

计 算 机 科 学 从 书

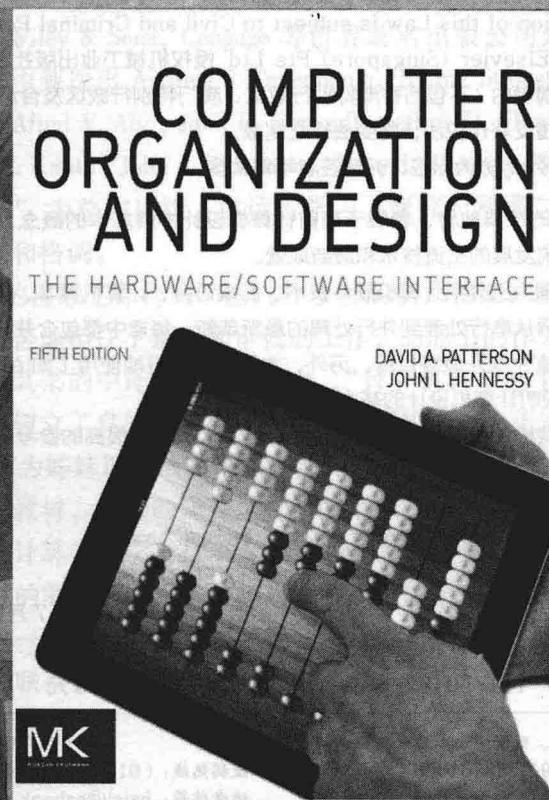
原书第5版

计算机组成与设计

硬件/软件接口

[美] 戴维 A. 帕特森 (David A. Patterson) 约翰 L. 亨尼斯 (John L. Hennessy) 著
加州大学伯克利分校 斯坦福大学
王党辉 康继昌 安建峰 等译
西北工业大学

Computer Organization and Design
The Hardware/Software Interface Fifth Edition



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

计算机组成与设计：硬件 / 软件接口（原书第 5 版）/（美）帕特森 (Patterson, D. A.), (美) 亨尼斯 (Hennessy, J. L.) 著；王党辉等译。—北京：机械工业出版社，2015.6
(计算机科学丛书)

书名原文：Computer Organization and Design : The Hardware/Software Interface,
Fifth Edition

ISBN 978-7-111-50482-5

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

中国版本图书馆 CIP 数据核字 (2015) 第 127169 号

本书版权登记号：图字：01-2014-2856

Computer Organization and Design: The Hardware/Software Interface, Fifth Edition

David A. Patterson and John L. Hennessy

ISBN:978-0-12-407726-3

Copyright © 2014 by Elsevier Inc. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

Copyright © 2015 by Elsevier (Singapore) Pte Ltd. All rights reserved.

Printed in China by China Machine Press under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR, Macau SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由 Elsevier (Singapore) Pte Ltd. 授权机械工业出版社在中国大陆境内独家出版和发行。本版仅限在中国境内（不包括香港特别行政区、澳门特别行政区及台湾地区）出版及标价销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

本书封底贴有 Elsevier 防伪标签，无标签者不得销售。

本书是计算机组成经典教材，着眼于当前计算机设计中最基本的概念，详细展示了软硬件间的关系，介绍当代计算机系统发展的主流技术和最新成就。

本书以 MIPS 处理器为例介绍计算机硬件技术、汇编语言、计算机算术、流水线以及存储器层次结构等基本技术。书中强调从串行处理到并行处理的最新革新，每章中都包含并行硬件和软件的主题，以软硬件协同设计发挥多核性能为最终目标。另外，本版与时俱进地使用了如 ARM Cortex A8 和 Intel Core i7 等现代设计来说明计算机设计的基本原理。

本书适合作为高等院校计算机专业教材，对广大技术人员也有很高的参考价值。

出版发行：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：姚 蕃

责任校对：殷 虹

印 刷：北京诚信伟业印刷有限公司

版 次：2015 年 7 月第 1 版第 1 次印刷

开 本：185mm×260mm 1/16

印 张：34.5

书 号：ISBN 978-7-111-50482-5

定 价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

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

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

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

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

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

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章教育

华章科技图书出版中心

本书赞誉 |

Computer Organization and Design: The Hardware/Software Interface

教材的选择是一个非常令人沮丧的折衷过程——教学方法、内容的覆盖面、行文造句的质量、描述的精确程度、成本等都需要考虑。这是一本不需要在这些方面进行折衷且涉及各个方面 的书。它不仅是一部计算机组成的教科书，也是所有计算机科学教科书的范本。

——Michael Goldweber, Xavier 大学

从第 1 版开始，我已经使用本书很多年了。第 5 版在现有的经典内容上做了显著的改进。从桌面计算到移动计算再到大数据，技术的发展开辟了新的研究领域，包括 ARM 之类的嵌入式处理器、软件和硬件如何进行交互以提高性能以及云计算等，这些在本版中都有所体现，且没有牺牲基础知识的内容。

——Ed Harcourt, St. Lawrence 大学

这是一本应该保存在书架上的计算机体系结构的教材。本书既陈旧又新颖，因为它不但介绍了基本原理——摩尔定律、抽象、加速大概率事件、冗余、存储器层级、并行和流水线，也使用如 ARM Cortex A8 和 Intel Core i7 等现代设计对这些基本原理进行了说明。

——Mark D. Hill, Wisconsin-Madison 大学

第 5 版与先进嵌入式和众核（GPU）系统的发展保持同步，它们的发展使平板电脑和智能手机很快变成新的桌面电脑。本书展示了这些变化，并提供了丰富的计算机组成与设计的基本原理，这些内容对于这类新型设备和系统的软硬件设计人员来说非常有用。

——Dave Kaeli, Northeastern 大学

当前，半导体加工工艺按比例缩小的困难在于系统功耗的限制，在移动系统和大数据处理器领域，该限制与不断增长的性能需求之间的矛盾越来越尖锐。第 5 版除了为读者提供计算机体系结构的介绍，还为读者准备了为应对该矛盾而必须在设计方面做出的修改。在这个计算技术的新领域，必须进行软硬件协同设计，另外，系统级体系结构优化与部件级优化一样重要。

——Christos Kozyrakis, Stanford 大学

Patterson 和 Hennessy 非常英明地指出持续发展的计算机硬件结构中的问题，强调硬件和软件模块在不同抽象层次上的交互。第 5 版通过一系列软硬件间的机制将 I/O 和并行概念贯穿全书，对后 PC 时代的计算机体系结构进行了全面的展示。对于在平板 PC 和云计算中面临能效问题和并行性挑战的软件和硬件专家而言，本书是一部必备的指导书。

——Jae C. Oh, Syracuse 大学

译者序

Computer Organization and Design: The Hardware/Software Interface

D. Patterson 和 J. Hennessy 是计算机领域的知名学者，为计算机学科和产业的发展做出了巨大贡献，他们合著的《Computer Organization and Design: The Hardware/Software Interface》一书现已发行了第 5 版。该书对计算机组成的研究和设计实践进行了全面系统的总结。目前，国际上许多大学的计算机原理或计算机体系结构课程都采用这本教材，国内也有不少大学采用这本教材。

第 5 版在保持计算机组成方面传统论题以及前 4 版特点的基础上，引入了许多近几年计算机领域发展的新论题，如移动计算、大数据等。另外，在实例方面也与时俱进地采用 ARM Cortex A8 和 Intel Core i7 等现代设计对计算机组成的基本原理进行说明。

感谢机械工业出版社华章公司一直关注本书的引进和中译本的出版工作，姚蕾编辑和朱秀英编辑为中译本的翻译工作提出了大量宝贵意见。

感谢清华大学郑纬民教授对前 3 版中译本所做的工作，是他使得这本重要教材在国内有了广泛的读者。感谢西北工业大学康继昌教授、樊晓桠教授和安建峰副教授对第 4 版中译本所做的工作，是他们使得第 5 版的翻译有了很好的基础，另外，康继昌教授和安建峰副教授也参与了本版的翻译工作。

西北工业大学计算机学院的研究生赵磊、刘朝锋、庄森、马鑫、朱迪、朱艳娜等也参加了本书的翻译和校对工作。本书的翻译工作还得到了国家自然科学基金项目（No. 61472322）的支持。

由于译者水平有限，书中难免存在一些翻译不当或理解欠妥的地方，希望读者批评指正。

王党辉
2015 年 2 月于西北工业大学

前 言 |

Computer Organization and Design: The Hardware/Software Interface

我们能体验的最美好的事物是神秘，它是所有真正的艺术和科学的源泉。

——阿尔伯特·爱因斯坦，《我的信仰》，1930

关于本书

在学习计算机科学与工程时，除了掌握计算的基本原理外，还应该了解该领域的最新进展。各种计算领域中的读者应有机会学习计算机系统的组成理论，因为这是决定计算机系统的功能、性能甚至成功与否的关键。

现代计算机技术需要各种计算方面的专家，他们应对硬件和软件都有深入的理解。硬件和软件在多个层次上的相互关系成为理解计算基本原理的框架。无论你的主要兴趣是硬件还是软件，是计算机科学还是电气工程，计算机组成与设计的基本思想都是相同的。因此，本书着重展示硬件与软件的关系，并重点介绍当今计算机中的基础概念。

近年来，处理器已经由单核发展为多核，这也印证了本书自第1版就预测的发展趋势。有些程序员忽略了这一发展趋势，他们希望计算机体系结构专家、编译器设计者和芯片工程师能够帮助他们，让程序不做任何修改就可以更快或更高效地在新型处理器上运行。但是，这样的时代已经一去不返了。为了使程序运行得更快，必须将其并行化。然而，许多研究者的目标是希望程序员在编写程序时不需要考虑硬件的并行特征，这一目标要很多年才能实现。至少在下一个十年里，大多数程序员必须理解硬件/软件接口，才能编写出能在并行计算机上高效运行的程序。

本书适合以下读者：在汇编语言或逻辑设计方面只有少许经验，需要理解基本计算机组成的读者；具有汇编语言或逻辑设计的基础，需要学习如何设计计算机，或者要进一步理解计算机系统如何工作的读者。

与本书相关的另一本书

有些读者可能已经熟悉作者的另一本书《Computer Architecture: A Quantitative Approach》^①。该书已广为流传，经常以作者姓名命名，称为“Hennessy and Patterson”（本书则常称为“Patterson and Hennessy”）。我们写该书的目的是要用坚实的工程基础和量化的性价比来描述计算机体系结构的原理。我们以商用系统为例，用测量的方法来描述实际的设计经验。我们的目标是用量化的方法而不是用描述的方法学习计算机体系结构，希望这一方法有助于培养能精确理解计算机的专业人才。

本书的大多数读者并不一定要成为计算机体系结构的设计者。但是，未来软件设计人员对与软件系统协同工作的基本硬件技术的理解程度，将直接影响软件系统的性能和能效。因此，编译器设计者、操作系统设计者、数据库程序员以及其他大多数软件工程师对本书的原理必须有充分的了解。同样，硬件设计者也必须清楚地理解他们的工作对软件的相应影响。

所以，本书的内容远多于“Hennessy and Patterson”，而且进行了大量修订，以适应不同专业的读者。我们对再版“Hennessy and Patterson”时删除的大量介绍性材料的效果感到满意，这使得新版与第1版内容的重叠大大降低，本书亦如此。

① 机械工业出版社已出版了本书的第3版、第4版和第5版影印书，书名为《计算机体系结构：量化研究方法》。——编辑注

第5版的变化

第5版有6个主要的目标：使用运行例子的方法论证理解硬件的重要性；对前面已经提到的重要技术采用黑体的方式进行强调；对例子进行了更新，以反映从PC时代到后PC时代的发展；将I/O吞吐率方面的材料贯穿在整本书中，而不是集中在一章中；对技术内容进行了更新，以反映自2009年第4版出版以来工业界的变化；将附录和可选章节（目录中带有图标）的内容放在互联网上（booksite.elsevier.com/9780124077263/index.php），而不是放在CD上，降低了本书的成本，也使该版本变成了一部电子书。

在详细介绍第5版的修订目标之前，首先看下表。该表给出了本书的主要内容，并为关注硬件和关注软件的两种读者分别进行了导读。其中，第1、4、5和6章对两种读者同等重要。第1章讨论了能耗的重要性和由其引发的处理器从单核向多核的转变，并介绍了计算机设计中的8个伟大思想。第2章对于硬件读者来说很可能是复习性材料，而对于软件读者来说则是重要的阅读材料，特别是希望深入学习编译器和面向对象编程语言的读者。第3章适合对定点运算或者浮点运算感兴趣的读者，有些读者可能不需要学习第3章。然而，我们将在本章介绍矩阵乘法运行的例子，展示如何采用子字并行方法将性能提高4倍，因此不要跳过3.6~3.8节。第4章介绍了流水线处理器。其中，4.1、4.5和4.10节给出了流水线概述，4.12节给出了进一步提高矩阵乘法运算性能的方法，这些小节对于软件设计者来说比较重要。对于硬件设计者，第4章是核心内容。另外，根据读者知识背景的不同，可以选择是否首先阅读附录C中的逻辑设计部分。最后一章是多核、多处理器和集群系统，这一章是全新的内容，因此所有读者都应该阅读。本版的重要组织结构是使许多思想的引入更加自然，例如GPU、仓储式计算机和集群系统中的关键——网络接口卡的软硬件接口。

章/附录	节	关注软件	关注硬件
第1章 计算机概要与技术	1.1~1.11		
	■ 1.12 (历史)		
第2章 指令：计算机的语言	2.1~2.14		
	■ 2.15 (编译器和Java)		
	2.16~2.20		
附录E RISC指令集体系结构	■ 2.21 (历史)		
	■ E.1~E.17		
第3章 计算机的算术运算	3.1~3.5		
	3.6~3.8 (子字并行)		
	3.9~3.10 (谬误)		
	■ 3.11 (历史)		
附录B 逻辑设计基础	B.1~B.13		
	4.1 (引言)		
	4.2 (逻辑设计惯例)		
	4.3~4.4 (简单实现)		
	4.5 (流水线概述)		
	4.6 (流水线数据通路)		
	4.7~4.9 (冒险和异常)		
第4章 处理器	4.10~4.12 (并行和实例)		
	■ 4.13 (Verilog 流水线控制)		
	4.14~4.15 (谬误)		
	■ 4.16 (历史)		

(续)

章/附录	节	关注软件	关注硬件
附录 D 控制通路的硬件实现	D. 1 ~ D. 6		
	5. 1 ~ 5. 10		
	5. 11 (廉价冗余磁盘阵列)		
	5. 12 (Verilog cache 控制器)		
	5. 13 ~ 5. 16		
	5. 17 (历史)		
第 5 章 大容量和高速度：开发存储器层次结构	6. 1 ~ 6. 8		
	6. 9 (网络)		
	6. 10 ~ 6. 14		
	6. 15 (历史)		
附录 A 汇编器、链接器和 SPIM 仿真器	A. 1 ~ A. 11		
附录 C 图形处理单元	C. 1 ~ C. 13		

仔细阅读

有时间阅读

作为参考

回顾或阅读

拓展阅读

第 5 版的第一个目标是使用一个例子来论证理解硬件对提高性能和能效的重要性。正如前面所述，在第 3 章，我们采用子字并行将矩阵乘法加速 4 倍，在第 4 章通过循环展开将性能翻倍，证明了指令集并行的价值。第 5 章通过分块技术对 cache 进行优化，再次将性能翻倍。第 6 章通过在 16 个处理器上的线程级并行获得了 14 倍的加速比。这 4 种优化技术仅仅在原始的矩阵乘法例子中的 C 代码上增加了 24 行。

第二个目标是通过提早介绍计算机体系结构设计中的 8 个伟大思想并在整本书中明确指出它们的应用之处来帮助读者理解计算机设计的精髓。我们采用黑体文字的方式向读者提醒这 8 个思想的应用情况，在书中大约有 100 次引用。每一章中至少有 7 处应用这些思想的例子，并且每个思想至少被引用 5 次。通过并行提高性能、流水线和预测技术是引用次数最多的 3 个思想，紧接着是摩尔定律。第 4 章讲述处理器，是例子最多的一章，也是最吸引计算机设计者的一章。每一章都能找到的伟大思想是通过并行提高性能，这是近年来计算机领域中的一个重要发展方向。

第三个目标是通过例子和材料来识别计算技术从 PC 时代进入后 PC 时代的变化。因此，第 1 章直接介绍了平板电脑而没有介绍 PC，第 6 章介绍了云计算的基础设施。另外，在指令集方面，我们介绍了后 PC 时代中个人移动设备里使用的 ARM 指令集，以及在 PC 时代和云计算中占主导地位的 x86 指令集。

第四个目标是将 I/O 吞吐率方面的材料贯穿在整本书中，而不是集中在一章中，这与第 4 版中将并行性贯穿全书一样，因此，本版可在 1. 4、4. 9、5. 2、5. 5、5. 11 和 6. 9 节中找到 I/O 相关的材料。我们的想法是如果不把这些内容集中在一章，则读者（和教师）能更好地学习与掌握 I/O。

计算机是一个快速发展的领域，对于本书新的版本也是如此，编写新版的一个重要目的是更新技术内容。实际的例子就是反映后 PC 时代特点的 ARM Cortex A8 和 Intel Core i7，其他的亮点包括新的 ARMv8 64 位指令集、讲解 CPU 特有术语的教程、组成云的仓储式计算机的内涵以及对 10G 以太网卡的深入理解。

为了保持纸质书的厚度及其与电子书的兼容性，我们一改以前版本的做法，将可选内容由随书 CD 改为网络在线的形式。

最后，我们更新了本书的所有练习题。

在对内容进行修订的同时，我们保留了以往版本中有用的元素。为使本书更好地作为参考书，我们还在新术语第一次出现时给出了定义。书中标题为“理解程序性能”部分的内容有助于读者理解程序的性能，以及如何提高性能，就像书中“硬件/软件接口”部分会帮助读者理解有关接口的权衡问题一样。“重点”部分仍然存在，以使读者看到整个“森林”而不是每一棵“树”。每章最后提供“小测验”部分的答案，帮助读者在第一时间加强对内容的理解。本版同样提供了 MIPS 参考数据（这是从 IBM System/360 “绿卡”得到的灵感），并对数据进行了更新，在编写 MIPS 汇编语言程序时，这应该是一个很好的参考。

教学支持[⊖]

我们收集了大量材料供教师授课使用，包括题解、图表、幻灯片等，可从出版商处获得。如需更多信息，请访问网址：textbooks.elsevier.com/9780124077263。

结语

从下面的致谢中，你可以发现我们花费了大量精力去修改本书的错误。由于本书印刷了多次，因此我们有机会做更多的校正。如果你发现有遗留的错误，请通过电子邮件与出版社联系：cod5bugs@mfp.com。

本版标志着 Hennessy 和 Patterson 自 1989 年以来长期合作的第二次中止。由于要管理一所世界知名的大学，Hennessy 校长将不能继续承担新版本的实际编写工作。留下 Patterson 一人感觉自己像个总是和伙伴们一起演出的演员，突然被推到前台独自表演。所以，在致谢名单中列出的人和 Berkeley 的同行们在本书的撰写过程中甚至起了更大的作用。

第 5 版致谢

在本书的每一版中，我们都非常幸运地得到了来自许多读者、评审者和其他人员的帮助。每个人的帮助都使本书更加完美。

由于第 6 章做了巨大的修改，因此我们对其思想和内容进行了单独的评审，并基于每位评审人的反馈意见做了修改。感谢 Stanford 大学的 Christos Kozyrakis，他建议在集群中使用网络接口来论证 I/O 的软硬件接口，并对该章的组织提出了意见。还要感谢 Stanford 大学的 Mario Flagsilk，他提供了 NetFPGA NIC 的细节、表格以及性能评估。另外，以下人员对本章提出了修改建议：Northeastern 大学的 David Kaeli、HP 实验室的 Partha Ranganathan、Wisconsin 大学的 David Wood 以及 Berkeley 大学的同事 Siamak Faridani、Shoaib Kamil、Yunsup Lee、Zhangxi Tan 和 Andrew Waterman。

我们要对 UC Berkeley 的 Rimas Avizienis 表示特别的感谢，他开发了不同版本的矩阵乘法程序，并提供了相应的性能数据。当我在 UCLA 读研究生时，我与他的父亲一起工作，能够与他一起在 UCB 共事是一件美好的事情。

我也要对我的长期合作伙伴——UC Berkeley 的 Randy Katz 表示感谢。我们共同讲授研究生的计算机体系结构课程，他在开发计算机体系结构的伟大思想方面提供了很大的帮助。

感谢 David Kirk、John Nickolls 和他们在 NVIDIA 的同事们（Michael Garland、John Mon-

[⊖] 爱思唯尔 (ELS)：关于本书教辅资源，使用教材的老师需通过爱思唯尔的教材网站 (www.textbooks.elsevier.com) 注册并通过审批后才能获取相关资源。具体方法如下：在 www.textbooks.elsevier.com 教材网站查找到该书后，点击“instructor manual”便可申请查看该教师手册。有任何问题，请致电 010-85208853。
——编辑注

trym、Doug Voorhies、Lars Nyland、Erik Lindholm、Paulius Micikevicius、Massimiliano Fatica、Stuart Oberman、Vasily Volkov) 提供了第一个深入介绍 GPU 的附录。再次感谢 Jim Larus, 他现在是 EPFL 计算机与通信科学学院的院长, 为本书发挥了在汇编语言方面的专长, 欢迎本书读者使用他所开发和维护的模拟器。

非常感谢 South Carolina 大学的 Jason Bakos, 他在第 4 版的基础上对本版的练习题进行了更新。第 4 版的练习题由以下人员编写: Perry Alexander (Kansas 大学)、Javier Bruguera (de Santiago de Compostela 大学)、Matthew Farrells (California 大学 Davis 分校)、David Kaeli (Northeastern 大学)、Nicole Kaiyan (Adelaide 大学)、John Oliver (Cal Poly, San Luis Obispo)、Milos Prvulovic (Georgia 理工大学) 和 HP 的 Jichuan Chang、Jacob Leverich、Kevin Lim、Partha Ranganathan。

感谢 Jason Bakos 开发了新的幻灯片。

感谢许多教师的贡献, 他们回答出版商的调查问卷、评审我们的提议、出席小组会议, 并对本版的计划进行分析和反馈。他们是:

2012 焦点小组: Bruce Barton (Suffolk County Community 学院), Jeff Braun (Montana 理工大学), Ed Gehringer (North Carolina State), Michael Goldweber (Xavier 大学), Ed Harcourt (St. Lawrence 大学), Mark Hill (Wisconsin 大学 Madison 分校), Patrick Homer (Arizona 大学), Norm Jouppi (HP 实验室), Dave Kaeli (Northeastern 大学), Christos Kozyrakis (Stanford 大学), Zachary Kurmas (Grand Valley 州立大学), Jae C. Oh (Syracuse 大学), Lu Peng (Louisiana 州立大学), Milos Prvulovic (Georgia 理工), Partha Ranganathan (HP 实验室), David Wood (Wisconsin 大学), Craig Zilles (Illinois 大学香槟分校)。

参考调查和审阅的学者: Mahmoud Abou-Nasr (Wayne 州立大学), Perry Alexander (Kansas 大学), Hakan Aydin (George Mason 大学), Hussein Badr (New York 州立大学 Stony Brook 分校), Mac Baker (Virginia Military 学院), Ron Barnes (George Mason 大学), Douglas Blough (Georgia 理工), Kevin Bolding (Seattle Pacific 大学), Miodrag Bolic (Ottawa 大学), John Bonomo (Westminster 学院), Jeff Braun (Montana 理工大学), Tom Briggs (Shippensburg 大学), Scott Burgess (Humboldt 州立大学), Fazli Can (Bilkent 大学), Warren R. Carithers (Rochester 理工学院), Bruce Carlton (Mesa Community 学院), Nicholas Carter (Illinois 大学香槟分校), Anthony Cocchi (New York 城市大学), Don Cooley (Utah 州立大学), Robert D. Cupper (Allegheny 学院), Edward W. Davis (North Carolina 州立大学), Nathaniel J. Davis (Air Force 理工学院), Molisa Derk (Oklahoma 城市大学), Derek Eager (Saskatchewan 大学), Ernest Ferguson (Northwest Missouri 州立大学), Rhonda Kay Gaede (Alabama 大学), Etienne M. Gagnon (UQAM), Costa Gerousis (Christopher Newport 大学), Paul Gillard (Newfoundland 纪念大学), Michael Goldweber (Xavier 大学), Georgia Grant (San Mateo 学院), Merrill Hall (The Master's 学院), Tyson Hall (Southern Adventist 大学), Ed Harcourt (St. Lawrence 大学), Justin E. Harlow (South Florida 大学), Paul F. Hemler (Hampden-Sydney 学院), Martin Herbordt (Boston 大学), Steve J. Hodges (Cabrillo 学院), Kenneth Hopkinson (Cornell 大学), Dalton Hunkins (St. Bonaventure 大学), Baback Izadi (New York 州立大学 New Paltz 分校), Reza Jafari, Robert W. Johnson (Colorado Technical 大学), Bharat Joshi (North Carolina 大学 Charlotte 分校), Nagarajan Kandasamy (Drexel 大学), Rajiv Kapadia, Ryan Kastner (California 大学 Santa Barbara 分校), E. J. Kim (Texas A&M 大学), Jihong Kim (Seoul 国立大学), Jim Kirk (Union 大学), Geoffrey S. Knauth (Lycoming 学院), Manish M. Kochhal (Wayne 州立大学), Suzan Koknar-Tezel (Saint Joseph 大学), Angkul Kongmunvattana (Columbus 州立大学), April Kontostathis (Ursinus

学院), Christos Kozyrakis (Stanford 大学), Danny Krizanc (Wesleyan 大学), Ashok Kumar, S. Kumar (Texas 大学), Zachary Kurmas (Grand Valley 州立大学), Robert N. Lea (Houston 大学), Baoxin Li (Arizona 州立大学), Li Liao (Delaware 大学), Gary Livingston (Massachusetts 大学), Michael Lyle, Douglas W. Lynn (Oregon 理工学院), Yashwant K Malaiya (Colorado 州立大学), Bill Mark (Texas 大学 Austin 分校), Ananda Mondal (Claflin 大学), Alvin Moser (Seattle 大学), Walid Najjar (California 大学 Riverside 分校), Danial J. Neebel (Loras 学院), John Nestor (Lafayette 学院), Jae C. Oh (Syracuse 大学), Joe Oldham (Centre 学院), Timour Palatashev, James Parkerson (Arkansas 大学), Shaunak Pawagi (SUNY Stony Brook 分校), Steve Pearce, Ted Pedersen (Minnesota 大学), Lu Peng (Louisiana 州立大学), Gregory D Peterson (Tennessee 大学), Milos Prvulovic (Georgia 理工), Partha Ranganathan (HP 实验室), Dejan Raskovic (Alaska 大学 Fairbanks 分校), Brad Richards (Puget Sound 大学), Roman Rozanov, Louis Rubinfield (Villanova 大学), Md Abdus Salam (Southern 大学), Augustine Samba (Kent 州立大学), Robert Schaefer (Daniel Webster 学院), Carolyn J. C. Schauble (Colorado 州立大学), Keith Schubert (CSU San Bernardino 分校), William L. Schultz, Kelly Shaw (Richmond 大学), Shahram Shirani (McMaster 大学), Scott Sigman (Drury 大学), Bruce Smith, David Smith, Jeff W. Smith (Georgia 大学, Athens), Mark Smotherman (Clemson 大学), Philip Snyder (Johns Hopkins 大学), Alex Sprintson (Texas A&M), Timothy D. Stanley (Brigham Young 大学), Dean Stevens (Morningside 学院), Nozar Tabrizi (Kettering 大学), Yuval Tamir (UCLA), Alexander Taubin (Boston 大学), Will Thacker (Winthrop 大学), Mithuna Thottethodi (Purdue 大学), Manghui Tu (Southern Utah 大学), Dean Tullsen (UC San Diego 分校), Rama Viswanathan (Beloit 学院), Ken Vollmar (Missouri 州立大学), Guoping Wang (Indiana-Purdue 大学), Patricia Wenner (Bucknell 大学), Kent Wilken (California 大学 Davis 分校), David Wolfe (Gustavus Adolphus 学院), David Wood (Wisconsin 大学 Madison 分校), Ki Hwan Yum (Texas 大学 San Antonio 分校), Mohamed Zahran (New York 城市学院), Gerald D. Zarnett (Ryerson 大学), Nian Zhang (South Dakota School of Mines & Technology), Jiling Zhong (Troy 大学), Huiyang Zhou (Central Florida 大学), Weiyu Zhu (Illinois Wesleyan 大学)。

特别感谢 Mark Smotherman 一遍又一遍地查找本书中的技术错误和写作错误, 他的工作显著地改进了这一版的质量。

还要感谢 Morgan Kaufmann 公司同意在 Todd Green 和 Nate McFadden 的领导下对本书进行再版, 没有他们的工作, 我不可能完成本书。我们还要感谢 Lisa Jones 和 Russell Purdy, 她们分别负责管理出版过程和封面设计。新封面将本版中后 PC 时代的内容和第 1 版的封面呼应了起来。

以上提到的近 150 名人士为本版提供了大量帮助, 使之成为我们希望的最好的书。

David A. Patterson

作者简介 |

Computer Organization and Design: The Hardware/Software Interface

David A. Patterson

自从 1977 年在加州大学伯克利分校任职开始一直讲授计算机体系结构课程，他现在是计算机科学系的执行主席。他的教学工作获得了加州大学优秀教学奖、ACM Karlstrom 奖、IEEE 的 Mulligan 教育奖章和本科生教学奖。因为对 RISC 的贡献，Patterson 获得了 IEEE 技术进步奖和 ACM Eckert-Mauchly 奖，另外因为对 RAID 的贡献，他与别人分享了 IEEE Johnson 信息存储奖。他和 John Hennessy 分享了 IEEE John von Neumann 奖章与 C & C 奖励。Patterson 是美国艺术与科学院、计算机历史博物馆、ACM 和 IEEE 院士，并且入选国家工程院、国家科学院和硅谷工程名人堂。他曾任美国总统信息咨询委员会成员、伯克利 EECS 系 CS 分会主席、计算研究学会主席和 ACM 主席。这个纪录使他获得了 ACM 和 CRA 的杰出服务奖。

在伯克利，Patterson 领导了 RISC I 的设计与实现，这是第一款精简指令系统计算机，并且是商用 SPARC 体系结构的基础。他是廉价磁盘冗余阵列（RAID）项目的负责人，RAID 技术引导许多公司开发出了高可靠存储系统。他也参加了工作站网络（NOW）项目，该项目先引导互联网公司使用集群技术，再引导了后来的云计算。这些项目获得了三个 ACM 最佳论文奖。他当前的研究包括算法 - 机器 - 人、面向可证明优化实现的高可靠高效算法与专家系统。AMP 实验室正在开发可扩展的机器学习算法、仓储式计算机编程模型以及密集资源工具，以从云中的大数据获得有价值的信息。ASPIRE 实验室使用深度的硬件和软件协同技术在移动和货架计算系统中获得最高性能和能效。

John L. Hennessy

斯坦福大学的第十任校长，他从 1977 年开始任职于斯坦福大学电子工程与计算机科学系。Hennessy 是 ACM 和 IEEE 会士、国家工程院成员、国家科学院成员、美国艺术与科学院院士。他获得了许多奖项，其中包括：因为对 RISC 的贡献获得的 2001 年 ACM Eckert-Mauchly 奖，2001 年 Seymour Cray 计算机工程奖，与 Patterson 分享了 2000 年 John von Neumann 奖。他还获得了七个荣誉博士学位。

1981 年，他在斯坦福大学与几个研究生开始了 MIPS 项目。在 1984 年完成了该项目后，他离开大学，与他人共同创建了 MIPS 计算机系统（现在的 MIPS 技术公司），该公司开发了第一款商用 RISC 微处理器。2006 年，MIPS 微处理器销售了 20 亿片，应用范围从视频游戏和掌上计算机到激光打印机和网络交换机。后来，Hennessy 领导了 DASH（共享存储器的主导体体系结构）项目，该项目建立了第一个可扩展 cache-一致性多处理器的原形系统，其许多关键思想已经应用在先进的多处理器中。除了技术活动与大学工作外，他还是多家创业公司的早期顾问和投资者。

目 录

Computer Organization and Design: The Hardware/Software Interface

出版者的话
本书赞誉
译者序
前言
作者简介

第1章 计算机概要与技术 1

1.1 引言 1
1.1.1 计算应用的分类及其特性 2
1.1.2 欢迎来到后 PC 时代 3
1.1.3 你能从本书学到什么 4
1.2 计算机系统结构中的 8 个伟大思想 6
1.2.1 面向摩尔定律的设计 6
1.2.2 使用抽象简化设计 6
1.2.3 加速大概率事件 6
1.2.4 通过并行提高性能 7
1.2.5 通过流水线提高性能 7
1.2.6 通过预测提高性能 7
1.2.7 存储器层次 7
1.2.8 通过冗余提高可靠性 7
1.3 程序概念入门 7
1.4 硬件概念入门 10
1.4.1 显示器 11
1.4.2 触摸屏 12
1.4.3 打开机箱 12
1.4.4 数据安全 15
1.4.5 与其他计算机通信 16
1.5 处理器和存储器制造技术 17
1.6 性能 20
1.6.1 性能的定义 20
1.6.2 性能的度量 22
1.6.3 CPU 性能及其因素 23
1.6.4 指令的性能 24
1.6.5 经典的 CPU 性能公式 25
1.7 功耗墙 27

1.8 沧海巨变：从单处理器向多处理器转变 29
1.9 实例：Intel Core i7 基准 31
1.9.1 SPEC CPU 基准测试程序 31
1.9.2 SPEC 功耗基准测试程序 32
1.10 谬误与陷阱 33
1.11 本章小结 35
1.12 历史观点和拓展阅读 36
1.13 练习题 36

第2章 指令：计算机的语言 40

2.1 引言 40
2.2 计算机硬件的操作 43
2.3 计算机硬件的操作数 44
2.3.1 存储器操作数 45
2.3.2 常数或立即数操作数 47
2.4 有符号数和无符号数 48
2.5 计算机中指令的表示 53
2.6 逻辑操作 58
2.7 决策指令 60
2.7.1 循环 61
2.7.2 case/switch 语句 63
2.8 计算机硬件对过程的支持 64
2.8.1 使用更多的寄存器 66
2.8.2 嵌套过程 67
2.8.3 在栈中为新数据分配空间 69
2.8.4 在堆中为新数据分配空间 70
2.9 人机交互 72
2.10 MIPS 中 32 位立即数和寻址 75
2.10.1 32 位立即数 75
2.10.2 分支和跳转中的寻址 76
2.10.3 MIPS 寻址模式总结 78
2.10.4 机器语言解码 79
2.11 并行与指令：同步 81
2.12 翻译并执行程序 83
2.12.1 编译器 83
2.12.2 汇编器 84

2.12.3 链接器	85	3.4.1 除法算法及其硬件结构	125
2.12.4 加载器	87	3.4.2 有符号除法	128
2.12.5 动态链接库	87	3.4.3 更快速的除法	128
2.12.6 启动一个 Java 程序	89	3.4.4 MIPS 中的除法	129
2.13 以一个 C 排序程序作为完整的例子	90	3.4.5 小结	129
2.13.1 swap 过程	90	3.5 浮点运算	130
2.13.2 sort 过程	91	3.5.1 浮点表示	131
2.14 数组与指针	96	3.5.2 浮点加法	135
2.14.1 用数组实现 clear	96	3.5.3 浮点乘法	138
2.14.2 用指针实现 clear	97	3.5.4 MIPS 中的浮点指令	139
2.14.3 比较两个版本的 clear	97	3.5.5 算术精确性	145
2.15 高级内容：编译 C 语言和解释 Java 语言	98	3.5.6 小结	146
2.16 实例：ARMv7 (32 位) 指令集	98	3.6 并行性和计算机算术：子字并行	148
2.16.1 寻址模式	99	3.7 实例：x86 中流处理 SIMD 扩展和高级向量扩展	149
2.16.2 比较和条件分支	100	3.8 加速：子字并行和矩阵乘法	150
2.16.3 ARM 的特色	100	3.9 谬误与陷阱	153
2.17 实例：x86 指令集	102	3.10 本章小结	155
2.17.1 Intel x86 的改进	102	3.11 历史观点和拓展阅读	158
2.17.2 x86 寄存器和数据寻址模式	103	3.12 练习题	159
2.17.3 x86 整数操作	105		
2.17.4 x86 指令编码	107		
2.17.5 x86 总结	108		
2.18 实例：ARMv8 (64 位) 指令集	108	第 4 章 处理器	162
2.19 谬误与陷阱	109	4.1 引言	162
2.20 本章小结	110	4.2 逻辑设计的一般方法	165
2.21 历史观点和拓展阅读	111	4.3 建立数据通路	167
2.22 练习题	112	4.4 一个简单的实现机制	173
第 3 章 计算机的算术运算	117	4.4.1 ALU 控制	173
3.1 引言	117	4.4.2 主控制单元的设计	175
3.2 加法和减法	117	4.4.3 为什么不使用单周期实现方式	181
3.3 乘法	121	4.5 流水线概述	182
3.3.1 顺序的乘法算法和硬件	121	4.5.1 面向流水线的指令集设计	186
3.3.2 有符号乘法	124	4.5.2 流水线冒险	186
3.3.3 更快速的乘法	124	4.5.3 对流水线概述的小结	191
3.3.4 MIPS 中的乘法	124	4.6 流水线数据通路及其控制	192
3.3.5 小结	125	4.6.1 图形化表示的流水线	200
3.4 除法	125	4.6.2 流水线控制	203

4.8.2 缩短分支的延迟	215	5.4 cache 性能的评估和改进	270
4.8.3 动态分支预测	216	5.4.1 通过更灵活地放置块来减少 cache 缺失	272
4.8.4 流水线小结	220	5.4.2 在 cache 中查找一个块	275
4.9 异常	221	5.4.3 替换块的选择	276
4.9.1 MIPS 体系结构中的异常 处理	221	5.4.4 使用多级 cache 结构减少 缺失代价	277
4.9.2 在流水线实现中的异常	222	5.4.5 通过分块进行软件优化	280
4.10 指令级并行	226	5.4.6 小结	283
4.10.1 推测的概念	227	5.5 可信存储器层次	283
4.10.2 静态多发射处理器	227	5.5.1 失效的定义	283
4.10.3 动态多发射处理器	231	5.5.2 纠正一位错、检测两位错的 汉明编码 (SEC/DED)	284
4.10.4 能耗效率与高级流水线	233	5.6 虚拟机	287
4.11 实例：ARM Cortex-A8 和 Intel Core i7 流水线	234	5.6.1 虚拟机监视器的必备 条件	289
4.11.1 ARM Cortex-A8	235	5.6.2 指令集系统结构 (缺乏) 对虚拟机的支持	289
4.11.2 Intel Core i7 920	236	5.6.3 保护和指令集系统结构	289
4.11.3 Intel Core i7 920 的性能	238	5.7 虚拟存储器	290
4.12 运行更快：指令级并行和矩阵 乘法	240	5.7.1 页的存放和查找	293
4.13 高级主题：通过硬件设计语言 描述和建模流水线来介绍数字 设计以及更多流水线示例	242	5.7.2 缺页故障	294
4.14 谬误与陷阱	242	5.7.3 关于写	297
4.15 本章小结	243	5.7.4 加快地址转换：TLB	297
4.16 历史观点和拓展阅读	243	5.7.5 集成虚拟存储器、TLB 和 cache	300
4.17 练习题	243	5.7.6 虚拟存储器中的保护	302
第5章 大容量和高速度：开发 存储器层次结构	252	5.7.7 处理 TLB 缺失和缺页	303
5.1 引言	252	5.7.8 小结	307
5.2 存储器技术	255	5.8 存储器层次结构的一般框架	309
5.2.1 SRAM 技术	256	5.8.1 问题 1：一个块可以被放在 何处	309
5.2.2 DRAM 技术	256	5.8.2 问题 2：如何找到一个块	310
5.2.3 闪存	258	5.8.3 问题 3：当 cache 缺失时 替换哪一块	311
5.2.4 磁盘存储器	258	5.8.4 问题 4：写操作如何处理	311
5.3 cache 的基本原理	259	5.8.5 3C：一种理解存储器层 结构行为的直观模型	312
5.3.1 cache 访问	261	5.9 使用有限状态机来控制简单的 cache	314
5.3.2 cache 缺失处理	265	5.9.1 一个简单的 cache	314
5.3.3 写操作处理	266	5.9.2 有限状态机	315
5.3.4 一个 cache 的例子：内置 FastMATH 处理器	267		
5.3.5 小结	269		

5.9.3 一个简单的 cache 控制器的有限状态机	316
5.10 并行与存储器层次结构: cache 一致性	317
5.10.1 实现一致性的基本方案	318
5.10.2 监听协议	319
5.11 并行与存储器层次结构: 冗余廉价磁盘阵列	320
5.12 高级内容: 实现 cache 控制器	320
5.13 实例: ARM Cortex-A8 和 Intel Core i7 的存储器层次结构	320
5.14 运行更快: cache 分块和矩阵乘法	324
5.15 谬误和陷阱	326
5.16 本章小结	329
5.17 历史观点和拓展阅读	329
5.18 练习题	329
第 6 章 从客户端到云的并行处理器	340
6.1 引言	340
6.2 创建并行处理程序的难点	342
6.3 SISD、MIMD、SIMD、SPMD 和向量机	345
6.3.1 在 x86 中的 SIMD: 多媒体扩展	346
6.3.2 向量机	346
6.3.3 向量与标量的对比	347
6.3.4 向量与多媒体扩展的对比	348
6.4 硬件多线程	350
6.5 多核和其他共享内存多处理器	352
6.6 图形处理单元简介	355
6.6.1 NVIDIA GPU 体系结构简介	356
6.6.2 NVIDIA GPU 存储结构	357
6.6.3 GPU 展望	358
6.7 集群、仓储级计算机和其他消息传递多处理器	360
6.8 多处理器网络拓扑简介	363
5.9 与外界通信: 集群网络	366
6.10 多处理器测试集程序和性能模型	366
6.10.1 性能模型	368
6.10.2 Roofline 模型	369
6.10.3 两代 Opteron 的比较	370
6.11 实例: 评测 Intel Core i7 960 和 NVIDIA Tesla GPU 的 Roofline 模型	373
6.12 运行更快: 多处理器和矩阵乘法	376
6.13 谬误与陷阱	378
6.14 本章小结	379
6.15 历史观点和拓展阅读	381
6.16 练习题	382
附录 A 汇编器、链接器和 SPIM 仿真器	389
附录 B 逻辑设计基础	437
索引	494

计算机概要与技术

在不关注具体过程中完成更多的操作，这种方法促进了文明的进步。

——Alfred North Whitehead, 《An Introduction to Mathematics》, 1911

1.1 引言

欢迎阅读本书！非常高兴有机会与大家一起分享令人兴奋的计算机系统世界。这是一个进步飞快、新思想层出不穷、非常有趣的领域。事实上，计算机是极度充满生气的信息技术工业的产物，其相关产品几乎占全美国民生产总值的 10%。在摩尔定律的推动下，美国的经济已经与信息技术密不可分。这个不寻常的工业具有惊人的发展速度。在过去 30 年里，出现了许多导致计算产业革命的新型计算机，但是它们很快就被更好的计算机所取代。

电子计算机自 20 世纪 40 年代后期诞生以来，其创新性的竞争带来了史无前例的进步。如果运输业能够以计算机工业的速度发展，那么我们只需要花一美分就可以在一秒钟之内从纽约赶到伦敦。想象一下，这样的进步将如何改变社会——生活在南太平洋的塔希提岛，而工作在旧金山，傍晚去莫斯科吃夜宵——你能够想象得出这种进步意味着什么。

沿着农业革命、工业革命的发展方向，计算机促进了人类的第三次革命——信息革命。信息革命使人类的能力成倍增长，自然而深刻地影响着人类的日常生活，甚至改变了寻求新知识的方法。现在有一种科学探索的新方式，即计算科学家联合理论和实验科学家，共同探索天文学、生物学、化学和物理学的前沿问题。

计算机革命一直在向前推进。每当计算成本降低 10 倍，计算机的发展机遇就会增加 10 倍。原本出于经济考虑不可行的应用突然变得可行了。例如，下述各项应用在过去曾经是“计算机科学幻想”：

- **车载计算机：**在 20 世纪 80 年代初微处理器的性能和价格得到极大改进之前，用计算机来控制汽车几乎是天方夜谭。而今天，用计算机控制的汽车发动机普遍应用，车载计算机不仅改进了燃油效率、减轻了污染，还通过盲点警告、车道偏离警告、移动目标检测和安全气囊实现了碰撞时对乘客的保护。
- **手机：**谁曾想到计算机系统的发展会使全球一半以上的人口拥有手机，并让人们几乎在全球的各个角落都可以自由通信？
- **人类基因项目：**以前用于匹配和分析人类基因序列的计算机设备价格达几亿美元。在过去的 15~25 年里，用于该项目的计算机设备的价格降低了 10~100 倍。随着计算机设备价格的持续下降，人们可以获得自己的基因序列，并利用其来治疗疾病。
- **万维网：**在编写本书第 1 版时，万维网尚不存在，而现在万维网已经改变了整个社会。在许多地方，它已取代了传统的图书馆和报纸。
- **搜索引擎：**随着万维网规模的扩大和价值的与日俱增，如何快速精确地找到所需信息变得越来越重要。今天，如果没有搜索引擎，许多人在万维网中将寸步难行。

显而易见，计算机技术的进步几乎影响着社会的每一个方面。硬件的进步使得程序员可以编写出各种优秀的应用软件，进而证实计算机几乎是无所不能的。今天的科学幻想在未来就会成为现实，诸如虚拟现实、无现金社会和无人驾驶汽车等。

1.1.1 计算应用的分类及其特性

从智能家电到手机再到最大型超级计算机，它们虽然使用了一套通用的硬件技术（参见1.4和1.5节），但这些不同的应用有着不同的设计需求，并以不同的方式通过硬件实现。概括地说，计算机主要包括以下三类应用：

个人计算机（Personal Computer, PC）也许是最为人所知的应用方式，本书的读者几乎都在大量使用。个人计算机强调对单用户提供良好的性能，价格低廉，通常运行第三方软件。尽管此类应用的出现只有短短的35年，但它推动了许多计算技术的革新。

- ➊ 个人计算机：用于个人使用的计算机，通常包含图形显示器、键盘和鼠标等。

服务器（server）是过去被称为大型机的现代形式，通常借助网络访问。服务器适用于执行大负载任务，可以执行单个复杂应用（科学的或工程的），也可以处理大量的简单作业，如大型Web服务器。这些应用通常基于其他来源的软件（例如数据库或仿真软件），并且往往为了特别的需要而加以修改或定制。服务器的制造技术和桌面计算机差不多，但能够提供更强的计算、存储和I/O能力。通常情况下，当发生故障时，服务器比个人计算机恢复的代价高得多，因此服务器更加强调可靠性。

- ➋ 服务器：用于为多用户运行大型程序的计算机，通常由多个用户并行使用，并且一般通过网络访问。

服务器的功能和价格有很大的伸缩范围。低端服务器可能比桌面计算机稍微贵些，不带显示器和键盘的大约需要1000美元，一般用于文档存储、小型商务应用或者简单的Web服务（见6.10节）。高端服务器称为**超级计算机**（supercomputer），一般由成百上千台处理器组成，内存为**terabyte**级，其价格可高达数千万甚至上亿美元。它们主要用于高端科学和工程计算，如天气预报、石油勘探、蛋白质结构计算和其他大规模问题。虽然这类超级计算机代表了最高的计算能力，但是它们只占服务器相对很小的一部分，在整个计算机市场份额中所占比例也很小。

- ➌ 超级计算机：具有最高性能和最高成本的一类计算机，一般配置为服务器，需要花费数千万甚至数亿美元。
- ➍ terabyte：一般简写作TB，原始定义为 $1\,099\,511\,627\,776 (2^{40})$ 字节，但有些通信和辅助存储系统将其重新定义为 $1\,000\,000\,000\,000 (10^{12})$ 字节。为了避免混淆，使用术语tebibyte(TiB)表示 2^{40} 字节，而terabyte指 10^{12} 字节。图1-1表示了十进制和二进制术语的范围。

十进制术语	缩写	数值	二进制术语	缩写	数值	数值差别
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

图1-1 通过为常用容量加一个二进制注释解决 2^x 与 10^y 字节的模糊性。最后一列表示了二进制术语大于相应的十进制术语的具体数值。在以bit为单位时，这些表示方法同样适用，因此gigabit(Gb)是 10^9 bit，而gigabit(Gib)是 2^{30} bit

嵌入式计算机（embedded computer）是数量最多的一类计算机，应用和性能范围十分广泛，包括汽车、电视中的微处理器以及用来控制飞机和货船的处理器网络。嵌入式计算系统的设计目标是运行单一应用程序或者一组相关的应用程序，并且通常和硬件集成在一起以单一系统的方式一并交付用户。因此，尽管嵌入式计算机的数量庞大，还是有很多用户从来没有意识到他们正在使用计算机。

● 嵌入式计算机：嵌入到其他设备中的计算机，一般运行预定义的一个或者一组应用程序。

面向单一应用需求的嵌入式应用通常对成本或功耗有严格限制。以音乐播放器为例，处理器只需尽量快速地执行有限的功能，除此之外，降低成本和功耗是最大的目标。除了低成本的要求之外，由于故障可能会让使用者感到不适（例如，新电视机无法正常收看节目），也可能会导致安全事故（例如，飞机或货船失事），因此嵌入式计算机对故障非常敏感。在面向消费者的嵌入式应用中（如数字家电）一般通过简单设计来获得可靠性——其重点在于尽可能地保证一项功能的正常运转。而在大型嵌入式系统中，采用了在服务器领域应用的多种冗余技术。尽管本书将重点放在通用计算机上，但是大多数概念可直接或者稍微修改之后用于嵌入式计算机。

01 精解 本书中的精解是正文中的一些段落，主要用来对读者可能感兴趣的内容做深入介绍。由于它并不影响后续内容的学习，因此对此部分不感兴趣的读者可以直接跳过。

许多嵌入式处理器使用处理器核。处理器核是利用硬件描述语言（如 Verilog 或 VHDL，见第 4 章）描述的处理器版本，它使得设计者能够把其他专用硬件与之集成起来制造在一块芯片上。

1.1.2 欢迎来到后 PC 时代

技术的持续进步给计算机硬件带来了革命性的变化，对整个信息技术工业产生了震动。就像 30 年前开始出现的个人计算机对产业带来的变化一样，我们已经从本书的上一版开始感受到这种变化。代替 PC 的是个人移动设备（Personal Mobile Device，PMD）。PMD 由电池供电，通过无线方式连接到网络，价格通常只有几百美元。另外，与 PC 一样，PMD 可下载软件（App）并进行运行。与 PC 不同的是，PMD 不再有键盘和鼠标，而采用触摸屏甚至语音作为输入。当今的 PMD 可以是智能手机或平板电脑，而明天的 PMD 可能会包括电子眼镜。图 1-2 给出了平板电脑和智能手机与 PC 和传统手机的增长速度的对比。

● 个人移动设备：连接到网络上的小型无线设备。PMD 由电池供电，通过下载 App 的方式安装软件。智能手机和平板电脑是典型的 PMD。

云计算（cloud computing）替代了传统的服务器，它依赖于称为仓储规模计算机（Warehouse Scale Computer，WSC）的巨型数据中心。像 Amazon 和 Google 这样的公司构建了包含 100 000 台服务器的 WSC，一些公司可以租用其中一部分为 PMD 提供软件服务，而不用自己构建 WSC。事实上，与 PMD 和 WSC 是硬件工业的革命类似，通过云计算实现的软件即服务（Software as a Service，SaaS）是软件工业的革命。当今的软件开发者通常在 PMD 和云上各运行其应用的一部分。

● 云计算：在网络上提供服务的大服务器集群，一些运营商根据应用需求出租不同数量的服务器。

- ② 软件即服务：在网络上以服务的方式提供软件和数据。其运行方式通常不是在本地设备上运行所有的二进制代码，而是通过诸如运行在本地客户端的浏览器等小程序登录到远程服务器上执行。典型的例子是 Web 搜索和社交网络。

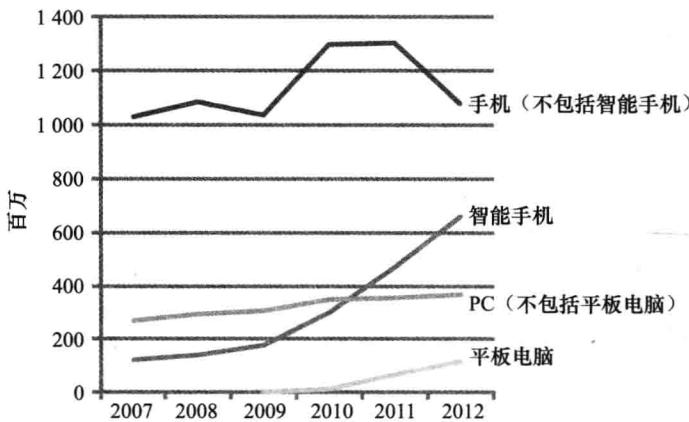


图 1-2 每年生产的平板电脑和智能手机与 PC 和传统手机的数量对比。平板电脑和智能手机代表着后 PC 时代。智能手机反映了手机工业的近期增长情况，并且在 2011 年超过了 PC 的产量。平板电脑产量增长最快，2012 年的产量几乎是 2011 年的两倍。PC 和传统手机的产量保持不变甚至在下降

1.1.3 你能从本书学到什么

成功的程序员总是关心其程序的性能，因为让用户快速得到结果对软件成功与否至关重要。在 20 世纪六七十年代，限制计算机性能的主要因素是内存容量。因而那时候程序员的信条是尽量少占用内存空间，以加速程序的运行速度。近十多年来，计算机和内存的设计技术有了长足进步。除了嵌入式系统以外，大多数用户对少占内存容量的需求大大减轻了。
7

现在，关心性能的程序员需要十分明确，20 世纪 60 年代的简单存储模型已经不复存在，现代计算机的特征是处理器的并行性和内存的层次性。另外，当今的程序员需要考虑运行在 PMD 或云上的程序的能效，这就要求他们了解自己的代码之下的很多细节（见 1.7 节）。因此，程序员为了创建有竞争力的软件版本，必须增加对计算机组成认知。

我们很荣幸有机会为你解释这些知识，阐述机箱覆盖之下的计算机内部软硬件是如何工作的。当你读完本书之后，我们相信，你将能够理解下面的问题：

- 用 C 或 Java 等高级语言编写的程序如何翻译成硬件之间的语言？硬件如何执行程序？领会这些概念是理解软硬件两者如何影响程序性能的基础。
- 什么是软硬件之间的接口，以及软件如何指导硬件完成其功能？这些概念对于许多软件的编写是十分重要的。
- 哪些因素决定了程序的性能？程序员如何才能改进其程序性能？从本书中我们将知道，程序性能取决于原始程序、将该程序转换为计算机语言的软件以及执行该程序的硬件的有效性。
- 什么技术可供硬件设计者用于改进性能？本书将介绍现代计算机设计的基本概念。有兴趣的读者可深入阅读我们的另一本进阶教材《Computer Architecture: A Quantitative Approach》。
- 硬件设计者可使用哪些技术提高能效？什么技术可供程序员提高或降低能效？
- 为什么串行处理近来发展为并行处理？这种发展带来的结果是什么？本书给出了解释，

并介绍了当今支持并行处理的硬件机制，全面评述了新一代的**多核微处理器**（multicore microprocessor）（见第6章）。

- 自1951年第一台商用计算机开始，计算机架构师们提出的哪些伟大的思想构成了现代计算机的基础？
- ② 多核微处理器：在一块集成电路上包含多个处理器（“核”）的微处理器。

8

如果无法理解这些问题，那么要在现代计算机上提升程序性能，或者要评估不同计算机解决特定问题的优劣将会是一个反复实验的复杂过程，而不是一个深入分析的科学过程。

本书第1章的目的是为其余各章奠定良好的基础。它介绍了各种基本概念和定义，指出如何正确地剖析软硬件，以及如何评价性能与功耗。它还介绍了集成电路（为计算机革命提供动力的技术），并在最后解释了向多核转移的原因。

在本章和后面几章里，读者会看到许多新的术语或者一些你听过却不知道其含义的术语。但是不用担心，在描述现代计算机时，确实会有很多专用术语，它们使我们能够精确描述计算机的功能或性能。另外，计算机设计人员（包括本书作者）喜欢用**首字母缩略词**（acronym），一旦知道了每个字母代表什么，就很容易理解了。为了帮助读者理解和记忆这些专用术语，在术语第一次出现时，我们会给出明确的定义。通过与这些术语的短时间接触，你将会熟练地正确使用这些术语的缩写，例如 BIOS、CPU、DIMM、DRAM、PCIe、SATA 和许多其他术语。

- ② 首字母缩略词：由一串单词中每个单词的首字母相连构成的单词。例如 RAM 是随机访问存储器（Random Access Memory）的缩略词，CPU 是中央处理单元（Central Process Unit）的缩略词。

为了加强对软件和硬件对于程序运行性能影响的理解，我们在全书中特别插入了“理解程序性能”，来对程序性能的理解加以概括。下面就是第一个。

01 理解程序性能 一个程序的性能取决于以下各因素的组合：程序所用算法的有效性，用来建立程序并将其翻译成机器指令的软件系统，计算机执行机器指令（可能包括 I/O 操作）的有效性。下表总结了硬件和软件是如何影响性能的。

软件或硬件组成元素	该元素如何影响性能	该论题出现的位置
算法	决定了源码级语句的数量和 I/O 操作的数量	其他书
编程语言、编译器和体系结构	决定了每条源码级语句对应的计算机指令数量	第2、3章
处理器和存储系统	决定了指令的执行速度	第4、5、6章
I/O 系统（硬件和操作系统）	决定了 I/O 操作可能的执行速度	第4、5、6章

9

为了说明本书中的思想的作用，在一连串章节中对完成一个矩阵与一个向量相乘的 C 程序进行了性能优化。每一步都可以帮助我们理解现代微处理器硬件如何能使性能提高 200 倍！

- 在第3章的数据级并行部分，使用C语言固有的子字并行使性能提高3.8倍。
- 在第4章的指令级并行部分，使用循环展开开发多指令发射和乱序执行硬件使性能再提高2.3倍。
- 在第5章的存储器层次优化部分，使用阻塞cache将大型矩阵处理性能再次提高2.5倍。

- 在第 6 章的线程级并行部分，在 OpenMP 中使用循环并行来开发多核硬件使性能再次提高 14 倍。

01 小测验

“小测验”的目的是帮助读者评估自己是否掌握了所学的概念，以及是否理解了这些概念的内涵。在这些小测验中，有些只有简单的答案，有些则是为了组内讨论。有些问题的答案可在章末找到。所有小测验只在节末出现，如果你确信自己对该部分内容完全理解，则可以跳过去。

1. 每年嵌入式处理器的售出数量远远超过 PC 处理器甚至后 PC 处理器的数量。根据自己的经验，你是支持还是反对这种看法？列举你家中使用的嵌入式处理器，它与你家中桌面处理器的数量相比如何？
2. 如前所述，软件和硬件都会影响程序的性能。请思考下述哪个例子属于性能瓶颈。
 - 所选算法
 - 编程语言或编译程序
 - 操作系统
 - 处理器
 - I/O 系统和设备

10

1.2 计算机系统结构中的 8 个伟大思想

现在介绍计算机设计者在过去 60 年的计算机设计中提出的 8 个伟大思想。这些思想非常有用，以至于在首台应用它们的计算机之后的很长时间里，设计师在设计新的处理器时都会使用这些思想。这些思想将会贯穿本章和后续章节。为了说明它们的影响，本节对这些思想的含义以及亮点进行介绍，在本书的后续章节中将会明确使用它们近 100 次。

1.2.1 面向摩尔定律的设计

计算机设计者面临的一个永恒的问题就是摩尔定律（Moore's Law）驱动的快速变化。摩尔定律指出单芯片上的集成度每 18 ~ 24 个月翻一番。摩尔定律是 Intel 公司的创始人之一 Gordon Moore 在 1965 年对集成电路集成度做出的预测。由于计算机设计需要几年时间，因此在项目结束时，单芯片的集成度相对于设计开始时很容易翻一番甚至翻两番。像一个双向飞碟运动员一样，计算机设计者必须预测其设计完成时的工艺水平，而不是设计开始时的。

1.2.2 使用抽象简化设计

计算机架构师和程序员必须发明能够提高产量的技术，否则设计时间也将会向资源规模一样按照摩尔定律增长。提高硬件和软件生产率的主要技术之一是使用抽象（abstraction）来表示不同的设计层次，在高层次中看不到低层次的细节，只能看到一个简化的模型。

1.2.3 加速大概率事件

加速大概率事件（common case fast）远比优化小概率事件更能提高性能。大概率事件通常比小概率事件简单，从而易于提高。大概率事件规则意味着设计者需要知道什么事件是经常发生的，这只有通过仔细的实验与评估才能得出（见 1.6 节）。可以把加速大概率事想象成一辆赛车，由于通常情况下只有一两名乘客，因此提高赛车的速度要比提高小型货车的速度容易。

11

1.2.4 通过并行提高性能

从计算的诞生开始，计算机设计者就通过并行执行操作来提高性能。在本书中将会看到许多并行性能（parallel performance）的例子。

1.2.5 通过流水线提高性能

在计算机系统结构中，一个特别的并行性场景就是流水线（pipelining）。例如许多西部片中，一些坏人在制造火灾，在消防车出现之前会有一个“消防队列”来灭火——小镇的居民们排成一排通过水桶接力快速将水桶从水源传至火场，而不是每个人都在来回奔跑。可以把流水线想象成一系列水管，其中每一块代表一个流水级。

1.2.6 通过预测提高性能

遵循谚语“求人准许不如求人原谅”，最后一个伟大的思想就是预测（prediction）。在某些情况下，如果假定从误预测恢复执行代价不高并且预测的准确率相对较高，则通过猜测的方式提前开始某些操作，要比等到确切知道这些操作应该启动时才开始要快一些。

1.2.7 存储器层次

由于存储器的速度通常影响性能、存储器的容量限制了解题的规模、当今计算系统中存储器的代价占了主要部分，因此程序员希望存储器速度更快、容量更大、价格更便宜。设计师们发现可以通过存储器层次（hierarchy of memory）来解决这些相互矛盾的需求。在存储器层次中，速度最快、容量最小并且每位价格最昂贵的存储器处于顶层，而速度最慢、容量最大且每位价格最便宜的存储器处于最底层。在第5章将会看到，程序员看到的主存同时具有存储器层次中顶层的高速度和底层中的大容量和便宜的特征。可以把存储器层次想象成一个堆叠的三角形，该形状表示速度、价格和容量：越靠近顶端，存储器速度越快、每位价格越高；底层宽度越大，存储器容量越大。

1.2.8 通过冗余提高可靠性

计算机不仅需要速度快，还需要工作可靠。由于任何一个物理器件都有可能失效，因此可以通过使用冗余部件的方式提高系统的可靠性（dependable），冗余部件可以替代失效部件并可以帮助检测错误。可以通过牵引式挂车来理解可靠性：牵引式挂车后轴两边具有双轮胎，在一个轮胎出问题时卡车仍然可以继续工作。（在一个轮胎出问题时，卡车司机立即直接开往修理厂进行修理，从而恢复冗余性。）

12

1.3 程序概念入门

在巴黎，我对当地人讲法语，他们只是瞪着我看；我从来没能让这些白痴理解他们自己的语言。

——马克·吐温，《异国奇遇》，1869

一个典型的应用程序，如字处理程序或大型数据库系统，可以由数百万行代码构成，并依靠软件库来实现异常复杂的功能。众所周知，计算机中的硬件只能执行极为简单的低级指令。从复杂的应用程序到简单的指令需要经过几个软件层次来将复杂的高层次操作逐步解释或翻译

成简单的计算机指令，这可以作为伟大思想抽象的一个例子。

图 1-3 给出了这些软件的层次结构，外层是应用软件，中心是硬件，**系统软件**（systems software）位于两者之间。

- **系统软件**：提供常用服务的软件，包括操作系统、编译程序、加载程序和汇编程序等。

系统软件有很多种，其中有两种对于现代计算机系统来说是必需的：操作系统和编译程序。**操作系统**（operating system）是用户程序和硬件之间的接口，为用户提供各种服务和监控功能。操作系统最为重要的作用是：

- 处理基本的输入和输出操作。
- 分配外存和内存。
- 为多个应用程序提供共享计算机资源的服务。

当前我们使用的操作系统主要有 Linux、iOS 和 Windows。

13 ● **操作系统**：为了使程序更好地在计算机上运行而管理计算机资源的监控程序。

编译程序（compiler）完成另外一项重要功能：把用高级语言（如 C、C++、Java 或 Visual Basic 等）编写的程序翻译成硬件能执行的指令。这个翻译过程是相当复杂的，这里仅作简要介绍，第 2 章和附录 A 将作深入介绍。

- 编译程序：将高级语言翻译为计算机所能识别的机器语言的程序。

从高级语言到硬件语言

谈到电子硬件，首先需要谈到电信号的发送。对于计算机来说，最简单的信号是“通”和“断”。因此，计算机只用 2 个字母来表示。正如英语 26 个字母写多少不受限制一样，计算机的 2 个字母写多少也不受限制。代表两个字母的符号是 0 和 1，我们通常认为计算机语言就是二进制数。每个字母就是二进制元数字中的一个二进制位（binary digit）或一位（bit）。计算机服从于我们的命令，即计算机术语中的指令（instruction）。指令是能被计算机识别并执行的位串，可以将其视为数字。例如，位串

1000110010100000

告诉计算机将 2 个数相加。第 2 章将解释为什么数字元既表示指令又表示数据。我们不希望在此处涉及第 2 章的具体内容，但是使用数字既表述指令又表示数据是计算机的基础。

- **二进制位**：也称为位。基数为 2 的数字中的 0 或 1，它是信息的基本组成元素。
- **指令**：计算机硬件所能理解并服从的命令。

第一代程序员是直接使用二进制数与计算机通信的，这是一项非常乏味的工作。所以他们很快发明了助记符，以符合人类的思维方式。最初助记符是手工翻译成二进制的，其过程显然过于烦琐。随后设计人员开发了一种称为汇编程序（assembler）的软件，可以将助记符形式的指令自动翻译成对应的二进制。例如，程序员写下

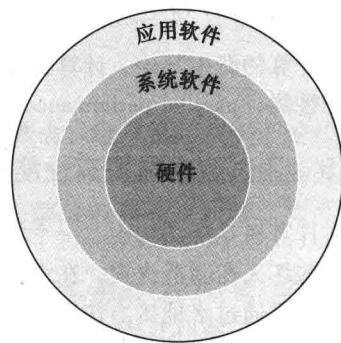


图 1-3 简化的硬件和软件层次图，将硬件作为同心圆的中心，应用程序软件作为最外层。在复杂的应用中，通常存在多层应用软件层。例如，一个数据库系统可运行于系统软件之上，而驻留在该系统软件上的某应用又反过来运行在该数据库之上

add A, B

汇编程序会将该符号翻译成

1000110010100000

该指令告诉计算机将 A 和 B 两个数相加。这种符号语言的名称今天还在用，即汇编语言（assembly language）。而机器可以理解的二进制语言是机器语言（machine language）。

- 汇编程序：将指令由助记符形式翻译成二进制形式的程序。
 - 汇编语言：以助记符形式表示的机器指令。
 - 机器语言：以二进制元形式表示的机器指令。

虽然这是一个巨大的进步，但汇编语言仍然与科学家用来模拟液体流动或会计师用来结算账目所使用的符号相去甚远。汇编语言需要程序员写出计算机执行的每条指令，要求程序员像计算机一样思考。

认识到可以编写一个程序来将需要更强大的高级语言翻译成计算机指令是计算机早期的一个重大突破。高级编程语言 (high-level programming language) 及其编译程序大大地提高了软件的生产率。图 1-4 表示了这些程序和编程语言之间的关系，这是抽象的另外一个例子。

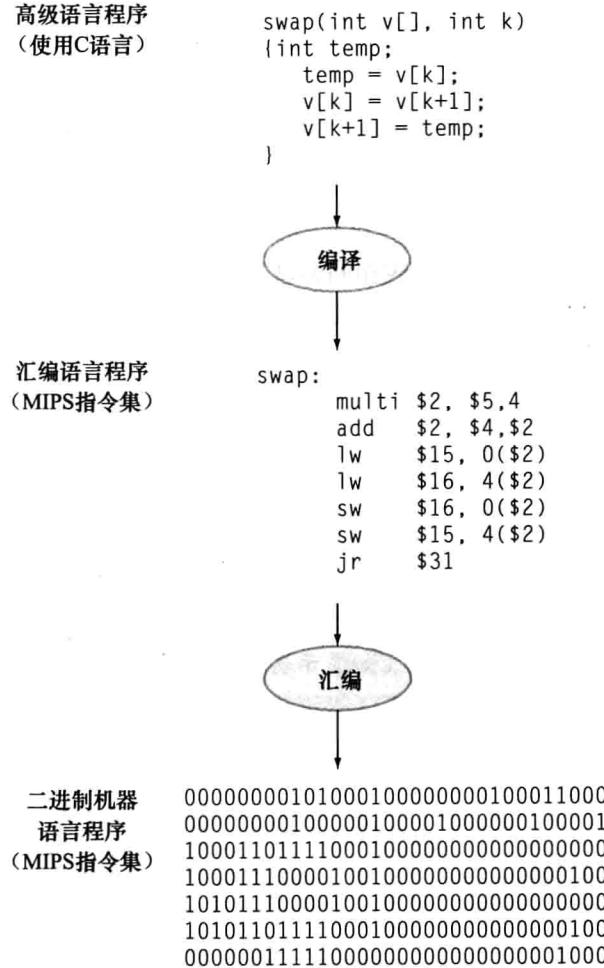


图 1-4 C 程序编译为汇编语言程序，再汇编为二进制机器语言程序。尽管将高级语言翻译成二进制的机器语言仅需要两步，但一些编译器将“中间人”砍掉，直接产生二进制的机器语言。这些语言和本图中列举的程序将在第 2 章详细检测

- ② 高级编程语言：如 C、C++、Java、Visual Basic 等可移植的语言，由一些单词和代数符号组成，可以由编译器转换为汇编语言。

编译程序使得程序员可以写出高级语言表达式：

A + B

编译程序将其编译为如下的汇编语言语句：

add A,B

然后，汇编程序将此语句翻译为二进制元指令，告诉计算机将这两个数 A 和 B 相加。

使用高级编程语言有以下几个好处。第一，可以使程序员用更自然的语言来思考，用英文和代数符号来表示，形成的程序看起来更像文字而不是密码表（见图 1-4）。而且，它们可按用途进行设计。例如，Fortran 是为科学计算设计的，Cobol 是为商业数据操作设计的，Lisp 是为符号操作设计的，等等。还有一些特定领域的语言，只为少数专业人群设计，如流体仿真的研究人员等。

第二，高级语言提高了程序员的生产率。如果使用较少行数的编程语言即可表示出设计用意，则可加速程序的开发，这是软件开发方面少有的共识之一。简明性是高级语言相对汇编语言最为明显的优势。

第三，采用高级语言编写程序提高了程序相对于计算机的独立性，因为编译程序和汇编程序能够把高级语言程序翻译成任何计算机的二进制元指令。高级编程语言的这些好处，使其直到今天仍应用广泛。

1.4 硬件概念入门

我们已经在上节通过程序揭示了计算机软件，在本节中我们将打开机箱盖学习其中的硬件。任何一台计算机的基础硬件都要完成相同的基本功能：输入数据、输出数据、处理数据和存储数据。本书的主题就是描述这些功能是怎样完成的，随后各章将分别讨论这 4 项任务。

本书在遇到重要知识点时，都会用“重点”标题加以强调，希望读者对其重点记忆。全书大致有 10 多个重要知识点，这里是第一个，即计算机是由完成输入、输出、处理和存储数据任务的 5 个部件构成的。

计算机的两个关键部件是输入设备（input device）和输出设备（output device），例如麦克风是输入设备，而扬声器是输出设备。输入为计算机提供数据，输出将计算结果送给用户。像无线网络等设备既是输入设备又是输出设备。

- ② 输入设备：为计算机提供信息的装置，如键盘。
- ② 输出设备：将计算结果输出给用户（如显示器）或其他计算机装置。

第 5 章和第 6 章将详细介绍 I/O 设备，这里由外部 I/O 设备开始先对计算机硬件做一些基本的介绍。

- 01 重点** 组成计算机的 5 个经典部件是输入、输出、存储器、数据通路（在计算机中也称运算器）和控制器，其中最后两个部件通常合称为处理器。图 1-5 表示了一台计算机的标准组成。该组成与硬件技术无关，你总能够把任何计算机（无论是现在的还是过去的）中的任何部件归于这 5 种之一。为了加深读者对这一重点的印象，我们将在每章开始都给出此图。

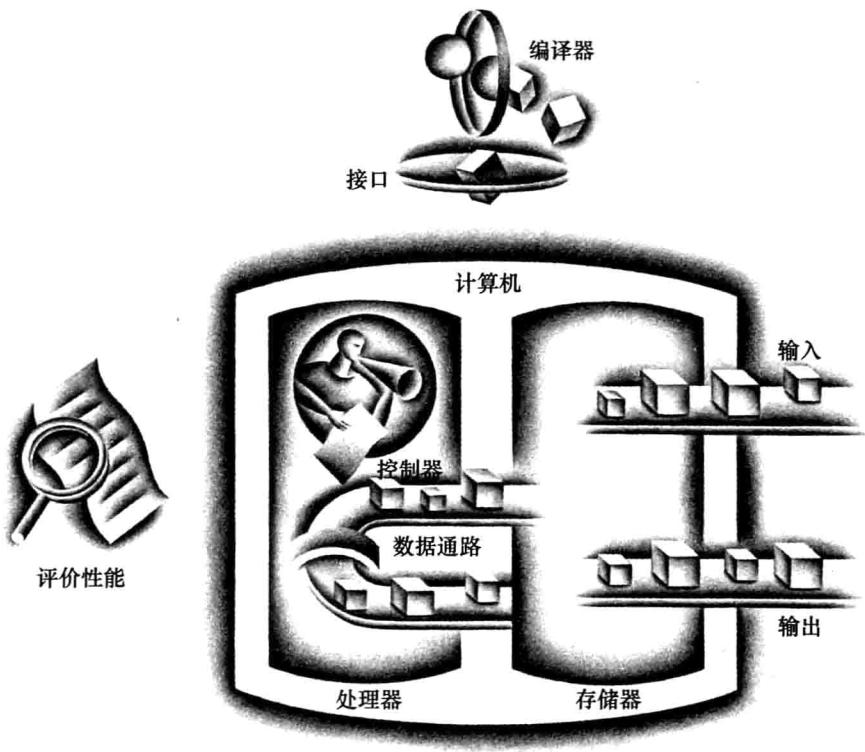


图 1-5 组成计算机的 5 个经典部件。处理器从存储器中得到指令和数据，输入部件将数据写入存储器，输出部件从内存中读出数据，控制器向数据通路、存储器、输入和输出部件发出命令信号

17

1.4.1 显示器

最吸引人的 I/O 设备应该是图形显示器了。大多数个人移动设备都用液晶显示 (Liquid Crystal Display, LCD) 来获得轻巧、低功耗的显示效果。LCD 并非光源，而是控制光的传输。典型的 LCD 内含棒状液态分子团形成的转动螺旋线，用来弯曲来自显示器后方的光线或者少量的反射光线。当电流通过时，液态分子棒不再弯曲，也不再使光线弯曲，由于两层相互垂直的偏光板之间充满液晶材料，如果它不弯曲则光线不能通过。（在不施加任何电压的情况下，液晶处于初始状态，并将入射光的方向扭转 90°，让背光源的入射光能够通过整个结构，在显示屏上呈现白色；而当施加电压时，光线不再弯曲，显示屏呈现为黑色。）今天，大多数 LCD 显示器采用动态矩阵显示 (active matrix display) 技术，其每个像素 (pixel) 都由一个晶体管精确地控制电流，使图像更清晰。在彩色动态矩阵 LCD 中，还有一个红 - 绿 - 蓝屏决定三种颜色分量的强度，每个点需要 3 个晶体管开关。

- ➊ 液晶显示：这是一种显示技术，用液体聚合物薄层的带电或者不带电来传输或者阻止光线的传输。
- ➋ 动态矩阵显示：一种液晶显示技术，使用晶体管控制单个像素上光线的传输。
- ➌ 像素：图像元素的最小单元。屏幕由成千上万的像素组成的矩阵而形成。

通过计算机显示器，我将飞机降落航空母舰的甲板上，观察到一个原子打到势阱中，乘着火箭以接近光的速度飞翔，同时我了解到计算机最深层的工作原理。

——Ivan Sutherland，计算机图形学之父，《Scientific American》，1984

图像由像素矩阵组成，可以表示成二进制位的矩阵，称为位图（bit map）。针对不同的屏幕尺寸及分辨率，典型的屏幕上显示矩阵的大小可以从 1024×768 到 2048×1536 。彩色显示器使用 8 位来表示每个三原色（红、绿和蓝），每个像素用 24 位表示，可以显示百万种不同的颜色。

计算机硬件采用光栅刷新缓冲区（又称为帧缓冲区）来保存位图以支持图像。要显示的图像保存在帧缓冲区中，每个像素的二进制值以刷新频率读出到显示设备。图 1-6 显示了一个用 4 位表示一个像素的简化设计的帧缓冲区。

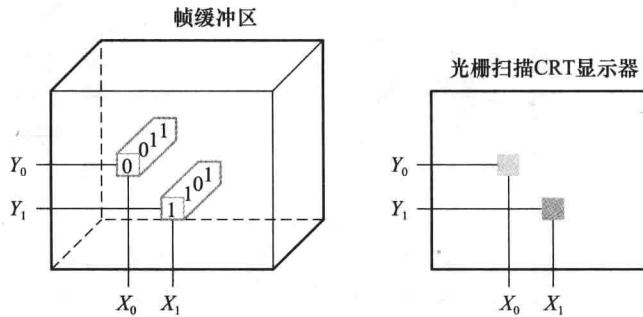


图 1-6 左边的帧缓冲区中每个坐标决定了右边光栅扫描 CRT 显示中相应坐标的灰度。像素 (X_0, Y_0) 的灰度值是 0011，小于像素 (X_1, Y_1) 的灰度值， (X_1, Y_1) 的灰度值是 1101

使用位图的目的是如实地在屏幕上进行显示。因为人眼可以分辨出屏幕上的细小变化，所以图像系统面临着挑战。

18

1.4.2 触摸屏

PC 使用 LCD，而后 PC 时代的平板电脑和智能手机使用接触敏感的显示设备替代了键盘和鼠标，拥有良好的用户界面，用户直接指向感兴趣的内容，而不需要使用鼠标。

触摸屏可采用多种方式实现，许多平板电脑采用电容感应实现。如果绝缘玻璃上覆盖一层透明的导体，人的手指接触到屏幕范围时，由于人是导体，将会使屏幕的电场发生变化，进而导致电容的变化。这种技术允许同时接触多个点，可提供非常好的用户界面。

1.4.3 打开机箱

图 1-7 给出了 Apple iPad 2 平板电脑的内部结构。不难看出，在计算机的五大传统部件中的 I/O 是该设备的主要部分。iPad 2 的 I/O 设备包括一个电容性的多触点 LCD、前置摄像头、后置摄像头、麦克风、耳机插孔、扬声器、加速计、陀螺仪、Wi-Fi 网络和蓝牙网络。其数据通路、控制器和存储器只占很小一部分。

图 1-8 中的小长方形是集成电路（integrated circuit），俗称芯片（chip）。其中心标有 A5 的芯片中含有两个运行频率为 1GHz 的 ARM 处理器。处理器是计算机中最活跃的部分。它严格按照程序中的指令运行，将数字相加，测试结果，并按结果发出控制信号使 I/O 设备做出动作，等等。有时候，人们把处理器称为中央处理单元（central processor unit），即 CPU。

- ② 集成电路：也叫芯片，一种将几十个至几百万个晶体管连接起来的设备。
- ② 中央处理器单元：也称为处理器，处理器是计算机中最活跃的部分，它包括数据通路和控制器，能将数字相加，测试结果，并按结果发出控制信号使 I/O 设备动作等。

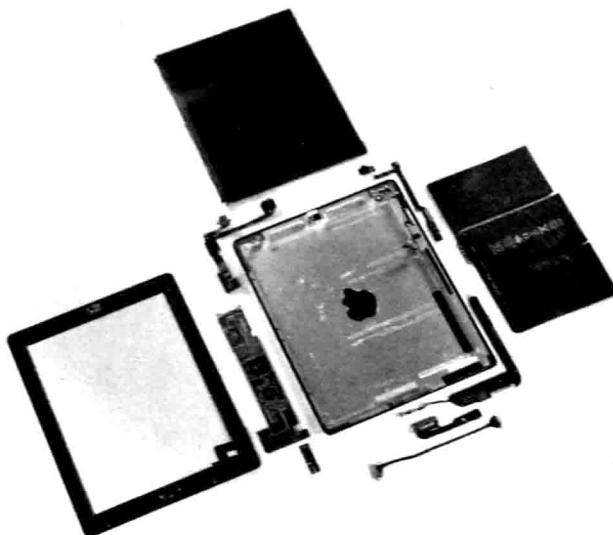


图 1-7 Apple iPad2 A1395 的组成。中间是 iPad 的金属背板（中心是倒置的 Apple 标志），顶部是电容性触摸屏和 LCD。最右端是 3.8V、25W·h 的聚合物电池，它包含三块锂离子电池芯，可以供电 10 小时。最左端是将 LCD 固定在背板上的金属外壳。在金属背板周围的小部件组成了我们熟知的计算机，它们在金属壳内位于电池旁边，呈 L 型排布。图 1-8 显示了靠近金属外壳左下部 L 型的逻辑主板的详细情况，上面有处理器和存储器，其下面的小方块中包含了提供无线通信的芯片，即 Wi-Fi、蓝牙和调频调谐器，它可以插在逻辑主板左下角的插槽中。外壳左上角是另外一个 L 型部件，它是前置摄像头组件，包括摄像头、耳机插孔和麦克风。外壳右上角的电路板除了加速计和陀螺仪，还包含了音量控制和静音/屏幕旋转锁定按钮。加速计和陀螺仪使得 iPad 可识别 6 向移动。旁边的小方块是后置摄像头。外壳右下角是 L 型的扬声器组件。底部的电缆连接逻辑主板和摄像/声音控制电路板。电缆和扬声器组件之间的电路板是电容性触摸屏的控制器（iFixit 友情提供，www.ifixit.com）

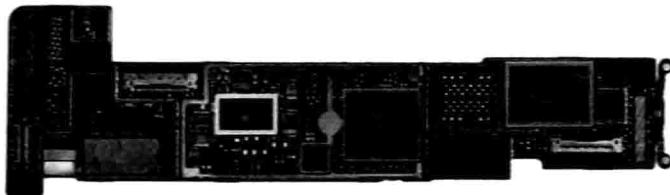


图 1-8 图 1-7 中 Apple iPad 2 的逻辑主板。图中突出了 5 块集成电路。中部的大 IC 是 Apple A5 芯片，包含了一个主频 1GHz 的双核 ARM 处理器和 512MB 的主存。图 1-19 是 A5 中处理器芯片的照片。左边大小相当的芯片是 32GB 的非易失性的闪存芯片。两块芯片之间的空间可以安装第二块存储器来扩展 iPad 的存储容量。A5 右边的芯片包含了电源控制和 I/O 控制芯片（iFixit 友情提供，www.ifixit.com）

为进一步理解硬件，图 1-9 展示了一款微处理器的内部细节。处理器从逻辑上包括两个主要部件：数据通路和控制器，分别相当于处理器的肌肉和大脑。数据通路（datapath）负责完成算术运算，控制器（control）负责指导数据通路、存储器和 I/O 设备按照程序的指令正确执行。第 4 章将进一步详细说明数据通路和控制器。

- 数据通路：是处理器中执行算术操作的部分。
- 控制器：处理器中根据程序的指令指挥数据通路、存储器和 I/O 设备的部分。



图 1-9 A5 内部的处理器集成电路。芯片尺寸为 $12.1\text{mm} \times 10.1\text{mm}$ ，采用 45nm 工艺制造（见 1.5 节）。左半部分靠中间的位置是两个相同的 ARM 处理器，左上角的四分之一部分是具有 4 条数据通路的图形处理器单元（Graphic Processor Unit, GPU），左下角和底部是与主存的接口（Chipworks 友情提供，www.chipworks.com）

图 1-8 中的 A5 芯片中还有两块存储器芯片，每块容量 2 gigabit，共 512MiB。内存（memory）是程序运行时的存储空间，它同时也用于保存程序运行时所使用的数据。内存由 DRAM 芯片组成。DRAM 是 dynamic random access memory（动态随机访问存储器）的缩写。内存由多片 DRAM 芯片组成，用来承载程序的指令和数据。与串行访问内存（如磁带）不同的是，无论数据存储在什么位置，DRAM 访问内存所需的时间基本相同。

- ① 内存：程序运行时的存储空间，同时还存储程序运行时所需的数据。
- ② DRAM：动态随机访问存储器，集成电路形式的存储器，可随机访问任何地址的内存。在 2012 年，其访问时间为 50ns，每 gigabyte 的价格为 5~10 美元。

进一步深入了解任何一个硬件部件会加深对计算机的理解。在处理器内部使用的是另外一

种存储器——缓存。缓存（cache memory）是一种小而快的存储器，一般作为 DRAM 的缓冲（缓存的一个非技术性的定义是一个隐藏事物的安全地方）。cache 采用的是另一种存储技术，称为静态随机访问存储器（Static Random Access Memory，SRAM），其速度更快而且不那么密集，因此价格比 DRAM 更贵（见第 5 章）。SRAM 和 DRAM 是存储器层次中的两层。

- ② 缓存：缓存是一种小而快的存储器，一般作为大而慢的存储器的缓冲。
- ② 静态随机访问存储器：一种集成电路形式的存储器，但是比 DRAM 更快，集成度更低。

19
21

如前所述，改进设计的一个伟大思想是抽象。最重要的抽象之一是硬件和底层软件之间的接口。鉴于其重要性，该抽象被命名为计算机的指令集体体系结构（instruction set architecture），或简称体系结构（architecture）。计算机体系结构包括程序员正确编写二进制机器语言程序所需的全部信息，如指令、I/O 设备等。一般来说，操作系统需要封装 I/O 操作、存储器分配和其他低级的系统功能细节，以便应用程序员无需在这些细节上分心。提供给应用程序员的基本指令集和操作系统接口合称为应用二进制接口（Application Binary Interface，ABI）。

- ② 指令集体体系结构：也叫体系结构，是低层次软件和硬件之间的抽象接口，包含了需要编写正确运行的机器语言程序所需要的全部信息，包括指令、寄存器、存储访问和 I/O 等。
- ② 应用二进制接口：用户部分的指令加上应用程序员调用的操作系统接口，定义了二进制层次可移植的计算机的标准。

计算机体系结构可以让计算机设计者独立地讨论功能，而不必考虑具体硬件。例如，我们讨论数字时钟的功能（如计时、显示时间、设置闹钟）时，可以不涉及时钟的硬件（如石英晶体、LED 显示、按钮）。计算机设计者将体系结构与体系结构的实现（implementation）分开考虑也是沿着同样的思路：硬件的实现方式必须依照体系结构的抽象。这些概念产生了另一个重点。

- ② 实现：遵循体系结构抽象的硬件。

01 重点 无论硬件还是软件都可以使用抽象分成多个层次，每个较低的层次把细节对上层隐藏起来。抽象层次中的一个关键接口是指令集体体系结构——硬件和底层软件之间的接口。这一抽象接口使得同一软件可以由成本不同、性能也不同的实现方法来完成。

22

1.4.4 数据安全

目前为止，我们已经理解了如何输入数据，如何使用这些数据进行计算，以及如何显示结果。然而，一旦关掉电源，所有数据就丢失了，因为计算机中的内存是易失性存储器（volatile memory）。与之不同的是，如果关掉 DVD 机的电源，所记录的内容将不会丢失，因为 DVD 采用的是非易失性存储器（nonvolatile memory）。

- ② 易失性存储器：类似 DRAM 的存储器，仅在加电时保存数据。
- ② 非易失性存储器：在掉电时仍可保持数据的存储器，用于存储运行间的程序，例如 DVD。

为了区分易失性存储器与非易失性存储器，我们将前者称为主存储器（main memory 或 primary memory），将后者称为二级存储器（secondary memory）。二级存储器形成了存储器层次中下面更低的一层。DRAM 自 1975 年起在主存储器中占主导地位，而磁盘（magnetic disk）在

二级存储器中占主导地位的时间更早。由于器件尺寸和前面所述的特点，非易失性半导体存储器——闪存（flash memory）在个人移动设备中替代了磁盘。图1-8所示的iPad 2中的芯片上包含了闪存。除了非易失性外，闪存比DRAM慢，但却便宜很多。虽然每位的价格高于磁盘，但是闪存在体积、电容、可靠性和能耗方面都优于磁盘。因此闪存是PMD的二级存储器的标准。遗憾的是，与硬盘和DRAM不同的是，闪存具有写100 000~1 000 000次后老化或损坏的弱点。因此，文件系统必须记录写操作的数目，而且具备避免存储器损坏的策略，例如，避免移动经常使用的数据。第5章将会详细介绍磁盘和闪存。

- ② **主存储器：**也叫主要存储器。这个存储器用来保持运行中的程序，在现代计算机中一般由DRAM组成。
- ③ **二级存储器：**非易失性存储器，用来保存两次运行之间的程序和数据。在个人移动设备中一般由闪存组成，在服务器中由磁盘组成。
- ④ **磁盘：**也叫硬盘（hard disk），是使用磁介质材料构成的以旋转盘片为基础的非易失性二级存储设备。因为是旋转的机械设备，所以磁盘的方位时间大约是5~20毫秒，2012年每g字节的价格大约为0.05~0.1美元。
- ⑤ **闪存：**一种非易失性半导体内存，单位价格和速度均低于DRAM，但单位价格比磁盘高，速度比磁盘快。其访问时间大约为5~50毫秒，2012年每g字节的价格大约为0.75~1美元。

1.4.5 与其他计算机通信

我们已经介绍了如何输入、计算、显示和保存数据，但对于今天的计算机来说，还有一项不可缺少的功能：计算机网络。如图1-5所示，处理器与存储器和I/O设备连接。通过网络，计算机可以与其他计算机通信，从而扩展计算能力。当今网络已经十分普遍，逐步成为了计算机系统的主干。一台新型个人移动设备或服务器如果没有网络接口将是十分可笑的。联网的计算机具有如下几个主要优点：

- **通信：**在计算机之间高速交换信息。
- **资源共享：**有些I/O设备可以由网络上的计算机共享，不必每台计算机都配备。
- **远距离访问：**用户可以不必在计算机的旁边，而是在很远的地方使用计算机。

根据传输速度以及信息传输的距离，通信代价随之增长，网络的传输距离和性能是多种多样的，最为普遍的网络类型是以太网。它的传输距离可达到1 000千米，传输速率可达到40Gbps。以太网的传输距离和速率可以将一个建筑物中同一层的计算机连接起来，这就形成了通常称为局域网（Local Area Network，LAN）的一个例子。局域网通过交换机进行连接，可以提供路由与安全服务。广域网可支持万维网（World Wide Web），作为因特网的骨干网，以光纤为基础并向通信公司租用。

- ② **局域网：**一种在一定地理区域（例如在同一栋大楼内）使用的传输数据的网络。
- ③ **广域网：**一种可将区域扩展到几百千米范围的网络。

在过去的30年间，因为广泛的使用和性能的大幅度提升，网络已经改变了计算的方式。在20世纪70年代，个人很难接触到电子邮件，网络和Web还不存在，物理上的邮件介质磁带成为传输两地之间大容量数据的主要载体。局域网根本不存在，少数几个广域网限制了容量和访问。

随着网络技术的进步，网络变得越来越便宜，速度越来越快。在30多年前，第一个标准局域网的最大带宽为10Mbps，支持数十台计算机共享工作。今天，局域网技术已能提供从100Mbps~10Gbps的带宽。光通信技术已经使广域网有了类似的发展，从几百Kbps到Gbps的

带宽，支持几百台到几百万台计算机与全球网络互连。网络规模的飞速扩大，伴随着带宽的急剧增长，使得网络技术成为最近 30 年来信息革命的中心。

最近 10 年来，新的联网创新变革了计算机通信的方式。推动后 PC 时代（PostPC Era）的无线技术广泛应用，加上原本用来生产无线电的廉价的半导体（CMOS）技术被用来生产存储器和微处理器，使其价格大幅度降低，产量剧增。当前无线通信技术（IEEE 标准 802.11）支持从 1Mbps 到近 100Mbps 的传输速率。无线技术和基于线路的网络相当不同，因为所有用户可以在最近的区域里共享电波。

01 小测验

半导体 DRAM、闪存和磁盘存储有很大差别。试从易失性、访问时间和价格三方面进行比较。

1.5 处理器和存储器制造技术

处理器和存储器正在以难以置信的速度进步，因为计算机设计者一直采用最新的电子技术进行设计，以期在竞争中取得优势。图 1-10 描述了不断进步的各种新型技术，包括其出现的时间和性价比。这些技术确定了计算机能够做什么，以及以多快的速度发展变化。我们相信，所有计算机专业人员都应该熟悉集成电路的基础知识。

年代	计算机中采用的技术	相对性价比
1951	真空管	1
1965	晶体管	35
1975	集成电路	900
1995	超大规模集成电路	2 400 000
2013	甚大规模集成电路	250 000 000 000

图 1-10 随着时间的发展，不同计算机实现技术的性价比。来源：波士顿计算机博物馆，其中 2013 年的数据由作者推断得到（见 1.12 节）

晶体管（transistor）仅仅是一种受电流控制的开关。集成电路（IC）是由成千上万个晶体管组成的芯片。当戈登·摩尔预测资源持续翻番时，他是在预测单芯片上晶体管数量的增长速度。为了描述这些晶体管从几百个增长到成千上万的情形，形容词“超大规模”被添加到术语中，简写为 VLSI，即超大规模集成电路（very large-scale integrated circuit）。

- 晶体管：一种由电信号控制的简单开关。
- 超大规模集成电路：由数十万到数百万晶体管组成的电路。

集成度的增长率是相当稳定的。图 1-11 表示自 1977 年以来 DRAM 容量的发展情况。近 20 多年以来，每隔 3 年 DRAM 的容量就增长 4 倍，累积增长已超过 16 000 倍。

为了理解集成电路的制造过程，我们从头开始介绍。芯片的制造从硅（silicon）开始，硅是沙子中的一种物质。因为硅的导电能力不强，因此称为半导体（semiconductor）。用特殊的化学方法对硅添加某些材料，可以把其细微的区域转变为以下三种类型之一：

- 良好的导电体（类似于细微的铜线或铝线）。
 - 良好的绝缘体（类似于塑料或玻璃膜）。
 - 可控的导电体或绝缘体（类似开关）。
- 硅：一种自然元素，它是一种半导体。
 - 半导体：一种导电性能不好的物质。

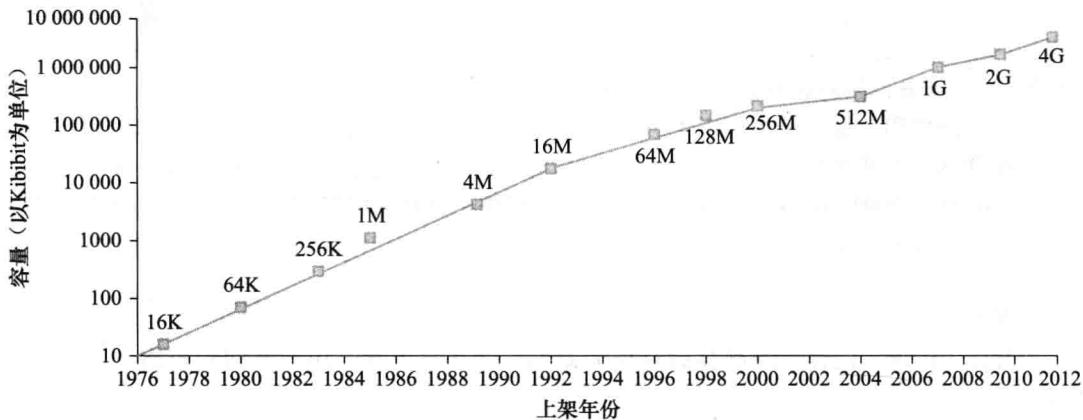


图 1-11 单片 DRAM 容量随时间的增长。纵轴单位为 Kb，其中 K 为 1024 (2^{10} 位)。在近 20 多年中，平均每隔 3 年 DRAM 容量扩大 4 倍，即每年增长约 60%。在最近几年中，增长速度有所下降，接近每 2 ~ 3 年翻一番的水平。

晶体管属于第三种 VLSI 电路，是由数亿个上述三种材料组合起来并封装在一起所制成的。

集成电路的制造过程对芯片的价格非常关键，因此对计算机设计者十分重要。图 1-12 表示了集成电路制造的整个过程。集成电路的制造是从硅锭 (silicon crystal ingot) 开始的，它像一根巨大的香肠。目前使用的硅锭直径约 8 ~ 12 英寸，长度约 12 ~ 24 英寸。硅锭经切片机切成厚度不超过 0.1 英寸的晶圆 (wafer)。这些晶圆经过一系列化学加工过程最终产生之前所讨论的晶体管、导体和绝缘体。如今的集成电路只包含一层晶体管，但是可能具有多个绝缘层间隔的 2 ~ 8 层金属导体。

- ② 硅锭：一块由硅晶体组成的棒。直径大约在 8 ~ 12 英寸，长度约 12 ~ 24 英寸。
- ③ 晶圆：厚度不超过 0.1 英寸的硅锭片，用来制造芯片。

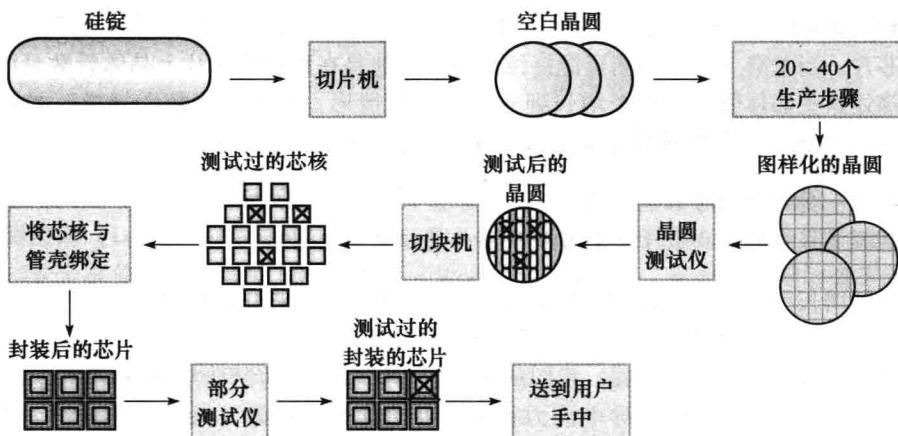


图 1-12 芯片制造的全过程。从硅锭切下来之后，空白的晶圆经过大约 20 ~ 40 步的加工，产生图样化的晶圆（见图 1-13）。这些图样化的晶圆由晶圆测试器进行测试，测试后生成一张图，表明哪些部分是合格的。之后，这些晶圆被进一步切成芯片（见图 1-9）。在本图中，一个晶圆能生产 20 个芯片，其中有 17 个通过测试（X 意味着这个芯片是坏的）。本例中芯片的良率（又称成品率）是 17/20，也就是 85%。这些合格芯片被封装起来并且在发布给用户之前经过多次测试。不合格的封装会在最终测试中被发现。

晶圆中或是在图样化的几十个步骤中出现一个细微的瑕疵就会使其附近的电路损坏，这些瑕疵（defect）使得制成一个完美的晶圆几乎是不可能的。有几种策略可以解决这一问题，最简单的策略是把晶圆切分成许多独立的晶圆，也就是现在所称的芯片（die），更正式的叫法是 chip。图 1-13 所示就是切分前的微处理器晶圆，而图 1-9 则是单个微处理器芯片。

26

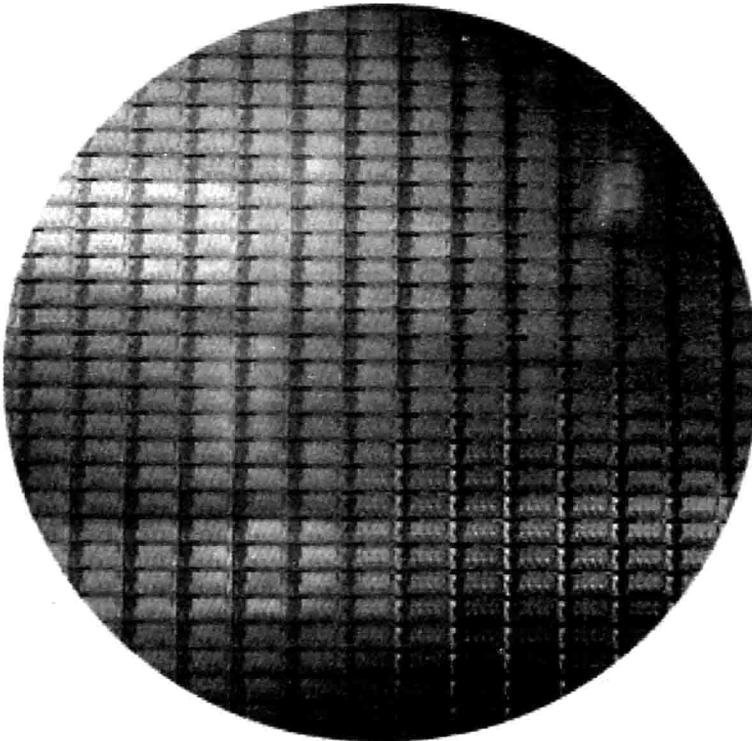


图 1-13 Intel Core i7 芯片的 12 英寸（300mm）晶圆（Intel 提供）。良率为 100% 的圆片中的晶圆的数目是 280，每个为 $20.7\text{mm} \times 10.5\text{mm}$ 。晶圆边缘几十个不完整的芯片是没用的。之所以包含它们，是因为这样给硅片生产掩膜相当容易。芯片使用 32nm 的工艺，这意味着最小的晶体管的尺寸几乎接近 32nm，尽管它们通常比实际的特征尺寸还要小，这个特征尺寸是将晶体管“图纸尺寸”和最终的生产尺寸相比得出的

通过切分，可以只淘汰那些有瑕疵的芯片，而不必淘汰整个晶圆。对这一过程的量化描述可以用成品率（yield）来表示，其定义为合格芯片数占总芯片数的百分比。

- ② 瑕疵：晶圆上一个微小的缺陷，或者在图样化的过程中因为包含这个缺陷而导致芯片失效。
- ③ 芯片：从晶圆中切割出来的一个单独的矩形区域，更加正式的英文名称是 chip。
- ④ 成品率：合格芯片数占总芯片数的百分比。

当芯片尺寸增大时，集成电路的价格会快速上升，因为成品率和晶圆中芯片的总数都下降了。为了降低价格，大芯片常采用下一代工艺进行尺寸收缩（包括晶体管和导线），从而改进每晶圆的芯片数和成品率。2012 年的典型工艺尺寸为 32nm，这意味着芯片上的最小特征尺寸是 32nm。

合格芯片要连接到 I/O 引脚上，这一过程称为封装。在封装之后，必须进行最后一次测试，因为封装过程也可能出错。最后芯片被交付给用户。

27

01 精解 集成电路的成本可以用下面 3 个简单公式来表示：

每芯片的价格 = 每晶圆的价格 / (每晶圆的芯片数 × 成品率)

每晶圆的芯片数 ≈ 晶圆面积 / 芯片面积

成品率 = $1/(1 + (\text{单位面积的瑕疵数} \times \text{芯片面积}/2))^2$

第1个公式是直接导出的。第2个公式是近似的，因为没有减去晶圆边上不满足芯片矩形要求的面积（参见图1-13）。第3个公式是基于集成电路工厂的成品率经验，与重要加工步骤的数量呈指数关系。

因此，芯片的成本取决于成品率以及芯片和晶圆的面积，与芯片面积之间的关系一般不是线性的。

01 小测验

产量是决定集成电路价格的关键因素。下列哪些理由说明了芯片产量越高成本就越低？

1. 高产量使得在制造过程中能够适当调节设计，从而提高成品率。
2. 设计高产量芯片的工作量比设计低产量芯片小。
3. 制造芯片用的掩膜很贵，产量高时每芯片的掩膜成本就低。
4. 工程开发的成本高，并且基本与产量无关，故产量高时每芯片的开发成本较低。
5. 产量高时，通常每芯片的面积比产量低时小，因此成品率较高。

1.6 性能

对计算机的性能进行评价是富有挑战性的。由于现代软件系统的规模及其复杂性，加上硬件设计者采用了大量先进的性能改进方法，使性能评价极为困难。

28

在不同的计算机中挑选合适的产品，性能是极其重要的因素之一。精确地测量和比较不同计算机之间的性能对于购买者和设计者都很重要。销售计算机的人也需要知道这些。销售人员通常希望用户看到他们的计算机表现最好的一面，无论这一面是否能准确地反映购买者的应用需求。因此，理解怎样才能更合理地测量性能以及测定所选择的计算机的性能限制相当重要。

本节将首先介绍性能评价的不同方法，然后分别从计算机用户和设计者的角度描述性能测量的度量标准，最后还要分析这些度量标准之间有什么联系，并提出经典的处理器性能方程式，我们在全书中都要使用它进行性能分析。

1.6.1 性能的定义

当我们说一台计算机比另一台计算机具有更好的性能时意味着什么？虽然这个问题看起来很简单，但实际上却内藏玄机。我们可以先用客机问题模拟一下。图1-14表示若干典型客机的型号、载客量、航程、航速等参数。如果我们要指出表中哪架客机的性能最好，那么我们首先要对性能进行定义。如果考虑不同的性能度量，那么性能最佳的客机是不同的。我们可以看到，巡航速度最高的是Concorde，航程最远的是DC-8-50，载客量最大的是747。

飞机	载客量	航程 (英里)	航速 (英里/小时)	乘客吞吐率 (载客量 × 巡航速度)
波音777	375	4 630	610	228 750
波音747	470	4 150	610	286 700
BAC/Sud Concorde	132	4 000	1 350	178 200
Douglas DC-8-50	146	8 720	544	79 424

图1-14 若干商用飞机的载客量、航程和航速。最后一列展示的是飞机运载乘客的速度，它等于容量乘以航行速度（忽略距离、起飞和降落次数）

即使假定用速度来定义性能，这里仍然有两种可能的定义。如果你关心点对点的到达时间，那么可以将只搭载一名旅客的巡航速度最快的客机认为是性能最好的。如果你关心的是运输 450 名旅客，那么如图中最后一列所示，747 的性能是最好的。与此类似，我们可以用若干不同的方法来定义计算机性能。

如果你在两台不同的桌面计算机上运行同一个程序，那么你可以说首先完成作业的那台计算机更快。如果你运行的是一个数据中心，它有好几台服务器供很多用户投放作业，那你应该说在一天之内完成作业最多的那台计算机更快。个人计算机用户会对降低响应时间（response time）感兴趣，响应时间是指从开始一个任务到该任务完成的时间，又称为执行时间。而数据中心感兴趣的常常是吞吐率（throughput）。因此，在很多情形下，和关注吞吐率的服务器相比，我们需要对嵌入式以及台式计算机采用不同的应用程序作为测试基准和不同的性能度量标准。

- ① 响应时间：也叫执行时间（execution time），是计算机完成某任务所需的总时间，包括硬盘访问、内存访问、I/O 活动、操作系统开销和 CPU 执行时间等。
- ② 吞吐率：也叫带宽（bandwidth），性能的另一种度量参数，表示单位时间内完成的任务数量。

01 例题·吞吐率和响应时间

下面两种改进计算机系统的方式能否增加其吞吐率或减少其响应时间，或既增加其吞吐率又减少其响应时间？

1. 将计算机中的处理器更换为更高速的型号。
2. 增加多个处理器来分别处理独立的任务，如搜索万维网。

01 答案

一般来说，降低响应时间几乎都可以增加吞吐率。因此，方式 1 同时改进了响应时间和吞吐率。方式 2 不会使任务完成得更快，只会增加其吞吐率。□

但是，当需要处理更多的任务时，系统可能需要令后续请求排队。在这种情况下，随着吞吐率的增加，可同时改进响应时间，因为这缩小了排队等待时间。所以，在实际的计算机系统中，响应时间和吞吐率往往相互影响。

在讨论计算机性能时，本书前几章将主要考虑响应时间方面。为了使性能最大化，我们希望任务的响应时间或执行时间最小化。对于某个计算机 X，我们可以表达为：

$$\text{性能}_x = 1 / \text{执行时间}_x$$

如果有两台计算机 X 和 Y，X 比 Y 性能更好，则

$$\begin{aligned} \text{性能}_x &> \text{性能}_y \\ 1 / \text{执行时间}_x &> 1 / \text{执行时间}_y \\ \text{执行时间}_y &> \text{执行时间}_x \end{aligned}$$

也就是说，如果 X 比 Y 快，那么 Y 的执行时间比 X 长。

在讨论计算机设计时，经常要定量地比较两台不同计算机的性能。我们将使用“X 是 Y 的 n 倍快”的表态方式，即

$$\text{性能}_x / \text{性能}_y = n$$

如果 X 比 Y 快 n 倍，那么在 Y 上的执行时间是在 X 上执行时间的 n 倍，即

$$\text{性能}_X / \text{性能}_Y = \text{执行时间}_Y / \text{执行时间}_X = n$$

01 例题·相对性能

如果计算机 A 运行一个程序只需要 10 秒，而计算机 B 运行同样的程序需要 15 秒，那么计

算机 A 比计算机 B 快多少？

01 答案

我们知道，A 是 B 的 n 倍快，则

$$\text{性能}_A / \text{性能}_B = \text{执行时间}_B / \text{执行时间}_A = n$$

故性能比为

$$15/10 = 1.5$$

因此 A 是 B 的 1.5 倍快。□

在以上的例子中，我们可以说，计算机 B 比计算机 A 慢 1.5 倍，因为

$$\text{性能}_A / \text{性能}_B = 1.5$$

意味着

$$\text{性能}_A / 1.5 = \text{性能}_B$$

简单地说，当我们试图将计算机的比较结果量化时，我们通常使用术语“比什么快”。因为性能和执行时间是倒数关系，提高性能就需要减少执行时间。为了避免对术语“增加”和“降低”潜在的误解，当我们想说“改善性能”和“改善执行时间”的时候，我们通常说“增加性能”或者“降低执行时间”。

1.6.2 性能的度量

如果用时间来度量计算机的性能，那么完成同样的计算任务，需要时间最少的计算机是最快的。程序的执行时间一般以秒为单位。然而，时间可以根据我们的计量方法选用不同的表示方法。对时间最直接的定义是墙上时钟时间（wall clock time），也叫响应时间（response time）、消逝时间（elapsed time）等。这些术语均表示完成任务所需的总时间，包括了硬盘访问、内存访问、I/O 操作和操作系统开销等一切时间。

多用户经常共享同一计算机，一个处理器需要同时运行几个程序。在这种情况下，系统可能更侧重于优化吞吐率，而不是最小化一个程序的响应时间。因此，我们往往要把运行我们自己的任务的时间与一般的响应时间区别开来。我们可以使用 CPU 执行时间（CPU execution time），简称 CPU 时间，它只表示在 CPU 上花费的时间，而不包括等待 I/O 或运行其他程序的时间。（需要注意的是，用户所感受到的是程序的响应时间，而不是 CPU 时间。）CPU 时间还可进一步分为用于用户程序的时间和操作系统为用户服务花去的 CPU 时间。前者称为用户 CPU 时间（user CPU time），后者称为系统 CPU 时间（system CPU time）。要精确区分这两种 CPU 时间是困难的，因为通常难以分清哪些操作系统的活动是属于哪个用户程序的，而且不同操作系统的功能也千差万别。

- CPU 执行时间：简称 CPU 时间，执行某一任务在 CPU 上所花费的时间。
- 用户 CPU 时间：在程序本身所花费的 CPU 时间。
- 系统 CPU 时间：为执行程序而花费在操作系统上的时间。

为了一致性，我们保持基于响应时间和基于 CPU 执行时间的性能差异。我们使用术语系统性能（system performance）表示空载系统的响应时间，并用术语 CPU 性能（CPU performance）表示用户 CPU 时间。本章我们概括介绍了计算机性能，既适用于响应时间的度量，也适用于 CPU 时间的度量，但本章的重点将放在 CPU 性能上。

01 理解程序性能 不同的应用关注计算机系统性能的不同方面。许多应用，特别是那些运行

在服务器上的应用，主要关注 I/O 性能，所以此类应用既依赖硬件又依赖软件，对墙上时钟时间最感兴趣。而在其他一些应用中，用户可能对吞吐率、响应时间或两者的复杂组合更为关注（例如，最差响应时间下的最大吞吐率）。要改进一个程序的性能，必须明确性能的定义，然后通过测量程序执行时间来寻找可能的性能瓶颈。在后面的章节中，我们将介绍如何在系统的各个部分寻找瓶颈，以改进性能。

虽然作为计算机用户我们关心的是时间，但当我们深入研究计算机的细节时，使用其他的度量可能更为方便。对计算机设计者来说，他们需要考虑如何度量计算机硬件完成基本功能的速度。几乎所有计算机都用时钟来驱动硬件中发生的各种事件。时钟间隔的时间称为时钟周期 (clock cycle)。也可用它的倒数来描述，称为时钟频率 (clock rate)。例如，时钟周期为 250ps，对应的时钟频率为 4GHz。在下一节，我们将形式化地定义硬件设计者的时钟周期和计算机使用者所指的秒之间的关系。

- ⌚ 时钟周期：也叫 tick、clock tick、clock period、clock 或 cycle，为计算机一个时钟周期的时间，通常是处理器时钟，一般为常数。
- ⌚ 时钟长度：每个时钟周期持续的时间长度。

01 小测验

1. 假设某个使用个人移动设备和云的应用受网络性能限制。那么对于下列 3 种方法，哪种只改进了吞吐率？哪种同时改进了响应时间和吞吐率？哪种都没有改进？
 - a. 在个人移动设备和云之间增加一条额外的网络信道，从而增加总的网络吞吐率，并减少获得网络访问的延迟（现在已经存在 2 条网络信道）。
 - b. 改进网络软件，从而减少网络通信延迟，但并不增加吞吐率。
 - c. 增加计算机的内存。
2. 计算机 B 运行给定的应用需要 28 秒，而计算机 C 的性能是计算机 B 的 4 倍。请问计算机 C 运行同样的应用需要多长时间？

1.6.3 CPU 性能及其因素

用户和设计者往往以不同的尺度看待性能。如果我们能掌握这些不同尺度之间的关系，就能确定一个设计的变化对性能的影响。由于我们都关注 CPU 性能，因而性能度量实际上针对的是 CPU 执行时间。下面一个简单的公式把最基本的尺度（时钟周期数和时钟周期时间）和 CPU 时间联系起来：

一个程序的 CPU 执行时间 = 一个程序的 CPU 时钟周期数 × 时钟周期时间
由于时钟频率和时钟周期时间互为倒数，故

一个程序的 CPU 执行时间 = 一个程序的 CPU 时钟周期数 / 时钟频率

这个公式清楚地表明，硬件设计者减少一个程序的 CPU 时钟周期数，或减少时钟周期时间，就能改进性能。在后面几章中我们将看到，设计者经常要面对这些因素之间的权衡。许多技术在减少时钟周期数的同时也会引起时钟周期时间的增加。

01 例题·性能的改进

某程序在一台时钟频率为 2GHz 的计算机 A 上运行需要 10 秒。现在将设计一台计算机 B，希望将运行时间缩短为 6 秒。计算机的设计者采用的方法是提高时钟频率，但这会影响 CPU 其余部分的设计，使计算机 B 运行该程序时需要相当于计算机 A 的 1.2 倍的时钟周期数。那么计

算机设计者应该将时钟频率提高到多少?

01 答案

我们首先要知道在 A 上运行该程序需要多少时钟周期数:

$$\text{CPU 时间}_A = \text{CPU 时钟周期数}_A / \text{时钟频率}_A$$

$$10 \text{ 秒} = \text{CPU 时钟周期数}_A / 2 \times 10^9 \text{ 周期数/秒}$$

$$\text{CPU 时钟周期数}_A = 10 \text{ 秒} \times 2 \times 10^9 \text{ 周期数/秒} = 20 \times 10^9 \text{ 周期数}$$

B 的 CPU 时间公式为:

$$\text{CPU 时间}_B = 1.2 \times \text{CPU 时钟周期数}_A / \text{时钟频率}_B$$

$$6 \text{ 秒} = 1.2 \times 20 \times 10^9 \text{ 时钟周期数/时钟频率}_B$$

$$\text{时钟频率}_B = 1.2 \times 20 \times 10^9 \text{ 时钟周期数/6 秒} = 0.2 \times 20 \times 10^9 \text{ 时钟周期数/秒}$$

$$= 4 \times 10^9 \text{ 时钟周期数/秒} = 4 \text{ GHz}$$

因此,要在 6 秒内运行完该程序, B 的时钟频率必须提高为 A 的 2 倍。 □

1.6.4 指令的性能

上述性能公式没有涉及程序所需的指令数。然而,由于计算机是通过执行指令来运行程序的,因此执行时间一定依赖于程序中的指令数。一种考虑执行时间的方法是,执行时间等于执行的指令数乘以每条指令的平均时间。所以,一个程序需要的时钟周期数可写为:

$$\text{CPU 时钟周期数} = \text{程序的指令数} \times \text{每条指令的平均时钟周期数}$$

术语 CPI (clock cycle per instruction) 表示执行每条指令所需的时钟周期数的平均值。不同的指令需要的时间可能不同,CPI 是一个程序全部指令所用时钟周期数的平均值。CPI 提供了比较相同指令集的不同实现方式的方法,因为一个程序执行的指令数是一样的。

- ② CPI: 每条指令的时钟周期数,表示执行某个程序或者程序片段时每条指令所需的时钟周期平均数。

01 例题·性能公式的使用

假设我们有相同指令集的两种不同实现方式。计算机 A 的时钟周期为 250ps,对某程序的 CPI 为 2.0;计算机 B 的时钟周期为 500ps,对同样程序的 CPI 为 1.2。对于该程序,请问哪台计算机执行的速度更快?快多少?

01 答案

我们知道,对于固定的程序,每台计算机执行的总指令数是相同的,我们用 I 来表示。首先,求每台计算机的 CPU 时钟周期数:

$$\text{CPU 时钟周期数}_A = I \times 2.0$$

$$\text{CPU 时钟周期数}_B = I \times 1.2$$

现在,可以计算每台计算机的 CPU 时间:

$$\text{CPU 时间}_A = \text{CPU 时钟周期数}_A \times \text{时钟周期时间} = I \times 2.0 \times 250\text{ps} = 500 \times I\text{ps}$$

同理

$$\text{CPU 时间}_B = I \times 1.2 \times 500\text{ps} = 600 \times I\text{ps}$$

显然,计算机 A 更快。快多少由执行时间之比来计算

$$\text{CPU 性能}_A / \text{CPU 性能}_B = \text{执行时间}_B / \text{执行时间}_A = 600 \times I\text{ps} / 500 \times I\text{ps} = 1.2$$

因此,对于该程序计算机 A 是计算机 B 的 1.2 倍快。 □

1.6.5 经典的 CPU 性能公式

现在我们可以用指令数 (instruction count)、CPI 和时钟周期时间来写出基本的性能公式：

$$\text{CPU 时间} = \text{指令数} \times \text{CPI} \times \text{时钟周期时间}$$

或

$$\text{CPU 时间} = \text{指令数} \times \text{CPI} / \text{时钟频率}$$

这些公式特别有用，因为它们把性能分解为三个关键因素。如果知道实现方案或替代方案如何影响这三个参数，我们可用这些公式来比较不同的实现方案或评估某个设计的替代方案。

- 指令数：执行某程序所需的总指令数量。

36

01 例题·代码段的比较

一个编译器设计者试图在两个代码序列之间进行选择。硬件设计者给出了如下数据：

	每类指令的 CPI		
	A	B	C
CPI	1	2	3

对于某行高级语言语句的实现，两个代码序列所需的指令数量如下：

代码序列	每类指令的数量		
	A	B	C
1	2	1	2
2	4	1	1

哪个代码序列执行的指令数更多？哪个执行速度更快？每个代码序列的 CPI 是多少？

01 答案

代码序列 1 共执行 $2 + 1 + 2 = 5$ 条指令。代码序列 2 共执行 $4 + 1 + 1 = 6$ 条指令。所以，代码序列 1 执行的指令数更少。

基于指令数和 CPI，我们可以用 CPU 时钟周期公式计算出每个代码序列的总时钟周期数为：

$$\text{CPU 时钟周期数} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

因此

$$\text{CPU 时钟周期数}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ 周期}$$

$$\text{CPU 时钟周期数}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ 周期}$$

故代码序列 2 更快，尽管它多执行了一条指令。由于代码序列 2 总时钟周期数较少，而指令数较多，因此它一定具有较小的 CPI。CPI 的计算公式为：

$$\text{CPI} = \text{CPU 时钟周期数} / \text{指令数}$$

$$\text{CPI}_1 = \text{CPU 时钟周期数}_1 / \text{指令数}_1 = 10/5 = 2$$

$$\text{CPI}_2 = \text{CPU 时钟周期数}_2 / \text{指令数}_2 = 9/6 = 1.5$$

□ 37

01 重点 图 1-15 给出了计算机在不同层次上的性能测试指标及其测试单位。通过这些指标的组合可以计算出程序的执行时间（单位为秒）：

$$\text{执行时间} = \text{秒}/\text{程序} = \text{指令数}/\text{程序} \times \text{时钟周期数}/\text{指令} \times \text{秒}/\text{时钟周期}$$

永远记住，唯一能够被完全可靠测量的计算机性能指标是时间。例如，对指令集减少指令数目的改进可能降低时钟周期时间或提高 CPI，从而抵消了改进的效果。类似地，CPI 与执行的指令类型相关，执行指令数最少的代码其执行速度未必是最快的。

性能的分量	测量单位
程序的CPU执行时间	程序执行的执行时间，以秒为单位
指令数目	程序执行的指令数目
指令的平均执行时钟周期 (CPI)	每条指令的平均执行的时钟周期数
时钟周期时间	每个时钟周期的长度，以秒为单位

图 1-15 基本的性能指标及其测量单位

如何确定性能公式中这些因素的值呢？我们可以通过运行程序来测量 CPU 的执行时间，并且计算机的说明书中通常介绍了时钟周期时间。难以测量的是指令数和 CPI。当然，如果确定了时钟频率和 CPU 执行时间，我们只需要知道指令数或者 CPI 两者之一，就可以依据性能公式计算出另一个。

可以通过用体系结构仿真器等软件工具预执行程序来测量出指令数，也可以用大多数处理器中的硬件计数器来测量执行的指令数、平均 CPI 和性能损失源等。由于指令数量取决于计算机体系结构，并不依赖于计算机的具体实现，因而我们可以在不知道计算机全部实现细节的情况下对指令数进行测量。但是，CPI 与计算机的各种设计细节密切相关，包括存储系统和处理器结构（我们将在第 4、5 章中看到），以及应用程序中不同类型的指令所占的比例。因此，CPI 对于不同应用程序是不同的，对于相同指令集的不同实现方式也是不同的。

上述例子表明，只用一种因素（如指令数）去评价性能是危险的。当比较两台计算机时，必须考虑全部三个因素，它们组合起来才能确定执行时间。如果某个因素相同（如上例中的时钟频率），则必须考虑不同的因素才能确定性能的优劣。因为 CPI 随着指令组合 (instruction mix) 而变化，所以必须比较指令的条数和 CPU，即使时钟频率是相同的。在本章最后的练习题中，有几道是关于计算机和编译程序改进后对时钟频率、CPI 和指令数目影响的评价。在 1.10 节，我们将讨论一种因没有全面考虑各种因素而导致的对性能的误解。

● 指令组合：在一个或多个程序中，指令的动态使用频度的评价指标。

01 理解程序性能 程序的性能与算法、编程语言、编译程序、体系结构以及实际的硬件有关。下表概括了这些成分是如何影响 CPU 性能公式中的各种因素的。

硬件或软件指标	影响什么	如何影响
算法	指令数，可能的 CPI	算法决定源程序执行指令的数目，从而也决定了 CPU 执行指令的数目。算法也可能通过使用较快或较慢的指令影响 CPI。例如，当算法使用更多的除法运算时，将会导致 CPI 增大
编程语言	指令数，CPI	编程语言显然会影响指令数，因为编程语言中的语句必须翻译为指令，从而决定了指令数。编程语言也可影响 CPI，例如，Java 语言充分支持数据抽象，因此将进行间接调用，需要使用较高的 CPI 指令
编译程序	指令数，CPI	因为编译程序决定了源程序到计算机指令的翻译过程，所以编译程序的效率既影响指令数又影响 CPI。编译器会以复杂的方式影响 CPI
指令集体系结构	指令数，CPI 时钟频率	指令集体系结构影响 CPU 性能的所有 3 个方面，因为它影响完成某功能所需的指令数、每条指令的周期数以及处理器的时钟频率

01 精解 也许你期望 CPI 最小值为 1.0。在第 4 章我们将看到，有些处理器在每个时钟周期可对多条指令取指并执行。有些设计者用 IPC (instruction per clock cycle) 来代替 CPI。如一个处理器每时钟周期可执行平均 2 条指令，则它的 $IPC = 2$, $CPI = 0.5$ 。

01 精解 虽然时钟周期时间传统上是固定的，但是为了节省能量或暂时提升性能，当今的计算机可以使用不同的时钟频率，因此我们需要对程序使用平均时钟频率。例如，Intel Core i7 处理器在处理器温度升高之前可以暂时将时钟频率提高 10%。Intel 称之为快速模式 (Turbo mode)。

01 小测验

某 Java 程序在桌面处理器上运行需时 15 秒。一个新版本的 Java 编译程序发行了，其编译产生的指令数量是旧版本 Java 编译程序的 0.6 倍，不幸的是，CPI 增加为原来的 1.1 倍。请问该程序在新版本的 Java 编译程序中运行速度是多少？从以下三个选项中选出正确答案。

- a. $15 \times 0.6 / 1.1 = 8.2$ 秒
- b. $15 \times 0.6 \times 1.1 = 9.9$ 秒
- c. $15 \times 1.1 / 0.6 = 27.5$ 秒

1.7 功耗墙

图 1-16 表示 30 年间 Intel 八代微处理器的时钟频率和功耗的增长趋势。两者增长几乎保持了几十年，但近几年来突然缓和下来。其原因在于两者是密切相关的，而且功耗已经到达了极限，无法再将处理器冷却下来。

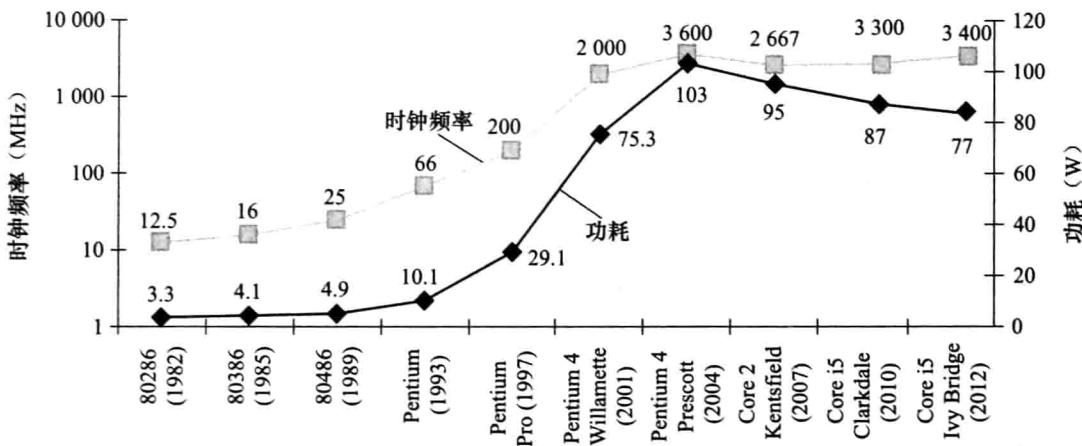


图 1-16 25 年间 Intel x86 八代微处理器的时钟频率和功耗。奔腾 4 处理器时钟频率和功耗提高很大，但是性能提升不大。Prescott 发热问题导致奔腾 4 处理器的生产线被放弃。Core 2 生产线恢复使用低时钟频率的简单流水线和片上多处理器。Core i5 采用同样的流水线

虽然功耗提供了能够冷却的极限，然而在后 PC 时代，能量是真正关键的资源。对于个人移动设备来说，电池寿命比性能更为关键。对于具有 100 000 个服务器的仓储式计算机来说，冷却费用非常高，因此设计者要尽量降低其功耗。就像在评价性能时，使用执行时间比使用 MIPS (见 1.10 节) 之类的比率更加可信一样，在评价功耗时，使用能耗比功耗更加合理，能耗的单位是焦耳/秒。

占统治地位的集成电路技术是 CMOS (互补型金属氧化半导体)，其主要的能耗来源是动态能耗，即在晶体管开关过程中产生的能耗，即晶体管的状态从 0 翻转到 1 或从 1 翻转到 0 消

耗的能量。动态能耗取决于每个晶体管的负载电容和工作电压：

$$\text{能耗} \propto \text{负载电容} \times \text{电压}^2$$

这个等式表示的是一个 $0 \rightarrow 1 \rightarrow 0$ 或 $1 \rightarrow 0 \rightarrow 1$ 的逻辑转换过程中消耗的能量。一个晶体管消耗的能量为：

$$\text{能耗} \propto 1/2 \times \text{负载电容} \times \text{电压}^2$$

每个晶体管需要的功耗是一个翻转需要的能耗和开关频率的乘积：

$$\text{功耗} \propto 1/2 \times \text{负载电容} \times \text{电压}^2 \times \text{开关频率}$$

开关频率是时钟频率的函数，负载电容是连接到输出上的晶体管数量（称为扇出）和工艺的函数，该函数决定了导线和晶体管的电容。

思考一下图 1-16 的趋势，为什么时钟频率增长为 1 000 倍，而功耗只增长为 30 倍呢？因为能耗和功耗是电压平方的函数，能够通过降低电压来大幅减少，每次工艺更新换代时都会这样做。一般来说，每代的电压降低大约 15%。20 多年来，电压从 5V 降到了 1V。这就是功耗只增长 30 倍的原因所在。

01 例题·相对功耗

假设我们需要开发一种新处理器，其负载电容只有旧处理器的 85%。再假设其电压可以调节，与旧处理器相比电压降低了 15%，进而导致频率也降低了 15%，问这对新处理器的动态功耗有何影响？

01 答案

$$\frac{P_{\text{新}}}{P_{\text{旧}}} = (\text{电容负载} \times 0.85) \times (\text{电压} \times 0.85)^2 \times \frac{(\text{开关频率} \times 0.85)}{(\text{电容负载} \times \text{电压}^2 \times \text{开关频率})}$$

因此功耗比为

$$0.85^4 = 0.52$$

新处理器的功耗大约为旧处理器的一半。 □

目前的问题是如果电压继续下降会使晶体管泄漏电流过大，就像水龙头不能被完全关闭一样。目前 40% 的功耗是由于泄漏造成的，如果晶体管的泄漏电流再大，情况将会变得无法收拾。

为了解决功耗问题，设计者连接大设备以增加冷却，而且将芯片中的一些在给定时钟周期内暂时不用的部分关闭。尽管有很多更加昂贵的方式来冷却芯片，但继续提高芯片的功耗（比如到 300 瓦）对个人计算机甚至服务器来说成本太高了，对个人移动设备就更不用说了。

由于计算机设计者遇到了功耗墙问题，因此他们需要开辟新的路径，选择不同于已经用了 30 多年的方法继续前进。

01 精解 虽然动态能耗是 CMOS 能耗的主要来源，但静态能耗也是存在的，因为即使在晶体管关闭的情况下，还是有泄漏电流存在。在服务器中，典型的电流泄漏占 40% 的能耗。因此，增加晶体管的数目，就会增加漏电功耗，即使这些晶体管总是关闭的。人们采用各种各样的设计和工艺创新来控制电流泄漏，但还是难以进一步降低电压。

01 精解 功耗成为集成电路设计的一个挑战有两个原因。首先，电源必须由外部输入并且分布到芯片的各个角落。现代微处理器通常使用几百个管脚作为电源和地！同样，多层次芯片互联仅仅为了解决芯片的电源和地的分布比例问题。其次，功耗作为热量形式散发，因此必须进行散热处理。服务器芯片的功耗可高达 100 瓦以上，因此芯片及外围系统的散热是仓储规模计算机的主要开销（见第 6 章）。

1.8 沧海巨变：从单处理器向多处理器转变

迄今为止，很多软件很像独唱者所写的音乐；使用当代的芯片，我们对于编写二重唱、四重唱以及小型合奏的经验很少，但是为大型交响乐或者合唱谱曲则是一个不同的挑战。

——Brian Hayes, 《Computing in a Parallel Universe》, 2007

功耗的极限迫使微处理器的设计产生了巨变。图 1-17 给出了桌面微处理器的程序响应时间的发展。从 2002 年起，其每年的增长速率从 1.5 下降到 1.2。

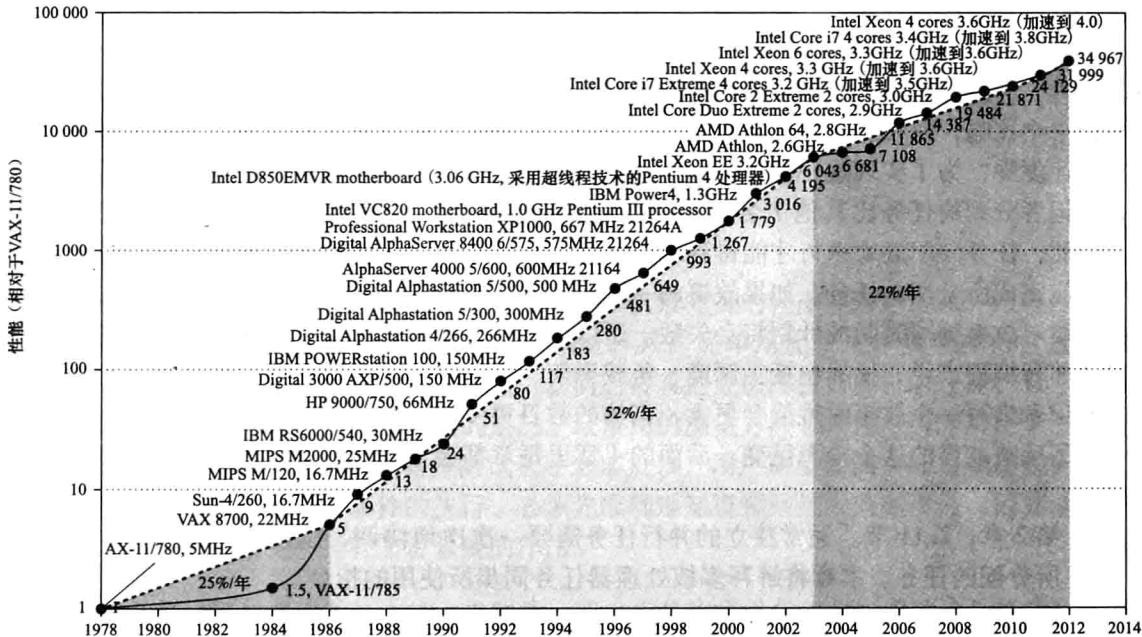


图 1-17 自 20 世纪 80 年代中期以来处理器性能的发展。本图描绘了和 VAX 11/780 相比，采用 SPECint 测试程序得到的性能数据（见 1.10 节）。在 20 世纪 80 年代中期以前，性能的增长主要靠技术驱动，平均每年增长 25%。在这个阶段之后，增长速度达到 52%，这归功于体系结构和组织方式的创新。从 20 世纪 80 年代中期开始，性能每年大约提高 52%，如果按照原先的 25% 的增长率计算，则到 2002 年的性能只有实际的 1/7。从 2002 年开始，受到功耗、指令级并行程度和存储器长延迟时间的限制，单核处理器的性能增长放缓，大约每年 22%

在 2006 年，所有桌面和服务器公司都在单片微处理器中加入了多个处理器，以求更大的吞吐率，而不再继续追求降低单个程序运行在单个处理器上的响应时间。为了减少 processor 和 microprocessor（微处理器）这两个词语之间的混淆，一些公司将 processor 作为“cores”的代称，这样 microprocessor 就是多核处理器了。因此，一个“四核”微处理器是一个包含了 4 个 processor 或者 4 个 core 的芯片。

在过去，程序员可以依赖于硬件、体系结构和编译程序的创新，无需修改一行代码，程序的性能每 18 个月翻一番。而今天，程序员要想显著改进响应时间，必须重写他们的程序以充分利用多处理器的优势。而且，随着核的数目不断加倍，程序员也必须不断改进他们的代码，以便在新微处理器上获得显著的性能提升。

为了强调软件和硬件系统的协同工作，我们在本书用“硬件/软件接口”的概念来进行描述，并对这一接口概括一些重要的观点，下面是本书中的第一个。

11 硬件/软件接口 并行性对计算性能一直十分重要，但它往往是隐蔽的。第 4 章将说明流水线，

它是一种漂亮的技术，通过指令重叠执行使程序运行得更快。这是指令级并行性的一个例子。在抽取了硬件的并行本质之后，程序员或编译程序可认为在硬件中指令是串行执行的。

迫使程序员意识到硬件的并行性，并显式地按并行方式重写其程序，曾经是计算机体系结构的“第三抱怨”，以致很多采用此种方式进行革新的公司都失败了（见6.15节）。从历史发展的角度来观察，整个IT行业已经把它放到了未来的发展方向上，程序员最终将成功地跃进到显式并行编程。

43

为什么程序员编写显式并行程序如此困难呢？第一个原因是并行编程以提高性能为目的，必然增加编程的难度。不仅程序必须要正确，能够解决重要问题，而且运行速度要快，还需要为用户或其他程序提供接口以便使用，否则编写一个串行程序就足够了。

第二个原因是为发挥并行硬件的速度，程序员必须将应用划分为每个核上有大致相同数量的任务，并同时完成。还要尽可能减小调度的开销，以不至于浪费并行性能。

44

作一个比喻，现在有一个写新闻故事的任务，如果由8名记者共同来完成，能否提高8倍的写作速度呢？为了实现这一目标，这个新闻故事需要进行划分，让每个记者都有事可做。假如某名记者分到的任务比其他7名记者加起来的任务还要多，那用8名记者的好处就不存在了。因此，任务分配必须平衡才能得到理想的加速。另一个存在的危险是记者要花费时间互相交流才能完成所分配的任务。如果故事的一部分，例如结论，在所有其他部分完成之前不能编写，则缩短故事编写时间的计划将会失败。所以，必须尽量减少通信和同步的开销。对于上述比喻和并行编程来说，挑战包括：调度、负载平衡、通信以及同步等开销。你也许会想到，当更多的记者来写一个故事时挑战会更大，当核的数目更多时，并行编程的挑战将更大。

为了反映业界的这个沧海巨变，后面的4章里每章都会至少有一节介绍有关并行性革命的内容：

- 第2章，2.11节。通常独立的并行任务需要一次次地协调，以便通报它们何时完成了所分配的任务。本章将解释多核处理器任务同步所使用的指令。
- 第3章，3.6节。并行性的最简单方式是将计算设备单元并行工作，例如两个向量相乘。**摩尔定律**提供了位宽更大且能同时处理多个操作数的算术单元，字并行就是利用这种资源的并行性的。
- 第4章，4.10节。尽管明确地知道并行编程的困难，但在20世纪90年代，人们依然付出了巨大的努力和投资用于从流水线开始研究硬件和编译程序的并行性。本章描述了这些技术，包括取指与多指令同时执行和通过预测的方式推测决策结果、指令执行等。
- 第5章，5.10节。降低通信开销的一个方法是让所有处理器使用同一个地址空间，任何处理器可以读写任何数据。今天的计算机都采用cache技术，即在处理器附近更快的存储器中，保持数据的一个临时复本。可以想象，如果多个处理器访问cache中的共享数据不一致的话，并行编程将尤为困难。本章将介绍保持所有cache数据一致性的机制。
- 第5章，5.11节。本节介绍如何使用许多磁盘共同构成一个能够提供更高吞吐率的系统，这就是廉价冗余磁盘阵列（RAID）的灵感。RAID流行的真正原因是它能够通过采用适当数量的冗余磁盘提供更高的可靠性。本节将介绍在不同RAID级别的性能、成本和可靠性。

45

除了这些章节之外，还有一整章介绍并行编程。第6章详细叙述了并行编程的挑战性，提出了两种方法来解决共享编址通信和显式消息传输，介绍了一种易于编程的并行性模型，讨论了使用基准测试程序对并行处理器进行评测的困难，为多核微处理器引入了一个新的简单性能模型，最后描述和评价了4种使用该种模型的多核微处理器。

如上所述，第3~6章使用矩阵向量相乘作为采用并行性提高性能的例子。

附录 C 介绍了一种在桌面计算机中越来越普及的图形处理器（Graphics Processing Unit, GPU）。它是为加速图像处理而发明的。得益于高度的并行性，GPU 表现出了优越的性能，并已发展为完善的编程平台。

附录 C 介绍了 NVIDIA GPU 及其并行编程环境。

1.9 实例：Intel Core i7 基准

我想，就像书一样，“计算机”是一个全世界广泛应用的概念。但我没有想到它会发展得如此迅速，因为我完全没有预料到我们在一块芯片上可以得到像我们最终得到的如此多的部件。晶体管的进步完全出乎我们的预料。它比我们预想的发展要快。

——J. Presper Eckert, ENIAC 的创建者之一, 1991

本书的每一章都有“实例”一节，它将本书中的概念与我们日常使用的计算机联系起来，这些小节涵盖了现代计算机中使用的技术。下面是本书中的第一个“实例”小节，我们将以 Intel Core i7 为例，说明如何制造集成电路，以及如何测量性能和功耗。

1.9.1 SPEC CPU 基准测试程序

用户日复一日使用的程序是用于评价新型计算机最完美的程序。所运行的一组程序集构成了工作负载（workload）。要评价两台计算机系统，只需简单地比较工作负载在两台计算机上的执行时间。然而大多数用户并不这样做，他们通过其他方法测量计算机的性能，希望这些方法能够反映计算机执行用户工作负载的情况。最常用的测量方法是使用一组专门用于测量性能的基准测试程序（benchmark）。这些测试程序形成负载，用户期望预测实际负载的性能。我们在前面提到，要加速大概率事件的执行，必须先准确地知道哪些是大概率事件，因此基准测试程序在计算机系统结构中具有非常重要的作用。

- ① 工作负载：运行在计算机上的一组程序，可以直接使用用户的一组实际应用程序，也可以从实际程序中构建。一个典型的工作负载必须指明程序和相应的频率。
- ② 基准测试程序：用于比较计算机性能的程序。

SPEC (system performance evaluation cooperative) 是由许多计算机销售商共同出资赞助并支持的合作组织，目的是为现代计算机系统建立基准测试程序集。1989 年，SPEC 建立了重点面向处理器性能的基准程序集（现在称为 SPEC89）。历经 5 代发展，目前最新的是 SPEC CPU 2006，它包括 12 个整数基准程序集（CINT 2006）和 17 个浮点基准程序集（CFP 2006）。CINT 2006 包括 C 编译程序、量子计算机仿真、下象棋程序等，CFP 2006 包括有限元模型结构化网格法、分子动力学质点法、流体动力学稀疏线性代数法等。

图 1-18 列举了 SPEC 整数基准程序及其在 Intel Core i7 上的执行时间、指令数、CPI 和时钟周期时间等组成的 SPEC 分值。注意，CPI 的最大值和最小值相差达到 5 倍。

为了简化测试结果，SPEC 决定使用单一的数字来归纳所有 12 种整数基准程序。具体方法是将被测计算机的执行时间标准化，即将被测计算机的执行时间除以一个参考处理器的执行时间，结果称为 SPECratio。SPECratio 值越大，表示性能越快（因为 SPECratio 是执行时间的倒数）。CINT2006 或 CFP2006 的综合测试结果是取 SPECratio 的几何平均值。

01 精解 在使用 SPECratio 比较两台计算机时采用的是几何平均值，这样可以使得无论采用哪台计算机进行标准化都可得到同样的相对值。如果采用的是算术平均值，结果会随选用的参考计算机而变。

描述	名称	指令数目 ($\times 10^9$)	CPI	时钟周期时间 ($\times 10^{-9}$ 秒)	执行时间 (秒)	参考时间 (秒)	Spec分值
字符串处理解释程序	perl	2 252	0.60	0.376	508	9 770	19.2
块排序压缩	bzip2	2 390	0.70	0.376	629	9 650	15.4
GNU C编译器	gcc	794	1.20	0.376	358	8 050	22.5
组合优化	mcf	221	2.66	0.376	221	9 120	41.2
go游戏(人工智能)	go	1 274	1.10	0.376	527	10 490	19.9
围棋游戏(人工智能)	hmmer	2 616	0.60	0.376	590	9 330	15.8
基因序列搜索	sjeng	1 948	0.80	0.376	586	12 100	20.7
量子计算机仿真	libquantum	659	0.44	0.376	109	20 720	190.0
视频压缩	h264avc	3 793	0.50	0.376	713	22 130	31.0
离散事件仿真库	omnetpp	367	2.10	0.376	290	6 250	21.5
游戏/寻找路径	astar	1 250	1.00	0.376	470	7 020	14.9
XML解析	xalancbmk	1 045	0.70	0.376	275	6 900	25.1
几何平均	-	-	-	-	-	-	25.7

图 1-18 SPECINTC2006 基准程序在 2.66GHz 的 Intel Core i7 920 上的运行结果。按照经典的 CPU 性能公式一节(原书第 36 页)的等式, 执行时间是本表的三个因素的乘积: 以亿为单位的指令数、每条指令的时钟数(CPI)以及纳秒级的时钟周期时间。SPECratio 仅仅是参考时间, 由 SPEC 提供, 被所测量的执行时间相除。SPECINTC2006 所引用的单个数目是 SPECratio 的几何平均数

几何平均值的公式是

$$\sqrt[n]{\prod_{i=1}^n \text{执行时间比}_i}$$

其中, 执行时间比_i是总共 n 个工作负载中第 i 个程序的执行时间按参照计算机进行标准化的结果, 并且

$$\prod_{i=1}^n a_i \text{ 表示 } a_1 \times a_2 \times \cdots \times a_n$$

1.9.2 SPEC 功耗基准测试程序

由于能耗和功耗日益重要, SPEC 增加了一组用于评估功耗的基准测试程序, 它可以报告一段时间内服务器在不同负载水平下(以 10% 的比例递增)的功耗。图 1-19 给出了在基于 Intel Nehalem 处理器的服务器上的测试结果, 与前面类似。

目标负载%	性能 (ssj_ops)	平均功耗 (瓦特)
100%	865 618	258
90%	786 688	242
80%	698 051	224
70%	607 826	204
60%	521 391	185
50%	436 757	170
40%	345 919	157
30%	262 071	146
20%	176 061	135
10%	86 784	121
0%	0	80
合计	4 787 166	1 922
$\sum \text{ssj_ops} / \sum \text{power} =$		2 490

图 1-19 SPECpower_ssj2008 在服务器上的运行结果。服务器的具体配置为双插槽 2.6GHz Intel Xeon X5650 处理器, 16GB DRAM, 100GB 固态硬盘

SPECpower 最早来自于面向 Java 商业应用的 SPEC 基准程序 (SPECJBB2005)，它主要测试处理器、cache、主存以及 Java 虚拟机、编译器、无用单元收集器、操作系统片段。性能采用吞吐率来测量，单位是每秒完成的操作次数。还是为了简化结果，SPEC 采用单个的数字来进行归纳，称为“overall ssj_ops per watt”，其计算公式是：

$$\text{overall ssj_ops per watt} = \left(\sum_{i=0}^{10} \text{ssj_ops}_i \right) / \left(\sum_{i=0}^{10} \text{power}_i \right)$$

式中， ssj_ops_i 为工作负载在每 10% 增量处的性能， power_i 是对应的功耗。

1.10 谬误与陷阱

科学一定开始于神话和对神话的批判。

—— Sir Karl Popper, 《The Philosophy of Science》, 1957

本书中每一章都会有“谬误与陷阱”一节，其目的是说明我们在实际中经常遇到的误解，我们称之为谬误。当讨论谬误时，我们会举出一个反例。我们也讨论陷阱，即那些容易犯的错误。通常陷阱是指一般原理只在有限的上下文中才是真的。本节旨在帮助你在设计或使用计算机时避免犯同样的错误。价格/性能谬误和陷阱使许多计算机架构师掉入圈套，包括我们下面开始介绍的本书的第一个陷阱，虽然它曾迷惑了许多设计者，却揭示了计算机设计中的一个重要关系。

陷阱：在改进计算机的某个方面时期望总性能的提高与改进大小成正比。

加速大概率事件的伟大思想导致的令人泄气的结果困扰着软件和硬件设计人员。它提醒我们一个事件需要的时间影响着改进的机会。

用一个简单的例子就可以很好地说明。假设一个程序在一台计算机上运行需要 100 秒，其中 80 秒的时间用于乘法操作。如果要把该程序的运行速度提高到 5 倍，乘法操作的速度应该改进多少？

改进以后的程序执行时间可用下面的 Amdahl 定律计算：

改进后的执行时间 = 受改进影响的执行时间 / 改进量 + 不受影响的执行时间
代入本例的数据进行计算：

$$\text{改进后的执行时间} = 80/n + (100 - 80)$$

由于要求快至 5 倍，新的执行时间应该是 20：

$$20 = 80/n + 20$$

$$0 = 80/n$$

② **Amdahl 定律：**阐述了“对于特定改进的性能提升可能由所使用的改进特征的数量所限制”的规则。它是“收益递减定律”的量化版本。

可见，如果乘法运算占总负载的 80%，则无论怎样改进乘法，也无法达到性能提高至 5 倍的结果。特定改进的性能提升由所使用的改进特征的数量所限制。这个概念也产生了日常生活中我们称为“收益递减”的定律。

当我们知道一些函数所消耗的时间及其潜在的加速时，我们就可以使用 Amdahl 定律预测性能的提升。将 Amdahl 定律与 CPU 性能公式结合，是一种很方便的性能评价工具。读者可以在本章练习中进一步体会。

Amdahl 定律还应用于并行处理器数量的实际限制中，我们将在第 6 章中的“谬误与陷阱”中介绍。

谬误：利用率低的计算机功耗低。

服务器的工作负载是变化的，所以在低利用率的情况下功率很重要。例如，Google 仓储式计算机中 CPU 利用率大多数时间在 10% ~ 50% 之间，只有不到 1% 的时间达到 100%。即使花费 5 年时间来研究如何很好地运行 SPECpower 基准测试程式，在 2012 年，根据最好的结果配置的计算机中，只有 10% 的工作负载能够消耗 1/3 的峰值功耗。在实际工作中的系统由于没有针对 SPECpower 进行配置，因此其结果将会更加糟糕。

由于服务器的工作负载差异大且消耗了峰值功耗的很大比例，Luiz Barroso 和 Urs Holzle 提出需要对硬件重新进行设计，已达到“按能量比例计算”。这就是说，在未来的服务器中，10% 的工作负载使用 10% 的峰值功耗，这将减少数据中心的电费和二氧化碳的排放。

谬误：面向性能的设计和面向能量效率的设计具有不相关的目标。

由于能耗是功耗和时间的乘积，在通常情况下，对于软硬件的优化而言，即使在优化的部分起作用时能耗可能高了一些，但是这些优化缩短了系统运行时间，因此整体上还是节约了能量。一个重要的原因是当一个程序运行时，计算机的其他部分仍在消耗能量，因此，即使优化的部分多消耗了能量，运行时间的减少也可以减少整个系统的能耗。

陷阱：用性能公式的一个子集去度量性能。

50 我们早就指出了一种谬误：简单地只用时钟频率、指令数和 CPI 之一去预测性能。另一种常见的错误是只用三种因素之二去比较性能。虽然这样做在有些条件下可能正确，但这种方法容易误用。实际上，几乎所有取代用时间去度量性能的方法都会导致歪曲的结果或错误的解释。

有一种用 **MIPS** (million instructions per second，每秒百万条指令) 取代时间以度量性能的方法。对于一个给定的程序，MIPS 表示为：

$$\text{MIPS} = \text{指令数} / (\text{执行时间} \times 10^6)$$

MIPS 是指令执行的速率，它规定了性能与执行时间成反比，越快的计算机具有越高的 MIPS 值。从表面看，MIPS 既容易理解，又符合人的直觉。

② MIPS：基于百万条指令的程序执行速度的一种测量。指令条数除以执行时间与 10^6 之积就得到了 MIPS。

其实，用 MIPS 作为度量性能的指标存在三个问题。首先，MIPS 规定了指令执行的速率，但没有考虑指令的能力。我们没有办法用 MIPS 比较不同指令集的计算机，因为指令数肯定是不同的。其次，在同一计算机上，不同的程序会有不同的 MIPS，因而一台计算机不会只有一个 MIPS 值。例如，将执行时间用 MIPS、CPI、时钟频率代入之后可得：

$$\text{MIPS} = \text{指令数} / (\text{指令数} \times \text{CPI} / \text{时钟频率} \times 10^6) = \text{时钟频率} / (\text{CPI} \times 10^6)$$

回顾一下，图 1-18 显示了 SPEC2006 在 Intel Core i7 上的 CPI 最大值和最小值是相差 5 倍的，MIPS 也是如此。最后一点，也是最重要的一点，如果一个新程序执行的指令数更多，但每条指令的执行速度更快，则 MIPS 的变化是与性能无关的。

01 小测验

某程序在两台计算机上的性能测量结果为：

测量内容	计算机 A	计算机 B
指令数	100 亿次	80 亿次
时钟频率	4GHz	4GHz
CPI	1.0	1.1

- 哪台计算机的 MIPS 值更高？
- 哪台计算机更快？

1.11 本章小结

那里……ENIAC 配备有 18 000 个真空管，重量达 30 吨，未来的计算机具有 1000 个真空管，可能仅仅有 1.5 吨重。

——《Popular Mechanics》，1949. 3

虽然很难准确预测未来计算机的成本与性能将发展到怎样的水平，但可以确定的是一定会比现在的计算机更好。计算机性能水平的提高是永无止境的，计算机设计者和程序员必须理解更广泛的问题。

硬件和软件设计者都采用分层的方法构建计算机系统，每个下层都对其上层隐藏本层的细节。抽象原理是理解当今计算机系统的基础，但这并不意味着设计者只要懂得抽象原理就足够了。也许最重要的抽象层次是硬件和底层软件之间的接口，称为指令集体系结构。将指令集体系结构作为一个常量可以使不同的实现方法（价格和性能可能不同）能够运行同一软件。这种方法产生的一个副效应是，要预先排除可能需要接口发生变化的那些革新结构。

有一个可靠的测定性能的方法，即用实际程序的执行时间作为尺度。该执行时间与我们能够通过下面公式测量到的其他重要指标相关：

$$\frac{\text{秒数}}{\text{程序}} = \frac{\text{指令数}}{\text{程序}} \times \frac{\text{时钟周期数}}{\text{指令数}} \times \frac{\text{秒数}}{\text{时钟周期数}}$$

本书中我们将多次使用这一公式及其组成因子。必须明确的是，任何一个独立的因子都不能确定性能，只有三个因子的乘积（即执行时间）才是可靠的性能度量标准。

01 重点 执行时间是唯一有效且不可推翻的性能度量方法。人们曾经提出许多其他度量方法，但均以失败告终。有些从一开始就没有反映执行时间，因而是无效的；还有一些只能在有限条件下有效，超出了限制条件则失效，或是没有清晰地说明有效性的限制条件。

现代处理器的关键硬件技术是硅。与理解集成电路技术同样重要的是理解我们所期望的摩尔定律中描述的技术进步速率。在硅技术加快硬件进步的同时，计算机组织的新思想也改进了产品的性价比。其中有两个重要的新思想：第一，在程序中开发并行性，目前的典型方法是借助多处理器；第二，开发存储器层次结构的访问局部性，目前的典型方法是通过 cache。

能量效率已经取代芯片面积，成为微处理器设计中最重要的资源。保存功耗并且改进性能的需求已经迫使硬件工业向多核微处理器跃进，从而迫使软件工业向并行硬件编程跃进。并行化现在是提高性能的必要途径。

计算机设计总是以价格和性能来度量的，也包括其他一些重要的因素，如能耗、可靠性、成本和可扩展性等。尽管本章的重点在于价格、性能和能耗，但是最佳的设计应该在特定的应用领域中取得所有因素之间适当的平衡。

本书导读

在抽象的底部是计算机的 5 个经典部件：数据通路、控制器、存储器、输入和输出（见图 1-5）。这 5 个部件也是本书后面几章的框架：

- 数据通路：第 3、4、6 章和附录 C
- 控制器：第 4、6 章和附录 C
- 存储器：第 5 章
- 输入：第 5 章和第 6 章
- 输出：第 5 章和第 6 章

如上所述，第4章介绍处理器如何开发隐式并行性，第6章介绍并行革命的核心——显式并行多核微处理器，附录C介绍高度并行的图像处理器芯片。第5章介绍如何开发层次存储结构的访问局部性。第2章介绍指令集（编译器和计算机之间的接口），并强调了编译器和编程语言在利用指令集特性方面的作用。附录A提供了第2章指令集的参考数据。第3章介绍计算机如何处理算术运算数据。附录B介绍逻辑设计。

53

1.12 历史观点和拓展阅读

活跃的科学领域就像一个巨大的蚂蚁窝；人们消失在互相对立的观点中，以光速传递着信息，将信息从一个地方传到另一个地方。

——Lewis Thomas, 《Lives of a cell》中的“自然科学”，1974

本书的每一章都有“历史观点和拓展阅读”一节，可在本书配套网站上找到。我们可以通过一系列的计算机来追踪某一思想的发展历程，或者叙述一些历史上重要的项目贡献，还提供参考资料以便进一步探究。

本章的“历史观点”提供了几个关键思想的历史背景，其目的是向你介绍对技术进步做出贡献的重要历史人物以及他们的事迹。通过理解过去，你可以更好地理解那些推动未来计算技术进步的力量。配套网站中每个历史观点之后都会提示进一步阅读，这部分具体内容见配套网站中的“进一步阅读”部分。在配套网站可下载1.12节的剩余部分。

1.13 练习题

完成练习所需的相对时间比率标示在题号之后的方括号中。平均来说，做标记[10]的练习用的时间是做标记[5]的练习的2倍。做题前应先阅读的章节则标示在尖括号中。例如，<1.4>表示你应该在读过1.4节后才能完成本题。

- 1.1 [2]<1.1>列举和描述除智能手机之外的4种类型的计算机。
- 1.2 [5]<1.2>计算机系统结构中的8个伟大思想与其他领域的思想相同。将计算机系统结构中的8个伟大思想“面向摩尔定律的设计”、“使用抽象简化设计”、“加速大概率事件”、“采用并行提高性能”、“采用流水线提高性能”、“采用预测提高性能”、“存储器层次”、“通过冗余提高可靠性”与其他领域的下列思想进行匹配：
 - a. 汽车制造中的组装生产线
 - b. 吊桥缆索
 - c. 采用风向信息的飞机和船舶导航系统
 - d. 高楼中的高速电梯
 - e. 图书馆的预定台
 - f. 通过增大CMOS晶体管的栅极面积来减小翻转时间
 - g. 增加电磁飞机弹射器（不同于流体驱动模型，它采用电驱动），允许有新型反应堆技术才生成更多的能量
 - h. 制造自动驾驶汽车，其控制系统是安装在汽车上的传感器系统，例如车道偏离检测系统和智能导航控制系统
- 1.3 [2]<1.3>讲述高级语言（例如C）编写的程序转化为能够直接在计算机处理器上执行的表示的步骤。
- 1.4 [2]<1.4>一个彩色显示器中的每个像素由三种基色（红，绿，蓝）构成，每种基色用8位表示，分辨率为 1280×1024 像素。
 - a. 为了保存一帧图像最少需要多大的缓存（以字节计算）？
 - b. 在100Mbit/s的网络上传输一帧图像最少需要多长时间？

54

- 1.5** [4] <1.6> 有 3 种不同的处理器 P1、P2 和 P3 执行同样的指令集，P1 的时钟频率为 3GHz，CPI 为 1.5；P2 的时钟频率为 2.5GHz，CPI 为 1.0；P3 的时钟频率为 4GHz，CPI 为 2.2。
- 以每秒钟执行的指令数目为标准，哪个处理器性能最高？
 - 如果每个处理器执行一个程序都花费 10 秒钟时间，求它们的时钟周期数和指令数。
 - 我们试图把执行时间减少 30%，但这会引起 CPI 增加 20%。问：时钟频率应该是多少才能达到时间减少 30% 的目的？
- 1.6** [20] <1.6> 同一个指令集体系结构有两种不同的实现方式。根据 CPI 的不同将指令分成 4 类（A、B、C 和 D），P1 的时钟频率为 2.5GHz，CPI 分别为 1、2、3 和 3；P2 时钟频率为 3GHz，CPI 分别为 2、2、2 和 2。
给定一个程序，有 1.0×10^6 条动态指令，按如下比例分为 4 类：A，10%；B，20%；C，50%；D，20%。
- 每种实现方式总的 CPI 是多少？
 - 计算两种情况下的时钟周期。
- 1.7** [15] <1.6> 编译程序对一个应用在给定的处理器上的性能有极深的影响。假定一个程序，如果采用编译程序 A，则动态指令数为 1.0×10^9 ，执行时间为 1.1s；如果采用编译程序 B，则动态指令数为 1.2×10^9 ，执行时间为 1.5s。
- 在给定处理器时钟周期为 1ns 时，找出每个程序的平均 CPI。
 - 假定编译程序是在两个不同的处理器上运行的。如果这两个处理器的执行时间相同，求运行编译程序 A 的处理器时钟相对于运行编译程序 B 的处理器的时钟快多少？
 - 假设开发了一种新的编译程序，只用 6.0×10^8 条指令，平均 CPI 为 1.1。求这种新的编译程序相对于原先编译程序 A 和 B 的加速比。
- 1.8** 2004 年发布的 Pentium 4 Prescott 处理器时钟频率为 3.6GHz，工作电压为 1.25V。假定平均情况下静态功耗为 10W，动态功耗为 90W。
2012 年发布的 Core i5 Ivy Bridge 时钟频率为 3.4GHz，工作电压为 0.9V。假定平均情况下静态功耗为 30W，动态功耗为 40W。
- 1.8.1** [5] <1.7> 分别求出每个处理器的平均电容负载。
- 1.8.2** [5] <1.7> 对于每种工艺，求出静态功耗占总功耗的比例和静态功耗相对于动态功耗的比率。
- 1.8.3** [15] <1.7> 如果要将整体功耗降低 10%，求出在保持漏电流不变的情况下电压要降低多少？
注意：功耗定义为电压与电流的乘积。
- 1.9** 在一个处理器中，假定算术指令、load/store 指令和分支指令的 CPI 分别是 1、12 和 5。另外假定一个程序在单个处理器核上运行时需要执行 2.56×10^9 条算术指令、 1.28×10^9 条 load/store 指令和 2.56E8 条分支指令，并假定处理器的时钟频率为 2GHz。
现假定程序并行运行在多核上，分配到每个处理器核上运行的算术指令和 load/store 指令数目为单核情况下相应指令数目除以 $0.7 \times p$ (p 是处理器的数量)，而每个处理器的分支指令的数量保持不变。
- 1.9.1** [5] <1.7> 求出当该程序分别运行在 1、2、4 和 8 个处理器核上的执行时间，并求出其他情况下相对于单核处理器的加速比。
- 1.9.2** [10] <1.6, 1.8> 如果算术指令的 CPI 加倍，对分别运行在 1、2、4 和 8 个处理器核上的执行时间有何影响？
- 1.9.3** [10] <1.6, 1.8> 如果要使单核处理器的性能与四核处理器相当，单处理器中 load/store 指令的 CPI 应该降低多少？假定四核处理器的 CPI 保持不变。
- 1.10** 假定一个直径 15cm 的晶圆的成本是 12，包含 84 块芯片，其缺陷参数为 $0.020 \text{ 瑕疵}/\text{cm}^2$ 。而一个直径 20cm 的晶圆的成本是 15，包含 100 块芯片，其缺陷参数为 $0.031 \text{ 瑕疵}/\text{cm}^2$
- 1.10.1** [10] <1.5> 分别求出每种芯片的成品率。
- 1.10.2** [5] <1.5> 分别求出每种芯片的价格。
- 1.10.3** [5] <1.5> 如每晶圆的芯片数增加 10%，每单位面积的瑕疵数增加 15%，求芯片面积和成品率。

55

56

- 1.10.4** [5] <1.5> 假设随着电子器件制造技术的进步，成品率从 0.92 上升到 0.95。给定芯片面积为 200mm²，求每一种技术下单位面积的瑕疵数。
- 1.11** SPEC CPU 2006 的 bzip2 基准程序在 AMD Barcelona 处理器上执行的总指令数为 2.38×10^{12} ，执行时间为 750s，参考时间为 9650s。
- 1.11.1** [5] <1.6, 1.9> 如果时钟周期时间为 0.333ns，求 CPI 值。
- 1.11.2** [5] <1.9> 求 SPEC 的分值。
- 1.11.3** [5] <1.6, 1.9> 如果基准程序的指令数增加 10%，CPI 不变，求 CPU 时间增加多少？
- 1.11.4** [5] <1.6, 1.9> 如果基准程序的指令数增加 10%，CPI 增加 5%，求 CPU 时间增加多少？
- 1.11.5** [5] <1.6, 1.9> 根据上题中指令数和 CPI 的变化，求 SPEC 分值的变化。
- 1.11.6** [10] <1.6> 假设开发了一款新的 AMD Barcelona 处理器，其工作频率为 4GHz，在其指令集中增加了一些新的指令，从而使程序中指令数目减少了 15%，程序的执行时间减少到了 700s，新的 CPI 分值为 13.7，求新的 CPI。
- 1.11.7** [10] <1.6> 当时钟频率由 3GHz 上升到 4GHz 时，上一小题算出的 CPI 比 1.11.1 的高。请确定 CPI 的升高是否与频率升高相同？如果不同，为什么？
- 57** **1.11.8** [5] <1.6> CPU 时间减少了多少？
- 1.11.9** [10] <1.6> 对第二个基准程序 libquantum，假定执行时间为 960ns，CPI 为 1.61，时钟频率为 3GHz。在时钟频率为 4GHz 时，在不影响 CPI 的前提下执行时间降低 10%，求指令数。
- 1.11.10** [10] <1.6> 在指令数和 CPI 保持不变的前提下，如果要将 CPU 时间进一步减少 10%，求时钟频率。
- 1.11.11** [10] <1.6> 在指令数保持不变的前提下，如果要将 CPI 降低 15%，CPU 时间减少 20%，求时钟频率。
- 1.12** 1.10 节引证了一个用性能公式的一个子集去计算性能的陷阱。为了说明它，考虑下面两种处理器。P1 的时钟频率为 4GHz，平均 CPI 为 0.9，需要执行 5.0×10^9 条指令；P2 的时钟频率为 3GHz，平均 CPI 为 0.75，需要执行 1.0×10^9 条指令。
- 1.12.1** [5] <1.6, 1.10> 一个常见的错误是，认为时钟频率最高的计算机具有最高的性能。这种说法正确吗？请用 P1 和 P2 来验证这一说法。
- 1.12.2** [10] <1.6, 1.10> 另一个错误是，认为执行指令最多的处理器需要更多的 CPU 时间。考虑 P1 执行 1.0×10^9 条指令序列所需的时间，P1 和 P2 的 CPI 不变，计算一下 P2 用同样的时间可以执行多少条指令？
- 1.12.3** [10] <1.6, 1.10> 一个常见的错误是用 MIPS（每秒百万条指令）来比较两台不同的处理器的性能，并认为 MIPS 最大的处理器具有最高的性能。这种说法正确吗？请用 P1 和 P2 验证这一说法。
- 1.12.4** [10] <1.10> 另一个常见的性能标志是 MFLOPS（每秒百万条浮点指令），其定义为

$$\text{MFLOPS} = \text{浮点操作的数目} / (\text{执行时间} \times (1 \times 10^6))$$
 它与 MIPS 有同样的问题。假定 P1 和 P2 上执行的指令有 40% 的浮点指令，求出程序的 MFLOPS。
- 1.13** 1.10 节引证另一个易犯的错误是通过只改进计算机的一个方面来改进计算机的总体性能。假如一台计算机上运行一个程序需要 250s，其中 70s 用于执行浮点指令，85s 用于执行 L/S 指令，40s 用于执行分支指令。
- 58** **1.13.1** [5] <1.10> 如果浮点操作的时间减少 20%，总时间将减少多少？
- 1.13.2** [5] <1.10> 如果总时间减少 20%，整数操作时间将减少多少？
- 1.13.3** [5] <1.10> 如果只减少分支指令时间，总时间能否减少 20%？
- 1.14** 假定一个程序需要执行 50×10^6 条浮点指令、 110×10^6 条整数指令、 80×10^6 条 L/S 指令和 16×10^6 条分支指令。每种类型指令的 CPI 分别是 1、1、4 和 2。假定处理器的时钟频率为 2GHz。
- 1.14.1** [10] <1.10> 如果我们要将程序运行速度提高至 2 倍，浮点指令的 CPI 需如何改进？
- 1.14.2** [10] <1.10> 如果我们要将程序运行速度提高至 2 倍，L/S 指令的 CPI 需如何改进？

1.14.3 [5] <1.10> 如果整数和浮点指令的 CPI 减少 40%，L/S 和分支指令的 CPI 减少 30%，程序的执行时间能改进多少？

1.15 [5] <1.8> 多处理器系统中的执行时间可分成例程计算时间加处理器之间的通信时间。

假定一个程序在单处理器上执行时需要 $t = 100\text{s}$ 。当它在 p 个处理器上运行时，每个处理器需要 $t/p\text{s}$ 的计算时间，另外还需要 4s 的开销，且开销与处理器数量无关。在处理器数目分别为 2、4、8、16、32、64 和 128 时，计算每个处理器的执行时间。在每种情况下，列出相对于单处理器的加速比和实际加速比与理想加速比的比值（理想加速比是指没有开销情况下的加速比）。

01 小测验答案

1.1 问题讨论：可以有多种答案。

1.3 DRAM 存储器：易失性，访问时间短（大约 $50 \sim 70\text{ns}$ ），每 GB 的价格（\$5 ~ \$10）。磁盘存储器：非易失性，访问时间比 DRAM 慢 $100\,000 \sim 400\,000$ 倍，每 GB 的价格比 DRAM 便宜 100 倍。Flash 存储器：非易失性，访问时间比 DRAM 慢 $100 \sim 1\,000$ 倍，每 GB 的价格比 DRAM 便宜 7 ~ 10 倍。

1.5 1、3、4 是正确答案，答案 5 一般可认为正确，因为产量高时能促使额外投资去减小芯片面积，例如减小 10%，这是一种经济决策，但并不总是正确。

1.6 1. 两者都改进，2. 延迟，3. 都不改进；7s。

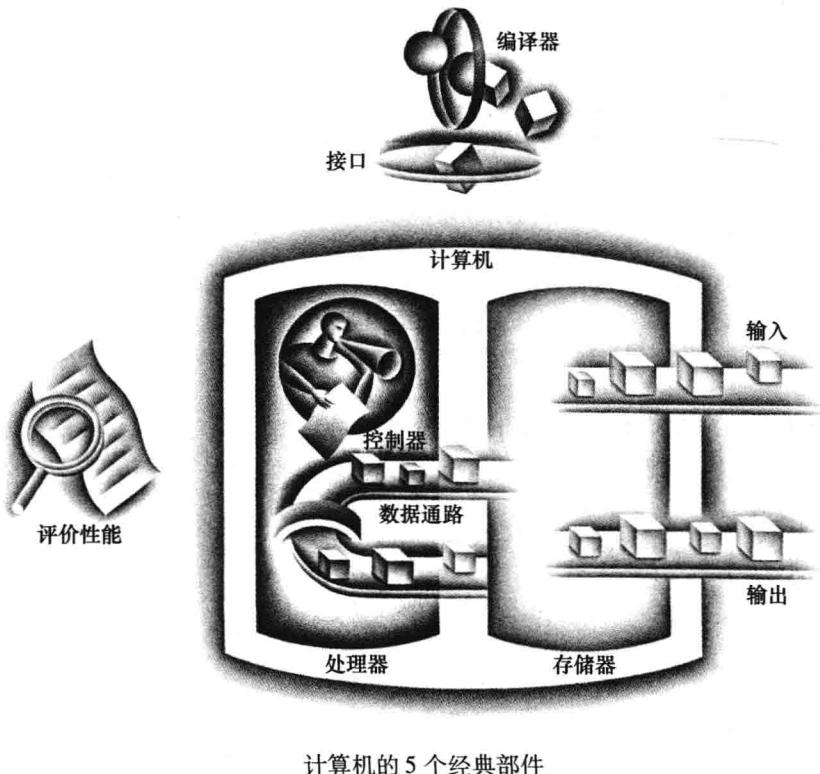
1.6 b

1.10 a. 计算机 A 有较高的 MIPS 值；b. 计算机 B 更快。

第2章 |

Computer Organization and Design: The Hardware/Software Interface

指令：计算机的语言



我对上帝说西班牙语，对女人说意大利语，对男人说法语，对我的马说德语。

——法王查理五世 (1500—1558)

2.1 引言

要计算机服从指挥，就必须用计算机的语言。计算机语言中的基本单词称为指令，一台计算机的全部指令称为该计算机的指令集（instruction set）。本章将介绍实际计算机指令集的两种形式：一种是人们编程书写的语言，另一种是计算机所能识别的形式。我们将以自顶向下的方式来介绍，从看似受约束的汇编语言助记符开始，逐步精炼到实际计算机的真实语言。第3章将继续采用这种向下探究的方式，揭示算术运算的硬件以及浮点数的表示方法。

● 指令集：一个给定的计算机体系结构所包含的指令集合。

尽管机器语言种类繁多，但它们之间十分类似，其差异性更像人类语言中的“方言”，而非各自独立的语言。因此，理解了一种机器语言，其他种类的机器语言也就容易理解了。

本书选择 MIPS 技术的指令集，它是自 20 世纪 80 年代以来出现的优秀指令集。通过简要介绍其他三种流行的指令集可以看出 MIPS 指令的优势。

1) ARMv7 与 MIPS 类似。2011 年，ARM 处理器芯片的产量超过 90 亿片，这使得 ARMv7 成为最流行的指令集。

2) 第二个例子是 Intel x86，在 PC 领域和后 PC 时代的云计算领域占统治地位。

3) 第三个例子是 ARMv8，它将 ARMv7 的地址范围由 32 位扩展到 64 位。而具有讽刺意味的是，这个 2013 年产生的指令集更加接近于 MIPS，而非 ARMv7。

这种相似性一方面是因为所有计算机都是基于基本原理相似的硬件技术所构建的，另一方面是因为所有计算机都必须提供一些基本操作。此外，计算机设计者有一个共同的目标：找到一种语言，可方便硬件和编译器的设计，且使性能最佳，同时使成本和功耗最低。但实现这个目标需要长期的探索。下述引文写于计算机出现不久的 1947 年，但今天它仍然适用：

用形式逻辑的方法可以很容易看到，在理论上存在着某种“指令集”足以控制任何的操作序列并使之执行……从当前的观点出发，在选择一个“指令集”时，真正的决定性因素是要更多地考虑其实际性质：“指令集”要求的设备简单性，它的应用对于解决实际重要问题的明确性以及它解决该类问题的处理速度。

——Burks, Goldstine, von Neumann, 1947

无论是对 20 世纪 50 年代的计算机而言，还是对现代的计算机来说，“设备简单性”都是值得考虑的重要问题。本章的目的就是讲解符合此原则的一种指令集，介绍它怎样用硬件表示，以及它和高级编程语言之间的关系。我们的示例使用 C 语言编写，2.15 节介绍了在使用像 Java 这样的面向对象语言时会有什么不同。

通过理解如何表述指令，读者也将发现计算的秘密：存储程序概念（stored-program concept）。此外，通过使用机器语言编程，并在本书提供的模拟器中运行，读者将进一步体会到编程语言和编译优化对程序性能的影响。本章结束时我们将简要介绍指令集的发展历史和其他的计算机“方言”。

◎ 存储程序概念：多种类型的指令和数据均以数字形式存储于存储器中的概念，存储程序型计算机即源于此。

我们结合计算机的结构，逐步讲解 MIPS 指令集。采用自顶向下、循序渐进的方法并结合各部件及其说明，尽量使机器语言变得不再枯燥。图 2-1 给出了本章将要介绍的指令集的总体情况。

MIPS 操作数

名字	示例	注释
32 个寄存器	\$s0 - \$s7, \$t0 - \$t9, \$zero, \$a0 - \$a3, \$v0 - \$v1, \$gp, \$fp, \$sp, \$ra, \$at	寄存器用于数据的快速存取。在 MIPS 中，只能对存放在寄存器中的数据执行算术操作，寄存器 \$zero 的值恒为 0，寄存器 \$at 被汇编器保留，用于处理大的常数
2 ³⁰ 个存储器字	Memory [0], Memory [4], …, Memory [4294967292]	存储器只能通过数据传输指令访问。MIPS 使用字节编址，所以连续的字地址相差 4。存储器用于保存数据结构、数组和溢出的寄存器

MIPS 汇编语言

类别	指令	示例	含义	注释
算术	加法	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	三个寄存器操作数
	减法	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	三个寄存器操作数
	立即数加法	addi \$s1, \$s2, 20	$\$s1 = \$s2 + 20$	用于加常数数据

图 2-1 本章要讲解的是 MIPS 汇编语言指令。示例含义注释信息也可以在 MIPS 参考数据卡的第 1 列中找到

MIPS 汇编语言

类别	指令	示例	含义	注释
数据传输	取字	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将一个字从内存中取到寄存器中
	存字	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	将一个字从寄存器中取到内存中
	取半字	lh \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将半个字从内存中取到寄存器中
	取无符号半字	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将半个字从内存中取到寄存器中
	存半字	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	将半个字从寄存器存到内存中
	取字节	lb \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将一个字节从内存取到寄存器中
	取无符号字节	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将一个字节从内存取到寄存器中
	存字节	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	将一个字节从寄存器存到内存中
	取链接字	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	取字作为原子交换的前半部
	存条件字	sc \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1; \$s1 = 0 or 1	存字作为原子交换的后半部分
逻辑	取立即数的高位	lui \$s1,20	\$s1 = 20 * 2 ¹⁶	取立即数并放到高16位
	与	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	三个寄存器操作数按位与
	或	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	三个寄存器操作数按位或
	或非	nor \$s1,\$s2,\$s3	\$s1 = ~(\$s2 \$s3)	三个寄存器操作数按位或非
	立即数与	andi \$s1,\$s2,20	\$s1 = \$s2 & 20	和常数按位与
	立即数或	ori \$s1,\$s2,20	\$s1 = \$s2 20	和常数按位或
	逻辑左移	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	根据常数左移相应位
条件分支	逻辑右移	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	根据常数右移相应位
	相等时跳转	beq \$s1,\$s2,25	if(\$s1 == \$s2) go to PC + 4 + 100	相等检测；和PC相关的跳转
	不相等时跳转	bne \$s1,\$s2,25	if(\$s1 != \$s2) go to PC + 4 + 100	不相等检测；和PC相关的跳转
	小于时置位	slt \$s1,\$s2,\$s3	if(\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	比较是否小于；beq, bne
	无符号数比较小于时置位	sltu \$s1,\$s2,\$s3	if(\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	比较是否小于无符号数
	无符号数比较小于立即数时置位	slti \$s1,\$s2,20	if(\$s2 < 20) \$s1 = 1; else \$s1 = 0	比较是否小于常数
无条件跳转	无符号数比较小于无符号立即数时置位	sltiu \$s1,\$s2,20	if(\$s2 < 20) \$s1 = 1; else \$s1 = 0	比较是否小于无符号常数
	跳转	j 2500	go to 10000	跳转到目标地址
	跳转至寄存器所指位置	jr \$ra	go to \$ra	用于switch语句，以及过程调用
	跳转并链接	jal 2500	\$ra = PC + 4; go to 10000	用于过程调用

图 2-1 (续)

2.2 计算机硬件的操作

毫无疑问，计算机必须有执行基本算术运算操作的指令。

——Burks、Goldstine、von Neumann, 1947

任何计算机必须能够执行算术运算。MIPS 汇编语言的下述记法

```
add a, b, c
```

表示将两个变量 b 和 c 相加，并将它们的和放入变量 a 中。

这种记法的表示方式是固定的：每条 MIPS 算术指令只执行一个操作，并且有且仅有 3 个变量。例如，若要将变量 b、c、d、e 之和放入变量 a 中（本节不深究“变量”的含义，下一节将给出其详细说明）。

下面的指令序列将完成此 4 个变量的相加：

```
add a, b, c      # The sum of b and c is placed in a
add a, a, d      # The sum of b, c, and d is now in a
add a, a, e      # The sum of b, c, d, and e is now in a
```

以上 3 条指令完成了 4 个变量的相加。

上述每行代码中，符号“#”右边的是注释，用于帮助人们理解程序，而计算机将忽略它们。注意与其他编程语言不同的是，这种语言的每一行最多只有一条指令。另一个与 C 语言不同的是，注释总是在一行的末尾结束。

与加法类似的指令一般都有三个操作数：两个进行运算的数和一个保存结果的数。要求每条指令有且仅有三个操作数，这一点符合硬件简单性的设计原则：操作数个数可变将给硬件设计带来更大的复杂性。这种情况说明了硬件设计三条基本原则的第一条：

设计原则 1：简单源于规整。

下面的两个示例程序展示了用高级编程语言编写的程序和用汇编语言编写的程序之间的关系。

01 例题·把 C 语言中两条赋值语句编译成 MIPS

本例中 C 语言程序包含 5 个变量，a、b、c、d 和 e。因为 Java 语言由 C 语言演化而来，所以本例及以后若干例子对这两种高级语言均适用：

```
a = b + c;
d = a - e;
```

将 C 语言程序转换为 MIPS 汇编指令是由编译器完成的。写出由编译器生成的 MIPS 代码。

01 答案

一条 MIPS 指令对来自两个源操作数寄存器的操作数进行运算，并将结果存入目的寄存器。因此上面两条简单的 C 语句可直接编译为如下两条 MIPS 汇编指令：

```
add a, b, c
sub d, a, e
```

01 例题·把 C 语言中一条复杂的赋值语句编译成 MIPS

下面一行复杂的 C 语句包含 5 个变量 f、g、h、i 和 j：

```
f = (g + h) - (i + j);
```

C 编译器将产生什么样的 MIPS 汇编语言代码？

01 答案

因为一条 MIPS 指令仅执行一个操作，所以编译器必须将这条 C 语句编译成多条汇编指令。若第一条指令计算 g 与 h 的和，其结果必须暂存在某一个地方。因此，编译器需创建一个临时

变量 t0：

```
add t0,g,h # temporary variable t0 contains g + h
```

虽然下一个操作是减法，但在做减法操作之前，必须先计算出 i 与 j 的和。因此，第二条指令将 i、j 之和存于由编译器创建的另一个临时变量 t1 中：

```
add t1,i,j # temporary variable t1 contains i + j
```

最后，用一条减法指令将两个临时变量中的值相减，结果存入变量 f，完成编译：

```
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)
```

□

01 小测验

对于一个给定的功能，用下列哪种编程语言实现可能花费的代码行数最多？将下面 3 种语言排序：

1. Java
2. C
3. MIPS 汇编语言

01 精解 为了增强可移植性，Java 最初被设定为依靠软件解释器执行的语言。解释器的指令集称作 Java 字节码（Java bytecode，参见 2.15 节），它与 MIPS 指令集有很大不同。为使性能与等效功能的 C 程序接近，Java 系统现在的典型做法是将字节码编译成类似 MIPS 这样的机器指令。因为通常 Java 完成编译的时间迟于 C，所以 Java 编译器常称为即时编译器（Just In Time，JIT）。2.12 节展示了在程序启动阶段 JIT 是如何迟于 C 编译器的，2.13 节展示了 Java 程序的编译执行和解释执行的性能比较。

2.3 计算机硬件的操作数

与高级语言程序不同，MIPS 算术运算指令的操作数是很严格的，它们必须来自寄存器。寄存器由硬件直接构建且数量有限，是计算机硬件设计的基本元素。当计算机设计完成后，寄存器对程序员是可见的，所以也可以把寄存器想象成构造计算机“建筑”的“砖块”。在 MIPS 体系结构中寄存器大小为 32 位，由于 32 位为一组的情况经常出现，因此在 MIPS 体系结构中将其称为字（word）。

66 ◀ 字：计算机中的基本访问单位，通常是 32 位为一组，在 MIPS 体系结构中与寄存器大小相同。

高级语言的变量与寄存器的一个主要区别在于寄存器的数量有限，MIPS 一类的典型的现代计算机中有 32 个寄存器（参见 2.21 节有关寄存器数目的演变历史）。下面继续以自顶向下的方式引入新的 MIPS 语言的符号表示。在本节中 MIPS 算术指令的三个操作数限定为必须从 32 个 32 位寄存器中选取。

寄存器个数限制为 32 个的理由可以表示为硬件设计三条基本原则中的第二条：

设计原则 2：越小越快。

大量的寄存器可能会使时钟周期变长，因为电信号传输更远的距离必然花费更长的时间。

当然，该原则也不是绝对的，31 个寄存器不见得比 32 个更快。但表象背后的物理事实值得计算机设计者认真对待。在这种情况下，设计者必须在程序期望更多寄存器和加快时钟周期之间进行权衡。另一个不使用多于 32 个寄存器的原因是受指令格式位数的限制，这在 2.5 节有相应介绍。

第4章论证了寄存器在硬件结构中所扮演的核心角色。正如该章所述，有效利用寄存器对于提高程序性能极为重要。

尽管可以简单使用序号0~31表示相应的寄存器，但MIPS约定书写指令时用一个“\$”符后面跟两个字符来代表一个寄存器。2.8节将解释这一做法的理由。现在，我们使用\$\\$s0, \$\\$s1, …来表示与C和Java程序中的变量所对应的寄存器；用\$\\$t0, \$\\$t1, …来表示将程序编译为MIPS指令时所需的临时寄存器。

01 例题·使用寄存器编译C赋值语句

将程序变量和寄存器对应起来是编译器的工作。以我们前面讲过的C赋值语句为例：

$f = (g + h) - (i + j);$

变量f、g、h、i和j依次分配给寄存器\$\\$s0、\$\\$s1、\$\\$s2、\$\\$s3和\$\\$s4。编译后的MIPS代码是什么？

67

01 答案

除了将变量用上述寄存器代替，将两个临时变量用\$\\$t0和\$\\$t1代替外，编译后生成的代码与前面例题中的代码非常相似：

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

□

2.3.1 存储器操作数

编程语言中，有像上面这些例题中仅含一个数据元素的简单变量，也有像数组或结构那样的复杂数据结构。这些复杂数据结构中的数据元素可能远多于计算机中寄存器的个数。计算机怎样来表示和访问这样大的结构呢？

回忆一下第1章所描述的计算机的5个组成部分。处理器只能将少量数据保存在寄存器中，但存储器有数十亿的数据元素。因此，数据结构（如数组和结构）是存放在存储器中的。

如上所述，MIPS的算术运算指令只对寄存器进行操作，因此，MIPS必须包含在存储器和寄存器之间传送数据的指令。这些指令叫作**数据传送指令**（data transfer instruction）。为了访问存储器中的一个字，指令必须给出存储器地址（address）。存储器就是一个很大的下标从0开始的一维数组，地址就相当于数组的下标。例如，在图2-2中，第三个数据元素的地址为2，存放的数据为10。

- **数据传送指令：**在存储器和寄存器之间移动数据的命令。
- **地址：**用于在存储器空间中指明某特定数据元素位置的值。

将数据从存储器复制到寄存器的数据传送指令通常叫取数（load）指令。取数指令的格式是操作码后接着目标寄存器，再后面是用来访问存储器的常数和寄存器。常数和第二个寄存器中的值相加即得存储器地址。实际的MIPS取数指令助记符为lw，为load word的缩写。

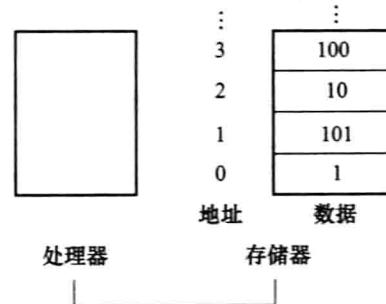


图2-2 存储器地址和该地址对应的数据。如果这些元素是字，那么这些地址就是错误的，因为MIPS实际上是按字节编址的，而一个字是4字节。图2-3给出了顺序字编址的内存寻址

68

01 例题·编译一个操作数在存储器中的C赋值语句

设 A 是一个含有 100 个字的数组，像前面的例题一样，编译器仍然将寄存器 \$s1、\$s2 依次分配给变量 g、h。又设数组 A 的起始地址（或称基址（base address））存放在寄存器 \$s3 中。试编译下面的 C 赋值语句：

```
g = h + A[8];
```

01 答案

虽然该 C 赋值语句只有一个操作，但其中一个操作数在存储器中，所以首先必须将 A [8] 传递到寄存器中。其地址是 \$s3 中的基址加上该元素序号 8。取回的数据应放在一个临时寄存器中以便下条指令使用。由图 2-2 可知，编译后生成的第一条指令为：

```
lw    $t0,8($s3) # Temporary reg $t0 gets A[8]
```

（这里是一种简化版描述，后面会对这条指令做相关的微调。）因为 A [8] 已取到寄存器 \$t0 中，下一条指令就可对 \$t0 进行操作。该指令将 h（在 \$s2 中）加上 A [8]（在 \$t0 中），并将结果放到对应于 g 的寄存器 \$s1 中：

```
add   $s1,$s2,$t0 # g = h + A[8]
```

数据传送指令中的常量（本例中为 8）称为偏移量（offset），存放基址的寄存器（本例中为 \$s3）称为基址寄存器（base register）。□

01 硬件/软件接口 除了将变量与寄存器对应起来，编译器还在存储器中为诸如数组和结构这样的数据结构分配相应的位置。然后，编译器可以将它们在存储器中的起始地址放到数据传送指令中。

很多程序都用到 8 比特的字节类型，且大多数体系结构按字节编址。因此，一个字的地址必和它所包括的 4 字节中某个的地址相匹配，且连续字的地址相差 4。例如，图 2-3 给出了图 2-2 的实际 MIPS 地址，其中第三个字的字节地址是 8。

因为 MIPS 是按字节编址的，所以字的起始地址必须是 4 的倍数。这叫对齐限制（alignment restriction），许多体系结构都有这样的限制（第 4 章说明了对齐能加快数据传送的理由）。

有两种类型的字节寻址的计算机：一种使用最左边或“大端”（big end）字节的地址作为字地址；另一种使用最右边或“小端”（little end）字节的地址作为字地址。MIPS 采用的是大端编址（big-endian）。由于使用相同的地址去访问一个字和 4 个字节时“端”才起作用，因此大多数情况下不需要关注该问题。（附录 A 中给出了在一个字中对字节进行记数的两种方法。）

字节寻址也影响到数组下标。在上面的代码中，为了得到正确的字节地址，与基址寄存器 \$s3 相加的偏移量必须是 4×8 ，即 32，这样才能正确读到 A [8]，而不会错读到 A [8/4]。（参见 2.19 节中相关陷阱的介绍。）

与取数指令相对应的指令通常叫作存数（store）指令；它将数据从寄存器复制到存储器。存数指令的格式和取数指令相似：首先是操作码，接着是包含待存储数据的寄存

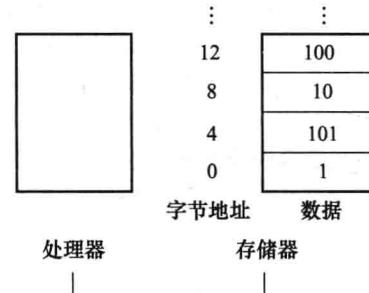


图 2-3 实际的 MIPS 存储器地址和该地址对应的数据。相对于图 2-2，变化的地址采用高亮显示。由于 MIPS 按字节编址，而字地址是 4 的倍数，因为每个字的长度为 4 字节

器，然后是数组元素的偏移量，最后是基址寄存器。同样，MIPS 地址由常数和基址寄存器内容共同决定。实际的 MIPS 存数指令为 sw，即 store word 的缩写。

- 对齐限制：数据地址与存储器的自然边界对齐的要求。

69

01 硬件/软件接口 由于 load 和 store 指令中的地址是二进制，因此作为主存的 DRAM 的容量使用二进制表示，而非十进制。例如，使用 gibibytes (2^{30}) 或 tebibytes (2^{40}) 表示，而不用 gigabytes (10^9) 或 terabytes (10^{12})，见图 1-1。

70

01 例题·用取数/存数指令进行编译

假设变量 h 存放在寄存器 \$s2 中，数组 A 的基址放在 \$s3 中。试编译下面的 C 赋值语句：
 $A[12] = h + A[8];$

01 答案

虽然该 C 语句只有一个操作，但是有两个操作数在存储器中，因此，需要更多的 MIPS 指令。前两条指令基本上与上个例题相同，除了本例在取数指令中选择 A [8] 时使用了字节寻址中正确的偏移量，并且加法指令将结果放在临时寄存器 \$t0 中：

```
lw $t0,32($s3) # Temporary reg $t0 gets A[8]
add $t0,$s2,$t0 # Temporary reg $t0 gets h + A[8]
```

最后一条指令使用 48 (4×12) 作为偏移量，寄存器 \$s3 作为基址寄存器，将加法结果存放到存储器单元 A [12] 中。

```
sw $t0,48($s3) # Stores h + A[8] back into A[12]
```

lw 和 sw 是 MIPS 体系结构中在存储器和寄存器之间复制字的指令。其他计算机有各自相应的取数/存数指令来传送数据。Intel x86 体系结构中类似的指令见 2.17 节。 □

01 硬件/软件接口 许多程序的变量个数要远多于计算机的寄存器个数。因此，编译器会尽量将最常用的变量保持在寄存器中，而将其他的变量放在存储器中，方法是使用取数/存数指令在寄存器和存储器之间传送变量。将不常使用的变量（或稍后才使用的变量）存回到存储器中的过程叫作寄存器溢出（spilling）。

根据硬件设计原则 2，存储器一定比寄存器慢，因为寄存器数量更少。事实的确如此，访问寄存器中的数据要远快于访问存储器中的数据。

另外，寄存器中的数据更容易利用。一条 MIPS 算术运算指令能完成读两个寄存器、对它们进行运算以及写回运算结果的操作。而一条 MIPS 数据传送指令只能完成读一个操作数或写一个操作数的操作，并且不能对它们进行运算。

寄存器与存储器相比，访问时间短、吞吐率高，寄存器中的数据访问速度快并易于利用，访问寄存器相对于访问存储器功耗更小。因此，为了获得高性能和节约功耗，指令集的体系结构必须拥有足够的寄存器，并且编译器必须高效率地利用这些寄存器。

2.3.2 常数或立即数操作数

程序中经常会在某个操作中使用到常数——例如，将数组的下标加 1，用以指向下一个数组元素。实际上，在运行 SPEC CPU 2006 测试基准程序集时，有超过一半的 MIPS 算术运算指令会用到常数作为操作数。

仅从已介绍过的指令看，如果要使用常数必须先将其从存储器中取出。（常数可能是在程序被加载时放入存储器的。）例如，要使寄存器 \$s3 加 4，可以使用下面的代码：

71

```
lw $t0, AddrConstant4($s1)    # $t0 = constant 4
add $s3,$s3,$t0                # $s3 = $s3 + $t0 ($t0 == 4)
```

假设 $\$s1 + \text{AddrConstant4}$ 是常量 4 的存储器地址。

避免使用取数指令的另一方法是，提供其中一个操作数是常数的算术运算指令。这种有一个常数操作数的快速加法指令叫作加立即数 (add immediate)，或者写成 addi。这样，上述操作只需写成：

```
addi   $s3,$s3,4           # $s3 = $s3 + 4
```

常数操作数出现频率高，而且相对于从存储器中取常数，包含常数的算术运算指令执行速度快很多，并且能耗较低。

常数 0 还有另外的作用，有效使用它可以简化指令集。例如，数据传送指令正好可以被视作一个操作数为 0 的加法。因此，MIPS 将寄存器 $\$zero$ 恒置为 0。(此寄存器编号也为 0。)根据使用频率来确定要定义的常数是加速大概率事件的另一个好办法。

01 小测验

根据寄存器的重要性，芯片中寄存器数目随时间的增长率符合下面哪种情况？

1. 非常快：像摩尔定律一样快，该定律预测，芯片上的晶体管数目每 18 个月翻一番。
2. 非常慢：由于程序是通过计算机语言实现的，而指令集体系结构具有惯性，因此寄存器数目的增长要与新指令集的可行性保持一致。

01 精解 虽然本书中讲到的 MIPS 寄存器都是 32 位的，但是也有 64 位版本的 MIPS 指令集，它具有 32 个 64 位的寄存器。为了加以区分，分别将它们称为 MIPS-32 和 MIPS-64。在本章中，我们使用 MIPS-32 的子集。附录 E 中介绍了 MIPS-32 和 MIPS-64 的区别。2.16 节和 2.18 节介绍了 ARMv7 的 32 位地址和 ARMv8 的 64 位地址之间更多显著的差别。

01 精解 MIPS 中偏移量加基址寄存器的寻址方式非常适合数组和结构，因为基址寄存器可指向结构的首地址，偏移量可用于选择所需的数据元素。在 2.13 节中我们将看到这样的例子。

01 精解 最初设计数据传送指令时，基址寄存器用于保存数组下标，而偏移量用来标示数组的起始地址。因而基址寄存器也叫作下标寄存器 (index register)。现在，存储器容量大大增加，数据分配的软件模型也更为复杂，所以数组的基地址通常放在寄存器中。如同下面将要看到的那样，基地址可能由于过大而不适宜用偏移量表示。

01 精解 由于 MIPS 支持负常数，所以 MIPS 中不需要设置减立即数的指令。

2.4 有符号数和无符号数

首先让我们快速回顾一下计算机是如何表示数的。我们所受的教育是以十进制为基础的，但数的进制可以是任意的。例如，十进制的 123 等于二进制的 1111011。

在计算机硬件中，数是以一串或高或低的电信号来体现的，这恰好可以被认为是基为 2 的数（与基为 10 的数称为十进制数一样，基为 2 的数称为二进制数）。

所有信息都由二进制数位 (binary digit) 或位 (bit) 组成，因此二进制数运算基本单位是 bit，取值可以是两种状态之一：高或低，开或关，真或假，1 或 0。

② 二进制数位：也称二进制位，二进制状态之一，即 0 或 1，是信息的基本组成单位。

推广到任意进制，第 i 位 d 的值是

$$d \times \text{Base}^i$$

这里， i 是从 0 开始并且从右向左递增。显而易见，计算一个数各位数值的方法是使用幂。我们在十进制数的右下角写上 10，在二进制数的右下角写上 2。例如，

1011_2

表示

$$\begin{aligned} & (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{10} \\ & = (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{10} \\ & = 8 + 0 + 2 + 1_{10} \\ & = 11_{10} \end{aligned}$$

73

在一个 32 位的字中，我们从右向左标记各位为 0, 1, 2, 3…，下面的图片表示了 MIPS 字中每一位的编号和数字 1011_2 的存放位置。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1		

(32 位宽)

由于字是在水平或垂直方向上书写的，用最左边或最右边表示大小带有不确定性，因此采用**最低有效位** (least significant bit) 表示最右边的一位（上图中的第 0 位），**最高有效位** (most significant bit) 表示最左边的一位（上图中的第 31 位）。

- ② 最低有效位：在 MIPS 字中最右边的一位。
- ② 最高有效位：在 MIPS 字中最左边的一位。

MIPS 的字有 32 位，可以表示 2^{32} 个不同的 32 位模式。很自然就可以使这些组合表示从 0 到 $2^{32} - 1$ ($4\,294\,967\,295_{10}$) 之间的数：

$$\begin{aligned} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 &= 0_{10} \\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 &= 1_{10} \\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 &= 2_{10} \\ \dots \\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 &= 4,294,967,293_{10} \\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 &= 4,294,967,294_{10} \\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 &= 4,294,967,295_{10} \end{aligned}$$

如下式，32 位的二进制数字也可以表示成每位的值乘以该位对应的 2 的幂次的形式（这里 x_i 表示数字 x 的第 i 位）：

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \dots + (x1 \times 2^1) + (x0 \times 2^0)$$

在稍后我们将看到，由于一些原因，这些正数被称为无符号数。

01 硬件/软件接口 二进制对人类来说不是自然的计数方法，我们有 10 个手指头，所以我们自然会采用十进制数。为什么计算机不使用十进制呢？事实上，第一台商用计算机确实提供了十进制算术。问题在于计算机仍然采用开关信号，所以一个十进制数将由几个二进制数来表示。事实证明十进制效率很低，所以后来的计算机都转向了二进制，只有在相对很少发生的 I/O 事件中才将数据转换成十进制。

需要注意的是，上式是二进制数的一般表示。实际上，数是由无穷多的位组成的，其中除了最右边的少数位以外其余大部分都是 0。正常情况下不用表示左边的 0。

硬件可以对这些二进制数进行加、减、乘、除操作。如果操作结果不能被最右端的硬件位所表示，那么就发生了溢出 (overflow)。如何处理溢出是由编程语言、操作系统和程序来决定的。

74

计算机程序对正数和负数都要进行计算，所以需要一种方法来区分正数和负数。显而易见的解决方案是增加一个独立的符号位，这种表示方法称为符号和幅值（sign and magnitude）表示法。

符号和幅值表示法有若干缺点。首先，符号位放在哪里不够明确，放在右边还是左边？早期的计算机对两种方法都尝试过。其次，因为不可能在计算时提前得知结果的符号，对于符号和幅值表示的数进行计算需要额外的一步来设置符号。最后，一个单独的符号位意味着在符号和幅值表示的数中不但有正零而且还有负零，这将给粗心的程序员带来问题。这些缺点导致这种表示方法很快就被放弃了。

在研究更具吸引力的替代方案时产生了这样一个问题，当我们试图用一个较小的数减去一个较大的数时，无符号数表示方法的结果将会是什么？答案是较小的数字将会从前面的0中借位，所有结果中前面的位都变成了一串1。

在没有其他明显更好选择的情况下，最终的解决方案是选择一种易于硬件实现的表达方式：前导位为0表示正数，前导位为1表示负数。这种常用的表示有符号二进制数的方法称为二进制补码（two's complement）。例如：

0000 0000 0000 0000 0000 0000 0000 0000 ₂	= 0 ₁₀
0000 0000 0000 0000 0000 0000 0001 ₂	= 1 ₁₀
0000 0000 0000 0000 0000 0000 0010 ₂	= 2 ₁₀
...	...
0111 1111 1111 1111 1111 1111 1111 1101 ₂	= 2,147,483,645 ₁₀
0111 1111 1111 1111 1111 1111 1111 1110 ₂	= 2,147,483,646 ₁₀
0111 1111 1111 1111 1111 1111 1111 1111 ₂	= 2,147,483,647 ₁₀
1000 0000 0000 0000 0000 0000 0000 0000 ₂	= -2,147,483,648 ₁₀
1000 0000 0000 0000 0000 0000 0000 0001 ₂	= -2,147,483,647 ₁₀
1000 0000 0000 0000 0000 0000 0000 0010 ₂	= -2,147,483,646 ₁₀
...	...
1111 1111 1111 1111 1111 1111 1111 1101 ₂	= -3 ₁₀
1111 1111 1111 1111 1111 1111 1111 1110 ₂	= -2 ₁₀
1111 1111 1111 1111 1111 1111 1111 1111 ₂	= -1 ₁₀

上面的数字中一半是正数，从0~2,147,483,647₁₀（2³¹-1），这些数字的表示方式与之前是一样的。紧接着的1000…0000₂表示最小的负数-2,147,483,648₁₀（-2³¹）。而后是按照绝对值递减的负数：从-2,147,483,647₁₀（1000…0001₂）到-1₁₀（1111…1111₂）。

二进制补码中的最小负数-2,147,483,648₁₀没有相应的正数与之对应。这种不平衡同样也会为粗心的程序员带来烦恼，但相比符号和幅值方法，该方法不会对程序员和硬件设计人员造成困扰。因此，现在所有计算机都采用二进制补码方法来表示有符号数。

采用二进制补码方法的优点在于所有负数的最高有效位都是1。硬件只需检测这一位就可以知道一个数是正数还是负数（这一位为0表示是正数）。因此，这个位通常叫作符号位。在理解了符号位之后，就可以使用2的幂次的方式来表示正的和负的32位数：

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \dots + (x1 \times 2^1) + (x0 \times 2^0)$$

符号位被-2³¹乘，其余的位仍按前面的方法计算。

01 例题·二进制到十进制的转换

下面这个用32位二进制补码表示的数对应的十进制数是多少？

1111 1111 1111 1111 1111 1111 1111 1100₂

01 答案

将数的位值代入上面的公式：

$$(1 \times (-2^{31})) + (1 \times 2^{30}) + (1 \times 2^{29}) + \dots + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) \\ = -2^{31} + 2^{30} + 2^{29} + \dots + 2^2 + 0 + 0$$

$$\begin{aligned}
 &= -2\ 147\ 483\ 648_{10} + 2\ 147\ 483\ 644_{10} \\
 &= -4_{10}^{\ominus}
 \end{aligned}$$

后面将给出从负数转换为正数的捷径。 □

就像无符号数的操作结果可能超过硬件允许的容量而发生溢出一样，对二进制补码数的操作也可能发生溢出。溢出发生在有限二进制数最左边的符号位与采用无穷多位表示该数时左边位的值不同的情况下（即符号位不正确）：该数是负数时符号位是 0，或该数是正数时符号位是 1。

01 硬件/软件接口 和算术运算一样，对取数指令来说有符号数和无符号数是有区别的。取回有符号数后需要使用符号位填充寄存器的所有剩余位，称为符号扩展，但其目的是在寄存器中放入数字正确的表示方式。取回无符号数只是简单地用 0 来填充数据左侧的剩余位，因为这种表示形式的数是没有符号的。

当把 32 位的字加载到 32 位的寄存器中时，上面的讨论是没有意义的，因为无符号数和有符号数的加载是完全一样的。MIPS 提供了两种字节加载的方法：一种是用于字节加载的 `lb` (`load byte`)，`lb` 将字节看作有符号数，使用符号扩展来填充寄存器的左侧 24 位；另一种是用于无符号整数加载的 `lbu` (`load byte unsigned`)。由于 C 程序几乎都是使用字节来表示字符，很少用来表示有符号短整数 (`short signed integers`)，所以实际中几乎所有字节加载都使用 `lbu`。

01 硬件/软件接口 与上面所讨论的数不同，存储器地址很自然地从 0 开始一直连续增加到最大的地址。换言之，负地址是没有意义的。因此，程序有时需要处理一些可以是正也可以是负的数，有时需要处理一些仅能是正的数。一些编程语言反映了这个区别。例如，C 语言将前者叫作整数 (`int`) 而后者叫作无符号整数 (`unsigned int`)。一些 C 编程风格的指导书甚至推荐用 `signed int` 来声明前一种数，以使区别更加明显。

我们来看两种处理二进制补码数的简单方法。第一种是对二进制补码数取反的快速方法。简单对每一位取反，0 变成 1，1 变成 0，然后对结果加 1。这种方法是基于这样的事实，一个数和它按位取反的结果相加，和一定是 $111\dots111_2$ ，即 -1 。因此 $x + \bar{x} = -1$ ，即 $x + \bar{x} + 1 = 0$ 或 $\bar{x} + 1 = -x$ 。（我们使用 \bar{x} 表示将 x 的每位取反）。

01 例题·求反的捷径

对 2_{10} 求反，然后通过对 -2_{10} 求反来对结果进行检查。

01 答案

$2_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2$ ，求反就是将这个数按位取反再加 1：

$$\begin{array}{r}
 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 \\
 + \hspace{10em} 1_2 \\
 \hline
 = \hspace{1em} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 \\
 = \hspace{1em} -2_{10}
 \end{array}$$

另一方面，将

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2$$

也按位取反再加 1：

⊖ 此公式从右往左第 3 项为 1×2^2 ，原书中为 1×2^1 ，原书有误。——译者注

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 \\
 + \hspace{10em} 1_2 \\
 \hline
 = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 \\
 = \quad 2_{10}
 \end{array}$$

77

□

第二种方法用于将一个用 n 位表示的二进制数转化成一个用多于 n 位表示的数。例如，在取数、存数、分支、加、小于则置位等指令中的立即数字段包含一个二进制补码表示的 16 位数，表示从 $-32\ 768_{10}$ (-2^{15}) 到 $32\ 767_{10}$ ($2^{15}-1$)。为了将这个立即数字段加到一个 32 位的寄存器，计算机必须将这个 16 位的数转换成数值上相等的 32 位的数。这种方法就是将原有的 16 位数简单复制到 32 位新数的低 16 位，其最高有效位（符号位）则以复制的方式填满新数的高 16 位。这种方法通常叫作符号扩展（sign extension）。

01 例题·符号扩展的方法

将 2_{10} 和 -2_{10} 从 16 位二进制数转换为 32 位二进制数。

01 答案

2_{10} 的16位二进制表示形式是

$$0000 \ 0000 \ 0000 \ 0010_2 = 2_{10}$$

将这个数转化成 32 位数的方法是：将最高有效位（0）复制 16 次放到 32 位字的左半部。右半部的 16 位保持原 16 位的值：

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$$

使用前面介绍的方法对 2 的 16 位二进制数求反。于是，

0000 0000 0000 0010,

变成

将该求反结果转换为 32 位数的方法就是将符号位复制 16 次放到 32 位字的左半部：

$$1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1110_2 = -2_{10}$$

这种方法之所以正确，是因为二进制补码表示的正数实际上在左侧有无限多个0，而负数在左侧有无限多个1。只是为了适应硬件的宽度，数的前导位被隐藏了，符号扩展只是简单地恢复了其中一部分。

78

小结

本节的主要内容是如何在给定的计算机字长中表示正整数和负整数。虽然各种表示方法都有各自的优缺点，但从 1965 年以来大多数计算机都采用了二进制补码方法。

01 精解 因为带符号十进制数没有长度的限制，所以常用“-”来表示负数。而在给定二进制或十六进制（见图 2-4）字长的情况下，可以将符号编码到位串中，因此通常不使用“+”和“-”来表示二进制或十六进制数。

01 小测验

下面这个 64 位二进制补码数对应的十进制数是多少？

- 1) -4_{10}
- 2) -8_{10}
- 3) -16_{10}
- 4) $18\ 446\ 744\ 073\ 709\ 551\ 609_{10}$

01 精解 二进制补码的得名来自下述规则：一个 n 位的数与它的相反数做无符号加法，结果是 $2n$ ，因此， x 的相反数 $-x$ 的二进制补码表示是 $2n - x$ 。

除了“二进制补码”和“符号和幅值”这两种表示法以外，第三种可选的表示法是所谓的“反码”(one's complement)。在反码中，一个数的相反数就是将这个数的每一位按位取反，0 变成 1，1 变成 0，这也是这种表示法名字的由来。在反码中 x 的相反数是 $2^n - x - 1$ 。与符号和幅值表示法相比，反码在某些方面是一个更好的解决方案，因此一些早期用于科学计算的计算机采用这种表示法。与补码相比，反码除了有 2 个零以外，其余都是相似的。其中正 0 是 $00\cdots00_2$ ，负 0 是 $11\cdots11_2$ 。绝对值最大的负数（即最小的负数）是 $10\cdots00_2$ ，它表示 $-2\ 147\ 483\ 647_{10}$ ，所以正数和负数的个数是平衡的。当采用反码时，加法器需要一个额外的步骤减去一个数来修正结果。因此，现在的计算机中补码方法占据了统治地位。

第 3 章将介绍一种浮点数的表示法。其中，最小的负数用 $00\cdots000_2$ 表示，最大的正数用 $11\cdots11_2$ 表示，0 一般用 $10\cdots00_2$ 表示。因为它通过将数加一个偏移使其具有非负的表示形式，所以称为偏移表示法 (biased notation)。

- ② 反码：使用 $10\cdots000_2$ 表示最小负数， $01\cdots11_2$ 表示最大正数，正数和负数的数量相同，但保留两个零，一个正零 ($00\cdots00_2$)，一个负零 ($11\cdots11_2$)。这种方法也用来表示按位求反，即 0 置为 1，1 置为 0。
- ③ 偏移表示法：最小的负数用 $00\cdots000_2$ 表示，最大的正数用 $11\cdots11_2$ 表示，0 一般用 $10\cdots00_2$ 表示，即通过将数加一个偏移使其具有非负的表示形式。

79

2.5 计算机中指令的表示

人操作计算机的方式与计算机看到指令的方式是不同的，现在我们就可以来解释其差别了。

指令在计算机内部是以若干或高或低的电信号的序列表示的，并且形式上和数的表示相同。实际上，指令的各部分都可看成一个独立的数，将这些数拼接在一起就形成了指令。

因为几乎所有的指令中都要用到寄存器，所以必须有一套规定，以将寄存器名字映射成数字。在 MIPS 汇编语言中，寄存器 \$s0 ~ \$s7 映射到寄存器 16 ~ 23，同时，寄存器 \$t0 ~ \$t7 映射到寄存器 8 ~ 15。因此，\$s0 表示寄存器 16，\$s1 表示寄存器 17，\$s2 表示寄存器 18……\$t0 表示寄存器 8，\$t1 表示寄存器 9，依次类推。在下面几节中，我们将介绍 32 个寄存器中其余寄存器的映射。

01 例题·将一条 MIPS 汇编语言指令翻译成一条机器指令

下面以 MIPS 汇编语言为例。对于符号表示为

add \$t0,\$s1,\$s2

的 MIPS 指令，首先给出其十进制数表示形式，接着给出其二进制数表示形式。

01 答案

其十进制表示为

0	17	18	8	0	32
---	----	----	---	---	----

机器指令分为若干字段 (field)。本例中第一个字段和最后一个字段 (0 和 32) 组合起来告诉 MIPS 计算机该指令要完成加法运算。第二个字段表示加法的第一个源操作数寄存器号 ($17 = \$s1$)，第三个字段表示加法的另一个源操作数寄存器号 ($18 = \$s2$)。第四个字段表示存放运算结果的目的寄存器号 ($8 = \$t0$)。第五个字段在这条指令中没有用到，故置为 0。这样，这条指令将寄存器 $\$s1$ 和寄存器 $\$s2$ 内容相加，并将和放在寄存器 $\$t0$ 中。

这条指令也可以表示成二进制的形式：

000000	10001	10010	01000	00000	100000
6位	5位	5位	5位	5位	6位

80

□

指令的布局形式叫作指令格式 (instruction format)。从位的数目可以看出，MIPS 指令占 32 位，与数据字的位数相等。为遵循简单源于规整的原则，所有 MIPS 指令都是 32 位长。

为了将它与汇编语言区分开来，把指令的数字形式称为机器语言 (machine language)，这样的指令序列叫作机器码 (machine code)。

- ② 指令格式：二进制数字段组成的指令表示形式。
- ② 机器语言：在计算机系统中用于交流的二进制表示形式。

为避免读写冗长乏味的二进制字串，可采用比二进制基数更大，但又易转化为二进制的表示形式来表示。由于几乎所有计算机的数据大小都是 4 的整数倍，因此十六进制 (hexadecimal) 表示形式变得很流行。16 是 2 的 4 次幂，因此可以很简单地通过将每 4 位二进制数替换为 1 位十六进制数来完成二进制到十六进制的转换，反之亦然。图 2-4 给出了十六进制和二进制之间的转化表。

- ② 十六进制：基数为 16 的数。

十六进制	二进制	十六进制	二进制	十六进制	二进制	十六进制	二进制
0_{16}	0000_2	4_{16}	0100_2	8_{16}	1000_2	c_{16}	1100_2
1_{16}	0001_2	5_{16}	0101_2	9_{16}	1001_2	d_{16}	1101_2
2_{16}	0010_2	6_{16}	0110_2	a_{16}	1010_2	e_{16}	1110_2
3_{16}	0011_2	7_{16}	0111_2	b_{16}	1011_2	f_{16}	1111_2

图 2-4 十六进制和二进制转换表。可以简单地把 1 位十六进制数替换为相应的 4 位二进制数，反之亦然。如果二进制数的位数不是 4 的整数倍，转化要从右往左进行

为了避免处理不同进制数时产生混淆，此处约定十进制数加下标 10，二进制数加下标 2，十六进制数加下标 16。（如果没有下标，那么默认为十进制。）顺便说明，C 和 Java 中用符号 $0xnnnn$ 来表示十六进制数。

01 例题·二进制和十六进制间的转换

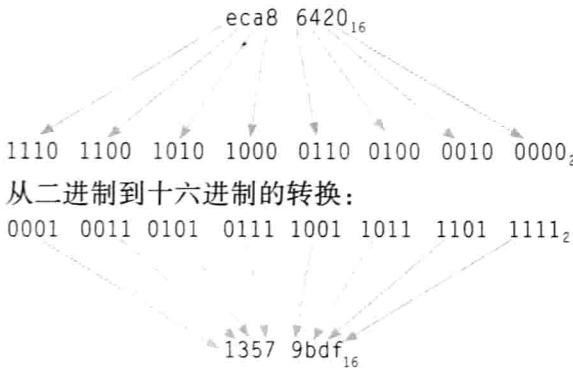
将下面的十六进制数转化成二进制数，二进制数转化成十六进制数：

$eca8\ 6420_{16}$

0001 0011 0101 0111 1001 1011 1101 1111₂

01 答案

按图 2-4 所示十六进制 - 二进制转换表查表得：



□

MIPS 字段

为了使讨论变得简单，给 MIPS 字段命名如下：

op	rs	rt	rd	shamt	funct
6位	5位	5位	5位	5位	6位

MIPS 指令中各字段名称及含义如下：

- op：指令的基本操作，通常称为操作码（opcode）。
- rs：第一个源操作数寄存器。
- rt：第二个源操作数寄存器。
- rd：用于存放操作结果的目的寄存器。
- shamt：位移量。（在 2.6 节中介绍移位指令和该术语，在此之前，指令都不使用这个字段，故此字段的内容为 0。）
- funct：功能。一般称为功能码（function code），用于指明 op 字段中操作的特定变式。

操作码：指令中用来表示操作和格式的字段。

当某条指令需要比上述字段更长的字段时，问题就会发生。例如，取字指令必须指定两个寄存器和一个常数。在上述格式中，如果地址使用其中的一个 5 位字段，那么取字指令的常数就被限制在 2^5 （即 32）之内。这个常数通常用来从数组或数据结构中选择元素，所以它常常比 32 大得多。5 位字段因太小而用处不大。

因此，既希望所有指令长度相同，又希望具有统一的指令格式，两者之间产生了冲突。这就引出了最后一条硬件设计原则。

设计原则 3：优秀的设计需要适宜的折中方案。

MIPS 设计者选择这样一种折中方案：保持所有的指令长度相同，但不同类型的指令采用不同的指令格式。例如，上述格式称为 R 型（用于寄存器）。另一种指令格式称为 I 型（用于立即数），立即数和数据传送指令用的就是这种格式。I 型的字段如下所示：

op	rs	rt	constant or address
6位	5位	5位	16位

16 位的地址字段意味着取字指令可以取相对于基址寄存器地址偏移 $\pm 2^{15}$ 或者 32 768 个字节 ($\pm 2^{13}$ 或者 8192 个字) 范围内的任意数据字。类似地，加立即数指令中常数也被限制不超过 $\pm 2^{15}$ 。可以看到在这种格式下，很难设置 32 个以上的寄存器，因为 rs 和 rt 字段都必须增加额外的位，这样就导致 32 位字长的指令很难满足要求。

我们来分析一下 2.3.1 节例子中的取字指令：

```
lw $t0,32($s3) # Temporary reg $t0 gets A[8]
```

这里，19（寄存器 \$s3）存放于 rs 字段，8（寄存器 \$t0）存放于 rt 字段，32 存放于 address 字段。注意，对于这条指令 rt 字段的意思已经改变：在一条取字指令中，rt 字段用于指明接收取数结果的目的寄存器。

虽然多种指令格式使硬件变得复杂，但是保持指令格式的类似性仍可降低复杂度。例如，R 型和 I 型格式的前 3 个字段长度相等，并且名称也一样；I 型格式的第四个字段和 R 型后 3 个字段长度之和相等。

也许你会想到，指令格式可以由第一个字段的值来区分：每种格式在第一个字段（op）占有不同的值区间，以便让计算机硬件知道指令后半部分是三字段（R 型）还是一字段（I 型）。图 2-5 给出了到目前为止已使用过的 MIPS 指令的每个字段的值。

指令	格式	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{10}	n. a.
sub (subtract)	R	0	reg	reg	reg	0	34_{10}	n. a.
add immediate	I	8_{10}	reg	reg	n. a.	n. a.	n. a.	常数
lw (load word)	I	35_{10}	reg	reg	n. a.	n. a.	n. a.	address
sw (store word)	I	43_{10}	reg	reg	n. a.	n. a.	n. a.	address

图 2-5 MIPS 指令编码。在上表中，“reg”代表寄存器的标号（从 0 ~ 31），“address”表示 16 位地址，“n. a.”（not applicable）表示这个字段在该指令格式中不出现。注意，add 和 sub 指令具有相同的 op 字段值，硬件根据 funct 字段的值来决定所进行的操作：add (32) 或 subtract (34)

83

01 例题·将 MIPS 汇编语言翻译成机器语言

现在可以给出一个例子来描述从程序员所编程序到机器执行指令的整个转换过程。如果数组 A 的基址存放在 \$t1 中，h 存放在 \$s2 中，下面的 C 赋值语句：

```
A[300] = h + A[300];
```

被编译成如下汇编语言：

```
lw $t0,1200($t1) # Temporary reg $t0 gets A[300]
add $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
sw $t0,1200($t1) # Stores h + A[300] back into A[300]
```

这三条 MIPS 指令的机器语言代码是什么？

01 答案

为方便起见，先使用十进制数表示机器语言指令。从图 2-5 中可以确定这三条机器语言指令：

op	rs	rt	rd	address/shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

lw 指令的第一个字段（op）值为 35（见图 2-5）。在第二个字段（rs）中指定基址寄存器 9（\$t1），在第三个字段（rt）中指定目的寄存器 8（\$t0）。在最后一个字段 address 中存放用于指定 A [300] 的偏移量 ($1200 = 300 \times 4$)。

下一条 add 指令由第一个字段（op）值 0 和最后一个字段（funct）值 32 共同确定。第二、三、四字段中的三个寄存器（18、8 和 8）分别对应 \$s2、\$t0 和 \$t0。

`sw` 指令由第一个字段的 43 识别。这条指令的其他部分和 `lw` 指令完全一样。

与上述十进制形式对应的二进制机器指令如下所示（十进制数 1 200 用二进制表示为 0000 0100 1011 0000₂）：

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

注意，第一条指令和最后一条指令的二进制表示非常相似，唯一不同的是从左边数第 3 位。

□

01 硬件/软件接口 定长指令的需求与设置尽可能多的寄存器的需求矛盾。寄存器数量的任何增长都需要在指令格式中的各个寄存器字段至少增加 1 位。综合考虑这些限制和越小越快的设计原则，当今的大多数指令系统中有 16 个或 32 个通用寄存器。

图 2-6 归纳了本节讲述的 MIPS 机器语言。正如将在第 4 章中讲述的那样，相关指令在二进制表示上的相似性可简化硬件设计。这种相似性也是 MIPS 体系结构规整性的又一佐证。

MIPS 机器语言

名字	格式	举例						注释
add	R	0	18	19	17	0	32	add \$s1, \$s2, \$s3
sub	R	0	18	19	17	0	34	sub \$s1, \$s2, \$s3
addi	I	8	18	17	100			addi \$s1, \$s2, 100
lw	I	35	18	17	100			lw \$s1, 100(\$s2)
sw	I	43	18	17	100			sw \$s1, 100(\$s2)
字段宽度		6 位	5 位	5 位	5 位	5 位	6 位	所有 MIPS 指令均为 32 位
R 型	R	op	rs	rt	rd	shamt	funct	算术指令格式
I 型	I	op	rs	rt	address			数据传送指令格式

图 2-6 2.5 节展示的 MIPS 体系结构。到目前为止所见到的 MIPS 指令都是 R 型和 I 型指令。所有指令的前 16 位都是相同的，都包含给出基本操作的 op 字段；给出第一源操作数的 rs 字段；给出第二源操作数的 rt 字段（取字指令除外，在取字指令中用于指定目的寄存器）。R 型指令将最后 16 位划分为 3 个字段：rd 字段指明目的寄存器；shamt 字段将在 2.6 节中介绍；funct 字段指明 R 型指令的特定辅助操作。I 型指令将最后 16 位合并为一个 address 字段

85

01 重点 当今计算机基于以下两个重要准则构建：

- 1) 指令用数的形式表示。
- 2) 和数据一样，程序存储在存储器中，并且可以读写。

这些原则引出存储程序（stored-program）的概念，这一发明释放了计算机的巨大潜力。图 2-7 显示了存储程序的强大功能。特别地，存储器可以存放编辑器程序的源代码、与之对应的编译后的机器码、编译后的程序需要使用的文本，甚至用于生成机器码的编译器。

指令表示成数的好处就是程序可以被当成二进制数的文件发行。商业上的意义就是计算机可以延用那些指令集兼容的现成软件。这种“二进制兼容”使得工业界围绕着几种指令集体系结构形成联盟。

86

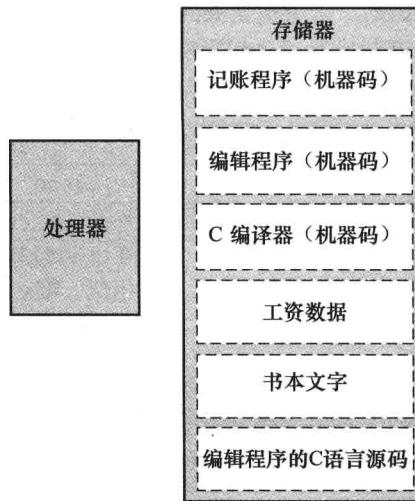


图 2-7 存储程序概念。各类存储程序允许将一台用于记账的计算机转眼间变成一台可以帮助作者写书的计算机。只要将程序和数据加载到存储器中并告诉计算机从给定的存储器地址开始执行程序即可。将指令和数据以相同的方式处理，极大地简化了计算机系统的存储器硬件和软件。尤其是用于数据的存储技术同样也适用于程序，如编译器，它能够将那些用易于人类使用的符号编写的代码翻译成机器能理解的代码。

01 小测验

下面的图表代表的是哪条 MIPS 指令？

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. sub \$t0, \$t1, \$t2
2. add \$t2, \$t0, \$t1
3. sub \$t2, \$t1, \$t0
4. sub \$t2, \$t0, \$t1

2.6 逻辑操作

“正相反，”叮当弟接着说，“如果那是真的，那它就可能是真的；如果那曾经是
真的，它就是真过；但是既然现在它不是真的，那么现在它就是假的。这就是逻辑。”

——Lewis Carroll, 《爱丽丝漫游仙境》, 1865

虽然早期的计算机仅对整字进行操作，但人们很快就发现，对字中由若干位组成的字段甚至对单个位进行操作是很有用的。例如，考查字里面每个由 8 位组成的字符（见 2.9 节）。于是，编程语言和指令集体系结构中增加了一些指令，用于简化对字中若干位进行打包或者拆包的操作。这些指令被称为逻辑操作。图 2-8 给出了 C、Java 和 MIPS 中的逻辑操作。

逻辑操作	C 操作符	Java 操作符	MIPS 指令
左移	<<	<<	sll
右移	>>	>>>	srl
按位与	&	&	and, andi
按位或			or, ori
按位取反	~	~	nor

图 2-8 C 和 Java 的逻辑操作符及相应的 MIPS 指令。MIPS 使用一个操作数为 0 的 NOR 指令实现取反操作

第一类逻辑操作称为移位 (shift)。它们将一个字里面的所有位都向左或向右移动，并在空出来的位上填充 0。例如，假设寄存器 \$s0 中的数据是：

0000 0000 0000 0000 0000 0000 1001₂ = 9₁₀

一条左移 4 位的指令执行后，得到的新值是：

0000 0000 0000 0000 0000 1001 0000₂ = 144₁₀

87

与左移相对应的是右移。左移和右移这两条指令在 MIPS 中的的确切名字是逻辑左移 (sll) 和逻辑右移 (srl)。下面的指令完成的就是上述操作，假设源操作数在 \$s0 中，结果存储到 \$t2 中：

sll \$t2,\$s0,4 # reg \$t2 = reg \$s0 << 4 bits

前面介绍 R 型指令格式时没有解释 shamt 字段，它在移位指令中被用于表示移位量 (shift amount)。因此，上述指令对应的机器语言是：

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

指令 sll 的编码在 op 字段和 funct 字段都为 0，rd 为 10（寄存器 \$t2），rt 为 16（寄存器 \$s0），shamt 为 4，rs 字段没有使用，被置为 0。

逻辑左移还有额外的好处，就是左移 i 位就相当于乘以 2^i ，这就像十进制数左移 i 位相当于乘以 10^i 。例如，上面的 sll 指令左移了 4 位，就相当于乘以 2^4 （即 16）。所以，原二进制数表示的值是 9，而 $9 \times 16 = 144$ ，恰好就是移位后的结果。

第二类有用的操作是按位与 (AND)。该操作仅当两个操作位均为 1 时结果才为 1。例如，如果寄存器 \$t2 的值为：

0000 0000 0000 0000 0000 1101 1100 0000₂

寄存器 \$t1 的值为：

0000 0000 0000 0000 0011 1100 0000 0000₂

那么，在执行下面的 MIPS 指令后

and \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 & reg \$t2

\$t0 中的值将是：

0000 0000 0000 0000 0000 1100 0000 0000₂

AND 提供了一种将源操作数中某些位置为 0 的能力，前提是另一个操作数中对应位为 0。后一个操作数传统上被称为掩码 (mask)，寓意其可“隐藏”某些位。

88

与 AND 对偶的操作是按位或 (OR)。该操作在两个操作位中任意一位为 1 时结果就为 1。为详细说明，仍假设 \$t1 和 \$t2 中的值都和上面的例子一样，那么下述 MIPS 指令

or \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 | reg \$t2

执行后 \$t0 的值是：

0000 0000 0000 0000 0011 1101 1100 0000₂

最后一类逻辑操作是按位取反 (NOT)。该操作仅有一个操作数，将 1 变成 0，0 变成 1。使用前面的符号，它可用来计算 \bar{x} 。为了保持三操作数的格式，MIPS 的设计者引入或非 NOR (NOT OR) 指令来取代 NOT。如果一个操作数是 0，那么对另一个操作数而言，结果就等价于 NOT: A NOR 0 = NOT (A OR 0) = NOT (A)。

如果寄存器 \$t1 中的值与上例保持不变，寄存器 \$t3 中的值是 0，那么下面 MIPS 指令

nor \$t0,\$t1,\$t3 # reg \$t0 = ~ (reg \$t1 | reg \$t3)

在寄存器 \$t0 中的执行结果是：

1111 1111 1111 1111 1100 0011 1111 1111₂

- ② 按位与：按位进行与操作，仅当两个操作位均为 1 时结果才为 1。
- ③ 按位或：按位进行或操作，当两个操作位中任意一位为 1 时结果就为 1。
- ④ 按位取反：按位进行非操作，仅有一个操作数，将 1 变成 0，0 变成 1。
- ⑤ 或非：按位先或后非操作，仅当两个操作位均为 0 时结果才为 1。

图 2-8 显示了 C 和 Java 的操作符与 MIPS 指令之间的关系。像在算术运算中一样，常数在 AND 和 OR 这些逻辑运算里也是很有用的，因此 MIPS 也提供了立即数与 (andi) 和立即数或 (ori) 指令。常数在 NOR 中出现得很少，因为 NOR 主要功能就是将单操作数按位取反，因此，MIPS 指令集体系结构没有设计支持 NOR 立即数的版本。

- 01 精解** MIPS 指令全集也包括异或 (XOR)，当两个操作数对应位不同时置 1，相同时置 0。C 语言允许在字内定义由若干位组成的一个或多个字段，并将其作为对象包装在一个字内，以适应如 I/O 设备等的外部接口需求。所有字段必须放在一个单字之中，并采用无符号整数。C 编译器使用 MIPS 的下列逻辑指令插入和提取字段：and、or、sll 以及 srl。
- 01 精解** 在与立即数进行逻辑与操作和逻辑或操作时，立即数的高 16 位补 0 后形成 32 位常数进行计算，而与立即数做加法运算时，将立即数进行符号扩展。

01 小测验

下面哪个操作可以将字中的一部分分离出来？

1. AND
2. 左移后再进行右移

89

2.7 决策指令

自动化计算机的实用性取决于重复使用给定指令序列的可能性，重复的次数取决于计算的结果……这一选择可以根据数的符号来决定（计算机认为 0 是正数）。因此，我们引入一条“指令”（条件转移“指令”），它根据给定数的符号从两条路径中选择正确的一条来执行。

——Burks、Goldstine、von Neumann, 1947

计算机与简单计算器的区别在于决策能力。根据输入数据和计算过程中产生的值，它可以执行不同的指令。程序语言通常使用 if 语句描述决策，有时也使用 go to 语句和标签。MIPS 汇编语言中有两条类似 if 和 go to 语句功能的指令。第一条是

beq register1, register2, L1

该指令表示：如果 register1 和 register2 中的数值相等，则转到标签为 L1 的语句执行。助记符 beq 代表如果相等则分支 (branch if equal)。

第二条指令是

bne register1, register2, L1

该指令表示：如果 register1 和 register2 中的数值不相等，则转到标签为 L1 的语句执行。助记符 bne 代表如果不相等则分支 (branch if not equal)。这两条指令传统上称为条件分支 (conditional branch) 指令。

- ② 条件分支：该指令先比较两个值，然后根据比较的结果决定是否从程序中的一个新地址开始执行指令序列。

01 例题·将 if-then-else 语句编译成条件分支指令

在下面这段代码中，`f`、`g`、`h`、`i`、`j` 都是变量，设该 5 个变量依次对应于从 `$s0` 到 `$s4` 的寄存器，求这条 C 语言 if 语句编译后形成的 MIPS 代码。

```
if (i == j) f = g + h; else f = g - h;
```

01 答案

图 2-9 是 MIPS 代码执行过程的流程图。第一个表达式比较 `i` 和 `j` 是否相等，需要一条 `beq` 指令。通常，通过测试分支的相反条件来跳过 if 语句后面的 `then` 部分，代码的效率会更高（标签 `Else` 将在后面定义）所以我们使用 `bne` 指令：

```
bne $s3,$s4,Else    # go to Else if i != j
```

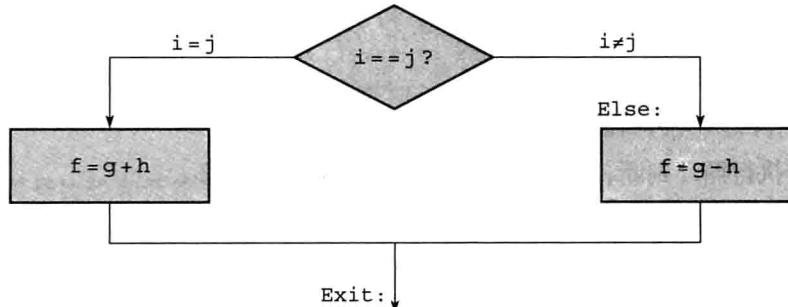


图 2-9 上述 if 语句的程序流程图。左边方框对应 if 语句的 then 部分，右边方框对应 if 语句的 else 部分

下一个赋值语句执行一个单操作，如果所有的操作数都分配给寄存器，那么它只是一条指令：

```
add $s0,$s1,$s2    # f = g + h (skipped if i ≠ j)
```

在 if 语句的结尾部分，需要引入另一种分支指令，通常叫作无条件分支指令（unconditional branch）。当遇到这种指令时，程序必须分支。为了区分条件分支和无条件分支，MIPS 将无条件分支指令命名为 `jump`，简写成 `j`（标签 `Exit` 将在后面定义）。

```
j Exit      # go to Exit
```

if 语句中 else 部分的赋值语句也可编译成一条指令。我们只需将标签 `Else` 加在这条指令前、标签 `Exit` 加在该条指令后面，表示 if-then-else 编译的代码结束：

```
Else:sub $s0,$s1,$s2    # f = g - h (skipped if i = j)
Exit:
```

□

91

注意，就像汇编器完成存数/取数指令的数据地址计算一样，它也完成分支指令的地址计算，这使得编译器和汇编语言程序员摆脱了乏味的地址计算任务（参见 2.12 节）。

01 硬件/软件接口

编译器经常创建一些在编程语言中没出现过的分支和标签。避免显式地编写这些标签和分支是使用高级编程语言的好处之一，也是其编码速度快的一个原因。

2.7.1 循环

无论是在二选一的 if 语句中，还是在迭代计算的循环语句中，决策都起着重要作用。但这两种情况下，关于决策的汇编语言指令是相同的。

01 例题·编译下面 C 语言 while 循环语句

下面是用 C 语言编写的传统循环程序：

```
while (save[i] == k)
    i += 1;
```

假设 i 和 k 存放在寄存器 $\$s3$ 和 $\$s5$ 中，数组 $save$ 的基址存放在寄存器 $\$s6$ 中。求这段 C 程序对应的 MIPS 汇编代码。

01 答案

第一步需要将 $save[i]$ 读入一个临时寄存器中。在读入之前，需要计算它的地址。在将 i 加到 $save$ 数组基址以形成访存地址前，由于系统按照字节寻址的缘故，先要将 i 乘以 4。幸运的是，我们可以使用逻辑左移指令实现这一乘法，因为左移 2 位等价于乘 4（见 2.6 节）。需要在该指令前增加一个标签 Loop，以便在循环末端能够跳回该指令。

```
Loop: sll $t1,$s3,2      # Temp reg $t1 = i * 4
```

为了得到 $save[i]$ 的地址，需要将 $\$t1$ 和 $\$s6$ 中 $save$ 的基址相加：

```
add $t1,$t1,$s6      # $t1 = address of save[i]
```

现在可用该地址将 $save[i]$ 读入一个临时寄存器中：

```
lw $t0,0($t1)      # Temp reg $t0 = save[i]
```

下一条指令执行循环判断，如果 $save[i] \neq k$ 则退出循环：

92

再下一条指令将 i 加 1：

```
addi $s3,$s3,1      # i = i + 1
```

在循环的末尾，程序跳转到循环的开始。随后增加了一个 Exit 标签，这样就完成了全部编译：

```
j      Loop      # go to Loop
Exit:
```

（见练习题中对该指令序列的优化。）

□

01 硬件/软件接口 以分支指令结束的这类指令序列对编译非常重要，因此它们有对应的专用术语：基本块。基本块（basic block）是没有分支（可能出现在末尾者除外）并且没有分支目标/分支标签（可能出现在开始者除外）的指令序列。编译最初阶段的任务之一就是将程序分解为若干基本块。

② 基本块：没有分支（可能出现在末尾者除外）并且没有分支目标/分支标签（可能出现在开始者除外）的指令序列。

93

最常见的判断语句可能是相等或不等，但有时判断一个变量是否小于另一个变量也非常有用。例如，for 循环就需要判断索引变量是否小于 0。在 MIPS 汇编语言中提供了一条指令来实现这种比较，该指令在比较两个寄存器内容之后，若第一个寄存器小于第二个寄存器，则将第三个寄存器设置为 1，否则设置为 0。该指令称为小于则置位（set on less than），即 slt 。例如，

```
slt      $t0, $s3, $s4      # $t0 = 1 if $s3 < $s4
```

表示当寄存器 $\$s3$ 的值小于寄存器 $\$s4$ 的值时，寄存器 $\$t0$ 被置为 1，否则寄存器 $\$t0$ 被置为 0。

在比较中经常使用常数操作数，所以有立即数版本的小于则置位指令。例如，为了测试寄存器 $\$s2$ 的值是否小于常数 10，可以使用如下指令：

```
slti     $t0,$s2,10      # $t0 = 1 if $s2 < 10
```

01 硬件/软件接口 MIPS 编译器使用 slt 、 $slti$ 、 beq 、 bne 和固定值 0（总是可以通过读

取寄存器 \$zero 来获得) 来创建所有的比较条件：相等、不等、小于、小于或等于、大于、大于或等于。

遵循冯·诺伊曼关于“设备”简单性的原则，MIPS 体系结构没有提供“小于则分支”指令，因为这种指令过于复杂，它会延长时钟周期时间，或增加平均执行每条指令的周期数(CPI)。两条更快的指令更加有用。

01 硬件/软件接口 比较指令应该具有分清有符号数和无符号数的能力。有时候二进制数最高位为 1 的数代表一个负数，它当然应该小于所有最高有效位为 0 的正数。另一方面，如果是无符号数，最高有效位为 1 的数将大于所有最高有效位为 0 的数。(我们将很快看到最高有效位具有双重意义在减少数组边界检查开销中所带来的优点。)

MIPS 为这两种情况提供两个版本的小于则置位指令。slt (set on less than) 和 slti (set on less than immediate) 指令用于处理有符号整数，而 sltu (set on less than unsigned) 和 sltiu (set on less than immediate unsigned) 指令则用于处理无符号整数。

01 例题·有符号比较和无符号比较的对比

假设寄存器 \$s0 中的二进制数为

1111 1111 1111 1111 1111 1111 1111 1111₂

而寄存器 \$s1 中的二进制数为

0000 0000 0000 0000 0000 0000 0000 0001₂

在执行以下两条指令后，寄存器 \$t0 和 \$t1 中的值分别是多少？

```
slt      $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

01 答案

如果有符号数，那么寄存器 \$s0 中的值为 -1_{10} ，寄存器 \$s1 中的值为 1_{10} ；如果是无符号数，那么寄存器 \$s0 中的值为 $4\ 294\ 967\ 295_{10}$ ，寄存器 \$s1 中的值仍为 1_{10} 。因此，寄存器 \$t0 中的值为 1，因为 $-1_{10} < 1_{10}$ ；寄存器 \$t1 中的值为 0，因为 $4\ 294\ 967\ 295_{10} > 1_{10}$ 。 □

将有符号数作为无符号数来处理，是一种检验 $0 \leq x < y$ 的低开销方法，常用于检查数组的下标是否越界。问题的关键是负数在二进制补码表示法中看起来像是无符号表示法中一个很大的数，因为在无符号数中最高有效位是符号位，而有符号数中最高有效位是具有最大权重的位。所以使用无符号比较 $x < y$ ，在检查 x 是否小于 y 的同时，也检查了 x 是否为一个负数。

01 例题·边界检查的简便方法

利用这个方法可以降低检验下标是否越界的开销：如果 $\$s1 \geq \$t2$ 或者 $\$s1$ 是负数则跳转到 IndexOutOfBounds。

01 答案

检查代码仅使用一条 sltu 指令即可同时进行两种检查：

```
sltu $t0,$s1,$t2 # $t0=0 if $s1>=length or $s1<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

□

2.7.2 case/switch 语句

大多数程序设计语言中都包括 case 或 switch 语句，使得程序员可以根据某个变量的值选择

不同分支之一。实现 switch 语句的最简单方法是借助一系列的条件判断，将 switch 语句转化为 if-then-else 语句嵌套。

有时候另一种更有效的方法是将多个指令序列分支的地址编码为一张表，即转移地址表 (jump address table) 或转移表 (jump table)，这样程序只需索引该表即可跳转到恰当的指令序列。转移地址表是一个由代码中标签所对应地址构成的数组。程序需要跳转的时候首先将转移地址表中适当的项加载到寄存器中，然后使用寄存器中的地址值进行跳转。为了支持这种情况，像 MIPS 这样的计算机提供了寄存器跳转指令 jr (jump register)，用来无条件地跳转到寄存器指定的地址。该指令将在下一节中介绍。
95

② 转移地址表：又称作转移表 (jump table)，指包含不同指令序列地址的表。

01 硬件/软件接口 虽然在 C 或 Java 这样的编程语言中有许多决策和循环语句，但是在指令集这一层次实现其功能的基本语句是条件分支。

01 精解 如果你曾经听说过延迟转移（将在第 4 章中介绍），那么不必对此表示担心：MIPS 汇编器会使其对汇编语言程序员不可见。

01 小测验

I. C 语言中有很多决策和循环语句，但是在 MIPS 中却很少。下述各句子有没有阐明这种不均衡？为什么？

1. 更多的决策语句使得代码更容易被阅读和理解。
2. 更少的决策语句简化了负责执行的底层工作。
3. 更多的决策语句意味着更少的代码量，这节约了编程的时间。
4. 更多的决策语句意味着更少的代码量，这意味着执行更少的操作。

II. 为什么 C 语言提供了两种与操作 (& 和 &&) 和两种或操作 (| 和 ||)，而 MIPS 没有提供呢？

1. 逻辑操作 AND 和 OR 实现 & 和 |，而条件分支实现 && 和 ||。
2. 上面的描述说反了：&& 和 || 对应于逻辑操作，而 & 和 | 对应于条件分支。
3. 它们是冗余的，并且是一回事。&& 和 || 都是简单继承于 C 程序设计语言的前身：B 语言。

2.8 计算机硬件对过程的支持

过程 (procedure) 或函数是程序员进行结构化编程的工具，两者均有助于提高程序的可理解性和代码的可重用性。过程允许程序员每次只需将精力集中在任务的一部分，由于参数能传递数值并返回结果，因此参数承担过程与其他程序、数据之间接口的角色。2.15 节描述了 Java 语言中过程的等价表示方法，但 Java 与 C 语言对计算机的要求完全相同。过程是软件中实现抽象的一种方法。

② 过程：根据提供的参数执行一定任务的存储的子程序。
96

你可以将过程想象成一个侦探，他离开时带着一项神秘的计划，为了完成该计划，需要获得资源、执行任务并隐匿行踪，最后带着预期的结果返回起点。一旦任务完成将不再对系统产生任何其他干扰。更重要的是，侦探是在“需要知道”的基础上工作的，所以侦探不需对雇主做任何假定。

同样，在过程运行中，程序必须遵循以下 6 个步骤：

- 1) 将参数放在过程可以访问的位置。
- 2) 将控制转交给过程。
- 3) 获得过程所需的存储资源。
- 4) 执行需要的任务。
- 5) 将结果的值放在调用程序可以访问的位置。
- 6) 将控制返回初始点，因为一个过程可能由一个程序中的多个点调用。

如上所述，寄存器是计算机中保存数据最快的位置，所以我们希望尽可能多地使用寄存器。MIPS 软件在为过程调用分配 32 个寄存器时遵循以下约定：

- \$a0 ~ \$a3：用于传递参数的 4 个参数寄存器。
- \$v0 ~ \$v1：用于返回值的两个值寄存器。
- \$ra：用于返回起始点的返回地址寄存器。

除了分配这些寄存器之外，MIPS 汇编语言还包括一条过程调用指令：跳转到某个地址的同时将下一条指令的地址保存在寄存器 \$ra 中。这条跳转和链接指令（jump-and-link instruction）的格式为：

```
jal ProcedureAddress
```

指令中的链接部分表示指向调用点的地址或链接，以允许过程返回到合适的地址。存储在寄存器 \$ra (31 号寄存器) 中的链接部分称为返回地址 (return address)。返回地址是必需的，因为同一过程可能在程序的不同部分调用。

- ⌚ 跳转和链接指令：跳转到某个地址的同时将下一条指令的地址保存到寄存器 \$ra 中的指令。
- ⌚ 返回地址：指向调用点的链接，使过程可以返回到合适的地址，在 MIPS 中它存储在寄存器 \$ra 中。

为了支持这种情况，类似 MIPS 的计算机使用了寄存器跳转 (jump register) 指令 jr，用于 case 语句，表示无条件跳转到寄存器所指定的地址：

```
jr    $ra
```

寄存器跳转指令跳转到存储在 \$ra 寄存器中的地址——这正是我们所希望的。因此，调用程序或称为调用者 (caller)，将参数值放在 \$a0 ~ \$a3，然后使用 jal X 跳转到过程 X (有时称为被调用者 (callee))。被调用者执行运算，将结果放在 \$v0 和 \$v1，然后使用 jr \$ra 指令将控制返回给调用者。

在存储程序概念中，使用一个寄存器来保存当前运行的指令地址是绝对必要的。尽管这个寄存器更为合理的名字可能应该是指令地址寄存器 (instruction address register)，但是出于历史原因，这个寄存器通常称为程序计数器 (program counter)，在 MIPS 体系结构中缩写为 PC。jal 指令实际上将 PC + 4 保存在寄存器 \$ra 中，从而将链接指向下一条指令，为过程返回做好准备。

- ⌚ 调用者：调用一个过程并为过程提供必要参数值的程序。
- ⌚ 被调用者：根据调用者提供的参数执行一系列存储的指令，然后将控制权返回调用者的程序。
- ⌚ 程序计数器 (PC)：包含在程序中正在被执行指令地址的寄存器。

2.8.1 使用更多的寄存器

假设对于一个过程，编译器需要使用多于 4 个参数寄存器和两个返回值寄存器。由于在任务完成后必须消除踪迹，因此调用者使用的任何寄存器都必须恢复到过程调用前所存储的值。这种情况可以看成是需要将寄存器换出到存储器的一个例子，如“硬件/软件接口”部分所提到的那样。

换出寄存器的最理想的数据结构是 **栈**（stack）——一种后进先出的队列。栈需要一个指针指向栈中最新分配的地址，以指示下一个过程放置换出寄存器的位置，或是寄存器旧值的存放位置。在每次寄存器进行保存或恢复时，**栈指针**（stack pointer）以字为单位进行调整。MIPS 软件为栈指针准备了第 29 号寄存器，并将其命名为 \$sp。由于栈的应用十分广泛，因此向栈传递数据或从栈中取数都有专用术语：将数据放入栈中称为 **压栈**（push），从栈中移除数据称为 **出栈**（pop）。

- ② **栈**：被组织成后进先出队列形式并用于寄存器换出的数据结构。
- ② **栈指针**：指示栈中最近分配的地址的值，它指示寄存器被换出的位置，或寄存器旧值的存放位置。在 MIPS 中，栈指针是寄存器 \$sp。
- ② **压栈**：向栈中增加元素。
- ② **出栈**：从栈中移除元素。

按照历史惯例，栈“增长”是按照地址从高到低的顺序进行的。这意味着将数据压栈时，栈指针值减小；而数据出栈时，栈长度缩短，栈指针增大。

01 例题·编译一个不调用其他过程的 C 过程

将 2.2 节的例子转化为一个 C 过程：

```
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

编译后的 MIPS 汇编代码是什么呢？

01 答案

参数变量 g、h、i 和 j 对应参数寄存器 \$a0、\$a1、\$a2 和 \$a3，f 对应 \$s0。编译后的程序是以如下标号开始的过程：

`leaf_example:`

下一步是保存过程中使用的寄存器。过程实体中的 C 赋值语句与 2.2 节的例子相同，使用了两个临时寄存器。因此，需要保存三个寄存器：\$s0、\$t0 和 \$t1。我们将旧值“压栈”，也就是在栈中建立三个字的空间，并将数据存入：

```
addi $sp, $sp, -12    # adjust stack to make room for 3 items
sw    $t1, 8($sp)     # save register $t1 for use afterwards
sw    $t0, 4($sp)     # save register $t0 for use afterwards
sw    $s0, 0($sp)     # save register $s0 for use afterwards
```

图 2-10 给出了在过程调用之前、之中和之后的栈。

接着的三条语句对应过程实体，与 2.2 节的例子相同：

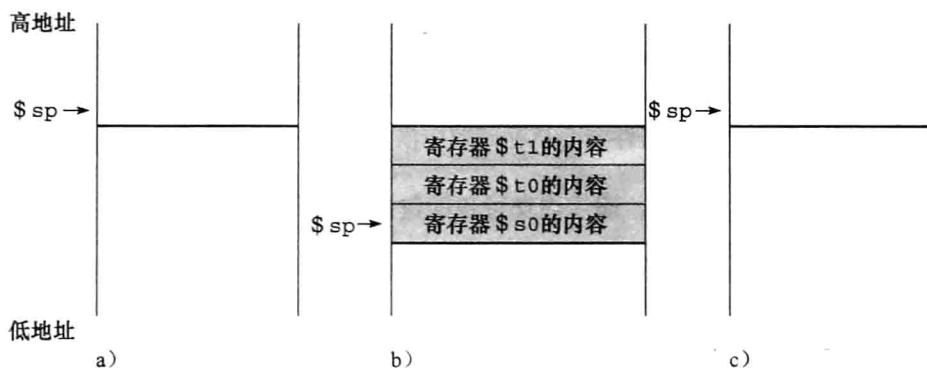


图 2-10 在过程调用之前 (a)、之中 (b) 和之后 (c)，栈指针以及栈的值。栈指针总是指向栈顶，或者说是图中栈的最后一个字

```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

为了返回 f 的值，我们将它复制到一个返回值寄存器中：

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

在返回前，我们通过从栈中“弹出”数据的方式恢复寄存器的三个旧值：

```
lw $s0, 0($sp) # restore register $s0 for caller
lw $t0, 4($sp) # restore register $t0 for caller
lw $t1, 8($sp) # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items
```

过程末尾处根据跳转寄存器中的返回地址跳转：

```
jr $ra # jump back to calling routine
```

□

前面的例子曾经使用了临时寄存器，并假设它们的旧值必须保存和恢复。为了避免保存和恢复一个其值未被使用过的寄存器（通常是临时寄存器），MIPS 软件将 18 个寄存器分为两组：

- \$t0 ~ \$t9：10 个临时寄存器，在过程调用中不必被调用者（被调用的过程）保存。
- \$s0 ~ \$s7：8 个保留寄存器，在过程调用中必须被保存（一旦被使用，由被调用者保存和恢复）。

99

这一简单约定减少了寄存器换出。在上面的例子中，因为调用者不希望在过程调用时保留寄存器 \$t0 和 \$t1，所以我们可以去掉有关两次保存和两次载入的代码。我们始终需要保存和恢复 \$s0，因为被调用者必须假设调用者需要该值。

2.8.2 嵌套过程

不调用其他过程的过程称为叶过程 (leaf procedure)。如果所有过程都是叶过程，那么情况就很简单，但实际并非如此。就像一个侦探，其任务的一部分是雇用其他侦探，被雇用的侦探进而雇用更多的侦探，某个过程调用其他过程也是这样。更进一步的是，递归过程甚至调用的是自身的“克隆”。就像在过程中使用寄存器需要十分小心一样，在调用非叶过程时需要更加小心。

例如，假设主程序将参数 3 存入寄存器 \$a0，然后使用 jal A 调用过程 A。再假设过程 A 通过 jal B 调用过程 B，参数为 7，同样存入 \$a0。由于 A 尚未结束任务，所以在寄存器 \$a0 的使用上存在冲突。同样，在寄存器 \$ra 保存的返回地址上也存在冲突，因为它现在保存着 B 的返回地址。除非我们采取措施阻止这类问题发生，否则这个冲突将导致过程 A 无法返回其调用者。

一个解决方法是将其他所有必须保留的寄存器压栈，就像将保存寄存器压栈一样。调用者将所有调用后还需要的参数寄存器（\$a0 ~ \$a3）或临时寄存器（\$t0 ~ \$t9）压栈。被调用者将返回地址寄存器 \$ra 和被调用者使用的保存寄存器（\$s0 ~ \$s7）都压栈。栈指针 \$sp 随着栈中寄存器个数调整。到返回时，寄存器会从存储器中恢复，栈指针也随着重新调整。

100

01 例题·编译一个递归 C 过程，演示嵌套过程的链接

下面是一个计算阶乘的递归过程：

```
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n - 1));
}
```

该过程的 MIPS 汇编代码是怎样的呢？

01 答案

参变量 n 对应参数寄存器 \$a0。编译后的程序以过程标签开始，然后在栈中保存两个寄存器，一个是返回地址，另一个是 \$a0：

```
fact:
    addi $sp, $sp, -8 # adjust stack for 2 items
    sw $ra, 4($sp) # save the return address
    sw $a0, 0($sp) # save the argument n
```

第一次调用 fact 时，sw 保存程序中调用 fact 的地址。紧接着的两条指令测试 n 是否小于 1，如果 $n \geq 1$ 则跳转到 L1。

```
slti $t0,$a0,1 # test for n < 1
beq $t0,$zero,L1 # if n >= 1, go to L1
```

如果 n 小于 1，fact 将 1 置入一个值寄存器并返回。具体做法是在 0 上加 1 再将和存入 \$v0。然后从栈中弹出两个已保存的值并跳转到返回地址：

```
addi $v0,$zero,1 # return 1
addi $sp,$sp,8 # pop 2 items off stack
jr $ra # return to caller
```

在从栈中退出两项之前，本应该加载 \$a0 和 \$ra。但由于 n 小于 1 时，\$a0 和 \$ra 没有变化，所以就跳过了这些指令。

如果 n 不小于 1，参数 n 减 1 后，使用减 1 后的值再次调用 fact：

```
L1: addi $a0,$a0,-1 # n >= 1: argument gets (n - 1)
    jal fact # call fact with (n - 1)
```

下一条指令是 fact 的返回位置。现在旧的返回地址和旧的参数以及栈指针都需要恢复：

```
lw $a0, 0($sp) # return from jal: restore argument n
lw $ra, 4($sp) # restore the return address
addi $sp, $sp, 8 # adjust stack pointer to pop 2 items
```

接下来，值寄存器 \$v0 得到旧参数 \$a0 和当前值寄存器的乘积。这里假设乘法指令是可用的，尽管直到第 3 章才涉及乘法指令。

```
mul $v0,$a0,$v0 # return n * fact (n - 1)
```

最后，fact 再次跳转到返回地址：

```
jr $ra # return to the caller
```

□

01 硬件/软件接口 C 语言中的一个变量通常对应存储中的一个位置，其解释取决于其类型（type）和存储方式（storage class）。这方面的例子包括整型和字符型（见 2.9 节）。C 语言包括两种存储方式：动态的（automatic）和静态的（static）。动态变量位于过程中，当

过程退出时失效。静态变量在进入和退出过程时始终存在。在所有过程之外声明的 C 变量，以及声明时使用关键字 static 的变量都被视作静态的，其余的变量都被视作动态的。为了简化静态数据的访问，MIPS 软件保留了另一个寄存器，称为全局指针（global pointer），即 \$gp。

② 全局指针：指向静态数据区的保留寄存器。

图 2-11 总结了过程调用时所需保存的内容。需要注意的是，一些方案保存了栈，以确保调用者出栈时得到与压栈时相同的数据。只需保证被调用者不在 \$sp 以上进行写操作，\$sp 以上的栈就可以得到保存；而 \$sp 本身的保存是通过按被调用者将减去值的相同数量重新加上来实现的，其他寄存器则通过将它们保存到栈（如果它们被使用到的话）再从栈中恢复它们来进行保存。

保留	不保留
保存寄存器：\$s0 ~ \$s7	临时寄存器：\$t0 ~ \$t9
栈指针寄存器：\$sp	参数寄存器：\$a0 ~ \$a3
返回地址寄存器：\$ra	返回值寄存器：\$v0 ~ \$v1
栈指针以上的栈	栈指针以下的栈

图 2-11 过程调用时，保留和不保留的内容。如果软件依赖于下面将讨论的帧指针寄存器或者全局指针寄存器，那么它们也需要保留

102

2.8.3 在栈中为新数据分配空间

栈的最后一点复杂性是栈还需要存储过程的局部变量，但这些变量不适用于寄存器，例如局部的数组或结构体。栈中包含过程所保存的寄存器和局部变量的片段称为过程帧（procedure frame）或活动记录（activation record）。图 2-12 显示了过程调用之前、之中和之后栈的状态。

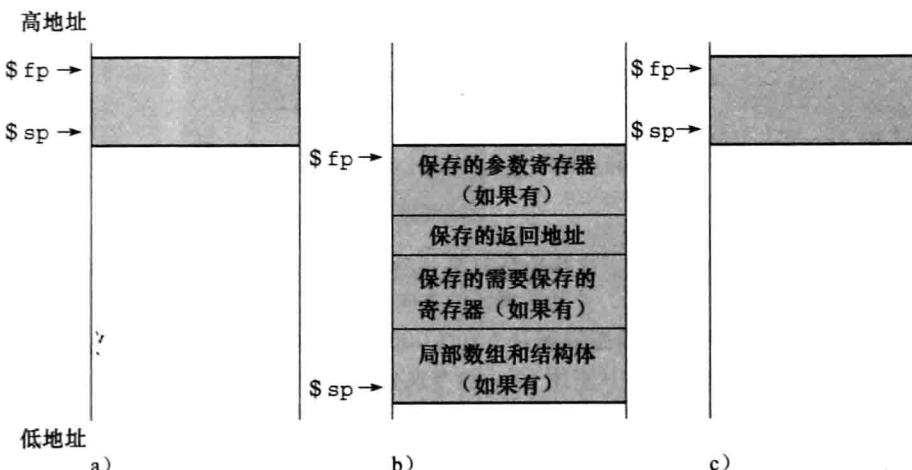


图 2-12 过程调用之前 (a)、之中 (b)、之后 (c) 栈的分配情况。帧指针 (\$fp) 指向该帧的第一个字（一般是保存的参数寄存器），而栈指针 (\$sp) 指向栈顶。栈可调整为有足够的空间来容纳所有的保存寄存器和驻留内存的局部变量。因为在程序运行期栈指针可能会改变，所以对于程序员而言，虽然使用栈指针和少量的地址运算就可能完成对变量的引用，但使用固定的帧指针引用变量会更为简单。如果在一个过程中栈内没有局部变量，编译器将可以不设置和不恢复帧指针以节省时间。当使用帧指针时，在调用中使用 \$sp 的地址进行初始化，而 \$sp 可以使用 \$fp 来恢复。相关内容可以在 MIPS 参考数据卡的第 4 列找到

103

某些 MIPS 软件使用帧指针 (frame pointer, \$fp) 指向过程帧的第一个字。在过程中栈指针可能会发生改变，因此存储器中对局部变量的引用在过程中的不同位置可能具有不同的偏移量，这使得过程更加难以理解。另一种方案，帧指针在一个过程中为局部存储器引用提供一个固定的基址寄存器。注意，无论是否使用显式的帧指针，活动记录都出现在栈中。我们通过避免在过程中修改 \$sp 来避免使用 \$fp，在我们的例子中，栈只在过程的入口和出口需要调整。

- ② 过程帧：也称作活动记录，栈中包含过程所保存的寄存器以及局部变量的片段。
- ③ 帧指针：指向给定过程中保存的寄存器和局部变量的值。

2.8.4 在堆中为新数据分配空间

除了动态变量对过程是局部有效之外，C 程序员还需要在内存中为静态变量和动态数据结构提供空间。图 2-13 给出了 MIPS 分配内存的约定。栈由内存高端开始并向下增长。内存低端的第一部分是保留的，之后是 MIPS 机器代码的第一部分，通常称为代码段 (text segment)。代码段之上的代码为静态数据段 (static data segment)，是存储常量和其他静态变量的空间。尽管数组通常具有固定长度因而能与静态数据段很好地匹配，但类似链表这样的数据结构通常会在生命期内增长或缩短。这类数据结构对应的段习惯上称为堆 (heap)，一般在存储器中放在静态数据段之后。注意，这种分配允许栈和堆相互生长，从而在两个段此消彼长的过程中达到内存的高效使用。

- ② 代码段：UNIX 目标文件中的段，包含源文件中例程对应的机器语言代码。

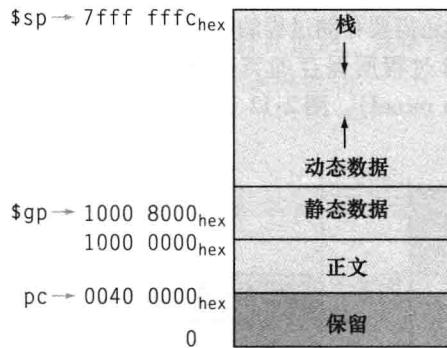


图 2-13 程序和数据的 MIPS 内存分配。这些地址只是一种软件规定，并非 MIPS 体系结构的一部分。栈指针初始化为 $7fff\ ffff_{16}$ ，并朝数据段的方向向下增长。在另一端，程序代码（代码段）从地址 $0040\ 0000_{16}$ 开始。静态数据从 $1000\ 0000_{16}$ 开始。然后是动态数据，在 C 中使用 malloc 命令分配，在 Java 中使用 new 命令来分配。动态数据在某一区域中朝着栈的方向向上生长，该区域称为堆。全局指针 \$gp 应设置为适当地址以便于访问数据。它初始化为 $1000\ 8000_{16}$ ，这样通过相对 \$gp 的正负 16 位的偏移量就可以访问从 $1000\ 0000_{16}$ 到 $1000\ ffff_{16}$ 之间的内存空间。关于这点可参见 MIPS 参考数据卡的第 4 列。

C 语言通过显式的函数调用在堆上分配和释放空间。malloc() 在堆上分配空间并返回指向它的指针，free() 释放指针指向的堆空间。内存分配由 C 程序控制，这是很多错误产生的根源。忘记释放空间会导致“内存泄漏”，它会逐渐耗尽大量内存以至于操作系统可能崩溃。过早释放空间会导致“悬摆指针”(dangling pointer)，会造成指针指向程序不想访问的位置。Java 使用自动的内存分配和无用单元回收机制来防止类似的错误发生。

图 2-14 总结了 MIPS 汇编语言的寄存器约定。这种约定是加速大概率事件的另外一个例子：传递 4 个参数、2 个寄存器用于返回结果、保存 8 个寄存器、10 个暂存器对于大多数过程调用来说足够使用。

名称	寄存器号	用途	调用时是否保存
\$zero	0	常数 0	不适用
\$v0 ~ \$v1	2 ~ 3	计算结果和表达式求值	否
\$a0 ~ \$a3	4 ~ 7	参数	否
\$t0 ~ \$t7	8 ~ 15	临时变量	否
\$s0 ~ \$s7	16 ~ 23	保存的寄存器	是
\$t8 ~ \$t9	24 ~ 25	更多临时变量	否
\$gp	28	全局指针	是
\$sp	29	栈指针	是
\$fp	30	帧指针	是
\$ra	31	返回地址	是

图 2-14 MIPS 寄存器约定。称为 \$at 的寄存器 1 被汇编器所保留（见 2.12 节），称为 \$k0 ~ \$k1 的寄存器 26 ~ 27 被操作系统所保留。关于这点也可见 MIPS 参考数据卡的第 2 列

01 精解 如果参数多于 4 个该怎么办呢？MIPS 约定将额外的参数放在栈中帧指针的上方。这样，过程从寄存器 \$a0 到 \$a3 中获得前 4 个参数，通过帧指针在内存中寻址获得其余参数。

如图 2-12 中所述，帧指针的方便性在于对过程中所有栈内的变量引用都具有相同的偏移。然而，帧指针并不是必需的。GNU MIPS C 编译器使用帧指针，而来自 MIPS 的 C 编译器则没有使用，它将寄存器 30 用作另一个保存的寄存器（\$s8）。

01 精解 一些递归过程可以不使用递归而用迭代的方式实现。通过消除过程调用的相关开销，迭代可以显著提高性能。例如，考虑下面一个用来求和的过程：

```
int sum (int n, int acc) {
    if (n > 0)
        return sum(n - 1, acc + n);
    else
        return acc;
}
```

考虑过程调用 sum(3, 0)。这将递归调用 sum(2, 3)、sum(1, 5) 和 sum(0, 6)，然后结果 6 将进行 4 次返回操作。这种求和的递归调用称为尾调用 (tail call)，而这个例子可以使用尾迭代 (tail recursion) 高效地实现（假设 \$a0 = n 且 \$a1 = acc）：

```
sum: slti $t0, $a0, 1          # test if n <= 0
    bne $t0, $zero, sum_exit  # go to sum_exit if n <= 0
    add $a1, $a1, $a0          # add n to acc
    addi $a0, $a0, -1          # subtract 1 from n
    j sum                      # go to sum
sum_exit:
    add $v0, $a1, $zero        # return value acc
    jr $ra                     # return to caller
```

105

01 小测验

下面关于 C 和 Java 的描述哪些是正确的？

- C 程序员显式地管理数据，而在 Java 中一般是自动的。
- C 比 Java 导致更多的指针错误和内存泄漏错误。