rwx------   1 root   staff      7851 Aug  3 14:31 ch_fc.txt
 rwx         1 root   system    17690 Aug  3 13:30 ch_fc.exe
-rwx------   1 root   staff     21180 Aug  2 17:29 ctime.c
comhilly [51]
```

图13-3　使用ls命令以传统纯文本方式列出的Unix目录

通过局域网将办公室的所有设备都互相连接起来的需求，使得办公室的计算环境益发复杂。这种发展的最初动力来自于共享激光打印机或中心文件服务器，以降低成本的意愿。但是，现在这个动机已经被交换电子邮件和访问Internet的需求所替代。Unix和Windows都提供对远程资源访问的支持，用户可以像访问本地资源那样访问网络上的资源。办公室中大规模及多样化的连网需求，推动了这些功能不断向前发展。

现在，软件包都表现为**桌面上的图标**，只需在屏幕上将指针移到上面，点击鼠标按键即可启动它们。指针在屏幕上的移动如何与鼠标协同一致，并没有太大问题。软件工作的方式只能从按键的功能中推断出来。简短的在线帮助是用户惟一的信息来源，最近RAM和磁盘硬件价格的下降，已经使得程序的大小变得不那么重要。为了使软件维持表面上的简化，软件提供商投入了大量的工作，以隐藏软件的复杂性。但是，如果一系列意想不到的事情发生，从而导致系统混乱后，所有这些好的意愿都不可避免地使情况变坏，因为这种情况下，还是需要人的干预来解决这种混乱局面。随着软件的日益复杂，也许对于计算机系统体系结构的清楚了解，最终会成为办公室工作人员最基本的生存技能。

为了支持应用级的程序的开发，Windows提供了许多例程库，可以通过API（Application Programming Interface，应用编程接口）来访问它们，使用这些例程库，可以为最终用户提供方便的交互：跟踪鼠标移动、支持菜单选择、窗口大小调整和刷新，以及对鼠标按键做出恰当的反应。但是**API层面**属于程序员的领域，而不是用户的领域。

13.3 系统管理：软件安装和维护

系统管理员一般负责维护现有的系统。他们眼中的计算机系统类似于仓库，他们关注的是计算机内现有的内容，以及可以容纳新内容的剩余空间。他们一般不从头编写程序。他们需要做的大部分事情之前都曾有人完成过，系统提供商或用户团体都制作了大量的程序和工具，可以为他们所用。在这一层次上使用的抽象模型或观点，包括文件系统体系结构（如上所述），还包括文件到物理磁盘驱动器的映射。现在，文件的大小和实际保存它们的磁盘驱动器变得重要了。系统如果没有足够的磁盘空间可供使用，就会停止工作。

在处理文件时，数据常常被表示成类似于水一样流过系统。**数据流**（Data Stream）从一个程序（或处理过程）传递到另一个程序（或处理过程），完成一项任务，如图13-4中文本格式化的例子所示。尽管系统管理员大多数情况下也是通过GUI与计算机交互，但他们还需要控制台（或命令窗口）。控制台（或命令窗口）接受用户键入的命令，将它们交由外壳（shell）程序或**CLI**（Command Line Interface，命令行接口）程序执行，通过它能够更全面地控制计算机的运行。

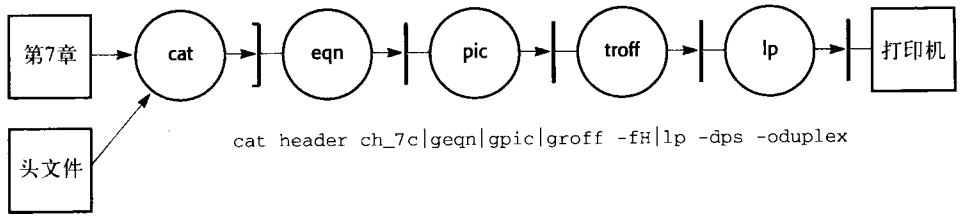


图13-4 Unix中编排和打印文本的处理流水线

尤其是Unix操作系统，它提供外壳脚本和诸多**工具**（如sort、grep和sed），使得系统管理员能够从更高层次上管理和配置整个系统。这些工具可以使用**管道**（pipe）机制组合在一起，将程序输出的数据直接送到另外的程序，而不需要存储在临时的磁盘文件中。

图13-4中简单的处理管道用来在我本地的PostScript激光打印机上打印第7章的内容。管道的第一部分，在将数据传递给一系列接通成流水线的**过滤器**（程序）之前，将头文件与含有第7章（第3版）的文件拼接起来。这些程序识别嵌入在文本内的专门的格式化命令，最后创建供打印使用的PostScript文件。lp是Unix行打印命令，它也能用在新的激光打印机上。指令“duplex”请求执行双向打印。

不同进程之间的数据流由**管道**（pipe）负责协调，管道在技术上类似于用来保存进出磁盘数据的存储缓冲区。流水线中的每个进程虽然依赖其他进程提供基本的输入数据，但它们都尽可能独立地运行。在没有数据时，进程被阻塞。更多的数据到达时，进程会尽最大努力处理数据，直到接收

到EOF（end of file，文件结束符）标志为止。我们也可以用C语言或其他语言编写自己的进程，用到流水线中。我们可以将它保存起来，用到以后的流水线中，成为一个工具程序。

```
rob@milly [80] /etc/mknod pipe1 p
rob@milly [80] /etc/mknod pipe2 p
rob@milly [80] /etc/mknod pipe3 p
rob@milly [81] ls -al pipe*
prw-------   1 rob csstaff 0 Oct 14 18:39 pipe1
prw-------   1 rob csstaff 0 Oct 14 18:39 pipe2
prw-------   1 rob csstaff 0 Oct 14 18:39 pipe3
rob@milly [82] cat letter.tmp >! pipe1 &
rob@milly [82] cat pipe1 >! pipe2 &
rob@milly [82] cat pipe2 >! pipe3 &
```

图13-5　Unix命名管道的使用

前面流水线的例子采用了“匿名”（anonymous）管道，因为管道都没有明确的名字。还有另外一种形式的管道，称为“命名”（named）管道，Unix “mknod p”和“mkfifo”命令可以示范命名管道的应用。如图13-5所示，这些命令可以在目录中创建管道缓冲区。

这个层次的计算机交互的鲜明特点，就是脚本或**命令文件**的使用。它们不过是含有击键序列的文件，在需要时可以“重放”（replayed）。由于人们发现这种机制十分有用，因而脚本语言所能够做到的事情，现在已被扩展得远远超过通过键盘直接完成的功能，最终演变成精致的，甚至十分古怪的编程语言（sh、csh和Perl）。由于它们是被逐行解释执行的，而不是通过编译执行，所以执行起来比常规的HLL程序要慢，效率也低，但就它所面向的应用来说，它依旧有很大的优势。从图13-6给出的例子可以看到，外壳脚本像使用函数一样，大量使用其他工具来执行具体的工作。有趣的是，脚本语言的调试支持比C语言、Java或BASIC要差得多。这是不是反映出系统管理员更加专业些，或者更自大呢？

```
#!/bin/sh
#
# Script converts sar data into graphs - PJN 20/10/1998
#
# Extend the PATH to include gnuplot
PATH=${PATH}:/usr/local/bin ; export PATH

# Procedure to remove non-data lines from log file.
remclutter() {
grep : | grep -v free | grep -v % | grep -v / | grep -v restarts
}
# Procedure to pad numbers with zeros to 2 digits.
padnum() {
NUM=$1
while [ `/bin/echo "${NUM}\c" | wc -c` -lt 2 ]; do
    NUM="0${NUM}"
done
echo $NUM
}
# Procedure to convert time of day timestamps to decimal days.
parsetimes() {
DAY=0
OLDHOUR=23
while read TIME DATA; do
    if [ "$DATA" = "" ]; then
       DATA="0 0 0 0 0 0 0 0 0 0 0 0"
    fi
    HOUR=`echo $TIME | cut -f1 -d:`
    MIN=`echo $TIME | cut -f2 -d:`
    if [ $HOUR -lt $OLDHOUR -a "$MIN" = "00" ]; then
       DAY=`expr $DAY + 1`
    fi
    PTIME=`expr \( \( \( $HOUR \* 60 \) + $MIN \) \* 100 \)/1440`
    PTIME=${DAY}.`padnum $PTIME`
    echo "$PTIME $DATA"
    OLDHOUR=$HOUR
done
}
# Procedure to get data from a named column.
getcol() {
tr -s ' ' '^' | cut -f1,${1} -d\^ | tr '^' ' '
}
```

图13-6　处理性能统计数据的Unix管理脚本

```
# Determine the i/p and o/p files (for last week's data).
WEEK=`date +%W`
WEEK=`expr $WEEK - 1`
if [ $WEEK -eq -1]; then
  WEEK=52
fi
WEEK=`padnum $WEEK`
DATAFILE=/var/adm/sa/sa$WEEK
OUTFILE=/tmp/$$.graphs

# Process virtual memory data from sar log.
echo "VM usage"
sar -f $DATAFILE -r > /tmp/$$.sar
cat /tmp/$$.sar | remclutter | parsetimes > /tmp/$$.sar-f
rm /tmp/$$.sar
cat /tmp/$$.sar-f | getcol 2 > /tmp/$$.freemem
cat /tmp/$$.sar-f | getcol 3 > /tmp/$$.freeswap
(cat << EOF
  set term postscript
  set time
  set xtic 0, 0.5
  set title "`hostname` virtual memory usage"
  f(x) = (x * 512 ) / 1048576
  g(x) = (x * `pagesize` ) / 1048576
  plot [0:7] []\
  "/tmp/$$.freemem" thru g(x) title "Free RAM (Mb)"with lines,\
  "/tmp/$$.freeswap" thru f(x) title "Free Swap (Mb)"with lines
  EOF
) | gnuplot > $OUTFILE
rm /tmp/$$.freemem /tmp/$$.freeswap /tmp/$$.sar-f

lp -d ps $OUTFILE
sleep 60; rm $OUTFILE
exit
```

图13-6 （续）

图13-6给出的例子是为Bourne外壳（sh）写的Unix管理脚本。它使用sar（System Activity Reporter，系统活动报告程序）获得系统的性能日志——这些日志常常写入到目录/var/adm/sa/中的文件中，对它们进行过滤，生成更直观高效的图形输出。脚本开始处定义了几个过程。在脚本的主体中会用到它们。过程remclutter()使用grep进程构成的流水线，滤去数据文件中所有不感兴趣的行。一般地，grep会抛弃不含有目标项的行，但grep -v则将含有目标项的行抛弃。

图13-6中给出的管理脚本展示出Unix脚本和常规HLL程序在许多方面的不同。脚本一般使用其他工具，比如进程流水线内的cat、grep、sed、sar和tr，来完成工作。变量在使用前不需声明。文件读写使用进程IO重定向。现在很明显的是，脚本的构成并不遵循结构化编码的指导方针。所有这些特征都是不同的，为脚本语言所特有。

Perl是一种新的脚本语言，它将之前由几个单独的实用工具提供的许多功能集成到一起，由于许多不同的操作系统包括Unix上，都能使用它，因而它取代了传统的“cat | sort | grep | awk”流水线和外壳脚本。许多程序员赞同这是一种实在的进步，因为它将许多不同的工具更有效地组织起来。但是，很明显，仅仅将传统的砖块替换成更大的混凝土预制板，并不会使城市的等次有所提高——许多摩天大厦由于不能维护只好推倒。小的总是好的吗？从另一方面讲，一种可以在所有计算机系统上使用的脚本语言，的确为用户提供很大的好处。越来越多的程序员使用Windows XP，统一脚本机制所带来的方便性是很显然的。

现今，计算机常常连接到网络中。系统管理员越来越多的职责是管理网络。因而，系统管理员所关注的领域已经从一台台分离的计算机，转变为互相连接的工作站群，常常还会涉及统一的文件

共享系统。这使得观点产生了根本性的转变，在所管理的网络属于大型公司网络的一部分的情况下，这种转变更为突出。

13.4 HLL程序员：Java、C++和BASIC

因为高级语言程序员的观点涉及到对计算机体系结构更深入的了解，所以我们将会多花一些篇幅和时间来介绍。对于那些运行复杂操作系统的计算机系统，我们几乎没有机会直接访问底层的硬件，除非费尽心思去尝试，或者是在设计会说话的下一代烤箱。至今为止，它们尚未运行Unix。由于MS Windows的广泛性，几乎所有PC程序都使用C++或C#编写。这也是我们接下来要考虑的编程类型。

有人声称程序员平均下来一天最多只能写5行能够工作、经过全面测试并且文档齐全的代码。对于那些老资格的黑客，这种估计甚至稍嫌乐观！令人意外的是，所采用的语言并不影响最终的数字。因此，使用Pascal或C编写程序，要比使用汇编语言时的效率高5～10倍，因为编译器将每行HLL代码转换成5～10行机器代码。或许HLL代码可以转换成多行机器代码，这种情况并不奇怪，但HLL表达式较之机器代码要简单得多。这种事实已经得到广泛的认知，大多数程序员都会争取采用最高级的语言来表达他们的解决方案。

HLL程序员的观点集中在所要解决的问题、使用的**算法**和方法（见图13-7），以及各种语言的词汇和语法等方面。一些HLL编译器需要固定格式的代码。Pascal要求数据声明和代码部分必须按照特定的次序，FORTRAN要求仔细地处理Tab符号，BASIC则使用行号。HLL代码的结构现在一般都分为顺序、迭代和选择（SEQ、IT和SEL），HLL使用WHILE、FOR和REPEAT循环以及IF和CASE分支，提供对这些代码结构的支持，见图13-8。

```
#include <stdio.h>

int bsort (char* pc[ ], int n ) {
  int gap, i, j;
  char* pctemp;
for (gap = n/2; gap > 0; gap /=2)
  for (i = gap; i < n; i++)
      for(j = i-gap; j>= 0; j -= gap) {
    if (strcmp (pc[j], pc[j+gap]) <=0) break;
    pctemp = pc[j]; pc[j] = pc[j+gap]; pc[j+gap] = pctemp;
  }
}

void main(void) {
  int i;
  char* names[ ] = {
      "Monday", "Tuesday", "Wednesday", "Thursday",
                  "Friday", "Saturday", "Sunday"};
  i = bsort(names, 7);
  for(i=0; i<7; i++) {
      printf("%s\n", names[i] );
  };
}
```

图13-7　HLL算法示例：冒泡排序

前面已经提到过，大多数计算机的**文件系统**都组织成层次树状结构，允许用户将相关的程序和数据文件组织在一起。这种做法类似于传统图书馆中对书籍进行分类的方法。大多数HLL程序员完全忽略磁盘数据存储的技术细节，仅仅考虑如何在文件系统内周游的问题。某种程度上，这就如同依照记忆中10年前的旧地图在轮敦驾车行驶一样！查找工具是至关重要的。在大型计算机中，即使是简单的参数：文件大小和剩余空间，现在都已完全忽略。随着个人计算机都装备以G为单位的驱动器，定期删除旧文件以释放更多空间的做法，已经逐渐失去意义。

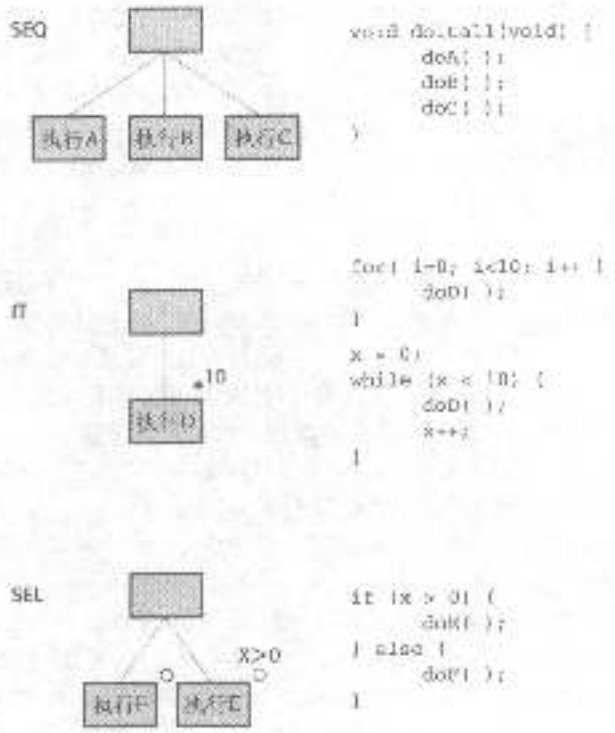


图13-8 SEQ、IT和SEL的结构图

HLL程序员认为主存提供存储程序变量所需的空间。我们可以对主存中的数据项执行写入、读出和更新操作。一般不考虑变量所占用的空间大小——只有类型声明与编译和执行的成功相关。变量类型伴有编译器认为合法的一套操作。如果代码中含有不认可的变量和操作的组合，在编译时就会出错。表13-1列出一些C++/C内建的数据类型。

表13-1 C和C++的数据类型

bool	无符号1位值
char	无符号8位值
wchar	无符号16位值
byte	无符号8位值
short	有符号16位整数
word	无符号16位整数
int	有符号32位整数
long	有符号32位整数
unsigned	无符号32位整数
dword	无符号32位整数
float	IEEE 32位实数
double	IEEE 64位实数

变量或函数代码在内存中的精确位置一般并不重要。至于CPU的寄存器，更是极少出现在HLL程序员的考虑范围之内。一些HLL程序员最初在区分局部和全局变量上存在困难，有时会导致运行时错误。这种错误在汇编级别的编程中从来不会出现。

为小型微控制器应用开发代码的程序员，依旧必须担心他们程序的大小，但幸运的是，由于第12章所述的虚拟内存机制的引入，Unix和Windows用户已经完全脱离了这种局限。

HLL程序员很少会仔细考虑数据IO。他们在考虑时，常常会将数据想像成在通道内传送。如同那些神秘的轮渡通道一样，亲人和朋友挥手后消失在安检处。他们还倾向于突然出现，常常是在错误的时间，身边还常伴有奇怪的行李。HLL IO指令一般依赖于操作系统例程执行各种操作。这使得HLL代码更容易做到与底层硬件无关，但也意味着编译器一般不能生成能够在“裸”计算机上运行的代码。同样，由于这种相关性，操作系统的变动会引起与硬件平台变动相同的问题。

HLL语言与实际计算机硬件无关的特性很有用。C语言的编译器一般是为新微处理器编写的第一个编译器。由于语言的结构与需要解决的问题相关，与CPU和内存地址没有关系，程序员能够将更

多的精力集中在问题本身，而非解决技术问题。内部指令和外部库过程之间的差异常常可以忽略。

操作系统提供的功能常常比我们想像的更有用，当编译器提供对额外命令的访问时，一些高级的程序员可能会禁不住诱惑，利用这些额外命令。这些语言使得程序员能够访问全部操作系统调用。实际上，最初使用Pascal的基本目标，可能是为了防止那些定期运行的程序造成系统崩溃！使用额外的操作系统功能，会使代码移植变得更加困难，因此应该尽可能避免。

13.5　系统编程：汇编和C

现在，底层的工作大部分属于经验丰富的程序员的领域。这些工作可能是为新设备编写设备驱动程序例程，优化图形算法或将代码集成到小型的微控制器中。我们已经指出过，一定不要忽视使用HLL语言带来的生产效率上的优势。但是，当我们的目的是学习更多有关软硬件接口知识的情况下，尝试底层的编程是最有效的方法。也许只有那些对性能极敏感的程序或者竞争激烈的游戏市场，才会完全使用汇编语言。现在更常见的做法是，仅用汇编语言编写最少量的代码，在需要时作为HLL程序的一个过程来调用。使用这种方式，至关重要的代码可以使用汇编语言手动编写，以获得最佳的效能，同时保证依旧可以使用HLL语言编写大部分代码。

使用汇编语言的另一项优点是，能够访问所有的硬件资源，HLL不能立即利用新CPU提供的新特性或功能。而汇编语言则可以立即访问任何新的寄存器或特殊的指令。同时，要全面理解HLL代码和硬件之间的关系，就肯定必须考虑作为中间层的汇编代码。编译器会检查HLL源代码的每一行，并将其转换成一系列的可执行机器指令。一些编译器还提供可选项，可以选择输出中间汇编代码文件，这对于教学活动或重要的调试活动都十分有用。基于类似的原因，我们曾看到（见7.8节），在使用Microsoft Developer Studio时，如何将汇编语言指令嵌入到C程序中。

HLL计算机程序使用各种数据结构（例如数组、堆栈和队列）来表达各种情况和事件序列。这些结构有时由硬件提供支持，有时由特定的专为实现或执行更有效率而设计的机器指令集提供支持。汇编语言文件以面向机器的方式，包括所有这些数据结构的细节：数组需要多少字节？使用哪个CPU寄存器作为指向数据记录的指针，以及执行代码的长度为多少？

汇编语言代码的格式并没有严格的规定，只有一些约定俗成的惯例。但是，由于程序员非常清楚这种自由的体制很快会导致混乱，所以局部的约定常常是必须严格遵守的。要在汇编语言程序中实现IT或SEL结构，需要显式地检查状态标志，还要使用条件分支指令跳回到标签（label）。单个IF循环需要几个互相关联的指令。为了更有效地编码，FOR循环常常倒计数至0。IF分支则需要在代码中插入目标标签。

图13-9中的示例代码为奔腾汇编助记符。提供它们是为了与图13-8中的C代码进行对比。

在使用汇编语言编写程序时，磁盘访问一般通过调用系统的例程来完成，这一点和HLL程序设计相同。但在数据传输之前，需要执行额外的初始化操作。数据缓冲区必须明确地保留和声明。误用或编程错误造成的后果会更为严重。

变量可以是字节、字或双字，CPU的数据寄存器可以临时性地存储变量。但一般的做法还是会为每个数据变量保留内存存储空间，同时赋予每个存储单元一个名称。有时，程序员必须知道这些存储单元的物理位置，但一般情况下，汇编程序会处理从存储单元标签到存储单元编号的转换，没有什么问题。

所有CPU寄存器都使用R0、D1、EAX等来引用。在第7章中，我们已经看到过奔腾的寄存器集。在编写汇编语言代码时，程序员必须跟踪程序执行期间所有CPU寄存器的使用情况。没有任何辅助手段能够协助完成这项任务，这也可能是复杂的缺陷（bug）的产生根源之一。

堆栈是系统程序员遇到的许多困难的核心。堆栈指针SP可能需要明确地初始化，从而让堆栈在RAM中正确的位置开始，并在过程调用/返回后进行检查。最困难的方面是参数的处理，在调用子例程前，参数必须以正确的次序压入到系统堆栈中，并在返回后从堆栈中移除。系统堆栈还保存所有的局部变量、过程参数和返回地址，这非常重要。堆栈溢出的问题是新软件中时有发生的问题之一。

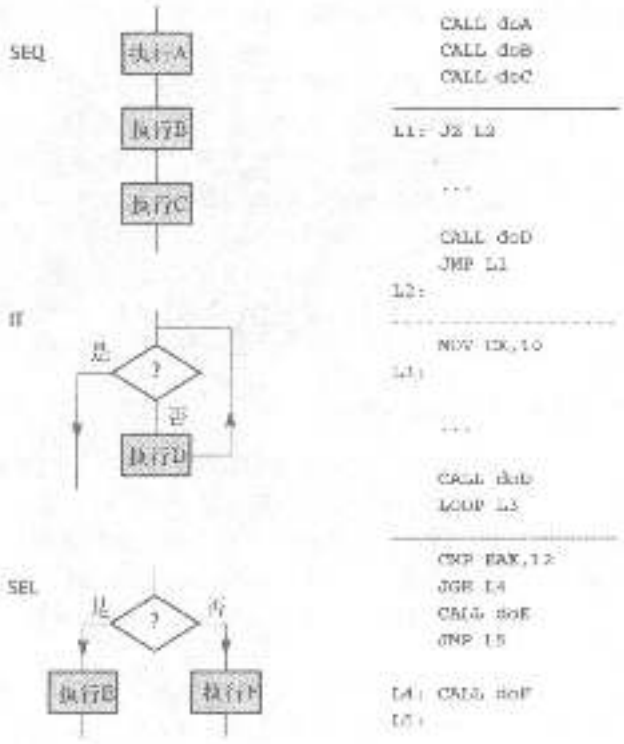


图13-9 SEQ, IT和SEL的流程图

内存地址对于汇编程序员十分重要，他们需要花些时间仔细考虑代码的起始地址和长度。但对于现代能够重定位的链接器，这种做法已不常见。物理地址（在用到时）常以十六进制表示！如果代码要安装到PROM中，那么区分PROM和RAM的地址范围，也是一项比较复杂的问题。在学习汇编语言程序设计时，一再出现的困难就是引用数组变量的方法。这可能需要涉及复杂的寻址方式，必须在最开始时就掌握。初学者常见的另一个问题来源于十六进制，数字地址和符号标签的使用。后面将会介绍更多这方面的内容。

在需要IO时，必须先找出接口芯片的绝对端口地址，以及所有内部寄存器的功能。如果没有帮助的文档，那么就需要检查硬件电路的示意图，从硬件译码器电路（见图6-13和图6-14）中推导，得出确切的端口地址。随着系统越来越复杂，操作系统，如Unix和Windows，都力图阻止程序员直接访问IO芯片。IO操作一般通过使用相应的参数调用操作系统的函数来完成，我们要做的就是等待传输的发生。在将一个程序移植到新的平台时，我曾遇到过尚无合适的编译器可供使用的情况。在注意到新的硬件拥有与老平台十分相似的CPU和UART之后，我取原来的二进制执行文件，使用十六进制编辑器搜索所有的端口IO操作。实际上这些IO操作仅仅限于getch()和putch()两个IO函数，很容易发现。直接将二进制文件中的端口地址改为新的端口地址，就可以成功地完成程序的移植。这种低层的可移植方案仅在没有操作系统调用干扰的情况下才可行。

并不需要记住每个汇编助记符！但是，在开始使用汇编语言编写程序之前，最起码要学习一些必不可少的CPU指令和寻址方式，和任何语言一样，其余部分可以逐渐学习。每个CPU都有类似的指令和方式，但它们会引发许多问题。汇编级别编程的复杂性，实际上来源于程序员必需同时考虑应用程序的算法和机器的架构。

汇编级别的程序员不可避免地要了解CPU状态标志的作用。在CPU内有一组状态位，这些状态位中，一些用来记录各种ALU运算的结果，它们的用途是将机器指令联系在一起，为跨几条机器指令的活动提供短期的存储空间。程序员有时会显式地使用这些标志（flag），但大多数时候，它们是

隐式地参与到条件指令的运算中。在此，HLL和汇编语言编程有一个根本的不同：单机器指令常常什么都完不成。

开始的初始化过程对汇编代码至关重要。所有的软件初始化工作，常常会放在一个汇编例程中，有时命名为cstart.asm。其中含有一些HLL不能完成的活动，比如设置堆栈和启动中断系统。

绝大多数情况下中断依旧是汇编程序员的圣地，因为所有对CPU寄存器的直接引用，必须在汇编级别做出。但是，一些专门的语言，比如ADA，的确为程序员提供了访问中断的机制。

以前，所有操作系统调用基本上都针对汇编程序，同时提供HLL库来处理参数列表。现在情况已经改变，现在手册一般按照C调用给出接口协议。

汇编器、链接器和载入程序的使用，看起来要比等价的HLL编译套件的使用难得多，这要归因于编译器提供商在隐藏产品的复杂性方面做出的努力。当出现问题时，复杂性就会很快出现。

13.6　硬件工程师：硬件的设计和维护

作为产品，计算机已经非常成熟，就日常的使用而言，硬件已经非常可靠，对维护工程师的需求已经大幅减少。同时，随IT市场的国际化，设计和开发新型产品的工程师越来越少。但客户依旧需要许多工程师来安装和维护设备。设计人员必须能够在数字逻辑层面将计算机可视化，维护工程师更倾向于将计算机划分成不同的可移动单元，比如插卡或磁盘驱动器。这种情况下，将计算机作为互相交换信息流的复杂单元来考虑，要比拘泥于个别的信号，关注它们的特性参数，如电压、脉冲宽度和上升时间要好得多。

确实有工程师依旧在使用分立元件和门电路，如图13-10所示，但更常见的情况是，这种电路往往被集成到集成电路的芯片上，人眼不可见。硬件设计人员依赖于CAD（Computer Aided Design，计算机辅助设计）软件包来设计和规划他们的电路，依赖于计算机模拟程序测试他们的想法。在试验电路板上制造原型电路，很快就会成为一项历史性的活动，可能只有学生才会去做！硬件设计工程师，如1.3节所述，常常使用与程序员所使用的工具十分类似的工具。当然，计算机屏幕现在比白板更重要。但硬件维护工程师并没有从这些有意义的变革中受益，他们还是得想办法找出发生故障的部件。寻找错误的活动常常退化成电路板切换的过程，这种做法对个人而言，并不会带来解决问题的成就感。

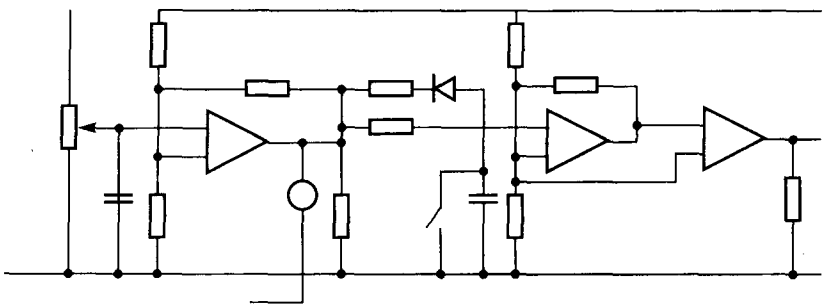


图13-10　电路示意图

13.7　分层虚拟机：体系结构简介

计算机系统由硬件和软件构成，我们可以将它表示成一系列不同功能的层。每个层接受邻近上层的命令，将它们转换成更为简化的等价指令，向下层传播。理想情况下，我们希望在顶部以英语直接发布请求，从底层的硬件获得正确的动作。我们距离这个梦想依旧遥远！解释器（或服务器）的层次结构还用在其他情况下，帮助划分和组织复杂的系统。在第15章考虑TCP/IP联网协议时，我们还会再次遇到分层的模型。

检查图13-11，它将计算机系统表示为多个“虚拟机”，我们可以根据本图来考虑各种功能在哪个层面完成。我们可以提出这样的问题，比如“浮点运算放在哪一层最好？”或“磁盘访问例程应该放在什么地方？”。过去，CISC计算机的前一代计算机（比如VAX 11和MC68000）中，将频繁发生的活动向下迁移的做法十分常见，这样可以加快执行速度，降低耗时。因此，一些活动或许会首先用HLL过程实现，实际的例子是对图形实体的操作，比如游戏中的精灵，此后，人们可能会用汇编语言将其改写，以获得更好的性能。然后，系统程序员可能会接到请求，在下一版本的操作系统中提供这个例程。这样，更多的用户就能够访问它。同样，为了提高性能，例程还可以在CPU内以微码实现，成为一条机器指令，具有自己的二进制代码和汇编语言助记符。要获得终极的速度，最后一步是设计专门的硬件来执行这个活动，对于图形运算，这可能会涉及生产次级处理器，专门处理图形图像。一些编译器甚至允许我们选择使用哪个版本的浮点算法：私有的HLL库例程、共享的操作系统例程，浮点协处理器指令或者主CPU的浮点指令。

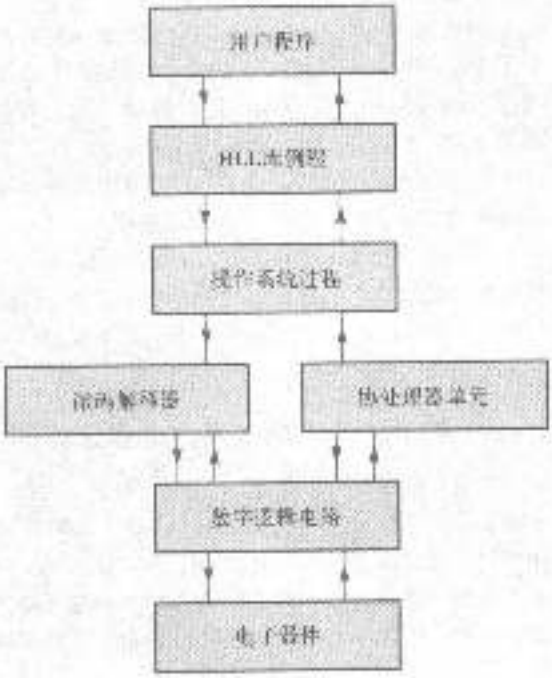


图13-11 计算机的分级结构

最近，随着RISC处理器的引入，功能又再次迁移到上面的层。我们将在第21章说明激生这种反向发展的原因。

除从上层接受命令，从下层得到信息以外，每个层还要负责开始（启动）的初始化工作。在接通电源之后，底层被唤醒，并准备就绪，但上层常常需要初始化才能就绪。操作系统代码常常存储在磁盘上，因而必须首先载入到内存中。编译器和汇编器很少能够立即使用。尽管最初的IBM PC在ROM中提供了BASIC解释器。我们可以将计算机系统想象成自底向上依次被唤醒。

* 在完成动作失败或发生错误时，虚拟机层将会向请求层返回一个错误代码。这也意味着每个请求必须有与之相关联的错误恢复方案。程序员常常会忘记这项职责，直到代码调试过程中真的发生问题之后，才不得不正视这个问题。

13.8 汇编器：简单的转换器

汇编器主要完成四项基本的转换工作，它们分别为：

1) 将指令助记符转换成二进制代码。

2) 将用户定义的符号转换成常量。

3) 数字表达方式的转换，一般为十进制到二进制。

4) 将位置标签符号转换成实际的物理地址。

汇编器在将预定义的命令助记符转换成二进制代码时，采用一种称为符号表（Symbol Table）的三列表格。符号表最初是所有的标准助记符以及它们对应的数字值，为汇编过程做好准备。随着汇编器一行行地处理源代码文件，每个新的用户符号不断连同相关联的数字值加入到符号表中。因此，start标签将会连同一个恰当的物理地址（相对于代码段起始位置的偏移）一同写入到符号表中，如果有对start的其他引用，正确的数字型地址值就可以立即得出来。

在符号尚未加入到符号表之前，第一次引入该符号（见表13-2）的情况称为“前向引用”，汇

编器可能会采用两种方式处理这种问题。它可能继续读取文件，并不断用数值更新符号表。在到达程序的结尾时，就能够发现所有的值，因而，此时就可以再次扫描源文件，完成之前尚未解决的前向引用。另外，为了避免再次扫描文件，汇编器可以在符号表中存储数值的位置插入指针，这样当数值最终确定后，就能够快速地遍历文件，将具体的数值写入代码中。

汇编过程完成后，一般会将符号表丢弃，也可以明确地指明需要保留它，供符号调试器使用。

汇编器的数制转换功能，允许程序员在处理状态寄存器时使用二进制值，引用地址时使用十六进制，处理算术运算时使用十进制。汇编器还可以转换ASCII代码值，从而让程序员避免查表的麻烦。

表13-2　符号表中的数据项

符号	类型	值
ADD	操作码	$FF
ADDA	操作码	
SUB	操作码	
MOVE	操作码	
start	已定义	
exit	未定义	
loop1	已定义	
spx	已定义	
sprite	已定义	

13.9　编译器：转换及其他诸多工作

编译C程序是一个多遍的过程，如图13-12所示。文本预处理主要完成头文件的插入（#include <stdio.h>）和常量定义（#define TMAX 100）。接下来顺序执行更复杂一些的词法分析、语法分析和代码生成。词法分析涉及将每个源文件语句划分成几个基本的标记，这些标记或者由语言预定义，或者由用户在程序中声明。编译器用代码编号来表示这些标记，并按照出现的先后顺序放入到表中。与此同时，汇编器产生符号表，协助确定逻辑符号的数字值。

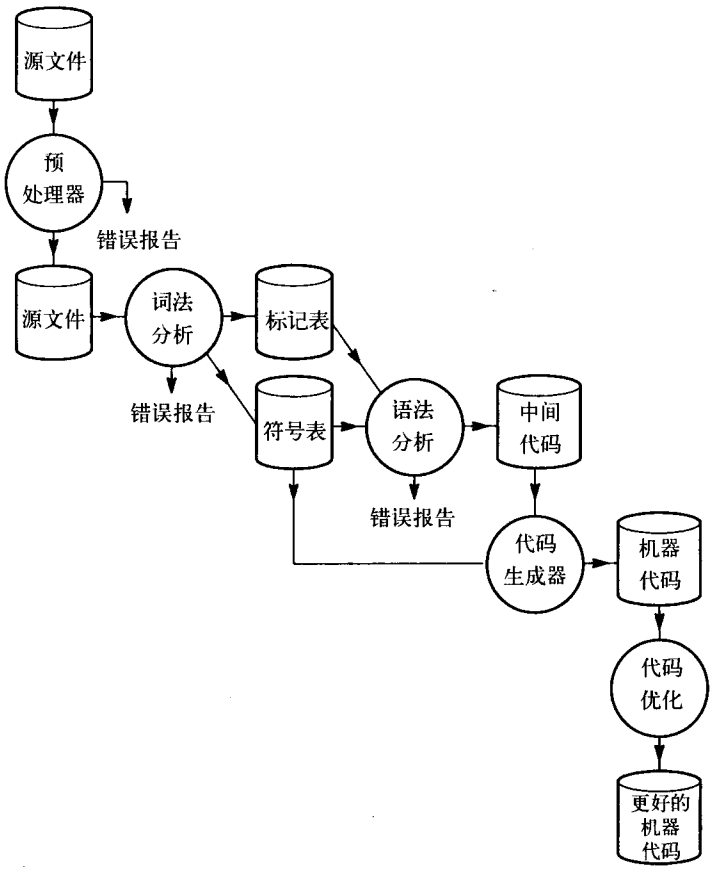


图13-12　编译过程

下一阶段是语法检查和分析，也称为解析。解析过程试图确定那些代表程序运行过程中需要执行的动作的标记组。这个阶段的输出使用中间代码格式，不针对任何特定的CPU。这种中间代码由下一阶段（代码生成）使用，转换成目标CPU所需的具体的机器代码。为了加快执行速度或降低代码大小，常常还会有进一步的处理过程，优化最终的可执行程序。编译器支持一些标志或命令行参数，程序员可以使用它们选择最佳的优化技术。

计算机系统的根本问题在于，用户表达他们需求的方式与计算机能够接受的语言之间存在巨大差异。用户使用英语或其他语言，但计算机的CPU只能理解二进制代码。翻译程序通过将一种语言转换成另一种语言，帮助使用不同语言的人进行沟通。编译器将HLL转换成机器代码，尽管当前的HLL已经十分精致且表达力很强，但依旧与自然语言存在很大的距离。编译只是一种连接“语义鸿沟”的方式，以帮助我们处理我们的意图与计算机所能够理解的指令之间的差异。

13.10 小结

- 不同的人使用计算机完成不同的任务，每个人都有不同的出发点，以及相应的专业词汇。计算机科学家必须对所有这些方面有所了解，理清它们之间的相互关系。将计算机表示成多层的系统，有助于进行这种分析。
- 应用程序的用户对计算机系统和互连网络的理解十分抽象。Windows界面减少了他们需要理解的内容，或许至少减轻了记住一些技术细节的需求。
- 计算机系统管理员维护现有的系统。他们越来越多地需要处理联网问题。他们常常编写简短的脚本程序，来协助完成日常的升级、安装、重启等繁重事务。
- HLL程序员创建新的应用程序。
- 系统程序员关心设备驱动程序和提高网络效能的高效算法。
- 参与设计计算机的硬件工程师，其工作一般为设计VLSI芯片。他们可能会使用HLL（如VHDL）来描述电路，当然在实现前还需要深入的模拟测试。
- 在描述计算机时，常常使用分层的体系构架来涵盖硬件和软件。
- HLL由编译器完成转换，而低级的汇编语言助记符则需要汇编程序来执行转换。

实习作业

如果这是新学期的第一次实习，最好首先复习一下上学期的作业，熟悉一下Microsoft Developer Studio环境。找出一些以前编写的程序，试着运行它们。完成这些任务之后，建议在Unix上试着执行一些脚本。第一个例子是基于tcsh命令行的流水线。

练习

1. 程序如何访问操作系统过程？
2. 每行汇编代码会产生多少条机器指令？每行HLL代码呢？
3. Perl是什么？什么时候会用到它？
4. CPU状态标志是什么？汇编程序会怎样使用它们？HLL程序又是如何使用CPU状态标志的呢？
5. 所有计算机语言中都存在的三种程序结构是什么？用两种不同的语言分别给出代码示例。
6. 汇编语言级别的程序设计当前的作用是什么？
7. 用户是否能够区分出用HLL编写的程序和用汇编语言编写的程序？
8. 列出使用HLL所具有的优势。
9. 系统管理员需要定期执行哪些工作？

   A. 命令行解释器是什么？举出三种不同的例子。

   B. 为什么Windows XP环境下对硬件的直接访问如此困难？办公室工作人员、系统管理员、应用程序的程序员和计算机科学的学生的需求都是相同的吗？

C. Unix管道机制是什么？主要用在什么情况下？

D. 列出现代计算机的功能（虚拟机）层次。举出计算机系统内各种不同层次中提供的功能。

E. 应用程序软件包的用户在启动程序之前，需要了解计算机的哪些方面？HLL程序员需要了解目标系统的哪些内容呢？

F. 描述CPU的哪些方面可以看做是对三种软件结构：SEQ、IT和SEL的支持。

10. 试着调试下面的C程序，使用Microsoft Developer Studio调试器的内存窗口查看IEEE浮点数的结构。

```
int main(void) {
float a = 1234.5625;
float b = -3.3125;
float c = 0.065625;
return 0;
}
```

课外读物

- Wall（1996）。
- Aho等著（1988）。
- 启动Netscape并检查下面的网站：

http://www.tizag.com/perlT

http://www.comp.leeds.ac.uk/Perl/basic.html

http://www.vectorsite.net/awk.html

http://www.cs.hmc.edu/tech_docs/qref/awk.html

- Heuring和Jordan（2004），从不同的角度看计算机。
- Patterson和Hennessy（2004），附录中有关汇编程序和链接程序的讨论。
- 这些网站可以通过本书的配套网站访问：

http://www.pearsoned.co.uk/williams

第14章　局　域　网

局域网（Local Area Network，LAN）由于可以为独计算机用户提供方便的通信和资源共享，已经变得越来越流行。PC只需插入一张网络扩展卡，就可以成为局域网工作站。以太网CSMA/CD调度方法的成功，源于它的简单，为了提高数据传输速率，最近传统的总线以太网已经逐渐被星形交换方案所取代。局域网使用的寻址方式由于必须与广域网兼容，因而会显得比较复杂。在物理器件的编号上再覆盖一个逻辑编号的方案，可为用户提供更多的灵活性。本章还将介绍如何使用socket编程进行局域网数据通信。

14.1　用户之间的纽带：电子邮件、打印机和数据库

最初为众多用户提供计算功能的是大型机（见图14-1），这些昂贵的庞然大物一般都在安全的

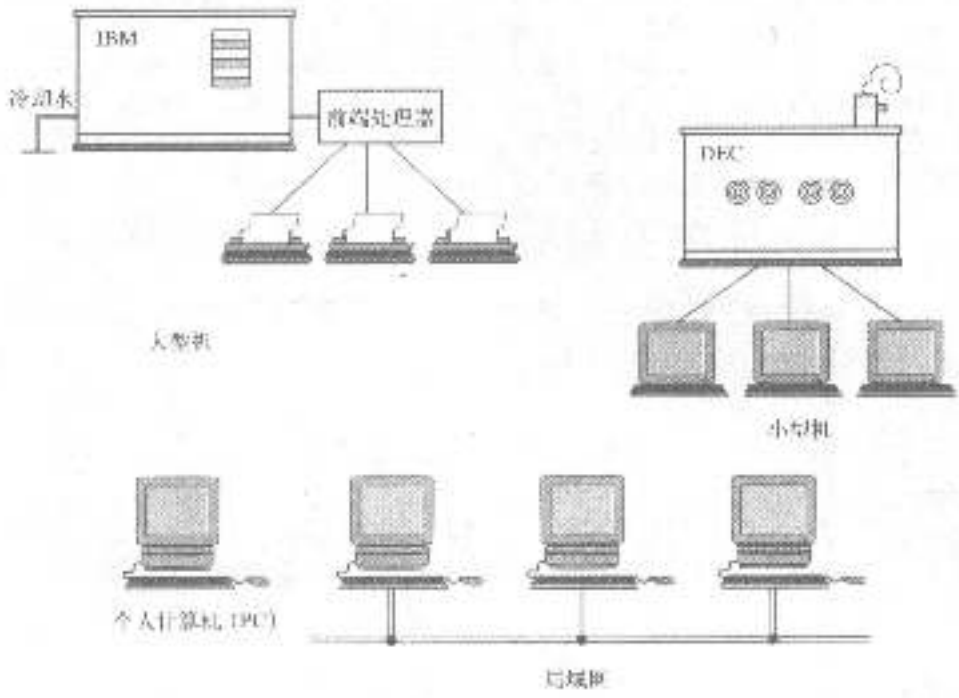


图14-1　计算环境的变迁

空调建筑物中运行，消耗大量的电力，每个主CPU单元都安装水冷管道，这让现在PC中使用的风扇相形见绌。有时，每个处理器可能会为数以百计的用户提供服务，并且常常需要几个技术人员定期维护，以保证它的正常运转。对于大多数应用，它们已经为不那么贪婪的小型机所取代。20世纪60、70年代，在PC网络再次改变计算现状之前，小型计算机逐渐成为办公室和实验室计算的最流行选择。DEC VAX-11小型计算机曾经是部门级计算的最流行选择。单个处理器服务于5到50个人的需求。这些计算机一般使用基于字符的VDU，比如DEC VT100设备，它使用专门的串行线单独连接到主机。就是在这些小型计算机上，Unix和Internet逐渐发展起来。奇怪的是，在桌面PC发展之初，共享文件和相互之间通信的明显需要被完全忽略，用户发现自己和同事分隔开来，很不方便。很快，局域网

的普及解决了这个问题，现在大多数办公室中的PC和所有的Unix工作站都将局域网接口作为基本配置。这使得大量各种设备（比如打印机、网关和CD-ROM光盘机）之间的通信得以标准化。PC一般需要在PCI扩展总线上的扩展槽内插入局域网接口卡。不同类型的线缆需要不同类型的局域网插座。Sun Unix工作站一般都在主板上提供局域网接口，现在的PC制造商也接受了这种做法。**局域网接口电路板**（在14.2节中更完整地论述）含有电子接口电路、快速的UART、RAM缓冲区和专门的微处理器，以及存储在EPROM中用来管理低级的数据传输操作的固件例程。所有这些内容组成了我们常常称之为**MAC**（Media Access Layer，介质访问层）的层，它超出了一般程序员的视野。尽管长期以来Unix都提供处理联网功能的必需软件，但Microsoft却经过一段时间之后才认识到联网软件的重要性，并考虑将其集成到主流操作系统中。在这种功能加入之前，Novell利用了Microsoft产品的盲区，成功地推出NetWare。Windows现在不仅包括TCP/IP，还提供最初由IBM委托开发的NetBIOS。NetBIOS联网协议的实现有很多，互相之间有时不完全兼容。由于这种差异性和正式标准的缺失，造成我们难以准确地描述NetBIOS，也就是现在的NetBEUI（NetBIOS Extended User Interface）。在此，重要的是要了解，NetBEUI是针对小型局域网设计的，所以不能跨广域网和Internet进行路由。

局域网技术可以追溯到20世纪70年代由夏威夷大学建立的探索性网络。这个大学需要维护各种部门以及分散在各个群岛上的大学校区之间的通信，因而他们决定开发一种基于无线电广播的新系统。这样做最初的目的是，演示信息能够通过这种手段在位于各个岛屿的校园之间有效交换，从而降低电话费用。它所基于的简单的想法是，数据发射器首先检查无线电波段是否空闲，而后启动传输。发送源一直监视发送的质量，如果另外的发射器启动，发送源会注意到这种冲突，并会尝试重新发送。这就是以太网**CSMA/CD**（Carrier Sense，Multiple Access/Collision Detect，带冲突检测的载波侦听多路访问）方案的前身，它的改进形式依旧在现今的分组无线网络中使用。

性能监视数据可以揭示局域网中包流量的变化。在图14-2中，Sun perfmeter工具展示出系统的通信量。中间的尖头信号为电子邮件，而随后巨大的波动是由于大型PDF文件跨网络传递给另一台主机造成的。

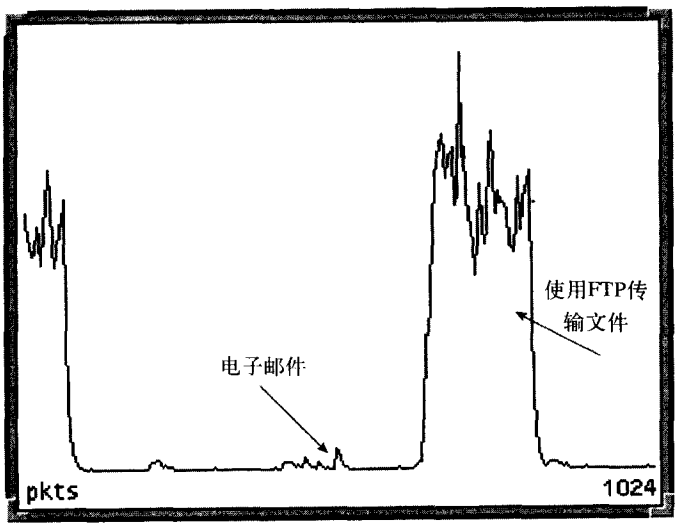


图14-2　使用Sun perfmeter监视局域网当前的通信量

20世纪80年代中期，商业用户广泛采用局域网，以获得大型机计算中资源共有的优点，这一优点曾因个人计算机把用户隔离开来而消失。随着各种各样不同类型的局域网越来越多，IEEE着手将其标准化，表14-1中列出了其中的几项。除LAN以外，现在还有其他几种类型的网络需要考虑。概括地说，根据它们的大小、访问的方式以及提供的服务，可以将它们分成**局域网**、**广域网**和**城域网**。

术语VAN（Value Added Network，增值网络）曾被用来强调通过网络能够使用的功能，而非指互连技术本身。本章介绍局域网络（LAN），第15章介绍更大的广域网络（WAN）。

表14-1 一些IEEE 802标准

802.3	CSMA/CD
802.4	令牌总线，或称标记总线
802.5	令牌网，或称标记环
802.6	MAN
802.11	无线局域网
802.12	100 Mb/s局域网

随着办公室局域网的发展，数据共享和用户间电子通信再次成为可能。通过局域网不需要个软盘携带珍贵的数据到处走动，就可以使用集中化的昂贵设备。

本章主要介绍运行TCP/IP协议的以太局域网，尽管还有一些其他类型的成功网络，比如Novell NetWare、MS NetBEUI和AppleTalk就是最知名的几种。这样选择的理由是Internet就基于TCP/IP，大多数大学和学院都拥有Unix或Windows，这两种操作系统都提供TCP/IP协议。

以太网最初的配置为共享介质的总线，所有连接的设备拥有相同的特权。这种没有特权的多路访问在流量负载超过60%后，存在严重的不足。局域网拓扑结构的最大优势（没有采用点对点连接），也造成了它的重大缺点。这是一种半双工的广播方案，所以它也继承了无线电和电视广播公司为之奋斗多年的问题。在任一时刻，一个通道只能有一个传输发生，否则就会相互干扰，破坏传送的消息。安全是个重大的问题，因为任何人都可以侦听任何会话。在几个共享资源的用户间公平地分配通道，也是一件棘手的问题。我们将会看到，一些难题已经通过从总线结构（见图14-3）转向星形局域网（见图14-4）得以解决。但是，不要认为星形结构能够完全解决单通道的问题。每个主机都有专门的连接并不能解决所有的问题。简单的集线器只能作为中心查询站，同一时刻只允许单个传输。它依靠于循环查询的方式响应所连接的主机的需求。

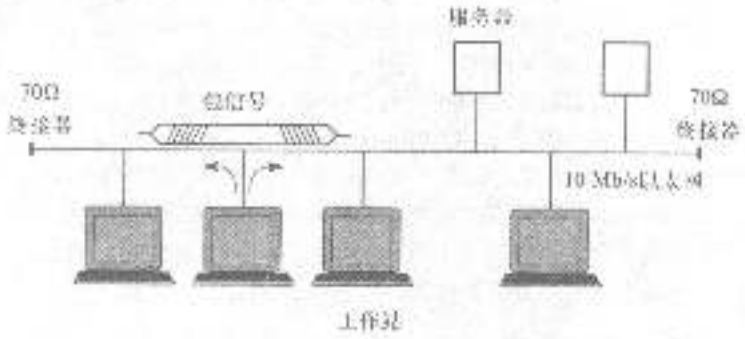


图14-3 传统的办公室总线局域网

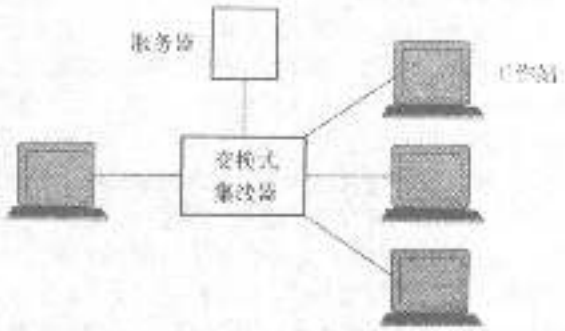


图14-4 星形拓扑、交换式集线器、以太网

集线器为工作站提供8到24个连接口，每个端口只能连接到单个设备。连接现在一般都不使用同轴电缆，而使用更为便宜的双绞线，这就是所谓的**10BaseT**标准。由两对线分别用于两个方向的传输，因而理论上的全双工交互是可行的。尽管软件可能并不能利用这种优点。在任何一个时刻，

集线器依旧只支持数据端口间的单一连接（见图14-5），但更高级的以太网交换机能够提供几路并行传输。由于同一时间几对不同端口间的会话能够同时进行，因而的确有助于提高可用带宽。几个工作站可以并行地独立进行通信；这是对局域网拓扑结构的明显改善。当然，我们将一段电缆替换成可编程的交换设备，性能必然会有所提高！集线器可以堆叠，以连接更多的主机。

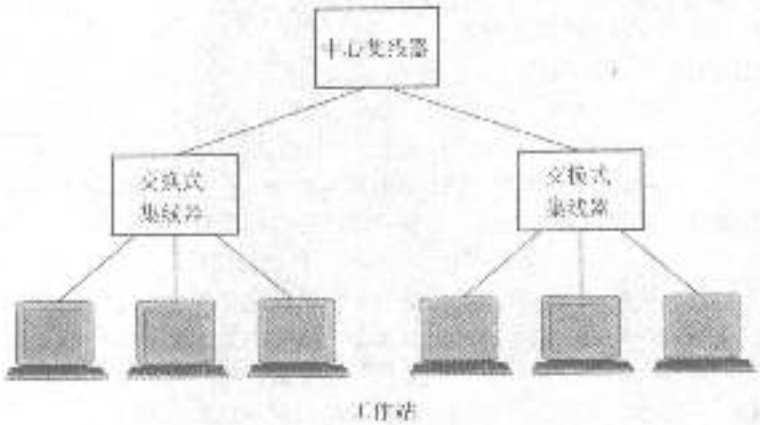


图14-5　使用分层集线器的星形拓扑结构

计算机用户已经完全接受了局域网络在共享资源上的优点。对于本地的处理工作，我们依旧保留控制，同时可以方便地访问其他的资源。快速文件传输、电子邮件分发和激光打印机共享方面的优点显而易见。由于减少使用集中化的昂贵大型机设备，从而可以减少技术支持人员的数量，降低对技能的要求。这些方面的削减，如果计划不周或缺乏维护，必然会造成网络实难数量的上升。局域网的随意扩展、CD-RW驱动器的无限制使用以及Internet浏览器的广泛使用，都会占用有限的带宽，这依旧是我们需要克服的问题。许多网络用户将分布式看做是一种苦难，管理100个用户的集中式服务器可能要比管理100台活跃的PC构成的网络要容易一些。使用大型网络的经验，以及应用Sun NFS产品跨网络透明地共享数据和程序，都印证了那句格言"天下没有免费的午餐"。网络互连给日常工作带来的好处会使整个系统对单点故障很敏感。对于普通的用户而言，如果他们发现完全依赖于某个不知名的藏在机柜中的计算机提供的服务，通常会大吃一惊。

14.2　PC网络接口：布线和接口卡

用于10Base2总线以太局域网的电缆叫做同轴电缆。它类似于用来将电视机连接到天线或有线电视插座的电缆，其结构为单个铜芯，周围是同心的屏蔽网。这意味着以太网必须以半双工模式运行，因为它没有专门的发送和接收通路。各种设备使用T形连接头连接到10Base2局域网中，如图14-7所示。尽管它比较廉价而且方便，但如果人们在没有正确处理T形连接头的情况下，就断开并移走设备时，它就会发生问题，因此，它在可靠性方面存在一些缺陷。局域网中单个故障就会使整个网络不工作。遗憾的是，发生故障时，它并不会因此拆分成两个独立的可以使用的网络！奇怪的是，这个缺点要归因于电子信号的速度（以光速的1/3传播）。在图14-3中，我们可以注意到局域网末端的终接器。如果拔去这些50Ω的电阻器，网络必定会发生故障。这是因为高速电子信号的行为类似于波，在开放的电缆端会反射回局域网中。反射的信号会干扰正在电缆中传播的信号，就如同水波一样，它们撞上海港的墙之后，反射回来与新到达的波相遇，产生复杂的干涉模型。终接器可以吸收电子能，将它转化成热，以太网的工作电压只有1.4V，不会产生太多的热量。

这种麻烦也有一些正面效应，就是电缆测试器可以根据电缆中没有正确终结的地方反射回来的信号，定位出以太网的故障所在。通过测量传播时间（到达故障处并返回），就有可能计算出信号传播的距离（速度为光速的1/3，即1×10^8m/s）。

从同轴电缆到5类多芯双绞线的变化，意味着从半双工到潜在的全双工传输的变革。数据传输可以用一对线发送，另一对接收。但应该注意的是，尽管硬件可能已经准备好支持这种变动，但软件能否立即利用这种功能，却是完全不同的另外一件事！从10Base2同轴电缆到10BaseT双绞线的转化，同样使线路错误的可能性增长。图14-6是RJ45接头的接线。10BaseT连接是两对，1+2和3+6，在正式文档中，它们被称为线对2和线对3。

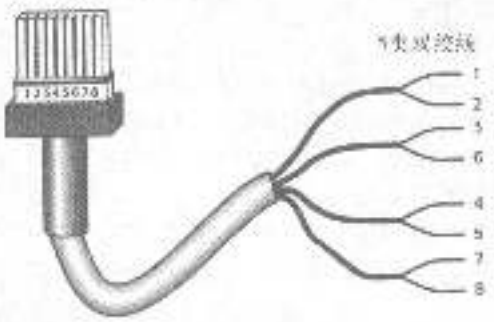


图14-6 RJ45局域网插头的连线

尽管网络连接最明显的部分是机器间的连线，但事实上，更重要的是保证消息能够正确地从源传送到目的地的软件。过去，大多数常见的局域网系统在PC上使用Novell NetWare，在Unix主机上使用TCP/IP。随着Windows NT也提供TCP/IP，情况已经有所变化。这些软件套件都定义了许多协议，也就是说，机器之间发送的消息必须按照预先规定的形式进行组织。通信要遵循一系列严格定义的规则。两套不同的协议是不兼容的。当在正确的软件时，它们可以在同一网络上同时工作。TCP/IP作为一种可靠协议，在广域网上取得了很大的成功，此外，由于它随Unix操作系统一同发布，这种优势使得它也成为局域网的标准。这种情况可能会有些不寻常，因为一种专门设计的协议，比如Novell的SPX/IPX和施乐的XNS，无疑更适用于这种领域。

Intel，AMD和摩托罗拉都提供局域网的接口芯片组，参见图14-7中的以太网卡。它们是复杂的专用处理器，这为从事低级编程的程序员提出另外一个问题。100 M位的数据速率需要使用比10.6节中介绍的RS232 UART远为复杂的硬件。最基本的问题是，要在端口的输入数据被后续的字节覆盖之前将它移走。如果用中断来完成这项任务，如COM1的UART，则它会以100 MHz的频率产生中断。这会使CPU完全没有机会从事其他活动，即使是500 MHz的奔腾处理器，也不能承受如此之高的中断频率。可行的解决方案是使用局部缓冲，而后再将输入的数据使用DMA控制器成块地传入。需要注意的是，连接到以太网不需要调制解调器，因为信号的频率就是数据的速率。这称做**基带信号传输**（Baseband Signalling），这里没有高频“载波”需要处理，不需要类似于ADSL和有线调制解调器的设备。有人可能会问，有线电视的线可以承载多个频道，为什么以太网却仅限于一个呢？这是因为电视信号都被调制到更高频率的载波信号中，有多种信号可供选择，因而能够同时在同一电缆上传送几个这样的高频信号，而不会受到太多相互干涉的困扰。这类使用无线电频率的宽带信号传输已经被ADSL和有线电视网络所采纳，在16.7节中还会进一步论述这方面的内容。它要求所有的发送器和接收器都必须配备调制器和解调制器，这样当然会增加连接的成本。

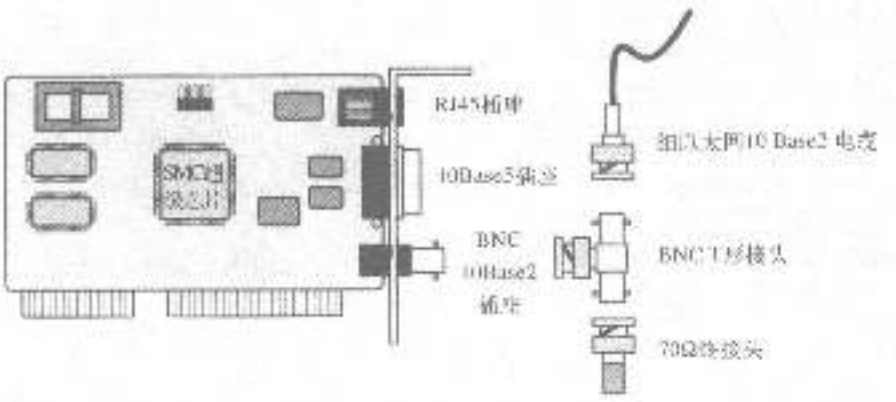


图14-7 提供10Base2和10BaseT连接头的PC网络接口卡

管理网络接口需要硬件和软件的共同参与。传统上描述网络接口的方式是按照功能分层，底层为硬件，上层为软件（见图14-8）。但这些并非绝对，在将来可能会有所变更。只有很慢的串行线路可以完全由软件来处理，因而，局域网接口也有可能成为完全由硬件驱动的设备。

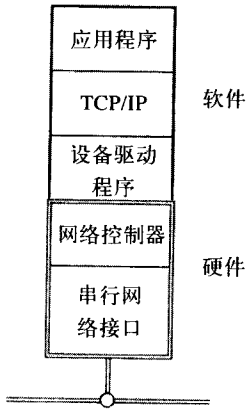


图14-8　管理局域网接口的硬件和软件层

和前面论述过的RS232串行数据传输一样，以太网也需要提供同步和流量控制问题的解决方案。流量控制将在第15章中讨论，因为它涉及到软件的TCP层，同步则可以在此处论述。

在10.2节中，我们已经解释过，接收方需要知道频率和相位信息，才能正确地捕获到达输入端口的数据信号（位）。同步的方法当然更好，但采用这种方法时，需要将发送方的时钟信号连同数据一同发送，这样接收方才能清晰地知道什么时候是位的中点。这种做法似乎需要另外的线路来承载时钟信号，但实际应用的技术是将发送方的时钟与数据流逻辑地组合在一起，然后将组合后的信号发送给接收方。这就是后来的Manchester编码，它要使用两倍于比特率的时钟。对于标准的以太网，这意味着200 MHz，如图14-9所示。

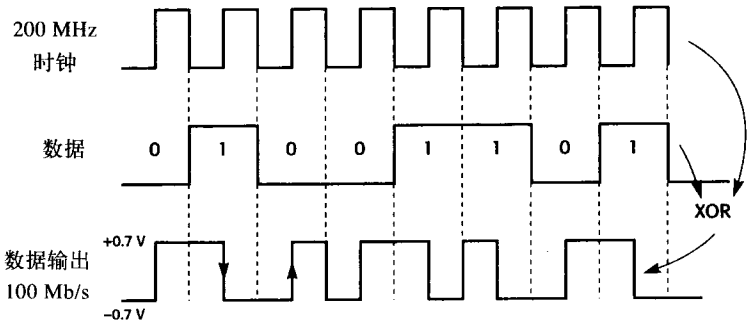


图14-9　数据和时钟的Manchester编码

待发送的位数据流在从发送方传出之前，与时钟信号进行XOR操作。最终传输的代码成为：

- 上升沿：0，下降沿：1

接收方看到的信号的边非常好，使得时钟信号的提取更加容易。提取出输入时钟的频率后，将它与本地振荡器的时钟脉冲进行比较（使用XOR门！），使用产生的差异信号调谐振荡器。它的工作十分可靠，使用这个时钟对读取进行计时，就能够精确地在位信号中间采样。如果没有采用Manchester编码，则当数据中含有连续的0或1时，会产生一个问题，因为接收方没有办法知道什么时候一个位结束，什么时候下一个位开始。这种情况类似于发送方的时钟信号被抹去的情形。为了协助接收方更好地同步，以太网在传送数据包之前，会发送由7个特殊标志字节构成的序列。

10101010 10101010 10101010 10101010 10101010 10101010 10101010

这叫做**前同步码**（preamble）模式，以太网接收方拥有专门检测8位标志序列的电路。如10.2节所述，通信都需要在三个层面同步：位、字节和消息。为了适合字节层面的同步，在包开始之前，前同步码会变为**SFD**（Start of Frame Delimiter，帧起始定界符）序列10101011。接收方将这个序列作为包即将到来的信号，并记录字节的开始位置。

10101010 10101010 10101010 10101010 10101011 [以太网数据包主体...]

↑

整个操作依赖于对SFD字节中这个起始位的可靠检测。这确实令人有些担心。前同步码并没有提供对字节同步的支持。

14.3 以太网：带冲突检测的载波侦听、多路访问

在拿起电话听筒时，我们希望听到拨号音，这是来自于本地交换机（local exchange）的一种信号，表示它已就绪，可以接受指令。如果听不到拨号音，那么此时拨号就没有任何意义，因为交换机已表示它太忙，不能提供连接（没有拨号音的故障，也可能来自于施工故障造成的电线物理中断）。电话交换机是共享的资源，但是很少遇到所有用户同时请求服务的情况。如何分配共享设备的使用时间的问题，也是局域网运行中的基本问题，解决方法一般都封装在访问协议中，对应以太网的协议为载波检测、多路访问。

以太网网络协议最初是由三家公司组成的联盟共同设计的，它们是Xerox、DEC和Intel。尽管自该标准首次发布以来（见表14-2），物理介质已经发生了变化，但最初的协议依旧保留了下来。访问协议必须能够适应简单的、没有主控设备的总线拓扑结构。它采用一种巧妙的技术避免多台工作站同时传输数据：CSMA/CD（Carrier Sense，Multiple Access/Collision Detect，带冲突检测的载波侦听、多路访问）。计算机想要发送数据包时，它会先检查线路是否空闲。如果没有数据在传送，则可以开始传输。但依旧存在两台工作站可能同时检测线路，然后同时开始发送数据的风险，这种情况下，它们发送的数据都会受到破坏。为了解决这种情形，所有工作站在传输过程中都得不断地监视线路，当检测到多于一个传输任务在进行时，则立即停下来。工作站有时甚至故意发送干扰信号，强制其他工作站意识到冲突事件，并正确地做出响应。它们随机地等待10～100ms，重新进行传输。如果再次发生冲突，则延长等待时间。没有工作站或数据包享有特权，尽管这对于所有的用户都很公平，但却使得实时语音或视频数据的传输没有保障，因为它们可能会被来自于其他工作站的传输活动阻塞，完全没办法预测这种情况的发生。

表14-2 以太网介质的各种标准

10Base5	10 Mb/s 网段最大长度500m 最小接入间隔2.5m 最多4个中继器 50Ω同轴电缆 插入式分接头	粗电缆以太网
10Base2	10 Mb/s 200m（165m）网段长度 最小接入间隔0.5m 最多4个中继器 70Ω同轴电缆 BNC T型连接器	细电缆以太网
10BaseT	10 Mb/s 100m网段长度 端到端，单工 100ΩAWG24双绞线 RJ45电信接头	交换型以太网
100BaseT	100 Mb/s 205m网段长度 端到端，单工 100ΩAWG24双绞线	
100BaseF	100 Mb/s 2000m网段长度 端到端，单工 光纤	光纤以太网

在审视冲突检测机制时，信号在电缆上的传播速度以及传输数据包所需的时间是重要的参数。光的速度是3×10^8m/s，电子信号在电线上的传输速度大约为1×10^8m/s。因此，信号只需25μs的时间，就能够从2500m局域网的一端传到另一端（2500/10^8）。选择2500m作为以太网网段的最大长度，主要考虑的是信号的衰减。电缆不是理想导体，其上的信号电压必然会随着传输而不断减小，直到接收方不能可靠地检测出它来为止。中继器可以放大信号，但必然会带来时间上的延迟。

在最后一位发送出去之前，发送方必须知道当前的数据包是否与其他数据包发生了冲突。在传输结束之前，冲突警告必须送回到发送端，否则就没有办法来挽救了。这意味着，如果我们考虑局域网中最坏的情况，两台由电缆连接在一起的相距2500m的工作站，冲突信号的返回路程是距离的两倍，据此可以计算最小数据包的长度。当其他主机加入到LAN中时，它们会使信号的质量退化，因而网段的最大长度规定为500m，最多可以使用4个中继器，以达到2500m的延伸范围。

$$T_{packet}=\frac{500\times5\times2}{1\times10^8}=50\mu s$$

$$t_{bit}=0.1\mu s$$

$$N_{1\,bit}=\frac{50}{0.1}=500\text{位}$$

$$N_{byte}=\frac{500}{8}=62.5\approx64\text{字节}$$

实践中，会将500个二进制位上舍入到64个字节。作为最大长度（2500m）10Base5以太网数据包的最小长度。从图14-10中我们可以看到，最小数据包越长，冲突检测的安全系数就越高。图中给出了最坏的情况，即第二台工作站恰恰在数据包的第一个位到达时开始发送数据。LAN上现有的任何信号当然会抑制第二台工作站开始发送。因此，危险时期是数据包开始发送和第一个位到达另一台工作站之间的区段。

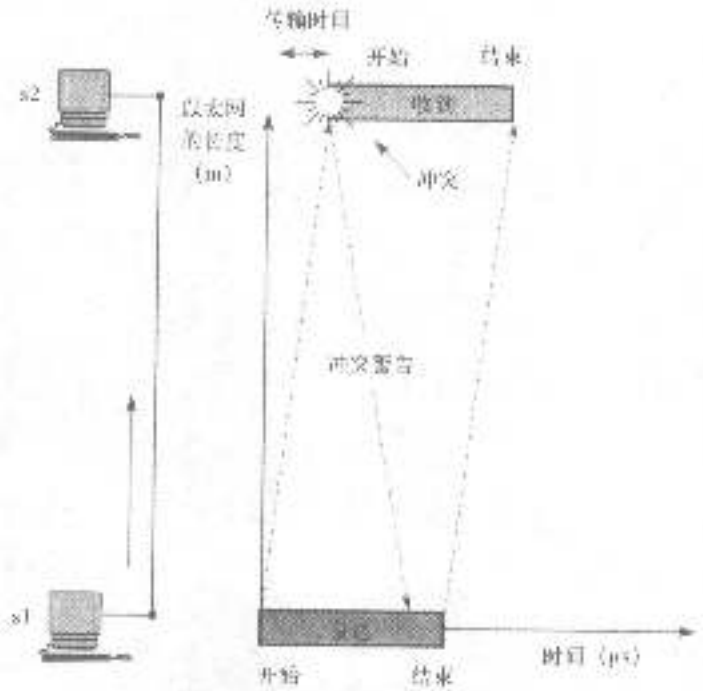


图14-10　以太网的冲突检测和传输时间

第二台工作站有可能在**传输时间**（transit time）内启动传输，为接收方制造混乱，参见图14-10。第一个数据包一经到达，第二台工作站就会意识到线路并非空闲。

100Base4T是一种值得关注的新标准，它将4对双绞线扎入一个电缆内。为了降低电缆的规格，进而压低成本，该标准选择传输速率低于30 Mb/s的线型。这造成一个问题：如何将100 Mb/s压入到30 Mb/s的线路中呢？解决方案是不采用两级信号传送，而采用三级编码。因此，我们有+0.7V、0和－0.7V，分别表示三种信号。在这里，如果我们采用三个数字符号－，0，+，我们可以将它们连同对应的二进制列在一张表中（参见表14-3）。如果我们想要使用三种信号表示任何字节（十进制0～255或二进制0000_0000～1111_1111），需要用到6个三进制数（三个位）。这是因为$3^5=243$，小于256，而下一个值是$3^6=729$，大于256，可以接受。因此，数字255可以用三个十进制数字、8个二进制位或6个三进制位来表示。这种数字编码方法叫做8B6T，很容易记！在第16章中，我们还会遇到2B1Q的编码方案，它主要用于ISDN传输。为了进一步帮助接收方，256三进制码并不是简单地从0开始，而是故意选择至少含有两个以上电压转变的数字。由于我们只需要729个可用模式中的256个，因此我们可以选择更合适的编码，将剩下的部分作为Hamming编码的“沟”，协助进行错误检测。如果对这部分内容有不清楚的部分，请参考10.3节。

表14-3 8B6T编码

二进制	三进制
0000 0000	+－00+－T
0000 0001	0+－+－+T
0000 0010	+－0+－0T
0000 0011	－0++－0T
0000 0100	－0+0+－T
0000 0101	0+－－0+T
0000 0110	+－0－0+T
0000 0111	－0+－0+T
0000 1000	－+00+－T
0000 1001	0－++－0T
0000 1010	－+0+－0T
0000 1011	+0－+－0T
0000 1100	+0－0+－T
0000 1101	0－+－0+T
0000 1110	－+0－0+T
0000 1111	+0－－0+T
…	…
1111 1111	+0－+00T

14.4 局域网的寻址：逻辑和物理方案

网络接口硬件制造商为每个网卡预设一个惟一的48位标识符或地址。这个地址有时被叫做**MAC**（Media Access Control，介质访问控制）编号。这些数字段的分配由IEEE注册授权机构监督。一般只有以太网芯片和网卡制造商才需要获得新的地址。尽管以太网地址一般在工厂内就已固定，但是，一些系统的以太网地址是可配置的，我们可以将工作站的IP编号作为以太网地址的一部分。以太网数据包报头（见图14-11）有三个字段，尾部含有一个CRC错误检验和。如果数据包的类型字段为800，则该数据包承载的是IP数据报。我们将在第15章介绍IP数据包的结构，还会介绍IP数据包中含有的IP地址对。48位MAC目的地址非常重要，因为需要用它来确定数据包转发的正确方向。在数据包经过时，以太网接收器会读入这个48位的MAC目的地址字段。如果数据包中有读取者的地址（或广播地址），则读取者会继续将它读入到输入缓冲区内。这种方案可以避免将经过的数据包全部读入，同时也不干扰CPU的任何活动。但是，如果主机使用这种方式来发现其他主机的硬件地址，则会引发许多问题，并且十分复杂。

如果以太网仅由少数几台工作站组成，可以将它们的硬件地址都手动写到一个参考文件中，然后复制到每台主机。如果数据包的地址不属于这个列表，那么它将会被忽略，或者重定向到作为**网关**（gateway）的计算机，它会将数据包传送到更广



图14-11 以太网数据包的内部结构

阔的网络中。但是，由于每天频繁的修改和硬件故障，这种解决方案很快变得不可维护。当许多局域网互相连接时，这种简单的手动方案完全不可行，因为它不能可靠地路由。

当地址与底层的硬件维系在一起时，还会发生另一个问题。以太网接口卡的更换会导致计算机的局域网标识完全改变，这种情况是不可接受的。如果每次听筒坏了之后，都得更换电话号码，人们能不能容忍这种方案呢？

为了克服前述问题，我们采用一种新的机制：引入第二套地址——实现虚拟ID。这就是我们所说的**IP**（Internet Protocol，Internet协议）编号。奇怪的是，它们的规定长度仅为32位，比物理MAC地址要短。IP地址一般以点号十进制形式来表示：164.11.9.90，每组数字（0～255）代表8个二进制位，而MAC地址则以十六进制冒号分隔的格式表示：08:00:20:8e:86:5f。本地系统管理员会为每个网络设备都分配一个IP编号。在Linux系统上，可以使用hostname -i命令查看IP编号，或是使用更通用的/sbin/ifconfig -a。在装有Windows的个人计算机上，对应的命令是ipconfig。如果以太网的硬件ID发生变化，并不会带来严重的问题，现有的IP编号会继续工作在新的网卡上。

IP协议第4版（简称IPv4）规定的编号，比如164.11.10.206，按照功能划分成四部分。顶端的几个位指定IP编号的类别，接下来是站点ID，然后是叫做子网标识的部分，最后，低端的部分标识实际的机器。理解这种字段划分，对于理解网络的配置和每个局域网网段需要分配的地址范围很重要。要记住，本地的管理员只能修改低端的子网＋主机值的部分；顶端的网络（域）部分由Internet编码授权机构，即NIC，进行分配。

图14-12列出了5类IP编码。其中最有用的是B类地址，它共有16 K个，每个可以容纳64 K台主机，人们常常将这些主机组织到256个子网中，每个子网接256台主机。遗憾的是，这些地址绝大部分都已经分配殆尽，因此，常常会发生单个组织拥有多个C类地址的情况。站点ID由集中化的部门进行管理分配，以避免地址冲突。管理网络的本地系统管理员需要设置子网值，以区分本地的不同局域网。网络/用户的划分由IP编号前面的位决定，子网/主机的划分则由本地的子网掩码（在系统初始化时装入）决定。在第15章中讲述Internet路由时，还会就相关细节展开论述。



图14-12　IPv4编号的5种形式以及它们各自的范围

但是，“目录查询”问题并不能通过IP地址的引入得以解决。如果说有什么不同的话，就是由于编号的分配不是根据地理或拓扑为准则，而是根据申请者的主机数量，从而使得情况进一步恶化！它还使需要管理的标识符加倍，同时引入动态改变的IP-MAC对。不过，到目前为止，IPv4方案工作得相当不错。每秒钟都有数以百万计的数据包通过因特网发送到世界各地。

对于小型的与外界隔离的局域网，IP地址到以太网MAC ID的映射可以静态指定。系统管理员

可以维护一个文件，发生变化后则将其重新复制到所有的工作站。在需要传输IP数据包时，可以从这个文件中使用目的IP地址获得以太网编号。遗憾的是，这种直接的方式很快就不能满足日常工作的需求。

为许多Unix系统所采用的方法是**ARP**（Address Resolution Protocol，地址解析协议）。主机自动维护一个查找表，其中保存IP和以太网（MAC）编号对。我们可以使用arp -a命令直接查看这个表（见图14-13），但是，我们可能必须找出它位于系统哪个目录中。管理员可以手动地向ARP表添加条目，但更常见的做法是，依赖于ARP进程本身维护这个表。如果在表中找不到匹配的MAC编号，它会发出一个查询包。局域网中所有邻近主机都能看到这个ARP请求包，其中含有未完成匹配的IP编号。如果任何主机识别出这个IP地址就属于自己，则会立即响应，发送自己详细的MAC地址。ARP交换的发起者就能够用返回的值更新本地的ARP表，并使用正确的地址将待发送的数据包发送出去。如果ARP请求没有收到任何应答，则将数据发送到默认网关。

```
rob@milly [20]/usr/sbin/arp -a
Net to Media Table
Device    IP Addr        Mask        Flags  Phys Addr
------------------------------------------------------
hme0      lentil    255.255.255.255     00:00:8e:06:07:cf
hme1      pb4       255.255.255.255     00:80:5f:cc:5c:20
hme0      rice      255.255.255.255     00:00:8e:06:07:e9
hme0      beans     255.255.255.255     00:00:8e:06:07:c4
hme1      ivor      255.255.255.255     08:00:20:1a:9d:16
hme0      carrot    255.255.255.255     00:00:8e:06:07:e6
hme1      router8   255.255.255.255     08:00:20:19:1c:9a
hme0      hops      255.255.255.255     00:80:8e:06:07:e3
rob@milly [21]
```

图14-13 ARP表：将IP地址转换成MAC地址

14.5 主机名：另外一个转换层

在MAC和IP编号之后，第三个名称（缩写名的使用，比如olveston.uwe.ac.uk）被引入进来，使情况更加复杂。它的意图是避免用户需要记住它们的32位IP编号。同样，此处也需要一些转换，这次是从缩略词到IP编号。人们对发送邮件给同事时使用32位的IP编号不很满意（尽管我们可以成功地管理11位数字构成的电话号码，如果用ASCII表示，是77个二进制位，如果以整数形式表示，则需要37个二进制位）。但是，记住友好的助记符较之记忆这些二进制位要简单得多。计算机可以协助维护一个查找表（主机表），其中保存IP编号和主机名对。在装配数据包时，可以从主机表中使用主机名获得目标地址。主机表必须定期更新，不管是由系统管理员手动完成（见图14-14），还是通过Internet的域名服务（Domain Naming Service，**DNS**）提供远程的域名查找服务。在第15章中会对此做进一步的介绍。

```
rob@olveston [20]cat /etc/hosts

# Internet host table
#
127.0.0.1          localhost
164.11.10.206      olveston       loghost
164.11.8.16        egg ns0
164.11.253.2       sister ns1
164.11.8.99        ada ns2
164.11.10.5        riff ns3

rob@olveston [21]
```

图14-14 主机表：将主机名转换成IP地址

14.6 分层和封装：TCP/IP软件堆栈

TCP/IP协议是一套规则和定义，它允许来自于不同提供商的计算机跨网络交换数据包。由于最初的意图是用在广域网中，因此我们将在第15章更详细地对它进行解释说明，但由于它们在要求

不那么高的局域网传输中应用得非常成功，所以我们在此对它进行简要的介绍。负责为数据通信实现TCP/IP协议的软件结构常常被组织在一个垂直的堆栈中。每层负责一种具体的协议。应用程序位于顶部，调用下面一层提供的过程。这个层从上面接收数据，对它进行重新包装，然后将它继续传递给较低的层。输入数据也采用类似的序列，但是自底向上通过TCP/IP堆栈，如图14-15所示。数据包被逐渐解开，真到最后，数据被提取出来，并传递给目标应用程序。

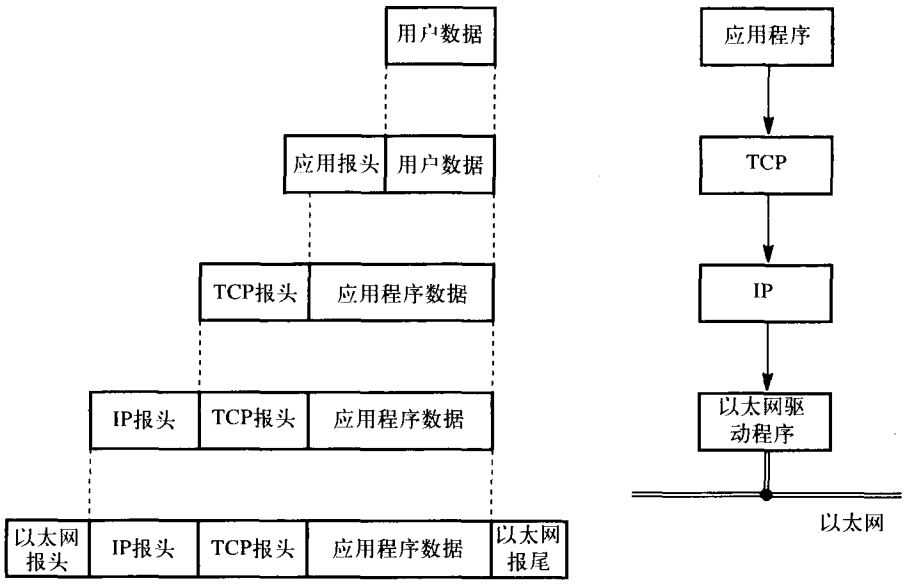


图14-15　联网软件的分层描述

14.7　网络文件系统：跨网络共享文件

Unix允许具有管理员权限的用户向根文件系统**挂接**（mount）或卸载文件系统。在安装新的驱动器时，这是扩展存储容量的基本方式。大多数Unix命令不能处理尚未正确挂接的新磁盘驱动器中的文件。在Unix上，文件/etc/mnttab保存了已挂接的文件系统的所有信息。

Sun微系统公司对这项功能进行了扩展，它引入了一种叫做**NFS**的方案，这种方案允许Unix主机以及其他通过网络连接在一起并装有恰当软件的计算机，跨网络透明地访问文件和目录。远程挂接方案并非仅仅限于运行Unix操作系统的计算机，PC-NFS使PC用户也能够将他们的文件系统集成到更宽广的结构中。这种方式依赖于已经存在的远程过程调用（Remote Procedure Call，RPC）机制，它允许运行在一台计算机上的程序启动和调用另一台计算机上的程序，并通过网络接收传回的结果。无论是系统管理员，还是普通的用户，都能够挂接远程文件系统，然后进行本地访问，这是一种实实在在的便利。它意味着我们不需要跨计算机复制文件——一切都变得不再必要，联网主机上的任何文件都可见，并且无需采用专门的传输指令，比如rcp、uucp或ftp，就可以访问它们。我们可以使用mount命令对文件系统进行检查，或使用更有效的命令df，如图14-16所示。但要注意，挂接和卸载文件系统需要管理员权限。

来自于Unix df（disc free）命令的信息，主要关系到文件系统使用的磁盘空间数量。它还揭示出工作站上所有的文件系统实际上位于什么地方。本例中Olveston的本地硬盘（c0t0d0）保存了根目录（/）、usr、var、tmp、cache和local目录，但是，我们可以看到/usr/misc目录来自于主机thalia，tutorials目录中的文件实际上保存在milly主机上。类似地，文件系统mail来自于主机mailhub，projects来自于ada，等等。NFS就是这样使得文件可以透明地通过网络访问。

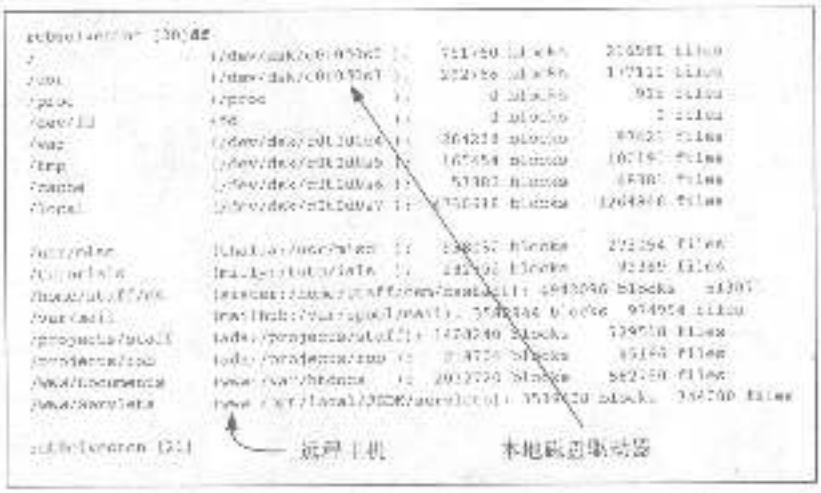


图14-16 使用df检查Unix文件系统的状态

Windows NT/XP中，通过驱动器映射选项可以获得类似的功能。这项功能允许用户将远程文件系统的某个部分指定为一个虚拟驱动器（当然首先需要具备必要的访问权限）。图14-17给出使用Windows资源管理器设置网络驱动器的画面。这项功能通过在【开始】按钮上单击鼠标右键，选择【资源管理器】→【工具】→【映射网络驱动器】获得。目录映射到机器中之后，它就如同本地的磁盘驱动器，不过访问时间会稍长一些！

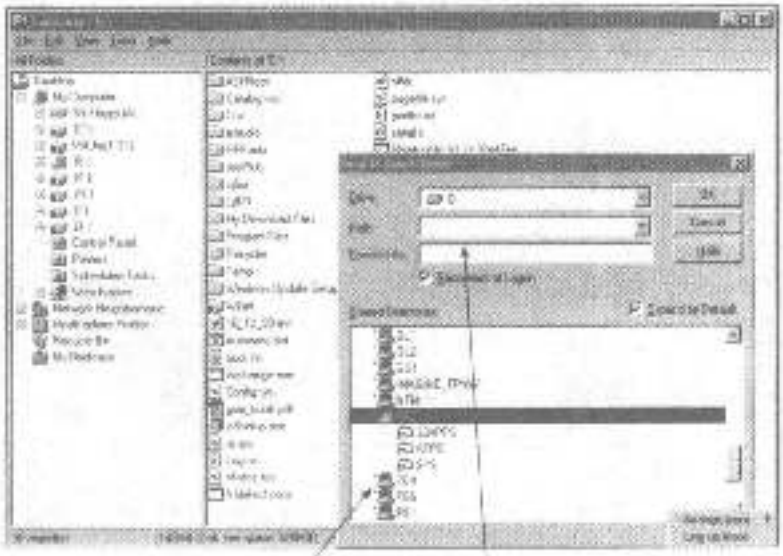


图14-17 在Windows中安装虚拟驱动器

14.8 网络的互连：网关

局域网常常会连接到其他局域网。这种连接有可能只是简单地通过将电缆物理地连接起来，把局域网扩展到另一个房间或建筑物内。但是，当互相之间的连接变得越来越复杂、或合并后的流量过大时，就必须对网络系统进行分区管理，以维护网络的效能，满足用户所需的服务。我们应该尽最大可

能地保持本地网段的独立性，避免让局域网充斥不必要的流量。当大部分的流量都是发往临近的主机或打印机时，不应该在所有网络上广播所有的数据包。这种分区技术就是网关的功能，如图14-18所示。网关是一台设备，或至少是有两个网络端口的计算机，它的工作是将有合适地址的数据包发送到另外的网络。网络上的计算机都知道这台作为网关的设备。如果数据包中含有ARP表中不存在的IP地址，则将网关的MAC地址赋予该数据包，将它发送给网关设备，期望它能够正确地处理这个数据包。

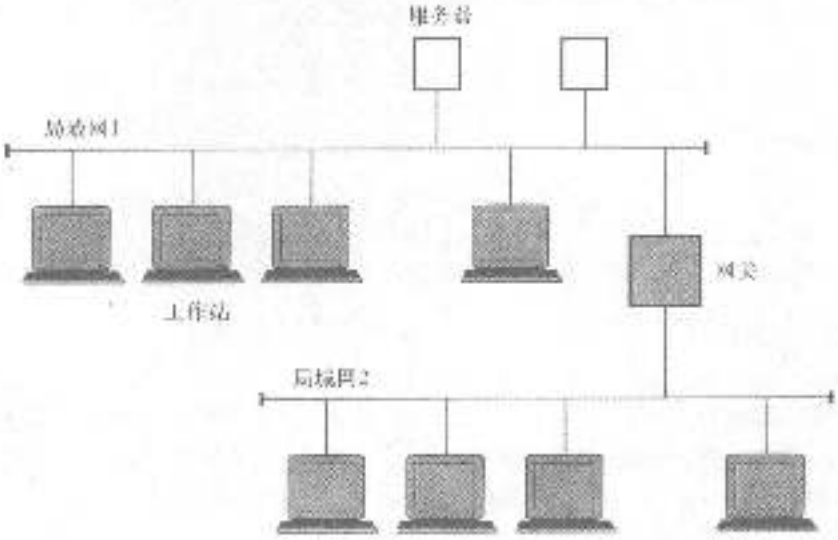


图14-18　使用网关连接总线形局域网

14.9　socket编程：WinSock简介

许多操作系统上都提供的最有用的功能之一就是socket。socket允许程序员建立不同计算机上运行的进程间的连接——前提是计算机必须通过TCP/IP网络连接在一起。由于该协议与TCP/IP定义相关联，而非专有的操作系统，因此它更有可能实现Unix上运行的程序与Windows（或其他任何操作系统）上运行的程序之间的相互通信。如图14-19所示，它提供一套通用的接口，就程序员来说，存在两种类型的socket，数据报socket允许计算机发送单个消息，而流socket支持进程间连续的数据交换或会话。

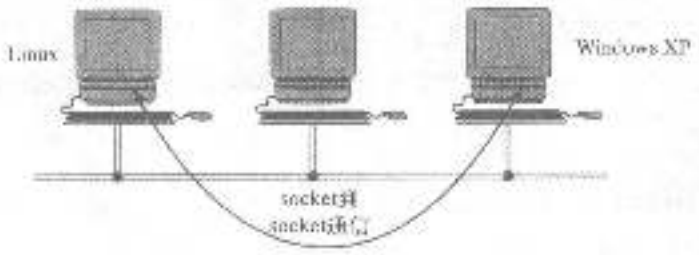


图14-19　相异系统上不同进程之间的socket通信

socket地址由两部分组成：主机的IP和标识主机上进程的端口号。端口号的作用将在第15章进一步描述。

程序员对socket的操作与文件十分类似，不过要用专门的系统调用来初始化和连接socket，传递数据并最后关闭连接。如果两个通信进程位于同一台Unix机器上，那么可以使用路径名来定位socket，但更常见的情况是，进程在不同的机器上，socket必须与机器的IP地址和一个端口号关联起来。socket与TCP和UDP端口紧密相关，在第15章中将更详细地介绍这部分内容。

由于这类通信中有两个进程参与其中，区分它们的角色十分重要。这两个进程中，一个进程作为**服务器**，另一个为**客户**。服务器等待来自于客户的请求。请求可能是读或写信息，因此不要混淆数据流的方向。客户发起；服务器响应。数据可能最终会以任一种方向流动，依客户的需求和服务器提供的服务而定。在客户成功地连接到服务器并使用它提供的功能之前，服务器必须确实存在，并事先知道地址和端口等情况。

图14-20给出的是一个流通道，作为服务器的计算机使用socket()调用创建一个公开的socket。然后，通过bind()，进程、本地IP地址和端口地址（见15.2节）被绑定在一起。这样，客户就可以使用服务器计算机的IP和端口号，建立与服务器进程之间的网络连接。服务器创建socket之后，就可以等待客户来请求服务。listen()调用表明服务器愿意接受来自于客户进程的输入连接，并乐观地打开一个队列。Accept()阻塞服务器进程，直到请求到达为止。客户使用connect()调用请求服务，服务器被唤醒后使用accept()来登记客户的地址。产生新的socket，客户和服务器之间进行快速的信息交换之后，连接通道就正式建立，可以用于数据传输，直到会话结束为止。为每个客户产生新socket的目的是释放公开的socket，使之可以接收新的连接请求。另外，使用fork()创建一个新进程来处理每个新socket的做法也很常见，在17.7节中会对此做进一步的介绍。Windows能够通过为每个新socket启动新的进程来完成类似的事情。之后，就可以使用send()和recv()函数进行数据交换，而不会影响到其他socket的活动。图14-21列出了Win32 socket函数调用及其参数清单。比起那些经过加工、尽量隐藏技术细节、用户友好的助记符来，socket函数的命名相当质朴。

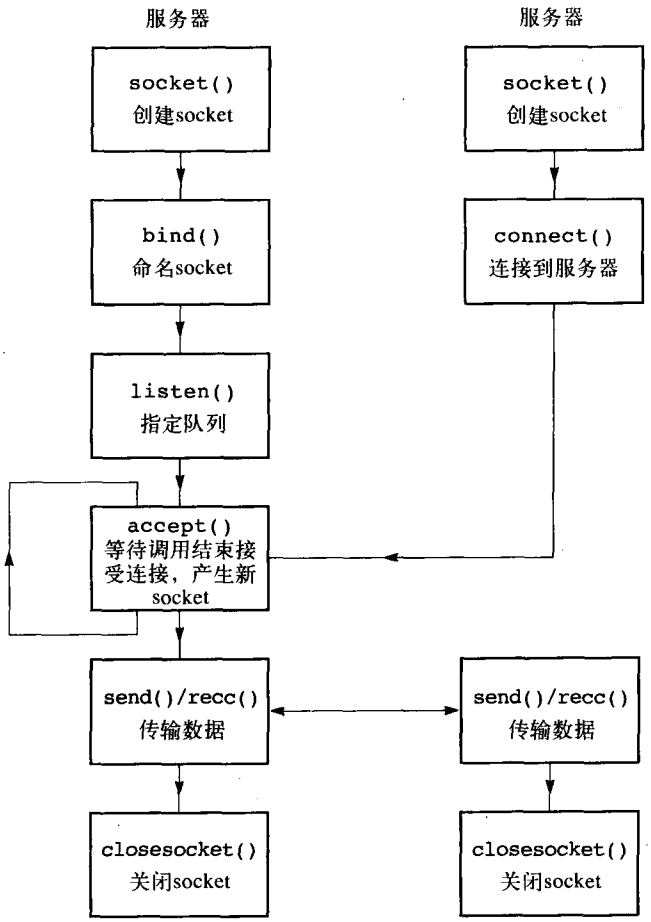


图14-20 客户–服务器方式基于连接的（流）socket通信

```
SOCKET socket(int af, int typesock, int protocol)
int bind(SOCKET mysock, const struct sockaddr *psock, int nlength)
int listen(SOCKET mysock, int qmax)
int connect(SOCKET yoursock, const struct sockaddr *sname, int nlength)
SOCKET accept(SOCKET mysock, struct sockaddr *psock, int *addrlen)
int send(SOCKET yoursock, const char *pdbuff, int dblen, int flags)
int recv(SOCKET mysock, char *pdbuff, int dblen)
int closesocket(SOCKET mysock)
```

图14-21　Win32 socket函数调用

在使用Microsoft Developer Studio为Windows编写socket代码时，最先要做的是，在源代码文件中包括头文件winsock.h，在链接时包括库文件WS2_32.lib。后一种操作通过选择Developer Studio项目选项的【Build】→【Settings】→【Object】→【Libraries】，在链接列表中加入wsock.lib来完成。不要忽视Developer Studio提供的在线帮助和Unix中相关的man帮助！图14-22给出一个稍微简单一些的socket连接的示意图，其中服务器只接收单条消息，不建立持久性的通信通道。由于每个消息都需要传递完整的目的地址，因此这里的函数调用更复杂一些。

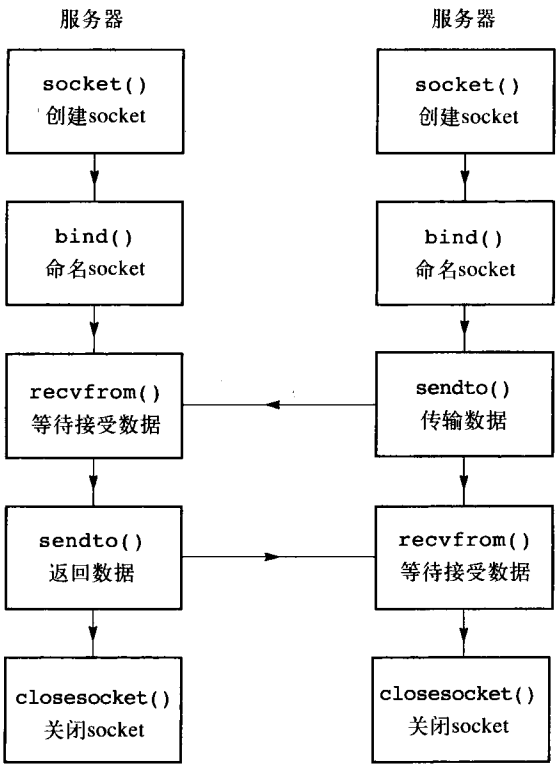


图14-22　客户-服务器方式无连接的（数据报）socket通信

14.10　小结

- 局域网（LAN）在连接个人工作站的用户上取得了巨大的成功，用户可以通过局域网交换电子邮件，受益于集中化的文件存储和更广泛的资源共享。

- 为了增加可用的带宽，传统共享总线方式的局域网拓扑结构，已经让位于层次结构的星形或树形配置。
- 局域网，包括以太网，依赖于广播方式发送数据。网络上每一台工作站可以看到所有的数据包，但它们只提取那些目的地址相符的数据包。
- PC机一般依旧需要在ISA或PCI插槽上安装网卡，才能完成网络连接。
- 以太网采用CSMA/CD网络分配策略。在流量较低的情况下，这种策略可以很好地工作，但共享的通道不能提供足够高的带宽（10/100 Mb/s）。同时，以太网也不存在数据包的优先顺序方案。
- 由硬件定义的编号带来的不方便性，可以籍由采用二级地址（IP编号）完成数据包的选址加以避免。但是，到硬件地址的转换依旧必须完成（使用ARP表）。
- TCP/IP网络协议是一套定义和规则，使用它可以将来自于不同提供商的计算机互连起来。设计它们的最初意图是应用到广域网中。
- 访问远程计算机上的文件系统是一项十分有用的网络功能。Unix和Windows都支持这项功能。
- 多个局域网可以通过我们称之为网关的设备连接在一起。它们负责转发邻近网络之间的数据包。
- 程序员可以使用socket通信函数，跨网络发送数据到远程计算机。

实习作业

我们推荐的实习作业包括访问你所在的局域网，开始时使用标准的实用程序（telnet、ftp、nfs），然后使用更为专业的工具（arp、ifconfig），最后用C语言编写简单的socket程序。这样就可以逐渐对网络产生清晰的认识。

练习

1. 列出将PC连接到局域网的优点和缺点。
2. 以太网的传输带宽（比特率）是多少？20 MB的图像文件在网络上传输，需要花费多长时间？哪些因素可以影响到最终的结果？
3. IP地址由多少个二进制位组成？以太网MAC地址由多少二进制位组成？请试着找出自己所在网络的这些地址。
4. 计算机如何将主机名转换成IP地址？它如何根据IP地址确定目的MAC地址呢？
5. 术语“基带信号传输”（baseband signalling）的含义是什么？这种方案需要调制解调器设备吗？基带信号传输的优点和缺点是什么？
6. 以太网中，由于源工作站同时广播时钟和数据信息，因而它使用的传输方式为同步传输。在只有一条线路可供使用的10Base2中，是如何做到同步传输的呢？依赖于单个位来判断数据包的开始是否明智呢？
7. 以太网使用什么接收方同步技术？使用哪种流量控制方法？
8. 10 Mb/s以太网最大的网段长度是多少？什么参数决定了这个数值？这个长度又是如何计算的呢？
9. 列出总线和星形这两种以太网拓扑技术的优点和缺点。阅读有关另外一种布局的资料：环形。
10. 试验后，回答下面这些Unix网络实用程序的用途是什么？可以从Unix在线手册man开始。

```
/usr/sbin/arp -a
/usr/sbin/ifconfig -a
telnet
ftp
```

使用XP的命令提示符窗口试着运行这些命令（此时使用ipconfig，而非ifconfig）。

11. TCP/IP协议适合用在局域网中吗？采用它作为一种局域网协议的主要原因是什么？Windows提

供TCP/IP后，Novell NetWare会消失吗？

12. 光导纤维给网络带来什么好处？光传输得比电子信号快，这会增加数据传输的速率吗？如果不能，为什么光纤分布式数据接口（Fibre Distributed Data Interface，FDDI）拥有比其他网络更高的带宽？高速并行总线的作用是否被弱化了？
13. 图14-12中介绍了IPv4编码方案。A类、B类和C类分别可以在单个IP域（网络）内容纳不同数量的主机。各种类别分别能够提供多少IP地址呢？每种类别分别适用于什么样的组织？IPv6会怎样划分它的2^{128}个编码呢？
14. 在局域网上，使用包监视软件，比如perfmeter，查看包的流量。
15. 勾画出如何对数据0001 1011 0110 1101进行Manchester编码。从0还是从1开始，会有影响吗？

课外读物

- Comer（2003）。
- Hodson（1997），一个不错的介绍性课本。
- Stevens（1994）。
- Stevens（1998）。
- 下面是一些有用的介绍性页面：

http://compnetworking.about.com/od/itinformationtechnology/1/aa083100a.htm

http://www.tangentsoft.net/wskfaq

- Heuring和Jordan（2004），关于局域网方面的内容。
- 下面是业界领先的一家设备提供商提供的介绍性页面：

http://www.cisco.com/en/US/support/inex.html

- Unix socket编程的完整介绍，可以参见下面的网页：

http://www.scit.wlv.ac.uk/~jphb/comms/sockets.html

- 这些网站可以通过本书的配套网站访问：

http://www.pearsoned.co.uk/williams

第15章 广 域 网

广域网的发展由美国的ARPAnet和NSFnet开始，但很快就发展到局域网技术领域，并迅速扩展到其他领域。TCP/IP协议成为网络技术标准化和传播的重要组成部分。Unix在TCP/IP的发展中起到了至关重要的作用，因为很早以前Unix就开始绑定TCP/IP。数据包要想通过“未知”网络进行传送，需要动态路由技术的支持。Internet上的主机使用一种独立的命名方案，我们需要从域名到IP编号的转换。这是通过使用分布式数据库DNS来完成的。万维网将众多用户带入到Internet中。搜索引擎提供基本的索引和查找功能，没有它就不能高效地使用万维网。

15.1 Internet的起源

广域网（wide area network，**WAN**）最近才为商业和技术领域所瞩目。政府已经意识到自身在基于国际**Internet**的通信设施建设中应该起到的核心作用。这种基础设施依赖于数以千计的各种网络的互相连接，由数以百万计的计算机组成。现今，将大量的计算机连接起来，除需要硬件链路以外，还需要专门的软件。万维网（World Wide Web，**WWW**）在开放Internet资源方面取得了显著的成功，现在个人和商业组织才刚刚开始通过各种方式利用这种资源。

所有这些都从ARPAnet开始。在20世纪60年代，美国国防部为研究广域网而投资建立了ARPAnet。当时，美国很担心核打击可能会使所有传统的通信方式失效，部分是因为电话系统依赖于大型的易受攻击的交换中心。因此，他们开始投入大量的金钱和人力，研究如何构建分散的、有弹性的、自配置的通信网络。在ARPAnet建成并运行之后，立即成为学术界日益流行的、交换当前信息和研究杂谈的方式。在意识到它的潜力之后，国家科学基金会（National Science Foundation，NSF）看到了这类网络的优点，开始在20世纪70年代中期建立用于非军事用途的NSFnet。这个网络围绕快速的核心网络构建，连接美国六所主要大学的计算机中心：Boulder、Champaign、Ithaca、Pittsburgh、Princeton和San Diego。硬件和通信软件都是基于运作ARPAnet所获得的经验。在Carnegie Mellon大学安装了一台网关，连接ARPAnet和NSFnet。但是，由于这套系统是政府投资的，主要的目的是支持研究工作，因此，商业组织不能使用它或开发这个资源。考虑到商业领域的需求，这个基本的方案被再次修订。这也就是当今的Internet，由大型电信公司负责为所有的用户运营主交换网络中心及互连干线。另一种商业的方案发展起来，不同的组织提供多种不同的干线，所有主要的站点间都通过这些干线连接在一起。基于冷战和战争设计的网络，现在用来在线订购比萨和联系遍布全球的朋友。只有美国才能在如此短的时间内完成这种惊人的转变。

电话网络（**PSTN**）——尽管最初是为了传输语音信号，但用于数据传输已经有较长时间。在电话网络上的数据传输，采用调制解调器设备将来自于计算机的数字信号转换成适合于电话网络的模拟电压。这样做效率不高，同时数据通信的频率必须限制在300～3300 Hz语音频宽以内。相当奇怪的是，许多长距离的干线现在都使用数字传输技术，以让多个呼叫者共享它们的宽带通道。但是，数据服务并没有从中受益：我们依旧得使用调制解调器，因为本地线路依旧是模拟的，尽管56 kb/s的调解调器确实利用了数字传输基础设施，如10.10节所述。Unix社团从20世纪70年代晚期，就开始使用拨号电话连接进行数据传输，在不同的站点间传递新闻以及传输Unix的最新版本。uucp（unix to unix copy）工具组就是为这种目的编写的。Internet主干线现在一般由ISP以及主要的交换中心之间的专用高容量链路（见图15-1）构成，因而，我们的局域网现在是由高容量的使用光纤的干线链接在一起，而非最初的拨号电话线或调制解调器。

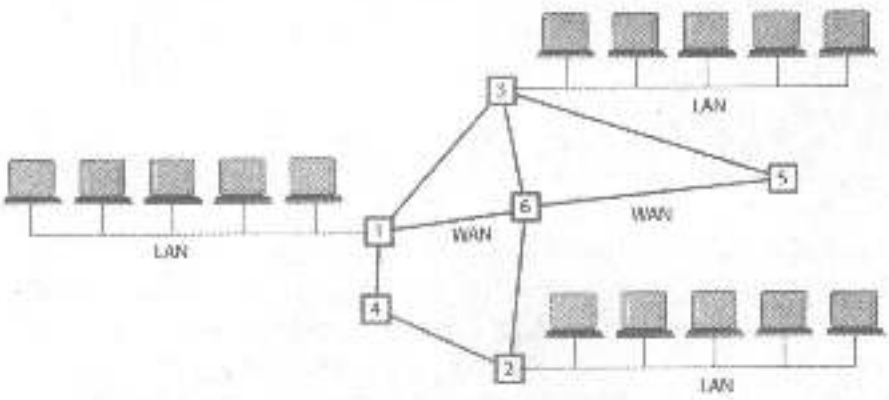


图15-1　广域网为局域网提供长距离互连

整个PSTN，包括本地的用户回路，都会逐渐向数字转化，以将数据和语音服务完全集成在一起（ISDN）。在英国，英国电信已提供System-X以达到这种目的，但新型的**DSL**（Digital Subscriber Line，数字用户线路）设备已经开始成为新的网络目标。**ISDN**为最终用户的终端设备提供64 kb/s的双向连接，而最近的DSL技术（参见16.6节）提供1到10 Mb/s的双向连接。

除PSTN以外，还有一种专门针对数据的网络，叫做**PSS**（Packet Switched System，包交换系统）。它使得计算机和终端设备之间能够以更快的速度交换信息，远高于PSTN加调制解调器所能提供的速度。为了传输，这些消息被拆分成128个字节的数据包。链接全世界计算机的Internet部分地依赖于这种方法，来完成快速的传输以及灵活的路由。

使用PSS替代PSTN进行远距离数据传输，能够降低成本以及发生错误的频率。PSS的收费各国都不相同，可能会根据安装、租用、本地线路连接时间或数据传输的量进行收费。

PSS的全部带宽常常受限于用户到PSS交换中心的本地连接。用户可能依旧需要使用调制解调器和铜质电话线路。对于简单的“字符”终端互连，数据包的组装与拆解在PSS本地交换中心的小型机（PAD）中完成。相应地，如果安装恰当的软件，这项工作也可以在用户的设备内就地完成。

英国政府资助**JANET**（Joint Academic Network，联合科研网）的开发，它使用X25协议，并不完全属于因特网。JANET现在提供完全的TCP/IP服务，并提供连接国际网络的网关。用于万维网访问、文件传输和电子邮件递送。这个学术领域的Internet网关位于ULCC（University of London Computer Centre，伦敦大学计算机中心），它连接到**LINX**（London Internet Exchange，伦敦Internet交换中心——英国国内一个重要的数据交换中心）。

15.2　TCP/IP基本协议

通信协议是一套规则和定义，为程序员提供有效的信息，使程序员能够生产出能与其他遵守同一协议的产品可靠地进行交互的软件产品。诸如声明什么类型变量、如何发起会话以及之后如何优雅地结束会话、发生错误时怎么处理等问题，都是TCP/IP协议所要处理的问题。在设计软件以处理计算机间快速的、半自治的通信时，问题的复杂性还是要用传统的模块化方式加以解决。这就产生了图14-8和图15-2所示的优美的分层结构。我们常将它称为**TCP/IP堆栈**，在许多不同的系统上多次实现。现在可以购买函数库来构建TCP堆栈，这样可以节省大量的编码工作。

作为介绍性的论述，我们必须有所选择，并对范围有所限制。因为网络领域和TCP/IP自身已经变得非常庞大。因此我们不会花太多时间在UDP上——对应TCP的另一种服务。UDP提供数据报递送，TCP提供完整的虚电路功能。选择的依据要根据数据传输的需要。单独的数据包最好使用UDP发送，而大规模的数据流则最好使用TCP。UDP和TCP都使用IP服务来发送它们的数据。

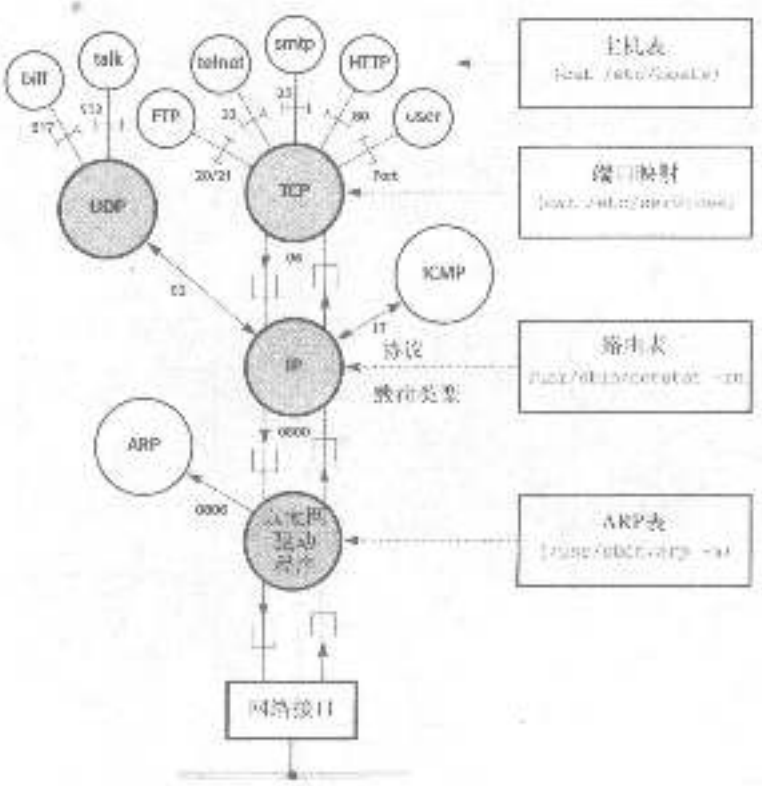


图15-2 实际中的TCP/IP堆栈

网络通信从应用程序将一些数据和目的地址向下传递给TCP层开始。应用程序通过合适的系统调用（与将数据输出到文件十分类似）完成这种动作。TCP层验证这个请求，将数据分段，进行封装，添加上含有正确端口号、序列计数的报头，然后将它们向下传递给IP层。TCP软件还要很好地处理传输问题。在预定的时间内，如果没有接收到目的主机的确认信息，则认为传输发生故障。这种情况下，必须重新发送数据。延迟计时、接收确认并将确认信息移走以及重传，都由TCP层处理。如果用户的数据太大，不能在单个包中传送时，它还会将数据切分成多个块进行传送。数据段最小可以到512字节，但大多数现代的系统都可以处理8 KB的数据包。尽管64 KB的数据包在理论上也是可行的，但并不常见。

IP层负责查找到达目的地的路径。它访问本地的路由表，选择最好的“下一跳”（next-hop）目的地，数据包将在那里重新路由。TCP帧前面为IP报头，其中含有目的IP和源地址。这些数据连同选定的下一跳路由器的IP地址都传送给MAC层，或数据链路层。IP层不关心传输的结果。为此，它被认为是一种“不可靠”服务。数据链路层与硬件及设备驱动程序交互。它还需要检索ARP表，找出与下一跳路由器的32位IP地址对应的48位MAC地址，如14.4节所述。最后，数据包与前面插入的MAC报头一同提供给以太网设备驱动例程，进行发送。处理输入数据包的过程正好相反，但更为复杂。因为在接收计算机上可能运行着许多进程，而输入的数据只能传递给其中的一个。底部的NI层会监视ARP包，只将合法的IP数据传递给IP处理过程。接下来，IP层检查协议字段，以确定将它承载的数据传送到何处。表15-1列出三个最常见的下一个目的层。参考14.9节的内容，我们能够知道UDP基于数据报socket，而TCP使用流socket。层间的数据传输可以放到整个TCP/IP软件堆栈的上下文中来分析，参见图15-2。此外，协议号在以太网数据包中的位置可以参见图15-3：“TTL最大跳计数”后面的8位字段。

表15-1 IP字段的值

数字	协议
02	ICMP
06	TCP
17	UDP

在考虑TCP/IP软件（见图15-2，常常也被称做TCP/IP堆栈）的动作时，可以结合包的结构（见图15-3）。以太网包头的内部结构并不十分复杂。在前同步码和SFD起始位之后，是6个字节的目的MAC地址，它的后面是另外6个字节，存储前一台主机的MAC地址。接下来的16位数表明包所承载的数据的长度。后面6个字节保存常数值，随后是2个字节的载荷数据类型字段。如果它的值为0800，则数据被的内容为IP包，ARP请求包的编码是0806。



图15-3　以太网、IP和TCP的封装

如果以太网数据包含有IP载荷，描述变得更为复杂。版本字段表明它是v4还是v6（很快就会出现）。下面的字段是报头的长度，给出载荷前有多少个32位字。由于它只有4位宽，因此报头最大为64字节。TOS（Type of Service，服务类型）字段的意图是提供传输的优先级，但并不常用。后面16位长的字段保存以字节为单位表示的IP包的总长度，因而最大限制为65 536字节。最大尺寸不常使用。在为了降低包的大小而对数据进行分段时，后面的序列号字段用做重构该消息。TTL（Time to Live，生存期）由路由器使用。路由器会丢弃那些已传送太远（64跳）依旧没有到达目的地的包。原始的发送者会将这个字段设为16、32或64，在包经过每个路由器时，这个值都会被减1。在将包丢弃后，路由器还会向源发送者发送消息，说明这种情况。这种状态信息将会以**ICMP**（Internet

Control Message Protocol，Internet控制消息协议）代码的形式发送。数据载荷使用的特定协议在后面的字段中（见图15-2）。报头检验和字段不涵盖数据载荷，数据载荷得使用自己的方式进行错误检测。随后是源和目的IP地址，每个占据4个字节。最后是载荷帧以及其上的数据。

看过IP之后，TCP报头就不那么可怕了。TCP报头前面的字段通过端口号给出进一步的路由信息。它们要么是**WKP**（Well-Known Port，知名端口），要么是短暂的客户程序端口。如图15-4所示，Unix可以提供活动端口号以及它们所安装的服务的清单。WKP专门用于知名的应用，比如电子邮件、ftp或万维网。它们的编号在1到255之间，现在好像已扩展到1024。再往上则是用户可以使用的端口，在创建应用时可以使用，不需要时则释放。

为了在字节层面进行滑动窗口流量控制，TCP报头中提供32位的序列号。10.4节曾介绍过这种流量控制技术。确认顺序字段（Acknowledged Sequence）是答复握手信号的一部分，它告诉发送者接收方期待的下一个数据包。再后面是4位的报头长度字段，其单位同样是32位的字，其后为6个状态标志位。Tx窗口大小提供接收端剩余的可用缓冲区空间的字节数。发送方可以使用这个字段来改变数据包的最大尺寸参数。紧急指针，如果使用的话，表示一些紧急数据的开始。最后是数据载荷，这就是TCP字段的全部结构。

```
>cat /etc/services
tcpmux      1/tcp
echo        7/tcp
echo        7/udp
discard     9/tcp    sink null
discard     9/udp    sink null
systat      11/tcp   users
ftp-data    20/tcp
ftp         21/tcp
telnet      23/tcp
smtp        25/tcp   mail
time        37/udp   timeserver
name        42/udp   nameserver
whois       43/tcp   nicname          #usually to sri-nic
gopher      70/tcp   #internet Gopher
finger      79/tcp
www         80/tcp   http             #World Wide Web
www         80/udp
hostnames   101/tcp  hostname         #usually to sri-nic
sunrpc      111/udp  rpcbind
sunrpc      111/tcp  rpcbind
```

图15-4 TCP端口号和它们的服务：/etc/services

15.3 TCP错误处理和流量控制

任何通信协议的重要职责，就是使接收方能够控制发送方发出的数据流。在RS232通信中，可以使用CTS/RTS控制线或发送Xoff/Xon代码，完成这项功能。网络也存在同样的问题：如果发送者发送数据太快，超过接收方的处理能力，接收方的缓冲区就有可能产生数据溢出。可供选择的技术有好几种。最简单的技术是，当数据在不恰当的时候到来时将其丢弃。这样就需要采用一些方法来修复这种信息的丢失。或者可以从一开始就坚持仅当接收方返回专门的确认信息给发送方之后，才继续发送下一项数据。采用这种方式时，如果接收方判断数据已被破坏，还可以积极地拒绝数据项。但它的根本目的是控制网络中的数据流量。接收方需要能够积极地确认数据包的成功接收，或者发送请求要求发送方重新传输，以及挂起数据的输入，等一切就绪后再重新开始。每种动作都需要专门的代码。此外，丢失数据包的风险总是存在。发送者对一段时间的平静应该做何反应呢？常用的策略是使用计时器，当确认信息在设定的时间内没有到达时，向发送器报警，发送方就可以重新传

输。为了避免使接收方混乱，序列中的数据包必须单独编号。从而，任何不经意的重复都可以容易地检测和矫正。或许令人吃惊的是，流量控制活动由TCP层上部的软件负责，同时还完成差错控制。相对于TCP，UDP不提供错误控制，只是简单的丢弃预料外的数据包。底层的IP层也会忽略影响数据的传输错误，并会丢弃不能传送的数据包。如果发送方能够保持所有最近传输的数据包的本地副本，还可以采用改进版的“一次一个数据包”的握手方案。采用这种方法时，传输活动会持续一段时间，即使确认信号尚未接收到，重传请求依旧可以实现，在它最终到达时，发送者会回头检查本地缓冲区中相关的数据包。仅当数据包得到确认，已成功传输后，发送缓冲区内的副本才会被删除。如果确认信号延迟，则发送方会继续发送数据包，直至本地缓冲区为未经确认的数据占满。之后，它必须停下来，等待确认信号。这种握手方案叫做**滑动窗口**，本地发送缓冲区的大小决定了窗口的大小。图15-5中，我们可以看到单个数据包成功发送并得到确认的情形，后面是同样成功的一组传输。如果接收方没有确认（ACK）后面的四个数据包，则会使发送缓冲区被未经确认的数据填满。结果是传输被暂停，直到延迟的确认（ACK）信号接收到为止。如果NAK响应到达，则发送方就得重新传送该数据包。如果通道以全双工模式工作，还可以使用更为精致的做法。可以将ACK/NAK响应附加到以相反的方向传递的数据包上，因而流量控制信号几乎不需要占用任何带宽。

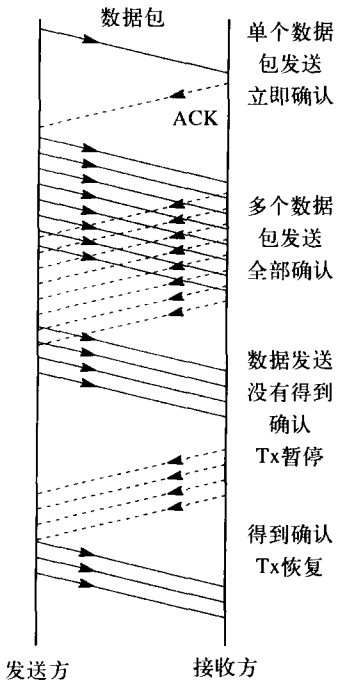


图15-5　使用四数据包缓冲区的流量控制

15.4　IP路由：数据包如何找到正确的路径

尽管从Internet的描述来看，它不过是一个网络的联合体，就好像现有局域网的简单扩展一样，但实际上，通过Internet将信息传送到目的主机时，所采用的方案和局域网有天壤之别。局域网可以采用**广播**策略——大声喊出，相应的人就可以听到——如果可能的接收者分布在世界各地，这种方案是不可行的（BBC依旧依赖这种手段进行传播）。我们必须找出一种将数据包单独路由的方法。Internet的路由是有选择性的，更像邮局（而非BBS广播电台）。我们在14.8节中已看到，如果仅存在一个默认网关，那么局域网上数据包的递送或路由就十分直接。任何目的地未知的数据包都直接定向到网关机器，然后就不再需要关心它。但是，如果局域网上存在两个或多个网关路由器，那么就必须决定选择哪一台进行路由。这就是Internet的开始。

一种可行的方法是，由发起者将一系列的IP地址插入到包头中，指定完整的路径，跨广域网从一端到另一端。这些地址必须精确地指定旅行的路线，如果任何中间地址有错误，数据包将会丢失。这种选择很少采用，因为Internet的结构十分复杂且总是在变动。更常见的技术是，由路途中的网关或路由器主机等确定实际的路径。

Unix提供一种实用工具，traceroute，它向指定名称的目的地发送探测包，试图跟踪出它们在Internet上经过的路径。这些探测包都被设定较小的TTL（Time to Live）值，从而，中间的路由主机在检测到输入的数据包超时之后，会返回错误消息。

大体上，数据包在到达网关时都要重新进行路由选择。由于长的消息会被拆分到几个数据包中进行传输，因此每个包实际采取的路线都可能不同。这会使得数据包到达目的地的次序与发送的次序产生差异。因此，TCP层必须对输入的数据包重新排列，以重构出最初的消息。IP层不关心这种细节！尽管我们认为TCP提供**虚电路**服务，但并不意味着含有TCP帧的数据包都会依次沿着相同的路线传送，只不过是TCP层透明地完成了重新定序和请求重新传送的工作而已。TCP（面向连接，虚电路）和UDP（无连接，数据报）包都使用相同的路由机制和策略。我们在第16章将会看到，

ATM网络中术语“虚电路”表示一种更可预测的东西。

Internet的核心是**基干路由器**（backbone router）。它们是从最初的NSFnet主机发展起来的，为Internet连接的所有网络维护完整的路由表。它们不记录单个主机的编号，只记录它们IP地址的网络部分。基本上，到达它们的任何数据包都应该能够定向到正确的网关，然后转发到目的网络。但是，如果所有的Internet流量都必须通过这些基干路由器来处理，系统将不能承担这样的负载，同时也易于受到单点故障的影响。这与ARPAnet的基本规范背道而驰。我们需要一种本地化的包控制方案。这种功能可以通过存储在网关或主机上动态更新的本地路由表来协助实现。

网关有几种类型，分别执行不同的功能，做出不同级别的决定。我们可以概略地将这些网关分为三个层次，它们分别处理电子信号、以太网包寻址和Internet包寻址（IP编号）。

从图15-6中给出的互连示意图中，我们可以看到互连设备中各种软件层扮演的不同角色。

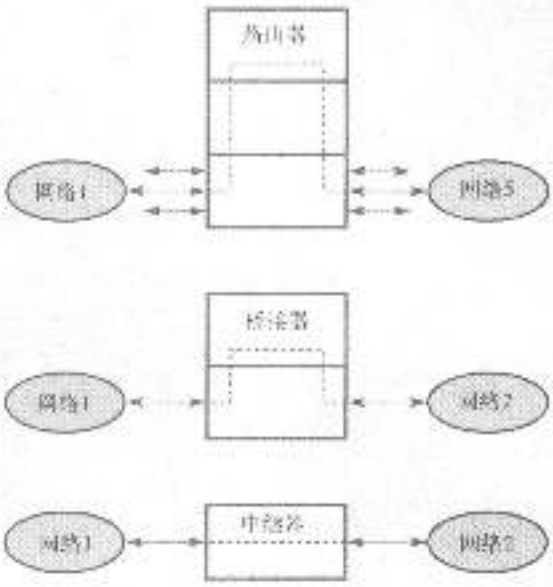


图15-6 中继器、桥接器和路由器之间的区别

中继器——如果需要完成信号放大或网络的隔离，则可以使用中继器。它只是简单地重复信号，不会理解或修改数据包承载的信息。中继器可以全速传递数据：10Base2以太网为10 Mb/s。

桥接器——桥接器会读取数据包并将它保存一段时间，仅当以太网（MAC）目的地址合适的情况下，才会将其传递给邻近网络。桥接器只能连接相同类型的局域网：以太网对以太网。它们是低级的流量过滤器，速度大约为50 000包每秒。大部分桥接器能够自动检测连接到网络的主机的MAC编号。之后，构建用于过滤操作的MAC表。只让那些目的MAC编号在对面网络找到的数据包通过。这种方法叫做生成树算法。

路由器——包会暂时读入并存储在路由器中，不同于桥接器，它使用IP层的地址，而非MAC地址。通过参考本地的路由表，路由器可以选择哪个输出端口更适合于通向目的机器。路由表必须定期更新，以维护其有效性。任何装有多个网卡的计算机，都可以运行路由程序，担当网络路由器的角色。但越来越普遍的做法是，使用专门的设备来完成这项功能。

如图15-7所示，我们可以使用**netstat -rn**命令查看Unix或Windows中的路由表。当路由表中的标志为网关（G标志），而非直接主机连接（无G标志）时，就可以取该网关的IP编号。从ARP表中获得相关联的MAC编号。数据包就以这个以太网地址作为开始，最初的目的IP地址保持不变。要注意，不要曲解H标志：它表示列出的目的地址是完整的IP地址（32位）；否则列出的地址为网络的地址——主机ID的位置均为0（零的个数依IP编号的类别而定）。这种区别在对表进行检索，寻求路由建议时有意义，因为在检索时首先会查看是否有与目的IP精确匹配的地址存在。如果有，则表示目的主机可以直接访问。然后，搜索路径到仅仅匹配地址的网络部分。最后，在均没有结果的情况下，则选择默认网关。

如果主机拥有多个以太网端口，可以选择其他端口继续传递。另一种可能是，主机接入的局域网可能支持多个网关。不管哪种情况，下一跳路由器必须使用MAC地址访问。从图15-3可以看到，数据包只含有最终目的地和始发地的IP地址，并未提供后续路由器的IP地址。这意味着选定的路由器必须在MAC层可见，并且只使用以太网编号就可以访问。这项约束就将下一跳路由器的距离限制在同一以太网网段，或至少中间仅隔一到二个桥接器！本地路由表应该为数据包的后续传输提供有效的信息，以选择临时的MAC地址。

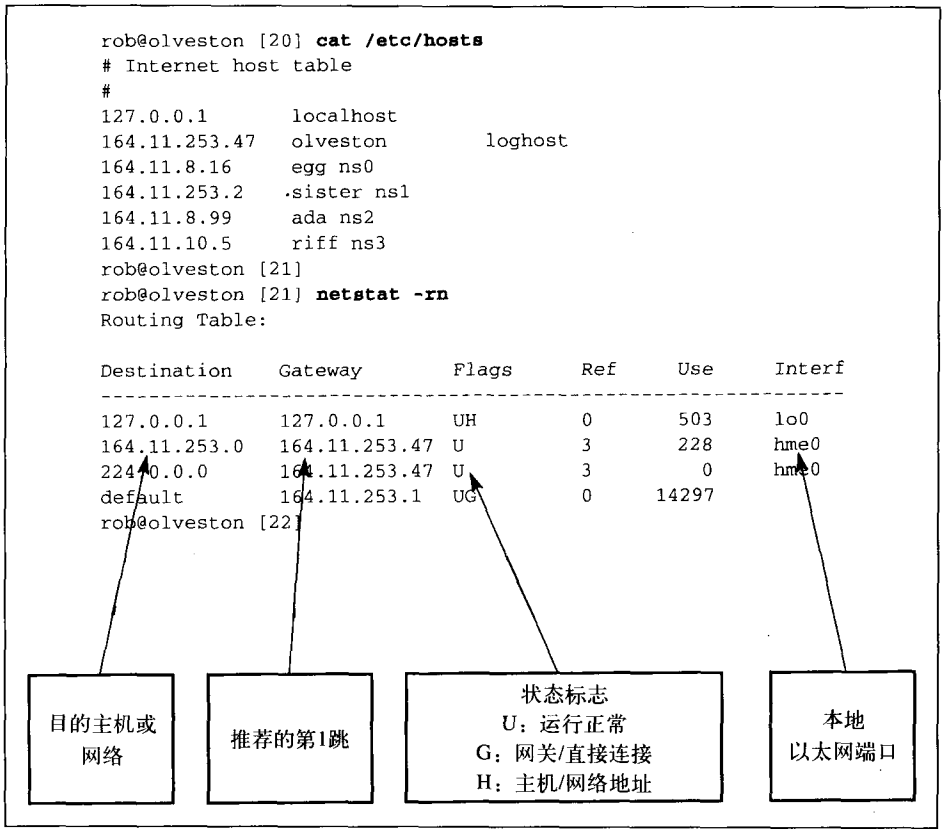


图15-7　使用Unix netstat实用工具显示路由表

路由表的维护是Internet运作的核心。该领域内有几种选项可供选择。最明显的是在启动时计算和安装一个静态的路由表。这个表不必十分庞大，因为它只需包含本地主机和网络。启动之后，默认网关就可以处理所有其他的数据包。查看图15-7中的屏幕转储，我们会注意到路由表的默认条目并不在主机文件中。因此，默认条目要么手工输入，或者动态地由路由进程产生。系统管理员可以使用实用工具route比较轻松地手动更新路由表。遗憾的是，这种想法不现实。互连环境变动如此频繁，要跟上这种变动，依赖手动修正根本不可行。另一种简单的策略是完全消除路由决定：广播所有的数据包，如同局域网。每个路由器只是盲目地将输入的数据包复制到所有的输出端口。显然，通信通道上很快就会被无用的数据包占满。有一些会发回到最初的地方，引起混淆。从图15-8我们可以看出，如果每个路由器仅将每个输入的数据包复制两份，在经过5跳之后，则会有$2^5=32$个完全相同的数据包在传递，试图传送给目的主机。如果路由器拥有三个可选的输出端口，数据包的个

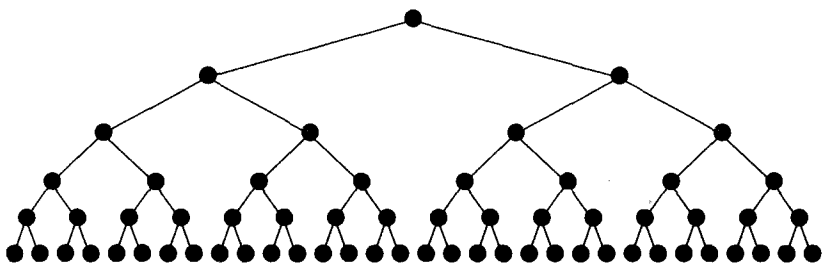


图15-8　没有路由决定会导致流量泛滥

数会增长到$3^5=243$。据说支持这种策略，一定能找出最少的跳转次数，但在插入这么多拥塞后，我怀疑这是否是最快的方式！当同一消息的多个副本同时传送时，还会引发新的问题。如果几个版本都到达期望的目的地，接收方需要能够区分出它们来。因此，必须为每个包提供惟一的编号，以使接收者能够将重复的副本丢弃。

“迷路的”数据包对网络造成的影响，可以通过固定的跳转次数后将数据包丢弃来加以限制。尽管对于广播技术它并不是一种万能药，但许多路由器依旧实行它以控制流量。数据包通过每个路由器时，**跳计数**（hop number）字段（每个数据包中都有，见图15-3，IP报头中的“TTL/最大跳计数”字段）会递减，直到这个值为零，数据包被丢弃为止。

IP编号中的最低符号位部分是主机标识符，中间部分是本地网络标识符（子网号），顶部即最高符号位为IP域编号——详细情况参见图14-12。所有广域网路由都只使用32位IP地址的顶部网络部分完成。仅在传送的最后部分，到达含有目的主机的IP域的入口网关后，才需要使用子网和主机编号。这个网关可能只是简单地在局域网上广播该数据包，或者（更有可能）将数据包传递给外部世界不可见的内部本地网关。主要的域（core domain）（见图15-12）是通过命名系统划分的，而非编号系统。对“域”一词的不同应用导致许多混淆。不同IP编码和域命名方案的共同存在有历史上的原因。由于包通过IP编号来路由，因此名字实际上并非必需。但是，大部分用户喜欢用名字来指代主机和用户。因此，Internet上专门提供执行这种转换的在线字典系统（DNS）。IPv4编号划分成不同的类别，并根据申请者的主机数量进行分配，但命名采用的策略完全不同。从一开始，命名系统就具有层次结构，并基于地理或组织的功能进行划分。

如前所述，路由器维护查找表，这样，当含有外部网络地址的数据包到达时，就可以查询这个表寻找建议。在得出邻近网关的地址之后，就可以将数据包发送到这个地址，以便进一步调度。这个表列出当前请求过的所有远程网络的最佳本地网关的IP地址。在启动时初始化后，这个表就需要定期更新。来自于其他路由器的**ICMP错误消息**对维护这个表很有帮助，当其他路由器判断数据包的路径选择有误时，就会返回这个消息。ICMP协议（在15.2节中介绍过）提供这种消息机制。路由器接收到数据包并根据路由表选择出最佳转送路线之后，如果发现数据包又被返回到它到来的端口，它就会立即向源路由器发送一个错误消息。其中会建议更合适一些的其他路由器。

另一种动态维护路由表，使之随时保持最新的方法基于**RIP**（Routing Information Protocol，路由信息协议）包（见图15-9）和Unix routed（route-demon），虽然现在依旧在使用，但也存在许多争议。在Sun Solaris系统上，它被包括在inetd中。这个方法的成长和被人接受，好像完全出于它能够工作。RIP技术涉及**请求**（solicitation）和**广告**（advertising）。在路由器启动时，它会向它的邻近路由器发出请求，要求共享它们的信息。接收到回应后，新路由器会基于该信息构建路由表。它还要响应来自于其他路由器的请求，并周期性地广播它的路由表，供其他对它感兴趣的设备使用。路由器使用的一种不太光明正大的技巧是，同时使用从请求包中得到的信息更新路由表！这就如同收集邮票时将信封上使用过的邮票揭下来，不管信封内有没有账单。

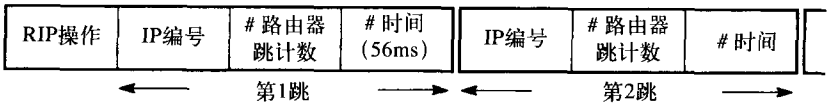


图15-9 RIP包的字段

路由表中的数据项都有时间限制。如果三分钟未更新，则会被标记为无效，短暂的延迟后，会从表中完全移走——十分粗暴的策略！

所有的路由器都有责任与最近的邻近路由器共享路由表。也可以将这种策略看做是一种缓慢地将本地存储的信息发布到网络中的方案。还记得吗，**直接连接的网络**以及一些远程的网络都表示在路由表中。这样，每个路由器都可以给予其他路由器附近网络的第一手信息，以及到远程网络的转

接时间等方面的经验。后者来自于跳计数——路由器与远程网络进行信息通信时实际参与的路由器数。使用这个数字，路由器可以排除效率低的路线，采用更好的方向。对RIP方法的批评，来自于它受到干扰（比如路由器故障）后需要很长时间才能回复，同时路由表中可能会产生环路。

一个日益增长的问题是中心路由表的大小。由于B类编号的耗尽，单个组织分配多个C类IP编号的情况越来越多。这也导致中心路由表的大小剧增，同时路由表中可能会出现数字上连续的几个条目均建议同一网关的情况。CIDR（Classless Inter-Domain Routing，无类别域间路由）的引入就是试图控制这些重要路由表的大小（见表15-2）。C类编号现在已开始参考地理位置进行发放，这样包路由会更易于执行。每个国家和地区会有各自保留的编号，十分类似于电话系统。路由器将会以不同的方式工作，不再仅仅检查IP地址的前端网络部分，而是使用整个32位地址，由指定位长度的掩码来标示需要考虑哪些位。这样，就可以将一组共2048个C类IP编号分配给一个组织。从A31到A11，从路由器的角度来看是相同的。A10到A8用于区分组织内部的子网，而A7到A0是常规的主机号，用以区分不同的工作站。CIDR路由器取32位IP编号，使用关联的掩码：255.255.248.0。

表15-2　CIDR IP编号分配

大陆	保留的IP编号
欧洲	194.000.000.000 – 195.255.255.255
北美	198.000.000.000 – 199.255.255.255
南美	200.000.000.000 – 201.255.255.255
亚太	202.000.000.000 – 203.255.255.255

随着IPv6的迫近，我们将看到一个分层的、按照地理位置进行组织的DNS方案会逐渐实现，不同的国家和商业服务提供商会得到新的128位IP编码块。也许Internet最终从PSTN的经验中学到了一些东西。

15.5　DNS：分布式域名数据库

跨Internet传输数据包，需要完成几种类型的标识符转换，如图15-10所示。

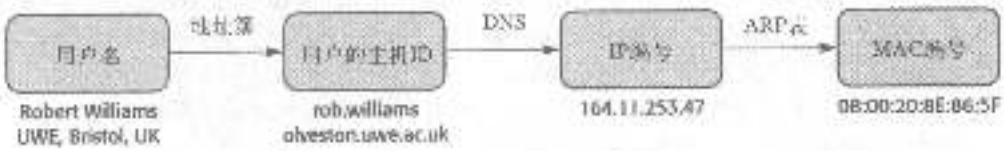


图15-10　传输方需要转换标识符

- 首先是手动地查询获得用户和主机ID。这一步仅需在传输开始前执行一次。另外两项转换则必须每次传输数据包时都得执行。

32位IP编号，即使以点号十进制数的方式表示，依旧不方便或不易记忆，因此我们一般使用别名（见图15-11）。Internet域名空间被划分成几个通用的域（.com、.edu、.org、.mil、.net和.gov）和许多国家性的域（.uk、.fr、.sp等），见图15-12。总共有13台根域名服务器负责保存权威的DNS列表。点号分隔的域名标识符的最高域名部分由主域名授权机构NIC（Network Information Center，网络信息中心；http://www.nic.xxx）颁布。这种权限还被分配给各个国家中小一些的域名组织。Network Solutions公司当前在美国发布全球性域名的授权许可。Nominet公司（http://www.nic.uk）负责管理.uk域，UKERA处理gov.ac和ac.uk子域内的域名申请工作。对于商业名称的使用存在不断的争论，因为它们不受版权法的保护。在一个众所周知的标识符被发现之前先垄断它，已成为一项小型的产业。

```
rob@n11v[13]ypcat hosts | more
164.11.13.5          gecko
164.11.5.80          ttdy
164.11.235.32        scar
164.11.243.225       valdoorican
164.11.13.56         starfish56
164.11.11.73         blackwell
164.11.253.47        olveston
164.11.8.203         dialin63
164.11.8.200         dialin60
164.11.11.15         wallaby
164.11.235.87        sharnon
164.11.253.138       new_gb2
164.11.253.249       oaggara
164.11.154.4         linux04
164.11.10.45         orpineo
164.11.11.71         wesley
164.11.235.122       elpaso
--more--
```

图15-11　查看本地的主机表

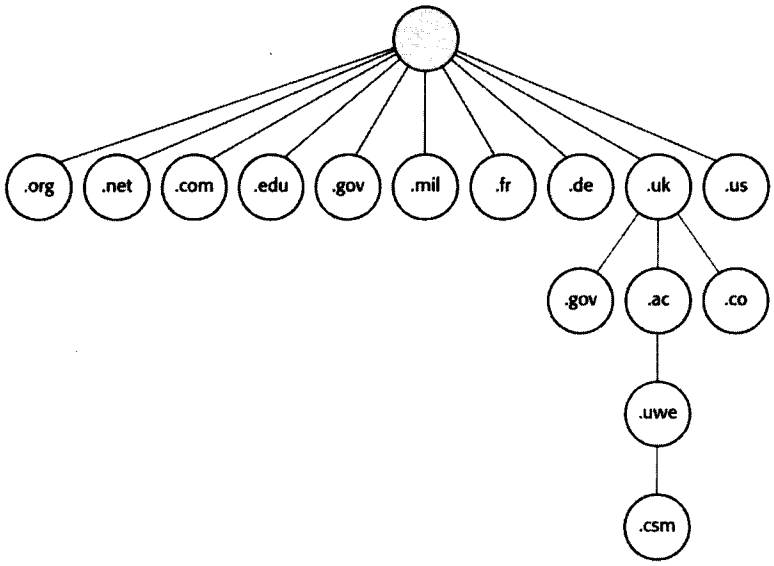


图15-12 域名的层次命名结构

最后，本地网络管理员需要指定域名中余下的部分，对本地的域进行合理的划分。通过这种方式，网络管理员可以决定将网络拆分成多少个子网，以及每个子网将会接入多少台计算机。

值得注意的是，IPv4的主机IP编号系统基于大小分类（见图14-12）。这样就不便于按照地理位置进行分组，或是实施分层次包路由。实际上，即使要求做到这两项功能，也不可能完成。但是，主机命名方案采用了一种不同的方式，它允许**域名**可以由国家或组织的状态来控制。使用这种方案可以达到更快的搜索速度，同时有助于网络的结构化以及更有效率的路由。它实际上是效仿老的电话系统。基于某些原因，**通用**域使用三个字母，而**国家**域则限制使用两个字母。

主机名，比如我当前的名称，olveston.uve.ac.uk，可以存储在本地主机名文件中，以备引用，或者（更常见）保存在可以通过网络访问的数据库中：**DNS**（Domain Name System）。DNS是分布式的数据库，它提供在线字典服务，使用网络主机的网络名，可以查出它的IP编号。每个DNS服务器都含有它自己的域中所有机器的域名和编号。针对其他地方的查询被上传到该Internet域的根服务器。这种分层的域名服务器试图**解析**它们本地域内的任何查询。

域名当然必须惟一指定，这是域名组织者的职责，就像不同目录中的文件可以共用一个文件名一样，为了区分域名，所有对域名的引用都必须使用全路径名，完整的网名可以区分全世界名字为R Williams的用户。而同一部门中有两个人名字为R Williams则是另外一种情况（凑巧的是，的确如此，我们必须使用中间名来解决我和同事之间名字上的冲突）。

Unix命令nslookup和whois可以测试本地的域名服务器。后者的结果很难预测，它们都使用DNS域名服务器查找域、网络或主机的IP编号。C语言程序一般通过调用函数gethostbyname()访问域名服务器。它会返回一条记录，其中有指定主机的IP地址。本地的域名服务器列在/etc/resolv.conf中，我们的查询就发送到这些地方。如图15-13所示。

```
rob@milly [33] cat /etc/resolv.conf
domain csm.uwe.ac.uk
search csm.uwe.ac.uk uwe.ac.uk
nameserver 164.11.8.16
nameserver 164.11.253.2
nameserver 164.11.253.11
nameserver 164.11.8.99

rob@milly [33] c/usr/sbin/nslookup
  Default Server: egg.csm.uwe.ac.uk
  Address: 164.11.8.16

> smilodon.cs.wisc.edu
  Server: egg.csm.uwe.ac.uk
  Address: 164.11.8.16

  Non-authoritative answer:
  Name: smilodon.cs.wisc.edu
  Address: 128.105.11.80

> ^D
rob@milly[35]
```

图15-13 使用DNS域名查找功能

DNS域名服务器能够独立地工作（只要情况允许），

曾经有一次，位于美国的7台核心路由主机之一（herndon.com）发生故障，完全不能工作，这次事件导致所有发往.com域名的电子邮件都被拒收。对于分布式网络，由于它的本意是创建能够抵御核打击的通信机制，所以这个灾难使许多用户都十分震惊。

15.6　万维网的起源

近年来，世界上重大的发展之一就是Internet以及万维网（World Wide Web，**WWW**）的快速增长。尽管这是已有技术的合成产生的效果，但它的发展却出乎许多人的意料。前面我们已经说过，Internet是网络的网络，从最初由美国国防部在20世纪70年代早期投资建立的ARPAnet开始，一直稳步地增长。逐渐地，越来越多来自于全球不同国家的网络连接到一起，结果是形成了现在的因特网。由于这种革命性的方式，Internet的拓扑是不规则的，但它依旧允许位于全世界的计算机彼此之间互通信息。

万维网从Tim Berners-Lee在CERN的研究开始。万维网使用超文本协议，为我们提供一种组织、访问和检索存储在许多不同计算机中的信息的方法。超文本是指内嵌指向其他文档文件的链接或指针的文本，所引用的文件可以在同一台计算机上，也可以在网络中其他的计算机上。以这种方式访问一个文件和FTP传输并无不同，在万维网出现之前，FTP已经广泛应用了许多年。Apple Macintosh计算机已经将超文本技术作为另一种用户友好的数据库使用多年。应用万维网技术可以产生文档网络，层次体系比较松散。“超媒体”一词指的是扩展到其他形式媒体的超文本文档，比如图像和声音。

万维网的基础是**HTML**（Hypertext Markup Language，超文本标记语言）以及**HTTP**（Hypertext Transmission Protocol，超文本传输协议）的使用。万维网没有以PostScript或PDF存储和传输文档，而是使用简单一些的方案，更类似于Unix nroff和troff实用工具。文本文档以纯ASCII编码，同时嵌入额外的格式编排指令作为控制序列。在需要查看这些文本时，浏览程序会重新编排该文档，遵照控制代码显示其内容。通过这种方式，超文本文档不依赖于任何特定的能力和功能。另外，创建者还可以利用超文本提供的诸多功能在文档中嵌入对其他文档、图像和声音的引用指针。超文本或超媒体存储在服务器计算机上，如果希望别人能够访问这些信息的话，一般需要将服务器链接到Internet中。同时，必须为每个文档分配一个惟一的地址——URL（Uniform Resource Locator，统一资源定位），正是URL将资源链接到一起。实际上，URL更像是查找指令，而非文件地址。万维网的用户需要使用称为Web浏览器的客户端软件，对超文本解码。Netscape Navigator和Microsoft Internet Explorer都是当前最广为人知的产品（或许，假以时日，Gnu会提供能够兼容Web的emacs）。

Java编程语言的发展对大学中计算机科学相关课程的影响很大。它的发展史起伏不定。尽管最初设计它的意图是作为家用设备的控制语言，类似于Forth语言，但从事万维网应用开发的程序员对它表现出极大的热心。它类似于C++，但避免了一些比较危险的缺陷。它从其他编程语言借鉴了许多概念，但当前它最与众不同的特性在于它需要解释程序——能够产生本地机器代码的Java编译器现在依旧是少之又少。但是，由于运行系统较小，且可以容易地移植到其他新的平台，因此它的应用依旧在日益增长。它与万维网的关联性来自于最新版本的Netscape Navigator和Internet Explorer都提供Java语言的解释程序。因而，Java代码可以像超文本那样容易地下载和运行。这为跨Internet发布程序提供了一种简单的方式，同时能够为静态Web页面提供更出色的动态功能。

这些（以及其他）发展意味着我们可以认为计算的未来与网络以及跨Internet的分布式计算紧密相关。新计算机的架构已经做出调整以适应这种新需求。人们对于支持多媒体（不过是更好的图像和更高品质的声音输出）表现出极大的热忱。许多家用PC现在都提供电话调制解调器，简化连接到万维网的工作。在Sun决定在工作站的主板上集成网络接口25年之后，PC主板也开始集成网络接口。

15.7　浏览Web：Netscape Navigator

Netscape Navigator是一种Web浏览器，如图15-14所示。我们可以使用它搜索万维网上的新文档，在屏幕上查看它们的内容。文档可以从本地硬盘驱动器获得，也可以通过网络从远程服务器上

下载。Navigator针对这些应用进行开发，它具备相应的功能，能够与其他程序交互，交换信息以及解释和显示接收到的文档。在访问Web页面之前，必须首先获得URL。URL由二部分组成：

URL = 协议://机器名/文件路径

比如：

http://www.cems.uwe.ac.uk/~rwilliam/CSA_sheets/sockets.pdf

尽管它们和文件路径名十分相似，但它们并非简单的位置说明符。

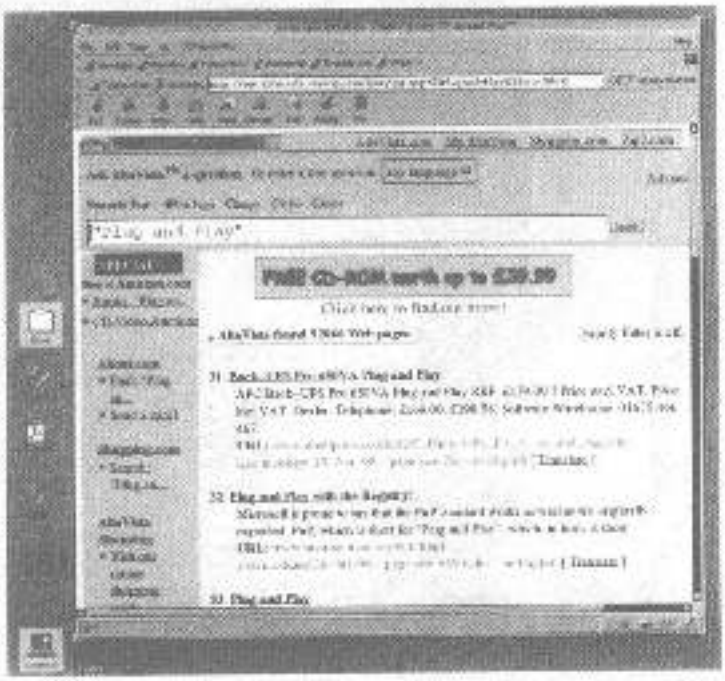


图15-14 Netscape Navigator Web浏览器

由于HTML是Web页面编排的标准，因此浏览器必须提供能够解释HTML的解释器，以生成用户可以查看的屏幕图像。现在大多数浏览器还能够处理纯文本、PostScript和PDF文件。如果曾经制作过Web页面，那么对HTML不会感到陌生。它由一系列叫做HTML标签（tag）的格式编排命令组成。它们被嵌入在文本中，这样接收方可以根据本地的条件重新编排这个页面，这样做使得浏览者可以充分地利用本地的资源。不同计算机在屏幕分辨率、色彩能力和字体文件上都有可能存在差异，接收文本将它以恰当的方式显示出来，而非简单地在屏幕上重画由位图图像文件提供的图像。这为文档的分发提供了一种极为灵活的方式。使用文本标记的方式分发文档比提供PDF文件更灵活。最类似的方案是Unix troff语言（本书最初就是使用这种方式）。HTML功能不算强大，相比于PostScript，其表现能力也有相当的差距，但它的优势在于更为紧凑。作为语言，它易于学习，甚至许多Web制作者可以使用专门的所见即所得（**WYSIWYG**）编辑器来制作他们的页面，根本就不需要查看底层的HTML代码。

图15-15屏幕的上部为emacs编辑器窗口，其中显示的是"Hello World!"的HTML代码。这个文件（world.html）由Netscape Navigator读入，它会解释尖括号"<…>"内的格式控制标签。将文本编排后显示给用户。非标签符号则简单地作为文本显示在Netscape的显示窗口内。表15-3列出基本的HTML文档标签。浏览器和Web服务器，尽管初衷是处理HTML文档，现今功能已增强，具备启动声音和视频文件的能力。浏览器需要识别用以表明特殊类型的数据到达的专用标签，恰当地将这些数据传送给声卡或视频播放窗口。更重要的发展是用户交互的引入。这意味着用户不但可以浏览或倾听，还可以回答问题，发送数据给服务器。这种功能由客户端和服务器端程序共同合作而实现。这些程序一般是用Perl、JavaScript、ASP（Active Server Pages）或PHP（PHP Hypertext

Preprocessor，PHP超文本预处理程序）语言编写的脚本。服务器端的程序要么使用CGI（Common Gateway Interface，公共网关接口）或**ASP**，要么使用更新的**PHP**。在15-16给出的例子中，我们使用telnet访问CGI执行程序，代码用C语言编写。浏览器，如Netscape，同样可以用来访问CGI执行程序，只需将CGI程序的URL输入到浏览器中即可。这个例子中，welcome.cgi执行文件位于我的机器的~rwilliam/public_html/目录下，使用下面的URL可以检查它是否可用：

http://www.cems.uwe.ac.uk/~rwilliam/

之后使用下面的URL访问并运行这个CGI：

http://www.cems.uwe.ac.uk/~rwilliam/welcome.cgi?Your%20Name%20Here



图15-15　在Unix上用Netscape打开world.html

表15-3　基本HTML标签

标　签	功　能
<HTML>...</HTML>	页界定符
<HEAD>...</HEAD>	页标题
<TITLE>...</TITLE>	http标题（不可见）
<BODY>...</BODY>	主文本界定符
<BASEFONT FACE="Helvetica" SIZE=+2>	主体文本字体选择
<FONT FACE="Arial"COLOR="#808040"SIZE=+2>...</FONT>	字体类型、大小和颜色
<Hx>...</Hx>	第x级子标题
<B>...</B>	加粗字体
<I>...</I>	斜体
<UL>...</UL>	无序列表
<OL>...</OL>	有序列表
<MENU>...</MENU>	菜单
<LI>	列表开始
 	文本分隔符（等于C语言中的\n）
<P>	新段落
<HR>	水平线
<PRE>...</PRE>	预格式化文本
<IMG SRC="...">	在此插入图像文件
<A HREF="http://www...">[Press]</A>	建立超链接

浏览器会联系我们的Web服务器（ferdy）位于www.cems.uwe.ac.uk（164.11.8.19），并请求它运行CGI程序（welcome.cgi），并将参数“Your Name Here”或任何你选定的字符串传递给它。需要注意的是，此处空格用十六进制值“%20”来表示（见表2-4）。输入参数对于CGI程序就好像是控制台会话中的命令行参数。对于多参数的情况，参数必须以“?”字符分隔。来自于CGI程序的标准输出会被Web服务器获得，并传递回浏览器，由浏览器将这些字符按照它们所能做到的最好的方式呈现给用户。如果字符流中含有HTML标签，浏览器会尝试依照它们给出的指示对文本进行编排。

```
rob> cat welcome.c

 #include <stdio.h>
 main (int argc, char *argv[]) {
     printf("Content-type: text/html\n");
     printf("\nHello %s again!\n", argv[1]);
 }

rob> gcc welcome.c -o ~/public_html/welcome.cgi
rob> chmod 755 ~/public_html/welcome.cgi
rob>
```

编译产生可执行文件，作为Web服务器的CGI程序

```
rob@localhost$ telnet cems.uwe.ac.uk 80
Trying 164.11.8.19...
Connected to ferdy.cems.uwe.ac.uk.
Escape character is '^]'.
GET /~rwilliam/welcome.cgi?Rob_Williams
Hello Rob_Williams again!
Connection closed by foreign host.
rob@localhost$
```

在桌面主机上使用telnet运行CGI

图15-16 CGI形式的C程序——输入输出参数演示

15.8 HTTP

从图15-4我们可以看到，TCP层的80端口被分配给HTTP。这表示外部对80端口的连接请求将会被定向到可以处理HTTP格式编码的命令的服务器守护程序。现在，HTTP使用ASCII编码，因而终端可以直接对它们进行处理。人们可能很熟悉使用Netscape Navigator等浏览器发送和接收HTTP命令，但手动执行也是可行的。图15-16和图15-17的例子中，我们使用telnet连接到端口为80的万维网服务器。然后就可以使用get命令请求Web页面。图15-16中的例子使用了HTTP参数传递机制中的一种，通过这种参数传递机制，浏览器可以向服务器传递字符串，服务器使用这个字符串修改网页的文字，之后发送给浏览器，显示给用户。

HTTP是一种客户机-服务器协议，主要用于从万维网上获取文档。文档使用URL寻址，如http://freespace.virgin.net/sizzling.jalfrezi/。HTTP协议并不复杂，它只提供支持客户机－服务器对话所需的几个命令。表15-4列出了其中的一部分。

表15-4 超文本传输协议的例子

PROXY URL	通过PROXY命令，我们可以定义一个代理HTTP服务器，后续的客户端命令就使用这个代理服务器。URL参数用以设置该代理服务器。将PROXY设为空表示关闭代理特性
HEAD URL HTTP/1.0	HEAD命令读取位于URL处文档的HTTP报头
GET URL HTTP/1.0	GET命令读取位于URL处的文档。文档的主体被写入文件。这个命令返回HTTP报头，参见之前的HTTP HEAD命令
POST URL filename_1 filename_2	POST命令将本地文件filename_1的内容发送到指定的URL。返回的内容写入到filename_2。这个命令返回HTTP报头，参见之前的HTTP HEAD命令
PUT URL file	PUT命令将file复制到URL。这个命令返回HTTP报头，参见之前的HTTP HEAD命令
DELETE URL	DELETE命令删除URL处的文档。这个命令返回HTTP状态信息

```
rob@localhost>   telnet www.cems.uwe.ac.uk 80
Trying 164.11.8.19...
Connected to www.cems.uwe.ac.uk (164.11.8.19).
Escape character is '^]'.
GET /~rwilliam/CSA_sheets/
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
 <HEAD>
  <TITLE>Index of /~rwilliam</TITLE>
 </HEAD>
 <BODY>
<H1>Index of /~rwilliam</H1>
<PRE><IMG SRC="/icons/blank.gif" ALT="          "> <A HREF="?N=D">Name</A>
                                                   <A HREF="?M=A">Last modified</A>
                                                   <A HREF="?S=A">Size</A>
                                                   <A HREF="?D=A">Description</A>
<HR>
<IMG SRC="/icons/back.gif" ALT="[DIR]"> <A HREF="/">Parent Directory</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="CSA_course/">CSA_course/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="CSA_cw/">CSA_cw/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="CSA_marks/">CSA_marks/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="CSA_sheets/">CSA_sheets/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="CSA_tests/">CSA_tests/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="ICS/">ICS/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="IO_prog/">IO_prog/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="RTSD_cw/">RTSD_cw/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="RTSD_exams/">RTSD_exams/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="Transf/">Transf/</A>
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="Unix_docs/">Unix_docs/</A>
<IMG SRC="/icons/text.gif" ALT="[TXT]"> <A HREF="http_spook">http_spook</A>
</PRE><HR>
<ADDRESS>Apache/1.3.26 Server at <A HREF="mailto:webmaster@cems.uwe.ac.uk">
                                        www.cems.uwe.ac.uk</A> Port 80</ADDRESS>
</BODY></HTML>
Connection closed by foreign host.
rob@localhost>
```

图15-17　使用telnet访问Web服务器

图15-17中，我们首先建立到www.cems.uwe.ac.uk的HTTP端口（80）的连接，并使用HTTP GET命令请求根页面。欢迎页面的HTML相当直观，除了长一些以外。我们可以将超链接按钮的定义看做一个块，另外要注意，传输完成后连接就会被中断。每个请求必须创建新的连接。现在人们正在重新审视这种无连接的策略，考虑如何改善页面服务器的性能。读取HTML页面并非一定要直接使用GET命令。试着访问纯文本文件，如图15-18所示。此处，我们使用HTTP GET

```
rob@localhost> telnet www.cems.uwe.ac.uk 80
Trying 164.11.8.19...
Connected to www.cems.uwe.ac.uk (164.11.8.19).
Escape character is '^]'.
GET /~rwilliam/http_spook

Hello, and welcome to the target
file in this telnet/http file GET exercise.

Connection closed by foreign host.
rob@localhost>
```

图15-18　使用http /GET替代ftp/get

取代人们更熟悉的FTP GET从服务器下载ASCII源文件。

15.9 搜索引擎Google

各种搜索引擎是万维网的核心所在，正是它们赋予了万维网强大的生命力。任何大型的图书馆如果没有有效的检索目录，在现实中都会没有用处。没有人会简单地在数英里长的书架中查找期望的书籍。如果没有搜索引擎，那么万维网也是如此。即使有了制作者在其页面内插入的如迷宫般的互相关联的超文本引用，在查找新的材料时也会产生问题。这就如同只允许使用现有图书的参考书目查找其他感兴趣的图书。Google、Yahoo、Lycos、AskJeeves以及众多其他类似网站，每天都处理数以百万计的搜索查询，这是万维网成功的关键所在。当前万维网的网页数量超过百亿，数据大小超过25 TB，这些网页的索引文件可能会达到2 TB。

在Internet充斥图像、音频和万维网之前，是安静的文本时代。许多Internet站点通过匿名**FTP**（File Transfer Protocol，文件传输协议）提供ASCII编码的文档。系统管理员只需改变这些文件的访问权限，允许任何用户ID为“anonymous”的用户读取它们即可。可执行的二进制文件也可以使用FTP来处理，但许多人更愿意使用Unix实用工具uuencode将它们以ASCII编码，以使传输更为安全。任何计算机用户可以通过Internet登录到这些服务器，或使用调制解调器端口直接拨号，检查索引文件，并将选定的文件下载到本地计算机内。为了这样做，每台计算机必须拥有ftp程序：在一台计算机上以服务器模式运行，其他以客户机模式运行。许多大学的院系都将提供匿名FTP服务器作为发布软件的一种方式。存在许多针对PC、Unix、Atari、Amiga、Macintosh和其他特殊兴趣的团体。这些站点基本上没有交互，一些站点在欧洲建立镜像站点，以减少跨大西洋的传输需求。查找未知FTP站点上有用材料的问题，部分地通过叫做Archie的数据库服务得以改善。它提供了首个电子目录，这种服务现在已经由当前的搜索引擎取而代之。

搜索引擎能够完成各种不同的操作，见表15-5，它不但要处理大小不断攀升的万维网数据，还要每天应付不计其数的查询。为此，Google搜索引擎在全世界范围内建立了数个大型计算机集群，每个集群都由数以千计的运行Linux的常规PC构成。硬件可靠性相关的许多重要问题，都通过功能冗余以及配备大量额外的资源得以解决，通过软件，它们能够处理硬件故障。通过采用并行处理，查询的吞吐量和响应时间得到了很大的改善和提高。例如，当浏览器向www.google.com发送查询时，它首先向DNS数据库请求IP编号（见图15-19）。DNS会返回Google最近的集群的IP编号，通过将处理负载分散给遍布全球的搜索引擎集群，减少了传输的延迟。之后，查询被分派给该IP编号，随后被发送给多个负责搜索索引的计算机，它们分别负责处理万维网索引的一部分。索引搜索本身又可以通过并行计算来加速，多个搜索可以同时进行。所有负责搜索索引的计算机都有多个复制品，这样还能够保证操作的弹性。

表15-5 Google搜索引擎的活动汇总

抓取网络资料	只要有可能，尽最大努力查找和下载每个能够访问得到的网页，对之进行分析
编制索引	更新词汇、文档页的索引和初步的单词索引
排序	建立所有单词的逆序索引；这部分内容将被划分成区，以便进行并行搜索
搜索	检查查询字符串，在主逆序索引文件中找出其中的单词，处理页面的点击，查询出最相关的结果
排名	选择搜索出来的最好结果，优先将它们显示出来

Google能够给出更好的查询结果，不仅因为它拥有高效的并行搜索技术，还因为它使用了一套专有的系统——页面排名（PageRank），将不相关的结果滤除掉。页面排名的原理是通过其他页面对该页面的引用数量评估它的重要性。页面的评估值不是来自于点击率，而是来自于引用清单。较之其他策略，这种方法能够提交更有用的清单。它对.edu和.gov站点有所偏爱，而对于不断增长的.com站点有所疏远，因为这些网站的信息以新闻和广告等信息居多。

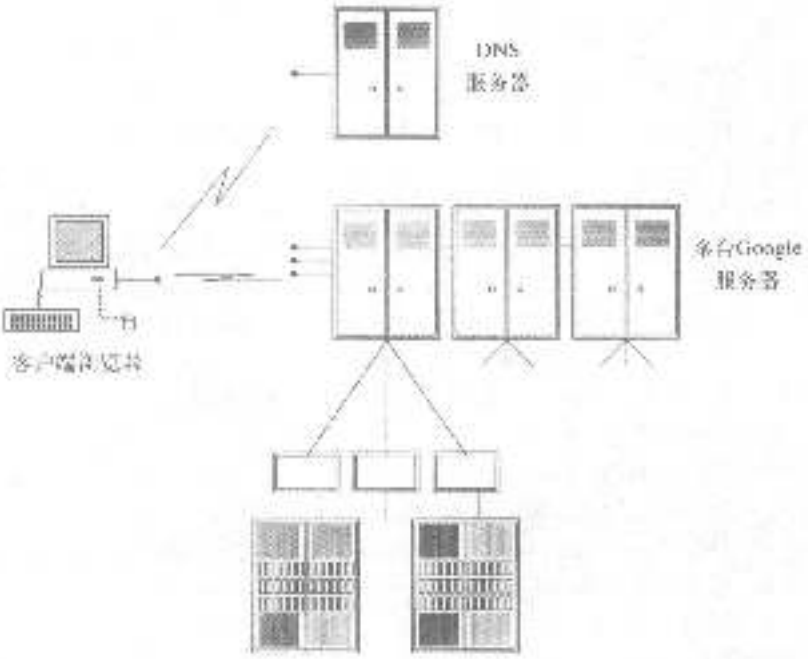


图15-19　Google搜索引擎示意图

Google在自己的硬盘上保存了整个万维网的副本，人们可能会对此感到意外。尽管页面均经过压缩，但数据量依旧会达到数TB，参见表15-6。文档服务器集群从那些负责索引查询和网页抓取的计算机中分离出来。索引查询的结果收集完毕、排序并经过过滤之后，需要将其标题和关键字所在上下文的片断，从保存在文档服务器中的页面副本中提取出来，为了提高查询的效能，工程师在设计Google时，利用大容量的DRAM内存尽可能避免磁盘查找操作。大部分索引代码以C/C++语言编写，在URL服务器和最重要的抓取代码部分，用到了Python。

表15-6　Google的存储容量统计，覆盖2400万网页（Brin和Page，2000）

网页的量	
压缩前	147.8 GB
压缩后（zlib）	53.5 GB
短波序索引	4.1 GB
全波序索引	37.2 GB
词典	293 MB
锚数据	6.6 GB
文档索引	9.7 GB
链接数据库	3.9 GB
Google需要在存储空间	108.7 GB

万维网每月都在快速增长，现已具有超过五十亿的网页，可访问的数据量达到数十太字节（TB）。这些直接寻找索引数据的大小达到数十吉字节（GB）。由于奔腾处理器的地址总线为32位，这就限制它能够直接访问的主存局限在4 GB，这是Intel制造64位安腾处理器的原因之一（参见第22章）。

对于这样规模庞大而且动态变化的数据集，其索引信息需要大大更新，不断将出现在网络上的新资料检索进来。图15-20中勾画出Google日常的活动。随时更新索引的任务由多台叫做Spider（蜘蛛）的计算机承担。这些计算机不断地抓取网络上的内容，根据链接不断地从一个页面跳转到另一个页面，搜索变动过的页面和新出现的资料。这种采用主动搜索的方式建立索引的技术，与其他一些依靠页面的作者书面提交请求进行检索的网站形成对比。所有的搜索引擎都倾向于更频繁地进行索引活动，其负面效应是建立索引的计算机频繁的访问，会降低Internet的有限带宽。

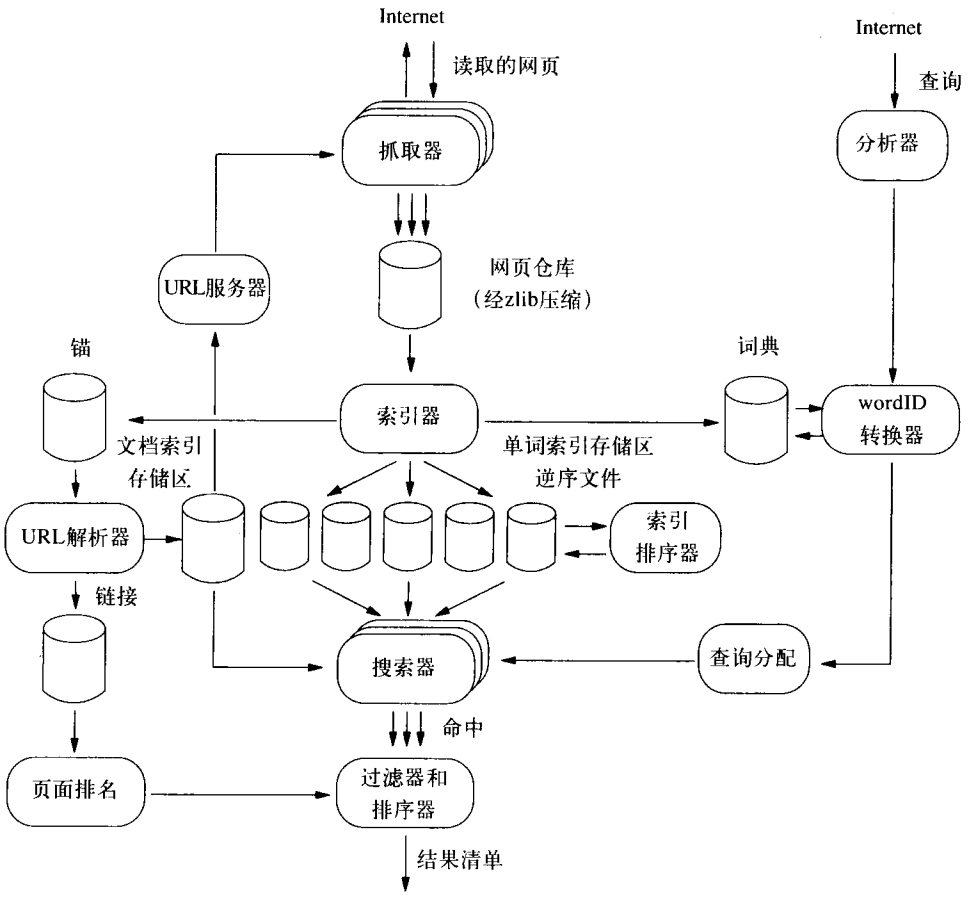


图15-20 Google数据流示意图（Brin和Page，2000）

如果你想和别人分享你的HTML页面，最好是将其注册到主流的搜索引擎中。尽管现在已经有付费的服务可以帮助你做这些事情，但自己将页面注册到Google和Yahoo也不困难。

15.10 操作系统互连：一种理想的方案

通信协议，如TCP/IP，是一些规则和过程，它们常常在软件中实现，可以使信息的传输以安全和可控制的方式进行。

在20世纪80年代早期，国际标准化组织（International Organization for Standardization，ISO）——隶属ITU-T，为数据通信网络提供了一个设计框架或称模型构架（见图15-21）。这就是**七层模型**（Seven Layer Model），每个层负责接收数据、处理它，并将它传递给下一层。层与层之间都有定义明确的接口，包括数据格式、命令码和功能性服务。这样做的目的是，希望通过这个模型，改善不同的制造商生产的设备之间的兼容性，包括软件和硬件。但是事实上的标准在商业压力下，已经部分淹没了这种好的本意。TCP/IP并没有完美地遵照这个描述性模型（见图15-22），但成功地控制了整个广域网市场。但是，国际性公认标准的存在确实影响到后来的通信软件设计人员，尽管它没有获得业界的一致接纳。或许是制订标准的官僚主义方式（组织一系列长时间的委员会议商讨抽象结构，而其他标准早就已经有了能够运行的代码）注定了最后它不过是学校考试中的常规科目而已。在那些日子中，ISO模型的竞争来自于IBM的SNA（Systems Network Architecture，系统网络体系结构）协议。

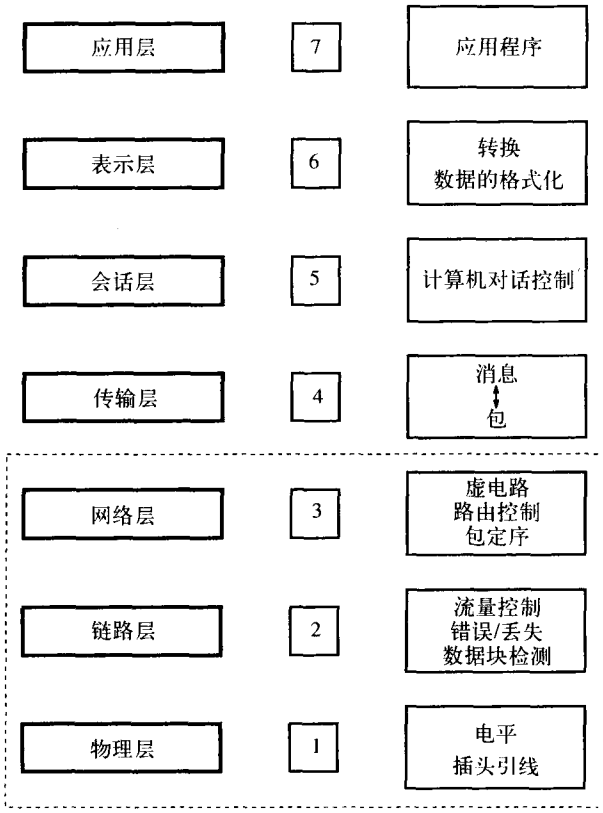


图15-21　ISO七层OSI模型

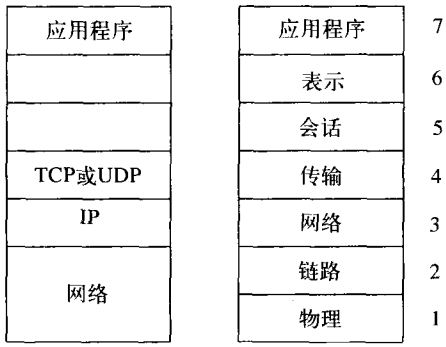


图15-22　TCP/IP与OSI七层模型的对比

术语“**对等**”（peer-to-peer）通信可以用在这类分层模型中。以发送堆栈中任何一层为例，它都从上面的层接收数据和指令，执行指令，并且可能在将数据发送到下一层之前，附加额外的信息和指令。这些额外指令的目的地，实际上是接收堆栈内等同的层，它们将在那里被读取并执行。每个层都会抓住机会附加额外的信息给自己在目的计算机上的搭档。实际上，这些信息常常必须通过中间主机和路由器。这些主机和路由器也会读取数据包，在必要的时候将它在输入堆栈中向最上层传递，然后再将它通过输出堆栈向下传递。因此，对等协议消息并不仅仅由目的层读取。

每个层都会与另一端的搭档交换错误和流量控制信息。消息被拆分成数据包，在发送前加上必需的路由和控制信息。作为WAN-PSS网络的访问协议，X25协议取得了一些成功，但上面的层依旧大部分为试验性的，常常并未实现。

15.11 小结

- 广域网络的发展由模拟电话开始，现已经扩展到计算机间的数据交换领域。
- 自万维网浏览工具和搜索引擎引入以来，Internet取得了巨大的成功。
- 在Internet以及大多数的局域网中，TCP/IP都是一种主要的协议。它提供定义和规则，用以控制数据包的传输和接收。
- 实现了TCP/IP规则的软件，一般都组织成由多个层构成的堆栈。
- 以太网使用广播方式在局域网上传送数据包。而广域网则需要不同的路由策略。路由表通过发送探测包并定期交换数据包递送成功与否的信息来维护。路由机制是分布式的，并不归属于单个主控路由器。每个主路由器的表中都保存所有网络的信息，但不包括所有的主机。
- 在IP和MAC层之上又引入另一种地址指定（或命名）层。域名的分配由地理位置或功能特征来控制。从域名到IP的转换由动态分布式数据库DNS负责。
- IPv4已经不堪重负，IPv6将IP编号从32位增加到128位。引入IPv6的同时，层次化的分配方案也会一同引入，这样数据的路由就会得以简化。
- 用简单的标记语言（HTML）对文本文档进行编码，并使用超文本协议通过Internet将它们散发出去，这种方式取得了巨大的成功。图像和音频文件也可以使用同样的方法来访问。
- 浏览器图形用户界面的引入，使得人们对万维网的兴趣呈爆炸性地增长。搜索引擎（如Google、AltaVista和Yahoo）对此功不可没。

实习作业

我们推荐的实习作业包括使用socket编程，在不同操作系统（如Unix和Windows）间跨网络通信。这使得我们可以在多个机器间传递数据，而不必考虑计算机提供商。也可以跟踪研究使用GSM手持式设备发送和接收电子邮件。连接到Internet需要网关的支持。

练习

1. 列出Internet的优点和缺点。它对我们的生活有什么深远的影响？
2. 区分中继器、桥接器和路由器。哪个有自己的MAC编号？哪个有IP编号？
3. TCP/IP堆栈中，哪个层使用下面的资源？
   - (a) /etc/hosts文件
   (b) ARP表
   (c) 路由表
   (d) /etc/services文件
4. IPv6 128位长。在CIDR方案中，有多少数字为欧洲保留？你认为够用吗？如果世界上所有的家用设备都在出厂前分配一个IP编号，128位是否够用？
5. 被隔离的局域网不需要动态路由或DNS，它是否需要拥有TCP/IP堆栈呢？
6. 更新IP路由表有哪些技术可以选择？可以手动完成吗？
7. 客户机-服务器网络和对等网络是否相同？
8. 如果你正在陌生的国度驾车行驶，想要到达渡假目的地。比较下面的导航策略。
   (a) 预先从图书馆的地图中规划精确的路线，将所有必要的左转和右转信息记在车内可以方便查看的清单上。
   (b) 购买地图，在需要时带在身边备用。
   (c) 不带任何地图和方向清单，在需要时停车询问本地人。
   (d) 完全依赖于路边的标志。
9. 为什么有些机器有两个IP编号？它们需要两个域名吗？
10. 试着使用ping实用工具。它会发出探索性的数据包，请求目的主机做出应答。

课外读物

- Heuring和Jordan（2004），介绍Internet。
- Buchanan（1998），第36章：introduction to TCP/IP and DNS。
- Tanenbaum（2000）。
- Stevens（1994）。
- 万维网团体的主要站点是：

http://www.w3.org/

试着阅读介绍协议的部分，找出更多关于HTTP的信息。

- 万维网所有的初始规范（RFC），可以在下面的地址找到：

http://www.FreeSoft.org/CIE/index.htm

- Internet的历史传记，可以参见ACM网站：

http://www.acm.org/crossroads/xrds2-1/inet-history.html

- 如何创建HTML文档：

http://www.ncsa.uiuc.edu/General/Internet/WWW/index.html

- 更多的万维网信息，参见：

http://freespace.virgin.net/sizzling.jalfrezi/iniframe.htm

- 有关Google的一些相当有价值的信息：

Barrosa等（2003）。

Brin和Page（2000）。

- 下面是一段2000年对Google核心技术的描述：

http://www-db.stanford.edu/~backrub/google.html

- 其他一些搜索领域的探讨文章：

http://labs.google.com/papers.html

- 这些网站可以通过本书的配套网站访问：

http://www.pearsoned.co.uk/williams

第16章 其他网络

所有需要数据通信的不同类型的网络和计算机应用，正在经受商业上的考验。过去一段时间内，计算机之间的数据通信链路使用传统的固定电话系统，但现今专为数据通信设计的网络，在操作的灵活性以及带宽和成本方面更有优势。新的ATM标准力图将数据和语音通信统一到单一网络中。广播网络不再仅仅用于出租汽车或紧急服务，现在它已经和动态路由及包交换技术结合起来。

16.1 PSTN：电话网络

虽然这是一门针对计算机学科学生的引导性课程，但近年来，电信行业飞快发展，它的扩张依赖于软件和微处理器技术。第1章介绍的技术趋同现象已经将电信和计算科学越来越紧密地联系在一起。现在，这个领域为我们提供许多激动人心的事业和研发机会，程序员和软件工程师不能视而不见。计算机联网为我们提供了共享信息、向读者表达自己的见解，以及寻找人来回答困难问题的途径。这些一直都是人类的重要活动，并因此衍生了第一个也是最大的通信网络——**PSTN**（Public Service Telephone Network，公共服务电话网络）。它始建于100年前，见第15章，它对计算机之间的长距离互连起到十分重要的作用。但是，由于它采用的是模拟电压，因而通过它传输数据时，调制解调器是必不可少的。有一段时间，电话公司，比如英国的英国电信，为客户提供全数字的服务，这种情况下就不再需要调制解调器。这也将电信和数据通信之间最显然的区分消除了。

但是，不同类型的互连之间根本性的区别依旧存在，在用户拨号建立电话连接时，信号传输的路线首先是铜线，通过交换设备，然后是中继线路，这条线路专为用户的会话服务，即使用户一言不发，依旧在使用电话公司提供的电路。用户租用的是电话公司提供的有保障的线路。整个电话系统专为可靠地传递话音信号而设计，根据用户使用的时间收取费用，即使没有人说话亦是如此。数据广域网则与这种情况迥然相异，付清连接（或称开通）费用后，用户只需为所传输的数据付费。这是因为用户和其他客户共享**带宽**。用户的数据包和Internet高速公路上的其他数据包一同传送。如果所有的人都试图同时传送数据，那么我们的数据包会由于排队而产生延迟。一般情况下，由于数据依旧能够在很短的时间内到达，因此没有什么影响。但它确实增加了传输的不可预测性，因此Internet不适合于传送高质量的语音通信。是否能够通过提高Internet高速公路的带宽，使得它远远超过客户所要求的速度，以使语音数据包能够通过Internet传输而不会产生太多的停顿呢？现在，这一问题依旧没有定论。如果Internet开始传送大量的音频流量，PSTN网络可能会发生巨变，或许PSTN可以服务于NAN（Neighbourhood Area Network，邻域网），主干线全部服务于Internet。

电话网络的基础是电路交换。人们需要使用电话时，需要由交换机建立从电话发起者到目的地之间的通道（如图16-1所示），并在会话期间一直保持。这种**电路交换**技术起源于“铜线通道”领域，现已扩展到高带宽的光纤通道（光纤通道将许多通话复用到同一通路中）。这称为**虚电路**的建立，在源和目的地之间没有专用的物理通道。其中，决定性的因素是，什么时候做路由决定？电路交换，包括虚电路，都是在呼叫开始或数据发送之前预先配置好的，而Internet支持数据包传送过程中的动态重新选路。

交换设备常常叫做**交换机**。现代交换设备的运作已经完全数字化，更像是计算机中心，而非手动开关面板！整个操作完全在**嵌入式处理器**的掌控之下，这些处理器互相通信，建立电路连接，将呼叫传送到世界的各个角落。交换中心之间控制和状态信息的传输和语音信号同样重要。PSTN的一项特性是**公共信道信令**（common channel signalling）——正是这项特性使得多种调制解调器设备的引入成为可能。这种方案将控制和状态信号从话音通道中分离出来。它和用户拨打存在很大的不

同。拨号时是在话音通道上叠加DTMF（Dual Tone Multi-Frequency，双音多频制）音频。对于用户线路，这样做可以接受。在长距离干线传输一般采用独立的通道来分发呼叫设置信息。这些控制和信号通道为通过电话网络传递日益复杂的命令提供便利。

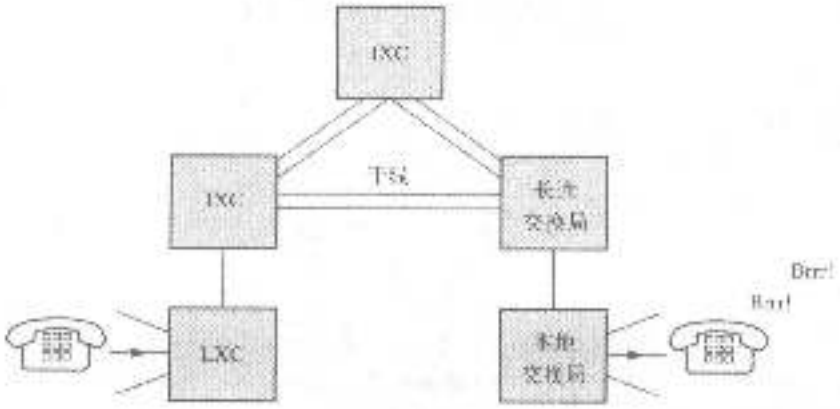


图16-1　传统的电话互连网络

DTMF信号传输称为“带内”传输，因为它公开地沿着话音通道传送，说话人和受话人都能够听到它。从图16-2我们可以看到，每个数字都由一对频率来表示。这种频率编码与调制解调器用来传输二进制数据的编码相似，不过要更复杂些，因为需要表示的符号不是2个，而是12个。接收方需要装备能够识别音频并得出它们所表示的数字的检测电路。使用这种带内方式的优点是，它既可以用来在呼叫前建立通路，也可以用在呼叫中控制用户设备，或在需要时进一步路由。

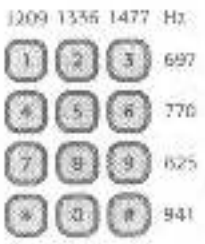


图16-2　DTMF：按键式键盘

如图16-3中所示，模拟语音的波形进入本地交换中心时，在LIC（Line Interface Card，线路接口卡）进行数字化。如果使用全数字化的ISDN连接，信号则由用户端的终端设备对信号进行数字化。不管哪种情况，一个专门的ADC（Analogue to Digital Converter，模拟数字转换器），又称编解码器（Codec），都会对语音电压信号以每秒8000次的频率采样，产生8位数据流。间隔为125μs，这个64 kb/s的数据流代表一路电话会话。同时还需要有一个返回流。编解码器还会压缩振幅以提高接收端声音的品质。这种非线性压缩给较低的声音更多的通道容量，并缩减音量声音的通道容量。欧洲（A-law）和北美（u-law）使用的压缩曲线的形状稍有不同，故而所有跨大西洋的语音信号都必须进行调整。在图16-4中，垂直轴上标出对应各种输入的输出。这个模式与人耳的对数敏感曲线匹配得很好，同时降低了传输噪音的影响。由于带宽更加有限，移动电话使用更复杂的压缩技术。需要注意的是，编解码器并不移动信号频率的波段，因而它们是基带设备，不同于电话调制解调器。

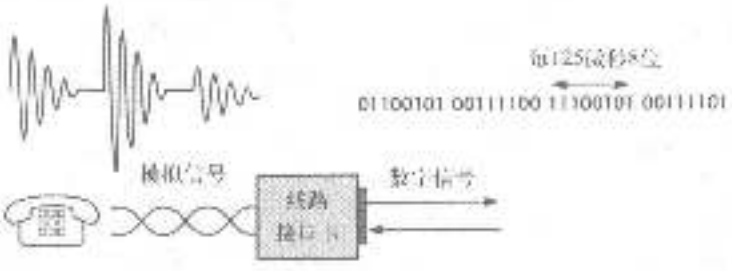


图16-3　电话语音信号的数字化

如果需要，图16-1中的**LXC**（Local Exchange Center，本地交换局），会将呼叫定向到最近的**IXC**（Inter or Trunk Exchange Centre，长途交换中心），建立经由长距离干线（trunk line）的连接。之后，这个通路就为这次通信单独保留，路线和分配的带宽不会变化。这种方式拥有明显的优点，即：在最初的启动延迟之后，数据的传输就不再需要任何再次的路由。连接之后，客户得到的服务水平能够得到保证，隐私也能够得到保护。在呼叫结束后，线路结束，带宽释放。

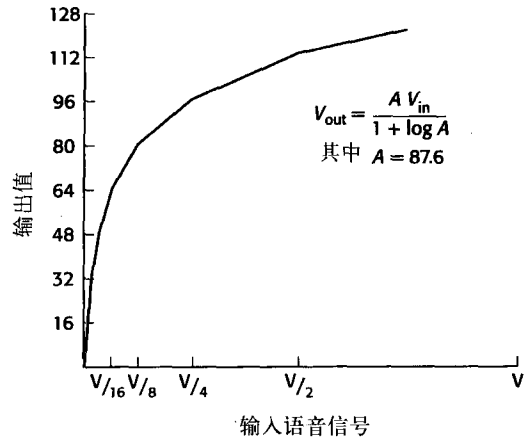


图16-4 电话传输的非线性话音压缩

由于通路专用（语音信号大部分时间是寂静的），电路交换的额外开销促成大家开始对包交换技术感兴趣，即VOIP（Voice over IP，通过IP网络传输语音）。再次提一下，计算机和电信技术的趋同现象十分明显。

模拟话音信号到二进制数字流的转换，对于长途传输尤其有用。这是因为干线并非完美的载体，模拟信号的长途传输，比如从伦敦到布里斯托尔，会使信号发生明显的恶化。因此，我们得在长途线路中安装许多放大器，以增加功率，使它依旧能够推动目的地的扬声器。二进制数据的传输比较简单，因为表示1和0的电压即使退化到可识别的极限，依旧可以恢复回来，并且不会对消息的质量造成影响。在邻近的模拟电压间不可能建立Hamming距离——允许范围内的任何电压都是合法的。二进制数据对于噪音影响具有更好的弹性，因为如果要对二进制数据造成影响，必须施加持续不变的相同影响，直至1和0变得不可识别。即使这样，如果损坏是间歇性的如10.3节所述，采用错误检测和纠正技术，也可以将二进制数据恢复出来。

二进制编码话音数据的长途干线传输一般使用**TDM**（Time Division Multiplexing，时分复用）方法，这种方法通过快速地依次传输会话数据，允许30路会话共享同一通道。这样做之所以能够实现，是因为即使最慢干线的带宽（2.048 Mb/s）也可以处理许多64 kb/s的声音会话。需要注意的是：

$$\frac{2.048\text{M}}{64\text{k}} = \frac{2\times10^6}{64\times10^3} = \frac{10^3}{32} = 32$$

从而干线能够承受32路复用通道。

其中的两个通道C1和C2，为建立呼叫和清除呼叫的信息传输而保留（见图16-5）。C1在数据之前，而C2插入在话音时隙15和16之间。其他都承载8位的数字化话音采样。

简单的数据多路复用电路的结构在4.3节中曾做过论述，数字化的话音流也可以用同样的方式进行处理。由于表示话音信号的二进制数值每125μs就需要刷新和传输，以重建原始语音信号的清晰副本，因此TDM通道以8000 Hz的频率在各个时隙（time slot）间切换。图16-5给出使用**TDM/PCM**（Time Division Multiplexed/Pulse Code Modulation，时分复用/脉冲编码调制）编码的2.048 MHz干线进行语音传输的图示。它可以同时承载30路独立的话音通道。在北美，这个标准稍有不同，它使用24个时间片，频率为1.54 MHz。但声音采样依旧需要每秒钟传送8000次，以维护不失真的声音链接。当然，为了满足正常的、双向的会话需求，我们还需要相反方向的第二条干线。

将电话机互连在一起的最直观方式是使用开关矩阵。图16-6说明了这种技术，并给出如何使用计算机控制整个运作。为了简化，图中只给出单向的电路。完整的系统还需要另一个同样的开关矩阵处理返回方向的数据。这个方案既可以用来交换传统的模拟语音信号，也可以用来交换数字化编码的语音信息，后者更加常用，因为不会对传输信号造成显著的退化。

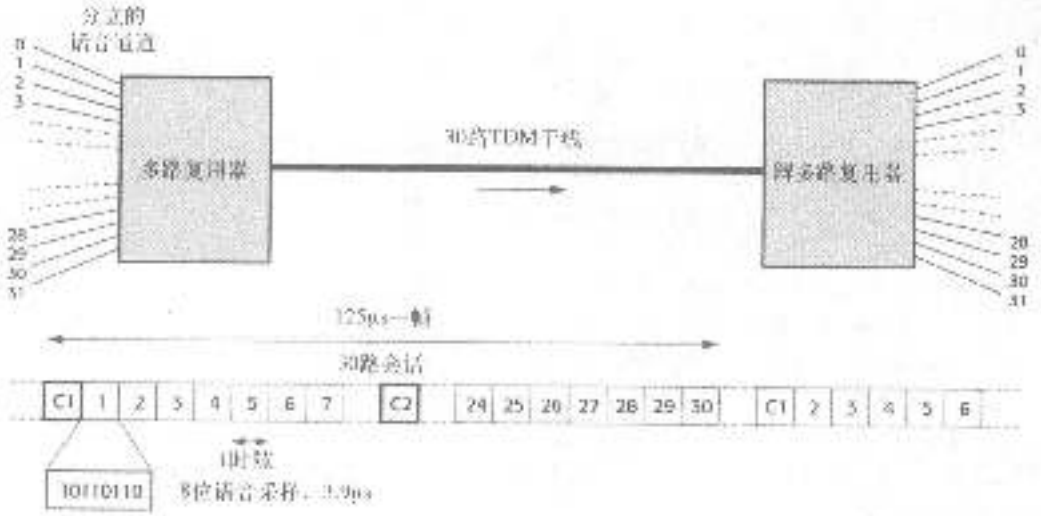


图16-5　十线共享的时分复用（TDM）技术

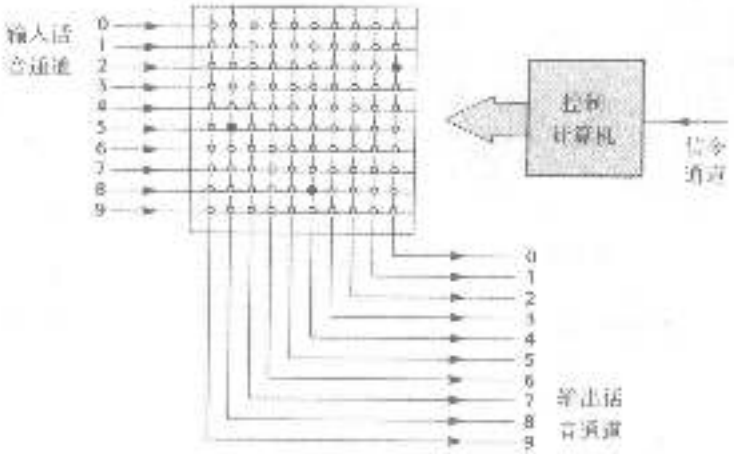


图16-6　带控制处理器的空分电路交换

图16-6中，输入线路2连接到输出线路0，输入线路5连接到输出线路8，输入线路8连接到输出线路4。交叉点开关完全在嵌入式处理器的控制之下，嵌入式处理器由公共的信令通道接收将哪个输入连接到哪个输出的指令。由于声音数据已被转换成数字形式，因而，此处使用的开关元件在操作上与图4-4中的数据流控制门十分相似。交换问题的另一种解决方案是时分交换。这种方式下，每路输入依次将它们的值复制到公共的数据总线，每路输出知道什么时候计划中的值占据总线，可供读取。这种交换技术有时也称做时隙互换。

图16-7所示的时分复用总线展示出含有时分传输对象的帧的循环模式。在为输入通道指定时隙后，控制器会负责安排将8位的数字值从该通道传入选定的时隙，8000次每秒。类似地，在输出端，控制器会周期性地将数据从时隙传输到目的通道。这样，通过安排端口在恰当的时间向TDM总线复制数据，使之可以被输出端口读取，就将输入线路与输出线路连接在一起。实际上就相当于选定的线路保持连接3.9μs来传输当前的声音采样数据。图16-7中的TDM总线上仅插入了两个多路复用卡，还可以插入更多卡，以获得更广泛的线路选择。同时，还可以将几个通道分配给同一时隙，这样就可以进行电话会议。

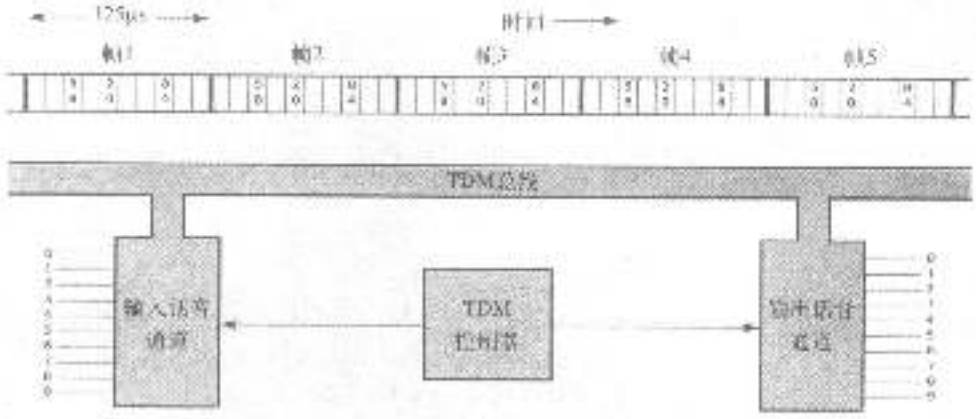


图16-7 时分电路交换

大多数电话交换设备同时使用时分和空分技术来建立呼叫者和受话者之间的通道。核心交换设备常常基于空间交换，输入和输出线路通过TDM（Time Division Multiplexed，时分复用）总线动态地连接到核心交换矩阵。时间和空间交换都由嵌入式处理器控制，它会接收路由选择信号并做出相应的动作。

大家在得知使用时分交换的电路并非连续地分配给任何一个呼叫之后，可能会有些惊奇。即使如此，和空分交换一样，我们依旧将其归为在呼叫期间保持通道。这种做法使得传统的电话电路交换与包交换的区别减小。在16.3节中，我们将会看到ATM进一步模糊了电路交换和包交换方法之间的差异。

电话产业公认的成就之一，就是建立并维护了由ETSI（European Telecommunications Standards Institute，欧洲电信标准化委员会）和CCITT（Comité Consultatif International Télégraphique et Téléphon，国际电报电话咨询委员会）——现在的ITU-T（International Telecommunications Union, Telephone Committee，国际电信联盟—电信标准部）——协商制订的标准。

PC电话卡是一种比较新的可供选择的网络接口。它可以完成小型办公室的电话交换，还可以取代传统自动应答电话记录话音消息的作用，它将数字化的声音数据存储在磁盘上，以备之后回放。它的出现是因为使用这种方案可以节省许多开支，下一代电话交换机可能就基于标准的PC硬件。Microsoft已经在Win32 API中加入**TAPI**（Telephone Application Programmer's Interface）扩展。电话卡制造商可以用它来作为产品的标准功能接口，这样程序员就可以更容易地开发可以运行在各种硬件平台上的软件。

16.2 Cellnet：移动通信提供商

蜂窝移动电话系统是一种新型的广域网，它部分基于短距离的无线链路，部分依赖于现有的电话干线。这是不同技术成功融合的又一实例。这类系统的设计、实现和维护，都需要多种技能的结合：无线电和天线工程、流量规划、协议协商、用户手持设备和地面交换机的实时程序设计、性能模拟和预测、呼叫路由和收费数据库的管理等等。通过将移动技术、短距离UHF无线电传输、地面干线网络和基于计算机的呼叫交换结合在一起，成功地克服了单个技术的局限性。虽然无线电通信总是要受到有限带宽的约束，但由于蜂窝传输只是低功率的本地信号，因此多个呼叫者可以共享无线电频谱。这种方案中，数据传输路程大部分在传统的地面网络中进行，不需要占用无线电带宽。呼叫者的移动性来自于快速呼叫切换的使用，不管呼叫者在什么位置，它都可以响应用户的连接请求。同时还可以处理由于呼叫者的移动而造成的信号强度的变化。无线电频段位于电磁波谱的中段，从超高频的γ射线到甚低频的长波无线电广播。图16-8给出了本章用到的一些特殊区段的名字，以及相应的传输介质。

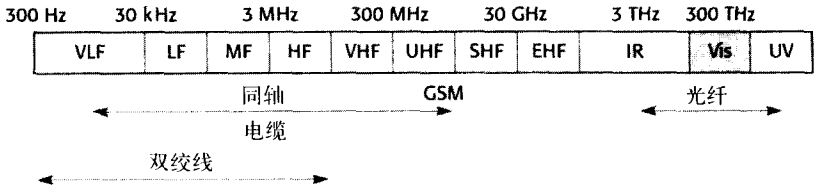


图16-8　电磁频谱中的波段

使用无线电传输信息时，需要将信息施加到高频（MHz～GHz）的无线电波上。模拟和数字传输都是如此。不在无线电广播中直接使用音频的原因，是由于它的频率范围十分有限。为了提高无线电广播的传输范围，我们使用远高于数据速率的载波频率，将需要传送的信号施加在它的上面。图16-9中给出经常使用的三种调制方式。**AM**（Amplitude Modulation，幅度调制）技术，它将无线电波“挤压”成需要传送的信号的形状。话音通道使用的无线电波，其振荡频率可能达到100 MHz，施加的信号最高只有3 kHz。无线电波按照期望传输的信号的形状，同步地变大或变小。**FM**（Frequency Modulation，频率调制）技术能够使无线电的频率与需要发送的信号同步变化。如果只有二进制流，载波频率会根据信号的1和0，在两个值之间变化。这就如同小提琴手在拉出一个音符的同时，上下移动手指变更声音的音调一样。这也是高品质无线电广播（UHF/FM）和GSM蜂窝数字传输所采用的调制方法。PM（Phase Modulation，相位调制）技术能够保持载波频率恒定，通过在两个固定的值之间切换正弦曲线的相位，来表示二进制数据。图16-9中，底部示意图波形的相位变动为180°（π），接收方可以检测出这种相位变化，并将它解释成二进制值的切换：1→0或0→1。

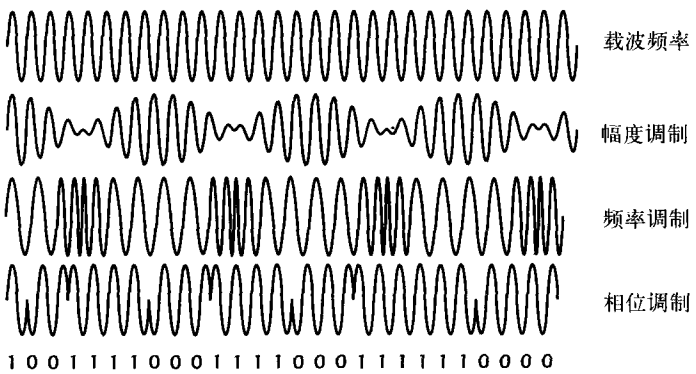


图16-9　无线电波幅度调制、频率调制和相位调制技术

根据10.9节中有关调制解调器的介绍，两种调制技术可以同时使用，以增加通过有限带宽无线电通道的数据传输速率。

欧洲模拟系统TACS/ETACS（Extended Total Access Communication System，欧洲全向接入通信系统），其最初的技术细节来自于美国的AMPS网络（AT&T，Advanced Mobile Phone System，AT&T先进移动电话系统）。在英国，政府分配1000个通道，约900 MHz的频段，由互相竞争的组织（Cellnet和Vodaphone）共享。移动电话的成功超过了每个人的预期。伦敦附近，M25环线以内，尽管引入额外的通道，并引入更多无线电天线以降低网络的传输距离，但资源的使用还是快速饱和。

蜂窝状方案的基本优势在于频率共享，也叫做“频谱重用”。这使得同一频率可以同时由大量的手持设备使用，而不会相互干扰，前提是它们不位于直接相邻的网格。与频率划分和时间划分相比，这是空间划分多路复用的实例。无线电频谱由位于不同地理位置、使用同一频段的设备共享。信号压缩方法可以进一步压缩所需的频带宽度。每个会话仅分配25 kHz。1000个可用通道中，有21个用于信号传输和控制，而非声音数据。

无线电天线服务于环绕它的一段区域，或称网络（cell）。相邻的传输天线可能使用地面线路连接到本地的基站或交换中心，参见图16-10。打开移动电话的电源时，它会搜索并锁定最强的控制信道，向本地基站登记它的存在。这种信息会由常规线路传送到主基站，这样就可以对呼叫进行重定向。如果移动电话想要发起一次呼叫，它会通过可用的控制信道发出信号，等待话音通道的分配。对移动终端的呼叫通过寻呼信道传送，通知移动设备有呼叫到达。

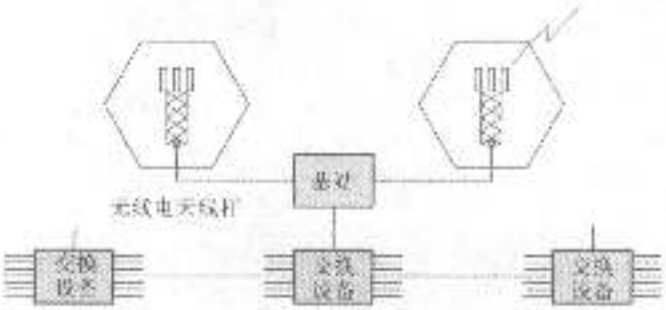


图16-10 无线电网络设备和互连

无线电波段（见表16-1）被过量订购多年。无线电通信的商业需求从20世纪50年代起就快速增长，有限的资源按照波段分配给英国不同的应用领域。

表16-1 无线频段的使用

0.1～0.3 GHz	VHF	海面电话和无线电广播
0.3～1.0 GHz	UHF	电视、GSM移动电话（0.9 GHz）、航空双向无线电、寻呼机
1.0～2.0 GHz		航海导航、GSM移动电话（1.8 GHz）
2.4～2.5 GHz		短距离无线电控制（蓝牙）
3.4～3.5 GHz		微波天线

除了公共的传输服务以外，实用设备进行无线电收发的需要也增长得很快。由于市场需求量巨大，以及制造能力的提升，手持电话的成本下降得很快。

在图16-11中，我们可以看到，任何相同标签的网格都隔开至少两个网格的宽度。这表示分配给网络2的无线电波段可以同时由所有网格2使用，只要严格控制传输功率，防止任何交叉干扰。最初的7六边形网络结构使用1000个呼叫波段，可以支持每网格1000/7=140个并发呼叫，没有频率重叠。因此，如果将某个区域划分成100个这样的网格，最大可以存在14 000个并发呼叫，相互之间的干扰相当小。使用4网格重复模式（见图16-12）从经济的角度来讲更好：1000/4=250，可以提供25 000个并发呼叫。通过降低重复因子以及网格大小，呼叫的数量还可增加；但是，这样会引发其他一些需要考虑的问题：构建基站和天线的成本，移动电话从一个网格到另一个网格的频繁切换，也会带来管理上的问题，这样也会增加邻近使用同一频率波段的网格之间的相互干扰。最后只能采用折衷的方法，运营商们最常使用的是7网格重复模式。

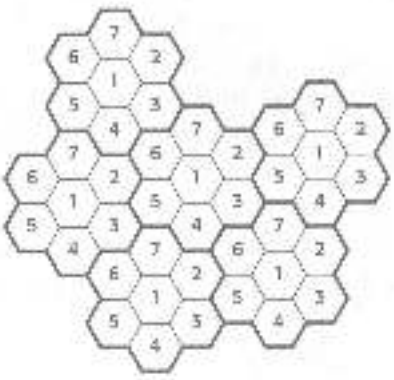


图16-11 无线电频率7段重复模式中的网格排列

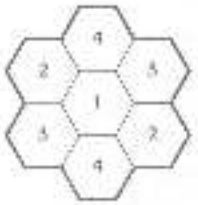


图16-12 4段重复模式的网格排列

在GSM中，无线电信道以FDMA/TDMA（Frequency Division Multiple Access/Time Division Multiple Access，频

分多路访问/时分多路访问）的方式组织。信道基于多频段TDMA信号格式。

为了进一步降低相互干扰，保存电池的电力，移动电话的传输强度会依照维护可接受的通话所需的最低水平进行动态调整。对于地面站的传输，信号强度被精心控制，以将无线电信号仅仅限制在网格以内。服务品质的统计数据必须不断收集，以响应传输环境由于新建筑物或竞争无线电发射源的出现而发生的变化。

900 MHz GSM划分成两个频段：上行链路（移动电话到基站）890～915 MHz，下行链路935～960 MHz。为了进行FDMA，这些频段又进一步划分成124个载波对，每个占据200 kHz的空间。每个网格最多可以拥有15个指定频段，网格的直径可以是1到5km之间的任意值，这要依本地的地理条件和用户的需求而定。每个宽度为200 kHz的通道使用固定的TDMA格式划分成多个时隙。数据传输可以达到270.833 kb/s（单个位的持续时间为3.69μs），一个时隙承载156.25个位，因而需要长度为577μs的时间片。每帧总共有8个时隙，长度为4.615ms。在通话期间，话音连接就使用8个时隙中的某个时隙。

对于移动无线电话，由于附近建筑物和物体的反射，信号强度可能会发生变化，并会产生多重信号的现象，因此还有一些特殊的问题需要考虑。如果在快速行驶的车上进行通信，载波频率会发生多普勒频移，就如同火车在疾驰时鸣笛的音质会发生变化一样。所有这些效应都由均衡电路负责调整。这项技术定期地发送已知的26位测试模型，进行微调和优化。尽管一般认为会话是全双工的，但实际上手持电话并不同时发送和接收，而是交替运行。从相对较强的输出脉冲信号中提取出极为微弱的输入信号并不容易。

手持电话和程序员也有联系，因为它们内部均装有软件来处理呼叫的发起及请求，同样还要用到话音压缩算法。尽管编解码器仅对来自内置式麦克风的模拟语音信号执行每秒8000次的13位采样，这样产生的数据速率还是太高。现在，在传输之前，这些信号都被压缩成13 kb/s的数据流。选择什么样的压缩算法、用户能够接受的语音质量下限是多少等问题，都是委员会议上激烈争辩的话题！编解码器实际上每20ms输出260位编码好的语音数据，也就是13 kb/s（260/0.02 b/s）。为了进一步降低传输的带宽，还对寂静时间加以探测并进行抑制。这样做不但能够降低需要传送的数据量，还能够进一步延长电池的寿命，降低无线电干扰效应。话音数据由CRC、卷积码和块交错对错误进行层层防护。这260位数据又被划分成不同的类别，前182位为Class-1位，编码，另外78位为Class-2位，不编码。这是因为语音采样中的某些位对于语音品质的影响大于其他位，因此只将错误控制有选择地应用到重要的分类。块交错指将相关的数据分布到几个传输块，这样就降低了突发的噪声干扰所造成的影响。在检测出错误时，常见的策略是使用刚刚过去的语音采样替代受损的数据。

根据处理的需求，GSM手持电话内的基本配备如下：无线电调制解调器信号接收均衡器、数据加密、语音传输的压缩编码、接收端的语音合成以及通道数据编解码。为了满足这些实时处理的要求（25 MIPS），手持电话内一般有几个CPU，至少有一个是专门针对语音数据优化处理的DSP（Digital Signal Processor，数字信号处理器）。各个基本单元之间的关系见图16-13。

用来通过模拟无线链路传输数字话音和数据的调制解调器类型为**GMSK**（Gaussian-filtered Minimum Shift Keying，高斯滤波最小移频键控）。**漫游**，或称呼叫的移交，在活动的客户从一个网格移到另一网格时，必须在300ms内完成，这是通过监控邻近基站的信号强度来做到的。在信号变得太微弱时，地面网络会将正在进行的通话重新安排切换到另一基站的信道上。

和所有的模拟传输一样，早期的TACS网络受到信号品质差和未经授权的窃听的困扰。当前的移动电话，**GSM**（Groupe Spécial Mobile de Conférence Europé ene des Administrations des Posters et des Télé communications，全球移动通信系统），通过使用GMSK载波调制技术，能够做到全数字传输。它提供更多通道，更安全、更复杂压缩技术的应用，能够提供更高的声音质量，它具备强大的错误纠正能力，数据传输也更简单。这也使得个人便携式传真设备的应用成为现实，能够浏览Internet的手持电话也已很普遍。移动网络的运营商已经开始支持短消息服务（Short Messaging Service，**SMS**）。这种服务十分类似双向寻呼机，它使用话音数据不使用的信号传输信道，快速传

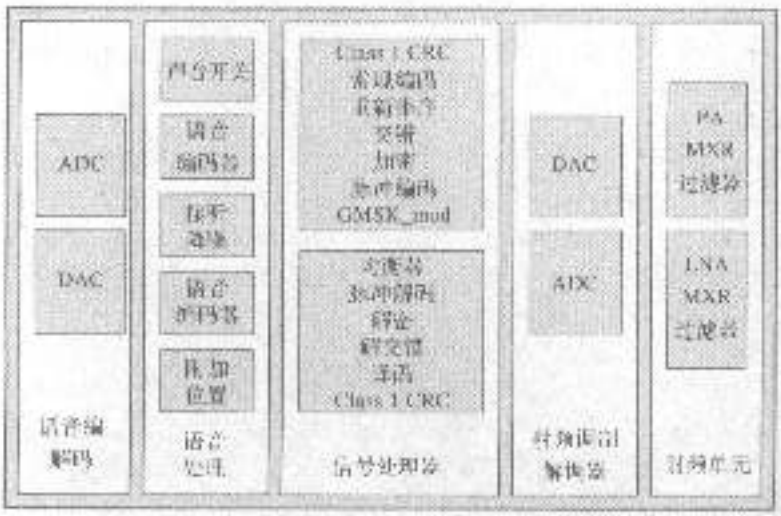


图16-13 GSM手持电话信号处理示意图（Redl等著，1995）

输最大160字节的文本消息。因此，在语音连接处于活动状态时，也可以发送和接收短消息，以需短的"电路交换模式"将GSM手持设备连接到PC，以接收或发送传真或数据时，最大传输速率是9600 b/s，这个速率要慢于固定电话。SMS功能方便，能够传输较短的文本消息，但不能扩展到耗时较长的文件传输。不过，现在研究存在提供转发服务的SMS——电子邮件网关，允许移动电话发送和接收内容较短的电子邮件。

GSM蜂窝式网络由三部分组成：移动电话、基站及天线，以及移动电话交换中心。我们现在对移动电话已相当熟悉，但对另外两个构成基础设施不可或缺的部分知识比较陌生。根据GSM的规定，我们可以将SIM（Subscriber Identity Module，用户标识模块）以智能卡的形式插入手持设备中（参见图16-14），以增加用户安全性。任何移动设备只有在插入新的SIM卡后，用户才能用它来拨打或接听电话。每个基站到格都通过基站子系统将移动电话交换中心与移动设备连接起来。基站子系统由无线电收发电台和基站控制器组成。网络根据基站天线的位置来定义。基站控制器管理无线电资源，并提供无线电信道与E1/TDM干线（用来与当地的移动电话交换中心通信）间的交换。移动电话交换中心控制呼叫信号的发送和处理，并协调手持设备从一个基站移动到另外一个基站时的交接工作。交换中心使用常规的点到点干线或微波通道，将多组邻近的基站链接起来。

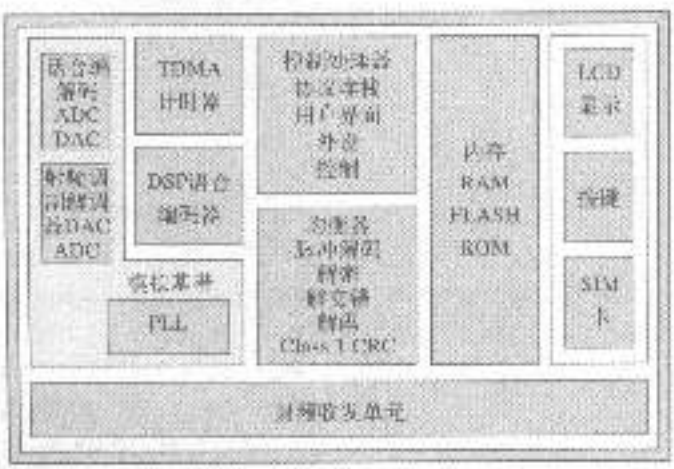


图16-14 GSM手持设备的功能模块（Redl等著，1995）

每个移动电话交换中心依次连接到本地的公共交换电话网络（PSTN或ISDN），以提供移动电话和固定电话用户之间，以及到全球其他移动网络的连接。移动电话交换中心还可以提供到公共数据网络（Public Data Network，PDN）的连接，比如包交换数据网，这样，用户还能够通过他们的移动设备发送或接收数据。

GSM需要使用大量的数据库管理手持设备的移动性。这是专为那些经常需要从一个服务系统转到另一个服务系统的漫游者设计的。系统内漫游的管理通过两个数据库完成：归属位置登记（Home Location Register，HLR）以及拜访位置登记（Visiting Location Register，VLR）。HLR维护和更新移动用户的位置以及他（或她）的服务信息。位置的更新可以帮助系统将输入的呼叫发送到移动电话。每个GSM网络逻辑上只有一个HLR。VLR含有从HLR得来的、进行呼叫控制和为漫游用户提供指定服务所必需的一些管理性信息。

为了提高GSM的数据传输能力，运营商在现有的基础上新增**GPRS**（General Packet Radio Service，通用分组无线业务）功能，它同时使用一帧内的所有八个时隙来运送数据，提供170 kb/s的带宽（见图16-15）。这样的数据速率能够支持移动电话、手持式Web浏览设备、远程电子邮件终端和类似的应用。每个具有GPRS功能的手持设备需要一个Internet编号和域名，如果目的终端有可能在网络中改变位置，那么如何将数据包有效地送达，就成为一个不易解决的问题。最近，一种新的Internet协议**WAP**（Wireless Application Protocol，无线应用协议）引入进来，它允许Web页面的作者指定页面的哪部分需要显示在移动电话上运行的微浏览器中。通过这种方式，那些更换了新WAP手持设备、拥有更大LCD屏幕的GSM用户，可以看到文字内容及一些图像。

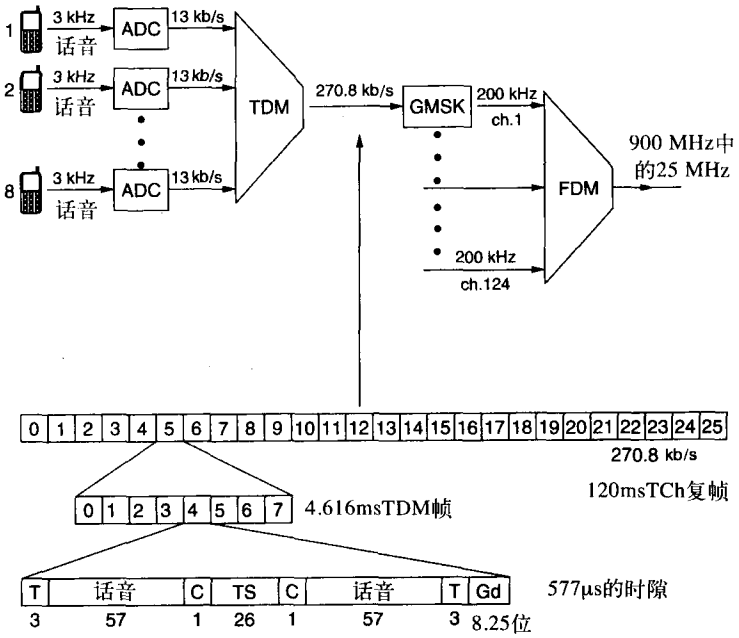


图16-15　GSM话音的包结构

GSM的数据结构和信号传输协议十分复杂。无线电频谱被划分成宽为200 kHz的波段，在10 MHz的波段内提供50个信道。数据传输使用相位调制技术，单个状态可以表示多个数据位。

通道中每个帧提供八个时隙。时隙接受148位数据包，其中含有114位话音数据。这些数据来自于将话音流压缩到13 kb/s的编码系统，它附加有错误纠正检验和，从而使速率提高到22.8 kb/s。如果想要使用有效载荷为114位的数据包成功地传送22.8 kb/s的数据，必须每5ms就得发送一个数据包（114/22.8=5）。实际上，由于额外的信号传输和控制信息，递送更加频繁，间隔大约为4.62ms。为

了进一步提高信号在恶劣条件下的成活率，错误纠正代码块作用到8个帧上，跨度为35ms，它增强了信号对突发损坏的抵抗性。

要注意，传统电话网络专有的8 kHz速率，已通过使用低比特率编码方案加以规避。每个帧有八个时隙，可以提供四个全双工的双向会话。如果时隙0分配给从基站到手持设备的传输，那么时隙2就可能用做相反方向的传输。包中间的26位字段用来完成接收方同步，以及提供当前信号强度的度量。通过它，可以有效地处理邻近建筑产生的多重反射对接收方的影响。编码后的话音数据在同步字段的两边。

射频传输的运行速率达到270.833 kb/s，每4.6ms能够完成8个数据包的顺序传输。

不同于之前的模拟方案，这个标准是国际通用的，这使得同一手持设备可以在整个欧洲使用。由于制造成本的下降，手持设备已经逐渐为大众所接受。某些国家电信设施可能比较落后，移动电话可以降低提供固定线路电话连接的压力。或许虚拟办公室和远程信息服务最终会成为现实。

GSM产业的一个副产品是，专为小型移动设备开发和优化的语音压缩技术不断发展。如16.1节所述，固定线路数字电话系统采用8位PCM方案，而对数压缩（A-law或μ-law）可以提供64 kb/s的比特流。GSM中使用的增强型参数化编解码，能够在16 kb/s（甚至可以降低到2.4 kb/s）上提供可以接受的语音质量。达到这么大压缩比的实时算法很耗CPU，有时需要20 MIPS的处理能力才能跟上人类的语音数据。所有这些都得装入小型的由电池驱动的设备中。使用专门设计的DSP可以加速算法的运算，以及执行越来越多的语音压缩。考虑到如果将电话会话所需的带宽减半，会使收入倍增这种情况，您就会了解电信公司为什么要为这类程序投入这么多资源。

16.3 ATM：异步传输模式

广域网提供局域网之间的长距离连接，所有这些网络构成了当今的Internet。如第14章和第15章所述，局域网和广域网使用的路由方法大不相同。局域网依赖于广播，而广域网需要路由交换才能将消息转送到它们的目的地。有几种网络技术可以用来支持广域网通信。英国的**PSS**（Packet Switched Service，包交换服务）、北美的SMDS（Switched Multimegabit Data Service）、帧中继、X25，以及最近引入的ATM。每种技术都有各自的优缺点，但此处我仅介绍ATM，因为它最有可能在将来获得成功。

ATM（Asynchronous Transfer Mode，异步传输模式）是真正的虚电路联网技术，由于Internet的不断成功，人们期望有一种统一并且相当灵活的方式来满足快速增长的传输需求，发展ATM就是为了响应这种发展趋势。如果单个网络既能够处理电话，又能够处理数据，则可节省许多资源。根据预测，如果能够解决观众通过电话线下载全长电影存在的技术问题，使他们不必到本地音像商店来回奔波，肯定会获得巨大的收益。标准家用电话线的带宽限制是一个需要克服的重大技术障碍——置换的成本在大多数领域是不可接受的，尽管一些提供有线电视的组织已经通过同时安装电视频道和升级电话连接对费用做了调控。

从图16-16我们可以看到，在通过电路交换网络建立呼叫时会有建立延迟，之后的工作速度就比数据报路由要快。这表示，从用户的角度，基于数据包的网络适用于短期的、偶发的通信，这类应用中频繁的启动开销是一种主要考虑因素。

ATM使用不同的策略。通信路线，或虚电路，是在会话开始时建立并保持不变，从而几乎消除了路由的开销。信元，或称小型数据包，并不承载完整的地址，而是由ATM路由器根据信元报头中的24位数完成交换。为了帮助重新路由和网络通信的组织管理，多个虚电路常常被归为一组，形成虚路径。在看完信元报头（见图16-17），并且认识到VPI（Virtual Path Indicator）和VCI（Virtual Circuit Indicator）两部分的划分之后，这种通信的机理更加明显。大多数数据信元通过网络时，仅仅上部的值（VPI）用以选择路线；只有在最后的交换中，使用VCI来选择通往客户设备的目的线路。

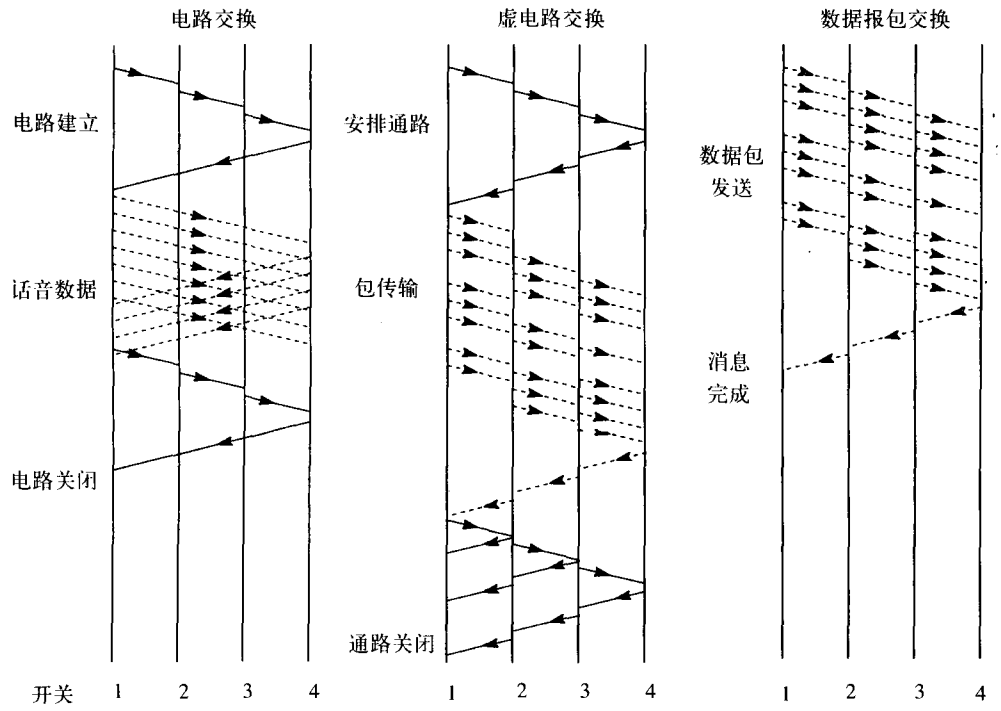


图16-16　电路交换和包交换时序的对比

ATM协议和路由算法的发展伴随线路驱动电路的飞跃。各种线路驱动技术通过降低电子噪声的影响以及通道间的交叉干扰，使工程师能够远远超越早期线路速度的限制。因而，当RS232最大只能达到19 600 b/s，以太网在10 Mb/s上下时，新型的驱动将通过无屏蔽双绞线的传输速度提升到52 Mb/s，在质量更好的电缆上，甚至能够到达155 Mb/s。采用同样的技术，可以将以太网升级到100 Mb/s。达到这么高的数据速率，确实需要一些复杂的编码方法，比如8B6T，如表14-3所示。

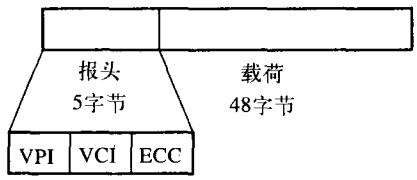


图16-17　ATM帧结构

建立ATM网络的意图是承载恒定速率的通信流量、声音和视频，以及变速率的数据传输。ATM的基本传输速率是155 Mb/s，可以使用光纤，也可以使用相对廉价的双绞线。尽管话音数据流的本性要求传输必须严格同步，但在不超载的情况下，ATM网络如此之快的速度应该能够传输64 kb/s的流。ATM使用类似于电话PABX的中心电路交换，但它提供的是小型数据包递送服务，而非电路租用。

ATM遵照局域网处理数据的办法，将数据拆分成数据包进行传送，不同的是，它使用更小的固定大小数据包——**信元**（cell）。信元由53个字节构成，其中48个字节为数据，报头中的5个字节为寻址信息保留。为降低路由交换时的处理时间，这个报头比以太网和IP中的报头要简单得多。

承载消息的信元都沿预先建立好的路线传递，因此，ATM是交换虚电路，是一种面向连接的技术。通过使用虚电路，ATM为传输提供最低带宽保证，并能够保证通过延迟不超过特定的值。呼叫的建立通过发送SETUP消息完成，在会话进行期间，带宽一直保留。人们在处理突发传输上投入了大量的精力，避免突发传输妨碍其他用户，同时保证传输的平均值依旧保持在最大流量范围以内。

ATM交换机有多种型号，如图16-18所示，能够提供16到1024个端口。它通过在交换机内部建立一个路由表，将每个活动的输入端口与一个输出端口关联起来，完成路由选择。到达的数据信元在报头中有一个代码编号，它被用来选择端口对，以将数据信元正确地路由到输出端口。数据信元

报头内的路由代码在整个旅程中并非保持不变。每次交换都会修改，以适应下一个节点。由于要从几个输入端口传递数据，每个都在155 Mb/s，因此交换机制需要尽可能地快。这就排除了基于软件查表的方式。大部分提供商越来越多地依赖于可配置的门矩阵（见4.4节）。除基本的路由功能——从输入端口到输出端口以外，ATM交换机必须能够处理冲突。冲突发生在两个输入信元同时请求同一输出端口的时候。为了处理这种情况，必须丢弃一个信元，或暂停其中一个信元的传输，让另一信元通过。后一种选择更通用。

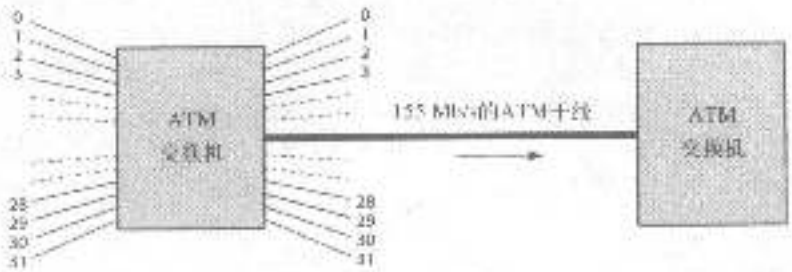


图16-18 ATM路由交换的示意图

图16-19给出8×8（3位地址）ATM交换机的示意性结构。市场上的交换机能够处理更多的输入—输出端口。因为这种复杂的互连通道，人们将这种类型的交换机称为Banyan交换机。整个设备由2×2交换元件构成，这种元件的连线方式是为提供到到列的路由。内部的路由逻辑叫做"构造"（fabric），因而Banyan交换也称做"自路由构造"。左侧的任何输入可以接到右侧的任何输出。在每个交换元件中，地址位为0选择其中一个输出，地址位为1选择另一个输出。交换机的第一列使用地址的最高符号位，中部的交换由地址中间的位控制，最后，右侧的列由地址的最低符号位切换。

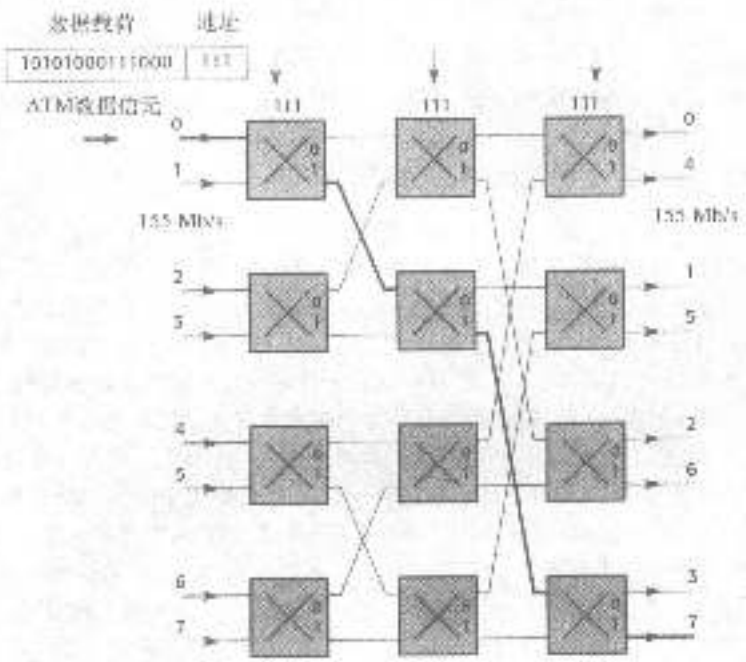


图16-19 ATM Banyan交换机内部的互连

在地址为7（111）的假想信元到达时，第一个开关选择较低的输出端口，将数据信元传输到中间的列。同样根据地址中间位的值（1）又是较低的端口被选择。最后，在最后的交换阶段，又是

较低的端口被选定，从而将数据传送到端口7。8个输入端口均可以使用这种方式工作。接下来可以试试不同的3位地址值。

带自路由构造的ATM交换机，其行为在某种方式上类似于缩缩版的Internet。下一跳地址随数据信元在构造中的传递而确定，和数据报由中转路由中心进行定向的方式相同。

如果多个交换单元的输入都要转到同一输出端口，就会发生冲突。解决这种交通拥塞的方法有几种。可以提供存储缓冲区保存数据，之后再发送，或者将交换速率提高至到达速率的双倍，这样就可以依次轮询处理两个输入。为了检查这些技术是否能够有效地降低数据包的丢失率，我们使用计算机模拟大型Banyan交换机输入满载时运行的结果。在内部链路以两倍于信元速率运行（因而能够在一个时隙中建立两条并行通路），在每个交换单元配备可容纳5个信元的缓冲区的情况下，可以安全地递送多达92%的输入信元。而不缓冲的方式只能达到25%。

ATM网络是由互相连接的几个ATM交换机组成的。考虑到外围的局域网和服务器时，我们可以看到Internet是如何形成的（见图16-20）。

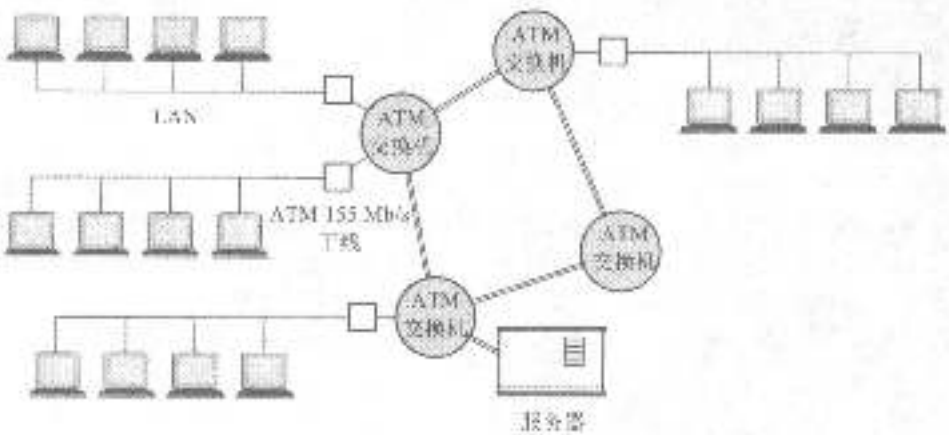


图16-20　ATM广域网链路将各种局域网结合起来，形成Internet的一部分

16.4　消息传递：无线寻呼和分组无线网络

在蜂窝式移动电话出现之前，通过无线终端接收或发送文本消息并没有吸引太多公众的关注。最初的寻呼机——常用在医院中，现已发展到能够处理数字代码、文本消息甚至较短的语音。使用普通的电话网络与某人建立联系（不管他身在何处）的能力是一项有价值的服务，同时它更廉价，也比蜂窝式移动电话服务更可靠。寻呼机是单向的通信媒介，其优点是不需要昂贵的充电电池，一次性电池可以持续供电3个月不用替换。通常使用的信号发送方法是载波频率调制。国际西欧无线电寻呼系统中，每次寻呼发送由6个100ms的音频脉冲串组成的序列。频率从10个不连续的音频中选取，因此可以提供上百万的地址，用以标识寻呼机。寻呼的间隔为200ms，最大寻呼频率是1.25个每秒。最新的数字寻呼系统使用更复杂的编码系统。ITU接受英国的POCAG（Post Office Code Standardization Advisory Group）协议作为国际寻呼机的标准。它使用20位地址，这20位地址连同10位BCH错误检测校验和一同封装到一个32位的单元（或称代码字）中。每个地址还可以选择四种动作之一，或使用2位功能码进行显示。在发送时，代码字都以16个为一组成批发送，前面为16位的同步报头。发送数字或字母消息时，地址字后面的代码字包含数据。这是通过前导位标记的。0表示地址字，1表示数据字。消息可以为任意长度，在下一个地址字到达时结束。寻呼机传输的带宽十分低（1024 b/s），但传输装置所覆盖的地理范围极大，出错的机率也相对较低。这是必需的，因为没有办法确认寻呼消息的成功接收，除非目标用户通过电话报告。数字寻呼字的格式见图16-21。

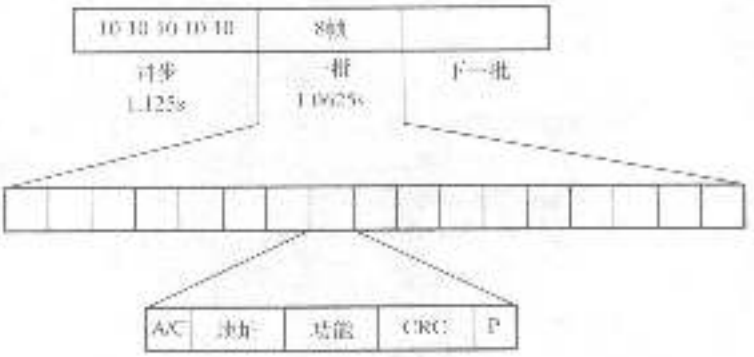


图16-21 数字寻呼字的格式

整个欧洲使用的国际字母消息标准都使用169 MHz波段。在购置寻呼机后，服务运营商提供接寻呼次数收费的计费方式。这就降低了管理性的开支，因为可以根据呼叫者的电话账单收集付费信息。寻呼机网络使用多重频率FSK系统对二进制数据进行编码。和上一代的电话调制解调器所使用的方式极为相似。如果希望寻呼机不止提供基本的警示服务，则需要配备数字或字母显示元件。这些元件常常是多字符的LCD面板，通过2 KB的静态CMOS RAM存储消息。最新的进展是手表寻呼机的发展，它提供LCD显示面板，覆盖范围很广的基于卫星的系统也已存在，适用于一些特殊的应用。

另一种双向消息服务基于**分组无线**（packet radio）技术。最初开发它是为了支持从事快递和服务的人员，但现在已经扩展到其他应用。比如街边停车位（见图16-22）和应急车辆的指挥，这个技术与最初在夏威夷Aloha无线广播中得来的经验紧密相关。当时，一个运营分组无线服务的运营商将自己拥有的无线电频段划分成30个通道，每个通道的数据传输速率为8 kb/s，消息包的大小为512字节。它并非即时发送，而是存储下来，等网络方便的时候才发送出去。这是一种文本寻呼服务，它还允许移动接收设备确认甚至回复消息。通信是秘密的，不连接到Internet。这个领域在不断发展，便携、完全兼容Internet的电子邮件终端也已经在市场上出现。这些设备需要有自己的IP编号，如何将IP数据包递送到移动终端是个问题。解决方案之一是，将所有数据包交给中心主机，由它来跟踪手持设备并转发数据包。这类系统不使用基于电路交换的蜂窝式技术，无线电带宽的共享方式类似于以太局域网，它以"先到先服务"的调度方式从许多发振接收数据，发送和接收站服务于邻近约25英里（40km）的区域，比GSM网络要大。

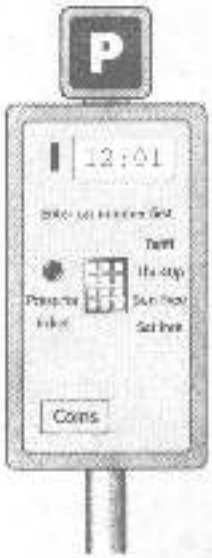


图16-22 分组无线消息递送的应用

16.5 ISDN：全数字

PTT在20世纪80年代早期开始计划大规模地将电话设备转换到全数字技术时，就已经预见到变革的巨大成本和困难。有趣的是，移动电话公司采用数字技术的步伐很快超过传统的固话运营商。在欧洲，家庭用户安装数字线路的寥寥无几。商用场所数字线路的采用要高些，但依旧远远落后于早先的乐观预期。也许机会已经逝去，ISDN技术将被最近基于有线电视网和DSL调制解调器的技术所取代。

ISDN（Integrated Services Digital Network，综合业务数字网）最初的目的传输数据和用户会话，增加端到端的带宽。传统电话线路只有4 kHz的带宽，因而提供64 kb/s显然对用户很有吸引力，可能正是基于这种判断，才确定这个目标。最初引入4 kHz的限制，是为了降低邻近呼叫之间相互

的电子干扰，从而最大限制地利用传输和交换设备。为达到64 kb/s的传输速率，实现ISDN需要彻底修改交换中心。同时还需要用户购买新的电话和接口设备。不管基于什么原因，这种方案未能实现，ISDN电话非常少见。但是，之前家庭用户对宽带模拟电话线的接受，最近也受到Internet普及的影响，他们对于渐长的下载时间越来越没有耐心，尽管更复杂的调制解调器可以在单个时钟脉冲内传输多个位（参见图16-9），但4 kHz的限制依旧存在。ISDN依然是一种超越这种构架的方式，它可以使用户得益于两条并行的64 kb/s话音和数据通道。

ISDN有两个版本：宽带（B-ISDN）和窄带（N-ISDN）。后者是为常规客户提供的服务（见图16-23），也常被称做2B-1D ISDN，因为它提供两个64 kb/s的基本话音通道以及一个16 kb/s的数据通道。这二个通道均为全双工。ISDN网络的接口由ITU定义，分别有R、S、T和U。NT1网络端子提供T标准点，供客户连接他们的ISDN兼容设备，专门的终端适配器还提供R接口，这是一种通用的非ISDN连接。

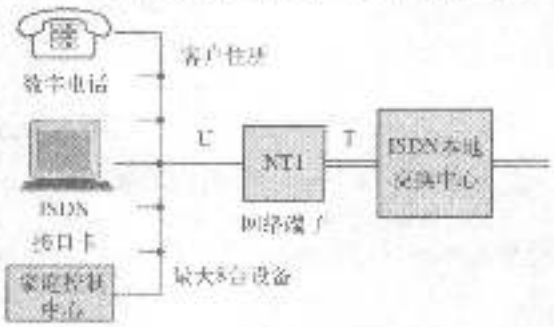


图16-23　窄带ISDN定义的接口

N-ISDN传输以基带信号进行，和以太网相似，但与最近的宽带（载波调制）服务相比，速度相对较慢。B-ISDN（宽带）一般只用于高容量的干线，因为它能够承载30路话音通道和1个公共信号传输通道。它现在基于ATM技术，在16.3节中已做过介绍。比较出人意料的是，N-ISDN是一种电路交换协议，而B-ISDN是包交换，尽管在实际的部署中使用虚电路模式。

窄带ISDN是为了利用本地交换局到客户家之间的现有低级双绞线而设计的。在这样的条件下，它可以提供两路数字话音通道，一个低速率数据通道。如果我们回到ISDN出现之初，技术上的看法就会大不相同。当时，计算机终端只提供字符显示，因而只需要1.2 kb/s（或9.6 kb/s——幸运的话）的串行链路。相比之下，话音成为高容量的数据流，需要64 kb/s。今天，情况已经有比较大的变化。现在，屏幕直接通过较短的高带宽电缆（能够传送30 MHz的视频信号）连接到PC上。局域网已从10 Mb/s升级到100 Mb/s。传输复杂图像的需求不断增长，视频压缩方面取得的成功完全可以和声音压缩相比拟。情况已经有了180°的变化——处理话音通信量已经不再是速度目标。商业兴趣现在集中在使用电话网络传输全长视频电影的可能性上。ISDN必须快速跟上这种变化，否则它就只能逐渐消失。

由于现有设备每秒处理8000次8位的数字转换，因此ISDN基本话音通道被设为64 kb/s。这样的通道，即使都充分利用起来，也不能满足当前在线视频的需求。当前家庭用户可以购买56 kb/s的调制解调器，公司办公室一般安装100 Mb/s的交换式以太局域网。付费升级到64 kb/s并没有什么太大的吸引力。相应地，是否应该继续开发ISDN技术也成了问题。现在正在争论的是新的技术，比如DSL，是否能够超越和替代2B-1D ISDN的角色，为客户提供固定线路连接。

N-ISDN用户连接采纳的基带（非调制）传输技术以全双工模式进行数字化传输，如图16-24所示。仅当电缆使用类似于V.32调制解调器（见10.9节）中使用的回声消除电路时，才可能实现全双工。为了进一步增加数据传输速率，在数字传输中还采用了多级传输码（见图16-25）。在欧洲，这需要三个电平（+V，0，−V），在美国使用四个电平（+V/2，+V/4，−V/4，−V/2）。这样，四位数据就可以使用二个三进制数表示，或者，在美国，两个位可以使用一个四进制数传送。注意数据帧中数据样本的排列，其中含有全部（话音）通道的两个采样，同样还包括D（数据）通道。

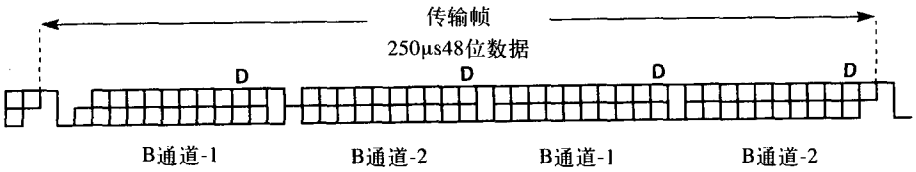


图16-24 2B-1D窄带ISDN协议时序

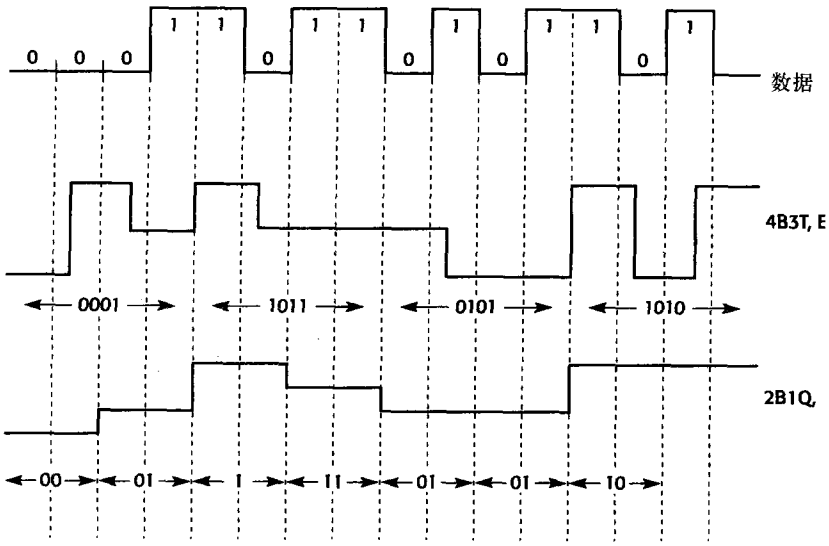


图16-25 多级基带编码增加数据位传输率

全部三级编码在表16-2中给出。由于可用的编码（3^3=27）多于传输16个数字所需的编码。额外的编码用来提供其他的值，维护线路在消息传输期间平均值为0。这可以协助电子工程师构建他们的电路。

表16-2 三级4B3T编码表

二进制数据	3级编码	
0000	+0−	
0001	−+0	
0010	0−+	
0011	+−0	
0100	++0	−−0
0101	0++	0−−
0110	+0+	−0−
0111	+++	−−−
1000	++−	−−+
1001	−++	+−−
1010	+−+	−+−
1011	+00	−00
1100	0+0	0−0
1101	00+	00−
1110	0+−	
1111	−0+	

16.6　DSL：数字用户线路

多年来，人们认为限制电话线传输速率的主要因素是铁质的铜线本地回路。但是，最近这种观点已经有所改变。人们都没有注意到电信行业使用同样类型的电缆，能够驱动30通道的PCM干线，传输速率达到2.048 Mb/s。现在一种新的提案已出现，它可以为家庭或小型办公室提供高速（1 Mb/s）的数据服务：DSL（Digital Subscriber Line，数字用户线路）。它采用保持现有的模拟话音技术，但它使用新型的高速调制解调器共享同一线路。见图16-26。线路共享是通过使用可听波段以外的载波频率进行数据传输来做到的。这也提高了最大数据速率的限制。为了避免宽带数据通过电话交换设备，数据和声音信号在进入本地交换局时被分开，声音遵照传统的路线，而数据，在解调制之后，进入基于ATM的网络。通过这种方式，用户依旧可以使用常规的拨号电话网络，同时可以直接访问数据网络，如同在办公室局域网一样。

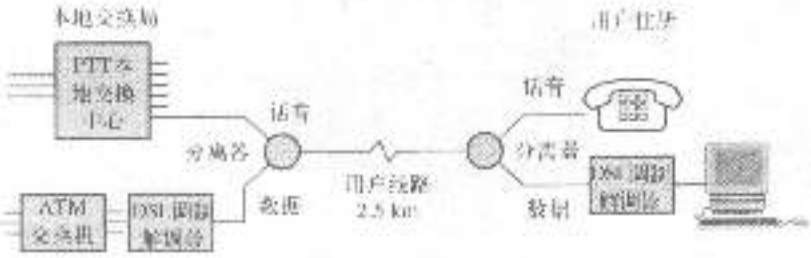


图16-26　数字用户线路的配置

异步数字用户线路（Asynchronous Digital Subscriber Line，ADSL）通过不对称地划分带宽，响应用户对快速Internet下载的更大需求。最佳情况下，文件下载可以达到4 Mb/s，而上行链接只有1 Mb/s（不要忘记电话呼叫也可以同时发生）。带宽分配由ADSL调制解调器和分离器完成。如图16-27所示。这个设备将频率带宽划分成4 kHz的话音通道，135 kHz的数据上行通道和1540 kHz的下载通道。通过采用10.9节所述的多级编码方案，比特率可以超过载波频率。因此，使用8级方案（QAM），我们可以获得4 Mb/s的下载数据和1 Mb/s的上行链接。这二个波段的相对分离，允许数据/话音的分离可以直接使用电子滤波完成，从而避免了复杂的回声消除电路的使用。

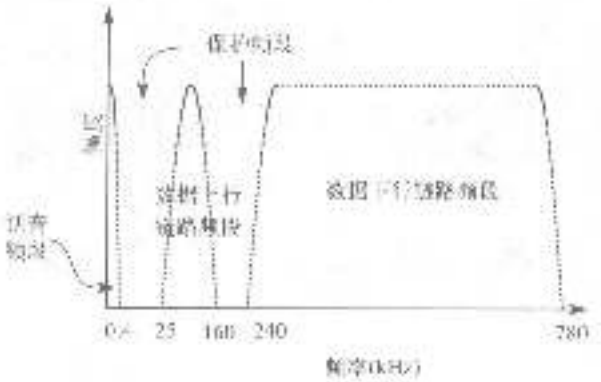


图16-27　ADSL通信中用户线路带宽的分配

目前，电信行业正在为DSL全球标准的建立而斗争。下面列出一些互相竞争的技术，早期用户对于DSL技术的混淆也可以由此得到解决：ADSL、ADSL-Lite、G.Lite、SDSL、HDSL、HDSL和VDSL。内置式PCI调制解调器已经可以买到，但在当地PTT支持DSL服务之前，它们肯定是毫无用处。

16.7　有线电视：数据传输设施

最初建设有线网络是为传输电视频道，但它也可以用来通话。英图通过使用光纤干线和同轴电缆/双绞线的混合技术，已经大体上实现了这种应用。传统的用户电话需要几千米的电缆连接到交换中心，而电缆的前级设备直接安装在邻近的街道上，这些接口将来自于高容量光纤通道的数据解调制和解多路复用，将电话和电视信号分离开来，然后将信号沿铜线传送到用户的家中。这就如同将

交换设备直接安装在马路边。图16-28说明了这种新型网络的组织方式。术语混合光纤同轴电缆网（Hybrid Fibre-Coax，HFC）指的就是这种由光纤、同轴电缆和双绞线构成的网络。本地干线使用光纤的优势是大幅地提高用户的潜在带宽。到目前为止，它已经应用到模拟电视广播和模拟电话中，但是，向数字编码的转变依旧在进行中。总部将来源于卫星和地面源的各种电视信号集合到一起。通过频移将它们多路复用到一起。连同经过调制的电话信号，一同传送到光发射器，传递给分布式光纤。在每个街边安装的设备中，光纤传送的光信号被转换成相应的电子信号。发送到用户的机顶盒，电视和电话信号就是在这里分离开来。VHF电视频道沿双芯电缆的同轴部分发送。这里，VHF电视信号被转换成UHF波段，以和现代的电视接收器和视频录像机兼容。双芯电缆的另一条线路提供几对双绞线，其中之一用于传输传统的模拟电话信号。

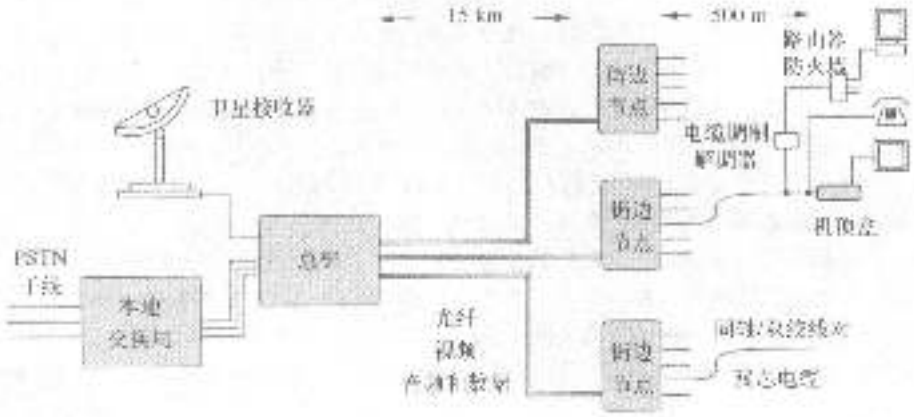


图16-28 有线电视电话网

745 MHz带宽的大部分用于传输59个电视频道，仅12 MHz仅电话使用。此处采用的是VHF模拟电视信号，它被调制到PHz（10^{15}Hz的光振荡器）的载波上。通过转向数字化，同样的波段可以传送700个数字压缩（MPEG-2）的视频流，供数字电视使用。

从图16-29所示的频谱分配方案可以得出，这个方案为交互式数字通信保留了相当大的区域，不同于传统的电话网络，邻域网的电缆远远没有饱和。



图16-29 邻域电缆网络的示意性带宽分配

高频有线调制解调器的出现加速了Internet的访问。它们的作用与传统调制解调器相同，但能够提供最高10 Mb/s的传输速率，而非56 kb/s，邻近的用户似乎共享10 Mb/s的访问通道，提供等同于以太网10Base2标准的局域网功能。遗憾的是，这种方案最终会受到同样问题的困扰，即：大量的需求可能会导致传输延迟不可预测。

通过电缆传递数据的新标准是**DOCSIS**（Data Over Cable Service Interface Specification，有线传输数据业务接口规范），对于克服不同网络设备的差异进行用户间通信来说，标准至关重要。

家庭用户常常也需要连接多台计算机到宽带有线调制解调器。推荐的方式是再投资购买一台防火墙路由器，它能够保护小型的家庭局域网，阻止不受欢迎的各种入侵，并提供多达8个以太网接

口，或者提供WiFi功能，连接更多设备（有时还包括那些满心欢喜的邻居）。

首先，我们得克服IP地址的技术问题。因为IP地址是严重缺乏的资源，许多Internet服务提供商（Internet Service Provider，ISP）仅为每个客户分配一个地址。大多数情况下，这个地址是由DHCP动态分配，因此，客户每次连接到ISP都可能会得到不同的地址。但是，不同的会话中，本地域名可以保持不变，这是因为有动态DNS服务器。大型的站点会购买更多的IP地址，但对于小型企业和家庭用户，它们一般只能使用单个动态变化的IP编号。表面上，这就限制这些用户在任何时候只能有一台计算机连接到Internet。但是，通过使用NAT网关，我们可以让多台本地计算机共享单个IP地址，从而能够同时访问Internet。外面并不知道这种情况，它们只能看到作为网关的IP地址。

所有连接到局域网的设备都需要不同的MAC和IP编号，这样才能保证数据包能够正确地路由。在只有一个IP编号可用的情况下，我们可以采用NAT（Network Address Translation，网络地址转换）服务提供的网关-路由器来解决这个问题。在局域网内采用了一系列非正式IP编号的情况下，NAT网关-路由器会截取每个向外部传递的数据包，将它的源IP地址字段改为ISP分配给我们的正式IP。因此，从外面无从知晓私有局域网中的非正式IP地址。当应答数据包传回时，问题就会发生。此时，网关-路由器必须确定本地哪台PC发送了初始的请求。通常情况下，目的IP地址字段保存了该信息，但现在这个字段中保存的是由ISP分配的有线调制解调器的IP地址。NAT通过接管所有输出数据包的**源端口**字段，用标识该PC的数字替换已有的端口值，来解决这个问题。当应答数据包到达时，NAT网关－路由器只需查看源端口字段，找出原来的PC，恢复它的IP编号，使数据包能够完成最后一段旅程，顺利地通过局域网传递给该PC。图15-3给出更多此处讨论的数据包中各个字段的细节。

NAT网关-路由器隐藏了连接到局域网上的PC的IP地址，使得它们更加安全。人们常常使用网关-路由器对数据包进行过滤，忽略Telnet、FTP和HTTP请求，从而阻止外界对本地系统的攻击。

16.8　小结

- 电话网络路由器使用电路交换。现在，这可能已经不再是端到端间的直接铜线通道，而是转变成带宽资源的分配，并据此收取费用。数据传输可以使用调制解调器进行。
- 在长距离的传输时，声音信号需要数字化，本地数字连接（ISDN），除特殊的商业需要以外，一般还是相当少见。干线一般使用32通道时分多路复用（TDM），传输速率达到2.048 Mb/s。
- 互连交换设备可以使用交叉点开关矩阵、时隙偏移或两者的组合完成呼叫的路由。
- 通过ISA/PCI接口的电话卡（CTI），我们可以获得建立呼叫中心所需的完全集成的数据/电信功能。TAPI编程接口支持这种类型的开发。
- GSM移动电话网络是无线与地面线路混合型的方案，它取得了巨大的成功。除话音外，通过低容量控制信道的SMS，以及170 kb/s的GPRS直接数字连接，它还提供传统的使用调制解调器的数据传输。
- ATM技术用于中继话音和数据线路。它是技术融合的又一实例。它使用固定长度的包（信元）和虚电路路由。
- 分组无线传输系统，尽管不那么有名，也可以为中低容量的应用提供无连接的数据传输。我们可以将它看做是一种双向寻呼扩展。
- ISDN全数字网络好像已经走到终点。用户版（N-ISDN）开销太大，它为最终用户提供的带宽（64 kb/s）现在看来已没有什么吸引力。它的性能受限于用户回路的基带传输，升级交换设备的费用高昂。长距离干线版（B-ISDN）已经融合到ATM（155 Mb/s）中。
- DSL提供从家庭到交换设备的射频调制解调器，能够和模拟声音波段共存。使用现有的双绞线，它可以为用户提供最大10 Mb/s的传输速率。
- 有线电视公司，使用付费电视的收入，安装了新型的混合光纤同轴电缆网基础设施。随着设备的安装，数字系统可以作为最初的模拟传输方案的有力补充。其带宽和ADSL相当——除非所有的邻居都在线。

实习作业

实习部分涉及一些研究工作，使用Internet获取移动电话的当前状态以及下一代（UMTS）的规划。尝试使用GSM移动电话检测Internet和电话网络提供商之间的网关。

练习

1. 分配给模拟电话的带宽一般是多少（以Hz为单位）？在8位数字化后需要多少b/s？广播质量的立体声音乐的带宽是多少？16位采样数字化后需要多少b/s？
2. 模拟电话线的频率上限是3.3 kHz，在这么有限的频率上，调制解调器如何完成高数据速率的传输呢（56 kb/s）？
3. 为了降低通话所需的带宽，可以采用哪些压缩技术呢？GSM使用哪种技术？
4. 分别画出AM、FM和PM调制技术的载波。试着组合这三项技术。
5. 局域网、IP和ATM路由方法的主要区别是什么？
6. 画出7六边形方案的网格重复模式。说说为什么4六边形重复方案中分配同一频段的网格之间的干扰会增加？
7. 为什么当前GSM数据传输的速率限制在9600 b/s？GSM中，SMS和GPRS使用的传输技术有什么不同？
8. 电话编解码器和调制解调器都将信号从模拟转换成数字，然后再转换回来，那这两者之间有什么区别呢？
9. N-ISDN提供的最大比特率是多少？与最新的ADSL标准相比如何？传输模拟电视信号需要6 MHz。在数字化且MPEG压缩后，可以降低到1.5 Mb/s的数据流。这些数据是如何传输到家庭中的呢？
10. 时分交换机需要传递8位采样的话音数据，对于每个处于连接状态的线路，每125μs必须传送一次采样数据。如果有1000条线路，那么可以接受的最大读写周期时间是多少？我们是否能够使用60ns的EDO DRAM作为数据缓冲区？

课外读物

- Tanenbaum（2002）。
- Comer（2003）。
- Macario（1997）。
- Schiller（1999）。
- Harte（2005）——介绍GSM技术。
- ATM技术，尤其是交换机设计，参见：
  http://www.rad.com/networks/2004/atm/main.htm
- GSM相关的信息及新闻：
  http://www.gsmworld.com
- 一些面向技术的引导性课程：
  http://www.cs.bris.ac.uk/Teaching/Resources/COMSM0106
- 一系列介绍GSM电话通信的技术文章：
  http://www.palowireless.com/gsm/tutorials.asp
- ADSL连接的介绍：
  http://compnetworking.about.com/od/dsldigitalsubscriberline
- Netgear设计和制造小型的路由器防火墙设备，可以和有线调制解调器和DSL调制解调器一同使用，构建家庭局域网：

http://www.netgear.com/products

- ISDN的介绍，参见：

http://www.ralphb.net/ISDN/

- 有关NAT的介绍：

http://www.vicomsoft.com/knowledge/reference/nat.html

- 想了解更多GSM技术，可以参见Redl等人的著作（1995）。
- 这些网站可以通过本书的配套网站访问：

http://www.pearsoned.co.uk/williams

第17章 操作系统

操作系统（Unix、Windows 95/98和NT/2000/XP、Linux）是一些特殊的程序或者程序组，它们专为帮助用户更安全、更高效地使用计算机的资源而设计。在多用户操作系统甚至单用户多任务操作系统上，共享资源的分配、仲裁和控制都是首要的功能。如果计算机由大量用户共享，数据的安全问题也会变得极为重要。越来越明显的趋势是，操作系统要能更便捷地访问网络和Internet。Unix仍是操作系统在商业应用上的成功范例，并且随着Linux版本的引入，正在吸引越来越多的爱好者加入到Unix和Linux的行列。

17.1 历史渊源：基本功能的发展

前面的章节已经介绍了操作系统一些重要的组成部分：存储管理、联网、WIMP界面、IO设备驱动程序。本章将会把它们结合成有机的整体，介绍它们之间的互动。前面已经提到过，相比于任何其他软硬件单元，操作系统更能够表现计算机的特征，用户可能根本不会注意到CPU的改变，但从MS-DOS到Windows，或者从VMS到Unix的变化，没有人会忽略！操作系统是硬件和用户应用程序代码的中间层。由于中间层的存在，应用程序代码可以便利地从一个硬件平台移植到另一个硬件平台，只要两个硬件平台都运行同样的操作系统，或最起码同一套API。API（Application Programming Interface，应用编程接口）是操作系统为应用软件提供的一套函数调用。Microsoft为其所有的操作系统：98、NT、CE、XP和2000，提供一套标准的API——WIN32。Unix也有一套类似的函数，提供所有常规的操作系统功能。在移植软件时，如果最初的平台和新平台提供不同的API，则移植过程可能会极为耗时，且充满未知的困难。

操作系统可以大致分为二类：批处理、在线或实时，参见图17-1中的示意图。批处理系统现已

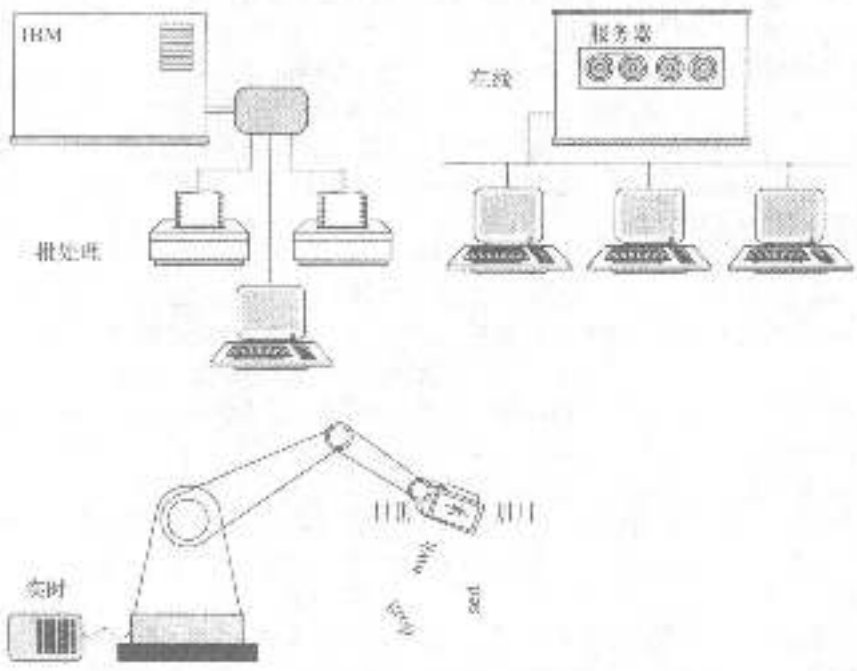


图17-1 计算机操作系统的类型

不如以前那样流行，或许因为它们仅在大型、昂贵的计算机上提供。另外，它们的基本结构也不允许正在执行的程序与用户的终端直接通信。对程序员讲，这没有什么。在线、分时共享、多用户的系统更易于进行程序开发和交互式计算，用户对CPU和其他资源的共享，是通过操作系统快速地在程序间切换读取一执行然点来完成的。在线系统又可以进一步划分成分时共享系统和个人计算系统。前者的实例可以见于旅游公司的业务处理。他们可以从大型的中心数据库中读取假日和旅行信息，或者见于银行对客户账户记录的更新。后一种类型的系统可以在家庭或办公室中看到。最后，实时系统专门用于需要对事件做出迅速响应的应用。常常用于设备和机器的监控和控制，但现在这个定义已经扩展到将类似于机顶盒和数据交换设备等都包括进来。实际上，这三种类别有相当大的重叠，各种操作系统提供的多种多样的功能也模糊了这种区分。例如，Unix提供批处理功能。实时执行系统VXWorks（由Wind River开发）提供不错的文件管理器和几种交互式外壳。

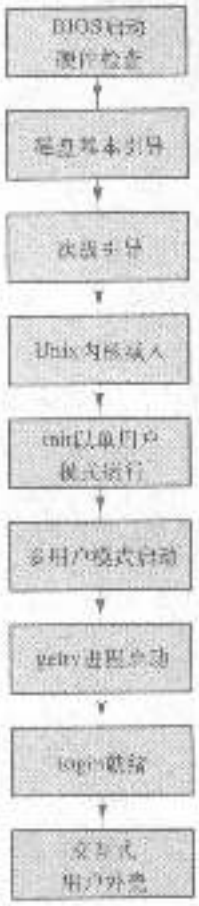


图17-2　Unix启动序列

小型嵌入式微处理器系统有时会完全不依赖于操作系统而运行。应用程序负责所有的任务调度，IO处理和内存分配。如果所需的功能比较固定且简单，这样做在开始时可能会成功，而且产品会十分紧凑和高效。但如果需要进一步的开发，或者应用程序必须响应未知环境的需求，这种方式最终就会难以为继。

如果没有程序，单纯的计算机硬件不能完成任何事。如何将第一个程序载入到裸内存中的问题，就是我们常说的引导问题。在20世纪60年代中期，人们必须使用二进制的前置面板开关，手工将一小段例程“切换”到内存中。现在，计算机内BIOS PROM中提供一个永久性的载入程序，在电源打开后，就可以立即执行。这是一个复杂的序列，如图17-2所示，在此期间需要完成硬件的检查和初始化，软件的载入和安装，最后是启动多用户登录界面。这种情况下，负责引导的载入程序会载入更为复杂的载入程序，由它从磁盘上将操作系统的主程序引入进来。因此，操作系统代码的一个重要功能，就是将自己载入到内存中，准备接受用户进一步的命令。

早期计算机的另一个缺点是，它们在同一时间只能服务于一个用户，它们是完全的单用户系统。早期的程序员，在编写的程序崩溃后，除能够使用基本的监视软件得到二进制代码的转储以外，几乎得不到其他方面的帮助。调试是一项耗时耗力的过程。甚至访问IO和辅助性的磁带存储设备，都需要应用程序的程序员亲力亲为。这种重复工作没有效率，人们很快就开始共享程序员编写并测试过的载入程序、IO和数学例程。令情况更坏的是，似乎早期的程序开发大部分是在午夜完成。因为那时计算机的时间更空闲些！当时，程序设计只能使用汇编代码，因为当时编译器还未发明出来。

第一代操作系统不时需要技术人员的操控，技术人员需要手动将用户程序从一叠叠的穿孔卡片装入到任务队列中，之后，在程序逐条执行时，他们需要密切监视。大多数用户编写他们自己的软件；程序员没有什么工具可以使用，因而程序的编译成为重要的考虑因素。为提高吞吐量，编译时间常常需进行组织安排。如果程序员想编译一个FORTRAN程序，可能必须得等到周二。那时FORTRAN编译器会载入计算机中！或许COBOL是周三。这是因为从磁带载入编译程序极为耗时。更令人失望的是，用户不可能和他们编写的正在运行的程序交互。输入数据从预先准备好的卡片上读入，输出一般发送到行打印机。如果程序崩溃——经常会发生，计算机会在纸上打印出八进制核心转储，供调试之用。在那些日子里，程序设计不但需要技术能力，而且还要具有不同不挠的精神。

随着磁盘设备的容量变得越来越大，用户开始在线存储他们的数据，因为这样更可靠。这有助于加速数据处理过程，但也带来新的问题。在您个人所有的磁带上输出数据块是可以接受的，因为程序设计中的任何错误都只影响到您本人。但访问共享的磁盘则会存在一系列的新风险。同时，对

于大多数程序员，控制主轴、磁头、磁道和扇区号都存在很大的难度。因此，我们需要一个逻辑的、安全的接口来保护用户的数据，让用户通过文件名访问，而非一系列的数字编号。开发磁盘**设备驱动程序**例程和**文件管理**程序，就是为保证所有对磁盘的访问都受到严格管制，且均由受信的例程来执行。结合载入程序，我们看到了操作系统的诞生。

大型机是一种集中化且昂贵的资源，因而我们需要更有效地利用它，时过境迁，这个目标已经随着廉价的桌面PC的到来而逐渐淡出。但是，让CPU等待慢速的数据输入显然不可接受，尤其是在机器耗资数百万美元的情况下。用以缓和这些缺点的方法是**多重处理**（multiprocessing）和IO **SPOOL**ing[⊖]（Simultaneous Peripheral Operation OnLine，外部设备联机并行操作）。前者允许多个程序同时运行，在等待数据时暂停下来，得到数据后恢复执行。后者通过将输出数据缓冲到磁盘上，将慢速的输出操作与快速的中央处理器分离，避免阻塞主CPU的执行。输入数据也有类似的技术。由于许多程序竞争使用CPU的处理时间，因而必须自动采用**调度**措施。人类操作员已经不再能够决定接下来应该运行哪个程序，因为这种切换每秒钟可能会发生上百次。

显然，设计、编写、测试和调度需要完成这些功能的代码并非易事。另外，计算机用户不断增长，他们已不再想花时间去了解软硬件编程的更多技术细节。他们有许多其他重要的事情要处理。因而，操作系统领域成为一个极为专业化的技能，只有少数的程序员参与其中。

17.2 Unix：操作系统的里程碑

尽管Unix（20世纪60年代晚期，由Ken Thomson和Dennis Ritchie在AT&T贝尔实验室开发）肯定不是第一个操作系统，但它的诞生标志着计算历史上的一个重要时刻。由于在设计之初就考虑到可移植性，因此Unix可以容易地重新编译以适应新的平台架构，这使得Unix成为开发者首选的操作系统。所需的不过是一个C编译器和许多热心的支持者。因此，大学和研究组织更愿意将Unix移植到新的平台，而非从头编写全新的操作系统。流行的DEC VAX和IBM机器都装备不同版本的Unix。随**Linux**的到来，人们对Unix的兴趣又被重新点燃。Linux拥有新版本的Unix内核，它是由Linus Torvalds在20世纪90年代早期重写完成的。Linux完全免费，且可以运行在奔腾PC的标准硬件平台上。在早期的i80486 CPU上，Linux也可以运行得十分流畅！Linux开发中最显著的特征是，Internet在想法的快速传播和缺陷修复中起到巨大的作用。在最初的怀疑（甚至嘲弄）之后，现在的制造商开始意识到提供预装Linux的产品的价值。所有这些又进一步扩展了Unix传统及编程工具的发展。

- 从一开始，Unix就是一种适合于程序员的环境。它就如同一个工作台，提供众多的工具，执行一项任务只需按照正确的顺序对这些工具进行组织。表17-1列出Unix提供的一些工具程序，同时给出它们的应用。

表17-1 一些Unix工具程序

工具程序	功 能	示 例
awk	文本处理语言	`cat file | awk '$0 !~ /^$/ {print}'`
cat	打开和拼接文件	`cat header ch_01 | groff -petfH > /tmp/tmp.ps`
diff	文件比较	`diff ch_01.a ch_01.b`
echo	将参数输出到标准输出	`echo $PATH`
find	文件查找实用工具	`find ~ -name"* rob*"-print`
grep	字符串（正则表达式）查找	`grep "rob" /etc/passwd`
lpr	打印守护程序	`lpr -Pnts -#25 ~/Sheets/unix_intro`
ls	目录清单	`ls -al`
more	文件查看	`more book.txt`
ps	进程清单	`ps -af`
sed	流编辑器	`sed 's/r-williams/rob.williams/g'<file1 >file2`

⊖ Spooling也称为“假脱机”，是关于慢速字符设备如何与计算机主机交换信息的一种技术。——译者注

（续）

工具程序	功　能	示　例
sort	字符串排序程序	cat file | sort -f -d
spell	拼写检查程序	spell letter.txt
tr	转换字符串	tr -cs A-Za-z '\012'
troff	文本格式编排程序	groff -pet13
uniq	重复行检测程序	sort | uniq -c | sort -n
users	联机用户列表	users > checkfile
wc	文件大小	wc -w assignment.txt
who	本地用户列表	who

Ken Thompson和Dennis Ritchie在使用C语言开发Unix的过程中，没有计算机可使用。最终还是邻近部门的经理批准他们使用一台不那么让人满意的DEC PDP7，如图17-3所示，如果没有他的慷慨和信任，整个Unix的开发工作可能就会半途而废。初始版本的简洁性、贝尔实验室高质量的工作，以及后来由伯克利大学Bill Joy、Sun微系统和GNU的Richard Stallman做出的修订和功能增强，使得Unix始终在许多方面领先竞争对手。另一方面，Microsoft诞生，在它的平台上存在大量的商业应用，远远超过Unix平台上的数量。从Unix方图获得市场广泛接受的历史中应该汲取的教训是，标准的差异性对Unix的推广造成很大的损害。产品间细小的不兼容性对用户会造成很大的妨碍。由于计算机的大型提供商之间在标准上缺乏协商，Unix受到很大伤害，或许免费软件Linux的发展会将它们抛在脑后，从而最终解决这类商业争论。

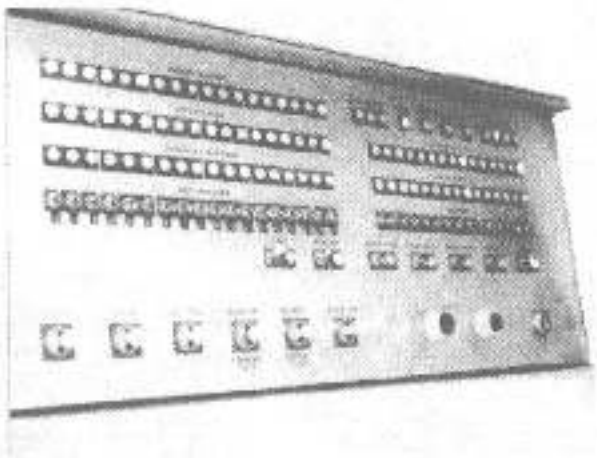


图17-3　PDP7的前面板

Unix是一个多用户、多任务、支持网络、支持多处理器和虚拟内存的操作系统。这些复杂的单词都传递着十分重要的信息，要细心解读。我们会一个一个地来认识它们。多用户表示几个不同的用户可以同时登录、共享资源。不是所有的操作系统都能够做到这一点——MS-DOS和Windows就不能。即使对于个人工作站，这依旧是一项十分有用的功能，它允许几个用户通过网络端口远程地获得对该工作站的访问，不需一定坐在屏幕前。这种情况下，必须采用好的密码方案，以确保用户的进程和文件都足够安全。多任务指的是操作系统的调度动作。多任务操作系统会快速地在进程间跳转，在跳到下一进程前，给予当前进程一个CPU时间片段。支持网络表示可以容易地建立到局域网的连接，进而能够访问Internet。多年来，Unix一直将TCP/IP软件作为标准配置提供。支持多处理器指Unix能够在多CPU平台上协同运行，只有少数几个操作系统能够成功做到，Unix就是其中之一。虚拟内存指允许物理内存（DRAM）溢出到更大的磁盘空间。

编写正规的文档是Unix最初的应用领域之一，它对本书具有直接的重要性，因为本书就是使用一些文本生成工具写成。其他的一些工具会在后面继续出现，它们包括：emacs、pic、eqn、tbl、groff、ghostview、gs和xv，本书的原始文字和原型版本就是在Sun Ultra 5工作站上准备并编排成PostScript。为了传输给出版商，我还编译了一个PDF版本。

和Windows 2000相比，Unix现在被认为是相对较小且高效的操作系统。在MS-DOS时代，情况并非如此，当时人们不接受Unix是因为它太大（需要20 MB的磁盘空间和100 KB的内存）。



图17-4 典型的操作系统分层结构

17.3 概要结构：模块化

操作系统软件一般分成准独立的模块，以利于开发和维护活动的进行。人们常常将这些模块表示成一系列的同心圆或层，如图17-4所示。通过这些图示，读者能够看到操作系统设计的体系结构。采用这种方式，个别模块可以单独升级而无须重新载入整个系统。有些操作系统不用重新启动就可以做到，其他一些操作系统则需要硬件重新启动。

17.4 进程管理：初始化和调度

所有操作系统都提供一系列的公共函数，尽管根据面向应用领域的不同，其着重点也有所不同。实时操作系统，比如Microware的OS-9，对进程间安全和磁盘文件系统的支持比较简单，而支持公司数据处理的系统，比如Windows，则拥有许多精致的安全特性和较强的进程间隔离。

进程以所执行的程序为特征。此处需要提及的是，程序文件含有可执行代码，但并不含有所有数据变量所需的空间。相反，数据所需的内存空间由编译器做出评估。之后，在程序载入到内存后，或是子例程在程序运行期间被调用时，才分配实际所需的空间。进程装入准备运行之前，程序首先得向操作系统注册。这个过程需要创建任务控制块。在操作系统决定接手用户进程的控制权之前，它需要一些至关重要的进程信息：代码在内存中的位置、堆栈指针的值、用户ID、访问权限、优先级别等等。这类数据称为**易失性环境**，它保存在**任务控制块**（Task Control Block，TCB）或**进程控制块**（Process Control Block）中。图17-5对这些内容做了汇总，实际存储的信息可能会更多。TCB保存所有操作系统运行进程所需的信息。信号量（Semaphore）和信号ID（Signal ID）字段仅当进程被阻塞，等待信号量，或者有输入信号在队列中需要处理时才会用到。在一些系统上，TCB中保留空间供存储CPU寄存器和其他一些构成进程易失性环境的值。这些值也可以保存在堆栈上供以后访问。TCB在进程创建时由操作系统负责初始化，操作系统使用TCB管理所有的进程，保持对整个系统的控制。

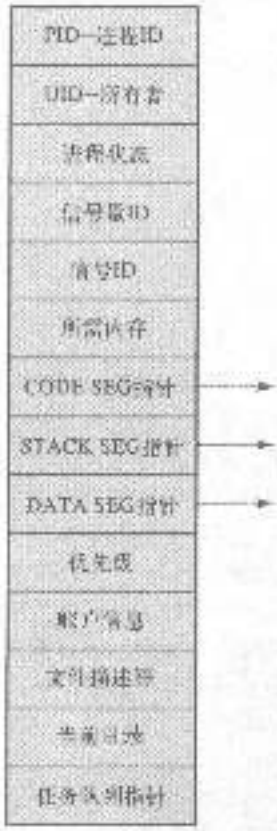


图17-5 TCB的内容汇总

读者可能已经注意到，图17-6中任务2和3共享同一代码块。仅当用来创建代码的编译器生成没有修改过的纯代码时这种情况才有可能实现。我们可以通过严格限制相对地址引用的使用，以及将数据与指令完全分离来做到这一点。现代编译器和CPU在设计之始，就考虑到将内存分段，避免重复载入相同的代码，节省内存空间。在8.6节中，我们已经看到当前函数如何在堆栈上分

配局部变量来使用。通过将数据存放在堆栈上，可以将这部分数据与执行代码分离开来，堆栈数据的访问使用ESP和EBP指针。在这样的系统中，多个用户可以共享同一段程序代码，甚至于用户可能根本不会察觉。在大型的多用户Unix计算机上，许多用户可能会同时运行编辑器代码的同一份副本，而各自的文本缓冲区则保留在独立的堆栈上。为了进一步降低载入内存中的代码的数量，许多系统提供共享内存驻留了例程库的机制。这些例程并非由链接器链接到用户的程序中，也就是说，用户的程序中并没有例程的私有副本。相反，我们让链接器知道存在可供使用的内存驻留代码模块。这就是动态链接库（Dynamic Link Libraries，DLL），它们在载入时或在运行期间链接到应用程序的代码中，链接器可以使用占位库例程（它的功能就是在运行期间访问恰当的DLL）来满足外部引用。通过这种方式，即使你和同事运行的不是相同的应用程序，只要它们调用了同样的DLL子例程，你们就能够共享代码。

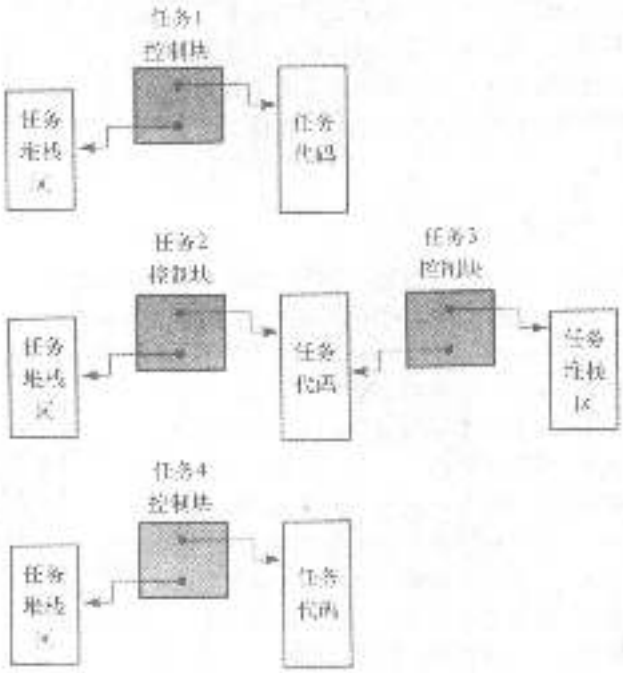


图17-6　任务控制块

操作系统常常会将所有当前活动任务的TCB组织成链表或队列。图17-7阐明了这种机制。尽管图中只给出一个队列，但实际上维护多任务环境需要几个队列。队列记录每个任务的完成状态：执行中、阻塞、等待或就绪。如果系统中只有单个CPU，则同一时刻只能有一个任务在运行。就绪队列保存只要被调度程序选定就能够立即运行的任务。阻塞的任务一般是在等待输入通道数据到来，或是等待输出通道数据发送完成。当然，其他一些原因也会导致阻塞。等待信号量操作的任务保存在单独的队列中，所处的状态为等待。以类似的方式，任务可能会故意地执行暂停指令，强制自己进入另外的等待队列中，直到信号到来为止。在分配的时间片中，任务可以全部执行。这种情况下，当时间片用完时，操作系统就必须强进行干预，将任务切换出去。进程执行完毕，或终止退出时，操作系统就会将该进程从操作系统表中移走。如果父进程不能捕获这种事件，则子进程会进入僵持（Zombie）状态。

使用状态图可以清晰地表示出进程的当前状态（见图17-8）。执行状态比较特殊，因为任何时候（单CPU的计算机）只能有一个进程处于执行状态，其他状态则可以同时为多个进程所有。交互式进程，比如编辑器，大部分时间处于阻塞状态，因为需要等待用户输入数据。

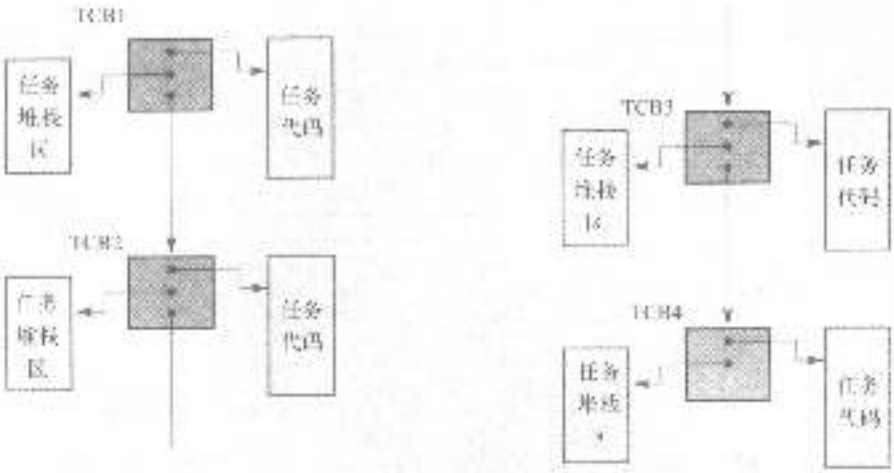


图17-7 任务控制块队列

大型的分时共享Unix系统在任何时候都可能有数以百计的进程在运行，有些进程是由用户运行的，但许多由系统拥有，执行至关重要的内务（housekeeping duty）。此时，在编辑这份文稿的Sun多处理器计算机（Solaris 5.5）上，全部进程的清单中有多达450个进程，参见图17-9。图17-10列出了我的Sun工作站中进程清单的缩略版（有55个活动进程）。

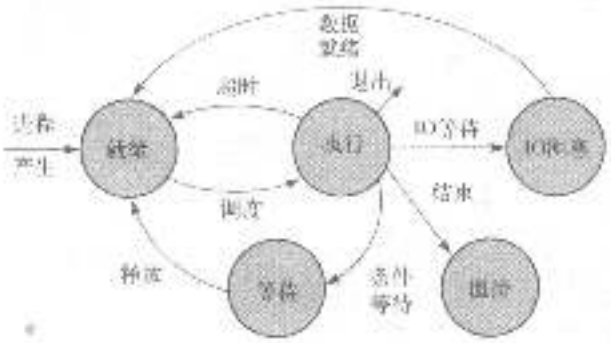


图17-8 任务生命周期的状态图

```
rob@olveston [100] rlogin milly
  Last login: Wed Mar 29 19:22:35 from
rob@milly [41]
rob@milly [41] ps -A | wc -l
                450
rob@milly [42]
rob@milly [42] logout
  Connection closed.
rob@olveston [101]
rob@olveston [101] ps -A | wc -l
                55
rob@olveston [102]
```

图17-9 用Unix ps命令统计运行中的任务

我们可以使用ps实用程序列出Unix计算机上运行的进程或任务。如图17-10所示，它给出所有与活动进程相关的信息。每一栏信息的含义可以参见Unix在线手册，为了方便起见，表17-2给出了一些简短的介绍。

表17-2 ps中显示的信息（取自ps的man页面）

UID	(-f)	进程具有者用户ID（使用-l选项打印登录名）
PID	(all)	进程标识符ID（在杀死进程时会生到这个数字）
PPID	(-f)	父进程的进程ID
STIME	(-f)	进程的起始时间，以小时、分、秒（如果进程在ps查询24小时之前产生，则显示日期）
TTY	(all)	进程的控制终端（？表示没有控制终端，如后台或守护进程）
TIME	(all)	进程的累计执行时间
CMD	(all)	命令名（使用-f选项可得到完整的命令名及其参数，最长80个字符）
S	(l)	进程的状态（使用-l选项） O Process：正在处理器上运行

（续）

		S Sleeping：进程等待事件完成 R Runnable：进程在运行队列中 Z Zombie：进程结束，但父进程未等待 T：进程被中止，或者通过任务控制信号，或者因为跟踪调试结束
C	(f,l)	处理器的利用率。使用-c选项时不打印出来

```
rob00lveston [141] ps -AF
   UID          PID    PPID    C    STIME    TTY        TIME    CMD
   root           0       0    0   Mar 16    ?          0:01    sched
   root           1       0    0   Mar 16    ?          0:02    /etc/init -
   root           2       0    0   Mar 16    ?          0:00    pageout
   root           3       0    0   Mar 16    ?          3:39    fsflush
   root         322     297    0   Mar 16    ?        145:37    /usr/openwin/bin/Xsun :0 -noban
   root         122       1    0   Mar 16    ?          0:00    /usr/sbin/inetd -s
   root         318       1    0   Mar 16    ?          0:00    /usr/lib/saf/sac -t 300
   root         102       1    0   Mar 16    ?          0:00    /usr/sbin/rpcbind
   root         112       1    0   Mar 16    ?          0:00    /usr/sbin/kerbd
   root         110       1    0   Mar 16    ?          0:00    /usr/lib/netsvc/yp/ypbind
   root         284       1    0   Mar 16    ?          0:02    /usr/sbin/vold
   root         226       1    0   Mar 16    ?          0:00    /usr/lib/autofs/automountd
   root         240       1    0   Mar 16    ?          0:01    /usr/sbin/cron
   root         230       1    0   Mar 16    ?          0:00    /usr/sbin/syslogd
   root         249       1    0   Mar 16    ?          0:01    /usr/sbin/nscd
   root         259       1    0   Mar 16    ?          0:01    /usr/lib/lpsched
   root         319       1    0   Mar 16    console    0:00    /usr/lib/saf/ttymon -g -h -p olves
   root         274       1    0   Mar 16    ?          0:00    /usr/lib/utmpd
   root         292       1    0   Mar 16    ?          0:00    /usr/lib/sendmail -q15m
   root         321     318    0   Mar 16    ?          0:00    /usr/lib/saf/ttymon
   rwilliam     340     323    0   Mar 16    ?          0:00    /bin/ksh /usr/dt/config/Xsession
   root         325       1    0   Mar 16    ?          0:00    /usr/openwin/bin/fbconsole -d :0
   rwilliam     408     407    0   Mar 16    ?          0:00    olwmslave
   rwilliam     342     340    0   Mar 16    ?          0:00    /bin/ksh /usr/dt/bin/Xsession
   rwilliam     388     378    0   Mar 16    ?          0:00    /bin/ksh /usr/dt/config/Xsession
   rwilliam     412       1    0   Mar 16    ??         0:00    /usr/openwin/bin/cmdtool -Wp 0 0
   rwilliam     378     342    0   Mar 16    ?          0:00    /bin/tcsh -c unsetenv _ PWD;
   rwilliam     389     388    0   Mar 16    ?          0:00    /bin/ksh /home/staff/csm/csstaff/
   rwilliam     407     389    0   Mar 16    ?          0:18    olwm -syncpid 406
   rwilliam   19576     407    0   10:07:01  ??         0:02    /usr/openwin/bin/xterm
   rwilliam     415     412    0   Mar 16    pts/3      0:00    /bin/tcsh
   rwilliam   19949   19577    0   14:04:16  pts/5      0:05    ghostview /tmp/tmp.ps
   rwilliam     469     407    0   Mar 16    ?         10:35    /usr/local/bin/emacs
   rwilliam    1061    1050    0   Mar 16    ?          0:00    (dns helper)
   rwilliam     553     407    0   Mar 16    ?          0:21    /usr/openwin/bin/filemgr
   rwilliam   11510       1    0   Mar 22    ?          4:08    /opt/simeon/bin/simeon.orig -u
   root       20818   19577    1   18:01:46  pts/5      0:00    ps -Af
   rwilliam   20304   19949    0   15:31:45  pts/5      0:06    gs -sDEVICE=x11 -dNOPAUSE -dQUIET
   rwilliam    1050     407    0   Mar 16    ?         14:39    /usr/local/netscape/netscape
   rwilliam   12251       1    0   Mar 22    ?          0:17    /usr/local/Acrobat4/Reader/sparcs
   rwllliam   19577   19576    0   10:07:02  pts/5      0:01    tcsh
```

sched	OS调度程序；注意它的PID值，启动后的重要进程。
init	引导时的启动进程；其他所有Unix进程均由它启动。
pageout	虚拟内存页处理程序。
fsflush	更新超级块，将数据写入磁盘。
Xsun	X窗口服务程序。
inetd	网络服务器守护进程；提供远程服务，如ftp、telnet、rlogin、talk。
sac	端口服务访问控制程序。
rpcbind	远程过程调用的地址映射程序。

图17-10　使用ps显示Unix任务清单（各种任务在图后列出）

kerbd	kerberos密钥的来源；用于网络用户鉴别。
ypbind	NIS分布式密码系统。
vold	针对CDROM和软盘驱动程序的文件系统（卷）管理程序。
automountd	处理远程文件系统挂接/卸载请求的守护进程。
cron	让任务在特定的时间运行。
syslogd	系统消息处理和路由。
nscd	域名服务的高速缓存守护进程。
lpsched	打印机服务调度程序。
ttymon	终端端口的监控器。
utmpd	用户监控守护进程。
sendmail	Internet邮件服务器。
ttymon	监控终端端口的活动。
ksh	Korn外壳。
fbconsole	控制台窗口。
olmslave	Open Look X窗口管理器。
ksh	第二个Korn外壳。
ksh	第三个Korn外壳。
cmdtool	命令工具窗口处理程序。
tcsh	tenex外壳。
ksh	第四个Korn外壳。
olwm	Open Look X窗口管理器。
xterm	X终端窗口。
tcsh	tenex外壳。
ghostview	PostScript屏幕查看器。
emacs	最好的编辑器！
dns	Internet域名到IP编号转换的域名服务。
filemgr	拖放式文件管理器，用于软盘。
simeon	邮件客户端。
ps	它产生了进程清单！
gs	ghostscript转换器。
netscape	Netscape Navigator Web浏览器。
acroread	用于查看PDF文档的Adobe Acrobat阅读器。
tcsh	另一个用户接口外壳。

图17-10 （续）

17.5 调度决策：时间片划分、抢先和协作

大型的分时共享Unix系统在任何时候都可能有数以百计的进程在运行。操作系统**调度程序**（scheduler）不断地评估进程的相对重要性，实际上，将进程载入到CPU中执行的是**分派程序**（dispatcher）。这个过程根据TCB恢复易失性环境，并将所有的CPU寄存器复原成进程被交换出去时的值。

几乎所有的操作系统调度程序都能通过设置硬件，使之在预定的时间段后中断当前运行的进程，将下一个进程从就绪队列切换进来。不要觉得10ms的**时间片**（time-slice）太短，在这段时间内，处理器可以执行几十万条指令，甚至更多。这种安排能够有效地利用处理资源，并且相当公平，每个进程最终都会得以执行。但这种做法也存在缺点，因为进程总有个轻重缓急，因此简单的“先到先服务”的调度方式，可能不足以满足我们的需要。考虑到这一点，Unix根据进程的优先级对就绪队列进行排队，并实现一种“老化”技术，即进程的基本优先级随进程在就绪队列中保持时间的增长而递增。这种方法力图以更合理的方式来处理调度问题。另一种常用在实时系统中的方案是**抢先式**

(preemptive)。基于时间分片的方法也能够打断（或称抢占）进程的运行。但在此，"抢先式"一词专门指中断驱动的IO操作。通过采用这种方式，操作系统可以响应环境的各种需要，对变化的响应更好。在一些较老的系统中，另外一种调度策略，即短任务优先，也常常得到采用。但是，这种方法需要有精确的剖析数据才能使进程高效地工作，如果负载发生变化，它们的效率将会下降。

比较调度策略时，需要测量和比较一些性能上的数据，遗憾的是，没有单个参数能够胜任。系统管理员希望看到CPU的利用率越高越好。这被看做是系统是否过载，是否需要购买新设备的判断依据。系统吞吐量，或每分钟完成任务数，实际上只对批处理工作环境有效，因为在这种工作环境下，任务的组合相对固定。这些例程的任何变动都会干扰这种测量。类似地，周转时间，即用户等待结果的时间，仅与受计算限制的活动相关。在现实生活中，周转时间更多依赖于人，而非计算机原始的处理能力。记录完成某项任务所浪费的等待时间虽然会有些用处，但这个结果也受到各种各样因素的影响。最终，对于大多数花费大量时间进行编辑或连接到Internet的用户来讲，平均的响应延迟是最基本的参数。交互式计算与批量数据处理具有不同的需求。

17.6　任务通信：管道和重定向

多任务系统发展程序员开发由几个并行运行的任务构成的程序。这种情况下，需要有一种让任务之间能够互通信息但又不互相干扰的机制。如13.3节所述，操作系统可以提供管道（pipe）机制，它能够使不同的任务以类似于读写文件的方式，进行相互之间的数据交换。Unix尤其倡导这种技术（见图17-11），它允许所有的任务通过标准输出（默认为屏幕）或标准输入（默认为键盘）通道向其他任务传递数据。



图17-11　Unix中任务间的数据管道

如前所述，我们可以将管道看做是磁盘的缓冲区（在内存中设立，保存来自于文件的数据块，满足随后的读取请求）。在Unix上，管道既可以由用户在命令行创建，也可以由执行程序使用系统调用创建。由于有了通信管道，设计多任务程序变得更具吸引力，因为任务可以通过"数据通道"机制向其他任务传递数据。

Unix提供专门的重定向机制，它可以接收一个任务的输出，将它以数据流的方式传递给另外的任务进行处理。简单一些的用法是，使用'>'字符将输出重定向到指定名称的文件中，类似地，也可以使用'<'符号将数据从文件定向到任务。

```
rob>  world > hello.txt

rob>  echo "Hello world!" > world.txt

rob>  tr -d "\r" < ch_10.asc > ch_10

rob>  cat header ch_*  > book

rob>  cat > letter << +++
  ?  Dear Craig
  ?  Here is the book that I promised to send
  ?  Rob
  ?  +++
rob>
```

图17-12　Unix中数据的重定向

图17-12给出一些使用'<'和'>'的实例。第一个运行"hello world!"程序，在正常情况下，它会将问候语显示在屏幕上。但是，由于有了'>'符号，这些文字被重定向到指定名称的文件hello.txt中。图17-12中第二个例子运行echo进程输出问候语。第三个例子使用tr实用程序去掉文本文件中由其他字处理程序插入的不需要的回车符号。第四个例子将多个文本文件连接成单个大文件book。最后一个例子有些古怪，它根据键盘的输入生成一个名为letter的文件，就如同在使用一个微型的编辑器。后者称做"立即文档"，表示文档出现的直接性。

为实现数据管道和重定向，Unix规定所有的进程都必须拥有统一的数据接口。每个进程具有三个标准通道：0、1和2，如图17-13所示。它们都以文件描述符的形式实现，其他进程可以接入这些通道，进行数据交换。为了允许这类数据的自由交换得以实现，Unix没有规定数据结构。其他操作系统需要识别各种各样的文件类型及数据记录的格式，相比之下，Unix只是提供一个简单的字节流，所有的数据结构都由应用程序的程序员负责处理。没有这种一致性，数据重定向不可能实现。

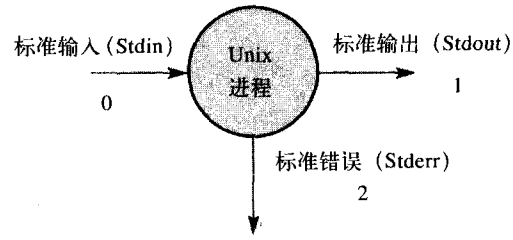


图17-13 Unix进程的标准IO

新进程刚刚创建时，通道0一般连接到控制台键盘，通道1连接到控制台屏幕；通道2也定向到屏幕，但专供输出错误消息使用。常规的Unix术语分别为**stdin 0**、**stdout 1**和**stderr 2**。进程开始时，这三个通道是这样设置的，但是它们均可重定向。这也正是‘|’、‘<’和‘>’操作符的用途所在。标准化的进程接口是Unix最有用的特性之一（其他操作系统不提供这一特性），它允许程序员将现有的程序组合起来，构建更大型的处理流水线。

进程标准IO的重定向是由负责启动进程运行的外壳进程完成的。我们在外壳中输入cat hello.c是为在屏幕上列出hello.c中的代码。但是，使用重定向符cat hello.c > world.c，外壳在接收到指令后会将输出重定向到文件中，或者创建新的文件，或者覆盖现有的world.c。如果的确希望将代码添加到现有的文件上，Unix提供下面的命令：hello.c >> world.c。谨慎的程序员可能会认为，重定向太容易造成灾难，它可以快速永久性地删除有价值的数据。为了避免这种风险，Unix可以拒绝重定向到已经存在的文件。要强行实施这个条件，只需将神秘的set noclobber命令输入到外壳中即可。

17.7 排斥和同步：信号量和信号

互斥（mutual exclusion）问题第一次出现是在9.5节中。当时，如果主线（mainline）进程正在更新数据的过程当中，中断的影响可能会破坏数据结构的一致性。在多任务系统中，这种情况的出现频率会被放大许多倍，因为每次调度程序抢占一个任务，都有可能发生“临界数据被破坏”事件。操作系统需要知道什么时候任务正在处理共享的数据结构，此时切换任务容易将数据破坏。这就是**信号量**（semaphore）的用途。

从数据的角度，信号量就如同控制任务访问数据的软件标志（见图17-14）。它们的操作很直接。当任务需要控制对独占资源（同一时间只能由一个任务使用）的访问时，它必须调用特殊的操作系统函数，请求信号量的分配。这个请求会被记录下来，从此时起，任何任务在访问临界数据区之前，都必须调用另一个操作系统函数WAIT(sem)，检查指定的信号量标志。在放弃对该资源的独占访问时，必须相应地调用SIGNAL(sem)。这样，围绕临界区域的WAIT和SIGNAL调用，就如同体育场的十字转门一样，控制对数据的访问。

```
WAIT (sem_buff)
      . . . .
      //critical region code
      . . . .
SIGNAL (sem_buff);
```

图17-14 信号量操作符的放置：WAIT和SIGNAL

信号量的起源要追溯到20世纪60年代Dijkstra的著作。有时候，等待和发信号分别用P和V来表示，这来源于荷兰语，意思是“等待和递增”。现在，使用信号量功能的最普遍做法是，将临界数据隐藏在对象中，仅能通过可信任的方法对它们进行检查或更新。之后，我们将WAIT和SIGNAL函数插入到这些访问函数中，控制对数据的访问。信号量编程一般比较困难，这要归因于必须保证WAIT()调用和SIGNAL()调用成对出现。至目前为止，信号量好像仅仅充当“忙标志”的角色，在资源可用时上升，在资源被占用时下降。如果被阻塞的程序不断地循环查询，等待标志升起，则会导致CPU处理时间的极大浪费。每个信号量还包括一个任务队列，操作系统调度程序使用它来暂时存放被阻塞的任务，相关的信号量表示资源再度空闲后，操作系统会将下一个任务从信号量队列中

取出执行。为此，我们可以将信号量看做是操作系统提供的调度机制的一个组成部分，而非仅仅是忙标志。图17-15说明了互锁信号量的使用。此处并非一定要使用三个信号量，但两个是必需的。两个信号量的必要性可以通过只使用单个sem_free信号量会导致死锁的情况加以说明。如果数据缓冲区为空且未被锁用，READER任务就会被接受，锁定对缓冲区的访问，阻止WRITER在其中存储数据项。类似地，如果数据缓冲区满且未被锁用，WRITER则会请求该资源，并将所有的READER锁在外面。双互锁信号量方案能够阻止这类死锁事件发生。

```
WRITER Task
 produce a new item for Consumer Task
  WAIT(sem_space)
   WAIT(sem_free)
    place item on buffer queue
   SIGNAL(sem_free)
  SIGNAL(sem_data)
 END_WRITER


READER Task
 WAIT(sem_data)
  WAIT(sem_free)
   take item from buffer queue
  SIGNAL(sem_free)
 SIGNAL(sem_space)
process the item
END_READER
```

图17-15　保护循环数据缓冲区的信号量

图17-15中使用了伪代码，部分是因为Unix对信号量的实现十分概括、灵活且易于混淆。查看semop的man页面，可以了解具体的情况，看看你是否对这种说法持有异议。

总体来说，相比于信号量，Unix程序员好像更倾于使用信号来同步任务。操作系统以类似于中断的方式，使用**信号**（signal）促使进程执行专门指定的函数。信号并不传送许多数据，一般只以整数的形式提供基本的自身标识信息。如果程序员希望在进程中接收信号，则必须安装信号处理函数。图17-16中的例子显示了信号处理器作为没有直接软件调用的函数。从这个角度看，信号很像中断，每个中断源都有一个类似于信号处理器的ISR（Interrupt Service Routine，中断服务例程）。另一种观点是将信号处理器代码看做是一个线程，该线程仅当操作系统将正确的信号发送给进程时，才会得以执行。

```
/* kbdcnt */
#include <stdio.h>
#include <signal.h>
#define MYSIG 44

/* Signal handler function, evoked by sig 44
   reinstalls after each sig hit, prints number of hits
*/
void getsig(int s)
{
static int count = 0;
printf("signal %d again, %dth time \n", s, ++count);
signal(MYSIG, getsig);
}

/* Process to demonstrate signals, sets up a sig handler to
   count the number of sig hits received. Loops forever.
   Start using "kbdcnt &" and make note of pid value returned
   Recommend to use "kill -44 pid" to send signals
   Remove using "kill -9 pid"
*/
int main(void)
{
signal(MYSIG, getsig);
printf("start counting kbd kills\n");
  while(1) {};
return 0;
}
```

图17-16　安装Unix信号处理器统计和显示信号发生次数

信号的动作可以用图17-16列出的代码来演示。此处，程序安装一个信号处理函数getsig()，并使用编号为44的信号来调用它。在Unix系统上选择信号编号时，最好先检查一下它们的用法。查看/usr/include/sys/signal.h，从中我们可以得到信号的清单，以及允许的最大值（定义为NSIG）。

信号捕获函数的安装是使用系统调用signal()完成的。要注意，每次信号事件发生后都得重新安装，因为系统会将其重置为默认处理器。信号处理器接收输入信号的编号作为参数，函数使用这个编号确定采取什么动作。因此，用单个函数捕获所有的输入信号是完全可行的。我们可以借用外壳级的命令kill向进程发送信号。尽管这个命令一般用来删除出错的进程，但它也可以发送非决定性的通知信号！编号为9的信号因为其致命的效果而闻名。但此处我们使用编号为44的信号，这个信号通常没有什么其他用途。图17-17给出了完整的编译、运行和测试过程。

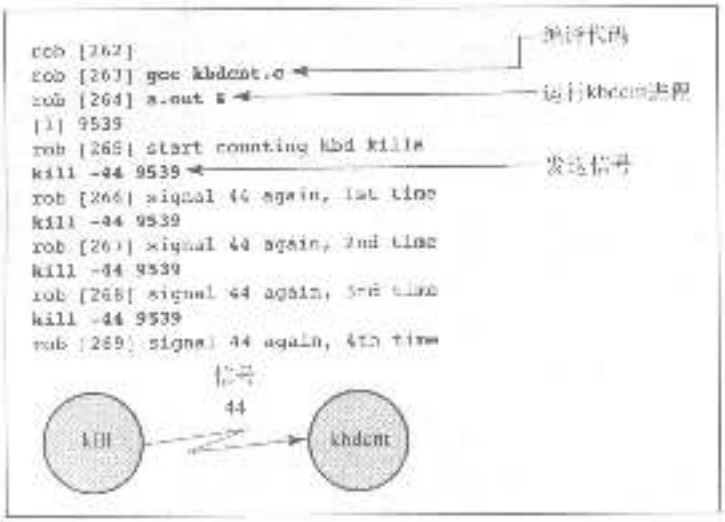


图17-17 使用信号通知Unix进程

需要注意的是，在命令后加上一个'&'（a.out &），可以使进程与键盘分离，转入后台运行。这个例子中，这种技术并不特别重要（因为我们可以使用另外一个窗口执行kill -44），但有时它是一项必需的功能。

图17-18列出的程序演示两个进程通过传递信号进行通信。它还演示了Unix如何使用fork命令创建新进程，将单个进程一分为二，这将会复制进程的环境和数据段，同时保持单一代码段。所产生的两个进程，即父进程和子进程，都运行同一份代码，根据它们的PID值来区分（见图17-19）。因此，在fork指令结束后，将会存在两个近乎相同的进程，两个进程均执行fork指令后的相同代码。它们可以作为克隆进程继续运行，或者使用它们不同的PID运行分化，改变各自的执行路线。在这个例子中，cpid变量用来将执行路径切换到IF和ELSE语句的两个部分，IF由子进程执行，而ELSE由父进程采用。这样启动新进程的方式乍看起来好像有些奇怪，但在需要进程间通信或同步运行时，它确实拥有一些优点。图17-20给出了fork活动的示意图。

使用fork()复制进程，实际上并非载入新的程序，它只是给予程序员将一个进程分为两个的机会。每个进程都执行同一份代码，但可以使用父进程和子进程之间PID值的不同来切换执行路线，执行代码的不同部分。如果需要运行完全不同的进程，Unix提供了exec()系统调用系列，它允许派生的子进程从磁盘载入完全不同的程序来执行。通过这种方式，进程可以在内存中产生，并通过互相发送信号来通信。

在使用信号之前，有一些问题需要注意。信号不会排队，操作系统至多为每个进程提供单个信号待决标志集。这意味着如果进程正在忙于执行某个信号处理器，后续针对该处理器的信号会被压套，等待得到执行的机会。如果具有同一类型的信号到达，则会被忽略。也许更需要考虑的是信号

和一些系统调用之间的相互作用。如果信号到达时，进程正在执行慢速的系统调用，比如读键盘，它可能会失败并返回错误码。因此，如果需要的话，检查所有系统调用的返回码，实现某种恢复机制就变得更加重要了。

```
#include <stdio.h>
#include <signal.h>
#include <errno.h>

#define PSIG 43  /* check the value of NSIG in      */
                 /*   /usr/include/sys/signal.h     */
#define CSIG 42  /* before choosing the signal values*/

int ccount = 0;
int pcount = 0;
char str[] = "error message ";

void psigfunc(int s)
{
     pcount++;
     signal(CSIG, psigfunc);
}

void csigfunc(int s)
{
     ccount++;
     signal(PSIG, csigfunc);
}

main()
{
int ke, tpid, ppid, cpid;

     ppid = getpid();
     cpid = fork();  /* spawn child process */
     if (cpid == -1)
     {
          printf("failed to fork\n");
          exit(1);
     }
     if (cpid == 0 )
     {
     /* Child process executes here */
          signal(PSIG, csigfunc);
          printf("Child started\n");
          while (1) {
            pause();
            printf("Child hit! count = %d\n",ccount);
            sleep(rand()%10);
            if( (kill(ppid, CSIG)) < perror(str);
            }
     }
     else
     {
     /* Parent process continues execution from here */
          signal(CSIG, psigfunc);
          printf("Parent started\n");
          while (1) {
            sleep(rand()%10);
            if( (kill(cpid, PSIG)) < perror(str);
            pause();
            printf("Parent hit! count = %d\n",pcount);
          }
     }
}
```

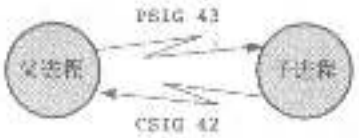


图17-18　Unix进程中信号的使用

在Unix系统中，我们还可以使用文件锁来保护临界资源。文件锁没有任何内容，重要的是它的存在与否！由于文件的创建和删除都以一致的方式实现，不允许中断或进程重新调度，因此可以将文件目录项当做简单的信号量。尽管这种处理独占资源问题的方式谈不上优雅，但它确实能够经受时间的考验。在我的Sun计算机上，Netscape Navigator每次运行时都会创建一个文件锁（~/.netscape/lock），这样做的目的是在程序已经运行的情况下，再次运行该程序时给用户一个警告。

```
rob@olveston [52] a.out
Child started
Parent started
Child hit! count = 1
Parent hit! count = 1
Child hit! count = 2
Parent hit! count = 2
Child hit! count = 3
Parent hit! count = 3
Child hit! count = 4
Parent hit! count = 4
Child hit! count = 5
^C
rob@olveston [53]
```

图17-19 在Unix上运行图17-16的程序

```
rob@olveston [40] trial &
rob@olveston [41] ps
    PID TTY      TIME CMD
    451 pts/4    0:00 tcsh
    469 pts/4    0:02 trial
rob@olveston [42]

rob@olveston [43] ps
    PID TTY      TIME CMD
    451 pts/4    0:00 tcsh
    469 pts/4    0:02 trial
    472 pts/4    0:02 trial
rob@olveston [44]
```

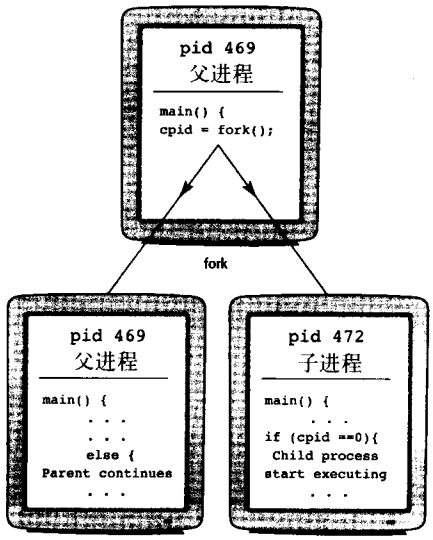


图17-20 使用fork()创建Unix进程

17.8 内存分配：malloc()和free()

由于Unix提供虚拟内存环境，程序员并不直接面对大小的限制。可用的虚拟内存空间，尽管理想状态下为每段4 GB，但实际上要受磁盘上交换文件容量的限制。在安装操作系统时，一般为交换文件分配两倍于主存的空间，并专门为它保留一个磁盘分区。

C程序员一般仅在处理动态数据变量时，才会意识到内存分配问题。这些内存不在编译时声明，也没有名字，程序运行时使用malloc()系统调用分配它们。malloc()取所需的数据区大小为参数。以这种方式保留的数据区不能用名字引用，因为程序员未在（也不可能）编译时给出它们的名字。取而代之的是，操作系统返回新数据区起始地址，程序必须能够处理这种地址。实际上，长久以来这一直是程序员们争议的一个话题，因为这种处理数据内存的方式会导致许多常见的缺陷。当动态分配的数据项不再需要时，程序员要负责将空间释放——需要以初始的指针值为参数调用free()。如果一切执行正确，操作系统会回收该数据区，使它可以为其他用户所用。遗憾的是，由于软件的易变性，这种简单的内务处理很容易被忽略或者完全遗忘。当这种情况发生时，就会有一块交换区域被占用，直到进程正确退出或者系统重启为止。

如12.5节所述，现代的虚拟内存系统中，每次内存访问都需要执行复杂的地址转换（见图17-21）。这涉及使用段基址寄存器将代码段、数据段和堆栈段放在单一的、线性地址空间。每个进程都对应内存中的一个页表，通过它可以将虚地址转换成物理地址值。在转换过程中，我们取虚地址的高位部分作为页编号，取低位部分作为页内偏移。页表可以将页编号转换成物理帧编号。采用这种方式，额外的内存访问（每个读取周期都需要执行）会使得计算机的运行速度变得极为缓慢。因而，计算机内有专门的内存管理硬件来缓存最近的页-帧对，以避免从虚拟到物理的转换所需的额外内存访问。

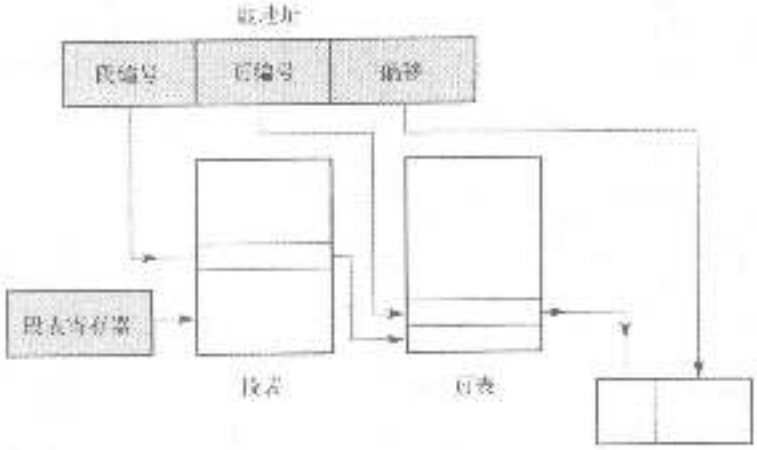


图17-21　虚地址到物理地址的转换

17.9　用户界面：GUI和外壳

用户对操作系统体验的一个重要部分就是命令接口。在Unix上，这是由叫做外壳（shell）的程序完成的。Unix上有好几种不同的外壳程序，有些还被移植到其他的操作系统。表17-3列出了最流行的几种外壳程序。登录后外壳就会运行，它负责解释所有的键盘命令，除非用户启动应用程序并直接与其进行通信。

表17-3　交互式外壳

sh	由Steve Bourne最初编写的外壳
csh	Bill Joy编写的C外壳
bash	Bourne again外壳
tcsh	Tenex外壳，甚比较喜欢
ksh	Korn外壳，十分流行

用户在外壳中输入命令时，比如输入ls、ps或emacs，外壳会派生出另一个外壳进程，由它（子进程）来执行命令程序，父外壳则等待子进程的执行。在命令结束后，子进程会结束，并向正在等待的父外壳返回一个退出值。外壳除作为命令行解释器，交互式地处理用户键入的输入以外，还能够处理命令文件或脚本。实际上，每次外壳启动时，都会查找一个特殊的启动文件来执行。这个启动文件中，可以含有根据任务需要剪裁环境的初始化指令。查看主目录中的.cshrc或类似的文件，我们将会发现其中全是外壳指令。能够执行命令文件的能力使得外壳在系统启动阶段十分有用，因为它们可以控制整个过程。Unix中的外壳还负责管理它们所派生的子进程的标准输入和标准输出的重定向，如图17-6所示。Unix另一项有用的特性是，它能够实现命令置换（command substitution），即将Unix命令的输出定向到外壳变量中。在使用基本的Unix工具编写复杂的处理脚本时，这项功能十分有用。

图17-22中给出几个命令置换的例子。注意，需要求值的命令被反引号（`）包围起来。不要将它们与单引号（'）混淆——单引号用来隐藏信息，不让外壳对其进行解释，也可以解释为内容的"引用"。其中的内容以字面形式作为数据项，不对外壳表示任何特殊的事情。双引号（""）的行为不那么偏激，它们常常用来将字符串拼接在一起。

```
rob [52] grep 'ofthe' `ls | egrep "ch_.."`
  ch_01: symbiosis establishes ofthe hardware and soft
  ch_11:ax little as it is, the start ofthe application
rob [53]
rob [53] mail `cat mail_list` < message
rob [54]
rob [54] echo "There are `who | wc -l` users logged in at `date`"
    there are 13 users logged i  at Tue June 20 20:23:55 BST 2000
rob [55]
```

图17-22　Unix上外壳命令置换的例子

17.10 输入输出管理：设备处理程序

前面曾提到过，操作系统的作用之一，就是隐藏硬件的复杂性和可变性，为程序员提供统一的接口。所有IO管理的核心问题之一，是如何处理CPU和机械式外接设备之间悬殊的速度差异。Unix的体系结构充分考虑到这些实际情况，其设计极为合理，IO子系统被并入文件系统中，使得设备就如同文件系统中的文件。设备分为块设备，如硬盘，或串行设备，如以太网。/dev目录中存储各种设备相关的文件。图17-23给出我使用的Sun工作站中/dev目录中文件的清单。从内核的角度，所有的文件和设备都是等同的。为了做到这一点，设备被表示为“特殊的文件”，它们参加到数据重定向中，和文件没有什么不同。可以通过下面的例子说明这种情况：启动两个xterm窗口，使用ps -al找出的ptty和pts（伪终端），然后使用下面的命令，就可以在它们之间发送文本消息：

```
echo hello other window > /dev/pts/0
```

最后的数字表示将消息发送到哪个X窗口，0，1，2……。

```
rob@olveston [154] ls /dev

arp         icmp        ptmajor ptyq5   ptyrd       syscon      ttypc   ttyr4
audio       ie          ptmx    ptyq6   ptyre       systty      ttypd   ttyr5
audioctl    ip          pts     ptyq7   ptyrf       tcp         ttype   ttyr6
bdoff       ipd         ptyp0   ptyq8   qe          term        ttypf   ttyr7
be          ipdcm       ptyp1   ptyq9   rawip       ticlts      ttyq0   ttyr8
conslog     ipdptp      ptyp2   ptyqa   rdiskette   ticots      ttyq1   ttyr9
console     isdn        ptyp3   ptyqb   rdiskette0  ticotsord   ttyq2   ttyra
cua         kbd         ptyp4   ptyqc   rdsk        tnfctl      ttyq3   ttyrb
diskette    kmem        ptyp5   ptyqd   rfd0        tnfmap      ttyq4   ttyrc
diskette0   kstat       ptyp6   ptyqe   rfd0a       tty         ttyq5   ttyrd
dsk         ksyms       ptyp7   ptyqf   rfd0b       ttya        ttyq6   ttyre
dtremote    le          ptyp8   ptyr0   rfd0c       ttyb        ttyq7   ttyrf
dump        llcl        ptyp9   ptyr1   rmt         ttyp0       ttyq8   udp
ecpp0       log         ptypa   ptyr2   sad         ttyp1       ttyq9   volctl
es          logindmux   ptypb   ptyr3   sehdlc      ttyp2       ttyqa   winlock
fb          m640        ptypc   ptyr4   sehdlc0     ttyp3       ttyqb   wscons
fb0         md          ptypd   ptyr5   sehdlc1     ttyp4       ttyqc   zero
fbs         mem         ptype   ptyr6   sound       ttyp5       ttyqd
fd          mouse       ptypf   ptyr7   sp          ttyp6       ttyqe
fd0         null        ptyq0   ptyr8   spcic       ttyp7       ttyqf
fd0a        openprom    ptyq1   ptyr9   stderr      ttyp8       ttyr0
fd0b        partn       ptyq2   ptyra   stdin       ttyp9       ttyr1
fd0c        printers    ptyq3   ptyrb   stdout      ttypa       ttyr2
hme         profile     ptyq4   ptyrc   swap        ttypb       ttyr3
rob@olveston [155]
```

图17-23 Unix系统中/dev目录下的设备驱动程序

图17-23中列出的每一项都是设备驱动程序或一些相关的资源。设备驱动程序是处理数据IO的一些例程。在Unix中，它们与内核关系最为密切，现在已经能够做到动态安装。换句话说，现在安装驱动程序时，不是必须重新编译操作系统内核，并重启计算机。如果用户只是想换台打印机，却不得不执行这些任务的话，好像有些难以接受，但Unix的支持者却并不在意！/dev目录中已经预先安装了大量的设备驱动程序，因此实际上大多数用户并不需要执行这么复杂的任务。

/dev中的null有些奇怪，它在进程管道中担当“字节桶”的角色。如果只想让大量数据经过进程流水线传递，但并不想保留生成的数据流，则可以将它重定向到字节桶中：> /dev/null。

Unix将设备驱动程序组织成两组。块设备驱动程序存储在一个数组中，字符设备存储在另一个数组中。设备通过它们所属的数组以及设备编号来区别。设备编号可以分成两部分，**主编号**标识设备在设备驱动程序数组中的正确位置，而次编号指定需要访问的具体设备。通过这种方式，类似的设备可以共享一个设备驱动程序，同时依旧让系统可以区分。

回顾图2-6，我们可以记起在用户程序处理之前，对键盘输入进行缓冲的问题。尽管它是由设备

驱动程序完成的，但由于每个学生在学习C语言的第一周就会遇到这个问题，因此还是将它提前讨论。tty设备驱动程序一般会缓冲用户的输入，允许用户进行单行编辑，直到回车键按下为止。在需要时，程序员也可以使用其他模式。早些时候，还没有X窗口系统之前，各种终端类型使程序员应接不暇。为了克服提供编辑功能的控制代码的多样性，Unix建立一个终端特性数据库文件，叫做termcap，现在依旧可以在/etc/termcap中找到这个文件。其中，我们可以看到以标准格式编码的所有的控制功能，包括数十个不同的提供商提供的数以百计的不同终端类型（见图17-24）。使用termcap元控制信息（见表17-4）编写软件，可以快速地将依赖于终端IO的程序从一种VDU移植到另一种。填充字符是必需的，因为终端和CPU相比较慢，甚至网络互连线路的传输速率也快于终端的帧速率。

```
rob@olveston [101] more /etc/termcap
        . . .
xterm|vs100|xterm terminal emulator (X Window System):\
    :AL=\E[%dL:DC=\E[%dP:DL=\E[%dM:DO=\E[%dB:IC=\E[%d@:U
    :al=\E[L:am:\
    :bs:cd=\E[J:ce=\E[K:cl=\E[H\E[2J:cm=\E[%i%d;%dH:co#8
    :cs=\E[%i%d;%dr:ct=\E[3k:\
    :dc=\E[P:dl=\E[M:\
    :im=\E[4h:ei=\E[41:mi:\
    :ho=\E[H:\
    :is=\E[r\E[m\E[2J\E[H\E[?7h\E[?1;3;4;61\E[41:\
    :rs=\E[r\E[m\E[2J\E[H\E[?7h\E[?1;3;4;61\E[41\E<:\
    :kl=\EOP:k2=\EOQ:k3=\EOR:k4=\EOS:kb=^H:kd=\EOB:ke=\E
    :kl=\EOD:km:kn#4:kr=\EOC:ks=\E[?1h\E=:ku=\EOA:\
    :li#65:md=\E[1m:me=\E[m:mr=\E[7m:ms:nd=\E[C:pt:\
    :sc=\E7:rc=\E8:sf=\n:so=\E[7m:se=\E[m:sr=\EM:\
    :te=\E[2J\E[?471\E8:ti=\E7\E[?47h:\
    :up=\E[A:us=\E[4m:ue=\E[m:xn:
```

图17-24　Unix中某xterm的termcap项

表17-4　一些Unix termcap元代码

代码	参数类型	填充	功能
al	str	(P*)	添加空行
am	bool		终端自动控制边距
bs	bool	(o)	终端可以使用^H键退格
cd	str	(P*)	清除全屏
ce	str	(P)	清除到行末
cl	str	(P*)	清除屏幕并重置光标
cm	str	(NP)	将光标移到第m行、第n列
cs	str	(NP)	改变从行m至n的滚动区域
ct	str	(P)	清除所有制表位
dc	str	(P*)	删除字符
dl	str	(P*)	删除行
im	str		进行插入模式
ho	str	(P)	光标复位

%d	从0开始的十进制数	%r	行/列的逆序
%2	同%2d	%i	原点定在1,1，不是0,0
%3	同%3d	%%	输出单个%
%.	与ASCII等价	%n	用0140对行和列执行XOR操作
%+v	相加后作为%	%B	BCD格式
%>xy	如果值大于x，则加y。没有传送	%D	逆向编码

17.11 小结

- 操作系统是大型的程序，常常随计算机硬件一同提供，目的是为用户提供远胜于裸硬件的增强功能。Unix、Linux和Windows NT都是操作系统。
- 宽泛地分，操作系统分为三类：批处理、在线和实时。现代操作系统一般会提供所有这三方面的功能。
- 如果都使用相同的操作系统，那么将应用程序软件从一个平台移植到另一个平台将会比较简单。否则，移植将会变成一场噩梦！
- 操作系统已经由早期程序员之间例程的共享，进化到当前负责分配和保护资源。
- 无论对于个人工作站还是Internet路由器和服务器，Unix都依旧是重要的多任务操作系统。多任务是指不需用户参与，计算机快速地从一个任务切换到另一个任务的能力。
- 对于多任务系统，判断什么时候切换任务有几种不同的策略：时间分片、抢先式或协同式。所有情况下，都需要使用中断将控制权传递给操作系统调度程序。之后，由它来决定接下来运行哪个任务。
- 任务间的通信由操作系统提供的功能支持和控制。应用程序可以使用操作系统提供的信号量、管道和信号。
- 操作系统的重要职责就是保护对独占资源的访问，保证共享数据的任务能够安全地读取数据。信号量和信号是最常用来完成这一目标的方法。
- 信号量是用来在多任务环境中保护临界资源的软件设施。信号量提供“忙标志”、用来测试的函数、清除和设置标志，以及供操作系统存放阻塞任务的任务队列。
- IO操作现在由IO子系统控制，它是操作系统的一部分。将数据传入或传出计算机的操作与任务调度方案密切相关。

实习作业

我们推荐的实习作业包括使用Unix工具，建立进程流水线，并在sh或csh中编写小段的脚本，这样可以逐渐熟悉Unix，它是程序员和管理员的工作平台。

练习

1. 操作系统在个人工作站中起到的作用是什么？
2. 在将程序移植到新平台时，要考虑哪些问题？
3. TCB（Task Control Block，任务控制块）是什么？它存储什么内容？
4. 图17-17第三行中的‘&’符号表示什么含义？下面的Unix管道完成什么工作？

```
cat file | tr '''\n' | sort | uniq -c | sort -n
```

符号‘|’和‘>’的不同之处是什么？

5. 信号处理函数如何与主进程通信？它有可能受到临界区问题的困扰吗？
6. 辨析信号量和信号之间的不同。修改图17-16中给出的代码，使它只统计DEL键按下的次数。DEL发送信号2，这个信号可以捕获和转移。Ctrl+C键有什么特殊之处？
7. 外壳是什么？如果正在使用X窗口系统，是否还需要使用外壳呢？列举出Unix自带的四种外壳。
8. 解释假脱机是什么？为什么要引入它？对于抢先式多任务操作系统，它是否依旧必不可少？
9. 设备驱动程序是什么？是否只有打印机接口才需要？
10. 最有用的Unix工具之一就是find。将名为sardine的文件深深地隐藏到你自己的目录树中，然后使用find ~ -name sardine -print命令找出它来。阅读find的手册页，修改该指令，使之自动删除丢失的文件。
11. 试着运行下面的管道，并找出它的功能：

```
du -a ~ | sort -n | tail
```

现在，找出5个最大的文件。在需要时，可以使用在线手册查找du、sort和tail工具。

课外读物

- Ritchie（1997），不错的简介。
- Silberschatz等著（2000），提供操作系统更详尽的介绍。
- Sobell（2005），最近的介绍Unix和Linux的教材。
- 如果对Unix、X和Internet感到好奇，对于基础技术，Wang（1997）依旧很有价值。
- 阅读一些最有用的Unix命令和工具的man页面：ls、ps、kill、grep、setenv、printenv、lp、whereis、which、who。
- 遗憾的是，Windows的帮助文件对于技术方面的介绍较少，没有提供什么有用的信息，但Microsoft Developers'Network（MSDN）提供了大量的文档。Microsoft Developer Studio内的帮助系统直接与MSDN连接，相当方便。
- Unix和Linux的简短历史，参见：
  http://crackmonkey.org/unix.html
- 大量的问答集：
  http://howto.linuxberg.com/
- Linux标准化：
  http://linuxunited.org/
- 欧洲的Linux源码网站：
  http://src.doc.ic.ac.uk/
- 相关信息：
  http://www.linuxhq.com/
- 商业Linux开发和支持：
  http://www.redhat.com/
- 这些网站可以通过本书的配套网站访问：
  http://www.pearsoned.co.uk/williams

第18章 Windows XP

Windows操作系统从Windows 3.1（它的运行还离不开MS-DOS）开始，随后是Windows 95、98、ME、NT、2000和XP。Windows操作系统提供不同的版本，供服务器、工作站和嵌入式系统使用，为我们提供集成图形用户界面的多任务环境。

18.1 Windows GUI：满足用户的需求

最初，微软在MS-DOS的基础上开发Windows操作系统，以满足用户对图形用户界面的需求。最初产品的最后一版是Windows 3.11。Windows 3.11以后，就可以不用再去使用DOS的命令行界面，但磁盘处理例程和文件管理器都保留下来。最初，Apple采取的法律诉讼限制窗口只能在屏幕上以平铺的方式出现，互相之间不能重叠。同时，系统中不能使用垃圾筒图标，因为Apple认为自己拥有该图例的版权。后来，微软与Apple签订了一项许可协议，这样才能在系统中使用垃圾筒图标，将窗口重叠排放。Windows 95/98已经不再需要MS-DOS即可独立运行。微软和IBM曾协同开发OS/2多任务操作系统。这个操作系统面向PC，并提供WIMP界面。在1988年，微软和IBM之间的伙伴关系解散，微软开始开发Windows NT，这是微软操作系统开发中的一大进步。微软从DEC挖来David Cutler加入Windows NT的开发，David Cutler在DEC负责业界著名的VMS操作系统（阴谋家会看出“IBM”这三个ASCII字符可以转换成“HAL”——只需将每个字符都减1。但是如果将“VMS”以同样的方式加1，则会得到“WNT”。这种助记符运算，使我回忆起小时候试着帮助妈妈做填字游戏时所感受到的望尘莫及）。

Windows NT完全从头开发，不是改写现有的Windows 3.x或Windows 95。它的目标是同时能够运行MS-DOS、OS/2、Windows 3.1/16位和POSIX兼容的代码。它是完全32位的操作系统，提供进程内的多线程能力、网络功能、抢先式多任务，以及时间片调度和更强的文件安全性。所定位的市场是那些要求能够灵活控制文件共享访问，同时降低日益增长的网络管理费用的公司网络。微软声称Windows NT由250个程序员耗时5年开发而成，共计1千4百万行代码；和Linux的起源大不相同。

和Unix一样，Windows NT不让用户程序直接访问IO端口，因为这将会带来安全上的风险。相反，所有的输入-输出操作都必须交由操作系统来处理，操作系统使用自己安全的设备驱动程序执行必需的数据传输工作。这不同于相对宽松的MS-DOS机制——Windows 95和98也继承了MS-DOS这一特点，如果扩展卡提供商不了解编写在Windows NT上运行的新设备驱动程序的复杂性，则会面临较大的困难。操作系统的安全问题是Windows NT设计的核心，部分是因为这一领域被认为是Unix的一个弱点，部分是为了满足银行和保险公司对安全的在线计算机系统日益增长的要求。此外，全球市场需要一种简单的本地化方案，将提示符和菜单转换成相应的本地语言。微软采用16位的Unicode字符集来取代有限的8位ASCII字符集，克服了这个难题。

Microsoft还要求Windows NT能够快速地移植到其他处理器构架。在开始时，需要支持的基本CPU是Intel奔腾处理器、DEC Alpha、IBM PowerPC和MIPS RISC。硬件无关性的宏伟计划没有成功，只有奔腾依旧支持Windows NT。尽管如此，操作系统中需要与硬件交换数据的大部分代码都面向**HAL**（Hardware Abstraction Layer，硬件抽象层）——硬件和操作系统之间的接口或缓冲层。这种方案的确使得向新硬件的移植工作更容易。

NT的软件架构分为用户部分和内核部分。执行程序运行在内核模式，提供低级的服务，包括进程时间分配、内存分配、中断和IO管理。所有处于内核模式的软件都拥有较高的优先级，可以使用机器指令集中的所有指令以及访问所有的内存单元。通过将基本的功能划分成独立的模块，微软

能够提供不同版本的Windows NT/XP，服务于多种用途。这也是不同版本的Server和Workstation共享同一资源代码模块的方式。

许多功能模块都以DLL的形式实现。在运行时与其他部分完成链接。这样，尽管从用户的角度看操作系统是铁板一块，但实际上，它是由许多不同的半独立单元组成。有意思的是，执行程序还含有一个对象管理器。它支持Windows NT中面向对象方案的软件实现。NT内核定义了一系列由内核自身、执行程序和子系统使用的内建集（对象类型）。Windows NT及其应用程序都是内核层面的可信任对象，以超级用户模式运行。

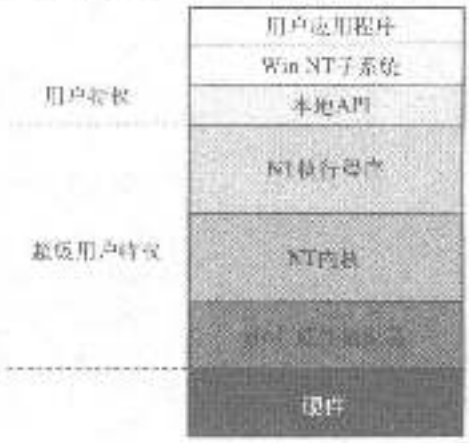


图18-1　Windows NT/XP的结构

从图18-1可以看到，HAL、NT内核和NT执行程序都在超级用户级别下运行，这使得它们可以访问整个机器。环境子系统在用户模式下运行，它为应用程序提供一个特别的API。

HAL通过将硬件转换成标准化的形式，隐藏了不同硬件特有的差异。它就如同一个内核模式下的DLL，其他操作系统进程都链接到它。有了HAL层，我们就可以不考虑由不同CPU构架决定的具体的绝对内存或端口地址之间的差异。HAL甚至能够屏蔽奔腾和Alpha处理器在中断结构方面的差异。硬件抽象层是一项重要的发明，但它也不可避免地会带来性能上的损失。由于可用的处理器范围已经近乎缩减到1种，还有必要保持这种“虚拟器”转换层吗？在David Cutler离开DEC加入微软后，DEC和微软签署了一份约一亿五千万美元的协议，其中包括微软必须继续支持DEC的Alpha处理器。

18.2　Win32：推荐的用户API

NT执行程序与其上的子系统层通信，为不同的应用环境提供各种服务。如前所述，在Windows NT上，运行按照POSIX标准编写的程序是可行的，有些程序可能需要用到OS/2服务，另外一些程序可能会使用Win32。Windows XP支持所有这些类型，为了保证与MS-DOS的后向兼容，操作系统甚至提供MS-DOS子系统，允许我们执行16位的MS-DOS程序。

程序员通过操作系统提供的功能性接口了解操作系统。这些功能性接口称为API（Application Programming Interface，应用编程接口），其数量十分庞大。Microsoft的API称为Win32。Windows 95、98、NT、2000和XP都提供这套API，它涉及屏幕的图形控制、多任务、数据IO，以及进程活动的其他许多重要方面。在本书的支持网站上，给出了Win32函数的清单。在确定所需函数的名称之后，最好查看Microsoft Developer Studio提供的在线帮助，获得参数和功能的详细信息。我发现这种方式比仅使用帮助浏览器定位相关的函数名要容易些。

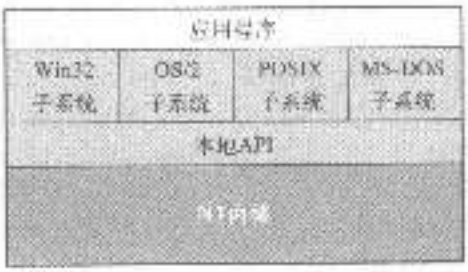


图18-2　Win32和内核

Microsoft提供几套不同的API，如图18-2所示，但Win32是Windows XP、98甚至家族的小成员——Windows CE（针对于持式计算机）平台上推荐的首选程序员接口。为一系列的操作系统提供单个接口标准，可以由此带来兼容性和可移植性方面的优势。实际上，Unix已经完成这种标准化多年。

18.3　进程和线程：多任务

程序载入并作为可执行进程登记到操作系统后，操作系统会为它分配必需的资源，并在它被调

度程序选中时，为它分配CPU的执行时间。程序可以由几个进程组成，每个都单独地由操作系统调度，这些进程要协同一致完成任务，相互之间必须交换数据和控制信息，这样做效率可能会比较低下，因为其间要涉及一些很耗时的系统开销。如果既想保持多任务的优点，又不想招致任务间通信的开销，可以使用线程（thread）。线程是轻量级的进程，它们共享同一执行环境。因此，程序可以编写成几个同时执行的线程，它们都可以访问同一数据环境，因而可以容易地互相通信。最新的Windows线程调度程序支持多种线程间切换的机制，它们包括时间分片、基于优先级的多级循环查询、抢先式和协同式。在Windows NT和XP上，进程的时间片由系统时钟产生，一般为20ms。为服务器设置的时间片一般要长于工作站，因为工作站需要提供更好的交互性能。Windows NT/XP提供32个进程队列，按照优先级进行组织，它们被划分成三个类别：实时、可变级和系统级。尽管Windows NT/XP并非实时操作系统，但它的确支持基于中断驱动IO的抢先式调度。但是，中断处理过程常常不是立即执行，而是经归类后，延期到另外的进程队列中。这就造成中断处理的完成会不确定被延迟。不过，随着3 GHz处理器的普遍安装，这类有关时间问题的担心逐渐变得有些学究气。



图18-3 使用任务管理器显示PC机的任务清单

运行Windows的PC有一个任务管理器实用程序（见图18-3），我们可以通过CTRL+ALT+DEL组合，或在任务栏上右击鼠标按钮调出它来。这个工具显示当前任务（进程）的清单，同时给出它们使用的内存以及CPU的利用率等统计数据。它还可以用来杀掉出问题的任务！

多线程代码的使用又一次使临界段的问题（参见9.5节）浮出水面。利用XP的多处理器功能时，这个问题会变得更难解决。这种情况下，必须预先考虑在不同CPU上运行的线程争用同一独占资源的情况。Windows NT/XP通过提供内核级的自旋锁处理这个问题。它用测试一设置（Test-and-Set）和总线锁定（Bus-Lock）机器指令（如果可能）实现，但在用户API层面，它们表现为信号量的等待（Wait）和信号（Signal）调用。

Windows执行程序使用NT内核提供的服务实现进程管理、内存分配、文件管理和IO设备处理。

18.4 内存管理：虚拟内存的实现

Windows NT和XP是请求驱动的分页虚拟内存操作系统。每个进程在创建时被赋予4 GB的虚拟地址空间，这个空间分成两半：底部的2 GB是用户空间，顶部的一半与系统共享，允许将系统的DLL映射到进程自身的地址空间。数据变量的空间也是在这个虚拟空间内分配。进程已分配的虚拟地址空间保存在磁盘的交换文件中，如12.5节所述，当代码开始运行后，MMU（Memory Management Unit，内存管理单元）将页面从虚拟空间映射到计算机主存内的物理空间。这样，在进程执行时，实际上只需当前的代码页和数据页位于RAM中即可。

18.5 Windows注册表：集中化的管理数据库

Windows注册表是一个重要的信息数据库，其中保存了操作环境诸多方面的配置数据。它取代了早期常常分散在PC文件系统各处的.INI和.PIF文件，这个数据库存储在两个文件中，USER.DAT和SYSTEM.DAT，它们一般存放在C:\WIN32或C:\WIN32\SYSTEM目录下。正是有了注册表，才使得用户选择读取或编辑一个数据文件时，操作系统能够自动识别并启动正确的应用程序。注册表保存文件扩展名以及与之关联的具体应用程序，比如doc与Microsoft Word字处理程序相关联。另一项

有助于笔补PC机中断线短缺的实用功能是硬件配置文件（Hardware Profile）。这项功能允许用户在启动时通过菜单选择具体的一套设备驱动程序。通过这种方式，可以暂停某些设备，释放出IRQ供其他设备使用。

最好使用regedt32程序访问注册表中的信息（见图18-4）。它就像注册表浏览器。信息分别以"项"和"值"的形式存储在注册表中。值可以是二进制或者文本信息。在编辑注册表时，需要加倍小心，避免干扰其他项。通过了解应用程序如何在关闭前将最后打开的文件名称保存在注册表中，我们可以对这个方案有直观的认识。注册表项的名称可以是LastDocument，它的值记录一系列的文件名。另一种更适合于其他情况的方案是层次化的安排，项下面有子项，子项下面再保存子项，直到每个子项都有一个与之相关联的文件名为止。



图18-4　Windows NT注册表编辑窗口

注册表的最上一级是5个主要项，它们是信息树的根，参见图18-4和表18-1。

表18-1　Windows NT注册表的顶级项

HKEY_LOCAL_MACHINE	保存硬件配置、已安装的设备驱动程序、网络协议、软件类别 本地计算机的配置参数 枚举设备配置参数 硬件串行端口的配置 网络用户登录信息 远程管理的安全权限 已安装的软件 系统引导信息
HKEY_CURRENT_CONFIG	保存当前系统硬件配置选项
HKEY_CLASSES_ROOT	保存文档类型、文件关联、外壳接口
HKEY_USERS	保存登录用户的软件偏好和桌面配置
HKEY_CURRENT_USER	保存当前用户偏好的副本

18.6 NTFS：Windows NT文件系统

Windows NT除继承DOS和95/98对FAT文件系统的支持以外，还支持一种新的文件系统，它就是**NTFS**。NTFS对Windows NT/XP的各种访问控制提供完全的支持，主要用在数据安全十分重要的领域。值得注意的是，未经授权的用户不能通过用软盘引导服务器绕过NTFS分区的安全控制。磁盘空间划分成不同的卷，我们可以将卷看做是分区的等价物，不过它可以包括几个分区，甚至驱动器。尽管NTFS声称是非FAT系统，但实际上数据依旧划分成簇，并由单个**LCN**（Logical Cluster Number，逻辑簇号）来索引。为了减少FAT-16系统中常见的磁盘碎片问题，NTFS一般不使用大的簇；实际上，小于512 MB的卷推荐使用512字节的簇。由于NTFS将文件长度按64位数字存储，因此最大文件大小为2^{64}字节，远远超过绝大多数应用的需要。LCN为32位宽，如果使用4 KB的簇，则它限制卷的大小只能达到16 TB。一般地，分区都采用4 KB的簇（与软盘相同）。这个数字与虚拟内存的页大小相匹配，肯定有助于提高内存管理的效率。

为了进一步方便用户，NTFS能够处理最大256个字符的文件名，并支持大小写字母。扩展的名称用在FAT-16文件系统时会被截取。所有磁盘访问活动均可记录下来，以备系统发生故障时恢复之用，这表示数据在这种情况下可以更容易地恢复。但是，这的确会增加处理时间。

尽管只有运行Windows NT及以后操作系统的计算机能够直接使用NTFS文件，但是网络用户，不管他们正在使用的是FAT或Unix文件系统，都可以访问NT服务器上的文件。NT以计算机可以接受的形式将文件提供给非NT计算机。如果Unix主机上安装了**Samba**文件服务器，那么通过Windows XP也能够访问Unix文件系统。尽管Windows提供不同的版本，分别针对桌面和文件服务器应用，但和Unix不同的是，Windows服务器依旧提供GUI用以完成本地的管理任务。让人意想不到的是，Windows XP现在提供Services For Unix（SFU），它包括使用NFS的远程文件访问，从而能够与Unix兼容。

Linux和Windows XP的双重引导系统，可以使用共享的FAT分区在两种操作环境间交换数据。对于大量的数据（大于10 GB），并且安全问题比较重要，建议不使用FAT，而使用NTFS！

Windows高速缓存管理器进程实现了磁盘动态高速缓存方案。文件打开后，它会将一块数据（256 KB）临时性地存储在主存的磁盘高速缓存中，以备再次访问之用。这个机制通过读入或写出较大的数据块，减少磁盘活动的数量。在文件使用期间，它会将文件映射到内存中。高速缓存管理器还可以预测对数据的需求，并预先将数据读入高速缓存中。

18.7 文件访问：ACL、权限和安全

和所有的操作系统一样，Windows XP的基本安全性依赖于一个安全的密码系统。这个密码系统比Unix中采用的密码系统要复杂得多，登录过程中，密码的验证由**域**内的安全访问管理器（Security Access Manager，**SAM**）完成。域是一组互相链接、相互协作的工作站，使用这些工作站的用户又可进一步划分成不同的**工作组**（即Workgroup。对于新手来说，术语总是最大的绊脚石）。如果SAM不在用户登录的工作站内，则需要在网络上传输密码，这样会存在潜在的安全风险。Windows XP采用“暗号”方法解决了这个问题。本地工作站将所有的登录密码传递给加密过程。之后，纯文本密码被删除。这种加密是一种单向的散列函数，除非拥有内幕信息，否则不可能逆向转换出原始的密码。运行SAM的域控制器会响应登录请求，将一个“暗号”送回工作站。这就如同在回答客户电话询问时，银行会要求客户提供一些奇怪的个人信息，比如母亲生日的中间两个数字，以验证客户的身份一样。

系统管理员经常会告诫我们，黑客获得密码更多地是通过写有密码的粘性贴纸和明文传送的数据包，而非出色的算术才能。由于我们想像力的缺乏，甚至于猜密码都不是一件难事。人们常常会选择汽车号码、姓名或生日或者食品的名称作为密码。由于在线字典的存在，试着将每个常用词作为密码进行检验相对容易。你的密码属于哪一类呢？

为了控制对网络中文件和目录的访问，Windows NT还提供密码以外的其他几项特性。文件或

目录的所有者可以设置共享权限，对它们的输出加以限制。由于有共享级别、目录级别和文件级别的权限需要协商，因而它比Unix要复杂得多。如图18-5所示，资源管理器窗口已经打开（在【Start】上单击鼠标右键，然后选择资源管理器）。在要求的目录（C:\TEMP）上单击鼠标右键，调出含有共享选项（【Sharing ... 】）的菜单。选择该项后，属性对话框会弹出来，我们可以设置目录的共享名。权限设置需要单击权限（【Permissions... 】）按钮。三种访问类型分别是：完全控制、不可访问、读取和更改。

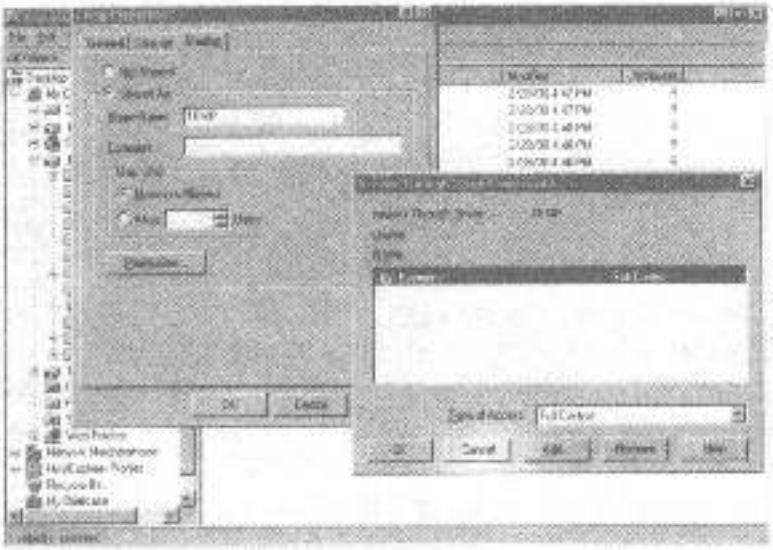


图18-5　设置目录的网络共享及权限

输出目录的新共享名最长可以为12个字符，在整个网络域内可见。要注意，Windows NT中字符串的有效长度很复杂，参见表18-2。尽管为目录设置共享别名一般看来好像没有什么实际的用处，我们可以使用【Add】（添加）按钮向权限对话框中插入用户或用户组。另一种不错的特性是"隐式共享"机制。我们可以输出一个目录，但故意使它不可见，只告诉那些我们想要他们知道的人该共享目录的存在！只需在共享名的尾部插入'$'就能够做到。读者或许已经注意到共享属性对话框上的备注字段。我们可以通过它为每个共享的目录设置一段有帮助的注解，帮助用户找到正确的文件。

表18-2　字符串有效长度的差异

文件名	256个字符
用户名	20个字符
密码	14个字符
机器名	15个字符
工作组名	15个字符
共享名	12个字符

"读取"访问权限的明显含义是提供程序执行权限，并能够访问子目录。"更改"也包括添加、追加和删除文件的权限。"完全控制"比较容易理解。

如果想独立于网络共享功能，进一步控制单个文件和目录的访问权限也是可行的，但仅限于NTFS文件系统，FAT-16文件系统则不行。在资源管理器或我的电脑中，用鼠标右键单击所述的文件或目录，属性对话框提供"权限"设置，可以设置"安全性"，使用这个对话框可以设置单个文件或目录的访问权限，可以针对用户组或单个用户进行设置。

在文件名上单击鼠标右键→Properties（属性）→Security（安全）→Permissions（权限）→Add（添加）→Members（成员）

这一级的文件访问权限最强大，可以改变网络共享的权限。

在另一端，需要访问服务器上甚至邻近工作站上输出的目录的用户，可以采用图18-6所示的技术，将目录安装为本地机器上的虚拟驱动器。

在Start（开始）按钮上单击鼠标右键→Explore（资源管理器）→Tools（工具）→Map Network Drive（映射网络驱动）

浏览器会显示网络上的域服务器或邻近工作站通过共享选项输出的所有可见目录。

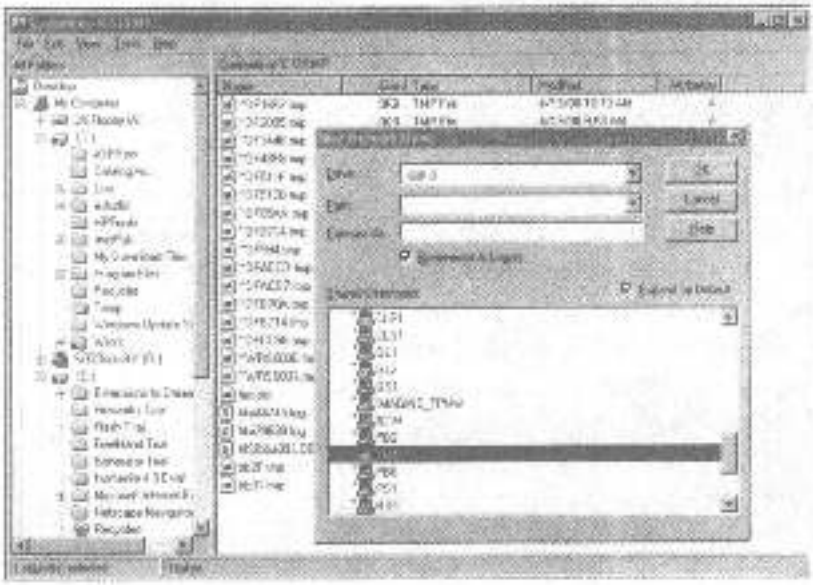


图18-6 将共享目录安装成本地硬盘驱动

18.8 共享软件组件：OLE、DDE和COM

从Windows引入之始，Microsoft一直都在力图提供一种健壮的程序间数据交换的机制。这个机制可以看做是文件和数据块共享的扩展。尽管对象链接和嵌入技术（Object Linking and Embedding，OLE）不过是提供组合文档的标准，但它是第一步。最初这项技术构建在动态数据交换（Dynamic Data Exchange，DDE）之上。遗憾的是，在需要支持音频和视频功能时，仅仅是DDE就不够了。当前的方法是组件对象模型（Component Object Model，COM），COM现在是OLE、ActiveX、ActiveX控件和Microsoft Transaction Server的底层技术。分布式COM（Distributed COM，DCOM）是COM的扩展，它加入了分布式机制。倡导组件对象模型的目的是建立软件组件市场，使得软件可以像芯片那样购买和销售。

18.9 Windows XP主机：Winframe终端服务器

类似于Unix X窗口系统，微软的Windows也可以在远程工作站上显示窗口。这个方案——Winframe，由Citrix开发。从那时起，它就被授予瘦客户端的称号。明显的意图是，重新将处理能力和文件设施都集中到计算机中心那些大型、强劲的服务器上。桌面工作站只需提供最低限度的屏幕和键盘设备以及网络访问。当服务于数百个（甚至上千个）远程窗口显示时，网络通信量会迅速增长，因此，我们至少需要100BaseT交换式星形拓扑结构。对于大型的网络，这种方案为系统管理员带来的优点是，他们能够更紧密地控制用户的计算环境，甚至可以规定用户的桌面配置。如果用户到处移动，那么这种方案也可以使他们不管在什么地方登录，都能够得到标准的屏幕和一系列的工具程序。但是，用户越来越不满意这样不灵活的方式。当与其他许多（未知）用户共享资源时，不可解决的困难是没有办法预测请求的响应时间。这也是早期的大型机用户最不满的问题之一。Windows服务器依旧存在这个问题。LAN传输率从来不会比PC和显示器之间VGA电缆更快，在繁忙的周一早晨，中心服务器根本不能让所有的用户都满意。

遗憾的是，这种方案的确如同返回到20世纪50年代，当时用户的需要得不到注意，使用计算机中心的各个部门之间冲突不断。结果是，如我们所见，与本地控制和更佳的灵活性相比，集中化计算黯然失色。我怀疑随着网络受到过多万维网下载的冲击，用户会再次要求在选择所需的计算资源上拥有更多的自主权和更大的权限。

18.10 小结

- 根据IBM OS/2和DEC VMS的经验，Windows NT面向大型的公司网络用户。
- Windows XP是抢先式、多任务、虚拟内存的网络操作系统。它基于模块化的面向对象设计。由于HAL虚拟机器层的存在，使得它可以更容易地移植到不同的处理器。
- 模块化使得操作系统能够同时支持几种API，从而能够执行MS-DOS、Windows 3.1、OS/2和POSIX程序。推荐的接口是Win32 API。
- 名为注册表的中心数据库，取代了早期操作系统中所有各种各样的初始化文件。
- Windows XP通过增强的密码处理以及扩展的权限机制，提供安全的联网计算。
- 改进的文件系统，即NTFS，可以处理更大的文件以及分布在网络上的数据。文件管理器内建了更好的故障恢复机制。
- XP工作站可以运行X窗口服务器，XP也可以访问Unix输出的文件，因此，Unix主机能够和XP网络协同工作。

实习作业

我们推荐的实习作业包括使用Windows XP操作系统。包括安全机制的应用（共享和权限），以及网络虚拟驱动的安装，还应调查研究一下个人密码和访问控制列表（Access Control List，ACL）。

练习

1. 用户对家用和商用电脑功能上的要求有什么不同？Windows XP如何解决这种不同？
2. 什么是Unicode？为什么Microsoft决定不再采用ASCII？
3. 向新处理器移植时，Windows XP的体系结构如何发挥作用？它如何处理中断设备的不同？
4. 文件高速缓存管理系统的目标是减少磁盘访问，加速程序的执行。编译器在这种活动中起到什么作用？
5. Windows XP通过什么方式与组织内其他运行不同操作系统的联网主机进行协作？
6. Win32是Windows XP的标准API。概括一下它为程序员提供的各种类型的功能。
7. Windows NTFS文件系统提供什么额外的机制，协助磁盘崩溃时的恢复工作？
8. Windows NT完成第4版的修订工作后，管理屏幕的图形代码被移入内核中，这是为了提高速度。你说说看，这种改变存在什么问题吗？
9. 线程和进程的区别是什么？在什么情况下会遇到临界数据的问题？如何解决这样的问题？
10. 由于Windows XP基于面向对象的准则，因此消息的有效传递至关重要。它为执行这种活动提供了什么机制？

课外读物

- 下面这个网站提供一些动画视频短片形式的教程：
  http://www.mistupid.com/tutorials/windowsxp
- Glenn和Northrup（2005）。
- 在Nutt（1999）的后半部分和Silberschatz等著（2000）中，对Windows做了简短的介绍。
- Karp等著（2005）。

- Simon等著（1995）。
- Pearce（1997），大量有关XP的资料。
- MSDN（Microsoft Developer Network，微软开发者网络）网站提供丰富的资料。它含有许多关于COM、OLE、DCOM以及其他许多主题的文章：
  http://msdn.microsoft.com/library/default.asp
- 微软Windows操作系统的发展史：
  http://en.wikipedia.org/wiki/History_of_Microsoft_Windows#Early_history
- 这些网站可以通过本书的配套网站访问：
  http://www.pearsoned.co.uk/williams

第19章　档案管理系统

档案管理，即组织和长期存储大量数据的能力，依旧是计算机提供的最重要的服务之一。数据库可以为所存储的数据提供更为灵活的访问，它可以替代许多应用程序中使用的单个文件。硬盘可以提供对数据的快速方便访问。档案管理系统涉及硬盘、软盘、CD-ROM、CD-RW和备份磁带。这个领域已经扩展到通过Internet能够访问到的任何文件。

19.1　数据存储：文件系统和数据库

几乎所有操作系统中，最有用的功能部件之一就是磁盘档案管理，通过它用户无需对底层的机制有很深的了解，就可以保存和使用数据和程序。

第12章中介绍了磁盘驱动器如何记录和还原大量二进制数据的技术细节。在此，我们主要关注操作系统如何有效地隐藏这些复杂活动，使用户可以更方便地使用磁盘硬件和其他大容量存储设备提供的功能。在输出到磁盘之前，操作系统首先将需要归档的数据按照某种次序划分成一系列的数据块。在构成文件的所有数据块中，查找特定的记录是十分耗时的任务，采用一些索引方法可以加快搜索的过程。如果应用程序自身能够提供目标数据块的编号，则可以直接将读取磁头定位到正确的磁头、柱面和扇区，避免顺序扫描工作。这种做法叫做直接文件访问。它在某种程度上部分地绕过了我们曾提到过的用户所熟知的逻辑文件访问。

另一种访问磁盘文件中数据记录的方式，是基于索引的顺序访问，即由单独的索引给出目标记录所在的数据块编号，进而得出它在磁盘上的位置。表19-1对比了这三种方法：顺序访问、直接访问和基于索引的访问。此外，现今数据记录之间的关系已不再仅仅限于单一的线性序列，越来越流行的做法是，使用记录间的多重链接关系。这需要数据库的支持，以跟踪数据记录以及它们之间的相互关系，并执行复杂的查找任务。

表19-1　数据档案管理和数据库

数据组织	应用类型	优点	缺点
顺序访问	批处理	简单，高效	维护困难；数据需要排序
基于索引的顺序访问	批处理	顺序或直接访问；无数据排序	索引需要空间；效率稍低
直接访问	在线	无数据排序；访问快速	空间效率低；不方便使用
数据库	在线	访问灵活	性能差；维护成本高

顺序文件组织方式是最简单的。数据记录按照关键字段顺序存储，数据记录都由指定的关键字段的值惟一区分。访问时顺序访问每个记录，直到找到目标为止，对于较大的文件，这个过程会很缓慢，但适合于执行批量处理，比如季度汽油费账单计算或每月薪资。

为了加速这个十分缓慢的技术，**直接文件访问**使用数据记录的关键字段值计算出磁盘数据块的地址，然后使用这个地址定位出磁盘上的记录。它涉及多对少的映射，这种映射常常使用散列算法来执行，执行这种方案不需要单独的查找索引。Unix只支持顺序访问文件，一些程序员认为这是Unix的一个缺点。

基于索引的顺序文件融合了前面介绍的两种方案。数据按照关键字段进行排序，但也可以使用与文件同时建立的辅助性索引直接访问。在索引的帮助下，可以直接定位数据记录，或者根据关键

字段的次序顺序访问。

数据库需要在数据记录中包括附加的字段，用做链接指针。它们实现了存储设备上数据项之间关系的逻辑组织。我们可以将它们看做是数据记录间的交叉引用。数据库记录依旧按照关键字段排序。但同时还依靠这些指针链接支持许多用户需要使用的更为复杂的SQL查询："所有那些生活在布里斯托尔，年龄在40以上，已婚，有住宅，但住宅的窗子尚未安装双层玻璃，每月的汽油账单大于200英镑的人的电话号码为多少？"为了提供对这类信息的快速访问，需要构建和维护所有数据记录的复杂索引。

任何使用过PC的人，都会熟悉文件浏览器中显示的层次化的档案管理方案。到Windows NT为止，该功能都由单独的程序来提供（见图19-1），但在Windows 2000和XP中，文件和万维网浏览功能都被统一到Internet Explorer（IE）中，IE成为统一的资源浏览器。这是迈向无缝浏览整个万维网的趋势，不管是本地机器，还是远隔万里的服务器。Linux中的Web浏览器也支持本地文件和目录的图形化显示，它协助用户浏览目录和文件，更好地组织自己的工作。图19-1中，左右侧窗格的中间可以看到文件Msdev.exe，这是Microsoft Developer Studio 5的主执行文件。它在目录中隐藏得很深：C:\Program Files\Common Files\DevStudio\SharedIDE\bin。如果尚未创建桌面图标，启动一个程序最简单的方式，是使用任务栏的Start（开始）→Run（运行）功能。

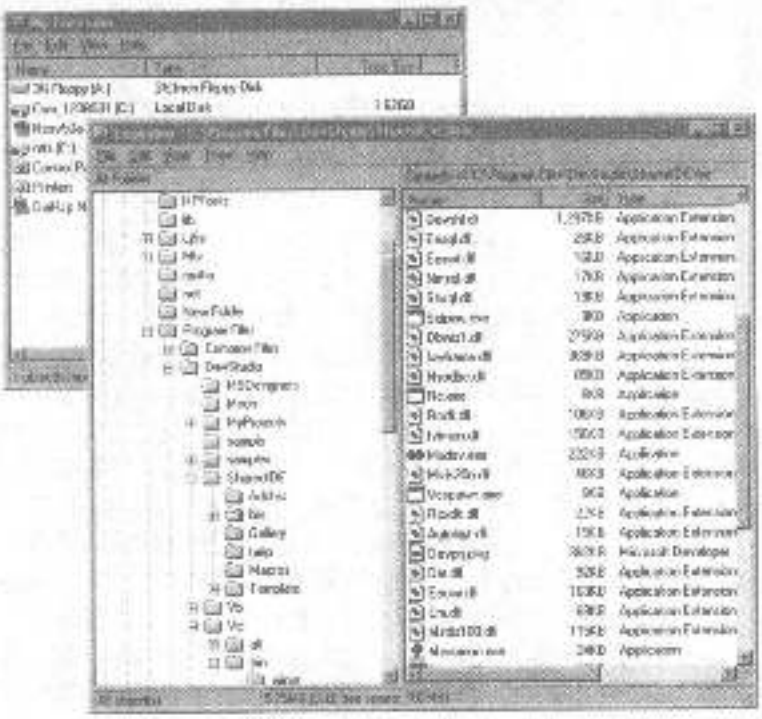


图19-1 Windows NT文件浏览器中的文件体系

Unix也一贯将其系统文件按照层次体系进行组织，为用户提供同样的功能。Sun上类似的文件浏览器是文件管理器（File Manager），参见图19-2（Unix用户在大多数情况下，依旧钟情于旧式的ls/cd方式）。

如果曾使用过平面文件系统——数以百计的文件只是简单地保存在单个顶层目录中，没有任何分类和归组，就会知道找出之前加工过的文件是多么困难。这就如同档案管理柜中没有抽屉，只有一个书架堆着所有的纸张。未来通过Google Desktop对内容建立索引，可以改善这种情形。处理层次结构的目录和文件是操作系统的重要职责。层次档案管理方案的第二项优点是能够容忍同名的文

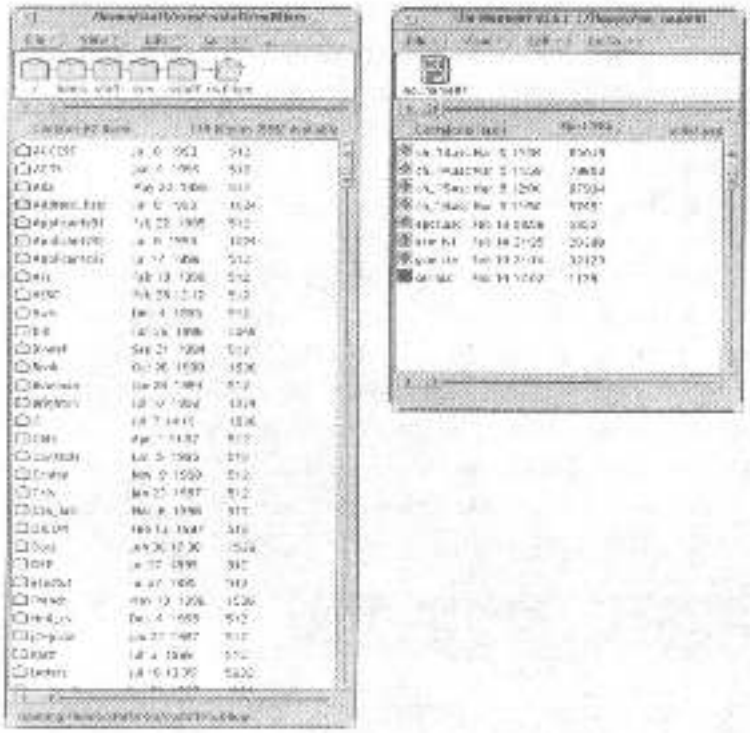


图19-2　Sun文件管理器及软盘浏览器

样。如果两个文件使用相同的名字，这种情况时有发生，只要文件在不同的目录中，操作系统就能够区分它们。这是因为实际上的标识符并非文件的名字，而是它的路径名，即它够的目录位置。比如，/home/users/staff/cs/rwilliam/Book/ch_19。我确实还有一个名为ch_19的文件，在/projects/robvob/book目录中，但这不会对操作系统造成直接影响。要得到含有本章文字的文件，我只需键入完整的绝对路径名，明确无误地告诉操作系统我需要那个文件。如果每次访问一个文件都使用完整的绝对路径名，肯定会很不方便。因此操作系统提供多种快捷方式，最有用的方式是使用相对路径名。外壳可以识别出绝对路径名，因为它总是以"/"字符开始。如果不以"/"字符开始，外壳则在文件名之前加上当前工作目录的路径名，以创建"/"字符开头的绝对路径。这种根据当前位置引用文件的方式，仅在用户能够很容易地使用cd命令改变当前工作目录的情况下才有效。另一种引用文件的快捷方式是相对于用户自己的主目录。默认情况下，登录后用户就位于这个目录中。Unix知道~（代字号）表示用户主目录的绝对路径名。这样，我就可以使用简写的引用方式~/Book/ch_19。

现在，文件系统一般都构造成层次体系，就好像大树四通八达的根系，每个文件都是上根系的末端。Unix一般将文件系统的顶点称为**root**。通过在目录中嵌入子目录，不仅可以帮助用户组织他们的工作，而且有助于减少系统定位文件所需的时间。

程序员可以简单地为每个目录、子目录和文件赋予惟一的名字，来保存和还原数据及程序。文件命名机制使得用户不再需要知道磁盘上所有数据块的磁道和扇区号。而在之前，维护基于磁盘的目录和文件十分繁琐，简直就是一场噩梦。操作系统还提供打开、读取、写入和关闭文件的函数。它们还必须管理磁盘空间的分配及恢复，还要提供一些控制访问的安全机制，以保证只有授权的用户能够读取、写入或删除文件。后面，我们还会再次讨论这个问题。

在向文件输出数据时，一般地，应用程序会将较小的数据项聚集成记录，之后发请求给操作系统将记录输出到磁盘。虽然我们说"写文件"，但文件只是一组能够通过惟一标识符方便访问的数据。数据之外，并没有物理保障可以确保文件的完整性，构成文件的各个数据块仅由指针链接在一起。

这为文件管理例程在磁盘上为文件分配空间和安置文件提供了充分的灵活性。在12.7节曾说过，数据以扇区为单位，一般为256字节或512字节，写入到磁盘的磁道上。这些扇区可能被进一步关联到一起，形成4 KB大小的簇（cluster），以克服空间分配上的问题。接下来，磁盘的读取或写入的最小单位都为4 K大小的块。这确实会浪费一些空间，因为并不是每个文件都能真满最后4 KB的簇。实际上，平均下来，每个文件将会浪费最后一簇空间的一半。这并不严重，但是值得注意。使用ls -R | wc -l，我发现我使用的Unix工作站具有超过4800个文件，这会造成约10 MB的浪费。今天即使容量最小的磁盘驱动器都提供超过8 GB的空间，这也许完全可以接受。每个驱动器我只失去0.125%的空间。

构成文件的多个数据块不必一定是连续扇区构成的连续簇；它们可以如此，并且常常如此，散布在磁盘的整个表面，只要磁盘尚有空闲空间可用。这样，一段时间后，经过文件的写入、删除和替换，磁盘上就会出现碎片。尽管这是不可避免的，但它会使文件的读取变慢，因为需要更频繁地移动磁头。在Windows XP上，为了维护磁盘的性能，可以用碎片整理程序来重新组织文件分散的数据块，尽量将它们安排在邻近的区域，以降低访问数据时磁头移动的次数。Unix采用不同的策略，它限制数据块必须分配在邻近的柱面，有些类似于建立大量分区以约束磁头的移动。

在图19-3中，构成文件的各个数据块分散在磁盘底部的单个表面上。读取或修改数据时将会需要大量的磁头移动。但是，如果将同样的数据重新写入单个柱面，同样如图所示，则在定位出第一个数据块之后，就不再需要移动磁头。

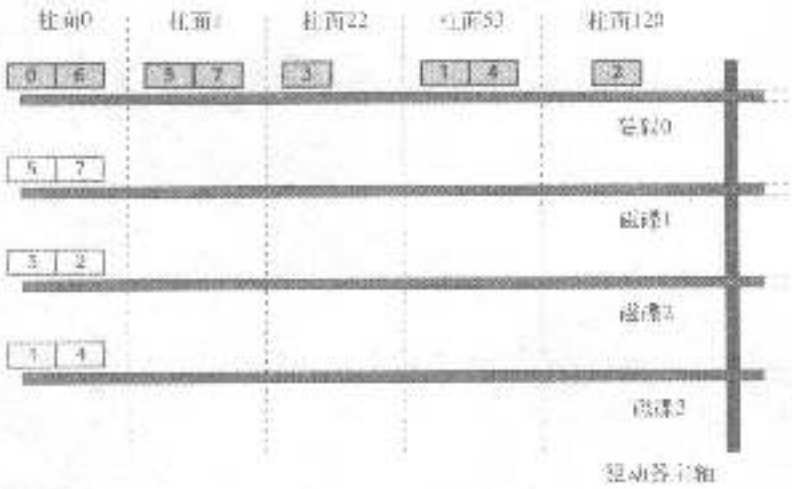


图19-3 数据块1、2、3、4、5、6和7的另一种布局
（可以降低访问时间，减少磁头移动带来的延迟）

根据应用的不同，文件中数据存储的格式多种多样：二进制整数、复杂的记录、简单的ASCII字符流或许多其他格式。一些操作系统识别这些变体，并提供专门的访问函数。其他操作系统，如Unix，避开了这类复杂性，只接受字节流。这样，所有的数据编排工作都由用户的应用程序处理，或至少由专门的库来执行。各种方式都存在争议；Unix最初是作为文本处理工具开始的，因此它仅专注于字节宽度的数据，这并不奇怪。

PC实用程序fdisk可以将磁盘的柱面划分到不同的分区中（见图19-4）。并能在每个分区的起始处记

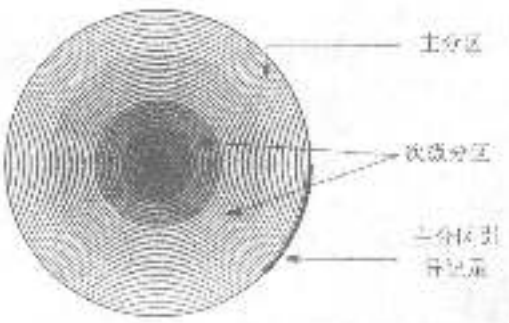


图19-4 含四个分区的硬盘

或分区引导记录（Partition Boot Record，PBR）（见图19-5）。fdisk中非正式的/mbr选项可以执行后一项操作。特殊地，主分区引导记录（Master Partition Boot Record，MPBR）长度为512字节，位于磁盘驱动器的第一个扇区（柱面0，磁头0，扇区1）。其中含有PC打开电源后需要运行的次级（second）程序。BIOS PROM显然会最先运行，它载入MPBR程序，并将执行控制权转移给它。值得注意的是，MPBR多年来一直是病毒攻击的目标，病毒的攻击会导致BIOS到操作系统的转移发生故障。所有PC机上的操作系统（MS-DOS、OS/2、Windows 3.1、Windows 95/98/NT/2000/XP、Linux和Solaris）都识别和使用同一个MPBR。它保存重要的信息，基本上是分区的位置，以及哪个分区是活动的，并提供引导时的操作系统代码。不方便的是，MPBR只提供四个分区的空间。但是，每个分区都可以设置引导不同的操作系统。如果不同操作系统之间需要交换文件，常见的做法是将一个分区设为MS-DOS FAT-16文件系统。现在大多数操作系统依旧可以处理这种类型的文件系统。

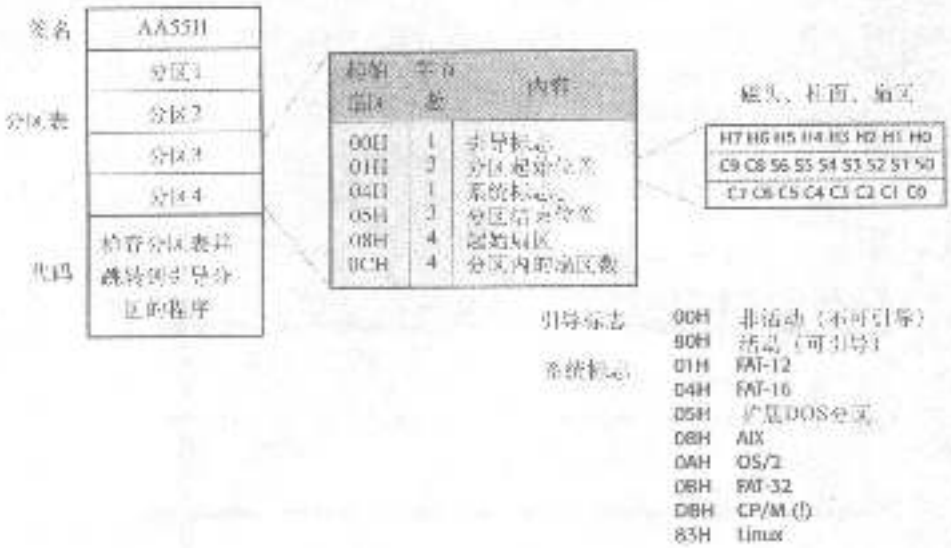


图19-5　PC磁盘的主分区引导记录

对于大容量的硬盘，考虑到Windows能够支持24个分区——分别用从C到Z的字母表示，四个分区显然不够。为了突破这种限制，在需要更多分区时，我们采用主分区和扩展分区，扩展分区可以进一步划分成多个逻辑的DOS卷。这些卷不记录在MPBR表中，因此不能引导。它们的初始柱面编号以链表的形式保存在各自的分区记录中，每个分区记录都指向下一个记录。

在磁头崩溃时，分区也可以提供一些安全措施。如果一个分区被破坏，其他分区不受影响，在系统从CD-ROM或软盘引导后，依旧可以读取它们中间存储的数据。

除非使用交互式的引导管理程序，比如LiLo，否则MPBR总是将引导过程转移到指定为活动的分区，在那里，存在另一个表——分区引导记录（Partition Boot Record，PBR），参见图19-6。它们位于磁盘分区的开始处，记录有重要的参数。它们由高级格式化实用程序format安装。

如本章开始处所述，所有操作系统都提供的一种功能就是，可以将名字作为文件的标识符，这就是目录的作用。它允许操作



图19-6　分区引导记录（非主分区）

系统将用户的文件名转换成磁盘驱动器需要的数字格式。在硬件层面，所有真正需要的实际参数是柱面、磁头和扇区（CHS）。之后，控制器就可以正确地将读取磁头定位到正确的位置，等待扇区的到达。仅在极少数情况下，我们需要使用CHS参数访问磁盘。即使在文件系统被破坏时，通过磁盘工具的帮助，比如Norton提供的工具软件，常规的干预依旧是在中间的逻辑块编号（Logical Block Number，LBN）层面进行。LBN只是按照数据块或簇在磁盘上出现的先后次序编制的序号。簇从头至尾依次编号，每个分区拥有单独的一系列编号。需要再次提及的是，索引号的宽度决定了可用存储空间的最大大小。

19.2 PC文件分配表：FAT

对磁盘数据的访问，最终依旧要通过告知磁盘控制器访问哪个柱面、磁道和扇区来完成。这就是柱面、磁道、扇区（Cylinder/Head/Sector，CHS）访问，代表最底层的物理层。但为将问题简化，操作系统执行逻辑到物理的转换。磁盘（或分区）内所有的物理扇区都被从外向内顺序编号。逻辑扇区编号最初为12位，为处理更大的磁盘，后来增长到16位，直至32位。物理扇区为512字节，使用逻辑扇区编号可以访问的最大卷为512×2^16字节=32 MB。这是远远不够的，为了访问更大的卷，人们将扇区分成4 KB大小的“逻辑扇区”，或称为簇。这种情况下，文件最小为4 KB。但将最大分区大小提高到256 MB。

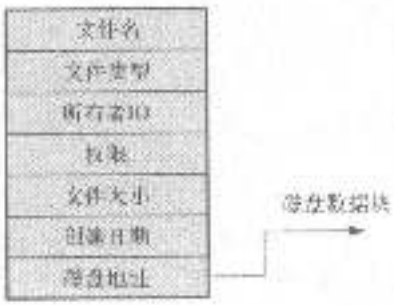


图19-7 目录项的基本信息

访问文件时，需要完成从逻辑文件标识符到物理数据扇区定位符的转换。操作系统通过目录完成这种功能。每个文件在创建之后，操作系统会在相关磁盘分区的目录中创建一个条目。这样做的目的是，将用户的文件名转换成磁盘控制器可以用来定位文件数据块的数字化磁盘地址。每个磁盘分区都有单独的目录。目录项还保存其他重要的管理性信息。参见图19-7，通过保存拥有者的ID以及访问权限参数，操作系统可以控制各个用户分别可以对文件进行什么样的操作。FAT文件系统是MS-DOS引入的文件管理技术，至今依旧在Windows中广泛使用。图19-8对其做了简单的描述。如我们前面所述，用户希望使用名字访问他们的文件，但磁盘控制器使用柱面、磁头和扇区（CHS）编号来定位读取磁头。另一种引用簇或使用块或簇的编号，即所谓的LBN（Logical Block Number，逻辑块编号）。FAT访问使用LBN，但将其称为簇编号，这意味着在访问文件数据之前，我们需要完成双重转换：使用文件名从目录获得首个4 KB簇的LBN，然后将这个LBN转换成正确的CHS。所有这些都由操作系统完成，不需要用户的干预。

使用FAT文件系统时，每个硬盘分区都需要一个独立的目录和FAT。如图19-8所示，通过DOS FAT方案访问文件时，首先需要在目录表中定位文件名和扩展名，找出该分区中每个文件起始簇的LBN。最初的LBN可以用来访问文件的第一个数据块。对于后续的簇，目录就无能为力了，必须引用FAT（File Allocation Table，文件分配表）中的数据。对于每个大小超过单个簇（4 KB）的文件，FAT中都会为其保存一个链表。将大文件写入到磁盘时，FAT中会建立一条由簇编号构成的链式结构，将各个数据块链接在一起。因此，文件是通过FAT中簇编号的链表结合起来的。在图19-8中，文件TEST.DAT的簇序列是4-7-6-10。我们可以从目录中保存的第一个节点定位出文件的簇序列。链式结构由FFFH标记终结，FAT簇链可以扩充、缩短，也可以插入和删除，十分灵活。这种方案可以处理的最大文件大小与簇的大小及LBN的宽度相关。LBN已经从最初的12增至16，现已达到32位，分别对应FAT-12、FAT-16、FAT-32。最后的版本（32位FAT）使得这种方案可以处理16 TB的磁盘卷。对于当前的应用来说已经足够了！由于FAT对于档案管理系统至关重要，并且没有采用任何纠错编码，为了安全起见，它在磁盘上有两份！目录被破坏后，我们就无法访问其中的数据文件。更坏的情况是，如果FAT发生混乱，有可能要花费数小时的时间从磁盘恢复宝贵的数据。

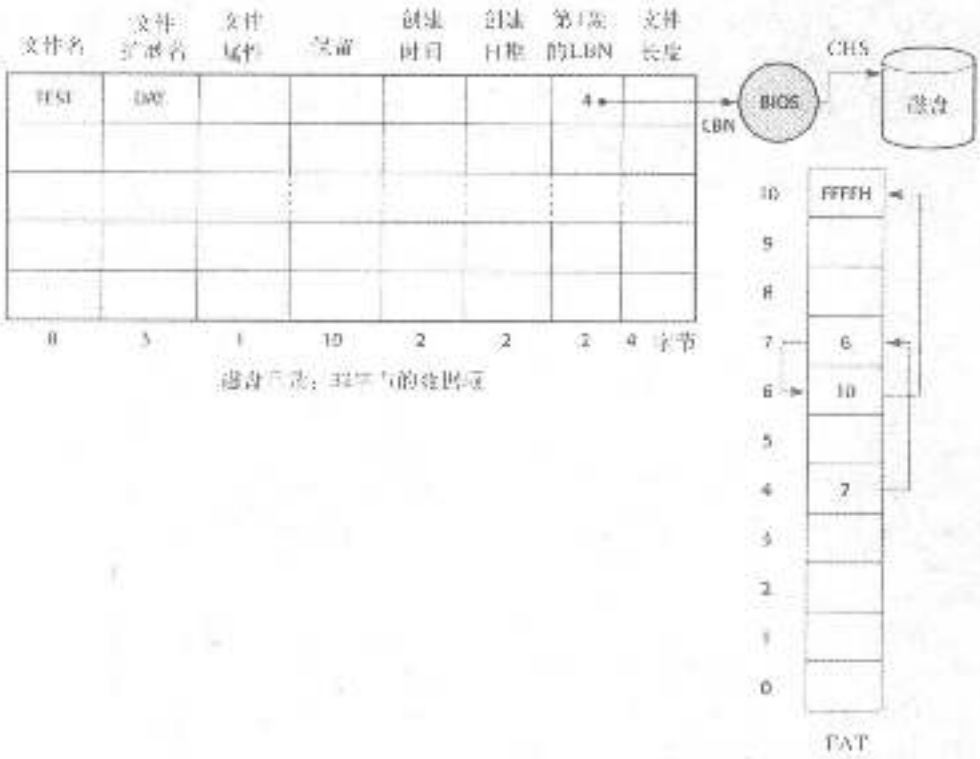


图19-8　FAT-16目录和文件分配表

MS-DOS和早期的Windows系统中，文件名都限于8个字符，Windows NT将其扩展到255个字符。为了维护兼容性，Windows NT采用一种聪明的剪裁方法——取255个字符的前6个字符，并附加一个编号，使之成为惟一的标识符。这样，robert_williams.txt将会缩写成robert~1.txt。如果还存在类似robert_something.txt的文件名，则后面的整数编号递增。

随着IDE驱动器的容量越来越大（大于200 GB），FAT方案在性能上的限制变得越来越关键，人们采纳Windows NT的部分动机，就是为了使用更适合于大量数据的NTFS。不过，一般磁盘上还是至少有一个FAT分区，人们依旧将软盘和闪存盘格式化为FAT文件系统。

初次格式化时，格式化过程会检测故障扇区，并将FAT中对应的条目标记为不可用。簇链中最后的簇所对应的FAT项被标为FFFFH，而未使用的簇被标记为0000H。由于任何表的长度都由寻址索引的宽度决定，因此，16位的情况下，FAT为65 536项，以这种形式，它只能管理最大64 K×1 KB=64 MB。现代的磁盘驱动器动辄80 GB以上，并且不断地增长，相对而言，这个容量太小了。为了将取值扩大到更实用的范围，首先要增加簇的单位，然后是簇索引本身需要扩展到32位。表19-2给出了简短的汇总。

表19-2　FAT簇大小和卷的容量

扇区大小（字节）	每簇扇区数	簇大小（KB）	簇索引大小（位）	最大卷容量
512	4	2	16	128MB
512	16	8	16	512MB
512	32	16	16	1GB
512	64	32	16	2GB
512	16	8	32	32TB

总之，为访问FAT系统中文件的数据，我们必须提供文件名。目录会将文件名转换成首簇的

LBN，根据它可以转换出首扇区的生面、磁头和扇区值。对于后续的数据块，目录就没有用处了，此时需要参照文件分配表中保存的LBN的链表。

19.3 Unix索引节点：不同的方式

尽管表面上层次文件系统的布局十分类似，但Unix组织目录和访问文件的方式与Windows不同。图19-9绘出文件系统的管理结构。它从含有系统装入程序的引导块（Boot Block）开始，接下来，超级块（Super Block）中记录的是有关文件系统的信息：最大大小，索引节点数，数据块数目。它还保有一个指向空闲索引节点块清单的指针，供创建新文件时使用。这些信息在引导期间会装入到主存中。在引导时，Unix至少需要一个文件系统。引导后我们可以使用mount命令添加额外的文件系统。遗憾的是，这些有意义的功能常常仅限于超级用户，可以试试/etc/mount，看看会发生什么（见图19-10）。

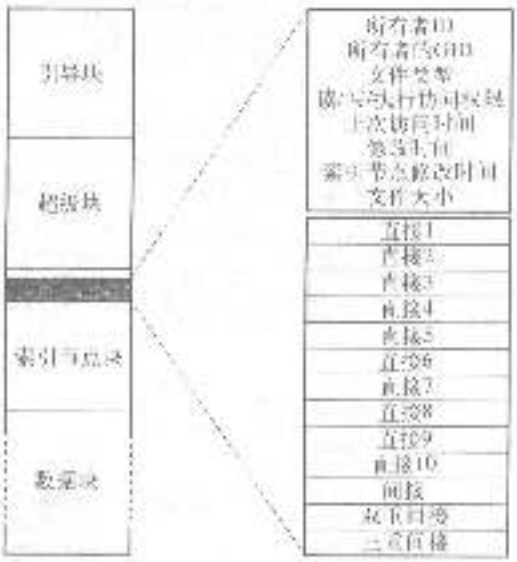


图19-9 Unix索引节点的文件访问记录

```
rob@milly [20]/usr/sbin/mount
/  on  /dev/dsk/c0t0d0s0 read/write/setuid on Mon Jul 19 08:12:44 2000
/usr on /dev/dsk/c0t0d0s3 read/write/setuid on Mon Jul 19 08:12:44 2000
/proc on /proc read/write/setuid on Mon Jul 19 08:12:44 2000
/dev/fd  on  fd read/write/setuid on Mon Jul 19 08:12:44 2000
/var  on  /dev/dsk/c0t0d0s4 read/write/setuid on Mon Jul 19 08:12:44 2000
/cache/cache1 on /dev/dsk/c0t0d0s6 setuid/read/write on Mon Jul 19 08:13:4
/cache/cache2 on /dev/dsk/c0t1d0s7 setuid/read/write on Mon Jul 19 08:13:4
/cache/cache3 on /dev/dsk/c0t2d0s7 setuid/read/write on Mon Jul 19 08:13:4
/cache/cache4 on /dev/dsk/c0t3d0s7 setuid/read/write on Mon Jul 19 08:13:4
/cache/cache5 on /dev/dsk/c0t1d0s7 setuid/read/write on Mon Jul 19 08:13:
/opt on /dev/dsk/c0t0d0s5 setuid/read/write on Mon Jul 19 08:13:45 2000
/tmp on /dev/dsk/c0t0d0s1 setuid/read/write on Mon Jul 19 08:13:45 2000
/tftpboot on /dev/dsk/c0t1d0s0 setuid/read/write on Mon Jul 19 08:13:45 19
/home/student/csn/BSc/CRTS/2 on /dev/dsk/c0t1d0s3 nosuid/read/write/quota
/home/student/csn/BSc/other on /dev/dsk/c0t1d0s4 nosuid/read/write/quota on
/home/student/csn/BA/other on /dev/dsk/c0t1d0s5 nosuid/read/write/quota on
/home/student/csn/PhD on /dev/dsk/c0t1d0s6 nosuid/read/write/quota on Mon
/home/student/csn/BSc/CxIS/2p  on  /dev/dsk/c0t2c0s1 nosuid/read/write/quota
 . . .
```

图19-10 Unix mount表，显示文件系统的挂接点

Unix文件系统十分依赖叫做索引节点（index node，缩写为**inode**）的数据结构。每个文件和目录都有惟一的索引节点块，其中保存指向文件数据块的指针。一般地，索引节点长度为64个字节（见图19-9），除含有访问控制信息外，还有10个指向文件数据块的指针，每个数据块可以为文件保存4 KB大小的数据。如果文件要求更多的容量，还有两个扩展的间接指针，每个连接到另外10个索引节点块，它们又会指向100个数据块。如果这样提供的文件容量依旧太小的话，则还有一个双倍间接指针可供使用，它能够访问1000个额外的数据块。索引节点块的结构可以在头文件/usr/include/sys/stat.h中得到。图19-11给出C结构stat，从中可以看到用来访问数据块的字段。记录字段st_ino最重要，因为它保存着索引节点的值，只有通过它才可以找出数据的第一个块。程序员可以使用函数int stat(const char *path, struct stat *buf)找出文件的索引节点记录。

```
struct stat {
        dev_t         st_dev;       /* device holding the relevant directory */
        long          st_pad1[3];   /* reserve for dev expansion, */
        ino_t         st_ino;       /* inode number */
        mode_t        st_mode;
        nlink_t       st_nlink;     /* number of active links to the file */
        uid_t         st_uid;       /* file owner's ID */
        gid_t         st_gid;       /* designated group id */
        dev_t         st_rdev;
        long          st pad2[2];
        off_t         st_size;      /* file size in bytes */
        long          st pad3;      /* reserve for future off_t expansion */
        timestruc_t   st_atime;     /* last access time */
        timestruc_t   st_mtime;     /* last write time (modification) */
        timestruc_t   st_ctime;     /* last status change time */
        long          st_blksize;
        long          st_blocks;
        char          st_fstype[_ST_FSTYPSZ];
        long          st_pad4[8];   /* expansion area */
```

图19-11　Unix文件系统中索引节点的结构

层次目录结构的管理，以及使用文件名替代索引节点号的机制，是通过**目录块**完成的。目录块保存特定目录的一系列索引节点指针/文件名对，在访问文件时，通过它们可以完成符号名称到数字的转换。重要的是要记住，索引节点块有几种类型，分别处理数据文件、目录、设备和管道。因此，数据文件索引节点提供表示数据块位置的指针。目录索引节点有表明索引节点指针/文件名对所在数据块的指针。

根目录索引节点在文件系统的最顶层，它有自己表示目录块的指针，每个表示一个子目录，比如/etc、/bin或/home。这些目录块中的数据项将会指向其他的目录索引节点，它们又会定位出各自的目录块……依次类推，构成文件系统的层次体系。观察图19-12，可以更好地理解这种思想。

512字节的磁盘扇区可以进一步归组成块，以加快访问速度，如前一节所述。BSD 4.3版的Unix为降低每个文件的最后数据块浪费的空间，采用可变块大小。这样，每个文件都由保存10个指针（32位扇区编号，指向该文件的数据块）的索引节点记录来表示。如果文件需要更多的空间，另外三个间接指针可以链接更多的索引节点块，这些索引节点块可以指向数据块，也可以指向其他的索引节点块。

目录是一种特殊类型的数据文件，其中含有文件名和对应的索引节点编号。实用程序fsck（文件系统检查）会彻底地检查文件系统的一致性，在遇到错误时会尝试修复。

目录实用程序ls显示当前目录块中所有的符号名称。ls -l可以列出每个文件的索引节点中的部分字段。Unix中，目录文件含有文件名与其分配的索引节点编号的清单。这对于访问文件的第一个

块来说已经足够。图19-12和图19-13展示索引节点如何标识数据块和目录。

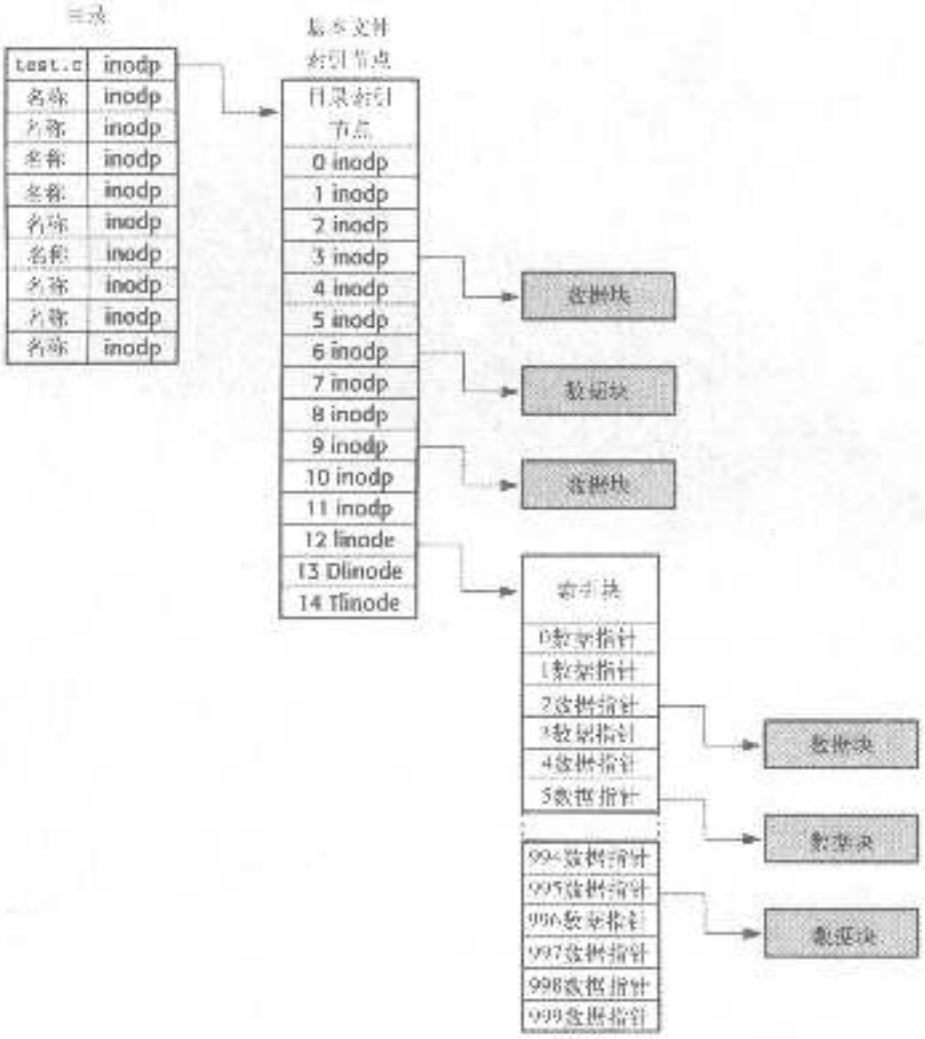


图19-12 表示文件数据块的Unix索引节点指针

Unix将文件当做字节的线性序列。任何内部的结构都由应用程序的程序员负责。IO设备也是一种文件类型，这样是为了提供更一致的方式来处理数据输入输出。磁盘、终端和打印机都在文件系统目录/dev中有指定的索引节点。我们可以将这些做是所有附加设备的软件映射。通过这种方式，我们可以将它们按照文件对待；例如，使用同一命令，可以直接将数据定向到设备：cat device.h test.c > /dev/tty。

为降低磁盘碎片，Unix采用一种有效的策略。它不再从单个超级块池中分配空闲的块，而是将它们划分成几个区域性的池。通过这种方式，块都就近分配，尽可能地使用同一柱面内的块。这会降低甚至消除由于磁盘碎片造成的额外的磁头寻道活动。

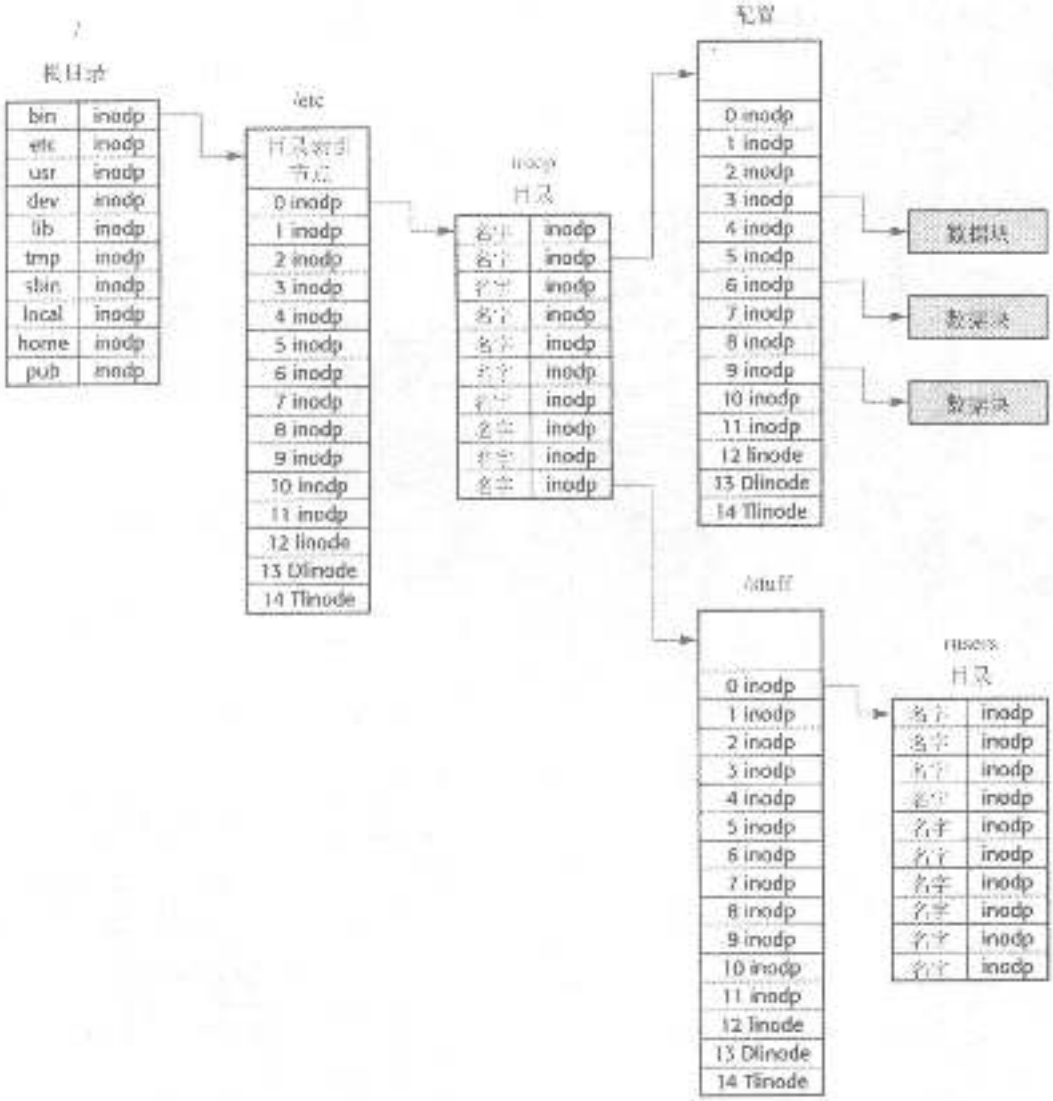


图19-15　Unix目录和索引节点表

19.4　Microsoft NTFS

Microsoft引入NTFS文件系统，以取代广泛采用但同时局限性也较大的FAT方案。FAT最初是在20世纪70年代为MS-DOS的首次发行而设计的，与早期的CP/M操作系统关系甚密。尽管FAT-16已经能够处理更大的磁盘卷，并且其后继者FAT-32也已实现，但在处理大型的文件时，它依旧在基本结构和功能方面存在严重的缺点。不过，由于众多的用户依旧使用FAT文件系统，因此它依旧为现在的操作系统所支持。FAT和NTFS存在明显的不同，它们不完全兼容。使用FAT分区或访问FAT软盘，需要运行不同的软件驱动程序和文件管理器。有意思的是，同样的硬件可以处理这两种配置。

如我们所预期，NTFS提供层次文件系统，与Unix类似。文件名标识符最大长度为256个Unicode字符，一个名字就需要500个字节的空间。而以前名字只占用8个字节的空间，这是一个标志性的改变。尽管大小写敏感最终会实现，但就目前而言，当前版本的Win32不支持大小写敏感。NTFS文件系统中文件最大可达2^{64}字节，所有的文件指针均为64位宽。驱动器名，比如“C:”，用来指代磁盘分区或驱动器，如我们在19.3节中所见，Unix通过在根目录挂接新的卷，使之在现有的文

件系统中可以访问，来处理这种情况。从Windows NT 5.0起，NTFS也提供这种功能。驱动器可以分区，每个分区可以含有2^{64}个簇。簇的大小在格式化时指定，从512字节到64 KB，在重新格式化分区之前保持固定。所以最大文件大小为$2^{16}\times 2^{64}=2^{80}$。MFT（Master File Table，主文件表，参见图19-14）和MS-DOS中的目录起到的作用相仿，每个文件和子目录都在其中拥有一个数据项，保存名字和指向数据簇的指针，MFT及防备故障恢复的部分副本，保存在磁盘上单独的文件中。

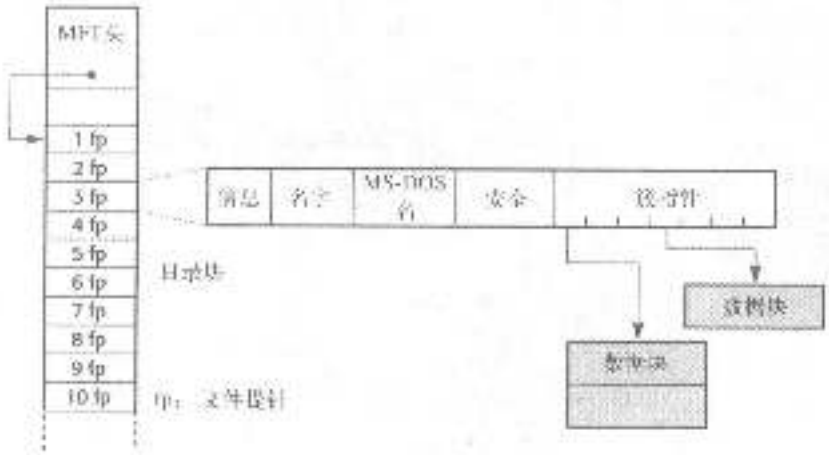


图19-14 Windows NT主文件表

目录被组织起来放置在靠近磁盘中心的位置，以降低磁头的移动。小于2 KB的文件可以存储在它们自己的目录记录中。为了提供更灵活的访问控制，每个文件和目录都可以与一个ACL（Access Control List，访问控制列表）关联起来，严格控制各个用户对文件或目录拥有何种类型的权限。

操作系统使用64位的文件引用号引用每个NTFS文件。文件引用号由48位文件标识符（用来定位MFT中的数据项）和16位序列号组成。后者是一个检验性的数字，每次MFT条目更改都会递增，这样，Windows NT可以对MFT中循环使用的位置进行一致性检查。

NTFS提供一种容错功能，在发生故障时能够对数据进行恢复。它顺序地驱动两个磁盘，如果一个发生故障，另一个可以立即替代前一个磁盘的工作。NTFS还维护一个事务日志文件，使用它可以回退最近的文件更动，纠正发生的任何错误。目录项的规格说明中一个不错的特性是，可以将小文件的数据存储在指针块中，这降低了管理小文件的开销。簇指针可以指向单独的簇，也可以指向一系列在磁盘上连续存储的簇。提供更紧密的安全控制是对Windows NT开发小组的一项重要需求。从开始时，访问控制列表就集成到文件管理中。

数据压缩和解压缩很程内建在读取和写入函数中，使用它可以节省磁盘空间，尤其在处理大型的图像或音频文件的情况下。

由于NTFS涉及额外的复杂性，因此，对于访问小于500 MB的文件，使用老式的FAT方案依旧是最快的方式。

19.5 RAID：更安全的磁盘子系统

商业服务器中使用的一项有用技术是磁盘镜像，两个独立的驱动器拥有同样的分区，这样它们可以从单个控制器接收同一命令，如果主文件系统由于某种原因报告错误，另一个磁盘上的副本可以立即供系统使用。理想情况下，用户应该能够完全察觉不到这种变更。但是，由于两个驱动器使用同一控制器，因此对错误的容忍程度是有限的。对于保护文件系统不受严重系统错误的影响，**RAID**（Redundant Array of Inexpensive Disks，廉价磁盘冗余阵列）是一种更健壮的方法。这种错误包括驱动器故障或控制器故障所导致的磁盘数据记录损坏。RAID实际上就是配置几个驱动器并

行工作，每个负责处理数据集的一部分。尽管在使用SCSI驱动器时，单个控制器可以驱动多个驱动器，但是这会降低系统的完整性。RAID错误检测和纠正通过为每块写入数据添加**ECC**（Error Correction Code，错误纠正代码），来完成恢复。即使整个磁盘驱动器发生故障，数据也不会丢失。RAID有时也表现为几个磁盘协同工作，作为单个“逻辑驱动器”。

这样还会加速数据的访问，因为所有的驱动器并行工作，读取或写入它们所负责的那部分数据，因而降低了数据传输的延迟。RAID可能涉及多个驱动器，以及独立的控制器，这样单点故障就不能造成不可挽回的数据丢失。RAID常见的配置有6种，每种适用于不同的情况。

如果加速数据传输是主要的问题，而非容错，可以选择RAID 0技术（见图19-15），它将一个数据块拆开写入到几个驱动器中，这就是**数据分段**（data striping），它遵循RAID并行地使用磁盘的方法，但一般并不包括ECC，以节省空间和加快速度。单个驱动器故障的处理与单磁盘系统相同。这种方法并不提高一致性，由于更加复杂，反而使可靠性有所变坏。

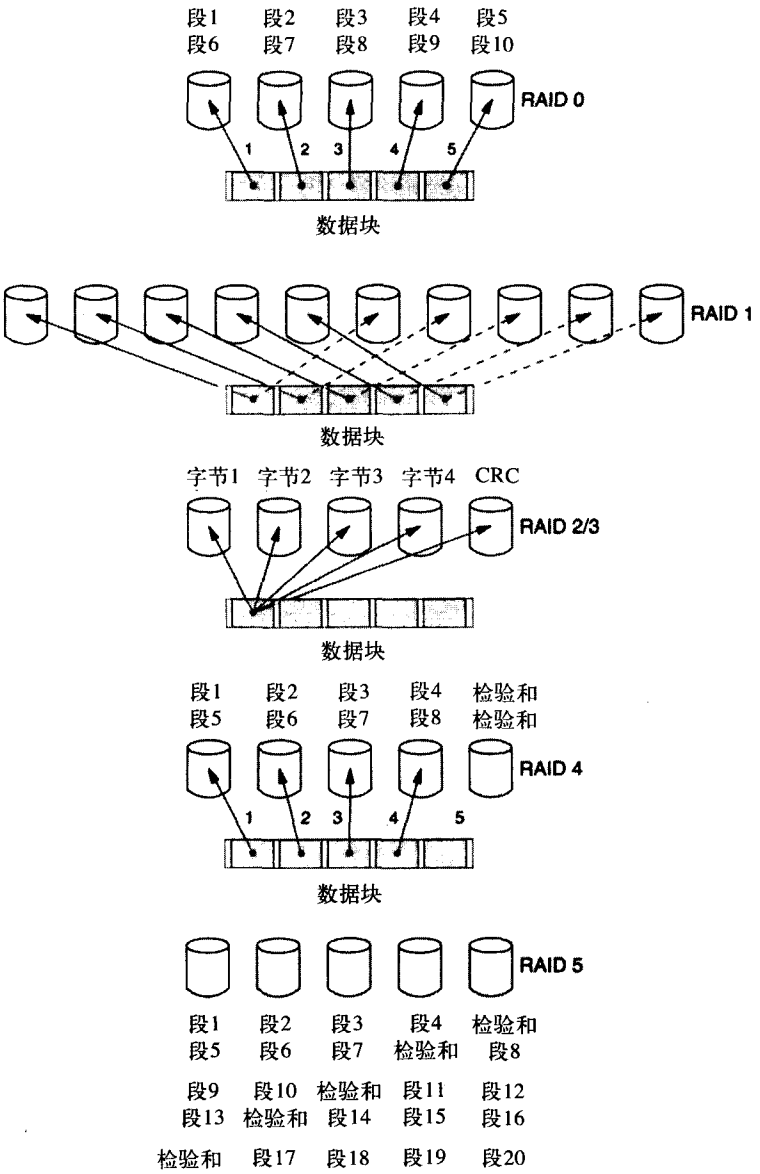


图19-15　RAID结构

RAID 1是磁盘镜像，简单地将磁盘驱动器的数量加倍，提供备份，以应对故障，但没有性能上的提高。

RAID 2和3是RAID 0的扩展。一个数据块被切分成几块，分布在几个驱动器中。不同之处在于划分的单位不是扇区，而是字，甚至字节。这会带来各种同步问题，因此实际中并不常用。RAID 3使用专门的驱动器负责奇偶校验。奇怪的是，这种单个位的奇偶校验不但能够检测错误，而且还能纠正错误，但仅当出错的驱动器彻底损坏（冒烟），告知系统发生错误时才能执行。

RAID 4和5通过使用更为复杂的ECC，提供完全的错误检测和数据恢复。它们之间的不同在于校验块的位置。RAID 4在单个驱动器上维护所有的检验和，因此在该驱动器上存在性能瓶颈，而RAID 5依次在每个驱动器上储存检验块，消除了这种流量问题。

19.6 文件安全：访问控制

Unix和Windows都使用密码来保护它们的资源，用户在登录时必须正确地输入它们。在Unix上，用户可以使用passwd或yppasswd命令改变这个密码。其他用户，只要认定属于三类用户之一，也可以获得对文件的部分访问权限，如表19-3所列。

表19-3 Unix文件和目录的访问控制选项

	读取	写入/删除	执行/连接
所有者	r-- --- ---	-w- --- ---	--x --- ---
组成员	--- r-- ---	----w- ---	--- --x ---
任何用户	--- --- r--	--- --- -w-	--- --- --x

访问权限只能在三个不同的特权级别进行分配：读取、写入/删除和执行。使用chmod命令，可以分别设定每个文件的所有者、组或任何人对文件所具有的权限。图19-16给出具体的应用。READ | WRITE | EXECUTE权限标志分别占用三个位，以十进制格式表现给用户。‘7’（111）给予所有的访问权限，‘0’（000）没有任何权限，而‘6’（110）给予READ和WRITE权限，但不包括EXECUTE。

```
rob@olveston [63] ls -al
-rwx------ 1 rob csstaff          280 Sep   5  1998 timezone
-rwx------ 1 rob csstaff           48 Sep  11  1999 tit
-rwx------ 1 rob csstaff          229 Jan  22  1999 to_arthur
-rwx------ 1 rob csstaff        25007 Apr   1  1999 unzipit
-rwx------ 1 rob csstaff          251 Sep   5  1998 vorc
-rwx------ 1 rob csstaff          243 Sep   5  1998 vorcorn
rob@olveston [64] chmod 666 unzipit
rob@olveston [65] ls -al unzipit
-rw-rw-rw-   1 rob csstaff      25007 Apr   1  1999 unzipit
rob@olveston [66] chmod 000 unzipit
rob@olveston [67] ./unzipit
 ./unziput: permission denied.
rob@olveston [68] ls -al unzipit
----------   1 rob csstaff      25007 Apr   1  1999 unzipit
rob@olveston [69] chmod 100 unzipit
rob@olveston [70] ./unzipit
rob@olveston [71]
rob@olveston [71] chmod 711 unzipit
rob@olveston [72] ls -al unzipit
-rwx--x--x   1 rob csstaff      25007 Apr   1  1999 unzipit
rob@olveston [73]
```

图19-16 Unix中设置文件的访问权限

用户可以使用groups命令查询所属的组（试着执行groups root，会得到十分有趣的结果）。但设立新的用户组需要超级用户权限，这是Unix的一个缺点。而Sun的ACL可以由常规用户操控，不需要专门的特权。相关的命令是setfacl和getfacl。使用它们可以将指定目录或文件的指定访问权限（rwx）赋予一组指定的用户。在使用组时，这样很有用。ACL功能的提供，是对之前功能的提高，尽管操作系统MULTICS和PRIMOS在20世纪70年代就提供了同样的功能。

出于管理的需要，超级用户，也叫做root，对任何事物都有完全的权限。由于保护root访问权限至关重要，因此，除常规的密码控制以外，系统还阻止从远程终端作为root登录，而Windows则直接禁用这个选项。对于一些商业组织来说，仅仅使用密码和权限方案显得功能十分有限。大型的站点中常常会有许多不同类型的应用程序用户，此时需要更细粒度的区分，以及更好的安全方法。为了满足这种需求，Microsoft和Sun都引入了访问控制列表（Access Control Lists，**ACL**）。ACL是经过核准的用户及其个人访问权限的清单，可以和任何文件或目录关联。不过，最初的Unix安全方法已经被专业程序员接受和使用多年，可能依旧会延续下去。一种被广泛采用的新方法是，由中心服务器跨网络提供密码。这样可以避免在每台工作站更改密码的麻烦。

Sun使用**NIS**（Network Information Services，网络信息服务）来管理网络主机间的密码验证。这种方法最初被称为YP（Yellow Page，黄页），现在依旧可以通过yp*命令，比如yppasswd，来改变一个用户在网络上所有主机中的密码。通过这种方式，中心主控密码文件由NIS服务器来维护并分发给其他的本地主机。在分发到网络上之前，任何密码修改都得先发送到NIS服务器进行检查和登记。

19.7　CD可移植文件系统：多个区段内容清单

可移动的CD-ROM和CD-RW碟片提供远大于软盘（1.4 MB）的容量（650 MB）。和往常一样，需要解决的问题是系统间的兼容性。各种操作系统如何读取CD-ROM的内容呢？这个问题在塔霍湖举行的一次会议上得到了解决。

光盘上的数据可以划分成session（区段）。CD-DA（Digital Audio）只需单个session，但为了索引的方便，CD-ROM常被划分成多个session。由于CD-RW光盘采用**ISO 9660**（High Sierra）标准，因此每个session必须连续刻录，不能间断，在盘片上登记每个session的管理开销是14 MB！这对音频轨道不算什么，但对于存储计算机数据而言，则浪费太多。CD-ROM一般有270 000个块，每个块保存2KB的数据以及报头和纠错码。16个字节的报头由12个字节的签名——标示块的开始，以及4个字节的标识构成。这个标识由三对BCD码构成：12-23-45，它们表示块的位置，分别为扇区、秒和分钟。就播放音频来讲，每秒钟75个扇区，60秒一分钟，一张盘片60分钟（实际上是70分钟）。第4个字节表示是否使用了Reed-Solomon ECC，这是一种十分高效的错误检测和纠正汉明码，长度为288字节。

1980年，Philips和Sony同意共同确定CD开发的标准。由于会议记录的封面为红色，因此这个标准被称为“红皮书”。从那时起，所有的标准都被赋予一种颜色加以区分。

最初的红皮书定义了音乐界使用的CD-DA唱片标准。红皮书CD最多可以拥有99个音轨，包括标题和唱片播放时间等数据。黄皮书规定了数据的标准，ISO 9660专门覆盖计算机相关的问题。绿皮书处理多媒体交互标准，针对机顶盒设备。CD-R和CD-RW由橙皮书来处理。还存在一些不那么知名的颜色和标准。

ISO-9660是国际公认的标准，它是一种专门的数据CD档案管理系统。它还被称做是CD逻辑格式，是一种组织CD上的文件和目录的方式，可以让任何操作系统访问。ISO-9660提供很好的兼容性，但不适合于记录零星的文件，因为它最初是针对工厂制造的一种只读格式，并非针对PC机上的刻录。另一种称为UDF（Universal Disc Format）的标准在CD刻录和DVD领域变得越来越流行，因为它对于可重写技术的支持更完善。

ISO 9660标准的Joliet扩展支持长文件名，用它来制作CD-ROM时，需要预先知道完整的最终内容。我们首先选择需要写出的文件，然后将它们连同目录（Table Of Content，TOC，定义了文件的数据在轨道何处）连续地写入到光盘上的轨道中。这些文件和TOC一同构成一个session。如果光盘上尚有剩余空间，之后我们还可以在上面写入新的session。新session的TOC会链接到之前的TOC。这种刻录方法不支持单文件的编辑或删除。因此，如果CD-RW盘片采用了ISO文件系统，删除文件时只能将所有文件都删除。因为这个原因，ISO文件系统一般只用在CD-R介质上。

结束session，连同TOC，大约需要14 MB的开销，因此，session越多，盘片的可用容量越小。

PC机上任何CD-ROM驱动器均能读取这种CD，在任何版本的Linux或Windows上，均不需要安装额外的软件。

UDF文件系统和ISO 9660存在较大的差异。CD-RW盘片首先被格式化成包，十分类似于磁盘上的扇区。将文件写到一系列包中的动作成为单独的操作，任何文件都能够单独地编辑和删除。在Windows XP中，借助UDF文件系统，用户可以像操作磁盘一样，将文件拖放到CD上。之前版本的Windows可以通过第三方的应用程序，如Direct CD，使用UDF文件系统。

Windows XP内建CD的刻录支持，用户可以选择文件，将它们拖放到CD驱动器的图标上，直接完成光盘的写入。尽管从用户的角度看，这些文件似乎是立即写入到光盘，但实际情况并非如此。实际上，这些文件被缓存起来，直到用户请求执行CD写入操作时，才真的写入到光盘中，前面选择的那些文件会以ISO 9660的方式写入到单个轨道中。

19.8 小结

- 将数据组织到文件中，并将它们存储到可以依赖的介质上，是操作系统功能的重要组成部分。
- 对归档数据的访问在逻辑上有几种不同的方式，包括数据库。
- 目录提供从逻辑文件名到首条数据记录磁盘位置的转换。为了确定其他数据块，需要使用FAT（文件分配表）或索引块。
- PC目录还保存每个文件的所有者、大小和访问权限等信息。
- PC曾使用过FAT-12、FAT-16，现在是FAT-32，以扩展其自身的容量，Windows NT提供另外一个更高级的技术：NTFS。
- Unix使用索引节点和索引块跟踪存储在磁盘上的数据块，并保存所有者关系及访问权限。
- Microsoft NTFS方案是为提供更佳的数据安全性而开发的，在磁头损坏的情况下，它能更好地保护数据的安全，同时提供更灵活的访问控制，以及对大文件更有效率的访问。
- 为了获得更好的数据安全性，我们可以将多个磁盘配置成RAID。通过记录冗余信息——数据重复或CRC编码，系统出现故障时的数据丢失可以恢复。
- CD-ROM需要使用独立的文件系统，以适用于各种各样不同的操作系统。软盘则需要主机能够识别它们的FAT-12文件系统。

实习作业

我们推荐的实习作业包括使用磁盘医生等磁盘工具查看软盘FAT文件系统。即使目录丢失或被破坏，这些工具也能恢复文件。对于使用FAT文件系统的硬盘，也可以执行类似的过程，但要冒较大的风险。

练习

1. 1 G的空间可以存储多少页的文字？
2. Norton Utilities是什么？什么时候最需要它们呢？
3. 列举基于FAT的文件系统的优点和缺点。
4. 磁盘数据的分布对应用程序（如Microsoft Word）的性能有什么影响？
5. 比较磁盘和半导体RAM的每字节价格。
6. 阅读任一微机操作系统对BIOS磁盘访问例程的基本系统调用。BIOS提供哪些功能呢？
7. 使用Sun Solaris（Unix）ACL机制，为项目组建立一个新的目录，使组中的每个成员都对存储自己工作的子目录拥有惟一的写权限，整个组可以读取任何目录或文件。另外建一个允许整个组写入的目录，将软件集合起来。如果需要，可以使用man setfacl命令。记着测试访问权限！
8. 使用Unix man yppasswd命令。本地机器的密码是不是可以不同于中心NIS服务器中的密码呢？为什么biscuit、caterpillar、splendid等单词不是好的密码呢？

9. 用户删除文件时，实际的信息是否被擦除了呢？什么时候擦除呢？
10. RAM盘和磁盘高速缓存有什么不同？RAM盘更好吗？

课外读物

- Heuring和Jordan（2004），对磁盘驱动器做了简短的介绍。
- Ritchie（1997），操作系统的简明介绍，第9章和第10章介绍档案管理系统。
- 查看PC杂志上磁盘的广告。
- 查看以下网站：

http://www.storagereview.com/guide/?\#topics

- Hex Workshop是一个不错的FAT扇区编辑器：

http://www.bpsoft.com

- CD技术和标准的介绍，参见：

http://www.pcguide.com/ref/cd/cdr.htm

- 这些网站可以通过本书的配套网站访问：

http://www.pearsoned.co.uk/williams

第20章　图形输出

当代计算机使用彩色图形屏幕和PostScript激光打印机提供常规的图形输出。这类设备要求的接口以及需要的各种数据格式和处理能力，对PC架构有深远的影响。不同设备制造商生产的设备之间的差异，如底层的接口和功能控制，大部分通过操作系统软件的各个层得以隐藏，不为用户和程序员所知。

20.1　计算机和图形：捕获、存储、处理和重现

在我们注视物体时，每个眼睛的透镜在眼球后部的视网膜上形成物体的倒像。视网膜是一个由大约1亿3千万个感光细胞组成的弯曲矩阵，每个感光细胞都会根据光的能量向大脑发送数字神经脉冲。因此，我们首先将世界看做是一对近乎相同的二维图像矩阵进行检测，接下来，视网膜和大脑内的神经网络执行特征检测，找出图像的边缘和表面。最终，我们看到一系列实心物体在三维世界中移动，我们能够识别出它们的名字，也能够用手触摸。计算机编码、处理和显示图像的方式有所不同。大部分打印机和显示设备，不管它们的输入如何，都只能提供二维的**像素**阵列供人类的眼睛查看。表20-1中列出这类图像的几种**格式**。

表20-1　不同显示类型的数据需求

显示	像素数据	图像文件大小（显示大小为1024×768像素）
全彩色	24位	2.25 MB
范围精简色	8位	0.75 MB
灰度单色	8位	0.75 MB
黑白单色	1位	96 KB

每种显示格式都有自己的优点和缺点。现在，我使用的显示器是高分辨率、21英寸的灰度单色监视器，由于受驱动它的计算机所限，它只能工作在黑白模式。在使用字处理软件时，没有什么问题；但对于图形建模或图像处理软件，这是不可接受的。显示全彩色版本的图像所需的数据量，可能是现在的24倍，我将不得不立即去就近的DRAM提供商那里购置设备。这样，我就很高兴能够不再受到红色的拼写检查警告的影响。

尽管一些早期图形显示终端的工作方式是根据设计好的序列在屏幕上画线，但这种**向量**显示方法已经完全为**位图光栅**（类似于电视）类型所取代，它制造成本较低，不但能够处理线段构成的图形，而且能够很容易地处理彩色图像。

CRT（Cathode Ray Tube，阴极射线管），如图20-1所示，依旧是大多数桌面终端的核心，尽管它尺寸较大，而且笨重。便携式电脑全部采用**LCD**（Liquid Crystal Display，液晶显示器），但为减轻重量、大小以及电力耗费，我们得支付额外的费用。随着LCD技术水平的提高，它们有可能在桌面市场上完全取代CRT。但十几年前，随着新型LED阵列的出现，人们也有类似的预言，但是到目前为止，CRT依旧出人意料地主宰电视和计算机终端市场。

我们需要稍微详尽地检查一下CRT，因为它处理图像数据的特殊方式不可避免地影响到PC的架构。想像一个巨大的抽成真空的玻璃瓶，内壁涂满特殊的发光漆（**荧光粉**），侧放在那里，一条加热线引入到盖子中，在瓶颈周围还有一些巨大的线圈，这就是CRT的基本结构。根据图20-1和图20-2，我们可以想像电子束在显示器内部来回**扫描**的情形。最初，读者可能会惊奇于其中为电子束，而非激光束；实际上，它根本不是光束。这是因为以这样的速率偏转光子十分困难：旋转反射镜方法已经试验过，但只取得了有限的成功。但在真空环境下，通过加热导线可以很容易地生成自由电子，然后可以使用电子线圈产生的磁场方便地偏转这些自由电子。如果拿一小块（一定要小！）磁铁靠近电视屏幕的玻璃，就会看到画面会受到磁场的干扰。磁场在电子束击打在前端玻璃屏幕的内壁之

面使之弯曲。仅当电子束击打荧屏屏幕内壁时，才会让屏幕的荧光涂层发光。这就是我们看到的屏幕发出的光。电视机的工作方式和计算机屏幕类似。但性能要求降低。RGB彩色显像管中必须有三种来自于可见光频谱不同部分的荧光，还必须有三个电子束分别扫描过每个荧光体。在这种方式下，白光是由三个荧光体同时发出相应色彩的光构成。每个像素都含有一些红、绿和黄荧光体，从而使之可以表现为任何希望的颜色。我们针对水平扫描线上的每个像素调制电子束的亮度（开或关）。这样，一个像素一个像素、一行行、一帧帧地生成图像，每秒60次。

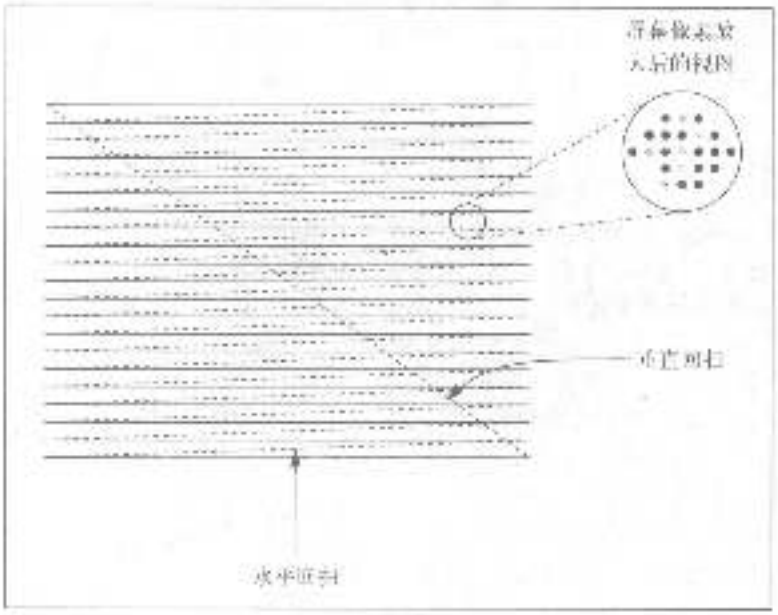


图20-1　位图光栅显示器

图20-2展示出金属荫罩如何阻止电子击打到错误的彩色荧光点。制作金属荫罩与荧光点之间排列的工艺，与用来制造硅芯片的投影光刻是同一类型。这再一次证明了只有应用更聪明的制造工艺，才能使好的概念成为现实！使用放大镜可以看到屏幕上的色点。但是，人的眼睛跟不上以60Hz变动的光线，因此，尽管以60Hz刷新的屏幕在16.67ms的周期内大部分时间黑暗，实际上我们感觉到屏幕连续地发光线，对于我的显示器（67Hz刷新），这个周期为14.9ms。

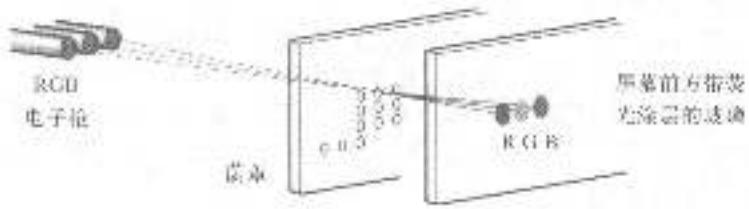


图20-2　彩色CRT内电子束通过荫罩照亮单个像素

实用电视机的水平扫描频率一般为15.75 kHz，帧率为60Hz。计算机监视器无论是水平扫描频率还是垂直扫描频率，都必须远快于电视机（见表20-2），才能降低闪烁效应对用户造成的影响，尤其是在环境光线较强的情况下。以这种速度显示图像需要高频驱动电路。PC监视器至少需要在30 MHz左右。制造商总是力图提供能够成为锁定在多个频率的多频扫描监视器。旧型号的监视器，有可能质量和大小都很好，但标准的图形卡有可能不支持它能够接受的同步频率。

表20-2 普通CRT的水平和垂直扫描频率

分辨率	垂直扫描频率（Hz）	水平扫描频率（kHz）
640×480	60	31.5
640×480	72	37.8
800×600	75	46.9
800×600	85	53.7
1024×768	75	60.0
1024×768	85	68.8
1152×864	85	77.6
1280×1024	75	80.0
1280×1024	85	91.2

从内存中读取图像数据驱动光栅扫描显示时，需要高带宽的访问，速率要足够快（30MHz），这样才能持续向一个电子束提供RGB亮度值。读者可以预先查看一下图20-9至图20-11，以及表20-3，了解一下常规的硬件配置。向屏幕输出单个像素的周期可以计算出来：

$$\frac{1}{60\times1024\times768}=21\text{ns/像素}$$

SRAM能够满足这个需求，而且乍看起来DRAM的访问时间似乎也能够胜任。从费用的角度来看，使用DRAM更好。但是，此处并没有考虑到几个负面因素。首先，DRAM需要时间刷新存储的数据，同时主处理器还需要频繁地访问存储单元以更新显示数据。在这些访问期间，屏幕数据的更新不得不暂时停止，以保证用户不会看到部分更新的图像。这是一种“临界数据”——资源需要供几个进程同时访问，因此我们需要采取9.5节中讨论过的保护措施。最后，DRAM的周期时间远大于其10ns的访问时间。对于大多数芯片，这个值约为100ns。这就暴露出一系列的问题。显卡制造商采用的解决方案是，在显存和屏幕驱动硬件之间安装一对快速的行缓冲区，它们交替工作，一个填充时另一个清空，轮流进行。这又是一例使用缓存加速连续读取的实例。此外，为克服内存周期延迟，可以配置多条内存模组，这样可以调整对内存的访问，在芯片完成充电之前不去再次读取其中存储的数据。通过采用更宽的数据总线，可以让内存一次提供16个像素的信息。这样可以降低所需的内存读取速率。

前面曾提到过，便携式计算机由于在重量、大小和电力供应上的限制，不能使用CRT。最常见的替代物是平板LCD屏幕（见图20-3）。它的结构为两个玻璃基板中间夹薄薄的一层（10μm）特殊的有机液体。这种液体含有较大的极化分子，它们能够以平行有序的方式聚集在一起（这也是“液晶”的由来）。这些分子会影响到通过的光线，根据光线相对于极化平面的方向，反射光线或者让光线透过。由于分子是极化的，因此可以使用电场来控制它们。

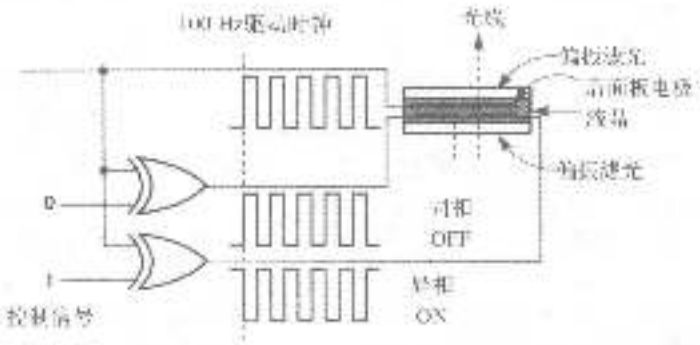


图20-3 液晶面板

为了利用这种物理特性，玻璃基板的内部沉淀有以一定模式排列的金属薄膜电极，通过它们，可以对有机液晶施加电场，也可以在容器表面上开槽，精确地控制分子的排列。分子依照凹槽排列，因此，如果凹槽精确平行，则分子的排列也会精确平行。在“扭转向列面板”中，前后玻璃面板上开槽的方向呈90°，这就强制液晶内的分子呈螺旋排列。

图20-4给出更多的物理细节。光线首先通过偏振器，然后透过玻璃，然后是金属薄膜，然后是基板的内表面，进入液晶。这些光线穿透液晶，以及另一面基板，在另一面可以看到。这种方案依赖于光的线性偏振。这也是实现太阳眼镜使用的效应。如果液晶内的分子感受到电场的存在，它们就会按照电场排列，此时，它们会影响到光波的透过，改变其偏振面。因此，如果前后玻璃面板都覆盖有偏振膜，且与基板的槽平行，相互呈90°排列，电场的作用与否就能够阻止或允许光波通过。它的原理就如同固态百叶窗。同样的原理还用在应急灯或背光LCD面板中。一个需要提及的技术细节是，电场必须以60 Hz到100 Hz的频率切换，以防止损坏液晶。因此，像素是通过切换前面板电压与背板电压同相或异相来控制的。这是XOR门的又一例应用（见图20-3）。LCD面板在制造时常常将像素标记为列和行的交叉点，复杂的驱动电路常常作为显示器件的一部分一同提供，并提供可以连接到计算机总线的直接接口。LCD基本上是数字设备，将来模拟VGA标准被废弃之后，与计算机存储设备之间的交互会变得极为简单。

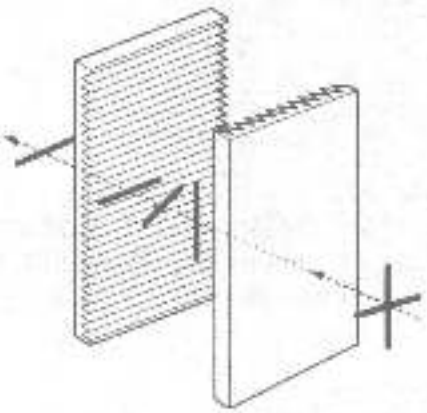


图20-4　扭转向列LCD面板

为了提高液晶面板的显示分辨率，ClearType字体采用一种比较聪明的做法，见图20-5。图中，左侧的插图给出我们面临的问题——使用相对较大的像素显示斜方条。每个像素由三个条带组成，分别产生红、绿和蓝。人眼对于色彩在空间上的变化不如对亮度在空间上的变化敏感，因此中间的图像——将斜方条表示为单色像素图，会出现锯齿。右侧是在子像素层面（RGB条带级）进行灰度控制的图像，其分辨率提供了三倍。这幅图看起来好多了。

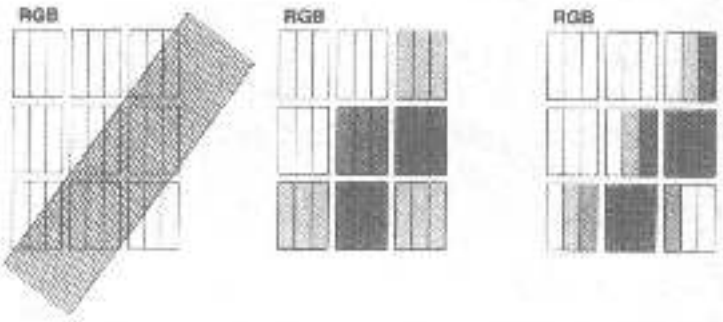


图20-5　为降低像素失真，针对LCD面板开发的ClearType子像素控制技术

现代操作系统和HLL的伟大成就之一，就是屏蔽了CRT和LCD之间巨大的差异。在程序员级别几乎已经完全不可见。所有不同的部分都由设备驱动程序和HLL库代码来隐藏，从而使得程序员在大部分情况下可以忽略任何功能上的差异。如果每次将字符发送到屏幕都需要考虑硬件的物理特性，软件将会变得十分复杂。

虽然显示器和打印机都生成位图图像供人们查看，但在计算机内部，使用简单位图格式的图片存在许多实际的缺点。数据的大小，即使经过恰当的压缩，依旧浪费大量磁盘空间，并且需要占用显卡上大量昂贵的RAM缓冲区。如表20-1所列，单个彩色SVGA（Super Video Graphics Adapter，

超级视频图形适配器）兼容的位图图像，未压缩时，需要2 MB。同时，不同的输出设备需要不同的位图格式，从而每次重新显示都得进行大量的重新处理工作。如果需要改变图像的大小，或提供缩放特性，位图也不是十分方便。

因为种种原因，在可能的情况下，应该尽量将图形数据以更高级的格式存储，即**抽象表示**，仅在需要显示给人看时才重新生成位图。我们已经了解到监视器屏幕如何以很高的速度清除位图的光栅模式，并在此过程中描绘新的点。激光打印机也生成位图，并将它们表示在纸张上。不管哪种情况，最初提供的图片都可能不是位图，尽管有时候我们所需的内容的确存储在位图文件中。一些常用来标识位图像素图像文件的扩展名是.IMG、.RAS、.PIC、.BMP、GIF和.JPG。如果需要通过照片扫描仪、Web站点或视频捕捉程序输入实际的图片，计算机都会先将它们作为大型的像数矩阵读入，然后存储成这类格式的文件。但从磁盘或Web站点读入线条图时并非如此，在这种情况下，图片有可能被表示为一系列的指令编码，正确地依据这些指令，就能够重新生成这个图片。实际上，本书大部分插图即采用这种方式。它们不是按照位图图像保存在磁盘上，而是根据十分精简的描述性脚本重新画出。事实上，我们可以说每个图示都是一个程序。插图的草稿用pic语言编写，最后的版本用PostScript语言创建。

pic中的脚本可以直接编写，或使用图表编辑器（如xfig）生成。但是，在显示之前，它们依旧必须转换成位图！图20-6给出生成图20-10草稿的代码。pic程序由解释器执行，它直接在位图矩阵中绘制图形。执行完这一步后，就可以立即将位图显示在屏幕上，或输出到打印机（实际情况下，这个过程会更复杂些，因为中间存在一个PostScript阶段，但原理是一致的）。所有的图形界面，比如Windows NT或X窗口系统，都使用类似的策略。我们将在20.6节中介绍Win32图形API，它的作用与pic解释程序类似，它允许程序员使用更方便的高级命令控制屏幕的显示。

```
.PS
define Driver { [line right 0.3 down 0.1
          line left 0.3 down 0.1; line up 0.2]}

Base:     box wid 6 ht 2 invis
          line up 2 from Base.sw
          line up 2 from Base.sw+0.4,0
          line right 0.4 from Base.w+0,0.4
          line right 0.4 from Base.w-0,0.4 """"image""data"below
          circle rad 0.02 at Base.w+0.2,0
          line up 0.5 from last circle; arrow right 2 "color#""" above
Palet:    box wid 1.2 ht 1
          ibox wid 1.2 ht 0.2 with .n at Palet.n "RGB Palette"
          ibox wid 1.2 ht 0.2 with .n at last box.s "table"
          ibox wid 1.2 ht 0.2 with .n at last box.s "R  G   B"
          {move up 0.1; line left 1.2}
          {move down 0.1
          line left 0.4; {line up 0.2}; line left 0.4
           {line up 0.2}; line left 0.4}
          line down 0.7 from Palet.s-0.3,0
          arrow right 1.2 "8 bits" below; [Driver]
           line right 0.2 from last [].e then up 0.25 then right 0.2
          box wid 0.6 ht 0.3 invis with .n at last [].s "DACs"
          line down 0.1 from Palet.s+0.3,0
          arrow right 0.6 "8 bits" below; [Driver]
           line right 0.2 from last [].e; line down 0.25; line right 0.2
          line down 0.4 from Palet.s
          arrow right 0.9 "8 bits" below; [Driver]
           line right 0.4 from last[].e then up 0.07 then right 0.4 then right 0.5 up
          line down 0.6 "CRT  "  rjust
          line left 0.5 up 0.2; line left 0.4; line up 0.07
          box wid 0.8 ht 0.2 at Base.w+1.1,0.2 "image base"
          box wid 0.8 ht 0.2 with .n at last box.s-0,0.1 "Y     X"
          {arrow <- right 0.2 from last box.e
            move down 0.1
             for i=1 to 2 do {
                line right 0.1 then up 0.2 then rigt 0.1 then down 0.2
              }
            line right 0.1
            }
          line from last box.n to last box.s
          arrow -> left 0.2 from last box.w
          box wid 0.6 ht 0.4 invis with .n at last box.s "image pointer"\
          "23" at Palet.nw+0,0.2; "0" at Palet.ne+0,0.2
          box wid 0.5 ht 0.3 invis "Memory" with .nw at Base.sw
.PE
```

图20-6 图20-10的pic脚本

20.2　PC图形接口卡：图形协处理器

曾经有一段时间，早期的微处理器程序员不得不使用回收的电报终端（ASR 33）输入程序代码并打印出程序的结果。这真是古老与现代的融合，依靠于缝纫机和干工打字机的技术来访问计算机提供的强劲动力！很快，人们就不能忍受这种方案，可以直接驱动CRT图像的新型图形控制器开发了出来，我们很快就意识到也可以将它们用做游戏，这样就可以较大地扩展PC机的用户。为了达到这种性能，含有图象数据的内存需要具备足够高的带宽以满足更新屏幕图像的要求。通常的RS232串行线根本不可能满足更新屏幕图像的需求。此时，人们决定将屏幕与微计算机的主要部件结合起来。Commodore PET、Atari ST和Apple Macintosh都在计算机主板上包括这种视频电路，而Apple II和IBM PC则提供大量的图形插卡。随着装备高分辨率彩色屏幕的个人微计算机的普及，将图形显示扩展到其他应用领域的机会来临。那个年代，使用大型计算机的重大困难之一，是类似于密码的命令行界面需要用户学习大量的缩略命令助记符，比如dir、chmod、cat和pip。基于这种原因，**WIMP**（Windows、Icons、Mice and Pointers）技术发展起来，这项技术的意义是支持图示风格的用户界面。但定期更新信息，显示大量图像的需求，对图形子系统造成很大的压力。如果新的窗口界面想取得成功，将图形计算和屏幕重画限制在2s内是最基本的要求。因为那些一直使用字符终端的计算机用户常常需要等待2s完成更新。

视频子系统的技术进步对交互式计算领域做出了重大贡献，线条图形、位图图像和文本都由图形适配器提供支持。之前，用户与计算机的交互依赖于基于字符的指令，用户需要使用键盘将它们键入到VDU中，这些VDU一般只提供2 KB的显示RAM，保存一屏能够显示的1920（24行×80个字符）个ASCII码，现代的SVGA图形卡最少都有4 MB的专用VRAM来处理显示位图，与之相比，许多VDU只提供字符字体！VRAM是双端口RAM，它支持两个连接，一个来自于主CPU，另一个连接到屏幕，效率更高。

如前所述，IBM PC规定了独立插卡的显示接口，这些插卡插入到PC/ISA扩展总线上。图20-7给出一个现代的版本。这个适配卡插在PCI插槽旁的**AGP**（Accelerated Graphics Port，图形加速接口）插槽中。这种将图形显示从主板中分离开来的硬件布局，既有优点也有缺点。尽管图形设备的升级变得相对容易，但屏幕的性能最初却受限于慢速的ISA总线传输速率（16 MB/s）。仅当图形适配卡上安装足够容纳完整屏幕图像的RAM的情况下，总线的限制才能得以克服。为了进一步降低这些访问速率限制，人们开发了独立的图形处理器，它们能够从主CPU接收指令，然后按照指令画线、填充方框，或是将图块或其他图形直接移到显存中。这种协作处理方式极大地提高了系统的整体性能，因为它减少了通过总线传输数据造成的延迟，同时将CPU解放出来，从事例行的底层任务。ISA总线瓶颈也被绕过，首先是采用更快的PCI总线，最近是AGP插槽，它可以在主存和图形卡间以266 MB/s的速度传输数据。各种各样的图形加速器不断涌现，性能各异。一般地，存在两类市场：运行Windows系统的商用机和运行游戏的家用机。

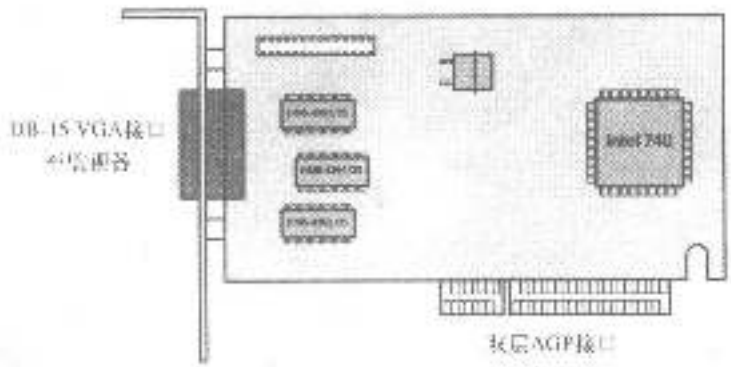
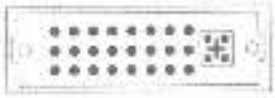


图20-7　AGP接口的SVGA图形适配卡

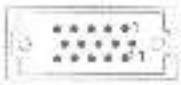
图形硬件性能的稳步提高在表20-3中列出。随着从数字到模拟信号传输的转变，连接监视器和图形适配卡的插座从老式的DB-9变为高密度的DB-15（见图20-8），这是为了让高频模拟信号从EGA卡传递到监视器。早期的卡只提供ON/OFF——图片的数字控制，这个系统基于之前的单色技术。随着新的DVI（Digital Video Interface，数字视频接口）和HDMI（High-Definition Multimedia Interface，高清多媒体接口）的出入，现在我们看到数字传输又回来了。对于全数字系统，这是一项很大的优点，尤其在使用LCD屏幕的情况下。在计算机内，数字首先被转换成模拟信号，进行传输，然后在显示设备内转换回数字形式，用于显示，这不可避免地会引入噪声，降低信号的质量。我们要解决的问题始终是发送SVGA图像所需的巨大带宽：1024×768×60×24=1.3 Gb/s，如果单独考虑红、绿和蓝三种颜色的通道，所需的带宽是340 Mb/s，需要三个独立的USB连接才能应付这项任务！光纤可以很容易地处理这种应用，但由于接口电路的成本问题，难以实现。

表20-3 PC屏幕显示标准的变革

单色	1981	文本模式，随着初始8088 PC一起提供	
Hercules graphics	1983	首个单色图形卡	720×348
CGA	1983	首个彩色（4色）图形卡，由IBM提供，如果强制为单色，水平分辨率加倍	320×200
EGA	1984	16色图形卡	640×350
VGA	1987	EGA兼容	
		16色（高分辨率）	
		256色	640×480
		256 K色（18位/像素）	320×200（CGA）
SVGA	1990	256色（8位/像素）	1024×768
	1995	24位真彩色（12平方/像素）	
XGA	1997	32 768色（15位）	1280×1024



DVI-I双标准监视器插座



DB-15 SVGA监视器插座

1	TMDS 数据2−	9	TMDS 数据1	17	TMDS 数据0−
2	TMDS 数据2+	10	TMDS 数据1+	18	TMDS 数据0+
3	TMDS 数据2/4 屏蔽	11	TMDS 数据1/3 屏蔽	19	TMDS 数据0/5 屏蔽
4	TMDS 数据4−	12	TMDS 数据3−	20	TMDS 数据5−
5	TMDS 数据4+	13	TMDS 数据3+	21	TMDS 数据5+
6	DDC时钟(SCL)	14	+5 V电源	22	TMDS时钟 屏蔽
7	DDC数据(SDA)	15	地	23	TMDS时钟+
8	模拟垂直同步	16	热或插检测	24	TMDS时钟−
C1	模拟红色分量				
C2	模拟绿色分量				
C3	模拟蓝色分量				
C4	模拟水平同步				
C5	模拟地返回				

29针DVI接头计算及信号名称

1	红分量
2	绿分量
3	蓝分量
4	
5	地
6	红分量回路
7	绿分量回路
8	蓝分量回路
9	控制引脚
10	同步回路
11	
12	监视器ID
13	水平同步
14	垂直同步
15	

图20-8 不同的监视器接口标准

DVI TMDS（Transition Minimized Differential Signal，最小化传输差分信号）是Silicon Image在20世纪90年代后期发布的，是一种用于从计算机向显示设备或从机顶盒到平板HDTV传输图像数据的纯数字化接口。由于用来降低传输比特率的编码技术是一项专利技术，对这项技术的推广造成不利因素，这可能是它未能在市场上广泛使用的原因之一。DVI链接中的重要组成部分是TMDS发

送器，它将数据编码并通过双绞差分线路串行传输到TMDS接收器。每台监视器都需要三个TMDS通道，分别传输RGB信息。每个编码器根据一位的控制数据或8位的像素数据编码产生10位的字。这些编码后的字连续地依次传输。前8位为编码后的数据，第9位标识编码方法，第10位用做DC线路的平衡。时间信号提供TMDS特征速率参考。因此，接收方可以根据输入的串行数据流产生正确的采样时间信号。

编码工作分为两个阶段，每次处理8个数据位，产生10位的字，在线路上表现为一系列电压的转变。在第一阶段，每个位与其前面位进行XOR或XNOR操作，第1位不做转换。编码器根据哪种方法在串行传输时产生更少的电压转换，决定使用XOR或XNOR，第9位表示采用哪种方式。在第二阶段，所有9位数据可能会被反相，第10位被添加进来，以平衡平均的DC电平。这种编码方式（8B10B）虽然看起来效率不高，但有助于将数字编码压缩到带宽规格有限的传输线路上。

20世纪80年代，监视器控制信号从数字到模拟的转变，使得我们无须增加针脚的数目就能提供更大范围的色彩和亮度。考虑一下，8位RGB色至少需要28个针脚（RGB：3×8、水平同步、垂直同步、地线、返回），或52个（如果需要独立的回路），这样的电缆太粗，不易于使用。将来，数字监视器（或许基于LCD技术）可能会使用光纤连接到主机，光纤可以很容易地提供30 MHz的带宽。另一种方式是将主板放在LCD面板后面，提供总线和LCD驱动电路间的直接连接。

在监视器屏幕上显示图片或字符，需要使用扫描电子束将数百万的荧光点激励起来，使它们放射光线。轰击哪个点，跳过哪个点，都由作为位图图像保存在显存中的屏幕数据决定。在电子束扫过屏幕时，它在图像数据的控制下不断打开/关闭。彩色屏幕需要三个荧光点和三个电子束。由于每个彩色电子枪都需要8位二进制数字来控制，因此每个像素需要存储24位数据。这也就是所谓的真彩色系统。它提供超过1 600万的不同颜色，需要大量的存储器。

视频硬件通过某个基址寄存器得知图像数据在存储器中存储的位置（参见图20-9），我们一般可以通过系统调用访问这个基址寄存器。如果要改变屏幕显示的内容，只需在显存构建一幅新的图像，并将基地址写到这个寄存器中。显示刷新硬件在垂直回扫期间会检查这个基址寄存器，这样就可以避免屏幕上出现部分的图像。

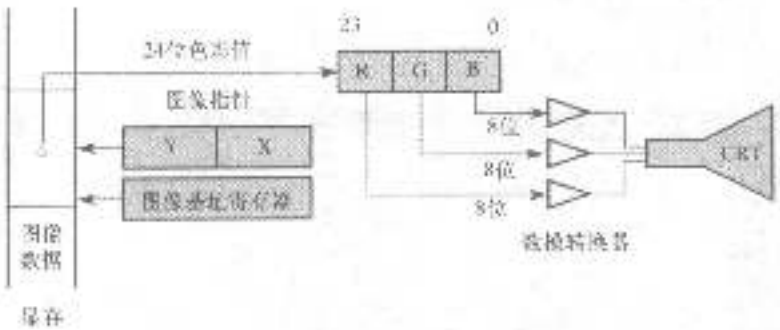


图20-9　真彩色屏幕

调色板常常用来降低所需的视频存储器的大小（参见图20-10）。这样就不用在显存中为每个像素保存24位的值。只要存储很短的数（4位）就能够满足要求。这个值用来查表，表中存储一系列的24位RGB数字。系统常常会为用户提供桌面工具，来调整这些RGB值，以满足用户的个人偏好。

三色控制线（RGB）承载模拟信号，这些模拟信号由**RAMDAC**产生。这些电路从图形存储器中快速连续读取下面的三个RGB二进制值，将它们转换成模拟形式，并将电压传送给监视器，控制电子束的强度。图片的亮度和颜色均通过这种方式来控制。

电子束的位置和强度由图形接口硬件而不是监视器控制，它必须控制电子束不脱离光栅。与此同时，从显示内存中读取数值，设置每个像素位置的亮度级别。为了完成这么复杂的任务，它使用高频率的振荡器生成水平和垂直同步脉冲，控制电路完成电子束的水平和垂直移动。频率除法器和计数器

芯片还提供并行读出操作，可以用来生成内存的地址，如图20.11所示。从而使电子束的位置与显示内存中的存储单元同步，RGB值就是从这些显示内存中读出指定给当前的像素。电子束在屏幕上一闪而过，只需63.5μs就能够输出每行的1000个像素。水平和垂直回扫线对于用户来说是不可见的，因为在这期间，电子束被关闭。一些图形控制器会向计算机发送信号脉冲，告诉计算机什么时候回扫抑制开始，CPU可以利用这些信息运行更新例程，比如移动光标或子图形。如果在正常的屏幕绘制期间做这些事，屏幕上就会出现各种各样的碎片和闪烁的颗粒。PC机中，IRQ2专为VBI（Vertical Blanking Interrupt，垂直消隐中断）保留，这个中断每16.7ms（60 Hz）发生一次。不是所有的软件驱动程序都使用这个功能，因此，为了防止在屏幕绘制期间改变图像而导致的闪烁效应，我们应该使用别的技术。

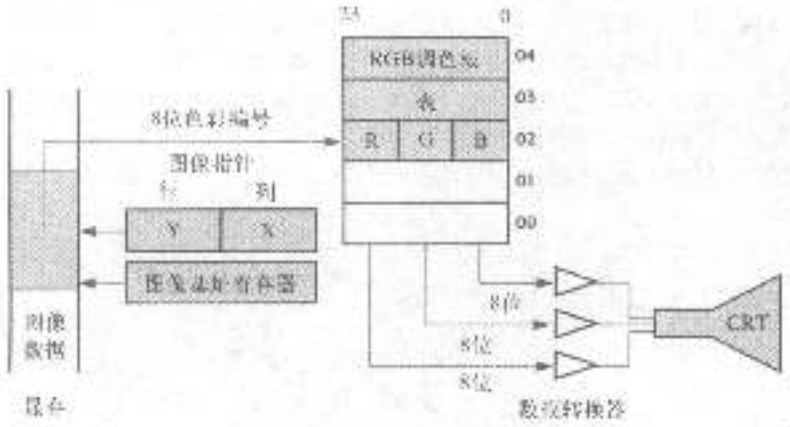


图20-10 使用调色板驱动PC屏幕

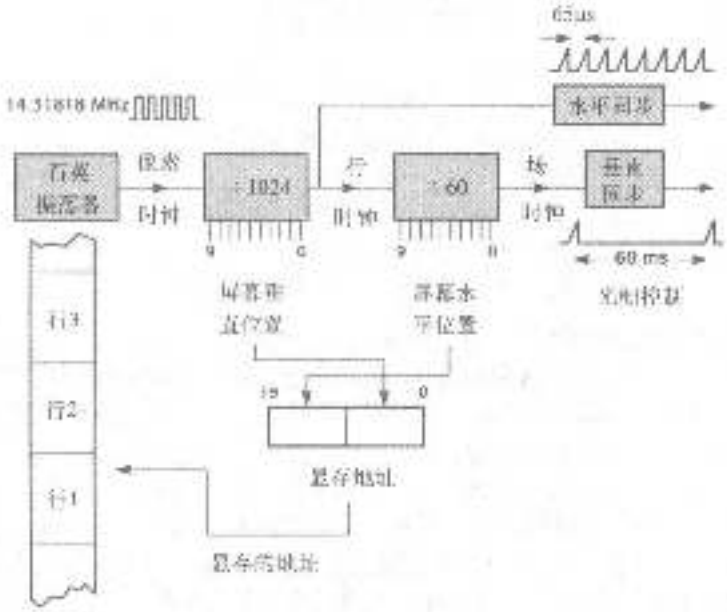


图20-11 屏幕光栅与显存访问的同步

20.3 激光打印机：机电一体化

当前使用的绝大部分打印机，都使用精细的垂直矩阵来生成每个字母的字形。不管使用哪种方法来完成，都使用磁性针脚击打色带，或者直接将精细的墨滴吹到纸上，或者使用扫描激光系统利

用静电原理将墨粉固定到滚动的磁鼓上，其基本原理都是相同的。

最简单的打印机在PROM中存储几套字体模型，可以直接使用文本文件的ASCII码作为存储地址，将对应的字模高效地读取出来。其他的字体则需要从主机下载到打印机中，然后使用它们将文本文件交付打印。或者，对于图形或图片打印机，可以发送位图图像文件。

激光打印机采用由光敏半导体制成的磁鼓（见图20-12），其材料可能为硒或非晶体硅。基本的物理过程和复印机完全相同，只是生成图像的光源不同。复印机将原始纸张的图像通过透镜投射到磁鼓上，但激光打印机使用扫描激光束绘出页面的图像。打印机首先在磁鼓表面喷射一层负电子（仅在表面暴露在强光之下时，这些电子才会消失或泄露）。由被打印页面的位图图像（需要事先在内存中生成）产生信号，调节激光二极管光源。廉价的激光打印机使用工作站的主存以节约成本，但打印时间会很长，同时在打印期间工作站被锁定。黑色墨粉通过小一些的磁鼓（转速为主磁鼓的两倍），从墨粉储料器供给主磁鼓。这个小磁鼓内部有一个磁铁，负责吸引墨粉微粒。同时还有一个除化的叶片，协助将墨粉均匀地分布在磁鼓的表面。负责散布墨粉的磁鼓向主磁鼓转动的过程中，墨粉微粒受静电的影响纷纷被吸到主磁鼓上，但仅限于那些激光束没有照射过的地方。两个磁鼓之间没有直接的联系。之后，墨粉颗粒通过接触转移到纸上，然后对纸张加热，使墨粉受热熔解，永久性地固定在纸张的表面上。

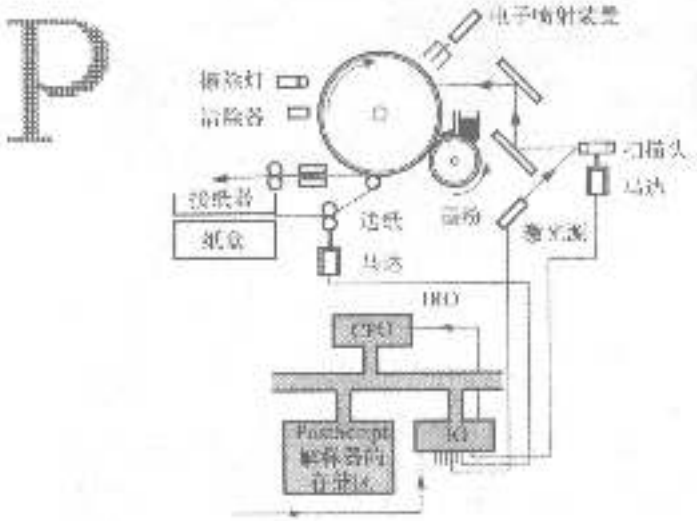


图20-12　激光打印机的示意图

相反的过程，即从纸面捕获图像，称为扫描。因为传真机在大批量生产方面的成功，图像扫描仪几年前价格突然下跌。现在它们一般避免采用旋转棱镜，转而采用集成光传感器阵列从纸张上拾取像素值。这种技术可以达到每英寸300－600的分辨率，扫描时，要么纸张经传感器阵列移动，要么平板扫描仪，纸张保持不动，阵列移动。

将A4幅面（210mm×297mm，或8.27in×11.69in）的页面转换成内存中的位图图像后，还可以使用压缩算法降低传输或磁盘归档的大小。在仅有文本字符的情况下，可以使用OCR（Optical Character Recognition，光学字符识别）软件对数据进行分析，将它从图像格式转换成ASCII码，这样做可以显著地降低数据的大小。对于600 dpi的图像（允许页边存在空白）：

$$\text{单幅A4页面的图像} = \frac{11 \times 7 \times 600 \times 600}{8} = 3.5\ \text{MB}$$

如果用ASCII来编码，同样的页面仅需要很少的数据即可表达：

A4页面上可以容纳的最大字符数 = 60 × 100 = 6000个字符

实际上，一页纸上平均的字符数接近2500，只需要2.5 KB的ASCII数据。因此，压缩比率将会达到：

$$\text{压缩比率} = \frac{2500}{3500000} = 0.0007$$

这样大幅度的缩小尺寸当然值得一试，但事实上OCR转换在最近才获得较大的提高，能够提供可以接受的速度和精确度。

就办公室应用而言，复印机、激光打印机和传真机的功能，现在正逐渐集成到单个机器中，这有效地降低了占用的空间，同时还可以向用户提供新的功能，比如文档扫描，从而可以开发出新的应用领域。

20.4 Adobe PostScript：页面描述语言

PostScript打印机的工作方式有所不同。**PostScript**是一种页面描述语言，由Adobe Systems公司提供，在激光打印机中广泛使用。发送到这些打印机的不是位图图像或ASCII文本文件，而是含有一系列绘图命令的程序。打印机含有处理器和存储解释器的PROM，它可以接收PostScript“程序”，执行它，输出一个位图图像供打印使用。PostScript程序总是解释执行，不经预编译环节，所以执行相当慢。并非所有的激光打印机都支持PostScript，但现在越来越普遍。打印机需要至少MC68030级别的处理器和16 MB的RAM。常见的配置为RISC处理器，128M RAM。有时，激光打印机会内置一个硬盘，以缓存输入的打印任务，并保存大量的字体词典。

图20-13中给出一段PostScript代码。它展示出这种语言的基本特征：反向波兰堆栈记法——操作数首先列出，后面是操作符。从图中第2行可以看到这种特征，它将绘画点移动到页面上的坐标位置(270,360)。指代页面位置的坐标系是常规的(x,y)图形系统，左下角为(0,0)，x轴水平，y轴垂直。基本单位是打印机的点，1/72英寸(353um)。PostScript代码的这些特征使得熟悉传统C或Java结构的程序员难以适应。Forth编程语言（曾用在血营早期的一些计算器中）使用类似的系统。类似于Forth，PostScript鼓励使用多层嵌套用户自定义过程。当然，程序员还需要知道开始新的线段需要命令newpath，填充线条时需要命令stroke，最后的showpage命令将图像绘制出来。图20-14给出一个更复杂的例子，它使用arc命令绘制弧线图形。要注意，使用fill可以为封闭的形状填充颜色。









图20-13 使用emacs和Ghostview进行PostScript开发

图20-14 更多的PostScript

依个人喜好的不同，有人可能喜欢直接使用PostScript语言编写例程，但更多的情况下，人们使用绘图软件包来生成这些代码。为了试验，可以将图20-13和图20-14中的代码输入到文件中，在前面插入报头%!PS-Adobe-3.0，之后，将文件直接发送给激光打印机。这个报头通知打印机，接下来的ASCII数据表示一个PostScript程序，应该按照PostScript进行解释。或者，如果机器上装有Ghostview实用程序，可以直接在屏幕上查看结果，这样可以显示更丰富的色彩。

通过软盘、电子邮件或FTP，能够以PostScript格式散发文件，提供一种新型的传真机制。我们可以在激光打印机上输出这类文件，但它的编辑困难且不方便：没有程序员愿意去应付由字处理软件包产生的PostScript代码。

这种类型的文档分发存在的一个常见问题是，源和目的地的字体并不完全等同。尽管PostScript支持设备无关，但在将文件传递到遥远的未知环境时，依旧存在困难。例如，如果文档需要一种特定风格和格式的字体，而最终的打印机上没有这种字体，此时就不得不用类似的字体来替代。这有可能会造成一些字母的宽度不恰当，甚至会使经过传输的文档与最初的版本产生相当大的差异。最直观的解决方案是，伴随每个PostScript文件传输完整的字体描述，但这样做可能会使文件增大20倍以上，显然不可接受。另一种刺激因素是，在线提供可供浏览的Web文档越来越常见。考虑到这些情况，Adobe引入了可移植文档格式（Portable Document Format，PDF）。PDF源于PostScript，但它随同文档提供字体，同时使用压缩技术降低文件的总大小。PDF文件采用多种压缩技术。文件中的数据使用LZW（Lempel-Ziv-Welch）压缩。这种复杂的数据压缩技术受专利保护，它广泛地用来保存和传播GIF图像文件。LZW通过字典或代码薄，用较短的代码数字替代原始数据中较长的位序列。PDF文件中的图像采用另外的JPEG压缩方法，进一步降低了文件的大小。

和PostScript一样，PDF也需要解释器，因此Adobe开发了新的浏览器——Adobe Acrobat，并免费供大家使用，可以用来阅读PDF文档。一般地，人们都从最近的Adobe网站下载Acrobat阅读器，然后用它来浏览PDF文档。

Adobe提供Distiller程序，它可以将PostScript转换成PDF。另一种选择是使用gnu Ghostscript程序，它提供许多命令行选项。下面的Unix命令行（在此拆分成两行）将PostScript格式的第16章稿件变成PDF：

```
gs -dNOPAUSE -dBATCH -r1200 -sDEVICE=pdfwrite -sOutputFile=ch_16.pdf ch_16.ps
```

图20-15中文件大小的变化值得思考。ASCII编码的文件（使用PC字处理软件的文本模式生成）为42 253字节，传输到Unix中在emacs下编辑后大小增长为52 114字节，PostScript版本猛地增长到1.2 MB，转换成PDF后，又回落到840 KB。切记：如果要打印PDF文件，必须使用Acrobat阅读器或Ghostscript将它再转换回PostScript。打印机尚不能直接接受这种格式（但可能仅仅是个时间问题！）。

```
rob[57]cat header ch_16|groff -t -e -p -fH>ch_16.ps
rob[58] gs -dNOPAUSE -dBATCH -r1200 -sDEVICE=pdfwrite -sOutputFile=ch_16.pdf ch_16.ps
....

rob[57]ls -al ch_16*
-rw-------     1 rwilliam csstaff 52114 Jul 1 08:42 ch_16
-rxw------     1 rwilliam csstaff 42253 Jun 28 13:26 ch_16.asc
-rw-------     1 rwilliam csstaff 840398 Jul 1 08:44 ch_16.pdf
-rw-------     1 rwilliam csstaff 1212385 Jul 1 08:43 ch_16.ps
-rw-------     1 rwilliam csstaff 22093 Feb 24 11:07 ch_16d
-rw-------     1 rwilliam csstaff 16437 Nov 5 1998 ch_16d~
-rw-------     1 rwilliam csstaff 23975 Jun 16 16:42 ch_16~
....
rob[58]acroread ch_16.pdf
....
rob[59]ghostview ch_16.ps
....

rob[60]
```

图20-15　比较文件的大小：ASCII、PostScript和PDF

需要注意的是，PDF代码比等价的PostScript要复杂得多。注意图20-16中列出的PDF代码，这段代码不过是打印常见的“Hello World!”欢迎词，将这些文本输入文件中，然后用Adobe Acrobat打开它，就会看到图20-17所示的消息。但要注意，Acrobat没容错能力，也不会提供什么有益的错误消息。

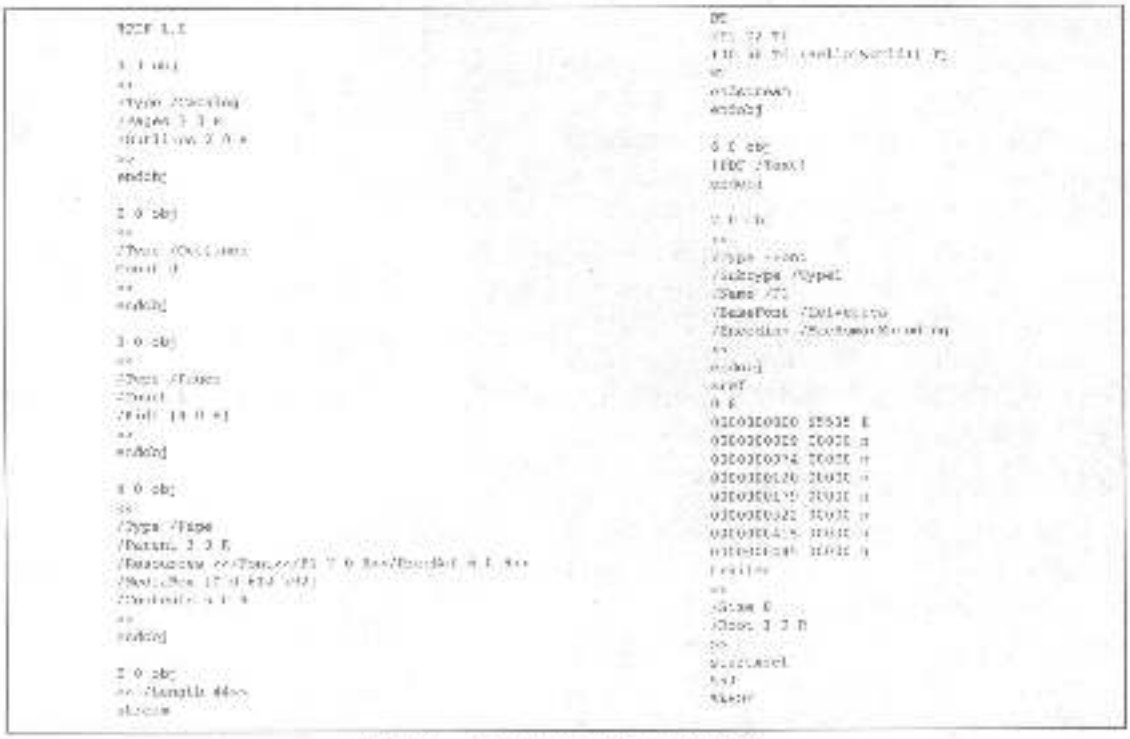


图20-16 “Hello World!”的PDF代码



图20-17 使用Adobe Acrobat浏览PDF文件

20.5 WIMP：重塑计算机的形象

自WIMP功能引入以来，用户与计算机交互的方式发生了根本性的变化。最初，屏幕为节省纸

张而生，用以更方便地进行全页面的编辑工作。现在，它已经成为人类和计算机交互的核心（参见图20-18）。人与计算机的交互依赖于屏幕完成输入和输出的即时显示，新的软件层已经开发出来，专门处理与图形相关的功能。为了跟踪活动的窗口、正确定向键盘输入以及恰当地响应鼠标按键点击，我们必须开发有别于硬件中断的新的事件处理技术。之前出现过的“虚拟桌面”模型，不足以描述现代窗口软件支持的动态交互，这就如同观看和参加网球比赛之间的差别。与系统如此紧密的交互，的确开始逐渐改变我们对事物的感知。

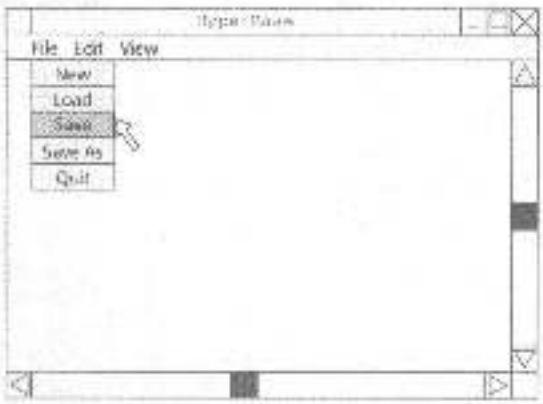


图20-18　典型的窗口布局

计算机交互的下一步可能会搁置键盘，采用语音识别软件。现在的软件已经能够提供相当的精确度，但需要进行较长的设置工作，它的普及可能还要假以时日。2D窗口格式（适合于手动操控设备和屏幕指针）是否能够容易地跟上这种形式的变化，依旧有待观察。不过，好像之前的“命令行”方案更适合于用户命令的语音输入。

20.6　Win32：图形API及其他

Win32是由一系列DLL（Dynamic Link Library，动态链接库）提供的API（Application Programming Interface，应用编程接口）。提供这些库的基本目标是使Windows编程更简单、更统一，使程序员更容易理解。图形函数现已在多个方面进行了扩展，远不止仅仅打开和关闭显示窗口，在本书的伴随网站上，提供函数名及其功能的庞大清单。提供它们是给读者一个起始的访问清单。要想获得更完整的信息，最好的办法是使用Microsoft Developer Studio提供的在线帮助。在需要的时候，可以在那里找到所有参数的列表以及操作上的约束。表20-4简短地汇总了操作系统提供的功能。

表20-4　Win32 API的功能

图形设备接口（GDI） 位图、图标和元文件 窗口创建 窗口操纵 屏幕菜单处理 鼠标和键盘事件的响应 对话框处理 时间事件
线程和进程调度 异常消息
空间内存管理 设备操纵 打印和文件输出 文件管理
通过剪贴板的数据交换 OLE/DDE数据交换
系统参数注册表管理 系统信息
DLL管理函数
网络访问例程
消息的传递和处理
音频数据管理

在第7章我们曾用到过Microsoft Developer Studio，当时是使用调试器在CPU寄存器层面检查代码的动作。现在我们需要进一步熟悉Win32提供的C语言库和函数。IDE（Integrated Development Environment，集成开发环境）提供的复杂功能是为了支持更加复杂的程序，相比之下，我们用它来试验的程序十分简单，但它对我们后面学习C++ Windows高级编程很有帮助，读者可以将之前的“Hello World!”程序重新编译一下，使之提供一个小小的窗口（见图20-19）。

如前所述，Win32函数库覆盖的范围很广。Microsoft不鼓励直接使用这些库，它推荐面向对象的MFC（Microsoft Foundation Classes，微软基本类库）。采用MFC后，程序员就完全置身于面向对象的方法、C++和向导代码生成器中。由于本课程是介绍性的，因此使用简单的C程序和Win32函数调用更清楚一些。



图20-19 第一个Windows应用程序

20.7 X窗口系统：分布式处理

Unix诞生的时候，还没有人想过需要图形用户界面。计算是一项专业的、有难度的技术能力，仅仅限于工程师或科学家尤儿的计算机科学家。社会在改变。现在，计算机已经成为一件平常的消费品。每个想要访问Internet的人都会需要它。这种巨大的变革来源于微处理器和WIMP的成功。Unix也开始配备WIMP界面。在20世纪80年代中期，MIT的研究人员开发了X窗口系统程序。当前版本（X-11）已经被移植到形形色色的操作系统。并不仅限于Unix，但作为Unix操作系统基本的窗口界面，它对Unix十分重要。

对于X窗口，一个有意思的特性是，一系列众多的通信协议占据至关重要的位置。这些协议都基于字符，它们将绘制所需内容的命令传达给屏幕或页面，这表示所有的屏幕都必须运行X解释器，类似于PostScript解释器，这种安排可以提供十分灵活的网络计算。实际的计算工作或数据库可以跨网络部署在任何地方，通过客户机-服务器协议与运行在工作站上的屏幕服务器交换遵循X协议的指令。即使是负责管理屏幕布局的窗口管理器，也可以在远程主机上运行。



图20-20 X窗口程序设计

X窗口编程可以在几个层面进行，如图20-20所示。基本的例程由X-Lib库提供，但对于大多数应用而言，它们太原始。再上一层是X-Intrinsics，再上为X Widgets，Widgets专为特定外观风格的应用程序设计，一般由商业产品Motif或OpenLook提供。

20.8 MMX技术：辅助图形计算

Intel为奔腾II增加了8个64位寄存器（见图20-21），以加速一些在图形程序设计中广泛使用的专门计算。它还在CPU指令集中加入约50条新指令，直接面向多媒体应用。这种性能增强的做法完全是CISC的方式！在这件事上依旧存在争议：这些更改真的有益吗？或者依赖于高级图形卡上提供的协处理器来完成这类专门的计算会更好？要知道，需要处理的图像实际上保存在插卡的视频RAM中，邻近协处理器，可以减少总线访问，提供第二个专门的处理器，与主CPU并行工作，应该可以极大地提高总的处理吞吐量。

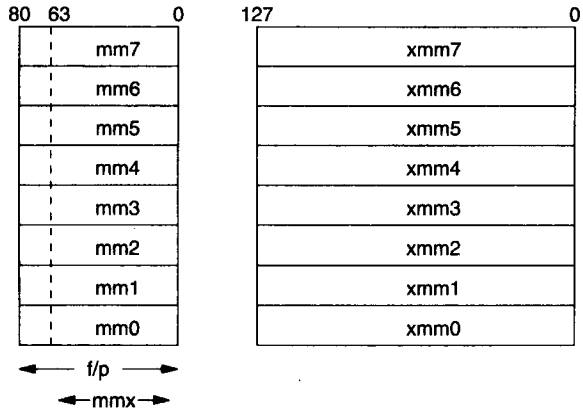


图20-21　MMX和SSE数据寄存器

额外的指令专为帮助矩阵运算而设计：单指令处理多项数据。这叫做**SIMD**（Single Instruction Multiple Data，单指令多数据）架构。由于它们显然主要集中处理数学运算，因此有人也将MMX（MultiMedia eXtension，多媒体扩展）称为矩阵数学扩展（Matrix Math eXtension）。这些指令只涉及整数运算，但奔腾FPU（Floating Point Unit，浮点单元）不幸卷入其中，因为实际上浮点寄存器和MMX寄存器均为一组寄存器，只是名称不同而已。因此，不要将新的MMX指令与FPU运算混在一起。

随着人们对于多媒体应用兴趣的持续增长，Intel引入SSE（Streaming SIMD Extensions，流式单指令多数据扩展）寄存器和指令。这些指令类似于MMX，但针对的是浮点数。Intel引入8个128位的新寄存器与这70个新的SIMD指令一同工作（见图20-21）。有了这套指令集后，我们可以使用单条指令同时处理四个浮点变量。

为了进一步加快处理速度，MMX奔腾的L1缓存增长到16 KB，指令流水线扩展到6段。这样必须修订译码电路，使之速度更快，允许更多的指令并行运行，而不会发生访问冲突。

20.9　小结

- 计算机的视频输出就是将大型（1024×768）二维像素矩阵以人眼能够接受的方式表示出来。它们常常以每秒60次的速度更新，在内存中，每个像素可能由24位数据表示。
- 当前，PC需要图形适配卡将RGB输入输出到模拟VDU监视器。随着LCD面板（本质上为数字化）的到来，情况可能会改变。
- 为了降低需要为每个像素保存的数据量，我们有时会使用调色板将有限的RGB数据保存在表中，然后使用对应每个像素的索引值来选择其中的RGB值。
- 激光打印机是光栅扫描设备，和VDU相似。它们一般会有强劲的微处理器，可以解释PostScript文件，绘制指定页面的图像。
- PostScript和PDF是专门的页面描述语言，主要供显示使用。它们提供灵活的图像放缩和设备无关性。
- Win32 API为程序员提供大量的函数调用，使用它们可以完成图形操作，以及其他诸多功能。
- X窗口系统是Unix上的窗口系统，它还提供分布式的客户机-服务器编程机制。在开发网络软件时，这一点十分有用。

实习作业

我们推荐的实习作业是熟悉PostScript语言。尽管我们很少会直接使用PostScript编程，但理解PostScript程序的结构（后缀波兰记法，基于堆栈的求值）有利于和常见的编程语言进行比较。

练习

1. 将PC图形控制器放在单独的插卡上，有哪些优点和缺点？
2. 使用字处理软件包生成一个PostScript文件（file.ps）。一般地，我们可以将打印文件存储在磁盘上，而不是发送给打印机，来生成该文件。在文本编辑器中查看file.ps，然后使用Ghostview查看。
3. 在Unix中，执行man gs，检查所有的标志选项。解释20.4节中的PostScript→PDF命令行。使用gs将刚才生成的PostScript文件转换成PDF，使用Acrobat阅读器查看它，比较PostScript和PDF文件的大小。
4. 解释LCD的工作原理。计算机如何影响光的亮度？LCD面板是模拟的还是数字的？常规的VGA监视器接口是否适用呢？
5. 说明下面四种情况下，字母在文件内的表示有什么不同：ASCII文本、PostScript文件、PDF文件、图像文件（BMP或IMG）。打印机如何在纸张上绘出字母？
6. 以这本书为例，估计一下它大概有多少字。如果将本书的文字输入ASCII文件，那么这个文件会多大？未经压缩的图像文件呢？
7. 调色板表是什么？它的值呢？VGA卡使用调色板表吗？
8. 显示同步问题是指什么？编写游戏的程序员为什么要考虑它？PC如何处理这个问题？
9. 15英寸显示器的最大分辨率为800×600像素。那么每英寸有多少个点呢？像素的大小是多少呢？
10. 单色VDU束只能通过开或关来控制，它是如何显示灰度的呢？
11. 找出你使用的部门激光打印机上，单个墨盒能送出多少张打印纸。然后估计每张纸平均分配到的墨粉量。这层薄薄的东西要花费多少钱呢？

课外读物

- Adobe Systems（1985）。
- PostScript入门导引：
  http://www.cs.indiana.edu/docproject/programming/postscript/postscript.html
- Adobe Systems公司的网站：
  http://www.adobe.com/
- Adobe Solutions Network的开发人员计划：
  http://partners.adobe.com/supportservice/devrelations/nonmember/acrosdk/docs.html
- 传奇性的W.R. Stephens写的几份有关排版和troff的文章：
  http://www.kohala.com/start
- 更多关于LCD显示技术的介绍，参见：
  http://www.pctechguide.com/07panels.htm
- 入门级的X窗口程序设计：Mansfield（1993）。
- X窗口的高级技术：X Window System系列，Vols 0-8，O'Reilly & Associates。详情请见http://unix.oreilly.com。
- Heuring和Jordan（2004），关于显示设备和打印机的相关内容。
- 这些网站可以通过本书的配套网站访问：
  http://www.pearsoned.co.uk/williams

第21章　RISC处理器：ARM和SPARC

计算机系统工程师长期遵循的路线，最近发生了翻天覆地的变化。RISC技术，设计相对简单但执行指令极为快速高效的处理器，能够大幅地提高计算机的性能。即使每条指令不如CISC指令完成的工作多，依旧可以获得性能的提高。采用这些简化的设计，RISC CU可以制作得更小，使CPU芯片上空出更多空间，供集成其他功能模块使用。为了最大限度地利用这种架构，一般需要开发专门的编译器，否则不能充分发挥出它的性能。嵌入式处理器——可以嵌入到客户设备的FPGA或专门的被控制器中——的市场十分巨大。我们将以ARM系列处理器为例展开介绍。ARM CPU也是现代RISC架构的典型实例，它提供的特性已经超越了表21-1中给出的初始的设计目标。

21.1　RISC的优点：更高的指令吞吐量

从计算机在20世纪50年代出现，直至20世纪80年代，CISC处理器的复杂度持续增长。一种通常的观点是：将功能移向计算机体系结构底层更快、更昂贵的核心资源，会获得更多的效能（见图21-1）。我们经常会见到，在应用程序代码中验证过的思想，之后被吸收到操作系统例程中，在操作系统中，它们被转换成微码以加快执行速度，最终在硬件逻辑器件中实现，以获得终极的效能。这种发展模式使得小型计算机领域内使用的处理器变得极度复杂，如DEC VAX-11或PRIME-950。设计人员紧跟制造技术的提高，将更多复杂性加入到ALU和CU中，但是，随HLL的不断发展并为程序员广泛采用，尽管每种新型机器都扩展大量功能，我们期望的性能上的提升却并不那么立竿见影。人们渐渐怀疑，简单地向底层硬件增加更复杂的电路可能不是正确的解决方案。举例来讲，在DEC VAX指令集中，有一条指令专门执行多项式求值，但是否真的有实际的用户程序使用过它，依旧是个未知数。DEC VAX和Motorola MC68000系列处理器可以看做是CISC处理器发展的顶峰。它们都有数百条指令、数十种寻址模式，指令异常复杂。需要复杂的能够使用微码编程的控制单元。

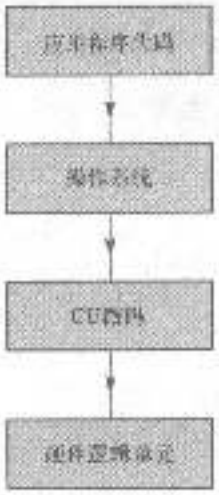


图21-1　功能体系

通过向底层转移功能提高系统吞吐量的做法，使我想起一个可能是杜撰的故事。一些系统程序员接收到加速公司数据库的任务。他们花费数周的时间研究日常的工作负载，使用最新的指令集剖析工具，

最终，他们设计出一种方案来确定最浪费时间的指令，以更流畅的方式重写了微码。一个月后，他们完成了工作，启动了新编码过的CPU。令他们惊奇的是，没有任何变化。什么地方不对劲呢？好像是操作系统内部有一些用来消耗时间的WHILE循环，不断地重复执行目标指令。向越来越精明的客户推销消耗时间的循环多么有效率，可能会很难——这对醉心于技术的工程师是一个警示。

在20世纪80年代早期，计算机设计人员开始质疑计算机开发的整个过程。根据对执行文件的分析，编译器经常使用的也就是指令集的20%，另外80%的指令很少使用，但这些指令在CPU中的存在不但影响到CPU的成本，还会带来性能上的损失。奇怪的是，Motorola在1970年左右做出的一项类似的调查，被作为进一步加大MC68000的指令集的证据！据我推想，研究课题有时的确是这样，进一步的研究表明，使用经过恰当调整的程序，简单的RISC架构最起码可以执行得和CISC架构一样好。根据直觉，简单的指令不会在更复杂的CPU上运行得慢，同时也不会需要更多的开发和实现开销。回顾计算机发展的历史，在硬件中提供越来越复杂的操作的动力，可能部分归因于基准性能表的销售效应。如果增强一条指令就可以提高某项基准测试的表现，这就会促使制造商采用CISC方

式执行。遗憾的是，对于用户而言，基准测试答案和桌面计算机中CPU需要处理的程序的日常执行没有什么太大的关系。

现实呼唤另一种实现方式，基于多种原因——我们随后将会一一查看——绝大部分制造商在他们将来的处理器中采用RISC方式（见图21-2）。根据这项调查发展起来的新一代革命性的架构，被称为是**RISC**（Reduced Instruction Set Computing，精简指令集计算），它反映出机器指令数目的减少，但这并不是严格的定义。区分RISC和CISC，要根据表21-1中列出的几项指导性原则，还要提一下的是，奔腾处理器是CISC设计，但做了一些RISC修正，以提高性能。

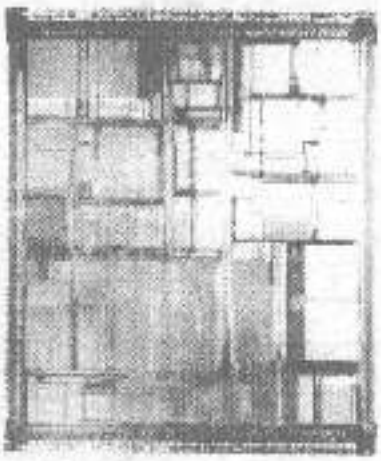


图21-2 Sun Microsystem的UltraSPARC II

表21-1 RISC CPU的基本特征

1. 单一长度的指令代码
2. 单个时钟周期执行时间
3. 支持有限的简单寻址模式
4. 提供大量的CPU寄存器
5. 提供有限的机器指令
6. 只支持简单寻址模式
7. 对浮点处理的硬件支持
8. 不支持结构化的数据类型
9. 提供编译器以支持这种架构
10. 硬件CU，流水线译码
11. 简化的中断机制

表21-1中的前两条原则，可以让RISC架构的运算和操作更小、更简单。这样有几个优点。在CPU芯片上省出的空间内，我们可以安装其他部件，比如高速缓存。由于紧邻CPU，因此能够获得更好的性能。由于与CPU之间的传输延迟得到降低，因此这些部件的运行速度得到极大的提升。通过让机器代码长度唯一，我们可以完全等同地处理所有指令。同时也表示可以用硬件连线的译码电路，来取代复杂性日益失控的微码技术。或许最重要的是，固定的指令大小和译码时间，使流水线并行译码技术成为可能。和其他方面相比，这个方面使RISC速度超过CISC。这是一种在基本顺序执行的环境中实现并行活动的方式。

通过在CPU中加入更多的寄存器——或许利用微码存储释放的空间，CPU可以将变量读入寄存器中，并保持更长时间，直到更多的变量到来，需要让出空间为止。这样程序访问存储在内存中的变量的请求会减少——这是关键性的系统瓶颈之一，导致RISC革命的研究还揭示出许多CISC编译器的优化是多么不充分。掌握这个教训之后，RISC设计人员开始制作定制的编译器，充分利用指定的CPU的潜能，这又需要汇编级别的编程。

当初在CISC CU中使用微码，是因为当时ROM介储器比磁性的核心存储器介面且速度更快。因此，在设计计算机时，使程序长度最小化，让每条指令执行的活动最大化，对每个人都有利。现在这种对比已经逆转，DRAM芯片的访问时间比微码存储器中使用的高密度PROM更快。对于快速的硬件，EPLD和FPGA是首选的实现，而非PROM。RAM价格低廉，而且容量巨大，程序的长度不再是需要考虑的问题。

现代CPU的速度（见图21-3）很容易就超过最快的内存周期频率。在过去的25年间，CPU的时钟频率从1 MHz提升到2 GHz，这是一项令人惊异的成就。但是，正是这种提高，使得CPU不得不在所有的内存访问总线周期内插入WAIT状态来浪费时间，暂时将CPU挂起，直到下一条指令或数据从内



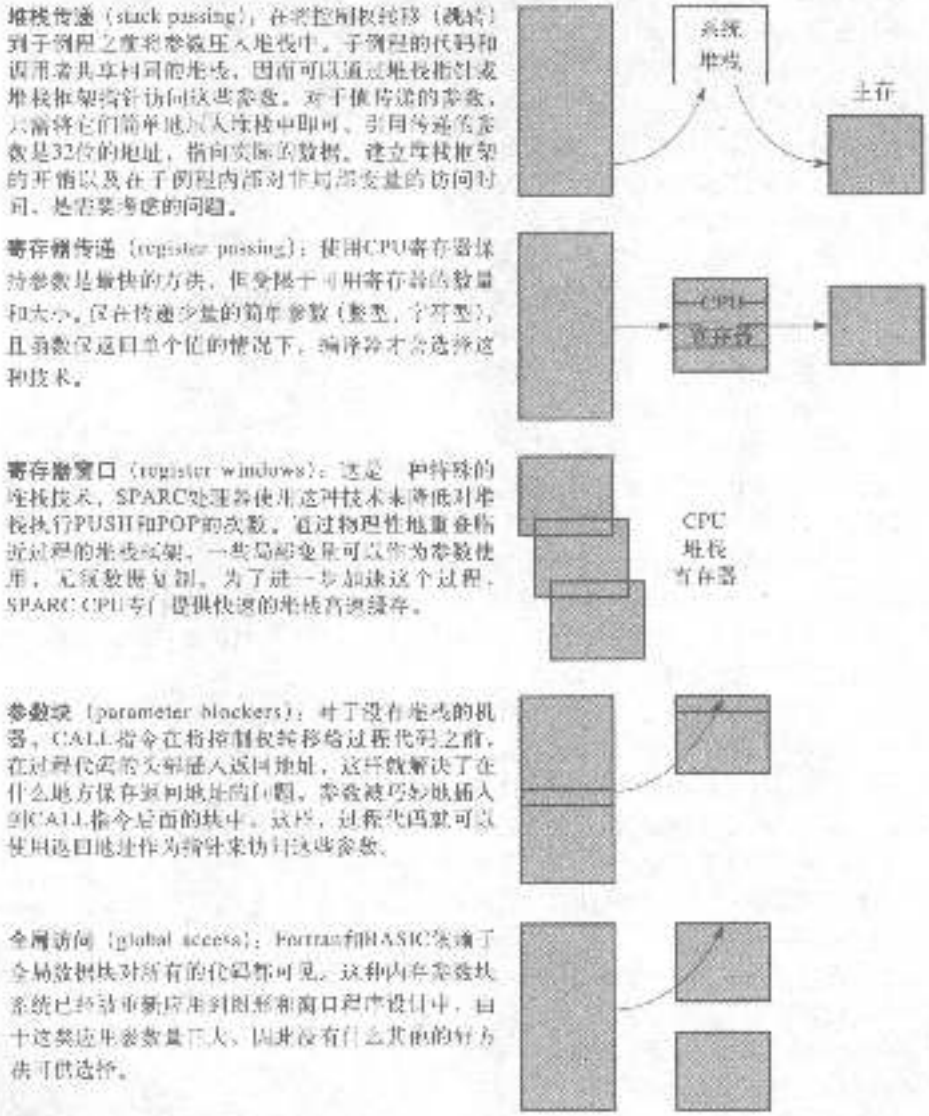
图21-3 微处理器日益增长的时钟速度

存中返回为止。更快的面向页或面向行的内存访问技术，可以通过突发性地传送指令流来缓解这个问题，但这时CPU就需要指令预取缓冲区，或本地高速缓存。小型RISC CU的优点再次体现出来。

在CPU芯片上，CU的物理大小是一个重要的因素。由于微码PROM占用了相对较大的硅晶体表面，从而使得其他合适的功能不能放置在芯片上临近ALU的优先位置。当信号离开芯片，传递到印制电路板上时，由于PCB布线日益增长的容抗，它们都会发生附加的传输延迟。RISC设计理念的主要优势，就是用硬件实现译码，省出更多表面，从而可以将高速缓存、MMU和FPU集成在芯片上。现在人们已经认识到，微码驱动的CU浪费大量的硅表面用于支持复杂的指令，而在实际的程序中，很少使用这些指令。

由于DRAM制造成本的下降以及访问速度的提升，最初两个推动力之一已经消失，即尽量少使用RAM，多使用PROM。微处理器重演了早期计算机的发展路线，现在我们需要从其他设备即高速缓存、MMU和FPU中获得更好的性能。空间允许的情况下，最好是将它们都集成到CPU芯片内。

CISC计算机常用的方法是使用系统堆栈将参数传递给函数。实际应用中，这种方法已经得到广泛的接受，以至于图21-4所示的另一种方法几乎被人遗忘。随RISC计算机引入，我们需要对这种使用



堆栈传递（stack passing）：在将控制权转移（跳转）到子例程之前将参数压入堆栈中。子例程的代码和调用者共享相同的堆栈，因而可以通过堆栈指针或堆栈框架指针访问这些参数。对于值传递的参数，只需将它们简单地压入堆栈中即可。引用传递的参数是32位的地址，指向实际的数据。建立堆栈框架的开销以及在子例程内部对非局部变量的访问时间，是需要考虑的问题。

寄存器传递（register passing）：使用CPU寄存器保持参数是最快的方法，但受限于可用寄存器的数量和大小。仅在传递少量的简单参数（整型、字符型），且函数仅返回单个值的情况下，编译器才会选择这种技术。

寄存器窗口（register windows）：这是一种特殊的堆栈技术，SPARC处理器使用这种技术来降低对堆栈执行PUSH和POP的次数。通过物理性地重叠临近过程的堆栈框架，一些局部变量可以作为参数使用，无须数据复制。为了进一步加速这个过程，SPARC CPU专门提供快速的堆栈高速缓存。

参数块（parameter blockers）：对于没有堆栈的机器，CALL指令在将控制权转移给过程代码之前，在过程代码的头部插入返回地址，这样就解决了在什么地方保存返回地址的问题。参数被巧妙地插入到CALL指令后面的块中。这样，过程代码就可以使用返回地址作为指针来访问这些参数。

全局访问（global access）：Fortran和BASIC依赖于全局数据块对所有的代码都可见。这种内存参数块系统已经被重新应用到图形和窗口程序设计中，由于这类应用参数量巨大，因此没有什么其他的好方法可供选择。

图21-4　各种参数传递方法

堆栈进行参数传递的方法研究推敲一下，因为采用这种方法时，大量的时间耗费在压入和弹出参数和返回地址上。RISC的任务之一就是降低内存访问的频率，这显然与现代软件对堆栈操作的热衷发生直接冲突。Intel安装快速的L1数据缓存来应对这种情况，而Sun在CPU芯片上提供专门的堆栈高速缓存。这样，面向堆栈的处理所具有的方便性和灵活性得以保持，但不会导致频繁的内存访问。ARM处理器提供一个链接寄存器，存放函数的返回地址。在不涉及堆栈操作，没有发生子例程的嵌套调用的情况下，这种方式工作得很好。

21.2　流水线技术：更多的并行操作

指令的并行执行或重叠执行，长期以来一直被认为是提高性能的途径。CISC变长的操作码以及不同的执行时间，使得这种方案难以实现。如果同时运行几条指令，而且它们在不同的时间结束，那么在协调CPU使之能够继续执行程序之前，首先需要克服重新同步的问题。并行指令执行有几种方法可以采用，到目前为止，最富有成效的方法是使用**标量流水线**（scalar pipeline）技术的指令级别的并行（Instruction Level Parallelism，ILP），使用这种方法时，多条指令沿多级处理线路传递。每个系统周期内，都有一条指令执行完毕，代码离开流水线。这样，尽管指令周期依旧需要几个（比如五个）阶段才能完成，但在流水线满负荷运行时，就如同读取-执行周期速度快了五倍。

图21-5给出5级流水线的示意图。其中，指令沿处理流水线传递，5个时钟周期后完成。

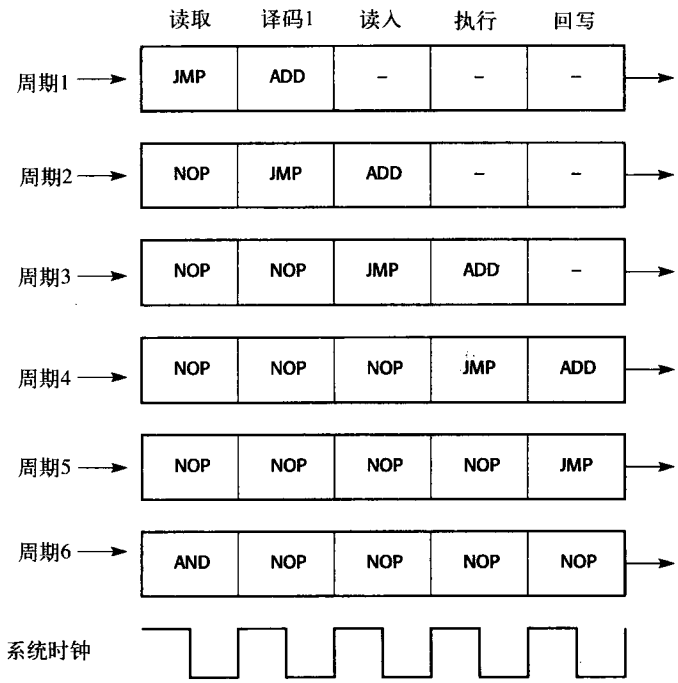


图21-5　多级流水线译码——并行处理

流水线译码本身没有什么问题。但在遇到条件分支指令时，如JZ或JNZ，指令预取器不得不预测程序会执行哪个分支。条件求值的结果，可能直到指令到达流水线的第三或第四级时才能得出。SPARC架构允许延迟分支选择或预执行，协助处理这种困境。仅当分支被选择时，其代码才有效，如果另一个分支被选中，则当前流水线的内容必须清空，然后重新访问内存，获得所需的指令。流水线越深，这个过程耗费的时间越长。

另一个难以处理的问题是相邻指令依赖于同一变量的情况。这种情况下，指令的执行将会停止，因为前一条指令的结果尚未从流水线中出来。这种事件会使译码器陷入混乱，浪费大量的时间。一

些RISC编译器通过插入NOP指令隔开对变量的引用，避开这类“流水线互锁”延迟的发生。这和系统总线上的WAIT状态相仿——延迟直至行动迟缓的部件跟上为止。另一种不同的解决方案是，由编译器将程序中的指令重新排序，以避免这类冲突。但这个选项仅在有限的情况下有效。

指令并行执行的负面影响是需要更多的加法单元。这是必需的，因为几个同时执行的流水线偶尔会同时需要加法器，以继续不受阻碍地执行，例如，操作数相加或计算地址指针。另外，还很有可能需要额外的内部总线，这样流水线的不同阶段才能同时进行数据传输，不会互相影响。

21.3　超标量方法：并行的并行

超标量（superscalar）一词用来描述那些拥有多个并行执行单元，能够在每个时钟周期内完成多条指令的CPU。通过为CPU装备几条独立的译码器（一般采用流水线类型），可以达成上述特性。奔腾和SPARC处理器采用的就是这种方式。要注意，指令流依旧只有一个，只有程序要求的本质上可以并行执行的操作，可以得益于在独立的硬件单元中同时执行几条指令的功能。尽管程序在编写时为一系列有序的指令清单，但是，一般在逻辑上并不要求指令必须以某种特定的顺序执行。程序中的指令并非一定能够并行运行，因为它们常常会依赖于早期操作的结果。因此，超标量CPU，和单流水线译码器一样，为了发挥性能上的优势，需要检测程序内指令之间的相关性。编译器在编译程序时所做的准备性工作，对这种预测十分有益，但当前大部分还是依赖于CU中的电路，来检测邻近指令之间**数据、控制和资源的相关性**。如果几条指令之间不存在互相依赖，则没有理由不并行执行，这就是超标量处理器希望达成的目标。

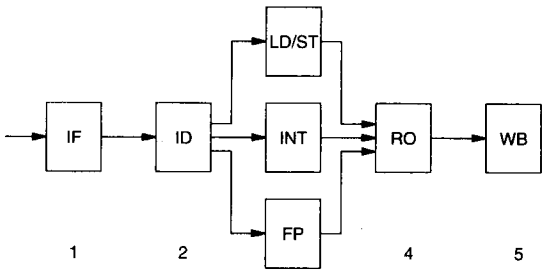


图21-6　含三个执行单元的5级超标量架构

从图21-6中，我们可以很容易地看出，如果设计者想传递足够多的部分译码的指令到第3级，保持所有三条执行单元忙碌，会遇到什么样的问题。为了让整型指令能够赶上前面的浮点操作，FP单元花费的时间比INT单元要长，这种情况下应该怎么做呢？标量（流水线）和超标量（同时执行）并行处理都必须处理指令之间的依赖性。这些依赖关系在表21-2中分类列出。避免这个问题的一种方式是，依赖于增强型的编译器来检测，并通过重新排列程序内的指令，来避免这类互锁依赖关系。尽管这似乎是一种理想的解决方案，但这样做需要得到新的编译器，每次处理器升级，都得重新编译所有的可执行代码——这种情况迫使Intel保持奔腾处理器与上一代80x86二进制兼容。可能正是因为这种策略，奔腾CPU使用专门的硬件单元来处理指令依赖关系。

表21-2　指令依赖关系

依赖类型	说　明	例　子	解除条件
数据依赖	RAW（read after write），写后读	MOV EAX, 10	
		ADD EBX, EAX	EAX已载入
	WAR（write after read），读后写	MOV EBX, EAX	
		MOV EAX, 10	EAX已读
	WAW（write after write），写后写	MUL 100	顺序
		ADD EAX, 10	正确
控制依赖	前一条指令的结果对指令的完成至关重要	CMP AL, 'q'	
		JZ exit	Z标志置位
资源依赖	硬件资源的有限性	浮点运算	单元空闲

21.4　寄存器存储：更多的CPU寄存器

由于放弃微码，CU的复杂性得到简化，使用节省出来的空间，可以集成大量的芯片内存储空

间。它们部分成为快速的L1高速缓存，部分用于增加CPU寄存器的数量，所有这些更改都是为了降低对主存的访问频率。为了进一步反映这种意图，RISC处理器的指令集都经过调整，遵照LOAD-STORE的思路进行定义。只有少数的指令可以访问主存，常规的访问均被定向到CPU的寄存器。

现代CPU中一种比较常见的做法是，将它们所有的寄存器组织成一块多端口RAM，称为寄存器文件。芯片上寄存器文件所占的空间，大约和访问端口数的平方成比例增长。这也意味着，寄存器文件能够支持的端口数量在实践中不可能没有限制地增长。这是CPU通常使用单独的寄存器文件存储整型和浮点数的原因之一。由于每种类型的运算使用不同的ALU，将多个整数、浮点数甚至向量/MMX执行单元连接到单个寄存器文件，会导致电路过度臃肿。

处理器，如Sun SPARC和Intel奔腾，都是超标量处理器。因为它们都能支持几条并行流水线独立地译码和执行。由于它们使用多端口寄存器文件，因此在实际应用中，流水线的数目存在限制，当前，将访问端口扩展到20以上，在技术上存在较大的难度。由于每个流水线译码器需要至少二个端口，实用的流水线数目存在一个最大限度。比较图7-11和图21-7，可以清晰地看出流水线处理器中，标量和超标量布局上的不同。

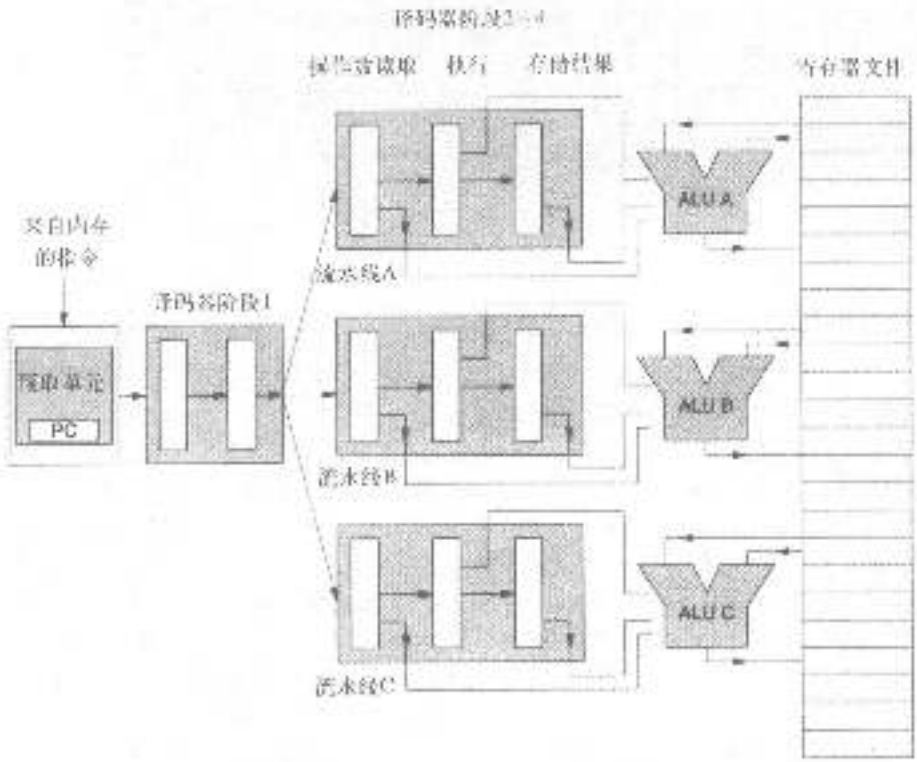


图21-7 超标量处理器中寄存器文件存储的使用

现在，一些处理器在每个流水线的头部插入额外的缓冲区，这些缓冲区用来延迟任何受到依赖关系直接影响的指令的处理，这类临时性推迟指令执行的技术称做搁置（shelving）。奔腾Pro和SPARC64处理器均实现了这种技术。

一些新型的处理器（AMD Athlon和奔腾4）使用一种十分出人意外的寄存器文件机制——**寄存器重命名**（register renaming）。这种技术允许编译器与CPU紧密协作，使用程序员不可见的临时寄存器保存变量。这种技术可以避免指令间一些相互施加的次序上的约束。当几条指令同时执行时，比如在标量流水线或超标量并行处理器中，这种效应就会显现。这个技术适用于WAR和WAW依赖关系——如21.3节所定义的。为了清楚地理解寄存器重命名，必须将更常用的二操作数指令转换成二操作数格式。这样就会存在两个源操作数，一个单独的目的地或结果，计算得出的结果常常会存

入某个源操作数所在的位置。在图21-8中，我们可以看到，在必要的时候，为了避免改写不安全的变量，对一个CPU寄存器的引用被重定向到另一个寄存器。这种重命名机制好像一种虚寄存器系统，和12.5节中介绍的虚拟内存十分相似。

寄存器虚拟化可以有效地解决WAW和WAR的依赖关系问题。依赖的控制问题（见表21-2），将在下一节以条件分支指令为例加以讨论。

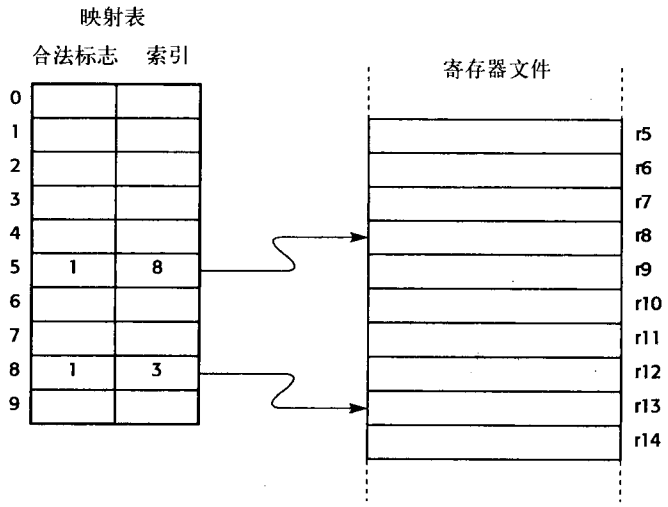


图21-8　寄存器重命名

21.5　分支预测方法：流水线的维护

与流水线译码相关的重要问题是，如何处理条件指令，如JZ、JNZ或LOOP。编译器一般不能确定这类指令后面的执行路线——这要依赖于运行期间求值的结果。在不采用流水线的CPU中，仅在前一条指令完全结束后，才从内存中读取下一条指令，没有什么影响。在下一条指令读入之前，分支决策已经做出。但在使用流水线译码器时，预取器同时保持几条指令，在尚未确定分支路线之前，条件分支指令代码已经走到译码流水线的中间。因此，预取器需要在做出选择之前，事先知道可能执行哪个分支。要么将所有的指令分支都读入进来，要么选择任一分支读取并执行，接受有可能执行错误分支的风险。如果预取了错误分支的指令，在条件指令完成求值后，就必须清空流水线，这不可避免地会导致执行上的停顿。此时，必须从内存中读入正确分支的指令。流水线停顿对于CPU的性能影响十分严重。

一些编译器提供一个选项，可以避免浪费时间的流水线清空动作和再次装入，使用这个选项时，编译器会对指令重新排序，或是在每个条件指令后面填充空操作（NOP）指令。这样就可以将分支指令的读取延后到决策做出，分支已知之后。

奔腾处理器采用硬件分支预测逻辑（Branch Prediction Logic，BPL，参见图7-4），它监视条件指令的结果，将它们存储下来，然后使用之前决策的结果猜测后续的分支。这叫做**推测执行**（speculative execution），对于那些特定的循环，比如查找特定数据项，十分有效。大部分时候循环跳回继续进行检验，仅当找到目标和失败后，才会采取其他动作。要实现这样一个策略，需要使用快速的高速缓存存储设备来跟踪条件指令的结果（BTB，见图7-4）。图21-9给出最简化的布局，这个表由三列组成，BPL在分支指令进入流水线时检查它们的地址。如果它在最近的时间段内首次执行，地址就会记入表中。如果指令已经在表中存在，

条件指令的地址	分支目标的地址	预测的可信度

图21-9　控制单元的分支预测表

BPL则使用第二项数据作为该分支的最佳选择，从该地址继续执行。第二项用于保存可信度，这个值根据条件分支早期执行的结果得出。

21.6 编译器支持：RISC的重要组成部分

HLL编译器现在已经被看做是RISC CPU重要的关联部件。硅芯片提供商，如Sun或Intel，推出微处理器后，不再依赖于第三方程序员编写编译器，提供专门开发的C编译器，是任何微处理器取得成功的基本要素，尤其是对RISC设备。这种情况部分是因为指令集的简化对软件库提出更大的要求，部分是为了让预处理软件能够充分利用RISC CPU提供的并行处理能力。21.3节讨论的重大问题（数据和代码的依赖关系），部分可以在编译时解决，这样代码就可以充分地利用标量和超标量并行机制。

在CPU提供译码流水线时，对于无条件分支指令，如CALL或JMP，编译器能够保证，预取器会在流水线执行完CALL指令（实际负责切换到新的位置）之前，将焦点切换到新的目标。否则，CALL后面的指令会进入流水线中，而这些指令是没用的。

编译器可以通过重新排列原始的代码序列，更好地利用并行功能，这样可以避免数据和代码之间的相互依赖，或者更充分地利用硬件资源。

21.7 ARM 32位CPU的起源

第一代32位ARM RISC CPU诞生于1985年，Acorn公司设计它来取代8位的6502芯片，它的运行频率只有8 MHz。当初的意图是在BBC微型教育用途计算机（一种十分成功的产品，人们亲切地将它称为Beeb，见图21-10）的新机型中采用快速的RISC处理器。已有的机型使用Rockwell 6502芯片，由于处理能力和寻址范围有限，限制了机器整体性能的提高。该计算机最初的型号提供彩色图形显示和32 KB的RAM，并能够在录音带上存储数据。后来，硬盘和软盘驱动接口出现。Acorn在Econet系统中就已经建设过局域网络。BBC微型计算机另一项新颖的特性是，它能够通过扩展端口（即Tube）接受附加的协处理器。对于连接性、互操作性和联网等技术，在PC世界真正实现它们很久以前，BBC微型计算机的用户早就对它们耳熟能详。

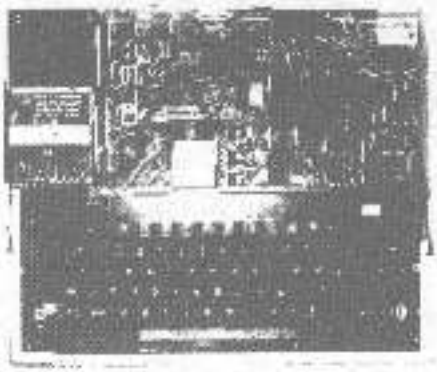


图21-10 早期的6502 Beeb

遗憾的是，新型的基于RISC的机器（Archimedes），由于不同于IBM PC，受到诸多诟责，仅在学校和大学内获得有限的市场份额。最初的RISC CPU设计方案经过几个版本的演化。当静态、低能耗的电路研发出来之后，可移动、手持式设备对于CPU的需求才刚刚起步。Apple、Acorn和芯片制造商VLSI技术公司合资组建了新的公司，ARM的含义也从"Acorn RISC Machines"平稳地转变为全新打造的"Advanced RISC Machines"。原来的工程师和设计人员中，许多依旧从事ARM的开发，有一些转为从事咨询工作。新公司在技术和营销上都得到Apple的大力支持。第一代便携设备采用低能耗的ARM技术——基于v3架构的ARM610微控制器，它就是Apple Newton。这种合作在Apple iPod身上得到继续，Apple iPod使用ARM v4架构的CPU。随着蜂窝移动电话的发展，手持设备的市场呈爆炸性增长，各种复杂的语音压缩/解压缩和错误纠正算法需要强劲的运算能力。这极大地加速了市场对低能耗、高效能CPU的需求。ARM v5架构加入额外的指令，如乘法和累加，以加速MPEG解码和实时数字波波过程中的DSP算法。在移动手持设备的开发过程中，最成功的微控制器是德州仪器公司的OMAP。它内部集成了ARM v4T架构的CPU，还提供十分高效的电力管理以及辅助DSP协处理器。

原来的Acorn公司也从教育PC的制造商演变成在CPU架构的供应、电路设计或知识产权领域内闻名的世界级领导者（见表21-3）。也就是说，其他OEM供应商在获得许可后，可以将描述完整CPU的VHDL代码用在它们的设备中。通过这种方式，DEC（Digital Equipment Corporation，数字设备公

司）购买了ARM v4设计方案的授权以及进一步扩展它的权利。这就是后来由DEC转让给Intel的SA-1100 StrongARM的由来。Intel继续开发ARM系列CPU，将其重命名为XScale，并将核心转换到ARM v5。其他一些开发商则购买许可后直接使用，完全遵循现有的ARM设计方案，不做更改。

表21-3　ARM架构的修订史

版　本	核　心	说　明	永　线
v1	ARM1	最早的处理器，26位地址，Beeb系统的协处理器	3
v2	ARM2	32位乘法器，包括寻处理器，Acorn Archimedes/A2000	3
v2a	ARM3	提供芯片内高速缓存，支持信号量机制和交换指令	3
v3	ARM6 and ARM7D1	32位地址，CPSR/SPSR，MMU，首个宏单元产品	3
v3M	ARM3M	乘法增强，支持64位结果	3
v4	StrongArm	8/16位接笔LD/ST，系统模式，iPAQ PDA	5
v4T	ARM7TDMI	压缩的指令集，MULA，用于多种移动手持设备	3
	ARM9TDMI		5
v5	XScale		3
v5TE	ARM9E and ARM10E	更好的MUL，额外的DSP支持	5
v5TEJ	ARM7EJ and ARM926EJ	Java选项	5
v6	ARM11		8

不同于其他一些更为传统的面向RISC的架构，ARM CPU只提供相对较少（16个）的数据/地址寄存器，以及一个保存操作状态的条件寄存器。所有的寄存器均为32位宽，加上其他20个"影子"寄存器，总共有37个CPU寄存器，见图21-11。寄存器r0－r12是通用寄存器，r13、r14和r15分别用做堆栈指

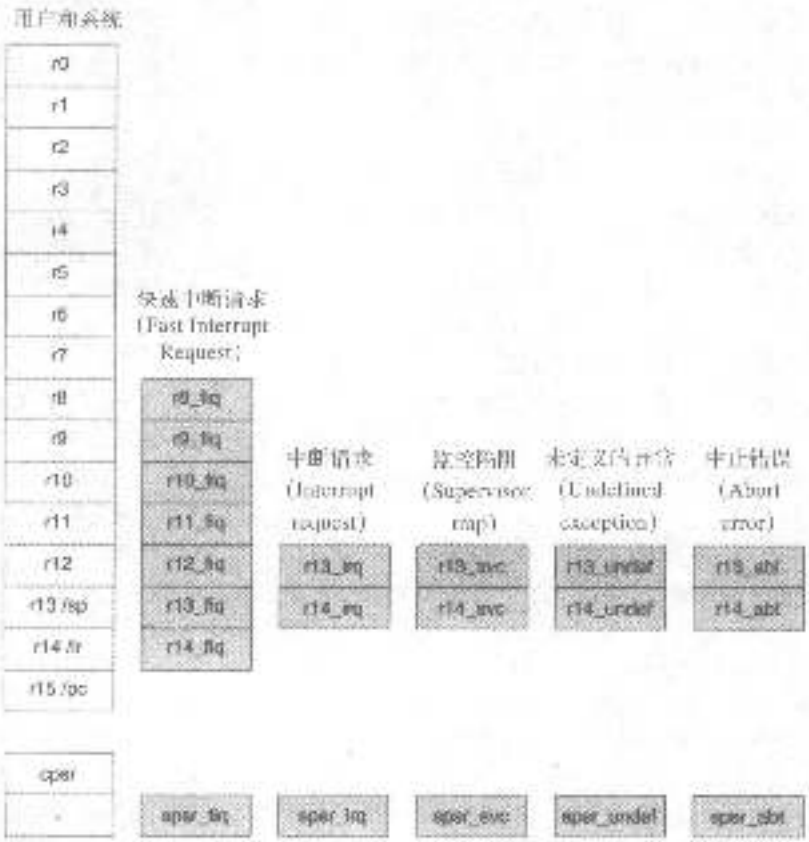


图21-11　ARM CPU寄存器

针、返回地址的链接寄存器和程序计数器。当前程序状态寄存器（Current Program Status Register，CPSR）是单独的。发生异常时，CPU会自动将某些寄存器切换到替换版本，这样能够保护原来的值，较之传统的将寄存器压入到堆栈或从堆栈弹出而言，这样运行速度更快。乍看起来，ARM硬件的组织方式十分直观，但在处理指令集时，情况显示会发生变化。这种方案经过深思熟虑，设计十分精巧。

ARM处理器的设计目标是小型、高效和强劲。它是32位流水线化的RISC，它采用LOAD/STORE架构，支持约50条机器指令，有些指令需要三个操作数，比如：

```
add R0, R1, R2  ; R0=R1+R2
```

这条指令将R1和R2中的内容相加，结果存入R0中。这个汇编语句采用的是从右到左的惯例，和Intel奔腾处理器采用的格式相同。ARM处理器完全遵循RISC风格，所有的内存访问均由单独的载入和存储指令来完成，它只能使用寄存器间接寻址表示内存存储单元：

```
ldr R0, [R5]  ; R0 = M[R5]
str R8, [R0]  ; M[R0] = R8
```

要注意，这对汇编助记符的左右次序是相反的。你可能以为存储指令应该是str [R1], R3，但事实并非如此。

ARM不支持直接载入32位的内存地址，这是因为ARM的指令均为32位，没有办法将32位的地址载入到寄存器中。ldr/str指令的确有12位的立即字段，加上符号位，在两个方向上均可以提供4 KB的偏移值，在位移模式下可以提供寄存器间接寻址：

```
ldr R0, [R5, #250]  ; R0=M[R5+250]
```

这种功能更常用来引用数据结构内部的数据。如果数据项为8个字节长，可以使用下面的指令：

```
ldr R0, [R5, #8]!  ; R0=M[++R5] preindexing
ldr R0, [R5], #8  ; R0=M[R5++] postindexing
```

注意指令中惊叹号!的使用，它表示指针寄存器（R5）在指令完成后会保持递增后的地址值。这不需要激活；如果愿意，原来的地址还可以保持，供以后使用。自动变址的寄存器间接寻址和位移寄存器间接寻址均可以使用这种功能完成。图21-12详细标示出该指令的各个字段。这可能是理解ldr和str这对指令的各种复杂选项和不可用组合的最好方式。我们得仔细考虑这些例子，它们实现了堆栈的PUSH和POP：

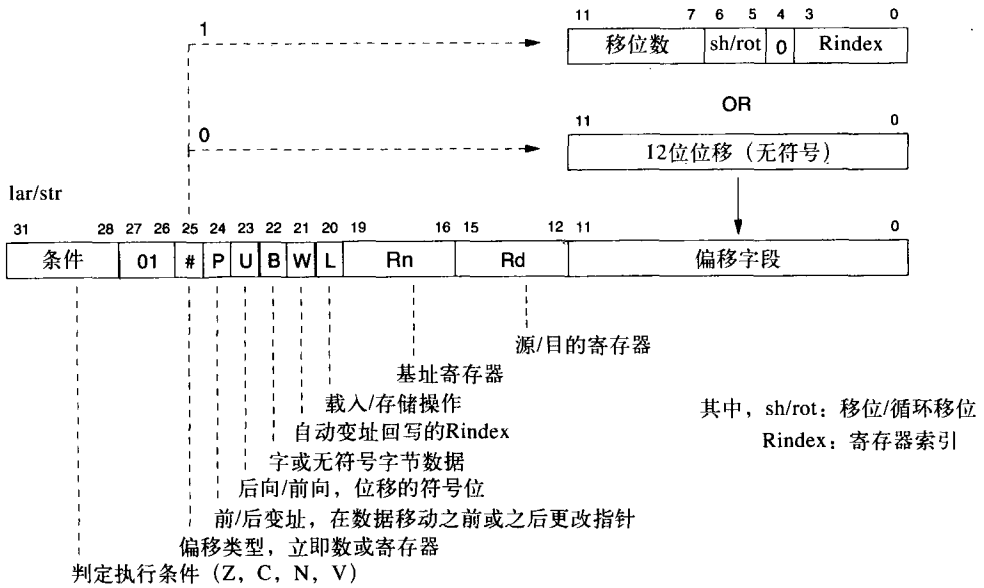


图21-12 ldr和str指令的指令格式

```
str R0, [R13,#-4]!  ; PUSH r0
ldr R0, [R13],#4  ; POP r0
```

操纵堆栈时一般使用ldm/stm指令，因为它们均能够同时处理多个寄存器：

```
stmfd R14!, {R0-R5}  ; PUSH R0-R5
ldmfd sp!, {R0-R5, R14}  ; POP R0-R5 and R14
```

这些被PUSH和POP的寄存器，都由指令内的16位字段来表示。因此，花括号内寄存器清单的次序并不重要。汇编器会对寄存器清单排序，将它们按从低到高的次序放在堆栈上。这种策略的负面效应之一是，我们不能使用传统的堆栈技术进行交换：

```
stmfd sp!, {R0, R1}
ldmfd sp!, {R1, R0}
```

这两条指令并不能交换两个寄存器的内容！

习惯上，R13用做堆栈指针，因此，汇编器允许程序员使用sp替代R13。fd指Full, Descending。它的作用是表示堆栈在内存中的增长方向，向下还是向上，堆栈指针是停留在空的存储单元，还是存有数据的存储单元。ARM建议使用Descending, Full选项。

ARM CPU为程序员提供的一项重要特性是，提供在线桶形移位电路，可以在同一机器周期内指令执行前，对指令的某一操作数移位或循环移位。下面是具体的例子：

```
mov R5, R6, lsl #4  ; R5=R6*16
```

或者将R0乘以10：

```
mov R0, R0, LSL #1  ; x2
add R0, R0, R0, LSL #2  ; x4, then add 2R0+8R0
```

下面的例子中，存储在R6中的32位值在移往R5的过程中，被乘以8——通过左移位逻辑运算符(lsl)。向左或向右移位能够作用于任何来自于寄存器或指令中立即数字段（8位常量）的操作数。移位的数量可以存储在第三个寄存器中：

```
mov R7, #3
...
mov R5, R6, lsl R7  ; R5=R6*8
```

当需要将常量、数值或地址载入到寄存器时，情况变得比较复杂。我们已经提到过，ARM没有32位的立即数或内存直接载入指令。mov指令的**立即数**字段只有12位宽，其中8位可以用来存储基本的常量值，剩下的4位用来存储左移位的计数。我们可以通过左移正确的位数，使用8位的值合成出32位的常量：

```
mov R0, #0xFF, LSL #24  ; load R0 with $FF_00_00_00
```

另一项小技巧是使用程序计数器间接寻址，使用12位的位移来访问存储单元。这样可以产生位置无关的代码：

```
mov R0, [pc, #123]
```

或者利用汇编器提供的功能，引用邻近的表：

```
mov R0, [pc, #lut1-{pc}-8]  ; 将12345678H载入到R0中
...
lut1: DCD 0x12_34_56_78
```

为了帮助汇编语言的程序员，汇编器一般提供两个伪指令，它们允许汇编器评估所需的常量，选择最好的方式——合成或间接——载入它：

```
ldr R0, =0xFF  ; load 32 bit constant into R0
```

和：

```
adr R0, hereX  ; load R0 with effective address 'hereX'
```

根据图21-12，我们知道最左侧的4位字段与CPU状态标志有关。图21-13将它们列了出来。所有

的算术和逻辑指令常常会影响状态标志，在某些处理器上，甚至于数据移动指令都会改变某些标志。有些程序员认为这样很不方便，因此，ARM指令包括控制操作对标志产生影响的功能。这是通过S位来完成的，见图21-14。具体的汇编操作见下例：

```
subs R0, [R1]  ; subtract and set status flags
mov R1, #34  ; no effect on status flags
bl finish  ; conditional branch
```

注意，减法会设置状态标志，但随后的mov不会。这样，状态信息将两条指令连接在一起，即使中间有其他操作发生。ARM引入条件指令集的概念，有时也被称做判定集。ARM没有依赖于几个条件跳转指令来实现程序的IF-ELSE和SWITCH-CASE结构，而是让所有的指令均根据条件执行。这对于“选择”代码结构影响流水线的方式有重大的正面效应。如果代码中只有短小的可选的分支，ARM将会读取和执行来自于所有分支的代码，使用条件字段来放弃来自于无效分支指令的结果。这样就避免了读取错误的分支后，不得不停下流水线，花费几个机器周期从另外的分支读取指令的情况。

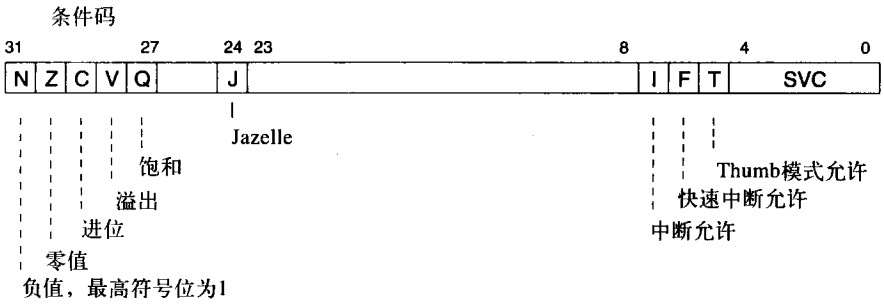


图21-13 ARM程序状态寄存器

28～31位中存储的数据经过编码，与直接使用4位掩码相比，这样能够提供更多的条件。通过表21-4可以对这个方案有个清晰的了解。

表21-4 ARM条件码

操作码	助记符	含 义	标 志
0000	EQ	相等的值	Z=1
0001	NE	不相等	Z=0
0010	CS	进位设置	C=1
0011	CC	进位清除	C=0
0100	MI	负值	N=1
0101	PL	正值	N=0
0110	VS	溢出	V=1
0111	VC	无溢出	V=0
1000	HI	无符号较大	C=1&&Z=0
1001	LS	无符号较小	C=0||Z=1
1010	GE	大于等于	N=V
1011	LT	小于	N!=V
1100	GT	大于	Z=0&&N=V
1101	LE	小于等于	Z=1||N!=V
1110	AL	总是	

程序状态寄存器含有5个条件标志，以及一些表示操作模式的标志。条件标志能够通过指令左侧的条件判定字段，影响大多数指令的输出。我们将在第22章中介绍安腾处理器时，再来介绍这项技术。



条件码	助记符	含义	效果
0000	and	逻辑按位与	Rd = Rn && Op2
0001	eor	逻辑按位异或	Rd = Rn ^ Op2
0010	sub	减法	Rd = Rn - Op2
0011	rsb	反向减法	Rd = Op2 - Rn
0100	add	算术加法	Rd = Rn + Op2
0101	adc	带进位输入的加法	Rd = Rn + Op2 + C
0110	sbc	带借位的减法	Rd = Rn - Op2 + C - 1
0111	rsc	带借位的反向减法	Rd = Op2 - Rn + C - 1
1000	tst	测试	Rn && Op2
1001	teq	测试相等性	RN ^ Op2
1010	cmp	比较	Rn - Op2
1011	cmn	取反比较	Rn + Op2
1100	orr	逻辑按位或	Rd = Rn | Op2
1101	mov	无修改数据	Rd = Op2
1110	bic	清除位	Rd = Rn && ~Op2
1111	mvn	取反后复制数据	Rd = ~Op2

移位类型编码	效果
00	Rn, LSL #移位数
01	Rn, LSR #移位数
10	Rn, ASR #移位数
11	Rn, ASL #移位数

图21-14　移动、算术和逻辑指令的指令格式

尤其要注意的是CPSR中的T和J标志，它们表示指令集的变动。前者对应于Thumb指令集，针对更小型的嵌控制器而制订，它使用由16位指令构成的有限指令集，而后者对应于Jazelle指令集，专为实现Java引擎而设，可以直接执行Java的字节码。这让人回想起早期人们基于专门的循码PDP 11 CPU开发和营销能够直接执行Pascal P代码的处理器的往事。其他解释型语言，如BASIC和Forth，也适合这种处理。BX和BLX指令——分支和交换——能够将ARM CPU在32位和16位指令集间切换。要注意，Thumb模式下数据依旧为32位，但只能访问寄存器R0～R7。

接触新的汇编指令集，如ARM-32指令集时，开始熟悉它时总会有绝望而生畏。我们可以从使用全部指令集的一小部分开始，如表21-5中所列。

表21-5 ARM处理器的入门级基本指令

mov Rn, Rm	在寄存器间复制数据
ldr Rn, [Rm]	从内存中获取变量
str Rn, [Rm]	将变量写回内存
add R0, R1, R2	将两个寄存器相加，结果存入第三个寄存器
cmp R0, R1	比较两个寄存器
b addr	跳到相对位置（±32 MB）
bl addr	调用子例程（±32 MB）
mov R15, R14	从子例程返回
ldmfd R13!, {Rm－Rn}	从堆栈上弹出寄存器
stmfd R13!, {Rm－Rn}	将寄存器压入堆栈中
ldr Rn, =constant	载入常量，宏汇编
adr Rn, label	获得地址，宏汇编

21.8 StrongARM处理器：32位微控制器

DEC在ARM v4 CPU设计的基础上，开发出高性能的32位StrongARM微控制器。Intel接手后继续进行开发，引入SA-1110和最近的XScale系列处理器——PXA270，其时钟达到624 MHz。StrongARM采用256针的小型球栅阵列封装（Ball Grid Array，BGA）。它集成的功能包括：6通道DMA、PCMCIA接口、28条并行IO线路、SDRAM接口、JTAG测试和5个高速串行通道——服务于RS232、I^2C和USB。

DEC在StrongARM中采用了为高端Alpha处理器开发的技术，第一代200 MHz StrongARM处理器使用0.35μmCMOS制作工艺，在50mm^2的芯片上采用三层金属掩模连接近二百五十万个晶体管，其时钟达到200 MHz，但散发的热量不足一瓦。它与奔腾处理器20～75W的热量形成鲜明的对比！DEC StrongARM采用五级译码流水线，数据转发支持降低了数据依赖的影响，并且拥有独立的16 KB代码和数据高速缓存。

由于芯片封装仅引出26条地址线，因此它能够访问的内存范围只有64 MB，地址中的5位用于选择存储体（bank），余下的27位用来在存储体内寻址。32个存储体中，每个存储体只有一半的容量能够被访问到。但是，32位的虚拟内存管理单元（Memory Management Unit，MMU），连同地址转换后援缓存（TLB），能够处理4 GB虚拟地址到64 MB物理页面之间的映射。SA1110还提供全内存地址的译码和段选择电路。

系统复位后，CPU开始从0000_0000H读取指令，该地址是系统的中断向量表的基地址。为了处理这种情况，我们必须在向量表中设置合法的分支指令来处理异常事件。

在StrongARM中，指令和数据缓存均在MMU之上，即它们使用虚地址。这不同于奔腾处理器，奔腾处理器会对地址进行预映射，将物理地址传递给L1和L2高速缓存。在虚拟空间中操作高速缓存会带来一个问题。如果我们将同一物理主存映射到多个虚地址，那么改写同一主存存储单元可能会涉及多个高速缓存项，想保持所有的高速缓存项都最新不易做到。对于共享的变量，这可能是个大问题，解决这种由于双重映射虚地址带来的缓存不一致问题，推荐的方案是阻塞高速缓存，或内建总线监视电路。

StrongARM含有存储器映射译码电路，通过它可以管理所有已安装存储设备和IO设备的可能布局。存储器映射被拆分成128 MB的页，见图21-15。这些页中，一些被内部映射，用于微控制器提供的各种功能，其他的页可以用于另外的外部设备。图21-16给出CPU的示意图，图中给出了单条执行流水线的五个阶段。图中没有给出数据转发路径。32位CPU被芯片上众多额外的支持电路和IO设备包围，这在当今的微控制器中十分常见。图12-17给出的框图展示出系统开发人员可以使用的众多设施。其中5条串行通道用于连接其他系统，以太局域网接口尚未提供。



图21-15　SA1110 StrongARM中4 GB的存储器映射

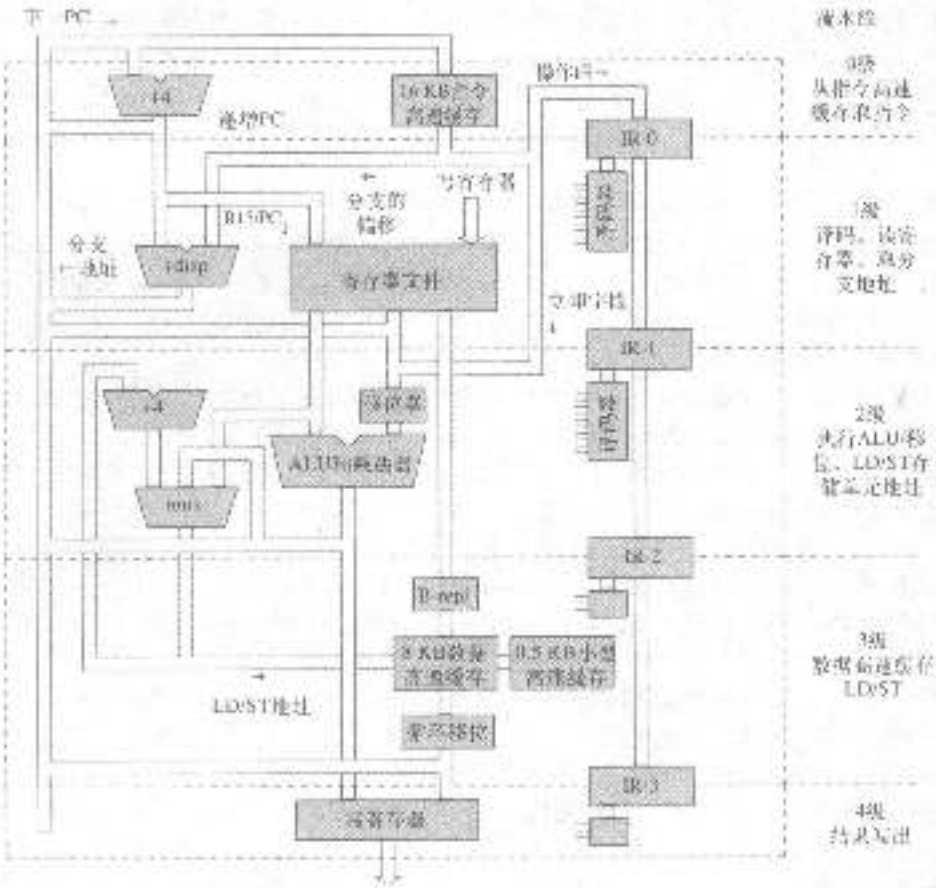


图21-16　StrongARM核心框图

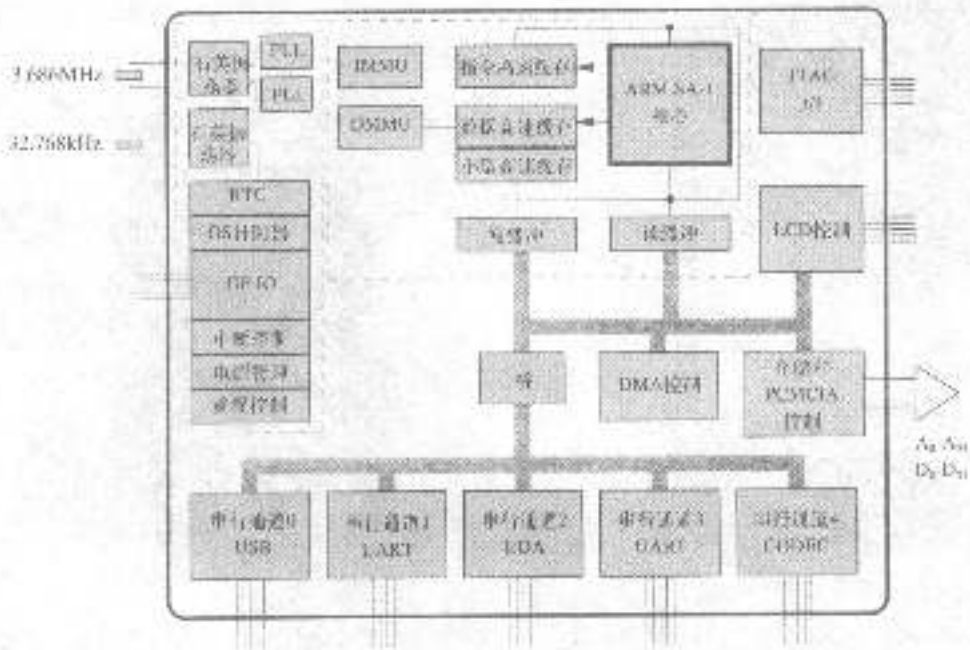


图21-17 Intel SA1110 StrongARM微控制器

所有采用流水线技术的CPU面临的一个重大问题，就是如何解决数据之间的依赖关系，否则会影响指令通过流水线的处理过程。当两个直接相邻的指令都使用某个寄存器作为操作数时，就会发生这种情况。第一条指令计算出一个结果，将它存储在寄存器Rn中，下一条指令的执行就需要使用Rn中的值。这叫做“写后读”。由于结果直到流水线的最后一步才写入到Rn中，此时下一条指令早就经过了执行阶段。一种解决方案是使用NOP将互相链接的指令分隔开来，或是由编译器对代码进行重新排序。StrongARM采用一种不同的方式，它提供一条特殊的“小路”，可以在所需的值写入到寄存器之前，快速地访问到它。这就是“数据转发”——尽管涉及的值往往是从ALU发出。

ARM芯片上的高速缓存管理不同寻常。它没有采用常见的“最近最少使用”（Least Recently Used，LRU）算法，在需要更多空间时，按照优先级别将过时的数据行移出，相反，它只是随机地移出数据行。经过模拟研究后，研究人员发现简单策略的效率和LRU相同，但实现起来却简单得多。当Intel接手StrongARM的开发之后，Intel决定在Dcache附近提供数据微缓存，给予程序员更多控制数据缓存活动的权力。在ARM v5之后，ARM提供PLD指令，预测未来的需求，将数据项预取到Dcache中。有准备的预取好像很切合实际，但依旧存在为准备空间存储预取的数据而将必需的数据移出缓存的风险。如果预取的数据最后没有被使用，则是程序员自己的责任了。

21.9 HP iPAQ：StrongARM PDA

康柏电脑在1999年首次引入iPAQ系列掌上电脑（见图21-18和图21-19），人们一度将其称为个人数字助理（Personal Digital Assistant，PDA）。惠普收购康柏后，放弃了自家的Jornada产品，继续iPAQ产品的研发。该设备基于Intel的StrongARM/XScale系列ARM微处理器，其上运行的是精简版的Windows。运行Microsoft Pocket PC版Windows的PDA叫做掌上电脑，早期的一些设备使用Windows CE。将Linux移植到iPAQ的活动也非常活跃。



图21-18　HP iPAQ hx4700掌上电脑，采用XScale PXA 200处理器（http://www.hp.com）

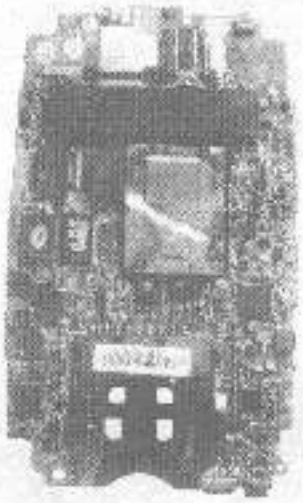


图21-19　HP iPAQ hx2400印制电路板。420 MHz的XScale被金属薄片覆盖以降低辐射（来自iPAQ备件和维修公司：http://www.ipaqrepair.co.uk）

HP iPAQ hx4700系列掌上电脑包括大量的功能。它是第一款提供触摸屏的掌上电脑，通过类似于鼠标的光标，使得屏幕使用起来十分类似于桌面电脑。这款iPAQ采用了新型的Intel PXA270 624 MHz处理器，性能强劲，还集成CF卡和SD卡插槽，用于存储和IO扩展。集成的无线设施包括无线局域网、蓝牙和快速红外线接口，允许用户在移动中始终保持连接。新的HP iPAQ hx4700还提供100mm的屏幕，支持16位VGA色，480×640分辨率，可以显示更大的图像和更高质量的分辨率。更大的屏幕能够让用户更容易地查看文档、图像和浏览网页。电子邮件、日历、联系人和文件等都存储在完全加密的闪存或外部存储卡上。为了防备设备被窃或丢失时信息泄密，我们可设置访问iPAQ时需要身份识别码或密码验证。如果用户经常通过Internet或无线局域网访问敏感的数据，这些增强的安全功能会深受用户欢迎。

对于那些忙于调试StrongARM代码在运行时出现的故障的程序开发人员而言，XScale引入JTAG调试功能是一个重要的进步。StrongARM上的JTAG端口仅为硬件工程师的需求而设，芯片焊接到印制电路板上之后，硬件工程师用它来测验芯片的功能。XScale还提供额外的寄存器，支持JTAG的电路内模拟（In Circuit Emulation，ICE）。通过这项机制，在程序抛出异常或是到达预设的断点之后，程序员可以截获寄存器中的数据和地址值。

21.10　Puppeteer：StrongARM SBC

下面介绍另一个使用StrongARM微控制器的嵌入式系统。Puppeteer（操纵木偶的人）单板计算机（Ansea Microsystems）使用为电视机顶译码盒开发的计算机板，但其应用更为广泛。

如图21-20和图21-21所示，Puppeteer单板计算机有一块支持双面、150×100mm的印制电路板，它支持一个StrongARM CPU、一个大的FPGA、两个CPLD、以太网芯片、SRAM、闪存、四个高密度扩展接口、稳压器、为实时时钟芯片提供电力的锂电池，以及一些让系统能够运行所必需的基本电子器件。其右侧有一个扩展槽，允许各种专门的IO卡通过它连接到主CPU板。图中给出一个富有

特性的可重新配置的CPLD卡。它可以模拟大量的IO设备，包括一些24位的并式设备，如8255I 该IO卡有一个JTAG端口，在需要时，可以通过它来重新配置该CPLD。

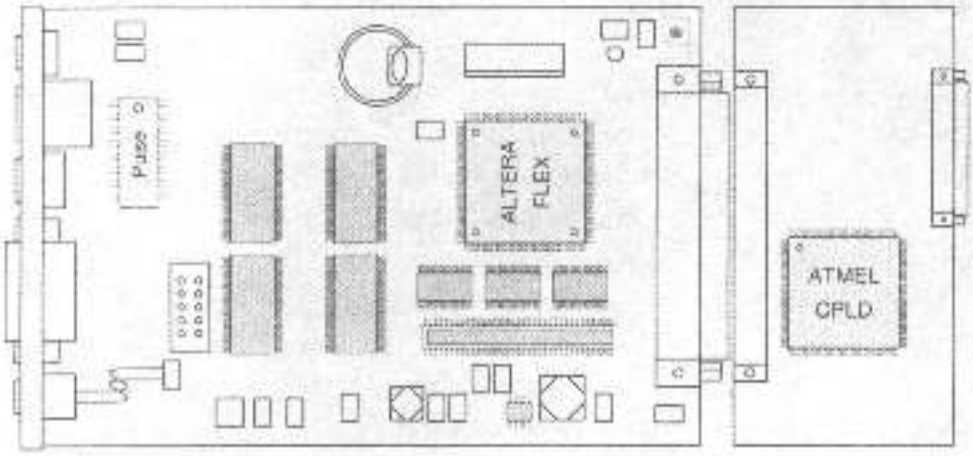


图21-20 Puppeteer单板计算机和IO扩展卡

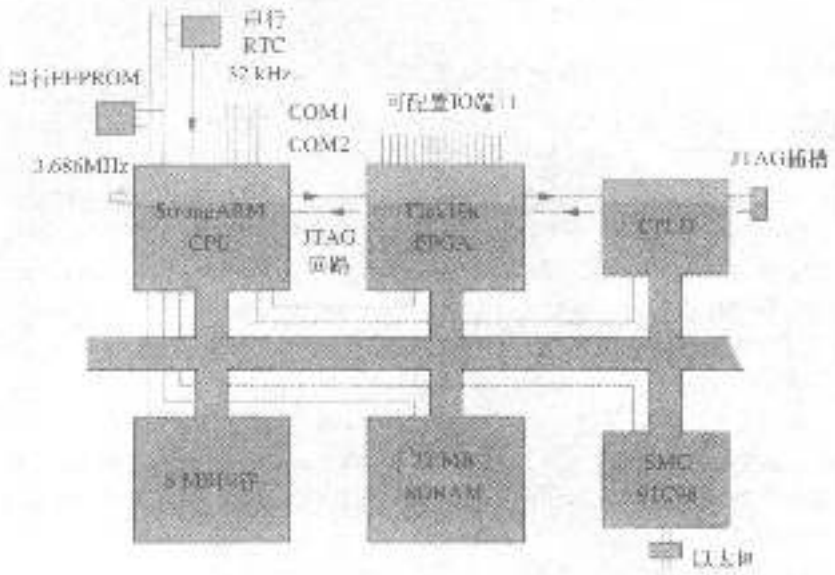


图21-21 StrongARM Puppeteer电路板的系统布局

Puppeteer在启动过程中，要经过几个不同的存储配置。最开始是让载入和硬件设置更简单，而后是受益于StrongARM MMU访问控制机制提供的安全性（见图21-22）。第一阶段，在加电和重置之后，禁用虚拟内存映射，在代码中直接使用32位逻辑地址作为32位的物理地址；1对1的映射。此外，所有的CPU缓存活动均被禁用。在启动的第二阶段，通过用户级进程的虚地址范围，CPU缓存启用。MMU页面映射表被用来控制访问，以及缓存在CPU上运行的进程，而不是为可用的4 GB空间提供扩展的交换区。另外，物理上安装的SDRAM之间的空隙，在虚拟空间中通过恰当的映射得以移除。此处不再需要将主存溢出到磁盘交换文件中。MMU在这里起到不同的作用——访问控制器。

设备类型	大小（MB）	线性地址	选通线	支持缓存的虚地址
ZeroBank	4	E000_0000-E03F_FFFF		
后台缓存				
SDRAM1	16	C800_0000-C8FF_FFFF	RAS1	8100_0000-81FF_FFFF
SDRAM0	16	C000_0000-C0FF_FFFF	RAS0	8000_0000-80FF_FFFF
ASB	4	B000_0000-B03F_FFFF		
MMU	4	A000_0000-A03F_FFFF		
	4	9000_0000-903F_FFFF		
系统控制	4	8900_0000-893F_FFFF		
PCMIA1	1	3000_0000-300F-FFFF	PSTsel	
以太网	4	2000_0300-2000_03FF		
FPGA	4	1040_0000-107F_FFFF		
闪存	8	0000_0000-007F_FFFF	CS0	9140_0000-91BF_FFFF

图21-22　StrongARM Puppeteer的存储器映射

21.11　Sun SPARC：RISC架构的标量处理器

Sun微系统公司在1988年推出SPARC处理器，取代MC68030作为新型Sun-4 Unix工作站的处理器。对于标量（scalar）一词，依旧存在一些歧义。有些文章将它看做“可伸缩的”（scalable）。Sun将标量处理与其他提供向量处理能力的高性能设计区分开来。SPARC可以在指令流上同时执行多个标量运算，而向量处理器则以同步、并行的方式运行。相应地，“可伸缩”是指制造技术规划方面的提高，从而使芯片更小、更廉价、更快速，Sun总是力图用一种基本架构支持各种CPU，从便携式计算机到强大的数据库服务器。增强型“超标量”架构指可以同时运行多条指令流水线，从而每个时钟周期内可以完成多条指令。

新事物是营销的关键因素，开发时间也受到越来越大的压力。最初的SPARC芯片从设计到推向市场，只用了10个月的时间，由于没有复杂的微码需要处理，因此后续的修订要容易得多。

和Sun在软件上推行的“开放系统”策略一样，Sun向其他硬件制造商提供授权，使得它们可以提高供应量，加速其他版本的开发。由于指令集简单，RISC处理器一般易于生产和制造，但达到高性能依旧需要采用多处理器方案。Sun微系统使用Fujitsu Gate Array构建它的首个芯片。其他获得许可的制造商使用不同的技术制造它们的CPU，从Sun最初的设计中发展出许多不同的设计，包括Fujitsu提供的SPARClite系列嵌入式处理器。

为提高性能，SPARC处理器集成了大量硬件设备。独立的FPU有自己单独的浮点指令流水线。

- 我们已经看到，现代CPU远快于现有的内存，人们尝试各种策略，以避免插入耗时的WAIT状态。Sun使用芯片内的SRAM高速缓存，以及交替式DRAM，来避免这个问题。64位宽总线一次可以读取四条指令。超标量SPARC处理器区分整数和浮点运算，分别将它们安排到两个流水线中，使指令并行执行。预取行为使得CPU的活动能够重叠，进一步提高CPU的性能。
- SPARC中通过使用CPU内的准栈高速缓存，减少了对主存中堆栈的访问频率。这样可以实现快速的参数传递机制。这称做寄存器窗口机制。CPU寄存器存储体被划分成多个组，或窗口。我们使用短指针表示哪个窗口处于活动状态。CPU内部含有当前窗口指针（Current Window Pointer，CWP）寄存器，我们可以使用指令SAVE和RESTORE前进或回退。一个窗口有24个寄存器：8个IN寄存器、8个LOCAL寄存器以及8个OUT寄存器，任何时候下面的情况都成立：

IN寄存器：输入参数
返回地址
返回值
LOCAL寄存器：自动变量（局部变量）
OUT寄存器：子例程的输出参数
子例程的返回地址

这项技术的优点依赖于：在使用SAVE和RESTORE指令时，调用者的OUT部分与被调用者的IN部分重叠，见图21-23。由于CPU寄存器集的大小有限，因此有时会需要与主存中的堆栈进行交换。

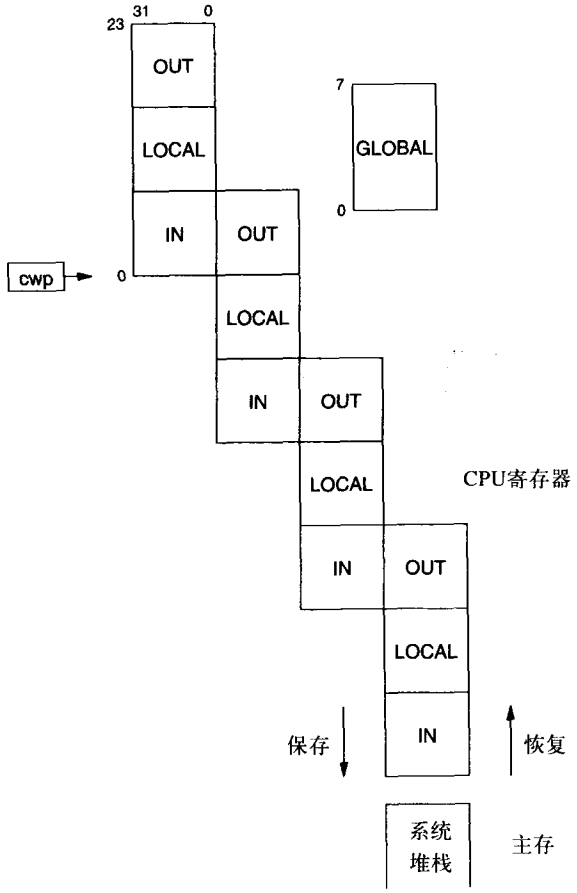


图21-23 过程调用期间寄存器文件的操作

21.12 嵌入式系统：交叉开发技术

微控制器设备的种类扩展得很快。以Infineon TriCore-32为例，它专为控制高性能引擎而设计，拥有专门的计时寄存器以及模拟电压转换电路——用以控制打火和燃料喷射。得益于这类可编程控制器提供的极佳的灵活性和精确性，我们能够控制引擎的喷射，提高燃料的效率。控制器区域网络（Controller Area Network，CAN）是一种专门的、富有弹性的网络标准，这一标准已经被几大汽车制造商采纳，用在驱动系统、子系统通信中。另外，模数转换器（ADC）也是必不可少的，我们需要它将来自于传感器的现实世界中的信号转换成适合于数字化处理的二进制格式。在这种应用领域，为了完成各种引擎管理功能，设备为计时提供非常多的硬件支持。

在进行基于微控制器的开发时，程序员需要了解并掌握**交叉开发**（Cross Development）技术。当目标计算机没有足够的资源支持实际的软件开发时，就会产生这个问题。或许它没有屏幕，甚至没有编译器。在这类环境中，就需要使用其他计算机作为开发主机，开发完成经过测试后再载入代码。图21-24说明了这种方案，图中微控制器安装在**单板计算机**（Single Board Computer，SBC）上。最后交付给客户的是单板计算机，而不是工作站。这类计算机常常叫做**嵌入式系统**，因为它们与它们监视或控制的设备紧密耦合在一起。

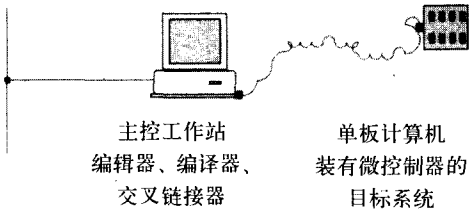


图21-24　微控制器系统的交叉开发

主机和目标设备之间，利用电缆下载执行代码以及在测试阶段提供调试信息。如果单板计算机提供以太网接口，则主机和目标设备之间可以通过局域网通信，这样更加方便。首次使用PC开发软件，代替完全不同的CPU运行编译器时，可能会觉得十分混乱。此外，链接过程也需要显式进行，相比于为Unix或基于Windows的主机链接程序，这种链接过程出问题的频率也更高。

XScale微控制器拥有32位处理能力，而洗衣机定序器、集中加热控制器或编织机的“大脑”，使用很简单的4位CPU就完全能够胜任。这类简单的微控制器中，最成功的是德州仪器公司生产的TMS1000，数以百万计的玩具中都内嵌了这一款芯片。这样，为了适用于任何可以想像得到的应用，甚至于停车收费表，人们设计出更多的微控制器。前面提到过，TI OMAP微控制器系列在单个芯片上提供ARM7和数字信号处理功能（DSP320C55），以满足移动手持设备的需求。毫无疑问，当语音处理算法足够可靠后，也会出现类似的芯片，来满足人们对语音控制设备的需求。所有这些都需要程序员的创造性努力。

21.13　小结

- ARM系列处理器对学术研究领域产生了深远的影响，同时它在低能耗移动设备上也得到广泛应用。
- 单芯片集成微控制器——包括CPU、存储器和IO端口，在商业上的重要性越来越大。
- StrongARM/XScale是高度集成的微控制器，专门针对由电池驱动的手持式PDA和移动电话等应用。
- 在使用流水线化的处理器时，程序员必须考虑邻近指令在数据和控制上互相依赖的可能性，否则程序的运行效能会受到流水线停顿的影响。
- HP iPAQ使用强劲的微控制器提供移动处理能力。
- 交叉开发指在宿主计算机上进行编辑、编译和调试，将代码从主机下载到目标设备的开发方法。

实习作业

我们推荐的实习作业包括着手调查CISC和RISC CPU架构的优点，在讨论会上演示得到的结果。可以为每种优点选择具体的例子并进行对比。

练习

1. 列出微码CPU架构的优点和缺点。除从开发角度出发外，还要从最终用户的角度来考虑。
2. 微控制器和微处理器之间的差别是什么？微控制器必须采用冯·诺依曼结构的CISC设计方案吗？有基于RISC的微控制器吗？如果有，请列举。
3. 如何决定将下面这些新功能插入到哪一层？

   在内存中成块地移动二进制数据；

浮点开方；
传真数据编码；
BST↔GMT系统时间转换。

4. 给出机器指令的五种类别，将下面这些ARM指令分别归入正确的类别：

```
mov R0, [R1]            addeq R0, R1, R2
cmp R1, #2              and R0, R0, R1
mov R15, R14            bgt DOIT
swi #10
```

5. 为什么寻址模式对于CISC更重要（相对于RISC CPU）？解释该指令的动作：MOV R0, #$FF, LSL #$F。在什么情况下编译器会使用这条指令？
6. 为什么RISC编译器比CISC编译器更具有针对性，体现在哪些方面？
7. 对于程序员而言，使用PC相对寻址最根本的优点是什么？ARM有这种模式，但仅支持32 KB的偏移。如何克服这种限制？
8. 你认为哪种寻址模式是最根本的，为什么？
9. 单条ARM指令能够使用多少种寻址模式？解释术语“有效地址”。
10. 下面这些指令都使用什么寻址模式？

```
movs r0, r1             add r0, #50
add r0, [r2]            ldr r3, [r0, #20]
str r4, [r0, r1]        ldm [r10], [r0, r4-r9]
orrseq r2, r3, r4, LSL #2
```

11. 说明这两条ARM指令的用途：ldmfs r13!, [r0, r1]和stmfd r13!, [r1-r5]。
12. 说明ARM CMP指令的动作。
13. 在程序执行期间，CPU状态寄存器的作用是什么？在CISC处理器中，它们可以看做是ALU和CU之间通信的一种手段。在RISC处理器中是这样吗？
14. 列举流水线化CPU架构的优点和缺点。

课外读物

- Furber（2000）。
- Sloss等著（2004）。
- Seal（2000）。
- Infineon网站上有关驱动系统CAN协议的介绍：
  http://www.infineon.com/cmc_upload/migrated_files/document_files/Application_Notes/CANPRES.pdf
- Intel公司的网站提供全套的StrongARM和XScale的手册：
  http://www.intel.com/design/intelxscale/
- ARM的网站：
  http://www.arm.com
- Heuring和Jordan（2004），以SPARC为例介绍RISC微处理器。
- Patterson和Hennessy（2004），介绍管道流水线方法。
- Hayes（1998），第5章。
- Sima等著（1997）。该书对本领域内的一些高级课题做了广泛的介绍。

第22章　VLIW处理器：EPIC安腾

安腾64位处理器由Intel和惠普合作开发。它和HP/Precision架构的CPU有大量共同之处。EPIC使用VLIW架构，在这种架构中，编译器起到至关重要的作用，流水线的选定和指令执行的次序，均由编译器负责处理。安腾处理器中，指令代码字的编码并不严格符合RISC和CISC计算机的特征，它更宽。它们以三个为一组从程序内存中读入。大部分指令均支持判定和条件执行。

22.1　安腾64位处理器简介

自1971年发明微处理器以来，从i4004开始，Intel就不断刷新用户能够期望的处理器性能。这部分是因为电路的物理尺寸不断得以缩小，部分是因为新技术的采用加快了读取-执行周期。现在看来，i80x86系列处理器作为Intel的旗舰产品的时光已经结束。32位奔腾处理器将被安腾处理器取代，安腾处理器原名Merced，它基于新的IA-64、EPIC架构（见表22-1）。它保持与当前的IA-32奔腾处理器的后向二进制兼容，不过，与该架构的断绝是不可避免的，因为IA-64不再与它的前辈们在指令级兼容。IA-64规范的诞生是惠普和Intel协作研究的产物，其目标是实现一种架构，能够充分利用近期从HP/PA RISC处理器系列得来的经验和教训。新的64位架构术语叫做显式并行指令运算（Explicitly Parallel Instruction Computing，EPIC），这是因为软件能够获得对流水线操作更多的控制。也有人将其称为**“精简编码计算机架构”**（Reduced Encoding Computer Architecture，RECA）。

表22-1　安腾2处理器的各项参数

L1指令高速缓存	16 KB，64字节每行
L1数据高速缓存	16 KB，64字节每行，写穿透
L2	256 KB，128字节每行，写返回
L3	3-9 MB，128字节每行，写返回
主存	< 1 PB（2^{64}）
时钟	1.66 GHz
系统总线	400 MHz，128位宽
处理器电力消耗	100W

摩尔定律再次使速度获得了较大的提升，同时也增加了电路的复杂度，Intel着眼于制造线路只有90nm的芯片。由于电路缩小而在硅芯片上空出的空间，可以用于各种不同用途。Intel选择增加字的宽度，并加入更多的执行流水线，以支持更大范围的**指令级并行机制**。**超标量**策略指连续地将指令压入多条流水线，使用硬件解决指令之间的依赖关系。而**VLIW**则是由编译器选出能够并行执行的指令，而后将几条指令装入单个字中。如前所述，超标量奔腾处理器在芯片上集成了依赖侦测单元，但如果由编译器来做这项工作，则可以释放出芯片的表面，供其他功能所用，比如提供更大的L1高速缓存。此外，复杂的电路不可避免地会限制硬件能够达到的最大时钟速度。电路越简单，时钟就可以运行得越快，相应地吞吐量也更大。

安腾架构试图开放对CPU的访问，让执行代码更明确地控制它的行为。具体的表现有：数据高速缓存区的内容对程序可见，并能够使用缓存行预取和清空指令对它进行操作。

现在，人们对于**VLIW**（Very Long Instruction Word，超长指令字）处理器有相当大的兴趣，采用VLIW可以同时为5～10个并行的译码流水线提供指令。对于安腾处理器，由于基本指令长度为41位，因此安腾系统只能归入**LIW**（Long Instruction Word，长指令字）机器类型。更长的指令能够减少开始执行要求的动作前CU需要执行的译码工作。同时，它为编译器选择操作组合提供前所未有的灵活性。其原因在于CU译码器，它需要解码的东西越多，编译器可见的操作越少。8个代码（000，001～110，111）只有8种不同的输出。但实际上可以使用的输出模式为256种（00000000，00000001～11111110，11111111）。如果指令字段从3位扩展到8位，编译器将有256种不同的操作可以使用，从而全面地控制计算机。

如果指令增长到96位，它就达到了之前CISC处理器中微码指令的长度。那么，译码和执行周期会非常快，编译器可以精确地指定机器周期的每个阶段会发生什么。为了支持这种应用，安腾每次读取三条指令，见图22-1。128位的包中含有三条41位的指令，每个指令都有6位的条件字段或**判定值**。这个包中还承载有来自于编译器的5位**模板代码**，用来表示指令是否能够并行执行。

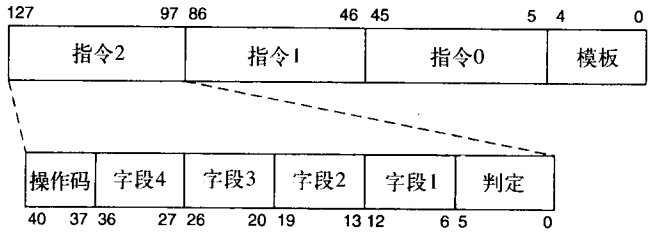


图22-1 IA-64架构的指令包

64位指令指针（IP）中保存含有当前指令的指令包的逻辑地址。由于指令包由16个字节组成，因此这是指令读取的最小单位，IP的最低4位永远为0。IA-64在设计时，就考虑到困扰流水线架构的几个问题，参见21.2节。IA-64架构处理器的特性之一，就是能够预取和执行IF-ELSE结构所有分支的指令，直到条件完成测试并确定为止。这样就减少了在运行期间预测应该预取哪个分支的需求，同时减少了流水线清空的风险，避免了流水线重载导致的性能损失。针对安腾处理器的编译器承担更多的责任，它负责将指令重新排序，以避免任何可能阻塞执行流水线的互相关。它使用的技术叫做**判定执行**（predication），ARM用户对这个词已经比较熟悉，前面介绍条件性指令时曾提及过这项技术。判定执行是一种用来降低条件分支指令数目的方法，使用这种方法实现的程序，不再需要CU对指令重新排序。我们直接在每条指令中嵌入条件测试功能。除ARM外，这个功能在过去仅仅限于少量的指令。奔腾提供条件传输指令（CMOV），8位的Z80支持条件调用和条件返回。在这些指令中，动作由CPU的状态标志控制。IA-64方案扩展了CPU状态标志的数量和用途，将传统的条件测试→标志设置、标志测试→条件分支和CPU状态标志绑定起来，扩展它们的作用域，使之影响到所有的指令。安腾处理器共有64个**判定标志**，见图22-2。它们能够控制几条并行执行指令的输出。判定执行技术通过将所有的指令都根据条件执行，有效地减少了代码中的条件分支指令。

所有128个通用寄存器均有一个与之相关联的第65位，叫做NAT状态位。NAT的意思是“Not A Thing”，它用来表示寄存器中保存的数据的合法性。如果数据所在的寄存器的NAT状态位被标记，当数据参与到其他的运算时，NAT状态会随着数据一同转发，执行推测数据载入时，这是一项十分重要的特性。

通用寄存器存储体都有一个相关联的当前**帧标记**（Current Frame Marker，CFM），它表示当前帧（保存局部变量和参数）在寄存器存储体中的位置。寄存器GR1到GR31专为通用的数据/地址活动保留，32到127可以用做子例程的数据帧。通过重叠调用者和被调用者的帧，我们可以引入一种完全不需要任何数据复制动作的参数传递方法。这类似于SPARC处理器的堆栈窗口技术（见图21-23）。要注意第一个寄存器GR0，在读取时它为0，在写入时会产生一个运行时错误（见表22-2）！

表22-2 安腾寄存器的用法

寄存器	名称	称谓	分类	用法
Gr_0	r0		常量	读取时为0，不允许改写
Gr_1	r1	gp	特殊	全局数据指针
Gr_2-Gr_3	r2-r3		临时	与addl一起使用
Gr_4-Gr_7	r4-r7		保持	
Gr_8-Gr_{11}	r8-r11	ret0-ret3	临时	函数返回值

（续）

寄存器	名称	称谓	分类	用法
Gr_{12}	r12	sp	特殊	堆栈指针
Gr_{13}	r13	tp	特殊	线程指针
Gr_{14}-Gr_{31}	r14-r31		临时	
Gr_{32}-Gr_{39}	r32-r39	in0-in7	自动	函数参数
Gr_{32}-Gr_{127}	r32-r127		自动	输入寄存器
		loc0-loc95	自动	本地寄存器
		out0-out95	自动	输出寄存器
			自动	循环寄存器（8个为一组）

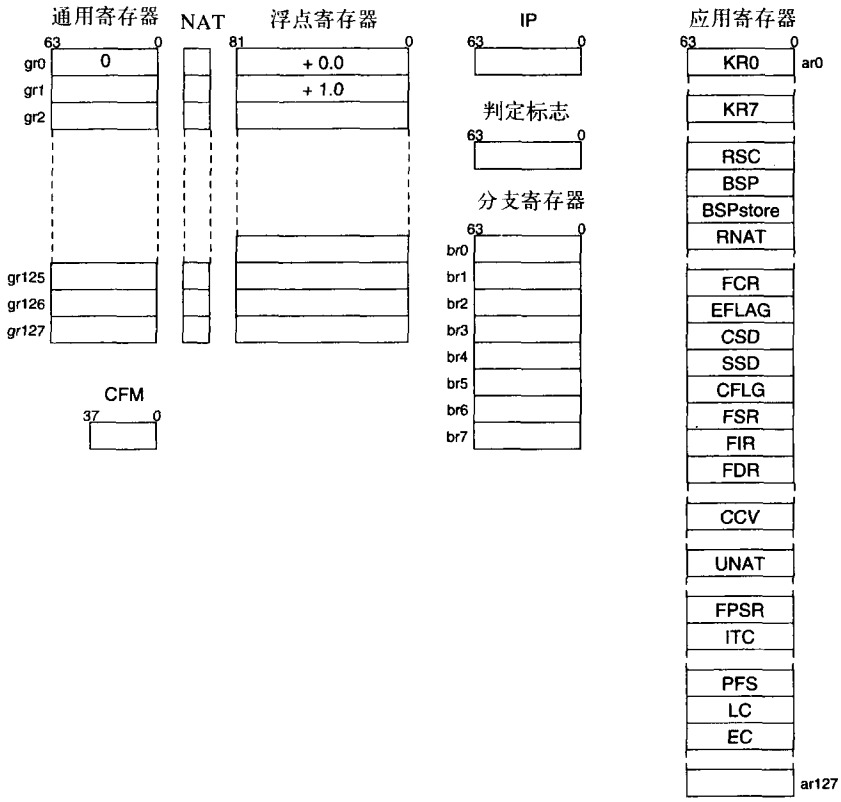


图22-2　Intel IA-64 安腾寄存器集

较高优先级的操作系统例程和用户应用程序之间的参数传递，由**内核寄存器**（Kernel Register）负责处理。对于这种情况，内核寄存器可以提供快速安全的通信。

有了判定执行指令集，我们可以处理多个指令流（如IF-ELSE）。无论是单一执行流水线还是部署超标量、并行设备，这都将很有帮助。更好的是，程序员可以安排他们的代码，使两个块之间互相保持独立——对于并行执行相当完美。来自于IF-ELSE所有分支的指令，将会沿着译码-执行流水线传递，直到条件性子句求值完毕，设置相关的判定执行标志，从而导致未被选定分支执行的结果被放弃，而来自于正确分支的结果得到确认为止。其间没有流水线清空，当然也就没有访问内存而带来的延迟。这种冒险执行方式的性能优势的重要来源之一是，内存管理的模式更有规律、更可预测。如果需要执行所有的路径，所有相关的缓存行必须读入，编译器可以很容易地处理这种情况，并在需要用到指令和数据之前，将它们从存储体系中移上来。在当前的方案中，任何可能导致性能

损失的预取，都是因为选择了错误的条件路径。但是，读者可能会忧虑这种设计是否会使过多的指令和数据移入高速缓存，并从那里进入CPU，而其中的一些是多余的。尽管这好像效率低下，但是编译器可以安排数据流，降低我们不希望发生的内存未命中事件。不管怎样，执行额外的不需要的指令，丢弃得出的结果，和高速缓存未命中或者更严重的缺页比较起来不算什么。

奔腾处理器装备有执行**指令重排**的电路，作为其依赖处理单元的一部分，目的是让处理器即时地决定编译器生成的指令次序是否还可以改进，以得益于并行（流水线）执行。由于奔腾还将比较复杂的指令拆分成微操作，进一步译码并执行，重排就在这一层面进行。将指令向前移动以获得更佳指令流的行为称为**提升**。IA-64架构的一个重要改动，是将这个决策交由编译器执行，因为编译器对代码的了解更深入，并且有更长的时间可以处理这个问题。最新的安腾2处理器有6条译码流水线（整型、载入/存储、分支和浮点），最大深度为8级。

IA-64编译器还需要注意预读数据的机会，以使操作数在被用到之前进入CPU的寄存器。这叫做**推测性载入**（Speculative Loading）。显然在编译时精确地预测变量的使用拥有更大的优势。但要注意，这可能造成高速缓存与虚拟内存控制器的相互影响，有可能会触发预读页面或高速缓存行的载入，进而可能导致内存异常的发生。如果内存异常和延迟由读取后续不需要的数据项造成，则这种负面效应尤其让人苦恼。为了避免这种损失，IA-64架构提供可以将内存未命中异常推后到专门的CHECK指令到来时再执行的能力。**NAT**（Not A Thing）标志用来表示相关的数据寄存器正在等待延迟的异常得到处理，新的值从内存中到来。通过这些机制，预读动作不会触发不必要的内存异常。这种激动人心的新方式是否能够为所有的应用领域（如数据库检索）带来性能上的提高，依旧有待观察。

IA-64规范的开发，得益于几项不同领域的研究工作。**EPIC**（Explicitly Parallel Instruction Computing，显式并行指令计算）架构的开发，需要CPU电路设计、指令集和编译器特性的通力协作。

安腾处理器中有128个通用寄存器、128个浮点寄存器和128个算术寄存器。这么多的寄存器，可以有效地降低内存访问的频繁程度。总共384个64位寄存器中，128个专门用做通用寄存器和整数运算，还有128个寄存器专门用来存储浮点变量。

和其他寄存器一样，IP也是64位宽，这就将内存的最大限度从当前的4 GB提升到16 EB。尽管桌面PC的物理内存依旧远低于4 GB的限制，但是在文件服务器和数据库引擎中，物理内存已经到达这个限制，4 GB内存的计算机已经十分普遍。作为临时性的措施，Intel将奔腾III的地址宽度拉伸为36位，之后Intel发布了地址总线为64位的至强处理器。

出于实践和学术上的考虑，安腾处理器的指令被归成6大比较宽泛的类型（见表22-3）。这些类型显然与特定的译码流水线相关联。IA-64架构的当前实现并不包含所有6类流水线，而是使用I或M执行A，而X类型的指令由I和B单元负责。这些在将来有可能发生改变。前面提到过，编译器通过每个指令包（见图22-1）内的模板字段，将5位代码传递给流水线调度器，见表22-4。编译器通过这种方式将指令送到正确的执行流水线。

表22-3 安腾指令分类

类 型	
A	ALU运算，整数和逻辑运算
I	多媒体，整数移位，专门的寄存器操作
M	内存载入/存储操作
B	分支、跳转和返回
F	浮点运算
X	特殊指令

表22-4　奔腾2的模板字段编码，图中标出指令间停顿的位置

模　板	指令段0	指令段1	指令段2
00	M单元	I单元	I单元
01	M单元	I单元	I单元‖
02	M单元	I单元‖	I单元
03	M单元	I单元‖	I单元‖
04	M单元	L单元	X单元
05	M单元	L单元	X单元‖
06			
07			
08	M单元	M单元	I单元
09	M单元	M单元	I单元‖
0A	M单元‖	M单元	I单元
0B	M单元‖	M单元	I单元
0C	M单元	F单元	I单元
0D	M单元	F单元	I单元‖
0E	M单元	M单元	F单元
0F	M单元	M单元	F单元‖
10	M单元	I单元	B单元
11	M单元	I单元	B单元‖
12	M单元	B单元	B单元
13	M单元	B单元	B单元‖
14			
15			
16	B单元	B单元	B单元
17	B单元	B单元	B单元‖
18	M单元	M单元	B单元
19	M单元	M单元	B单元‖
1A			
1B			
1C	M单元	F单元	B单元
1D	M单元	F单元	B单元‖
1F	M单元	I单元	I单元

由于安腾处理器将三条指令（41位）并为一个包进行处理，这就使得64位立即数的使用成为可能。在常规的RISC指令集中，这是不可能的，因为指令的操作数字段永远不可能大到能够容纳这么长的数。还记得吗？ARM处理器使用左移位操作将8位立即数扩展成完整的32位常量。另一项由CISC处理器使用的技术是，紧跟在程序代码中指令后面存储大的常量。之后，我们只需简单地使用程序计数器（PC）作为数据指针，使用寄存器间接寻址，可以直接读取这些常量的值。RISC的拥趸不会接受这种对定长指令所做的调整，即使预取单元现在或许有能力处理偶然到达的数据项。由于安腾处理器对指令做出扩展，单条指令可以占据包中两个41位组，它还能够跨两个指令的位置，处理64位的常量。依赖于具体指令的不同，它们在B单元（长分支或调用）或I单元（常量载入）上执行。

模板代码的另一项重要特性是，编译器能够通过它传递指令的互相关信息。在汇编代码中，程序员可以采用两个分号的形式（;;），手工地插入这种信息，这叫做执行**停靠**（stop）。在执行过程中，结构性的停靠向硬件表明，停靠处之前的指令与停靠处之后的指令可能会存在某种形式的资源依赖。21.2节中介绍过如何处理这类代码或数据依赖的问题。推荐的方案包括在指令流中插入多个NOP，重排指令，重命名寄存器和快速转发结果。EPIC安腾处理器架构中，由编译器负责决定在何种情况下使用哪种方法。

22.2 安腾汇编语言：对CPU控制更多

下面我们就以selection_sort.c为例，看看gcc/gas为安腾处理器生成的汇编代码：

```
#include <stdio.h>

/* selection_sort.c A good basic sort routine.
Works by scanning up through the array finding the "smallest" item,
which it then swaps with the item at the start of the scan. It then
scans again, starting at the second position, looking for the next
smallest item...and so on.
*/
int selectionsort(char *pc[ ], int n ) {
  int min, i, j, k;
  char *pctemp;
      for (i = 0; i < n; i++) {
           min = i;
           for(j = i+1; j < n; j++)
               if (strcmp(pc[j], pc[min]) > 0) min = j;
           pctemp = pc[min];
           pc[min] = pc[i];
           pc[i] = pctemp;
           for(k=0; k<n; k++)
              printf("%s ", pc[k]);
           printf("\n");
      }
  return 0;
}
void main() {
  int i;
  char *names[7], *testset[7] = {
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",
      "Sunday"};

for(i=0; i < 7; i++)
          names[i] = testset[i];
printf("\n\nSelection Sort\n");
i = selectionsort(names, 7);
}
```

安腾汇编器的输出，即selection_sort.s，列在图22-3中。这是由gcc/gas套件生成的。注意其中的停靠标记，它们将可能含有互相关的那些指令分隔开来。

尤其需要注意的指令是条件分支——(p6) br.cond.dptk .L19，子例程调用——br.call.sptk.many b0 = strcmp#，以及子例程返回——cbr.ret.sptk.many b0。这些例子均可以在图22-3的清单中找到。相对于IP的条件分支指令显示相关判定标志（p6）——之前设置，目标地址（.L19）——必须是指令包中第一条指令，预取单元提示——sptk（Static Predict Not Taken，静态预测未做），以及建议的高速缓存控制器——many。因此较之以前的CISC或RISC架构，程序员，或者至少编译器，对CPU的控制更多。

```
        .file    "selection_sort.c"
        .pred.safe_across_calls p1-p5,p16-p63
        .section    .rodata
        .align 8
.LC0:   stringz   "%s  "
        .align 8
.LC1:   stringz   "\n"
        .text
        .align 16
        .global selectionsort#
        .proc selectionsort#
selectionsort:
        .prologue 14, 34
        .save ar.pfs, r35
        alloc r35 = ar.pfs, 2, 4, 2, 0
        .vframe r36
        mov r36 = r12
        adds r12 = -48, r12
        mov r37 = r1
        .save rp, r34
        mov r34 = b0
        .body
        ;;
        adds r14 = -32, r36
        ;;
        st8 [r14] = r32
        adds r14 = -24, r36
        ;;
        st4 [r14] = r33
        adds r14 = -16, r36
        ;;
        st4 [r14] = r0
.L2:                                    ; FOR i loop
        adds r14 = -16, r36
        adds r15 = -24, r36
        ;;
        ld4 r16 = [r14]
        ld4 r14 = [r15]
        ;;
        cmp4.gt p6, p7 = r14, r16
        (p6) br.cond.dptk .L5
        br .L3
        ;;                              ; Exit i FOR loop
.L5:    adds r15 = -20, r36
        adds r14 = -16, r36
        ;;
        ld4 r14 = [r14]
        ;;
        st4 [r15] = r14
        adds r15 = -12, r36
        adds r14 = -16, r36             ; get ipntr
        ;;
        ld4 r14 = [r14]                 ;
        ;;                              ;
        adds r14 = 1, r14               ; } incr i
        ;;                              ;
        st4 [r15] = r14                 ;

.L7:    mov r16 = r36                   ; get temppntr
        adds r14 = -20, r36
        ;;
        ld4 r14 = [r14]
        ;;
        sxt4 r14 = r14
        ;;
        shladd r15 = r14, 3, r0

.L6:                                    ; FOR j loop
        adds r14 = -12, r36
        adds r15 = -24, r36
        ;;
        ld4 r16 = [r14]                 ; get npntr
        ld4 r14 = [r15]                 ; get jpntr
        ;;
        cmp4.gt p6, p7 = r14, r16
        (p6) br.cond.dptk .L9
        br .L7
        ;;                              ; Exit j FOR loop
.L9:    adds r14 = -12, r36
        ;;
        ld4 r14 = [r14]
        ;;
        sxt4 r14 = r14
        ;;
        shladd r15 = r14, 3, r0
        adds r16 = -32, r36
        ;;
        ld8 r14 = [r16]
        ;;
        add r16 = r15, r14
        adds r14 = -20, r36
        ;;
        ld4 r14 = [r14]
        ;;
        sxt4 r14 = r14
        ;;
        shladd r15 = r14, 3, r0
        adds r17 = -32, r36
        ;;
        ld8 r14 = [r17]
        ;;
        add r14 = r15, r14
        ld8 r38 = [r16]
        ;;
        ld8 r39 = [r14]
        br.call.sptk.many b0 = strcmp#
        mov r1 = r37
        mov r14 = r8
        ;;
        cmp4.ge p6, p7 = 0, r14
        (p6) br.cond.dptk .L8
        adds r14 = -20, r36
        adds r15 = -12, r36
        ;;
        ld4 r15 = [r15]
        ;;
        st4 [r14] = r15
.L8:    adds r15 = -12, r36
        adds r14 = -12, r36
        ;;
        ld4 r14 = [r14]
        ;;
        adds r14 = 1, r14
        ;;
        st4 [r15] = r14
        br .L6
        ;;
.L11:                                   ; FOR k loop
        adds r14 = -8, r36              ; get kpntr
        adds r15 = -24, r36
        ;;
        ld4 r16 = [r14]
        ld4 r14 = [r15]
        ;;
        cmp4.gt p6, p7 = r14, r16
```

图22-3　gcc生成的安腾汇编代码

```
        adds r17 = -32, r36
        ;;
        ld8 r14 = [r17]
        ;;
        add r14 = r15, r14
        ;;
        ld8 r14 = [r14]              ;
        ;;                           ; } pctemp=pc[min]
        st8 [r16] = r14              ;
        adds r14 = -20, r36
        ;;
        ld4 r14 = [r14]
        ;;
        sxt4 r14 = r14
        ;;
        shladd r15 = r14, 3, r0
        adds r16 = -32, r36
        ;;
        ld8 r14 = [r16]
        ;;
        add r16 = r15, r14
        adds r14 = -16, r36          ; get ipntr
        ;;
        ld4 r14 = [r14]
        ;;
        sxt4 r14 = r14               ; sign extend i
        ;;
        shladd r15 = r14, 3, r0      ; i x 8
        adds r17 = -32, r36
        ;;
        ld8 r14 = [r17]
        ;;
        add r14 = r15, r14           ; build pntr
        ;;
        ld8 r14 = [r14]              ;
        ;;                           ; } pc[min]=pc[i]
        st8 [r16] = r14              ;
        adds r14 = -16, r36          ; get ipntr
        ;;
        ld4 r14 = [r14]              ; get i
        ;;
        sxt4 r14 = r14
        ;;
        shladd r15 = r14, 3, r0      ; i x 8
        adds r16 = -32, r36          ; get pc[] base
        ;;
        ld8 r14 = [r16]
        ;;
        add r15 = r15, r14           ; build pc[i] pntr
        mov r14 = r36                ; get pctemppntr
        ;;
        ld8 r14 = [r14]              ;
        ;;                           ; } pc[i]=pctemp
        st8 [r15] = r14              ;
        adds r14 = -8, r36           ; get kpntr
        ;;
        st4 [r14] = r0               ; zero k
```

```
        .text                        ; Code Section
        .align 16
        .global main#
        .proc main#
main:   .prologue 14, 32
        .save ar.pfs, r33
        alloc r33 = ar.pfs, 0, 4, 2, 0
        .vframe r34
        mov r34 = r12                ; set FP from SP
        adds r12 = -144, r12         ; open stack frame
        mov r35 = r1
        .save rp, r32
        mov r32 = b0                 ; save branch reg
        .body
```

```
        (p6) br.cond.dptk .L14
        br .L12
        ;;                           ; Exit k FOR loop
.L14:   adds r14 = -8, r36
        ;;
        ld4 r14 = [r14]
        ;;
        sxt4 r14 = r14
        ;;
        shladd r15 = r14, 3, r0
        adds r17 = -32, r36
        ;;
        ld8 r14 = [r17]
        ;;
        add r15 = r15, r14
        addl r14 = @ltoffx(.LC0), r1
        ;;
        ld8.mov r38 = [r14], .LC0
        ld8 r39 = [r15]
        br.call.sptk.many b0 = printf#
        mov r1 = r37
        adds r15 = -8, r36           ; get dest kpntr
        adds r14 = -8, r36           ; get src kpntr
        ;;
        ld4 r14 = [r14]              ;
        ;;                           ; } incr k
        adds r14 = 1, r14            ;
        ;;
        st4 [r15] = r14              ;
        br .L11
        ;;
.L12:   addl r14 = @ltoffx(.LC1), r1
        ;;
        ld8.mov r38 = [r14], .LC1
        br.call.sptk.many b0 = printf#
        mov r1 = r37
        adds r15 = -16, r36          ; get Dest ipntr
        adds r14 = -16, r36          ; get Src ipntr
        ;;
        ld4 r14 = [r14]              ;
        ;;                           ; } incr i
        adds r14 = 1, r14            ;
        ;;
        st4 [r15] = r14              ;
        br .L2                       ; Bottom of i loop
        ;;
.L3:    mov r14 = r0
        ;;
        mov r8 = r14
        mov ar.pfs = r35             ; save FP
        mov b0 = r34
        .restore sp
        mov r12 = r36
        br.ret.sptk.many b0
        ;;
        .endp selectionsort#
```

```
.L16:                                ; start of FOR i loop
        adds r15 = -128, r34
        ;;                           ; names[]=testset[]
        ld4 r14 = [r15]              ; get i
        ;;
        cmp4.ge p6, p7 = 6, r14
        (p6) br.cond.dptk .L19       ; test end of FOR loop
        br .L17
        ;;
.L19:   adds r15 = -112, r34         ; get Destpntr
        adds r16 = -128, r34         ; get ipntr
        ;;                           ; local variables
        ld4 r14 = [r16]              ; get i
        ;;
```

图22-3 （续）

```
;;
adds r15 = -48, r34          ; build tablepntr
addl r14 = @ltoffx(.LC2), r1 ; build RAM pntr
;;                           ; using offset+base pntr
ld8.mov r14 = [r14], .LC2
;;
st8 [r15] = r14              ; save strnpntr in table
adds r16 = 8, r15            ;bump pntr
addl r14 = @ltoffx(.LC3), r1
;;                           ; "Tuesday"
ld8.mov r14 = [r14], .LC3
;;
st8 [r16] = r14              ; save strnpntr in table
adds r16 = 16, r15
addl r14 = @ltoffx(.LC4), r1
;;                           ; "Wednesday"
ld8.mov r14 = [r14], .LC4
;;
st8 [r16] = r14              ; save strnpntr in table
adds r16 = 24, r15
addl r14 = @ltoffx(.LC5), r1
;;                           ; "Thursday"
ld8.mov r14 = [r14], .LC5
;;
st8 [r16] = r14              ; save strnpntr in table
adds r16 = 32, r15
addl r14 = @ltoffx(.LC6), r1
;;                           ; "Friday"
ld8.mov r14 = [r14], .LC6
;;
st8 [r16] = r14              ; save strnpntr in table
adds r16 = 40, r15
addl r14 = @ltoffx(.LC7), r1
;;                           ; "Saturday"
ld8.mov r14 = [r14], .LC7
;;
st8 [r16] = r14              ; save strnpntr in table
adds r15 = 48, r15
addl r14 = @ltoffx(.LC8), r1
;;
ld8.mov r14 = [r14], .LC8
;;                           ; "Sunday"
st8 [r15] = r14              ; save strnpntr in table
adds r14 = -128, r34         ; build ipntr
;;
st4 [r14] = r0               ; clear i
        sxt4 r14 = r14               ; sign extend
        ;;
        shladd r14 = r14, 3, r0      ; i x 8
        ;;
        add r16 = r14, r15           ; index Destpntr
        adds r15 = -48, r34          ; get Srcpntr
        adds r17 = -128, r34         ; get ipntr
        ;;
        ld4 r14 = [r17]              ; get i
        ;;
        sxt4 r14 = r14               ; sign extend
        ;;
        shladd r14 = r14, 3, r0      ;ix8
        ;;
        add r14 = r14, r15           ; index Srcpntr
        ;;
        ld8 r14 = [r14]              ; I get next_word
        ;;                           ;I
        st8 [r16] = r14              ; I put next_word
        adds r15 = -128, r34         ; get ipntr
        ;;
        ld4 r14 = [r15]              ; get i
        ;;
        adds r14 = 1, r14            ; incr i
        adds r16 = -128, r34         ; get ipntr
        ;;
        st4 [r16] = r14              ; save i
        br .L16                      ; bottom of FOR
        ;;
.L17:   addl r14 = @ltoffx(.LC9), r1
        ;;
        ld8.mov r36 = [r14],.LC9     ; print banner
        br.call.sptk.many b0 = printf#
        mov r1 = r35                 ; return value
        adds r14 = -112, r34         ; get Destpntr
        ;;
        mov r36 = r14                ; param1=Destpntr
        addl r37 = 7, r0             ; param2=7
        br.call.sptk.many b0 = selectionsort# ; call sort routine
        mov r1 = r35                 ; get return value
        mov r14 = r8
        adds r17 = -128, r34         ; get ipntr
        ;;
        st4 [r17] = r14              ; store in i
        mov ar.pfs = r33
        mov b0 = r32                 ; restore branch addr
        .restore sp
        mov r12 = r34                ; restore SP
        br.ret.sptk.many b0          RETURN to shell
        ;;
        .endp main#
        .ident  "GCC: (GNU) 3.3.5 (Debian 1:3.3.5-13)"
        .section  .rodata            ; Constants Section
        .align 8
.LC2:   stringz  "Monday"            ; constant strings store
        .align 8
.LC3:   stringz  "Tuesday"
        .align 8
.LC4:   stringz  "Wednesday"
        .align 8
.LC5:   stringz  "Thursday"
        .align 8
.LC6:   stringz  "Friday"
        .align 8
.LC7:   stringz  "Saturday"
        .align 8
.LC8:   stringz  "Sunday"
        .align 8
.LC9:   stringz  "\n\nSelection Sort\n"
```

图22-3　(续)

22.3 运行时调试：gvd/gdb

在试图理解陌生的新汇编语言时，使用现代的调试器，通过类似于Microsoft Developer Studio调试器提供的那些功能，对学习的作用是无可替代的。图22.3中列出的selection_sort.c的安腾代码，是在Linux-64系统中使用gcc生成的，以gvd GUI作为前端界面。可以使gdb命令行调试器更易于使用。在图22-4中，selection_sort.c的代码已经为调试而编译完毕，正在单步调试模式下运行。顶部的窗口显示128个通用寄存器，中间的窗口保持C源代码，还有执行指令和断点标记。下面的窗口显示汇编助记符，由反汇编器即时生成。底部的窗口是命令、状态窗口。

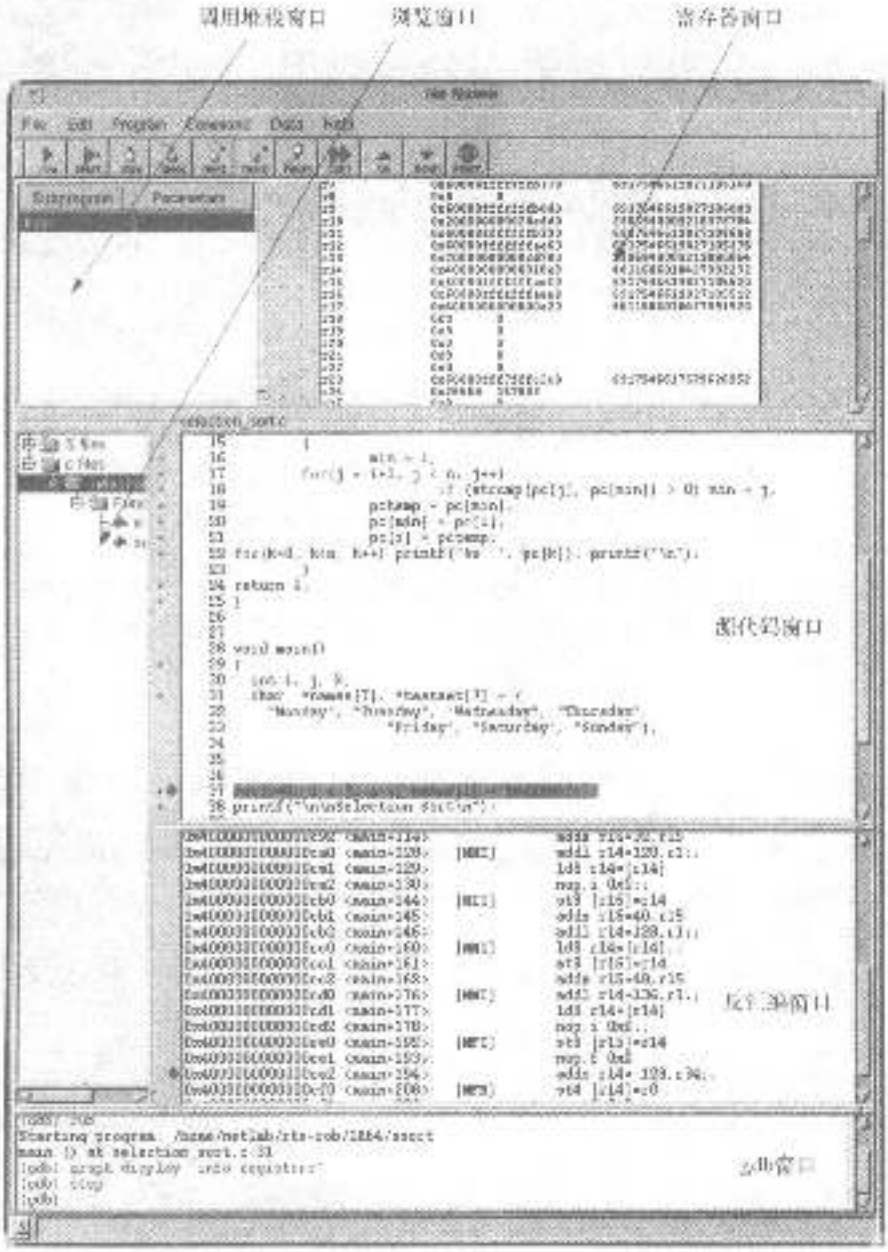


图22-4 在Linux-64下使用gvd/gdb调试器查看安腾代码

22.4　未来的处理器设计

在讨论完不同制造商提供的几种不同的CPU之后，我们可以看出，在过去的20多年间，系统设计人员的选择越来越少。处理器的选择现在仅仅局限于有限的商业处理器。由于制造工厂的启动成本已经上升到数亿美元，因此市场提供的选择范围逐步减少。有关RISC或CISC方案哪个最适合于将来计算机架构的争论，不会由学术界的争论而决定，而是再次依靠市场的选择。在将来，技术上的变革必然会存在许多阻力。两大阵营的基本差异在于将系统的功能分配在哪个架构层次。RISC设计人员将功能性放在比CISC设计人员更高的层次，他们认为语言转换器可以更高效地完成到底层代码的映射。CISC设计人员在底层架构中建立更多的功能性，因为它更快速。

采用RISC技术的外部影响是，采用更小、更简单的器件可以带来开发上的优势。当前，市场对流行十分敏感：新CPU需要在电视黄金时段做广告。它们必须准时摆上商店的货架，以得益于圣诞节的营销活动。而开发复杂的、微码驱动的CISC CPU，与这种快速转变的理念不相符。股东们喜欢RISC!

当前，语义鸿沟的问题通过引入支持工具来解决。集成开发环境、CASE工具和设计人员工作平台，都是为了简化将高级的思想转换成可运行机器代码的工作。RISC CPU的引入虽然加大了语义鸿沟，但这最起码表示至少我们已经克服了一个历史性的难题。正交性是CISC设计人员针对的另一目标，从RISC架构获得统治地位后，已经被束之高阁。现在，人们已经认识到，编译器是设计和开发新型微处理器的重要一环，这和早期的8位处理器形成鲜明的对比，它们当时只提供一个没有经过测试的汇编器。指令集的简单性——RISC先驱们的信条，随每平方毫米芯片上集成的门电路数量的增长，好像也逐渐被人们丢弃。

向后兼容现在成为对新处理器的基本需求，因为客户需要保护在软件上的大量投入。现在不能期望软件能够免费升级以消除这个问题。随着商业化软件的大量发售，这已经成为占主流的市场话题。VLIW处理器的试探性引入连同与之密切相关的编译器，将会对现有的市场产生极大的影响。CPU内部细小的架构性升级——或许是降低信号延迟，可能需要新的编译器，或至少需要对已有的工具集进行参数上的改动。向后兼容将不再是软件的基本特征。

我们还注意到，VLIW程序比等同的传统超标量程序要大得多。这部分是因为不是所有的译码流水线在所有时间都忙，但程序代码中的指令字依旧需要为这些非活动的流水线保留字段，从而会浪费一些空间。

•22.5　小结

- 对于服务器系统，32位处理器4 GB的寻址能力已经逐渐不能满足我们的需要。服务器接下来将会采用64位寻址。
- 随VLIW架构的引入，人们开始重新评价CISC/RISC传统上复杂的硬件译码。VLIW实际上是使用部分编码的机器指令，从而也就不可避免地会宽一些，但它对CPU的运行提供更深入和更灵活的控制。
- 新的VLIW处理器的编译器较之前面的版本更专有化。它们需要了解全部的流水线架构，以及所有指令之间的相互作用。
- 安腾处理器的理念是保持CPU尽可能地忙，为了使CPU尽可能地全速运行，减少停顿，它预取指令并将它们连续发送到正确的流水线。
- 安腾处理器还提供三级高速缓存：L1、L2和L3，它们的大小分别为16 KB、256 KB和9 MB。
- 判定指令集扩展了可用的状态标志，在运行几个并行流水线译码器，这很重要。此外，通过读取和执行所有的分支，可以有效地减少分支预测失败的情形。剔除不需要的结果之后，就是最后的结果。
- 为了减少耗时的内存访问周期，IA-64装备了大量的寄存器。系统的调用—返回堆栈包含在由128个通用寄存器构成的存储体中。

实习作业

我们推荐的实习作业包括比较奔腾处理器和安腾处理器的汇编语言代码。分别在奔腾PC和安腾服务器上编译一个示例程序。如果不想用22.2节给出的例子，可以自己选择合适的程序。

练习

1. 列出VLIW架构的优点和缺点。不但要考虑运行，还要考虑开发。
2. 如果指令有三个参数，总共有127个通用寄存器和50个操作码，那每条指令需要多少位呢？
3. 讨论数字计算机中的代码体系。
4. 解释下面这句话："中断不过是微码中的轮询"。CISC或RISC处理器谁能够更快地做出响应？
5. 阅读图22-3中给出的选择排序例程中的安腾处理器汇编语言代码。你能找出什么地方你可以做得比编译器更好吗？编译器是否恰当地使用寄存器来减少内存的载入/存储操作？它使用判定机制的频率如何？
6. RISC处理器中，根本性的性能增强是什么？EPIC安腾与RISC或CISC架构中的哪一种更接近一些？
7. 你认为处理器芯片上空出的区域最好做什么用？扩大L1高速缓存或扩大CPU寄存器文件是一种好的选择吗？
8. 用奔腾编译器编译并运行selection_sort.c，将汇编输出与22.2节中给出的安腾汇编代码进行比较，哪个更长？
9. 为什么要预取指令？
10. 为什么设计RISC的技术人员要赋予编译器更多责任？

课外读物

- Intel的网站提供IA-64架构的技术信息：
  http://www.intel.com/design/ia64/
- 惠普网站也提供许多有关安腾计算机的详细信息：
  http://www.hp.com
- Evans和Trimper（2003）。

•

第23章 并行处理

高效的并行处理是几代计算机科学家为之不懈奋斗的目标。用户一段时间曾经满足于最快的单处理器提供的性能，但现在超标量CPU和多核CPU已比比皆是。人们依旧热心于研究如何增加处理能力和吞吐量。迄今为止，可用的手段不外乎通过CPU译码流水线实现ILP（Instruction-level parallelism，指令级并行机制），以及大规模集群。总之，成功来自于硬件，问题则出在算法和软件上。多核芯片正在紧锣密鼓的开发过程中，它们针对的是高容量、面向娱乐的应用领域。

23.1 并行处理基础

传统的多任务环境依靠截获应用程序代码的系统调用或中断事件，将控制权返还给操作系统，并重新安排队列中就绪的任务执行。单处理器情况下，所谓*并发*地运行任务和线程，实际上是单个处理器通过快速地在不同的任务和线程之间切换执行。由于人类的反应比计算机要慢得多，从而看起来好像多个任务和线程在同时运行。装备两个或多个处理器的计算机，才能够真正地以并行模式运行，每个处理器都独立地读取和执行它自己的任务代码。

确定应用程序的并行单位或粒度是第一步，也是重要的一步。它可以是机器指令、过程、线程、任务甚至程序。如果拥有足够的资源，所有这些都有可能并行工作，或多或少地提高系统的效能。一段时间，人们达成这样的共识，即大部分现有的计算机程序都以顺序的方式设计和编码，自动将这些程序转换成几路并行的流十分困难。细粒度的任务常常相对较小，粗粒度的任务较大，在到达同步点之前能够以并行模式执行伪独立指令更长时间。将“数据并行”从“代码并行”中区分出来，也对提高系统效能有帮助。前者（数据并行）发生在多个CPU运行同样的代码，引用同一中心数据集的情况下，例如万维网服务器或搜索引擎。后者（代码并行）更为常见，这种情况下，多个处理器执行相关但不相同的任务。

一些应用程序更适合于并行计算，我们可以容易地将它们划分成大小适中的“颗粒”，用于并行执行。计算机生成的影像的渲染，由于所有的图像帧之间相对独立，适合于并行执行。让并行运行的处理器群集同时处理几百帧的渲染工作不是难事。搜索大型的数据集时，我们可以将大的数据集拆分成子集，然后同时在多个子集中搜索，或者制作数据集的多个副本，每个均由并行运行的处理器进行处理。大型的、动态变化的数据库则不容易进行并行处理，但并非绝无可能。数值型的运算经过专门的修改之后，能够适合于专门的算术处理器阵列，但这要求对应用程序的算法、编译器的运作、操作系统活动和硬件能力有较深入的了解，颇具挑战性！

透过周六早上可怕的购物经历，我们可以看到不同层次的并行处理。首先，我们列出购物单，其上是需要从几家商店采购的物品。最简单的情况是爸爸（或妈妈）发动车，然后照着清单，一样样的购买。这是慢速的单处理器模式。但超级市场的收款处最起码是三级流水线：将东西堆在传送带上，逐一扫描价格，然后打包。爸爸可以将所有的待购物品放在手推车中，同时占住几个POS收款处，以足够快的速度将物品分配给各个收银员。最后可能会产生几个账单，整理起来也很费时间，但这样做确实有可能加速结账过程（如果后面的购物者能够容忍），这就是超标量的高效购物。另外，他还可以使用几个手推车，组织手推车内的物品，使之利于最后的结账。这有些类似于多线程，尤其是在别的家庭成员能够接受这种做法，愿意负责额外的手推车的情况下。如果还需要到其他的商店，同样，可以每个商店派一名家庭成员并行地购物，以咖啡馆作为集合点，检查最后的结果，这就类似于使用多处理器。而集群计算或许可以比做访问Amazon和Expedia在线购物。

在理想情况下——所有的代码都可以并行化，加速比为P，即并行处理器的数目：两个处理器会带来2×的加速；10个处理器会带来10×的加速。但实际情况并非如此。Gene Amdahl从20世纪60年代起，就专注于多处理器超级计算机领域的研究工作。他的研究结果表明，在多处理器系统上运行现有的软件时，不能期望能够得到太多的性能提升。

如果一个现有的程序使用N条指令，在单处理器上，它要花$T\tau$秒的时间运行完毕，其中τ是平均指令执行时间。在理想的世界中，当使用P个处理器时，运行的时候会变为$T\tau/P$秒。现在，考虑到实际代码中只有部分代码能够并行化，因此总共的运行时间由两个部分组成，第一部分是串行代码的执行时间，第二部分是能够并行运行的代码所需的时间：

$$\text{总时间}=\text{串行时间}+\text{并行时间}=N\tau f+\frac{N\tau(1-f)}{P}$$

f是问题中由于数据或控制依赖关系必须顺序执行的部分，$(1-f)$是可以并行运行的部分。要注意，$f+(1-f)$的结果为1。P是可用的处理器数。

加速比应该是单处理器运行时间除以（小一些的）多处理器时间，即：

$$\text{加速比}=\frac{N\tau}{N\tau f+\dfrac{N\tau(1-f)}{P}}=\frac{1}{f+(1-f)/P}$$

这就是Amdahl定律，它计算现有算法的并行实现，在给定顺序代码(f)和并行代码$(1-f)$相对比例的情况下，可以达到的最大加速比。根据这个公式能够得到加速比的估计值。它采用乐观的假定，认为处理器间的通信或更复杂的内务活动（housekeeping activities）不会导致时间的损失。这十分不现实，小型的处理器集群常常会花费25%以上的时间，处理与邻近计算机的协同问题。

请记住，f是处理的问题中由于数据或控制依赖关系而必须顺序执行的部分，而P是可用的处理器数目。当f趋向于0时，S约等于P，这种并行算法是完美的。当P趋向于无穷大，S接近$1/f$。这不奇怪，如果f接近1，表明代码中只有极小的一部分$(1-f)$能够并行执行，采用额外处理器得不偿失。

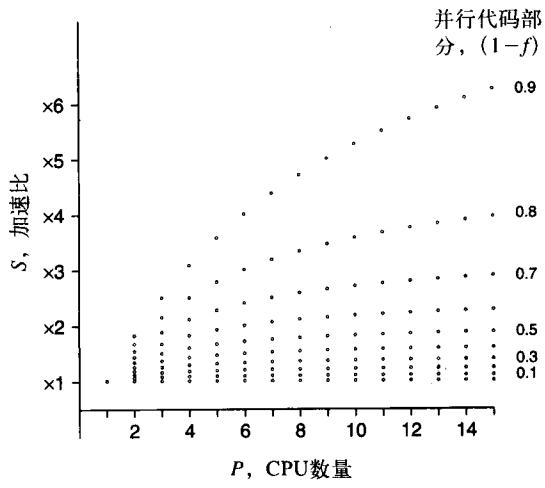


图23-1 Amdahl定律（1～15个处理器，0.1～0.9的适合代码）

当前，程序中能够容易地识别出来的、可以并行运行的代码是有限的，这要归因于程序员常用的一些算法结构。从图23-1我们可以看出，即使15个处理器并行工作，在仅有90%的代码适合于并行计算的情况下，所获得的性能也就在6倍左右。因此，并行硬件能够带来的效能大多数情况下增加一倍左右，和实际处理器的数目无关。吞吐量常常决定于最慢的部分！为了超越这个被挤压的图表，人们建议过在更大尺度上引入**推测执行**（speculative execution）。ARM中的条件指令集和安腾处理器提供的判定指令即为这种方式。推测执行尝试增加可以并行运行的代码，但在这种体制下，所执行的指令总数会增长。

为了发明更有效的配置支持并行运算，人们做了各种各样的尝试，其中大部分可以按照Flynn的处理器分类法进行归类（见表23-1）。每个指令流必须有自己的IP，每个数据流只能由操作数构成。

表23-1　Flynn的处理器分类法

	指令流	数据流		
SISD	单个	单个	个人电脑	字处理
SIMD	单个	多个	向量处理器	地质模拟
MISD	多个	单个	可能不存在	
MIMD	多个	多个	集中式服务器	万维网搜索引擎

大致说来，当前大多数正在运行的并行计算机系统不是单指令多数据（SIMD）流架构，就是多指令多数据（MIMD）流架构。

23.2　指令级并行：流水线化

开发提供指令级并行（ILP）特性的CPU，主要是基于商业上的考虑，即任何通过改变CPU硬件获得的性能提高，会对所有现存代码生效，而非仅仅限于重写过的代码。通过采用串行流水线来承载计算活动，是很多领域中提高性能的可靠手段（参见图13-4），基于明显的原因，人们相信这种策略也适用于新型的RISC处理器。近年来，多级流水线译码技术已经成为能够CPU性能提高的重要部分，通过取自的流水线达成并行的概念，如图7-11和7-12所示，则是RISC处理器运行远快于CISC处理器的主要原因。

23.3　超标量：多执行单元

"超标量"（superscalar）CPU装备多个执行单元。尽管吞吐量有所提高，但也带来一系列的困难。一般地，这样的CPU有一个整数单元、一个浮点单元、一个载入/存储单元和一个分支单元，如图21-6所示。这些单元应该能够独立运行。如果没有数据依赖的限制，它们还能同时运行。由于各种执行单元很少能够在相同的时间内完成它们的动作，因此这里有一项至关重要的需求，就是重新同步来自于不同执行单元的结果，这刺激了动态乱序执行方案的发展。这些可以实现在硬件中，如高腾处理器；或者作为编译器的功能，如ARM和安腾处理器。

23.4　未来的对称，共享内存并行处理

最简单的并行运算方案是，几个CPU共享总线，可以访问同一主存（见图23-2）。通过共享内存，运行在不同处理器上的任务可以交换数据，但共享内存对性能的提高会有负面影响，因为我们在程序中需要对访问进行控制以保护共享内存中的临界数据。在采用抢先式、中断驱动的多任务应用的单处理器系统中，也存在同样的问题，这个问题的解决不是关闭中断，而是锁定总线。一般只有支持多路处理的处理器才提供总线锁定指令，保证所有对临界共享数据的访问中间，不会有其他CPU使用总线。这种控制相当严格，它停止所有的总线通信，其中大部分与当前的临界数据毫无关系。

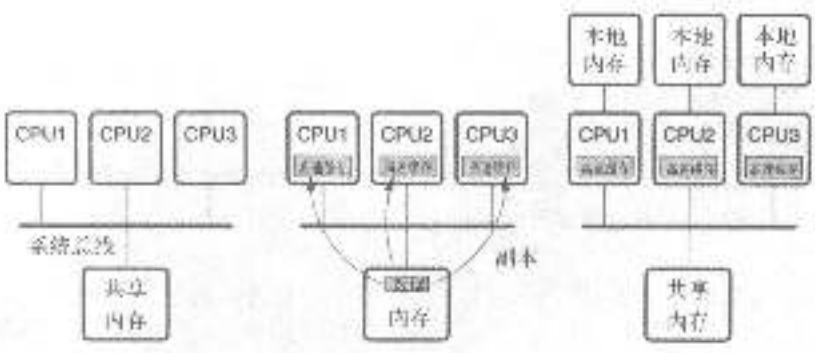


图23-2　总总线、共享内存的多路处理（SMP）

由于每个CPU个体均安装有本地L1和L2高速缓存，问题变得更加复杂。因此，主存中的共享数据经过本地数据读/写操作后，可能在不同的高速缓存中有不同的副本。这就是高速缓存一致性问题。在这种环境中维护高速缓存的一致性极具挑战性。一种流行的硬件解决方案是总线监听(snooping)，即高速缓存控制器监视系统总线上的写周期，将截获的地址与当前保存在高速缓存内的地址进行匹配。如果匹配，则将该行标记为失效，下次读取数据时，会将它清除出高速缓存。随后，任何读操作都会导致对主存的访问。这种策略依赖于所有的高速缓存控制器实现"写穿透"方案，保证当高速缓存中的数据项被当前运行的任务修改后，主存中的本数据会立即得到改写（见表23-2）。这样，其他的高速缓存控制器也能够同时得到数据的更改，即使它们发生在运行于其他处理器上的任务。另一种高速缓存方案是"写返回"，采用这种方案时，经过改写的数据保存在高速缓存中，等到专门的事件发生后，才一次性地将所有更改过的数据写回到主存中。采用这种方法时，高速缓存控制器不知道其他高速缓存中数据的状态，使得数据的一致性成为一个严重的问题。

表23-2 采用写穿透的高速缓存一致协议

高速缓存	事件	动作
读	命中	高速缓存读
	未命中	主存读 高速缓存更新
写	命中	主存写 高速缓存标志为失效
	未命中	主存写

由于制造工艺和方法的改进，电路现在越来越小，从而使得多处理器CPU芯片空得越来越实用，它将几个CPU集成到单个硅芯片上。这重新唤醒了人们对于操作系统层次的对称多处理（SMP）的兴趣。Linux 2.6 SMP内核最多能够处理64个紧耦合CPU的进程调度工作。处理器间通信和同步的方式，带来另一个十分重要的问题。两种解决方法是：共享内存和消息传递。两种方法各有优缺点，提供并放消息传递接口（Message Passing Interface，MPI）的标准化库，已经在大量的并行系统中得到采纳（见图23-3）。

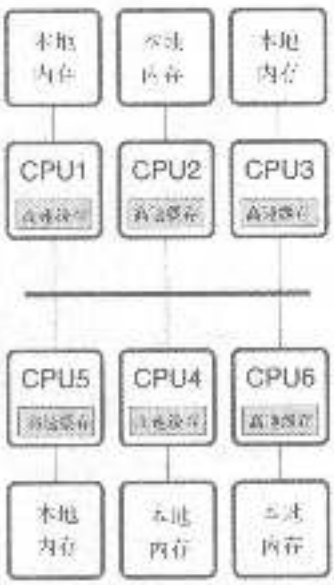


图23-3 MPI系统的硬件配置

MPI可以用在大量不同的并行超级计算机和工作站上，这使得它成为一种乘具弹性、伸缩性很强的选择。MPI库的标准化是它最具吸引力的特性之一，因为有了它，程序员能够基于MPI库调用，生产出能够在任何安装有MPI库的主机上运行的代码。进程之间的交互，只能通过相互间显式地传递消息或数据包来完成。我们可以使用MPI函数调用MPI_Send()、MPI_Recv()和MPI_Bcast()来完成这项工作，后面将会详细说明这些函数。在进程发送消息前，它必须将相关的数据打包后，放在本地内存的发送缓冲区中。类似地，当进程收到传递来的消息时，它必须从声明的本地缓冲区中读取数据。消息传递函数基于底层的TCP/IP socket机制。

需要注意的是，MPI消息发送函数会阻塞，直至消息写成功结束。在C语言中，该函数为：

```
int MPI_Send(void*          sendBuf,
             int            count,
             MPI_Datatype   datatype,
             int            destinationRank,
             int            tag,
             MPI_Comm       comm)
```

其中，sendBuf是用于向外发送数据的传输缓冲区，count是缓冲区的长度，datatype是该缓冲区的数据类型，destinationRank是接收处理器的等级，tag是一种特殊的标识符，可以用来区分相同的两个

处理器上的不同源，comm是通信域。

MPI消息接收也是阻塞操作，在C语言中，该函数为：

```
int MPI_Recv(void*          recvBuf,
             int            count,
             MPI_Datatype   datatype,
             int            sourceRank,
             int            tag,
             MPI_Comm       comm,
             MPI_Status     *status)
```

其中，recvBuf是接收输入消息的接收缓冲区，count是缓冲区长度，datatype是该缓冲区的数据类型，sourceRank是发送处理器的等级，tag是一种特殊的标识符，可以用来区分相同的两个处理器上的不同源，comm是通信域，status是一个指向接收操作成功状态信息的指针。

最后，如果进程想发送一些信息给所有其他进程，我们使用MPI_Bcase命令：

```
int MPI_Bcast (void* buffer,
               int count,
               MPI_Datatype datatype,
               int root,
               MPI_Comm comm )
```

MPI_Bcast()的参数类似于MPI_Send()和MPI_Recv()，在此不再逐一解释。

23.5　单芯片多处理器：IBM Cell

处理器制造商，比如Intel、AMD和Sun，从技术上讲，均具备以合理的成本在单个芯片上集成多个CPU核心的能力。Intel的奔腾4D处理器含有两个CPU核心，其运行频率达到3 GHz，每个核心拥有单独的1 MBL2高速缓存。在双核处理器中，每个核心通过共享的系统总线，同时处理来自于共享主存的指令流。

为了最大限度地利用双核处理器，操作系统必须能够识别多线程，而软件也必须使用多线程机制。并发式多线程（Simultaneous Multi-Threading，SMT）能够让多个线程并行地运行，每个CPU都独立地处理需要完成的任务。没有SMT，软件只能使用一个CPU。

双核处理器不同于多处理器系统。后者装备两个独立的CPU，两个CPU各自拥有自己的资源。而前者资源共享，CPU在同一芯片内。多处理器系统比双核处理器快，而双核系统比单核系统快。

双核处理器的优点之一，就是它们能够装在现有的主板上——只要处理器的插座适合。一般用户，在没有多线程应用软件可用的情况下，只有运行多任务操作系统时，才能体会到性能上的不同。配置多个双核处理器的服务器，当然会在性能上获得惊人的增长。在不远的将来，四核和八核处理器将会很快出现。

IBM、索尼和东芝合作研发新一代多处理器芯片——Cell（见图23-4）。它将成为下一代PlayStation游戏机的中枢，但对于这个行业更重要的，也许是它在处理器架构上的变革是否成功。

Cell芯片含有一个64位、双线程IBM PowerPC核心，以及八个专用的协处理器件（Synergistic Processing Element，SPE），其中的处理器（SPU）提供几种新的特性。

SPU是一个SIMD（单指令，多数据）处理器，它具有向量处理能力。它使用专门的总线接口单元和DMA机制连接到快速的EIB（Element Interface Bus，器件接口总线）。这样，SPE能够通过提供四个128位数据通道的器件接口总线（EIB）和其他SPE以及PowerPC的中枢进行通信。它还连接到共享的L2高速缓存和内存接口FlexIO前端总线控制器。EIP是环形总线，它提供两个回路，能够在相反的方向传输数据。双通道内存控制器连接到最大256 MB的外部Rambus XDR内存（见图23-5）。

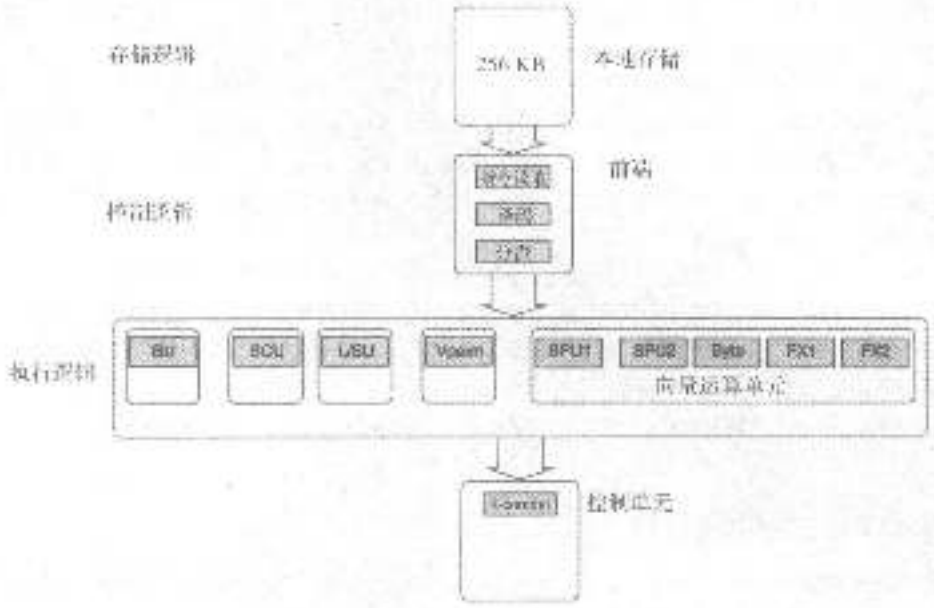


图23-4 IBM Cell处理器的SPE单元

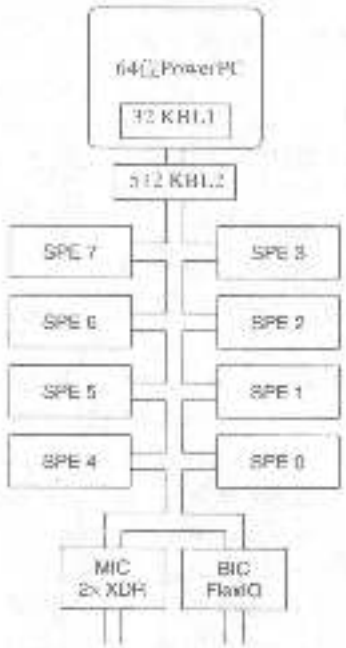


图23-5 Cell架构示意图

IBM Cell芯片的原型运行在4 GHz，理论最大处理能力达到256 GFLOPS（FLOPS，Floating-point OPerations per Second，每秒浮点运算次数），使得它立即成为将来多芯片超级计算机设计的焦点。该芯片采用90nm制造工艺流程，需要大约二亿三千四百万个晶体管。

SPE处理器的结构支持基于SIMD的架构，32位宽的指令的编码采用三操作数指令格式。指令指定最多4个操作数，所有的指令都直接访问128个寄存器，不管是标量还是向量运算，由128个128位寄存器构成的单个寄存器文件处理所有标量、条件和地址运算，如条件运算、分支和内存访问。

芯片上没有高速缓存。相反，它提供快速的本地RAM，访问这些RAM时以16字节为单位（128位每行），而读取32位指令时以32个为一组。

32位浮点运算——最用于多媒体和3D图形应用，只实现了IEEE 745标准的子集。尽管未能完全兼容IEEE对某些用户可能存在问题，但设计人员认为对于目标市场，这并不重要，提供更高的实时性能是最先需要考虑的问题。同时，对于非关键性的设备，饱和算法要比IEEE产生异常的方式更好。由于饱和而导致的图形显示停顿是可以容忍的，但如果由于异常处理而导致显示的帧不完整，图像缺失或视频抖动，则是不能接受的。

SPU CPU不再采用那些复杂的处理，比如乱序处理，寄存器重命名和动态分支预测，而是采用一种十分简洁直观的设计，执行顺序指令由编译器设置，遵循安腾处理器的VLIW方式，不同于奔腾处理器。从而，每周期能够启动两条指令：一条计算指令，一条内存操作。没有动态分支预测，同样依赖于编译器给定应该执行那个分支。

另一个有趣的架构特性是内存访问通过DMA执行。数据传输可以由主控处理器发起，也可以由任何SPU发起，由DMA子系统完成。

23.6　集群和网格：应用级并行

计算机集群是一组松耦合的计算机，它们通过快速的网络一起工作。因而从用户的角度看，它们就好像是单个强劲的计算机。集群一般（但并不总是）通过快速的局域网连接在一起（见图23-6），除了能以少于超级计算机的代价获得更大的处理能力以外，这也是一些要求高可用性应用的需要。高性能的集群（High-Performance Cluster，HPC）的目标是，通过将计算任务分配给集群中不同的节点，达成更高的性能，最常用在科学计算中。为高性能计算而配置的集群，一般由运行Linux和Beowulf中间件的节点构成。这类集群一般运行为利用HPC的并行机制而定制的应用。许多这类程序使用专为在HPC上运行科学应用而编写的库，如MPI。如15.9节所述，Google，运营着世界上最著名的集群。

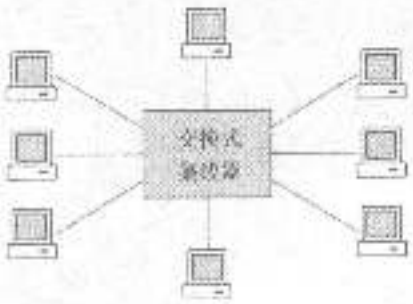


图23-6　星形拓扑集群——交换式集线器，100 MBit/s以太网

从1993年开始，Beowulf集群就使用基于Linux的PC，基于松耦合的网络配置。采用经由快速局域网连接的多台PC主机，提供高伸缩性的效能。采用开源软件时，开发人员能够根据需要调整软件，一般只需要重新配置，而非重新编写，就可以进一步提高性能。构成集群的PC硬件有许多可供选择，而且一般均大规模生产，许多由于未能满足最新的性能标准而被降价。

Beowulf集群基于一组Linux下的函数库，因此，它和标准的Unix网络（通过网络文件系统NFS，和中心密码服务器连接在一起）相差甚远。任何这样的网络，均允许用户登录到任何主机，在不同的机器上执行远程命令，这有些类似于Beowulf系统的本质。Beowulf使用几种分布式应用程序设计环境，在不同的计算节点间传递消息。最常使用的是并行虚拟机（Parallel Virtual Machine，PVM）和消息传递接口（Message Passing Interface，MPI）。它们均以API的形式，提供对一系列库函数的访问，这些库函数会与内核代码打交道。这些程序员编程环境，尤其是PVM，为在混合主机环境下运行而设计，可以减少设置的开销。

另一种选择是使用Mosix，它修改Linux内核，创造更高效、负载均衡的集群。任务可以透明地在系统间迁移，不需要用户显式地干预。更有用的是，服务器和工作站方便地加入和离开集群，不需要完全重新配置。Mosix对于应用程序的程序员完全透明，不需要重新编译和链接到新的库，因为所有这些都发生在内核一级。但是，每个希望加入集群的节点，必须运行相同版本的内核，Mosix在科学和数学计算领域中，拥有大量的应用程序。

最基本的Beowulf可以只有两台主机，之后可以扩展到1024个计算节点。对于它过去的成功以及将来的寿命，很重要的一个影响因素就是系统软件的选择。Linux内核、GNU工具和开发软件，以及标准化的消息传递函数PVM和MPI，使得这种集群具有相当的生命力，因为在更高的软件层次，PC主机可以很容易地升级。为Beowulf编写的应用程序能够在改善后的集群上运行，不管处理器或网络由谁制造。集群中的计算机通过快速的局域网连接在一起，而网格则分布更广，依赖于Internet进行通信。

新近创造出来的术语“网格计算”有几种定义，这造成一些混乱。日内瓦的CERN成立了研究中心，是万维网的诞生之处，也是开发网格计算的关键机构，当初开发网格计算，是作为一种共享计算机处理能力和数据存储资源的服务（见图23-7）。

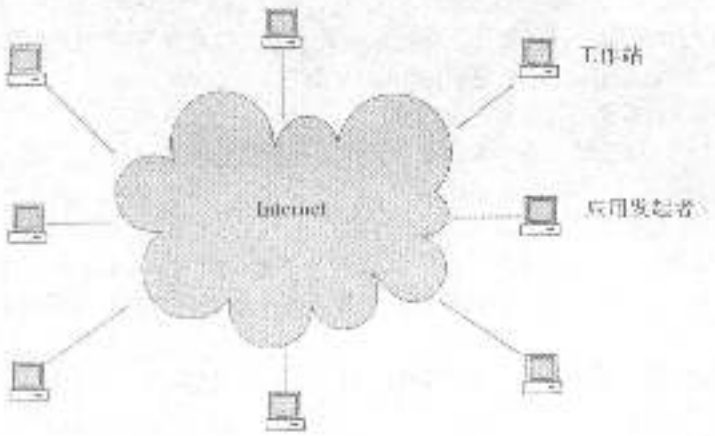


图23-7 网格计算

当用户和资源分布在地理上比较广阔的区域时，网格计算就变得具有吸引力。特别地，一些自发性的组织和那些专注于开放系统的人，已经接受了网格计算的概念。对于那些有复杂的计算需求，需要尽最大可能利用昂贵资源的大型商业企业来说，它也颇具吸引力。或许最有名的例子就是SETI项目，它尝试使用Internet将空闲的桌面计算机利用起来，进行是否存在外星文明的科学研究。网格计算也为其他许多组织所用，完成的任务包括蛋白质合并、药物研究、基因调查和气象建模。这些项目依赖于PC拥有者的协助，在机器中安装专门的屏幕保护程序。这样，每台PC都可以在空闲的时候处理一部分数据。比如，网格计算提供一种分布式模型，可以利用那些被闲置的PC的处理能力，解决需要极大计算量的问题。

23.7 小结

- 实现高效并行处理的目标是达到更大的吞吐量，更快得出结果，选择运行并行运算的粒度和具体的应用相关。
- Amdahl定律表明使用常规的软件——仅有一部分能够并行运行，购买和安装更多的CPU并不能带来期望的性能改善。
- 当前，对于并行处理贡献最大的就是ILP流水线。
- 采用多处理器时，处理器间的通信和结果的同步，要么使用共享内存，要么使用点对点消息传递。
- 为了降低内存的延迟，高速缓存在计算机中到广泛的应用，但当多个处理器访问相同的数据时，它们存在一致性的问题。
- MPI库和MOSIX Linux内核修改版已经成为当前最流行的多处理方式。
- Intel和IBM都在研发多核芯片，IBM的Cell芯片针对索尼的PlayStation。

- 人们对于计算机集群和松耦合计算网格的兴趣增长很快，部分是为了提高性能，部分是应对设备故障。

实习作业

建议进行学期末回顾总结。

练习

1. 列出并行架构的优点和缺点。在考虑管理和运行时阶段的同时，也要考虑开发。
2. 为什么并行程序那么难写？
3. 查看Transputer的历史，这是一种32位的微处理器，其中为MPP系统设计了通信通道。
4. 如何确定应该在哪一层实现并行功能，或者多级并行是否为更好的选择？
5. 增强型流水线或超标量是否是利用ILP的最佳方式？
6. RISC处理器根本性的性能增强在哪里？
7. 你认为扩大L1高速缓存或CPU寄存器文件，是利用处理器芯片上空闲空间的更好方式吗？
8. VLIW（Very Long Instruction Word，超长指令字）架构的优点是什么？
9. 为什么指令预取有优点，但也有风险？
10. Barroso等著（2003）根据他们对Google搜索引擎的研究工作声称：我们的测量结果表明，现代处理器中的乱序执行和推测执行，已经开始产生负面效应。处理器的架构师们会不会关注这种批评呢？
11. 如果某个应用在你的双处理器系统中获得了1.5×的加速比，如果安装八个处理器，同样的应用会得到什么样的加速比呢？

课外读物

- Intel的网站提供多核CPU的技术信息：
  http://www.intel.com/
- AMD公司的网站是：
  http://www.amd.com/
- Beowolf Linux集群的介绍：
  http://www.beowolf.org/overview/index.html
- Cell架构的细节：
  http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html

一些基本的分析和讨论：

- Stone（1993）。
- Vrenios（2002）。
- Vrenios（2006）。
- Sloan（2004）。
- Pacheco（1997）。
- Foster（2000）。
- Barroso等著（2003）。

附录 Microsoft Visual Studio 8 Express版

1. Microsoft VC++ Developer Studio：关于安装

本书第1版的随书光盘上提供Microsoft Developer Studio 6的学生版。现在，微软提供版本8的Express版，供个人免费下载使用，许可条件限制用户只能将该软件用于教育和评估用途，商业用户依旧需要购买常规的许可。下载和安装需要一些时间才能完成，最多可能需要1.5 GB的磁盘空间，要依是否包括MSDN的相关部分而定。因此，事先要有心理准备，即使通过宽带，下载和安装也可能需要1小时左右的时间。下载网站的URL是（见图1）：

http://msdn.microsoft.com/vstudio/express/visualc/

在线帮助系统十分重要，尽管通过网络访问它也可以，但最好还是安装在本地，不管您是初学者，还是有经验的程序员。微软将所有的文档都统一放到**MSDN**（Microsoft Developer Network，微软开发人员网络），并使之与HTML浏览器兼容。因此，我们可以使用Internet Explorer或Netcape Navigator阅读它。安装好Developer Studio之后，任何时候都可以使用浏览器查看这些资料。微软网站上提供更多的相关信息：

http://msdn.microsoft.com

在开始安装之前，首先检查机器是否有足够的磁盘空间，并且当前用户是否具有管理员权限。如果没有足够的访问权限，安装会在下载完成后被阻塞。我们强烈建议大家使用不同的登录ID和密码，来区分系统管理员和用户/程序员，以避免一些恶性灾难的发生。

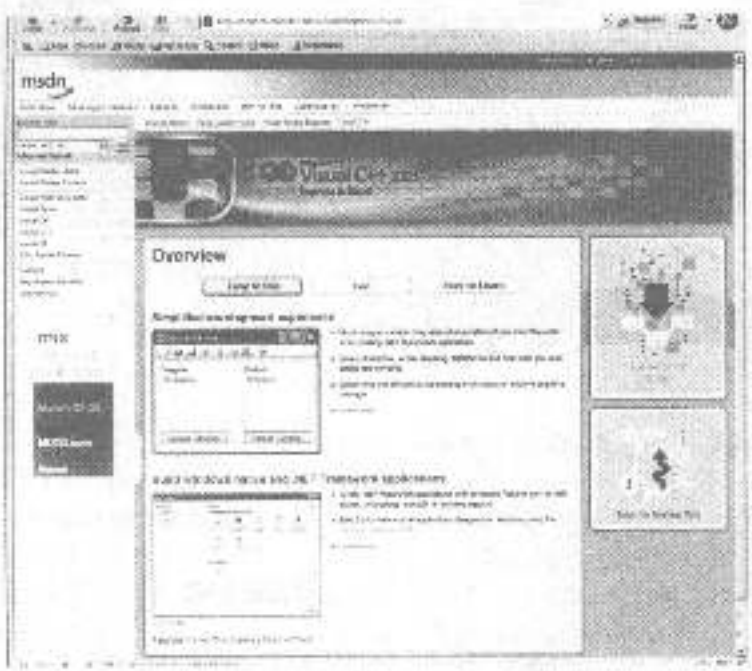


图1 微软下载网站：http://msdn.microsoft.com/vstudio/express/visualc/

2. 安装VC++ Developer Studio 8

打开计算机，以管理员身份登录。如果需要，关闭其他应用程序，检查一下，确保之前安装的

所有Microsoft Developer Studio VC++已经被删除。在桌面上，单击My Computer（我的电脑）图标，然后在C:驱动器图标上单击鼠标右键，检查剩余磁盘空间。所需的空间从360 MB到1.5 GB不等，依用户选择安装哪些组件而定。然后，将微软的URL输入到Internet Explorer或Netscape浏览器中。

开始下载（见图2），暂时先不要管安装向导其余的部分。下载需要花些时间。我下载时大约花了90分钟。该向导允许用户选择特定的目标文件夹进行安装，默认的位置是C:\Program Files\Microsoft Visual Studio 8\Common。在Windows 2000和Windows XP上，向导完成安装工作之后，不需要重启系统，就能够开始使用该软件。

最后，我们就可以使用下面的序列访问Developer Studio：[Start]→[Programs]→[Visual C++ 2005 Express Edition]，或者在文件夹

```
C:\Program Files\Microsoft Visual Studio 8\Common7\IDE
```

中直接点击VCExpress.exe。我们可以通过点击并拖拉VCExpress.exe执行文件到桌面上，创建启动Developer Studio的快捷方式。这样，以后启动该程序时会容易些。如果需要，我们还可以修改快捷方式的名字。

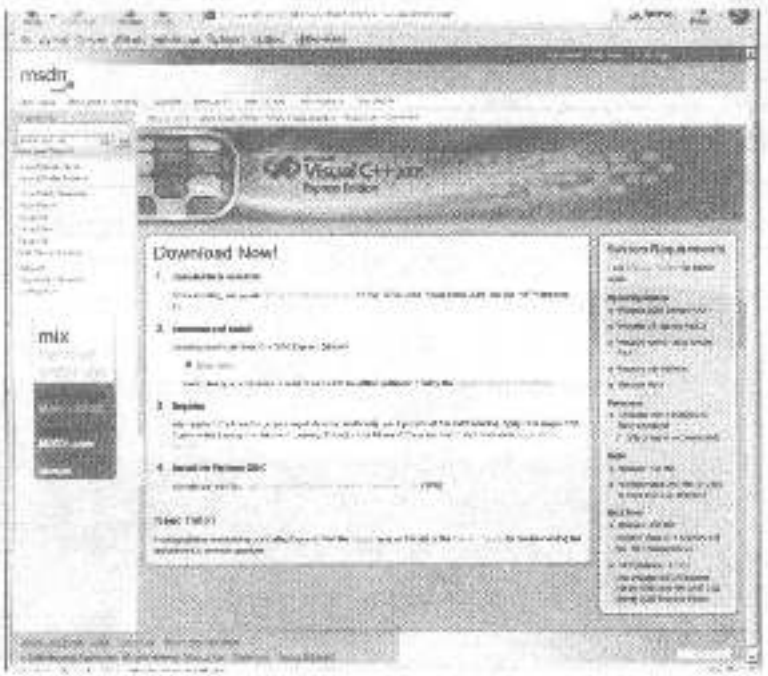


图2　下载面板

3. Microsoft Developer Studio：创建简单的C程序

Microsoft Developer Studio是一个集成开发环境（Integrated Development Environment，IDE）。它不但提供编辑器、编译器和链接器，还通过工作区和项目支持大规模的系统开发，其中内嵌程序生成工具以协助编译和链接。在开始编写代码前，必须组织硬盘上的目录结构，好让它们接受你的工作。在组织目录时，要预先仔细计划，估计自己将会需要多少项目，仔细考虑将要使用的项目名，永远不要使用test.c，prog.c，final 1.c，在做实验的文件夹中记录笔记，写下作业在硬盘上的位置。如果将目录共享，那么你就有可能错误地调试了别人的同名的程序，浪费了宝贵的时间（Unix中的确有一个系统程序叫做test，它在/bin目录下，初学者要注意区分）。

点击桌面的图标，或是使用[Start]菜单选择程序的名字，可以启动Developer Studio。

如果使用的是大学的网络，Developer Studio程序可能存储在中心服务器计算机上，这样所有共

中的软件可以保存在单个磁盘上，使得管理和维护更为简单。在这种情况下，安装Developer Studio的驱动器可能会是F:或G:或H:，而非一般的C:。

初始的屏幕（见图3）没有什么用处，但其中列出的一系列文章比较有用，可以有选择地阅读。遵循微软程序一贯的做法，对于每个按钮按钮，鼠标在上面停几秒钟后，就会出现一小段相关的解释文字。有时候会显得有些多余，同样的动作，也可以通过下拉菜单或是键盘快捷方式，或是单击菜单栏上的图标来完成。依用户的个人喜好而定！

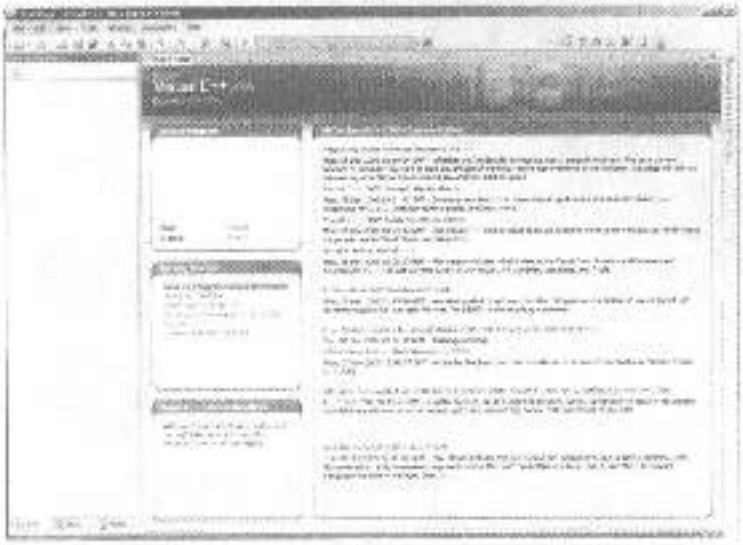


图3 Visual C++ Developer Studio的欢迎屏幕

开始键入C代码之前，需要先创建新的项目（project）。从Developer Studio的主菜单中选择[File]—[New]，创建新的项目，确保[Project]标签在最上层，然后选择[Win32 Console Application]，这样我们就能够编译和运行简单的，类DOS的基于文本的应用程序。在开始时，最好不要冒进，直接去开发完全的Windows应用程序，因为那会涉及许多复杂性。Developer Studio会提供新项目的选项设置窗口（见图4）。我建议在没有掌握这种开发环境之前，暂时先选择[Empty Project]（见图5）。



图4 开始新项目

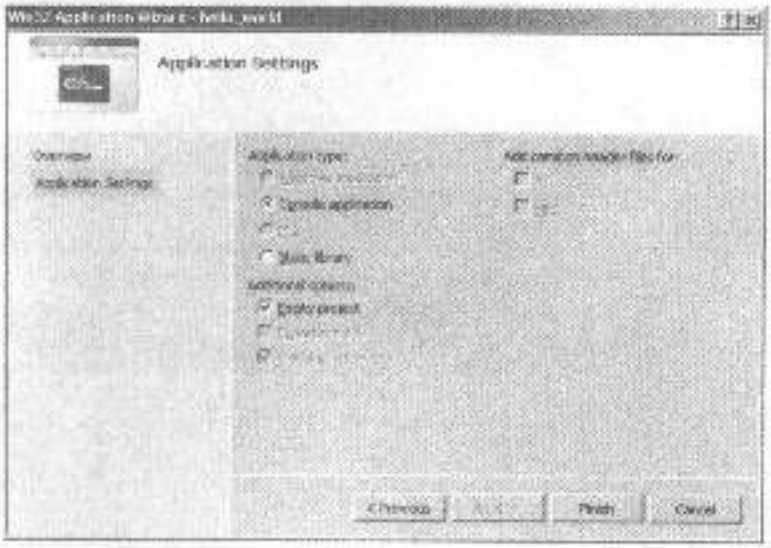


图5 控制台应用程序的选项

Developer Studio要求用户为每个项目赋予一个名字，这对于后来的引用比较重要，因此请花些时间，想出一个不容易与之前或之后将要开发的项目同名的名字。从图6中，我们可以看到项目的名字被用来建立新的子目录，检查是否为新项目输入了正确的目录和文件夹。在250 GB的磁盘上，很容易丢失自己的代码！我们可以使用它提供的默认目录。但总是使用默认目录会造成混淆和混乱，你也可以使用类似于图6的目录结构来组织自己的工作。

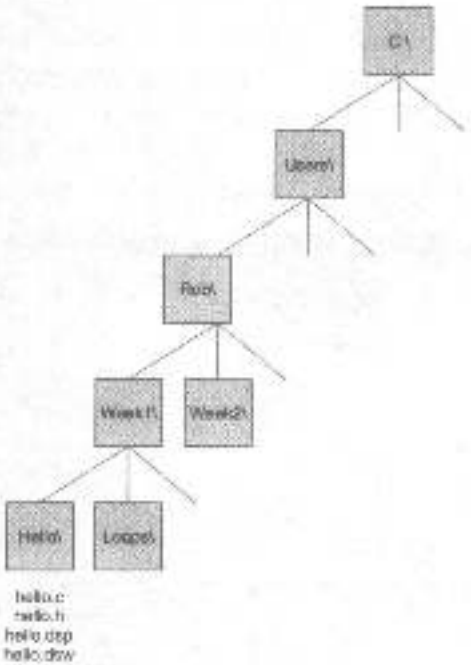


图6 项目目录的结构示例

定义完项目之后，就可以使用编辑器打开新文件，进行代码编辑工作。从Developer Studio主菜单中，选择[File]→[New]，确保[File]标签在最上层，选择[C++ Source File]并保证[Add to Project]被选

中。同样，还要检查目标位置是否正确，之后输入程序的名字，以.cpp为扩展名（见图7）。如果不是在开发C++代码，不要使用.cpp扩展名。如果你忽略这个建议，链接时会产生奇怪的连接错误。

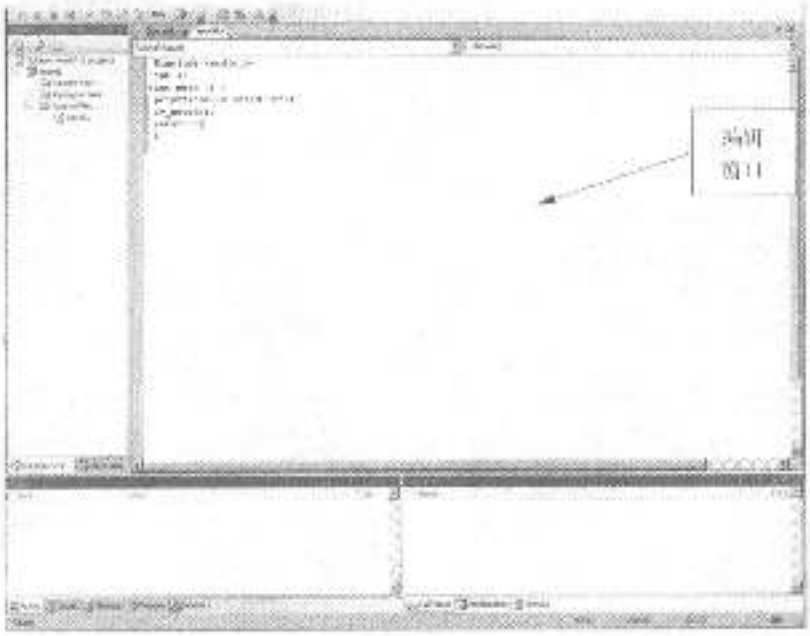


图7 第一个项目

输入下面的传统C程序：

```
#include <stdio.h>

int main(void)
  {
      printf("Hello World!\n");
      return 0;
  }
```

或者，如果喜欢C++，可以使用等价的代码：

```
#include <iostream.h>
int main(void)
  {
      cout << "Bonjour tout le monde\n";
      return 0;
  }
```

但在命名这个源文件时，要记得使用.cpp扩展名。定义完项目并命名文件之后，编辑器出现在屏幕的右方。这个编辑器很棒（尽管我最喜欢的编辑器还是emacs），它提供帮助功能，并在需要时可以提供集成调试信息。主界面共有三个显示窗口，代码编辑窗口是不言而喻的，边上是视图窗口，它可以提供三种不同的显示内容：ClassView、FileView或InfoView，最后一个视图提供对在线文档的访问。ClassView以类的方式显示当前的工作区，在开始使用C++时，这会比较有用。FileView以文件的方式显示当前的工作区。底部是输出窗口，显示编译器和链接器输出的结果，它还用在调试过程中。

顺便提一下，[Home]将插入点移到行首，[End]将插入点移到行尾。

按下鼠标左键可以选取文本，之后可以使用[^X]剪切、[^C]复制或[^V]粘贴等选项。在

Developer Studio内，还有大量的键盘快捷方式可供使用：

[^S]	保存文件
[F7]	编译文件
[F4]	跳到编辑窗口中下一处错误
[F7]	生成（编译+链接）程序
[^F5]	运行程序
[F9]	设置断点
[F10]	单步调试，不进入函数
[F11]	单步调试，进入函数
[F1]	得到标记项的帮助

第一次使用Developer Studio时，在线帮助功能（见图8）是使用者印象最深刻的特性之一。将光标放在关键字并按F1键，信息窗口中会立即显示出一些有用的帮助信息。

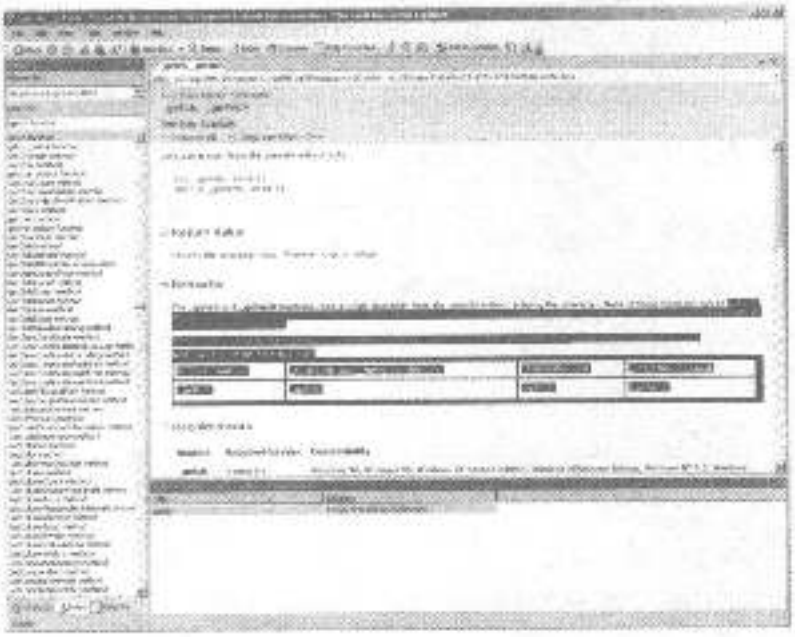


图8　帮助窗口

如果只想编译程序，检查语法，可以选择工具栏的图标或[^F7]。如果存在错误，按[F4]会将焦点返回到编辑窗口，并将光标停在含有下一个错误的行。

完整的编译和链接，即生成，同样使用工具栏或键盘快捷方式[F7]来完成。要运行编写的代码，按[^F5]或选择[=>]图标。要调试运行，则按[F5]或[F10]，或选择[>]图标。

输出必须保持足够长的时间，这样用户才能读到最终的消息（见图9）。一种方式是等待用户输入一个字符后再结束程序，使用 n= _getch()可以完成这项功能。为了避免警告消息，我们需要在程序的顶部添加#include <conio.h>。



图9　控制台程序的屏幕输出

为了能够感受一下Windows程序设计，可以试试下面给出的例子。开始一个新的项目，选择[File]→[New]→[Project]，将项目的类型设为

[Win32 Application]，检查项目的命名和位置。然后开始新文件[File]→[New]→[File]，选择[C++ Source]，将文件命名为hello.cpp，编辑代码，并生成程序：

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE a, HINSTANCE b, LPSTR c, int d)
{
    MessageBox(NULL, "Hello World!", "WSM", MB_OK);
    return NULL;
}
```

作为练习，我们也可以通过命令行编译和链接这个程序（C>提示符）。通过[Start]按钮菜单启动命令行提示符，转到项目所在目录：cd c:\temp\rob\wsm。使用dir命令检查一下是否能够看到源文件hello.c，然后使用下面的命令编译并链接它：

```
C:\Temp\Rob\WSM > CL hello.c
```

CL的意思是Compile & Link，也可以使用小写的cl。遗憾的是，上面的命令不一定立即就能使用，因为有关编译器和链接器的环境变量可能尚未正确设置，设置环境变量最简单的方式，是运行Developer Studio提供的批处理文件，然后再来运行CL命令：

```
C:\Temp\Rob > C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat
```

无论是在Unix世界还是在PC世界，通过头文件引入相关数据和函数声明，为库的链接做准备的做法都很常见。因此，如果链接失败，报告未解析的外部引用（unresolved externals）错误消息，其原因可能是：

1）未包括正确的头文件，如：#include <winbase.h>。

2）未将正确的库插入到链接清单中，如：FABLIB.lib。

3）PATH中未给出库目录的正确位置，如：C:\Program Files\VisualStudio\VC98\lib。

在用到WIN32系统调用，如Sleep()时，我们需要知道应该使用哪个头文件和库文件。最简单的方式是，使用帮助系统的搜索功能，查找涉及的关键字，或相关的主题。如果在搜索或索引面板中找到该系统调用，则用鼠标左键点击系统调用。这会调用相关的主题页，给出函数行为的简短描述。单击[Quick Info]按钮，我们会立即看到纯文本形式的必要的头文件和库文件（见图10）。



图10 第一个Windows应用程序

如果程序通过编译和链接，但却不按照我们预期的方式工作，则需要使用调试器！如果要运行运行期间的调试，必须在编译代码前设置相应的编译标志。

随Visual Studio一同提供的调试器很不错，它提供大量有用的特性，帮助我们解决运行时的错误，监视各种处理器对代码执行读取、执行周期的过程。第一次启动调试器时（见图11），使用[F10]，一些重要的显示面板可能没有打开。此时，我们首先要保证调试工具栏可见。如果没有，则在窗口右上的空白区域，关闭X按钮下面，鼠标右键单击。这会调起一个菜单，Debug（调试）和Build（生成）应该已经激活。再后打开主菜单的Tools项，选择列表底部的Customize（定制）。选择Commands标签并在左侧的类中选定Debug。现在，你就可以选择和拖放右侧的栏中任何的项到Debug工具栏上。我建议选择下面这些项：

Break all（全部中断）；
Disassembly（反汇编）；
Memory（内存）；
Output（输出）；
Registers（寄存器）；
Run to Cursor（运行到光标）；
Start/Continue（开始/继续）；
Stop Debugging（停止调试）；
Watch（监视）。

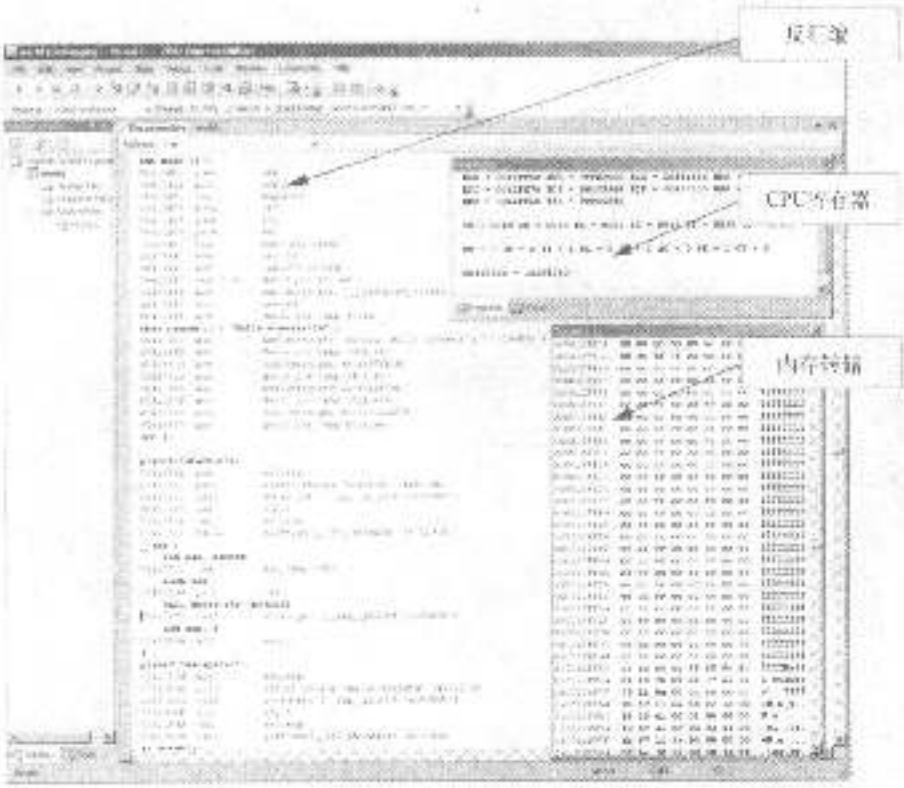


图11　使用调试器

术 语 表

10BaseT：10 Mb/s以太网标准，它使用星形拓扑结构、双绞线以及中心交换机或集线器。

Absolute path name（绝对路径名）
完整的文件访问地址，如/projects/rob/book/ch_01或F:\users\staff\rob\hello.c。

Access time（访问时间）：存储设备响应请求的时间间隔。

Accumulator（累加器）：基本的数据寄存器，在CPU中用于变量的算术运算或其他通常的操作。奔腾处理器中为EAX。

ACL（Access Control List，访问控制列表）：与文件或目录关联的用户清单，还可以包括具体的权限（读、写、执行）。所有其他的用户都被排除在外。

Active（活动）：当前处理器正在执行的任务（进程）的状态。

ADC（Analogue to Digital Converter，模数转换器）：用来将模拟信号接入到数字计算机。

Address（地址）：计算机内标识具体内存存储单元的惟一的名字或数字。

Addressing mode（寻址模式）：用来形成访问内存中或CPU寄存器中数据项的地址的方法。

Address register（地址寄存器）：保存数据项在内存中的地址的CPU寄存器，如mov [EDI],0。

Address unit（寻址单元）：处理器中计算指令和数据项地址的硬件单元。

Address width（地址宽度）：构成地址的位数。

ALE（Address Latch Enable，地址锁存允许信号）：一种总线控制信号，表示什么时候地址总线上是合法的地址。

Algorithm（算法）：解决某种问题的一系列指令。

ALU（Arithmetic Logic Unit，算术逻辑单元）：CPU内执行逻辑和算术指令的硬件电路。

AM（Amplitude Modulation，振幅调制）：一种传输信号的技术，它将信号叠加到载波的振幅（电平）上。

Analogue（模拟）：使用对应系统中的连续值来表示一个值的表达方法，例如，使用电压来表示温度。

API（Application Programming Interface，应用编程接口）：一系列可供应用程序使用的库函数，通过它们可以完成各种活动，如图形、进程调度、Internet访问。

Application program（应用程序）：能够完成用户请求的任务，而且和计算机系统的管理没有直接关系的计算机程序。

Arithmetic Logic Unit，参见**ALU**：美国标准信息交换代码（American Standard Code for Information，ASCII）：使用7个二进制位（有时为8位）表示的基本字符集。

Assembler（汇编器）：一种程序，可以将汇编助记符语言文件转换成由二进制代码构成的等价机器程序。

Assembly language（汇编语言）：一种计算机语言，由助记符和标签写成，它们与机器指令之间存在一对一的对应关系。

Associative memory（关联内存）：一段内存，这段内存的访问不通过连续的数字地址，而是通过预先定好的惟一标识符。

Asynchronous bus（异步总线）：没有同步主时钟的总线。

Asynchronous handshake（异步握手）：使用由接收方向消息发送方返回的确认信号，来协调和控制数据传输的一种技术。

Asynchronous transmission（异步传输）：没有共享时钟的串行通信。接收方依靠不定期的信号代码来重新同步（相位，而非频率）它的本地时钟，如START位或SFD位。

ATM（Asynchronous Transfer Mode，异步传输模式）：一种数据传输协议，使用较短的固定长度数据包，即信元，来传输数据。

Availability（可用性）：由于故障或预先计划好的维修任务，系统的关键功能不能为用户所用的时间比率。

Backward channel（反向通道）：与基本数据流方向相反的数据通道。它的速率可能会很低，

以全双工模式运行。

Bandwidth（带宽）：通道能够用来传输数据的频率范围。通道有最大和最小截止频率，比如常规电话线路为300～3000 Hz。在处理数字通道时，常常会将带宽表示成b/s（比特率，每秒位数）。

Barrel shifter（桶形移位器）：ALU中的一种电路，用来横向移位和旋转二进制模式。

Base address（基地址）：数据结构的起始地址，比如数组中第一项的地址。

Baseband transmission（基带传输）：一种传输系统，信息在编码成合适的波形后，就直接通过通道发送，不使用高频调制技术。

Batch file（批处理文件）：含有操作系统外壳命令的文件。

Batch processing（批处理）：数据处理的一种传统方法，任务排队运行，不允许用户在线交互。在现在的工资和账单事务中，还可以见到这种系统。

Baud rate（波特率）：符号通过通道发送的速率。

Beam scanning（电子束扫描）：控制电子束的横向移动，在屏幕上绘出一条扫描线。

big-endian：多字节整数的一种格式，它将低字节（Least Significant Byte，LSB）存储在高位，高字节（Most Significant Byte，MSB）存储在低位。Motorola和Sun采用这种格式。

Binary（二进制）：以2为基的计数系统，只使用符号0和1。

BIOS（Basic Input-Output System，基本输入输出系统）：一系列控制数据输入输出的例程。常常和引导例程一起存储在主板上的EPROM中。

Bit（位）：二进制数字：0或1，是最小的信息单位。

Bit pattern（位模式）：用来表示数据或指令的一系列的位。

Bit rate（比特率）：二进制位跨通道传输的速率。单位为每秒多少位（b/s）。

Block checksum（块校验和）：一种错误检测方法，这种方法使用原始数据块计算一个检验数。在经历过风险之后，再次计算这个值，如果最初的校验和与后来计算的值相一致，则可以判定没有错误发生。

Blocking（阻塞）：当请求的数据尚未到达时，暂停程序运行的一种技术（有时是需要的）。

Bootstrap（引导程序）：一种程序，在执行时载入更大、更复杂的程序，然后将控制权转移给载入的程序。

BPL（Branch Prediction Logic，分支预测逻辑）：为提供正确的指令预取，在CPU内，用来猜测条件分支（IF-ELSE）会走哪条路径的技术。

Bridge（桥接器）：用来链接两个相似局域网的硬件设备，完成两个局域网间的数据包传递。

Broadband signalling（宽带信号传输）：使用载波调制技术进行数据传输，如AM、FM或PM，甚至三种技术的结合。

Broadcast routing（广播路由选择）：将数据传输给所有侦听者，假定只有指定的侦听者会读取其中的数据。

Buffer（缓冲区）：内存中的一段数据区，用来临时性地存储数据项。可以用它来平衡源和目的地之间速度上的差异。

Bundles（指令包）：由三条41位奔腾指令，以及用来组织并行执行的相关判定值和编译器模板代码构成的128位长字。

Bursting（突发）：快速连续地传输几个数据项，从而使所有数据项都不再需要单独的设置过程的行为。

Bus（总线）：用来连接各个单元的一系列电子线路。这种连接是公共的，而不是点对点的。

Byte（字节）：八个二进制位。

Cache（高速缓存）：为加速内存访问而提供的快速本地存储器。

Cache write-through（高速缓存写穿透）：一种数据高速缓存策略，将所有被改写的数据项立即输出到基本内存。

Cardinal（基数）：一个正整数。

Carrier（载波）：一种通过通信通道发送的连续信号（常常为正弦波）。信息通过改变其内在的参数（频率、振幅或相位）作用其上。在接收时，信息可以通过解调制从载波中提取出来。

CCITT（国际电报电话咨询委员会）：现为ITU-T（国际电信联盟-电信标准部）。

Cell（信元）：ATM包。

Central Processing Unit（CPU，中央处理器）：任何计算机的核心部件，如奔腾、SPARC。

C flag（C标志）：一种CPU条件代码，表示前次运算中发生算术进位。

CGI（Common Gateway Interface，公共网关接口）：

可执行程序和HTML网页之间的一种接口标准。

Channel（通道或信道）：信息通信的路径。

CHS（Cylinder Head Sector，柱面、磁头、扇区）：使用物理位置编号进行磁盘数据寻址的方案。

Circuit switching（电路交换）：网络中的一项技术，它在通话开始时在发送方和接收方之间建立一条物理通道，并一直保持到通话结束。电话即基于这种方式。

CISC（Complex Instruction Set Computer，复杂指令集计算机）：如MC68000。

CLI（Command Line Interface，命令行接口）：提供用户接口的程序，如C外壳。

Client（客户机）：客户机-服务器系统中的请求方。

Client-server（客户机-服务器）：一种计算范例，它将工作负载划分到客户端软件（常常存储在工作站中）和较大的服务器计算机中。它起源于X窗口系统和Unix。

Clock pulse（时钟脉冲）：用来同步大量不同设备活动的定期脉冲信号。

Cluster（簇）：由几个磁盘扇区组成的数据块，主要是为了访问更方便。

Codec（编解码）：电话应用中使用的ADC/DAC，常常提供非线性压缩功能。

COM1，COM2：PC串口。

Combinatorial logic（组合逻辑）：不涉及存储设备（即没有触发器）的数字逻辑电路。

Command register（命令寄存器）：IO芯片中的初始化寄存器，它们决定IO芯片执行什么动作。

Common channel（命令信道）：几个数据信道共享的信号传输信道。

Compiler（编译器）：一种程序，它将HLL指令代码文件转换成等价的机器指令。

Conditional branch（条件分支）：程序控制的跳转，或转移——依据单个或多个条件标志的状态。

Condition code（条件代码）：表示运算结果或状态的单位标志。

Control statement（控制语句）：影响程序执行路线选择的语句，如IF-ELSE。

Control Unit（CU，控制单元）：CPU中负责执行读取-执行周期的重要组成部分。它负责协调计算机的全部活动。

CPU：参见中央处理器（Central Processing Unit）。

CRC：参见循环冗余校验（Cyclic Redundancy Check）。

Critical resource（临界资源）：有可能被几个任务或执行线程同时访问（从而有可能发生错误）的数据项或设备。

CRT（Cathode Ray Tube，阴极射线管）：电视或非平板监视器中的显示器件。

CSMA/CD（Carrier Sense，Multiple Access/Collision Detect，带冲突检测的载波侦听多路访问）：以太局域网采用的访问协议，以太网没有主控仲裁器，每台主机的优先级相同，它的运行依赖于网络上每台主机的配合与协作。

CU：参见控制单元（Control Unit）。

Cycle time（周期）：完成重复性序列的单个部分所需的时间段。

Cyclic Redundancy Check（CRC，循环冗余校验）：一种校验和技术，它使用模2除法，不但能够检测出错误，而且还可以标识出受到影响的位。

Cylinder（柱面）：在多磁碟磁盘部件中，半径相同的磁道称为柱面。

DAC（Digital to Analogue Converter）：数模转换器。

Database（数据库）：一系列围绕公共主题组织起来的信息。

Datagram（数据报）：一种包递送服务，随数据包的到达进行选路，没有预先的路由规划。UDP即为这类服务。

Data register（数据寄存器）：用来保持数据的寄存器。如果它在CPU中，则临时性地保存变量或中间结果。

Debugger（调试器）：协助诊断运行时错误的工具。它们一般提供CPU寄存器和活动内存区段状态的详细信息。

Decoder（译码器）：一种逻辑电路，根据输入的编号选择输出中的某个线路。

Dedicated polling（专注式轮询）：一种编程技术，代码完全由状态检查循环构成。

Demand paging（请求式页面调度）：一种虚拟内存方案，仅当实际需要时，才将代码段（或页）或数据段（或页）从磁盘调入主存中。

De Morgan's Theorem（德·摩根定律）：通过调整相关的反相器，可以互换电路中的AND和OR门的定律。

DIMM（Dual In-line Memory Module，双边接触内存模组）：含有168针插件的一块小型电路板，其上是几个DRAM存储芯片。当前最大

容量为128 MB。

Direct addressing（直接寻址）：指令中直接提供有效地址的寻址模式。

Directory（目录）：链接文件名和文件的数字化地址的表，可以用来访问数据。

Dirty bit（“脏”标志位）：表示数据项已被更改的状态标志。

Disassembly（反汇编）：一种将二进制机器代码逆向转换回汇编助记符的方法，有助于调试工作。

Disk capacity（磁盘容量）：磁盘可以保存的数据量，单位为字节。

DLL（Dynamic Link Library，动态链接库）：目标代码库模块，载入内存后可供任何程序使用。

DMA（Direct Memory Access，直接存储器存取）：数据直接通过计算机总线从一个部件传送到另一个部件。一般在数据IO时使用。在过程结束之前，中央处理器不参与传输。

DNS（Domain Naming Service，域名服务）：Internet分布式数据库服务，它将域名转换成IP地址。

Dongle（软件狗）：插在计算机端口上的小盒。它含有存储代码编号的设备，可以鉴别程序的许可状况。

Download（下载）：从远程计算机获取信息。

DRAM（Dynamic read-write Random Access Memory，动态读写随机访问内存）

Driver（驱动程序）：和硬件设备打交道的软件，一般是操作系统的一部分。

DSL（Digital Subscriber Line，数字用户线路）：新型的高速调制解调器设备，具备150 kb/s到8 Mb/s的数据传输能力。

DSP（Digital Signal Processor，数字信号处理器）：一种专为分析复杂波形而设计的CPU，如语音分析。

DTMF（Dual Tone Multi-Frequency，双音多频制）：电话按键信号发送标准。

Duplex channel（双工通道）：双路同时传输。任何时间均双向发送。

Dynamic linking（动态链接）：将应用程序代码与目标库的链接延迟到运行时才进行。

ECC（Error-Correcting Code，错误纠正编码）。

Echo（回送）：将接收到的信息送回发送方以进行检验。它也是一种简单的流量控制手段。

EDO（Extended Data Output，扩展数据输出）：允许突发传送几个相邻的数据项以加速平均访问时间的DRAM。

Effective address（有效地址）：用来访问内存中数据的最终地址。

EIDE（Extended Integrated Drive Electronics，增强型集成驱动电路）：硬盘最新的接口标准。

emacs：由Richard Stallman的GNU实现的全功能程序员编辑器。

Email（电子邮件）：TCP/IP套件的流行应用。它使用SMTP（Simple Mail Transfer Protocol，简单邮件传输协议）在“邮局”间发送邮件。流行的用户前端软件包括Eudora、Nupop、Elm和PINE。

Embedded processor（嵌入式处理器）：专门针对特定控制任务的处理器，常常位于相关的设备上，如复印机、引擎控制器件、自动贩卖机等。

Encyrption（加密）：对消息进行安全编码的过程，只有授权的阅读者才能访问到纯文本形式的数据。

Environment pointer（环境指针）：在堆栈上保存的前一帧的指针，使用它可以访问到前一作用域内的变量。

EPLD（Erasable Programmable Logic Device，可编程逻辑器件）：一种可重新配置的逻辑电路。

Ethernet（以太网）：DEC/Intel/Xerox为局域网制订的标准。它使用CSMA/CD调度访问。

Ethernet switch（以太网交换机）：中央集线器或交换机，它将几个以太网网段互连起来，常常使用10BaseT电缆。

Exception（异常）：CPU中断的总括。

Executable program（可执行程序）：含有CU能够识别并执行的机器指令的文件。

Executive（执行程序）：操作系统的简单形式。实时执行程序对于时间敏感型的应用比较有效。

Expansion slot（扩展槽）：用于外接各种插卡的槽。

Facsimile，fax（传真机）：一种传输纸面文档的方法，它扫描页面，将每个点阵或图形元素（像素）编码。一般还使用高级的数据压缩技术。

Failover（故障切换）：将发生故障的器件切换成替代设备的过程。

FAT：参见文件分配表（File Allocation Table）。

Fdisk：MS-DOS磁盘工具，它可以对磁盘分区，安装主分区引导记录（Master Partition Boot Record，MPBR），在执行高级格式化之前进行。

FET（Field Effect Transistor，场效应晶体管）：一种电子开关元件，在栅极引出线加上微弱的电压，就能够控制其他两端间的电流。

Fetch-execute cycle（读取-执行周期）：由CPU连续执行的一系列动作，将下一条指令读入到CU中并执行。

FIFO（First In，First Out，先进先出）：数据队列的一种。

File Allocation Table（FAT，文件分配表）：由一系列磁盘数据块编号构成的数组，它表示分配给文件的数据块。未分配的数据块用NULL来表示。

File browser（文件浏览器）：一种实用程序，它组织目录，并将其可视化，同时还可以完成一般的文件处理任务。

Firmware（固件）：保存在PROM中的程序。它们常常与硬件的低级操作有关。

First come，first served（先到先服务）：一种十分简单的调度策略。

Flip-flop（触发器）：一种时钟驱动的同步锁存器，它可以作为存储一个二进制位的内存单元，也可以作为频率等分器（halver）。

Floating point（浮点）：一种数字方案，用来表达和操作拥有小数部分的数字，如3.123。浮点数存储和操作采用的格式是IEEE 754。

Floating-point unit（浮点单元）：专门处理浮点运算的ALU。它可以是CPU的一部分，也可以单独安装。

Flow control（流量控制）：在数据到来太快，来不及处理时，由接收方用来控制数据传输的一种方法。

FM（Frequency Modulation，频率调制或调频）：一种信号传输方法，它将信号施加到载波的频率上。

Forward Error Correction（FEC，前向错误纠正）：不需重传就能在接收端纠正错误的技术。它需要原始消息中含有冗余信息。

FPU，参见浮点单元（Floating-Point Unit）。

Fragmentation（碎片）：可用空间中被划分成很小单位的部分，它们很难被再次利用。

Frame base pointer（框架基址指针）：CPU地址寄存器，用来表示当前保存局部变量的堆栈框架的位置。奔腾处理器使用EBP。

FSB（Front Side Bus，前端总线）：来源于Slot I/II接口的一个术语，现常用来指代主板的时钟速度，如66或100 MHz。

FSK（Frequency Shift Keying，频移键控）：调制解调器中用来使用不同的音调传输二进制1和0的技术。

FTP（File Transfer Protocol，文件传输协议）：TCP/IP协议的一种应用。FTP用来从远程主机下载文件。匿名FTP服务器先于WWW出现。

Full adder（全加器）：三端输入、二端输出的加法电路。

Function（函数）：有返回值的子例程。

Garbage collection（垃圾回收）：用来将空闲内存收集在一起的技术。

Gateway（网关）：一种用来链接局域网的硬件设备，它可以使用包的目的IP地址做出路由决定。

Ghostview：PostScript屏幕预览程序。

Gopher：基于菜单的客户机-服务器信息检索系统。使用Gopher浏览器可以连接到Gopher服务器。Gopher早于WWW出现。

GPRS（General Packet Radio Service，通用分组无线业务）：为蜂窝移动电话提供的170 kb/s的文本传输机制。

Granularity（粒度）：子单元的大小。

GSM（Groupe Spécial Mobile de Conférence Européene des Administrations des Postes et des Télécommunications，全球移动通信系统）：数字蜂窝网络。

GUI（Graphical User Interface，图形用户界面）：WIMP界面。

Half adder（半加器）：二端输入、二端输出的加法电路。

汉明编码：在数据字内使用多个奇偶位检测错误的编码技术，它能够标示被破坏的位。

汉明距离：从一个合法数字转到另一个合法的数字需要改变的位数。

Hardware（硬件）：构成计算机的电路或硬件器件。

Hexadecimal，hex（十六进制）：四个二进制位缩写成1位的表达方式。它使用数字0～9和A～F。用它来表示较长的二进制地址更方便些。

High-Level Language（HLL，高级语言）：为解决

问题而设计的语言，与机器架构无关，如COBOL、C、C++和Java。

HLL，参见高级语言（High-Level Language）。

Host（主机）：计算机的另一个名称。

HTML（HyperText Markup Language，超文本标记语言）

HTTP（HyperText Transfer Protocol，超文本传输协议）

Hub（集线器）：交换式以太局域网中心的互连设备，可能使用10BaseT的连线。

Hypertext（超文本）：通过链接引用其他文档的文本，使用这些链接可以跨网络从文档各自所在的主机下载这些文档。所使用的协议是HTTP。

ICMP（Internet Control Message Protocol，网际控制报文协议）

IDC（Insulation Displacement Connector，绝缘位移连接器）：一种不需要焊接的带状电缆连接器，该连接器有尖锐的长钉，可以刺破绝缘层，接触到其中的铜线。

IDE（Integrated Device Electronics，集成设备电子部件）：一种硬盘接口标准。

IEEE 754：浮点数存储和操作的标准。

Immediate operand（立即数）：一种寻址模式，常数值直接和指令保存在一起，使用指令指针地址直接访问。

Indexed register addressing（带索引的寄存器寻址）：一种寻址模式，它使用地址寄存器和偏移访问内存变量。

Indirect addressing（间接寻址）：有效地址已经在寄存器中，常常为地址寄存器，它指向内存中的数据。

Initialization（初始化）：设备和数据结构正处于准备状态的预备阶段。

IO（Input-Output，输入-输出）：用来将外部设备接入计算机总线，使数据能够传入或传出计算机的接口硬件。

Input-output port（输入-输出端口）：一个可寻址的存储单元，可以用它访问硬件设备，常常要经过字节宽度的锁存器。

Input port（输入端口）：一个可寻址的存储单元，可以用它访问硬件设备，有时要经过字节宽度的锁存器。

Instruction（指令）：由操作码、地址信息构成的位模式，用来指定处理器需要执行的操作。

Instruction hoisting（指令提升）：一种编译器技术，它可以改变程序指令的次序，以改善代码的运行时间。

Instruction pipeline（指令流水线）：现代CU使用的多阶段译码，是RISC处理器的特性之一。

Instruction pointer（指令指针）：CPU内的地址寄存器，它含有下一条需要执行的指令的地址。在奔腾中为EIP，有时也称做程序计数器。

Instruction register（指令寄存器）：CPU中保存正在执行的指令的寄存器。

Integer（整数）：或正或负但没有小数部分的数字。

Integrated circuit（集成电路）：完全的电子电路，分布在硅晶片上，可能含有数千个缩小的晶体管。

Interface circuit（接口电路）：允许计算机与外部设备交换数据的电路。

Intermittent polling（间歇式轮询）：一种程序设计技术，让代码定期检查状态值。

Internet（因特网，也作互联网）：全球性的互连网络。

Interpreter（解释器）：一种程序，它从其他文件中读取指令，一次一条，根据读入的指令选择合适的动作执行。

Interrupt（中断）：一种信号，它强制CPU临时性地暂停当前的程序，执行与中断源关联的指定例程（ISR）。

Interrupt-driven IO（中断驱动的IO）：一种程序设计技术，它使用中断信号将数据传入或传出计算机。

Interrupt Service Routine（ISR，中断服务例程）：由硬件中断信号或专门的软件TRAP指令调用的子例程。

Interrupt vector（中断向量）：ISR标识数字，一般由中断设备返回，让CPU选择正确的ISR。

IP（Internet Protocol）：网际协议。

IRQ（Interrupt ReQuest，中断请求）：PC提供的15条中断线。

ISDN（Integrated Services Digital Network，综合业字服务网）：全数字电话系统的一种标准。

ISP（Internet Service Provider，因特网服务提供商）：提供通往Internet主干线链路的组织或机构。

ISR：参见中断服务例程（Interrupt Service Routine）

ITU/TSS：CCITT的新名称。

IVR（Interrupt Vector Register，中断向量寄存器）：保存中断向量的寄存器，外设可以通过它向

CPU标识自身的ISR。

IVT（Interrupt Vector Table，中断向量表）：保存所有ISR地址的内存数组，CPU使用它来响应中断请求。

JANET（Joint Academic Network，联合科研网）：英国的学术研究网络，提供Internet访问。

Java：近年来兴起的面向对象的HLL，对于Internet应用尤其有吸引力。

Kernel（内核）：操作系统最核心的代码。负责进程调度、中断处理和内存管理。

Label（标签）：程序中用来标识指令的名称。它是符号地址。

LAN：参见局域网（Local Area Network）。

Latch（锁存器）：由交叉耦合的NOR或NAND门构成的存储单元。与触发器密切相关。有时用来指字节宽度的设备。

LBN（Logical Block Number，逻辑块编号）：用来访问磁盘数据块的值。

LCD（Liquid Crystal Display，液晶显示器）

Level 1 cache（1级高速缓存）：最接近CPU的高速缓存。

Level 2 cache（2级高速缓存）：紧随在1级高速缓存之后的高速缓存。

Library procedure（库过程）：保存在库中的预先转换完毕的过程，可以直接链接到程序中，执行一些标准的基本处理任务。

LIC（Line Interface Card，线路接口卡）：连接电话线路的电路。

LIFO（Last In，First Out，后入先出）：一种堆栈数据结构。

Linker（链接器，链接程序）：用来将编译后的代码与所需的库例程、任何外部数据或者编译过的子例程结合到一起的工具程序。

Linux：Unix的一个变种，最初由Linus Torvalds编写，并在爱好者之间免费分发。它现在已经扩展到商业领域，并迅速流行起来。

LINX（London Internet eXchange，伦敦Internet交换中心）

Little-endian：多字节整数的一种格式，它将高字节（Most Significant Byte，MSB）存储在高位，低字节（Least Significant Byte，LSB）存储在低位。Intel和IBM采用这种格式。

LIW（Long Instruction Word，长指令字）：用来减少译码延迟的一种CU技术。

Load address（载入地址）：执行程序时，程序被载入到内存中的地址。

Loader（载入程序）：将可执行程序从磁盘载入到主存中的系统程序。

Local Area Network（LAN，局域网）：最大不超过几公里范围的网络，如以太网，连接计算机和类似的设备。局域网使用的典型路由技术是广播含有目的地标识的报文。

Localization（本地化）：使用参数指定子例程的精确动作，使之更适合于调用者的需求。

Local variable（局部变量）：在函数内声明的位于堆栈上的数据项，退出函数时会丢失。

Logical address（逻辑地址）：理想化地址空间中的地址集，在程序执行之前需要映射到物理内存。

Logical block（逻辑块）：读写磁盘驱动器的数据单位。

Logic gate（逻辑门）：使用几个晶体管实现一种逻辑功能（如AND、OR或NOT）的电路。

Login/logon（登录）：提供用户ID和密码，证明自己为系统合法用户。

LOOK：硬盘控制器使用的局部请求调度算法。

Loopback（回送）：将通道回送给自身的一种技术，用于测试。

LRU（Least Recently Used，最近最少使用）：操作系统需要载入新数据时，判断应该牺牲哪个页面或缓存行时使用的一种准则。

MAC（Media Access Control，介质访问控制）：以太网接口最下面的软件层。

Mainframe（大型机）：大型、昂贵的计算机，常常安装在安全的中枢地点，提供大量的内存和磁盘存储容量，常常以批处理模式运行。

Main memory（主存储器或主存）：存储可执行程序及数据，供处理器在读取-执行周期中访问的存储器。

make：协助组织和记录程序编译的实用程序。

makefile：为make实用程序准备的命令文件。

man pages：由Unix提供的一系列在线帮助。

MPBR（Master Partition Boot Record，主分区引导记录）：硬盘的第一个扇区，其中含有磁盘和分区的状态信息。这些信息由fdisk写入。

MC68000：Motorola 68000 32位CISC微处理器。

Memory address register（内存地址寄存器）：在总线传输期间，用来临时性地保存内存单元地址的地址寄存器。

Memory buffer register（内存缓冲区寄存器）：在

总线传输期间，用来存储数据的寄存器。

Memory direct addressing（内存直接寻址）：通过保存在程序指令中的地址直接访问内存中内容的寻址模式。

Memory management（存储器管理）：计算机内数据移动的规划和实现。

Memory segment（内存段）：为了安全原因，可以将内存划分成特征化的块，如代码段、堆栈段。

MAN（Metropolitan Area Network，城域网）：覆盖城镇或城市范围的网络。

Microinstruction（微指令）：CPU内的一个微程序指令，CU使用它来译码和执行当前的机器级指令。

Microprogram（微程序）：由微指令构成的程序，CU用它来执行构成处理器指令集的机器级指令规定的动作。

Minicomputer（小型计算机）：中等大小的计算机，适合于25个人左右的部门。

Mnemonic（助记符）：短小的字母序列，可以由汇编器转换成关联的数字机器代码。

Modem（调制解调器）：将计算机（数字）连接到电话系统（模拟）进行数据传输的接口设备。

Moore's Law（摩尔定律）：由Intel的Gordon Moore总结的经验公式，用来描述VLSI容量的不断增长。

Motherboard（主板）：容纳CPU、主存和总线互连线路的大型电路板。

Mount（挂接）：Unix命令，用来将磁盘存储设备插入到文件体系结构中。文件系统还可以通过NFS从远程主机导入。

Multiplexer（多路复用器）：将几个独立的通道合并成单个高带宽通道的设备。

Multisync monitor（多重扫描监视器）：可以运行在不同扫描频率的显示监视器。

Nameserver（域名服务器）：将网络名转换成IP编号的一种工具。本地域名服务器遇到不知道的域名或地址时，会将请求转发给域中下一个更高的级别，直到找出相关信息为止。

Nesting（嵌套）：同时处理几个函数调用的能力，指函数间的调用。

Network（网络）：一组互相连接的计算机。相互之间的连接一般为公共的通道，而非点对点连接。

Network address（网络地址）：网络通信中用来标识主机的惟一的主机名或编号。

NFS（Network File System，网络文件系统）：由Sun公司推出。这是一种虚拟磁盘系统，它使用TCP/IP协议，允许网络上的计算机共享文件和磁盘空间，为用户提供无缝的单一文件系统。

NIC（Network Information Center）：网络信息中心。

Non-volatile memory（非易失性存储器）：在断电后能够保持其内容的存储设备，它可以使用备用电池。

NTFS：Windows NT中的文件系统。

Object module（目标模块）：由编译器生成的文件，其中含有可执行的机器码，但在执行前需要将其他一些必要的例程链接进来。同时，有可能还要注意地址引用问题。

Offset（偏移）：指令或数据项距离模块或数据结构起始地址的字节数。

Online（在线）：通过远程登录到大型主机，访问特定计算机功能的方法，如旅行查询。

Operating system（操作系统）：扩展硬件提供的功能，帮助所有的用户更好地使用计算机的软件。硬件提供商常常将操作系统与硬件一同提供。具体的例子是Unix和Windows NT。

Operation code，opcode（操作码）：一种数字代码，表示CPU应该执行的处理任务。

Optical fibre（光导纤维）：非常纤细的实心玻璃或塑料管，低功率的激光信号可以通过它传输，在远程端重新拾取出来。光的频率非常高（~10^{27} Hz），在使用光作为载波时，调制速率可以很快。

Order of magnitude（数量级）：10的幂。例如1000（10^3）比10（10^1）大两个数量级。

Overflow（溢出）：溢出指的是算术运算的结果导致符号错误。例如，如果使用2的补码进行运算，两个大正整数相加可能会得到负值。

PABX（Private Automatic Branch Exchange，专用自动交换机）：客户的电话交换机，负责为分机号码的呼叫建立通路。一般为计算机控制，全数字化。

Packet（包）：含有相应报头信息（包括目的地址和内容指示）的数据块。

Packet switching（包交换）：基于包交换的网络（PSS）在发送数据包时，不建立永久性的连接。路由选择是在每次转发时完成的。每个

中间节点都将数据包全部接收并保存下来，直到转发至一下个节点为止。

PAD（Packet Assemble and Disassemble，数据包组装/分解设备）：PSS接口设备。

Page（页）：虚拟内存分配的单位，一般为4 K字节。

Page table（页表）：联系物理和逻辑（虚拟）地址的查找表。

Parallel adder（并行加法器）：可以对两个二进制字做加法或减法的电路。

Parallel port（并行端口）：一种IO端口，宽度常常为单个字节，如打印机端口LPT1。

Parallel transmission（并行传输）：数字信息通过平行线路（如总线）同时发送。

Parameter（参数）：为子例程预先初始化好的局部变量。由调用者负责设置它们的初值。

Parity bit（奇偶位）：附加在数据上的额外冗余位，用来保证数字中0或1的个数或为奇或为偶。这样，读取者就能够检查单个位发生错误的情况。但是，这种方法不能检测出同时发生两个错误的情况。

Partition table（分区表）：保存磁盘驱动器上布局信息的数据结构。

Path name（路径名）：文件在磁盘上的逻辑地址。

Peer-to-peer（对等网络）：对等网络指分层系统中对等层之间的软件交互。

Pentium（奔腾）：Intel i386系列CISC处理器的最后成员。其中引入了一些RISC特性。

Peripheral（外部设备或外设）：连接到计算机端口的外部设备，如打印机、调制解调器。

Phosphor（荧光粉）：监视器中使用的一种发光化学制品。

Physical address（物理地址）：数据在内存中的真实地址。

PIC（Peripheral Interrupt Controller）：外设中断控制器。

Pipeline processor（流水线处理器）：一种内建多级译码RISC线路的CPU，可以有效地支持指令的并行执行。

Pixel（像素）：图片的单个独立元素。

Polling（轮询）：询问某个设备，看它是否就绪，可以执行某项动作。

Polynomial（多项式）：曲线的数学表示。

POSIX（Portable Operating System Interface，可移植操作系统接口）：一套国际公认的API或操作系统调用。它基于Unix已经提供的功能（而非Windows）。目标是为程序员提供一套标准，使得代码跨不同平台的移植更容易。Unix成为标准接口的基础，是因为它并不为单个提供商所有。此外，由于Unix存在几个不同的变种，因此对于开发一套标准的Unix系统调用有强烈的需求。

PostScript：由Adobe公司开发的页面描述语言，在激光打印机中广泛使用。

Preamble（前同步码）：数据传输之前发送的一系列标志位，接收方可以用它与发送方同步。

Predicate flag（判定标志）：条件性标志。

Predication（判定）：根据条件性标志控制所有机器指令的执行。

Pre-fetch buffer（预取缓冲区）：小型的芯片内高速缓存，可以预先将指令读入其中，以保持CU 100%地工作。

Prioritization（优先化）：按照优先次序调度例程的执行。

Procedure（过程）：函数或子例程。

Process（进程）：载入到主存中可以执行的程序，也称为任务。它可以和其他进程同时运行。

Process coordination（进程协调）：对任务进行同步，使它们能够共同完成更大活动的能力。

Processor（CPU，处理器）：计算机系统的基本硬件部件，它可以执行机器级的指令，并且能够完成逻辑和算术运算。

Program counter（程序计数器）：也称为指令指针。

Programmability（可编程性）：可编程性指设备具有的、通过载入新的命令清单或新的互连方案，可以改变其动作的功能。

Programmed transfers（程控传输）：由CPU在程序的控制下直接处理数据，常常与数据IO相关。

PROM（Programmable Read-Only Memory，可编程只读存储器）：可以随机访问。

Protocol（协议）：为了在计算机间交换数据，经过协商达成一致的一套命令和数据格式。通信需要在规则和约定的控制下才能进行。

PSS（Packet Switched System，包交换系统）：由PTT提供的数据传输网络。

PSTN（Public Switched Telephone Network，公共交换电话网）：电路交换技术。

PTT（Public Telephone and Telegraph authority，

公共电话电报局)：这个词现已成为当地电话运营商的泛称，如BT、法国电信、Southern Bell等。

QAM（Quadrature phase and Amplitude Modulation，正交相位和振幅调制)：高性能的调制解调技术，它将相位和振幅调制方法结合起来，可以传送更多的数据。

Queue（队列)：FIFO数据结构。

Radix（基数)：计数系统中使用的不同数字的数量。十进制使用10个数字（0～9)，因此它的基数为10。

RAID（Redundant Array of Inexpensive Disks，廉价磁盘冗余阵列)：有效降低数据错误对存储文件影响的配置方案。

RAM（read-write Random Access Memory)：随机存取存储器。

RAMDAC：在图形卡中用来驱动监视器的数字到模拟的转换电路。

RAM disk（RAM磁盘)：一种将部分主存划出，做快速文件存储之用的技术。

Real number（实数)：可以有小数部分的数字。一般以浮点格式存储。

Real-time computing（实时计算)：必须在严格的时间限制下完成的处理过程。

Re-entrant code（可重入代码适)：可以同时为多个进程使用的代码。进程从不会更改这种代码。

Reference parameter（引用参数)：即地址参数。

Refresh cycle（刷新周期)：DRAM存储器需要定期执行的一种动作——重新生成保存在存储单元电容器上的数据。

Register（寄存器)：小段的存储器，出现在CPU内，或其他硬件设备中。

Register file（寄存器文件)：一块存储区域，一般用做堆栈，CPU对它们的访问可以和访问寄存器一样快速。

Register indirect addressing（寄存器间接寻址)：使用地址寄存器指向内存中存储单元的寻址模式。

Register window（寄存器窗口)：寄存器文件当前的活动区域。

Relative path name（相对路径名)：文件在磁盘上相对于当前位置的地址。

Repeater（中继器/重复器)：大型局域网系统的功率放大器。

Resource sharing（资源共享)：多任务系统中平等安全地使用公共设施的需求。

Response delay（响应延迟)：器件，如存储器，返回所请求的数据项需要耗费的时间。

Return address（返回地址)：子例程在执行结束后，用来跳回到调用代码的地址。

RIP(Routing Information Protocol，路由信息协议)：用来共享Internet路由信息的协议。

Ripple-through carry（行波传送进位)：并行加法器需要将进位信号从一个阶段传播到另一个阶段的情形。

RISC（Reduced Instruction Set Computer，精简指令集计算机)：如SPARC、ARM。

Roaming（漫游)：漫游指移动终端，比如蜂窝移动电话，可以在通话过程中从一个基站切换到另一个基站。

ROM（Read-Only random access Memory，只读随机存取存储器)

Root：1) 目录体系的最顶层。2) Unix系统的管理员。

Rotational latency（旋转延迟)：等待磁盘驱动器将所需的数据送到读取磁头下发生的延迟。

Router（路由器)：在数据包传输过程中，负责重新选路，将它们向目的地传送的网络设备。

Routing table（路由表)：路由器用来转发数据包的动态信息表。

RS232：电子工业联盟为慢速串行线路（如COM1或COM2）制订的标准。

SCAN：一种硬盘调度算法。

Scan code（扫描码)：当某个键被按下或释放时，由键盘生成的代码。

Scheduling（调度)：指计算机内决定接下来运行哪个进程的活动。实际运行下一个进程的代码称为分派程序。

SDRAM：Synchronous burst access DRAM，同步式突发访问DRAM。

Seek time（寻道时间)：磁盘驱动器将读取磁头移动到新的柱面需要花费的平均时间。

Segment（段)：数据在磁盘上存储的单位，一般为512个字节。

Semantic gap（语义鸿沟)：设计者的视角与机器指令提供的功能之间存在的断层。

Semantics（语义)：某些事物的含义。

Semaphore（信号量)：一种标示性的变量，它保护共享资源不被同时访问，因为同时访问有

可能会破坏数据。

Serial communication（串行通信）：串行通信中信息沿单个二进制位通道连续发送。

Server（服务器）：1）向其他系统提供服务的主机。

2）客户机-服务器结构中提供服务的一端。

sh：Unix最初的Bourne外壳。

Shell script（外壳脚本）：由外壳解释执行的命令文件，比如Perl。

Shortest first（处理时间最短者优先）：首先处理小型任务的一种常见调度策略。

Signal（信号）：在电话系统中，信号执行呼叫管理活动，比如建立通话、路由和记账。

SIMM（Single In-line Memory Module，单边接触内存模组）：在72针的连接卡上，最大为64 MB。

Simulation（模拟）：运行计算机模型以试验复杂系统各种未知特性的方法。

Single stepping（单步执行）：一种调试功能，它允许程序员每次一条指令地缓慢执行代码，观察每一步产生的结果。

Skew（畸变）：电路会不等时将脉冲延时的电子现象，这会导致平行总线上的数据被破坏。这是由于这些通道不同的传播特性所致。

SLIP（Serial Line Interface Protocol，串行线路接口协议）：一种允许计算机通过标准电话线和调制解调器直接访问Internet的协议。点到点协议（PPP）完成同样的工作，但相对更好一些。

SMS（Short Message Service，短消息服务）：GSM短消息服务，最长可以传输长度为160个字符的文本消息。

Socket：一种软件结构，更类似于文件，允许程序跨网络交换数据。

Software interrupt（软件中断）：通过执行指令强行产生的中断。

Source program（源程序）：可以用编译器进行转换的HLL程序。

Speculative loading（推测性载入）：根据对后续需求的预测预读指令和数据。

Spooling（假脱机）：Spooling一词来源于Simultaneous peripheral operation online（外部设备联机并行操作），这种技术将输出到慢速打印机的数据缓冲到快速的磁盘上，这就将CPU从耗时的打印过程中解放出来。

SRAM：静态RAM。

Stack（堆栈）：一种后入先出数据结构（LIFO）。系统堆栈有一个专门的CPU指针——SP，以加速数据项的压入和弹出。

Stack frame base pointer（堆栈框架基址指针）：保存当前堆栈框架（其中含有局部变量）地址的CPU地址寄存器。

Stack pointer（堆栈指针）：保存堆栈顶地址的CPU地址寄存器。

Statement（语句）：HLL中指令的单位。

Status register（状态寄存器）：含有条件标志位的寄存器，它存储前一项操作的状态或结果。

Store-and-forward（存储转发）：一种传输技术，网络中的每个节点都有缓冲区以保存输入的数据包。

Subroutine（子例程）：能够完成某项任务的指令集合，可以供调用者多次调用。

Swap space（交换空间）：磁盘上保留的一段区域，专门用来保存主存不能容纳的溢出内容。

Switch（交换机）：一种用来转发通信流量的交换设备。

Symbol table（符号表）：在编译器或汇编器内部使用的一种表，它保存变量名和过程名，以及它们在模块内的偏移，供链接器使用。

Synchronous bus（同步总线）：几个使用同一时钟信号的器件之间的互连通路。

Synchronous transmission（同步传输）：传输的一种形式，接收方和发送方共享同一公共时钟。这个时钟常常由发送方提供。

Syntax（语法）：语言的语法，定义符号的合法和不合法组合。

System bus（系统总线）：将处理器、存储器和输入-输出部分互相连接在一起的主总线。它由负责数据、地址和控制信息的部分构成。

System call（系统调用）：一条将控制权从执行进程转移到操作系统的指令，这样所请求的操作才能得以执行。

System clock（系统时钟）：用来计时和同步计算机系统内部所有活动的中心振荡器。

System program（系统程序）：用来组织和管理计算机系统运作的软件。

System throughput（系统吞吐量）：计算机完成工作的量度。

Tag（标签）：用在相关高速缓存存储器中的标识性记号。

TAPI（Telephony Application Programming

Interface)：电话应用系统编程接口。

Task（任务）：参见进程（Process）。

TCP/IP（Transmission Control Protocol/Internet Protocol，传输控制协议/因特网协议）：这是Internet使用的通信套件。它是一系列的规则和定义，允许跨网络进行信息交换。

TDM（Time Division Multiplexed，时分多路复用）：一种让参与者定期轮换共享信道的技术。

TDM/PCM（Time Division Multiplexed/Pluse Code Modulation，时分多路复用/脉冲编码调制）：主干线传输线路的通信标准。子通道经过数字编码后，分时间片共享总线。

Technological convergence（技术趋同）：专为表达电话、计算机和电视技术的融合而创造的新词。

Telex（电传打字机）：古老的基于文本的电路交换网络，它连接50 b/s的终端（由键盘和打印机组成）。ASR33（仅有少量的存储设备）可以提升到110 b/s。

Telnet：TCP/IP软件包中提供的一个可执行程序。它提供远程登录和基本的终端模拟服务。

Terminator（终接器）：接入在局域网端点的电阻，用来吸收信号功率。

Thrashing（系统颠簸）：当虚拟内存系统过载时发生的不利情况。

TLB（Translation Lookaside Buffer，转换后备缓冲区）：小型的快速缓冲区，其中保存内存管理单元的转换表。它可以加速虚拟地址到物理地址的转换。

Trap（陷阱），软件中断

Truth table（真值表）：帮助将输入条件映射到所需的输出的表格。

TTL（Transistor-Transistor Logic，晶体管-晶体管逻辑电路）：德州仪器公司第一款成功的集成电路技术。编号从74开始，如74F138。

2的补码（two's complement）：正负整数的一种表达方式。它允许减法可以和加法一样执行。

UART（Universal Asynchronous Receiver and Transmitter，通用异步收发器）：一种硬件设备（芯片），它可以作为RS232串行链路的接口。它将并行的数据转换成串行数据进行传输，在接收时再转换回来，同时还包括流量控制和错误控制。

Unix：一种多任务操作系统，最初由Ken Thompson开发，后来Bell实验室的Dennis Ritchie加入其中。

Unresolved external reference（未解决的外部引用）：由链接器报告的一种错误消息，表示对变量或函数的引用尚未完成。

URL（Uniform Resource Locator，统一资源定位符）：Internet上网站和服务的惟一定位方法，例如http://www.uwe.ac.uk/。

Usenet：全球范围的基于Unix的网络，它支持消息的分发。它是一种“存储转发”协议，现已大部分被其他Internet功能取代。

UUCP（Unix to Unix Copy）：早期的文件传输实用程序，目的是将不同的Unix系统连接在一起。

Value parameter（值参数）：参数的一种，子例程需要的是它的值（而非它的地址）。

VAN（Value-Added Network，增值网络）：提供附加服务的网络，目的是提高营收的潜能。

VHDL（超高速集成电路硬件设计描述语言）：一种专为硬件设计者设计的HLL语言，表面上类似于ADA。

Virtual address（虚地址）：理想化的地址，从0开始，不考虑实际系统的物理限制。

Virtual circuit（虚电路）：数据包传送前在两端建立的通信。TCP和ATM都提供这类服务。

Virtual memory（虚拟内存）：将主存寻址范围扩展到磁盘区的方案。

VLSI（Very Large-scale Integrated Circuit）：超大规模集成电路）。

Volatile memory（易失性存储器）：当断电时所存储的内容会丢失的存储器。

VT100：由DEC公司生产的ASCII终端，在1960年到1970年间，在Unix和其他非IBM系统上应用得十分普遍。

WAN（Wide Area Network）：广域网。

Waveform（波形）：重复的模拟图形，如电压随时间变化的图形。

Web browser（Web 浏览器）：可以显示Internet上网页中的文字和图像的软件，如Netscape Navigator、Mosaic和Microsoft Internet Explorer。

Wideband（宽带）：能够进行高速率数据传输的通道。它们采用调制和解调制技术。

WIMP：Windows、Icons、Menus和Pointers的缩写。使用交互式图形作为用户界面的方式，由Xerox和Apple发起。

Win32：标准化的Windows API，由数百个函数调

用组成。

Windows：Microsoft的GUI（图形用户界面）。

WinSock：Windows sockets API：它在一个DLL中，帮助用户建立本地TCP socket，通过网络与不同计算机上的进程通信。

WKP（Well-Known Port，知名端口）：公开的服务所使用的TCP/IP端口。

Workspace（工作空间）：分配给进程用来存储状态头、堆栈和变量的存储区。

World Wide Web（WWW，万维网）：庞大的互相引用的文档集合，保存在世界各地各种不同的计算机上，可以通过Internet下载，或使用Web浏览程序查看。

Worle：Weston-super-Mare的缩写。当地的学生用worle.c替代有名的world.c。

WWW，参见万维网（World Wide Web）。

WYSIWYG（What You See Is What You Get，所见即所得）：能够立即在屏幕上将输入的文本编排好显示出来的编辑器。

X terminal（X终端）：telnet的窗口化版本，可以远程登录系统。

X Window system（X窗口系统）：客户机-服务器软件架构，最初为Unix设计，它提供GUI。

Z80：由Zilog制造的8位或16位微处理器，它的出现早于8086。在推出后成为工作站最常采用的CPU。

Z flag：CPU条件代码，表示ALU执行的操作是否产生0值。

参考文献

Adobe Systems (1985). *PostScript Language: Tutorial and Cookbook*. Reading, MA: Addison-Wesley.

Aho A. V., Kernighan B. W. and Weinberger P. J. (1988). *The AWK Programming Language*. Reading, MA: Addison-Wesley.

Anderson D. (1997). *Universal Serial Bus System Architecture*. Reading, MA: Addison-Wesley.

Anderson D. and Shanley T. (1995). *Pentium Processor System Architecture*. Reading, MA: Addison-Wesley.

Barroso L, Dean J. and Holzle U. (2003). Web Search for a Planet. *IEEE Micro*, March-April.

Brin S. and Page L. (2000). The Anatomy of a Large Scale Hypertext Web Search Engine. Computer Science Dept., Stanford University.

Buchanan W. (1998). *PC Interfacing: Communications and Windows Programming*. Reading, MA: Addison-Wesley.

Comer D. E. (2003). *Computer Networks and Internets*, 3rd edn. Upper Saddle River, NJ: Prentice Hall.

Evans J. and Trimper G. (2003). *Itanium Architecture for Programmers*. Upper Saddle River, NJ: Prentice Hall.

Foster I. (2000). *Designing and Building Parallel Programs*. Reading, MA: Addison-Wesley.

Furber S. (2000). *ARM system-on-chip architecture*. Reading, MA: Addison-Wesley.

Glenn, W. and Northrup, T. (2005). *MCSA/MCSE Installing, Configuring & Administering Windows XP Professional*. Redmond, WA: Microsoft Press.

Hamacher V. C., Vranesic Z. G. and Zaky S. G. (2002). *Computer Organization*, 5th edn. New York: McGraw-Hill.

Harte L. (2005). *Introduction to GSM*. Fuquay Varina, NC: Althos Books.

Hayes J. (1998). *Computer Architecture and Organization*. New York: McGraw-Hill.

Heuring V. P. and Jordan H. F. (2004). *Computer Systems Design and Architecture*, 2nd edn. Menlo Park, CA: Addison-Wesley.

Hodson P. (1997). *Computing: Local Area Networks*. London: Letts Educational.

Hyde J. (2002). *USB Design by Example: A Practical Guide to Building I/O Devices*. Mt Prospect, IL: Intel Press.

Intel (1997). *Intel Architecture Software Developer's Manual*, Vol. 1: Basic Architecture. Mt Prospect, IL: Intel Corporation. Also available as a PDF file from: `http://www.intel.co.uk/design/mobile/manuals/243190.htm`.

Karp, D. *et al.* (2005). *Windows XP in a Nutshell*. Sebastopol, CA: O'Reilly.

Kidder T. (1981). *The Soul of a New Machine*. Boston, MA: Little, Brown.

Macario, R. C. V. (1997). *Cellular Radio*. Basingstoke: Macmillan.

Mansfield M. (1993). *The Joy of X: An Overview of the X Window System*. Wokingham: Addison-Wesley.

Messmer H.-P. (2001). *The Indispensable PC Hardware Book*, 4th edn. Wokingham: Addison-Wesley.

Nutt G. (1999). *Operating Systems: A Modern Perspective*, 2nd edn. Reading, MA: Addison-Wesley.

Pacheco P. (1997). *Parallel programming with MPI*. San Francisco: Morgan Kaufmann.

Patterson D. A. and Hennessy J. L. (2004). *Computer Organization and Design: The Hardware/Software Interface* 3rd edn. San Francisco: Morgan Kaufmann.

Pearce E. (1997). *Windows NT in a Nutshell.* Sebastopol, CA: O'Reilly.
Redl S. M., Weber M. K. and Oliphant M. W. (1995). *An Introduction to GSM.* Boston, MA: Artech House.
Reid R. H. (1997). *Architects of the Web: 1,000 Days that Built the Future of Business.* New York: John Wiley & Sons.
Ritchie C. (1997). *Computing: Operating Systems: Incorporating Unix and Windows.* London: Letts Educational.
Schiller J. (1999). *Mobile Communications.* Harlow: Addison-Wesley.
Seal D. (2000). *ARM Architecture Reference Manual.* Reading, MA: Addison-Wesley.
Silberschatz A., Galvin P. and Gagne G. (2000). *Applied Operating System Concepts.* New York: John Wiley.
Sima D., Fountain T. and Kaczuk P. (1997). *Advanced Computer Architecture.* Reading, MA: Addison-Wesley.
Simon R. J., Gouker M. and Barnes B. (1995). *Windows 95: Win32 Programming API Bible.* Indianapolis, IN: Waite Group.
Sloan J. (2004). *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and KPI.* Sebastopol, CA: O'Reilly.
Sloss A., Symes D. and Wright C. (2004). *ARM Systems Developer's Guide.* Amsterdam: Elsevier.
Sobell M. (2005). *A Practical Guide to Linux Commands, Editors, and Shell Programming.* Upper Saddle River, NJ: Prentice Hall.
Stevens W. R. (1994). *TCP/IP Illustrated,* Vol. 1: The Protocols. Reading, MA: Addison-Wesley.
Stevens W. R. (1998). *Unix Network Programming.* Upper Saddle River, NJ: Prentice Hall.
Stone H. (1993). *High Performance Computer Architecture.* Reading, MA: Addison-Wesley.
Tanenbaum A. S. (2000). *Structured Computer Organization,* 4th edn. Upper Saddle River, NJ: Prentice Hall.
Tanenbaum A. S. (2002). *Computer Networks,* 4th edn. Upper Saddle River, NJ: Prentice Hall.
Texas Instruments (1984). *The TTL Data Book.* Dallas, TX: Texas Instruments, Inc.
Thewlis P. J. and Foxon B. N. T. (1983). *From Logic to Computers.* Oxford: Blackwell Scientific.
Vrenios A. (2002). *Linux Cluster Architecture.* Caterham: SAM's Books.
Vrenios A. (2006). *Linux High Performance Clusters.* Caterham: SAM's Books.
Wall L. (1996). *Programming Perl.* Sebastopol, CA: O'Reilly.
Wang P. S. (1997). *Introduction to UNIX with X and the Internet.* Boston, MA: PWS.