计 算 机 科 学 丛 书

原书第5版

# 计算机组成与设计

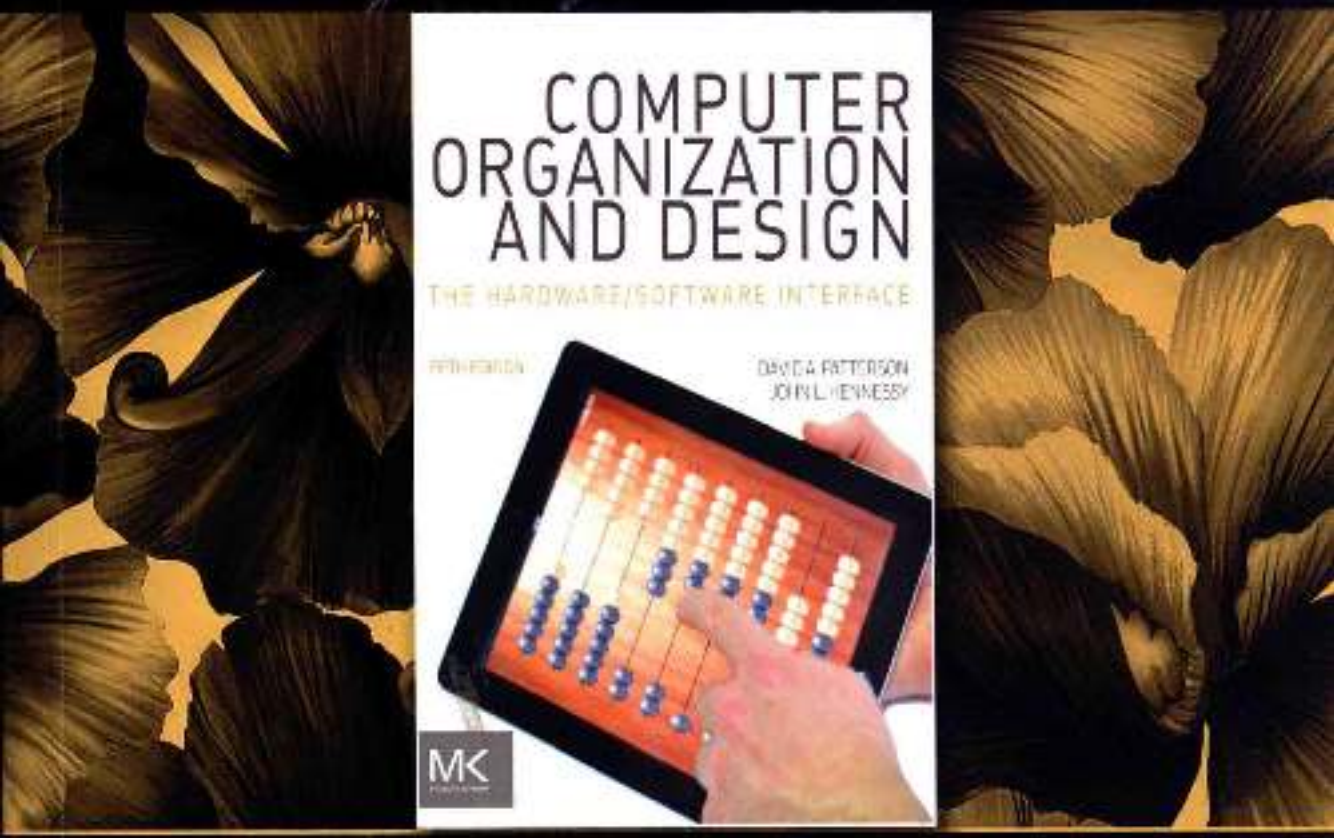
## 硬件/软件接口

[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

计 算 机 科 学 丛 书

原书第5版

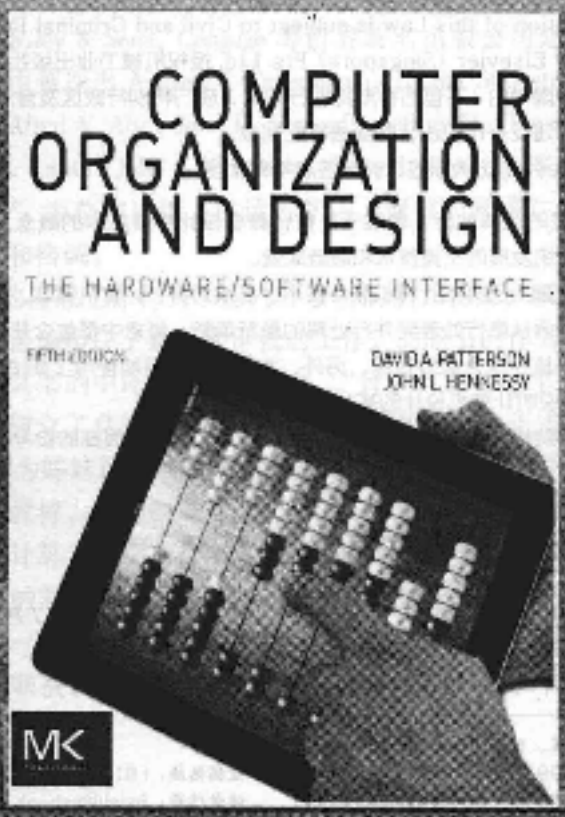
# 计算机组成与设计

## 硬件/软件接口

[美] 戴维 A. 帕特森 (David A. Patterson) 加州大学伯克利分校
约翰 L. 亨尼斯 (John L. Hennessy) 斯坦福大学 著
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。

本书是计算机组成的经典教材，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：毓　蕾　　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
有效无防伤标均为盗版
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，不仅奠定了研究的范畴，还揭示了学术的源流，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

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

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

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，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，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

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

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：(010) 88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育

华章科技图书出版中心

# 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。本书既新旧又新颖，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

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、庄泰、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

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》[⊖]。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权衡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。

---

⊖ 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（目录中带有图图标的章节）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 计算机抽象与技术 | 1.1～1.11 | ■■■ | ■■■ |
| | 图1.12（历史） | ■■□ | ■■□ |
| 第2章 指令：计算机的语言 | 2.1～2.14 | ■■■ | ■■□ |
| | 图2.15（编译器和Java） | ■■□ | |
| | 2.16～2.20 | ■■■ | ■■□ |
| | 图2.21（历史） | ■■□ | ■■□ |
| 附录E RISC指令集体系结构 | 图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） | ■■□ | ■■□ |
| | 图3.11（历史） | ■■□ | ■■□ |
| 附录B 逻辑设计基础 | B.1～B.13 | | ■■□ |
| 第4章 处理器 | 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） | | ■■■ |
| | 4.10～4.12（并行和实例） | ■■■ | ■■■ |
| | 图4.13（Verilog流水线控制） | | ■■□ |
| | 4.14～4.15（谬误） | ■■■ | ■■■ |
| | 图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 | 图 D.1－D.6 | | |
| 第 5 章　大容量和高速度：开发存储器层次结构 | 5.1－5.10 | | |
| | 图 5.11（廉价冗余磁盘阵列） | | |
| | 图 5.12（Verilog cache 控制器） | | |
| | 5.13－5.16 | | |
| | 图 5.17（历史） | | |
| 第 6 章　从客户端到云的并行处理器 | 6.1－6.8 | | |
| | 图 6.9（网络） | | |
| | 6.10－6.14 | | |
| | 图 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 | 图 C.1－C.13 | | |

仔细阅读　有时间阅读　作为参考　阅顾或阅读　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 GPU 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@ mkp. 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

tryn、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 Farrens（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 Paltashev，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，而引导了后来的云计算。这些项目获得了二个 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。

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
2.12.4 加载器 …… 87
2.12.5 动态链接库 …… 87
2.12.6 启动一个 Java 程序 …… 89
2.13 以一个 C 排序程序作为完整的例子 …… 90
2.13.1 swap 过程 …… 90
2.13.2 sort 过程 …… 91
2.14 数组与指针 …… 96
2.14.1 用数组实现 clear …… 96
2.14.2 用指针实现 clear …… 97
2.14.3 比较两个版本的 clear …… 97
2.15 高级内容：编译 C 语言和解释 Java 语言 …… 98
2.16 实例：ARMv7（32 位）指令集 … 98
2.16.1 寻址模式 …… 99
2.16.2 比较和条件分支 …… 100
2.16.3 ARM 的特色 …… 100
2.17 实例：x86 指令集 …… 102
2.17.1 Intel x86 的改进 …… 102
2.17.2 x86 寄存器和数据寻址模式 …… 103
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
2.19 谬误与陷阱 …… 109
2.20 本章小结 …… 110
2.21 历史观点和拓展阅读 …… 111
2.22 练习题 …… 112

**第 3 章 计算机的算术运算 …… 117**

3.1 引言 …… 117
3.2 加法和减法 …… 117
3.3 乘法 …… 121
3.3.1 顺序的乘法算法和硬件 …… 121
3.3.2 有符号乘法 …… 124
3.3.3 更快速的乘法 …… 124
3.3.4 MIPS 中的乘法 …… 124
3.3.5 小结 …… 125
3.4 除法 …… 125
3.4.1 除法算法及其硬件结构 …… 125
3.4.2 有符号除法 …… 128
3.4.3 更快速的除法 …… 128
3.4.4 MIPS 中的除法 …… 129
3.4.5 小结 …… 129
3.5 浮点运算 …… 130
3.5.1 浮点表示 …… 131
3.5.2 浮点加法 …… 135
3.5.3 浮点乘法 …… 138
3.5.4 MIPS 中的浮点指令 …… 139
3.5.5 算术精确性 …… 145
3.5.6 小结 …… 146
3.6 并行性和计算机算术：子字并行 …… 148
3.7 实例：x86 中流处理 SIMD 扩展和高级向量扩展 …… 149
3.8 加速：子字并行和矩阵乘法 …… 150
3.9 谬误与陷阱 …… 153
3.10 本章小结 …… 155
3.11 历史观点和拓展阅读 …… 158
3.12 练习题 …… 159

**第 4 章 处理器 …… 162**

4.1 引言 …… 162
4.2 逻辑设计的一般方法 …… 165
4.3 建立数据通路 …… 167
4.4 一个简单的实现机制 …… 173
4.4.1 ALU 控制 …… 173
4.4.2 主控制单元的设计 …… 175
4.4.3 为什么不使用单周期实现方式 …… 181
4.5 流水线概述 …… 182
4.5.1 面向流水线的指令集设计 …… 186
4.5.2 流水线冒险 …… 186
4.5.3 对流水线概述的小结 …… 191
4.6 流水线数据通路及其控制 …… 192
4.6.1 图形化表示的流水线 …… 200
4.6.2 流水线控制 …… 203
4.7 数据冒险：旁路与阻塞 …… 206
4.8 控制冒险 …… 214
4.8.1 假定分支不发生 …… 215

4.8.2 缩短分支的延迟 …… 215
4.8.3 动态分支预测 …… 216
4.8.4 流水线小结 …… 220
4.9 异常 …… 221
4.9.1 MIPS 体系结构中的异常处理 …… 221
4.9.2 在流水线实现中的异常 …… 222
4.10 指令级并行 …… 226
4.10.1 推测的概念 …… 227
4.10.2 静态多发射处理器 …… 227
4.10.3 动态多发射处理器 …… 231
4.10.4 能耗效率与高级流水线 … 233
4.11 实例：ARM Cortex-A8 和 Intel Core i7 流水线 …… 234
4.11.1 ARM Cortex-A8 …… 235
4.11.2 Intel Core i7 920 …… 236
4.11.3 Intel Core i7 920 的性能 … 238
4.12 运行更快：指令级并行和矩阵乘法 …… 240
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
4.14 谬误与陷阱 …… 242
4.15 本章小结 …… 243
4.16 历史观点和拓展阅读 …… 243
4.17 练习题 …… 243

**第5章 大容量和高速度：开发存储器层次结构** …… 252

5.1 引言 …… 252
5.2 存储器技术 …… 255
5.2.1 SRAM 技术 …… 256
5.2.2 DRAM 技术 …… 256
5.2.3 闪存 …… 258
5.2.4 磁盘存储器 …… 258
5.3 cache 的基本原理 …… 259
5.3.1 cache 访问 …… 261
5.3.2 cache 缺失处理 …… 265
5.3.3 写操作处理 …… 266
5.3.4 一个 cache 的例子：内置 FastMATH 处理器 …… 267
5.3.5 小结 …… 269
5.4 cache 性能的评估和改进 …… 270
5.4.1 通过更灵活地放置块来减少 cache 缺失 …… 272
5.4.2 在 cache 中查找一个块 …… 275
5.4.3 替换块的选择 …… 276
5.4.4 使用多级 cache 结构减少缺失代价 …… 277
5.4.5 通过分块进行软件优化 …… 280
5.4.6 小结 …… 283
5.5 可信存储器层次 …… 283
5.5.1 失效的定义 …… 283
5.5.2 纠正一位错、检测两位错的汉明编码（SEC/DED） … 284
5.6 虚拟机 …… 287
5.6.1 虚拟机监视器的必备条件 …… 289
5.6.2 指令集系统结构（缺乏）对虚拟机的支持 …… 289
5.6.3 保护和指令集系统结构 …… 289
5.7 虚拟存储器 …… 290
5.7.1 页的存放和查找 …… 293
5.7.2 缺页故障 …… 294
5.7.3 关于写 …… 297
5.7.4 加快地址转换：TLB …… 297
5.7.5 集成虚拟存储器、TLB 和 cache …… 300
5.7.6 虚拟存储器中的保护 …… 302
5.7.7 处理 TLB 缺失和缺页 …… 303
5.7.8 小结 …… 307
5.8 存储器层次结构的一般框架 …… 309
5.8.1 问题1：一个块可以被放在何处 …… 309
5.8.2 问题2：如何找到一个块 … 310
5.8.3 问题3：当 cache 缺失时替换哪一块 …… 311
5.8.4 问题4：写操作如何处理 …… 311
5.8.5 3C：一种理解存储器层次结构行为的直观模型 …… 312
5.9 使用有限状态机来控制简单的 cache …… 314
5.9.1 一个简单的 cache …… 314
5.9.2 有限状态机 …… 315

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
6.9 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

*在不关注具体过程的情况下完成复杂的重要操作，这种方法极大地促进了文明的进步。*

*——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。

- 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，但能够提供更高的计算、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。

- 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1 000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。

- 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。
- 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计算解决$2^n$与$10^n$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gigibit（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。 5

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。

**细解** 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 6
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。

- 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。

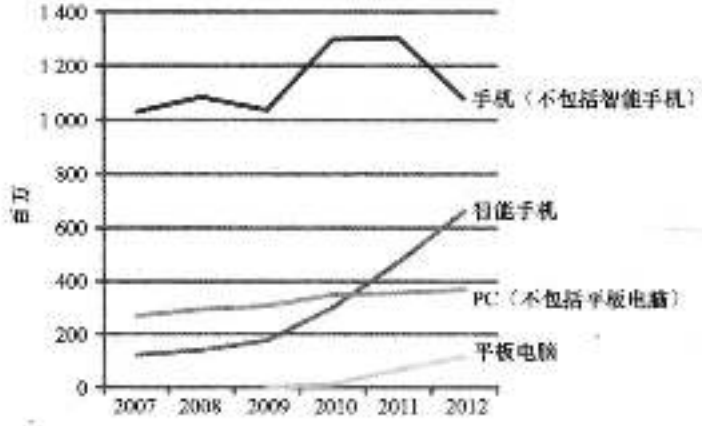



图 1-2 每年生产的平板电脑和智能手机与 PC 和传统手机的数量对比。平板电脑和智能手机代表着后 PC 时代。智能手机反映了手机工业的近期增长情况，并且在 2011 年超过了 PC 的产量。平板电脑产量增长最快，2012 年的产量几乎是 2011 年的两倍。PC 和传统手机的产量保持不变甚至在下降

### 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。



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的认知。

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年的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？

8

⊃ 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。

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。

⊃ 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。

### 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' 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。




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其应用又反过来运行在该数据库之上

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，其过程显然过于原始。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

```
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：以二进制无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。

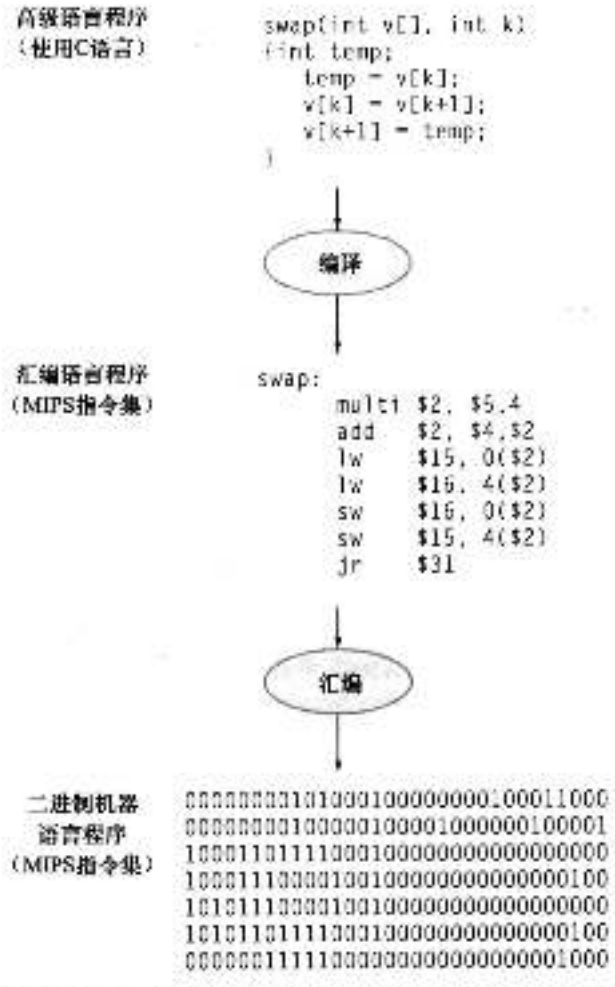



图 1-4 C 程序编译为汇编语言程序，再汇编为二进制机器语言程序。尽管将高级语言翻译成二进制的机器语言仅需要两步，但一些编译器将“中间人”砍掉，直接产生二进制的机器语言。这些语言和本图中列举的程序将在第 2 章详细检测

- 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。

- 输入设备：为计算机提供信息的装置，如键盘。
- 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。

16

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。

**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（在计算机中也称为算术）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）中的每个部件归于这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。

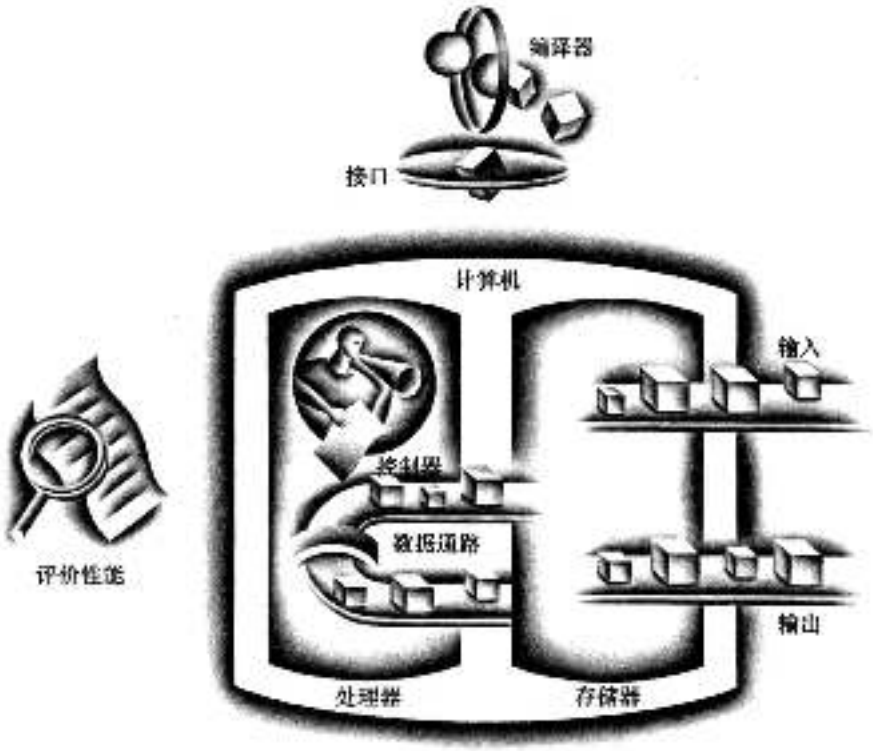



图 1-5 组成计算机的 5 个经典部件。处理器从存储器中得到指令和数据，输入部件将数据写入存储器，输出部件从内存中读出数据，控制器向数据通路、存储器、输入和输出部件发出命令信号

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。

- 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。
- 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。
- 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，我将飞机降落在航空母舰的甲板上，观察到一个原子打到墙壁中，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，典型的屏幕中显示矩阵的大小可以从1 024×768到2 048×1 536。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。

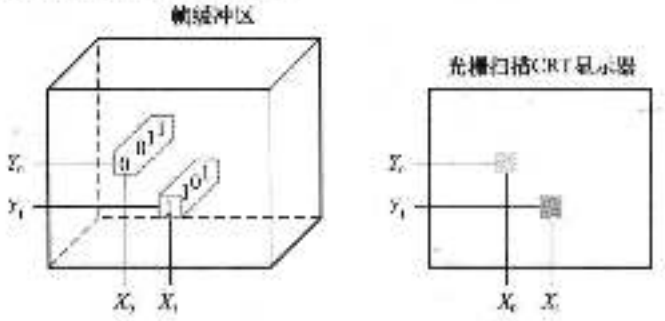



图1-6 左边的帧缓冲区中每个坐标决定了右边光栅扫描CRT显示中相应坐标的灰度。像素（$X_0$，$Y_0$）的灰度值是0011，小于像素（$X_1$，$Y_1$）的灰度值，（$X_1$，$Y_1$）的灰度值是1101

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。

- **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。
- **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 的锂合物电池，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。

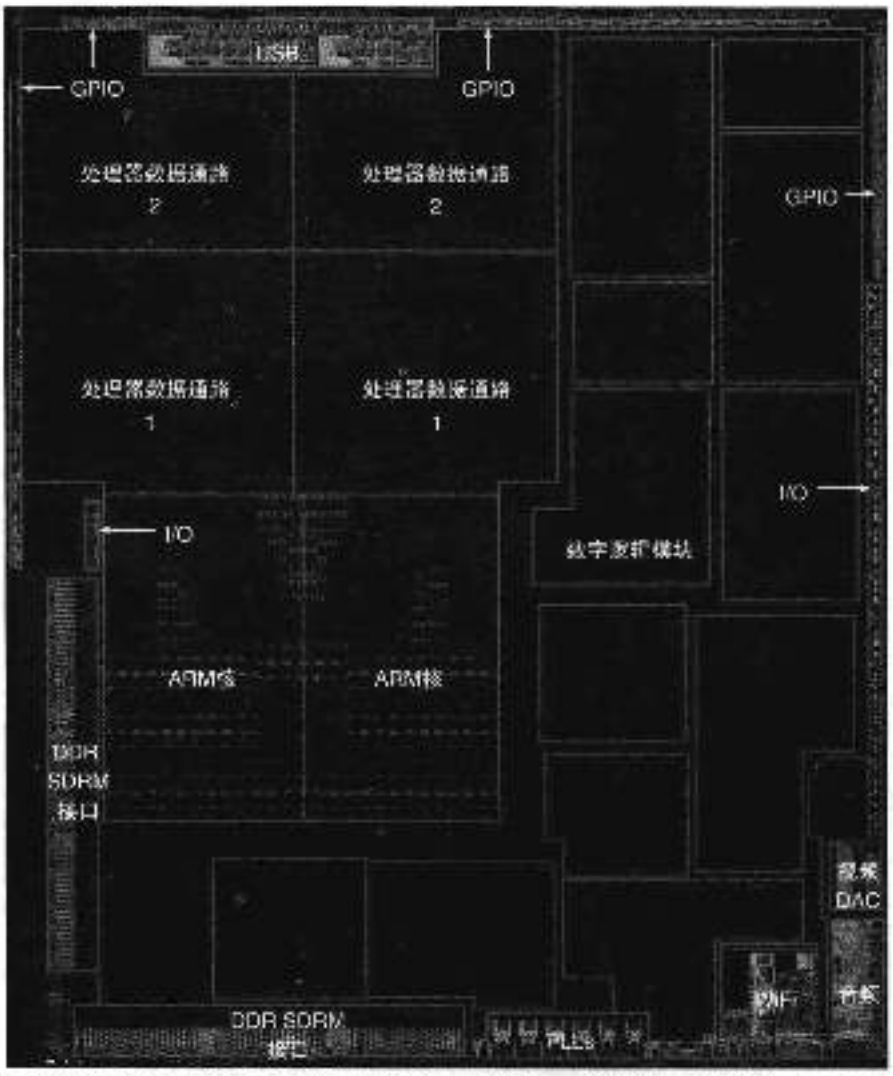



图 1-9 A5 内部的处理器集成电路。芯片尺寸为 12.1mm×10.1mm，采用 45nm 工艺制造（见 1.5 节）。左半部分靠中间的位置是两个相同的 ARM 处理器，左上角的四分之一部分是具有 4 条数据通路的图形处理器单元（Graphic Processor Unit，GPU），左下角和底部是与主存的接口（Chipworks 友情提供，www.chipworks.com）

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。

- 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。
- 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，其访问时间大约为 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。

19-21

- 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。
- 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。

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）。

- 指令集体系结构：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。
- 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。

- 实现：遵循体系结构抽象的硬件。

**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。

### 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）。

- 易失性存储器：类似 DRAM 的存储器，仅在加电时保存数据。
- 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。 22

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。

- 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。
- 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。
- 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，所以磁盘的存取时间大约是5～20毫秒，2012年每g字节的价格大约为0.05～0.1美元。
- 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。

23

- 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。
- 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 Eva）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。

**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。 24

| 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）。 25
- 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。

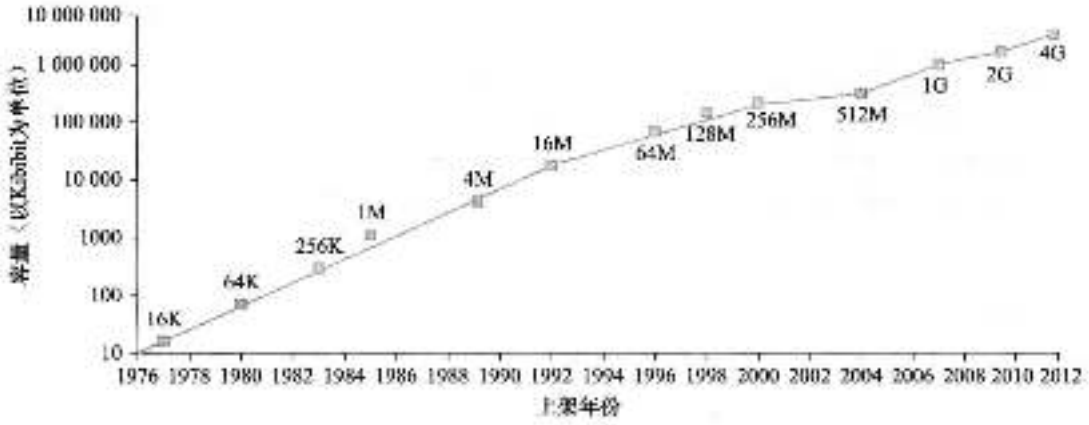



图 1-11 单片 DRAM 容量随时间的增长。纵轴单位为 Kb，其中 K 为 1 024（$2^{10}$位）。在近 20 多年中，平均每隔 3 年 DRAM 容量扩大 4 倍，即每年增长约 60%。在最近几年中，增长速度有所下降，接近每 2～3 年翻一番的水平

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。

- 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。
- 晶圆：厚度不超过 0.1 英寸的硅锭片，用来制造芯片。

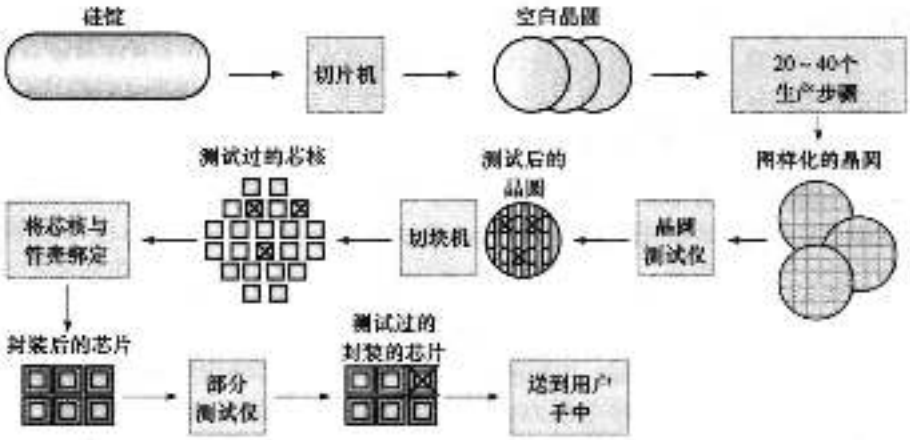



图 1-12 芯片制造的全过程。从硅锭切下来之后，空白的晶圆经过大约 20～40 步的加工，产生图样化的晶圆（见图 1-13）。这些图样化的晶圆由晶圆测试器进行测试，测试后生成一张图，表明哪些部分是合格的。之后，这些晶圆被进一步切成芯片（见图 1-9）。在本图中，一个晶圆能生产 20 个芯片，其中有 17 个通过测试（X 意味着这个芯片是坏的）。本例中芯片的良率（又称成品率）是 17/20，也就是 85%。这些合格芯片被封装起来并且在发布给用户之前经过多次测试。不合格的封装会在最终测试中被发现

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

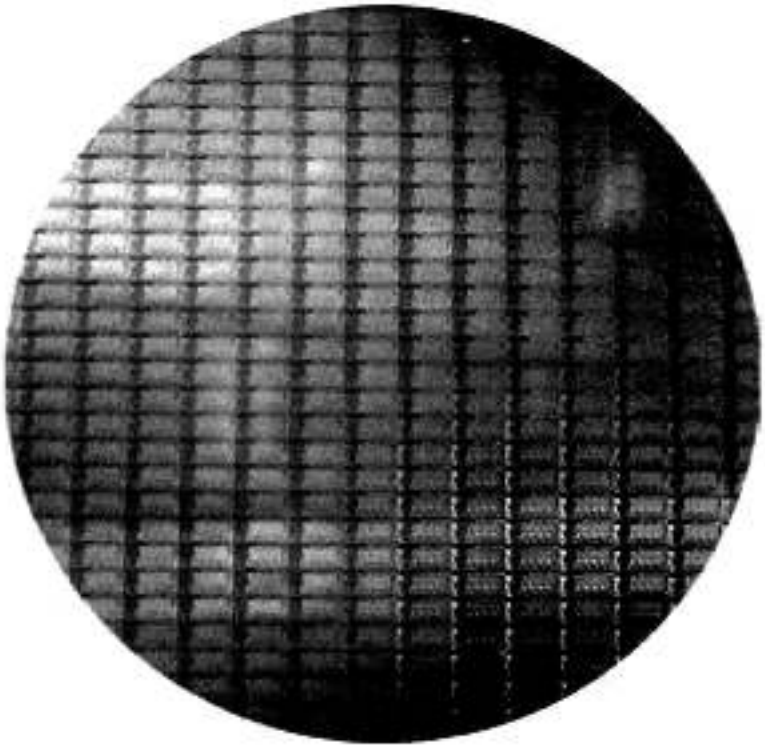


图 1-13 Intel Core i7 芯片的 12 英寸（300mm）晶圆（Intel 提供）。良率为 100%的图片中的晶圆的数目是 280，每个为 20.7mm × 10.5mm。晶圆边缘几十个不完整的芯片是没用的。之所以包含它们，是因为这样给硅片生产掩膜相当容易。芯片使用 32nm 的工艺，这意味着最小的晶体管的尺寸几乎接近 32nm，尽管它们通常比实际的特征尺寸还要小，这个特征尺寸是将晶体管“图纸尺寸”和最终的生产尺寸相比得出的

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。

- 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。
- 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。
- 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。 27

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。

**01 精解** 集成电路的成本可以用下面 3 个简单公式来表示：

$$\text{每芯片的价格} = \text{每晶圆的价格} / (\text{每晶圆的芯片数} \times \text{成品率})$$

$$\text{每晶圆的芯片数} \approx \text{晶圆面积} / \text{芯片面积}$$

$$\text{成品率} = 1/(1 + (\text{单位面积的缺陷数} \times \text{芯片面积}/2))^2$$

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。

**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。 29

- 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。
- 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。

**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。

**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

$$\text{性能}_X > \text{性能}_Y$$

$$1/\text{执行时间}_X > 1/\text{执行时间}_Y$$

$$\text{执行时间}_Y > \text{执行时间}_X$$

也就是说，如果 X 比 Y 快，那么 Y 的执行时间比 X 长。 30

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”的表述方式，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

$$\text{性能 } X/\text{性能 } Y = \text{执行时间 } Y/\text{执行时间 } X = n$$

**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?

**Q1 答案**

我们知道，A 是 B 的 n 倍快，则

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

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

31

$$\text{性能}_A/1.5 = \text{性能}_B$$

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。

**Q1 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，对增上时 32
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。

**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 33
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：

$$\text{一个程序的 CPU 执行时间} = \text{一个程序的 CPU 时钟周期数} \times \text{时钟周期时间}$$

由于时钟频率和时钟周期时间互为倒数，故

$$\text{一个程序的 CPU 执行时间} = \frac{\text{一个程序的 CPU 时钟周期数}}{\text{时钟频率}}$$

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。

**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？

**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：

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

$$10\text{ 秒} = \text{CPU 时钟周期数}_A / 2\times10^9 \text{ 周期数/秒}$$

34

$$\text{CPU 时钟周期数}_A = 10\text{ 秒} \times 2\times10^9 \text{ 周期数/秒} = 20\times10^9 \text{ 周期数}$$

B 的 CPU 时间公式为：

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

$$6\text{ 秒} = 1.2\times20\times10^9 \text{ 时钟周期数}/\text{时钟频率}_B$$

$$\text{时钟频率}_B = 1.2\times20\times10^9 \text{ 时钟周期数}/6\text{ 秒} = 0.2\times20\times10^9 \text{ 时钟周期数/秒}$$

$$= 4\times10^9 \text{ 时钟周期数/秒} = 4\text{GHz}$$

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。

➲ 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。

**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

35 计算机执行的速度更快？快多少？

**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

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

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

**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？

**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

**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）：

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

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设计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，执行指令数减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，
38 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。

| 硬件或软件部件 | 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，当算法使用更多的浮点运算时，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。有些设计采用 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 × 0.6/1.1 = 8.2 秒

b. 15 × 0.6 × 1.1 = 9.9 秒

c. 15 × 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。两者的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。

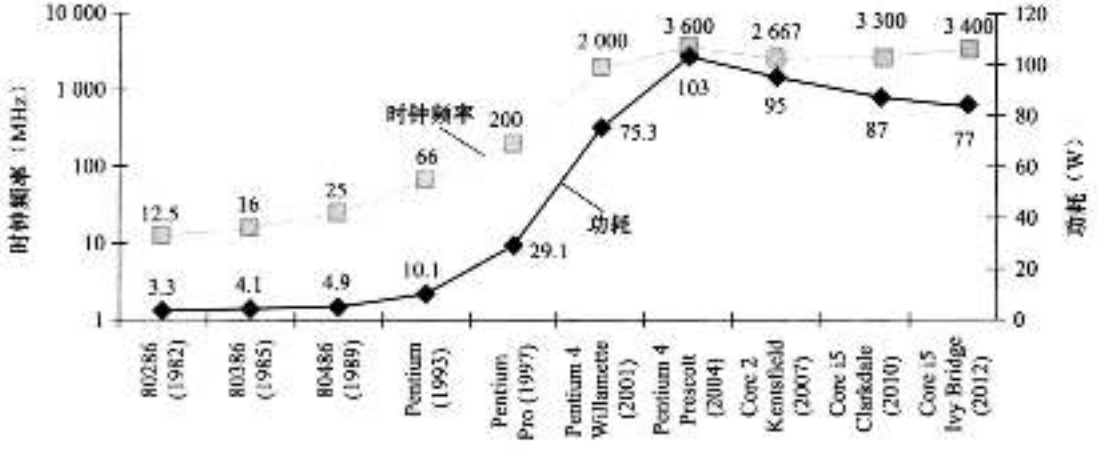



图 1-16 25 年间 Intel x86 八代微处理器的时钟频率和功耗。奔腾 4 处理器时钟频率和功耗提高很大，但是性能提升不大。Prescott 发热问题导致奔腾 4 处理器的生产线被放弃。Core 2 生产线恢复使用低时钟频率的简单流水线和片上多处理器。Core i5 采用同样的流水线

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→1→0或1→0→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。

**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
41 态功耗有何影响？

**答案**

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

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，设计者还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。

**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，就会增加静态功耗，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。

**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较问题。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台式
42 机计算机的主要开销（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。

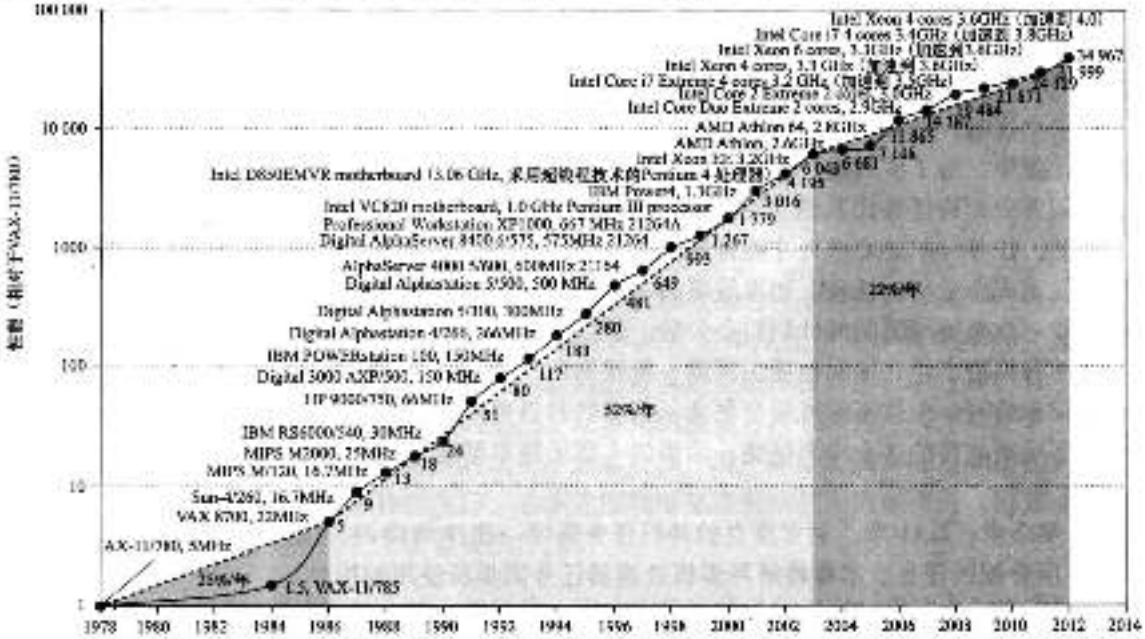



图 1-17 自 20 世纪 80 年代中期以来处理器性能的发展。本图描绘了和 VAX 11/780 相比，采用 SPECint 测试程序得到的性能数据（见 1.10 节）。在 20 世纪 80 年代中期以前，性能的增长主要靠技术驱动，平均每年增长 25%。在这个阶段之后，增长速度达到 52%，这归功于体系结构和组织方式的创新。从 20 世纪 80 年代中期开始，性能每年大约提高 52%，如果按照原先的 25% 的增长率计算，则到 2002 年的性能只有实际的 1/7。从 2002 年开始，受到功耗、指令级并行程度和存储器长延迟时间的限制，单核处理器的性能增长放缓，大约每年 22%

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。

**硬件/软件接口** 并行性对计算性能一直十分重要，但它往往是隐藏的。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。在拍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“第三波浪”，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
43 将成功地跃迁到显式并行编程。

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。

第二个原因是为了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。

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
44 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，子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
45 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。

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，就像每一样，“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。虽然管的速度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。

- 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。
- 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 46
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。

**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。 47

| 描述 | 名称 | 指令数目（×10⁹） | CPI | 时钟周期时间（×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 |
| 图像游戏（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 |
| Σssj_ops / Σpower = | | 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 48
行归纳，称为“overall ssj_ops per watt”，其计算公式是：

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

式中，$\text{ssj\_ops}_i$ 为工作负载在每 10% 增量处的性能，$\text{power}_i$ 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以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网套，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：

$$\text{改进后的执行时间} = 80/n + (100 - 80)$$ 49

由于要求快至 5 倍，新的执行时间应该是 20：

$$20 = 80/n + 20$$

$$0 = 80/n$$

⊃ 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 基准测试程试，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。另一种常犯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。

⊃ 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。

**1.1** 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 |

a. 哪台计算机的 MIPS 值更高？

51 b. 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其不同的实现方法（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，要首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。 52

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
53 机如何处理算术运算数据。附录B介绍逻辑设计。

## 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郊狼窝：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

54 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1 280×1 024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？

**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。

a. 以每秒钟执行的指令数目为标准，哪个处理器性能最高？

b. 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。

c. 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$1.0 \times 10^6$条动态指令，按如下比例分为4类：A，10%；B，20%；C，50%；D，20%。

a. 每种实现方式总的CPI是多少？

b. 计算两种情况下的时钟周期。

55

**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$1.0 \times 10^9$，执行时间为1.1s；如果采用编译程序B，则动态指令数为$1.2 \times 10^9$，执行时间为1.5s。

a. 在给定处理器时钟周期为1ns时，找出每个程序的平均CPI。

b. 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？

c. 假设开发了一种新的编译程序，只用$6.0 \times 10^8$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$2.56 \times 10^9$条算术指令、$1.28 \times 10^9$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。

56

**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瑕疵/$cm^2$。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瑕疵/$cm^2$

**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。

**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200mm$^2$，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$2.38\times10^{12}$，执行时间为750s，参考时间为9 650s。

**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$5.0\times10^9$条指令；P2的时钟频率为3GHz，平均CPI为0.75，需要执行$1.0\times10^9$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$1.0\times10^9$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

$$\text{MFLOPS} = \text{浮点操作的数目} / (\text{执行时间}\times(1\times10^6))。$$

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节引证为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$50\times106$条浮点指令、$110\times106$条整数指令、$80\times106$条L/S指令和$16\times106$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 $t = 100s$。当它在 $p$ 个处理器上运行时，每个处理器需要 $t/p$ s 的计算时间，两外还需要 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 50～70ns），每 GB 的价格（$5 - $10）。磁盘存储器：非易失性，访问时间比 DRAM 慢 100 000～400 000 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 100～1 000 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。

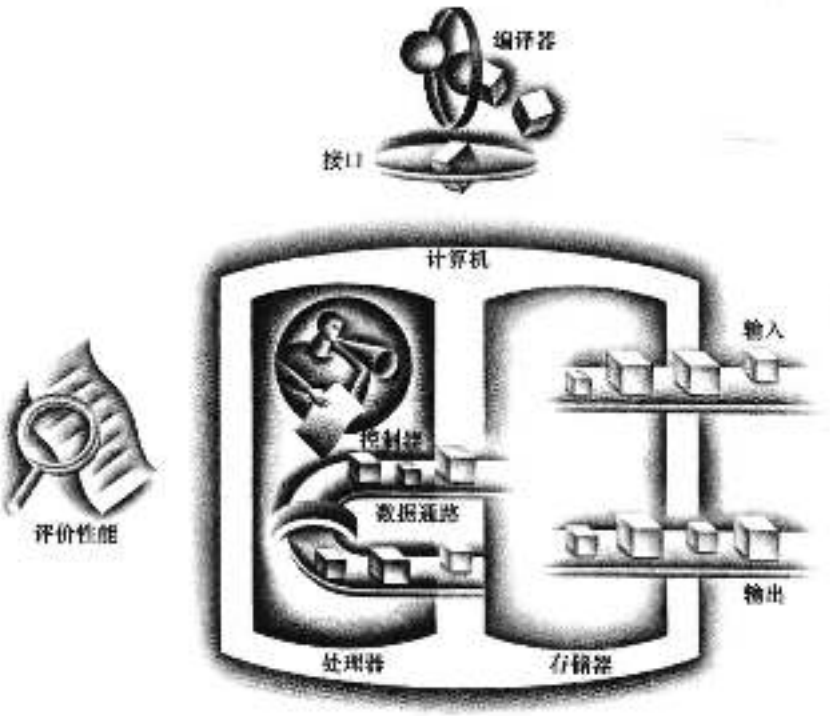
**1.6** b

**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




计算机的5个经典部件

*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。*

——*法国国王查理五世 (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形式，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。通过简要介绍其他二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”足以按规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*

60 ~ 62

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 |
| | 取立即数的高位 | lui $s1,20 | $s1 = 20 × $2^{16}$ | 取立即数并放到高16位 |
| 逻辑 | 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 |
| | 逻辑右移 | srl $s1,$s2,10 | $s1 = $s2 >> 10 | 根据常数右移相应位 |
| 条件分支 | 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 | sltui $s1,$s2,20 | if($s2 < 20) $s1 = 1; else $s1 = 0 | 比较是否小于常数 |
| | 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 |
| 无条件跳转 | 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二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。

**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。

**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
```

□

**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？

**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)
```

□

**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

**解答** 为了增强可移植性，Java 最初被设定为条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。因为现在 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。

**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

**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。

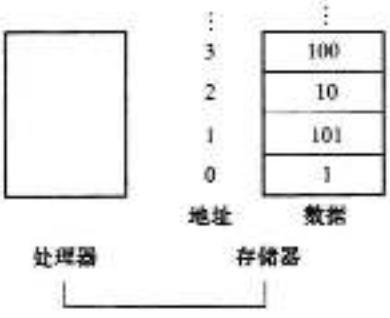



图2-2 存储器地址和该地址对应的数据。如果这些元素是字，那么这些地址就是错误的，因为MIPS实际上是按字节编址的，而一个字是4字节。图2-3给出了顺序字编址的内存寻址

- 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。

68

**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];
```

**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]传送到寄存器中。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）。 □

**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，当大多数体系结构按字节编址。因此，一个字的地址必须和它所包括的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。

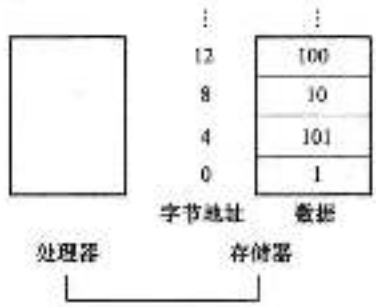



图2-3 实际的MIPS存储器地址和该地址对应的数据，相对于图2-2，变化的地址采用高亮显示。由于MIPS按字节编址，而字地址是4的倍数，因为每个字的长度为4字节

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

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

**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 gebibytes（$2^{30}$）或 tebibytes（$2^{40}$）表示，而不用 gigabytes（$10^9$）或 terabytes（$10^{12}$），见图 1-1。 70

**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];
```

**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。 □

**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。 71

## 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：

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

假设 $s1 + 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。

**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。

**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。

72

**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。

**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，基地址可能由于太大而不适宜用偏移量表示。

**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。

⊃ 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，

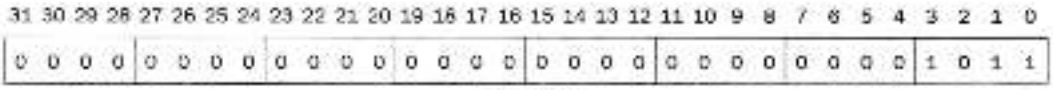
$1011_2$

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。




（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）。

- 最低有效位：在 MIPS 字中最右边的一位。
- 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：

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

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 $xi$ 表示数字 $x$ 的第 $i$ 位）：

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

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。

**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。最简单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 0000₂ = 0₁₀
0000 0000 0000 0000 0000 0000 0000 0001₂ = 1₁₀
0000 0000 0000 0000 0000 0000 0000 0010₂ = 2₁₀
...                                        ...
0111 1111 1111 1111 1111 1111 1111 1101₂ = 2,147,483,645₁₀
0111 1111 1111 1111 1111 1111 1111 1110₂ = 2,147,483,646₁₀
0111 1111 1111 1111 1111 1111 1111 1111₂ = 2,147,483,647₁₀
1000 0000 0000 0000 0000 0000 0000 0000₂ = -2,147,483,648₁₀
1000 0000 0000 0000 0000 0000 0000 0001₂ = -2,147,483,647₁₀
1000 0000 0000 0000 0000 0000 0000 0010₂ = -2,147,483,646₁₀
...                                        ...
1111 1111 1111 1111 1111 1111 1111 1101₂ = -3₁₀
1111 1111 1111 1111 1111 1111 1111 1110₂ = -2₁₀
1111 1111 1111 1111 1111 1111 1111 1111₂ = -1₁₀
```

上面的数字中一半是正数，从 $0-2\,147\,483\,647_{10}$（$2^{31}-1$），这些数字的表示方式与之前是一样的。紧接着的 $1000\cdots0000_2$ 表示最小的负数 $-2\,147\,483\,648_{10}$（$-2^{31}$）。而后是按照绝对值递减的负数：从 $-2\,147\,483\,647_{10}$（$1000\cdots0001_2$）到 $-1_{10}$（$1111\cdots1111_2$）。

二进制补码中的最小负数 $-2\,147\,483\,648_{10}$ 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。

75

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：

$$(x31\times-2^{31})+(x30\times2^{30})+(x29\times2^{29})+\cdots+(x1\times2^{1})+(x0\times2^{0})$$

符号位被 $-2^{31}$ 乘，其余的位仍按前面的方法计算。

**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？

$1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1100_2$

**答案**

将数的位值代用上面的公式：

$$(1\times(-2^{31}))+(1\times2^{30})+(1\times2^{29})+\cdots+(1\times2^{2})+(0\times2^{1})+(0\times2^{0})$$
$$=-2^{31}+2^{30}+2^{29}+\cdots+2^{2}+0+0$$

$= -2\,147\,483\,648_{10} + 2\,147\,483\,644_{10}$

$= -4_{10}$ ⊖

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。

**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。

76

**硬件/软件接口** 与上面所讨论负数不同，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$111\cdots111_2$，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）。

**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。

**答案**

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

```
      1111 1111 1111 1111 1111 1111 1111 1101₂
+                                          1₂
-------------------------------------------
-     1111 1111 1111 1111 1111 1111 1111 1110₂
-     -2₁₀
```

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$1 \times 2^2$，原书中为$1 \times 2^1$，原书有误。——译者注

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 \\
+ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_2 \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 \\
= & 2_{10}
\end{array}
$$

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）。

**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。

**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，

$0000\ 0000\ 0000\ 0010_2$

变成

$$
\begin{array}{ll}
 & 1111\ 1111\ 1111\ 1101_2 \\
+ & \qquad\qquad\qquad\qquad 1_2 \\
\hline
= & 1111\ 1111\ 1111\ 1110_2
\end{array}
$$

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：

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

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
78 恢复了其中一部分。

## 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。

**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。

**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？

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000_2$

1）$-4_{10}$

2）$-8_{10}$

3）$-16_{10}$

4）$18\,446\,744\,073\,709\,551\,609_{10}$。

**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 $10\cdots000_2$，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）。

- 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。
- 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。

**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。

**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）。

- 指令格式：二进制数字段组成的指令表示形式。
- 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。

- 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 0x*xxxx* 来表示十六进制数。

**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：

$eca8\ 6420_{16}$

81

$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_2$

**答案**

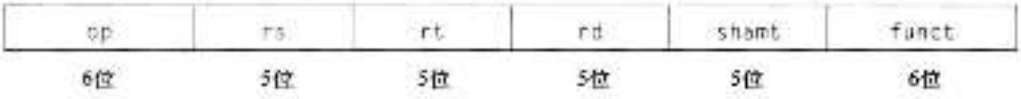
按图 2-4 所示十六进制－二进制转换表查表得：




□

## MIPS 字段

为了使讨论变得简单，给 MIPS 字段命名如下：




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。

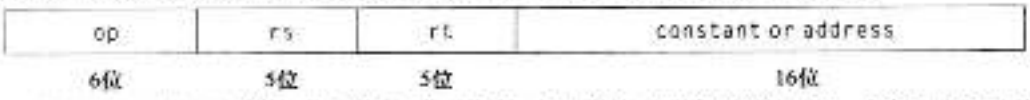
操作码：指令中用来表示操作和格式的字段。

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。 82

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：




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substract（34）

83

### 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$s1中，h存放在$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？

### 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（$s1），在第三个字段（rt）中指定目的寄存器8（$t0）。在最后一个字段address中存放用于指定A[300]的偏移量（1200 = 300 × 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 0000$_2$）：

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

84

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。

□

**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将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

**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运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，它能够将那些用高于人类使用的符号编写的代码翻译成机器能理解的代码

**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，"叮当弟弟看说，"如果那是真的，那它就可能是真的；如果那曾经是真的，它就是真话；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 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 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

- 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。
- 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。
- 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。
- 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设备等的外部接口请求。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

89 2. 左移后再进行右移

## 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。

- 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。

**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;
```

**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
```

90

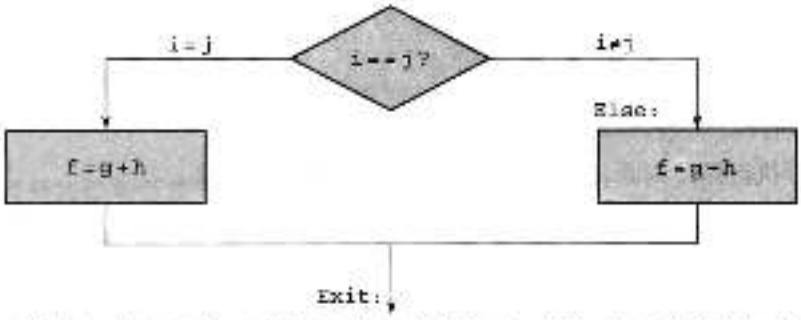



图 2-9 上述 if 语句的程序流程图。左边方框对应 if 语句的 then 部分，右边方框对应 if 语句的 else 部分

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）。

**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。

**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。

**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] ≠ k 则退出循环：

92

```
bne $t0,$s5, Exit    # go to Exit if save[i] != k
```

再下一条指令将 i 加 1：

```
add $s3,$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。） □

**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。

⇒ 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，则将第二个寄存器设置为 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
```

**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

或者存器 $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。

**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。

**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

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

而寄存器 $s1 中的二进制数为

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$$

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
```

**答案**

如果是有符号数，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 $4\,294\,967\,295_{10} > 1_{10}$。 □ 94

将有符号数作为无符号数来处理，是一种检验 $0 \le x < y$ 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。

**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 ≥ $t2 或者 $s1 是负数则跳转到 IndexOutOfBounds。

**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
95 存器指定的地址。该指令将在下一节中介绍。

⊃ 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。

**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。

**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。

**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。

96 ⊃ 过程：根据提供的参数执行一定任务的存储的子程序。

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
```

97

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过程。
- 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”部分所要到的那样。

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）。

- 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。
- 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。
- 压栈：向栈中增加元素。
- 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。

**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;
}
```

98 编译后的MIPS汇编代码是什么呢？

**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：

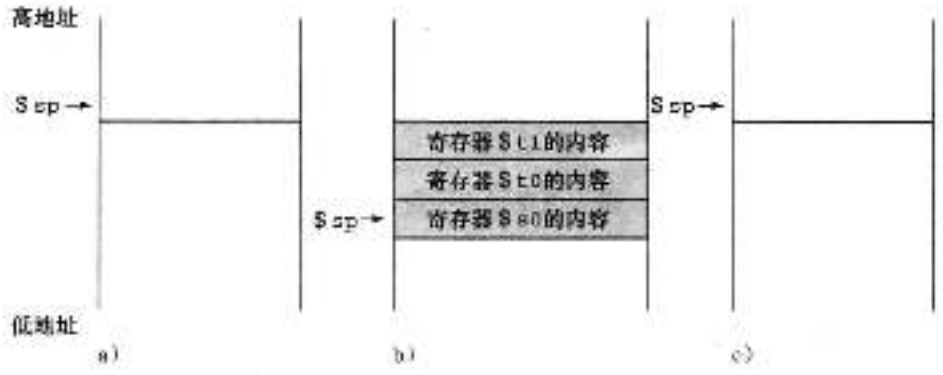



图 2-10 在过程调用之前（a）、之中（b）和之后（c），栈指针以及栈的值。栈指针总是指向栈顶，或者说是图中栈的最后一个字

```
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
100 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。

**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？

**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≥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)
```
101

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
```

□

**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。

⊃ 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。

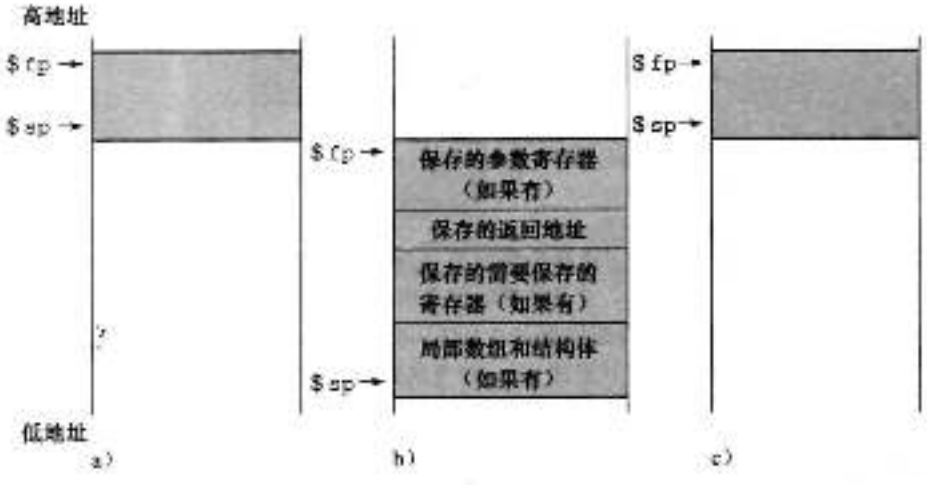



图 2-12 过程调用之前（a）、之中（b）、之后（c）栈的分配情况。帧指针（$fp）指向该帧的第一个字（一般是保存的参数寄存器），而栈指针（$sp）指向栈顶。栈可调整为有足够的空间来容纳所有的保存寄存器和驻留内存的局部变量。因为在程序运行期栈指针可能会改变，所以对于程序员而言，虽然使用栈指针和少量的地址运算就可能完成对变量的引用，但使用固定的帧指针引用变量会更为简单。如果在一个过程中栈内没有局部变量，编译器将可以不设置和不恢复帧指针以节省时间。当使用帧指针时，在调用中使用 $sp 的地址进行初始化，而 $sp 可以使用 $fp 来恢复。相关内容可以在 MIPS 参考数据卡的第 4 列找到

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。

- 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。
- 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运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增长，从而在两个段此消彼长的过程中达到内存的高效使用。

- 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。

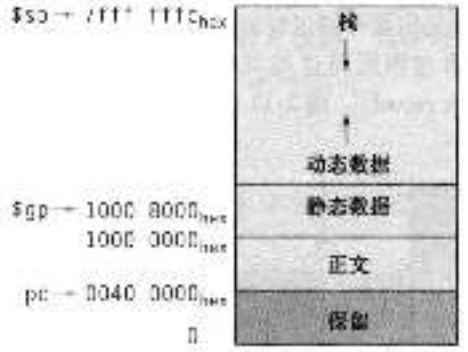



图 2-13 程序和数据的 MIPS 内存分配。这些地址只是一种软件规定，并非 MIPS 体系结构的一部分。栈指针初始化为 7fff fffc$_{16}$，并朝数据段的方向向下增长。在另一端，程序代码（代码段）从地址 0040 0000$_{16}$ 开始。静态数据从 1000 0000$_{16}$ 开始。然后是动态数据，在 C 中使用 malloc 命令分配，在 Java 中使用 new 命令来分配。动态数据在另一区域中朝着栈的方向向上生长，该区域称为堆。全局指针 $gp 应设置为适当地址以便于访问数据。它初始化为 1000 8000$_{16}$，这样通过相对 $gp 的正负 16 位的偏移量就可以访问从 1000 0000$_{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“悬挂指针”（dangling pointer），会造成指针指向程序不想访问的位置。
104 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

**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）。

**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）：

105

```
sum: slt1 $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
```

**小测验**

下面关于 C 和 Java 的描述哪些是正确的？

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

## 2.9 人机交互

*! ( @ | - > /www open tab at bar is great)*

——键盘诗《*Hatless Atlas*》的第4行，*1991*

（对ASCII字符的一些命名："!"是wow，"("是open，"|"是bar，等等）

发明计算机是为了数字计算，不过计算机很快被用于商业方面的文字处理。今天大多数计算机使用8位的字节来表示字符，也就是几乎每个人都遵循的ASCII（American Standard Code for Information Interchange）码。图2-15对ASCII进行了总结。

| ASCII值 | 字符 | ASCII值 | 字符 | ASCII值 | 字符 | ASCII值 | 字符 | ASCII值 | 字符 | ASCII值 | 字符 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | space | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | $ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | ( | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 | ) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | \| |
| 45 | - | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | DEL |

图2-15 字符的ASCII码表示。注意所有大写字母和对应小写字母的差均为32，这个观测结果可以得到一条检查和切换大小写的简便方法。没有给出的ASCII值包括格式化字符。例如，8代表退格，9代表tab字符，而13代表回车。另外一个有用的值0表示null，C编程语言用这个来标记字符串的结尾。这些内容可以在MIPS参考数据卡的第3列中找到

106

**例题** • ASCII码与二进制数对比

我们可以使用一串ASCII码而不用整数来表示数字。如果用ASCII码表示10亿这个数将比用32位整数表示增加多少存储呢？

**答案**

10亿就是1 000 000 000，需要使用10位ASCII码表示，每一个ASCII码都是8位长。所以存储将增长到（10×8）/32，即2.5倍。除了存储空间要增加外，用于对这些十进制数字进行加法、减法、乘法和除法的硬件的设计也是困难的。这些困难解释了为什么计算专家越来越相信使用二进制的计算机是自然的，而偶然出现的十进制计算机则是奇怪的。 □

可以使用一系列指令从一个字中提取出一个字节，所以字的读取和存储同样可以完成对字节的传输。然而，由于在某些程序中对文本的操作十分普遍，所以MIPS还提供字节传输指令。字节读取lb（load byte）指令从内存中读出一个字节，并将其放在一个寄存器最右边的8位。

字节存储 sb（store byte）指令把一个寄存器最右边的 8 位取出来然后写到内存中。这样，我们可以按下面的顺序复制一个字节：

```
lb $t0,0($sp)        # Read byte from source
sb $t0,0($sp)        # Write byte to destination
```

字符通常被组合为字符数目可变的字符串。表示一个字符串的方式有三种选择：1）保留字符串的第一个位置用于给出字符串的长度；2）附加一个带有字符串长度的变量（如在结构体中）；3）字符串最后的位置用一个字符来标识其结尾。C 语言使用第三种选择，用一个值为 0（ASCII 码中的 null）的字节来结束字符串。所以，字符串“Cal”在 C 中用 4 字节表示，用十进制表示分别为：67、97、108、0。（下面即将看到，Java 采用第一种表示方法。）

107

**例题：通过编译一个字符串复制过程，来展示如何使用 C 字符串**

strcpy 过程将 C 语言中约定使用 null 字节结束的字符串 y 复制到字符串 x：

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
    i += 1;
}
```

编译后的 MIPS 汇编代码是什么？

**答案**

下面是基本的 MIPS 汇编代码段。假定数组 x 和 y 的基地址在 $a0 和 $a1 中，而 i 在 $s0 中。strcpy 调整栈指针然后将保存的寄存器 $s0 保存在栈中。

```
strcpy:
    addi  $sp,$sp,-4   # adjust stack for 1 more item
    sw    $s0, 0($sp)  # save $s0
```

为了将 i 初始化为 0，下一条指令通过对 0 和 0 做加法并将和放到 $s0 中的方法将 $s0 置为 0：

```
add   $s0,$zero,$zero # i = 0 + 0
```

这是循环的开始。y［i］地址的形成是通过把 i 加到 y［］上：

```
L1: add   $t1,$s0,$a1  # address of y[i] in $t1
```

注意我们不必将 i 乘以 4，因为 y 是字节的数组而并非字的数组，和前面的例子一样。

为了读取 y［i］中的字符，我们使用无符号字节读取指令，将字符放入 $t2 中：

```
lbu   $t2, 0($t1)  # $t2 = y[i]
```

108

采用类似的计算方式将 x［i］的地址放在 $t3 中，然后将 $t2 中的字符保存到该地址中。

```
add   $t3,$s0,$a0  # address of x[i] in $t3
sb    $t2, 0($t3)  # x[i] = y[i]
```

接下来，如果字符是 0 则退出循环。也就是说，如果它是字符串的最后一个字符则退出：

```
beq   $t2,$zero,L2 # if y[i] == 0, go to L2
```

如果不是，将 i 加 1 继续循环：

```
addi  $s0, $s0,1   # i = i + 1
j     L1           # go to L1
```

如果不继续循环，那就是到了字符串的最后一个字符，我们还原 $s0 和栈指针，然后返回。

```
L2: lw    $s0, 0($sp)  # y[i] == 0: end of string.
                       # Restore old $s0
    addi  $sp,$sp,4    # pop 1 word off stack
    jr    $ra          # return
```

在C中字符串复制通常使用指针而不是数组，从而避免上面代码中对i的操作。详见2.14节数组和指针对比的相关解释。 □

由于strcpy是一个叶过程，编译器可以把i放在临时寄存器中以避免对$s0进行保存和恢复。因此，我们可以不把$t寄存器用做临时寄存器，而是将其用作被调用者可以方便使用的寄存器。当编译器遇到一个叶过程时，它会在用完所有临时寄存器之后，才使用那些必须保存的寄存器。

## Java中的字符和字符串

Unicode是大多数人类语言中字母的通用编码。图2-16是一个Unicode字母表的示例，Unicode中字母数和ASCII编码中有用的字符数一样多。为了更有包容性，Java对字符使用Unicode，它默认使用16位来表示一个字符。

109

| Latin | Malayalam | Taghanwa | General Punctuation |
|---|---|---|---|
| Greek | Sinhala | Khmer | Spacing Modifier Letters |
| Cyrillic | Thai | Mongolian | Currency Symbols |
| Armenian | Lao | Limbu | Combining Diacritical Marks |
| Hebrew | Tibetan | Tai Le | Combining Marks for Symbols |
| Arabic | Myanmar | Kangxi Radicals | Superscripts and Subscripts |
| Syriac | Georgian | Hiragana | Number Forms |
| Thaana | Hangul Jamo | Katakana | Mathematical Operators |
| Devanagari | Ethiopic | Bopomofo | Mathematical Alphanumeric Symbols |
| Bengali | Cherokee | Kanbun | Braille Patterns |
| Gurmukhi | Unified Canadian Aboriginal Syllabic | Shavian | Optical Character Recognition |
| Gujarati | Ogham | Osmanya | Byzantine Musical Symbols |
| Oriya | Runic | Cypriot Syllabary | Musical Symbols |
| Tamil | Tagalog | Tai Xuan Jing Symbols | Arrows |
| Telugu | Hanunoo | Yijing Hexagram Symbols | Box Drawing |
| Kannada | Buhid | Aegean Numbers | Geometric Shapes |

图2-16 Unicode字母表示例。Unicode 4.0版本有超过160个"块"，每个块是一个符号集的名字，且是16的幕数倍。例如，希腊字符（Greek）从$0370_{16}$开始，西里尔字符（Cyrillic）从$0400_{16}$开始。前三列以Unicode的数字顺序粗略地列出了48个块对应的48种人类语言。最后一列中的16个块是多种语言，并没有按照顺序排列。默认的是16位编码，称为UTF-16。一种称为UTF-8的变长编码，将ASCII子集保持为8位，其余字符用16或32位来表示。UTF-32使用32位表示一个字符。更多内容请参见www.unicode.org

MIPS指令集包含显式的读取和存储16位半字（halfword）的指令。读取半字指令lh（load half）从存储器中读出一个半字，然后将其放在寄存器的最右边16位。与读取字节类似，读取半字指令lh也将半字看作有符号数并进行符号扩展，以填充寄存器左侧的16位。而无符号读取半字指令lhu（load halfword unsigned）将半字看作无符号数，与lh相比，这条指令更加常用。存储半字指令sh（store half）将寄存器最右边的16位写入存储器。我们按照下面的序列来复制半字：

```
lhu $t0,0($sp) # Read halfword (16 bits) from source
sh $t0,0($gp)  # Write halfword (16 bits) to destination
```

字符串是一个标准的 Java 类，它对连接、比较、转换的方法提供了专门的内建支持和预定义方法。与 C 不同的是，Java 包含一个字来给出字符串长度，这和 Java 数组相似。 110

**精解** MIPS 软件试图保持栈和字地址的对齐，这样就允许程序总是使用 lw 和 sw（要求必须是对齐的）来访问栈。这一约定意味着一个 char 类型变量在栈中被分配 4 字节，尽管它并不需要这么多。然而，一个 C 字符串变量或一个字节数组会把每 4 字节压缩为 1 个字，而一个 Java 字符串变量或 short 类型数组会把每 2 个半字压缩为 1 个字。

**精解** 为了反映 web 的全球性特征，当今的大部分 web 页面采用 Unicode，而非 ASCII。

**小测验**

Ⅰ. 下面关于 C 和 Java 中字符和字符串的陈述哪些是正确的？

1. C 中一个字符串占用的内存是 Java 中同样字符串的一半。
2. C 和 Java 中字符串只是一个一维字符数组的非正规名字。
3. C 和 Java 中采用 null（0）来标记字符串的结尾。
4. 对字符串的操作，例如求长度，在 C 中比在 Java 中更快。

Ⅱ. 下面哪种类型的变量存放 $1\,000\,000\,000_{10}$ 占用的内存空间最大？

1. C 语言的 `int`
2. C 语言的 `string`
3. Java 的 `string`

## 2.10 MIPS 中 32 位立即数和寻址

虽然保持所有 MIPS 指令为 32 位长简化了硬件，但有时使用 32 位常量或 32 位地址更加方便。本节先介绍使用较大常量的一般解决方法，然后描述了用于分支和跳转指令寻址的优化措施。 111

### 2.10.1 32 位立即数

尽管常数往往比较短而且适于 16 位字段，但有时它们会更大。MIPS 指令集中的读取立即数高位指令 `lui`（load upper immediate）专门用于设置寄存器中常数的高 16 位，允许后续指令设置常数的低 16 位。图 2-17 描述了 `lui` 的操作。

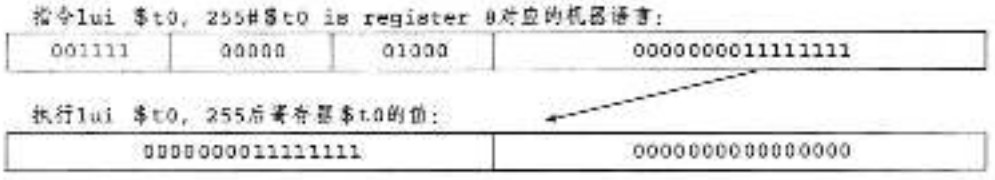



图 2-17 `lui` 指令的效果。`lui` 指令将 16 位立即数常量存放到寄存器的高 16 位，低 16 位用 0 填充

**例题：加载 32 位常量**

加载下面这个 32 位常量到寄存器 $s0 的 MIPS 汇编代码是什么？

```
0000 0000 0011 1101 0000 1001 0000 0000
```

**答案**

首先，我们使用命令 `lui` 加载高 16 位，十进制表示是 61：

```
lui $s0, 61   # 61 decimal = 0000 0000 0011 1101 binary
```

执行上面的指令后，寄存器 $s0 的值为

```
0000 0000 0011 1101 0000 0000 0000 0000
```

下一步是插入低 16 位，十进制表示是 2304：

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

寄存器 $s0 中的最终值就是所需要的值：

112

```
0000 0000 0011 1101 0000 1001 0000 0000
```

□

**硬件/软件接口** 编译器或汇编程序必须把大的常数分解为若干小的常数，然后再合并到一个寄存器中。正如你想象的那样，立即数字段大小的限制，无论在取/存数指令中对存储器的地址还是在立即数指令中对常数都可能带来问题。如果这项工作由汇编程序来做，如 MIPS 软件，那么汇编程序必须有一个可用的临时寄存器来创建长整数值。这是给汇编程序保留 $at 寄存器的一个原因。

因此，MIPS 机器语言的符号表示不再受到硬件限制，但仍受汇编程序构造者所选择包括的内容的限制（见 2.12 节）。我们以靠近硬件层的方式解释计算机的体系结构，需要注意的是，我们所使用汇编程序的增强扩展语言，在实际处理器中是不存在的。

**解释** 构造 32 位常数时必须小心。指令 addi 将指令最左边的 16 位立即数字段复制到一个字的高 16 位中。2.6 节的立即数逻辑或操作（logical or immediate）把 0 送到高 16 位中，所以可被汇编程序用于和 lui 一起创建 32 位常数。

## 2.10.2 分支和跳转中的寻址

MIPS 跳转指令寻址采用最简单的寻址方式。它们使用最后一种 MIPS 指令格式，称为 J 型。J 型除了 6 位操作码之外，其余位都是地址字段。所以

```
j   10000   # go to location 10000
```

可以汇编为下面的格式（实际中要更加复杂一些，正如我们后面将看到的那样）：

| 2 | 10000 |
|---|---|
| 6位 | 26位 |

其中跳转操作码的值为 2，跳转地址为 10000。

和跳转指令不同，条件分支指令除了规定分支地址之外还必须指定两个操作数。因此

```
bne  $s0,$s1,Exit  # go to Exit if $s0 ≠ $s1
```

被汇编为下面的指令，只保留了 16 位用于指定分支地址：

113

| 5 | 16 | 17 | Exit |
|---|---|---|---|
| 6位 | 5位 | 5位 | 16位 |

如果让程序地址适应该 16 位字段，则意味着任何程序都不能大于 $2^{16}$，这在今天来说太小，因此是一种很不现实的选择。另一个可选的办法是指定一个总是加到分支地址上的寄存器，这样分支指令的地址可按如下方式计算：

$$\text{程序计数器} = \text{寄存器} + \text{分支地址}$$

这个求和结果允许程序的大小达到 $2^{32}$，并且仍能使用条件分支，从而解决了分支地址大小的问题。随之而来的问题是使用哪个寄存器呢？

答案取决于条件分支是如何使用的。条件分支在循环和 if 语句中都可以找到，它们倾向于转到附近的指令。例如，在 SPEC 基准测试程序中，大概一半条件分支的跳转距离小于 16 条指令。因为程序计数器（Program Counter，PC）包含当前指令的地址，如果我们使用 PC 来作为

增加地址的寄存器，我们可转移到离当前指令距离为 $\pm 2^{15}$ 个字的地方。几乎所有循环和 if 语句都远远小于 $2^{15}$ 个字，因此 PC 是一个理想的选择。

这种分支寻址形式称为 **PC 相对寻址**（PC-relative addressing）。正如在第 4 章中将会看到的那样，提前递增 PC 来指向下一条指令会对硬件带来很多方便。所以，MIPS 寻址实际上是相对于下一条指令的地址（PC + 4），而不是相对于当前指令（PC）。寻址附近的指令是**加速大概率事件**的另外一个例子。

⊃ PC 相对寻址：一种寻址方式，它将 PC 和指令中的常数相加作为寻址结果。

像近期的大多数计算机一样，MIPS 对所有条件分支使用 PC 相对寻址，因为这些指令的跳转目标一般都比较接近其分支地址。另一方面，跳转链接指令并非总是靠近调用者的过程，所以它们通常使用其他寻址方式。因此，MIPS 体系结构通过使用跳转和跳转链接指令的 J 型格式来为过程调用提供长地址。

因为所有 MIPS 指令都是 4 字节长，所以在 PC 相对寻址时所加的地址被设计为字地址而不是字节地址。相对于 16 位的字节地址，16 位的字地址跳转范围扩大了 4 倍。同样，跳转指令的 26 位字段也是字地址，它可以表示 28 位的字节地址。

**01 精解** 因为 PC 是 32 位，所以有 4 位必须来自于跳转指令之外的其他地方。MIPS 跳转指令仅仅代替 PC 的低 28 位，而高 4 位保持不变。装载器和链接器（见 2.12 节）必须十分小心以避免程序超过 256 MB 的寻址界限（6 400 万条指令）；否则，该跳转必须替换为寄存器跳转指令，并在执行前使用其他指令将完整的 32 位地址加载到一个寄存器中。 114

**01 例题 - 在机器语言中描述分支偏移**

假设 2.7.1 节的 while 循环语句被编译成下面的 MIPS 汇编代码：

```
Loop: sll  $t1,$s3,2      # Temp reg $t1 = 4 * i
      add  $t1,$t1,$s6    # $t1 = address of save[i]
      lw   $t0,0($t1)     # Temp reg $t0 = save[i]
      bne  $t0,$s5, Exit  # go to Exit if save[i] ≠ k
      addi $s3,$s3,1      # i = i + 1
      j    Loop           # go to Loop
Exit:
```

如果我们假设把 loop 的开始位置放在内存的 80 000 处，那么该循环的 MIPS 机器代码是什么呢？

**01 答案**

汇编指令和它们的地址如下：

| | | | | | | |
|---|---|---|---|---|---|---|
| 80000 | 0 | 0 | 19 | 9 | 2 | 0 |
| 80004 | 0 | 9 | 22 | 9 | 0 | 32 |
| 80008 | 35 | 9 | 8 | 0 | | |
| 80012 | 5 | 8 | 21 | 2 | | |
| 80016 | 8 | 19 | 19 | 1 | | |
| 80020 | 2 | 20000 | | | | |
| 80024 | ... | | | | | |

注意 MIPS 指令使用字节寻址，所以相邻字的地址相差 4，即一个字中的字节的数量。第 4 行的 bne 指令将 2 个字或是 8 字节加到下一条指令地址（80016）上，使用相对下一条指令的偏移（8 + 80016）指明跳转目标，而不是使用相对该分支指令的偏移（12 + 80012），也不是使用完整的目的地址（80024）。最后一行的跳转指令采用完整的地址（20 000 × 4 = 80 000），

115 对应于 Loop 标签。 □

**硬件/软件接口** 大多数条件分支都转移到一个附近的位置，但有时也会转移很远，距离超过条件分支指令的16位可以表示的范围。汇编器实现方法就像处理大立即数或大常数的方法一样：插入一个跳转到分支目标的无条件跳转，并将条件取反以便由分支决定是否跳过该无条件跳转指令。

**例题 • 远距离的分支转移**

假设在寄存器 $s0 与寄存器 $s1 值相等时需要跳转，可以使用如下指令：

```
beq    $s0, $s1, L1
```

用两条指令替换上面的指令，以获得更远的转移距离。

**答案**

可用下面的指令替换短地址的条件分支指令：

```
      bne    $s0, $s1, L2
      j      L1
L2:
```

□

## 2.10.3 MIPS 寻址模式总结

多种不同的寻址形式一般统称为**寻址模式**（addressing mode）。图 2-18 给出了每种寻址模式的操作数如何识别。MIPS 寻址模式如下所示：

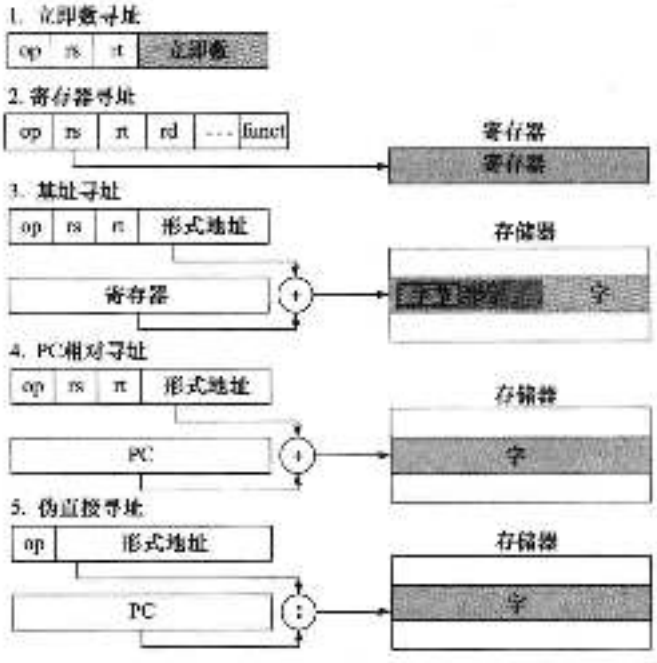



图 2-18 MIPS 5 种寻址模式的说明。阴影部分为操作数。模式 3 的操作数在内存中，而模式 2 的操作数是寄存器。注意，读数和存数对字节、半字或字有多种版本。模式 1 的操作数是指令自身的 16 位字段。模式 4 和模式 5 寻址的指令在内存中，模式 4 把 16 位地址左移 2 位与 PC 相加，而模式 5 把 26 位地址左移 2 位与 PC 计数器的高 4 位相连。注意，一种操作可能可以使用多种寻址模式，例如，加法可以使用立即数寻址（addi）和寄存器寻址（add）

寻址模式：根据对操作数和/或地址的使用不同加以区分的多种寻址方式中的一种。

1）立即数寻址（immediate addressing），操作数是位于指令自身中的常数。

2）寄存器寻址（register addressing），操作数是寄存器。

3）基址寻址（base addressing）或偏移寻址（displacement addressing），操作数在内存中，其地址是指令中基址寄存器和常数的和。

4）PC 相对寻址（PC-relative addressing），地址是 PC 和指令中常数的和。

5）伪直接寻址（pseudodirect addressing），跳转地址由指令中 26 位字段和 PC 高位相连而成。 116

**硬件/软件接口** 虽然我们把 MIPS 系统结构按 32 位地址描述，但是几乎所有的微处理器（包括 MIPS）都能进行 64 位地址扩展（见附录 E 和 2.18 节）。这些扩展主要是针对大型程序的需要。指令集的扩展使得体系结构发展的同时，保持软件和下一代体系结构的向上兼容性。 117

## 2.10.4 机器语言解码

有时候必须通过逆向工程将机器语言恢复到最初的汇编语言，比如检查“核心转储”（core dump）时。图 2-19 描述了 MIPS 机器语言对各个字段的编码。该图可用于汇编语言和机器语言之间的手动翻译。

**例题：机器码解码**

下面这条机器指令对应的汇编语言语句是什么？

```
00af8020hex
```

**答案**

第一步是将十六进制转换到二进制，以便找到操作码字段：

```
(Bits: 31 28 26                          5   2 0)
       0000 0000 1010 1111 1000 0000 0010 0000
```

我们查看操作码字段来决定指令的操作类型。参照图 2-19，当 31～29 位是 000 且 28～26 位也是 000 时，它是 R 型指令。参照图 2-20，将该二进制指令按照 R 型指令字段重新排列：

```
op      rs      rt      rd      shamt   funct
000000  00101   01111   10000   00000   100000
```

| | op(31:26) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 28～26<br>31～29 | 0(000) | 1(001) | 2(010) | 3(011) | 4(100) | 5(101) | 6(110) | 7(111) |
| 0(000) | R 格 | Bltz/gez | 跳转 | 跳转并链接 | 分支 eq | 分支 ne | blez | bgtz |
| 1(001) | 立即数加法 | addiu | 小于 0 即数置位 | 小于无符号立即数时置位 | andi | ori | xori | 取立即数高位 |
| 2(010) | TLB | FlPt | | | | | | |

图 2-19 MIPS 指令解码。该标记根据行和列确定字段的值。例如，图的顶部在第 4 行（指令的第 31～29 位为 $100_2$）第 3 列（指令的第 28～26 位为 $011_2$）描述了取字指令，因此相应操作码字段（第 31～26 位）的（R 型）值是 $100011_2$。下划线表示该字段在其他地方被使用。例如，第 0 行第 0 列（op = $000000_2$）的 R 型在图的底部定义。因此，底部第 4 行第 2 列的 subtract 意味着指令 funct 字段（第 5～0 位）是 $100010_2$ 而操作码字段（第 31～26 位）是 $000000_2$。第 2 行第 1 列的 FlPt 在第三章的图 3－18 中定义。Bltz/gez 是附录 B 中 4 条指令的操作码：bltz、bgez、bltzal 和 bgezal。附录 A 涵盖所有的指令

| op(31:26) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3(011) | | | | | | | | |
| 4(100) | 取字节 | 取半字 | lwl | 取字 | 取无符号字节 | 取无符号半字 | lwr | |
| 5(101) | 存字节 | 存半字 | swl | 存字 | | | swr | |
| 6(110) | 取链接字 | ll | | | | | | |
| 7(111) | 存条件字 | sc | | | | | | |

| op(31:26) = 010000(协处), rs(25:21) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 23~21<br>25~24 | 0(000) | 1(001) | 2(010) | 3(011) | 4(100) | 5(101) | 6(110) | 7(111) |
| 0(00) | mfc0 | | cfc0 | | mtc0 | | ctc0 | |
| 1(01) | | | | | | | | |
| 2(10) | | | | | | | | |
| 3(11) | | | | | | | | |

| op(31:26) = 000000(R型), funct(5:0) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2~0<br>5~3 | 0(000) | 1(001) | 2(010) | 3(011) | 4(100) | 5(101) | 6(110) | 7(111) |
| 0(000) | 逻辑左移 | | 逻辑右移 | sra | sllv | | srlv | srav |
| 1(001) | jump register | jalr | | | syscall | break | | |
| 2(010) | mfhi | mthi | mflo | mtlo | | | | |
| 3(011) | mult | multu | div | divu | | | | |
| 4(100) | add | addu | subtract | subu | and | or | xor | not or(nor) |
| 5(101) | | | set l t | set l t<br>无符号操作 | | | | |
| 6(110) | | | | | | | | |
| 7(111) | | | | | | | | |

图 2-19 （续）

| 名称 | 字段 | | | | | | 备注 |
|---|---|---|---|---|---|---|---|
| 字段大小 | 6 位 | 5 位 | 5 位 | 5 位 | 5 位 | 6 位 | 所有 MIPS 指令都是 32 位 |
| R 型 | op | rs | rt | rd | shamt | funct | 算术指令型 |
| I 型 | op | rs | rt | 地址/立即数 | | | 传输、分支和立即数型 |
| J 型 | op | 目标地址 | | | | | 跳转指令型 |

图 2-20 MIPS 指令的格式

图 2-19 的底部确定了 R 型指令的操作。在本例中，5～3 位是 100，2～0 位是 000，因此该二进制指令为 add 指令。

下面我们通过查找字段值来解码指令的剩余部分。rs 字段的十进制值是 5，rt 是 15，rd 是 16（shamt 未使用）。图 2-14 说明这些数字表示寄存器 $a1、$t7 和 $s0。现在可以给出转换后的汇编指令：

```
add $s0,$a1,$t7
```

图 2-20 给出了所有 MIPS 指令的格式。第 2.2 节的图 2-1 汇总了本章出现的所有汇编指令。其他 MIPS 指令主要处理算术运算和实数，将在第 3 章介绍。 □

**01** 小测验

Ⅰ. 在 MIPS 中条件分支的地址范围（K = 1024）是多大？

1. 地址在 0 ~ 64K - 1 之间
2. 地址在 0 ~ 256K - 1 之间
3. 分支前后地址范围各大约 32K
4. 分支前后地址范围各大约 128K

Ⅱ. 在 MIPS 中跳转和跳转链接指令的地址范围（M = 1024K）是多大？

1. 地址在 0 ~ 64M - 1 之间
2. 地址在 0 ~ 256M - 1 之间
3. 分支前后地址范围各大约 32M
4. 分支前后地址范围各大约 128M
5. 由 PC 提供高 6 位地址的 64M 大小的块中任意地址
6. 由 PC 提供高 4 位地址的 256M 大小的块中任意地址

Ⅲ. 机器指令 $0000\ 0000_{16}$ 对应的 MIPS 汇编语言指令是什么？

1. j
2. R 型
3. addi
4. sll
5. mfc0
6. 未定义的操作码：没有对应 0 的合法指令

118 ~ 120

## 2.11 并行与指令：同步

当任务之间相互独立的时候，任务的并行执行是比较容易的。但往往任务之间需要相互协作，这种协作通常意味着某些任务写的结果是其他任务需要读取的值。这时执行读任务的一方要知道写任务什么时候完成了写操作，才能安全地读回数据。就是说，任务之间需要同步（synchronize），否则就有发生**数据竞争**（data race）的危险，导致读数据错误而引起程序运行结果的改变。

➲ 数据竞争：假如来自不同线程的两个访存请求访问同一个地址，它们连续出现，并且至少其中一个是写操作，那么这两个存储访问形成数据竞争。

例如，回忆第 1 章 1.8 节所提到的 8 个记者共同写作一个故事的例子。假设一个记者要写总结，他要阅读所有之前的章节。因此，他必须知道其他记者什么时候可以完成各自的章节，然后他再撰写总结，这样他就不用担心写好总结后其他记者再对各自章节进行修改。所以，他们就需要很好地同步各个章节撰写和阅读的过程，这样总结才能和前面章节中所写的内容相一致。

在计算中，同步机制要依赖硬件提供的同步指令，这些指令可由用户调用。本节我们重点讨论加锁（lock）和解锁（unlock）同步操作的实现。采用加锁和解锁可以直接创立一个仅允许单个处理器操作的区域，叫作互斥（mutual exclusion）区。更复杂的同步机制实现也与此类似。

在多处理器中实现同步需要一组硬件原语，提供对存储单元进行原子读和原子写的能力，使得在进行存储器原子读或原子写操作时任何其他操作都不得插入。如果没有这样的硬件原语，那么建立同步机制的代价将会变得很高，并且随着处理器数量的增加情况将更为恶化。

建立基本硬件原语有若干可选的方案，这些方案都可以实现原子读和原子写的功能，并能

用某种方法表示这些操作是否为原子操作。通常，体系结构设计人员并不希望基本硬件原语被用户使用，而是希望这些原语被系统程序员用来建立同步库，建立同步库的过程通常很复杂且难度较大。

我们用原子交换原语（atomic exchange 或 atomic swap）来演示如何建立基本同步机制。这个原语是将寄存器中的一个值和存储器中的一个值相互交换。

为了展示该原语建立同步原理的基本过程，假定使用存储器中某个单元来表示一个锁变量：其数值为0时表示解锁，为1时表示加锁。一个处理器尝试对锁单元加锁，方法是用一个寄存器中的1与该锁单元的值进行交换。交换以后该锁单元的新值为1，返回值（锁单元的原
121 值）如果是1，表明这个锁已被其他处理器占用；否则返回值为0，表示锁是自由的，尝试加锁成功。此时锁单元已被修改成1，以防止任何其他处理器再来占用。

例如，考虑有两个处理器同时尝试进行交换操作，它们的竞争关系就会被破坏。因为其中只能有一个处理器先执行交换操作，并且返回0。那么第二个处理器执行完交换操作的时候返回值就变成了1。用交换原语实现同步的关键是操作的原子性：交换操作是不可分割的，并且由硬件对两个同时执行的交换操作进行排序。有可能两个处理器同时尝试置位同步变量，但这两个处理器认为它们同时成功设置了同步变量是不可能的。

实现单个的原子存储器操作给处理器的设计者带来了若干挑战，因为这要求存储器的读、写操作都是有单条不可被中断的指令完成。

一种可行的方法是采用指令对，其中第二条指令返回一个表明这对指令是否原子执行的标志值。假如处理器的操作都是在这对指令之前或之后执行，这对指令就是原子的。因此，当一个指令对是原子的，没有哪个处理器能改变这两个指令执行之间的数据值。

在MIPS处理器中这一指令对包括一条叫作链接取数（load linked）的特殊取数指令和一条叫作条件存数（store conditional）的特殊存数指令。我们顺序地使用这两条指令：如果链接取数指令所指定的锁单元的内容在相同地址的条件存数指令执行前已被改变，那么条件存数指令就执行失败。我们定义条件存数指令完成以下功能：保存寄存器的值，并且如果执行成功则将寄存器的值修改为1，如果失败则修改为0。因为链接取数指令返回锁单元的原始值，条件存数指令执行成功的时候才返回1，下面的指令序列实现了存储器单元的原子交换。存储器单元的地址由$s1中的值指出。

```
again: addi $t0,$zero,1      ;copy locked value
       ll   $t1,0($s1)       ;load linked
       sc   $t0,0($s1)       ;store conditional
       beq  $t0,$zero,again  ;branch if store fails
       add  $s4,$zero,$t1    ;put load value in $s4
```

在ll和sc两条指令之间的任何时候有处理器插入，并修改了该锁单元的值，指令sc都会将$t0置为0，引起这段指令序列重新执行。在指令序列的最后，寄存器$s4中的值和$s1指向的锁单元的值发生了原子交换。

**Q1 精解** 尽管我们讲述的同步是在多处理器系统中的，但是原子操作对于单个处理器上运行的操作系统在处理多个进程时也是十分有用的。在单处理器中，为了保证执行不被任何事件所干扰，条件存数指令在处理器两条指令之间进行上下文切换（context switch）时也会
122 失败（见第5章）。

链接取数/条件存数机制的优点是：可以通过它们来构造其他的诸如原子比较和交换（atomic compare and swap）或者原子取后加（atomic fetch-and-increment）等同步原语。这些同步原语可以被用在一些并行编程模型中。这些同步原语的实现需要在ll指令和sc指令之间插入更多的指令，但不需要太多。

因为在链接取数指令执行之后，任何试图修改锁单元值的操作或者任何异常都将导致条件存数指令执行失败，所以在选择 ll 和 sc 之间的指令时就要格外注意。特别需要注意的是，允许使用的并且不会造成可预约只有寄存器－寄存器指令，而处理器可能由于重复的页错误而导致始终无法完成 sc 指令，从而使处理器处于一种死锁的状态。另外，链接取数和条件存数之间的指令数一定要尽可能少，这样才可以减少不相关的事件或者竞争资源的处理器所引起条件存数指令执行失败的频率。

**小测验**

什么时候才会用到像链接取数（load linked）和条件存数（store conditional）这样的原语？

1. 当一个并行程序中相互协作的线程需要同步以获得对共享数据的正确的读写行为时
2. 当运行在单处理器上的相互协作的处理过程需要同步以获得对共享数据的正确的读写行为时

## 2.12 翻译并执行程序

本节描述了将存储在硬盘文件中的 C 程序转换为可执行程序的 4 个步骤，图 2-21 所示是语言翻译的层次。尽管某些系统可能合并部分步骤以减少转换时间，但从逻辑上讲，这 4 个步骤是程序转换流程所必经的 4 个阶段。本节将根据这种翻译层次进行描述。

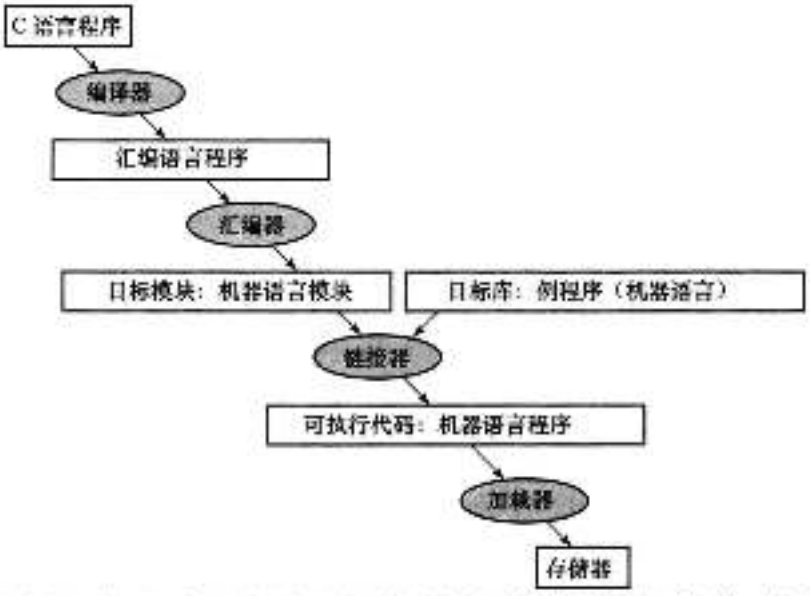



图 2-21 C 语言的翻译层次。用高级语言编写的程序首先需要被编译成为汇编语言，然后被汇编成机器语言组成的目标模块。链接器将多个模块和库程序组合在一起解析所有的引用。加载器将可执行程序加载到内存的适当位置，然后处理器就可以执行了。为了加快翻译的速度，某些步骤被跳过或和其他步骤组合在一起。一些编译器直接产生目标模块，一些系统使用带链接功能的加载器直接完成后面两步。为了确定文件的类型，UNIX 使用文件的后缀，x.c 代表 C 源文件，x.s 表示汇编文件，x.o 表示目标文件，x.a 表示静态链接库，x.so 表示动态链接库，默认情况下，a.out 表示可执行文件。MS-DOS 使用后缀 .C，.ASM，.OBJ，.LIB，.DLL 和 .EXE 来完成同样的功能

### 2.12.1 编译器

编译器将 C 程序转换成一种机器能理解的符号形式的汇编语言程序（assembly language

program)。高级语言编写的程序比使用汇编语言编写的代码少得多，所以程序员效率更高。

1975年，因为存储器容量较小并且编译器效率不高，所以许多操作系统和汇编器都用**汇编语言**（assembly language）编写。如今单DRAM芯片容量增长500 000倍，减轻了人们对程序大小的关注，并且今天优化的编译器能够产生出几乎与一个汇编语言专家所写的程序一样好的汇编程序，对于大型程序有时甚至效果更好。

⇒ 汇编语言：一种符号语言，能被翻译成二进制的机器语言。

## 2.12.2 汇编器

因为汇编语言对于高层次软件是一个接口，所以汇编器也能够处理一些机器语言指令的常见变种，就像这些变种是它自己的指令一样。硬件不需要实现这些指令，然而它们在汇编语言中的存在简化了程序转换和编程。这类指令称为**伪指令**（pseudoinstruction）。

⇒ 伪指令：汇编语言指令的一个变种，通常被看作一条汇编指令。

如前所述，MIPS硬件确保寄存器$zero保持0值。即一旦使用寄存器$zero，它都提供0，而且程序员不能修改寄存器$zero的值。寄存器$zero用于生成汇编语言指令move，move的功能是将一个寄存器中的内容复制到另一个中。因此即使MIPS体系结构中不存在这条指令，MIPS汇编器也能够识别它：

123
|
124

```
move $t0,$t1        # register $t0 gets register $t1
```

汇编器将这条汇编语言指令转换成功能等价的如下机器语言指令：

```
add $t0,$zero,$t1 # register $t0 gets 0 + register $t1
```

在2.7.1节的例子中提到，MIPS汇编器将blt（branch on less than，小于则分支）转换成两条指令：slt和bne。其他例子包括bgt、bge和ble。它也将一个到远距离的分支指令拆成一个分支指令和一个跳转指令。如前所述，MIPS汇编器允许将32位常量加载到一个寄存器中，不用考虑立即数指令的16位限制。

总的来说，伪指令使MIPS剧有比硬件所实现的更为丰富的汇编语言指令集。唯一的代价是保留了一个由汇编器使用的寄存器$at。如果你打算写汇编程序，请使用伪指令来简化你的任务。为了理解MIPS体系结构并保证获得最好的性能，可以学习图2-1和图2-19中真正的MIPS指令。

汇编器同样接受不同基数的数字。除了二进制和十进制，它们通常还使用比二进制更为紧凑而又容易转化为位模式的基数。MIPS汇编器使用十六进制。

这种特性相当方便，但是汇编器的主要任务是汇编成机器代码。汇编器将汇编语言程序转换成目标文件（object file），它包括机器语言指令、数据和指令正确放入内存所需要的信息。

为了产生汇编语言程序中每条指令对应的二进制表示，汇编器必须处理所有标号对应的地址。汇编器将分支和数据传输指令中用到的标号都放入一个**符号表**（symbol table）中。正如你所想的，这个表由标号和地址成对构成。

⇒ 符号表：一个用来匹配标记名和指令所在内存字的地址的列表。

UNIX系统中的目标文件通常包含以下6个不同的部分：

- 目标文件头，描述目标文件其他部分的大小和位置。
- 代码段，包含机器语言代码。
- 静态数据段，包含在程序生命周期内分配的数据。（UNIX系统允许程序使用静态数据，它

存在于整个程序中；也允许使用动态数据，它随程序的需要而增长或缩小。见图 2-13。）

- 重定位信息，标记了一些在程序加载进内存时依赖于绝对地址的指令和数据。
- 符号表，包含未定义的剩余标记，如外部引用。 125
- 调试信息，包含一份说明目标模块如何编译的简明描述，这样，调试器能够将机器指令关联到 C 源文件，并使数据结构也变得可读。

下一小节描述了如何链接已经汇编完成的子程序，如库程序。

### 2.12.3 链接器

到目前为止我们所描述的内容表明，对于源程序任意一行代码的修改都需要重新编译和汇编整个程序。全部重新翻译是对计算资源的严重浪费。这种重复对于标准库程序尤为浪费，因为程序员要编译和汇编那些在定义上几乎从未改变过的过程。另一种方法是单独编译和汇编每个过程，以使得某一行代码的改变只需要编译和汇编一个过程。这种方法需要一个新的系统程序，称为**链接编辑器**或（link editor）**链接器**（linker），它把所有独立汇编的机器语言程序“拼接”在一起。

⊃ 链接器：也称链接编辑器。它是一个系统程序，把各个独立汇编的机器语言程序组合起来并且解决所有未定义的标记，最后生成可执行文件。

链接器的工作分 3 个步骤：

1）将代码和数据模块象征性地放入内存。

2）决定数据和指令标签的地址。

3）修补内部和外部引用。

链接器使用每个目标模块中的重定位信息和符号表，来解析所有未定义标签。这种引用发生在分支指令、跳转指令和数据寻址处，所以这个程序的工作非常像一个编辑器：它寻找所有旧地址并用新地址取代它们。编辑是“链接编辑器”或链接器名字的简称。采用链接器的原因是修补代码比重新编译和汇编要快得多。

如果所有外部引用都解析完，链接器接着决定每个模块将要占用的内存位置。回忆 2.8.4 节的图 2-13，它描述了 MIPS 在内存中为程序和数据分配空间的方式。因为文件是单独汇编的，所以汇编器不可能知道该模块的指令和数据相对于其他模块而言将会被放到哪里。当链接器将一个模块放到内存中的时候，所有绝对引用（absolute reference），即与寄存器无关的内存地址必须重定位以反映它的真实地址。

链接器产生一个**可执行文件**（executable file），它可以在一台计算机上运行。通常，这个文件与目标文件具有相同的格式，但是它不包含未解决的引用。具有部分链接的文件是可能的，如库程序，在目标文件中仍含有未解决的地址。

⊃ 可执行文件：一个具有目标文件格式的功能程序，不包含未解决的引用。它可以包含符号表和调试信息。“剥离的可执行程序”不包含这些信息，可能包含加载器所需的重定位信息。 126

**例题 · 目标文件的链接**

将下面的两个目标文件链接。给出最终可执行文件中前几条指令对应的更新过的地址。为了便于理解，我们使用汇编语言来表示指令，在实际文件中，这些指令由数字表示。

注意目标文件中，我们已将必须在链接进程中更新的地址和标记高亮显示了，分别是引用过程 A 和过程 B 的地址的指令，以及引用数据字 X 和 Y 的地址的指令。

| 目标文件头 | | | |
|---|---|---|---|
| | 名字 | 过程A | |
| | 正文大小 | $100_{16}$ | |
| | 数据大小 | $20_{16}$ | |
| 代码段 | 地址 | 指令 | |
| | 0 | lw $a0,0($gp) | |
| | 4 | jal 0 | |
| | … | … | |
| 数据段 | 0 | (X) | |
| | … | … | |
| 重定位信息地址 | 地址 | 指令类型 | 依赖 |
| | 0 | lw | X |
| | 4 | jal | B |
| 符号表 | 标记 | 地址 | |
| | X | | |
| | B | — | |
| 目标文件头 | | | |
| | 名字 | 过程B | |
| | 正文大小 | $200_{16}$ | |
| | 数据大小 | $30_{16}$ | |
| 代码段 | 地址 | 指令 | |
| | 0 | sw $a1,0($gp) | |
| | 4 | jal 0 | |
| | … | … | |
| 数据段 | 0 | (Y) | |
| | … | … | |
| 重定位信息地址 | 地址 | 指令类型 | 依赖 |
| | 0 | sw | Y |
| | 4 | jal | A |
| 符号表 | 标记 | 地址 | |
| | Y | — | |
| | A | — | |

127

**答案**

过程A需要找到lead指令中标号为X的变量的地址和jal指令中过程B的地址。过程B需要找到store指令中标号为Y的变量的地址和jal指令中过程A的地址。

从2.8.4节的图2-13中，我们可以看到代码段从地址 $40\ 0000_{16}$ 开始而数据段从地址 $1000\ 0000_{16}$ 开始。过程A的正文被放置在第一个地址而它的数据被放置在第二个地址。过程A的目标文件头表明其代码段大小是 $100_{16}$ 字节而数据段大小是 $20_{16}$ 字节，这样过程B的代码段开始地址就是 $400\ 100_{16}$，数据段开始地址是 $1000\ 0020_{16}$。

| 可执行文件头 | | | 可执行文件头 | | |
|---|---|---|---|---|---|
| | 正文大小 | $300_{16}$ | | $0040\ 0104_{16}$ | $1a1\ 400\ 000_{16}$ |
| | 数据大小 | $50_{16}$ | | … | … |
| 代码段 | 地址 | 指令 | 数据段 | 地址 | |
| | $0040\ 0000_{16}$ | lw $a0,8000$_{16}($gp) | | $1000\ 0000_{16}$ | (X) |
| | $0040\ 0004_{16}$ | jal $40\ 0100_{16}$ | | … | … |
| | … | … | | $1000\ 0020_{16}$ | (Y) |
| | $0040\ 0100_{16}$ | sw $a1,8020$_{16}($gp) | | … | … |

图 2-13 也表明了代码段的起始地址是 $40\ 0000_{16}$，数据段的起始地址是 $1000\ 0000_{16}$。过程 A 的正文被放置在第一个地址而它的数据被放置在第二个地址。过程 A 的目标文件头表明其代码段大小是 $100_{16}$ 字节而数据段大小是 $20_{16}$ 字节，这样过程 B 的代码段开始地址就是 $400100_{16}$，数据段开始地址是 $10000020_{16}$。[⊖]

现在链接器更新了指令的地址字段。它使用指令类型字段得到待编辑地址的格式。这里共有两种类型：

1）jal 类型比较简单。因为它们使用伪直接寻址。对于地址 $40\ 0004_{16}$ 处的 jal，其地址字段是 $40\ 0100_{16}$（程序 B 的地址），而地址 $40\ 0104_{16}$ 处的 jal 的地址字段是 $40\ 0000_{16}$（程序 A 的地址）。

2）存取数指令对应的地址更为复杂，因为它们和基址寄存器有关。本例使用全局指针作为基址寄存器。图 2-13 表明 $gp 的初始值为 $10008000_{16}$。为了得到地址 $10000000_{16}$（字 X 的地址），我们设置位于地址 $400000_{16}$ 处的 lw 的地址字段中为 $8000_{16}$。同样，为了得到地址 $10000020_{16}$（字 Y 的地址），可以设置位于地址 $400100_{16}$ 处的 sw 的地址字段中为 $8020_{16}$。 □ 128

**01 精解** 回忆前面讲过 MIPS 指令是按字对齐的。所以 jal 指令丢弃最右侧 2 位来增加指令寻址范围。这样，它就可以使用 26 位来产生一个 28 位的字节地址。因此，本例中 jal 指令的低 26 位存放的实际地址是 $10\ 0040_{16}$，而不是 $40\ 0100_{16}$。

## 2.12.4 加载器

现在可执行文件已经在磁盘中，操作系统可以将其读入内存并启动执行它。在 UNIX 系统中，**加载器**（loader）按照如下步骤工作：

1）读取可执行文件头来确定代码段和数据段的大小。

2）为正文和数据创建一个足够大的地址空间。

3）将可执行文件中的指令和数据复制到内存中。

4）把主程序的参数（如果存在）复制到栈顶。

5）初始化机器寄存器，将栈指针指向第一个空位置。

6）跳转到启动例程，它将参数复制到参数寄存器并且调用程序的 main 函数。当 main 函数返回时，启动例程通过系统调用 exit 终止程序。

➲ 加载器：把目标程序装载到内存中以准备运行的系统程序。

附录 A 中的 A.3 节和 A.4 节更加详细地描述了链接器和加载器。

## 2.12.5 动态链接库

> 事实上，计算机科学中的每个问题可以在其他层次上间接地解决。
>
> ——David Wheeler

本节的第一部分将描述程序运行前链接库文件的传统方法。尽管这种静态的方法是最快的调用库程序的办法，但它有以下缺点：

- 库程序成为可执行代码的一部分。这样如果发布新版本的库以修正一些错误或支持新的硬件设备，静态链接的程序中使用的还是旧版本。
- 在程序运行时，尽管可能不会使用库中的所有部分，但它们还是会被全部加载进来。相对程序而言，库可能会很大，例如，标准的 C 库有 2.5MB。

---

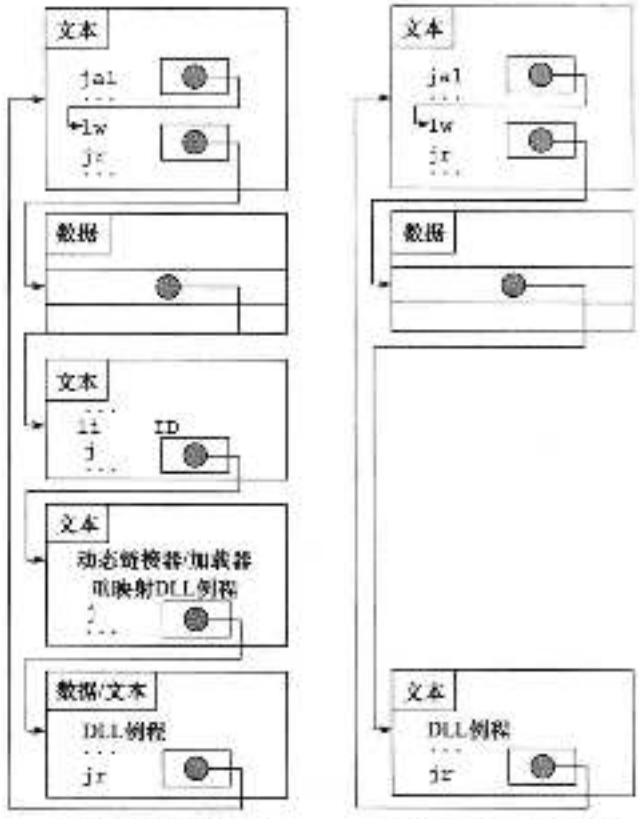
⊖ 原书中表格前后的文字除开头外一模一样，此处按原书翻译。——译者注

这些不足导致了**动态链接库**（dynamically linked library，DLL）的产生，也就是说，直到程序运行的时候，这些库例程才会被链接并加载。程序和库例程都会在非局部的过程和名字中保存额外的信息。在最初版本的 DLL 中，加载器调用一个动态链接器，使用文件中的额外信息来找到适当的库并且更新所有外部引用。

129 ➲ *动态链接库：在程序执行过程中才被链接的库例程。*

最初版本 DLL 的缺点是它仍链接库中所有在程序运行时可能调用的例程，而不是仅仅链接程序运行时实际调用的例程。由此产生 DLL 的晚过程链接（lazy procedure linkage）版本，该版本中每个例程只有在它被调用后才被链接。

就像这个领域中的许多创新一样，这个技巧采用了一种间接的方法。图 2-22 展示了该技术。它以一个非局部例程开始，该例程的末尾调用了一组虚例程，每个非局部例程都有一个入口。每个虚入口都包含一个间接跳转。




a）第一次调用DLL例程　　b）DLL例程的后续调用

图 2-22　通过晚过程链接方式链接动态链接库。a）第一次调用 DLL 的步骤；b）在随后的调用中，查找例程、重映射例程和链接例程被跳过。我们将在第 5 章看到，操作系统通过虚拟内存管理方式来重映射例程以避免复制所需例程

第一次调用库例程的时候，程序首先调用虚入口，然后执行间接跳转。它通过将一个数字放入寄存器来识别所需的库例程，然后跳转到动态链接器或加载器。链接器或加载器找到所需的例程，将其重映射并改变间接跳转位置的地址，使其指向这个例程。然后跳转到这个例程。这个例程完成时，将返回到初始调用点。此后，它都会间接跳转到这个例程而不去执行额外的中间过程。

总的来说，DLL需要额外的空间来存储动态链接的信息，但是不需要复制或链接整个库。仅仅在例程的第一次调用时开销较大，此后就只需一个间接跳转。注意，从库返回的操作不需要额外的开销。微软的 Windows 广泛地依赖动态链接库，如今在 UNIX 系统中程序执行的默认方式也是使用动态链接库。

### 2.12.6 启动一个 Java 程序

前面讨论了程序执行的传统模式，重点是以一个特定的指令集体系结构甚至这个体系结构的特定实现为目标的程序的快速执行。实际上，可以像 C 那样来执行 Java 程序。然而，Java 是为了不同的目标而发明的，其中之一就是能够安全地运行在每台计算机上，尽管这可能延长执行时间。

图 2-23 展示了典型的 Java 翻译和运行步骤。Java 程序会首先被编译成易于解释的指令序列——Java 字节码（Java bytecode）指令集（见 2.15 节），而不是编译成目标计算机可识别的汇编语言。这个指令集被设计得非常接近 Java 语言，这样，编译步骤相对简单，事实上它没有做任何优化。就像 C 语言编译器那样，Java 编译器会检查数据类型并且为每种类型提供正确的操作。Java 程序将转化成这些字节码的二进制形式。

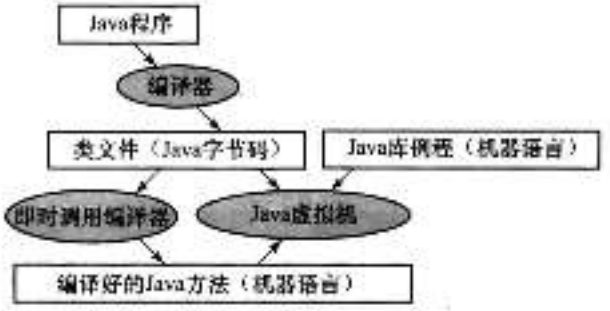



图 2-23 Java 的翻译层次。一个 Java 程序首先被编译成一个二进制版本的 Java 字节码形式，其中由编译器定义所有的地址。此时，Java 程序已可在解释器上运行，称为 Java 虚拟机（JVM）。在程序运行的时候，JVM 链接 Java 库中一些需要调用的方法。为了得到更好的性能，JVM 能够调用即时（just in time，JIT）编译器，在运行它的机器上能够选择性地把一些方法编译成宿主机上的本地机器语言

⊃ Java 字节码：为了解释 Java 程序而设计的指令集中的指令。

一个叫作 Java 虚拟机（Java Virtual Machine，JVM）的软件解释器能够执行 Java 字节码文件。解释器是一个用来模拟指令集体系结构的程序。例如，本书所使用的 MIPS 模拟器就是一个解释器。由于翻译非常简单，所以地址可以由编译器填写或在运行时被 JVM 发现，不需要再单独进行汇编。

130 ↓ 131

⊃ Java 虚拟机：解释 Java 字节码的程序。

解释的优势是可移植性。软件实现的 Java 虚拟机的可用性意味着在 Java 公布以后，大部分人都可以立即编写和运行 Java 程序。今天 Java 虚拟机可以用在从手机到网络浏览器等数亿的设备中。

解释的不足是性能较差。20 世纪 80 年代和 90 年代解释在执行性能上的飞速提高使它可用于很多重要的应用程序，但是与传统的编译好的 C 程序相比，10 倍的性能差距使 Java 对一些应用程序毫无吸引力。

为了既保持可移植性又提高执行速度，开发 Java 的下一阶段目标是实现程序执行的同时可以进行翻译的编译器。这个**即时编译器**（Just In Time complier）通过记录运行的程序来找到称为“热点”的方法，然后将它们直接编译成 Java 虚拟机运行的宿主机的指令序列，编译过的部分保存起来以便下次程序运行时调用，这样，以后每次运行会更快。解释和编译的平衡随着时间的推移逐步形成，届时，经常运行的 Java 程序的解释开销变得非常小。

- 即时编译器：一类通用编译器的名称，编译器能够在运行时将解释的代码段翻译成宿主计算机上的机器语言。

随着计算机的速度越来越快，编译器能做的事情也越来越多。而随着研究者不断地发明更好的技术来编译 Java 程序，Java 与 C 或 C++ 在性能上的差距越来越小。2.15 节将进一步介绍 Java 程序、Java 字节码、JVM 和 JIT 编译器。

**小测验** 对 Java 设计者来说，你认为与翻译器相比，解释器在那些方面的优点是最重要的？

1. 解释器便于编写。
2. 更准确的错误消息。
3. 更少的目标代码。
4. 机器独立性。

## 2.13 以一个 C 排序程序作为完整的例子

以片断的方式展示汇编代码的危险之处在于，你无法知道整个汇编语言程序的全貌。本节，我们给出了两个 C 过程对应的 MIPS 代码：一个用于交换（swap）数组的元素，另一个用
[132] 于对数组元素排序（sort）。

### 2.13.1 swap 过程

我们从图 2-24 中的过程 swap 开始。这个过程简单地交换内存中两个位置的内容。我们按照以下常见的步骤把它从 C 程序手动翻译为汇编程序：

1）为程序变量分配寄存器。

2）为过程体生成汇编代码。

3）保存过程调用间的寄存器。

本小节将按照这三个步骤描述 swap 程序，在最后把它们总结在一起。

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

图 2-24 一个交换内存中两个不同位置所存的数值的 C 过程。本小节要在排序的例子中使用这个过程

**1. 为 swap 分配寄存器**

如 2.8 节所述，在 MIPS 中，实现参数传递通常使用寄存器 $a0、$a1、$a2、$a3。由于 swap 只需要两个参数，v 和 k，它们将被分配在寄存器 $a0 和 $a1 中。由于 swap 是一个叶过程（见 2.8.2 节），所以我们为唯一的剩余变量 temp 分配寄存器 $t0。这些寄存器的分配与图 2-24 中的 swap 过程的第一部分变量的声明相对应。

**2. 为 swap 过程体生成代码**

swap 剩余部分的 C 代码如下所示：

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

回忆一下 MIPS 是按字节在内存中寻址的，字由 4 字节组成。因此我们需要把 k 乘以 4，
[133] 再与地址相加。忘记连续的字之间的地址相差 4 而不是 1，是汇编语言程序设计中常见的错误。因此获得 v [k] 地址的第一步就是通过左移 2 位来使 k 乘以 4：

```
sll   $t1, $a1,2     # reg $t1 = k * 4
add   $t1, $a0,$t1   # reg $t1 = v + (k * 4)
                     # reg $t1 has the address of v[k]
```

接下来使用 $t1 来取 v [k] 的值，在使 $t1 加 4 得到 v [k+1] 的地址：

```
lw    $t0, 0($t1)    # reg $t0 (temp) = v[k]
lw    $t2, 4($t1)    # reg $t2 = v[k + 1]
                     # refers to next element of v
```

最后将 $t0 和 $t2 存储到需要交换数据的地址中：

```
sw    $t2, 0($t1)    # v[k] = reg $t2
sw    $t0, 4($t1)    # v[k+1] = reg $t0 (temp)
```

至此，我们已经为该过程分配了寄存器并翻译好了程序体的代码。保存在 swap 中使用的保存寄存器的代码还没有完成。但是，由于这是一个叶过程，并没有使用保存寄存器，所以没有需要保留的东西。

### 3. 完整的 swap 程序

现在我们已经得到完整的例程了，包括程序标号和返回的跳转。为了方便读者的理解，在图 2-25 中，我们标明了过程中每个代码块的目的。

**过程体**

```
swap: sll   $t1, $a1, 2          # reg $t1 = k * 4
      add   $t1, $a0, $t1        # reg $t1 = v + (k * 4)
                                 # reg $t1 has the address of v[k]
      lw    $t0, 0($t1)          # reg $t0 (temp) = v[k]
      lw    $t2, 4($t1)          # reg $t2 = v[k + 1]
                                 # refers to next element of v
      sw    $t2, 0($t1)          # v[k] = reg $t2
      sw    $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

**过程返回**

```
      jr    $ra                  # return to calling routine
```

图 2-25 图 2-24 中 swap 过程的 MIPS 汇编代码 134

## 2.13.2 sort 过程

为保证你能够认识到汇编语言编程的严格性，我们提供了第二个更长的例子。在这个例子中，我们将编写一个调用 swap 过程的例程。这个例程对数组中的整数进行排序，使用的是冒泡或交换排序算法，这种排序算法虽然不是最快的，但却是最简单的。图 2-26 给出了该程序的 C 代码。我们还是使用几个步骤来演示翻译的过程，最后再把它们总结到一起。

```
void sort (int v[], int n)
{
   int i, j;
   for (i = 0; i < n; i += 1) {
       for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
           swap(v,j);
       }
   }
}
```

图 2-26 一个对数组 v 中元素进行排序的 C 程序

### 1. sort 的寄存器分配

为过程 sort 的两个参数 v 和 n 分配参数寄存器 $a0 和 $a1，为变量 i 和 j 分别分配寄存器 $s0 和 $s1。

2. 为 sort 过程体生成代码

过程体包含两个嵌套的 for 循环和一个有参数的 swap 调用。我们将从外到内来展开代码。第一步来翻译最外面的 for 循环。

```
for (i = 0; i < n; i += 1) {
```

回忆 C 语言中 for 的声明有三个参数：初始值、循环判断条件和迭代增量。for 语句的第一部分是将 i 初始化为 0，这需要一条指令，

```
move    $s0, $zero    # i = 0
```

（请记住 move 是为了方便汇编程序员而由汇编器提供的伪指令，见 2.12.2 节。）for 语句的最后部分，需要一条语句来增加 i：

135
```
addi    $s0, $s0, 1     # i += 1
```

循环要在条件 i<n 非真的时候退出，换句话说，当 i≥n 时循环退出。如果 $s0 < $a1，那么小于则置位指令将 $t0 置 1，否则置 0。因为我们要测试 $s0≥$a1，所以当寄存器 $t0 为 0 时，执行分支指令。这需要两条指令：

```
for1tst:slt  $t0, $s0, $a1     # reg $t0 = 0 if $s0 >= $a1 (i>=n)
        bne  $t0, $zero,exit1  # go to exit1 if $s0 >= $a1 (i>=n)
```

循环的底部仅仅需要跳回循环判断的地方：

```
        j    for1tst           # jump to test of outer loop
exit1:
```

第一个 for 循环的框架代码为

```
        move $s0, $zero        # i = 0
for1tst:slt $t0, $s0, $a1      # reg $t0 = 0 if $s0 >= $a1 (i>=n)
        beq  $t0, $zero,exit1  # go to exit1 if $s0 >= $a1 (i>=n)
             . . .
             (body of first for loop)
             . . .
        addi $s0, $s0, 1       # i += 1
        j    for1tst           # jump to test of outer loop
exit1:
```

（后面的练习将会进一步探索为类似的循环编写更快的代码。）

第二个 for 循环的 C 语句如下：

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

这个循环的初始化部分仍然是一条指令：

```
addi    $s1, $s0, -1 # j = i - 1
```

循环末尾 j 的自减（减 1）也是一条指令：

```
addi    $s1, $s1, -1 # j -= 1
```

循环判断由两个部分组成。任何一个条件为假就退出循环，所以第一个条件如果为假（j<0）就要退出循环：

```
for2tst: slti $t0, $s1, 0      # reg $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
```

136 这将跳过第二个条件测试，如果没有跳过，则 j≥0。

第二个测试条件当 v[j] > v[j + 1]非真的时候退出，或 v[j] ≤ v[j + 1]时退出。为得到地址，我们首先将 j 乘以 4（我们需要字节地址），然后将它与 v 的基地址相加：

```
sll     $t1, $s1, 2    # reg $t1 = j * 4
add     $t2, $a0, $t1  # reg $t2 = v + (j * 4)
```

现在取 v[j]：

```
lw      $t3, 0($t2)    # reg $t3 = v[j]
```

因为我们知道第二个元素恰好是下一个字，所以我们将寄存器 $t2 值加 4，得到 v[j + 1]

的地址：

```
lw    $t4, 4($t2)   # reg $t4 = v[j + 1]
```

测试 v[j]≤v≤[j+1]与测试 v[j+1]≥v[j]相同，所以测试退出的两条指令如下：

```
slt   $t0, $t4, $t3    # reg $t0 = 0 if $t4 ≥ $t3
beq   $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
```

循环末尾跳回到内层循环测试处：

```
j     for2tst   # jump to test of inner loop
```

将这些片段组合到一起，可得第二个 for 循环的框架如下：

```
        addi $s1, $s0, -1      # j = i - 1
for2tst:slt $t0, $s1, 0        # reg $t0 = 1 if $s1 < 0 (j < 0)
        bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
        sll $t1, $s1, 2        # reg $t1 = j * 4
        add $t2, $a0, $t1      # reg $t2 = v + (j * 4)
        lw  $t3, 0($t2)        # reg $t3 = v[j]
        lw  $t4, 4($t2)        # reg $t4 = v[j + 1]
        slt $t0, $t4, $t3      # reg $t0 = 0 if $t4 ≥ $t3
        beq $t0, $zero, exit2 # go to exit2 if $t4 > $t3
            . . .
            (body of second for loop)
            . . .
        addi $s1, $s1, -1      # j -= 1
        j  for2tst             # jump to test of inner loop
exit2:
```

### 3. sort 中的过程调用

下一步翻译第二个 for 循环的循环体：

```
swap(v,j);
```

调用 swap 很容易：

```
jal   swap
```

137

### 4. sort 中的参数传递

当我们想传递参数时问题出现了，因为 sort 过程需要使用寄存器 $a0 和 $a1 中的值，而 swap 过程需要将它的参数放入这些寄存器。一种解决办法是在过程执行的早期将 sort 的参数复制到其他的寄存器中，使 swap 过程可以使用寄存器 $a0 和寄存器 $a1。（这个复制的过程比在栈中保存后再取回要快得多。）在过程中我们首先将寄存器 $a0 和 $a1 的值复制到寄存器 $s2 和 $s3。

```
move  $s2, $a0     # copy parameter $a0 into $s2
move  $s3, $a1     # copy parameter $a1 into $s3
```

然后用下面两条指令将参数传递给 swap

```
move  $a0, $s2     # first swap parameter is v
move  $a1, $s1     # second swap parameter is j
```

### 5. 在 sort 中保留寄存器

仅剩保存和恢复寄存器值的代码了。因为 sort 是一个过程并且它要递归使用，所以很明显需要用寄存器 $ra 保存返回地址。sort 过程还使用了 $s0、$s1、$s2 和 $s3 保存寄存器，它们的值也必须被保存。所以 sort 过程头如下：

```
addi  $sp,$sp,-20  # make room on stack for 5 registers
sw    $ra,16($sp)  # save $ra on stack
sw    $s3,12($sp)  # save $s3 on stack
sw    $s2, 8($sp)  # save $s2 on stack
sw    $s1, 4($sp)  # save $s1 on stack
sw    $s0, 0($sp)  # save $s0 on stack
```

过程末尾只需反向执行这些指令，然后为了返回加上 jr 指令。

6. 完整的 sort 过程

现将所有片段合起来放入图 2-27。注意 for 循环中对寄存器 $a0 和 $a1 的引用已经被替换成对寄存器 $s2 和 $s3 的引用。为了方便阅读，我们再一次将过程中每一块的用途标了出来。本例中，9 行 C 语言编写的 sort 过程被翻译成 35 行的 MIPS 汇编语言代码。

| 保存寄存器值 | | | |
|---|---|---|---|
| | sort: | addi | $sp,$sp,-20 # make room on stack for 5 registers |
| | | sw | $ra,16($sp) # save $ra on stack |
| | | sw | $s3,12($sp) # save $s3 on stack |
| | | sw | $s2,8($sp) # save $s2 on stack |
| | | sw | $s1,4($sp) # save $s1 on stack |
| | | sw | $s0,0($sp) # save $s0 on stack |
| 过程体 | | | |
| 移动参数 | | move | $s2,$a0 # copy parameter $a0 into $s2 (save $a0) |
| | | move | $s3,$a1 # copy parameter $a1 into $s3 (save $a1) |
| 循环外部 | | move | $s0,$zero # i = 0 |
| | for1tst: | slt | $t0,$s0,$s3 # reg $t0 = 0 if $s0 >= $s3 (i >= n) |
| | | beq | $t0,$zero,exit1 # go to exit1 if $s0 >= $s3 (i >= n) |
| 循环内部 | | addi | $s1,$s0,-1 # j = i - 1 |
| | for2tst: | slti | $t0,$s1,0 # reg $t0 = 1 if $s1 < 0 (j < 0) |
| | | bne | $t0,$zero,exit2 # go to exit2 if $s1 < 0 (j < 0) |
| | | sll | $t1,$s1,2 # reg $t1 = j * 4 |
| | | add | $t2,$s2,$t1 # reg $t2 = v + (j * 4) |
| | | lw | $t3,0($t2) # reg $t3 = v[j] |
| | | lw | $t4,4($t2) # reg $t4 = v[j + 1] |
| | | slt | $t0,$t4,$t3 # reg $t0 = 0 if $t4 >= $t3 |
| | | beq | $t0,$zero,exit2 # go to exit2 if $t4 >= $t3 |
| 传递参数和调用 | | move | $a0,$s2 # 1st parameter of swap is v (old $a0) |
| | | move | $a1,$s1 # 2nd parameter of swap is j |
| | | jal | swap # swap code shown in Figure 2.25 |
| 循环内部 | | addi | $s1,$s1,-1 # j -= 1 |
| | | j | for2tst # jump to test of inner loop |
| 循环外部 | exit2: | addi | $s0,$s0,1 # i += 1 |
| | | j | for1tst # jump to test of outer loop |
| 恢复寄存器值 | | | |
| | exit1: | lw | $s0,0($sp) # restore $s0 from stack |
| | | lw | $s1,4($sp) # restore $s1 from stack |
| | | lw | $s2,8($sp) # restore $s2 from stack |
| | | lw | $s3,12($sp) # restore $s3 from stack |
| | | lw | $ra,16($sp) # restore $ra from stack |
| | | addi | $sp,$sp,20 # restore stack pointer |
| 过程返回 | | | |
| | | jr | $ra # return to calling routine |

图 2-27 图 2-26 中 sort 过程的 MIPS 汇编版本

**精解** 这个例子可以使用的一种优化方法是内联过程（procedure inlining）。在代码中调用swap过程的地方，编译器将swap的过程体的代码复制过来，而不是通过传递参数并通过jal指令来调用这段代码。本例中使用内联可以省掉4条指令。使用内联优化的缺点是如果内联过程需要在多个地方调用，编译后产生的代码将会变多。如果这种代码扩展导致cache的缺失率上升，将导致性能的下降（见第5章）。

138 | 139

**理解程序性能** 图2-28展示了编译器优化对排序程序的性能、编译时间、时钟周期、指令数和CPI的影响。注意没有优化的代码具有最好的CPI，使用O1优化的代码具有最少的指令数，但是O3优化的执行速度最快。这告诉我们执行时间是准确衡量程序性能的唯一指标。

图2-29比较了编程语言、编译执行或解释执行和算法对排序程序性能的影响。第四列表明在执行冒泡排序时没有优化的C程序比解释型的Java程序快8.3倍。使用即时编译器可以使Java比没有优化的C程序快2.1倍，比最佳优化的C代码慢不到1.13倍。（2.15节将给出关于解释执行和编译执行Java的更多细节以及冒泡排序的Java和MIPS代码。）在第五列中，快速排序的性能比就没那么接近了，这大概是因为在这样短的执行时间内分摊运行时编译的时间是非常困难的。最后一列展示了更好的算法带来的影响。当对100 000个元素进行排序时，性能达到了3个数量级的提升。即第五列中解释执行的Java与第四列中最优化的C代码相比，快速排序法要比冒泡法快50倍（0.05×2468或者用123/2.41）。

| gcc优化选项 | 相对性能 | 时钟周期（百万） | 指令数（百万） | CPI |
|---|---|---|---|---|
| 无 | 1.00 | 158 615 | 114 938 | 1.38 |
| O1（中等） | 2.37 | 66 990 | 37 470 | 1.79 |
| O2（完全） | 2.38 | 66 521 | 39 993 | 1.66 |
| O3（过程集成） | 2.41 | 65 747 | 44 993 | 1.46 |

图2-28 冒泡排序中编译器优化对性能、指令数、CPI的影响比较。程序对含有100 000个字的数组初始化为随机数的数组进行排序；程序运行在3.06GHz的奔腾4处理器上，前端系统总线是533MHz，具有2GB的PC2100 DDR SDRAM。操作系统使用Linux 2.4.20

140

| 编程语言 | 执行方式 | 优化选项 | 冒泡排序相对性能 | 快速排序相对性能 | 快速排序相对冒泡排序加速比 |
|---|---|---|---|---|---|
| C | 编译器 | 无 | 1.00 | 1.00 | 2468 |
| | 编译器 | O1 | 2.37 | 1.50 | 1562 |
| | 编译器 | O2 | 2.38 | 1.50 | 1555 |
| | 编译器 | O3 | 2.41 | 1.91 | 1955 |
| Java | 解释器 | — | 0.12 | 0.05 | 1050 |
| | 即时编译器 | — | 2.13 | 0.29 | 338 |

图2-29 两个排序算法的性能比较。算法分别用C和Java实现，Java分别使用解释执行和优化编译来与未优化的C版本比较。最后一列是快速排序比冒泡排序在每种语言和执行方式下速度提高多少。这些程序运行的系统与图2-28相同。JVM是Sun的1.3.1版本，JIT是Sun Hotspot的1.3.1版本

**精解** MIPS的编译器总是在栈上为参数保留空间以便它们得以保存，所以实际上$sp总是减16来给4个参数寄存器（16字节）分配空间。这样做的原因是C提供一个vararg选项，该选项允许选择一个指针，例如过程的第三个参数。当编译器遇到这种少见的vararg时，它就将4个参数寄存器的值都复制到栈上已经保留的位置中。

## 2.14 数组与指针

理解指针对任何一个C程序新手来说都是具有挑战的。通过对比使用数组和数组标记的汇编代码和使用指针的汇编代码，可以从本质上来理解指针。本节将展示C和MIPS汇编版本的两个清除内存中连续字的过程：一个使用数组标记；另一个使用指针。图2-30给出了这两个C过程。

```
clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}
clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

图2-30 两个将数组清零的C过程。clear1使用下标，而clear2使用指针。对不熟悉C的人，第二个过程需要做一些解释。变量的地址使用&表示，指针所指向的对象用*表示。声明部分说明array和p都是指向整数的指针。clear2的for循环中第一个部分将array的第一个元素的地址赋值给指针p。for循环的第二部分判断这个指针是否指向了array的最后一个元素之外。for循环的最后部分，对这个指针每次递增（增1），意味着将指针移到它声明的空间中的下一个对象。由于p是一个指向整数的指针，编译器将会产生MIPS指令，让p按照4递增，4是MIPS中整数的字节数目。循环体中将0赋值给p所指向的对象

本节的目的是展示指针是如何映射到MIPS指令的，而不是赞同这种过时的编程风格。我们在本节的末尾将看到现代编译器的优化对这两个过程带来的影响。

### 2.14.1 用数组实现clear

我们从数组版本的clear1开始，主要关注循环体，而忽略过程链接相关的代码。假设两个参数array和size分别在寄存器$a0和$a1中，i保存在$t0中。

for循环的第一部分，初始化变量i：

```
move    $t0,$zero       # i = 0 (register $t0 = 0)
```

为了将array[i]清0，我们首先需要得到它的地址。首先把i乘以4得到字节地址：

```
loop1: sll   $t1,$t0,2      # $t1 = i * 4
```

因为数组的起始地址在寄存器中，所以我们必须将它与下标相加以得到array[i]的地址，使用下面的加法指令：

141

```
add   $t2,$a0,$t1     # $t2 = address of array[i]
```

然后，我们就将0保存在这个地址：

```
sw    $zero, 0($t2)  # array[i] = 0
```

这条指令是循环体最后一条指令，下一步是增加i值（加1）：

```
addi $t0,$t0,1      # i = i + 1
```

循环测试条件检测i是否小于size：

```
slt  $t3,$t0,$a1      # $t3 = (i < size)
bne  $t3,$zero,loop1  # if (i < size) go to loop1
```

现在，我们已经得到过程所有的片断。下面则是使用数组下标对数组清零的MIPS汇编码：

```
        move  $t0,$zero       # i = 0
loop1:  sll   $t1,$t0,2       # $t1 = i * 4
        add   $t2,$a0,$t1     # $t2 = address of array[i]
        sw    $zero, 0($t2)   # array[i] = 0
        addi  $t0,$t0,1       # i = i + 1
        slt   $t3,$t0,$a1     # $t3 = (i < size)
        bne   $t3,$zero,loop1 # if (i < size) go to loop1
```

（只要 size 大于0，这些代码就能正确工作；ANSI C 需要在循环前测试 size 值，但是我们跳过了这个。）

142

## 2.14.2 用指针实现 clear

第二个过程是使用指针的，该过程将两个参数 array 和 size 分配到寄存器 $a0 和 $a1，将 p 分配到寄存器 $t0。在第二个过程开始时需要将数组的首地址赋值给指针 p：

```
move  $t0,$a0          # p = address of array[0]
```

接下来的代码将是 for 循环体，它仅仅是简单地将 0 存到地址 p：

```
loop2: sw  $zero,0($t0)    # Memory[p] = 0
```

这条指令实现了循环体，所以下一条指令将是迭代子自增，即改变 p 使其指向下一个字：

```
addi  $t0,$t0,4        # p = p + 4
```

在 C 中将指针加 1 意味着将指针指向序列中下一个对象。因为 p 是一个指向整数的指针，整数占用 4 字节，编译器将对 p 加 4。

接着就是循环测试。首先计算 array 最后一个元素的地址。先将 size 乘以 4 得到字节地址。

```
sll   $t1,$a1,2        # $t1 = size * 4
```

然后，将乘积与数组的首地址相加以获得数组后面第一个字的地址：

```
add   $t2,$a0,$t1      # $t2 = address of array[size]
```

循环测试仅仅是简单地判断 p 是否比 array 最后一个元素的地址小：

```
slt   $t3,$t0,$t2      # $t3 = (p<&array[size])
bne   $t3,$zero,loop2  # if (p<&array[size]) go to loop2
```

所有的代码片段都已经完成，现在我们可以看到指针版本的数组清零了：

```
        move $t0,$a0          # p = address of array[0]
loop2:  sw   $zero,0($t0)     # Memory[p] = 0
        addi $t0,$t0,4        # p = p + 4
        sll  $t1,$a1,2        # $t1 = size * 4
        add  $t2,$a0,$t1      # $t2 = address of array[size]
        slt  $t3,$t0,$t2      # $t3 = (p<&array[size])
        bne  $t3,$zero,loop2  # if (p<&array[size]) go to loop2
```

143

与第一个例子一样，这段代码也假定 size 大于 0。

注意，尽管数组的末地址一直保持不变，但是这个程序循环的每次迭代都要计算它。一种快速的执行方式是将数组末地址的计算放到循环体外面：

```
        move $t0,$a0          # p = address of array[0]
        sll  $t1,$a1,2        # $t1 = size * 4
        add  $t2,$a0,$t1      # $t2 = address of array[size]
loop2:  sw   $zero,0($t0)     # Memory[p] = 0
        addi $t0,$t0,4        # p = p + 4
        slt  $t3,$t0,$t2      # $t3 = (p<&array[size])
        bne  $t3,$zero,loop2  # if (p<&array[size]) go to loop2
```

## 2.14.3 比较两个版本的 clear

将两段代码放在一起进行比较可以说明数组下标和指针的不同（指针版本带来的变化被高

完显示）：

```
        move  $t0,$zero      # i = 0
loop1:  sll   $t1,$t0,2      # $t1 = i * 4
        add   $t2,$a0,$t1    # $t2 = &array[i]
        sw    $zero,0($t2)   # array[i] = 0
        addi  $t0,$t0,1      # i = i + 1
        slt   $t3,$t0,$a1    # $t3 = (i < size)
        bne   $t3,$zero,loop1# if (i) go to loop1
```

```
        move  $t0,$a0        # p = &array[0]
        sll   $t1,$a1,2      # $t1 = size * 4
        add   $t2,$a0,$t1    # $t2 = &array[size]
loop2:  sw    $zero,0($t0)   # Mem[p] = 0
        addi  $t0,$t0,4      # p = p + 4
        slt   $t2,$t0,$t2    # $t3=(p<&array[size])
        bne   $t2,$zero,loop2# if (i) go to loop2
```

左边的版本必须在循环中有“乘”和加操作，因为 i 值增加了，每个地址都将从新下标开始被重新计算。右边存储器指针版本的代码直接增加指针 p。指针版本通过把一些操作拿到循环外部，将每次迭代执行的指令从 6 条减少到 4 条。这种手动的优化与编译器的强度减少（用移位代替乘）和变量消除（消除循环中的数组地址计算）是一致的。2.15 节叙述了这两种优化和其他一些优化。

**01 精解** 正如前面提到的，C 编译器需要增加测试来保证 size 一定大于 0。一个方法是在循环的第一条指令之前加入一条跳转到 slt 的跳转指令。

144

**01 理解程序性能** 以往经常教育人们要在 C 中使用指针来获得数组所无法获得的更高的效率。然而，“使用指针，甚至会使你自己都无法理解代码的含义。”现代的优化编译器可以为数组版本产生同样好的代码。现在大部分程序员更喜欢让编译器去做更繁重的工作。

## 2.15 高级内容：编译 C 语言和解释 Java 语言

本节将简要概述 C 编译器如何工作和 Java 是如何执行的。因为编译器将对计算机的性能产生重要影响，所以理解编译器技术是理解性能的关键。要知道“编译器的构建”课程的学习一般需要 1 个或 2 个学期，所以我们这里将仅仅介绍一些基本内容。

本节的第二部分是为对**面向对象语言**（objected oriented language）（例如 Java）在 MIPS 体系结构上执行感兴趣的读者准备的。本节将展示被用于解释执行的 Java 字节码和前面章节中用 C 编写的程序段的 Java 版本的 MIPS 代码，包括冒泡排序。本节将包括 Java 虚拟机和即时编译器。

本节的剩余内容在配套网站上。

➲ 面向对象语言：一种针对对象而不是动作的编程语言，或者针对数据而不是逻辑的编程语言。

## 2.16 实例：ARMv7（32 位）指令集

在嵌入式设备领域中最流行的指令集体系结构是 ARM，2011 年有超过 90 亿部各种各样的设备使用 ARM 处理器，并且以每年 20 亿的数量增长。ARM 最初代表 Acorn RISC Machine，稍后被改为 Advanced RISC Machine。ARM 与 MIPS 处理器在同年发布并遵循相同的简洁的设计哲学。图 2-31 列出了 ARM 与 MIPS 的相似性。它们二者的主要区别是 MIPS 有更多的寄存器，而 ARM 有更多的寻址模式。

图 2-32 展示了 MIPS 与 ARM 在算术逻辑和数据传输指令方面具有相似的核心指令集。

| | ARM | MIPS |
|---|---|---|
| 发布时间 | 1985 | 1985 |
| 指令大小（位） | 32 | 32 |
| 寻址空间（大小，模式） | 32 位，平坦 | 32 位，平坦 |
| 数据对齐 | 对齐 | 对齐 |
| 数据寻址模式 | 9 | 3 |
| 整数寄存器（个数，模式，大小） | 15 通用寄存器×32 位 | 31 通用寄存器×32 位 |
| I/O | 存储器映射 | 存储器映射 |

图 2-31 ARM 和 MIPS 指令集的相同点

| 指令名 | | ARM | MIPS |
|---|---|---|---|
| 寄存器－寄存器 | 加法 | add | addu，addiu |
| | 加法（溢出捕获） | adds；swivs | add |
| | 减法 | sub | subu |
| | 减法（溢出捕获） | subs；swivs | sub |
| | 乘法 | mul | mult，multu |
| | 除法 | | div，divu |
| | 与 | and | and |
| | 或 | orr | or |
| | 异或 | eor | xor |
| | 取寄存器高位 | — | hi |
| | 逻辑左移 | lsl[1] | sllv，sll |
| | 逻辑右移 | lsr[1] | srlv，srl |
| | 算术右移 | asr[1] | srav，sra |
| | 比较 | cmp，cmn，tst，teq | slt/i，slt/iu |
| 数据传输 | 取有符号字节 | ldrsb | lb |
| | 取无符号字节 | ldrb | lbu |
| | 取有符号半字 | ldrsh | lh |
| | 取无符号半字 | ldrh | lhu |
| | 取字 | ldr | lw |
| | 存字节 | strb | sb |
| | 存半字 | strh | sh |
| | 存字 | str | sw |
| | 读、写特殊寄存器 | mrs，msr | move |
| | 原子交换 | swp，swpb | ll；sc |

图 2-32 ARM 的寄存器－寄存器指令和数据传输指令与 MIPS 核心指令是等价的。横线表示体系结构不支持该操作或不能用一些指令来实现该操作。如果有几条可供选择的指令都与 MIPS 核心指令等价，那么用逗号分隔这些指令。ARM 中每条数据操作指令都有移位的部分，所以移位指令用了上标 1，它们基本是 move 指令的变种，例如 lsr[1]。注意 ARM 中没有除法指令

## 2.16.1 寻址模式

图 2-33 展示了 ARM 支持的数据寻址模式。不同于 MIPS，ARM 不需要使用专门的寄存器来保存 0 这个数值。尽管 MIPS 仅有 3 种简单的数据寻址模式（见图 2-18），ARM 却有 9 种寻址模式之多，包括十分复杂的计算的寻址模式。例如，ARM 的一种寻址模式可以把一个寄存器中的数移动任意位，将移位后得到的数与另外一个寄存器中的值相加产生地址，然后将产生

145 ~ 146

的新地址存入一个寄存器中。

| 寻址模式 | ARM | MIPS |
|---|---|---|
| 寄存器操作数 | × | × |
| 立即数操作数 | × | × |
| 寄存器+偏移（转移或基址） | × | × |
| 寄存器+寄存器（下标） | × | — |
| 寄存器+寄存器倍乘（缩放） | × | — |
| 寄存器+偏移和更新寄存器 | × | — |
| 寄存器+寄存器和更新寄存器 | × | — |
| 自增，自减 | × | — |
| 相对PC的数据 | × | — |

图2-33 数据寻址模式的总结。ARM具有分离的寄存器间接寻址和寄存器+偏移寻址模式，而不是仅仅在后一种模式的偏移地址上填0。为了增加寻址范围，如果是对半字或字进行操作，ARM对偏移左移1位或2位

## 2.16.2 比较和条件分支

MIPS使用寄存器中的值来决定条件分支是否执行。而ARM使用传统的存储在程序状态字中的4位条件码来决定条件分支是否执行。这4个条件码是：负值（negative）、零（zero）、进位（carry）和溢出（overflow）。这些条件码可以被任何算术或逻辑指令设置，不同于早期的体系结构，这些设置功能是每条指令的可选功能。明确的选项会使流水化的实现变得更加容易。ARM使用条件分支来测试条件码以判断所有有符号和无符号的关系。

CMP指令用一个操作数减去另一个操作数，用它们的差设置条件码。CMN指令将一个操作数与另一个操作数相加，用它们的和来设置条件码。TST指令将两个操作数进行逻辑与，然后设置除溢出位外其他的条件码。TEQ指令是用异或结果来设置条件码的前三位。

ARM具有这样一个不寻常的特点，每条指令都有一个可选的执行条件，这个条件决定于条件码。每条指令开始的4位字段决定这条指令将执行空操作（nop）还是执行真实的指令操作，这种选择也取决于条件码。因此，条件分支也可以被认为是有条件的执行无条件分支指令。条件执行指令可以取代仅为了跳过一条指令的分支指令，不仅占用的代码空间更少，而且也会节省运行时间。

图2-34展示了ARM和MIPS的指令格式。它们之间的主要区别有两点：每条指令的4位条件执行字段不同；ARM因为只用MIPS一半数量的寄存器，所以具有相对较小的寄存器字段。

147

## 2.16.3 ARM的特色

图2-35列举了ARM处理器所特有的一些算术逻辑指令，这些指令在MIPS中是不存在的。由于没有专门的寄存器用来存储0，所以ARM需要单独的操作码来完成一些在MIPS中可以简单使用$zero来完成的操作。另外，ARM支持多个字的算术操作。

ARM解释12位立即数字段的方式非常新颖。首先将右侧低8位的有效位填0扩展到32位，然后将所得的数循环右移，移动的位数由高4位的值乘以2决定。这种解释方式的优点是可以在32位字的范围内表达所有2的幂次。为什么这种分割所表示的数字多于简单的12位字段是一个有趣的问题。

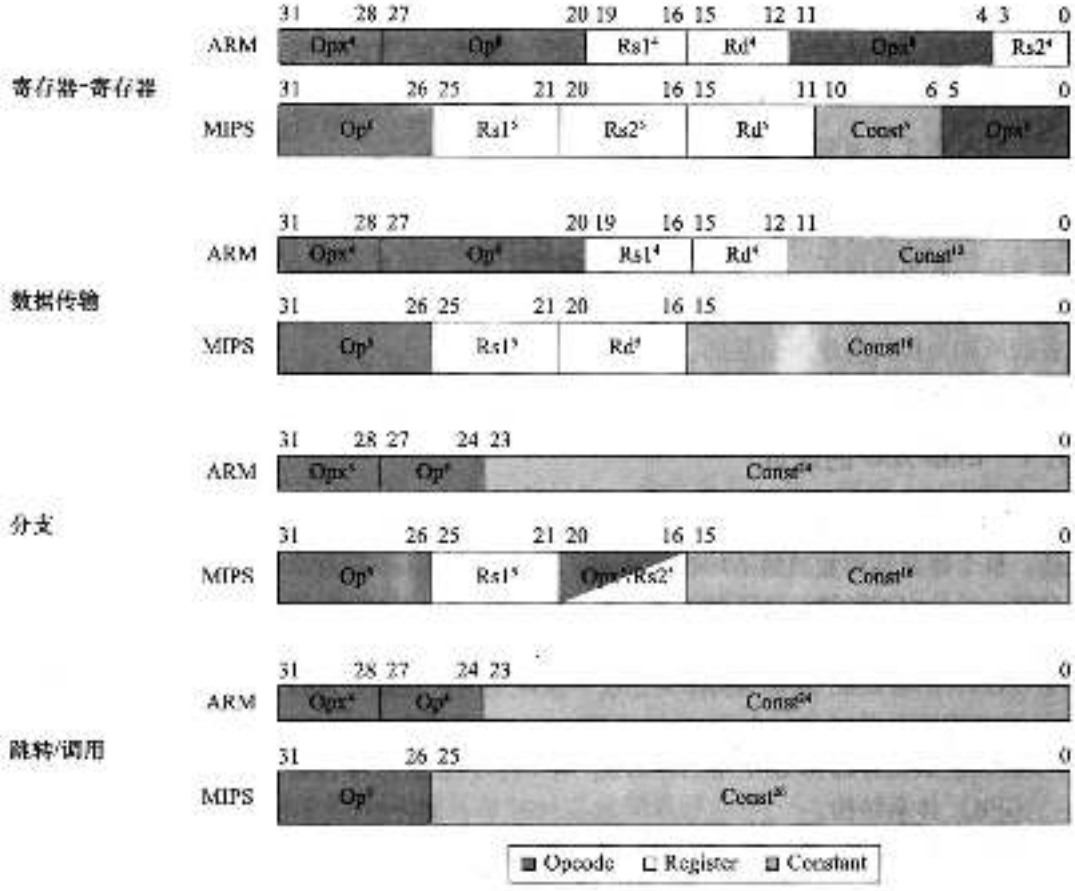



图 2-34 ARM 和 MIPS 的指令格式。区别在于体系结构中是有 16 个还是 32 个寄存器

| 名字 | 定义 | ARM | MIPS |
|---|---|---|---|
| 取立即数 | Rd = Imm | mov | addi $0, |
| 非 | Rd = ~(Rs1) | mvn | nor $0, |
| 移动 | Rd = Rs1 | mov | or $0, |
| 循环右移 | Rd = Rsi >> i<br>$Rd_{0\cdots i-1} = Rs_{21-i\cdots 21}$ | ror | |
| 寄存器与另一寄存器的非进行与操作 | Rd = Rs1 & ~(Rs2) | bic | |
| 反向减 | Rd = Rs2 - Rs1 | rsb, rsc | |
| 支持多个参数字的加 | CarryOut, Rd = Rd + Rs1 + OldCarryOut | adcs | — |
| 支持多个参数字的减 | CarryOut, Rd = Rd - Rs1 + OldCarryOut | sbcs | — |

图 2-35 MIPS 中没有的 ARM 算术/逻辑指令

对操作数的移位并不仅限于立即数。所有算术和逻辑运算操作的第二个寄存器操作数都可以在执行操作之前进行移位。可选的移位方式是逻辑左移、逻辑右移、算术右移和循环右移。

148

ARM 还对寄存器组的操作提供了指令支持，这些指令叫作*块加载和存储*（block loads and stores）。在指令的 16 位掩码的控制下，16 个寄存器中的任意组合都可以被一条指令加载或存储到内存中。这些指令可以保存和恢复程序调用和返回时的寄存器。这些指令也可以被用于存

储器块的复制，现在这些存储器块的复制是对这些指令的主要应用。

## 2.17 实例：x86 指令集

情人眼里出西施。

——*Margaret Wolfe Hungerford,《Molly Bawn》, 1877*

指令集的设计者有时提供比 ARM 和 MIPS 更强大的操作。这样做的目的是减少程序需要执行的指令数。其风险在于，在设备简单性方面需付出一定的代价，并且可能使程序执行时间变长，这是因为指令执行需要更长的时间，这可能是由于时钟周期变长或者是比更简单的序列需要更多的周期来执行程序所引起的。

通向复杂操作的道路困难重重。2.19 节将阐述复杂性的陷阱。

### 2.17.1 Intel x86 的改进

ARM 和 MIPS 都是由单独的小组在 1985 年推出的。这种体系结构的每部分配合在一起非常合适，整个体系结构能被简洁地描述出来。但是 x86 却不是这样，它是由一些相互独立的小组开发的，并且它被持续改进了超过 35 年，不断在原来指令集的基础上增加新的特性，这就像有些人往包装好的包里添加衣服。下面是 x86 发展的一些重要的里程碑。

149

- 1978：Intel 8086 体系结构作为之前一款成功的 8 位微处理器的汇编语言的可兼容的扩展被发布。8086 是一个 16 位的体系结构，所有内部的寄存器都是 16 位长。与 MIPS 不同，它的寄存器都是专用的，因此 8086 并不是**通用寄存器**（general-purpose register，GPR）体系结构。
- 1980：Intel 8087 浮点协处理器发布。这个体系结构在 8086 的基础上增加了 60 条浮点指令。它通过栈来代替寄存器（见 2.21 节和 3.7 节）。
- 1982：80286 在 8086 的基础上把地址空间扩展到 24 位，并设计了精妙的内存映射和保护模式（见第 5 章），还增加了一些指令去丰富整个指令集以及控制保护模式。
- 1985：80386 在 80286 体系结构的基础上将地址空间扩展到 32 位。除了 32 位的寄存器和 32 位的地址空间，80386 也增加了一些新的寻址模式和额外的操作。增加的指令使得 80386 几乎就是通用寄存器的处理器。80386 还增加了对页的支持并提供了段寻址（参见第 5 章）。与 80286 一样，80386 也提供能运行不经修改的 8086 程序的模式。
- 1989～1995：接下来在 1989 年发布了 80486，1992 年发布 Pentium 处理器，1995 年发布 Pentium Pro 处理器。这些处理器都是以获得更高的性能为目的的，仅有 4 个指令被增加到用户可见的指令集中，其中 3 个有助于多处理技术（参见第 6 章），另一个是条件传送指令。
- 1997：在 Pentium 和 Pentium Pro 销售后，Intel 公司宣称他们将用多媒体扩展 MMX（Multi Media Extension）来扩展 Pentium 和 Pentium Pro 的体系结构。这个新指令集包含 57 条指令，使用浮点栈来加速多媒体和通信应用程序。MMX 通过传统的单指令多数据（single instruction，multiple data，SIMD）的方式来一次处理多个短的数据元素（参见第 6 章）。Pentium Ⅱ没有引入任何新的指令。
- 1999：Intel 添加了另外 70 条指令，将 SSE（Streaming SIMD Extension）作为 Pentium Ⅲ的一部分。主要的变化是添加了 8 个独立的寄存器，把它们的长度增加到 128 位，并且增加了一个单精度浮点数据类型。因此，4 个 32 位的浮点操作就可以并行进行。为了改进内存性能，SSE 还包括 cache 的预取指令，以及可以绕过缓冲器直接写内存的流存储指令。

- 2001：Intel 公司增加了另外 144 条指令。这次命名为 SSE2。增加的新的数据类型是双精度算术，它允许并行操作 64 位浮点型数据对。这 144 条指令几乎都对应着一些已经存在的 MMX 和 SSE 指令，这些指令并行操作 64 位数据。这种变化不仅允许更多的多媒体操作，并且与单独的栈架构相比，编译器多了一个新的浮点操作目标。编译器可以使用 8 个 SSE 寄存器来充当浮点寄存器。这种改进大大增强了第一个包括 SSE2 指令集的微处理器 Pentium 4 的浮点性能。 150
- 2003：这次是 AMD 改进了 x86 体系结构，把地址空间从 32 位增加到 64 位。与 1985 年在 80386 上从 16 位到 32 位的转变类似，AMD64 把所有的寄存器都拓宽到 64 位，并且把寄存器的数目增加到 16，把 128 位的 SSE 寄存器数目增加到 16 个。ISA 的主要变化是新增加了一个模式叫长模式（long mode），用 64 位的地址和数据来重新定义所有 x86 指令的执行。为了寻址更多的寄存器，给指令增加了新前缀。根据计算方式，长模式还添加了 4 – 10 条新的指令并且去掉了 27 条旧指令。PC 相对数据寻址是另一个扩展。AMD64 仍然有一个和 x86 相同的模式（遗产模式）并且增加了一个模式，以限制用户程序使用 x86 模式，但是却允许操作系统使用 AMD64 模式（兼容模式）。这些模式使它成为比 HP/Intel IA-64 更好地从 32 位过渡到 64 位寻址的处理器。
- 2004：Intel 屈服并吸纳了 AMD64，重新标记为 Extended Memory64 Technology（EM64T），主要的区别是 Intel 增加了 128 位的原子比较和交换指令，这个本应在 AMD64 上可能具有的指令。同时，Intel 发布了新一代媒体扩展。SSE3 添加了 13 条指令来支持复杂算术，包括在结构数组上进行的图形操作、视频编码、浮点转换以及线程同步（见 2.11 节）。AMD 会在以后的芯片中提供对 SSE3 的支持。而且它几乎肯定能够把原先没有的原子交换指令添加到 AMD64 使其与 Intel 二进制兼容。
- 2006：作为 SSE4 的一部分扩展，Intel 发布了 54 条新指令。这些扩展都是针对像如下影响性能的因素：绝对差求和、数组结构的点积计算、窄数据到较宽的数据的符号或零扩展、序列中非零的数目统计等。还增加了对虚拟机的支持（见第 5 章）。
- 2007：作为 SSE5 的一部分，AMD 发布了 170 条指令，包括为 46 条基本指令集中的指令增加了像 MIPS 的 3 操作数的版本。
- 2011：Intel 发布了高级向量扩展，同时将 SSE 寄存器从 128 位扩展到 256 位，因此重新定义了 250 条指令并新增了 128 条指令。 151

⊃ 通用寄存器：可用于存储任何指令的地址或数据的寄存器。

这段历史说明了兼容性这个“金手铐”对 x86 的影响。体系结构的改变不允许对已有的软件产生任何的危害。

尤论 x86 结构有多失败，该指令集一直对个人计算机的更新换代起着很大的推动作用，在后 PC 时代占据着很大的份额。表面上看起来，3.5 亿 x86 芯片的年产量相对于 ARMv7 芯片的 90 亿片要小很多，但是许多公司都想去控制这个市场。无论如何，这个多变的家族带来的是一个难以解释并且不讨人喜欢的体系结构。

请鼓起勇气来面对你将要看到的内容！不要带着需要编写 x86 程序的担心来阅读这一节，实际上，本节的目的是让你熟悉这一世界上最流行的台式机体系结构的优缺点。

本节我们主要关心的是 80386 的 32 位指令子集，而不是整个 16 位、32 位和 64 位指令集。我们从寄存器和寻址模式开始说明，接下来是整数操作，最后考虑指令编码。

## 2.17.2 x86 寄存器和数据寻址模式

80386 的寄存器展示了指令集的进化（如图 2-36 所示）。80386 把 16 位寄存器（除了段寄

152 存器）扩展为32位。并用前缀E来标示32位版本。它们通常被称为通用寄存器。80386只有8个通用寄存器，这意味着MIPS程序使用4倍数量的寄存器，而ARMv7可以使用2倍数量的寄存器。

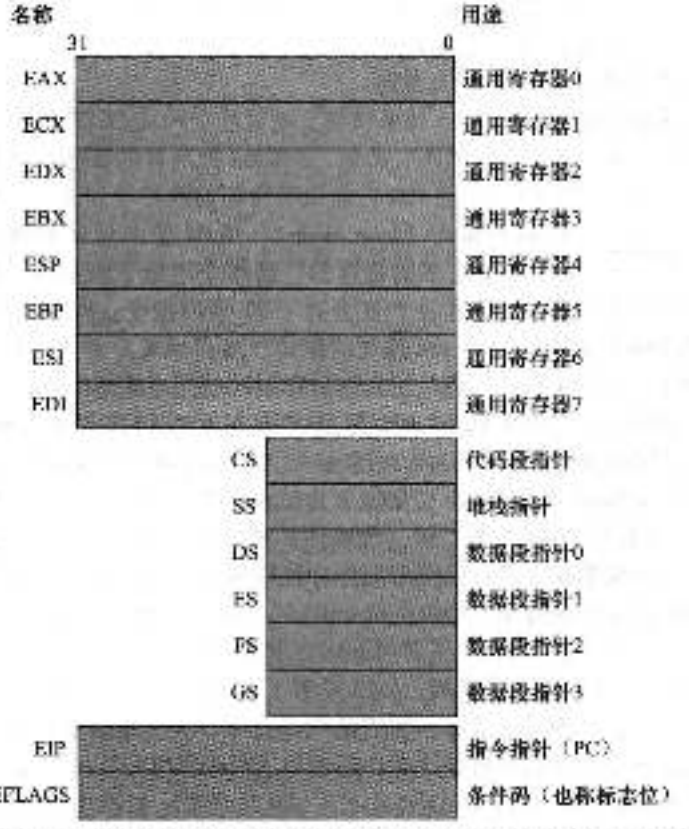



图2-36 80386寄存器组。从80386开始，上面的8个寄存器扩展到32位并可以当做通用寄存器使用

图2-37展示了两个操作数的算术、逻辑和数据传输指令。它们有两个重要的不同之处。首先x86的算术和逻辑指令中的一个操作数必须既是源操作数又是目的操作数，而ARMv7和MIPS的源操作数和目的操作数是不同的寄存器。这种限制给有限的寄存器带来更大的压力，因此一个源寄存器必须被改变。第二个重要的不同之处在于一个操作数可以在存储器中。这样，实质上任何指令都可能有一个操作数在存储器中。这与ARMv7和MIPS不同。

| 第1/目标操作数类型 | 第二个源操作数 |
|---|---|
| 寄存器 | 寄存器 |
| 寄存器 | 立即数 |
| 寄存器 | 存储器 |
| 存储器 | 寄存器 |
| 存储器 | 立即数 |

图2-37 算术、逻辑和数据传输指令的指令格式。x86所允许的组合见上表。唯一的限制是没有存储器-存储器模式。立即数可以是8位、16位或32位；寄存器可以是图2-36中14个主要的寄存器（不能是EIP或EFLAGS）的任意一个

后面将会详细阐述数据的存储器寻址模式，在指令中提供两种位长的地址。这种所谓的偏移（displacements）既可能是8位也可能是32位。

尽管存储器操作数可以使用任何寻址模式，但是每种模式使用哪些寄存器是有限制的。

图 2-38展示了 x86 寻址模式和每种模式下哪个 GPR 是不允许使用的，并说明如何使用 MIPS 指令集来达到相同效果。

| 模式 | 描述 | 寄存器限制 | 等价的 MIPS |
|---|---|---|---|
| 寄存器间接寻址 | 地址在寄存器中 | 不能为 ESP 或 EBP | `lw $s0,0($s1)` |
| 8 位或 32 位偏移寻址模式 | 地址是基址寄存器与偏移量之和 | 不能为 ESP | `lw $s0,100($s1)#<=16 bit #displacement` |
| 基址加比例下标寻址 | 地址是：<br>基址 +（$2^{比例}$ × 下标）<br>比例是 0、1、2 或 3 | 基址：任何 GPR<br>下标：不能为 ESP | `mul $t0,$s2,`<br>`add $t0,$t0,$s1`<br>`lw $s0,0($t0)` |
| 8 位或 32 位偏移量的基址 + 比例下标寻址 | 地址是<br>基址 +（$2^{比例}$ × 下标）+ 偏移量<br>比例是 0、1、2 或 3 | 基址：任何 GPR<br>下标：不能为 ESP | `mul $t0,$s2,4`<br>`add $t0,$t0,$s1`<br>`lw $s0,100($t0)#<=6 bit #displacement` |

图 2-38 x86 有寄存器使用限制的 32 位寻址模式及等价的 MIPS 代码。ARM 和 MIPS 所没有的，基址加比例下标寻址模式，包含在 x86 中以避免将寄存器中的下标乘 4（使用比例因子 2）变成字节地址（见图 2-25 和图 2-27）。比例因子 1 用于 16 位数据，3 用于 64 位数据。比例因子 0 意味着这个地址不需要按比例增加。在第二种或第四种模式中如果偏移量比 16 位长，等价的 MIPS 需要额外的两条指令：`lui` 取偏移量的高 16 位，`add` 将高 16 位与寄存器 `$s1` 相加。（Intel 的基址寻址模式还有另外的名字基址和下标，但是它们本质上是等同的，我们在这里将它们合并。）

## 2.17.3 x86 整数操作

8086 提供对 8 位（字节）和 16 位（字）数据类型的支持。80386 在 x86 结构中加入了 32 位的地址和数据（双字）。（AMD64 又增添了 64 位的地址和数据，叫作四字；本小节我们将关注 80386。）数据类型的不同也造成了寄存器操作和存储器访问的不同。

几乎所有操作都能在 8 位和一个更长的数据上进行。这个最长的数据大小取决于运行的模式，可能是 16 位也可能是 32 位。

显然，有些程序希望操作所有三种长度的数据，于是 80386 系统结构提供一种不用明显增加代码长度的方便途径来指定每一种形式。它们认为大多数程序中 16 位或 32 位数据占绝大多数，于是设定一个默认的较长长度是有意义的。这个默认的数据长度由代码段寄存器中的一位指定。若要改变默认数据长度，需在指令前附加 8 位前缀告诉机器这条指令使用其他数据长度。

使用前缀是从 8086 借鉴过来的，8086 可使用多种前缀来改变指令的行为。最初的三个前缀包括忽略默认的段寄存器，给总线加锁来支持同步（见 2.11 节），或重复后面的指令直到寄存器 ECX 减少到 0。最后一个前缀要配合一个字节传送指令使用以便传送可变数目的字节。80386 还加入一个前缀以改变默认的地址长度。

x86 整数操作可以分为 4 个主要的类：

1）数据传送指令，包括 move、push 和 pop。

2）算术和逻辑指令，包括测试、整数和小数算术运算。

3）控制流，包括条件分支、无条件跳转、调用和返回。

4）字符串指令，包括字符串传送和字符串比较。

153
154

除了算术和逻辑操作指令的结果既可以保存在寄存器也可以保存在存储器地址外，前两个种类没有值得关注之处。图 2-39 展示了典型的 x86 指令和它们的功能。

x86 的条件分支像 ARMv7 一样基于条件码（condition code）或标志位（flag）。条件码是作为一些操作的副作用被设置的，大部分被用作将结果与 0 比较，然后使用分支指令测试条件码。PC 相对分支地址必须以字节数来指定，这与 ARMv7 和 MIPS 不同，80386 的指令不都是 4 字节长的。

| 指令 | 功能 |
|---|---|
| jz name | if equal(condition code)(EIP = name): EIP - 128 <= name < EIP + 128 |
| jmp name | EIP = name |
| call name | SP = SP - 4; M[SP] = EIP + 5; EIP = name; |
| movw EBX,[EDI + 45] | EBX = M[EDI + 45] |
| push ESI | SP = SP - 4; M[SP] = ESI |
| pop EDI | EDI = M[SP]; SP = SP + 4 |
| add EAX,#6765 | EAX = EAX + 6765 |
| test EDX,#42 | Set condition code (flags) with EDX and 42 |
| movs l | M[EDI] = M[ESI]; EDI = EDI + 4; ESI = ESI + 4 |

图 2-39 x86 的一些典型指令和它们的功能。常用操作的列表在图 2-40 中。CALL 将下一条指令的 EIP 保存在栈上。(EIP 是 Intel 的程序计数器。)

字符串指令是 x86 的祖先 8080 的一部分，在大部分程序中都不使用。它们常常比同等功能的软件例程要慢（见 2.19 节的谬误）。

图 2-40 列出了一些 x86 的整数指令。这些指令大部分都同时有字节和字格式。

| 指令 | 含义 |
|---|---|
| **控制指令** | **条件和无条件分支** |
| jnz,jz | 条件成立跳转到 EIP + 8 位偏移量；JNE (for JNZ)，JE (for JZ) 两者之一 |
| jmp | 无条件跳转——8 位或 16 位偏移量 |
| call | 过程调用——16 位偏移量；返回地址压入栈中 |
| ret | 从栈中弹出返回地址并跳转到该地址处 |
| loop | 循环分支——递减 ECX；如果 ECX 非零，则跳转到 EIP + 8 位偏移处 |
| **数据传输** | **在寄存器之间或寄存器和存储器之间传递数据** |
| move | 在两个寄存器之间或寄存器和存储器之间传递数据 |
| push,pop | 将源操作数压栈；将栈顶数据取到寄存器中 |
| les | 从存储器中取 ES 和一个 GPR |
| **算术、逻辑** | **使用数据寄存器和存储器的算术和逻辑操作** |
| add,sub | 将源操作数与目的操作数相加；从目的操作数中减去源操作数；寄存器 - 存储器格式 |
| cmp | 比较源和目的操作数；寄存器 - 存储器格式 |
| shl,shr,rcr | 左移；逻辑右移；循环右移并用条件码填充 |
| cbw | 将 EAX 最右 8 位字节转换成 EAX 最右 16 位字 |
| test | 将源操作数和目的操作数进行逻辑与，并设置条件码 |
| inc,dec | 递增目的操作数，递减目的操作数 |
| or,xor | 逻辑或；异或；寄存器 - 存储器格式 |
| **字符串** | **在字符串操作数之间移动；由重复前缀给出长度** |
| movs | 通过递增 ESI 和 EDI 从源字符串复制到目的字符串；可能使用重复 |
| lods | 从字符串中取字节，字或双字到寄存器 EAX |

图 2-40 一些典型的 x86 操作。很多操作使用寄存器 - 存储器格式，这种格式要求源操作数或目的操作数可以是存储器，另一个操作数可以是寄存器或立即数

### 2.17.4 x86 指令编码

把最糟的放在最后——80386 的指令编码是非常复杂的，有多种不同指令格式。当没有操作数的时候，80386 的指令可以是 1 字节，最长到 15 字节。

图 2-41 展示了图 2-39 中几条指令的格式。操作码字节中通常有一位用来表明操作数是 8 位还是 32 位。一些指令的操作码可能还包含寻址模式和寄存器，例如，很多的指令具有如下形式“寄存器 = 寄存器操作立即数”。其他指令使用寻址模式的“后置字节”或额外的操作码字节，标记为“mod，reg，r/m”（模式、寄存器、寄存器/存储器）。这个后置字节在寻址存储器的很多指令中都被用到。基址加比例下标的寻址模式需要使用第二个后置字节，标记为“sc，indcx，basc”（比例，下标，基址）。

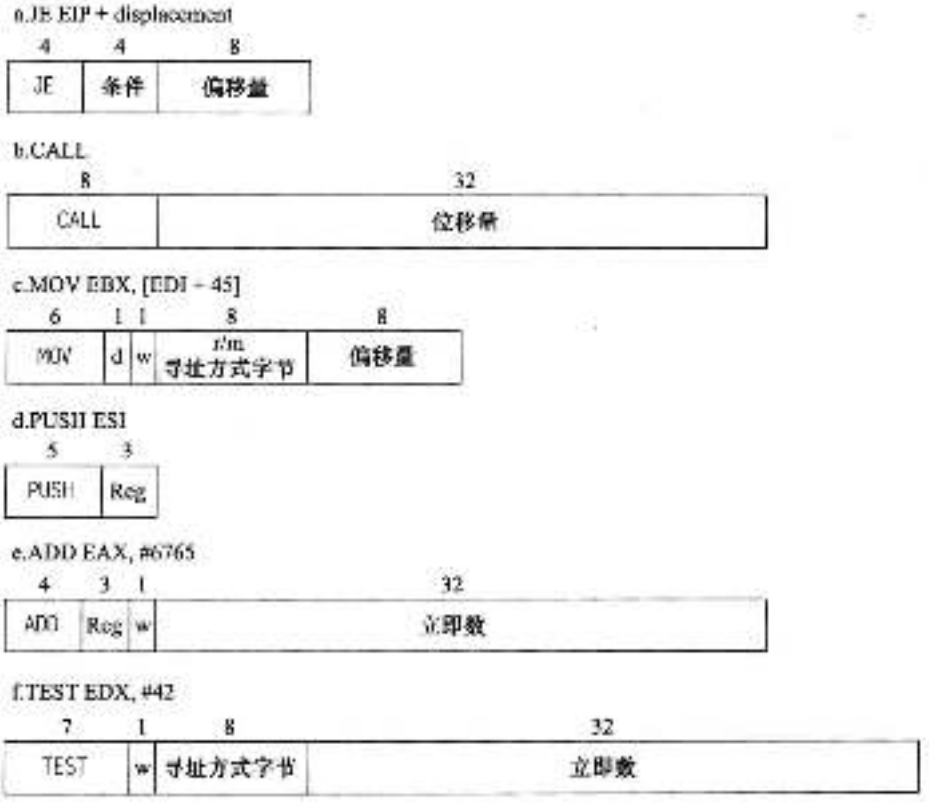



图 2-41 典型的 x86 指令格式。图 2-42 给出后置字节（posthyte）的编码。很多指令包含 1 位的 w 段，这个字段说明操作的是一个字节还是一个双字。MOV 中 d 字段用于从存储器中传出或传入数据的指令并指明传输方向。ADD 指令需要 32 位的立即数字段，因为在 32 位模式下，立即数或者是 8 位或者是 32 位。TEST 中的立即数字段也是 32 位长，是因为在 32 位模式下没有 8 位的立即数要判断。总的来说，指令长度可以从 1 字节到 15 字节变化。较长的长度产生于额外的 1 字节前缀，该长度具有 4 字节的立即数和 4 字节的偏移地址，使用 2 字节的操作码，并使用比例下标模式说明符，这还需要一个额外的字节

图 2-42 展示了 16 位和 32 位模式的两个后置字节地址指定的编码。不幸的是，为了全面理解哪个寄存器和哪种寻址模式可用，你需要看所有寻址模式的编码，有时甚至需要看指令编码。

| reg | w = 0 | w = 1 | | r/m | mod = 0 | | mod = 1 | | mod = 2 | | mod = 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 16b | 32b | | 16b | 32b | 16b | 32b | 16b | 32b | |
| 0 | AL | AX | EAX | 0 | addr = BX + SI | = EAX | same | same | same | same | same |
| 1 | CL | CX | ECX | 1 | addr = BX + DI | = ECX | addr as | addr as | addr as | addr as | as |
| 2 | DL | DX | EDX | 2 | addr = BP + SI | = EDX | mod = 0 | mod = 0 | mod = 0 | mod = 0 | reg |
| 3 | BL | DX | EBX | 3 | addr = BP + SI | = EDX | + disp8 | + disp8 | + disp16 | + disp32 | field |
| 4 | AH | SP | ESP | 4 | addr = SI | = (sib) | SI + disp8 | (sib) + disp8 | SI + disp8 | (sib) + disp32 | * |
| 5 | CH | BP | EBP | 5 | addr = DI | = disp32 | DI + disp8 | EBP + disp8 | DI + disp16 | EBP + disp32 | * |
| 6 | DH | SI | ESI | 6 | addr = disp16 | = ESI | BP + disp8 | ESI + disp8 | BP + disp16 | ESI + disp32 | * |
| 7 | BH | DI | EDI | 7 | addr = BX | = EDI | BX + disp8 | EDI + disp8 | BX + disp16 | EDI + disp32 | * |

图 2-42 x86 的第一个地址说明符的编码：mod，reg，r/m。前 4 列表示 3 位的 reg 字段，它依赖于操作码中的 w 位以及机器是工作在 16 位（8086）模式还是 32 位（80386）模式。余下的列解释了 mod 和 r/m 字段。3 位的 r/m 字段依赖于 2 位的 mod 字段和地址的大小。用于地址计算的寄存器列在第六和第七列中，mod = 0 时依赖于寻址模式，mod = 1 时加上 8 位的偏移量，mod = 2 时加上 16 位或 32 位的偏移量。例外的情况有以下几种：1）当 mod = 1 或 mod = 2，在 16 位模式时，r/m = 6 选择 BP 加上偏移。2）当 mod = 1 或 mod = 2，在 32 位模式时，r/m = 5 选择 EBP 加上偏移量。3）当 mod 不等于 3，在 32 位模式时，r/m = 4，(sib) 代表使用图 2-38 中的比例下标模式。当 mod = 3 时，r/m 字段指定一个寄存器，与 w 位组合在一起和 reg 字段的编码相同

### 2.17.5 x86 总结

Intel 的 16 位微处理器比它的竞争对手的更优秀的体系结构（如 Motorola 68000）早两年问世，这个领先使得 IBM 选用 8086 作为其 PC 的 CPU。Intel 的工程师普遍认识到 x86 要比 ARMv7 和 MIPS 的计算机更难制造，但是巨大的市场意味着 AMD 和 Intel 可以投入更多的资源来克服这些额外的复杂性。数量上的巨大优势弥补了风格上的缺点，这使得 x86 前景美好。

155
|
157

x86 中最常使用的体系结构组成部分是不难实现的，从 1978 年开始 AMD 和 Intel 就展示了整数程序性能的快速改进。为了获得这样的性能，编译器必须避免那些难于实现快速执行的体系结构部分。

然而，在后 PC 时代，虽然有大量的体系结构和制造专家基于 x86 做工作，但是 x86 在个人移动设备里面还不具有竞争力。

## 2.18 实例：ARMv8（64 位）指令集

在一个指令集所具有的所有潜在问题中，最不可能解决的就是地址空间太小的问题。x86 是第一个扩展为 32 位地址的，并且是第一个扩展为 64 位地址的指令集，许多其他的指令系统都被落在了后面。例如，虽然具有 16 位地址的 MOStek 6502 指令集统治了 Apple Ⅱ，但是 Apple Ⅱ即使是第一个成功的商用个人计算机的领头羊，却也由于其地址空间上的缺陷饱受诟病。

虽然 ARM 体系结构遇到了 32 位地址空间的限制，在 2007 年开始设计具有 64 位地址的 ARM，并最终在 2013 年完成。与 x86 中为了使寄存器加宽为 64 位只做了很小改变不同，ARM 作了完全的改进。如果你了解 MIPS，那就非常容易了解 64 位版本的 ARMv8。

首先，与 MIPS 相比，ARM 舍弃了 v7 中并不常用的一些特性：

158

- v8 中没有条件执行字段，而在 v7 中几乎每条指令都有该字段。
- 立即数字段仅仅是一个 12 位的常数，而在 v7 中是产生一个常数的函数的输入。
- ARM 舍弃了 Load Multiple 和 Store Multiple 指令。
- PC 不再是一个寄存器，因此如果对其进行写操作将会导致非预期的分支转移。

其次，ARM 添加了一些 MIPS 中有用的特征：

- v8 有 32 个通用寄存器，编译器设计者非常喜欢该特点。与 MIPS 相同，一个寄存器永远存放 0，虽然在 load 和 store 指令中该寄存器将由栈指针替代。
- ARMv8 的寻址方式是用于所有的字长，而在 ARMv7 中并非如此。
- 它包含了 ARMv7 中省掉的除法指令。
- 它增加了 MIPS 中的相等或不等的条件分支指令。

由于 v8 相对于 v7 而言，其指令集更像 MIPS，因此我们的结论是 ARMv7 和 ARMv8 的主要相同点仅仅是名字。

## 2.19 谬误与陷阱

谬误：更强大的指令意味着更高的性能。

x86 的一个强大的地方是可以通过前缀来改变后续指令的执行。某个前缀可以重复执行后面的指令直到一个计数器减少至 0。因此，为了在存储器中传输数据，看起来最自然的指令序列应该是使用加了重复前缀的 move 指令来实现 32 位的存储器到存储器的传输。

另外一种方法是使用所有计算机上都有的标准指令，将数据取到寄存器后再存回存储器。这种形式通过代码复制来减少循环开销，复制操作大约快 1.5 倍。第二种方式，使用更大的浮点寄存器代替 x86 的整数寄存器，复制操作比使用复杂指令快 2 倍。

谬误：使用汇编语言编程来获得较高的性能。

在一段时间内，编程语言的编译器经常产生很低级的指令序列。通过不断改进，编译器产生的代码与手工编写的代码在性能上的差距正在快速缩小。事实上，为了与当今编译器竞争，汇编程序员需要深刻理解第 4 章和第 5 章中的计算机体系结构概念（包括处理器流水线和存储器层次）。 159

编译器和汇编程序员之间的斗争正在逐渐消失。例如，C 为程序员提供一个指示编译器把变量保存在寄存器中而不是换出到存储器中的机会。当编译器在寄存器分配上能力较差时，这种指示对性能至关重要。事实上，一些较老的 C 语言课本花费大量的时间给出了有效的寄存器指示的例子。今天的 C 语言编译器通常忽略这种指示，因为编译器能比程序员更好地分配寄存器。

即使手工编写会产生更快的代码，汇编语言编写还是存在很多危险：需要更多时间编码和调试，可移植性差，难于维护。软件工程中少数几个被广泛接受的公理之一是编写的程序行数越多所花时间也越多。很明显使用汇编语言编写的程序比 C 或 Java 更长。一旦代码写好，下一个危险将是它会变成一个流行的程序。这种程序存在的时间总是比预期要长，意味着程序员需要每隔几年就更新一下代码使新的版本可以运行在新的操作系统和新机器上。高级语言而不是汇编语言编写的程序不仅可以使未来的编译器为未来的机器生成代码，还可以使软件易于维护并运行在其他类型的计算机上。

谬误：商用计算机二进制兼容的重要性意味着成功的指令集不需改变。

在向后的二进制兼容是神圣不可侵犯的同时，图 2-43 显示了 x86 指令集的快速发展。在 35 年中，平均每个月至少增加一条新的指令。

陷阱：忘记在字节寻址的机器中，连续的字地址相差不是 1。

很多汇编程序员假定下一个字地址可以通过将寄存器的值加 1 来获得，而不是增加一个字的字节数，这使他们犯下很多错误。提前注意以便有所准备！

陷阱：在自动变量的定义过程外，使用指针指向该变量。

处理指针的常见错误是使用指向一个过程中局部数组的指针，从该过程传出结果。遵从图 2-12 中的栈规则，当过程返回时，包含局部数组的存储器将立即被重新使用。指向自动变量的指针会造成混乱。 160

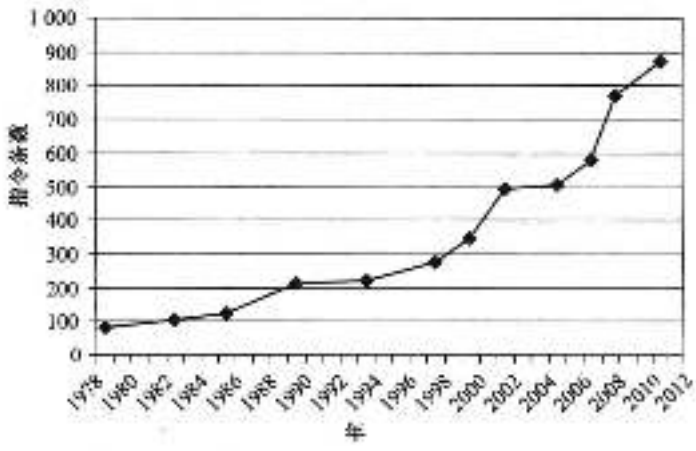



图 2-43 随时间推移 x86 指令集的增长。这种扩展是有一定的技术价值的，迅速的变化也增加了其他公司试图做兼容处理器的难度

## 2.20 本章小结

*少就是多。*

*——Robert Browning, 《Andrea del Sarto》, 1855*

存储程序计算机的两个准则是指令的使用与数字没有区别，以及使用可修改的存储器。这些准则使一台计算机可以在不同的领域辅助环境科学家、经济顾问和小说家。选择机器可以理解的指令集需要精妙的平衡程序执行需要的指令数目、指令执行所需的时钟周期数和时钟的速度。就像本章所描述的，在做精妙平衡时有 3 条准则可以指导设计者：

1）简单源于规整。规整性使 MIPS 指令集具有很多特点：所有指令长度统一、算术指令总是需要三个寄存器操作数和寄存器字段在每种指令格式的位置相同。

2）越小越快。对速度的要求导致 MIPS 只有 32 个寄存器而不是更多。

[161] 3）优秀的设计需要好的折中。一个 MIPS 例子是在指令中提供更大地址与常数，并且保持所有的指令具有相同的长度之间的折中。

与计算机体系结构相同，在指令集中也会看到“加速大概率事件”的伟大思想。该思想在 MIPS 中的体现包括条件分支的 PC 相对寻址和大常数操作数的立即数寻址。

机器语言之上是人们可读的汇编语言。汇编器将翻译为机器可以理解的二进制数，它甚至通过创造硬件中没有的符号指令来“扩展”指令集。例如，较大的常量和地址被切割成合适的大小，常用的指令变体都有它们自己的名字，等等。图 2-44 列举了到目前为止我们讲过的 MIPS 指令，包括实际指令和伪指令。在更高级别隐藏细节是伟大思想“抽象”的另外一个例子。

每一类 MIPS 指令与编程语言中出现的结构相关：

- 算术指令对应于赋值语句中的运算。
- 传输指令很可能发生在处理像数组和结构体这样的数据结构时。
- 条件分支被用于 if 语句和循环。
- 无条件分支被用于过程调用和返回以及 case/switch 语句。

这些指令出现频率不相等，少数指令出现频率较大。例如，图 2-45 展示了 SPEC CPU 2006 中每类指令出现的频率。指令出现频率的不同在数据通路、控制通路和流水线的特征分析中扮演重要角色。

| MIPS 指令 | 名称 | 格式 |
|---|---|---|
| 加 | add | R |
| 减 | sub | R |
| 加立即数 | addi | I |
| 取字 | lw | I |
| 存字 | sw | I |
| 取半字 | lh | I |
| 取无符号半字 | lhu | I |
| 存半字 | sh | I |
| 取字节 | lb | I |
| 取无符号字节 | lbu | I |
| 存字节 | sb | I |
| 取链接字 | ll | I |
| 存条件字 | sc | I |
| 取立即数高位 | lui | I |
| 与 | and | R |
| 或 | or | R |
| 或非 | nor | R |
| 与立即数 | andi | I |
| 或立即数 | ori | I |
| 逻辑左移 | sll | R |
| 逻辑右移 | srl | R |
| 相等时跳转 | beq | I |
| 不相等时跳转 | bne | I |
| 小于时置位 | slt | R |
| 小于立即数时置位 | slti | I |
| 小于无符号立即数时置位 | sltiu | I |
| 跳转 | j | J |
| 跳转至寄存器所指位置 | jr | R |
| 跳转和链接 | jal | J |

| MIPS 伪指令 | 名称 | 格式 |
|---|---|---|
| 移位 | move | R |
| 乘 | mult | R |
| 乘立即数 | multi | I |
| 取立即数 | li | I |
| 小于时跳转 | blt | I |
| 小于或等于时跳转 | ble | I |
| 大于时跳转 | bgt | I |
| 大于或相等时跳转 | bge | I |

图 2-44 到目前为止介绍过的 MIPS 指令集，左侧是真实的 MIPS 指令，右侧是伪指令。附录 A（A.10 节）描述了完整的 MIPS 体系结构。图 2-1 展示了与本章相关的更细致的 MIPS 体系结构。这里给出的信息可在 MIPS 参考数据卡的第 1 和第 2 列查到

| 指令类别 | MIPS 范例 | 相应的高级语言 | 出现频率 | |
|---|---|---|---|---|
| | | | 整型 | 浮点 |
| 算术 | add,sub,addi | 赋值语句中的操作 | 16% | 48% |
| 数据传输 | lw,sw,lb,lbu,lh,lhu,sb,lui | 对数据结构的引用，例如数组 | 35% | 36% |
| 逻辑 | and,or,nor,andi,ori,sll,srl | 赋值语句中的操作 | 12% | 4% |
| 条件分支 | beq,bne,slt,slti,sltiu | if 语句和循环 | 34% | 8% |
| 跳转 | j,jr,jal | 过程调用，返回，case/switch 语句 | 2% | 0% |

图 2-45 MIPS 指令分类、范例以及相应的高级编程语言结构和 SPEC CPU 2006 测试程序执行时定、浮点指令所占的比例。第 3 章中的图 3-26 展示了每条 MIPS 指令执行时所占的平均比例

在第 3 章解释计算机算术运算之后，我们将继续揭示 MIPS 指令集体系结构。

## 2.21 历史观点和拓展阅读

本节概述了指令集体系结构（ISA）的历史，我们介绍了编程语言和编译器的简短历史。

162 ~ 163

ISA 包括累加器体系结构、通用寄存器体系结构、栈体系结构和 ARM 及 x86 的简史。我们还回顾了高级语言计算机体系结构中的争议问题和精简指令集体系结构。编程语言的历史包括 Fortran、Lisp、Algol、C、Cobol、Pascal、Simula、Smalltalk、C++和 Java。编译器的历史包括重要的里程碑和实现它们的先驱。本节剩余部分在配套网站中的 2.21 节中。

## 2.22 练习题

附录 A 描述了对这些练习有帮助的 MIPS 的模拟器。尽管模拟器可以接受伪指令，但是在要求产生 MIPS 代码的习题中，尽量不要使用伪指令。你的目的是学习实际的 MIPS 指令集。如果统计指令数，你所给出的答案必须反映实际执行的指令数而不是伪指令。

有些情况必须使用伪指令（例如，当汇编时不知道真实值时，使用 la 指令）。还有些情况下，使用伪指令会更方便并使代码可读性更好（例如，li 和 move 指令）。如果你因为这些原因选择使用伪指令，请在伪指令开始的地方加上一两句话，说明你使用伪指令的原因。

**2.1** [5] <2.2> 下面的 C 语言表达式对应的 MIPS 汇编语言代码是什么？假设给定变量 f、g、h 和 i，像在 C 程序中声明的一样它们都是 32 位的整数，使用最少的 MIPS 汇编指令。

```
f = g + (h - 5);
```

**2.2** [5] <2.2> 下面的 MIPS 汇编语言程序段对应的 C 语言表达式是什么？

```
add  f, g, h
add  f, i, f
```

164

**2.3** [5] <2.2, 2.3> 下面的 C 语言表达式对应的 MIPS 汇编代码是什么？假设变量 f、g、h、i 和 j 分别赋值给寄存器 $s0、$s1、$s2、$s3 和 $s4。假设数组 A 和 B 的基地址分别在寄存器 $s6 和 $s7 中。

```
B[8] = A[i - j];
```

**2.4** [5] <2.2, 2.3> 下面的 MIPS 汇编语言程序段对应的 C 语言表达式是什么？假设变量 f、g、h、i 和 j 分别赋值给寄存器 $s0、$s1、$s2、$s3 和 $s4。假设数组 A 和 B 的基地址分别在寄存器 $s6 和 $s7 中。

```
sll  $t0, $s0, 2     # $t0 = f * 4
add  $t0, $s6, $t0   # $t0 = &A[f]
sll  $t1, $s1, 2     # $t1 = g * 4
add  $t1, $s7, $t1   # $t1 = &B[g]
lw   $s0, 0($t0)     # f = A[f]
addi $t2, $t0, 4
lw   $t3, 0($t2)
add  $t3, $t0, $s0
sw   $t0, 0($t1)
```

**2.5** [5] <2.2, 2.3> 在不改变功能的前提下，重写习题 2.4 中的 MIPS 程序使其指令数目尽可能少。

**2.6** 下表表示在主存中存放的一个数组的 32 位数据。

| Address | Data |
|---|---|
| 24 | 2 |
| 28 | 4 |
| 32 | 3 |
| 36 | 6 |
| 40 | 1 |

165

**2.6.1** [5] <2.2, 2.3> 基于上表中数据在存储器中的位置，编写一段 C 代码，将数据从小到大排序，最小的数放在地址最低的位置。（假设这段数据代表了 C 的一个 Int 型数组 Array，并且这台特别的机器是按照字节寻址的，且一个字包含 4 字节。）

**2.6.2** [5] <2.2, 2.3> 基于上表中数据在存储器中的位置，编写一段 MIPS 代码，将数据从小到大排序，最小的数放在地址最低的位置。（使用最少的 MIPS 汇编指令，假设 Array 的基地址保存在寄存器 $s6 中。）

**2.7** [5] <2.3> 分别画出数据 0xabcdef12 在大端编址和小端编址的机器上是如何分布在存储器中的。（假定数据从地址 0 开始存储。）

**2.8** [5] <2.4> 将 0xabcdef12 转化为十进制。

**2.9** [5] <2.2, 2.3> 把下面的 C 代码翻译为 MIPS 代码。假定变量 f、g、h、i 和 j 分别赋值给寄存器 $s0、$s1、$s2、$s3 和 $s4。假定数组 A 和数组 B 的基地址分别存放在 $s6 和 $s7 中。假定数组 A 和数组 B 中的元素均为 4 字节的字：

```
B[8] = A[i] + A[j];
```

**2.10** [5] <2.2, 2.3> 把下面的 MIPS 代码翻译为 C 代码。假定变量 f、g、h、i 和 j 分别赋值给寄存器 $s0、$s1、$s2、$s3 和 $s4。假定数组 A 和数组 B 的基地址分别存放在 $s6 和 $s7 中。

```
addi $t0, $s6, 4
add  $t1, $s6, $0
sw   $t1, 0($t0)
lw   $t0, 0($t0)
add  $s0, $t1, $t0
```

**2.11** [5] <2.2, 2.5> 对于每条 MIPS 指令，写出操作码（OP）、源操作数（RS）和目标操作数（RT）的值（value）。对于 I 型指令，写出立即数字段的值。对于 R 型指令，写出目的寄存器（RD）字段的值。 166

**2.12** 假定寄存器 $s0 和 $s1 分别存放数值 0x80000000 和 0x00000000。

**2.12.1** [5] <2.4> 下面汇编代码的 $t0 的值是多少？

```
add $t0, $s0, $s1
```

**2.12.2** [5] <2.4> $t0 中的结果是期望的结果还是发生溢出后的结果？

**2.12.3** [5] <2.4> 对于上面定义的寄存器 $s0 和 $s1 的内容，下面汇编代码的 $t0 的值是多少？

```
sub $t0, $s0, $s1
```

**2.12.4** [5] <2.4> $t0 中的结果是期望的结果还是发生溢出后的结果？

**2.12.5** [5] <2.4> 对于上面定义的寄存器 $s0 和 $s1 的内容，下面汇编代码的 $t0 的值是多少？

```
add $t0, $s0, $s1
add $t0, $t0, $s2
```

**2.12.6** [5] <2.4> $t0 中的结果是期望的结果还是发生溢出后的结果？

**2.13** 假定 $a0 中的值为 $128_{10}$。

**2.13.1** [5] <2.4> 对于指令 add $t0, $s0, $a1，求使结果产生溢出的 $a1 的值的范围。

**2.13.2** [5] <2.4> 对于指令 sub $t0, $s0, $a1，求使结果产生溢出的 $a1 的值的范围。

**2.13.3** [5] <2.4> 对于指令 sub $t0, $s1, $a0，求使结果产生溢出的 $a1 的值的范围。

**2.14** [5] <2.2, 2.5> 写出下面的二进制数值对应的类型和汇编语言指令：

0000 0010 0001 0000 1000 0000 0010 0000$_2$

**2.15** [5] <2.2, 2.5> 给出下面指令的类型和十六进制表示：sw $t1, 32 ($t2) 167

**2.16** [5] <2.5> 写出用下面 MIPS 字段描述的指令的类型、汇编语言指令和二进制表示：

```
op=0, rs=3, rt=7, rd 3, shamt=0, funct=34
```

**2.17** [5] <2.5> 写出用下面 MIPS 字段描述的指令的类型、汇编语言指令和二进制表示：

```
op=0x23, rs=1, rt=2, const=0x4
```

**2.18** 假设可以将 MIPS 寄存器文件扩展到 128 个寄存器，并将指令集中的指令数扩展为原来的 4 倍。

**2.18.1** [5] <2.5> 这将如何影响 R 型指令的每个位字段的大小？

**2.18.2** [5] <2.5> 这将如何影响 I 型指令的每个位字段的大小？

**2.18.3** [5] <2.5, 2.10> 在提出的这两种变化中，每种变化如何减少一个 MIPS 汇编程序的大小？另一方面，如何增大一个 MIPS 汇编程序的大小？

**2.19** 假设如下寄存器内容：

```
$t0 = 0xAAAAAAAA, $t1 = 0x12345678
```

**2.19.1** [5] <2.6> 对于以上的寄存器内容，执行下面的指令序列后 $t2 的值是多少？

```
slt $t2, $t0, 44
or  $t2, $t2, $t1
```

**2.19.2** [5] <2.6> 对于以上的寄存器内容，执行下面的指令序列后 $t2 的值是多少？

```
sll  $t2, $t0, 4
andi $t2, $t2, -1
```

**2.19.3** [5] <2.6> 对于以上的寄存器内容，执行下面的指令序列后 $t2 的值是多少？

```
srl  $t2, $t0, 3
andi $t2, $t2, 0xFFEF
```

168

**2.20** [5] <2.6> 找出完成如下功能的最短的 MIPS 指令序列：从寄存器 $t0 中提取第 16 位到第 11 位，然后使用这些位替换寄存器 $t1 的第 31 位到第 26 位，保持其他位不变。

**2.21** [5] <2.6> 写出可用来实现下面伪指令的 MIPS 指令集的最小子集：

```
not $t1, $t2      // bit-wise invert
```

**2.22** [5] <2.6> 对于下面的 C 语言表达式，写一个能够完成同样操作的最短 MIPS 汇编指令程序段。假设 $t1 = A，$t2 = B，$s1 是 C 的基地址。

```
A = C[0] << 4;
```

**2.23** [5] <2.7> 假设 $t0 中存放数值 0x00101000，在执行下列指令后 $t2 的值是多少？

```
      slt  $t2, $0,  $t0
      bne  $t2, $0,  ELSE
      j    DONE
ELSE: addi $t2, $t2, 2
DONE:
```

**2.24** [5] <2.7> 假设程序计数器（PC）被设置为 0x2000 0000，是否可以使用 MIPS 的跳转（j）指令将 PC 设置为地址 0x4000 0000？是否可以使用 MIPS 的相等则分支（beq）指令将 PC 设置为该地址？

**2.25** MIPS 指令集不包含下面的指令：

```
rpt $t2, loop # if(R[rs]>0) R[rs]=R[rs]-1, PC=PC+4+BranchAddr
```

**2.25.1** [5] <2.7> 如果要在 MIPS 指令集中实现该指令，哪种指令格式最合适？

169

**2.25.2** [5] <2.7> 能够实现相同操作的最短 MIPS 指令序列是什么？

**2.26** 考虑如下的 MIPS 循环：

```
LOOP: slt  $t2, $0,  $t1
      beq  $t2, $0,  DONE
      subi $t1, $t1, 1
      addi $s2, $s2, 2
      j    LOOP
DONE:
```

**2.26.1** [5] <2.7> 假设寄存器 $t1 的初始值为 10，假设 $t2 初始值为 0，则循环执行完毕时寄存器 $t2 的值是多少？

**2.26.2** [5] <2.7> 对于上面的循环体，写出等价的 C 代码例程。假定寄存器 $s1、$s2、$t1 和 $t2 分别为整数 A、B、i 和 temp。

**2.26.3** [5] <2.7> 假定寄存器 $t1 的初始值为 N，上面的 MIPS 汇编循环执行了多少条指令？

**2.27** [5] <2.7> 将下面的 C 代码翻译为 MIPS 汇编代码。要求使用的指令数目最少。假设值 a、b、i 和 j 分别存放在寄存器 $s0、$s1、$t0 和 $t1 中。另外假设寄存器 $s2 中存放着数组 D 的基地址。

```
for(i=0; i<a; i++)
   for(j=0; j<b; j++)
      D[4*j] = i + j;
```

**2.28** [5] <2.7> 实现习题 2.27 中的 C 代码用了多少条 MIPS 汇编指令？如果变量 a 和 b 分别初始化为 10 和 1，并且 D 中所有元素初始化为 0，将整个循环执行完成时，一共执行了多少条 MIPS 指令？

**2.29** [5] <2.7> 将下面的循环翻译成 C 代码。假定寄存器 $t1 中存放 C 语言级的整数 i，$s2 中存放 C 语言级的整数 result，$s0 存放整数数组 MemArray 的基地址。

```
        addi $t1, $0, $0
LOOP:   lw   $s1, 0($s0)
        add  $s2, $s2, $s1
        addi $s0, $s0, 4
addi $t1, $t1, 1
slt1 $t2, $t1, 100
bne  $t2, $s0, LOOP
```

170

**2.30** [5] <2.7> 将习题 2.29 中的循环重写以减少执行的 MIPS 指令。

**2.31** [5] <2.8> 使用 MIPS 汇编实现下面的 C 代码。该函数一共执行了多少条 MIPS 指令？

```
int fib(int n){
   if (n==0)
     return 0;
   else if (n == 1)
     return 1;
   else
     return fib(n-1) + fib(n-2);
```

**2.32** [5] <2.8> 函数经常被编译器实现为内联 "in-line" 的形式。内联函数是将函数体复制到程序空间中，以消除函数调用的开销。对于上面的函数，请用 MIPS 汇编实现内联版本的 C 代码。请问实现这个函数总共可以减少多少条 MIPS 汇编指令？（假设 C 的变量 n 被初始化为 5。）

**2.33** [5] <2.8> 对于每一次函数调用，画出调用后栈的内容。（假定栈指针被初始化为 0x7FFFFFFC，寄存器的使用情况和图 2-11 相同。）

**2.34** 将下面的函数翻译成 MIPS 汇编语言。如果需要使用寄存器 $s0 到 $s7，请从编号小的寄存器开始使用。假设函数 func 的声明为 "int f (int a, int b);"，函数 f 的代码如下：

```
int f(int a, int b, int c, int d){
  return func(func(a,b),c+d);
}
```

171

**2.35** [5] <2.8> 请问这个函数可以使用尾调用优化吗？如果不能，请说明原因。如果能，请说明优化前后执行 f 的指令数的差别。

**2.36** [5] <2.8> 在习题 2.34 中函数 f 返回之前，我们可以知道寄存器 $t5、$s3、$ra 和 $sp 的内容吗？（注意，我们知道函数 f 的全部，但是我们只知道函数 func 的声明。）

**2.37** [5] <2.9> 用 MIPS 汇编语言写一段代码将包含十进制正整数和负整数的 ASCII 码的数串转换成整数。在程序中使用寄存器 $a0 处理由数字 0 - 9 组成的非空串的地址。程序应该计算与这个数字串等值的整数，并将这个整数存放在寄存器 $v0 中。如果在字符串的任意位置出现非数字字符，程序停止并将 -1 存入 $v0。例如，如果寄存器 $a0 指向 3 字节的序列 $50_{10}$、$52_{10}$、$0_{10}$（非终结的字符串 "24"），当程序停止的时候，寄存器 $v0 中的值应该是 $24_{10}$。

**2.38** [5] <2.9> 对于如下代码：

```
lbu $t0, 0($t1)
sw  $t0, 0($t2)
```

假设寄存器 $t1 中存放地址 0x1000 0000，寄存器 $t2 中存放地址 0x1000 0010。注意 MIPS 体系结构使用大端地址。假设地址 0x1000 0000 的数据是 0x11223344。寄存器 $t2 指向的地址中存放的数值是多少？

**2.39** [5] <2.10> 请编写能产生 32 位常数 0010 0000 0000 0001 0100 1001 0010 0100，的 MIPS 代码，并将值存储到寄存器 $t1 中。

**2.40** [5] <2.6, 2.10> 如果当前 PC 值是 0x00000000，可以使用单独的跳转指令跳转到练习题 2.39 中所指定的 PC 地址吗？

**2.41** [5] <2.6, 2.10> 如果当前 PC 值是 0x00000600，可以使用单独的分支指令跳转到练习题 2.39 中所指定的 PC 地址吗？

172

**2.42** [5] <2.6, 2.10> 如果当前 PC 值是 0x1FFF FDD0，可以使用单独的分支指令跳转到练习题 2.39 中所指定的 PC 地址吗？

**2.43** [5] <2.11> 写出实现下面 C 代码的 MIPS 汇编代码：

```
lock(lk);
shvar=max(shvar,x);
unlock(lk);
```

假设变量 lk 的地址在 $a0 中，变量 shvar 的地址在 $a1 中，变量 x 的地址在 $a2 中。你所编写的这个重要部分的代码不能包含任何函数调用。使用 ll/sc 指令实现 lock() 操作，而 unlock() 操作可以简单地使用存数指令。

**2.44** [5] <2.11> 重新解决练习题 2.43 中的问题，不过这次使用 ll/sc 直接完成 shvar 变量的原子更新操作，不使用 lock() 和 unlock()。注意这个问题中没有变量 lk。

**2.45** [5] <2.11> 以练习题 2.43 中的代码为例，解释当两个处理器同时执行这段临界区域时，将发生什么情况？假设每个处理器执行一条指令正好需要一个周期。

**2.46** 假设给定处理器的算术指令的 CPI 是 1，取数/存数指令的 CPI 是 10，分支指令的 CPI 是 3。假设一个程序由 5 亿条算术指令、3 亿条取数/存数指令和 1 亿条分支指令组成。

**2.46.1** [5] <2.19> 假设向指令集中添加了新的、功能更强的算术指令。通过使用这些功能更强大的算术指令平均可以减少程序执行所需要的 25% 的算术指令，而时钟周期的开销增长了 10%。请问这是好的设计选择吗？为什么？

**2.46.2** [5] <2.19> 假设我们找到一种可以使算术指令性能达到原来两倍的方法。请问我们机器的整体加速是多少？假设我们找到一种可以使算术指令性能达到原来 10 倍的方法，那么机器的性能整体加速又是多少？

173

**2.47** 假设一给定程序共执行了 70% 的算术指令、10% 的取数/存数指令和 20% 的分支指令。

**2.47.1** [5] <2.19> 假设执行一条算术指令、取数/存数指令和分支指令分别需要 2 个周期、6 个周期和 3 个周期，求平均 CPI。

**2.47.2** [5] <2.19> 在取数/存数指令和分支指令执行时间不变的情况下，如果要使性能提升 25%，则算术运算指令的平均执行时间应该为多少？

**2.47.3** [5] <2.19> 在取数/存数指令和分支指令执行时间不变的情况下，如果要使性能提升 50%，则算术运算指令的平均执行时间应该为多少？

### 小测验答案

**2.2** MIPS，C，Java

**2.3** 2. 非常慢

**2.4** 2. $-8_{10}$。

**2.5** 4. sub $t2, $t2, $t1

**2.6** 都可以。将“逻辑与”和全“1”的掩码一起使用会导致除了想要的区域之外，都变成 0。正确的左移位操作将左边的位数都移走。合适的右移将一个字最右边的区域都移走，将 0 留在字中。注意到“逻辑与”操作会保留原始的值，移位操作对将需要的区域移动到字的最右边。

**2.7** I. 全对，II. 1。

**2.8** 两个都正确。

**2.9** I. 1 和 2，II. 3。

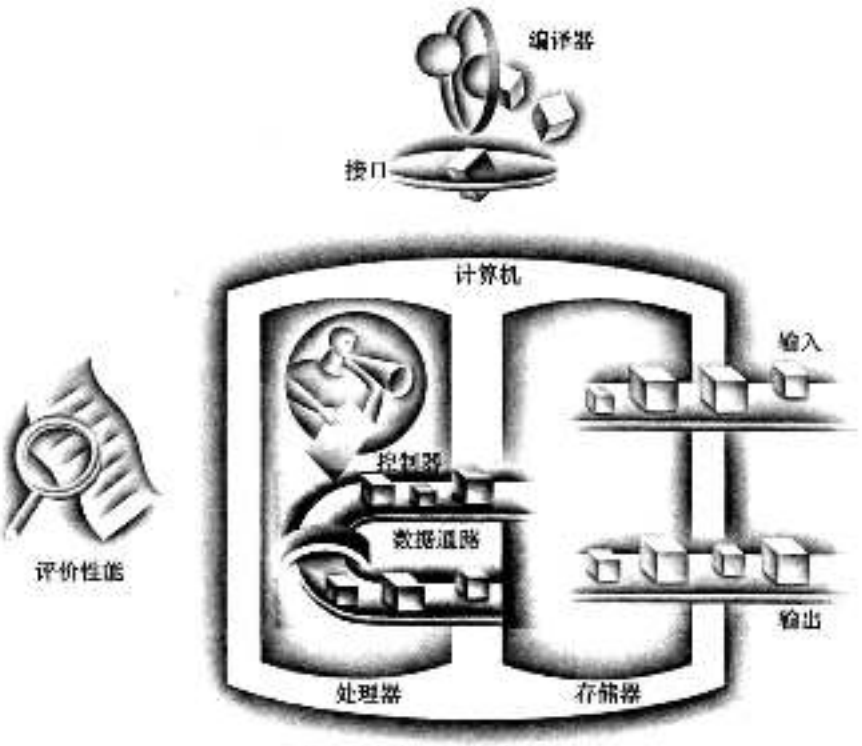
**2.10** I. 4. + 128K，II. 6. 一个 256M 的块，III. 4. all。

174 ~ 175

**2.11** 两个都正确。

**2.12** 4. 与机器无关。

# 计算机的算术运算




计算机的5个经典部件

数值的精确度是科学的灵魂。

——*Sir D'arcy Wentworth Thompson,《On Growth and Form》, 1917*

## 3.1 引言

计算机中的字由位组成。因此，字可以用二进制数来表示。第2章里提到整数可以表示成十进制或者二进制形式，但是其他常用的数据如何表示？例如：

- 小数和其他实数如何表示？
- 当一个操作生成了一个无法表示的大数时如何处理？
- 上述问题隐含着一个秘密：怎样用硬件真正地做乘法和除法？

本章的目的就是要揭示这些秘密，包括实数的表示方法、算术的算法、实现这些算法的硬件，以及如何在指令集中表示有关的内容。有了这些知识后，你就能解释在使用计算机的过程中遇到的各种不明白的事情了。另外，我们还将介绍如何使用该技术加速算术运算密集型程序的运行。

## 3.2 加法和减法

减法：加法的微妙朋友。

——*No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman 等,《Book of Top Ten Lists》, 1990*

加法是计算机中必备的操作。数据从右到左逐位相加，同时进位也相应地向左传播，就如手动计算一样。减法也可采用加法实现：减数在简单的取反之后再进行加法操作。

**例题：二进制加法和减法**

在二进制下，首先计算$7_{10}$加上$6_{10}$，然后计算$7_{10}$减去$6_{10}$。

```
  0000 0000 0000 0000 0000 0000 0000 0111₂  = 7₁₀
+ 0000 0000 0000 0000 0000 0000 0000 0110₂  = 6₁₀
------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 1101₂  = 13₁₀
```

只有右边4位发生变化。图3-1给出了和位与进位。其中，进位放在括号里，箭头标记了进位的方向。

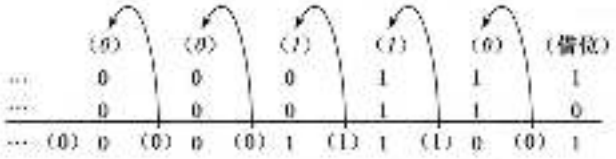



图3-1 二进制加法，显示了从右到左的进位。最右边的位将1和0相加，得到该位的和为1，该位的进位为0。因此，右边第二位数的操作是0+1+1。该操作的和为0，进位为1。第三位是1+1+1的和，得到的进位为1，和为1。第四位是1+0+0，和为1，无进位

176 ~ 178

**答案**

$7_{10}$减去$6_{10}$可以直接操作：

```
  0000 0000 0000 0000 0000 0000 0000 0111₂  = 7₁₀
- 0000 0000 0000 0000 0000 0000 0000 0110₂  = 6₁₀
------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0001₂  = 1₁₀
```

或者通过加上−6的二进制补码来实现：

```
  0000 0000 0000 0000 0000 0000 0000 0111₂  = 7₁₀
+ 1111 1111 1111 1111 1111 1111 1111 1010₂  = −6₁₀
------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0001₂  = 1₁₀
```

□

硬件规模总是有一定限制的，如字宽只有32位。当运算结果超过这个限制时，就会发生溢出。加法在什么情况下会溢出呢？当相加的两个源操作数符号相异时，不会发生溢出，原因是和必然不会大于其中一个操作数。如 −10 + 4 = −6。因为源操作数可以用32位的字表示，而“和”不会大于其中任何一个源操作数，所以和也可以用32位来表示。因此，当正数和负数相加时不会发生溢出。

在做减法时也会有类似的情况，只不过采用的规则相反：当源操作数的符号相同时，不会发生溢出。我们知道，$c-a=c+(-a)$，这是因为减法是把第二个源操作数变相反符号然后相加，所以当两个同符号的数作减法时，实际上是把两个符号相异的数相加，也不会发生溢出。

知道溢出在加减法中何时不会发生固然重要，但如何检测它何时发生？很明显，加或者减两个32位的数可能产生需要用33位来表示的结果。

如果缺少了第33位，则溢出发生时，符号位就可能被数值位占用而产生错误。因此，当两个正数相加但结果为负时，就说明发生了溢出。反之亦然。这个问题的和表示计算过程中发生了向符号位的进位操作。

在做减法时，如果用一个正数减去一个负数得到一个负的结果，或者用一个负数减去一个正数然后得到一个正的结果，则发生了溢出。这也意味着借位占用了符号位。图3-2给出了发生溢出的条件。

179

| 操作 | 操作数A | 操作数B | 表示结果有溢出的条件 |
|---|---|---|---|
| A+B | ≥0 | ≥0 | <0 |
| A+B | <0 | <0 | ≥0 |
| A-B | ≥0 | <0 | <0 |
| A-B | <0 | ≥0 | ≥0 |

图3-2 加减法的溢出条件

上面介绍了如何检测计算机中的二进制补码操作的溢出，但无符号整数的溢出情况是如何的呢？由于无符号数通常用于表示内存地址，这种情况下的溢出可以忽略。

因此，计算机设计者必须提供一种方法，能够在某些情况下忽略溢出的发生，而在另一些情况下则能进行溢出的检测。MIPS采用两种类型的算术指令来解决这个问题：

- 加法（add）、立即数加法（addi）和减法（sub），这三条指令在溢出时产生异常。
- 无符号加法（addu）、立即数无符号加法（addiu）和无符号减法（subu），这三条指令在发生溢出时不会产生异常。

因为C语言忽略溢出，所以MIPS C编译器总是采用无符号的算术指令addu、addiu和subu，而不必考虑变量的类型。但是MIPS Fortran编译器会根据操作数的类型来选择相应的算术指令。

附录B描述了做加减法的**算术逻辑单元**（arithmetic logic unit，ALU）的硬件实现。

- 算术逻辑单元（ALU）：用于执行加法、减法，通常也包括如逻辑与、逻辑或等逻辑操作的硬件。
- 异常：也叫中断，一种打断正常程序执行过程的事件，用于溢出检测。
- 中断：来自处理器外部的异常。（在某些体系结构中所有的异常都称为中断。）

**精解** 对于addiu的一个常见困惑是它的名字和对其立即数字段做什么操作。u代表无符号数，这意味着加法操作不会产生溢出异常。然而，与addi、slti和sltiu指令类似，16位立即数字段要符号扩展为32位。因此，即使操作是“无符号”的，立即数字段也是有符号的。

**硬件/软件接口** 计算机设计者必须考虑如何处理算术溢出。但是一些编程语言（如C和Java）会忽略整数溢出，而Ada和Fortran语言则需要通知程序溢出。因此程序员或者是编程环境必须决定在溢出发生时如何处理。

MIPS检测到溢出时会产生异常（exception），在许多计算机系统中也叫作中断（interrupt）。从本质上来说，异常或中断是一种打断正常过程的系统调用。产生溢出的指令地址保存在一个寄存器中，而后计算机会跳到一个预先设定好的地址去执行相应的异常处理程序。保存异常地址的目的是为了在某些条件下能够在异常处理程序执行完后返回原程序继续执行。（4.9节给出了有关异常的更多细节；第5章和第6章中描述了异常和中断发生的其他条件。） 180

MIPS使用命名为异常程序计数器（Exception Program Counter，EPC）的寄存器来保存导致异常的指令地址。指令mfc0（move from system control）用来将EPC存入一个通用寄存器，从而使MIPS软件可以通过寄存器跳转指令返回到导致异常的指令那里。

## 小结

本节主要指出，无论采用哪种数的表示方法，具有有限字长的计算机在进行算术操作时都

可能发生溢出。无符号数的溢出是容易检测的，但无符号数通常用于地址计算，因为程序通常并不需要检测地址计算的溢出，通常情况是使用自然数，所以这些溢出往往被忽略。有符号数的溢出检测比较麻烦，但是有些软件系统需要检测溢出，所以今天所有的计算机都支持溢出检测。

**小测验** 某些程序语言支持字节或者半字的二进制补码的整数算术，而 MIPS 只有整字的整数算数操作。回顾一下第 2 章的内容，MIPS 中也有字节和半字的数据传送指令。那么将会使用哪些 MIPS 指令？

1. 取数使用 lbu、lhu；算术操作采用 add、sub、mult、div；存数采用 sb、sh。
2. 取数使用 lb、lh；算术操作采用 add、sub、mult、div；存数采用 sb、sh。
3. 取数使用 lb、lh；算术操作采用 add、sub、mult、div；采用 AND 来屏蔽每次运算的结果到 8 位或者 16 位；存数采用 sb、sh。

**精解** 饱和（saturating）操作是通用微处理器中一个不常出现的特性。饱和意味着当计算结果溢出时，结果被设置为最大的正数或者最小的负数，而不像二进制补码运算那样采用取模操作来获得结果。饱和操作一般更适合多媒体操作。例如，当不断旋转收音机音量的旋钮时，刚初声音逐渐增大，但如果大到一定值后声音突然变小，那么这样的收音机设计是不令人满意的。然而，对一台有饱和操作的收音机，当向最大值方向旋转音量旋钮到一定程度后，即使再旋转，音量也只会停在最大值上。标准指令集上媒体扩展通常提供饱和算法。

181

**精解** MIPS 在溢出时会产生异常，但和其他许多计算机不同，它没有测试溢出的条件分支。一个 MIPS 指令序列可以发现溢出。对于有符号加法，这个序列如下（见 2.6 节溢出与 xor 指令的精解）：

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs =,
                             # so no overflow
xor $t3, $t0, $t1 # signs =; sign of sum match too?
                  # $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs =; goto overflow
```

对于无符号加法（$t0 = $t1 + $t2），测试则为：

```
addu $t0, $t1, $t2      # $t0 = sum
nor $t3, $t1, $zero     # $t3 = NOT $t1
                        # (2's comp - 1: 2^N - $t1 - 1)
sltu $t3, $t3, $t2      # (2^N - $t1 - 1) < $t2
                        # ⇒ 2^N - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^N-1<$t1+$t2) goto overflow
```

**精解** 在前文中我们说过，可以通过 mfc0 指令将 EPC 内容复制到一个寄存器，然后通过跳转寄存器返回到被中断的代码。这样做会导致一个有趣的问题：既然必须首先使用跳转寄存器传输 EPC 到一个寄存器，那么跳转寄存器该如何返回到被中断的位置，并恢复所有寄存器的原值呢？如果先恢复所有寄存器的原值，则来自 EPC 的返回地址就会被破坏。如果在恢复所有寄存器的原值时保留那个返回地址的寄存器不变，这样可以进行正确跳转，但是这也意味着在程序执行的任何时刻，异常会导致一个寄存器的值无法被恢复。两者都是不可行的。

为了将硬件设计从这一困境中解救出来，MIPS 允许程序员将寄存器 $k0 和 $k1 预留给操作系统。这些寄存器在异常时不会恢复。仅仅当 MIPS 编译器避免使用 $at 寄存器时，汇编程序可以使用它作为临时寄存器（见 2.10 节的硬件/软件接口），编译器也可以避免使用寄存器 $k0 和 $k1，从而使它们空出来给操作系统。异常处理程序将返回地址放在其中的一个寄存器中，然后利用跳转寄存器返回指令地址。

**精解** 确定进位到达高位的速度变快，加法的速度也随之加快。有许多方案可以用来加速这个进位，最坏情况下的进位时间是加法器位长的 $\log_2$ 的函数。预期信号传输更快是因为它们经过了更少的门电路序列，而加速进位需要更多门电路，最流行的结构是超前进位（carry lookahead）加法器，见附录 B 的 B.6 节。 182

## 3.3 乘法

*乘法令人恼怒，除法更甚；比例运算因就着我，做练习令我发疯。*

——*佚名，《Elizabethan manuscript》，1570*

现在我们已经完成了对加法、减法的解释，本节开始分析更复杂的乘法操作。

首先，通过用普通写法表示的十进制数乘法来回忆一下乘法的步骤和操作数的名称。为简单起见，我们只用十进制数中的 0 和 1 来作为例子，计算 $1000_{10}$ 乘以 $1001_{10}$：

```
被乘数          1000₁₀
乘数      ×     1001₁₀
                 1000
                0000
               0000
              1000
积            1001000₁₀
```

第一个源操作数称为被乘数（multiplicand），第二个源操作数称为乘数（multiplier），最终的结果称为积（product）。你会回忆起在学校学过的乘法规则：每次从右到左选取乘数的一位，乘以被乘数，然后相对上一个中间积，将当前积左移一位。

可以观察到，积的位数远远大于被乘数和乘数。事实上，如果我们忽略符号位，若被乘数为 $n$ 位，乘数为 $m$ 位，则积的位数为 $n+m$。即需要 $n+m$ 位来表示所有可能的积。因此，像加法一样，乘法也需要处理溢出，因为我们经常需要两个 32 位长的数相乘产生一个 32 位长的积。

在这个例子中，我们只使用了十进制中的 0 和 1。因为只有两个选择，所以每一步的乘法都很简单：

1）当乘数位为 1 时，只需要将被乘数（1 × 被乘数）复制到合适的位置。

2）当乘数位为 0 时，将 0（0 × 被乘数）放置到合适的位置。

虽然上面十进制的例子是限制使用了 0 和 1，但二进制数的乘法只能使用 0 和 1，因此也只有这两种选择。

分析了乘法的基本原则之后，一般来讲下一步就会马上开始介绍乘法硬件及其优化。但为了更好地理解这一问题，我们打破这一传统，先通过倍数的生成来展示乘法硬件和算法的进化过程。首先，我们假设只使用正数为源操作数。 183

### 3.3.1 顺序的乘法算法和硬件

该设计模拟我们在小学学过的算法。图 3-3 给出了硬件结构。我们画出了硬件，使得数据

流从顶至下，很像我们用纸和笔计算的方法。

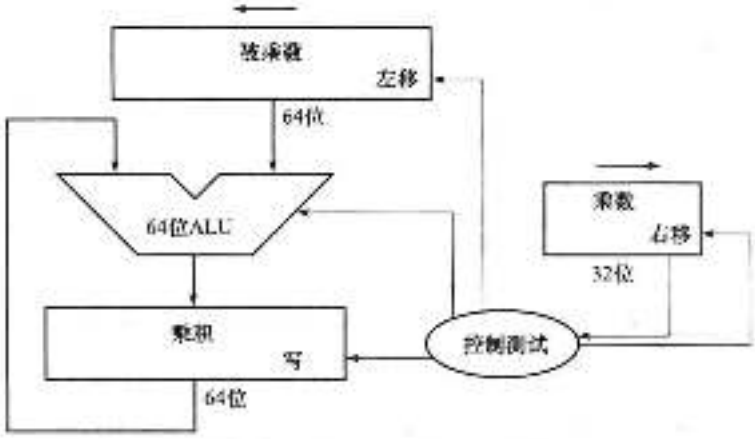



图 3-3 第一版乘法器硬件结构。被乘数寄存器、ALU 和积寄存器都是 64 位长，而乘数寄存器为 32 位长。（附录 B 对 ALU 进行了描述。）32 位的被乘数在开始时放置在被乘数寄存器的右半部分，然后每次左移一位。乘数则每次向相反的方向移动。算法开始时，积被初始化为 0。控制逻辑决定何时对被乘数和乘数寄存器进行移位，以及何时将新值写入积寄存器

假设乘数放置在 32 位的乘数寄存器中，64 位的积寄存器被初始化为 0。从采用纸和笔计算的方法中，我们可以清楚地看到被乘数在每步需要左移一位，因为它需要与前面的中间结果相加。在经过 32 步后，32 位长的被乘数将要左移 32 位。因此，我们还需要一个 64 位的被乘数寄存器，且在初始化时 32 位的被乘数放在右半部分，左半部分清 0。然后，每执行一步，这个寄存器中的值就左移一位，将被乘数与 64 位积寄存器中的中间结果对齐并累加到中间结果。

图 3-4 给出了对于操作数的每一位的三个基本执行步骤。乘数的最低位（乘数的第 0 位）决定了被乘数是否被加到积寄存器上。第二步中的左移起着将被乘数左移的作用，就如同用纸和笔做乘法一样。第三步中的右移给出了下一个迭代中要用的乘数位。这三个步骤要重复执行 32 次来获得积。如果每步需要一个时钟周期，这个算法将需要大概 100 个时钟周期来完成两个 32 位的数相乘。像乘法这样的算术操作的相对重要性因程序而异，一般加法和减法出现的次数要比乘法频繁 5～100 倍。因此，在许多应用程序中，多步乘法不会显著影响性能。但 Amdahl 定律（见 1.10 节）告诉我们，如果一个慢速操作在程序中占据一定比重的话，也会限制程序的性能。

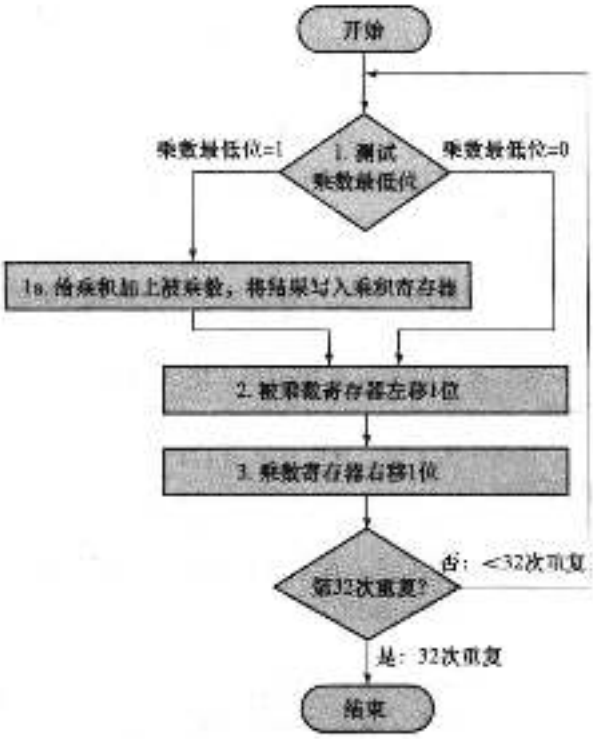



图 3-4 第一种乘法算法。其硬件结构见图 3-3。如果乘数的最低有效位为 1，则将被乘数加在积上，否则，进入下一步。在下两步中进行被乘数的左移和乘数的右移。这三个步骤需要重复 32 次

这个算法和硬件结构可以很容易改进

成每一步只需要一个时钟周期。这些操作可以并行化来加速执行：当乘数位为1时，将乘数和被乘数进行移位，同时将被乘数和积相加。这时需要保证硬件测试的是乘数最右边的位，而且得到的是被乘数移位前的值。注意到加法器和寄存器中有未使用的部分后，可以通过将加法器和寄存器的位长减半来进一步优化这个硬件结构。如图3-5所示为修正后的硬件。 184 ! 185

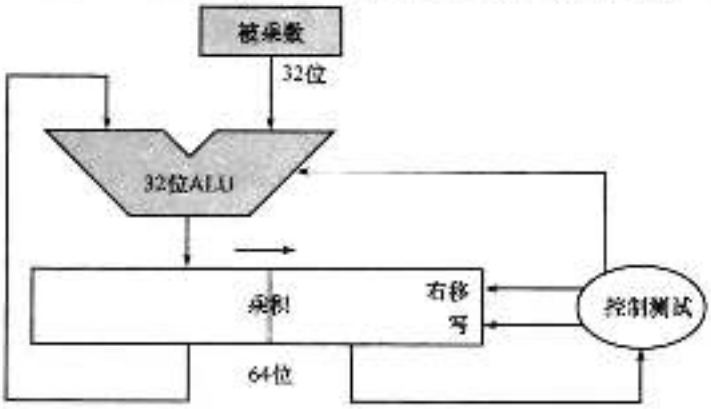



图3-5 乘法器硬件的改进版。与图3-3中的第一版硬件结构相比，被乘数寄存器、ALU、乘数寄存器都是32位长，只有积寄存器是64位长。现在将积进行右移，单独的乘数寄存器也撤销了。乘数放在积寄存器的右半部分（乘法寄存器实际上应该是65位，以保存加法器的进位，但这里给出的是64位，以突出从图3-3的演变。）

**硬件/软件接口** 当乘数为常数时，乘法也可以用移位来替代。一些编译器将有短常数的乘法替换为一系列的移位和加法。因为左移一位就是将一个数放大两倍，左移和乘以2为底的指数有着等同的效果。正如第2章所提到的，几乎每个编译器都将以2为底的指数乘法替换为移位来进行优化。

**例题·乘法算法**

为了节省空间，使用的是4位长的数，计算$2_{10}\times3_{10}$，或$0010_2\times0011_2$的积。

**答案**

图3-6给出了按图3-4中标出的每一步执行后各个寄存器的值，最终结果为$0000\ 0110_2$，即$6_{10}$。加粗数据表示每步中寄存器值的变化。带圈的位用于决定下一步的操作。 186

| 迭代次数 | 步骤 | 乘数 | 被乘数 | 乘积 |
|---|---|---|---|---|
| 0 | 初始值 | 0011 | 0000 0010 | 0000 0000 |
| 1 | 1a：1⇒乘积=乘积+被乘数 | 0011 | 0000 0010 | **0000 0010** |
| | 2：左移被乘数 | 0011 | **0000 0100** | 0000 0010 |
| | 3：右移乘数 | **001①** | 0000 0100 | 0000 0010 |
| 2 | 1a：1⇒乘积=乘积+被乘数 | 0001 | 0000 0100 | **0000 0110** |
| | 2：左移被乘数 | 0001 | **0000 1000** | 0000 0110 |
| | 3：右移乘数 | **000①** | 0000 1000 | 0000 0110 |
| 3 | 1：0⇒无操作 | 0000 | 0000 1000 | 0000 0110 |
| | 2：左移被乘数 | 0000 | **0001 0000** | 0000 0110 |
| | 3：右移乘数 | **000①** | 0001 0000 | 0000 0110 |
| 4 | 1：0⇒无操作 | 0000 | 0001 0000 | 0000 0110 |
| | 2：左移被乘数 | 0000 | **0010 0000** | 0000 0110 |
| | 3：右移乘数 | **0000** | 0010 0000 | 0000 0110 |

图3-6 使用图3-4中算法的乘法例子。圆圈圈起来的是下一步需要检测的位 □

### 3.3.2 有符号乘法

到目前为止，我们处理的对象都是正数。对于理解如何处理有符号乘法，最简单的方法是首先将被乘数和乘数转化为正数，并记住原来的符号位。这样，就可用上述最后的算法迭代31次，符号位不必参与运算。当符号相异时，积为负。

这表明后面的算法对于有符号数同样适用，只要知道这里要使用32位数据来表示通常要处理的无限数字。因此移位步骤需要对有符号数的乘积进行扩展。当算法结束时，低位字将是32位乘积。

### 3.3.3 更快速的乘法

摩尔定律为我们提供了非常充足的资源，使硬件设计者可以设计更快速的乘法器。我们可以在乘法运算开始的时候通过检查乘数的32位，来判定被乘数是否被加上。快速的乘法运算主要的思想是为乘数的每一位提供一个32位的加法器：一个用来输入被乘数和一乘数位相与的结果，另一个是上一个加法器的输出。

一种直接的方法是将每个右边的加法器的输出作为左边加法器的输入，形成一个高32的加法器栈。一种替换的方法是将32个加法器组织成一个并行树，如图3-7所示。这样，我们
187 只需要等待 $\log_2(32)$，即5次32位长加法的时间，而不是等待32次加法的时间。

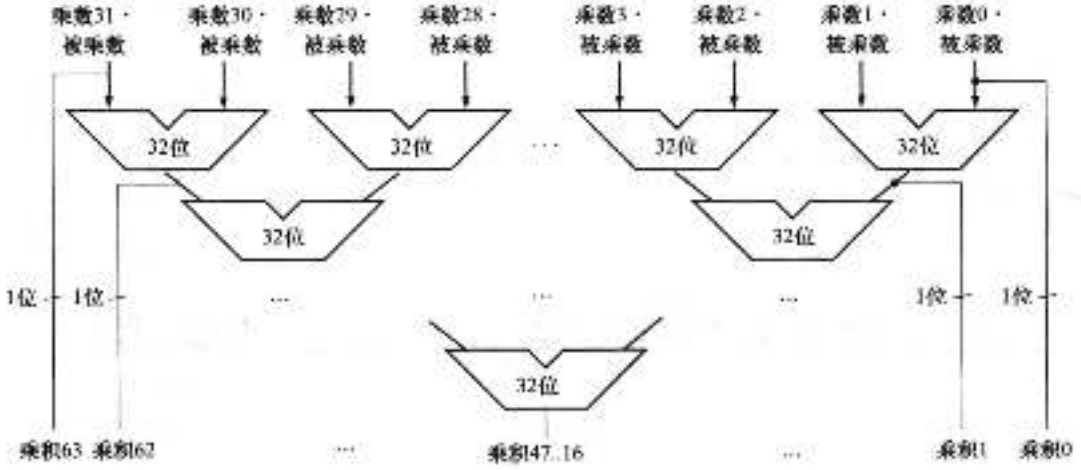



图3-7 快速乘法器硬件结构。这个结构使用31个加法器“展开循环”来实现最小的时延，而不再是使用单个32位的加法器31次

事实上，通过使用进位保留加法器（见附录B的B.6节），乘法的计算速度可以快于5次加法。而且由于易于应用流水线设计执行，这样的结构可以同步支持多个乘法运算（见第4章）。

### 3.3.4 MIPS中的乘法

MIPS提供了一对单独的32位寄存器来容纳64位的积，称为Hi和Lo。为了产生正确的有符号积和无符号积，MIPS提供了两条指令：乘法（mult）和无符号乘法（multu）。为了取得32位的整数积，程序员需要使用mflo指令（move from lo）。MIPS汇编器为乘法生成了一条伪指令，它使用了三个通用寄存器，用mflo和mfhi指令将积送入指定的寄存器。

### 3.3.5 小结

乘法硬件只是简单地移位和加法，其算法类似于采用纸和笔的计算方法。编译器甚至会用移位指令来代替乘数为 2 的幂次的乘法操作。通过使用更多硬件的方法，可以并行做加法操作，从而提高运算速度。

**硬件/软件接口** MIPS 乘法指令都忽略溢出，所以要由软件来检测是否因积过大而 32 位不够表示。对于 multu 指令，如果 Hi 为 0 则无溢出；对于 mult 指令，如果 Hi 为 Lo 的符号位则也无溢出。可以使用指令 mfhi（move from hi）将 Hi 的值移入一个通用寄存器来检测溢出。

188

## 3.4 除法

*Divide et impera*

——拉丁语，意为“分而治之”，引自 Machiavelli 的一句政治箴言，1512

和乘法相反的操作是除法，它用得较少，但很诡异。它甚至可能会出现数学上的无效操作：除数为 0。

首先通过十进制数的长除来回忆一下操作数的命名和小学时学习的除法算法。类似于前面，为简单起见，我们只使用十进制中的 0 和 1。这个例子是计算 $1\,001\,010_{10}$ 除以 $1000_{10}$：

$$
\begin{array}{rl}
 & 1001_{10}\ \text{商} \\
\text{除数 } 1000_{10} & \overline{)\,1001010_{10}}\ \text{被除数} \\
 & -1000 \\
 & \quad 10 \\
 & \quad 101 \\
 & \quad 1010 \\
 & \quad -1000 \\
 & \quad\ \ 10_{10}\ \text{余数}
\end{array}
$$

除法中的两个源操作数，称为**被除数**（dividend）和**除数**（divisor），结果称为**商**（quotient），还有一个第二结果，称为**余数**（remainder）。这里用一种方式来表达它们之间的关系：

$$\text{被除数} = \text{商} \times \text{除数} + \text{余数}$$

这里余数要小于除数。在某些场合，程序使用除法指令只是为了获得余数，而忽视商。

- 被除数：被除的数。
- 除数：用于对被除数进行除法的数。
- 商：除法的主要结果；乘以除数并加上余数产生被除数的数。
- 余数：除法的第二个结果，加在商和除数的乘积上产生被除数的数。

这个过程中每次都尝试看最大能减掉多少，然后以此产生商。我们小心地选择出只用 0 和 1 的十进制例子，从而很容易判断出需要将多少倍的除数从被除数中减去：要么是 1 倍，要么是 0 倍。二进制数仅包含 0 和 1，所以二进制除法也仅有这两种选择，从而简化了二进制除法。

现在我们假设被除数和除数都为正，因此商和余数也都非负。除法的源操作数和两个结果都是 32 位宽，我们暂且忽略符号位。

### 3.4.1 除法算法及其硬件结构

图 3-8 给出了模拟小学学过的除法算法的硬件结构。在开始时，32 位的商寄存器设为 0。

算法每次的迭代将除数向右移一位。所以我们需要在开始时将除数放置在64位除数寄存器的
189 左半边，然后每次右移一位来和被除数对齐。余数寄存器初始化为被除数。

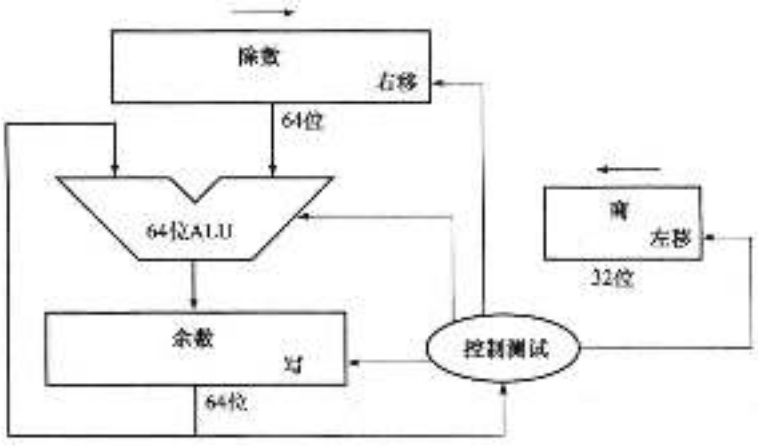



图3-8 第一种除法器硬件结构。除数寄存器、ALU、余数寄存器都是64位宽。只有商寄存器是32位宽。32位的除数开始放置在除数寄存器的左半部分，然后每次迭代右移一位。余数寄存器初始化为被除数。控制逻辑决定何时对除数和商寄存器进行移位以及何时将新值写入余数寄存器

图3-9给出了第一种除法算法的3个步骤。不像人那样聪明，计算机不可能提前知道除数是否小于被除数。所以需要在第1步中减去除数这与在小于时设置指令中的比较操作相同；如果结果为正，则除数小于等于被除数，所以我们取商为1（第2a步）。如果结果为负，则通过将除数加上余数来恢复上一次的值，然后取商为0（第2b步）。除数右移，然后再次迭代。迭代完成后，余数和商存放在以它们命名的寄存器中。

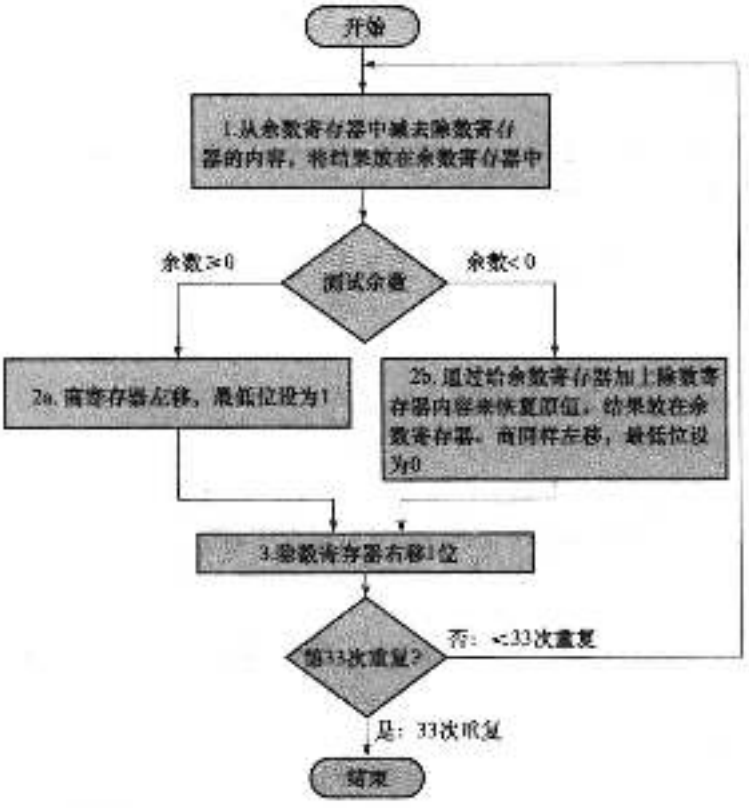



图3-9 第一种除法算法，其硬件结构见图3-8。如果余数为正，则将除数从被除数中减去，然后在第2a步取商为1。如果第1步之后余数为负，则意味着除数不能从被除数中减去，所以在第2b步中商0并将除数加到余数上，即做第1步减法的逆操作。在第3步，进行最后的移位，为下一个迭代的被除数，将除数适当对齐。这些步骤将要重复33次

**例题：除法算法**

为了节省篇幅，我们使用4位的数据。计算$7_{10}$除以$2_{10}$，即0000 0111$_2$除以0010$_2$。

**答案**

图3-10给出了每步中各个寄存器的值，其中，商为$3_{10}$，余数为$1_{10}$。注意到，在第2步中检测余数的正负只需要简单地测试余数寄存器的符号位是0还是1即可。令人惊讶的是，这个算法需要$n+1$步来获得适当的商和余数。

| 迭代次数 | 步骤 | 商 | 除数 | 余数 |
|---|---|---|---|---|
| 0 | 初始值 | 0000 | 0010 0000 | 0000 0111 |
| 1 | 1：余数=余数-除数 | 0000 | 0010 0000 | ①**110 0111** |
| | 2b：余数<0⇒+除数，商左移，上最低位上0 | **0000** | 0010 0000 | **0000 0111** |
| | 3：除数右移 | 0000 | **0001 0000** | 0000 0111 |
| 2 | 1：余数=余数-除数 | 0000 | 0001 0000 | ①**111 0111** |
| | 2b：余数<0⇒+除数，商左移，上最低位上0 | **0000** | 0001 0000 | **0000 0111** |
| | 3：除数右移 | 0000 | **0000 1000** | 0000 0111 |
| 3 | 1：余数=余数-除数 | 0000 | 0000 1000 | ①**111 1111** |
| | 2b：余数<0⇒+除数，商左移，上最低位上0 | **0000** | 0000 1000 | **0000 0111** |
| | 3：除数右移 | 0000 | **0000 0100** | 0000 0111 |
| 4 | 1：余数=余数-除数 | 0000 | 0000 0100 | ①**000 0011** |
| | 2a：余数≥0⇒商左移，上最低位上1 | **0001** | 0000 0100 | 0000 0011 |
| | 3：除数右移 | 0001 | **0000 0010** | 0000 0011 |
| 5 | 1：余数=余数-除数 | 0001 | 0000 0010 | ①**000 0001** |
| | 2a：余数≥0⇒商左移，上最低位上1 | **0011** | 0000 0010 | 0000 0001 |
| | 3：除数右移 | 0011 | **0000 0001** | 0000 0001 |

图3-10 除法的例子，采用图3-9中的算法。图中圈起来的位用于决定下一步的操作

算法和对应的硬件结构分别可以改进得更快、更便宜。加速是通过将源操作数和商移位与减法同时进行。注意到寄存器和加法器有未用的部分，可以通过将加法器和寄存器的位长减半来改进硬件结构，如图3-11所示为改进后的硬件结构。

190 ↓ 191

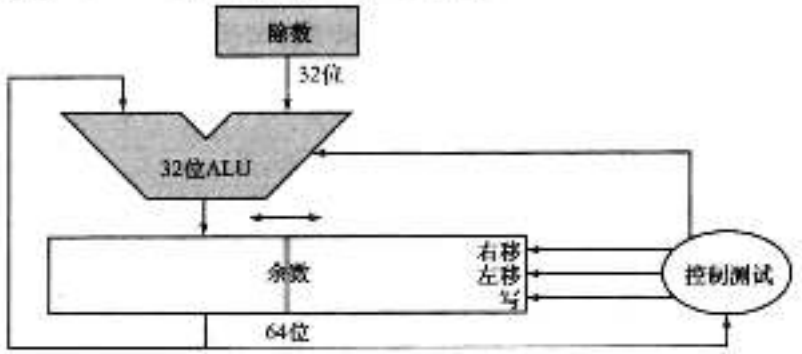



图3-11 除法器的一种改进版本。除数寄存器、ALU、商寄存器都是32位，只有余数寄存器为64位。同图3-8相比，ALU和除数寄存器都是位宽减半，余数进行左移。这个结构将商寄存器和余数寄存器的右半部分进行了拼接（正如图3-5中的那样，余数寄存器应该是65位从而保证加法器的进位不会丢失。）□

### 3.4.2 有符号除法

到目前为止，我们一直忽略有符号数的除法。最简单的办法是记住除数和被除数的符号。
192 如果两者的符号相异，则商为负。

**例1** **精解** 有符号除法一个比较麻烦的地方是必须设置余数的符号。记住，下面的公式必须满足：

$$被除数 = 商 \times 除数 + 余数$$

为了理解如何设置余数的符号，我们来看 $+7_{10}$ 除以 $+2_{10}$ 这个例子的各种情况。第一种情况很简单：

$$(+7) \div (+2):商 = +3,\quad 余数 = +1$$

检查结果：

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

如果我们改变被除数的符号，商就会改变：

$$(-7) \div (+2):商 = -3$$

重写基本公式来计算余数：

$$余数 = (被除数 - 商 \times 除数) = (-7) - [-3 \times (+2)] = (-7) - (-6) = -1$$

从而，

$$(-7) \div (+2):商 = -3, 余数 = -1$$

再次检查结果：

$$-7 = (-3) \times 2 + (-1) = -6 - 1$$

商是 -4 且余数是 -1 同样满足基本公式，但不能取这个结果，其原因是如果这样，商的绝对值将会根据被除数和除数的符号而改变！很明显，如果

$$-(x \div y) \neq (-x) \div y$$

编程将会面临更大的挑战。保持被除数的符号和余数的符号相同，而不管除数和商的符号如何，就可以避免这种异常的情况。

我们采用相同的规则计算其他情况：

$$(+7) \div (-2):商 = -3,\quad 余数 = +1$$

193 $$(-7) \div (-2):商 = +3,\quad 余数 = -1$$

因此，正确的有符号除法算法无论操作数的符号相反时商为负，同时使非零余数的符号和被除数的相同。

### 3.4.3 更快速的除法

与乘法相同，摩尔定律同样适用于除法。我们使用许多加法器来加速乘法，但这一招对除法却不管用。因为除法算法每次迭代前需要知道减法结果的符号，而乘法却可以立刻生成32个部分积。

有一些技术可以每步生成不仅一个商位。如被称为SRT的除法算法，通过查找表的方法来尝试猜测每步几个商位，其中查找表基于被除数和余数的高位部分来进行。它依赖后面的步骤来修正错误的猜测。如今典型值是4位。算法的关键是猜测要试的值。对于二进制算法，只有一种选择。可用余数的6位和除数的4位来索引查找表，从而决定每步的猜测。

这个快速算法的正确性取决于查找表中的值是否合适。在3.9节给出了如果查找表不正确将会出现的情况。

### 3.4.4 MIPS 中的除法

你可能已经注意到图 3-5 和图 3-11 中相同的顺序执行硬件结构既可以做乘法，又可以做除法。唯一需要的是一个 64 位的可左右移位的寄存器和一个能做加减法的 32 位宽的 ALU。因此，MIPS 用 32 位的 Hi 和 32 位的 Lo 寄存器来处理乘法和除法。

我们从上面的算法中可能已经猜出，在除法指令执行完后，Hi 存放着余数，Lo 存放着商。

为了处理有符号整数和无符号整数，MIPS 采用两条指令：除（div）和无符号除（divu）。MIPS 汇编器允许除指令使用三个寄存器，且采用 mflo 和 mfhi 指令将运算结果放入指定的通用寄存器。

### 3.4.5 小结

乘法和除法共用硬件的方案允许 MIPS 提供一对单独的 32 位寄存器来支持乘法和除法运算，可以通过预测多位商的方法来加速除法运算，在预测错误时及时进行恢复。图 3-12 汇总了前面两节中 MIPS 体系结构的优化处理。

194

MIPS 汇编语言

| 类别 | 指令 | 例题 | 含义 | 备注 |
|---|---|---|---|---|
| 算术运算 | 加 | add $s1,$s2,$s3 | $s1 = $s2 + $s3 | 三个操作数;检测溢出 |
| | 减 | sub $s1,$s2,$s3 | $s1 = $s2 - $s3 | 三个操作数;检测溢出 |
| | 加立即数 | addi $s1,$s2,100 | $s1 = $s2 +100 | 加常数;检测溢出 |
| | 无符号加 | addu $s1,$s2,$s3 | $s1 = $s2 + $s3 | 三个操作数;不检测溢出 |
| | 无符号减 | subu $s1,$s2,$s3 | $s1 = $s2 - $s3 | 三个操作数;不检测溢出 |
| | 无符号加立即数 | addiu $s1,$s2,100 | $s1 = $s2 +100 | 加常数;不检测溢出 |
| | 从协处理器寄存器中获得 | mfc0 $s1,$epc | $s1 = $epc | 复制异常 pc 到专用寄存器 |
| | 乘 | mult $s2,$s3 | Hi,Lo = $s2 × $s3 | 64 位有符号积存在 Hi,Lo 中 |
| | 无符号乘 | multu $s2,$s3 | Hi,Lo = $s2 × $s3 | 64 位无符号积存在 Hi,Lo 中 |
| | 除 | div $s2,$s3 | Lo = $s2/$s3, Hi = $s2 mod $s3 | Lo = 商,Hi = 余数 |
| | 无符号除 | divu $s2,$s3 | Lo = $s2/$s3, Hi = $s2 mod $s3 | 无符号商和余数 |
| | 从 Hi 中获得 | mfhi $s1 | $s1 = Hi | 用来获得 Hi 的复本 |
| | 从 Lo 中获得 | mflo $s1 | $s1 = Lo | 用来获得 Lo 的复本 |
| 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存到内存中 |
| | 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 | 将半个字从寄存器中存到内存中 |
| | 取无符号字节 | lbu $s1,20($s2) | $s1 = Memory[$s2 +20] | 将一个字节从内存中取到寄存器中 |
| | 存字节 | sb $s1,20($s2) | Memory[$s2 +20] = $s1 | 将一个字节从寄存器中存到内存中 |
| | 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 或 1 | 存字作为原子交换的后半部 |
| | 立即数读入高 16 位 | lui $s1,100 | $s1 = 100 × $2^{16}$ | 取立即数并放在高 16 位 |

图 3-12 MIPS 核心体系结构。为了节省篇幅，没有给出 MIPS 体系结构的存储器和寄存器，但增加了 Hi 和 Lo 寄存器来支持乘法和除法。在 MIPS 参考数据卡中列出了 MIPS 机器语言

195

| 类型 | 指令 | 举例 | 含义 | 备注 |
|---|---|---|---|---|
| 逻辑运算 | 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 |
| | 与立即数 | ANDI $s1,$s2,100 | $s1 = $s2 & 100 | 和常数按位与 |
| | 或立即数 | ORI $s1,$s2,100 | $s1 = $s2 \| 100 | 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 |
| | 逻辑右移 | srl $s1,$s2,10 | $s1 = $s2 >> 10 | 根据常数右移相应位 |
| 条件跳转 | 相等时转移 | beq $s1,$s2,25 | if($s1 == $s2)跳至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)跳至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) $s1 = 1;否则$s1 = 0 | 比较是否小于;补码形式 |
| | 小于立即数时置位 | slti $s1,$s2,100 | if($s2 < 100) $s1 = 1;否则$s1 = 0 | 比较是否小于常数;补码形式 |
| | 无符号数比较小于时置位 | sltu $s1,$s2,$s3 | if($s2 < $s3) $s1 = 1;否则$s1 = 0 | 比较是否小于;自然数 |
| | 无符号数比较小于立即数时置位 | sltiu $s1,$s2,100 | if($s2 < 100) $s1 = 1;否则$s1 = 0 | 比较是否小于常数;自然数 |
| 无条件跳转 | 跳转 | j 2500 | 跳至10000 | 跳转到目标地址 |
| | 跳转至寄存器所指的位置 | jr $ra | 跳至$ra | 用于switch语句,以及过程调用返回 |
| | 跳转并链接 | jal 2500 | $ra = PC+4;跳至10000 | 用于过程调用 |

图3-12 （续）

**硬件/软件接口** MIPS处理器除法指令忽略溢出，所以需要软件来检测商是否溢出。除了溢出，除法还可能产生不适当的计算：除数为0。一些计算机会分辨这两种异常事件。和溢出一样，MIPS软件必须通过检查除数来确定是否会发生此类情况。

**精解** 一种更快的算法是在余数为负时，不需要立即将除数加回去。它只是在下一步简单地将被除数加到移位后的余数上，因为 $(r+d)\times 2 - d = r\times 2 + d\times 2 - d = r\times 2 + d$。这种不恢复（nonrestoring）除算法每步需要一个时钟周期，将会在练习题中给出更多的分析；而前面介绍的算法称为恢复（restoring）除法。第三种算法称为不执行（nonperforming）除算法，这种算法在余数为负时，不保存减法的结果。它平均减少了三分之一的算术操作。

## 3.5 浮点运算

*如果方向错了，再快也白搭。*

——美国谚语

除了有符号和无符号整数，编程语言也支持带小数的数字，即数学上的实数。如：

$3.14159265\cdots_{10}$（pi）

$2.71828\cdots_{10}$（e）

$0.000000001_{10}$即$1.0_{10}\times 10^{-9}$（纳秒级）

$3155760000_{10}$即$3.15576_{10}\times 10^{9}$（一百年的秒数）

请注意，在最后的例子中，那个数并不是小数，而是比32位的有符号整数还要大的数。

这种表达上述例子中后两个数的记数法称为**科学记数法**（scientific notation）。一个采用科学记数法表示的数，若没有前导零且小数点左边只有一位整数，则可称为**规格化**（normalized）数。例如，$1.0_{10} \times 10^{-9}$就是规格化的科学记数，但$0.1_{10} \times 10^{-8}$和$10.0_{10} \times 10^{-10}$就不是。

- 科学记数法：十进制小数点左边只有一位整数的记数法。
- 规格化数：没有前导零的浮点记数法。

正如可以用科学记数法来表示十进制数那样，我们也可以用科学记数法来表示二进制数，如：

$$1.0_2 \times 2^{-1}$$

为了使二进制数规格化，需要定义一个基数，这个基数可用来移位使小数点左边只保留一位非零数。只有基数为2才满足要求。因为基数不是10，所以我们称这时的小数点为二进制小数点（binary point）。

这类计算机算术支持浮点数（floating point）的计算，称为浮点数是因为其表示的二进制小数点是不固定的，与整数相似。C语言用float来表示这类数。正如科学记数那样，数被表示为二进制小数点左边只有一位非零数的形式。在二进制中，其格式为：$1.xxxxxxxxx_2 \times 2^{yyyy}$。（尽管计算机对指数也同其他数一样表示为以2为基的形式，但这里为了简化记数，我们用十进制来表示指数。）

- 浮点数：二进制小数点不固定的表达数的记数法。

对实数采用规格化形式的标准科学记数法有三个优点：简化了浮点数的数据交换；简化了浮点算术算法；提高了用一个字存储的数的精度，因为无用的前导零可能占用的位被二进制小数点右边的有效位替代了。

### 3.5.1 浮点表示

浮点表示的设计者必须在**尾数**（fraction）位宽和**指数**（exponent）位宽之间找出折中的办法，因为字的大小是固定的，有一部分增加一位，则另一部分就要减少一位。折中是在精度和表示范围间进行权衡：增加小数部分会增加表示精度，而增加指数部分会增加数的表示范围。正如我们在第2章中所提到的设计方针讲的那样，好的设计需要好的折中。

- 尾数：位于浮点数的尾数字段，其值在0和1之间。
- 指数：位于浮点数的指数字段，表示小数点的位置。

浮点数通常是多个字的宽度。MIPS中的浮点数表示如下：$s$为浮点数的符号（1表示负数），指数域为8位宽（包括指数的符号位），尾数域为23位宽。这种表示称为符号和数值（sign and magnitude），因为符号和数值的位置是相互分离的。

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $s$ | 指数 | | | | | | | | 尾数 | | | | | | | | | | | | | | | | | | | | | | |
| 1位 | 8位 | | | | | | | | 23位 | | | | | | | | | | | | | | | | | | | | | | |

一般浮点数表示为这样的形式：

$$(-1)^S \times F \times 2^E$$

$F$为小数域的值，$E$为指数域的值。这些域之间具体的关系后面会详细讲解。（我们将会简单地看到MIPS所做的稍有技巧性的改变。）

浮点数表示法使MIPS计算机有很大的数值表示范围，可以小到$2.0_{10} \times 10^{-38}$，大到$2.0_{10} \times$

$10^{38}$。但它和无穷还是不同的，所以依然有可能会因数太大而不能表示。因此，正如在整数运算中那样，溢出中断在浮点运算中也会发生。注意，这里的溢出（上溢）（overflow）表示指数太大而不能在指数域表示。

⇒ 溢出（上溢）：正的指数太大而导致指数域放不下的情况。

浮点也会出现一种新的异常事件。正如程序员想要知道何时他们计算的数太大而不能表示那样，他们同样也想知道一个非零的小数是否会太小而不能表示；任何一种事件都会引起程序给出错误答案。为了和上溢区分开来，将其称为**下溢**（underflow）。下溢发生的条件是负的指数太大而不能在指数域中表示出来。

一种减少上溢和下溢的方法是采用更大指数的格式。在C语言中称为double，基于double的操作称为**双精度**（double precision）浮点算术；**单精度**（single precision）浮点就是前面的格式。

⇒ 下溢：负的指数太大而导致指数域放不下的情况。
⇒ 双精度：浮点数由两个32位的字表示。
⇒ 单精度：浮点数由一个32位的字表示。

双精度浮点数占用了两个MIPS字，如下所示。其中，$s$表示符号，指数域为11位，尾数域为52位。

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $s$ | 指数 | | | | | | | | | | | | 尾数 | | | | | | | | | | | | | | | | | | |
| 1位 | 11位 | | | | | | | | | | | | 20位 | | | | | | | | | | | | | | | | | | |
| 尾数（续） | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 32位 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

MIPS双精度的表示范围从$2.0_{10} \times 10^{-308}$到$2.0_{10} \times 10^{308}$。尽管双精度增加了指数范围，它主要的优势还是通过提供更多的有效位数来实现更大的表示精度。

这些格式已经超出了MIPS体系结构。实际上它们是IEEE 754浮点标准的一部分，从1980年以来的每台计算机都遵循该标准。该标准既简化了浮点程序的接口，又提高了计算机算术的质量。

为了将更多的数据位打包到有效位数（significand）部分，IEEE 754甚至隐藏了规格化二进制数的前导位1。因此，在单精度下，数有24位宽（隐含的1和23位尾数），在双精度下为53位宽（1+52）。为了精确，我们用术语有效位数来表示24位或者53位的数，就是隐含1加
198 尾数。[⊖]因为0没有前导位1，它的指数保留为0，所以硬件就不会将前导位1加到尾数上面。

因此00…00$_2$代表0；其他数的表示依然采用前面的形式，就是加上了隐含1：

$$(-1)^S \times (1+F) \times 2^E$$

其中，$F$表示的是0和1之间的数，$E$表示的是指数域中的值。如果我们从左到右标记小数为$s1$，$s2$，$s3$，…，则数的值为

$$(-1)^S \times [1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \cdots] \times 2^E$$

图3-13给出了IEEE 754浮点数的编码。IEEE 754标准的其他特点是用特殊的符号来表示异常事件。例如，软件可以将结果设置成某种格式来表示$+\infty$或者$-\infty$，以替代除0中断；最大的指数保留下来标识那些特殊符号。当程序员打印结果时，程序会打印出一个无穷符号。（对于有数学训练的人，无穷的目的是形成实数的拓扑闭集。）

⊖ 由于“有效位”和“尾数”相比，“有效位”多了一位“1”，而在浮点运算中，常用“尾数”的术语，因此在后面多处“尾数”即表示“有效位”。——译者注

| 单精度 | | 双精度 | | 表示对象 |
|---|---|---|---|---|
| 指数 | 尾数 | 指数 | 尾数 | |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 非0 | 0 | 非0 | ±非规格化数 |
| 1-254 | 任何值 | 1-2046 | 任何值 | ±浮点数 |
| 255 | 0 | 2047 | 0 | ±无穷 |
| 255 | 非0 | 2047 | 非0 | NaN（非数，即不是数） |

图 3-13 IEEE 754 浮点数的编码。用一个单独的符号位来决定正负。在 3.5 节的精解中描述了非规格化数。这个信息也可以在 MIPS 参考数据卡的第 4 列中找到

IEEE 754 甚至给出了一种表示无效操作结果（如 0/0 或者无穷减无穷）的符号——NaN（Not a Number），即非数的意思。设立 NaN 的目的是让程序员推迟程序中的一些测试和决定，等到方便的时候再进行。

IEEE 754 的设计者还希望浮点表示能够比较容易处理整数比较，特别是排序的时候。这就是为什么符号放在最高位的原因，这样就可以快速地测试出小于、大于、等于 0 的情况。（比起简单的整数分类，它稍显复杂，因为这种记数法本质上是符号和数值的形式，而不是补码形式。）

将指数放在有效数前也能简化用整数比较指令来处理的浮点数分类，因为在有着相同符号的情况下，指数大的数其值就大。

199

负的指数对简化分类形成一个挑战。如果我们用补码或者其他的记数法，可能会使负指数的高位为 1，从而使一个负指数显得是一个大数了。例如，$1.0_2 \times 2^{-1}$ 表示如下：

| 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 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | . | . | . |

（需要注意的是，尾数中隐含前导 1。）而数 $1.0_2 \times 2^{+1}$ 看起来似乎是一个较小的二进制数。

| 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 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | . | . | . |

因此希望记数法能将最小的负指数表示为 $00\cdots00_2$，而最大的正指数表示为 $11\cdots11_2$。这种记数法称为带偏阶的记数法（biased notation）。需要从带偏阶的指数中减去偏阶，才能获得真实的值。

IEEE 754 规定单精度的偏阶为 127，所以指数为 -1，则会表示为 $-1+127_{10}$，即 $126_{10} = 0111\ 1110_2$，而 +1 表示为 1 +127，即 $128_{10} = 1000\ 0000_2$。双精度的指数偏阶为 1023。给指数带偏阶后，浮点数表示为

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

从而，单精度数的表示范围从

$$\pm 1.000\ 000\ 000\ 000\ 000\ 000\ 000\ 00_2 \times 2^{-126}$$

到

$$\pm 1.111\ 111\ 111\ 111\ 111\ 111\ 111\ 11_2 \times 2^{+127}$$

让我们演示一下浮点表示。

200

**例题 · 浮点表示**

演示用 IEEE 754 的单精度和双精度格式来表示 $-0.75_{10}$。

**答案**

$-0.75_{10}$ 也可表示为

$$-3/4_{10} \quad 或 \quad -3/2^2_{10}$$

它的二进制小数形式为

$$-11_2/2^2_{10} \quad 或 \quad -0.11_2$$

用科学记数表示的形式为

$$-0.11_2 \times 2^0$$

采用规格化的科学记数，为

$$-1.1_2 \times 2^{-1}$$

单精度的通用表达式为

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

将 $-1.1_2 \times 2^{-1}$ 的指数减去 127，得到

$$(-1)^1 \times (1 + 0.1000\ 0000\ 0000\ 0000\ 0000\ 000_2) \times 2^{(126-127)}$$

所以 $-0.75_{10}$ 的单精度二进制格式为

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

1位　　8位　　23位

双精度表示为

$$(-1)^1 \times (1 + 0.1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2) \times 2^{(1023-1023)}$$

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

1位　　11位　　20位

| 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 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

32位

201

让我们再看反方向的一个例子。 □

**例题·二进制转十进制浮点**

十进制数如何用单精度浮点表示？

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | . | . |

**答案**

符号位为1，指数域的值为129，尾数域的值为 $1 \times 2^{-2} = 1/4$，即0.25。使用基本公式，

$$\begin{aligned}(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(128-127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 = -5.0\end{aligned}$$

□

在下面的部分，我们将给出浮点加法和乘法的算法。其核心部分是对尾数进行相应的移数操作，但也需要额外的工作去处理指数部分并对结果进行规格化。我们先给出直观上的十进制算法，然后在图中给出有更多细节的二进制版本。

**精解** 遵循IEEE准则，在标准发布20年后，IEEE 754委员会依据需求对标准进行了更新。更新后的IEEE 754－2008标准中包含了IEEE 754－1985标准中几乎全部内容，并且增加了16位（半精度）和128位（四精度）。到目前为止还没有硬件支持四精度，但日后必会出现。更新后的标准也支持十进制浮点运算，这在IBM大型机中已经实现。

**精解** 为了在保持尾数位宽不变的情况下增大表示范围，一些早于 IEEE 754 标准的计算机采用了大于 2 的基数。例如，IBM 360 和 370 大型计算机以 16 为基数。因此每当 IBM 机的指数改变 1，尾数就将移 4 位，所以基为 16 的规格化数的前导零可能会多达 3 个！也就意味着有 3 个有效位要从尾数中去掉，从而在浮点算术精度上产生较大的问题。最近 IBM 大型机不但支持十六进制模式也开始支持 IEEE 754 标准。 □ 202

## 3.5.2 浮点加法

为了说明浮点加法中的问题，我们将用科学记数法表示的两个数相加：$9.999_{10} \times 10^{1} + 1.610_{10} \times 10^{-1}$。假设我们只能存储 4 个十进制有效数和两个十进制指数。

步骤 1：为了能让两数相加，我们必须将有较小指数的数向有较大指数的数对齐。

因此，需要将指数较小 $1.610_{10} \times 10^{-1}$ 的向指数大的数对齐。由于一个非规格化的浮点数可以有多种科学计数法的表示形式，可以利用这一特性完成指数的对齐。

$$1.610_{10} \times 10^{-1} = 0.161\,0_{10} \times 10^{0} = 0.016\,10_{10} \times 10^{1}$$

最右边的表示形式是我们所需要的，因为它和较大的数 $9.999_{10} \times 10^{1}$ 的指数相同。因此第一步要向右移动较小数的有效数，使其指数和较大数的指数相同。由于我们只能表示 4 位十进制数，所以，移位后得到的数为

$$0.016_{10} \times 10^{1}$$

步骤 2：将有效数相加：

$$\begin{array}{r} 9.999_{10} \\ +\ 0.016_{10} \\ \hline 10.015_{10} \end{array}$$

和为 $10.015_{10} \times 10^{1}$。

步骤 3：因为和不是规格化科学记数的形式，所以我们需要调整：

$$10.015_{10} \times 10^{1} = 1.001\,5_{10} \times 10^{2}$$

因此，在加法后，我们可能需要对和移位，将其变为规格化形式，同时相应地调整指数。在这个例子中是右移，但是如果一个数为正，一个数为负，则和可能会有许多前导 0，从而需要左移。无论指数是增加还是减少，我们都需要检查上溢或者下溢——我们必须保证指数能够被固定位宽的指数域所表示。

步骤 4：因为我们假设有效数只有 4 位十进制数那么长（不包括符号位），所以我们需要进行舍入。如果右边多余的数在 0 和 4 之间就舍掉，如果右边多余的数在 5 和 9 之间，则加 1。数

$$1.001\,5_{10} \times 10^{2}$$

舍入为有 4 个十进制数有效位的数

$$1.002_{10} \times 10^{2}$$ 203

这是因为第四位右边的数在 5 和 9 之间。注意，如果我们不幸将 1 加到了一串 9 上，则和不能再规格化，我们需要返回到步骤 3。

图 3-14 按照这个十进制例子给出了二进制浮点加算法。步骤 1 和步骤 2 与上面例子讨论的类似：调整有较小指数的数，使其指数与有较大指数的数对齐，然后将两个数的有效数相加。步骤 3 规格化结果，并强制检查上溢和下溢。步骤 3 中上溢和下溢的检查依赖于源操作数的精度。回忆一下，指数全 0 保留下来用来表示 0；指数全 1 保留下来标记指定的值和超出正常浮点数范围的情况（见 3.5 节的精解）。在下面的例子中需要注意，对于单精度，最大的指数为 127，最小的指数为 −126。

**例题·二进制浮点加法**

按照图3-14中的算法，尝试将$0.5_{10}$和$-0.437\,5_{10}$用二进制相加。

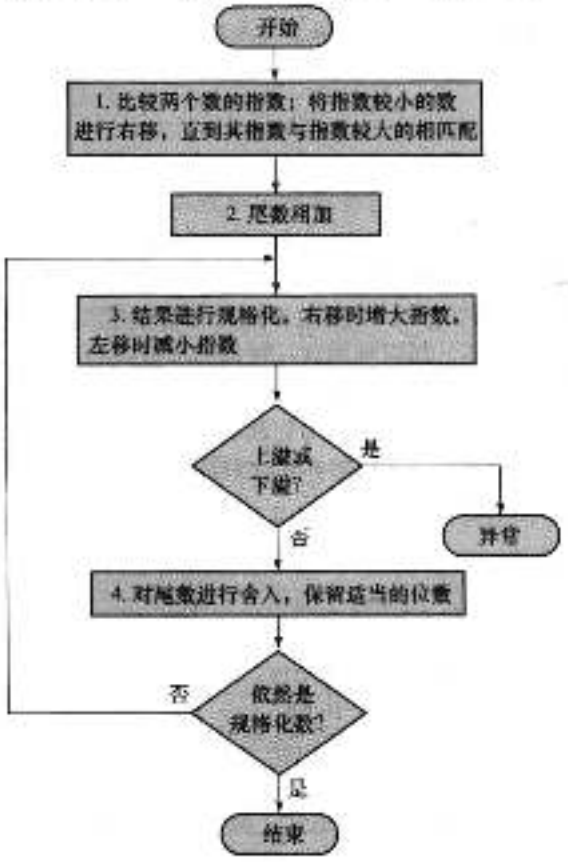



图3-14 浮点加。正常的路径是执行一次步骤3和步骤4，但如果舍入使和变为未规格化，则需要重复步骤3

**答案**

首先，我们看一下这两个数用规格化科学记数法的二进制表示。这里假设我们使用4位精度：

$$0.5_{10} = 1/2_{10} = 1/2^1_{10}$$
$$= 0.1_2 = 0.1_2 \times 2^0 = 1.000_2 \times 2^{-1}$$
$$-0.437\,5_{10} = -7/16_{10} = -7/2^4_{10}$$
$$= -0.011\,1_2 = -0.011\,1_2 \times 2^0 = -1.110_2 \times 2^{-2}$$

现在，我们按照如下的算法执行：

步骤1：将有最小指数的数（$-1.11_2 \times 2^{-1}$）的有效数进行右移，直到其指数和较大数相匹配：

$$-1.110_2 \times 2^{-2} = -0.111_2 \times 2^{-1}$$

204
↓
205

步骤2：将有效数相加：

$$1.000_2 \times 2^{-1} + (-0.111_2 \times 2^{-1}) = 0.001_2 \times 2^{-1}$$

步骤3：将和规格化，并检查上溢和下溢：

$$0.001_2 \times 2^{-1} = 0.010_2 \times 2^{-2} = 0.100_2 \times 2^{-3} = 1.000_2 \times 2^{-4}$$

因为$127 \geqslant +4 \geqslant -126$，所以没有上溢和下溢。（带偏阶的指数为$-4+127$，即123，其在

最小的指数 1 和最大的未保留的带偏阶指数 254 之间。）

步骤 4：舍入和：

$$1.000_2 \times 2^{-4}$$

这个和已经是精确地用 4 位来表示了，所以不需要再做舍入。

这个和是

$$1.000_2 \times 2^{-4} = 0.000\,100\,0_2 = 0.000\,1_2$$
$$= 1/2^4_{10} \qquad = 1/16_{10} \qquad = 0.062\,5_{10}$$

这就是 $0.5_{10}$ 和 $-0.437\,5_{10}$ 的和。 □

许多计算机会使用硬件来尽可能快地运行浮点操作。图 3-15 给出了浮点加的基本结构示意图。

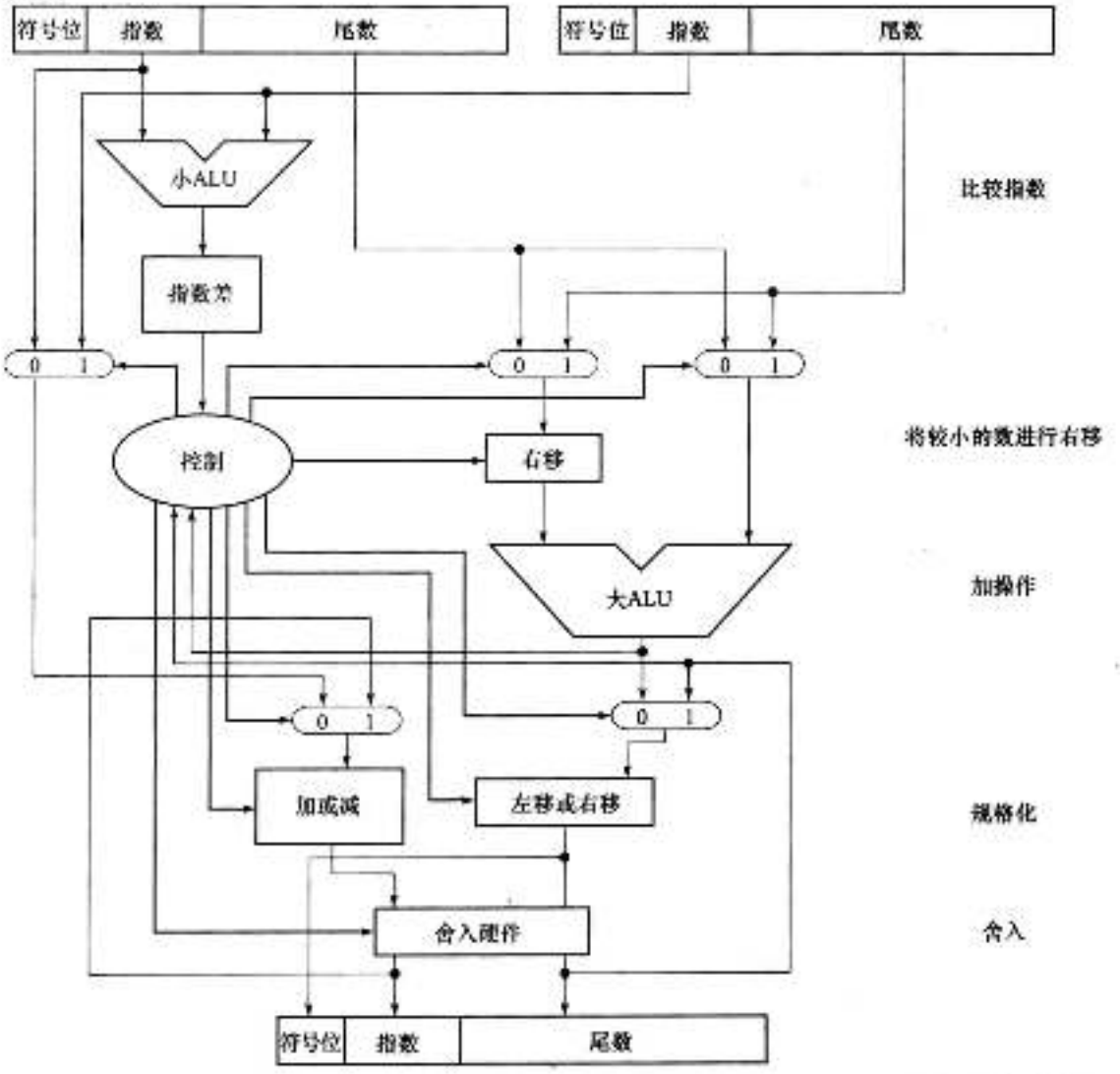



图 3-15 用于浮点加的算术单元的结构框图。图 3-14 的每步从顶向下对应到每个方框。首先，使用一个小的 ALU 将两个指数相减来决定哪个指数大及大多少。指数差将控制三个多路复用器：从左到右，选择出较大的指数、较小数的有效数和较大数的有效数。较小数的有效数通过右移后，和较大数的有效数用一个大的 ALU 相加。规格化步骤将和左移或者右移，同时增加或者减少指数。舍入产生最后的结果，这样也有可能需要再次规格化，然后产生最后的结果

### 3.5.3 浮点乘法

首先，我们手算一个十进制乘法的例子，其中数用科学记数法表示：$1.110_{10} \times 10^{10} \times 9.200_{10} \times 10^{-5}$。假设我们只可以存储4位尾数和2位指数。

步骤1：不像加法，我们只是简单地将源操作数的指数相加来作为积的指数：

$$\text{新的指数} = 10 + (-5) = 5$$

现在我们处理带有偏阶的指数并要确定获得相同的结果：$10 + 127 = 137$，而 $-5 + 127 = 122$，所以

$$\text{新的指数} = 137 + 122 = 259$$

这个结果对于8位的指数域来说太大，所以肯定有什么地方出错了！问题出在偏阶上，当
206 我们将指数相加时，也对偏阶实行了相加：

$$\text{新的指数} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

因此，当将带偏阶的数相加时，为了得到正确的带偏阶的和，我们需要将一个偏阶从和中减去：

$$\text{新的指数} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

5就是我们刚开始计算的实际指数。

步骤2：下面计算有效数的乘法：

$$\begin{array}{r} 1.110_{10} \\ \times\ 9.200_{10} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{10} \end{array}$$

每个源操作数十进制小数点右边都有三位，所以积的尾数的十进制小数点在从右边数第6位处：

$$10.212\,000_{10}$$

假设我们只可以保留十进制小数点右边三位数，则积为 $10.212_{10} \times 10^5$。

步骤3：这个积是未规格化的，所以我们需要规格化它：

$$10.212_{10} \times 10^5 = 1.021\,2_{10} \times 10^6$$

因此，在乘法后，积需要右移一位来变成规格化形式，同时指数加1。此刻，我们要检查上溢和下溢。当两个源操作数都很小时——两者都有非常大的负指数时，就有可能发生下溢。

步骤4：因为之前我们假设有效数只有4位宽（不包括符号），所以我们必须对结果进行舍入。将

$$1.021\,2_{10} \times 10^6$$

舍入为只有4位的有效数

$$1.021_{10} \times 10^6$$

步骤5：积的符号取决于原始源操作数的符号。当它们相同时，符号为正；否则，符号为负。因此，积为

207
~
208

$$+1.021_{10} \times 10^6$$

在加法算法中，和的符号由有效数相加来决定，但在乘法中，积的符号由源操作数来决定。

再一次，如图3-16所示，二进制浮点乘的步骤和我们刚做完的步骤类似。首先，我们将

带偏阶的指数相加，并减去一个偏阶，获得积的指数。接着是有效数的乘法，紧跟一个可选的规格化步骤。指数的大小用来检查上溢和下溢，然后对积进行舍入。当舍入引起进一步的规格化时，我们需要再次检查指数的大小。最后，如果源操作数的符号相异，就将符号位设为1（积为负）；如果相同，设为0（积为正）。

**例题·二进制浮点乘法**

按照图3-16中的步骤，试计算$0.5_{10}$和$-0.437\,5_{10}$的乘积。

**答案**

在二进制下，也就是将$1.000_2\times2^{-1}$和$-1.110_2\times2^{-2}$相乘。

步骤1：将不带偏阶的指数相加：

$$-1+(-2)=-3$$

或者，使用带偏阶的表达：

$$\begin{aligned}&(-1+127)+(-2+127)-127\\&=(-1-2)+(127+127-127)\\&=-3+127=124\end{aligned}$$

步骤2：将有效数相乘：

$$\begin{array}{r}1.000_2\\\times\quad 1.110_2\\\hline 0000\\1000\\1000\\1000\\\hline 1110000_2\end{array}$$

积是$1.110\,000_2\times2^{-3}$，但是我们需要保存4位，所以为$1.110_2\times2^{-3}$。

步骤3：现在我们检查积以确保其是规格化的，然后检查指数以确定上溢和下溢是否发生。这个积已经是规格化的，并且，因为$127\geqslant-3\geqslant-126$，所以没有上溢和下溢。（使用带偏阶的表达，$254\geqslant124\geqslant1$，所以指数域可以表达。）

步骤4：对积舍入没有使其发生变化：

$$1.110_2\times2^{-3}$$

步骤5：因为初始的源操作数的符号相异，所以积的符号为负。因此，积为

$$-1.110_2\times2^{-3}$$

为了检查结果，将其转化为十进制：

$$-1.110_2\times2^{-3}=-0.001\,110_2=-0.001\,11_2=-7/2^5_{10}=-7/32_{10}=-0.218\,75_{10}$$

而$0.5_{10}$和$-0.437\,5_{10}$的积确实是$-0.218\,75_{10}$。□

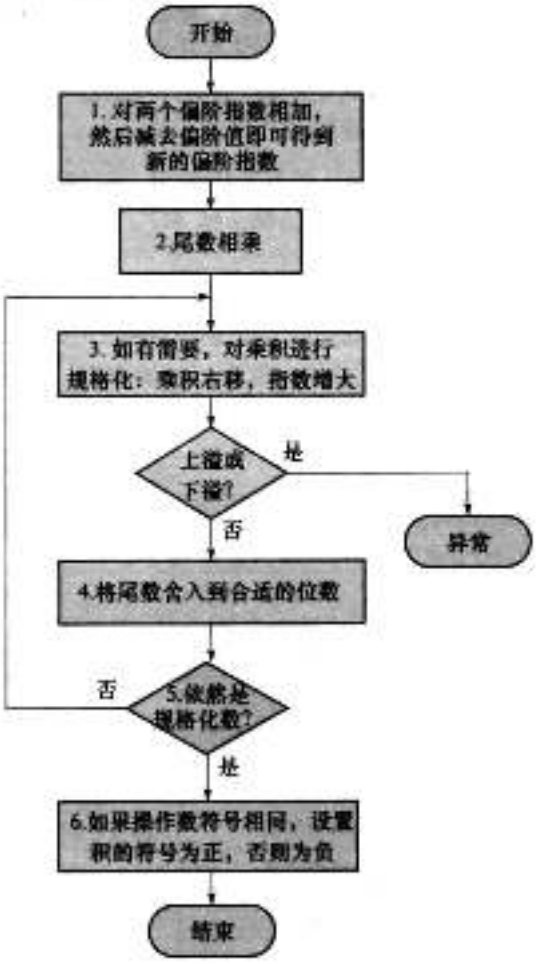



图3-16 浮点乘法。正常的路径是执行一次步骤3和步骤4，但如果舍入使积变为非规格化数，则需要重复步骤3

209 ≈ 210

## 3.5.4 MIPS中的浮点指令

MIPS有如下指令来支持IEEE 754的单精度和双精度格式：

- 浮点单精度加（add.s）和双精度加（add.d）。
- 浮点单精度减（sub.s）和双精度减（sub.d）。
- 浮点单精度乘（mul.s）和双精度乘（mul.d）。
- 浮点单精度除（div.s）和双精度除（div.d）。
- 浮点单精度比较（c.x.s）和双精度比较（c.x.d），其中，x可能是等于（eq）、不等于（neq）、小于（lt）、小于等于（le）、大于（gt）或大于等于（ge）。
- 浮点比较为真跳转（bc1t）和浮点比较为假跳转（bc1f）。

根据比较条件，浮点比较将比较结果设为真或者假，然后浮点跳转将比较结果作为条件决定是否跳转。

MIPS设计增加了单独的浮点寄存器——称为$f0，$f1，$f2，……——用于单精度及双精度。因此，也有单独的针对浮点寄存器的存和取指令：lwc1和swc1。浮点数据传送的基寄存器仍然采用整数寄存器。从内存载入两个单精度数，将其相加，然后再将和存入内存的MIPS代码可能是这样的：

```
lwc1    $f4,x($sp)  # load 32-bit F.P. number into F4
lwc1    $f6,y($sp)  # load 32-bit F.P. number into F5
add.s   $f2,$f4,$f6 # F2 = F4 + F5 single precision
swc1    $f2,z($sp)  # Store 32-bit F.P. number from F2
```

双精度寄存器是一组单精度寄存器的偶数－奇数对，并使用偶数寄存器编号作为其名称。因此，一对单精度寄存器$f2和$f3形成一个双精度寄存器，称为$f2。

图3-17汇总了本章介绍过的MIPS体系结构中的浮点部分，其中为支持浮点而增加的部分用加粗标记。类似于第2章中的图2-19，图3-18给出了这些指令的编码。

211

**MIPS浮点操作数**

| 名称 | 例题 | 备注 |
|---|---|---|
| 32个浮点寄存器 | $f0，$f1，$f2，…，$f31 | 成对地使用MIPS浮点寄存器来保存双精度数 |
| $2^{32}$个存储字 | Memory[0]，<br>Memory[4]，…，<br>Memory[4294967292] | 仅仅被数据传输指令访问。MIPS使用字节地址，所以连续的字地址相差4。存储器用来保存像数组这样的数据结构和在过程调用中溢出的寄存器 |

**MIPS浮点汇编语言**

| 分类 | 指令 | 实例 | 含义 | 备注 |
|---|---|---|---|---|
| 算术 | 浮点单精度加 | add.s $f2,$f4,$f6 | $f2 = $f4 + $f6 | 浮点加（单精度） |
| | 浮点单精度减 | sub.s $f2,$f4,$f6 | $f2 = $f4 - $f6 | 浮点减（单精度） |
| | 浮点单精度乘 | mul.s $f2,$f4,$f6 | $f2 = $f4 × $f6 | 浮点乘（单精度） |
| | 浮点单精度除 | div.s $f2,$f4,$f6 | $f2 = $f4/$f6 | 浮点除（单精度） |
| | 浮点双精度加 | add.d $f2,$f4,$f6 | $f2 = $f4 + $f6 | 浮点加（双精度） |
| | 浮点双精度减 | sub.d $f2,$f4,$f6 | $f2 = $f4 − $f6 | 浮点减（双精度） |
| | 浮点双精度乘 | mul.d $f2,$f4,$f6 | $f2 = $f4 × $f6 | 浮点乘（双精度） |
| | 浮点双精度除 | div.d $f2,$f4,$f6 | $f2 = $f4/$f6 | 浮点除（双精度） |
| 数据传输 | 从内存中取字到浮点寄存器 | lwc1 $f1,100($s2) | $f1 = 存储[$s2 + 100] | 32位的数据传给浮点寄存器 |
| | 从浮点寄存器中存字到内存 | swc1 $f1,100($s2) | 存储[$s2 + 100] = $f1 | 32位的数据传给存储器 |

图3-17 以前介绍过的MIPS浮点体系结构。附录A.10有更详细的介绍。这个信息也可以在MIPS参考数据卡的第2列旦找到

| 分类 | 指令 | 例子 | 含义 | 备注 |
|---|---|---|---|---|
| 条件跳转 | 浮点标志真则跳转 | bc1t 25 | 如果（cond = = 1）<br>跳至 PC + 4 + 100 | 如果浮点标志为真则执行 PC 相关联的跳转 |
| | 浮点标志假则跳转 | bc1t 25 | 如果（cond = = 0）<br>跳至 PC + 4 + 100 | 如果浮点标志为假则执行 PC 相关联的跳转 |
| | 浮点单精度比较（eq, ne, lt, le, gt, ge） | c.lt.s $f2, $f4 | 如果（$f2 < $f4）<br>则 cond = 1；否则 cond = 0 | 浮点单精度比较，如果小于则置 cond |
| | 浮点双精度比较（eq, ne, lt, le, gt, ge） | c.lt.d $f2, $f4 | 如果（$f2 < $f4）<br>则 cond = 1；否则 cond = 0 | 浮点双精度比较，如果小于则置 cond |

MIPS 浮点机器语言

| 名称 | 格式 | 例子 | | | | | | 备注 |
|---|---|---|---|---|---|---|---|---|
| add.s | R | 17 | 16 | 6 | 4 | 2 | 0 | add.s $f2, $f4, $f6 |
| sub.s | R | 17 | 16 | 6 | 4 | 2 | 1 | sub.s $f2, $f4, $f6 |
| mul.s | R | 17 | 16 | 6 | 4 | 2 | 2 | mul.s $f2, $f4, $f6 |
| div.s | R | 17 | 16 | 6 | 4 | 2 | 3 | div.s $f2, $f4, $f6 |
| add.d | R | 17 | 17 | 6 | 4 | 2 | 0 | add.d $f2, $f4, $f6 |
| sub.d | R | 17 | 17 | 6 | 4 | 2 | 1 | sub.d $f2, $f4, $f6 |
| mul.d | R | 17 | 17 | 6 | 4 | 2 | 2 | mul.d $f2, $f4, $f6 |
| div.d | R | 17 | 17 | 6 | 4 | 2 | 3 | div.d $f2, $f4, $f6 |
| lwc1 | I | 49 | 20 | 2 | 100 | | | lwc1 $f2, 100 ($s4) |
| swc1 | I | 57 | 20 | 2 | 100 | | | swc1 $f2, 100 ($s4) |
| bc1t | I | 17 | 8 | 1 | 25 | | | bc1t 25 |
| bc1f | I | 17 | 8 | 0 | 25 | | | bc1f 25 |
| c.lt.s | R | 17 | 16 | 4 | 2 | 0 | 60 | c.lt.s $f2, $f4 |
| c.lt.d | R | 17 | 17 | 4 | 2 | 0 | 60 | c.lt.d $f2, $f4 |
| 域宽 | | 6 位 | 5 位 | 5 位 | 5 位 | 5 位 | 6 位 | 所有 MIPS 指令都是 32 位宽 |

图 3-17 （续）

| | op[31:26]： | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 28 ~ 26<br>31 ~ 29 | 0(000) | 1(001) | 2(010) | 3(011) | 4(100) | 5(101) | 6(110) | 7(111) |
| 0(000) | Rfmt | Blts/cc | j | jal | beq | bne | blez | bgtz |
| 1(001) | addi | addiu | slti | sltiu | ANDi | ORi | xORi | lui |
| 2(010) | TLB | FlPt | | | | | | |
| 3(011) | | | | | | | | |
| 4(100) | lb | lh | lwl | lw | lbu | lhu | lwr | |
| 5(101) | sb | sh | swl | sw | | | swr | |
| 6(110) | lwc0 | lwc1 | | | | | | |
| 7(111) | swc0 | swc1 | | | | | | |

图 3-18 MIPS 浮点指令编码。标记是按照行和列给出域值。例如，在图的顶端部分，在第 4 行（指令的 31 ~ 29 位为 $100_2$）和第 3 列（指令的 28 ~ 26 位为 $011_2$）可以发现 lw，所以 op 域（31 ~ 26 位）相应的值为 $100011_2$。带下划线意味着该域用于其他地方。例如，在第 2 行第 1 列的 FlPt（op = $010001_2$）定义在图的底端。因此，在图底部第 0 行第 1 列的 sub.f 意味着 funct 域（指令的 5 ~ 0 位）为 $000001_2$ 且 op 域（31 ~ 26 位）是 $010001_2$。注意在图的中间给出的 5 位的 rs 域，决定了操作是单精度（$f = s$，所以 rs = 10000）还是双精度（$f = d$，所以 rs = 10001）。类似地，指令的 16 位决定了指令 bc1.c 是测试为真（16 位 = 1 = > bc1.t）还是为假（16 位 = 0 = > bc1.f）。加粗的指令是在第 2 章或者本章描述过的，附录 A 给出了全部指令。这个信息也可以在 MIPS 参考数据卡的第 2 列里找到

212 ~ 213

op(31:26)=010001(COP1); if(fct(16:16)=0=>s=f, fct(16:16)=1=>s=d), rs(25:21);

| 23~21<br>25~24 | 0(000) | 1(001) | 2(010) | 3(011) | 4(100) | 5(101) | 6(110) | 7(111) |
|---|---|---|---|---|---|---|---|---|
| 0(00) | mfc1 | | cfc1 | | mtc1 | | ctc1 | |
| 1(01) | bc1.c | | | | | | | |
| 2(10) | f=单精度 | f=双精度 | | | | | | |
| 3(11) | | | | | | | | |

op(31:26)=010001(COP1); if(二进制位: 10000=0=>f=s; 10001=0=>f=d), funct(5:0);

| 2~0<br>5~3 | 0(000) | 1(001) | 2(010) | 3(011) | 4(100) | 5(101) | 6(110) | 7(111) |
|---|---|---|---|---|---|---|---|---|
| 0(000) | add.f | sub.f | mul.f | div.f | | abs.f | mov.f | neg.f |
| 1(001) | | | | | | | | |
| 2(010) | | | | | | | | |
| 3(011) | | | | | | | | |
| 4(100) | cvt.s.f | cvt.d.f | | | cvt.w.f | | | |
| 5(101) | | | | | | | | |
| 6(110) | c.f.f | c.un.f | c.eq.f | c.ueq.f | c.olt.f | c.ult.f | c.ole.f | c.ule.f |
| 7(111) | c.sf.f | c.ngle.f | c.seq.f | c.ngl.f | c.lt.f | c.nge.f | c.le.f | c.ngt.f |

图3-18 (续)

**硬件/软件接口** 在支持浮点算术方面，体系结构设计者面临着一个问题：是否和整数指令使用相同的寄存器，或者为浮点增加一组专用的寄存器。因为程序通常对不同的数据执行算数和浮点操作，独立的寄存器会增加程序中要执行的指令数目。主要的影响是需要建立一组独立的指令用于浮点寄存器和内存之间数据的传输。

单独的浮点寄存器的好处是倍增了寄存器数目而不需要在指令格式中增加更多的位数，同时因为使用了相互独立的整数和浮点寄存器堆而倍增了寄存器带宽。另外，还可以量身定做针对浮点的寄存器，例如，一些计算机将寄存器中各种大小的源操作数转化为一种单一的内部格式。

**例题：将浮点C程序编译为MIPS汇编代码**

将华氏温度转为摄氏温度：

```
float f2c (float fahr)
{
        return ((5.0/9.0) *(fahr - 32.0));
}
```

假设浮点变量fahr存放在$f12中，结果存放在$f0中。（不像整数寄存器，浮点寄存器0也可以存储数据。）那么MIPS汇编代码是什么？

**答案**

我们假设编译器将三个浮点常数放置在内存中，并且可以用全局指针$gp很容易地获得。首先前两个取数指令将常数5.0和9.0载入浮点寄存器：

```
f2c:
    lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 in memory)
    lwc1 $f18,const9($gp) # $f18 = 9.0 (9.0 in memory)
```

然后相除得到分数5.0/9.0：

```
    div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
```

（许多编译器在编译的时候就做了5.0除以9.0的操作，并将单精度常数5.0/9.0存入内存，从而在运行的时候避免做除法。）下面，我们将常数32.0载入，然后将其从fahr（$f12）中减去：

```
lwc1 $f18, const32($gp)# $f18 = 32.0
sub.s $f18, $f12, $f18 # $f18 = fahr - 32.0
```

最后，我们将两个中间结果相乘，积作为返回结果放在$f0中，然后程序返回

```
mul.s $f0, $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr $ra                # return
```

现在，让我们做浮点矩阵操作，其代码在科学计算程序中是常见的。 □

**例题**·将二维矩阵的浮点C程序编译为MIPS

许多浮点计算都采用双精度。现在做矩阵乘法C = C + A* B，这通常称为双精度通用矩阵乘法（简称DGEMM）。我们将在3.8节以及后面的第4、5、6章中都会看到该版本。假定A、B、C都是32×32的矩阵。

```
void mm (double c[][], double a[][], double b[][])
{
        int i, j, k;
        for (i = 0; i != 32; i = i + 1)
        for (j = 0; j != 32; j = j + 1)
        for (k = 0; k != 32; k = k + 1)
          c[i][j] = c[i][j] + a[i][k] *b[k][j];
}
```

数组的开始地址都是参数，存在$a0、$a1、$a2中。假设整数变量分别存在$s0、$s1、$s2中。这段程序的MIPS汇编代码是什么？

**答案**

注意到c[i][j]处于上面循环的最里面。因为循环变量是k，不影响c[i][j]，所以我们可以避免在每次迭代时载入和存储c[i][j]。相反，编译器每次在循环外将c[i][j]载入一个寄存器，然后将a[i][k]和b[k][j]的积累加到这个寄存器里，在最里层的循环结束后将和存入c[i][j]。 215

为了保持代码简洁，我们使用汇编语言的伪指令li（其将一个常数载入一个寄存器）、l.d和s.d（汇编器将其变为一对数据传送指令、lwc1和swc1，向一对浮点寄存器传送数据）。

程序段首先将循环结束值32存入一个临时寄存器中，然后初始化循环变量：

```
  mm:...
     li      $t1, 32  # $t1 = 32 (row size/loop end)
     li      $s0, 0   # i = 0; initialize 1st for loop
L1:  li      $s1, 0   # j = 0; restart 2nd for loop
L2:  li      $s2, 0   # k = 0; restart 3rd for loop
```

要计算c[i][j]的地址，我们首先要知道一个32×32的二维矩阵是如何在内存中存储的。正如你所料，它的排布如同32个有32个元素的一维矩阵。所以获得我们需要的元素的第一步是跳过第i个“一维矩阵”或者第i行。因此，我们用首维的索引乘以行的大小，32。因为32是以2为底的指数值，所以我们可以用移位来替代：

```
sll  $t2, $s0, 5      # $t2 = i * 2^5 (size of row of c)
```

现在我们加上第二维的索引来获得我们想要的那行的第j个元素：

```
addu  $t2, $t2, $s1   # $t2 = i * size(row) + j
```

为了将这个和转化为按字节的索引，我们给它乘上矩阵元素所占的字节大小。因为每个元素都是双精度的，所以占了8字节，我们用左移3位来代替：

```
sll  $t2, $t2, 3      # $t2 = byte offset of [i][j]
```

下面我们将这个和加上c的基地址，得到c[i][j]的地址，然后将双精度数c[i][j]载入$f4寄存器中：

```
addu  $t2, $a0, $t2   # $t2 = byte address of c[i][j]
l.d   $f4, 0($t2)     # $f4 = 8 bytes of c[i][j]
```

接着的5条指令类似于刚才的5条：计算双精度数b[k][j]的地址，然后将其载入。

```
L3: sll $t0, $s2, 5     # $t0 = k * 2^5 (size of row of b)
    addu $t0, $t0, $s1  # $t0 = k * size(row) + j
    sll $t0, $t0, 3     # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0  # $t0 = byte address of b[k][j]
    l.d $f16, 0($t0)    # $f16 = 8 bytes of b[k][j]
```

类似地，下面的5条指令像刚才的5条一样：计算双精度数a[i][k]的地址，然后将其
216 载入。

```
sll     $t0, $s0, 5     # $t0 = i * 2^5 (size of row of a)
addu    $t0, $t0, $s2   # $t0 = i * size(row) + k
sll     $t0, $t0, 3     # $t0 = byte offset of [i][k]
addu    $t0, $a1, $t0   # $t0 = byte address of a[i][k]
l.d     $f18, 0($t0)    # $f18 = 8 bytes of a[i][k]
```

现在已经载入了所有的数据，我们终于可以做一些浮点操作了！我们将分别存在$f18和$f16中的a、b的元素相乘，然后累加到$f4中。

```
mul.d $f16, $f18, $f16 # $f16 = a[i][k] * b[k][j]
add.d $f4, $f4, $f16   # f4 = c[i][j] + a[i][k] * b[k][j]
```

最后的部分将循环变量k加1，如果索引值没到32，则再次返回循环。如果到了32，则结束最里层的循环，将放在$f4中的累加和存入c[i][j]。

```
addiu  $s2, $s2, 1     # sk = k + 1
bne    $s2, $t1, L3    # if (k != 32) go to L3
s.d    $f4, 0($t2)     # c[i][j] = $f4
```

类似地，最后4条指令增加中间和最外层的循环变量，如果没有到32则返回循环，否则在到达32后退出循环。

```
addiu  $s1, $s1, 1     # $j = j + 1
bne    $s1, $t1, L2    # if (j != 32) go to L2
addiu  $s0, $s0, 1     # $i = i + 1
bne    $s0, $t1, L1    # if (i != 32) go to L1
```

后面的图3-22给出了DGEMM的x86汇编语言代码，该版本与图3-21中的DGEMM版本略有不同。 □

**解释** 上面例子中的阵列排布，称为行主序列，用于许多C和其他编程语言中。但Fortran采用的是列主序列，即阵列是一列一列地存储。

**解释** 32个MIPS浮点寄存器中，只有16个能用于双精度操作：$f0，$f2，$f4，…，$f30。计算中，双精度使用了成对的单精度寄存器。奇数编号的浮点寄存器只是载入和存储64位浮点数的右半部分。MIPS-32给指令集增加了l.d和s.d指令。MIPS-32也为所有浮点指令增加了“单精度配对”（paired single）版本，这里每个单指令能够对两个32位的源操作数并行执行浮点操作，这两个32位的源操作数存在64位的寄存器中（见3.6节）。例如，add.ps $f0，$f2，$f4等价于add.s $f0，$f2，$f4和add.s $f1，$f3，
217 $f5。

**解释** 将整数和浮点寄存器分开的另外一个原因是在20世纪80年代的处理器还没有足够的晶体管将浮点单元和整数单元放在同一个芯片上。因此，浮点单元，包括浮点寄存器，只是一个备选的辅助芯片。这个可选的加速芯片称为协处理器。按首字母缩写的MIPS的浮点load指令lwc1的意思是载入一个字到协处理器1，浮点单元。（协处理器0处理虚拟内存，

第5章对其进行描述。）自20世纪90年代早期，微处理器已经将浮点单元和其他功能单元集成在一个芯片上。因此，由加速器和内置存储器组成的协处理器的术语已经过时了。

**解释** 正如3.4节提到的，加速除法比乘法更有挑战性。除了SRT，还有一种利用乘进乘法器的技术，称为牛顿迭代，它将除法变换为通过寻找函数的零点来求倒数1/c，然后将其乘以另一操作数。如果不计算足够多的位，迭代技术是无法进行正确舍入的。如TI的一款芯片通过计算倒数更多有效位的方法来解决这一问题。

**解释** Java在定义浮点数据类型和操作时遵循IEEE 754标准。因此，可以更好地生成第一个例子中的代码，是一种经典的将华氏温度转换为摄氏温度的方法。

第二个例子里使用了多维矩阵，不被Java显式支持。Java允许在数组中嵌套数组，但是不支持像C中的多维矩阵，每个数组可能有自己的长度。像第2章中的那些例子，第二个例子的Java版本需要大量的代码来进行数组的边界检查，包括在行访问后对新的长度进行计算。它可能还需要检查对象引用是否非空。

### 3.5.5 算术精确性

与整数可以精确地表示在最大数和最小数之间的所有数不同，浮点数通常是一个无法表示的数的近似。原因是，假定在0和1之间，实数就有无穷多个，而双精度最多可以精确表示$2^{62}$个。我们能做到最好的就是给出最接近实际数的浮点表示。因此，IEEE 754提供了几种舍入模式来供程序员选择他们想要的近似策略。

- 保护位：在浮点数中间计算中，在右边多保留的两位中的首位；用于提高舍入精度。
- 舍入位：在浮点数中间计算中，在右边多保留的两位中的第二位；使浮点中间结果满足浮点格式，得到最接近的数。

舍入听起来很简单，但它需要硬件支持在计算中产生更多的有效位。在前面的例子中，我们对中间结果占有多少位未做介绍，但很明显的是，如果每个中间结果都截短成准确的位数，就没法做舍入了。IEEE 754因此在中间计算中，右边总是多保留两位，分别称为**保护位**（guard）和**舍入位**（round）。我们用一个十进制的例子来说明它们的作用。 218

**例题·使用保护位来舍入**

将$2.56_{10}\times10^{0}$和$2.34_{10}\times10^{2}$相加，假设我们有3位十进制尾数。首先使用保护位和舍入位将其舍入到只有三位尾数的最近数，然后不用保护位和舍入位再做一次（舍入）。

**答案**

首先我们右移较小数以对齐指数，所以$2.56_{10}\times10^{0}$变为$0.025\,6_{10}\times10^{2}$。因为有了保护位和舍入位，所以当我们对齐指数时可以表示两个最低位。保护位为5而舍入位为6。求和：

$$\begin{array}{r} 2.340_{10} \\ +\ 0.025\,6_{10} \\ \hline 2.365\,6_{10} \end{array}$$

因此，和为$2.365\,6_{10}\times10^{2}$。因为需要舍入掉两位，所以我们需要以50为分水岭，在其值为0～49之间时舍掉，在51～99之间时向上舍入。向上舍入这个和，变为$2.37_{10}\times10^{2}$。

在计算中，在没有保护位和舍入位的情况下舍入掉两位。新的和为：

$$\begin{array}{r} 2.34_{10} \\ +\ 0.02_{10} \\ \hline 2.36_{10} \end{array}$$

答案是 $2.36_{10} \times 10^{7}$，比上面的结果在最低位上少 1。 □

因为最精确的情况是实际的数在两个浮点表示的中间，浮点的精确性通常是用尾数的最低位上有多少位的误差来衡量。这种衡量称为**尾数最低位**（unit in the last place）的数目，即 ulp。如果一个数在最低位上少 2，则称其少了 2 个 ulp。在没有上溢、下溢或无效操作异常的情况下，IEEE 754 保证了计算机使用的数的误差都在半个 ulp 以内。

⊃ 尾数最低位：在实际数和能表达的数之间的有效数最低位上的误差位数。

**精解** 尽管上面的例子实际只多需要一位，但乘法需要两位。一个二进制乘积可能有一位前导 0；因此，规格化步骤必须将积左移一位。这个移位会将保护位移入积的最低位，留下舍入位来精确地舍入乘积。

IEEE 754 有四种舍入模式：总是向上舍入（向 $+\infty$），总是向下舍入（向 $-\infty$），截断舍入，向最靠近的偶数舍入。最后一种模式给出了当数值在中间时如何去做。美国国税局（IRS）也许为了自身的利益，总是将 0.50 美元向上舍入。一种更公平的办法是：一半时间里使用向上舍入，另一半时间里使用向下舍入。IEEE 754 处理这种中间情况的方法是如果最后一位是奇数，就加 1；如果是偶数，则减去。这种方法总是在中间情况下将最低位设为 0，这就是舍入模式的名称。这种模式是用得最多的，而且是唯一被 Java 支持的模式。

219

使用额外的舍入位的目的是让计算机获得相同的结果，就如同是先以无穷的精度计算中间结果，然后执行舍入那样。为了支持这个目标并向最靠近的偶数舍入，IEEE 754 标准在保护位和舍入位之后还有一位粘贴位（sticky bit）；当舍入位右边的数非零时将它置 1。粘贴位可以让计算机在舍入时，能够区分 $0.50\cdots00_{10}$ 和 $0.50\cdots01_{10}$。

粘贴位可能被置 1，例如，在加法中，当较小数右移时就可能这样。假设在前面的例子里我们将 $5.01_{10} \times 10^{-1}$ 和 $2.34_{10} \times 10^{2}$ 相加。即使有保护位和舍入位，我们将 0.0050 和 2.34 相加，得到 2.3450。粘贴位会被置 1，因为右边是非零的。假设没有粘贴位来记住是否有 1 被移走，我们会假设这个数等于 $2.345000\cdots00$，然后向最靠近的偶数舍入得到 2.34。使用粘贴位记住这个数是大于 $2.345000\cdots00$ 的，我们舍入后会得到 2.35。

⊃ 粘贴位：同保护位和舍入位一样用于舍入的位，当舍入位右边有非零的数据时将其置 1。

**精解** PowerPC、SPARC64、AMD SSE5 和 Intel AVX 体系结构提供了一个单独的指令来对三个寄存器执行乘法和加法操作：$a = a + (b \times c)$。很明显，因为这个操作常用，所以这条指令潜在地允许更高的浮点性能。同样重要的是替换掉了两次舍入——在乘法后和在加法后——其可能在分开的指令中出现，乘加指令只是在加法后执行一次舍入。一次舍入步骤增加了乘加的精度。这样的一次舍入的操作称为混合乘加（fused multiply add）。它已被加入修订的 IEEE 754-2008 标准里（见 3.11 节）。

⊃ 混合乘加：一条浮点指令，其执行一次乘法和一次加法，但只在加法后执行一次舍入。

### 3.5.6 小结

下面的重点再次强调了第 2 章中存储程序的概念：不能仅仅看看数据位就决定信息的含义，因为即使是相同的位也代表了不同的目标。这一节给出的计算机算术是有限精度的，因此

和自然的算术不同。例如，IEEE 754 的标准浮点表示为

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$$

几乎总是一个实数的近似。计算机系统必须小心弥合计算机算术和真实世界的算术之间的差距，而程序员有时也需要小心这种近似值的含义。

**重点** 位模式并没有内在的含义，它们可能表示有符号整数、无符号整数、浮点数和指令等。具体代表什么意思要看指令对该字的硬件值进行操作。 220

计算机数和真实世界里的数的主要不同是计算机数的大小是有限制的，为此限制了其精度；计算的数字有可能太大或太小而无法在一个字中表示。程序员必须记住这些限制并相应地编程。

| C 类型 | Java 类型 | 数据传送 | 操作 |
|---|---|---|---|
| int | int | lw,sw,lui | addu, addiu, subu, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti |
| Unsigned int | — | lw,sw,lui | addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu |
| char | — | lb,sb,lui | add, addi, sub, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti |
| — | char | lb,sb,lui | addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu |
| float | float | lwc1,swc1 | add.s,sub.s,mult.s,div.s,c.eq.s,c.lt.s,c.le.s |
| double | double | l.d,s.d | add.d,sub.d,mult.d,div.d,c.eq.d,c.lt.d,c.le.d |

**硬件/软件接口** 在上一章，我们提出了编程语言 C 的分配分类（见 2.7 节的硬件/软件接口部分）。上表给出了一些 C 和 Java 的数据类型、MIPS 数据传送指令，以及对出现在第 2 章和本章的那些数据类型的操作指令。注意 Java 省略了无符号整数。

**小测验**

假设有一个 16 位的 IEEE 754-2008 浮点格式，其中有 5 位指数位。那么它可能表示的数的范围是多少？

1. $1.0000\ 0000\ 00 \times 2^{0}$ 到 $1.1111\ 1111\ 11 \times 2^{31}$，0
2. $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ 到 $\pm 1.1111\ 1111\ 1 \times 2^{15}$，$\pm 0$，$\pm \infty$，NaN
3. $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ 到 $\pm 1.1111\ 1111\ 11 \times 2^{15}$，$\pm 0$，$\pm \infty$，NaN
4. $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ 到 $\pm 1.1111\ 1111\ 11 \times 2^{14}$，$\pm 0$，$\pm \infty$，NaN

**精解** 为了进行可能包含 NaN 的比较，IEEE 754 标准包含了有序和无序作为比较的选项。因此，完整的 MIPS 指令集有许多用于比较的指令来支持 NaN。（Java 不支持无序比较。） 221

为了从一次浮点操作中最大限度地获得精度，标准允许一些数以非规格化的形式出现。标准允许有非规格化数（也称为非规格化或者亚规格化），目的是使 0 和最小规格化数之间的间隙更小。在指数为零而有效数非零时，允许一个有效数逐步变小直到 0，称为逐步下溢（gradual underflow）。例如，最小的正的单精度规格化数为

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_2 \times 2^{-126}$$

而最小的单精度非规格化数为

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_2 \times 2^{-126}, \text{即 } 1.0_2 \times 2^{-149}$$

对于双精度，非规格化间隙为从 $1.0_2 \times 2^{-1022}$ 到 $1.0_2 \times 2^{-1074}$。

对于想建立一个快速浮点单元的设计者来说，可能偶尔出现的非规格化源操作数是一件令人头疼的事情。因此，许多计算机在源操作数为非规格化数时产生异常，让软件来处理相应的操作。尽管软件执行可以完美地处理，但它们低效的表现降低了非规格化数在可移植的浮点软件中的受欢迎程度。再者，如果程序员并不期望得到非规格化数，他们所写程序的执行效率之低也许会令他们感到惊讶。

## 3.6 并行性和计算机算术：子字并行

由于每台桌面计算机都有自己的图形显示器，因此，随着处理器中晶体管数量的增加，用于桌面计算机的微处理器不可避免地要增加支持图形操作的功能。

许多图形系统最初都是用8位数据来表示三种基本颜色中的一种，外加8位数据来表示像素的位置。电话会议中和视频游戏中使用的扬声器和麦克风要求对声音进行支持。音频采样需要8位以上的精度，但是16位精度已经足够。

每种处理器对于字节或半字都有特殊的支持，从而使得在存储器中占据较少的空间（见2.9节）。然而，在典型的整数程序中，对这些数据类型的算术操作非常少，因此几乎不支持除数据传送之外的操作。设计师们发现许多视频和音频应用中通常对这类数据的向量做相同的操作。通过在128位内对进位链进行分割，处理器可以同时对16个8位、8个16位、4个32位或2个64位的运算同时进行并行操作。对加法器进行这样的分割的开销非常小。

将这种在一个宽字内部进行若并行操作称为子字并行，也可将其称为更加通用的数据级并
222 行。它们也被称为向量或SIMD（单指令多数据，见6.6节）。多媒体应用的日益广泛促使支持易于并行实现窄位宽操作的算术运算指令的出现。

例如，ARM在NEON多媒体指令集中增加了100多条指令来支持子字并行，这些扩展的指令可以在ARMv7或ARMv8中实现。NEON中增加了宽度为256字节的寄存器，它们可以当作32个8字节宽度的寄存器或者16个16字节宽度的寄存器使用。除了64位浮点数之外，NEON支持你能够想到的任何子字数据类型。

- 8位、16位、32位和64位无符号整数和带符号整数。
- 32位浮点数。

图3-19给出了NEON基本指令的总结。

| 数据传送 | 算术运算 | 逻辑/比较 |
| --- | --- | --- |
| VLDR.F32 | VADD.F32, VADD{I,W}{S8,U8,S16,U16,S32,U32} | VAND.64, VAND.128 |
| VSTR.F32 | VSUB.F32, VSUB{I,W}{S8,U8,S16,U16,S32,U32} | VORR.64, VORR.128 |
| VLD{1,2,3,4}{8,16,32} | VMLL.F32, VMLL{S8,U8,S16,U16,S32,U32} | VEOR.64, VEOR.128 |

图3-19 子字并行的ARM NEON指令总结。使用大括号{}表示基本操作的可选对象：{S8, U8, 8}表示8位有符号和8位无符号整数或者与类型无关的8位数据，16个这些类型的数据可映射为一个128位寄存器；{S16, U16, 16}表示16位有符号和16位无符号整数或者与类型无关的16位数据，8个这些类型的数据可映射为一个128位寄存器；{S32, U32, 32}表示32位有符号和32位无符号整数或者与类型无关的32位数据，4个这些类型的数据可映射为一个128位寄存器；{S64, U64, 64}表示64位有符号和64位无符号整数或者与类型无关的64位数据，2个这些类型的数据可映射为一个128位寄存器；{F32}表示32位有符号或无符号浮点数，4个这种类型的数据可映射为一个128位寄存器；向量装载（vector load）把一个n元的结构从存储器读入1个、2个、3个或4个NEON寄存器中。它把一个n元（element）的结构装载到一个线性结构中（见6.6节），寄存器中没有进行装载的部分保持不变。向量存储（vector store）将1个、2个、3个或4个NEON寄存器中的内容写入存储器中的一个结构中去

| 数据传送 | 算术运算 | 逻辑/比较 |
| --- | --- | --- |
| VST[1,2,3,4],[I8,I16,I32] | VMLA.F32,VMLAL[S8,U8,S16,U16,S32,U32] | VBIC.64, VBIC.128 |
| VMOV.[I8,I16,I32,F32],#imm | VMLS.F32,VMLSL[S8,U8,S16,U16,S32,U32] | VORN.64, VORN.128 |
| VMVN.[I8,I16,I32,F32],#imm | VMAX.[S8,U8,S16,U16,S32,U32,F32] | VCEQ.[I8,I16,I32,F32] |
| VMOV.[I64,I128] | VMIN.[S8,U8,S16,U16,S32,U32,F32] | VCGE.[S8,U8,S16,U16,S32,U32,F32] |
| VMVN.[I64,I128] | VABS.[S8,S16,S32,F32] | VCGT.[S8,U8,S16,U16,S32,U32,F32] |
| | VNEG.[S8,S16,S32,F32] | VCLE.[S8,U8,S16,U16,S32,U32,F32] |
| | VSHL.[S8,U8,S16,U16,S32,U32,S64,U64] | VCLT.[S8,U8,S16,U16,S32,U32,F32] |
| | VSHR.[S8,U8,S16,U16,S32,U32,S64,U64] | VTST.[I8,I16,I32] |

图3-19 （续）

**精解** 除了有符号和无符号整数外，ARM还包含4种大小的“定点”格式，分别是I8、I16、I32和I64，16个I8、8个I16、4个I32或2个I64可以映射到1个128位的寄存器。定点数的一部分是尾数（二进制小数点的右边），另一部分是整数（二进制小数点的左边）。二进制小数点的位置在软件层面上可见。许多ARM处理器没有浮点硬件，因此浮点操作必须使用库例程来实现。定点算术运算要比软件实现的库例程快得多，但是程序员要做更多的工作。

223

## 3.7 实例：x86中流处理SIMD扩展和高级向量扩展

最初，x86中的MMX（MultiMedia eXtension，多媒体扩展）指令和SSE（Streaming SIMD Extension，流处理SIMD扩展）指令与ARM NEON中的操作类似。第2章中提到在2001年Intel在其体系结构中增加了144条指令作为SSE2的一部分，包括了双精度浮点寄存器和操作。它包含了可用作浮点操作数的8个64位寄存器。AMD将寄存器数量扩展到16个，作为AMD64的一部分，称之为XMM，这些寄存器被Intel重新称为EM64T。图3-20总结了SSE和SSE2指令。

| 数据传送 | 算术 | 比较 |
| --- | --- | --- |
| MOV[A/U][SS/PS/SD/PD] xmm, mem/xmm | ADD[SS/PS/SD/PD] xmm, mem/xmm | CMP[SS/PS/SD/PD] |
| | SUB[SS/PS/SD/PD] xmm, mem/xmm | |
| MOV[H/L][PS/PD] xmm, mem/xmm | MUL[SS/PS/SD/PD] xmm, mem/xmm | |
| | DIV[SS/PS/SD/PD] xmm, mem/xmm | |
| | SQRT[SS/PS/SD/PD] mem/xmm | |
| | MAX[SS/PS/SD/PD] mem/xmm | |
| | MIN[SS/PS/SD/PD] mem/xmm | |

图3-20 x86的SSE/SSE2浮点指令。xmm是指一个128位SSE2寄存器操作数；mem/xmm是指该操作数要么是一个存储器操作数，要么是一个SSE2寄存器操作数。用大括号[]表示基本操作可选择的类型：[SS]表示标量的单精度浮点数，或128位SSE2寄存器中的1个32位操作数；[PS]表示组合的单精度浮点数，或者128位SSE2寄存器中的4个32位操作数；[SD]表示标量双精度浮点数，或128位寄存器中的一个64位操作数；[PD]表示组合的双精度浮点数，或者128位SSE2寄存器中的2个64位操作数；[A]表示存储器中对齐的128位操作数；[U]表示存储器中不对齐的128位操作数；[H]表示传送128位操作数的高半部分；[L]表示传送128位操作数的低半部分

224

在一个寄存器中，除了能够存放一个单精度或双精度数之外，Intel 允许将多个操作数组合在一起放在一个 128 位 SSE2 寄存器中：4 个单精度或 2 个双精度数。因此，SSE2 的 16 个浮点寄存器实际上是 128 位宽。如果操作数能够在存储器中组织为 128 位对齐的数据，则每条 128 位的数据传送指令可以装载（load）或保存（store）多个操作数。这种组合的浮点数格式也可以并行进行 4 个单精度（PS）或 2 个双精度（PD）数运算的算术操作支持。

2011 年，Intel 使用高级向量扩展（advanced vector extension，AVX）将寄存器宽度再次加倍，现在称之为 YMM。因此，现在单精度操作可以指定 8 个 32 位浮点运算或 4 个 64 位浮点运算。现在，SSE 和 SSE2 指令可以对 YMM 寄存器的低 128 位进行操作。因此，为了使用 128 位和 256 位操作，在 SSE2 汇编指令操作码前加上前缀字母 v（表示向量），然后使用 YMM 寄存器名字替代 XMM 寄存器名字。例如，进行 2 个 64 位浮点加法[⊖]操作的 SSE2 指令。

```
addpd   %xmm0, %xmm4
```

变为

```
vaddpd  %ymm0, %ymm4
```

该指令进行 4 个 64 位浮点加法[⊜]。

**精解** AVX 也对 x86 增加了 3 地址指令。例如，vaddpd 可以有如下形式：

```
vaddpd %ymm0, %ymm1, %ymm4 # %ymm4 = %ymm1 + %ymm2
```

而标准的 2 地址指令为：

```
addpd  %xmm0, %xmm4 # %xmm4 = %xmm4 + %xmm0
```

与 MIPS 不同，x86 的目标操作数位于右边。3 地址可以减少计算所需的寄存器数量和指令数量。

## 3.8 加速：子字并行和矩阵乘法

为了说明子字并行对性能的影响，我们将在 Intel Core i7 上在有无 AVX 的情况下运行相同的代码。图 3-21 给出了一个用 C 语言写的矩阵乘法的代码，它还没有进行优化。就像我们在 3.5 节看到的，该程序通常称为 DGEMM，表示双精度通用矩阵乘法。从该版本出发，我们增加一个新的称为“加速”的小节来说明在底层硬件的基础上使用适应的软件的性能提升。这里的底层硬件是 Intel Core i7 的 Sandy Bridge。在第 3、4、5、6 章中，该小节将逐步使用每章介绍的思想来提高 DGEMM 的性能。

图 3-22 给出了图 3-21 中内循环的 x86 汇编代码。以 v 开头的 5 条浮点指令与 AVX 指令类似，但是需要注意的是它们使用的是 XMM 寄存器，而不是 YMM 寄存器，另外它们在指令名
225 字里包含了 sd，代表着向量双精度。我们将对子字并行指令作简要定义。

由于编译程序员最终能够使用 x86 的 AVX 指令生成高质量代码，因此现在我们必须使用 C 循环体的属性，通过“欺骗”的方式，告诉编译器如何生成高质量的代码。图 3-23 是图 3-21 的加强版，给出了 Gnu C 编译器产生的 AVX 代码。图 3-24 给出了带注释的 x86 代码，它是在编译时使用 gcc-O3 级优化选项的输出。

图 3-23 第 6 行的声明使用了 __m256d 的数据类型，用来告诉编译器变量将保存 4 个双精度浮点值。第 6 行的内嵌函数 _mm256_loadu_pd( ) 使用 AVX 指令从矩阵 C 中并行的（_pd）取出 4 个双精度浮点数到 c0。第 6 行的地址计算 C + i + j * n 表示元素 C[i + j * n]。与之相应的

---

⊖ 原文为“乘法”。——译者注

⊜ 原文为“乘法”。——译者注

是在第 11 行中的最后一步，使用内蕴函数_mm256_store_pd()将 c0 中的 4 个双精度浮点数保存到矩阵 C 中。由于在每次迭代时处理 4 个元素，第 4 行中的外层 for 循环的循环变量 i 做加 4 操作，而不像图 3-21 中第 3 行的加 1 操作。

```
1.  void dgemm (int n, double* A, double* B, double* C)
2.  {
3.    for (int i = 0; i < n; ++i)
4.      for (int j = 0; j < n; ++j)
5.      {
6.        double cij = C[i+j*n]; /* cij = C[i][j] */
7.        for( int k = 0; k < n; k++ )
8.          cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.        C[i+j*n] = cij; /* C[i][j] = cij */
10.     }
11. }
```

图 3-21 未经优化的双精度矩阵乘法的 C 语言版本，称之为双精度通用矩阵乘法（GEMM）。因为我们通过参数 n 传递矩阵的维数，该版本的 DGEMM 使用的是矩阵 C、A、B 的一维版本，并且强调使用算术运算来获得更好的性能，而不像 3.5 节中使用直观的二维阵列。注释提醒我们使用这种更加直观的符号

```
1.  vmovsd (%r10),%xmm0             # Load 1 element of C into %xmm0
2.  mov    %rsi,%rcx                # register %rcx = %rsi
3.  xor    %eax,%eax                # register %eax = 0
4.  vmovsd (%rcx),%xmm1             # Load 1 element of B into %xmm1
5.  add    %r9,%rcx                 # register %rcx = %rcx + %r9
6.  vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
7.  add    $0x1,%rax                # register %rax = %rax + 1
8.  cmp    %eax,%edi                # compare %eax to %edi
9.  vaddsd %xmm1,%xmm0,%xmm0        # Add %xmm1, %xmm0
10. jg     30 <dgemm+0x30>          # jump if %eax > %edi
11. add    $0x1,%r11d               # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)             # Store %xmm0 into C element
```

图 3-22 图 3-21 中嵌套循环体使用优化编译后对应的 x86 汇编语言代码。虽然只是 64 位数据，编译器使用了 AVX 版本指令，而不是 SSE2，因此每条指令可以使用 3 个地址而不是 2 个地址（见 3.7 节的精解）

在循环体内部，首先在第 9 行又一次使用_mm256_loas_pd()取入 A 的 4 个元素。为了将这些元素与 B 的一个元素相乘，第 10 行使用了内蕴函数_mm256_broadcast_sd()，它将标量双精度数复制为相同的 4 份——在这种情况下，B 的一个元素在一个 YMM 寄存器中。在第 9 行中，使用_mm256_mul_pd()同时乘 4 个双精度结果。最后，第 8 行的_mm256_add_pd()将 4 个乘积加到 c0 的 4 个和上。

图 3-24 给出了编译器生成的内循环体的 x86 代码。可以看到 5 条 AVX 指令——它们全部

以 v 开头，并且其中 4 条使用了 pd 表示并行的双精度——与前面提到的 C 内在属性一致。代码与图 3-22 中所示代码非常类似：都使用 12 条指令，整数指令几乎相同（但使用不同的寄存器），浮点指令的不同之处仅仅在于使用 XMM 寄存器的标量双精度（sd）和使用 YMM 寄存器的并行双精度（pd）。不同之处在于图 3-24 的第 4 行，A 中每个元素必须与 B 中每个元素相乘。一种解决方法是将 64 位 B 元素的 4 个相同的备份依次放入 256 位的 YMM 寄存器中，正如 vbroadcastsd 指令所做的工作。

对于 32 乘 32 的矩阵，图 3-21 中未优化的 DGEMM 在一个 2.6GHz 的 Intel Core i7（Sandy Bridge）的一个核上运行时性能为 1.7GFLOPS（每秒浮点操作次数）。图 3-23 中的优化代码的性能为 6.4GFLPOS。由于使用了了字并行，在许多操作中可以获得 4 倍的加速，因此 AVX 版本要比原始版本快 3.85 倍，性能增加了接近 4 倍。

```
1.  #include <x86intrin.h>
2.  void dgemm (int n, double* A, double* B, double* C)
3.  {
4.    for ( int i = 0; i < n; i+=4 )
5.      for ( int j = 0; j < n; j++ ) {
6.        __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.        for( int k = 0; k < n; k++ )
8.          c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k,j] */
9.                  _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                 _mm256_broadcast_sd(B+k+j*n)));
11.       _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.     }
13. }
```

图 3-23 DGEMM 的优化 C 版本，使用 C 的循环体属性为 x86 生成 AVX 子字并行指令。图 3-24 显示了内循环编译后的汇编语言

```
1.  vmovapd (%r11),%ymm0            # Load 4 elements of C into %ymm0
2.  mov     %rbx,%rcx               # register %rcx = %rbx
3.  xor     %eax,%eax               # register %eax = 0
4.  vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5.  add     $0x8,%rax               # register %rax = %rax + 8
6.  vmulpd  (%rcx),%ymm_,%ymm_      # Parallel mul %ymm1,4 A elements
7.  add     %r9,%rcx                # register %rcx = %rcx + %r9
8.  cmp     %r10,%rax               # compare %r10 to %rax
9.  vaddpd  %ymm1,%ymm0,%ymm0       # Parallel add %ymm1, %ymm0
10. jne     50 <dgemm+0x50>         # jump if not %r10 != %rax
11. add     $0x1,%esi               # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)            # Store %ymm0 into 4 C elements
```

图 3-24 编译图 3-23 中优化的 C 代码生成的嵌套循环体的 x86 汇编语言。注意与图 3-22 相同，区别仅在于 5 个浮点操作现在使用 YMM 寄存器和 pd 版本的指令来进行并行双精度操作，而不是 sd 版本的标量双精度

**1** **精解** 同1.6节中的精解相同，Intel提供了Turbo模式，暂时运行在较高时钟频率下，直到芯片过热。Intel Core i7（Sandy Bridge）在Turbo模式可将频率从2.6GHz增加到3.3GHz。以上的结果是在关闭Turbo模式下获得的。如果将Turbo模式打开，由于时钟频率提高了3.3/2.6=1.27倍，未优化的DGEMM性能将提升为2.1GFLOPS，AVX性能将提升为8.1GFLOPS。当一个八核芯片中只使用一个核时，Turbo模式会很好的工作，因为在这种情况下，单个核可以使用比共享情况下更多的功耗。 228

## 3.9 谬误与陷阱

*数学可以被定义为这样的学科，我们绝不知道我们在谈论什么以及我们所谈论的是否正确。*

——伯兰特·罗素，《近来关于数学原理的发言》，*1901*

算术中常见的谬误与陷阱通常是由计算机算术的有限精度和自然算术的无限精度之间的差异引起的。

谬误：正如整数乘法中左移指令可以代替与2的幂次方数相乘一样，右移指令也可以代替与2的幂次方数相除。

回忆一下二进制数$x$，其中$xi$代表第$i$位，有

$$\cdots + (x3 \times 2^3) + (x2 \times 2^2) + (x1 \times 2^1) + (x0 \times 2^0)$$

将$x$右移$n$位看起来似乎是和被$2^n$相除相同。[⊖]事实上，对于无符号整数确实如此。问题出在有符号整数上。例如，假设我们用$-5_{10}$除以$4_{10}$，商就是$-1_{10}$。$-5_{10}$的补码形式是

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_2$$

根据这个谬误，右移2位就是被$4_{10}$除（$2^2$）：

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2$$

由于符号位上是0，所以结果很明显是错的。右移后的值实际是$1\,073\,741\,822_{10}$而不是$-1_{10}$。

一种解决办法是算术右移时，进行符号位扩展而不是移入0。$-5_{10}$算术右移2位得到

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

结果是$-2_{10}$而不是$-1_{10}$，虽然很接近，但依然不正确。

陷阱：浮点加法是不能使用结合律的。

结合律适用于一系列整型的二进制补码加法，即使在计算过程中发生溢出。然而，因为浮点数是实数的近似表示，且计算机算术精度有限，结合律不能适用于浮点数。假定浮点数可以表示一个很大的数的范围，当两个不同符号的大数与一个小数相加时就会发生问题。例如，对于$c+(a+b)=(c+a)+b$，假设$c=-1.5_{10}\times10^{38}$，$a=1.5_{10}\times10^{38}$，$b=1.0$，它们都是单精度数。 229

$$\begin{aligned} c+(a+b) &= -1.5_{10}\times10^{38} + (1.5_{10}\times10^{38} + 1.0) \\ &= -1.5_{10}\times10^{38} + 1.5_{10}\times10^{38} \\ &= 0.0 \\ (c+a)+b &= (-1.5_{10}\times10^{38} + 1.5_{10}\times10^{38}) + 1.0 \\ &= (0.0)_{10} + 1.0 \\ &= 1.0 \end{aligned}$$

由于浮点数精度有限且结果是实数结果的近似值，$1.5_{10}\times10^{38}$远远大于$1.0_{10}$，因此$1.5_{10}\times10^{38}+1.0$仍然是$1.5_{10}\times10^{38}$，这就是为什么$c$、$a$、$b$的和根据浮点加法的计算顺序不同有0.0

⊖ 此处原书中$x$为C，我们认为此处$x$更合适，因为后边表示用$xi$。原书中$x^3$、$x^2$有误，3、2不应为上角。$2^n$也有误，$n$也不应为上角。——译者注

和1.0两种结果，所以 $c+(a+b) \neq (c+a)+b$，因此浮点加法不能使用结合律。

谬误：并行执行策略不仅适用于整型数据类型，同样也适用于浮点数据类型。

一般情况下，首先编写串行运行的程序，然后再编写并行运行的程序，这就自然产生一个问题："两个版本的程序能否得到相同的结果？"如果是否定的答案，那么你就得推断并行程序中有一个需要消除的 bug。

该方法假定将串行结构转化为并行结构时，计算机算术不会影响计算结果。这就是说，如果要将100万个数相加，无论使用1个处理器还是使用100个处理器应该得到相同的结果。该假定适用于二进制补码整数，因为整数加法可结合。然而，因为浮点加法不能结合，所以该假定不适用。

这个谬误有一个更加令人烦恼的情况在并行机上可能发生。并行机上操作系统调度器会根据并行程序的运行情况来使用不同数目的处理器。对并行无意识的程序员可能会因为程序每次运行结果总有些不同而苦恼，即使是相同的代码和输入，这是因为每次运行使用不同数目的处理器可能导致浮点求和以不同的顺序进行。

在这个困境下，写并行代码并使用了浮点数的程序员需要验证结果是否可信，即便结果可能与顺序执行的结果不一致。处理这个问题的领域称为数值分析，关于该问题本身就可以写一本教科书。这也是像 LAPACK 和 SCALAPAK 这样的数学库流行的一个原因。这些数学库在顺序和并行执行下都被验证是有效的。

230 陷阱：MIPS 指令 addiu（无符号立即数加）会对16位立即数域进行符号扩展。

当我们不关心上溢时，addiu 经常用于将常数和有符号整数相加。由于 MIPS 没有立即数减的指令，所以 MIPS 体系结构设计者决定对该指令的立即数域进行符号扩展，以支持立即数为负数时的需要。

谬误：只有理论数学家才会关心浮点精度。

1994年11月的报纸新闻头条证明了这个观点是错误的（见图3-25）。下面的故事在标题的后面。




图3-25 1994年11月的一些报刊文章，包括《纽约时代》《圣何塞信使报》《旧金山新闻》《信息世界》。Pentium 浮点 bug 甚至成为电视节目"David Letterman Late Show"的"十大新闻"。Intel 最后花了3亿美元来替换掉有 bug 的芯片

Pentium 用一种标准的浮点除算法每步生成多个商位，使用除数的最高几位和被除数猜测商的下面 2 位。这个猜测是利用一个含 -2、-1、0、+1、+2 的查找表。猜测结果和除数相乘，然后从余数中减去，获得新的余数。像不恢复除法一样，如果前面的一个猜测使得余数太大，部分余数将在下面的执行中进行调整。

很明显，Intel 认为有 5 个来自 80486 查找表的元素不会访问到，因此，他们在 Pentium 中优化了此查找表，在一些情况下返回 0 而不是 2。Intel 错了：前 11 位总是正确的，但错误会偶 231
尔在 12 位和 52 位之间出现，或者说十进制下在第 4 位到第 15 位出现。

弗吉尼亚州林奇伯格学院的数学家托马斯·内斯里（Thomas Nicely）在 1994 年 9 月发现了这个 bug。在拨打了 Intel 技术支持电话但没有获得官方回应后，他将他的发现公布在了因特网上。这引发了商业杂志上的一个故事，进一步引发了 Intel 发布了一条声明。Intel 称其为一个“小故障”，仅对理论数学家有影响，对于电子制表软件的用户来说，该缺陷只有 27 000 年才会发生一次。IBM 研究院很快提出反对，指出电子制表软件的用户平均每 24 天就能遇到一次这样的错误。很快，1994 年 12 月 21 日，Intel 发布了如下声明：

我们 Intel 对最近发布的 Pentium 处理器的缺陷处理真诚地道歉。‘Intel Inside’标记的含义是指您的计算机拥有一颗在质量和性能上首屈一指的微处理器。有上千的 Intel 雇员为了实现这个目标而努力工作。但是，没有一款微处理器是完美的，Intel 会继续相信，从技术层面上来讲，任何一个微小的问题都有它的生命期。尽管 Intel 肯定会对 Pentium 处理器的这个版本负责到底，但我们也意识到了用户的关切。我们要解决这种关切。任何消费者在他们计算机生命期的任何时刻，只要他们需要，Intel 会免费为其更换 Pentium 处理器，使浮点除缺陷得到纠正。

分析家估计这次召回花费了 Intel 5 亿美元，Intel 的工程师那年没有拿到圣诞节奖金。

这次事件对每个人来说，都有一些值得思考的地方。如果在 1994 年的 7 月修复了这个 bug 会少花多少钱？修复 Intel 的名声需要多大的代价？一个广泛应用的，像微处理器这样的产品出现 bug，其相关的责任有多么重大？

## 3.10 本章小结

在过去的几十年里，计算机算术在很大程度上被标准化，这极大地增强了程序的可移植性。在当今的每台计算机中，都有二进制补码整数算术，如果它支持浮点，则提供 IEEE 754 二进制浮点算术。

计算机算术和用纸和笔手算的算术不同的地方是受到有限精度的约束。这个限制可能会因为计算中数大于或者小于预先的设定而导致无效操作。这种异常称为“上溢”或“下溢”，可能导致异常、中断或类似于意外的子程序调用。第 4 章和第 5 章更详细地讨论了异常。

浮点算术因为是对实际的数字的近似而增加了挑战性，而且要小心确保所选的计算机数能 232
最接近地表示实际数字。不精确和有限的表达带来的挑战是数值分析领域灵感的部分来源。最近转向**并行性**的趋势使得数值分析再次被关注起来，在顺序计算机上是完全安全的东西，在并行计算机里需要重新考虑，在寻找快速的算法的同时也要有正确的结果。

数据级并行，特称为子字并行，为算术操作密集型（无论是整数或者浮点数操作）性能的提高开辟了一条简单的路径。我们展示了可以使用同时进行 4 个浮点操作的指令来将矩阵乘法加速大约 4 倍。

在本章和第 2 章解释计算机算术时，更多地采用 MIPS 指令集进行描述。容易混淆的一点是这两章讲到的指令和 MIPS 芯片中执行的指令以及 MIPS 汇编器接受的指令之间的关系。图 3-26和图 3-27 试图讲明白这点。

图 3-26 列出了本章和第 2 章中提到的 MIPS 指令。我们将图中左边的指令集称为 MIPS 核

心指令。在右边的指令称为 MIPS 算术核心。图 3-27 的左边是包含了 MIPS 处理器执行的但图 3-26中没有的指令。我们将全部的硬件指令集称为 MIPS-32。在图3-27 的右边是被编译器接受但不属于 MIPS-32 的指令。我们称为伪 MIPS 指令。

| MIPS 核心指令 | 名称 | 格式 |
|---|---|---|
| 加法 | add | R |
| 立即数加法 | addi | I |
| 无符号加法 | addu | R |
| 立即数无符号加法 | addiu | I |
| 减法 | sub | R |
| 无符号减法 | subu | R |
| 与 | AND | R |
| 立即数与 | ANDi | I |
| 或 | OR | R |
| 立即数或 | ORi | I |
| 或非 | NOR | R |
| 逻辑左移 | sll | R |
| 逻辑右移 | srl | R |
| 取立即数的高位 | lui | I |
| 取字 | lw | I |
| 存字 | sw | I |
| 取无符号半字 | lhu | I |
| 存半字 | sh | I |
| 取无符号字节 | lbu | I |
| 存字节 | sb | I |
| 链接取（原子更新） | ll | I |
| 条件存（原子更新） | sc | I |
| 相等则跳转 | beq | I |
| 不相等则跳转 | bne | I |
| 跳转 | j | J |
| 跳转并链接 | jal | J |
| 寄存器跳转 | jr | R |
| 小于则置位 | slt | R |
| 小于立即数则置位 | slti | I |
| 无符号比较、小于则置位 | sltu | R |
| 无符号比较、小于立即数则置位 | sltiu | I |

| MIPS 算术核心 | 名称 | 格式 |
|---|---|---|
| 乘 | mult | R |
| 无符号乘 | multu | R |
| 除 | div | R |
| 无符号除 | divu | R |
| 取自 Hi 寄存器 | mfhi | R |
| 取自 lo 寄存器 | mflo | R |
| 取自系统控制寄存器（EPC） | mfc0 | R |
| 浮点单精度加 | add.s | R |
| 浮点双精度加 | add.d | R |
| 浮点单精度减 | sub.s | R |
| 浮点双精度减 | sub.d | R |
| 浮点单精度乘 | mul.s | R |
| 浮点双精度乘 | mul.d | R |
| 浮点单精度除 | div.s | R |
| 浮点双精度除 | div.d | R |
| 浮点单精度取字 | lwc1 | I |
| 浮点单精度存字 | swc1 | I |
| 浮点双精度取字 | ldc1 | I |
| 浮点双精度存字 | sdc1 | I |
| 浮点真则跳转 | bc1t | I |
| 浮点假则跳转 | bc1f | I |
| 浮点单精度比较 | c.x.s | R |
| (x=eq,neq,lt,le,gt,ge) | | |
| 浮点双精度比较 | c.x.d | R |
| (x=eq,neq,lt,le,gt,ge) | | |

图 3-26 MIPS 指令集。本书集中介绍左列的指令。这个信息也可以在 MIPS 参考数据卡的第 1 列和第 2 列里找到

图 3-28 给出了 SPEC CPU2006 整数和浮点基准测试程序中 MIPS 指令的使用率。所有列出来的指令至少占执行指令的 0.2%。

注意，尽管程序员和编译器编写人员可能为了更多的选项而使用 MIPS-32，MIPS 核心指令主宰了 SPEC CPU2006 整数程序，而算数核心以及算术核心主宰了 SPEC CPU2006 浮点程序，正如下表所列。

| 指令集 | 整数 | 浮点 |
|---|---|---|
| MIPS 核心 | 98% | 31% |
| MIPS 算术核心 | 2% | 66% |
| 剩余的 MIPS-32 | 0% | 3% |

| 保留的 MIPS-32 | 名称 | 格式 |
|---|---|---|
| 异或（s⊕t） | xor | R |
| 异或立即数 | xori | I |
| 算术右移 | sra | R |
| 可变的逻辑左移 | sllv | R |
| 可变的逻辑右移 | srlv | R |
| 可变的算术右移 | srav | R |
| 移至 Hi | mthi | R |
| 移至 Lo | mtlo | R |
| 取半字 | lh | I |
| 取字节 | lb | I |
| 取字的左边（非对齐） | lwl | I |
| 取字的右边（非对齐） | lwr | I |
| 存字的左边（非对齐） | swl | I |
| 存字的右边（非对齐） | swr | I |
| 链接取（原子更新） | ll | I |
| 条件存（原子更新） | sc | I |
| 零则移 | movz | R |
| 非零则移 | movn | R |
| 乘和加（s 或uns） | madds | R |
| 乘和减（s 或uns） | msubs | I |
| 大于等于零则跳转并链接 | bgezal | I |
| 小于零则跳转并链接 | bltzal | I |
| 跳转并链接寄存器 | jalr | R |
| 和零比较并跳转 | bxz | I |
| 与零近似比较并跳转 | bxzl | I |
| (x=lt,le,gt,ge) | | |
| 与寄存器值近似比较并跳转 | bxl | I |
| 与寄存器值比较并自陷 | tx | R |
| 与立即数比较并自陷 | txi | I |
| (x=eq,neq,lt,le,gt,ge) | | |
| 异常返回 | rfe | R |
| 系统调用 | syscall | I |
| 中断（引起异常） | break | I |
| 从浮点移至整数 | mfc1 | R |
| 从整数移至浮点 | mtc1 | R |
| 浮点移（s 或者d） | mov*f* | R |
| 如果零则浮点移（s 或者d） | movz*f* | R |
| 如果非零则浮点移（s 或者d） | movn*f* | R |
| 浮点平方根（s 或者d） | sqrt*f* | R |
| 浮点绝对值（s 或者d） | abs*f* | R |
| 浮点变号（s 或者d） | neg*f* | R |
| 浮点转换（w、s 或者d） | cvt*ff* | R |
| 浮点比较（s 或者d） | c*xf* | R |

| 伪 MIPS | 名称 | 格式 |
|---|---|---|
| 绝对值 | abs | rd,rs |
| 变号（signed 或者unsigned） | negs | rd,rs |
| 旋转左移 | rol | rd,rs,rt |
| 旋转右移 | ror | rd,rs,rt |
| 乘且不检查上溢（signed 或者uns） | muls | rd,rs,rt |
| 乘且检查上溢（signed 或者uns） | mulos | rd,rs,rt |
| 除且检查上溢 | div | rd,rs,rt |
| 除且不检查上溢 | divu | rd,rs,rt |
| 求余（signed 或者uns） | rems | rd,rs,rt |
| 取立即数 | li | rd,imm |
| 取地址 | la | rd,addr |
| 取双字 | ld | rd,addr |
| 存双字 | sd | rd,addr |
| 非对齐取字 | ulw | rd,addr |
| 非对齐存字 | usw | rd,addr |
| 非对齐取半字（signed 或者unsigned） | ulhs | rd,addr |
| 非对齐存半字 | ush | rd,addr |
| 跳转 | b | label |
| 等于零时跳转 | beqz | rs,l. |
| 比较并跳转（signed 或者unsigned） | bxx | rs,rt,l. |
| (x=lt,le,gt,ge) | | |
| 相等则置位 | seq | rd,rs,rt |
| 不相等则置位 | sne | rd,rs,rt |
| 比较并置位（signed 或者unsigned） | sxx | rd,rs,rt |
| (x=lt,le,gt,ge) | | |
| 取数给浮点（s 或者d） | l*f* | rd,addr |
| 浮点数存储（s 或者d） | s*f* | rd,addr |

图 3-27 保留的 MIPS-32 和伪 MIPS 指令集。*f* 代表单（s）或者（d）精度浮点指令，s 代表有符号和无符号（u）版本。MIPS-32 也有浮点指令，包括乘和加/减（madd.*f*/msub.*f*）、向上舍入（ceil.*f*）、截断（trunc.*f*）、舍入（round.*f*）和倒数（recip.*f*）。下划线表示这个字母表示数据类型

| MIPS 核心指令 | 名称 | 整数 | 浮点 |
|---|---|---|---|
| 加法 | add | 0.0% | 0.0% |
| 立即数加法 | addi | 0.0% | 0.0% |
| 无符号加法 | addu | 5.2% | 3.5% |
| 立即数无符号加法 | addiu | 9.0% | 7.3% |
| 无符号减法 | subu | 2.2% | 0.6% |
| 与 | AND | 0.2% | 0.1% |
| 立即数与 | ANDI | 0.7% | 0.2% |
| 或 | OR | 4.0% | 1.2% |
| 立即数或 | ORi | 1.0% | 0.2% |
| 或非 | NOR | 0.4% | 0.2% |
| 逻辑左移 | sll | 4.4% | 1.9% |
| 逻辑右移 | srl | 1.1% | 0.5% |
| 取立即数的高位 | lui | 3.3% | 0.5% |
| 取字 | lw | 18.6% | 5.8% |
| 存字 | sw | 7.6% | 2.0% |
| 取字节 | lbu | 3.7% | 0.1% |
| 存字节 | sb | 0.6% | 0.0% |
| 相等则跳转（a） | beq | 8.6% | 2.2% |
| 不相等则跳转（a） | bne | 8.4% | 1.4% |
| 跳转并链接 | jal | 0.7% | 0.2% |
| 寄存器跳转 | jr | 1.1% | 0.2% |
| 小于则置位 | slt | 9.9% | 2.3% |
| 小于立即数则置位 | slti | 3.1% | 0.3% |
| 无符号比较，小于则置位 | sltu | 3.4% | 0.8% |
| 无符号比较，小于立即数则置位 | sltiu | 1.1% | 0.1% |

| 算术核心+MIPS-32 | 名称 | 整数 | 浮点 |
|---|---|---|---|
| 浮点双精度加 | add.d | 0.0% | 10.6% |
| 浮点双精度减 | sub.d | 0.0% | 4.9% |
| 浮点双精度乘 | mul.d | 0.0% | 15.0% |
| 浮点双精度除 | div.d | 0.0% | 0.2% |
| 浮点单精度加 | add.s | 0.0% | 1.5% |
| 浮点单精度减 | sub.s | 0.0% | 1.8% |
| 浮点单精度乘 | mul.s | 0.0% | 2.4% |
| 浮点单精度除 | div.s | 0.0% | 0.2% |
| 浮点双精度取字 | l.d | 0.0% | 17.5% |
| 浮点双精度存字 | s.d | 0.0% | 4.9% |
| 浮点单精度取字 | l.s | 0.0% | 4.2% |
| 浮点单精度存字 | s.s | 0.0% | 1.1% |
| 浮点真则跳转 | bc1t | 0.0% | 0.2% |
| 浮点假则跳转 | bc1f | 0.0% | 0.2% |
| 浮点双精度比较 | c.x.d | 0.0% | 0.6% |
| 乘 | mul | 0.0% | 0.2% |
| 算术右移 | sra | 0.5% | 0.3% |
| 取半字 | lhu | 1.3% | 0.0% |
| 存半字 | sh | 0.1% | 0.0% |

图 3-28 在 SPEC2006 整数和浮点数中 MIPS 指令的使用频率。表中的所有指令要占到至少 1% 的份额。伪指令在执行前转化为 MIPS-32 指令，所以这里没有出现

本书的剩余部分，我们专注于 MIPS 核心指令——除了乘法、除法以外的整型指令集，以使计算机设计变得易于解释。正如你所看到的，MIPS 核心包含了绝大多数流行的 MIPS 指令；我们认为，理解运行 MIPS 核心的计算机将会给你足够的背景知识，去理解更为复杂的计算机。无论什么指令集或者其大小——MIPS、ARM、x86——永远不要忘记位模式没有内在的含义。同样，位模式可能表示一个带符号整数、无符号整数、浮点数、串、指令，等等。在存储程序计算机中，对位模式的操作决定其含义。

## 3.11 历史观点和拓展阅读

Gresham 法则（"劣币驱逐良币"），对于计算机则是"快的总比慢的，即使快的是错误的"。

——W. Kahan, *1992*

绝不要放弃，绝不要，永远，永远，永远，不要放弃——任何事情，不论大小——绝不要放弃。

——温斯顿·丘吉尔，在 *Harrow* 学校的演讲，*1941*

233 – 236

本节回溯到冯·诺依曼来纵览浮点历史，包括有争议的 IEEE 标准的令人惊讶的成就，以及 x86 的 80 位浮点堆栈结构的基本原理。见配套网站上 3.11 节。

## 3.12 练习题

**3.1** [5] <3.2>5ED4 - 07A4 是无符号 16 位十六进制数时如何表示？结果必须使用 16 进制表示。

**3.2** [5] <3.2>5ED4 - 07A4 是带符号 16 位十六进制数且以符号 - 数值形式存放时如何表示？结果必须使用 16 进制表示。

**3.3** [10] <3.2>将 5ED4 转换成二进制数。使用十六进制表示计算机中的数值很具有吸引力的原因是什么？

**3.4** [5] <3.2>写出 4365 - 3412 使用无符号 12 位八进制数的表示形式。结果必须使用八进制表示。

**3.5** [5] <3.2>写出 4365 - 3412 使用带符号 12 位八进制数且以符号 - 数值形式存放的表示形式。结果必须使用八进制表示。

**3.6** [5] <3.2>假定 185 和 122 是无符号 8 位十进制整数。计算 185 - 122。是否有上溢或者下溢？

**3.7** [5] <3.2>假定 185 和 122 是带符号 8 位十进制整数且以符号 - 数值形式存放。计算 185 + 122。是否有上溢或者下溢？

**3.8** [5] <3.2>假定 185 和 122 是带符号 8 位十进制整数且以符号 - 数值形式存放。计算 185 - 122。是否有上溢或者下溢？

**3.9** [10] <3.2>假定 151 和 214 是带符号 8 位十进制整数且以补码形式存放。使用饱和算术计算 151 + 214。结果必须使用十进制。

**3.10** [10] <3.2>假定 151 和 214 是带符号 8 位十进制整数且以补码形式存放。使用饱和算术计算 151 - 214。结果必须使用十进制。

**3.11** [10] <3.2>假定 151 和 214 是无符号 8 位十进制数。使用饱和算术计算 151 + 214。结果必须使用十进制。

**3.12** [20] <3.3>使用图 3-3 所示的硬件描述计算八进制无符号 6 位整数 62 和 12 的乘积，并给出一个类似于图 3-6 中的表。必须给出每个步骤中每个寄存器的内容。 237

**3.13** [20] <3.3>使用与图 3-6 相同的一张表，使用图 3-5 所示的硬件描述计算十六进制无符号 8 位整数 62 和 12 的乘积。必须给出每个步骤中每个寄存器的内容。

**3.14** [10] <3.3>如果一个整数是 8 位宽，且每个步骤的操作需要 4 个时间单位，使用图 3-3 和图 3-4 的方法计算执行一次乘法必需的时间。假定在步骤 1a 中，无论是否加了被乘数还是加 0，加法都要执行。另外假设寄存器已经初始化（只需要计算执行乘法循环本身所需要的时间）。如果是在硬件中执行，对被乘数和乘数的移位可以同时进行；如果是在软件中执行，则会一个做完再做下一个。对每种情况都给出解答。

**3.15** [10] <3.3>计算采用书中的方法（31 个垂直的加法堆栈）来执行乘法所需要的时间。设整数位宽是 8，一个加法需 4 个单位时间。

**3.16** [20] <3.3>计算采用图 3-7 中的方法来执行乘法所需要的时间。设整数位宽是 8，一个加法需 4 个单位时间。

**3.17** [20] <3.3>正如书中讨论的，一种增强性能的办法是做一次移位和加法来代替一次实际的乘法。例如，因为 9×6 可以写成（2×2×2+1）×6，所以我们可以通过将 6 左移 3 次再加上 6 来计算 9×6。给出用移位和加/减法来计算 0×33×0×55 的最好的方法。假设输入都是 8 位无符号整数。

**3.18** [20] <3.4>使用图 3-8 中的硬件结构计算 74 除以 21，并给出一个类似于图 3-10 中的表。你需要给出每一步中各个寄存器的值。假设输入都是 6 位无符号整数。

**3.19** [30] <3.4>用图 3-11 中的硬件结构计算 74 除以 21，并给出一个类似于图 3-10 中的表。你需要给出每一步中各个寄存器的值。假设 A 和 B 都是 6 位无符号整数。这个算法使用一个和图 3-9 中稍微不同的方法。这个算法你可能会认为很难，做一次或者两次试验，或者去网上寻找办法来让它正确工作。（提示：一种可能的解决方案是利用图 3-11 中暗示的那个余数寄存器既可右移也可左移的事实。）

**3.20** [5] <3.5>如果是补码整数，则这些位模式 0X0C000000 代表的十进制是多少？如果是无符号整

数呢?

**3.21** [10] <3.5> 如果位模式 0x0C000000 放在指令寄存器中，那么将执行什么 MIPS 指令?

**3.22** [10] <3.5> 如果是浮点数，则位模式 0x0C000000 代表的十进制数是多少? 使用 IEEE 754 标准。

238

**3.23** [10] <3.5> 写出十进制数 63.25 的二进制表达。设采用 IEEE 754 单精度格式。

**3.24** [10] <3.5> 写出十进制数 63.25 的二进制表达。设采用 IEEE 754 双精度格式。

**3.25** [10] <3.5> 写出十进制数 63.25 的二进制表达。设采用 IBM 单精度格式存储（基数为 16 而不是 2，有 7 位指数位）。

**3.26** [20] <3.5> 写出 $-1.5625 \times 10^{-1}$ 的二进制位模式。设采用一种类似 DEC PDP-8 使用的格式（左 12 位是以补码形式存储的指数，而右 24 位是以补码形式存储的尾数。）没有隐含 1。同 IEEE 754 标准的单精度和双精度比较，评估这个 36 位位模式的范围和精确度。

**3.27** [20] <3.5> IEEE 754-2008 包含一种“半精度”格式，只有 16 位宽。最左边仍为符号位，指数有 5 位宽且以余 -16（excess -16）的形式存储。尾数有 10 位宽。具有隐含 1。写出 $-1.5625 \times 10^{-1}$ 的这种格式的二进制位模式。同 IEEE 754 标准的单精度比较，评估这个 16 位位模式的范围和精确度。

**3.28** [20] <3.5> 惠普 2114、2115 和 2116 采用这样一种格式，其最左边 16 位以补码形式存储着尾数，紧跟着在另一个 16 位域里，左边 8 位是尾数的扩展（使尾数达到 24 位宽），右边 8 位表示指数。然而，作为一种有趣的变更，指数以符号 - 数值的形式存储且符号位在最右端! 写出 $-1.5625 \times 10^{-1}$ 的这种格式的二进制位模式。没有隐含 1。同 IEEE 754 标准的单精度比较，评估这个 32 位位模式的范围和精确度。

**3.29** [20] <3.5> 手算 $2.6125 \times 10^{1}$ 和 $4.150\,390\,625 \times 10^{-1}$ 的和，设 A 和 B 以练习题 3.27 中提到的 16 位半精度格式存储。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤。

**3.30** [30] <3.5> 手算 $-8.054\,687\,5 \times 10^{3}$ 和 $-1.799\,316\,406\,25 \times 10^{-1}$ 的积，设 A 和 B 以练习题 3.27 中提到的 16 位半精度格式存储。假设 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤；然而，作为书中已经做过的例子，你可以以人们可读的格式来做这个乘法，而不用练习题 3.12 到练习题 3.14 中描述的技术。注明是否上溢或者下溢。分别以练习题 3.27 中的 16 位浮点模式和十进制数写出你的答案。你的结果精确程度如何? 和你用计算器取得的结果相比呢?

239

**3.31** [30] <3.5> 手算 $8.625 \times 10^{1}$ 除以 $-4.875 \times 10^{2}$。给出必要的步骤。假设有 1 个保护位、1 个舍入位和 1 个粘贴位，并在必要时使用。以练习题 3.27 中的 16 位浮点格式和十进制格式给出最终的结果，并比较十进制结果和用计算器得到的结果。

**3.32** [20] <3.9> 手算 $(3.984\,375 \times 10^{-1} + 3.437\,5 \times 10^{-1}) + 1.771 \times 10^{3}$，设每个数值都以练习题 3.27 中提到的 16 位半精度格式存储（书中也有介绍）。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤，并以 16 位浮点格式和十进制格式给出你的答案。

**3.33** [20] <3.9> 手算 $3.984\,375 \times 10^{-1} + (3.437\,5 \times 10^{-1} + 1.771 \times 10^{3})$，设每个数值都以练习题 3.27 中提到的 16 位半精度格式存储（书中也有介绍）。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤，并以 16 位浮点格式和十进制格式给出你的答案。

**3.34** [10] <3.9> 根据练习题 3.32 和练习题 3.33 的结果，计算 $(3.984\,375 \times 10^{-1} + 3.437\,5 \times 10^{-1}) + 1.771 \times 10^{3} = 3.984\,375 \times 10^{-1} + (3.437\,5 \times 10^{-1} + 1.771 \times 10^{3})$ 是否成立?

**3.35** [30] <3.9> 手算 $(3.417\,968\,75 \times 10^{-1} \times 6.347\,656\,25 \times 10^{-1}) \times 1.056\,25 \times 10^{2}$，设每个值都以练习题 3.27 中提到的 16 位半精度格式存储（书中也有介绍）。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤，并以 16 位浮点格式和十进制格式给出你的答案。

**3.36** [30] <3.9> 手算 $3.417\,968\,75 \times 10^{-2} \times (6.347\,656\,25 \times 10^{-3} \times 1.056\,25 \times 10^{2})$，设每个值都以练习题 3.27 中提到的 16 位半精度格式存储（书中也有介绍）。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤，并以 16 位浮点格式和十进制格式给出你的答案。

**3.37** [10] <3.9> 根据练习题 3.35 和练习题 3.36 的结果，计算 $(3.417\,968\,75 \times 10^{-2} \times 6.347\,656\,25 \times 10^{-3}) \times 1.056\,25 \times 10^{2} = 3.417\,968\,75 \times 10^{-2} \times (6.347\,656\,25 \times 10^{-3} \times 1.056\,25 \times 10^{2})$ 是否成立？

**3.38** [30] <3.9> 手算 $1.666\,015\,625 \times 10^{0} \times (1.976\,0 \times 10^{4} + (-1.9744) \times 10^{4})$，设每个值都以练习题 3.27 中提到的 16 位半精度格式存储（书中也有介绍）。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤，并以 16 位浮点格式和十进制格式给出你的答案。

**3.39** [30] <3.9> 手算 $(1.666\,015\,625 \times 10^{0} \times 1.976\,0 \times 10^{4}) + (1.666\,015\,625 \times 10^{0} \times (-1.9744) \times 10^{4})$，设每个值都以练习题 3.27 中提到的 16 位半精度格式存储（书中也有介绍）。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤，并以 16 位浮点格式和十进制格式给出你的答案。 [240]

**3.40** [10] <3.9> 根据练习题 3.38 和练习题 3.39 的结果，计算 $(1.666\,015\,625 \times 10^{0} \times 1.976\,0 \times 10^{4}) + (1.666\,015\,625 \times 10^{0} \times (-1.974\,4) \times 10^{4}) = 1.666\,015\,625 \times 10^{0} \times (1.976\,0 \times 10^{4} + (-1.9744) \times 10^{4})$ 是否成立？

**3.41** [10] <3.5> 按照 IEEE 754 浮点格式，写出 -1/4 的位模式。你能精确表示 -1/4 吗？

**3.42** [10] <3.5> 如果将 -1/4 自加 4 次得到多少？$-1/4 \times 4$ 是多少？它们相同吗？它们应该是多少？

**3.43** [10] <3.5> 写出数值 1/3 的尾数的位模式，其浮点格式采用二进制编码的尾数。假设有 24 位，并且不需要进行规格化。这种表达精确吗？

**3.44** [10] <3.5> 写出尾数的位模式，其浮点格式采用 BCD 编码（基 10）而不是基 2 的尾数。假设有 24 位，并且不需要进行规格化。这种表达精确吗？

**3.45** [10] <3.5> 写出尾数的位模式，其浮点格式采用基 15 编码而不是基 2 的尾数。（基 16 编码使用符号 0 ~ 9 和 A ~ F。基 15 编码使用 0 ~ 9 和 A ~ $F_{15}$。）假设有 24 位，并且不需要进行规格化。这种表达精确吗？

**3.46** [20] <3.5> 写出尾数的位模式，其浮点格式采用基 30 编码而不是基 2 的尾数。（基 16 编码使用符号 0 ~ 9 和 A ~ F。基 30 编码使用 0 ~ 9 和 A ~ $T_{30}$。）假设有 20 位，并且不需要进行规格化。这种表达精确吗？

**3.47** [45] <3.6, 3.7> 下面的 C 代码实现了一个 4 阶 FIR 滤波器，其输入为数组 sig_ in。假设所有的数组元素为 16 位定点数。

```
for (i=3;i<125;i++)
sig_out[i]=sig_in[i-3] * f[0]+sig_in[i-2] * f[1] +
  +sig_in[i-1] * f[2]+sig_in[i] * f[3];
```

假设你要面向一个具有 SIMD 指令集且有 128 位寄存器的处理器，使用汇编语言对该代码进行优化。在不知道指令集细节的情况下，简要介绍一下你该怎样实现该代码，最大限度地使用子字并行操作，并且使寄存器和存储器间的数据传送量最少。指明你对使用的指令集的假设。

## 小测验答案

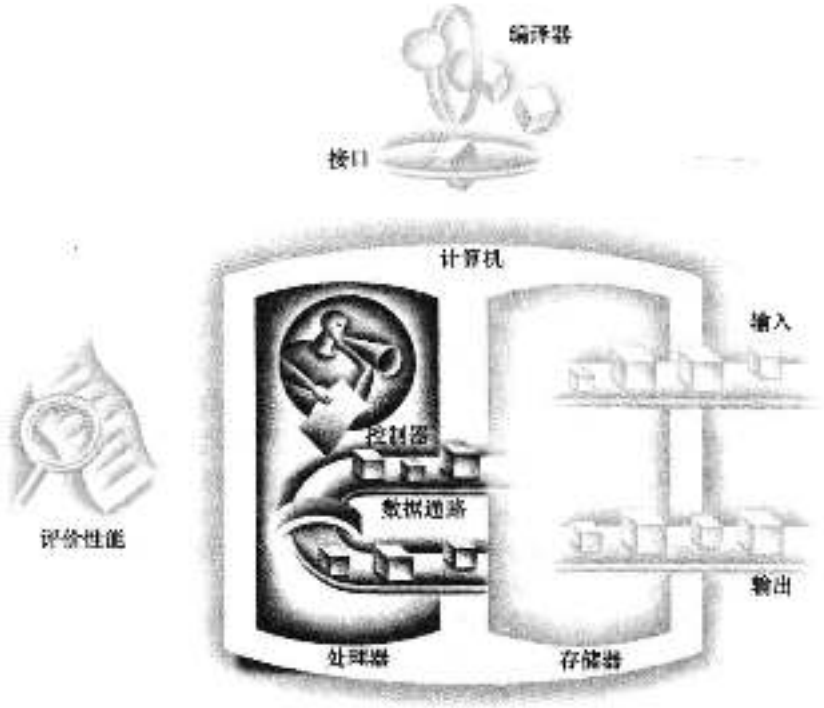
**3.2** 2

**3.5** 3 [241]

第4章

Computer Organization and Design: The Hardware/Software Interface

# 处 理 器




计算机的5个经典部件

在关键问题上，没有什么细节是小事。

法国谚语

## 4.1 引言

在第1章中，我们看到一台计算机的性能由三个关键因素决定：指令数目、时钟周期长度和每条指令所需时钟周期数（CPI）。我们在第2章阐明编译器和指令集决定了一个程序所需的指令数目。而处理器的实现方式则决定了时钟周期长度和CPI。在本章中，我们为MIPS指令集的两种不同实现方式分别建立数据通路和控制单元。

本章包含了实现一个处理器所需的原理与技术。先从一个高度抽象和简化的概述开始，再建立数据通路，并进一步构建一个简单的处理器以实现像MIPS这样的指令集。本章的其余部分还包括：一个更实际的流水化的MIPS实现，以及实现更复杂的指令集（如x86）时所需要的概念。

对理解指令的高层解释及其对程序性能的影响感兴趣的读者，4.5节给出了流水线的基本概念。4.10节介绍了最近的趋势。4.11节描述了最新的Intel Core i7和ARM Cortex-A8体系结构。4.12节展示了如何通过指令级并行将3.8节的矩阵乘法性能提高两倍以上。这几节为在高

层次理解流水线概念提供了必要的背景知识。

对希望能更深入地理解处理器及其性能的读者，4.3 节、4.4 节和 4.6 节很有用。对如何建立一个处理器感兴趣的读者可以阅读 4.2 节、4.7 节、4.8 节和 4.9 节。对现代硬件设计感兴趣的读者，可以参考 4.13 节，其中介绍了实现硬件时使用的硬件设计语言与 CAD 工具，以及如何使用硬件设计语言来描述一个流水化的实现。它对于理解流水化硬件执行的细节有很大帮助。

## 一个基本的 MIPS 实现

我们将要设计的实现方式包含了 MIPS 指令集的一个核心子集：

- 存储器访问指令：取字（lw）和存字（sw）。
- 算术逻辑指令：加法（add）、减法（sub）、与运算（AND）、或运算（OR）和小于则设置（slt）。
- 分支指令：相等则分支（beq）和跳转（j），我们放到最后实现。

242 ~ 244

这个子集没有包含所有的整数指令（如不包含乘、除指令和移位指令），也没有包含任何浮点指令。然而，使用该子集可以说明在建立数据通路和控制单元时的关键原理，并可以在此基础之上实现其他指令。

在学习此实现方式时，我们将会有机会看到指令集如何决定实现方式的多个方面，以及实现策略如何影响计算机的时钟速度和 CPI。在第 1 章介绍的许多关键设计原理，如“简单源于规整”的指导思想，都将体现出来。并且，本章中用于实现 MIPS 子集的大多数概念与很多计算机的基本构造思想是一致的，包括从高性能服务器到通用微处理器、嵌入式处理器等各式各样的计算机。

### 实现方式概述

在第 2 章中，我们学习了 MIPS 的核心指令，包括整数算术逻辑指令、存储访问指令及分支指令。这些指令的实现过程大致相同，而与具体的指令类型无关。实现每条指令的前两步是一样的：

1）程序计数器（PC）指向指令所在的存储单元，并从中取出指令。

2）通过指令字段内容，选择读取一个或两个寄存器。对于取字指令，只需读取一个寄存器，而其他大多数指令要求读取两个寄存器。

这两步之后，为完成指令而进行的步骤则取决于具体的指令类型。幸运的是，对三种指令类型（存储访问、算术逻辑、分支）的每一种而言，其动作大致相同，与具体指令无关。MIPS 指令集的简洁和规整使许多指令的执行很相似，因而简化了实现过程。

例如，除跳转指令外的所有指令在读取寄存器后，都要使用算术逻辑单元（ALU）。存储访问指令用 ALU 计算地址，算术逻辑指令用 ALU 执行运算，分支指令用 ALU 进行比较。在使用 ALU 之后，完成不同指令所需的动作就有所不同了。存储访问指令需要访问内存以便读取和存储数据。算术逻辑指令或取数指令将来自 ALU 或存储器的数据写入寄存器。对分支指令，我们需要根据比较的结果决定是否改变下一条指令地址；如果不修改下一条指令地址，则下一条指令地址默认是当前指令地址 +4。

图 4-1 给出了一种 MIPS 实现的抽象视图，图中主要描述了不同的功能单元及其互连关系。尽管该图给出了绝大多数数据在处理器中的流动方式，但它仍然忽略了指令执行过程中的两个重要方面。

245

首先，在图 4-1 中的许多位置，某个单元的数据可能来自于两个不同的单元。例如，写入

PC 的值可能来自两个加法器中的一个，写入寄存器堆的数据可能来自 ALU 或数据存储器，ALU 的第二个输入可能来自寄存器或指令中的立即数字段。实际上，不能简单地直接将这些数据线连在一起，必须增加一个逻辑单元用以从不同的数据来源中选择一个送给目标单元。这个选择过程通常是由一个叫多选器（multiplexor）的逻辑单元完成的，尽管该单元叫数据选择器可能更合适。附录 B 详细描述了多选器根据控制信号选择不同输入的过程。控制信号主要由当前执行指令中包含的信息决定。

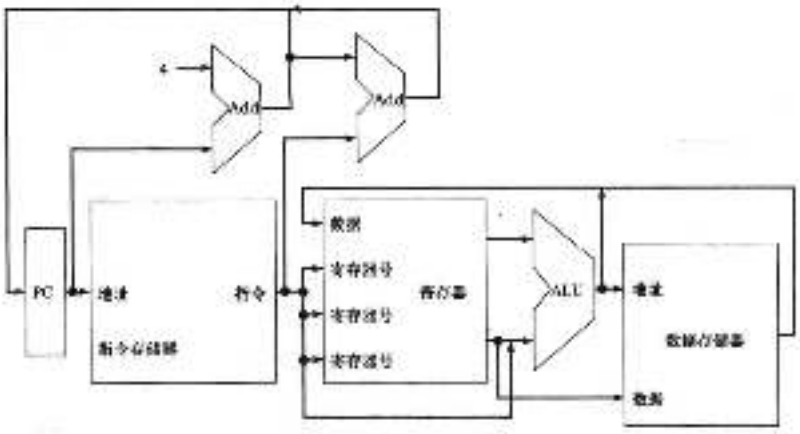



图 4-1 一个 MIPS 子集实现的抽象视图，描述了主要功能单元及其连接。所有指令都开始于使用程序计数器获得指令在指令存储器中的地址。在取到指令后，指令所使用的寄存器操作数由指令中的对应字段决定。在取到寄存器操作数之后，可以用来计算存储器地址（对于存取类指令），或者计算算术运算结果（对于算数算术逻辑类指令），或者进行比较（对于分支类指令）。如果是算术逻辑类指令，ALU 的结果必须写回寄存器；如果是存取类指令，ALU 的结果可作为读写存储器的地址。ALU 或存储器的结果可写回寄存器堆。分支操作需要使用 ALU 的输出来决定下一个指令地址，下一个指令地址可能来自 ALU（在 ALU 中完成 PC 值与分支偏移量相加），也可能来自加法器（当前 PC 值加 4）。连接功能单元的粗线表示总线，其中包含多个信号。箭头用来指示信息流动的方向。因为信号线在图上可能相交，所以在相交信号线确实相连时用一个黑点来表示

246

其次，图 4-1 中的许多单元的控制依赖于当前执行指令的类型。例如，存取指令读写数据存储器，取数指令和算术逻辑指令写入寄存器堆。很显然，ALU 根据不同的指令执行不同的操作。（附录 B 给出了 ALU 的设计细节）。类似于多选器，这些操作都由控制信号确定，而控制信号是由指令的某些字段所决定的。

图 4-2 在图 4-1 的基础上增加了三个必需的多选器和主要功能单元的控制信号。图中还增加了一个控制单元（control unit），它以指令为输入，决定功能单元和两个多选器的控制信号。第三个多选器用来决定是将 PC + 4 还是分支目的地址写入 PC，该多选器在执行 beq 指令时，根据 ALU 进行比较时设置的 Zero 标志位选择写入 PC 的数值。MIPS 指令集的简单与规整使得只需简单的译码即可生成控制信号。

在本章后面的部分，我们将会为图 4-2 加入更多的细节，包括更多的功能单元和单元间的连接，并增强控制单元功能以控制不同类型的指令执行过程。4.3 节和 4.4 节描述了每条指令使用一个时钟周期的简单实现方式，它将遵循图 4-1 和图 4-2 的一般形式。在第一个设计中，每条指令在一个时钟沿开始执行，然后在下一个时钟沿完成执行。

尽管这种方法易于理解，但是并不实际，因为时钟周期必须设置为足够容纳执行时间最长的指令。在设计完这种简单计算机的控制后，我们将会讨论一种流水的实现方式及其带来的复杂性和异常。

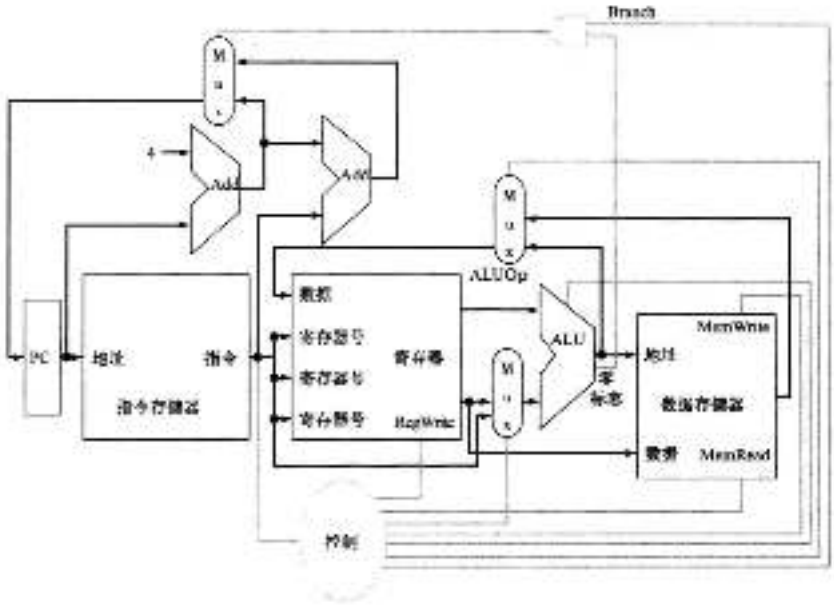



图 4-2 一个 MIPS 子集的基本实现，其中包含了必要的多选器和控制信号。最上面的多选器控制写入 PC 的值（PC+4 或分支目的地址）。该多选器由一个门控制，该门将 ALU 的零输出与一个指示是否为分支指令的信号相“与”。中间输出到寄存器堆的多选器，用来选择将被写入寄存器堆中的是 ALU 的输出（算术逻辑指令时）还是数据存储器的输出（取数指令时）。最后，最下面的多选器决定 ALU 的第二个输入是来自寄存器堆（算术逻辑指令或分支指令时）还是指令的偏移量字段（存取指令时）。新增的控制信号直接控制 ALU 的操作、数据存储器的读写和寄存器堆的写入等。控制信号在图中用灰色线标识出来

247

**小测验**

图 4-1 和图 4-2 包含了本章开始给出的计算机五大经典部件中的那几个？

## 4.2 逻辑设计的一般方法

在考虑计算机的设计时，必须决定机器的逻辑实现以及机器时钟。本节将继续讨论一些本章经常用到的数字逻辑的关键思想。如果你缺乏数字逻辑方面的知识，那么在继续学习之前，阅读附录 B 将有所帮助。

MIPS 实现中的数据通路功能部件包括两种不同的逻辑单元：处理数据值的单元和存储状态的单元。处理数据值的单元都是**组合单元**（combinational element），它们的输出只取决于当前的输入。当输入相同时，组合单元产生的输出也相同。出现在图 4-1 中并在附录 B 中详细论述的 ALU 就是组合单元。因为其没有内部存储功能，当给定一组输入时总是产生同样的输出。

设计中的其他单元是非组合单元，而是包含状态的。如果一个单元带有内部存储功能，它就包含状态，称之为**状态单元**（state element），因为关机后重启计算机，通过恢复状态单元的原值，计算机可继续运行如同没有断电一样。也就是说，这些状态单元完全描述了计算机的状态。图 4-1 中的指令存储器、数据存储器和寄存器都是状态单元。

- 组合单元：一个操作单元，如与门或 ALU。
- 状态单元：一个存储单元，如寄存器或存储器。

248

一个状态单元至少有两个输入和一个输出。两个必要的输入为要写入单元的数据值和决定

何时写入的时钟信号。状态单元的输出提供了在前一个时钟信号写入单元的数据值。例如，逻辑上最简单的一种状态单元是D触发器（参见附录B），它有两个输入（一个数据值和一个时钟）和一个输出。除了触发器，MIPS的实现中还用了另外两种状态单元：存储器和寄存器，这些在图4-1中都已给出。时钟用于决定状态单元何时被写入。状态单元随时可读。

包含状态的逻辑部件又被称为时序的（sequential），因为它们的输出由输入和内部状态共同决定。例如，代表寄存器的功能单元的输出取决于所提供的寄存器号和以前写入寄存器的内容。组合单元和时序单元的有关操作及它们的结构都在附录B中有详细论述。

### 时钟方法

**时钟方法**（clocking methodology）规定了信号可以读出和写入的时间。规定信号读写的时间非常重要，因为若一个信号同时被读出和写入，则所读出的信号可能是写入前的值，也可能是新写入的值，甚至是两者的混合。显然，这种不确定性在计算机的设计中是不允许的。时钟方法即是为避免这种情况而提出的。

为简单起见，我们假定采用**边沿触发的时钟**（edge-triggered clocking）方法，即在时序逻辑单元中存储的所有值都只允许在时钟跳变的边沿时改变（见图4-3）。因为只有状态单元能存储数据值，所有的组合逻辑都必须从状态单元集合接收输入，并将输出写入状态单元集合中。其输入为之前某时钟周期写入的数据，其输出可供之后某时钟周期使用。

- 时钟方法：用来确定数据相对于时钟何时稳定和有效的方法。
- 边沿触发的时钟：一种所有的状态改变发生于时钟沿的时钟机制。

249




图4-3 组合逻辑、状态单元和时钟周期的关系。在一个同步的数字系统中，时钟信号决定了数值何时写入状态单元。在有效的时钟边沿导致状态变化之前，状态单元的输入信号必须达到稳定（也就是说，状态单元的值保持不变，直到时钟沿到来）。本章假定所有状态单元（包括存储器）都是上升沿触发的，即这些信号都是在时钟的上升沿发生变化。

图4-3描述了一个组合逻辑单元及与其相连的两个状态单元。组合逻辑单元的操作在一个时钟周期内完成：所有信号在一个时钟周期内从状态单元1经组合逻辑到达状态单元2，信号到达状态单元2所需的时间决定了时钟周期的长度。

为简单起见，若某状态单元在每个有效的时钟边沿都进行写入操作，则可忽略写**控制信号**（control signal）。相反，若某状态单元不是每个周期都进行修改，那么它就需要一个写控制信号。写控制信号和时钟信号都是输入信号，只有时钟边沿到来并且当写控制信号有效时，状态单元才改变状态。

我们将使用术语**有效**（asserted）表示信号为逻辑高，**无效**（deasserted）表示信号为逻辑低。另外，我们之所以要使用术语“有效”和“无效”，是因为在进行硬件实现时，数字1有时表示逻辑高，有时表示逻辑低。

- 控制信号：用来决定多路器选择或指示功能单元操作的信号；它与数据信号相对应，数据信号包含由功能单元操作的信息。
- 有效：信号为逻辑高或真。
- 无效：信号为逻辑低或假。

使用如图 4-4 所示的一种边沿触发的方法，可以在一个时钟周期内读出一个寄存器的值，然后使之经过一些组合逻辑，同时将新值写入该寄存器。选择在时钟的上升沿（从低到高）还是下降沿（从高到低）进行写操作无关紧要，因为组合逻辑的输入只有在所规定的时钟边沿才可能发生变化。本书我们使用时钟的上升沿。这种边沿触发时钟方法在一个时钟周期内不会出现反馈，图 4-4 中的逻辑可以正确地工作。在附录 B 中，还介绍了其他的一些时序约束（如建立和保持时间）和一些时序方法。




图 4-4 一种边沿触发方法，支持状态单元在同一个时钟周期内同时读写而不会因竞争而出现中间数据。当然，必须保证时钟周期足够长，以使得当有效的时钟边沿到来时输入已经稳定。状态单元的改变由时钟边沿触发，所以不可能在一个时钟周期之内出现反馈。如果有反馈，这个设计就不能正常工作。本章和下一章的设计都采用边沿触发的时钟方法，结构与本图类似。

对 32 位 MIPS 体系结构而言，几乎所有这些状态和逻辑单元的输入和输出都为 32 位，因为处理器处理的大多数数据的宽度为 32 位。若某单元的输入或输出不是 32 位，我们会特别指出。图示中用粗线表示总线，即宽度为 1 位以上的信号。有时要把几根总线合起来构成更宽的总线，例如可能将两根 16 位总线合成一根 32 位总线。在这种情况下，总线标注将作出相应说明。另外还加上箭头以指明单元间数据传输的方向。最后，灰色线表示的控制信号将其与数据信号区分开来，两者的差别将随本章的进展愈趋明显。 250

**小测验**

是非判断：由于寄存器堆在一个时钟周期内既要写入又要读出，所以任何使用边沿触发方式写入的 MIPS 数据通路中必须包含至少一份寄存器堆的备份。

**精解** 还有一种 64 位版本的 MIPS 系统结构，其中绝大多数数据通路都是 64 位宽。

## 4.3 建立数据通路

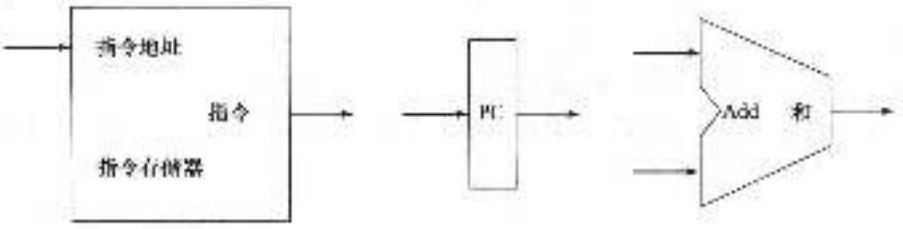
设计数据通路比较合理的方法是首先分析执行每种 MIPS 指令时需要哪些主要部件。下面先来看看每条指令需要什么**数据通路部件**（datapath element）。在指出数据通路部件的同时，我们也会指出它们的控制信号。我们将从自底向上开始，使用抽象来对此进行说明。

图 4-5a 给出了我们需要的第一个元素：一个存储单元，它用于存储程序的指令，并在给定地址时提供指令。图 4-5b 展示了**程序计数器**（program counter，PC），在第 2 章曾经出现过，用于保存当前指令的地址。最后，我们需要一个加法器来计算 PC 的值以指向下条指令的地址。这个加法器是一个组合单元，可以用附录 B 中设计的 ALU 实现，只需将其中的控制信号设为总是进行加法操作即可。如图 4-5c 所示，我们将给这样的 ALU 加上“Add”标记，以表明它成为了一个加法器而不能再进行其他 ALU 操作。

- 数据通路部件：一个用来操作或保存处理器中数据的单元。在 MIPS 实现中，数据通路部件包括指令存储器、数据存储器、寄存器堆、ALU 和加法器。
- 程序计数器：存放下一条将要被执行指令的地址的寄存器。

要执行任何一条指令，首先要从存储单元中将指令取出。为准备执行下一条指令，也必须增加程序计数器使其指向下一条指令，即向后移动 4 字节。此时的数据通路如图 4-6 所示，使

同了图4-5中的3个部件，它可以取指令并能自增PC以获得下一条指令的地址。




a）指令存储器　　b）程序计数器　　c）加法器

图4-5　存取指令需要的两个状态单元，以及计算下一条指令地址所需要的加法器。两个状态单元分别是指令存储器和程序计数器。因为数据通路没有写指令，所以指令存储器只提供读访问。因为指令存储器是只读的，我们将它视为组合逻辑：任意时刻的输出都反映了输入地址处的内容，而不需要读控制信号。（在读取程序时需要写入指令存储器，但是这很容易实现，所以为了简单起见我们将其忽略。）程序计数器是一个32位的寄存器，它在每个时钟周期末都会被写入，所以不需要写控制信号。加法器采用只进行加法的ALU，它将输入的两个32位数相加，将结果输出

现在讨论R型指令（参见图2-20）。这类指令读两个寄存器，对它们的内容进行ALU操作，再写回结果。我们将这类指令称为R型指令或算术逻辑指令（因为它们执行算术或逻辑运算）。这个指令集合包括第2章介绍的add、sub、AND、OR和slt指令。回忆一下，此类指令的典型形式是add $t1, $t2, $t3，它将读取$t2和$t3，并将结果写回$t1。

251

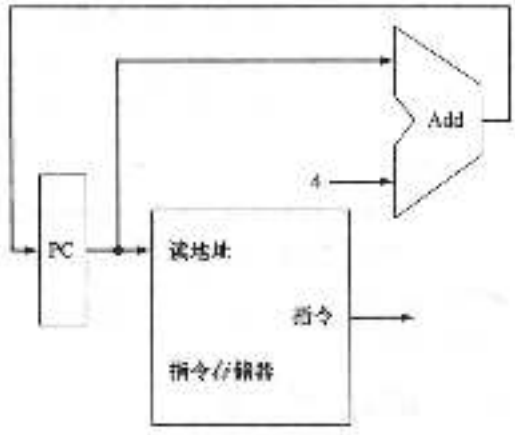



图4-6　用于取指和程序计数器自增的数据通路部分。取出的指令被数据通路的其他部分使用

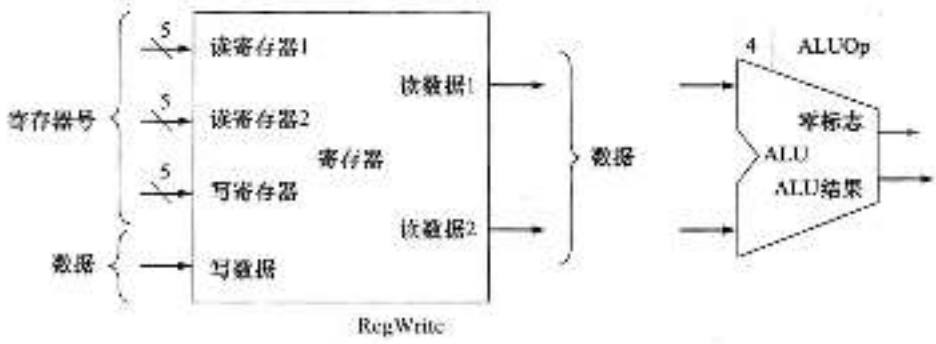
处理器的32个通用寄存器位于一个叫作**寄存器堆**（register file）的结构中。寄存器堆即寄存器集合，其中的寄存器都可通过指定相应的寄存器号来进行读写。寄存器堆包含了计算机的寄存器状态。另外，还需要一个ALU来对从寄存器读出的数值进行运算。

➲　寄存器堆：包含一系列寄存器的状态单元，可以通过提供寄存器号进行读写。

由于R型指令有3个寄存器操作数，每条指令都要从寄存器堆读出两个数据字，再写入一个数据字。为读出一个数据字，寄存器堆需要输入一个要读的寄存器号和一个从寄存器堆读出结果的输出指示。为写入一个数据字，寄存器堆要有两个输入：一个提供要写的寄存器号（register number），另一个提供要写的数据（data）。寄存器堆总是根据输入的寄存器号输出相应的寄存器内容，而写操作由写控制信号控制，在写操作发生的时钟边沿，写控制信号必须是有效的。这样，我们一共需要4个输入（3个寄存器号和1个数据）和两个输出（两个数据），如图4-7a所示。输入的寄存器号为5位，可指示32个寄存器中的某一个（$32 = 2^5$），而一条数据输入总线和两条数据输出总线宽度均为32位。

图4-7b所示为ALU，该ALU有两个32位输入、一个32位输出，还有一个1位输出指示

其结果是否为0。ALU的4位控制信号在附录B中有详细的描述。在需要了解如何设置ALU控制信号时，我们将进行简要的回顾。 252




a) 寄存器堆 b) ALU

图4-7 实现R型指令的ALU操作所需的两个单元——寄存器堆和ALU。寄存器堆包括了所有的寄存器，有两个读端口和一个写端口。多端口寄存器堆的设计在附录B的B.8节讨论。寄存器堆的读输出总是对应于读寄存器号，不需要其他的控制信号。但是写寄存器必须明确使能写控制信号。注意写操作是边沿触发的，所以所有的写操作的输入（要写的内容、寄存器号、写控制信号）必须在时钟边沿有效。因为寄存器堆的写入是边沿触发的，故可以在同一时钟周期内读出和写入同一寄存器：读操作将读出以前写入的内容，而写入的内容在下一时钟周期才可读。寄存器号的输入都是5位的，数据线为32位。若采用附录B中的ALU设计，则ALU的操作可由4位ALU操作信号控制。我们使用ALU的零检测输出信号实现分支指令。溢出信号在4.9节讲述异常时才会用到，在此之前我们先忽略它

下面考虑MIPS的存取指令，其一般形式为：`lw $t1,offset_value($t2)`或`sw $t1, offset_value($t2)`。在这类指令中，通过将基址寄存器$t2的内容与指令中的16位带符号偏移地址相加，得到存储器地址。如果是存储指令，要从寄存器$t1中读出要存储的数据；如果是取数指令，则要将从存储器中读出的数据存入指定的寄存器$t1中。所以，图4-7中的寄存器堆和ALU都会用到。

另外，还需要一个单元将16位的偏移地址**符号扩展**（sign-extend）为32位的带符号值，以及一个保存读出或写入数据的存储单元。数据存储单元在存储指令时被写入，所以它有读写控制信号、地址输入和写入存储器的数据输入。

⇒ 符号扩展：为增加数据项的长度，将原数据项的最高位复制到新数据项多出来的高位。

beq指令有3个操作数，其中两个为寄存器，用于比较是否相等，另一个是16位偏移量，用于计算相对于分支指令所在地址的**分支目标地址**（branch target address）。它的指令格式为`beq $t1,$t2,offset`。为了实现该指令，我们必须将PC值与符号扩展后的指令偏移量字段相加以得到分支目标地址。分支指令（见第2章）的定义中有两个需要注意的地方：

- 指令集规定计算分支地址时使用的基地址，是分支指令的下一条指令的地址。原因是我们在取指通路中计算了PC+4（下一条指令的地址），用这个值作为计算分支目标地址时的基地址比较容易实现。
- 系统结构还规定偏移量左移2位以指示以字为单位的偏移量，这样偏移量的有效范围就扩大了4倍。

为了处理后面这种情况，我们需要把偏移量左移2位。

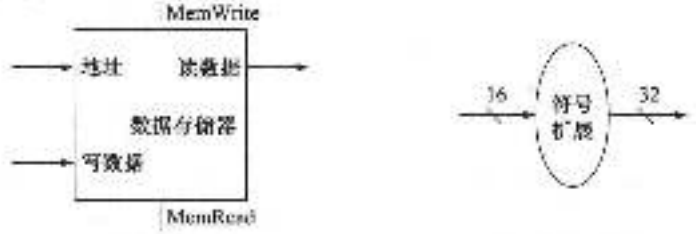
除了计算分支目标地址，还必须确定是顺序执行下一条指令，还是去执行分支目标地址处

的指令。当分支条件为真（例如，操作数相等）时，分支目标地址成为新的 PC，我们就说分支发生（branch taken）了。若操作数不等，自增后的 PC 将取代当前 PC（就像其他普通指令一样），这时就说分支未发生（branch not taken）。

- 分支目标地址：该地址指定了一个分支，如果分支发生，那么它将成为新的程序计数器（PC）。在 MIPS 架构中，指令偏移域与分支指令的下一条指令地址之和组成分支目标。
- 分支发生：分支条件满足而 PC 变为分支目标地址的分支。所有的无条件跳转都是发生的分支。
- 分支未发生：分支条件不满足而 PC 变为分支指令的下一条指令地址。

所以，分支数据通路需要进行两个操作：计算分支目标地址和比较操作数。（很快我们还将讲到，分支指令也影响数据通路的取指部分。）图 4-9 为分支数据通路。为计算分支目标地址，分支数据通路包含了一个如图 4-8 所示的符号扩展单元和一个加法器。为了进行比较，要由图 4-7a 的寄存器堆提供两个寄存器操作数（但不需向寄存器堆写入数据）。另外，比较可由在附录 B 设计的 ALU 完成。因为 ALU 提供一个指示结果是否为 0 的输出信号，故可以把两个寄存器数作为输入，并将 ALU 设置为减法。若 ALU 输出的零信号有效，则可知两操作数相等。尽管零输出信号始终指示结果是否为 0，但我们只用它来实现分支时的等值测试。稍后将详细介绍将 ALU 用于数据通路时，怎样连接它的控制信号。

253
↓
254




a）数据存储器单元　　b）符号扩展单元

图 4-8　除了图 4-7 中的寄存器堆和 ALU，存储指令和取数指令还需要两个单元——数据存储器单元和符号扩展单元。数据存储器单元是一个状态单元，两个输入为地址和所写数据，一个输出为读出的数据。读、写控制信号都是独立的，尽管任意时钟只能激活其中一个。不像寄存器，存储器单元需要一个读控制信号，因为读一个无效地址可能会出问题，我们在第 5 章会看到这种情况。符号扩展单元有一个 16 位的输入，符号扩展为 32 位后输出（见第 2 章）。假定数据存储器的写是边沿触发的。标准的存储芯片实际上有一个写使能信号用于写操作。尽管标准存储芯片的写使能信号不是边沿触发的，我们的边沿触发的设计可以很容易地应用于真正的存储芯片。关于存储器芯片工作细节的讨论，见附录 B 的 B.8 节

跳转指令将偏移地址的低 26 位左移两位后，以之代替 PC 的低 28 位。移位通过给偏移量后面加上两个 0 实现（如第 2 章所述）。

**精解**　在实际 MIPS 指令集中，分支指令是“延迟的”，即无论分支条件是否满足，它之后的那条指令总被执行。条件不满足时，情况与一般分支指令相同；条件满足时，延迟的分支指令先执行它下面的那条指令，然后再跳转到指定的目标地址。将分支指令设计为延迟的原因是减轻流水线对分支指令的影响（见 4.8 节）。为简单起见，本章仅实现非延迟的 beq 指令。

255　延迟的分支：不管分支条件是否满足，分支指令之后的那条指令总被执行的一种分支。

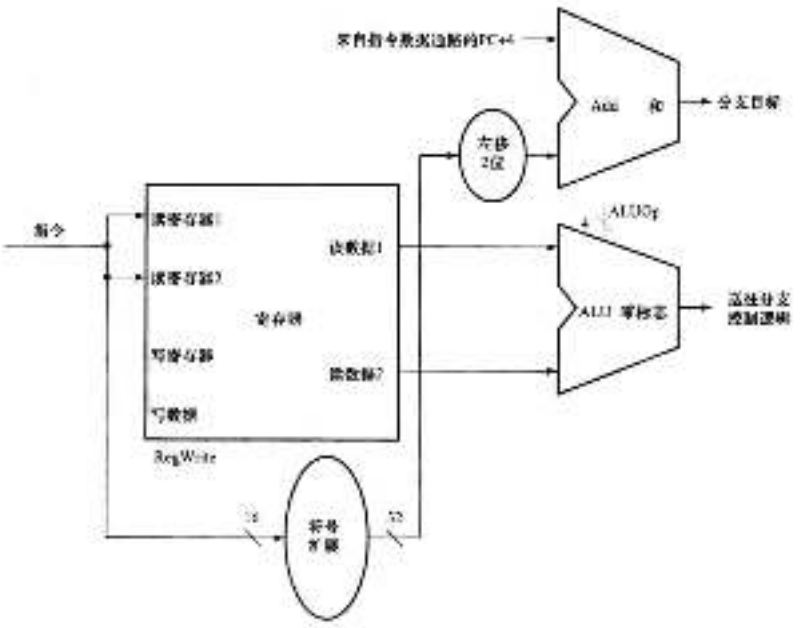



图 4-9 在分支指令的数据通路中，用 ALU 计算分支条件是否成立，用另外的加法器将自增后的 PC 值与符号扩展后左移两位的指令低 16 位（分支偏移量）相加，得到分支目标地址。标有“左移两位”的单元只是输入到输出之间一条简单的数据通路，它给符号扩展后的偏移量字段的低位加上两个 0（二进制）。因为“移动”的距离是固定的，所以并不需要真正的移位电路。我们知道偏移量是从 16 位扩展而来的，所以移位只会丢掉“符号位”。控制逻辑根据 ALU 的零输出决定是用自增的 PC 还是分支目标地址来取代当前的 PC

## 创建一个简单的数据通路

我们已经讨论了不同指令类型所需要的数据通路单元，可以把它们连在一起并加上控制来完成一个最简单的 MIPS 子集实现方案。这个最简单的数据通路中，每个时钟周期执行一条完整的指令。这意味着每条指令执行过程中任何数据通路单元都只能被用一次，如果需要使用多次则必须将该数据通路单元复制多份。所以我们除了需要一个指令存储器外，还需要一个数据存储器。尽管有的功能单元需要复制，但在执行不同指令时，很多功能单元也可以被共享。

为了在两种不同类型的指令间共享数据通路单元，我们需要让功能单元有多个输入，而使用多选器和控制信号来从多个输入中进行选择。 256

### 例题：建立一个数据通路

算术逻辑指令（或 R 型指令）的数据通路与访存指令的数据通路很相似。它们的主要区别为：

- 算术逻辑指令使用 ALU，并且其输入来自两个寄存器。存储指令也使用 ALU 来进行地址计算，但 ALU 的第二个输入是对指令中 16 位偏移地址进行符号扩展后的值。
- 存入目标寄存器中的值来自于 ALU（对 R 型指令而言）或者存储器（对取数操作而言）。

试设计存储指令和算术逻辑指令操作部分的数据通路，只能使用一个寄存器堆和一个 ALU，可增加必要的多选器。

### 答案

为了只用一个 ALU 和一个寄存器堆来创建一个数据通路，ALU 的第二个输入和要存入寄存器堆的数据都需要两个不同的来源。所以，要在 ALU 的输入和寄存器堆的输入数据处各加

入一个多选器。图4-10给出了合并后的数据通路。

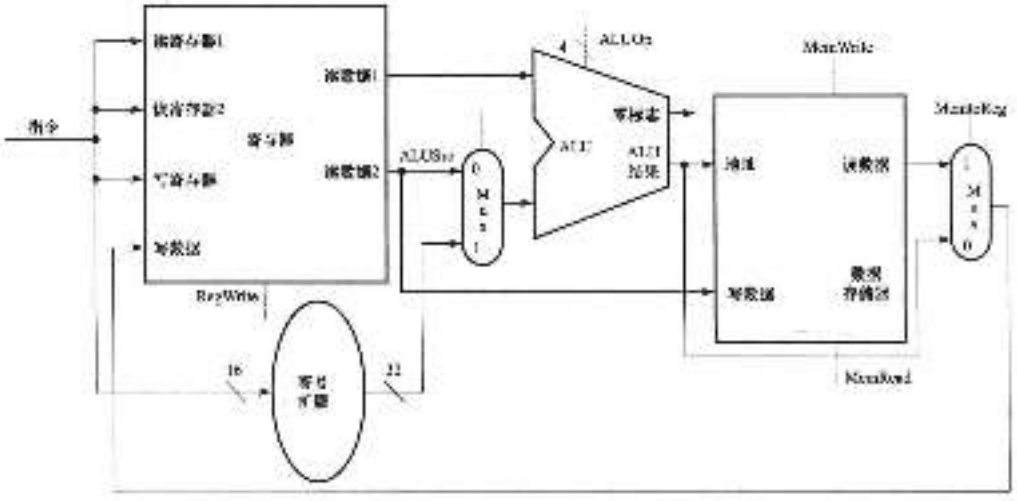



图4-10 访存指令和R型指令数据通路的合并。这个例子说明了如何通过加入多选器将图4-7和图4-8合并成一个数据通路，其中增加了两个多选器 □

现在，加上图4-6的取指数据通路、图4-9的分支数据通路、图4-10的R型指令和访存指令数据通路，我们可以把所有部件合并在一起建立一个简单的MIPS体系结构数据通路，如图4-11所示。由于分支指令用主ALU对寄存器操作数进行比较，所以还需要图4-9中的加法器完成分支目标地址的计算。此外还增加了一个多选器，用于选择是将顺序的指令地址（PC+4）还是分支目标地址写入PC。

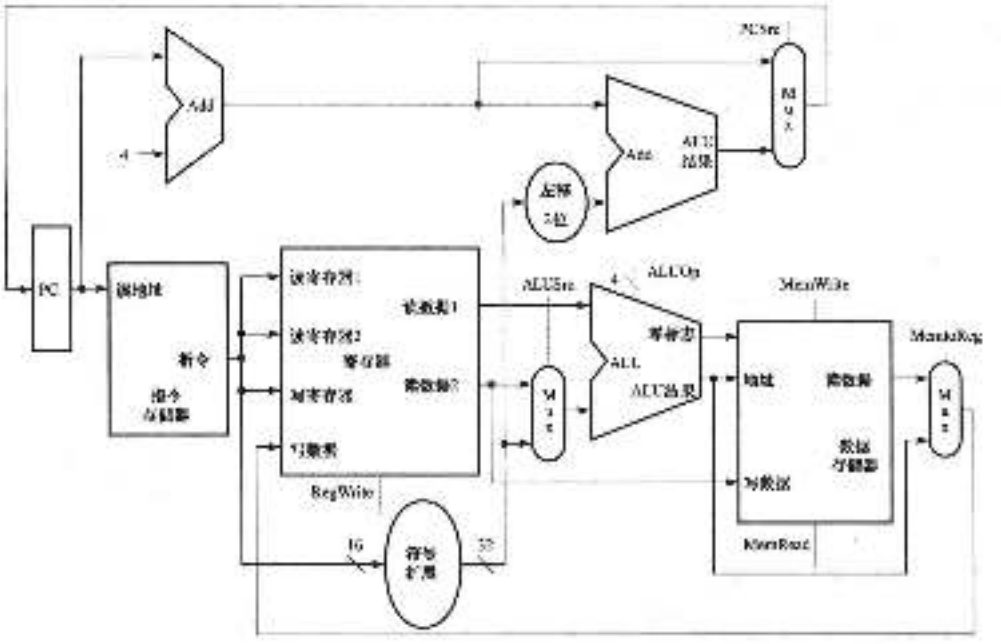



图4-11 将不同类型指令所需的功能单元合并在一起实现的一个简单的核心MIPS系统结构数据通路。图中的部件来自图4-6、图4-9和图4-10。这个数据通路可以在一个时钟周期内完成基本的指令（存取字、ALU操作和分支）。为了支持分支指令还增加了一个额外的多选器。对跳转指令的支持将在以后增加

在完成这个简单的数据通路后，可以加上控制单元。控制单元必须能够接收输入，能够产生每个状态单元的写信号、每个多选器的选择信号和 ALU 的控制信号。由于 ALU 的控制比较特殊，因此最好先设计 ALU，随后再设计控制单元的其他部分。

**小测验**

257
258

Ⅰ. 对取数指令来说，以下哪个是正确的？参考图 4-10。

a. MemtoReg 信号线应该被设置为将存储器中的数据发送至寄存器堆。

b. MemtoReg 信号线应该被设置为将正确的目标寄存器的数据发送至寄存器堆。

c. 对取数指令而言，MemtoReg 信号线的设置无关紧要。

Ⅱ. 本节描述的单周期数据通路必须有独立的指令存储器和数据存储器，因为：

a. MIPS 中指令与数据的格式是不同的，所以需要不同的存储器。

b. 使用独立的存储器会比较便宜。

c. 因为处理器在一个周期内只能操作每个部件一次，而在一个周期内不可能对一个单端口存储器进行两次存取。

## 4.4 一个简单的实现机制

在本节中，我们将学习如何实现最简单的 MIPS 子集。我们用上一节的数据通路和增加一个简单的控制单元来实现一个 MIPS 体系结构。这一结构实现了取字（lw）、存字（sw）、相等则分支（beq）和算术逻辑指令加法（add）、减法（sub）、与运算（AND）、或运算（OR）和小于则置位（set on Less than），后面我们还将实现跳转指令（j）。

### 4.4.1 ALU 控制

附录 B 中描述的 MIPS ALU 在 4 位控制信号上定义了 6 种有效的输入组合：

| ALU 控制信号 | 功能 | ALU 控制信号 | 功能 |
|---|---|---|---|
| 0000 | 与 | 0110 | 减 |
| 0001 | 或 | 0111 | 小于则置位 |
| 0010 | 加 | 1100 | 或非 |

根据指令类型的不同，ALU 将执行上述 5 种功能中的一种。（或非操作在我们目前实现的 MIPS 子集中暂时没有使用。）对于取字和存储字指令，ALU 用加法计算存储器地址。对于 R 型指令，根据指令低 6 位的 funct 字段（见第 2 章），ALU 执行 5 种操作中的一种（与、或、减、加、小于则置位）。对相等则分支指令，由 ALU 执行减法操作。

259

使用一个小的控制单元即可生成 4 位的 ALU 控制信号，其输入为指令的 funct 字段和 2 位的 ALUOp 字段。ALUOp 指明要进行的操作是存取指令需要的加法（00）、beq 需要的减法（01），还是由指令的 funct 字段（10）决定。该 ALU 控制单元输出 4 位信号，即前面介绍的 4 位控制信号，直接对 ALU 进行控制。

图 4-12 说明了怎样根据 2 位的 ALUOp 和 6 位的 funct 功能字段生成 ALU 的控制信号。在本章的后面将会看到怎样由主控制单元生成 ALUOp。

这种多级译码的方法（主控制单元生成 ALUOp 作为 ALU 控制单元的输入，再由 ALU 控制单元生成真正控制 ALU 的信号）是一种常用的实现方式。使用多级译码可以减小主控制单元的规模。使用多个小控制单元还可能提高控制单元的速度。这种优化是很重要的，因为控制单元的性能对时钟周期非常关键。

| 指令操作码 | ALUOp | 指令操作 | funct 字段 | ALU 动作 | ALU 控制信号 |
|---|---|---|---|---|---|
| 取字 | 00 | 取字 | XXXXXX | 加 | 0010 |
| 存字 | 00 | 存储字 | XXXXXX | 加 | 0010 |
| 相等则分支 | 01 | 相等分支 | XXXXXX | 减 | 0110 |
| R 类型 | 10 | 加 | 100000 | 加 | 0010 |
| R 类型 | 10 | 减 | 100010 | 减 | 0110 |
| R 类型 | 10 | 与 | 100100 | 与 | 0000 |
| R 类型 | 10 | 或 | 100101 | 或 | 0001 |
| R 类型 | 10 | 小于则置位 | 101010 | 小于则置位 | 0111 |

图 4-12 如何根据 ALUOp 控制位和 R 型指令的 funct 字段设置 ALU 的控制信号。第一列是操作码，操作码决定了 ALUOp 位。所有的编码以二进制给出。注意，当 ALUOp 为 00 或 01 时，ALU 的动作不依赖于 funct 字段，故功能字段记为 XXXXXX。当 ALUOp 为 10 时，funct 字段用于设置 ALU 的控制信号。详情见附录 B

有多种不同方法把 2 位的 ALUOp 和 6 位的 funct 字段映射为 4 位 ALU 控制信号。因为 funct 功能字段的 64 种可能取值中只有很小一部分有意义，并且只有当 ALUOp 取值为 10 时才使用功能字段，我们可以用一个小逻辑单元去识别可能取的值，以生成正确的 ALU 控制信号。

260

为设计这个逻辑单元，有必要为 ALUOp 和 funct 字段有意义地组合生成一张真值表，如图 4-13 所示。该**真值表**（truth table）说明了如何根据两个输入字段得到 4 位的 ALU 控制信号。由于完整的真值表很大（$2^8=256$ 项），我们并不关心所有的输入组合，只列出了使 ALU 控制信号有效的部分表项，而忽略那些恒为 0 或无关的项。在本章中，我们将一直采用这样的方式表示真值表（这样做的缺点在附录 D 的 D.2 节中讨论）。

由于在许多情况下对某些输入的取值并不关心，为了简化真值表，我们也列出无关项（don't-care term）。真值表中的无关项（在输入列中用 X 表示）表明，输出与该列对应的输入取值无关。如图 4-13 的第一列所示，当 ALUOp 取 00 时，无论 funct 字段取何值，ALU 控制总为 0010。这时，真值表中此行的 funct 字段就是无关项。在后面，还会有另一种无关项的例子。无关项的概念在附录 B 中有更多的讨论。

- 真值表：逻辑操作的一种表示方法，即列出输入的所有情况和每种情况下的输出。
- 无关项：逻辑函数的一个元素，表示输出与该输入取值无关。无关项可以用不同的方式指定。

真值表建好以后，可以进行优化并转化成门电路。这是一个完全机械的过程。所以将此过程及其结果放在附录 D 中的 D.2 节讨论。

| ALUOp | | Funct 字段 | | | | | | 操作 |
|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | |
| 0 | 0 | X | X | X | X | X | X | 0010 |
| X | 1 | X | X | X | X | X | X | 0110 |
| 1 | X | X | X | 0 | 0 | 0 | 0 | 0010 |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 0110 |
| 1 | X | X | X | 0 | 1 | 0 | 0 | 0000 |
| 1 | X | X | X | 0 | 1 | 0 | 1 | 0001 |
| 1 | X | X | X | 1 | 0 | 1 | 0 | 0111 |

图 4-13 4 位 ALU 控制信号（称为操作）的真值表。该真值表的输入为 ALUOp 和 funct 字段。在此只列出了 ALU 控制有效的项，也包括一些无关项。例如，ALUOp 不使用编码 11，故真值表包含 1X 和 X1 项，而不是 10 和 01 项。同样，当使用 funct 字段时指令的前两位（F4 和 F5）总是 10，所以它们是无关项，在真值表中用 XX 代替

## 4.4.2 主控制单元的设计

我们已经描述了如何使用 funct 和 2 位信号作为输入来进行 ALU 控制单元的设计，现在来看看控制的其他部分。在开始之前，首先看一条指令的各个字段和图 4-11 所示的数据通路所需的控制信号。为了理解怎样将指令的各个字段与数据通路相连，需要复习一下三种指令类型的格式：R 型指令、分支指令和存取指令，如图 4-14 所示。 261




图 4-14 三种指令类型（R 型、存取和分支）使用的两种指令格式。后面我们马上会讲到，跳转指令使用另一种格式。a）R 型指令的格式，操作码为 0，寄存器操作数有 3 个：rs、rt 和 rd。rs 和 rt 字段为源操作数，rd 字段为目的操作数。funct 字段指出 ALU 功能，由前面设计的 ALU 控制单元译码。我们实现的 R 型指令有 add、sub、AND、OR 和 slt。shamt 字段只用于移位指令，本章中暂不考虑。b）取数指令（操作码 = $35_{10}$）和存储指令（操作码 = $43_{10}$）的格式。rs 寄存器作为基址与 16 位的地址字段相加以得到访存地址。对取数指令，rt 是取出数据的目的寄存器。对存储指令，rt 是要存入存储器的数据所在的寄存器。c）相等则分支指令（操作码 = 4）的格式。rs 和 rt 是源寄存器，用于比较是否相等。16 位地址进行符号扩展、移位后与 PC + 4 相加以得到分支目标地址

MIPS 的指令格式遵循以下规则：

- Op 字段，第 2 章亦称操作码（opcode），总是为 31:26 位。我们将用 Op[5:0] 来表示。
- 对于 R 型指令、分支指令和存取指令，要读取的两个寄存器为 rs 和 rt 字段，分别为 25:21 位和 20:16 位。
- 存取指令的基址寄存器在 25:21 位中（rs 字段）。
- 相等则分支指令、存取指令的 16 位偏移量在 15:0 位中。
- 有两个地方存放目标寄存器。对取数指令为 20:16 位（rt 字段），对 R 型指令为 15:11 位（rd 字段）。所以我们需要一个多选器，以指示要写的寄存器号在哪个字段中。

⇒ 操作码：指示指令操作和格式的字段。

从第 2 章得到的第一个设计原则——简单源于规整——在这里就体现出来了。 262

根据上述信息，可以给简单的数据通路加上指令标记并增加一个多选器（用于选择寄存器堆的写寄存器号），图 4-15 展示了这些增加的部件和 ALU 控制块、状态单元的写信号、数据存储器的读信号和多路选择器的控制信号。由于所有的多路选择器都是两个输入端，因此每个多路选择器都需要一条单独的控制线。

图 4-15 给出了 7 个 1 位控制信号和 2 位 ALUOp 控制信号。我们已经说明了 ALUOp 控制信号如何工作，在继续说明指令执行过程中如何设置这些控制信号之前，最好定义一下其他 7 条控制信号如何工作。图 4-16 说明了这 7 个控制信号的功能。

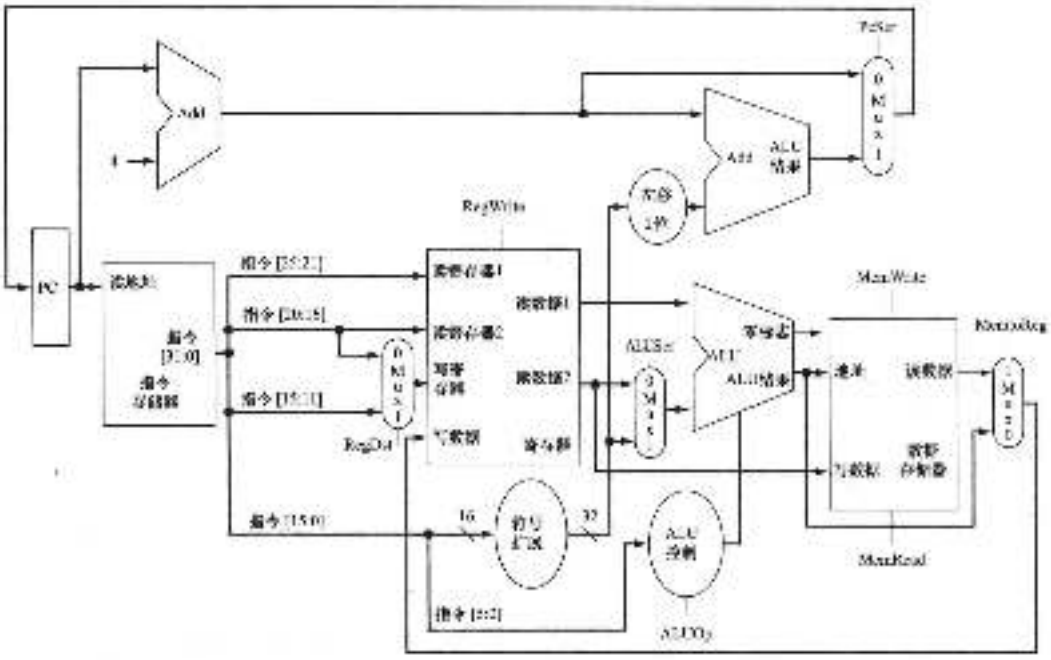



图4-15 在图4-11的数据通路上增加了所有必需的多选器，并标识出了所有的控制信号。控制信号以灰色线表示。还增加了ALU控制单元。PC不需要写控制，因为它在每个时钟周期末都被写入一次。分支控制逻辑决定给PC自增还是写入分支目标地址

| 控制信号名 | 无效时的含义 | 有效时的含义 |
|---|---|---|
| RegDst | 写寄存器的目标寄存器号来自rt字段（位20:16） | 写寄存器的目标寄存器号来自rd字段（位15:11） |
| RegWrite | 无 | 寄存器堆写使能有效 |
| ALUSrc | 第二个ALU操作数来自寄存器堆的第二个输出（读数据2） | 第二个ALU操作数为指令低16位当符号扩展 |
| PCSrc | PC由PC+4取代 | PC由分支目标地址取代 |
| MemRead | 无 | 数据存储器读使能有效 |
| MemWrite | 无 | 将写入数据输入端的数据写入到由地址指定存储器单元中取 |
| MemtoReg | 写入寄存器的数据来自ALU | 写入寄存器的数据来自数据存储器 |

图4-16 7个控制信号的作用。当同路多选器的控制信号有效时，选择第1个输入，否则选择第0个输入。需要注意的是，所有状态单元都有一个默认输入和时钟信号，且用于写操作的控制。时钟信号从来不在状态单元之外通过任何门电路，因为这样可能导致时序问题（附录B中对此问题有进一步的讨论。）

了解了每个控制信号的功能之后，再来看看它们如何设置。除PCSrc控制信号外，所有控制信号都可由控制单元只根据指令的操作码来确定。而PCSrc信号有效的条件是指令为相等则分支（由控制单元确定），且用于等值比较的ALU的零输出有效。为生成PCSrc信号，需将一个来自控制单元称为“Branch”（分支）的信号与ALU的零输出信号相“与”。

263

现在，这9位控制信号（图4-16的7位和2位ALUOp）的状态可根据控制单元的6位输入信号（操作码位31:26）来设置。图4-17给出了包含控制单元和控制信号的数

据通路。

在设计控制单元的计算公式或真值表之前，这里定义一下控制功能。由于控制信号的状态仅由操作码决定，我们需要定义在每种操作码下每个控制信号的取值：0、1 或任意值 X。根据图 4-12，图 4-16 和图 4-17，图 4-18 定义了对应于每种操作码的控制信号状态。

### 1. 数据通路的操作

根据图 4-16 和图 4-18 包含的信息，可以设计出控制单元逻辑，但在此之前先分析一下每条指令是如何使用数据通路的。接下来的一些图说明了 3 种类型的指令在数据通路上的执行过程。在这些图中，有效的控制信号和数据通路部件已着重标出。需要注意的是，对于多选器其控制为 0 时，即使其控制信号没有着重标出，它也有相应的动作。对于多位信号，只要其中任何信号有效，就将其着重标出。

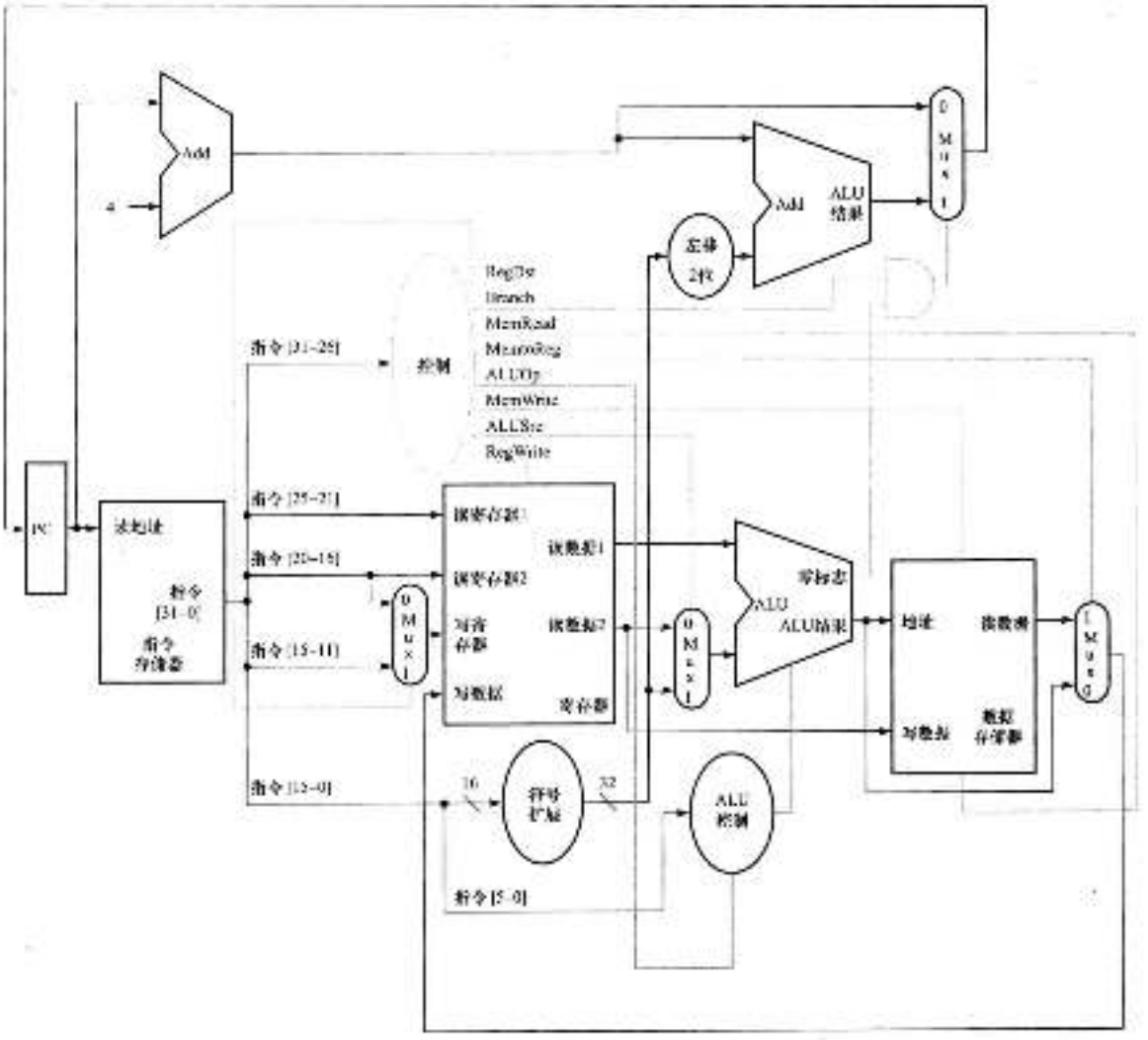



图 4-17 包含控制单元的简单数据通路。控制单元的输入为指令的 6 位操作码。控制单元的输出包括 3 个控制多选器的 1 位信号（RegDst、ALUSrc 和 MemtoReg），3 个控制寄存器堆和存储器读写的信号（RegWrite、MemRead 和 MemWrite），一个决定是否可以转移的 1 位信号（Branch），和一个 ALU 的 2 位控制信号（ALUOp）。分支控制信号与 ALU 的零输出一起送入一个与门，其输出控制下一个 PC 的选择。注意 PCSrc 是一个间接信号，而不是从控制单元直接得来。所以在图中我们没有标出这个信号名称

| 指令 | RegDst | ALUSrc | MemtoReg | RegWrite | MemRead | MemWrite | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|---|
| R 型 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

图 4-18 按指令操作码设置的控制信号。表的第一行对应于 R 型指令（add、sub、AND、OR 和 slt）：源寄存器字段都为 rs 和 rt，目的寄存器字段为 rd，这决定了 ALUSrc 和 RegDst 信号如何设置；并且，R 型指令写寄存器（RegWrite = 1），但是不读写数据存储器。当 Branch 控制信号为 0 时，PC 无条件地由 PC + 4 取代；反之，如果 ALU 的零输出也为高，则 PC 由分支目标地址取代。当 R 型指令的 ALUOp 为 10 时，ALU 的控制信号应由 funct 字段生成。本表的第二行和第三行给出了 lw、sw 指令的控制信号设置：ALUSrc 和 ALUOp 被设为进行地址计算；MemRead 和 MemWrite 被设为进行存储器访问；最后，RegDst 和 RegWrite 被设为在装载指令中将结果存入寄存器 rt 中。分支指令与 R 型指令相似，因为它将寄存器 rs 和 rt 送入 ALU；分支指令的 ALUOp 字段被设为进行减法（ALUOp = 01），以进行等值的测试。注意，RegWrite = 0 时 MemtoReg 的设置无关紧要——因为寄存器没有被写入，寄存器写端口的数据值不被使用，所以最后两行 MemtoReg 的值由于不被关心而用 X 取代。RegWrite = 0 时，RegDst 的值也可用 X 取代。这种无关项必须由设计者加入，因为它依赖于对数据通路工作原理的了解

图 4-19 给出了执行 R 型指令（如 add $t1，$t2，$t3）时的数据通路操作。尽管一切都发生在一个时钟周期内，但我们可以考虑分 4 步来执行指令，具体如下：

1）从指令存储器中取出指令，PC 自增。

2）从寄存器堆中读出寄存器 $t2 和 $t3。同时，主控制单元计算出各控制信号的状态。

3）ALU 根据 funct 字段（指令的 5:0 位）确定 ALU 的功能，对从寄存器堆读出的数据进行操作。

4）将 ALU 的结果写入寄存器堆，根据指令的 15:11 位选择目标寄存器（$t1）。

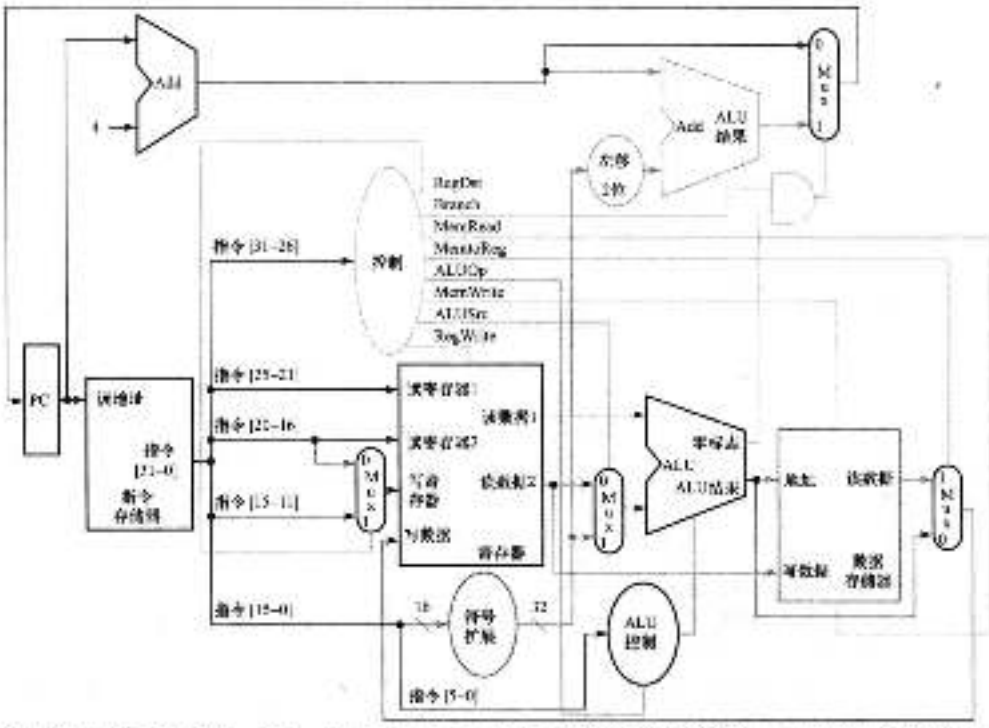



图 4-19 执行 R 型指令（如 add $t1，$t2，$t3）时数据通路的操作。操作中用到的控制信号、数据通路单元和连接均用灰色线显示

我们可以用和图4-19类似的方式描述取数指令（如 `lw $t1,offset($t2)`）的执行。图4-20给出了取数时有效的功能单元和控制信号。可以考虑将取数指令的执行分为5步（与将R型指令的执行分为4步类似）：

1）从指令存储器取指，PC自增。

2）从寄存器堆读出寄存器 `$t2` 的值。

3）ALU将从寄存器堆读出的值与符号扩展后的指令低16位值（`offset`）相加。

4）将ALU的结果作为数据存储器的地址。

5）存储单元的数据写入寄存器堆，目标寄存器由指令的20:16位（`$t1`）指出。

264 ~ 266

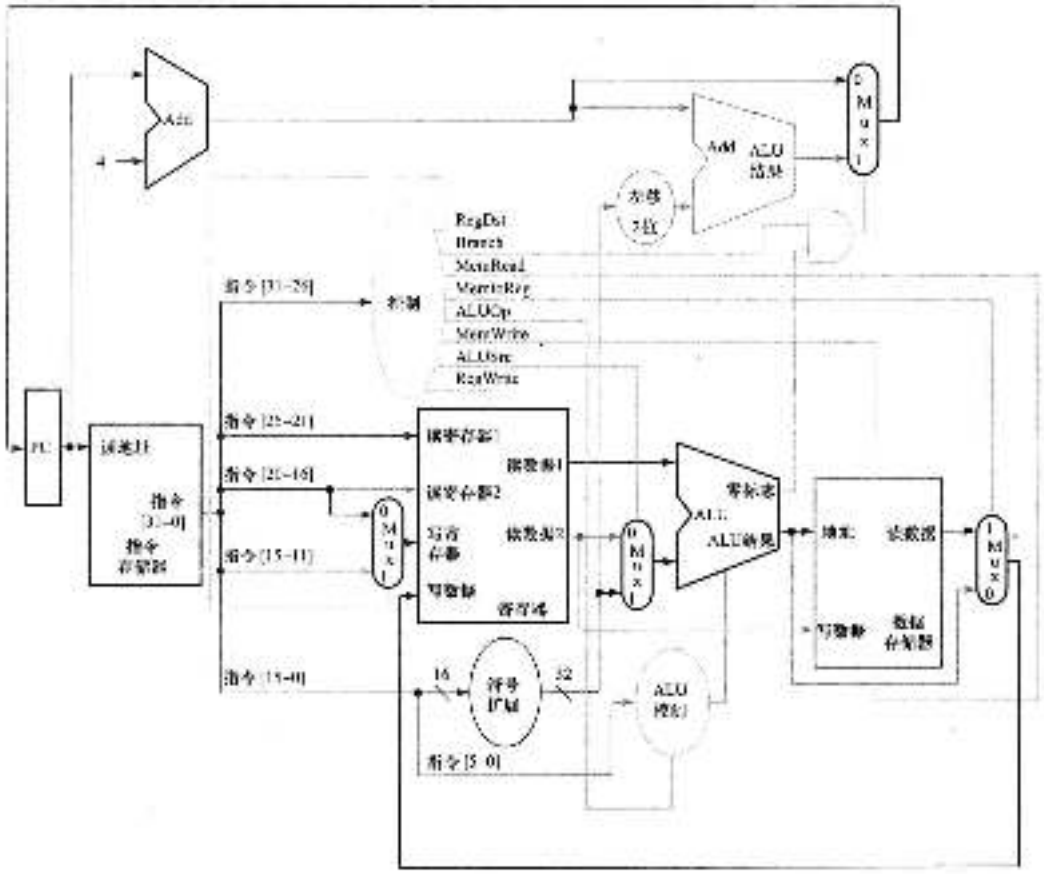



图4-20 执行取数指令时数据通路的操作。操作中用到的控制信号、数据通路单元和连接用灰色线显示。存储指令的操作与此类似，主要区别在于数据存储器的控制将指明要进行写操作，而不是读操作，读出的第二个寄存器的值将作为要存储的数据，并且不会有将数据存储器的内容写入寄存器的操作

最后，以同样方式说明相等则分支指令（如 `beq $t1,$t2,offset`）的执行过程。它的操作类似于R型指令，但ALU的零输出用于决定PC自增为PC+4还是置为分支目标地址。图4-21给出了执行的4步：

1）从指令存储器中取指，PC自增。

2）从寄存器堆读出寄存器 `$t1` 和 `$t2` 的值。

3）ALU将从寄存器堆读出的两数相减。PC+4的值与符号扩展并左移2位后的指令低16位（`offset`）相加，结果即分支目标地址。

4）根据ALU的零输出决定哪个加法器的结果存入PC中。

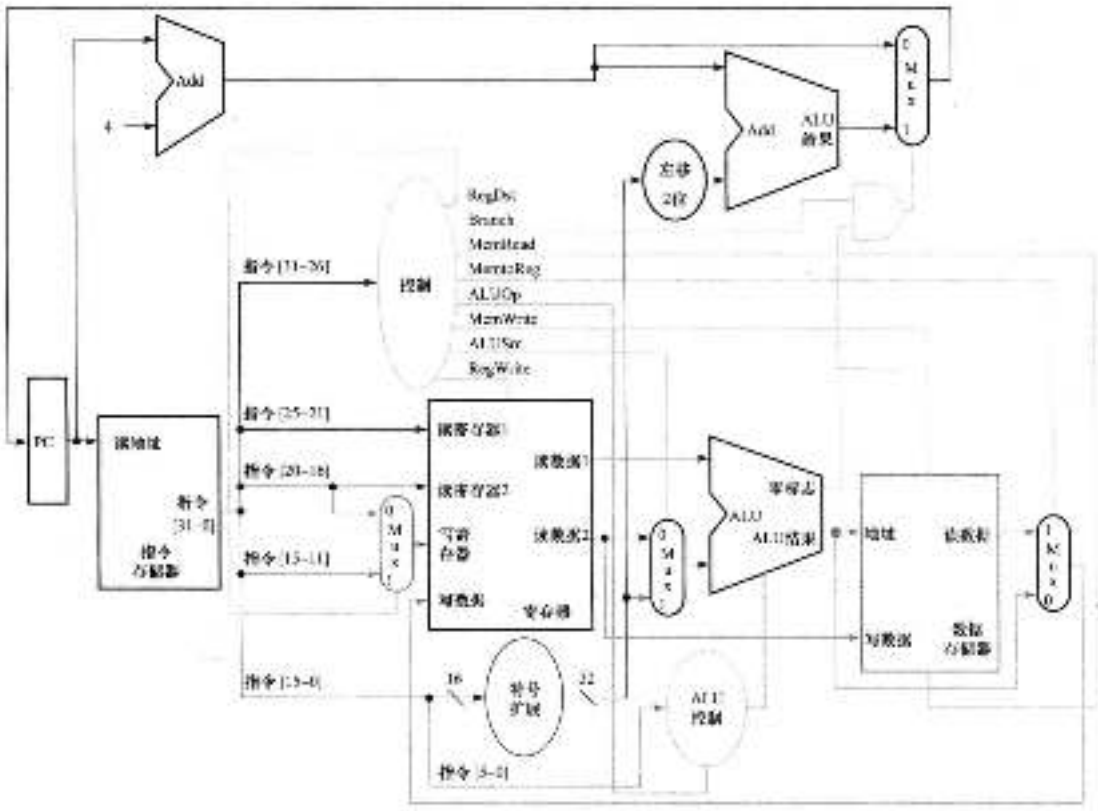



图 4.21 执行相等则分支指令时数据通路的操作。控制信号、数据通路单元和连接使用灰色线显示。在使用寄存器堆和 ALU 进行比较操作之后，ALU 的零输出用于在两种可能的 PC 中选择其一

## 2. 控制的结束

讨论过指令的操作之后，现在继续讨论控制单元的实现。控制单元的功能可由图 4-18 精确定义，其输入为 6 位操作码 Op[5:0]，输出为控制信号。这样，可以基于操作码的二进制编码为每个输出建立一张真值表。

根据这些信息，可以把控制单元（包括所有输出的逻辑综合）描述在一张大的真值表中，如图 4-22 所示。它完整地描述了控制功能，可以自动地转换为门电路实现，附录 D 的 D.2 节对此进行了描述。

| 输入或输出 | 信号名 | R 型 | lw | sw | beq |
|---|---|---|---|---|---|
| 输入 | Op5 | 0 | 1 | 1 | 0 |
| | Op4 | 0 | 0 | 0 | 0 |
| | Op3 | 0 | 0 | 1 | 0 |
| | Op2 | 0 | 0 | 0 | 1 |
| | Op1 | 0 | 1 | 1 | 0 |
| | Op0 | 0 | 1 | 1 | 0 |

图 4.22 简单的单周期实现的控制功能真值表。表的上半部分为输入，其包括操作码（对应于指令的 31:26 位的 Op[5:0]）的 4 种组合。表的下半部分为 4 种组合的输出。因此，RegWrite 对于两种不同的输入组合是有效的。如果只考虑这张表中的 4 个操作码，则可以用输入部分的无关项简化真值表。例如，可以由表达式 $Op5 \cdot \overline{Op2}$ 确定是否为 R 型指令，因为这已经足够将 R 型指令与 lw、sw 和 beq 指令区分开。之所以不用这种简化，是因为在 MIPS 指令集的完整实现中会用到其他操作码

267
|
269

| 输入或输出 | 信号名 | R型 | lw | sw | beq |
|---|---|---|---|---|---|
| 输出 | RegDst | 1 | 0 | X | X |
| | ALUSrc | 0 | 1 | 1 | 0 |
| | MemtoReg | 0 | 1 | X | X |
| | RegWrite | 1 | 1 | 0 | 0 |
| | MemRead | 0 | 1 | 0 | 0 |
| | MemWrite | 0 | 0 | 1 | 0 |
| | Branch | 0 | 0 | 0 | 1 |
| | ALUOp1 | 1 | 0 | 0 | 0 |
| | ALUOp0 | 0 | 0 | 0 | 1 |

图 4-22 （续）

既然我们已经有了包含 MIPS 核心指令集中绝大多数指令的**单周期实现**（single-cycle implementation），在此基础之上我们再加上跳转指令，看看怎样通过扩展基本数据通路和控制通路，来实现指令集中的其他指令。

⊃ 单周期实现：也被称为单时钟周期实现（single clock cycle implementation），即一个时钟周期执行一条指令的实现机制。虽然它很容易理解，但现实中，由于它太慢而不实用。

**例题：跳转的实现**

图 4-17 给出了第 2 章中提到的许多指令的实现，但没有给出跳转指令的实现。请对图4-17的数据通路和控制通路进行扩展，从而支持跳转指令，并给出控制信号的设置方式。

**答案**

跳转指令类似于分支指令，但它以不同的方式计算目标 PC，且是无条件的。与分支指令一样，跳转地址的最低两位恒为 00$_2$。32 位跳转地址的次低 26 位来自指令的 26 位立即数，如图 4-23 所示。跳转地址的高 4 位来自于跳转指令的 PC +4。也就是说，实现跳转指令即将下面 3 个部分拼接为跳转地址：

- 当前 PC +4 的高 4 位（下条指令地址的 31:28 位）。
- 跳转指令的 26 位立即数字段。
- 低位 00$_2$。

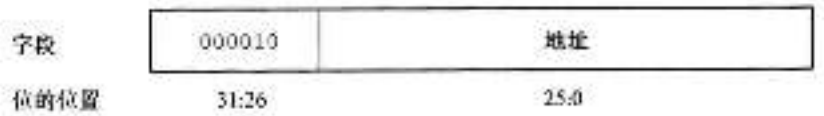



图 4-23 跳转指令的格式（操作码 =2）。跳转指令的目的地址由当前 PC +4 的高 4 位与跳转指令中的 26 位地址连接，再将 00 作为最低两位形成

图 4-24 所示为在图 4-17 基础上增加了对跳转指令的支持。为了在 PC +4，分支目标 PC 和跳转目标 PC 中选择新 PC 值的来源，加上了一个多选器。这个多选器需要一个控制信号 Jump。只有当操作码为 2，即指令为跳转指令时，该控制信号才有效。 □ 270

## 4.4.3 为什么不使用单周期实现方式

虽然单周期设计也可以正确地工作，但现代设计中并不采取这种方式，因为它的效率太低。究其原因，是在单周期设计中，时钟周期对所有指令等长，这样时钟周期要由执行时间最长的那条指令决定。这条指令几乎可以肯定是取数指令，它依次使用了 5 个功能单元：指令存储器、寄存器堆、ALU、数据存储器、寄存器堆。虽然 CPI 为 1（见第 1 章），单周期实现方式的总体性能并不一定很好，因为时钟周期实在是太长了。

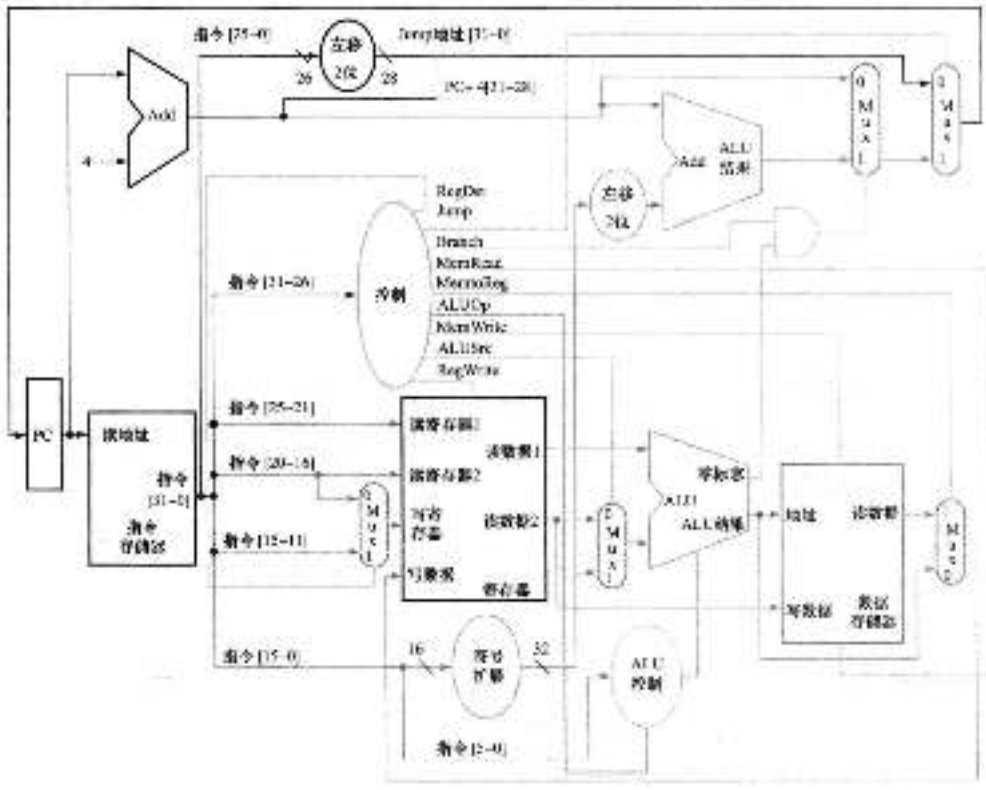



图 4-24 扩展后能处理跳转指令的简单控制和数据通路。加入了一个多选器（右上角）用来选择分支目标地址、跳转目标地址和下一指令地址三者之一。该多选器由 Jump 信号控制。跳转目标地址通过将 jump 指令中低 26 位地址左移两位，从而高效地增加 00 作为低位，然后将 PC+4 的高 4 位作为高位，从而产生 32 位地址

271 使用单周期设计的代价虽然很大，但对于小指令集来说，或许是可以接受的。事实上，早期具有简单指令集的计算机就曾经采用过这种实现方式。然而，若要实现包含浮点或更复杂指令的指令集，这样的单周期设计根本不能胜任。

因为时钟周期必须满足所有指令中最坏的情况，故不能使用那些缩短常用指令执行时间而不改善最坏情况的实现技术。这样，单周期实现方式违背了第 1 章中加速大概率事件这一设计原则。

在下一节，我们将看到一种称为流水线的实现技术，使用与单周期类似的数据通路，但效率更高。从方法来说，流水线是通过重叠多条指令的执行来提高效率的。

**小测验**

观察图 4-22 中的控制信号，你能在其基础上进行整合吗？其中是否有控制信号可以被其他控制信号取反来替代（提示：将无关项考虑进去）？如果有，不加反向器是否可以直接用一个控制信号替代另一个呢？

## 4.5 流水线概述

**绝对不要浪费时间。**

——美国谚语

流水线（pipelining）是一种实现多条指令重叠执行的技术。目前，流水线技术广泛应用。

⊃ 流水线：一种实现多条指令重叠执行的技术，与生产流水线类似。

本节对流水线的概念及其相关问题进行了概述。如果只是想对流水线技术有一个大致的了解，可以详细看完本节，然后直接跳到4.10节和4.11节学习在最近的处理器（Intel Core i7 和ARM Cortex-A8）中所使用的高级流水线技术。如果想深入了解基于流水线技术的计算机，4.6～4.9节给出了相关细节。

任何一个经常光顾洗衣店的人都会不自觉地使用流水线技术。非流水线方式的洗衣过程包括如下几个步骤：

1）把一批脏衣服放入洗衣机里清洗。

2）洗衣机洗完后，把衣服取出并放入烘干机中。

3）烘干衣服后，将衣服从烘干机中取出，然后放在桌子上叠起来。

4）叠好衣服后，请你的室友帮忙把桌子上的衣服收好。

当你的室友把这批干净衣服从桌子上拿走后，再开始洗下一批脏衣服。

采用流水线的方法将节省大量的时间，如图4-25所示。当把第一批脏衣服从洗衣机里取出放入烘干机之后，就可以把第二批脏衣服放入洗衣机里进行清洗了。当第一批衣服被烘干之后，就可以将它们叠起来，同时把洗净的下一批湿衣服放入烘干机中，同时再将下一批脏衣服 [272]
放入洗衣机里清洗。接着让你的室友把第一批衣服从桌子上收好，而你开始叠第二批衣服，这时烘干机中放的是第二批衣服，同时可以把第四批脏衣服放入洗衣机清洗了。这样，所有的洗衣步骤（流水线的步骤）都在同时操作。只要在每一个操作步骤中都有独立的工作单元时，我们就可以采用流水线的方式来快速完成任务了。

流水线的奇妙之处在于，对于单独的一批衣服来说，从它进洗衣机到烘干机，再到折叠、收拾，整个过程总的处理时间并没有缩短。而在有多批任务时流水线之所以快的原因是所有的工作都在并行地进行。因此，单位时间内能够完成的工作量就大大地增加了。流水线实际上是改善了洗衣系统的吞吐率。虽然洗每一件衣服的时间没有缩短，但如果有很多衣服要洗，吞吐率的改善就减少了完成整个工作的时间。

如果所有的步骤所需的时间一样，并且有足够的工作可做，那么从流水线得到的速度提高倍数等于流水线中步骤的数目，洗衣房的例子是4倍：清洗、烘干、折叠和收拾。采用流水线方式工作与非流水线方式工作的洗衣房相比在速度上提高了4倍：前者洗完20批衣服所需的时间是洗完一批衣服所需时间的5倍，而后者洗完20批衣服所需的时间是洗完一批衣服的20倍。在图4-25中，流水线方式只将处理速度提高了2.3倍的原因是图中只显示了清洗4批衣服的处理过程。注意图4-25中的流水线版本在开始和结束阶段的负载情况，可以看出其流水线未完全充满。当任务数量相对于流水线级数不足很大时，突然启动和逐渐结束会影响流水线的性能。在本例中，如果任务数量远大于4，那么绝大多数时候流水线都将是充满的，这时吞吐率的提升就非常接近于4倍。

同样的原理也可以应用到处理器中，即采用流水线方式执行指令。通常，一个MIPS指令包含如下5个处理步骤：

1）从指令存储器中读取指令。

2）指令译码的同时读取寄存器。MIPS的指令格式允许同时进行指令译码和读寄存器。

3）执行操作或计算地址。

4）从数据存储器中读取操作数。

5）将结果写回寄存器。

因此，本章讨论的MIPS流水线具有5个处理步骤。正如流水线能加速洗衣店的工作一样，下面的例子将说明流水线如何加快指令的总体执行时间。

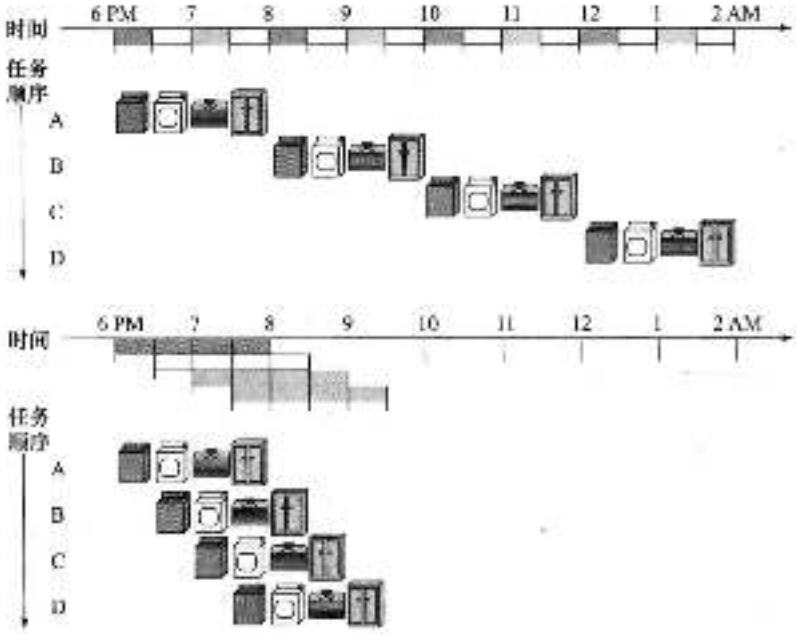



图 4-25 以洗衣店为例类比流水线的动作过程。安妮、布朗、凯西和唐每个人都有一些脏衣服要清洗、烘干、折叠及收拾。洗衣机、烘干机、"折叠机"和"收拾机"每个都需要 30 分钟来完成各自的任务。顺序的洗涤方法将花费 8 小时的时间洗完 4 批衣服，而流水线的洗涤方法只需要花费 3.5 小时。图中在二维时间线中通过资源的 4 次复制给出了不同工作负载的流水线，事实上每种资源只有一个

### 例题：单周期指令模型与流水线性能

为了使问题具体化，我们首先创建一个流水线结构。在本例以及本章剩余的部分中，我们将只考虑以下 8 条指令：取字（lw）、存字（sw）、加（add）、减（sub）、与（AND）、或（OR）、小于则置位（slt）和相等则分支（beq）。

本例将比较流水线指令执行与单周期指令执行的平均执行时间，其中在单周期模型中所有指令的执行都花费一个时钟周期。假设主要功能单元的操作时间为存储器访问：200ps；ALU 操作：200ps；寄存器堆的读写：100ps。在单周期模型中，每一条指令都只花费一个时钟周期，因此，时钟周期必须满足最慢的指令。

273 ~ 274

### 答案

8 条指令中每一条指令所需要的执行时间如图 4-26 所示。单周期模型的设计必须考虑到最慢的指令，在图 4-26 中是 lw，因此，每一条指令所需要的执行时间为 800ps。与图 4-25 类似，图 4-27 比较了三条取数指令非流水线与流水线方式的执行过程，其中在非流水线方式中，第一条与第四条指令之间的时间差是 3 × 800ps = 2 400ps。

| 指令类型 | 取指令 | 读寄存器 | ALU 操作 | 数据存取 | 写寄存器 | 总时间 |
|---|---|---|---|---|---|---|
| 取字（lw） | 200ps | 100ps | 200ps | 200ps | 100ps | 800ps |
| 存字（sw） | 200ps | 100ps | 200ps | 200ps | | 700ps |
| R 型（add、sub、AND、OR、slt） | 200ps | 100ps | 200ps | | 100ps | 600ps |
| 分支（beq） | 200ps | 100ps | 200ps | | | 500ps |

图 4-26 根据各功能单元所需时间计算出来的每条指令的总执行时间。假设多选器、控制单元、PC 访问和符号扩展单元都没有延时

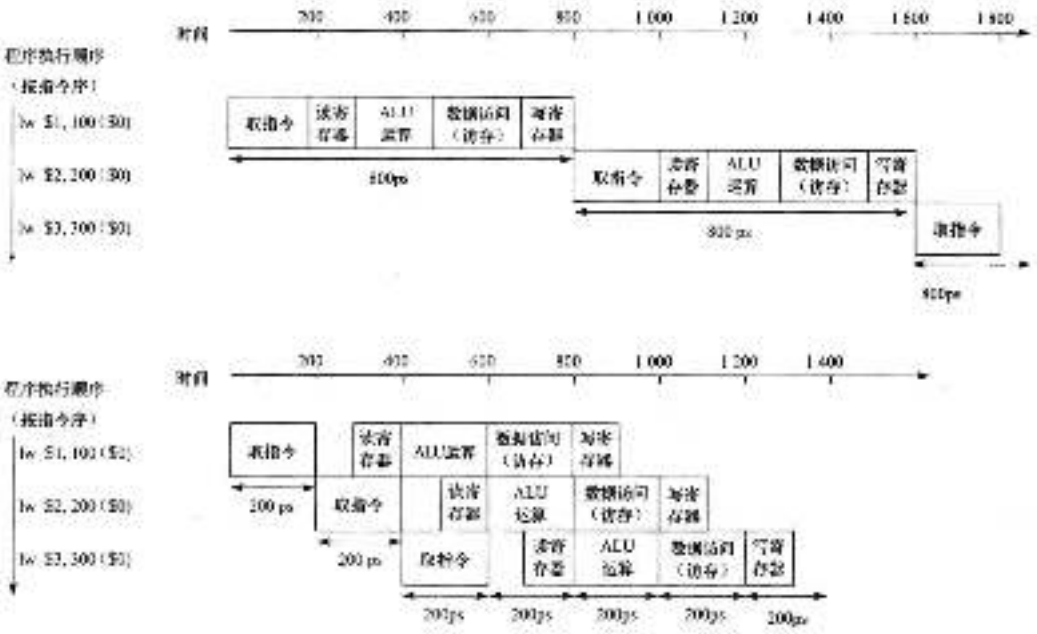



图 4-27 单周期、非流水线的指令执行过程（上图）与流水线的指令执行过程（下图）。两者采用相同的功能单元，各功能单元的处理时间如图 4-26 所示。在这种情况下，指令的执行速度提高了 4 倍，即从 800ps 降到了 200ps。将本图与图 4-25 比较：在洗衣服的例子中，我们假设所有步骤需要的处理时间都是相等的。如果烘干机运行得最慢，那么就把烘干的时间设定为一个步骤需要的处理时间。计算机流水线的处理时间也受限于最慢的处理步骤，即 ALU 操作和存储器访问。同时我们假设对寄存器堆的写操作发生在时钟周期的前半段，对寄存器堆的读操作发生在时钟周期的后半段，本章后面将一直遵循这个假设 □

所有的流水级（pipeline stage）都只花费一个时钟周期的时间，因此，时钟周期必须能够满足最慢操作的执行需要。这就像在单周期模型中虽然有些快的指令的执行只需要 500ps，但它必须选择在最坏情况下的 800ps 作为时钟周期一样，流水线执行模型的时钟周期也必须选择最坏情况下的 200ps 而不是有些步骤可以达到的 100ps。流水线能够将性能提高 4 倍：第一与第四条指令之间的时间差距缩短为 $3 \times 200\text{ps} = 600\text{ps}$。

我们可以把上面讨论的流水线模型能够获得的性能加速比归纳成一个公式。如果流水线各阶段操作平衡，那么在流水线机器上的指令执行时间为（在理想情况下）：

$$\text{指令执行时间}_{\text{流水线}} = \text{指令执行时间}_{\text{非流水线}} / \text{流水线级数}$$

即在理想情况且有大量指令的情况下，流水线所带来的加速比与流水线的级数近似相同。例如一个 5 级流水线能获得的加速比接近于 5。

这个公式说明一个 5 级流水线在 800ps 的非流水线执行时间的基础上获得接近 5 倍的速度提高，即相当于 160ps 的时钟周期。然而，在例子中显示，各级间并不是完全平衡的。另外，流水线引入了一些开销，开销的来源问题稍后会更加清楚。所以，在流水线机器中每一条指令的执行时间会超过这个最小的可能值，因此流水线能够获得的加速比也就小于流水线的级数。

此外，即使我们在前面的分析中断言能将指令的执行速度提高 4 倍，但在本例中并没有反映出来，它实际获得的加速比为 2 400ps/1 400ps，这是因为执行指令的数量不够多。如果增加执行指令的数目将会发生什么呢？我们首先将前面图中的指令增加到 1 000 003 条，也就是说，在上面的流水线例子中加入 1 000 000 条指令，每一条指令都将会使整个的执行时间增加 200ps，因此，整个的执行时间就变成 $1\,000\,000 \times 200\text{ps} + 1400\text{ps}$，即 200 001 400ps。在非流水线的例子中，我们也加入 1 000 000 条指令，每条指令的执行时间是 800ps，因此整个的执行时

间为 1 000 000 × 800ps + 2 400ps，即 800 002 400ps。在这些条件下，非流水线程序与流水线程序的实际执行时间的比值就非常接近于两者指令平均执行时间的比值，即为

275 ↑ 276

$$800\,002\,400\text{ps}/200\,001\,400\text{ps} \approx 800\text{ps}/200\text{ps} \approx 4.00$$

流水线所带来的性能提高是通过增加指令的吞吐率，而不是减少单条指令的执行时间实现的。由于实际程序都会执行成千上万条指令，因此，指令的吞吐率是一个很重要的参数。

### 4.5.1 面向流水线的指令集设计

尽管上面的例子只对流水线进行了最简单的说明，我们也能够通过它讨论面向流水线执行的 MIPS 指令集的设计。

第一，所有的 MIPS 指令的长度都是相同的。这一限制简化了流水线的第一级取指与第二级译码。在诸如 x86 之类的指令集中，指令的长度并不相同，从 1 字节到 15 字节不等，这样将会给流水线的执行带来更大的挑战。最近的 x86 体系结构实现实际上是将 x86 指令转化成类似 MIPS 指令的简单操作，然后再将这些简单操作进行流水，而不是直接对原始的 x86 指令流水！（见 4.10 节）

第二，MIPS 只有很少的几种指令格式，并且每一条指令中的源寄存器字段位置都是相同的。这种对称性意味着流水线的第二级在确定取指类型的同时就能够开始读寄存器堆。如果 MIPS 的指令格式是非对称的，我们就需要将第二级一分为二，从而使得流水线的级数变为 6。稍后我们将看到长流水线的缺点。

第三，MIPS 中的存储器操作数仅出现在存取指令中。这一限制意味着可以利用执行级计算存储器地址，然后可以接着在下一级访问存储器。如果可以直接操作内存中的操作数（就像在 x86 中那样），那么第三级与第四级将会扩展为地址计算、存储访问和执行阶段。

第四，如第 2 章所述，所有操作数必须在存储器中对齐。因此，我们不需要担心一个数据传输指令需要访问两次存储器的情况，所请求的数据可以在一级流水线内在处理器与存储器之间完成传输。

### 4.5.2 流水线冒险

流水线有这样一种情况，在下一个时钟周期中下一条指令不能执行。这种情况称为冒险（hazard）。我们将介绍三种流水线冒险。

#### 1. 结构冒险

第一种冒险叫作**结构冒险**（structural hazard）。即硬件不支持多条指令在同一时钟周期执行。在洗衣店的例子中，如果用洗衣烘干一体机代替独立的洗衣机与烘干机，或者如果你的室友正在做其他的事情而不能帮助你将衣服收拾好，都会发生结构冒险。如果发生上述情况，那我们精心构筑起来的流水线就会受到破坏。

277 ➲ 结构冒险：因缺乏硬件支持而导致指令不能在预定的时钟周期内执行的情况。

正如我们在上面所说的那样，MIPS 的指令集是为流水线设计的。因此，它就使得设计者在设计流水线时能够非常容易地避免结构冒险。假设图 4-27 的流水线结构只有一个存储器而不是两个存储器，那么如果有第四条指令的话，第一条指令在访问存储器的同时第四条指令将会在同一存储器中预取指令，流水线就会发生结构冒险。

#### 2. 数据冒险

**数据冒险**（data hazard）发生在由于一条指令必须等待另一条指令的完成而造成流水线暂停的情况下。假设你在折叠衣服时发现有一只短袜找不到与之配对的另一只。你可能做的是下

楼到你的房间，在衣橱中找，看是否能找到另一只。很明显，当你在找的时候，已经烘干且正需要折叠的衣服以及已经洗完且正需要烘干的衣服不得不捆置一边。

⇒ 数据冒险：也称为流水线数据冒险（pipeline data hazard），即因无法提供指令执行所需数据而导致指令不能在预定的时钟周期内执行的情况。

在计算机流水线中，数据冒险是由于一条指令依赖于更早的一条还在流水线中的指令造成的（这是一种在洗衣店例子中不存在的情况）。例如，假设有一条加法指令，它之后紧跟着一条减法指令，而减法指令要使用加法指令的和（$s0）：

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

在不做任何干涉的情况下，这一数据冒险会严重地阻碍流水线。加法指令直到第五步才能写回它的结果，这就意味着在流水线中浪费了三个时钟周期。

虽然可以试图通过编译器来避免这种数据冒险的发生，但实际上这种努力很难令人满意。因为这种冒险的发生过于频繁而且导致的延迟太长，因此不可能指望编译器把我们从这种困境当中解脱出去。

一种最基本的解决方法是基于以下发现：在解决数据冒险问题之前不需要等待指令的执行结束。对于上述的代码序列，一旦 ALU 生成了加法运算的结果，就可以将它用作减法运算的一个输入项。从内部资源中直接提前得到缺少的运算项的过程称为前推（forwarding）或者旁路（bypassing）。

⇒ 前推：也称为旁路。一种解决数据冒险的方法，具体做法是从内部寄存器而非程序员可见的寄存器或存储器中提前取出数据。

**例题** 两条指令间的旁路

对于上述的两条指令，说明如何使用旁路将流水线各级连接起来。图 4-28 描述了流水线的五级，与图 4-25 中的洗衣店流水线类似，每条指令的数据通路排成一行。 278

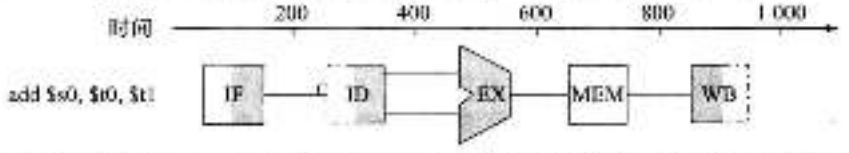



图 4-28 指令流水线的图形表示，其与图 4-25 中的洗衣店流水线类似。本图以及本章均使用图形符号来代表流水线各级使用的物理资源。这些符号在五级流水线中所代表的意义分别是：IF 表示取指阶段，其外方框表示指令的存储器；ID 表示指令的译码或寄存器堆的读取阶段，外边的虚线方框表示要读取的寄存器堆；EX 表示指令的执行阶段，外边的图符表示 ALU；MEM 表示存储器访问阶段，包围它的方框代表数据存储器；WB 表示写回阶段，包围它的虚线方框代表被写回的寄存器堆。阴影表示该资源被指令所使用。因为 add 指令在这一步并不读取数据存储器，所以 MEM 没有阴影。寄存器堆或存储器右半边的阴影表示它们在此步骤中被读取，左半边的阴影表示它们在此步骤中被写入。因此，由于第二步需要读取寄存器堆，ID 的右半边有阴影，而由于第五步中需要写入寄存器堆，WB 的左半边有阴影

**答案**

图 4-29 表示了把 add 指令执行后的 $s0 中的值作为 sub 指令执行的输入的旁路连接。□

在图 4-29 中，只有当目标步骤在时间上晚于源步骤时旁路的路径才有效。例如，从前一条指令存储器访问的输出至下一条指令执行的输入就不能实现旁路，因为那样的话将意味着时间的倒流。

旁路可以工作得很好，其具体内容将在 4.7 节详细介绍。然而它并不能够避免所有流水线阻塞的发生。例如，假设第一条指令不是 add 而是装载 $s0 寄存器的内容，正如图 4-29 所描 279

述的那样，由于数据间的依赖，所需要的数据只有在前一条指令流水线首第四级完成之后才能生效，这对于 sub 指令的第二级输入来说就太迟了。因此，如图 4-30 所示，即使采用了旁路机制，在遇到**取数－使用型数据冒险**（load-use data hazard）时，流水线不得不阻塞一个步骤。图中显示了一个重要的流水线概念，正式的叫法是**流水线阻塞**（pipeline stall），但是它经常被形象地称为**气泡**（bubble）。我们经常会在流水线中看到阻塞的发生。4.7 节将给出处理这种复杂情况的方法，即采用硬件上检测阻塞和软件上重新安排代码顺序等方法来避免装载－使用型数据冒险。

- **取数－使用型数据冒险**：一类特殊的数据冒险，指当装载指令要取的数还没取回来时其他指令就需要使用的情况。
- **流水线阻塞**：也称为气泡。为了解决冒险而实施的一种阻塞。

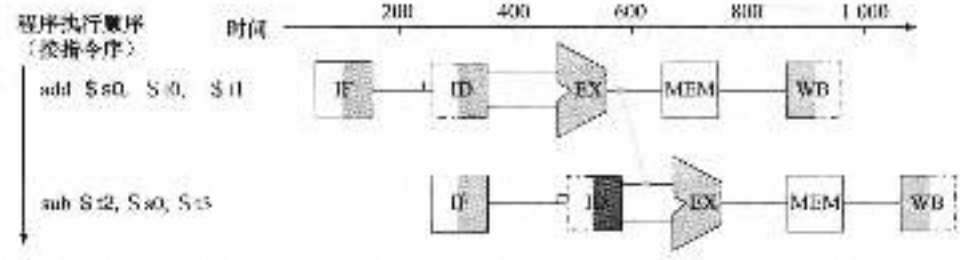



图 4-29 旁路的图形表示。图中的连接表示从 add 指令的 EX 操作输出到 sub 指令的 EX 操作输入的旁路路径，从而替换掉在 sub 的第二步从寄存器 $s0 读取的值

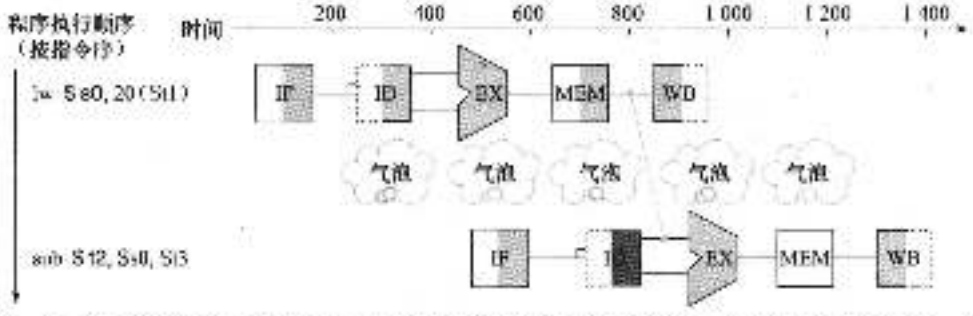



图 4-30 当一条 R 型指令之后紧跟着一条需要使用其结果的装载指令时，即使使用了旁路机制，仍然会产生一次阻塞。如果不进行一次阻塞，从存储器访问的输出到执行级的输入之间的路径在时间上将是倒着的，这显然是不可能的。事实上，这仅是一个示意图，因为直到减法指令取指和译码之后，我们才知道是否需要阻塞。4.7 节详细介绍了这种冒险情况

**例题：重新安排代码以避免流水线阻塞**

考虑下面这段 C 代码：

```
a = b + e;
c = b + f;
```

下面是这段 C 代码对应的 MIPS 指令，假设所有的变量都在存储器中，且以 $t0 为基址进行寻址：

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1,$t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1,$t4
sw   $t5, 16($t0)
```

280

试找出上述代码段中存在的冒险并试着重新安排指令顺序以避免流水线阻塞。

**例1** 答案

两条 add 指令都存在冒险，因为它们都依赖于上一条 lw 指令。注意，通过旁路可以消除一些潜在的冒险，包括第一条 add 指令对第一条 lw 指令的依赖和 sw 指令导致的冒险。而将第 3 条 lw 指令上移到第 3 条指令的位置则可以进一步消除所有冒险：

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1,$t2
sw   $t3, 12($t0)
add  $t5, $t1,$t4
sw   $t5, 16($t0)
```

在一个具有旁路功能的流水线处理器中，执行这个重排序后的指令序列要比上面那个指令序列快 2 个时钟周期。 □

在前面介绍了设计适应流水线的 MIPS 体系结构指令集的 4 个原则，由旁路可以得到设计 MIPS 体系结构指令集的另一个原则：即每条 MIPS 指令最多只写一个结果并且在流水线的最后一级执行。如果每条指令要写多个结果或写在流水线更早阶段进行则旁路设计要复杂得多。

**例1** 精解 “前推”这个名称来源于将结果从前面的指令直接发送到后面的指令的思想。“旁路”这个名称来源于把寄存器堆中的结果直接传递到需要的单元中。

### 3. 控制冒险

第二种冒险叫作控制冒险（control hazard）。这种冒险会在下面的情况下出现：决策依赖于一条指令的结果，而其他指令正在执行中。

➲ 控制冒险：也称为分支冒险（branch hazard），因为取到的指令并不是所需要的（或者说指令地址的变化并不是流水线所预期的）而导致指令不能在预定的时钟周期内执行。

假设洗衣店的店员们接到了一个令人高兴的任务：为一个足球队清洗队服。由于衣服非常脏，我们需要确定清洗剂的用量以及设置水温以保证能够将衣服清洗干净，但同时要保证清洗剂的用量不能过大，以避免过度磨损衣物。在洗衣店流水线中，店员只有等到第二步烘干衣服以后才能确定是否需要改变设置。在这种情况下应该怎么办呢？

有两种办法可以解决洗衣店的控制冒险，同样的方法也可以应用到计算机中。

阻塞（stall）：在第一批衣服被烘干之前按串行的方式操作，并且重复这一过程直到找到正确的洗衣设置为止。

这种保守的方法当然可以保证正常工作，但它的速度比较慢。 281

计算机中的决策就是分支指令。注意，在取分支指令之后，紧跟着就会取下一条指令。但是流水线并不知道下一条真正要执行的指令在哪里，因为它才刚刚从指令存储器中把分支指令给取出来！跟洗衣店的例子一样，一种可能的解决方法是取分支指令后立即阻塞流水线，直到流水线确定分支指令的结果并知道下一条真正要执行的指令在哪为止。

假设可以加入足够多的硬件使得在流水线的第二级能测试寄存器、计算分支地址并更新 PC（详情见 4.8 节）。通过使用这些额外的硬件，包含条件分支的流水线执行情况如图 4-31 所示。如果分支指令的判定条件失败，即分支不执行，则要执行 lw 指令，它在启动之前被阻塞一个 200ps 的额外时钟周期。

**例1** 例题：阻塞对分支性能的影响

评价分支阻塞对单位指令时钟周期数（CPI）的影响。假设其他所有指令的 CPI 都为 1。

**例1** 答案

第3章的图3-27说明在SPECint2006中，分支指令约占执行指令的17%。由于其他指令的CPI都为1，而分支指令阻塞要多一个时钟周期，因此平均CPI为1.17。与理想的情况相比，现在的速度下降了1.17倍。 □



图4-31 在每一个条件分支上阻塞是避免流水线控制冒险的一种解决方法。这个例子假设分支发生，并且分支目标地址处是一条OR指令。分支指令之后会插入一个周期的流水线阻塞，或者叫气泡。事实上，产生阻塞的过程有些复杂，我们会在4.8节说明这一点。这种方法对性能的影响与插入一个气泡是一样的

282

如果不能在第二级解决分支问题（这种情况在较长的流水线中经常发生），那么分支结构上的阻塞将导致更大的速度下降。对很多计算机来说，这种阻塞的方法代价太大。因此也就产生了另外一种消除控制冒险的方法（该方法使用第1章中提到的伟大思想来应对控制冒险）：

预测（predict）：如果你有自信正确地设置洗衣设备来洗涤那些队服（可以预测它的正确工作条件），那么就可以在第一批衣服烘干的同时清洗第二批衣服。

这种做法在预测正确的时候不会降低流水线的速度，但是一旦预测错误，就不得不将已经洗过的队服重新洗一遍。

计算机的确是采用预测的方法来处理分支的。一种简单的预测方法就是总预测分支未发生。当预测正确（分支未发生）的时候，流水线会全速地执行。只有当分支发生时流水线才会阻塞。图4-32给出了这样一个例子。

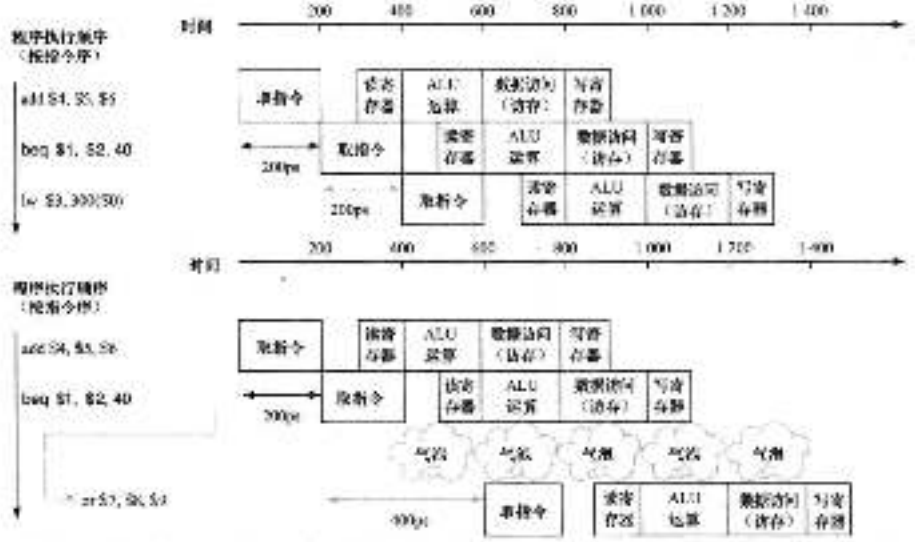


图4-32 预测分支未发生是一种避免流水线控制冒险的解决方法。上图显示的是分支未发生的流水线，下图显示的是分支发生了的流水线。正如我们在图4-31中提到的那样，这种插入气泡的方式是一种简化的表示方法，至少对紧跟分支指令的下一个时钟周期而言是这样。4.8节将给出其中的细节

283

一种更加成熟的**分支预测**（branch prediction）方法是预测一些分支发生而预测另一些分支不发生。如在上面洗衣店的例子中，夜晚和主场比赛的队服使用一个洗衣设备设置，而白天或客场比赛的队服则使用另一个设置。在计算机程序中，循环体底部的分支总是会跳回到循环体的顶部。在此种情况下，由于分支总是发生并且向前跳转，因此我们可以预测分支会跳转到前面的某一地址处。

⊙ 分支预测：一种解决分支冒险的方法。它预测分支结果并立即沿预测方向执行，而不是等真正的分支结果确定后才开始执行。

这种分支预测的方法依赖于始终不变的行为，它没有考虑特定分支指令的特点。动态硬件预测器与这种方法截然不同，它的预测取决于每一条分支的行为，并且在整个程序生命期内可能改变分支的预测结果。用洗衣店的例子来说，使用动态预测方法，店员将会观察衣服脏的程度并预测一个洗衣设备的设置，然后在本次预测成功的基础上调整下一次的预测行为。

计算机中动态预测方法的一种比较普遍的实现方式是保存每次分支的历史记录，然后利用这个历史记录来预测。稍后我们将看到，历史记录的数量和类型足够多时，这种动态分支预测的方式能够达到90%的正确率（见4.8节）。当预测错误时，流水线控制必须确保被错误预测的分支后面的指令执行不会生效，并且必须在正确的分支地址处重新开始启动流水线。在洗衣店的例子中，我们必须停止接受新的任务，从而可以重新执行错误预测的任务。

如同其他解决控制冒险的方法一样，较长的流水线会恶化预测的性能，并会提高错误预测的代价。控制冒险的解决办法在4.8节中将有更加详细的描述。

01 **精解** 还有一种解决控制冒险的方法，即延迟决定（delayed decision）。与洗衣店的例子类比，每当要决定如何洗衣服时，就将一批非足球队的衣服放进洗衣机里，同时等待足球队的制服被烘干。只要有足够多不需要决策的脏衣服，这种方法就很有效。

在计算机中这种方法被称为延迟分支（delayed branch），在MIPS体系结构中也得到了实际应用。延迟分支顺序执行下一条指令，在一条指令延迟之后再开始执行分支。由于编译器会自动排列指令，使得分支的行为达到程序员的要求，因此这个过程对MIPS的汇编程序员是透明的。MIPS编译器会在延迟分支指令的后面紧跟着放一条不受该分支影响的指令。发生了的分支会改变这条安全指令之后的指令地址。在我们的例子中，图4-31中分支前的add指令不影响分支，所以可以把它移到分支之后以完全隐藏分支延迟。因为只有当分支延迟较短时，延迟分支才有效，所以没有处理器使用超过一个时钟周期的延迟分支。对更长的分支延迟，一般都使用硬件分支预测器。 284

## 4.5.3 对流水线概述的小结

流水线是一种在顺序指令流中利用指令间并行性的技术。与多处理器编程相比，其优势在于它对程序员是不可见的。

在以下几节中，我们首先使用4.4节单周期实现方式的MIPS指令子集及其简化的流水线方式介绍关于流水线的一些基本概念，然后讨论引入流水线所带来的一些问题以及流水线在一些典型情况下所能获得的性能提升。

如果想了解更多软件和流水线对性能的意义，并且你已经具有足够的背景知识，可以直接跳到4.10节。4.10节介绍了一些高级流水线概念，如超标量、动态调度等。4.11节介绍了一些最新的微处理器流水线。

反之，如果你想深入了解流水线的实现方式和如何处理冒险现象，可以接着阅读后面的几节。4.6节介绍了一个流水线的数据通路和基本控制设计。在4.6节的基础上，你可以在4.7

节中学习旁路和阻塞的实现。紧接着4.8节介绍了处理分支冒险的方法。而4.9节则介绍了异常是如何处理的。

**小测验**

对下面每个指令序列，说明哪个必须阻塞，哪个只使用旁路就可以避免阻塞，而哪个既不需要阻塞也不需要旁路就可以执行。

| 指令序列1 | 指令序列2 | 指令序列3 |
| --- | --- | --- |
| lw $t1,0($t0)<br>add $t1,$t0,$t1 | add $t1,$t0,$t2<br>addi $t2,$t0,#5<br>addi $t4,$t1,#5 | addi $t1,$t0,#1<br>addi $t2,$t0,#2<br>addi $t3,$t0,#3<br>addi $t4,$t0,#4<br>addi $t5,$t0,#5 |

**理解程序性能** 除了存储系统以外，流水线的有效运作是决定处理器CPI乃至其性能最重要的因素。正如我们将在4.10节看到的那样，理解现代多发射流水线处理器的性能是一项复杂的任务，相对简单流水线处理器而言需要理解更多的问题。不管怎样，结构冒险，数据冒险和控制冒险在简单流水线处理器和更复杂的流水线处理器中都是非常重要的。

对现代流水线而言，结构冒险经常出现在浮点单元附近。浮点单元是一个几乎不可能完全流水的地方。与之对比，控制冒险一般出现在整数程序中，因为其中分支出现的概率更高，也更难预测。数据冒险在整数和浮点程序中都可能成为性能瓶颈。一般来说，浮点程序中的数据冒险更容易处理，因为低的分支出现频率和规则的存储器存取使得编译器有更大的空间调度指令以避免冒险。与之相比，在整数程序中涉及大量的指针，存储器的存取更不规则，做这样的优化就要困难一些。正如我们将在4.10节看到的那样，有很多编译器和基于硬件的技术通过试图来减少数据间的依赖。

285

**重点** 流水线增加了同时执行的指令数目以及指令开始和结束的速率。流水线并不能够减少单一指令的执行时间，也称为延迟（latency）。例如，一个五级流水线仍然需要5个时钟周期来完成一条指令。同第1章的术语来描述就是流水线提高了指令的吞吐率而不是减少了单条指令的执行时间或延迟。

⇨ 延迟：流水线的级数或者顺序执行过程中两条指令间的级数。

对流水线的设计者来说，指令集既可能将事物简单化，也可能将事物复杂化。流水线设计者必须解决结构冒险、控制冒险和数据冒险。而分支预测、旁路和阻塞机制能够在保证得到正确结果的前提下提高计算机的性能。

## 4.6 流水线数据通路及其控制

*看起来容易很多，其实不然。*

——*Tallulah Bankhead, remark to Alexander Woollcott, 1922*

图4-33是摘自4.4节的一个单时钟周期的数据通路。将指令划分为5个阶段意味着一个流水线采用5级，也就意味着在任何一个单时钟周期内，最多会执行5条指令。因此必须把数据通路分为5个部分，每一部分用与之对应的指令执行阶段来命名。

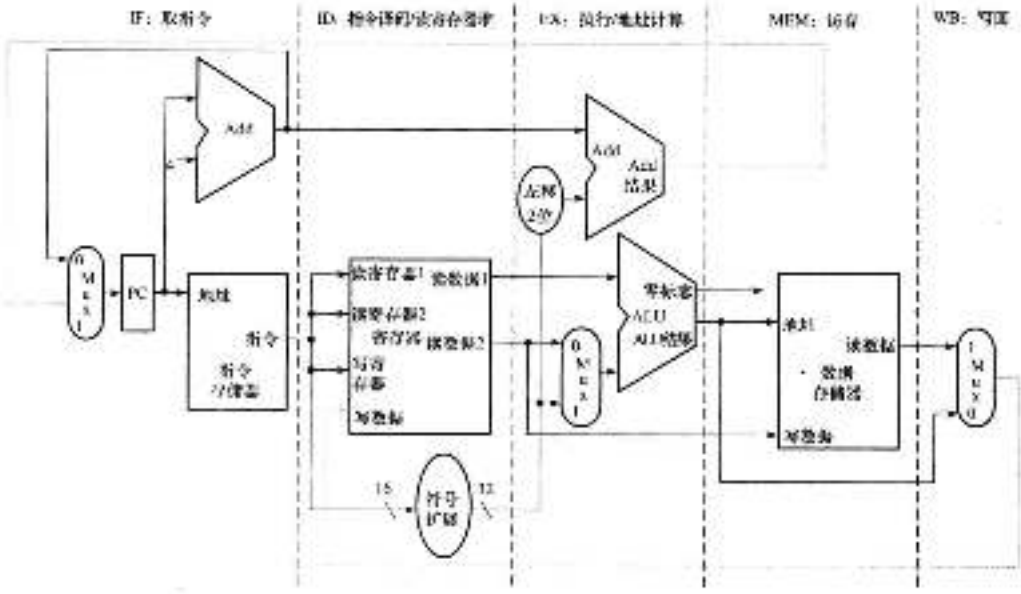



图4-33 4.4节中的单时钟周期数据通路（与图4-17类似）。图中自左至右把指令的每一步映射到数据通路中。PC更新与写回过程是唯一的例外（图中用灰色线表示），其发送ALU结果或存储器数据到左边的寄存器堆中。（我们通常使用灰色线表示控制，但在这里表示数据线。）

1）IF：取指令。

2）ID：指令译码，读寄存器堆。

3）EX：执行或计算地址。

4）MEM：数据存储器访问。

5）WB：写回。

图4-33的5个部分大致与数据通路相符：指令与数据随着执行过程从左到右依次通过五级流水线。正如洗衣店的例子一样，衣服沿着一条工作线依次完成清洗、烘干和整理，而不会反向移动。 [286]

然而，在从左到右的指令流中有两个例外：

- 写回阶段，它把结果写回数据通路中间的寄存器堆中。
- 选择PC的下一个值时，需在自增的PC和MEM级的分支地址间进行选择。

这两个从右向左的数据流不会影响当前指令；只有当前指令以后的指令才会受到这种数据反向活动的影响。需要注意的是，第一个例外会导致数据冒险，而第二个会导致控制冒险。

一种表示流水线数据通路的方法是假定每一条指令都有它独立的数据通路，然后把这些数据通路放在同一时间轴上表示出它们之间的关系。图4-34在同一时间轴上表示了图4-27中指令执行过程中各自的数据通路（我们仍然使用图4-33中的格式来表示图4-34中的关系）。

从表面来看，图4-34中的三条指令似乎需要三条数据通路。事实上，通过增加保存中间数据的寄存器，使得在指令执行过程中可以共享部分数据通路。

例如，如图4-34所示，指令存储器只在每条指令的5个步骤中的一步中用到，因此我们允许它在其他4步中被其他的指令共享。为了在其他4步中保持指令的值，从指令存储器中读出的数据必须保存在寄存器中。将同样的方法应用到每个流水线级中，我们需要在图4-33中各级间有分割线的地方都加入寄存器。再回到洗衣店的例子中，这里可以用篮子在两个步骤间存放下一步的衣服。

图4-35描述了流水线的数据通路，其中流水线寄存器用灰色表示。在每个时钟周期中所

有指令都会从一个流水线寄存器传递到另一个流水线寄存器中。寄存器以被该寄存器分开的两个阶段来命名，如 IF 和 ID 之间的流水线寄存器叫作 IF/ID。

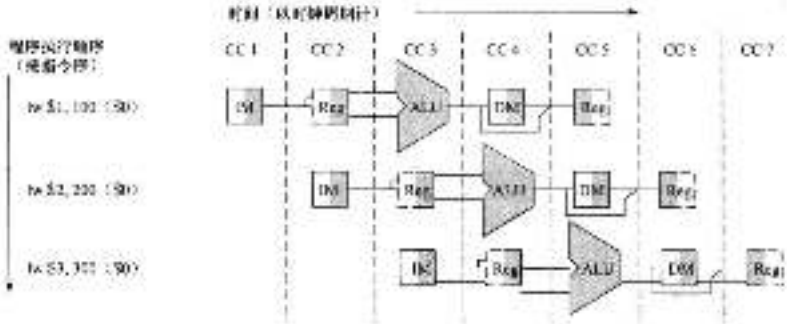


图 4-34 按图 4-33 中的单时钟周期数据通路执行的指令（假定以流水线方式执行）。与图 4-28 到图 4-30 类似，本图假设每一条指令有它独立的数据通路，并根据使用情况将相应的部分涂上阴影。与这些图不同的是，流水线的每一级都用该级使用的物理资源标示，分别对应图 4-33 中数据通路的相应部分。IM 表示指令存储器与取指令阶段的 PC，Reg 表示指令译码/寄存器堆读取阶段（ID）的寄存器堆和符号扩展单元，等等。为了保持正确的时序，这种形式的数据通路把寄存器堆从逻辑上划分为两个部分：寄存器读取（ID）阶段的寄存器读和写回（WB）阶段的寄存器写。这种复用在图中表示为：在 ID 级当寄存器堆没有被写入时，将没有阴影的寄存器堆的左半部分用虚线表示；而在 WB 级，当寄存器堆没有被读取时，将没有阴影的右边部分用虚线表示。与以前一样，假设在时钟周期的前半部分写寄存器堆而在时钟周期的后半部分读寄存器堆

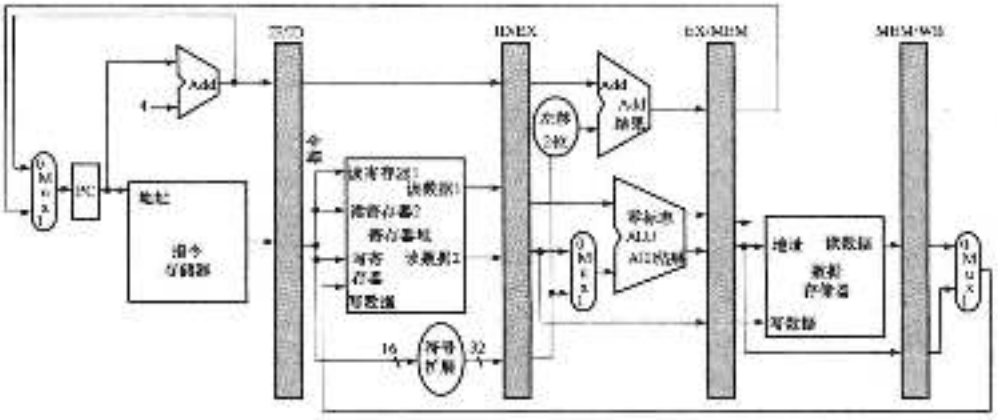


图 4-35 图 4-33 数据通路的流水线版本。流水线寄存器（以灰色标识）将流水线的各部分分开。例如，IF/ID 将取指令和指令译码阶段分开。为了存储所有穿过它的数据（用线条表示），寄存器的宽度必须足够大。例如，因为 IF/ID 寄存器必须同时保存从存储器中读取出来的 32 位指令及 32 位 PC 自增地址，所以它的宽度必须是 64 位。我们将在本章中逐渐增加寄存器宽度，目前另外三个流水线寄存器的宽度分别是 128 位、97 位和 64 位

需要注意的是，在写回阶段的后面没有流水线寄存器。所有指令都会更新机器中的某些状态，如寄存器堆、存储器或 PC 等，因此各个流水线寄存器对于更新后的状态来说是多余的。例如，装载指令会把它的结果放入 32 个寄存器中的某一个，以后任何需要此数据的指令只需要读取相应的寄存器就可以了。

当然，每条指令都会更新 PC，不管是自增还是设置为分支目的地址。PC 可以看成一个流水线寄存器，给流水线的 IF 级提供数据。不像图 4-35 中那些灰色的流水线寄存器，PC 是可见

体系结构寄存器的一部分，发生异常时必须保存它的内容，而那些流水线寄存器的内容可被丢弃。用洗衣店的例子来说，你可以把 PC 看成洗涤步骤之前装脏衣服的篮子。

为了描述流水线的工作方式，本章将使用一系列图片来表示这些顺序的操作。这些内容需要一定时间去理解，但不要害怕，这些图片实际上比它们看上去要容易理解，因为可以对比观察每一个时钟周期内所发生的变化。4.7 节将介绍流水线指令间发生数据冒险的情况，这里暂时忽略。

287
2
289

图 4-36 ~ 图 4-38 表示了装载指令在通过流水线的五级时数据通路的活动部分。先讨论装载指令是因为它完全使用了流水线的五级。正如图 4-28 ~ 图 4-30 所显示的那样，当寄存器或存储器被读取时，图 4-35 中活动的数据通路部件用灰色表示。在图中用阴影表示其右半部分；而当它们被写入时，用阴影来表示其左半部分。

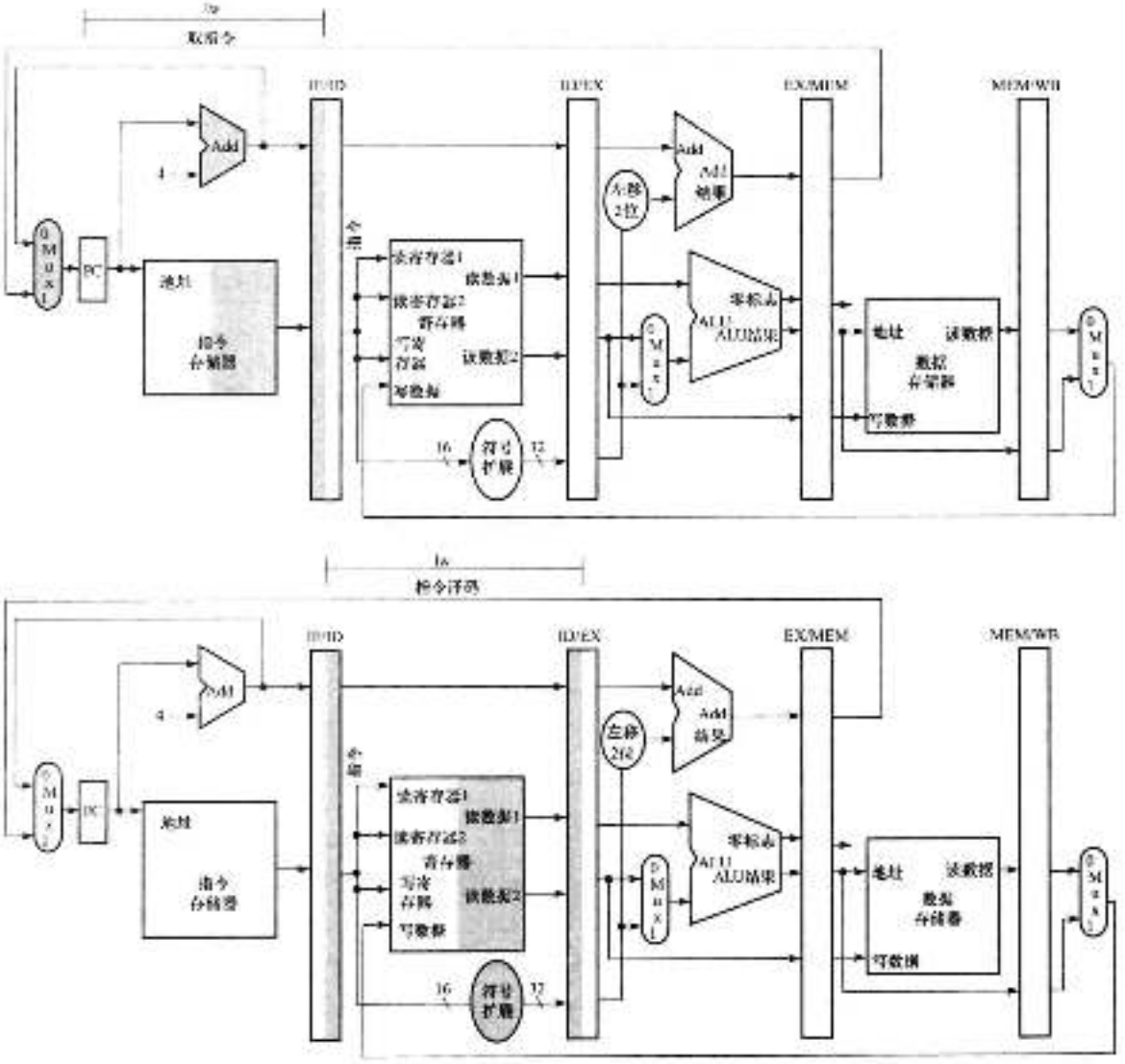



图 4-36 IF 和 ID：一条指令在流水线中的第一、二步。这种灰色的表示方法与图 4-28 相同。正如 4.2 节中介绍的那样，读写寄存器并不会发生冲突，因为寄存器内容的变化只在时钟的边缘发生。虽然 lw 指令只需要第二级中寄存器 1 的值，但由于处理器并不知道当前是哪一条指令正在被译码，因此它把符号扩展后的 16 位常量及两个寄存器的值都读入 ID/EX 流水线寄存器中。我们并不一定需要所有这三个操作数，但是保留全部三个操作数能简化控制

我们把每一幅图中活动的流水线级用指令的缩写 lw 和流水级名称标出。具体情况如下：

1）取指令：图 4-36 的顶端表示指令使用 PC 中的地址从存储器中读取数据，然后将数据放入 IF/ID 流水线寄存器中。PC 地址加 4 然后写回 PC 以便为下个时钟周期做好准备。增加后的地址同时也存入了 IF/ID 流水线寄存器中以备后面的指令使用（如 beq）。计算机并不知道所取指令的类型，所以必须考虑到所有可能的指令，并沿流水线传递所有可能有用的信息。

2）指令译码与寄存器堆的读取：图 4-36 的底部显示的是 IF/ID 流水线寄存器的指令部分，其中包括一个 16 位的立即数（可扩展为带符号的 32 位数）和两个寄存器号（用于读取寄存器）。这三个值和自增的 PC 地址一起存入 ID/EX 流水线寄存器中。这里同样必须传递后面指令可能需要的所有信息。

3）执行或者地址计算：图 4-37 表示装载指令从 ID/EX 流水线寄存器中读取由寄存器 1 传过来的值以及经符号扩展后的立即数，并将 ALU 将它们相加，和值存入 EX/MEM 流水线寄存器中。

4）存储器访问：图 4-38 的顶部表示装载指令使用从 EX/MEM 流水线寄存器中得到的地址读取数据存储器，并将数据存入 MEM/WB 流水线寄存器中。

5）写回：图 4-38 的底部表示了最后一个步骤，即从 MEM/WB 流水线寄存器中读取数据并将它写回图中部的寄存器堆。

290
~
291

对装载指令整个过程的描述表明任何后面的流水线级可能用到的信息必须通过流水线寄存器传递。存储指令也是如此。下面是存储指令的 5 个执行步骤：

1）取指令：利用 PC 中的地址从存储器中读出指令，然后将指令放入 IF/ID 流水线寄存器中。这个步骤发生在指令译码之前，所以图 4-36 中顶端部分既适用于装载指令也适用于存储指令。

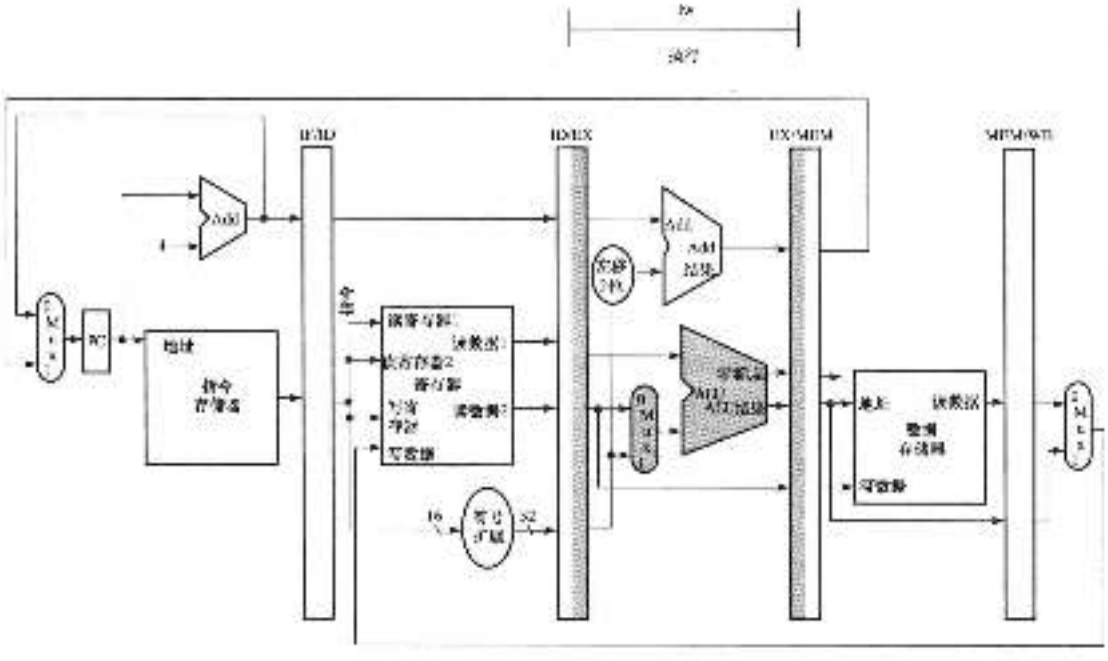



图 4-37 EX：lw 指令在流水线中的第三步，图 4-35 中活动的数据通路部件用灰色表示。将寄存器的值与经过符号扩展的立即数相加，其和放入 EX/MEM 流水线寄存器中

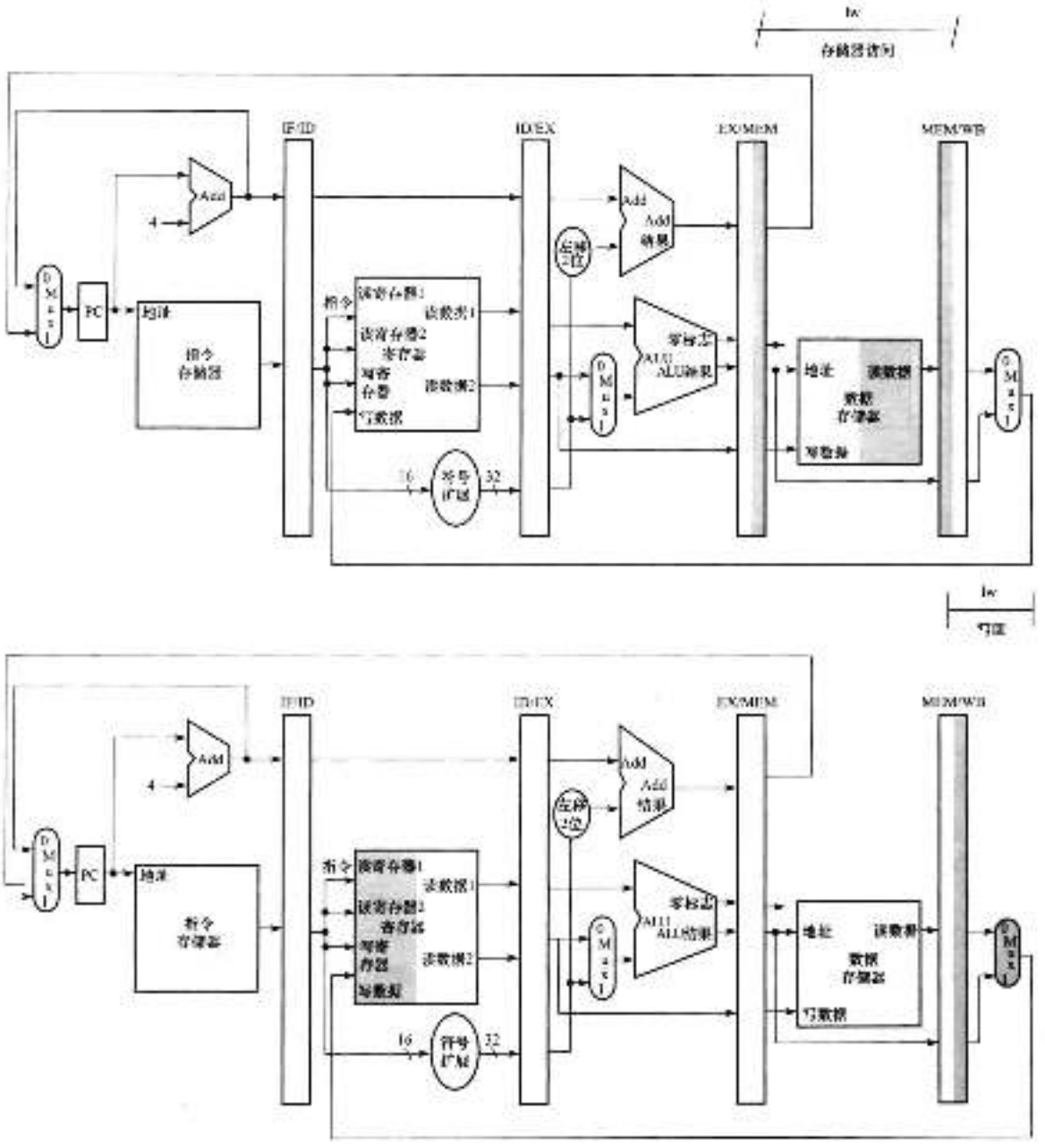



图 4-38 MEM 和 WB：lw 指令在流水线中的第四步和第五步，图 4-35 中活动的数据通路部件用灰色表示。利用 EX/MEM 流水线寄存器中包含的地址读取数据存储器，并将读取的数据放入 MEM/WB 流水线寄存器中，然后从 MEM/WB 流水线寄存器中读取数据写回数据通路中部的寄存器堆。请注意：这里有一个错误，将在后面的图 4-41 中修复

2）指令译码与寄存器堆的读取：IF/ID 流水线寄存器中的指令包括用于读取寄存器的两个寄存器号和用于符号扩展的 16 位立即数。读出的两个寄存器值和符号扩展后的 32 位立即数都存放在 ID/EX 流水线寄存器中。图 4-36 中的底部同时也可描述装载指令的第二个流水级。由于此时并不知道要执行的指令类型，因此所有指令都执行这两个步骤。

3）指令执行或地址计算：图 4-39 描述了 sw 指令在流水线中的第三步，有效地址存放在 EX/MEM 流水线寄存器中。

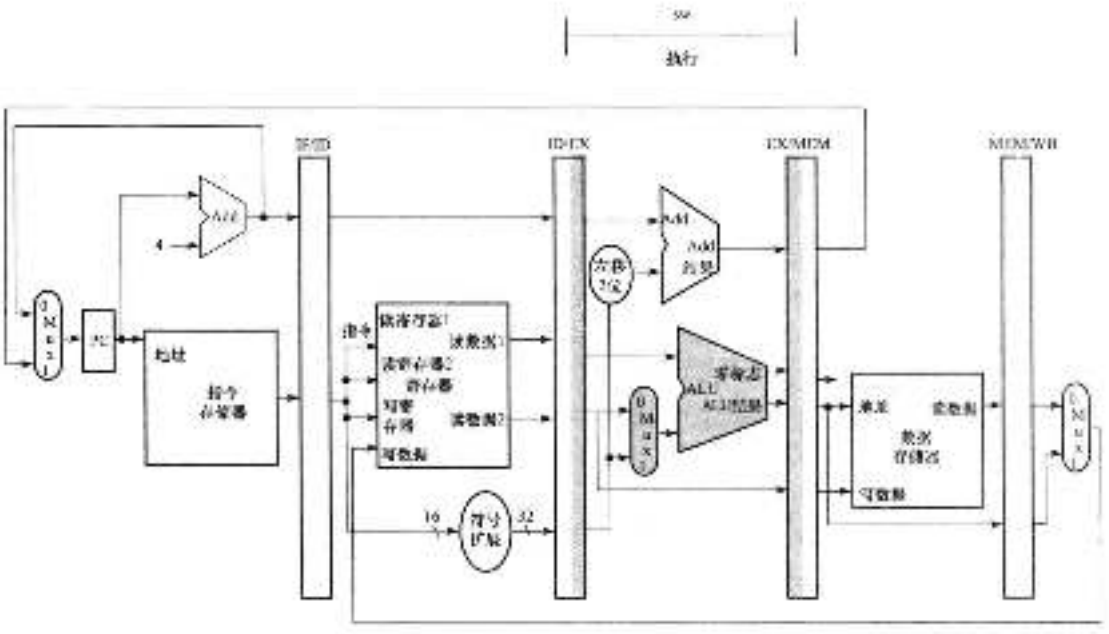



图 4-39 EX：sw 指令在流水线中的第三步。与图 4-37 中装载指令的第三个流水级不同的是，第二个寄存器中的数据被装入 EX/MEM 流水线寄存器中，并被用于下个流水级。虽然总是将第二个寄存器中的数据装入 EX/MEM 流水线寄存器中并不会产生什么不良影响，但为了使流水线更易于理解，我们只在存储指令中才写第二个寄存器的内容

4）存储器访问：图 4-40 的顶端描述的是数据写入存储器的过程。值得注意的是，需要写入存储器的数据在较早的流水级中已经读出并存放在 ID/EX 中。在 MEM 级唯一获得这个数据的方法就是把数据放入 EX 步骤中的 EX/MEM 流水线寄存器中，这一过程与将有效地址放入 EX/MEM 中类似。

5）写回：图 4-40 的底部描述了存储指令的最后一步。存储指令在写回步骤中不做任何事情。由于存储指令后的每一条指令都已经进入流水线中，所以无法加速这些指令。因此，任何一条指令都必须经过流水线的每一个步骤，即使在这个步骤中它实际上什么都没有做，这是因为后面的指令已经按照最大的速率在流水线中进行处理。

存储指令再次说明在流水线中为了从前面的流水级向后面的流水级传递信息，必须将信息放入流水线寄存器中，否则当下一条指令进入该流水级时这些信息将会丢失。在存储指令中，需要将一个寄存器中的内容在 ID 级读出然后在 MEM 级写入存储器。这些数据首先放在 ID/EX 流水线寄存器中，然后传送到 EX/MEM 流水线寄存器中。

取数指令与存储指令的执行过程还表明了另一个重要特性，即数据通路中的每一个逻辑单元（如指令存储器、寄存器读取端口、ALU、数据存储器以及寄存器写入端口）都只能在一个流水级中使用，否则就会产生结构冒险（见 4.5 节）。所以这些单元及其控制可以和一个流水级相联系。

292 ↓ 294

现在我们可以修复图 4-38 中装载指令设计的错误了。你发现这个错误了吗？在装载指令执行的最后一级写回了哪个寄存器呢？更确切地说，哪条指令提供了写寄存器号呢？在 IF/ID 流水线寄存器中的指令提供了写寄存器号，但是很显然现在这条指令已经是装载指令之后的指令了！

因此，我们要在装载指令中保存目的寄存器号。就像存储指令为了 MEM 的需要将寄存器的内容从 ID/EX 传送到 EX/MEM 中一样，为了 WB 级使用的需要，装载指令必须把寄存器号从 ID/EX 经过 EX/MEM 传送到 MEM/WB 中。从另一个角度来考虑寄存器号的传递，为了共享流水线的数据通路，我们需要在 IF 中保存读取的指令，因此每一个流水线寄存器都要保存当

前和后续流水级所需的部分指令。

图 4-41 给出了修正后的数据通路。首先将写寄存器号传送到 ID/EX 寄存器，然后送到 EX/MEM 寄存器，最后送到 MEM/WB 寄存器。在 WB 级使用寄存器号指定了要写入的寄存器。图 4-42 是一个简单的数据通路图，它标出了从图 4-36 到图 4-38 装载指令在所有 5 个流水级中要使用的硬件。阅读 4.8 节可以了解如何使分支指令按期望的方式工作。

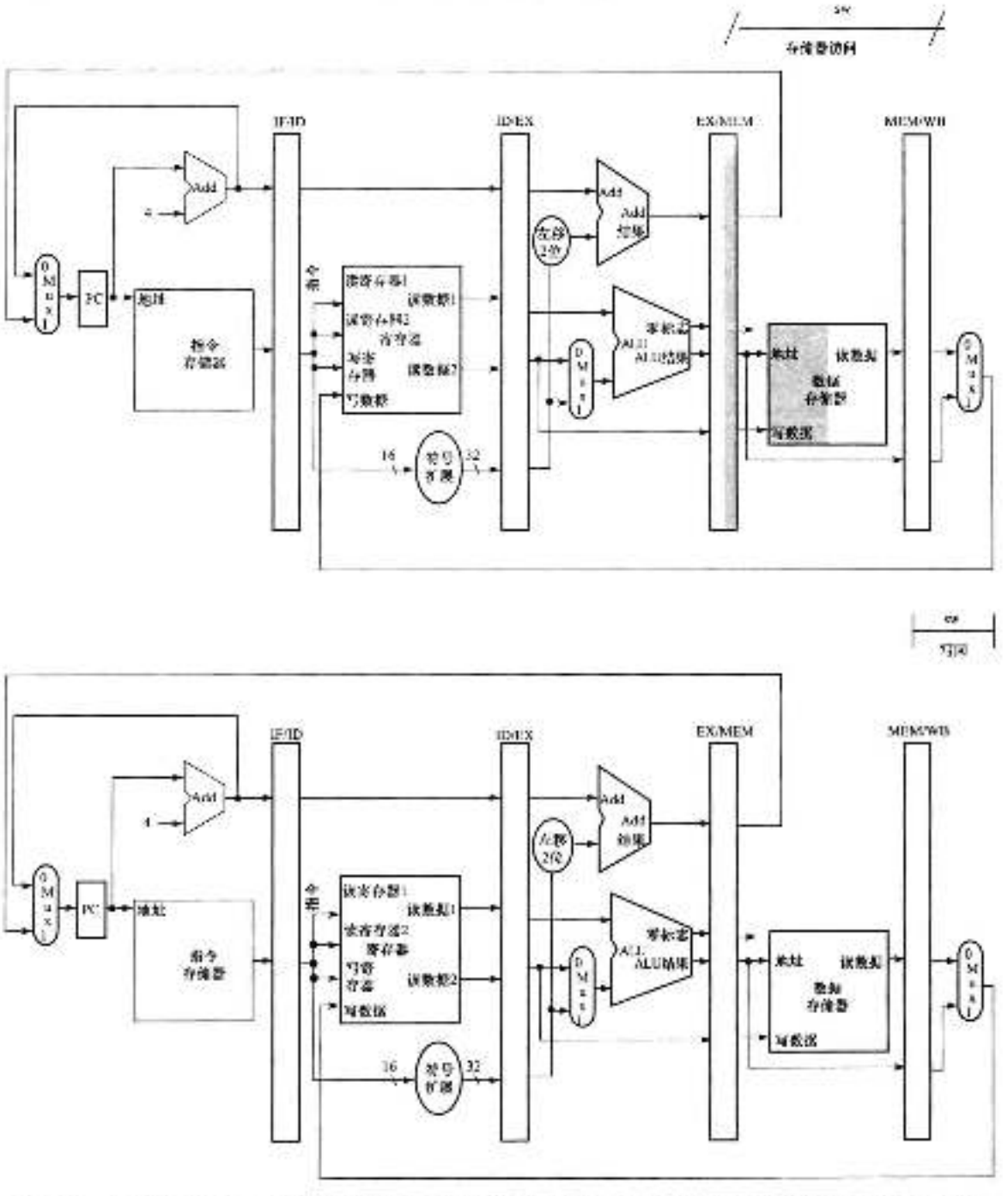



图 4-40 MEM 和 WB：sw 指令在流水线中的第四步和第五步。第四步将数据写入数据存储器中，写入数据来自于 EX/MEM 流水线寄存器。MEM/WB 流水线寄存器没有改变。一旦数据写入存储器，存储指令就没有什么可做的了，所以在第五步中存储指令并不做任何处理

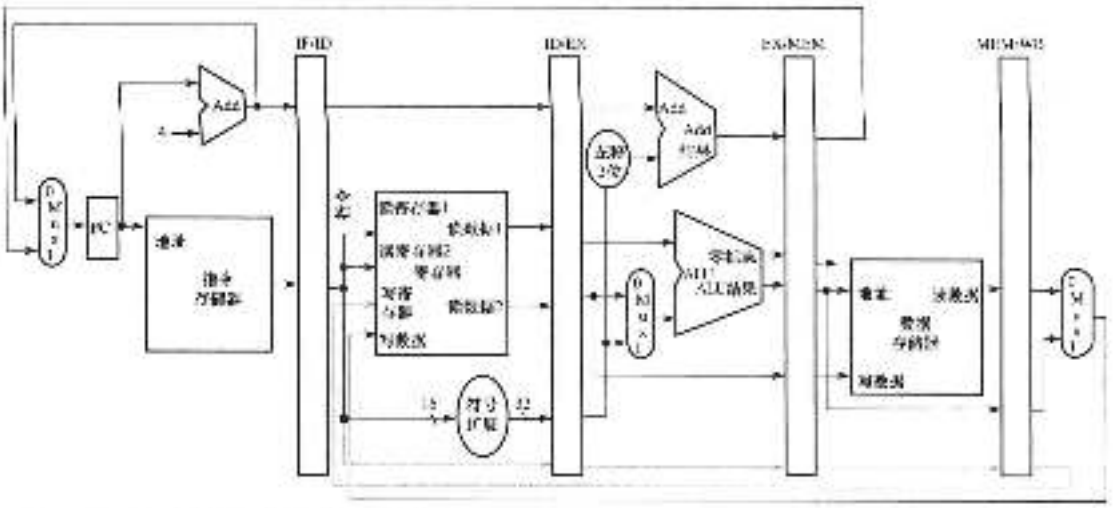



图 4-41 可正确执行读取指令的流水线数据通路。写寄存器号与数据一起从 MEM/WB 流水线寄存器中得到。通过在最后的三个流水线寄存器上分别增加 5 位，寄存器号就能从 ID 流水线一直传送到 MEM/WB 流水线寄存器。新的路径以灰色线标识

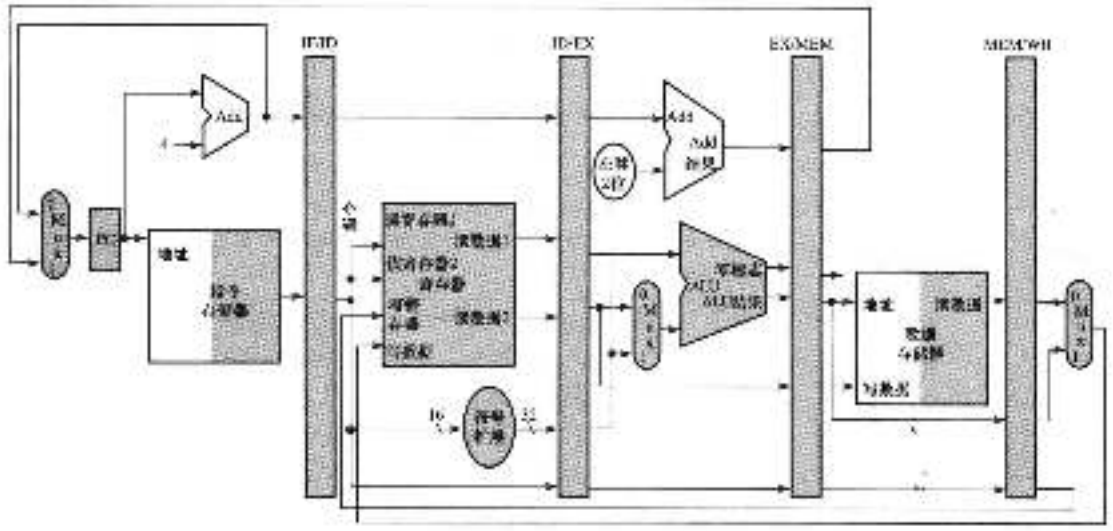



图 4-42 图 4-41 中指令的五级流水线中用到的全部数据通路

## 4.6.1 图形化表示的流水线

295
~
296

流水线技术比较难以理解，因为在每一个时钟周期内同时会有很多指令在一个数据通路中执行。为了帮助理解流水线，有两种基本书表示流水线的图形化方法，即多时钟周期的流水线图（见图 4-34）和单时钟周期的流水线图（见图 4-36～图 4-40）。多时钟周期图显然简单但不包括所有的细节。下面以这 5 条指令构成的指令序列为例进行说明：

```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

图 4-43 表示的是该指令序列的多时钟周期流水线图。与图 4-25 中洗衣店流水线的表示方法类似，时间从左到右前进，指令从上到下前进。沿着指令轴分别表示各流水级以及所占据的时钟周期。这些程式化的数据通路用图形的方式展示了流水线的 5 个级别，但用方框来命名每个流水线等级也是很好的表示方法。图 4-44 给出了一个更加传统的多时钟周期流水线图的表示方法。需要注意的是，图 4-43 中描述的是每个步骤中使用的物理资源，而图 4-44 描述的是每个步骤的名称。

单时钟周期流水线图表示的是在一个时钟周期内整个数据通路的状态，通常所有 5 个流水级中的指令都在各流水级上做相应的标志。这种流水线图描述了在每一个时钟周期内流水线中所发生事件的细节。通常，可使用一组单时钟周期流水线图来表示在一系列时钟周期内的流水 297
线操作，而使用多时钟周期流水线图对流水线总体进行全局描述。（如果你对图 4-43 的细节感兴趣，可参考 4.13 节中对单时钟周期图的描述）。从多时钟周期图中抽出一个时钟周期就表示了单时钟周期图流水线的状态，其中显示了流水线中每条指令对数据通路的使用。例如，图 4-45的单时钟周期图对应的就是图 4-43 和图 4-44 的第 5 个时钟周期。很明显，单时钟周期图可以表现更多的细节，但表示同样多时钟周期时所占空间要比多时钟周期图大得多。本章后面的练习会要求你根据其他的指令序列画出对应的流水线图。

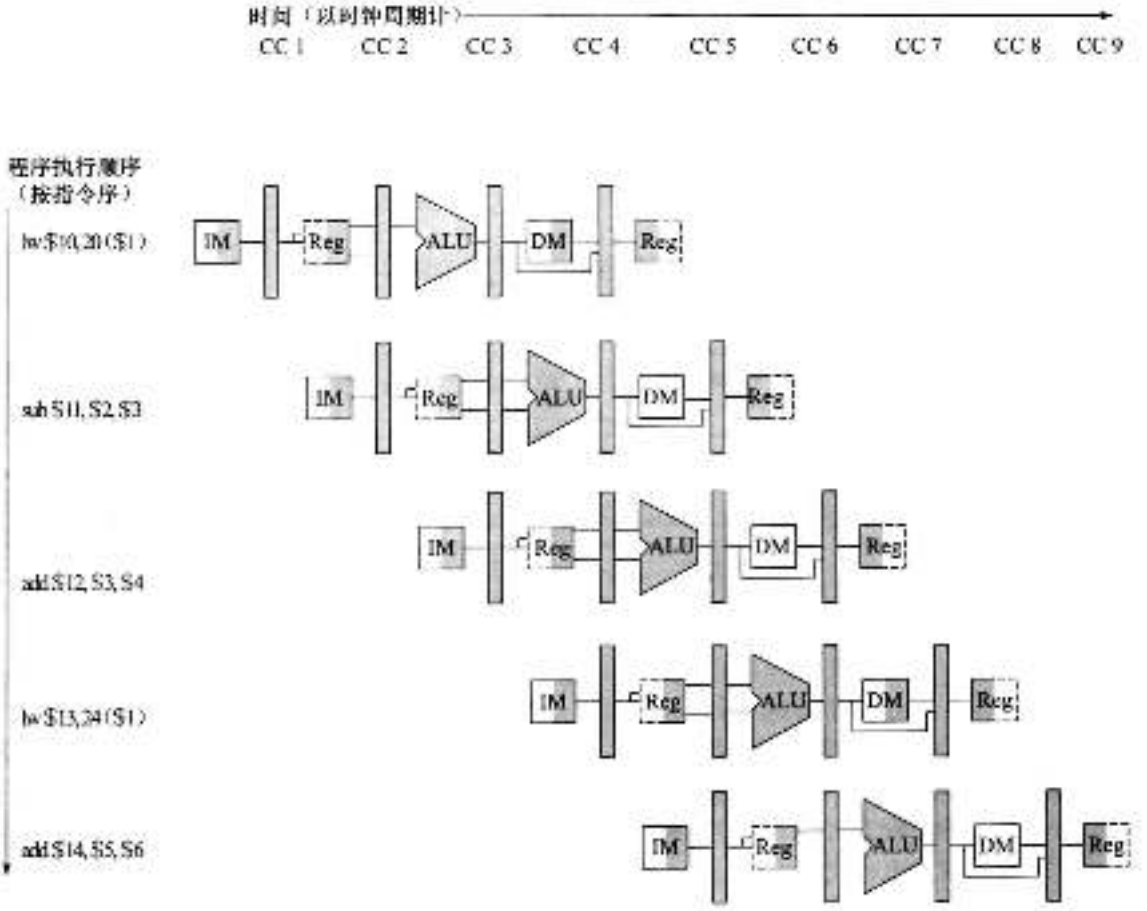



图 4-43 5 条指令的多时钟周期流水线图。此种流水线图在一幅图中表示了指令序列的完整执行过程。指令从上到下按照执行的顺序被排列，时钟周期从左向右前进。与图 4-28 流水线表示方法不同的是，本图给出了每一级的流水线寄存器。图 4-44 给出了这种图更为传统的表示方法

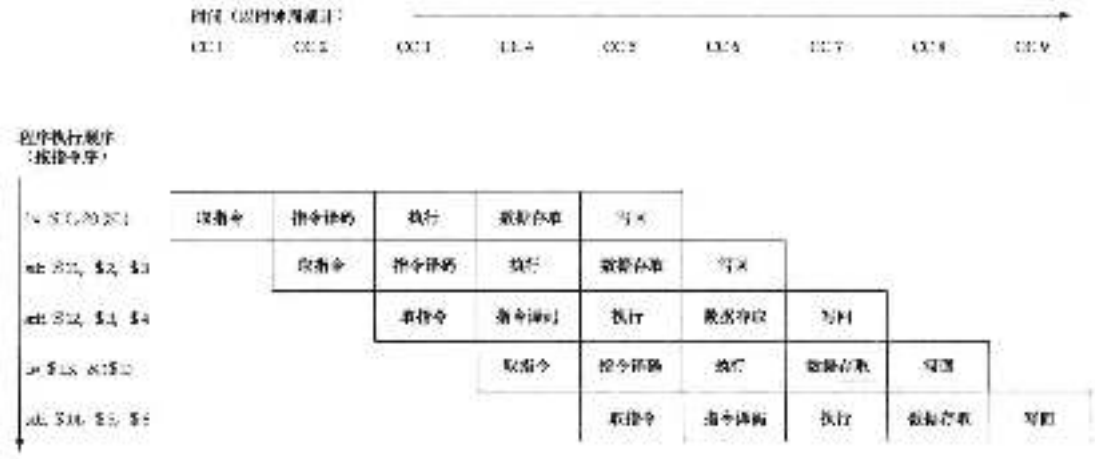


图 4-44 相对图 4-43 更为传统的多时钟周期流水线图

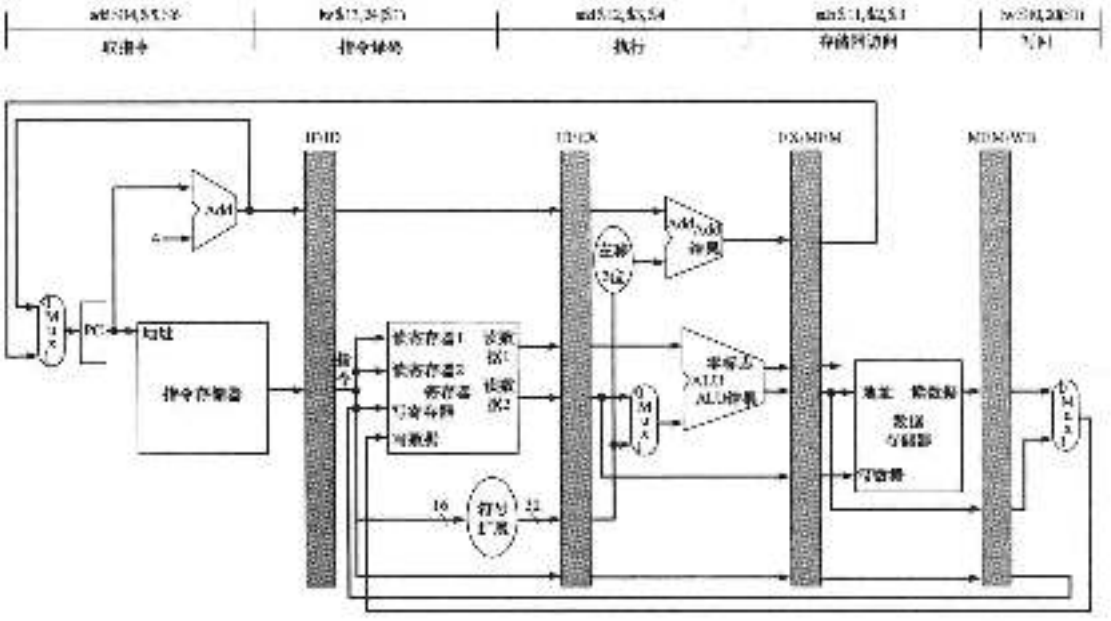


图 4-45 对应图 4-43 和图 4-44 的流水线第 5 个时钟周期的单时钟周期流水线。从图中可以看出，单时钟周期图就是从多时钟周期图中抽出的一列

## 小测验

几个学生在讨论五级流水线的效率问题。有一个学生指出并非所有流水级中的指令都是活动的。在忽略冒险的情况下，他们作出了以下几个断言，其中哪一个是正确的？

298 ~ 299

1）允许跳转、分支、ALU 指令使用比 5 级（装载指令需要的级数）更少的级数将在所有情况下增加流水线的性能。

2）允许一些指令使用更少的级数并不能提高性能，因为吞吐率是由时钟周期决定的。每条指令所需的流水线级数仅影响它的延迟时间，而不影响吞吐率。

3）不可能减少 ALU 指令所需的时钟周期数，因为它们需要写回结果。不过分支和跳转指令是可以减少时钟周期数的，因此存在改善性能的机会。

4）相对于尝试减少指令所需的时钟周期数，我们可以延长流水线的级数，虽然每条指令

花费更多的时钟周期数，但时钟周期的长度变短了，这样才能提高性能。

## 4.6.2 流水线控制

相对小型的任何计算机，6600 型计算机的控制系统是大不相同的。

——James Thornton，《Design of a Computer: The Control Data 6600》，1970

4.3 节介绍了在单周期数据通路加入控制的方法，下面我们将介绍在采用流水线的数据通路中如何加入控制。首先我们在带有诸多限制条件下通过一个简单设计方案了解流水线控制。

我们首先要做的工作就是标识已有数据通路上的控制信号，如图 4-46 所示。我们尽量借用图 4-17 中简单数据通路的控制方法，特别是使用相同的 ALU 控制逻辑、分支逻辑、目的寄存器号多选器和控制信号。尽管图 4-12、图 4-16 以及图 4-18 中已给出了这些功能单元的定义，为了使下面的内容更易于理解，图 4-47～图 4-49 重新对其进行了解释。

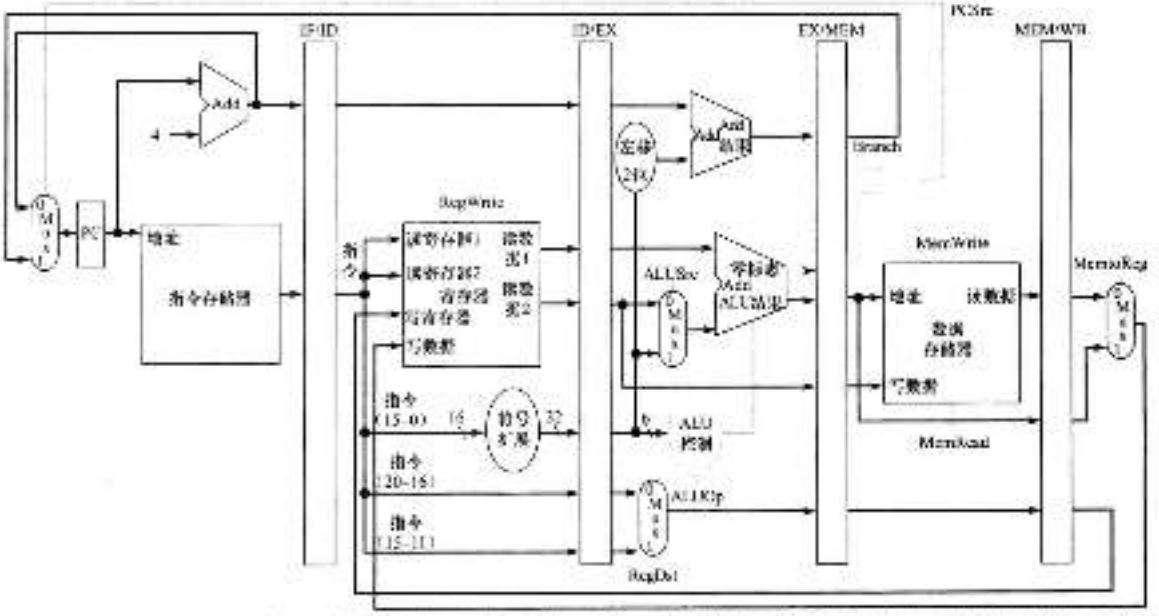



图 4-46 在图 4-41 上增加了控制信号的流水线数据通路。这个数据通路采用了与 4.4 节中相同的 PC 源控制逻辑、寄存器目标号和 ALU 控制。需要注意的是，这时在 EX 流水级中指令需要一个 6 位的功能字段（功能码）作为 ALU 控制的输入，所以该 6 位字段必须存放在 ID/EX 流水线寄存器中。而该 6 位字段是指令中立即数的低 6 位，由于在对立即数进行符号扩展时低 6 位没有发生变化，所以 ID/EX 流水线寄存器可以从立即数中获得这 6 位数

与单时钟周期实现方法一样，我们假定在每个时钟周期内都会写 PC，因此就不需要单独的 PC 写信号。同理，流水线寄存器（IF/ID、ID/EX、EX/MEM 和 MEM/WB）也不需要单独的写信号，因为在每个周期它们也会写入一次。

为了详细说明流水线的控制问题，我们只需要在每一个流水级中都设置相应的控制信号。由于每一个控制信号只与某个流水级中的某个功能单元相关，因此我们可以根据流水线的 5 级将控制信号分成 5 组：

1）取指令：读指令存储器和写 PC 的控制信号总是有效的，因此在取指阶段没有特别需要控制的内容。

2）指令译码/寄存器堆读：与第一步类似，在每个时钟周期内本阶段所做的工作都是完全相同的，因此不需要设置控制信号。

3）指令执行/地址计算：控制信号有 RegDst、ALUOp 和 ALUSrc（见图 4-47 和图 4-48）。

300
↓
301

根据这些信号选择结果寄存器、ALU 操作，并为 ALU 读取数据 2 或符号扩展后的立即数。

4）存储器访问：这一步的控制信号有 Branch、MemRead 和 MemWrite。这些控制信号分别由相等则分支、装载指令和存储指令设置。除非控制电路确定是一条分支指令并且 ALU 结果为 0，否则将选择线性地址中的下一条指令作为图 4-48 中的 PCSrc 信号。

5）写回：控制信号有 MemtoReg 和 RegWrite，其中前者决定是将 ALU 结果还是将存储器数据传送到寄存器堆，后者决定是否写入寄存器堆。

由于采用流水线方式的数据通路并不改变控制信号的意义，因此可以使用与简单数据通路相同
302 的控制信号。图 4-49 就与 4.4 节具有相同的控制信号，只是这 9 个控制信号按流水级进行了分组。

| 指令操作码 | ALUOp | 指令类型 | 功能码 | ALU 操作 | ALU 控制信号 |
|---|---|---|---|---|---|
| LW | 00 | 取字 | XXXXXX | 加 | 0010 |
| SW | 00 | 存字 | XXXXXX | 加 | 0010 |
| 相等则分支 | 01 | 相等则分支 | XXXXXX | 减 | 0110 |
| R 型 | 10 | 加 | 100000 | 加 | 0010 |
| R 型 | 10 | 减 | 100010 | 减 | 0110 |
| R 型 | 10 | 与 | 100100 | 与 | 0000 |
| R 型 | 10 | 或 | 100101 | 或 | 0001 |
| R 型 | 10 | 小于则置位 | 101010 | 小于则置位 | 0111 |

图 4-47 图 4-12 的副本。本图描述了如何根据 ALUOp 控制位和不同 R 型指令的功能码设置 ALU 控制信号的值

| 信号名 | 该无效时的效果（0） | 该有效时的效果（1） |
|---|---|---|
| RegDst | 写入寄存器的目标号来自 rt 字段（20:16 位） | 写入寄存器的目标号来自 rd 字段（15:11 位） |
| RegWrite | 无 | 写入寄存器的源寄存器设置为输入的写入数据 |
| ALUSrc | 第二个 ALU 操作数来自第二个寄存器堆的输出（读数据 2） | 第二个 ALU 操作数是指令低 16 位的符号扩展 |
| PCSrc | PC 被 PC+4 替代 | PC 被分支目标地址替代 |
| MemRead | 无 | 输入地址对应的数据存储器的内容为读数据的输出 |
| MemWrite | 无 | 输入地址对应的数据存储器的内容替换为写数据的输入 |
| MemtoReg | ALU 提供寄存器写数据的输入 | 数据存储器提供寄存器写数据的输入 |

图 4-48 图 4-16 的副本。图中定义了 7 个控制信号的功能。ALUOp 已经在图 4-47 的第二列中定义。当一个二路多选器的控制位有效时，多选器选择 1 对应输入；否则，如果控制位无效，多选器选择 0 对应输入。注意 PCSrc 是由图 4-46 的一个与门控制的。如果分支信号与 ALU 的零信号都有效，则 PCSrc 为 1，否则为 0。控制单元仅在 beq 指令中才设置分支信号有效，其他时候 PCSrc 都会为 0

| 指令 | 执行地址计算阶段的控制信号 | | | | 存储器访问阶段的控制信号 | | | 写回阶段的控制信号 | |
|---|---|---|---|---|---|---|---|---|---|
| | RegDst | ALUOp1 | ALUOp0 | ALUSrc | Branch | MemRead | MemWrite | RegWrite | MemtoReg |
| R 型 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

图 4-49 按流水线最后三级分为三组的控制信号，其值与图 4-18 相同

实现控制就是为每一条指令的每一个步骤中的 9 个控制信号设置合适的值。最简单的实现方法就是扩展流水线寄存器使之包含这些控制信号。

由于控制从 EX 级开始，因此可以在指令译码阶段创建控制信号。图 4-50 描述了当指令在

流水线中传递时控制信号的使用方法，这一点与图 4-41 中执行装载指令时目的寄存器号在流水线中的传递过程类似。图 4-51 描述了带有扩展流水线寄存器且将控制信号连接到相应流水级的完整数据通路。（如果你想知道更多的细节，4.13 节给出了更多 MIPS 代码在流水线硬件中执行的单时钟周期流水线图。）

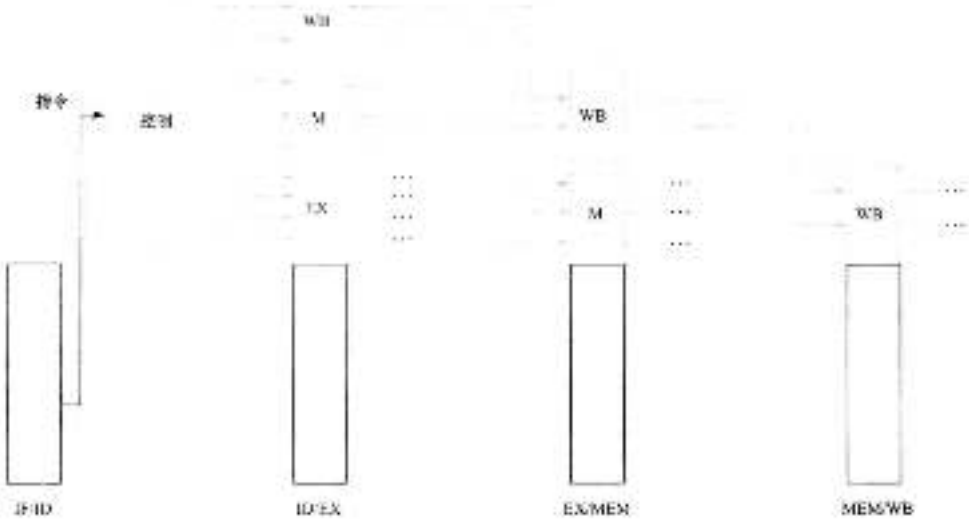



图 4-50 流水线最后三级的控制信号。需要注意的是，9 个控制信号中有 4 个用于 EX 级，而剩下的 5 个控制信号被传递到扩展的保存控制信号的 EX/MEM 流水线寄存器中；传递来的 5 个控制信号中有 3 个用于 MEM 级，剩下的 2 个传递到 MEM/WB 并用于 WB 级

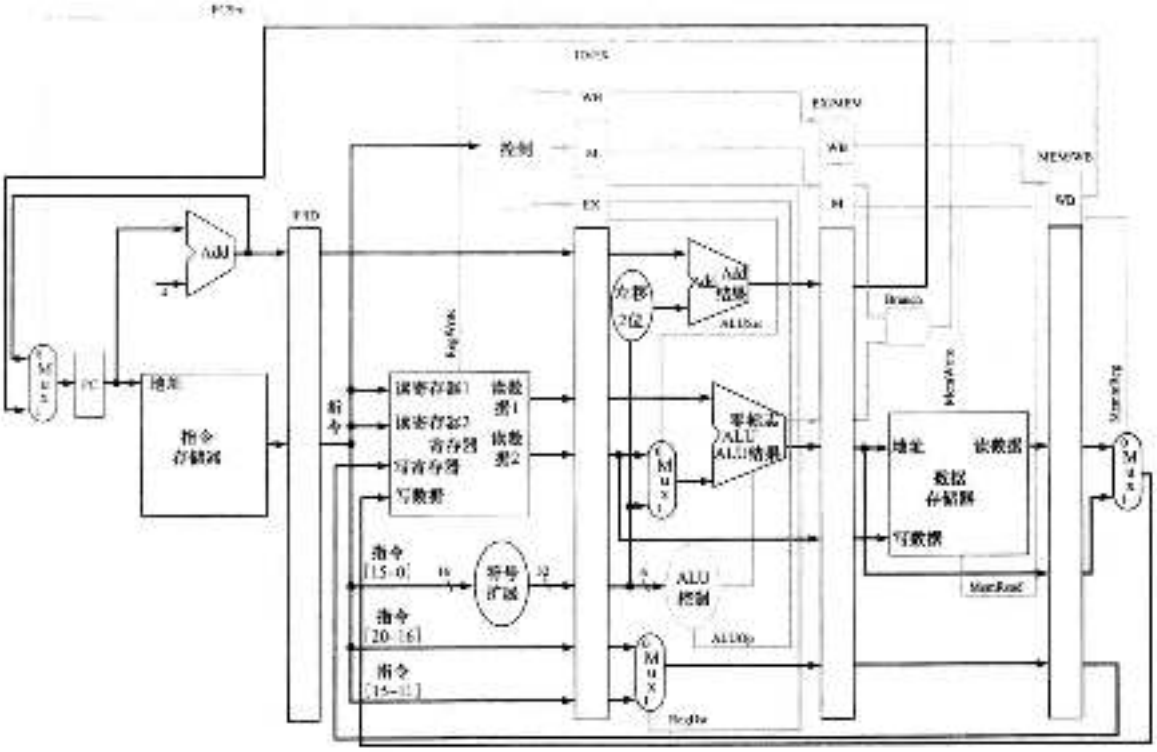



图 4-51 图 4-46 中的流水线数据通路，已将控制信号连接到流水线寄存器的控制部分。流水线最后三级的控制信号是在指令译码阶段创建的，随后放入 ID/EX 流水线寄存器。每个流水级使用相应的控制信号，并将剩余的控制信号传递到下个流水级

## 4.7 数据冒险：旁路与阻塞

*这是什么意思，为什么要构建它？这是旁路，你必须构建旁路。*

——*Douglas Adams,《The Hitchhiker's Guide to the Galaxy》, 1979*

上节的例子介绍了流水线的强大功能以及硬件如何以流水线的方式执行任务。本节我们避开这些光环，看看流水线在实际程序中的情况。图4-43～图4-45中的各指令之间是相互独立的，其中任何一条指令都没有用到任何其他指令的计算结果。然而，在4.5节中我们就已经发现数据冒险是影响流水线执行的主要原因之一。

让我们分析下面这个带有许多相关性的指令序列（依赖关系以粗体标出）：

```
sub   $2, $1,$3      # Register $2 written by sub
and   $12,$2,$5      # 1st operand($2) depends on sub
or    $13,$6,$2      # 2nd operand($2) depends on sub
add   $14,$2,$2      # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)    # Base ($2) depends on sub
```

303
↓
304

后4条指令都依赖于第一条指令得到的寄存器$2的结果。如果寄存器$2在sub指令执行之前的值为10，而在sub指令执行之后的值为-20，程序员认为后4条指令访问到的寄存器$2的值为-20。

这个指令序列在流水线中是如何执行的呢？图4-52用多时钟周期流水线图进行了表示。为了在当前流水线中表示这个指令序列的执行过程，图4-52的顶部给出了寄存器$2中的值，可以看出寄存器$2的值在第5个时钟周期的中间发生改变，也就是sub指令写结果的时候。

最后一个潜在的冒险可以通过设计相应的寄存器堆硬件解决。当一个寄存器在同一时钟周期内同时读和写时会发生什么呢？这里我们假设写寄存器操作发生在时钟周期的前半段而读寄存器操作发生在时钟周期的后半段，因此读操作将读取到最新写入的内容。大多数寄存器堆的实现方法与我们的假设是一致的，而且在这种假设条件下不会发生数据冒险。

图4-52表明如果在第5个时钟周期之前读寄存器$2，读操作得到的寄存器值就不会是sub指令的结果。因此，指令add和sw可得到正确结果-20，而指令AND和OR将得到错误结果10。使用这种风格的流水线图，当一条依赖关系的方向与时间轴相反时，该问题就变得很明显。

正如4.5节所提到的那样，sub指令在EX级（第3个时钟周期）的末尾就可以得到需要的结果。那么AND指令和OR指令什么时候真正需要该数据呢？应该是在AND指令和OR指令的EX级开始前，分别是第4个和第5个时钟周期。所以只要我们在刚得到数据时就将其旁路给所需的单元而不是等待其可以从寄存器堆中读出来，就可以无阻塞地执行这两条指令了。

旁路到底是怎样工作的呢？在本节下面的部分，为了简化讨论，我们仅考虑如何直接传送EX段产生的数据，该数据可能是ALU运算的结果，也可能是地址计算的结果。这意味着如果一条指令试图在EX级使用前面一条指令在WB级才写入寄存器堆的数据时，我们需要提前将数据送到ALU的输入端。

一种更精确的表示相关性的方法是使用流水线寄存器字段。例如，ID/EX.RegisterRs表示一个需要流水线寄存器ID/EX获得的源寄存器号，其值可在流水线寄存器中ID/EX中找到。这个名称的第一部分，即点号的左边，表示流水线寄存器的名称；第二部分表示寄存器中字段的名称。使用这种表示方法，4个冒险条件分别是：

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

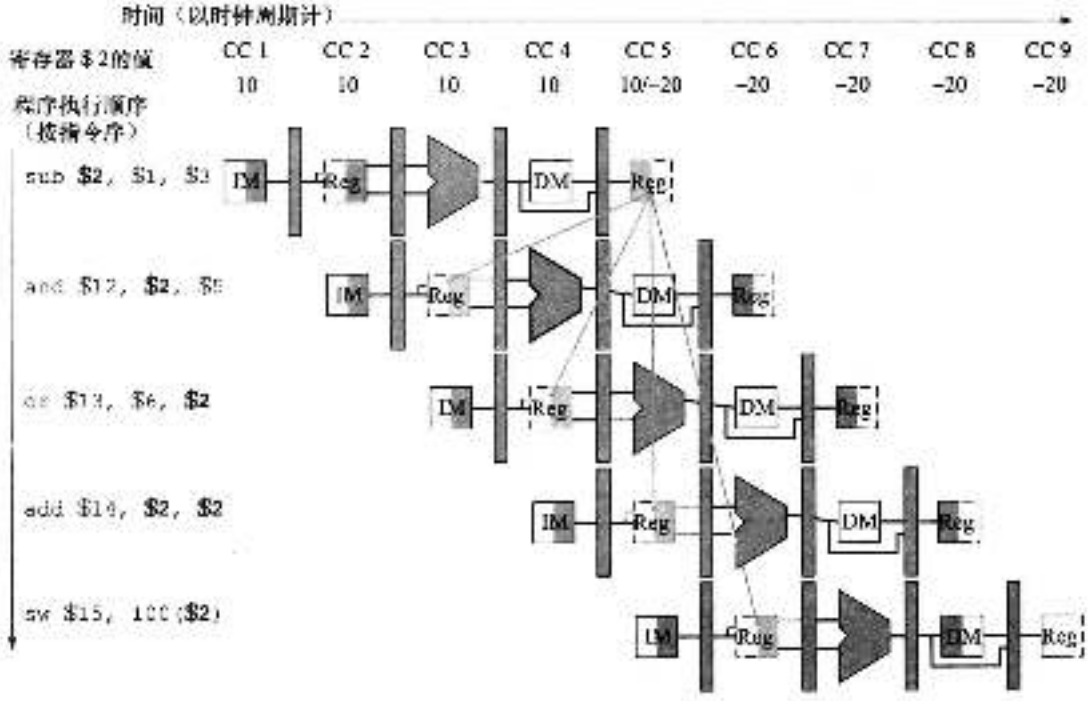



图 4-52 使用简化数据通路表示一个由 5 条指令组成的指令序列中的流水线相关情况来说明相关问题。所有的相关都用灰色标记出来，顶部的“CC 1”表示第 1 个时钟周期。指令序列中第一条指令写寄存器 $2，后 4 条指令读寄存器 $2。寄存器 $2 在第 5 个时钟周期被写入，所以在此之前它的值都是无效的。（当这样的写操作发生时，一个时钟周期中寄存器的读操作返回该周期前半段写入的值。）数据相关性用数据通路中从顶部到底部的灰线表示。那些导致时间后退的依赖就是流水线数据冒险 305

本节开始给出的指令序列的第一个冒险发生在 sub $2, $1, $3 的结果和 and $12, $2, $5 的第一个读操作数之间。这个冒险在 and 指令处于 EX 级而 sub 指令处于 MEM 级时就能检测出来，这就是冒险 1a：

EX/MEM.RegisterRd = ID/EX.RegisterRs = $2。

**例题：相关性检测**

将前面指令序列中的相关性进行分类：

```
sub $2,   $1, $3  # Register $2 set by sub
and $12,  $2, $5  # 1st operand($2) set by sub
or  $13,  $6, $2  # 2nd operand($2) set by sub
add $14,  $2, $2  # 1st($2) & 2rd($2) set by sub
sw  $15,  100($2) # Index($2) set by sub
```

306

**答案**

如上所述，sub-and 是一个 1a 类冒险。其余的冒险分别是：

- sub-or 是一个 2b 类冒险：

  MEM/WB.RegisterRd = ID/EX.RegisterRt = $2

- sub-add 上的两个相关性都不是冒险，因为在 add 的 ID 级寄存器堆已能提供相应的数据。
- sub 指令和 sw 指令之间也不存在数据冒险，因为 sw 指令在 sub 指令写寄存器 $2 后才读取 $2。 □

但是，直接采用总是旁路的方式解决冒险是不正确的，因为某些指令可能不写回寄存器，

就会产生一些不必要的旁路。一种简单的解决方法是检测 RegWrite 信号是否是活动的，即通过检测流水线寄存器在 EX 和 MEM 级的 WB 控制字段以确定 RegWrite 是否被有效。而且，MIPS 要求 $0 始终为 0。这就需要在目标寄存器是 $0 的情况下（如 sll $0，$1，2），必须避免把 $0 按非零结果旁路，从而使得汇编程序员和编译器不必考虑 $0 作为目标寄存器的情况。因此，需要在第一类冒险条件中加入附加条件 EX/MEM.RegisterRd≠0，在第二类冒险条件中加入附加条件 MEM/WB.RegisterRd≠0。

至此，我们介绍了检测冒险的方法，问题已经解决了一半，但仍然需要解决旁路数据策略的问题。

图 4-53 描述了图 4-52 的指令序列中流水线寄存器和 ALU 输入间的相关性。与图 4-52 不同的是，这里的相关性开始于一个流水线寄存器而不是等待 WB 级写操作到寄存器堆。由于流水线寄存器保存了需要旁路的数据，因此后面的指令能够获得相应的数据。

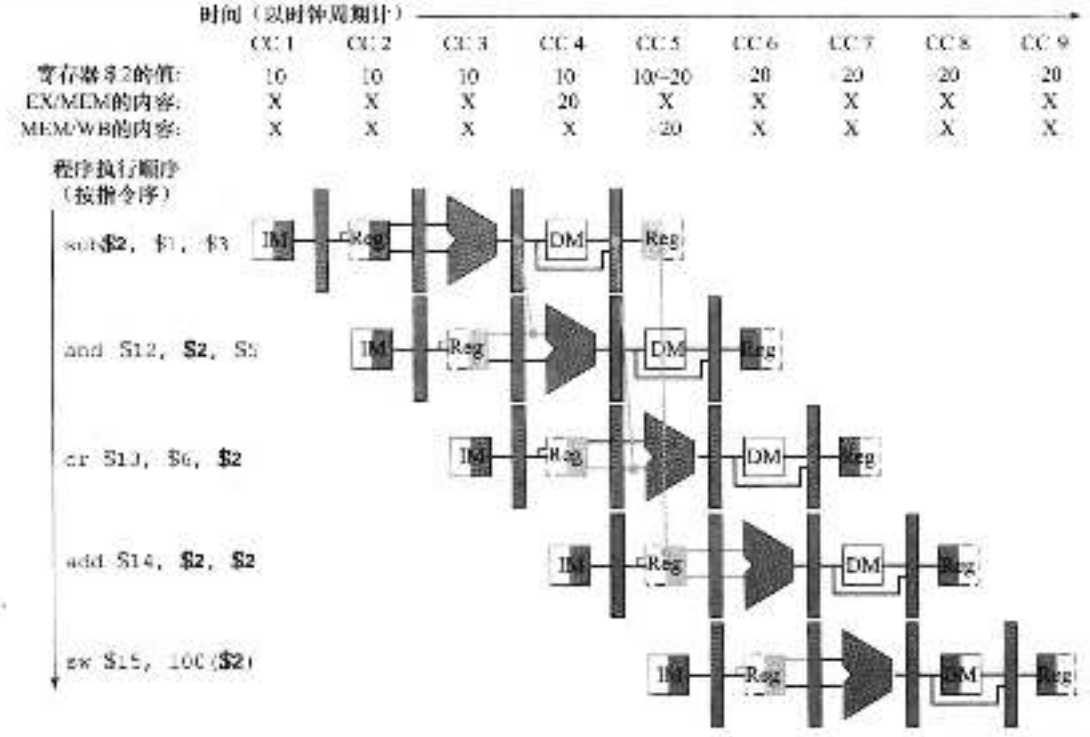
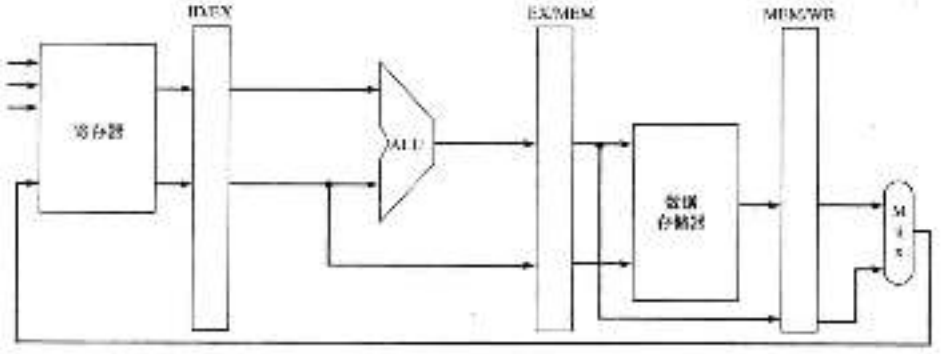



图 4-53 流水线寄存器间的相关随着时间移动，所以通过旁路流水线寄存器中保存的结果就有可能提供 AND 指令和 OR 指令所需的 ALU 输入。流水线寄存器存有相应的值，在数据写入寄存器堆之前就已经有效了。如果寄存器堆可在同一时钟周期内旁路要读写的数据，add 指令就不用重复了。这种寄存器堆的旁路的值不是来自于流水线寄存器而是来自寄存器堆。它使得寄存器 $2 中的值在第 5 个时钟周期的开始是 10，而在周期结束时是 -20，即在这一时钟周期里读操作读到的值是写操作写入的值。在本节下面的部分，我们将处理所有的旁路（除了存储指令要存的数值之外）

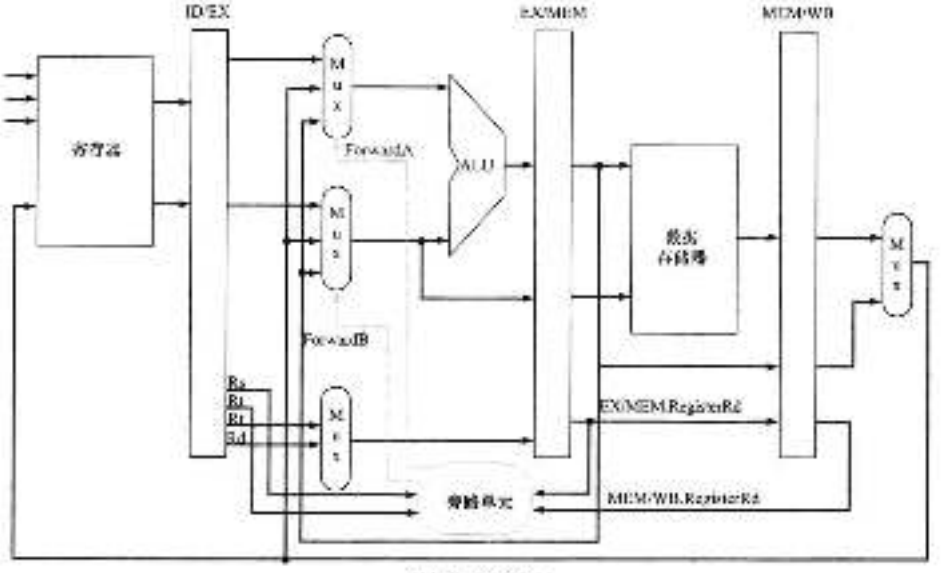
如果可以从任何流水线寄存器而不仅仅从 ID/EX 中得到 ALU 的输入，那么就可以旁路所需的数据。通过在 ALU 的输入中加入多选器和正确的控制策略，就可以在存在相关性的情况下仍然能够全速运行流水线。

现在，假设需要旁路的指令只有 4 个 R 型指令：add、sub、AND 和 OR。图 4-54 给出了在加入旁路机制前后，ALU 和流水线寄存器的示意图。图 4-55 给出了在寄存器堆值和某一旁路的数值间进行选择的 ALU 多选器控制信号的值。

因为 ALU 旁路多选器在 EX 中，所以旁路控制也在这一级中完成。因此，我们必须通过 ID/EX 流水线寄存器从 ID 级中获得操作数寄存器号，以决定是否旁路相应的值。我们已经有了 rt 字段（20～16 位）。在支持旁路前，ID/EX 流水线寄存器未保存 rs 字段。因此，为支持旁路，rs(25～21 位）被加入 ID/EX 流水线寄存器中。




a）未使用旁路




b）使用旁路

图 4-54 加入旁路机制前后的 ALU 和流水线寄存器。下图使用多选器增加了旁路路径，并标识了旁路单元。新硬件用灰色表示。本图只是一个示意图，没有标识诸如符号扩展硬件之类的细节。需要注意的是，尽管 ID/EX. RegisterRt 字段在图中标识了两次，一根连接到多选器，一根连接到旁路单元，但实际上它是一个信号。如前所述，这里还忽略了旁路存储指令中数据的情况。还有一点要注意的是，这一机制也适用于 slt 指令

307 ~ 309

| 多选器控制 | 源 | 解释 |
|---|---|---|
| ForwardA = 00 | ID/EX | 第一个 ALU 操作数来自寄存器堆 |
| ForwardA = 10 | EX/MEM | 第一个 ALU 操作数由上一个 ALU 运算结果旁路获得 |
| ForwardA = 01 | MEM/WB | 第一个 ALU 操作数从数据存储器或者前面的 ALU 结果中旁路获得 |
| ForwardB = 00 | ID/EX | 第二个 ALU 操作数来自寄存器堆 |
| ForwardB = 10 | EX/MEM | 第二个 ALU 操作数由上一个 ALU 运算结果旁路获得 |
| ForwardB = 01 | MEM/WB | 第二个 ALU 操作数由数据存储器或者前面的 ALU 结果旁路获得 |

图 4-55 图 4-54 中旁路多选器的控制信号。作为 ALU 另一个输入的带符号立即数将在本节的“精解”部分中解释

下面将给出检测冒险的条件以及解决冒险的控制信号：

1）EX 冒险：

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

注意，EX/MEM. RegisterRd 字段是 ALU 指令（来自 Rd 字段）或装载指令（来自 Rt 字段）的目标寄存器号。

这种情况是将前一条指令的结果旁路到任何一个 ALU 输入中。如果前一条指令要写寄存器堆且要写的寄存器号与 ALU 输入要读的寄存器号（A 或 B）一致（只要不是寄存器 0），那么就调整多选器从流水线寄存器 EX/MEM 中读取数值。

2）MEM 冒险：

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

如上所述，在 WB 级不会发生冒险，这是因为我们假设在 ID 级指令读取的寄存器与 WB 级指令写入的寄存器是同一寄存器时，就由寄存器堆提供正确的结果。这样，寄存器堆实现了另一种形式的旁路，但这种旁路只发生在寄存器堆内部。

更为复杂的带在数据冒险发生在 WB 级的指令结果、MEM 级的指令结果和 ALU 级的指令源操作数之间。例如，在一个寄存器中对多个数字进行求和运算时，一系列连续的指令将会读写到同一寄存器：

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
. . .
```

310

在这种情况下，由于 MEM 级的结果是最新的，因而结果是由 MEM 级旁路得到。这样，对 MEM 冒险的控制策略为（额外加入的条件采用粗体表示）：

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

图 4-56 给出了为了支持旁路 EX 级结果所增加的必要硬件设备。注意，图中 EX/MEM. RegisterRd 字段是一条 ALU 指令（来自 Rd 字段）或装载指令（来自 Rt 字段）的目标寄存器。

4.13 节给出了两段 MIPS 代码，其中存在需要使用旁路解决的冒险，你可以使用单时钟周期流水线图对这些例子进行深入分析。

**精解** 旁路还可以帮助解决因存储指令依赖其他指令而导致的冒险。由于存储指令在

MEM 级只使用一个数据，所以旁路应当容易。在 MIPS 架构中，由于存储器之间的复制很频繁，必须考虑复制时存储指令后紧跟着的是装载指令的情况。为了提高复制的速度，我们需要加入更多的旁路硬件。如果我们查阅图 4-53，并分别使用 lw 和 sw 指令代替 sub 和 AND 指令，我们将发现这时也可能避免一次阻塞，只要装载指令的 MEM/WB 寄存器中存在的数据能够及时地提供给存储指令在 MEM 级使用。为了实现这个功能，我们需要在存储器访问阶段加入旁路。我们将如何对其修改作为练习留给读者。

311
~
312

此外，图 4-56 中省略了装载指令和存储指令所需的输入到 ALU 的带符号立即数。由于中央控制决定如何在寄存器和立即数之间进行选择，而且旁路单元选择流水线寄存器作为 ALU 的一个寄存器输入，因此最简单的解决方法就是加入一个 2:1 的多选器，由它在 ForwardB 多选器的输出和带符号立即数之间进行选择。图 4-57 描述了这种变化。

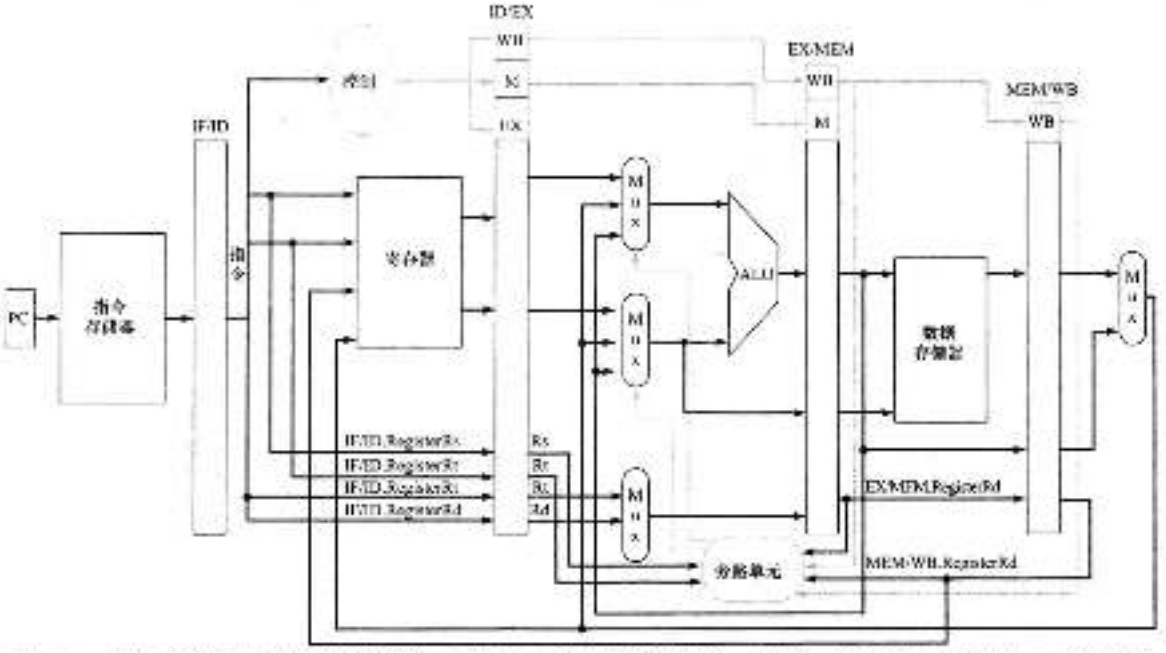



图 4-56 通过旁路解决冒险的数据通路。与图 4-51 的数据通路相比，本图在 ALU 的输入部分加入了多选器。为了使表述更加清楚，图中忽略了完整数据通路中的一些细节，如分支硬件和符号扩展硬件等

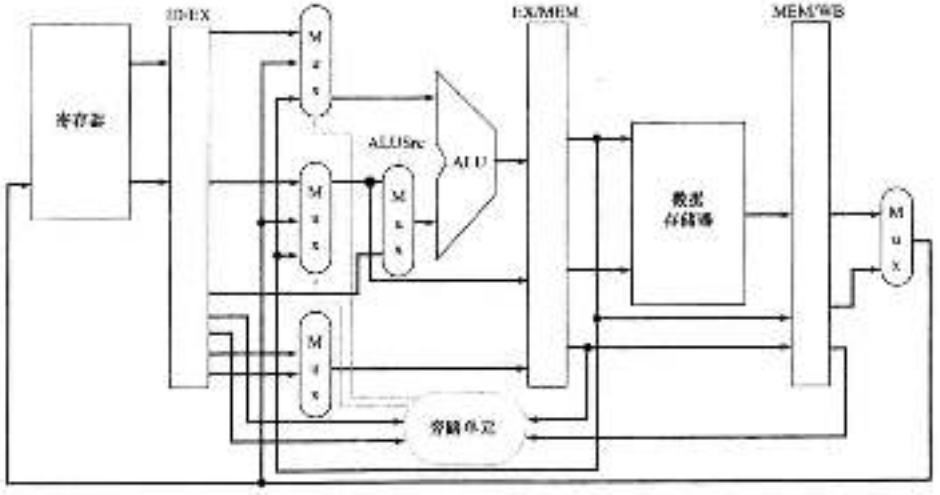



图 4-57 在图 4-54 中的数据通路加入了一个 2:1 的多选器，用以选择带符号立即数作为 ALU 的输入

## 数据冒险与阻塞

*如果你第一次没有成功，那就重新定义成功是什么。*

——佚名

如4.5节所述，当一条指令试图读取一个由前一条装载指令读入的寄存器时，就无法使用旁路解决冒险了。图4-58说明了这个问题。当ALU正在执行后续指令的操作时，数据仍然是在第4个时钟周期从内存中读出的。所以，当装载指令后紧跟着一个需要读取它的结果的指令时，必须采用相应的机制阻塞流水线。

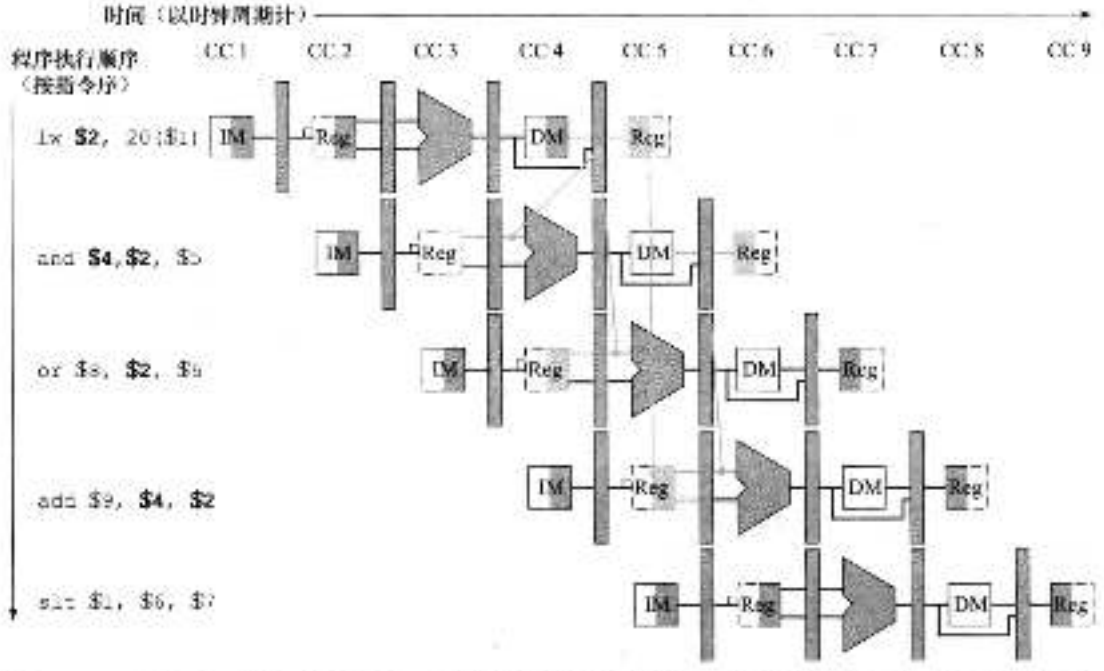



图4-58 一个指令序列的流水线图。由于装载指令和紧随其后的and指令之间的相关性在时间上是回溯的。这种冒险不可能通过旁路来解决。因此，这类指令组合导致冒险检测单元产生阻塞

313 因此，除了一个旁路单元以外，还需要一个冒险检测单元。它工作在ID级，从而可以在装载指令与紧随其后需要它的结果的指令间插入阻塞。这个冒险检测单元检测装载指令，它的控制满足如下条件：

```
if (ID/EX.MemRead and
  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
   (ID/EX.RegisterRt = IF/ID.RegisterRt)))
   stall the pipeline
```

因为读取数据存储器的指令一定是装载指令，所以第一行条件检查指令是否是一条装载指令。后面的两行是检测在EX级的装载指令的目的寄存器是否与在ID级的指令的某一个源寄存器相匹配。如果条件成立，指令将阻塞一个时钟周期。经过这一个周期的阻塞，旁路逻辑就可以处理相关性并继续执行程序了（如果没有采用旁路，那么图4-58中的指令还需要阻塞一个周期）。

如果处于ID级的指令被阻塞，那么处于IF级的指令也必须被阻塞，否则，已经取到的指令就会丢失。防止这两条指令继续执行的方法是保持PC寄存器和IF/ID流水线寄存器不变。如果这些寄存器内容保持不变，在IF级的指令将继续使用相同的PC取指令，而在ID级将继续使用IF/ID流水线寄存器中的相同的指令字段读寄存器堆。再回到我们熟悉的洗衣店的例子

中，这一过程就好像是你重新打开洗衣机统相同的衣服而让烘干机继续空转一样。当然，就像烘干机一样，从 EX 开始的流水线后半部分必须“空转”，它们执行的指令必须不产生任何效果，即**空指令**（nop）。

⊃ 空指令：一种不进行任何操作或不改变任何状态的指令。

那我们怎么在流水线中插入空指令（就像气泡一样）呢？从图 4-49 中我们知道，在 EX、MEM 和 WB 级如果将所有 9 个控制信号都清除（置为 0），就会产生一个“什么都不做”的指令，即空指令。通过识别 ID 级的冒险，可以在流水线中插入一个气泡，方法是把 ID/EX 流水线寄存器的 EX、MEM 和 WB 级的控制信号都置为 0。这些控制信号在每个时钟周期都向前传递，但不会产生不良后果，因为如果控制信号都是 0，那么所有寄存器和存储器都不进行写操作。

图 4-59 描述了该指令序列的运行过程：与 AND 指令相关的流水线执行槽被插入一条空指令，这样从 AND 开始的所有指令都被延迟一个时钟周期。就像水管中的气泡，一个阻塞的气泡会延缓后面所有指令的执行，同时在每个时钟周期，气泡也沿着流水线向后推进一级，直到它退出流水线为止。在这个例子中，冒险强迫指令 AND 和 OR 在第 4 个时钟周期重复第 3 个时钟周期所做的内容，即指令 AND 读存储器并进行译码，指令 OR 从指令存储器中取指令。这种重复的工作就像阻塞一样，但它的效果是拉伸了指令 AND 和 OR，并且延迟了第二个 add 指令的取数操作。

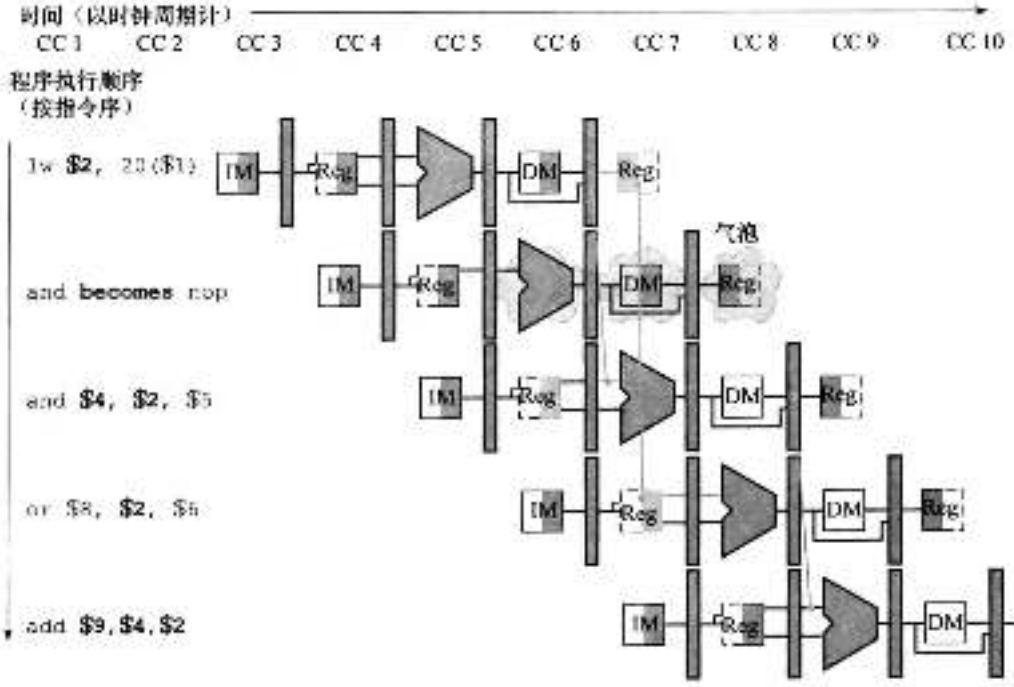



图 4-59 在流水线中插入阻塞的方法。在第 4 个时钟周期中，通过将 and 指令变成 nop 插入了一个气泡。注意，and 指令的 IF 和 ID 级在第 2 个和第 3 个时钟周期，但它的 EX 级被推迟到第 5 个时钟周期（不阻塞的话应该在第 4 个时钟周期）。与此类似，OR 指令的 IF 级在第 3 个时钟周期，但它的 ID 级被推迟到第 5 个时钟周期（不阻塞的话应该在第 4 个时钟周期）。在插入气泡后，所有的相关性时间前进，冒险不再发生

图 4-60 给出了冒险检测单元和旁路单元的流水线连接。和前面的介绍一样，旁路单元控制 ALU 多选器，从而可以用相应的流水线寄存器的值代替通用寄存器的值。冒险检测单元控制 PC 和 IF/ID 流水线寄存器的写入，以及在实际控制信号与全 0 中进行选择的多选器。如果上面的取 314

指令冒险条件为真，冒险检测单元就阻塞并清除所有的控制字段。如果你想了解更多细节，4.13 节给出了一段 MIPS 代码，其中含有会导致阻塞的冒险，并附带了对应的单时钟周期流水线图。

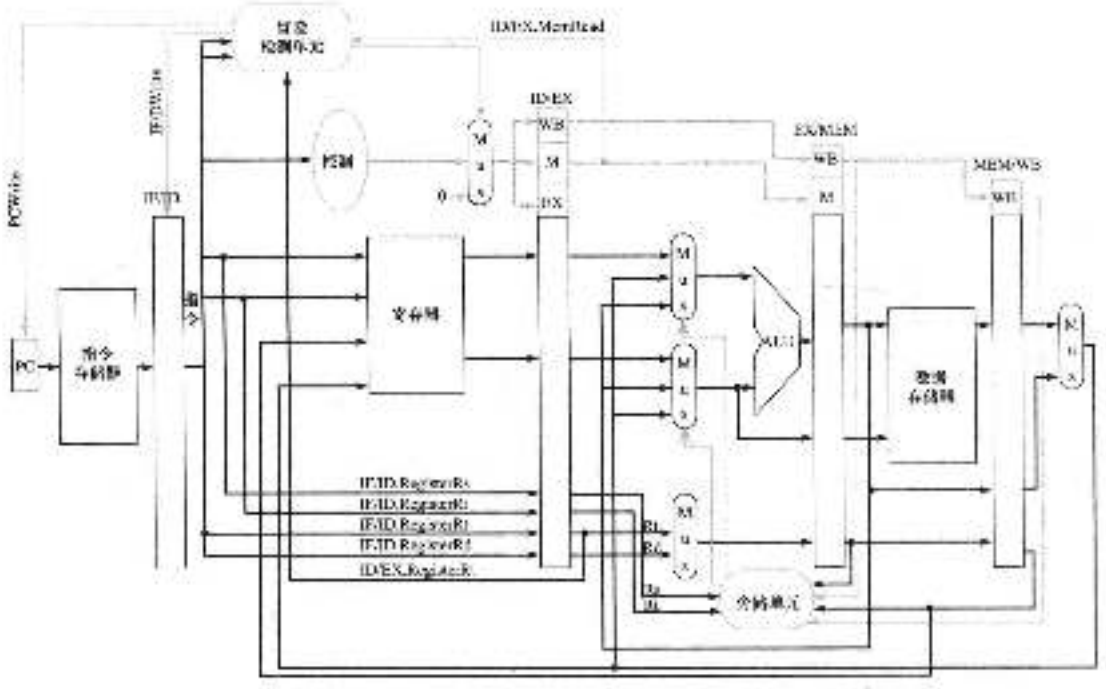



图 4-60 流水线控制概述，其中包括两个旁路多选器，一个冒险检测单元和一个旁路单元虽然简化了 ID 和 EX 级（省略了经过符号扩展的立即数和分支逻辑），但本图说明了旁路的基本硬件支持

**重点** 尽管编译器通常依赖于硬件解决冒险相关性以保证指令正确执行，但为了获得最好的效果，编译器的设计者必须了解流水线。否则，未预料到的阻塞会降低编译代码的执行效率。

**精解** 前面提到为了避免写寄存器或存储器而将所有的控制信号都置为 0。事实上，只需将信号 RegWrite 和 MemWrite 置为 0，而不用关心其他控制信号。

## 4.8 控制冒险

*即使对邪恶从侧面进行上千次攻击，也比不上从根源上进行一次攻击。*

——*Henry David Thoreau*，《*Walden*》，*1854*

315 - 316

直到现在，我们只把冒险的概念局限在算术运算和数据传输中。但正如 4.5 节中所提到的那样，还有一类包含分支的流水线冒险。图 4-61 描述了一个指令序列，同时说明了在流水线中何时会发生分支。为了维持流水线的运行，每个时钟周期都必须取指，但在我们的设计中必须等到 MEM 级才能确定是否执行分支。如 4.5 节所述，与前面讨论的数据冒险相对应，这种为了确保预取正确指令而导致的延迟叫作控制冒险（control hazard）或分支冒险（branch hazard）。

因为控制冒险相对易于理解，它们出现的频率也比数据冒险要小得多，而且与采用旁路就能有效地解决数据冒险相比，还没有有效的方法能够解决控制冒险。因此，这一节关于控制冒险的讨论要比前一节的数据冒险短得多。本节将介绍两种解决控制冒险的方案，并进行了优化。

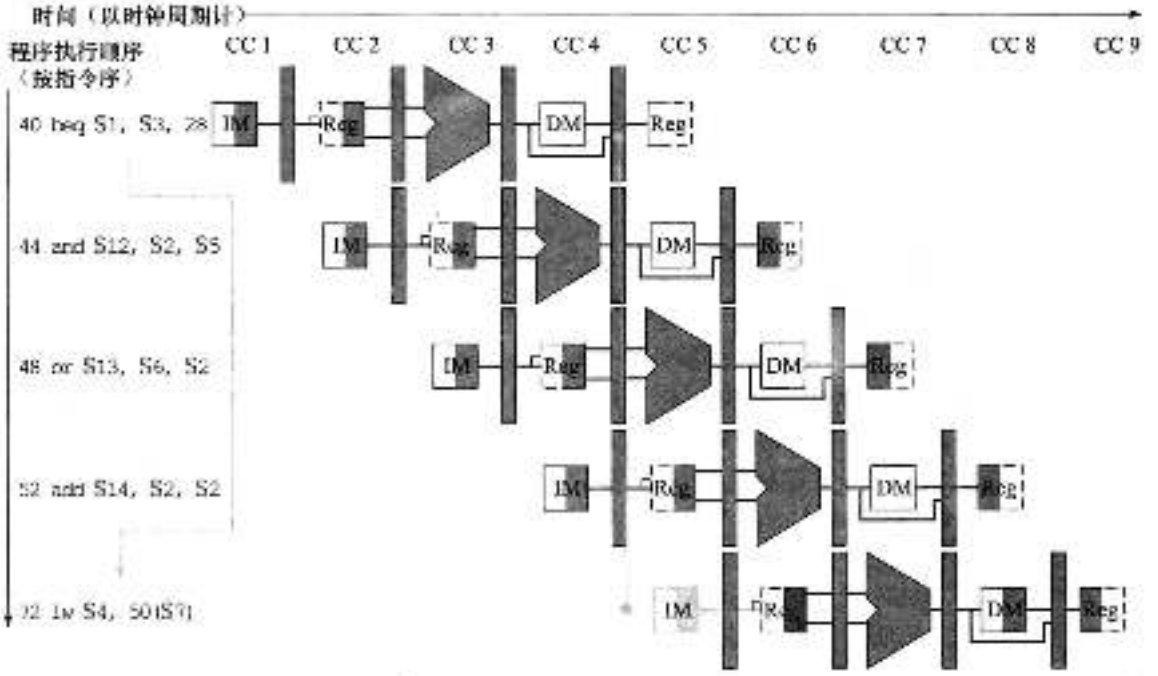



图 4-61 分支指令对流水线的影响。指令左边的数字（40，44，…）表示指令的地址。由于分支指令在 MEM 级（beq 指令对应于时钟周期 4）才能决定是否执行分支，分支后面三条指令都将被取回并执行。如果不加干涉，这三条指令将在 beq 指令跳转到地址 72 执行 lw 之前就开始执行了（图 4-31 通过引入额外的硬件从而将控制冒险减少到一个时钟周期，本图使用的是没有经过优化的数据通路）

## 4.8.1 假定分支不发生

如 4.5 节所述，采用阻塞直到分支判断完毕来处理控制冒险的速度实在太慢。一种比较普遍的提高速度的方法是假设分支不发生，并继续执行顺序的指令流。如果分支发生，就丢弃已经读取并译码的指令，并按分支目标继续执行。如果分支不发生的可能性是 50%，同时丢弃指令的代价很小，那么这种优化方法可以将控制冒险的代价减半。

为了丢弃指令，只需要将最初的控制信号置为 0 即可，这一点与阻塞解决装载指令的数据冒险类似。其不同之处在于当分支到达 MEM 级时必须分别改变在 IF、ID 和 EX 级的三条指令的控制信号，而对于装载指令的阻塞只需要将 ID 级的控制信号置为 0，并将其从流水线中退出即可。分支冒险中的丢弃指令意味着必须能够将流水线的 IF、ID 和 EX 级的指令都清除（flush）。

⊃ 清除：因发生了意外而丢弃流水线中的指令。

## 4.8.2 缩短分支的延迟

一种提高分支效率的方法是缩短分支的执行时间。直到现在，我们都假设在 MEM 级才能确定分支结构要执行的下一条指令的 PC。确定分支目标地址的时间越早，需要清除的指令就越少。MIPS 体系结构是面向支持快速的单周期分支设计的。设计者注意到许多分支仅仅需要简单的判断（如相等或正负），这些简单的判断并不需要完整的 ALU 操作而仅使用简单的一些逻辑门就足够了。如果分支条件更复杂，一般有一条单独的指令使用 ALU 来进行比较——这种情况类似于第 2 章中提到的分支条件码。

为了将分支决策提前，需要提前两个动作：计算分支目标地址和判断分支条件。分支目标

地址的计算是比较简单的。我们在 IF/ID 流水线寄存器中已经有了 PC 的值和立即数字段，所以只需要将分支地址计算从 EX 级移到 ID 级就可以了。当然，尽管分支目标地址对所有指令都会计算，但仅在需要时才会使用。

判断分支条件比较复杂。为了判断分支的执行条件，需要比较从 ID 级取到的两个寄存器的值是否相等。判断相等的方法可以是先将对应的位进行异或操作，然后将结果按位进行或操作。为了把分支条件判断提前到 ID 级，还需要额外的旁路和冒险检测硬件，因为分支条件的判断可能依赖于还在流水线中的结果。例如，为了实现相等则分支（或不等则分支），我们需要旁路结果至 ID 级进行相等检测。这里有两个比较复杂的因素：

317 – 318

1）在 ID 级指令译码后，决定是否需要将所需数据旁路到相等检测单元进行相等检测。如果是分支指令，就可以把 PC 替换为分支目标地址。旁路分支指令某操作数以前是由 ALU 旁路单元来完成的，但 ID 级相等检测单元的引入需要一个新的旁路单元。必须注意的是，需要旁路的分支指令源操作数可能来自 ALU/MEM 或 MEM/WB 流水线寄存器。

2）因为 ID 级进行分支比较所需的数据可能在后面才能产生，因此有可能会发生数据冒险，这样就需要阻塞流水线。例如，如果分支指令前刚好是一条 ALU 指令，而这条 ALU 指令的结果恰是分支指令比较所需要的，那么必然产生阻塞，因为 ALU 指令的 EX 级将在分支指令的 ID 级后发生。再举一个例子，如果分支指令前刚好是一条装载指令，而装载指令的结果恰是分支指令判断所需要的，则必须产生两个阻塞，因为装载指令的结果将在装载指令的 MEM 级结束时产生，但在分支指令的 ID 级开始时就会用到。

尽管有这些困难，将分支执行提前到 ID 级依然是值得的，因为它将分支预测错误的代价减小到只有一条指令，就是分支执行时正在取的那条指令。下面的例题对旁路路径和检测冒险的实现细节进行了讨论。

为了在 IF 级清除指令，我们加入了一条称为 IF.Flush 的控制信号，即将 IF/ID 流水线寄存器的指令字段置为 0。清零寄存器的结果是将预取到的指令转变成为空指令，该指令不作任何操作，不改变机器状态。

**例题 • 流水线分支**

假定流水线对分支不发生进行了优化，并且分支的执行提前到流水线的 ID 级。试说明下面的指令序列在分支发生时的执行情况：

```
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC relative branch to 40+4+7*4=72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```

**答案**

图 4-62 描述了分支产生时指令序列的执行情况。与图 4-61 不同，这里在一个发生的分支
319 上只有一个流水线气泡。 □

## 4.8.3 动态分支预测

假设分支不发生是一种粗略的分支预测方法。在这种情况下，我们总是预测分支不发生，如果预测错误就清空流水线。对简单的五级流水线而言，这种方法结合基于编译器的预测就已经足够了。从时钟周期数的角度来说，使用更深的流水线时分支代价将增加。类似地，以丢弃的指令数来计算，对多发射（见 4.10 节）的分支代价也将增加。这种组合意味着在一个高性

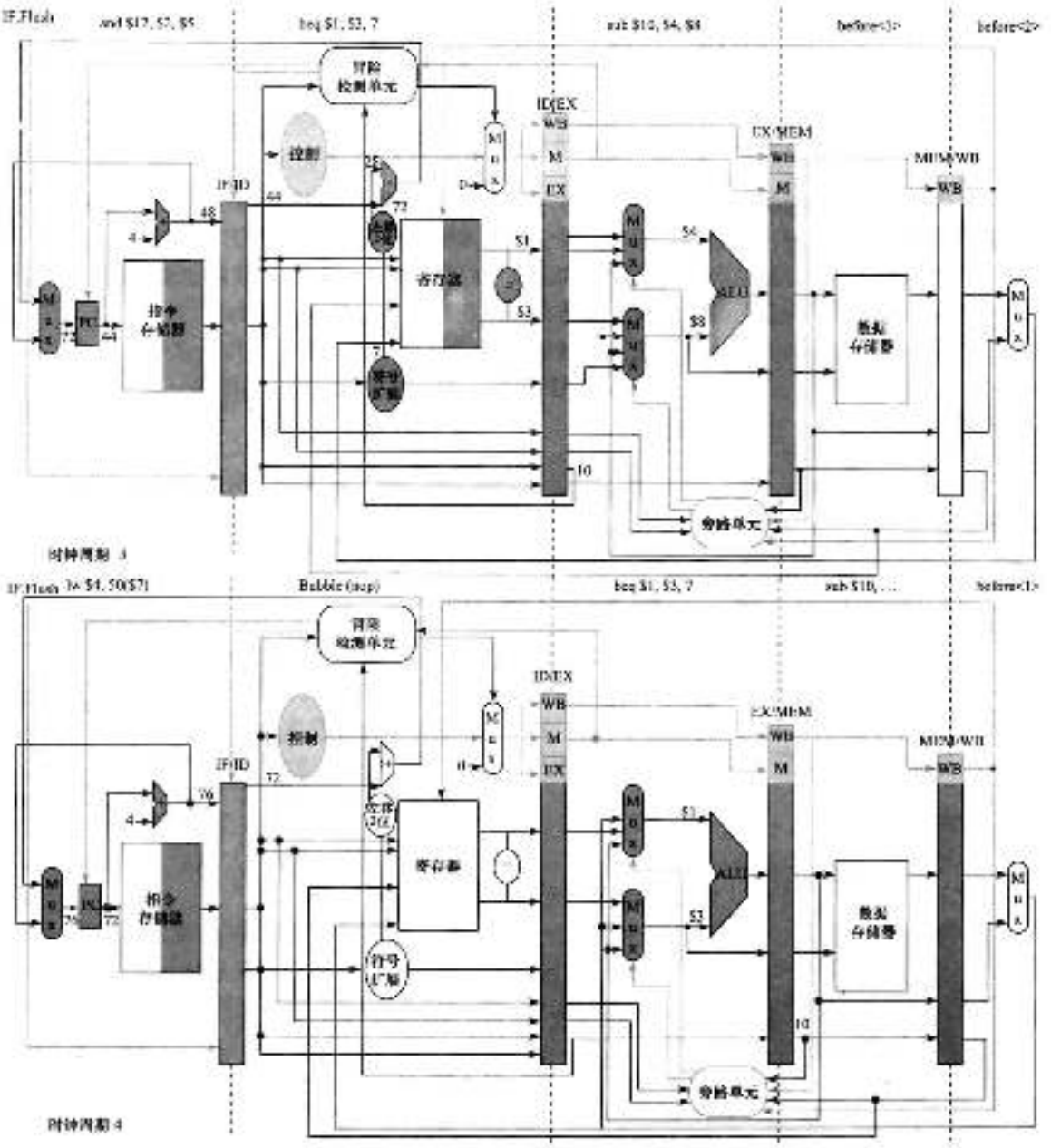



图 4-62 在第三个时钟周期 ID 级确定分支发生，因此地址 72 被选为下一个 PC 地址，同时将为下一个时钟周期预取的指令置为 0。时钟周期 4 的图描述了地址为 72 的指令被取回，并且分支发生的后果是在流水线中产生了一个气泡或者一条空指令（由于空指令实际上是 sll $0, $0, 0，所以时钟周期 4 的 ID 级是否应该标出还有待商榷）

能的流水线设计中，简单的静态预测机制将可能浪费大量的性能。如 4.5 节所述，如果有更多的硬件支持，我们就可能实现一些其他的分支预测方法。

一种策略是通过查找指令的地址观察上一次执行该指令时分支是否发生，如果上次执行时分支发生，就从上次分支发生的地方开始取新的指令。这种技术称为**动态分支预测**（dynamic branch prediction）。

这种策略的一种实现方法就是采用**分支预测缓存**（branch prediction buffer）或**分支历史记录表**（branch history table）。分支预测缓存是一小块按照分支指令的地址低位索引的存储器区，

其中包括一位或多位数据用以说明最近是否发生过分支。

- 动态分支预测：根据运行信息在运行中进行分支预测。
- 分支预测缓存：也称为分支历史记录表。一小块按照分支指令的低位地址索引的存储器区，其中包括一位或多位数据用以说明最近是否发生过分支。

这是最简单的一类缓存，我们实际上并不知道预测是否正确，而只它还可能由其他具有相同地址低位的分支设置，但这并不影响这种方法的准确率。预测只是对正确分支方向的一种假设，在这个基础上，沿着预测的方向进行取指。如果这种假设错误，预测错误的指令将被删除，预测位将取反，并返回原来的位置，继续按照正确的方向取指并执行。

使用一位预测位的简单预测方法具有性能上的缺陷：即使一个分支几乎总是发生，但它一旦未发生就将导致两次（而不是一次）预测错误。下面的例子说明了这种情况。

**例题** 循环与预测

我们来看一个循环分支，它在一行代码上的分支发生了9次，而不是发生了一次。假设分支的预测位保存在预测缓存中，这种分支预测的正确率是多少？

**答案**

静态预测方法会在第一次和最后一次的循环迭代时预测错误。由于分支在一行上发生了9次，因此预测位在最后一次循环时被设为分支发生，而且这次预测错误是不可避免的。而在第一次迭代时发生预测错误是因为预测位在循环的上一次迭代时被前一个执行设置为不执行（在那次退出的迭代中分支并没有发生）。因此这个预测方法在分支发生90%的情况下预测的正确性只有80%（两次错误预测，8次正确预测）。 □

320 ~ 321

在理想的情况下，在这种高度规律的分支结构中预测的正确性与发生分支的频率相匹配。为了弥补这一缺陷，经常使用两位预测位的方案。在一个两位预测位的方案中，两次发生预测错误时才改变预测。图4-63给出了两位预测位的有限状态机。

分支预测缓存可以使用在IF级指令地址能够访问的小容量专用缓存实现。如果指令预测分支发生，那么一旦获得新的PC就从该目标地址开始取指（如4.8.3节所述，在ID级就可以获得PC），否则就顺序取指并继续执行。如果预测的结果是错误的，就按照图4-63说明的方法改变预测位。

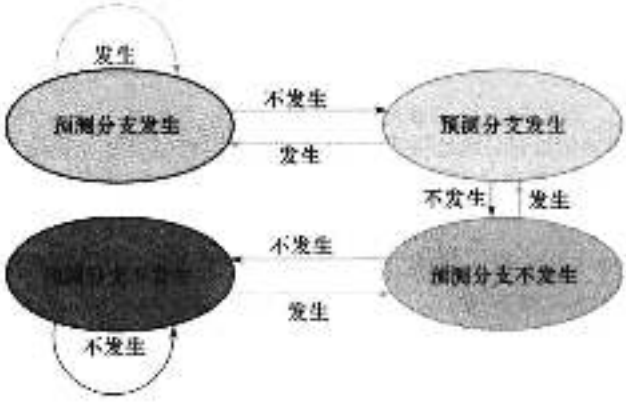



图4-63 两位预测位机制的状态图。通过使用两位（不是一位）预测位，在分支经常发生或经常不发生的情况下（大多数分支都是这样）只会发生一次预测错误。两位数据在系统中可以表示4种状态。这种两位方案是基于计数器预测方法的一个应用。基于计数器的预测方法是当预测成功时计数器加1，预测失败时计数器减1，然后使用计数器表示范围的中点作为分支与不分支的分界点

**精解** 如4.5节所述，在五级流水线中，通过延迟定义分支，我们可以将控制冒险等化为一种可调节特性。延迟分支可执行下一条指令，但分支指令后的第二条指令仍将受到分支的影响。

编译器和汇编器都会试图把总在分支后执行的那条指令放入分支延迟时间槽（branch delay slot）。这些软件的作用就是使后续的指令有效并且有用。图 4-64 给出了三种调度分支延迟时间槽的方法。

延迟分支调度的限制在于：对能够被调度到分支延迟时间槽中的指令的限制；编译时对分支发生与否的预测能力。

对每个时钟周期发射一条指令的五级流水线处理器而言，延迟分支是一种简单有效的方法。随着处理器向更深流水线以及单周期多指令的方向发展（见 4.10 节），分支延迟变得更长，单延迟时间片实际上并没有多大作用。所以，与开销大但更灵活的动态预测方法相比，延迟分支技术已经失去了吸引力。同时，根据摩尔定律使动态预测的成本相对更低。

⇒ 分支延迟时间槽：紧跟延迟分支指令的时间片。在 MIPS 体系结构中，用不影响分支的一条指令填充到该时间片中。 322

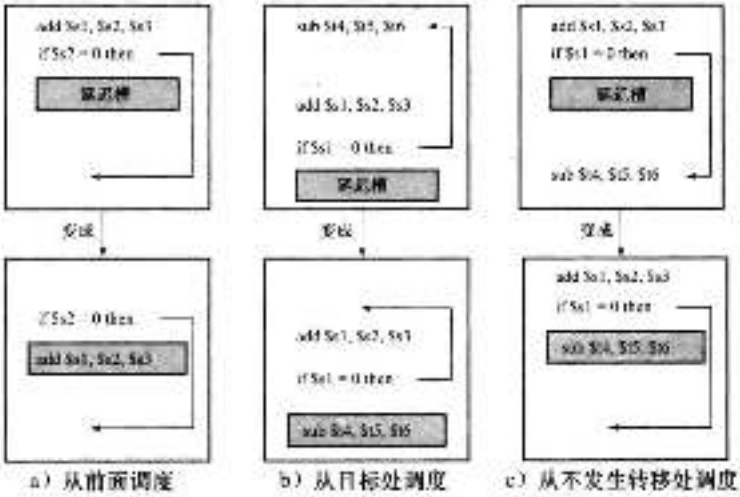



图 4-64 分支延迟时间槽的调度。每一对方框中的上面一个表示调度前的代码，下面一个表示调度后的代码。在方案 a 中，延迟时间片通过插入分支之前的一条与分支无关的指令实现，这是一种最佳的选择。当方案 a 无法实现时，就使用方案 b 和方案 c。在方案 b 和方案 c 的代码序列中，分支条件中使用了 $1，因而不能将 add 指令（它的目的寄存器是 $1）移入分支延迟时间槽。方案 b 中的分支延迟时间槽是按照分支目标地址调度的。由于目标指令可以通过其他路径访问到，通常需要将它们进行复制。当分支发生的可能性比较大时，一般选择方案 b，如循环分支。最后，也可能采用方案 c 预测分支不发生的下一条指令进行调度。为了使方案 b 和方案 c 中的优化合法，sub 指令必须在分支预测错误时也能“正常”执行。“正常”意味着虽然有些工作是多余的，但程序依然能够正确执行。例如，当分支预测错误且 $t4 是未被使用的临时寄存器时，就是这种情况 323

**01 精解** 分支预测器告诉我们分支是否会发生，但我们依然需要计算分支目标地址。在五级流水线中，计算分支目标地址需要一个时钟周期，即分支发生将需要一个时钟周期的开销。延迟分支是消除这个开销的一种方法。另一种方法是使用分支目标缓存（branch target buffer）保存分支目标地址或分支目标指令。

两位的动态预测机制仅使用某个特定分支的信息。研究人员发现，在使用相同数量的预测位的情况下，同时使用局部分支和最近执行分支的全局行为信息，能够产生更高的预测精度。这种预测器称为相关预测器（correlating predictor）。一个典型的相关预测器为每个分支提供两个两位的预测器，其选择依据是上次分支执行的结果（分支发生与否）。这样，全局分支行为可以被看成是在预测查找表中加入额外的索引位。

最新的分支预测方法是竞争预测器。竞争预测器（tournament branch predictor）对每

个分支使用多个预测器，并记录哪个预测器的预测结果最好。目前竞争预测器的预测是最准确的。典型的竞争预测器对每个分支进行两个预测：一个基于局部信息，一个基于全局分支行为。有一个选择器用于选择哪个预测器的预测结果，选择作类似于一位或两位的预测器，而两位预测器的精度更高。一些最新的微处理器使用了这种预测器。

- 分支目标缓存：一种用于缓存分支目标地址或分支目标指令的结构，其一般形式为带标志位的 cache，因而其硬件开销大于简单的分支预测缓存器。
- 相关预测器：综合考虑特定分支的局部行为和最近执行分支的全局行为的分支预测器。
- 竞争预测器：具有多种预测机制的分支预测器，其带有一个选择器，对给定分支可选择其中一个作为预测结果。

**小贴士** **精解** 一种减少条件分支数量的方法是加入条件移动指令（conditional move instruction）。不同于条件分支指令改变 PC 值，条件移动指令将根据条件改变移动的目的寄存器。如果条件不成立，条件移动指令就相当于一条 nop 指令。例如，某版本的 MIPS 体系结构指令集包含 movn（move if not zero）和 movz（move if zero）两条指令。例如，movn $8, $11, $4，如果寄存器 $4 的值为非零，该指令复制寄存器 $11 的内容至寄存器 $8；否则，该指令什么也不做。

ARMv7 指令集在绝大多数指令中都有条件字段。因此，ARM 程序一般比 MIPS 程序的条件分支要少一些。

### 4.8.4 流水线小结

我们从洗衣店的例子开始，介绍了日常生活中的流水线原理。用这个例子类比，逐步解释了指令的流水化，即在单周期数据通路的基础上逐步增加流水线寄存器、旁路路径、数据冒险检测、分支预测和异常时指令的清除。图 4-65 给出了最终的数据通路及控制。现在我们已经准备好处理另一种控制冒险：异常。

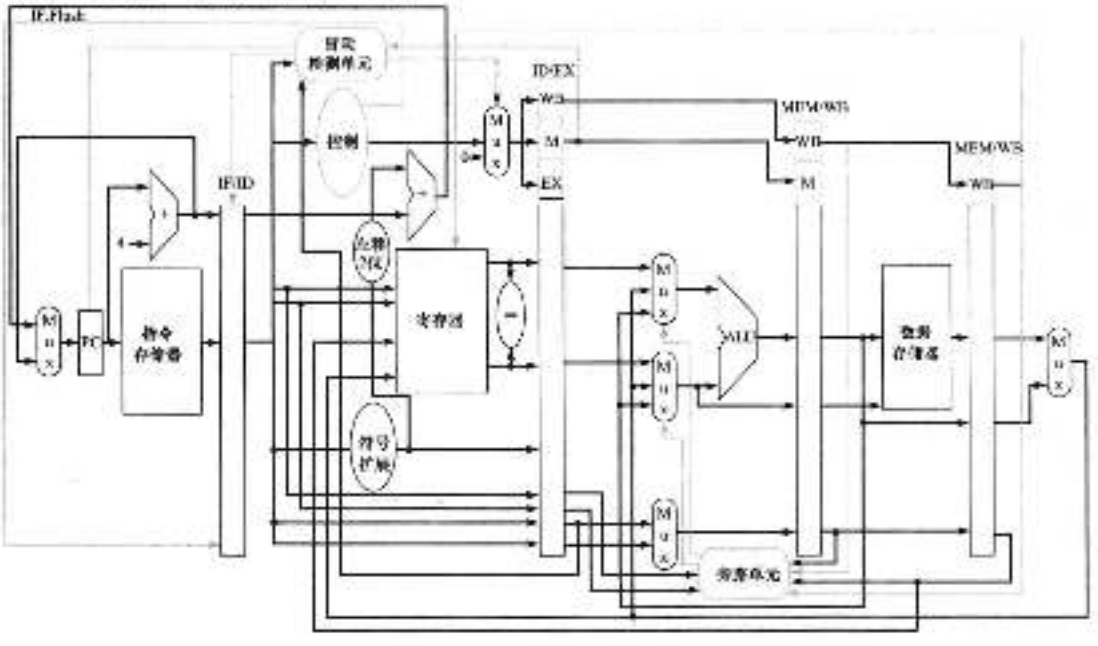



图 4-65 本章最终的数据通路与控制。注意，这是一个概略图，没有覆盖到数据通路的所有细节，如图 4-57 中的 ALUsrc 多选器和图 4-51 中的多选器控制都没有标识出来

**小测验**

考虑三种分支预测机制：预测分支不发生、预测分支发生和动态分支预测。假定它们在预测正确时无开销，预测错误时开销为两个时钟周期，动态预测器的平均准确率为90%。在此情况下，对下面的分支而言哪种预测器是最好的选择？ 324

1. 分支发生概率为5%。
2. 分支发生概率为95%。
3. 分支发生概率为70%。

## 4.9 异常

> 使一台计算机具有自动程序中断能力并非一件简单的事，因为中断发生时处于不同执行阶段的指令数量可能非常多。
>
> —— *Fred Brooks, Jr.,《Planning a Computer System: Project Stretch》, 1962*

控制是处理器设计中最具挑战性的一个方面：它最难达到正确，也最难提高速度。控制中最难的部分之一是实现**异常**（exception）和**中断**（interrupt）——除分支以外改变正常指令执行顺序的事件。异常和中断最初是用来处理来自处理器内部的意外事件，如算术溢出。在第5章中我们将看到，它们也可用于I/O部件与处理器的通信。

- 异常：也称为中断，指打断程序正常执行的突发事件，用于检测溢出等。
- 中断：来自处理器外部的异常。（某些体系结构也用“中断”一词表示所有的异常。）

许多体系结构和作者不区分中断和异常，统称为中断，如Intel x86。我们遵循MIPS的习惯，术语异常指控制流中任何意外的改变，而无论其产生原因是来自处理器内部还是外部，术语中断指由外部引起的事件。下面5个例子说明了在处理器内部或外部的事件情况。

| 事件类型 | 来源 | 对应的MIPS术语 |
|---|---|---|
| I/O设备请求 | 外部 | 中断 |
| 用户程序进行操作系统调用 | 内部 | 异常 |
| 算术溢出 | 内部 | 异常 |
| 使用未定义的指令 | 内部 | 异常 |
| 硬件故障 | 内部或外部 | 异常或中断 |

导致异常发生的不同情况对异常处理的支持提出了诸多要求。在第5章中我们将再次讨论这个话题，从而更加清楚地理解这一问题。本节讨论两种异常的检测机制，这两种异常由我们讨论过的指令集及其实现方式产生。

检测异常情况并采取适当举措，通常处于处理器的关键路径上。该路径决定了时钟周期的长度以及处理器性能。如果在控制单元的设计中没有充分考虑异常，那么在复杂实现中加入异常支持会明显降低性能，并使正确的设计更加复杂。

### 4.9.1 MIPS体系结构中的异常处理

目前的实现中可能产生的两种异常是未定义指令的执行和算术溢出。在接下来的部分，我们将使用 `add $1, $2, $1` 指令作为算术溢出类型异常的例子。异常发生时处理器必须进行的基本操作是：在异常程序计数器（Exception Program Counter，EPC）中保存出错指令的地址，并把控制权转交给操作系统的特定地址。 325 | 326

操作系统可采取适当的行动，如给用户程序提供一些服务，对溢出情况进行事先定义的操作，或者终止程序的执行并报告错误。在完成处理异常所需动作后，操作系统可以终止程序，

也可以继续执行程序，此时由 EPC 决定重新开始执行的地方。在第5章将更详细地讨论重新开始执行的问题。

为了处理异常，操作系统除了要知道是哪条指令引起异常之外，还必须知道引起异常的原因。主要有两种方法用于表示产生异常的原因。MIPS 使用的方法是设置一个状态寄存器（称为 Cause 寄存器），其中有一个字段用于记录异常产生的原因。[⊖]

另一种方法是使用**向量中断**（vectored interrupt）。在向量中断中，控制权被转移到由异常原因决定的地址处。[⊜]例如，为处理前面的两种异常，可定义如下的两个异常向量地址：

⊜ 向量中断：由异常原因决定中断控制转移地址的中断。

| 异常类型 | 异常向量地址（十六进制） |
|---|---|
| 未定义指令 | $8000\ 0000_{16}$ |
| 算术溢出 | $8000\ 0180_{16}$ |

操作系统根据引起异常的地址得知导致异常的原因。地址由 32 字节或 8 条指令进行区分，并且操作系统必须记录异常的原因，并依此顺序执行一些有限的处理。当出现的异常不属于向量异常时，单个入口点供所有异常使用，并且操作系统对状态寄存器进行译码以便找到原因。

通过给基本的实现加上一些额外的寄存器和控制信号，并将控制进行一些扩展，就可以处理异常。假定我们实现的是 MIPS 体系结构的异常处理系统，统一入口地址为 $8000\ 0180_{16}$（事实上，实现向量异常也不难），需要给当前的 MIPS 实现加上两个寄存器：

- EPC：32 位寄存器，用于保存发生异常的指令地址（向量中断也需要这样一个寄存器）。
- Cause：记录异常原因的寄存器。在 MIPS 体系结构中它是 32 位的，虽然其中一些位现在还没有用到。假定使用一个 5 位的字段对前面两种异常原因进行编码：未定义指令 = 10，算术溢出 = 12。

## 4.9.2 在流水线实现中的异常

在流水线实现中，异常可被视作另一种形式的控制冒险。例如，假设指令 add 产生了一个算术溢出。正如上一节对分支发生的处理，我们必须清除流水线中 add 指令后的一系列指令并从新的地址开始取指。我们将使用与之相同的机制，不过这次是由异常重置控制信号。

327

在处理分支预测错误时，我们已经知道如何通过将 IF 级的指令转换成 nop 指令来清除指令。为了清除 ID 级的指令，我们使用 ID 级已有的多选器，将控制信号清零以产生阻塞。一个称为 ID.Flush 的新控制信号与冒险检测单元的阻塞信号相或，可以在 ID 级进行清除。为了清除 EX 级的指令，我们使用一个称为 EX.Flush 的新信号，用它控制新的多选器将控制信号清零。为了从地址 $8000\ 0180_{16}$（MIPS 异常地址）开始取指令，只要简单地加入一个额外的输入到 PC 的多选器，由它将 $8000\ 0180_{16}$ 传送到 PC。图 4-66 具体描述了这种变化。

这个例子指出了异常存在的一个问题，即如果不在指令执行期间中止指令的执行，程序员将无法看到导致溢出的寄存器 $1 中的原始值，因为它将作为指令 add 的目标寄存器被冲掉。这一问题可以通过下面的方法解决：异常溢出在 EX 级检测出来，可用 EX.Flush 信号避免 EX 级的指令在 WB 级写回结果。许多异常需要我们能够最终正常执行引起异常的指令。做到这一点最简单的方法是先清除这条指令，然后在异常处理完后再重新执行这条指令。

异常处理的最后一步是将导致异常的指令的地址保存到 EPC 中。实际上，我们保存的是原始地址 +4，因此异常处理例程必须先从保存的地址中减去 4。图 4-66 给出了一个数据通路，其中包括分支硬件以及为处理异常所进行的必要调整。

---

⊖ 所有异常使用同一入口地址，操作系统根据状态寄存器确定异常原因。——译者注

⊜ 操作系统通过异常向量地址得知异常原因。——译者注

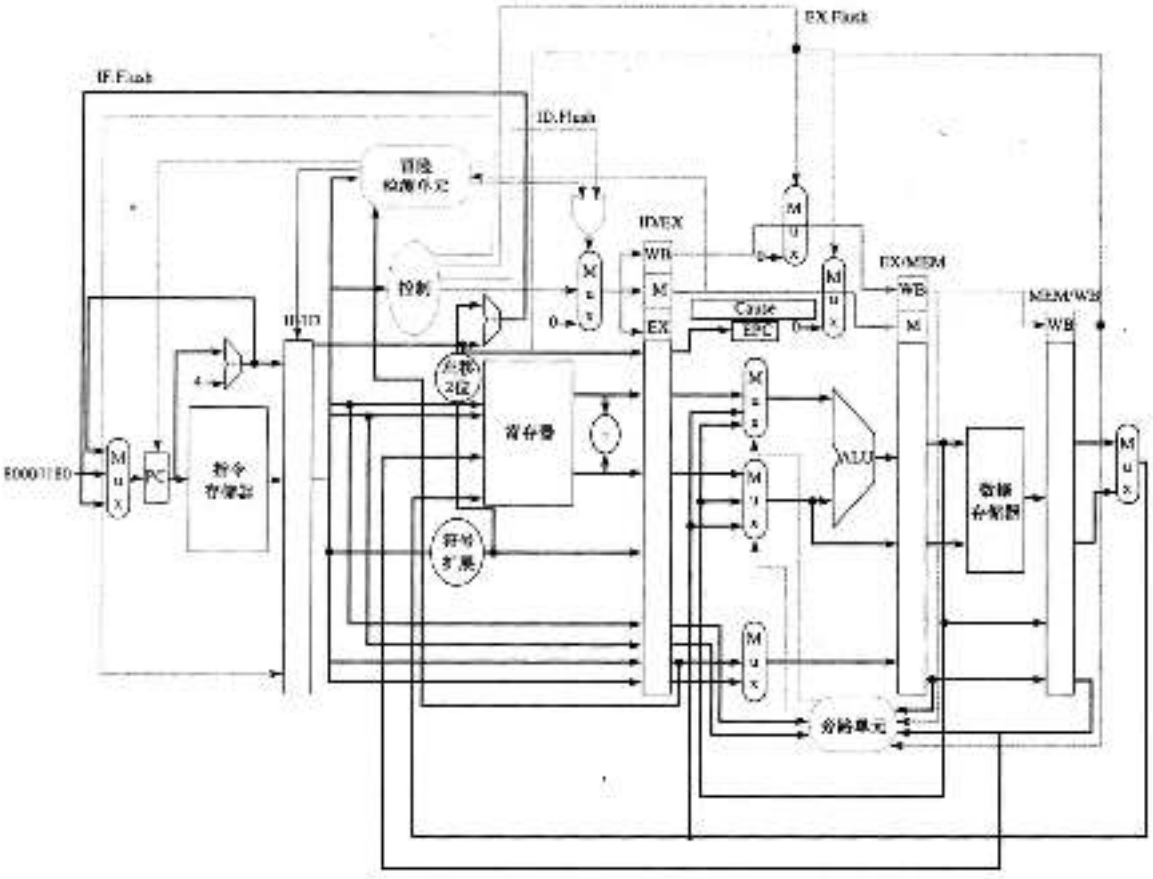



图 4-66 处理异常的数据通路与控制。主要增加了以下部分：在 PC 多选器中增加了一个新的输入 $8000\ 0180_{16}$、一个记录异常发生原因的 Cause 寄存器以及一个保存导致异常的指令地址的 EPC 寄存器。$8000\ 0180_{16}$ 是发生异常时开始取指令的地址。尽管图中没有表示出 ALU 溢出信号，但它也是控制单元的一个输入

### 例题：流水线计算机中的异常

给出以下指令序列：

```
40hex  sub  $11, $2, $4
44hex  and  $12, $2, $5
48hex  or   $13, $2, $6
4Chex  add   $1, $2, $1
50hex  slt  $15, $6, $7
54hex  lw   $16, 50($7)
...
```

假定异常处理程序的开始部分如下：

```
80000180hex  sw   $26, 1000($0)
80000184hex  sw   $27, 1004($0)
...
```

给出 add 指令发生溢出异常时流水线的情况。

### 答案

图 4-67 给出了从 add 指令的 EX 级开始发生的情况。溢出在 EX 级被检测到，$8000\ 0180_{16}$ 被强制放入 PC。在第 7 个时钟周期，add 指令及其后面的指令被清除，并且异常代码的第一条指令被取出。注意，保存的地址是 add 指令下一条指令的地址（$4C_{16}+4=50_{16}$）。

328 ~ 329

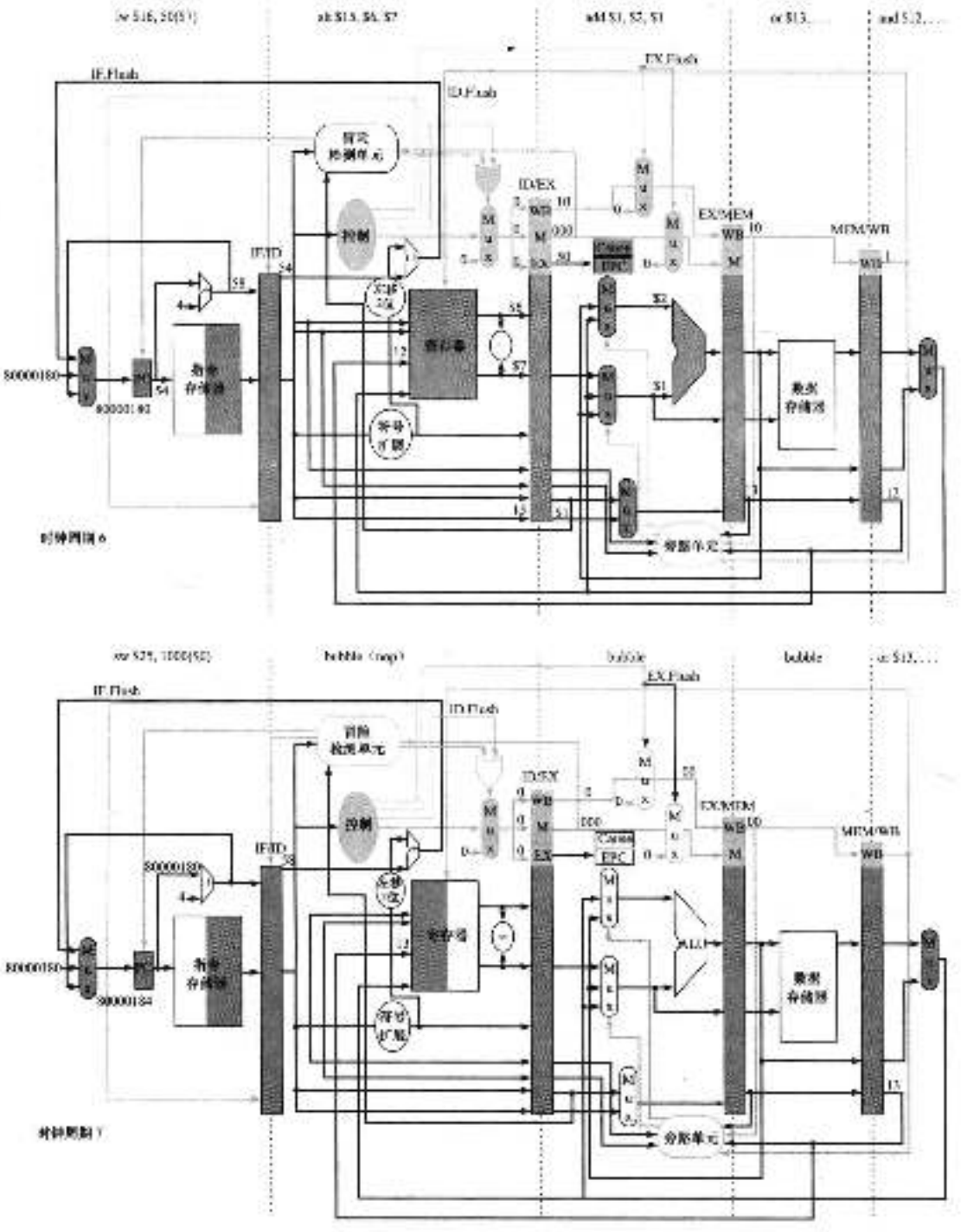



图 4-67 add 指令算术溢出导致的异常。溢出在第 6 个时钟周期的 EX 级检测到，因此将 add 后面的指令地址（$4C + 5 = 50_{16}$）保存到 EPC 寄存器。溢出导致在该周期后面所有的 Flush 信号都设置为 1，并置 add 的控制信号为无效（置为 0）。时钟周期 7 显示了流水线中转化为气泡的指令和取异常处理程序的第一条指令 sw $25, 1000（$0）（从指令地址 $8000\ 0180_{16}$ 处取得）。需要注意的是，位于 add 指令前的 AND 指令和 OR 指令仍然会执行完毕。虽然图中没有画出 ALU 溢出信号，但它也是控制单元的一个输入 □

在前面我们曾提到5个异常的例子，在第5章我们还会看到其他的例子。任何时钟周期流水线中都有5条活动的指令，问题是如何确定到底是哪条指令引起了异常。而且，一个时钟周期内还可能发生多个异常。通常的解决方法是对异常划分优先级，这样多个异常同时发生时就知道先处理哪个。在大多数MIPS实现中，硬件对异常进行排序从而使得最先发生异常的指令被中断。

I/O设备请求与硬件故障并不与特定的指令相关，因此它们在流水线中断时机的实现上具有一定的灵活性。因此，用于其他异常的机制在这里也可以很好地工作。

EPC捕捉中断指令的地址，而MIPS的Cause寄存器在一个时钟周期内记录下所有可能的异常，因此异常处理软件判断出该指令发生了何种异常。一个重要的判断依据是某一类异常可能在哪一个流水线阶段发生。例如，未定义的指令异常发生在ID级，而调用操作系统异常发生在EX级。如果在Cause寄存器中保存有多个异常，当优先级最高的异常处理之后，会继续导致硬件中断，从而处理后面的异常。

**硬件/软件接口** 硬件与操作系统必须协同工作才能按照我们期望的方式处理异常。硬件一般暂停指令流中导致异常的指令，同时执行完该指令前的所有指令，清除该指令后的所有指令，并且设置一个寄存器指记异常发生的原因，保存导致异常发生的指令的地址，然后跳转到预先确定的地址开始执行。操作系统则查看异常发生的原因并采取相应的操作。对于一个未定义指令异常，硬件错误异常或算术溢出异常，操作系统通常终止执行的程序并返回原因描述。对于I/O设备请求或操作系统服务调用，操作系统保存程序的当前状态，执行期望的任务，然后重新载入程序继续运行。在I/O设备请求的情况下，我们可能需要在继续执行发出I/O设备请求的任务前先运行另一个任务，因为该任务一般在I/O完成之后才能继续执行。这就是保存和恢复任务状态如此重要的原因。一个最重要且频繁出现的异常是页缺失与TLB异常。第5章描述了更多关于这些异常及其处理的细节。

**精解** 在流水线计算机中将每一个异常与导致异常的相应指令对应起来的难度很大，因此一些计算机设计者在一些非关键情况下放松了这种要求，这种处理器一般称为具有非精确中断（imprecise interrupt）或者非精确异常（imprecise exception）。在上面的例子中，尽管导致异常的指令地址是$4C_{16}$，但在检测到异常后下一个时钟周期开始时PC的值通常为$58_{16}$。具有非精确异常处理的处理器可能会将$58_{16}$放入EPC中，而让操作系统确定是哪一条指令导致了异常。MIPS以及当前的大量主流处理器都提供精确中断（precise interrupt）或精确异常（precise exception）（我们将在第5章中看到，原因之一是为了支持虚拟存储器）。 331

- 非精确中断：也称为非精确异常。流水线计算机中的中断或异常不与导致中断或异常的指令精确地关联。
- 精确中断：也称为精确异常。流水线计算机中的中断或异常与导致中断或异常的指令精确地关联。

**精解** 尽管MIPS对绝大多数异常使用$8000\ 0180_{16}$作为异常入口地址，但为了提高性能，对TLB缺失异常使用$8000\ 0000_{16}$作为异常入口地址（参见第5章）。

**小测验**

在下面的指令序列中会首先识别哪个异常？

```
1. add $1, $2, $1        #算术溢出
2. XXX $1, $2, $1        #未定义指令
3. sub $1, $2, $1        #硬件错误
```

## 4.10 指令级并行

首先说明一下，本节是对一些高级主题的概述。如果你希望了解更多的细节，可以参考另一本教材：《Computer Architecture: A Quantitative Approach》第5版。本节大约13页的内容在该书中扩充到近200页（含附录）。

**流水线**挖掘了指令间潜在的并行性。这种并行性被称为**指令级并行**（instruction-level parallelism，ILP）。有两种方法可以增加潜在的指令级并行程度。第一种是增加流水线的深度以重叠更多的指令。还是用洗衣店的例子来说明，假设洗衣机周期比其他机器的周期要长，我们可以把洗衣机划分成三个机器，分别完成原洗衣机洗、漂、甩三个功能。这样我们就将四级流水线变成了六级流水线。为了达到完全的加速效果，我们需要重新平衡其他步骤使得它们的长度相同，在处理器和洗衣店中都是这样。因为更多的操作被重叠，有更多的并行性被挖掘出来。因为时钟周期缩短的缘故，性能会得到潜在的增强。

另一种方法是复制计算机内部部件的数量，使得每个流水级可以启动多条指令。这种技术一般被称为**多发射**（multiple issue）。一个多发射的洗衣店会把原有的一台洗衣机和烘干机替换为三台洗衣机和三台烘干机。还需要雇用更多的洗衣工来折叠和存储三倍于原来的衣服。这种方法的缺点是需要额外的工作让所有机器同时运转并将负载传到下个流水级。

- 指令级并行：指令间的并行性。
- 多发射：一种每时钟周期内发射多条指令的机制。

332

每个阶段同时启动多条指令允许指令执行速率超过时钟速率，换句话说，就是CPI小于1。正如第1章介绍的，有时候使用IPC，即每时钟周期执行的指令数作为度量会更方便。例如，一个4GHz四路多发射微处理器能以每秒160亿指令的峰值速率执行，其最好情况下的CPI达到0.25，IPC达到4。假设是五级流水线，这个处理器任何时刻都可能有20条指令在同时执行。现在的高端微处理器尝试在每个时钟周期发射3～6条指令，甚至中端设计的目标都是峰值IPC为2。然而，一般来说存在很多约束，例如哪些类型的指令可以同时执行，当相关性增加时会发生什么新的问题等。

实现一个多发射处理器主要有两种方式，其区别是将主要工作分给编译器来做还是硬件来做。由于不同的实现方式将导致某些决策是静态进行的（在编译时）还是动态进行的（在执行时），所以这两种方式有时也被称为**静态多发射**（static multiple issue）和**动态多发射**（dynamic multiple issue）。正如我们即将见到的，两种方式还有其他别名。

多发射流水线必须处理以下两个问题：

1）往**发射槽**（issue slot）中发射多条指令：处理器如何确定在给定的时钟周期发射多少条指令以及发射何种指令呢？在大多数静态发射处理器中，这个过程至少有很大一部分是由编译器处理的。而在动态发射处理器中，这个问题一般是由处理器在运行时处理的，尽管编译器也会尽其所能通过调整指令顺序加以协助。

2）处理数据冒险和控制冒险：在静态发射处理器中，部分甚至全部的数据冒险和控制冒险是由编译器静态处理的。相反，绝大多数的动态发射处理器通过硬件技术在执行时至少消除某些类别的冒险。

尽管这里我们把它们看成两种不同的方法，实际上这两种方法经常借用对方的技术，没有哪一种方法可以称得上是完全独立的。

- 静态多发射：实现多发射处理器的一种方法，其中决策是在执行前的编译阶段做出的。
- 动态多发射：实现多发射处理器的一种方法，其中决策是由处理器在执行阶段做出的。
- 发射槽：在给定时钟周期内能够发射指令的位置，可以类比于短跑比赛中的起点位置。

### 4.10.1 推测的概念

推测是一种寻找和挖掘更大 ILP 的最重要的方法。**推测**（speculation）是一种为了使依赖于被推测指令的其他指令可以执行，而允许编译器或处理器“猜测”指令结果的方法。例如，我们可以推测分支指令的结果，这样分支后的其他指令就可以提前执行了。另一个例子是假设 load 指令前有一条 store 指令，我们可以推测它们不对同一存储器地址进行访问，这样就可以把 load 指令提到 store 指令前执行。推测的问题在于可能会猜错。所以，任何推测技术必须包含一种机制，它能检查推测的正确性并在推测错误时能取消根据原推测结果执行指令的影响。实现这种回卷能力增加了额外的复杂性。 333

⊃ 推测：一种编译器或处理器推测指令结果以消除执行其他指令对该结果依赖的方法。

推测可以由编译器或硬件来完成。例如，编译器可以利用推测对指令进行重排序，将一条指令移过分支，也可将 load 指令与 store 指令交换。使用本节后面讨论的技术，处理器硬件可以在运行时实现同样的变换。

推测错误时的恢复机制对软硬件是非常不同的。对软件来说，编译器经常插入额外的指令检查推测的正确性并提供专门的修复例程供推测错误时使用。对硬件来说，处理器经常缓存推测的结果直至推测的结果得到确认。如果推测是正确的，缓存的结果写回寄存器堆和存储器。如果推测是错误的，硬件将清除缓存并重新执行正确的指令序列。

推测还可能导致另一个问题：对某些指令的推测会导致原本不存在的异常发生。例如，假设推测执行一条装载指令，但是在推测错误的情况下，该指令所使用的地址是非法的。结果，一个本不应该发生的异常发生了。这个问题之所以复杂是因为，如果这条装载指令本来不是推测执行，那么该异常必然发生。在基于编译器的推测中，这类问题的处理方法是加入额外的推测支持，使得这样的异常暂时被忽略，直至可以确定异常会发生为止。在基于硬件的推测中，异常被简单地缓存起来，直到导致异常的指令确定会执行。在异常真正发生时，就会执行正常的异常处理程序。

推测在设计正确时能改善性能，而不慎使用可能降低性能，所以需要做大量的工作来决定何时采用推测更为合适。在本节的后半部分，我们将介绍静态和动态的推测技术。

### 4.10.2 静态多发射处理器

所有的静态多发射处理器都使用编译器来帮助封装多条指令并处理冒险。在一个静态发射处理器中，可以在给定时钟周期内发射多条指令，也称为**发射包**（issue packet）。发射包可被视为一条完成多个操作的长指令。这种看法不仅是为了类比。因为静态多发射处理器一般对一个时钟周期内能发射的多条指令有所限制，因此把发射包看成允许同时进行很多操作的一条指令是可行的。这种观点引出了这种方法的最初名字：**超长指令字**（Very Long Instruction Word，VLIW）。 334

⊃ 发射包：在一个时钟周期内发射的多条指令的集合。这个包可以由编译器静态生成，也可以由处理器动态生成。

⊃ 超长指令字：一类可以同时启动多个操作的指令集，其中操作在单个指令中相互独立，并且一般都有独立的操作码字段。

绝大多数静态多发射处理器也依赖编译器处理数据冒险和控制冒险。编译器的任务可能包括静态分支预测和代码调度，以减少冒险或阻止所有的冒险。在描述更先进的处理器中所采用的技术之前，先来看一个简单的静态多发射 MIPS 处理器的例子。

### 一个例子：MIPS 指令集的静态多发射

为了感受一下静态多发射，我们考查一个简单的双发射 MIPS 处理器，其中一条指令可以是整型 ALU 操作或分支，另一条指令可以是装载指令或存储指令。在某些嵌入式 MIPS 处理器中就是这么设计的。每个时钟周期发射两条指令意味着需要取回和译码 64 位的指令。在许多静态多发射处理器中，甚至是所有的 VLIW 处理器中，严格限制了可同时发射指令的所处位置以简化译码和发射过程。因此，我们要求两条指令成对地放在一个 64 位对齐的内存区域中，并且 ALU 指令或分支指令必须放在前面。此外，如果找不到另一条与之可以同时发射的指令，就用 nop 指令代替它。这样，指令总是可以成对发射，当然其中可能有一条 nop 指令。图 4-68给出了指令成对在流水线中运行的情况。

| 指令类型 | 流水线阶段 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ALU 或分支 | IF | ID | EX | MEM | WB | | | |
| load 或 store | IF | ID | EX | MEM | WB | | | |
| ALU 或分支 | | IF | ID | EX | MEM | WB | | |
| load 或 store | | IF | ID | EX | MEM | WB | | |
| ALU 或分支 | | | ID | ID | EX | MEM | WB | |
| load 或 store | | | IF | ID | EX | MEM | WB | |
| ALU 或分支 | | | | IF | ID | EX | MEM | WB |
| load 或 store | | | | IF | ID | EX | MEM | WB |

图 4-68 静态双发射流水线。ALU 指令与数据传输指令同时发射。这里我们假设使用与单发射相同的五级流水线。尽管这并非严格的要求，但这样做确实会带来一些好处。特别是寄存器堆的写操作位于流水线的最后，可以简化异常处理和降低实现精确异常的难度，这些问题在多发射处理器中将变得更加难以处理

静态多发射处理器之间的不同在于处理潜在的数据冒险和控制冒险的方式。在有的设计中，编译器负责避免所有的冒险，它通过调度指令和插入 no-ops 等方法使得代码在执行时完全不需要冒险检测和硬件产生阻塞。在另外一些设计中，硬件检测数据冒险并在两个发射包间产生阻塞，而编译器只负责避免一个指令对中两条指令之间的依赖。尽管如此，冒险仍会使包含依赖指令的整个发射包阻塞。不管是软件必须处理所有的冒险还是只负责减少不同发射包之间的冒险，都会增加一次完成多个操作的长指令的情况。在这个例子中，我们假定使用第二种方法。

335

为了并行发射一个 ALU 操作和数据传输操作，首先需要增加一些硬件：除了通常的冒险检测和阻塞逻辑之外，还有寄存器堆的额外端口（见图 4-69）。在一个时钟周期内，我们需要为 ALU 操作读两个以上寄存器，为存储操作读两个以上寄存器，为 ALU 操作写一个端口，为装载操作写一个端口。因为 ALU 要用来进行 ALU 操作，所以需要一个额外的加法器来为数据传输计算有效地址。如果没有这些额外的硬件资源，我们的双发射流水线将不可避免地遭遇结构冒险。

显然，双发射处理器最多能将性能提高两倍。事实上，为了达到这一点，需要双发射流水线中重叠的指令数翻倍。额外的重叠使数据冒险和控制冒险带来的相对性能损失也增加了。例如，在我们简单的五级流水线中，装载指令有一个时钟周期的使用延迟（use latency），以防止一条指令无阻塞地使用其结果。在一个双发射五级流水线中，装载指令的结果不能在下个时钟周期使用。这意味着下两条指令不能无阻塞地使用装载的结果。而且，原本在简单的五级流水线中没有使用延迟的 ALU 指令现在有一个周期的使用延迟，因为其结果不能在与其配对的存储指令或装载指令中使用。为了有效地挖掘多发射处理器中潜在的并行性，需要使用更高级的编译器或硬件调度技术，其中静态多发射对编译器有更高的要求。

⊃ 使用延迟：在装载指令与可以无阻塞使用其结果的指令间相隔的时钟周期数。

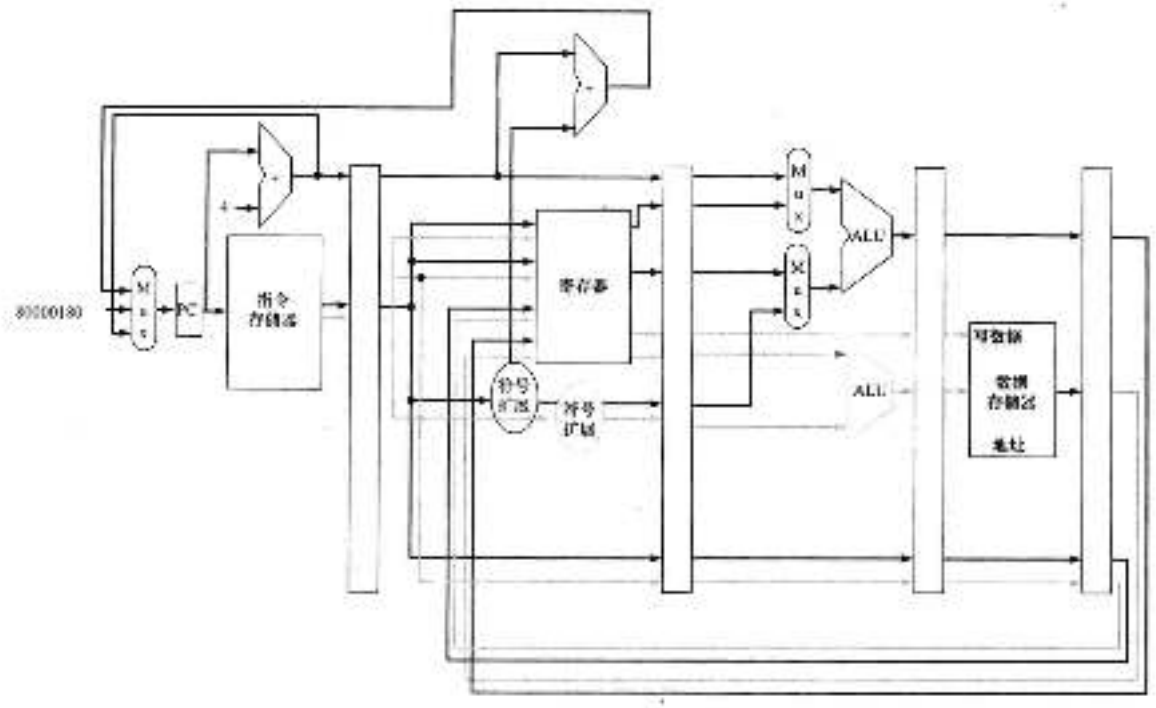



图 4-69 一个静态双发射的数据通路。双发射所需的额外硬件用灰色线显示，主要包括：来自指令存储器的额外 32 位输出，寄存器堆多出的两个读端口和一个写端口，还有一个额外的 ALU。这里假设下面那个 ALU 处理数据传输时的地址计算，而上面那个 ALU 处理所有的其他操作

### 例题：简单的多发射代码调度

在一个 MIPS 静态双发射流水线中，下面这个循环将如何调度？

```
Loop: lw   $t0, 0($s1)    # $t0=array element
      addu $t0,$t0,$s2# add scalar in $s2
      sw   $t0, 0($s1)# store result
      addi $s1,$s1, -4# decrement pointer
      bne  $s1,$zero,Loop# branch $s1!=0
```

对该指令序列进行重排序，以尽可能地避免流水线阻塞。假设分支是可预测的，即控制冒险由硬件处理。

### 答案

前三条指令间存在数据相关性，最后两条指令间也是如此。图 4-70 给出了这些指令的最佳调度方式。注意，只有一对指令同时使用了两个发射槽。每次循环需要花费 4 个时钟周期。在 4 个时钟周期内执行 5 条指令，与最好情况下 0.5 的 CPI 和 2.0 的 IPC 相比，CPI 只有 0.8 而 IPC 只有 1.25。注意，在计算 CPI 或 IPC 时，我们没有把执行的 nop 指令也算到有效的指令中去。如果算进去能提高 CPI，但并不能提高真实的性能。

| | ALU 或分支指令 | 数据传输指令 | 时钟周期 |
|---|---|---|---|
| Loop: | | lw $t0,0($s1) | 1 |
| | addi $s1,$s1,-4 | | 2 |
| | addu $t0,$t0,$s2 | | 3 |
| | bne $s1,$zero,Loop | sw $t0,4($s1) | 4 |

图 4-70 在双发射 MIPS 流水线中调度的代码。空白槽中是 nop 指令 □

有一种重要的从循环中获得更多性能的编译技术叫**循环展开**（loop unrolling）。循环展开时循环体会被复制多份。循环展开后，通过重叠不同循环体中的指令可以获得更高的指令级并行（ILP）。

⇨ 循环展开：一种从存取数组的循环中获取更多性能的技术，其中循环体会被复制多份并且不同循环体中的指令可能会调度到一起。

**例题** 多发射流水线中的循环展开

试着对上面的例子进行循环展开和调度，看其效果如何。为简单起见，假设循环起始地址与32位内存边界对齐。

**答案**

为了无延迟地调度循环，我们需要把循环体复制4份。在展开和消除不必要的循环开销指令后，将得到4个备份，每份包含lw指令、add指令和sw指令，还有addi指令和bne指令各一条。图4-71给出了展开并调度后的代码。

| | ALU或分支指令 | | 数据传输指令 | | 时钟周期 |
|---|---|---|---|---|---|
| Loop: | addi | $s1,$s1,-16 | lw | $t0, 0($s1) | 1 |
| | | | lw | $t1,12($s1) | 2 |
| | addu | $t0,$t0,$s2 | lw | $t2, 8($s1) | 3 |
| | addu | $t1,$t1,$s2 | lw | $t3, 4($s1) | 4 |
| | addu | $t2,$t2,$s2 | sw | $t0, 16($s1) | 5 |
| | addu | $t3,$t3,$s2 | sw | $t1,12($s1) | 6 |
| | | | sw | $t2, 8($s1) | 7 |
| | bne | $s1,$zero,Loop | sw | $t3, 4($s1) | 8 |

图4-71 对图4-70中的代码进行循环展开并在一个静态双发射MIPS流水线中调度后的代码。空白槽中是nop指令。因为循环中的第一条指令将$s1寄存器中的值减16，而装载指令的地址又是$s1寄存器中的原值，所以这个地址依次减4、减8、减12 □

在循环展开过程中，编译器引入了几个临时寄存器（$t1，$t2，$t3）。这个过程被称为**寄存器重命名**（register renaming），目的是消除一些虚假的数据依赖，这些虚假的数据依赖可能导致潜在的冒险或妨碍编译器灵活地调度代码。考虑一下如果只使用$t0展开的代码是什么样子的。在指令sw $t0,4($s1)后面会有多对lw $t0,0($s1)指令和addu $t0,$t0,$s2指令。如果不管$t0寄存器的使用，那么这些指令应该是完全无关的，即一个指令序列与下一个指令序列之间没有任何数据流动。这就是**反相关**（antidependence），也被称为**名字相关**（name dependence），即只是因为重用寄存器名引起的相关，而并非一个真实的数据相关。

⇨ 寄存器重命名：由编译器或硬件对寄存器进行重命名以消除反相关。

⇨ 反相关：也被称为名字相关，因为寄存器名的重用导致的相关，并非由两条指令中使用同一个值导致的真正相关。

338 在循环展开中的寄存器重命名促使编译器移动不相关的那个指令，从而得到更好的代码。重命名的过程消除了名字相关，同时保留了真正的相关。

注意，既然循环中14条指令中的12条以指令对的形式被执行，4次循环将花费8个时钟周期，即每次循环2个时钟周期，CPI为8/14=0.57。双发射加上循环展开与调度使得性能提高了将近两倍，这一方面是因为减少了循环控制指令，另一方面是因为双发射的缘故。这种性能提高的代价是使用了4个而非一个临时寄存器，同时代码长度也增长了很多。

### 4.10.3 动态多发射处理器

动态多发射处理器通常也称为超标量处理器，或简称**超标量**（superscalar）。在最简单的超标量处理器中，指令顺序发射，每个周期处理器决定是发射0条、1条，还是多条指令。显然，在这种处理器上要达到较好的性能仍然依赖编译器对指令的调度，通过错过依赖关系以达到较高的指令发射速率。尽管使用了编译器进行调度，这种简单的超标量处理器与VLIW处理器仍有显著不同。在超标量处理器中，不管代码是否经过调度，都是由硬件来保证执行的正确性。并且，编译得到的代码应当始终正确执行，而与指令发射速率和处理器的流水线结构无关。在某些VLIW的设计中情况并非如此，当把代码从一个处理器移到另一个处理器上运行时，可能需要重新编译。在其他一些静态发射处理器上，代码可以在不同的处理器上实现正确运行，但效果可能很差以至于不得不进行更加有效的编译。

许多超标量处理器扩展了基本的动态发射决策，将**动态流水线调度**（dynamic pipeline scheduling）也包含在内。动态流水线调度选择某个时钟周期内将执行的指令，约束条件是尽量不产生冒险和阻塞。让我们从一个简单的数据冒险的例子出发来进行说明。考虑下面的指令序列：

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

即使sub指令准备好执行，它也必须等待lw和addu指令先结束才行。如果内存很慢（第5章解释了有时访存操作会很慢的原因，即高速缓存缺失），sub指令可能会等待很多个时钟周期。动态流水线调度可以部分或者完全避免这种冒险。

- 超标量：一种高级流水线技术，可以使每个周期处理器能执行的指令数超过一条。
- 动态流水线调度：对指令进行重排序以避免阻塞的硬件支持。

#### 动态流水线调度

动态流水线调度选择下一条要执行的指令，可能的话会重排指令以避免阻塞。在这种处理器中，流水线被划分为3个主要单元：取指与发射单元、多个功能单元（在2013年的高端处理器中有一打或更多）和一个**提交单元**（commit unit）。图4-72描述了这个模型。第一个单元取指并译码，然后将每条指令发送到相应的功能单元执行。每个功能单元都有自己的缓冲区（称为**保留站**（reservation station）），用来保存操作数和操作（下一节我们将讨论许多最新处理器中使用的保留站的替代选择）。当缓冲区中包含了所有的操作数，并且功能单元就绪时，结果就被计算出来。结果得到后，被发送到等待该结果的保留站和提交单元。提交单元缓存这个结果，在确定安全时，再将这个结果写回寄存器堆或存储器（对存储指令）。提交单元中的缓冲区通常称为**重排序缓冲区**（reorder buffer），它也可以用来提供操作数，其工作方式类似于静态调度流水线中的旁路逻辑。一旦结果写回寄存器堆，其可以从寄存器堆中直接被取出，和一般的流水线完全一样。

[339]

- 提交单元：位于动态流水线和乱序流水线中的一个单元，用以决定何时可以安全地将操作结果送至程序员可见的寄存器和存储器。
- 保留站：功能单元的缓冲区，用来保存操作数和操作。
- 重排序缓冲区：动态调度处理器中用于暂时保存执行结果的缓冲区，等到安全时才将其中的结果写回寄存器或存储器。

将操作数缓存在保留站中并将结果放在重排序缓冲区中，实际上提供了一种寄存器重命名

机制，类似于前面循环展开例子中编译器所做的工作。为了在概念上分析其工作方式，考虑如
340 下几个步骤：

1）发射指令时，它先被复制到合适功能单元的保留站。如果它的操作数在寄存器堆中或重排序缓冲区中可用，那么操作数立即被复制到保留站中。除非所有的操作数和执行单元可用，否则指令一直缓存在保留站中。如果指令已经被发射，那么其操作数对应的寄存器堆副本不再需要，如果此时发生了对该寄存器的写请求，其值可以被覆盖。

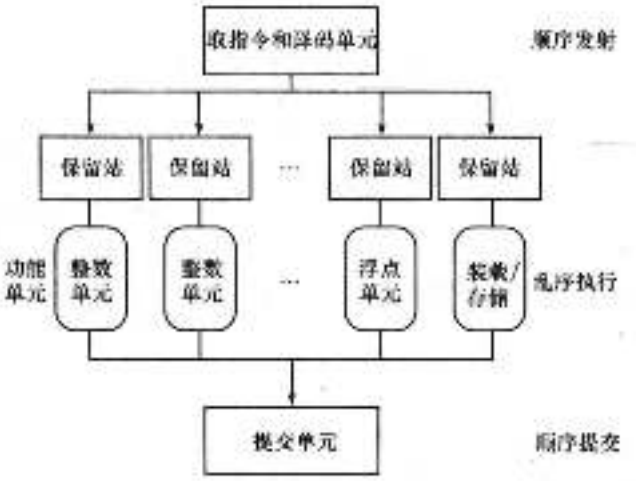



图 4-72 动态调度流水线中的三个主要单元。最后一个更新状态的步骤也被称为提交

2）如果操作数不在寄存器堆或重排序缓冲区中，那么它应该被某个功能单元以计算结果的形式输出。硬件将帮助定位产生这个结果的功能单元。当该单元计算出结果时，这个结果将直接从功能单元复制到保留站，而跳过寄存器堆。

上面这两步可以有效地利用重排序缓冲区和保留站以实现寄存器重命名。

从概念上讲，可以把动态调度流水线想象为对程序数据流结构的分析过程。处理器在不违背程序原有的数据流顺序的前提下以某种顺序执行各条指令。这种执行方式被称为**乱序执行**（out-of-order execution），因为执行指令的顺序可以与取指的顺序不同。

为了使程序表现得像是在一条简单的顺序流水线上执行，取指和译码单元必须能够顺序发射指令，以记录程序中的依赖关系。而提交单元也必须按照程序顺序将结果写回寄存器堆和存储器。这种保守的方案称为**顺序提交**（in-order commit）。所以当异常发生时，处理器可以找到最后执行的那条指令，而只有这条导致异常的指令之前的指令才能对寄存器状态进行修改。虽然处理器的前端（取指和发射）和后端（提交）按照顺序操作指令，各功能单元可以在获得所需数据的条件下随时开始执行过程。目前所有的动态调度流水线都采用顺序提交。

- 乱序执行：流水线执行的一种情况，即执行的指令被阻塞时不会导致后面的指令等待。
- 顺序提交：流水线执行的结果以取指顺序写回程序员可见寄存器的一种提交方式。

动态调度经常与基于硬件的推测机制相结合，特别是对分支指令的推测。通过对分支指令的方向进行推测，动态调度处理器可以在推测方向上进行取指和执行。由于指令是顺序提交的，我们可以在分支指令及所有推测执行的指令提交前知道推测是否正确。一个推测执行的动态调度流水线同样可以对装载指令的目的地址进行推测、对存取指令进行重排序和利用提交单元避免错误的推测。在下一节中我们将讨论 Intel Core i7 处理器的动态调度流水线设计与推测
341 机制。

**理解程序性能** 既然编译器可以根据数据依赖关系调度代码，你可能会问，为什么还需要超标量处理器来进行动态调度？这里面主要有三个原因。

第一，并不是所有的阻塞都是可以事先知道的。尤其是 cache 缺失（参见第 5 章）会导致不可预测的阻塞。动态调度使得处理器在一些指令阻塞时，可以调度其他指令继续执行，以掩盖阻塞。

第二，如果处理器采用动态分支预测推测分支的结果，那么由于这些信息依赖于预测和分支指令的真实执行情况，编译器无法得知指令的精确顺序。采用动态推测而不使用动态调度，会极大地限制可开发的指令级并行度（ILP）。

第三，由于流水线延迟和发射宽度等指令处理器的具体实现的不同有很大的差别，所以最佳的编译代码顺序也并不固定。例如，调度一个相互依赖的指令序列的具体方式与发射宽度和延迟存在着密切关系。流水线的结构同样会影响循环展开的尝试，才能避免可能的阻塞。它还会影响基于编译器的寄存器重命名的过程。动态调度使得硬件将这些细节隐藏起来。因此，用户和软件发行商就不用针对同一指令集的不同处理器发行相应的软件了。同样，以前的代码也能从更新的处理器上获得好处而不用重新编译。

**重点** 流水线和多发射都提高了指令的吞吐率并致力于开发指令级并行。然而，由于处理器有时必须等待依赖关系明确后才能继续工作，所以程序中的数据相关和控制相关往往限制了可达性能的上限。基于软件的指令级并行开发主要依赖于编译器来寻找依赖关系，并尽量减少这些依赖关系可能造成的不良后果。基于硬件的指令级并行开发主要依赖于流水线和多发射机制。推测执行可以由硬件或编译器完成，它可以增加指令行并行。但是使用时必须小心，因为错误的推测可能会降低性能。 342

**硬件/软件接口** 现代的高性能微处理器可以在一个时钟周期内发射多条指令。遗憾的是，持续这样的高发射速率是相当困难的。例如，尽管我们有一个单时钟周期可以发射 4～6 条指令的处理器，只有很少的应用程序能保持每周期发射两条以上指令。这里面主要有两个原因。

首先，由于使用了流水线，主要的性能瓶颈在于那些不能立即解决的相关性。这就限制了指令间的并行度，因此也就限制了发射速率。虽然对于真正的数据相关而言没有什么好的解决方法，但是一般情况下硬件或编译器对于相关是否确实存在都不知道，因而也就只能保守地假设相关存在了。例如，使用了指令的程序由于有更多的内存别名问题，往往存在隐式相关的可能性更大。反之，对于数组访问而言，由于有更大的规则性，使得编译器可以检测出没有相关存在的情况。同样，不能在编译期或运行期被准确预测的分支同样会限制指令级并行的开发。一般来说，指令级并行总是有开发的空间的，但是因为并行度较为分散（有时可能存在于上千条指令之间），编译器和硬件往往会显得力不从心。

其次，存储器层次（这是第 5 章的主题）中的缺失同样会使流水线难以满负荷运转。尽管一些访存引起的阻塞可以被掩盖掉，但是有限的指令级并行度同样会使阻塞被掩盖的程度有所下降。

## 4.10.4 能耗效率与高级流水线

通过动态多发射和推测执行开发指令级并行的负面问题是功耗效率。每项发明都成功地将更多的晶体管转化为性能，但是这种转化往往极其缺乏效率。因为功耗墙的原因，最新的处理器是单片多核式的，而非其前辈的深流水线或贪婪式推测。

尽管简单的处理器没有复杂的处理器那么快，但是在同样的能耗下却能得到更高的性能。所以当设计的约束更多来自能耗而非晶体管数量时，简单的处理器能在单芯片上获得更高的性能。

图4-73给出了一些处理器的流水线级数、发射宽度、推测级别、时钟频率、每芯片的核数和功耗等。注意从单核发展到多核时流水线级数和功耗的减少。

**精解** 提交单元负责寄存器堆和存储器的更新。一些动态调度处理器在执行过程中即时更新寄存器堆，而使用额外的寄存器来实现重命名功能并保存之前寄存器的复本直到更新该寄存器的指令不再是推测得出的。其他处理器通常把结果缓存在重排序缓冲器中，由提交单元在随后更新寄存器堆。在指令提交之前，写内存的数据必须先缓存在存储缓冲器（见第5章）或重排序缓冲器中。提交单元允许缓冲器在地址和数据有效时并且store操作
343 不依赖于预测的分支时写内存。

**精解** 非阻塞cache（nonblocking cache）在cache缺失（参见第5章）时能够继续提供cache访问服务。为了使指令在cache缺失时能继续执行，乱序执行处理器需要非阻塞cache的支持。

| 处理器 | 年份 | 时钟频率 | 流水线级数 | 发射宽度 | 乱序/推测 | 核数目/片 | 功耗 | |
|---|---|---|---|---|---|---|---|---|
| Intel 486 | 1989 | 25MHz | 5 | 1 | No | 1 | 5 | W |
| Intel Pentium | 1993 | 66MHz | 5 | 2 | No | 1 | 10 | W |
| Intel Pentium Pro | 1997 | 200MHz | 10 | 3 | Yes | 1 | 29 | W |
| Intel Pentium 4 Willamette | 2001 | 2 000MHz | 22 | 3 | Yes | 1 | 75 | W |
| Intel Pentium 4 Prescott | 2004 | 3 600MHz | 31 | 3 | Yes | 1 | 103 | W |
| Intel Core | 2006 | 2 930MHz | 14 | 4 | Yes | 2 | 75 | W |
| Intel Core i5 Nehalem | 2010 | 3 300MHz | 14 | 4 | Yes | 1 | 87 | W |
| Intel Core i5 Ivy Bridge | 2012 | 3 400MHz | 14 | 4 | Yes | 8 | 77 | W |

图4-73 Intel和微处理器的流水线复杂度、核数和功耗的发展情况。其中，Pentium 4的流水线级数没有包括提交级，如果加上提交级，Pentium 4的流水线级数会更深一些

**小测验**

说明下列开发指令级并行度的技术或单元主要是基于硬件还是基于软件。对某些项来说两者都有可能。

1. 分支预测
2. 多发射
3. 超长指令字（VLIW）
4. 超标量
5. 动态调度
6. 乱序执行
7. 推测机制
8. 重排序缓冲区
9. 寄存器重命名

## 4.11 实例：ARM Cortex-A8和Intel Core i7流水线

344 图4-74给出了本节将要考察的两个处理，它们的目标是后PC时代的标志性产品。

| 处理器 | ARM A8 | Intel Core i7 920 |
|---|---|---|
| 市场 | 个人移动设备 | 服务器，云计算 |
| 功耗 | 2W | 130W |
| 时钟频率 | 1GHz | 2.66GHz |
| 核/芯片 | 1 | 4 |
| 浮点 | 无 | 有 |
| 多发射 | 动态 | 动态 |
| 峰值指令数/周期 | 2 | 4 |
| 流水线级数 | 14 | 14 |
| 流水线调度 | 静态顺序 | 动态乱序猜测执行 |
| 分支预测 | 2级 | 2级 |
| 1级cache/核 | 32KiB指令，32KiB数据 | 32KiB指令，32KiB数据 |
| 2级cache/核 | 128～1 024KiB | 256KiB |
| 3级cache（共享） | | 2～8MiB |

图4-74 ARM Cortex-A8和Intel Core i7 920处理器的特点

## 4.11.1 ARM Cortex-A8

ARM Cortex-A8处理器主频为1GHz，具有14级流水线。它采用动态多发射技术，每个时钟周期可以发射两条指令，其流水线为静态顺序流水线，指令发射、执行和提交顺序执行。流水线包含取指令、指令译码和执行三个阶段。图4-75给出了流水线的整体情况。

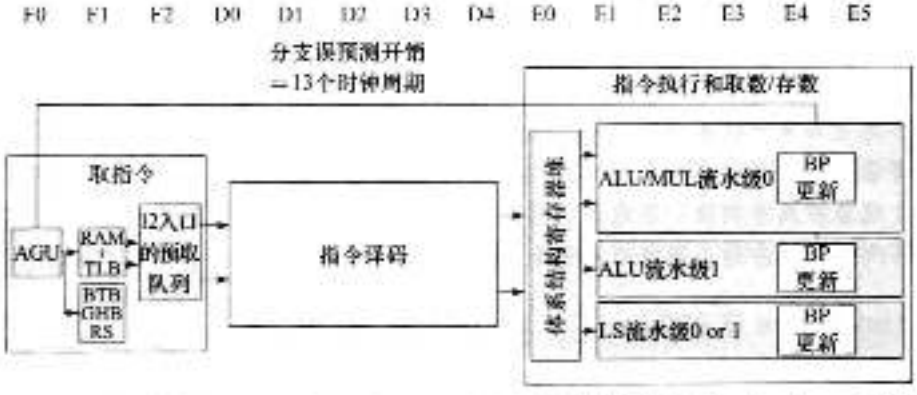



图4-75 A8的流水线。前端的三个流水级为12入口的指令预取缓存取入指令。地址产生单元（AGU）使用一个分支目标缓存（BTB）、全局历史缓存（GHB）和返回栈（RS）进行分支预测以保持预取队列满状态。指令译码有5个流水级，指令执行有6个流水级

前端的三个流水级每次取入两条指令，从而使得能够保存放12条指令的预取缓存维持满状态。它使用了一个两级的分支预测器，其中包含了一个512入口的分支目标缓存、4096入口的全局历史缓存和一个用于预测未来返回操作的8入口的返回栈。当分支预测错误时清空流水线，导致13个时钟周期的误预测开销。

译码阶段的5个流水级确定一对指令间是否存在可导致顺序执行的相关，并且确定将指令送往哪条执行流水线。

指令执行阶段的6个流水级提供了一条用于load和store指令的流水线、两条算术操作流水线，其中只有第一对可以处理乘法。指令对中的任何一条指令都可以发射到load-store流水线。三条流水线间在执行级具有全旁路机制。

图4-76给出了A8在使用从SPEC2000基准程序中衍生出来的一个小版本程序集的CPI。虽然理想的CPI是0.5，但是这里最好情形只有1.4，一般情形是2.0，而最差情形是5.2。在一
345 般情形下，80%的流水线阻塞是源于冒险，20%源于存储层次。流水线阻塞是由分支误预测、结构冒险和指令对间的数据相关造成的。由于A8是静态流水线，使用编译器来尽可能避免结构冒险和数据相关。

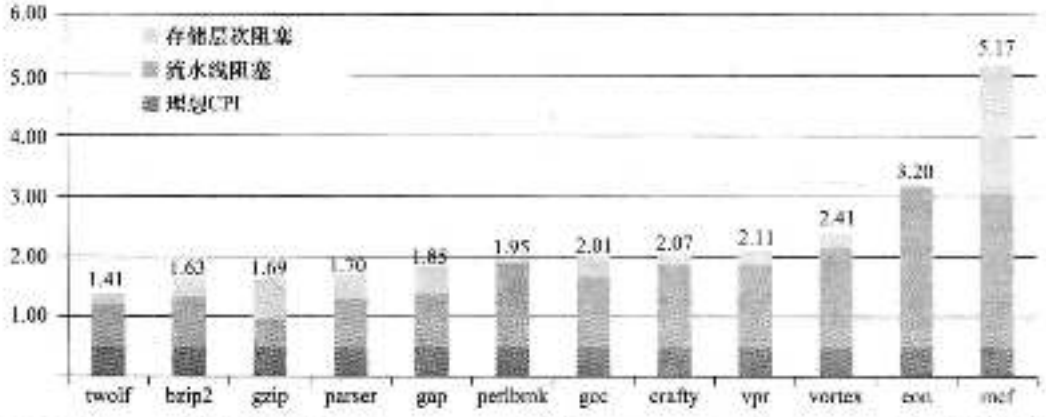



图4-76 Minnespec基准测试程序在ARM Cortex-A8上的CPI。Minnespec是SPEC2000的一个简化版本。这些基准程序使用更小的输入集使运行时间减少几个数量级。小的输入集极大地低估了存储层次对CPI的影响（见第5章）

**解释** Cortex-A8是一个支持ARMv7指令集的可配置内核。它以IP（Intellectual Property，知识产权）核方式交付使用。IP核是嵌入式、个人移动设备和相关市场中用于交付的主要形式：数十亿的ARM和MIPS处理器都是由这种IP核产生的。

注意IP核与Intel i7多核计算机中的核不同。一个IP核（自身可能就是多核）的设计目标是与其他逻辑集成在一起（因此是一个芯片的“核心”）形成一个对某种应用优化的处理器，这里其他逻辑包括专用处理器（例如视频编解码器）、I/O接口和存储器接口。虽然处理器核几乎相同，但最终的芯片可能有许多不同。一个参数就是L2 cache的容量，它在不同专用用户需要可以高达8倍。

## 4.11.2 Intel Core i7 920

x86采用了复杂的流水线技术，在其14级流水线中综合使用了动态多发射、乱序执行和推测执行的动态流水线调度技术。正如第2章中提到的，处理器依然面临着实现复杂x86指令集的挑战。Intel取入x86指令，将其翻译为类MIPS指令，Intel称之为“微操作”。微操作由复杂的基于推测执行的动态调度流水线执行，该流水线每个时钟周期最多可执行6个微操作。本
346 节集中讨论微操作流水线。

当我们考虑复杂的动态调度处理器的设计时，功能单元、cache和寄存器堆、指令发射和整个流水线控制的设计将混在一起，使得把数据通路和流水线分开变得很困难。因此，许多工程师和研究人员使用术语**微体系结构**（microarchitecture）来描述处理器内部体系结构的细节。

➲ 微体系结构：处理器的组织，包括主要的功能单元及它们的互连关系与流水线控制。

Intel Core i7使用重排序缓冲区和寄存器重命名技术来解决反相关和推测错误。寄存器重命名技术显式地将处理器中的**体系结构寄存器**（architectural register）（在64位版本的x86体系

结构中是 16 个）重命名为一组更大的物理寄存器集合。Core i7 使用寄存器重命名技术来消除反相关。寄存器重命名需要处理器维护体系结构寄存器和物理寄存器之间的映射关系，要能指出那个物理寄存器才是某个体系结构寄存器的最新备份。通过跟踪已经发生的重命名，寄存器重命名提供了另一种推测错误时的恢复方法：简单地撤销所有第一条推测错误指令后建立的所有映射。这会使处理器的状态返回到最后一条正确执行的指令处，并保持结构寄存器与物理寄存器之间的正确映射关系。

⊃ 体系结构寄存器：处理器中的可见寄存器。如在 MIPS 中，有 32 个整数寄存器和 16 个浮点寄存器是可见的。

图 4-77 给出了 Core i7 的整体组合和流水线结构。下面是一条 x86 指令的执行需要经历的 8 个阶段。

1）取指令——处理器使用一个多级分支目标缓冲器在速度和预测准确性方面做平衡。另外还有一个返回地址栈用于加速函数返回。误预测将导致 15 个周期的开销。取指部件使用预测地址从指令 cache 中取入 16 字节。

2）这 16 字节放入预译码指令缓冲器——预译码阶段将这 16 字节转换为独立的 x86 指令。 347
因为 x86 指令长度可以是 1～15 字节不等，预译码操作必须扫描多个字节以确定指令长度。所以预译码操作非常复杂。每条 x8 指令放入一个 18 入口的指令队列。

3）微操作译码——每条 x86 指令被翻译为微操作。有三个译码器将 x86 指令直接翻译为一个微操作。而对于具有复杂语法功能的 x86 指令，则使用一个微代码引擎产生一个微操作序列；它可以在每个周期生成 4 个微操作直到必需的微操作序列生成为止。这些微操作按照 x86 指令顺序放入 28 入口的微操作缓冲器。

4）微操作缓冲器执行循环流检测——如果有一个小的指令序列（少于 28 条指令或长度小于 256 字节）包含一个循环，循环流检测器将识别该循环，并直接从缓冲器中发射微操作，从 348
而减少了指令预取和指令译码。

5）执行基本指令发射——在将微操作发射到保留站之前，在寄存器表中查找寄存器位置、对寄存器进行重命名，分配重排序缓冲器入口、从寄存器或重排序缓冲中取结果。

6）i7 使用一个被 6 个功能单元共享的 36 入口的集中式保留站。在每个周期内最多可以向功能单元分派 6 个微操作。

7）各个功能单元执行微操作，执行结果不但送往寄存器提交部件，在已知指令将不再预测的情况下更新寄存器状态，还可送往任何一个等待的保留站。重排序缓冲中与指令对应的入口标记为完成。

8）当前面的一条或多条指令已经被标记为完成，则执行寄存器提交部件中未决的写操作，指令从重排序缓冲器中移走。

**精解** 第二步和第四步中的硬件能够将操作进行合并，从而减少需要执行的微操作的数量。第二步中的宏操作合并进行 x86 指令的合并，例如将比较后面紧跟一个分支合并成一个操作。第四步中的微操作合并将 load/ALU 操作和 ALU/store 之类的微操作对进行合并，并将它们发射到一个保留站中（在这里它们依旧可以独立发射），从而提高了缓冲器的利用率。在研究微操作合并和宏操作合并有为的 Intel 核体系结构时，Bird 等［2007］发现微操作合并对性能影响很小，而宏操作合并似乎对整数性能有适度的正面影响，对浮点性能影响很小。

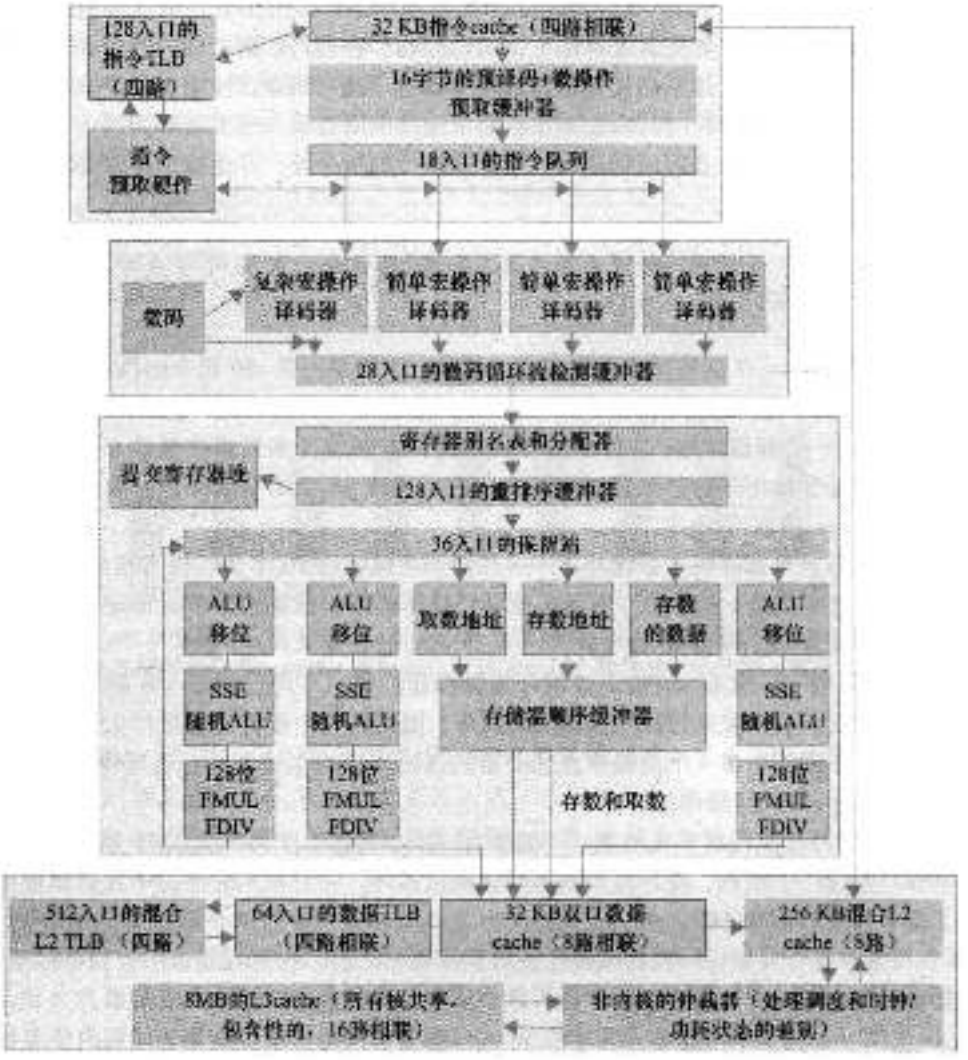



图4-77 包含存储部件的 Core i7 流水线结构。流水线总深度为14级，误预测的代价是17个时钟周期。该设计可以缓存48个load操作和32个store操作。6个相关部件在每个时钟周期可以开始执行一个RISC操作

### 4.11.3 Intel Core i7 920 的性能

图4-78给出了每个SPEC2006基准程序在Intel Core i7上的CPI。虽然理想的CPI是0.25，但是最佳情况是0.44，平均情况是0.79，而最差情况只有2.67。

在动态乱序执行流水线中，很难区分流水线阻塞和访存阻塞，但是我们却能有效地区分分支预测和推测执行。图4-79显示了分支误预测的比例和与所有的微操作分派相关的工作中未提交（即它们的结果被取消）的比例（使用分派进入流水线的微操作数量衡量）。误预测的最小值、平均值和最大值分别是0%、2%和10%。对于浪费的工作，分别是1%、18%和39%。

浪费的工作在有些情况下与误预测率相匹配，例如gobmk和astar。在许多情况下，例如
349 mcf，浪费的工作的比例相对要大于误预测比例。这种差别可能来源于存储器行为。只要在访存时保留站有足够的空间，mcf只有很高的数据cache缺失率，在误预测时刻分派许多条指令。当最

终确定推测执行许多指令中的一个分支是误预测时，与所有这些指令相关的微操作将被清除。

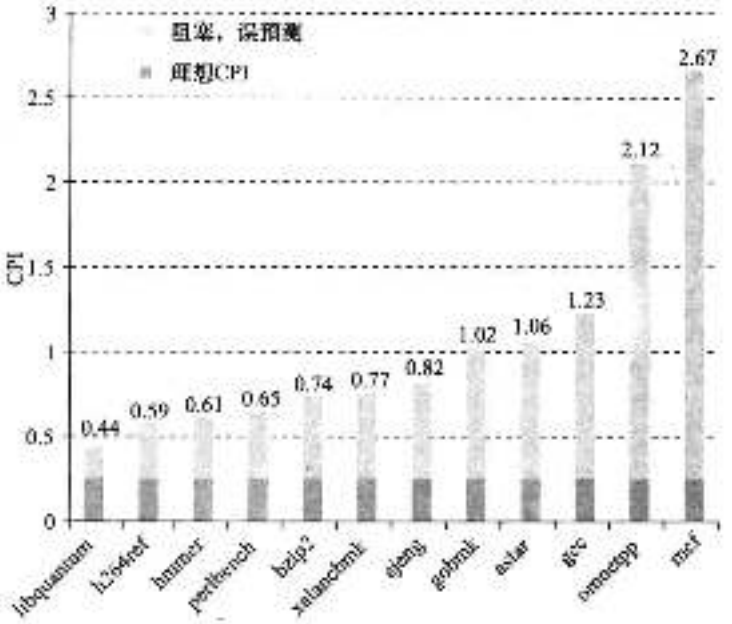



图 4-78 在 Intel Core i7 920 上运行 SPEC2006 整数基准程序的 CPI

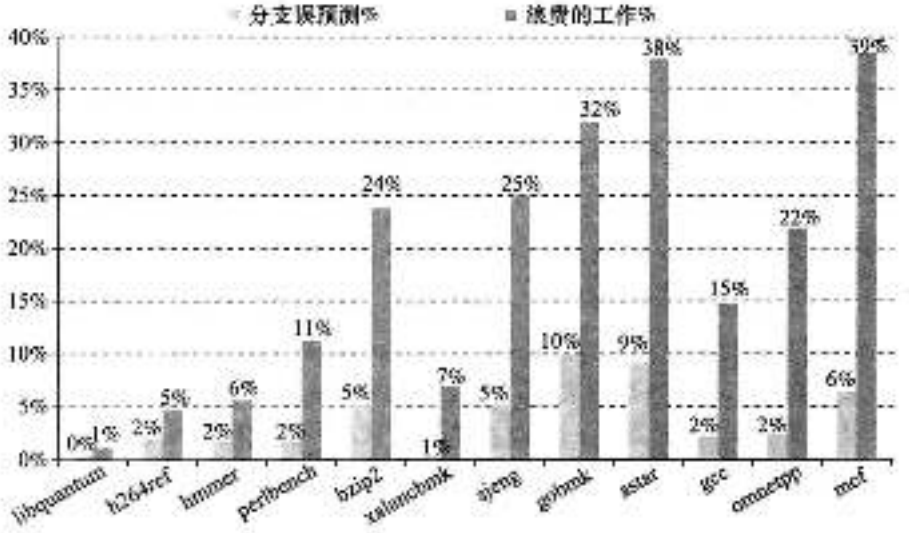



图 4-79 在 Intel Core i7 920 上运行 SPEC2006 整数基准程序时，分支误预测的比例和无效的推测所浪费的工作的比例

**理解程序性能** Intel Core i7 同时使用一个 14 级的流水线和激进的多发射来获取高性能。在保持背对背操作低延迟的同时，也消除了数据依赖的影响。对运行在这个处理器上的程序而言，最严重的潜在性能瓶颈在哪里？下面的列表包含了一些潜在的性能问题，最后三个问题在任何高性能流水线处理器中都会以某种形式出现。

- 使用了不能映射成几条简单微操作的 x86 指令。
- 难于预测的分支，会导致预测错误时的阻塞和推测失败时的重启。
- 长依赖——典型情况是执行时间很长的指令或存储器层次——这会导致阻塞。
- 存储器访问延迟增大（见第 5 章）将导致的处理器阻塞。

## 4.12 运行更快：指令级并行和矩阵乘法

回到第3章的DGEMM的例子，我们可以看到通过循环展开使得多发射乱序执行处理器有更多的指令用于调度，从而对指令级并行有影响。图4-80给出了图3-23中程序的循环展开版本，图3-23包含了使用C循环体的特点生成的AVX指令。

```
1  #include <x86intrin.h>
2  #define UNROLL (4)
3
4  void dgemm (int n, double* A, double* B, double* C)
5  {
6      for ( int i = 0; i < n; i+= UNROLL*4 )
7          for ( int j = 0; j < n; j++ ) {
8              __m256d c[4];
9              for ( int x = 0; x < UNROLL; x++ )
10                 c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12             for( int k = 0; k < n; k++ )
13             {
14                 __m256d b = _mm256_broadcast_sd(B+k+j*n);
15                 for (int x = 0; x < UNROLL; x++)
16                 c[x] = _mm256_add_pd(c[x],
17                     mm256_mul_pd( mm256_load_pd(A+n*k+x*4+i), b));
18             }
19
20             for ( int x = 0; x < UNROLL; x++ )
21                 _mm256_store_pd(C+i+x*4+j*n, c[x]);
22             }
23     }
```

图4-80 使用C循环体特点为x86生成AVX子字并行指令的优化的DGEMM C语言版本（图3-23）来使用循环展开开发指令级并行。图4-81给出了使用编译器产生的内循环的汇编语言，将3个for循环体并行展开以展示指令级并行

与前面的图4-71中的循环展开的例子一样，我们将循环展开4次（在C代码中我们使用常数UNROLL来控制我们希望展开的次数）。与图3-23中手工将C循环中每个循环体复制4份不同，我们可以依赖于gcc编译器的-O3优化选项来做展开。我们将每个循环体使用一个简单的for循环包起来形成4个迭代（第9、14和20行），将图3-23中的向量C0使用一个4元素数组c[ ]（第8、10、16和21行）替换。

图4-81给出了展开后的汇编语言代码。正如所期望的一样，图3-24中的每条AVX指令在图4-81中有4个版本，只有一个例外。因为在循环中，我们可以反复使用寄存器ymm0中的B元素的4个副本，所以我们只需要vbroadcastsd指令的一个副本。因此，图3-24中的5条AVX指令变成了图4-81中的17。另外，虽然常数和地址根据循环展开进行变化，7条整数指令在两种情况下没有变化。所以，即使循环展开了4次，循环体中的指令数目只是翻倍：由12条变为24条。

350
|
351

图4-82给出了使用DGEMM计算32×32的矩阵从未优化AVX到使用循环展开的AVX时性能的提升情况。可以看出，循环展开使得性能增加了一倍以上，由6.4GFLOPS增加到了14.6GFLOPS。相对于图3-21的未优化的DGEMM，子字并行和指令级并行共同作用获得了8.8倍的加速比。

```
1  vmovapd (%r11),%ymm4              # Load 4 elements of C into %ymm4
2  mov    %rbx,%rcx                  # register %rax = %rbx
3  xor    %ecx,%ecx                  # register %ecx = 0
4  vmovapd 0x20(%r11),%ymm3          # Load 4 elements of C into %ymm3
5  vmovapd 0x40(%r11),%ymm2          # Load 4 elements of C into %ymm2
6  vmovapd 0x60(%r11),%ymm1          # Load 4 elements of C into %ymm1
7  vbroadcastsd (%rcx,%r9,1),%ymm0    # Make 4 copies of B element
8  add    $0x8,%rcx                  # register %rcx = %rcx + 8
9  vmulpd (%rax),%ymm0,%ymm5         # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4          # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5     # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3          # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5     # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0     # Parallel mul %ymm1,4 A elements
15 add    %r8,%rax                   # register %rax = %rax + %r8
16 cmp    %r10,%rcx                  # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2          # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1          # Parallel add %ymm0, %ymm1
19 jne    68 <dgemm+0x68>            # jump if not %r8 != %rax
20 add    $0x1,%esi                  # register % esi = % esi + 1
21 vmovapd %ymm4,(%r11)              # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)          # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)          # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)          # Store %ymm1 into 4 C elements
```

图 4-81 对图 4-80 中循环展开后的 C 代码编译产生的嵌套循环体的 x86 汇编语言

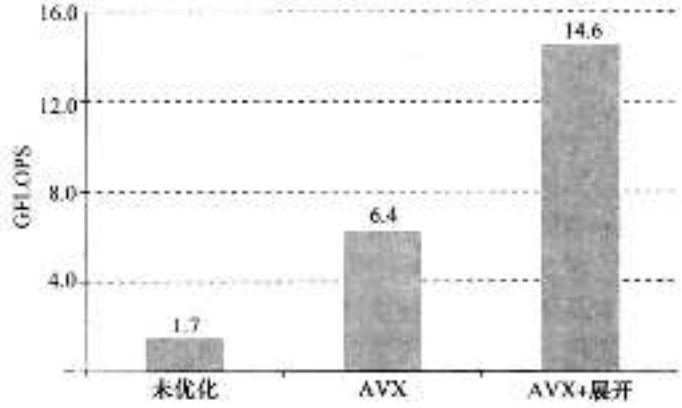



图 4-82 三个用于计算 32×32 矩阵的 DGEMM 版本的性能。相对于图 3-21 中未优化的代码，子字并行和指令级并行将性能提高了近 9 倍

**精解** 如 3.8 节中精解所提到的，这些结果是在 Turbo 模式关闭情况下获得的。如果将 Turbo 模式打开，与第 3 章相同，时钟频率提高了 3.3/2.6 = 1.27 倍，未优化的 DGEMM 性能将提升为 2.1GFLOPS，AVX 性能将提升为 8.1GFLOPS，循环展开后的 AVS 性能将提升为 18.6 GFLOPS。如 3.8 节所述，当一个六核芯片中只使用一个核时 Turbo 模式会很好的工作。 352

**精解** 虽然寄存器 symmb 在第 9～17 行被重用，但是由于 Intel Core i7 流水线对寄存器进行了重命名，因此没有流水线阻塞。

**小测验**

判断下列表述的正误。

1. Intel Core i7 使用多发射流水线直接执行 x86 指令。
2. A8 和 Core i7 都使用动态多发射。
3. Core i7 微体系结构中的寄存器比 x86 所要求的更多。
4. Intel Core i7 的流水线级数比早期 Intel Pentium 4 Prescott 的一半还少（见图 4-73）。

## 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

现代数字设计是用硬件描述语言和现代的计算机辅助综合工具完成的，其中综合工具能使用库和逻辑综合将描述转化为具体的硬件设计。关于这些语言和它们在数字设计中的使用有相关书籍说明。本节（在配套网站上）仅进行简单的介绍，并展示如何用一种硬件设计语言（Verilog）分别从行为级和可综合级描述 MIPS 控制。接着还提供了用 Verilog 描述的 MIPS 五级流水线行为级模型。最初的模型忽略了冒险，随后增加的部分着重于支持旁路、数据冒险和分支冒险所做的改变。

353 ~ 354

我们接着提供了大量使用单时钟周期图形化流水线表示的示意图，以帮助读者更好地理解执行一连串 MIPS 指令时流水线的工作细节。

## 4.14 谬误与陷阱

谬误：流水线是一种简单的结构。

本书证明了正确设计流水线必须非常谨慎。我们的另一本教程《Computer Architecture：A Quantitative Approach》的第 1 版尽管经过了上百人的校对，并且曾经在 18 个大学的课堂上使用过，它仍然有一个流水线方面的错误。直到有人根据该书设计处理器时才发现了这个错误。用 Verilog 来描述一个如 Intel Core i7 的流水线需要几千行代码，从中可以看出流水线的复杂性，因此设计流水线必须非常小心。

谬误：流水线概念的实现与工艺无关。

当芯片上晶体管的数量和速度决定五级流水线是最好的解决方案时，延迟分支（见 4.3 节的精解）是一种简单的控制冒险的方法。但对于长流水线、超标量执行和动态分支预测，延迟分支就成为多余的方法了。在 20 世纪 90 年代初期，动态流水线调度需要耗费大量资源并且无法得到很好的性能，但随着晶体管的预算持续加倍和逻辑电路变得比存储器更快，多个功能单元和动态流水线变得更加实用。当今，由于要考虑功耗问题，因此不能采用太激进的设计。

陷阱：没有考虑指令集的设计反过来会影响流水线。

许多流水线中遇到的困难都是由指令集的复杂性造成的，例如：

- 指令长度和指令运行时间变化太大会导致各流水级的不均衡，从而阻碍了某个流水级的运行，而且它们还会使冒险检测的实现复杂化。这个问题已经解决，最初在 20 世纪 80 年代后期的 DEC VAX 8500 中，采用了微操作和微流水线的方案，正如今天的 Intel Core i7 所采用的一样。当然，在微操作和实际指令间的转化和一致性维护上，开销依然是存在的。
- 复杂的寻址模式可能引起很多问题。更新寄存器的寻址模式会使冒险的检测复杂化。而需

要多次访问存储器的寻址模式会使流水线的控制复杂化，并且难以保持流水线平稳流动。

- 最好的例子大概是 DEC Alpha 和 DEC NVAX。通过比较可以看到，Alpha 的新指令集使得它的性能是 DEC NVAX 性能的两倍。另一个例子是，Bhandarkar 和 Clark [1991] 使用 SPEC 基准测试程序比较了 MIPS M/2000 和 DEC VAX 8700，他们得到了如下结论：尽管 MIPS M/2000 执行了更多的指令，但是 VAX 的平均时钟周期数是 MIPS 的 2.7 倍，所以总体上 MIPS 更快一些。 355

## 4.15 本章小结

智慧十足八九体现在恰当的时机。

——美国谚语

在这一章我们看到，处理器的数据通路和控制通路的设计，可以从指令集系统和对工艺基本特性的理解开始。在 4.3 节，我们看到了在指令集体系结构确定和决定使用单周期实现的基础上，如何构造 MIPS 处理器的数据通路。当然，背后的工艺也影响许多设计决策，如数据通路中哪些部件可用，以及单周期实现是否有意义等。

流水线提高了吞吐率，但不能提高指令的内在执行时间（指令延迟（instruction latency））；对某些指令而言，指令延迟与单周期实现的延迟类似。多发射增加了额外的允许每个时钟周期发射多条指令的数据通路硬件，但是却增加了有效延迟。为了减少简单的单周期实现数据通路的时钟周期，提出了流水线技术。相比之下，多发射关注于减少每条指令的时钟周期数（CPI）。

⇨ 指令延迟：执行一条指令所真正花费的时间。

流水线和多发射都试着开发指令级并行。开发更高指令级并行的主要限制因素是存在数据相关和控制相关。在软硬件上都使用预测来调度和推测，是降低相关带来影响的主要手段。

我们展示了将 DGEMM 的循环展开 4 次来开发指令级并行，利用 Core i7 的乱序执行机制可使性能提升一倍以上。

20 世纪 90 年代中期我们开始使用更长的流水线、多发射和动态调度，这些技术帮助我们维持了从 20 世纪 80 年代早期以来每年 60% 的处理器性能增长速度。正如第 1 章中所提到的，这些微处理器依旧使用顺序执行程序模型，但是它们最终会遇到功耗墙。因此，工业界被迫转向在更粗粒度上开发并行性的多处理器（这是第 6 章的主题）。这种趋势也迫使设计者们对 20 世纪 90 年代中期一些发明的功耗－性能含义重新进行评价，其结果是在最新的微体系结构中使用了更简单而不是复杂的流水线。

为了维持通过并行处理器带来的计算性能提高，Amdahl 定律预言了系统中的其他部件会成为瓶颈。这个瓶颈就是下一章要讨论的主题——存储器层次。 356

## 4.16 历史观点和拓展阅读

这一部分放在配套网站中，讨论了第一个流水线处理器、最早的超标量处理器、乱序执行与推测执行技术的发展以及同时期编译器技术的发展。

## 4.17 练习题

**4.1** 考虑如下指令：

指令：AND Rd,Rs,Rt

解释：Reg[Rd] = Reg[Rs] AND Reg[Rt]

4.1.1 [5] <4.1> 对上述指令而言，图4-2中的控制单元将产生哪些控制信号？

4.1.2 [5] <4.1> 对上述指令而言，将用到哪些功能单元？

4.1.3 [10] <4.1> 哪些功能单元会产生输出，但输出不会被以上指令用到？对以上指令而言，哪些功能单元不产生任何输出？

4.2 图4-2中基本的单周期MIPS实现仅能实现某些指令。可以在这个指令集中加入新的指令，但决定是否加入取决于给处理器的数据通路和控制通路增加的成本和复杂度。对下面的新指令而言，请回答下列3个问题：

指令：LWI Rt,Rd(Rs)

解释：Reg[Rt] = Mem[Reg[Rd] + Reg[Rs]]

4.2.1 [10] <4.1> 对上述指令而言，哪些已有的单元还可以被使用？

4.2.2 [10] <4.1> 对上述指令而言，还需要增加哪些功能单元？

357 4.2.3 [10] <4.1> 为了支持这些指令，需要在控制单元增加哪些信号？

4.3 当处理器设计者考虑改进处理器数据通路时，往往要考虑性能与成本的折中。假设我们从图4-2的数据通路出发，其中指令存储器、加法器、多选器、ALU、寄存器堆、数据存储器和控制单元的延迟分别为400ps、100ps、30ps、120ps、200ps、350ps和100ps，相应的成本分别为1000、30、10、100、200、2000和500。

考虑给ALU增加一个乘法，这将使ALU的延时增加300ps，同时ALU的成本增加600。这样做的结果是需要执行的指令减少了5%，主要是由于不再需要模拟MUL指令。

4.3.1 [10] <4.1> 改进前后的时钟周期分别是多少？

4.3.2 [10] <4.1> 改进后将获得多大的加速比？

4.3.3 [10] <4.1> 比较改进前后的性价比。

4.4 本练习题中的题目假定在实现一个处理器的数据通路时，逻辑模块的延时如下：

| I-Mem | Add | Mux | ALU | Regs | D-Mem | Sign-Extend | Shift-Left-2 |
|---|---|---|---|---|---|---|---|
| 200ps | 70ps | 20ps | 90ps | 90ps | 250ps | 15ps | 10ps |

4.4.1 [10] <4.3> 如果处理器只需做连续取指这一件事（见图4-6），那么时钟周期是多少？

4.4.2 [10] <4.3> 考虑一个与图4-11类似的数据通路，但是假设处理器只需处理无条件相对跳转指令，那么时钟周期是多少？

4.4.3 [10] <4.3> 重做练习题4.4.2，但这次假设只需处理有条件相对跳转指令。

本练习题中剩下的三个问题是关于Shift-Left-2数据通路单元的：

4.4.4 [10] <4.3> 哪些类型的指令需要该单元？

4.4.5 [20] <4.3> 对哪些类型的指令而言，该单元位于关键路径上？

4.4.6 [10] <4.3> 假设仅需支持beq指令和add指令，讨论该单元的延迟变化对处理器时钟周期的影响。假定其他单元的延迟不变。

358

4.5 本练习题的问题中，假定没有流水线阻塞。各种类型所占的比例如下：

| add | addi | not | beq | lw | sw |
|---|---|---|---|---|---|
| 20% | 20% | 0% | 25% | 25% | 10% |

4.5.1 [10] <4.3> 数据存储器平均用了多少时钟周期？

4.5.2 [10] <4.3> 符号扩展电路的输入平均用了多少时钟周期？在未用到该输入的其他时间，符号扩展电路在做什么？

4.6 在制造硅芯片时，材料（例如，硅）的缺陷和制造错误会导致电路失效。一个非常普遍的问题是一根线上的信号会对相邻线上的信号产生影响，这被称为串扰。有一类串扰问题是这样的，某些线上的信号为高值（如电源线），该线附近的线也被固定为0（stuck-at-0）或1（stuck-at-1）。下

面的问题中的缺陷发生在图 4-24 中寄存器堆的写寄存器输入端的第 0 位。

**4.6.1** [10] <4.3, 4.4> 假设这样测试处理器的缺陷：先给 PC、寄存器堆、数据和指令存储器中设置一些值（可以自己选择），执行一条指令，然后读出 PC、寄存器堆和存储器中的值；最后检查这些值以判断处理器中是否存在缺陷。你能设计一个方案检查该信号上是否有固定为 0 缺陷吗？

**4.6.2** [10] <4.3, 4.4> 重做练习题 4.6.1，这次检查固定为 1 缺陷。你能只设计一个测试方案同时检查固定为 0 缺陷和固定为 1 缺陷吗？如果可以，请解释如何实现；如果不能，请说明理由。

**4.6.3** [60] <4.3, 4.4> 如果我们知道一个处理器在该信号上有一个固定为 1 缺陷，它还能用吗？为了使这个处理器仍然可用，我们必须将原来能在正常 MIPS 处理器上运行的程序作一些变换，使之可以在这个处理器上运行。假设指令存储器和数据存储器都很大，足够容纳变换后的程序。提示：将因为该缺陷不能用的指令替换为一系列能用的指令，这一系列指令与原指令功能相同。

**4.6.4** [10] <4.3, 4.4> 重做练习题 4.6.1，这次检测控制信号 MemRead 是否存在该缺陷？如果 RegDst 控制信号为 0 时，MemRead 控制信号为 0，则有缺陷，否则无缺陷。

**4.6.5** [10] <4.3, 4.4> 重做练习题 4.6.4，这次检测控制信号 Jump 是否存在该缺陷？如果 RegDst 控制信号为 0 时 Jump 控制信号为 0，则有缺陷，否则无缺陷。

359

**4.7** 本练习题将讨论单周期数据通路中指令的执行细节。本练习题中的问题基于单周期处理器取入如下的指令字后的一个时钟周期：

```
10101100011000100000000000010100
```

假定数据存储器是全 0 且处理器的寄存器在取入以上指令字的时钟周期的开始时的内容如下：

| r0 | r1 | r2 | r3 | r4 | r5 | r6 | r8 | r12 | r31 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | -1 | 2 | -3 | -4 | 10 | 6 | 8 | 2 | -16 |

**4.7.1** [5] <4.4> 对该指令字而言，符号扩展单元和左移两位单元（图 4-24 的左上角）的输出是什么？

**4.7.2** [10] <4.4> 对该指令字而言，ALU 控制单元的输入是什么？

**4.7.3** [10] <4.4> 该指令执行后的新 PC 值是什么？在图 4-24 中灰线决定该新 PC 值的路径。

**4.7.4** [10] <4.4> 对给定的指令字和寄存器堆初值，给出每个多选器数据输出的值。

**4.7.5** [10] <4.4> 给出 ALU 和两个加法器数据输入的值。

**4.7.6** [10] <4.4> 给出寄存器堆所有输入信号的值。

**4.8** 本练习题讨论流水线对处理器时钟周期的影响。下面给出了数据通路中不同阶段的延迟情况：

| IF | ID | EX | MEM | WB |
|---|---|---|---|---|
| 250ps | 350ps | 150ps | 300ps | 200ps |

另外，假定处理器执行各种指令的比率如下面所示：

| alu | beq | lw | sw |
|---|---|---|---|
| 45% | 20% | 20% | 15% |

**4.8.1** [5] <4.5> 流水线处理器与非流水线处理器的时钟周期分别是多少？

**4.8.2** [10] <4.5> lw 指令在流水线处理器和非流水线处理器中的总延迟分别是多少？

360

**4.8.3** [10] <4.5> 如果可以将原流水线数据通路的一级划分为两级，每级的延迟是原级的一半，那么你会选择哪一级进行划分？划分后处理器的时钟周期为多少？

**4.8.4** [10] <4.5> 假设没有阻塞和冒险，数据存储器的利用率是多少（占总周期数的百分比）？

**4.8.5** [10] <4.5> 假设没有阻塞和冒险，寄存器堆的写寄存器端口的利用率是多少？

**4.8.6** [30] <4.5> 假设一种多周期的处理器设计，其中每条指令需要多个时钟周期完成，但上一条指令完成前不取下一条指令。在这种设计中，指令仅经过其所需的阶段（例如，存储指令仅需 4 个时钟周期，因为其不需要 WB 阶段）。比较单周期设计、多周期设计和流水线设计三者的时钟周期和总执行时间。

**4.9** 本练习题讨论数据相关如何影响4.5节中基本五级流水线的运行。结合下面的指令序列完成问题：

```
or r1,r2,r3
or r2,r1,r4
or r1,r1,r2
```

另外，假定每种相关旁路的周期时间如下：

| 无旁路 | 全旁路 | ALU-ALU 旁路 |
|---|---|---|
| 250ps | 300ps | 290ps |

**4.9.1** [10] <4.5> 指出指令序列中存在的相关及其类型。

**4.9.2** [10] <4.5> 假设该流水线处理器没有旁路，指出指令序列中存在的冒险并加入 nop 指令以消除冒险。

**4.9.3** [10] <4.5> 假设该流水线处理器中有充分的旁路。指出指令序列中存在的冒险并加入 nop 指令以消除冒险。

**4.9.4** [10] <4.5> 该指令序列在无旁路和有充分的旁路时，总执行时间分别是多少？后者相对于前者的加速比是多少？

**4.9.5** [10] <4.5> 如果仅有 ALU 至 ALU 的旁路（没有从 MEM 到 EX 的旁路），如何加入 nop 指令以消除可能的冒险？

**4.9.6** [10] <4.5> 该指令序列在仅有 ALU 至 ALU 的旁路时，总执行时间分别是多少？与无旁路的情况相比，加速比是多少？

361

**4.10** 本练习题将考查资源冒险、控制冒险和指令集体系结构设计如何影响流水线的执行。本练习题中的问题涉及下面的 MIPS 代码片段

```
sw  r16,12(r6)
lw  r16,8(r6)
beq r5,r4,Label # Assume r5!=r4
add r5,r1,r4
slt r5,r15,r4
```

假定每个流水级的延迟如下：

| IF | ID | EX | MEM | WB |
|---|---|---|---|---|
| 200ps | 120ps | 150ps | 190ps | 100ps |

**4.10.1** [10] <4.5> 假设所有的分支都被正确预测（控制冒险完全被消除）且没有使用延迟时间槽，并且只有一个存储器（既存储指令又存储数据）。如果一个时钟周期内同时取指和取数据就会发生结构冒险。为保证前进，该冒险必须始终以有利于取数指令的方式解决。该指令序列在仅有一个存储器的五级流水线中执行的总时间是多少？我们知道插入 nop 指令可以消除数据冒险，可以用同样的方法消除这里的结构冒险吗？为什么？

**4.10.2** [20] <4.5> 假设所有的分支都被正确预测（控制冒险完全被消除）且没有使用延迟时间槽。如果我们改变存取指令的格式，仅使用寄存器（不含偏移地址）进行寻址，那么这些指令不再需要使用 ALU。结果是 MEM 级和 EX 级可以重叠成一级，整个流水线也就成为四级。改变该代码以适应上述 ISA 的改变。假设这个改变不影响时钟周期，对该指令序列而言，这个改变造成的加速比是多少？

**4.10.3** [10] <4.5> 假设在分支时进行阻塞且没有使用延迟时间槽，那么在 ID 级确定分支方向相对于在 EX 级确定分支方向的加速比是多少？

**4.10.4** [10] <4.5> 在给定的流水级延迟下，重做练习题 4.10.2，考虑可能的时钟周期变化。如果 EX 级和 MEM 级重叠起来，它们的大部分工作可以并行执行。这样重叠后的 EX/MEM 级的延迟是原来两级的较大者，不能并行工作时延迟还要再加 20ps。

**4.10.5** [10] <4.5> 在给定的流水线延迟下，重做练习题4.10.3，考虑可能的时钟周期变化。假设分支方向判断从EX级移到ID级时，ID级的延迟增加50%，而EX级的延迟减少10ps。 362

**4.10.6** [10] <4.5> 假设在分支时进行阻塞且没有使用延迟时间槽，如果beq指令的地址计算移到MEM级，时钟周期将变为多少？该指令序列的总执行时间将变为多少？加速比是多少？假设分支方向判断从EX级移到MEM级时，EX级的延迟减少20ps，而MEM级的延迟不变。

**4.11** 考虑下面的循化：

```
loop:lw  r1,0(r1)
     and r1,r1,r2
     lw  r1,0(r1)
     lw  r1,0(r1)
     beq r1,r0,loop
```

假定使用了完美的分支预测（没有控制冒险导致的阻塞），没有延迟槽，流水线支持完全旁路。另外假定循环结束前该循环迭代了很多次。

**4.11.1** [10] <4.6> 画出循环第三次执行的流水线图，从取出循环的首条指令开始至取出下次循环的首条指令（不包括该次取指）结束。给出这段时间内流水线中的所有指令（不仅仅是第三次循环中的那些指令）。

**4.11.2** [10] <4.6> 在这段时间内有百分之多少五级流水线都在做有用的工作？

**4.12** 本练习题讨论流水线处理器中旁路的成本/复杂度/性能折中。参考图4-45的流水线数据通路，假设指令中有部分存在RAW（read after write，写后读）数据相关。RAW数据相关根据生成结果的流水级（EX或MEM）和使用结果的流水级（1st意味着生成结果后的第一条指令，2nd意味着生成结果后的第二条指令）确认。假设在时钟周期的前半部分写寄存器，在后半部分读寄存器，这样“EX to 3rd”和“MEM to 3rd”相关不会产生数据冒险。最后假设无数据冒险时处理器的CPI为1。

| 仅EX to 1st | 仅MEM to 1st | 仅EX to 2nd | 仅MEM to 2nd | EX to 1st和MEM to 2nd | 其他RAW相关 |
|---|---|---|---|---|---|
| 5% | 20% | 5% | 10% | 10% | 10% |

363

假定各级流水线延迟如下。其中EX级给出了不同旁路情况下的延迟。

| IF | ID | EX（无旁路） | EX（全旁路） | EX（仅有EX/MEM的旁路） | EX（仅有MEM/WB的旁路） | MEM | WB |
|---|---|---|---|---|---|---|---|
| 150ps | 100ps | 120ps | 150ps | 140ps | 130ps | 120ps | 100ps |

**4.12.1** [10] <4.7> 如果不使用旁路，会有百分之多少的时钟周期因为数据冒险阻塞？

**4.12.2** [5] <4.7> 如果使用完全的旁路（旁路所有可以旁路的结果），会有百分之多少的时钟周期因为数据冒险阻塞？

**4.12.3** [10] <4.7> 假设不能提供二输入多选器（对完全的旁路是必需的），我们必须确定从EX/MEM流水线寄存器旁路（旁路下一个周期）还是从MEM/WB流水线寄存器旁路（旁路2个周期）更好？哪种方法会产生更少的数据阻塞？

**4.12.4** [10] <4.7> 对给定的冒险概念和流水级延迟，完全的旁路相对于无旁路的加速比是多少？

**4.12.5** [10] <4.7> 如果加入能消除所有数据冒险的时间旅行旁路，其相对于具有旁路机制的处理器的加速比是多少？假设在EX级完全旁路的基础上加入这个还没发明的时间旅行旁路的代价是增加100ps的延迟。

**4.12.6** [20] <4.7> 重做练习题4.12.3，这次问哪种方法会产生更小的CPI。

**4.13** 本练习题讨论旁路，冒险检测和指令集设计之间的关系。根据下面的指令序列回答下列问题。假设其在一个五级流水线数据通路上执行。

```
add r5,r2,r1
lw  r3,4(r5)
lw  r2,0(r2)
or  r3,r5,r3
sw  r2,0(r5)
```

364

4.13.1 [5] <4.7> 如果没有旁路或冒险检测电路，请插入 nop 指令以保证正确执行。

4.13.2 [10] <4.7> 重做练习题 4.13.1，这次仅当通过改变或重排非空指令都不能避免冒险时才插入 nop 指令。假设可以使用寄存器 R7 作为临时寄存器。

4.13.3 [10] <4.7> 如果处理器中存在旁路，但忘了实现冒险检测单元（以为实现了），代码执行时会发生什么情况？

4.13.4 [20] <4.7> 如果存在旁路，在执行指令序列的前 5 个时钟周期，通过图 4-60 中的冒险检测和旁路单元指出每个周期中哪些信号是有效的。

4.13.5 [10] <4.7> 如果没有旁路，对图 4-60 中的冒险检测单元来说还需要哪些新的输入输出信号？以该指令序列为例，说明为什么需要这些信号。

4.13.6 [20] <4.7> 对练习题 4.13.5 中新的冒险检测单元来说，给出执行时前 5 个时钟周期中每个周期的输出信号（使有效）。

4.14 本练习题讨论流水线处理器的延迟时间槽、控制冒险和分支执行之间的关系。假设下列 MIPS 代码在一个五级流水线、有完全旁路和预测分支总发生的处理器上运行。试根据下面的两个序列回答下列问题。

```
        lw r2,0(r1)
label1: beq r2,r0,label2 # not taken once, then taken
        lw r3,0(r2)
        beq r3,r0,label1 # taken
        add r1,r3,r1
label2: sw r1,0(r2)
```

4.14.1 [10] <4.8> 画出该指令序列的流水线执行图，假设没有延迟时间槽而且在 EX 级执行分支。

4.14.2 [10] <4.8> 重做练习题 4.14.1，但是假设使用了延迟时间槽。给定的代码中，跟在分支之后的指令是该分支的延迟槽指令。

4.14.3 [20] <4.8> 不同于需要一个 ALU 操作、另一种显而确定分支方向的方法是使用“bez rd,label”和“bnez rd,label”之类的条件分支指令。该类条件分支指令根据寄存器值是否为零决定是否分支。变换原指令序列以使用该类条件分支指令而非 beq 指令。假设寄存器 $8 [⊖]是临

365 时寄存器，并且可以使用 R 型指令 seq（set if equal，相等置位）。

4.8 节说明了如何把分支执行提前到 ID 级以减少控制冒险。这个方法需要在 ID 级增加一个专用的比较器，如图 4-62 所示。但是这个方法增加了 ID 级的延迟，并且需要额外的旁路逻辑和冒险检测。

4.14.4 [10] <4.8> 以指令序列中第一条分支指令为例，说明图 4-62 中为支持在 ID 级执行分支而加入的冒险检测逻辑。该逻辑需要检测什么类型的冒险？

4.14.5 [10] <4.8> 对给定的指令序列，把执行分支移到 ID 级带来的加速比是多少？为什么？假设 ID 级进行的额外比较不影响时钟周期。

4.14.6 [10] <4.8> 以指令序列中第一条分支指令为例，说明为支持在 ID 级执行分支而加入的旁路。比较新旁路单元与图 4-62 中旁路单元的复杂度。

4.15 一个好的分支预测器有多重要取决于条件分支指令的频率，它与分支预测器的精度共同决定误预测分支导致的开销时间长短。在本练习题中，假设指令的动态执行频度如下：

| R 型 | beq | jmp | lw | sw |
|---|---|---|---|---|
| 40% | 25% | 5% | 25% | 5% |

⊖ 原书为 R8，有误。——译者注

假定分支预测器的精度如下：

| 分支总发生 | 分支总不发生 | 2 位预测器 |
|---|---|---|
| 45% | 55% | 85% |

**4.15.1** [10] <4.8> 误预测分支导致的阻塞将增加 CPI。对分支总发生预测器而言，误预测分支将导致 CPI 增加多少？假设分支方向在 EX 级确定，没有数据冒险且不使用延迟时间槽。

**4.15.2** [10] <4.8> 重做练习题 4.15.1，这次改为分支总不发生预测器。

**4.15.3** [10] <4.8> 重做练习题 4.15.1，这次改为 2 位分支预测器。

**4.15.4** [10] <4.8> 对 2 位分支预测器而言，将一半分支指令用 ALU 指令替代（一条 ALU 指令替代一条分支指令）将获得的加速比是多少？假设被正确预测的分支指令和被不正确预测的分支指令被取代的概率相同。

366

**4.15.5** [10] <4.8> 对 2 位分支预测器而言，将一半分支指令用 ALU 指令替代（两条 ALU 指令替代一条分支指令）将获得的加速比是多少？假设被正确预测的分支指令和被不正确预测的分支指令被取代的概率相同。

**4.15.6** [10] <4.8> 有些分支是非常容易预测的。假设 80% 的分支指令都是非常容易预测的循环返回分支，那么 2 位分支预测器对剩下的 20% 分支指令的预测精度是多少？

**4.16** 本练习题讨论不同分支预测器对给定分支模式（如循环）的预测精度。给定的分支模式为：T，NT，T，T，NT。

**4.16.1** [5] <4.8> 对该分支模式，分支总发生预测器与分支总不发生预测器的准确率分别是多少？

**4.16.2** [5] <4.8> 对该分支模式的前 4 个分支而言，2 位分支预测器的准确率是多少？假设预测器的初始状态与图 4-63 左下角状态相同（预测未发生）。

**4.16.3** [10] <4.8> 如果该分支模式一直重复下去，2 位分支预测器的准确率是多少？

**4.16.4** [30] <4.8> 如果该分支模式一直重复下去，设计一个能取得最高准确率的预测器。这个预测器必须是一个时序电路，有一个输出表示预测结果（1 表示发生，0 表示未发生），除了时钟和指示当前指令是条件分支指令的信号外没有其他输入。

**4.16.5** [10] <4.8> 如果有一个分支模式与该分支模式完全相反且一直重复下去，那么在练习题 4.16.4 中你设计的预测器对这个分支的准确率是多少？

**4.16.6** [20] <4.8> 重做练习题 4.16.4，这次你的预测器最终（可能需要一个热身过程）可以同时完美地预测该分支模式及完全相反的分支模式（假设分支模式一直重复下去）。这个预测器应该有一个输入告诉它真实的分支结果。提示：这个输入可以帮助预测器判断是两个分支模式中的哪一个。

**4.17** 本练习题讨论异常处理对流水线设计的影响。根据下面两条指令回答前三个问题：

| 指令 1 | 指令 2 |
|---|---|
| BNE R1，R2，Label | LW R1，0(R1) |

**4.17.1** [5] <4.9> 每条指令分别可能产生什么异常？对每个可能产生的异常，指出其将在哪个流水线级被检测到。

367

**4.17.2** [10] <4.9> 如果每个异常都有独立的处理程序地址，流水线应该怎样设计才能处理异常？假设设计处理器时已知每个异常处理程序的地址。

**4.17.3** [10] <4.9> 如果第二条指令紧跟第一条指令从表中取出，试说明第一条指令发生异常（见练习题 4.17.1）时流水线的运行情况。给出从第一条指令取指开始到异常处理程序第一条指令完成时的流水线运行图。

**4.17.4** [20] <4.9> 在向量异常处理中，异常处理程序地址表在数据存储器的一个固定位置。改变流水线的实现以支持向量异常处理。重做练习题 4.17.3，这次使用支持向量异常处理的流水线。

**4.17.5** [15] <4.9> 我们想要在仅有一个固定处理程序地址的处理器上模拟向量异常处理（见练习题4.17.4），写出相应的程序。提示：这段程序应识别异常类型，从异常向量表中获得正确地址，然后跳转到该异常处理程序处。

**4.18** 本练习题比较单发射和双发射处理器的性能，并考虑对双发射处理器进行程序优化。根据下表的C代码分别回答下列问题。

```
for(i=0;i!=j;i+=2)
  a[i]=a[i]-a[i+1];
```

在编写MIPS代码时，假设变量被保存在寄存器中，如下表所示。除了空闲的寄存器，其余寄存器都被用来保存变量的值，因此不能再用作其他用途。

| i | j | a | b | x | 空闲 |
|---|---|---|---|---|---|
| R5 | R6 | R1 | R2 | R3 | R10, R11, R12 |

**4.18.1** [10] <4.10> 将这段C代码翻译成MIPS代码。这种翻译必须是直接的，不允许对代码进行重排序以达到更好的性能。

**4.18.2** [10] <4.10> 如果循环仅执行两次后就退出，画出练习题4.18.1中MIPS代码在图4-69的双发射处理器中执行的流水线图。假设处理器能进行完美的分支预测，并且一个周期能取任意两条指令（不仅仅是连续的两条指令）。

**4.18.3** [10] <4.10> 重排序练习题4.18.1中的MIPS代码，以在图4-69的双发射静态调度处理器上获得更好的性能。

368

**4.18.4** [10] <4.10> 重做练习题4.18.2，但这次使用练习题4.18.3中的MIPS代码。

**4.18.5** [10] <4.10> 从单发射处理器到图4-69的双发射处理器，性能的加速比是多少？在单发射和双发射处理器分别运行练习题4.18.1的代码，假设循环执行1 000 000次。与练习题4.18.2相同，假设处理器能进行完美的分支预测，并且一个双发射处理器在同一个周期能取任意两条指令。

**4.18.6** [10] <4.10> 重做练习题4.18.5，这次假设双发射处理器中一条指令可以是任意类型的，而另一条指令必须是非存取指令。

**4.19** 本练习题讨论性能与功耗的关系。假设数据通路各部件（指令存储器、寄存器、数据存储器）的功耗如下表所示，其他部件的功耗可以忽略。

| 指令存储器 | 一次读寄存器 | 写寄存器 | 读数据存储器 | 写数据存储器 |
|---|---|---|---|---|
| 140pJ | 70pJ | 60pJ | 140pJ | 120pJ |

假定数据通路上的部件延迟如下表所示，其他功能部件的延迟可以忽略。

| 指令存储器 | 控制 | 寄存器读或写 | ALU | 数据存储器读或写 |
|---|---|---|---|---|
| 200ps | 150ps | 90ps | 90ps | 250ps |

**4.19.1** [10] <4.3, 4.6, 4.14> 在单周期设计和五级流水线设计中执行一条加法指令的功耗分别是多少？

**4.19.2** [10] <4.6, 4.14> 功耗消耗最大的MIPS指令是那一条？执行这条指令的功耗是多少？

**4.19.3** [10] <4.6, 4.14> 如果功耗是最重要的约束，应该怎样设计流水线？在这种流水线下执行一条lw指令时的功耗减小的比例是多少？

**4.19.4** [10] <4.6, 4.14> 如果像练习题4.19.3中那样设计流水线，其对性能会造成多大的影响？

**4.19.5** [10] <4.6, 4.14> 我们可以去掉MemRead控制信号，即每个周期都读数据存储器（MemRead恒为1）。解释为什么去掉该控制信号后处理器依然能正常工作。它对时钟频率和功耗又有什么影响？

**4.19.6** [10] <4.6, 4.14> 如果一个单元的空闲功耗仅为正常工作时的10%，每个周期指令存储器的功耗是多少？指令存储器消耗的功耗中有多少是空闲功耗？

369

### 小测验答案

**4.1** 控制器，数据通路、存储器，少了输入和输出。

**4.2** 错。边沿触发状态单元可以同时进行读写。

**4.3** I. a; II. c。

**4.4** 是，Branch 与 ALUOp0 是相同的。而且，MemtoReg 和 RegDst 是相反的，不需要额外的反相器。仅使用另外一个信号，并翻转多路选择器的输入即可。

**4.5** I. 因为 lw 的结果而阻塞；II. 旁路第一个 add 的结果写入 $s1；III. 不需要阻塞或旁路。

**4.6** 2 和 4 正确，其余错误。

**4.8** 1. 预测不发生；2. 预测发生；3. 动态预测。

**4.9** 第一条指令，因为在逻辑上它最先执行。

**4.10** 1. 都有；2. 都有；3. 软件；4. 硬件；5. 硬件；6. 硬件；7. 都有；8. 硬件；9. 都有。

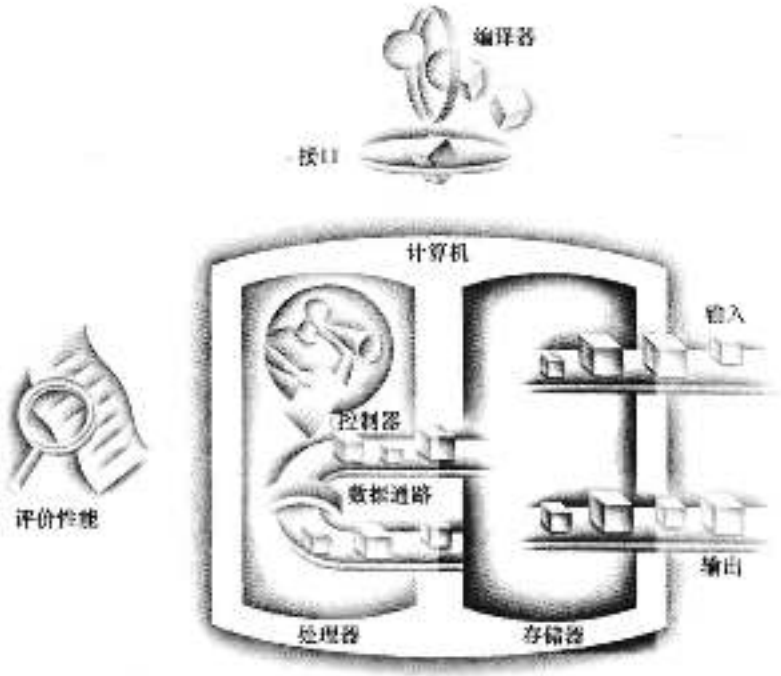
**4.11** 前两个错误，后两个正确。

370
~
371

第5章

Computer Organization and Design: The Hardware/Software Interface

# 大容量和高速度：开发存储器层次结构




计算机的5个经典部件

*在理想情况下，我们希望存储器容量可以无限大，这样，任何特定的情况下……都可以立刻得到需要用到的字……在实际中，我们需要构建一个具有层次结构的存储器，其中的每一层都比它的上一层拥有更大的容量，但访问速度更慢。*

——*A. W. Burks, H. H. Goldstine* 和 *J. von Neumann*

《*Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*》, 1946

## 5.1 引言

从最早期的计算开始，程序员就希望快速存储器的容量可以无限大。这一章主要探讨如何帮助程序员构建一个容量无限大的虚拟快速存储器。在这之前，让我们通过简单的类比方式来介绍将要使用的关键原理和机制。

假如你正在完成一份关于计算机硬件重要历史性发展的论文，你可以从图书馆的书架上精心挑选一些经典计算机书籍，并将它们放在书桌上。你从这些书中找到了需要写的几种重要的计算机，但是没有找到关于 EDSAC 的，因此，你返回书架去寻找其他书，并在早期的英国计算机书籍中找到了一本有关 EDSAC 的书。一旦在你的书桌上有了选好的一些书，你就有可能从这些书中找到你需要的许多内容，这样一来，你的大部分时间只需花在阅读这些书上，而无需返回书架。试比较这两种情况：一种是在你的书桌上有好几本书；另一

种是书桌上只有一本书，你不得不频繁地返回书架，进行还书后取另一本书。在书桌前放一些书会更节省时间。

同样，我们可以构建一个大容量的虚拟存储器，它能像小容量的存储器那样被快速访问。就像你不会同时以相同的概率查阅图书馆中的每一本书那样，一个程序也不会同时以相同的概率访问它全部的代码或数据。否则，不可能让存储器在保持大容量的同时又能快速访问，就像不能既要求你把图书馆中所有的图书放在书桌上，还要求你能保持快速查找一样。

局部性原理不仅适用于在图书馆查找资料的工作方式，而且适用于程序执行的方式。局部性原理表明了在任何时间内，程序访问的只是地址空间相对较小的一部分内容。以下是两种不同类型的局部性：

- **时间局部性**（temporal locality）：如果某个数据项被访问，那么在不久的将来它可能再次被访问。就如刚拿了一本书到书桌上查阅，那么很可能你会很快地再次查阅它。
- **空间局部性**（spatial locality）：如果某个数据项被访问，与它地址相邻的数据项可能很快也将被访问。例如，当你找到一本关于 EDSAC 的早期经典计算机的书籍时，也许紧挨着它的另一本关于早期工业计算机的书籍里同样有你所需的材料，因为图书馆通常将主题相同的书放在同一个书架上以提高空间定位效率。后面我们将看到空间局部性原理如何应用于存储器层次结构。

⇨ 时间局部性：某个数据项在被访问之后可能很快被再次访问的特性。

⇨ 空间局部性：某个数据项在被访问之后，与其地址相近的数据项可能很快被访问的特性。

372 ~ 374

正如查阅书桌上的资料体现了自然的局部性，程序的局部性起源于简单自然的程序结构。例如，大多数程序都包含了循环结构，因此这部分指令和数据将被重复地访问，呈现出了很高的时间局部性。由于指令通常是顺序执行的，因此程序也呈现了很高的空间局部性。对数据的访问同样显示了一种自然的空间局部性。例如，对数组或者记录中的元素进行顺序访问都体现了高度的空间局部性。

我们可以利用局部性原理将计算机存储器组织成为**存储器层次结构**（memory hierarchy）。存储器层次结构由不同速度和容量的多级存储器构成。快速存储器每比特的成本要比慢速存储器高很多，因而通常它们的容量也比较小。

⇨ 存储器层次结构：一种由多存储器层次组成的结构，存储器的容量和访问时间随着离处理器距离的增加而增加。

如图 5-1 所示，较快的存储器靠近处理器，而较慢的、便宜的存储器层次较低。其目的是以最低的价格向用户提供尽可能大的存储容量，同时存取速度与最快的存储器相当。

同样，数据也可以组织成层次化结构：靠近处理器那一层中的数据是那些较远层次中的子集，所有的数据则被存在最慢的底层。我们依然使用图书馆的例子来进行类比，书桌上的书籍是图书馆藏书的一个子集，进而也是学校中所有图书馆藏书的一个子集。而且，离处理器越远的层次访问时间也越长，就像我们在学校图书馆系统中可能遇到的情况一样。

存储器层次结构可以由多层构成，但是数据每次只能在相邻的两个层次之间进行复制。因此我们将注意力重点集中在两个层次上。高层的存储器靠近处理器，比低层存储器容量小但访问速度更快，这是因为它采用了成本更高的技术来实现。如图 5-2 所示，我们将一个两级层次结构中存储信息交换的最小单元称为**块**（block）或**行**（line），就像在图书馆中，一个信息块就是一本书。

375




图5-1 存储器层次的基本结构。存储系统采用层次结构后，用户对于存储器的认识就是：它的容量和容量最大的那层存储器相同，而访问速度和最快的那层存储器相当。在很多个人移动终端设备中，闪存已经代替了磁盘，对于台式计算机和服务器来说可能会在存储器层次中引入新的一层；见5.2节

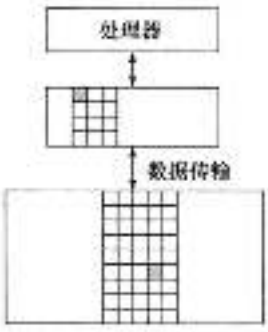



图5-2 存储器层次结构中的每两个层次可以被认为一个是高层次，一个是低层次。在每一层中，存储信息交换的最小单元称为块或者行。通常在层次之间复制时按整块进行传输

如果处理器需要的数据存放在高层存储器中的某个块中，则称为一次命中（这就好像正好从书桌上的一本书中找到所需的信息一样）。如果在高层存储器中没有找到所需的数据，这次数据请求则称为一次缺失。随后访问低层存储器来寻找包含所需数据的那一块（如同从书桌旁走到书架前去寻找所需的书籍）。**命中率**（hit rate）或**命中比率**（hit ratio），是在高层存储器中找到数据的存储访问比例，通常被当成存储器层次结构性能的一个衡量标准。**缺失率**（miss rate）（1－命中率）则是在高层存储器中没有找到数据的存储访问比例。

追求高性能是我们使用存储器层次结构的主要目的，因而命中时间和缺失时间就显得尤为重要。**命中时间**（hit time）是指访问存储器层次结构中的高层存储器所需要的时间，包括了判断当前访问是命中还是缺失所需的时间（相当于浏览书桌上书籍所花费的时间）。**缺失代价**（miss penalty）是将相应的块从低层存储器替换到高层存储器中，以及将该信息块传送给处理器的时间之和（也就是从书架上取另一本书并将它放到桌上的时间）。由于较高存储器层次容量较小并且使用了快速的存储器部件，因此比起对存储器层次中较低层的访问，命中时间要少得多，这也是缺失代价的主要组成部分。（同样，查找书桌上书籍的时间比站起来到书架前查找一本新书所需的时间要少得多。）

- 块或行：可存在于或不存在于 cache 中的信息的最小单元。
- 命中率：在高层存储器中找到目标数据的存储访问比例。
- 缺失率：在高层存储器中没有找到目标数据的存储访问比例。
- 命中时间：访问某存储器层次结构所需要的时间，包括了判断当前访问是命中还是缺失所需的时间。
- 缺失代价：将相应的块从低层存储器替换到高层存储器所需的时间，包括访问块、将数据送层传输、将数据插入发生缺失的层和将信息块传送给请求者的时间。

376

在这一章中我们也将看到，用来构建存储器层次结构的这些概念将影响计算机的许多其他方面，包括操作系统如何管理存储器和 I/O，编译器如何产生代码，甚至对应用程序如何使用计算机也产生一定影响。当然，由于所有程序花费大量时间访问存储器，因而存储系统必然成为评估机器性能的一个主要指标。利用存储器层次结构来达到性能的提升，意味着在过去程序员可以把存储器看成是一个线性的随机访问的存储设备，而现在必须理解存储器层次结构如何

工作才能获得良好的性能。稍后我们将在图 5-18 的示例中说明其重要性，在 5.14 节说明如何使矩阵乘法加倍性能。

由于存储系统对性能至关重要，计算机设计人员在这些系统上花费了大量精力，并致力于开发复杂的机制来提高存储系统的性能。本章我们主要讨论概念性的观点，为了不至于使篇幅过长和使内容太复杂，对许多概念进行了简化和抽象。

**重点** 程序不仅表现出时间局部性，即重复使用最近被访问的数据项的趋势，同时也表现出了空间局部性，即访问与最近被访问过的数据项地址空间相近的数据项的趋势。存储器层次结构利用了时间局部性，将最近被访问的数据项放在靠近处理器的地方；同时它也利用了空间局部性，将一些包含连续字的块移至存储器层次结构的较高层次。

如图 5-3 所示，在存储器层次结构中，离处理器越近的层次容量越小，速度越快。因此，数据在层次结构中的最高层命中将被很快处理。而越关后，需要访问容量大但速度慢的低层存储器层次。如果命中率足够高，存储器层次结构就会拥有接近最高（而且最快）层次的访问速度和接近最低（也是最大）层次的容量。

在很多系统中，存储器是一个真实的层次结构，这意味着除非数据在第 $i+1$ 层存在，否则绝不可能在第 $i$ 层存在。




图 5-3 这幅图说明了存储器层次结构：离处理器越远，容量越大。当采用合适的操作机制时，这种结构允许处理器的访问时间主要由层次结构中的第 1 层来决定，而整个存储器的容量则和第 n 层一样大。本章的主题就是要实现这种结构。尽管本地磁盘一般位于存储器层次结构的底层，但是一些系统会使用磁带或者局域网内的文件系统作为层次结构的更下一层

**小测验**

下面哪些表述通常是正确的？

1. 存储器层次利用了时间局部性。
2. 在一次读操作中，返回的值取决于哪些块在 cache 中。
3. 存储器层次结构的大部分成本处于最高层。
4. 存储器层次结构的大部分容量处于最低层。

377

## 5.2 存储器技术

目前，构建存储器层次结构主要有 4 种技术。主存储器由 DRAM（动态随机存取存储器）实现，靠近处理器的那层（cache）则由 SRAM（静态随机存取存储器）来实现。DRAM 每比特成本要低于 SRAM，但是速度比 SRAM 慢。价格的差异源于 DRAM 每比特占用的存储器空间较少，因此等量的硅制造的 DRAM 的容量会比 SRAM 的要大。速度的差异则由多种因素造成，我们将在附录 B 的 B.9 节中介绍。第三种技术是闪存，这种非易失存储器用作个人移动设备中的二级存储器。第四种技术是磁盘，它通常是服务器中容量最大且速度最慢的一层。以上这些技术的访问时间和每比特的成本变化很大，如下表所示（表中使用的是 2012 年的典型数据）。

| 存储器技术 | 典型访问时间（ns） | 2012 年每 GB 的价格（美元） |
|---|---|---|
| SRAM | 0.5～2.5 | 500～1 000 |
| DRAM | 50～70 | 10～20 |
| Flash | 5 000～50 000 | 0.75～1.00 |
| 磁盘 | 5 000 000～20 000 000 | 0.05～0.10 |

378 本节的余下部分将分别讲授每种存储器技术。

## 5.2.1 SRAM 技术

SRAM 是一种组织成存储单列结构的简单集成电路，通常具有一个读写端口。虽然读写访问时间可能不同，但 SRAM 对任何数据访问时间都是固定的。

SRAM 不需要刷新，并且其访问时间与周期时间非常相近。为了防止读操作时信息丢失，SRAM 的一个基本存储单元通常由 6～8 个晶体管组成。在空闲模式下，SRAM 只需要最小的功率来保持电荷。

过去，在大多数 PC 和服务器系统中通常将 SRAM 芯片从它们的一级、二级，甚至三级 cache 中分离出来。由于摩尔定律的推动，当今的处理器芯片中集成了多层次的 cache，因此独立的 SRAM 芯片几乎在市场上消失了。

## 5.2.2 DRAM 技术

只要给 SRAM 加电，其中的数据就会保持。而在动态 RAM（DRAM）中，存储单元使用电容保存电荷的方式来存储数据。为了对保存的电荷进行读取或写入，使用一个晶体管对该电容进行访问。因为 DRAM 存储每一位都只使用一个晶体管，所以它比 SRAM 密度要高得多，且价格也要便宜很多。由于 DRAM 在电容上保存电荷，因此不能长久地保持数据，从而必须周期性地刷新。与静态存储器 SRAM 相比，这就是将该存储结构称为动态的原因。

为了对单元进行刷新，只需要读出其内容然后写回即可。DRAM 单元中的电荷可以保持几十秒。如果 DRAM 中的每个比特位需要独立的读出后写回，则必须不停地进行刷新操作，这将导致没有时间可用于正常的访问操作。幸运的是，DRAM 采用了一种两级译码结构，可以通过在一个读周期后紧跟一个写周期的方式一次刷新一整行（一行单元共用一个字线）。

图 5-4 给出了一个 DRAM 的内部组织结构，图 5-5 给出了多年来 DRAM 的密度、成本、访问时间的变化。

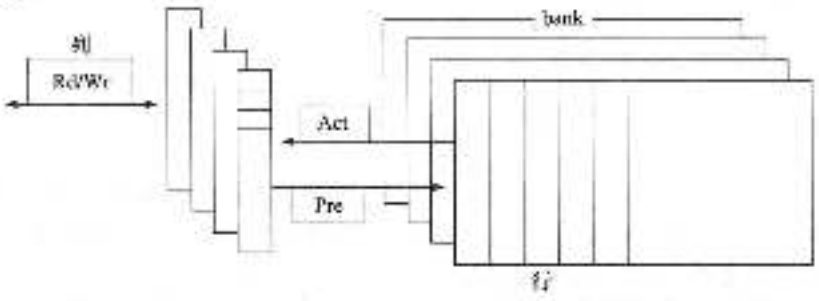



图 5-4 DRAM 的内部组织。现代 DRAM 以 bank（存储块）方式组织，典型的 DDR3 中有 4 个 bank。每个 bank 由多个行组成。发送一条 Pre（预充电）命令能够打开或者关闭一个 bank。使用 Act（激活）命令发送一个行地址，将对应的行中的数据传送到一个缓冲器中。当一行数据在缓冲器中时，无论 DRAM 数据宽度（典型情况为 4、8 或 16 位）是多少，都可以通过指定要传送的数据块大小和数据块在缓冲器中的起始地址的方式连续传送相邻地址的数据。与数据块的传送一样，每条命令使用时钟进行同步

| 生产年份 | 芯片容量 | 每GiB价格 | 访问新的一行/一列的时间 | 访问已经在缓冲器的行中的一列的时间 |
|---|---|---|---|---|
| 1980 | 64Kib | $1 500 000 | 250ns | 150ns |
| 1983 | 256Kib | $500 000 | 185ns | 100ns |
| 1985 | 1Mib | $200 000 | 135ns | 40ns |
| 1989 | 4Mib | $50 000 | 110ns | 40ns |
| 1992 | 16Mib | $15 000 | 90ns | 30ns |
| 1996 | 64Mib | $10 000 | 60ns | 12ns |
| 1998 | 128Mib | $4 000 | 60ns | 10ns |
| 2000 | 256Mib | $1 000 | 55ns | 7ns |
| 2004 | 512Mib | $250 | 50ns | 5ns |
| 2007 | 1Gib | $50 | 45ns | 1.25ns |
| 2010 | 2Gib | $30 | 40ns | 1ns |
| 2012 | 4Gib | $1 | 35ns | 0.8ns |

图 5-5 直到 1996 年，DRAM 芯片容量每 3 年增加为原来的 4 倍，之后增长速度下降非常快。访问时间的减少虽然很慢，但是仍然在持续减少。虽然价格受到其他诸如可用性和需求等因素的影响，但是也基本上按照存储密度增加的速度在降低。每 GiB 的价格没有按照通货膨胀进行调整

行组织结构不但有助于刷新，还有助于性能的提高。为了提高性能，DRAM 为了进行重复访问而对多行进行缓冲。缓冲器与 SRAM 类似：在下一行被访问之前，可通过改变地址来访问缓冲器中的任何一个比特位。由于访问该行中数据的时间短了很多，因此极大地减小了数据访问时间。更宽的芯片也可以增加芯片的存储器带宽。当一行数据在缓冲器中时，无论 DRAM 数据宽度（典型情况为 4、8 或 16 位）是多少，都可以通过指定要传送的数据块大小和数据块在缓冲器中的起始地址的方式连续传送相邻地址的数据。

为了进一步优化与处理器的接口，DRAM 增加了时钟，因此称之为同步 DRAM，简写为 SDRAM。SDARM 的优势在于使用时钟对存储器和处理器保持同步。其速度上的优势主要源于不需要额外指定地址位以突发方式传送多个数据的能力，是在时钟的控制下以突发方式传送连续的数据。最快的版本称为双数据速率（DDR）SDRAM。该名称表示在时钟的上升沿和下降沿都要传送数据，因此可以获得双倍的数据带宽。该技术的最新版本是 DDR4。一个 DDR4-3200 DRAM 每秒可以传输 3 200 兆次，即其时钟频率为 1 600MHz。 379

要支持如此高的带宽需要在 DRAM 内部进行精心组织。与只有一个快速的行缓冲器不同，DRAM 内部可以组织成对多个 bank 进行读或写操作，每个 bank 都有自己的行缓冲器。向不同的 bank 发送一个地址可以允许同时对它们进行读或写操作。例如，对于 4 个 bank 而言，只需要一次访问时间，然后以轮转方式对这 4 个 bank 进行访问就可以提供 4 倍的带宽。这种轮转的访问方式成为地址交叉。

虽然 iPad（见第 1 章）之类的个人移动设备使用独立的 DRAM 颗粒，但服务器的存储器通常是以称为双列存储器模块（Dual Inline Memory Module，DIMM）的小电路板方式买卖。DIMM 通常含有 4～16 块 DRAM 芯片，它们组织成 8 字节位宽。一个使用 DDR4-3200 SDRAM 的 DIMM 每秒可以传送 8 × 3 200 = 25 600 兆字节。这类 DIMM 以其带宽进行命名：PC25600。一个 DIMM 可以有如此多的 DRAM 芯片，但是在特定的传送中只使用其中一部分，因此需要一个术语来表示 DIMM 上共享公共地址线的芯片子集。为了避免与 DRAM 内部的行和 bank 的名字混淆，使用存储器 rank 来表示 DIMM 中的芯片的一个子集。

**练习题** 一种测试 cache 之外的存储器系统的性能的方法是使用流基准程序 [McCalpin, 1995]。它主要测试长向量操作的性能。它们没有时间局部性，并且它们访问的序列比测试的计算机中的 cache 要大。

### 5.2.3 闪存

闪存是一种电可擦除的可编程只读存储器（EEPROM）。

与磁盘和 DRAM 不同，而与其他 EEPROM 技术类似，对闪存的写操作可以使存储位损耗。为了应对该限制，大多数闪存产品都有一个控制器，用来将写操作从已经写入很多次的块中映射到写入次数较少的块中，从而使写操作尽量分散。这种技术称为损耗均衡（wear leveling）。采用损耗均衡技术，个人移动设备很难超过闪存的写极限。这种均衡技术虽然降低了闪存的潜在性能，但是不需要在高层次的软件中监控块的损耗情况。闪存控制器的这种损耗均衡也将制造过程中出错的存储单元屏蔽掉，从而提高其成品率。

### 5.2.4 磁盘存储器

如图 5-6 所示，一个磁质硬盘包含一组圆形磁盘片，它们绕着轴心每分钟转动 5 400 ~ 15 000 周。金属盘片的两侧均被磁性存储材料覆盖，其磁性材料与盒式磁带和录像带的材料相同。为了对硬盘上的信息进行读写，每层的表面有一个包含小的电磁线圈的读写磁头。整个驱动器被永久地密封起来以控制驱动器中的环境，从而使得磁头可以距离驱动器表面非常近。

380 ~ 381

每个磁盘的表面划分为同心圆盘，称为磁道（track）。每个面通常有几万条磁道。每条磁道同样被划分为用于存储信息的扇区（sector）；每条磁道有几千个扇区。每个扇区的容量通常是 512 ~ 4 096 字节。信息在磁介质上保存的顺序为扇区号、一个间隙、包含该扇区纠错码（见 5.5 节）的信息、一个间隙、下一扇区的扇区号。



图 5-6 具有 10 个盘面来读写头的磁盘。当今磁盘的直径是 2.5 ~ 3.5 英寸，并且每个驱动器通常有 1 ~ 2 个圆形磁盘片

- 磁道：位于磁盘表面的数万个同心圆环中的任意一个圆环称为一个磁道。
- 扇区：构成磁盘上磁道的基本单位，是磁盘上数据读写的最小单位。

访问每个盘面的磁头连在一起相互协调运动，因此每个盘面的磁头位于相同的扇区。术语柱面用来表示磁头在给定点时访问到所有盘面上的所有扇区的集合。

为了访问数据，操作系统必须对磁盘进行三步操作。第一步是将磁头移动到适当的磁道之上，这称为寻道（seek），将磁头移动到目标磁道所需的时间称为寻道时间。

- 寻道：把读写磁头移动到磁盘上适当的磁道上面的过程。

磁盘供应商在他们的手册中报告寻道的最小、最大和平均时间。前面两个寻道时间数据比较容易测量，但是平均寻道时间却因与寻道距离相关而难测量。工业界计算平均寻道时间的方

达是对所有可能的寻道时间取平均值。平均寻道时间通常在3～13ms，但是，由于应用程序以及磁盘访问调度策略的不同，且磁盘数据具有局部性，所以实际的平均寻道时间通常只有标称数据的25%～33%。由于对同一文件通常会做连续访问，其操作系统也会尽量把这些访问一起进行调度，所以这种局部性会增加。 382

一旦磁头到达了正确的磁道，就必须等待要访问的扇区转动到读写头下面。该等待时间称为**旋转延时**（rotational latency）。平均延时通常是磁盘转动一周时间的一半。磁盘每分钟转动5 400～15 000周，5 400周的磁盘的平均旋转延时为

$$\text{平均旋转延时} = \frac{0.5\ \text{周}}{5\,400\ \text{RPM}} = \frac{0.5\ \text{周}}{5\,400\ \text{RPM}/\left(60\ \frac{\text{秒}}{\text{分钟}}\right)} = 0.005\,6\ \text{秒} = 5.6\ \text{毫秒}$$

磁盘访问的最后一部分是传输时间，即传输一块数据需要的时间。传输时间是扇区大小、旋转速度和磁道信息密度的一个函数。2012年的传输速率在每秒100～200MB。

⇨ 旋转延时：在磁头定位后，指定扇区通过读写头的所需时间。通常是磁盘转动一周时间的一半。

大多数磁盘控制器的一个复杂问题是，它有一个仅保存最近传输过的扇区数据的cache，从cache中传输数据的速率通常更高，在2012年到了每秒750MB（每秒6Gb）。

现在，块号存放在哪里不能再凭直觉了。前面所述的扇区－磁道－柱面模型有如下假定：邻近的块在同一磁道上；因为访问同一柱面上的块不需要寻道时间，所以访问时间较短；一些磁道与其他磁道距磁头更近。变化的原因是磁盘接口层次的提升。为了加速数据传输，高层次的接口将磁盘组织地更像磁带，而不像随机访问设备。在一个磁面上，逻辑块以弯曲形式顺序排列，尽可能使所有扇区的数据密度相同，从而获得最好的性能。因此，顺序的块可能在不同的磁道上。

概括起来，磁盘和半导体存储器技术的主要差别是磁盘的访问速度慢，这主要是因为它们是机械器件——闪存比磁盘快1 000倍，DRAM比磁盘快100 000倍——但是它们却因为使用适度的成本即可获得很大的存储容量而使得每比特位的成本低了许多——磁盘会便宜10～100倍。与闪存类似，磁盘是非易失的，但却不存在写损耗问题。然而，闪存更加坚固，因此更加适用于个人移动设备。

## 5.3 cache的基本原理

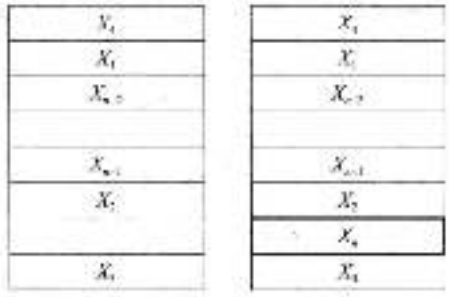
*cache*：**一个隐藏式者存储信息的安全场所。**

——《*Webster's New World Dictionary of the American Language*》，*Third College Edition, 1988*

在前面介绍的图书馆例子中，书桌就好比是高速缓存（cache）——一个存放待用事物（书籍）的安全场所。在早期的商业计算机中，cache是处理器和主存之间的特殊层次。在第4章的数据通路中，存储器就被cache简单地替代了。现在，尽管cache的使用占据了主导地位，但该术语也用来指代那些基于局部性原理来管理的存储器。cache最早出现在20世纪60年代早期的研究型计算机中，后期则被应用于产品型计算机。如今生产的每一台通用计算机，从服务器到低功耗嵌入式处理器，都含有cache。 383

在这一节中，我们先来看一个简单的cache，处理器每次请求一个字，每个块也由一个单

独的字组成（已经熟悉 cache 基本原理的读者可以跳至 5.4 节）。图 5-7 就是一个简单的 cache，要访问的数据项最初不在 cache 中。在请求发出之前，cache 中保存了最近所访问过的数据项 $X_1$，$X_2$，…，$X_{n-1}$ 的集合，而当前处理器所要访问的数据项 $X_n$ 并不在 cache 中。该请求导致了一次缺失，$X_n$ 被从主存调入 cache 之中。



a）访问$X_n$之前　　b）访问$X_n$之后

图 5-7　对字 $X_n$ 访问前后 cache 中的内容。最初 $X_n$ 不在 cache 中。这次访问引起了一次缺失，并促使 cache 从存储器中取回 $X_n$，然后将 $X_n$ 放入 cache 中

观察图 5-7 中的情景，有两个问题需要解决：怎样知道一个数据项是否在 cache 中？此外，如果数据项在 cache 中，如何找到它？这两个问题的答案是相关的。如果每个字都放在 cache 中确定的位置，那么只要它在 cache 中，我们就能直接找到它。在 cache 中为主存中每个字分配一个位置的最简单方法就是根据这个字的主存地址进行分配，这种 cache 结构称为**直接映射**（direct mapped）。每个存储器地址对应到 cache 中一个确定的地址。对直接映射 cache 来说，主存地址和 cache 位置之间的典型映射通常比较简单。例如，几乎所有的直接映射 cache 都使用以下的映射方法：

（块地址）mod（cache 中的块数）

⊖　直接映射：一种 cache 结构，其中每个存储器地址仅仅对应到 cache 中的一个位置。

如果 cache 中的块数是 2 的幂，取模的计算就很简单，只需要取地址的低 $\log_2$（块中的 cache 容量）位。因此，一个 8 块的 cache 可以使用块地址中最低的三位（$8=2^3$）。例如，图 5-8 中，直接映射的 cache 块大小为 8 个字，存储器地址 $1_{10}$（$00001_2$）到 $29_{10}$（$11101_2$）被映射到 cache 中 $1_{10}$（$001_2$）到 $5_{10}$（$101_2$）的位置。

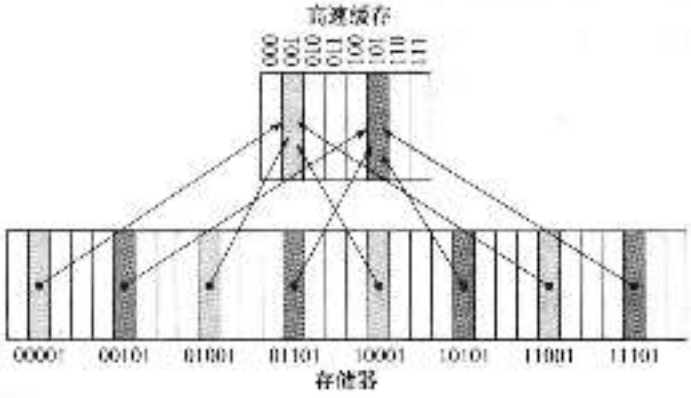



图 5-8　主存地址 0～31 被映射到 cache 中的相同位置，该 cache 中有 8 个字。由于 cache 中有 8 个字，地址 $X$ 被映射到直接映射 cache 字 $X$ mod 8，即低 $\log_2(8)=3$ 位被用作 cache 索引。因此，地址 $00001_2$、$01001_2$、$10001_2$ 和 $11001_2$ 都对应于 cache 中第 $001_2$ 块，而地址 $00101_2$、$01101_2$、$10101_2$ 和 $11101_2$ 都对应于 cache 中第 $101_2$ 块

由于 cache 中每个位置可能对应于主存中多个不同的地址，我们如何知道 cache 中的数据项是否是所请求的字呢？即如何知道所请求的字是否在 cache 中？我们可以在 cache 中增加一组标记（tag），标记中包含了地址信息，这些地址信息可以用来判断 cache 中的字是否就是所

请求的字。标记只需包含地址的高位，也就是没有用来检索 cache 的那些位。例如，在图 5-8 中，标记位只需使用 5 位地址中的高两位，地址低 3 位的索引域则用来选择 cache 中的块。按照定义，任何一个可以放入相同 cache 块中的字的地址的索引域一定是那个块的块号，因此标记位无需包含这些冗余的索引位。

⇒ 标记：表中的一个字段，包含了地址信息，这些地址信息可以用来判断 cache 中的字是否就是所请求的字。

我们还需要一种方法来判断 cache 块中确实没有包含有效信息。例如，当一个处理器启动时，cache 中没有数据，标记域中的值没有意义。甚至在执行了一些指令后，cache 中的一些块 |384| 依然为空，如图 5-7 所示。因此，在 cache 中，这些块的标记应该被忽略。最常用的方法就是增加一个**有效位**（valid bit）来标识一个块是否含有一个有效地址。如果该位没有被设置，则不能使用该块中的内容。

⇒ 有效位：表中的一个字段，用来标识一个块是否含有一个有效数据。

在本节的剩余部分，我们将重点说明如何在 cache 中进行读操作。通常来说，由于读操作不会改变 cache 中的内容，因而处理时比写操作要简单一些。在探讨了读操作和 cache 缺失如何处理的基本原理后，我们将介绍实际计算机中 cache 的设计以及 cache 如何处理写操作。

**重点** cache 可能是预测技术中最重要的例子。它依赖于局部性原理尽可能在存储器层次结构的更高一层中寻找需要的数据，并且当预测错误时提供能够从存储器层次的更低一层中找到并获取正确数据的机制。现代计算机中 cache 预测命中率通常高于 95%（见图 5-47）。

## 5.3.1 cache 访问

下面是对一个容量为 8 块的空 cache 进行 9 次访问的一个序列，包括每次访问的行为。图 5-9 给出了每一次缺失后 cache 内容的变化。由于 cache 中有 8 个块，地址的低 3 位给出了块号。

| 访问的十进制地址 | 访问的二进制地址 | 在 cache 中命中/缺失 | 分配的 cache 块（查找或者放置的位置） |
|---|---|---|---|
| 22 | $10110_2$ | 缺失（图 5-9b） | $(10110_2 \bmod 8) = 110_2$ |
| 26 | $11010_2$ | 缺失（图 5-9c） | $(11010_2 \bmod 8) = 010_2$ |
| 22 | $10110_2$ | 命中 | $(10110_2 \bmod 8) = 110_2$ |
| 26 | $11010_2$ | 命中 | $(11010_2 \bmod 8) = 010_2$ |
| 16 | $10000_2$ | 缺失（图 5-9d） | $(10000_2 \bmod 8) = 000_2$ |
| 3 | $00011_2$ | 缺失（图 5-9e） | $(00011_2 \bmod 8) = 011_2$ |
| 16 | $10000_2$ | 命中 | $(10000_2 \bmod 8) = 000_2$ |
| 18 | $10010_2$ | 缺失（图 5-9f） | $(10010_2 \bmod 8) = 010_2$ |
| 16 | $10000_2$ | 命中 | $(10000_2 \bmod 8) = 000_2$ |

| 索引 | 有效位（V） | 标记 | 数据 |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | N | | |
| 111 | N | | |

a）上电后 cache 的初始状态

| 索引 | 有效位（V） | 标记 | 数据 |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | $10_2$ | 主存（$10110_2$） |
| 111 | N | | |

b）处理地址（$10110_2$）缺失后的 cache 状态

| 索引 | 有效位（V） | 标记 | 数据 |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | Y | $11_2$ | 主存（$11010_2$） |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | $10_2$ | 主存（$10110_2$） |
| 111 | N | | |

c）处理地址（$11010_2$）缺失后的 cache 状态

| 索引 | 有效位（V） | 标记 | 数据 |
|---|---|---|---|
| 000 | Y | $10_2$ | 主存（$10000_2$） |
| 001 | N | | |
| 010 | Y | $11_2$ | 主存（$11010_2$） |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | $10_2$ | 主存（$10110_2$） |
| 111 | N | | |

d）处理地址（$10000_2$）缺失后的 cache 状态

| 索引 | 有效位（V） | 标记 | 数据 |
|---|---|---|---|
| 000 | Y | $10_2$ | 主存（$10000_2$） |
| 001 | N | | |
| 010 | Y | $11_2$ | 主存（$11010_2$） |
| 011 | Y | $00_2$ | 主存（$00011_2$） |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | $10_2$ | 主存（$10110_2$） |
| 111 | N | | |

e）处理地址（$00011_2$）缺失后的 cache 状态

| 索引 | 有效位（V） | 标记 | 数据 |
|---|---|---|---|
| 000 | Y | $10_2$ | 主存（$10000_2$） |
| 001 | N | | |
| 010 | Y | $10_2$ | 主存（$10010_2$） |
| 011 | Y | $00_2$ | 主存（$00011_2$） |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | $10_2$ | 主存（$10110_2$） |
| 111 | N | | |

f）处理地址（$10010_2$）缺失后的 cache 状态

图 5.9 对相应的地址序列给出每次请求缺失后 cache 中的内容、索引和标记域（二进制表示）。cache 初始为空，所有的有效位关闭（N）。处理器请求以下地址：$10110_2$（缺失）、$11010_2$（缺失）、$10110_2$（命中）、$11010_2$（命中）、$10000_2$（缺失）、$00011_2$（缺失）、$10000_2$（命中）、$10010_2$（缺失）以及 $10000_2$（命中）。这些图描出了依次出现的每一次缺失处理后 cache 中的内容。当地址 $10010_2$（18）被访问时，地址为 $11010_2$（26）中的项就要被替换掉，随后再访问 $11010_2$ 会引起缺失。标记域只包含地址的高位部分。cache 块 $i$、标记域为 $j$ 的字的完整地址是 $j \times 8 + i$，或者等效为地址域 $j$ 和索引 $i$ 的连接。例如，上面的 f 图中，索引 $010_2$、标记为 $10_2$ 的块，对应地址 $10010_2$

385
|
387

由于 cache 初始为空，第一次访问的一些数据都会发生缺失。图 5-9 对每一次访问行为进行了描述。第 8 次访问将会对 cache 中的一个块产生冲突的请求。地址 18（$10010_2$）的字将被取到 cache 的第 2 块（$010_2$）中。因此，它将替换掉原先存在于 cache 第 2 块（$010_2$）中的地址为 26（$11010_2$）中的字。这种行为令 cache 具有时间局部性：最近访问过的字替换掉较早访问的字。

这种情况就好比要从书架上取一本书，而书桌上已经没有任何地方可以放这本书了，因此原先摆在书桌的某本书必须被放回书架。在直接映射 cache 中，只有一个位置可以存放最新请求的数据项，因此对于哪个数据项被替换也只有一种选择。

对每个可能的地址，在 cache 中进行如下查找：地址的低位用来找到 cache 中与该地址匹配的唯一项。图 5-10 说明一个地址可以划分为：

- 标记域：用来与 cache 中标记域的值进行比较。
- cache 索引：用来选择块。

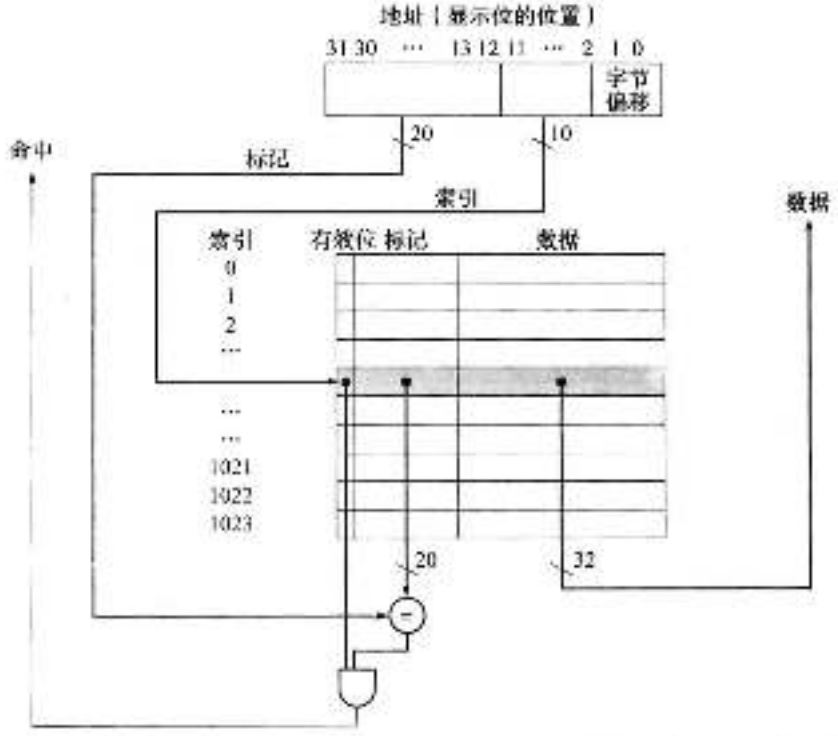



图 5-10 对这个 cache，地址的低位用来选择由数据字和标记组成的一个 cache 项。这个 cache 中有 1 024 个字，即 4KiB。在这一章中，我们假设使用 32 位的地址。cache 中的标记与地址高位相比较，判断 cache 中的项是否符合请求的地址。由于 cache 有 $2^{10}$（1 024）个字，块大小为 1 个字，因此，索引 cache 需要 10 位，剩下的 32 − 10 − 2 = 20 位用来和标记相比较。如果标记和地址的高 20 位相等，并且有效位开启，那么请求在 cache 中命中，相应的字被提供给处理器。否则，发生缺失

cache 块的索引以及标记唯一确定了 cache 块中存放内容的主存地址。由于索引域用来寻址，而且一个 $n$ 位的域有 $2^n$ 种值，直接映射 cache 中项的总数必须为 2 的幂。在 MIPS 体系结构中，由于字是以 4 字节的倍数对齐的，每个地址至少有两位用来指定字中的一个字节。因此当选择块中的一个字时至少两位被忽略。

由于 cache 不仅存储数据也存储标记位，cache 所需的总位数是 cache 大小和地址位数的函数。在前文中提及的块大小为 1 个字，但通常块大小为多字。就像下面的情况：

- 32 位地址。
- 直接映射 cache。
- cache 大小为 $2^n$ 个块，因此 $n$ 位被用来索引。
- 块大小为 $2^m$ 个字（$2^{m+2}$ 字节），因此 $m$ 位用来查找块中的字，两位是字节偏移信息。

标记域的大小为

$$32 - (n + m + 2)$$

直接映射的 cache 总位数为

$$2^n \times (\text{块大小} + \text{标记域大小} + \text{有效位域大小}) \quad \boxed{388}$$

由于块大小为 $2^m$ 个字（$2^{m+5}$ 位），同时我们需要 1 位有效位，因此这样一个 cache 的位数是

$$2^n \times (2^m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 31 - n - m)$$

尽管以上计算是实际的大小，但是通常对 cache 命名只考虑数据的大小而不考虑标记域和有效位域的大小。因此，图 5-10 中是一个 4KiB 的 cache。

**例题：cache 中的位数**

假设一个直接映射的 cache，有 16KiB 的数据，块大小为 4 个字，地址为 32 位，那么该 cache 总共需要多少位？

**答案**

我们知道 16KiB 是 4 096($2^{12}$) 个字，块大小是 4 个字 ($2^2$)，那么就有 1 024 ($2^{10}$) 个块。每个块有 4 × 32 即 128 位的数据，加上 32 − 10 − 2 − 2 位的标记域，再加上一个有效位。因此，总的 cache 大小是

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{12} \times 147 = 147\text{Kib}$$

即能装 16KiB 数据的 cache 总共需要 18.4KiB 的容量。这个 cache 的总位数是数据存储量的 1.15 倍。 □

**例题：将一个地址映射到多字大小的 cache 块中**

考虑一个 cache 中有 64 个块，每块大小为 16 字节，那么字节地址 1 200 将被映射到 cache 中的哪一块？

**答案**

块由下面公式给出：

(块地址) mod (cache 中的块数)

其中块地址为

字节地址 / 每块字节数

389 ~ 390

注意，这个块地址包含了所有在

字节地址 / 每块字节数 × 每块字节数

和

字节地址 / 每块字节数 × 每块字节数 +(每块字节数 − 1)

之间的地址。

因此，由于每个块有 16 字节，字节地址 1 200 对应的块地址为

1 200/16 = 75

对应于 cache 中的块号 (75 mod 64) = 11。事实上，地址 1 200 和 1 215 之间的所有地址都映射在这一块。 □

较大的 cache 块能更好地利用空间局部性以降低缺失率。如图 5-11 所示，增加块大小通常会引起缺失率下降。而当块大小在 cache 容量中所占比例增加到一定程度时，缺失率也随之增加。这是因为此时 cache 中块的数量变得很少，对于这些块将会有大量的竞争发生。结果，就造成一个块中的数据在被多次访问之前就被替换出 cache。另一方面，对于一个太大的块，块中各个字之间的空间局部性也会降低，缺失率降低所带来的益处也会相应减少。

391

仅仅增加块大小所带来的一个更加严重的后果是缺失成本的增加。由较低存储器层次取出块并存放至 cache 中所花费的时间决定了缺失代价。取出块的时间可以分为两部分：第一个字的延迟时间和剩余部分块的传输时间。很显然，除非改变存储系统，否则，传输时间，也就是缺失代价将随着块大小的增大而增加。此外，当块越来越大时，缺失率的改善也开始降低。而当块过于大时，缺失代价的增长超过了缺失率的降低，因此 cache 的性能也随之降低。当然，如果把存储器设计得能更有效地传输较大的块时，我们就能增加块的大小并且进一步改善 cache 性能。这一点我们将在下一节讨论。

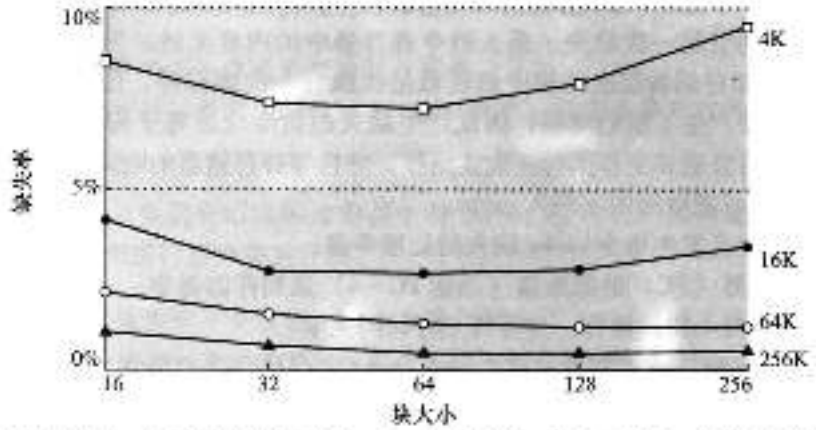



图 5-11 缺失率与块大小。注意到如果相对于 cache 容量来说，块大小太大，缺失率实际上是上升的。每条曲线代表不同容量的 cache（图中没有考虑相联度，稍后讨论）。不幸的是，如果包括块大小，那么 SPEC CPU 2000 追踪信息将花费太长的时间，因此这些数据都基于 SPEC92

**Q1 精解** 缺失时，较大的块会带来长延迟从而增加了缺失代价。要减少这一部分延迟尽管比较困难，但我们可以通过隐藏一些传输时间来有效地降低缺失代价。最简单的方法是提前重启（early restart），即当请中所需字一旦返回就马上继续执行，而不需要等到整个块都传过来之后再执行。许多处理器利用这种技术进行指令访问，效果甚佳。大部分指令访问都具有连续性，因此存储系统每个时钟周期都能传送一个字，只要存储系统能保证及时传递新的指令字，那么当所请求的字返回时，处理器就可以重新开始操作。将这种技术应用于数据 cache 时效果要低一些，这是因为所请求的字可能以一种无法预知的方式分布，而在传输结束前处理器请求另一块中的字的可能性也很高。如果数据传输正在进行，处理器就无法访问数据 cache，因而它必然阻塞。

另一种更复杂的机制是重新组织存储器，使得被请求的字先从存储器传到 cache 中，然后再传送该块的剩余部分，从所请求的字的下一个地址开始传送，再回到块的开始。这种技术被称为请求字优先（requested word first）或者关键字优先（critical word first），它比提前重启要快一些，但与提前重启一样，会因为同样的问题而受到限制。

## 5.3.2 cache 缺失处理

在研究一个真实系统中的 cache 之前，让我们先来看一下控制单元是如何处理 **cache 缺失**（cache miss）的（在 5.9 节将详细介绍 cache 控制器）。控制单元必须能检测到缺失的发生，然后从主存（或者较低一级 cache）中取回所需的数据来处理缺失。如果在 cache 中命中，计算机继续使用该数据，就好像什么都没有发生过。

⊃ cache 缺失：由于数据不在 cache 中而导致被请求的数据不能满足。

命中时，对处理器控制的修改不太重要；缺失时则需要增加一些额外的工作。cache 缺失处理由两部分共同完成：处理器控制单元，以及一个进行初始化主存访问和重新填充 cache 的独立控制器。cache 缺失引起流水线阻塞（见第 4 章），这与中断不同，中断发生时需要保存所有寄存器的状态。当 cache 缺失，我们等待主存操作完成时，整个处理器阻塞，临时寄存器和 392
程序员可见的寄存器中的内容基本被冻结。与之相比，更为复杂的乱序执行处理器在等待 cache 缺失处理的同时，依然能执行其他一些指令。但是，在本节中，我们均假定为顺序执行处理器，当 cache 缺失时其被阻塞。

我们再来讨论一下指令发生缺失时将如何处理，同样的方法略加修改便可以用来处理数据缺失。如果指令访问引起一次缺失，那么指令寄存器中的内容无效。为了将正确的指令取回cache，我们必须通知存储器层次结构中的较低层次执行一次读操作。由于在执行的第一个时钟周期，程序计数器产生了新的增量，因此产生缺失的指令地址等于程序计数器中的值减4。当地址产生时，就可以通知主存执行一次读操作，并且等待存储器的响应（访问主存可能需要多个时钟周期），随后把取回的字写入cache。

现在我们可以定义发生指令cache缺失的处理步骤：

1）把程序计数器（PC）的原始值（当前PC－4）送到存储器中。

2）通知主存执行一次读操作，并等待主存访问完成。

3）写cache项，将从主存取回的数据写入cache中存放数据的部分，并将地址的高位（从ALU中得到）写入标记域，设置有效位。

4）重启指令执行第一步，重新取指，这次该指令在cache中。

数据访问时对cache的控制基本相同：发生缺失时，处理器发生阻塞，直到从存储器中取回数据后才响应。

### 5.3.3 写操作处理

写操作略微不同。如果有一个store指令，我们只将该数据写入数据cache（而不改变主存的内容）；那么，在写入cache之后，主存与cache相应位置中的值将不同。在这种情况下，cache和主存被认为不一致（inconsistent）。保持主存和cache一致性最简单的方法就是将这个数据同时写入主存和cache中，这种方法称为写直达（write-through）法。

➲ 写直达：也译为写通过或写穿。写操作总是同时更新cache和下一存储器层次，以保持二者一致性。

写操作要考虑的另一个主要方面是发生写缺失的情况。我们首先从主存中取出块中的字。数据块被取回并存入cache中后，我们就可以将引起缺失的字重新写入cache中。同时，我们使用全地址将该字写入主存。

尽管这种设计方案能简单地处理写操作，但却无法提供良好的性能。使用写直达的机制，每次写操作都要把数据写入主存之中。这些写操作将花费大量的时间，可能至少要花费100个处理器时钟周期，并且大大降低了机器速度。例如，假设10%的指令是store指令，没有cache缺失
393 的情况下CPI为1.0，每次写操作要额外花费100个周期，就使得CPI为$1.0+100\times10\%=11$，性能降低10倍多。

这个问题的一种解决方法是采用写缓冲（write buffer）。当一个数据在等待被写入主存时，先将它放入写缓冲中。当把数据写入cache和写缓冲后，处理器可以继续执行。当写主存操作完成后，写缓冲里的数据项也得到释放。如果写缓冲已经满了，那么当处理器执行到一个写操作时就必须停下来直到写缓冲中有一个空位置。当然，如果存储器完成写操作的速度比处理器产生写操作的速度慢，那么再多的缓冲器也没有用，因为产生写操作比存储系统接收它们要快。

产生写操作的速度也可能比存储器接收它们的速度慢，尽管这样，仍有可能发生阻塞。当写操作突发产生时，这种情况就会发生。为了减少这种阻塞的发生，通常需要增加处理器写缓冲的深度。

除了写直达，另一种可供选择的方法为写回（write-back）。在写回机制中，当发生写操作时，新值仅仅被写入cache块中。只有当修改过的块被替换时才需要写到较低层存储结构中。写回机制可以提高系统的性能，尤其是当处理器产生写操作的速度和主存处理写操作的速度一

样快甚至更快时；但是，写回机制的实现也比写直达要复杂得多。

- 写缓冲：一个保存等待写入主存数据的缓冲队列。
- 写回：当发生写操作时，新值仅仅被写入 cache 块中，只有当修改过的块被替换时才写到较低层存储结构中。

在本节的剩余部分，我们介绍实际处理器中的 cache，探讨它们如何处理读和写操作。在 5.8 节，我们会对写操作进行更详细的介绍。

**旁注** **精解** 写操作将读操作中不存在的一些复杂情况引入了 cache。这里我们讨论其中的两种情况：写缺失时的策略以及使用写回机制的 cache 中写操作的有效执行。

考虑在写直达机制下的 cache 缺失，最常使用的策略是分配 cache 中的一块，称为写分配（write allocate）。数据块从主存中取回，并且在该块中的恰当区域重写数据。另一种策略则是只更新主存中块的一部分，而不写入 cache 中，这种方法称为写不分配（no write allocate）。这种机制产生的原因是，有时程序会写整个块，就像有时操作系统会将存储器中的一页全部复零一样。在这种情况下，由初始的写缺失引起的取数据就不必要了。一些计算机允许基于每一页来更改写分配策略。

使用写回策略的 cache 比使用写直达策略的 cache 实现有效存储要复杂得多。在写直达的 cache 中，可以将数据写入 cache 并且读标记，如果标记不匹配，就发生缺失。由于 cache 采用写直达策略，在 cache 中重写数据块并不会有危险，因为主存中存储了正确的值。在写回 cache 中，如果 cache 中的数据被重写过并且此时发生缺失，就必须把整块写回主存中。如果在不知道 cache 是否命中（在写直达的 cache 中可以知道）的情况下就简单地根据存储指令重写块，我们就破坏了块的内容。而块本身也没有在存储层的较低层进行备份。

394

在写回 cache 中，由于无法重写块，存储操作需要两个周期（一个周期用来检查命中情况，下一个周期才真正执行写操作），或者需要一个写缓冲来保存数据——通过流水线有效地使存储操作只花费一个周期。如果使用存储缓冲区，处理器在正常的 cache 访问周期内查找 cache 并把数据放入存储缓冲区中。如果 cache 命中，在下一个还没有用到的 cache 访问周期，新数据被从存储缓冲区写入 cache 中。

相比较而言，在写直达 cache 中，写操作总是可以在一个周期内完成。我们读标记位，并且写被选择块的部分区域。如果标记与被写块的地址相同，处理器通常可以继续执行，因为正确的块已经被更新过了。如果标记与被写块的地址不同，处理器产生写缺失并去取对应于该地址块的剩余部分。

很多写回机制的 cache 也使用写缓冲，当在发生缺失替换一个被修改的块时，写缓冲可以起到降低缺失代价的作用。在这种情况下，被修改的数据块移入与 cache 相联的写回缓冲器，同时从主存中读出所需要的数据块。随后，写回缓冲器再将数据写入主存。如果下一次缺失没有立刻发生，当脏数据块必须被替换时，这种方法可以减少一半的缺失代价。

### 5.3.4 一个 cache 的例子：内置 FastMATH 处理器

内置 FastMATH 处理器是一个快速的嵌入式微处理器，它采用 MIPS 架构，cache 实现很简单。在本章的最后，我们将介绍 ARM 和 Intel 微处理器中更为复杂的 cache 设计，但是出于教学的目

的，我们首先分析这个简单的实例。图5-12给出了内置FastMATH处理器数据cache的结构。

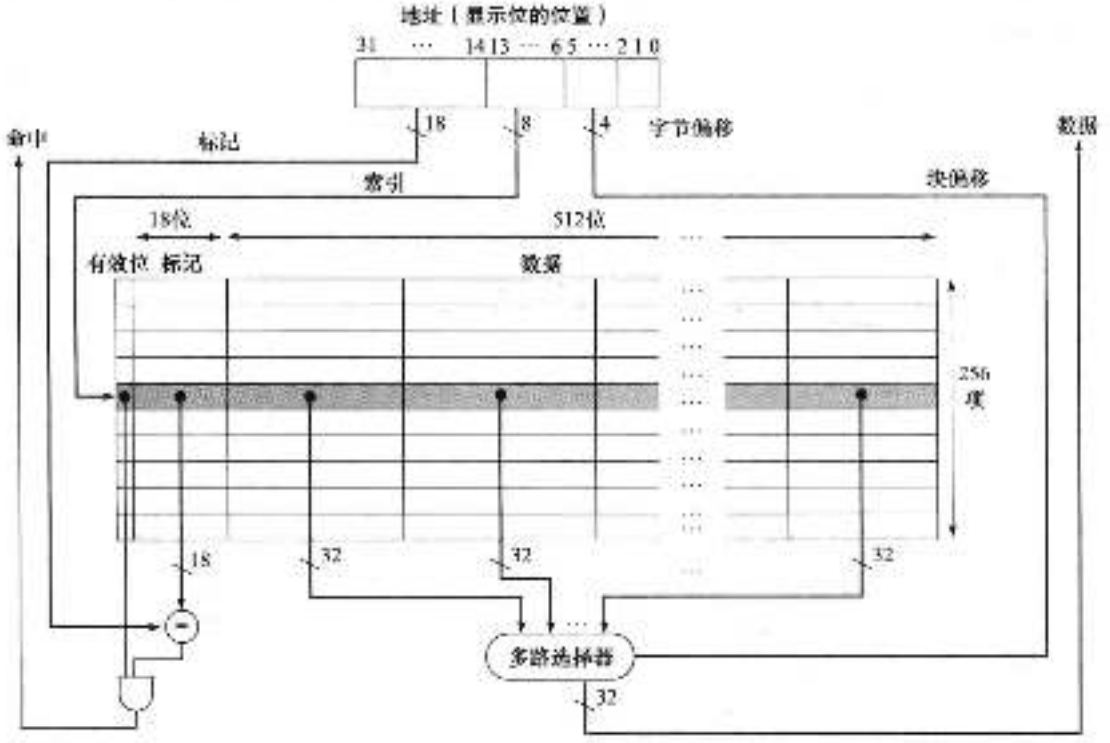



图5-12 内置FastMATH处理器的16KiB的cache，cache中有256块，每块16个字。标记域是18位，索引域是8位，另有一个4位（2～5位）的域用来索引块，并使用一个16选1的多路选择器从块中选择所需的字。实际上，为了消除多路选择器，cache使用一个大容量的RAM单独存放数据，一个更小的RAM则用来存放标记，大容量数据RAM所需的额外地址位由块偏移提供。这样，大容量RAM中字长为32位，字数必须为cache中块数的16倍

该处理器采用12级流水线结构。当以峰值速度执行时，处理器每个时钟周期可以请求一个指令字和一个数据字。为了满足不阻塞流水线的需求，使用了分离的指令cache和数据cache。每个cache容量为16KiB，即4 096个字，每块有16个字。

对cache的读请求很简单，由于使用了分离的指令cache和数据cache，读写每个cache都需要各自独立的控制信号（记住当发生缺失时，需要更新指令cache）。因此，对任何一个cache执行读请求的步骤如下：

1）将地址送到适当的cache中去，该地址来自程序计数器（对于指令访问），或者来自于ALU（对于数据访问）。

2）如果cache发出命中信号，请求的字就出现在数据线上。由于在请求的数据块中有16个字，因此需要选择那个正确的字。块索引域用来控制多路选择器（如图5-12底部所示），从
395 检索到的块中选择16个字中的某个字。

3）如果cache发出缺失信号，我们把地址送到主存。当主存返回数据时，把它写入cache后再读出以满足请求。

对于写操作，内置FastMATH处理器同时提供写直达和写回机制，由操作系统来决定某种应用该使用哪个机制。它有一个只包含一项的写缓冲。

内置FastMATH处理器采用的cache结构的缺失率是怎样的呢？图5-13给出了指令cache和数据cache的缺失率。综合缺失率是在考虑了指令和数据的不同访问频率后每个程序每次访

间的实际缺失率。

| 指令缺失率 | 数据缺失率 | 综合缺失率 |
|---|---|---|
| 0.4% | 11.4% | 3.2% |

图 5-13 内置 FastMATH 处理器执行 SPEC CPU2000 测试程序时指令和数据的近似缺失率。综合缺失率是将 16KiB 的指令 cache 和 16KiB 的数据 cache 结合起来考虑的实际缺失率。它是以指令和数据访问频率为权重，分别考虑指令和数据缺失率后得到的

尽管缺失率是 cache 设计的一个重要标准，但最终的衡量标准是存储系统对程序执行时间的影响。我们将简要介绍缺失率与执行时间之间的关系。

**精解** 混合 cache 容量等于两个分离 cache（split cache）容量的总和。通常来说，混合 cache 具有较高的命中率，其原因是混合 cache 没有将指令所占 cache 块数与数据所占 cache 块数严格区分出来。不过，很多处理器使用分离的指令和数据 cache 以提高 cache 的带宽（同时也可以减少冲突引起的缺失，见 5.8 节）。

下面是与内置 FastMATH 处理器中 cache 容量相同的 cache 的缺失率，混合 cache 的容量等于两个分离 cache 容量之和。

- 总的 cache 容量：32KiB。
- 分离 cache 的实际缺失率：3.24%。
- 混合 cache 的缺失率：3.18%。

分离 cache 的缺失率只是稍差一点。

通过支持指令和数据同时访问来使 cache 带宽加倍，这一优点很容易就克服了缺失率稍微增加的缺点。这一事实也提醒我们缺失率不是衡量 cache 性能的唯一标准，正如 5.4 节所示。

⊃ 分离 cache：一级 cache 由两个独立的 cache 组成，两者可以并行工作，一个处理指令，另一个处理数据。

## 5.3.5 小结

前面我们从介绍最简单的 cache 开始：每块只有一个字的直接映射 cache。在这样的 cache 结构中，命中和缺失都很简单，因为每个字都明确地被写入到一个位置，同时每个字都有单独的标记。为了保持 cache 和主存的一致性，可以使用写直达机制，这样，每次对 cache 进行写操作都会引起主存的更新。不同于写直达机制，写回机制仅在 cache 中有需要被替换的块时才将相应的块复制到主存中去。在后面的章节中我们将进一步讨论这一机制。

396 | 397

为了利用空间局部性，cache 中的块大小必须大于一个字。使用较大的块可以降低缺失率，减少 cache 中与数据存储量相关的标记存储量，从而提高 cache 的效率。尽管块容量的增大可以降低缺失率，但同时也会带来缺失代价的增加。如果缺失代价与块容量成线性关系增长，那么较大的数据块很轻易就能导致性能变差。

为了避免性能损失，可以通过增加主存的带宽来更高效地传输数据块。增加 DRAM 外部带宽最常用的方法包括：增加存储器位宽和交叉存取。DRAM 设计者还改进了处理器和存储器之间的接口以增加突发模式下传输的带宽，以减小使用更大 cache 块带来的开销。

**小测验**

存储系统的速度影响了设计人员如何选择 cache 块的大小。下面哪些 cache 设计者的指导思想是正确的？

1. 存储器延迟越短，cache 块越小。
2. 存储器延迟越短，cache 块越大。
3. 存储器带宽越高，cache 块越小。
4. 存储器带宽越高，cache 块越大。

## 5.4 cache 性能的评估和改进

在这一节中，我们首先探讨评估和分析 cache 性能的方法。随后我们对两种改进 cache 性能的不同技术进行研究。第一种技术是通过减少存储器中不同数据块争用 cache 中同一位置的概率来降低缺失率。第二种技术通过在存储器层次结构中额外增加一层来减少缺失代价。这种技术被称为多级高速缓存（multilevel caching），最初出现在 1990 年售价超过 100 000 美元的高
398 端计算机中，此后该技术被广泛应用于个人移动设备中，而售价只有几百美元。

CPU 时间可以划分为 CPU 执行程序花费的时钟周期和 CPU 等待存储系统花费的时钟周期。通常来说，我们假定 cache 访问命中的开销是 CPU 正常执行周期的一部分。因此，

CPU 时间 =（CPU 执行时钟周期数 + 存储器阻塞的时钟周期数）× 时钟周期

我们假设存储器阻塞的时钟周期数主要来自于 cache 缺失，同时我们将讨论限制在存储系统的简化模型上。在实际的处理器中，由读、写操作引起的阻塞可能十分复杂，并且对性能的准确预测通常需要对处理器和存储系统进行细致的模拟。

存储器阻塞的时钟周期数可以被定义为读操作与写操作引起阻塞的时钟周期数之和：

存储器阻塞时钟周期数 = 读操作引起阻塞的时钟周期数 + 写操作引起阻塞的时钟周期数

读操作阻塞的时钟周期数可以根据每个程序中读的次数、读操作发生缺失时的代价（缺失处理需要的时钟周期）以及读缺失率来定义：

读操作阻塞的时钟周期数 =（读的次数 / 程序数）× 读缺失率 × 读缺失代价

写操作的情况就要复杂一些。对于写直达机制，有两种情况引起阻塞：一种是写缺失，它通常要求在继续执行写操作之前取回数据块（详情参考 5.3.3 节精解中关于写处理的详细介绍）；另一种是写缓冲区阻塞，当写操作发生时写缓冲已满则可能发生这种情况。因此，写操作阻塞的时钟周期数为这两种情况阻塞的时钟周期数之和：

写操作阻塞的时钟周期数 =［（写的次数 / 程序数）× 写缺失率 × 写缺失代价］+ 写缓冲区阻塞

由于写缓冲区阻塞不仅仅取决于频率，还取决于写操作的执行时机，因此这样的阻塞不能由一个简单公式来计算。幸运的是，如果系统中写缓冲区的深度合适（例如，4 个或多个字），并且存储器接收写操作的速率要明显超过程序中平均写频率（例如，是它的两倍），写缓冲区的阻塞将变得很少，可以将其忽略。如果系统不能达到这些标准，则说明它设计得不够好；设
399 计人员应该使用更深的写缓冲区或者使用写回机制。

写回机制同样可能产生额外的阻塞。阻塞的产生原因是，当数据块被替换时需要将其写回到主存中。我们将在 5.8 节中对此进行更详细的讨论。

在大部分写直达 cache 结构中，读和写的缺失代价是一样的（都是从主存中取回数据块的时间）。如果假设写缓冲区阻塞可以被忽略，那么我们可以合并读写操作并共用一个缺失率和缺失代价：

存储器阻塞时钟周期数 =（存储器访问次数 / 程序数）× 缺失率 × 缺失代价

也可以表示如下：

存储器阻塞时钟周期数 =（指令数 / 程序数）×（缺失数 / 指令）× 缺失代价

让我们通过一个简单的例子来帮助理解 cache 的性能对处理器性能的影响。

**例题：计算 cache 性能**

假设指令 cache 的缺失率为 2%，数据 cache 的缺失率为 4%，处理器的 CPI 为 2，没有存储器阻塞，且每次缺失的代价为 100 个时钟周期，那么配置一个从不发生缺失的理想的 cache，处理器的速度快多少？这里假定全部 load 和 store 的频率为 36%。

**答案**

根据指令计数器（$I$），由指令缺失引起的时钟周期损失数为

$$\text{指令缺失时钟周期数} = I \times 2\% \times 100 = 2.00 \times I$$

由于所有 load 和 store 指令出现的频率为 36%，我们可以计算出数据缺失引起的时钟周期损失数：

$$\text{数据缺失时钟周期数} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

400

总的存储器阻塞时钟周期数为 $2.00 \times I + 1.44 \times I = 3.44 \times I$，每条指令的存储器阻塞超过 3 个时钟周期。因此，包括存储器阻塞在内的总的 CPI 是 2 + 3.44 = 5.44。由于指令计数器或时钟频率都没有改变，CPU 执行时间的比率为

$$\begin{aligned}&\text{有阻塞的 CPU 执行时间} / \text{完美理想 cache 的 CPU 执行时间}\\ &= (I \times \text{CPI}_{\text{阻塞}} \times \text{时钟周期}) / (I \times \text{CPI}_{\text{理想}} \times \text{时钟周期})\\ &= \text{CPI}_{\text{阻塞}} / \text{CPI}_{\text{理想}} = 5.44/2\end{aligned}$$

因此，配置了理想的 cache 的 CPU 的性能是原来的 5.44/2 = 2.72 倍。 □

如果处理器速度很快，而存储系统却不快，又会发生什么？在第 1 章介绍的 Amdahl 定律提醒我们这样一个事实：存储器阻塞花费的时间占据执行时间的比例会上升。一些简单的例子会说明这个问题有多严重。假设我们加速上面例子中的计算机，通过改进流水线，在不改变时钟频率的情况下，将 CPI 从 2 降到 1。那么具有 cache 缺失的系统的 CPI 为 1 + 3.44 = 4.44，而配置理想的 cache 的系统性能是它的

$$4.44/1 = 4.44$$

倍。存储器阻塞所花费的时间占据整个执行时间的比例则从

$$3.44/5.44 = 63\%$$

上升到

$$3.44/4.44 = 77\%$$

同样，仅仅提高时钟频率而不改进存储系统也会因 cache 缺失的增加而加剧性能的流失。

前面的例子和等式是建立在命中时间不计入计算 cache 性能的假设之上。很明显，如果命中时间增加，那么从存储系统中存取一个字的总时间也会增加，继而导致处理器时钟周期的增加。我们还将看到其他一些实例以了解导致命中时间略微增加的原因，一个例子是 cache 容量 401
的增加。显然，一个大容量的 cache 访问时间也较长，就像图书馆的书桌很大（有 $3m^2$），要找到桌上的一本书必然要花费很长的时间。命中时间的增加相当于又增加了一级流水线，因为 cache 命中操作需要多个时钟周期完成。尽管计算深度流水对性能的影响会更复杂，但在某种程度上，大容量 cache 命中时间的增加反而会影响命中率的改进使其不起作用，从而导致处理器性能的下降。

为了分别找到在命中和缺失情况下数据访问时间对性能影响的证据，设计人员有时会使用平均存储器访问时间（AMAT）作为检测 cache 设计的方法。平均存储器访问时间是综合考虑了命中、缺失以及不同访问的频率后得出的访存平均时间，它等于下面的公式：

$$\text{AMAT} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

**例题：计算平均存储器访问时间**

处理器时钟周期的时间为 1ns，缺失代价是 20 个时钟周期，缺失率为每条指令 0.05 次缺失，cache 访问时间（包括命中判断）为 1 个时钟周期。假设读操作和写操作的缺失代价相同并且忽略其他写阻塞。请计算 AMAT。

**答案**

每条指令的平均存储器访问时间为

$$\text{AMAT} = \text{命中时间} + \text{缺失率} \times \text{缺失代价} = 1 + 0.05 \times 20 = 2 \text{ 个时钟周期}$$

即 2ns。 □

下一节我们将讨论另一种 cache 组织结构，这种结构减少了缺失率，但是有时可能会增加命中时间。在 5.15 节中我们将给出其他的例子。

### 5.4.1 通过更灵活地放置块来减少 cache 缺失

到目前为止，我们将一个块放入 cache 中，采用的是最简单的定位机制：一个块只能放到 cache 中一个明确的位置。正如前面所述，这种方法称为直接映射，因为存储器中任何一块都被直接映射到存储器层次结构中较高层的唯一位置。实际上，有一整套放置块的方法。直接映射，是一种极端的情况，此时一个块被精确地放到一个位置。

402

另一种极端方式是：一个块可以被放置在 cache 中的任何位置。这种机制称为**全相联**（fully associative），因为存储器中的块可以与 cache 中任何一项相关。在全相联 cache 中要找一个指定的块，由于该块可能被存放在 cache 中的任何位置，因此需要检索 cache 中所有的项。为了使检索更加有效，它是由一个与 cache 中每个项都相关的比较器并行完成的。这些比较器加大了硬件开销，因而，全相联只适合块数较少的 cache。

介于直接映射和全相联之间的设计是**组相联**（set associative）。在组相联 cache 中，每个块可被放置的位置数是固定的。每个块有 $n$ 个位置可放的 cache 被称作 $n$ 路组相联 cache。一个 $n$ 路组相联 cache 由很多个组构成，每个组中有 $n$ 块。根据索引域，存储器中的每个块对应到 cache 中唯一的组，并且可以放在这个组中的任何一个位置上。因此，组相联映射将直接映射和全相联映射结合起来：一个块首先被直接映射到一个组，然后检索该组中所有的块判断是否匹配。例如，图 5-14 是根据这三种策略，块 12 被放置在一个容量为 8 块的 cache 中的情况。

- 全相联 cache：cache 的一种组织方式，块可以放置到 cache 中的任何位置。
- 组相联 cache：cache 的另一种组织方式，块可以放置到 cache 中的部分位置（至少两个）。

回想直接映射的 cache，一个存储块的位置是这样给出的：

（块号）mod（cache 中的块数）

而在组相联 cache 中，包含存储块的组是这样给出的：

（块号）mod（cache 中的组数）

由于该块可能被放在组中的任何一个位置，因此组中所有块的标记都要被检索。而在全相联 cache 中，块可以被放在任何位置，因此 cache 中全部块的标记都要被检索。

我们同样可以把所有的块定位策略看成是组相联的一个特例。图 5-15 显示了一个 8 块的 cache 可能的相联结构。直接映射 cache 是一个简单的一路组相联 cache：cache 的每项有一个

块，并且每组只有一个元素。有 $m$ 项的全相联 cache 可以看成是一个简单的 $m$ 路组相联 cache，它只有一个组，组里有 $m$ 块，每一项可以放在该组的任何一块中。

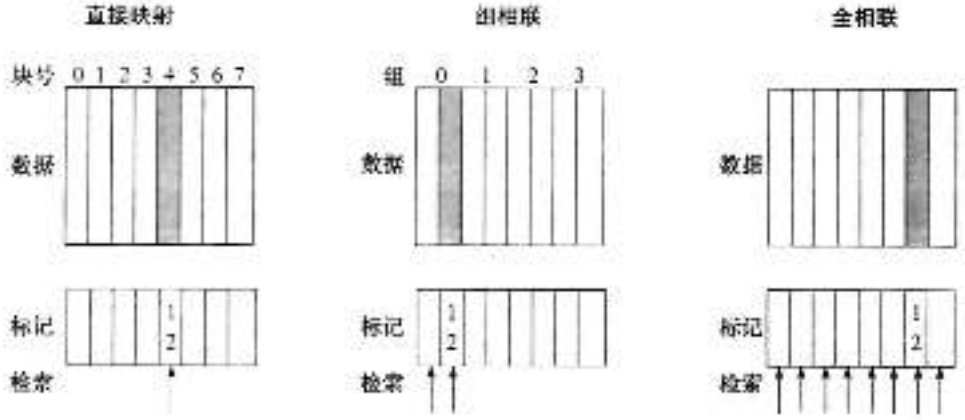



图 5-14 地址为 12 的主存块在 cache 中的位置。cache 容量为 8 块。采用直接映射、组相联以及全相联机制。在直接映射方式下，主存块 12 只能放置在 cache 中唯一的块中，该块为（12 mod 8）=4。在两路组相联 cache 中，有 4 个组，主存块 12 必须放在第（12mod 4）=0组中；主存块可以放在该组的任何位置。在全相联方式下，块地址为 12 的主存块可以放在 cache 中 8 个块的任意一块

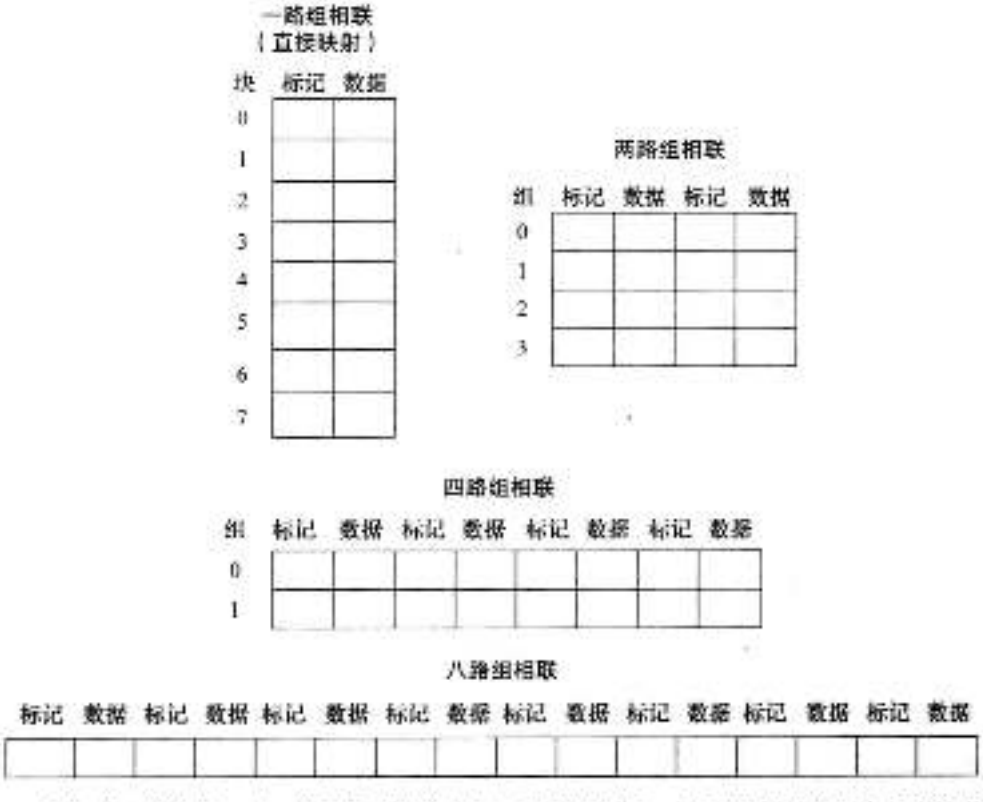



图 5-15 一个拥有 8 个块的 cache 被配置成直接映射、两路组相联、四路组相联以及全相联结构。cache 中块的总数等于组数乘以相联度。因此，对于一个固定大小的 cache，增加相联度的同时减少了组数，同时也就增加了每组的块数。对于容量为 8 个块的 cache，一个八路组相联的 cache 也就等同于一个全相联 cache

提高相联度的好处在于它通常能够降低缺失率，如下例所示。而主要的缺点则是增加了命中时间，稍后我们将详细讨论。

**例题：cache 的缺失与相联度**

假设有三个小的 cache，每个 cache 都有 4 个块，块大小为 1 个字。第一个 cache 是全相联方式，第二个是两路组相联，第三个是直接映射。若按以下地址 0、8、0、6、8 依次访问时，求每个 cache 的缺失次数。

**答案**

直接映射 cache 最简单，首先让我们判断每个地址对应的 cache 块：

| 块地址 | cache 块 |
|---|---|
| 0 | (0 mod 4) = 0 |
| 6 | (6 mod 4) = 2 |
| 8 | (8 mod 4) = 0 |

现在，在每次引用后我们填入 cache 的内容，空白项表示无效的块。加粗的项表示在相关引用中，有一个新的项被加入 cache 中，未加粗的项则表示 cache 中旧的项。

| 被访问的存储器的块地址 | 命中/缺失 | 引用后 cache 中的内容 | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| 0 | 缺失 | **主存 [0]** | | | |
| 8 | 缺失 | **主存 [8]** | | | |
| 0 | 缺失 | **主存 [0]** | | | |
| 6 | 缺失 | 主存 [0] | | **主存 [6]** | |
| 8 | 缺失 | **主存 [8]** | | 主存 [6] | |

直接映射 cache 的 5 次访问产生 5 次缺失。

组相联 cache 有两组（组 0 和 1），每组有两个块。我们首先来确定每个块地址映射到哪一组：

| 块地址 | cache 组 |
|---|---|
| 0 | (0 mod 2) = 0 |
| 6 | (6 mod 2) = 0 |
| 8 | (8 mod 2) = 0 |

由于缺失时，我们需要选择替换组中的某一项，因此需要一个替换规则。组相联 cache 通常会选择替换一组中最近最少使用的块；也就是说，在过去最久的时间用到的那一块将被替换（稍后我们将详细讨论其他替换规则）。使用这个替换策略，每次引用后组相联 cache 中的内容如下所示：

4035

| 被访问的存储器的块地址 | 命中/缺失 | 引用后 cache 中的内容 | | | |
|---|---|---|---|---|---|
| | | 组 0 | 组 0 | 组 1 | 组 1 |
| 0 | 缺失 | **主存 [0]** | | | |
| 8 | 缺失 | 主存 [0] | **主存 [8]** | | |
| 0 | 命中 | 主存 [0] | 主存 [8] | | |
| 6 | 缺失 | 主存 [0] | **主存 [6]** | | |
| 8 | 缺失 | **主存 [8]** | 主存 [6] | | |

注意到当块 6 被访问时，它将块 8 替换掉了。因为比起块 0，块 8 是最近最少被使用的那一块。两路组相联 cache 总共有 4 次缺失，比直接映射的 cache 少一次。

全相联 cache 有 4 个块（在一组中），存储器中任意一块可放到 cache 的任何位置。全相联 cache 性能最好，仅有 3 次缺失。

| 被访问的存储器的块地址 | 命中/缺失 | 引用后 cache 中的内容 | | | |
|---|---|---|---|---|---|
| | | 块 0 | 块 1 | 块 2 | 块 3 |
| 0 | 缺失 | 主存 [0] | | | |
| 8 | 缺失 | 主存 [0] | 主存 [8] | | |
| 0 | 命中 | 主存 [0] | 主存 [8] | | |
| 6 | 缺失 | 主存 [0] | 主存 [8] | 主存 [6] | |
| 8 | 命中 | 主存 [0] | 主存 [8] | 主存 [6] | |

对于这一系列的访问，三次缺失是我们可以得到的最好结果，因为有三个不同地址的块被访问。注意，如果 cache 中有 8 个块，两路组相联 cache 将不会发生替换（请读者自己验证），并且缺失次数与全相联 cache 的一样多。同样，如果有 16 块，这 3 种 cache 会有相同的缺失次数。上面的例子已经说明了在判断 cache 性能时，cache 容量和相联度不能分开考虑。 □

相联度能使缺失率下降多少呢？图 5-16 显示了一个容量为 64KiB，块大小为 16 字的数据 cache，当相联度从直接映射到八路组相联变化时性能的改进情况。从一路组相联到两路组相联，缺失率下降了大约 15%，但是更高的相联度对缺失率的改善就很小了。 406

## 5.4.2 在 cache 中查找一个块

现在，我们考虑在组相联的 cache 中如何查找一个 cache 块。正如在直接映射 cache 中一样，组相联 cache 中每一块都包含一个地址标记用来给出块地址。在被选中的组中每一块的标记都要进行检测，从而判断是否和来自处理器的块地址相匹配。图 5-17 解析了地址。索引值用来选择包含所需地址的组，该组中所有块的标记都将被检索。由于速度是最根本的，所以被选中的组中所有块的标记并行检索。就像在全相联 cache 中一样，组相联 cache 使用顺序检索将使得命中时间太长。

| 相联度 | 数据缺失率 |
|---|---|
| 1 | 10.3% |
| 2 | 8.6% |
| 4 | 8.3% |
| 8 | 8.1% |

图 5-16 使用与内置 FastMATH 处理器相似的 cache 结构，相联度从一路到八路，采用 SPEC CPU2000 基准测试程序测出的数据 cache 缺失率。10 个 SPEC CPU2000 测试程序的结果来自 Hennessy 和 Patterson（2003）

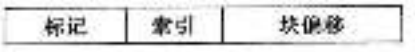



图 5-17 组相联或者直接映射 cache 中地址的三个组成部分。索引位用来选择一个组，标记位用来和选中组中的块进行比较来选择块，块偏移是块中被请求数据的地址

如果 cache 总容量保持不变，提高相联度就增加了每组中的块数，也就是并行查找时同时比较的次数：相联度每增加到两倍就会使每组中的块数加倍而使组数减半。相应地，相联度每增加到两倍，检索位就会减少 1 位，标记位增加 1 位。在全相联 cache 中，只有一组有效，所有块必须并行检测，因此没有索引，除了块偏移地址，整个地址都需要和每个 cache 块的标记进行比较。换句话说，我们不使用索引位就可以查找整个 cache。

在直接映射 cache 中，只需要一个比较器，这是因为每一项只能对应 cache 中唯一的块，并且，我们通过索引就能很简单地访问 cache。图 5-18 是一个四路组相联 cache，需要 4 个比较器以及一个 4 选 1 的多路选择器，用来在选定组中的 4 个成员之间进行选择。cache 访问包括检索相应的组，然后在组中检测标记。一个组相联 cache 的开销包括额外的比较器以及由于对组里数据块进行比较和选择而产生的延迟。

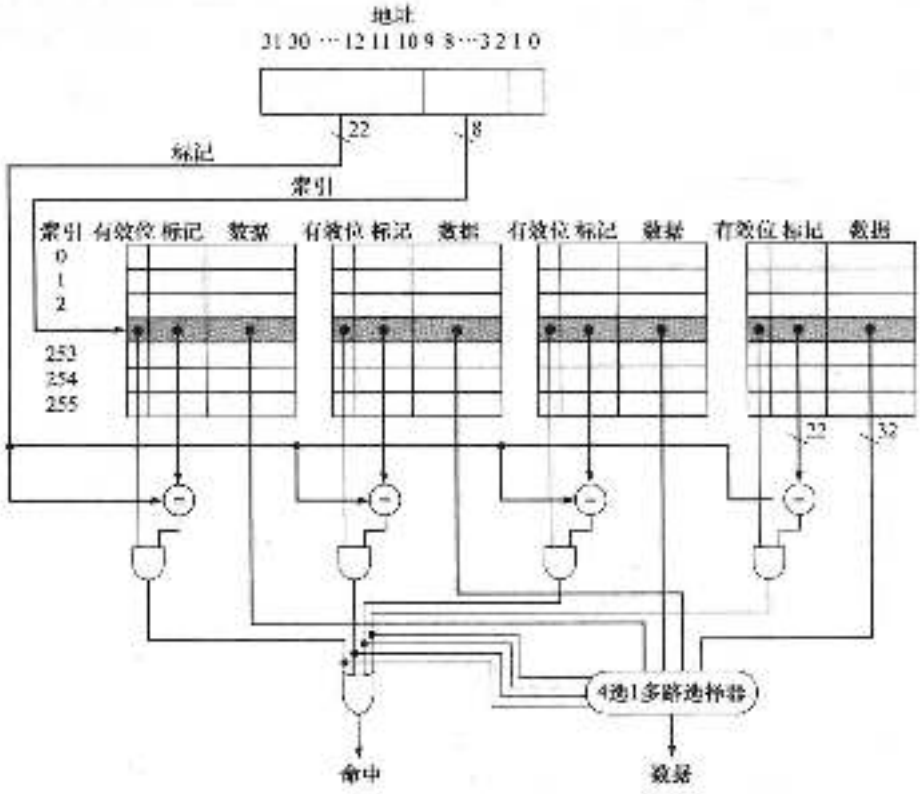



图 5-18 实现一个四路组相联的 cache 需要 4 个比较器和一个 4 选 1 的多路选择器来判断被选中的组中哪一个单元（如果有）与标记匹配。比较器的输出通过使用带有译码选择信号的多路选择器在选中组里的 4 个块之中选择一个数据。在一些具体实现中，cache RAM 数据部分的输出使能信号可以用来选择驱动输出的组中的数据项。输出使能信号来自比较器，使得匹配的单元驱动数据的输出。这种结构不需要使用多路选择器

在任何存储器层次结构中选择直接映射、组相联，还是全相联映射，需要在缺失代价和相联度实现的代价之间进行权衡。既要考虑时间，也要考虑额外的硬件。

**精解** 内容可寻址存储器（Content Addressable Memory，CAM）是一种将比较器和存储单元结合在一个部件上的电路结构。它不像 RAM 那样根据输入读数据，而是由用户提供数据，然后 CAM 查看它是否有副本并且返回匹配项的索引。CAM 的出现意味着设计者能提供更高的相联度，这比在 SRAM 和比较器之外需要构建额外才能实现的相联度还要高。在 2013 年，CAM 更大的容量和功耗使得两路和四路组相联结构一般采用标准的 SRAM 和比较器构建，八路以及更多路组相联的结构则由 CAM 构建。

407
↓
408

## 5.4.3 替换块的选择

当直接映射的 cache 发生缺失时，被请求的块只能放置于 cache 中唯一位置，而原先占据

那个位置的块就必须被替换掉。在相联的 cache 中，被请求的块放置在什么位置需要进行选择，因此替换哪一块也要进行选择。在全相联 cache 中，所有的块都将可能被替换。在组相联 cache 中，我们将在选中的组中挑选被替换的块。

最常用的方法是**最近最少使用**（Least Recently Used，LRU）法，也是我们在前面例子中使用的方法。在 LRU 算法中，被替换的块是最久没有使用的那一块。前面组相联的例子中就使用了 LRU 算法，这也是为什么我们替换主存（0）那块而不是主存（6）。

⇨ 最近最少使用：一种替换策略，总是替换很长时间没有使用的块。

LRU 替换算法的实现是通过跟踪每一块的相对使用情况。对于一个两路组相联 cache，跟踪组中两个数据项的使用情况可以这样实现：在每组中单独保留一位，通过设置该位指出哪一项被访问过。当相联度提高时，LRU 的执行就变得困难些；在 5.8 节中，我们将会讨论另一种替换机制。

**例题：标记位大小与组相联**

提高相联度需要更多比较器，同时 cache 块中的标记位数也需要增加。假设一个 cache 有 4 096 个块，块大小为 4 个字，地址为 32 位，请分别计算在直接映射、两路组相联、四路组相联和全相联映射中，cache 的总组数以及总的标记位数。

**答案**

由于块大小为 16（$=2^4$）字节，32 位地址域中的 32 − 4 = 28 位用来提供索引和标记位。直接映射中组数和块数一样，由于 $\log_2(4\,096) = 12$，因此有 12 位是索引位；因此总的标记位数是（28 − 12）× 4 096 = 16 × 4 096 = 66Kb。

相联度每增加 1 倍，组数就会减少 1/2，因此用来索引 cache 的位数也要相应减 1，而标记位则是增 1。因此，对于一个两路组相联 cache，有 2 048 个组，总的标记位数为（28 − 11）× 2 × 2 048 = 34 × 2 048 = 70Kb。而四路组相联中组数为 1 024，那么总的标记位数为（28 − 10）× 4 × 1 024 = 72 × 1 024 = 74Kb。

对于全相联 cache，只有一个有 4 096 个块的组，标记位是 28 位，因此总的标记位数是 28 × 4 096 × 1 = 115Kb。 □ 409

## 5.4.4 使用多级 cache 结构减少缺失代价

所有现代计算机都使用 cache。为了进一步减小现代处理器高时钟频率与日益增长的 DRAM 访问时间之间的差距，大多数微处理器都会增加额外一级 cache。这种二级 cache 通常位于芯片内，当一级 cache 缺失时就会访问它。如果二级 cache 中包含所需要的数据，那么一级 cache 的缺失代价就是二级 cache 的访问时间，这要比访问主存快得多。如果一级和二级 cache 中均不包含所需的数据，就需要访存，这样就会产生更大的缺失代价。

使用二级 cache 后，性能能改进多少？下面这个例子将会告诉我们。

**例题：多级 cache 的性能**

假定我们的处理器基本的 CPI 为 1.0，所有访问在一级 cache 中均命中，时钟频率为 4GHz。假设主存访问时间为 100ns，其中包括缺失处理时间。设一级 cache 中每条指令缺失率为 2%。如果增加一个二级 cache，命中或缺失访问的时间都是 5ns，而且容量大到必须使访问主存的缺

失率减少到0.5%，这时的处理器速率能提高多少？

**答案**

主存的缺失代价为

100ns/(0.25ns/时钟周期) = 400个时钟周期

只有一级cache的处理器的有效CPI由下列公式给出：

总为CPI = 1.0 + 每条指令的存储器阻塞时钟周期 = 1.0 + 2% × 400 = 9

对于两级cache，一级cache缺失时可以由二级cache或者主存来处理。访问二级cache时的缺失代价为

410

5ns/(0.25ns/时钟周期) = 20个时钟周期

如果缺失能由二级cache处理，那么这就是整个缺失代价。如果缺失处理需要访存，总的缺失代价就是二级cache和主存的访问时间之和。

因此，对一个两级的cache，总的CPI是两级cache的阻塞时钟周期和基本CPI的总和：

总的CPI = 1 + 一级cache中每条指令的阻塞 + 二级cache中每条指令的阻塞

= 1 + 2% × 20 + 0.5% × 400 = 1 + 0.4 + 2.0 = 3.4

因此，有二级cache的处理器性能是没有二级cache处理器性能的

9.0/3.4 = 2.6

倍。

我们还可以使用另一种方法来计算阻塞时间。在二级cache命中的阻塞周期为（2% − 0.5%）×20 = 0.3；而访问主存的阻塞周期必须同时包括访问二级cache和访问主存的时间，为0.5% ×（20 + 400）= 2.1。对它们求和为1.0 + 0.3 + 2.1，同样等于3.4。 □

一级cache和二级cache的设计思想明显不同，这是因为对于单级cache，另一级cache的存在改变了最佳选择。特别是两级cache的结构使得一级cache致力于减少命中时间获得较短的时钟周期或者较少的流水级，二级cache则主要针对改善缺失率以减少长时间的访存代价。

通过将每一级cache与最优化单级cache的设计进行比较，我们可以看出这些变化对两级cache的影响。与单级cache相比，**多级cache**（multilevel cache）中的一级cache通常很小。另外，一级cache的块容量通常也很小，再伴随小容量的cache使得缺失代价降低。相比之下，由于二级cache的访问时间不是关键，因此二级cache的容量比一般的单级cache要大得多，块容量也比单级cache中的要大。它还经常使用比一级cache更高的相联度以减少缺失率。

⊃ 多级cache：存储系统由多级cache组成，而不仅仅只有主存和一个cache。

**理解程序性能** 我们用各一些方法对冒泡排序（Bubble Sort）、快速排序（Quicksort）和基数排序（Radix Sort）等进行分析，希望找到最好的排序算法。图5-19a说明了使用基数排序和快速排序时，指令执行的情况。果然，对于大的数据，在操作次数上，基数排序比快速排序要有优势。图5-19b是每个排序项平均所需的时间，而不是执行的指令数。我们可以看到

411

开始两条曲线的轨迹与图5-19a中相似，但是随着排序数据的增加，基数排序的曲线开始偏离，这是为何？图5-19c用每项排序平均cache缺失数解答了这个问题：快速排序一直有比基数排序少得多的每项缺失数。

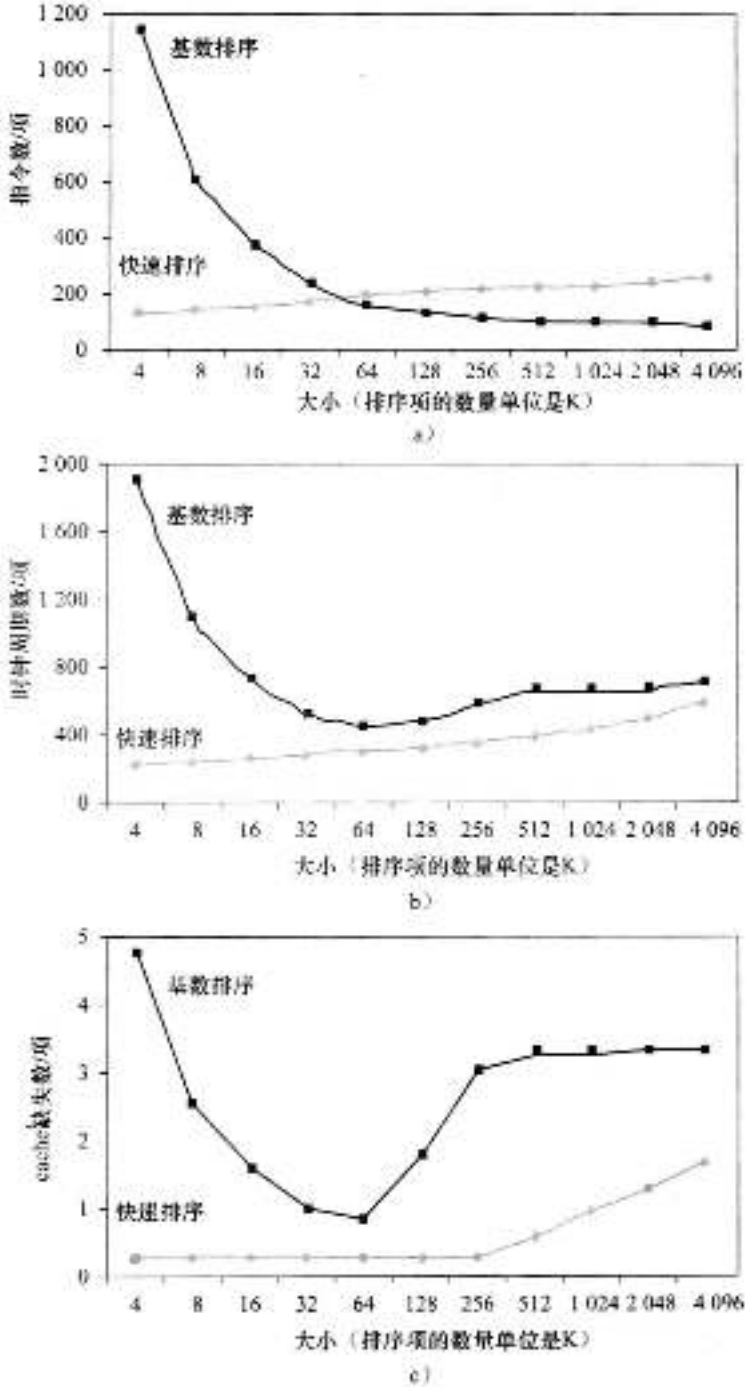



图 5-19 比较快速排序和基数排序。a）每个排序项平均执行指令数；b）每个排序项平均时间；c）每个排序项平均 cache 缺失数。数据来自于 LaMarca 和 Ladner 在 1996 年的一篇文章。由于这些结果，人们又发明了新版本的基数排序，将存储器层次结构考虑进来，以重新获得算法的优势（见 5.15 节）。cache 优化的基本思想是在某个块被替换前，重复使用该块中所有的数据

标准算法分析通常会忽视存储器层次结构的影响，正如更快的时钟频率和摩尔定律让体系结构设计者从指令流中获取所有的性能，合理地使用存储器层次结构是获得高性能的关键。如我们在概述中所说的，理解存储器层次结构的行为对于理解当今计算机的程序性能是十分关键的。

### 5.4.5 通过分块进行软件优化

由于存储器层次对程序性能具有非常重要的影响，因此许多软件优化技术通过对 cache 中的数据进行重用，提升了数据的时间局部性并因此降低缺失率，这些优化技术可以大大提高 cache 的性能。

在处理数组时，如果能够将数组元素按照访问顺序存放在存储器中，则能够获得很好性能。假定要处理多个数组，这些数组有些按行访问，而另外一些按列访问。因为在每次循环迭代中，既有按行访问的数组，又有按列访问的数组，因此无论采用按行存储或按列存储的方式都不能解决问题。

与对一个数组进行按行或按列操作不同，分块算法对子矩阵（或称为块）进行操作。其目标是在数据被替换出去之前，最大限度地对已装入 cache 的数据进行访问，即通过提升时间局部性的方法来降低 cache 缺失率。

例如，DGEMM 的内循环（第 3 章中图 3-21 中的 4～9 行）如下：

```
for (r= j = 0; j < r; ++j)
    {
      couble c1j = C[i+j*n]; /* c1j = C[i][j] */
      for( int k = 0; k < n; k++ )
       c1j += A[i+k*n] * B[k+j*n]; /* c1j += A[i][k]*B[k][j] */
      C[i+j*n] = c1j; /* C[i][j] = c1j */
      }
}
```

该程序段读取了数组 B 中的所有 N×N 个元素，另外反复读取了数组 A 中对应行中的 N 个元素，并且对数组 C 中对应行的 N 个元素进行了写操作。（注释使得矩阵的行列更容易识别。）图 5-20 给出了访问三个数组的大致情况。一个深色阴影表示一次最近访问的元素，浅色阴影表示早期访问的元素，而白色表示还没有被访问的元素。

412
↓
413

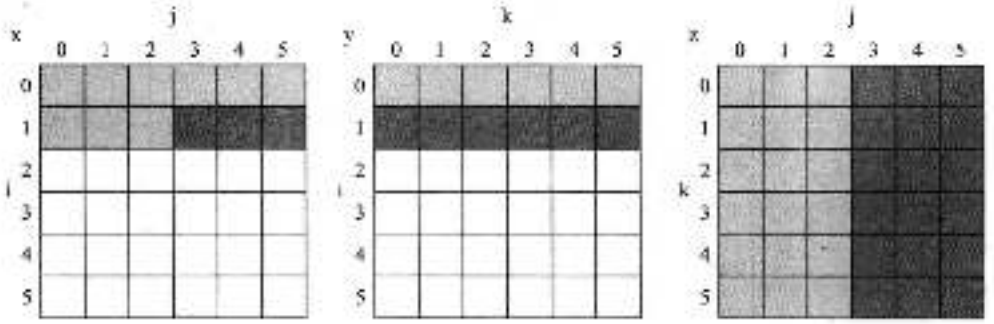



图 5-20 三个数组 C、A、B 的访问情况，N = 6，i = 1。数组元素的访问时间情况用阴影表示：白色表示该元素未被访问，浅色阴影表示早期访问，深色阴影表示新的访问。与图 5-21 相比，反复读取 A 和 B 的元素以计算新的元素 x。行列边上的变量 i、j 和 k 用于表示对数组的访问

很明显，缺失的次数依赖于 N 和 cache 的容量。如果在不发生冲突的前提下三个 N×N 矩阵的所有元素都在 cache 中，则没有任何问题。与第 3 章和第 4 章相同，假定 DGEMM 中的矩阵大小为 32×32。每个矩阵有 32×32 = 1 024 个元素，且每个元素为 8 字节。因此这三个矩阵将占据 24KiB 的空间，这意味着 Intel Core i7（Sandy Bridge）中 32KiB 的数据 cache 足够使用。

如果 cache 中能够保持一个 N×N 的矩阵和一个长度为 N 的行，则 A 的第 i 行和数组 B 可驻留在 cache 中。如果 cache 容量再小，将可能导致 B 和 C 访问都缺失。在最坏情况下，$N^3$ 次操作需要 $2N^3 + N^2$ 次存储器字的访问。

为了确保正在访问的元素能够在 cache 中命中，可把原先的程序改为每次循环迭代只计算一个子矩阵。因此，可通过参数 BLOCKSIZE 使得第 4 章图 4-80 中的 DGEMM 程序循环处理大小为 BLOCKSIZE 的数组，其中 BLOCKSIZE 称为分块因子（blocking factor）。

图 5-21 给出了 DGEMM 的分块版本。do_block 函数使用三个新的参数 si、sj 和 sk 表示每个子数组的起始位置。do_block 的内层循环以 BLOCKSIZE 为步进长度进行计算，而不是 n 和 C 的长度。gcc 优化器通过 "inling" 功能去除任何调用开销；也就是说，它将直接插入代码以避免传统的参数传递和返回地址的保存与恢复。

```
1  #define BLOCKSIZE 32
2  void do_block (int n, int si, int sj, int sk, double *A, double
3  *B, double *C)
4  {
5      for (int i = si; i < si+BLOCKSIZE; ++i)
6          for (int j = sj; j < sj+BLOCKSIZE; ++j)
7              {
8                  double cij = C[i+j*n];/* cij = C[i][j] */
9                  for( int k = sk; k < sk+BLOCKSIZE; k++ )
10                     cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11                 C[i+j*n] = cij;/* C[i][j] = cij */
12             }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17         for ( int si = 0; si < n; si += BLOCKSIZE )
18             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19                 do_block(n, si, sj, sk, A, B, C);
20 }
```

图 5-21　图 3-21 中 DGEMM 的 cache 分块版本。假定 C 初始化为 0。do_block 函数来源于第 3 章中的基本 DGEMM，使用了新参数来指明大小为 BLOCKSIZE 的子矩阵的起始位置。gcc 优化器通过内联的 do_block 函数消除函数开销

图 5-22 显示了使用分块之后对三个数组的访问情况。只考虑容量失效，访问存储器的总字数为 $2N^3/BLOCKSIZE + N^2$ 次，大约降低到原来的 1/BLOCKSIZE。分块技术同时利用了空间局部性和时间局部性，其中访问 A 时利用了空间局部性，访问 B 时则利用了时间局部性。 414

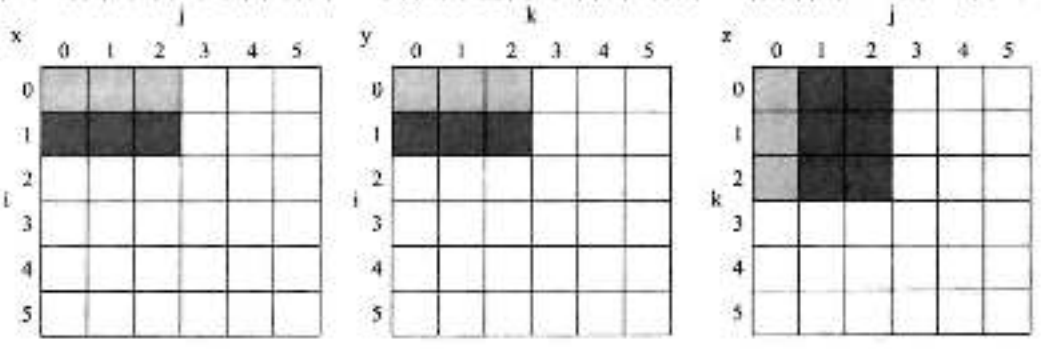



图 5-22　当 BLOCKSIZE = 3 时数组 C、A 和 B 的访问时间。注意，与图 5-20 相比，访问的元素减少了

虽然分块技术的目标是降低 cache 缺失率，但是分块技术也可用来帮助寄存器的分配。通过采用较小的分块，使得一个块可以驻留在寄存器中，程序中可以将 load 和 store 操作的数量大大减少，从而提高性能。

图 5-23 给出了采用分块技术的 DGEMM 的性能产生的影响，其中矩阵尺寸逐渐增加。可以看出，当矩阵尺寸增大到不能在 cache 中完全容纳这三个矩阵时，性能下降为最优情况的一半。即

使矩阵尺寸为960×960时（是第3章和第4章矩阵尺寸的900倍），性能仅下降了不到10%。

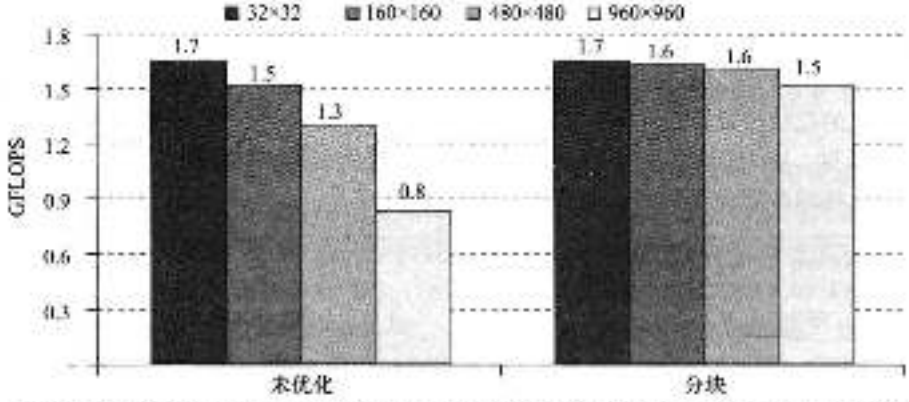



图5-23 未优化的DGEMM（图3-21）和采用cache分块技术的DGEMM（图5-21）的性能比较。在采用cache分块技术时，矩阵尺寸由32×32（三个矩阵均留在cache中）到960×960

**01 精解** 使用多级cache会产生一些复杂情况。首先，存在多种不同类型的缺失以及相应的缺失率。在"多级cache的性能"的例子中，我们看见了一级cache缺失率以及全局缺失率（global miss rate），即在所有级cache中都缺失的那部分访问。同时还有二级cache缺失率，是二级cache所有缺失次数和访问次数的比率。这个缺失率称为二级cache的局部缺失率（local miss rate）。由于一级cache过滤了一些访问，特别是那些具有较好的空间局部性和时间局部性的访问，这就使得二级cache的局部缺失率要大大高于全局缺失率。在"多级cache的性能"的例子中，可以计算出二级cache的局部缺失率为0.5%/2% =25%！幸运的是，全局缺失率决定了访问主存的次数。

- 全局缺失率：在多级cache的所有级中都缺失的那部分访问。
- 局部缺失率：在多级cache中，某一级cache的缺失率。

415 ~ 416

**01 精解** 乱序处理器（见第4章）在缺失时仍能执行指令，因而性能更加复杂。我们使用每条指令缺失数来代替指令缺失率和数据缺失率，公式如下：

存储器阻塞周期数/指令数 = 缺失数/指令数 ×（总的缺失延迟 − 重叠的缺失延迟）

计算重叠的缺失延迟没有通用的方法，因此对乱序处理器的存储器层次结构进行评估需要模拟处理器和存储器层次结构。只有观测到每次缺失时处理器的执行情况，我们才能知道缺失时处理器是阻塞下来等待数据还是在执行其他工作。一个指导方针是处理器通常会隐藏在一级cache缺失而在二级cache命中时的那部分缺失代价中，但是却很少隐藏二级cache的缺失代价。

**01 精解** 对算法性能的挑战在于：对相同的结构采用不同的实现方法，包括cache容量、相联度、块大小以及cache的数量，都会使得存储器层次结构变得多样化。为了应对这些变化，近来一些数值库将它们的算法变得参数化，通过实时搜索参数空间来找到特定计算机上的最佳组合。这种方法称为自动调节（autotuning）。

**01 小测验**

有关多级cache的设计，下面哪些是正确的？

1. 一级cache更关注命中时间，二级cache则更关注缺失率。
2. 一级cache更关注缺失率，二级cache则更关注命中时间。

### 5.4.6 小结

这一节集中讨论了4个问题：cache性能、利用相联度来降低缺失率、利用多级cache结构来降低缺失代价和采用软件优化技术提高cache的有效性。

存储系统对程序执行时间有着重要影响。存储器阻塞时钟周期数取决于缺失率和缺失代价。在5.8节中将会看到我们面临的挑战，就是如何降低这些因素中的一个而不会影响到存储器层次结构中的其他关键因素。

为了降低缺失率，我们对相联定位方法进行研究。这种方法通过将数据块更灵活地放置在cache中以降低缺失率。全相联机制允许将块放在cache中的任何位置，但是仍然需要查找cache中的每一块以找到所需的数据块。较高的成本使得大容量的全相联cache的实现不切实际。而组相联cache则更加可行，我们只需要在索引唯一选中的组中进行查找。组相联cache缺失率更高，但是访问速度更快。使用何种相联度能达到最佳性能不仅取决于技术本身，还取决于实现的细节。

我们探讨了多级cache技术，它通过使用一个大的二级cache来处理一级cache的缺失，从而降低了缺失代价。二级cache已经逐渐普遍，这是因为设计者发现由于硅的局限以及高时钟频率的要求，一级cache的容量已经无法更大了。而二级cache的容量通常是一级cache的10倍甚至更多，因而能处理很多一级cache缺失引起的访问。在这些情况下，缺失代价就是二级cache的访问时间（通常小于10个处理器周期）而不是主存访问时间（通常大于100个处理器周期）。和相联度考虑相似，在二级cache容量和访问时间之间的权衡取决于实现过程中的很多方面。 417

最后，针对存储器层次对性能影响的重要性，我们讨论了如何对算法进行变换来提高cache性能，主要讨论了针对大数组进行分块的技术。

## 5.5 可信存储器层次

本章前面所有的讨论集中在如何提高存储器层次的性能上，但是不能忘记如果可信性不够，即使速度再快也不具有吸引力。正如在第1章所述，最好的可信性方法是冗余技术。本节将首先重温与失效相关的术语定义和度量，然后将讲述如何采用冗余技术构造可靠的存储器。

### 5.5.1 失效的定义

假定有某类服务的需求，针对该需求，用户可以看到一个系统在两种服务状态之间做出选择：

1）服务实现：交付的服务与需求相符。

2）服务中断：交付的服务与需求不符。

失效导致状态1到状态2的转换，而由状态2转换到状态1的过程称为恢复。失效可以是永久性的，也可以是间歇性的。间歇性失效更加复杂一些，因为当一个系统因间歇性失效在两个状态间摇摆时，诊断将会非常困难。而永久性失效的诊断要容易许多。

这种定义将引出两个术语：可靠性和可用性。

可靠性是一个系统或模块能够持续提供用户需求的服务的度量，即从开始使用到失效的时间间隔。因此，平均无故障时间（mean time to failure，MTTF）是一个可靠性度量方法。与之相关的一个术语是年失效率（annual failure rate，AFR），它是指在给定MTTF情况下，在一年内预期的器件失效比例。由于从MTTF中可能会得到误导性的结果，因此AFR会获得更加直观的结果。

**例题：磁盘的 MTTF 和 AFR**

当今的一些磁盘号称其 MTTF 为 1 000 000 小时，大约是 1 000 000/(365 × 24) = 114 年，这意味着这些磁盘永远不会失效。运行搜索引擎等网络服务的仓储式计算机可能有 50 000 台服务
418 器，假定每台服务器有两块磁盘，使用 AFR 计算每年将会有多少块磁盘失效。

**答案**

一年有 365 × 24 = 8 760 小时。1 000 000 小时的 MTTF 意味着 AFR 为 8 760/1 000 000 = 0.876%。由于系统中有 100 000 块磁盘，因此每年将有 876 块磁盘失效，即平均每天有超过两块磁盘失效！ □

服务中断使用修复平均时间（mean time to repair，MTTR）来度量。失效间隔平均时间（mean time between failure，MTBF）= MTTF + MTTR。虽然 MTBF 广泛应用，MTTF 却更加确切。可用性是指系统正常工作时间在连续两次服务中断间隔时间中所占的比例：

$$\text{可用性} = \text{MTTF}/(\text{MTTF} + \text{MTTR})$$

需要注意的是，可靠性和可用性是可以量化的，而可信性是不可量化的。与增加 MTTF 类似，减少 MTTR 同样可以提高可用性。例如，可以采用故障检测、诊断和修复的工具来减少故障维修花费的时间，从而提高可用性。

我们希望系统具有很高的可用性。一种简化的表示方法是“每年中可用性的 9 的数量”。例如，一个很好的网络服务器可提供 4 个或 5 个 9 的可用性。一年有 365 × 24 × 60 = 526 000 分钟，简化表示如下：

1 个 9：90%　　=>　36.5 天的维修时间/年
2 个 9：99%　　=>　3.65 天的维修时间/年
3 个 9：99.9%　=>　526 分钟的维修时间/年
4 个 9：99.99%　=>　52.6 分钟的维修时间/年
5 个 9：99.999%　=>　5.26 分钟的维修时间/年

以此类推。

为了提高 MTTF，可以提高器件的质量，也可以设计能够在器件失效的情况下继续工作的系统。由于一个器件的失效可能不会导致系统的失效，因此需要根据具体情况定义失效。为了明确差别，使用术语故障来表示一个器件的失效。有如下三种方式可以提高系统的MTTF：

1）故障避免技术（fault avoidance）：通过合理构建系统来避免故障的出现。

2）故障容忍技术（fault tolerance）：采用冗余措施，当发生故障时，通过冗余措施保证系统仍然正常工作。

419 3）故障预报技术（fault forecasting）：对故障进行预测，从而允许在器件失效前进行替换。

## 5.5.2 纠正一位错、检测两位错的汉明编码（SEC/DED）

理查德·汉明（Richard Hamming）发明了一种广泛应用于存储器的冗余技术，并因此获得 1968 年的图灵奖。二进制数间的距离对于理解冗余码很有帮助。两个等长二进制数的汉明距离是两个数对应位置不同的位的数量。例如，011011 和 001111 的距离为 2。在一种编码中，如果码字之间的最小距离为 2，且其中有 1 位错误，将会发生什么？这将会将一个有效的码字转化为无效码字。因此，如果能够检测出一个码字是否有效，则可检测出 1 位的错误，称为 1 位**错误检测编码**。

- 错误检测编码：这种编码方式能够检测出数据中有 1 位错误，但是不能对错误位置进行精确定位，因此不能纠正错误。

汉明使用奇偶校验码进行错误检测。在奇偶校验码中，计算码字中 1 的数量是奇数个还是偶数个。当一个字写入存储器时，奇偶校验位也被写入（1 代表奇数，0 代表偶数）。这就是说，$N+1$ 位码字中 1 的个数永远为偶数。因此，当读出数据时，校验码也被读出并进行检测。如果计算出的校验码与保存的不符，则发生错误。

**例题**

计算十进制数 31 对应的 8 位二进制数的奇偶性，并写出存储器中的表示形式。假设奇偶校验位在最右边，并且假定存储器中最高位发生了翻转，然后将其读回。请问能否检测到错误？如果最高两位同时翻转呢？

**答案**

十进制数 31 的二进制形式为 00011111，有 5 个 1。为了使编码后的码字为偶性，需要向校验位写入 1，也就是 000111111。如果最高位发生翻转，读回的将是 100111111，具有 7 个 1。由于期望码字为偶性，但是计算结果却是奇性，因此报告出错。如果最高两位同时发生翻转，则得到110111111，具有 8 个 1 或者说具有偶性，因此不能报告出错。 □

如果有两位同时出错，该错形下码字的奇偶性不变，因此一位奇偶校验无法检测到该错误。（实际上，一位奇偶校验可以检测到任意奇数个错误，但实际情况是，出现三位错的概率远小于出现两位错的概率，所以一位奇偶校验码主要用于检测 1 位出错。）

当然，奇偶校验码不能纠正错误，汉明想要做到检错的同时又能纠错。如果我们采用最小距离为 3 的码组，那么任意一个发生 1 位错的码字与其对应的合法码之间的距离要小于该非法码与其他合法码字的距离。汉明提出了一种易于理解的映射方法，该方法将数据映射到距离为 3 的码字，为了表达对他的敬意，我们称这种编码方法为汉明纠错码（Hamming Error Correction Code，ECC）。我们采用额外的校验位确定单个错误的位置。下面是计算汉明纠错码的步骤： 420

1）对数据部分从左到右由 1 开始依次编号，这跟通常采用的从最右边开始由 0 开始编号的做法相反。

2）将所有编号为 2 的整数次幂的位标记为奇偶校验位（1，2，4，8，16，…）。

3）其他剩余位置用作数据位（位置 3，5，6，7，9，10，11，12，13，14，15，…）。

4）奇偶校验位的位置决定了其对应的数据位（图 5-24 用图形的方式进行了说明）如下所示：

| 位置 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 编码之后的数据位 | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| 奇偶校验位覆盖范围 | p1 | X | | X | | X | | X | | X | | X | |
| | p2 | | X | X | | | X | X | | | X | X | |
| | p4 | | | | X | X | X | X | | | | | X |
| | p8 | | | | | | | | X | X | X | X | X |

图 5-24 用于 8 位数据的汉明纠错码，其中奇偶校验位、数据位及覆盖范围如图所示

- 校验位 1（$0001_2$）检查第 1，3，5，7，9，11，…位，这些数位编号的二进制形式最右边一位均为 1（$0001_2$，$0011_2$，$0101_2$，$0111_2$，$1001_2$，$1011_2$，…）。

- 校验位2（$0010_2$）检查第2，3，6，7，10，11，14，15，…位，这些数位编号的二进制形式右边起第二位均为1。
- 校验位4（$0100_2$）检查第4～7，12～15，20～23，…位，这些数位编号的二进制形式右边起第三位均为1。
- 校验位8（$1000_2$）检查第8～15，24～31，40～47，…位，这些数位编号的二进制形式右边起第四位均为1。

注意到每个数据位都被至少两个奇偶校验位覆盖。

5）设置奇偶校验位，对各组进行偶校验。

如同变魔术一样，你可以通过查看校验位来确定数据位是否出错。采用图5-24当中的12位码字，如果4个校验位组成的二进制数（$p_8$，$p_4$，$p_2$，$p_1$）是$0000_2$，这说明没有发生错误。但是，如果校验位组成的二进制数为$1010_2$，也就是十进制数10时，汉明纠错码告诉我们第十
421 位（d6）出错了。由于是二进制数，所以只需将第十位的数进行取反，就完成了纠错。

**例题**

假定存在某个单字节数据$10011010_2$。首先写出对应的汉明纠错码，然后把第10位取反，说明纠错码如何找到并纠正该错误。

**答案**

将校验位的位置空出来，12位的码字 _ _ 1 _ 001 _ 1010。

位置1检查第1，3，5，7，9，11位，为使该组为偶校验，我们应当把第1位填0。

位置2检查第2，3，6，7，10，11位，为使该组为偶校验，我们在第2位填入1。

位置4检查第4，5，6，7，12位，所以我们在第4位填入1。

位置8检查第8，9，10，11，12，所以我们在第8位填入0。

最终得到的码字为011100101010。把数据位第10位取反之后变成011100101110。

校验位1为0（**0**1**1**10**0**10**1**11**0**有4个1，为偶性，故该组无错误）。

校验位2为1（0**1**1**1**00**1**01**1**10有5个1，为奇性，故该组某个位置上有错误）。

校验位4为1（011**1**00**1**01**1**10有两个1，为偶性，故该组无错误）。

校验位8为1（0111001**1**01**1**10有3个1，为奇性，故该组某个位置上有错误）。

校验位2和10不正确。因为2＋8＝10，第10位肯定是错的。因此，我们将其翻转为011100101010，即完成了纠错。 □

汉明并没有止步于1位纠错码。通过付出增加一位的代价，可以让码组中的最小汉明距离变到4。这就意味着我们可以做到纠正1位错并检测2位错。该方法增加了1位奇偶校验码，对整个字进行计算校验。这里我们以4位的数据字为例，这只需要7位就能完成单位错检测。计算出汉明奇偶校验位H（$p_1$ $p_2$ $p_3$），这里仍然采用偶校验，最后计算当整个字的偶校验位$p_4$：

1 2 3 4 5 6 7 8

$p_1$ $p_2$ $d_1$ $p_3$ $d_2$ $d_3$ $d_4$ $\underline{p_4}$

上述用于纠正1位错同时检测2位错的算法仍像之前那样先计算出纠错码组的奇偶性H，最后再计算一下全组的奇偶校验位$p_4$就可以了。以下是可能出现的4种情况：

1）H为偶并且$p_4$为偶，这表示没有错误发生。

2）H为奇并且$p_4$为奇，这表明出现了一位可纠正错误。（当出现一位错时，$p_4$应当为奇。）

422 3）H为偶并且$p_4$为奇，这说明出现的仅仅是$p_4$，因此将$p_4$取反即可。

4）H为奇并且$p_4$为偶，这表示出现了两位错。（当出现两位错时，$p_4$应当为偶。）

纠1位错检2位错（SEC/DED）的技术现在广泛应用于服务器的内存。方便的是，8字节的数据块做SEC/DED时只需要给好一个字节的额外开销。这也是为什么许多双列直插式存储模块宽度为72位。

**5.1 精解** 为了计算出SEC需要的位数，假定$p$表示校验位的位数，$d$表示数据位的位数，则整个码字为$p+d$位。如果采用$p$个纠错位指示错误（码字长度为$p+d$的情况下），再加上没有出现错误的情况，不难得到下面的不等式：

$$2^p \geqslant p+d+1 \text{ 位}, \quad \text{因此 } p \geqslant \log(p+d+1)$$

例如，对8位的数据而言，$d=8$，并且$2^p \geqslant p+8+1$，所以$p=4$。类似的，数据长度为16位时$p=5$，32位时$p=6$，64位时$p=7$，以此类推。

**5.1 精解** 在大型系统中，出现多位错的概率和整个内存芯片出错的概率变得显著起来。为解决这一问题，IBM引进了一对叫作chipkill的技术，之后很多大型系统都应用了该项技术（Intel称他们所用的为SDDC）。chipkill本质上类似于磁盘阵列中采用的RAID技术（见5.11节），将数据和校验码分散开来，因此当某一内存芯片全部出错时，可以通过其他内存芯片中的内容对出错的内容进行重建。假定现有一个由10 000个处理器构成的集群，其中每个处理器配备4GiB内存，IBM针对为期三年的运行时间计算出了以下不可恢复内存错误出现的比率：

- 仅采用奇偶校验——出现大约90 000次不可恢复（或者不可检测）错误，换句话说，每17分钟就出现一次。
- 仅采用SEC/DED——出错大约3 500次，换句话说，每7.5小时出现一次不可检测或者不可恢复的错误。
- 采用chipkill——出错6次，换句话说，每两个月出现一次不可检测或不可恢复的错误。

因此，基于数据仓库级别的计算机需要采用chipkill技术。

**5.1 精解** 虽然存储器系统出现1位错或者2位错的情况比较典型，但是网络系统中可能会出现突发型错误。解决该问题的一个方法是采用循环冗余校验。对于一个具有$k$位的字块来说，发送端生成一个$n-k$位长度的帧校验序列。这样最终发送出的是一个长度为$n$位的序列，并且该序列构成的数字可以被某个数整除。接收端用那个数去除接收到的帧。如果余数为0，就认为没有发生错误。如果余数不为0，接收端将收到的消息丢弃，并通知发送端重新发送。从第3章你不难看到，对于某些二进制数，可以利用移位寄存器方便地完成除法运算，这使得即便在硬件价格十分昂贵的时代，CRC校验码也可以被广为采用。更进一步说，里德－索罗蒙（Reed-Solomon）编码使用伽罗瓦（Galois）域来纠正多位传输错误，数据被看作是多项式系数，校验码被看作是多项式的值。里德－索罗蒙计算复杂度远远高于二进制除法运算！ 423

## 5.6 虚拟机

虚拟机（Virtual Machine，VM）最早出现于20世纪60年代中期，这些年来一直是大型机中的重要组成部分。尽管在20世纪80年代和90年代期间，它们大多被单用户计算机领域所忽略，但是最近才受到人们的关注，这是因为

- 在现代计算机系统中，隔离和安全的重要性在增长。
- 标准操作系统在安全性和可靠性方面存在缺陷。
- 在多个不相关的用户间共享一台计算机，尤其是在云计算中。

• 在过去10年里，处理器速度大幅增长，使得虚拟机引起的开销降至可接受的范围内。

最广泛的虚拟机的定义包括所有基本的仿真方法。这些方法提供一个标准的软件接口，如Java虚拟机。在这一节中，我们对虚拟机感兴趣的地方在于，在二进制指令集系统结构（ISA）的层次上提供一个完整的系统级环境。尽管一些虚拟机在本地硬件上运行不同的指令集系统结构，但我们假设它们都能与硬件匹配。这样的虚拟机被称为（操作）系统虚拟机（system virtual machine），如IBM VM/370、VirtualBox、VMware ESX Server以及Xen。

系统虚拟机让用户觉得自己在使用包括操作系统的副本在内的整个计算机。一台运行多个虚拟机的计算机可以支持多个不同的操作系统。在一个传统的平台上，一个单独的操作系统拥有所有的硬件资源，但是通过使用虚拟机，多个操作系统共享硬件资源。

支持虚拟机的软件被称为虚拟机监视器（virtual machine monitor，VMM）或者管理程序（hypervisor）；VMM是虚拟机技术的核心。底层的硬件平台称为主机（host），它的资源被客户端（guest）虚拟机共享。VMM决定如何将虚拟资源映射到物理资源；物理资源可能是分时共享、划分，甚至通过软件模拟的。VMM比传统的操作系统小很多；一个VMM的隔离区可能只需要10 000行代码。

尽管我们所感兴趣的是虚拟机可以提供保护功能，但是在商业意义上，虚拟机还有其他两个优势：

1）软件管理：虚拟机提供一个可以运行完整软件堆的抽象，甚至包含像DOS这样的旧操作系统。虚拟机典型的调度包括：一些虚拟机运行旧的操作系统，多数虚拟机运行当前的操作系统，少数虚拟机用来测试下一代操作系统版本。

2）硬件管理：需要多个服务器的一个原因是为了让每个应用程序运行在一台单独的计
424 算机上，并拥有与之兼容的操作系统，这样的分隔能改善可靠性。虚拟机允许这些独立的软件堆能在共享硬件的同时独立运行，因此合并了服务器的数量。另一个例子是，一些VMM支持将正在运行的虚拟机移植到另一台计算机上，这样可以平衡负载或在硬件故障时实施迁移。

**旧1 硬件/软件接口** 亚马逊Web服务（AWS）在其云计算平台中使用虚拟机提供EC2主要有5个原因：

1）在用户共享同一个服务器时，AWS允许可提供用户间的保护。

2）它简化了仓储式计算机上软件的分布。用户只需要安装一个虚拟机映象，并配置合适的软件，AWS为用户分配并所需的所有服务。

3）当用户完成工作后，用户（和AWS）可以“系统”一个VM来控制资源的使用。

4）虚拟机隐藏了运行用户应用软件的硬件特性，这意味着AWS可以连续使用老的服务器并引入新的更有效的服务器。用户希望获得的机器性能与“EC2计算单元”能够匹配，AWS将其定义为：提供与1.0～1.2 GHz 2007 AMD Opteron或2007 Intel Xeon处理器相等的CPU性为。依据摩尔定律，很明显，新的服务器能够提供更多的EC2计算单元，但是出于经济型的考虑，AWS仍然能够运行老旧服务器。

5）虚拟机监视器可以控制一个VM使用处理器、网络和磁盘的比率，这就使得AWS可以在相同的底层硬件上提供许多价格不同的节点类型。例如，2012年时AWS提供14种不同类型的节点：从0.08美元/小时的小标准节点到3.10美元/小时的高I/O性能节点。

通常来说，处理器虚拟化的开销取决于工作量。用户级处理器限制型程序没有虚拟化开销，这是因为操作系统很少被调用，因此所有的程序都能以本来速度运行。I/O密集型负载通常也是操作系统密集型的，它们会执行许多系统调用和特权指令，从而导致很高的虚拟化开

销。另一方面，如果 I/O 密集型负载同样也是 I/O 限制型的，由于在等待 I/O 时，处理器通常处于空闲状态，因此处理器的虚拟化开销就完全能被掩藏。

开销取决于需要由 VMM 进行模拟的指令数目以及模拟速度的快慢。因此，假设客户端虚拟机和主机运行同样的 ISA 时，系统结构和 VMM 的目标是尽可能在本地硬件上运行所有指令。 425

## 5.6.1 虚拟机监视器的必备条件

虚拟机监视器需要做什么？它给客户软件提供了一个软件接口，分开每个客户端的状态，并且需要将自己从客户端软件中（包括客户操作系统）隔离。定性的需求是：

- 除了性能相关的行为或因多虚拟机共享而造成的固定资源限制以外，客户软件在虚拟机上的运行应该和它在本地硬件上的运行完全相同。
- 客户软件不能直接改变实际系统中的资源分配。

为了对处理器进行“虚拟化”，VMM 必须能控制一切——访问特权状态、I/O、异常和中断——尽管客户虚拟机和当前运行的操作系统临时使用它们也不受影响。

例如，在定时器中断的情况下，VMM 需要挂起当前正在运行的客户虚拟机，保存其状态，处理中断，然后决定下面该运行哪个客户虚拟机，并读取其状态。依赖定时器中断的客户虚拟机会有由 VMM 提供的一个虚拟定时器和模拟的定时器中断。

为了方便管理，VMM 必须运行在一个比用户虚拟机更高的特权级别下，其中，用户虚拟机通常运行在用户模式下，这也确保了任何特权指令的执行都需要由 VMM 来处理。系统级虚拟机的基本必备条件如下：

- 至少两个处理器模式，系统级和用户级。
- 特权级指令集合只能在系统模式下使用，如果在用户模式下执行将会产生 trap 中断；所有系统资源只能由这些指令控制。

## 5.6.2 指令集系统结构（缺乏）对虚拟机的支持

如果在 ISA 设计过程中考虑到了虚拟机的使用，那么由 VMM 执行的指令数目和模拟这些指令所花费的时间会相对减少些。允许虚拟机直接在硬件上执行的系统结构被冠以可虚拟化（virtualizable）的名称，IBM 370 就是如此。

由于虚拟机只是近期才考虑应用于桌面系统和基于 PC 的服务器，所以大部分指令集在创建时都没有考虑虚拟化的思想。x86 和大部分 RISC 系统结构，包括 ARM v7 和 MIPS 都是如此。 426

VMM 必须保证客户系统只能和虚拟资源交互，因此常规的客户操作系统在 VMM 的顶层运行用户模式程序。如果客户操作系统试图通过特权指令访问或者修改相关硬件资源的信息——例如，读/写一个状态位来启动中断——它会向 VMM 发出 trap 中断。VMM 会进行适当的调整来对应实际资源。

因此，如果任何指令试图在用户模式下读/写这样敏感的信息 trap，VMM 将截获它并且如客户操作系统所需的那样，支持敏感信息的虚拟版本。

如果上述条件不具备，那么需要其他的方法。VMM 必须使用特殊的预防措施来定位所有存在问题的指令，并且确保它们能被客户操作系统正确执行，这就增加了 VMM 的复杂度，同时也降低了虚拟机的运行性能。

## 5.6.3 保护和指令集系统结构

保护需要同时依赖于体系结构和操作系统，但是随着虚拟存储器的广泛使用，体系结构设

计者需要对指令集体系结构中一些不方便使用的细节进行修改。

例如，x86 的指令 POPF 从存储器堆栈的顶部加载标记寄存器。其中有一个标记是中断使能标记位（IE）。如果在用户模式下运行 POPF，它只是简单地改变除了 IE 位以外的所有标记位，而不是发生 trap 中断。如果在系统模式下，它确实会改变 IE 位。但是有一个问题，运行在虚拟机用户模式下的客户操作系统希望看见 IE 位的改变。

在过去，IBM 的大型机硬件和 VMM 采用以下三个步骤来改善虚拟机的性能：

1）降低处理器虚拟化的开销。

2）降低由虚拟化引起的中断开销。

3）中断发生时交给相应的虚拟机，而不用调用 VMM，从而降低中断开销。

在 2006 年，AMD 和 Intel 提出新的计划尽力满足第一个要点，即降低处理器虚拟化的开销。体系结构和 VMM 需要经过多少代的改进才能完全满足上面三点？21 世纪的虚拟机需要经过多长时间才能像 20 世纪 70 年代的 IBM 大型机和 VMM 一样有效呢？这些都是令人感兴趣的研究。

## 5.7 虚拟存储器

> 已经设计出这样的系统：对程序员来说，复合的存储结构看起来像单层的存储器，所需的数据传输也会自动完成。
>
> ——Kilburn 等，《One-level storage system》，1962

在前面的章节中，我们知道了 cache 是如何对程序中最近访问的代码和数据提供快速访问的。同样，主存也可以为通常由磁盘实现的辅助存储器充当“cache”。这项技术被称作**虚拟存储器**（virtual memory）。从历史观点来说，构造虚拟存储器有两个主要动机：允许云计算在多个虚拟机之间有效而安全地共享存储器；消除一个小而受限的主存容量对程序设计造成的影响。50 年后，第一条变成主要设计动机。

427

⊃ 虚拟存储器：一种将主存用作辅助存储器高速缓存的技术。

当然，为了允许多个虚拟机共享同一个存储器，我们必须在虚拟机之间进行保护，确保每个程序只能对划分给它的那部分主存进行读写操作。主存只需存放众多程序中活跃的那部分，就像 cache 中只存放一个程序的活跃部分一样。因此，局部性原理也同样适用于虚拟存储器，虚拟存储器使得我们能更有效地共享处理器和主存。

在编译的时候，我们不知道哪些虚拟机将和其他虚拟机共享存储器。事实上，当虚拟机在执行的时候，虚拟机共享存储器的情况是动态变化的。由于这种动态的相互影响，我们希望将每个程序都编译到它自己的地址空间（address space）——存储器中只能有该程序访问的独立的一些串地址。虚拟存储器实现程序地址空间到**物理地址**（physical address）的转换。这种地址转换处理加强了各个程序地址空间之间的**保护**（protection）。

⊃ 物理地址：主存储器的地址。

⊃ 保护：一组确保共享处理器、主存、I/O 设备的多个进程之间没有故意地、无意地读写其他进程的数据机制，这些保护机制可以将操作系统和用户的进程隔离开来。

使用虚拟存储器的第二个动机就是允许单用户程序使用超过主要存储器的容量。以前，如果一个程序对存储器来说太大，将它变成合适的大小就是程序员的责任。程序员将程序划分成许多段，并且将这些段标记成为互斥的。这些程序段（overlay）在执行过程中由用户程序控制装入或换出，由程序员保证程序不会访问没有装载的程序段，并且装载的程序段不会超过存储

器的总容量。传统的程序段被构造成模块，每一个都包含了代码和数据。不同模块之间的过程调用将导致一个模块覆盖掉另一个模块。

可以想象，这种责任对程序员来说是很大的负担。虚拟存储器的发明就是为了将程序员从这些困境中解脱出来，它自动管理由上存（为了区别虚拟存储器，有时也称为物理存储器）和辅助存储器组成的两级存储器层次结构。

尽管虚拟存储器和 cache 的工作原理是一样的，但是不同的历史根源决定它们要使用不同的术语。虚拟存储器中，块被称为页（page），访问缺失则被称为缺页（page fault）。在虚拟存储器中，处理器产生一个虚拟地址（virtual address），再结合软硬件转换成一个物理地址（physical address），然后就可以被用来访问主存了。图 5-25 显示了一个分页的虚拟寻址的存储器被映射到主存中。这个过程被称作地址映射或者地址转换（address translation）。如今，个人 428
移动设备中由虚拟存储器控制的两级存储器层次结构是 DRAM 和闪存，而在服务器中则是 DRAM 和磁盘（见 5.2 节）。如果还拿图书馆作类比，我们可以认为一本书的书名就是虚拟地址，物理地址则是这本书在图书馆中的位置，它可能是图书馆的索书号。

- 缺页：访问的页不在主存储器中。
- 虚拟地址：虚拟空间的地址。当需要访问主存时需要通过地址映射转换为物理地址。
- 地址转换：也称为地址映射。在访问内存时将虚拟地址映射为物理地址的过程。

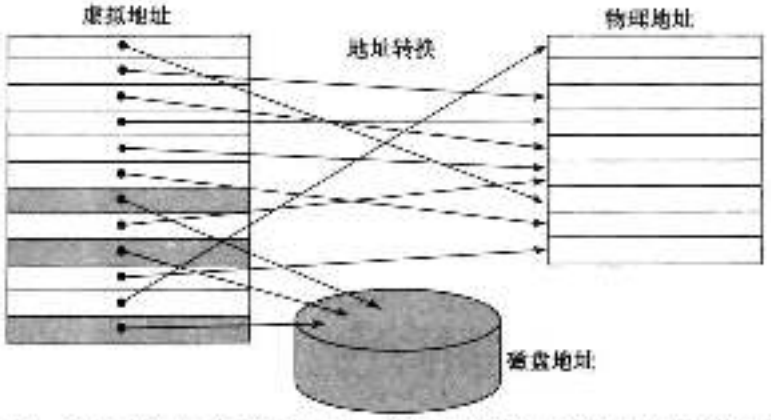



图 5-25 在存储器中，主存中的块（称为页）从一组地址（称为虚拟地址）映射到另一组地址（称为物理地址）。访问主存使用物理地址，而处理器产生虚拟地址。虚拟存储器和物理存储器都被划分成页，因此一个虚页被映射到一个物理页。当然，一个虚页也可能不在主存中，因此无法映射到物理地址；在这种情况下，页就被存在磁盘上。物理页可以被两个指向相同物理地址的虚拟地址共享。这种方法用来使两个不同的程序共享数据或代码

虚拟存储器还提供重定位（relocation）来简化执行时的程序加载过程。在用地址访存之前，重定位将程序所用的虚拟地址映射到不同的物理地址。重定位的方法允许我们将程序加载到主存中的任何位置。另外，现今所有的虚拟存储器系统将程序重定位成一组固定大小的块（页），因此减少了寻找主存中连续的块来放置程序的必要；取而代之的是，操作系统只需要在主存中找到足够数量的页。

在虚拟存储器中，地址被划分为虚页号（virtual page number）和页偏移（page offset）。图 5-26 所示是虚页号到物理页号（physical page number）的转换。物理页号构成物理地址的高位部分，而页偏移是不变的，构成物理地址的低位部分。页偏移域的位数决定了页的大小。虚拟地址可寻址的页数与物理地址可寻址的页数可以不同。拥有比物理页数多得多的虚页数是描述一个没有容量限制的虚拟存储器的假象的基础。 429

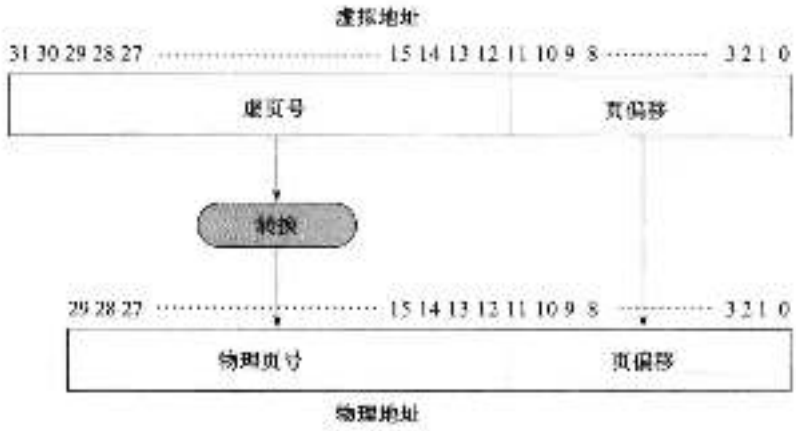



图 5-26 虚拟地址到物理地址的映射。页大小为 $2^{12}$ = 4KiB。由于物理页号有 18 位，存储器中物理页数为 $2^{18}$。因此，最多可以支持 1GiB 的主存，而虚拟地址空间为 4GiB

页缺失引发的高代价是许多设计选择虚拟存储系统的原因，缺失在虚拟存储器中通常称为缺页。一次缺页处理将花费数百万个时钟周期（5.2 节的表指出了主存储器大概比磁盘快 100 000 倍）。这一巨大的缺失代价，主要由取得标准大小的页中第一个字所需的时间来确定，因此在设计虚拟存储系统时需要考虑一些关键性的因素：

- 为了弥补较长的访问时间，页应该足够大。目前典型的页大小从 4 ~ 16KiB。能支持 32 ~ 64KiB 页的新型台式计算机和服务器正进行研发，但是新的嵌入式系统走的是相反的方向，页大小为 1KiB。
- 能降低缺页率的组织结构具有吸引力。这里用到的主要技术是允许存储器中的页以全相联方式放置。
- 缺页可以用软件处理，这是因为与访问磁盘的时间相比，这样的开销不算大。此外，软件可以使用一些更先进的算法来选择替换页，只要缺失率减少很小一部分就足以弥补算法的开销。
- 由于写时间太长，因此在虚拟存储器中，写直达机制不能很好地管理写操作。因此虚拟存储系统中都采用写回机制。

430

下面几节将把这些因素融入虚拟存储器的设计中去。

中1 **精解** 我们引入虚拟存储器是由于许多虚拟机器共享存储器，但是，虚拟存储器发明的最初原因是在分时系统中，许多程序可以共享一台计算机。由于当今的许多读者没有使用分时系统的经验，本节使用虚拟机作为引入共享存储器的动机。

中1 **精解** 对台式机和服务器来说，32 位地址的处理器已经很有问题了。通常我们认为虚拟地址要远大于物理地址，但是如果相对于存储技术，处理器地址字较小的话，相反的情况也会出现。单个程序或虚拟机不会受害，但是一组程序或虚拟机同时执行就可能因无法交换到主存，或者在单个处理器上执行而受害。

中1 **精解** 本书对虚拟存储器的讨论主要集中于使用固定大小的块的页式虚拟存储。还有一种可变长度块的机制称为段式管理（segmentation）。在段式存储中，地址由两部分组成：段号和段内偏移。段号被映射到物理地址，然后与段内偏移量相加来找到实际物理地址。因为段大小是可变的，所以还需要进行边界检查以确定偏移量是否在段内。分段最主要的应用就是支持更多有效的保护方法，以及共享地址空间。与分页相比，大多数操作系统的教科书都会更多地讨论分段，以及如何利用分段来逻辑共享地址空间。分段的主要缺点在于

它将地址空间划分为许多逻辑上独立的块，因而这些块就由两部分地址控制：段号和段内偏移。相反，分页使得页号和偏移量的界限对于程序员和编译器都是不可见的。

分段也曾被用作不改变计算机字的大小而扩展地址空间的方法。然而这些尝试都没有获得成功，这是由于程序员和编译器必须意识到使用两部分地址本身的不便和性能代价。

许多体系结构将地址空间划分成固定大小的大块以简化操作系统和用户程序之间的保护，同时提高分页实现的效率。尽管这些划分通常称为“段”，但是这种结构比块大小可变的分段要简单得多，并且对用户程序不可见。稍后我们对此进行详细讨论。

⊃ 段式管理：一种可变长度的地址映射策略，其中每个地址由两部分组成：映射到物理地址的段号和段内偏移。

### 5.7.1 页的存放和查找

由于缺页的代价尚得惊人，设计人员通过对页的放置进行优化从而降低缺页频率。如果允许一个虚拟页映射到任何一个物理页，那么当缺页发生时，操作系统可以选择任意一个页进行替换。例如，操作系统可以使用复杂的算法和复杂的数据结构来跟踪页的使用情况以选择在较长一段时间内不会被用到的页。使用更先进更灵活的替换策略降低了缺页率，也使全相联方式下页的放置变得更简单。 431

正如在5.4节中提到的，全相联映射的困难在于项的定位，这是由于它可能在较高的存储器层次中的任何位置。全部进行检索是不切实际的。在虚拟存储系统中，我们使用一个索引存储器的表来定位页；这种结构称为页表（page table），它被存放在存储器中。页表使用虚拟地址中的页号作索引，以找到相应的物理页号。每个程序都有它自己的页表，用来将程序的虚拟地址空间映射到主存中。让我们再用图书馆进行类比，页表对应于书名和藏书位置之间的映射。就像卡片目录可能会包含学校中另一个图书馆中书的条目，而不仅仅是本地的分馆，我们将看见页表也可能含有不在存储器中的页的条目。为了指出页表在存储器中的位置，硬件包含一个指向页表首地址的寄存器；我们称之为页表寄存器（page table register）。现在假定页表存在存储器中一个固定的连续区域内。

⊃ 页表：保存着虚拟地址和物理地址之间转换关系的表。页表保存在主存中，通常使用虚页号来索引，如果这个虚页当前在主存中，页表中的对应项将包含虚页对应的物理页号。

**硬件/软件接口** 页表、程序计数器以及寄存器，确定了一个虚拟机的状态（state）。如果我们想让另一个虚拟机使用处理器，我们必须保存该状态。随后，在恢复了该状态之后，虚拟机就可以继续执行。我们通常称该状态为一个进程（process）。如果一个进程占据了处理器，那么这个进程就是活跃的（active），否则就认为它是非活跃的（inactive）。操作系统可以通过加载进程的状态令一个进程活跃起来，同时激活程序计数器，进程将会在程序计数器中保存的值处开始执行。

进程的地址空间，以及它在主存中可以访问的所有数据，都由驻在主存中的页表所定义。操作系统只是简单地加载页表寄存器用来指向它想激活的进程的页表，而不是保存整个页表。由于不同进程使用相同的虚拟地址，因此每个进程有各自的页表。操作系统负责分配物理主存和更新页表，因此不同进程的虚拟地址空间不会发生冲突。我们很快会看到，使用分离的页表同样能分别保护进程。 432

图5-27使用页表寄存器、虚拟地址以及被指向的页表来说明硬件是如何形成物理地址的。每个页表项使用1位有效位，就像在cache中设计的一样。如果该位为无效，该页就不在主存中，就发生一次缺页。如果该位为有效，表明该页在主存中，并且该项包含有物理页号。

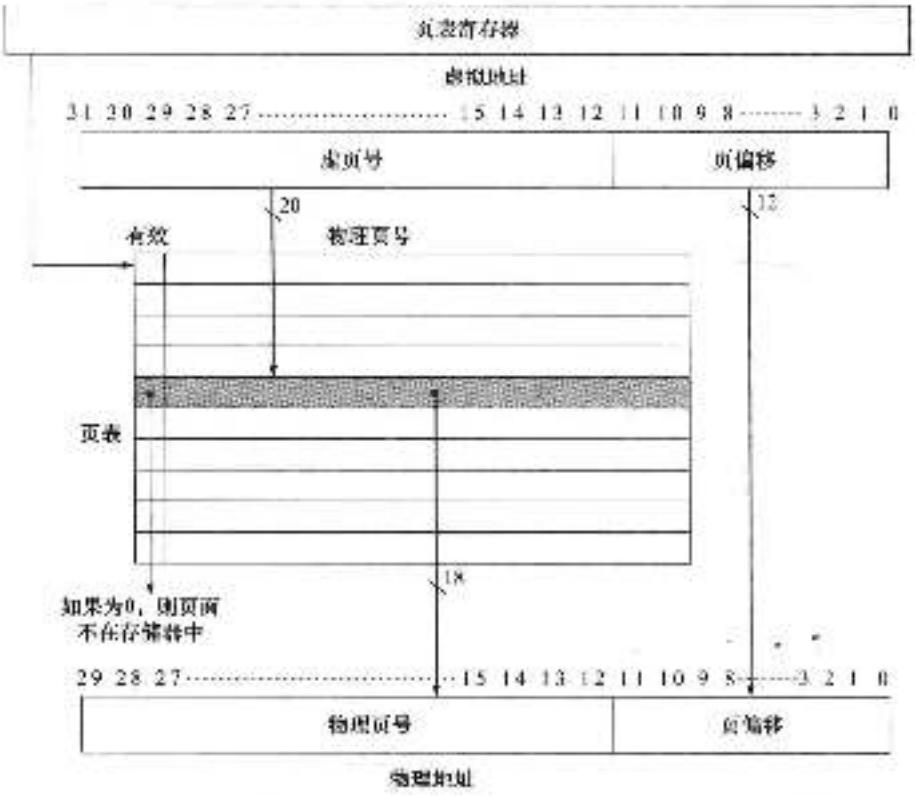



图5-27 用虚页号来索引页表以获得对应的物理地址部分。假定地址为32位。页表的首地址由页表寄存器给出。在本图中，页大小为$2^{12}$字节，即4KiB，虚拟地址空间为$2^{32}$字节，即4GiB，物理地址空间为$2^{30}$字节，可以支持高达1GiB的主存。页表项数为$2^{20}$，即100万项。每一项的有效位表明了该页是否合法。如果该位为0，那么该页就不在主存中。尽管图中所示的页表项宽度只需19位，但为了寻址方便，通常让它有32位。其他位则用来存放每页都需保留的基本的附加信息，如保护信息。

由于页表包含了每个可能的虚拟页的映射，因此不需要标记位。在cache术语中，索引是用来访问页表的，由整个块地址即虚页号组成。

## 5.7.2 缺页故障

如果虚拟页的有效位关闭，就会发生缺页故障。操作系统获得控制权。控制的转移由异常机制完成，这点在第4章已经看到并在本节稍后进行讨论。一旦操作系统获得控制权，它必须在下一级存储器层次（通常是闪存或磁盘）中找到该页，然后决定将其放到主存中。

虚拟地址本身并不会马上告诉我们页在磁盘中的位置。还拿图书馆作类比，我们不能仅仅依靠书名就找到图书的具体位置。而是按目录查找，获得书在书架上的位置信息，比如说图书馆的索引书号。同样，在虚拟存储系统中，我们必须保持跟踪记录虚拟地址空间的每一页在磁盘上的位置。

由于我们无法提前获知存储器中的某一页什么时候将被替换出去，因此操作系统在创建进程的时候通常会在闪存或磁盘上为进程中所有的页创建空间。这一磁盘空间称为**交换区**（swap

space）。同时，它也会创建一个数据结构来记录每个虚拟页在磁盘上的存放位置。这个数据结构可能是页表中的一部分，也可能是辅助数据结构，寻址方式和页表一样。图 5-28 是一个包含物理页号或磁盘地址的单个表的结构。

⇨ 交换区：为进程的全部虚拟地址空间所预留的磁盘空间。

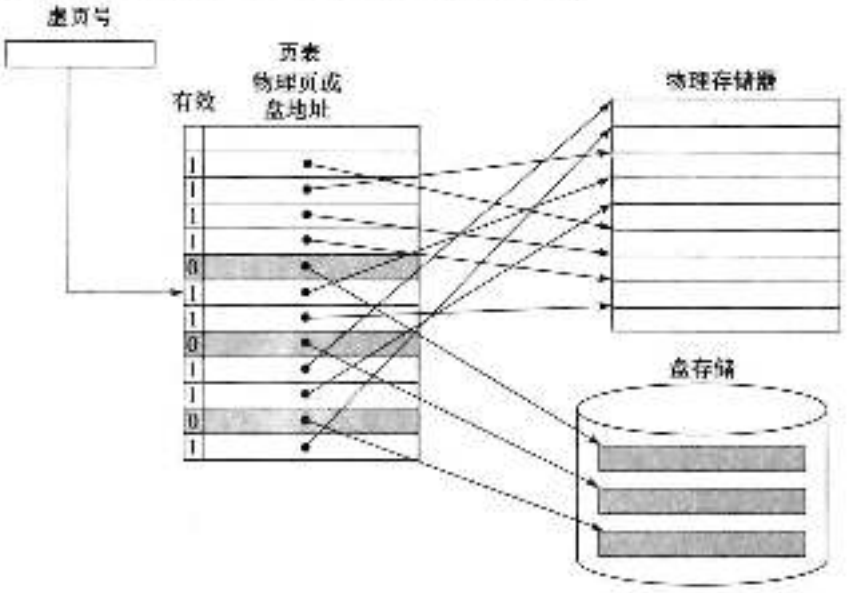



图 5-28 页表将虚拟存储器中的每一页映射到主存中的一页或者存储结构的下一层（磁盘上的一页）。虚页号用来检索页表。如果有效位开启，页表提供虚页对应的物理页号（如存储器中该页的首址址）。如果有效位关闭，那么该页就只存在磁盘上的某个指定的磁盘地址。在许多系统中，物理页地址和磁盘页地址的表在逻辑上是一个表，但是保存在两个独立的数据结构中。因为即使有些页当前不在主存中，我们也必须保存所有页的磁盘地址。所以使用双表在某种程度上是合理的。请记住主存中的页和磁盘上的页大小相等

操作系统同样会创建一个数据结构来跟踪记录使用每个物理页的是哪些进程和哪些虚拟地址。当一次缺页发生时，如果主存中所有的页都在使用，操作系统仍必须选择一页进行替换。我们希望最小化缺页的次数，因而大多数操作系统都会选择它们认为近期内不会被使用的页进行替换。使用过去的信息来预测未来的使用情况，操作系统遵循我们在 5.4 节中提到的最近最少使用（LRU）替换策略。操作系统查找最近最少使用的页，假定某一页在很长一段时间都没有被访问，那么该页再被访问的可能性比最近经常访问的页的可能性要小。被替换的页写入磁盘的交换区。如果还不是很明白，可以把操作系统看成是另一个进程，而那些控制主存的表也在主存中；这看起来似乎有些矛盾，稍后将具体解释。 434

**硬件/软件接口** 要完全准确地执行 LRU 算法的代价太高了，因为每次存储器访问时都需要更新数据结构。作为替代，大多数操作系统通过跟踪哪些页最近被使用，哪些页最近没有用到来近似地实现 LRU 算法。为了帮助操作系统估算最近最少使用的页，一些计算机提供了一个引用位（reference bit）或者称为使用位（use bit），当一页被访问时该位被置位。操作系统定期将引用位清零，然后再重新记录，这样就可以判定在这段特定时间内哪些页被访问过。有了这些使用信息，操作系统就可以从那些最近最少访问的页中选择一页（通过检查其引用位是否关闭）。如果硬件没有提供这一位，操作系统就要通过其他的方法来估计哪些页被访问过。

- 引用位：也称为使用位。每当访问一个页面时该位被置位，通常用来实现LRU或其他替换策略。

**精解** 虚拟地址为32位，页大小为4KiB，页表每一项为4字节，我们可以计算总的页表容量为：

$$\text{页表项数} = 2^{32}/2^{12} = 2^{20}$$

$$\text{页表容量} = 2^{20}\text{个页表项} \times 2^{2}\text{字节/页表项} = 4\text{MiB}$$

也就是说，每个程序在执行的任何时候都需要4MiB的存储器空间。对单个进程来说，这个大小并不差。但是如果计算机中同时有成百上千的进程同时运行时，每一个程序有各自的页表，这将会怎样？我们又如何处理64位地址，通过这个计算需要$2^{52}$个字？

一系列的技术已经被用于减少页表所需的存储量。下面5种技术都是针对减少所需的最大存储量以及减少用于页表的主存：

1）最简单的技术是使用一个界限寄存器，对给定的进程限制其页表的大小。如果虚拟页号大于界限寄存器中的值，就必须在页表中加入该项。这种技术允许页表随着进程消耗空间的增多而增长。因此，只有当进程使用了虚拟地址空间的许多页时，页表才会变得很大。这种技术要求地址空间只朝一个方向扩展。

2）允许地址空间只朝一个方向增长并不够，因为多数语言需要两种大小可扩展的区域：一个用来保留栈，一个用来保留堆。由于这种二元性，如果将页表划分，使其既能从最高地址向下扩展，也能从最低地址向上扩展，就方便多了。这也就意味着有两个独立的页表和两个独立的界限。两个页表的使用将地址空间分成两段。地址的高位用来判断该地址使用了哪个段和哪个页表。由于段由地址的高位部分决定，每一段可以在地址空间的一半大。每段的界限寄存器指定了当前段的大小，该大小以页为单位增长。这种类型的段被应用于很多体系结构，包括MIPS结构。不同于5.7节的第三个精解中讨论的段，这种形式的段对应用程序是不可见的，尽管它对操作系统可见。这种机制主要的缺陷在于当以一种零散方式使用地址空间而不是一组连续的虚拟地址时，它的执行效果就不太好。

3）另外一种减小页表容量的方法是对虚拟地址使用哈希函数，这样，页表需要的容量仅仅是主存中的物理页数。这种结构称为反置页表（inverted page table）。当然，反置页表的查找过程略微有些复杂，因为我们不能仅仅依靠索引来访问页表。

435
!
436

4）多级页表同样可以用来减少页表存储量。第一级映射到虚拟地址空间中较大的固定大小的块，一共有64～256项。这些大的具有时候称为段，而第一级的映射表有时称为段表。对用户来说段表是不可见的。段表中的每一项指出了该段中是否有页被分配，如果有，就指向该段的页表。地址转换发生在第一次段表查找时，使用地址的高位部分。如果段地址有效，下一组高位地址则用来索引由段表项指向的页表。这种机制允许以一种稀疏的方式（多个不相连的段同时处于活跃状态）来使用地址空间而不用分配整个页表。对很大的地址空间和在需要非连续地址分配的软件系统中，这种机制尤为有效。但是这种两级映射方式的主要缺陷在于地址转换过程更为复杂。

5）为了减少页表占用的实际主存空间，现在，多数系统也允许将页表可分页。这听起来很复杂，但是它的工作原理和虚拟存储器一样，并且允许页表驻留在虚拟地址空间中。另外，还有一些很小却很关键的问题，例如，要避免不断出现的缺页。如何处理这些问题需要描述地很细节，并且一般对机器的依赖性很高。简而言之，要避免这些问题，可以将全部页表置于操作系统地址空间中，并且至少要把操作系统中一部分页表放在主存中的可物理寻址的一块区域中，这部分页表总是存在于主存而非磁盘中。

### 5.7.3 关于写

访问 cache 和主存的时间相差上百个时钟周期，写直达机制也可以使用，但是我们需要一个写缓冲区来隐藏写延迟。在虚拟存储器系统中，对存储器层次结构中下一层（磁盘）的写操作需要数百万个处理器时钟周期；因此，创建一个缓冲区用来允许系统用写直达的方式对磁盘进行写的方法是完全不可行的。相反，虚拟存储器系统必须使用写回机制，对存储器中的页进行单独的写操作，并且在该页被替换出存储器时再被复制到磁盘中去。

**硬件/软件接口** 在虚拟存储系统中，写回机制有另一个主要的优点。因为相对于磁盘访问时间，其传输时间要少得多，因此，把整页复制回磁盘比把单个字写回要高效得多。尽管写回操作比传输单个字更高效，但是开销却很大。因此，当某一页被替换时，我们希望知道该页是否需要被复制写回。为了追踪读入主存中的页是否被写过，可以在页表中增加一个脏位（dirty bit）。当页中任何字被写时就将这一位置位。如果操作系统选择替换某一页，脏位指明了在把该页所占用的主存让给另一页之前，是否需要将该页写回磁盘。因此，一个修改过的页也通常被称为脏页（dirty page）。

437

### 5.7.4 加快地址转换：TLB

由于页表存放在主存中，因此程序每次访存至少需要两次：第一次访存先获得物理地址，第二次访存才获得数据。提高访问性能的关键在于依靠页表的访问局部性。当一个转换的虚页号被使用时，它可能在不久的将来再次被用到，因为对该页中字的引用同时具有时间局部性和空间局部性。

因此，现代处理器都包含一个特殊的 cache 以跟踪最近使用过的地址变换。这个特殊的地址转换 cache 通常称为**快表**（Translation-Lookaside Buffer，TLB）（将其称为地址变换高速缓存更精确）。TLB 就相当于记录目录中的一些书的位置的小纸片；我们在纸片上记录一些书的位置，并且将小纸片当成图书馆索书号的 cache，这样就不用一直在整个目录中搜索了。

⊃ 快表：用于记录最近使用地址的映射信息的高速缓存，从而可以避免每次都要访问页表。

如图 5-29 所示，TLB 的每个标记项存放虚页号的一部分，每个数据项中存放了物理页号。由于我们每次访问的是 TLB 而不是页表，TLB 需要包括其他状态位，如脏位和引用位。

每次访问，我们都要在 TLB 中查找虚页号。如果命中，物理页号就用来形成地址，相应的引用位被置位。如果处理器执行的是写操作，脏位同样要被置位。如果 TLB 发生缺失，我们必须判断是发生缺页还是仅仅是一次 TLB 缺失。如果该页在主存中，那么 TLB 缺失只是一次转换缺失。在这种情况下，处理器可以通过将页表中的变换装载到 TLB 中并且重新访问来进行缺失处理。如果该页不在主存中，TLB 缺失就是一次真的缺页。在这种情况下，处理器调用操作系统的异常处理。由于 TLB 中的项比主存中的页数少得多，发生 TLB 缺失会比缺页频繁得多。

TLB 缺失既可以通过硬件处理，也可以通过软件处理。实际上，两种方法的性能差别很小，这是因为无论哪种方法，需要执行的基本操作都是一样的。

在发生了 TLB 缺失，并且已经在页表中找到了缺失的变化时，我们就需要从 TLB 中选择一项进行替换。由于 TLB 表项中包含了引用位和脏位，当替换某一项时，需要把这些位复制回页表项中。这些位是 TLB 表项中唯一可以修改的部分。利用写回策略——只是在缺失的时候将这些表项写回而不是任何写操作都写回——是非常有效的，因为我们期望 TLB

缺失率更低。一些系统使用共生技术来近似引用位和脏位，以消除除了缺失后装入新表项之外写 TLB 的必要。

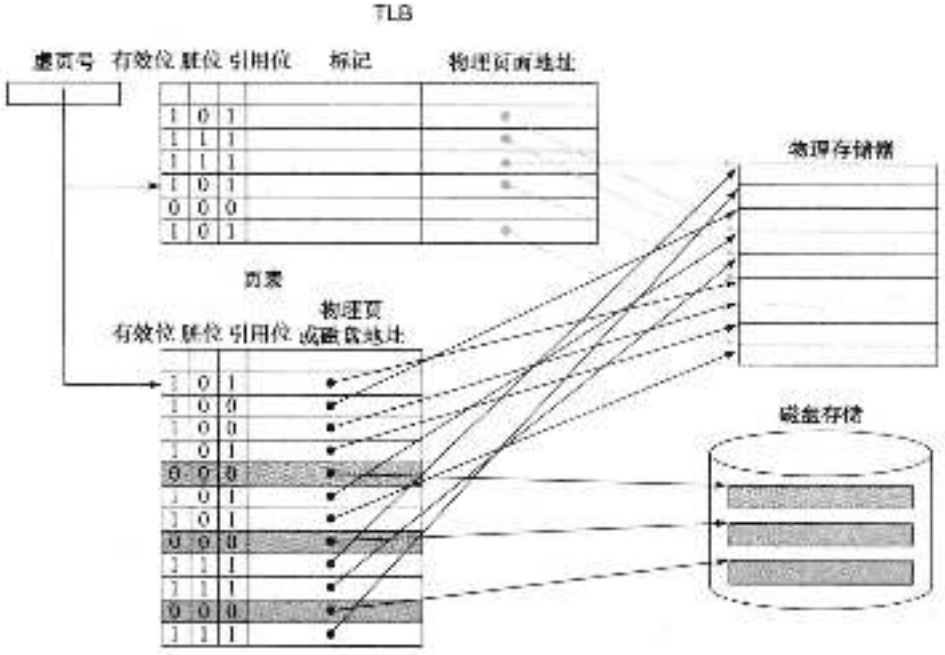



图 5-29 TLB 作为页表的 cache，用于存放映射到物理页中的那些项。TLB 包含了页表中虚页到物理页映射的一个子集。TLB 映射以灰线显示。因为 TLB 是一个 cache，它必须有标记域。如果一个页在 TLB 中没有匹配的项，就必须检查页表。页表或者提供该页的物理页号（可用来创建一个 TLB 项），或者指出该页在磁盘上，这时就会发生缺页。由于页表对于每个虚页都有一个相应的项，并不需要标记；换句话说，不同于 TLB，页表并不是 cache

438

TLB 的一些典型的值为：

- TLB 大小：16～512 个项。
- 块大小：1～2 个页表项（通常每个为 4～8 字节）。
- 命中时间：0.5～1 个时钟周期。
- 缺失代价：10～100 个时钟周期。
- 缺失率：0.01%～1%。

设计者在 TLB 设计中相联度的设置非常多样化。有些系统使用小的全相联的 TLB，这是由于全相联有较低的缺失率；此外，由于 TLB 很小，全相联映射的成本也不会太高。其他一些系统通常使用相联度低且容量大的 TLB。在全相联映射这方式下，由于用硬件实现 LRU 策略的代价很大，因此替换项的选择就很复杂。另外，由于 TLB 的缺失比缺页要频繁得多，因此需要用较低的代价来处理缺失，而不能像缺页处理那样选择一个开销大
439 的软件算法。所以很多系统都支持随机地选择替换表项的方法。在 5.8 节中我们将会详细讨论替换策略。

### 内置 FastMATH TLB

为了弄清楚这些想法是如何实际应用到处理器中的，我们来进一步研究内置 FastMATH 的 TLB。存储系统页大小为 4KiB，地址空间为 32 位，因此，虚页号长为 20 位，如图 5-30 顶部所示。物理地址和虚拟地址长度相等。TLB 包含了 16 个项，采用全相联映射，由指令和数据访问共享。每个表项宽为 64 位，包含了 20 位的标记位（作为该 TLB 表项的虚页号）、相应的物

理页号（也是20位）、一个有效位、一个脏位以及一些其他管理操作位。与大多数 MIPS 系统类似，它采用软件来处理 TLB 缺失。

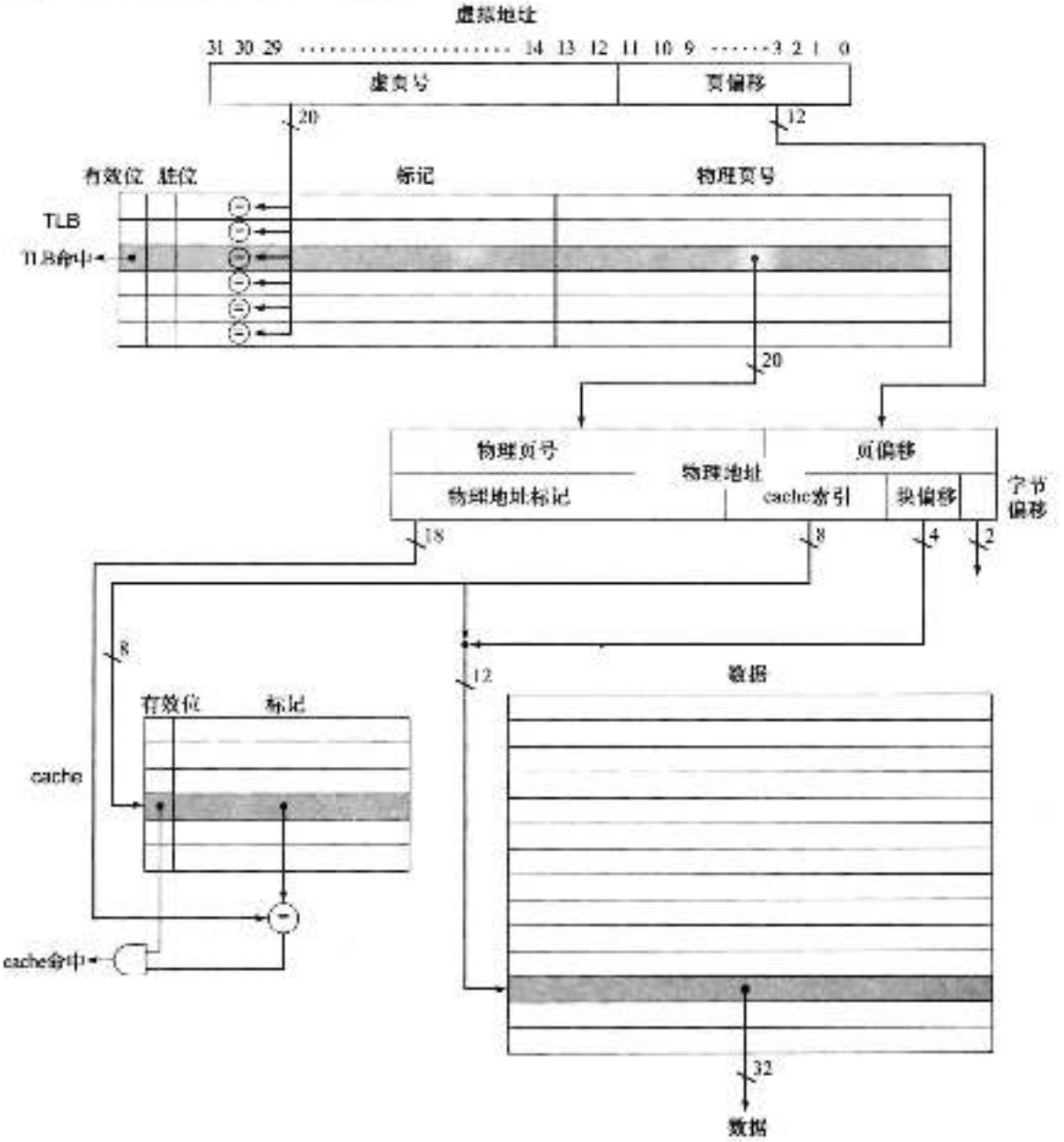



图 5-30 内置 FastMATH 中 TLB 和 cache 实现了从虚拟地址到数据项的转换过程。本图描述了 TLB 和数据 cache 的结构，这里假设页大小是 4KiB。本图主要研究读操作，图 5-31 则描述了如何处理写操作。注意到不同于图 5-12，标记和数据 RAM 是分开的。用 cache 索引和块偏移来寻址长而窄的数据 RAM，无需使用 16:1 的多路选择器我们也能选出块中所需的字。当 cache 采用直接映射方式时，TLB 是全相联的。由于需要的项可能在 TLB 中的任何位置，因此要实现全相联 TLB 需要将每个 TLB 标记都与虚页号进行比较（参考 5.4.2 节精解的内容）。如果匹配表项的有效位有效，那么 TLB 访问命中，物理页号与页偏移中的位共同形成访问 cache 的索引

图 5-30 是 TLB 和一个 cache，图 5-31 则说明了处理一次读或写请求的步骤。当一次 TLB 缺失发生时，MIPS 硬件把被访问的页号保存在一个特殊寄存器中，并产生一次异常。异常请求操作系统通过软件处理缺失。为了找到缺失的页的物理地址，TLB 缺失程序用虚拟地址的页

号，以及能指出活跃进程页表起始地址的页表寄存器来检索页表。通过使用一系列更新 TLB 的特殊指令，操作系统将页表中的物理地址放入 TLB 中。假设代码和页表项都在指令 cache 和数据 cache 中，那么一次 TLB 缺失大概需要花费 13 个时钟周期（在 5.7.7 节，我们将讨论 MIPS TLB 代码）。如果页表项中没有有效的物理地址，就会发生一次真的缺页，硬件保存着被建议替换项的索引，而这一项则是随机选取的。

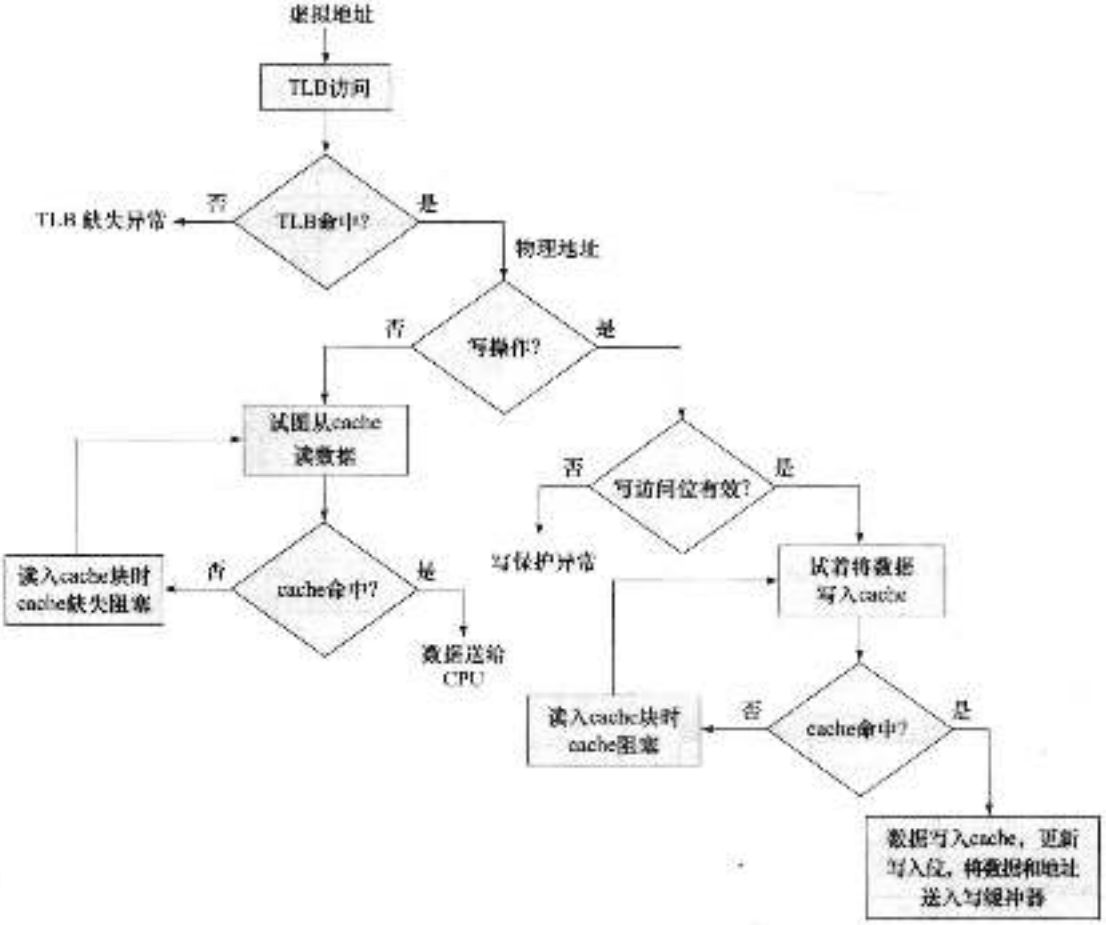



图 5-31 在内置 FastMATH 的 TLB 和 cache 中处理读或者写首达操作。如果 TLB 命中，最终的物理地址就可以用来访问 cache。对于读操作，当从存储器中取数据时，cache 产生命中或缺失，提供数据或者引起阻塞。对于写操作，若命中，cache 某数据项中的一部分内容将被重写。如果采用写直达策略还要将数据送到写缓冲区中。写缺失和读缺失相同，只是在数据块从存储器中读出后会被修改。写回策略需要将 cache 的写入位置位，并且只有当读或写缺失时，如果被替换的块处于修改状态，才将整块写入写缓冲。注意，TLB 命中和 cache 命中是相互独立的事件，但是 cache 命中只可能发生在 TLB 命中之后，这就意味着数据必须在主存中。TLB 缺失和 cache 缺失之间的联系将在接下来的例子和本章最后的习题中进一步研究

对于写请求来说，有一个额外的复杂情况：必须检查 TLB 中的写访问位。该位可以阻止程序向它仅具有读权限的页中进行写操作。如果程序试图写，且写访问位是关闭的，则会产生异常。写访问位构成了保护机制的一部分，我们将在稍后讨论。

### 5.7.5 集成虚拟存储器、TLB 和 cache

虚拟存储器和 cache 系统就像一个层次结构一样共同工作。因此除非数据在主存中，否则

它不可能在 cache 中出现。操作系统帮助管理该层次结构，当它决定将某一页移到磁盘上去时，就在 cache 中将该页中的内容刷新。同时，操作系统修改页表和 TLB，而后尝试访问该页上的数据都将发生缺页。

在最好的情况下，虚拟地址由 TLB 进行转换，然后被送到 cache，找到相应的数据，取回并送入处理器。在最坏的情况下，访问在存储器层次结构的 TLB、页表和 cache 这三个部件中都发生缺失。下面的例子将详细介绍这些交互作用。

440 - 442

### 例题：存储器层次结构的全部操作

存储器层次结构类似于图 5-30，由一个 TLB 和一个 cache 组成。一次存储器访问可能遇到三种不同类型的缺失：TLB 缺失、缺页以及 cache 缺失。考虑这三种缺失发生一个或多个时所有可能的组合（7 种可能性）。对每种可能性，说明这种情况是否会真的发生，在什么条件下发生。

### 答案

图 5-32 说明了所有可能发生的组合以及事实上它们是否真的可能发生。

| TLB | 页表 | cache | 这种情况可能发生么？如果可能，在什么情况下发生？ |
|---|---|---|---|
| 命中 | 命中 | 缺失 | 可能，但若 TLB 命中就不可能检查页表 |
| 缺失 | 命中 | 命中 | TLB 缺失，但在页表中找到表项；重试后在 cache 中找到数据 |
| 缺失 | 命中 | 缺失 | TLB 缺失，但在页表中找到表项；重试后在 cache 中未找到数据 |
| 缺失 | 缺失 | 缺失 | TLB 缺失，随后发生缺页；重试后在 cache 中必须不到数据 |
| 命中 | 缺失 | 缺失 | 不可能：如果页不在主存中，TLB 中没有此转换 |
| 命中 | 缺失 | 命中 | 不可能：如果页不在主存中，TLB 中没有此转换 |
| 缺失 | 缺失 | 命中 | 不可能：如果页不在主存中，数据不允许在 cache 中存在 |

图 5-32 在 TLB、虚拟存储器系统以及 cache 中可能发生的事件组合。在这些组合中，有二种是不可能的，有一种是可能的但是永远不可能检测到（TLB 命中，虚拟存储器命中，cache 缺失） □

**解释** 图 5-32 假定在访问 cache 之前，所有存储器地址都被转换成物理地址。在这种结构中，cache 是按物理地址索引（physically indexed）并且是物理标记（physically tagged）的（所有 cache 的索引和标记都用物理地址，而不是虚拟地址）。在这个系统中，假定 cache 命中，那么访问主存的时间要包括对 TLB 访问和 cache 访问的时间。当然，这些访问可以流水地执行。

另外，处理器可以用一个完整的或者部分虚拟的地址来索引 cache。这称为虚拟寻址 cache（virtually addressed cache），它使用虚拟地址作为标记；因此这种 cache 是按虚拟地址索引（virtually indexed）并且是虚拟标记（virtually tagged）的。在这种 cache 中，地址转换硬件（TLB）在正常的 cache 访问过程中没有被用到。这是因为使用的是没有被转换成物理地址的虚拟地址来访问 cache 的。这样就把 TLB 排除在关键路径之外，减少了 cache 延时。当 cache 访问缺失时，处理器需要将该地址转换成物理地址以便从主存中取出 cache 块。

当使用虚拟地址访问 cache，并且进程之间共享页（进程可能使用不同的虚拟地址访问页）时，就可能有别名（aliasing）存在。当同一个对象有两个名字时就会产生别名——在这种情况下，两个虚拟地址对应于同一个页。这种多义性就会产生一个问题，由于页上的一个字可能存在于 cache 中的两个不同位置，每个位置对应不同的虚拟地址。这就会出现一个程序写数据，而另一个程序并不知道数据已经改变的情况。完全虚拟寻址的 cache 或者对 cache 和 TLB 的设计进行限制以减少别名，或者需要操作系统也可以是让用户来采取措施以保证别名不会发生。

这两种设计观点常用的折中方法是采用虚拟地址索引的 cache——有时仅仅使用地

址的页偏移部分，由于没有被转换，因此实际上是物理地址——但使用物理标记。这项采用虚拟索引和物理标记的设计，试图同时拥有虚拟地址索引 cache 的优越性能以及物理寻址 cache（physical addressed cache）的管理结构。例如，在这种情况下就没有别名的问题。图5-30假定的页大小为4KiB，但实际上有16KiB，区块内置 FastMATH 就使用了这种方法。要实现这种方法，必须在最小页大小、cache 大小以及相联度之间进行谨慎的权衡。

443

- 虚拟寻址 cache：一种使用虚拟地址而不是物理地址访问的 cache。
- 别名：使用两个地址访问同一个目标的情形，一般发生在虚拟存储器中两个虚拟地址对应到同一个物理页时。
- 物理寻址 cache：使用物理地址寻址的 cache。

### 5.7.6 虚拟存储器中的保护

虚拟存储器最重要的功能就是允许多个进程共享一个主存，同时为这些进程和操作系统提供存储保护。保护机制必须确保：尽管多个进程在共享同一个主存，但是无论有意或是无意，一个恶意进程不能写另一个用户进程或者操作系统的地址空间。TLB 中的写访问位可以防止一个页被改写。如果没有这一级保护，计算机病毒将更加泛滥。

**硬件/软件接口** 为了使操作系统能保护虚拟存储系统，硬件至少提供下面总结的三种基本能力。注意，由于前两者都需要管理态，因此其需求相同。

1）支持至少两种模式，并指出当前运行的进程是用户进程还是操作系统进程，操作系统进程也称为超级用户管理（supervisor）进程、核心进程或者主管进程。

2）提供一部分处理器状态，这部分内容是用户进程可读而不可写的。这包括指示处理器是处于用户态还是管理态的用户/管理模式位，页表指针以及 TLB。操作系统可以通过使用只能在管理态下可用的特殊指令对它们进行写操作。

444

3）提供能让处理器在用户态和管理态下相互切换的机制。从用户态到管理态的转换通常是由系统调用（system call）异常处理实现的，它用特殊指令（如 MIPS 指令集中的 syscall）将控制权传到管理代码空间的指定位置。和其他异常处理一样，系统调用处的程序计数器中的值被保存在异常程序计数器中（EPC），处理器被置于管理态。从异常中返回至用户模式，使用异常返回（return from exception）指令，将重置用户模式，并且跳转到 EPC 中的地址处。

通过使用这些机制并且把页表保存在操作系统的地址空间中，操作系统可以更改页表，并且阻止用户进程改变它们，确保用户进程只能访问由操作系统提供给它的存储部分。

- 超级用户管理模式：也称作核心模式（kernel mode）。运行操作系统进程的模式。
- 系统调用：将控制权从用户模式转换到管理模式的特殊指令，触发进程中的一个异常机制。

我们同样要防止一个进程读取另一个进程的数据。例如，当成绩放在处理器的主存中，我们不希望学生程序读到它们。一旦我们开始共享主存，必须赋予进程保护数据防止被其他进程读或写的能力；否则，共享主存将变得乱七八糟。

每个进程有它自己的虚拟地址空间。因此，如果操作系统管理页表的组织，使独立的虚拟

页映射到不相交（disjoint）的物理页上，就能使得一个进程无法访问另一个进程的数据了。当然，这也要求一个用户进程不能改变页表的映射。如果操作系统能防止用户进程更改自己的页表，那么安全性也就有了保证。然而，这样一来，操作系统必须负责修改页表。将页表放在操作系统的保护地址空间就能满足所有要求。

当进程希望以受限的方式共享信息时，操作系统必须协助它们，这是因为访问另一个进程的信息需要改变访问进程的页表。写访问位可以用来把共享限制为只读，并且和页表中其他位一样，该位只能被操作系统修改。为了允许另一个进程，设为 P1，去读属于进程 P2 的一页，P2 就要请求操作系统在 P1 地址空间中为一个虚拟页生成页表项，指向 P2 想要共享的物理页。如果 P2 要求，操作系统可以使用写保护位以防止 P1 对数据进行改写。由于只有 TLB 缺失才会访问页表，任何决定页的访问权限的位不仅要包含在页表中，还要包含在 TLB 中。 445

**解释** 当操作系统决定从执行进程 P1 切换到执行进程 P2（称为上下文切换（context switch），或者进程切换）时，它必须保证 P2 不能访问 P1 的页表，否则不利于数据保护。如果没有 TLB，只要把页表寄存器转而指向 P2 的页表（而不是 P1 的）就足够了；如果有 TLB，我们必须在其中清除属于 P1 的表项——不仅是为了保护 P1 的数据，而且是为了迫使 TLB 装入 P2 的表项。如果进程切换的频率很高，这一举措的效率就很低。例如，在操作系统切换回 P1 之前，P2 可能只装入了很少的 TLB 表项。不幸的是，P1 随后发现它所有的表项都不见了，无此不得不通过 TLB 缺失来重新加载这些表项。产生这个问题是因为 P1 和 P2 使用同一个虚拟地址空间，并且我们必须清除 TLB 以防止地址混淆。

另一种常见的方法则是通过增加进程标识符（process identifier）和任务标识符（task identifier）来扩展虚拟地址空间。为此内置 FastMATH 有 8 位地址空间标识域（ASID）。就是这个域标识了当前正在运行的进程：当进程切换时，它保存在由操作系统装入的寄存器中。进程标识符与 TLB 的标记部分相连接，因此只有在页号和进程标识符同时匹配时，TLB 才会发生命中。这样的话，除非特殊情况，否则我们就不需要清除 TLB。

同样的问题可能在 cache 中发生，这是由于在进程切换的时候，cache 包含正在执行的进程的数据。对物理寻址和虚拟寻址的 cache 来说，这些问题以不同方式产生，并且有不同的解决方法，比如使用进程标识符来确保一个进程只能获得它自己的数据。

⇒ 上下文切换：为允许另一个不同的进程使用处理器，改变处理器内部的状态，并保存当前进程返回时需要的状态。

### 5.7.7 处理 TLB 缺失和缺页

尽管当 TLB 命中时，利用 TLB 将虚拟地址转换成物理地址是很简单的，但是处理 TLB 缺失和缺页要复杂得多。当 TLB 中没有一个表项能匹配虚拟地址时，TLB 缺失就会发生。TLB 缺失有下面两种可能性之一：

1）页在主存中，只需要创建缺失的 TLB 表项。

2）页不在主存中，需要将控制权交给操作系统来解决缺页。

MIPS 通常采用软件来处理 TLB 缺失。它从主存中取出页表项装入 TLB，然后重新执行引起 TLB 缺失的那条指令，这时就会得到 TLB 命中。如果页表项指出该页不在主存中，此时就会发生缺页异常。

处理 TLB 缺失或者缺页需要使用异常机制来中断活跃的进程，将控制权传给操作系统，然

后恢复执行被中断的进程。缺页将在主存访问时钟周期的某一时刻被发现。为了在缺页处理完毕后重新启动引起缺页的指令，必须保存该指令的程序计数器中的值。正如第 4 章所述，并常
446 程序计数器（Exception Program Counter，EPC）用来保存这个值。

另外，TLB 缺失或者缺页异常必须在访存发生的同一个时钟周期的末尾被判定，因此下一个时钟周期就开始进行异常处理而不是继续正常的指令执行。如果在这个时钟周期没有断定缺页发生，一条 load 指令可能改写寄存器，而当我们试图重新启动指令时，这可能是灾难性的错误。例如，考虑指令 lw $1,0($1)：计算机必须防止写流水级发生，否则，就不能重新启动指令，因为 $1 的内容将被破坏。store 指令也会发生类似复杂情况。当发生缺页而没有完成处理时，我们必须阻止写主存的操作；这通常是通过令到主存写控制线为无效来完成。

**硬件/软件接口** 在操作系统开始进行异常处理和保存处理器所有状态位的时候，操作系统特别脆弱。例如，如果在操作系统中正在处理第一个异常时，另一个异常又发生了，控制单元将重写异常程序计数器，就不能返回引起缺页的那条指令。我们可以通过提供禁止异常（disable exception）和使能异常（enable exception）来避免这种错误的发生。当异常第一次发生时，处理器设置一个管理态模式位，禁止其他异常的发生；这可以与处理器设置管理态模式位同时进行。随后操作系统保存足够的状态，如果有另一个异常发生——异常程序计数器（EPC）和异常引发寄存器也能保存这些状态。异常程序计数器和异常引发寄存器是协助处理异常、TLB 缺失以及缺页的两个特殊控制寄存器；图 5-33 列出了其他的寄存器。而后操作系统可以重新允许异常发生。这些步骤保证了异常不会使处理器丢失任何状态，因此也就不会出现无法重新执行中断指令的情况。

使能异常：也称为中断使能（interrupt enable），用于控制处理器是否响应异常的信号或动作；在处理器安全地保存重启所需信息之前，必须阻止异常的发生。

| 寄存器 | CP0 寄存器号 | 说明 |
|---|---|---|
| EPC | 14 | 异常之后重启的位置 |
| Cause | 13 | 异常的原因 |
| BadVAddr | 8 | 引发异常的地址 |
| Index | 0 | TLB 中读/写的位置 |
| Random | 1 | TLB 中伪随机位置 |
| EntryLo | 2 | 物理页地址和标记位 |
| EntryHi | 10 | 虚页地址 |
| Context | 4 | 页表地址和页号 |

图 5-33 MIPS 控制寄存器。这些寄存器被视为位于协处理器 0 中，因此读时使用 mfc0，写时使用 mtc0

一旦操作系统知道了引起缺页的虚拟地址，它必须完成以下三个步骤：

1）使用虚拟地址查找页表项，并在磁盘上找到被访问的页的位置。

2）选择替换一个物理页；如果被选中的页被修改过，需要在把新的虚拟页装入之前将这个物理页写回到磁盘上。

447 3）启动读操作，将被访问的页从磁盘上取回到所选择的物理页的位置上。

当然，最后一个步骤将花费数百万个时钟周期（如果被替换的页被重写过，那么第二步也需要花费这么多时间）；因此，操作系统通常都会选择另一个进程在处理器上执行直到磁盘访问结

束。由于操作系统已经保存了当前进程的状态，因此它可以很方便地将控制权交给另一个进程。

当从磁盘读页的操作完成后，操作系统可以恢复原先引起缺页的进程状态，并且执行从异常返回的指令。该指令将处理器从核心态恢复到用户态，同时也恢复程序计数器的值。用户进程接着重新执行引发缺页的那条指令，成功地访问请求的页，然后继续执行。

数据访问引起的缺页异常很难处理，这是由于以下三个特性：

1）它们发生于指令中间，不同于指令缺页。

2）在异常处理前指令没有结束。

3）异常处理之后，指令必须重新执行，就好像什么都没发生过一样。

要保持指令**可重新启动**（restartable），这样异常被处理之后，指令也能继续执行，这在类似于 MIPS 的结构中的实现相对简单。因为每条指令只能写一个数据项并且只能在指令周期的最后进行写操作，我们就可以阻止指令的完成（不执行写操作）并且在开始处重新启动指令。

⊃ 可重启指令：一种在异常被处理之后能从异常中恢复而不会影响指令的执行结果的指令。

我们再来看 MIPS 的一些细节。当 TLB 发生缺失时，MIPS 的硬件将被引用的页号保存在一个叫 BadVAddr 的特殊寄存器里，然后产生异常。 448

这个异常请求操作系统通过软件来处理缺失。控制权被传到地址 $8000\ 0000_{16}$——TLB 缺失**处理程序**（handler）的位置。为了找到缺失页的物理地址，TLB 缺失处理程序使用虚拟地址的页号，以及指向活动进程页表起始地址的页表寄存器来检索页表。为了能快速地检索，MIPS 将所需的一切信息都放在特殊的现场寄存器（Context）中：高 12 位是页表的基准地址，接下来的 18 位是缺失页的虚拟地址。每个页表项是 1 个字，因此最后两位为 0。因此，头两条指令将现场寄存器中的内容复制到内核临时寄存器 $k1 中，然后根据其中的地址将页表项装入 $k1。回想 $k0 和 $k1 是为操作系统保留的不做保存的寄存器；这样做的主要原因是使得 TLB 缺失处理程序执行得更快。下面是典型的 TLB 缺失处理程序的 MIPS 代码：

```
TLBmiss:
   mfc0   $k1,Context    # copy address of PTE into temp $k1
   lw     $k1,0($k1)     # put PTE into temp $k1
   mtc0   $k1,EntryLo    # put PTE into special register EntryLo
   tlbwr                 # put EntryLo into TLB entry at Random
   eret                  # return from TLB miss exception
```

⊃ 处理程序：用于“处理”异常或中断的软件程序的名字。

正如上面所示，MIPS 有一组特殊的系统指令用来更新 TLB。指令 tlbwr 把控制寄存器 EntryLo 中的内容复制到由控制寄存器 Random 所选择的 TLB 表项中。Random 实现随机替换，所以它基本上是一个独立运行的计数器。TLB 缺失大概要花费 12 个时钟周期。

注意到 TLB 缺失处理程序并不检查页表项是否有效。因为发生 TLB 表项缺失异常比缺页异常要频繁得多，所以操作系统对页表中的表项并不做检查就直接装入 TLB 并重新执行指令。如果表项无效，另一个不同的异常就发生，操作系统认为缺页。这种方法让频繁发生的 TLB 缺失处理得快一些，但是对不频繁发生的缺页处理就会有一些性能损失。

一旦产生缺页的进程被中断，控制权就被转到 $8000\ 0180_{16}$，这是一个与 TLB 缺失处理程序不相同的地址。它是处理异常的通用地址；TLB 缺失有一个专门的入口点是为了减少 TLB 缺失代价的。操作系统使用异常引发寄存器来判断产生异常的原因。由于是缺页异常，操作系统知道需要进一步处理。因此，不同于 TLB 缺失，它保存了活动进程的全部状态，包括所有的通用寄存器和浮点寄存器、页表地址寄存器、EPC 和异常引发寄存器的状态。由于浮点寄存器在异常处理程序不常使用，因此通用入口点并没有保存它们，而是留给少数需要它们的处理者。 449

图 5-34 描述了异常处理程序的 MIPS 代码。我们使用 MIPS 代码来保存和恢复状态，注意何时允许和禁止异常，但是我们调用 C 代码来处理特殊的异常。

| | | | |
|---|---|---|---|
| **保存状态** | | | |
| Save GPR | addi | $k1,$sp, -XCPSIZE | # save space on stack for state |
| | sw | $sp, XCT_SP($k1) | # save $sp on stack |
| | sw | $v0, XCT_V0($k1) | # save $v0 on stack |
| | ... | | # save $v1, $a1, $s1, $t1,... on stack |
| | sw | $ra, XCT_RA($k1) | # save $ra on stack |
| 保存hi, lo | mfhi | $v0 | # copy Hi |
| | mflo | $v1 | # copy Lo |
| | sw | $v0, XCT_HI($k1) | # save Hi value on stack |
| | sw | $v1, XCT_LO($k1) | # save Lo value on stack |
| 保存异常寄存器 | mfc0 | $a0, $cr | # copy cause register |
| | sw | $a0, XCT_CR($k1) | # save $cr value on stack |
| | ... | | # save $v1,.... |
| | mfc0 | $a3, $sr | # copy status register |
| | sw | $a3, XCT_SR($k1) | # save $sr on stack |
| 设置sp | move | $sp, $k1 | # sp = sp - XCPSIZE |
| **寄存器允许嵌套异常** | | | |
| | andi | $v0, $a3, MASK1 | # $v0 = $sr & MASK1, enable exceptions |
| | mtc0 | $v0, $sr | # $sr = value that enables exceptions |
| **调用C异常处理程序** | | | |
| Set $gp | move | $gp, GPINIT | # set $gp to point to heap area |
| Call C code | move | $a0, $sp | # arg1 = pointer to exception stack |
| | jal | xcpt_deliver | # call C code to handle exception |
| **恢复状态** | | | |
| Restore most GPR, hi, lo | move | $at, $sp | # temporary value of $sp |
| | lw | $ra, XCT_RA($at) | # restore $ra from stack |
| | ... | | # restore $t0,...., $a1 |
| | lw | $a0, XCT_A0($k1) | # restore $a0 from stack |
| 恢复状态寄存器 | lw | $v0, XCT_SR($at) | # load old $sr from stack |
| | li | $v1, MASK2 | # mask to disable exceptions |
| | and | $v0, $v0, $v1 | # $v0 = $sr & MASK2, disable exceptions |
| | mtc0 | $v0, $sr | # set status register |
| **异常返回** | | | |
| Restore $sp and rest of GPR used as temporary registers | lw | $sp, XCT_SP($at) | # restore $sp from stack |
| | lw | $v0, XCT_V0($at) | # restore $v0 from stack |
| | lw | $v1, XCT_V1($at) | # restore $v1 from stack |
| | lw | $k1, XCT_EPC($at) | # copy old $epc from stack |
| | lw | $at, XCT_AT($at) | # restore $at from stack |
| Restore EPC and return | mtc0 | $k1, $epc | # restore $epc |
| | eret | $ra | # return to interrupted instruction |

图 5-34 异常时保存状态和恢复状态的 MIPS 码

引发缺失的虚拟地址取决于当前缺失是指令缺失还是数据缺失。产生缺失的指令地址在 EPC 中。如果是指令缺页，EPC 中包含了缺失页的虚拟地址；否则，缺失页的虚拟地址可以通过查看指令（指令地址在 EPC 中），找到基址寄存器和偏移量来计算得到。

**01 精解** 这个简化版本假设了堆栈指针（sp）有效。为了避免执行低层异常代码时发生缺页的问题，MIPS 预留了一部分不会产生缺页的地址空间，称为非映射（unmapped）。操作系统将异常入口点代码和异常堆栈存放在非映射的内存中。MIPS 硬件将虚拟地址

$8000\ 0000_{16}$ – $BFFF\ FFFF_{16}$转换成物理地址时，虚拟地址的高位忽略不计，即把这些地址放在物理内存的低位。因此，操作系统就将异常入口点和异常堆栈放置于非映射的主存中。

⊃ 非映射：地址空间中的一个部分，在这个区域不会导致缺页异常。

**解释** 图5-34中的代码显示了MIPS-32的异常返回序列。早先的MIPS-I架构采用rfe和jr来代替eret。

**解释** 对于有着更为复杂指令的处理器来说，可能会访问主存中的很多位置并且写很多数据项，这就使指令重新启动变得更加困难。处理一条指令可能无指令中写产生多次缺页。例如，x86处理器有能访问跨百上千数据字的块移动指令。在这样的处理器中，指令通常无法像在MIPS中那样从起始位置重新启动。相反，指令必须被中断，稍后从执行中断处继续执行。在执行的中间恢复一条指令通常需要保存一些特殊状态，处理异常，然后恢复那些特殊状态。要正确地执行这项工作需要在操作系统的异常处理代码和硬件中进行细致而详细的协调。

**解释** 与每次存储器访问都需要一次间接寻址不同，虚拟机支持一个影子页表（shadow page table）用于进行用户虚拟地址到硬件物理地址的转换。通过检测对用户页表的所有修改，虚拟机可以确保硬件正在用于转换的影子页表表项与用户操作系统中的页表表项一致，不同的是在用户页表中使用正确的物理地址替代了实地址。因此，虚拟机必须在用户操作系统试图改变其页表或访问页表指针时产生自陷。这通常由用户操作系统通过对用户页表进行写保护和对页表指针的任何访问产生自陷来实现。如前所述，如果是特权操作访问页表指针后会发生后面一种情况。

450 ↓ 451

**解释** 体系结构中需要虚拟化的最后一部分是I/O。由于计算机中I/O设备数量和类型不断增加，I/O虚拟化就变成了系统虚拟化中最困难的一部分。另外一个难点是多个虚拟机之间共享实际的设备。还有一个问题是要支持大量的设备驱动程序，这在一个支持多个用户操作系统的虚拟系统上更加严重。它为每种虚拟机中各种类型的I/O设备提供一个通用的驱动，并且将其留给VMM以管理实际的I/O。

**解释** 除了要对指令集进行虚拟之外，另一个挑战是虚拟存储器的虚拟化，这主要是因为每种虚拟机上的操作系统要维护自己的页表。为了使其工作，虚拟机中实存储器和物理存储器是两个不同的概念（这两个概念通常被认为是相同的），实存储器是位于虚拟存储器和物理存储器之间的一个独立的层次。（有人使用虚拟存储器、物理存储器和机器存储器来表示相同的三个层次。）用户操作系统通过其页表将虚拟存储器映射到实存储器，虚拟机页表将用户实存储器映射到物理存储器，虚拟存储器体系结构要么如IBM VM/370和x86一样通过页表实现，要么如MIPS一样通过TLB实现。

## 5.7.8 小结

虚拟存储器是管理主存和磁盘之间数据缓存的一级存储器层次。虚拟存储器允许单个程序在主存有限的范围内扩展地址空间。更重要的是，虚拟存储器以一种保护的方式，同时支持多个活跃的进程共享主存。

管理主存和磁盘之间的存储器层次结构很具有挑战性，这是因为缺页的代价很高。通常采用下面一些技术来降低缺失率：

1）增大页的容量以便利用空间局部性并降低缺失率。

2）使用页表实现的虚拟地址和物理地址之间的映射采用全相联的方式，这样虚拟页就可以被放置到主存中的任何位置。

452 3）操作系统使用类似 LRU 和访问位之类的技术来选择替换哪一页。

写磁盘的代价是很高的，因此虚拟存储器使用写回机制并且跟踪是否有一页更改过（采用脏位）以避免把没有变化的页写回到磁盘。

虚拟存储器机制提供了从被程序使用的虚拟地址到用来访问主存的物理地址空间之间的转换。这个地址转换允许对主存进行受保护的共享，同时还提供了很多额外的好处，如简化了存储器分配。为了保证进程间受到保护，要求只有操作系统才能改变地址变换，这是通过防止用户程序更改页表来实现的。可以在操作系统的帮助下实现在进程之间受控制地共享页，页表中的访问位被用来指出用户程序对页进行读访问还是写访问。

如果对于每一次访问，处理器不得不访问主存中的页表来进行转换，这样虚拟存储器的开销将很大，cache 也将失去意义。相反，对于页表，TLB 扮演了地址转换 cache 的角色，利用 TLB 中的变换，将虚拟地址转换为物理地址。

cache、虚拟存储器以及 TLB 都建立在一组共同的原理和策略基础上。下一节讨论这个共同的架构。

**01 理解程序性能** 尽管虚拟存储器能使一个小容量的存储器看起来像大容量的存储器，但二级存储器和主存之间的性能差异意味着，如果一个程序经常访问比它拥有的物理存储器多的虚拟存储器，程序运行速度就会很慢。这样的程序会不断地在存储器和磁盘之间交换页面，称为抖动（thrashing）。抖动的发生将会是灾难，但很少见。如果你的程序产生抖动，那么最简单的解决方式就是让你的程序在一个有着更大存储器的计算机上运行，或者为你的计算机增加存储器。一个复杂的办法是重新检查所使用的算法和数据结构，看看能否改变它的局部性，从而减少程序同时使用的页数。这一组页通常被称为工作集（working set）。

一个更常见的性能问题是 TLB 缺失。由于 TLB 同时只能处理 32～64 个页表项，一个程序很容易会有较高的 TLB 缺失率，因为处理器只能直接访问不到 64×4KiB＝0.25MiB。例如，对于基数排序，TLB 缺失通常是一个挑战。为了缓解这个问题，现在很多计算机体系结构都支持可变的页大小。例如，除了 4KiB 的标准页面，MIPS 硬件还支持 16KiB、64KiB、256KiB、1MiB、4MiB、16MiB、64MiB 和 256MiB 大小的页面。因此，如果一个程序使用大容量的页面，就能直接访问更多主存而不会有 TLB 缺失。

453 令操作系统允许程序选择这些大容量的页面也是一个实际的难题。同样，减少 TLB 缺失更为复杂的方法是重新检查算法和数据结构以减少页面工作集；另外，由于存储器访问对于性能以及 TLB 缺失频率至关重要，所以一些工作集较大的程序已经在这方面做了重新设计。

**01 小测验**

将左边的存储器层次结构组成部分与右边最匹配的说明连线：

| | |
|---|---|
| 1. 一级 cache | a. cache 的 cache |
| 2. 二级 cache | b. 磁盘的 cache |
| 3. 主存 | c. 主存的 cache |
| 4. TLB | d. 页表项的 cache |

## 5.8 存储器层次结构的一般框架

到目前为止，我们已经知道了不同类型的存储器层次结构共用许多原理。尽管存储器层次结构中很多方面都有量的区别，但是决定层次结构如何运作的许多策略和特征在本质上是相同的。图 5-35 给出了存储器层次结构的一些量的特征区别。在本节的剩余部分，我们将讨论存储器层次结构的共同运作方面以及这些方面将如何决定它们的行为。我们通过一系列适用于存储器层次结构两层之间的 4 个问题来研究这些策略，为了简单起见，我们主要使用 cache 中的术语。

| 特征 | 一级 cache 的典型值 | 二级 cache 的典型值 | 页式存储器的典型值 | TLB 的典型值 |
|---|---|---|---|---|
| 块的总容量 | 250 ~ 2 000 | 2 500 ~ 25 000 | 16 000 ~ 250 000 | 40 ~ 1 024 |
| 以 KB 计量的总容量 | 16 ~ 64 | 125 ~ 2 000 | 1 000 000 ~ 1 000 000 000 | 0.25 ~ 16 |
| 块的字节数 | 16 ~ 64 | 64 ~ 128 | 4 000 ~ 64 000 | 4 ~ 32 |
| 缺失代价的时钟周期数 | 10 ~ 25 | 100 ~ 1 000 | 10 000 000 ~ 100 000 000 | 10 ~ 1 000 |
| 缺失率（二级 cache 是全局缺失） | 2% ~ 5% | 0.1% ~ 2% | 0.000 01% ~ 0.000 1% | 0.01% ~ 2% |

图 5-35 计算机中存储器层次结构主要组成部分的关键定量设计参数。本图是这些层次在 2012 年的典型值。值的范围很大，一部分原因是许多值是随着时间的变化而变化的；例如，当 cache 容量变大以克服较高的缺失代价时，块容量也随之增长。图中没有显示的是，服务器处理器中还有三级 cache，容量通常为 2 ~ 8MiB，块数比二级 cache 多很多。三级 cache 使二级 cache 的缺失代价降低到 30 ~ 40 个时钟周期

### 5.8.1 问题 1：一个块可以被放在何处

我们已经看到，可以根据很多机制将块放置到存储器层次的较高层结构中，从直接映射到组相联，再到全相联。就像前面所提到的，这些机制都可以看成是组数和每组块数各不相同的组相联方案的特例：

| 机制 | 名称组数 | 每组块数 |
|---|---|---|
| 直接映射 | cache 中的块数 | 1 |
| 组相联 | cache 中的块数/相联度 | 相联度（一般为 2 ~ 16） |
| 全相联 | 1 | cache 中的块数 |

增加相联度的好处在于它通常能降低缺失率。缺失率的改进来自于减少竞争同一位置而产生的缺失。我们稍后将详细讨论。首先来看能获得多少性能改进。图 5-36 显示了不同的 cache 容量，在相联度从直接映射到八路组相联之间变化的缺失率。最大的改进出现在直接映射变化到两路组相联，缺失率下降了 20% ~ 30%。当 cache 容量增加时，相联度的提高对性能改进作用很小；这是因为大容量 cache 的总的缺失率很低，从而改进缺失率的机会减少，并且由相联度引起的缺失率的绝对改进明显减少。如前所述，相联度增加的潜在缺点是增加了代价以及访问时间。

454 ~ 455

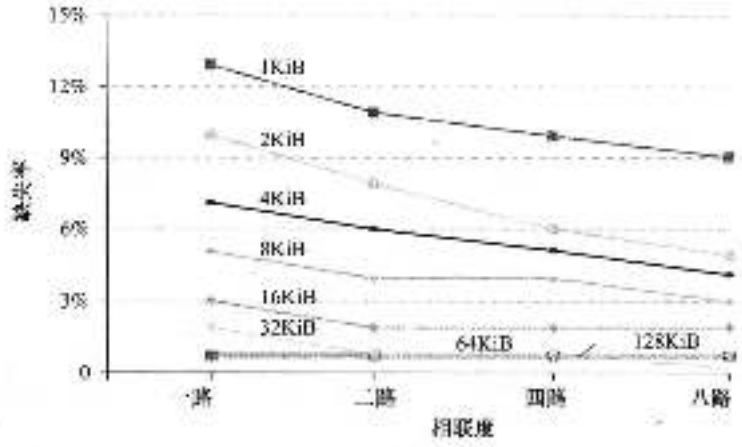



图 5-36 当相联度增加时，8 种不同容量数据 cache 各自的缺失率。从一路（直接映射）到两路组相联变化时获益明显，进一步增加相联度所获得的好处就小一些了（例如，从两路到四路提高了 1% ~10%，而从一路到两路提高了 20% ~30%）。从四路到八路组相联，缺失率的改进更小，它们反而接近于全相联 cache 的缺失率。容量小的 cache 由于其本身缺失率较高，因此从相联度所获得的好处就很明显。图 5-16 解释了这些数据是如何收集的

## 5.8.2 问题 2：如何找到一个块

我们如何选择一个块的存放位置取决于块放置机制，因为它指明了可能存放位置的数量。我们可以把这些机制总结如下：

| 相联度 | 定位方法 | 需要比较的次数 |
|---|---|---|
| 直接映射 | 索引 | 1 |
| 组相联 | 索引组，查找组中元素 | 相联的度 |
| 全相联 | 存放所有 cache 项 | cache 的容量 |
| | 独立的查找表 | 0 |

在存储器层次结构中选择直接映射、组相联还是全相联映射取决于缺失代价和相联度实现代价的权衡情况，包括了时间和额外硬件开销。在片内拥有二级 cache 允许实现更高的相联度，这是因为命中时间不再关键，设计者也不用依靠标准 SRAM 芯片来构建块。除非容量很小，否则 cache 不使用全相联映射方式，在小容量 cache 中，比较器的开销并不是压倒性的因素，其绝对缺失率的改进才是最明显的。

在虚拟存储器系统中，页表是一个独立的映射表，它用来索引存储器。除了表本身需要占用存储资源外，使用索引表还会引起额外的存储器访问。选择全相联映射和额外的页表有以下几个原因：

1）全相联有其优越性，因为缺失的代价非常高。

2）全相联允许软件使用复杂的替换策略以降低缺失率。

3）全映射很容易被索引，而不需要额外的硬件，也不需要进行查找。

因此，虚拟存储系统通常使用全相联映射。

组相联映射通常用于 cache 和 TLB，访问时包括索引和在小组内查找。一些系统使用直接映射的 cache，这是因为访问时间短并且实现简单。访问时间短是因为不需要比较就能找到被请求的块。这样的设计选择取决于许多细节的实现，如 cache 是否集成在片上，实现 cache 的

技术以及 cache 访问时间对处理器时钟周期的重要性。

### 5.8.3 问题3：当 cache 缺失时替换哪一块

在相联的 cache 中发生缺失时，我们必须决定替换哪一块。如果是全相联 cache，所有的块都是被替换的候选者。如果 cache 是组相联的，我们必须在某一组的块中进行选择。当然，直接映射的 cache 的替换很简单，因为只有一个可以替换的候选者。

在组相联或者全相联 cache 中，有两种主要的替换策略：

- 随机法：随机选择候选块，可能使用一些硬件协助实现。例如，对于 TLB 缺失，MIPS 支持随机替换。
- 最近最少使用算法：被替换的块是最久没有被使用过的块。

实际应用中，在相联度不低（典型的是两路到四路）的层次结构中实现 LRU 的代价太高了，这是因为跟踪使用信息的代价很高。尽管对于四路组相联，LRU 通常也是近似实现的——例如，跟踪记录哪一对块是最近最少使用的（需要使用 1 位），然后跟踪记录每对块中哪一块又是最近最少使用的（要求每对使用 1 位）。

对于更高的相联度的层次结构，可以用近似的 LRU 算法，也可以采用随机替换策略。在 cache 中，替换算法是由硬件实现的，这意味着算法应该容易实现。随机替换算法用硬件很容易实现，而对于两路组相联的 cache，使用随机替换算法的缺失率要比 LRU 替换算法的缺失率高 1.1 倍。随着 cache 变得更大，所有替换策略的缺失率都下降了，绝对差别也变小了。事实上，有时候，随机替换算法的性能比用硬件简单实现的近似 LRU 算法的性能还要好。

在虚拟存储器中，LRU 的一些形式都是近似的，因为当缺失代价很大时，缺失率即使只有微小的降低都是很重要的。通常提供引用位或者其他等价的功能使操作系统更方便地追踪一组最近最少使用的项。由于缺失的代价特别高，并且相对来说不经常发生，主要用软件来近似这项信息的做法是可行的。

### 5.8.4 问题4：写操作如何处理

对任何存储器层次结构来说，一个关键的问题是如何处理写操作。我们已经看到了两种基本选项：

- 写直达：信息被同时写到 cache 的块和存储器层次结构较低层的块中（对 cache 来说是主存）。5.3 节中的 cache 使用这个机制。
- 写回：信息仅仅写到 cache 中的块。被改写的块只有在它被替换时才写回到存储器层次结构的较低层中。虚拟存储器系统通常采用写回策略，原因在 5.7 节中讨论过。

写回和写直达策略有其各自的优点，写回的主要优点如下：

- 处理器可以以 cache 而不是存储器能接收的速度写单个的字。
- 多次写同一块中的字只需对存储器层次结构较低层进行一次写操作。
- 当块被写回时，由于写一整块，系统可以充分利用高带宽传输。

写直达的优点如下：

- 缺失比较简单，缺失代价也较小，这是因为不需要把整个块写回到较低层存储系统中。
- 尽管为了可行性，写直达的 cache 需要一个写缓冲区，然而写直达还是比写回更易于实现。

在虚拟存储器系统中，由于写到存储器层次结构的较低层（磁盘）的延迟很大，因此只有写回策略是可行的。尽管允许存储器的物理、逻辑宽度更宽，并对 DRAM 采用突发模式，然

457

而处理器产生写操作的速度通常还是超过存储系统可以处理它们的速度。因此，现在最低一级的 cache 通常采用写回策略。

**重点** cache、TLB 和虚拟存储器可能一开始看起来非常不同，但是它们都基于相同的两个定位原理，并且可以通过这 4 个问题的各自答案来理解。

问题 1：一个块可以被放在何处？

答：一个位置（直接映射），一些位置（组相联），或者是任何位置（全相联）。

问题 2：如何找到一个块？

答：有 4 种方法：索引（在直接映射的 cache 中），有限的检索（在组相联的 cache 中），全部检索（在全相联的 cache 中）和专用查找表（在页表中）。

问题 3：当 cache 缺失时替换哪一块？

答：通常是最近最少使用的块或者是随机选取的一块。

问题 4：写操作如何处理？

[458] 答：层次结构中的每一层都可以使用写直达或者写回策略。

## 5.8.5 3C：一种理解存储器层次结构行为的直观模型

在这一节中，我们来看一个模型，通过它能够很好地洞察存储器层次结构中引起缺失的原因以及层次结构的变化对缺失的影响。我们从 cache 方面来解释这个观点，尽管这个观点对其他层次也都直接适用。在这个模型中，所有的缺失被分成下面三类（3C 模型（three Cs model））：

- **强制缺失**（compulsory miss）：对从没有在 cache 中出现的块第一次进行访问引起的缺失。也称为**冷启动缺失**（cold-start miss）。
- **容量缺失**（capacity miss）：由于 cache 容纳不了一个程序执行所需要的所有块而引起的 cache 缺失，当某些块被替换出去，随后再被调入时，将发生容量缺失。
- **冲突缺失**（conflict miss）：在组相联或者直接映射的 cache 中，多个块竞争同一个组时而引起的 cache 缺失。冲突缺失在直接映射或组相联 cache 中存在，而在同样大小的全相联 cache 中不存在。这种 cache 缺失也称为**碰撞缺失**（collision miss）。

⊃ 3C 模型：将所有的 cache 缺失都归为三种类型的 cache 模型，三类分别为：强制缺失、容量缺失和冲突缺失。同其三类名称的英文单词首字母均为 c 而得名。

⊃ 强制缺失：也称为冷启动缺失。对没有在 cache 中出现过的块第一次访问时产生的缺失。

⊃ 容量缺失：由于 cache 在全相联时都不可能容纳所有请求的块而导致的缺失。

⊃ 冲突缺失：也称为碰撞缺失。在组相联或者直接映射 cache 中，很多块为了竞争同一个组导致的缺失。这种缺失在使用相同大小的全相联 cache 中是不存在的。

图 5-37 显示了缺失率是如何按照引起的原因被分为三种的。改变 cache 设计中的某一方面就能直接影响这些缺失的原因。冲突缺失是因为争用同一个 cache 块而引起的，因此提高相联度就可以减少冲突缺失。然而，提高相联度会延长访问时间，导致整个性能的降低。

容量缺失可以简单地通过增大 cache 容量来减少；的确，多年来二级 cache 的容量总是在不断地增加。当然，在增大 cache 的同时，我们也必须注意访问时间的增长，这将导致整体性能的降低。因此，尽管二级 cache 也在增大，但是增大地非常缓慢。

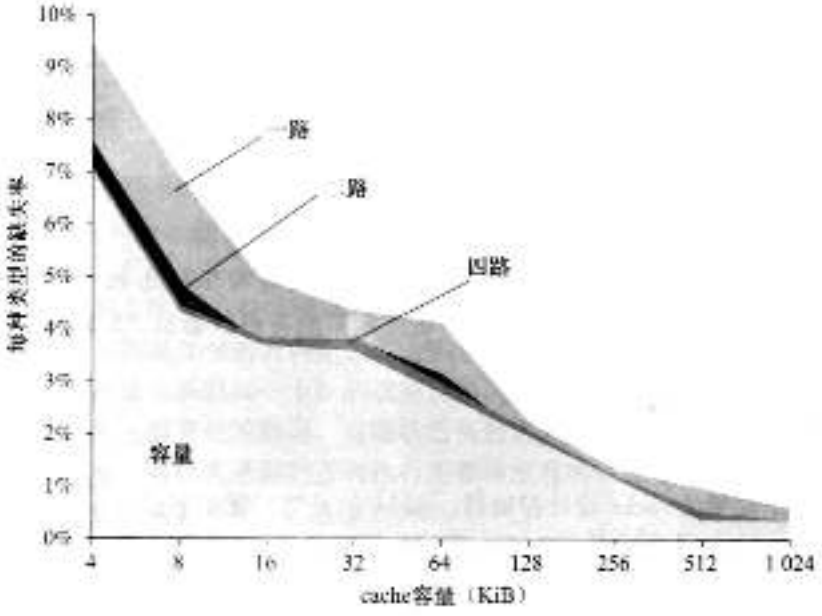



图 5-37 根据缺失原因缺失率被分成三种。这幅图显示了不同容量 cache 的总缺失率及其组成部分。数据与图 5-36 出自同一来源，都是由 SPEC CPU 2000 整型和浮点基准程序测试得到的。强制缺失部分只占 0.006%，在图中看不出来。下一部分是容量缺失，取决于 cache 的容量。冲突缺失部分既取决于相联度，又取决于 cache 的容量。图中给出了相联度从一路到八路的冲突缺失率。在每种情况下，当相联度从下一个更高度变化到标记地方的相联度时，标记地方对应缺失率的增加。例如，标有两路的部分说明当 cache 相联度从四路变化到两路时缺失增加。因此，同样大小的直接映射 cache 和全相联 cache 的缺失率的差别由标记着八路、四路、两路和一路的各部分之和给出。八路和四路之间变化太小，以至于在图中很难看出

由于强制缺失是对块的第一次访问产生的，因此，对 cache 系统来说，减少强制缺失次数最主要的方法是增加块的大小。由于程序将由较少的 cache 块组成，因此这就减少了对程序每一块都要访问一次的情况下的总的访问次数。如前所述，块容量增加太多可能对性能产生负面影响，因为缺失代价会增长。 [459]

将缺失分成 3C 是个有用的定性模型。在实际 cache 设计中，许多设计的选择是相互影响的，改变 cache 的一个特征通常会影响另一些缺失率的组成部分。尽管存在这些缺点，3C 模型对于观察 cache 设计的性能来说仍是一种有效的方法。

**重点** 存储器层次结构设计所面临的挑战在于：任何一个改进缺失率的设计同时也可能对整体性能产生负面的影响，如图 5-38 所示。正面与负面作用的结合就使得存储器层次结构的设计令人关注。

| 设计变化 | 对缺失率的影响 | 可能对性能产生的负面影响 |
|---|---|---|
| 增加 cache 容量 | 减少了容量缺失 | 可能增加访问时间 |
| 提高相联度 | 由于减少了冲突缺失，因此降低了缺失率 | 可能增加访问时间 |
| 增加块的容量 | 由于空间局部性，因此对很宽范围内变化的块大小，都能降低缺失率 | 增加缺失代价，块太大还会增加缺失率 |

图 5-38 存储器层次结构设计面临的挑战

**小测验**

下面哪些表述（如果有）是正确的？

1. 没有减少强制缺失的方法。
2. 全相联 cache 中没有冲突缺失。
3. 在减少缺失方面，相联度比容量更为重要。

## 5.9 使用有限状态机来控制简单的 cache

就像我们在第 4 章中对单周期、流水线数据通路实现控制一样，现在我们可以实现对 cache 的控制。本节从定义一个简单的 cache 开始，随后对有限状态机（finite-state machine，FSM）进行介绍。最后介绍了这个简单 cache 的控制器的有限状态机。5.12 节用一种新的硬件描述语言更深入地介绍了 cache 和控制器。

### 5.9.1 一个简单的 cache

460 ~ 461

我们将为一个简单的 cache 设计控制器。cache 的关键特征如下：

- 直接映射的 cache。
- 写回机制，采用写分配策略。
- 块大小为 4 个字（16 字节或者 128 位）。
- cache 大小为 16KiB，因此它能容纳 1 024 个块。
- 32 字节地址。
- cache 中每个块包含一个有效位和写入位。

根据 5.3 节，我们可以计算出 cache 的地址域：

- cache 索引位为 10 位。
- 块偏移为 4 位。
- 标记位为 32 - (10 + 4) = 18 位。

处理器和 cache 之间的信号为：

- 1 位读/写信号。
- 1 位有效信号，指示是否有一个 cache 操作。
- 32 位地址。
- 32 位数据（从处理器到 cache）。
- 32 位数据（从 cache 到处理器）。
- 1 位准备信号，指示 cache 操作完成。

存储器和 cache 之间的接口与处理器和 cache 之间一样有相同的域，除了数据域这里是 128 位宽。如今，一般的微处理器都有额外的存储器位宽，在处理器中可以处理 32 位或 64 位的字，而 DRAM 控制器通常是 128 位。为了简化设计，可以使 cache 块匹配 DRAM 的位宽。下面是一些信号：

- 1 位读/写信号。
- 1 位有效信号，指示是否有一个存储器操作。
- 32 位地址。
- 128 位数据（从 cache 到存储器）。
- 128 位数据（从存储器到 cache）。
- 1 位准备信号，指示存储器操作完成。

请注意，到存储器的接口并没有固定的周期数。我们假设当存储器读或写完成后，存储器控制器通过准备信号来通知 cache。

在介绍 cache 控制器之前，我们需要回顾一下有限状态机，它支持控制一个花费多个时钟周期的操作。

462

## 5.9.2 有限状态机

为了给单周期的数据通路设计控制单元，我们使用一组真值表，根据指令的分类来指定控制信号的设置。对于 cache，由于操作可以是一系列的步骤，因此控制变得更加复杂。对 cache 的控制既要指定在任何步骤中信号的设置，又要依次指出下一步的步骤。

最常见的多步控制方法基于**有限状态机**（finite-state machine），通常以图形化表示。有限状态机由一组状态以及状态改变的方向组成。方向由**下一状态函数**（next-state function）来定义，它将当前的状态和输入映射到一个新的状态。当我们使用有限状态机控制时，每个状态还要在当时的状态下指出一组有效的输出。有限状态机的实现通常假定那些没有明确置为有效的输出是无效的。类似地，对数据通路的正确执行需要将没有明确设置为有效的信号设置成无效状态，而不是对信号置位采取不关心的态度。

- 有限状态机：由一组输入和输出，以及下一状态函数和输出函数组成的时序逻辑函数。下一状态函数将当前状态和当前输入映射为一个新的状态，输出函数将当前状态和当前输入映射为一组确定的输出。
- 下一状态函数：根据当前状态及当前输入来确定有限状态机下一状态的组合函数。

多路选择控制略微有一些不同，它们从输入（0 或 1）中选择一个。因此，在有限状态机中，我们总是指定我们关心的所有多路选择控制的设置。当我们使用逻辑实现有限状态机时，设置为 0 的控制可能就是默认值，因此不需要任何门电路。一个简单的有限状态机的例子在附录 B 中给出，如果不熟悉有限状态机的概念，在继续学习之前，读者可能需要花一些时间来研究附录 B。

一个有限状态机的实现包括：一个保持当前状态的临时寄存器和一个组合逻辑，组合逻辑用来决定有效的数据通路信号和下一状态。图 5-39 显示了可能的实现效果图。附录 D 详细介绍了使用这个结构如何实现有限状态机。在 B.3 节中，一个有限状态机的组合逻辑由 ROM（read-only memory，只读存储器）或 PLA（programmable logic array，可编程逻辑阵列）来实现。（附录 B 中对这些逻辑单元也进行了描述。）

**精解** 注意，这是一个阻塞式 cache，因此处理器必须等到 cache 处理完请求之后才能继续执行。5.12 节中将会讲述另外一种称为非阻塞式 cache 的结构。

**精解** 本书中的有限状态机的类型被称作 Moore 型有限状态机，以 Edward Moore 来命名。它的标识特征是输出仅仅取决于当前的状态。对于 Moore 型有限状态机，标记着组合控制逻辑的逻辑单元可以被分成两部分：一部分包括控制输出，并且仅有状态输入；另一部分仅包含下一状态输出。

另一种状态机是 Mealy 型有限状态机，以 George Mealy 命名。Mealy 型有限状态机的输出取决于输入和当前的状态。Moore 型有限状态机潜在的实现优势在于速度和控制单元的规模。由于在时钟周期开始就需要控制输出，而该输出与输入无关，仅仅取决于当前的状态，因此有助于速度的提升。在附录 B 中，用逻辑门就可以实现这种有限状态机，因而可以很明显地看出它在规模上的优势。Moore 型有限状态机潜在的缺点是它可能需要额外的状态。例如，在两个状态序列中仅有一个状态不同的情况下，Mealy 状态机会通过使用输出依赖输入的方法将状态统一。

463

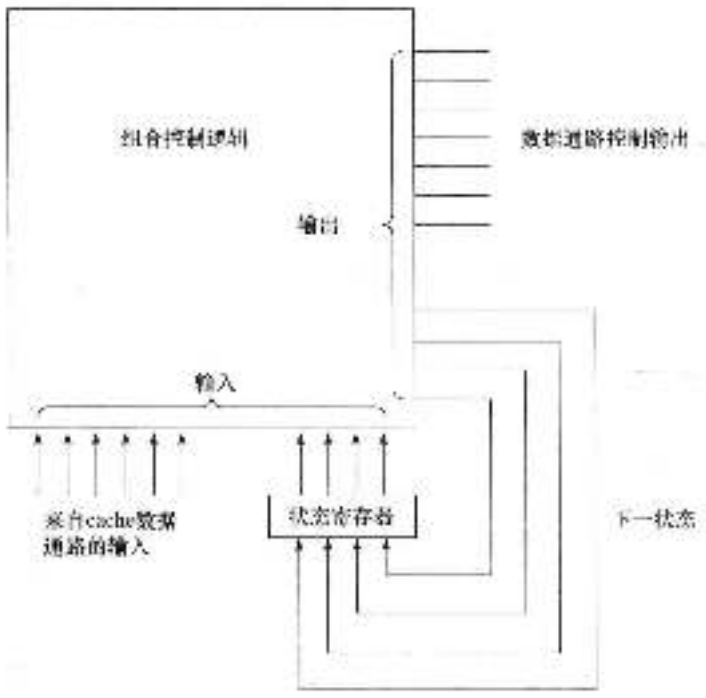



图5-39 典型的有限状态机控制器由一个组合逻辑和一个保存当前状态的寄存器来实现。组合逻辑的输出是下一个状态号以及当前状态的有效控制信号。组合逻辑的输入是当前的状态以及用来决定下一状态的一些输入。注意到，在本章所使用的有限状态机中，输出仅由当前状态来决定，而与输入无关。对此，稍后更详细地进行了解释

## 5.9.3 一个简单的cache控制器的有限状态机

图5-40是简单cache控制器的4个状态：

- 空闲：这个状态等待从处理器发出有效的读/写请求，使得有限状态机转移到标记比较的状态。
- 标志比较：如名称所示，这个状态主要检测该读/写请求是命中还是缺失。地址的索引部分用来选择比较用的标记。如果它的有效位和地址的标记部分与标记位相匹配，则命中。这时，或者从选中的字中读出数据，或者将数据写入选中的字。随后cache准备信号被置位。如果是写操作，还要将脏位设置为1。注意，如果是写命中，还要设置有效位和标记域；这些设置看起来并不需要，却还是要设置，因为标记使用单独的存储器，因此，改变写入位时，我们也需要改变有效位和标记域。如果请求命中并且cache块有效，有限状态机返回到空闲状态。发生一次缺失时首先要更新cache标记，随后，如果这个位置的块的写入位为1，则转入写回状态；如果写入位为0，则进入分配状态。
- 写回：这个状态根据标记和cache索引组合的地址，将128位的块写回存储器。我们继续等留在该状态等待存储器返回准备信号。当存储器写回完成时，有限状态机进入分配状态。
- 分配：新的块从存储器中取回。我们继续停留在该状态等待从存储器返回准备信号。当存储器读操作完成时，有限状态机转入标记比较状态。尽管我们可以转移到一个新的状态来完成操作，而不再使用标记比较状态，但是这个操作中有很多重复，包括当访问是写操作时更新块中恰当的字。

464
-
465

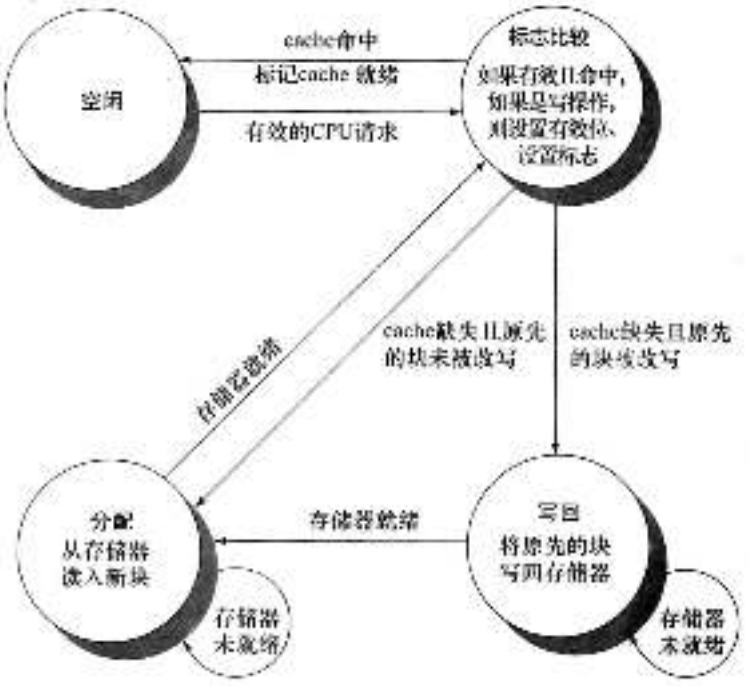



图 5-40 简单控制器的 4 个状态

这个简单的模型可以很方便地扩展到多个状态以改进性能。例如，标记比较状态在一个单独的时钟周期里既要比较，又要读/写 cache 数据。通常，比较和 cache 访问被放在分离的状态中，以改进时钟周期。另一个优化是增加一个写缓冲，这样我们就可以先保存旧块，然后再读出新的块。这样，当一个旧块缺失时，处理器就不用等待两次存储器访问。随后，cache 将从写缓冲器中将脏块写回，同时处理器正在处理被请求的数据。

在 5.12 节将对有限状态机进行更深入的研究，用硬件描述语言描述了整个控制器，并显示了这个简单 cache 的方框图。

## 5.10 并行与存储器层次结构：cache 一致性

多核多处理器意味着在单芯片上有多个处理器，这些处理器可能会共享一个公共的物理地址空间。cache 共享数据带来了一个新的问题，由于两个不同的处理器所保存的存储器视图是通过各自的 cache 得到的，如果没有其他的防范措施，两个处理器可能分别得到两个不同的值。图 5-41 解释了这个问题，并且说明了为什么两个不同的处理器对存储器相同位置进行操作会得到不同的值。这个问题通常称为 cache 一致性问题。

| 时间 | 事件 | CPU A 的 cache 内容 | CPU B 的 cache 内容 | 存储器位置 X 的内容 |
|---|---|---|---|---|
| 0 | | | | 0 |
| 1 | CPU A 读 X | 0 | | 0 |
| 2 | CPU B 读 X | 0 | 0 | 0 |
| 3 | CPU A 向 X 写入 1 | 1 | 0 | 1 |

图 5-41 cache 一致性问题：两个处理器（A 和 B）对同一个存储器位置 X 进行读写操作。我们假设最初两个 cache 中都不包含该变量并且 X 的值为 0。假设是写直达 cache；如果是写回 cache 则会带来额外的更加复杂的情况。当 X 的值被 A 改写后，A 的 cache 和存储器中的副本都被了更新，但是 B 的 cache 没有，如果 B 读 X，得到的值为 0

一般情况下，如果在一个存储器系统中读取任何一个数据项的返回结果总是最近写入的值，那么可以认为该存储器具有一致性。这个定义尽管看起来是正确的，但仍很模糊而且过于简单；实际情况复杂得多。这个简单的定义包括了存储器系统行为的两个不同方面，它们对于编写正确的共享存储程序是至关重要的。第一个方面称为一致性（coherence），它定义了读操作可以返回什么样的数值。第二个方面称为连贯性（consistency），它定义了写入的数据什么时候才能被读操作返回。

首先来看一致性。如果一个存储系统满足如下条件，那么认为该存储系统是一致的：

1）处理器P对位置X的写操作后面紧跟着处理器P对X的读操作，并且在这次读操作和写操作之间没有其他处理器对X进行写操作，这时读操作总是返回P写入的数值。因此，在图5-41中，如果CPU A在时间3之后读X，它将得到数值1。

2）在其他处理器对X的写操作后，处理器P对X执行读操作，这两个操作之间有足够的间隔并且没有其他处理器对X进行写操作，这时，读操作返回的是写入的数值。因此，在图5-41中，我们需要一个机制，以便在时间3，CPU A向存储器地址X写入数据1之后，CPU B的
466 cache中的数值0被数值1所替换。

3）对同一个地址的写操作是串行执行的（serialized）；也就是说，任何两个处理器对同一个地址的两个写操作在所有处理器看来都有相同的顺序。例如，如果在时间3之后，CPU B又向存储器地址X中写入2，那么处理器绝不会从该地址中先读出2再读出1。

第一个性质保证了程序的顺序——即使在单处理器中也要保证这个性质。第二个性质定义了存储器的一致性意味着什么：如果一个处理器总是读到旧的数值，我们就认为这个存储器是非一致性的。

写操作串行化的要求更加细致，但也同等重要。假如我们没有将写操作串行化，处理器P1写入地址X之后，紧跟着处理器P2也会写入地址X。写操作串行化保证了每个处理器都能在某个时间看到P2写入的结果。如果没有将写操作串行化，就会出现一些处理器先看到P2写入的结果再看到P1写入的结果，从而可能保留了P1写入的数值。避免这种情况最简单的方法就是保证对同一个地址的写操作在所有处理器看来都具有相同的顺序，这个性质称为写串行化（write serialization）。

## 5.10.1 实现一致性的基本方案

在支持cache一致性的多处理器系统中，cache提供共享数据的迁移（migration）和复制（replication）。

- 迁移：数据项可以移入本地cache并以透明的方式使用。迁移不但减少了访问远程共
467 享数据项的延迟，而且减少了对共享存储器带宽的需求。
- 复制：当共享数据被同时读取时，cache在本地对数据项做了备份。复制减少了访问延迟和读取共享数据时的竞争现象。

对这种迁移和复制的支持对于访问共享数据的性能来说是至关重要的，因此许多多处理器引入硬件协议来维护cache一致性。这个用于维护多个处理器一致性的协议称为cache一致性协议（cache coherence protocol）。实现cache一致性协议的关键在于跟踪所有共享数据块的状态。

最常用的cache一致性协议是监听（snooping）协议。每个含有物理存储器中数据块副本的cache还要保留该数据块共享状态的副本，但是并不集中地保存状态。cache可以通过一些广播媒介（总线或者网络）访问，所有的cache控制器对媒介进行监视或者监听，来确定它们是否含有总线或者交换机上请求的数据块副本。

在后面章节我们将介绍用共享总线实现基于监听的 cache 一致性方法，任何可以向所有处理器广播 cache 缺失的通信媒介都可以用来实现基于监听的一致性机制。这种向所有 cache 广播的方法使得监听协议的实现变得简单，但是也限制了其可扩展性。

## 5.10.2 监听协议

实现一致性的一种方法是：在处理器写数据之前，保证该处理器能独占地访问该数据项。这种协议称为写无效协议（write invalidate protocol），因为它在执行写操作的时候令其他副本无效。独占访问确保了写操作执行时不存在其他可读或可写的数据项副本：cache 中该数据项的其他所有副本都是无效的。

图 5-42 给出了一个基于监听总线的写无效协议的例子，其中 cache 使用写回机制。为了说明这个协议如何保证一致性，我们令写操作后面紧跟着其他处理器执行读操作的情况：由于写操作需要独占访问，执行读操作的处理器中保存的任何副本就要被置无效（协议因此得名）。因此，当执行读操作时，在 cache 中发生缺失，cache 需要取回新的数据副本。对于写操作，我们要求执行写操作的处理器可以独占访问，以防止其他处理器同时执行写操作。如果两个处理器试图同时对同一个数据项进行写操作，它们中的一个会在竞争中获胜，这就使得另一个处理器的副本被置为无效。竞争失败的处理器要完成写操作，就必须取得新的数据副本，这个副本中已经包含了更新后的数据。因此，这个协议也强制了写操作的串行化。

468

| 处理器动作 | 总线动作 | CPU A 的 cache 内容 | CPU B 的 cache 内容 | 存储器中位置 X 的内容 |
|---|---|---|---|---|
| | | | | 0 |
| CPU A 读 X | X 在 cache 中缺失 | 0 | | 0 |
| CPU B 读 X | X 在 cache 中缺失 | 0 | 0 | 0 |
| CPU A 向 X 写 1 | 令 X 无效 | 1 | | 0 |
| CPU B 读 X | X 在 cache 中缺失 | 1 | 1 | 1 |

图 5-42 以对单个 cache 块 X 读写的过程为例（采用写回机制），说明监听总线上执行无效协议的过程。我们假设最初两个 cache 中都没有 X，而在存储器中 X 的值为 0。CPU 和 X 的存储器内容是处理器和总线动作都完成后的数值。空格表示没有动作或者没有存放副本。当 B 发生第二次缺失时，CPU A 回应，同时取消来自存储器的响应。随后，B 的 cache 和 X 的存储器内容都得到更新。这种当块共享时对存储器进行更新的方法简化了协议，但是可能只有当块被替换时才有跟踪所有权并强制写回。这就需要引入一个被称为“所有者”（owner）的额外状态，它表明块可以被共享，但是当块被改变或是替换时，由所有者处理器负责更新其他处理器和存储器

**硬件/软件接口** 一种观点是：块大小对 cache 一致性起着重要作用。以对一个 cache 监听为例，cache 的块大小为 8 个字，两个处理器可以对块中的一个字进行读/写操作。多数协议会在两个处理器之间交换整个块，因此增加了所需要的一致性带宽。

大的块同样会引起所谓的假共享（false sharing）：当两个不相关的共享变量存在相同的 cache 块中时，尽管每个处理器访问的是不同的变量，但是在处理器之间还是将整个块进行交换。因此，程序员和编译器需要谨慎放置数据以避免发生假共享。

假共享：当两个不相关的共享变量放在相同的 cache 块中时，尽管每个处理器访问的是不同的变量，但是在处理器之间还是将整个块进行交换。

**解答** 尽管前面给三个属性已经论充分保证一致性，但是何时能看见写的值，这个问题同样很重要。让我们来看看为什么。注意到在图5-41中，我们不能要求对X的读操作立刻能看见其他处理器对X执行写操作的值。例如，假设一个处理器对X的写操作稍稍先于另一个处理器对X的读操作，这样就不能保证读操作返回的数值是被写入数据，甚至紧邻一刻，被写的数据可能甚至还没有离开处理器。连贯性仅仅是定义了写数据何时能被读操作看见。

469

我们做下面两个假设：第一，直到所有处理器看见写操作的结果，这个写操作才能完成（没有完成之前可以允许下一个写操作发生）；第二，处理器不能改变与存储器读值相关的写操作的次序。这两个条件意味着：如果处理器在写位置X之后写写位置Y，那么，任何处理器在看到Y的新值之前必须看见X的新值。这些限制条件允许处理器对读操作可以重新排序，但是迫使处理器以程序执行的顺序完成写操作。

**解答** 由于输入操作可在不改变cache内容的情况下改变存储器内容，另外，在写回cache中，输出操作需要最新的内存储器内容，因此在单处理器系统中也存在I/O和cache之间一致性问题，这与多处理器间cache一致性问题相同。cache一致性问题对于多处理器和I/O（见第6章）来说，尽管原因相同，但是却有不同的特性，从而影响了解决方法。为了平衡少量有多个数据副本的I/O不同——只要可能大就应该避免——程序运行在多个处理器上时，cache中通常都有相同数据的副本。

**解答** 除了分布地保存共享块状态的监听式cache一致性协议，基于目录的cache一致性协议将物理存储器的共享块的状态存放在一个地点，称之为目录（directory）。尽管基于目录的一致性比监听式一致性协议升级略高一些，但是这种方法可以减少cache之间的通信，并且因此可以扩展更多的处理器。

## 5.11 并行与存储器层次结构：冗余廉价磁盘阵列

本节内容在网站上，讲述了如何采用许多块磁盘并行工作来提高吞吐率，该技术是冗余廉价磁盘阵列（Redundant Arrays of Inexpensive Disks，RAID）产生的灵感所在。然而，RAID技术真正流行的原因在于其通过采用适当数量的冗余磁盘来提高可靠性。本节讲述了不同RAID级别在性能、开销和可靠性等方面的差别。

## 5.12 高级内容：实现cache控制器

本节内容在网站中，介绍了如何实现cache的控制，就像我们在第4章中实现对单周期、流水的数据通路的控制一样。这一节开始介绍了有限状态机以及在简单的数据cache中实现cache控制器，包括用硬件描述语言来描述cache控制器。随后详细介绍了一个cache一致性协议的实例以及实现的难点。

470

## 5.13 实例：ARM Cortex-A8和Intel Core i7的存储器层次结构

本节将讲述第4章中提到的两种微处理器（ARM Cortex-A8和Intel Core i7）的存储器层次。本节内容基于《计算机体系结构：量化研究方法》第5版的2.6节。

图5-43总结了这两种处理器的地址尺寸和TLB。注意，A8中包含了2个具有32位虚拟地址空间和物理地址空间的TLB。而Core i7中包含了3个具有48位虚拟地址空间和44位物理地址空间的TLB。虽然Core i7中的64位寄存器能够支持更大的虚拟地址空间，但是没有软件需要如此大的空间，48位的虚拟地址不但缩小了页表的踪迹（footprint），也简化了TLB的硬件。

| 特点 | ARM Cortex-A8 | Intel Core i7 |
|---|---|---|
| 虚拟地址 | 32 位 | 48 位 |
| 物理地址 | 32 位 | 44 位 |
| 页表大小 | 可变：4，16，64KiB，1，16MiB | 可变：4KiB，2/4MiB |
| TLB 组织 | 1 个指令 TLB 和 1 个数据 TLB<br>两个 TLB 均为全相联，32 个入口，<br>轮转替换策略<br>硬件处理 TLB 缺失 | 每核 1 个指令 TLB 和 1 个数据 TLB<br>两个 L1 TLB 均为 4 路组相联，<br>LRU 替换策略<br>L1 I-TLB 对于小尺寸页面有 128 个入口，每线程对于大页面有 7 个入口<br>L1 D-TLB 对于小页面有 64 个入口，大页面有 32 个入口<br>L2 TLB 4 路组相联，LRU 替换策略<br>L2 TLB 有 512 个入口<br>硬件处理 TLB 缺失 |

图 5-43 ARM Cortex-A8 和 Intel Core i7 920 的地址转换和 TLB 硬件。两个处理器均支持用于操作系统或映射为帧缓冲器的大页面。大页面技术避免了将一个对象映射到多个入口的情况

图 5-44 给出了它们的 cache。需要注意的是 A8 只有 1 个核，而 Core i7 中有 4 个核。两个处理器的 L1 指令 cache 组织结构相同，容量均为 32KiB，都是 4 路组相联，cache 块均为 64 字节。A8 的数据 cache 和指令 cache 相同，而 Core i7 把相联度提高到了 8 路。虽然 A8 L2 cache 容量可以在 128KiB ~ 1MiB 间变化，而 Core i7 的 L2 cache 容量固定在 256KiB。但是两个处理器的 L2 cache 都采用 64 字节的 cache 块。由于 Core i7 用于服务器，因此它也提供了共享的片上 471
L3 cache，其容量与核的数目相关。对于 4 核而言，L3 cache 容量为 8MiB。

| 特点 | ARM Cortex-A8 | Intel Nehalem |
|---|---|---|
| L1 cache 组织 | 数据指令分离 cache | 数据指令分离 cache |
| L1 cache 容量 | 数据/指令 cache 均为 32KiB | 每核的数据/指令 cache 均为 32KiB |
| L1 cache 相联度 | 4 路（I），4 路（D）组相联 | 4 路（I），8 路（D）组相联 |
| L1 替换策略 | 随机 | 近似 LRU |
| L1 块大小 | 64 字节 | 64 字节 |
| L1 写策略 | 写回，按写分配（?） | 写回，不按写分配 |
| L1 命中时间（load） | 1 个时钟周期 | 4 个时钟周期，流水执行 |
| L2 cache 组织 | 统一（指令和数据） | 每个核统一（指令和数据） |
| L2 cache 容量 | 128KiB ~ 1MiB | 256KiB（0.25MiB） |
| L2 cache 相联度 | 8 路组相联 | 8 路组相联 |
| L2 替换策略 | 随机（?） | 近似 LRU |
| L2 块大小 | 64 字节 | 64 字节 |
| L2 写策略 | 写回，按写分配（?） | 写回，按写分配 |
| L2 命中时间 | 11 个时钟周期 | 10 个时钟周期 |
| L3 cache 组织 | — | 统一（指令和数据） |
| L3 cache 容量 | — | 8MiB，共享 |
| L3 cache 相联度 | — | 16 路组相联 |
| L3 替换策略 | — | 近似 LRU |
| L3 块大小 | — | 64 字节 |
| L3 写策略 | — | 写回，按写分配 |
| L3 命中时间 | — | 35 个时钟周期 |

图 5-44 ARM Cortex-A8 和 Intel Core i7 920 的 cache

在 A8 和 Core i7 中，cache 设计者面临的一个巨大挑战是要支持每个时钟周期执行一条以上的访存指令。通常的一种做法是将 cache 分成多个体（bank），从而在不发生体冲突时，能够对多个体并行进行访问。该技术与 DRAM 中的存储体间交叉类似（见 5.2 节）。

Core i7 采用了另外一些优化技术来降低缺失开销。第一种是请求字优先策略。另外，在 cache 缺失时，继续执行访问数据 cache 的指令，这是设计者在设计乱序执行处理器时，为了隐藏 cache 缺失开销通常采用的技术，称为**非阻塞 cache**（nonblocking cache）。它们实现了无阻塞的两个特点：缺失命中（hit under miss）允许在缺失期间有其他的 cache 命中；缺失情况下的缺失（miss under miss）允许有多个未解决的 cache 缺失。这两者中第一个致力于用其他工作来隐藏一部分缺失延迟，而第二个的目标在于重叠两个不同缺失的延迟。

⇨ 非阻塞 cache：在处理器处理前面的 cache 缺失时仍可正常访问的 cache。

要重叠多个未解决的缺失的大部分缺失时间，需要一个高带宽的存储系统来并行地处理多个缺失。在个人移动设备中，存储器只能获得这项功能的有限的益处，但是大型服务器和多处理器通常拥有的存储系统能并行处理不止一个缺失。

[472]

Core i7 采用了数据访问预取技术，在数据缺失前，根据缺失数据的特点来预测下次数据访问的地址，并使用该地址进行数据预取。该技术在访问循环中的数组时非常有效。

这些芯片的存储器层次非常复杂，且芯核上很大一部分用作 cache 和 TLB。这些都是为了解决处理器运行和存储访问速度之间的巨大差异的结果。

## A8 和 Core i7 存储器层次的性能

对 Cortex-A8 的存储器层次进行仿真时，L2 cache 组织为 1MiB 八路组相联结构，使用整数的 Minnespec 基准测试程序。如第 4 章所述，Minnespec 基准测试程序由 SPEC2000 基准测试程序组成，但是使用了不同的输入集来将运行时间减少几个数量级。虽然使用小输入集没有改变指令的组合情况，但是影响了 cache 行为。例如，在 SPEC2000 中访存最为密集的稀数程序 mcf 中，对于 32KiB 的 cache 来说，Minnespec 的 cache 缺失率只有完整版本 SPEC2000 的 65%。而对于 1MiB 的 cache，该数据只有 1/6。因此，不能将 Minnespec 和 SPEC2000 之间进行比较，更不能和用于 Core i7 的更大的 SPEC2006 比较。Core i7 在 SPEC2006 上的仿真结果如图 5-47 所示。然而，像在第 4 章中一样，在考察 L1 和 L2 cache 缺失率和整体 CPI 的相对影响时，这些数据非常有用。

对于这些基准测试程序而言，A8 的指令 cache 缺失率非常低，即使对于 L1 cache 也是如此：大部分程序的 L1 指令 cache 的缺失率接近于 0，所有程序的 L1 指令 cache 的缺失率在 1% 以下。其原因主要是 SPEC 程序属于计算密集型，且四路组相联的 cache 组织结构消除了大多数的块冲突。图 5-45 给出了 A8 的数据 cache 的结果，其 L1 和 L2 的缺失率都比较高。1GHz 的 Cortex-A8 的 L1 缺失率的开销为 11 个时钟周期，L2 缺失率的开销为 60 个时钟周期。图 5-46 给出了每次数据访问的平均缺失开销。

图 5-47 给出了基准测试程序 SPEC2006 在 Core i7 上运行时的 cache 缺失率。一级指令 cache 的缺失率在 0.1% ~ 1.8% 之间，平均值比 0.4% 高一点点。该缺失率与基准测试程序 SPEC2006 在其他研究中的缺失率一致。一级数据 cache 的缺失率在 5% ~ 10%，有时高一些，二级和三级 cache 的重要性就显而易见了。对于二级 cache 而言，缺失时的访存开销在 100 个时钟周期以上，且缺失率为 4%，因此三级 cache 非常关键。假定有一半的指令是取数或存数指令，如果没有二级 cache，则二级 cache 缺失将导致 CPI 增加 2 倍！1% 的三级数据 cache 缺失率仍然有点高，但是比二级 cache 缺失率低 4 倍，比一级 cache 缺失率低 6 倍。

[473]

**精解** 由于推测执行有时会推测错误（见第 4 章），有一些对一级数据 cache 进行访问的取数或存数指令最终没有执行，则 5-45 中的数据是对所有数据请求的统计，包括那些最

终被取消的访问。真正完成的数据访问的缺失率要高 1.6 倍（一级数据 cache 平均缺失率为 95% 比 5.9%）。

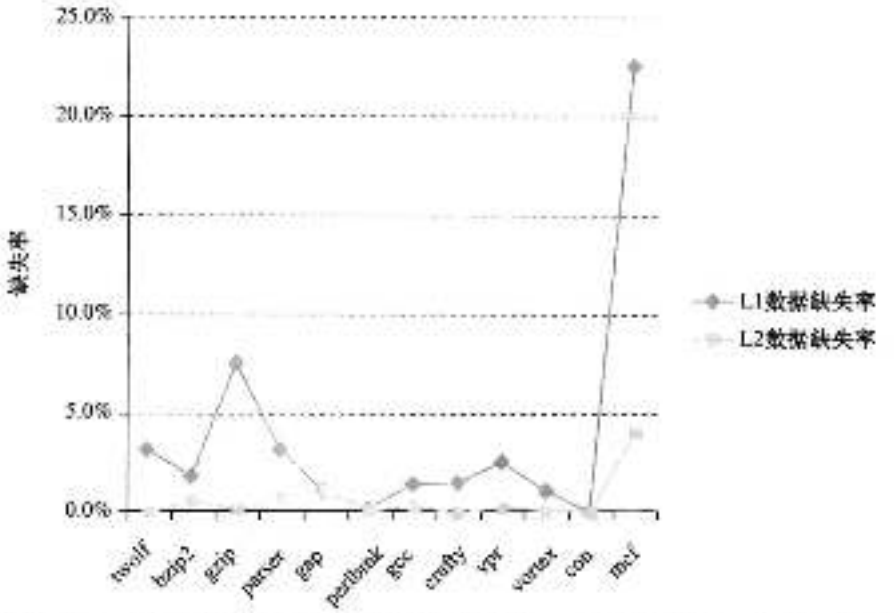



图 5-45 在 ARM Cortex-A8 上运行 SPEC2000 的简化版本 Minnespec 时的数据 cache 缺失率。对存储器需求大的应用的一级 cache 和二级 cache 的缺失率较高。需要注意的是二级 cache 缺失率是全局缺失率，也就是说，包括那些一级 cache 命中的所有访问（见 5.4 节的精解）。mcf 是 cache 不友好的程序。注意，该图与第 4 章图 4-76 使用相同的系统和基准测试程序

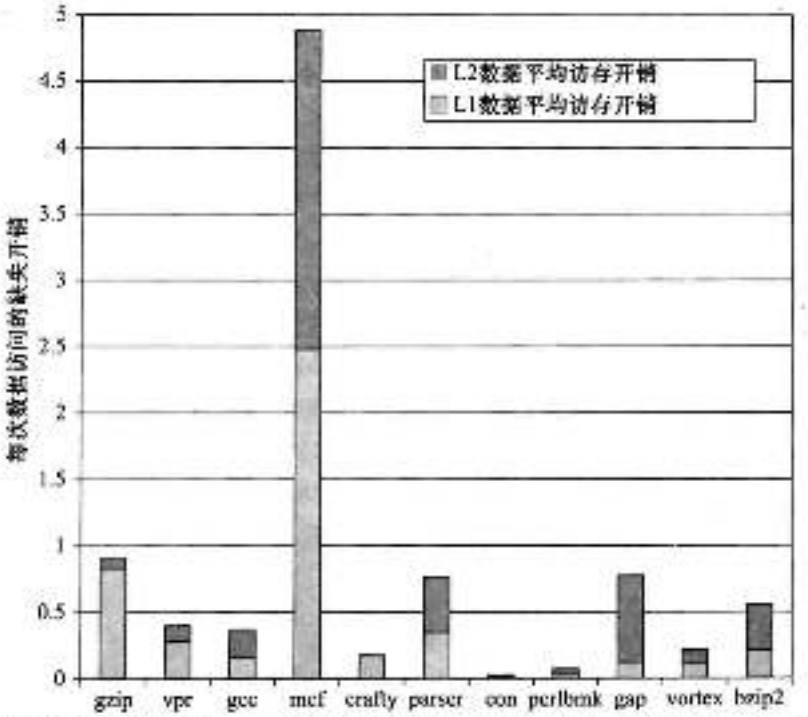



图 5-46 在 ARM 处理器上运行 Minnespec 时，每次数据存储器访问时一级和二级 cache 的平均访存开销（以时钟周期计）。虽然一级 cache 非常高，但是二级 cache 的缺失开销高了 5 倍以上，这意味着二级 cache 缺失对性能影响非常大

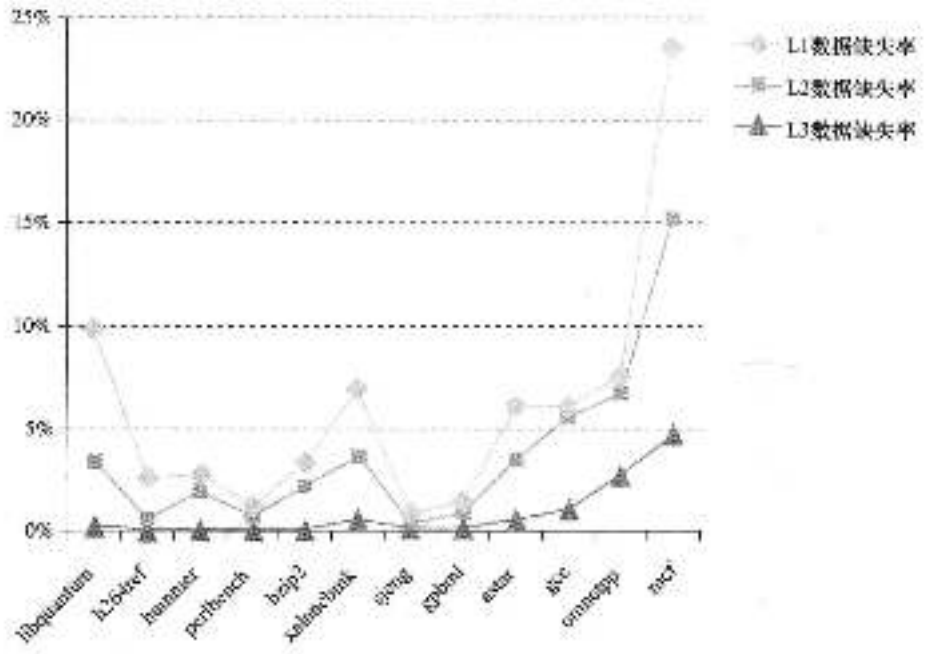



图5-47 在Intel Core i7 920上运行完整的SPEC2006整数程序是一级、二级和三级数据cache的缺失率

## 5.14 运行更快：cache分块和矩阵乘法

在第3章和第4章中已经通过子字并行和指令级并行来优化DGEMM的性能，可进一步通过在硬件上采用cache分块技术继续对其性能进行优化。图5-48给出了图4-80中DGEMM的分块版本。其变化与从图3-21中未做优化的DGEMM版本到图5-21的分块版本类似。此处使用第4章中循环展开后的DGEMM版本，并将其在A、B、C的子矩阵上调用多次。事实上，除了第7行中循环次数增量不同外，图5-48中第28～34行和第7～8行分别与图5-21中第14～20行和第5～6相同。

474
↓
475

与前面章节不同，本节没有给出对应的x86代码，这主要是因为分块技术对于计算没有影响，只是访问存储器中的数据顺序发生了变换，从而内循环的代码与图4-81中的代码几乎相同。不同点在于使用了用于实现循环的bookkeeping整数指令。图4-80中内循环之前有14条指令，内循环之后有8条指令，而图5-48中产生的bookkeeping代码中，内循环之前有40条指令，内循环之后有28条指令。但是，相比cache缺失率减少对性能的贡献，这些额外的指令开销就显得微不足道了。图5-49对比了未采用优化技术的性能和采用子字并行、指令级并行和cache优化技术的性能对比。对于大矩阵运算，分块技术使未采用展开的AVX代码性能提升了2～2.5倍。如果同时采用这二种优化技术，则性能提高8～15倍，且矩阵越大，性能提升越大。

**解惑** 如3.8节中解释所述，这些结果都是在将Turbo模式关闭时获得的。与第3章和第4章相同，如果将Turbo模式打开，则时钟频率将暂时提高到3.6/2.6=1.27倍。在这种情况下，因为只使用了八个核中的一个核，所以Turbo模式运行得很好。然而，如果要运行得更快，则需要使用所有内核，这将在第6章中讲述。

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3 #define BLOCKSIZE 32
4 void do_block (int n, int si, int sj, int sk,
5                 double *A, double *B, double *C)
6 {
7   for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
8     for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
9       __m256d c[4];
10      for ( int x = 0; x < UNROLL; x++ )
11        c[x] = _mm256_load_pd(C+i+x*4+j*n);
12      /* c[x] = C[i][j] */
13      for( int k = sk; k < sk+BLOCKSIZE; k++ )
14      {
15        __m256d b = _mm256_broadcast_sd(B+k+j*n);
16      /* b = B[k][j] */
17        for (int x = 0; x < UNROLL; x++)
18          c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                  _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20      }
21
22
23      for ( int x = 0; x < UNROLL; x++ )
24        _mm256_store_pd(C+i+x*4+j*n, c[x]);
          /* C[i][j] = c[x] */
25    }
26 }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30   for ( int sj = 0; sj < n; sj += BLOCKSIZE )
31     for ( int si = 0; si < n; si += BLOCKSIZE )
32       for ( int sk = 0; sk < n; sk += BLOCKSIZE )
33         do_block(n, si, sj, sk, A, B, C);
34 }
```

图 5-48 对图 4-80 中的 DGEMM 使用 cache 分块优化的 C 版本。与图 5-21 中的变化相同。编译器为 do_block 函数生成的汇编代码与图 4-81 中的代码几乎相同。需要再次强调的是，由于编译器才用内联函数调用，do_block 的调用没有开销

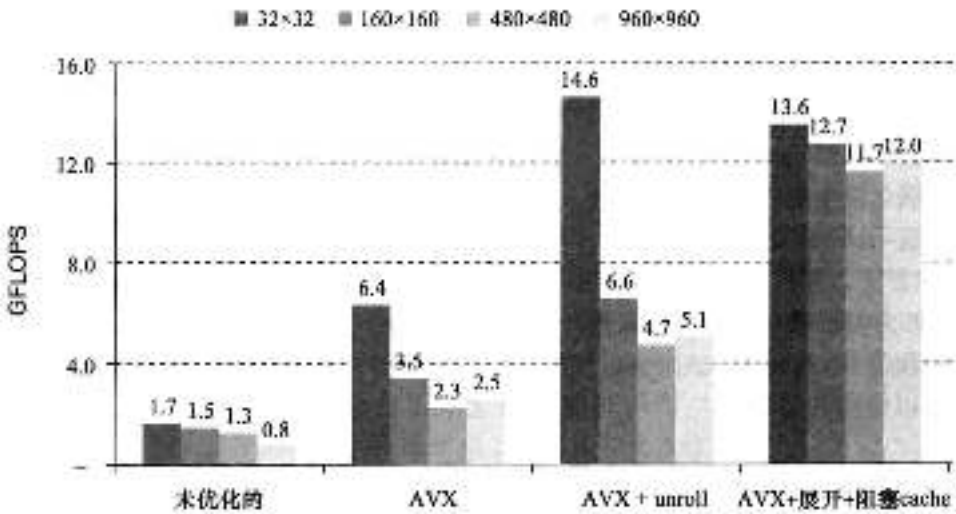



图 5-49 当矩阵规模从 32×32 增加到 960×960 时 4 种版本的 DGEMM 的性能。对于规模最大的矩阵，完全优化的代码的性能几乎是第 3 章图 3-21 中未优化代码的 15 倍

## 5.15 谬误和陷阱

作为计算体系结构中的定量原则，存储器层次结构似乎不易受到谬误和陷阱的影响。但实际上却大相径庭，很多人不仅已经有了很多的谬误，还遭遇了陷阱，而且其中的一些还导致了很多负面的结果。下面以学生在练习和考试中经常遇到的陷阱开始讲解。

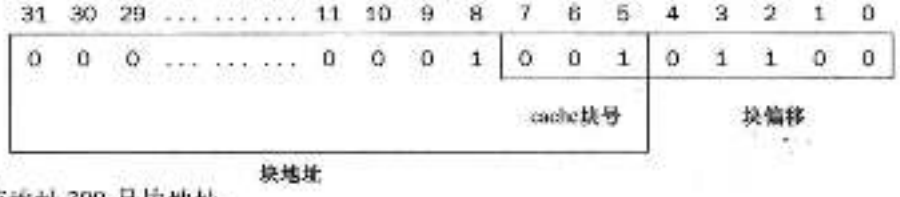
陷阱：在写程序或编译器生成代码时忽略存储系统的行为。

这可以很容易地写成一个谬误："在写代码时，程序员可以忽略存储器层次"。图5-19中的排序和5.14节的cache分块技术证明了如果程序员在设计算法时考虑存储系统的行为，则可很容易地将性能翻倍。

陷阱：在模拟cache的时候，忘记说明字节编址或者cache块大小。

当模拟cache的时候（手动或者通过计算机），我们必须保证，在确定一个给定的地址被映射到哪个cache块中时，一定要说明字节编址和多字块的影响。例如，如果我们有一个容量为32字节的直接映射的cache，块大小为4字节，则字节地址36映射到cache的块1，因为字节地址36是块地址9，而（9mod 8）=1。另一方面，如果地址36是字地址，那么它就映射到块（36mod 8）=4。要保证清楚地说明层次地址。

同样，我们必须说明块的大小。假设我们有一个256字节大小的cache，块大小为32字节。那么字节地址300将落入哪一块中？如果我们将地址300划分成域，就能看到答案：




字节地址300是块地址

$$\lfloor 300/32 \rfloor = 9$$

cache中的块数是

$$[256/32] = 8$$

478 块号9对应于cache块号（9 mod 8）=1。

许多人，包括作者（在早期的书稿中）和那些忘记自己预期的地址是字、字节或块号的教师们，都犯过这个错误。当你做练习时一定要注意这个易犯的错误。

陷阱：对于共享cache，组相联度少于核的数量或者共享该cache的线程数。

如果不特别注意，一个运行在$2^x$个处理器或者线程上的并行程序为数据结构分配的地址可能映射到共享二级cache同一个组中。如果cache至少是$2^x$路组相联，那么通过硬件可以隐藏这些程序偶尔发生的冲突。如果不是，程序员可能要面对明显不可思议的性能缺陷——事实上是由于二级cache冲突缺失引起的——在程序迁移时发生，假定从一个16核的机器迁移到一个32核的机器上，并且如果它们都使用16路组相联的二级cache。

陷阱：用存储器平均访问时间来评估乱序处理器的存储器层次结构。

如果处理器在cache缺失时阻塞，那么你可以分别计算存储器阻塞时间和处理器执行时间，因此可以使用存储器平均访问时间来独立地评估存储器层次结构（见5.4节第2个例子）。

如果处理器在cache缺失时继续执行指令，而且甚至可能维持更多的cache缺失，那么唯一可以用来准确评估存储器层次结构的办法是模拟乱序处理器和存储器结构。

陷阱：通过在大分段地址空间的顶部增加段来扩展地址空间。

在20世纪70年代，许多程序都变得很大，以至于不是所有的代码和数据都能仅用16位地址寻址。于是，计算机修改为32位地址，一种方法是直接使用未分段的32位地址空间（也称为平面地址空间），另一种方法是给已经存在的16位地址再增加16位长度的段。从市场观点来看，增加程序员可见的段，并且迫使程序员和编译器将程序划分成段，这样可以解决寻址问题。但遗憾的是，任何时候，一种程序设计语言要求的地址大于一个段的范围就会有麻烦，比如说大数组的索引、无限制的指针或者是引用参数。此外，增加段可以将每个地址变成两个字——一个是段号，另一个是段内偏移——这些在使用寄存器中地址时就会出现问题。

谬误：实际的磁盘故障率和规格书中声明的一致。

最近的两项研究评估了大量磁盘，目的是检查实际结果和规格之间的关系。其中一项研究了将近100 000个磁盘，他们标称其MTTF为1 000 000～1 500 000小时或者说具有0.6%～0.8%的AFR。他们发现2%～4%的AFR是常见的，通常比设定的故障率高3～5倍［Schroeder和Gibson，2007］。另一项研究了100 000个磁盘，这些磁盘标称具有1.5%的AFR，以及在第一年中，磁盘故障率为1.7%，到第三年，磁盘的故障率上升到8.6%，也就是说，大约是规格书中指定的故障率的6倍之多［Pinheiro、Weber和Barroso，2007］。 [479]

谬误：操作系统是调度磁盘访问的最好地方。

如5.2节所提到的，以高层接口为宿主操作系统提供逻辑块地址。假设在这样的高层抽象层OS可以通过将逻辑块的地址按照递增的顺序排序以得到最好的性能。然而，由于磁盘知道逻辑地址被映射到扇区、磁道上以及磁面上的实际的物理地址，这样通过调度就可以减少旋转以及寻道的时间。

例如，假设以下工作负载是4个读操作［Anderson，2003］：

| 操作 | LBA的起始地址 | 长度 |
|---|---|---|
| 读 | 724 | 8 |
| 读 | 100 | 16 |
| 读 | 9 987 | 1 |
| 读 | 26 | 128 |

宿主OS可能对4个读操作重新进行调度，编排成逻辑块的读操作的顺序：

| 操作 | LBA的起始地址 | 长度 |
|---|---|---|
| 读 | 26 | 128 |
| 读 | 100 | 16 |
| 读 | 724 | 8 |
| 读 | 9 987 | 1 |

依赖于数据在磁盘中的相对位置，如图5-50所示，重新编排I/O顺序可能会使情况变得更糟。磁盘调度的读操作在磁盘的3/4的旋转周期就全部完成，而操作系统调度的读操作花费了3个旋转周期。

陷阱：在不为虚拟化设计的指令集体系结构上实现虚拟机监视器。

在20世纪70年代和80年代，很多计算机体系结构设计者并没有刻意去保证所有读写相关的硬件资源指令都是特权指令。这种放任的态度导致了VMM在这些体系结构上存在问题，包括x86，这里我们就以它为例。

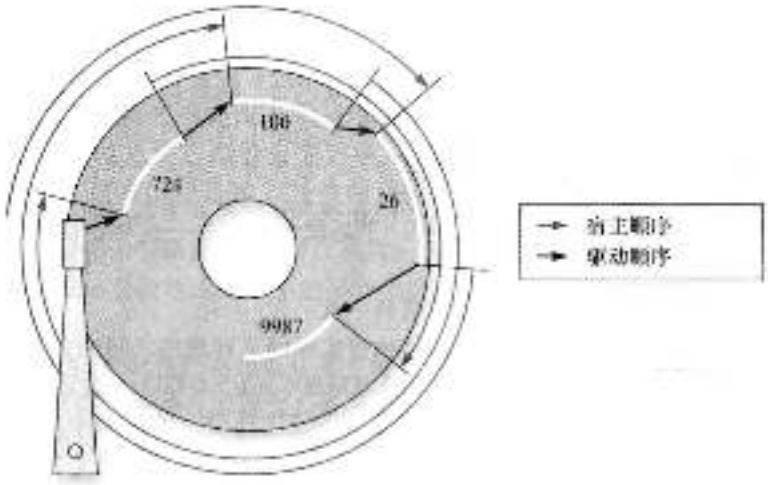



图5-50 OS算度与磁盘调度访问的例子，标记为寻址顺序和驱动顺序。前者完成4个读操作需要3个旋转周期，而后者完成4个读操作仅仅在一个3/4的旋转周期即可完成（资料来源：Anderson[2003]）

图5-51指出了虚拟化产生问题的18条指令［Robin和Irvine，2000］。其中两大类指令是：

- 在用户模式下读控制寄存器，暴露了在虚拟机上运行的guest操作系统（如前面提到的POPF）。
- 检查分段的体系结构所需的保护，但却假设操作系统在最高的特权级运行。

| 问题种类 | x86的问题指令 |
| --- | --- |
| 当运行在用户模式时，访问敏感寄存器无须trap中断 | 存储全局描述符表寄存器（SGDT）<br>存储局部描述符表寄存器（SLDT）<br>存储中断描述符表寄存器（SIDT）<br>存储机器状态字（SMSW）<br>标志入栈（PUSHF，PUSHFD）<br>标志出栈（POPF，POPFD） |
| 在用户模式下访问虚拟存储机器时，x86保护检查指令失效 | 从段描述符读取访问权限（LAR）<br>从段描述符读取段的边界（LSL）<br>如果段描述符可读，进行读校验（VERR）<br>如果段描述符可写，进行写校验（VERW）<br>段寄存器出栈（POP CS，POP SS，…）<br>段寄存器入栈（PUSH CS，PUSH SS，…）<br>远调用不同的特权级（CALL）<br>远返回至不同的特权级（RET）<br>远跳转至不同的特权级（JMP）<br>软中断（INT）<br>存储段选择寄存器（STR）<br>移入/移出段寄存器（MOVE） |

图5-51 虚拟化产生问题的18条x86指令的概述［Robin和Irvine，2000］。上面一组的前5条指令允许程序在用户模式下读控制寄存器，而无须trap中断，例如描述符表寄存器。标记出栈指令会修改包含敏感信息的控制寄存器，但在用户模式下将失效而无任何提示。x86体系结构中段的保护检查在下面的一组指令中，当读取控制寄存器时，作为指令执行的一部分，都会隐式地检查特权级。进行检查时操作系统必须运行在最高特权级，但是对客户虚拟机并没有这样的要求。只有在移入段寄存器操作时会试图修改控制状态，但是，保护检查同样会阻止它这么做

为了简化在 x86 上实现 VMM，AMD 和 Intel 都提出过新的模式扩展体系结构。Intel 的 VT-x 为虚拟机运行提供了一个新的执行模式，一个面向虚拟机状态的体系结构定义、快速虚拟机切换指令，以及一大组用来选择调入 VMM 环境的参数。总之，VT-x 在 x86 中加了 11 条新指令。AMD 的 Pacifica 做了相似的改进。

另一种方法通过修改硬件来对操作系统做细微的修改以简化虚拟化。这种技术称为*类虚拟化*（paravirtualization），例如开源的虚拟机监视器 Xen 就是一个很好的例子。Xen 虚拟机监视器提供给客户操作系统一个抽象虚拟机，它仅仅使用了供虚拟机监视器运行的 x86 物理硬件中易于虚拟化的一部分。

## 5.16 本章小结

无论在最快的计算机还是最慢的计算机中，构成主存的原材料——DRAM 本质是相同的，并且是最便宜的，这使得构建一个和快速处理器保持同步的存储系统变得更加困难。

局部性原理可以用来克服存储器访问的长延迟——这个策略的正确性已经在存储器层次结构的各级都得到了证明。尽管层次结构中的各级从量的角度来看非常不同，但是在它们的执行过程中都遵循相似的策略，并且利用相同的局部性原理。

多级 cache 可以更方便地使用更多的优化。这有两个原因。第一，较低级 cache 的设计参数与一级 cache 不同。例如，由于较低级 cache 的容量一般很大，因此可能使用更大容量的块。第二，较低级 cache 并不像一级 cache 那样经常被处理器用到。这让我们考虑当较低级 cache 空闲时让它做一些事情以预防将来的缺失。

另一个趋势是寻求软件的帮助。使用大量的程序转换和硬件设备有效地管理存储器层次结构是增强编译器作用的主要焦点。现在有两种不同的观点。一种是重新组织程序结构以增强它的空间和时间局部性。这种方法主要针对以大数组为主要数据结构的面向循环的程序；大规模的线性代数问题就是一个典型的例子，例如 DGEMM。通过重新组织访问数组的循环增强了局部性——也因此改进了 cache 性能。

还有一种方法是预取（prefetching）。在预取机制中，一个数据块在真正被访问之前就被取入 cache 中了。许多微处理器使用硬件预取尝试预测访问，这对软件可能比较困难。

⊃ 预取：使用特殊指令将未来可能用到的指定地址的 cache 块提前搬到 cache 中的一种技术。

第三种方法是使用优化存储器传输的特殊 cache 感知（cache-aware）指令。例如，在第 6 章的 6.10 节中，微处理器使用了一个优化设计：当发生写缺失时，由于程序要写整个块，因而并不从主存中取回一个块。对于一个内核来说，这种优化明显减少了存储器的传输。

我们将在第 6 章中看到，对并行处理器来说，存储系统也是一个重要的设计问题。存储器层次结构决定系统性能的重要性在不断增长，这也意味着在未来的几年内，这一领域对设计者和研究者来说将成为焦点。

## 5.17 历史观点和拓展阅读

本节网站中的内容描述了存储器技术的概况，从汞延迟线到 DRAM，存储器层次结构的发明，保护机制以及虚拟机，最后以操作系统的一个简单发展历史作总结，包括 CTSS、MULTICS、UNIX、BSD UNIX、MS-DOS、Windows 和 Linux。

## 5.18 练习题

**5.1** 在这个练习中，考虑矩阵计算中存储器的局部特性。下面的代码由 C 语言编写，在同一行中的元素被连续存放。假定每个字是 32 位整数。

```
for (i=0; i<8; i++)
  for (j=0; j<8000; j++)
    A[i][j] = B[i][0]+A[j][i];
```

5.1.1 [5] <5.1> 16字节的cache块中可以存放多少32位的整数?

5.1.2 [5] <5.1> 访问哪些变量会显示出时间局部性?

5.1.3 [5] <5.1> 访问哪些变量会显示出空间局部性?

局部性同时受访问顺序和数据存放位置的影响。在同一列的矩阵元素是连续存放的情况下，同样的计算机也可以用下面的不同于C的Matlab来写。

```
for i=1:8
  for j=1:8000
    A(i,j)=B(i,0)+A(j,i);
  end
end
```

483

5.1.4 [10] <5.1> 存放全部将被访问的32位矩阵元素需要多少16字节的cache块?

5.1.5 [5] <5.1> 访问哪些变量会显示出时间局部性?

5.1.6 [5] <5.1> 访问哪些变量会显示出空间局部性?

5.2 cache为处理器提供了一个高性能的存储器层次结构，因此十分重要。下面是一个32位存储器地址引用的列表，给出的是字地址。

3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253

5.2.1 [10] <5.3> 已知一个直接映射的cache，有16个块，块大小为1个字。对于每次访问，请标识出二进制地址，标记以及索引。假设cache最开始为空，那么请列出每次访问是命中还是缺失。

5.2.2 [10] <5.3> 已知一个直接映射的cache，有8个块，块大小为2个字。对于每次访问，请标识出二进制地址，标记以及索引。假设cache最开始为空，那么请列出每次访问是命中还是缺失。

5.2.3 [20] <5.3, 5.4> 对已知的访问来优化cache的设计。这里有三种直接映射的cache设计方案，每个容量都为8个字：C1块大小为1个字，C2块大小为2个字，C3块大小为4个字。根据缺失率，哪种cache设计最好?如果缺失阻塞时间为25个周期，C1的访问时间为2个周期，C2为3个周期，C3为5个周期，那么哪种cache设计最好?

这里有许多对cache整体性能很重要的不同的设计参数。下面列出了对于不同的直接映射cache设计的参数：

cache数据量：32KiB

cache块大小：2个字

cache访问时间：1个周期

5.2.4 [15] <5.3> 假定32位的地址，计算上面列出的cache所需的总位数。给定总的大小，找出最接近的直接映像cache的总的大小，该cache块的大小为16个字长或更大。请解释为什么第二种cache比第一种cache的访问速度更慢，尽管第二种cache的数据量更大。

5.2.5 [20] <5.3, 5.4> 在一个2KiB的两路组相联cache上产生一系列读请求时的缺失率要比在表中cache上执行该请求的缺失率低。请给出一个可能的解决方案，使得表中列出的cache的缺失率等于或者低于2KiB cache的缺失率。讨论这种解决方案的优点和缺点。

5.2.6 [15] <5.3>5.3节的公式说明了用来索引直接映射cache的典型方法：(块地址) mod (cache中的块数)。假设地址为32位，cache中有1024个块，考虑一个不同的索引函数：(块地址[31:27]^XOR 块地址[26:22])。可以使用这个公式来索引直接映射的cache吗?如果可以，请解释原因，并且讨论可能需要对cache做的一些改动。如果不可以，请解释原因。

484

5.3 对于一个32位地址的直接映射的cache设计，下面的地址位用来访问cache。

| 标记 | 索引 | 偏移量 |
|---|---|---|
| 31~10 | 9~5 | 4~0 |

5.3.1 [5] <5.3> cache 块大小是多少（单位为字）？

5.3.2 [5] <5.3> cache 有多少项？

5.3.3 [5] <5.3> 这样的 cache 执行时所需的总位数与数据存储位数之间的比率是多少？

下表记录了从上电开始的 cache 访问的字节地址。

| 地址 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 16 | 132 | 232 | 160 | 1 024 | 30 | 140 | 3 100 | 180 | 2 180 |

5.3.4 [10] <5.3> 有多少块被替换？

5.3.5 [10] <5.3> 命中率是多少？

5.3.6 [20] <5.3> 列出 cache 的最终状态，每个有效项以记录的形式 < 索引，标记，数据 > 表示出来。

5.4 回忆一下两个写策略和写分配策略，它们结合起来既可以在一级 cache 中执行，也可以在二级 cache 中执行。假定一级和二级 cache 如下：

| 一级 cache | 二级 cache |
|---|---|
| 写直达，写不分配 | 写回，写分配 |

5.4.1 [5] <5.3，5.8> 在存储器层次结构中的不同层使用缓冲器来降低访问延迟。对这个给定的配置，列出一级 cache 与二级 cache 之间，以及二级 cache 与存储器之间可能需要的缓冲器。

5.4.2 [20] <5.3，5.8> 描述处理一级 cache 写缺失的过程，考虑里面的组件以及替换一个脏块的可能性。

5.4.3 [20] <5.3，5.8> 对于一个多级独占 cache（一个块只能存放在一个 cache 中，或者在一级 cache 中，或者在二级 cache 中）配置，描述处理一级 cache 写缺失的过程，考虑到里面的组件以及替换一个脏块的可能性。

考虑下面的方案和 cache 行为。

485

| 每 1000 条指令中数据读的次数 | 每 1000 条指令中数据写的次数 | 指令 cache 缺失率 | 数据 cache 缺失率 | 块大小（字节） |
|---|---|---|---|---|
| 250 | 100 | 0.30% | 2% | 64 |

5.4.4 [5] <5.3，5.8> 对于一个使用写直达法、写分配策略的 cache，如果 CPI 为 2，所需最小的读/写带宽是多少（以每周期字节数来度量）？

5.4.5 [5] <5.3，5.8> 对于一个使用写回法、写分配策略的 cache，假定 30% 被替换的数据块为脏块，那么如果 CPI 为 2，所需最小的读/写带宽是多少？

5.4.6 [5] <5.3，5.8> 如果要实现 CPI = 1.5 的性能，所需的最小带宽是多少？

5.5 播放音频或视频文件的多媒体应用是一类被称为“流”的负载的一部分；即它们取回大量的数据，但是大部分数据都不会再使用。考虑一个视频流负载依次访问一个 512KiB 的工作集的情况，地址流如下：

0，2，4，6，8，10，12，14，16…

5.5.1 [5] <5.4，5.8> 假设有一个 64KiB 的直接映射 cache，cache 块大小为 32 字节。那么对于上面的地址流，缺失率是多少？当 cache 大小或者工作集变化时，cache 的缺失率如何随之变化？根据 3C 模型，这些缺失如何被分类？

5.5.2 [5] <5.1，5.8> 当 cache 块大小分别为 16 字节、64 字节和 128 字节时，重新计算缺失率。该负载所采用的是哪种局部性？

5.5.3 [10] <5.13> “预取”是一种技术：当一个特殊 cache 块被访问时，预测地址模式并推测地取回其他 cache 块。预取的一个例子是流缓冲区，当一个特定的 cache 块被取回时，将与其相邻的 cache 块也依次预取回到一个独立的缓冲区中。如果所需的数据在预取缓冲区中，那么看成是一

次命中并且将数据移入 cache，同时预取下一个 cache 块。假设一个流缓冲区有两项，并且假设 cache 延迟满足：在先前 cache 块的计算完成之前可以加载下一个 cache 块。那么对于上面的地址流，缺失率是多少？

cache 块大小（B）影响了缺失率和缺失延迟。假设有下面的缺失率表，并假定 CPI 为 1 的机器中，每条指令平均访问次数（指令和数据）为 1.35，给定不同容量的 cache 的缺失率，试找出最优的 cache 块大小。

| 8:4% | 16:3% | 32:2% | 64:1.5% | 128:1% |
|---|---|---|---|---|

5.5.4 [10] <5.3> 缺失延迟为 20 × B 个周期时，最佳的块大小是多少？

5.5.5 [10] <5.3> 缺失延迟为 24 + B 个周期时，最佳的块大小是多少？

486 5.5.6 [10] <5.3> 缺失延迟为恒定值时，最佳的块大小是多少？

5.6 在这个练习中，我们将研究不同容量对整体性能的影响。通常来说，cache 访问时间与 cache 容量成正比。假设访问主存需要 70ns，并且在所有指令中，有 36% 的指令需要访存。下表是 P1 和 P2 两个处理器各自的一级 cache 的数据。

| | 一级 cache 容量 | 一级 cache 缺失率 | 一级 cache 命中时间 |
|---|---|---|---|
| P1 | 2KiB | 8.0% | 0.66ns |
| P2 | 4KiB | 6.0% | 0.90ns |

5.6.1 [5] <5.4> 假定一级 cache 命中时间决定了 P1 和 P2 的周期时间，它们各自的时钟频率是多少？

5.6.2 [5] <5.4> P1 和 P2 各自的 AMAT（平均存储器访问时间）分别是多少？

5.6.3 [5] <5.4> 假定在没有任何存储器阻塞时基本的 CPI 为 1.0，P1 和 P2 各自的总 CPI 分别是多少？哪个处理器更快？

对下面的三个问题，我们考虑在 P1 中增加二级 cache，以弥补一级 cache 容量的限制。在解决这些问题时，依然使用上表中一级 cache 的容量和命中时间。二级 cache 缺失率是它的局部缺失率。

| 二级 cache 容量 | 二级 cache 缺失率 | 二级 cache 命中时间 |
|---|---|---|
| 1MiB | 95% | 5.62ns |

5.6.4 [10] <5.4> 增加二级 cache 后，P1 的 AMAT 是多少？有了二级 cache，AMAT 是更好还是更差了？

5.6.5 [5] <5.4> 假定在没有任何存储器阻塞时基本的 CPI 为 1.0，增加二级 cache 后，P1 的总的 CPI 是多少？

5.6.6 [10] <5.4> P1 有了二级 cache 后，哪个处理器更快？如果 P1 更快，那么 P2 中一级 cache 的缺失率需要为多少才能匹配 P1 的性能？如果 P2 更快，那么 P1 中一级 cache 的缺失率需要为多少才能匹配 P2 的性能？

5.7 这个练习研究了不同 cache 设计的效果，特别将关联的 cache 与 5.4 节中的直接映射的 cache 进行比较。练习中使用的是练习题 5.2 中的地址流。

5.7.1 [10] <5.4> 使用练习题 5.2 中的访问信息，对于一个 3 路组相联、块大小为 2 个字、总容量为 24 个字、使用 LRU 替换算法的 cache，指出 cache 中最终的内容。对每个访问，标识出索引位、标记位，块换移位，以及当前访问是命中还是缺失。

5.7.2 [10] <5.4> 使用练习题 5.2 中的访问信息，对于一个全相联、块大小为 1 个字、总容量为 8 个字、使用 LRU 替换算法的 cache，指出 cache 中最终的内容。对每个访问，标识出索引位、标记位，以及当前访问是命中还是缺失。

487

5.7.3 [15] <5.4> 使用练习题 5.2 中的访问信息，对于一个全相联、块大小为 2 个字、总容量为 8 个字、使用 LRU 替换算法的 cache，需何缺失率是多少？如果替换为 MRU（最近最常使用）算法，

缺失率是多少？在这些替换策略下，最好的情况下，cache 缺失率是多少？

多级 cache 是一项重要技术，它在克服一级 cache 提供的有限空间的同时仍然保持了速度。假设一个处理器的参数如下：

| 没有存储器阻塞的基本 CPI | 处理器速度 | 主存访问时间 | 每条指令的一级 cache 缺失率 | 直接映射的二级 cache 的速度 | 包含直接映射的二级 cache 时的全局缺失率 | 8 路组相联的二级 cache 的速度 | 包含 8 路组相联的二级 cache 时的全局缺失率 |
|---|---|---|---|---|---|---|---|
| 1.5 | 2GHz | 100ns | 7% | 12 个周期 | 3.5% | 28 个周期 | 1.5% |

**5.7.4** [10] <5.4> 计算表中处理器的 CPI：1）只有一级 cache；2）一个直接映射的二级 cache；3）一个 8 路组相联的二级 cache。如果主存访问时间加倍，CPI 如何变化？如果主存访问时间减半，CPI 又如何变化？

**5.7.5** [10] <5.4> 拥有比两级 cache 更多的 cache 层次是可能的。已知上述的处理器拥有一个直接映射的二级 cache，一个设计者希望增加一个三级 cache，其访问时间为 50 个周期，并且全局缺失率降低到 1.3%。这种设计能提供更好的性能吗？通常来说，增加一个三级 cache 的优点和缺点分别是什么？

**5.7.6** [20] <5.4> 在以前的处理器中，如 Intel Pentium 或 Alpha 21264，二级 cache 在远离主处理器和一级 cache 的片外（放置在不同的芯片上）。这使得二级 cache 很大，访问延迟也高得多，同时由于二级 cache 以较低的频率运行，带宽通常也较低。假设一个 512KiB 的片外二级 cache 的全局缺失率为 4%。如果 cache 每增加 512KiB 容量可以降低 0.7% 的全局缺失率，并且 cache 总的访问时间为 50 个周期，那么 cache 容量为多大时才能匹配表中直接映射的二级 cache 的性能？如果匹配表中 8 路组相联的 cache 性能，cache 容量又需要是多少？

**5.8** 失效时间间隔（mean time between failure，MTBF）、替换时间（mean time to replacement，MTTR）和无故障工作时间（mean time to failure，MTTF）对于评估一个存储资源的可靠性和可用性非常有用。通过使用如下参数回答下列问题：

| MTTF | MTTR |
|---|---|
| 3 年 | 1 天 |

[488]

**5.8.1** [5] <5.5> 计算表中每台设备的 MTBF。

**5.8.2** [5] <5.5> 计算表中每台设备的可用性。

**5.8.3** [5] <5.5> 如果 MTTR 接近于 0，则可用性如何变化？这是一个合理的情形吗？

**5.8.4** [5] <5.5> 如果 MTTR 非常高，例如一台设备非常难维修，则可用性如何变化？这是否意味着该设备可用性很低？

**5.9** 本练习题检查纠正一位错检测两位错（SEC/DED）的汉明码。

**5.9.1** [5] <5.5> 如果要对 128 位字采用 SEC/DED 编码进行保护，最少需要多少位的奇偶位？

**5.9.2** [5] <5.5>5.5 节指出，现代服务器存储器模块（DIMM）对于 64 位数据，使用 8 位奇偶位来实现 SEC/DEC ECC。计算该编码的开销/性能比，并与 5.9.1 进行比较。在这里开销是指所需的相对的奇偶位，性能是指相对的能够纠正的错误数量。哪种编码比较好？

**5.9.3** [5] <5.5> 考虑一个采用 4 位奇偶位来保护 8 位字的 SEC。如果读出值为 0x375，是否有错？如果有错，对错误进行纠正。

**5.10** 对于一个高性能系统如 B-tree 索引数据库，页的大小主要由数据量和磁盘性能决定。假设一个 B-tree索引页（项数固定）使用了 70%。使用的页就是 B-tree 的深度，用 $\log_2$（项数）来计算。下表显示的是 10 年前的一个拥有 16 字节表项的磁盘，延迟为 10ms，传输率为 10MB/s，最优的页大小是 16K。

| 页大小（KiB） | 页的使用/Bi-true 次数（保存的磁盘访问次数） | 调用页的访问开销（ms） | 效用/代价 |
|---|---|---|---|
| 2 | 6.49 或 $\log_2(2\,048/16\times0.7)$ | 10.2 | 0.64 |
| 4 | 7.49 | 10.4 | 0.72 |
| 8 | 8.49 | 10.8 | 0.79 |
| 16 | 9.49 | 11.6 | 0.82 |
| 32 | 10.49 | 13.2 | 0.79 |
| 64 | 11.49 | 16.4 | 0.70 |
| 128 | 12.49 | 22.8 | 0.55 |
| 256 | 13.49 | 35.6 | 0.38 |

**5.10.1** [10] <5.7> 如果项数为 128 字节，最佳的页大小是多少？

**5.10.2** [10] <5.7> 根据练习题 5.10.1，如果页处于半满状态，最佳的页大小是多少？

**5.10.3** [20] <5.7> 根据练习题 5.10.2，如果使用的是最新的磁盘，延时 3ms，而传输率为 100MB/s 时，最佳的页大小是多少？请解释为什么未来的服务器可能用较大的页？

489

在 DRAM 保存“频繁使用”的页（“热门”的页）可以避免磁盘访问。但是对于一个系统，我们如何判断“频繁使用”的精确含义？数据工程师利用 DRAM 和磁盘访问之间的开销比率对频繁使用页的重用时间阈值进行量化。磁盘访问的开销是 $Disk/accesses_per_sec，将页保存在 DRAM 中的开销是 $DRAM_MiB/page_size。在某些年代中，典型的 DRAM 和磁盘开销、典型的数据库页大小如下表所示：

| 年代 | DRAM 开销（$/MiB） | 页大小（KiB） | 磁盘开销（$/disk） | 磁盘访问率（访问/秒） |
|---|---|---|---|---|
| 1987 | 5 000 | 1 | 15 000 | 15 |
| 1997 | 15 | 8 | 2 000 | 64 |
| 2007 | 0.05 | 64 | 80 | 83 |

**5.10.4** [10] <5.1，5.7> 对于这三种技术时代，重用时间阈值是多少？

**5.10.5** [10] <5.7> 如果我们保持使用相同的 4K 大小的页，重用时间阈值是多少？这里趋势是什么？

**5.10.6** [20] <5.7> 为了保持使用相同的页大小（因此避免软件重写），其他方面应该如何变化？讨论与当前技术和成本趋势的相似性。

**5.11** 如 5.7 节所述，虚拟存储器使用一个页表来追踪虚拟地址到物理地址之间的映射。这个练习说明了当地址被访问时，页表如何更新。下表是在一个系统上所看见的虚拟地址流。假设 4KiB 的页，一个 4 项的全相联 TLB，使用严格的 LRU 替换算法。如果必须从磁盘中取回页，那么增加下一次能取的最大页数：

4669, 2227, 13916, 34587, 48870, 12608, 49225

TLB

| 有效位 | 标记位 | 物理页号 |
|---|---|---|
| 1 | 11 | 12 |
| 1 | 7 | 4 |
| 1 | 3 | 6 |
| 0 | 4 | 9 |

490

页表

| 有效位 | 物理页/在磁盘上 | 有效位 | 物理页/在磁盘上 |
|---|---|---|---|
| 1 | 5 | 0 | 磁盘 |
| 0 | 磁盘 | 1 | 4 |
| 0 | 磁盘 | 0 | 磁盘 |
| 1 | 6 | 0 | 磁盘 |
| 1 | 9 | 1 | 3 |
| 1 | 11 | 1 | 12 |

**5.11.1** [10] <5.7> 根据给出的地址流，以及 TLB、页表的初始状态，请给出系统的最终状态。对于每次访问，请列出是否在 TLB 中命中，是否在页表中命中或是发生缺页。

**5.11.2** [15] <5.7> 重复练习题 5.11.1，但是这次使用 16KiB 的页来代替 4KiB 的页。使用更大的页的好处有哪些？缺点又有哪些？

**5.11.3** [15] <5.4, 5.7> 如果使用两路组相联的 TLB，请指出 TLB 中最终的内容。如果 TLB 是直接映射的，同样指出 TLB 中最终的内容。讨论使用 TLB 来获得高性能的重要性。如果没有 TLB，那么如何处理虚拟存储器访问？

有一些参数对页表整个大小会有影响。下面就列出一些关键的页表参数。

| 虚拟地址位数 | 页的大小 | 页表项的大小 |
|---|---|---|
| 32 位 | 8KiB | 4 字节 |

**5.11.4** [5] <5.7> 已知上表中的参数，一个系统用了一半的内存来运行 5 个应用程序，计算该系统使用的页表总大小。

**5.11.5** [10] <5.7> 已知上表中的参数，一个系统用了一半的内存来运行 5 个应用程序，给定一个两级的有 256 项的页表，计算该系统使用的页表总大小。假设主页表中每项是 6 字节，计算所需的最小和最大的内存容量。

**5.11.6** [10] <5.7> 一名 cache 设计者要将一个 4KiB 的虚拟索引、物理标记的 cache 的容量增大，已知页的大小在上表中列出，那么能否构建一个 16KiB 的直接映射 cache，假设块大小为 2 个字？设计者如何增加 cache 的数据大小？

491

**5.12** 在这个练习中，我们将研究对页表进行空间/时间的优化。下表是一个虚拟存储器系统的参数。

| 虚拟地址［位］ | 物理 DRAM | 页大小 | PTE 大小［字节］ |
|---|---|---|---|
| 43 | 16GiB | 4KiB | 4 |

**5.12.1** [10] <5.7> 对于一个单级页表，需要多少页表项（PTE）？存放页表需要多少物理存储器？

**5.12.2** [10] <5.7> 使用多级页表可以降低物理存储器中页表的消耗，它在物理存储器中只保存活跃的 PTE。这种情况下，需要多少级的页表？如果 TLB 缺失，地址转换需要访问多少次存储器？

**5.12.3** [15] <5.7> 反置页表可以用来进一步优化空间和时间。存放页表需要多少 PTE？假设执行一个哈希表，当 TLB 缺失时，在正常情况下和最差情况下的存储器访问次数分别是多少？

下表是一个有 4 项内容的 TLB：

| 入口 ID | 有效位 | 虚拟地址页 | 修改位 | 保护位 | 物理地址页 |
|---|---|---|---|---|---|
| 1 | 1 | 140 | 1 | RW | 30 |
| 2 | 0 | 40 | 0 | RX | 34 |
| 3 | 1 | 200 | 1 | RO | 32 |
| 4 | 1 | 280 | 0 | RW | 31 |

**5.12.4** [5] <5.7> 在什么样的情况下第二项的有效位被置为0?

**5.12.5** [5] <5.7> 当一条指令写入虚拟地址页号为30处时，会发生什么情况？什么时候软件管理的TLB比硬件管理的TLB速度快?

**5.12.6** [5] <5.7> 当一条指令写入虚拟地址页200时，会发生什么情况?

**5.13** 在这个练习中，我们将研究替换策略是如何影响缺失率的。假设一个有4个块的2路组相联cache。你会发现页表对于解决练习题中的问题没有帮助，如下面的示范，地址序列为“0，1，2，3，4”：

| 被访问的主存块的地址 | 命中/缺失 | 逐出的块 | cache块的内容 | | | |
|---|---|---|---|---|---|---|
| | | | 组0 | 组0 | 组1 | 组1 |
| 0 | 缺失 | | 主存[0] | | | |
| 1 | 缺失 | | 主存[0] | | 主存[1] | |
| 2 | 缺失 | | 主存[0] | 主存[2] | 主存[1] | |
| 3 | 缺失 | | 主存[0] | 主存[2] | 主存[1] | 主存[3] |
| 4 | 缺失 | 0 | 主存[4] | 主存[2] | 主存[1] | 主存[3] |
| … | | | | | | |

492

考虑如下的地址序列：0，2，4，8，10，12，14，16，0。

**5.13.1** [5] <5.4，5.8> 假定使用LRU替换策略，在这组地址序列中有多少次命中?

**5.13.2** [5] <5.4，5.8> 假定使用MRU（最近最常使用）替换策略，在这组地址序列中有多少次命中?

**5.13.3** [5] <5.4，5.8> 通过掷硬币来模拟随机替换策略。例如，“正面”表示逐出组中第一块，“反面”表示逐出组中第二块。在这组地址序列中有多少次命中?

**5.13.4** [10] <5.4，5.8> 为了最大化命中次数，每次替换时哪个块应该被逐出？如果使用了这个“最优的”策略，在这组地址序列中有多少次命中?

**5.13.5** [10] <5.4，5.8> 请说明为什么实现这种对所有地址序列来说都是最优的cache替换策略很难?

**5.13.6** [10] <5.4，5.8> 假设在每次主存引用时，可以决定被请求的地址是否要被缓存，这对缺失率有什么影响?

**5.14** 为了支持多虚拟机，需要对两级存储器进行虚拟化。每个虚拟机依然控制从虚拟地址（VA）到物理地址（PA）之间的映射，同时管理程序将每个虚拟机的物理地址（PA）映射到实际的机器地址（MA）。为了加速映射过程，一种被称为“影子分页”（shadow paging）的软件方法在管理程序中复制了每个虚拟机的页表，并且侦听从虚拟地址到物理地址的映射变化，以保证两个副本的一致性。为了消除影子页表（shadow page table）的复杂性，一种被称为嵌套页表（nested page table，NPT）的硬件方法可以支持两种页表（VA⇒PA 和 PA⇒MA），并且完全依靠硬件来查找这些表。

考虑下面的操作序列：（1）创建进程；（2）TLB缺失；（3）缺页；（4）上下文切换。

**5.14.1** [10] <5.6，5.7> 对于给定的操作序列，对影子页表和嵌套页表分别会发生什么情况?

**5.14.2** [10] <5.6，5.7> 假设一个基于x86架构的4级页表同时存放在客户页表（guest page table）和嵌套页表中，那么在处理本地页表（native page table）TLB缺失和嵌套页表TLB缺失时，分别需要多少次存储器访问?

**5.14.3** [15] <5.6，5.7> 在TLB缺失率、TLB缺失延迟、缺页率、缺页处理延迟之间，对影子页表来说，哪些度量标准更重要？而对于嵌套页表来说，哪些度量标准更重要?

493

下表是影子分页系统的参数。

| 每1 000条指令TLB缺失数 | 嵌套页表TLB缺失延迟 | 每1 000条指令缺页数 | 影子页缺失代价 |
|---|---|---|---|
| 0.2 | 200个时钟周期 | 0.001 | 30 000个时钟周期 |

**5.14.4** [10] <5.6> 一个基准测试程序的本地执行 CPI 为 1。如果使用影子页表，CPI 是多少？如果使用嵌套页表（假设只有页表虚拟化开销），CPI 是多少？

**5.14.5** [10] <5.6> 使用什么技术可以减少影子页表所带来的开销？

**5.14.6** [10] <5.6> 使用什么技术可以减少嵌套页表所带来的开销？

**5.15** 广泛使用虚拟机最大的障碍之一是运行虚拟机所导致的执行开销。下表列出了不同的性能参数和应用程序行为。

| 基本的 CPI | 每 10 000 条指令中特权 O/S 访问次数 | 对客户 O/S 执行 trap 中断的性能影响 | 对 VMM 执行 trap 中断的性能影响 | 每 10 000 条指令中 I/O 访问次数 | I/O 访问时间（包括 trap 中断客户 O/S 的时间） |
|---|---|---|---|---|---|
| 1.5 | 120 | 15 个时钟周期 | 175 个时钟周期 | 30 | 1 100 个时钟周期 |

**5.15.1** [10] <5.6> 对上面列出的系统计算 CPI，假设没有 I/O 访问。如果 VMM 性能影响加倍，CPI 是多少？如果减半呢？如果一个虚拟机软件公司希望获得 10% 的性能损失，那么对 VMM 执行 trap 中断的最长的时间代价是多少？

**5.15.2** [10] <5.6> I/O 访问对系统整体性能有着很大的影响。使用上面性能特征值的机器的 CPI，假设是非虚拟化的系统。如果使用虚拟化的系统，CPI 又是多少？如果系统中 I/O 访问减半，那么这些 CPI 如何变化？请解释为什么 I/O 限制性应用受虚拟化影响很小。

**5.15.3** [30] <5.6, 5.7> 比较并对比虚拟存储器和虚拟机的概念。各自的目标是什么？各自的利弊是什么？列出一些需要虚拟存储器的情况，以及一些需要虚拟机的情况。

**5.15.4** [20] <5.6> 5.6 节讨论了虚拟化，其中假设虚拟化的系统和底层硬件运行相同的 ISA。然而，虚拟化的一种可能的用途是对非本地的 ISA 进行仿真。QEMU 就是这样一个例子，可以用来仿真多种 ISA，如 MIPS、SPARC 以及 PowerPC。与这种虚拟化相关的一些难点是什么？被模拟的系统可能比在本地 ISA 上运行得更快吗？

494

**5.16** 在这个练习中，我们将研究处理器 cache 控制器中带写缓冲器的控制单元。可以使用图 5-40 的有限状态机作为设计有限状态机的出发点。假设 cache 控制器适用于 5.9 节图 5-40 所描述的简单的直接映射 cache，但是需要增加一个写缓冲器，其容量为 1 个块。

回忆一下，写缓冲器的目的是用来临时存储，因此处理器在发生脏块缺失时就不用等待两次存储器访问。比起在读新的块之前就写回脏块，写缓冲器缓存了脏块并且立即开始读新的块。而脏块随后被写入主存，同时处理器也在工作。

**5.16.1** [10] <5.8, 5.9> 如果处理器发出一个请求并且在 cache 中命中，同时一个块将从写缓冲器被写回到主存，此时会发生什么？

**5.16.2** [10] <5.8, 5.9> 如果处理器发出一个请求并且在 cache 中缺失，同时一个块将从写缓冲器被写回到主存，此时会发生什么？

**5.16.3** [30] <5.8, 5.9> 设计一个能够使用写缓冲器的有限状态机。

**5.17** cache 一致性考虑的是多个处理器看到的是同一个 cache 块。下表显示了两个处理器以及它们对一个 cache 块 X 中两个不同字的读/写操作（初始值 X[0] = X[1] = 0）。假定整数为 32 位。

| P1 | | P2 | |
|---|---|---|---|
| X[0] ++; | X[1] = 3; | X[0] = 5; | X[1] += 2; |

**5.17.1** [15] <5.10> 当执行一个正确的 cache 一致性协议时，列出给定 cache 块可能的值。如果协议没有保证 cache 一致性，至少列出一个 cache 块可能的值。

**5.17.2** [15] <5.10> 对于监听协议，列出每个处理器/cache 完成上面的读/写操作时正确的操作顺序。

**5.17.3** [10] <5.10> 在最好和最差情况下，完成列出的读/写指令，cache 缺失次数分别是多少？

存储器一致性考虑的是看到的多个数据项。下表显示了两个处理器以及它们对不同的 cache 块

的读/写操作（A 和 B 的初始值为 0）：

| P1 | | | | P2 | |
|---|---|---|---|---|---|
| A = 1; | B = 2; | A += 2; | print B; | C = B; | D = A; |

[495]

5.17.4 [15] <5.10> 若使用 5.10 节列出的保证一致性协议的假设，请列出 C 和 D 的值。

5.17.5 [15] <5.10> 如果假设不成立，那么至少列出一对 C 和 D 可能的值。

5.17.6 [15] <5.3，5.10> 对于写策略和写分配策略的多种组合，哪些组合可以简化协议的执行？

5.18 单芯片多处理器（chip multiprocessor，CMP）在单个芯片上有多个核和 cache。设计 CMP 的片上二级 cache 时都会进行权衡。下表列出了两个基准测试程序在私有二级 cache 和共享二级 cache 中的缺失率和命中延迟。假设每 32 条指令发生一次一级 cache 缺失。

| | 私有 | 共享 |
|---|---|---|
| 基准测试程序 A 的每指令缺失 | 0.30% | 0.12% |
| 基准测试程序 B 的每指令缺失 | 0.06% | 0.03% |

下表列出了命中延迟：

| 私有 cache | 共享 cache | 存储器 |
|---|---|---|
| 5 | 20 | 180 |

5.18.1 [5] <5.13> 对于每个基准测试程序，哪种 cache 设计更好？请用数据来支持你的结论。

5.18.2 [5] <5.13> 共享 cache 的延迟随着 CMP 规模的增长而增长。如果共享 cache 的延迟加倍，请选出最好的设计。当 CMP 中核的数量增加时，片下带宽就变成瓶颈，如果片下存储器访问延迟加倍，请选出最好的设计。

5.18.3 [10] <5.13> 讨论共享二级 cache 和私有二级 cache 对于执行单线程、多线程以及多道程序负载时的利弊情况；如果还有片上二级 cache，请重新考虑这些问题。

5.18.4 [15] <5.13> 假设两个基准测试程序的基本 CPI 都为 1（理想的二级 cache）。如果使用非阻塞 cache 能将同时发生二级 cache 缺失的平均次数从 1 变为 2，那么当使用共享二级 cache 时，性能能提升多少？如果是私有二级 cache，性能又能达到多少？

5.18.5 [10] <5.13> 假设新一代的处理器每 18 个月处理器核的数量就会翻倍。为了保证每个核的性能处于相同水平，那么一个 3 年后的处理器需要多少片下存储器带宽？

[496]

5.18.6 [15] <5.13> 考虑整个存储器层次结构，哪种优化可以改进同时发生的缺失数量？

5.19 在这个练习中，我们介绍了网络服务器日志的定义，并且为了改进日志处理速度，我们对代码优化进行了研究。日志的数据结构定义如下：

```
struct entry {
 int srcIP;   // remote IP address
 char URL[128]; // request URL (e.g., "GET index.html")
 long long refTime; // reference time
 int status;  // connection status
 char browser[64]; // client browser name
} log [NUM_ENTRIES];
```

假定日志的处理函数如下：

```
topK_sourceIP (int hour);
```

5.19.1 [5] <5.15> 对于给定的日志处理函数，在一个日志项中哪些字段将被访问？假设 cache 块为 64 字节，没有预取，那么给定的函数平均每个项会引发多少次 cache 缺失？

5.19.2 [10] <5.15> 为了改善 cache 的应用和局部访问，你会如何重新组织数据结构？请给出结构代

确定义。

**5.19.3** [10] <5.15>请举例说明另一种不同数据结构的日志处理函数。如果两个函数都很重要，为了改进整体性能，将如何重写程序？补充对代码片段和数据的讨论。

对于下面的问题，每对基准测试程序使用的数据来自“SPEC CPU2000 基准测试程序测出的 cache 性能”（http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/），如下表所示。

| | |
|---|---|
| a. | Hmmer / gcc |
| b. | mcf / swim |

**5.19.4** [10] <5.15>64KiB 的数据 cache 使用不同的组相联度，对于每个基准测试程序，每种缺失类型（强制、容量和冲突缺失）相应的缺失率分别是多少？

**5.19.5** [10] <5.15>为两个基准测试程序共享的一级数据 cache 选择组相联度，其中 cache 大小为 64KiB。如果一级 cache 是直接映射的，那么为 1MiB 的二级 cache 选择组相联度。

**5.19.6** [20] <5.15>请列举一个缺失率表的例子说明较高的相联度实际上能增加缺失率。并构建一个 cache 配置以及访问流来给出证明。

## 小测验答案

**5.1** 1 和 4。（3 是错误的，因为每个计算机的存储器层次结构的开销是不同的，但是在 2013 年开销最高的通常是 DRAM。）

**5.3** 1 和 4。更低的缺失代价可以允许使用更小的 cache 块，因为没有更多的延迟；而更高的存储带宽通常导致更大的块，因为缺失代价只是稍微大了一些。

**5.4** 1。

**5.7** 1－a，2－c，3－b，4－d。

**5.8** 2。（大容量的块和预取都能降低强制缺失，因此 1 是错误的。）

497 ↓ 498

第6章

Computer Organization and Design: The Hardware/Software Interface

# 从客户端到云的并行处理器




多处理器和集群

*我打算看大球，用尽一切力量，我要么获得巨大的成功，要么一败涂地。我喜欢活得很洒脱。*

——*Babe Ruth*，美国棒球运动员

## 6.1 引言

*"在月球的山脉上，沿着阴影笼罩的山谷，前进，勇敢的前进！"阴影回应道——"如果你在寻找理想国！"*

——*詹姆斯·爱伦·彼，《理想国》，第4节，1849*

计算机设计者一直在寻求计算机设计的"黄金之城"（El Dorado）：只需将现有的多个较小计算机简单地连接在一起来构成功能强大的计算机。这就是**多处理器**（multiprocessor）产生的根源。在理想情况下，用户可以按照自己的支付能力购买足够多的处理器，从而获得对应数量的性能。因而，多处理器软件必须设计为能在不同数量处理器的情况下工作。如第1章所述，无论是数据中心还是微处理器，功耗已经成为一个首要问题。在软件可以有效地使用每个处理器的情况下，用很多小型高性能处理器代替大型低效能处理器，可在每单位焦耳上获得更高的性能。这样，对多处理器而言，可以通过可伸缩的性能来提高功效。

⊃ 多处理器：至少含有两个处理器的计算机系统。与之对应的概念是单处理器（uniprocessor），它仅含一个处理器。

由于多处理器软件支持可变数量的处理器，并且一些设计支持在受损硬件上正常工作；也就是说，如果在包含 $n$ 个处理器的多处理器系统中有一个处理器失效，该系统将继续使用 $n-1$ 个处理器提供服务。因此，多处理器也提高了可用性（见第5章）。

高性能意味着处理单独一个任务时的高吞吐量。这被称作**任务级并行**（task-level parallelism）或**进程级并行**（process-level parallelism）。这些任务是独立的单线程应用程序，这在多处理器计算机中非常重要并且普遍使用。与之相对的方法是在多个处理器上运行一个作业。我们使用术语

**并行处理程序**（parallel processing program）来表示同时运行在多个处理器上的单一程序。

- 任务级并行或进程级并行：通过同时运行独立程序的方法来利用多处理器。
- 并行处理程序：同时运行在多个处理器上的单一程序。

在过去数十年里，很多科学问题都需要更快的计算机，同时这些问题也用于评价新型的并行计算机。这些问题中有些在今天处理起来很简单，使用由封装在不同独立服务器上的多个微处理器组成的**集群**（cluster）即可完成计算（见 6.7 节）。除了科学问题以外，集群还可以运行对等请求应用程序，如搜索引擎、Web 服务器、电子邮件服务器和数据库。

500 ~ 502

- 集群：通过局域网连接的一组计算机，其作用等同于一个大型的多处理器。

如第 1 章所述，多处理器因功耗问题已成为研究焦点，未来处理器性能的提高显然不再是依赖更高的主频和改进 CPI，而是借助于硬件并行。就像我们在第 1 章所看到的，为了避免名称上的冗长，称为**多核微处理器**（multicore microprocessor）而不是多处理器微处理器。因此，处理器在多核芯片内一般称为核（core）。核的数量预计以摩尔定律（Moore's Law）增长。这些多核处理器通常都是**共享内存处理器**（shared memory processor，SMP），因为它们通常共享一个单独的物理地址空间。我们会在 6.5 节更深入地讨论 SMP。

- 多核微处理器：在单一集成电路上包含多个处理器（“核”）的微处理器。基本上目前所有的台式机和服务器都是多核微处理器。
- 共享内存处理器：共享一个物理地址空间的并行处理器。

当今的技术意味着关心性能的编程人员必须成为并行编程的程序员，因为串行程序就意味着速度慢的程序。

业界面临的巨大挑战是如何构建易于正确编写并行处理程序的软硬件系统，不仅程序能够有效执行，而且性能和功耗可以随着单芯片内核的数量而相应改变。

微处理器设计的这种突然转变导致很多设计人员措手不及，因而会有很多关于微处理器设计方面的术语及其内涵的困惑。图 6-1 试图阐述串行（serial）、并行（parallel）、顺序（sequential）和并发（concurrent）等术语之间的差异。该图中的每一列代表固有顺序或并发的软件，每一行表示串行或并行的硬件。例如，编写编译器的程序员认为编译器是顺序程序，因为编译的主要过程包含分析、代码生成和优化等。与之相反，编写操作系统的程序员一般认为操作系统是并发程序，因为操作系统需要协同处理一个计算机中多个独立作业产生的各种 I/O 事件。

| | | 软件 | |
|---|---|---|---|
| | | 顺序 | 并发 |
| 硬件 | 串行 | 在 Intel Pentium4 上运行的使用 Matlab编写的矩阵乘法 | 在 Intel Pentium4 上运行的 Windows Vista 操作系统 |
| | 并行 | 在 Intel Core i7 上运行的使用 Matlab 编写的矩阵乘法 | 在 Intel Core i7 上运行的 Windows Vista 操作系统 |

图 6-1 硬/软件分类以及若干并发应用程序与并行硬件的对比实例

图 6-1 说明了以下两点：第一，并发软件可以运行于串行硬件上（如操作系统可以运行在 Intel Pentium 4 单处理器上），也可以运行于并行硬件上（如操作系统可以运行在 Intel Core i7 上）；第二，反过来顺序软件也是类似的。如 Matlab 程序员认为矩阵乘是顺序执行的，但是它可以串行地在 Intel Pentium 4 上运行，也可以并行地在 Intel Core i7 上运行。

也许你会认为并行的唯一挑战是如何使一个固有顺序执行的软件在并行硬件上获得更高性能，但实际上如何让并发程序在多处理器上随处理器数量增加而提高性能也是一个难点。为了加以区别，本章后面的部分使用并行处理程序（parallel processing program）或并行软件（parallel software）表示运行在并行硬件上的顺序软件或并发软件。本章下一节讲述为什么很难
503 编写高效的并行处理程序。

在进一步讨论并行方法之前，我们需要回顾一下前面章节的下述内容：

- 第2章，2.11节：并行与指令：同步。
- 第3章，3.6节：并行性和计算机算术：子字并行。
- 第4章，4.10节：指令级并行。
- 第5章，5.10节：并行与存储器层次结构：cache一致性。

**小测验**

是非判断题：为了从多处理器获得好处，应用程序必须是并发的。

## 6.2 创建并行处理程序的难点

并行的难点不在于硬件，而是目前只有极少量重要的应用程序经过重新编写后能在多处理器上，从而获得更快的执行时间。事实上，在多处理器上编写程序来提高执行效率很困难，而且随着处理器数量的增加会变得更加困难。

为什么会这样呢？为什么并行处理程序相对于顺序程序更加难开发呢？

首要原因是必须使用并行处理程序才能在多处理器上获取更高性能或功耗利用率；否则，就只能在单处理器上使用顺序程序，因为编写顺序程序相对较简单。事实上，单处理器设计技术（如超标量和乱序执行）充分利用了指令级并行（见第4章），而且通常不需要程序员的介入。这些技术不需要改写程序，因此程序员不做任何事情就可以在新的计算机上获得更高性能。

为什么编写更快的并行处理程序非常困难（尤其是让执行速度可随处理器数量的增加而增加）？在第1章中我们打了个比方，让8个记者同时编写同一故事，希望获得8倍的速度完成该项工作。为了实现目标，任务必须被分解为等量的8份，否则会有一些记者处于空闲状态等待其他工作量较大的人员完成任务。另外一个影响性能的障碍是记者们必须花费大量时间进行互相交流，而不是专心编写自己所负责的那部分故事。无论是这个类比还是并行编程，都要面
504 临如下挑战：调度，将任务分割成可并行的部分，负载均衡，同步时间和通信开销。而且，相对于使用更多记者完成一篇新闻报道，使用多处理器完成并行编程要复杂得多。

我们在第1章中还讨论了另外一个障碍，即Amdahl定律。它提示我们为了充分利用多核，程序中任何一个很小的部分都需要并行化。

**例题：加速比的挑战**

如果希望在100个处理器上获得加速比90，请问原始计算中最多有多少可以是顺序执行的呢？

**答案**

根据第1章描述的Amdahl定律：

改进后的执行时间 = 受改进影响的执行时间 / 改进量 + 未受改进影响的时间

使用加速比的形式重新表示Amdahl定律：

加速比 = 改进前的执行时间 /((改进前的执行时间 − 受影响的执行时间)
+ 受影响的执行时间 / 改进量)

该公式通常被改写为假定改进前的执行时间为 1 个时间单元的形式，受改进影响的执行时间可以视作与原始执行时间的比值：

加速比 = 1/((1 − 受影响的执行时间比例) + 受影响的执行时间比例 / 改进量)

将加速比替换为 90，将改进量替换为 100，代入上述公式中：

90 = 1/((1 − 受影响的执行时间比例) + 受影响的执行时间比例 /100) 505

然后简化该公式并对受影响的执行时间比例进行求解：

$$90 \times (1 - 0.99 \times \text{受影响的执行时间比例}) = 1$$

$$90 - (90 \times 0.99 \times \text{受影响的执行时间比例}) = 1$$

$$90 - 1 = 90 \times 0.99 \times \text{受影响的执行时间比例}$$

$$\text{受影响的执行时间比例} = 89/89.1 = 0.999$$

因此，为了在 100 个处理器上获得加速比 90，顺序执行部分最多占 0.1%。

然而，还是有大量具有固有并发特征的应用程序。 □

**例题：加速比的挑战：更大规模的问题**

执行两个加法：一个加法是 10 个标量的求和，另一个加法是一对 10 × 10 二维矩阵的求和。我们目前假设只有矩阵求和可以并行化，以后会看到如何对标量求和进行并行化。使用 10 个和 40 个处理器达到的加速比分别是多少呢？如果矩阵维数是 20 × 20 呢？

**答案**

我们假定性能是加法时间 $t$ 的函数，并且假定有 10 次加法不能从并行处理器中获益，100 次加法可以获益。如果在单处理器上的执行时间为 $110t$，那么在 10 个处理器上的执行时间是

改进后的执行时间 = 受改进影响的执行时间 / 改进量 + 未受影响的执行时间

改进后的执行时间 = $100t/10 + 10t = 20t$

所以使用 10 个处理器的加速比是 $110t/20t = 5.5$。使用 40 个处理器的执行时间是

$$\text{改进后的执行时间} = 100t/40 + 10t = 12.5t$$

所以使用 40 个处理器的加速比是 $110t/12.5t = 8.8$。因此，对于该问题规模，我们使用 10 个处理器达到了潜在加速比的 55%，但是使用 40 个处理器仅达到了潜在加速比的 22%。 506

当增大矩阵规模时会发生什么。顺序程序的执行时间为 $10t + 400t = 410t$。10 个处理器的执行时间是

$$\text{改进后的执行时间} = 400t/10 + 10t = 50t$$

所以 10 个处理器的加速比是 $410t/50t = 8.2$。40 个处理器的执行时间是

$$\text{改进后的执行时间} = 400t/40 + 10t = 20t$$

所以 40 个处理器的加速比是 $410t/20t = 20.5$。因此，对于较大的问题规模，我们使用 10 个处理器获得了大约 82% 的潜在加速比，使用 40 个处理器获得了超过 51% 的潜在加速比。 □

这些例子说明为了在多处理器上获得更高加速比，保持问题规模不变相对于增加问题规模会更加困难。为此我们引入两个术语来描述按比例缩放的方式。

**强比例缩放**（strong scaling）指保持问题规模固定所测得的加速比。**弱比例缩放**（weak scaling）指问题规模随处理器数量按比例增加所获得的加速比。假定问题规模 M 是主存中的工作集，处理器数量为 P，那么每个处理器所占用的内存对于强比例缩放大约是 M/P，对于弱比例缩放大约是 M。

- 强比例缩放：在多处理器上不需增加问题规模即可获得的加速比。
- 弱比例缩放：在多处理器上增加处理器数量的同时按比例增加问题规模所能获得的加速比。

注意，传统认知认为强比例缩放会比强比例缩放简单，但是存储器层次结构可能对这一传统认知产生影响。例如，如果弱比例缩放数据集不再适用于多核处理器高速缓冲存储器的最后一层，会导致系统性能比使用强比例缩放更加糟糕。

可根据不同的应用程序选择不同的比例缩放方法。例如，TPC-C 借贷数据库基准测试程序需要按与每分钟内的事务处理次数成比例地增加客户数量。这是因为如果银行装备了更快的计算机，我们也不能假定客户从此以后每天使用 100 次 ATM，很明显这是没有实际意义的。因此，如果希望证明系统可以将每分钟内处理的事务次数提高 100 倍，应当在顾客数量提高 100 倍的情况下进行实验。更大规模的问题需要更多的数据，这是弱缩放方法的特征。

最后一个例子说明了负载均衡的重要性。

**例题：加速比的挑战：负载均衡**

507 在上个例子中，我们使用 40 个处理器在较大问题规模中实现了加速比 20.5。其中假定了负载是完全均衡的。也就是说，40 个处理器中每一个都完成 2.5% 的工作。事实上，如果一个处理器的负载高于其他处理器，则加速比会受到影响。请计算其中一个处理器完成两倍于负载（5%）和五倍于负载（12.5%）时的加速比。对于其他处理器的利用率如何？

**答案**

如果一个处理器负责 5% 的并行负载，那么它需要完成 5% 乘以 400，即 20 次加法，其他的 39 个处理器分担剩余的 380 次加法。由于它们是同时运算的，我们可以取两者工作时间的最大值。

$$\text{改进后的执行时间} = \text{Max}(380t/39, 20t/1) + 10t = 30t$$

加速比从 20.5 降低至 $410t/30t = 14$。剩下的 39 个处理器的利用率不及原来的一半：当等待任务最重的处理器完成 $20t$，它们只执行了 $380t/39 = 9.7t$。

如果一个处理器完成 12.5% 的负载，它必须执行 50 次加法。公式为：

$$\text{改进后的执行时间} = \text{Max}(350t/39, 50t/1) + 10t = 60t$$

加速比进一步降低至 $410t/60t = 7$。其余的处理器的利用率不到（$9t/50t$）的 20%。这个例子说明了负载均衡的重要性：仅在一个处理器的负载是其他处理器的两倍时，加速比几乎降低了二分之一；一个处理器的负载是其他处理器的 5 倍时，加速比几乎降低到了原来的三分之一。 □

现在我们对并行处理的目标和挑战有了更好的理解，我们在这里给出本章后面内容的一个概览。下一节（6.3 节）介绍了一个比图 6-1 更古老的分类方法。另外，该节也给出了两种可以使串行程序运行在并行硬件上的指令集，即单指令多数据指令（SIMD）和向量指令（vector）。6.4 节介绍了多线程（multithreading），这个概念常常容易与多进程（multiprocessing）混淆，一部分原因是由于它们都依赖于程序中细微的并行性。6.5 节介绍了基本并行硬件的两种类型，它们的区别在于系统中所有处理器是否采用单一的物理地址。这两种类型的常见形式分别是共享存储多处理器（shared memory multiprocessor）和集群（cluster），而该节讲述的是前者。6.6 节介绍了一种来自图形硬件处理领域的相对较新的计算机，称为图形处理单元（GPU），它同样是共享一个物理地址空间的。（附录 C 更详细地介绍了 GPU）。6.7 节介绍了集群，这是使用多个物理地址空间的一个很常见的例子。6.8 节介绍了将多个处理器（可以是集群中的多个服务节点，也可以
508 是微处理器中的多个核）链接起来的分类方法。6.9 节介绍了通过以太网使集群中的多个节点进行通信的硬件和软件。该节展示了如何使用用户软件和硬件优化性能。我们接下来在 6.10 节探讨了寻找并行测试集程序的困难。该节也包含了一个简单但是很有启发意义的性能模型，这个模型可用于辅助应用程序及体系结构的设计。在 6.11 节，我们同时使用该模型和并行测试集程序对一个多核计算机和一个 GPU 进行比较。6.12 节展示了加速矩阵乘法这一旅程的最后也是最庞大的一个步骤。对于无法在 cache 中放下的矩阵，使用 16 个核的并行处理得到 14 倍的性能加速。本

章最后解析了一些常见谬误和陷阱，并进行了总结。

在下一节，我们介绍代表不同类型的并行计算机的英文首字母缩写，这些你可能以前已经见过了。

**小测验**

是非判断：强比例缩放不遵守 Amdahl 定律。

## 6.3 SISD、MIMD、SIMD、SPMD 和向量机

20 世纪 60 年代提出了并行硬件的一种分类方法，并且一直沿用至今。该分类基于指令流的数量和数据流的数量。图 6-2 给出了该分类方法。这样，常规的单处理器具有单一的指令流和单一的数据流，而常规的多处理器具有多个指令流和多个数据流。这两种类别分别称为 SISD 和 MIMD。

- SISD：单指令流单数据流的单处理器。
- MIMD：多指令流多数据流的多处理器。

| | | 数据流 | |
|---|---|---|---|
| | | 单 | 多 |
| 指令流 | 单 | SISD：Intel Pentium 4 | SIMD：x86 的 SSE 指令 |
| | 多 | MISD：至今没有实例 | MIMD：Intel Core i7 |

图 6-2 基于指令流和数据流数量的硬件分类和实例：SISD、SIMD、MISD 和 MIMD

在 MIMD 计算机上可以编写独立的程序并运行在不同的处理器上，而且这些程序可以协同完成一个共同的大型目标。但是编程人员通常仅编写单一程序，将其运行在 MIMD 计算机的所有处理器上，然后使用条件控制语句使不同的处理器执行不同的代码段。这种风格被称作**单程序多数据**（single program multiple data，SPMD），它是 MIMD 计算机编程的正常方式。

- SPMD：单程序多数据流。传统的 MIMD 编程模型，其中一个程序运行在所有处理器之上。

最接近多指令流单数据流（MISD）的处理器应该算是“流处理器”了，流处理器在流水线中对一个单独的数据流执行一系列计算：从网络中解析输入，分析数据，解压数据，查找匹配，等等。相反，SIMD 更常见一些。SIMD 计算机对向量数据进行操作。例如，一个单一的 SIMD 指令可以把 64 个数相加，只需要把 64 个数据流发送到 64 个 ALU，就可以在一个时钟周期内得到 64 个和。我们在 3.6 节和 3.7 节中见到的子字并行指令是另一个 SIMD 的例子；实际上，Intel 的 SSE 中的第二个 S 就代表 SIMD。 [509]

- SIMD：单指令流多数据流。同样的指令在多个数据流上操作，和在向量处理器中的一样。

SIMD 的优点是所有并行执行单元都是同步的，它们都对源自同一程序计数器（PC）的同一指令作出响应。从程序员的角度来看，非常接近于已经熟悉的 SISD。尽管每个单元都执行相同指令，但是每个执行单元都有自己的地址寄存器，这样每个单元都有不同的数据地址。因此，根据图 6-1，一个顺序应用程序编译后可能运行于串行硬件上并按 SISD 组织，也可能运行于并行硬件上按 SIMD 组织。

SIMD 的初衷是在几十个执行单元之间均摊控制单元成本。另外一个优点是降低指令宽度和空间——SIMD 只需要同时执行代码的一个副本，而消息传递的 MIMD 可能需要在每个处理器都有一份副本，共享存储器 MIMD 可能需要多个指令缓存。

SIMD 在使用 for 循环语句处理数组时最为有效。因此，为了在 SIMD 中并行工作，必须有大量相同结构的数据，一般称之为**数据级并行**（data-level parallelism）。SIMD 在使用 case 或 switch 语句时效率最低，此时每个执行单元必须根据不同的数据执行不同的操作。带有错误数据的执行单元必须被弃齐，而带有正确数据的执行单元将继续执行。若有 $n$ 个 case，SIMD 处理器将会以最高 $1/n$ 的性能运行。

⊃ 数据级并行：对不同数据执行相同操作所获得的并行。

激发了 SIMD 类型产生的阵列处理器正逐渐成为历史（见网站上的 6.15 节），但是直到现在 SIMD 的两种表示依然并存。

### 6.3.1 在 x86 中的 SIMD：多媒体扩展

正如第 3 章所述，在 1996 年，对于窄位宽整型数据的子字并行化激发了 x86 指令集中的多媒体扩展（MMX）指令的产生。随着摩尔定律的发展，加入了更多的指令，产生了最初的 SSE 扩展，现在为 AVX 扩展。AVX 支持同时执行 4 个 64 位的浮点数据。操作和寄存器的位宽编码到多媒体指令的操作码中。随着操作和寄存器位宽的变大，多媒体指令的操作码数量也在增加，现在已经有数百条 SSE 和 AVX 指令（见第 3 章）。

### 6.3.2 向量机

SIMD 的一个更加古老和优雅的称呼是向量体系结构，它几乎等同于西摩克雷（Seymour Cray）在 20 世纪 70 年代制造的计算机。向量机结构与具有大量数据并行的问题非常匹配。除了具有 64 个 ALU 可以同时计算 64 次加法之外，与早期的阵列处理器类似，向量体系结构将 ALU 流水化，从而在低成本下获得高性能。向量体系结构的基本理念是从存储器中收集数据元，并将它们按顺序放到一大组寄存器中，然后在寄存器中使用流水化的执行单元对它们依次操作，最后将结果写回存储器。向量体系结构的关键特征是拥有一组向量寄存器。这样，向量体系结构可能拥有 32 个向量寄存器，每个寄存器包含 64 个 64 位宽的数据元。

510

**例题 • 向量机与常规处理器在代码上的区别**

假设我们基于 MIPS 指令集体系结构进行扩展，增加向量指令和向量寄存器。向量操作的名称与 MIPS 原有操作相同，但是在其后增加一个字母“V”。例如，addv.d 表示将两个双精度向量相加。向量指令的输入可以是一对向量寄存器（addv.d），也可以一个是向量寄存器一个是标量寄存器（addvs.d）。对于后者，标量寄存器的值被用于所有操作的输入——addvs.d操作将会把标量寄存器的内容加到向量寄存器中每个数据元上。关键词 lv 和 sv 分别代表向量的读入和写回，它们完成整个双精度数据向量的读入或写回。lv 和 sv 后一个操作数是要读入或写回的向量寄存器；另一个操作数是一个 MIPS 的通用寄存器，用来给出向量在存储器中的起始地址。在简要说明之后，我们看下面的一小段代码如何从常规的 MIPS 代码转换成向量 MIPS 代码：

$$Y = a \times X + Y$$

其中 $X$ 和 $Y$ 是 64 位双精度浮点数的向量，并且最初保存在存储器中；$a$ 是一个双精度标量。（这个例子就是所谓的 DAXPY 循环，其构成了 Linpack 基准测试程序的内部循环。DAXPY 表示 double precision $a \times X$ plus $Y$。）假定 $X$ 和 $Y$ 的起始地址分别保存在 $s0 和 $s1 中。

**答案**

针对 DAXPY 的常规 MIPS 代码是：

```
        l.d     $f0,a($sp)      ;load scalar a
        addiu   $t0,$s0,#512    ;upper bound of what to load
loop:   l.d     $f2,0($s0)      ;load x(i)
        mul.d   $f2,$f2,$f0     ;a x x(i)
        l.d     $f4,0($s1)      ;load y(i)
        add.d   $f4,$f4,$f2     ;a x x(i) + y(i)
        s.d     $f4,0($s1)      ;store into y(i)
        addiu   $s0,$s0,#8      ;increment index to x
        addiu   $s1,$s1,#8      ;increment index to y
        subu    $t1,$t0,$s0     ;compute bound
        bne     $t1,$zero,loop  ;check if done
```

针对 DAXPY 的向量 MIPS 代码是：

```
l.d     $f0,a($sp)      ;load scalar a
lv      $v1,0($s0)      ;load vector x
mulvs.d $v2,$v1,$f0     ;vector-scalar multiply
lv      $v3,0($s1)      ;load vector y
addv.d  $v4,$v2,$v3     ;add y to product
sv      $v4,0($s1)      ;store the result
```

针对上面两段代码有几点值得注意。最引人注目的是向量处理器大大降低了动态指令带宽需求，仅用6条指令就完成了接近600条MIPS指令的工作。降低的原因一是向量操作是在64个数据元上同时进行的，二是MIPS中接近一半开销的循环指令在向量机代码中不存在了。正如你所想的一样，取指和执行次数的降低也会节省能耗。

另外一个重要的不同点是流水线相关的频率（见第4章）。在我们直接编写的MIPS代码中，每个add.d必须等待mul.d，并且每个s.d必须等待add.d，另外每个add.d和mul.d必须等待l.d。在向量处理器中，每条向量指令只会在每个向量的起始数据元阻塞，在随后的数据元都会顺畅地通过流水线。因此，流水线阻塞在每次向量操作时只会发生一次，而不是每次对向量数据元进行操作时都会发生一次。在这个例子中，MIPS中的流水线阻塞频率大约比MIPS的向量版本高64倍。当然，MIPS可以采用循环展开技术降低流水线阻塞频率（见第4章），但是指令带宽的巨大差异是无法减小的。

由于向量元素是相互独立的，它们可以并行地执行，很像AVX指令的子字并行。现在所有的向量计算机都带有多个并行流水线（称为向量通道，见图6-2和图6-3）的向量功能单元，每个流水线在一个时钟周期可以产生两个甚至更多的结果。

**精解** 上面的例子中循环次数恰好等于向量长度。当循环次数更小时，向量体系结构可以使用降低向量操作长度的寄存器。当循环次数更大时，我们可以增加记录代码来迭代全长度向量操作，最后处理剩余部分。后面的处理过程称作条状开发法（strip mining）。

### 6.3.3 向量与标量的对比

与常规的指令集体系结构（本部分将其称为标量体系结构）相比，向量指令具有几个重要的属性：

- 一条向量指令指定了大量需要完成的工作——它等价于执行一个循环。因而对取指和译码带宽的需求显著降低了。
- 通过使用向量指令，编译器或程序员隐含指明向量中每个结果的计算与同一向量中其他结果的计算是不相关的，因而硬件无需检查一条向量指令内的数据相关。
- 相对于MIMD多处理器，包含数据级并行的应用程序采用向量体系结构和编译器能够更加容易地编写高效代码。

- 硬件只需在两条向量指令之间对每个向量操作数检查一次数据相关，而不是对向量内每个数据元检查一次。相关检查次数的降低也会使得能耗降低。
- 访问存储器的向量指令具有确定的存取模式。如果向量的每个元素的地址都是连续的，那么从一组交叉存储器组中取回一个向量将会很快。因此，对整个向量而言，主存延迟的开销看上去只有一次，而不是对向量中每个字都有一次。
- 因为整个循环用具有预定义行为的向量指令所替换，循环转移所引起的控制相关就不存在了。
- 节省的指令带宽和相关检查以及存储器带宽的有效使用，使得向量体系结构在能耗方面优于标量体系结构。

由于这些原因，在同样的数据量前提下，向量操作比一组标量操作序列更快，并且如果应用程序可以频繁使用这些向量操作，就会促使设计者加入向量单元。

### 6.3.4 向量与多媒体扩展的对比

与x86 AVX多媒体指令扩展类似，向量指令可以指定多种操作。然而，多媒体扩展一般仅指定几种操作，而向量可以指定几十种操作。与多媒体扩展不同的是，向量操作中分量的数量不在操作码中，而是在一个单独的寄存器中。这个区别意味着不同版本的向量体系结构只需修改该寄存器的值，就能够实现不同的分量数量，并且能够保持二进制代码的兼容性。相比之下，在x86的多媒体扩展体系结构中（MMX、SSE、SSE2、AVX、AVX2、…），每次"向量"长度改变时都需要加入大量新的操作码。

还有一点与多媒体扩展不同，数据传输不必是连续的。向量同时支持按步长存取（strided access）和变址存取（indexed access），前者是硬件每隔$n$个存储器中的数据元读取一次，后者是按照数据项地址读取到向量寄存器中。变址存取也称作聚集分散（gather-scatter），变址的读取操作将内存中的数据元素聚集成连续的向量元素，变址的存储操作将向量元素分散到内存中。

与多媒体扩展类似，向量机可以灵活地支持不同数据宽度，因此它既可以在32个64位数据上进行向量操作，也可以在64个32位数据、128个16位数据或者256个8位数据上进行向量操作。向量指令的并行特性可以使其采用深度流水的功能单元、并行功能单元阵列或并行功能单元与流水功能单元的组合来实现。图6-3说明了如何采用并行流水线执行一条向量加法指令来提高向量的性能。

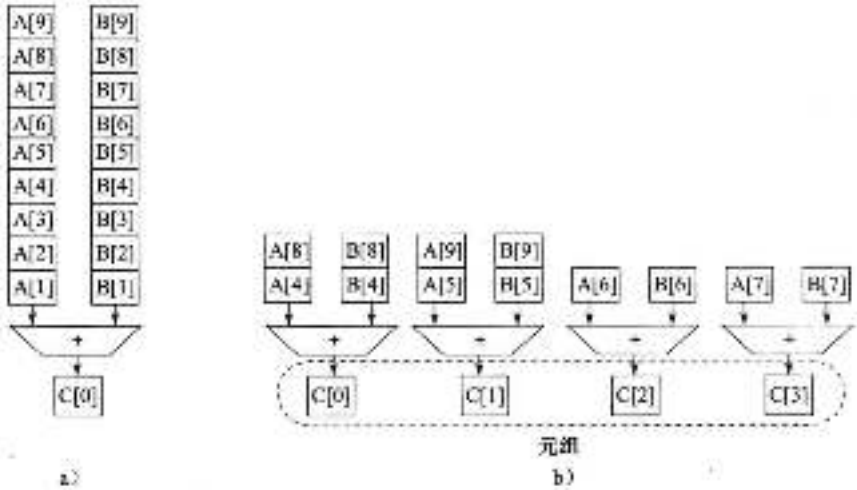



图6-3 使用多个功能单元来提升单个向量加法指令，C = A + B。左侧的向量处理器有一条加法流水线，并且可以在一个周期完成加法操作。右侧的向量处理器有4条加法流水线，并且可以在一个周期完成4条加法操作。一条向量加法指令的数据元素被分叉地放到4个通道中

向量算术运算指令通常只允许一个向量寄存器的元素 N 与另一个向量寄存器的元素 N 进行计算。通过多个向量通道（vector lane）的方式构建高度并行化的向量单元，极大地简化了高度并行化的向量单元的实现。由于有高速的通道，我们可以通过增加更多的通道数量来提高向量单元的最大吞吐率。图 6-4 展示了一个四通道的向量单元的结构。因此，通过将通道数从一个增至 4 个，使每条向量指令的周期数减少了大约 4 倍。由于多通道具有优势，故应用程序和体系结构都必须支持长向量。否则，指令会很快地执行完毕却得不到新的指令去执行，而去要求像第 4 章介绍的指令级并行提供足够的向量指令。 513

⇨ 向量通道：一个或多个向量功能单元与一部分向量寄存器。由为提高交通流量的高速公路的通路数启发而来，多个通道同时执行向量操作。

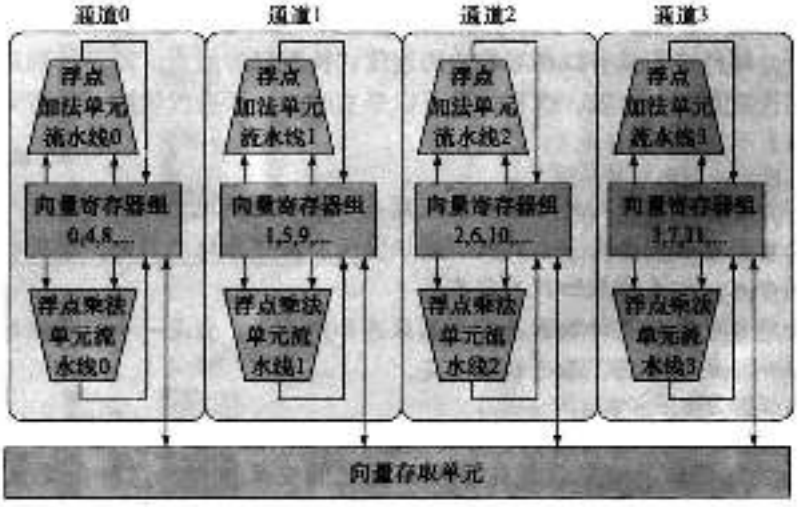



图 6-4 有 4 个通道的向量处理器的结构。向量寄存器等量地分配给每个通道，每个通道所占有的寄存器直接相隔为 4。图中画出了三个向量功能单元：一个浮点加法器，一个浮点乘法器和一个存取单元。每一个向量算术单元都包含 4 个执行流水线，每个通道一个，每条流水线执行一条指令。注意向量寄存器的每一部分是如何只需要为自己的通道提供足够的读和写端口（见第 4 章）的

总的来说，向量体系结构是执行数据平行处理程序的一种有效途径；相对多媒体扩展，向量机与编译器技术更加接近；并且相对于对 x86 体系结构进行多媒体扩展，向量技术更加容易随时间推移而得到不断改进。

给出了这些经典的分类方法，我们接下来看看如何发掘指令的并行流来提高一个单独处理器的性能，我们还会将该方法应用到多处理器中。

**小测验**

是非判断：以 x86 为例，多媒体扩展可以被视作一种采用短向量的仅支持顺序向量数据传输的向量体系结构。 514

**精解** 在了解了向量体系结构如此之多的优点之后，为何向量机却没有在高性能计算领域之外流行呢？主要原因包括：向量寄存器的巨大状态增加了上下文切换时间；向量存取产生的缺页故障难以处理；SIMD 指令也可以获得向量指令的部分优势。另外，只要指令级并行可以提供摩尔定律要求的性能提升，就没有理由要去改变体系结构的类型。

**精解** 向量和多媒体扩展的另外一个优点是一个长量指令集体系结构更易于扩展，从而使其提高数据并行操作的性能。

**精解** Intel 的 Haswell x86 处理器支持 AVX2 指令集，AVX2 指令级只有聚集（gather）操作而没有分散（scatter）操作。

## 6.4 硬件多线程

从程序员的角度来看，**硬件多线程**（hardware multithreading）就是一个和 MIMD 相关的概念。MIMD 依靠多个进程（process）或**线程**（thread）来努力使多个处理器处于忙碌状态，而硬件多线程允许多个线程以重叠的方式共享一个处理器的功能单元，以有效地利用硬件资源。为了支持共享，处理器必须为每个线程复制独立的状态。例如，每个线程必须拥有寄存器文件和 PC 的独立备份。存储器自身可以通过虚拟存储器机制实现共享，多道程序设计中已经支持这种方法。此外，硬件必须具有以相对较快的速度切换线程的能力。需要特别指出的是，线程切换相对进程切换应该更加有效，线程切换可以是实时的，而进程切换一般需要数百个到数千个处理器周期。

- 硬件多线程：在线程阻塞时处理器可切换到另一线程的实现。
- 进程：一个进程包含一个或多个线程、地址空间和操作系统。因此一次进程切换通常需要操作系统的介入，但是线程切换不需要。
- 线程：一个线程包含程序计数器、寄存器状态和内存栈。它是一个轻量级的进程；多个线程通常共享一个地址空间，而进程则不是。

硬件多线程主要有两种实现方法。**细粒度多线程**（fine-grained multithreading）在每条指令执行后都进行线程切换，结果就是在多个线程之间交叉执行。这种交叉通常以循环方式进行，并在循环时，在每个时钟周期跳过处于阻塞状态的线程。为了实现细粒度多线程，处理器必须能够在每个时钟周期进行线程切换。细粒度多线程的一个主要优点是可同时隐藏由短阻塞和长阻塞引起的吞吐量损失，因为当一个线程阻塞时可以执行其他线程的指令。细粒度多线程的主要缺点是降低了单个线程的执行速度，因为就绪状态的线程会因为其他线程而被迟执行。

- 细粒度多线程：硬件多线程的一种形式，每条指令执行之后都进行线程切换。

**粗粒度多线程**（coarse-grained multithreading）是细粒度多线程的一种替代方案。粗粒度多线程仅在高开销阻塞时才进行线程切换，如最后一级缓存缺失。这种改变对高速的线程切换降低了要求，并且几乎不会降低单个线程的执行速度，因为粗粒度多线程仅在当前线程遇到高开销阻塞时才会发射其他线程的指令。然而，粗粒度多线程有一个严重的缺点：它在隐藏吞吐量损失的能力方面受限，特别是短阻塞。这种限制源自粗粒度多线程中的流水线启动开销。因为粗粒度多线程处理器从单一线程发射指令，在阻塞发生时，必须清空或冻结流水线。阻塞之后开始执行的新线程必须在指令能够完成之前填充流水线。由于启动开销，粗粒度多线程更加适合用来降低高开销阻塞带来的性能损失，因为在这种情况下，与阻塞时间相比，流水线重新填充时间是可以忽略的。

515，516

- 粗粒度多线程：硬件多线程的一种形式，仅在一些重要事件（如最后一级缓存缺失）之后进行线程切换。

同时多线程（simultaneous multithreading，SMT）是硬件多线程的一个变种，它使用多发射动态调度流水线处理器的资源来挖掘线程级并行，并同时保持指令级并行（见第4章）。提出SMT的主要原因是在多发射处理器中通常有单线程难以充分利用的多个并行功能单元。而且，借助于寄存器重命名和动态调度（见第4章），不需考虑它们之间的相关性即可发射来自不同线程的多条指令；相关性的解决可以由动态调度机构来处理。

⊃ 同时多线程：多线程的一种形式，利用多发射、动态调度微体系结构中的资源实现多线程，从而降低多线程的开销。

SMT依赖于现有的动态机制，且不用每个周期切换资源。事实上，SMT总是执行来自多个线程的指令，由硬件将指令槽和重命名寄存器与适当的线程关联起来。

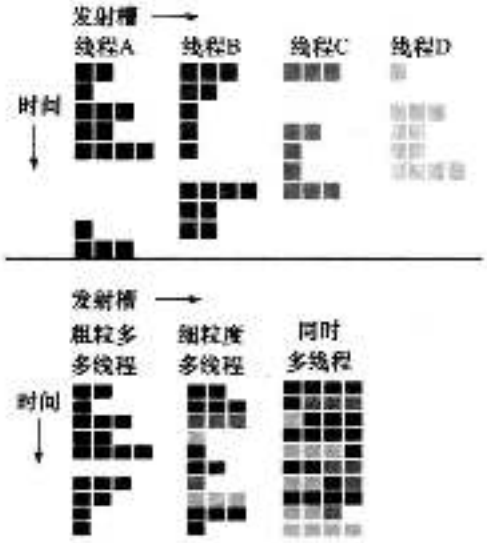



图6-5 4个线程如何以不同方式利用超标量处理器中的发射槽。上面的4个线程表示独立运行在不支持多线程的标准超标量处理器上的情况。下面给出了三个线程以三种不同多线程模式一起执行时的情况。水平方向表示每个时钟周期的指令发射量。垂直方向表示时钟周期的序列。空块（白块）表示在该周期没有利用相应的发射槽。不同灰度表示多线程处理器中的4个不同线程。尽管粗粒度多线程中额外的流水线启动开销在本图中没有标识，但其会导致更多的吞吐量损失

图6-5说明了使用一个处理器的不同配置开发超标量资源的能力上的差别。上面的部分表 517
示4个线程如何在不支持多线程的超标量处理器上独立运行。下面的部分表示4个线程如何以3种不同的多线程方式在处理器上更加有效地运行：

- 支持粗粒度多线程的超标量。
- 支持细粒度多线程的超标量。
- 支持同时多线程的超标量。

在不支持硬件多线程的超标量处理器中，指令发射槽的使用受到指令级并行性的限制。而且，绝大多数阻塞，如指令缓存缺失，会使整个处理器空闲。

在粗粒度多线程超标量处理器中，通过切换到其他使用该处理器资源的线程可以部分隐藏长阻塞。尽管这能降低完全空闲的时钟周期数量，但是流水线的启动开销仍然会带来空闲周期，并使ILP受到限制，也就是说，并非所有发射槽都能得到有效利用。在细粒度多线程中，线程的交叉执行几乎不会出现发射槽全空的情况。但是，由于在一个给定的时钟周期仅有单一线程发射指令，指令级并行的限制仍会导致某些时钟周期出现空闲发射槽。

在SMT中，线程级并行和指令级并行都得到充分利用，在一个时钟周期多个线程共同使用发射槽。理想情况下，发射槽的使用仅受多个线程间资源失衡和资源可用性的限制。实际上，还有一些其他因素限制可用发射槽的多少。尽管图6-5大大简化了这些处理器的真实操作

情况，但是它确实从整体上给出了多线程潜在的性能优势，特别是 SMT。

图 6-6 画出了在 Intel Core i7 960 的一个处理器上运行多线程时的性能和能耗优势，Intel Core i7 960 支持两个线程。平均加速比为 1.31，这对于有少量额外资源执行硬件多线程的情况来说不算坏。平均能耗效率提升为 1.07，效果很好。总之，对于在能耗不变的前提下性能得到了提升总是使人高兴的。

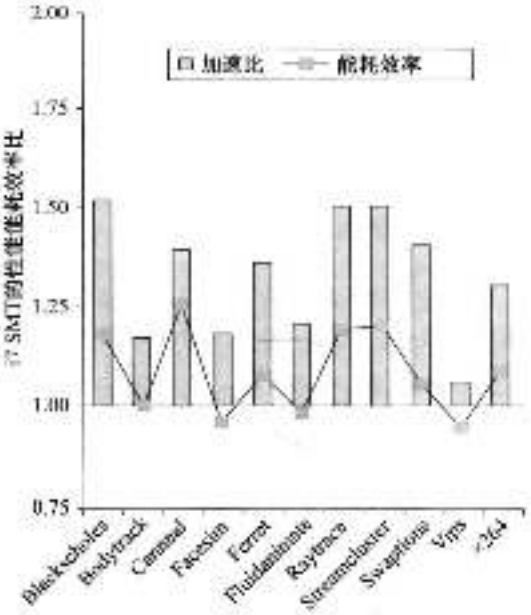



图 6-6 在 i7 处理器上的一个核上使用多线程运行 PARSEC 测试集程序（见 6.9 节），平均加速比为 1.31，功耗效率提升 1.07。该数据是由 Esmaeilzadeh 等 [2011] 收集分析得来

现在我们看到了如何通过多个线程更有效地使用一个处理器的资源，接下来看看如何利用多线程来发掘多处理器的资源。

**小测验**

1. 是非判断：多线程和多核都依赖并行来获得更高效率。
2. 是非判断：同时多线程（SMT）使用线程提高动态调度的乱序处理器的资源使用率。

## 6.5 多核和其他共享内存多处理器

尽管硬件多线程在很小的代价下提升了处理器的效率，但在过去的 10 年中的主要挑战是通过有效地编程利用单个芯片上数量不断增长的处理器以使性能以摩尔定律继续增长。

由于对原有程序进行重写，使之在并行硬件上很好地运行有困难，一个自然的问题是计算机设计者如何简化该工作。一种方法是为所有处理器提供一个共享的单一物理地址空间，以使程序不必考虑它们的数据在哪里，只要知道程序能够并行执行就可以了。在这种方法中，一个程序的所有变量对其他任何处理器在任何时刻都是可见的。另一种方法是每个处理器采用独立的地址空间，则必须进行显式共享；我们将在 6.7 节描述这种情况。当物理地址空间公用时，通常由硬件提供 cache 一致性，以便保证共享存储器的一致性（参见 5.8 节）。

518
|
519

综上所述，为程序员提供跨越所有处理器的单一物理地址空间的多处理器称为共享内存多处理器（SMP）——对于多核芯片总是这样——尽管更加准确的术语应该是共享地址多处理器（shared-address multiprocessor）。处理器通过存储器中的共享变量互相通信，所有处理器都能通过存取指令访问任何存储器位置。图 6-7 给出了 SMP 的典型组成。注意，即使这些系统共享同一个物理地址空间，它们仍然可以在自己的虚拟地址空间中单独地运行程序。

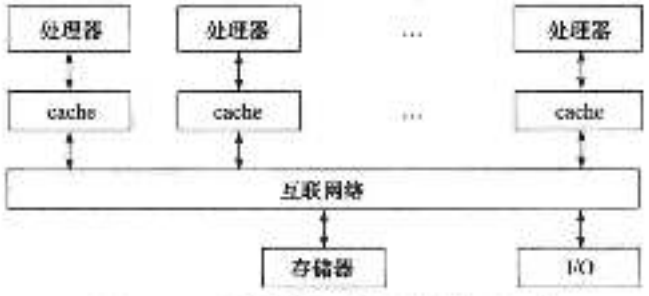



图 6-7 一个共享内存多处理器的典型组成

单一地址空间的多处理器有两种类型。第一种类型的访存时间不依赖于是哪个处理器提出访存请求，也无论要访存哪个字。这类机器称为**统一存储访问**（Uniform Memory Access，UMA）多处理器。对于第二种类型，一些访存请求会比其他的快，这取决于是哪个处理器访问哪个字，这是由于主存被分割并分配给同一个芯片上的不同的处理器或内存控制器。这类机器称为**非统一存储访问**（Nonuniform Memory Access，NUMA）多处理器。NUMA 多处理器的编程难度要高于 UMA 多处理器，但 NUMA 机器可以扩展到更大规模，并且 NUMA 访问附近的存储器时具有较低的延迟。

- **统一存储访问**：无论访存的是哪个处理器，也无论访存的是哪个字，访存时间都大致相同的多处理器。
- **非统一存储访问**：使用单一地址空间多处理器的一种类型，某些存储访存速度高于其他访存，访存速度与访问哪个处理器及访问哪个字相关。

由于处理器并行执行一般都需要共享数据，所以它们在操作共享数据时需要进行协调；否则，一个处理器可能会在其他处理器尚未完成对共享数据的操作时就开始使用该数据了。这种协调称为**同步**（synchronization），正如我们在第 2 章看到的。在使用单一地址空间支持的共享时，必须提供一套独立的同步机制。一种方法是为每个共享变量使用**锁**（lock）。在一个时刻只能有一个处理器获得锁，其他需要操作该共享数据的处理器必须等待，直到该处理器解锁该变量为止。第 2 章 2.11 节描述了 MIPS 中关于锁操作的指令。

- **同步**：对可能运行于不同处理器上的两个或者更多进程的行为进行协调的过程。
- **锁**：一个时刻仅允许一个处理器访问数据的同步装置。 520

**例题** 一个共享地址空间的简单并行处理程序

假设我们需要在一个处理器数量为 64 的共享存储多处理器计算机上对 64 000 个数求和，该计算机具有统一的存储器访问时间。

**答案**

第一步是保证每个处理器的负载是均衡的，所以我们将这组数分成等量的子集。由于该机器具有单一的存储器空间，因此我们不把这些子集分配到不同的存储器空间上；我们只给每个处理器分配不同的起始地址。用 Pn 表示不同处理器的编号，取值范围在 0～63 之间。所有处理器启动程序运行一个循环来完成它们子集中数的求和：

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i += 1)
  sum[Pn] += A[i]; /*sum the assigned areas*/
```

（注意，在 C 程序代码中，i += 1 是 i = i + 1 的简写形式。）

下一步是将这 64 个部分和加起来，称为**约简**（reduction）。我们采用分而治之的方法。首先用一半处理器对部分和求和，然后再用四分之一处理器对新的部分和求和，以此类推直到获得最终的和。图 6-8 对约简的过程进行了说明。

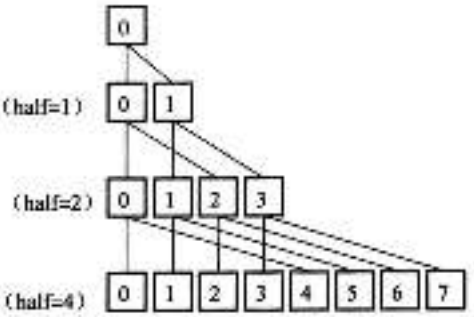



图 6-8 自底向上的最后 4 级求和过程。对于所有编号 i 小于 half 的处理器，将自己产生的部分和与编号 i + half 的处理器产生的部分和相加

- **约简**：处理一个数据结构并返回单一值的函数。 □

在该例子中，“消费者”处理器在读取由“生产

521 者”处理器写入结果的存储器位置之前必须同步；否则，消费者可能读取到数据的旧值。我们希望每个处理器拥有自己的循环计数器变量 i，因此我们将其声明为“私有”变量。下面是相应的代码（half 也是私有变量）：

```
half = 64; /*64 processors in multiprocessor*/
do
    synch(); /*wait for partial sum completion*/
    if (half%2 != 0 && Pn == 0)
        sum[0] += sum[half-1];
        /*Conditional sum needed when half is
        odd; Processor0 gets missing element */
        half = half/2; /*dividing line on who sums */
        if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1); /*exit with final sum in Sum[0] */
```

**硬件/软件接口** 由于人们长久以来对并行编程都有着浓厚的兴趣，现在已经有了上百种创建并行编程系统的尝试。一个相当限定但是很常用的例子就是 OpenMP。它只是一个带有一些编译器提示、环境变量和动态链接库的对现有标准语言进行扩展的应用程序接口（Application Programmer Interface，API）。它为共享存储器的多处理器提供了一个便于携带的、可伸缩并且简单的编程模型。它的最初目标是对循环和递归进行并行化。

大部分 C 语言编译器已经提供了对 OpenMP 的支持。在 UNIX 下的 C 语言编译器中使用 OpenMP API 的命令如下：

```
cc -fopenmp foo.c
```

OpenMP 使用 pragma 对 C 语言进行扩展，就像 C 宏预处理器命令#include 和#define 一样。举上面的例子中一样，我们要使用 64 个处理器，命令如下：

```
#define P 64 /* define a constant that we'll use a few times */
#pragma omp parallel num_threads(P)
```

这样，运行时库就会使用 64 个并行线程。

要将一个串行的 for 循环变为一个并行的 for 循环，并且要把任务等份地分割成我们指定的线程数，我们只需要写如下代码（这里假设 sum 初始为 0）：

```
#pragma omp parallel for
for (Pn = 0; Pn < P; Pn += 1)
  for (i = 0; 1000*Pn; i < 1000*(Pn+1); i += 1)
    sum[Pn] += A[i]; /*sum the assigned areas*/
```

522

对于递归，我们可以使用另一个命令告诉 OpenMP 什么是递归操作符和用什么变量代替递归运算的结果。

```
#pragma omp parallel for reduction(+ : FinalSum)
for (i = 0; i < P; i += 1)
    FinalSum += sum[i]; /* Reduce to a single number */
```

注意，现在就要靠 OpenMP 库来找到使用 64 个处理器来完成 64 个数字相加的最佳代码了。

尽管 OpenMP 使得编写并行代码更加简单，但是对于调试并不是很有帮助，所以很多并行程序员使用比 OpenMP 更复杂的并行编程系统，就像今天有很多程序员使用比 C 语言效率更高的编程语言一样。

⇒ OpenMP：一个为运行在 UNIX 或 Microsoft 平台上的 C、C++或 Fortran 语言的共享存储器多处理器的 API。它包含了一些给编译器的提示、一个库和一些运行时提示。

以上给出了一个经典的 MIMD 硬件和软件的例子，我们的下一个关于 MIMD 的例子继承于一个十分不同的体系结构，并且对于并行编程更具有挑战性。

**小测验**

是非判断：共享存储多处理器不能利用任务级并行性的优势。

**精解** 一些作者使用 SMP 作为同步处理器（symmetric multiprocessor）的简称，以此来说明无论哪个处理器访问存储器，延时都是一样的。这么做是为了和大规模 NUMA 多处理器做区别，因为两者都是共享一个地址空间。由于集群比大规模 NUMA 多处理器更为常见，在本书中，我们仍然使用 SMP 来表示它最原始的含义（对共享内存多处理器），并用它来区别使用多个地址空间的处理器，例如集群。

**精解** 除了共享物理地址空间之外，还有一种方法是使用独立的物理地址空间，但共享同一虚地址空间，由操作系统负责处理通信。这种方法已经有过尝试，但为了向注重性能的程序员提供一个实用的共享存储器抽象，它的开销显得过大。 523

## 6.6 图形处理单元简介

在现有体系结构中增加 SIMD 指令的一个最初理由是许多微处理器都连接到 PC 或工作站中的图形显示设备上，并且用于图形显示的处理时间所占比例越来越大。因此，当微处理器设计中可用晶体管数量随着摩尔定律的发展增加时，提高图形处理能力就变得有必要了。

提高图形处理能力的主要动力是计算机游戏产业，包括 PC 和专用的游戏终端（如 Sony PlayStation）。快速增长的游戏市场使许多公司增加了快速图形硬件方面的研发，这种正反馈使得图形处理能力的增长超过了主流微处理器的通用处理能力。

由于图形和游戏开发与微处理器开发有着不同的目标，故而图形处理采用了自己的一套处理风格和术语。随着图形处理器地位的上升，它们将自己命名为图形处理单元（Graphics Processing Unit，GPU），以便区分于 CPU。

如今人们可以只花几百美元就能买到带有上百个并行浮点运算单元的 GPU，这使得进行高性能计算更容易了。这种趋势与程序语言相结合，促进了人们对 GPU 计算的兴趣的增长同时使得 GPU 更易于编程。因此，很多科学计算和多媒体程序的编程人员开始犹豫是使用 CPU 还是 GPU。

（本节专注于使用 CPU 进行计算。要想知道使用 GPU 进行计算是如何与其作为图形加速卡这一原始功能进行结合的，请参阅附录 C。）

下面是 GPU 与 CPU 的几个主要差别：

- GPU 是补充 CPU 的加速器，因此它们不必执行 CPU 的全部任务。这种定位使得它们专注于图形方面的资源。对于一个同时具有 GPU 和 CPU 的系统来说，GPU 可以对某些任务执行效率很低甚至不能完成时，可以让 CPU 在必要的时候完成。
- GPU 解决的问题规模通常为几百 MB 到 GB，而不是几百 GB 到 TB。

这些差异导致体系结构的设计风格不同：

- 也许最大的不同就是 GPU 不像 CPU 一样依赖多级缓存来隐藏访问存储器的长延迟。事实上，GPU 依赖硬件多线程（6.4 节）来隐藏到访存的延迟。也就是说，在存储器请求和数据到达之间，GPU 会执行数以百计甚至数以千计的与该请求无关的线程。 524
- GPU 的主存是面向带宽的而不是面向延迟的。甚至有面向 GPU 的特殊图形 DRAM 芯片，相对于面向 CPU 的 DRAM，它的宽度更大并能提供更大带宽。除此之外，GPU 存

储器历来都小于常规微处理器的存储器。在 2013 年，GPU 一般有不超过 4 - 6GiB 的存储器，而 CPU 一般在 32 - 256GiB 之间。最后，需要注意对于通用计算，必须将数据在 CPU 存储器和 GPU 存储器之间的传输时间包含进来，因为 GPU 是一个协处理器。

- 考虑到 GPU 是通过多线程并行执行来获取高存储器带宽的，除了多线程，GPU 还可以提供许多并行处理器（MIMD）。因此，每个 GPU 相比于 CPU 有更多的线程，并且拥有更多的处理器。

**01 硬件/软件接口** 尽管 GPU 是为众多应用程序中很小一部分设计的，但是一些程序员希望能以某种形式编制他们的应用，以利用 GPU 内潜在的高性能。在厌倦了使用图形 API 语言描述问题之后，他们开发了类 C 编程语言，可以直接在 GPU 上编程。NVIDIA 的 CUDA（compute unified device architecture）是其中一个例子，它使得程序员可以编写直接在 GPU 运行的 C 程序，尽管仍有一些限制。附录 C 给出了 CUDA 代码的例子。（OpenCL 是一个由多个公司发起的一种轻型编程语言，它可以提供很多 CUDA 中的功能。）

NVIDIA 决定将所有形式的并行都定义为 CUDA 线程（CUDA thread）。将这种最底层的并行作为编程原语，编译器和硬件可以在 GPU 上将上千个 CUDA 线程聚集起来使用各种类型的并行去执行：多线程、MIMD、SIMD 和指令级并行。这些线程被聚集成线程块，以 32 个为一组一起执行。GPU 内部的多线程处理器执行这些线程块，一个 GPU 一般由 8 - 32 个这种处理器组成。

### 6.6.1 NVIDIA GPU 体系结构简介

我们使用 NVIDIA 系统作为 GPU 体系结构的代表。特别地，我们使用 CUDA 并行编程语言中的术语并使用 Fermi 体系结构作为例子。

525 与向量体系结构一样，GPU 只对数据级并行问题才有效果。这两种体系结构都有聚集 - 分散数据传输，但是 GPU 处理器有比向量处理器更多的寄存器。与向量体系结构不同，GPU 也依赖于多线程 SIMD 处理器中的硬件多线程以隐藏访存延时（见 6.4 节）。

多线程 SIMD 处理器与向量处理器很相似，但是前者有更多的并行功能单元，而不像后者只有少数几个高度并行化的功能单元。

就像前面提到过的，一个 GPU 包含多个多线程的 SIMD 处理器；也就是说，GPU 是一个由多个多线程 SIMD 处理器组成的 MIMD 处理器。例如，NVIDIA 的 Fermi 结构有 4 种不同的配置，根据价格的不同分别含有 7、11、14 或 15 个多线程 SIMD 处理器。对于含有不同数量多线程 SIMD 处理器的 GPU，为了实现透明的伸缩性，GPU 使用线程块调度器（thread block scheduler）将线程块分配给多线程 SIMD 处理器。图 6-9 给出了一个简化的多线程 SIMD 处理器的结构图。

再往下深入一层，硬件产生、管理、调度并执行的机器目标代码是一个由 SIMD 指令组成的线程（thread of SIMD instruction），我们也称其为一个 SIMD 线程（SIMD thread）。它就是一个传统意义上的线程，但是它包含这些相互独立的 SIMD 指令。这些 SIMD 线程有它们自己的程序计数器并且它们运行在一个多线程 SIMD 处理器上。SIMD 线程调度器（SIMD thread scheduler）含有一个控制器，这个控制器可以告诉调度器那些 SIMD 指令线程已经准备就绪可以执
526 行了，并且将这些线程送给分派单元，然后分派到多线程 SIMD 处理器上执行。这个调度器同传统多线程处理器中的硬件线程调度器（见 6.4 节）一样，只是它调度的是 SIMD 指令。因此，GPU 硬件有两层硬件调度器：

1）线程块调度器（thread block scheduler）将线程块分配到多线程 SIMD 处理器上。

2）当 SIMD 线程准备就绪时，SIMD 处理器内部的 SIMD 线程调度器进行调度。

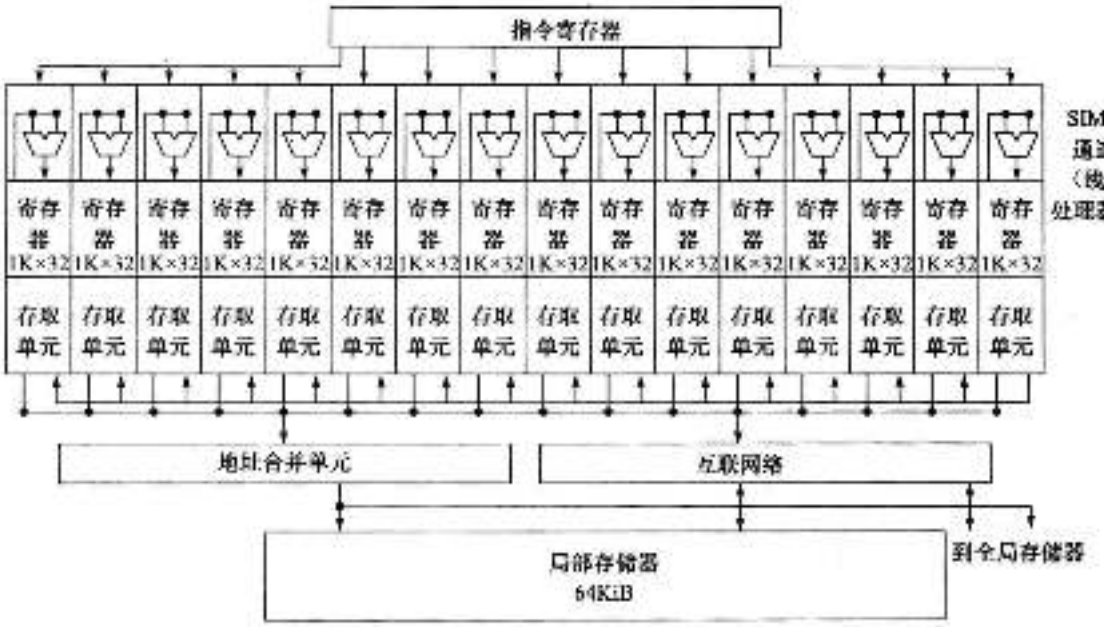



图 6-9 多线程 SIMD 处理器的数据通路的简略结构图。它有 16 个 SIMD 通道。SIMD 线程调度器用很多相互独立的 SIMD 线程来调度执行

这些线程的 SIMD 指令的宽度为 32，所以每一个 SIMD 指令线程都会对 32 个元素进行计算。由于线程是由 SIMD 指令组成，SIMD 处理器必须有并行功能单元来执行这些操作。我们称为 SIMD 通道（SIMD lane），它们同 6.3 节的向量通道非常相似。

**中** **精解** GPU 版本的不同，每个 SIMD 处理器中的通道数量也有所不同。对于 Fermi，每个 32 宽度的 SIMD 指令线程被映射到 16 个 SIMD 通道上。所以 SIMD 指令线程中的每一条指令需要两个时钟周期来完成。每一个 SIMD 指令线程都是同步执行。继续将 SIMD 处理器比作一个向量处理器，我们可以说它有 16 个通道，并且向量宽度为 32。这种宽且浅的特性使得我们称之为 SIMD 处理器而不是向量处理器，因为这样更直观一些。

根据定义，SIMD 指令线程之间是相互独立的，SIMD 线程调度器可以挑选任何准备就绪的线程去执行，而不需要考虑同一线程中指令的顺序。因此，若使用 6.4 节中的术语，则它使用的是细粒度多线程。

为了保存数据元素，一个 Fermi SIMD 处理器有着多达 32 768 个 32 位寄存器。就像向量处理器一样，这些寄存器根据向量通道（或称为 SIMD 通道）进行逻辑划分。每个 SIMD 线程有至多 64 个寄存器，所以我们可以认为一个 SIMD 线程有至多 64 个向量寄存器，且每个向量寄存器可以存放 32 个 32 位宽的数据元素。

由于 Fermi 有 16 个 SIMD 通道，所以共有 2048 个寄存器。每个线程可以从其中的一个向量寄存器中获得一个数据。注意，一个 CUDA 线程只是将一个 SIMD 指令中的线程进行纵向划分，对应于一个 SIMD 通道执行一个数据元素。请注意 CUDA 线程同 POSIX 线程非常不同；因为不能在一个 CUDA 线程中执行系统调用和同步操作。

## 6.6.2 NVIDIA GPU 存储结构

图 6-10 展示了一个 NVIDIA GPU 的存储结构。我们称每个多线程 SIMD 处理器专用的片上存储器为局部存储器（local memory）。它是由同一个多线程 SIMD 处理器中的所有 SIMD 通道所共享的，但是多个多线程 SIMD 处理器之间不共享它。我们称整个 GPU 和所有线程块共享的片外存储器为 GPU 存储器（GPU memory）。

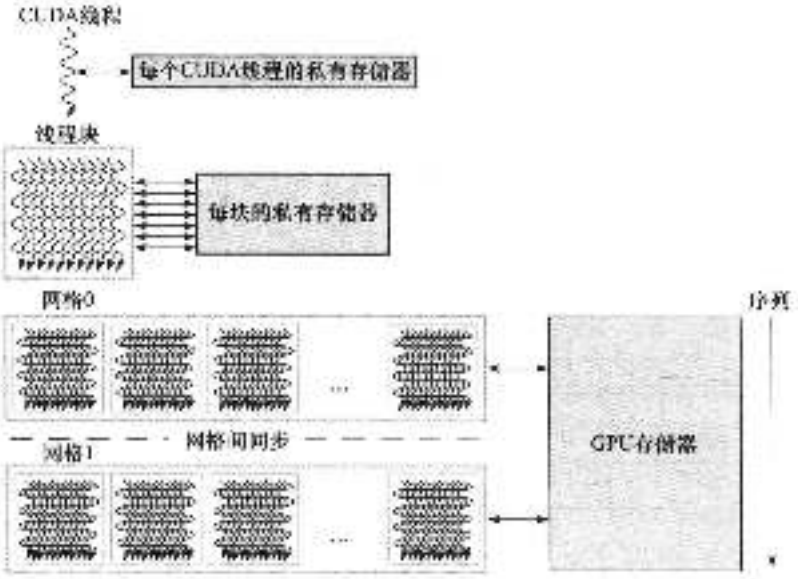



图 6-10 GPU 存储器结构：GPU 存储器被向量化的循环所共享，同一个线程块中的所有线程共享局部存储器

GPU 不依赖于大容量的 cache 来保存整个应用程序的工作集，而是依赖于小容量的流 cache 和大量的 SIMD 多线程来隐藏访问 DRAM 的访存延时，因为这些工作集通常为上百 MB。因此这些数据无法在多核处理器的最后一级 cache 中放下。为了使用硬件多线程来隐藏访问 DRAM 时的延时，就将处理器中用来放置 cache 的芯片面积在 GPU 中替换为计算资源和大量的寄存器，用来执行大量的 SIMD 线程。

527

**精解** 尽管隐藏访存延时是基本的原则，但是注意，最新的 GPU 和向量处理器都增加了 cache。例如，最近的 Fermi 结构增加了 cache，但是并不把它们作为为了减少访问 GPU 存储器访问次数的带宽过滤器，成为了少数不能通过多线程来隐藏访存延时的变量的加速器。为线程、函数调用和寄存器溢分而设计的局部存储器可以被认为是 cache，因为它们的延时对函数调用有影响。cache 也对减少能耗有好处，因为访问片上的 cache 需要的能耗比访问多个片外的 DRAM 要小很多。

## 6.6.3 GPU 展望

在高层次上，拥有 SIMD 指令扩展的多核计算机的确与 GPU 有一些共同特点。图 6-11 总结了它们之间的相似点与不同点。尽管 GPU 有更多的处理器和更多的通道数，但是它们两者都是通过使用多个 SIMD 通道来实现 MIMD 功能的。尽管 GPU 的硬件支持更多的线程，但是两者都是通过使用硬件多线程来提高处理器利用率的。尽管 GPU 使用的是小容量的流 cache，而多核计算机使用的是尽量把整个工作集都放进去的大容量多层次结构的 cache，但是两者都有 cache 结构。尽管 GPU 的物理主存小很多，但是两者都使用 64 位的地址空间。尽管 GPU 提供

| 特点 | 使用SIMD的多核 | GPU |
|---|---|---|
| SIMD处理器数目 | 4～8 | 8～16 |
| SIMD通道数目/处理器 | 2～4 | 8～16 |
| 支持SIMD线程的硬件数量 | 2～4 | 16～32 |
| 最大的cache容量 | 8MiB | 0.75MiB |
| 存储器地址大小 | 64位 | 64位 |
| 主存容量 | 8GiB～64GiB | 4～6GiB |
| 页面级存储保护? | 是 | 是 |
| 需要分页? | 是 | 否 |
| cache一致性? | 是 | 否 |

图 6-11 带有多媒体 SIMD 扩展的多核处理器与最近的 GPU 的相似点与不同点

页面集的内存保护，但是它目前还不支持请求页面调度。

SIMD 处理器也很像向量处理器。GPU 中的多个 SIMD 处理器像独立的 MIMD 核一样工作，就像向量计算机有多个向量处理器一样。这样，可以将 Fermi GTX 580 认为是一个带有硬件多线程功能的 16 个核的机器，每个核含有 16 个通道。两者最大的区别在于多线程，多线程对于 GPU 是最基本的概念，而在向量处理器中却不存在。

GPU 结构和 CPU 结构并没有相同的祖先；并没有缺少了哪一环来解释这种现象。这种不同寻常的继承关系，使得 GPU 并没有使用 CPU 领域中常用的术语，这让人们开始对 GPU 是什么以及 GPU 是如何工作的产生困惑。为了解决这个问题，图 6-12（从左到右）列出了本节使用过的更具有说明性的术语，首先是主流计算领域中最接近的术语，然后是 NVIDIA GPU 官方的术语（如果你感兴趣），最后是该术语简单的解释。这个“GPU 罗塞塔石碑”可以将本节的内容和想法与附录 C 中介绍的更传统的 CPU 描述联系起来。

| 分类 | 描述性名称 | 传统描述名称 | CUDA/NVIDIA 官方术语 | 教科书的定义 |
|---|---|---|---|---|
| 程序抽象 | 可向量化循环 | 可向量化循环 | 网格 | 在 CPU 上执行的一个可向量化循环由一个或多个可并行执行的线程块（循环展开后的程序块）组成 |
| | 向量化了的循环体/循环展开后的程序块 | （切分）向量化循环体/循环展开后的程序块 | 线程块 | 在 SIMD 多线程处理器上执行的被展开的循环体（向量化了的循环），由一个或多个 SIMD 指令线程组成。这些线程间数据通信通过局部存储单元实现 |
| | SIMD 单元执行序列 | 标量循环的一次迭代 | CUDA 线程 | 一个 SIMD 指令线程对应于一个 SIMD 的标量元素执行单元的执行序列。其计算结果的保存依赖于屏蔽寄存器以及预测寄存器的内容 |
| 机器目标代码 | SIMD 指令的一个线程 | 向量指令线程 | Warp 块 | 一个仅包含 SIMD 指令的传统线程，在 SIMD 多线程处理器中执行。其结果的保存依赖于单元素屏蔽寄存器的内容 |
| | SIMD 指令 | 向量指令 | PTX 指令 | 横跨多个 SIMD 元素执行单元（SIMD Lanes，一个 SIMD 执行单元包含对应向量内标量元素数量的执行模块）上执行的一个 SIMD 指令 |
| 处理硬件 | 多线程 SIMD 处理器 | （多线程）向量处理器 | 流多线程处理器 | 执行 SIMD 指令线程，一个 SIMD 多线程处理器独立于其他 SIMD 处理器 |
| | 线程块调度器 | 标量处理器 | 主线程调度引擎 | 分配多个线程块（循环展开后的程序块）到多个 SIMD 处理器上 |
| | SIMD 线程调度器 | 一个多线程处理器中的线程调度器 | Warp 调度器 | 当 SIMD 线程可执行时，硬件单元调度并发射 SIMD 指令，调度机制由一个用于确定 SIMD 线程执行的记分牌构成 |
| | SIMD 元素执行单元 | 向量元素执行单元 | 线程处理器 | SIMD 元素执行单元执行的一个 SIMD 线程对应于向量中的单一标量元素。其计算结果的保存依赖于屏蔽寄存器的内容 |
| 存储器硬件 | GPU 片外存储器 | 主存储器 | 全局存储器 | GPU 中所有多线程处理器均可访问的 DRAM 存储器 |
| | 本地存储器 | 本地存储器 | 共享存储器 | 一个多线程 SIMD 处理器私有的快速本地 SRAM 存储器，其他 SIMD 处理器不能访问 |
| | SIMD 元素执行单元寄存器 | 向量元素执行单元寄存器 | 线程处理器寄存器 | 横跨一个完整线程块（被展开的循环体），SIMD 元素执行单元内的寄存器 |

图 6-12 GPU 术语的快速介绍。第一列列出了硬件术语。这 12 个术语被分成 4 组。从上到下为：程序抽象、机器目标代码、处理硬件和存储器硬件

528 ~ 529

尽管GPU正在向主流计算进军，但是它们还是不能舍弃继续发展图形加速卡的功能。由于有很多硬件是为了加速图形处理的，因此只有当架构师开始考虑如何扩展GPU以使它可以进行更多类型的计算时，对GPU进行设计才会更有意义。

本节给出了两种共享地址空间的MIMD类型计算机，我们下节将要介绍每一个处理器都有自己独立的地址空间的并行处理器。这会使构建更大型的系统变得更简单。你每天都在使用的Internet服务就是依靠这些系统工作的。

**精辟** 尽管传统上的GPU是具有独立内存空间的处理器，但AMD和Intel都宣称已经有可以让GPU和CPU共享一个内存的"混合"产品。对于这种混合结构来说，挑战就是如何维护一个高的存储带宽，这也是GPU的基本问题。

**小测验**

是非判断题：GPU依靠图形DRAM芯片来减少访存延时，并以此来提高图形应用程序的性能。

## 6.7 集群、仓储级计算机和其他消息传递多处理器

对处理器来说，另一个共享地址空间的方法是每个处理器具有自己私有的物理地址空间。图6-13给出了具有多个私有地址空间的多处理器的典型组成。这种多处理器必须通过显式的**消息传递**（message passing）进行通信，传统上也把这类计算机称为消息传递计算机。只要系统提供**发送消息例程**（send message routine）和**接收消息例程**（receive message routine），协调工作就可以通过消息传递来完成，因为发送处理器知道何时发送消息，接收处理器也知道何时消息到达。如果发送者需要确认消息已经送达，那么接收处理器可以向发送者返回一个确认消息。

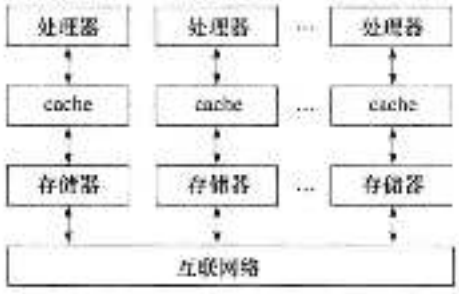



图6-13 具有多个私有地址空间的多处理器的组成，传统上称为消息传递多处理器。与图6-7中的SMP不同，互联网络不是在cache和存储器之间，而是在处理器-存储器的节点之间

- 消息传递：通过显式发送和接收信息的方式在多个处理器之间的通信。
- 发送消息例程：具有私有存储器的机器中一个处理器将消息发送给另一个处理器的例程。
- 接收消息例程：具有私有存储器的机器中一个处理器接收来自其他处理器消息的例程。

530 ~ 531

历史上曾经有过几次基于高性能消息传递网络构建大规模计算机的尝试。相对于使用局域网构建的集群，它们确实可以提供更高的性能。确实，今天很多超级计算机使用自己特有的网络。但是问题是它们比以太网这样的局域网花销更大。很少有应用程序能够为更高的性能支付更多的成本。

**硬件/软件接口** 相比于共享一致性cache，依赖于消息传递机制的计算机对于硬件设计者来说更容易构建（见5.8节）。这对于编程人员来说也有好处。那就是通信都是显式的，这意味着相比于共享一致性cache或隐式通信来说性能方面的惊喜更少。对于编程人员的坏处是把顺序程序移植到消息传递机制的计算机上更困难，因为所有的通信都需要提前考虑，否则程序就不会工作。cache一致性共享存储器允许硬件指明那些数据需要进行通信，这使得移植更容易。考虑到隐式通信的优点与不足，关于哪种方式最有利于获得高性能存在分歧。但

是在今天的市场上却并没有这种困惑。多核微处理器使用共享物理内存机制进行通信，而集群的节点之间使用消息传递机制进行通信。

一些并发程序可以在并行硬件上运行地很好，而与该硬件提供的是共享地址机制还是消息传递机制无关。特别的，使用任务级并行和通信比较少的应用程序——如 Web 搜索、邮件服务器和文件服务器——不需要共享地址机制就可以运行地很好。结果是，集群（cluster）已经变成了当今基于消息传递机制的并行计算机最普遍使用的例子。由于有独自的存储器，集群的每一个节点都运行操作系统的一个独立的副本。相反，微处理器的所有核在一个芯片上通过一个高速的互联网络相连，并且多个芯片共享的存储系统通过存储互联网络通信。存储互联网络有更高的带宽和更小的延时，可以让共享内存的多处理器有更好的通信性能。

⊃ 集群：通过标准的网络开关将 I/O 链接起来的计算机集合，以此来构建消息传递机制的多处理器。

从并行编程的角度来讲，将用户内存的负担分配到各个独立的部分提高了系统的可靠性（见 5.5 节）。由于集群是由通过局域网络连接起来的独立计算机组成，所以相比于共享存储器多处理器而言，在不影响整个系统性能的前提下替换其中的某一个计算机比较容易。从根本上讲，共享地址意味着很难在不需要大量操作系统和服务器设计进行大量工作情况下将每个处理器隔离开来并且进行替换。当一个服务器坏掉时，整个系统很容易降级。因此提高了可靠性。由于集群上的软件是运行在每个独立计算机上的操作系统中的，所以很容易断开一个计算机与互联网络的连接并且进行替换。 532

由于集群是由多个计算机和独立可配置的互联网络组成的，这种隔离使得在不卸载运行在集群上的软件的前提下对系统进行扩展变得很容易。

尽管与大规模共享存储器多处理器相比，集群在通信方面性能很弱，但它的低成本、高可用性和快速可扩展性使得它对于 Internet 服务提供者具有很大的吸引力。上亿人每天都在使用的搜索引擎就依赖于该技术。Amazon、Facebook、Google、Microsoft 和其他商业巨头都有多个数据中心，而每个数据中心都是由含有成千上万个服务器的集群构成。很显然，将多个处理器应用在 Internet 服务公司的做法已经获得了巨大的成功。

## 仓储级计算机

*每个人都可以构建一个快速的 CPU，关键是如何构建一个快速的系统。*

——*Seymour Cray*，超级计算机之父

为了支持上面提到的 Internet 服务，必须构建新的建筑、电力系统以及对 100 000 台服务器进行冷却。尽管它们可以归类为大型的集群，但是它们的体系结构和操作复杂很多。它们就像一个巨大的计算机，连接和安放 50 000 – 100 000 台服务器，造价、电力和冷却系统、服务器成本以及互联设备总共需要 15 亿美元。我们把这归类于一类新的计算机，叫做仓储级计算机（Warehouse-Scale Computer，WSC）。

**硬件/软件接口** WSC 中批处理最流行的架构是 MapReduce [Dean，2008] 和它们开源孪生兄弟 Hadoop。受 Lisp 中同名函数的启发，Map 首先对每个逻辑输入记录使用一个程序员提供的函数，然后 Map 在数以千计的服务器上运行并产生一个由关键值对组成的中间结果。Reduce 将这些分布的任务的输出结果收集起来，并使用另外一个程序员提供的函数来对其进行压缩。通过适当的软件处理，这两部分可以高度并行化并且易于使用。在

30分钟之内，一个编程新手就可以在上千个服务器上执行MapReduce。

例如，一个MapReduce程序要计算一大堆文档中每个单词出现的次数。下面是该程序的一个简化版本，它只给出了最内层的循环，并且假设在一个文档中所有单词只出现一次。

```
mapString key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
    emitIntermediate(w, "1"); // Produce list of all words reduceString key, Iterator values):
// key: a word
// values: a list of counts
    int result = 0;
    for each v in values:
    result += ParseInt(v); // get integer from key value pair
    Emit(AsString(result));
```

533

Map函数中使用的EmitIntermediate函数可以将文档中的每一个单词输出，同时还带有一个1值。然后Reduce函数逐个将同一个单词的所有value值加起来，这其中使用ParseInt()函数得到所有文档中每个单词的value值。MapReduce运行时环境将map任务和reduce任务调度到WSC中的服务器上。

在这种极端的规模下，WSC需要在电源分布、冷却、监控和操作上都做出创新，WSC可以算是20世纪70年代的超级计算机的后代——这使得Seymour Cray成为了当今WSC体系结构之父。他的超级计算机可以解决其他计算机无法解决的问题，但是却过于昂贵，以至于只有少数几个公司有能力购买它。而现在WSC的目标是为全世界提供信息技术，而不是专门为科学家和工程师提供高性能计算。因此，WSC在今天的社会中扮演着比Cray的超级计算机在那个年代更加重要的角色。

尽管它们都与服务器在目标上有一些共同点，但是WSC还是有以下3点主要区别：

1）大量简单的并行：对于服务器架构师，一个需要着重考虑的因素是目前市场上的应用程序是否具有能在并行硬件上运行得足够的并行性，以及为了发掘这些并行性而使用足够多的硬件是否代价过于高昂。但是WSC的架构师没有这方面的顾虑。首先，像MapReduce这样的批处理程序可以从大量需要独立处理的数据集中获利，例如网页抓取中数以亿计的网页。其次，交互式Internet服务应用，也称作**软件即服务**（software as a service，SaaS），可以从数以百万计的相互独立的交互式Internet服务用户中获利。在SaaS中，读和写之间依赖关系很少，所以SaaS基本上不使用同步操作。例如，查找操作只使用一个只读的索引，电子邮件通常是读和写独立的信息。我们将这种简单的并行称作请求级并行（request-level parallelism），因为很多独立的工作可以很自然地执行，并且只需要很少的通信和同步操作。

➲ 软件即服务：相比于出售那些安装并运行在用户计算机上的软件，软件是运行在远程的一个站点上的，并且是通过Internet（通常是面向用户的网页接口）向用户提供服务。SaaS用户是基于是否使用而非是否拥有来收费的。

2）经营成本：传统上，服务器架构师通常在有限的开销上设计可以达到最佳性能的系统，并且要时刻确保冷却系统的容量不会超过它的额定面积。他们经常忽略服务器的经营开销，就像经营开销和购买的开销比起来暗淡无光一样。WSC拥有更长的寿命——建筑和电力以及冷却系统通常使用超过10年的分期付款来购买——所以经营成本加起来：能耗、电源分布和冷却系统在这10年中总共超过WSC价格的30%。

3）规模以及规模带来的机遇和问题：为了建造一个WSC，你必须购买100 000台服务器

以及相关的设施，这意味着会有总额折扣。因此，WSC 的内部如此庞大以至于即使没有很多的 WSC 你也会得到规模经济。这种规模经济导致了云计算（cloud computing）出现，因为每 534
个单元更低的花费意味着云计算公司可以用比用户自己购买这些服务更低的价格将服务出租给用户以获得利润。规模经济的负面影响就是需要解决这种规模下的故障率。即使一个服务器有着高达 25 年（200 000 小时）的平均无故障时间（mean time to failure），WSC 架构师也要考虑每天会出现 5 个服务器故障的可能。5.15 节提到了 Google 测试得到的年均磁盘故障率（AFR）为 2% ~4%。如果每个服务器有 4 个磁盘并且它们的年均故障率为 2%，WSC 架构师应该每小时都发现一个磁盘故障。因此，容错性对于 WSC 架构师比服务器架构师要重要得多。

WSC 带来的规模经济使得长久以来人们梦寐以求的将计算变成一种设施的梦想成为现实。云计算意味着任何有好的想法、一个商业模式以及一个信用卡的人都可以在任何地点使用数以千计的服务器向全世界传播他的想法。当然，也存在阻挡云计算发展的障碍——例如安全、隐私、标准化以及 Internet 带宽的增长率——但是我们可以预见这些障碍都会被解决，并且 WSC 和云计算终将繁荣起来。

考虑到云计算的发展率，2012 年亚马逊网络服务宣布它每天都会增加相当于 2003 年亚马逊的所有设施的服务器数量，那时候亚马逊是一个拥有 6 000 名员工的年收入额为 52 亿美元的公司。

对于云计算而言，现在我们理解了消息传递机制多处理器的重要性，接下来要介绍将 WSC 中个节点连接起来的方法。归功于摩尔定律和每芯片上不断增加的核数，我们现在在芯片内部也需要互联网络，所以这些拓扑结构无论在小规模计算机上还是大规模计算机上都很重要。

**01 精解** MapReduce 架构在 Map 阶段的最后将关键值对进行移动和分类，并生成所有共享相同关键值的组。然后这些组会被传递给 Reduce 阶段。

**01 精解** 另一种大规模计算是网格计算（grid computing），所有计算机被分布在很大的范围之内，运行于其上的软件通过长距离的网络进行通信。最常见并且特殊的网格计算形式要属 SETI@ home 项目设计的。当数以百万计的计算机在空闲的时候，若有人开发了一个可以使用这些计算机的软件，并且将能够将任务分成独立的部分分配给这些计算机运行，这些计算机就会被征集起来并获得很好的利用。最早的一个例子是搜索地外文明的项目（Search for ExtraTerrestrial Intelligence，SETI），该项目是在 1999 年在加州大学伯克利分校启动的。超过 200 个国家的 500 万计算机用户签署了 SETI@ home 项目，其中有超过 50% 的非美国用户。到 2011 年年底，SETI@ home 网格计算的平均性能为 3.5PetaFLOPS。 535

**01 小测验**

1. 是非判断题：同 SMP 相同，消息传递机制的计算机基于锁来进行同步操作。
2. 是非判断题：集群有独立的存储器，所以每个都有一份操作系统的副本。

## 6.8 多处理器网络拓扑简介

多核芯片需要使用片上网络将各个核连接到一起，集群需要局域网将服务器连接到一起。本节讨论不同互联网络拓扑的优点与缺点。

网络成本包括开关的数量、每个开关连接到网络上的链路数量、每条链路的宽度（比特数）以及网络映射到芯片时链路的长度。例如，某些核或服务器可能是相邻的，而其他的可能

在芯片或数据中心的另一端。网络性能也是多方面的。它包括在一个无负载的网络中发送和接收消息的延迟，按照在给定时间周期内能够传输的最大消息数量所给出的吞吐量，由于网络冲突导致的延迟，以及由通信模式决定的可变性能。网络的另一个责任是容错。因为系统可能需要在存在部件受损的情况下继续工作。最后，在这芯片设计受能耗限制的时代里，不同组织结构具有不同功效，能耗可能超越其他考虑因素而成为主导因素。

网络通常绘制为图形表示，图中的每条边表示通信网络中的一条链路。在本章的图中，处理器-存储器节点用一个黑色方块表示，而开关用一个灰色圆形表示。我们假设所有链路都是双向的；也就是说，信息可以向两个方向流动。所有网络都由开关构成，开关负责建立处理器-存储器节点到其他开关的链接。第一个网络将若干节点组成的序列连接到了一起：



该拓扑叫作环（ring）。由于一些节点不是直接连接的，一些信息将不得不经过中间节点最终到达目标节点。

和总线不同——总线允许一组连线向所有相连的节点发送广播——环可以同时进行多个
536 传输。

因为有众多的拓扑可以选择，所以需要辨别这些不同设计的性能度量。主要有两个常用的性能度量。第一个是总网络带宽（network bandwidth），它是每条链路带宽与链路数量的乘积。该度量表示带宽的峰值。对于上面的环网络，如果处理器数量为 $P$，那么总网络带宽就是一条链路带宽的 $P$ 倍；一条总线的总网络带宽仅仅是该总线的带宽。

⇨ 网络带宽：非正式用语，用于表示网络传输速度的峰值；既可以指单一链路的速度，也可以指网络中全部链路的类同的传输速度。

为了不只评估最好情况下的性能，我们引入一个接近于最差情况的度量：**切分带宽**（bisection bandwidth）。它的计算是通过将机器分割为两半，然后将跨越假想分割线的链路带宽加起来。环的切分带宽是链路带宽的两倍，是总线链路带宽的一倍。如果单一链路和总线一样快，那么环在最差情况下是总线速度的两倍，而在最好情况下是总线的 $P$ 倍。

⇨ 切分带宽：多处理器中两个相等部分之间的带宽。这种测量可以表示对多处理器的最差拆分情况。

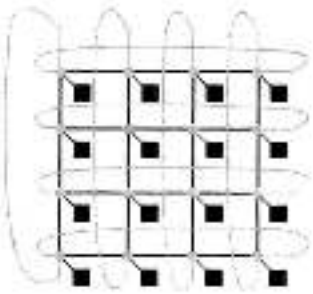
某些网络拓扑是非对称的，那么在切分网络时会产生一个问题：在哪里进行假想切分。由于切分带宽是一个针对最差情况的度量，因此答案就是选择会导致最差网络性能的切分方式。换句话说，就是计算所有可能的切分带宽，然后选择其中最小的一个作为最终结果。我们之所以选择这种最差情况，是因为并行程序常常受通信链中最薄弱链路的限制。

相对于环的另一个极端是**全连接网络**（fully connected network），其中每个处理器都与其他处理器具有一个双向链路。对全连接网络，总网络带宽是 $P\times(P-1)/2$，而对分带宽是 $(P/2)^2$。

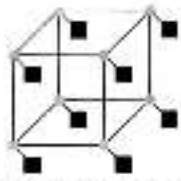
⇨ 全连接网络：通过专用通信链路连接所有处理器-存储器节点的网络。

全连接网络对性能的极大提升被成本的急剧增加所抵消了。这激励工程师不断创造出介于环的成本和全连接网络的性能之间的新型拓扑。评估是否成功，很大程度上依赖于计算机上所运行的并行程序负载的通信特征。

各种公开发布的不同拓扑可能难以计数，但是只有少数几个已经用于商业并行处理器中。图6-14给出了两种常见拓扑。



a）16个节点的2-D网格结构

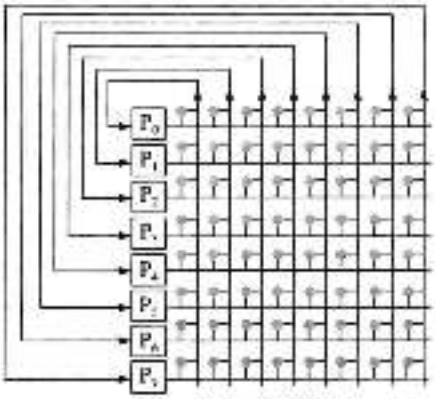


b）8个节点的n-立方树（由于8=2³，所以n=3）

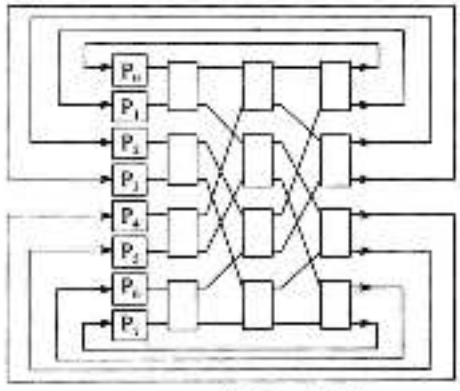
图 6-14 已经出现在商业并行处理器中的网络拓扑，其中灰色圆形表示开关，而黑色方块表示处理器－存储器节点。尽管一个开关可以有多个链路，但是通常只有一个连接到处理器。布尔 $n$ 维立方体拓扑是一个使用 $2n$ 个节点构成的 $n$ 维立走，每个开关需要 $n$ 个链路（并加上一个处理器链路），因而存在 $n$ 个最近相邻节点。这些基本拓扑常常会补充一些额外链路，从而提高性能和可靠性

除了在网络中每个节点都放置一个处理器之外，也可以在某些节点只保留开关。这些开关相对处理器－存储器－开关节点更小，因此可以放置得更密集一些，进而缩短距离提高性能。这样的网络一般称为**多级网络**（multistage network），因为信息需要多级传输才能到达目的地。多级网络的类型和单级网络是一样多的；图 6-15 给出了两种常见的多级结构。**全连接网络**或**交叉开关网络**（crossbar network）允许任何节点一次就可以通过网络与其他任何节点通信。Omega网络相对交叉开关网络使用更少的硬件（前者需要 $2n\log_2 n$ 个开关，后者需要 $n^2$ 个开关），但是消息之间可能会发生冲突，这取决于通信模式。例如，图 6-15 中的 Omega 网络在从 $P_0$ 向 $P_6$ 发送信息的同时，不能从 $P_1$ 向 $P_4$ 发送信息。 537

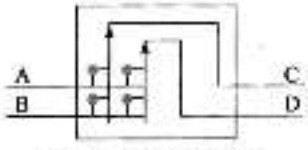
- 多级网络：每个节点提供一个小开关的网络。
- 交叉开关网络：任何一个需一次即可与其他任意一个节点通信的网络。




a）交叉开关




b）Omega网络




c）Omega网络开关盒

图 6-15 常见的八节点多级网络拓扑。本图中的开关相对前面的更加简单，因为本图的链路是单向的；数据从左边进入，从右边的链路退出。图 C 中的开关盒可以将 A 传送到 C，将 B 传送到 D，或将 B 传送到 C，将 A 传送到 D。交叉开关使用 $n^2$ 个开关，其中 $n$ 是处理器的数量，而 Omega 网络需要 $2n\log_2 n$ 个大的开关盒，其中每个开关盒逻辑上由 4 个更小的开关组成。在这种情况下，交叉开关网络需要 64 个开关，而 Omega 网络需要 12 个开关盒，相当于 48 个开关。但是，交叉开关网络可以支持处理器消息传递的任意组合，而 Omega 网络却不能

### 网络拓扑实现

本节对所有网络简单分析的时候，忽略了在网络构建时需要考虑的实际因素。在高速时钟下，链路的距离影响通信的成本——一般来说，距离越长，在高速时钟下的成本越大。较短的距离也会更加容易地将更多的连线增加到同一链路中，因为连线越短，驱动连线的能耗就会越低。较短的连线也比较长的连线便宜。另外一个实际限制是三维拓扑连线必须映射到芯片的二维媒介上。最后一点需要考虑的是能耗。例如，能耗可能迫使多核芯片必须采用简单网格拓扑。总之，在黑板上画上很美的拓扑，在使用硅工艺或数据中心制造时可能是不切实际的。

现在我们已经了解了集群的重要性，并且看到了将它们连接起来的方法，接下来我们要看一看网络与处理器的软硬件接口。

#### 小测验

538 是非判断：对于一个有 $P$ 个节点的环，总网络带宽与切分带宽的比为 $P/2$。

## 6.9 与外界通信：集群网络

这个网上章节讲述了用来连接集群节点的网络硬件和软件。例子中使用了采用 PCIe 连接到计算机上的 10Gb/s 的以太网。这个例子展示了软硬件优化如何提升互联网络的性能，包括零拷贝消息传递、用户空间通信、使用轮询机制代替 I/O 中断以及使用硬件计算校验总和。尽
539 管例子是互联网络，但是本节介绍的这些技术也可以应用到存储控制器和其他的 I/O 设备。

在这个网上章节中从底层详细讲述了互联网络的性能后，下节从更高的层次介绍了如何测试评价各种类型的多处理器。

## 6.10 多处理器测试集程序和性能模型

在第 1 章中我们看到，基准测试系统一直是一个敏感话题，因为它是判断哪个系统更好的一种最为直观的方式。测试结果不仅影响商业系统的销售，而且影响这些系统设计者的声誉。因此，每个参加测试者都希望自己获胜，但是如果别人获胜，他们也希望确信获胜者的系统真正是一个更好的系统。这些期望导致测试结果不能只是针对测试程序的简单技俩，而应该能够真正促进实际应用程序性能的提高。

为了避免可能的作弊，一个典型的原则是你不能修改基准测试程序。源代码和数据集是固定的，并且只有唯一的正确结果。对这些原则的任何违反都会使得测试结果无效。

许多多处理器基准测试程序都遵守这些惯例。一个共同的例外是允许增加问题规模，这样你就可以在有不同数量处理器的系统上运行。也就是说，许多基准测试程序允许弱比例缩放而不是强比例缩放，但即使如此，在比较不同问题规模的程序结果时仍要小心。

图 6-16 是对几种并行基准测试程序的总结。描述如下：

- Linpack 是一组线性代数例程，这些例程执行高斯消元。前面示例中给出的 DGEMM 例程就是 Linpack 基准测试程序中的一小部分代码片段，但是它占用了该基准测试程序的大部分执行时间。它允许弱比例缩放，让用户选择任何规模的问题。而且，它允许使用者以几乎任何形式和任何语言重写 Linpack，只要保持计算结果的正确性以及对于同样规模大小的问题进行相同次数的浮点运算。每隔两年计算 Linpack 最快的 500 台计算机会公布在 www.top500.org 上。排名第一的被新闻界认为是世界上最快的计算机。
- SPECrate 是一个基于 SPEC CPU 基准测试程序（如 SPEC CPU 2006，见第 1 章）的吞吐量度量。SPECrate 不是报告单个程序的性能，而是同时运行该程序的很多副本。因此，
540 它主要测量任务级并行，因为这些任务之间没有通信。程序的副本数量是不受限制的，因此这也是弱比例缩放的形式。

- SPLASH 和 SPLASH 2（Stanford Parallel Applications for Shared Memory）是 20 世纪 90 年代斯坦福大学的研究成果，目的是提供类似于 SPEC CPU 一样的并行基准测试程序。它由核心程序和应用程序构成，许多都来自高性能计算领域。尽管该程序提供了两组数据集，但仍需要强比例缩放。
- NAS（NASA Advanced Supercomputing）并行基准测试程序是 20 世纪 90 年代以来对多处理器基准测试程序的另一尝试。它由 5 个核心程序构成，都是来源于流体动力学。它允许通过定义几个数据集实现弱比例缩放，像 Linkpack，这些基准测试程序可以被重写，但是编程语言只能使用 C 或 Fortran。
- 最近的 PARSEC（Princeton Application Repository for Shared Memory Computer）基准测试程序集由采用 Pthread（POSIX 线程）和 OpenMP（Open MultiProcessing，见 6.5 节）的多线程程序组成。它们主要专注于计算领域，由 9 个应用程序和 3 个核构成。其中 8 个依赖数据并行，3 个依赖流水并行，另外一个依赖非结构化并行。
- 在云的前端，Yahoo! Cloud Serving Benchmark（YCSB）的目标是比较云数据服务的性能。它通过使用 Cassandra 和 HBase 作为具有代表性的例子，提供了一个易于让用户评测新数据服务的框架。[Cooper，2010]

Pthread：创建和操作线程的一个 UNIX API。它被组织成一个库的形式。

| 基准测试程序 | 可扩展性 | 可编程性 | 描述 |
| --- | --- | --- | --- |
| Linpack | 弱 | 是 | 稠密矩阵线性代数 [Dongarra，1979] |
| SPECrate | 弱 | 否 | 独立任务并行化 [Henning，2007] |
| 面向共享存储器的斯坦福并行应用 SPLASH2 [Woo et al.，1995] | 强（虽然提供了两种问题规模） | 否 | 复杂 1D FFT<br>模块化 LU 分解<br>模块化稀疏厄拉斯基分解<br>整数基数排序<br>巴尔内斯小屋<br>适应性快速多极算法<br>海洋仿真<br>光线跟踪<br>声音渲染器<br>空间数据结构的水仿真<br>非空间数据结构的水仿真 |
| NAS 并行 Benchmark [Bailey et al.，1995] | 弱 | 是（只能是 C 或 Fortran） | EP：非空间数据结构的水仿真<br>MG：简化的多重网格计算<br>CG：面向共轭梯度方法的非结构化网格<br>FT：使用 FFT 的 3-D 偏微分方程<br>IS：大型整数排序 |
| PARSEC Benchmark 集 [Bienia et al. 2008] | 弱 | 否 | Blackscholes——使用毕苏期权定价模式的期权定价<br>Bodytrack——人体跟踪<br>Canneal——使用 cache 感知的模拟退火进行布线优化<br>Dedup——采用数据去重的下一代压缩<br>Facesim——人脸运动仿真<br>Ferret——内容相似性搜索服务器<br>Fluidanimate——SPH 方法的流体动力学动画<br>Freqmine——常见物品集合的数据挖掘<br>Streamcluster——输入流的在线分类<br>Swaptions——Pricing of a portfolio of swaptions<br>Vips——图像处理<br>x264——H.264 视频编码 |

图 6-16 并行基准测试程序的实例

| 基准测试程序 | 可扩展性 | 要求浮点 | 描述 |
| --- | --- | --- | --- |
| 伯克利设计数据集 [Asanovic et al. 2006] | 强或弱 | 是 | 有限状态机<br>组合逻辑<br>图的遍历<br>结构化网格<br>稠密矩阵<br>稀疏矩阵<br>波前法<br>动态程序设计<br>N体问题<br>MapReduce：（云计算领域的一个专有名词）<br>回溯/分支与边界<br>图模型推导<br>非结构化网格 |

图 6-16 （续）

基准测试程序原有约束所造成的负面影响是创新被局限到体系结构和编译器中。更好的数据结构、算法、编程语言等通常不能使用，因为这些可能导致容易误解的结果。这样系统可能不是由于硬件或编译器的原因获得更高性能，例如算法。

尽管这些准则在计算基础相对稳定时是可以理解的——因为它们是在 20 世纪 90 年代提出的，而且是在 90 年代的前 5 年——但是，这些准则在编程变革中就不合时宜了。为了变革的成功，我们需要鼓励在所有层次上的创新。

加利福尼亚大学伯克利分校的研究人员提出了一个最新的方法。他们确定了 13 个面向未来应用程序的设计模式。这些设计模式使用框架或核心实现。一些实例包括稀疏矩阵、结构化网格、有限状态机、映射规约和图遍历等。通过将定义保持在高级别层次，他们希望鼓励在系统的任何层次进行创新。因此，速度最快的稀疏矩阵求解器的系统除了使用新型体系结构和编译器之外，还可以使用任何数据结构、算法和编程语言。

## 6.10.1 性能模型

和测试集程序相关的一个话题就是性能模型。就像我们在本章中看到的不断增加的体系结构多样性——多线程、SIMD、GPU——如果我们能拥有一个简单的模型来分析不同体系结构设计的性能，将是十分有益的。这个模型不需要是完美的，只要有所见地就行。

541 ~ 542

第 5 章的对于 cache 性能测试的 3C 模型是一个性能模型的例子，它并不是一个完美的性能模型，因为它忽略了一些潜在的重要因素，如块尺寸大小、块分配策略和块替换策略。而且，它还含有一些含糊其辞的地方。例如，在一个设计中缓存缺失的原因可能是因为容量，但在另一个相同大小的缓存中可能是因为冲突。然而 3C 模型已经流行了 25 年，因为它提供了深刻理解程序行为的一个途径，有助于体系结构设计者和程序员基于模型的洞察来改进他们的创新。

为了找到这样一个并行计算机的模型，让我们从小的核心程序开始，就像图 6-16 中的 13 个 Berkeley 设计模式。尽管这些核心程序的不同数据类型有许多版本，但是浮点在几种实现中是最常见的。因此，在给定的计算机上峰值浮点性能是这类核心程序的速度瓶颈。对于多核芯片，峰值浮点性能是芯片上所有处理器核峰值性能的总和。如果系统中包含多处理器，那么应当将每芯片的峰值性能与芯片数量相乘。

对存储器系统的需求可以用峰值浮点性能除以每访问一字节所包含浮点操作数的平均值来估算：

浮点操作数 / 秒 ÷ 浮点操作数 / 字节 = 字节 / 秒

存储器每访问一字节所包含的浮点操作比例称作**算术密度**（arithmetic intensity）。它的计算可以用程序中总的浮点操作数除以程序执行期间主存传输数据总的字节数。图 6-17 给出了图 6-16 中几种 Berkeley 设计模式的算术密度。

⇨ 算术密度：一个程序中浮点操作数量与访问主存字节数量的比值。

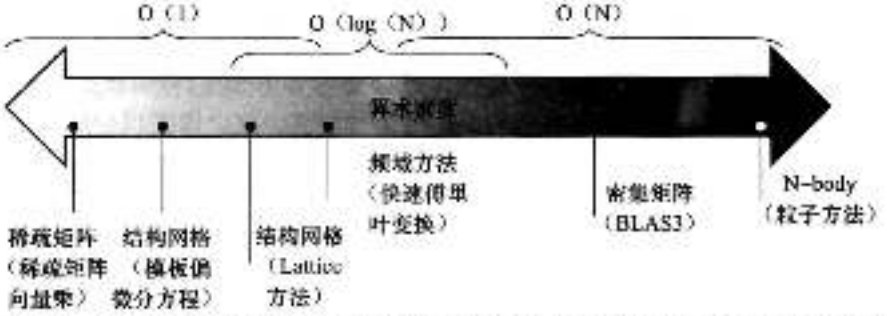



图 6-17 算术密度，计算方式为用运行程序中总的浮点操作数除以访问主存总的字节数［Williams，Patterson，2009］。一些核心程序的算术密度与问题规模成比例扩展，如 Dense Matrix，但是也有许多核心程序与问题规模无关。对于前者，弱比例缩放会导致不同的结果，因为它对存储系统的需求不是很大

543

## 6.10.2 Roofline 模型

本节提出的简单模型将浮点性能、算术密度和存储性能联系在一个二维图中［Williams，Waterman，Patterson，2009］。峰值浮点性能可以在上面谈到的硬件规格说明书中找到。我们这里考虑的核心程序的工作集不适合使用片上缓存，因此峰值存储器性能可以使用缓存之后的存储器来定义。获得峰值存储性能的一种方法是使用 Stream 基准测试程序。（见第 5 章的精解。）

图 6-18 给出了针对一台计算机的模型，注意不是针对每个核心程序的模型。纵轴 Y 表示浮点性能，从 0.5 到 64.0 GFLOPs/s。横轴 X 表示算术密度，从 1/8 FLOPs/DRAM 字节到 16 FLOPs/DRAM 字节。注意该图采用 log-log 的比例。

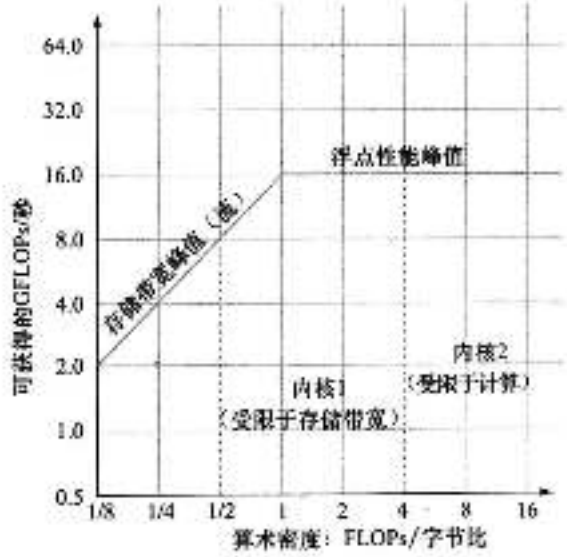



图 6-18 Roofline 模型［Williams，Waterman，Patterson，2009］。本例具有 16GFLOPS/s 的峰值浮点性能和 16GB/s 的峰值存储带宽，该数据来自流测试程序（由于流实际上是 4 次测量，图中的线是 4 次的均值）。左边的灰色点垂线标识内核 1，其计算密度为 0.5FLOPs/byte。在 Opteron X2 上，受限于低于 8GFLOPS/s 的存储器带宽。右边的点垂线标识内核 2，计算密度为 4FLOPs/byte，它只受限于 16GFLOPS/s 的计算。（该数据基于 AMD Opteron X2（版本 F），使用运行在双socket系统中的 2GHz 的双核）

对给定的核心程序，我们可以基于其算术密度在X轴找到对应点。如果我们在该点画一条垂直线，那么该核心程序在计算机上的性能一定在该垂直线的某个位置上。我们可以画一条水平线表示该计算机的峰值浮点性能。显然，实际的浮点性能不会超过该水平线，因为这是一个
544 硬界限（hardware limit）。

我们如何画出峰值存储性能呢（单位为字节/秒）？由于X轴是FLOPs/byte，Y轴是FLOPs/s，所以byte/s只是图中一条45°的对角线。因此，我们画出第二条线来表示对于给定的算术密度该计算机存储系统所能支持的最大浮点性能。我们可以用下面的公式表示该界限，以便在图6-18中画出该线：

可达到的GFLOPs/s = Min(峰值存储器带宽 × 算术密度，峰值浮点性能)

水平线和对角线给出了简单模型的名称并标出了对应值。这个像屋顶一样的轮廓线设定了一个核心程序在不同算术密度下的性能上界。给定一个计算机的Roofline模型，你可以重复地使用它，因为它不会随核心程序而变化。

如果我们认为算术密度是支撑屋顶的一个杆，那么它要么支撑屋顶的平坦部分，这表示性能受计算限制；要么支撑屋顶的倾斜部分，这表示性能受存储器带宽限制。在图6-18中，核心程序1属于前者，而核心程序2属于后者。

需要注意的是“脊点”，它是屋顶平坦部分与倾斜部分的交叉点，这对计算机来说是一个关键点。如果它过于靠右，那么只有极高算术密度的核心程序才能获得最大性能。如果它过于靠左，那么几乎所有核心程序都可以达到最大性能。

## 6.10.3 两代Opteron的比较

四核的AMD Opteron X4（Barcelona）是两核Opteron X2的后续版本。为了简化主板设计，它们使用了相同的插座。因此，它们具有相同的DRAM通道，也就具有相同的峰值存储带宽。除了将核心程序数量加倍之外，Opteron X4还将每核的峰值浮点性能提高到原来的两倍：Opteron X4核每时钟周期可发射两条浮点SSE2指令，而Opteron X2核最多只能发射一条。由于我们比较的两个系统具有接近的时钟频率——Opteron X2为2.2GHz，Opteron X4为2.3GHz——所以Opteron X4的峰值浮点性能是Opteron X2的4倍，而两者DRAM带宽完全相同。Opteron X4还有2MiB的三级缓存，而Opteron X2没有。

图6-19比较了两个系统的Roofline模型。正如我们所期望的那样，脊点向右进行了移动，从Opteron X2的1移到了Opteron X4的5。因此，为了看到下一代Opteron处理器性能的改进，核心程序的算术密度必须大于1，或者核心程序的工作集必须适合Opteron X4的缓存。

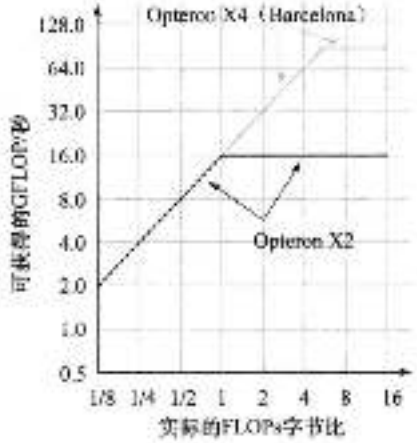



图6-19 两代Opteron的Roofline模型。Opteron X2的屋顶线与图6-18相同，使用黑色绘制，而Opteron X4的屋顶线使用灰色绘制。Opteron X4更大的脊点意味着原来在Opteron X2中是计算受限的核心程序在Opteron X4中可能是存储性能受限

Roofline 模型给出了性能的上界。假设你的程序远远低于该上界，那么你应进行哪些优化呢？这些优化的优先级顺序是什么？

为了克服计算瓶颈，下面的两种优化可以改进几乎任何核心程序：

1）浮点操作混合。对一台计算机而言，峰值浮点性能一般需要几乎同时到达的等量加法和乘法。这种均衡不仅是因为计算机支持融合的乘加指令（见第 3 章的精解），也因为浮点单元具有相同数量的浮点加法器和浮点乘法器。最佳性能也需要大部分浮点操作和非整数指令混合。

2）提高指令级并行并应用 SIMD。对当代的体系结构，最高性能在每个时钟周期取指、执行并提交 3～4 条指令时才能获得（见 4.10 节）。这一步的目标是从编译器上改进代码来增加 ILP。一种方法是循环展开，就像我们在 4.12 节看到的。对 x86 体系结构而言，一个单一的 AVX 指令可以对 4 个双精度操作数进行操作，因此它们应该被尽量使用（见 3.7 节和 3.8 节）。

为了克服存储瓶颈，可以采用下面的两种优化方法：

1）软件预取指（software prefetching）。最高性能通常需要保持许多存储器操作一直运行，这使得通过执行软件预取指令来预测访存更加容易，而不用等到计算需要该数据时才进行访存。

2）内存关联（memory affinity）。现在大多数的微处理器都在片内包含了内存控制器，它能提高存储器层次的性能。如果系统中含有多个芯片，这就会使一些地址访问本地 DRAM，而其他地址需要通过芯片互连才能访问对于其他芯片是本地的 DRAM。这种分割导致了我们在 6.5 节介绍的非一致性访存。通过另一个芯片进行访存会降低性能。第二种优化方法是分配数据后尽量让线程操作属于同一存储器－处理器对上的数据，这样处理器几乎不会访问其他芯片上的存储器。

Roofline 模型可以帮助决定选用哪些优化，以及优化的实施顺序。我们可以认为这些优化方法中的每一个都是适当屋顶线下面的一层“天花板”，也就是说，在没有实施相应优化的情况下不能突破该层天花板。

计算性能屋顶线可以在手册中找到，而存储屋顶线则可以通过运行流基准测试程序获得。计算性能天花板，如浮点均衡，也可来自该计算机的手册。存储天花板，例如存储器关联，需要在每台计算机上运行实验，从而决定它们之间的间隙。一个好消息是这一过程在每台计算机上只需进行一次，只要有人完成了对该计算机天花板的评估，任何人都可以将该结果用于指导该计算机优化的先后次序。

图 6-20 相对于图 6-18 中的屋顶线模型增加了天花板，其中上图给出了计算天花板，下图给出了存储带宽天花板。尽管较高的天花板没有标记，但是其隐含使用了全部优化手段；为了突破最高的天花板，首先必须突破所有下面的天花板。

天花板之间间隙的宽度和下一个更高的限制表示优化之后的收益。因此，图 6-20 建议优化 2 和 4。其中 2 是改善 ILP，对于改善该计算机的计算有很大益处；4 是改善内存关联，对于改善该计算机的存储带宽有很大益处。

图 6-21 将图 6-20 中的天花板整合到一张图中。核心程序的算术密度决定了优化的区域，优化区域反过来又给出了哪些优化手段可以尝试。需要注意的是，对大多数算术密度，计算优化和存储带宽优化都是重叠的。图 6-21 中有三处不同的阴影标记，用于区分不同的优化策略。例如，核心程序 2 落在右边灰色梯形区域，表示只工作在计算优化上。核心程序 1 落在灰色与浅灰色平行四边形区域，表示两种优化均可尝试。而且，它建议从优化 2 和优化 4 开始。注意到核心程序 1 的垂直线低于浮点失衡优化，因此优化 1 是没有必要的。如果核心程序落在左下角的浅灰色三角形区域，则表示只需进行存储优化即可。

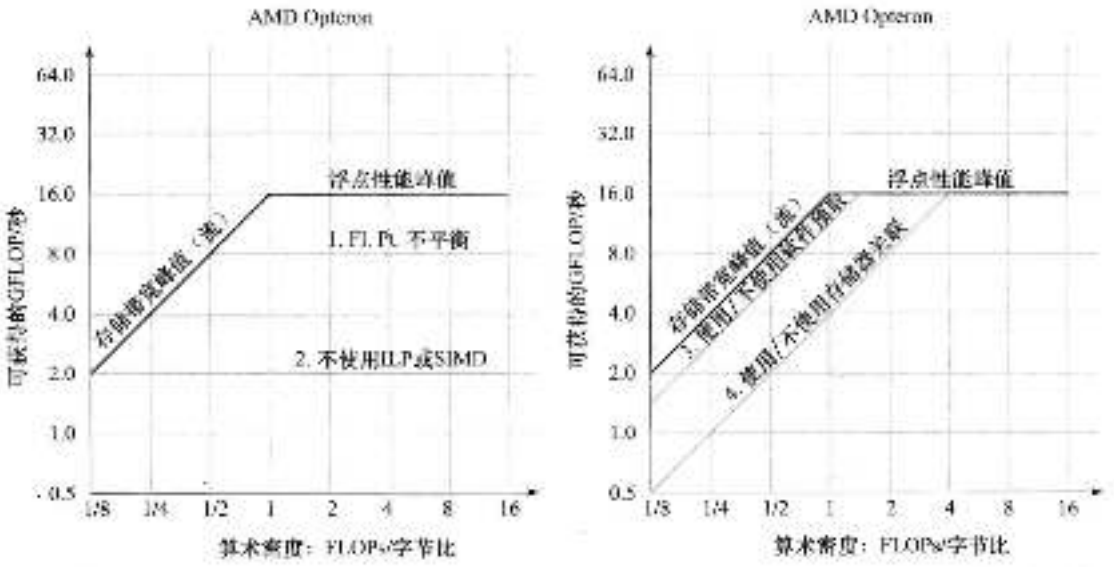



图6-20 带有天花板的Roofline模型。其中上面的图表示计算性能的"天花板"，1表示浮点操作混合失衡情况下性能为8GFLOPs/s，2表示同时未使用ILP和SIMD下的性能为2GFLOPs/s。下面的图表示存储带宽的天花板，其中3表示没有软件预取指时的带宽为11GB/s，4表示同时没有优化内存关联的带宽为4.8GB/s。

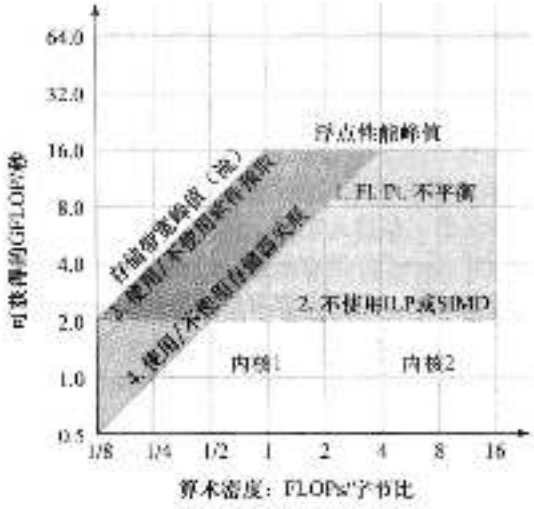



图6-21 将图6-18中两幅重叠的Roofline模型。算术密度处于右边灰色梯形区域的核心程序应当着重于计算优化，而处于浅灰色三角形区域的核心程序应当着重于存储带宽优化。处于灰色和浅灰色平行四边形区域的核心程序两种优化都应当考虑。例如核心程序1落在中间的平行四边形中，可尝试优化ILP和SIMD、内存关联、软件预取指等。核心程序2落在右边的梯形区域，可尝试优化ILP和SIMD以及浮点操作均衡

到目前为止，我们一直假定算术密度是固定的，但是实际情况并非如此。首先，有些核心程序的算术密度会随问题规模增长，如稠密矩阵和多体问题（见图6-17）。事实上，这就是程序员处理弱比例缩放比强比例缩放更成功的原因之一。第二，存储器层次结构的效应影响存储器的访问次数，因此改善缓存性能的优化也能改善算术密度。一个例子是通过循环展开，并将使用相近地址的语句分组到一起来改善时间局部性。许多计算机提供特殊的缓存指令，故可以先将数据分配到缓存中，而不用先从存储器中填充，因为它可能很快被改写。这些优化降低了存储器流量，因此可以将算术密度乘以一个系数（如1.5）向右移动。这种右移会使核心程序移到一个不同的优化区域。

虽然上面的例子展示的是如何帮助程序员提高程序的性能，但同时架构师也可以利用这个

模型决定硬件的哪些部分应该优化，以提升他们认为重要的核心程序的性能。

下一节使用 Roofline 模型分析一个多核微处理器和一个 GPU 的性能差异，以及这些差异是否反映了真实程序的性能。

548 - 549

**精解** 天花板是分层次的，最低的天花板是最容易优化的。显然，程序员可以按任意顺序优化，但是遵从建议的顺序可以避免将时间浪费在因其他约束而无效的优化上。和 3C 模型类似，只要模型进行了抽象，就会存在一些理想的假设。例如，屋顶线模型是假定程序在所有处理器间负载均衡的。

**精解** 一种替换 Stream 基准测试程序的方法是使用原始 DRAM 带宽作为屋顶线。尽管原始带宽构成了硬件上界，但是存储器的实际性能往往与此相差甚远，因此可用性不高。也就是说，没有程序能够接近该上界。使用 Stream 的不利因素是非常仔细的编程有可能获得高于 Stream 的结果，因此存储器屋顶线不像计算屋顶线那样坚实。我们坚持使用 Stream 是因为很少有程序员能够做到这一点。

**精解** 尽管屋顶线模型是针对多核处理器的，但是它也可以用于单处理器。

**小测验**

是非判断：评测并行计算的常规方法的主要缺陷是确保公平性的同时压制了创新。

## 6.11 实例：评测 Intel Core i7 960 和 NVIDIA Tesla GPU 的 Roofline 模型

一组 Intel 的研究人员发表了一篇论文［Lee 等，2010］，对带有多媒体 SIMD 扩展的四核 Intel Core i7 960 与前一代的 GPU（NVIDIA Tesla GTX 280）进行了对比。图 6-22 列出了两个系统的特点。这两个系统都是在 2009 年秋天购买的。Core i7 使用的是 Intel 的 45 纳米半导体工艺，而 GPU 使用的是 TSMC 的 65 纳米工艺。虽然让一个中立机构或一个对两种产品都感兴趣的机构进行评估可能更公平一些，但是本章的重点不是为了说明哪个产品运行地更快，快的比慢的快多少等问题，而是尝试着理解这两种截然不同的结构的特性。

| | Core i7-960 | GTX 280 | GTX 480 | 280/i7 的比例 | 480/i7 的比例 |
|---|---|---|---|---|---|
| 处理单元的数量（核或线程数量） | 4 | 30 | 15 | 7.5 | 3.8 |
| 时钟频率（GHz） | 3.2 | 1.3 | 1.4 | 0.41 | 0.44 |
| 芯核尺寸 | 263 | 576 | 520 | 2.2 | 2.0 |
| 工艺 | Intel 45nm | TSMC 65nm | TSMC 40nm | 1.6 | 1.0 |
| 功率（芯片，非模块） | 130 | 130 | 167 | 1.0 | 1.3 |
| 晶体管数量 | 700M | 1 400M | 3 030M | 2.0 | 4.4 |
| 存储带宽（G字节/秒） | 32 | 141 | 177 | 4.4 | 5.5 |
| 单精度SIMD宽度 | 4 | 8 | 32 | 2.0 | 8.0 |
| 双精度SIMD宽度 | 2 | 1 | 16 | 0.5 | 8.0 |
| 峰值单精度向量FLOPS（GFLOP/秒） | 26 | 117 | 63 | 4.6 | 2.5 |
| 峰值单精度SIMD FLOPS（GFLOP/秒） | 102 | 311或933 | 515或1344 | 3.0-9.1 | 6.6-13.1 |
| （SP 1加或乘） | 不支持 | (311) | (515) | (3.0) | (6.6) |
| （SP 1融合乘加的指令） | 不支持 | (622) | (1 344) | (6.1) | (13.1) |
| （Rare SP双通道融合乘加和乘） | 不支持 | (933) | 不支持 | (9.1) | – |
| Peal双精度SIMD FLOPS（GFLOP/秒） | 51 | 78 | 515 | 1.5 | 10.1 |

图 6-22 Intel Core i7-960、NVIDIA GTX 280 以及 GTX 480 的指标。最右面的两列展示了 Tesla GTX 280 和 Fermi GTX 480 与 Core i7 的对比。尽管例子研究的是 Tesla 280 和 i7，但是我们也给出了 Fermi 480 与 Tesla 280 的对比，因为本章中介绍到了这点。注意这里的存储带宽比图 6-23 中的要高，因为这里的是 DRAM 引脚的带宽，而图 6-23 中的是通过测试程序得到的处理器中的带宽（本表来自于 Lee 等．［2010］中的表 2）

图 6-23 中给出的 Core i7 960 和 GTX 280 的 Roofline 模型展示了这两个计算机的区别。GTX 280 不仅有着更高的存储带宽和双精度浮点性能，而且它的双精度拐点更靠左。GTX 280 的双
550 精度拐点在 0.6，而 Core i7 的在 3.1。上面曾提到，若 Roofline 模型的拐点更靠左，则更容易达到它的性能峰值。对于单精度性能，这两个计算的观点都更靠右一些，所以要想达到最佳的单精度计算性能会更难。注意，算术运算强度是基于访问主存的字节数的，而不是基于访问 cache 的字节数。因此，就像之前提到的，若大部分访存都访问 cache，则缓存可以改变一个核心程序在计算机上的运算强度。再次注意，对于这两种结构，带宽是单位步长的访问。我们即将看到，真实的聚集-分散地址在 GTX 280 和 Core i7 上会更慢一些。

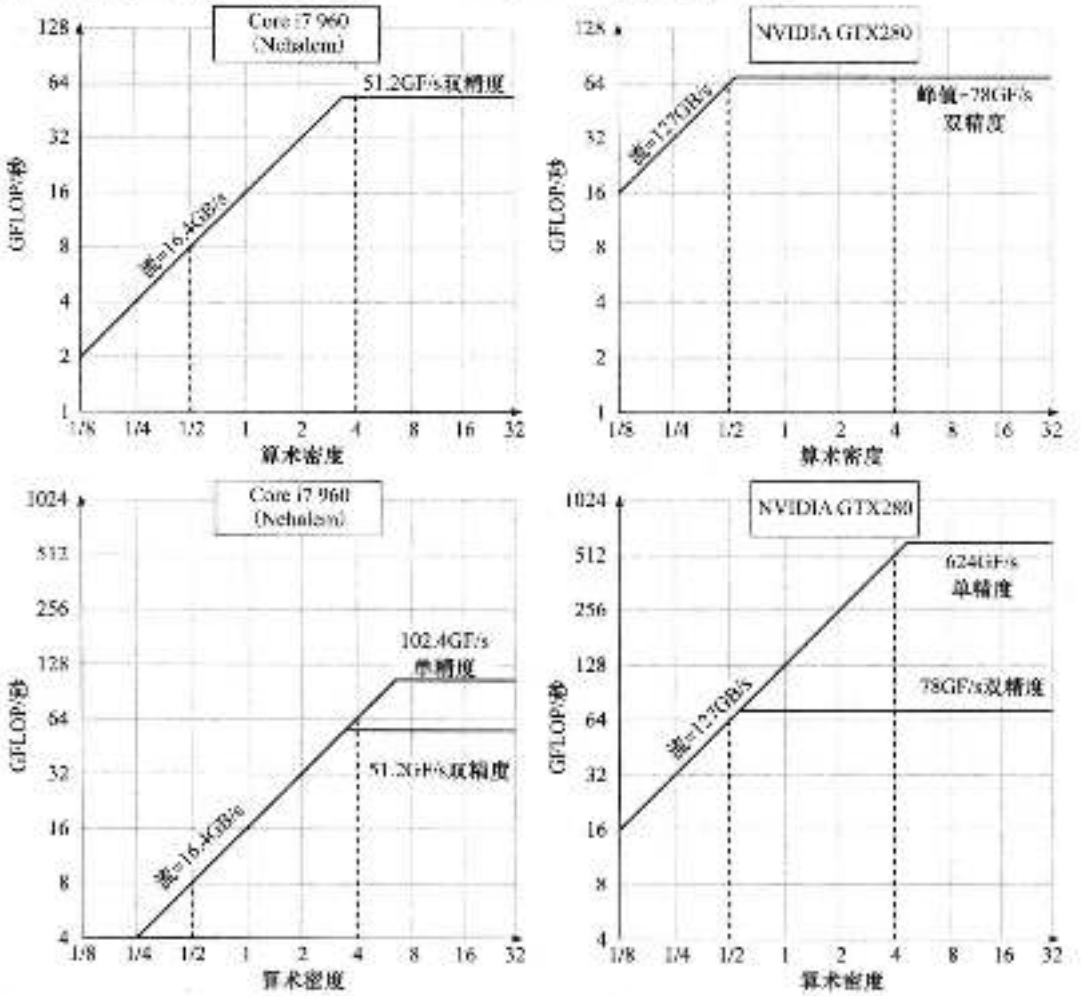



图 6-23 Roofline 模型［Williams，Waterman，Patterson，2009］。上面一行给出了双精度浮点性能，下面一行给出了单精度浮点性能。（双精度浮点性能的屋顶线同时也在下面一行的图中画出了，已给出对比。）左边的 Core i7 960 的最高双精度浮点性能为 51.2GFLOP/s，最高单精度浮点性能为 102.4GFLOP/s，最高存储带宽为 16.4GB/s。NVIDIA GTX 280 的最高双精度浮点性能为 78GFLOP/s，最高单精度浮点性能为 624GFLOP/s，最高存储带宽为 127GB/s。图中左侧的垂直虚线表示 0.5FLOP/byte 的算术运算强度。Core i7 的算术运算强度受存储带宽的限制不能超过 8GFLOP/s（对于双精度浮点和单精度浮点都如此）。图中右侧的垂直虚线表示 4FLOP/byte 的运算强度。在 Core i7 上限制在 51.2GFLOP/s（双精度浮点性能）到 102.4GFLOP/s（单精度浮点性能）之间，在 GTX 280 上限制在 78GFLOP/s（双精度浮点性能）到 512GFLOP/s（双精度浮点性能）之间。为了在 Core i7 上达到最高的运算率，需要使用全部 4 个核和 SSE 指令，并且乘法和加法的数量相等。对于 GTX 280，你需要在所有多线程的 SIMD 处理器上使用混合的乘加指令

551 ~ 553

通过分析，研究者们最近提出了4个测试集程序的计算和访存特性，然后对可以捕获这些特性的各种计算核心程序集建模来选择测试用的程序。图6-24给出了性能结果，数字越大表明速度越快。Roofline模型帮助解释了本例子中的相对性能。

| 内核 | 单位 | Core i7-960 | GTX 280 | GTX 280/i7-960 |
|---|---|---|---|---|
| SGEMM | GFLOP/秒 | 94 | 364 | 3.9 |
| MC | 十亿条路径/秒 | 0.8 | 1.4 | 1.8 |
| Conv | 百万像素/秒 | 1 250 | 3 500 | 2.8 |
| FFT | GFLOP/秒 | 71.4 | 213 | 3.0 |
| SAXPY | G字节/秒 | 16.8 | 88.8 | 5.3 |
| LBM | 百万次查找/秒 | 85 | 426 | 5.0 |
| Solv | 帧/秒 | 103 | 52 | 0.5 |
| SpMV | GFLOP/秒 | 4.9 | 9.1 | 1.9 |
| GJK | 帧/秒 | 67 | 1 020 | 15.2 |
| Sort | 百万个元素/秒 | 250 | 198 | 0.8 |
| RC | 帧/秒 | 5 | 8.1 | 1.6 |
| Search | 百万次查询/秒 | 50 | 90 | 1.8 |
| Hist | 百万像素/秒 | 1 517 | 2 583 | 1.7 |
| Bilat | 百万像素/秒 | 83 | 475 | 5.7 |

图6-24 对两个平台测量得到的原始和相对性能数据。在这项研究中，SAXPY只被用来对存储带宽进行测量，所以右边的单位为GBytes/s而不是GFLOP/s（基于[Lee等，2010]中的表3。）

GTX 280的标准性能在慢2.5倍（时钟频率）到快7.5倍（每个芯片上的核数）之间，而它的实际性能为慢2.0倍到快15.2倍之间，Intel的研究者们决定找到其中的原因：

- *存储带宽*：GPU有4.4倍的存储带宽，这解释了为什么LBM和SAXPY运行地快5.0倍和5.3倍；它们的工作集为几百兆字节，因此数据不能在Core i7的cache中全放下。（所以由于集中地访存，它们刻意不使用第5章中的缓存阻塞技术）。因此，Roofline模型的斜率解释了它们的性能。SpMV同样也有一个大的工作集，但是它的运行只快了1.9倍，这是因为GTX 280的双精度浮点运算只比Core i7的快1.5倍。
- *计算带宽*：剩下的核心程序中有5个是受计算制约的：SGEMM、Conv、FFT、MC和Bilat。GTX运行这5个程序时分别快3.9、2.8、3.0、1.8和5.7倍。这其中的前三个使用单精度浮点运算，而且GTX 280的单精度运算要快3－6倍。MC使用双精度浮点运算，这解释了为什么它只快1.8倍，因为双精度浮点运算性能只比Core i7快1.5倍。Bilat使用GTX 280直接支持的超越函数。Core i7执行Bilat时，三分之二的时间用来计算超越函数，所以GTX 280要快5.7倍。这些观察帮助指明了使用硬件支持负载的特定操作（双精度浮点运算、超越函数）有何意义。
- *从cache获得的好处*：在GTX上RC运行地只快1.6倍，只是因为在GPU上Core i7的缓存阻塞技术可以防止程序变得受存储带宽的影响（参见5.4节和5.14节），而在GTX上会出现这种情况。缓存阻塞技术还可以帮助Search程序。如果索引树小到可以在cache中装下，则Core i7的速度会快2倍。大的索引树会使程序受存储带宽的限制。总体来说，GTX在运行Search程序时快1.8倍。缓存阻塞技术也对Sort程序有利。尽管大部分程序员不会在一个SIMD处理器上运行Sort，但是它可以用称作split的1位Sort原语编写。但是，split算法比标量Sort程序多执行更多的指令。结果是，Core i7比GTX 280快1.25倍。注意，cache对于运行在Core i7上的其他核心程序也有帮助，因为缓存阻塞技术允许SGEMM、FFT和SpMV变为受计算制约的程序。这项观察再次强调了第5章介绍的缓存阻塞技术的重要性。
- *聚集－分散*：如果数据被分散到主存的各个地方，多媒体SIMD扩展就不会有太大的

用途；只有访问的数据是16字节对齐时才会获得最佳的性能。因此GJK程序在Core i7上从SIMD中获得的好处很少。就像前面提到过的，GPU提供向量结构支持而大部分SIMD扩展不支持的聚集－分散技术。内存控制器甚至会将访问同一个DRAM页的请求打包（见5.2节）。这两点使得GTX 280在运行GJK时比Core i7快15.2倍，这比图6-22中的任何物理参数都要大。这项观察再一次证实了聚集－分散技术对于缺乏SIMD扩展的向量和GPU结构的重要性。

- 同步：同步的性能受原子更新的制约，这占据了Core i7 28%的总执行时间，尽管Core i7有硬件读取并自增（fetch-and-increment）指令。因此，Hist程序在GTX 280上只比Core i7快1.7倍。Solv程序使用少量指令和一个同步操作来解决一批独立的约束条件。Core i7可以从原子指令以及一致性存储模型之中获益，一致性存储模型可以保证结果是正确的，尽管不是所有之前的访存指令都已经执行完毕。由于没有一致性存储模型，当GTX 280从系统处理器那里得到一批操作时，它的性能只是Core i7的0.5倍。这项观察指出了同步的性能对于一些数据并行问题是多么重要。

由Intel研究者选择的这些核心程序揭示出的Tesla GTX 280的不足，在Tesla的后续版本中都得到了解决；Fermi拥有更快的双精度浮点性能、更快的原子操作以及cache。另一个有趣的是，支持向量结构的聚集－分散机制早于SIMD指令几十年就出现了，并且对SIMD扩展的有效执行非常重要，这在这次比较之前就已经预言到了。Intel的研究者们注意到这14个核心程序中的6个可以在对Core i7提供足够的聚集－分散支持的前提下更有效地发掘SIMD。这项研究当然也证实了缓存阻塞技术的重要性。

554

现在我们看到了评价不同多处理器得出的很多结果，接下来让我们看看我们需要对DGEMM程序的C代码进行多大的修改才可以发挥多处理器的性能优势。

## 6.12 运行更快：多处理器和矩阵乘法

这一节是我们根据Intel Core i7（Sandy Bridge）已经改变结构的DGEMM程序以获得性能提升的最后一步，也是最大的一步。每个Core i7有8个核，我们用的计算机有2个Core i7。所以我们有16个核来运行DGEMM程序。

图6-25给出了使用这些核的OpenMP版本的DGEMM程序。注意，第30行是相对于图5-48唯一增加的一行代码，以使程序可以运行在多处理器上：使用了一个OpenMP的pragma语句告诉编译器对最外层for循环使用多线程。它告诉计算机将最外层for循环的任务分配给所有线程去执行。

图6-26画出了一个经典的多处理器加速比图，它展示了当线程数量增加时，相对于单线程的性能提升。这个图让人们很容易地看到强比例缩放相对于弱比例缩放的挑战。当所有数据都可以放入一级数据cache中时，例如32×32矩阵，增加线程的数量实际上会损坏性能。在这种情况下，16个线程的DGEMM程序的性能只是单线程的一半。相反，最大的那两个矩阵在使用16个线程时，性能提升了14倍，所以得到了图6-26最上面的两条线。

图6-27给出了当我们将线程的数量从1个增加到16个时的绝对性能增长。对于960×960的矩阵，DGEMM程序以174GFLOPS的速率执行。图3-21给出的未经任何优化的C版本的DGEMM程序是以0.8GFOPS的速率执行，所以通过第3～6章根据硬件对代码进行的优化，性能提升了200倍！

接下来我们给出了多进程的谬误与陷阱。计算机系统结构的菜地中充满了忽略这些谬误与陷阱的并行项目。

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3 #define BLOCKSIZE 32
4 void do_block (int n, int si, int sj, int sk,
5                double *A, double *B, double *C)
6 {
7   for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
8     for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
9       __m256d c[4];
10      for ( int x = 0; x < UNROLL; x++ )
11        c[x] = _mm256_load_pd(C+i+x*4+j*n);
12      /* c[x] = C[i][j] */
13      for( int k = sk; k < sk+BLOCKSIZE; k++ )
14      {
15        __m256d b = _mm256_broadcast_sd(B+k+j*n);
16      /* b = B[k][j] */
17        for (int x = 0; x < UNROLL; x++)
18          c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                 _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20      }
21
22      for ( int x = 0; x < UNROLL; x++ )
23        _mm256_store_pd(C+i+x*4+j*n, c[x]);
24        /* C[i][j] = c[x] */
25    }
26 }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30 #pragma omp parallel for
31   for ( int sj = 0; sj < n; sj += BLOCKSIZE )
32     for ( int si = 0; si < n; si += BLOCKSIZE )
33       for ( int sk = 0; sk < n; sk += BLOCKSIZE )
34         do_block(n, si, sj, sk, A, B, C);
35 }
```

图 6-25 图 5-48 中的 DGEMM 程序的 OpenMP 版。第 30 行是唯一一条 OpenMP 语句，它使最外层的 for 循环并行执行。这一行代码是与图 5-48 中的唯一区别

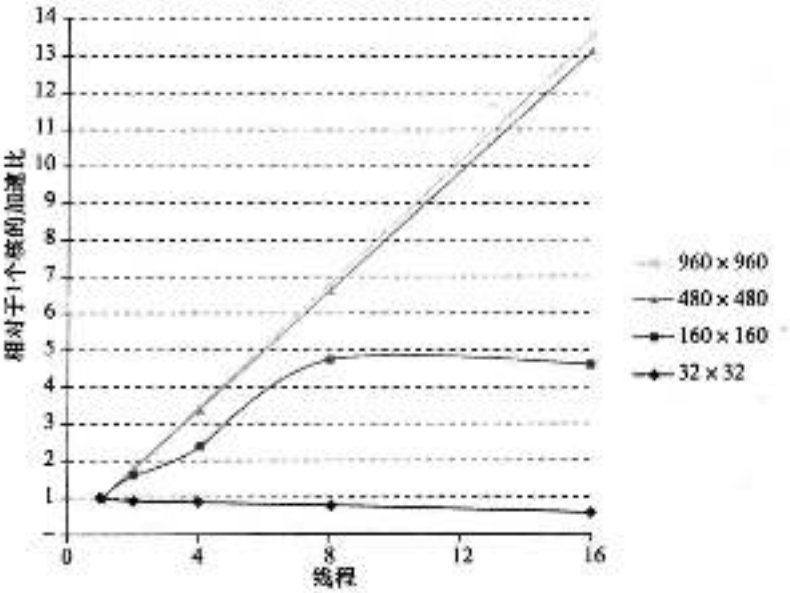



图 6-26 与单线程相比，当线程数增多时的性能提升。最客观的方法是拿多线程的性能与最优的单线程的性能相比，这也是我们的做法。这个图是与图 5-48 中没有使用 OpenMP 的pragma语句的代码相比较的

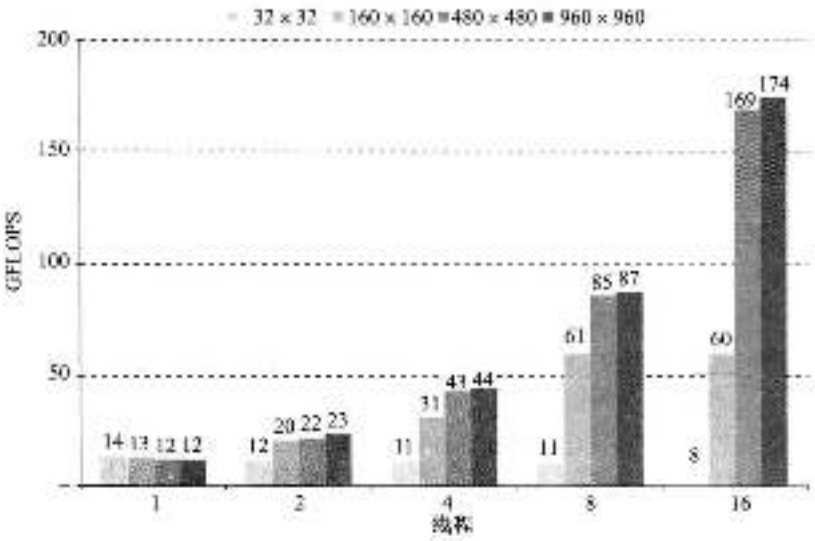



图6-27 4个不同大小的矩阵的DGEMM程序的性能。对于960×960矩阵，在使用16个线程时与图3-21中的未经任何优化的代码相比，性能提升了212倍

**小1 精解** 这些结论是在Turbo模式关闭的情况下得到的。在这个系统中，我们使用的是一个双芯片系统，所以事实之中，我们无论使用1个线程（只使用一个芯片上的一个核）还是2个线程（每个芯片上使用一个核）都能得到完整的Turbo加速（3.3/2.6 = 1.27）。当我们增加线程数目（使用的核数也增多），从Turbo模式的获益将减少，因为会有更少的功耗来使用这些核。对于4个线程，平均Turbo加速比对于4个线程、8个线程和16个线程分别为1.23、1.13和1.11。

555 ~ 557

**小1 精解** 虽然Sandy Bridge支持每个核两个硬件线程，但当使用32个线程时我们无法得到更多的性能提升。这是因为一个AVX硬件被同一个核上的两个线程所共享，所以当为一个核分配两个线程时实际上会对性能有所损害。

## 6.13 谬误与陷阱

*十多年来，一直有人在多台单处理器的组织形式已经到达了性能极限，并且性能的真正改进只能通过将多台计算机互连从而以这种方式支持协同计算……事实证明单处理器的性能一直在不断增长……*

——*Gene Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", Spring Joint Computer Conference, 1967*

对并行处理的大量研究揭示了诸多谬误和陷阱。我们在这里讨论其中4个。

**谬误：Amdahl定律不适用于并行计算机。**

1987年，一个研究组织的负责人宣称Amdahl定律已经被多处理器所打破。为了试图理解这些媒体报道的依据，我们首先看一下对Amdahl定律的相关引用[1967, p. 483]：

*此时可以得出的一个相当直观的结论是：花费在获得高并行处理速度上的努力都是无用的，除非顺序处理速度提高的数量级也与其十分接近。*

这句话依然是正确的：程序中被忽视的部分必然限制性能。该定律的一种解释可得到下面一条引理：每个程序中都有一部分是顺序的，因此必然有一个经济的处理器数量上界——比如说是100。通过给出使用1 000个处理器也可以达到线性增长，证明该引理是错误的；因而得出了Amdahl定律被打破的结论。

这些研究人员的方法是使用弱比例缩放：他们不是在相同的数据集上将速度提高1000倍，而是在可比较时间内将计算量提高1000倍。对于他们的算法，程序中顺序执行的比例是常数，与问题的输入规模无关，而其余部分则是完全并行的——因此，使用1000个处理器时依然为线性增长。

Amdahl定律显然也适用于并行处理器。这项研究确实指出了更快的计算机主要用途之一是完成更大规模的问题。只要确保用户真的关心这些问题并把这些作为购买更贵的计算机的理由，而不是关心为找到可以使很多处理器保持忙碌的问题。

谬误：峰值性能可代表实际性能。

超级计算机业界在市场中曾经使用该度量方法，并且该谬误在并行机中更加严重。市场营销人员不仅在单处理器节点使用这种几乎不可能达到的峰值性能指标，而且还将其乘以处理器的总个数，从而假定并行机可以达到完美的加速度！Amdahl定律指出达到两种峰值是多么困难；将两者相乘就错上加错了。屋顶线模型有助于达到合乎比例的峰值性能。

陷阱：在利用和优化多处理器体系结构时不开发软件。

在很长的时间里并行软件一直落后于并行硬件，可能是因为软件问题困难得多。我们给出一个例子说明这一问题，但是可供选择的例子还有很多！ 558

在将为单处理器设计的软件移植到多处理器环境时经常会遇到这样一个问题。例如，SGI操作系统最初假定页分配不频繁，从而通过一个锁来保护页表。在单处理器中，这不是一个性能问题。在多处理器中，对某些程序会成为一个主要的性能瓶颈。考虑一个程序在启动时需要初始化大量页的情况，正如UNIX为静态分配页所做的操作那样。假设该程序被并行化以便多核进程分配页。由于页的分配需要使用页表，而页表在每次使用时必须锁定，即使操作系统内核支持多线程，如果这些进程试图同时请求分配页（这恰好就是我们在初始化时所预期的情况）也会因此串行执行。

页表的串行操作影响了初始化时的并行，并对整个并行性能有着很大的影响。该性能瓶颈甚至在任务级并行中也存在。例如，假设我们将并行处理程序分为若干独立的作业并分别在一个处理器上运行一个作业，这样在不同作业之间就没有任何共享。（这恰好是一个用户的做法，因为他合乎情理地相信性能问题是由于应用程序中非预期的共享或冲突所造成的。）不幸的是，锁机制依然将所有工作串行化——因此说明即使互相独立的工作性能也会很低。

该陷阱说明当软件在多处理器上运行时，这种微妙但对性能有极大影响的错误会显现出来。和其他许多主要软件一样，操作系统的算法和数据结构在多处理器上需要重新考虑。在页表的更小区域加锁可以有效地避免这个问题。

谬误：可以在不提升存储器带宽的前提下得到好的向量计算性能。

从Roofline模型中可以看到，存储带宽对各种体系结构都很重要。DAXPY每个浮点操作需要1.5个存储访问，对于很多科学计算代码这是一个很标准的比例。即使浮点操作不需要花费时间，一个Cray-1计算机也不会增加DAXPY向量序列的性能，因为它的存储受到限制。当编译器使用阻塞机制改变计算，以使数据可以保存在向量寄存器中时，Cray-1运行Linpack的性能有了跳跃式提升。这个方法降低了每个浮点运算的访存次数并使性能提升了将近两倍。因此，Cray-1的存储带宽对于之前有更多带宽需求的循环来说足够了，这正是Roofline模型所预言的。 559

## 6.14 本章小结

*我们正在将未来产品的开发专注于多核设计。我们相信这对工业界是一个重要的拐点。……这不是一场竞赛。这是计算的翻天覆地的变化……*

*——Paul Otellini，Intel总裁，Intel开发者论坛，2004*

自从计算开始之日，人们就梦想着通过简单的集成若干处理器构建计算机。然而，构建并充分有效利用并行处理器的进程是缓慢的。其原因一方面是受软件难点的限制，另一方面是为

了提高可用性和效率，多处理器的体系结构在不断改进。本章中我们讨论了许多软件方面的挑战，包括编写由于 Amdahl 定律可获得高加速比程序的难点。不同并行体系结构之间往往存在巨大差异，所取得的性能提升也非常有限，而且过去许多并行体系结构的生命周期非常短暂，这些因素使得软件更加困难。网站上的 6.15 节讨论了这些多处理器的历史。要对本章所讲述的主题有更深入的理解，请参阅《计算机体系结构：量化研究方法》第 5 版第 4 章中的更多关于 GPU 以及 CPU 与 GPU 之间进行对比的内容，还有第 6 章中关于 WSC 的内容。

正如第 1 章所述，信息技术业的未来与并行计算是紧密联系在一起的。像过去一样，尽管有很多努力会失败，但是依然有很多理由让我们充满希望：

- 显然，软件即服务（SaaS）的重要性正在增长中，并且集群已经被证实为提供此类服务的一种非常成功的方法。通过提供高层次的冗余、包括地理分布的数据中心，此类服务可以为全世界的客户提供 24×7×365 的可用性。
- 我们相信仓储级计算机正在改变服务器设计的目标和原则，就像移动客户端的需求正在改变微处理器设计的目标和原则一样。这两者同样也都在革新软件行业。移动客户端软件和 WSC 硬件都受一美元所能带来多少性能和每焦耳所能带来多少性能的驱动，并行是提供这些目标的关键。
- SIMD 和向量操作很适合在后 PC 时代占据重要地位的多媒体应用。它们比经典的并行 MIMD 编程更简单并且在功耗方面效率更高。为了认识到 SIMD 与 MIMD 的重要性，图 6-28 给出了随着时间的增长 MIMD 中的核数，以及 x86 计算机中 SIMD 模式下每时钟周期的 32 位和 64 位操作个数。对于 x86 计算机，我们期待每两年在芯片上增加两个核并且每四年 SIMD 宽度增加一倍。在这些假设的前提下，在下一个十年后，SIMD 的并行加速会是 MIMD 并行的两倍。给出了 SIMD 对于多媒体的效率和它在后 PC 时代越来越大的重要性，这个推论可能是正确的。因此，至少应理解 SIMD 并行和 MIMD 并行同等重要，尽管后者已经得到了更多的关注。

560

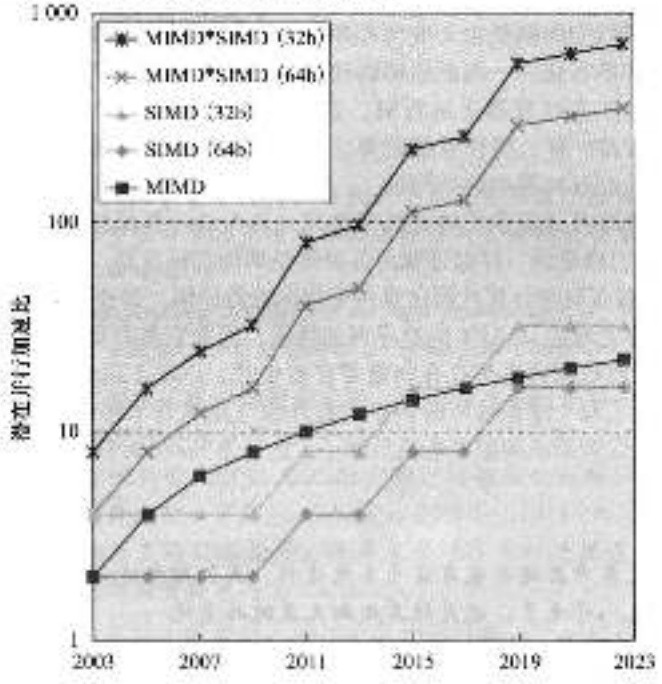



图 6-28 MIMD 和 SIMD 的潜在并行加速比，以及 x86 计算机的 MIMD 和 SIMD 随着时间的变化。该图假设每两年单芯片上 MIMD 的核数增加两个，并且每四年 SIMD 操作的数目增加一倍

- 并行处理在科学计算和工程计算等领域中非常普遍。此类应用领域对计算能力几乎充满无限的渴望。而且有很多应用具有天然的并行性。集群再一次地占据了此类应用领域。例如，根据 2012 Top 500 报告，集群占据了 500 个最快的 Linpack 报告的 80%。
- 为了获得更高性能，所有的桌面和服务器微处理器制造商正在构建多处理器，顺序应用程序不会像过去一样再有获取更高性能的捷径。正如我们之前所说的，串行程序现在就是慢的程序。因此，需要更高性能的程序员必须将自己的代码并行化，或者编写全新的并行处理程序。 561
- 在过去，微处理器和多处理器在成功上的定义是不同的。当缩放单处理器性能时，如果单线程性能随硅面积的开方增长，微处理器设计者会感觉很满意。也就是说，他们满足于性能随资源数量的亚线性增长。多处理器的成功在过去通常定义为与处理器数量相关的线性加速比函数，并假定 $n$ 个处理器的购买成本或管理成本是单一处理器的 $n$ 倍。目前并行正在以片上多核的形式实现，我们可以使用已经获得成功的传统微处理器标准来获得亚线性的性能提升。
- 运行时编译技术和自动运行的成功使得软件更容易适应每芯片上核数量的增长，提供受静态编译器限制所不能提供的灵活性。
- 与过去不同的是，开放源代码运动已经成为软件业的一个关键部分。这项运动可以改善工程解决方案，促进开发者之间的知识共享。它也鼓励创新，在改变旧软件时欢迎新的语言和软件产品。这种开放式的文化必将有益于目前日新月异的时期。

为使读者接受这个改变，我们通过快速浏览第 3～6 章的章节来展示如何通过 Intel Core i7（Sandy Bridge）处理器发掘矩阵乘法的潜在并行：

- 第 3 章中的数据级并行通过使用 256 位的 AVX 指令并行执行 4 个 64 位浮点运算使性能提升了 3.85 倍，这展示了 SIMD 的价值。
- 第 4 章中的指令级并行 4 次展开循环给乱序执行的硬件提供了更多的指令去调度，这又使性能提升了 2.3 倍。
- 第 5 章中的 cache 优化使用 cache 阻塞来减少 cache 缺失，这对不能放进 L1 cache 的矩阵性能提升了 2.0－2.5 倍。
- 本章中的线程级并行通过使用多核芯片上的所有 16 个核使无法放入单一 L1 cache 的矩阵的性能提升了 4－14 倍，这展示了 MIMD 的价值。我们是通过加入了一行 OpenMP pragma 语句实现的。

使用本书中的方法并且根据该计算机对软件进行改变，在 DGEMM 程序上加了 24 行代码。对于 32×32、160×160、480×480 和 960×960 的矩阵，通过这几行代码和本书的方法得到的总的性能加速比为 8、39、129 和 212！ 562

软硬件接口上的并行变革也许是过去 60 年来所面临的最大挑战。你也可以把它当做最大的机遇，就像我们快速浏览各章章节时所展示的。这个变革在 IT 界内外提供了大量研究和商业前景，并且主导多核的公司并不一定与主导单处理器的公司相同。在理解了硬件发展的潮流以及学会了如何根据硬件来改变软件之后，也许你就会抓住其中的机会，成为创新者中的一员。我们期待从你的发明创造中获益！

## 6.15 历史观点和拓展阅读

本节在网上主要给出了近 50 年来多处理器的发展历史。

**参考文献**

G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. IEEE Computer, 37(11):48–58, 2004.

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud computing, June 10–11, 2010, Indianapolis, Indiana, USA, doi:10.1145/1807128.1807152.

## 6.16 练习题

**6.1** 首先写一个每周你通常需要完成的日常活动的列表。例如，你可能会起来、淋浴、穿衣服、吃早饭、弄干头发、刷牙。确保列表中至少包含10项活动。

563 **6.1.1** [5]<6.2>考虑哪些活动已经利用了某种形式的并行性（例如，是同时刷多颗牙还是一次只刷一颗牙，是一次只带一本书到学校，还是将所有书装到背包里一次"并行"携带）。对每个活动都分析是否已经并行工作，如果没有，分析其原因。

**6.1.2** [5]<6.2>接下来考虑那些活动可以并发执行（例如，吃早餐和听新闻）。对每个活动都分析那些活动可以与其配对并发执行。

**6.1.3** [5]<6.2>对练习题6.1.2，可以通过改变现有系统（例如，淋浴设备、衣服、电视机、汽车等）中的什么来让我们并行执行更多的任务？

**6.1.4** [5]<6.2>如果你能尽可能多地并行执行任务，估计完成这些任务可以缩短的时间是多少？

**6.2** 假设需要你制作3块蓝莓蛋糕。蛋糕的配料如下：

1杯黄油，软化后再用

1杯糖

4个大鸡蛋

1茶匙香草精

0.5茶匙盐

0.25茶匙肉豆蔻

1.5杯面粉

1杯蓝莓

蛋糕的制作流程如下：

第1步：烤箱预热至160°C（325°F）。在烤盘上抹黄油和一层薄薄的面粉。

第2步：在一只大碗中使用搅拌器以中速将奶油和糖混合在一起，直到松发。再加鸡蛋、香草精、盐和肉豆蔻，搅拌到完全混合。将搅拌器降到低速，一次加入0.5杯面粉，搅拌到完全混合。

第3步：最后轻轻加入蓝莓，将蛋糕均匀地放在烤盘中，烘烤约60分钟。

**6.2.1** [5]<6.2>你的任务是尽可能高效率地完成3块蛋糕。假定只有一个能容纳一块蛋糕的烤箱，一个大碗、一个烤盘、一个搅拌器，请做出合理的调度以尽可能快地完成任务，并分析瓶颈所在。

**6.2.2** [5]<6.2>假设你现在有3个碗、3个蛋糕盘子和3个搅拌器。你拥有这些增加的资源后，现在的工序加快了多少？

564

**6.2.3** [5]<6.2>假设你现在有两个朋友，可帮你烹饪，并且你有一个可容纳3个蛋糕的大烤箱。这些将对练习题6.2.1中的计划有何改变？

**6.2.4** [5]<6.2>将制作蛋糕与并行计算机中的循环迭代进行类比。分析制作蛋糕的循环中存在的数据级并行和任务级并行。

**6.3** 许多计算机应用程序需要在一组数据中进行搜索和对数据进行排序。为了减少这些任务的执行时间，已经实现了几种高效的搜索和排序算法。在本练习中，我们将考虑如何将这些任务的并行最大化。

**6.3.1** [10] <6.2> 请看下面的二进制搜索算法（一种经典的分而治之算法），该算法可以在已经排序的 $N$ 元素数组 $A$ 中搜索值 $X$，并返回匹配项的索引号：

```
BinarySearch(A[0..N-1], X) {
  low = 0
  high = N - 1
  while (low <= high) {
      mid = (low + high) / 2
      if (A[mid] > X)
         high = mid -1
      else if (A[mid] < X)
         low = mid + 1
      else
         return mid // found
  }
  return -1 // not found
}
```

假设 BinarySearch 运行在具有 $Y$ 个核的多核处理器上，且 $Y$ 远远小于 $N$。请问预期的加速比是多少？请画图表示。

**6.3.2** [5] <6.2> 接下来，假设 $Y$ 与 $N$ 相同，这会对你前面的结论有何影响？如果要求你获得尽可能高的加速比（强比例缩放），请问该如何修改代码？

**6.4** 请看下面的 C 代码片段：

```
for (j=2;j<1000;j++)
    D[j] = D[j-1]+D[j-2];
```

565

与之对应的 MIPS 代码如下所示：

```
        addiu   $s2,$zero,7992
        addiu   $s1,$zero,16
loop:   l.d     $f0, -16($s1)
        l.d     $f2, -8($s1)
        add.d   $f4, $f0, $f2
        s.d     $f4, 0($s1)
        addiu   $s1, $s1, 8
        bne     $s1, $s2, loop
```

每种指令的延迟如下（以周期为单位）：

| add.d | l.d | s.d | addiu |
|---|---|---|---|
| 4 | 6 | 1 | 2 |

**6.4.1** [10] <6.2> 执行一次循环所有的指令需要多少周期？

**6.4.2** [10] <6.2> 在循环中，如果后面重复执行的指令会依赖于前面指令产生的结果，我们说循环内重复存在循环进位相关性（loop-carried dependence）。请分析上面代码中的循环进位相关性，识别其中相关的程序变量和汇编级寄存器。可忽略循环变量 j。

**6.4.3** [10] <6.2> 第 4 章中描述了循环展开。对此循环进行展开，并考虑将此代码运行在一个 2 节点的基于消息传递的分布式存储器系统中。假定我们采用 6.7 节描述的消息传递机制，操作 send（x，y）可向节点 x 发送值 y，操作 receive() 可等待正在发送的数。再假定 send 操作的发射需要 1 个周期（也就是说，同一节点的后续指令可在下个周期执行），而接收节点需要 10 个周期接收。接收指令会阻塞接收节点上指令的执行，一直等到接收节点完成消息接收为止。假设循环会执行 4 次，请计算在该基于消息传递的系统中完成循环所需的周期数。

**6.4.4** [10] <6.2> 互联网络的延迟是决定消息传递系统效率的重要因素之一。请问为了让练习题 6.4.3 中的分布式系统能获得任意加速比，互联网络需要提供多快的速度？

**6.5** 考虑下面的归并排序算法（另一种经典的分而治之算法）。归并排序由 John Von Neumann 于 1945

年首先提出。其基本思想是将含有 n 个元素的未排序序列 x 分为两个子序列，其中每个序列长度
566 都大约是原来的一半。然后对每个子序列重复类似的动作，直到每个子序列的长度均为 1。再从长度为 1 的子序列开始，将两个子序列“归并”为一个排序的序列。

```
Mergesort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = Mergesort(left)
        right = Mergesort(right)
        result = Merge(left, right)
        return result
```

下面的代码实现归并步骤：

```
Merge(left,right)
    var list result
    while length(left) >0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    if length(left) >0
        append rest(left) to result
    if length(right) >0
        append rest(right) to result
    return result
```

**6.5.1** [10] <6.2> 假设 MergeSort 运行在具有 Y 个核的多核处理器上，且 Y 远远小于 n（长度）。请问预期的加速比是多少？请画图表示。

**6.5.2** [10] <6.2> 接下来，假设 Y 与 n（长度）相同，这会对你前面的结论有何影响？如果要求你获
567 得尽可能高的加速比（例如，强比例缩放），请问该如何修改代码？

**6.6** 矩阵乘在大量应用中都扮演重要角色。两个矩阵可以相乘的条件是第一个矩阵的列数和第二个矩阵的行数相同。

假设我们有一个 $m \times n$ 的矩阵 $A$，还有一个 $n \times p$ 的矩阵 $B$ 与之相乘，乘法结果为一个 $m \times p$ 的矩阵 $AB$（或 $A \cdot B$）。如果令 $C = AB$，$C_{i,j}$ 代表在矩阵 $(i, j)$ 位置处的值，则 $1 \leqslant i \leqslant m$ 且 $1 \leqslant j \leqslant p$。现在我们考虑是否可以将 $C$ 的计算并行化。假设矩阵在存储器中的存放顺序为：$a_{1,1}$，$a_{2,1}$，$a_{3,1}$，$a_{4,1}$，……

**6.6.1** [10] <6.5> 假设我们分别在单核/四核共享存储器的系统计算 $C$，请问四核相对于单核的预期加速比是多少？可忽略存储器相关的问题。

**6.6.2** [10] <6.5> 如果对 $C$ 的更新会导致 cache 缺失（例如更新一行中连续的元素时可能引起伪共享），重新计算练习题 6.6.1 中的问题。

**6.6.3** [10] <6.5> 有什么办法消除可能出现的伪共享问题？

**6.7** 下面的两个程序同时运行在一个包含 4 个处理器的 SMP（对称多核处理器）中。假设在开始运行之前，x 和 y 的初值均为 0。

核 1：x = 2;

核 2：y = 2;

核3：w = x + y + z;

核4：z = x + y;

**6.7.1** [10] <6.5> w、x、y、z 所有可能的结果分别是什么？对每种可能的情况，通过分析指令的交错情况，解释其产生的原因。

**6.7.2** [5] <6.5> 采用什么措施能让执行变成更有确定性，以便只产生一种结果。

**6.8** 哲学家就餐问题是一个经典的同步和并发问题。该问题假设坐于一个圆桌周围的哲学家们可以做两件事之一：吃饭或思考。当他们吃饭时，他们不能思考，反之亦然。在圆桌中心有一碗通心粉。每两个哲学家之间有一只叉子，这样每个哲学家左面有一把叉子，右面也有一把叉子。按照吃通心粉的方式，哲学家需要两把叉子才能吃通心粉，而且只能使用紧挨着他左右的两把叉子。哲学家不能和其他人说话。

568

**6.8.1** [10] <6.7> 请描述没有任何哲学家可以吃通心粉的情景。什么样的事件序列会导致该问题发生？

**6.8.2** [10] <6.7> 如何通过引入优先级的概念来解决这一问题？这样可以对所有哲学家公平对待吗？请解释原因。

现在假定我们增加一个服务员负责为哲学家们分配叉子。只有在服务员允许之下他们才可以拿起叉子。服务员也知道所有叉子的状态。而且我们要求所有哲学家总是先请求拿起左边的叉子再请求拿起右边的叉子，这样可以避免死锁。

**6.8.3** [10] <6.7> 对服务员请求的实现，可以将请求放入一个队列，也可以让请求周期性地重试。采用队列方式，请求可以按收到的顺序依次处理。使用队列的问题是即使请求排在队列的最前面，我们也不能保证总是为其提供服务，因为可能缺乏所需的资源。试想使用1个队列为5个哲学家服务的情景，即使有的哲学家有两把叉子都可用但仍然不能为其服务（因为他的请求排在队列的后部）。

**6.8.4** [10] <6.7> 如果我们让请求周期性地重试直到资源变为可用，这样是否就解决了练习题6.8.3中的问题？请给出原因。

**6.9** 请看下面的3种CPU结构：

CPU SS：一个双核超标量微处理器，支持在两个功能单元（FU）上的乱序发射。每个核只能运行单一线程。

CPU MT：一个细粒度的多线程处理器，支持来自两个线程中指令的并发执行（也就是说，有两个功能单元），尽管每个周期只能从一个线程发射一条指令。

CPU SMT：SMT处理器支持来自两个线程的指令并发执行（也就是说，有两个功能单元），并且发射的指令可来自任一线程或者两个线程。

假定我们在这些CPU上运行线程X和线程Y，具体操作如下：

| 线程X | 线程Y |
|---|---|
| A1：需二个周期执行 | B1：需两个周期执行 |
| A2：无相关性 | B2：与B1使用的一个功能单元冲突 |
| A3：与A1使用的一个功能单元冲突 | B3：需要B2的结果 |
| A4：需要A3的结果 | B4：无相关性并且需要两个周期执行 |

569

除非特别标记或者遇到相关阻塞，假定所有的指令都是单周期执行。

**6.9.1** [10] <6.4> 如果使用一个SS CPU，执行这两个线程需要多少个周期？相关阻塞浪费了多少发射槽？

**6.9.2** [10] <6.4> 如果使用两个SS CPU，执行这两个线程需要多少个周期？相关阻塞浪费了多少发射槽？

**6.9.3** [10] <6.4> 如果使用一个MT CPU，执行这两个线程需要多少个周期？相关阻塞浪费了多少发射槽？

**6.10** 虚拟化软件正在用于降低管理高性能服务器的成本。包括 VMWare、Microsoft 和 IBM 公司在内的很多公司正在开发一系列的虚拟化产品。第5章中介绍的管理程序层（hypervisor layer）位于硬件和操作系统之间，使得多个操作系统可以共享同一物理硬件。管理程序层负责分配 CPU 和存储器资源，同时处理原本由操作系统完成的服务（如I/O）。

虚拟化为宿主操作系统和应用软件提供了底层硬件的一个抽象层，使得若干操作系统可并行运行在共享的 CPU 和存储器上。我们需要重新考虑未来如何设计多核和多处理器系统来对此进行支持。

**6.10.1** [30] <6.4> 选择市场上的两种管理程序，比较它们虚拟化和管理底层硬件（CPU 和存储器）的方式。

**6.10.2** [15] <6.4> 为了更好地满足未来多核 CPU 平台的资源需求，可采取哪些措施？例如，多线程技术是否可以减轻计算资源间的竞争？

**6.11** 我们将讨论如何高效地执行下面的代码。假设我们有两种不同的机器，一种是 MIMD，另一种是 SIMD。

```
for (i=0; i < 2000; i++)
  for (j=0; j<3000; j++)
      X_array[i][j] = Y_array[j][i] + 200;
```

**6.11.1** [10] <6.3> 对一个包含4个 CPU 的 MIMD 机器，请给出每个 CPU 上执行的 MIPS 指令序列。此 MIMD 机器的加速比是多少？

**6.11.2** [20] <6.3> 对一个宽度为8的 SIMD 机器（也就是说，包含8个并行的 SIMD 功能单元），使用你自己的对 MIPS 的 SIMD 扩展编写一个执行该循环的汇编程序，并比较 SIMD 和 MIMD 上执行指令的数量。

570

**6.12** MISD 机器的一个例子是脉动阵列（systolic array）。它是一个由数据处理单元构成的流水线网络或波阵面。这些单元都不需要程序计数器，因为执行是通过数据到达触发的。时钟脉动阵列以与每个处理器相"锁步"的方式进行计算，而这些处理器承担了交替的计算和通信。

**6.12.1** [10] <6.3> 分析脉动阵列的各种实现机制（可以在互联网或由版物中查找相关资料），然后使用 MISD 模型对练习题6.11中的循环进行编程，并对遇到的问题进行讨论。

**6.12.2** [10] <6.3> 应用数据级并行中的各种术语，分析 MISD 和 SIMD 之间的相似点和不同点。

**6.13** 假定我们要执行本章讲述 NVIDIA 8800 GTX GPU 时提到的 DAXPY 循环。在这一问题中，我们假定所有算术操作是单精度浮点数运算（因此我们将其重新命名为 SAXPY）。假定指令的执行周期数如下所示：

| Loads | Stores | Add.s | Mult.s |
|---|---|---|---|
| 5 | 2 | 3 | 4 |

**6.13.1** [20] <6.6> 请描述一下在一个8核处理器中如何构建 warp 来完成 SAXPY 循环？

**6.14** 从 http://www.nvidia.com/object/cuda_get.html 下载 CUDA Toolkit 和 SDK。注意使用代码的 emurelease（Emulation Mode）版本（此版本可在没有 NVIDIA 硬件的情况下运行）。编译 SDK 中提供的示例程序，并尝试它们运行在仿真器上。

**6.14.1** [90] <6.6> 以 SDK 的示例程序为起点，编写一个完成如下向量操作的 CUDA 程序：

1）$a - b$（向量减法）

2）$a \cdot b$（向量点积）

向量 $a = [a_1, a_2, \cdots, a_n]$ 和 $b = [b_1, b_2, \cdots, b_n]$ 的点积定义如下：

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

运行编写的程序并验证结果是否正确。

**6.14.2** [90] <6.6> 如果你有可用的 GPU 硬件，请完成对程序的性能分析，并查看在向量大小不同的情况下 GPU 和一个 CPU 版本的计算时间，并解释其中的原因。

571

6.15 AMD 最近宣布将把 GPU 与 x86 核集成到一个封装中，尽管两者的时钟不同。这是我们能在不远的将来看到的一种异构多处理器系统商业化产品。设计的关键之一是如何支持 CPU 和 GPU 之间的高速数据通信。当前，必须在 CPU 和 GPU 之间进行通信，这在 AMD 的 Fusion 体系结构中发生了变化。目前的计划是采用多个（至少 16 个）PCI express 通道来实现高速通信。Intel 也使用 Larrabee 芯片进行了类似的研究，通信计划采用 QuickPath 互联技术。

6.15.1 [25] <6.6> 比较这两种互联技术的带宽和延迟。

6.16 参照图 6-14b 中给出的 3 阶 $n$ 维立方体互连拓扑结构，其将 8 个节点进行了互连。$n$ 维立方体互连拓扑的一个优势是在部分互连损坏的情况下依然可以保持连接性。

6.16.1 [10] <6.8> $n$ 维立方体中最多有多少互连损坏时还能保证任何节点依然能够连通？请写出计算公式。

6.16.2 [10] <6.8> 比较 $n$ 维立方体和全互联网络的可靠性。画图比较两种拓扑分别在多少连接损坏增加时会导致连接失效。

6.17 基准测试程序用于在指定的计算平台上运行有代表性的工作负荷，从而比较不同系统之间的性能。在本练习题中，我们将比较两种 benchmark：Whetstone CPU benchmark 和 PARSEC Benchmark suite。从 PARSEC 中选择一个程序。所有程序都可从网上免费下载。考虑将 Whetstone 的多份备份或 PARSEC Benchmark 运行在 6.11 节中描述的各个系统上。

6.17.1 [60] <6.10> 两种工作负载运行在这些多核系统上的本质区别是什么？

6.17.2 [60] <6.10> 使用 Roofline 模型的相关术语，分析在运行了这些 benchmark 时，运行情况与工作负荷中共享和同步的数量相关性有多大？

6.18 在计算稀疏矩阵时，存储器的延迟至关重要。由于稀疏矩阵缺乏矩阵操作中常见的空间局部性，所以需要研究新的矩阵表示方法。

最早的稀疏矩阵表示方法之一是 Yale 稀疏矩阵格式。它使用 3 个一维数组存储维数 $m \times n$ 的矩阵 $M$。令 $R$ 代表 $M$ 中的非零项数目。我们构造一个长度为 $R$ 的数组 $A$ 存储 $M$ 中的所有非零项（按照从左到右、从上到下的顺序）。我们再构造一个长度为 $m+1$ 的数组 $IA$。$IA(i)$ 包含第 $i$ 行中第一个非零项在 $A$ 中的索引号。原矩阵中的第 $i$ 行的元素可从 $A(IA(i))$ 到 $A(IA(i+1)-1)$ 中得到。第二个数组 $JA$ 包含 $A$ 中每个元素的列号，因此它的长度也为 $R$。 [572]

6.18.1 [15] <6.10> 分析下面的稀疏矩阵 $X$，并编写 C 程序将其存储为 Yale 稀疏矩阵格式。

```
Row 1 [1, 2, 0, 0, 3, 0]
Row 2 [0, 0, 1, 1, 3, 0]
Row 3 [0, 0, 0, 0, 3, 0]
Row 4 [2, 0, 0, 0, 3, 2]
Row 5 [0, 0, 3, 3, 3, 7]
Row 6 [1, 3, 0, 0, 3, 1]
```

6.18.2 [10] <6.10> 在存储空间方面，假定矩阵 $X$ 中的每个元素都是单精度浮点格式，如果用 Yale 稀疏矩阵格式存储上面的矩阵，请计算共需多少存储空间。

6.18.3 [15] <6.10> 执行下面给出的矩阵 $X$ 和矩阵 $Y$ 的矩阵乘。

```
[2, 4, 1, 99, 7, 2]
```

将该计算放入循环中，并对执行过程进行计时。确保增加循环执行的次数，以在你的时间测量中获得较好的分辨率。比较矩阵的原始表示的运行时间和 Yale 稀疏矩阵格式的运行时间。

6.18.4 [15] <6.10> 你是否能够找到更加有效的稀疏矩阵表示方法（考虑空间和计算开销）？

6.19 在未来的系统中，我们期待能够看到由异构 CPU 构成的异构计算平台。在嵌入式处理相关市场，一些同时包含浮点 DSP 和微控制 CPU 的多芯片模块包的系统已经开始呈现。

假定你有三类 CPU：

CPU A——每周期可执行多条指令的中断多核 CPU（具有浮点单元）。

CPU B——每周期可执行单条指令的快速单核微型 CPU（例如，无浮点单元）。

CPU C——每周期可执行同样指令的多个副本的慢速向量 CPU（具备浮点能力）。 [573]

假定我们的处理器在下面的频率运行：

| CPU A | CPU B | CPU C |
|---|---|---|
| 1GHz | 3GHz | 250MHz |

在每个时钟周期，CPU A 可以执行 2 条指令，CPU B 可以执行 1 条指令，CPU C 可以执行 8 条指令（尽管是相同指令），假定所有的操作在单周期延迟中完成执行，且没有任何冒险。

三个 CPU 均可执行整数算术，尽管 CPU B 不能直接执行浮点算术。CPU A 和 B 具有与 MIPS 处理器相似的指令集。CPU C 仅能执行浮点加、减和存储器存、取操作。假定所有 CPU 均可访问共享存储器，并且同步的开销为零。

我们的任务是比较两个矩阵，$X$ 和 $Y$，它们每个都包含 1 024 × 1 024 个浮点元素。输出结果应是指示矩阵 $X$ 中何处的值比矩阵 $Y$ 中的值大或相等的一系列数。

**6.19.1** [10] <6.11> 请描述如何划分该问题到 3 个不同的 CPU 上，以获得最佳性能。

**6.19.2** [10] <6.11> 你会向向量 CPU C 中增加哪类指令，以获得更好的性能？

**6.20** 假定一个云核计算机系统可以处理每秒钟具有稳定状态率的数据库事务。到目前，每个事务平均花费固定的时间来处理。下表给出了几对事务延迟和处理速率。

| 平均事务延迟 | 最大事务处理速率 |
|---|---|
| 1ms | 5 000/s |
| 2ms | 5 000/s |
| 1ms | 10 000/s |
| 2ms | 10 000/s |

对于表中的每一对数据，回答如下问题：

**6.20.1** [10] <6.11> 在任意给定的瞬间，平均有多少请求被处理？

**6.20.2** [10] <6.11> 如果移到 8 核的系统中，理想情况下，系统的吞吐量将发生什么变化（例如，计算机每秒处理多少请求）？

574 **6.20.3** [10] <6.11> 讨论为什么通过简单地添加核的数量，我们很少获得这种加速？

## 小测验答案

**6.1** 错误。任务级并行可以帮助串行应用，可以使串行应用在并行硬件上运行，尽管会有很多挑战。

**6.2** 错误。弱缩放可以补偿程序的串行部分，强缩放的缩放性会被串行部分所限制。

**6.3** 正确。但是它们缺少可以提升向量体系结构性能的特性，如聚集 - 分散和向量长度寄存器。（就像这节中的精解中提到的，AVX2 SIMD 扩展通过聚集操作提供了变址加载但不通过分散操作提供变址存储。Haswell x86 微处理器是第一个支持 AVX2 的处理器。）

**6.4** 1. 正确。2. 正确。

**6.5** 错误。由于共享地址是物理地址，且多任务中的每个任务都在它们自己的虚拟地址空间中，因而可在共享存储器多处理器上良好地运行。

**6.6** 错误。图形 DRAM 因其更高的带宽而被设计。

**6.7** 1. 错误。发送和接收消息是一个隐式的同步，同样也是一种共享数据的方式。2. 正确。

**6.8** 正确。

575 **6.10** 正确。我们或许需要在硬件的所有层次和软件栈上进行革新，以促并行计算成功。

# 汇编器、链接器和 SPIM 仿真器

James R. Larus
微软研究院

对思想中错的恐惧，不能成为阻止言论和集会自由的借口。

——*Louis Brandeis, Whitney v. California, 1927*

## A.1 引言

将指令编码为二进制数字对计算机来说自然且有效。然而人类理解和处理这些数字有很大的困难。人们读和写符号（文字）比读和写一长串的数字容易多了。第 2 章说明了我们不需要在数字和文字之间做出选择，因为计算机指令可以具备很多种表达方式。人类可以读而且写这些符号，并且计算机可以执行等价的二进制数字。本附录描述了人类可读的程序被处理的过程：把一种程序形式转换成另外一种计算机可以执行的方式。提供了一些编写汇编程序的暗示，并且解释如何在 SPIM 上运行这些程序，SPIM 是一个执行 MIPS 程序的仿真器。SPIM 的 UNIX、Windows 以及 Mac OS X 版本在 CD 中可以得到。

汇编语言是计算机二进制编码——机器语言（machine language）的符号表示。汇编语言比机器语言更具备可读性，因为它使用符号而不是二进制数字。这些汇编语言中的符号名字通常以二进制模式出现，例如，操作码和寄存器指示符，所以可以阅读并记住它们。另外，汇编语言允许编程者使用 labels 来识别和指定保存指令或数据的内存字。

A-1
|
A-3

➲ 机器语言：在计算机系统内部通信的二进制表示。

一个被称为**汇编器**（assembler）的工具来将汇编语言转换成二进制指令。汇编器提供了比机器 0 和机器 1 更友好的表达，使得写程序和读程序都简化了。操作和地址的符号名称是这种表达方式的一个方面。另一个方面是编程设备增加了程序的清晰度。例如，A.2 节讨论的宏（macros）允许程序员通过定义新操作来扩展汇编语言。

➲ 汇编器：将符号版本的指令翻译成二进制版本的一段程序。

➲ 宏：一种模式匹配和替换机制，提供了简单的机制来命名经常使用的指令序列。

汇编器读入一个汇编语言的源文件，产生一个包含机器指令以及帮助将几个目标文件整合成一段程序的标签信息的目标文件。图 A-1-1 说明了如何构建一个程序。很多程序由多个文件组成——也被称为模块——这些文件被分开单独编写、单独编译、单独汇编。一个程序可能使用程序库中提供的预先写好的例程。一个模块通常包含到子例程的引用以及在别的模块和库中定义的数据。模块中的代码直到对其他目标文件或者库的标签的**未确定的引用**（unresolved reference）全部解决时才能执行。另一个工具被称为**链接器**（linker），将目标代码和库文件整合成一个可执行文件，这个文件是计算机可以执行的。

➲ 未确定的引用：一个需要从外部源代码获取更多信息才能完成的引用。

- 链接器：也称为**链接编辑器**，是一个将独立的汇编机器语言程序组装起来，处理其中未定义的标签形成可执行文件的一个系统程序。

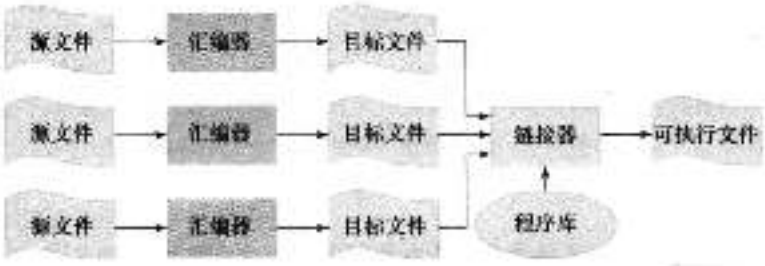
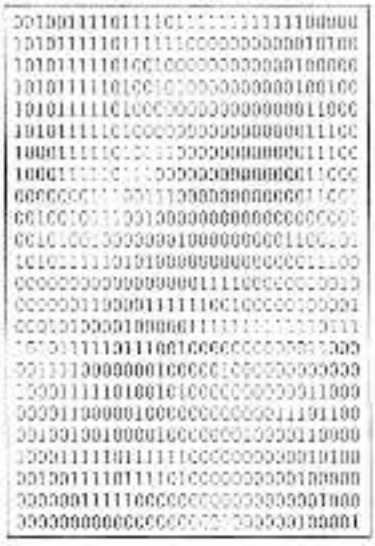



图 A-1-1 产生可执行文件的过程。一个汇编器将一个用汇编语言写的文件翻译成一个目标代码文件，这个目标代码文件又和其他的文件和库链接组成可执行文件

为了理解汇编语言的优势，考虑下面一系列图，这些图包含了一个短程序，这个程序计算并且打印出 0～100 的整数的平方和。图 A-1-2 展示一个 MIPS 计算机可以执行的机器语言。付出很多努力，你可以使用第 2 章指令表中的编码和指令格式来将指令转换成图 A-1-3 的类似的符号化程序。这个程序的形式是相当容易读的，因为操作和操作数使用符号，而不是使用二进制模式写的。然而，汇编语言仍然很难遵循，因为内存位置通过地址来指定而不是通过符号化标签来指定。

A-4



```
addiu   $29, $29, -32
sw      $31, 20($29)
sw      $4, 32($29)
sw      $5, 36($29)
sw      $0, 24($29)
sw      $0, 28($29)
lw      $14, 28($29)
lw      $24, 24($29)
multu   $14, $14
addiu   $8, $14, 1
slti    $1, $8, 101
sw $8,  28($29)
mflo    $15
addu    $25, $24, $15
bne     $1, $0, -9
sw      $25, 24($29)
lui     $4, 4096
lw      $5, 24($29)
jal     1048812
addiu   $4, $4, 1072
lw      $31, 20($29)
addiu   $29, $29, 32
jr      $31
move    $2, $0
```

图 A-1-2 MIPS 用来计算和打印出 0～100 的整数的平方和的机器语言代码

图 A-1-3 同一个程序（图 A-1-2）的汇编语言版。然而，这个程序的代码没有标记寄存器或者内存地址，也没有包含注释

图 A-1-4 展示了汇编语言使用记忆名称来标志内存地址指令。很多程序员喜欢以这种方式来读和写指令。那些名字前有个点，例如.data 以及.globl，是**汇编指令**（assembler directive），告诉汇编器如何翻译程序，但是不需要产生机器指令。名字后面跟一个冒号，如str:或者 main:，这些标签是下一个内存地址的名字。这个程序和汇编语言程序一样具有可读性（除了没有翻眼的注释），但是它还是很难遵循，因为需要很多简单的操作来完成简单的任务，因为汇编语言缺乏控制流结构，为程序的操作提供很少的暗示。

⊃ 汇编指令：一个告诉汇编器如何翻译程序，但是不会产生机器指令的操作。通常它以圆点开始。

对比之下，图 A-1-5 的 C 程序不但短而且很清晰，因为具有记忆名字的变量和循环是显式的，而不是分支结构的。实际上，C 程序是唯一一个我们自己写的程序。其他形式的程序都是 C 编译器和汇编器产生的。

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 28($sp)
        sd      $s0, 16($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 28($sp)
        addu    $sp, $sp, 32
        jr      $ra

        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

图 A-1-4 使用带有标签（label）的汇编语言写的同一个程序，但是没有注释。以圆点开始的指令是汇编指令（见 A.10 节）。.text 指示后续的行包含着指令。.data 指示出它们包含数据。.align n指示后面这些行的元素应该是按照 $2^n$ 来边界对齐的。因此，.align 2就是下一个元素按照字对齐。.globl main 声明了 main 是一个全局的符号，应当对于其他文件中的代码来说是可见的。最后，.asciiz 保存了内存中的空终结符

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

图 A-1-5 使用 C 程序语言编写的程序

通常，汇编语言扮演两个角色（见图 A-1-6）。第一个角色是编译器的输出语言。编译器将使用高级语言（C 或者 Pascal）写的程序翻译成机器语言或者汇编语言表示的等价的程序。 A-5
高级语言被称为**源语言**（source language），而编译器的输出是目标语言。

⊃ 源语言：一种直接用来编写程序的高级语言。

汇编语言的另一个角色是作为一种编程语言。这个角色通常是它的主要功能。然而，今天由于大的内存以及更优良的编译器，很多程序员使用高级语言编写程序，而且很少看见计算机执行的指令。然而，当速度和面积很关键或者为了开发硬件特性，而高级语言中没有这些特性时，汇编语言仍然是很重要的。




图 A-1-6 汇编语言由程序员编写或由编译器输出

虽然本附录关注 MIPS 汇编语言，但汇编语言编程在很多的其他机器上也是很相似的。CISC 机器（如 VAX）中的附加指令以及寻址模式，可使得汇编程序变短，但是不会改变程序的汇编流程，而且为汇编语言提供了高级语言的优势，例如，类型检测以及结构控制流。

A-6

## A.1.1 什么时候使用汇编语言

与高级语言相比，使用汇编语言编程的主要原因是，它在速度和代码大小方面具有优势，而这两者极为重要。例如，一台算机，它控制着机器的一个部分，如汽车刹车。一台计算机被合并到另一个设备中，例如一辆汽车，该计算机就被称作嵌入式计算机。这种类型的计算机需要对外部世界的事件做出快速的可预测的反应。由于编译器对操作所花费的时间引入了不确定性，程序员可能会发现很难保证高级语言编写的程序能在给定的时间间隔（传感器检测到轮胎打滑后的一毫秒内）做出响应。另一方面，一个汇编语言程序员，具有对指令执行的紧密控制。另外，在嵌入式应用中，减小了代码的体积，可使用更小的存储芯片，减小嵌入式计算机的代价。

一种混合的方法是，程序大部分用高级语言编写，时间关键部分用汇编语言编写，同时利用这两种语言。程序通常花费很多时间执行程序源代码中的很少一部分。这种发现就是 cache 中的局部性原理（见第 5 章的 5.1 节）。

A-7
A-8

程序分析测量一个程序在哪里花费了时间并找出其时间关键部分。大多数情况下，程序的这个部分可以使用更好的数据结构或者算法来实现。然而，有时候，显著的性能提高只能通过用汇编语言重写那段关键代码得到。

这种改进并不意味着高级语言的编译器就失效了。在为整个程序产生统一的高质量机器代码方面，编译器通常比程序员效果更好。然而，程序员比编译器在更深层次上理解程序的算法和行为，而且能够通过大量的努力和精巧的设计提高小段代码的质量。尤其，编程人员在编写代码时，同时考虑好几个子程序段。编译器通常单独编译一个程序段，而且必须遵循严格的规则，在程序段的边界处管理寄存器的使用。通过在寄存器中保存那些经常被使用的值，甚至跨越程序边界，编程人员可以使得程序运行得更快。

汇编语言的另一个主要优点是能够利用定制的指令——例如，字符串复制指令或者模式匹配指令。很多时候，编译器不能确定一个循环程序能不能被一条指令替代掉。然而，编写循环的那个程序员能够很容易地使用一个指令将其替换掉。

目前，由于编译技术的提高以及机器流水线的存在带来的复杂度（见第 4 章），程序员很难比编译器更具优势。

使用汇编语言的最后一个原因是，没有哪个高级语言能够适用于一个特定的计算机。很多老的或者定制的计算机没有编译器，所以编程人员唯一的选择就是汇编语言。

## A.1.2 汇编语言的缺点

汇编语言的很多缺点极大地限制了它的广泛使用。也许它的主要缺点就是使用汇编语言

编写的程序本质上是针对特定机器的，而且如果需要在另一种计算机结构中运行，就必须重写。第1章讨论了计算机的快速发展，意味着体系结构变得过时。一个汇编语言程序仍旧和它的原始体系结构紧紧地绑定在一起，即使该计算机已被崭新、快速、性价比更高的机器所遮蔽。

汇编语言的另一个缺点是，汇编程序比等价的高级语言程序更长。例如，图A-1-5的C程序仅有11行，但是图A-1-4的汇编程序有31行。在更复杂的程序中，汇编和高级语言程序的比率（扩展因子）将更长，远不止像在这个例子中的3倍这样。不幸的是，实际的研究表明，程序员能够每天大约编写同高级语言行数一样多的汇编语言。这就意味着程序员使用高级语言大约会产生$x$倍的生产率，这里的$x$是汇编语言扩展因子。 A-9

长的程序更难阅读和理解，而且这些代码会包含更多的错误使得问题更为恶化。汇编语言使得这个问题恶化，因为它缺乏完备的结构。常见的编程用语，例如if-then语句和循环，汇编语言必须通过分支和跳转来实现。导致程序变得很难读懂，因为读者必须从汇编语言的每一句来为每个高级语言结构重建。这是很困难的。例如，看图A-1-4并回答下面的问题：使用的什么循环？它的下界和上界分别是什么？

01 **精解** 不需要汇编器，编译器直接产生机器语言。与使用汇编器作为编译的一部分的那些编译器相比，这些编译器通常执行得更快。然而，产生机器码的编译器必须执行一个汇编器通常执行的那些任务，例如，翻译地址，将指令编码成二进制数字。在编译速度和编译器的简洁性之间进行折中。

01 **精解** 尽管有这些考虑，一些嵌入式应用使用高级语言编写。很多这样的应用的程序很大，而且很复杂，这样的程序必须极其可靠。汇编语言程序相对于高级语言程序更长而且更难编写。这极大地增加了使用汇编语言编写程序的代价，使得验证这些程序的正确性极其困难。事实上，这些考虑导致为这些嵌入式系统规模的国防部门开发Ada——一种编写嵌入式系统的新高级语言。

## A.2 汇编器

汇编器将汇编语言文件翻译成二进制机器指令和二进制数据组成的文件。翻译过程有两个主要步骤。第一步，找到标签（label）对应的内存地址，因此符号名字和地址之间的关系在指令被翻译的时候就确定了。第二步，将每个汇编语句的数字化的操作码、寄存器指示器和标签翻译成合法的指令。如图A-1-1所示，汇编器产生一个输出文件，叫作目标文件，目标文件包含机器指令、数据和书签信息。

目标文件通常不能被执行，因为它引用了其他文件中的过程或数据。如果标签目标可以被定义它的文件之外的其他文件所应用，那么这个就是**外部标签**（external label）（也称为**全局标签**）。如果仅仅能在定义它的文件内部被引用，则标签是局部的。很多的汇编器，默认标签是局部的，而且必须是显式声明为全局的。子程序和全局变量需要外部标签，因为它们在一个程序中被很多文件所引用。**局部标签**（local label）隐藏了对别的模块不可见的名字——例如，C中的静态函数仅仅被同一个文件中的函数所调用。另外，编译产生的名字——例如，一个循环的开始处的指令的名字——就是局部的，这样编译器就不需要为每个文件产生唯一的名字。 A-10

⊃ 外部标签：也称为全局标签。标签对应一个目标，这个目标可以在定义这个标签的文件之外被引用。

⇨ 局部标签：标签对应一个目标，这个目标仅仅可以被定义这个标签的文件内部引用。

**例1** 例题：局部和全局标签

考虑图 A-1-4 中的程序。这程序只有一个外部标签（全局）main，它包含了两个局部标签——loop 和 str——它们仅仅在这个汇编文件中是可见的。最终，包含一个对外部标签 printf 来说未处理（unresolved）的引用，printf 是一个打印数值的库程序。图 A-1-4 中的标签能否从另一个文件引用？

**例1** 答案

仅全局标签在外部是可见的，所以仅 main 标签可以在外部被引用。 □

因为汇编器独立地处理每一个文件，它仅知道每个局部标签的地址。汇编器依赖于别的工具，如利用链接器（linker）将目标文件以及库文件整合起来形成可执行文件，并且将外部标签处理掉。汇编器通过提供标签的列表以及未处理的引用来辅助链接器工作。

然而，局部标签对汇编器来说还是个令人感兴趣的挑战。和大部分高级语言中的名字不同，汇编标签可能在它们定义之前就使用。例如，在图 A-1-4 中，标签 str 在定义之前就被 la 指令使用。一个前向引用（forward reference）的可能性，就像前面这个例子，强迫汇编器将一个程序的翻译过程分成两步：首先找到所有的标签，然后产生指令。例如，当汇编器看到指令 la，它不知道这个标签为 str 的字在哪里，或者甚至不知道 str 这个标签到底是指令还是数据。

A-11

⇨ 前向引用：一个标签在被定义之前就被使用。

第一遍，汇编器将汇编文件的每一行读入，将其分解成几个部分。这些部分叫作词汇单位，都是独立的字、数字和标点符号。例如，下面的行

```
ble   $t0, 100, loop
```

包含6个词汇单元：ble 指令的操作码、寄存器说明符 $t0、逗号、数字 100、逗号，还有符号 loop。

如果一行以标签作为开始，汇编器在它的符号表（symbol table）中记录标签的名字，以及指令在内存中占据的内存字的地址。汇编器接着计算当前行中这个指令内存中占据多少个内存字。通过跟踪指令大小，汇编器可以确定下一条指令在哪里。为了计算一个可变长度的指令大小，例如在 VAX 中，汇编器必须仔细地确定这些。然而，对于固定长度的指令，像 MIPS 中的那些，仅仅需要粗略计算。汇编器采用类似的办法计算数据语句需要的空间。当汇编器到达一个汇编文件末尾时，符号表记录了文件中每个标签的位置。

⇨ 符号表：用来将标签的名字和指令占用的内存字的地址相匹配的一个表。

第二遍，汇编器使用整个文件的符号表中的信息，在这一遍产生机器代码。汇编器再一次检查文件中每一行。如果一行中包含了指令，汇编器将其指令码和操作数（寄存器指示器或者内存地址）组合成一条合法的指令。这个过程和第2章的2.5节的做法很相似。引用在另一个文件中定义的外部标签指令和数据字不能完全地汇编（因为它们是未决的），因为符号的地址不在符号表中。汇编器确实不能对这些未决的引用发牢骚，因为对应的标签很可能在另一个文件中定义。

**重点** 汇编语言是一种编程语言。它和高级语言（如 BASIC、Java 和 C）的主要不同是

汇编语言提供了很少、简单的数据以及控制流。汇编语言程序不能指定一个变量中的数据类型。相反，编程人员必须对一个值使用恰当的操作（例如，整数或者浮点加法）。另外，在汇编语言中，程序的所有控制流必须使用 go to 实现。这两个因素使得汇编语言编程对于任何机器——MIPS 或者 x86——比使用高级语言编程更困难而且更容易出错。 [A-12]

**精解** 如果汇编器的速度比较重要，两步的过程可以采用反向修补（backpatching）技术一次遍历汇编文件来实现。在这一次遍历中，汇编器构建每个指令的一个（可能不完整的）二进制表示。如果指令引用了一个还没有定义的标签，汇编器在表中记录下这个标签和指令。当标签被定义后，汇编器查询这个表，找到包含对标签的所有前向（forward）引用的所有指令。汇编器回卷并校正它们的二进制表示，然后将它们并入标签地址。反向修补技术使加速汇编的原因在于：汇编器对输入只读一次。然而，它要求汇编器将程序的整个二进制表示保持在内存中，这样指令才可以被反复修补。这个需求会限制被汇编的程序大小。这个过程被那种具有几种类型的跨度范围不同的分支的机器复杂化了。当汇编器第一次在分支指令中见到没有处理的标签时，它必须要么使用最大的分支，要么冒险返回去，并且重新调整很多指令，以便为大的分支指令腾出位置。

⊃ 反向修补：一种将汇编语言翻译成机器指令的办法，其中汇编器在第一遍扫描程序时就构建一个（可能不完整的）每个指令的二进制表示，然后返回对前面没有定义的标签进行替换。

## A.2.1 目标文件的格式

汇编器产生目标文件。UNIX 上的目标文件包含 6 个不同的部分（见图 A-2-1）：

- 目标文件头描述了文件中其他段的大小和位置。
- **代码段**（text segment）包含了源文件中程序的机器语言代码。这些程序可能是不可执行的，因为包含了未处理的引用。
- **数据段**（data segment）包含了源文件中数据的二进制表示。数据可能是不完整的，因为未解决的引用可能包含在其他文件中。
- **重定位信息**（relocation information）指明指令和数据字依赖于**绝对地址**（absolute address）。如果程序的这些部分在内存中被移动，这些引用必须改变。
- 符号表中包含源文件中外部标签对应的地址，列出未处理的引用。
- 调试信息包含了被编译的程序的简洁描述，这样调试器可以找到源文件中对应行的指令地址，而且打印出可读形式的数据结构。

⊃ 代码段：UNIX 目标文件的一个段，包含源文件中程序的机器语言代码。

⊃ 数据段：UNIX 目标文件或者可执行文件的一个段，包含程序初始所使用的数据的二进制表示。

⊃ 重定位信息：UNIX 目标文件的一个段，根据绝对地址来区别数据字和指令。

⊃ 绝对地址：内存中变量或者程序的实际地址。




图 A-2-1 目标文件。UNIX 的汇编器产生一个具有 6 个不同段的目标文件

汇编器产生包含程序和数据的二进制表示的目标文件，以及其他有助于将程序的片段连接
A-13 起来的信息。

重定位信息是必要的，因为当一个程序片段或者数据块和程序剩余的部分链接后，汇编器不知道这些程序或者代码将会被存放到内存的什么位置。一个文件的程序和数据被保存在内存中的一个连续的区域，但是汇编器不知道这段内存如何定位。汇编器还会将一些符号表入口传递给链接器。尤其，汇编器必须记录哪个外部符号在一个文件中定义，这个文件中哪些引用没有解决。

**精解** 为方便起见，汇编器假设每个文件从相同的地址开始（例如，地址0），当它们在内存中分配地址时，期望链接器把代码和数据重新定位。汇编器产生重定位信息，这些信息包含一个入口，描述文件中的每个指令或数据字，这些字说明绝对地址。对于MIPS，仅仅子程序调用、装载和保存指令引用绝对地址。例如分支，使用指令PC相对寻址，不需要定位。

## A.2.2 附加工具

汇编器提供一类方便的特性帮助汇编程序变得短而且容易写，但是没有从根本上改变汇编语言。例如，数据布局指令允许一个编程者来描述以一种比二进制方式更简明和自然的方式来表示数据。

在图A-1-4中，指令

```
.asciiz "the sun from 0 .. 100 is %d\n"
```

在内存中保存字符串的字符。将这条指令和它的各个字符的ASCII值（这些字符的ASCII表示见第2章的图2-15）进行比较：

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

.asciiz描述更容易读懂，因为它使用字母表示字符，而不是使用二进制数字。汇
A-14 编器能够比人更快速且更准确地将字符转换成它们的二进制表示。数据布局指令指定一个人类可读的数据格式，汇编器将其转换成二进制。其他字符串指令布局指令在A.10节描述。

**例题** 使用这个指令定义一串字节：

```
.asciiz "the quick brown fox jumps over the lazy dog"
```

**答案**

```
.byte 84,  104, 101, 32,  113, 117, 105, 99
.byte 107, 32,  98,  114, 111, 119, 110, 32
.byte 102, 111, 120, 32,  106, 117, 109, 112
.byte 115, 32,  111, 118, 101, 114, 32,  116
.byte 104, 101, 32,  108, 97,  122, 121, 32
.byte 100, 111, 103, 0
```

□

宏（macro）是一种模式匹配和替换工具，提供一种简单的机制来命名一个经常使用的指令序列。不用每次使用同样的指令时重复输入，程序员只要启动宏，汇编器使用对应的指令序列替换这个宏调用。与子程序类似，宏允许程序员为一个公用操作产生和命名一个新的抽象。和子程序不同的是，宏不会导致一个子程序调用，也不会在程序运行时返回，因为一个宏的调

用会在程序汇编的时候被一个宏体替换。在替换完毕后，产生的汇编程序和没有使用宏的对等程序没有区别。

### 例题·宏

例如，假设程序员需要打印很多数字。一个库例程 printf 接受一个格式化的字符串，以及一个或更多个要打印的值作为其参数。程序员能够使用下面的指令打印出寄存器 $7 中的整数：

```
        .data
int_str: .asciiz"%d"
        .text
        la   $a0, int_str # Load string address
                          # into first arg
        mov  $a1, $7      # Load value into                    A-15
                          # second arg
        jal  printf       # Call the printf routine
```

.data 指令告诉汇编器将字符串保存到程序的数据段，而且 .text 指令告诉汇编器将指令保存到代码段。

然而，以这种方式打印很多数字（程序写起来）相当乏味，而且产生的冗长的程序让人很难读懂。一种可供选择的办法是引入宏，print_int，来打印一个整数：

```
        .data
int_str:.asciiz "%d"
        .text
        .macro print_int($arg)
        la  $a0, int_str # Load string address into
                         # first arg
        mov $a1, $arg    # Load macro's parameter
                         # ($arg) into second arg
        jal printf       # Call the printf routine
        .end_macro
print_int($7)
```

宏有一个**形式参数**（formal parameter）$arg，它是用来为宏的参数命名。当宏被展开时，贯穿宏体的形式参数被来自调用的参数替换。之后汇编器使用最新扩展的宏体替换这个宏调用。对于第一次 print_int 的调用，参数是 $7，所以宏扩展成以下代码：

```
la  $a0, int_str
mov $a1, $7
jal printf
```

➲ 形式参数：过程或者宏的参数变量，一旦这个变量被参数替换，宏就被展开。 A-16

在第二次调用 print_int 时，也就是说，print_int($t0)，参数是 $t0，宏被展开为：

```
la  $a0, int_str
mov $a1, $t0
jal printf
```

调用 print_int($a0)展开后的结果是什么？

### 答案

```
la  $a0, int_str
mov $a1, $a0
jal printf
```

这个例子暴露了宏的一个缺点。程序员使用宏必须意识到 print_int 使用寄存器 $a0，所以不能正确地打印那个寄存器的值。 □

**硬件/软件接口** 一些编译器也实现了伪指令（pseudoinstruction），这是汇编器提供的指令，但是在硬件上没有实现。第2章包含很多MIPS汇编器如何综合伪指令和寻址方式的例子，该寻址方式来自spartan MIPS硬件指令集。例如，第2章的2.7节描述了汇编器如何从其他两个指令（slt和bne）综合blt指令。通过扩展指令集，MIPS汇编器使得汇编语言编程更容易，而没有使硬件变得更复杂。很多伪指令能够使用宏来模拟，但是有这些指令，MIPS汇编器能产生更好的代码，因为它使用专用的寄存器（$at），能够优化产生的代码。

**精解** 汇编器有条件地将代码汇编起来，这允许当汇编程序时，编程者可以将一组指令包含进去，或者将一组指令剔除出去。当几个版本的程序在一定程度上不同时，这个特性尤其有用。不是将这些程序放在单独的文件中——这样会将通用代码中的固定错误（bug）复杂化——编程者通常将几个版本组合成一个文件。代码的一个特定的版本被有条件地汇编，以使综合程序的其他版本时，这部分代码可带除在外。

如果宏和条件汇编有用，为什么UNIX系统很少提供汇编器？一个原因是，在这些系统上很多编程者使用像C这样的高级语言编写程序。大部分汇编代码由编译器产生，编译器发现重复代码比定义宏更方便。另一个原因是，UNIX上的其他工具——例如，C的预处理器cpp，或者一个通用的宏处理器m4——能提供汇编程序的宏定义以及条件汇编。

A-17

## A.3 链接器

**单独编译**（separate compilation）允许程序被分割成多个片段，它们被保存在不同的文件中。每个文件包含一个逻辑相关的子程序以及数据结构组成的模块，这些文件形成一个大的程序。文件能够被编译，而且和其他的文件一样单独被汇编，所以一个模块的修改不需要重新编译整个程序。就像我们在上面讨论的，单独编译需要一个格外的链接步骤，以将单独的模块组成一个目标文件，将其未解决的引用解决。

➲ 单独编译：将程序划分成多个文件，每个文件被编译时，并不知道其他文件的信息。

将多个文件融合在一起的工具叫作链接器（linker）（见图A-3-1）。它执行三个任务：

- 为了寻找程序所使用的库程序而查询程序库。
- 为每个模块中的代码将要占用的内存确定内存地址，通过调整绝对引用，将这些指令重定位。
- 解决文件间的引用。

链接器的第一个任务是确保程序不包含没有定义的标签。链接器匹配外部的符号以及程序文件中未解决的引用。如果一个文件中外部符号和另一文件中的引用具有相同名字的标签，则未决的引用被确定。不匹配的引用意味着一个符号被使用，但是在程序的任何地方都没有定义。

在链接期间发现未解决的引用并不一定意味着程序员犯了错误。程序可能引用了一个库函数，该库函数的代码不在传递到链接器的目标代码中。在程序中匹配符号完毕后，链接器搜寻系统的程序库，目的是找程序中引用的预定义的子程序以及数据结构。基本库包含了读和写数据，分配和收回内存，执行数字操作。别的库包含访问数据库或者操作终端窗口。一个引用未解决符号的程序不在任何一个库中是错误的而且不能被链接。当程序使用了库例程，链接器从库中提取例程代码，并将其合并到程序的代码段。这个新的例程反过来可能依靠别的库例程，所以链接器继续读取别的库例程，直到没有外部引用是没有解决的或者没有哪个程序是不能被找到的。

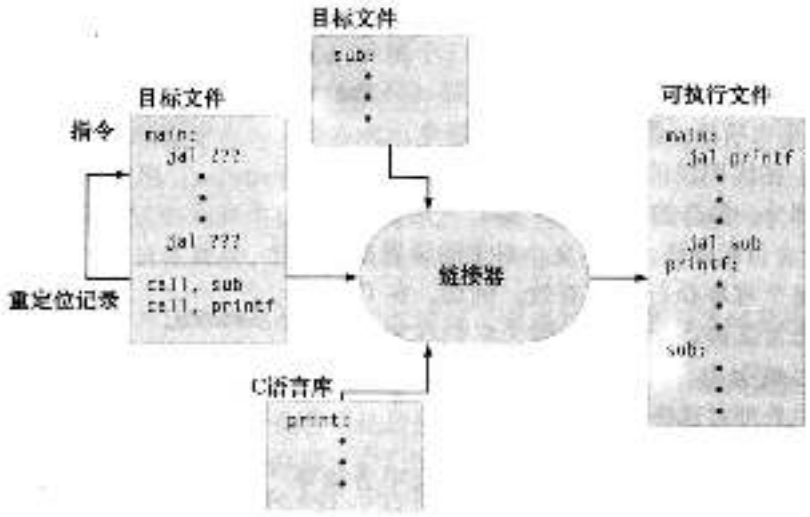



图 A-3-1 链接器搜查一组目标文件和程序库，寻找在程序中使用的非局部函数，将其合并成一个可执行文件，而且解决不同文件间的程序的引用

如果所有的外部引用被解决了，链接器接下来确定每个模块将占用的内存地址。因为文 A-18
件在汇编上是独立的，汇编器不知道一个模块的指令或者数据相对于别的模块放在哪里。当链接器在内存中放一个模块时，所有的绝对引用必须重定位以便反映其真实的地址。因为链接器具有重定位信息来指出所有重定位引用，它能够高效地找到以及反向修补这些引用。

链接器产生一个可执行的文件，这个文件可以在计算机上运行。典型的，除了不包含没有被解决的引用或者重定位信息，这个文件具有和目标文件一样的格式。

## A.4 加载

程序在链接阶段没有错误就可以运行。在运行之前，程序保存在像磁盘这样的二级存储的 A-19
一个文件中。在 UNIX 系统中，操作系统核心将一个程序加载到内存并且开始运行。为了启动一个程序，操作系统执行以下步骤：

1）读取可执行文件的头，目的是确定代码段和数据段的大小。

2）产生程序的一个新地址空间。这个地址空间足够大，装得下代码段和数据段，还有堆栈段（见 A.5 节）。

3）将可执行文件中的指令和数据复制到一个新的地址空间。

4）将传递给程序的参数复制到堆栈上。

5）初始化机器寄存器。通常，大部分寄存器被清零，但是堆栈寄存器指针必须被赋值为堆栈地址的初始地址（见 A.5 节）。

6）跳转到一个启动程序，这个程序从堆栈中把程序的参数复制到寄存器，而且调用程序的 main 程序。如果 main 程序返回，启动程序退出系统调用，终止程序的执行。

## A.5 内存的使用

下面几节描述本书前面提到的 MIPS 体系结构。前面几章主要关注硬件，以及硬件和低级软件的关系。这些章节主要关注汇编语言编程者如何使用 MIPS 硬件。描述在很多 MIPS 系统上的一组规则。很多情况下，硬件不会影响这些规则。相反，为了使不同的人编写的程序集合在

一起时能够工作，能有效地利用 MIPS 的硬件，这些规则代表了编程人员必须遵守的一种约定。

基于 MIPS 的系统通常将内存分割成三个部分（见图 A-5-1）。第一部分，接近地址空间的底部（开始地址是 $400000_{16}$），是代码段，保存的是程序的指令。

第二部分，在代码段的上面，称为数据段，它被进一步分割成两部分。**静态数据**（static data）（开始地址是 $10000000_{16}$）包含目标代码，它的大小对于编译器是已知的，其内容在整个程序执行期间有效。例如，在 C 语言中，全局变量通常是静态分配的，因为它们在程序执行的任何时候都可被引用。链接器既为静态的对象在数据段分配地址，也处理对这些对象的引用。

A-20

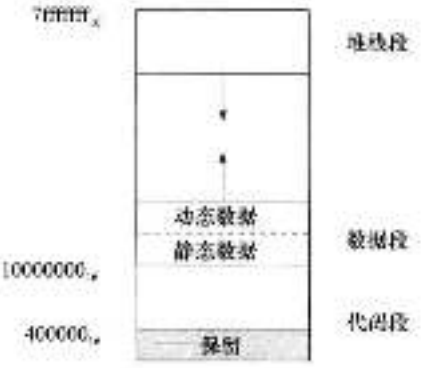



图 A-5-1 内存布局

⇨ 静态数据：包含数据的那部分内存，其大小为编译器所知，生命周期为整个程序的运行时间。

紧靠着静态数据之上的就是动态数据。这个数据，正如其名字所暗示的一样，是在程序执行过程中分配的。在 C 程序中，malloc 库例程发现并返回一个新的内存块。因为编译器不能预测一个程序将需要分配多大的内存，操作系统扩展了动态内存数据区域来满足这个需求。如图 A-5-1 中向上的箭头所指示的，malloc 通过使用系统调用 sbrk 扩展了动态区域。调用这个函数会导致操作系统在动态数据段之上为程序的虚拟地址空间加载更多的页（见第 5 章的 5.7 节）。

第三部分，**程序堆栈段**（stack segment）存在于虚拟地址空间的顶部（从地址 $7fffffff_{16}$ 开始）。和动态数据相似，一个程序的堆栈段的最大尺寸不能够被预先知道。当程序向堆栈段压入变量时，操作系统会自动向下（数据段方向）扩展堆栈段。

⇨ 堆栈段：程序用来保存过程调用项的那段内存。

这种三段分割的内存格局不是唯一的格局。然而，它具备两个重要的特性：动态可以扩展的段尽量隔得很远，而且能够扩展，以便将整个程序的地址空间全部用完。

**硬件/软件接口** 由于数据段的起始地址远远高于程序的起始地址 $10000000_{16}$，存取指令不能直接使用它们的 16 位偏移域引用数据对象（见第 2 章的 2.5 节）。例如，为了加载位于数据段地址 $10010020_{16}$ 处字到寄存器 $v0 需要两条指令：

```
lui  $s0, 0x1001 # 0x1001 means 1001 base 16
lw   $v0, 0x0020($s0) # 0x10010000 + 0x0020 = 0x10010020
```

（数字之前的 0x 表示这个数字是十六进制的值。例如，0x8000 是 $8000_{16}$ 或者 $32768_{10}$。）

为了在每个存取指令中避免重复 lui 指令，MIPS 系统通常使用一个专用的寄存器（$gp）作为全局指针指向静态数据段。这个寄存器包含了地址 $10008000_{16}$，所以存取指令可以使用 16 位的偏移来访问静态数据段的第一个 64KB。拥有这个全局指针，我们可以将以上例子改写为一条指令：

```
lw $v0, 0x8020($gp)
```

当然，一个全局指针寄存器使得寻址 $10000000_{16}$ – $10010000_{16}$ 比别的地址更为快。MIPS 编译器通常将全局变量存储在这个范围，因为这些变量具备固定的地址，而且比别的全局数据（例如数组）更合适。

## A.6 过程调用规范

当程序中的过程被分别编译时，管理寄存器使用的规范是必要的。为了编译一个特定过程，编译器必须知道需要哪些寄存器，以及哪些寄存器的信息需要为其他过程保留。寄存器使用的规则被称作**寄存器使用**（register use）或者**过程调用规范**（procedure call convention）。顾名思义，大多数情况下，这些规则主要用于约束软件，而不是硬件必须遵守的。然而，很多编译器以及编程者努力遵循这些规范，因为违反这些规则会导致诡异的错误。

⇒ *寄存器使用：或者称为过程调用规范。管理过程（调用）使用寄存器的软件协议。*

本节描述的调用规范是gcc编译器所遵循的一个规范。MIPS的原始编译器使用一个更为复杂的规范，而且这个规范会导致程序执行得比较快。

MIPS CPU包含32个通用目的寄存器，它们的编号是0～31。寄存器$0的值总是0。

- 寄存器$at(1)、$k0(26)和$k1(27)是预留给汇编器和操作系统的，不能被用户程序或者编译器使用。
- 寄存器$a0～$a3（4～7）被用来传递初始4个参数到例程（其他的参数传递到堆栈中）。寄存器$v0以及$v1（2，3）被用来返回来自函数的值。
- 寄存器$t0～$t9（8～15，24，25）是**调用者保存的寄存器**（caller-saved register），被用来保存临时变量，这些值在调用的时候不需要保存（见第2章2.8节）。

  ⇒ *调用者保存的寄存器：调用程序保存的寄存器。*

- 寄存器$s0～$s7(16～23)被称为**被调用者保存的寄存器**(callee-saved register)，保存长期存活值，这些值应当在程序调用时保存。

  ⇒ *被调用者保存的寄存器：被调用者程序保存的寄存器。*

- 寄存器$gp（28）是一个全局指针，指向64K的静态数据内存块。
- 寄存器$sp（29）是堆栈指针，指向堆栈的最后地址。寄存器$fp（30）是数据帧指针。jal指令写寄存器$ra(31)，是过程调用的返回地址。这两个寄存器将在下一节说明。

两个字母的缩写以及这些寄存器的名字——如$sp代表的是栈指针——反映了寄存器在过程调用规范中所起的作用。在描述这样一个规范时，我们将使用寄存器的名字，而不是寄存器的编号。图A-6-1罗列了这些寄存器及其用途。

### A.6.1 过程调用

本节描述一个程序（调用者，the caller）调用另一段程序（被调用者，the callee）的步骤。程序员使用像C或者Pascal这样的高级语言编程，从来都看不到一个程序调用另一个程序的细节。因为编译器负责低级的书签工作。然而，汇编语言程序员必须明确地实现每个程序调用和返回。

很多与调用相关的书签操作围绕着一个内存块，这个内存块被称为**过程调用帧**（procedure call frame）。这段内存被用作以下目的：

⇒ *过程调用帧：用来保存被调用过程的参数，保存可能会被过程修改的寄存器的值，但是这些寄存器的值不会被调用者所修改，并为被调用程序的局部变量提供空间。*

- 保持作为参数传递给过程的数值。
- 保存一个过程可能会修改的寄存器，但是过程的调用者却不希望这些寄存器的值被修改。

A-21
|
A-22

| 寄存器名称 | 编号 | 使用规则 | 寄存器名称 | 编号 | 使用规则 |
|---|---|---|---|---|---|
| $zero | 0 | 恒为0 | $s0 | 16 | 保存值可改（过程调用前备） |
| $at | 1 | 为汇编器保留 | $s1 | 17 | 保存值可改（过程调用前备） |
| $v0 | 2 | 表达式求值以及函数的结果 | $s2 | 18 | 保存值可改（过程调用前备） |
| $v1 | 3 | 表达式求值以及函数的结果 | $s3 | 19 | 保存值可改（过程调用前备） |
| $a0 | 4 | 参数1 | $s4 | 20 | 保存值可改（过程调用前备） |
| $a1 | 5 | 参数2 | $s5 | 21 | 保存值可改（过程调用前备） |
| $a2 | 6 | 参数3 | $s6 | 22 | 保存值可改（过程调用前备） |
| $a3 | 7 | 参数4 | $s7 | 23 | 保存值可改（过程调用前备） |
| $t0 | 8 | 临时（不为过程调用前备） | $t8 | 24 | 临时（不为过程调用前备） |
| $t1 | 9 | 临时（不为过程调用前备） | $t9 | 25 | 临时（不为过程调用前备） |
| $t2 | 10 | 临时（不为过程调用前备） | $k0 | 26 | 为OS内核保留 |
| $t3 | 11 | 临时（不为过程调用前备） | $k1 | 27 | 为OS内核保留 |
| $t4 | 12 | 临时（不为过程调用前备） | $gp | 28 | 全局区域的指针 |
| $t5 | 13 | 临时（不为过程调用前备） | $sp | 29 | 堆栈指针 |
| $t6 | 14 | 临时（不为过程调用前备） | $fp | 30 | 帧指针 |
| $t7 | 15 | 临时（不为过程调用前备） | $ra | 31 | 返回地址（过程调用使用） |

图 A-6-1 MIPS 寄存器和使用规则

- 为过程的局部变量提供空间。

在大部分编程语言中，过程调用和返回遵循一个严格的后进先出（LIFO）的顺序，所以在一个栈中内存能被分配以及被再次分配，这就是为什么这些内存块有时称作堆栈帧。

图 A-6-2 展示了一个典型的堆栈帧。这个帧由以下部分组成：帧指针（$fp），这个指针指向这个帧的第一个字；堆栈指针（$sp），指向帧的最后一个字。栈从内存的高地址开始向下增长，所以帧指针指向堆栈指针的上面。一个过程的执行使用帧指针来快速地访问堆栈帧中的值。例如，一个堆栈帧中的参数可以使用以下命令来加载到寄存器 $v0：

A-23

```
lw $v0, 20($fp)
```

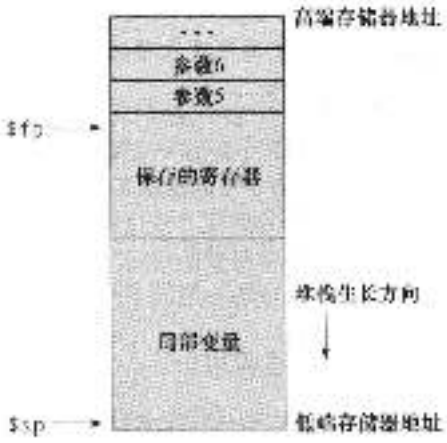



图 A-6-2 堆栈帧的示意图。帧指针（$fp）指向当前执行过程的堆栈帧的第一个字。堆栈指针（$sp）指向该帧的最后一个字。最前面4个参数被传送到寄存器中，所以第五个参数成为栈中第一个被保存的参数

堆栈帧可以有好几种不同的构建方式；然而，调用者和被调用者必须遵从一系列步骤。下面来描述在大部分 MIPS 机器上使用的调用规范步骤。这个规范在过程调用中的三个阶段出现：在调用者激活被调用者之前，被调用者开始执行，并且被调用者返回到调用者之前。在第一种情况下，调用者将过程调用参数放在指定的地方，激活被调用者做如下事情：

1）传递参数。根据规范，第一批的4个参数被传送到寄存器 $a0 - $a3。任何剩余的参数将被压入堆栈中，而且出现在被调用过程的堆栈帧的开始。

2）保存调用者寄存器。被调用过程可以直接使用这些寄存器（$a0 ~ $a3 以及 $t0 ~ $t9），而不需要首先保存这些寄存器的值。如果调用者在调用之后还想使用这些寄存器，那么它必须在调用之

前保存寄存器的值。

3）执行一个 jal 指令（见第 2 章的 2.8 节），这个指令跳转到被调用者的第一个指令并将返回地址保存到寄存器 $ra 中。

A-24 A-25

在一个被调用的例程开始运行之前，必须经过以下几步来建立它的堆栈帧：

1）为帧分配内存空间，通过栈指针来减小帧的大小。

2）在帧中保存被调用者的寄存器。调用者必须在修改这些寄存器之前保存这些寄存器的值（$s0 - $s7、$fp 和 $ra），由于调用者期望发现这些寄存器在调用之后保持不变。寄存器 $fp被每个过程保存，这个指针为过程分配一个新的堆栈帧。然而如果被调用者调用别的程序，寄存器 $ra 仅仅需要被调用者保存。别的被调用者保存的寄存器被使用的话，也必须被保存。

3）设置栈帧指针，其值为栈帧大小减去 4 加上 $sp，保存在寄存器 $fp 中。

**硬件/软件接口** MIPS 寄存器使用规范提供被调用者以及调用者保存的寄存器，因为这两种类型的寄存器在不同的环境中各具优势。被调用者保存的寄存器最好用来保存生存期长的值，例如来自用户程序的变量。如果被调用者期望使用这个寄存器，这个寄存器就仅仅在过程调用中被保存。另一方面，调用者保存的寄存器最好被用来保存短期存在的量，这些值在过程调用中不长期存在，例如地址计算中的立即值。在一个过程调用中，被调用者也可以使用那些保存临时值的寄存器。

最终，通过执行以下几步，被调用者返回到调用者：

1）被调用者是一个具备返回值的函数，就将返回值放到寄存器 $v0。

2）恢复所有被调用者保存的寄存器，这些寄存器保存前一个过程的入口。

3）向 $sp 加上帧大小，将帧从栈中弹出。

4）跳转到寄存器 $ra 中的地址处。

**精解** 编程语言不允许递归过程（recursive procedure）——一个过程通过一串调用，可以间接或者直接地调用自己——不需要在堆栈中分配帧。在一个非递归的语言中，每个过程的帧可能被静态分配，因为在同一时间，仅仅允许一个过程处于活动状态。旧版本的 Fortran 禁止递归，在一些比较老的机器中静态分配帧产生代码比较快。然而，在类似 MIPS 这样的存取体系结构中，堆栈帧的速度也可能很快。因为一个堆栈指针寄存器直接指向活动堆栈帧，这允许一个存取指令访问这个帧中的值。另外，递归是一种很有价值的编程技巧。

⊃ 递归过程：就是指某个过程能通过调用链直接或间接地调用自己。 A-26

## A.6.2 过程调用举例

作为一个例子，考虑以下 C 程序：

```
main ()
{
   printf ("The factorial of 10 is %d\n", fact (10));
}

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

这个函数计算而且打印10!（10的阶乘，10! $=10\times9\times\cdots\times1$）。fact是一个递归例程，计算n!，通过对n乘以(n-1)!。这段代码对应的汇编代码说明程序如何管理堆栈帧。

在入口上，例程main创建一个堆栈帧，而且保存被调用者将会修改的两个寄存器：$fp和$ra。一个帧的大小比两个寄存器大，因为调用规范所需要的一个堆栈帧的最小大小是24字节。最小的帧可以容纳4个寄存器参数（$a0～$a3）以及$ra的返回地址，再加上一个双字边界（一共24字节）。由于main函数也需要保存$fp，它的堆栈帧必须是两个字大小（记住：堆栈指针保持双字对齐）。

```
    .text
    .globl main
main:
    subu  $sp,$sp,32      # Stack frame is 32 bytes long
    sw    $ra,20($sp)     # Save return address
    sw    $fp,16($sp)     # Save old frame pointer
    addiu $fp,$sp,28      # Set up frame pointer
```

main程序然后调用阶乘例程而且将它的唯一参数10传给阶乘函数。在fact函数返回后，
A-27 main调用printf，而且给printf传递一个格式化字符串，以及从fact返回的结果两个参数。

```
    li    $a0,10          # Put argument (10) in $a0
    jal   fact            # Call factorial function

    la    $a0,$LC         # Put format string in $a0
    move  $a1,$v0         # Move fact result to $a1
    jal   printf          # Call the printf function
```

最终，在打印出阶乘结果后，main返回。但是首先，它必须恢复以下这些寄存器的值，将它们从栈中弹出：

```
    lw    $ra,20($sp)     # Restore return address
    lw    $fp,16($sp)     # Restore frame pointer
    addiu $sp,$sp,32      # Pop stack frame
    jr    $ra             # Return to caller

    .rdata
$LC:
    .ascii  "The factorial of 10 is %d\n\000"
```

阶乘函数的结构和main函数很相似。首先，阶乘函数创建一个堆栈帧，把它可能会使用的被调用者寄存器保存起来。另外，还保存$ra以及$fp。fact函数也保存它的参数（$a0），这个参数在递归调用的时候会被使用：

```
    .text
fact:
    subu  $sp,$sp,32      # Stack frame is 32 bytes long
    sw    $ra,20($sp)     # Save return address
    sw    $fp,16($sp)     # Save frame pointer
    addiu $fp,$sp,28      # Set up frame pointer
    sw    $a0,0($fp)      # Save argument (n)
```

fact例程的核心执行C程序计算。这个函数测试它的参数是否比0大。如果不是，例程返回值1。如果参数比0大，例程递归地调用它自己，计算fact(n-1)而且乘以n：

```
    lw    $v0,0($fp)      # Load n
    bgtz  $v0,$L2         # Branch if n > 0
    li    $v0,1           # Return 1
    jr    $L1             # Jump to code to return

$L2:
    lw    $v1,0($fp)      # Load n
    subu  $v0,$v1,1       # Compute n - 1
A-28 move  $a0,$v0         # Move value to $a0
```

```
    jal     fact            # Call factorial function

    lw      $v1,0($fp)      # Load n
    mul     $v0,$v0,$v1     # Compute fact(n-1) * n
```

最后，阶乘函数恢复被调用者保存的那些寄存器而且返回寄存器 $v0 中的值：

```
$L1:                        # Result is in $v0
    lw      $ra, 20($sp)    # Restore $ra
    lw      $fp, 16($sp)    # Restore $fp
    addiu   $sp, $sp, 32    # Pop stack
    jr      $ra             # Return to caller
```

**例题** 递归过程中的栈

图 A-6-3 展示了 fact(7)的调用栈。main 最先运行，所以它的帧在栈的最深处。main 调用了 fact(10)，它们的栈帧挨着。每个调用，递归调用 fact 函数来计算更低一级的阶乘。栈帧和这些函数的调用按照 LIFO 的顺序并行着。当 fact(10)返回的时候，栈看起来是什么样子？




图 A-6-3 调用 fact(7)过程中的栈帧

A-29

**答案**




□

**精解** MIPS 编译器和 gcc 编译器之间的差异是，MIPS 编译器通常不需要帧指针，所以这个寄存器可作为另一个被调用者保存寄存器 $s8 使用。这种改变在过程调用和返回序列中节省了一对指令。然而，这使得代码产生变得复杂，因为一个过程必须使用 $sp 来访问栈帧，如果有数值被压到栈中，它的值可以在一个过程执行中变化。

## A.6.3 另外一个过程调用的例子

作为另一个例子，考虑下面的程序，它计算 tak 函数，这是一个被广泛使用的基准测试程序，由 Ikuo Takeuchi 创建。这个函数不计算任何有用的东西，但这个函数是深度的递归程序，用它可以说明 MIPS 调用的规范。

```
int tak (int x, int y, int z)
{
    if (y < x)
        return 1+ tak (tak (x - 1, y, z),
                  tak (y - 1, z, x),
                  tak (z - 1, x, y));
    else
        return z;
}

int main ()
{
    tak(18, 12, 6);
}
```

这段程序的汇编代码将在下面展示。tak 函数首先保存它的返回地址到堆栈帧中，而且将

它的参数保存在被调用保存的寄存器中，因为例程或许会调用那些需要使用寄存器 $s0 - $s2
A-30 以及 $ra 的例程。函数使用被调用者保存的寄存器，由于它们在函数的整个生命期中保持有效，这期间包含几个可能会修改寄存器值的函数调用。

```
        .text
        .globl  tak

tak:
        subu    $sp, $sp, 40
        sw      $ra, 32($sp)

        sw      $s0, 16($sp)    # x
        move    $s0, $a0
        sw      $s1, 20($sp)    # y
        move    $s1, $a1
        sw      $s2, 24($sp)    # z
        move    $s2, $a2
        sw      $s3, 28($sp)    # temporary
```

通过测试，如果 y < x，例程开始执行。否则，分支转到标签 L1 处，如下所示。

```
        bge     $s1, $s0, L1    # if (y < x)
```

如果 y < x，那它就执行例程的主体，主体包含了 4 个递归的调用。第一个调用使用几乎和它的母体相同的参数：

```
        addiu   $a0, $s0, -1
        move    $a1, $s1
        move    $a2, $s2
        jal     tak             # tak (x - 1, y, z)
        move    $s3, $v0
```

注意到，第一个递归调用的结果被保存到寄存器 $s3，这样便于不久后使用。

函数现在为第二个递归调用做准备。

```
        addiu   $a0, $s1, -1
        move    $a1, $s2
        move    $a2, $s0
        jal     tak             # tak (y - 1, z, x)
```

在下面的指令中，来自递归调用的结果被保存到寄存器 $s0。但是首先，也是最后一次，
A-31 我们需要读这个寄存器的值，其中保存第一个参数的值。

```
        addiu   $a0, $s2, -1
        move    $a1, $s0
        move    $a2, $s1
        move    $s0, $v0
        jal     tak             # tak (z - 1, x, y)
```

在三个内部递归调用之后，我们准备最后的递归调用。调用之后，函数的结果保存到 $v0 中，控制函数流程的跳转。

```
        move    $s1, $s3
        move    $s2, $s0
        move    $s3, $v0
        jal     tak             # tak (tak(...), tak(...), tak(...))
        addiu   $v0, $v0, 1
        j       L2
```

标签 L1 处的代码是一个 if-then-else 语句序列。它仅仅将参数 z 的值传递到返回寄存器而落入到函数格局中。

```
L1:
        move    $v0, $s2
```

以下的代码是函数末尾，它恢复被保存的寄存器并将函数值返回给它的调用者。

```
L2:
    lw      $ra, 32($sp)
    lw      $s0, 16($sp)
    lw      $s1, 20($sp)
    lw      $s2, 24($sp)
    lw      $s3, 28($sp)
    addiu   $sp, $sp, 40
    jr      $ra
```

main 函数使用最初的参数来调用 tak 函数，然后得到计算结果 result（7）而且使用 SPIM 系统调用来打印整数的值。

```
    .globl  main
main:
    subu    $sp, $sp, 24
    sw      $ra, 16($sp)

    li      $a0, 18
    li      $a1, 12
    li      $a2, 6
    jal     tak             # tak(18, 12, 6)

    move    $a0, $v0
    li      $v0, 1          # print_int syscall
    syscall

    lw      $ra, 16($sp)
    addiu   $sp, $sp, 24
    jr      $ra
```

A-32

## A.7 异常和中断

第4章的4.9节描述了MIPS异常机制，包括指令执行中发出错误导致的异常以及I/O设备引起的外部中断。本节描述异常以及中断处理（interrupt handling）的更多细节。[⊖]在MIPS处理器中，CPU中一个被称为 coprocessor 0 的部分记录软件处理异常和中断所需要的信息。MIPS 仿真器 SPIM 没有实现 coprocessor 0 的寄存器，因为在一个仿真器中，不需要很多寄存器，或者寄存器不是内存系统的一个部分，SPIM 就没有实现它。然而，SPIM 确实提供了下列 coprocessor 0 的寄存器：

A-33

- 中断处理：一段代码，作为异常或者中断的执行结果。

| 寄存器名称 | 寄存器编号 | 使用 |
|---|---|---|
| BadVAddr | 8 | 一个会发生内存引用冲突的内存地址 |
| Count | 9 | 计时器 |
| Compare | 11 | 一个用来和计时器进行比较，当它的值和计时器匹配时，会发生中断 |
| Status | 12 | 中断掩码以及使能位 |
| Cause | 13 | 异常类型以及中断挂起位 |
| EPC | 14 | 引起异常的指令地址 |
| Config | 16 | 机器的配置 |

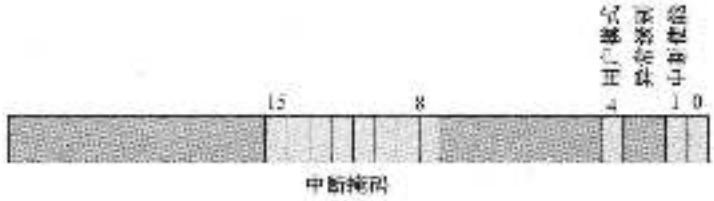
⊖ 本节讨论 MIPS-32 体系结构中的异常，这些异常是 SPIM 的7.0及以后的版本所实现的。SPIM 的早期版本实现了 MIPS-I 体系结构，但是其中对异常的处理有些不同。将这些不同版本的程序转变成在 MIPS-32 上运行不是一件困难的事情，因为只有状态和原因寄存器的域需要改变，使用 rfe 指令替换 eret 指令。

这7个寄存器是coprocessor 0处理器状寄存器组的一部分。它们通过mfc0以及mtc0指令来访问。异常之后，寄存器EPC包含了在执行时发生异常的那条指令的地址。如果异常是外部中断引起的，那么指令将不需要重新开始执行。除了导致问题的指令处于分支或跳转指令的延迟槽中之外，所有其他的异常均由执行EPC处的指令引起。在那种情况下，EPC指向分支或者跳转指令而且原因寄存器中的BD位被设置。当这些位设置好后，异常处理函数必须查看引起异常的EPC+4。然而，在别的情况下，异常处理函数通过返回到指令的EPC地址处恢复被中断的程序。

如果指令所引起的异常导致一个内存访问，寄存器BadVAddr包含被引用的内存地址的地址。

Count寄存器是一个计数器，当SPIM运行时，它按照一定的频率递增（默认，每10毫秒一次）。当Count寄存器中的值和比较寄存器中的值匹配时，处于5级的硬件中断就会发生。

图A-7-1展示了MIPS仿真器SPIM实现的状态寄存器域。interrupt mask域为6个硬件包含了6个位和2个软件中断层次。如果掩码位的值是1，就是允许处理器这个级别上的中断；如果掩码位的值是0，就是不允许处理器这个级别上的中断。当中断到达时，中断在其原因寄存器中设置中断挂起位，即使掩码位是无效的。当一个中断被挂起时，当随后其掩码位被允许时，它会中断处理器。




图A-7-1 状态寄存器

当处理器运行在核心模式，用户模式位是0；如果用户模式位是1，则说明处理器处于用户态。对于SPIM，这一位固定是1，因为SPIM处理器没有实现核心模式。异常级别位通常是0，但是当异常发生时就被设置成1。当这一位是1时，中断被禁止，而且如果另一个异常发生，EPC也不会被更新。这一位阻止一个异常处理被别的中断或者异常打断，但是它应当在异常处理结束时可以复位。如果interrupt enable位是1，中断就被允许。如果这一位是0，它们就被禁止。

图A-7-2展示了SPIM中的原因寄存器字段的子集。如果最后一个异常发生时，一个正在执行的指令处于分支延迟槽中，分支延迟位则为1。当一个中断处于给定的硬件或者软件层 A-34 时，中断挂起位变成1。异常代码寄存器通过以下代码描述了一个异常的原因：

| 编号 | 名称 | 异常产生的原因 |
|---|---|---|
| 0 | Int | 中断（硬件） |
| 4 | AdEL | 地址错误异常（加载或者取指令） |
| 5 | AdES | 地址错误异常（存储） |
| 6 | IBE | 取指令总线错误 |
| 7 | DBE | 加载数据或者存储数据总线错误 |
| 8 | Sys | 系统调用异常 |
| 9 | Bp | 断点异常 |
| 10 | RI | 保留指令异常 |
| 11 | CpU | 没有实现的协处理器 |
| 12 | Ov | 算术溢出异常 |
| 13 | Tr | 陷阱 |
| 15 | FPE | 浮点 |




图 A.7.2 原因寄存器

异常和中断导致 MIPS 处理器跳转到一段代码，地址是 80000180$_{hex}$（在核心态，而不是在用户的地址空间），被称作异常处理代码。这个代码检查异常的原因，而且跳转到操作系统的一个合适的点。操作系统对一个异常会做出以下的响应：结束一个引起异常的进程或者执行一些动作。进程所引起的错误，例如执行一个没有被实现的指令，就会被操作系统终止。另一方面，别的异常，例如送给操作系统的来自进程的缺页错误就是要执行一个服务，即从磁盘取回 A-35
一个页。操作系统处理这些请求，然后恢复这个发出请求的进程。最后一种类型的异常是外部设备发出的中断。这些通常会导致操作系统将数据搬运到 I/O，或者从 I/O 把数据搬运回来，然后恢复被中断的进程。

下面例子中的代码是一个简单异常处理程序，它启动一个为每个异常打印消息的程序（但不是中断）。这个代码与 SPIM 仿真器使用的异常处理（exceptions.s）相似。

**例题** 异常处理

异常处理程序首先保存寄存器 $at，这个符号在处理程序代码的伪代码中被使用，然后保存 $a0 和 $a1，这两个值之后将被用来传递参数。异常处理程序不能在堆栈中保存这些寄存器的旧值，作为一个一般的程序，因为异常产生的原因可能是一个内存引用在堆栈指针中使用了一个坏的值（如0）。相反，异常处理程序在一个异常处理寄存器（$k1，因为不使用 $at，它不能访问内存）以及两个内存地址（save0 和 save1）中保存这些寄存器的值。如果异常处理程序本身可以被中断，两个地址可能不充分，因为第二个异常可能会改写这些第一个异常保存的值。然而，在允许中断之前，这个简单的异常处理程序结束运行，所以这些问题不会出现。

```
.ktext 0x80000180
mov $k1, $at    # Save $at register
sw  $a0, save0  # Handler is not re-entrant and can't use
sw  $a1, save1  # stack to save $a0, $a1
                # Don't need to save $k0/$k1
```

异常处理程序然后将原因寄存器和 EPC 寄存器保存到 CPU 的寄存器中。原因寄存器和 EPC 寄存器不是 CPU 寄存器组的一个部分。相反，它们是协处理器 0 的寄存器，协处理器是 CPU 处理异常的一个部分。指令 mfc0 $k0，$13 将协处理器 0 的寄存器 13（原因寄存器）保存到 CPU 的寄存器 $k0。注意到异常处理不需要保存这些寄存器 $k0 和 $k1，因为用户程序不被认为会使用这些寄存器。异常处理程序使用来自原因寄存器的值来测试异常是否被一个中断所引起（参见前面的表）。如果是这样，则异常就会被忽略。如果异常不是中断，程序就会调用 print_excp 来打印一条信息。 A-36

```
mfc0    $k0, $13        # Move Cause into $k0

srl     $a0, $k0, 2     # Extract ExcCode field
andi    $a0, $a0, 0xf

bgtz    $a0, done       # Branch if ExcCode is Int (0)

mov     $a0, $k0        # Move Cause into $a0
mfc0    $a1, $14        # Move EPC into $a1
jal     print_excp      # Print exception error message
```

在返回之前，异常处理程序清除原因寄存器；重设状态寄存器，目的是为了使得中断允许。而且清除 EXL 位，这使得后续的异常来修改 EPC 寄存器；而且恢复寄存器 $a0、$a1 和 $at。然后执行 eret 指令（异常返回），这个指令返回 EPC 所指向的指令。这个异常处理程序返回到引起异常的指令的后面的那条指令。所以不会重新执行那条错误的指令，而且不会再次引起异常。

```
done:   mfc0    $k0, $14        # Jump EPC
        addiu   $k0, $k0, 4     # Do not re-execute
                                # faulting instruction
        mtc0    $k0, $14        # EPC

        mtc0    $0, $13         # Clear Cause register

        mfc0    $k0, $12        # Fix Status register
        andi    $k0, 0xfffd     # Clear EXL bit
        ori     $k0, 0x1        # Enable interrupts
        mtc0    $k0, $12

        lw      $a0, save0      # Restore registers
        lw      $a1, save1
        mov     $at, $k1

        eret                    # Return to EPC

        .kdata
save0:  .word 0
save1:  .word 0
```

A-37

**精解** 在一个实际的 MIPS 处理器中，从异常处理程序返回的过程相当复杂。异常处理程序不能经常挑选到 EPC 的下一条指令。例如，如果引起异常的指令处于分支指令的延迟槽中（见第 4 章），下一条被执行的指令可能就不是内存中的下一条指令。

## A.8 输入和输出

SPIM 仿真一个 I/O 设备：一个内存映射的控制台，在这个控制台中可以读和写字符。当一个程序正在运行时，SPIM 将其终端（一个独立的控制台窗口，是 X-window 版本的 xspim 或者 Windows 版本的 PCSpim）连接到处理器上。运行在 SPIM 上的一个 MIPS 程序可以读取你输入的字符。另外，如果 MIPS 程序可以在终端写字符，字符将出现在 SPIM 的终端或者控制台窗口。这个规则的一个异常是 control-C：这个字符没有被传递到程序中，但是导致 SPIM 停止，而且返回到命令行模式。当程序停止运行（例如，因为你输入 control-C，或者因为程序遇到一个断点）时，终端被重新连接到 SPIM，这样你就可以输入 SPIM 指令了。

为了使用内存映射的 I/O（见下面），spim 或者 xspim 必须使用 -mapped_io 标志来启动。PCSpim 通过一个命令行标志可以允许内存映射的 I/O。或者通过“设置”对话框来实现。

终端设备由两个独立的单元组成：一个“接收者”和一个“发送者”。接收者读来自键盘的字符。发送者在终端（控制台）显示字符。两个单元完全独立。这意味着，例如，从键盘输入的字符不能自动重复显示。相反，一个程序通过从接收者那里读一个字符，而且将其写到发送者那里。

一个程序控制着一个具有 4 个内存映射的设备寄存器的终端，如图 A-8-1 所示。“内存映射”意味着每个寄存器作为一个特殊的内存地址。一个接收者控制寄存器是在地址 $1000000_{16}$，实际仅用到它的两个位。位 0 称作“ready”（预备）；如果它是 1，这意味着一个字符从键盘到

达，但是还没有从数据接收器的寄存器中读出。ready 位是只读的；对它的写操作会被忽略。当字符从键盘输入时，ready 位从 0 转换到 1，而且当字符从接收器数据寄存器读取时，ready 位从 1 转换到 0。 A-38

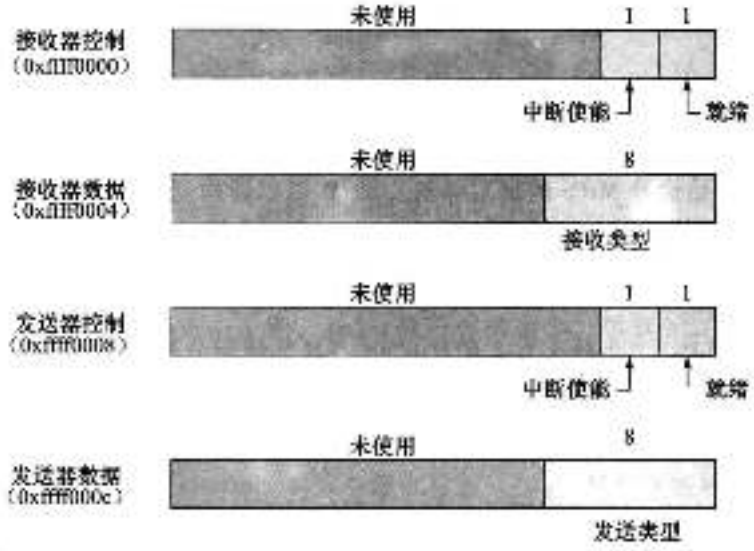



图 A-8-1 终端被 4 个设备寄存器控制着，每个寄存器在给定地址作为一个内存地址。仅仅这些寄存器的一些位实际上被使用。别的位通常被读成 0，对它们的写被忽略

接收器的控制寄存器的第 1 位是键盘“中断允许位”。这个位可以被程序读，也可以被写。中断允许位初始为 0。如果它被程序设置成 1，无论何时字符被输入，终端在硬件级 1 请求一个中断。然而，中断影响到处理器，中断必须在状态寄存器中设置成允许（见 A.7 节）。接收器的控制寄存器别的位都没有被使用。

第二个终端设备寄存器是接收器数据寄存器（在地址 $ffff0004_{16}$）。这个寄存器的低 8 位包含着从键盘输入的最后字符，别的位都是 0。这个寄存器是只读而且仅当一个新的字符从键盘被输入时才改变。读取接收器数据寄存器导致接收器控制寄存器的 ready 位复位成 0。如果接收器控制寄存器是 0，这个寄存器中的值没有被定义。

第三个终端设备寄存器是发送者控制寄存器（在地址 $ffff0008_{16}$）。当这个寄存器只有低两位被使用。它们的行为和接收器控制寄存器很相似。第 0 位称作“ready”，而且是只 A-39
读的。如果这个位是 1，发送者准备为输出接受一个新的字符；如果是 0，发送者将仍然忙于写前一个字符。第 1 位是“终端允许位”，而且是可读可写的。如果这个位被设置成 1，无论发送者是否准备好一个新的字符，终端在硬件 0 级请求一个中断，然后 ready 位变成 1。

最后一个设备寄存器是发送者数据寄存器（在地址 $ffff000c_{16}$）。当一个值被写入这个地址处时，它的低 8 位（例如，第 2 章的图 2-15 中的一个 ASCII 字符）被发送到控制台。当发送者数据寄存器被写时，发送者控制寄存器的 ready 位被设置成 0。这个位保持为 0，直到字符发送到终端花费了足够的时间；然后 ready 位又一次变成 1。发送者数据寄存器应当仅仅当发送者控制寄存器的 ready 位是 1 的时候可以被写入。如果发送者没有准备好，写入到发送者数据寄存器的数据会被忽略（写入会成功，但是字符没有输出）。

实际的计算机需要时间来将字符发送到控制台或者终端。这些时间延迟会被 SPIM 仿真器模拟。例如，在发送者开始写一个字符之后，发送者的 ready 位之后不久就变成 0。SPIM 按照

指令的执行来测量时间，而不是按照实际的时间。这就意味着发送者不会变成 ready，直到处理器执行一个固定数量的指令。如果你停止机器，而且查看其 ready 位，它是不会改变的。然而，如果你让机器运行，这个位最终会变成 1。

## A.9 SPIM

SPIM 是一个软件仿真器，它运行为处理器编写的汇编语言程序，实现 MIPS-32 体系结构，尤其体系结构的版本 1 具备固定的内存映射，没有 cache，而且仅仅有协处理器 0 和协处理器 1。[⊖]SPIM 的名字恰恰是 MIPS 的倒写拼法。SPIM 可以读写且可以立即执行汇编语言文件。
A-40 SPIM 是一个自含的系统，用于运行 MIPS 程序。它包含一个调试器，并提供一些类似操作系统的服务。SPIM 比实际的计算机要慢得多（100 倍或更多）。然而，它代价小，可用性广泛，是真实硬件所无法比拟的。

一个明显的问题是，“为什么在人们拥有 PC，且其使用的处理器比 SPIM 运行快得多时，却还使用仿真器？”原因之一是这些 PC 中的处理器是 Intel 的 80x86，它们的结构不太规则，而且复杂，难以理解，比 MIPS 处理器编程困难。MIPS 结构可能是一个简单、整洁的 RISC 机器的缩影。

另外，仿真器能够为汇编编程提供一个比实际机器更好的环境，因为它们能检测出更多的错误，提供一个比实际计算机更好的接口。

最后，仿真器是研究计算机和在其上运行的程序的有用工具。因为它们是以软件方式实现的，而不是硬。所以对于添加新指令，构建像多处理器这样的新系统，或者收集数据这样简单的事情，使用仿真器容易验证也容易修改。

### A.9.1 虚拟机的仿真

由于延迟分支、延迟加载、受限制的地址模式等原因，基本的 MIPS 体系结构很难直接编程。这个困难是可忍受的，因为这些计算机被设计用来使用高级语言编程，所以是给编译器，而不是给汇编语言程序员提供接口。编程复杂度的很大一部分是由延迟指令导致的。一个延迟的分支需要两个周期来执行（见第 4 章的 4.5 节和 4.8 节的精解）。在第二个周期，执行那条紧跟分支指令的指令。这条指令能执行有用的工作，而正常情况下该工作可能在分支指令之前已经完成。它可能是没有任何操作的 nop 指令。同样，延迟加载需要两个周期来将一个值从内存中取回，所以紧跟其后的指令无法使用这个值（见第 4 章的 4.2 节）。

MIPS 一般通过汇编语言实现的虚拟机（virtual machine）隐藏复杂度。虚拟机没有延迟的分支、加载，而且具有比实际硬件更丰富的指令集。汇编器将这些指令重新组织（分派）到延迟槽中。虚拟机也提供伪指令，这些指令看起来和汇编语言程序中的真实指令一样。然而，硬件完全不知道这些伪指令，所以汇编器必须将其翻译成实际机器指令的等价序列。例如，当一个寄存器等于 0 或者不等于 0 的时候，MIPS 硬件仅仅提供分支指令。对于其他的条件分支，例如那种当一个寄存器的值比另一个寄存器的值大时就进行分支的类型，分支指令会被综合成
A-41 两个寄存器的比较，然后当其比较的结果是真（非零）时就执行分支。

⇒ 虚拟机：一种虚拟计算机，它分支和取数指令没有延迟，且指令集比实际硬件更丰富。

---

[⊖] SPIM 的早期版本（7.0 以前的版本）实现了 MIPS-1 的体系结构，使用庞杂的 MIPS R2000 处理器。这个体系结构几乎是 MIPS-32 体系结构的子集，不同在于异常处理的方式。MIPS-32 也引入了将近 60 个新指令，SPIM 支持这些指令。程序可以在 SPIM 的早期版本中运行，而且不使用异常的程序可以不加修改在新版本 SPIM 上运行。使用异常的程序将需要少许的修改。

默认情况下，SPIM 模拟指令集更丰富的虚拟机，因为这是一个对很多程序员来说很有用的机器。然而，SPIM 也能模拟实际的硬件的延迟分支以及延迟取数操作。下面，我们描述虚拟机并且仅提及其和实际的硬件没有关系的特性。这样做，我们遵循了 MIPS 的汇编程序员（汇编器）的规程，他们将扩展的机器当成是由硬实现的机器那样使用。

## A.9.2 从 SPIM 开始

本附录的剩余的部分介绍 SPIM 和 MIPS R2000 汇编语言。你不用关注过多的细节，然而，大量的信息很多时候会模糊以下事实：SPIM 是个简单易用的程序。本节我们先以 SPIM 的快速使用教程开始，教会你加载、调试、运行简单的 MIPS 程序。

对于不同类型的计算机系统，SPIM 有几个不同的版本。其中一个经久不变的，是最简单的版本，称为 spim，它是运行在控制窗口下的一个命令行驱动程序。它和很多控制台程序一样操作：输入一行文本，按 return 键，spim 执行你的命令。尽管 spim 缺乏精美的界面，但它可以做具有精美界面的同类版本可以做的任何事情。

spim 拥有两个界面精美的版本。运行在 UNIX 或者 Linux 系统上的 X-windows 环境下的版本称为 xspim。与 spim 相比，xspim 更易于学习和使用，因为它的指令总是在屏幕上可见的，且持续显示机器的寄存器和内存。另一个版本是 PCspim，运行在微软的 Windows 系统下。SPIM 的 UNIX 和 Windows 版本都在本书配套网站上。xspim、pcSpim、spim 的教程和 SPIM 命令行选项都在网站上。

如果你打算在运行微软 Windows 系统的 PC 上运行 SPIM，你应当先阅读网站上的 PCSpim 教程。如果你打算在运行 UNIX 或 Linux 的 PC 上运行 SPIM，你应当阅读网站上的 xspim 教程。

## A.9.3 令人惊讶的特性

尽管如实地仿真了 MIPS 计算机，但 SPIM 作为一个仿真器，和实际计算机必定是不相同的。最明显的区别是，指令的时序和内存系统不同。SPIM 不模拟 cache 或者存储器的延迟，也不会精确反映浮点操作、乘法或者除法指令的延迟。另外，浮点指令不检测错误条件，而这在实际机器上将导致异常。 A-42

另一个令人惊讶的特性（这种情形在真实机器上也会发生）是将伪指令扩展成多条机器指令。当你单步调试或者检查存储器，你所看到的指令和原始的程序不相同。两组指令之间的对应性相当简单，因为 SPIM 并没有为了填充延迟槽而重组指令。

## A.9.4 字节顺序

处理器能对字中的字节进行编号，这样编号最小的字节不是在最左边就是在最右边。机器使用的该约定称为字节顺序。MIPS 处理器可以依大端字节顺序或者小端字节顺序进行操作。例如，在大端机器下，指令 byte 0,1,2,3 将引起一个内存字包含：

| 字节号 | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |

但是在小端机器下，一个字可能包含：

| 字节号 | | | |
|---|---|---|---|
| 3 | 2 | 1 | 0 |

SPIM 以两种字节顺序操作。SPIM 的字节顺序和运行仿真器的底层机器的字节顺序是一样的。例如，在 Intel 80x86 处理器上，SPIM 是小端，然而在 Macintosh 或者 Sun SPARC 处理器上，SPIM 是大端。

### A.9.5 系统调用

SPIM 通过系统调用（syscall）指令提供了一小组类似操作系统的服务。为了请求一个服务，一个程序加载系统调用代码（见图 A-9-1）到寄存器 $v0，将参数加载到寄存器 $a0 ~ $a3（或用于浮点值的 $f12）。系统调用将返回值放到 $v0（或用于浮点值的 $f0）。例如，下面的代码将打印"the answer = 5"：

```
        .data
str:
        .asciiz "the answer = "
        .text
        li      $v0, 4      # system call code for print_str
        la      $a0, str    # address of string to print
        syscall             # print the string

        li      $v0, 1      # system call code for print_int
        li      $a0, 5      # integer to print
        syscall             # print it
```

A-43

| 服务 | 系统调用代码 | 参数 | 结果 |
|---|---|---|---|
| print_int | 1 | $a0 = integer | |
| print_float | 2 | $f12 = float | |
| print_double | 3 | $f12 = double | |
| print_string | 4 | $a0 = string | |
| read_int | 5 | | integer (in $v0) |
| read_float | 6 | | float (in $f0) |
| read_double | 7 | | double (in $f0) |
| read_string | 8 | $a0 = buffer, $a1 = length | |
| sbrk | 9 | $a0 = amount | address (in $v0) |
| exit | 10 | | |
| print_char | 11 | $a0 = char | |
| read_char | 12 | | char (in $v0) |
| open | 13 | $a0 = filename (string),<br>$a1 = flags, $a2 = mode | file descriptor (in $a0) |
| read | 14 | $a0 = file descriptor,<br>$a1 = buffer, $a2 = length | num chars read (in $a0) |
| write | 15 | $a0 = file descriptor,<br>$a1 = buffer, $a2 = length | num chars written (in $a0) |
| close | 16 | $a0 = file descriptor | |
| exit2 | 17 | $a0 = result | |

图 A-9-1 系统服务

给 print_int 系统调用传递一个整数并在终端上打印出来。print_float 打印一个单精度浮点数；print_double 打印出来一个双精度数；而给 print_string 传递一个指向空的终止数据串的指针，这个字符串写到终端上。

系统调用 read_int、read_float 以及 read_double 来读取一个完整的输入行，并包

含一个新行。数字后面的字符串将被忽略。read_string 具有和 UNIX 库例程 fgets 相同的语义。它将读取的 $n-1$ 个字符存到缓冲区，并使用一个空字节作为结束符。如果当前行中的字符数少于 $n-1$ 个，read_string 将读取到新行，并再次使用一个空字节作为结束符。 A-44

警告：使用系统调用从终端读取数据的程序不应当使用内存映射的 I/O（见 A.8 节）。

sbrk 返回一个指向包含 $n$ 个额外字节块的存储器指针。exit 可以终止 SPIM 正在执行的程序。exit2 终止 SPIM 程序，并且当 SPIM 仿真器终止时，传递给 exit2 的参数将变成返回值。

print_char 和 read_char 分别读和写单个字符。open、read、write 和 close 是 UNIX 的标准库调用。

## A.10 MIPS R2000 汇编语言

MIPS 处理器由整型处理单元（CPU）和一系列协处理器（用于执行辅助工作或诸如浮点等其他数据类型的操作）组成（见图 A-10-1）。SPIM 可模拟两个协处理器。协处理器 0 用于处理异常和中断。协处理器 1 是浮点运算单元。SPIM 模拟本单元的大多数功能。

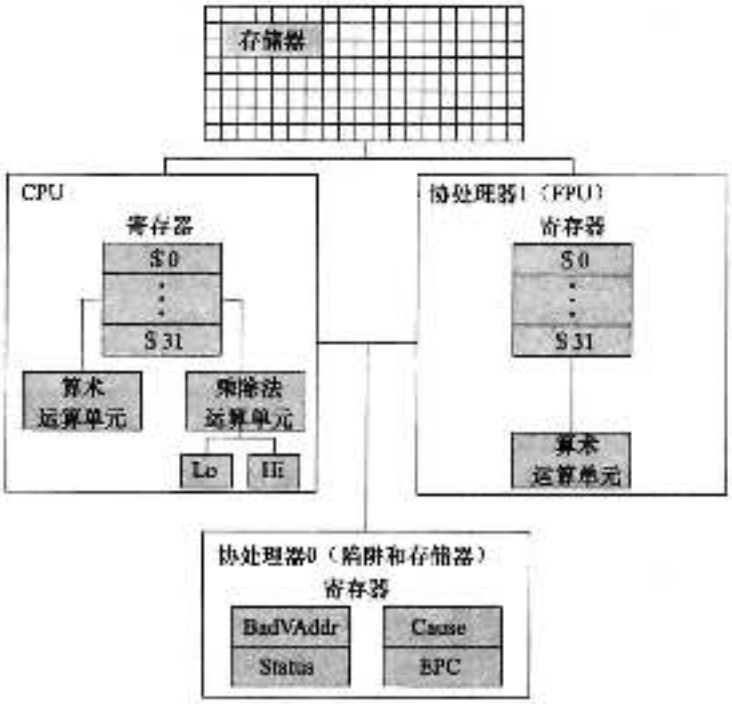



图 A-10-1 MIPS R2000 CPU 和 FPU

### A.10.1 寻址模式

MIPS 采取加载和存储体系结构，也就是说，只有加载和存储指令访问存储器。计算指令只对寄存器中的值进行处理。机器本身只提供一种存储器寻址模式：c(rx)。它把立即数 c 和寄存器 rx 的值相加作为地址。虚拟机则为加载和存储指令提供了以下几种寻址方式：

| 格式 | 地址计算 |
|---|---|
| (寄存器) | 寄存器内容 |
| 立即数 | 立即数 |
| 立即数 (寄存器) | 立即数 + 寄存器内容 |

（续）

| 格式 | 地址计算 |
|---|---|
| 标识 | 地址标识 |
| 标识 + 立即数 | 地址标识 + 或 - 立即数 |
| 标识 ± 立即数（寄存器） | 地址标识 + 或 - （立即数 + 寄存器内容） |

大多数加载和存储指令只对对齐数据进行处理。以字节为单位，当数据所在的存储器地址是其大小的整数倍时，我们就说数据是对齐的。因此，半字对象必须存放在偶地址，而全字对象必须存放在4的整数倍的地址。但是，MIPS还提供了另外一些指令，可以对非对齐数进行操作（如lwl，lwr，swl和swr）。 A-45

**精解** MIPS汇编器（SPIM也一样）通过在对数据引用之前产生一条或多条指令来计算复杂的地址，因此它也支持一些复杂的寻址模式。例如，假设标识table指向存储器地址0x10000004，并且程序包含一条这样的指令：

```
ld $a0, table + 4($a1)
```

汇编器会将这条指令转化为下面三条指令：

```
lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)
```

第一条将标记地址的高位送入寄存器$at（该寄存器是汇编器为自己保留的）。第二条将寄存器$a1的内容加到标识的局部地址上。最后，加载指令用硬件寻址模式将标识地址的低位和寄存器$at中相对原始指令的偏移量相加。

## A.10.2 汇编语法

汇编文件中的注释行以“#”开始。所有以“#”开头的指令行都会被忽略。

标识符由字母、数字、下划线（_）和点（.）构成，但不能以数字开头。指令操作码是一些保留字，不能用作标识符。标识是这样表示的：将其放在行首，后跟冒号（:）。例如：

```
        .data
item:   .word 1
        .text
        .globl main         # Must be global
main:   la      $t0, item
```

数值默认是十进制。如果数值以0x开始，则表明它们是十六进制数。因此，256和0x100所表示的数值是相同的。

字符串用双引号（"）括起来。字符串中的特殊字符遵从C语言规范：

- 换行 \n
- 制表 \t
- 引号 \"

SPIM还支持一些MIPS汇编指令：

| | |
|---|---|
| .align n | 将数据以$2^n$字节分界。例如，.align 2将数据以字为单位分界；.align 0关闭.half、.word、.float和.double的自动分界方法，直到出现.data或.kdata为止。 |
| .ascii str | 将字符串str存入主存中，但不以空字符结束。 A-46 |
| .asciiz str | 将字符串str存入主存，并以空字符结束。 A-47 |

| | |
|---|---|
| .byte b1,…,bn | 将 $n$ 个值存入主存的连续字节中。 |
| .data <addr> | 将后续项存入数据段中。如果给出了可选参数 addr，则后续项存入以 addr 开始的主存地址中。 |
| .double d1,…,dn | 将 $n$ 个双精度浮点数存入连续的主存单元。 |
| .extern sym size | 声明存储在 sym 中大小为 size 字节的全局变量。该指令允许汇编器将数据存放到数据段中，这样可以用 $gp 寄存器快速存取。 |
| .float f1,…,fn | 将 $n$ 个单精度浮点数存入连续的主存单元。 |
| .globl sym | 声明 sym 是全局标记，可以在其他文件中引用。 |
| .half h1,…,hn | 将 $n$ 个 16 位数据存入连续的主存单元。 |
| .kdata <addr> | 将后续数据项存入核心数据段中。如果给出了可选参数 addr，则后续项存入以 addr 开始的主存地址中。 |
| .ktext <addr> | 将后续项放入核心代码段。在 SPIM 中，这些后续项只能是指令或字（参看下面的 .word 指令）。如果给出了可选参数 addr，则后续项存入以 addr 开始的主存地址中。 |
| .set noat 和 .set at | 前一指令阻止 SPIM 对后续指令中使用 $at 寄存器的警告，后一指令恢复这种警告。由于伪指令展开成指令时会用到寄存器 $at，程序员必须谨慎使用寄存器 $at。 |
| .space n | 在当前段分配 $n$ 字节（SPIM 中则必须为数据段）。 A-48 |
| .text <addr> | 将后续项送入用户代码段中。在 SPIM 中，这些后续项只能是指令或字（参看下面的 .word 指令）。如果给出了可选参数 addr，则后续项存入以 addr 开始的主存地址中。 |
| .word w1,…,wn | 将 $n$ 个 32 位数据存入连续的主存字中。 |

SPIM 不区分数据段的不同部分（.data，.rdata 和 .sdata）。

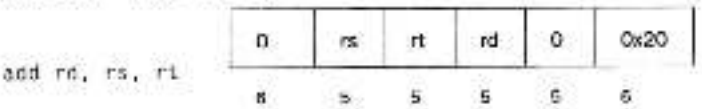
## A.10.3 MIPS 指令编码

图 A-10-2 描述了 MIPS 指令是如何以二进制数进行编码的。每一列包含指令字段（邻接的一组二进制位）的编码。左边界的数字是对应字段的值。例如，操作码 j 在操作码字段的值为 2。每列顶上的文字定义了一个字段，并且指出了占用指令中的哪些位。例如，op 字段对应指令中的 26－31 位。该字段对大多数指令进行了编码。然而，有些指令组用到了附加字段以区别相关的指令。例如，不同的浮点数指令用 0－5 位进行区别。第一列的箭头表明哪些操作码用到了这些附加字段。

## A.10.4 指令格式

本附录的剩余部分将对由 MIPS 硬件实现的指令和 MIPS 汇编器实现的伪指令进行描述。这两种指令很容易区分。实际指令的字段用对应的二进制来表示。例如：

**加法操作（带溢出位）**




add 指令由 6 个字段组成。字段的长度标在字段下面。该指令由 6 位 0 开始。寄存器标识符以 r 开始，因此接下来的字段是称为 rs 的 5 位寄存器标识符。它与本行左边汇编代码中的第二个参数相同。另一个常用字段是 $imm_{16}$，它是一个 16 位立即数。 A-49

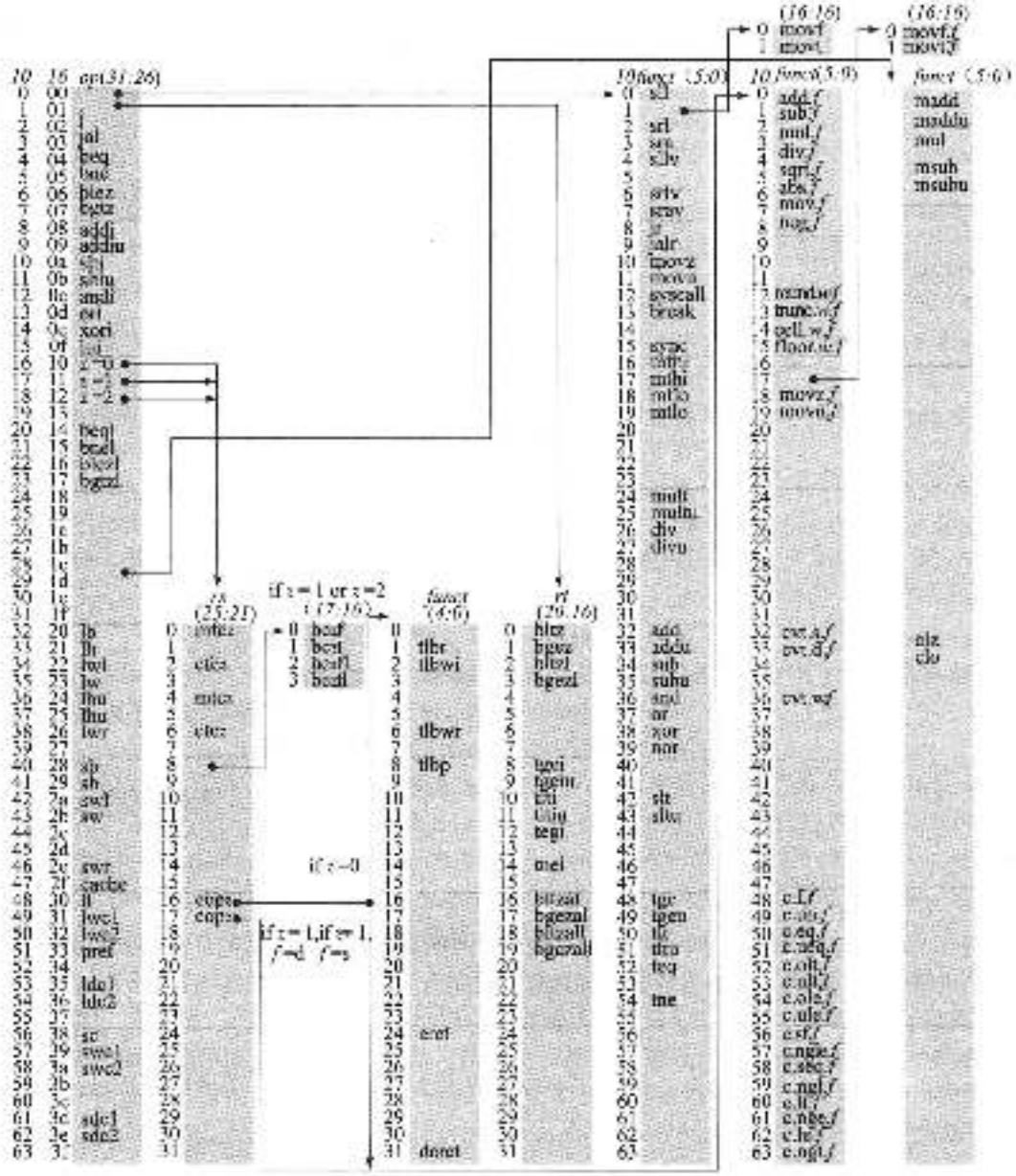


图 A-10-2 MIPS 操作码图。每个字段的数值在它的左侧显示。第一列、第二列是第三列中操作符字段（31～26 位）对应的十进制值和十六进制值。该操作符字段能表达除了 6 个操作数（0，1，16，17，18，19）以外的任何 MIPS 操作。这些操作由其他字段确定。虚指针进行识别。如果 rs = 16，op = 17，最后的字段（funct）用 "f" 表示 "s"；如果 rs = 17，op = 17，则 "f" 表示 "d"。如果 op = 16，17，18，19，则第二列（rs）用 "x" 分别表示："0"，"1"，"2"，"3"。如果 rs = 16，则操作由别处定义：如果 x = 0，则操作在第四个字段中定义（4～0 位）；如果 x = 1，则操作在最后的字段中，且 f = s。如果 rs = 17 且 x = 1，则操作在最后的字段中，且 f = d

伪指令大体上遵循这些约定，但省略了指令编码信息。例如：

**乘操作（不带溢出位）**

```
mul rdest, rsrc1, src2        伪指令
```

在伪指令中，rdest 和 rsrc1 表示寄存器，而 src2 表示寄存器或立即数。通常情况下，汇编器和 SPIM 将一条通用的指令格式（例如，add $v1, $a0, 0x55）转化为特定的形式（例如，addi $v1, $a0, 0x55）。

## A.10.5 算术和逻辑指令

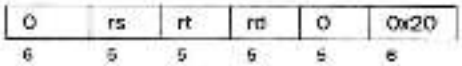
**绝对值**

abs rdest, rsrc　　伪指令

将寄存器 rsrc 的值求绝对值再存入寄存器 rdest 中。

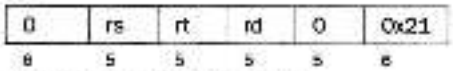
**加法（带溢出位）**

add rd, rs, rt




**加法（不带溢出位）**

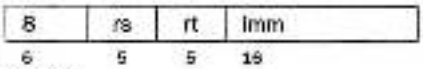
addu rd, rs, rt




将寄存器 rs 和 rt 的和存入寄存器 rd 中。

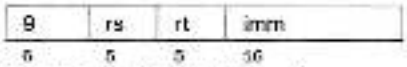
**立即数加（带溢出位）**

addi rt, rs, imm




**立即数加（不带溢出位）**

addiu rt, rs, imm

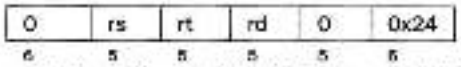



将寄存器 rs 与立即数之和存入寄存器 rt 中。

A-50
?
A-51

**逻辑与**

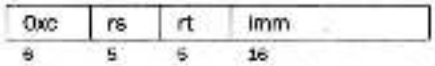
and rd, rs, rt




将寄存器 rs 与 rt 进行逐位逻辑与，结果存入寄存器 rd。

**立即数与**

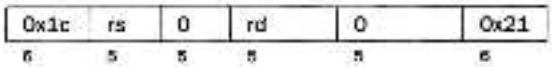
andi rt, rs, imm




将寄存器 rs 同立即数进行逐位逻辑与，结果存入寄存器 rt。

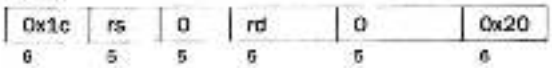
**统计起始为 1 的个数**

clo rd, rs




**统计起始为 0 的个数**

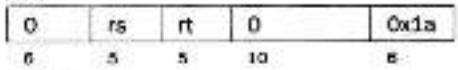
clz rd, rs




将寄存器 rs 中数据起始为 1（0）的个数存入寄存器 rd，如果字中都是 1（0），则结果为 32。

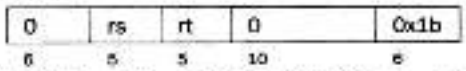
**除法（带溢出位）**

div rs, rt




**除法（不带溢出位）**

divu rs, rt




寄存器 rs 被寄存器 rt 除，将商存入寄存器 lo，将余数存入寄存器 hi。如果其中有某个操作数是负数，则余数取决于运行 SPIM 的计算机系统，而与 MIPS 体系结构无关。

A-52

**除法（带溢出位）**

```
div rdest, rsrc1, src2        伪指令
```

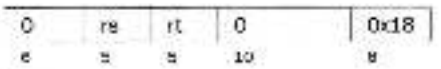
**除法（不带溢出位）**

```
divu rdest, rsrc1, src2       伪指令
```

将寄存器 rsrc1 和 src2 的商存入寄存器 rdest。

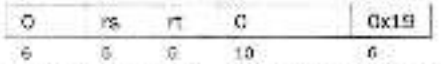
**乘法**

```
mult rs, rt
```




**无符号数乘法**

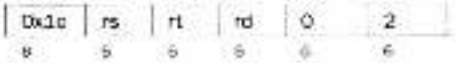
```
multu rs, rt
```




将寄存器 rs 和 rt 的数据相乘，乘积的低位字和高位字分别存入寄存器 lo 和 hi。

**乘积（不带溢出位）**

```
mul rd, rs, rt
```




将 rs 和 rt 乘积的低 32 位存入寄存器 rd 中。

**乘积（带溢出位）**

```
mulo rdest, rsrc1, src2       伪指令
```

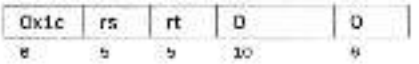
**无符号数相乘（带溢出位）**

```
mulou rdest, rsrc1, src2      伪指令
```

A-53 将寄存器 rsrc1 和 src2 的乘积结果的低 32 位存入寄存器 rdest。

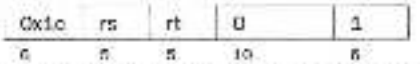
**乘加**

```
madd rs, rt
```




**无符号乘加**

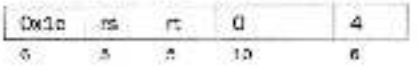
```
maddu rs, rt
```




将寄存器 rs 和 rt 的乘积所得的 64 位结果与连接寄存器 lo 和 hi 中的 64 位值相加。

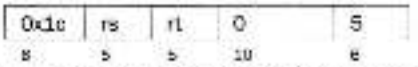
**乘减**

```
msub rs, rt
```




**无符号乘减**

```
msubu rs, rt
```




将寄存器 rs 和 rt 的乘积所得的 64 位结果与连接寄存器 lo 和 hi 中的 64 位值相减。

**求相反数（带溢出位）**

```
neg rdest, rsrc       伪指令
```

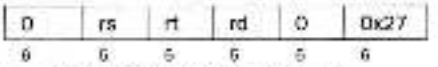
**求相反数（不带溢出位）**

```
negu rdest, rsrc      伪指令
```

将寄存器 rsrc 的相反数存入寄存器 rdest。

**异或**

```
nor rd, rs, rt
```




A-54 将寄存器 rs 和 rt 的异或结果存入寄存器 rd。

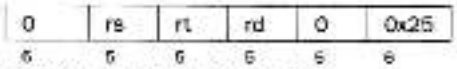
**取反**

```
not rdest, rsrc        伪指令
```

将寄存器 rsrc 逐位取反存入寄存器 rdest。

**逻辑或**

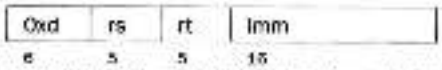
```
or rd, rs, rt
```




将寄存器 rs 和 rt 按位逻辑或的结果存入寄存器 rd。

**逻辑或（立即数）**

```
ori rt, rs, imm
```




将寄存器 rs 和0扩展立即数按位逻辑或的结果存入寄存器 rt。

**求余数**

```
rem rdest, rsrc1, rsrc2        伪指令
```

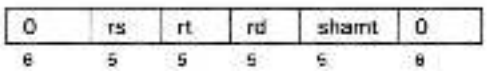
**求无符号数的余数**

```
remu rdest, rsrc1, rsrc2        伪指令
```

寄存器 rsrc1 被寄存器 rsrc2 除，将余数存入寄存器 rdest。注意，如果其中有某个操作数是负数，则余数取决于运行 SPIM 的计算机系统，而与 MIPS 体系结构无关。

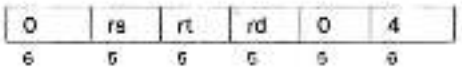
**逻辑左移**

```
sll rd, rt, shamt
```




**逻辑左移变量**

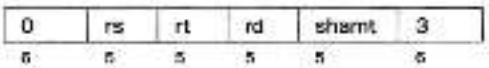
```
sllv rd, rt, rs
```




A-55

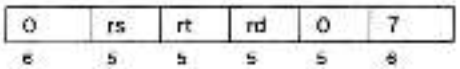
**算术右移**

```
sra rd, rt, shamt
```



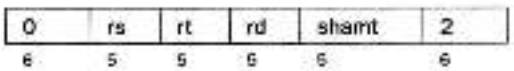

**算术右移变量**

```
srav rd, rt, rs
```



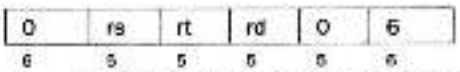

**逻辑右移**

```
srl rd, rt, shamt
```




**逻辑右移变量**

```
srlv rd, rt, rs
```




由立即数 shamt 或寄存器 rs 指定寄存器 rt 的左移或右移位数，并将结果存入寄存器 rd。注意，变量 rs 被 sll、sra 和 srl 所忽略。

**循环左移**

```
rol rdest, rsrc1, rsrc2        伪指令
```

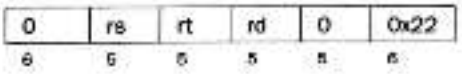
**循环右移**

```
ror rdest, rsrc1, rsrc2        伪指令
```

将寄存器 rsrc1 左移或右移由 rsrc2 指定的位数，然后将结果存入寄存器 rdest。

**减法（带溢出位）**

```
sub rd, rs, rt
```




A-56

**减法（不带溢出位）**

sub rd, rs, rt

| 0 | rs | rt | rd | 0 | 0x23 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

将寄存器 rs 减去寄存器 rt 并将结果存入寄存器 rd。

**异或**

xor rd, rs, rt

| 0 | rs | rt | rd | 0 | 0x26 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

将寄存器 rs 和 rt 按位逻辑异或的结果存入寄存器 rd。

**异或（同立即数）**

xori rt, rs, imm

| 0xe | rs | rt | imm |
|---|---|---|---|
| 6 | 5 | 5 | 16 |

将寄存器 rs 和 0 扩展立即数按位逻辑异或的结果存入寄存器 rt。

## A.10.6 常数操作指令

**立即数高位取指令**

lui rt, imm

| 0xf | 0 | rt | imm |
|---|---|---|---|
| 6 | 5 | 5 | 16 |

将立即数 imm 的低半字节存入寄存器 rt 的高半字节地址，并将寄存器的低位值置为 0。

**取立即数**

li rdest, imm　　为指令

将立即数 imm 存入寄存器 rdest。

## A.10.7 比较指令

**小于指令**

slt rd, rs, rt

| 0 | rs | rt | rd | 0 | 0x2a |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

A-57

**小于无符号数指令**

sltu rd, rs, rt

| 0 | rs | rt | rd | 0 | 0x2b |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

若寄存器 rs 比 rt 小，则将寄存器 rd 置为 1；否则，将 rd 置为 0。

**小于立即数**

slti rt, rs, imm

| 0xa | rs | rt | imm |
|---|---|---|---|
| 6 | 5 | 5 | 16 |

**小于立即数（无符号数）**

sltiu rt, rs, imm

| 0xb | rs | rt | imm |
|---|---|---|---|
| 6 | 5 | 5 | 16 |

若寄存器 rs 比符号扩展立即数小，则将寄存器 rt 置为 1；否则，将 rt 置为 0。

**等于**

seq rdest, rsrc1, rsrc2　　为指令

若寄存器 rsrc1 与寄存器 rsrc2 的数值相等，则将寄存器 rdest 置为 1；否则，将 rdest 置为 0。

**大于等于**

sge rdest, rsrc1, rsrc2　　为指令

**大于等于无符号数**

```
sgeu rdest, rsrc1, rsrc2        伪指令
```

若寄存器 rsrc1 大于等于寄存器 rsrc2 的值，则将寄存器 rdest 置为 1；否则，将 rdest 置为 0。

**大于**

```
sgt rdest, rsrc1, rsrc2         伪指令
```

A-58

**大于无符号数**

```
sgtu rdest, rsrc1, rsrc2        伪指令
```

如果寄存器 rsrc1 的值大于 rsrc2 的值，那么令寄存器 rdest 的值为 1，否则为 0。

**小于等于**

```
sle rdest, rsrc1, rsrc2         伪指令
```

**小于等于无符号数**

```
sleu rdest, rsrc1, rsrc2        伪指令
```

如果寄存器 rsrc1 的值小于等于 rsrc2 的值，那么令寄存器 rdest 的值为 1，否则为 0。

**不等**

```
sne rdest, rsrc1, rsrc2         伪指令
```

如果寄存器 rsrc1 的值不等于 rsrc2 的值，那么令寄存器 rdest 的值为 1，否则为 0。

## A.10.8 分支指令

分支指令使用了一个有符号的 16 位指令偏移域；因此，指令跳转的范围可以是向前的 $2^{15}-1$ 条指令（非字节），或者向后的 $2^{15}$ 条指令。跳转指令包含了一个 26 位的地址域。在实际的 MIPS 处理器中，分支指令是延迟的分支，直到分支指令后面的指令（延迟槽）执行后，才能进行控制转移（见第 4 章）。当分支发生时，由于需要计算相关的延迟槽指令（PC+4）的地址，因此延迟的分支会影响偏移量的计算。除非明确指定 -bare 或者 -delayed_branch 的标志，否则 SPIM 不模拟延迟槽。

在汇编语言中，偏移址并不具体指定为数字。而是用一个指向标记的指令，并用汇编器计算出分支指令和目标指令之间的距离。

在 MIPS-32 中，所有实际的（不是伪的）条件分支指令都有相似的变体（例如，与 beq 相似的变体是 beql），如果分支没有发生，那么分支延迟槽中的指令就不能执行。不要使用这些指令，在后续的体系结构版本中，它们可能将被删除。SPIM 实现了这些指令，但并没有做深入讨论。

A-59

**分支指令**

```
b label         伪指令
```

无条件转移到标记的指令。

**分支协处理器假**

```
bc1f cc label
```

| Ox11 | 8 | cc | 0 | Offset |
|---|---|---|---|---|
| 6 | 5 | 3 | 2 | 16 |

**分支协处理器真**

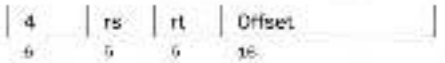
```
bc1t cc label
```

| Ox11 | 8 | cc | 1 | Offset |
|---|---|---|---|---|
| 6 | 5 | 3 | 2 | 16 |

如果浮点协处理器条件标记 cc 为假（真），条件转移的指令数由偏移址所指定。如果 cc 被指令所忽略，条件码标记为 0。

**相等分支**

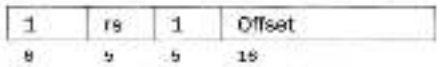
```
beq rs, rt, label
```




如果寄存器值 rs 和 rt 相等，条件转移的指令数由偏移量所指定。

**大于等于 0 分支**

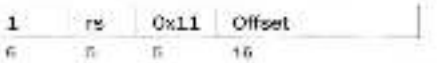
```
bgez rs, label
```




A-60 如果寄存器 rs 的值大于等于 0，条件转移的指令数由偏移量所指定。

**大于等于 0 分支并链接**

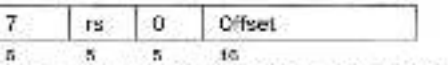
```
bgezal rs, label
```




如果寄存器 rs 的值大于等于 0，条件转移的指令数由偏移量所指定，并将下一条指令地址保存在寄存器 31 中。

**大于 0 分支**

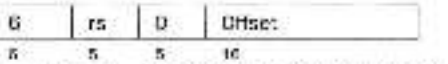
```
bgtz rs, label
```




如果寄存器 rs 的值大于 0，条件转移的指令数由偏移量所指定。

**小于等于 0 分支**

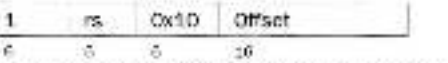
```
blez rs, label
```




如果寄存器 rs 的值小于等于 0，条件转移的指令数由偏移量所指定。

**小于 0 分支并链接**

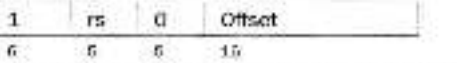
```
bltzal rs, label
```




如果寄存器 rs 的值小于 0，条件转移的指令数由偏移量所指定，并将下一条指令地址保存在寄存器 31 中

**小于 0 分支**

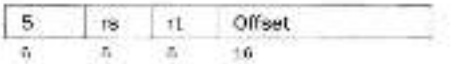
```
bltz rs, label
```




A-61 如果寄存器 rs 的值小于 0，条件转移的指令数由偏移量所指定。

**不相等分支**

```
bne rs, rt, label
```




如果寄存器 rs 与 rt 中的值不相等，条件转移的指令数由偏移量所指定。

**等于 0 分支**

```
beqz rsrc, label        伪指令
```

如果 rsrc 等于 0，条件转移到标记的指令那里。

**大于等于分支**

```
bge rsrc1, rsrc2, label        伪指令
```

**大于等于无符号数分支**

```
bgeu rsrc1, rsrc2, label        伪指令
```

如果寄存器 rsrc1 的值大于等于 rsrc2 的值，条件转移到标记的指令那里。

**大于分支**

```
bgt rsrc1, src2, label        伪指令
```

**大于无符号数分支**

```
bgtu rsrc1, src2, label        伪指令
```

如果寄存器 rsrc1 的值大于 src2 的值，条件转移到标记的指令那里。

**小于等于分支**

```
ble rsrc1, src2, label         伪指令
```

A-62

**小于等于无符号数分支**

```
bleu rsrc1, src2, label        伪指令
```

如果寄存器 rsrc1 的值小于等于 src2 的值，条件转移到标记的指令那里。

**小于分支**

```
blt rsrc1, rsrc2, label        伪指令
```

**小于无符号数分支**

```
bltu rsrc1, rsrc2, label       伪指令
```

如果寄存器 rsrc1 的值小于 rsrc2 的值，条件转移到标记的指令那里。

**不等于0分支**

```
bnez rsrc, label       伪指令
```

如果寄存器 rsrc 的值不等于0，条件转移到标记的指令那里。

## A.10.9 跳转指令

**跳转**

```
j target
```

| 2 | target |
|---|---|
| 6 | 26 |

无条件跳转到目标指令。

**跳转并链接**

```
jal target
```

| 3 | target |
|---|---|
| 6 | 26 |

无条件跳转到目标指令，并将下一条指令地址保存到 $ra 中。

A-63

**跳转并链接到寄存器**

```
jalr rs, rd
```

| 0 | rs | 0 | rd | 0 | 9 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

无条件跳转到由寄存器 rs 指定的指令（指令地址在寄存器 rs 中），并将下一条指令地址保存在寄存器 rd 中（默认为31）。

**寄存器跳转**

```
jr rs
```

| 0 | rs | 0 | 8 |
|---|---|---|---|
| 6 | 5 | 16 | 6 |

无条件跳转到由寄存器 rs 指定的指令。

## A.10.10 陷阱指令

**等于陷阱**

```
teq rs, rt
```

| 0 | rs | rt | 0 | 0x34 |
|---|---|---|---|---|
| 6 | 5 | 5 | 10 | 6 |

如果寄存器 rs 的值等于寄存器 rt 的值，引发陷阱异常。

**等于立即数陷阱**

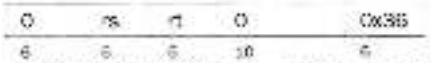
```
teqi rs, imm
```

| 1 | rs | 0xc | imm |
|---|---|---|---|
| 6 | 5 | 5 | 16 |

如果寄存器 rs 的值等于符号扩展值——imm，引发陷阱异常。

**不等于陷阱**

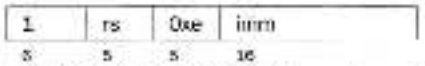
teq rs, rt




如果寄存器 rs 的值不等于寄存器 rt 的值，引发陷阱异常。

**不等于立即数陷阱**

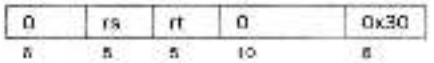
teqi rs, imm




A-64 如果寄存器 rs 的值不等于符号扩展值——imm，引发陷阱异常。

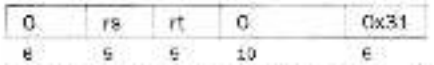
**大于等于陷阱**

tge rs, rt




**大于等于无符号数陷阱**

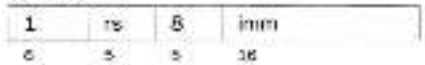
tgeu rs, rt




如果寄存器 rs 的值大于或等于寄存器 rt 的值，引发陷阱异常。

**大于等于立即数陷阱**

tgei rs, imm




**大于等于无符号立即数陷阱**

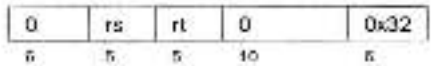
tgeiu rs, imm




如果寄存器 rs 的值大于等于符号扩展值——imm，引发陷阱异常。

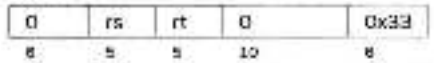
**小于陷阱**

tlt rs, rt




**小于无符号数陷阱**

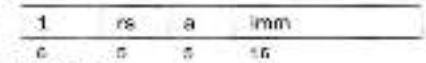
tltu rs, rt




如果寄存器 rs 的值小于寄存器 rt 的值，引发陷阱异常。

**小于立即数陷阱**

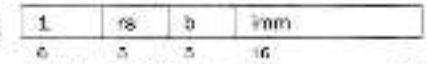
tlti rs, imm




A-65

**小于无符号立即数陷阱**

tltiu rs, imm




如果寄存器 rs 的值小于符号扩展值——imm，引发陷阱异常。

## A.10.11 取数指令

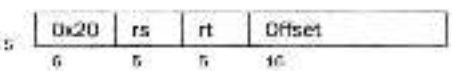
**取地址**

la rdest, address 伪指令

将计算的地址——不是地址中的内容——保存到寄存器 rdest 中。

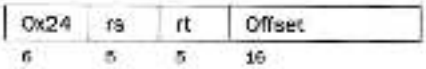
**取字节**

lb rt, address



**取字节（无符号）**

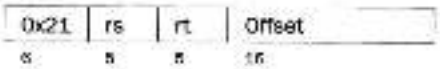
```
lbu rt, address
```




将地址 address 中的字节内容存入寄存器 rt 中，字节由 lb 符号扩展，而不是由 lbu。

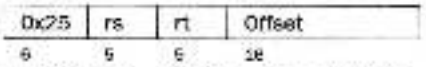
**取半字**

```
lh rt, address
```




**取半字（无符号）**

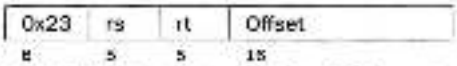
```
lhu rt, address
```




将地址 address 中 16 位数值（半字）存入寄存器 rt 中，半字由 lh 符号扩展，而不是由 lhu。 A-66

**取字**

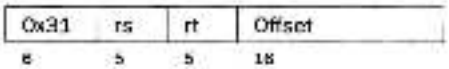
```
lw rt, address
```




将地址 address 中 32 位数值（字）存入寄存器 rt 中。

**协处理器1 取字**

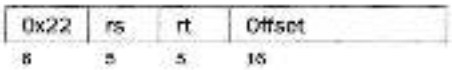
```
lwc1 ft, address
```




将地址 address 中的字以浮点单元的形式存入寄存器 ft 中。

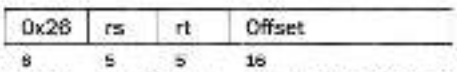
**取左半字**

```
lwl rt, address
```




**取右半字**

```
lwr rt, address
```




将可能非对齐地址 address 中值的左（右）半字存入寄存器 rt 中。

**取双字**

```
ld rdest, address        伪指令
```

将地址 address 对应的 64 位数值存入寄存器 rdest 和 rdest +1 中。

**非对齐地址中取半字**

```
ulh rdest, address       伪指令
```

A-67

**非对齐地址中取半字（无符号）**

```
ulhu rdest, address      伪指令
```

将可能非对齐地址 address 中 16 位数值（半字）存入寄存器 rdest 中，半字由 ulh 符号扩展，而不是由 ulhu。

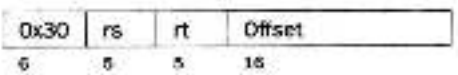
**非对齐地址中取字**

```
ulw rdest, address       伪指令
```

将可能非对齐地址 address 中 32 位数值（字）存入寄存器 rdest 中。

**链接取**

```
ll rt, address
```



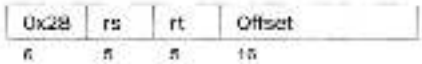

将 address 中 32 位数值存入寄存器 rt 中，并且开始执行原子读—修改—写操作。该操作由条件存指令（sc）来完成，但如果其他处理器对包含有被取字的块进行写操

作时，该操作将失败。由于SPIM不能模拟多处理器，因而条件存操作总是可以成功执行的。

## A.10.12 保存指令

**存字节**

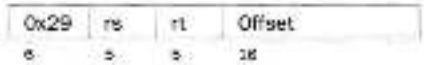
```
sb rt, address
```




将寄存器rt的低字节保存到地址address中。

**存半字**

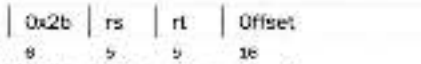
```
sh rt, address
```




A-68 将寄存器rt的低16位值（半字）保存到地址address中。

**存字**

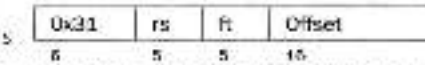
```
sw rt, address
```




将寄存器rt中的字保存到地址address中。

**协处理器1存字**

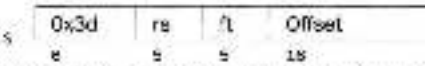
```
swc1 ft, address
```




将浮点协处理器中寄存器ft中的值以浮点类型存入地址address中。

**协处理器1存双字**

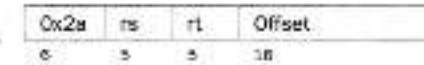
```
sdc1 ft, address
```




将浮点协处理器中寄存器ft和ft+1中的数值以浮点类型存入地址address中。寄存器ft必须偶数化。

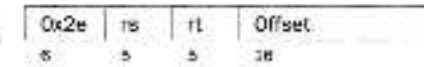
**存左半字**

```
swl rt, address
```




**存右半字**

```
swr rt, address
```




将寄存器rt中的左（右）半字保存到可能非对齐地址address中。

**存双字**

```
sd rsrc, address        伪指令
```

A-69 将寄存器rsrc和rsrc+1中的64位数值保存到地址address中。

**非对齐地址中存半字**

```
ush rsrc, address       伪指令
```

将寄存器rsrc中的低16位（半字）保存到可能的非对齐地址address中。

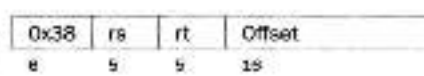
**非对齐地址中存字**

```
usw rsrc, address       伪指令
```

将寄存器rsrc中的字保存到可能的非对齐地址address中。

**条件存**

```
sc rt, address
```



将寄存器 rt 中的32位数值（字）存入内存地址 address 中，并完成原子读—修改—写操作。如果原子操作成功执行，内存中的字被修改，寄存器 rt 的值设置为 1。如果由于其他处理器对包含地址字的块进行写操作而导致原子操作失败，该指令不能修改内存，并将寄存器 rt 的值设置为 0。由于 SPIM 不能模拟多处理器，因而该指令总是可以成功执行的。

## A.10.13 数据传送指令

**传送指令**

```
move rdest, rsrc        伪指令
```

将寄存器 rsrc 中的数值传送到寄存器 rdest 中。

**从 hi 寄存器传送**

```
mfhi rd
```

| 0 | 0 | rd | 0 | 0x10 |
|---|---|---|---|---|
| 6 | 10 | 5 | 5 | 6 |

A-70

**从 lo 寄存器传送**

```
mflo rd
```

| 0 | 0 | rd | 0 | 0x12 |
|---|---|---|---|---|
| 6 | 10 | 5 | 5 | 6 |

乘法和除法单元将处理的结果存入 hi 和 lo 这两个额外的寄存器中。这些指令向（从）这些寄存器中传送数据。乘、除、取余伪指令像使用通用寄存器那样使用这些单元，并在计算结束后传送结果。

将寄存器 hi（lo）中的数值传送到寄存器 rd 中。

**传送至 hi 寄存器**

```
mthi rs
```

| 0 | rs | 0 | 0x11 |
|---|---|---|---|
| 6 | 5 | 15 | 6 |

**传送至 lo 寄存器**

```
mtlo rs
```

| 0 | rs | 0 | 0x13 |
|---|---|---|---|
| 6 | 5 | 15 | 6 |

将寄存器 rs 的值传送至 hi（lo）寄存器。

**从协处理器 0 中传送**

```
mfc0 rt, rd
```

| 0x10 | 0 | rt | rd | 0 |
|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 11 |

**从协处理器 1 中传送**

```
mfc1 rt, fs
```

| 0x11 | 0 | rt | fs | 0 |
|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 11 |

协处理器有它们自己的寄存器集合。这些指令在协处理器的寄存器和 CPU 寄存器之间传送数据。

将协处理器中寄存器 rd（在 FPU 中是 fs）的值传送至 CPU 寄存器 rt 中。浮点单元使用协处理器 1。

A-71

**从协处理器 1 中传送双字**

```
mfc1.d rdest, frsrc1        伪指令
```

将浮点寄存器 frsrc1 和 frsrc1 +1 中的值传送到 CPU 寄存器 rdest 和 rdest +1 中。

**传送到协处理器 0**

```
mtc0 rd, rt
```

| 0x10 | 4 | rt | rd | 0 |
|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 11 |

**传送到协处理器1**

```
mtc1 rd, rs    | 0x11 | 4 | rt | fs | 0 |
               |  6   | 5 |  5 |  6 | 11 |
```

将CPU中寄存器rt的值传送到协处理器的寄存器rd中（或者FPU的寄存器fs中）。

**非零条件传送**

```
movn rd, rs, rt | 0 | rs | rt | rd | 0x13 |
                | 6 |  5 |  5 |  5 |   11 |
```

如果寄存器rt的值不为0，将寄存器rs中的数值传送到寄存器rd中。

**零条件传送**

```
movz rd, rs, rt | 0 | rs | rt | rd | 0x12 |
                | 6 |  5 |  5 |  5 |   11 |
```

如果寄存器rt的值为0，将寄存器rs中的数值传送到寄存器rd中。

**FP值为假时条件传送**

```
movf rd, rs, cc | 0 | rs | cc | 0 | rd | 0 | 1 |
                | 6 |  5 |  3 | 2 |  5 | 5 | 6 |
```

如果FPU条件码标记cc为0，将CPU寄存器rs中的值传送至寄存器rd中。如果cc被指
[A-72] 令忽略，那么条件码标记为0。

**FP值为真时条件传送**

```
movt rd, rs, cc | 0 | rs | cc | 1 | rd | 0 | 1 |
                | 6 |  5 |  3 | 2 |  5 | 5 | 6 |
```

如果FPU条件码标记cc为1，将CPU寄存器rs中的值传送至寄存器rd中。如果cc被指令忽略，那么条件码标记为0。

## A.10.14 浮点运算指令

MIPS中有专门的浮点协处理器（序号为1），可以执行单精度浮点数（32位）和双精度浮点数（64位）。协处理器有自己的寄存器，寄存器从$f0 - $f31。由于这些寄存器位宽为32位，因此两个浮点寄存器一起使用可以实现双精度浮点数值。浮点协处理器还有8个条件码（cc）标记，序号0~7，由比较指令设置，分支（bc1f和bc1t）和条件转移指令完成校验。

lwc1、swc1、mtc1和mfc1指令每次能从寄存器传送或者移出一个字（32位）。ldc1和sdc1指令，或者像下面描述的l.s、l.d、s.s和s.d伪指令每次能像寄存器传送或者移出一个双字（64位）。

在下面的实际指令中，单精度指令的21~26位为0，双精度指令的21~26位则为1。在下面的伪指令中，fdest是浮点寄存器（如$f2）。

**双精度浮点数的绝对值**

```
abs.d fd, fs   | 0x11 | 1 | 0 | fs | fd | 5 |
               |  6   | 5 | 5 |  5 |  5 | 6 |
```

**单精度浮点数的绝对值**

```
abs.s fd, fs   | 0x11 | 0 | 0 | fs | fd | 5 |
```

计算寄存器fs中双精度（单精度）浮点数的绝对值，并将计算结果存入寄存器fd中。

**双精度浮点加法**

```
add.d fd, fs, ft | 0x11 | 0x11 | ft | fs | fd | 0 |
[A-73]           |  6   |  6   |  5 |  5 |  5 | 6 |
```

**单精度浮点加法**

```
add.s fd, fs, ft
```

| 0x11 | 0x10 | ft | fs | fd | 0 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

计算寄存器 fs 和 ft 中双精度（单精度）浮点数之和，并将计算结果存入寄存器 fd 中。

**浮点数向上舍入**

```
ceil.w.d fd, fs
```

| 0x11 | 0x11 | 0 | fs | fd | 0xe |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

```
ceil.w.s fd, fs
```

| 0x11 | 0x10 | 0 | fs | fd | 0xe |
|---|---|---|---|---|---|

将寄存器 fs 中双精度（单精度）数值向上舍入，并转换成 32 位的定点值，将结果存放在寄存器 fd 中。

**双精度相等比较**

```
c.eq.d cc fs, ft
```

| 0x11 | 0x11 | ft | fs | cc | 0 | FC | 2 |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 3 | 2 | 2 | 4 |

**单精度相等比较**

```
c.eq.s cc fs, ft
```

| 0x11 | 0x10 | ft | fs | cc | 0 | FC | 2 |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 3 | 2 | 2 | 4 |

比较寄存器 fs 和 ft 中双精度（单精度）浮点数是否相等，如果相等，将浮点条件标记位 cc 设置为 1。如果 cc 被忽略，条件码标记为 0。

**双精度小于等于比较**

```
c.le.d cc fs, ft
```

| 0x11 | 0x11 | ft | fs | cc | 0 | FC | 0xe |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 3 | 2 | 2 | 4 |

**单精度小于等于比较**

```
c.le.s cc fs, ft
```

| 0x11 | 0x10 | ft | fs | cc | 0 | FC | 0xe |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 3 | 2 | 2 | 4 |

A-74

将寄存器 fs 和 ft 中双精度（单精度）浮点数进行比较，如果 fs 中的数值小于等于 ft 中的数值，将浮点条件标记位 cc 设置为 1。如果 cc 被忽略，条件码标记为 0。

**双精度小于比较**

```
c.lt.d cc fs, ft
```

| 0x11 | 0x11 | ft | fs | cc | 0 | FC | 0xc |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 3 | 2 | 2 | 4 |

**单精度小于比较**

```
c.lt.s cc fs, ft
```

| 0x11 | 0x10 | ft | fs | cc | 0 | FC | 0xc |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 3 | 2 | 2 | 4 |

将寄存器 fs 和 ft 中双精度（单精度）浮点数进行比较，如果 fs 中的数值小于 ft 中的数值，将浮点条件标记位 cc 设置为 1。如果 cc 被忽略，条件码标记为 0。

**单精度到双精度的转换**

```
cvt.d.s fd, fs
```

| 0x11 | 0x10 | 0 | fs | fd | 0x21 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

**整型到双精度的转换**

```
cvt.d.w fd, fs
```

| 0x11 | 0x14 | 0 | fs | fd | 0x21 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

将寄存器 fs 中的单精度浮点数或者整型数转换成双精度（单精度）浮点数，并存入寄存器 fd 中。

**双精度到单精度的转换**

```
cvt.s.d fd, fs
```

| 0x11 | 0x11 | 0 | fs | fd | 0x20 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

**整型到单精度的转换**

```
cvt.s.w fd, fs
```

| 0x11 | 0x14 | 0 | fs | fd | 0x20 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

A-75 将寄存器 fs 中的双精度浮点数或者整型数转换成单精度浮点数，并存入寄存器 fd 中。

**双精度到整型的转换**

```
cvt.w.d fd, fs
```

| 0x11 | 0x11 | 0 | fs | fd | 0x24 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

**单精度到整型的转换**

```
cvt.w.s fd, fs
```

| 0x11 | 0x10 | 0 | fs | fd | 0x24 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

将寄存器 fs 中的双精度浮点数或者单精度浮点数转换成整型数，并存入寄存器 fd 中。

**双精度浮点除法**

```
div.d fd, fs, ft
```

| 0x11 | 0x11 | ft | fs | fd | 3 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

**单精度浮点除法**

```
div.s fd, fs, ft
```

| 0x11 | 0x10 | ft | fs | fd | 3 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

将寄存器 fs 和 ft 中的双精度（单精度）浮点数相除，并将计算结果存入寄存器 fd 中。

**浮点数向下舍入**

```
floor.w.d fd, fs
```

| 0x11 | 0x11 | 0 | fs | fd | 0xf |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

```
floor.w.s fd, fs
```

| 0x11 | 0x10 | 0 | fs | fd | 0xf |
|---|---|---|---|---|---|

将寄存器 fs 中的双精度（单精度）数值向下舍入，并将结果存放在寄存器 fd 中。

**取双精度浮点数**

A-76 `l.d fdest, address` 伪指令

**取单精度浮点数**

`l.s fdest, address` 伪指令

将地址 address 相应的双精度（单精度）浮点数存入寄存器 fdest 中。

**双精度浮点数的传送**

```
mov.d fd, fs
```

| 0x11 | 0x11 | 0 | fs | fd | 6 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

**单精度浮点数的传送**

```
mov.s fd, fs
```

| 0x11 | 0x10 | 0 | fs | fd | 6 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。

**条件为假时双精度浮点数传送**

```
movf.d fd, fs, cc
```

| 0x11 | 0x11 | cc | 0 | fs | fd | 0x11 |
|---|---|---|---|---|---|---|
| 6 | 5 | 3 | 2 | 5 | 5 | 6 |

**条件为假时单精度浮点数传送**

```
movf.s fd, fs, cc
```

| 0x11 | 0x10 | cc | 0 | fs | fd | 0x11 |
|---|---|---|---|---|---|---|
| 6 | 5 | 3 | 2 | 5 | 5 | 6 |

如果条件码标记 cc 为0，将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。如果 cc 被忽略，条件码标记为0。

**条件为真时双精度浮点数传送**

movt.d fd, fs, cc

| 0x11 | 0x11 | cc | 1 | fs | fd | 0x11 |
|---|---|---|---|---|---|---|
| 6 | 5 | 3 | 2 | 5 | 5 | 6 |

**条件为真时单精度浮点数传送**

movt.s fd, fs, cc

| 0x11 | 0x10 | cc | 1 | fs | fd | 0x11 |
|---|---|---|---|---|---|---|
| 6 | 5 | 3 | 2 | 5 | 5 | 6 |

A-77

如果条件码标记 cc 为 1，将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。如果 cc 被忽略，条件码标记为 0。

**非零条件双精度浮点数传送**

movn.d fd, fs, rt

| 0x11 | 0x11 | rt | fs | fd | 0x13 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

**非零条件单精度浮点数传送**

movn.s fd, fs, rt

| 0x11 | 0x10 | rt | fs | fd | 0x13 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

如果处理器寄存器 rt 中的值不等于 0，那么将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。

**等于零条件双精度浮点数传送**

movz.d fd, fs, rt

| 0x11 | 0x11 | rt | fs | fd | 0x12 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

**等于零条件单精度浮点数传送**

movz.s fd, fs, rt

| 0x11 | 0x10 | rt | fs | fd | 0x12 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

如果处理器寄存器 rt 中的值等于 0，那么将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。

**双精度浮点乘**

mul.d fd, fs, ft

| 0x11 | 0x11 | ft | fs | fd | 2 |
|---|---|---|---|---|---|
| 6 | 6 | 5 | 5 | 5 | 6 |

**单精度浮点乘**

mul.s fd, fs, ft

| 0x11 | 0x10 | ft | fs | fd | 2 |
|---|---|---|---|---|---|
| 6 | 6 | 5 | 5 | 5 | 6 |

将寄存器 fs 和 ft 中的双精度（单精度）浮点数相乘，并将计算结果存入寄存器 fd 中。

**对双精度数求反**

neg.d fd, fs

| 0x11 | 0x11 | 0 | fs | fd | 7 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

A-78

**对单精度数求反**

neg.s fd, fs

| 0x11 | 0x10 | 0 | fs | fd | 7 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

对寄存器 fs 中的双精度（单精度）浮点数求反，并将结果存入寄存器 fd 中。

**对浮点数四舍五入**

round.w.d fd, fs

| 0x11 | 0x11 | 0 | fs | fd | 0xc |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

round.w.s fd, fs

| 0x11 | 0x10 | 0 | fs | fd | 0xc |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

将寄存器 fs 中的双精度（单精度）数四舍五入，转换成 32 位的定点数，并存入寄存器 fd 中。

**对双精度数求平方根**

```
sqrt.d fd, fs
```

| 0x11 | 0x11 | 0 | fs | fd | 4 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

**对单精度数求平方根**

```
sqrt.s fd, fs
```

| 0x11 | 0x10 | 0 | fs | fd | 4 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

对寄存器 fs 中的双精度（单精度）数求平方根，并存入寄存器 fd 中。

**保存双精度浮点数**

```
s.d fdest, address
```

伪指令

**保存单精度浮点数**

```
s.s fdest, address
```

伪指令

将寄存器 fdest 中的双精度（单精度）浮点数存入地址 address 中。

**双精度浮点减法**

```
sub.d fd, fs, ft
```

| 0x11 | 0x11 | ft | fs | fd | 1 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

A-79

**单精度浮点减法**

```
sub.s fd, fs, ft
```

| 0x11 | 0x10 | ft | fs | fd | 1 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

将寄存器 fs 和 ft 中的双精度（单精度）浮点数相减，并将计算结果存入寄存器 fd 中。

**将浮点数截取为字**

```
trunc.w.d fd, fs
```

| 0x11 | 0x11 | 0 | fs | fd | 0xd |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

```
trunc.w.s fd, fs
```

| 0x11 | 0x10 | 0 | fs | fd | 0xd |
|---|---|---|---|---|---|

对寄存器 fs 中的双精度（单精度）浮点数进行截取操作，转换成 32 位定点数，并将结果存入寄存器 fd 中。

## A.10.15 异常和中断指令

**异常返回**

```
eret
```

| 0x10 | 1 | 0 | 0x18 |
|---|---|---|---|
| 6 | 1 | 19 | 6 |

将协处理器 0 的状态寄存器中的 EXL 位设置为 0，并返回协处理器 0 中 EPC 寄存器指向的指令。

**系统调用**

```
syscall
```

| 0 | 0 | 0xc |
|---|---|---|
| 6 | 20 | 6 |

寄存器 $v0 中保存了 SPIM 提供的系统调用的个数（见图 A-9-1）。

**跳出**

```
break code
```

| 0 | code | 0xd |
|---|---|---|
| 6 | 20 | 6 |

产生异常码，异常 1 为调试程序保留。

**空操作**

```
nop
```

| 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 |

A-80

不做任何操作。

## A.11 小结

用汇编语言进行程序设计需要程序员放弃高级语言中的一些有益的特点——如数据结构、类型检查以及控制结构——以获得对机器执行指令的完全控制。一些应用的外部约束，如响应时间、程序大小等，需要程序员密切关注每条指令。然而，和高级语言程序相比，这种级别的关注带来的是更长、编写更费时、更难维护的汇编语言程序。

此外，三个趋势导致不必再用汇编语言来编写程序。第一个趋势是编译器的改进。现在，编译器生成的代码可以与最好的手工书写的代码相媲美——有时候甚至会更好。第二个趋势是新处理器的速度不仅更快，而且对于那些可以同时执行多条指令的处理器，手工编程也变得更加困难。此外，现代计算机的快速发展也支持高级语言程序不再依赖单一的体系结构。最后，我们见证了日渐复杂的应用趋势，不仅有复杂的图形界面，而且还有许多先前不曾遇见的特征。由程序员组成的团队合作开发的大规模应用程序需要有由高级语言提供的模块化设计思想和语义检查的特点。

## 拓展阅读


Aho, A., R. Sethi, and J. Ullman [1985]. *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley.

*Slightly dated and lacking in coverage of modern architectures, but still the standard reference on compilers.*

Sweetman, D. [1999]. *See MIPS Run*, San Francisco, CA: Morgan Kaufmann Publishers.

*A complete, detailed, and engaging introduction to the MIPS instruction set and assembly language programming on these machines.*

Detailed documentation on the MIPS-32 architecture is available on the Web:

MIPS32™ Architecture for Programmers Volume I: Introduction to the MIPS32™ Architecture
*(http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00082-2B-MIPS32INT-AFP-02.00.pdf/getDownload)*

MIPS32™ Architecture for Programmers Volume II: The MIPS32™ Instruction Set
*(http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00086-2B-MIPS32BIS-AFP-02.00.pdf/getDownload)*

MIPS32™ Architecture for Programmers Volume III: The MIPS32™ Privileged Resource Architecture
*(http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00090-2B-MIPS32PRA-AFP-02.00.pdf/getDownload)* A-81


## A.12 练习题

**A.1** [5] <A.5> A.5 节描述了在大多数 MIPS 系统中，内存是如何划分的。请采用其他的方法，实现相同结果。

**A.2** [20] <A.6> 用更少的指令重写 `fact` 程序。

**A.3** [5] <A.7> 用户程序使用寄存器 $k0 或 $k1 时总是安全的么？

**A.4** [25] <A.7> A.7 节介绍了一种非常简单的异常处理代码。这种处理方式的一个严重的缺陷在于它需要很长的时间来使中断无效。这意味着快速 I/O 设备发出的中断会丢失。请编写更好的可中断的异常处理程序，能尽快使中断有效。

**A.5** [15] <A.7> 简单的异常处理程序总是跳回异常之后的指令。这种操作运行良好，除非导致异常的指令处在分支指令的延迟槽中。这种情况下，下一条指令即是转移的目标。编写更好的程序，

使用 EPC 寄存器来决定异常之后执行哪一条指令。

**A.6** [5] <A.9> 使用 SPIM，编写、验证一个加法器程序：重复读入整数并对它们相加求和。当输入为 0 时停止程序，并输出累加和。使用 A.9 节介绍的 SPIM 系统调用。

**A.7** [5] <A.9> 使用 SPIM，编写、验证一个程序：读入三个整数，对两个最大的数求和并输出结果。使用 A.9 节介绍的 SPIM 系统调用。你可以任意中断程序。

**A.8** [5] <A.9> 使用 SPIM，编写、验证一个程序：使用 SPIM 的系统调用读入一个正整数。如果整数为非正，程序终止，输出 "Invalid Entry"；否则程序输出整数每个数字的名称，以空格分隔。例如，如果用户输入 "728"，输出 "Seven Two Eight"。

**A.9** [25] <A.9> 用 MIPS 汇编语言编写程序并验证：计算并输出前 100 个素数。如果除了 1 和 n 之外没有哪个数能整除 n，那么 n 为素数。你应该实现两个例程：

- `test_prime(n)` 如果 n 是素数，返回 1；如果不是，则返回 0。
- `main()` 循环测试每个整数是否为素数，并输出前 100 个素数。

A-82 在 SPIM 上验证你的程序。

**A.10** [10] <A.6, A.9> 使用 SPIM，编写、验证一个递归程序，来解决汉诺塔问题（需要使用堆栈帧来支持递归）。汉诺塔有三根杆子（1、2 和 3）和 n 个盘子（n 是可变的，典型的数值在 1～8 之间）。盘子 1 比盘子 2 小，盘子 2 比盘子 3 小，以此类推，盘子 n 是最大的。最开始，所有的盘子都在杆子 1 上，盘子 n 在最下面，上面是盘子 n－1，以此类推，盘子 1 在最上面。目标是将所有的盘子移到杆子 2 上。每次只能移动一个盘子，也就是说，任何一个杆子最上面的盘子只能移到另外两个杆子的顶端。此外，还不能将大盘子放置在小盘子上。

下面的 C 程序会对你用汇编语言编程有所帮助。

```
/* move n smallest disks from start to finish using
extra */

void hanoi(int n, int start, int finish, int extra){
    if(n != 0){
        hanoi(n-1, start, extra, finish);
        print_string("Move disk");
        print_int(n);
        print_string("from peg");
        print_int(start);
        print_string("to peg");
        print_int(finish);
        print_string(".\n");
        hanoi(n-1, extra, finish, start);
    }
}
main(){
    int n;
    print_string("Enter number of disks)");
    n = read_int();
    hanoi(n, 1, 2, 3);
    return 0;
}
```

A-83

# 逻辑设计基础

我一直很喜欢这个词：布尔。

——*Claude Shannon, IEEE Spectrum, April 1992*

(Shannon 的硕士论文表明，由 *George Boole* 在 19 世纪初发明的代数可以代表电器开关的工作原理。)

## B.1 引言

本附录仅对逻辑设计的基本原理进行了讨论，无法替代逻辑设计的课程，也不能保证你可以设计出可以很好工作的逻辑系统。如果你很少接触，甚至从没接触过逻辑设计，本附录将提供足够的背景知识，让你了解本书中提到的内容。另外，本附录将帮助你了解计算机内部的实现机制。如果你对该部分内容感兴趣，附录后面的参考文献还可以为你提供更多的信息。

B.2 节介绍了逻辑块中的基本单元：门。B.3 节中使用这些逻辑块来构建一个不含存储器的简单组合逻辑。如果你对逻辑电路或数字电路有所了解，前两部分将不会感到陌生。B.5 节讲述了怎样利用 B.2 节和 B.3 节的概念设计一个 MIPS 处理器的 ALU。B.6 节讲述了怎样设计一个快速加法器，如果对该部分不感兴趣，直接跳过即可。B.7 节简单介绍了时钟，如果想知道存储器如何工作，必须对时钟有所了解。B.8 节介绍了存储器的基本单元，B.9 节介绍了随机存取存储器，这两部分不仅介绍了存储器的特点，而且讲述了构建多层存储体系的背景知识。其中了解存储器的特点对如何使用存储器有很大帮助，详细内容在第 4 章介绍，构建存储体系在第 5 章进行介绍。B.10 节介绍了如何设计和使用时序逻辑块——有限状态机。如果你要掌握附录 D 的内容，那么你需要了解 B.2 – B.10 节所有的内容。如果你只要掌握第 4 章的知识，则可以直接跳到 B.11 节。B.11 节是为需要深入了解时钟方法和时序的人准备的，这一部分介绍了边缘触发时钟的工作原理，引入了另一种时钟策略，并且介绍了异步输入的同步问题。

B-1
B-3

在附录 B.4 节中，介绍了怎样用 Verilog 来描述逻辑电路。同时，在 CD 中可以找到更加广泛和完整的 Verilog 教程。

## B.2 门、真值表和逻辑方程式

现代计算机的内部电路为数字电路。数字电路仅工作在两个电压：高电压和低电压。其他所有的电压值均为瞬时值，出现在电压值间的过渡阶段。(正如后面要讲到的，数字电路设计中可能存在一个陷阱，当无法确定电压值属于高电压还是低电压时，则对该电压进行采样。)计算机采用二进制数也是促使其采用数字电路进行设计的一个重要原因，因为二进制可以和数字电路中的底层抽象相匹配。在不同的逻辑大家庭中，两个电压值间的关系和对应的值均有所不同。因此，我们不去谈论电压值的高低，而只需考虑电压值对应的逻辑值信号 0 和 1。其中逻辑 1 也称为“真”或**有效信号**（asserted signal），逻辑 0 也称为“假”或**无效信号**（deasserted signal）。逻辑 0/1 也称为 1/0 的补逻辑值或反逻辑值。

- **有效信号**：信号为逻辑 1 或真。
- **无效信号**：信号为逻辑 0 或假。

根据是否包含存储器件，逻辑电路被分为两大类。不包含存储器件的逻辑电路称为组合逻辑，组合逻辑的输出只取决于当前的输入。而包含有存储器件的电路中，输出不仅与当前的输入有关，而且与存储器件中存储的值有关，将存储器件中存储器的值称为逻辑电路的状态。在B.2节和B.3节中，我们只介绍**组合逻辑**（combinational logic）。在B.8节中介绍完各种存储元件后，我们再介绍包含电路状态的**时序逻辑**（sequential logic）。

B-4

- 组合逻辑：组合逻辑不包含存储元件，因此当输入相同的数据时，将得到相同的输出。
- 时序逻辑：时序逻辑包含有存储元件，因此输出取决于输入和当前存储元件的内容。

## B.2.1 真值表

由于组合逻辑不包含存储元件，因此可以对每一个可能的输入集，定义对应的输出值，通过这种方法就可以指定一个逻辑电路。通常我们用真值表来描述组合逻辑。对一个包含$n$个输入的组合电路来说，有$2^n$种可能的输入组合，因此真值表中有$2^n$项。真值表中的每一项都指定了特定输入对应的所有输出的数值。

**例B.1 例题：真值表**

假设一个逻辑函数包含三个输入$A$、$B$、$C$和三个输出$D$、$E$、$F$。函数的定义如下：如果有一个输入为真，则$D$为真；如果有两个输入为真，则$E$为真；如果三个输入都为真，则$F$为真。请写出该函数的真值表。

**例B.1 答案**

真值表包含$2^3 = 8$项。如下所示：

| 输入 | | | 输出 | | |
|---|---|---|---|---|---|
| $A$ | $B$ | $C$ | $D$ | $E$ | $F$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |

□

真值表可以描述任意的组合逻辑函数，但是真值表中的项数随着输入的增加将变得很大，同时真值表也不容易理解。有时，我们需要构造一个逻辑函数，其中很多输入组合均为0，此时我们可以只描述非0的输出组合。这种方法在第4章和附录D中使用。

B-5

## B.2.2 布尔代数

描述组合逻辑函数的另一种方法是使用逻辑方程式，这可以通过使用布尔代数（以19世纪数学家布尔的名字命名）来完成。在布尔代数中，所有的变量均取值为0或1，在典型的表达式中，包含如下三种操作符：

- 或操作可以记为+，例如$A + B$。或操作的结果如下：如果任意一个变量为1，则或操作的结果为1。由于任一变量为1，或操作结果都为1，因此或操作也被称为逻辑和。

- 与操作可以记为·，例如$A \cdot B$。只有当所有输入均为1时，与操作的结果才为1。由于所有输入为1时，与操作结果才为1，因此与操作也称为逻辑乘。
- 非操作可以记为$\overline{A}$，如果输入为0，则非操作的结果为1。非操作将会对逻辑值进行取反操作（如果输入为0，则输出为1，反之亦然）。

布尔代数中有几条定律，对逻辑方程式的操作很有帮助。

- 同一定律：$A+0=A$，$A \cdot 1=A$
- 0和1定律：$A+1=1$，$A \cdot 0=0$
- 互补定律：$A+\overline{A}=1$，$A \cdot \overline{A}=0$
- 交换律：$A+B=B+A$，$A \cdot B=B \cdot A$
- 结合律：$A+(B+C)=(A+B)+C$，$A \cdot (B \cdot C)=(A \cdot B) \cdot C$
- 分配律：$A \cdot (B+C)=(A \cdot B)+(A \cdot C)$，$A+(B \cdot C)=(A+B) \cdot (A+C)$

另外，还有两条很有用的定律，称为德·摩根定律，德·摩根定律将在练习题中进行深入介绍。

任何逻辑方程式组都可以写成一系列的逻辑等式，其中等式的左边为输出，等式右边为变量及上述三种操作符的组合。 B-6

**例1 例题·逻辑等式**

请写出上个例题中逻辑函数$D$、$E$、$F$的逻辑等式。

**例1 答案**

$D$的逻辑等式为：

$$D=A+B+C$$

$F$的逻辑等式为：

$$F=A \cdot B \cdot C$$

逻辑函数$E$需要一点技巧。将其分为两部分：$E$肯定为真的情况（三个输入中的两个必须为真），$E$肯定不会为真的情况（三个输入都不能为真）。由此$E$的逻辑等式可以描述为：

$$E=((A \cdot B)+(A \cdot C)+(B \cdot C))+(\overline{A \cdot B \cdot C})$$

我们也可以通过另一种方法得到$E$的逻辑等式。考虑到只有当两个输入为真时，$E$才为真，因此我们可以将$E$写成三个式子的或操作，其中每个式子为两个输入为真，一个输入为假的与操作，如下所示：

$$E=(A \cdot B \cdot \overline{C})+(A \cdot C \cdot \overline{B})+(B \cdot C \cdot \overline{A})$$

可以在练习题中验证两个逻辑等式是等价的。

在Verilog中，我们通过赋值声明来描述组合逻辑，这部分将在B.4节中进行描述。我们可以通过Verilog中的异或操作来定义$E$：`assign E=(A^B^C)*(A+B+C)^(A*B*C)`，这也是一种表示逻辑函数的方法。$D$和$F$的定义就更加简单了，与C语言的差别不大，如下所示：`D=A|B|C,F=A&B&C`。 □ B-7

## B.2.3 门

逻辑块是由可以实现基本逻辑功能的门（gate）构成的。例如，一个与门可以实现与操作，或门可以实现或操作。因为与和或操作是可交换、可结合的操作，因此与门、或门可以有多种输入，输出为所有输入的与、或操作。非操作通过一个反向器实现，反向器只有一个输入。这三种逻辑门的标准表示形式如图B-2-1所示。

⇨ 门：实现基本逻辑功能的硬件设备，比如与门、或门。

在描述非门时，更常见的形式并不是明确的画出反向器，而是在需要取反的输入或输出中加一个“气泡”（即小圆圈）。如图 B-2-2 所示，对于逻辑操作$\overline{A+B}$，左侧为使用反向器的表示形式，右侧为使用“气泡”的表示形式。

任何逻辑函数都可以通过与门、或门和非门来实现，有几个练习题要求使用门来实现一些通用逻辑函数。下一节中，我们将介绍如何通过这些门来实现任意的逻辑函数。



图 B-2-1 从左到右，依次为与门、或门、非门的标准表示形式。每个门的左侧信号为输入信号，右侧信号为输出信号。与门和或门有两个输入信号，非门只有一个输入信号

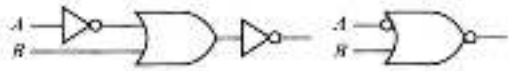



图 B-2-2 用逻辑门实现$\overline{A+B}$，左侧的输入输出均明确地画出了反向器，右侧则使用了“气泡”。该逻辑函数可以简化为$A \cdot \overline{B}$，或使用 Verilog 来表示 A & ~B

事实上，所有逻辑函数都可以通过单一的门来实现，只要这种门是反相的。两种常见的门为或非门（NOR gate）和与非门（NAND gate），其中或非门是对或门的输出进行取反操作，与非门是对与门的输出进行取反操作。或非门和与非门称为万能门，因为任何逻辑函数都可以通过其中的一种门来实现。下面的练习题将进一步探索这种观点。

⇨ 或非门：或门的输出进行取反。

⇨ 与非门：与门的输出进行取反。

**小测验**

下面的两个逻辑表达式等价吗？如果不等价，证明它们不等价。

- $(A \cdot B \cdot C) + (A \cdot \overline{C} \cdot B) + (B \cdot C \cdot A)$
- $B \cdot (A \cdot \overline{C} + C \cdot \overline{A})$



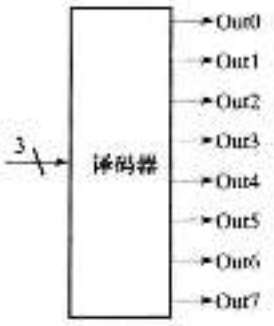
## B.3 组合逻辑

本节将介绍经常用到的较大的逻辑块。同时，我们将讨论结构化逻辑块的设计，这些逻辑块可以通过一种翻译程序，自动由逻辑等式或真值表来实现。最后，我们将讨论逻辑块组成的阵列。

### B.3.1 译码器

在设计大型逻辑单元中，**译码器**（decoder）是经常用到的一种逻辑块。最常见的译码器有 $n$ 个输入，$2^n$ 个输出，对每一种输入组合，只有一个输出信号置为 1。译码器将输入的 $n$ 位数据转化为该数据对应的二进制形式。译码器的输出常通过数字来标记，如 Out0、Out1…Out$2^n-1$。如果输入数据对应的值为 $i$，则 Out$i$ 被置为 1，其他所有的输出信号均为 0。图 B-3-1 为一个 3 位的译码器及对应的真值表。由于这种译码器有 3 个输入和 8 个输出，因此也成为 3-8 译码器。相对于译码器，编码器的功能正好相反，编码器有 $2^n$ 个输入和 $n$ 个输出。

⇨ 译码器：拥有 $n$ 位输入和 $2^n$ 输出的逻辑块。对每一种输入组合，只有一个输出位号为真。




a) 3位译码器

| 输入 | | | 输出 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| I2 | I1 | I0 | Out7 | Out6 | Out5 | Out4 | Out3 | Out2 | Out1 | Out0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

b) 3位译码器的真值表

图 B-3-1 一个3位译码器包含3个输入（I2、I1、I0）和8个输出（Out0 – Out7）。正如真值表所示，只有与输入信号对应的二进制数据相关的输出被置为1。译码器输入端的3表示输入信号为3位位宽

B-9

## B.3.2 多路选择器

第4章中，我们经常用到一个逻辑块：多路选择器。多路选择器之所以称为选择器，是因为它的输出来自输入信号中的某一个，而到底来自哪一个输入信号由控制信号决定。下面考虑两输入多路选择器。图 B-3-2 中，左侧包含三个输入：两个数据信号和一个选择（控制）信号（selector（control）value）。其中控制信号决定哪一个输入信号将成为输出信号。图 B-3-2 右侧的两输入多路选择器对应的逻辑函数为 $C = (A \cdot S) + (B \cdot S)$。

➲ 选择信号：也称为控制信号。控制信号用来选择某一个输入信号，来作为多路选择器的输出信号。

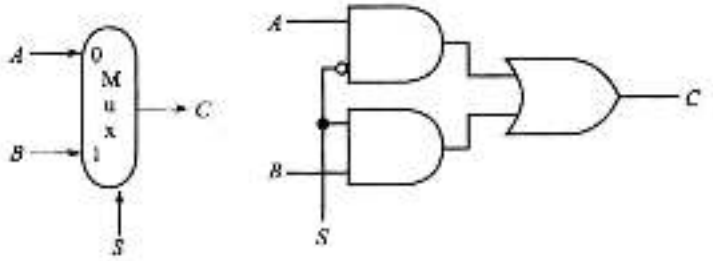



图 B-3-2 左侧为两输入的多路选择器，右侧为对应的实现。多路选择器包含两个输入（A 和 B），分别标记为0和1，并且包含一个选择输入信号（S）和一个输出信号（C）。用 Verilog 来实现多路选择器需要较多的工作量，尤其是当输入信号数量大于2时，我们将在 B.4 节中进行介绍

多路选择器可以有任意数量的输入信号。如果多路选择器只有两个输入信号，则当选择信号为1时，选择其中的一个输入作为输出；如果选择信号为0，则选择另一个输入作为输出。如果有 $n$ 个数据输入，则需要 $\lceil \log_2 n \rceil$ 个选择信号，此时的多路选择器包含三个部分：

1）生成 $n$ 个信号的译码器，每一个信号代表一个不同的输入信号。

2）$n$ 个与门阵列，每个与门都包含 $n$ 个信号中的一个。

3）一个较大的或门，用来将与门的输出进行合并。

为了将输入信号与控制信号联系起来，我们经常将输入数据用数字进行标记（如0，

1，2，…，n－1），同时将控制信号转化为二进制形式。有时，我们也使用未解码的选择信号。

在 Verilog 中，通过 if 语句可以很简单地实现多路选择器。对于大型的多路选择器，使用
B-10 case 语句将更加方便，但是在对组合逻辑进行综合的时候，需要十分小心。

### B.3.3 两级逻辑和 PLA

如上一节所述，任何逻辑函数都可以通过与、或和非门实现。事实上，还有更加规整的实现方法。任意的逻辑函数都可以描述成规范形式，即输入信号要么为真，要么为假，并且只有两级门：与门和或门，如果需要，可以在最后的输出中加一个反向器。这类表示法称为两级表示法，它有两种形式：乘积和（sum of products）、和项积。乘积和表示所有乘积（即与操作）的逻辑和（即或操作）；和项积正好相反。在前面的例子中，输出 $E$ 有两种形式：

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

和

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

其中第二个表达形式即为乘积和；它包含两级逻辑，并且非操作只发生在单个变量上面。第一个表达形式包含三级逻辑。

⇨ 乘积和：一种逻辑表达形式，即对所有乘积（由与操作实现）进行逻辑求和（或操作）。

**精解** 我们也可以将 $E$ 写成和项积的形式：

$$E = \overline{(A + \overline{B} + C)} \cdot \overline{(\overline{A} + \overline{C} + B)} \cdot \overline{(\overline{B} + C + A)}$$

为了得到这种表达形式，需要使用德·摩根定律，德·摩根定律在附录 A 中讨论。

在这本书里，我们使用乘积和的形式。显而易见，对于任何逻辑函数来说，我们都可以从它的真值表中构造出乘积和的形式。真值表中该函数为 1 的表项对应一个乘积项。乘积项为所有输入的乘积或输入取反后的乘积，是否取反取决于真值表中该变量对应的信号是 1 还是 0。
B-11 通过一个例子可以更容易理解。

**例题** 乘积和

填写出下面真值表中 $D$ 的乘积和的表达式。

| 输入 | | | 输出 |
|---|---|---|---|
| $A$ | $B$ | $C$ | $D$ |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

**答案**

由于真值表中有 4 个表项对应的 $D$ 为 1，因此总共有 4 个乘积项，如下：

$$\overline{A} \cdot \overline{B} \cdot C$$
$$\overline{A} \cdot B \cdot \overline{C}$$
$$A \cdot \overline{B} \cdot C$$
$$A \cdot B \cdot C$$

由此，我们可以写出 $D$ 的乘积项和的形式：

$$D = \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot \overline{C} + A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot C$$

需要注意到真值表中只有 $D$ 为 1 的表项，才能生成对应的乘积项。

可以利用真值表和两级门表示方法之间的关系，为任何逻辑函数生成一个门级的实现。一个真值表对应的逻辑函数集包含多个输出列，正如在 B.2 节中看到的一样。每一个输出列都对应一个不同的逻辑函数，都可以直接从真值表中构造出来。

乘积和的表示方法对应一种常见的称为**可编程逻辑阵列**（Programmable Logic Array，PLA）的结构化逻辑实现方法。PLA 包含一组输入、输入取反的信号（通过反向器来实现）和两级逻辑。第一个逻辑是一个与门阵列，用来生成**乘积项**（product term）（也称为**最小项**（minterm）），每一个乘积项都由输入信号或对应的反向信号构成。第二级为一个或门阵列，每一个或门都生成任意数量的乘积项对应的和。图 B-3-3 显示了 PLA 的基本构成。 □ B-12

➲ 可编程逻辑阵列（PLA）：是一种结构化逻辑单元。PLA 由一组输入信号及反向信号和一个两级逻辑构成。其中第一级逻辑用来生成输入信号和反向信号的乘积项，第二级逻辑用来生成这些乘积项的和。因此，PLA 的逻辑功能为实现乘积项的和。

➲ 最小项：也称为乘积项。由一组输入信号通过与操作形成。乘积项形成了 PLA 的第一级逻辑。

通过多个输入和多个输出，一个 PLA 可以直接实现真值表的功能，其中真值表看作一组逻辑函数。真值表中输出为真时，就产生一个乘积项，PLA 中就需要对应地生成一行。真值表中每一个输出都与或门阵列中潜在的某一行对应。或门的数量与真值表中输出为真的数量相对应。如图 B-3-3 所示，PLA 的大小等于与门阵列和或门阵列的大小之和。通过观察图 B-3-3 可以发现，与门阵列的大小等于输入信号的数量乘以不同乘积项的数量，或门阵列的大小等于输出信号的数量乘以乘积项的数量。

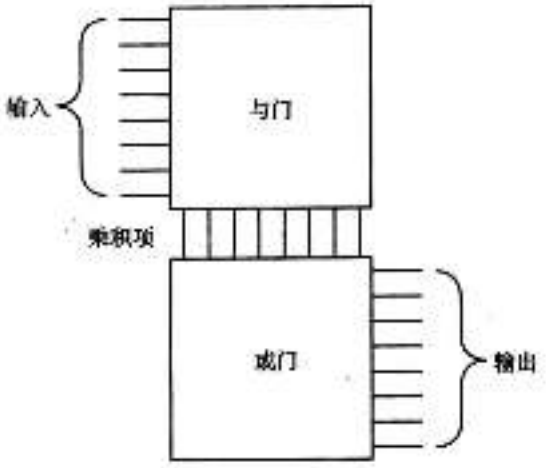



图 B-3-3 PLA 由一个与门阵列和紧跟的或门阵列构成。与门阵列的每一个输入都是若干输入信号或其反向信号的乘积。或门阵列的每一个输入为若干数量的乘积项的和

PLA 有两个特点，这两个特点使 PLA 成为实现逻辑函数组的有效方法。首先，真值表的每一项中，至少一个输出为真时，才需要对应的逻辑门。其次，在 PLA 中，不同的乘积项只对应一个输入，即使该乘积项被多个输出使用也不例外。下面让我们来看一个例子。

**例题 • PLA**

考虑 B.2 节中定义的一组逻辑函数。写出 $D$、$E$、$F$ 的 PLA 实现方法。 B-13

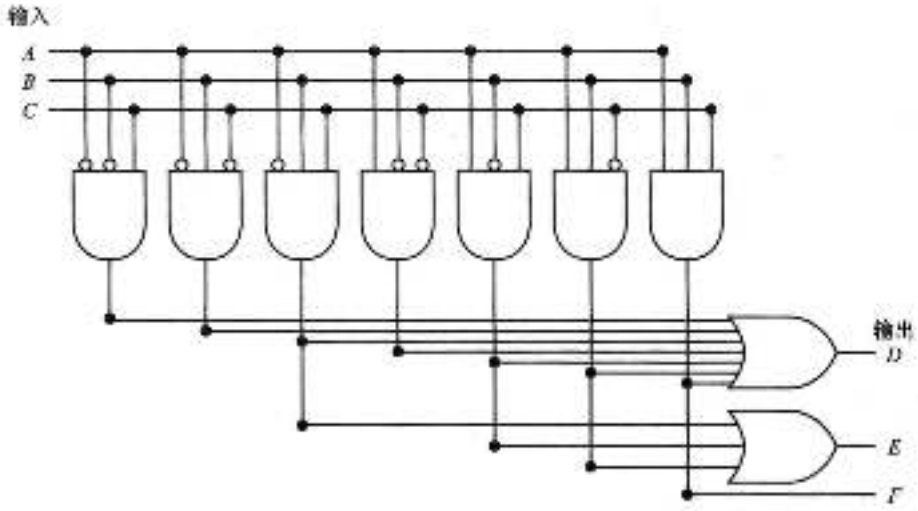
**答案**

这是我们前面构造的真值表。

| 输入 | | | 输出 | | |
|---|---|---|---|---|---|
| A | B | C | D | E | F |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |

由于真值表中至少有一个输出为1的表项有7个，因此与门阵列将有7列。与门阵列中行数为3（因为共有3个输入信号），同时或门阵列中也将包含3行（因为共有3个输出信号）。图B-3-4为最终的PLA，其中的乘积项与真值表中自顶向下的表项相对应。 □

图B-3-4将所有的门都画了出来，事实上，设计者常常只画出与门和或门的位置。当乘积项对应的信号线与输入信号或输出信号交叉时，需要使用点来标注。图B-3-4中的PLA使用这种方法时，结果如图B-3-5所示。当PLA被创建时，PLA的功能就固定下来了。也存在类似PLA结构的逻辑块，称为PALs，当设计者需要时，可以通过电子编程的方式来使用PALs。




图B-3-4 例题中逻辑函数对应的PLA实现结果

## B.3.4 ROM

另一类可以实现一组逻辑函数的结构化逻辑叫作只读存储器（Read-Only Memory，ROM）。ROM被称为存储器是因为它包含一组可以进行读操作的位置，然而，这些位置的内容是固定的，一般在制造的时候就固定下来。除此之外，还有一种可编程只读存储器（Programmable ROM，PROM），这类存储器可以电写入。还存在可擦除可编程只读存储器，这类设备需要一个缓慢的擦除过程，过程中需要使用紫外线，因此除了设计和调试外，这类设备只用做只读存储器。

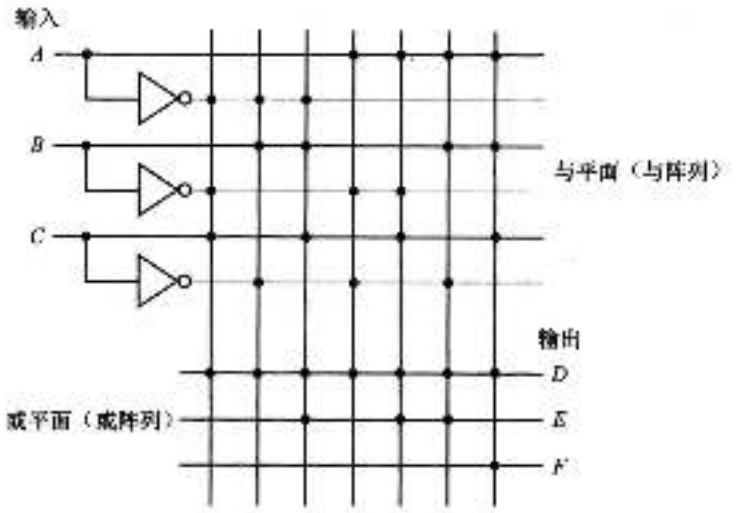



图 B-3-5 在矩阵中，用点来表示乘积项及这些乘积项之和的 PLA 结构。这里，在门的选择上，不是用反相器，而是所有输入信号以实际值及其补码形式连接到与阵列的每个输入上。与阵列中的一个点表示该输入，或其相反值在该乘积项中存在。或阵列中的一个点表示相应的乘积项出现在相应的输出上

- 只读存储器（ROM）：一类存储器，它的数据在制造时就固定下来，之后其数据只能被读。ROM 作为结构化逻辑，可以将逻辑函数组中的项作为输入地址、将输出作为存储器中的一个字，以此来实现逻辑函数组。
- 可编程 ROM（PROM）：一类只读存储器，但是当设计者知道其中的数据时，可以对其进行编程。

ROM 包含一组地址输入线和一组输出。ROM 可寻址的入口数量决定了地址线的数量：如 B-14
果 ROM 包含 $2^m$ 个可寻址入口（称为高度），则需要 $m$ 条地址线。每一个可寻址入口包含的二进制数量等于输出信号数量，有时也成为 ROM 的宽度。ROM 中总的二进制数量等于高度乘以宽度。有时将高度和宽度统称为 ROM 的形状。

ROM 可以直接通过真值表对逻辑方程式组进行编码。例如，对于有 $m$ 个输入的 $n$ 个方程组来说，ROM 需要 $m$ 条地址线（$2^m$ 个入口），其中每一个入口都为 $n$ 位宽。真值表中的输入代表着 ROM 中地址的入口，同时，真值表中的输出代表着 ROM 中存储的内容。当真值表中的输入序列构成一个二进制序列时（正如我们展示的所有真值表一样），ROM 的输出也将是有序的。在 B. 3 节的例子中，共有 3 个输入和 3 个输出，因此 ROM 有 $2^3 = 8$ 个入口地址，每一个入口包含 3 位二进制数据。ROM 中入口对应的数据量可直接由上面例子中的真值表中的输出得到。

ROM 和 PLA 间联系很密切。ROM 是完全译码的：对每一个可能的输入组合，都会输出一个字。而 PLA 是部分译码的。这意味着 ROM 将比 PLA 包含更多的入口。如前面的真值表所 B-15
示，ROM 包含了所有可能的 8 个输入入口，而 PLA 只包含了 7 个乘积项。随着输入数量的增加，ROM 中的入口数量呈指数增长。与此相反，对实际的逻辑函数来说，乘积项数量的增长很缓慢（参考附录 D 中的例子）。ROM 和 PLA 间的这种差异，使得 PLA 成为实现组合逻辑函数更有效的方法。ROM 的优势在于，当输入、输出数量匹配时，ROM 可以实现任意的逻辑函数。这种优势使得当逻辑函数发生变化时，ROM 中的内容很容易就随之变化，原因在于 ROM 的大小不需要改变。

除了 ROM 和 PLA 外，现代的逻辑综合系统也将小的组合逻辑块转化为一系列门的组合，

自动完成布局布线。尽管这些门的组合占面积较大，但面积的开销仍然比景板的结构化 ROM 和 PLA 的面积开销要小，因此成为逻辑实现的首选方法。

对于设计个定制或半定制的集成电路来说，更常用的方法是使用现场可编程器件，我们将
B-16 在 B.12 节中讨论这些器件。

## B.3.5 无关项

在实现组合逻辑时，有时我们并不在乎某些输出的值，其原因可能是另一个输出为真，或者是输入组合的子集决定了输出的值。我们称这种情况为*无关项*。因为可以简化逻辑函数的实现，因此无关项很重要。

无关项包含两种类型：无关项的输出和无关项的输入，两者都在真值表中体现出来。当我们对一些输入组合产生的输出不太关心时，就产生了无关项的输出。这类输出在真值表中以 X 代替。当一个输出对于一些输入的组合来说属于无关项时，设计者或逻辑优化程序就可以自由地对这些输入产生的输出赋值为 1 或 0。当输出只取决于一部分输入时，就产生了无关项的输入，在真值表中也记为 X。

**例 B.1 例题：无关项**

考虑一个包含 $A$、$B$、$C$ 三个输入的逻辑函数，其定义如下：

- 不管 $B$ 的值为多少，只要 $A$ 或 $C$ 为真，则输出 $D$ 为真。
- 不管 $C$ 的值为多少，只要 $A$ 或 $B$ 为真，则输出 $E$ 为真。
- 虽然 $D$ 和 $E$ 都为真时，我们不关心 $F$ 的值，但是如果三个输入中一个为真，则输出 $F$ 为真。

请写出这个逻辑函数完整的真值表和带有无关项时的真值表。对每一个真值表，PLA 各需要多少个乘积项？

**例 B.1 答案**

下面是不带无关项的完整的真值表：

| 输入 | | | 输出 | | |
|---|---|---|---|---|---|
| $A$ | $B$ | $C$ | $D$ | $E$ | $F$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 |

B-17

这个真值表对应的 PLA 需要 7 个乘积项。带有无关项的输出时的真值表如下：

| 输入 | | | 输出 | | |
|---|---|---|---|---|---|
| $A$ | $B$ | $C$ | $D$ | $E$ | $F$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 |

（续）

| 输入 | | | 输出 | | |
|---|---|---|---|---|---|
| A | B | C | D | E | F |
| 0 | 1 | 1 | 1 | 1 | × |
| 1 | 0 | 0 | 1 | 1 | × |
| 1 | 0 | 1 | 1 | 1 | × |
| 1 | 1 | 0 | 1 | 1 | × |
| 1 | 1 | 1 | 1 | 1 | × |

当加入无关项的输入时，真值表可以被进一步简化，如下所示：

| 输入 | | | 输出 | | |
|---|---|---|---|---|---|
| A | B | C | D | E | F |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| × | 1 | 1 | 1 | 1 | × |
| 1 | × | × | 1 | 1 | × |

简化后的真值表对应的 PLA 只需要 4 个最小项，或者可以采用一个两输入的与门和三个或门来实现（其中两个或门包含三个输入，另一个包含两个输入）。而原始的真值表需要 7 个最小项，可能需要 4 个与门。 □

逻辑最小化对获得高效的逻辑实现很重要。对任一逻辑进行手工最小化的一个有效工具是卡诺图。卡诺图将真值表以图的形式表示出来，因此可以很容易看出哪些乘积项可以进行合并。但是，由于卡诺图的尺寸和其复杂性，对实际逻辑函数进行手工最小化是不太可能的。幸运的是，逻辑最小化的过程已经高度机械化，可以通过设计工具来完成。在最小化的过程中，设计工具利用了无关项的这个优势，因此，识别出哪些是无关项很重要。附录最后的参考文献中提供了更多的内容，包括逻辑最小化、卡诺图和逻辑最小化算法背后的原理。

### B.3.6 逻辑单元阵列

对数据的组合逻辑操作中，很多操作一次需要处理整个字（32 位二进制数）。因此我们常常构建一个逻辑单元的阵列，这可以通过将一个操作作用在整个输入的集合中来实现。大多数时候，我们在机器内部需要在一对总线中进行选择。总线（bus）是若干数据线的集合，这些数据线被当做单一的逻辑信号对待。（名词“总线”也用来表示一组由多个信号源共享使用的信号线。） B-18

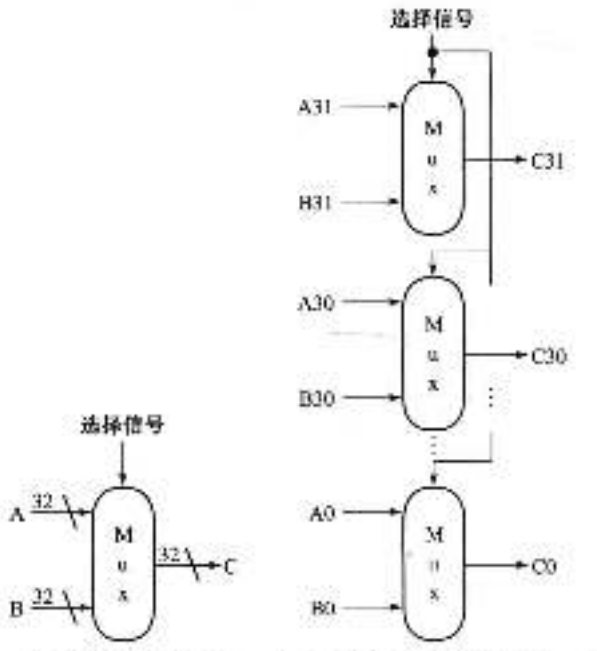
⊃ 总线：在逻辑设计中，由多个数据线构成的一条逻辑线，这些数据线被同时执行；被多个源使用的一组信号线也可以称为总线。

例如，在 MIPS 指令集中，指令运行的结果被写入寄存器中，而寄存器中的数据可能有一个或两个来源。此时，需要用一个多路选择器来决定哪一个总线上的数据（32 位）将被写入寄存器中。前面提到的 1 位多路选择器，在这里需要被复制 32 次。

在画图时，为了区分信号线是总线还是一条 1 位的信号线，我们在信号线上画一条较粗的

线来表示总线。大多数总线都是32位宽，如果不是32位宽，就明确地写出其位宽。当一个逻辑单元的输入和输出为总线时，意味着逻辑单元必须被复制足够的次数来满足输入的位宽。图B-3-6显示了一个多路选择器，这个多路选择器在一对32位宽的总线间进行选择。同时，图中也显示了该多路选择器是如何通过1位多路选择器实现的。有时，我们需要构造逻辑单元的阵列，其中有些元件的输入来自于前面元件的输出。例如，多位宽的ALU就是这样构造的。在这一类例子中，我们必须明确地显示出如何构造更宽的阵列，因为此时阵列中的单个元件并不是独立存在的。正如32位宽多路选择器的例子中显示的那样。

B-19




a) 一个32位宽的2选1选择器　　b) 32位宽的选择器实际上是一个包含32个1位宽的选择器的阵列

图B-3-6 为了在两个32位宽的输入中进行选择，多路选择器需要被复制32次。需要注意，对所有32个1位多路选择器来说，只使用一位的数据选择信号

### 小测验

对于奇偶性校验函数来说，它的输出取决于输入中1的数量。对于偶校验函数来说，如果输入中1的数量为偶数，则输出1。假设用ROM来实现包含4位输入的偶校验函数，*A*、*B*、*C*、*D*中哪一个可以表示ROM中的内容？

| 地址 | *A* | *B* | *C* | *D* |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 1 |
| 5 | 0 | 1 | 1 | 0 |
| 6 | 0 | 1 | 0 | 1 |
| 7 | 0 | 1 | 1 | 0 |
| 8 | 1 | 0 | 0 | 1 |
| 9 | 1 | 0 | 1 | 0 |
| 10 | 1 | 0 | 0 | 1 |
| 11 | 1 | 0 | 1 | 0 |
| 12 | 1 | 0 | 0 | 1 |
| 13 | 1 | 0 | 1 | 0 |
| 14 | 1 | 0 | 0 | 1 |
| 15 | 1 | 0 | 1 | 0 |

## B.4 使用硬件描述语言

当前，处理器和相关硬件系统的设计都是通过**硬件描述语言**（hardware description language）完成的。硬件描述语言有两个作用。首先，它提供了对硬件的一种抽象描述，通过这种描述可以对设计进行模拟和测试。其次，借助于综合工具和硬件汇编工具，硬件描述语言可以被编译成硬件的实现方法。

- 硬件描述语言：一种描述硬件的编程语言，硬件描述语言用来模拟硬件设计，同时也作为综合工具的输入来生成具体的硬件。

本节中，我们将介绍硬件描述语言 Verilog，并介绍如何使用 Verilog 来进行组合逻辑的设计。在附录的其他部分中，我们将 Verilog 的使用扩展到时序逻辑的设计上。在网站上第 4 章选读部分中，我们使用 Verilog 来描述处理器的实现。在网站上第 5 章的选读部分中，我们使用 System Verilog 来描述 cache 控制器的实现。System Verilog 为 Verilog 增加了一些结构和其他有用的特征。

Verilog 是两种基本硬件描述语言中的一种，另一种是 VHDL。Verilog 基于 C 语言，相对基于 Ada 的 VHDL 在工业界的使用更加频繁。对 C 语言比较熟悉的读者会发现，附录中我们用到的 Verilog 基本原理会很容易理解。如果对 C 语言的语法有所了解的话，对 VHDL 比较熟悉的读者将会发现 Verilog 的概念很简单。 B-20

- Verilog：两种常用硬件描述语言中的一种。
- VHDL：两种常用硬件描述语言中的一种。

Verilog 可以将数字系统在行为级和结构级进行描述。**行为级**（behavioral specification）描述方法描述了数字系统的功能特性。**结构级**（structural specification）描述方法描述了数字系统的详细组织，并且结构级描述常采用层次描述。结构级描述可以在基本元件的层次结构方面描述硬件系统，比如在门级和开关级。因此，我们可以使用 Verilog 来描述真值表的具体内容和最后章节的数据通路。

- 行为级描述：描述一个数字系统在功能方面是怎么工作的。
- 结构级描述：描述一个数字系统是如何通过基本元件的层次化连接进行组织的。

随着**硬件综合工具**（hardware synthesis tool）的出现，大多数设计者都使用 Verilog 或 VHDL，只对数据通路进行结构级描述，之后通过逻辑综合从行为级描述中生成控制系统。另外，大多数 CAD 系统都提供了广泛的标准元件库，如 ALU、多路选择器、寄存器、存储器和可编程逻辑块，当然也包含基本的门电路。

- 硬件综合工具：一种计算机辅助设计软件，该软件可以通过数字系统的行为级描述来生成门级的设计结果。

利用库和逻辑综合进行设计时，如果想得到可接受的结果，需要着眼于最终的综合及所需的输出，并据此来写描述语言。对于我们简单的设计来说，需要考虑的就是搞清楚哪些需要用组合逻辑来实现，哪些需要用时序逻辑来实现。在本节及剩余附录中的大部分例子中，写 Verilog 代码时，我们需要将最终的综合放在脑海里。

### B.4.1 Verilog 的数据类型和操作类型

Verilog 包含两种基本数据类型：

1）wire 表示一个组合信号。

2）reg（寄存器）存储一个数据，该数据随着时间的推移而变化。尽管实际实现中 reg 常常与一个寄存器相关连，但并不一定必须这样做。

- wire：在 Verilog 中表示一个组合逻辑信号。
- reg：在 Verilog 中表示一个寄存器。

假设有一个 wire 或 reg，命名为 X，当 X 为 32 位宽时，可以这样声明：reg[31:0] X 或 wire[31:0] X。通过最后的索引 0 来划定最低有效位。由于经常访问 reg 或 wire 的子字段，我们可以通过［起始位：结束位］访问 reg 或 wire 的一段连续位，其中的起始位和结束位必须为常数。

reg 的一组阵列可以用来表示寄存器文件或存储器。声明语句

```
reg [31:0] registerfile[0:31]
```

声明了一个类似于 MIPS 的寄存器文件，其中寄存器 0 是第一个寄存器。当访问存储阵列时，
B-21 我们可以使用 registerfile［regnum］访问一个数据，与 C 语言一样。

Verilog 中 reg 或 wire 型数据可能的取值有：

- 0 或 1，表示逻辑假或真。
- X，表示取值未知，所有寄存器初始化数据、未被连接的 wire 数据均为 X。
- Z，表示三态门处于高阻态，在该附录中不对其进行讨论。

常量可以被指定为十进制、二进制、八进制或十六进制。通常我们需要确切地知道一个常量包含多少二进制位。我们通过在常量前面加一个前缀来表示该常量包含多少二进制位，例如：

- 4'b0100 表示包含 4 位二进制常量的数据 4，等价于 4'd4。
- {A[31:16],B[15:0]}创建了一个数值，其中高 16 位来自 A，低 16 位来自 B。

Verilog 从 C 语言中继承了一元组和二进制操作符，包括算术运算符（+，-，*，/）、逻辑运算符（&，|，~），比较运算符（==，!=，>，<，<=，>=）、移位运算符（<<，>>）和 C 语言的条件运算符（使用格式为 condition? expr1:expr2，当 condition 为真时返回 expr1，否则返回 expr2）。Verilog 中增加了一组逻辑运算符（&，|，~），这类运算符对操作数的所有位均进行逻辑操作。例如，&A 返回 A 中所有位进行与操作的结果。^A 返回 A 中所有位异或的结果。

**小测验**

下面的定义中，哪些定义了相同的数值？

1. 8'b11111000
2. 8'hF0
3. 8'd240
4. {4{1'b1},4{1'b0}}
B-22 5. {4'b1,4'b0}

### B.4.2 Verilog 程序的结构

Verilog 程序是由模块的组合构成的。这些模块最小可以是一个逻辑门，最大可以是一个完

整的系统。Verilog 中的模块与 C++ 中的类类似，但没有类那样强大的功能。一个模块定义了它的输入和输出，输入和输出分别对应了模块与外部进行连接时的输入接口和输出接口。模块也可能声明一些附加的变量。一个模块的主体有以下几个部分构成：

- initial 结构，该结构对 reg 型变量进行初始化。
- 连续赋值语句，这类语句只出现在组合逻辑中。
- always 结构，该结构既可以用在组合逻辑中，也可以用在时序逻辑中。
- 模块实例化，该结构用来对已经定义的模块进行实例化操作。

### B.4.3 Verilog 构造复杂的组合逻辑

关键字 assign 表示连续赋值语句，连续赋值语句对应的组合逻辑函数为：输出被连续地赋值，并且只要输入的值发生变化，输出的值也马上发生变化。wire 型变量只能通过连续赋值语句进行赋值。通过连续赋值语句，我们可以定义一个模块来实现半加器，如图 B-4-1 所示。

使用 Verilog 来构造组合逻辑时，推荐使用连续赋值语句。但是，当需要构造更复杂的结构时，连续赋值语句将变得很笨拙和乏味。另一种可行方法是使用模块中的 always 语句，来描述组合逻辑单元，但是需要小心使用。always 语句中允许使用控制语句，比如 if-then-else、case 语句、for 语句和 repeat 语句。这些语句与 C 语言中的类似，只有少许变化。

```
module half_adder (A,B,Sum,Carry);
   input A,B; //two 1-bit inputs
   output Sum, Carry; //two 1-bit outputs
   assign Sum = A ^ B; //sum is A xor B
   assign Carry = A & B; //Carry is A and B
endmodule
```

图 B-4-1 使用连续赋值语句定义的一个半加器 Verilog 模块

always 语句块指定了一个信号列表，语句块对这些信号敏感（信号列表以@开始）。如果敏感信号列表中任一信号发生变化，always 语句块都将重新执行。如果省略了敏感信号列表，则 always 语句块将一直被不停地重新执行。当 always 语句块表示组合逻辑时，**敏感信号列表**（sensitivity list）需要包含所有的输入信号。如果 always 语句块中包含多条 Verilog 语句，这些语句将被关键字 begin 和 end 环绕，就像 C 语言中的{和}。一个 always 块如下所示：

B-23

```
always @(list of signals that cause reevaluation) begin
   Verilog statements including assignments and other
control statements end
```

⇒ 敏感信号列表：一些信号的列表，当这些信号中任一信号发生变化时，always 块都将重新执行。

reg 型变量只能在 always 块内部进行赋值，需要使用过程性赋值语句（与前面介绍的连续赋值不同）。总共有两种不同的过程赋值语句。其中操作符 = 与 C 语言中的类似，右侧语句计算出结果，并赋值给左侧变量。而且与 C 语言的执行一致，在下一个赋值语句执行前，该赋值语句完成执行。因此，操作符 = 被称为**阻塞性赋值**（blocking assignment）。阻塞性赋值对构造时序逻辑来说很有用，我们很快就会再次介绍它。另一种过程赋值语句为**非阻塞赋值**（nonblocking assignment），记为 <=。在 always 块中的非阻塞赋值语句，当所有非阻塞赋值语句计算出右侧的结果时，就立即赋值给对应的左侧变量。图 B-4-2 为 4 选 1 多路选择器的实现，该例子为使用 always 语句块实现的组合逻辑，为了简单化，我们使用了 case 语句。case 语句与 C 语言中的 switch 语句类似。图 B-4-3 显示了 MIPS 中 ALU 的实现，其中也使用了 case 语句。

⇒ 阻塞赋值：Verilog 中，在下一个赋值语句执行前，阻塞赋值完成执行。

⇒ 非阻塞赋值：一种赋值语句，只有计算出所有非阻塞语句右侧结果时，才进行赋值。

由于在always块中只能给reg型变量赋值，如果希望使用一个always块描述组合逻辑，则必须非常小心，以确保这个reg变量不被综合为一个寄存器。下面的"精解"中给出了许多"陷阱"。

```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);
   input [31:0] In1, In2, In3, In4; //four 32-bit inputs
   input [1:0] Sel; //selector signal
   output reg [31:0] Out;// 32-bit output
   always @(In1, In2, In3, In4, Sel)
   case (Sel) //a 4->1 multiplexor
      0: Out <= In1;
      1: Out <= In2;
      2: Out <= In3;
      default: Out <= In4;
   endcase
endmodule
```

图 B-4-2 使用case语句实现的4选1多路选择器，该多路选择器包含32位输入。case语句与C语言中的switch语句类似，不同之处在于，在Verilog中，只有被case选择到的语句会被执行（就好像每一个case状态后面都加了break一样），并且不能转到下一个分支去

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //recvaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

图 B-4-3 MIPS中ALU的Verilog行为级定义。通过使用包含基本算术和逻辑操作的模块库，就可以对其进行综合

**陷阱** **精解** 组合逻辑中经常使用连续赋值语句，但是在其他一些Verilog结构中，逻辑综合中可能会产生意想不到的结果，即使是always块也有这种问题。最常见的问题是，使用已经存在的锁存器或寄存器来实现时序逻辑，这将导致生成的结果比预期的要慢，并且开销要大。为了保证你设计的组合逻辑可以按这种方式被综合，你需要按下面的注意事项操作：

1）将所有的组合逻辑放在连续赋值语句或always块中。

2）保证作为输入的所有信号都出现在always块的敏感信号列表中。

3）保证每一个通过always块的数据通路，都将值赋给同一位组。

最后一点是最容易被忽略的。请读一下图B-5-15中的例子，来说服自己为何要坚持最后这条准则。

B-24 ~ B-25

**小测验**

假设所有变量都初始化为 0，下面的 Verilog 包含在 always 块中，则执行完下面的语句后，A、B 的值分别为多少？

```
C=1;
A <= C;
B = C;
```

## B.5 构建基本的算术逻辑单元

ALU：被所有计算机系统作为标准来使用的一个随机数生成器。

——*Stan Kelly-Bootle，魔鬼词典，1981*

**算术逻辑单元**（arithmetic logic unit，ALU）是计算机的核心，ALU 用来执行算术运算，比如加法和减法，也可以用来执行逻辑运算，比如与操作和或操作。本节中，我们将借助 4 个硬件块（与门、或门、反向器和多路选择器）来构造一个 ALU，并演示组合逻辑是如何工作的。下一节中，我们将展示如何通过更加聪明的设计来加速加法器的操作。

因为 MIPS 中一个字为 32 位宽，因此我们需要一个 32 位宽的 ALU。假设我们使用 32 个 1 位宽 ALU 来构建所需的 ALU。我们将从如何构建 1 位宽 ALU 开始。

### B.5.1 1 位 ALU

逻辑操作是最简单的，因为它们直接映射为图 B-2-1 中的硬件元件。

与和或对应的 1 位逻辑单元如图 B-5-1 所示。多路选择器选择是进行 a AND b 操作还是 a OR b 操作，如何选择取决于 Operation 的值为 0 还是 1。为了与数据信号线进行区分，多路选择器的控制信号画了颜色。需要注意的是，我们需要为多路选择器的控制线和输出线进行重命名，以便反映它们在 ALU 中的功能。

下一个需要加入的函数是加法。一个加法器必须包含两个输入操作数，并输出一位和。同时，需要另外一个输出来传递进位，称为 CarryOut。因为来自相邻加法器的进位是作为输入对待的，因此加法器需要第三个输入。这个输入称为 CarryIn。图 B-5-2 显示了一位加法器的输入和输出。我们知道加法操作的作用是什么，因此我们可以通过输入来指定对应的输出，如图 B-5-3 所示。

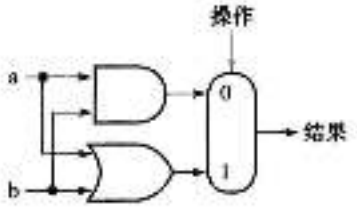



图 B-5-1 与和或的 1 位逻辑单元

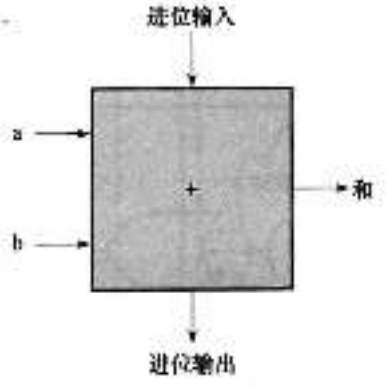



图 B-5-2 1 位加法器。该加法器称为全加器，它也称为（3，2）加法器，因为它有 3 个输入端和 2 个输出端。如果一个加法器只有 a 和 b 两个输入，则称为（2，2）加法器或半加器

| 输入 | | | 输出 | | 注解 |
|---|---|---|---|---|---|
| a | b | 进位输入 | 进位输出 | 和 | |
| 0 | 0 | 0 | 0 | 0 | $0+0+0=00_2$ |
| 0 | 0 | 1 | 0 | 1 | $0+0+1=01_2$ |
| 0 | 1 | 0 | 0 | 1 | $0+1+0=01_2$ |
| 0 | 1 | 1 | 1 | 0 | $0+1+1=10_2$ |
| 1 | 0 | 0 | 0 | 1 | $1+0+0=01_2$ |
| 1 | 0 | 1 | 1 | 0 | $1+0+1=10_2$ |
| 1 | 1 | 0 | 1 | 0 | $1+1+0=10_2$ |
| 1 | 1 | 1 | 1 | 1 | $1+1+1=11_2$ |

图 B-5-3 1 位加法器输入输出定义

我们可以用逻辑等式的方式来表示输出信号 CarryOut 和 Sum，这些逻辑等式又可以通过逻辑门来实现。以 CarryOut 为例，图 B-5-4 显示了当 CarryOut 为 1 时，对应输入的值。

我们可以将真值表转化为逻辑等式：

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

如果 a · b · CarryIn 为真，则剩余的三个乘积项也必然为真，因此我们可以根据真值表的第 4 行将最后一项省略掉。简化后的等式为：

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

| 输入 | | |
|---|---|---|
| a | b | 进位输入 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

图 B-5-4 当 CarryOut 为 1 时，各个输入的值

图 B-5-5 显示了加法器黑盒子内部的硬件，其中 CarryOut 由 3 个与门和一个或门组成。三个与门分别对应上式中括号内的乘积项，或门用来得到三个乘积项的和。

当有一个输入为 1 或三个输入都为 1 时，Sum 设置为 1。Sum 对应的布尔等式较为复杂（回乙一下，$\bar{a}$ 表示 NoT a），如下所示：

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

B-26 ~ B-28

如何画出加法器中 Sum 对应的逻辑，留给读者作为练习。

图 B-5-6 所示的是用之前的部件组成的加法器得到的 1 位 ALU。有时设计人员也希望 ALU 能完成再多一些简单的操作，比如生成 0。最简单增加一个操作的方法是扩大由操作线控制的多路选择器，例如，为了将 0 直接连到扩展的多路选择器的新输入端。

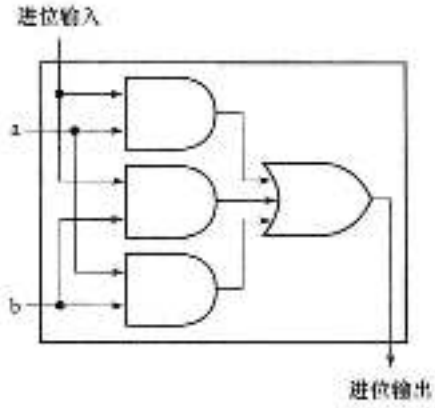



图 B-5-5 加法器中产生 CarryOut 信号所需的硬件。加法器硬件的剩余部分是本页等式中和（Sum）的输出逻辑

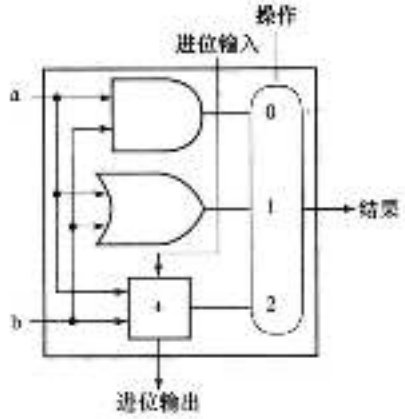



图 B-5-6 完成“与”、“或”和“加法”运算的一个 1 位 ALU（见图 B-5-5）

## B.5.2 32 位的算术运算单元（ALU）

现在我们已经实现了 1 位的 ALU，由临近的“黑盒子”连接成 32 位的 ALU。用 $x_i$ 表示 $x$ 的第 $i$ 位，图 B-5-7 所示的是一个 32 位的 ALU。犹如一块石头能使一个平静的湖激起涟漪，一个最不重要的进位（结果 0）能通过所有的加法器带来显著的进位（结果 31）。因此，通过直接连接 1 位进位的加法器称为行波进位加法器。从 B.6 节开始我们将看到一种更快连接 1 位加法器的方法。

减法就是将第二个源操作数变相取反后用加法实现。快速求一个二进制数补码的方法是，将这个数按位取反，然后加 1。为了反转每一位，我们只需在 b 和 $\overline{b}$ 之间添加用来选择的 2：1 多路选择器，如图 B-5-8 所示。

假设将 32 个 1 位的 ALU 连接到一起，如图 B-5-7 所示。所添加的多路转换器给出 b 的选项或它的反码，这取决于 Binvert，但是这仅是求二进制数补码的一个步骤。注意到，最低位仍然有一个 CarryIn 信号，即使它对加法是不必要的。如果我们用 1 代替 0 来设置 CarryIn 信号，将会发生什么？加法器会计算 $a + b + 1$。通过将 b 取反，就能得到我们想要的结果：

$$a + \overline{b} + 1 = a + (\overline{b} + 1) = a + (-b) = a - b$$

一个二进制补码加法器的简单设计有助于解释为什么二进制的补码表示已经成为整数计算机运算的通用标准。

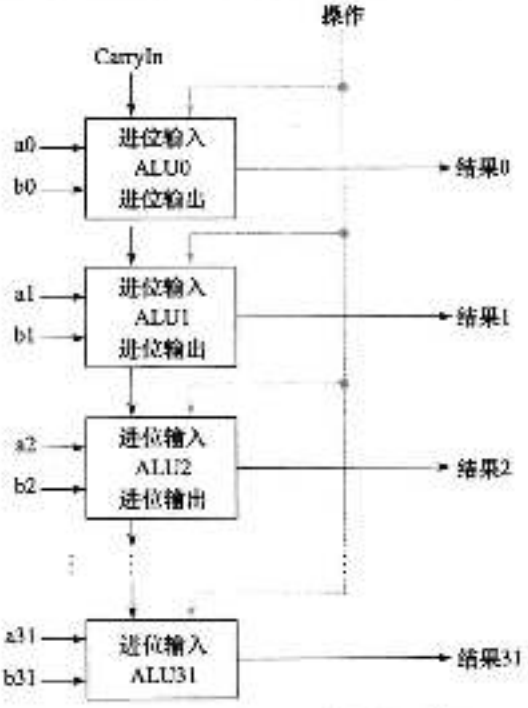



图 B-5-7 由 32 个 1 位 ALU 构成的 32 位 ALU。不太重要的位的 CarryOut 信号连接到较重要的位的 CarryIn 信号上，这种组成方式称为行波进位

B-29

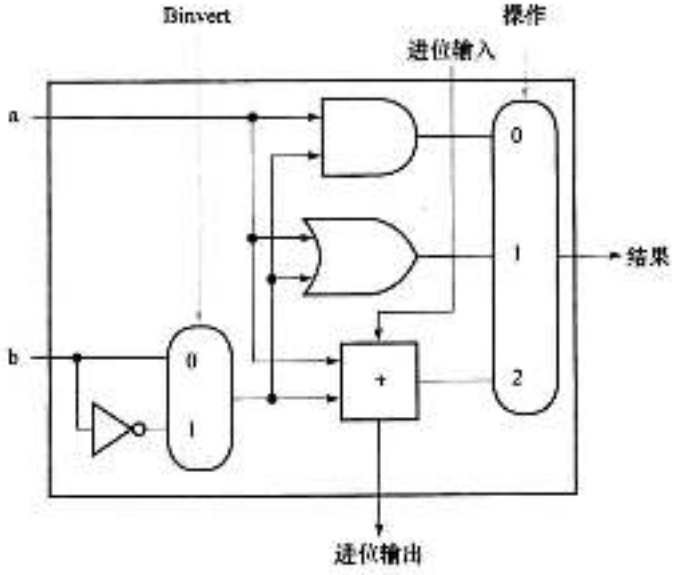



图 B-5-8 一个 1 位的 ALU 执行“与”、“或”以及加法 $a + b$ 和 $a + \overline{b}$。通过选择 $\overline{b}$（Binvert = 1），并在影响不显著的位上将 CarryIn 设置为 1，得到 b 的二进制补码减法，而不是 b 到 a 的加法

一个 MIPS ALU 需要或非（NOR）功能，我们可以通过重复使用 ALU 内部已经有的硬件来实现这种功能，而不是单独增加一个或非门。或非表达式表示如下：

$$\overline{(a+b)} = \bar{a} \cdot \bar{b}$$

即 a 或 b 的非和非 a 与非 b 是相等的，这也被称为德·摩根定律，在练习题中进行更加深入的探究。

ALU 上已经有了与门和非 b，只需要再增加非 a。图 B-5-9 所示的是改变后的结构。

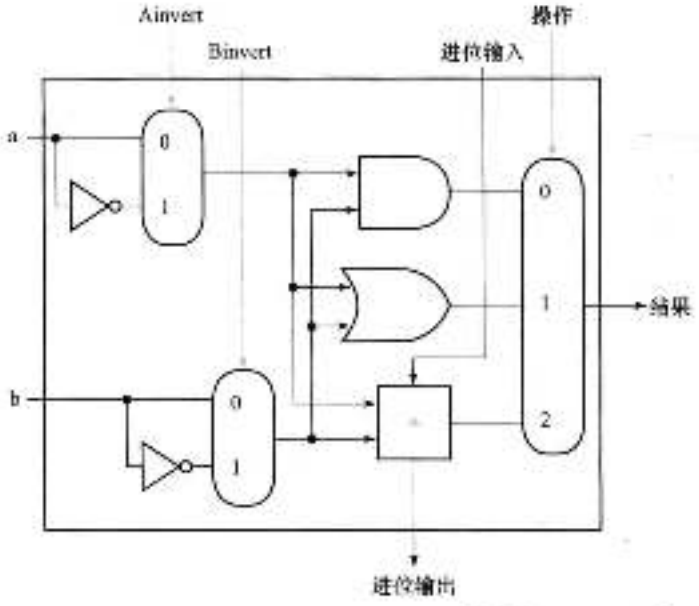



图 B-5-9　1 位 ALU 实现“与”、“或”以及 $a+b$ 或者 $\bar{a}+\bar{b}$，通过选择 $\bar{a}$（Ainvert = 1）和 $\bar{b}$（Binvert = 1），能得到 $\overline{a+b}$ 而不是 $a+b$

## B.5.3　将 32 位的 ALU 添加至 MIPS

加法、减法、与、或 4 个操作几乎在每一台计算机里面都能找到，并且大多数的 MIPS 指令都能由 ALU 实现，但是 ALU 的设计不是完美的。

B-30
↕
B-31

仍然需要支持的一条指令是 slt 指令，回想前面的内容，如果 rs < rt，操作过的记为 1，反之，结果为 0。因此，slt 指令根据比较将所有不显著的位都进行设置，但是都将它们设置为 0。为了让 ALU 执行 slt 指令，首先需要扩大二输入的多路选择器，在图 B-5-8 中为 slt 结果增加一个输入。我们称之为 Less，仅在 slt 中使用。

图 B-5-10 顶部的图所示的是右扩展的多路选择器的 1 位 ALU。从以上对 slt 的描述中，我们得知必须将上面 31 位 ALU 的输入端 Less 连接到 0，因为这些位经常被置为 0。剩下要考虑的是在 slt 指令中如何比较和设置最不显著的位。

如果用 a 减去 b 会发生什么？如果结果为负值，那么 a < b，因为

$$(a-b) < 0 \Rightarrow ((a-b)+b) < (0+b)$$

$$\Rightarrow a < b$$

如果 a < b，我们就把 slt 操作中最低位设置为 1；也就是说，a − b 为负数时结果为 1，为正时结果为 0。期望值完全对应的标志位置为：1 代表负值，0 代表正值。遵循这个论点，仅需要将加法器输出的标志位连接到最低位上，进而得到 slt。

不幸的是，图 B-5-10 顶部的图的 slt 操作中，ALU 输出的 Result 至最高位不是加法器的

输出；加法器 slt 操作输出明显是输入值 Less。

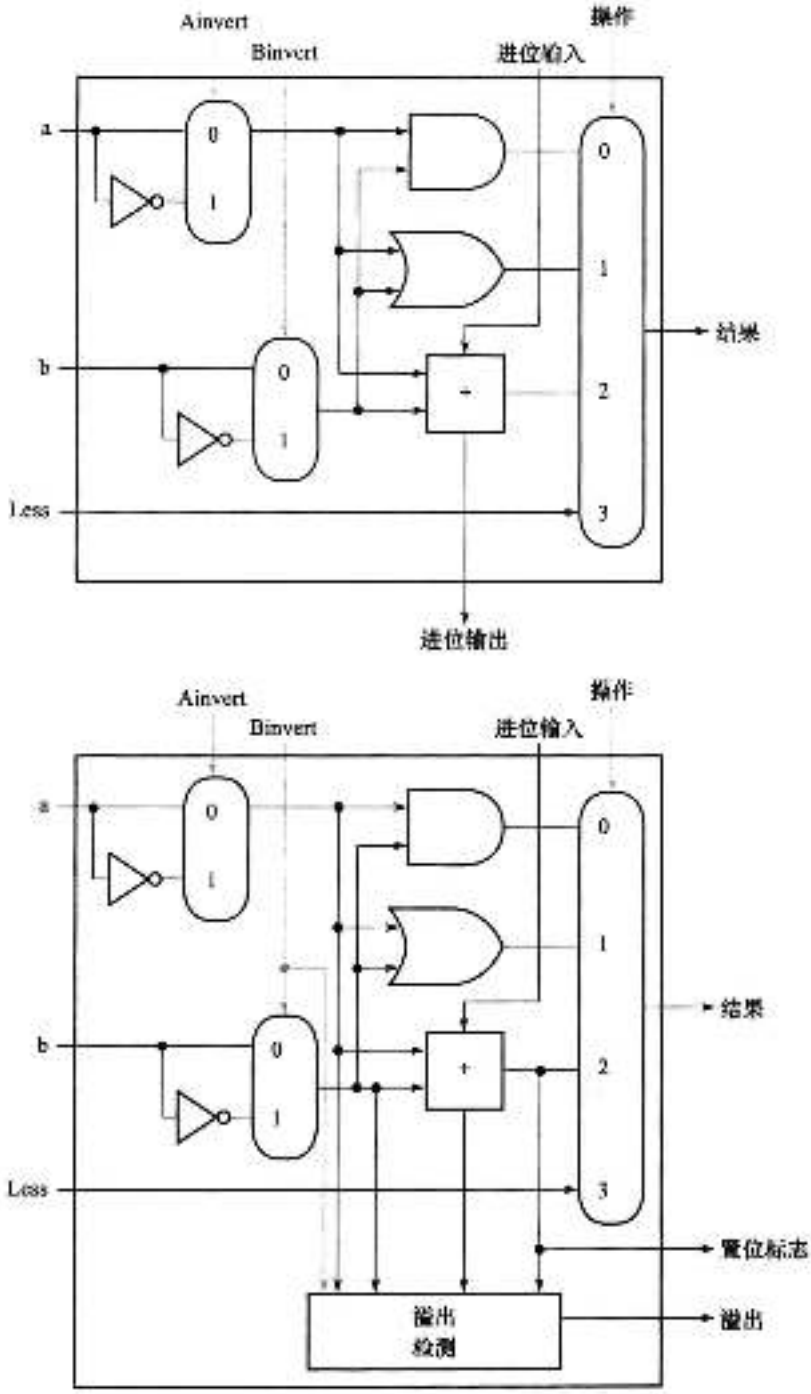



图 B-5-10 （顶部）1 位 ALU 执行“与”、“或”以及加法 $a+b$ 或者 $a+\overline{b}$，（底部）一个 1 位 ALU 最显著位。顶部的图包括连接到 slt 操作上的直接输入（见图 B-5-11）；底部的图有 Less 操作的直接输入，称为 Set（见附录末的练习题 B.24，了解如何用较少的输入计算溢出）

因此，最高位需要一个新的 1 位的 ALU，它有额外的输出位：加法器的输出。图 B-5-10 底部的图所示的设计是有新输出 Set 的加法器，并且仅用在 slt 中。我们只需要一个特殊的最高位，于是增加了溢出检测逻辑，因为它和最高位是关联的。

B-32
↓
B-34

由于溢出问题的存在，slt 的检测比之前描述的要更加复杂，如在练习题中探究的。

图 B-5-11 所示的是 32 位的 ALU。

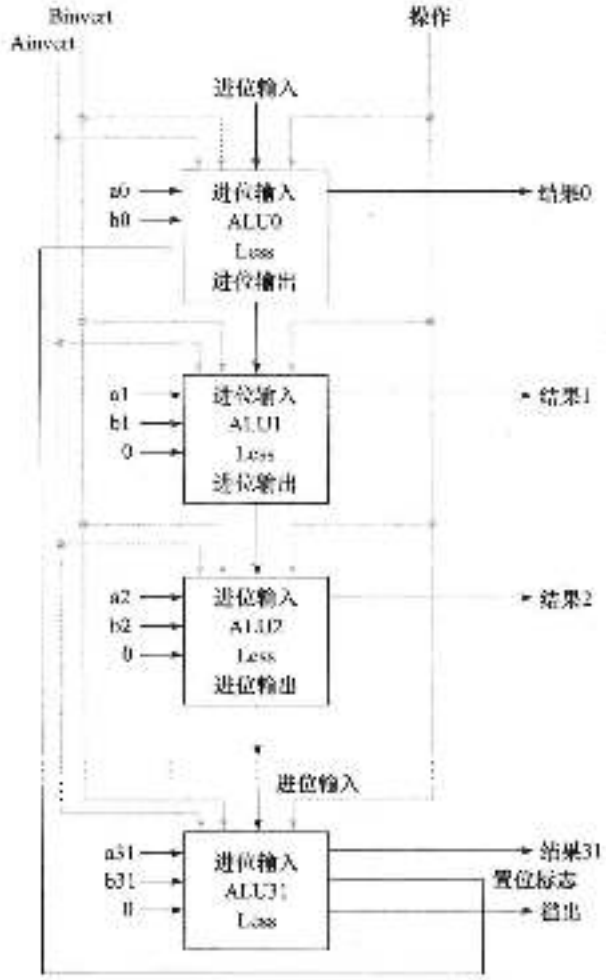



图 B-5-11 一个复制 31 个图 B-5-10 顶部结构和一个图 B-5-10 底部结构组成的 32 位 ALU。除了最低位，Less 输入都连接到 0，最低位连接至 Set 输出端的最高位。如果 ALU 执行 a − b，我们选择在图 B-5-10 中多路选择器的输入端为 3，如果 a < b，那么 Result = 0…001，其他 Result = 0…000

注意每次要 ALU 做减法运算，都要将 CarryIn 和 Binvert 信号置为 1。对于加法或者逻辑运算，我们想要控制线均为 0。因此可以通过将 CarryIn 和 Binvert 结合为一根控制线，称为 Bnegate，从而简化了 ALU 的控制。

为了使 ALU 更进一步适合 MIPS 指令集，必须支持条件分支指令。这些指令分支或者是两个寄存器相等，或者它们不相等。最简单的方法是用 ALU 测试 a − b，并测试其结果是否为 0，因为

$$(a - b = 0) \Rightarrow a = b$$

因此，如果增加硬件来测试结果是否为 0，就能测试是否相等。最简单测试方法是将所有的输出取“或”，然后将结果连接至反相器：

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \cdots + \text{Result2} + \text{Result1} + \text{Result0})}$$

图 B-5-12 所示的是修改后的 32 位 ALU。考虑将 1 位 Ainvert 线、1 位 Binvert 线以及 2 位的 Operation 线作为 4 位的 ALU 控制线结合在一起，执行加法、减法、与、或，或者 slt 指令。图 B-5-13 所示的为 ALU 控制线以及相应的 ALU 操作。

最后，我们已经看到一个 32 位的 ALU 的内部结构，我们将用通用的符号表示一个完整的 ALU，如图 B-5-14 所示。

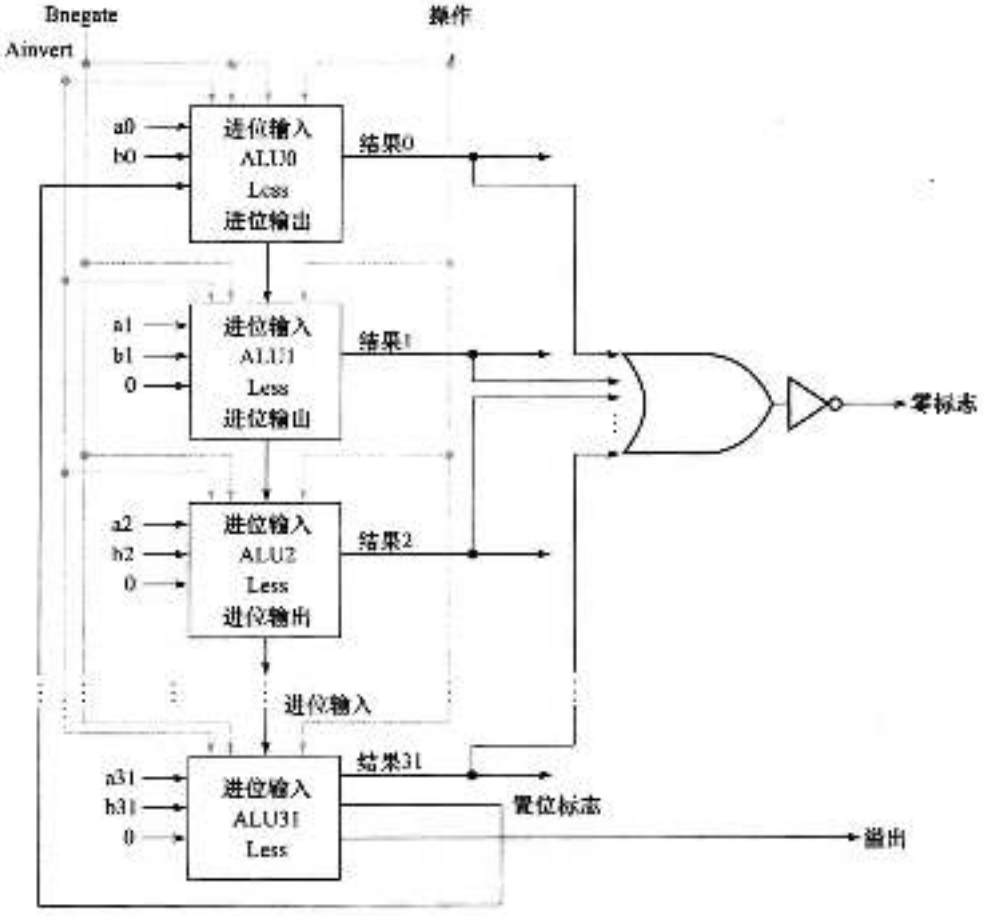



图 B-5-12 最终的 32 位 ALU。在图 B-5-11 的结构上增加了一个 0 检测器

| ALU控制线 | 功能 |
|---|---|
| 0000 | 与 |
| 0001 | 或 |
| 0010 | 加 |
| 0110 | 减 |
| 0111 | 小于则置位 |
| 1100 | 或非 |

图 B-5-13 ALU 三个控制线，Bnegate 和 Operation 的值以及对应的操作

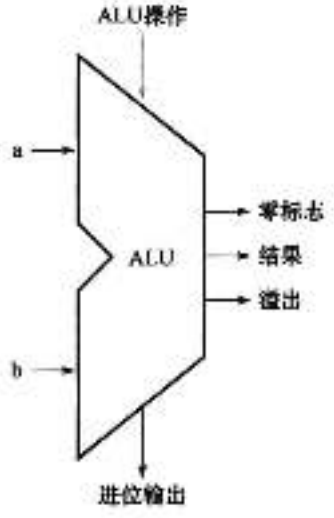



图 B-5-14 图 B-5-12 所示 ALU 的符号图。这个符号也被用来表示加法器，因此通常使用 ALU 或 Adder 标记

## B.5.4 用 Verilog 定义 MIPS ALU

图 B-5-15 所示的是如何用 Verilog 定义一个组合 MIPS ALU 的结合；这样的规范可能会用一个加法器的被实例化的标准库编译。为了完整性，我们在图 B-5-16 中展示了 MIPS 的 ALU 控制器（在第 4 章使用过），在这个控制器上我们建立了一个 Verilog 版本的 MIPS 数据通路。

```
module MIPSALU (ALUCtl, A, B, ALUOut, Zero);
   input [3:0] ALUCtl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut == 0); //Zero is true if ALUOut is 0
   always @(ALUCtl, A, B) begin //reevaluate if these change
      case (ALUCtl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
   end
endmodule
```

图 B-5-15 一个 MIPS ALU 的 Verilog 行为定义

下一个问题是，“ALU 将两个 32 位的操作数相加能有多快？”我们能决定 a 和 b 的输入，但是 CarryIn 的输入取决于相邻的 1 位加法器的操作。如果我们通过依赖关系跟踪所有的路径，将最高位连接到最低位上，所有和的最高位必须等待所有 32 个 1 位加法器来依次计算。这种顺序链的反应太慢以至于不能在时间关键的硬件电路中使用。下一节将探究如何加快加法的速度，这个论题对于理解附录的其余部分不是至关重要的，可以跳过。

B-35

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] reg ALUCtl;
   always case (FuncCode)
   32: ALUOp<=2; // add
   34: ALUOp<=6; //subtract
   36: ALUOP<=0; // and
   37: ALUOp<=1; // or
   39: ALUOp<=12; // nor
   42: ALUOp<=7; // slt
   default: ALUOp<=15; // should not happen
   endcase
endmodule
```

图 B-5-16 MIPS ALU 控制：一个简单的组合逻辑控制

**小测验** 假设想增加 NOT（a AND b）操作，称为与非（NAND），应如何修改 ALU？

1. 没有改变。你可以用当前的 ALU 快速计算出 NAND，因为 $\overline{(a \cdot b)} = \bar{a} + \bar{b}$，而且已经有 $\bar{a}$、$\bar{b}$ 以及或门。
2. 必须扩大多路选择器以增加另外的输入，然后增加新的逻辑电路来计算 NAND。

## B.6 快速加法：超前进位

提高加法器速度的关键是提高高阶进位的速度。有各种各样的方案来预测进位，最坏的情况是加法器中所有位数的对数的函数。由于进位经过的逻辑门较少，所以这些预期信号执行得

比较快，但是要预测到更准确的进位，需要增加更多的逻辑门。

理解快速进位的关键是要理解：无论输入何时改变硬件都是并行执行的，这一点与软件不同。

## B.6.1 使用“无限”硬件的快速进位

B-36 ↓ B-38

正如前面提到的，任何一个等式都能用两个逻辑层表示。因为只有外部输入是两个操作数和 CarryIn 到加法器的最低位，在理论上我们可以仅在两个逻辑层计算出到加法器所有剩余位的 CarryIn 值。

例如，CarryIn 为 2，CarryOut 恰好为 1，因此公式为

$$\text{CarryIn2} = (\text{b1} \cdot \text{CarryIn1}) + (\text{a1} \cdot \text{CarryIn1}) + (\text{a1} \cdot \text{b1})$$

类似地，CarryIn1 可以定义为

$$\text{CarryIn1} = (\text{b0} \cdot \text{CarryIn0}) + (\text{a0} \cdot \text{CarryIn0}) + (\text{a0} \cdot \text{b0})$$

用 ci 代替 CarryIni，上式改写为

$$c2 = (\text{b1} \cdot \text{c1}) + (\text{a1} \cdot \text{c1}) + (\text{a1} \cdot \text{b1})$$
$$c1 = (\text{b0} \cdot \text{c0}) + (\text{a0} \cdot \text{c0}) + (\text{a0} \cdot \text{b0})$$

将表达式 c1 带入第一次公式 c2，可得：

$$c2 = (\text{a1} \cdot \text{a0} \cdot \text{b0}) + (\text{a1} \cdot \text{a0} \cdot \text{c0}) \cdot (\text{a1} \cdot \text{b0} \cdot \text{c0}) + (\text{b1} \cdot \text{a0} \cdot \text{b0}) + (\text{b1} \cdot \text{a0} \cdot \text{c0}) + (\text{b1} \cdot \text{b0} \cdot \text{c0}) + (\text{a1} \cdot \text{b1})$$

可以想象一下，当加法器得到更高的位时，方程式会如何扩大？它将随着位数的增加而快速增加。这一复杂性可以反应在快速进位的硬件开销上，把这个简单结构广泛用于加法器过于昂贵。

## B.6.2 用第一级抽象快速进位：进位传播函数和进位产生函数

大多数的快速进位方法使用限制方程式的复杂性来简化硬件，同时与行波进位相比速度得到了大幅度提高。超前进位加法器（carry-lookahead adder）就是这样一种结构。在第 1 章中，已经介绍了计算机系统是用不同的抽象级别来配合其复杂性的。一个超前进位加法器依赖于不同的抽象级在其内部的实现。

首先考虑原始方程的各个因子：

$$ci + 1 = (\text{bi} \cdot \text{ci}) + (\text{ai} \cdot \text{ci}) + (\text{ai} \cdot \text{bi})$$
$$= (\text{ai} \cdot \text{bi}) + (\text{ai} + \text{bi}) \cdot \text{ci}$$

如果用这个公式重写 c2 的方程，我们将会看到一些重复的部分：

$$c2 = (\text{a1} \cdot \text{b1}) + (\text{a1} \cdot \text{b1}) \cdot ((\text{a0} \cdot \text{b0}) + (\text{a0} + \text{b0}) \cdot \text{c0})$$

注意到（ai · bi）和（ai + bi）在上面的公式中重复出现，这两个重要函数通常称为进位产生函数（gi）和进位传输函数（pi）：

B-39

$$\text{gi} = \text{ai} \cdot \text{bi}$$
$$\text{pi} = \text{ai} + \text{bi}$$

用它们来定义 ci + 1，可得：

$$ci + 1 = \text{gi} + \text{pi} \cdot \text{ci}$$

为了理解信号是从哪里得到的，假设 gi = 1，即

$$ci + 1 = \text{gi} + \text{pi} \cdot \text{ci} = 1 + \text{pi} \cdot \text{ci} = 1$$

也就是说，加法器生成的进位输出（ci + 1）和进位输入（ci）是独立的。假设 gi = 0，pi = 1，则

$$ci + 1 = \text{gi} + \text{pi} \cdot \text{ci} = 0 + 1 \cdot \text{ci} = \text{ci}$$

也就是说，加法器将进位输入传递到进位输出。将以上二者放在一起可得，gi = 1 或者 pi = 1 且 CarryIni = 1，可得 CarryIni + 1 = 1。

作为比喻，想象一排多米诺骨牌，通过推倒远处的一张牌可将最后一张推倒，假设两张牌之间没有间隙。类似地，一个进位可以通过生成因子而使其为真，只要它们之间所有的传递因子均为真。

根据传递因子和生成因子的定义，我们将其作为第一抽象级，能更加经济地描述进位输入信号。下面所示的是4位的：

$$c1 = g0 + (p0 \cdot c0)$$

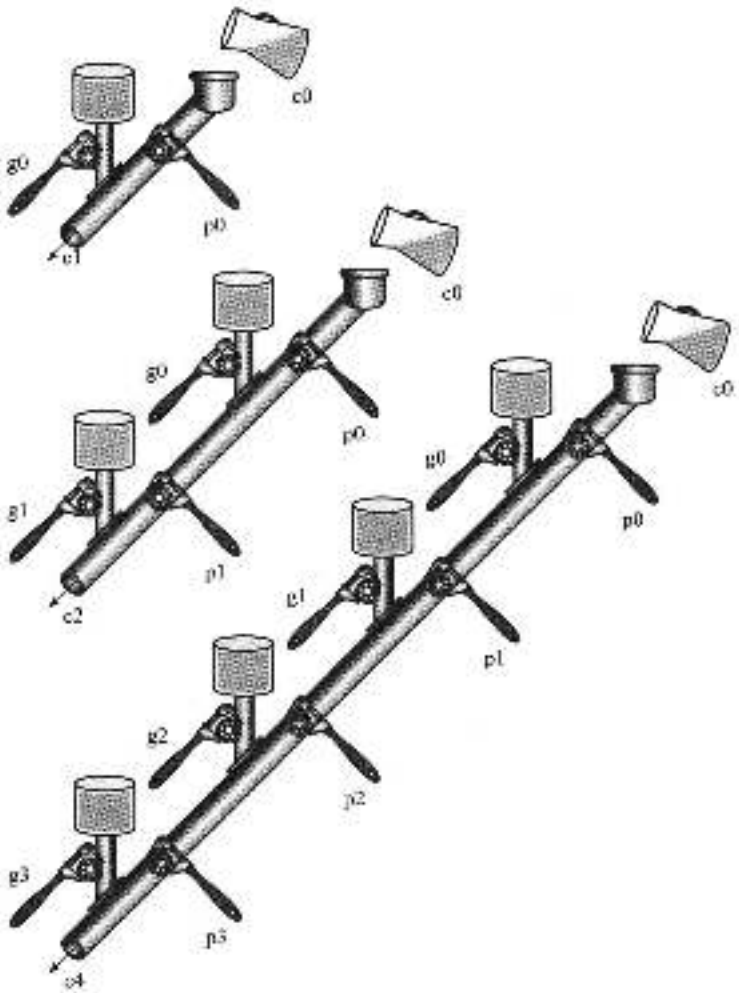
$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

这些公式只代表一般情况：如果之前的加法器生成了一个进位，并且所有其中间的传递因子传输了这个进位，那么CarryIni = 1。图B-6-1就是用这种方法解释超前进位。

但是这种简化的形式使方程变得很长，因此考虑一个16位的加法器的逻辑，试着转到两个抽象级上实现。




图B-6-1 1位、2位、4位超前进位的管道表示以及对应的值。扳手有开/关两个值，水用灰色部分表示，管道的输出（ci+1）会满，如果最近的进位产生函数的值（gi）处于打开状态或者第i个进位传输函数（pi）也是打开的，这样上游会有水，或者从之前生成的或者是从后面传过来的水。进位输入（c0）能在没有任何进位产生函数的情况下输出一个进位，但是需要所有的进位传输函数

## B.6.3 用第二级抽象快速进位

首先，我们考虑4位的加法器，它的超前进位逻辑作为一个单独块。如果将4位加法器的行波进位相连接，形成一个16位的加法器，这将比原始有更多硬件的加法器速度快。 B-40

为了执行得更快，需要将超前进位放置在更高层上。为了执行4位加法器的超前进位，需要将传递因子和生成因子也置于较高的层次。下面是4位加法器的块：

$$P0 = p3 \cdot p2 \cdot p1 \cdot p0$$
$$P1 = p7 \cdot p6 \cdot p5 \cdot p4$$
$$P2 = p11 \cdot p10 \cdot p9 \cdot p8$$
$$P3 = p15 \cdot p14 \cdot p13 \cdot p12$$

即“超”传递信号的4位抽象（Pi）为真，当且仅当组中的每一位都将传递一个进位。

对于“超”生成信号（Gi），我们只关心4位的组中最显著的位中是否有一个进位。如果对于大部分最显著位的生成因子为真，这些情况是显而易见的。如果较早的一个生成因子为真，而且包括大多数的最显著位的中间所有的传递因子也为真，以上情况也是会出现的。

$$G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)$$
$$G1 = g7 + (p7 \cdot p6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)$$
$$G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)$$
$$G3 = g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)$$

图 B-6-2 用管道作为类比，以证明 P0 和 G0。

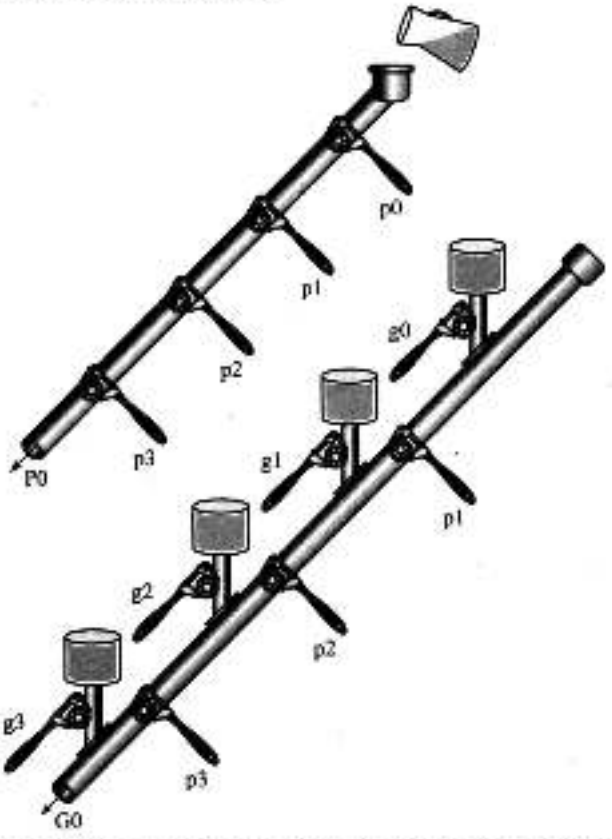



图 B-6-2 下一级超前进位信号 P0 和 G0 的管道分析。仅当所有 4 个进位传输函数（pi）都打开时 P0 是打开的，G0 里是有水流的，仅当至少有一个进位产生函数（gi）是打开的，并且从该生成因子所有的进位传输函数的下游是开放的

图 B-6-3 中 16 位加法器的 C1、C2、C3、C4 与 B.6.2 节的 4 位加法器的 c1、c2、c3、c4 的很相似：

$$C1 = G0 + (P0 \cdot c0)$$
$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$
$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$
$$C4 = C3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

图 B-6-3 所示的是 4 位的加法器连接到一个超前进位单元。练习题中会探究这些进位方案
B-41 速度的差异，对位传递因子和生成因子信号的不同表示，以及 64 位加法器的设计。

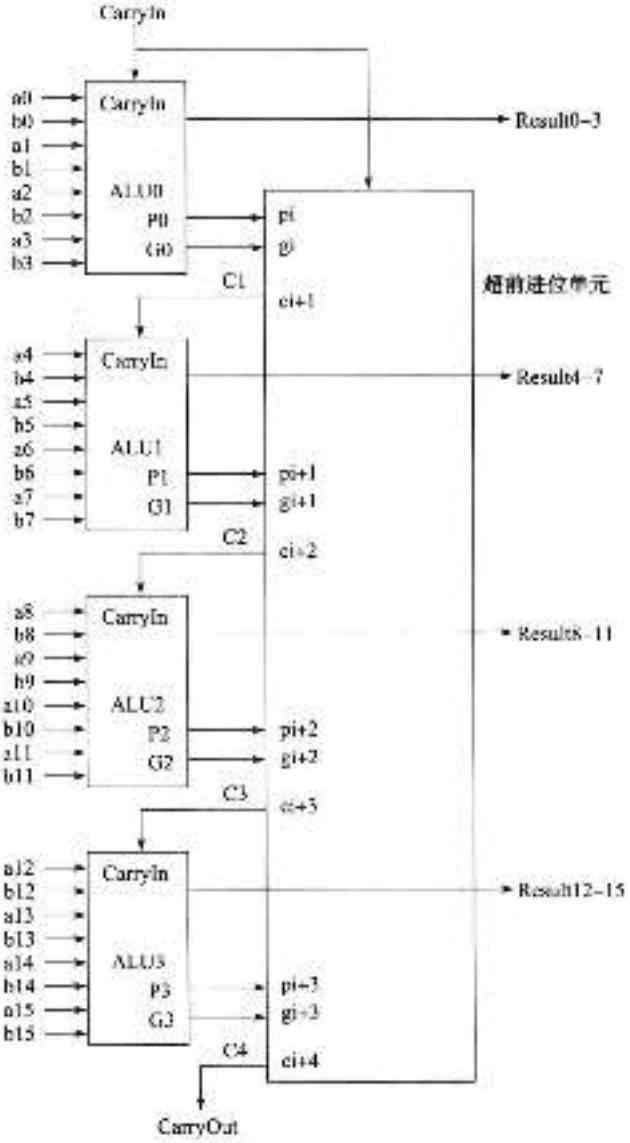



图 B-6-3 用 4 个 4 位超前进位单元串接形成 16 位加法器。注意，进位均来自超前进位单元，而不是 4 位的 ALU

**例题 • 不同级的进位传输函数和进位产生函数**

确定两个 16 位数的 gi、pi、Pi 以及 Gi 值：

```
a:      0001  1010  0011  0011,
b:      1110  0101  1110  1011;
```

同样的，CarryOut15（C4）的值是多少？

**01 答案**

将各位对准，很容易得到进位产生函数 gi(ai · bi) 和进位传输函数 pi(ai + bi) 的值：

```
a:      0001  1010  0011  0011
b:      1110  0101  1110  1011
gi:     0000  0000  0010  0011
pi:     1111  1111  1111  1011
```

从左到右依次标记为 15 ~ 0，“超”进位传输函数（P3、P2、P1、P0）是低级进位传输函数简单相与。

$$P3 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P2 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0$$

“超”进位产生函数较复杂一些，用下式表示：

$$\begin{aligned} G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0 \\ G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1 \\ G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \\ G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \end{aligned}$$

最后，CarryOut15 为：

$$\begin{aligned} C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\ &= 0 + 0 + 1 + 0 + 0 = 1 \end{aligned}$$

因此，当加上这些 16 位的数之后会有一个进位输出。 □

B-42 ~ B-44

超前进位能快速进位的原因是，当时钟周期开始时所有的逻辑单元同时开始计算，并且每个门停止变化时输出不会改变。通过利用更少的门发送进位信号这种快捷方式，门的输出将很快停止变化，因此加法器延迟时间就少了。

为了更好地理解超前进位的重要性，我们需要计算它与行波进位加法器之间的相对性能。

**01 例题·行波进位加法器和超前进位加法器速度的比较**

一个建立时间逻辑模型的简单方法是假设通过每个与门或者或门需要的时间是相同的。通过简单计算通过逻辑路径上门的数量估计时间，比较 16 位加法器路径上门延迟的数量，一个用行波进位，另一个用的是两级的超前进位。

**01 答案**

B.5 节中的图 B-5-5 所示的每个进位输出信号需要两个门延迟，最低位上的进位输入和最高位上的进位输出之间的门延迟为 16 × 2 = 32。

对超前进位加法器来说，最高位的进位输出正是例子中定义的 C4。用 Pi 和 Gi 两个层次的逻辑上定义 C4（几个 AND 组成的 OR 式）。Pi 是用 pi（与门）定义的，Gi 是用 pi 和 gi 共同来定义

的。所以，下一抽象级最差的情况是两级逻辑。pi 和 gi 都是用 ai 和 bi 定义的一级逻辑。在这个方程中，如果假设每个逻辑级都是一个门延迟，那么最坏的情况是 2+2+1=5 个门延迟。

因此，对于一个进位输入到进位输出的通路，16 位超前进位加法器的速度是行波进位加法器的 6 倍，用这种方法可以简单估计硬件的速度。 □

B-45 ↑ B-46

## 小结

超前进位加法器比 32 个 1 位组成的 32 位行波进位加法器的速度快，这个快速通路的两个主要信号是进位产生函数和进位传播函数。

前者忽略了进位输入，后者是沿着进位传播的。超前进位加法器的抽象概念在计算机设计解决复杂化问题时的重要性方向也是很好的示例。

**小测验**

用门延迟对硬件执行速度进行简单评估，一个 8 位的行波进位加法器和一个 64 位的超前进位加法器的相对性能如何？

1. 一个 64 位超前进位加法器能快 3 倍：8 位加法器有 16 个门延迟，64 位的有 7 个门延迟。
2. 它们的速度大约相等，因为 64 位加法器需要 16 位加法器有更多的逻辑层。
3. 8 位加法器比 64 位的快，即使有超前进位。

**解答** 除了一个算术逻辑操作之外，我们已经描述了核心 MIPS 指令集的全部操作；图 B-5-14 忽略了移位指令，这可能会扩大 ALU 多路选择器，包括左移一位和右移一位。但硬件设计人员设计了一种电路，称为桶形移位器（barrel shifter），它可以从 1 移到 31 位，消耗的时间和将两个 32 位的数字相加的时间相差不大，所以移位操作通常是在 ALU 外部完成的。

**解答** B.5 节中，全加器和前边的逻辑方程可以用一个比与门和或门能力更强的门来简单表示。如果两个操作数不同，并或门输出为真，即

$$x \neq y = 1 \quad 且 \quad x = = y = 0$$

在一些技术中，异或门比与门和或门的速度效率更高，用⊕来表示异或运算，则等式可以重新表达为：

$$Sum = a \oplus b \oplus CarryIn$$

同样，我们用这种传统的门级表示方法来表示 ALU 电路。当今的计算机都是用 CMOS 晶体管设计的，CMOS ALU 以及桶形移位器利用了开关的优点，而且比文中的设计使用的多路选择器少，但是设计原则是相似的。

**解答** 当有两个以上的级别的层次结构时，用小写和大写字母来区分进位生成和进位传播符号。$g_{i,j}$ 和 $p_{i,j}$ 代表从 i 位到 j 位的进位产生函数和进位传播函数，因此，$g_{1,1}$ 表示产生 1，$g_{4,1}$ 表示产生 4 到 1，同样，$g_{16,1}$ 表示从 16 到 1。

B-47

## B.7 时钟

在我们讨论存储元件和时序电路之前有必要简要地讨论一下时钟。这一节主要讨论这一主题，并且本节内容同 4.2 节的讨论类似。更多的关于时钟和时序策略的细节将在 B.11 节讨论。

时钟在时序电路中非常重要，它决定了处于某一状态的存储元件何时被更新。时钟是一个具有固定周期时间的不停运转的信号，时钟频率是周期时间的倒数。如图 B-7-1 所示，时钟周期时间或者说时钟周期被分割为两部分，高电平时钟和低电平时钟。在此我们只使用**边沿触发时钟**（edge-triggered clocking）。这意味着所有的状态改变都将发生在时钟边沿。我们之所以使用一种基于边沿触发的时钟策略，是因为它易于解释。从工艺学的角度来看，很难说依赖于边沿触发的时钟策略的**时钟同步方法**（clocking methodology）是否是最好的选择。

- 边沿触发时钟：一种时钟机制，在这种机制下所有的状态改变都发生在时钟边沿。
- 时钟同步方法：一种根据时钟来决定数据何时有效和稳定的方法。

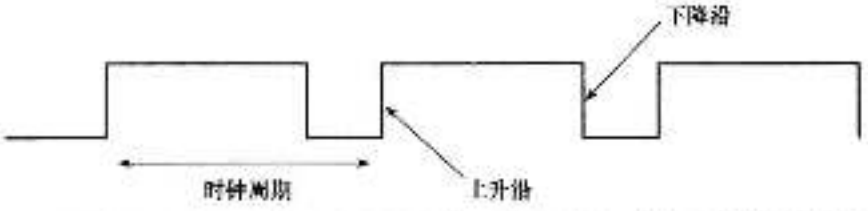



图 B-7-1 时钟信号在高电平和低电平之间震荡。时钟周期是一个完整的最小间隔的时间。在边沿触发的设计中，可使用有效的上升沿或者下降沿来改变状态

在一种边沿触发的方法中，可使用有效的上升沿或者下降沿来改变状态。在下一节中，**状态单元**（state element）在一个边沿触发的设计中实现，且状态单元的内容只有在一个有效的时钟沿才能改变。选择哪一个时钟边沿作为有效边沿受设计策略的影响，且不会影响逻辑设计中所涉及的概念。

时钟边沿作为一个采样信号，会导致状态单元的输入值被采样且存储在状态单元中。使用一个边沿触发器意味着采样过程实际上是瞬时的，可以消除信号在很小的时间差内被采样可能导致的问题。

时钟系统即**同步系统**（synchronous system），最主要的限制是被写入状态单元的信号在有效时钟边沿必须是有效的。一个信号在它是稳定（不改变）时才是有效信号，并且在输入不变时，信号值不会改变。因为组合电路没有反馈结果，在组合逻辑单元的输入没有改变的情况下，组合逻辑单元的输出将最终有效。 B-48

- 状态单元：存储元件。
- 同步系统：一个使用时钟的存储系统，且数据信号只有在当前时钟下是处于稳定状态时才可被读取。

图 B-7-2 表示处于一个同步时序逻辑设计中的状态单元和组合逻辑单元的关系。状态单元的输出只在时钟边沿之后改变，且状态单元组合逻辑块提供有效的输入。为了保证在有效时钟边沿写入状态单元的数据是有效的，时钟周期必须足够长才能保证所有在组合逻辑块中的信号都稳定后，在时钟边沿采样这些数据，将其存储在状态单元中。这个限制将时钟周期的长度限制在一个较小的范围，即时钟周期必须足够长以满足所有的状态单元的输入都是有效的。

在附录的其他部分和第 4 章，我们通常忽略时钟信号，因为我们假设所有的状态单元都会在同一时钟边沿更新。一些状态单元会在所有的时钟边沿被写入，而其他一些仅仅在确定的条件下被写入（如某个寄存器被更新）。在这种情况下，我们会使用一个显式的写信号来控制这个状态单元。写信号同时钟信号一起控制状态单元的更新，只有在时钟边沿且写信号有效时状态单元才会被更新。我们将在下一节学习和使用这一机制。




图 B-7-2 组合逻辑块的输入来自状态单元，同时其输出也将被写入状态单元。时钟边沿决定了状态单元的内容何时被更新

边沿触发机制的另一优势是可以将一个状态单元同时作为同一组合逻辑块的输入和输出，如图 B-7-3 所示。实际上，在这种情况下必须要防止竞争，同时要保证时钟周期足够长。这一问题将在 B.11 节讨论。

B-49 现在我们已经讨论了时钟是如何用来更新状态单元的，我们将讨论如何构建这种状态单元。

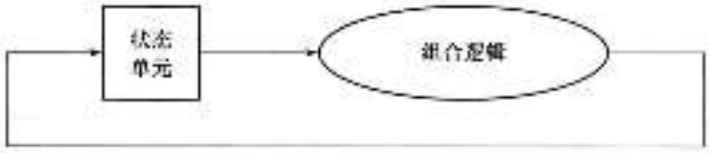



图 B-7-3 边沿触发策略允许一个状态单元在一个时钟周期内被读写，前提是在这个时钟周期内没有会导致不确定值的竞争。因此时钟周期必须足够长，在有效时钟沿到来前所有状态单元的输入都是有效的

1 **精解** 设计者经常发现，在大多数状态单元中，使得少量的状态单元在相反的时钟沿进行状态变化非常有用。但是在使用这种方法时需要十分小心，因为会影响到状态单元的输入和输出。为什么设计者还要这么做呢？考虑这样的情况，一部分作为状态单元输入或者输出的组合逻辑十分小以至于它们可以在半个周期内完成，而不是通常的完整时钟周期。因此状态单元可以在半个周期的时钟边沿被写入，因为它的输入和输出在半个时钟后更新是有效的。这种技术经常被用在寄存器文件（register file）中，在寄存器文件中，简单的寄存器文件读写通常发生在半个周期。第 4 章使用这种策略来减少流水线开销。

⊃ 寄存器文件：包含一系列寄存器的状态单元，通过提供一个访问的寄存器号可以读写这些寄存器。

## B.8 存储元件：触发器、锁存器和寄存器

在本节及下一节中，我们将讨论存储元件的基本原理，从触发器、锁存器开始，再介绍寄存器文件，最后介绍存储器。所有的存储元件都存储着一些状态；存储元件的输出不仅取决于当前的输入，而且与当前存储的数据有关。因此所有包含存储元件的逻辑块都包含有状态信
B-50 息，属于时序逻辑。

最简单的存储元件类型是无时钟的，即这些元件都没有任何的时钟输入。因为无时钟的锁存器是最简单的存储元件，所以我们将先讨论这种元件，尽管在本节中我们只使用带时钟的存储元件。图 B-8-1 为一个 S-R 锁存器（set-reset 锁存器），该锁存器由一对或非门构成（或非

门即一个或门加一个反向器）。输出信号 $Q$ 和 $\overline{Q}$ 表示存储的数据及其反向数据。当 $S$ 和 $R$ 都没有被激无效时，交叉耦合的或非门就是一个反向器，存储先前的 $Q$ 和 $\overline{Q}$ 的值。

例如，如果输出 $Q$ 为真，下面的反向器将产生一个值为假的输出（也就是 $\overline{Q}$），这个输出又成为上面反向器的输入，上面的反向器产生一个值为真的输出，即 $Q$，之后一直循环下去。如果 $S$ 为有效信号，输出 $Q$ 的值拉低，$\overline{Q}$ 的值被拉高；如果信号 $R$ 有效，则输出 $\overline{Q}$ 为假，输出信号 $Q$ 被拉高。如果 $S$ 和 $R$ 都有效，则 $Q$ 和 $\overline{Q}$ 最后的数值将被存储在交叉耦合结构内。同时将 $S$ 和 $R$ 置为有效信号，可能会导致错误的操作：这取决于 $S$ 和 $R$ 是如何被拉高的，对锁存器来说，结果可能会不停地摆动，也可能处于亚稳态（这部分将在 B.11 节中详细介绍）。

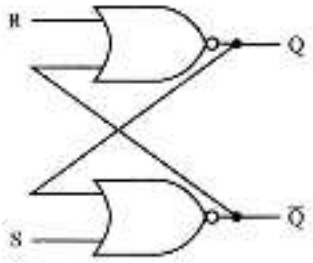



图 B-8-1 一对交叉耦合的 NOR（或非门）可以存储一位数据。输出信号 $Q$ 取反后得到 $\overline{Q}$，再次 $\overline{Q}$ 取反后得到 $Q$，因此 $Q$ 是可再生的。如果 $R$ 或 $\overline{Q}$ 中的一个为有效信号时，$Q$ 就会被拉低，反之亦然

这种交叉耦合结构是我们构造复杂存储元件的基本结构，构造出来的复杂存储元件可以存储数据。这些存储元件包含额外的门用来存储信号，并且在包含时钟时对存储的数据状态进行更新。下一节将讲述如何构建这些存储元件。

### B.8.1 触发器和锁存器

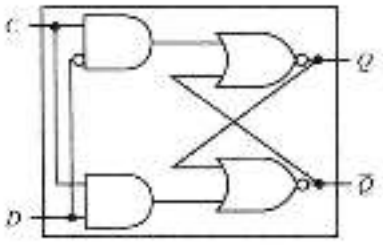
**触发器**（flip-flop）和**锁存器**（latch）是最简单的存储元件。在触发器和锁存器中，输出信号的值都与存储元件中存储的状态一致。而且，与上面提到的 S-R 锁存器不同，从现在开始，我们使用的所有触发器和锁存器都是带时钟的，这意味着这些存储元件将包含时钟输入信号，并且时钟将触发存储状态的变化。触发器与锁存器间的差别在于，引起存储状态发生变化的时钟位置不同。在一个包含时钟的锁存器中，只要时钟信号有效，输入信号发生变化就会引起存储状态的变化。然而在触发器中，只有在时钟信号的边沿，存储元件的状态才会变化。因为本文中，我们使用边沿触发的时钟方法，即存储状态只在时钟边沿发生变化，因此我们只使用触发器。触发器大都由锁存器构成，因此我们先介绍简单的带时钟的锁存器，介绍完它的一些操作后，再介绍由这些锁存器构成的触发器的一些操作。

- **触发器**：一种存储元件，它的输出与内部存储的状态一致，并且内部状态只在时钟的边沿发生变化。
- **锁存器**：一种存储元件，它的输出与内部存储的状态一致，并且当时钟有效时，只要输入发生变化，存储状态就会随之发生变化。

对计算机应用程序来说，触发器和锁存器的功能都是存储信号。一个 D 锁存器或 D 触发器（D flip-flop）将输入的数据信号存储在内部元件中。尽管有很多类型的锁存器和触发器，但我们只需要 D 型的。一个 D 锁存器包含两个输入和两个输出。两个输入中一个是要存储的数据（$D$），一个是时钟信号（$C$），时钟信号用来指示锁存器什么时候读取输入 $D$ 的值并进行存储。输出信号就是内部状态 $Q$ 和其反向信号 $\overline{Q}$。当输入时钟 $C$ 有效时，锁存器称为打开状态，此时输出信号 $Q$ 的值为输入信号 $D$ 的值。当输入时钟无效时，锁存器处于关闭状态，此时锁存器的输出 $Q$，等于锁存器最后一次打开时所存储的数据。

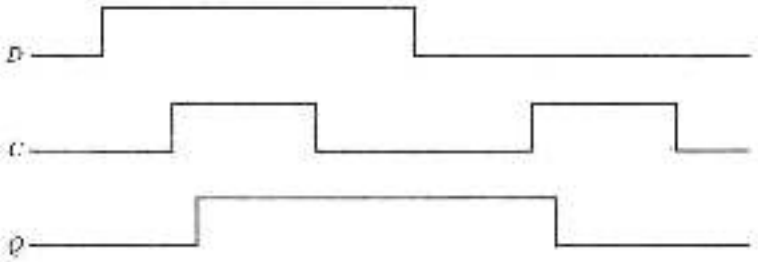
⊃ D触发器：一个包含一个输入数据的触发器，这类触发器只在时钟信号的边沿，才将输入信号存储到内部元件中。

图B-8-2显示了如何用交叉耦合的或非门，两个额外的门来构造D锁存器。由于当锁存器处于打开状态时，输出$Q$的值随输入$D$的改变而变化，因此这种结构有时也称为透明锁存器。图B-8-3显示了D锁存器是如何工作的，图中假设输出$Q$初始化为假，并且$D$最先变化。



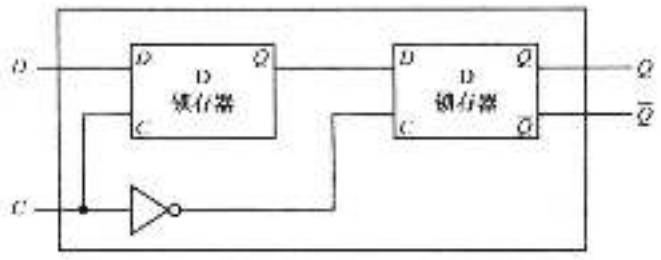

图B-8-2 一个用或非门实现的D锁存器。如果其他的输入为0，则或非门只作为反向器使用。因此，交叉耦合的或非门只有在输入时钟$C$有效时才进行存储，此时，输入$D$将替代$Q$，并被存储起来。在时钟信号$C$由有效变为无效时，必须保证输入信号$D$的稳定

正如前面提到的那样，我们使用触发器作为基本构造单元，而不是使用锁存器。触发器不是透明的；它们的输出只在时钟边沿发生变化。一个触发器可以设计成在时钟上升沿或下降沿进行触发，在本书的设计中我们可以使用任意一种类型。图B-8-4显示了如何用一对D锁存器来构造下降沿触发的D触发器。在D触发器中，输出在时钟边沿存储。图B-8-5显示了这个D触发器是如何操作的。




图B-8-3 D锁存器的操作，假设输出信号被初始化为无效信号。当时钟$C$有效时，锁存器被打开，输出信号$Q$的值被赋为输入$D$的值

[B-52]




图B-8-4 下降沿触发的D触发器。第一个锁存器称为主锁存器。当输入时钟$C$有效时，主锁存器打开，输入数据$D$。当输入时钟$C$被拉低时，主锁存器关闭，但第二个锁存器打开，并且主锁存器的输出作为第二个锁存器的输入信号。第二个锁存器称为从锁存器

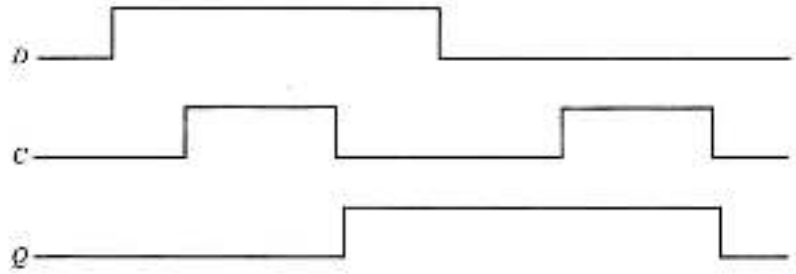



图 B-8-5 下降沿触发的 D 触发器的操作，假设其输出被初始化为无效。当时钟 $C$ 从有效变为无效时，输出 $Q$ 将输入信号 $D$ 的值进行存储。与图 B-8-3 中的 D 锁存器相比，在带时钟的锁存器中，只要时钟 $C$ 为高电平，存储的数据和输出 $Q$ 就发生变化，相反触发器只在时钟发生转变时才发生变化

下面是上升沿触发的 D 触发器的 Verilog 代码，假设信号 $C$ 为输入时钟，$D$ 为输入数据：

```
module DFF(clock,D,Q,Qbar);
   input clock, D;
   output reg Q; // Q is a reg since it is assigned in an
                 // always block
   output Qbar;
   assign Qbar = ~ Q; // Qbar is always just the inverse
                      //  of Q
   always @(posedge clock) // perform actions whereve~ the
                           // clock rises
      Q = D;
endmodule
```

由于输入 $D$ 在时钟边沿被取样，因此在时钟边沿之前和之后的这段时间内，$D$ 必须保持有效。在时钟发生跳变前，输入信号需要保持有效的最短时间，称为**建立时间**（setup time）；在 B-53
时钟跳变后，输入信号需要保持有效的最短时间，称为**保持时间**（hold time）。因此任何触发器（或任何由触发器构造的设备）的输入必须在一个时间窗口内保持有效，这个时间窗口开始于时钟跳变前 $t_{setup}$ 时间，结束于时钟跳变后的 $t_{hold}$ 时间，如图 B-8-6 所示。B.11 节中，详细介绍了时钟和时序约束，包括触发器的传播延时。

- **建立时间**：在时钟发生跳变前，输入信号必须保持有效的最短时间。
- **保持时间**：在时钟跳变后，输入信号需要保持有效的最短时间。

我们可以使用 D 触发器的阵列来构建一个寄存器，构建的寄存器可以存储多位数据，比如一个字节或一个字。在第 4 章中，我们将在数据通路中使用寄存器。

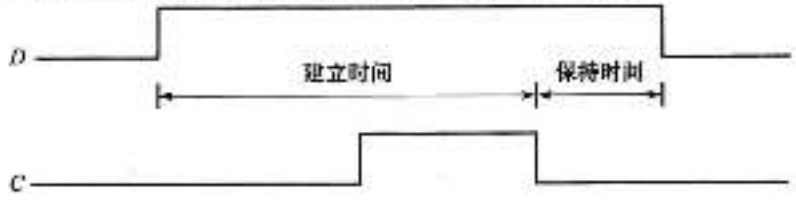
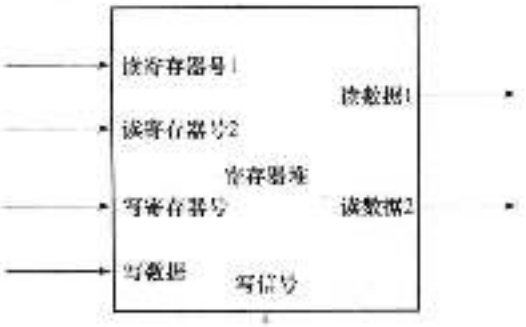


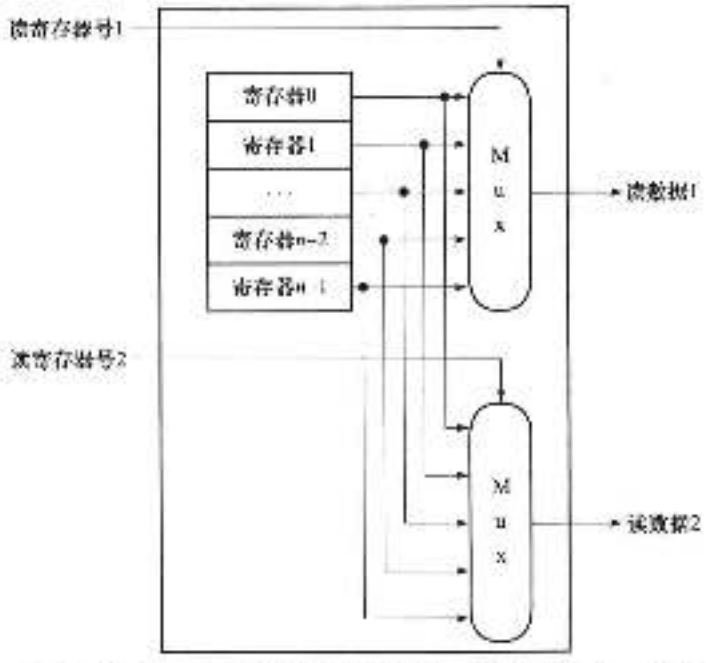

图 B-8-6 下降沿触发的 D 触发器的建立时间和保持时间。输入信号在时钟跳变前和跳变后都需要保持一段时间的有效值。时钟跳变前，输入信号需要保持有效的最短时间称为建立时间；时钟跳变后，输入信号需要保持有效的最短时间称为保持时间。如果违反了最小建立时间和最小保持时间，触发器的输出可能变得不准确，正如 B.11 节中要讲的。保持时间要么为 0，要么是一个很小的值，因此不需要担心建立时间

## B.8.2 寄存器文件

寄存器文件是数据通路中一个重要的核心结构。寄存器文件包含一组可读写的寄存器，寄存器的读写是通过指定寄存器号进行的。通过一个由D触发器构成的寄存器阵列，并对每一个输入或输出端口添加译码器，就可以构造出一个寄存器文件。如果对寄存器文件只进行读操作，不会改变其状态，因此我们只需提供一个寄存器号作为输入，输出的结果即为该寄存器号对应寄存器中的数据。对于写操作，我们需要三个输入：寄存器号、要写入的数据和一个控制写入的时钟。第4章中，我们使用了一个包含两个读端口和一个写端口的寄存器文件。该寄存器文件如图B-8-7所示。其中读端口可以通过一对多路选择器来实现，每一个多路选择器的位宽与寄存器文件中单个存储器的位宽相等。图B-8-8为32位宽寄存器文件读端口的实现方法。




图B-8-7 一个包含两个读端口和一个写端口的寄存器文件，该寄存器文件包含5个输入和2个输出。输入控制写信号用灰色表示




图B-8-8 对于包含两个读端口、n个寄存器的寄存器文件，可以通过一对n选1多路选择器来实现读端口，每个多路选择器为32位宽。读操作寄存器号用来作为多路选择器的选择信号。图B-8-9显示了如何实现写端口

实现寄存器写端口有点复杂，因为我们只能更改需要更改的寄存器内容。为了达到这个目的，可以使用一个译码器来生成一个信号，用该信号来决定是对哪个寄存器进行写操作。

图 B-8-9 显示了如何实现一个寄存器文件的写端口。需要注意的是，触发器的状态只在时钟边沿发生变化。在第 4 章中，我们明确地对寄存器文件中的写信号打了勾，并且假设图 B-8-9 中的时钟默认被加入。

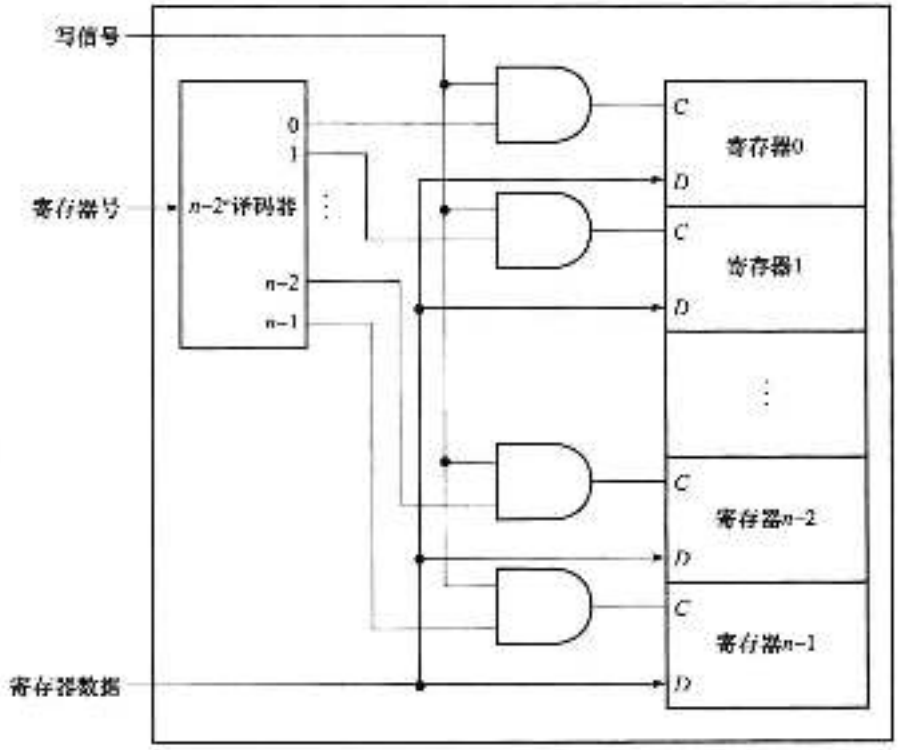



图 B-8-9 寄存器文件的写端口通过一个译码器来实现，译码器与写控制信号一起生成信号 C 输入到寄存器中。所有的三个输入（寄存器号、数据、写控制信号）都存在建立时间和保持时间的约束，以保证正确的数据被写到寄存器文件中

如果在一个时钟周期中，对寄存器同时进行读和写，将会发生什么？因为写寄存器文件出现在时钟边沿，因此在读操作时，寄存器是有效的，正如图 B-7-2 中看到的那样。读出的数据将是上一个时钟周期写入的数据。如果我们想要读出当前正在写入的数据，则需要在寄存器内部或外部添加额外的逻辑。第 4 章广泛使用了这类寄存器。 B-54

## B.8.3 用 Verilog 描述时序逻辑

用 Verilog 来描述时序逻辑，我们必须知道如何生成一个时钟，如何描述何时将数据写入寄存器中，及如何指定时序控制。我们先来描述一个时钟。时钟不是 Verilog 中预定义的变量，我们可以在一个声明前使用符号#n 来生成一个时钟，这将导致该声明在 n 个时钟延迟之后被执行。在大多数 Verilog 模拟器中，产生一个时钟来作为外部输入也是可以的，这将允许用户在模拟过程中，指定需要模拟器运行的时钟周期数。

图 B-8-10 中的代码实现了一个简单的时钟，该时钟高电平和低电平都保持一个模拟时间单元，之后就转化为反向状态。为了实现时钟，我们也使用了延迟和阻塞赋值语句。

```
reg clock; // clock is a register
always
#1 clock = 1; #1 clock = 0;
```

图 B-8-10 一个时钟的 Verilog 描述 B-55 | B-56

接下来，我们必须实现边沿触发寄存器的操作。在 Verilog 中，这是通过使用 always 块的敏感信号列表实现的，并且相应的使用 posedge 或 negedge 来指定上升沿触发还是下降沿

触发。因此，下面的 Verilog 代码中，在时钟上升沿中，寄存器 A 写入数据 b。

通过本章内容及第 4 章的 Verilog 部分，我们将勾画一个上升沿触发的设计。图 B-8-11 显示了一个 MIPS 寄存器文件的 Verilog 代码，代码中包含了两次读操作和一次写操作，其中只有写操作是包含时钟的。

**小测验**

图 B-8-11 中寄存器文件的 Verilog 代码中，正在进行读操作的寄存器对应的输出端口使用的是连续赋值语句。但是寄存器的写入使用的是 always 块。下面哪一项是其原因？

a. 没有特殊原因，只是为了方便。

b. 因为 Data1 和 Data2 是输出端口，WriteData 是输入端口。

c. 因为读操作是一个组合事件，而写操作则是一个时序事件。

```
    reg [31:0] A;
    wire [31:0] b;

    always @(posedge clock) A <= b;

module registerfile (Read1,Read2,WriteReg,WriteData,RegWrite,
Data1,Data2,clock);
   input [5:0] Read1,Read2,WriteReg; // the register numbers
                                     // to read or write
   input [31:0] WriteData; // data to write
   input RegWrite, // the write control
     clock; // the clock to trigger write
   output [31:0] Data1, Data2; // the register values read
   reg [31:0] RF [31:0]; // 32 registers each 32 bits long

   assign Data1 = RF[Read1];
   assign Data2 = RF[Read2];

   always begin
      // write the register with new value if RegWrite is
      // high
      @(posedge clock) if (RegWrite) RF[WriteReg] <=
WriteData;
   end
endmodule
```

图 B-8-11 用行为级描述的 MIPS 寄存器文件。该寄存器文件在时钟上升沿进行写操作

## B.9 存储元件：SRAM 和 DRAM

寄存器和寄存器文件可以作为基本构建单元来构造小容量存储器，但是，大容量存储器要么是用 SRAM（static random access memory，静态随机访问存储器），要么是用 DRAM（动态随机访问存储器）来构建。我们先来介绍比较简单的 SRAM，再来介绍 DRAM。

### B.9.1 SRAM

SRAM 是包含存储阵列的集成电路，存储阵列通常包含一个访问端口，该端口可以用来表示读或写。SRAM 对任一单元的访问时间都是固定的，尽管读操作和写操作的特征不同。依据可寻址单元和每个可寻址单元的位宽，SRAM 芯片形成一个特殊的格局。例如，一个 4M×8 的 SRAM，可以提供 4M 的入口，每一个入口包含 8 位二进制数据。因此它共有 22 条地址线

$(4M = 2^{22})$、8 位宽的输出和 8 位宽的输入。与 ROM 类似，可寻址单元的个数称为高度，每个可寻址单元的位宽称为宽度。因为多种技术原因，最新最快的 SRAM 常常使用较窄的配置：×1 和×4。图 B-9-1 显示了一个 2M × 16 的 SRAM 的输入和输出信号。

B-57
B-58

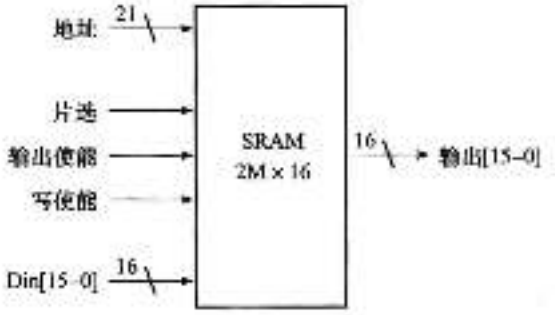



图 B-9-1 一个 32K × 8 的 SRAM，其中包括 21 位地址线（$32K = 2^{15}$）和 16 位输入线，3 条控制线和 16 位输出线

⇒ SRAM：一种存储器，其中的数据是静态存储的（如触发器），而不是动态存储的（如 DRAM）。SRAM 比 DRAM 快，但是密度较小，每位的价格更高。

为了启动读写操作，片选信号必须处于有效状态。为读取数据，必须激活用来控制被地址选中的数据能否驱动到管脚上的输出使能信号。这样，输出使能信号允许多个存储器连接到单输出总线上，并且用于决定由哪个存储器来驱动总线。SRAM 读取数据所需的时间通常被定义为从输出使能信号有效一直到数据输出总线上为止。2004 年，拥有最快的 CMOS 部件的 SRAM 的读取时间为 2 ~ 4ns，但这样的 SRAM 容量较小，数据宽度较窄，更大部件的 SRAM 的读取时间为 8 ~ 20ns。2004 年，已经实现的最大的 SRAM 能容纳 32M 位数据。过去 5 年间，消费类产品和数码设备对低功耗 SRAM 的需求增长很快。这些 SRAM 通常具有更低的待机和访问功耗，但通常要比普通的 SRAM 慢 5 ~ 10 倍。最近，类似同步 DRAM（下一节讨论）的同步 SRAM 也开发出来了。

进行写操作时，输入端必须提供要写入的数据、目的地址以及写控制信号。当写使能信号和片选信号为真时，数据线上的值就写入由地址线指定的地址单元。正如我们在介绍 D 触发器和 D 锁存器时所提到的，该操作期间要保证地址线和数据线上的信号量保持的时间足够长。同时，写使能信号不是时钟触发边沿，而是有最小宽度约束的脉冲。写操作完成时间则受到建立时间、保持时间以及写使能脉冲宽度的制约。

大容量的 SRAM 不能通过寄存器文件的方式组建。其根本原因在于寄存器文件中的 32-1 多路选择器是切实可行的，但想把 64K-1 多路选择器用于 64K × 1 SRAM 是不现实的。大容量存储器不采用多路选择器，而是通过一条公共信号线（称为位线）来完成，这样存储阵列中有多个存储单元就能被选中。为了满足多个存储单元驱动一条信号线，需要用到三态缓冲装置。三态缓冲器有两个输入：数据信号线和输出使能信号，还有一个输出信号，输出信号有三种状态：有效、无效或高阻。如果输出使能信号有效，其输出值为输入值或输入值取非。否则，其输出为高阻态，这时将由其他的输出使能有效的三态缓冲器共享输出数据。

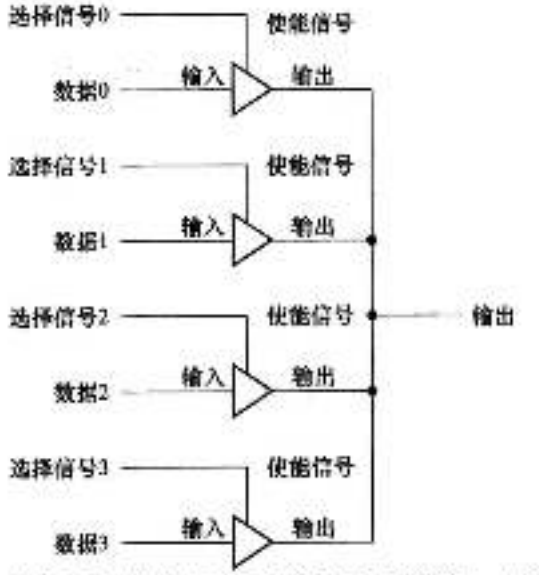



图 B-9-2 用 4 个三态门实现多路选择器。4 个可选择的输入中只有一个有效。三态缓冲器在输出使能信号无效时输出高组态，在输出使能有效时驱动共享的输出线

图 B-9-2 描述了用一组三态缓冲器组成带译码输入的多路选择器。该电路的关键在于需要考虑到任意时刻至多允许一个缓冲器的输出

使能有效；否则，三态缓冲器将会发生输出线竞争现象。在 SRAM 中，每个存储单元使用三态缓冲器，就能实现存储单元对输出信号线共享。采用分布式的三态缓冲器比大规模集中式的多路选择器效率更高。三态缓冲器通常被嵌入到组成 SRAM 的触发器中。图 B-9-3 描述了小容量
B-59 的 4×2 SRAM 的实现，其中用到了带有使能输入的 D 锁存器来控制三态输出。

图 B-9-3 的设计电路中没用到多路选择器，但是它还是用到了大容量的译码器和许多字线。例如，在 4M×8 型 SRAM 中，我们需要用到 22-4M 的译码器，以及 4M 条字线（用于各触发器使能）！为解决这个问题，大容量的存储器被组成矩阵阵列，并且还用到了二级译码装置。图 B-9-4 表明了 4M×8 型 SRAM 是如何利用二级译码来实现的。众所周知，二级译码对于实现 DRAM 的运作至关重要。

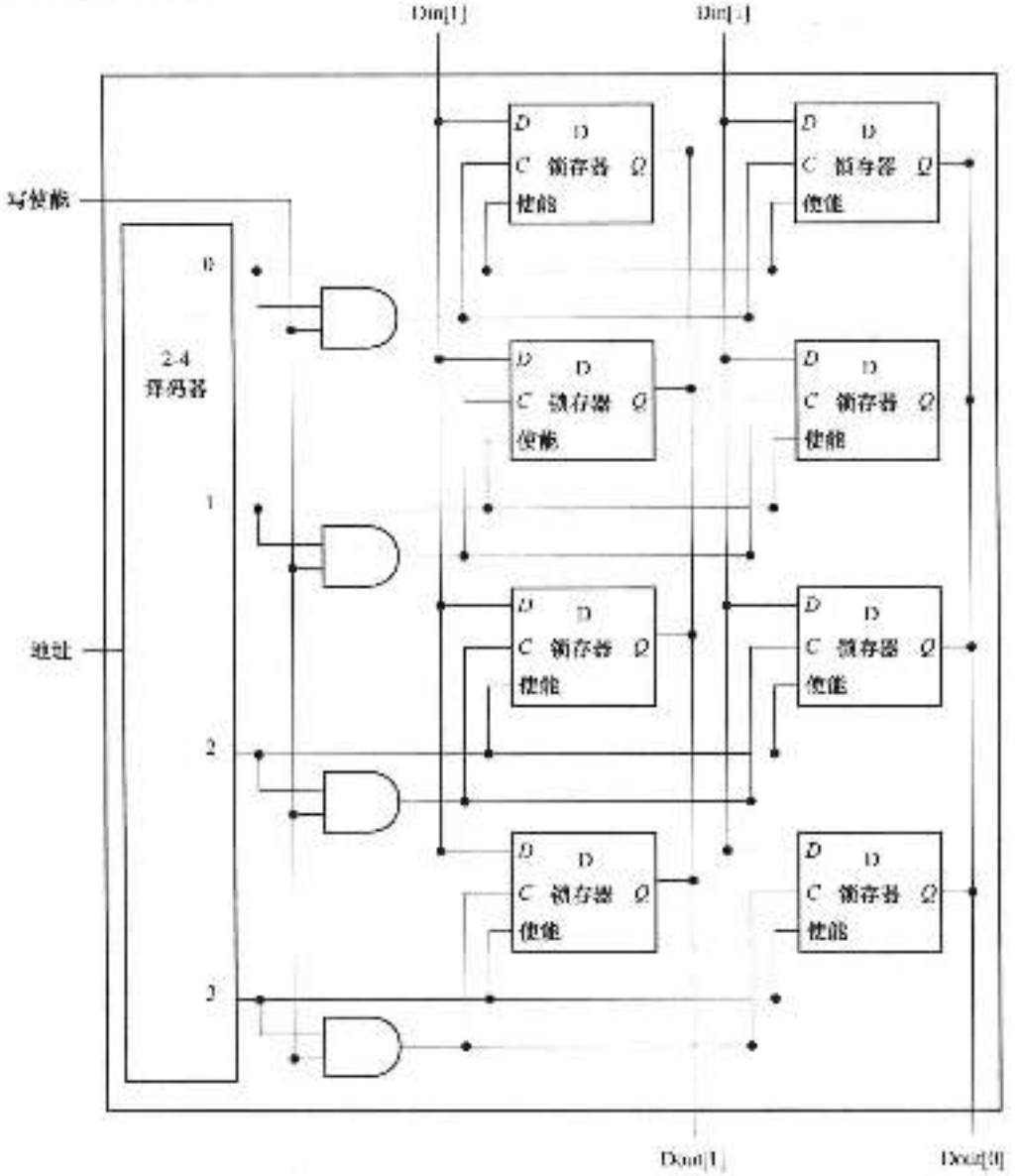



图 B-9-3 4×2 SRAM 的基本结构，其中采用译码器作为片选信号。被激活的单元采用三态输出连接到垂直位数据线，而选择单元的地址信息则通过水平地址线中的某条线（称为字线）传送。为简单起见，此处省略了输出使能信号和片选信号，但它们很容易通过与门接入

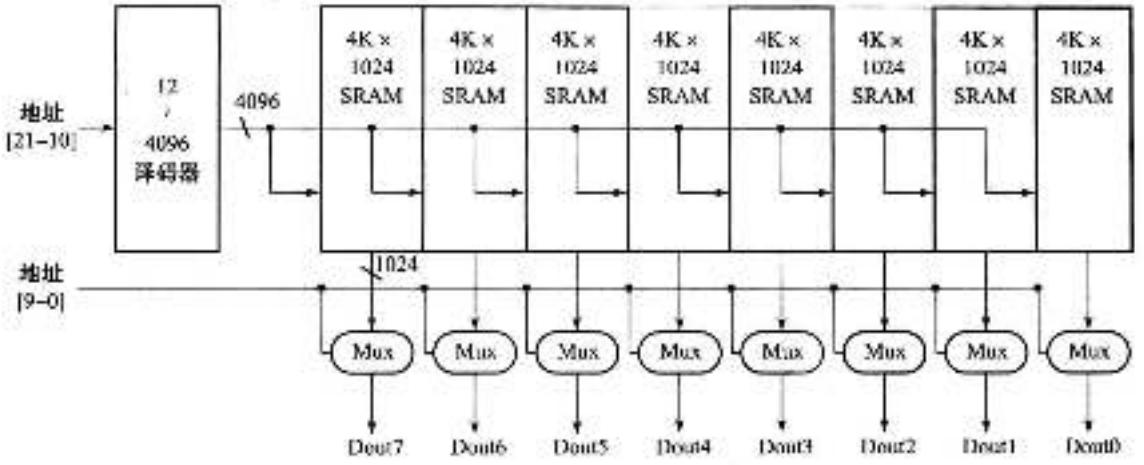



图 B-9-4 用 4K × 1024 阵列实现 4M × 8 SRAM。译码器 1 产生 8 个 4K × 1024 阵列的地址，然后由多路选择器（Mux）从每个 1024 位阵列中选出 1 位，该设计远比单级译码器简单，而单级译码器需要一个庞大的多路选择器。实际上，现在这个大小的 SRAM 可能使用更多数量的模块，并且每一块会更小

近年来，我们看到了同步 SRAM 和同步 DRAM 的发展历程。同步 RAM 的优点在于它能在存储阵列中将顺序地址中的数据以突发方式传输。突发传送的大小同起始地址和突发长度有关。同步 RAM 的卓越之处在于它能将突发数据快速存取，而无需指定额外的地址位。但突发数据中的连续字位受到了时钟的控制。在突发传输模式下，省去指定地址的开销将大大增强传输相邻数据的效率。正因为这个优点，同步 SRAM 和同步 DRAM 在计算机存储系统中大量应用。在下一节和第 5 章中，更详细地讨论了存储系统中同步 DRAM 的使用。 B-60

## B.9.2 DRAM

在静态 RAM（SRAM）中，数据保存在一对反向门电路中，所以只要给它供电，其内部数据一直会保留。而在动态 RAM（DRAM）中，数据是以电荷量的形式被保存在电容中，通过晶体管来存取数据。在动态 RAM 中，对每一位数据的读取只用到了一个晶体管，其密度更高，单位价格更低。相比而言，静态 RAM 中单位比特（每一位）就需要 4 ~ 6 个晶体管。由于动态 RAM 中的数值是以电荷量的形式保存在电容中，所以它需要不断刷新来保持数值。这就是该存储结构称作“动态”的缘由。

为实现对存储单元的刷新，我们需要定期读出该内容并且回写到原单元中去。电荷量通常能维持几毫秒，相当于 1 000 000 个时钟周期。目前，单芯片存储控制器能独立完成刷新功能。如果我们只能将动态 RAM 中的内容逐位读出再逐位回写，那么所有时间只能全用来处理几兆字节大容量存储器的刷新了，根本没有其他时间去完成真正意义上的数据存取。幸好在动态 RAM 中也采用了二级译码装置，这就可以在读周期后紧跟一个写周期实现整行刷新（共享一条字线）。通常，刷新工作只占了动态 RAM 的 1% ~ 2% 的时间，剩下的 98% ~ 99% 的时间可以用来处理外界的数据存取。

**01 精解** 动态 RAM 如何实现读/写存储器单元的信息呢？其关键元件是晶体管，它在存储单元中充当了开关作用，允许存放在电容中的电荷量被读取或写回。图 B-9-5 就是单个晶体管存储单元的外观。电子晶体管的开关作用如下：当字线上的信号有效时，开关处于“关”状态，将电容连到字线上。如果有写请求产生，相应的写入数据就

放到该字线上。如果该数值为1，则电容被充电；否则，数值为0时电容放电。由于动态RAM必须事先检测电容中的电量，所以读操作是很难为复杂。通常，在激活字线准备读出数据前，该信号线先被充电到一半状态。这样通过激活字线，电容上的电荷可被读出到位线。这时位线只需往高电平或低电平方向稍稍偏移就能满足要求，这种变化能通过敏感放大器检测到。

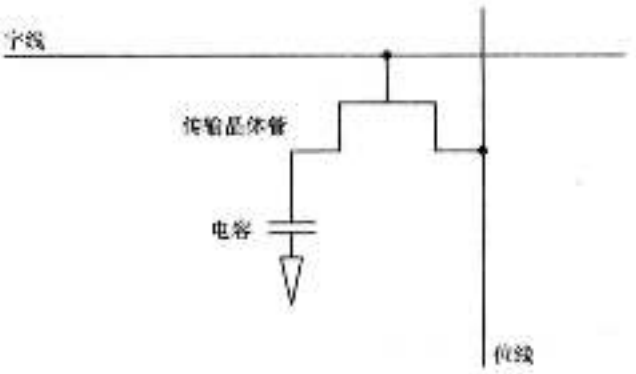



图 B-9-5 用单个晶体管实现的DRAM单元。存储单元内容采用电容实现，读写则通过晶体管实现

B-61
:
B-63

动态RAM有二级译码器，分别实现行存取和列存取，如图B-9-6所示。其中行存取选中一行，然后激活对应的数据字线。于是激活态的行所在的所有列的内容被保存到一组锁存器之中。列存取则是从列锁存器中选取相应的数据。为了节省管脚并进一步减少封装开销，行地址/列地址将共享地址线。其中采用RAS（行存取选通脉冲）和CAS（列存取选通脉冲）来决定是行还是列地址。刷新过程只是简单地将列信息读入列锁存器，然后再回写到列单元中去。于是在一个周期之内就可以完成行刷新。二级寻址方法，再加上中间转换电路，会导致DRAM的存取时间变长。一般为静态RAM的5～10倍。2004年，典型的DRAM存取时间为45～65ns，256Mbit的DRAM已量产，2004年第一季度第一个1GB的DRAM样品也生产出来了。因此，单位比特低成本的DRAM是实现主存的首选，而高速缓存通常由SRAM来完成。

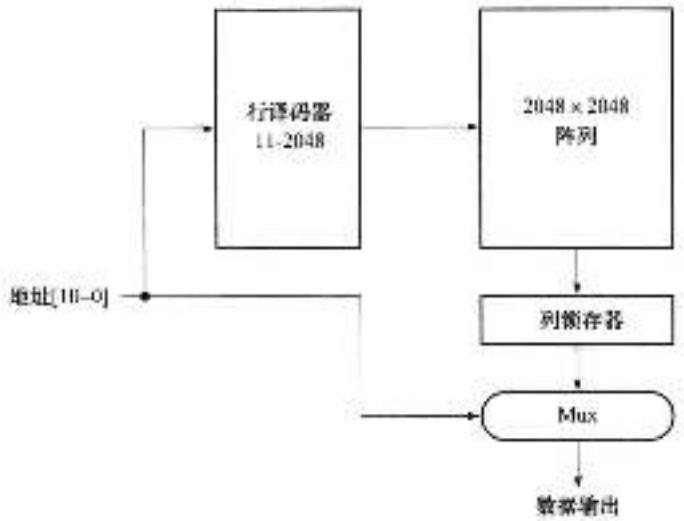



图 B-9-6 用2048×2048阵列组成4M×1 DRAM。采用11位地址选择行，然后再锁存到2048个1位锁存器中。一个多路选择器从2048个锁存器中选择输出。RAS和CAS信号则分别控制地址线是与译码器相连还是与列多选器相连

读者可能会注意到，64M×4的DRAM在每次行存取时能同时存取8K位，而在列存取时就丢弃了几乎所有位，只剩下4位。DRAM设计师早已通过DRAM内部结构实现了更宽的带宽。这个过程可以完成：无需改变行地址，只需能够改变列地址即可，这样就能实现对列锁存

器中的其他位的存取。为了使这个过程更快更精确，地址输入被同步，这样产生了目前主要使用的 DRAM 形式：同步 DRAM 或称为 SDRAM。

1999 年以来，SDRAM 成为大多数基于高速缓存的内存系统中存储芯片的首选。在同步信号控制下，SDRAM 可以快速访问一次突发传输中一行顺序传输的比特流。2004 年，DDRRAM（双倍数据传输率 RAM）是使用最多的 SDRAM 类型。之所以称为双倍数据传输率，是因为在外部时钟的上升沿和下降沿都能传输数据。第 5 章已经提到，这些模式的存储器有利于增加主存的存取带宽，从而进一步匹配 CPU 和高速缓存的速度。

## B.9.3 纠错

因为在大容量存储器中难免发生数据错误，故而大多数系统都会采用各种校验码来检测可能的数据错误。最简单也最常用的是奇偶校验码。在奇偶校验码中，数据中的 1 的个数是有记录的。如果记录数为奇数，则属于奇校验，否则属于偶校验。当数据往内存写入时，校验位也被写上 1 或 0。同时，当数据被读出时，该位也被读出并校验。当内存中的校验位同读出的校验位不一致时，说明数据出错。

B-64 ~ B-65

1 位奇偶校验能检测出大多数仅有一位出错的数据。当某数据有两位出错时，1 位校验法就可能不再奏效。实际上，1 位校验法能测出任何有奇数位出错的数据。但因为出错位数超过 3 位的概率实在很小，所以我们常用 1 位校验码来检测数据中是否有一位出错。遗憾的是，该方法无法确定哪位数字出错。

1 位奇偶校验是一种**检错码**（error detection code）；另一种称为纠错码（Error Correction Code，ECC）的编码则既能检测错误，还能对错误进行纠正。对于大容量主存，许多系统采用的纠错码不仅能检测出两位之内出错的情况，并且还具有纠正功能。这些方法采用多位编码方式，例如，对主存实施编码的最常用方法是每 128 位数据中加入 7 或 8 位纠错码。

**解惑** 1 位奇偶校验码是距离为 2 的编码方法，这就是说，对于数据和校验位而言，任何 1 位数字的改变都会被检测出该数据出错。例如，当改变某个数据位时，校验位就出错；反之亦然。当然，如果我们同时改变两位（两个数据位，或一个数据位和一个校验位），那么奇偶校验位同数据依旧匹配，也就无法检测出错误了。因此，我们将这种校验定义为距离为 2 的校验码。

为了能检测出多于 1 个的错误或纠正一个错误，我们需要距离为 3 的校验码。也就是说，为了判别带校验码的数据是否正确，至少需要有 3 位数字与其他数据不同。假设存在这样的校验码，并且数据中有一位出错，这时，我们就能检测到数据中有一位数字出错，并能纠正；如果有两位数字出错，我们能检测到错误的发生，但无法纠正它。请参考下表的例子，这是对 4 位数据项的距离为 3 的校验码。

B-66

| 数据 | 校验位 | 数据 | 校验位 |
|---|---|---|---|
| 0000 | 000 | 1000 | 111 |
| 0001 | 011 | 1001 | 100 |
| 0010 | 101 | 1010 | 010 |
| 0011 | 110 | 1011 | 001 |
| 0100 | 110 | 1100 | 001 |
| 0101 | 101 | 1101 | 010 |
| 0110 | 011 | 1110 | 100 |
| 0111 | 000 | 1111 | 111 |

为了说明校验过程，我们不妨以0110为例。0110的纠错码是011。该数据发生一位错误的可能情况有以下4种：1110，0010，0100，0111。请读者注意，011既是数据0110的校验码，也是数据0001的校验码。如果校验电路检测到某数值有一位数字出错，它就能肯定是数据0110或0001出错。由于0110的4个1位出错数有两位数字同数据0001不同，因此该校验码能迅速判断是0110数据有误，同时还能加以纠正。为了使两位错误能够被检测出来，简单的方法就是发生两位错的所有组合都有另外一种编码。使用相同编码的方法使码字中有三位不同。但是如果想纠正2位错误，就会得到错误的结果，因为该纠正机制仅对1位出错有效。如果我们想实现对1位、2位都具有纠错功能，那就需要一个距离为4的校验码。

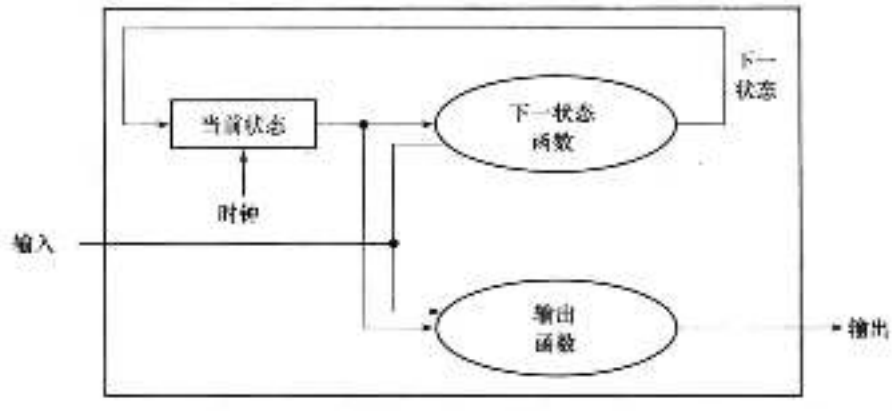
我们在上例中将数据和校验码给区分开了。但事实上，纠错码把编码和数据看作一个更长的字（在例子中是7位）。因此，编码和数据的错误等同对待。

尽管上例中为n位数据需n－1位校验码，但随着数据位数的增加，校验的位数增长较慢。例如，在距离为3的校验码中，64位数据只需7位校验码，128位数据只需8位校验码就能实现。这种校验码叫汉明码，是由R. Hamming首先发明的。

## B.10 有限状态机

前面已经讲过，数字逻辑电路可分为组合电路和时序电路。时序系统的状态存放在存储器中，它们的操作不仅依赖于输入信号，同时也与存储器中的数据、系统的初始状态有关。因此，时序电路无法用真值表加以描述。相反，可以用**有限状态机**（finite-state machine）来描述时序系统。有限状态机有一组状态量和两个函数（输出函数和**下一状态函数**（next-state function））。状态集包括在存储单元中可能出现的所有状态量。这样，对于n位存储器，就可能有$2^n$个状态量。下一状态函数是这样一种组合函数：通过给定输入值和当前状态量来确定后续状态。输出函数根据当前状态量和输入量产生一组输出。图B-10-1是有限状态机的图示。

- 有限状态机：一个包含一套输入、输出函数和下一状态函数的时序逻辑函数。其中下一状态函数根据当前状态和输入产生一个新的状态，输出函数根据当前状态和输入（有时不需要输入）确定输出的控制信号。
- 下一状态函数：是一个组合函数，根据输入和当前状态确定有限状态机的下一状态。




图B-10-1 一个状态机包含存储状态的内部存储单元和两个组合函数：下一状态函数和输出函数。通常输出函数严格视当前状态量为输入，这样就不会改变时序机的能力，但会对内部值造成影响

此处和第4章讨论的状态机都是同步的。也就是说，状态量随着时钟周期变化，并且每个时钟周期内都有新的状态量产生。根据前面的约定，状态量的更新发生在时钟触发沿到来时刻。本节和整个第4章都使用了该方法，但通常不显式的支持时钟。在第4章中，我们用到了 B-67
状态机来控制处理机的执行，并实现了数据路径操作。

为说明有限状态机的操作和设计过程，我们引用简单经典的“交通灯控制”实例加以说明（第4、5章详细描述了利用有限状态机来控制处理机的执行过程）。若将有限状态机用作控制器，输出函数将仅仅依赖于当前状态，这样的状态机称作摩尔机。在本书中，我们都采用了这种有限状态机。如果输出函数既依赖于当前输入，也依赖于当前状态，这样的状态机称为米利机。这两种状态机在功能上是等价的，二者可以互相转化。摩尔机的优点是快速，而米利机则结构小巧（因为后者的状态量个数比摩尔机少）。第5章已详细讨论了它们之间的差别，并展示了使用米利状态机实现的有限状态机的Verilog版本。

下面的例子是关于交通灯控制逻辑的，背景为东西大道和南北大街相交的十字路口。为简单起见，这里只考虑红灯和绿灯；练习题中有另外加上黄灯的逻辑设计。我们希望灯切换的逻辑周期≤30秒。因此采用了频率为0.033Hz的时钟信号，这样就能保证状态间的控制周期≤30秒。其中有两个输出信号： B-68

- NSlite：当信号有效时，南北方向的交通灯为绿色；当信号无效时，南北方向的交通灯为红色。
- EWlite：当信号有效时，东西方向的交通灯为绿色，反之为红色。

另外还有两个输入量：

- NScar：说明有汽车停在监视器处，监视器置于南北方向的交通灯之前。
- EWcar：说明有汽车停在监视器处，监视器置于东西方向的交通灯之前。

只有当其他方向有汽车在等待时，交通灯才会在红绿灯之间切换；否则，交通灯的状态保持为绿色，直到该方向上所有汽车都顺利通过为止。

为完成这个简单的控制，我们还需要两个状态量：

- NSgreen：南北方向的交通灯为绿色。
- EWgreen：东西方向的交通灯为绿色。

下面，我们建立一个下一状态函数表：

| | 输入 | | |
|---|---|---|---|
| | NScar | EWcar | Next state |
| NSgreen | 0 | 0 | NSgreen |
| NSgreen | 0 | 1 | EWgreen |
| NSgreen | 1 | 0 | NSgreen |
| NSgreen | 1 | 1 | EWgreen |
| EWgreen | 0 | 0 | EWgreen |
| EWgreen | 0 | 1 | EWgreen |
| EWgreen | 1 | 0 | NSgreen |
| EWgreen | 1 | 1 | NSgreen |

注意，算法中我们并没有提及当两个方向同时有汽车通行时该怎么办。如果出现这样的情况，上面的状态表需要修改以保证不会导致某一方向出现交通堵塞。

有限状态机可通过指定输出函数加以实现。

| | 输出 | |
|---|---|---|
| | NSlite | EWlite |
| NSgreen | 1 | 0 |
| EWgreen | 0 | 1 |

在考察如何实现这个有限状态机之前，让我们先来看一看用于有限状态机的图形表示。在
B-69 图解中，节点表示状态量，节点中的一串输出是状态值。有向弧用于指出下一状态函数值，带标记有向弧是将输入条件作为逻辑函数。图 B-10-2 为该有限状态机的图形表示。

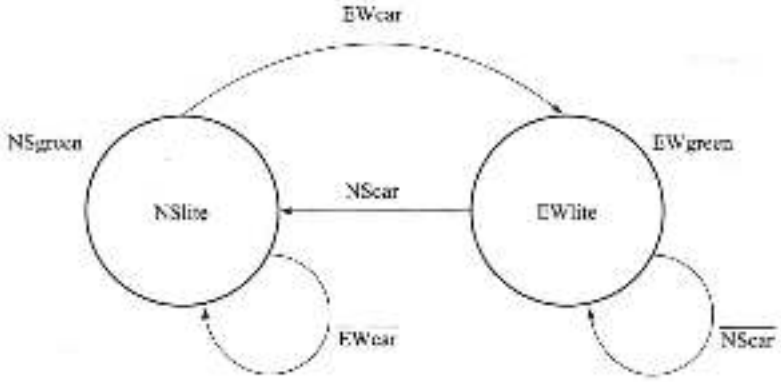



图 B-10-2 2 状态交通信号灯的控制图。其中简化了状态传递的逻辑函数。例如，此状态表中的 NSgreen 到 EWgreen 的传输信号是 $(\overline{NScar} \cdot EWcar) + (NScar \cdot EWcar)$，与 EWcar 相等

有限状态机可这样实现：由寄存器保持当前状态，组合电路计算出下一状态函数和输出函数。图 B-10-3 描述了一个有限状态机的框图：共有 16 个状态量，每个状态量的带宽为 4 位。在实现有限状态机之前，我们先将每个状态标上号，该过程叫状态分配。例如，我们将 NSgreen 标为状态 0，EWgreen 标为状态 1。状态寄存器则只保存 1 位内容。下一状态函数可由以下公式得以计算：

$$NextState = (CurrentState \cdot EWcar) + (CurrentState \cdot \overline{NScar})$$

B-70

其中 CurrentState 是状态寄存器的内容（0 或 1）。NextState 是指下一状态量，该值将在时钟周期结束之前写入寄存器。输出函数也很简单：

$$NSlite = \overline{CurrentState}$$

$$EWlite = CurrentState$$

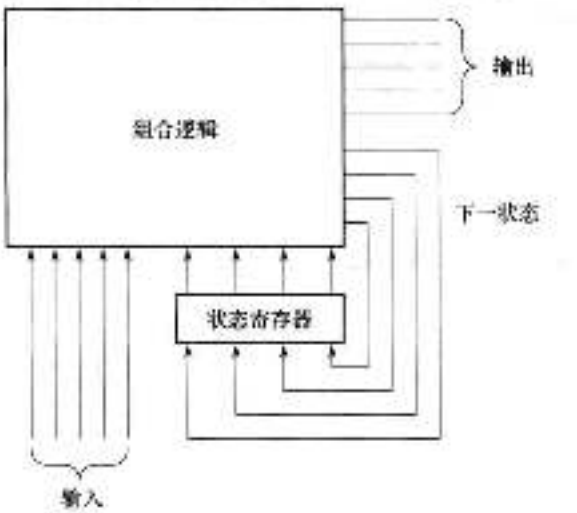



图 B-10-3 有限状态机的实现。有限状态机通过状态寄存器来实现。该寄存器内容包括当前状态和用于计算下一状态和输出函数的组合逻辑块。后面的两个通常采用分离的逻辑模块进行实现，这可能需要更少的门电路

组合电路则采用结构化的逻辑电路得以实现，譬如采用 PLA。PLA 能自动根据下一状态表和输出函数表实现逻辑设计。其实，也有现成的 CAD 工具，它们先将有限状态机图形化或文本化，然后再自动实现优化电路设计。在第 4、5 章中，有限自动机用于控制处理机的执行。附录 D 则详细讨论了用 PLA 和 ROM 来

实现这些控制。

为了说明我们如何使用 Verilog 写出控制逻辑，图 B-10-4 给出了一个用于综合的 Verilog 版本。注意，对于这个简单的控制功能，米利机没有用处，但是在第 5 章中为实现控制功能使用的这种类型定义就是米利机，它比摩尔机控制器拥有的状态更少。 B-71

```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);

   input EWCar, NSCar,clock;
output EWLite,NSLite;

reg state;

initial state=0;  //set initial state

//following two assignments set the output, which is based
// only on the state variable
assign NSLite = ~ state; //NSLite on if state = 0;
assign EWLite = state; //EWLite on if state = 1

always @(posedge clock) // all state updates on a positive
                        //  clock edge
   case (state)
      0: state = EWCar; //change state only if EWCar

      1: state = NSCar; //change state only if NSCar

   endcase
endmodule
```

图 B-10-4 交通灯控制器的 Verilog 描述

**小测验**

要满足米利机所需的状态数比摩尔机所需的状态数少这个条件，摩尔机最少的状态数是多少？

a. 2，因为 1 状态的米利机有可能做相同的事情。

b. 3，因为可以构造一个简单的摩尔机，跳转到两个不同状态之一，并且在此之后总是返回先前状态。有可能构造出两状态米利机实现这一功能。

c. 需要至少 4 个状态，才能体现出米利机的优越性。

## B.11 时钟控制原理

本附录中，我们全部采用时钟边沿作为触发信号。这是因为时钟边沿触发方法比电平触发方法更易于描述和理解。本节将较为详细地阐述时钟控制原理，同时也介绍有关电平触发时钟信号的内容。本节末简单地讨论一下有关异步信号和同步信号的基本原理，这是数字设计中的一个重要问题。 B-72

本节的主要目的是介绍有关时钟控制原理的重要概念。放在章节开始首先假定了一些重要的条件；若有兴趣深入了解时钟控制，可参阅附录末的参考文献。

采用时钟边沿触发方法的优点有两个：易于描述，应用简单。首先，我们假设时钟信号足够长，并且所有时钟都同时到达，那么我们就能保证：对于边沿触发的寄存器位于组合逻辑电路之间的系统，所有的操作都能正确执行，而不会发生竞争现象。如若状态值依赖于不同逻辑单元的相对速度，那么就会发生竞争。在时钟边沿触发电路设计中，时钟周期必须有一定长

度，这样才能满足传输时间（即信号从某个触发器传输到另一个触发器所必须满足的建立时间）。图 B-11-1 描述了采用上升沿触发的系统所必须满足的时钟条件。时钟周期必须至少和下式一样大：

$$t_{pcq} + t_{combinational} + t_{setup}$$

3 个分量表示最差情况下的 3 个延时，分别定义如下：

- $t_{pcq}$是信号通过触发器传播的时间，有时也称为 clock-to-Q。
- $t_{combinational}$是对于任何组合逻辑的最长延时（两个触发器包围的部分）。
- $t_{setup}$即在上升沿到来前，触发器输入必须有效的时间。

另一个假设条件是触发的保持时间要求已满足。这在现代逻辑设计中几乎就不是个问题。

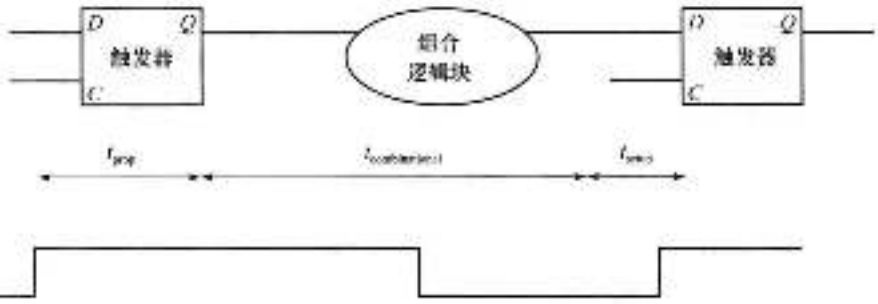



图 B-11-1 在时钟边沿触发的逻辑电路中，需要保证时钟周期足够长，以便保证在下一个时钟触发沿到来之前信号在建立时间内已经有效。信号从触发器输入端传播到触发器输出端的时间为 $t_{pcq}$，然后经过 $t_{combinational}$的时间通过组合逻辑，并在下一个时钟沿到来之前至少 $t_{setup}$时刻有效

B-73

在边沿触发的设计中还必须考虑的一个复杂问题是**时钟偏斜**（clock skew）。时钟偏斜是指两个状态单元看到同一个时钟沿时的绝对时间差。时钟偏斜产生的原因是时钟信号经常沿两条不同的路径传播，导致到达两个状态单元在时间上有差异。如果时钟偏斜足够大，有可能导致一个状态单元发生变化，从而使得另一个触发器的输入端在该触发器看到时钟沿之前变化。

图 B-11-2 剖析了该问题的产生，这里忽略了建立时间和触发器的传输延时。为避免这种错误，可增大时钟周期以克服最大时钟偏移。这样，时钟周期至少应大于：

$$t_{pcq} + t_{combinational} + t_{skew} + t_{setup}$$

有了这个前提条件，就算两个时钟的到达先后次序颠倒，即第二个时钟早到了 $t_{skew}$，整个电路依旧能正常工作。设计人员为减少偏移度，通常需要仔细设计时钟信号经过的路由，争取将偏移减少到最小。另外，聪明的设计师还通过稍稍变大时钟周期的方法来减少时钟偏斜；这些变化在逻辑单元和电源上是允许的。正因为考虑到时钟偏斜会影响保持时间，所以使偏移度尽量小至关重要。

⇨ 时钟偏斜：两个状态单元看到时钟沿的绝对时间差别。

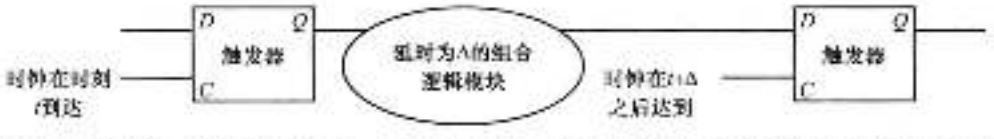



图 B-11-2 时钟偏斜可能引发竞争现象，引起错误操作。由于两个触发器检测时钟信号的时刻存在差异，则可能导致存储在每片的信号都会向前传输，在触发器 2 的时钟有效信号到达之前改变了触发器 2 的输入值

时钟边缘触发有两个缺点：需要额外的逻辑电路，有时会增加时延。比较 D 触发器和电平触发锁存器，我们会发现前者需要更多的逻辑电路。另一种方法是采用**电平触发时钟控制**（level-sensitive clocking）。因为电平触发机制下状态变化不是瞬间完成的，为使各项操作能正确执行，该方法会更加复杂，需要考虑更多因素。

⊃ **电平触发时钟控制**：一种在时钟高电平或低电平期间进行状态改变的时序控制方法，其状态变化没有边沿触发方式快。 B-74

## B.11.1 电平触发时控原理

在电平触发时控机制下，状态量的改变发生在高电平或低电平期间，因为它们不采用时钟边沿触发，所以这些变化并不能瞬间完成，于是就会有竞争现象产生。为了保证在时钟足够慢的情况下，电平触发电路照样能正常工作，设计中用到了双相时钟控制。双相时钟控制中用到了两个互不重叠的时钟信号。参照图 B-11-3，任何时刻至多有一个时钟信号处于高电平状态。这样，我们采用双相时钟（记作 $\phi_1$、$\phi_2$）构建的系统就克服了竞争现象，效果与时钟边沿触发电路一样。

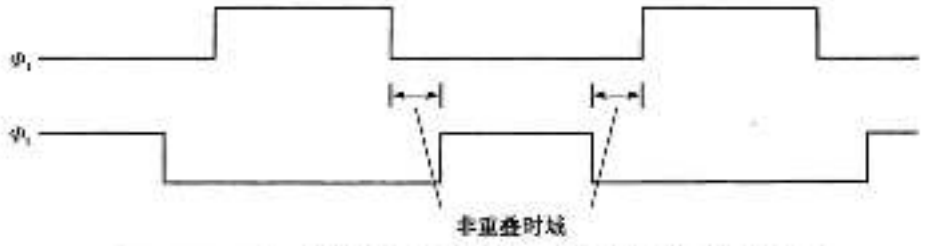



图 B-11-3 双相时钟机制下展示了每个时钟周期和非重叠的阶段

一种简单的设计方案是采用在 $\phi_2$ 打开的锁存器取代在 $\phi_1$ 打开的锁存器。因为两个时钟不是同时处于有效状态，于是竞争现象就不会发生。如果组合电路的输入控制信号为 $\phi_1$，那么它的输出态会在 $\phi_2$ 时钟周期内被锁存起来。这样当输入锁存器关闭时，其输出值只能在 $\phi_2$ 的有效信号期间开放，于是能保证输出信号有效。图 B-11-4 描述了双相时钟控制系统和交替锁存器机理是如何实现的。如同在时钟边沿触发电路设计中那样，我们必须考虑并协调好时钟偏斜，尤其是时钟的相位关系。如果能减少两个时钟信号在相位上的重叠分量，就会减少出错的概率。如果每个时钟信号周期足够长，双相不重叠部分足够大，那就可以有效地保证电路运行的正确性。 B-75




图 B-11-4 双相时钟机制下的系统操作和时钟信号。图示表明系统在两个时钟状态下如何运作，在同本身 C 输入反相的时钟状态下，锁存器保持稳定。所以在时域 $\phi_2$ 中，第一个组合块有一个稳定的输入，其输出则可以被 $\phi_2$ 锁存。同理，第二个组合块也在其反相时域 $\phi_1$ 中保持稳定的输入。因此经过组合块的时延决定了各时钟必须有效的最短时间。非重叠时域则由任意逻辑块的最大时钟偏移和最小时延共同决定

## B.11.2 异步输入和同步器

如果时钟偏斜问题得到了解决，只要采用单时钟信号或双相时钟机制，就能完全消除竞争

现象。但是，要想在整个系统中仅使用一个时钟信号并且做到时钟偏斜很小，这在实际电路中不太现实。实际系统中，CPU有自己的时钟信号，而I/O设备也有自己的时钟信号。一个异步设备可以通过一系列的握手与CPU进行通信。为了将异步输入信号转化为同步信号并用于改变系统的状态量，就需要采用同步器。在同步器中，输入为异步信号和独立的时钟信号，其输出是与此输入时钟同步的信号。

要设计同步器，首先得采用D触发器，如图B-11-5所示，其输入信号D是异步信号。我们还采用了握手通信协议。因为异步信号将一直保持着有效状态直到被确认，所以，无论是在当前周期还是后续时钟周期检测到异步信号并不很重要，于是除了一个小问题之外，就不难理解这种简单设计足以实现正确采样了。

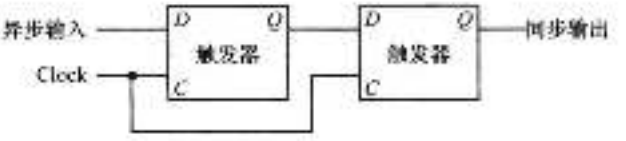



图B-11-5 由D触发器组成的同步器。用于采样异步信号，并将产生和时钟保持同步的输出信号。这个同步器不能完全正确工作

但实际电路设计中还存在一个**亚稳态**（metastability）的问题。假设在时钟触发沿到来之前，异步信号一直在高低电平间振荡；那么，显然难以判断到底是信号的高电平还是低电平被锁存起来了。虽然这个问题可以克服，但真正头痛的问题在于：如果采样信号的建立时间和保持时间不满足基本要求，触发器可能会进入一种亚稳态。这时，输出信号既不是高电平也不是
B-76 低电平，而是介于二者之间的电平。并且，还将无法保证触发器在有限的时间内稳定下来。同时，与该触发器相关的逻辑模块读到的输出信号也不一致：有时是0，有时是1，这种现象叫作**同步失败**（synchronizer failure）。

- 亚稳态：如果采样时信号不满足建立时间和保持时间的要求，采样所得的数据可能是介于高电平和低电平之间的一个错误值。
- 同步失败：触发器进入亚稳态状态，并且有些逻辑模块读到触发器输出为0，而另外一些模块读到触发器的输出为1。

在一个同步系统中，通过使建立时间和保持时间满足要求，可以确保系统避免同步失败，但是当输入是异步信号时会出现例外。唯一可能的解决方案是当输出信号稳定，或当触发器退出亚稳态后再接收该同步器的输入信号，但这样会增加等待时间。那究竟该等多长时间呢？触发器处于亚稳态的概率是按指数级衰减的，在较短的时间内，此概率就衰减得很小；但永远不会为0！因此要想使同步失败的概率很小，就需经过几年甚至几千年时间。




图B-11-6 只要亚稳态的时间小于时钟周期，该同步信号器就能工作正常。尽管第一个触发器的输出端可能会是亚稳态，但任何其他的逻辑单元在第二个时钟之前看不到亚稳态。在第二个D触发器在第二个时钟采样信号时，第一个触发器的输出将不再处于亚稳态

通常情况下，经过几个建立时间后，失败的概率就衰减得很小了。当时钟周期比亚稳态周期要长时，就需要两个D触发器以保证安全性（见图B-11-6）。有兴趣的读者可进一步阅读参考文献。

**小测验** 假设要设计的电路具有非常大的时钟偏斜，其时钟偏斜程度超过了寄存器的传播时间

(propagation time)。那么是否有可能通过设计来减慢时钟以使得逻辑操作能正确运行?

a. 可以，即使时钟偏斜非常大，但是只要时钟慢到一定的程度，那么信号总是能正常地传输，整个电路设计就能正确运行。

b. 不可以，因为有可能存在这样一种情况：两个寄存器对于时钟边沿的感知时间相差过大，导致在同一个时钟边沿时，其中一个寄存器会发现另一个寄存器已经被触发，并且输出已经传输出去了。

➲ 传播时间：将触发器的输入传播到触发器的输出需要的时间。 B-77

## B.12 现场可编程设备

对于定制或半定制芯片，设计者可以利用底层结构提供的灵活性方便地实现组合或时序逻辑。对于那些不想使用定制或半定制 IC 而又想利用高级集成电路实现复杂逻辑功能的设计者来说，**现场可编程器件**（field programmable device，FPD）是使用最普遍的器件了。FPD 是包括组合逻辑（可能还有内存设备）的集成电路，FPD 可由使用者配置。

➲ 现场可编程器件（FPD）：一种包含组合逻辑，可能也包含存储设备的集成电路，最终用户可以对其进行配置。

FPD 主要分为两个阵营：**可编程逻辑器件**（programmable logic device，PLD），它是纯粹的组合逻辑；**现场可编程门阵列**（field programmable gate array，FPGA），它提供组合逻辑和触发器。PLD 由两种形式组成：**简单 PLD**（simple PLD，SPLD），就是通常的 PLA 或者**可编程阵列逻辑**（programmable array logic，PAL）；还有就是复杂 PLD，它包括多于一个的逻辑模块以及模块间的可配置互连线路。当谈到 PLD 中的 PLA 时，通常是指带有用户可编程与阵列和或阵列的 PLA。PAL 类似于 PLA，除了它的或阵列是固定的以外。

➲ 可编程逻辑器件（PLD）：包含最终用户可配置功能的组合逻辑的集成电路。

➲ 现场可编程门阵列（FPGA）：一种可配置的包含组合逻辑模块和触发器的集成电路。

➲ 简单 PLD（SPLD）：一种可编程逻辑器件，通常包含一块 PAL 或 PLA。

➲ 可编程阵列逻辑（PAL）：由一个可编程的与阵列后跟一个可编程的或阵列组成的可编程逻辑电路。

在讨论 FPGA 以前，先来看看 FPD 是如何配置的。配置的主要问题就是在何处建立或打破连接。门电路和寄存器是静态的，但是连接是可配置的。注意，通过配置连接，用户可以决定实现何种逻辑功能。考虑一个可配置的 PLA：通过决定与阵列和或阵列在何处连接，用户决定 PLA 中运算的逻辑。FPD 中的连接可以是永久的，也可以是可配置的。永久连接涉及两个连线之间建立或破坏连接。现在的 FPLD 都使用反熔丝（antifuse）技术，允许在编程时建立连接然后再永久固定下来。配置 CMOS FPLD 的另外一种方法是使用 SRAM。在上电时，配置信息下载到 SRAM 中，这些内容控制开关设定进而决定哪些金属线连接起来。FPD 使用 SRAM 控制的好处在于可以通过修改 SRAM 的内容进行重新配置。基于 SRAM 控制的两个缺点是：配置信息是易失的，必须在上电时重新加载；为了切换使用主动晶体管会给线路增加一点电阻。

➲ 反熔丝：集成电路中的一种结构，当对其进行编程时，将导致线间的永久性连接。

包含逻辑和存储设备的 FPGA 通常是二维阵列结构，划分行、列的通道用来进行阵列单元间的互连。每个单元是门和触发器的组合，可以编程执行特定功能。因为可编程的 RAM 通常 B-78

很小，它们也被称为**查找表**（lookup table，LUT）。更新的 FPGA 包括更复杂的构建模块，例如加法器和用来构建寄存器文件的存储模块。一些大型的 FPGA 甚至包含 32 位的 RISC 核心。

⇒ 查找表（LUT）：现场可编程器件中的单元的名称，包含少量的逻辑和 RAM。

除了可以对每个单元进行编程执行特定的功能，单元间的互连也是可编程的，这就使得现在包含上百模块和上千门电路的 FPGA 可以实现复杂的逻辑功能。互连是可定制芯片的最大挑战，对于 FPGA 也是如此，因为单元不能表示结构化设计分解的自然功能单元。许多 FPGA 有 90% 的部分用来实现互连，只有 10% 是逻辑和存储模块。

正如你不可能不使用 CAD 工具来设计定制或半定制芯片，你也需要 CAD 工具来设计 FPD。已经开发出针对特定 FPGA 的逻辑合成工具，帮助从结构或行为 Verilog 描述中使用 FPGA 生成系统。

## B.13 结论

本附录介绍了逻辑设计的一些基本概念和原理。在了解了这些内容之后，请参阅第 4，5 章，那些内容是本附录的应用和进一步深入。

### 拓展阅读

关于逻辑电路设计，有很多好书，以下列出了其中一些。

一本详细介绍用 Verilog 进行逻辑设计的书：

Ciletti, M. D. [2002]. *Advanced Digital Design with the Verilog HDL*. Englewood Cliffs, NJ: Prentice Hall.

一本关于逻辑设计的书：

Katz, R. H. [2004]. *Modern Logic Design*, 2nd ed., Reading, MA: Addison Wesley.

B-79

一本关于逻辑设计的书：

Wakerly, J. F. [2000]. *Digital Design: Principles and Practices*, 3rd ed., Englewood Cliffs, NJ: Prentice Hall.

## B.14 练习题

**B.1** [10] <B.2> 除了这一部分我们讨论过的基本规律之外，还有两个重要的定理，叫作德·摩根定理：

$$\overline{A + B} = \overline{A} \cdot \overline{B} \quad \text{和} \quad \overline{A \cdot B} = \overline{A} + \overline{B}$$

使用下面的真值表对上面的德·摩根定理进行证明：

| A | B | $\overline{A}$ | $\overline{B}$ | $\overline{A+B}$ | $\overline{A}\cdot\overline{B}$ | $\overline{A\cdot B}$ | $\overline{A}+\overline{B}$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

**B.2** [15] <B.2> 用德·摩根定理和 B.2 节中的结合公理证明例题中关于 E 的两个表达式是等价的。

**B.3** [10] <B.2> 证明对于 $n$ 输入的逻辑函数，对应的真值表有 $2^n$ 项。

**B.4** [10] <B.2> 异或函数具有多种用途（可用于加法器或用来计算校验码）。对于二输入的异或函数，当且仅当一个输入值为“真”时输出才为“真”。写出二输入异或函数的真值表，并用与门、或门和反相器实现该函数。

**B.5** [15] <B.2> 通过使用二输入的或非门实现与、或、非三种逻辑功能，证明利用或非门可以实现任何逻辑功能。

**B.6** [15] <B.2> 通过使用二输入的与非门实现与、或、非三种逻辑功能，证明利用与非门可以实现任何逻辑功能。

**B.7** [10] <B.2, B.3> 写出四输入奇校验函数的真值表（关于错误校验的内容参见 B.9.3 节）。

**B.8** [10] <B.2, B.3> 用输入端和输出端带有反向小圆圈的与门和或门实现四输入的奇校验函数。

**B.9** [10] <B.2, B.3> 用 PLA 实现四输入的奇校验函数。 B-80

**B.10** [15] <B.2, B.3> 通过使用多路选择器实现与非门（或者或非门）。证明二输入多路选择器可以实现任何逻辑功能。

**B.11** [5] <4.2, B.2, B.3> 假设 $X$ 由三位 x2、x1、x0 组成。分别写出下列 4 个逻辑表达式（当且仅当满足下面的条件时逻辑表达式为“真”）：

- $X$ 中只有一个 0。
- $X$ 中有偶数个 0。
- 当 $X$ 被当做无符号二进制数时，$X$ 小于 4。
- 当 $X$ 被当做有符号二进制数时，$X$ 是负数。

**B.12** [5] <4.2, B.2, B.3> 用 PLA 实现练习题 B.11 的 4 个逻辑函数。

**B.13** [5] <4.2, B.2, B.3> 假设 $X$ 由二位 x2、x1、x0 组成，$Y$ 由三位 y2、y1、y0 组成。写出下列 3 个逻辑表达式（当且仅当满足下面的条件时逻辑表达式为“真”）：

- 当 $X$、$Y$ 被当做无符号二进制数时，$X<Y$。
- 当 $X$、$Y$ 被当作有符号（二进制补码）数时，$X<Y$。
- $X=Y$。

使用可以扩展到多位的层次表达方法，写出如何扩展为 6 位比较。

**B.14** [5] <B.2, B.3> 用逻辑电路实现开关网络：输入为 $A$ 和 $B$；输出为 $C$ 和 $D$；控制信号为 $S$。当 $S=1$ 时，网络为直通模式，即 $C=A$，$D=B$；当 $S=0$ 时，网络为交叉模式，即 $C=B$，$D=A$。

**B.15** [15] <B.2, B.3> 由 B.2 节中 $E$ 的“和之积”形式推出其“积之和”形式。你需要使用德·摩根定理。

**B.16** [30] <B.2, B.3> 设计一个算法，该算法能够对任何包含与、或、非逻辑的函数构建其“积之和”形式的表达式。算法应当具有递归性，并且在整个过程中不能产生真值表。

**B.17** [5] <B.2, B.3> 写出多路选择器的真值表（输入为 $A$、$B$ 和 $S$，输出为 $C$），通过使用无关项来简化真值表。 A-81

**B.18** [5] <B.3> 下面的 Verilog 模块实现了何种功能：

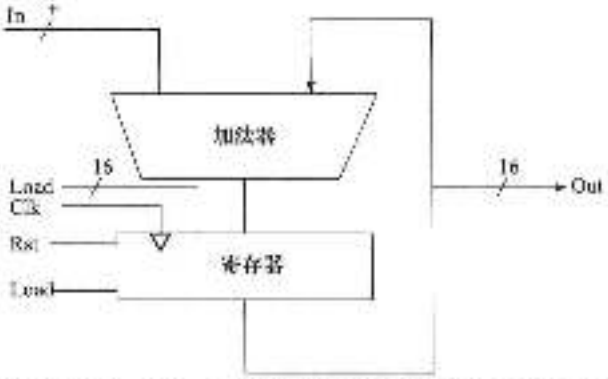
```
module FUNC1 (I0, I1, S, out);
      input I0, I1;
      input S;
      output out;
      out = S? I1: I0;
endmodule

module FUNC2 (out,ctl,clk,reset);
      output [7:0] out;
      input ctl, clk, reset;
      reg [7:0] out;
      always @(posedge clk)
      if (reset) begin
                  out <= 8'b0 ;
      end
      else if (ctl) begin
                  out <= out + 1;
      end
      else begin
                  out <= out - 1;
      end
endmodule
```

**B.19** [5] <B.4> B.8.1 节给出了 D 触发器的 Verilog 代码，请给出 D 锁存器的 Verilog 代码。

**B.20** [10] <B.3，B.4> 写出 2－4 译码器（与/或阵结构）的 Verilog 模块实现。

**B.21** [10] <B.3，B.4> 根据下面给出的累加器逻辑图，写出它的 Verilog 模块实现。假定使用正边沿触发寄存器和异步 Rst。

A-82




**B.22** [20] <B.3，B.4，B.5> 3.3 节介绍了乘法器的基本操作和可能的实现。这个实现的基本单元是一个移位加法单元。给出这个单元的 Verilog 实现，并说明如何使用这个单元建立 32 位乘法器。

**B.23** [20] <B.3，B.4，B.5> 根据上一题，实现无符号除法器。

**B.24** [15] <B.5> ALU 支持只使用加法器的符号位设置小于（slt）。用这种方法比较 $-7_{10}$ 和 $6_{10}$。为简单起见，使用 4 位二进制表示：$1001_2$ 和 $0110_2$。

1001 − 0110 = 1001 + 1010 = 0011，

这个结果表示 −7 > 6，这显然是错误的。同此在判断时必须考虑到溢出，修改图 B-5-10 中的 1 位 ALU 来正确处理 slt。为了节省时间可以直接复印图，在其上改。

**B.25** [20] <B.6> 在加法中检查溢出的一个简单方法是看最高有效位的 CarryIn 是否和最高有效位的 CarryOut 相同。证明这个方法和图 3-2 是一样的。

**B.26** [5] <B.6> 使用新定义重写 B.6.3 节中 16 位加法器的超前进位逻辑公式。第一，使用加法器独立位 CarryIn 信号的名字，即使用 c4，c8，c12，…，而不是使用 C1，C2，C3，…；另外，$P_{i,j}$ 表示 $i$ 位到 $j$ 位的传播信号，$G_{i,j}$ 表示 $i$ 位到 $j$ 位的生成信号，例如，公式

B-83

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

可改写成

$$c8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c0)$$

这个更通用的定义在建立位数更宽的加法器时有用。

**B.27** [15] <B.6> 使用练习题 B.26 的新定义写出 64 位加法器的超前进位逻辑公式，使用 16 位加法器作为基础模块。并给出类似图 B.6.3 的图。

**B.28** [10] <B.6> 下面计算加法器的相对性能。假定针对某个公式的硬件运行时间为一个时间单位 $T$，这个公式只包含与运算或者或运算，例如 B.6.2 节的 $pi$ 和 $gi$ 公式。由几个与项进行或运算构成的公式运行时间需要 $2T$，例如 B.6.2 节中的公式 c1、c2、c3 和 c4。这个时间包括计算与运算的时间 $T$ 和计算或运算的额外时间 $T$。分别计算 4 位行波进位加法器和超前进位加法器的运算次数来评价性能。如果公式中的项由其他公式定义，增加中间公式带来的相应时延。反复迭代直到公式中使用的都是加法器的实际输入为止。并且给每个加法器标出计算时延并标明最坏情况时延。

**B.29** [15] <B.6> 类似练习题 B.28，不过这次只计算 16 位的相对速度。加法器的结构分别是：（1）行波进位加法器；（2）4 位一组，组内超前进位，组间行波进位；（3）采用 B.6.2 节所示的超前进位加法器。

**B.30** [15] <B.6> 与练习题 B.28 和 B.29 类似，本题计算 64 位加法器的相对速度，加法器的结构分别是：（1）行波进位加法器；（2）4 位一组，组内超前进位，组间行波进位；（3）16 位一组，组内超前进位，组间行波进位；（4）采用练习题 B.27 中的超前进位加法器。

**B.31** [10] <B.6> 如果我们不把加法器看成一个把两个数相加然后与进位连接到一起的装置，而是将其看成可以把三个数（ai，bi，ci）相加，并且产生两个输出（s，ci+1）的硬件装置。当进行两个数的加法时，我们并不能据此做些什么。但是当我们进行两个以上操作数的加法时，就可以通过上述想法降低进位开销。该想法是构造两个独立的“和”，分别叫作 S'（和数位）和 C'（进位位）。在这一过程的末尾，我们需要用一个普通的加法器把 S'和 C'加到一起。这个把进位传播推迟到加法运算最后阶段的技巧称为进位保留加法。图 B-14-1 右下角的模块图显示了该加法器的结构，该结构中两个进位保留加法器通过一个普通加法器连接到一起。

对于具有 4 个 16 位二进制数的加法运算，分别计算采用完全超前进位加法器和带有超前进位加法器（用来形成最终的累加和）的进位保留加法器的时延。（时间单位 T 与练习题 B.28 相同） B-84

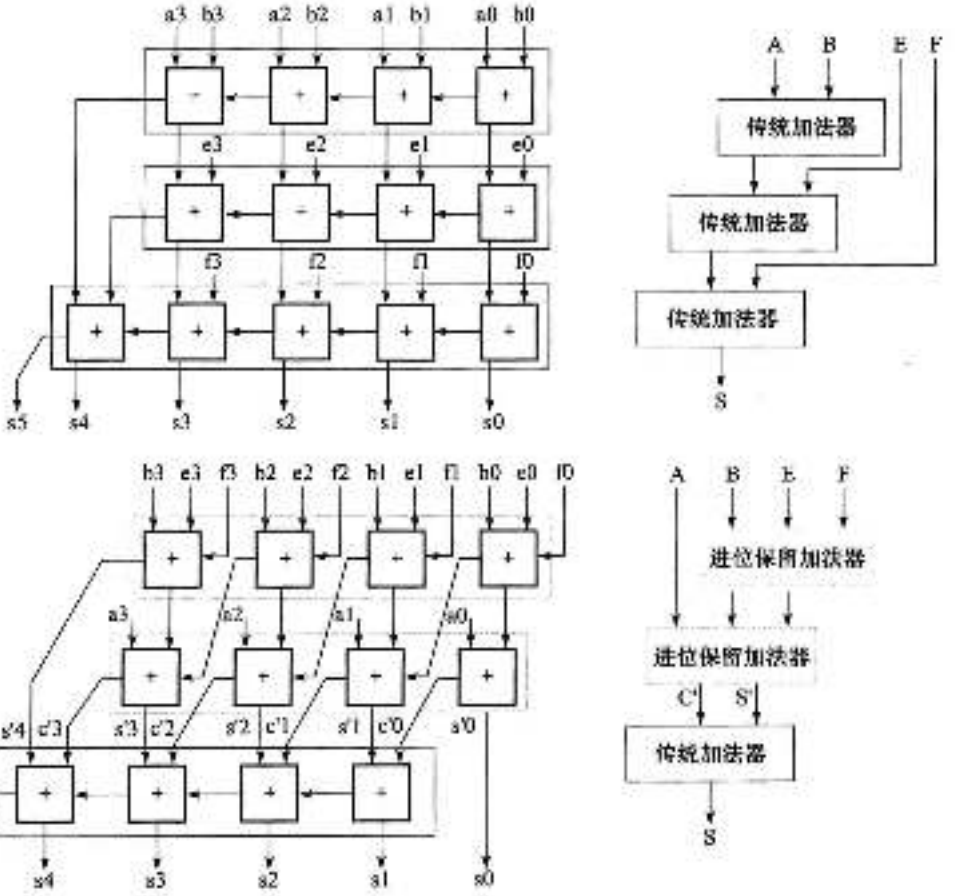



图 B-14-1 4 个 4 位数相加的传统行波进位和进位保留加法器。加法器细节见左边，单独信号小写表示，相应高层模块见右图，组合信号大写表示。注意，4 个 n 位数的和需要 n+2 位

**B.32** [20] <B.6> 在计算机当中最有可能同时把多个数相加到一起的情形恐怕就是当我们试图在一个时钟周期中通过采用多个加法器将多个数相加的办法来加快乘法运算的速度。相比于第 3 章提到的乘法算法，具有多个加法器的进位保留方案可以实现 10 倍以上的乘法运算速度。本习题对采用组合逻辑乘法器计算两个 16 位正数乘法的开销和速度进行评估。假设存在 16 个部分积 M15，M14，…，M0，这些部分积分别表示被乘数与乘数的每一位（m15，m14，…，m0）进行“与”运算的结果。我们的想法是用进位保留加法器将 n 个操作数减少到 2n/3 个并行组，每组 3 个，反复迭代直至得到两个大数，最后用普通加法器把二者加到一起。

首先，根据图 B-14-1 右半部分所示，画出 16 位进位保留加法器的组织结构，用来实现 16 个部分积的相加。然后计算把这 16 项加到一起的时延。将计算出的结果与第 3 章中的迭代乘法方案进行

比较，但需要注意的是，这里假定16次迭代过程中使用的是具有完全超前进位的16位加法器，
B-85 该加法器的速度在练习题B.29中已经计算过。

**B.33** [10] <B.6> 有时用户想要将一组数一同加起来。例如，使用1位完全加法器将4个4位数（A，B，E，F）加起来。现在忽略超前进位，将1位加法器按图B-14-1的组织形式连接起来。在传统组织形式之下是完全加法器的新组织形式。使用这两种组织结构实现4个数的加法，并确保能够得到相同结果。

**B.34** [5] <B.6> 首先给出如图B-14-1所示的16位进位保留加法器的模块组织结构。假定通过一个1位加法器的时间是2T，计算上下两个组织结构进行4个4位数加法所需的时间。

**B.35** [5] <B.8> 很多时候你可能希望得到这样的时序图，该时序图包含了对发生在数据输入端 $D$ 和时钟输入端 $C$（类似于B.8节图B-8-3和图B-8-6）的变化的描述，通常D锁存器和D触发器的输出端波形（$Q$）是不同的。用一两句话描述使二者输出端波形相同的条件（即输入信号需要满足的条件）。

**B.36** [5] <B.8> B.8节图B-8-8描述了MIPS数据通路的寄存器文件实现。假设需要建立一个新的寄存器文件，但是只有两个寄存器和一个读端口，并且每个寄存器只有两位数据。重绘图B-8-8，使得每根连接线仅与一位数据相连（不像图B-8-8中那样，有些连接线为5位，有些则为32位）。采用D触发器重绘图中的寄存器。无需画出D触发器或多路选择器的具体实现。

**B.37** [10] <B.10> 有个朋友想让你帮忙设计一个防安全装置的“电子眼”。该设备由排成一行的三个灯组成，这三个灯分别受输出Left、Middle和Right控制，即当这三个信号当中的某一个有效时，对应的灯被点亮。每次仅有一个灯被点亮，并且灯光先从左到右“移动”，然后再从右到左，这样可以吓跑那些误以为该设备正在监视其行踪的小偷。画出用于控制该“电子眼”的有限状态机图示。需要注意的是，“眼睛”的移动速率受时钟速度（不应当过高）的控制，并且根本没有输入信号。

**B.38** [10] <B.10> 为上题中的有限状态机分配状态编码，并写出对应于每个输出信号的包含下一状态
B-86 位的逻辑表达式。

**B.39** [15] <B.2，B.8，B.10> 用3个D触发器和若干逻辑门构造一个3位计数器。计数器的输入包括复位信号reset，计数值增加信号inc。计数结果作为计数器输出。当计数值为7并且继续增加时，计数值应当重新归零。

**B.40** [20] <B.10> 格雷码是具有这样性质的二进制序列，即该序列当中相邻的两个编码最多有一位不同。例如，下面是一个3位格雷码序列：000，001，011，010，110，111，101，和100。用三个D触发器和一个PLA实现一个3位格雷码计数器，要求该计数器具有两个输入信号：复位信号reset和增量信号inc，其中reset信号将计数器设为000，inc信号将使计数器进入序列中的下一状态。需要注意的是，该编码序列是循环的，所以100的下一个值为000。

**B.41** [25] <B.10> 我们希望在B.10节的交通灯例子中添加一个黄灯。通过将时钟频率改为0.25Hz（时钟周期4秒，即黄灯的持续时间）。为防止绿灯和红灯循环过快，我们加入了一个30秒的计时器。该计时器只有一个输入信号TimerReset，该信号用于对计时器进行重启；计时器输出信号TimerSignal表示30秒时间已经过去。而且，为了把黄灯包含进去，我们必须重新定义交通信号。我们通过给每个灯定义两个输出信号（green和yellow）来实现。如果输出NSgreen有效，绿灯被点亮；如果输出NSyellow有效，黄灯被点亮。如果两个信号都无效，则红灯被点亮。green和yellow信号不能同时有效，否则美国司机见到之后肯定会感到困惑，即使欧洲司机明白其中的含义！画出上述改进后的控制器对应的有限状态机图示。状态名称不要和输出信号同名。

**B.42** [15] <B.10> 写出练习题B.41中交通灯控制器的下一状态表和输出函数表。

**B.43** [15] <B.2，B.10> 为练习题B.41的交通灯分配状态编码，并根据练习题B.42的表格写出每个输出信号的逻辑表达式，包括下一状态的输出。

**B.44** [15] <B.3，B.10> 用PLA实现练习题B.43的逻辑表达式。

## 01 小测验答案

**B.2** 否。如果 $A=1$，$C=1$，$B=0$，那么第一个为真，第二个为假。

**B.3** C

**B.4** 全部相同。

**B.4** $A=0$，$B=1$。

**B.5** 2。

**B.6** 1。

**B.8** c。

**B.10** b。

**B.11** b。

B-87

# 索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。注意：在网上的信息以章节后续页码的方式列出（OL.3.11-7），附录中的信息以字母后续页码的方式列出。


1-bit ALU（一位 ALU），见 Arithmetic logic unit（ALU），B-26-29
  adder（加法器），B-27
  CarryOut，B-28
  for most significant bit（最高有效位），B-33
  illustrated（配有插图），B-29
  logic unit forAND/OR（与/或逻辑运算单元），B-27
  performing AND，OR，and addition（执行与或和加法），B-31，B-33
32-bit ALU（32 位 ALU），见 Arithmetic logic unit（ALU），B-29-38
  defining in Verilog（在 Verilog 中定义），B-35-38
  from31 copies of 1-bit ALU（复制 31 个一位 ALU），B-34
  illustrated（举例说明），B-36
  ripple carry adder（行波进位加法器），B-29
  tailoring to MIPS（适用于 MIPS），B-31-35
  with 321-bit ALUs（用 32 个一位 ALU），B-30
32-bit immediate operands（32 位立即数），112-113
7090/7094 hardware（32 位立即数），OL3.11-7

A

Absolute references（绝对引用），126
Abstractions（抽象），
  hardware/software interface（软/硬件分界面），22
  principle（原理），22
  to simplify design（为简化设计），11
Accumulator architectures（累加器结构），OL2.21-2
Acronyms（首字母缩略词），9
Active matrix（有源矩阵），18
add（Add）（加），64
add.d（FP Add Double，双精度浮点数加法），A-75
add.s（FP Add Single，单精度浮点数加法），A-74
Add unsigned instruction（无符号整数加法指令），180
addi（Add Immediate）（加立即数），64
Addition（加法），见 Arithmetic，178-182.
  binary（二进制），178-179
  floating-point（浮点型），203-206，211，A-75-74
  instructions（指令），A-51
  operands（操作数），179
  significands（有效数），203
  speed（速率），182
addiu（Add Imm. Unsigned）（加无符号整型立即数），119
Address interleaving（地址交错），381
Address select logic（地址选择逻辑），D-24，D-25
Address space（地址空间），428，431
  extending（扩展），479
  flat（平面），479
  ID（ASID）（地址空间标识符），446
  inadequate（不足），OL5.17-6
  shared（共享的），519-520
  single physical（单一物理地址空间），517
  unmapped（未映射的），450
  virtual（虚拟的），446
Address translation（地址转换）
  for ARM cortex-A8（ARM 公司 cortex-A8 芯片），471
  defined（定义），429
  fast（快速），438-439
  for Intel core i7，471（英特尔酷睿 i7），
  TLB for（快表），438-439
Address-control lines（地址控制线），D-26
Addresses（地址）
  32-bit immediates（32 位立即寻址），113-116
  base（基址），69
  byte（字节型），69
  defined（定义），68
  memory（主存），77
  virtual（虚拟的），428-431，450
Addressing（寻址）

32-bit immediates（32 位立即寻址），113-116
base（基址），116
displacement（移位），116
immediate（立即寻址），116
in jumps and branches（跳转和分支），113-116
MIPS modes（MIPS 方式），116-118
PC-relative（相对于程序计数器），114，116
pseudodirect（伪直接），116
register（寄存器），116
x86 modes（x86 方式），152
Addressing modes（寻址方式），A-45-47
desktop architectures（台式机体系结构），E-6
addu（Add Unsigned，加无符号整数），64
Advanced Vector Extensions（AVX，高级适量扩展指令集），225，227
AGP（加速图形接口），C-9
Algol-60，OL2.21-7（算法语言-60），
Aliasing（别名使用），444
Alignment restriction（调整限制），69-70
All-pairs N-body algorithm（All-pairs N-body 算法），C-65
Alpha architecture（Alpha 体系结构）
bit count instructions（位计算指令），E-29
floating-point instructions（浮点数指令），E-28
instructions（指令），E-27-29
no divide，E-28
PAL code（可编程逻辑阵列代码），E-28
unaligned load-store（未对齐存储），E-28
VAX floating-point formats（浮点格式虚拟地址扩展器），E-29
ALU control（ALU 控制），见 Arithmatic logic unit（ALU），259-261
bits（比特），260
logic（逻辑），D-6
mapping to gates（映射到逻辑门），D-4-7
truth tables（真值表），D-5
ALU control block（ALU 控制块），263
defined（定义），D-4
generating ALU control bits（生成 ALU 控制位），D-6
ALUOp（ALU 操作码），260，D-6
bits（比特），260，261
control signal（控制信号），263
Amazon Web Services（AWS）（亚马逊网络服务），425
AMD Opteron X4（Barcelona）（超微半导体公司皓龙处理器 X4（巴塞罗那）），543，544
AM（又称“x86-64”或“x64”是一种 64 位的计算机处理器架构），D64，151，224，OL2.21-6
Amdahl's law（Amdahl 定律），401，503
corollary（推论），49
defined（定义），49
fallacy（谬误），556
and（AND）（与），64
AND gates（与门），B-12，D-7
AND operation（与操作），88
AND operation（与操作），A-52，B-6
andi（And Immediate）（立即数与），64
Annual failure rate（AFR）（年故障率），418
$1/j$ MTTF of disks（磁盘平均无故障时间），419-420
Antidependence（反依赖），338
Antifuse（反熔丝），B-78
Apple computer（苹果计算机公司），OL1.12-7
Apple iPad 2 A1395，20（苹果平板计算机），
logic board of（逻辑板），20
processor integrated circuit of（处理器集成电路），21
Application binary interface（ABI，应用二进制接口），22
Application programming interface（API，应用程序编程接口）
defined（定义），C-4
graphics（图像），C-14
Architectural registers（体系结构级寄存器），347
Arithmetic（算术运算），176-236
addition（加法），178-182
addition and subtraction（加减法），178-182
division（除法），189-195
fallacies and pitfalls（谬误及陷阱），229-232
floating-point（浮点型），196-222
historical perspective（历史展望），236
multiplication（乘法），183-188
parallelism and（并行与操作），222-223
Streaming SIMD Extensions and advanced vector extensions in x86（单指令流多数据流扩展和高级矢量扩展指令集），224-225
subtraction（减法），178-182
subword parallelism（子字并行），222-223
subword parallelism and matrix multiply（子字并行和矩阵乘法），225-228
Arithmetic instructions（算术运算指令），见 Instructions

desktop RISC（台式精简指令集计算机），E-11
embedded RISC（嵌入式精简指令集计算机），E-14
logical（逻辑），251
MIPS（每秒百万条指令），A-51-57
operands（操作数），66-73
Arithmetic intensity（算术运算强度），541
Arithmetic logic unit（ALU，算术逻辑运算单元），见 ALU control; Control units
1-bit（1 比特），B-26-29
32-bit（32 比特），B-29-38
before forwarding（在转发之前），309
branch datapath（分支数据通路），254
hardware（硬件），180
memory-reference instruction use（存储器相关指令的使用），245
for register values（寄存器值），252
R-format operations（寄存器型操作），253
signed-immediate input（有符号立即数输入），312
ARM Cortex-A8，（ARM Cortex-A8 芯片），244，345-346
address translation for（地址转换），471
caches in（高速缓存），472
data cache miss rates for（数据 cache 缺失率），474
memory hierarchies of（存储器层次体系），471-475
performance of（性能），473-475
specification（规格），345
TLB hardware for（快表硬件），471
ARM instructions（ARM 指令），145-147
12-bit immediate field（12 位立即数字段），148
addressing modes（寻址方式），145
block loads and stores（块加载和存储），149
brief history（简史），OL2.21-5
calculations（运算），145-146
compare and conditional branch（比较和条件转移），147-148
condition field（条件字段），324
data transfer（数据传送），146
features（特性），148-149
formats（格式），148
logical（逻辑），149
MIPS similarities（类 MIPS），146
register-register（寄存器-寄存器型），146
unique（唯一的），E-36-37
ARMv7（ARMv7），62
ARMv8（ARMv8），158-159
ARPANET（阿帕网络），OL1.12-10
Arrays（数组），415
logic elements（逻辑部件），B-18-19
multiple dimension（多维），218
pointers versus（指针），141-145
procedures for setting to zero（置零程序），142
ASCII
binary numbers versus（二进制数），107
character representation（字符表示），106
defined（定义），106
symbols（标志），109
Assembler directives（汇编指令），A-5
Assemblers（汇编器），124-126，A-10-17
conditional code assembly（条件语句汇编代码），A-17
defined（定义），14，A-4
function（函数），125，A-10
macros（宏命令），A-4，A-15-17
microcode（微程序），D-30
number acceptance（数据接收），125
object file（目标文件），125
pseudoinstructions（伪指令），A-17
relocation information（重定位信息），A-13，A-14
speed（速率），A-13
symbol table（符号表），A-12
Assembly language（汇编语言），15
defined（定义），14，123
drawbacks（缺陷），A-9-10
floating-point（浮点型），212
high-level languages versus（高级语言），A-12
illustrated（配有插图），15
MIPS（每秒百万条指令），64，84，A-45-80
production of（生产），A-8-9
programs（程序集），123
translating into machine language（翻译为机器语言），84
when to use（何时使用），A-7-9
Asserted signals（声明信号），250，B-4
Associativity（相联度）
in caches（在高速缓存中），405
degree increasing（程度增加），404，455
increasing（增加），409
set tag sizeversus（设置标识位大小），409
Atomic compare and swap（原子操作比较与调换），123
Atomic exchange（原子交换），121

Atomic fetch-and-increment（原子预取与增量），123
Atomic memory operation（原子存储器操作），C-21
Attribute interpolation（修改属性），C-43-44
Automobiles computer application in（汽车计算机应用），4
Average memory access time（AMAT，平均访存时间），402
  calculating（计算），403

B

Backpatching（反向修补），A-13
Bandwidth（带宽），30-31
  bisection（二分法），532
  external to DRAM（在动态随机存储器之外的），398
  memory（主存），380-381，398
  network（网络），535
Barrier synchronization（障碍同步），C-18
  defined（定义），C-20
  for thread communication（用于线程通信），C-34
Base addressing（基地址），69，116
Base registers（基址寄存器），69
Basic block（基本块），93
Benchmarks（基准测试程序），538-540
  defined（定义），46
  Linpack（线性系统软件包），538，OL3.11-4
  multicores（多核），522-529
  multiprocessor（多处理器），538-540
  NAS parallel（NAS 并行），540
  parallel（并行），539
  PARSEC suite（PARSEC 程序集），540
  SPECCPU（SPEC CPU），46-48
  SPEC power（SPEC 功耗），48-49
  SPECrate（SPEC 分值），538-539
  Stream（流），548
beq（Branch On Equal，相等则转移），64
bge（Branch Greater Than or Equal，大于等于则转移），125
bgt（Branch Greater Than，大于则转移），125
Biased notation（移码表示法），79，20
Big-endian byte order（大端法排列字节次序），70，A-43
Binary numbers（二进制数），81-82
ASCII versus（ASCII 码），107
  conversion to decimal numbers（转换为十进制数），76
  defined（定义），73
Bisection bandwidth（对分带宽），532
Bit maps（位图），
  defined（定义），18，73
  goal（目标），18
  storing（存储），18
Bit-Interleaved Parity（RAID 3，比特交错奇偶校验），OL5.11-5
Bits（二进制数字），
  ALUOp（ALU 操作码），260，261
  defined（定义），14
  dirty（脏数据），437
  guard（保护），220
  patterns（模式），220-221
  reference（引用），435
  rounding（舍入），220
  sign（符号），75
  state（状态），D-8
  sticky（粘后），220
  valid（有效），383
ble（Branch Less Than or Equal，小于等于则转移），125
Blocking assignment（阻塞赋值），B-24
Blocking factor（块因子），414
Block-Interleaved Parity（RAID 4，块交叉奇偶方式），OL5.11-5-5.11-6
Blocks（块）
  combinational（组合的），B-4
  defined（定义），376
  finding（查找），456
  flexible placement（灵活放置），402-404
  least recently used（LRU，最近最少使用），409
  loads/stores（取/存），149
  locating in cache（位于高速缓存内），407-408
  miss rate and（失效率），391
  multiword mapping addresses to（多字地址映射），390
  placement locations（存储位置），455-456
  placement strategies（存储策略），404
  replacement selection（替换选择），409
  replacement strategies（替换策略），457
  spatial locality exploitation（空间局部性开发），391
  state（状态），B-4
  valid data（有效数据），386

blt（Branch Less Than，小于则转移），125
bne（Branch On Not Equal，不相等则转移），64
Bonding（绑定），28
Boolean algebra（布尔代数），B-6
Bounds check shortcut（边界检查的简便方法），95
Branch datapath（分支数据通路），
  ALU（算术逻辑单元），254
  operations（操作码），254
Branch delay slots（分支延迟槽），
  defined（定义），322
  scheduling（调度），323
Branch equal（相等则转移），318
Branch instructions（转移指令），A-59-63
  jump instruction versus（跳转指令），270
  list of（列表），A-60-63
  pipeline impact（流水线的影响），317
Branch not taken（转移无效），
  assumption（假定），318
  defined（定义），254
Branch prediction（分支预测），
  as control hazard solution（控制冒险解决方案），284
  buffers（缓冲区），321，322
  defined（定义），283
  dynamic（动态的），284，321-323
  static（静态的），335
Branch predictors（分支预测器），
  accuracy（准确性），322
  correlation（相关），324
  information from（信息来源），324
  tournament（竞赛），324
Branch taken（转移成功），
  cost reduction（成本降低），318
  defined（定义），254
Branch target（转移目标），
  addresses（地址），254
  buffers（缓冲区），324
Branches（转移），见 Conditional branches
  addressing in（地址输入），113-116
  compiler creation（编译器创建），91
  condition（条件），255
  decision moving up（决策上移），318
delayed（延迟），96，255，284，318-319，322，324
  ending（结束），93
  execution in ID stage（在指令译码阶段执行），319
  pipelined（流水线），318
  target address（目标地址），318
  unconditional（无条件），91
Branch-on-equal instruction（相等则转移指令），268
Bubble Sort（冒泡排序），140
Bubbles（气泡），314
Bus-based coherent multiprocessors（基于总线的一致性多处理器），OL6.15-7
Buses（总线），B-19
Bytes（字节），
  addressing（寻址），70
  order（命令），70，A-43

C

C. mmp，OL6.15-4
C language（C 语言）
  assignment compiling into MIPS（任务编译为 MIPS 指令），65-66
  compiling（编译），145，OL2.15-2-2.15-3
  compiling assignment with registers（使用寄存器编译赋值语句），67-68
  compiling while loops in（将 while 循环编译为），92
  sort algorithms（排序算法），141
  translation hierarchy（翻译层次），124
  translation to MIPS assembly language（翻译为 MIPS 汇编语言），65
  variables（变量），102
C++ language（C++语言），OL2.15-27，OL2.21-8
Cache blocking and matrixmultiply（cache 阻塞和矩阵相乘），475-476
Cache coherence（cache 一致性），466-470
  coherence（一致性），466
  consistency（相容性），466
  enforcement schemes（强制执行），467-468
  implement techniques（实现技术），OL5.12-11-5.12-12
  migration（迁移），467
  problem（问题），466，467，470
  protocol example（协议示例），OL5.12-12-5.12-16
  protocols（协议），468
  replication（复制），468
  snooping protocol（监听协议），468-469
  snoopy（监听），OL5.12-17
  statediagram（状态图），OL5.12-16
Cache coherency protocol（cache 一致性协议），

OL5. 12-12-5. 12-16
finite-state transition diagram（有限状态转换图），OL5. 12-15
functioning（功能），OL5. 12-14
mechanism（机制），OL5. 12-14
state diagram（状态图），OL5. 12-16
states（状态），OL5. 12-13
write-back cache（写回式 cache），OL5. 12-15
Cache controllers（cache 控制器），470
coherent cache implementation techniques（一致性 cache 实现技术），OL5. 12-11-5. 12-12
implementing（实现），OL5. 12-2
snoopy cache coherence（监听 cache 一致性），OL5. 12-17
System Verilog（System Verilog 语言），OL5. 12-2
Cache hits（cache 命中），443
Cache misses（cache 缺失）
block replacement on（块替换），457
capacity（容量），459
compulsory（强制的），459
conflict（冲突），459
defined（定义），392
direct-mapped cache（直接映射 cache），404
fully associative cache（全相联 cache），406
handling（处理），392-393
memory-stall clock cycles（存储器停顿时钟周期），399
reducing with flexible block placement（通过灵活的块替换降低），402-404
set-associative cache（组相联 cache），405
steps（步骤），393
in write-through cache（在写直达式 cache 中），393
Cache performance（cache 性能），398-417
calculating（计算），400
hit time and（命中时间），401-402
impact on processor performance（对处理器性能的影响），400
Cache-aware instructions（cache 感知指令），482
Caches，见 Blocks，383-398
accessing，386-389
in ARM cortex-A8，（在 ARM cortex-A8 上），472
associativity in（结合性），405-406
bits in（位数），390
bits needed for（需要的位数），390
contents illustration（内容说明），387
defined（定义），21，383-384
direct-mapped（直接映射），384，385，390，402
empty（空），386-387
FSM for controlling（控制有限状态机），461-462
fully associative（全相联映射），403
GPU（图形处理器），C-38
inconsistent（不一致的），393
index（索引），388
in Intel Core i7，（英特尔睿睿 i7 处理器），472
Intrinsity FastMATH example（内置 FastMATH 处理器），395-398
locating blocks in（块定位），407-408
locations（定位），385
multilevel（多级），398，410
nonblocking（非阻塞），472
physically addressed（物理地址），443
physically indexed（物理索引），443
physically tagged（物理标记），443
primary（主要的），410，417
secondary（次要的），410，417
set-associative（组相联映射），403
simulating（模拟），478
size（尺寸），389
split（分裂），397
summary（摘要），397-398
tag field（标志字段），388
tags（标志），OL5. 12-3，OL5. 12-11
virtual memory and TLB integration（虚拟存储器的快表集成），440-441
virtually addressed（虚拟地址），443
virtually indexed（虚拟索引），443
virtually tagged（虚拟标志），443
write-back（写回），394，395，458
write-through（写直达），393，395，457
writes（写操作），393-395
Callee（被调用者），98，99
Callee-saved register（保存被调用者的寄存器），A-23
Caller（调用者），98
Caller-saved register（保存调用者的寄存器），A-23
Capabilities（能力），OL5. 17-8
Capacity misses（容量缺失），459
Carry lookahead（先行进位），B-38-47
4-bit ALUs using（4 位 ALU 的使用），B-45
adder（加法器），B-39
fast with first level of abstraction（用第一级抽象

快速进位), B-39-40
fast with "infinite" hardware (使用"无限"硬件的快速进位), B-38-39
fast with second level of abstraction (用第二级抽象快速进位), B-40-46
plumbing analogy (管道类比), B-42, B-43
ripple carry speed versus (行波进位速率), B-46
summary (摘要), B-46-47
Carry save adders (进位存储加法器), 188
Cause register (原因存储器),
defined (定义), 327
fields (字段), A-34, A-35
OLC 6600 (连续计算机), OL1.12-7, OL4.16-3
Cell phones (移动电话), 7
Central processor unit (CPU) (中央处理器), 见 Processors
classic performance equation (经典的性能方程), 36-40
coprocessor0 (协处理器0), A-33-34
defined (定义), 19
execution time (执行时间), 32, 33-34
performance (性能), 33-35
system time (系统时间), 32
time (时间), 399
time measurements (时间测定), 33-34
user time (用户时间), 32
Cg pixel shader program (计算机绘图像素着色程序), C-15-17
Characters (字符),
ASCII representation (ASCII码表示法), 106
in Java (在Java中), 109-111
Chips (芯片), 19, 25, 26
manufacturing process (制造工艺), 26
Classes (级别),
defined (定义), OL2.15-15
packages (包), OL2.15-21
Clock cycles (时钟周期),
defined (定义), 33
memory-stall (存储阻塞), 399
number of registers and (寄存器数量), 67
worst-case delay and (最坏情况下延迟), 272
Clock cycles per instruction (CPI) (每条指令执行的时钟周期数), 35, 282
one level of caching (一级 cache), 410
two levels of caching (二级 cache), 410
Clock rate (时钟频率),
defined (定义), 33
frequency switched as function of (频率转换功能), 41
power and (功率), 40
Clocking methodology (时钟同步法), 249-251, B-48
edge-triggered (边沿触发), 249, B-48, B-73
level-sensitive (层次敏感), B-74, B-75-76
for predictability (可预测性), 249
Clocks (时钟), B-48-50
edge (边沿), B-48, B-50
in edge-triggered design (边沿触发设计), B-73
skew (偏斜), B-74
specification (说明书), B-57
synchronous system (同步系统), B-48-49
Cloud computing (云计算), 533
defined (定义), 7
Cluster networking (集群网络), 537-538, OL6.9-12
Clusters (集群), OL6.15-8-6.15-9
defined (定义), 30, 500, OL6.15-8
isolation (隔离), 530
organization (组织), 499
scientific computing on (科学计算), OL6.15-8
Cm *, OL6.15-4
CMOS (complementary metal oxide semiconductor) (互补金属氧化物半导体), 41
Course-grained multithreading (粗粒度多线程), 514
Cobol (Cobol 语言), OL2.21-7
Code generation (代码生成), OL2.15-13
Code motion (代码移动), OL2.15-7
Cold-start miss (冷启动失效), 459
Collision misses (冲突失效), 459
Column major order (按列顺序), 413
Combinational blocks (组合块), B-4
Combinational control units (组合控制单元), D-4-8
Combinational elements (组合元素), 248
Combinational logic (组合逻辑), 249, B-3, B-9-20
arrays (数组), B-18-19
decoders (译码器), B-9
defined (定义), B-5
don't cares (无关项), B-17-18
multiplexors (多路选择器), B-10
ROMs (只读存储器), B-14-16
two-level (二级), B-11-14
Verilog (Verilog 语言), B-23-26

Commercial computer development（商用计算机开发），OL1.12-4-1.12-10
Commit units（提交单元）
buffer（缓冲区），339-340
defined（定义），339-340
in update control（更新控制），343
Common case fast（加速大概率事件），11
Common subexpression elimination（公共子表达式消除法），OL2.15-6
Communication（通信），23-24
overhead reducing（开销减少），44-45
thread（线程），C-34
Compact code（压缩码），OL2.21-4
Comparison instructions（比较指令），A-57-59
floating-point（浮点型），A-74-75
list of（列表），A-57-59
Comparisons（比较），93
constant operands in（常量操作数），93
signed versus unsigned（有符号数与无符号数），94-95
Compilers（编译器），123-124
branch creation（创建分支），92
brief history（简史），OL2.21-9
conservative（保守的），OL2.15-6
defined（定义），14
front end（前端），OL2.15-3
function（函数），14，123-124，A-5-6
high-level optimizations（高层次优化），OL2.15-4
ILP exploitation（指令级并行开发），OL4.16-5
Just In Time（JIT）（准时），132
machine language production（机器语言生成），A-8-9，A-10
optimization（优化），141，OL2.21-9
speculation（推测），333-334
structure（结构），OL2.15-2
Compiling（编译）
C assignment statements（C 赋值语句），65-66
C language（C 语言），92-93，145，OL2.15-2-2.15-3
floating-point programs（浮点运算程序），214-217
if-then-else（if-then-else 语句），91
in Java（Java 中）OL2.15-19
procedures（过程），98，101-102
recursive procedures（递归过程），101-102
while loops（while 循环），92-93
Compressed sparse row（CSR）matrix（压缩稀疏行矩阵），C-55，C-56
Compulsory misses（强制性失效），459
Computer architects（计算机设计），11-12
abstraction to simplify design（简化设计抽象），11
common case fast（加速大概率事件），11
dependability via redundancy（通过冗余增加可靠性），12
hierarchy of memories（存储体系层次），12
Moore's law（摩尔定律），11
parallelism（并行），12
pipelining（流水线），12
prediction（预测），12
Computers（计算机），
application classes（应用类别），5-6
applications（应用程序），4
arithmetic for（算法），176-236
characteristics（特性），OL1.12-12
commercial development（商业性开发），OL1.12-4-1.12-10
component organization（组成结构），17
components（组件），17，177
design measure（设计尺度），53
desktop（台式机），5
embedded（嵌入式），5，A-7
first（第一），OL1.12-2-1.12-4
in information revolution（信息革命），4
instruction representation（指令表示），80-87
performance measurement（性能测定），OL1.12-10
PostPC Era（后计算机时代），6-7
principles（法则），86
servers（服务器），5
Condition field（条件字段），324
Conditional branches（条件分支），
ARM（Advanced RISC Machines 公司），147-148
changing program counter with（更改程序计数器），324
compiling if-then-else into（将 if-then-else 语句编译为），91
defined（定义），90
desktop RISC（精简指令集台式机），E-16
embedded RISC（嵌入式精简指令集），E-16
implementation（实现），96
in loops（in 循环），115
PA-RISC（HP 公司 PA-RISC 芯片），E-34，E-35
PC-relative addressing（PC 相对寻址），114

RISC（精简指令集计算机），E-10-16
SPARC（可扩充处理器架构），E-10-12
Conditional move instructions（条件移动指令），324
Conflict misses（冲突失效），459
Constant memory（常量存储器），C-40
Constant operands（常数操作数），72-73
in comparisons（与……比较），93
frequent occurrence（频发），72
Constant-manipulating instructions（常量操作指令），A-57
Content Addressable Memory（CAM，按内容访问存储器），408
Context switch（上下文切换），446
Control（控制），
ALU（运算器），259-261
challenge（挑战），325-326
finishing（完成），269-270
forwarding（转发/旁路），307
FSM（有限状态机），D-8-21
implementation optimizing（实现优化），D-27-28
for jump instruction（跳转指令），270
mapping to hardware（映射到硬件），D-2-32
memory（存储器），D-26
organizing to reduce logic（组织简化逻辑），D-31-32
pipelined（流水线），300-303
Control flow graphs（控制流图），OL2.15-9-2.15-10
illustrated examples（举例说明），OL2.15-9, OL2.15-10
Control functions（控制功能），
ALU mapping to gates（算术逻辑单元映射到逻辑门），D-4-7
defining（定义），264
PLA implementation（可编程逻辑阵列实现），D-7，D-20-21
ROM encoding（只读存储器编码），D-18-19
for single-cycle implementation（单周期实现），269
Control hazards（控制冒险），281-282，316-325
branch delay reduction（减少分支延迟），318-319
branch not taken assumption（假定不跳转），318
branch prediction as solution（使用分支预测作为解决方案），284
delayed decision approach（延迟决定方法），284
dynamic branch prediction（动态分支预测），321-323
logic implementation in Verilog（Verilog 中的逻辑实现），OL4.13-8
pipeline stalls as solution（流水线停顿解决方案），282
pipeline summary（流水线小结），324
simplicity（简单），317
solutions（解决方案），282
static multiple-issue processors（静态多发射处理器），335-336
Control lines（控制线），
asserted（有效），264
in datapath（在数据通路中），263
execution/address calculation（执行/地址计算），300
final three stages（最后三个阶段），305
instruction decode/register file read（指令译码/读寄存器文件），300
instruction fetch（取指令），300
memory access（存储器访问），302
setting of（一组），264
values（数值），300
write-back（写回），302
Control signals（控制信号）
ALUOp（ALU 操作），263
defined（定义），250
effect of（影响），264
multi-bit（多比特），264
pipelined datapaths with（流水线数据通路），300-303
truth tables（真值表），D-14
Control units（控制单元），见 Arithmetic logic unit (ALU)，247
address select logic（地址选择逻辑），D-24，D-25
combinational implementing（组合的实现），D-4-8
with explicit counter（使用显式计数器），D-23
illustrated（举例说明），265
logic equations（逻辑方程），D-11
main designing（主要部分设计），261-264
as microcode（作为微程序），D-28
MIPS（每秒百万条指令），D-10
next-state outputs（下一状态输出），D-10，D-12-13
output（输出），259-261，D-10
Conversion instructions（转换指令），A-75-76
Cooperative thread arrays（CTA，合作线程阵列），C-30
Coprocessors（协处理器），A-33-34

defined（定义），218
move instructions（传送指令），A-71-72
Core MIPS instruction set（核心 MIPS 指令集），见 MIPS，236，
abstract view（抽象图），246
desktop RISC（精简指令集台式机），E-9-11
implementation（实现），244-248
implementation illustration（实现举例），247
overview（综述），245
subset（子集），244
Cores（核），
defined（定义），43
number per chip（每片数量），43
Correlation predictor（相关预测器），324
Cosmic Cube（宇宙立方），OL6.15-7
Count register（计数寄存器），A-34
CPU（中央处理器），9
Cray computers（克雷（公司）计算机），OL3.11-5-3.11-6
Critical word first（关键字优先），392
Crossbar networks（交叉开关网络），535
CTSS（CompatibleTime-Sharing System，兼容分时系统），OL5.18-9
CUDA programming environment（CUDA 程序设计环境），523，C-5
barrier synchronization（屏蔽同步），C-18，C-34
development（开发），C-17，C-18
hierarchy of thread groups（线程群组层次），C-18
kernels（核心程序），C-19，C-24
key abstractions（关键抽象），C-18
paradigm（范例），C-19-23
parallel plus-scan template（并行 plus-scan 软件模板），C-61
per-block shared memory（以块为共享单位的存储器），C-58
plus-reduction implementation（减少时钟节拍的实现），C-63
programs（程序集），C-6，C-24
scalable parallel programming with（可扩展的并行程序设计），C-17-23
shared memories（共享存储器），C-18
threads（线程），C-36
Cyclic redundancy check（循环冗余校验），423
Cylinder（柱面），381

D

D flip-flops（D 触发器），B-51，B-53
D latches（D 型锁存器），B-51，B-52
Data bits（数据位），421
Data flow analysis（数据流分析），OL2.15-11
Data hazards（数据冒险），见 Hazards，278，303-316
forwarding（转发/旁路），278，303-316
load-use（装载－使用），280，318
stalls and（停顿），313-316
Data layout directives（数据布局规则），A-14
Data movement instructions（数据传送指令），A-70-73
Data parallel problem decomposition（数据并行问题分解），C-17，C-18
Data race（数据竞争），121
Data segment（数据段），A-13
Data selectors（数据选择器），246
Data transfer instructions（数据传送指令），见 Instructions
defined（定义），68
load（装载），68
offset（偏移量），69
store（存储），71
Datacenters（数据处理中心），7
Data-level parallelism（数据级并行），508
Datapath elements（数据路径部件），
defined（定义），251
sharing（共享），256
Datapaths（数据路径）
branch（分支），254
building（生成），251-259
control signal truth tables（控制信号真值表），D-14
control unit（控制单元），265
defined（定义），19
design（设计），251
exception handling（异常处理），329
for fetching instructions（取指令），253
for hazard resolution via forwarding（通过旁路解决冒险），311
for jump instruction（跳转指令），270
for memory instructions（访存指令），256
for MIPS architecture（MIPS 体系结构），257
in operation for branch-on-equalinstruction（相等则转移指令执行），268
in operation for load instruction（装入指令执行），267

in operation for R-type instruction（寄存器型指令执行），266
operation of（……的操作），264-269
pipelined（流水的），286-303
for R-type instructions（寄存器型指令），256，264-265
single creating（单一创建），256
single-cycle（单周期），283
static two-issue（静态双发射），336
Deasserted signals（无效信号），250，B-4
Debugging information（调试信息），A-13
DEC PDP-8，OL2.21-3（DEC PDP-8 机器），
Decimal numbers（十进制数），
binary number conversion to（二进制数转换），76
defined（定义），73
Decision-making instructions（决策指令），90-96
Decoders（译码器），B-9
two-level（二级），B-65
Decoding machine language（机器语言译码），118-120
Defect（缺陷），26
Delayed branches（延迟转移），见 Branches，96
as control hazard solution（作为分支冒险的解决方案），284
defined（定义），255
embedded RISCs and（嵌入式精简指令集计算机），E-23
for five-stage pipelines（五级流水线），26，323-324
reducing（裁剪），318-319
scheduling limitations（调度限制），323
Delayed decision（决策延迟），284
DeMorgan's theorems（德摩根定理），B-11
Denormalized numbers（非规格化数），222
Dependability via redundancy（通过冗余提高可靠性），12
Dependable memory hierarchy（可靠存储器体系层次），418-423
failure defining（失效定义），418
Dependences（相关），
between pipeline registers（在流水线寄存器之间），308
between pipeline registers and ALU inputs（在流水线寄存器和 ALU 输入之间），308
bubble insertion and（气泡插入），314
detection（检测），306-308
name（命名），338
sequence（时序），304
Design（设计），
compromises and（折衷），161
datapath（数据通路），251
digital（数字），354
logic（逻辑），248-251，B-1-79
main control unit（主控制单元），261-264
memory hierarchy challenges（存储器体系层次挑战），460
pipelining instruction sets（流水化的指令集），277
Desktop and server RISCs（精简指令集台式机和服务器），见 Reduced instruction set computer（RISC）architectures
addressing modes（寻址模式），E-6
architecture summary（体系结构概述），E-4
arithmetic/logical instructions（算术/逻辑指令），E-11
conditional branches（条件转移），E-16
constant extension summary（常数扩展小结），E-9
control instructions（控制指令），E-11
conventions equivalent to MIPS core（约定为与 MIPS 核一致），E-12
data transfer instructions（数据传输指令），E-10
features added to（将特征添加到），F-45
floating-point instructions（浮点型指令），E-12
instruction formats（指令格式），E-7
multimedia extensions（多媒体扩展），E-16-18
multimedia support（多媒体支持），E-18
types of（类型），E-3
Desktop computers defined（台式计算机定义），5
Device driver（设备驱动程序），OL6.9-5
DGEMM（Double precision General Matrix Multiply）（双精度通用矩阵相乘），225，352，413，553
cache blocked version of（cache 的分块版本），415
optimized C version of（优化的 C 版本），226，227，476
performance（性能），354，416
Dicing（切割），27
Dies（模具），26，26-27
Digital design pipeline（数字设计流水线），354
Digital signal-processing（DSP）extensions（数字信号处理扩展），E-19
DIMMs（dual inline memory modules，双列直插存储器模块），OL5.17-5
Direct Data IO（DDIO，直接数据传输），OL6.9-6
Direct memory access（DMA，直接存储器访问），OL6.9-4

Direct3D（直接），C-13
Direct-mapped caches（直接映射 cache），见 Caches
address portions，（地址部分）407
choice of（……的选择），456
defined（定义），384，402
illustrated（举例说明），385
memory block location（存储器块定位），403
misses（缺失），405
single comparator（一位比较），407
total number of bits（总比特数），390
Dirty bit（脏位），437
Dirty pages（脏页），437
Disk memory（磁盘存储器），381-383
Displacement addressing（偏移地址），116
Distributed Block-Interleaved Parity（RAID 5，分布式块交叉校验），OL5.11-6
div（Divide）（除法），A-52
div.d（FP Divide Double，双精度浮点除法），A-76
div.s（FP Divide Single，单精度浮点除法），A-76
Divide algorithm（除法算法），190
Dividend（被除数），189
Division（除法），189-195
algorithm（算法），191
dividend（被除数），189
divisor（除数），189
Divisor（除数），189
divu（Divide Unsigned，无符号数除法），见 Arithmetic，A-52
faster（更快的），194
floating-point（浮点型），211，A-76
hardware（硬件），189-192
hardware improved version（硬件改进版），192
instructions（指令），A-52-53
in MIPS（在 MIPS 指令集中），194
operands（操作数），189
quotient（商），189
remainder（余数），189
signed（有符号数），192-194
SRT（一种除法算法的名称），194
Don't cares（无关的），B-17-18
example（实例），B-17-18
term（术语），261
Double data rate（DDR，双倍数据速率），379
Double Data Rate RAMs（DDRRAM，双倍数据速率 RAM），379-380，B-65
Double precision（双精度），见 Single precision Defined（定义），198
FMA（混合乘加单元），C-45-46
GPU（图形处理器），C-45-46，C-74
Representation（表达），201
Double words（双字），152
Dual inline memory module（DIMM，双列直插存储器模块），381
Dynamic branch prediction（动态分支预测），见 Control hazards，321-323
branch prediction buffer（分支预测缓冲器），321
loops and（循环），321-323
Dynamic hardware predictors（动态硬件预测器），284
Dynamic multiple-issue processors（动态多发射处理器），见 Multiple issue，333，339-341
pipeline scheduling（流水线调度），339-341
superscalar（超标量），339
Dynamic pipeline scheduling（动态流水线调度），339-341
commit unit（提交单元），339-340
concept（概念），339-340
hardware-based speculation（基于硬件的推断），341
primary units（主单元），340
reorder buffer（重排序缓冲器），343
reservation station（保留站），339-340
Dynamic random access memory（DRAM，动态随机访问存储器），378，379-381，B-63-65
bandwidth external to（外部带宽），398
cost（代价），23
defined（定义），19，B-63
DIMM（双列直插式存储模块），OL5.17-5
Double Date Rate（DDR，双倍数据速率），379-380
early board（早期的标志），OL5.17-4
GPU（图形处理器），C-37-38
growth of capacity（容量的增长），25
history（历史记录），OL5.17-2
internal organization of（内部组织），380
pass transistor（传输晶体管），B-63
SIMM（单点插存储器模块），OL5.17-5，OL5.17-6
single-transistor（单晶体管），B-64
size（尺寸），398
speed（速率），23
synchronous（SDRAM，同步的（同步动态随机存储器）），379-380，B-60，B-65
two-level decoder（两级译码器），B-65
Dynamically linked libraries（DLL，动态链接库），129-131

Defined（定义），129
lazy procedure linkageversion（惰性过程链接版本），130

E

Early restart（提前重启），392
Edge-triggered clocking methodology（边沿触发时钟同步法），249，250，B-48，B-73
advantages（优势），B-49
clocks（时钟），B-72
drawbacks（劣势），B-74
illustrated（举例说明），B-50
rising edge/falling edge（上升沿/下降沿），B-48
EDSAC（Electronic Delay Storage Automatic Calculator）（电子延时存储自动计算器），OL1.12-3，OL5.17-2
Eispack（计算矩阵特征值与特征向量的软件库由Fortran语言编写），OL3.11-4
Electrically erasable programmable read-only memory（EEPROM，电可擦除可编程只读存储器），381
Elements（元素）
combinational（组合的），248
datapath（数据通路），251，256
memory（存储器），B-50-58
state（状态），248，250，252，B-48，B-50
Embedded computers（嵌入式计算机），5
application requirements（应用需求），6
defined（定义），A-7
design（设计），5
growth（增长），OL1.12-12-1.12-13
Embedded Microprocessor Benchmark Consortium（EEMBC，嵌入式微处理器基准测试联盟），OL1.12-12
Embedded RISCs（嵌入式精简指令系统计算机），见Reduced instruction set computer（RISC）
architectures（体系结构）
addressing modes（寻址方式），E-6
architecture summary（体系结构概述），E-4
arithmetic/logical instructions（算术/逻辑指令），E-14
conditional branches（条件转移），E-16
constant extension summary（常量），E-9
control instructions（控制指令），E-15
data transfer instructions（数据传输指令），E-13
delayed branch and（延迟转移），E-23
DSP extensions（数字信号处理扩展），E-19
general purpose registers（通用寄存器），E-5
instruction conventions（指令约定），E-15
instruction formats（指令格式），E-8
multiply-accumulate approaches（乘累加方法），E-19
types of（类型），E-4
Encoding（编码），
defined（定义），D-31
floating-point instruction（浮点指令），213
MIPS instruction（MIPS指令集），83，119，A-49
ROM control function（ROM控制函数），D-18-19
ROM logic function（ROM逻辑函数），B-15
x86 instruction（x86指令），155-156
ENIAC（Electronic Numerical Integrator and Calculator，电子数字积分计算机），OL1.12-2，OL1.12-3，OL5.17-2
EPIC（Explicitly Parallel Instruction Code，显式并行指令代码），OL4.16-5
Error correction（纠错），D-65-67
Error Detecting and Correcting Code（RAID 2，错误检测及校正码），OL5.11-5
Error detection（错误检测），B-66
Error detection code（错误检测码），420
Ethernet（以太网），23
EX stage（执行阶段），
load instructions（加载指令），292
overflow exception detection（溢出异常检测），328
store instructions（存储指令），294
Exabyte（艾字节（2的16次方）），6
Exception enable（异常使能），447
Exception handlers（异常处理程序），A-36-38
defined（定义），A-35
return from（从……返回），A-38
Exception program counters（EPC，异常程序计数器），326
address capture（地址头），331
copying（复制），181
defined（定义），181，327
in restart determination（重启决策），326-327
transferring（转移），182
Exceptions（异常），325-332，A-33-38
association（关联），331-332
datapath with controls for handling（带有控制处理的数据路径），329
defined（定义），180，326

detecting（检测），326
event types and（事件类型），326
imprecise（不精确的），331-332
instructions（指令），A-80
interrupts versus（与中断相对），325-326
in MIPS architecture（在 MIPS 体系结构中），326-327
overflow（溢出），329
PC（程序计数器），445，446-447
pipelined computer example（流水线计算机实例），328
in pipelined implementation（在流水线实现中），327-332
precise（精确的），332
reasons for（原因），326-327
result due to overflow in add instruction（由于加法指令溢出所产生的结果），330
saving/restoring stage on（存储/取出阶段），450
Exclusive OR（XOR）instructions（异或指令），A-57
Executable files（可执行文件），A-4
defined（定义），126
linker production（链接程序生成），A-19
Execute or address calculation stage（执行或地址计算阶段），292
Execute/address calculation
control line（执行或地址计算控制线），300
load instruction（加载指令），292
store instruction（存储指令），292
Execution time（执行时间），
as valid performance measure（作为有效的性能指标），51
CPU（中央处理器），32，33-34
pipelining and（流水线技术），286
Explicit counters（显式计数器），D-23，D-26
Exponents（范例），197-198
External labels（外部标记），A-10

F

Facilities（设备），A-14-17
Failures synchronizer（故障同步器），B-77
Fallacies（谬误），见 Pitfalls add immediate unsigned（加无符号立即数），227
Amdahl's law（阿姆达尔定律），556
arithmetic（算法），229-232
assembly language for performance（汇编语言性能），159-160
commercial binary compatibility importance（商用计算机二进制兼容的重要性），160
defineded（定义），49
GPUs（图形处理器），C-72-74，C-75
low utilization uses little power（低利用率低能耗），50
peak performance（峰值性能），556
pipelining（流水线技术），355-356
powerful instructions mean higher
performance（强大的指令意味着更高的性能），159
right shift（右移），229
False sharing（伪共享），469
Fast carry（快速进位），
with "infinite" hardware（带有"无限的"硬件），B-38-39
with first level of abstraction（第一级抽象），B-39-40
with second level of abstraction（第二级抽象），B-40-46
Fast Fourier Transforms（FFT，快速傅里叶变换），C-53
Fault avoidance（故障避免），419
Fault forecasting（故障预测），419
Fault tolerance（容错），419
Fermi architecture.（费尔米体系结构），523，552
Field programmable devices（FPD，现场可编程器件），B-78
Field programmable gate arrays（FPGA，现场可编程门阵列），B-78
Fields（字段）
Cause register（原因寄存器），A-34，A35
defined（定义），82
format（格式），D-31
MIPS（每秒百万条指令），82-83
names（命名），82
Status register（状态寄存器），A-34，A-35
Files Register（文件寄存器），252，257，B-50，B-54-56
Fine-grained multithreading（细粒度多线程），514
Finite-state machines（FSM，有限状态机），451-466，B-67-72
control（控制），D-8-22
controllers（控制器），464
for multicycle control（多周期控制），D-9
for simple cache controller（简单的高速缓存控制

器），464-466
implementation（实现），463，B-70
Mealy（米利机），463
Moore（摩尔（音译）），463
next-state function（下一状态函数），463，B-67
output function（输出函数），B-67，B-69
state assignment（状态指定），B-70
state register implementation（状态寄存器实现），B-71
style of（类型），463
synchronous（同步的），B-67
SystemVerilog（系统 Verilog），OL5.12-7
traffic light example（交通灯实例），B-68-70
Flash memory（闪存），381
characteristics（特征），23
defined（定义），23
Flat address space（单层地址空间），479
Flip-flops（触发器），
D flip-flops（D 触发器），B-51，B-53
defined（定义），B-51
Floating point（浮点），196-222，224
assembly language（汇编语言），212
backward step（后退一步），OL3.11-4-3.11-5
binary to decimal conversion（二-十进制转换），212
branch（分支），211
challenges（挑战），232-233
diversity versus portability（多样性与便携性），OL3.11-3-3.11-4
division（除法），211
first dispute（第一个争议），OL3.11-2-3.11-3
form（形态），197
fused multiply add（乘法加法混合），220
guard digits（保护位），218-219
history（历史），OL3.11-3
IEEE 754 standard（IEEE 754 标准），198，199
instruction encoding（指令编码），213
intermediate calculations（中间计算器），218
machine language（机器语言），212
MIPS instruction frequency for（MIPS 指令频率），236
MIPS instructions（MIPS 指令），211-213
operands（操作数），212
overflow（溢出），198
packed format（压缩格式），224
precision（精度），230
procedure with two-dimensional matrices（带有二维矩阵的过程），215-217
programs compiling（程序编译），214-217
registers（寄存器），217
representation（表述），197-202
rounding（舍入），218-219
sign and magnitude（符号和量级），197
SSE2 architecture（流处理单指令多数据扩展结构），224-225
subtraction（减法），211
underflow（下溢），198
units（单位），219
in x86（在 x86 中），224
Floating vectors（浮点向量），OL3.11-3
Floating-point addition（浮点数加法），203-206
arithmetic unit block diagram（算术运算器框图），207
binary（二进制），204
illustrated（举例说明），205
instructions（指令），211，A-73-74
steps（步骤），203-204
Floating-point arithmetic（GPU，浮点算法（图形处理器）），C-41-46
basic（基础的），C-42
double precision（双精度），C-45-46，C-74
performance（性能），C-44
specialized（专门的），C-42-44
supported formats（支持的格式），C-42
texture operations（结构操作），C-44
Floating-point instructions（浮点指令），A-73-80
absolute value（绝对值），A-73
addition（加法），A-73-74
comparison（比较），A-74-75
conversion（转换），A-75-76
desktop RISC（台式精简指令系统计算机），E-12
division（除法），A-76
load（加载），A-76-77
move（移动），A-77-78
multiplication（乘法），A-78
negation（逻辑非），A-78-79
SPARC（可扩充处理器架构），E-31
square root（平方根），A-79
store（存储），A-79
subtraction（减法），A-79-80
truncation（截断），A-80
Floating-point multiplication（浮点乘法），206-210

binary（二进制），210-211
illustrated（举例说明），209
instructions（指令），211
significants（有效的），206
steps（步骤），206-210
Flow-sensitive information（流敏感信息），OL2.15-15
Flushing instructions（冲刷指令），318，319
defined（定义），319
exceptions and（异常），331
For loops（for 循环），141，OL2.15-26
inner（内部），OL2.15-24
SIMD and（单指令多数据结构），OL6.15-2
Formal parameters（形式参数），A-16
Format fields（格式字段），D-31
Fortran（Furtran 语言），OL2.21-7
Forward references（向前引用），A-11
Forwarding（传递/旁路），303-316
ALU before（算术逻辑单元），309
control（控制），307
datapath for hazard resolution（冒险决策数据通路），311
defined（定义），278
functioning（功能），306
graphical representation（图解表示法），279
illustrations（插图），OL4.13-26-4.13-26
multiple results and（多重返回值），281
multiplexors（多路旋转装器），310
pipeline registers before（流水线寄存器），309
with two instructions（带有两条指令），278
Verilog implementation（Verilog 程序实现），OL4.13-2-4.13-4
Fractions（分数），197，198
Frame buffer（帧缓冲器），18
Frame pointers（帧指针），103
Front end（前端），OL2.15-3
Fully associative caches（全相联 cache），见 Caches
block replacement strategies（块替换策略），457
choice of（选择），456
defined（定义），403
memory block location（存储器块定位），403
misses（缺失），406
Fully connected networks（全互连网络），535
Function code（功能码），82
Fused-multiply-add（FMA）operation（混合乘加操作），220，C-45-46

## G

Game consoles（游戏控制），C-9
Gates（门），B-3，B-8
AND（与），B-12，D-7
delays（延迟），B-46
mapping ALU control function to（将 ALU 控制功能映射到……），D-4-7
NAND（与非），B-8
NOR（或非），B-8，B-50
Gather-scatter（聚集－分散），511，552
General Purpose GPUs（GPGPU，通用图形处理器），C-5
General-purpose registers（通用寄存器），150
architecture（体系结构），OL2.21-3
embedded RISCs（嵌入式精简指令系统计算机），E-5
Generate（生成）
defined（定义），B-40
example（实例），B-44
super（超级），B-41
Gigabyte（吉字节），6
Global common subexpression elimination（全局公共子表达式消去），OL2.15-6
Global memory（全局存储器），C-21，C-39
Global miss rates（全局缺失率），416
Global optimization（全局优化），OL2.15-5
code（代码），OL2.15-7
implementing（实现），OL2.15-8-2.15-11
Global pointers（全局指针），102
GPU computing（GPU 计算），见 Graphics processing units（GPUs）
defined（定义），C-5
visual applications（可视化应用程序），C-6-7
GPU system architectures（GPU 系统架构），C-7-12
graphics logical pipeline（图形逻辑流水线），C-10
heterogeneous（异构），C-7-9
implications for（实现），C-24
interfaces and drivers（界面和驱动），C-9
unified（统一的），C-10-12
Graph coloring（图着色），OL2.15-12
Graphics displays（图像显示），
computer hardware support（计算机硬件支持），18
LCD（液晶显示屏），18

Graphics logical pipeline（图形逻辑流水线），C-10
Graphics processing units（GPU，图形处理单元），
见 GPU computing，522-529
as accelerators（加速器），522
attribute interpolation（属性差值），C-43-44
defined（定义），46，506，C-3
evolution（演变），C-5
fallacies and pitfalls（谬误及陷阱），C-72-75
floating-point arithmetic（浮点运算），C-17，C-41-46，C-74
GeForce 8-series generation（GeForce 8 代系列），C-5
general computation（运用计算），C-73-74
General Purpose（GPGPUs）（通用 GPU），C-5
graphics mode（图形模式），C-6
graphics trends（趋势图），C-4
history（历史），C-3-4
logical graphics pipeline（逻辑图形流水线），C-13-14
mapping applications to（将应用映射到……），C-55-72
memory（存储器），523
multilevel caches and（多级 cache），522
N-body applications（多体问题应用），C-65-72
NVIDIA architecture（NVIDIA 体系结构），523-526
parallel memory system（并行存储系统），C-36-41
parallelism（并行性），523，C-76
performance doubling（性能加倍），C-4
perspective（观点），527-529
programming（编程），C-12-24
programming interfaces to（编程接口），C-17
real-time graphics（实时图像），C-13
summary（总结），C-76
Graphics shader programs（图形着色程序），C-14-15
Gresham's Law（葛莱欣法则），236，OL3.11-2
Grid computing（网格计算），533
Grids（网格），C-19
GTX（GTX 公司），280，548-553
Guard digits（保护位），
defined（定义），218
rounding with（舍入），219

H

Half precision（半精度），C-42
Halfwords（半字），110
Hamming，Richard，420
Hamming distance（汉明距离），420
Hamming Error Correction Code（ECC，汉明纠错码），420-421
calculating（计算），420-421
Handlers（处理程序）
defined（定义），449
ILB miss（ILB 缺失），448
Hard disks（硬盘），
access times（访问时间），23
defined（定义），23
Hardware（硬件），
as hierarchical layer（作为分层的层次），13
language of（……的语言），14-16
operations（操作），63-66
supporting procedures in（支持过程），96-106
synthesis（综合），B-21
translating microprograms to（将微程序转换为……），D-28-32
virtualizable（虚拟化），426
Hardware description languages（硬件描述语言），
见 Verilog defined（定义），B-20
using（使用），B-20-26
VHDL（一种硬件描述语言），B-20-21
Hardware multithreading（硬件多线程），514-517
coarse-grained（粗粒度的），514
options（选择），516
simultaneous（同时的），515-517
Hardware-based speculation（基于硬件的推测），341
Harvard architecture（哈佛结构），OL1.12-4
Hazard detection units（冒险检测单元），313-314
functions（函数），314
pipeline connections for（流水线连接），314
Hazards（冒险），见 Pipelining control，277-278，281-282，316-325
data（数据），278，303-316
forwarding and（传送），312
structural（结构的），277，294
Heap（堆）
allocating space on（在……上分配空间），104-106
defined（定义），104
Heterogeneous systems（异构系统），C-4-5
architecture（体系结构），C-7-9
defined（定义），C-3
Hexadecimal numbers（十六进制数），81-82
binary number conversion to（将二进制数转换

为……), 81-82
Hierarchy of memories (存储器层次), 12
High-level languages (高级语言), 14-16, A-6
benefits (收益), 16
computer architectures (计算机体系结构), OL2.21-5
importance (重要性), 16
High-level optimizations (高层次优化), OL2.15-4-2.15-5
Hit rate (命中率), 376
Hit time (命中时间)
cache performance and (cache性能), 401-402
defined (定义), 376
Hit under miss (命中少于缺失), 472
Hold time (保持时间), B-54
Horizontal microcode (水平型微码), D-32
Hot-swapping (热插拔), OL5.11-7
Human genome project (人类基因组计划), 4

I

I/O (输入/输出), A-38-40, OL6.9-2, OL6.9-3
memory-mapped (存储器映射), A-38
on system performance (系统性能方面), OL5.11-2
I/O benchmarks (输入/输出基准测试程序), 见 Benchmarks
IBM 360/85, (国际商用机器公司), OL5.17-7
IBM 701, (国际商用机器公司), OL1.12-5
IBM 7030, (国际商用机器公司), OL4.16-2
IBM ALOG (IBM ALOG), OL3.11-7
IBM Blue Gene (IBM蓝色基因), OL6.15-9-6.15-10
IBM Personal Computer (IBM 个人计算机), OL1.12-7, OL2.21-6
IBM System/360 computers (IBM 系统/360 计算机), OL1.12-6, OL3.11-6, OL4.16-2
IBM z/VM (IBM z/VM), OL5.17-8
ID stage (译码阶段)
branch execution in (分支执行), 319
load instructions (装入指令), 292
store instruction in (存储指令), 291
IEEE 754 floating-point standard (IEEE 754 浮点标准), 见 Floating point first chips (最早的芯片), 198, 199, OL3.11-8-3.11-10, OL3.11-8-3.11-9
in GPU arithmetic (在图形处理单元算法中), C-42-43
implementation (实现), OL3.11-10
rounding modes (舍入模式), 219
today (现今), OL3.11-10
If statements (条件语句), 114
I-format (I格式), 83
If-then-else (If-then-else语句), 91
Immediate addressing (立即寻址), 116
Immediate instructions (立即指令), 72
Imprecise interrupts (不精确中断), 331, OL4.16-4
Index-out-of-bounds check (指针越界检查), 94-95
Induction variable elimination (消除引导变量), OL2.15-7
Inheritance (继承), OL2.15-15
In-order commit (按序提交), 341
Input devices (输入设备), 16
Inputs (输入), 261
Instances (实体), OL2.15-15
Instruction count (指令计数), 36, 38
Instruction decode/register file read stage (指令译码/读寄存器堆阶段)
control line (控制线), 300
load instruction (装入指令), 289
store instruction (存储指令), 294
Instruction execution illustrations (指令执行图解), OL4.13-16-4.13-17
clock cycle 9, (时钟周期), OL4.13-24
clock cycles 1 and 2 (时钟周期1和2), OL4.13-21
clock cycles 3 and 4 (时钟周期3和4), OL4.13-22
clock cycles 5 and 6 (时钟周期5和6), OL4.13-23, OL4.13-23
clock cycles 7 and 8 (时钟周期7和8), OL4.13-24
examples (举例), OL4.13-20-4.13-25
forwarding (旁路/转发), OL4.13-26-4.13-31
no hazard (无冒险), OL4.13-17
pipelines with stalls and forwarding (流水线中的暂停和转移), OL4.13-26, OL4.13-20
Instruction fetch stage (取指令阶段)
control line (控制线), 300
load instruction (装入指令), 289
store instruction (存储指令), 294
Instruction formats (指令格式), 157
ARM (ARM公司), 148
defined (定义), 81
desktop/server RISC architectures (台式机/服务

器精简指令体系结构), E-7
embedded RISC architecture (嵌入式精简指令体系结构), E-8
I-type (I型), 83
J-type (J型), 113
jump instruction (跳转指令), 270
MIPS (每秒百万条指令), 148
R-type (R型), 83, 261
x86 (x86计算机), 157
Instruction latency (指令延迟), 356
Instruction mix (指令混合比), 39, 014. 12-10
Instruction set architecture (指令集架构)
ARM (ARM公司), 145-147
branch address calculation (转移地址计算), 254
defined (定义), 22, 52
history (历史), 163
maintaining (维护), 52
protection and (保护), 427
thread (线程), C-31-34
virtual machine support (虚拟机支持), 426-427
Instruction sets (指令集), 235, C-49
ARM (ARM公司), 324
design for pipelining (流水线设计), 277
MIPS (每秒百万条指令), 62, 161, 234
MIPS-32 (MIPS-32指令集), 235
Pseudo MIPS (伪MIPS指令), 233
x86 growth (x86增长), 161
Instruction-level parallelism (ILP, 指令级并行), 见 Parallelism, 354
compiler exploitation (编译器开发), 014. 16-5-4. 16-6
defined (定义), 43, 333
exploitation increasing (开发增长), 343
and matrix multiply (矩阵乘法), 351-354
Instructions (指令), 见 Arithmetic instructions; MIPS; Operands, 60-164, E-25-27, E-40-42
add immediate (加立即数), 72
addition (加法), 180, A-51
Alpha (阿尔法), E-27-29
arithmetic-logical (算术-逻辑), 251, A-51-57
ARM (ARM公司), 145-147, E-36-37
assembly (汇编), 66
basic block (基本块), 93
branch (分支), A-59-63
cache-aware (cache感知), 482
comparison (比较), A-57-59
conditional branch (条件分支), 90
conditional move (条件传送), 324
constant-manipulating (常量操作), A-57
conversion (转换), A-75-76
core (核), 233
data movement (数据移动), A-70-73
data transfer (数据传送), 68
decision-making (决策制定), 90-96
defined (定义), 14, 62
desktop RISC conventions (台式精简指令集约定), E-12
division (除法), A-52-53
as electronic signals (电子信号), 80
embedded RISC conventions (嵌入式精简指令集约定), E-15
encoding (编码), 82
exception and interrupt (异常和中断), A-80
exclusive OR (异或), A-57
fetching (读取), 253
fields (字段), 80
floating-point (x86) (浮点型 (x86)), 224
floating-point (浮点型), 211-213, A-73-80
flushing (冲刷), 318, 319, 331
immediate (立即数), 72
introduction to (引言), 62-63
jump (跳转), 95, 97, A-63-64
left-to-right flow (从左向右流动), 287-288
load (装入), 68, A-66-68
load linked (装入链接), 122
logical operations (逻辑运算), 87-89
M32R (瑞萨M32R系列32位RISC微机), E-40
memory access (存储器访问), C-33-34
memory-reference (存储器引用/访问), 245
multiplication (乘法), 188, A-53-54
negation (逻辑非), A-54
nop (空操作), 314
PA-RISC (惠普公司的RISC芯片), E-34-36
performance (性能), 35-36
pipeline sequence (流水顺序), 313
PowerPC (是一种RISC架构的CPU其基本的设计源自IBM的POWER), E-12-13, E-32-34
PTX (PTX), C-31, C-32
remainder (余数), A-55
representation in computer (在计算机中的表示), 80-87
restartable (可重启的), 450

resuming（恢复），450
R-type（R 型），252
shift（移位），A-55-56
SPARC（可扩充处理器架构），E-29-32
store（存储），71，A-68-70
store conditional（条件存储），122
subtraction（减法），180，A-56-57
SuperH（SuperH），E-39-40
thread（线程），C-30-31
Thumb（重点），E-38
trap（陷阱），A-64-66
vector（向量），510
as words（以字的形式），62
x86（x86 计算机），149-155
Instructions per clock cycle（IPC，每时钟周期执行的指令条数），333
Integrated circuits（IC，集成电路），见 specific chips，19
cost（成本/开销），27
defined（定义），25
manufacturing process（制造工艺），26
very large-scale（VLSI，超大规模集成电路），25
Intel Core i7（英特尔酷睿 i7），46-49，244，501，548-553
address translation for（地址转换），471
architectural registers（体系结构级寄存器），347
caches in（……中的 cache），472
memory hierarchies of（存储器层次），471-475
microarchitecture（微体系结构），338
performance of（……的性能），473
SPEC CPU benchmark（SPEC CPU 基准测试程序），46-48
SPEC power benchmark（SPEC 功耗基准测试程序），48-49
TLB hardware for（TLB 硬件），471
Intel Core i7 920（英特尔酷睿 i7），346-349
microarchitecture（微体系结构），347
Intel Core i7（英特尔酷睿 i7），960
benchmarking and rooflines of（屋顶层轮廓基准测试），548-553
Intel Core i7 Pipelines（英特尔酷睿 i7 流水线技术），344，346-349
memory components（存储组件），348
performance（性能），349-351
program performance（程序性能），351
specification（规格说明书），345
Intel IA-64 architecture（64 位的英特尔架构），OL2.21-3
Intel Paragon（英特尔典范），OL6.15-8
Intel Threading Building Blocks（英特尔线程构建模块），C-60
Intel x86 microprocessors（英特尔 x86 微处理器），
clock rate and power for（时钟频率和功率），40
Interference graphs（干扰图），OL2.15-12
Interleaving（交叉存取），398
Interprocedural analysis（过程间分析），OL2.15-14
Interrupt enable（中断使能），447
Interrupt handlers（中断处理程序），A-33
Interrupt-driven I/O（中断驱动输入/输出），OL6.9-4
Interrupts（中断），
defined（定义），180，326
event types and（事件类型），326
exceptions versus（异常与……），325-326
imprecise（不精确的），331，OL4.16-4
instructions（指令），A-80
precise（精确的），332
vectored（向量），327
Intrinsity FastMATH processor（内置 FastMATH 处理机），395-398
caches（高速缓存），396
data miss rates（数据缺失率），397，407
read processing（读处理），442
TLB（快表），440
write-through processing（写直达处理），442
Inverted page tables（反置页表），436
Issue packets（事件包），334

J

j（Jump）（跳转），64
jal（Jump And Link）（跳转链接指令），64
Java
bytecode（字节码），131
bytecode architecture（字节码结构），OL2.15-17
characters in（……特性），109-111
compiling in（编译……），OL2.15-19-2.15-20
goals（目标），131
interpreting（翻译），131，145，OL2.15-15-2.15-16
keywords（关键字），OL2.15-21
method invocation in（方法调用），OL2.15-21
pointers（指针），OL2.15-26

primitive types（基本数据类型），OL2.15-26
programs starting（程序开始），131-132
reference types（引用类型），OL2.15-26
sort algorithms（排序算法），141
strings in（字符串），109-111
translation hierarchy（翻译层次结构），131
while loop compilation in（while 循环结构编译），OL2.15-18-2.15-19
Java Virtual Machine（JVM，Java 虚拟机），145，OL2.15-16
jr（Jump Register，跳转寄存器），64
J-type instruction format（J 型指令格式），113
Jump instructions（跳转指令），254，E-26
branch instruction versus（分支指令与……），270
control and datapath for（控制和数据通路），271
implementing（实现），270
instruction format（指令格式），270
list of（……的清单），A-63-64
Just In Time（JIT）compilers（即时编译器），132，560

K

Karnaugh maps（卡诺图），B-18
Kernel mode（核心态），444
Kernels（核心程序），
CUDA（统一计算架构），C-19，C-24
defined（定义），C-19
Kilobyte（千字节），6

L

Labels（标签），
global（全局），A-10，A-11
local（局部），A-11
LAPACK（线性代数程序包），230
Large scale multiprocessors（大规模多处理机），OL6.15-7，OL6.15-9-6.15-10
Latches（锁存器），
D latch（D 锁存器），B-51，B-52
defined（定义），B-51
Latency（延迟）
instruction（指令），356
memory（存储器），C-74-75
pipeline（流水线），286
use（使用），336-337
lbu（Load Byte Unsigned，无符号数装入寄存器），64
Leaf procedures（叶子过程），见 Procedures
defined（定义），100
example（实例），109
Least recently used（LRU，最近最少使用），
as block replacement strategy（作为块替换策略），457
defined（定义），409
pages（页），434
Least significant bits（最低有效位），B-32
defined（定义），74
SPARC（可扩充处理器架构），E-31
Left-to-right instruction flow（从左向右指令流），287-288
Level-sensitive clocking（对层次敏感的时钟），B-74，B-75-76
defined（定义），B-74
two-phase（二相），B-75
lhu（Load Halfword Unsigned）（装入半字节无符号数），64
li（Load Immediate，立即数装入），162
Link（链接），OL6.9-2
Linkers（链接器），126-129，A-18-19
defined（定义），126，A-4
executable files（可执行文件），126，A-19
function illustration（功能说明），A-19
steps（步骤），126
using（使用），126-129
Linking object files（目标文件链接），126-129
Linpack（线性系统软件包），538，OL3.11-4
Liquid crystal displays（LCD，液晶显示器），18
LISP SPARC support（LISP 语言可扩充处理器架构支持），E-30
Little-endian byte order（低位低址字节次序），A-43
Live range（生命期），OL2.15-11
Livermore Loops（Livermore 循环），OL1.12-11
ll（Load Linked，装入链接），64
Load balancing（负载平衡），505-506
Load instructions（装入指令），见 Store instructions
access（访问），C-41
base register（基址寄存器），262
block（块），149
compiling with（用……编译），71
datapath in operation for（运行中的数据通路），267
defined（定义），68

details（细节），A-66-68
EX stage（执行阶段），292
floating-point（浮点型），A-76-77
halfword unsigned（半字无符号数），110
ID stage（指令译码阶段），291
IF stage（取指阶段），291
linked（已链接的），122，123
list of（……的列表），A-66-68
load byte unsigned（无符号数装入寄存器），76
load half（装入半字），110
load upper immediate（装入立即数高字节），112，113
MEM stage（访存阶段），293
pipelined datapath in（流水线数据通路），296
signed（有符号的），76
unit for implementing（执行单元），255
unsigned（无符号的），76
WB stage（写回阶段），293
Load word（装入字），68，71
Loaders（装载器），129
Loading（装载），A-19-20
Load-store architectures（装载－存储体系结构），OL2.21-3
Load-use data hazard（装入－使用数据冲突），280，318
Load-use stalls（装入－使用停顿），318
Local area networks（LAN，局域网），见 Networks，24.
Local labels（局部标记），A-11
Local memory（局部存储器），C-21，C-40
Local miss rates（局部缺失率），416
Local optimization（局部优化），见 Optimization，OL2.15-5
implementing（实现），OL2.15-8
Locality（局部性），
principle（原则），374
spatial（空间的），374，377
temporal（时间的），374，377
Lock synchronization（同步锁），121
Locks（锁），518
Logic（逻辑）
address select（地址选择），D-24，D-25
ALU control（ALU 控制），D-6
combinational（组合的），250，B-5，B-9-20
components（组件），249
control unit equations（控制部件方程式），D-11
design（设计），248-251，B-1-79
equations（方程式），B-7
minimization（最小化），B-18
programmable array（PAL，可编程阵列），B-78
sequential（顺序的），B-5，B-56-58
two-level（两级），B-11-14
Logical operations（逻辑操作），87-89
AND（与），88，A-52
ARM（ARM 公司），149
desktop RISC（台式精简指令系统计算机），E-11
embedded RISC（嵌入式精简指令系统计算机），E-14
MIPS（每秒百万条指令），A-51-57
NOR（或非），89，A-54
NOT（非），89，A-55
OR（或），89，A-55
shifts（转换），87
Long instruction word（LIW，长指令字），OL4.16-5
Lookup tables（LUT，查找表），B-79
Loop unrolling（循环展开），
defined（定义），338，OL2.15-4
for multiple-issue pipelines（多发射流水线），338
register renaming and（寄存器重命名），338
Loops（循环），92-93
conditional branches in（条件分支），114
for（for 语句），141
prediction and（预测），321-323
test（测试），142，143
while compiling（while 语句编译），92-93
lui（Load Upper Imm.，装入立即数高位），64
lw（Load Word，装入字），64
lwc1（Load FP Single，装入单精度浮点数），A-73

## M

M32R（瑞萨 M32R 族 32 位 RISC 微机），E-15，E-40
Machine code（机器代码），81
Machine instructions（机器指令），81
Machine language（机器语言），15
branch offset in（分支偏移），115
decoding（译码），118-120
defined（定义），14，81，A-3
floating-point（浮点型），212

illustrated（举例说明），15
MIPS（每秒百万条指令），85
SRAM（静态随机存储器），21
translating MIPS assembly language into（将 MIPS 汇编语言转换为……），84
Macros（宏命令），
defined（定义），A-4
example（实例），A-15-17
use of（用于……），A-15
Main memory（主存储器），见 Memory，428
defined（定义），23
page tables（页表），437
physical addresses（物理地址），428
Mapping applications（映射应用），C-55-72
Mark computers（标记计算机），OL1.12-14
Matrix multiply（矩阵乘法），225-228，553-555
Mealy machine（Mealy 有限状态机），463-464，B-68，B-71，B-72
Mean time to failure（MTTF，平均无故障时间），418
improving（改进），419
versus AFR of disks（相对于磁盘的容许故障率），419-420
Media Access Control（MAC）address（媒体访问控制地址），OL6.9-7
Megabyte（兆字节），6
Memory（存储器），
addresses（地址），77
affinity（紧密性理），545
atomic（原子性），C-21
bandwidth（带宽），380-381，397
cache（高速缓冲存储器），21，383-398，398-417
CAM（计算机辅助制造），408
constant（常量），C-40
control（控制），D-26
defined（定义），19
DRAM（动态随机存取存储器），19，379-380，B-63-65
flash（闪存），23
global（全局），C-21，C-39
GPU（图形处理单元），523
instructions datapath for（指令……的数据通路），256
layout（布局），A-21
local（局部），C-21，C-40
main（主要的），23
nonvolatile（非易失性），22
operands（操作数），68-69
parallel system（并行系统），C-36-41
read-only（ROM）（只读存储器），B-14-16
SDRAM（同步动态随机存储器），379-380
secondary（辅助的），23
shared（共享的），C-21，C-39-40
spaces（空间），C-39
SRAM（静态随机存储器），B-58-62
stalls（停顿），400
technologies for building（生成技术），24-28
texture（结构），C-40
usage（用法），A-20-22
virtual（虚拟），427-454
volatile（易失性），22
Memory access instructions（访存指令），C-33-34
Memory access stage（访存阶段），
control line（控制线），302
load instruction（装入指令），292
store instruction（存储指令），292
Memory bandwidth（存储器带宽），551，557
Memory consistency model（存储一致性模型），469
Memory elements（存储单元），B-50-58
clocked（时钟控制），B-51
D flip-flop（D 触发器），B-51，B-53
D latch（D 锁存器），B-52
DRAMs（动态随机存取存储器），B-63-67
flip-flop（触发器），B-51
hold time（保持时间），B-54
latch（锁存器），B-51
setup time（建立时间），B-53，B-54
SRAMs（静态随机存储器），B-58-62
unclocked（不受时钟约束），B-51
Memory hierarchies（存储器层次），545
of ARM cortex-A8（ARM cortex-A8 芯片），471-475
block（or line）（块（或行）），376
cache performance（cache 性能），398-417
caches（高速缓存），383-417
common framework（通用框架），454-461
defined（定义），375
design challenges（设计挑战），461
development（发展），OL5.17-6-5.17-8
exploiting（开发），372-498
of Intel core i7（英特尔酷睿 i7 处理器），471-475
level pairs（水平对），376

multiple levels（多层次），375
overall operation of（整体操作），443-444
parallelism and（并行度），466-470，OL5.11-2
pitfalls（陷阱），478-482
program execution time and（程序执行时间），417
quantitative design parameters（定量设计参数），454
redundant arrays and inexpensive disks（廉价磁盘冗余阵列），470
reliance on（可信性），376
structure（结构），375
structure diagram（结构图），378
variance（变化），417
virtual memory（虚拟存储器），427-454
Memory rank（存储体），381
Memory technologies（存储技术），378-383
disk memory（磁盘存储器），381-383
DRAM technology（动态随机存取存储器技术），378，379-381
flash memory（闪存），381
SRAM technology（静态随机存储器技术），378，379
Memory-mapped I/O（存储器映射输入/输出），OL6.9-3
use of（用作……），A-38
Memory-stall clock cycles（存储器停顿时钟周期），399
Message passing（消息传送），
defined（定义），529
multiprocessors（多处理器），529-534
Metastability（亚稳态），B-76
Methods（方法），
defined（定义），OL2.15-5
invoking in Java（在 Java 中调用），OL2.15-20-2.15-21
static（静态），A-20
mfc0（Move From Control）（从控制器中移出），A-71
mfhi（Move From Hi）（从 Hi 中移出），A-71
mflo（Move From Lo）（从 Lo 中移出），A-71
Microarchitectures（微体系结构），347
Intel Core i7（英特尔酷睿 i7 处理器），920，347
Microcode（微码）
assembler（汇编器），D-30
control unit as（控制单元），D-28
defined（定义），D-27
dispatch ROMs（只读存储器调度），D-30-31
horizontal（水平的），D-32
vertical（垂直的），D-32
Microinstructions（微指令），D-31
Microprocessors（微处理器），
design shift（设计转换），501
multicore（多核），8，43，500-501
Microprograms（微程序），
as abstract control representation（作为抽象控制的表达），D-30
field translation（现场翻译），D-29
translating to hardware（转换为硬件），D-28-32
Migration（迁移），467
Million instructions per second（MIPS）（每秒百万条指令），51
Minterms（最小项），
defined（定义），B-12，D-20
in PLA implementation（在可编程逻辑阵列中的实现），D-20
MIP-map（MIP 映射），C-44
MIPS（MIPS 技术公司），64，84，A-45-80
addressing for 32-bit immediates（32 位立即数寻址），116-118
addressing modes（寻址方式），A-45-47
arithmetic core（算法核心），233
arithmetic instructions（算术运算指令），63，A-51-57
ARM similarities（类似 ARM），146
assembler directive support（汇编指令支持），A-47-49
assembler syntax（汇编语法），A-47-49
assembly instruction mapping（汇编指令映射），80-81
branch instructions（分支指令），A-59-63
comparison instructions（比较指令），A-57-59
compiling C assignment statements into（将 C 的赋值语句编译为……），65
compiling complex C assignment into（将复杂的 C 赋值编译为……），65-66
constant-manipulating instructions（常量操作指令），A-57
control registers（控制寄存器），448
control unit（控制单元），D-10
CPU（中央处理器），A-46
divide in（分成），194
exceptions in（……中的异常），326-327
fields（字段），82-83

floating-point instructions（浮点指令），211-213
FPU（浮点运算单元），A-46
instruction classes（指令类别），163
instruction encoding（指令编码），83，119，A-49
instruction formats（指令格式），120，148，A-49-51
instruction set（指令集），62，162，234
jump instructions（跳转指令），A-63-66
logical instructions（逻辑运算指令），A-51-57
machine language（机器语言），85
memory addresses（存储器地址），70
memory allocation for program and data（程序和数据的存储器分配），104
multiply in（在……中相乘），188
opcode map（操作码映射），A-50
operands（操作数），64
Pseudo（伪），233，235
register conventions（寄存器使用约定），105
static multiple issue with（静态多发射），335-338
MIPS core（MIPS 核），
architecture（体系结构），195
arithmetic/logical instructions not in（不包含算术/逻辑指令），E-21，E-23
common extensions to（通用扩展），E-20-25
control instructions not in（不包含控制指令），E-21
data transfer instructions not in（不包含数据传送指令），E-20，E-22
floating-point instructions not in（不包含浮点指令），E-22
instruction set（指令集），233，244-248，E-9-10
MIPS-16（MIPS），
16-bit instruction set（16 位指令集），E-41-42
immediate fields（立即数字段），E-41
instructions（指令），E-40-42
MIPS core instruction changes（MIPS 核指令变化），E-42
PC-relative addressing（PC 相对寻址），E-41
MIPS-32 instruction set（MIPS32 位指令集），235
MIPS-64 instructions（MIPS 64 位指令），E-25-27
conditional procedure call instructions（有条件的过程调用指令），E-27
constant shift amount（连续的位移量），E-25
jump/call not PC-relative（PC 不相关跳转/调用），E-26
move to/from control registers（跳转到/来自寄存器），E-26
nonaligned data transfers（不成行的（边界未对齐的）数据传送），E-25
NOR（或），E-25
parallel single precision floating-point operations（并行单精度浮点运算），E-27
reciprocal and reciprocal square root（倒数和倒数平方根），E-27
SYSCALL（系统调用），E-25
TLB instructions（TLB 指令），E-26-27
Mirroring（镜像），OL5.11-5
Miss penalty（缺失损失），
defined（定义），376
determination（决策），391-392
multilevel caches reducing（多级高速缓存减少），410
Miss rates（缺失率），
block size versus（块的大小），392
data cache（数据高速缓存），455
defined（定义），376
global（全局），416
improvement（改进），391-392
Intrinsity FastMATH processor（内置的 FastMATH 处理器），397
local（本地），416
miss sources（缺失原因），460
split cache（缓存拆分），397
Miss under miss（缺失情况下的缺失），472
MMX（MultiMedia eXtension）（多媒体扩展），224
Modules（模块），A-4
Moore machines（摩尔机），463-464，B-68，B-71，B-72
Moore's law（摩尔定律），11，379，522，OL6.9-2，C-72-73
Most significant bit（最高位），
1-bit ALU for（1 位 ALU），B-33
defined（定义），74
move（传送），139
Move instructions（传送指令），A-70-73
coprocessor（协处理信息），A-71-72
details（细节），A-70-73
floating-point（浮点），A-77-78
MS-DOS（微软－磁盘操作系统），OL5.17-11
mul.d（FP Multiply Double，双精度浮点乘法），A-78
mul.s（FP Multiply Single，单精度浮点乘法），A-78

mult（Multiply，乘法），A-53
Multicore（多核），517-521
Multicore multiprocessors（多核多处理器），8，43
  defined（定义），8，500-501
MULTICS（Multiplexed Information and Computing Service，复用信息和计算服务），OL5.17-9-5.17-10
Multilevel caches（多级缓存），见 Caches complications（并发性），416
  defined（定义），398，416
  miss penalty reducing（减少缺失代价），410
  performance of（性能），410
  summary（总结），417-418
Multimedia extensions（多媒体扩展），
  desktop/server RISCs（桌面/服务器 RISCs），E-16-18
  as SIMD extensions to instruction sets（作为 SIMD 指令集的扩展），OL6.15-4
  vector versus（矢量），511-512
Multiple dimension arrays（多维数组），218
Multiple instruction multiple data（MIMD，多指令多数据），558
  defined（指令），507，508
  first multiprocessor（第一个多处理器），OL6.15-14
Multiple instruction single data（MISD，多指令单数据），507
Multiple issue（多发射），332-339
  code scheduling（代码调度），337-338
  dynamic（动态），333，339-341
  issue packets（事件包），334
  loop unrolling and（循环展开），338
  processors（处理器），332，333
  static（静态的），333，334-339
  throughput and（吞吐量），342
Multiple processors（多个处理器），553-555
Multiple-clock-cycle pipeline diagrams（多时钟周期流水线图），296-297
  five instructions（五条指令），298
  illustrated（说明），298
Multiplexors（多路选择器），B-10
  controls（控制），463
  in datapath（数据通路），263
  defined（定义），246
  forwarding control values（转发/旁路控制值），310
  selector control（选择控制），256-257
  two-input（双输入），B-10
Multiplicand（被乘数），183
Multiplication（乘法），见 Arithmetic，183-188.
  fast hardware（快速，硬件），188
  faster（更快），187-188
  first algorithm（第一种算法）185
  floating-point（浮点），206-208，A-78
  hardware（硬件），184-186
  instructions（指令），188，A-53-54
  in MIPS（在 MIPS 中），188
  multiplicand（被乘数），183
  multiplier（乘法器），183
  operands（操作数），183
  product（乘积），183
  sequential version（时序版本），184-186
  signed（有符号的），187
Multiplier（乘法器），183
Multiply algorithm（乘法算法），186
Multiply-add（MAD，乘累加），C-42
Multiprocessors（多处理器），
  benchmarks（基准程序），538-540
  bus-based coherent（基于总线的一致性），OL6.15-7
  defined（定义），500
  historical perspective（历史的角度），561
  large-scale（大规模），OL6.15-7-6.15-8，OL6.15-9-6.15-10
  message-passing（消息传递），529-534
  multithreaded architecture（多线程架构），C-26-27，C-35-36
  organization（组织），499，529
  for performance（性能），559
  shared memory（共享存储器），501，517-521
  software（软件），500
  TFLOPSOL6.15-6
  UMA（一致性存储器访问），518
Multistage networks（多级网络），535
Multithreaded multiprocessor architecture（多线程多处理器架构），C-25-36
  conclusion（结论），C-36
  ISA（指令集体系结构），C-31-34
  massive multithreading（大规模多线程），C-25-26
  multiprocessor（多处理器），C-26-27
  multiprocessor comparison（多处理器比较），C-35-36
  SIMT（单指令多线程），C-27-30
  special function units（SFU，特殊功能单元），C-35

streaming processor (SP, 流处理器), C-34
thread instructions (线程指令), C-30-31
threads/thread blocks management (线程/线程块管理), C-30
Multithreading (多线程), C-25-26
coarse-grained (粗粒度), 514
defined (定义), 506
fine-grained (细粒度), 514
hardware (硬件), 514-517
simultaneous (SMT, 同时(同时多线程)), 515-517
multu (Multiply Unsigned, 无符号乘法), A-54
Must-information (必备信息), OL2.15-5
Mutual exclusion (互斥), 121

## N

Name dependence (名字相关), 338
NAND gates (与非门), B-8
NAS (NASA Advanced Supercomputing, 一种并行的基准测试程序), 540
N-body (N体)
all-pairs algorithm (完全算法), C-65
CPU simulation (CPU模拟), C-71
mathematics (数学), C-65-67
multiple threads per body (每个体多个线程), C-68-69
optimization (优化), C-67
performance comparison (性能比较), C-69-70
results (结果), C-70-72
shared memory use (存储器共享使用), C-67-68
Negation instructions (求相反数的指令), A-54, A-78-79
Negation shortcut (求相反数的简便方法), 76
Nested procedures (嵌套过程), 100-102
compiling recursive procedure showing (对递归过程进行编译的展示), 101-102
NetFPGA 10-Gigagit Ethernet card (NetFPGA 10-G以太卡), OL6.9-2, OL6.9-3
Network of Workstations (工作站的网络), OL6.15-8-6.15-9
Network topologies (网络拓扑结构), 534-537
implementing (实现), 536
multistage (多级), 537
Networking (网络), OL6.9-4
operating system in (操作系统), OL.6.9-4-6.9-5
performance improvement (性能提高), OL6.9-7-6.9-10
Networks (网络), 23-24
advantages (优势), 23
bandwidth (带宽), 535
crossbar (交叉开关), 535
fully connected (全互连), 535
local area (LAN, 局域网), 24
multistage (多级), 535
wide area (WAN, 广域网), 24
Newton's iteration (牛顿迭代), 218
Next state (下一状态),
nonsequential (非时序), D-24
sequential (时序), D-23
Next-state function (下一状态函数), 463, B-67
defined (定义), 463
implementing with sequencer (序列发生器的实现), D-22-28
Next-state outputs (下一状态输出), D-10, D-12-13
example (示例), D-12-13
implementation (实现), D-12
logic equations (逻辑方程), D-12-13
truth tables (真值表), D-15
No Redundancy (RAID 0) (无冗余), OL5.11-4
No write allocation (没有写分配), 394
Nonblocking assignment (非阻塞分配), B-24
Nonblocking caches (非阻塞高速缓存), 344, 472
Nonuniform memory access (NUMA, 非一致性存储器访问), 518
Nonvolatile memory (非易失性存储器), 22
Nope (空操作), 314
nor (NOR) (或), 64
NOR gates (或门), B-8
cross-coupled (交叉耦合), B-50
D latch implemented with (D锁存实现), B-52
NOR operation (或操作), 89, A-54, E-25
NOT operation (非操作), 89, A-55, B-6
Numbers (数)
binary (二进制), 73
computer versus real-world (计算机与现实世界) 221
decimal (十进制), 73, 76
denormalized (非规范化), 222
hexadecimal (十六进制), 81-82

signed（有符号），73-78
unsigned（无符号），73-78
NVIDIA GeForce 8800，C-46-55（NVIDIA GeForce 8800），
all-pairs N-body algorithm（对 N 体的所有算法），C-71
dense linear algebra computations（稠密线性代数算法），C-51-53
FFT performance（FFT 性能），C-53
instruction set（指令集），C-49
performance（性能），C-51
rasterization（网格化），C-50
ROP（ROP），C-50-51
scalability（可扩放性），C-51
sorting performance（排序性能），C-54-55
special function approximation statistics（特殊功能的近似统计），C-43
special function unit（SFU，特殊功能单元），C-50
streaming multiprocessor（SM，流式多处理器），C-48-49
streaming processor（流处理器），C-49-50
streaming processor array（SPA，流处理器阵列），C-46
texture/processor cluster（TPC，纹理/处理器集群），C-47-48
NVIDIA GPU architecture（NVIDIA 图形处理单元结构），523-526
NVIDIA GTX，280，548-553
NVIDIA Tesla CPU，548-553

O

Object files（目标文件），125，A-4
debugging information（调式信息），124
defined（定义），A-10
format（格式），A-13-14
header（头部），125，A-13
linking（链接），126-129
relocation information（重定位信息），125
static data segment（静态数据段），125
symbol table（符号表），125，126
text segment（正文段），125
Object-oriented languages（面向对象的语言），见 Java brief history（简史），OL2.21-8
defined（定义），145，OL2.15-5
One's complement（补码），79，B-29
Opcodes（操作码）
control line setting and（控制线的设置），264
defined（定义），82，262
OpenGL（开放图形语言），C-13
OpenMP（Open Multiprocessing，开放多处理器），520540
Operands（操作数），见 Instructions，66-73
32-bit immediate（32 位立即数），112-113
adding（加），179
arithmetic instructions（算术运算指令），66
compiling assignment when in memory（编译分配存储器），69
constant（常量），72-73
division（除法），189
floating-point（浮点），212
memory（存储器），68-69
MIPS（一种采用精简指令集的处理器架构），64
multiplication（乘），183
shifting（移位），148
Operating systems（操作系统）
brief history（简史），OL5.17-9-5.17-12
defined（定义），13
encapsulation（封装），22
in networking（在网络中），OL6.9-4-6.9-5
Operations（操作），
atomicimplementing（原子实现），121
hardware（硬件），63-66
logical（逻辑），87-89
x86 integer（x86 整数），152，154-155
Optimization（优化），
class explanation（类说明），OL2.15-14
compiler（编译器），141
control implementation（控制实现），D-27-28
global（全局的），OL2.15-5
high-level（高级），OL2.15-4-2.15-5
local（局部的），OL2.15-5，OL2.15-8
manual（手册），144
or（或），64
OR operation（或操作），89，A-55，B-6
ori（Or Immediate，或立即数），64
Out-of-order execution（乱序执行），
defined（定义），341
performance complexity（性能复杂性），416
processors（处理器），344
Output devices（输出装置），16

Overflow（溢出）
defined（定义），74，198
detection（检测），180
exceptions（异常），329
floating-point（浮点），198
occurrence（事件），75
saturation and（饱和度），181
subtraction（减法），179

P

P + Q redundancy（RAID 6，P + Q 冗余（RAID 6）），OL5.11-7
Packed floating-pointformat（浮点格式），224
Page faults（缺页故障），见 Virtual memory，434
for data access（数据访问），450
defined（定义），428
handling（处理），429，446-453
virtual address causing（虚拟地址导致的），449，450
Page tables（页表），456
defined（定义），432
illustrated（举例说明），435
indexing（索引），432
inverted（倒置），436
levels（级），436-437
main memory（主存储器），437
register（寄存器），432
storage reduction techniques（减少存储技术），436-437
updating（更新），432
VMM（虚拟机），452
Pages（页），见 Virtual memory
defined（定义），428
dirty（脏），437
finding（查找），432-434
LRU（最近最少使用算法），434
offset（偏移），429
physical number（物理号），429
placing（放置），432-434
Size（尺寸），430
virtual number（虚号），429
Parallel bus（并行总线），OL6.9-3
Parallel execution（并行执行），121
Parallel memory system（并行存储系统），见 Graphics processing units（GPUs），C-36-41
caches（高速缓存），C-38
constant memory（常量存储器），C-40
DRAM considerations（动态随机访问存储器方面的考虑），C-37-38
global memory（全局存储器），C-39
load/store access（装载/存储访问），C-41
local memory（本地存储器），C-40
memory spaces（存储空间），C-39
MMU（存储器管理单元），C-38-39
ROP（ROP），C-41
shared memory（共享存储器），C-39-40
surfaces（表面），C-41
texture memory（纹理存储器），C-40
Parallel processing programs（并行处理器程序），502-507
creation difficulty（创造难度），502-507
defined（定义），501
for message passing（消息传递），519-520
great debates in（主要的考虑），OL6.15-5
for shared address space（共享地址空间），519-520
use of（使用），559
Parallel reduction（并行的降低），C-62
Parallel scan（并行扫描），C-60-63
CUDA template（CUDA 模板），C-61
inclusive（包容性），C-60
tree-based（基于树），C-62
Parallel software（并行软件），501
Parallelism（并行性），12，43，332-344
and computers arithmetic（计算机算术），222-225
data-level（数据级），233，508
debates（讨论），OL6.15-5-6.15-7
GPUs and（图形处理单元），523，C-76
instruction-level（指令级），43，332，343
memory hierarchies and（存储器层次），466-470，OL5.11-2
multicore and（多核），517
multiple issue（多发射）332-339
multithreading and（多线程），517
performance benefits（性能提升），44-45
process-level（进程级），500
redundant arrays and inexpensive disks（廉价磁盘冗余阵列），470
subword（子字），P-17
task（任务），C-24
task-level（任务级），500
thread（线程），C-22

Paravirtualization（部分虚拟化），482
PA-RISC（PA-RISC），E-14，E-17
branch vectored（分支向量），E-35
conditional branches（条件分支），E-34，E-35
debug instructions（调式指令），E-36
decimal operations（十进制运算），E-35
extract and deposit（提取和存放），E-35
instructions（指令），E-34-36
load and clear instructions（加载并清除指令），E-36
multiply/add and multiply/subtract（乘法/加法和乘法/除法），E-36
nullification（无效），E-34
nullifying branch option（无效分支选项），E-25
store bytes short（存储短字节），E-36
synthesized multiply and divide（合成的乘法和除法），E-34-35
Parity（奇偶），OL5.11-5
bits（位），421
code（码），420，B-65
PARSEC（Princeton Application Repository for Shared Memory Computers，多线程应程序组成的测试程序集），540
Pass transistor（导通晶体管），B-63
PCI-Express（PCIe），537，C-8，OL6.9-2
PC-relative addressing（程序计数器相对寻址），114，116
Peak floating-point performance（浮点性能峰值），542
Pentium bug morality play（Pentium bug 闹剧），231-232
Performance（性能），28-36
assessing（评估），28
classic CPU equation（经典的 CPU 方程），36-40
components（组件），38
CPU（图形处理单元），33-35
defining（定义），29-32
equation using（方程使用），36
improving（改善）34-35
instructions（指令），35-36
measuring（测量），33-35，OL1.12-10
program（程序），39-40
ratio（比率），31
relative（相对的），31-32
response time（响应时间），30-31
sorting（排序），C-54-55
throughput（吞吐量），30-31
time measurement（时间测量），32
Personal computers（PC，个人计算机），7
defined（定义），5
Personal mobile device（PMD，个人移动设备），
define（定义），7
Petabyte（千万亿字节），6
Physical addresses（物理地址），428
mapping to（映射），428-429
space（空间），517，521
Physically addressed caches（物理地址高速缓存），448
Pipeline registers（流水线寄存器），
before forwarding（转发前），309
dependence（相关），308
forwarding unit selection（转发单元选择），312
Pipeline stall（流水线阻塞），280
avoiding with code reordering（避免代码重排序），280
data hazards and（数据冒险），313-316
insertion（插入），315
load-use（加载使用），318
as solution to control hazards（作为解决方案以控制冒险），282
Pipelined branches（流水线分支），319
Pipelined control（流水线控制），见 Control，300-303
control lines（控制线），300，303
overview illustration（概述说明），316
specifying（指定），300
Pipelined datapaths（流水线数据通路），286-303
with connected control signals（连接的控制信号），304
with control signals（控制信号），300-303
corrected（纠正），296
illustrated（举例说明），289
in load instruction stages（加载指令阶段），296
Pipelined dependencies（流水线相关），305
Pipelines（流水线），
branch instruction impact（分支指令的影响），317
effectiveness improving（有效性改善），OL4.16-4-4.16-5
execute and address calculation stage（执行和地址计算阶段），290，292
five-stage（5 阶段），274，290，299
graphic representation（图示），279，296-300
instruction decode and register file read stage（指令译码和读寄存器文件阶段），289，292

instruction fetch stage（取指令阶段），290，292
instruction sequence（指令序列），313
latency（延迟），286
memory access stage（存储器访问阶段），290，292
multiple-clock-cycle diagrams（多时钟周期图），296-297
performance bottlenecks（性能瓶颈），343
single-clock-cycle diagrams（单时钟周期图），296-297
stages（阶段），274
static two-issue（静态双发射），335
write-back stage（写回阶段），290，294
Pipelining（流水线），12，272-286
advanced（先进/高级），343-344
benefits（收益），272
control hazards（控制冒险），281-282
data hazards（数据冒险），278
exceptions and（异常），327-332
execution time and（执行时间），286
fallacies（谬误），355-356
hazards（冒险），277-278
instruction set design for（指令集设计），277
laundry analogy（以洗衣为例），273
overview（概述），272-286
paradox（悖论），273
performance improvement（性能提高），277
pitfall（陷阱），355-356
simultaneous executing instructions（同时执行指令），286
speed-up formula（加速化公式），273
structural hazards（结构冒险），277，294
summary（总结），285
throughput and（吞吐量），286
pitfalls（陷阱），见 Fallacies
address space extension（地址空间扩展），479
arithmetic（算术），229-232
associativity（相联度），479
defined（定义），49
GPUs（图形处理单元），C-74-75
ignoring memory system behavior（忽略存储器系统行为），478
memory hierarchies（存储器层次），478-482
out-of-order processor evaluation（乱序处理器评估），479
performance equation subset（性能方程子集），50-51
pipelining（流水线），355-356
pointer to automatic variables（指向自动变量的指针），160
sequential word addresses（连续的字地址），160
simulating cache（模拟高速缓存），478
software development with multiprocessors（多处理器软件发展），556
VMM implementation（虚拟机实现），481，481-482
Pixel shader example（像素着色器阴影部分示例），C-15-17
Pixels（像素），18
Pointers（指针），
arrays versus（数组），141-145
frame（帧），103
global（全局），102
incrementing（递增），143
Java（Java），OL2.15-26
stack（栈），98，102
Polling（查询），OL6.9-8
Pop（弹出），98
Power（功率），
clock rate and（时钟速率），40
critical nature of（关键特性），53
efficiency（效率），343-344
relative（相对的），41
PowerPC
algebraic right shift（代数右移），E-33
branch registers（分支寄存器），E-32-33
condition codes（条件码），E-12
instructions（指令），E-12-13
instructions unique to（独有的指令），E-31-33
load multiple/store multiple（加载/存储多个），E-33
logical shifted immediate（立即数逻辑移位），E-33
rotate with mask（使用掩码移位），E-33
Precise interrupts（精准中断），332
Prediction（预测），12
2-bit scheme（2 位方法），322
accuracy（准确的），321，324
dynamic branch（动态分支），321-323
loops and（循环），321-323
steady-state（稳态），321
Prefetching（预取），482，544
Primitive types（基本数据类型），OL2.15-26
Procedure calls（过程调用），
convention（约定），A-22-33

examples（示例），A-27-33
frame（帧），A-23
preservation across（在……期间保持不变），102
Procedures（过程），96-106
compiling（编译），98
compiling showing nested procedure linking（编译显示嵌套过程链接），101-102
execution steps（执行步骤），96
frames（帧），103
leaf（叶子），100
nested（嵌套），100-102
recursive（递归），105，A-26-27
for setting arrays to zero（设置阵列为0），142
sort（排序），135-139
strcpy（字符串复制），108-109
string copy（字符串复制），108-109
swap（交换），133
Process identifiers（进程标识符），446
Process-level parallelism（进程级并行），500
Processors（处理器），242-356
as cores（作为核心），43
control（控制），19
datapath（数据通路），19
defined（定义），17，19
dynamic multiple-issue（动态多发射），333
multiple-issue（多发射），333
out-of-order execution（乱序执行），344，416
performance growth（性能提高），44
ROP（ROP），C-12，C-41
speculation（推测），333-334
static multiple-issue（静态多发射），333，334-339
streaming（流），C-34
superscalar（超标量），339，515-516，OL4.16-5
technologies for building（生成技术），24-28
two-issue（双发射），336-337
vector（矢量），508-510
VLIW（超长指令字），335
Product（乘积），183
Product of sums（和的乘积），B-11
Program counters（PC，程序计数器），251
changing with conditional branch（条件分支转换），324
defined（定义），98，251
exception（异常），445，447
incrementing（递增），251，253
instruction updates（指令更新），289
Program libraries（程序库），A-4
Program performance（程序性能），
elements affecting（元素影响），39
understanding（理解），9
Programmable array logic（PAL，可编程阵列逻辑），B-78
Programmable logic array（PLA，可编程逻辑阵列），
component dots illustration（组成点说明），B-16
control function implementation（控制功能实现），D-7，D-20-21
defined（定义），B-12
example（示例），B-13-14
illustrated（描述），B-13
ROMs and（只读存储器），B-15-16
size（尺寸），D-20
truth table implementation（真值表实现），B-13
Programmable logic device（PLD，可编程逻辑器件），B-78
Programmable ROM（PROM，可编程ROM），B-14
Programming languages（编程语言），见 specific languages
brief history of（简史），OL2.21-7-2.21-8
object-oriented（面向对象的），145
variables（变量），67
Programs（程序），
assembly language（汇编语言），123
Java starting（Java开始），131-132
parallel processing（并行处理），502-507
starting（开始），123-132
translating（翻译），123-132
Propagate（传播），
defined（定义），B-40
example（示例），B-44
super（超级），B-41
Protected keywords（受保护的关键字），OL2.15-21
Protection（保护）
defined（定义），428
implementing（实现），444-446
mechanisms（机制），OL5.17-9
VMs for（虚拟机），424
Protection group（保护组），OL5.11-5
Pseudo MIPS（伪MIPS），
defined（定义），233
instruction set（指令集），235
Pseudodirect addressing（伪直接寻址），116
Pseudoinstructions（伪指令），

defined（定义），124
summary（总结），125
Pthreads（POSIX thread，线程的一个标准），540
PTX instructions（PTX 指令），C-31，C-32
Public keywords（公共关键字），OL2.15-21
Push（压栈）
defined（定义），98
using（使用），100

## Q

Quad word（四字），154
Quicksort（快速排序），411，412
Quotient（商），189

## R

Race（竞争），B-73
Radix sort（基数排序），411，412，C-63-65
CUDA code（CUDA 代码），C-64
implementation（实现），C-63-65
RAID，见 Redundant arrays of inexpensive disks（RAID）
RAM（随机访问存储器），9
Raster operation（ROP）processors（光栅操作处理器），C-12，C-41，C-50-51
fixed function（固定功能），C-41
Raster refresh buffer（光栅刷新缓冲区），18
Rasterization（光栅化），C-50
Ray casting（RC）（光线投射（RC）），552
Read-only memories（ROM，只读存储器），B-14-16
control entries（控制项），D-16-17
control function encoding（控制功能编码），D-18-19
dispatch（分派），D-25
implementation（实现），D-15-19
logic function encoding（逻辑函数编码），B-15
overhead（开销），D-18
PLAs and（可编程逻辑阵列 PLA），B-15-16
programmable（PROM，可编程 ROM），B-14
total size（总容量），D-16
Read-stall cycles（读停顿周期），399
Read-write head（读写头），381
Receive message routine（接收消息例程），529
Receiver Control register（接收控制寄存器），A-39
Receiver Data register（接收数据寄存器），A-38，A-39
Recursive procedures（递归过程），见 Procedures，105，A-26-27.
clone invocation（克隆调用），100
stack in（准栈中），A-29-30
Reduced instruction set computer（RISC）
architecture（精简指令集计算机体系结构），见 Desktop and，E-2-45，OL2.21-5，OL4.16-4.
server RISCs；Embedded RISCs group types（组类型），E-3-4
instruction set lineage（指令集族系），E-44
Reduction（减少），519
Redundant arrays of inexpensive disks（RAID）（廉价冗余磁盘阵列），OL5.11-2-5.11-8
history（简史），OL5.11-8
RAID 0，OL5.11-4
RAID 1，OL5.11-5
RAID 2，OL5.11-5
RAID 3，OL5.11-5
RAID 4，OL5.11-5-5.11-6
RAID 5，OL5.11-6-5.11-7
RAID 6，OL5.11-7
spread of（传播），OL5.11-6
summary（总结），OL5.11-7-5.11-8
use statistics（使用统计），OL5.11-7
Reference bit（参考位），435
References（参考文献），
absolute（完全的），126
forward（转发/旁路），A-11
types（类型），OL2.15-26
unresolved（未决的），A-4，A-18
Register addressing（寄存器地址），116
Register allocation（寄存器分配），OL2.15-11-2.15-13
Register files（寄存器文件），B-50，B-54-56
defined（定义），252，B-50，B-54
in behavioral Verilog（Verilog 行为），B-57
single（单个），257
two read ports implementation（两个读端口的实现），B-55
with two read ports/one write port（两个读端口/一个写端口），B-55
write port implementation（写端口实现），B-56
Register-memory architecture（寄存器-存储架构），OL2.21-3

Registers（寄存器），152，153-154
architectural（结构），325-332
base（基于），69
callee-saved（被调用方保存），A-23
caller-saved（调用者保存），A-23
Cause（导致），A-35
clock cycle time and（时钟周期时间），67
compiling C assignment with（C 编译赋值），67-68
Count（计数），A-34
defined（定义），66
destination（目的），83，262
floating-point（浮点），217
left half（左半边），290
mapping（映射），80
MIPS conventions（MIPS 约定），105
number specification（指定号码），252
page table（页表），432
pipeline（流水线），308，309，312
primitives（基元），66
Receiver Control（接收控制），A-39
Receiver Data（接收数据），A-38，A-39
renaming（重命名），338
right half（右半边），290
spilling（溢出），71
Status（状态），327，A-35
temporary（暂时），67，99
Transmitter Control（发送控制），A-39-40
Transmitter Data（发送数据），A-40
usage convention（使用约定），A-24
use convention（使用约定），A-22
variables（变量），67
Relative performance（相对性能），31-32
Relative power（相对功率），41
Reliability（可靠性），418
Relocation information（重定位信息），A-13，A-14
Remainder（余数），
defined（定义），189
instructions（指令），A-55
Reorder buffers（重排序缓冲器），343
Replication（复制），468
Requested word first（请求字优先），392
Request-level parallelism（请求级并行），532
Reservation stations（保留站），
buffering operands in（缓冲操作数），340-341
defined（定义），339-340
Response time（响应时间），30-31
Restartable instructions（重启指令），448
Return address（返回地址），97
Return from exception（ERET，从异常中返回），445
R-format（R 格式），262
ALU operations（ALU 操作），253
defined（定义），83
Ripple carry（行波进位），
adder（加法器），B-29
carry lookahead speed versus（先行进位速度），B-46
Roofline model（顶层模型），542-543，544，545
with ceilings（天花板），546，547
computational roofline（可计算的屋顶轮廓线），545
illustrated（举例说明），542
Opteron generations（皓龙处理器时代），543，544
with overlapping areas shaded（阴影部分的重叠面积），547
peak floating-point performance（峰值浮点性能），542
peak memory performance（峰值存储性能），543
with two kernels（两个内核），547
Rotational delay（旋转延迟），见 Rotational latency
Rotational latency（旋转延迟），383
Rounding（舍入），218
accurate（精确），218
bits（位），220
with guard digits（带保护位的数字），219
IEEE 754 modes（IEEE754 格式），219
Row-major order（行优先顺序排列），217，413
R-type instructions（R 型指令），252
datapath for（数据通路），264-265
datapath in operation for（数据通路操作），266

## S

Saturation（饱和），181
sb（Store Byte，存储字节），64
sc（Store Conditional，存储条件），64
SCALAPAK（SCALAPAK），230
Scaling（规模），
strong（强缩放），505，507
weak（弱缩放），505
Scientific notation（科学计数法），
adding numbers in（加入数字），203
defined（定义），196

for reals（对实数），197
Search engines（搜索引擎），4
Secondary memory（辅助存储器），23
Sectors（扇区），381
Seek（寻找），382
Segmentation（分段），431
Selector values（选择符值），B-10
Semiconductors（半导体），25
Send message routine（发送消息例程），529
Sensitivity list（敏感值列表），B-24
Sequencers（序列发生器），
  explicit（显式），D-32
  implementing next-state function with（实现下一个状态的函数），D-22-28
Sequential logic（时序逻辑），B-5
Servers（服务器），见 Desktop and server RISC，OL5
  cost and capability（花费和能力），5
Service accomplishment（服务完成），418
Service interruption（服务中断），418
Set instructions（设置指令），93
Set-associative caches（组相连高速缓存），见 Caches，403
  address portions（高速缓存地址部分），407
  block replacement strategies（块替换策略），457
  choices of（选择），456
  four-way（四路），404，407
  memory-block location（存储器块位置），403
  miss（缺失），405-406
  n-way（n 路），405
  two-way（2 路），404
Setup time（建立时间），B-53，B-54
sh（Store Halfword，存储半字），64
Shaders（着色器）
  defined（定义），C-14
  floating-point arithmetic（浮点算术），C-14
  graphics（图像），C-14-15
  pixel example（像素示例），C-15-17
Shading languages（图形编程语言），C-14
Shadowing（阴影），OL5.11-5
Shared memory（共享存储器），见 Memory
  as low-latency memory（作为低延迟存储），C-21
  caching in（缓存中），C-58-60
  CUDA（CUDA），C-58
  N-body and（N 体），C-67-68
  per-CTA（Per-CTA），C-39
  SDRAM banks（SRAM 体），C-40
Shared memory multiprocessors（SMP，共享主存多处理器），517-521
  defined（定义），501，517
  single physical address space（单一物理地址空间），517
  synchronization（同步），518
Shift amount（移动数量），82
Shift instructions（移位指令），87，A-55-56
Sign and magnitude（符号和大小），197
Sign bit（符号位），76
Sign extension（符号扩展），254
  defined（定义），76
  shortcut（简便方法），78
Signals（信号），
  asserted（有效），250，B-4
  control（控制），250，263-264
  deasserted（无效），250，B-4
Signed division（有符号数除法），192-194
Signed multiplication（有符号数乘法），187
Signed numbers（有符号的数字），73-78
  sign and magnitude（符号和大小），75
  treating as unsigned（作为无符号数处理），94-95
Significands（有效数），198
  addition（另外的加法），203
  multiplication（乘），206
Silicon（硅），25
  as key hardware technology（作为关键硬件技术），53
  crystal ingot（晶锭），26
  defined（定义），26
  wafers（晶圆），26
Silicon crystal ingot（硅晶锭），26
SIMD（Single Instruction Multiple Data，单指令多数据），507-508，558
  computers（计算机），OL6.15-2-6.15-4
  data vector（数据向量），C-35
  extensions（扩展），OL6.15-4
  for loops and（for 循环），OL6.15-3
  massively parallel multiprocessors（大规模并行处理器），OL6.15-2
  small-scale（小规模），OL6.15-4
  vector architecture（向量结构），508-510
  in x86，508（在 x86 中）
SIMMs（single inline memory modules，单列直插式存储器模块），OL5.17-5，OL5.17-6
Simple programmable logic devices（SPLD，简单的

可编程逻辑器件), B-78
Simplicity (简化), 161
Simultaneous multithreading (SMT, 同时多线程), 515-517
  support (支持), 515
  thread-level parallelism (线程级并行), 517
  unused issue slots (未使用的发射槽), 515
Single error correcting/Double error correcting (SEC/DEC, 纠正单位/两位错误), 420-422
Single instruction single data (SISD, 单指令单数据), 507
Single precision (单精度), 见 Double precision
  binary representation (二进制表示), 201
  defined (定义), 198
Single-clock-cycle pipeline diagrams (单时钟周期流水图), 296-297
  illustrated (说明), 299
Single-cycle datapaths (单周期数据通路), 见 Datapaths
  illustrated (描述), 287
  instruction execution (指令执行), 288
Single-cycle implementation control function for (单周期执行控制功能), 269
  defined (定义), 270
  nonpipelined execution versus pipelined execution (非流水执行和流水执行), 276
  non-use of (未使用), 271-272
  penalty (代价), 271-272
  pipelined performance versus (流水线性能), 274
Single-instruction multiple-thread (SIMT, 单指令多线程), C-27-30
  overhead (开销), C-35
  multithreaded warp scheduling (多 warp 调度), C-28
  processor architecture (处理器架构), C-28
  warp execution and divergence (warp 执行和分支), C-29-30
Single-program multiple data (SPMD, 单程序多数据), C-22
sll (Shift Left Logical, 逻辑左移), 64
slt (Set Less Than, 小于则置位), 64
slti (Set Less Than Imm., 小于立即数置位), 64
sltiu (Set Less Than Imm. Unsigned, 小于无符号立即数则置位), 64
sltu (SetLess Than Unsig., 无符号数操作, 小于则置位), 64
Smalltalk-80 (Smalltalk-80), 01.2. 21-8
Smart phones (智能手机), 7
Snooping protocol (监听协议), 468-470
Snoopy cache coherence (基于监听的高速缓存一致性), 01.5. 12-7
Software optimization (软件优化)
  via blocking (通过阻塞), 413-418
Sort algorithms (排序算法), 141
Software (软件)
  layers (层), 13
  multiprocessor (多处理器), 500
  parallel (并行), 501
  as service (服务), 7, 532, 558
  systems (系统), 13
Sort procedure (排序过程), 见 Procedures, 135-139
  code for body (整体的代码), 135-137
  full procedure (全过程), 138-139
  passing parameters in (传递参数), 138
  preserving registers in (保持寄存器不变), 138
  procedure call (过程调用), 137
  register allocation for (寄存器分配), 135
Sorting performance (排序性能), C-54-55
Source files (源文件), A-4
Source language (源语言), A-6
Space allocation (空间分配)
  on heap (堆上的空间分配), 104-106
  on stack (堆栈中), 103
SPARC (可扩充处理器架构)
  annulling branch (取消分支), E-23
  CASA, E-31
  conditional branches (条件分支), E10- 12
  fast traps (快速陷阱), E-30
  floating-point operations (浮点操作), E-31
  Instructions (指令), E-29-32
  least significant bits (最低位), E-31
  multiple precision floating-point results (多精度浮点结果), E-32
  nonfaulting loads (无故障的装载), E-32
  overlapping integer operations (重叠的整数操作), E-31
  quadruple precision floating-point arithmetic (四倍精度浮点算术), E-32
  register windows (寄存器窗口), E-29-30
  support for LISP and Smalltalk (支持 LISP 和 Smalltalk), E-30
Sparse matrices (稀疏矩阵), C-55-58
Sparse Matrix-Vector multiply (SpMV, 稀疏矩阵向

量乘法), C-55, C-57, C-58
CUDAversion (CUDA 版本), C-57
serial code (串行代码), C-57
shared memory version (共享存储器版本), C-59
Spatial locality (空间局部性), 374
large block exploitation of (大块区的开发), 391
tendency (趋势), 378
SPEC, OL1.1.12-11-1.12-12
CPU benchmark (CPU 基准测试程序), 46-48
power benchmark (功率基准测试程序), 48-49
SPEC2000, OL1.12-12
SPEC2006, 233, OL1.12-12
SPEC89, OL1.12-11
SPEC92, OL1.12-12
SPEC95, OL1.12-12
SPECrate (SPEC 分值), 538-539
SPECratio (SPEC 比率), 47
Special function units (SFU, 特殊功能单元), C-35, C-50
defined (定义), C-43
Speculation (推测), 333-334
hardware-based (基于硬件的), 341
implementation (实现), 334
performance and (性能), 334
problems (问题), 334
recovery mechanism (恢复机制), 334
Speed-up challenge (提高速度的挑战), 503-505
balancing load (负载均衡), 505-506
bigger problem (更大的问题), 504-505
Spilling registers (溢出寄存器), 71, 98
SPIM (SPIM), A-40-45
byte order (字节顺序), A-43
features (特征), A-42-43
getting started with (从……开始), A-42
MIPS assembler directives support (MIPS 汇编指令的支持), A-47-49
speed (速度), A-41
system calls (系统调用), A-43-45
versions (版本), A-42
virtual machine simulation (虚拟机模拟), A-41-42
Split algorithm (分割算法), 552
Split caches (拆分缓存), 397
Squareroot instructions (平方根指令), A-79
sra (Shift Right Arith, 算术右移), A-56
srl (Shift Right Logical, 逻辑右移), 64
Stack architectures (堆栈架构), OL2.21-4
Stack pointers (堆栈指针)
adjustment (调整), 100
defined (定义), 98
values (值), 100
Stack segment (堆栈段), A-22
Stacks (堆栈)
allocating space on (分配栈空间), 103
fonarguments (参数), 140
defined (定义), 98
pop (弹出), 98
push (压入), 98, 100
recursive procedures (递归过程), A-29-30
Stalls (暂停), 280
as solution to control hazard (作为解决方案以控制冒险), 282
avoiding with code reordering (避免代码重排序), 280
behavioral Verilog with detection (Verilog 行为检测), OL4.13-6-4.13-8
data hazards and (数据冒险), 313-316
illustrations (举例说明), OL4.13-23, OL4.13-30
insertion into pipeline (插入流水线), 315
load-use (装载使用), 318
memory (存储器), 400
write-back scheme (回写方案), 399
write buffer (写缓冲区), 399
Standby spares (待机备用), OL5.11-8
State (状态)
in 2-bit prediction scheme (在 2 比特预测方案中的状态), 322
assignment (分配), B-70, D-27
bits (比特), D-8
exception saving/restoring (异常保存/恢复), 450
logic components (逻辑部件), 249
specification of (规范), 432
State elements (状态元素)
clock and (时钟), 250
combinational logic and (组合逻辑与), 250
defined (定义), 248, B-48
inputs (输入), 249
in storing/accessing instructions (存储/访问指令), 252
register file (寄存器文件), B-50
Static branch prediction (静态分支预测), 335
Static data (静态数据)

as dynamic data（动态数据），A-21
defined（定义），A-20
segment（段），104
Static multiple-issue processors（静态多发射处理器），见 Multiple issue，333，334-339
control hazards and（控制冒险），335-336
instruction sets（指令集），335
with MIPS ISA（MIPS 指令集体系结构），335-338
Static random access memories（SRAM，静态随机访问存储器），378，379，B-58-62
array organization（阵列组织），B-62
basic structure（基本结构），B-61
defined（定义），21，B-58
fixed access time（固定访问时间），B-58
large（大），B-59
read/write initiation（读/写启动），B-59
synchronous（SSRAM，同步），B-60
three-state buffers（三态缓冲器），B-59，B-60
Static variables（静态变量），102
Status register（状态寄存器），
fields（状态寄存器字段），A-34，A-35
Steady-state prediction（稳态预测），321
Sticky bits（粘结位），220
Store buffers（存储缓冲区），343
Store instructions（存储指令），见 Load instructions
access（访问），C-41
base register（基址寄存器），262
block（块），149
compiling with（用……进行编译），71
conditional（有条件的），122
defined（定义），71
details（细节），A-68-70
EX stage（执行阶段），294
floating-point（浮点），A-79
ID stage（译码阶段），291
IF stage（取指阶段），291
instruction dependency（指令相关），312
list of（列表），A-68-70
MEM stage（访存阶段），295
unit for implementing（实现单元），255
WB stage（写回阶段），295
Store word（存储字），71
Stored program concept（存储程序概念），63
as computer principle（计算机原理），86
illustrated（举例说明），86
principles（原理），161
Strcpy procedure（字符串复制过程），见 Procedures，108-109
as leaf procedure（作为叶子程序），109
pointers（指针），109
Stream benchmark（流基准测试程序），548
streaming multiprocessor（SM，流多处理器），C-48-49
Streaming processors array（SPA，流处理器阵列），C-34，C-49-50
Streaming SIMD Extension 2（SSE2，单指令多数据流扩展），C-41，C-46
floating-point architecture（浮点结构），224
Streaming SIMD Extensions（SSE）and advanced vector extensions in x86（单指令多数据流扩展和 x86 中的高级向量扩展），224-225
Stretch computer（可伸缩计算机），OL4.16-2
Strings（字符串），
defined（定义），107
in Java（在 Java 中），109-111
representation（表述），107
Strip mining（露天开采），510
Striping（条状化），OL5.11-4
Strong scaling（强缩放比例），505，517
Structural hazards（结构冒险），277，294
sub（Subtract，减法），64
sub.d（FP Subtract Double，双精度浮点减法），A-79
sub.s（FP Subtract Single，单精度浮点减法），A-80
Subnormals（低于正常的），222
Subtraction（减法），见 Arithmetic，178-182
binary（二进制），178-179
floating-point（浮点），211，A-79-80
instructions（指令），A-56-57
negative number（负数），179
overflow（溢出），179
subu（Subtract Unsigned，无符号减法），119
Subword parallelism（子字并行），222-223，352，E-17
and matrix multiply（矩阵乘法），225-228
Sum of products（乘积和），B-11，B-12
Supercomputers（超级计算机），OL4.16-3
defined（定义），5
SuperH（SuperH），E-15，E-39-40
Superscalars（超标量），
defined（定义），339，OL4.16-5
dynamic pipeline scheduling（动态流水线调

度），339
multithreading options（多线程选项），516
Surfaces（表面），C-41
sw（StoreWord，存储字），64
Swap procedure（交换过程），见 Procedures，133
body code（程序体代码），135
full（满），135，138-139
register allocation（寄存器分配），133
Swap space（交换空间），434
swc1（Store FP Single，存储单精度一的浮点数），A-73
Symbol tables（符号表），125，A-12，A-13
Synchronization（同步），121-123，552
barrier（障碍），C-18，C-20，C-34
defined（定义），518
lock（锁），121
overhead reducing（开销减少），44-45
unlock（未死锁），121
Synchronizers（同步器）
defined（定义），B-76
failure（失败），B-77
from D flip-flop（D 触发器），B-76
Synchronous DRAM（SDRAM，同步 DRAM），379-380，B-60，B-65
Synchronous SRAM（SSRAM，同步 SRAM），B-60
Synchronous system（同步系统），B-48
Syntax tree（语法树），OL2.15-3
System calls（系统调用），A-43-45
code（代码），A-43-44
defined（定义），445
loading（加载），A-43
Systems software（系统软件），13
System Verilog（一种建模语言）
cache controller（cache 控制器），OL5.12-2
cache data and tag modules（缓存数据和标签模块），OL5.12-6
FSM（有限状态机），OL5.12-7
simple cache block diagram（简单的高速缓存框图），OL5.12-4
type declarations（类型声明），OL5.12-2

T

Tablets（表），7
Tags（标志）
defined（定义），384
in locating block（定位块），407
page tables and（页表），434
size of（尺寸），409
Tail call（尾调用），105-106
Task identifiers（任务标识符），446
Task parallelism（任务并行），C-24
Task-level parallelism（任务级并行），500
Tebibyte（TiB，太字节），5
Telsa PTX ISA（Telsa PTX 指令集体系结构），C-31-34
arithmetic instructions（算术运算指令），C-33
barrier synchronization（障碍同步），C-34
GPU thread instructions（GPU 线程指令），C-32
memory access instructions（存储器访问指令），C-33-34
Temporal locality（时间局部性），374
tendency（趋势），378
Temporary registers（临时寄存器），67，99
Terabyte（TB，太字节），6
defined（定义），5
Text segment（文本段），A-13
Texture memory（纹理存储器），C-40
Texture/processor cluster（TPC，纹理/处理器簇），C-47-48
TFLOPS multiprocessor（万亿次每秒多处理器），OL6.15-6
Thrashing（系统颠簸），453
Thread blocks（线程块），528
creation（创建），C-23
defined（定义），C-19
managing（管理），C-30
memory sharing（存储器共享），C-20
synchronization（同步），C-20
Thread parallelism（线程并行），C-22
Treads（线程）
creation（创建），C-23
CUDA（CUDA），C-36
ISA（指令集体系结构），C-31-34
managing（管理），C-30
memory latencies and（存储器延迟），C-74-75
multiple per body（多重每个体），C-68-69
warps（warps），C-27
Three Cs model（3C 模式），459-461
Three-state buffers（三态缓冲寄存器），B-59，B-60
Throughput（吞吐量）

defined（定义），30-31
multiple issue and（多发射），342
pipelining and（流水线），286，342
Thumb（拇点），E-15，E-38
Timing（时间），
asynchronous inputs（异步输入），B-76-77
level-sensitive（电平敏感），B-75-76
methodologies（方法），B-72-77
two-phase（两相），B-75
TLB misses（TLB 缺失），见 Translation-look aside buffer（TLB），439
entry point（入口指针），449
handler（处理程序），449
handling（处理），446-453
occurrence（事件），446
problem（问题），453
Tomasulo's algorithm（Tomasulo 算法），OL4.16-3
Touchscreen（触摸屏），19
Tournament branch predicators（Tournament 分支预测器），324
Tracks（轨迹），381-382
Transfer time（传输时间），383
Transistors（晶体管），25
Translation-Lookaside buffer（TLB，地址转换缓冲），见 TLB misses，438-439，E-26-27，OL5.17-6
associativities（相关性），439
illustrated（举例说明），438
integration（集成），440-441
Intrinsity FastMATH（内建 FastMATH），440
typical values（典型值），439
Transmit driver and NIC hardware time versus receive driver and NIC hardware time（传输驱动和 NIC 硬件时间接收驱动和 NIC 硬件时间），OL6.9-8
Transmitter Control register（发送控制寄存器），A-39-40
Transmitter Data register（发送数据寄存器），A-40
Trap instructions（陷阱指令），A-64-66
Tree-based parallel scan（基于树的并行扫描），C-62
Truth tables（真值表），B-5
ALU control lines（ALU 控制线），D-5
for control bits（控制位），260-261
datapath control outputs（数据通路输出），D-17
datapath control signals（数据通路控制信号），D-14
defined（定义），260
example（例子），B-5
next-state output bits（下一状态输出位），D-15
PLA implementation（PLA 实现），B-13
Two's complement representation（二进制补码表示），75-76
advantage（优势），75-76
negation shortcut（求反的简便方法），76
rule（规则），79
sign extension shortcut（符号扩展），78
Two-level logic（两级逻辑），B-11-14
Two-phase clocking（两相时钟），B-75
TX-2 computer（TX-2 计算机），OL6.15-4

## U

Unconditional branches（无条件分支），91
Underflow（下溢），198
Unicode（同一的字符标准编码）
alphabets（字母表），109
defined（定义），110
example alphabets（字母表示例子），110
Unified GPU architecture（统一的 GPU 架构），C-10-12
illustrated（举例说明），C-11
processor array（处理器阵列），C-11-12
Uniform memory access（UMA，一致性存储器访问），518，C-9
multiprocessors（多处理器），519
Units（单位）
commit（提交确认），339-340，343
control（控制），247-248，259-261，D-4-8，D-10，D-12-13
defined（定义），219
floating point（浮点），219
hazard detection（冒险检测），313，314-315
for load/store implementation（装载/存储执行），255
special function（SFU，特殊功能），C-35，C-43，C-50
UNIVAC I（通用自动计算机一号），OL1.12-5
UNIX（一种多用户的计算机操作系统），OL2.21-8，OL5.17-9-5.17-12
AT&T（美国电话电报公司），OL5.17-10.
Berkeley version（BSD，Berkeley 版本），OL5.17-10
genius（天才品牌名），OL5.17-12
history（简史），OL5.17-9-5.17-12

Unlock synchronization（同步解锁），121
Unresolved references（未决的引用）
  defined（定义），A-4
  linkers and（链接器），A-18
Unsigned numbers（无符号数），73-78
Use latency（利用延迟时间）
  defined（定义），336-337
  one-instruction（一条指令），336-337

V

Vacuum tubes（真空管），25
Valid bit（有效位），386
Variables（变量）
  C language（C 语言），102
  programming language（程程语言），67
  register（寄存器），67
  static（静态），102
  storage class（存储类别），102
  type（类型），102
VAX architecture（VAX 体系结构），OL2.21-4，OL5.17-7
Vector lanes（向量线路），512
Vector processors（向量处理器），见 Processors，508-510
  conventional code comparison（传统代码比较），509-510
  instructions（指令），510
  multimedia extensions and（多媒体扩展），511-512
  scalar versus（标量），510-511
Vectored interrupts（向量中断），327
Verilog
  behavioral definition of MIPS ALU（MIPS ALU 的行为定义），B-25
  behavioral definition with bypassing（旁路行为的定义），OL4.13-4-4.13-6
  behavioral specification with stall for loads（加载风险的行为定义），OL4.13-6-4.13-8
  behavioral specification（行为定义），B-21，OL4.13-2-4.13-4
  behavioral specification of multicycle MIPS design（多周期 MIPS 行为定义的设计），OL4.13-12-4.13-13
  behavioral specification with simulation（行为定义仿真），OL4.13-2
  behavioral specification with stall detection（获得停顿的行为定义），OL4.13-6-4.13-8
  behavioral specification with synthesis（综合行为定义），OL4.13-11.-4.13-16
  blocking assignment（阻塞赋值），B-24
  branch hazard logic implementation（分支冒险逻辑实现），OL4.13-8-4.13-10
  combinational logic（组合逻辑），B-23-26
  datatypes（数据类型），B-21-22
  defined（定义），B-20
  forwarding implementation（转发的实现），OL4.13-4
  MIPS ALU definition in（MIPS ALU 定义），B-35-38
  modules（模块），B-23
  multicycle MIPS datapath（多周期 MIPS 数据通路），OL4.13-14
  nonblocking assignment（非阻塞赋值），B-24
  operators（运算符），B-22
  program structure（程序结构），B-23
  reg（寄存器），B-21-22
  sensitivity list（敏感列表），B-24
  sequential logic specification（时序逻辑定义），B-56-58
  structural specification（结构定义），B-21
  wire（线网），B-21-22
Vertical microcode（垂直微码），D-32
Very large-scale integrated（VLSI）circuits（超大规模集成电路），25
Very Long Instruction Word（VLIW，超长指令字）
  defined（定义），334-335
  first generation computers（第一代计算机），OL4.16-5
  processors（处理器），335
VHDL（一种硬件编程语言），B-20-21
Video graphics array（VGA）controllers（视频图形阵列监控器），C-3-4
Virtual addresses（虚拟地址）
  causing page faults（导致缺页），449
  defined（定义），428
  mapping from（映射），428-429
  size（尺寸），430
Virtual machine monitors（VMMs，虚拟机监控器）
  defined（定义），424
  implementing（实现），481，481-482
  laissez-faire attitude（不干涉的态度），481
  page tables（页表），452
  in performance improvement（在提高性能方

面），427
requirements（要求），426
Virtual machines（VM，虚拟器），424-427
benefits（收益），424
defined（定义），A-41
illusion（描述），452
instruction set architecture support（指令集体系结构支持），426-427
performance improvement（性能提高），427
for protection improvement（防护的改进），424
simulation of（模拟的），A-41-42
Virtual memory（虚拟存储器），见 Pages，427-454
address translation（地址转换），429，438-439
integration（集成），440-441
mechanism（机制），452-453
motivations（动机），427-428
page faults（缺页），428，434
protection implementation（保护的实现），444-446
segmentation（分段），431
summary（总结），452-453
virtualization of（虚拟化），452
writes（写），437
Virtualizable hardware（虚拟化硬件），426
Virtually addressed caches（虚地址缓存），443
Visual computing（视觉计算），C-3
Volatile memory（易失性存储器），22

## W

Wafers（晶元），26
defects（缺陷），26
dies（小片），26-27
yield（产量），27
Warehouse Scale Computers（WSC，仓储式计算机），7，531-533，558.
Warps（Warps），528，C-27
Weak scaling（弱缩放），505
Wear levelling（损耗均衡），381
While loops（while 循环），92-93
Whirlwind（旋风计算机），OL5.17-2
Wide area networks（WAN，广域网），见 Networks，24
Words（字）
accessing（访问），68
defined（定义），66
double（双两个），152
load（负载），68，71
quad（四四个），154
store（存储），71
Working set（工作集），453
World Wide Web（万维网），4
Worst-case delay（最坏情况下的延迟），272
Write buffers（写缓冲器）
defined（定义），394
stalls（栈），399
write-back cache（写回方式的高速缓存），395
Write invalidate protocols（写无效协议），468，469
Write serialization（系列写），467
Write-back caches（写回高速缓存），见 Caches
advantages（优点），458
cache coherency protocol（高速缓存一致性协议），OL5.12-5
complexity（复杂性），395
defined（定义），394，458
stalls（线），399
write buffers（写缓冲），395
Write-back stage（写回阶段）
control line（控制线），302
load instruction（加载指令），292
store instruction（存储指令），294
Writes（写）
complications（并发），394
expense（开销），453
handling（处理），393-395
memory hierarchy handling of（存储器层次的处理），457-458
schemes（方案），394
virtual memory（虚拟存储），437
write-back cache（写回式高速缓存），394，395
write-through cache（写直达高速缓存），394，395
Write-stall cycles（写停顿周期），400
Write-through caches（写直达高速缓存），见 Caches
advantages（优点），458
defined（定义），393，457
tag mismatch（标签不匹配），394

## X

x86（x86 计算机），149-158
Advanced Vector Extensions in（高级矢量扩展），225
brief history（简史），OL2.21-6
conclusion（结论），156-158
data addressing modes（数据寻址方式），152，

153-154
evolution（评估），149-152
first address specifier encoding（首地址符编码），158
historical timeline（历史时间表），149-152
instruction encoding（指令编码），155-156
instruction formats（指令格式），157
instruction set growth（指令集增长），161
instruction types（指令类型），153
integer operations（整数运算），152-155
registers（寄存器），152，153-154
SIMD in（单指令流多数据流），507-508，508
Streaming SIMD Extensions in（单指令多数据流扩展），224-225
typical instructions/functions（典型的指令/功能），155
typical operations（典型操作），157
Xerox Alto computer（Xerox Alto 计算机），C.13.12-8
XMM（扩充过的一种寄存器），224

## Y

Yahoo! Cloud Serving Benchmark（YCSB，云服务基准测试程序），540
Yield（收益），27
YMM（将寄存器宽度再次扩充的一种寄存器），225

## Z

Zettabyte（泽字节（十万亿亿字节）），6

# 推荐阅读




**深入理解计算机系统（第2版）**

作者：Randal E. Bryant David R. O'Hallaron
译者：龚奕利 雷迎春
中文版：978-7-111-32133-0，99.00元
英文版：978-7-111-32631-1，128.00元




**计算机系统概论（第2版）**

作者：Yale N. Patt Sanjay J. Patel
译者：梁阿磊 蒋兴昌 林凌
中文版：7-111-21556-1，49.00元
英文版：7-111-19766-6，66.00元




**数字设计和计算机体系结构（第2版）**

作者：David Harris Sarah Harris
译者：陈俊颖
英文版：978-7-111-44810-5，129.00元
中文版：2016年4月出版




**计算机系统：系统架构与操作系统的高度集成**

作者：Umakishore Ramachandran 等
译者：陈文光
英文版：978-7-111-31955-9，69.00元
中文版：978-7-111-50636-2，99.00元

# 推荐阅读




## 计算机体系结构：量化研究方法（英文版·第5版）

作者：John L. Hennessy 等 ISBN：978-7-111-36458-0 定价：138.00元

"本书之所以会成为不朽的经典之作，是因为每次再版都不仅仅是一次重新补充，而是全面修订，针对这个激动人心且快速变化的领域，给予及们最及时的信息和仅适的解读。即便对于我这样已从业二十多年的人来说，再次阅读本书时，依旧自我学无止境，感觉于两位卓越大师的渊博学识和深厚功底。"

—— Luiz Andre Barroso，Google公司

"本书是一部经典，犹如美酒，历久而弥醇。只有Hennessy和Patterson才可能接触到谷歌、亚马逊、微软等云计算与互联网规模的应用提供商的大部人士，对这一重要领域，业内的介绍材料无出其右。"

—— James Hamilton，Amazon Web服务

"《计算机体系结构：量化研究方法（第5版）》继续发扬传统，为学习计算机体系结构的学生提供了当前计算平台的最新信息，使他们能够洞悉体系结构，便于设计未来系统。这一版的亮点在于大幅修订了数据级并行和一章，用传统的体系结构术语清晰地解读了GPU体系结构。"

—— Krste Asanovi，加州大学伯克利分校

"本书已经成为一本经典教科书了，这一版突出介绍了各种形式并行技术（数据、线程、请求）的兴起，各用整整一章来描述。数据并行一章尤为夺目：通过向量SIMD、指令级SIMD和GPU的对比，避开每种体系结构的专用术语，揭示了这些体系结构之间的相似与区别。"

—— Kunle Olukotun，斯坦福大学

"这一版新增加了一章非常丰富的内容，用来讨论向量、SIMD和GPU体系结构中的数据级并行技术。它解释了应用于大众市场的GPU大部的关键体系结构概念，给出这些概念与传统术语的对应关系，并与向量和SIMD体系结构进行了对比。这一内容非常及时，与业内转向GPU并行计算的潮流相当应。《计算机体系结构：量化研究方法》继续独领风骚，全面地介绍了体系结构方面的重大新进展！"

—— John Nickolls，NVIDIA

## OPCODES, BASE CONVERSION, ASCII SYMBOLS

③

| MIPS (31:26) | (1) MIPS (5:0) | (2) MIPS (5:0) | Binary | Hexa-decimal | Decimal | ASCII Character | Decimal | Hexa-decimal | ASCII Character |
|---|---|---|---|---|---|---|---|---|---|
| (1) | | | 000 0000 | 0 | 0 | NUL | 64 | 40 | @ |
| | | | 000 0001 | 1 | 1 | SOH | 65 | 41 | A |
| | | | 000 0010 | 2 | 2 | STX | 66 | 42 | B |
| | | | 000 0011 | 3 | 3 | ETX | 67 | 43 | C |
| | | | 000 0100 | 4 | 4 | EOT | 68 | 44 | D |
| | | | 000 0101 | 5 | 5 | ENQ | 69 | 45 | E |
| | | | 000 0110 | 6 | 6 | ACK | 70 | 46 | F |
| | | | 000 0111 | 7 | 7 | BEL | 71 | 47 | G |
| | | | 000 1000 | 8 | 8 | BS | 72 | 48 | H |
| | | | 000 1001 | 9 | 9 | HT | 73 | 49 | I |
| | | | 000 1010 | 10 | a | LF | 74 | 4a | J |
| | | | 000 1011 | 11 | b | VT | 75 | 4b | K |
| | | | 000 1100 | 12 | c | FF | 76 | 4c | L |
| | | | 000 1101 | 13 | d | CR | 77 | 4d | M |
| | | | 000 1110 | 14 | e | SO | 78 | 4e | N |
| | | | 000 1111 | 15 | f | SI | 79 | 4f | O |
| | | | 001 0000 | 16 | 10 | DLE | 80 | 50 | P |
| (2) | | | 001 0001 | 17 | 11 | DC1 | 81 | 51 | Q |
| | | | 001 0010 | 18 | 12 | DC2 | 82 | 52 | R |
| | | | 001 0011 | 19 | 13 | DC3 | 83 | 53 | S |
| | | | 001 0100 | 20 | 14 | DC4 | 84 | 54 | T |
| | | | 001 0101 | 21 | 15 | NAK | 85 | 55 | U |
| | | | 001 0110 | 22 | 16 | SYN | 86 | 56 | V |
| | | | 001 0111 | 23 | 17 | ETB | 87 | 57 | W |
| | | | 001 1000 | 24 | 18 | CAN | 88 | 58 | X |
| | | | 001 1001 | 25 | 19 | EM | 89 | 59 | Y |
| | | | 001 1010 | 26 | 1a | SUB | 90 | 5a | Z |
| | | | 001 1011 | 27 | 1b | ESC | 91 | 5b | [ |
| | | | 001 1100 | 28 | 1c | FS | 92 | 5c | \ |
| | | | 001 1101 | 29 | 1d | GS | 93 | 5d | ] |
| | | | 001 1110 | 30 | 1e | RS | 94 | 5e | ^ |
| | | | 001 1111 | 31 | 1f | US | 95 | 5f | _ |
| | | | 010 0000 | 32 | 20 | Space | 96 | 60 | ` |
| | | | 010 0001 | 33 | 21 | ! | 97 | 61 | a |
| | | | 010 0010 | 34 | 22 | " | 98 | 62 | b |
| | | | 010 0011 | 35 | 23 | # | 99 | 63 | c |
| | | | 010 0100 | 36 | 24 | $ | 100 | 64 | d |
| | | | 010 0101 | 37 | 25 | % | 101 | 65 | e |
| | | | 010 0110 | 38 | 26 | & | 102 | 66 | f |
| | | | 010 0111 | 39 | 27 | ' | 103 | 67 | g |
| | | | 010 1000 | 40 | 28 | ( | 104 | 68 | h |
| | | | 010 1001 | 41 | 29 | ) | 105 | 69 | i |
| | | | 010 1010 | 42 | 2a | * | 106 | 6a | j |
| | | | 010 1011 | 43 | 2b | + | 107 | 6b | k |
| | | | 010 1100 | 44 | 2c | , | 108 | 6c | l |
| | | | 010 1101 | 45 | 2d | - | 109 | 6d | m |
| | | | 010 1110 | 46 | 2e | . | 110 | 6e | n |
| | | | 010 1111 | 47 | 2f | / | 111 | 6f | o |
| | | | 011 0000 | 48 | 30 | 0 | 112 | 70 | p |
| | | | 011 0001 | 49 | 31 | 1 | 113 | 71 | q |
| | | | 011 0010 | 50 | 32 | 2 | 114 | 72 | r |
| | | | 011 0011 | 51 | 33 | 3 | 115 | 73 | s |
| | | | 011 0100 | 52 | 34 | 4 | 116 | 74 | t |
| | | | 011 0101 | 53 | 35 | 5 | 117 | 75 | u |
| | | | 011 0110 | 54 | 36 | 6 | 118 | 76 | v |
| | | | 011 0111 | 55 | 37 | 7 | 119 | 77 | w |
| | | | 011 1000 | 56 | 38 | 8 | 120 | 78 | x |
| | | | 011 1001 | 57 | 39 | 9 | 121 | 79 | y |
| | | | 011 1010 | 58 | 3a | : | 122 | 7a | z |
| | | | 011 1011 | 59 | 3b | ; | 123 | 7b | { |
| | | | 011 1100 | 60 | 3c | < | 124 | 7c | \| |
| | | | 011 1101 | 61 | 3d | = | 125 | 7d | } |
| | | | 011 1110 | 62 | 3e | > | 126 | 7e | ~ |
| | | | 011 1111 | 63 | 3f | ? | 127 | 7f | DEL |

(1) opcode[31:26] = 0

(2) opcode[31:26] = $17_{ten}$ ($11_{two}$): if fmt[25:21] = $16_{ten}$ ($10_{two}$) r = s (single); if fmt[25:21] = $17_{ten}$ ($11_{two}$) r = d (double)

## IEEE 754 FLOATING-POINT STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

Where Single Precision Bias = 127, Double Precision Bias = 1023.

**IEEE Single Precision and Double Precision Formats:**



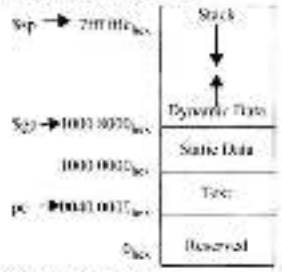

④

**IEEE 754 Symbols**

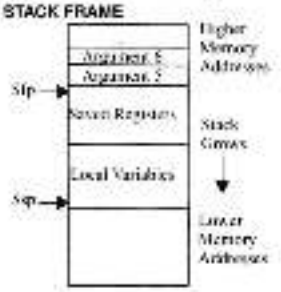
| Exponent | Fraction | Object |
|---|---|---|
| 0 | 0 | ± 0 |
| 0 | ≠0 | ± Denorm |
| 1 to MAX - 1 | anything | ± FL. Pt. Num. |
| MAX | 0 | ± ∞ |
| MAX | ≠0 | NaN |

S.P. MAX = 255, D.P. MAX = 2047

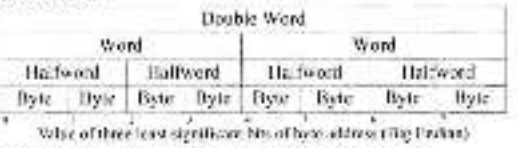
## MEMORY ALLOCATION




## STACK FRAME

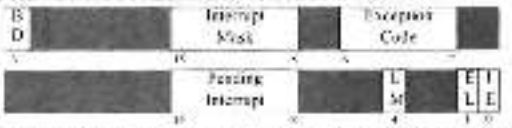



## DATA ALIGNMENT




Value of three least significant bits of byte address (Big Endian)

## EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS




BD = Branch Delay, UM = User Mode, EL = Exception Level, IE = Interrupt Enable

## EXCEPTION CODES

| Number | Name | Cause of Exception | Number | Name | Cause of Exception |
|---|---|---|---|---|---|
| 0 | Int | Interrupt (hardware) | 9 | Bp | Breakpoint Exception |
| 4 | AdEL | Address Error Exception (load or instruction fetch) | 10 | RI | Reserved Instruction Exception |
| 5 | AdES | Address Error Exception (store) | 11 | CpU | Coprocessor Unimplemented |
| 6 | IBE | Bus Error on Instruction Fetch | 12 | Ov | Arithmetic Overflow Exception |
| 7 | DBE | Bus Error on Load or Store | 13 | Tr | Trap |
| 8 | Sys | System Exception | 15 | FPE | Floating Point Exception |

## SIZE PREFIXES

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |

© 2014 by Elsevier, Inc. All rights reserved. From Patterson and Hennessy, *Computer Organization and Design*, 5th ed.



# 计算机组成与设计 硬件/软件接口 原书第5版

**Computer Organization and Design** The Hardware/Software Interface Fifth Edition

这是一本包括很多在当今计算机体系结构的教材。本书既涵盖历史背景，同方它不仅介绍了基本原理，摩尔定律，抽象，加速大概率事件，冗余，存储器层次，并行和流水线，也使用64ARM Cortex A8和Intel Core i7等现代设计师这些基本思想进行了说明。

——Mark D. Hill，威斯康星大学麦迪逊分校计算机科学系教授，系主任

教材的选择是一个非常令人困扰的决策过程。教学方法，习惯的遵循性，行文流畅的质量，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做折衷且涉及各个方面的书。它不仅是一部计算机经典的教科书，也是所有计算机科学系列书的范本。

—— Michael Goldweber，Xavier大学计算机科学系教授

当前，华为作为ICT行业的领先的国际是系统对系统的挑战，在移动系统和大数据处理等领域，我们与不断增长的长需求之间的差距越来越大。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的改变。在这个计算技术的新纪元，必须进行处理器和网设计，另外，系统软件系统结构优化与硬件级优化一样重要。

—— Christos Kozyrakis，斯坦福大学电子工程和计算机科学系副教授

本书是计算机组成与设计的经典畅销教材，第5版经过全面更新，关注后PC时代发生在计算机体系结构领域的革命性变革——从单核处理器到多核处理器，从串行到并行。本书特别关注移动计算和云计算，通过平板电脑、云体系结构以及ARM（移动计算设备）和x86（云计算）体系结构来探索和揭示这场技术变革。

与前几版一样，本书采用MIPS处理器讲解计算机硬件技术、汇编语言、计算机算术、流水线、存储器层次结构以及I/O等基本功能。

## 本书特色

- 更新例题、练习题和参考资料，重点关注移动计算和云计算这两个新领域。
- 涵盖从串行计算到并行计算的革命性变革，第6章专门介绍并行处理器，每章中都涉及并行硬件和软件的相关主题。
- 全书采用Intel Core i7、ARM Cortex A8和NVIDIA Fermi GPU作为实例。
- 增加“运行更快”这一新实例，说明正确理解硬件技术的重要性，它能使软件性能提高200倍。
- 讨论并强调计算机体系结构的“8个伟大思想”——通过并行提高性能、通过流水线提高性能、通过预测提高性能、面向摩尔定律的设计、存储器层次、使用抽象简化设计、加速大概率事件和通过冗余提高可靠性。
- 访问本书配套网站 | textbooks.elsevier.com/9780124077263 | 可获取更多资源，包括深入学习所需的高级内容、索引、术语表、参考文献以及推荐阅读材料。




本书译自原版*Computer Organization and Design: The Hardware/Software Interface, Fifth Edition*
并由Elsevier授权出版

ELSEVIER 爱思唯尔

上架指导：计算机/计算机组成

ISBN 978-7-111-50482-5

9 787111 504825 >

定价：99.00元

机械工业出版社：(010) 88379604
客服热线：(010) 88379561 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上书店：www.china-pub.com
数字服务：www.hzmedia.com.cn