计 算 机 科 学 丛 书

原书第4版

计算机组成与设计

硬件/软件接口

（美）David A. Patterson John L. Hennessy 著 康继昌 樊晓桠 安建峰 等译

Computer Organization and Design

The Hardware / Software Interface Fourth Edition



附光盘

机械工业出版社
China Machine Press

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

同以往版本一样，本书采用 MIPS 处理器作为展示计算机硬件技术、汇编语言、计算机算术、流水线、存储器层次结构以及 I/O 等基本功能的核心。书中强调了计算机从串行到并行的最新革新，在每章中都纳入了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。

本书适合作为高等院校相关专业的本科生和研究生教材，对广大技术人员也有很高的参考价值。

David A. Patterson and John L. Hennessy: Computer Organization and Design: The Hardware/Software Interface, Fourth Edition (ISBN 978-0-12-374493-7).

Copyright © 2009 by Elsevier Inc. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

ISBN: 978-981-272-338-3

Copyright © 2012 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 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.

本书简体中文版由机械工业出版社与 Elsevier (Singapore) Pte Ltd. 在中国大陆境内合作出版。本版仅限在中国境内（不包括中国香港、澳门特别行政区及中国台湾地区）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。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2009-3517

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 4 版）/（美）帕特森（Patterson, D. A.），（美）亨尼斯（Hennessy, J. L.）著；康继昌，樊晓桠，安建峰等译. —北京：机械工业出版社，2011.11

（计算机科学丛书）

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

ISBN 978-7-111-35305-8

Ⅰ. 计…　Ⅱ. ①帕…　②亨…　③康…　④樊…　⑤安…　Ⅲ. 计算机体系结构　Ⅳ. TP303

中国版本图书馆 CIP 数据核字（2011）第 136488 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：朱秀英

北京诚信伟业印刷有限公司印刷

2012 年 1 月第 1 版第 1 次印刷

185mm × 260mm · 34.5 印张

标准书号：ISBN 978-7-111-35305-8

ISBN 978-7-89433-047-5（光盘）

定价：99.00 元（附光盘）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

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

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

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

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

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

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



华章教育
华章科技图书出版中心

译者序

Computer Organization and Design: The Hardware/Software Interface, 4E

David A. Patterson 和 John L. Hennessy 是目前国际知名院校计算机专业领域的双巨擘。他们合著的《Computer Organization and Design: The Hardware/Software Interface》又发行了第4版。该书是他们对计算机组织研究和实践的全面而系统的总结。目前，世界上很多大学的计算机原理课程都采用这本教材，国内也有不少大学采用这本教材。

我们认为第4版最主要的特点是强调了计算机从串行到并行的最新变革。本版在每章中都强调了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。本版特别描述了一种评测多核性能的 Roofline 模型，使用 SPEC 2006 程序集更新了所有处理器的性能评测结果。此外，本版还首次描述了面向可视计算优化的高度多线程多处理器 GPU。

感谢清华大学郑纬民教授对前三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。

除封面署名之外，西北工业大学计算机学院的史莉雯、姚涛、任向隆、郑乔石、韩立敏等也参加了本书的翻译和校对工作。由于译者水平有限，文中肯定存在一些翻译不当或理解欠妥的地方，希望读者批评指正。

康继昌

2011 年 10 月于西北工业大学

神秘是一种我们能够体验到的最美丽的东西。
它是所有真正艺术和科学的源泉。

——阿尔伯特·爱因斯坦，《我的世界观》，1930

关于本书

我们认为，在学习计算机科学与工程时，除了掌握计算的基本原理外，还应该了解该领域当今的最新状态。我们也感觉到，计算领域中各种读者希望有机会欣赏到计算机系统的组织范例。后者决定了计算机系统的功能、性能，甚至成功与否。

现代计算机技术需要各种计算方面的专家，他们不仅能理解硬件，而且能理解软件。硬件和软件之间在许多层次上的相互关系，提供了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，计算机组成与设计中的中心思想是相同的。因而，本书着重于展示硬件与软件之间的相互关系，重点介绍概念，这是当今计算机的基础。

最近单处理器已发展为多核微处理器，这也印证了本书自第 1 版就预测的这一发展前景。有些程序员忽视了这一发展趋势，他们仍希望计算机体系结构专家、编译器编写者和芯片工程师能够帮助他们，让程序不作任何改进就可以更快地运行在新型处理器上。但是，这样的时代已经过去了。为了使程序更快地运行，必须将其并行化。程序员在编程时不用考虑硬件的并行特性，这一目标要很多年才能实现。我们认为，至少在下一个十年里，大多数程序员必须理解软硬件接口，才能使程序在并行计算机上有效地运行。

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，需要理解基本的计算机组成的读者；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，或要进一步理解计算机系统是如何工作的读者。

与本书相关的另一本书

有些读者可能已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”（本书则经常被称为“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。我们以商用产品为例，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一起工作的基本硬件技术的理解程度，将严重影响软件系统的性能和能效。因此，编译器编写者、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提出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应用软件的影响。

所以，本书的内容远多于“Hennessy and Patterson”，而且这些内容已大量修订，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，本书亦是如此。

⊖ 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》。

第 4 版的修订目的

第 4 版的修订目的包括：第一，描述微处理器的多核革命，全书将贯穿并行软硬件的思想；第二，梳理已有的内容以腾出篇幅介绍并行性；第三，从总体上提高教材水平；第四，更新技术内容，以反映自 2004 年第 3 版出版以来业界的新变化；第五，利用当今互联网时代的有利条件，提供了大量有用的练习题。

在详细介绍第 4 版的修订目的之前，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 7 章对两种读者是同样重要的。第 1 章更新了引言部分，增加了功耗重要性和由其引出的微处理器从单核转向多核的讨论，以及性能评价和基准测试程序的相关材料（这在第 3 版中是独立的一章）。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是重要的阅读材料，特别是想要深入学习编译器和面向对象语言的读者。它包括第 3 版中第 3 章的内容，介绍了完整的 MIPS 体系结构（浮点指令除外）。第 3 章适合对定点运算或者对浮点运算感兴趣的读者，有些人可能不需要学习第 3 章，可以跳过去。第 4 章是把第 3 版的两章合并起来介绍流水线处理器。其中，4.1 节、4.5 节和 4.10 节为关注软件的读者提供了流水线概述。关注硬件的读者将发现第 4 章提供了流水线处理器的核心技术，读者需要根据自己的专业背景，决定是否首先阅读附录 C 中提供的逻辑设计部分。第 5 章和第 6 章描述的存储器对关注软件的读者是极为重要的，如果时间允许，其他读者也应该尽量深入阅读。第 7 章介绍了多核、多处理器和集群，是业界最新的内容，每个人都应该阅读。

章/附录	节	关注软件	关注硬件
第 1 章　计算机概要与技术	1.1 ~ 1.9	①	①
	1.10（历史）	④	④
第 2 章　指令：计算机的语言	2.1 ~ 2.14	①	②
	2.15（编译器 & Java）	③	
	2.16 ~ 2.19	①	②
	2.20（历史）	④	④
附录 E　RISC 指令集体系结构	E.1 ~ E.19	③	
第 3 章　计算机的算术运算	3.1 ~ 3.9	②	②
	3.10（历史）	④	④
附录 C　逻辑设计基础	C.1 ~ C.13		②
第 4 章　处理器	4.1（引言）	①	①
	4.2（逻辑设计惯例）		①
	4.3 ~ 4.4（简单实现）	②	①
	4.5（流水线概述）	③	①
	4.6（流水线数据通路）	②	①
	4.7 ~ 4.9（冒险、异常）		①
	4.10 ~ 4.11（并行、实例）	①	①
	4.12（Verilog 流水线控制）		③
	4.13 ~ 4.14（谬误）	①	①
	4.15（历史）	④	④
附录 D　控制通路的硬件实现	D.1 ~ D.6		③
第 5 章　大容量和高速度：开发存储器层次结构	5.1 ~ 5.8	①	①
	5.9（实现 cache 控制器）		③
	5.10 ~ 5.12	①	①
	5.13（历史）	④	④

PDG

（续）

章/附录	节	关注软件	关注硬件
第6章　存储器和其他 I/O 主题	6.1～6.10	①	③
	6.11（网络）	③	③
	6.12～6.13	①	③
	6.14（历史）	④	④
第7章　多核、多处理器和集群	7.1～7.13	①	①
	7.14（历史）	④	④
附录 A　图形和计算 GPU	A.1～A.12	③	③
附录 B　汇编器、链接器和 SPIM 仿真器	B.1～B.12	⑤	⑤

仔细阅读：①　　有时间可读：③　　作为参考：⑤
回顾或阅读：②　　拓展阅读：④

第4版修订的第一个目的是使第3版位于光盘中作为单独一章的并行性成为本书最为重要的内容，其中最为明显的例子是第7章。特别需要说明的是，第7章引入了 Roofline（屋顶线）性能模型，并将之用于对4个新型多核体系结构的性能评价。评价结果表明屋顶线模型对于多核微处理器具有相当的洞察力，可以媲美于 cache 的 3C 模型。

在明确了并行性的重要地位之后，除了在第7章专门讲述并行之外，本版在前6章中的每一章都专门开辟一节强调了并行性。

- 1.6　沧海巨变：从单处理器向多处理器转变　指出功耗的限制如何迫使业界转向并行性以及并行性为什么是有益的。
- 2.11　并行与指令：同步　讨论了共享变量的加锁，尤其是 MIPS 的 Load Linked 和 Store Conditional 指令。
- 3.6　并行性和计算机算术：结合律　讨论了数值精度与浮点运算的挑战。
- 4.10　并行和高级指令级并行　讨论了各种高级指令级并行（ILP），包括超标量、推测和超长指令字（VLIW）、循环展开和乱序操作（OOO），同时也对流水线深度和功耗之间的关系进行了讨论。
- 5.8　并行与存储器层次结构：cache 一致性　讨论了 cache 一致性、连贯性和侦听协议等。
- 6.9　并行性与 I/O：廉价磁盘冗余阵列。将 RAID 描述成 I/O 系统和高效可用的 ICO 系统。

第7章总结了发展并行性的乐观理由，分析了为何本次并行性的发展应该比过去更加成功。

令我特别高兴的是，NVIDIA 的首席科学家 David Kirk 和首席架构师 John Nickolls 为本版撰写了关于图形处理器 GPU 的附录 A。GPU 是对计算机体系结构的一种新的、有趣的推动，附录 A 第一次对 GPU 进行了深入介绍。该附录基于本版的并行主题，提出了一种计算风格：允许程序员以多指令多数据（MIMD）的方式思考，然而硬件在任何可能的时候仍尽量以单指令多数据（SIMD）的风格执行。由于 GPU 价格便宜并且使用广泛——甚至在很多笔记本电脑中都可找到它们——并且它们的编程环境是免费可用的，所以它们提供了一个可用于许多人进行实验的并行硬件平台。

第二个目的是梳理该书，以便为介绍并行方面的新内容留出空间。第一步是简单地使用更细致的梳理方式对前三版累积下来的所有段落从前到后进行检查，看它们是否仍有在书中存在的需要。粗略的改变是章节的合并以及主题的舍弃。Mark Hill 建议舍弃书中多周期处理器的实现这部分内容，取而代之的是，在存储器层次的章节中增加有关多周期 cache 控制器的内容。这

使得处理器可以由单独的一章而不是两章来呈现，并且有关处理器的内容通过删除得到了加强。在第3版中单独作为一章的有关性能的内容，在本版中被合并到了第1章。

第三个目的是提升本书的教学方法。现在第1章变得更加充实，内容包括性能、集成电路、功耗，为全书奠定了基础。第2、3章原本以演进的风格进行编写，以“单细胞”的体系结构开始，并以第3章最后的完整MIPS体系结构结束。这种松散的写作风格不能很好地适应现代读者的需要。本版将所有的整型指令集归并到第2章，使第3章成为多数读者可以选读的内容，并且每节各自独立，读者不再需要阅读之前的所有节。因此，与之前版本相比，现在第2章是一个更好的参考资料。由于多周期实现会分散读者的注意力，而处理器现在变为单独的一章，所以第4章的编写效果更好。第5章新增了构建cache控制器的部分，此外，CD中新增的部分包含了该cache的Verilog代码。

第3版配有CD，使得书的页数减少，从而降低了书的价格。而且，有兴趣的读者还可更深入地阅读其中的参考资料。但是，我们积极减少页数的同时，读者却要过于频繁地在书和CD之间来回使用。本版中将不会出现这个问题。现在，CD中有每章的拓展阅读，以及四个章节的更加深入的材料。另外，所有练习都集中在书中，在书和CD之间进行交替使用的次数应该比较少了。

对于那些想知道为什么我们在本书中包含了CD的读者，答案也很简单：CD中包含了那些我们觉得无论读者在哪里都应该很容易并且即刻可以获得的内容。如果你对更进一步的内容感兴趣，或者你想复习一个VHDL教程（举例来说），它就在CD中，可供你使用。CD的另一个特点是，能极大地加强你对材料的学习：它包含了一个搜索引擎，使你可以搜索书中或CD本身的文本中的任何字符串。如果你正在寻找书的索引中没有包含的内容，你可以简单地输入你要搜索的文本和想要显示在搜索结果中的页码。这是一个非常有用的特点，我们希望当你阅读和回顾本书的时候，可以经常使用。

这是一个快速发展的领域，并且对于本书新的版本也是同样的情况，编写新版的一个重要的目的就是更新技术内容。AMD Opteron X4模型2356（代码为Barcelona）用来运行书中第1、4、5和7章的示例。第1、6章增加了SPEC中新的功耗测试程序的结果。第2章增加了ARM体系结构的部分，ARM是当前世界上最流行的32位指令集体系结构。第5章新增了一部分内容介绍虚拟机，其重要性再次呈现出来。第5章对Opteron X4多核的cache性能测量进行了详细的描述，以及对其竞争对手Intel Nehalem（将在本版书出版之后进行发布）的性能测量进行了一些细节描述。第6章第一次描述了Flash存储器，同时也对Sun公司的卓越的小型服务器进行了描述，它包含由8个核、16个DIMM和8个碟片组成的1Ubit的磁盘。第6章还描述了关于长期磁盘失效的最新研究结果。第7章涵盖了有关并行的大量话题，包括多线程、SIMD、向量、GPU、性能模型、测试程序和多处理器网络，并描述了Opteron X4额外的3个多核处理器：Intel Xeon模型e5345（Clovertown）、IBM Cell模型OS20和Sun微系统T2模型5120（Niagara 2）。

最后的目的是，在这个网络时代，尽量使习题对教师有用，因为布置家庭作业一直是学习资料的一个重要方式。然而，几乎是在本书出现的同时，习题答案就会立刻被贴出。对此，我们采取两种方式。首先，专家撰稿人一直在努力为书中的每一章编写全新的习题。第二，大多数习题都有一个含有多种可供替换的量化参数的表格，这些参数用于回答该问题，这种方式为练习题提供了量化描述支持。对指导教师如何选择布置练习而言，绝对的数量加上灵活性使得学生很难在线找到与习题对应的答案。指导教师还可以按照自己的意愿改变这些量化参数，有效阻止那些依赖互联网寻找固定不变的习题集答案的学生。我们认为这种新方法是对本书有价值的补充，无论你是学生还是教师，请让我们知道对于你来说它的效果如何。

我们保留了以往版本中有用的书本元素。为使本书更好地作为参考书，我们还在新术语第

一次出现的页的页脚放置了定义。书中标题为“理解程序性能”部分的内容用于帮助读者理解他们的程序性能，以及如何进行提高，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。“宏观图”部分仍然存在，以使读者看到整个“森林”而不是每一棵“树”。“小测验”部分通过在每章的最后提供答案，帮助读者在第一时间加强他们对内容的理解。本版同样提供了 MIPS 参考数据，这是从 IBM System/360 得到的灵感，并对可去掉的数据进行了更新，在编写 MIPS 汇编语言程序时，应该是一个方便的参考。

教学支持

我们已收集了大量材料供教师授课使用，包括题解、各章测验、本书的图表、讲义注解和幻灯片等，都可从出版商处获得。如需更多信息，请访问以下网址：

textbooks. elsevier. com/9780123744937

结语

从下面的致谢中，你会知道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：cod4bugs@ mkp. com。

本版标志了 Hennessy 和 Patterson 自 1989 年以来长期合作的中止。由于要管理一所世界知名大学，Hennessy 校长将不能继续实质性地承担新版本的编写工作。留下的作者感觉像个总是和伙伴一起演出的演员，突然被推到戏台上独自表演。所以，在致谢名单中列出的人和 Berkeley 的同行们在撰写本书的过程中甚至起了更大的作用。

第 4 版致谢

感谢 **David Kirk**、**John Nickolls** 和他们在 NVIDIA 的同事们（Michael Garland、John Montrym、Doug Voorhies、Lars Nyland、Erik Lindholm、Paulius Micikevicius、Massimiliano Fatica、Stuart Oberman 和 Vasily Volkov）提供了第一个深入介绍 GPU 的附录 A。再次感谢 Microsoft Research 的 **Jim Larus** 贡献了他在汇编语言方面的专长，并欢迎本书读者使用他开发并维护的仿真器。

也要感谢许多专家的贡献，他们为新版编写了大量新的练习题。写出好的练习题并不是一件容易的任务，在此感谢每位贡献者长期而艰苦地开发具有挑战性并吸引人的题目：

- 第 1 章：**Javier Bruguera**（Universidade de Santiago de Compostela）
- 第 2 章：**John Oliver**（Cal Poly，San Luis Obispo）、**Nicole Kaiyan**（University of Adelaide）和 **Milos Prvulovic**（Georgia Tech）
- 第 3 章：**Matthew Farrens**（University of California，Davis）
- 第 4 章：**Milos Prvulovic**（Georgia Tech）
- 第 5 章：**Jichuan Chang**、**Jacob Leverich**、**Kevin Lim** 和 **Partha Ranganathan**（均来自 Hewlett-Packard），以及 **Nicole Kaiyan**（University of Adelaide）
- 第 6 章：**Perry Alexander**（University of Kansas）
- 第 7 章：**David Kaeli**（Northeastern University）

感谢 **Peter Ashenden** 编辑和评价了所有的新习题，并完成本书的 CD 和新的幻灯片的制作。

感谢普林斯顿大学的 **David August** 和 **Prakash Prabhu** 提供了每章测验题。

感谢硅谷的同行们提供了大量新的技术数据：

- **AMD**——Opteron X4（Barcelona）的详细数据和测量数据：**William Brantley**、**Vasileios Liaskovitis**、**Chuck Moore** 和 **Brian Waldecker**。

- **Intel**——在 Intel Nehalem 上的预报信息：**Faye Briggs**。
- **Micron**——第 6 章中闪存的背景信息：**Dean Klein**。
- **Sun Microsystems**——第 2 章中 SPEC2006 基准测试程序的混合指令测量和第 6 章中 Sun Server x4150 的详细数据和测量数据：**Yan Fisher**、**John Fowler**、**Darryl Gove**、**Paul Joyce**、**Shenik Mehta**、**Pierre Reynes**、**Dimitry Stuve**、**Durgam Vahia** 和 **David Weaver**。
- **U. C. Berkeley**——**Krste Asanovic**（在第 7 章中提供了软件并发与硬件并行的思想），**James Demmel** 和 **Velvel Kahan**（有关并行性和浮点计算的注释），**Zhangxi Tan**（在第 5 章中设计了 cache 控制器及其 Verilog 程序），**Sam Williams**（在第 7 章中提供了屋顶线模型及其多核测量结果的数据），以及我在 **Par Lab** 中的所有同事，他们对全书的并行性主题给出了大量的建议和回馈。

感谢许多教师的贡献，他们回答出版商的问卷调查，评审我们的提议，出席小组会议，对第 4 版计划进行分析和回答。他们是中心组：Mark Hill（Wisconsin 大学，Madison），E. J. Kim（Texas A&M 大学），Jihong Kim（Seoul National 大学），Lu Peng（Louisiana 州立大学），Dean Tullsen（UC San Diego），Ken Vollmar（Missouri 州立大学），David Wood（Wisconsin 大学，Madison），Ki Hwan Yum（Texas 大学，San Antonio），以及评审：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 at Stony Brook），Mac Baker（Virginia Military Institute），Ron Barne（George Mason 大学），Douglas Blough（Georgia Institute of Technology），Kevin Bolding（Seattle Pacific 大学），Miodrag Bolic（Ottawa 大学），John Bonomo（Westminster College），Jeff Braun（Montana Tech），Tom Briggs（Shippensburg 大学），Scott Burgess（Humboldt 州立大学），Fazli Can（Bilkent 大学），Warren R. Carithers（Rochester Institute of Technology），Bruce Carlton（Mesa Community College），Nicholas Carter（Illinois 大学 at Urbana-Champaign），Anthony Cocchi（City 大学 of New York），Don Cooley（Utah 州立大学），Robert D. Cupper（Allegheny College），Edward W. Davis（North Carolina 州立大学），Nathaniel J. Davis（Air Force Institute of Technology），Molisa Derk（Oklahoma City 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（Memorial 大学 of Newfoundland），Michael Goldweber（Xavier 大学），Georgia Grant（College of San Mateo），Merrill Hall（The Master's College），Tyson Hall（Southern Adventist 大学），Ed Harcourt（Lawrence 大学），Justin E. Harlow（South Florida 大学），Paul F. Hemler（Hempden-Sydney College），Martin Herbordt（Boston 大学），Steve J. Hodges（Cabrillo College），Kenneth Hopkinson（Cornell 大学），Dalton Hunkins（St. Bonaventure 大学），Baback Izadi（州立大学 of New York—New Paltz），Reza Jafari，Robert W. Johnson（Colorado Technical 大学），Bharat Joshi（North Carolina 大学，Charlotte），Nargarajan Kandasamy（Drexel 大学），Rajiv Kapadia，Ryan Kastner（California 大学，Santa Barbara），Jim Kirk（Union 大学），Geoffrey S. Knauth（Lycoming College），Manish M. Kochhal（Wayne 州立大学），Suzan Koknar-Tezel（Saint Joseph's 大学），Angkul Kongmunvattana（Columbus 州立大学），April Kontostathis（Ursinus College），Christos Kozyrakis（Stanford 大学），Danny Krizanc（Wesleyan 大学），Ashok Kumar，S. Kumar（Texas 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，Duoglas W. Lynn（Oregon Institute of Technology），Yashwant K. Malaiya（Colorado 州立大学），Bill Mark（Texas 大学 at Austin），Annanda Mondal（Claflin 大学），Alvin Moser（Seattle 大学），Walid Najjar（California 大学，Riverside），Danial J. Neebel（Loras College），John Nestor（Lafayette College），Joe Oldham（Centre College），Timour Paltashev，James Parkerson（Ar-

kansas 大学），Shaunak Pawagi（SUNY at Stony Brook），Steve Pearce，Ted Pedersen（Minnesota 大学），Gregory D Peterson（Tennessee 大学），Dejan Raskovic（Alaska 大学，Fairbanks），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 College），Carolyn J. C. Schauble（Colorado 州立大学），Keith Schubert（CSU San Bernardino），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），Philip Snyder（Johns Hopkins 大学），Alex Sprintson（Texas A&M），Timothy D. Stanley（Brigham Young 大学），Dean Stevens（Morningside College），Norza 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），Rama Viswanathan（Beloit College），Guoping Wang（Indiana-Purdue 大学），Patricia Wenner（Bucknell 大学），Kent Wilken（California 大学，Davis），David Wolfe（Gustavus Adolphus College），David Wood（Wisconsin 大学，Madison），Mohamed Zahran（City College of New York），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 Berkeley 大学的相关人士，他们为本版最具挑战性的内容（第 7 章和附录 A）提供了大量回馈信息。他们是 **Krste Asanovic**、**Christopher Batten**、**Rastilav Bodik**、**Bryan Catanzaro**、**Jike Chong**、**Kaushik Data**、**Greg Giebling**、**Anik Jain**、**Jae Lee**、**Vasily Volkov** 和 **Samuel Williams**。

感谢 **Mark Smotherman** 一遍又一遍地反复寻找本书中的技术错误和写作错误，显著改进了这一版的写作质量。鉴于本版像是个人演出，他所发挥的作用就更为重要了。

感谢 Morgan Kaufmann 公司的 **Denise Penrose** 同意再次出版本书。**Nathaniel McFadden** 是本版的策划编辑，每周都与我讨论本书内容。**Kimberlee Honjo** 负责组织用户调查和回馈。

感谢 **Dawnmarie Simpson** 管理本书的出版过程，同时感谢对本版做出贡献的许多自由职业者，特别是 Multiscience 出版社的 Alan Rose 和 diacriTech 公司，为本书完成了排版。

以上提到的近 200 名人士为本版提供了大量帮助，使之成为我所希望的最好的书。

David A. Patterson

目 录

Computer Organization and Design: The Hardware/Software Interface, 4E

出版者的话
译者序
前言

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 你能从本书学到什么 …… 3
1.2 程序概念入门 …… 4
1.3 硬件概念入门 …… 7
1.3.1 剖析鼠标 …… 8
1.3.2 显示器 …… 8
1.3.3 打开机箱 …… 9
1.3.4 数据安全 …… 12
1.3.5 与其他计算机通信 …… 13
1.3.6 处理器和存储器制造技术 …… 14
1.4 性能 …… 15
1.4.1 性能的定义 …… 15
1.4.2 性能的测量 …… 17
1.4.3 CPU 性能及其因素 …… 18
1.4.4 指令的性能 …… 19
1.4.5 经典的 CPU 性能公式 …… 19
1.5 功耗墙 …… 21
1.6 沧海巨变：从单处理器向多处理器转变 …… 23
1.7 实例：制造以及 AMD Opteron X4 基准 … 25
1.7.1 SPEC CPU 基准测试程序 …… 27
1.7.2 SPEC 功耗基准测试程序 …… 28
1.8 谬误与陷阱 …… 29
1.9 本章小结 …… 31
1.10 拓展阅读 …… 32
1.11 练习题 …… 32
第 2 章 指令：计算机的语言 …… 42
2.1 引言 …… 42
2.2 计算机硬件的操作 …… 43
2.3 计算机硬件的操作数 …… 46
2.3.1 存储器操作数 …… 47
2.3.2 常数或立即数操作数 …… 49
2.4 有符号和无符号数 …… 50
2.5 计算机中指令的表示 …… 54
2.6 逻辑操作 …… 59
2.7 决策指令 …… 61
2.7.1 循环 …… 62
2.7.2 case/switch 语句 …… 64
2.8 计算机硬件对过程的支持 …… 65
2.8.1 使用更多的寄存器 …… 66
2.8.2 嵌套过程 …… 68
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 位立即数和地址的寻址 … 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 …… 84
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 …… 88
2.13 以一个 C 排序程序为例 …… 89
2.13.1 swap 过程 …… 89
2.13.2 sort 过程 …… 90
2.14 数组与指针 …… 95
2.14.1 用数组实现 clear …… 96
2.14.2 用指针实现 clear …… 96
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：ARM 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 …… 101
2.17.1 Intel x86 的改进 …… 101
2.17.2 x86 寄存器和数据寻址模式 …… 103
2.17.3 x86 整数操作 …… 104
2.17.4 x86 指令编码 …… 106
2.17.5 x86 总结 …… 107
2.18 谬误与陷阱 …… 107
2.19 本章小结 …… 108
2.20 拓展阅读 …… 110
2.21 练习题 …… 110
第 3 章 计算机的算术运算 …… 135
3.1 引言 …… 135
3.2 加法和减法 …… 135
3.2.1 多媒体算术运算 …… 137
3.2.2 小结 …… 138
3.3 乘法 …… 139
3.3.1 顺序的乘法算法和硬件 …… 139
3.3.2 有符号乘法 …… 141
3.3.3 更快速的乘法 …… 142
3.3.4 MIPS 中的乘法 …… 142
3.3.5 小结 …… 142
3.4 除法 …… 143
3.4.1 除法算法及其硬件结构 …… 143
3.4.2 有符号除法 …… 145
3.4.3 更快速的除法 …… 146
3.4.4 MIPS 中的除法 …… 146
3.4.5 小结 …… 147
3.5 浮点运算 …… 148
3.5.1 浮点表示 …… 149
3.5.2 浮点加法 …… 152
3.5.3 浮点乘法 …… 154
3.5.4 MIPS 中的浮点指令 …… 157
3.5.5 算术精确性 …… 162
3.5.6 小结 …… 164
3.6 并行性和计算机算术：结合律 …… 165
3.7 实例：x86 的浮点 …… 165
3.7.1 x86 浮点体系结构 …… 166
3.7.2 Intel SIMD 流扩展 2（SSE2）浮点体系结构 …… 167
3.8 谬误与陷阱 …… 168
3.9 本章小结 …… 170
3.10 拓展阅读 …… 172
3.11 练习题 …… 173
第 4 章 处理器 …… 182
4.1 引言 …… 182
4.1.1 一个基本的 MIPS 实现 …… 183
4.1.2 实现方式概述 …… 183
4.2 逻辑设计惯例 …… 185
4.3 建立数据通路 …… 187
4.4 一个简单的实现机制 …… 192
4.4.1 ALU 控制 …… 192
4.4.2 主控制单元的设计 …… 194
4.4.3 数据通路的操作 …… 197
4.4.4 控制的结束 …… 199
4.4.5 为什么不使用单周期实现方式 … 201
4.5 流水线概述 …… 202
4.5.1 面向流水线的指令集设计 …… 205
4.5.2 流水线冒险 …… 205
4.5.3 对流水线概述的小结 …… 210
4.6 流水线数据通路及其控制 …… 211
4.6.1 图形化表示的流水线 …… 219
4.6.2 流水线控制 …… 222
4.7 数据冒险：转发与阻塞 …… 225
4.8 控制冒险 …… 234
4.8.1 假定分支不发生 …… 234
4.8.2 缩短分支的延迟 …… 235
4.8.3 动态分支预测 …… 237
4.8.4 流水线小结 …… 239
4.9 异常 …… 240
4.9.1 异常在 MIPS 体系结构中的处理 … 241
4.9.2 在流水线实现中的异常 …… 242
4.10 并行和高级指令级并行 …… 245
4.10.1 推测的概念 …… 246
4.10.2 静态多发射处理器 …… 247
4.10.3 动态多发射处理器 …… 250
4.11 实例：AMD Opteron X4（Barcelona）流水线 …… 253
4.12 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 …… 255
4.13 谬误与陷阱 …… 255
4.14 本章小结 …… 256
4.15 拓展阅读 …… 257
4.16 练习题 …… 257

第 5 章　大容量和高速度：开发存储器层次结构 …… 280
5.1　引言 …… 280
5.2　cache 的基本原理 …… 283
5.2.1　cache 访问 …… 285
5.2.2　cache 缺失处理 …… 288
5.2.3　写操作处理 …… 289
5.2.4　一个 cache 的例子：内置 FastMATH 处理器 …… 290
5.2.5　设计支持 cache 的存储系统 …… 292
5.2.6　小结 …… 294
5.3　cache 性能的评估和改进 …… 295
5.3.1　通过更灵活地放置块来减少 cache 缺失 …… 297
5.3.2　在 cache 中查找一个块 …… 300
5.3.3　替换块的选择 …… 302
5.3.4　使用多级 cache 结构减少缺失代价 …… 302
5.3.5　小结 …… 305
5.4　虚拟存储器 …… 305
5.4.1　页的存放和查找 …… 308
5.4.2　缺页 …… 309
5.4.3　关于写 …… 312
5.4.4　加快地址转换：TLB …… 312
5.4.5　集成虚拟存储器、TLB 和 cache … 315
5.4.6　虚拟存储器中的保护 …… 317
5.4.7　处理 TLB 缺失和缺页 …… 318
5.4.8　小结 …… 322
5.5　存储器层次结构的一般架构 …… 323
5.5.1　问题 1：一个块可以被放在何处 … 323
5.5.2　问题 2：如何找到一个块 …… 324
5.5.3　问题 3：当 cache 缺失时替换哪一块 …… 325
5.5.4　问题 4：写操作如何处理 …… 325
5.5.5　3C：一种理解存储器层次结构行为的直观模型 …… 326
5.6　虚拟机 …… 328
5.6.1　虚拟机监视器的必备条件 …… 329
5.6.2　指令集系统结构（缺乏）对虚拟机的支持 …… 329
5.6.3　保护和指令集系统结构 …… 329
5.7　使用有限状态机来控制简单的 cache … 330
5.7.1　一个简单的 cache …… 330
5.7.2　有限状态机 …… 331
5.7.3　一个简单的 cache 控制器的有限状态机 …… 333
5.8　并行与存储器层次结构：cache 一致性 …… 334
5.8.1　实现一致性的基本方案 …… 335
5.8.2　监听协议 …… 335
5.9　高级内容：实现 cache 控制器 …… 336
5.10　实例：AMD Opteron X4（Barcelona）和 Intel Nehalem 的存储器层次结构 …… 337
5.10.1　Nehalem 和 Opteron 的存储器层次结构 …… 337
5.10.2　减少缺失代价的技术 …… 339
5.11　谬误和陷阱 …… 340
5.12　本章小结 …… 342
5.13　拓展阅读 …… 343
5.14　练习题 …… 343
第 6 章　存储器和其他 I/O 主题 …… 355
6.1　引言 …… 355
6.2　可信度、可靠性和可用性 …… 357
6.3　磁盘存储器 …… 359
6.4　快闪式存储器 …… 362
6.5　连接处理器、内存以及 I/O 设备 …… 363
6.5.1　互联基础 …… 364
6.5.2　x86 处理器的 I/O 互联 …… 365
6.6　为处理器、内存和操作系统提供 I/O 设备接口 …… 366
6.6.1　给 I/O 设备发送指令 …… 367
6.6.2　与处理器通信 …… 368
6.6.3　中断优先级 …… 369
6.6.4　在设备与内存之间传输数据 …… 370
6.6.5　直接存储器访问和内存系统 …… 371
6.7　I/O 性能度量：磁盘和文件系统的例子 …… 372
6.7.1　事务处理 I/O 基准程序 …… 372
6.7.2　文件系统和 Web I/O 的基准程序 …… 373
6.8　设计 I/O 系统 …… 373
6.9　并行性与 I/O：廉价磁盘冗余阵列 … 374
6.9.1　无冗余（RAID 0） …… 376
6.9.2　镜像（RAID 1） …… 376

6.9.3 错误检测和纠错码（RAID 2） … 376
6.9.4 位交叉奇偶校验（RAID 3） …… 376
6.9.5 块交叉奇偶校验（RAID 4） …… 376
6.9.6 分布式块交叉奇偶校验（RAID 5） …… 377
6.9.7 P+Q 冗余（RAID 6） …… 378
6.9.8 RAID 小结 …… 378
6.10 实例：Sun Fire x4150 服务器 …… 379
6.11 高级主题：网络 …… 383
6.12 谬误与陷阱 …… 383
6.13 本章小结 …… 386
6.14 拓展阅读 …… 387
6.15 练习题 …… 387
第 7 章 多核、多处理器和集群 …… 394
7.1 引言 …… 394
7.2 创建并行处理程序的难点 …… 396
7.3 共享存储多处理器 …… 398
7.4 集群和其他消息传递多处理器 …… 400
7.5 硬件多线程 …… 403
7.6 SISD、MIMD、SIMD、SPMD 和向量机 …… 404
7.6.1 在 x86 中的 SIMD：多媒体扩展 … 405
7.6.2 向量机 …… 406
7.6.3 向量与标量的对比 …… 407
7.6.4 向量与多媒体扩展的对比 …… 408
7.7 图形处理单元简介 …… 408
7.7.1 NVIDIA GPU 体系结构简介 …… 410
7.7.2 深入理解 GPU …… 411
7.8 多处理器网络拓扑简介 …… 412
7.9 多处理器基准测试程序 …… 415
7.10 Roofline：一个简单的性能模型 …… 417
7.10.1 Roofline 模型 …… 418
7.10.2 两代 Opteron 的比较 …… 419
7.11 实例：使用屋顶线模型评估四种多核处理器 …… 422
7.11.1 4 个多核系统 …… 422
7.11.2 稀疏矩阵 …… 424
7.11.3 结构化网格 …… 425
7.11.4 生产率 …… 426
7.12 谬误与陷阱 …… 427
7.13 本章小结 …… 428
7.14 拓展阅读 …… 429
7.15 练习题 …… 429
附录 A 图形和计算 GPU …… 439
A.1 引言 …… 439
A.1.1 GPU 发展简史 …… 439
A.1.2 异构系统 …… 440
A.1.3 GPU 发展成了可扩展的并行处理器 …… 440
A.1.4 为什么使用 CUDA 和 GPU 计算 …… 440
A.1.5 GPU 统一了图形和计算 …… 441
A.1.6 GPU 可视化计算的应用 …… 441
A.2 GPU 系统架构 …… 441
A.2.1 异构 CPU-GPU 系统架构 …… 442
A.2.2 GPU 接口和驱动 …… 443
A.2.3 图形逻辑流水线 …… 443
A.2.4 将图形流水线映射到统一的 GPU 处理器 …… 443
A.2.5 基本的统一 GPU 结构 …… 444
A.3 可编程 GPU …… 445
A.3.1 为实时图形编程 …… 446
A.3.2 逻辑图形流水线 …… 446
A.3.3 图形渲染程序 …… 447
A.3.4 像素渲染示例 …… 447
A.3.5 并行计算应用编程 …… 448
A.3.6 使用 CUDA 进行可扩展并行编程 …… 449
A.3.7 一些限制 …… 453
A.3.8 体系结构隐含的问题 …… 453
A.4 多线程的多处理器架构 …… 454
A.4.1 大规模多线程 …… 454
A.4.2 多处理器体系结构 …… 455
A.4.3 单指令多线程（SIMT） …… 456
A.4.4 SIMT warp 执行和分支 …… 457
A.4.5 管理线程和线程块 …… 457
A.4.6 线程指令 …… 458
A.4.7 指令集架构（ISA） …… 458
A.4.8 流处理器（SP） …… 461
A.4.9 特殊功能单元（SFU） …… 461
A.4.10 与其他多处理器的比较 …… 461
A.4.11 多线程多处理器总结 …… 462
A.5 并行存储系统 …… 462
A.5.1 DRAM 的考虑 …… 462

A.5.2 cache …… 463
A.5.3 MMU …… 463
A.5.4 存储器空间 …… 463
A.5.5 全局存储器 …… 463
A.5.6 共享存储器 …… 464
A.5.7 局部存储器 …… 464
A.5.8 常量存储器 …… 464
A.5.9 纹理存储器 …… 464
A.5.10 表面 …… 465
A.5.11 load/store 访问 …… 465
A.5.12 ROP …… 465
A.6 浮点算术 …… 465
A.6.1 支持的格式 …… 465
A.6.2 基本算术 …… 465
A.6.3 专用算术 …… 466
A.6.4 性能 …… 467
A.6.5 双精度 …… 467
A.7 资料：NVIDIA GeForce 8800 …… 468
A.7.1 流处理器阵列（SPA） …… 468
A.7.2 纹理/处理器簇（TPC） …… 469
A.7.3 流多处理器（SM） …… 470
A.7.4 指令集 …… 471
A.7.5 流处理器（SP） …… 471
A.7.6 特殊功能单元（SFU） …… 471
A.7.7 光栅化 …… 471
A.7.8 光栅操作处理器（ROP）和存储系统 …… 471
A.7.9 可扩展性 …… 472
A.7.10 性能 …… 472
A.7.11 密集线性代数性能 …… 472
A.7.12 FFT 性能 …… 473
A.7.13 排序性能 …… 474
A.8 资料：将应用映射到 GPU …… 474
A.8.1 稀疏矩阵 …… 475
A.8.2 在共享存储器中进行缓存 …… 477
A.8.3 扫描和归约 …… 478
A.8.4 基数排序 …… 480
A.8.5 GPU 上的 N-Body 应用 …… 482
A.9 谬误与陷阱 …… 486
A.10 小结 …… 489
A.11 拓展阅读 …… 489
附录 B 汇编器、链接器和 SPIM 仿真器 …… 490
B.1 引言 …… 490
B.1.1 什么时候使用汇编语言 …… 493
B.1.2 汇编语言的缺点 …… 493
B.2 汇编器 …… 494
B.2.1 目标文件的格式 …… 495
B.2.2 附加工具 …… 496
B.3 链接器 …… 498
B.4 加载 …… 499
B.5 内存的使用 …… 499
B.6 过程调用规范 …… 500
B.6.1 过程调用 …… 502
B.6.2 过程调用举例 …… 503
B.6.3 另外一个过程调用的例子 …… 505
B.7 异常和中断 …… 507
B.8 输入和输出 …… 509
B.9 SPIM …… 511
B.10 MIPS R2000 汇编语言 …… 513
B.10.1 寻址方式 …… 514
B.10.2 汇编语法 …… 515
B.10.3 MIPS 指令编码 …… 515
B.10.4 指令格式 …… 516
B.10.5 常数操作指令 …… 520
B.10.6 比较指令 …… 520
B.10.7 分支指令 …… 521
B.10.8 跳转指令 …… 523
B.10.9 陷阱指令 …… 523
B.10.10 取数指令 …… 525
B.10.11 保存指令 …… 526
B.10.12 数据传送指令 …… 527
B.10.13 浮点运算指令 …… 528
B.10.14 异常和中断指令 …… 532
B.11 小结 …… 533
B.12 参考文献 …… 533
B.13 练习题 …… 533

第1章

Computer Organization and Design: The Hardware/Software Interface, 4E

计算机概要与技术

我们可以完成更多重要的操作而不必考虑其完成的过程，这促进了文明的进步。

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

1.1 引言

欢迎阅读本书！非常高兴有这样的机会和大家一起共享令人兴奋的计算机系统世界。这并不是一个枯燥无味的领域，其进步不像冰河期那般漫长，新思想也不会因为受到忽视而萎缩。事实上，计算机是一种令人难以置信的、激动人心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%，并按摩尔定律一直持续增长。在过去的二十多年里，出现了许多导致计算产业革命的新型计算机，但是这些计算机很快就被更好的计算机所取代。

电子计算自20世纪40年代后期出现以来，其创新性的竞争导致了史无前例的进步。如果运输业的发展速度也像计算器工业那样快，那么今天我们从纽约到伦敦的旅行时间只需1秒钟，花费只有几美分。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。

计算机已导致人类文明的第三次革命——信息革命，它是沿着农业革命、工业革命的发展方向产生的。信息革命导致了人类能力和智慧的成倍增长，自然而深刻地影响着我们的日常生活，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。原本经济上不可行的应用，突然变得可行了。例如，下述的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，用计算机控制汽车发动机是极为普遍的应用，车载计算机不仅改进了燃油效率，减轻了污染，还通过防险刹车和安全气囊实现了撞车保护。
- *手机* 谁曾想到计算机系统的发展会产生移动电话，让人们几乎在全世界的任何地方都可以自由通信。
- *人类基因项目* 目前用于绘图和分析人类基因序列的计算机设备价值达几亿美元，这在10多年前就更加昂贵了。然而，随着计算机设备价格的持续下降，有望在未来实现按个人基因序列来治疗疾病。
- *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。
- *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，诸如虚拟世界、实用级别的语音识别和个性化保健等技术正在向我们走来。

1.1.1　计算应用的分类及其特性

计算机的应用领域十分广阔，从智能家电到手机，再到最大型的超级计算机。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：

桌面计算机㊀也许是最为人所知的应用方式，其典型代表为个人计算机，本书的读者几乎都在大量使用。桌面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 30 多年，但是已经带来大量新的计算技术革新。

服务器㊁是过去被称为大型机、小型机或超级计算机的现代形式，通常借助网络访问。服务器面向大型用户，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（例如数据库或仿真软件）的软件，并且往往为了特别的需要而加以修改或定制。服务器的制造技术和桌面计算机差不多，但能够提供更强的计算或 I/O 能力。服务器的设计通常强调可靠性，因为它的当机开销要比单用户的桌面计算机大得多。

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一千美元，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**㊂，一般由成百上千台处理器组成，内存为 **terabyte**㊃级，外存为 **petabyte**㊄级，价格从几百万至几亿美元不等。它们主要用于高端科学和工程计算，如天气预报、石油勘探和蛋白质结构计算等大规模问题。

诸如 eBay 和 Google 等公司所使用的互联网**数据中心**㊅也包含数以千计的处理器、terabyte 级的内存和 petabyte 级的外存，尽管它们一般不称为超级计算机。此类数据中心一般由大量计算机集群（见第 7 章）构成。

嵌入式计算机㊆是数量最多的一类，应用和性能范围十分广泛。包括在汽车、手机、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。

图 1-1 显示，最近几年中移动电话对嵌入式计算机的需求增长要比桌面计算机快得多。应该注意的是，除此之外，嵌入式计算机还大量应用于数字电视、机顶盒、汽车、数码相机、音乐播放器、视频游戏机等消费品，这些应用更加拉大了嵌入式计算机和桌面计算机之间数量需求的差距。

面向单一应用需求的嵌入式应用通常被严格限制其成本或功耗。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，除此以外降低成本和功耗是最大的目标。除了低成本的要求之外，嵌入式计算机对故障非常敏感，因为故障可能会让使用者心烦意乱（例如新电视机无法正常收看节目），也可能会导致安全事故（例如飞机失事）。在面向消费者的嵌入式应用中，如数字家电，可靠性是通过设计的简单性获得的——其重点在于尽可能地保证一项功能的正常

㊀ 桌面计算机（desktop computer）：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。
㊁ 服务器（server）：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。
㊂ 超级计算机（supercomputer）：具有最高性能和最贵成本的一类计算机，一般配置为服务器，需要花费数百万美元。
㊃ 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。
㊄ petabyte：1000TB 或 1024TB，视具体情况而定。
㊅ 数据中心（datacenter）：可满足大量服务器用电、散热和网络需求的房间或建筑物。
㊆ 嵌入式计算机（*embedded computer*）：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。

运转。而在大型嵌入式系统中，采用了多种冗余技术（见 6.9 节）。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。

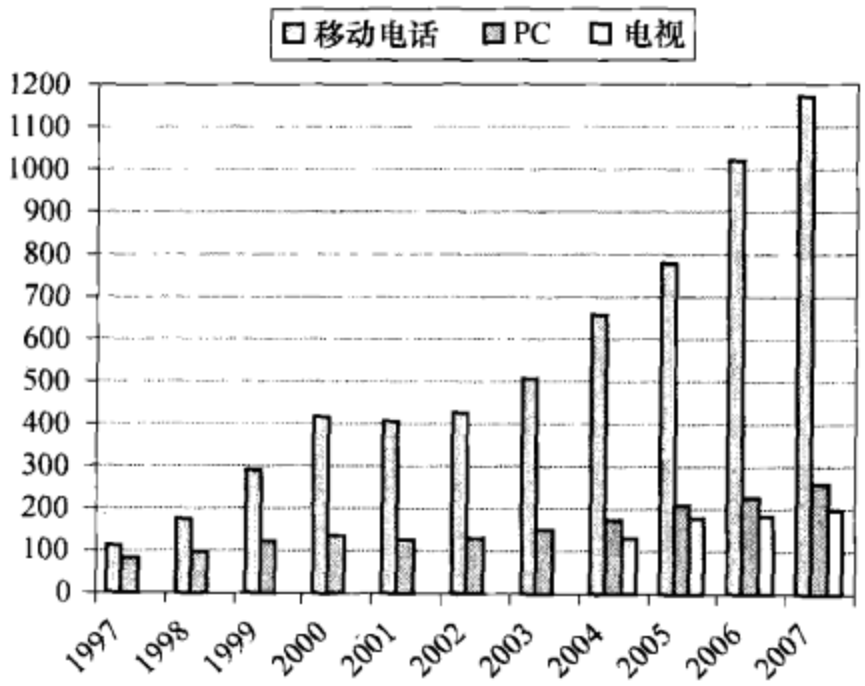


图 1-1　1997 ~ 2007 年之间每年生产的移动电话、PC 和电视的数量（电视数量从 2004 年开始统计）

在 2006 年，交付了超过十亿部新的移动电话。在 1997 年，移动电话的销量仅为 PC 机的 1.4 倍，但是在 2007 年达到了 4.5 倍。在 2004 年，使用中的电视约为 20 亿台，移动电话约为 18 亿部，PC 机约为 8 亿台。而在 2004 年世界人口约为 64 亿，相当于每 8 个人就有 1 台 PC 机、2.2 部移动电话和 2.5 台电视机。在 2006 年对美国家庭的一项调查中发现，平均每个家庭拥有 12 台家电，包括 3 台电视机、2 台 PC 机以及其他电器，如游戏终端、MP3 播放器和移动电话等。

精解：本书中的精解是正文中的一个小节，主要用来对读者可能感兴趣的内容做深入介绍。对此部分不感兴趣的读者可以直接跳过，因为它并不影响后续内容的学习。

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 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　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。因此，程序员为了创建高性能的编译器、操作系统、数据库以至应用程序，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》。
- 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[⊖]（见第 7 章）。

⊖ 多核微处理器（multicore microprocessor）：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，那么就无从改进你的程序在现代计算机上的性能，也无法评价各种计算机在解决特定问题时的优劣。

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等。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。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**[⊖]，一旦熟悉了就很容易理解。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，我们会在页底给出明确的定义，这样你很快会对这些定义熟悉起来。

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”小节，来对程序性能的理解加以概括。下面就是第一个：

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。

硬件或软件部件	如何影响性能	何处介绍
算法	决定源代码的行数和执行 I/O 操作的数量	参见其他书
编程语言、编译程序和体系结构	决定每行源代码对应机器指令的数量	第 2、3 章
处理器和存储器系统	决定指令执行的速度快慢	第 4、5、7 章
I/O 系统（硬件和操作系统）	决定 I/O 操作执行的速度快慢	第 6 章

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。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尾找到。所有小测验只在节尾出现，如果你确信自己对该部分内容完全理解，则可以跳过去。

1）1.1 节指出，每年嵌入式处理器的售出数量远远超过桌面处理器的数量。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。请思考下述的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

1.2　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，《The Innocents Abroad》（《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，可以由数百万行代码构成，并依靠软件

⊖ 首字母缩略词（acronym）：由一串单词中每个单词的首字母相连构成的单词。

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，才能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。

图 1-2 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**[㊀]位于两者之间。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**[㊁]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、MacOS 和 Windows 等。



图 1-2　简化的硬件和软件层次图，将硬件作为同心圆的中心，应用程序软件作为最外层

在复杂的应用中，通常存在多层应用软件层。例如，一个数据库系统可运行于系统软件之上，而驻留在该系统软件上的某应用又反过来运行在该数据库之上。

编译程序[㊂]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 B 将作深入介绍。

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。代表这些两个字母的符号是 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**一位**[㊃]。计算机服从于我们的命令，即计算机术语中的**指令**[㊄]。指令是能被计算机识别并执行的位串，可以被视为数字。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。

第一代程序员是直接使用二进制数与计算机通信的，这是一项非常乏味的工作。所以他们很快发明了助记符，以符合人类的思维方式。最初助记符是手工翻译成二进制的，其过程显然过于烦琐。随后设计人员开发了一种称为**汇编程序**[㊅]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
```

㊀ 系统软件（systems software）：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。
㊁ 操作系统（operating system）：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。
㊂ 编译程序（compiler）：将高级语言翻译为计算机所能识别的机器语言的程序。
㊃ 位（binary digit 或 bit）：基 2 数字中的 0 或 1，它是信息的基本组成元素。
㊄ 指令（instruction）：计算机硬件所能理解并服从的命令。
㊅ 汇编程序（assembler）：将指令由助记符形式翻译成二进制形式的程序。

汇编程序会将该符号翻译成

```
1000110010100000
```

该指令告诉计算机将两个数 A 和 B 相加。这种符号语言的名称今天还在用，即**汇编语言**[⊖]。而机器可以理解的二进制语言，是**机器语言**[⊜]。

虽然这是一个巨大的进步，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。

认识到程序编程需要更强大的高级语言是计算机早期的一个重大突破。**高级编程语言**[⊝]及其编译程序大大地提高了软件的生产率。图 1-3 表示了这些程序和编程语言之间的关系。

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 2 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 1-3）。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。编程语言使用较少行数即可表示出设计用意。简明性是高级语言相对汇编语言最为明显的优势。

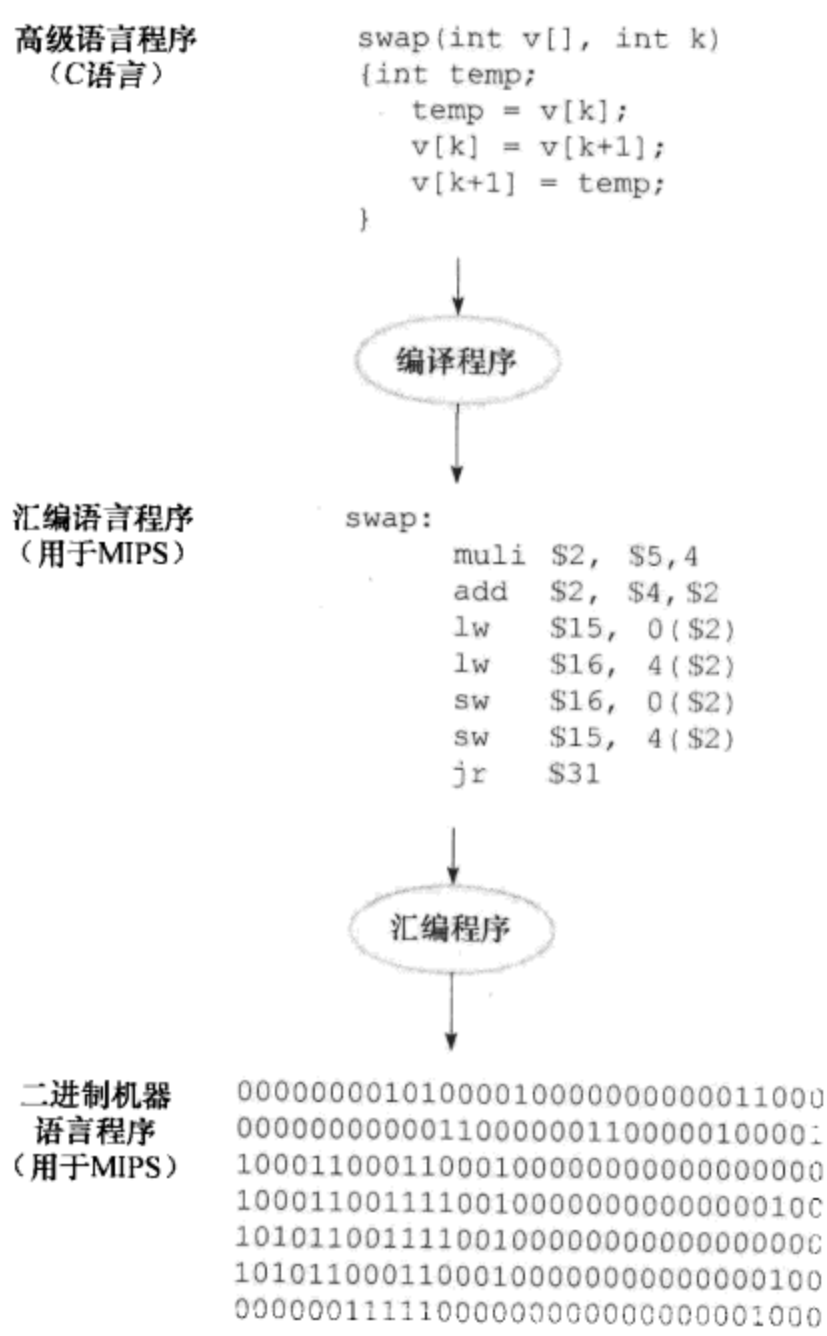


图 1-3　C 程序编译为汇编程序，再汇编为二进制机器语言

尽管将高级语言翻译成二进制的机器语言在上图仅需要两步，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。

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，使其直到今天仍被广泛应用。

⊖ 汇编语言（assembly language）：以助记符形式表示的机器指令。

⊜ 机器语言（machine language）：以二进制元形式表示的机器指令。

⊝ 高级编程语言（high-level programming language）：诸如 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。

1.3　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四项任务。

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。全书大致有十多个重要知识点，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五个部件构成的。

重点

组成计算机的五个典型部件是输入、输出、存储器、运算器和控制器，其中最后两个部件通常合称为处理器。图 1-4 表示了一台计算机的标准组成。该组成与硬件技术无关，你总能够把任何计算机（无论是现在的还是过去的）中的任何部件归于这五种之一。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。

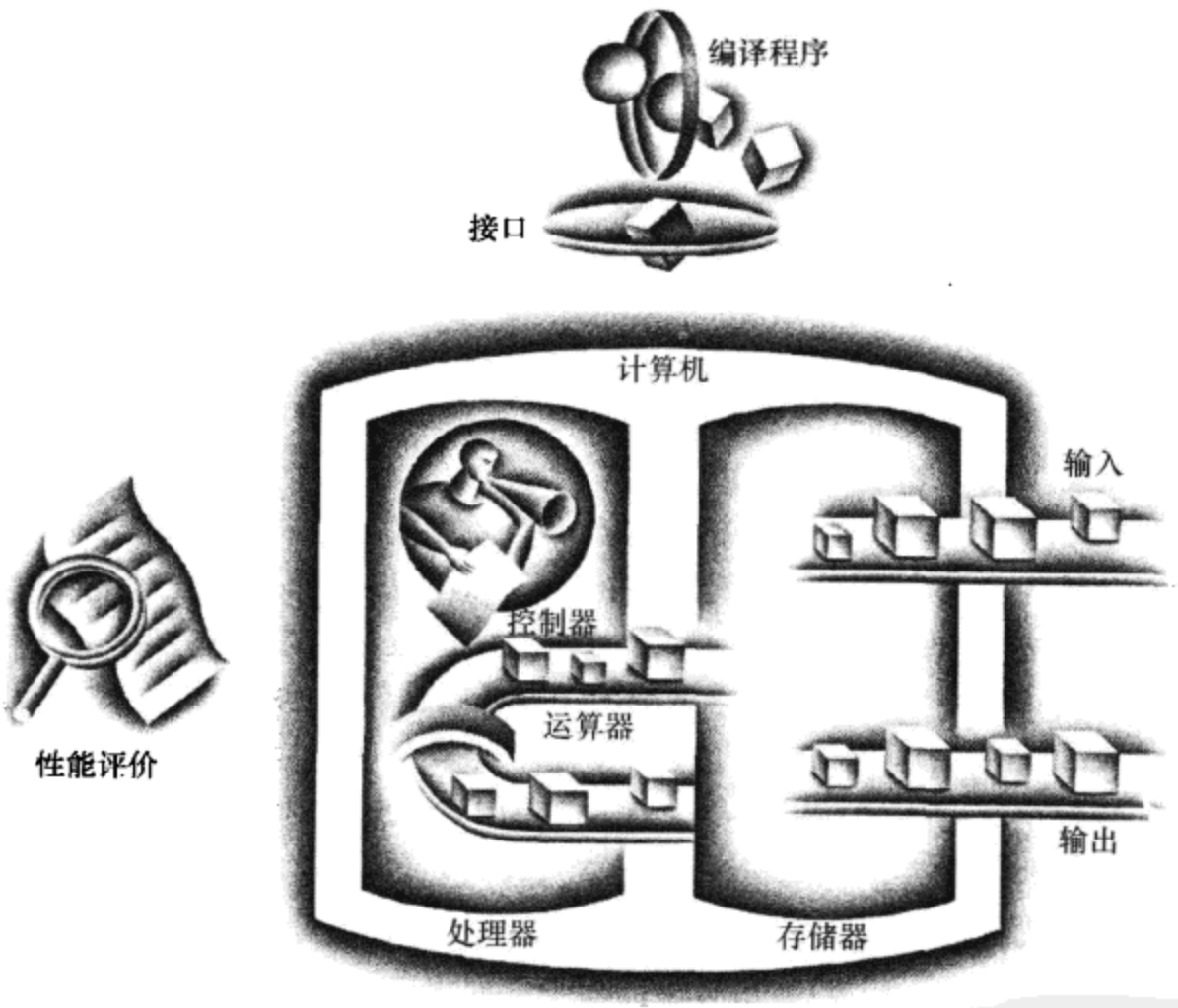


图 1-4　组成计算机的五个典型部件

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，控制器向运算器、存储器、输入和输出部件发出命令信号。

图 1-5 给出了一台计算机的照片，它带有键盘、无线鼠标和显示器。该照片展示了组成计算机的两个关键部件：**输入设备**[⊖]，如键盘和鼠标；**输出设备**[⊜]，如显示器。输入是为计算机提供数据，输出则是将计算结果反馈给用户。有些设备例如网络和硬盘，既有输入，又有输出。

⊖ 输入设备（input device）：为计算机提供信息的装置，如键盘和鼠标。
⊜ 输出设备（output device）：输出计算结果给用户或其他计算机的装置。

第6章将详细介绍 I/O 设备，这里我们先对计算机硬件做一些基本的介绍，由外部 I/O 设备开始。



图 1-5　桌面计算机

液晶显示器（liquid crystal display，LCD）是主要的输出设备，键盘和鼠标是主要的输入设备，右边是一条以太网电缆，用于将笔记本电脑连接到网络上。笔记本电脑包括处理器、内存和额外的 I/O 设备。该系统由 Macbook Pro 15 英寸笔记本电脑连接到一台外部显示器而构成。

1.3.1　剖析鼠标

在参加一次计算机会议的演讲中，我产生了鼠标的概念。那个演讲者的演讲非常乏味，于是我开始做白日梦，萌生了这个想法。

——Doug Engelbart

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势阱中，乘着火箭以接近光的速度飞翔，同时我了解到计算机最深层的工作原理。

——Ivan Sutherland，计算机图形学之“父”，科学美国人，1984

虽然许多用户认为使用鼠标是理所当然的事情，但将鼠标作为指点设备的概念是由 Doug Engelbart于 1967 年最先提出的。当时，他演示了他的样机原型。而所有工作站（包括 Macintosh 和 Windows 操作系统）都使用鼠标作为指点设备的灵感则是始于 1973 年，由 Alto 提出。到了 20 世纪 90 年代，所有桌面计算机都使用了鼠标，它成为了基于图形显示器的用户接口标准。

最初的鼠标是电动机械式的，用一个大球在平面上滚动，产生坐标 x、y 两个计数器增量，每个增量显示鼠标移动了多远。

现在，电动机械式鼠标大多被光电鼠标取代了。光电鼠标实际上是一个小型的光处理器，使用 LED 提供光源，带有一个极小的黑白照相机和一个简单的光处理器。当 LED 照亮鼠标底下的平面时，照相机以每秒 1500 次的采样频率将拍摄的照片连续地输送给光处理器。光处理器通过对比照片，就可判定鼠标移动的方向和距离。电动机械式鼠标被光电鼠标所取代，说明了一个普遍的现象：由于电子学可降低成本并提高可靠性，因此电子技术能够取代老式的机电技术。在稍后我们还将看到另一个例子：闪存。

1.3.2　显示器

最吸引人的 I/O 设备应该是图形显示器了。所有的笔记本电脑、手持计算机、计算器、手机和几乎所有的桌面计算机现在都用 **LCD**[⊖]来获得轻巧、低功耗的显示效果。LCD 并非光源，而是

⊖ 液晶显示（liquid crystal display）：这是一种显示技术，将液体聚合物的薄层带电或者不带电，来传输或者阻止光线的传输。

控制光的传输。典型的 LCD 内含棒状液态分子团，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 度，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 显示器采用一种**动态矩阵显示**㊀（active matrix display）技术，其每个**像素**㊁（pixel）都有一个三极管精确地控制电流，使图像更清晰。在彩色有源矩阵中，还有一个红 – 绿 – 蓝屏决定三种颜色分量的强度，每个点需要有三个三极管开关。

图像是由像素矩阵组成的，可由位图（bit map）来表示。根据屏幕的大小和分辨率，显示矩阵的大小范围为从 640 ×480 像素到 2560 × 1600 像素（2008 年）。彩色显示器上的每色可用 8 位表示，每像素用 24 位，可表示几百万种不同的颜色。

支持图像的计算机硬件主要是光栅刷新缓冲（raster refresh buffer），或帧缓冲（frame buffer），用于保存位图。屏幕上的图像实际上保存在帧缓冲中，每个像素的位元模式以指定的刷新速率逐个读出并发送到显示器上。图 1-6 给出了一个简化的帧缓冲，其每个像素只有 4 位。

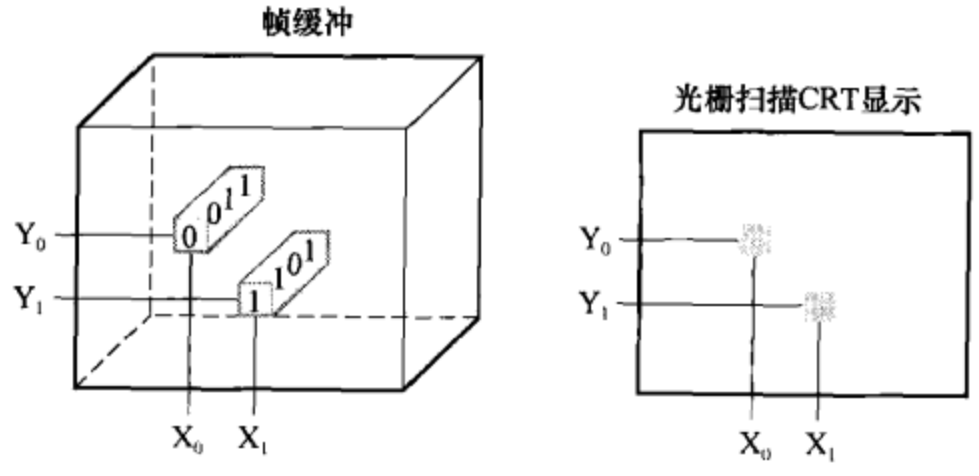


图 1-6　左边帧缓冲中每个坐标点决定了右边光栅扫描 CRT 显示中对应点的色度

像素（X_0，Y_0）的位元模式为 0011，相对于像素（X_1，Y_1）的位元模式 1101 来说更亮一些。

位图的目的是忠实地将图像在屏幕上复现出来，其难点在于人眼能精确发现屏幕上任何细微的变化。

1.3.3　打开机箱

如果打开计算机机箱，我们会看到吸引人的薄塑料底板，上面有几十个灰色或黑色的长方块。图 1-7 显示了图 1-5 中笔记本电脑的内部，图中上部是**主板**㊂，前面是两个磁盘驱动器（左边是硬盘驱动器，右边是 DVD 驱动器），中间的空间用于存放笔记本电脑的电池。

主板上那些小的长方块是**集成电路**㊃（integrated circuit），俗称**芯片**（chip）。主板由三部分组成：连接前面提到的各种 I/O 设备的电路、内存和处理器。

内存㊄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。图 1-8 给出了内存的照片，其中每个内存由 8 片集成电路构成。图 1-8 中的内存由多片 **DRAM**㊅芯片组成，被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。

㊀ 动态矩阵显示（active matrix display）：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。
㊁ 像素（pixel）：图像元素的最小单元。屏幕是由成千上万的像素组成的矩阵而形成。
㊂ 主板（*motherboard*）：包含一组集成电路芯片的塑料板，包括处理器、cache、内存以及连接 I/O 设备（如网络、硬盘等）的接口。
㊃ 集成电路（integrated circuit）：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。
㊄ 内存（memory）：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。
㊅ DRAM（dynamic random access memory）：动态随机访问内存，可随机访问任何地址的内存。

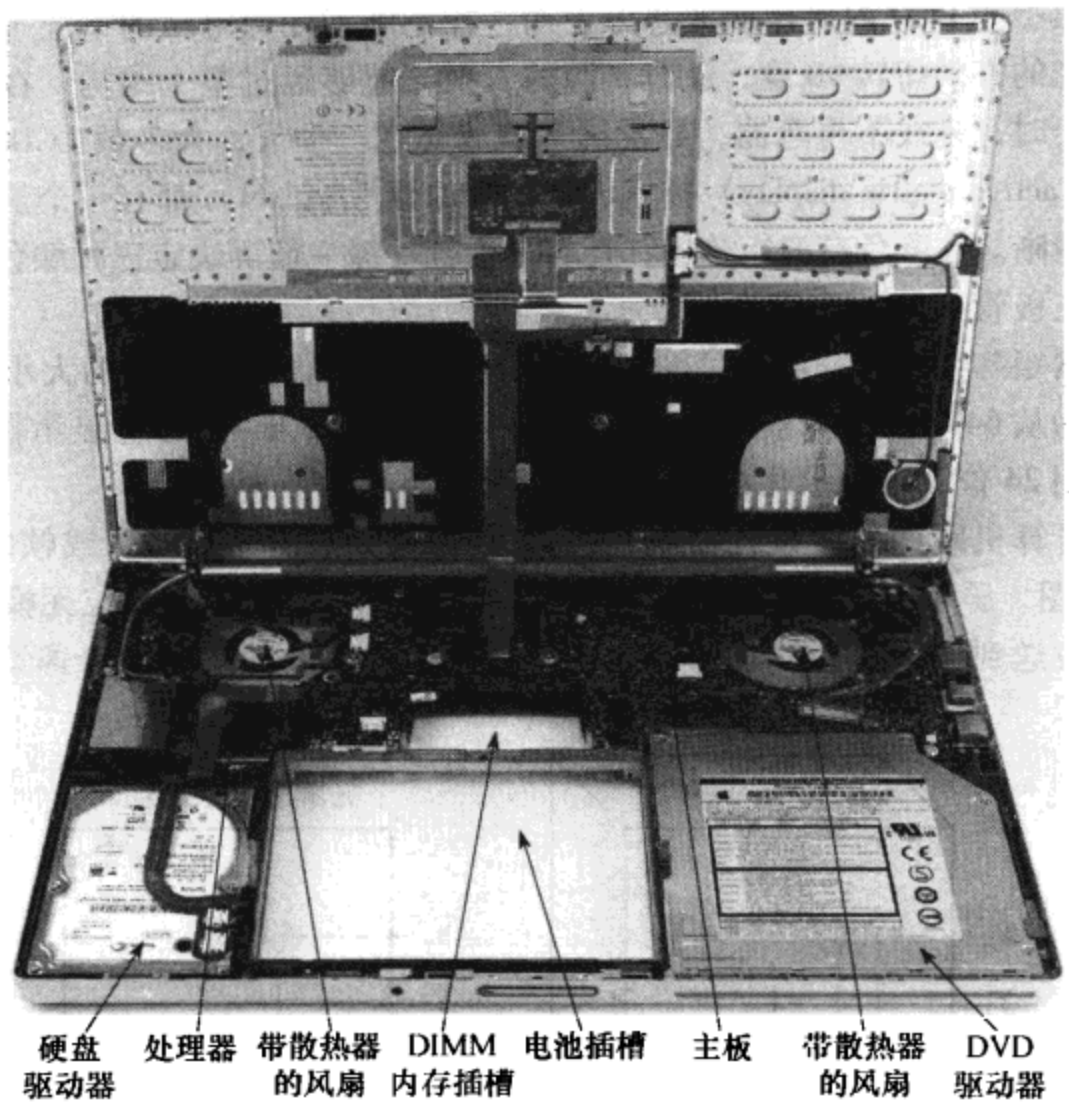


图 1-7　图 1-5 中笔记本电脑的内部

左下方具有白色标签的闪亮盒子是一个 100 GB SATA 硬盘驱动器，右下方那个闪亮的金属盒子则是 DVD 驱动器。它们两个之间的洞是笔记本电脑电池的位置。电池洞上的那个小洞则是存储器 DIMM。图 1-8 是 DIMMs 的一个特写图，它会从笔记本电脑的底层被插入。在电池洞和 DVD 驱动器的上方是一个印制电路板（PC 板），也被称为主板，它包含了计算机的大多数电子器件。本图上半部分的两个闪亮的圈是两个具有覆膜的风扇。处理器就是左边风扇的下面的那个大的突起的矩形。本图版权属于 OtherWorldComputing. com。

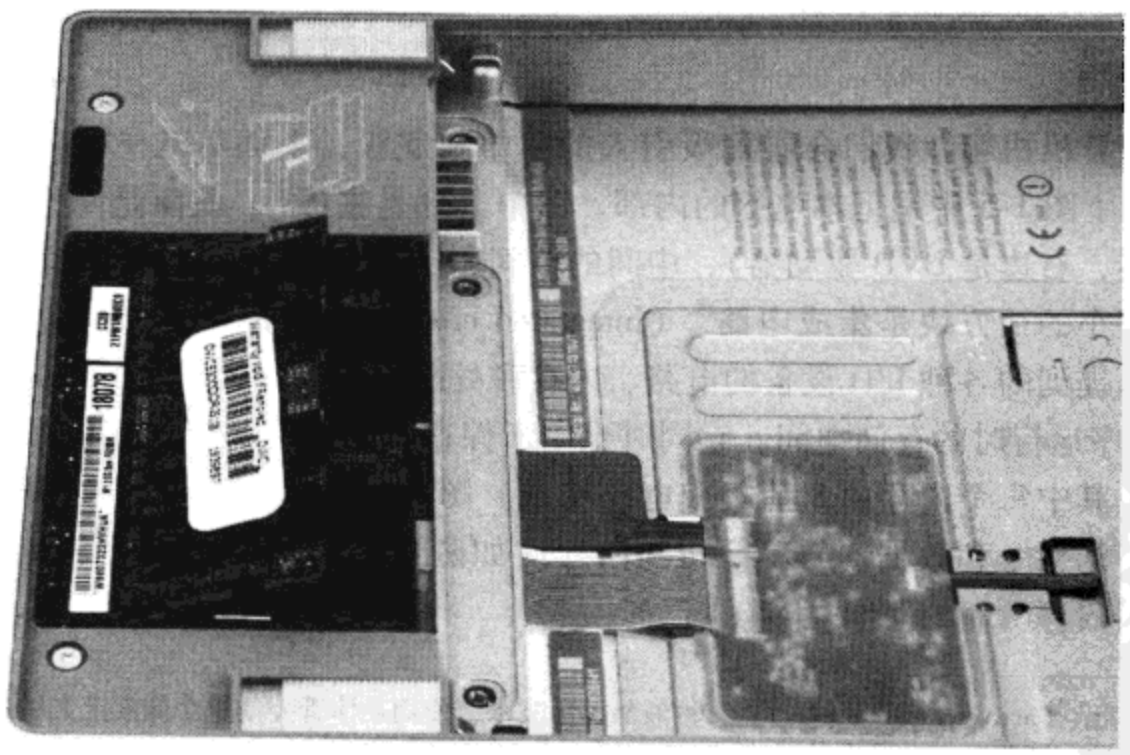


图 1-8　打开笔记本电脑底部所看到的内存

主存在左边一块或多块小板子上，右边是存放电池的空间。DRAM 安装在这些小板子（叫做 DIMM，dual inline memory module，双列直插内存模块）上并插入连接器。本图片由 OtherWorldComputing. com 提供。

处理器是主板上最活跃的部分。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 设备作出动作。处理器上面有风扇和散热器（见图 1-7 的左边）。有时候人们把处理器称为**中央处理器**㊀，即 **CPU**。

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**㊁负责完成算术运算，**控制器**㊂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 章将对数据通路和控制器进一步详细说明。

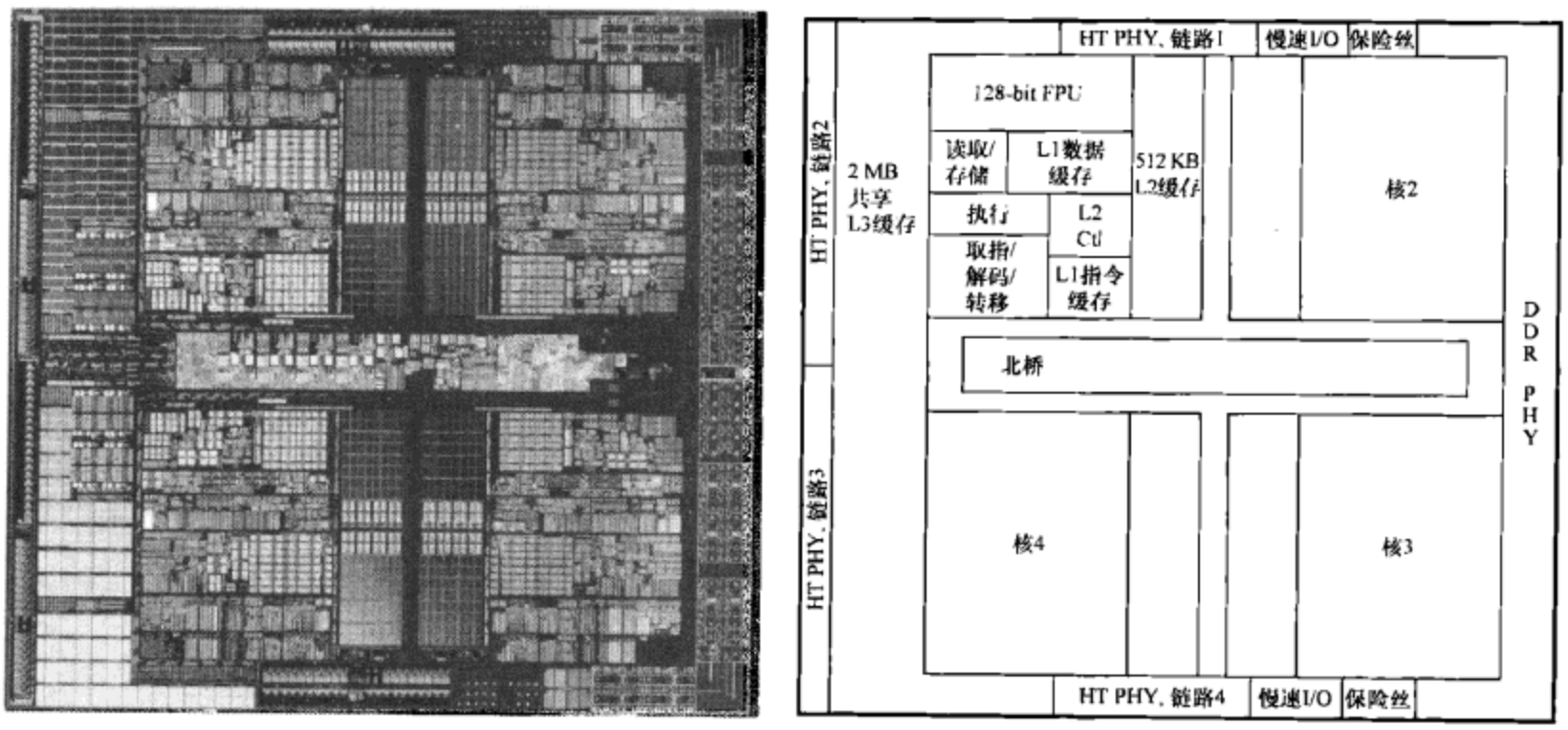


图 1-9　AMD Barcelona 微处理器内部

左边是 AMD Barcelona 微处理器芯片的显微镜照片，右边是该处理器包含的主要模块。图中的芯片由 4 个处理器构成，或称四核（core）处理器。而图 1-7 中笔记本电脑的每个芯片由 2 个核构成，称为 Intel Core 2 Duo。

在处理器内部使用的是另外一种存储器——**缓存**㊃。缓存是一种小而快的存储器，一般作为 DRAM 的缓冲。cache 采用的是另一种存储技术，称为**静态随机访问存储器**（SRAM）㊄，其速度更快而且不那么密集，因此价格更贵（见第 5 章）。

读者可能已经注意到，在软件和硬件的描述上有一个共同点：越是深入，展示的信息就越多；反过来，将低层的细节隐藏起来，就可以在高层次描述上采用较简洁的模型。使用“分层”或“**抽象**”㊅的方法，是设计复杂计算机系统的一种主要技术。

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**㊆，或简称**体系结构**（architecture）。计算机体系结构包括了程序员正确编写二进制机器语言程序所需的一切东西，如指令、I/O 设备，等等。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

㊀ 中央处理器单元（central processor unit）：也被称为处理器，处理器是主板上最活跃的部分。它严格按程序中的指令运行，将数字相加，测试结果，并按结果发出控制信号使 I/O 设备动作等。

㊁ 数据通路（datapath）：是处理器中执行算术操作的部分。

㊂ 控制器（control）：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。

㊃ 缓存（cache memory）：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。

㊄ 静态随机访问存储器（static random access memory）：一种存储器的集成电路，但是更快，比 DRAM 集成度低。

㊅ 抽象（abstraction）：一种掩盖底层计算机系统细节的模型，为了方便构建复杂的系统，暂时（temporarily）是不可见的。

㊆ 指令集体系结构（instruction set architecture）：也叫体系结构。是低层次软件和硬件之间的抽象接口，包含了需要写机器语言程序正确运行的所有信息，包括指令、寄存器、存储访问和 I/O 等。

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**㊀（ABI）。

计算机体系结构可以让计算机设计者独立地讨论功能，而不必考虑具体的硬件。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**㊁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。

重点

无论硬件还是软件都可以分成多个层次，每个较低的层次把细节对上层隐藏起来。硬件设计者和软件设计者都用这种抽象原则来应对计算机体系的复杂性。计算机体系结构是抽象层次中的一个关键接口——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。

1.3.4　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**易失性的**㊂。与之不同的是，如果关掉 DVD 游戏机的电源，所记录的内容将不会丢失，因为它采用的是**非易失性存储器**㊃。

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**㊄（main memory 或 primary memory），将后者称为**二级存储器**㊅（secondary memory）。DRAM 自 1975 年起在主存储器中占主导地位，而**磁盘**㊆自 1965 年起在二级存储器中占主导地位。在所有服务器和工作站中的非易失性存储器都是磁盘。**闪存**㊇也是一种非易失性存储器，主要用于手机，目前在音乐播放器甚至笔记本电脑中正在逐步取代磁盘。

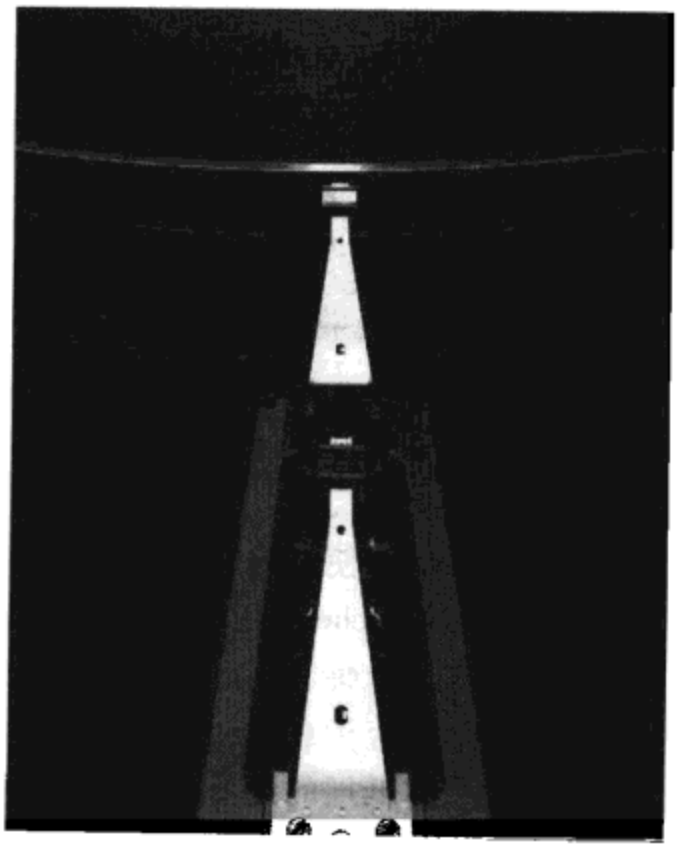


图 1-10　具有 10 张盘片的磁盘及其读写头

如图 1-10 所示，磁盘通常由多个盘片组成，盘片以每分钟 5400 ~ 15 000 转的速度绕轴高速旋转。金属盘片两面涂有磁性材料，相似的材料用在录音带或录影带上。为了读写硬盘上的信息，一个装有小线圈的活动臂紧靠两边盘面，称为“读写头”（read- write head）。整个驱动内部是密封的，使读写头更加靠近盘面。

硬盘的直径通常为 1 ~ 3.5 英寸。为了满足新产品的需要，硬盘直径越来越小，工作站、服务器、台式机、笔记本电脑、掌上电脑以及数码相机等新产品不断采用新型硬盘。一般来说，硬

㊀ 应用二进制接口（application binary interface）：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。

㊁ 实现（implementation）：遵循体系结构抽象的硬件。

㊂ 易失性内存（*volatile memory*）：类似 DRAM 的内存，仅在加电时保存数据。

㊃ 非易失性内存（nonvolatile memory）：在掉电时仍可保持数据的内存用于存储运行间的程序，例如磁盘。

㊄ 主存储器（main memory）：也叫主要存储器。这个存储器被用来保持运行中的程序，在现代计算机中一般由 DRAM 组成。

㊅ 二级存储器（secondary memory）：非易失性存储器，用来保存两次运行之间的程序和数据；在现代计算机中，一般由磁盘组成。

㊆ 磁盘（magnetic disk）：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存储设备。

㊇ 闪存（flash memory）：一种非易失性半导体内存，价格和速度均低于 DRAM，但比磁盘要快。

盘的尺寸越大性能越高，而尺寸越小单位价格越低，每 **GB**[⊖] 的最佳成本不同。大多数的硬盘驱动器都安装在计算机内部（如图 1-7 所示），也有一些硬盘驱动器通过外部接口连接（如 USB）。

由于磁盘采用了机械部件，因而其访问速度远远低于 DRAM，磁盘为 5 ~ 20 毫秒，而 DRAM 为 50 ~ 70 纳秒，比磁盘快约 100 000 倍。但是，相同容量的磁盘价格要比 DRAM 便宜得多，因为磁盘存储器的制造成本低于集成电路的制造成本。在 2008 年，每 GB 的磁盘价格比 DRAM 便宜 30 ~ 100 倍。

因此，磁盘和主存有三个主要差别：磁盘是非易失性的，因为它使用磁介质；磁盘的访问速度慢，因为它是机械装置；磁盘每 GB 价格相对较低，因为它容量很大，价格适当。

曾经有许多人试图发明一种新型存储技术，价格比 DRAM 便宜，而速度比磁盘快，但是大多都失败了。那些挑战者从来没能在正确的时机拿出产品。在挑战者的新产品发布时，DRAM 和磁盘取得了更大的进步，而且成本相应地大幅度下降，导致挑战者的产品立即变得过时。目前闪存是这一技术领域的一个重要的挑战者，它属于半导体存储器，像硬盘一样是非易失性的，并具有与其相近的带宽，而时延比硬盘快 100 ~ 1000 倍。闪存在音乐播放器、数码相机中的应用已经很普遍了，因为它体积小、更稳定、功耗低，尽管在 2008 年的时候它相对硬盘每 GB 价格贵6 ~ 10 倍。与硬盘和 DRAM 不同的是，闪存具有写 100 000 ~ 1 000 000 次后老化损坏的弱点。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 6 章将更详细地介绍闪存。

除了硬盘和闪存之外，目前还有几种正在应用的外存技术：

- 光盘（包括 CD、DVD）是最常见的可移动外存。蓝光（BluRay Disk，BD）是继 DVD 之后的下一代光盘格式标准。
- 基于闪存的可移动存储卡，通常采用 USB 接口，用于文件传输。
- 磁带，只能提供慢速串行传输，一直用于磁盘备份，现在常用冗余磁盘代替。

光盘的工作原理与磁盘不同。CD 通过在光盘表面烧制小坑（直径约 1 微米）的方法来记录数据。在读取 CD 时，使用激光照射 CD 表面，通过检测反射光来判定该处是一个坑还是平面。DVD 采用同样的技术，只不过激光可以聚焦多层，并大大减小每个坑的面积，从而具有更大的容量。蓝光则使用波长更短的激光进一步压缩每个坑的面积，从而增加存储的容量。

个人计算机上的光盘刻录机使用激光烧坑，速度相对较慢，刻录过程需要几分钟（CD 全片）至几十分钟（DVD 全片）。因此大量生产时一般采用压制技术，每片的成本只有几美分。

可擦写的 CD 或者 DVD 使用一种特殊的记录表面，这个表面具有结晶层、反射材料；所形成的坑不像一次性的 CD 或者 DVD 那样形成反射。为了擦写 CD 或者 DVD，其表面被加热，然后慢慢冷却，使用一种退火工艺使得表面的记录层恢复它的晶体结构；对于只读磁盘——被用来发布软件、音乐、电影——其磁盘成本和记录成本都低很多。

1.3.5　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 1.4 所示，处理器被连接到存储器和 I/O 设备。通过网络，计算机可以与其他的计算机通信，从而扩大计算能力。当今网络已经十分普遍，逐步成为了计算机系统的主干。一台新型计算机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。

⊖ GB（gigabyte）：一般是 1 073 741 824（2^{30}）字节，尽管一些通信和二级存储系统将其重新定义成 1 000 000 000（10^9）字节。相似的，根据上下文，megabyte 也就是 2^{20} 或者 10^6 字节。

- 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 1000 公里，传输速率可达到 10 Gbps。近距离范围之内通信使用**局域网**[㊀]（local area network，LAN），跨州、跨省通信则用**广域网**[㊁]（wide area network，WAN）。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。

在过去的 25 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，几个少数存在的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。在二十多年以前，第一个标准局域网的最大带宽为 10 Mbps，支持数十台计算机的共享工作。今天，局域网技术已能提供从 100 Mbps ~ 10 Gbps 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二十多年来信息革命的中心。

最近十年来，新的联网创新变革了计算机通信的方式。广泛使用的无线技术和笔记本电脑相结合，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得价格大幅度降低。当前无线通信技术，IEEE 标准 802.11，支持从 1 Mbps 到近 100 Mbps 的传输速率。无线技术和基于线路的网络相当不同，因为所有的用户在最近的区域里共享电波。

小测验

半导体 DRAM 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。

1.3.6　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在进步，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 1-11 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。1.7 节探讨了 1975 年以来支持计算器工业发展的技术和在可预见的未来技术的发展趋势。因为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应该熟悉集成电路的基础知识。

年份	计算机中使用的技术	相对性能/单价
1951	**真空管**[㊂]	1
1965	晶体管	35
1975	集成电路	900
1995	大规模集成电路	2 400 000
2005	超大规模集成电路	6 200 000 000

图 1-11　随着时间发展和技术进步，计算机技术的性价比不断增长

资料来源：Computer Museum，Boston，其中 2005 年的数据是由作者进行的预测。见光盘中的 1.10 节。

晶体管[㊃]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。为了描述这些晶体管从几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**大规模集成电路**[㊄]。

集成度的增长率是相当稳定的。图 1-12 表示自 1977 年以来 DRAM 容量的发展情况。近二十

㊀ 局域网（local area network）：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。
㊁ 广域网（wide area network）：一种可以区域能扩展到一片大陆那么大范围的网络。
㊂ 真空管（vacuum tube）：一种电子元件，是晶体管的前身，因工作的电极封装在 5 ~ 10 厘米长的真空玻璃管中而得名，使用电子束传输数据。
㊃ 晶体管（transistor）：一种由电信号控制的简单开关。
㊄ 大规模集成电路（very large-scale integrated circuit）：由数十万到数百万晶体管组成的电路。

多年以来，每隔3 年 DRAM 的容量就增长到4 倍，累积增长已超过16 000 倍。这就是集成电路领域中著名的摩尔定律：芯片中的晶体管容量每隔 18 ~ 24 个月将翻一倍。它由 Intel 公司创办人之一 Gordon Moore 于 20 世纪 60 年代提出。

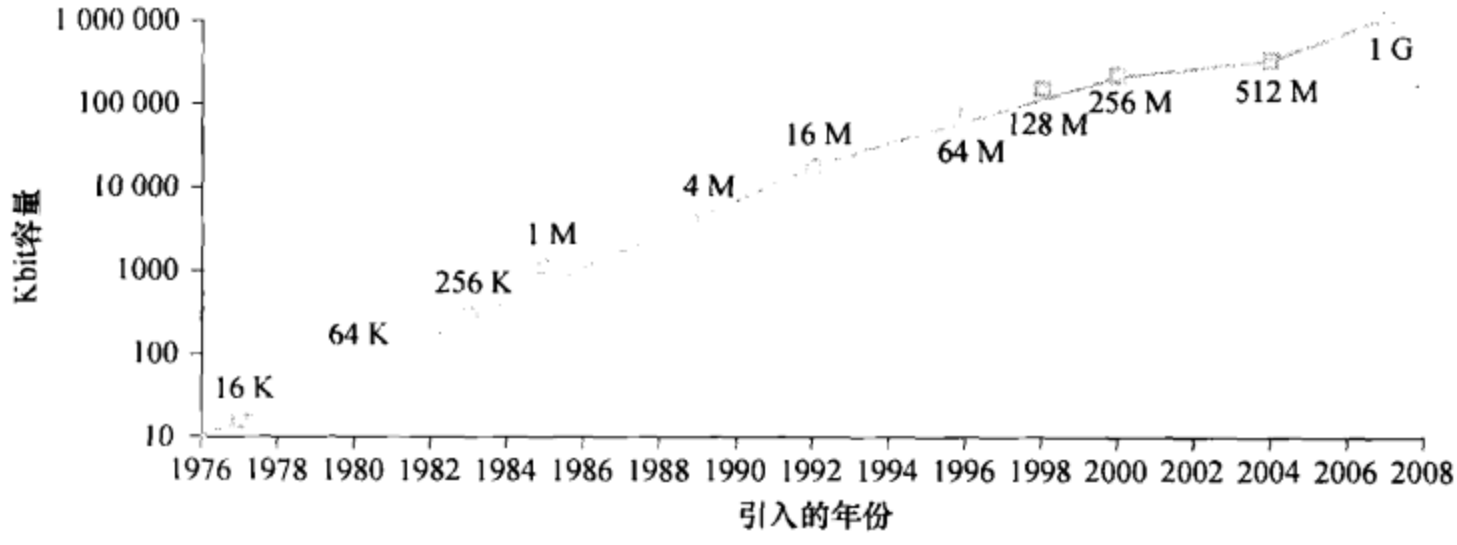


图 1-12　单片 DRAM 容量随时间的增长

纵轴单位为 Kb，其中 K 为 1024（2^{10}）。在近二十多年中，平均每隔三年 DRAM 容量扩大至 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。

这一增长速率几乎维持了 40 年，在制造技术方面需要大量难以置信的创新才能实现。1.7 节将讨论如何制造集成电路。

1.4　性能

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

在不同的计算机中挑选合适的产品，性能是极其重要的一个因素。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。通常销售人员希望你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。

1.4.1　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来看一下。图 1-13 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个客机的性能最好，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。

飞机型号	旅客容量	航程（英里）	巡航速度（英里/小时）	旅客吞吐率（旅客数 × 巡航速度）
波音777	375	4630	610	228 750
波音747	470	4150	610	286 700
BAC/Sud Concorde	132	4000	1350	178 200
道格拉斯DC-8-50	146	8720	544	79 424

图 1-13　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，它是容量乘以航行速度（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，即 Concorde。如果你关心的是旅客吞吐率，那么 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。作为个人计算机的用户，对**降低响应时间**[⊖]感兴趣。而数据中心感兴趣的常常是**吞吐率**[⊜]。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。

举例　**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？

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 不会使任务完成的更快，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 Y 的执行时间比 X 长。

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

$$\text{性能}_X/\text{性能}_Y = n$$

举例　**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？

答案

我们知道，A 是 B 的 n 倍快，则

$$\text{性能 A}/\text{性能 B} = \text{执行时间 B}/\text{执行时间 A} = n$$

故性能比为

$$15/10 = 1.5$$

⊖ 响应时间（response time）：也叫执行时间（execution time），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（throughput）：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。

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

$$性能_A / 性能_B = 1.5$$

意味着

$$性能_A / 1.5 = 性能_B$$

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个倒数关系，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”。

1.4.2　性能的测量

如果计算机的性能用时间来度量，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，时间可以用不同的方法来表示。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**[㊀]，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**[㊁]，后者称为**系统 CPU 时间**[㊂]。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。

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测量，也适用于 CPU 时间的测量，但本章的重点将放在 CPU 性能上。

理解程序性能

不同的应用关注计算机系统的不同方面。许多应用，特别是那些运行在服务器上的应用，主要关注 I/O 性能，所以此类应用既依赖硬件又依赖软件，对墙上时钟时间最感兴趣。而在其他一些应用中，用户可能对吞吐率、响应时间或两者的复杂组合更为关注（例如最差响应时间情况下的最大吞吐率）。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会更为方便。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**[㊃]。也可

㊀ CPU 执行时间（CPU execution time）：简称 CPU 时间（CPU time），执行某一任务在 CPU 上所花费的时间。

㊁ 用户 CPU 时间（user CPU time）：在程序本身所花费的 CPU 时间。

㊂ 系统 CPU 时间（system CPU time）：为执行程序而花费在操作系统上的时间。

㊃ 时钟周期（clock cycle）：也叫 tick、clock tick、clock period、clock 或 cycle 等，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。

用它的倒数来描述，称为时钟频率（clock rate）。例如，时钟周期为 250 ps，对应的时钟频率为 4 GHz。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。

小测验

1）假设某个使用桌面客户端和远程服务器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. 在客户端和服务器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少时间？

1.4.3　CPU 性能及其因素

用户和设计者往往以不同的尺度看待性能。如果我们能掌握这些不同尺度之间的关系，就能确定一个设计的变化对性能的影响。由于我们都关注 CPU 性能，因而性能测量实际上针对的是 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：

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

这个公式清楚地表示，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。

由于时钟频率和时钟周期时间互为倒数，故

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

由此可见，提高时钟频率也能改进性能。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。

举例　**性能的改进**

某程序在一台时钟频率为 2 GHz 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 A1.2 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：

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

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

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

B 的 CPU 时间公式为：

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

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

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

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

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。

1.4.4　指令的性能

上述的性能公式没有涉及程序所需的指令数。（在第 2 章中，我们将看到指令是如何组成程序的。）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**[⊖]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。

举例　**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 250 ps，对某程序的 CPI 为 2.0；计算机 B 的时钟周期为 500 ps，对同样程序的 CPI 为 1.2。对于该程序，请问哪台计算机执行的速度更快？快多少？

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

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

因此，对于该程序计算机 A 是计算机 B 的 1.2 倍快。

1.4.5　经典的 CPU 性能公式

现在我们可以用**指令数**[⊜]、CPI 和时钟周期时间来写出基本的性能公式：

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

或

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

这些公式特别有用，因为它们把性能分解为三个关键因素。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。

举例　**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：

⊖ CPI（clock cycles per instruction）：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。

⊜ 指令数（instruction count）：执行某程序所需的总指令数量。

	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 2 执行的指令数更多。

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：

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

因此，代码序列 1 的 CPU 时钟周期数 =(2 ×1) +(1 ×2) +(2 ×3) =10 周期，代码序列 2 的 CPU 时钟周期数 =(4 ×1) +(1 ×2) +(1 ×3) =9 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，它一定具有较小的 CPI。CPI 的计算公式为：

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

代入相应数据可得

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

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

重点

图 1-14 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）：

执行时间 = 秒/程序 = 指令数/程序 × 时钟周期数/指令 × 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改进可能降低时钟周期时间或提高 CPI，从而抵消了改进的效果。类似地，CPI 与执行的指令类型相关，执行指令数最少的代码其执行速度未必是最快的。

性能指标	测量单位
程序的CPU执行时间	程序执行的秒数
指令数	程序执行的指令数
CPI	每条指令的平均时钟周期数
时钟周期时间	每时钟周期的秒数

图 1-14　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。

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

上述的例子表明，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），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，这样指令的条数和 CPU 必须被比较，即使时钟频率是相同的。在本章最后的练习题中，有几个是关于计算机和编译程序的性能评价。在 1.8 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。

理解程序性能

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

硬件或软件指标	影响什么	如何影响
算法	指令数，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 语言对数据抽象进行了大量支持，将进行间接调用，需要使用 CPI 较高的指令。
编译程序	指令数，CPI	编译程序的效率，既影响指令数，又影响 CPI。因为编译程序决定了最后生成的计算机指令，所以会以复杂的方式影响 CPI。
指令集体系结构	指令数，时钟频率，CPI	指令集体系结构影响 CPU 性能的所有方面，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。

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

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，但不幸的是 CPI 增加为原来的 1.1 倍。请问该程序在新版本的 Java 编译程序中运行速度是多少？从以下三个选项中选出正确答案。

A. $15 \times 0.6/1.1 = 8.2$ 秒

B. $15 \times 0.6 \times 1.1 = 9.9$ 秒

C. $15 \times 1.1/0.6 = 27.5$ 秒

1.5　功耗墙

图 1-15 表示 25 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将近 20 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。

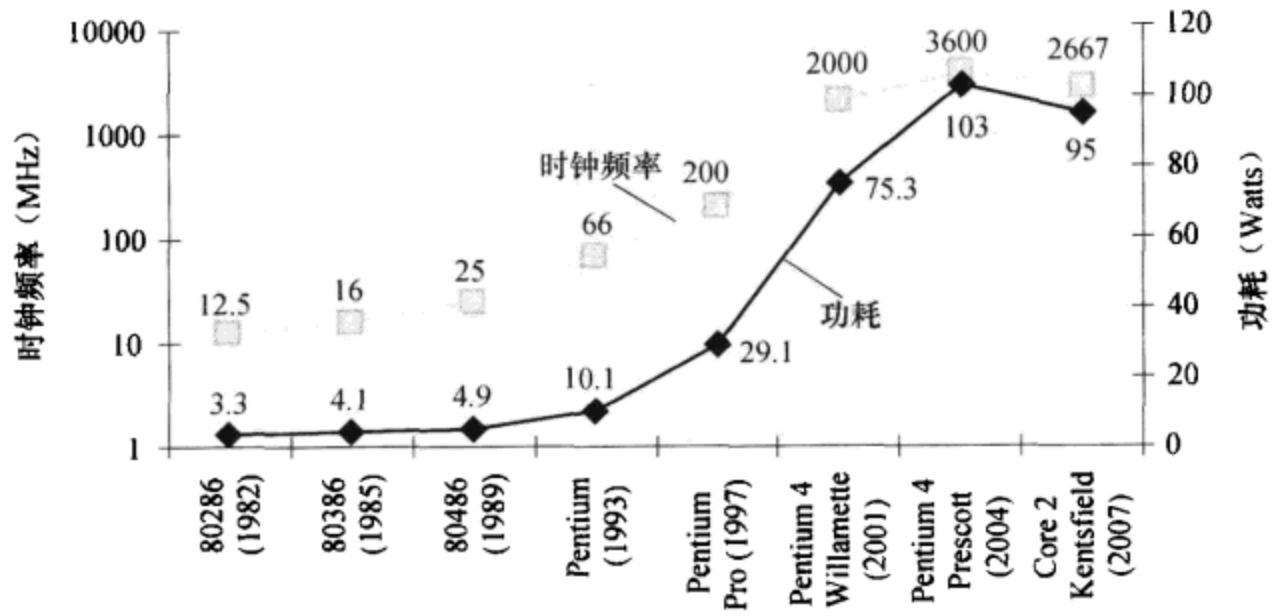


图 1-15　25 年间 Intel x86 八代微处理器的时钟频率和功耗

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（thermal）问题导致奔腾 4 处理器（生产线）的放弃。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。

占统治地位的集成电路技术是 CMOS（互补型金属氧化半导体），其主要的功耗来源是动态功耗，即在晶体管开关过程中产生的功耗。动态功耗取决于每个晶体管的负载电容、工作电压和晶体管的开关频率：

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

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。

为什么时钟频率增长为 1000 倍，而功耗只增长为 30 倍呢？因为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 5 V 降到了 1 V。这就是功耗只增长为 30 倍的原因所在。

举例　**相对功耗**

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

答案

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

因此功耗比为

$$0.85^4 = 0.52$$

新处理器的功耗大约为旧处理器的一半。

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

为了解决功耗问题，设计者连接大设备，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，而继续提高芯片的功耗到 300 瓦特，但对桌面计算机来说成本太高了。

由于计算机设计者遇到了功耗墙问题，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：虽然动态功耗是 CMOS 功耗的主要来源，但静态功耗也是存在的。因为即使在晶体管关闭的情况下，还是有泄漏电流存在。2008 年时典型的电流泄漏占 40% 的功耗。因此，增加晶体管的数目，就会增加漏电功耗，即使这些晶体管总是关闭的。各种各样的设计和工艺创新被用来控制电流泄漏，但还是难以进一步降低电压。

1.6　沧海巨变：从单处理器向多处理器转变

迄今为止，很多软件很像独唱编写者所写的音乐；使用当代的芯片，我们对于编写二重唱、四重唱，以及小型的合奏具有少量的经验，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，《并行领域的计算》，2007

功耗的极限迫使微处理器的设计产生了巨变。图 1-16 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 下降到不足 1.2。

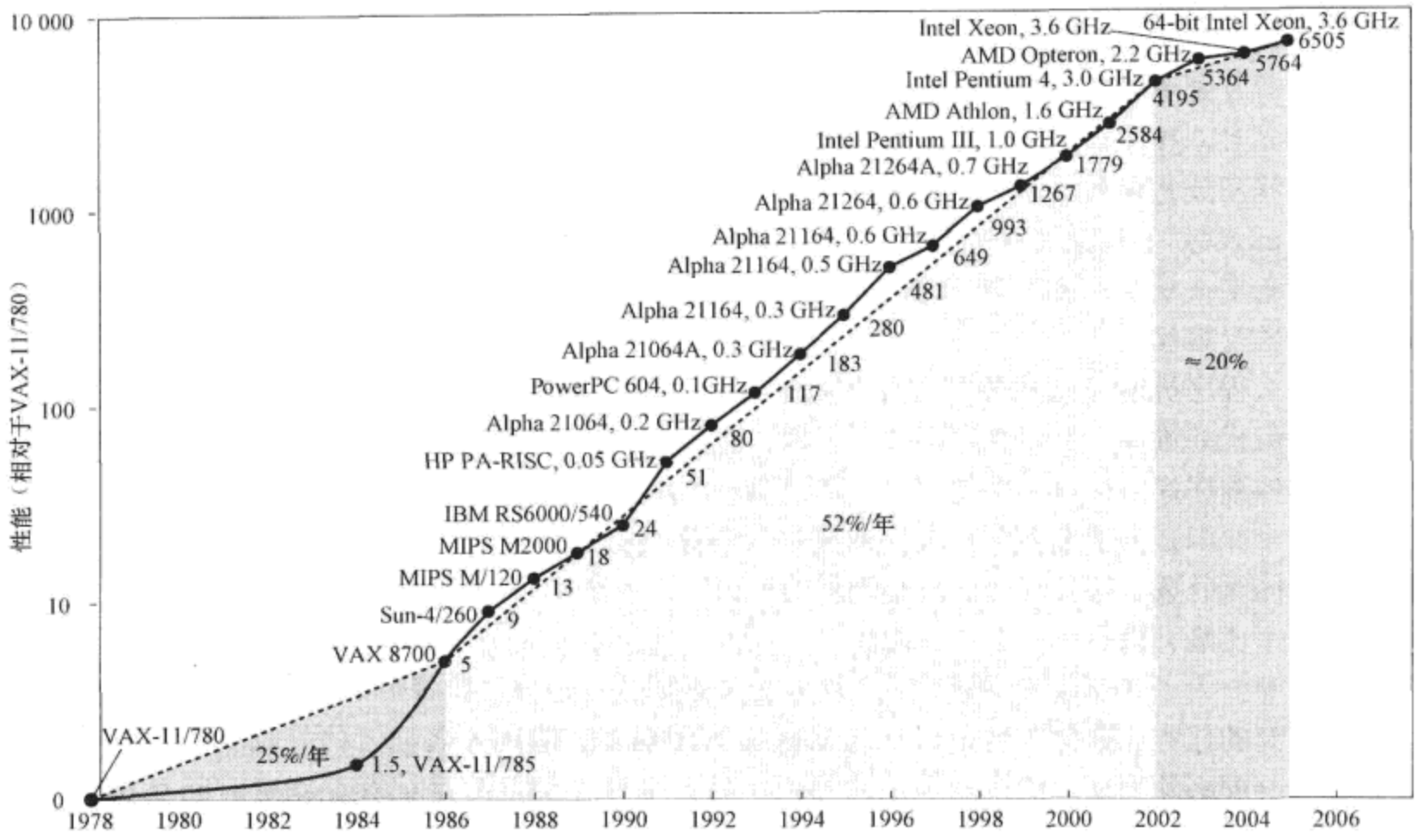


图 1-16　自 20 世纪 80 年代中期以来处理器性能的发展

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 1.8 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%，这些归功于体系结构的创新和结构变化。到 2002 年，这种性能增长发生了变化，大约是 7 的因子（a factor of seven），面向浮点计算的性能大幅度增长。从 2002 年开始，受到功耗、指令级并行程度和长的存储器延迟的限制，单核处理器的性能增长放缓，大约每年 20%。

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，这样的微处理器（microprocessor）就是多核处理器了。因此，一个“四核”微处理器是一个芯片，包含了 4 个 processor 或者 4 个 core。

图 1-17 给出了最近微处理器中核的数目、功耗和时钟频率。在许多公司宣布的产品计划中，

核的数目大约每 2 年将会翻一番（见第 7 章）。

产品	AMD Opteron X4 (Barcelona)	Intel Nehalem	IBM Power 6	Sun Ultra SPARC T2 (Niagara 2)
每片核数	4	4	2	8
时钟频率	2.5 GHz	约2.5 GHz	4.7 GHz	1.4 GHz
微处理器的功耗	120 W	约100 W	约100 W	94 W

图 1-17　2008 年多核微处理器每芯片的核数、时钟频率和功耗

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。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。

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，但它往往是隐蔽的。第 4 章将说明流水线，它是一种漂亮的技术，通过指令重叠执行使程序运行得更快。这是*指令级并行性*的一个例子。在抽取了硬件的并行本质之后，程序员或编译程序可认为在硬件中指令是串行执行的。

迫使程序员意识到硬件的并行性，并显式地按并行方式重写其程序，曾经是计算机体系结构的“第三抱怨”，以致很多采用此种方式进行革新的公司都失败了（见光盘上 7.14 节）。从历史发展的角度来观察，整个 IT 行业已经把它放到了未来的发展方向上，程序员最终将成功地跃进到显式并行编程。

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，以不至于把并行的性能都浪费掉。

作一个比喻，现在有一个写新闻故事的任务，如果由 8 名记者共同来完成，能否提高 8 倍的写作速度呢？为了实现这一目标，这个新闻故事需要进行划分，让每个记者都有事可做。假如某名记者分到的任务比其他 7 名记者加起来的任务还要多，那用 8 名记者的好处就不存在了。因此，任务分配必须平衡才能得到理想的加速。另一个存在的危险是记者要花费时间互相交流才能完成所分配的任务。如果故事的一部分，例如结论，在所有其他部分完成之前不能编写，则你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。对于本文的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，或是核的数目更多时，并行编程的挑战将更大。

为了反映业界的这个沧海巨变，后面的五章里每章都会有一节介绍有关并行性革命的内容：

- *第 2 章，2.11 节：并行与指令：同步*。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：并行性和计算机算术：结合律*。许多并行程序员往往从正在使用中的串行程序开始。要确认他们的并行版程序是否能工作就要回答以下问题：是否和串行程序得到了同样的结果？如果答案是否定的，那么并行版的新程序就存在错误。这个逻辑假定计算机运算是遵守结合律的：将一百万个数相加，无论次序如何，得到的和是相同的。本章将解释该逻辑对整数是成立的，但对浮点数并不成立。

- 第 4 章，4.10 节：并行和高级指令级并行。尽管明确地知道并行编程的困难，在 20 世纪 90 年代依然付出了巨大的努力和投资用于研究硬件和编译程序的并行性。本章描述了其中的一些技术，包括取指与多指令同时执行和推测决策结果、指令执行等。
- 第 5 章，5.8 节：并行与存储器层次结构：cache 一致性。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制。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 6 章，6.9 节：并行性与 I/O：廉价磁盘冗余阵列（RAID）。如果你在并行性革命中忽略了 I/O，那么你的并行程序将在等待 I/O 上浪费大量的时间。本章介绍的 RAID 技术，可以加速外存访问的速度，这体现了并行性的另一个优点：利用资源的多个复制，即使有一个复制失效了，系统仍能继续工作。因此，RAID 能同时改进 I/O 性能和可用性。

除了这些章节之外，还有一整章介绍并行编程。第 7 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，描述和评价了四种使用该种模型的多核微处理器。

本书从这一版开始在附录 A 中介绍了 GPU（graphics processing unit）。GPU 是一种在桌面计算机中越来越普及的图形处理器，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 A 介绍了 NVIDIA GPU 及其并行编程环境。

1.7　实例：制造以及 AMD Opteron X4 基准

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

——J. Presper Eckert，ENIAC 的创建者之一，言论发表于 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到的技术。下面是本书中的第一个“实例”小节，我们将以 AMD Opteron X4 为例，说明如何制造集成电路，以及如何测量性能和功耗。

芯片的制造从**硅**㊀开始。硅是从沙子中发现的一种物质。由于它导电性能不好，所以称为**半导体**㊁。用特殊的化学方法对硅添加某些材料，可以把其细微的区域转变为以下三种类型之一：

- 良好的导电体（类似于细微的铜线或铝线）
- 良好的绝缘体（类似于塑料或玻璃膜）
- 可控的导电体或绝缘体（类似开关）

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决定芯片的价格非常关键，因此对计算机设计者十分重要。图 1-18 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**㊂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。硅锭经切片机切成片厚度不超过 0.1 英寸的**晶圆**㊃。这些晶圆经过大约 20 ~ 40 步化学加工最终产生之前所讨论的晶体管、导体和绝缘

㊀ 硅（silicon）：一种自然元素，它是一种半导体。
㊁ 半导体（semiconductor）：一种导电性能不好的物质。
㊂ 硅锭（silicon crystal ingot）：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。
㊃ 晶圆（wafer）：厚度不超过 0.1 英寸的硅锭片，被用来制造芯片。

体。如今的集成电路包含一层晶体管，但是可能具有多个绝缘层间隔的 2 ~ 8 层金属导体。

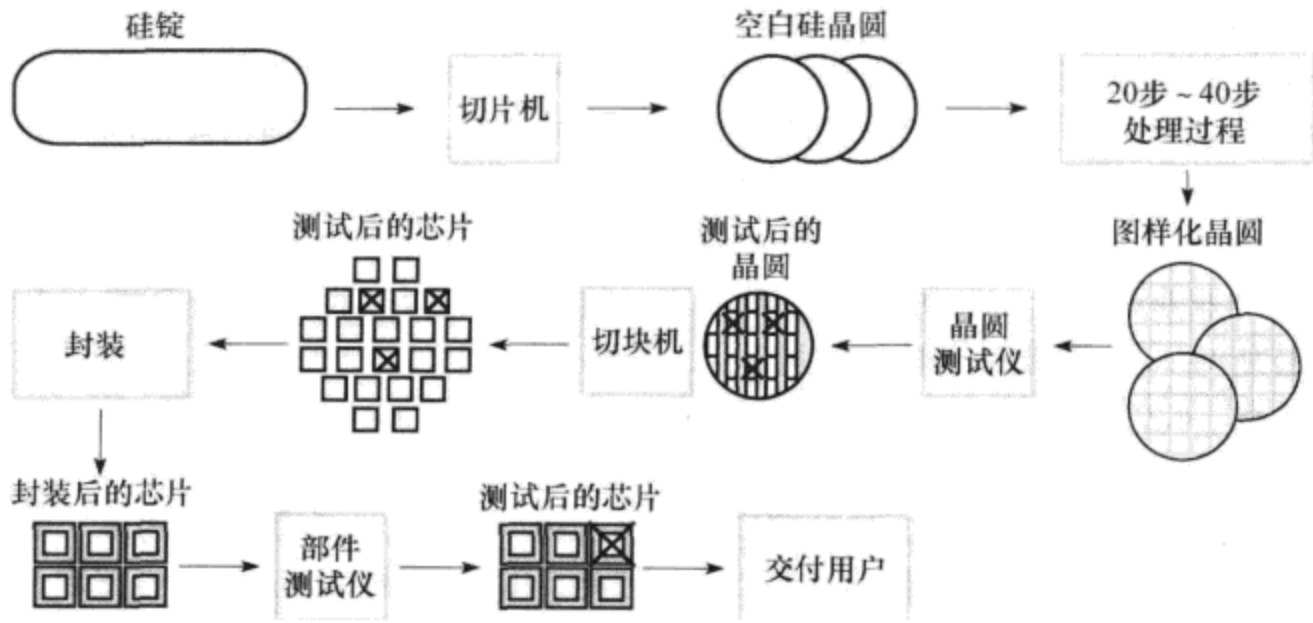


图 1-18　芯片制造的全过程

从硅锭切下来之后，空白的晶圆经过大约 20 步 ~ 40 步的加工，产生图样化的晶圆（见图 1-19）。这些图样化的晶圆被一个晶圆测试器所测试，产生一个表明哪些部分是好的图。之后，这些晶圆被进一步切成芯片（见图 1-19）。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/成品率是 17/20，也就是 85%。这些合格芯片被封装而且发布给用户之前经过多次测试。一个坏的封装会在最终的测试中被发现。

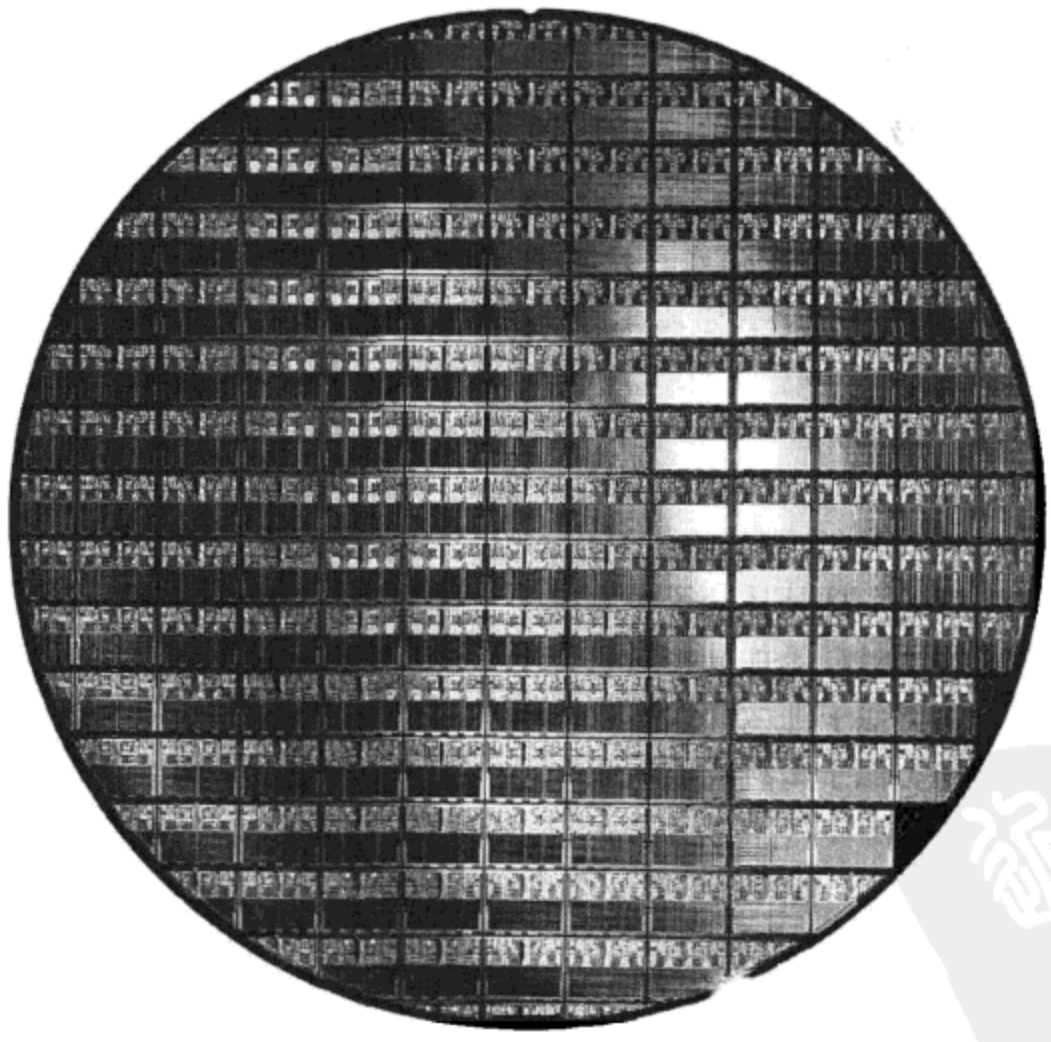


图 1-19　AMD Opteron X2 芯片的 12 英寸（300 mm）晶圆，Opteron X4 芯片的上代产品（AMD 提供）

良率为 100% 的圆片中的晶圆的数目是 117。晶圆边缘几十个不完整的芯片是没用的。之所以包含它们，因为这样给硅片生产掩膜相当容易。晶圆使用 90 nm 的工艺，这意味着最小的晶体管的尺寸几乎接近 90 nm，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。

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[㊀]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**[㊁]。图 1-19 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及其主要部件。

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**[㊂]来表示，其定义为合格芯片数占总芯片数的百分比。例如，假设总芯片有 22 片，其中有瑕疵的芯片为 3 片，则成品率为 19/22 = 86%。

当芯片尺寸增大时，集成电路的价格会快速上升，因为成品率和硅片中芯片的总数都下降了。为了降低价格，一个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）的方法，从而改进每硅片的芯片数和成品率。

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。最后芯片将被交付用户。

如之前提到的，功耗是在设计过程中所面临的不断增长的压力，主要有两个原因：第一，芯片的工作必须供电，并且电源必须均匀地分布在芯片上，现代微处理器通常用几百个引脚满足供电和接地的需求，并使用多层互连来使电源和接地均匀分布到芯片上。第二，功耗是以散热的形式从芯片中排除的，而芯片的面积非常小。例如，2008 年的 AMD Opteron X4 2356 2.0 GHz 处理器功耗为 120 瓦，而其表面积只有约 1 平方厘米。

精解：集成电路的成本可以用下面 3 个公式来表示：

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

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

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

第 1 个公式是直接导出的。第 2 个公式是近似的，因为没有减去晶圆边上不满足芯片矩形要求的面积（参见图 1-19）。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、芯片和晶圆的面积，与芯片的面积之间的关系一般不是线性的。

1.7.1　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**[㊃]。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运行时间。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，从而决定最终的选择。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**[㊄]。这些测试程序形成负载，用户期望预测实际负载的性能。

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，目的是为现代计算系统建立基准测试程序集。1989 年，SPEC 建立了重点面向处理器性能的基准程序集（现在称为 SPEC89）。历经 5 代发展，目前最新的是 SPEC CPU2006，它包括 12 个整数基准程序集（CINT2006）和 17 个浮点基准程序集（CFP2006）。CINT2006 包括 C 编译程序、量子计算机仿真、下象棋程序等，CFP2006 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 1-20 列举了 SPEC 整数基准程序及其在 AMD Opteron X4 上的执行时间、指令数、CPI 和时

㊀ 瑕疵（defect）：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。
㊁ 芯片（die）：从晶圆中切割出来的一个单独的矩形区域，更加正式的叫法是芯片（chip）。
㊂ 成品率（yield）：合格芯片数占总芯片数的百分比。
㊃ 工作负载（workload）：运作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。
㊄ 基准测试程序（benchmark）：用于比较计算机性能的程序。

钟周期时间等。注意：CPI 的最大值和最小值相差达到 13 倍。

描述	名称	指令数 × 10^9	CPI	时钟周期时间（秒 × 10^9）	执行时间（秒）	参考时间（秒）	SPECratio
解释性串处理	perl	2118	0.75	0.4	637	9770	15.3
块分类压缩	bzip2	2389	0.85	0.4	817	9650	11.8
GNU C编译器	gcc	1050	1.72	0.4	724	8050	11.1
组合优化	mcf	336	10.00	0.4	1345	9120	6.8
go游戏（人工智能）	go	1658	1.09	0.4	721	10 490	14.6
搜索基因序列	hmmer	2783	0.80	0.4	890	9330	10.5
象棋游戏（人工智能）	sjeng	2176	0.96	0.4	837	12 100	14.5
量子计算机仿真	libquantum	1623	1.61	0.4	1047	20 720	19.8
视频压缩	h264avc	3102	0.80	0.4	993	22 130	22.3
离散事件仿真库	omnetpp	587	2.94	0.4	690	6250	9.1
游戏/路径寻找	astar	1082	1.79	0.4	773	7020	9.1
XML语法分析	xalancbmk	1058	2.70	0.4	1143	6900	6.0
几何平均值							11.7

图 1-20　SPECINTC2006 基准程序在 AMD Opteron X4 model 2356（Barcelona）上的运行结果

按照 1.4.5 节的等式，执行时间是本表的三个因素的乘积：上亿的指令数、每个执行的时钟数（CPI），以及纳秒级的时钟周期时间。SPECratio 仅仅是参考时间，由 SPEC 所提供，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。图 5-40 展示的是 mcf、libquantum、omnetpp 以及 xalancbmk 具有相对高的 CPI，因为这些负载的 cache 缺失率比较高。

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个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。

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 是执行时间按参照计算机进行标准化的结果，$\prod_{i=1}^{n} a_i$ 表示 $a_1 \times a_2 \times \cdots \times a_n$。

1.7.2　SPEC 功耗基准测试程序

目前，SPEC 提供了十几种不同的基准测试程序，使用真实的应用程序、严格制定的执行规则以及报告需求，达到测试不同的计算环境的目的。其中最新的是 SPECpower，它可以报告服务器在不同负载水平下（以 10% 的比例递增）的功耗。图 1-21 给出了在基于 Barcelona 处理器的服务器上的测试结果。

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、caches、主存以及 Java 虚拟机、编译器、无用单元收集器、操作系统片段。性能采用吞吐率来测量，单位是每秒完成的操作次数。还是为了简化结果，SPEC 采用单个的数字来进行归纳，称为“overall ssj_ops per Watt”，其计算公式是：

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

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

目标负载	性能 (ssj_ops)	平均功耗（瓦）
100%	231 867	295
90%	211 282	286
80%	185 803	275
70%	163 427	265
60%	140 160	256
50%	118 324	246
40%	92 035	233
30%	70 500	222
20%	47 126	206
10%	23 066	180
0%	0	141
总和	1 283 590	2605
Σssj_ops / Σpower =		493

图 1-21　SPECpower_ssj 2008 在服务器上的运行结果

服务器的具体配置为双插槽 2.3 GHz AMD Opteron X4 2356（Barcelona）处理器，16 GB DDR2-667 DRAM，500 GB 硬盘。

小测验

产量是决定集成电路价格的一个关键因素。下列哪些理由说明了芯片产量越高成本就越低？

A. 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。

B. 设计高产量芯片的工作量比设计低产量芯片小。

C. 制造芯片用的掩膜很贵，产量高时每芯片的掩膜成本就低。

D. 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。

E. 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。

1.8　谬误与陷阱

科学一定开始于神话和对神话的批判。

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

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

软硬件设计者都曾碰到过这个陷阱。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：

$$\text{改进后的执行时间} = \text{受改进影响的执行时间}/\text{改进量} + \text{不受影响的执行时间}$$

代入本例的数据进行计算：

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

由于要求快至 5 倍，新的执行时间应该是 20：

⊖ Amdahl 定律（Amdahl's law）：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。

$$20 = 80/n + 20$$

$$0 = 80/n$$

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。这个概念也产生了在每天的生活中我们称为“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。

硬件设计的共同主题是 Amdahl 定律的一个推论：*加速常见事件*。这个推论提示我们，在许多情况下某些事件的发生频率可能比其他事件高得多。因此，改进量的大小受事件占用时间的多少的限制。因此，加快常见事件相对于加快罕见事件更能提高性能。具有讽刺意味的是，常见事件往往比罕见事件更简单，因而更容易实现。

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 7 章中的“谬误与陷阱”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 服务器中 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%。图 1-22 给出了三种服务器在 100% 负载、50% 负载、10% 负载和空闲时的最好 SPECpower 结果。从中可以看到，即使服务器的利用率只有 10%，也会消耗掉大约 2/3 的峰值功耗。

服务器制造商	微处理器	总核数/插槽数	时钟频率	性能峰值 (ssj_ops)	100% 负载下的功耗	50% 负载下的功耗	50% 负载/100% 功耗	10% 负载下的功耗	10% 负载/100% 功耗	主动空闲下的功耗	主动空闲/100% 功耗
HP	Xeon E5440	8/2	3.0 GHz	308 022	269 W	227 W	84%	174 W	65%	160 W	59%
Dell	Xeon E5440	8/2	2.8 GHz	305 413	276 W	230 W	83%	173 W	63%	157 W	57%
Fujitsu Seimens	Xeon X3220	4/1	2.4 GHz	143 742	132 W	110 W	83%	85 W	65%	80 W	60%

图 1-22　三种服务器在最佳 ssj_ops per watt 时的 SPECPower（2007 年第 4 季度）

三种服务器的 ssj_ops per watt 依次为 698、682 和 667。上面两种服务器内存为 16 GB，最下面的服务器内存为 8 GB。

陷阱：用性能公式的一个子集去度量性能。

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。

例如，曾经有一种用 **MIPS**[⊖]（每秒百万条指令）取代时间去度量性能的方法。对于一个给定的程序，MIPS 表示为：

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

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。

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，

⊖ MIPS（million instructions per second）：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。

但没有考虑到指令的能力。我们没有办法用 MIPS 去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，将执行时间用指令数、CPI、时钟频率代入之后可得：

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

回顾一下，SPEC2006 在 AMD Opteron X4 上的 CPI 最大值和最小值是相差 13 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。

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	4 GHz	4 GHz
CPI	1.0	1.1

1）哪台计算机的 MIPS 值更高？

2）哪台计算机更快？

1.9　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 1 000 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是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。这个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。产生的一个副效应：这些预先排除可能需要接口发生变化的那些革新结构。

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

秒数/程序 =（指令数/程序）×（时钟周期数/指令数）×（秒数/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。

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，或是没有清晰的说明有效性的限制条件。

现代处理器硬件的关键技术是硅。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，开发存储层次结构的访问局部性，目前的典型方法是通过 cache。

功耗已经取代芯片面积，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。

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，如功耗、可靠性、成本和可扩展性等。尽管本章的重点放在价格、性能和功耗上，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

本章介绍了计算机的基本概念，以及计算机的五个常见部件：运算器、控制器、存储器、输入和输出（见图 1-4）。这五个部件也是本书后面几章的框架：

- 运算器：第 3、4、7 章和附录 A
- 控制器：第 4、7 章和附录 A
- 存储器：第 5 章
- 输入：第 6 章
- 输出：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 7 章介绍并行革命的核心——显式并行多核微处理器，附录 A 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 B 提供了第 2 章指令集的参考数据。第 3 章介绍计算机如何处理算术运算。附录 C 在光盘上，介绍逻辑设计。

1.10　拓展阅读

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

——Lewis Thomas，《细胞生命的礼赞》中的“自然科学”，1974

本书的每一章都有“拓展阅读”一节，可在本书所附的光盘上找到。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，或者叙述一些重要的历史贡献，还提供参考数据以便你进一步探究。

本章的“拓展阅读”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作出贡献的重要历史人物。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。CD 中每个拓展阅读之后都会提示进一步阅读，这部分具体内容会在 CD 中的“进一步阅读”部分。在 CD 中能看到 1.10 节的剩余部分。

1.11　练习题

练习由 Universidade de Santiago de Compostela（圣地亚哥大学 · 德孔波斯特拉）的 Javier Bruguera 提供。

本版的习题大都设计成以定性的介绍辅以可选的参数表。这些参数是解题所必需的，你可以决定采用任意一个，或者全部的参数来解题（每个题目需要多少个参数由你决定）。例如，可以将作业布置成“用表中 A 行的参数完成练习 4.1.1”。教师也可以依需要更换参数来定制习题以求新的解题方案。

定量习题的数目各章不同，主要取决于教材内容。当定量的方法不适合时就提供一些常规

的练习。

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，<1.3>表示你应该在读过 1.3 节后才能完成本题。

习题 1.1

从下面的列表中找出与问题最为匹配的一项，用项号回答，每项只能使用一次。

1. 虚拟世界	8. 数据中心	15. 编译程序	22. 高级语言
2. 桌面计算机	9. 嵌入式计算机	16. 位	23. 系统软件
3. 服务器	10. 多核处理器	17. 指令	24. 应用软件
4. 低端服务器	11. VHDL	18. 汇编语言	25. Cobol
5. 超级计算机	12. RAM	19. 机器语言	26. Fortran
6. terabyte	13. CPU	20. C	
7. petabyte	14. 操作系统	21. 汇编程序	

1.1.1 ［2］<1.1>用于运行大规模问题，并通常通过网络访问的计算机

1.1.2 ［2］<1.1>10^{15}或2^{50}字节

1.1.3 ［2］<1.1>由成百上千个处理器和若干 terabyte 级内存组成的计算机

1.1.4 ［2］<1.1>目前尚为科学幻想的应用，但是很可能即将成为现实

1.1.5 ［2］<1.1>一种称为随机访问内存的内存

1.1.6 ［2］<1.1>一种称为中央处理器的计算机部件

1.1.7 ［2］<1.1>上千个处理器形成的大集群

1.1.8 ［2］<1.1>在同一个芯片中含有几个处理器的微处理器

1.1.9 ［2］<1.1>没有显示器和键盘的桌面计算机，通常通过网络访问

1.1.10 ［2］<1.1>当今最大的一类计算机，运行一个应用或一组相关的应用

1.1.11 ［2］<1.1>用于描述硬件部件的特殊语言

1.1.12 ［2］<1.1>对单用户以低成本提供高性能的个人计算机

1.1.13 ［2］<1.2>将高级语言语句翻译成汇编语言的程序

1.1.14 ［2］<1.2>将符号指令翻译成二进制指令的程序

1.1.15 ［2］<1.1>商业数据处理用的高级语言

1.1.16 ［2］<1.1>处理器能够理解的二进制语言

1.1.17 ［2］<1.1>处理器能够理解的命令

1.1.18 ［2］<1.1>科学计算用的高级语言

1.1.19 ［2］<1.1>机器指令的符号表示

1.1.20 ［2］<1.1>用户程序和硬件之间的接口，能提供许多服务和监视功能

1.1.21 ［2］<1.1>用户开发的软件/程序

1.1.22 ［2］<1.1>二进制位（值为 0 或 1）

1.1.23 ［2］<1.1>应用软件和硬件之间的软件层，包括操作系统和编译程序

1.1.24 ［2］<1.1>用于编写应用程序和系统软件的高级语言

1.1.25 ［2］<1.1>由字和代数符号组成的可移植语言，在计算机中运行前必须翻译成汇编程序

1.1.26 ［2］<1.1>10^{12}或2^{40}字节

习题 1.2

1.2.1 ［10］<1.3>一个彩色显示器中的每个像素由三种基色（红，绿，蓝）构成，每种基色用 8 位表示，

分辨率为 1280 × 800 像素。请问为了保存一帧图像需要多大的缓存（以字节计算）?

1.2.2 [5] <1.3> 如果一台计算机有一个 2 GB 的主存，并且该主存没有存储其他信息，它最多可保存多少帧图像?

1.2.3 [5] <1.3> 如果一台计算机连到 1 Gbps 以太网上，发送 256 KB 的文件需要多长时间?

1.2.4 [5] <1.3> 假定 cache 比 DRAM 快 10 倍，DRAM 比磁盘快 100 000 倍，闪存比磁盘快 1000 倍。如果从 cache 读取一个文件需要 2 微秒，请问从 DRAM、磁盘和闪存读取需要多长时间?

习题 1.3

有3 种不同的处理器 P1、P2 和 P3 执行同样的指令集，其时钟频率和 CPI 如下表：

处理器	时钟频率	CPI
P1	2 GHz	1.5
P2	1.5 GHz	1.0
P3	3 GHz	2.5

1.3.1 [5] <1.4> 哪个处理器性能最高?

1.3.2 [5] <1.4> 如果每个处理器执行一个程序都花费 10 秒钟时间，求它们的时钟周期和指令数。

1.3.3 [10] <1.4> 我们试图把时间减少 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?

以下的习题采用下表的信息。

处理器	时钟频率	指令数	时间
P1	2 GHz	20×10^9	7 s
P2	1.5 GHz	30×10^9	10 s
P3	3 GHz	90×10^9	9 s

1.3.4 [10] <1.4> 求每个处理器的 IPC（每周期的指令数）。

1.3.5 [5] <1.4> 求 P2 的时钟频率为多少才能将其执行时间减少到与 P1 的一样?

1.3.6 [5] <1.4> 求 P2 的指令数为多少才能将其执行时间减少到与 P3 的一样?

习题 1.4

同一个指令集体系结构有 2 种不同的实现方式。有 A，B，C 和 D 4 类指令，每种实现方式的时钟频率和 CPI 由下表给定。

	时钟频率	CPI（A 类）	CPI（B 类）	CPI（C 类）	CPI（D 类）
P1	1.5 GHz	1	2	3	4
P2	2 GHz	2	2	2	2

1.4.1 [10] <1.4> 给定一个程序，有 106 条指令，按如下比例分为 4 类：A 10%；B 20%；C 50%；D 20%。问哪种实现方式更快?

1.4.2 [5] <1.4> 每种实现方式总的 CPI 是多少?

1.4.3 [5] <1.4> 两种情况下的时钟周期各是多少?

下表所示为某一程序的指令数。

算术	存储	取数	分支	总数
500	50	100	50	700

1.4.4 [5] <1.4>假定算术指令用时 1 个周期，存储和取数用时各 5 个周期，分支用时 2 周期。该程序在 2 GHz的 CPU 上运行，执行时间是多少?

1.4.5 [5] <1.4>求该程序的 CPI 是多少?

1.4.6 [10] <1.4>如果取数指令减少一半，则增速和 CPI 分别是多少?

习题 1.5

有 2 种不同的实现方式 P1 和 P2，具有同样的指令集，分为 A，B，C，D 和 E 5 类。每类的时钟频率和 CPI 由下表给定。

		时钟频率	CPI（A 类）	CPI（B 类）	CPI（C 类）	CPI（D 类）	CPI（E 类）
a	P1	1.0 GHz	1	2	3	4	3
a	P2	1.5 GHz	2	2	2	4	4
b	P1	1.0 GHz	1	1	2	3	2
b	P2	1.5 GHz	1	2	3	4	3

1.5.1 [5] <1.4>假定将一台计算机执行任何指令序列所能达到的最快速率定义为峰值性能。求 P1，P2 的峰值性能，单位为每秒指令数。

1.5.2 [5] <1.4>如果在某个程序中执行的指令数均等地分布于除 A 类以外的各类指令，A 类指令数是其他类指令数的 2 倍。问：哪台计算机速度更快？快多少?

1.5.3 [5] <1.4>如果某个程序中执行的指令数均等地分布于除 E 类以外的各类指令，E 类指令数是其他类指令数的 2 倍。问：哪台计算机速度更快？快多少?

		指令数				
		计算	取数	存数	分支	总数
a	程序 1	1000	400	100	50	1550
b	程序 4	1500	300	100	100	1750

1.5.4 [5] <1.4>假定计算指令用时 1 周期，取数、存数指令用时 10 周期，分支指令用时 3 周期，求：每个程序在一台 3 GHz MIPS 处理器上运行的执行时间。

1.5.5 [5] <1.4>假定计算指令用时 1 周期，取数、存数指令用时 2 周期，分支指令用时 3 周期，求：每个程序在一台 3 GHz MIPS 处理器上运行的执行时间。

1.5.6 [5] <1.4>假定计算指令用时 1 周期，取数、存数指令用时 10 周期，分支指令用时 3 周期，如果计算指令数减半，求程序的增速（speed-up）。

习题 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。本题将探究编译程序对执行时间的影响。

	编译程序 A		编译程序 B	
	指令数	执行时间	指令数	执行时间
a.	1.00E +09	1 s	1.20E +09	1.4 s
b.	1.00E +09	0.8 s	1.20E +09	0.7 s

1.6.1 [5] <1.4>同样的程序，采用 2 个不同的编译程序。上表表示了不同情况下的执行时间。求：在给定处理器时钟周期为 1 ns 时，每个程序的平均 CPI。

1.6.2 [5] <1.4>假定平均 CPI 就是上题求得的值，但编译程序是在 2 个不同的处理器上运行的。如果这 2

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？

1.6.3 [5] <1.4> 假设开发了一种新的编译程序，只用 6 亿条指令，平均 CPI 为 1.1。求这种新的编译程序相对于 1.6.1 题中编译程序 A 和 B 的加速比。

有同一指令集的 2 个实现方式 P1 和 P2。指令集中有 5 类指令（A，B，C，D，E）。P1 的时钟频率为 4 GHz，P2 的时钟频率为 6 GHz。每类指令相对于 P1 和 P2 的平均周期数见下表。

	类	CPI-P1	CPI-P2		类	CPI-P1	CPI-P2
a.	A	1	2	b.	A	1	2
	B	2	2		B	1	2
	C	3	2		C	1	2
	D	4	4		D	4	4
	E	5	4		E	5	4

1.6.4 [5] <1.4> 假定将一台计算机执行任何指令序列所能达到的最快的速率定义为峰值性能。求 P1，P2 的峰值性能，单位为每秒指令数。

1.6.5 [5] <1.4> 如果在某个程序中执行的指令数均等地分布于除 A 类以外的各类指令，A 类指令数是其他类指令数的 2 倍。求 P2 比 P1 快多少？

1.6.6 [5] <1.4> 上题中，P2 的频率为多少时，其性能和 P1 相同？

习题 1.7

下表表示 28 年来 Intel 公司 8 代处理器的时钟频率和功耗的增长。

处理器	时钟频率	功耗
80286（1982）	12.5 MHz	3.3 W
80386（1985）	16 MHz	4.1 W
80486（1989）	25 MHz	4.9 W
Pentium（1993）	66 MHz	10.1 W
Pentium Pro（1997）	200 MHz	29.1 W
Pentium 4 Willamette（2001）	2 GHz	75.3 W
Pentium 4 Prescott（2004）	3.6 GHz	103 W
Core 2 Ketsfield（2007）	2.667 GHz	95 W

1.7.1 [5] <1.5> 分别求出相邻 2 代处理器的时钟频率之比和功耗之比的几何平均值。

1.7.2 [5] <1.5> 分别求出不同两代间的处理器的时钟频率和功耗的相对变化量的最大值。

1.7.3 [5] <1.5> 分别求出最后一代处理器比第一代处理器在时钟频率和功耗上增长的倍数。

考虑下列各代处理器的电压值。

处理器	电源电压值	处理器	电源电压值
80286（1982）	5	Pentium Pro（1997）	3.3
80386（1985）	5	Pentium 4 Willamette（2001）	1.75
80486（1989）	5	Pentium 4 Prescott（2004）	1.25
Pentium（1993）	5	Core 2 Ketsfield（2007）	1.1

1.7.4 [5] <1.5> 假定静态功耗不计，求平均电容负载。

1.7.5 [5] <1.5> 相邻哪两代处理器间电压相对变化最大？

1.7.6 [5] <1.5> 求从 Pentium 型号开始的不同代处理器间的电压比的几何平均值。

习题 1.8

假设我们开发了处理器的新版本，其特性如下

版本	电压	时钟频率
第 1 版	5 V	0.5 GHz
第 2 版	3.3 V	1 GHz

1.8.1 [5] <1.5> 如果两版之间动态功耗减少了 10%，求电容负载减少了多少？

1.8.2 [5] <1.5> 如果电容负载不变，动态功耗减少了多少？

1.8.3 [5] <1.5> 假定第 2 版的电容负载是第 1 版的 80%，如果第 2 版的动态功耗与第 1 版相比减少了 40%，求第 2 版的电压。

假定工业发展趋势显示新一代处理器的生产比例如下：

电容	电压	时钟频率	面积
1	$1/2^{-1/4}$	$2^{1/2}$	$2^{-1/2}$

1.8.4 [5] <1.5> 动态功耗的比例因子是多少？

1.8.5 [5] <1.5> 求单位面积的电容的比例。

1.8.6 [5] <1.5> 用 1.7 题的数据，求下一代双核处理器的电压和时钟频率。

习题 1.9

虽然动态功耗是 CMOS 功耗的主要组成部分，但漏电流还是会产生静态功耗 $V \times I_{漏}$。芯片尺寸越小，静态功耗越重要。下表设定了若干代处理器的静态和动态功耗的数据。

	工艺	动态功耗	静态功耗	电压
a.	250 nm	49 W	1 W	3.3 V
b.	90 nm	75 W	45 W	1.1 V

1.9.1 [5] <1.5> 求静态功耗占总功耗的百分比。

1.9.2 [5] <1.5> 如果静态功耗取决于漏电流，求每种工艺的漏电流。

1.9.3 [5] <1.5> 求每种工艺的静态功耗与动态功耗之比。

下表表示某一处理器的两种不同版本在 3 种电压下的动态功耗。

	1.2 V	1.0 V	0.8 V
a.	80 W	70 W	40 W
b.	65 W	55 W	30 W

1.9.4 [5] <1.5> 设静态与动态功耗之比为 0.6，求每种版本在 0.8 V 电压时的静态功耗。

1.9.5 [5] <1.5> 求每种版本在 0.8 V 电压时的漏电流。

1.9.6 [5] <1.5> 设静态与动态功耗之比为 1.7，问在 1.0 V 电压和 1.2 V 电压两种情况下，哪种情况的静态功耗更大？

习题 1.10

下表表示一个给定的应用分别在 1，2，4，8 个处理器上运行时指令类型的分配。用这些数据，你可以探究应用在并行处理器上的加速比。

	处理器	每处理器的指令数			CPI		
		计算	取数/存数	分支	计算	取数/存数	分支
a.	1	2560	1280	256	1	4	2
	2	1280	640	128	1	4	2
	4	640	320	64	1	4	2
	8	320	160	32	1	4	2
b.	1	2560	1280	256	1	4	2
	2	1350	800	128	1	6	2
	4	800	600	64	1	9	2
	8	600	500	32	1	13	2

1.10.1 [5] <1.4，1.6> 上表表示完成一个程序运行，每个处理器执行的指令数。每个处理器执行的总指令数是多少？所有处理器执行的总指令数是多少？

1.10.2 [5] <1.4，1.6> 上表的右边表示 CPI 值，设定每个处理器的时钟频率为 2 GHz。求这个程序在 1，2，4，8 台处理器上运行的总时间？

1.10.3 [10] <1.4，1.6> 如果计算指令的 CPI 增加 1 倍，那么会对这个程序在 1，2，4，8 台处理器上的执行时间有何影响？

下表表示一个多核处理器的每处理器核数，每核的指令数，以及在 1，2，4，8 个核上执行程序时的平均 CPI 值。用这些数据你可以探究多核处理器的加速比。

	每处理器的核数	每核的指令数	平均 CPI
a.	1	1.00E+10	1.2
	2	5.00E+9	1.3
	4	2.50E+9	1.5
	8	1.25E+9	1.8
b.	1	1.00E+10	1.2
	2	5.00E+9	1.2
	4	2.50E+9	1.2
	8	1.25E+9	1.2

1.10.4 [10] <1.4，1.6> 设时钟频率为 3 GHz，用 1，2，4，8 个核时，分别求该程序的执行时间。

1.10.5 [10] <1.4，1.6> 设一个处理器核的功耗计算公式为

$$\text{功耗} = (5.0\ \text{mA/MHz}) \times \text{电压}^2$$

其中，处理器的工作电压的计算公式为

$$\text{电压} = \text{频率}/5 + 0.4$$

其中，频率的单位为 GHz。所以在 5 GHz 时，电压是 1.4 V。设每个核工作在 3 GHz 的时钟频率，求该程序分别在 1，2，4，8 个核上执行时的功耗。类似地，求核工作在 500 MHz 的时钟频率时程序分别在 1，2，4，8 个核上执行时的功耗？

1.10.6 [10] <1.4，1.6> 根据 1.10.5 中的公式，计算程序在 3 GHz 和 500 MHz 的时钟频率下分别在 1，2，4，8 核上执行时消耗的能量。

习题 1.11

下表所示为两种处理器的制造数据。

	晶圆直径（cm）	芯片数/晶圆	瑕疵数/单位面积（cm^2）	价格/晶圆
a.	15	90	0.018	10
b.	25	140	0.024	20

1.11.1 [10] <1.7> 分别求出每种芯片的成品率。

1.11.2 [5] <1.7> 分别求出每种芯片价格。

1.11.3 [10] <1.7> 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。

假设随着电子器件制造技术的进步，成品率变化如下表所示。

	技术 1	技术 2	技术 3	技术 4
成品率	0.85	0.89	0.92	0.95

1.11.4 [10] <1.7> 给定芯片面积为 200 mm^2，求每一种技术下单位面积的瑕疵数。

1.11.5 [5] <1.7> 用图表表示成品率和单位面积的瑕疵数的变化。

习题 1.12

下表表示一个 SPEC2006 基准程序在 AMD Barcelona 芯片上运行的结果。

	名字	指令数 $\times 10^9$	执行时间（s）	参考时间（s）
a.	perl	2118	500	9770
b.	mcf	336	1200	9120

1.12.1 [5] <1.7> 如时钟周期时间为 0.333 ns，求 CPI 值。

1.12.2 [5] <1.7> 求 SPEC 的比值。

1.12.3 [5] <1.7> 求这两个基准程序的几何平均值。

下表显示的还是基准程序的数据。

	名字	CPI	时钟频率	SPEC 比
a.	sjeng	0.96	4 GHz	14.5
b.	omnetpp	2.94	4 GHz	9.1

1.12.4 [5] <1.7> 如果基准程序的指令数增加 10%，CPI 不变，求 CPU 时间增加多少?

1.12.5 [5] <1.7> 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?

1.12.6 [5] <1.7> 根据上题中指令数和 CPI 的变化，求 SPEC 比值的变化。

习题 1.13

假设我们要开发 AMD 处理器的 4 GHz 的新版本。我们在指令集中增加一些指令，使习题 1.12 中的基准程序的指令数减少了 15%，得到的执行时间如下表所示。

	名字	执行时间（s）	参考时间（s）	SPEC 比
a.	perl	450	9770	21.7
b.	mcf	1150	9120	7.9

1.13.1 [10] <1.8> 求新的 CPI 值。

1.13.2 [10] <1.8> 一般来说，新的 CPI 大于前面同样基准程序的 CPI，这主要是由于两种情况的时钟频率不同，分别为 3 GHz 和 4 GHz。试回答 CPI 的增长与时钟频率的增长是否相似？如果不相似，请说明原因。

1.13.3 [5] <1.8> CPU 时间减少了多少?

下表还是有关基准程序的数据。

	名字	执行时间（s）	CPI	时钟频率（GHz）
a.	sjeng	820	0.96	3
b.	omnetpp	580	2.94	3

1.13.4 [10] <1.8> 如执行时间再减少 10%，CPI 不变，时钟频率为 4 GHz，求指令数。

1.13.5 [10] <1.8> 要使 CPU 时间再减少 10%，时钟频率应为多少？（指令数和 CPI 不变。）

1.13.6 [10] <1.8> 如果 CPI 减少 15%，CPU 时间减少 20%，指令数不变，求时钟频率。

习题 1.14

1.8 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，下表是在不同的处理器中执行10^6 条指令序列的有关数据。

处理器	时钟频率	CPI
P1	4 GHz	1.25
P2	3 GHz	0.75

1.14.1 [5] <1.8> 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。

1.14.2 [10] <1.8> 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 10^6 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？

1.14.3 [10] <1.8> 一个常见的错误是用 MIPS（每秒百万条指令）来比较 2 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。

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 MFLOPS = 浮点操作的个数/执行时间 $\times 10^6$。它与 MIPS 有同样的问题。考虑下表所示的程序，在时钟频率为 3 GHz 的处理器上运行。

	指令数	读/写	浮点	分支	CPI（读/写）	CPI（浮点）	CPI（分支）
程序 a	10^6	50%	40%	10%	0.75	1	1.5
程序 b	3×10^6	40%	40%	20%	1.25	0.70	1.25

1.14.4 [10] <1.8> 求程序的 MFLOPS 值。

1.14.5 [10] <1.8> 求程序的 MIPS 值。

1.14.6 [10] <1.8> 求程序的性能，并与 MFLOPS 和 MIPS 值作比较。

习题 1.15

1.8 节引证的另一个易犯的错误是通过只改进计算机的一个方面来改进计算机的总体性能。这是可行的，但并不总是可行。下表表示一台计算机运行程序的 CPU 时间。

	浮点指令	整数指令	读/写指令	分支指令	总时间
a.	35 s	85 s	50 s	30 s	200 s
b.	50 s	80 s	50 s	30 s	210 s

1.15.1 [5] <1.8> 如浮点指令减少 20%，总时间将减少多少？

1.15.2 [5] <1.8> 如总时间减少 20%，整数指令时间将减少多少？

1.15.3 [5] <1.8> 如只减少分支指令时间，总时间能否减少 20%？

下表表示一个应用在不同数目的处理器中运行时，每个处理器的不同类型的指令数分布。

	处理器数	浮点指令	整数指令	读/写指令	分支指令	CPI（浮点）	CPI（整数）	CPI（读/写）	CPI（分支）
a.	1	560×10^6	2000×10^6	1280×10^6	256×10^6	1	1	4	2
b.	8	80×10^6	240×10^6	160×10^6	32×10^6	1	1	4	2

设每个处理器的时钟频率为 2 GHz。

1.15.4 [10] <1.8> 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？

1.15.5 [10] <1.8> 如果我们要将程序运行速度提高至 2 倍，读写指令的 CPI 需如何改进？

1.15.6 [5] <1.8> 如果整数和浮点指令的 CPI 减少 40%，读写和分支指令的 CPI 减少 30%，程序的执行时间能改进多少？

习题 1.16

还有一个易犯的错误是有关在多处理器系统中运行，希望只改进一部分例行程序来改进整体性能。下表表示某个程序的 5 个例程在不同数目处理器中的执行时间。

	处理器数	例程 A（ms）	例程 B（ms）	例程 C（ms）	例程 D（ms）	例程 E（ms）
a.	2	20	80	10	70	5
b.	16	4	14	2	12	2

1.16.1 [10] <1.8> 求总的执行时间。如果例程 A，C 和 E 的时间改进 15%，总的执行时间能减少多少？

1.16.2 [10] <1.8> 如果例程 B 的时间改进 10%，总的执行时间能减少多少？

1.16.3 [10] <1.8> 如果例程 D 的时间改进 10%，总的执行时间能减少多少？

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。下表给出了例程计算时间和通信时间。在这种情况下，通信时间是总时间的重要组成部分。

处理器数	例程 A（ms）	例程 B（ms）	例程 C（ms）	例程 D（ms）	例程 E（ms）	通信（ms）
2	20	78	9	65	4	11
4	12	44	4	34	2	13
8	1	23	3	19	3	17
16	4	13	1	10	2	22
32	2	5	1	5	1	23
64	1	3	0.5	1	1	26

1.16.4 [10] <1.8> 每当处理器数量加倍时，求新的计算时间与旧的计算时间之比和新与旧的通信时间之比。

1.16.5 [5] <1.8> 用比值的几何平均值，推算在 128 台处理器的系统中的计算时间和通信时间。

1.16.6 [10] <1.8> 求在 1 个处理器系统中的计算时间和通信时间。

小测验答案

1.1 节　问题讨论：可以有多种答案。

1.3 节　盘内存：非易失性，访问时间长（ms 级），价格（$0.20 ~ $2.00)/GB。半导体内存：易失性，访问时间短（ns 级），价格（$20 ~ $75)/GB。

1.4.2 节　1）A：两者，B：延迟，C：都不改进；2）7 秒。

1.4.5 节　B。

1.7.2 节　A、C 和 D 是正确理由。理由 E 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。

1.8 节　1）计算机 A 有较高的 MIPS 值；2）计算机 B 更快。

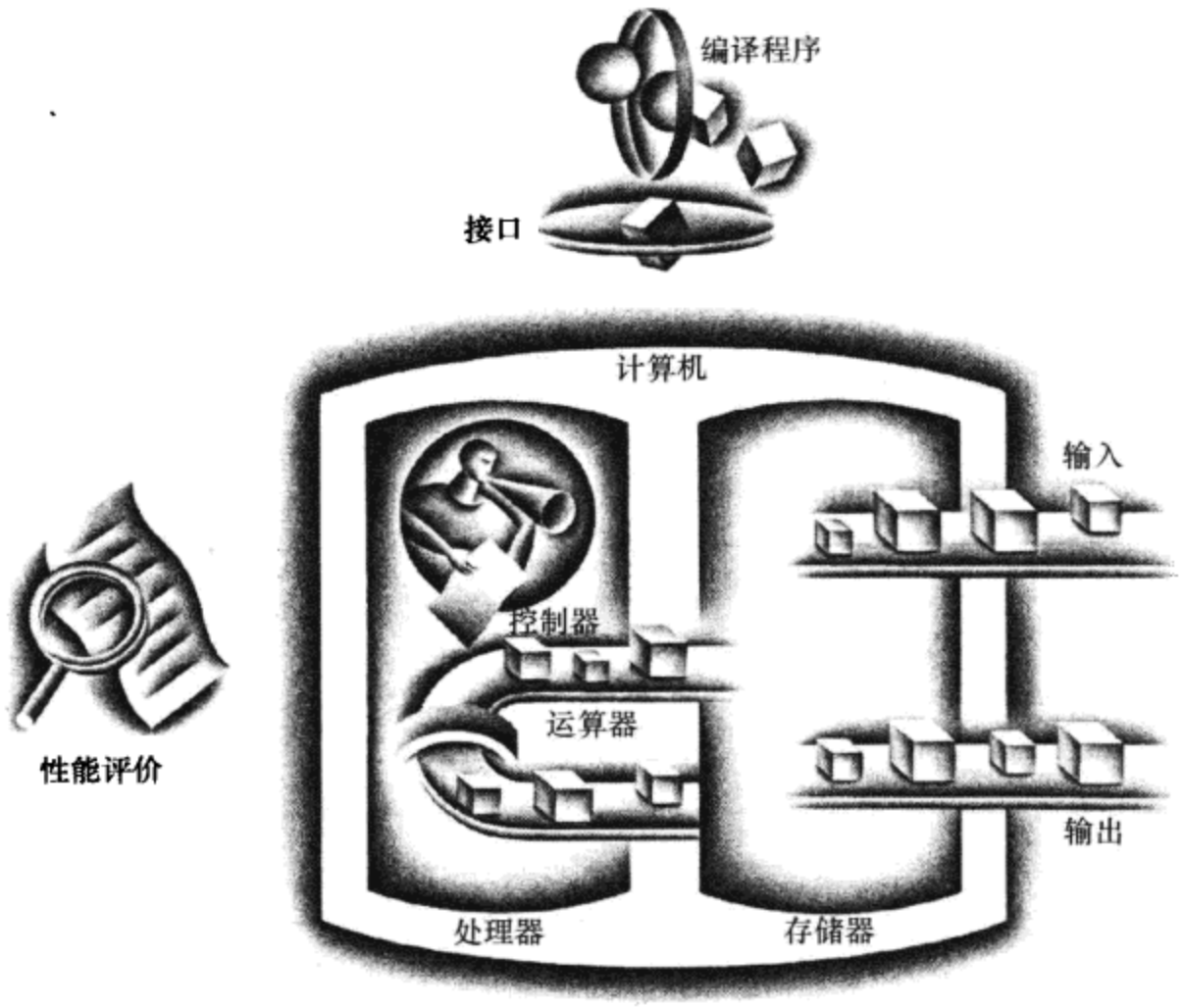
第2章

Computer Organization and Design: The Hardware/Software Interface, 4E

指令：计算机的语言

我对上帝说西班牙语，对女人说意大利语，对男人说法语，对我的马说德语。

——法国国王查理五世（1337—1380）



计算机的五大经典部件

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**[⊖]。本章将展现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。

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，其他的也就容易理解了。这种相似性一方面是因为所有计算机都是基于相似基本原理的硬件技术所构建的，另一方面是因为有一些基本操作是所有计算机都必须提供的。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、成本和功耗最低。但这个目标需要长期的探索。下述引文写于

⊖ 指令集（instruction set）：一个给定的计算机体系结构所包含的指令集合。

计算机出现不久的1947年，但今天它仍然适用：

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

Burks、Goldstine和冯·诺依曼（von Neumann），1947

无论是对20世纪50年代的计算机而言，还是对现代的计算机来说，“设备简单性”都是值得考虑的重要问题。本章的目的就是讲解符合此原则的一种指令集，介绍它怎样用硬件表示，以及它和高级编程语言之间的关系。我们的示例使用C语言编写，光盘上的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**[⊖]。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，读者将更加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概括指令集的历史发展和其他的计算机“方言”。

本章所选指令集来自MIPS公司，它是20世纪80年代以来出现的各类指令集的优秀代表之一。然后，我们将简略介绍另外两个常见的指令集。一个是ARM指令集，它与MIPS非常相似，2008年ARM处理器在嵌入式设备中的使用量超过了30亿个。另一个是Intel x86指令集，2008年所销售的3.3亿台PC中大多安装了Intel处理器。

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、结合部件及其说明的方法，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。

2.2　计算机硬件的操作

毫无疑问，必须有执行基本算术运算操作的指令。

Burks、Goldstine和冯·诺依曼，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，并且有且仅有三个变量。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四个变量的相加：

```
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.
```

可知，使用3条指令完成了4个变量的相加。

上述代码每一行中，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，注释总是在一行之尾结束。

⊖ 存储程序概念（stored-program concept）：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。

MIPS 操作数

名字	举例	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	加常数
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 +0] = $s1; $s1 =0 or 1	取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

图 2-1　本章要讲解的 MIPS 汇编语言

类别	指令	示例	含义	注释
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	slti $s1,$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 1 0000	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）

这个信息也可以在文前的 MIPS 参考数据的第 1 列中找到。

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，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四条基本原则的第一条：

设计原则 1：简单源于规整。

下面的 2 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。

举例　**把 C 语言中两条赋值语句编译成 MIPS**

本例中 C 程序包含 5 个变量，a、b、c、d 和 e。因为 Java 语言演化自 C 语言，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;
```

C 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
```

举例　**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
```

下一个操作是减法，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：

A. Java

B. C

C. MIPS 汇编语言

精解：为了增强可移植性，Java 最初被设定为依靠软件解释器执行的语言。解释器的指令集被称作 Java字节码（Java bytecode）（参见光盘上 2.15 小节），它与 MIPS 指令集有很大的不同。为使性能与等效功能的 C 程序接近，Java 系统现在的典型做法是将字节码编译成类似 MIPS 这样的机器指令。因为通常 Java 编译过程比 C 靠后，所以 Java 编译器常被称为即时编译器（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

不同于高级语言程序，算术运算指令的操作数是受限的，它们必须来自寄存器。寄存器由硬件直接构建，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”[⊖]。

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。典型的现代计算机如 MIPS 中有 32 个寄存器。（参见光盘上 2.20 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四条基本原则中的第二条：

设计原则 2：越少越快。

大量的寄存器可能会使时钟周期变长，因为需要更远的电信号传输距离。

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，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 到 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，

⊖ 字（word）：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。

$s1……来代表与 C 和 Java 程序中的变量所对应的寄存器；用 $t0，$ti……来代表将程序编译为 MIPS 指令时所需的临时寄存器。

举例　**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。求编译后的 MIPS 代码。

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的结构呢？

回忆一下第 1 章和本章首页图片所描述的计算机的五个组成部分。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，复杂数据结构（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。这些指令叫做**数据传送指令**[⊖]。为了访问存储器中的一个字，指令必须给出存储器**地址**[⊜]。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，存放的值为 10。

将数据从存储器拷贝到寄存器的数据传送指令，通常叫取数指令（load）。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。

举例　**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。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，第一条编译后生成的指令为：

```
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），

⊖ 数据传送指令（data transfer instruction）：在存储器和寄存器之间移动数据的命令。

⊜ 地址（address）：用于在存储器空间中指明特定数据元素位置的值。

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。

很多程序都用到字节类型，且大多数体系结构按字节编址。因此，一个字的地址必和它所包括的四个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。

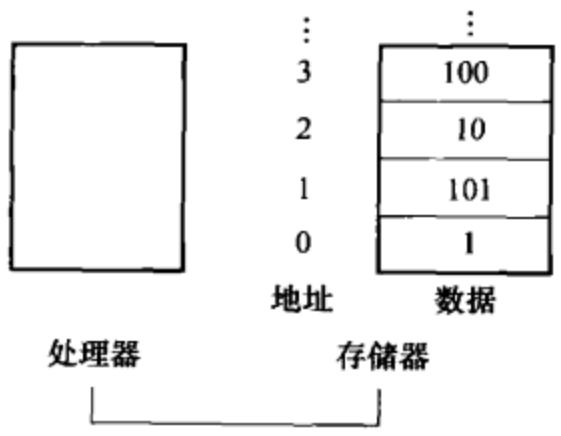


图 2-2　存储器地址和该地址对应的数据

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 个字节。图 2-3给出了顺序字编址的内存寻址。

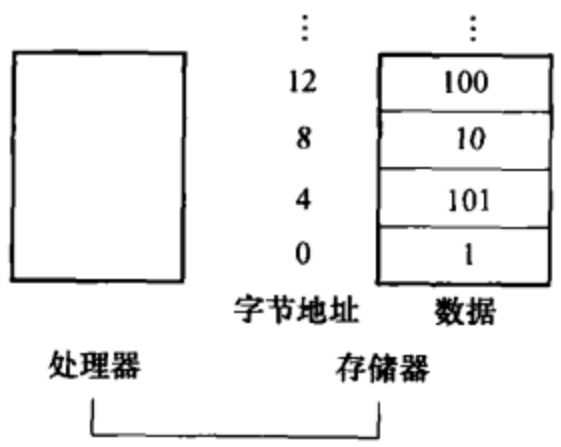


图 2-3　实际的 MIPS 存储器地址和该地址对应的数据

因为 MIPS 是按字节编址的，所以字的地址是 4 的倍数：一个字有 4 个字节。

在 MIPS 中，字的起始地址必须是 4 的倍数。这叫**对齐限制**[⊖]，许多体系结构都有这样的限制（第 4 章说明了地址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**大端编址**（big-endian）。（附录 B 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 2.18 节中相关陷阱介绍。）

与取数指令相对应的指令通常叫做**存数指令**（store）；它将数据从寄存器拷贝到存储器。存数指令的格式和取数指令相似：操作码，接着是包含待存储数据的寄存器，然后是选择数组元素的偏移量，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 存数指令的名字为 sw，为 store word 的缩写。

举例　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

⊖ 对齐限制（alignment restriction）：数据地址与存储器的自然边界对齐的要求。

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拷贝字的指令。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做寄存器溢出（spilling register）。

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、并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，编译器必须高效率地利用寄存器。

2.3.2　常数或立即数操作数

程序中经常会在某个操作中使用到常数——例如，将数组的下标加1，用以指向下一个数组元素。实际上，在运行 SPEC2006 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做加立即数（add immediate），或者写成 addi。这样，上述操作只需写成：

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

带有立即数的指令说明了硬件设计四条原则的第三条，这一原则在第1章的“谬误与陷阱”中曾提到过。

设计原则3：加速执行常用操作。

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。）

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？

A. 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。

B. 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（见光盘）中介绍了MIPS-32和MIPS-64的区别。

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。

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做*下标寄存器*（index register）。现在，存储器容量大大增加，数据分配的软件模型也更为复杂，所以数组的基地址通常放在寄存器中。如同下面将要看到的那样，基地址可能由于过大而不适宜用偏移量表示。

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。人类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二进制数。所有的信息都由**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。

推广到任意进制，第i位d的值是

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

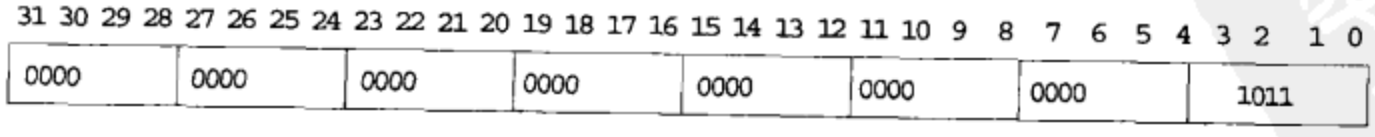
这里，i是从0开始并且从右向左递增。因此一种明显的计算一个数各位数值的方法是使用幂。我们在十进制数的右下角写上10，在二进制数的右下角写上2。

例如，1011_2

表示

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

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

⊖ 二进制数位（binary digit或binary bit）：二进制状态之一，即0或1，是信息的基本组成单位。

最低有效位[⊖]表示最右边的一位（上图中的第0位），**最高有效位**[⊖]50表示最左边的一位（上图中的第31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：

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$
…　　…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 4\,294\,967\,293_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 4\,294\,967\,294_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 4\,294\,967\,295_{10}$

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）：

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

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。

硬件可以被设计成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。

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

符号和幅值方法有若干缺点。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使硬件简单的表达方式：前导位为0表示正数，前导位为1表示负数。这种常用的表示有符号二进制数的方法称为*二进制补码*（two's complement）。例如：

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$
…　　…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 2\,147\,483\,645_{10}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 2\,147\,483\,646_{10}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 2\,147\,483\,647_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2\,147\,483\,648_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = -2\,147\,483\,647_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = -2\,147\,483\,646_{10}$
…　　…

⊖ 最低有效位（least significant bit）：在MIPS字中最右边的一位。
⊖ 最高有效位（most significant bit）：在MIPS字中最左边的一位。

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = -3_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = -1_{10}$

上面的数字中一半是正数，从 0 到 $2\ 147\ 483\ 647_{10}$（$2^{31}-1$），这些数字的表示方式与之前是一样的。紧接着的 $1000\cdots0000_2$ 表示最大的负数 $-2\ 147\ 483\ 648_{10}$（-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}（$1111\cdots1111_2$）。

二进制补码中的最大负数 $-2\ 147\ 483\ 648_{10}$ 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，该方法不会对硬件设计人员造成困扰。因此，现在所有计算机都采用二进制补码方法来表示有符号数。

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

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

符号位被 -2^{31} 乘，其余的位仍按前面的方法计算。

举例　**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：

$$\begin{aligned}&(1 \times 2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \cdots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0})\\ &= -2^{31} + 2^{30} + 2^{29} + \cdots + 2^{2} + 0 + 0\\ &= -2\ 147\ 483\ 648_{10} + 2\ 147\ 483\ 644_{10}\\ &= -4_{10}\end{aligned}$$

后面将给出从负数转换为正数的捷径。

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

硬件 软件接口

与上面所讨论的数不同，存储器地址很自然地从 0 开始一直连续增加到最大的地址。换言之，负地址是没有意义的。因此，程序有时则需要处理一些可以是正也可以是负的数，有时则需要处理一些仅能是正的数。一些编程语言反映了这区别。例如 C 语言将前者叫做整数（int）而后者叫做无符号整数（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捷径。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。这个捷径是基于这样的事实，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 $\bar{x} + 1 = -x$。

举例　**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：

$$
\begin{array}{rl}
 & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 \\
+ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_2 \\
\hline
= & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 \\
= & -2_{10}
\end{array}
$$

另一方面，将

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2$$

也按位取反再加1：

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

第二种捷径用于将一个用 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。这个捷径就是将原有的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。这个捷径通常叫做*符号扩展*（sign extension）。

举例　**符号扩展的捷径**

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）拷贝16次放到32位字的左半部。右半部的16位保持原16位的值：

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

使用前面介绍的捷径对2的16位二进制数求反。于是，

$$0000\ 0000\ 0000\ 0010_2$$

变成

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

将该求反结果转换为32位数的捷径就是将符号位拷贝16次放到32位字的左半部：

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

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

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。

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？

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

A. -4_{10}

B. -8_{10}

C. -16_{10}

D. $18\ 446\ 744\ 073\ 709\ 551\ 609_{10}$

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 2^n，因此，x 的相反数 $-x$ 的二进制补码表示是 $2^n - x$。

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**”[⊖]。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，最大的负数用 $00\cdots000_2$ 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**[⊜]。

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，可以将符号编码进位串中，因此通常不使用“+”和“-”来表示二进制或十六进制数。

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，并且看上去和数的表示是一样的。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，以使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，以此类推；`$t0` 表示寄存器 8，`$t1` 表示寄存器 9，依此类推。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。

举例　**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

⊖ 反码（*one's complement*）：使用 $10\cdots000_2$ 表示最大负数，$01\cdots11_2$ 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（biased notation）：最大的负数用 $00\cdots000_2$ 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。

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和的目的寄存器号（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

指令的布局形式叫做**指令格式**[一]。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**[二]，这样的指令序列叫做机器码（machine code）。

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。由于几乎所有的计算机的数据大小都是 4 的整数倍，因此**十六进制**[三]表示形式变得很流行。由于 16 是 2 的 4 次幂，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	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。

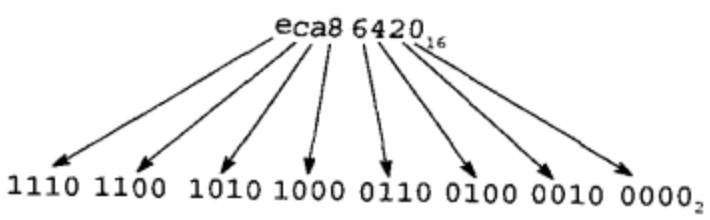
为了避免处理不同进制数时产生混淆，此处约定十进制数加下标 10，二进制数加下标 2，十六进制数加下标 16。（如果没有下标，那么默认为十进制。）顺便说明，C 和 Java 中用符号 0x*nnnn* 来表示十六进制数。

举例　**二进制和十六进制间的转换**

将下面的十六进制数转化成二进制数，二进制数转化成十六进制数：

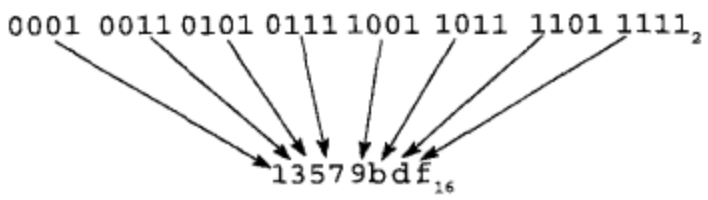
$eca8\ 6420_{16}$
$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_2$

答案　按图 2-4 所示十六进制 - 二进制转换表查表得：



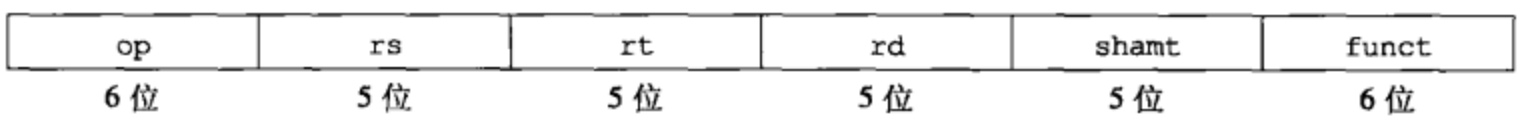
从二进制到十六进制的转换：

㊀ 指令格式（instruction format）：二进制数字段组成的指令表示形式。
㊁ 机器语言（machine language）：在计算机系统中用于交流的二进制表示形式。
㊂ 十六进制（hexadecimal）：基数为 16 的数。



MIPS 字段

为了使讨论变得简单，给 MIPS 指令的字段命名如下：



其中各字段名称及含义如下：

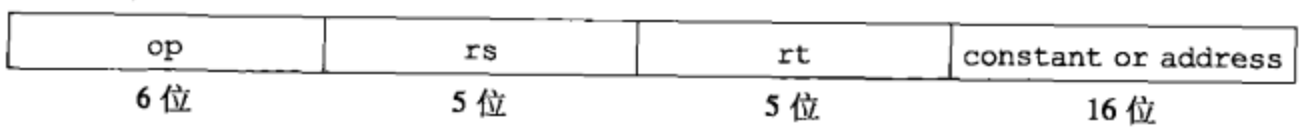
- op：指令的基本操作，通常称为**操作码**[⊖]。
- rs：第一个源操作数寄存器。
- rt：第二个源操作数寄存器。
- rd：用于存放操作结果的目的寄存器。
- shamt：位移量。（在 2.6 节中介绍移位指令和该术语、在此之前，指令都不使用这个字段，故此字段的内容为 0。）
- funct：功能。一般称为*功能码*（function code），用于指明 op 字段中操作的特定变式。

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取元素，所以它常常比 32 大得多。5 位字段因太小而用处不大。

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

设计原则 4：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，设置多于 32 个寄存器是困难的，因为 rs 和 rt 字段都必须增加额外的位，而 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。

⊖ 操作码（opcode）：指令中用来表示操作和格式的字段。

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 型后三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.	constant
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 到 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 sub（34）。

举例　**将 MIPS 汇编语言翻译成机器语言**

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

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

被编译成如下汇编语言：

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

这三条 MIPS 指令的机器语言代码是什么？

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

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

lw 指令的第一个字段（op）值为 35（见图 2-5）。在第二个字段（rs）中指定基址寄存器 9（$t1），在第三个字段（rt）中指定目的寄存器 8（$t0）。在最后一个字段 address 中存放用于指定 A[300]的偏移量（1200 = 300 × 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 1200 用二进制表示为 0000 0100 1011 0000）：

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

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

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，都包含给出基本的操作的 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。

重点

如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，乃至用于生成机器码的编译器。

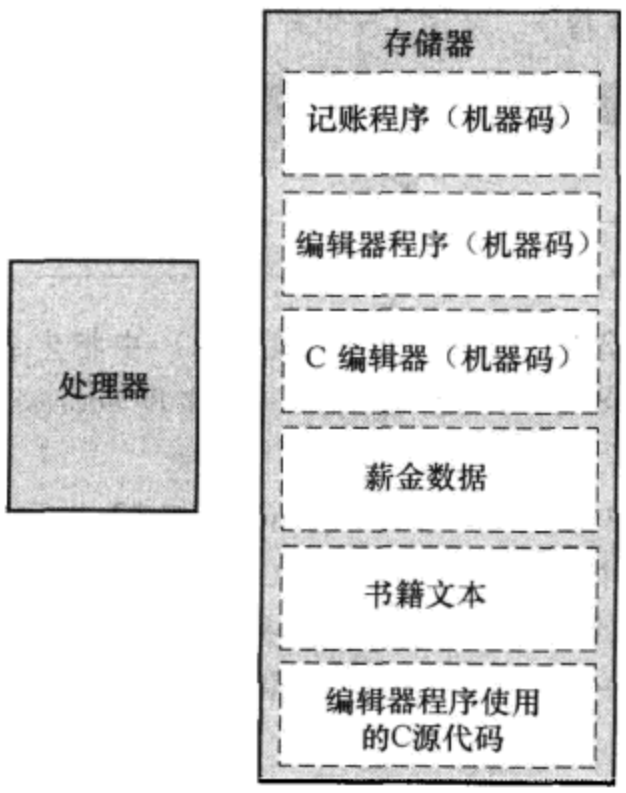


图 2-7　存储程序概念

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在给定的存储器地址开始执行程序即可。将指令和数据以相同的方式处理，极大地简化了计算机系统的存储器硬件和软件。尤其是用于数据的存储技术同样也适用于程序，如编译器，它能够将那些用易于人类使用的符号编写的代码翻译成机器能理解的代码。

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沿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。

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

A. `add $s0,$s1,$s2`

B. `add $s2,$s0,$s1`

C. `add $s2,$s1,$s0`

D. `sub $s2,$s0,$s1`

2.6　逻辑操作

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

Lewis Carroll，《爱丽丝漫游仙境》，1865

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

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

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

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

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_2 = 9_{10}$

一条左移 4 位的指令执行后，得到的新值是：

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_2 = 144_{10}$

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 16（即 2^4）。所以，原二进制数表

示的值是 9，而 $9 \times 16 = 144$，怡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 0000_2

寄存器 $t1 的值为：

0000 0000 0000 0000 0011 1100 0000 0000_2

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 0000_2

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。

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 0000_2

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。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)
```

的执行结果是：

1111 1111 1111 1111 1100 0011 1111 1111_2

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 中出现的很少，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。

精解：MIPS 指令全集也包括异或（XOR），当 2 个操作数对应位不同时置 1，相同时置 0。C 语言允许在字内定义由若干位组成的一个或多个字段，并将其作为对象包装在一个字内，以适应如 *I/O* 设备等的外部接口需求。所有字段必须放在一个单字之中，并采用无符号整数。C 编译器使用 MIPS 的下列逻辑指令插入和提取字段：and、or、sll 以及 srl。

⊖ 按位与（AND）：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（OR）：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（NOT）：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（NOR）：按位先或后非操作，仅当两个操作位均为 0 时结果才为 1。

小测验

下面哪个操作可以将字中的一部分分离出来？

A. AND

B. 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 和 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 2 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 2 条指令传统上称为**条件分支指令**。[⊖]

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，设该五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
```

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，通常叫做**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。

⊖ 条件分支指令（conditional branch）：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。

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:
```

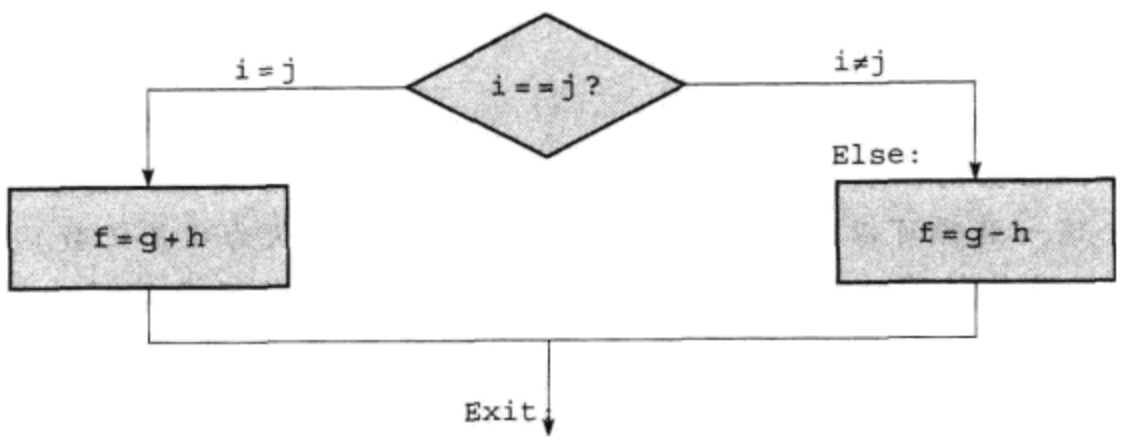


图 2-9 上述 if 语句的程序流程图

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。

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解除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。避免显式的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

决策无论是在二选一的 if 语句中，还是在迭代计算的循环语句中，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。

举例 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]读入到一个临时寄存器中。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（见前面 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]读入到一个临时寄存器中：

```
    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：

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

再下一条指令将 i 加 1：

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

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

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

（见练习题中对该指令序列的优化。）

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。

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

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

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

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

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

硬件/软件接口

MIPS 编译器使用 slt、slti、beq、bne 和固定值 0（总是可以通过读取寄存器 $zero 来获得）来创建所有的比较条件：相等、不等、小于、小于或等于、大于、大于或等于。

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

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。

举例　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

⊖ 基本块（basic block）：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。

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

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 $4\ 294\ 967\ 295_{10} > 1_{10}$。

将有符号数作为无符号数来处理，是一种检验 $0 \leqslant x < y$ 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。

举例 **边界检查的捷径**

利用这个捷径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**[⊖]，这样程序只需索引该表即可跳转到恰当的指令序列。转移地址表是一个由代码中标签所对应地址构成的数组。程序需要跳转的时候首先将转移地址表中适当的项加载到寄存器中，然后使用寄存器中的地址值进行跳转。为了支持这种情况，像 MIPS 这样的计算机提供了寄存器跳转指令 jr（jump register），用来无条件地跳转到寄存器指定的地址。该指令将在下一节中介绍。

硬件/软件接口

虽然在 C 或 Java 这样的编程语言中有许多分支判断和循环语句，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），那么不必对此担心：MIPS 汇编器会使其对汇编语言程序员不可见。

小测验

1）C 语言中有很多分支和循环语句，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？

A. 更多的决策语句使得代码更容易被阅读和理解。

B. 更少的决策语句简化了负责执行的底层工作。

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

C. 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。

D. 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。

2）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？

A. 逻辑操作AND和OR实现 & 和 |，而条件分支实现 && 和 ||。

B. 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。

C. 它们是冗余的并且其实是一回事：&& 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[⊖]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读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，参数担任过程与其他程序、数据之间接口的角色，因为它们能传递数值并返回结果。在CD上的2.15节中，描述了Java语言中过程的等价表示方法，但对计算机的要求，Java与C语言完全相同。

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。

同样地，在过程运行期间，程序必须遵循以下六个步骤：

1）将参数放在过程可以访问到的位置。

2）将控制转移给过程。

3）获得过程所需的存储资源。

4）执行请求的任务。

5）将结果的值放在调用程序可以访问到的位置。

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 ~ $a3`：用于传递参数的四个参数寄存器。
- `$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**[⊜]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**[⊝]。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。

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
```

⊖ 过程（procedure）：根据提供的参数执行一定任务的存储的子程序。

⊜ 跳转和链接指令（jump - and - link instruction）：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（return address）：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。

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**㊀，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**㊁）。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**㊂，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。

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四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，调用者使用的任何寄存器都必须恢复到过程调用前所存储的值。这种情况可以看成是需要将寄存器换出到存储器的一个例子，如“硬件/软件接口”部分所提到的那样。

换出寄存器的最理想的数据结构是**栈**㊃——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。**栈指针**㊄按照每个被保存或恢复的寄存器以字为单位进行调整。MIPS 软件为栈指针保留了第 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**㊅，从栈中移除数据称为**出栈**㊆。

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

举例　**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);
      reture f;
}
```

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

㊀ 调用者（caller）：调用一个过程并为过程提供必要参数值的程序。
㊁ 被调用者（callee）：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。
㊂ 程序计数器（program counter，PC）：包含在程序中正在被执行指令地址的寄存器。
㊃ 栈（stack）：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。
㊄ 栈指针（stack pointer）：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。
㊅ 压栈（push）：向栈中增加元素。
㊆ 出栈（pop）：从栈中移除元素。

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。

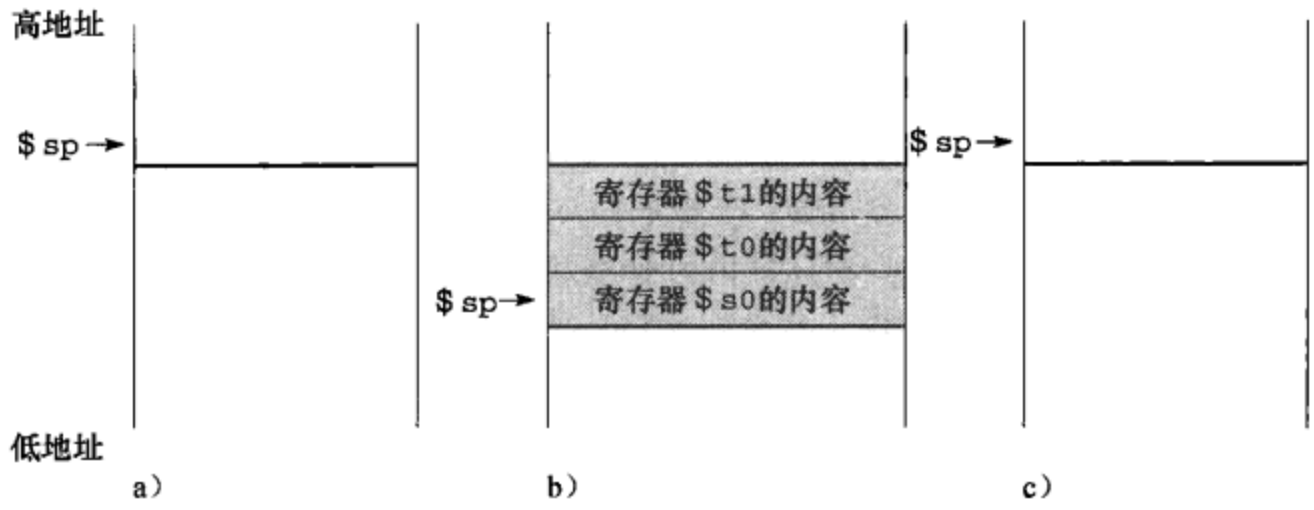


图 2-10　在过程调用之前（a）、之中（b）和之后（c）栈指针以及栈的值
栈指针总是指向栈顶，或者说是图中栈的最后一个字。

接着的三条语句对应过程实体，与 2.2 节的例子相同：

```
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）。

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存寄存器 $t0 和 $t1，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佣其他侦探一样，被雇佣的侦探进而雇佣更多的侦探，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。同样地，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存的寄存器压栈，就像将保留寄存器压栈一样。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 $ra 和被调用者使用的保留寄存器（$s0 ~ $s7）都压栈。栈指针 $sp 随着栈中寄存器个数调整。到返回时，寄存器会从存储器中恢复，栈指针也随着重新调整。

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。然后从栈中退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 caller1
```

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

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

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

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

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

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

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

最后，fact 再次跳转到返回地址：

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

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**[⊖]，即 $gp。

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。

2.8.3　在栈中为新数据分配空间

最后一点复杂性是栈还需要存储相对过程来说是局部的变量，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**[⊜]或**活动记录**。图2-12显示了过程调用之前、之中和之后栈的状态。

某些 MIPS 软件使用**帧指针**[⊝]（$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

⊖ 全局指针（global pointer）：指向静态数据区的保留寄存器。

⊜ 过程帧（procedure frame）：也称作活动记录（activation record），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（frame pointer）：指向给定过程中保存的寄存器和局部变量的值。

$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。

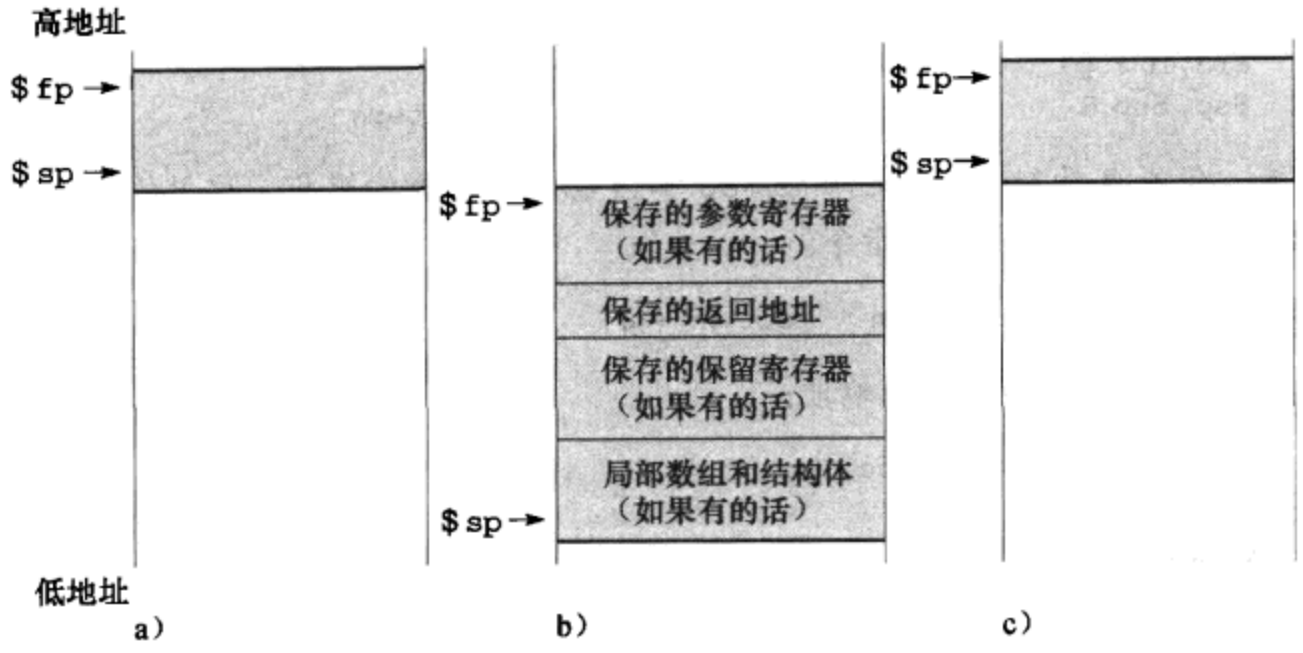


图 2-12 过程调用之前（a）、之中（b）、之后（c）栈的分配情况

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容纳所有的保留寄存器和驻留内存的局部变量。因为在过程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。相关内容可以在本书文前的 MIPS 参考数据的第 4 列找到。

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**正文段**[⊖]。正文段之上的代码为**静态数据段**（static data segment），是存储常量和其他静态变量的空间。尽管数组通常具有固定长度因而能与静态数据段很好地匹配，但类似链表这样的数据结构通常会随生命期增长或缩短。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。

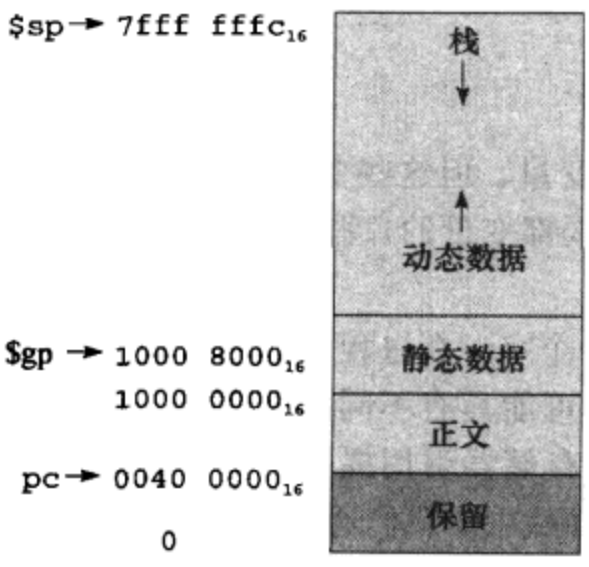


图 2-13 程序和数据的 MIPS 内存分配

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（正文段）从地址 0040 0000$_{16}$ 开始。静态数据从 1000 0000$_{16}$ 开始。然后是动态数据，在 C 中使用 malloc 命令分配，在 Java 中使用 new 命令来分配。动态数据在某一区域中朝着栈的方向向上生长，该区域称为堆。全局指针 $gp 应设置为适当地址以便于访问数据。它初始化为 1000 8000$_{16}$，这样通过相对 $gp 的正负 16 位的偏移量就可以访问从 1000 0000$_{16}$ 到 1000 ffff$_{16}$ 之间的内存空间。关于这点可参见本书文前的 MIPS 参考数据的第 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

⊖ 正文段（text segment）：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。

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

图2-14总结了MIPS汇编语言的寄存器约定。

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	保留寄存器	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。关于这点也可见本书文前的MIPS参考数据的第2列。

精解：如果参数多于四个该怎么办呢？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中获得前四个参数，通过帧指针在内存中寻址获得其余参数。

如图2-12的标题所述，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用做另一个保留寄存器（$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）：

```
sum:slti    $a0,1                    # test if n <=0
    beq     $a0,$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？

A. C程序员显式地管理数据，而在Java中一般是自动的。

B. 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

！（@ | => （wow 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）。

计算机发明是为了数字计算，不过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。

硬件 软件接口

二进制对人类来说不是自然计数方法的，我们有10个手指头，所以我们自然的采用十进制数。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。十进制被证明效率很低，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。

举例　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。

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捷径。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。这些内容也可以在本书文前的MIPS参考数据的第3列中找到。

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( $gp)        # Write byte to destination
```

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，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 将字节看做有符号数，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的字节加载都是使用 lbu。

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 个字节表示，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。）

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留寄存器 $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]
```

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，而是将其用做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。

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 也将半字看做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）将半字看做无符号数，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。

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 个字节压缩为 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。

Latin	Malayalam	Tagbanwa	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 的整数倍。举例来说，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 16～32 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 16～32 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。

小测验

1）下面关于 C 和 Java 中字符和字符串的陈述哪些是正确的？

A. C 中一个字符串占用的内存是 Java 中同样字符串的一半。

B. C 和 Java 中字符串只是一个一维字符数组的非正规名字。

C. C 和 Java 中采用 null（0）来标记字符串的结尾。

D. 对字符串的操作，例如求长度，在 C 中比在 Java 中更快。

2）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？

A. C 语言的 `int`

B. C 语言的 `string`

C. Java 的 `string`

2.10　MIPS 中 32 位立即数和地址的寻址

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。

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。

举例　**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：

0000 0000 0011 1101 0000 1001 0000 0000

lui $t0,255 # $t0 is register 8 的机器码：

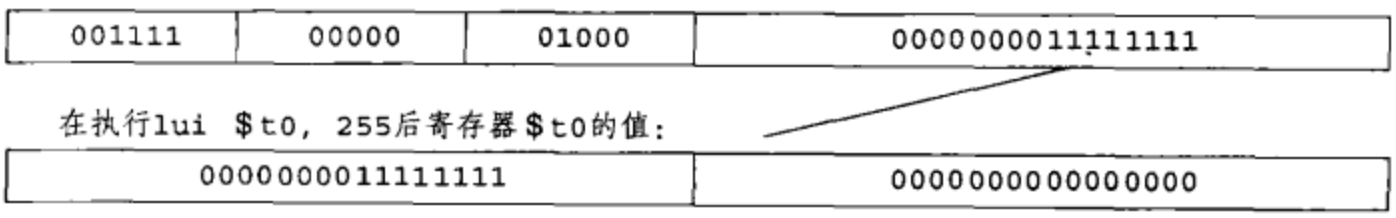


图 2-17 lui 指令的效果

lui 指令将 16 位立即数常量值存放到寄存器的高 16 位，低 16 位用 0 填充。

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选择包括的内容所限（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。

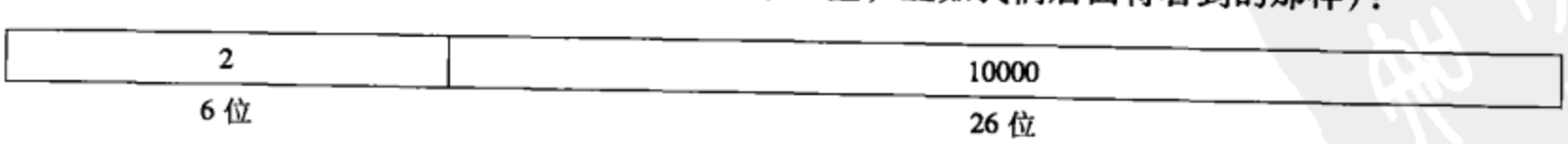
精解： 构造 32 位常数时必须小心。指令 addi 将指令最左边的 16 位立即数字段复制到一个字的高 16 位中。2.6 节的立即数逻辑或操作（logical or immediate）把 0 读到高 16 位中，所以可被汇编程序用于和 lui 一起创建 32 位常数。

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）：



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：

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，这样分支指令可能按如下方式计算：

程序计数器 = 寄存器 + 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^{16}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**⊖。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）。

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很可能接靠近其分支地址。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。

既然所有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。同样地，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。

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位保持改变。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（6400万条指令）；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。

举例　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80000处，那么该循环的MIPS机器代码是什么呢？

答案　汇编指令和它们的地址如下：

⊖ PC相对寻址（PC-relative addressing）：一种寻址方式，它将PC和指令中的常数相加作为寻址结果。

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 个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（20000 × 4 = 80000），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。汇编器的解决方法就像处理对大地址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**[⊖]，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：

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）基址或偏移寻址（base or 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），跳转地址是指令中 26 位字段和 PC 高位相连而成。

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（见光盘中的附录 E）。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。

注意一种操作可以使用不止一种的寻址模式。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（addressing mode）：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。

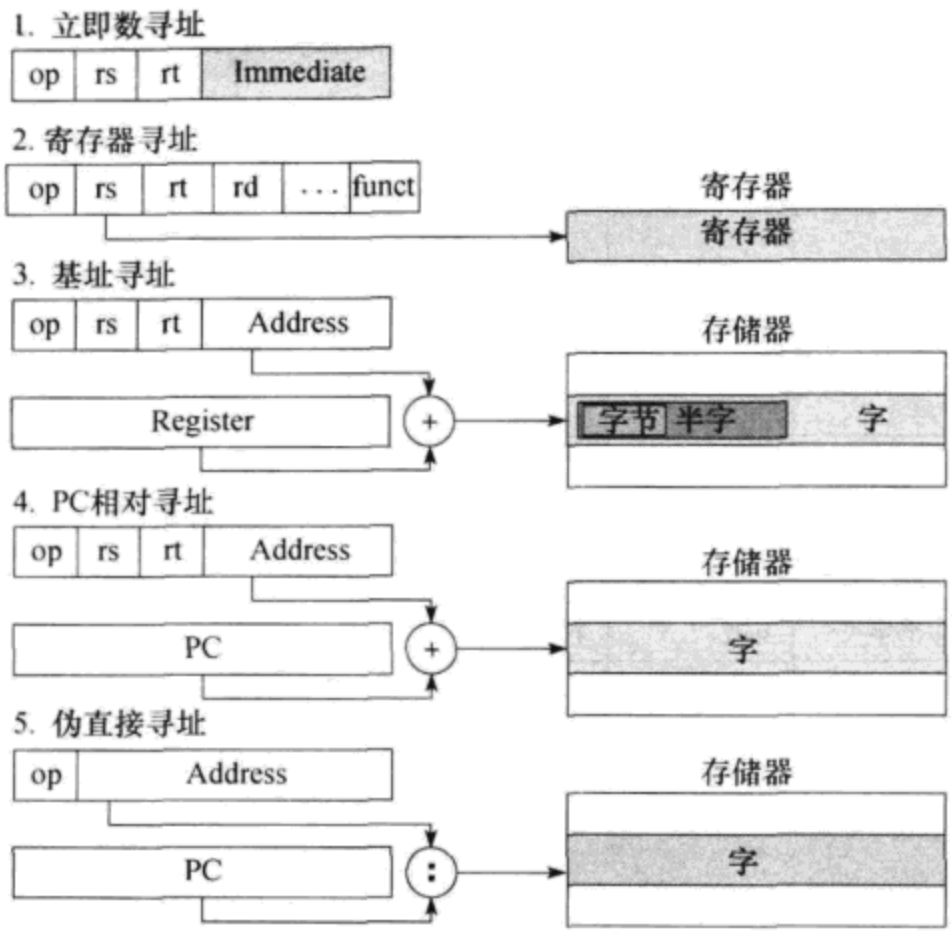


图 2-18　MIPS 5 种寻址模式的说明

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。

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

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 而 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
```

op（31:26）								
28 – 26 31 – 29	0(000)	1(101)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R 型	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	小于立即数置位	小于无符号立即数时置位	andi	ori	xori	取立即数高位
2(010)	TLB	FIPt						
3(011)								
4(100)	取字节	取半字节	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	lwc1						
7(111)	存条件字	swc1						

op（31:26）=010000（TLB），rs（25:21）								
23 – 21 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 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 reg.	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. *unsigned*				
6(110)								
7(111)								

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）第三列（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 在第三张的图 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 B 涵盖所有的指令。

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。

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

小测验

1）在 MIPS 中条件分支的地址范围（K = 1024）是多大？

A. 地址在 0 ~ 64 K − 1 之间

B. 地址在 0 ~ 256 K − 1 之间

C. 分支前后地址范围各大约 32 K

D. 分支前后地址范围各大约 128 K

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

A. 地址在 0 ~ 64 M − 1 之间

B. 地址在 0 ~ 256 M − 1 之间

C. 分支前后地址范围各大约 32 M

D. 分支前后地址范围各大约 128 M

E. 由 PC 提供高 6 位地址的 64 M 大小的块中任意地址

F. 由 PC 提供高 4 位地址的 256 M 大小的块中任意地址

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

A. `J`

B. R 型

C. `addi`

D. `sll`

E. `mfc0`

F. 未定义的操作码：没有对应 0 的合法指令

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**[⊖]的危险，导致读数据错误而引起程序运行结果的改变。

例如，回忆第 1 章 1.6 节所提到的 8 个作者共同写作一个故事的例子。假设一个作者要写总结，他要阅读所有之前的章节。因此，他必须知道其他作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作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。

⊖ 数据竞争（data race）：假如来自不同线程的两个内存地址访问同一个地址，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。

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直接一个仅允许单个处理器操作的区域，叫做*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，提供原子读和原子写存储器单元的能力，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，建立同步库的过程常常是复杂而艰难的。

我们用*原子交换原语*（atomic exchange or 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。发出的竞争交换指令也获得值 0。

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、写操作都是单个的、不可被打断的指令。

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做*链接取数*（load linked）的特殊取数指令和一条叫做*条件存数*（store conditional）的特殊存数指令。我们顺序的使用这两条指令：当由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的话，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。

```
try:add   $t0, $zero, $s4     ;copy exchange value
    ll    $t1,0( $s1)         ;load linked
    sc    $t0,0( $s1)         ;store conditional
    beq   $t0, $zero,try      ;branch store fails
    add   $s4, $zero, $t1     ;put load value in $s4
```

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。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。

精解：尽管我们讲述的同步是在多处理器系统中的，但是原子操作在单个处理器上运行的操作系统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，条件存指令在处理器两条指令之间进行上下文切换（context switch）时也会失败（见第 5 章）。

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问题的只有寄存器 - 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的少，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。

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。

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？

A. 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

B. 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四个步骤，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，这四个步骤是程序转换流程所必经的四个阶段。本节将根据这种翻译层次进行描述。

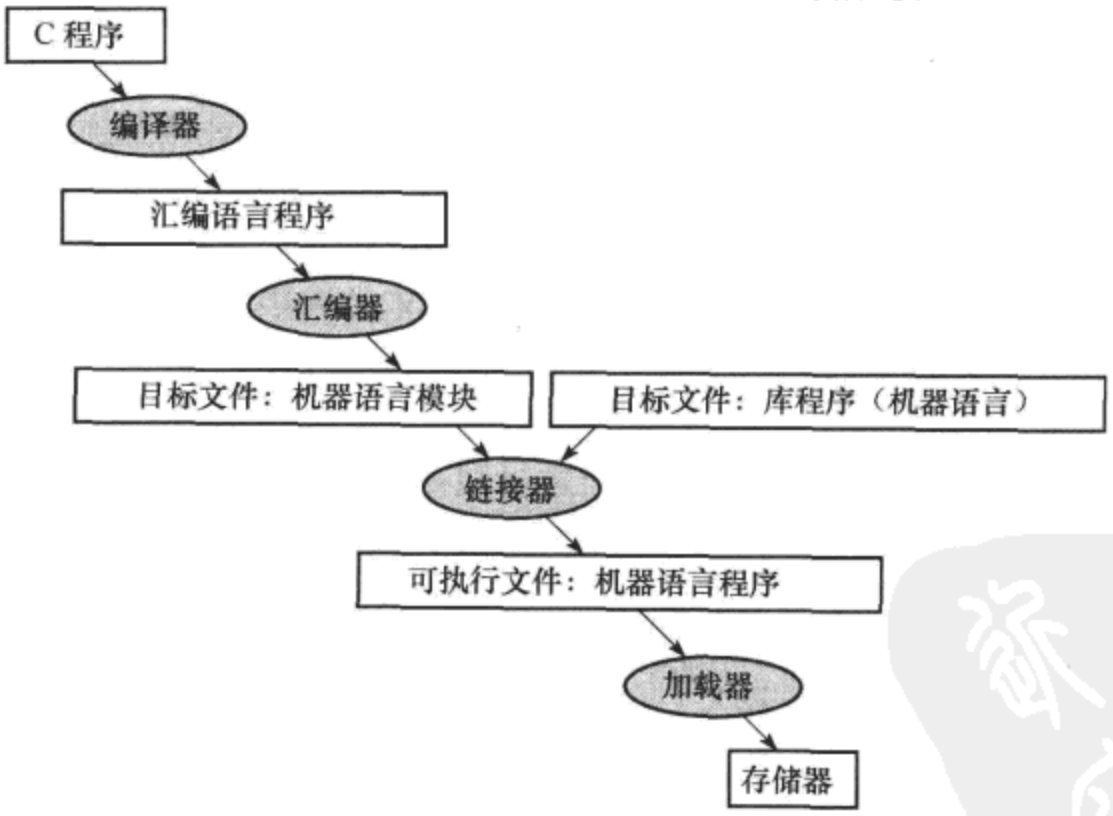


图 2-21　C 语言的翻译层次

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文件。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，一些系统使用带连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**[⊖]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。

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**[⊜]（pseudoinstructions）。

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：

```
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2.10.1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拥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。

UNIX系统中的目标文件通常包含以下六个不同的部分：

- *目标文件头*，描述目标文件其他部分的大小和位置。

⊖ 汇编语言（assembly language）：一种符号语言，能被翻译成二进制的机器语言。
⊜ 伪指令（pseudoinstruction）：汇编语言指令的一个变种，通常被看做一条汇编指令。
⊝ 符号表（symbol table）：一个用来匹配标记名和指令所在内存字的地址的列表。

- 正文段，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。
- 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**链接器**[⊖]，它把所有独立汇编的机器语言程序“拼接”在一起。

链接器的工作分三个步骤：

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另一个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，所有绝时引用（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**[⊜]，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。

举例　**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 X 和 Y 的地址的指令。

⊖ 链接器（linker）：它是一个系统程序，它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。

⊜ 可执行文件（executable file）：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。

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

答案　过程 A 需要找到 load 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，我们可以看到正文段从地址 $400\,000_{16}$ 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正文段大小是 100_{16} 字节而数据段大小是 20_{16} 字节，这样过程 B 的正文段开始地址就是 $40\,0100_{16}$，数据段开始地址是 $1000\,0020_{16}$。

可执行文件头		
	正文大小	300_{16}
	数据大小	50_{16}
正文段	地址	指令
	$0040\,0000_{16}$	lw $a0,$8000_{16}$($gp)
	$0040\,0004_{16}$	jal $40\,0100_{16}$
	…	…
	$0040\,0100_{16}$	sw $a1,$8020_{16}$($gp)

可执行文件头		
	$0040\,0104_{16}$	jal $400\,000_{16}$
	…	…
数据段	地址	
	$1000\,0000_{16}$	(X)
	…	…
	$1000\,0020_{16}$	(Y)
	…	…

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 中表明 $gp 的初始值为 $1000\,8000_{16}$。为了得到地址 $1000\,0000_{16}$（字 X 的地址），我们设置位于地址 $40\,0000_{16}$ 处的 lw 的地址字段中为 8000_{16}。同样，为了得到地址 $1000\,0020_{16}$（字 Y 的地址），可以设置位于地址 $40\,0100_{16}$ 处的 sw 的地址字段中为 8020_{16}。

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**[⊖]按照如下步骤工作：

1）读取可执行文件头来确定正文段和数据段的大小。

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。

附录 B 中的 B.3 小节和 B.4 小节更加详细的描述了链接器和加载器。

2.12.5　动态链接库

本小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 2.5 MB。

这些不足导致了**动态链接库**[⊖]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。

最初版本 DLL 的缺点是它仍将链接库中所有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。

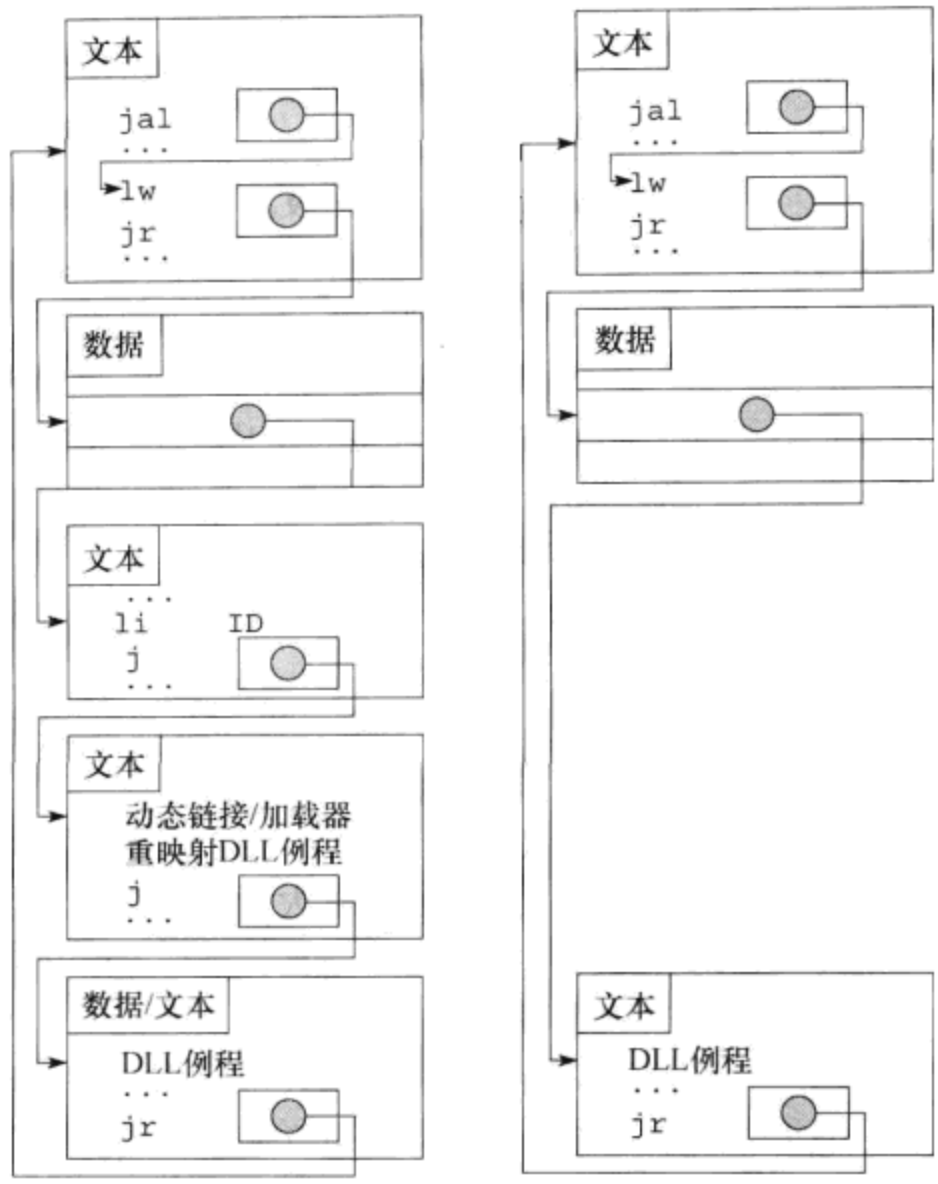
⊖ 加载器（loader）：把目标程序装载到内存中以准备运行的系统程序。

⊖ 动态链接库（*dynamically linked libraries*，DLL）：在程序执行过程中才被链接的库例程。

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。

第一次调用库例程的时候，程序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。



a）首次调用DLL例程　　b）后续调用DLL例程

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等步骤被跳过。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。

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**[⊖]指令集（见 CD 上的 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

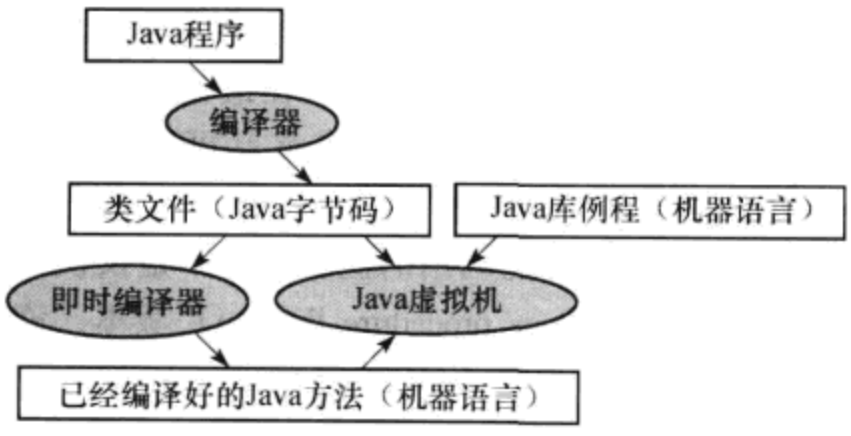


图 2-23　Java 的翻译层次

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函数。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（Java bytecode）：为了解释 Java 程序而设计的指令集中的指令。

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。

一个叫做 **Java 虚拟机**[⊖]（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。

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**[⊖]（JIT）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。

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 与 C ++ 在性能上的差距越来越小。光盘上的 2. 15 节将进一步介绍 Java 程序、Java 字节码、JVM 和 JIT 编程器。

小测验

对 Java 设计者来说，你认为与翻译器相比解释器在哪些方面的优点是最重要的？

A. 解释器便于编写。

B. 更准确的错误消息。

C. 更少的目标代码。

D. 机器独立性。

2. 13　以一个 C 排序程序为例

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看起来是怎样的。本小节，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，另一个用于对数组元素排序（sort）。

```
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位置所存的值的 C 过程

本小节要在排序的例子中使用这个过程。

2. 13. 1　swap 过程

我们从图 2-24 中的过程 swap 开始。这个过程简单的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。

⊖ Java 虚拟机（Java Virtual Machine）：解释 Java 字节码的程序。

⊖ 即时编译器（Just In Time complier）：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。

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。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。

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 4，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 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 中使用的保留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留寄存器，所以没有需要保存的东西。

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

2.13.2　sort 过程

为保证你会认识到汇编语言编程的严格性，我们尝试提供了这第二个更长的例子。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

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，j 分别分配寄存器 $s0 和 $s1。

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：

```
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：

```
forltst: slt   $t0,$s0,$a1      # reg $t0 =0 if   $s0≥ $a1(i≥n)
         beq   $t0,$zero,exitl  # go to exitl if  $s0≥ $a1(i≥n)
```

循环的底部仅仅需要跳回循环判断的地方：

```
     j    forltst              # jump to test of outer loop
exitl:
```

第一个 for 循环的框架代码为

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

（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)
```

这将跳过第二个条件测试，如果没有跳过的话，j 一定大于等于 0。

第二个测试条件当 v[j] > v[j + 1]非真的时候退出。为得到地址，我们首先将 j 乘 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: 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, $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:
```

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
```

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的较早的地方将 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
```

在 sort 中保存寄存器

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 等保留寄存器，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。

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
```

图 2-27　图 2-26 中 sort 过程的 MIPS 汇编版本

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)
循环体外		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
		ja	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
恢复寄存器的值			
	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　（续）

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。使用内联的缺点是如果内联过程需要在多个地方调用的话，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）。

理解程序的性能

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。（CD 中的 2.15 小结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个字的被初始化为随机数的数组进行排序。程序运行在3.06 GHz的奔腾4处理器上，前端系统总线是533 MHz，具有2 GB的PC2100 DDR SDRAM。操作系统使用Linux 2.4.20。

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没优化的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，它就将四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。本小节将展示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，第二段需要做一些解释。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循环的第一个部分将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。

本小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。我们将在本小节的末尾，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 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：

```
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
```

循环测试条件是 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，这些代码就能正确的工作；ANSI C 需要在循环前测试 size 值，但是我们跳过了这个。）

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个字节，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的判断p是否比数组的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
```

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标记和指针的不同（指针版本带来的变化被高亮显示）：

```
      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<size)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)       # 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
```

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 i 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。CD 中的 2.15 节叙述了这两种优化和其他一些优化。

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。

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

本小节将简要概述 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。请记住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。

本小节的第二部分是为对**面向对象语言**[⊖]（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。

本小节的剩余内容在 CD 上。

2.16　实例：ARM 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，每年都有超过30亿部各种各样的设备使用 ARM 处理器。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。

	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

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指令集核。

⊖ 面向对象语言（objected oriented language）：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。

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	—	lui
	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和 MIPS 是等价的

横线表示在这个指令集下没有该指令。如果有几条可供选择的指令都与 MIPS 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 $\texttt{lsr}^1$。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，然后将产生的新地址存入一个寄存器中。

寻址模式	ARM v. 4	MIPS
寄存器操作数	X	X
立即数操作数	X	X
寄存器 + 偏移（转移或基地址）	X	X
寄存器 + 寄存器（下标）	X	—
寄存器 + 寄存器倍乘（倍乘）	X	—
寄存器 + 偏移和更新寄存器	X	—
寄存器 + 寄存器和更新寄存器	X	—
自增，自减	X	—
相对 PC 的数据	X	—

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：负的（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置位，不同于早期的体系结构，这些置位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，用它们的差置位条件码。CMN 指令将一个操作数与另一个操作数相加，用它们的和来置位条件码。TST 指令将两个操作数进行逻辑与，然后置位除溢出位外其他的条件码。TEQ 指令是用异或结果来置位条件码的前三位。

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。

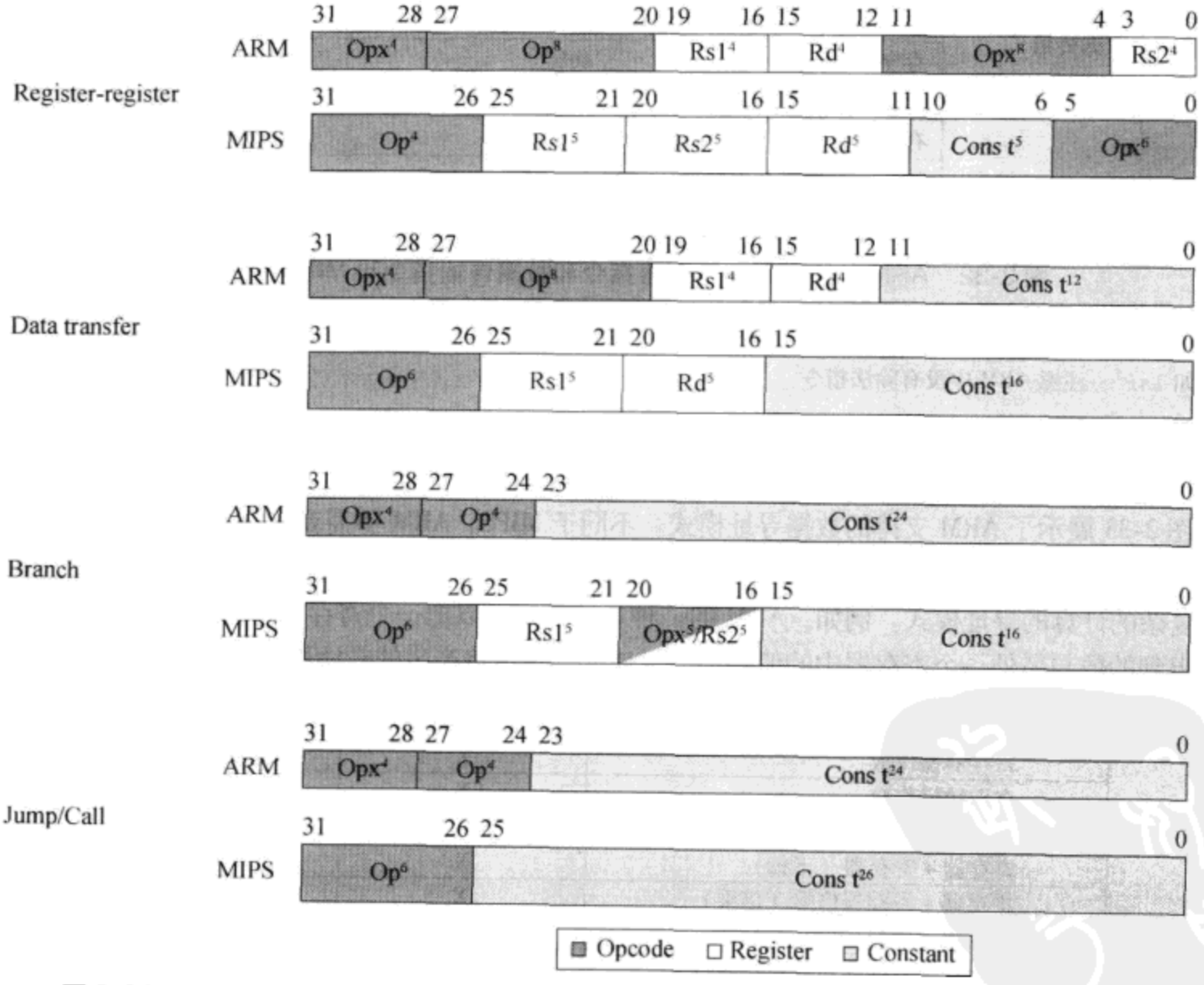


图 2-34　ARM 和 MIPS 的指令格式。区别在于体系结构中是有 16 个还是 32 个寄存器

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。

名字	定义	ARM v. 4	MIPS
取立即数	Rd = Imm	mov	addi，$0，
非	Rd = ~(Rs1)	mvn	nor，$0，
移动	Rd = Rs1	mov	or，$0，
右旋转	$Rd = Rsi >> i$ $Rd_{0\cdots i-1} = Rs_{31-i\cdots 31}$	ror	
和寄存器非的与	Rd = Rs1 & ~(Rs2)	bic	
反向减	Rd = Rs2 - Rs1	rsb，rsc	
支持多个整数字的加	CarryOut，Rd = Rd + Rs1 + OldCarryOut	adcs	—
支持多个整数字的减	CarryOut，Rd = Rd - Rs1 + OldCarryOut	sbcs	—

图2-35　MIPS中没有的ARM算术/逻辑指令

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，移动的位数由高四位的值乘以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。

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。

ARM还对寄存器组的操作提供了指令支持，这些指令叫做*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。这样做的目的是为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。为避免这些问题，设计者会选择更简单的指令。2.18小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30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。

- **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**[⊖]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（见光盘中的 2.20 小节和 3.7 小节）。
- **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，仅有四个指令被增加到用户可见的指令集中，其中三个有助于多处理技术（参见第 7 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 7 章）。Pentium II 没有引入任何新的指令。
- **1999**：Intel 添加了 70 个指令，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。因此四个 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 个指令。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 个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。
- **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 到 10 个新的指令并且去掉了 27 个旧指令。PC 相对寻址是另一个扩展。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/Inte 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，

⊖ 通用寄存器（general-purpose register，GPR）：可用于存储任何指令的地址或数据的寄存器。

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 小节）。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。
- **2008**：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。

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。如果你仔细研究 x86 的扩展过程，你会发现这种体系结构平均每个月就会扩展一条指令。

无论 x86 结构有多失败，该体系结构家族在桌面计算机上的应用比任何其他体系结构都要多，并以每年 2.5 亿的速度增长。然而，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，它也是在当今的体系结构中用到的，而不是整个 16 位和 32 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。

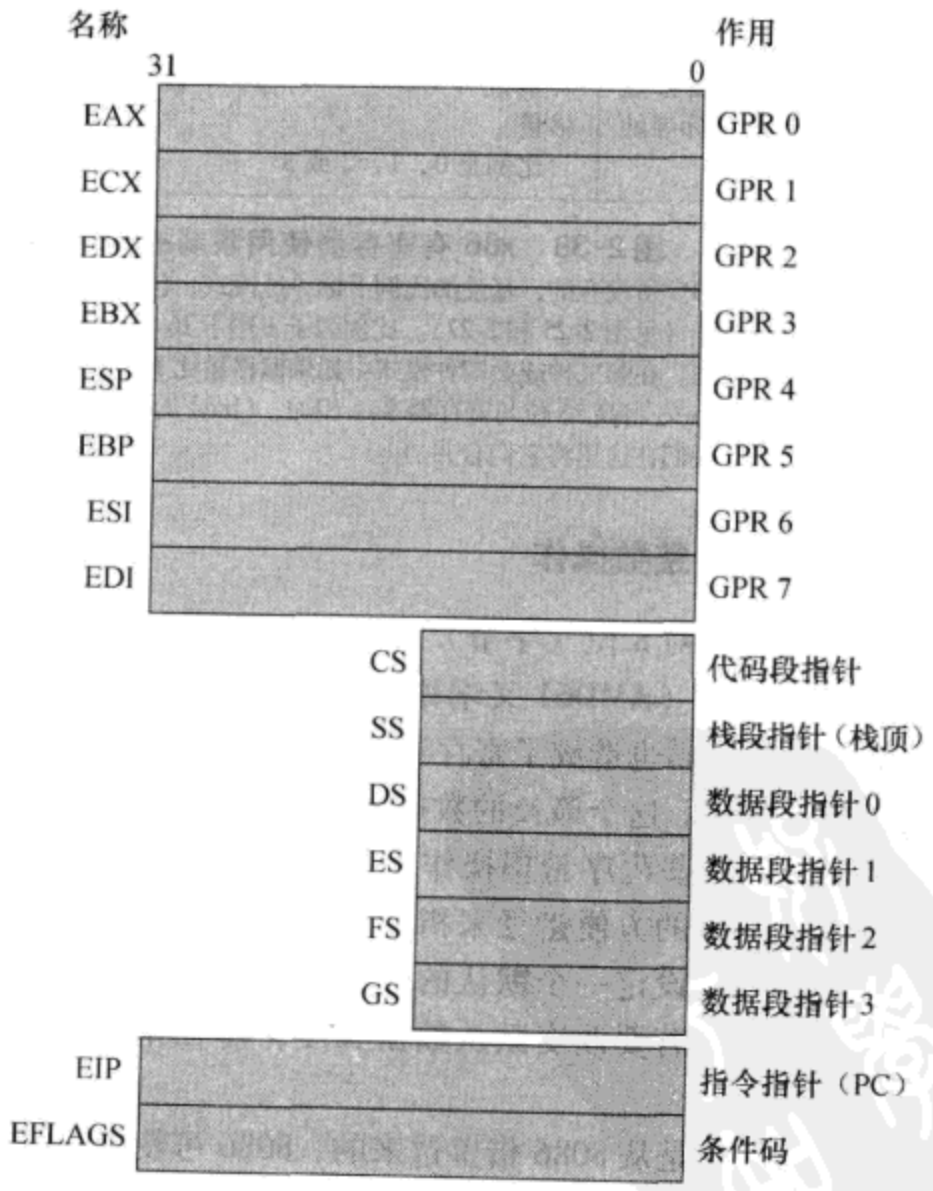


图 2-36　80386 寄存器组

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。

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（图 2-36 所示）。80386 把 16 位寄存器（除了段寄存器）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（general-purpose register，GPR）。80386 只有 8 个通用寄存器，这意味着 MIPS 程序使用四倍数量的寄存器，而 ARM 可以使用 2 倍数量的寄存器。

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 ARM 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ARM和MIPS不同。

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。

源/目的操作数类型	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中主要的14个寄存器（不能是EIP或EFLAGS）的任意一个。

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
	基址＋($2^{比例}$ * 下标)
	比例是0，1，2或3
		基址：任何GPR
		下标：不能为ESP
			`mul $t0,$s2,`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

8位或32位偏移量的基址＋比例下标寻址	地址为
	基址＋($2^{比例}$ * 下标)＋偏移量
	比例是0，1，2或3
		基址：任何GPR
		下标：不能为ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0)#ð6-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）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，叫做四字；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 整数操作可以分为四个主要的类：

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。

除了算术和逻辑操作指令的目的既可以是寄存器也可以是存储器地址外，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。

指令	功能
je 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。）

x86 的条件分支像 ARM 一样基于条件码（condition codes）或标志位（flags）。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 ARM 和 MIPS 不同，80386 的指令不都是 4 字节长的。

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 2.18 节的误解）。

字符串指令是 IA-32 的 8080 家族的一部分，它在大部分程序中都不被使用。它们常常比等同的软件例程要慢（参考 2.17 节）。

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	**在寄存器或寄存器和存储器之间传递数据**
move	在寄存器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 GPRs

图 2-40　一些典型的 x86 操作

指令	含义
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，并设置标志位
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

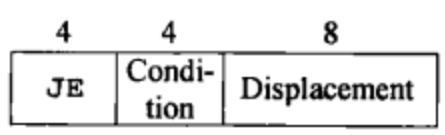
图 2-40　（续）

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。

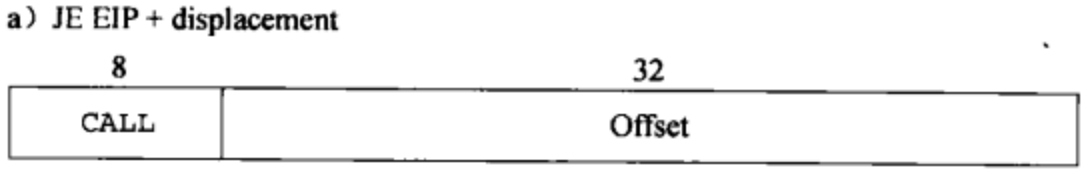
2.17.4　x86 指令编码

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。

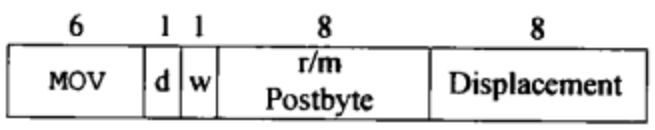
图 2-41 展示了图 2-39 中几条指令的格式。操作码字节中的有一位用来表明操作数是 8 位还



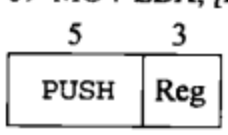
a）JE EIP + displacement



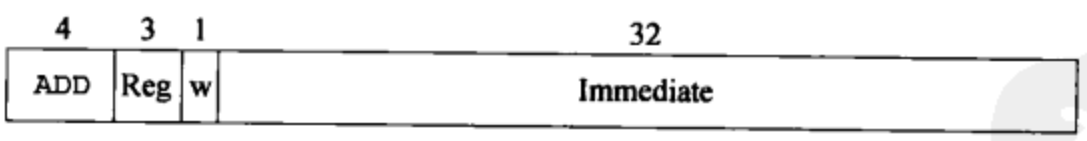
b）CALL



c）MOV EBX, [EDI + 45]



d）PUSH ESI



e）ADD EAX, #6765



f）TEST EDX, #42

图 2-41　典型的 x86 指令格式

图 2-42 给出后置字节（postbyte）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 17 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。

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，index，base。”（比例，下标，基址）。

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。不便的是，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 addr as mod = 0 + disp8*	*same addr as mod = 0 + disp8*	*same addr as mod = 0 + disp16*	*same addr as mod = 0*	*same*
1	CL	CX	ECX	1	addr = BX + DI	= ECX					*as*
2	DL	DX	EDX	2	addr = BP + SI	= EDX					*reg*
3	BL	BX	EBX	3	addr = BP + SI	= EBX				*+ 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。余下的字段解释了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ARM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。

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。

2.18　谬误与陷阱

误解：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。第三种方式，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。

误解：使用汇编语言编程来获得最高的性能。

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）。

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的分配寄存器。

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。

误解：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 30 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。

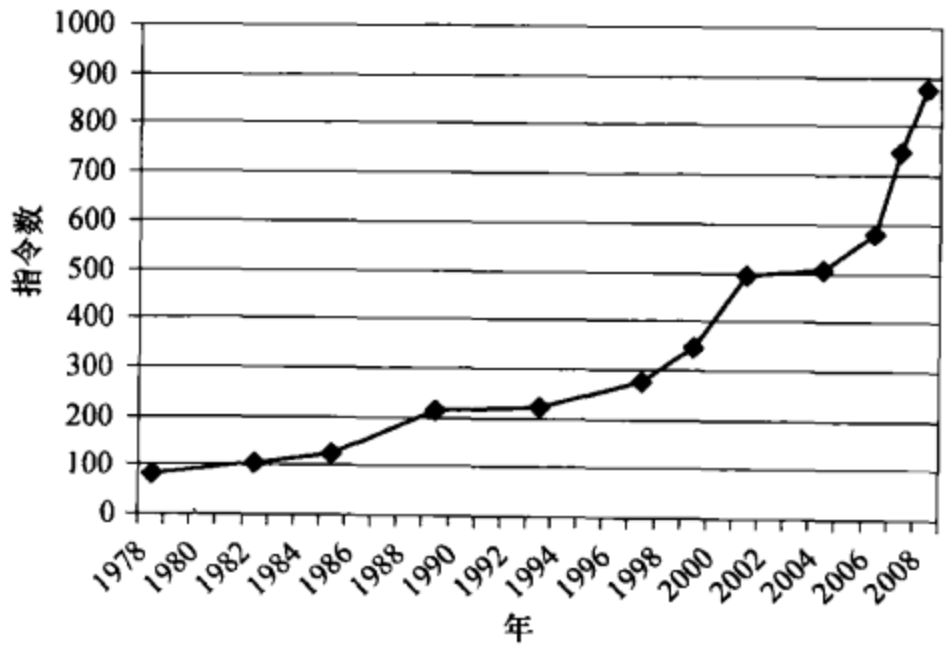


图 2-43　随时间推移 x86 指令集的增长

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。

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。

2.19　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，在做精妙平衡时有四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。

3）加速常用操作。MIPS 加速常用操作的例子包括条件分支中 PC 相对寻址和为大的常数操作数使用立即数寻址。

4）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。

机器语言上面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。

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	bne	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 B（B.20 小节）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 体系机构。这里给出的信息可在本书文前的 MIPS 参考数据的第一和第二列查到。

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。
- 数据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 SPEC2006 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。

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,jar	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 SPEC2006 测试程序执行时所占的比例
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。

2.20　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。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 CD 中。

2.21　练习题

本章习题由加州大波莫纳分校圣路易斯奥比斯波的 John Oliver 提供，并且感谢阿德莱德大学的 Nicole Kaiyan 以及佐治亚理工学院的 Milos Prvulovic。

附录 B 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，但是在求产生的 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，如果问你指令数，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变好（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。

习题 2.1

以下问题是关于从 C 翻译到 MIPS 的。假设给定变量 g、h、i 和 j，像在 C 程序中声明的一样它们都是 32 位的整数。

a.	f = g + h + i + j;
b.	f = g + (h + 5);

2.1.1 [5] <2.2> 对于上面的 C 语句，求对应的 MIPS 汇编代码。（使用最少的 MIPS 汇编指令。）

2.1.2 [5] <2.2> 对于上面的 C 语句，需要多少条 MIPS 汇编指令来完成？

2.1.3 [5] <2.2> 如果变量 f、g、h、i 和 j，分别取值 1、2、3、4 和 5，求最后 f 的值。

以下问题是关于从 MIPS 翻译到 C 的。假设给定变量 g、h、i 和 j，像在 C 程序中声明的一样它们都是 32 位的整数。

a.	`add f, g, h`
b.	`addi f, f, 1` `add f, g, h`

2.1.4 [5] <2.2> 对于上面的 MIPS 语句，求对应的 C 语句。

2.1.5 [5] <2.2> 如果变量 f、g、h 和 i，分别取值 1、2、3 和 4，求最后 f 的值。

习题 2.2

以下问题是关于从 C 翻译到 MIPS 的。假设给定变量 g、h、i 和 j，像在 C 程序中声明的一样它们都是32 位的整数。

a.	`f = f + f + i;`
b.	`f = g + (j + 2);`

2.2.1 [5] <2.2> 对于上面的 C 语句，求对应的 MIPS 汇编代码。(使用最少的 MIPS 汇编指令。)

2.2.2 [5] <2.2> 对于上面的 C 语句，需要多少条 MIPS 汇编指令来完成?

2.2.3 [5] <2.2> 如果变量 f、g、h 和 i，分别取值 1、2、3 和 4，求最后 f 的值。

以下问题是关于从 MIPS 翻译到 C 的。对于以下的习题，假设给定变量 g、h、i 和 j，像在 C 程序中声明的一样它们都是 32 位的整数。

a.	`add f, f, h`
b.	`sub f, $0, f` `addi f, f, 1`

2.2.4 [5] <2.2> 对于上面的 MIPS 语句，求对应的 C 语句。

2.2.5 [5] <2.2> 如果变量 f、g、h 和 i，依次取值 1、2、3 和 4，求最后 f 的值。

习题 2.3

以下问题是关于从 C 翻译到 MIPS 的。假设给定变量 f、g、h、i 和 j，像在 C 程序中声明的一样它们都是 32 位的整数。

a.	`f = f + g + h + i + j + 2;`
b.	`f = g - (f + 5);`

2.3.1 [5] <2.2> 对于上面的 C 语句，求对应的 MIPS 汇编代码。(使用最少的 MIPS 汇编指令。)

2.3.2 [5] <2.2> 对于上面的 C 语句，需要多少条 MIPS 汇编指令来完成?

2.3.3 [5] <2.2> 如果变量 f、g、h、i 和 j，分别取值 1、2、3、4 和 5，求最后 f 的值。

以下问题是关于从 MIPS 翻译到 C 的。对于以下的习题，假设给定变量 g、h、i 和 j，像在 C 程序中声明的一样它们都是 32 位的整数。

a.	`add f, -g, h`
b.	`addi h, f, 1` `sub f, g, h`

2.3.4 [5] <2.2> 对于上面的 MIPS 语句，求对应的 C 语句。

2.3.5 [5] <2.2> 如果变量 f、g、h 和 i，依次取值 1、2、3 和 4，求最后 f 的值。

习题 2.4

以下问题是关于从 C 翻译到 MIPS 的。假定变量 f、g、h、i 和 j，依次分配到寄存器 $s0、$s1、$s2、$s3 和 $s4。假设数组 A 和 B 的基地址依次存放在寄存器 $s6 和 $s7 中。

a.	`f=g+h+B[4];`
b.	`f=g - A[B[4]];`

2.4.1 [10] <2.2, 2.3> 对于上面的 C 语句，求对应的 MIPS 汇编代码。

2.4.2 [5] <2.2, 2.3> 对于上面的 C 语句，需要多少条 MIPS 汇编指令来完成？

2.4.3 [5] <2.2, 2.3> 对于上面的 C 语句，需要多少个不同寄存器来完成该功能？

以下问题是关于从 MIPS 翻译到 C 的。假定变量 f、g、h、i 和 j，依次分配到寄存器 $s0、$s1、$s2、$s3 和 $s4。假设数组 A 和 B 的基地址依次存放在寄存器 $s6 和 $s7 中。

a.	`add $s0, $s0, $s1`
	`add $s0, $s0, $s2`
	`add $s0, $s0, $s3`
	`add $s0, $s0, $s4`

b.	`lw $s0, 4($s6)`

2.4.4 [10] <2.2, 2.3> 对于上面的 MIPS 汇编指令，求对应的 C 语句。

2.4.5 [5] <2.2, 2.3> 对于上面的 MIPS 汇编指令，求实现相同功能可用的最少的汇编代码。

2.4.6 [5] <2.2, 2.3> 对于上面的 MIPS 汇编代码，使用多少个寄存器能实现功能？对于上题如果重写了代码，至少要多少个寄存器才能实现功能？

习题 2.5

以下问题是关于 MIPS 处理器中存储器操作的。下表展示了存储在存储器中的数组的值。

a.

Address	Data
12	1
8	6
4	4
0	2

b.

Address	Data
16	1
12	2
8	3
4	4
0	5

2.5.1 [10] <2.2, 2.3> 基于上表中数据在存储器中的位置，求一段 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 的一个整数数组，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。）

2.5.2 [10] <2.2, 2.3> 基于上表中数据在存储器中的位置，求一段 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。）

2.5.3 [5] <2.2, 2.3> 为了对数组排序，MIPS 代码需要多少条指令？如果不允许在使用 lw 和 sw 指令的时候使用立即数字段，那么需要多少条 MIPS 指令完成相同功能？

以下问题是关于将十六进制数字转换成其他进制格式。

a.	`0x12345678`
b.	`0xbeadf00d`

2.5.4 [5] <2.3>将上表中的十六进制数转换成10进制数。

2.5.5 [5] <2.3>分别画出上表中的数据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。）

习题2.6

以下问题是关于从C翻译到MIPS的。假定变量f、g、h、i和j，分别分配到寄存器$s0、$s1、$s2、$s3和$s4。假设数组A和B的基地址依次存放在寄存器$s6和$s7中。

a.	`f = -g + h + B[1];`
b.	`f = A[B[g] + 1];`

2.6.1 [10] <2.2，2.3>对于上面的C语句，求对应的MIPS汇编代码。

2.6.2 [5] <2.2，2.3>对于上面的C语句，需要多少条MIPS汇编指令来完成该功能？

2.6.3 [5] <2.2，2.3>对于上面的C语句，使用MIPS汇编语言需要多少个不同寄存器来完成该功能？

以下问题是关于从MIPS翻译到C的。假定变量f、g、h、i和j，依次分配到寄存器$s0、$s1、$s2、$s3和$s4。假设数组A和B的基地址依次存放在寄存器$s6和$s7中。

a.

```
add   $s0, $s0, $s1
add   $s0, $s3, $s2
add   $s0, $s0, $s3
```

b.

```
addi  $s6, $s6, -20
add   $s6, $s6, $s1
lw    $s0, 8 ($s6)
```

2.6.4 [5] <2.2，2.3>对于上面的MIPS汇编指令，求对应的C语句。

2.6.5 [5] <2.2，2.3>对于上面的MIPS汇编指令，假设寄存器$s0、$s1、$s2和$s3分别取指10，20，30和40，并假设寄存器$s6中的值为256，存储器中包含下列值：

地址	值
256	100
260	200
264	300

求最终寄存器$s0中的值。

2.6.6 [10] <2.3，2.5>求每条MIPS指令的下列字段值：op、rs和rt字段。对于I型指令，求立即数字段的值。对于R型指令，求rd字段的值。

习题2.7

以下问题是关于从有符号和无符号二进制数转换到十进制数的。

a.	$1010\ 1101\ 0001\ 0000\ 0000\ 0000\ 0000\ 0010_2$
b.	$1111\ 1111\ 1111\ 1111\ 1011\ 0011\ 0101\ 0011_2$

2.7.1 [5] <2.4>假定上表中是两个补码表示的整数，求它们表示的十进制数。

2.7.2 [5] <2.4>假定上表中是两个无符号整数，求它们表示的十进制数。

2.7.3 [5] <2.4>求上表中位模式表示的十六进制数。

以下问题是关于从十进制数转换到有符号和无符号二进制数的。

a.	2147483647_{10}
b.	1000_{10}

2.7.4 [5] <2.4>求上表中的十进制数的二进制表示的补码表示形式。

2.7.5 [5] <2.4>求上表中的十进制数的十六进制表示的补码表示形式。

2.7.6 [5] <2.4>求上表中的十进制数的相反数的十六进制表示的补码表示形式。

习题 2.8

以下问题关于符号扩展和溢出。寄存器 $s0 和 $s1 中的值见下表。按照 MIPS 指示操作这些寄存器并求得结果。

a.	$s0 = 70000000_{16}, $s1 = 0x0FFFFFFF_{16}$
b.	$s0 = $0x40000000_{16}$, $s1 = 0x40000000_{16}$

2.8.1 [5] <2.4>寄存器 $s0 和 $s1 的值由上表指定，求执行下面汇编代码后寄存器 $t0 中的值。

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

$t0 中的值是希望得到的结果还是发生了溢出？

2.8.2 [5] <2.4>寄存器 $s0 和 $s1 的值由上表指定，求执行下面汇编代码后寄存器 $t0 中的值。

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

$t0 中的值是希望得到的结果还是发生了溢出？

2.8.3 [5] <2.4>寄存器 $s0 和 $s1 的值由上表指定，求执行下面汇编代码后寄存器 $t0 中的值。

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

$t0 中的值是希望得到的结果还是发生了溢出？

以下问题是关于不同的 MIPS 指令对一对寄存器 $s0 和 $s1 进行操作。$s0 和 $s1 的值由每题单独指定，指出是否发生溢出。

a.	`add  $s0, $s0, $s1`
b.	`sub  $s0, $s0, $s1` `sub  $s0, $s0, $s1`

2.8.4 [5] <2.4>假设寄存器中的值分别为 $s0 = 0x70000000 and $s1 = 0x10000000。对于上表中的指令序列，会发生溢出吗？

2.8.5 [5] <2.4>假设寄存器中的值分别为 $s0 = 0x40000000 and $s1 = 0x20000000。对于上表中的指令序列，会发生溢出吗？

2.8.6 [5] <2.4>假设寄存器中的值分别为 $s0 = 0x8FFFFFFF and $s1 = 0xD0000000。对于上表中的指令序列，会发生溢出吗？

习题 2.9

下表包含了寄存器 $s1 的不同的值。对于给定的操作，判断是否发生溢出。

a.	2147483647_{10}
b.	$0xD0000000_{16}$

2.9.1 [5] <2.4>假设寄存器 $s0 中的值是 0x70000000，寄存器 $s1 中的值由上表给出。如果执行指令 `add $s0, $s0, $s1`，会发生溢出吗？

2.9.2 [5] <2.4>假设寄存器 $s0 中的值是 0x80000000，寄存器 $s1 中的值由上表给出。如果执行指令 `sub $s0, $s0, $s1`，会发生溢出吗？

2.9.3 [5] <2.4>假设寄存器 $s0 中的值是 0x7FFFFFFF，寄存器 $s1 中的值由上表给出。如果执行指令

sub $s0, $s0, $s1，会发生溢出吗？

下表包含寄存器 $s1 的不同值，求对于给定的操作，是否发生溢出。

a.	$1010\ 1101\ 0001\ 0000\ 0000\ 0000\ 0000\ 0010_2$
b.	$1111\ 1111\ 1111\ 1111\ 1011\ 0011\ 0101\ 0011_2$

2.9.4 [4] <2.4>假设寄存器 $s0 中的值是 0x70000000，寄存器 $s1 中的值由上表给出。如果执行指令 add $s0, $s0, $s1，会发生溢出吗？

2.9.5 [4] <2.4>假设寄存器 $s0 中的值是 0x70000000，寄存器 $s1 中的值由上表给出。如果执行指令 add $s0, $s0, $s1，求最终的结果。（用十六进制表示。）

2.9.6 [4] <2.4>假设寄存器 $s0 中的值是 0x70000000，寄存器 $s1 中的值由上表给出。如果执行指令 add $s0, $s0, $s1，求最终的结果。（用十进制表示。）

习题 2.10

以下问题是关于将表中表示指令操作的位条目转换成汇编代码并且指出每条 MIPS 指令的格式。

a.	$1010\ 1110\ 0000\ 1011\ 0000\ 0000\ 0000\ 0100_2$
b.	$1000\ 1101\ 0000\ 1000\ 0000\ 0000\ 0100\ 0000_2$

2.10.1 [5] <2.5>求上面的位条目分表表示什么指令。

2.10.2 [5] <2.5>求上面的位条目所表示的指令的类型（I 型或 R 型）。

2.10.3 [5] <2.5>如果上面的位条目是数据，求上面数字的十六进制表示。

以下问题是关于将表中的 MIPS 指令转换成位模式并且指出每条 MIPS 指令的格式类型。

a.	`add $t0, $t0, $zero`
b.	`lw $t1, 4 ($s3)`

2.10.4 [5] <2.4, 2.5>求上表中指令的十六进制表示。

2.10.5 [5] <2.5>求上面指令的类型（I 型或 R 型）。

2.10.6 [5] <2.5>求指令的十六进制表示的 opcode、rs 和 rt 字段；对于 R 型指令，求十六进制表示的 rd 和 funct 字段；对于 I 型指令，求十六进制表示的立即数字段。

习题 2.11

以下问题是关于将表中表示指令操作的位条目转换成汇编代码并且指出每条 MIPS 指令的格式。

a.	0xAE0BFFFC
b.	0x8D08FFC0

2.11.1 [5] <2.4, 2.5>求上面十六进制数所表示的二进制数。

2.11.2 [5] <2.4, 2.5>求上面十六进制数所表示的十进制数。

2.11.3 [5] <2.5>求上面十六进制数所表示的指令。

以下问题是根据表中所含的 MIPS 指令不同字段的值来求得指令并指出指令的类型。

a.	op = 0, rs = 1, rt = 2, rd = 3, shamt = 0, funct = 32
b.	op = 0x2B, rs = 0x10, rt = 0x5, const = 0x4

2.11.4 [5] <2.5>求上表所表示的指令的类型（I 型或 R 型）。

2.11.5 [5] <2.5>求上表所表示的汇编指令。

2.11.6 [5] <2.4，2.5>求上表所表示的指令的二进制表示。

习题2.12

在下面的问题中，数据表中包含一些对 MIPS 指令集体系结构的修改。指出这些修改对 MIPS 体系结构指令集格式的影响。

a.	8 registers
b.	10 bit immediate constants

2.12.1 [5] <2.5>如果 MIPS 处理器的指令集改变了，指令的格式也将被改变。对于上表中的每种修改，求 R 型指令格式不同字段的位数，每条指令一共需要多少位。

2.12.2 [5] <2.5>如果 MIPS 处理器的指令集改变了，指令的格式也将被改变。对于上表中的每种修改，求 I 型指令格式不同字段的位数，每条指令一共需要多少位。

2.12.3 [5] <2.5，2.10>请说明为什么上述修改会减少整个 MIPS 汇编程序的大小？为什么上述修改又会增加整个 MIPS 汇编程序的大小？

在下面的问题中，数据表中包含十六进制的值。指出这些值表示的 MIPS 指令是什么，并指出这些指令的类型。

a.	0x01090010
b.	0x8D090012

2.12.4 [5] <2.5>求上表中十六进制数所表示的十进制值。

2.12.5 [5] <2.5>求上表中十六进制数所表示的指令。

2.12.6 [5] <2.4，2.5>求上表中十六进制数所表示指令的类型，并求 op 和 rt 字段的值。

习题2.13

在下面的问题中，数据表中包含寄存器 $t0 和 $t1 的值。按照下面的逻辑指令对这些寄存器进行操作。

a.	$t0 = 0x55555555, $t1 = 0x12345678
b.	$t0 = 0xBEADFEED, $t1 = 0xDEADFADE

2.13.1 [5] <2.6>求执行下面的指令序列后寄存器 $t2 的值。

```
sll   $t2, $t0, 4
or    $t2, $t2, $t1
```

2.13.2 [5] <2.6>求执行下面的指令序列后寄存器 $t2 的值。

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

2.13.3 [5] <2.6>求执行下面的指令序列后寄存器 $t2 的值。

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

在下面的练习中，数据表中包含不同的 MIPS 逻辑操作，对于给定的不同寄存器 $t0 和 $t1 值，求这些操作最终的结果。

a.	sll $t2, $t0, 1 or $t2, $t2, $t1	**b.**	srl $t2, $t0, 1 andi $t2, $t2, 0x00F0

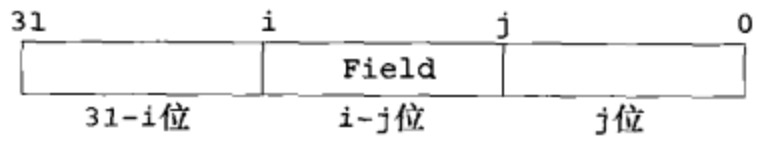
2.13.4 [5] <2.6>假设 $t0 =0x0000A5A5，$t1 =00005A5A。求执行上表中的指令后最终寄存器 $t2 中的值。

2.13.5 [5] <2.6>假设 $t0 =0xA5A50000，$t1 = A5A50000。求执行上表中的指令后最终寄存器 $t2 中的值。

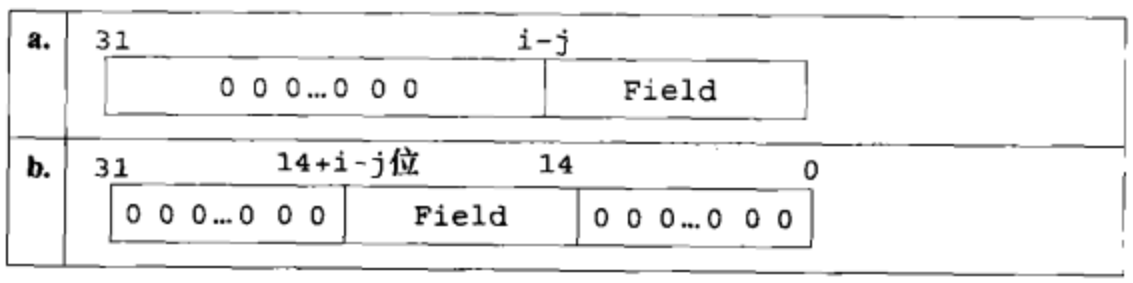
2.13.6 [5] <2.6>假设 $t0 =0xA5A5FFFF，$t1 = A5A5FFFF。求执行上表中的指令后最终寄存器 $t2 中的值。

习题 2.14

下表展示了寄存器 $t0 中位字段的放置情况。



在下面的问题中，你将要使用 MIPS 指令从寄存器 $t0 中提取“Field”字段，然后将该字段存入下表中所示的寄存器 $t1 的相应位置。

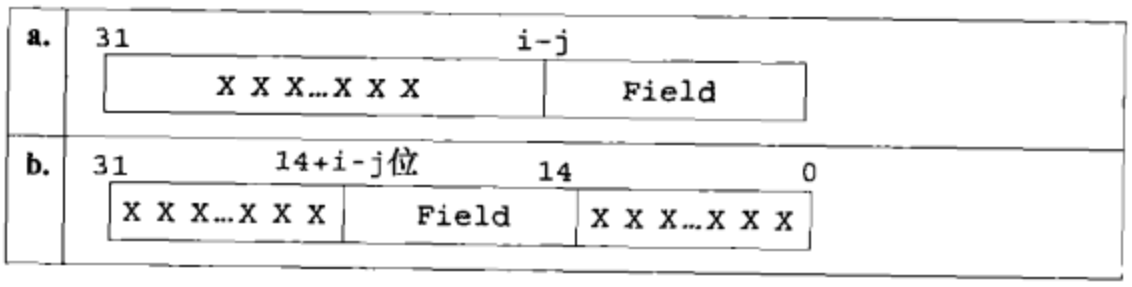


2.14.1 [20] <2.6>求最短的 MIPS 指令序列，当 i =22，j =5 时，该指令序列从寄存器 $t0 中提取 Field 字段，并存入上表中所显示的寄存器 $t1 的相应位置。

2.14.2 [20] <2.6>求最短的 MIPS 指令序列，当 i =4，j =0 时，该指令序列从寄存器 $t0 中提取 Field 字段，并存入上表中所显示的寄存器 $t1 的相应位置。

2.14.3 [20] <2.6>求最短的 MIPS 指令序列，当 i =31，j =28 时，该指令序列从寄存器 $t0 中提取 Field 字段，并存入上表中所显示的寄存器 $t1 的相应位置。

在下面的问题中，你将要使用 MIPS 指令从寄存器 $t0 中提取“Field”字段，然后将该字段存入下表中所示的寄存器 $t1 的相应位置。标记为“XXX”的位值是不变的。



2.14.4 [20] <2.6>求最短的 MIPS 指令序列，当 i =17，j =11 时，该指令序列从寄存器 $t0 中提取 Field 字段，并存入上表中所显示的寄存器 $t1 的相应位置。

2.14.5 [20] <2.6>求最短的 MIPS 指令序列，当 i =5，j =0 时，该指令序列从寄存器 $t0 中提取 Field 字段，并存入上表中所显示的寄存器 $t1 的相应位置。

2.14.6 [20] <2.6>求最短的 MIPS 指令序列，当 i =31，j =29 时，该指令序列从寄存器 $t0 中提取 Field 字段，并存入上表中所显示的寄存器 $t1 的相应位置。

习题 2.15

在下面的问题中，表中所含的逻辑指令都不是 MIPS 指令集中所包含的。请问如何实现这些指令？

a.	`andn $t1, $t2, $t3　　// bit-wise AND of $t2,! $t3`
b.	`xnor $t1, $t2, $t3　　// bit-wise exclusive-NOR`

2.15.1 [5] <2.6>对于上表中不包含在 MIPS 指令集中的逻辑指令，如果寄存器 $t2 =0x00FFA5A5，$t3 = 0xFFFF003C，最终寄存器 $t1 的值是多少？

2.15.2 [10] <2.6> 上面的逻辑指令不包括在 MIPS 指令集中，但是可以使用一条或多条 MIPS 汇编指令实现。请用最少的 MIPS 指令序列来代替上表中的指令。

2.15.3 [5] <2.6> 对于 2.15.2 中的指令序列，写出每条指令的位级的表示。

下表展示了 C 语言级别上不同的逻辑语句。请用 MIPS 汇编指令实现这些 C 语言功能。

a.	`A = B & C[0];`
b.	`A = A ? B: C[0]`

2.15.4 [5] <2.6> 上表中展示了不同的 C 语句的逻辑操作。如果存储器中 C [0] 位置的值为 0x00001234，整数 A 和 B 的初始值是 0x00000000 和 0x00002222，求最终 A 的值。

2.15.5 [5] <2.6> 对于上面的 C 语句，请用最少的 MIPS 汇编指令序列实现相同的功能。

2.15.6 [5] <2.6> 对于 2.15.5 中的指令序列，请写出每条指令的位级的表示。

习题 2.16

下表是寄存器 $t0 的不同二进制值。对于给定的 $t0 值，请计算不同分支的效果。

a.	$1010\ 1101\ 0001\ 0000\ 0000\ 0000\ 0000\ 0010_2$
b.	$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$

2.16.1 [5] <2.7> 假设寄存器 $t0 的值来自上表，寄存器 $t1 的值为：

$0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_2$

求执行下面的指令后寄存器 $t2 的值。

```
        slt     $t2, $t0, $t1
        beq     $t2, $zero,ELSE
        j       DONE
ELSE:   addi    $t2, $zero,2
DONE:
```

2.16.2 [5] <2.7> 假设寄存器 $t0 的值来自上表。使用下面的 MIPS 指令，将 $t0 与值 X 相比较。求使 $t2 值为 1 的值 X。

```
slti    $t2, $t0,X
```

2.16.3 [5] <2.7> 假设程序计数器 PC 被设置为 0x0000 0020。请问可以使用 MIPS 汇编指令集中的跳转指令将 PC 设置为上表中的值吗？有可能使用等于则跳转指令（beq）将 PC 设置为上表中的地址值吗？

对于以下问题，表中的值为寄存器 $t0 的不同的二进制值。对于给定的值 $t0，请计算不同分支的效果。

a.	0x00001000
b.	0x20001400

2.16.4 [5] <2.7> 假设寄存器 $t0 的值来自上表。求执行下面的指令序列后 $t2 的值。

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

2.16.5 [5] <2.6, 2.7> 假设寄存器 $t0 的值来自上表。求执行下面的指令序列后 $t2 的值？

```
sll     $t0,   $t0,   2
slt     $t2,   $t0,    $zero
```

2.16.6 [5] <2.7> 假设程序计数器 PC 被设置为 0x2000 0000。请问可以使用 MIPS 汇编指令集中的跳转指令将 PC 设置为上表中的值吗？有可能使用等于则跳转指令（beq）将 PC 设置为上表中的地址值吗？

习题 2.17

对于以下问题，表中是几条 MIPS 指令集中没有的指令。

a.	`abs  $t2, $t3  # R[rd] = |R[rt]|`
b.	`sgt  $t1, $t2, $t3  # R[rd] = (R[rs] > R[rt])?1:0`

2.17.1 [5] <2.7> 上表中给出了 MIPS 指令集中没有的指令和每条指令的描述。请问这些指令为什么没被包含到 MIPS 指令集中？

2.17.2 [5] <2.7> 上表中给出了 MIPS 指令集中没有的指令和每条指令的描述。如果这些指令已经在 MIPS 指令集中实现，请问最可能的指令格式是什么？

2.17.3 [5] <2.7> 对于上表中的指令，求实现相同功能的最短的 MIPS 指令序列。

对于以下问题，表中是 MIPS 汇编代码段。评估每个代码段，并熟悉不同的 MIPS 分支指令。

a.

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

b.

```
LOOP:   addi   $t2,   $0,   0xA
LOOP2:  addi   $s2,   $s2,  2
        subi   $t2,   $t2,  1
        bne    $t2,   $0,   LOOP2
        subi   $t1,   $t1,  1
        bne    $t1,   $0,   LOOP
DONE:
```

2.17.4 [5] <2.7> 对于上面 MIPS 汇编语言的循环，假定寄存器 $t1 被初始化为 10，$s2 的初始值为 0，求 $s2 的最终值。

2.17.5 [5] <2.7> 对于上面的每个循环，求等价的 C 过程。（假定寄存器 $s1，$s2，$t1 和 $t2 依次代表整数 A，B，i 和 temp。）

2.17.6 [5] <2.7> 对于上面的 MIPS 汇编写的循环，假定寄存器 $t1 的初始值为 N。求一共执行了多少条 MIPS 指令。

习题 2.18

对于以下的问题，表中是 C 代码。请把这些 C 代码转变成 MIPS 汇编代码。

a.

```
for(i=0;  i<10;  i++)
    a +=b;
```

b.

```
while (a<10){
    D[a]=b+a;
    a+=1;
}
```

2.18.1 [5] <2.7> 求上表中的 C 代码的控制流程图？

2.18.2 [5] <2.7> 将上表中的 C 代码转换成 MIPS 汇编代码，要求使用最少数量的指令。假设 a，b，i 和 j 的值依次在寄存器 $s0，$s1，$t0 和 $t1 中，并且假设数组 D 的基地址在 $s2 中。

2.18.3 [5] <2.7> 请问实现 C 代码的功能需要多少条 MIPS 指令？如果变量 a 和 b 依次被初始化为 10 和 1，数组 D 的所有元素初始化为 0，求完成循环一共需要执行多少条 MIPS 指令？

对于以下问题，表中是 MIPS 汇编代码段。评估每个代码段，并熟悉不同的 MIPS 分支指令。

a.

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

b.

```
        addi    $t1,  $0,  400
LOOP:   lw      $s1,  0($s0)
        add     $s2,  $s2, $s1
        lw      $s1,  4($s0)
        add     $s2,  $s2, $s1
        addi    $s0,  $s0, 8
        bne     $t1,  $0,  LOOP
```

2.18.4 [5] <2.7> 求一共执行多少条 MIPS 指令？

2.18.5 [5] <2.7> 请将上面的循环转化为 C 程序，假设寄存器 `$t1`，`$t2` 依次存放 C 语言级的整数 `i` 和 `result`，`MemArray` 的基地址存放在 `$s0` 中。

2.18.6 [5] <2.7> 请将上面的 MIPS 汇编代码重写，以减少执行的 MIPS 指令数。

习题 2.19

对于以下问题，表中是 C 函数。假设表中第一个函数叫做函数 first。将这些 C 代码过程转换成 MIPS 汇编。

a.

```
int compare(int a,  int b){
    if(sub(a, b) >=0)
        return 1;
    else
        return 0;
}
int sub(int a,  int b){
    return a - b;
}
```

b.

```
int fib_iter(int a,  int b,  int n){
    if(n  == 0)
        return b;
    else
        return fib_iter(a +b, a, n -1);
}
```

2.19.1 [15] <2.8> 请将上表中的 C 代码用 MIPS 汇编实现。请问执行这个函数总共需要多少条 MIPS 汇编指令？

2.19.2 [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拷贝到程序空间中，以消除函数调用的开销。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 MIPS 汇编指令？（假设 C 的变量 n 被初始化为 5。）

2.19.3 [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。）

下面的三个问题是关于函数 `f` 调用另外一个函数 `func` 的。函数 `func` 的 C 代码已经使用图 2-14 的 MIPS 调用约定编译到另一个模块。函数 `func` 的声明为“`int func(int a,int b);`”，函数 `f` 的代码如下：

a.

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

b.

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

2.19.4 [10] <2.8> 将函数 `f` 翻译成 MIPS 汇编，同样使用图 2-14 的 MIPS 调用约定。如果需要使用寄存器 `$t0` 到 `$t7`，请从编号小的寄存器开始使用。

2.19.5 [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。

2.19.6 [5] <2.8> 在习题 2.19.4 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。）

习题 2.20

以下问题是关于递归过程调用。对于这些问题，表中是求数的阶乘的汇编代码段。然而，表中的代码

有错误，请找出这些错误。

a.

```
FACT:  addi   $sp,$sp,-8
       sw     $ra,4($sp)
       sw     $a0,0($sp)
       slti   $t0,$a0,1
       beq    $t0,$0,L1
       addi   $v0,$0,1
       addi   $sp,$sp,8
       jr     $ra
L1:    addi   $a0,$a0,-1
       jal    FACT
       lw     $a0,4($sp)
       lw     $ra,0($sp)
       addi   $sp,$sp,8
       mul    $v0,$a0,$v0
       jr     $ra
```

b.

```
FACT:  addi   $sp,$sp,-8
       sw     $ra,4($sp)
       sw     $a0,0($sp)
       slti   $t0,$a0,1
       beq    $t0,$0,L1
       addi   $v0,$0,1
       addi   $sp,$sp,8
       jr     $ra
L1:    addi   $t0,$t0,-1
       jal    FACT
       lw     $a0,4($sp)
       lw     $ra,0($sp)
       addi   $sp,$sp,8
       mul    $v0,$a0,$v0
       jr     $ra
```

2.20.1 [5] <2.8> 上面的 MIPS 汇编程序计算给定输入的阶乘。输入的整数通过寄存器$a0 传入，值通过$v0 返回。汇编代码中有些错误，请改正这些错误。

2.20.2 [10] <2.8> 对于上面的 MIPS 阶乘递归程序，假设输入为 4。求程序的非递归形式。（仅能使用寄存器$s0 ~ $s7。）比较这个非递归程序的指令数和递归形式的指令数。

2.20.3 [5] <2.8> 假设输入为 4，画出每次函数调用后栈的内容。

对于以下问题，表中的汇编代码段用来计算斐波那契数（Fibonacci number）。然而，表中的代码有错误，请找出这些错误。

a.

```
FIB:   addi   $sp,$sp,-12
       sw     $ra,0($sp)
       sw     $s1,4($sp)
       sw     $a0,8($sp)
       slti   $t0,$a0,1
       beq    $t0,$0,L1
       addi   $v0,$a0,$0
       j      EXIT
L1:    addi   $a0,$a0,-1
       jal    FIB
       addi   $s1,$v0,$0
       addi   $a0,$a0,-1
       jal    FIB
       add    $v0,$v0,$s1
EXIT:  lw     $ra,0($sp)
       lw     $a0,8($sp)
       lw     $s1,4($sp)
       addi   $sp,$sp,12
       jr     $ra
```

b.

```
FIB:   addi   $sp,$sp,-12
       sw     $ra,0($sp)
       sw     $s1,4($sp)
       sw     $a0,8($sp)
       slti   $t0,$a0,1
       beq    $t0,$0,L1
       addi   $v0,$a0,$0
       j      EXIT
L1:    addi   $a0,$a0,-1
       jal    FIB
       addi   $s1,$v0,$0
       addi   $a0,$a0,-1
       jal    FIB
       add    $v0,$v0,$s1
EXIT:  lw     $ra,0($sp)
       lw     $a0,8($sp)
       lw     $s1,4($sp)
       addi   $sp,$sp,12
       jr     $ra
```

2.20.4 [5] <2.8> 上面的 MIPS 汇编程序计算给定输入值的 Fibonacci 数。输入的整数通过寄存器$a0 传入，值通过$v0 返回。汇编代码中有些错误，请改正这些错误。

2.20.5 [10] <2.8> 对于上面的 MIPS 阶乘递归程序，假设输入为 4。求程序的非递归形式。（仅能使用寄存器$s0 ~ $s7。）比较这个非递归程序的指令数和递归形式的指令数。

2.20.6 [5] <2.8> 假设输入为 4，画出每次函数调用后栈的内容。

习题 2.21

假设栈和静态数据段都是空的并且栈指针和全局指针依次指向地址 0x7fff fffc 和 0x1000 8000。调用习

惯如图 2-11，函数的输入使用寄存器$a0，返回值使用$v0。假定页函数仅可以使用保留寄存器。试回答下面问题。

a.

```
main ()
{
     leaf_function(1);
}
     int leaf_function(int f)
{
     int result;
     result = f + 1;
     if(f > 5)
          return result;
     leaf_function(result);
}
```

b.

```
int my_global = 100
main ()
{
     int x = 10;
     int y = 20;
     int z;
     z = my_function(x,my_global);
}
int my_function(int x,int y)
{
     return x - y;
}
```

2.21.1 [5] <2.8> 画出每次函数调用后栈和静态数据段的内容。

2.21.2 [5] <2.8> 请将上表中的代码转换成 MIPS 代码。

2.21.3 [5] <2.8> 如果页函数允许使用临时寄存器（$t0,$t1 等），请将上表中的代码转换成 MIPS 代码。

a.

```
f:  sub   $s0,$a0,$a3
    sll   $v0,$s0,0x1
    add   $v0,$a2,$v0
    sub   $v0,$v0,$a1
    jr    $ra
```

b.

```
f:  addi  $sp,$sp,8
    sw    $ra,4($sp)
    sw    $s0,0($sp)
    move  $s0,$a2
    jal   g
    add   $v0,$v0,$s0
    lw    $ra,4($sp)
    lw    $s0,0($sp)
    addi  $sp,$sp,-8
    jr    $ra
```

2.21.4 [10] <2.8> 这段代码包含一个违反 MIPS 调用约定的错误。请问这个错误是什么？如何修正？

2.21.5 [10] <2.8> 求与之等价的 C 代码，假定 C 函数的参数名字是 a,b,c 等。

2.21.6 [10] <2.8> 当函数被调用时，寄存器$a0，$a1，$a2 和$a3 的值分别是 1，10，1000 和 30。求函数的返回值。（如果 f 调用另一个函数 g，假定 g 的返回值是 500。）

习题 2.22

以下问题是关于 ASCII 和 Unicode，下表中是字符组成的串。

a.	A byte
b.	Computer

2.22.1 [5] <2.9> 将字符串转换成十进制的 ASCII 码。

2.22.2 [5] <2.9> 将字符串转换成 16 位的 Unicode（使用十六进制标记法和 Basic Latin 字符集）编码方式。下表是十六进制的 ASCII 码字符值。

a.	61 64 64
b.	73 68 69 66 74

2.22.3 [5] <2.5，2.9> 将上表中的十六进制 ASCII 码转换成文本。

习题 2.23

在本练习中，你将编写一段将字符串转化成表中指定的数格式的 MIPS 汇编程序。

a.	十进制正整数串
b.	二进制补码的十六进制整数

2.23.1 [10] <2.9> 用 MIPS 汇编语言写一段代码将上表中所给条件的 ASCII 码的数串转换成整数。在程序中使用寄存器$a0 处理由数字 0～9 组成的非终结串的地址。程序应该计算与这个数字串等值的整数，并将这个整数存放在寄存器$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}。

习题 2.24

假设寄存器$t1 中包含地址 0x1000 0000，寄存器$t2 中包含地址 0x1000 0010。

a.	`lb $t0,0($t1)` `sw $t0,0($t2)`	b.	`lb $t0,0($t1)` `sb $t0,0($t2)`

2.24.1 [5] <2.9> 假设地址 0x1000 0000 处的数据（十六进制）是：

1000 0000	12	34	56	78

求寄存器$t2 中的地址指向的存储器中的值是多少？假定$t2 指向的寄存器位置的初始值是 0xFFFF FFFF。

2.24.2 [5] <2.9> 假设地址 0x1000 0000 处的数据（十六进制）是：

1000 0000	80	80	80	80

求寄存器$t2 中的地址指向的存储器中的值是多少？假定$t2 指向的寄存器位置的初始值是 0x0000 0000。

2.24.3 [5] <2.9> 假设地址 0x1000 0000 处的数据（十六进制）是：

1000 0000	11	11	00	FF

求寄存器$t2 中的地址指向的存储器中的值是多少？假定$t2 指向的寄存器位置的初始值是 0x5555 5555。

习题 2.25

在这个练习中，将探索 32 位的 MIPS 常量。以下问题将使用下表中的二进制数据。

a.	$1010\ 1101\ 0001\ 0000\ 0000\ 0000\ 0000\ 0010_2$
b.	$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$

2.25.1 [10] <2.10> 请编写能产生上表中的 32 位常量的 MIPS 代码，并将值存储到寄存器$t1 中。

2.25.2 [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上表中所指定的 PC 地址吗？

2.25.3 [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上表中所指定的 PC 地址吗？

2.25.4 [5] <2.6，2.10> 如果当前 PC 值是 0x00400600，可以使用单独的分支指令跳转到上表中所指定的 PC 地址吗？

2.25.5 [10] <2.10> 如果 MIPS 指令的立即数字段只有 8 位长，请编写能产生上表中所列的 32 位常量的 MIPS 代码，并将值存储到寄存器$t1 中。（不允许使用 lui 指令。）

对于以下问题，将使用下表中的 MIPS 汇编代码。

a.	`lui $t0, 0x1234` `ori $t0, $t0, 0x5678`	b.	`ori $t0, $t0, 0x5678` `lui $t0, 0x1234`

2.25.6 [5] <2.6,2.10>执行上表中的代码序列后，求寄存器$t0 中的值。

2.25.7 [5] <2.6，2.10>请写出与上表中的汇编语言等价的 C 代码。（假定可以取到32 位整数中的最大常量是 16 位。）

习题 2.26

本练习将探索 MIPS 中分支和跳转指令的范围。以下问题使用下表中的十六进制数据。

a.	0x00001000
b.	0xFFFC0000

2.26.1 [10] <2.6，2.10>如果 PC 中的地址是 0x00000000，请问使用多少条分支指令（不能使用跳转指令）才能使 PC 指向上表中的地址？

2.26.2 [10] <2.6，2.10>如果 PC 中的地址是 0x00000000，请问需要使用多少条跳转指令（不能使用寄存器跳转指令和分支指令）才能使 PC 指向上表中的地址？

2.26.3 [10] <2.6，2.10>为了减少 MIPS 程序的大小，MIPS 的设计者已经决定将 I 型指令中立即数字段从 16 位变为 8 位。如果 PC 指向地址 0x0000000，请问需要多少条分支指令才能将 PC 设置为上表中的地址？

以下问题将用到对 MIPS 指令集体系结构的修改。

a.	8 寄存器
b.	10 位立即数/地址字段

2.26.4 [10] <2.6，2.10>如果 MIPS 处理器的指令集改变，那么指令的格式也必须变化。对于上表中每种修改建议，请问对 beq 指令地址范围的有什么影响？假设指令还是32 位长并且任何对 I 型指令的修改只能是增加或减少 beq 指令的立即数字段。

2.26.5 [10] <2.6，2.10>如果 MIPS 处理器的指令集改变，那么指令的格式也必须变化。对于上表中每种修改建议，请问对于跳转指令的地址范围有什么影响？假设指令还是 32 位长并且任何对 J 型指令格式的修改只能是影响跳转指令的地址字段。

2.26.6 [10] <2.6，2.10>如果 MIPS 处理器的指令集改变，那么指令的格式也必须变化。对于上表中每种修改建议，请问对于寄存器跳转指令的地址范围有什么影响？假设指令还是 32 位长。

习题 2.27

以下问题是关于探索 MIPS 指令集体系结构中不同的寻址模式的。这些不同的寻址模式在下表中列出。

a.	寄存器寻址
b.	PC 相对寻址

2.27.1 [5] <2.10>上表中是 MIPS 指令集的不同寻址模式。请为不同的 MIPS 寻址模式给出示例性的 MIPS 指令。

2.27.2 [5] <2.10>对于 2.27.1 题中的指令，请问这些指令的指令格式是什么？

2.27.3 [5] <2.10>请列出每种 MIPS 寻址模式的优缺点，并写出展示这些优缺点的 MIPS 代码。

以下问题将使用下表中的 MIPS 汇编代码来探索 MIPS I 型指令中立即数字段的权衡。

a.	0x00000000	`lui $s0, 100`
	0x00000004	`ori $s0, $s0, 40`

（续）

b.	0x00000100	addi	$t0, $0, 0x0000
	0x00000104	lw	$t1, 0x4000($t0)

2.27.4 [15] <2.10> 对于上面的 MIPS 语句，请用十六进制数展现每条指令的位级表示。

2.27.5 [10] <2.10> 通过减少 I 型和 J 型指令的立即数字段的大小，我们可以节省表示指令的位数。如果 I 型指令的立即数字段是 8 位，J 型指令的立即数字段是 18 位，请重写上面的 MIPS 代码来反映这些变化。（避免使用 lui 指令。）

2.27.6 [5] <2.10> 请问与上表中的代码相比，2.27.5 中多使用了多少条指令？

习题 2.28

下表中的 MIPS 汇编代码是加锁时使用的。

```
try:    MOV    R3,R4
        MOV    R6,R7
        LL     R2,0(R2)
        LL     R5,0(R1)
        SC     R3,0(R1)
        SC     R6,0(R1)
        BEQZ   R3,try
        MOV    R4,R2
        MOV    R7,R5
```

2.28.1 [5] <2.11> 对于条件存每一次测试和失败，需要执行多少条指令？

2.28.2 [5] <2.11> 对于上面的加锁取或条件存，请解释为什么这段代码会失败。

2.28.3 [15] <2.11> 重写上面的代码使其可以正确操作，保证避免任意的竞争条件。

下表中的每个条目都是代码和不同寄存器中的内容。标记“（$s1）”表示寄存器$s1 指向的存储器位置中的内容。每个表中的汇编代码是被共享内存空间的并行处理器在相应周期执行的。

a.

处理器 1	处理器 2	周期	处理器 1		内存（$s1）	处理器 2	
			$t1	$t0		$t1	$t0
		0	1	2	99	30	40
ll $t1,0($s1)	ll $t1,0($s1)	1					
sc $t0,0($s1)		2					
	sc $t0,0($s1)	3					

b.

处理器 1	处理器 2	周期	处理器 1			内存（$s1）	处理器 2		
			$s4	$t1	$t0		$s4	$t1	$t0
		0	2	3	4	99	10	20	30
	try:add $t0,$0,$s4	1							
try:add $t0,$0,$s4	ll $t1,0($s1)	2							
ll $t1,0($s1)		3							
sc $t0,0($s1)		4							
beqz $t0,try	sc $t0,0($s1)	5							
add $s4,$0,$t1	beqz $t0,try	6							

2.28.4 [5] <2.11> 对于每个给定的周期，填写表中寄存器的值。

习题 2.29

本练习中前三个问题是关于下面这个形式的临界区的。

```
lock(lk);
operation
unlock(lk);
```

“operation” 使用局部变量（非共享）x，更新共享变量 shvar，代码如下：

	Operation
a.	shvar = shvar + x;
b.	shvar = min(shvar,x);

2.29.1 [10] <2.11> 请为这个临界区编写 MIPS 汇编代码。假设变量 lk 的地址在$a0 中，变量 shvar 的地址在$a1 中，变量 x 的值在$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，例如，你可能使用包含 lock(),unlock(),max() 和 min() 操作的 MIPS 指令。请使用 ll/sc 指令来实现 lock() 操作，unlock() 操作可以简单地使用原始的存指令来实现。

2.29.2 [10] <2.11> 重新解决 2.29.1 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。

2.29.3 [10] <2.11> 比较 2.29.1 和 2.29.2 的代码的最好情况的性能，假设每条指令需要一个周期执行。注意：最好情况意味着 ll/sc 总是执行成功；当我们要 lock() 的时候锁都是被解开的；如果遇到分支指令，将执行完成操作所用指令最少的路径。

2.29.4 [10] <2.11> 以 2.29.2 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。

2.29.5 [10] <2.11> 请解释为什么 2.29.2 中寄存器$a1 中是变量 shvar 的地址，而不是它的值。为什么寄存器$a2 中是变量 x 的值而不是地址？

2.29.6 [10] <2.11> 如果我们要在同一个临界区域中对 2 个共享变量（例如，shvar1 和 shvar2）原子性地执行相同的操作，我们可以简单地使用 2.29.1 题中的方法来完成这个功能（简单地将两个更新放在 lock 操作和相应的 unlock 操作之间）。请解释为什么我们不能使用 2.29.2 题中的方法，例如，为什么不能使用 ll/sc 来访问两个共享变量，通过这种方式来保证两个更新像一个原子操作一样执行。

习题 2.30

汇编伪指令并不是 MIPS 指令集的一部分，但是经常在 MIPS 程序中出现。下表中包含一些 MIPS 伪指令，编译后将翻译成其他的 MIPS 汇编指令。

a.	move $t1, $t2
b.	beq $t1, small, LOOP

2.30.1 [5] <2.12> 对于上表中的每一条伪指令，请写出完成相同功能的最短的实际 MIPS 指令序列。（某些情况下你可能要使用临时寄存器。表中 large 表示这个数字需要 32 位来表示，small 表示这个数字正好适合 16 位表示。）

下表中包含一些 MIPS 伪指令，编译后将翻译成其他的 MIPS 汇编指令。

a.	la $s0, v
b.	blt $a0, $v0, Loop

2.30.2 [5] <2.12> 上表中的指令在链接阶段需要编辑吗？为什么？

习题 2.31

下表中包含两个不同过程的链接级的细节。本练习中你将代替链接器来完成工作。

a.

过程A

	地址	指令	
文本段	0	lw $a0,0($gp)	
	4	jal 0	
	…	…	
数据段	0	(X)	
	…	…	

重定向信息	地址	指令	依赖
	0	lw	X
	4	jai	B

符号表	地址	符号	
	—	X	
	—	B	

过程B

	地址	指令	
文本段	0	sw $a1,0($gp)	
	4	jal 0	
	…	…	
数据段	0	(Y)	
	…	…	

重定向信息	地址	指令	依赖
	0s	sw	Y
	4	jal	A

符号表	地址	符号	
	—	Y	
	—	A	

b.

过程A

	地址	指令	
文本段	0	lui $at,00	
	4	ori $a0,$at,0	
	8	jal 0	
	…	…	
数据段	0	(X)	
	…	…	

重定向信息	地址	指令类型	依赖
	0	lui	X
	4	ori	X
	8	jal	B

符号表	地址	符号	
	—	X	
	—	B	

过程B

	地址	指令	
文本段		sw $a0,0($gp)	
	4	jmp0	
	…	…	
	0x180	jr $ra	
	…	…	
数据段	0	(Y)	
	…	…	

重定向信息	地址	指令类型	依赖
	0	sw	Y
	4	jmp	FOO

符号表	地址	符号	
	—	Y	
	0x180	FOO	

2.31.1 [5] <2.12> 链接上面的目标文件组成可执行文件头。假设过程 A 的文本大小是 0x140，数据大小是 0x40。过程 B 的文本大小是 0x300，数据大小是 0x50。并假设存储器的分配策略如图 2-13 所示。

2.31.2 [5] <2.12> 如果存在的话，请说出可执行文件大小有何限制。

2.31.3 [5] <2.12> 请写出你对分支和跳转指令限制的理解。为什么汇编器在目标文件中直接实现分支和跳转指令会有问题？

习题 2.32

本练习的前三个问题中，函数 swap 使用 C 语言定义如下，而不是图 2-24 中的代码。

a.

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

b.

```
void  swap(int * p) {
      int temp;
      temp = * p;
      * p = * (p + 1);
      * (p + 1) = * p;
}
```

2.32.1 [10] <2.13> 请将这个函数转换成 MIPS 汇编代码。

2.32.2 [5] <2.13> sort 函数需要进行什么修改?

2.32.3 [5] <2.13> 如果我们对 8 位的字节排序，而不是 32 位字，2.32.1 题中的 swap 的 MIPS 代码需要做什么修改?

本练习剩下的 3 个问题中，我们假设对图 2-27 中的 sort 函数进行以下修改：

a.	使用 s 寄存器替换 t 寄存器。
b.	使用 bltz 指令替换

2.32.4 [5] <2.13> 请问这些修改对图 2-27 中保存和恢复寄存器的代码有影响吗。

2.32.5 [10] <2.13> 当对一个已经排好序的含有 10 个元素的数组进行排序的时候，这些修改对执行指令的数量有什么影响?

2.32.6 [10] <2.13> 当对一个已经按降序（和 sort() 函数的方向相反）排好序的含有 10 个元素的数组进行排序的时候，这些修改对执行指令的数量有什么影响?

习题 2.33

本练习中的问题涉及下面的函数，其代码是基于数组的：

a.

```
int find(int a[],  int n,  int x) {
  int i;
  for(i = 0;i ! = n;i + +)
        if(a[i] = = x)
              return i;
  return -1;
}
```

b.

```
int count(int a[],  int n,  int x) {
  int res = 0;
  int i;
  for(i = 0;i ! = n;i + +)
        if(a[i] = = x)
              res = res + 1;
  return res;
}
```

2.33.1 [10] <2.14> 请将函数转换成 MIPS 汇编代码。

2.33.2 [10] <2.14> 请将函数转换成基于指针的代码（用 C 语言）。

2.33.3 [10] <2.14> 请将 2.33.2 题中基于指针的 C 代码转换成 MIPS 汇编代码。

2.33.4 [5] <2.14> 请将 2.33.1 题中基于数组的代码和 2.33.3 题中基于指针的代码在最坏情况下，每次非最后一次循环迭代（nonlast loop iteration）的执行指令的数量做对比。注意：最坏的情况发生在代码执行时条件分支都选择最长路径执行时，例如，如果有一个 if 语句，那么条件检查的结果是执行较多指令的路径。然而，如果条件检查的结果将导致循环退出的话，那么我们假设执行的路径是使我们继续执行循环的路径。

2.33.5 [5] <2.14> 请比较 2.33.1 题中基于数组的代码和 2.33.3 题中基于指针的代码所需临时寄存器（t-registers）的数量。

2.33.6 [5] <2.14> 如果寄存器 $t0 ~ $t7 和 $a0 ~ $a3 在 MIPS 调用规范中都是被调用者保存的，就像 $s0 ~ $s7，请问 2.33.4 题的答案有什么变化?

习题2.34

下表包含 ARM 汇编语言代码。以下问题是关于把 ARM 汇编语言代码翻译成 MIPS。

a.		MOV	r0,	#10	;init loop counter to 10
	LOOP:	ADD	r0,	r1	;add r1 to r0
		SUBS	r0,	1	;decrement counter
		BNE	LOOP		;if Z=0 repeat loop
b.		ROR	r1,	r2, #4	;r1 = $r2_{3:0}$ concatenated with $r2_{31:4}$

2.34.1 [5] <2.16> 将上表中的 ARM 汇编语言代码翻译成 MIPS 汇编语言代码。假设 ARM 寄存器 r0、r1 和 r2 依次保存与 MIPS 寄存器$s0、$s1 和$s2 的相同的值。如果需要可以使用 MIPS 临时寄存器($t0 等)。

2.34.2 [5] <2.16> 对于上表中的 ARM 汇编语言指令，请用位字段表示这些 ARM 指令。

下表中包含 MIPS 汇编语言代码。以下问题是关于把 MIPS 汇编语言代码翻译成 ARM。

a.	slt	$t0,	$s0,	$s1
	blt	$t0,	$0,	FARAWAY
b.	add	$s0,	$s1,	$s2

2.34.3 [5] <2.16> 请为上表中的 ARM 汇编语言代码找到相应的 MIPS 汇编语言代码序列。

2.34.4 [5] <2.16> 请写出 ARM 汇编语言代码的位字段表示。

习题2.35

ARM 处理器提供一些 MIPS 不支持的寻址模式。以下问题是关于这些新的寻址模式。

a.	LDR	r0, [r1]	;r0 = memory[r1]
b.	LDMIA	r0, {r1,r2,r4}	;r1 = memory[r0], r2 = memory[r0 + 4] ;r4 = memory[r0 + 8]

2.35.1 [5] <2.16> 请说明上表中的 ARM 汇编代码的寻址模式的类型是什么。

2.35.2 [5] <2.16> 请为上表中的 ARM 汇编指令写出完成同样数据传输功能的 MIPS 汇编指令序列。

以下的问题，你将比较用 ARM 和 MIPS 指令集编写的代码。下表中的代码用 ARM 指令集中的指令编写。

a.		LDR	r0,	=Table1	;load base address of table
		LDR	r1,	#100	;initialize loop counter
		EOR	r2,	r2, r2	;clear r2
	ADDLP:	LDR	r4,	[r0]	;get first addition operand
		ADD	r2,	r2, r4	;add to r2
		ADD	r0,	r0, #4	;increment to next table element
		SUBS	r1,	r1, #1	;decrement loop counter
		BNE	ADDLP		;if loop counter ! =0,go to ADDLP
b.		ROR	r1,	r2, #4	;r1 = $r2_{3:0}$ concatenated with $r2_{31:4}$

2.35.3 [10] <2.16> 对于上面的 ARM 汇编代码，请写出功能相等的 MIPS 汇编代码例程。

2.35.4 [5] <2.16> 请问这段代码一共要执行多少条 ARM 汇编指令？一共要执行多少条 MIPS 汇编指令？

2.35.5 [5] <2.16> 假设 MIPS 汇编例程的平均 CPI 与 ARM 汇编例程的平均 CPI 相等，并且 MIPS 处理器的操作频率是 ARM 处理器的 1.5 倍。请问 ARM 处理器比 MIPS 处理器快多少？

习题2.36

ARM处理器支持立即数常数的方法很有趣，本练习将研究这些区别。下表中是ARM指令。

a.	`ADD, r3, r2, r1, LSL #3 ;r3 = r2 + (r1 < <3)`
b.	`ADD, r3, r2, r1, ROR #3 ;r3 = r2 + (r1,rotated_right 3 bits)`

2.36.1 [5] <2.16> 请为上表中的ARM汇编代码编写等价的MIPS代码。

2.36.2 [5] <2.16> 如果寄存器R1包含常量值8，请重新编写MIPS代码使得所需的MIPS汇编指令最少。

2.36.3 [5] <2.16> 如果寄存器R1包含常量值0x06000000，请重新编写MIPS代码使得所需的MIPS汇编指令最少。

下表中是MIPS指令。

a.	`addi r3, r2, 0x1`
b.	`addi r3, r2, 0x8000`

2.36.4 [5] <2.16> 请为上表中的MIPS汇编代码编写等价的ARM汇编代码。

习题2.37

本练习是探索MIPS和x86指令集之间的区别的。下表中包含x86汇编代码。

a.	`mov edx, [esi + 4 * ebx]`

b.	`STRRT: mov ax, 00101100b`
	`mov cx, 00000011b`
	`mov bx, 11110000b`
	`and ax, bx`
	`or ax, cx`

2.37.1 [10] <2.17> 请为给定的例程编写伪代码。

2.37.2 [10] <2.17> 写出与给定例程功能相等的MIPS指令序列。

下表中包含x86汇编指令。

a.	`mov edx, [esi + 4 * ebx]`
b.	`add eax, 0x12345678`

2.37.3 [5] <2.17> 对于每一条汇编指令，请写出表示这个指令的每个位字段的大小。将标签MY_FUNCTION当做32位的常量。

2.37.4 [10] <2.17> 请编写等价的MIPS汇编语句。

习题2.38

x86指令集包含REP前缀，这个前缀可以使指令重复执行给定的次数或一直执行直到条件得到满足。本练习的前三个问题涉及以下x86指令。

	Instruction	Interpretation
a.	REP MOVSB	Repeat until ECX is zero; Mem8[EDI] = Mem8[ESI], EDI = EDI + 1, ESI = ESI + 1, ECX = ECX - 1
b.	REP MOVSD	Repeat until ECX is zero; Mem32[EDI] = Mem32[ESI], EDI = EDI + 1, ESI = ESI + 1, ECX = ECX - 1

2.38.1 [5] <2.17> 请问在典型情况下，这条指令用在什么地方？

2.38.2 [5] <2.17> 请编写与表中指令完成相同操作的 MIPS 代码。假设$a0、$a1、$a2 和$a3 分别与 ECX、EDI、ESI 和 EAX 对应。

2.38.3 [5] <2.17> 假设 x86 指令读、写存储器各需要一个周期，寄存器更新需要一个周期，MIPS 每执行一条指令也需要一个周期。请问当 ECX 非常大的时候，使用 x86 指令来代替功能相等的 MIPS 代码，程序执行速度能加快多少？假设 x86 和 MIPS 的时钟周期相等。

本练习剩下的三个问题涉及下面的函数，我们以 C 和 x86 汇编形式给出。对于 x86 指令，我们还给出了在 x86 编程指令格式下该指令的长度和对指令的解释（这条指令是做什么的）。注意：与 MIPS 相比 x86 体系结构的寄存器非常少，所以 x86 的调用规范将所有的参数都压栈。x86 函数的返回值通过寄存器 EAX 传递给调用者。

	C code	x86 code

a.	`int f(int a,int b) {`
	`  return a+b;`
	`}`
		`f:push % ebp ;1B,push % ebp to stack`
		`mov % esp,% ebp ;2B,move % esp to % ebp`
		`mov 0xc(% ebp),% eax ;3B,load 2nd arg to % eax`
		`add 0x8(% ebp),% eax ;3B,add 1st arg to % eax`
		`pop % ebp ;1B,restore % ebp`
		`ret ;1B,return`

b.	`void f(int * a,int * b) {`
	`  * a=* a+* b;`
	`  * b=* a;`
	`}`
		`f:push % ebp ;1B,push % ebp to stack`
		`mov % esp,% ebp ;2B,move % esp to % ebp`
		`mov 8(% ebp),% eax ;3B,load 1st arg into % eax`
		`mov 12(% ebp),% ecx ;3B,load 2nd arg into % ecx`
		`mov(% eax),% edx ;2B,load * a into % edx`
		`add(% ecx),% edx ;2B,add * b to % edx`
		`mov % edx,(% eax) ;2B,store % edx to * a`
		`mov % edx,(% ecx) ;2B,store % edx to * b`
		`pop % ebp ;1B,restore % ebp`
		`ret ;1B,return`

2.38.4 [5] <2.17> 请将该函数翻译成 MIPS 汇编代码，并比较 x86 代码和 MIPS 代码的大小（占用指令存储器的字节数）。

2.38.5 [5] <2.17> 如果处理器每周期可以执行 2 条指令，那么每周期内处理器应至少有能力读 2 条连续的指令。请解释在 MIPS 中和在 x86 中分别如何实现上述要求？

2.38.6 [5] <2.17> 如果 MIPS 指令执行需要一个周期，x86 指令执行需要一个周期再加上每个要执行的读或写操作的一个周期，请问使用 x86 代替 MIPS 所获得的加速比是多少？假定 x86 和 MIPS 的时钟周期相同，函数执行时选择尽可能最短的路径执行（例如，每个循环都是立即退出的并且每个 if 语句都选择能引向函数返回的方向来执行）。注意 x86 中的 ret 指令从栈中读返回地址。

习题 2.39

下表中给出不同指令类型的 CPI 值。

	算术	存/取	分支
a.	2	10	3
b.	1	10	4

2.39.1 [5] <2.18> 假定执行指定程序中有下面给定的指令：

	指令（百万条）
算术	500
存/取	300
分支	100

如果操作频率是 5 GHz，求处理器的执行时间。

2.39.2 [5] <2.18>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？

2.39.3 [5] <2.18>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十倍的方法，那么机器的性能整体加速又是多少？

下表是指令执行中不同类型指令的比例。

	算术	存/取	分支
a.	60%	20%	20%
b.	80%	15%	5%

2.39.4 [5] <2.18>对于上表中给出的指令混合比例，假设算术指令执行需要 2 周期，存/取指令需要 6 周期，分支指令需要 3 周期，求平均 CPI。

2.39.5 [5] <2.18>为了提高 25% 的性能，在存/取指令和分支指令执行时间不变的情况下，平均情况下算术指令执行要多少个周期？

2.39.6 [5] <2.18>为了提高 50% 的性能，在存/取指令和分支指令执行时间不变的情况下，平均情况下算术指令执行要多少个周期？

习题 2.40

本练习的前三个问题与下面 MIPS 汇编语言给出的函数有关。注意，编写这个函数的程序员落入了如下陷阱：将MIPS 当做一个字编址的机器。而事实上 MIPS 按字节编制。

a.

```
;int f(int a[],  int n,  int x);
f:      move    $v0,$zero       ;  ret=0
        move    $t0,$zero       ;  i=0
L:      add     $t1,$t0,$a0     ;  &(a[i])
        lw      $t1,0($t1)      ;  read a[i]
        bne     $t1,$a2,S       ;  if(a[i]==x)
        addi    $v0,$v0,1       ;  ret++;
S:      addi    $t0,$t0,1       ;  i++
        bne     $t0,$a1,L       ;  repeat if i!=n
        jr      $ra             ;  return ret
```

b.

```
;void f(int * a,int * b,int n);
f:    move    $t0,  $a0         ;p=a
      move    $t1,  $a1         ;p=b
      add     $t2,  $a2,$a0     ;&(a[n])
L:    lw      $t3,  0($t0)      ;read * p
      lw      $t4,  0($t1)      ;read * q
      add     $t3,  $t3,$t4     ;* p+* q
      sw      $t3,  0($t0)      ;* p=* p+* q
      addi    $t0,  $t0,1       ;p=p+1
      addi    $t1,  $t1,1       ;q=q+1
      bne     $t0,  $t2,L       ;repeat if p!=&(a[n])
      jr      $ra               ;return
```

注意，MIPS 汇编语言中字符“;”表示这行剩余的部分都是注释。

2.40.1 [5] <2.18>MIPS 体系结构访问（lw 和 sw）字大小时，需要存储器字对齐，例如地址的最低 2 个有效位必须都是 0。如果地址不是字对齐的，处理器将置“bus error”异常。请说明这种对齐对执行这个函数有什么影响。

2.40.2 [5] <2.18>假设“a”是一个指向数组首地址的指针，这个数组中的元素都是一字节大小。如果用 lb（load byte）和 sb（store byte）依次替换 lw 和 sw，请问这个函数正确吗？注意：lb 从存储器读出

一字节，进行符号扩展，然后将数据放入目的寄存器，而 sb 将寄存器最低 8 个有效位存到存储器。

2.40.3 [5] <2.18> 修改代码，使之对于 32 位的整数可以正确执行。

本练习中剩下 3 个问题是关于为数组分配存储器，将一些数据填入数组，调用图 2-27 中的 sort 函数，然后打印数组。程序的 main 函数如下表（用 C 和 MIPS 给出）：

main code in C	MIPS version of the main code

main() {
　　int * v;
　　int n=5;
　　v=my_alloc(5);
　　my_init(v,n);
sort(v,n);
.
.
.
}
	main:
	li　　$s0, 5
	move　　$a0, $s0
	jal　　my_alloc
	move　　$s1, $v0
	move　　$a0, $s1
	move　　$a1, $s0
	jal　　my_init
	move　　$a0, $s1
	move　　$a1, $s0
	jal　　sort

函数 my_alloc 的定义如下（用 C 和 MIPS 给出）。注意，编写这个函数的程序员落入如下陷阱：在函数定义的范围之外使用了指向函数内定义的自动变量的指针 arr。

my_ alloc in C	MIPS code for my_ alloc

int * my_alloc(int n) {
　　int arr[n];
　　return arr;
}
	my_alloc:
	　　addu　　$sp,$sp, -4　　;Push
	　　sw　　$fp,0($sp)　　;$fp to stack
	　　move　　$fp,$sp　　;Save $sp in $fp
	　　sll　　$t0,$a0, 2　　;We need 4* n bytes
	　　sub　　$sp,$sp, $t0　　;Make room for arr
	　　move　　$v0, $sp　　;Retuen address of arr
	　　move　　$sp,$fp　　;Return $sp from $fp
	　　lw　　$fp, 0(sp)　　;Pop $fp
	　　addiu　　$sp,$sp,4　　;from stack
	　　jr　　ra

函数 my_init 的定义如下（MIPS 代码）：

a.

```
my_init:
   move     $t0, $zero      ; i=0
   move     $t1, $a0
L: sw       $zero, 0($t1)   ; v[i]=0
   addiu    $t1, $t1, 4
   addiu    $t0, $t0, 1     ; i=i+1
   bne      $t0, $a1, L     ; until i==n
   jr       $ra
```

b.

```
my_init:
   move     $t0, $zero      ; i=0
   move     $t1, $a0
L: sub      $t2, $a1, $t0
   sw       $t2, 0($t1)     ; a[i]=n-i
   addiu    $t1, $t1, 4
   addiu    $t0, $t0, 1     ; i=i+1
   bne      $t0, $a1, L     ; until i==n
   jr       $ra
```

2.40.4 [5] <2.18> 当 main 代码执行后，请问在“`jal sort`”指令执行前，数组 v 中的数据（所有 5 个元素的值）是多少？

2.40.5 [15] <2.18，2.13> 当执行到 `sort` 函数进入最外层循环执行第一次迭代前，请问此时数组 v 中的数据是多少？假设当 main 代码开始执行时（在指令“`li $s0,5`”执行前），寄存器`$sp`、`$s0`、`$s1`、`$s2` 和`$s3` 中的值分别是 0x1000、20、40、7 和 1。

2.40.6 [10] <2.18，2.13> 在指令“`jal sort`”执行后，返回 main 函数时，由 v 指向的含有 5 个元素的数组中的数据是什么？

小测验答案

2.2 节　CBA

2.3 节　B

2.4 节　B

2.5 节　D

2.6 节　AB 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。合适的右移将一个最字右边的区域都移走，将 0 留在字中。注意到“逻辑与”操作会保留原始的值，移动操作对将需要的区域移动到字的最右边。

2.7 节　1）全对。2）A

2.8 节　两个都对。

2.9 节　1）B　2）C

2.10 节　1）D　2）F　3）D

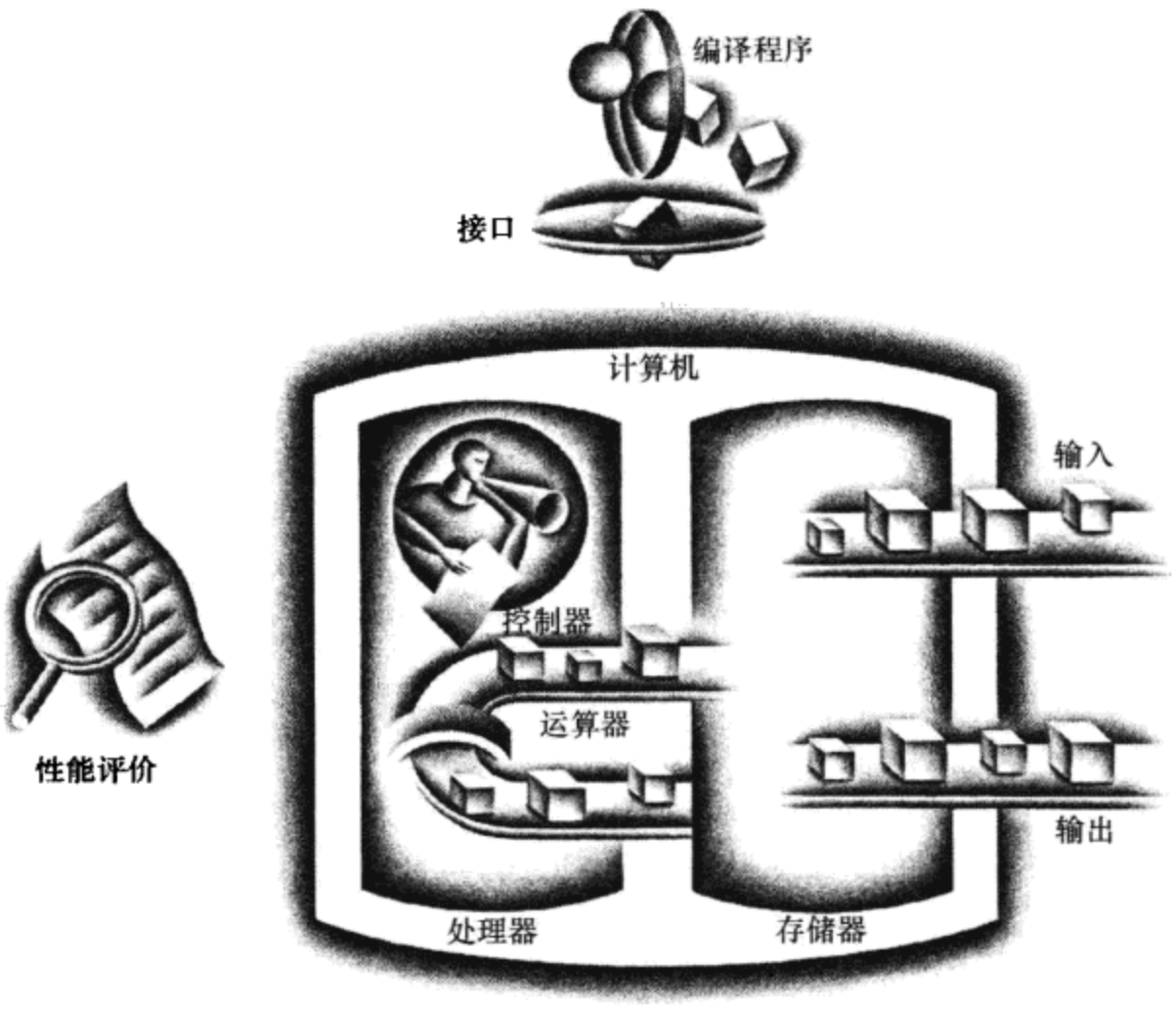
2.11 节　两个都对。

2.12 节　D。

计算机的算术运算

数值的精确度是科学的灵魂。

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



计算机的五大经典部件

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？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基本原理，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。

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：减数在简单的取反后再进行加法操作。

举例　**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}。

答案

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 = 7_{10} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_2 = 6_{10} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_2 = 13_{10} \end{array}$$

只有右边四位发生变化。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。

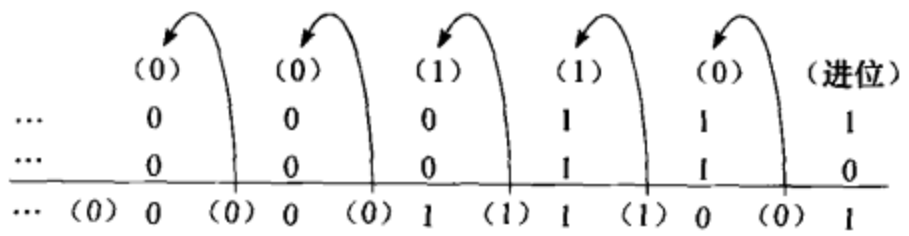


图3-1　二进制加法，显示了从右到左的进位

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。

7_{10}减去6_{10}可以直接操作：

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 = 7_{10} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_2 = 6_{10} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10} \end{array}$$

或者通过加上-6的二进制补码来实现：

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 = 7_{10} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_2 = -6_{10} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10} \end{array}$$

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，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，$x-y=x+(-y)$，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。

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。

操作	源操作数A	源操作数B	发生溢出时的结果
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 编译器会根据源操作数的类型来选择相应的算术指令。

光盘中的附录 C 描述了做加减法的**算术逻辑单元**[㊀]的硬件实现。

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**[㊁]，在许多计算机系统中也叫做**中断**[㊂]。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。）

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。

3.2.1　多媒体算术运算

因为桌面微处理器都有自身的图形显示设备，所以随着晶体管生产预算的增加，处理器不可避免地需要增加对图形处理操作的支持。

在许多图像系统中，使用 8 位表示三原色中的一种，再用额外的 8 位表示像素的位置。此外在电话会议和视频游戏中，对扬声器和麦克风的使用需要对声音进行表达和处理。音频采样需要高于 8 位的精度，一般 16 位是足够的。

一般微处理器都有对 8 位和半字位长的特殊支持，以便在存储时可占用更少的空间（见 2.9 节）。但是，因为在传统的整数程序中针对这些位长的算术操作使用较少，从而除了数据传输操作外，处理器很少支持此类位长的操作。现在，体系结构设计者认识到在图像和音频应用中，总是需要对这些称为短向量的 8 位或半字位长数据进行相同的操作。通过把 64 位加法器的进位链分段，处理器就可以同时处理八个 8 位长的源操作数，或者四个 16 位长的源操作数，或者两个 32 位长的源操作数。而加法器分段的代价很小。这样的扩展称为向量计算或单指令多数据计算（见 2.17 节和第 7 章）。

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

㊀ **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。
㊁ **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。
㊂ **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。）

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，起初声音逐渐增大，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合理的。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。图 3-3 给出了现代指令集中多媒体扩展常用的一些算术和逻辑操作。

指令类别	源操作数
无符号加/减	八个8位长或者四个16位长
饱和加/减	八个8位长或者四个16位长
最大/最小	八个8位长或者四个16位长
平均值	八个8位长或者四个16位长
右移/左移	八个8位长或者四个16位长

图 3-3　桌面计算机中支持多媒体操作的汇总表

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 节描述 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 ≠; go to 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: 2^32 - $t1 - 1)
sltu  $t3,$t3,$t2                #(2^32 - $t1 - 1) < $t2
                                 #⇒2^32 - 1 < $t1 + $t2
bne   $t3,$zero,Overflow         #if(2^32 - 1 < $t1 + $t2) goto overflow
```

3.2.2　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，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。

随着多媒体应用的逐渐流行，出现了支持易于并行执行的短操作数的算术指令。

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。那么将会使用哪些 MIPS 指令？

A. 取数使用 lbu、lhu；算术操作采用 add、sub、mult、div；存数采用 sb、sh。

B. 取数使用 lb、lh；算术操作采用 add、sub、mult、div；存数采用 sb、sh。

C. 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： 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 log2 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，见光盘中的附录 C.6。

3.3　乘法

乘法令人恼怒，除法更甚；比例运算困扰着我，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　　1000_{10}

乘数　×　　1001_{10}

1000

0000

0000

1000

积　　1001000_{10}

第一个源操作数称为被乘数，第二个源操作数称为乘数，最终的结果称为积。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。

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 3-4 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。

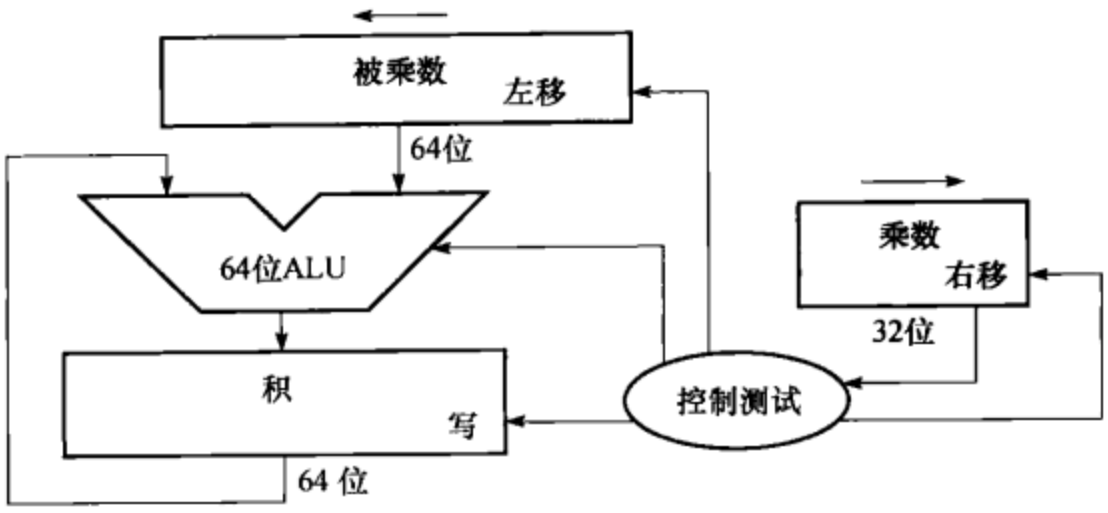


图 3-4　第一版乘法器硬件结构

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。（光盘中的附录 C 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 3-5 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 1.8 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。

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 3-6 所示为修正后的硬件。

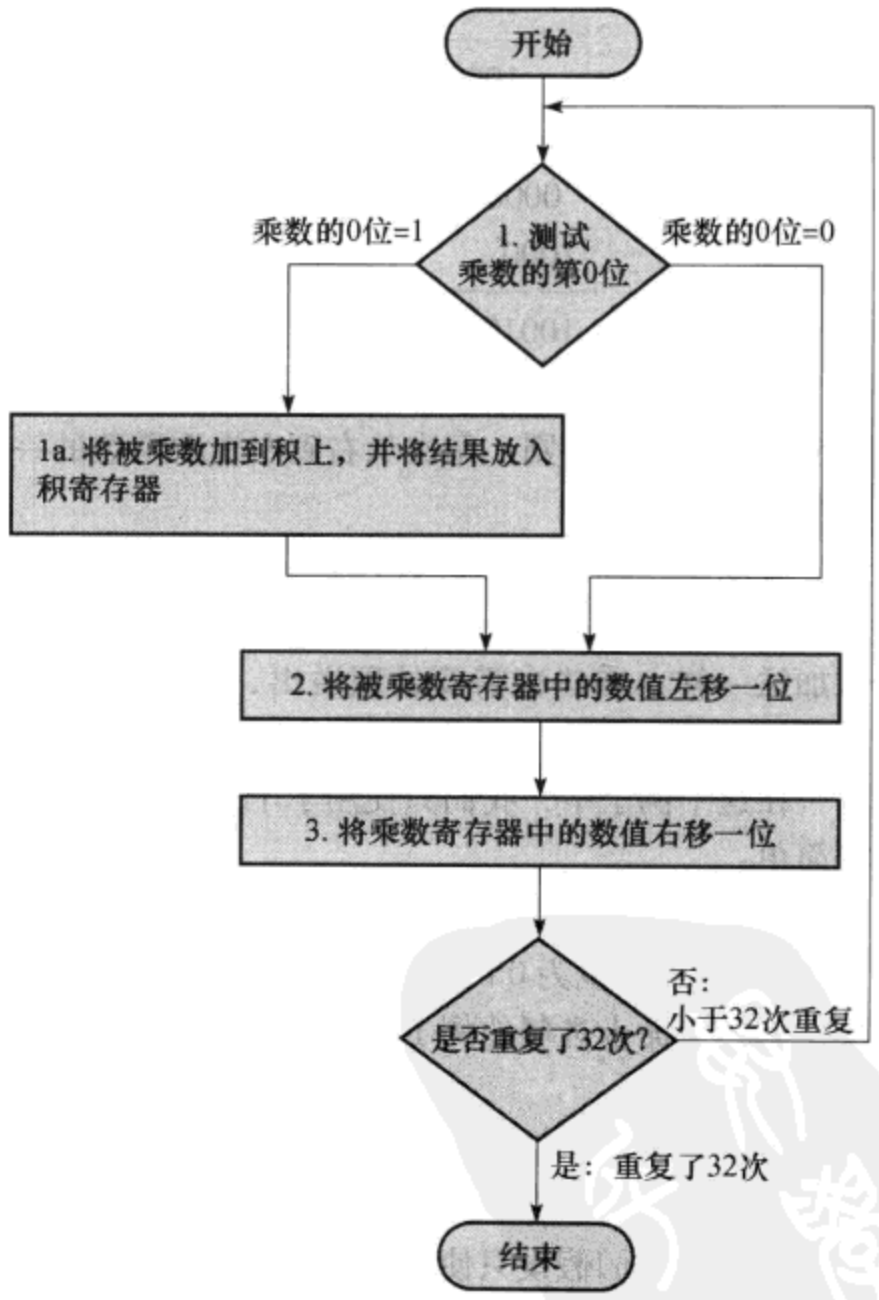


图 3-5　第一种乘法算法

其硬件结构见图 3-4。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。

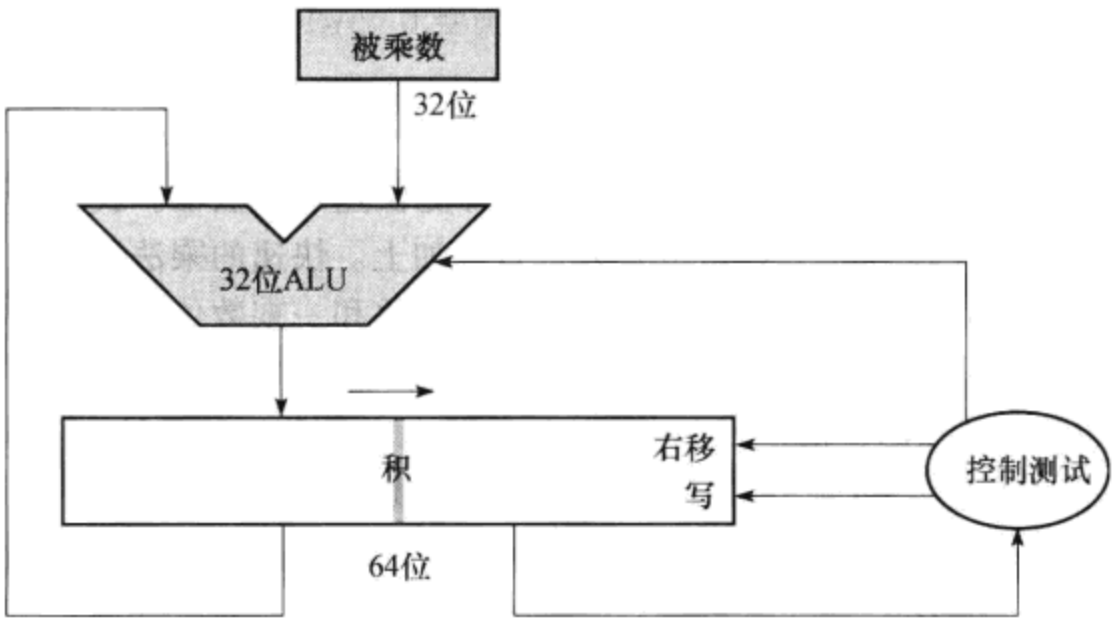


图 3-6　乘法器硬件的改进版

与图 3-4 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。这些变化使用加粗标明。（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 3-4 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。

举例　乘法算法

为了节省空间，使用的是 4 位长的数，计算 $2_{10} \times 3_{10}$，或 $0010_2 \times 0011_2$ 的积。

答案

图 3-7 给出了按图 3-5 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。

迭代	步骤	乘数	被乘数	积
0	初始值	001①	0000 0010	0000 0000
1	1：1⇒积=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	**000①**	0000 0100	0000 0010
2	1：1⇒积=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：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：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 3-7　使用图 3-5 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。

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 31[⊖] 的加法器栈。一种替换的方法是将 31[⊖] 个加法器组织成一个并行树，如图 3-8 所示。这样，我们只需要等待 $\log_2(32)$ 即 5 次 32 位长加法的时间，而不是等待 31[⊖] 次加法的时间。

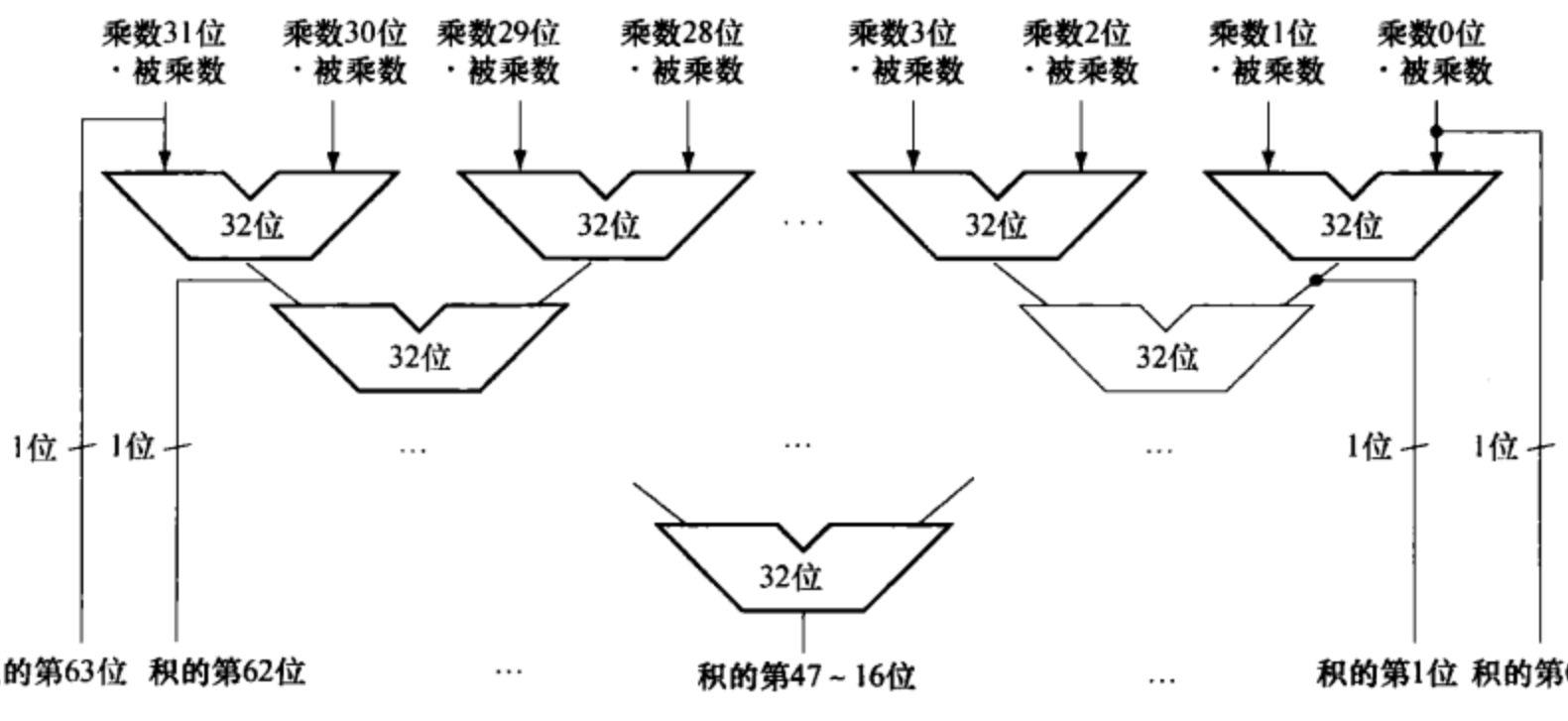


图 3-8　快速乘法器硬件结构

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*（见光盘中的附录 C.6），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（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 提供了两个指令：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

乘法硬件只是简单的移位和加法，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。

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不能被 32 位所表示。对于

⊖ 原文为 32。——译者注
⊜ 原文为 32。——译者注
⊜ 原文为 32。——译者注

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。

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，1532 年

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，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}：

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

除法中的两个源操作数，称为**被除数**㊀和**除数**㊁，结果称为**商**㊂，还有一个第二结果，称为**余数**㊃。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。

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 3-9 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 3-10 给出了第一种除法算法的三个步骤。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。所以需要在第一步中减去除数；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，则通过将除数加到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。

㊀ 被除数：被除的数。
㊁ 除数：用于对被除数进行除法的数。
㊂ 商：除法的主要结果；乘以除数并加上余数产生被除数的数。
㊃ 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。

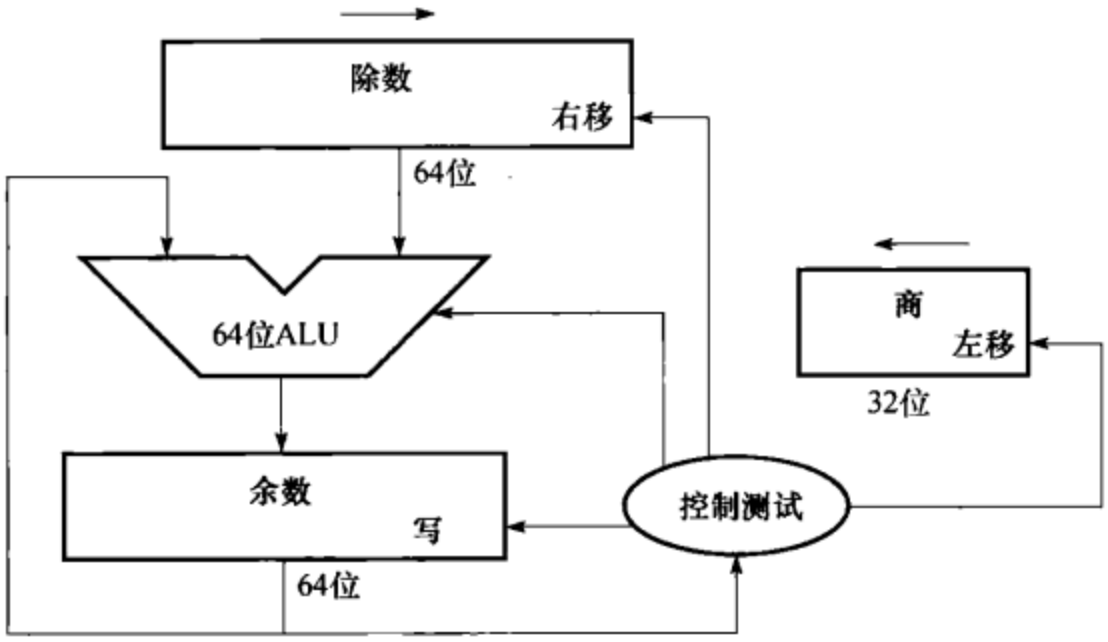


图 3-9　第一种除法器硬件结构

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。

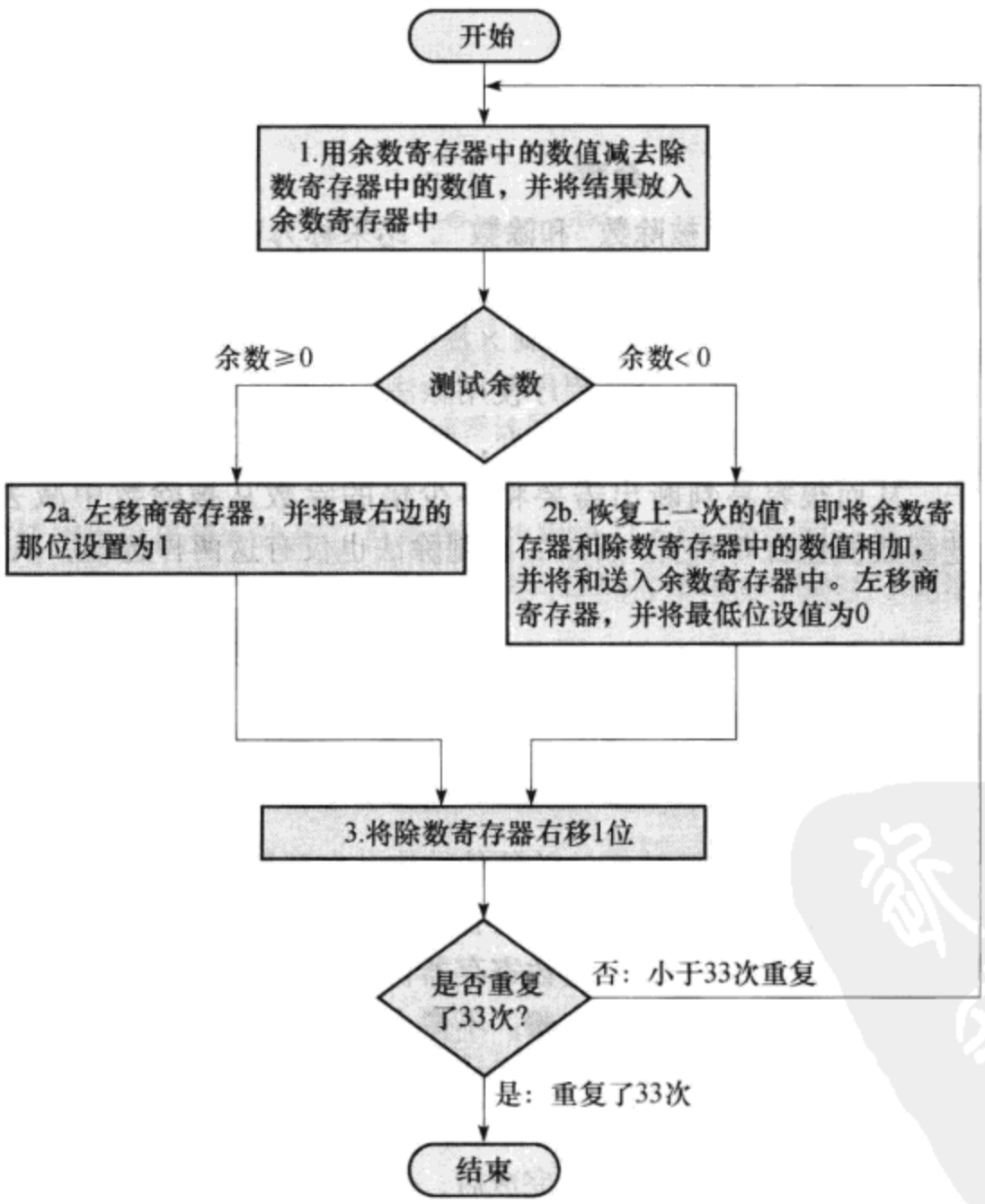


图 3-10　第一种除法算法，其硬件结构见图 3-9

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，根据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。

举例　**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3-11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。这个算法令人惊讶的是，需要n+1步来获得适当的商和余数。

迭代	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	**①110 0111**
	2：余数<0⇒加上除数，左移商，商的第0位置0	**0000**	0010 0000	**0000 0111**
	3：右移除数	0000	**0001 0000**	0000 0111
2	1：余数=余数-除数	0000	0001 0000	**①111 0111**
	2：余数<0⇒加上除数，左移商，商的第0位置0	**0000**	0001 0000	**0000 0111**
	3：右移除数	0000	**0000 1000**	0000 0111
3	1：余数=余数-除数	0000	0000 1000	**①111 1111**
	2：余数<0⇒加上除数，左移商，商的第0位置0	**0000**	0000 1000	**0000 0111**
	3：右移除数	0000	**0000 0100**	0000 0111
4	1：余数=余数-除数	0000	0000 0100	**⓪000 0011**
	2：余数≥0⇒左移商，商的第0位置1	**0001**	0000 0100	0000 0011
	3：右移除数	0001	**0000 0010**	0000 0011
5	1：余数=余数-除数	0001	0000 0010	**⓪000 0001**
	2：余数≥0⇒左移商，商的第0位置1	**0011**	0000 0010	0000 0001
	3：右移除数	0011	**0000 0001**	0000 0001

图3-11　除法的例子，采用图3-10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3-12所示为改进后的硬件结构。

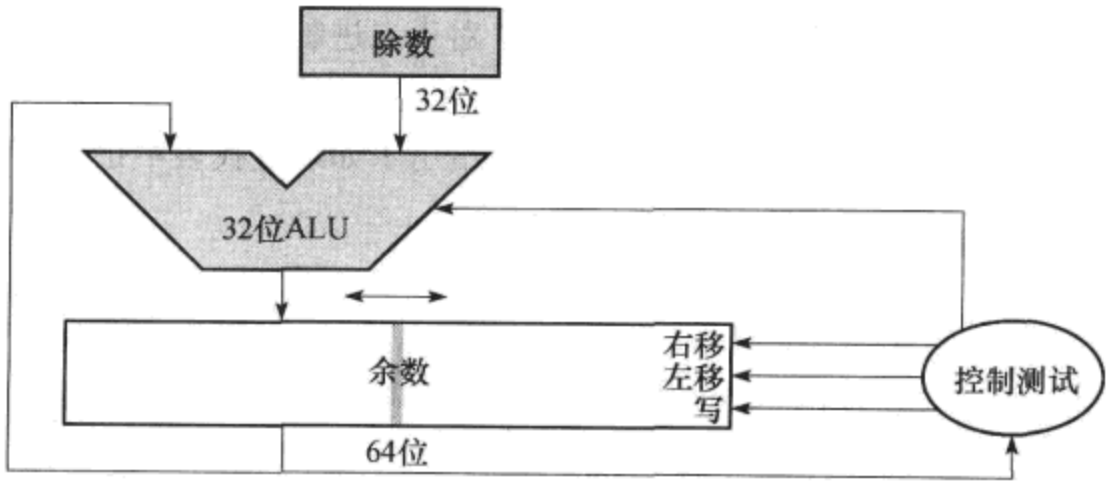


图3-12　除法器的一种改进版本

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3-9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3-6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。如果两者的符号相异，则商为负。

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 $\pm 7_{10}$ 除以 $\pm 2_{10}$ 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，其原因是如果那样，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：

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

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

因此，正确的有符号除法算法在源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

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减的值。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 3.8 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 3-6 和图 3-12 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。图 3-13 汇总了 MIPS 体系结构为本方案所添加的指令。

MIPS 汇编语言

类别	指令	举例	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 的拷贝
	从 Lo 中获得	mflo $s1	$s1 = Lo	用来获得 Lo 的拷贝
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
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

图 3-13　MIPS 核心结构

类别	指令	举例	含义	备注
条件跳转	相等时跳转	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 3-13 （续）

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。在本书文前的 MIPS 参考数据中列出了 MIPS 机器语言。

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。而同溢出一样，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）

$3\,155\,760\,000_{10}$即 $3.155\,76_{10}\times 10^{9}$（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**[⊖]。一个采用科学计数法表示的数，若没有前导零且小数点左边只有一位整数，则可称为**规格化数**[⊜]。例如，$1.0_{10}\times 10^{-9}$就是规格化的科学

⊖ 十进制小数点左边只有一位整数的记数法。

⊜ 没有前导零的浮点记数法。

计数，但 $0.1_{10} \times 10^{-8}$ 和 $10.0_{10} \times 10^{-10}$ 就不是。

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）。

这类计算机算术称为**浮点**[㊀]计算，因为其表示的二进制小数点是不固定的，与整数相似。编程语言 C 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：

$$1.xxxxxxxxx_2 \times 2^{yyyy}$$

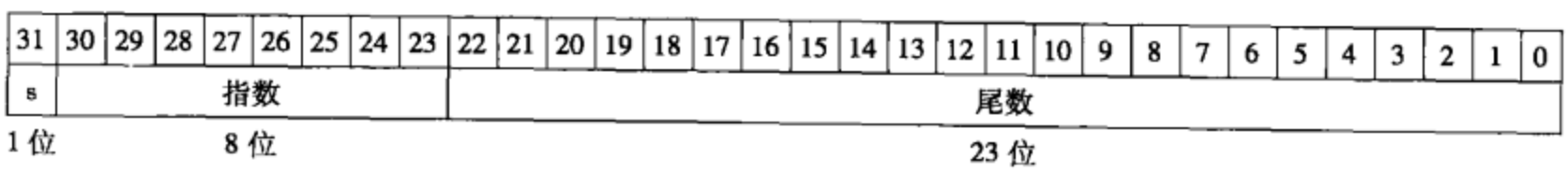
（尽管计算机对指数也同其他数一样表示为以 2 为基的形式，但这里为了简化记数，我们用十进制来表示指数。）

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**[㊁]位宽和**指数**[㊂]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。

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。



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**[㊃]）表示指数太大而不能在指数域表示。

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

㊀ 二进制小数点不固定的表达数的记数法。
㊁ 位于浮点数的尾数字段，其值在 0 和 1 之间。
㊂ 位于浮点数的指数字段，表示小数点的位置。
㊃ 正的指数太大而导致指数域放不下的情况。

给出错误答案。为了和上溢区分开来，将其称为**下溢**[⊖]。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**[⊜]浮点算术；**单精度**[⊜]浮点就是前面的格式。

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。



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 $00\cdots00_2$ 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3-14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。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，无穷的目的是为了形成实数的拓扑闭集。）

单精度		双精度		表示的数
指数	尾数	指数	尾数	
0	000	0	000	0
0	非零	0	非零	±非规格化数
1～254	任意	1～2046	任意	±浮点数
255	0	2047	0	±无穷
255	非零	2047	非零	NaN（非数）

图3-14　IEEE 754浮点数的编码

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。这个信息也可以在本书文前的MIPS参考数据的第4列中找到。

⊖ 负的指数太大而导致指数域放不下的情况。
⊜ 浮点数由两个32位的字表示。
⊜ 浮点数由一个32位的字表示。

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 的目的是为了让程序员推迟程序中的一些测试和决定，等到方便的时候才进行。

IEEE 754 的设计者还希望浮点表示能够容易地处理整数比较，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做的浮点数分类，因为在有着相同的符号的情况下，指数大的数其值就大。

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。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

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

从而，单精度数的表示范围从

$$\pm 1.0000\ 0000\ 0000\ 0000\ 0000\ 000_2 \times 2^{-126}$$

到

$$\pm 1.1111\ 1111\ 1111\ 1111\ 1111\ 111_2 \times 2^{+127}。$$

让我们演示一下浮点表示。

举例　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

$$-3/4_{10} \text{ 或者 } -3/2^2_{10}$$

它的二进制小数形式为

$$-11_2/2^2_{10} \text{ 或者 } -0.11_2$$

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

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_2) \times 2^{(126-127)}$$

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
0	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

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2) \times 2^{(1022-1023)}$$

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	0	1	0	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

让我们再看反方向的一个例子。

举例　**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	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，

$$\begin{aligned}(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0\end{aligned}$$

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整数操作，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：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 16 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 大型机也开始支持 IEEE 754 标准。

3.5.2　浮点加法

为了说明浮点加法中的问题，我们将用科学计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，我们需要将有较小指数的数向较大指数的数对齐：

$$1.610_{10} \times 10^{-1} = 0.1610_{10} \times 10^0 = 0.01610_{10} \times 10^1$$

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：

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

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：

$$10.015_{10} \times 10^1 = 1.0015_{10} \times 10^2$$

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

$$1.0015_{10} \times 10^2$$

舍入为有 4 个十进制数有效位的数

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

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 3-15 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 和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）。因此，对于单精度，最大的指数为 127，最小的指数为 −126。双精度则分别为 1023 和 −1022。

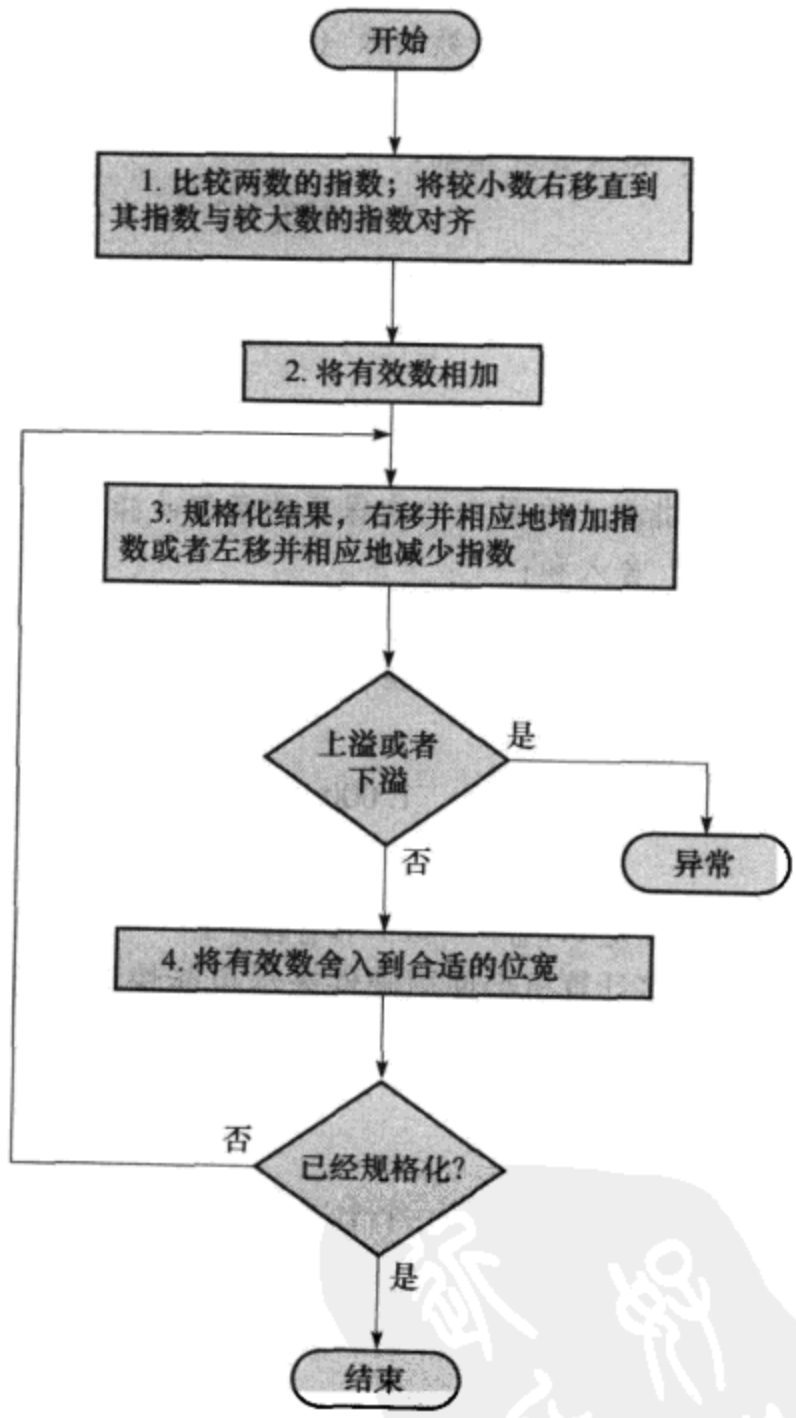


图 3-15　浮点加

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。

举例　**二进制浮点加法**

按照图3-15中的算法，尝试将0.5_{10}和-0.4375_{10}用二进制相加。

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：

$$\begin{aligned} 0.5_{10} &= 1/2_{10} &&= 1/2^{1}_{10} \\ &= 0.1_{2} &&= 0.1_{2}\times 2^{0} &&= 1.000_{2}\times 2^{-1} \\ -0.4375_{10} &= -7/16_{10} &&= -7/2^{4}_{10} \\ &= -0.0111_{2} &&= -0.0111_{2}\times 2^{0} &&= -1.110_{2}\times 2^{-2} \end{aligned}$$

现在，我们按照如下的算法执行：

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

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

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

$$\begin{aligned} 1.000_{2}\times 2^{-4} &= 0.0001000_{2} &&= 0.0001_{2} \\ &= 1/2^{4}_{10} &&= 1/16_{10} &&= 0.0625_{10} \end{aligned}$$

这就是0.5_{10}和-0.4375_{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3-16给出了浮点加的基本结构示意图。

3.5.3　浮点乘法

首先，我们手算一个十进制乘法的例子，其中数用科学计数法表示：$1.110_{10}\times 10^{10}\times 9.200_{10}\times 10^{-5}$。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：

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

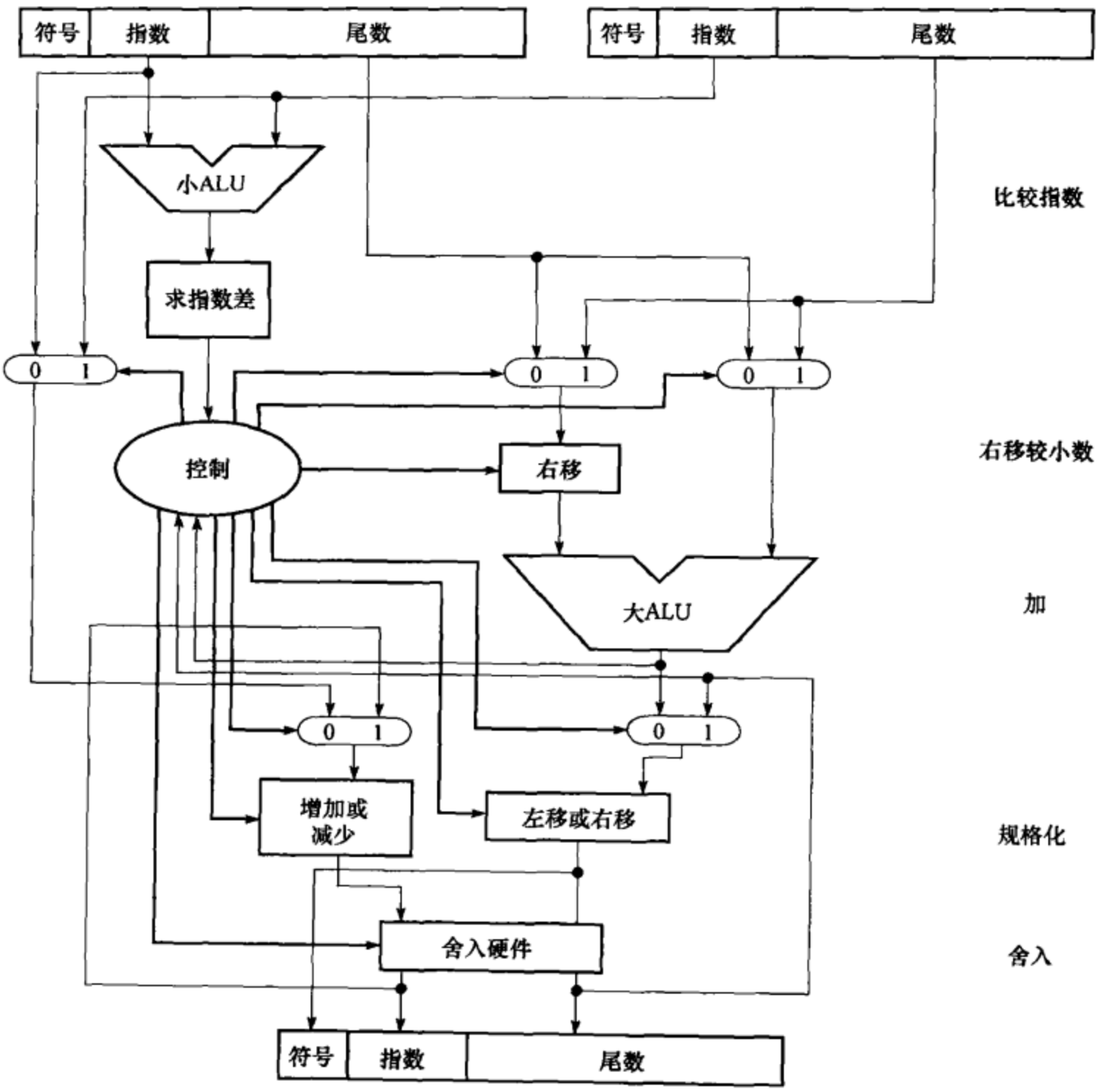


图 3-16　用于浮点加的算术单元的结构框图

图 3-15 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。

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：

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

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：

$$10.212000_{10}$$

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：

$$10.212_{10} \times 10^5 = 1.0212_{10} \times 10^6$$

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

$$1.0212_{10} \times 10^6$$

舍入为只有四位的有效数

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

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

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

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 3-17 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）。

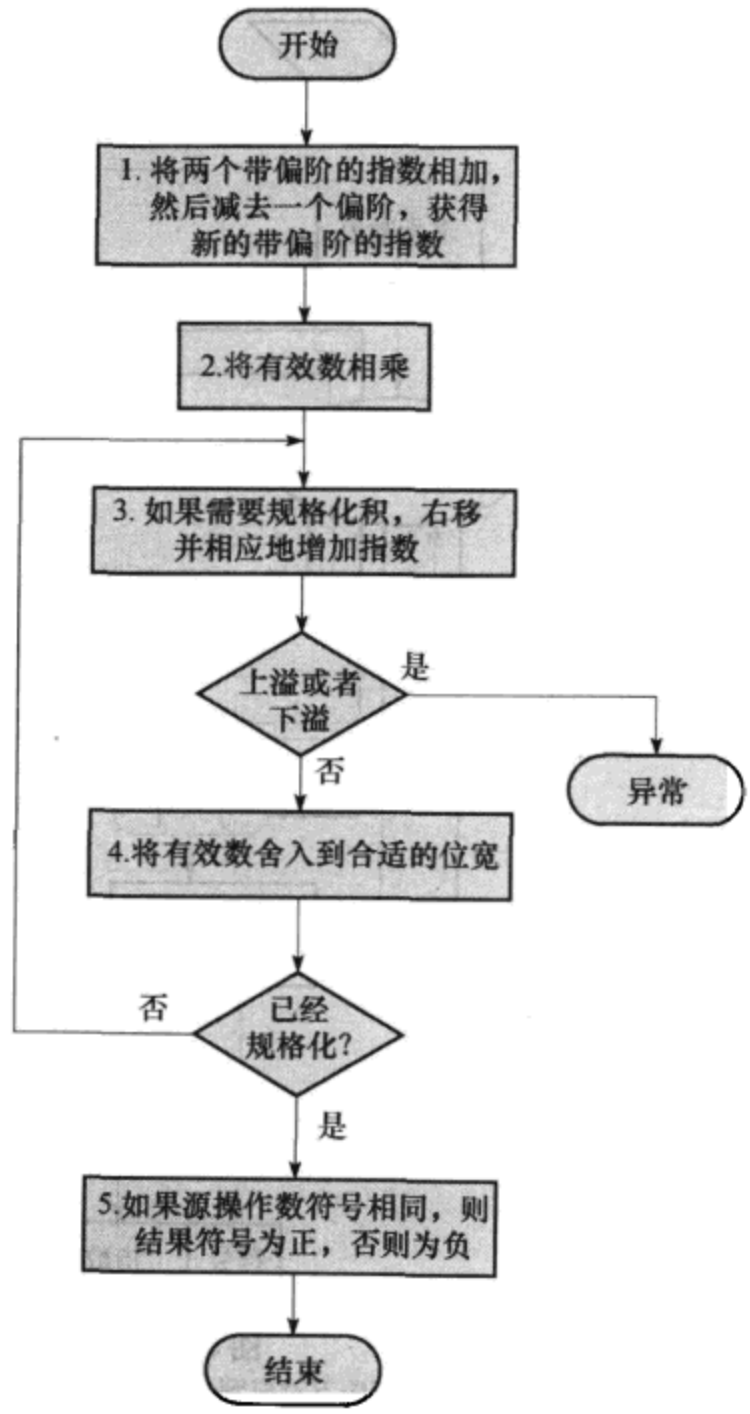


图 3-17　浮点乘法

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，则需要重复步骤 3。

举例　**二进制浮点乘法**

按照图 3-17 中的步骤，试计算 0.5_{10} 和 -0.4375_{10} 的乘积。

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

答案

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：

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127) = -3 + 127 = 124$$

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 $1.110000_2 \times 2^{-3}$，但是我们需要保存4位，所以为 $1.110_2 \times 2^{-3}$。

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，$254 \geqslant 124 \geqslant 1$，所以指数域可以表达。）

步骤4：对积舍入没有使其发生变化：

$$1.110_2 \times 2^{-3}$$

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

$$-1.110_2 \times 2^{-3}$$

为了检查结果，将其转化为十进制：

$$-1.110_2 \times 2^{-3} = -0.001110_2 = -0.00111_2 = -7/2^5_{10} = -7/32_{10} = -0.21875_{10}$$

而 0.5_{10} 和 -0.4375_{10} 的积确实是 -0.21875_{10}。

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（bclt）和浮点比较为假跳转（bclf）

浮点比较根据比较条件，将比较结果设为真或者假，然后浮点跳转根据比较结果条件决定是否跳转。

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 F6
add.s     $f2,$f4,$f6    #F2 = F4 + F6 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 3-18 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 3-19 给出了这些指令的编码。

MIPS 浮点操作数

名称	举例	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^{30} 个存储字	Memory [0], Memory [4], ..., 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换出的寄存器

MIPS 浮点汇编语言

分类	指令	举例	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
条件跳转	浮点标志真则跳转	bc1t 25	如果（cond = =1）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	bc1f 25	如果（cond = =0）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）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）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	举例						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 3-18　以前介绍过的 MIPS 浮点体系结构

附录 B. 10 有更详细的介绍。这个信息也可以在本书文前的 MIPS 参考数据的第 2 列里找到。

op (31: 26):								
28~26 31~29	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0 (000)	Rfmt	Bltz/gez	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						

op (31: 26) =010001 (FlPt), (rt (16: 16) =0 =>c=f, rt (16: 16) =1 =>c=t), rs (25: 21):								
23~21 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 (FlPt), (上面的f: 10000 =0 =>f=s; 10001 =0 =>f=d), funct (5: 0):								
2~0 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3-19 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B给出了全部指令。这个信息也可以在本书文前的MIPS参考数据的第2列里找到。

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整数和浮点操作，单独的寄存器会稍微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单独的用于浮点寄存器和内存之间传送数据的指令。

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。

举例 **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存在内存中)
        lwc1    $f18,const9($gp)#$f18=9.0(9.0存在内存中)
```

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                 #返回
```

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。

举例 **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X=X+Y*Z，其中X、Y、Z都是32×32的矩阵。

```
void mm(double x[][],double y[][],double z[][])
{
     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)
        x[i][j]=x[i][j]+y[i][k]* z[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x[i][j]处于上面循环的最里面。因为循环变量是k，不影响x[i][j]，所以我们可以避免在每次迭代时载入和存储x[i][j]。相反，编译器每次在循环外将x[i][j]载入一个寄存器，然后将y[i][k]和z[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x[i][j]。

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   (行大小/循环终止)
        li   $s0,0           #i =0;     初始化第一个循环
L1:     li   $s1,0           #j =0;     开始第二个循环
L2:     li   $s2,0           #k =0;     开始第三个循环
```

要计算 x[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 (x 的行大小)
```

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* 行大小 +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 个字节，我们用左移 3 位来代替：

```
sll     $t2,$t2,3          #$t2 =[i][j]的字节偏移量
```

下面我们将这个和加上 x 的基地址，得到 x[i][j]的地址，然后将双精度数 x[i][j]载入到 $f4 寄存器中：

```
addu    $t2,$a0,$t2        #$t2 =x[i][j]的字节地址
l.d     $f4,0($t2)         #$f4 =8 个字节的 x[i][j]
```

接着的 5 条指令类似于刚才的 5 条：计算双精度数 z[k][j]的地址，然后将其载入。

```
L3:     sll     $t0,$s2,5          #$t0 =k* 2^5 (z 的行大小)
        addu    $t0,$t0,$s1        #$t0 =k* 行大小 +j
        sll     $t0,$t0,3          #$t0 =[k][j]的字节偏移量
        addu    $t0,$a2,$t0        #$t0 =z[k][j]的字节地址
        l.d     $f16,0($t0)        #$f16 =8 个字节的 z[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 y[i][k]的地址，然后将其载入。

```
        sll     $t0,$s0,5          #$t0 =i* 2^5 (y 的行大小)
        addu    $t0,$t0,$s2        #$t0 =i* 行大小 +k
        sll     $t0,$t0,3          #$t0 =[i][k]的字节偏移量
        addu    $t0,$a1,$t0        #$t0 =y[i][k]的字节地址
        l.d     $f18,0($t0)        #$f18 =8 个字节的 y[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 y、z 的元素相乘，然后累加到 $f4 中。

```
mul.d   $f16,$f18,$f16     #$f16 =y[i][k]* z[k][j]
add.d   $f4,$f4,$f16       #$f4 =x[i][j] +y[i][k]* z[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 x[i][j]。

```
addiu   $s1,$s2,1          #$k k +1
```

```
bne     $s2,$t1,L3          #if(k!=32)go to L3
s.d     $f4,0($t2)          #x[i][j]=$f4
```

类似地，最后四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
...
```

精解：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。

精解：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。例如，add.ps $f0，$f2，$f4等价于add.s $f0，$f2，$f4和add.s $f1，$f3，$f5。

精解：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，协处理器和加速器等术语已经过时了。

精解：正如3.4节提到的，加速除法比乘法更有挑战性。除了SRT，还有一种利用快速乘法器的技术，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/x，然后将其乘以另一源操作数。如果不计算更多的位，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。

精解：Java在定义浮点数组类型和操作时遵循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

不像整数那样可以精确地表示在最大数和最小数之间的所有数，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^{53}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。

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**[⊖]和**舍入位**[⊖]。让我们用一个十进制的例子来说明它们的作用。

⊖ 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。

⊖ 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。

举例　**使用保护位来舍入**

将 $2.56_{10} \times 10^0$ 和 $2.34_{10} \times 10^2$ 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 $2.56_{10} \times 10^0$ 变为 $0.0256_{10} \times 10^2$。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：

$$\begin{array}{rr} & 2.3400_{10} \\ + & 0.0256_{10} \\ \hline & 2.3656_{10} \end{array}$$

因此，和为 $2.3656_{10} \times 10^2$。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到49之间时舍掉，在51到99之间时向上舍入。向上舍入这个和，变为 $2.37_{10} \times 10^2$。

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：

$$\begin{array}{rr} & 2.34_{10} \\ + & 0.02_{10} \\ \hline & 2.36_{10} \end{array}$$

答案是 $2.36_{10} \times 10^2$，比上面的结果在最低位上少1。

因为最糟糕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后位（units 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。

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，则截去。这种模式总是在中间情况下将最低位设为0，正如舍入模式的名称叫的那样。这种模式是用得最多的，而且是唯一被Java支持的模式。

使用额外的舍入位的目的是为了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按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 $0.50\ldots00_{10}$ 和 $0.50\ldots01_{10}$。

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...00然后按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...00的，我们舍入后会得到2.35。

精解：PowerPC、SPARC64和AMD SSE5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，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

⊖ 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。

⊜ 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。

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**[⊖]。它已被加入到修订的 IEEE 754 标准里（见 CD 中的 3.10 节）。

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哪些位进行操作。

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，因此限制了其精度；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	lh,sh,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 的存储分类（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 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？

A. $1.0000\ 0000\ 00 \times 2^{0}$ 到 $1.1111\ 1111\ 11 \times 2^{31}$，0

B. $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ 到 $\pm 1.1111\ 1111\ 1 \times 2^{15}$，$\pm 0$，$\pm \infty$，NaN

C. $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ 到 $\pm 1.1111\ 1111\ 11 \times 2^{15}$，$\pm 0$，$\pm \infty$，NaN

D. $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ 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。）

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

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_{2} \times 2^{-126}，即\ 1.0_{2} \times 2^{-149}$$

⊖ 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。

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：结合律

通常情况下，程序首先是为顺序执行编写的，而后才考虑并行的问题，所以一个很自然的问题就是："这两个版本能得到相同的答案吗?" 如果答案是否定的，你会假设在并行版本中有一个错误需要捕捉。

这个做法假设在从顺序执行变为并行时，计算机算术不会影响结果。即，如果将一百万个数相加，那么，无论是用 1 个处理器，还是 1000 个处理器，结果都会相同。这个假设对补码整数是成立的，即使有上溢也是如此。换句话说就是整数加法是符合结合律的。

然而，由于浮点数是实数的近似而且计算机算术是有限精度的，它并不对浮点数成立。即，浮点加法是不符合结合律的。

举例　**测试浮点加法的结合律**

看看 $x+(y+z)=(x+y)+z$ 是否成立。例如，$x=-1.5_{10}\times10^{38}$，$y=1.5_{10}\times10^{38}$，$z=1.0$，并且全部采用单精度表示。

答案

假定浮点能表示很大范围的数，当将两个异号的大数相加，然后再加上一个小数时，就会出现问题，正如我们看到的：

$$x+(y+z)=-1.5_{10}\times10^{38}+(1.5_{10}\times10^{38}+1.0)=-1.5_{10}\times10^{38}+1.5_{10}\times10^{38}=0.0$$

$$(x+y)+z=(-1.5_{10}\times10^{38}+1.5_{10}\times10^{38})+1.0=(0.0_{10})+1.0=1.0$$

因此 $x+(y+z)\neq(x+y)+z$，所以浮点加法是不符合结合律的。

因为浮点数精度是有限的，并且导致对实际结果的近似，$1.5_{10}\times10^{38}$ 比 1.0 大许多，所以 $1.5_{10}\times10^{38}+1.0$ 仍然是 $1.5_{10}\times10^{38}$。就是为什么 x、y 和 z 的和是 0.0 或者是 1.0 的原因。这依赖于浮点加的顺序，因此浮点加是不符合结合律的。

这个缺陷的一个更加令人烦恼的情况是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。

精解：结合律问题可能导致如下情况发生：当两个处理器以不同的顺序执行一个冗余计算时，可能获得稍微不同的结果，尽管两个结果都是足够精确的。但当用条件跳转指令比较这个浮点数时，可能产生一种 bug，两个处理器选择了不同的跳转方向，而按照常识它们应该向同一个方向跳转。

3.7　实例：x86 的浮点

x86 有着规则的、完全对寄存器进行操作的乘法和除法指令，而不像 MIPS 采用 Hi 和 Lo。（事实上，MIPS 指令集的后续版本增加了类似的指令。）

主要的不同体现在浮点指令上。x86 浮点体系结构不同于世界上任何一种计算机。

3.7.1　x86 浮点体系结构

在 1980 年，Intel 发布了 8087 浮点协处理器。该体系结构扩展了 8086 指令集，包含大约 60 条浮点指令。

Intel 对浮点指令提供了一种堆栈结构：载入指令将数压栈，操作使用栈顶的两个单元作为源操作数，存储进行弹栈。Intel 对这个堆栈结构补充了一些指令和寻址模式从而允许这个体系结构获得一些寄存器 - 存储器模式的益处。除了用栈顶的两个元素作为源操作数外，一个源操作数可以在内存中，或者在栈顶下 7 个片上寄存器之一中。也可以说，在一个完全的堆栈指令集外，补充了有限的寄存器 - 内存指令。

然而，这种混合仍然是一种受限制的寄存器 - 内存模式，因为 load 指令总是将数据移到堆栈的顶端，同时将栈顶指针加 1，而 store 指令只能将栈顶数据移到内存中。Intel 使用 ST 表示栈顶，而用 ST(i)表示栈顶下第 i 个寄存器。

这种体系结构另外一个新颖的特点是源操作数在寄存器堆栈中比在内存中更宽，并且所有的操作都是以这样宽的内部精度来执行。不像 MIPS 最大的位宽为 64 位，在堆栈中的 x86 源操作数达到 80 位宽。数字在载入时自动转换为这种内部的 80 位格式，而在写回内存时转换回相应的大小。尽管双精度扩展精度对于编写数学软件的程序员来说是很有用的，但并不被程序语言支持。

内存数据可以是 32 位的（单精度）或者 64 位的（双精度）浮点数。这些指令的寄存器 - 存储器版本在执行操作之前将内存数据转换为 Intel 的 80 位数据格式。这些数据传送指令可以自动将 16 位和 32 位的整数转化为浮点数。对于整数的 load 和 store 指令，反之亦然。

x86 浮点操作主要可以分为四类：

1）数据移动指令，包括 load、load 常数和 store。

2）计算机算术指令，包括加、减、乘、除、开方根和求绝对值。

3）比较指令，包括将结果发送给整数处理器使其能跳转的指令。

4）超越函数指令，包括正弦、余弦、对数和指数。

图 3-20 给出了 60 条浮点操作中的一部分。注意当我们对这些操作引入操作数模式时，可以得到更多的组合。图 3-21 给出了浮点加的多种情况。

数据传送	算术	比较	超越函数
F{I}LD mem/ST(i)	F{I}ADD{P}mem/ST(i)	F{I}COM{P}	FPATAN
F{I}ST{P}mem/ST(i)	F{I}SUB{R}{P}mem/ST(i)	F{I}UCOM{P}{P}	F2xM1
FLDPI	F{I}MUL{P}mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F{I}DIV{R}{P}mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

图 3-20　x86 浮点指令

我们使用大括号 {} 给出基本操作可选的变量：{I} 表示指令有个整数版本，{P} 表示在操作后会执行弹栈，而 {R} 表示在操作中更换源操作数的顺序。第一列给出了数据传送指令，可将数据移动到内存或者栈顶下的一个寄存器。第一列最后的三个操作是将常数压栈：pi，1.0 和 0.0。第二列给出了上面描述过的算术操作。注意，最后三个只对栈顶进行操作。第三列是比较指令。因为没有专门的浮点跳转指令，比较结果会通过 FSTSW 指令将结果发送给整数 CPU 的 AX 寄存器或者内存中，然后紧跟一条 SAHF 指令对条件码进行设置。这样，浮点比较指令就可以用整数跳转指令来测试。最后一列给出了高级浮点操作。并不是所有的被标记的组合都存在。因此，F{I}SUB{R}{P}代表了这些在 x86 中存在的指令：FSUB, FISUB, FSUBR, FISUBR, FSUBP, FSUBRP。对于整数减法指令，并没有弹栈（FISUBP）或者交换弹栈（FISUBRP）。

指令	操作数	备注
FADD		两个操作数都在堆栈中；结果放在栈顶
FADD	ST(i)	一个源操作数在栈顶下第 i 个寄存器中；结果放在栈顶
FADD	ST(i),ST	一个源操作数在栈顶；结果放在栈顶下第 i 个寄存器中
FADD	mem32	一个源操作数是内存中的 32 位数据；结果放在栈顶
FADD	mem64	一个源操作数是内存中的 64 位数据；结果放在栈顶

图 3-21　有不同源操作数的 x86 浮点加的多种情况

浮点指令使用 8086 的 ESC 操作码和末字节地址指示符来编码（见图 2-47）。内存操作保留了 2 位用于决定操作数是 32 位或者 64 位的浮点数，还是 16 位或者 32 位的整数。同样的 2 位也用于并不访存的版本，以用于决定是否在操作后弹栈，和决定栈顶或者低一些的寄存器是否能得到结果。

在过去，x86 系列的浮点性能远远落后于其他计算机。因此，作为 SSE2 的一部分，Intel 设计了一种更加传统的浮点体系结构。

3.7.2　Intel SIMD 流扩展 2（SSE2）浮点体系结构

第 2 章提到在 2001 年，Intel 给它的体系结构增加了 144 条指令，其包括双精度浮点寄存器和操作。它使用了 8 个 64 位的寄存器用于浮点操作，比起独特的堆栈结构，对浮点操作给了编译器一种不同的目标。编译器可以选择使用 8 个 SSE2 寄存器作为浮点寄存器，就像其他计算机那样。AMD 则扩展到 16 个寄存器，作为 AMD64 的一部分，后者被 Intel 重新标记为 EM64T 来使用。图 3-22 对 SSE 和 SSE2 指令进行了汇总。

除了将一个单精度或者双精度数存放在寄存器中，Intel 还允许多个浮点源操作数一起打包放在一个 128 位的 SSE2 寄存器中：4 个单精度或者 2 个双精度。因此，对于 SSE2 的 16 个浮点寄存器实际上是 128 位宽。如果源操作数可以以 128 位对齐数据的形式存放在内存中，则一条指令就可以进行 128 位的数据传送，从而存取多个源操作数。能够同时执行 4 个单精度（PS）或者 2 个双精度（PD）操作的浮点算术能够支持这种打包的浮点数据格式。这种体系结构的性能是堆栈结构的两倍还要多。

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 3-22　x86 的 SSE/SSE2 浮点指令

xmm 是指一个源操作数是在一个 128 位的 SSE2 寄存器中，而 mem/xmm 是指另一个源操作数或者在内存中或者在一个 SSE2 寄存器中。我们使用大括号 {} 来表示基本操作的可选变量：{SS} 代表标量单精度浮点，即一个 32 位的源操作数存在一个 128 位的寄存器中；{PS} 代表打包的单精度浮点，即四个 32 位的源操作数存在一个 128 位的寄存器中；{SD} 代表标量双精度浮点，即一个 64 位的源操作数存在一个 128 位的寄存器中；{PD} 代表打包的双精度浮点，即两个 64 位的源操作数存在一个 128 位的寄存器中；{A} 代表 128 位的源操作数在内存中对齐；{U} 代表 128 位的源操作数在内存中没有对齐；{H} 代表移动 128 位源操作数的高部分；{L} 代表移动 128 位源操作数的低部分。

3.8　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。

常见的谬误：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

$$\ldots + (x3 \times 2^3) + (x2 \times 2^2) + (x1 \times 2^1) + (x0 \times 2^0)$$

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

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_2$$

根据这个谬论，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。

常见的陷阱：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。

常见的谬误：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3-23）。

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。猜测结果和除数相除，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。



图3-23　1994年11月的一些报刊杂志的文章，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甚至成为电视节目的“十大新闻”。Intel最后花了3亿美元来替换掉有bug的芯片。

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。

下面是Pentium浮点bug事件的时间表：

- 1994 年 7 月：Intel 在 Pentium 上发现了这个 bug。修复这个 bug 需要几十万美元。在正常的 bug 修复程序之后，还要花数月的时间来做修改，重新验证，并把矫正后的芯片投入生产。Intel 计划在 1995 年 1 月将正确的芯片投入生产，估计将会生产 3 到 5 百万带有这个 bug 的芯片。
- 1994 年 9 月：弗吉尼亚州林奇伯格学院的数学家，托马斯 · 内斯里，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。它迅速得到了回应，一些人指出当乘法中有大数时，即使一些小的错误都会被放大：例如，有罕见疾病的人口比例乘以欧洲的人口，可能导致患病人口的错误估计。
- 1994 年 11 月 7 日：《电子工程时代》将这个事情公布在它的首页，然后迅速被其他报纸转载。
- 1994 年 11 月 22 日：Intel 发布了新闻公告：称其为一个“小故障”。Pentium“可能在第 9 位产生错误... 即使是大多数工程师和经济分析师也只需要精确到小数点后第 4 或第 5 位，电子制表软件和文字处理器的使用者不需要担忧……这个世界上只有很少的人会受到影响。到目前为止，我们只听到一名……［仅仅］理论数学家（在夏天之前买的 Pentium 计算机）可能需要关注。”令许多人厌恶的是，消费者需要将自己的应用告诉 Intel，然后由 Intel 决定他们的应用是否需要获得一个新的没有除法 bug 的 Pentium 处理器。
- 1994 年 12 月 5 日：Intel 宣布对于电子制表软件的用户来说，这个缺陷只有 27 000 年才会发生一次。Intel 假设用户每天做 1000 次除法，乘以错误比率，后者是假设浮点数是随机的，为 90 亿分之一而获得的。然后得到 9 百万天，即 27 000 年。事情开始平息下来，尽管 Intel 并没有解释一个普通的用户为什么会随机地访问浮点数。
- 1994 年 12 月 12 日：IBM 研究部门分析了 Intel 对错误比率的计算（可以在 www. mkp. com/books_catalog/cod/links. htm 看到这篇文章）。IBM 声称普通的电子制表程序一天里每 15 分钟会重新计算一次，可以在 24 天里就发生一次和 Pentium 相关的错误。IBM 假设那 15 分钟里每秒有 5000 次除法，这样平均一天就有 420 万次除法。同时假设数字并不是随机分布的，改为以亿分之一的概率来计算。作为结果，IBM 立即停止了所有基于 Pentium 的 IBM 个人计算机的出货。对 Intel 来说，事情又麻烦起来。
- 1994 年 12 月 21 日：Intel 发布了由 Intel 总裁、首席执行官、首席运营官和董事会主席签署的声明：

  “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 五亿美元，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？

在 1997 年 4 月，在 Pentium Pro 和 Pentium II 微处理器里，发现了另一个浮点 bug。当浮点转整数 store 指令（`fist`，`fistp`）碰到一个太大的浮点负数时，在转成整数后数值太大无法放在 16 位或者 32 位字里面，从而将 FPO 状态字设置错误（精度异常代替了无效操作异常）。为了 Intel 的信誉，这次他们发布了对这个 bug 的声明并提供了一个软件补丁来解决——比起 1994 年是完全不一样的反应。

3.9　本章小结

存储程序计算机有一个副作用，就是位模式没有内在的含义。相同的位模式可能代表有符号整数、无符号整数、浮点数和指令等。对这个字操作的指令决定了其意义。

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。

过去的多年里，计算机算术变得非常标准化，极大地增加了程序的可移植性。当今销售的大量计算机都采用了二进制整数补码算术和 IEEE 754 二进制浮点算术。例如，在本书第一次印刷时，所有市面上销售的桌面计算机就开始采用这些标准。

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 3-24 和图 3-25 试图讲明白这点。

MIPS 核心指令	名称	格式	MIPS 算术核心指令	名称	格式
加法	add	R	乘	mult	R
立即数加法	addi	I	无符号乘	multu	R
无符号加法	addu	R	除	div	R
立即数无符号加法	addiu	I	无符号除	divu	R
减法	sub	R	取自 Hi 寄存器	mfhi	R
无符号减法	subu	R	取自 Lo 寄存器	mflo	R
与	AND	R	取自系统控制寄存器（EPC）	mfc0	R
立即数与	ANDi	I	浮点单精度加	add.s	R
或	OR	R	浮点双精度加	add.d	R
立即数或	ORi	I	浮点单精度减	sub.s	R
或非	NOR	R	浮点双精度减	sub.d	R
逻辑左移	sll	R	浮点单精度乘	mul.s	R
逻辑右移	srl	R	浮点双精度乘	mul.d	R
取立即数的高位	lui	I	浮点单精度除	div.s	R
取字	lw	I	浮点双精度除	div.d	R
存字	sw	I	浮点单精度取字	lwc1	I
取无符号半字	lhu	I	浮点单精度存字	swc1	I
存半字	sh	I	浮点双精度取字	ldc1	I
取无符号字节	lbu	I	浮点双精度存字	sdc1	I
存字节	sb	I	浮点真则跳转	bclt	I
链接取（原子更新）	ll	I	浮点假则跳转	bclf	I
条件存（原子更新）	sc	I	浮点单精度比较	c.x.s	R
相等则跳转	beq	I	(x = eq,neq,lt,le,gt,ge)		
不相等则跳转	bne	I	浮点双精度比较	c.x.d	R
跳转	j	J	(x = eq,neq,lt,le,gt,ge)		
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			

图 3-24　MIPS 指令集

本书集中介绍左列的指令。这个信息也可以在文前的 MIPS 参考数据的第 1 列和第 2 列里找到。

图 3-24 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 3-25 的左边是包含了 MIPS 处理器执行的但图 3-24 中没有的指令。我们将全部的硬件指令集称为 MIPS-32。在图 3-25 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。

保留的 MIPS-32	名称	格式	伪 MIPS	名称	格式
异或（rs ⊕ rt）	xor	R	绝对值	abs	rd, rs
异或立即数	xori	I	变号（有符号或者无符号）	negs	rd, rs
算术右移	sra	R	旋转左移	rol	rd, rs, rt
可变的逻辑左移	sllv	R	旋转右移	ror	rd, rs, rt
可变的逻辑右移	srlv	R	乘且不检查上溢（有符号或者无符号）	muls	rd, rs, rt
可变的算术右移	srav	R	乘且检查上溢（有符号或者无符号）	mulos	rd, rs, rt
移至 Hi	mthi	R	除且检查上溢	div	rd, rs, rt
移至 Lo	mtlo	R	除且不检查上溢	divu	rd, rs, rt
取半字	lh	I	求余（有符号或者无符号）	rems	rd, rs, rt
取字节	lb	I	取立即数	li	rd, imm
取字的左边（非对齐）	lwl	I	取地址	la	rd, addr
取字的右边（非对齐）	lwr	I	取双字	ld	rd, addr
存字的左边（非对齐）	swl	I	存双字	sd	rd, addr
存字的右边（非对齐）	swr	I	非对齐取字	ulw	rd, addr
链接取（原子更新）	ll	I	非对齐存字	usw	rd, addr
条件存（原子更新）	sc	I	非对齐取半字（有符号或者无符号）	ulhs	rd, addr
零则移	movz	R	非对齐存半字	ush	rd, addr
非零则移	movn	R	跳转	b	Label
乘和加	madds	R	等于零时跳转	beqz	rs, L
乘和减	msubs	I	比较并跳转（有符号或者无符号）	bxs	rs, rt, L
大于等于零则跳转并链接	bgezal	I	(x = lt,le,gt,ge)		
小于零则跳转并链接	bltzal	I	相等则置位	seq	rd, rs, rt
跳转并链接寄存器	jalr	R	不相等则置位	sne	rd, rs, rt
和零比较并跳转	bxz	I	比较并置位（有符号或者无符号）	sxs	rd, rs, rt
与零近似比较并跳转	bxzl	I	(x = lt,le,gt,ge)		
(x = lt,le,gt,ge)			取数给浮点（s 或者 d）	l.f	rd, addr
与寄存器值近似比较并跳转	bxl	I	浮点数存储（s 或者 d）	s.f	rd, addr
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.f.f	R			
浮点比较（s 或者 d）	c.xn.f	R			

图 3-25　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。

图 3-26 给出了 SPEC2006 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 0.3%。

MIPS 核心指令	名称	整数	浮点	算术核心 + MIPS-32	名称	整数	浮点
加法	add	0.0%	0.0%	浮点双精度加	add.d	0.0%	10.6%
立即数加法	addi	0.0%	0.0%	浮点双精度减	sub.d	0.0%	4.9%
无符号加法	addu	5.2%	3.5%	浮点双精度乘	mul.d	0.0%	15.0%
立即数无符号加法	addiu	9.0%	7.2%	浮点双精度除	div.d	0.0%	0.2%
无符号减法	subu	2.2%	0.6%	浮点单精度加	add.s	0.0%	1.5%
与	AND	0.2%	0.1%	浮点单精度减	sub.s	0.0%	1.8%
立即数与	ANDi	0.7%	0.2%	浮点单精度乘	mul.s	0.0%	2.4%
或	OR	4.0%	1.2%	浮点单精度除	div.s	0.0%	0.2%
立即数或	ORi	1.0%	0.2%	浮点双精度取字	l.d	0.0%	17.5%
或非	NOR	0.4%	0.2%	浮点双精度存字	s.d	0.0%	4.9%
逻辑左移	sll	4.4%	1.9%	浮点单精度取字	l.s	0.0%	4.2%
逻辑右移	srl	1.1%	0.5%	浮点单精度存字	s.s	0.0%	1.1%
取立即数的高位	lui	3.3%	0.5%	浮点真则跳转	bclt	0.0%	0.2%
取字	lw	18.6%	5.8%	浮点假则跳转	bclf	0.0%	0.2%
存字	sw	7.6%	2.0%	浮点双精度比较	c.x.d	0.0%	0.6%
取字节	lbu	3.7%	0.1%	乘	mul	0.0%	0.2%
存字节	sb	0.6%	0.0%	算术右移	sra	0.5%	0.3%
相等则跳转	beq	8.6%	2.2%	取半字	lhu	1.3%	0.0%
不相等则跳转	bne	8.4%	1.4%	存半字	sh	0.1%	0.0%
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%				

图 3-26 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。

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 SPEC2006 整数程序，而整数核心以及算术核心主宰了 SPEC2006 浮点程序，正如下表所列。

指令子集	整型	浮点
MIPS 核心	98%	31%
MIPS 算术核心	2%	66%
剩余的 MIPS-32	0%	3%

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。

3.10 拓展阅读

Gresham 法则（“劣币驱逐良币”），对于计算机则是“快的淘汰慢的，即使快的是错误的。”

——W. Kahan，1992

绝不要放弃，绝不要，永远，永远，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

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 CD 上 3.10 节。

3.11　练习题

习题 3.1

本书介绍了怎样对二进制和十进制数做加减法。然而，其他数字系统在计算机中也很流行。八进制数（基数为 8）系统就是其中之一。下表给出了两组八进制数。

	A	B
a.	5323	2275
b.	0147	3457

3.1.1 [5] <3.2> A 和 B 如果是 12 位的无符号八进制数，它们的和是多少？结果用八进制表示。给出你的计算过程。

3.1.2 [5] <3.2> A 和 B 如果是 12 位的有符号八进制数并以符号 – 数值的形式存储，它们的和是多少？结果用八进制表示。给出你的计算过程。

3.1.3 [10] <3.2> 假设 A 是无符号的，将 A 转换为十进制数。假设 A 是以符号 – 数值的形式存储，再做一次转换。给出你的计算过程。

下表也给出了两组八进制数。

	A	B
a.	2762	2032
b.	2646	1066

3.1.4 [5] <3.2> A 和 B 如果是 12 位的无符号八进制数，它们的差是多少？结果用八进制表示。给出你的计算过程。

3.1.5 [5] <3.2> A 和 B 如果是 12 位的有符号八进制数并以符号 – 数值的形式存储，它们的差是多少？结果用八进制表示。给出你的计算过程。

3.1.6 [10] <3.2> 将 A 转换为二进制数。为什么在计算机里以 8 为基数（八进制）表达数值是一种有吸引力的数字系统？

习题 3.2

在计算机里用十六进制（基数为 16）表达数值也是一种常用的数字系统。事实上，它比八进制更流行。下表给出了两组十六进制数。

	A	B
a.	OD34	DD17
b.	BA1D	3617

3.2.1 [5] <3.2> A 和 B 如果是 16 位的无符号十六进制数，它们的和是多少？结果用十六进制表示。给出你的计算过程。

3.2.2 [5] <3.2> A 和 B 如果是 16 位的有符号十六进制数并以符号 – 数值的形式存储，它们的和是多少？结果用十六进制表示。给出你的计算过程。

3.2.3 [10] <3.2> 假设 A 是无符号的，将 A 转换为十进制数。假设 A 是以符号 – 数值的形式存储，再做一次转换。给出你的计算过程。

下表也给出了两组十六进制数。

	A	B
a.	BA7C	241A
b.	AADF	47BE

3.2.4 [5] <3.2> A 和 B 如果是 16 位的无符号十六进制数，它们的差是多少？结果用十六进制表示。给出你的计算过程。

3.2.5 [5] <3.2> A 和 B 如果是 16 位的有符号十六进制数并以符号 - 数值的形式存储，它们的差是多少？结果用十六进制表示。给出你的计算过程。

3.2.6 [10] <3.2> 将 A 转换为二进制数。为什么在计算机里以 16 为基数（十六进制）表达数值是一种有吸引力的数字系统？

习题 3.3

在给定字的宽度的情况下，如果结果太大而不能正确表示，则发生上溢。如果数太小而不能正确表示，则发生下溢——例如，在做有符号算术时产生了负结果（将两个负整数相加，产生正结果，这种情况也经常被称为下溢。但在本书中，我们将其称为上溢）。下表给出了两组十进制数。

	A	B
a.	69	90
b.	102	44

3.3.1 [5] <3.2> 假设 A 和 B 是 8 位无符号十进制整数，计算 A - B。有上溢、下溢还是都没有？

3.3.2 [5] <3.2> 假设 A 和 B 是 8 位有符号十进制整数并以符号 - 数值的形式存储，计算 A + B。有上溢、下溢还是都没有？

3.3.3 [5] <3.2> 假设 A 和 B 是 8 位有符号十进制整数并以符号 - 数值的形式存储，计算 A - B。有上溢、下溢还是都没有？

下表也给出了两组十进制数。

	A	B
a.	200	103
b.	247	237

3.3.4 [10] <3.2> 假设 A 和 B 是 8 位有符号十进制整数并以补码的形式存储，用饱和算术计算 A + B。结果用十进制表示。给出你的计算过程。

3.3.5 [10] <3.2> 假设 A 和 B 是 8 位有符号十进制整数并以补码的形式存储，用饱和算术计算 A - B。结果用十进制表示。给出你的计算过程。

3.3.6 [10] <3.2> 假设 A 和 B 是 8 位无符号十进制整数，用饱和算术计算 A + B。结果用十进制表示。给出你的计算过程。

习题 3.4

让我们更详细地了解乘法。我们使用下表中的数字。

	A	B
a.	50	23
b.	66	04

3.4.1 [20] <3.3> 用图 3-4 中的硬件结构计算 6 位的八进制无符号整数 A 和 B 的积，并给出一个类似于图 3-7 中的表。你需要给出每一步中各个寄存器的值。

3.4.2 [20] <3.3> 用图 3-6 中的硬件结构计算 8 位的十六进制无符号整数 A 和 B 的积，并给出一个类似于

图 3-7 中的表。你需要给出每一步中各个寄存器的值。

3.4.3 [60] <3.3> 用图 3-4 中的方法计算无符号整数 A 和 B 的积，写出 MIPS 汇编程序。

下表给出两组八进制数。

	A	B
a.	54	67
b.	30	07

3.4.4 [30] <3.3> 当进行有符号数乘法时，一种获得正确结果的方法是首先将被乘数和乘数转化为正数，并保留其原始符号，然后再相应地调整最终结果。用图 3-4 中的硬件结构计算 A 和 B 的积，并给出一个类似于图 3-7 中的表。你需要给出每一步中各个寄存器的值。假设 A 和 B 都是以 6 位的符号 - 数值格式存储的。

3.4.5 [30] <3.3> 当右移一位寄存器时，有数种方法决定移入的位是多少。它可以总是 0，或者总是 1，或者是右边移出的那位（循环移位），或者是简单地将最左边的位保留下来（称为算术右移，因为它保留了被移位数的符号。）用图 3-6 中的硬件结构计算 6 位的补码数 A 和 B 的积，并给出一个类似于图 3-7 中的表。右移采用算术右移。注意，文中描述的算法需要做一些修改才能做这件工作——特别是，如果乘数是负数时，会是不同的做法。给出每一步中各个寄存器的值。

3.4.6 [60] <3.3> 写出 MIPS 汇编程序，用来计算有符号整数 A 和 B 的积。声明你用的是练习题 3.4.4 的方法还是练习题 3.4.5 的方法。

习题 3.5

有很多原因促使我们设计更快的乘法器。有许多不同的方法可以实现这个目标。在下表中，A 代表整数的位宽，B 代表了执行一步操作所需要的时间。

	A（位宽）	B（时间单位）
a.	4	3tu
b.	32	7tu

3.5.1 [10] <3.3> 计算采用图 3-4 和图 3-5 中的方法来执行乘法所需要的时间。设整数位宽是 A，每步操作需 B 个单位时间。假设在第 1a 步中，无论是否加了被乘数还是加 0，加法都得到执行。再假设寄存器已经初始化过了（你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。

3.5.2 [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 A，一个加法需 B 个单位时间。

3.5.3 [20] <3.3> 计算采用图 3-8 中的方法来执行乘法所需要的时间。设整数位宽是 A，一个加法需 B 个单位时间。

习题 3.6

在本练习中，我们将看看几个其他的办法来提高乘法性能，其主要是基于多执行移位和少执行算术操作的思想。下表给出两组十六进制数。

	A	B
a.	24	c9
b.	41	18

3.6.1 [20] <3.3> 正如书中的讨论，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A×B的最好的方法。假设A和B都是8位无符号整数。

3.6.2 [20] <3.3>给出用移位和加法来计算A×B的最好的方法。假设A和B都是8位有无符号整数且以符号－数值的形式存储。

3.6.3 [60] <3.3>写出MIPS汇编程序，用来计算有符号整数的相乘，使用练习题3.6.1中描述的移位和加法。

下表再给出两组十六进制数。

	A	B
a.	42	36
b.	9F	8E

3.6.4 [30] <3.3>Booth算法是另一种做乘法的方法，其可以减少所需的算术操作次数。这个算法已经出现许多年了，关于它如何工作的细节可以在网上找到。它的基本原理是假设一次移位所需时间要少于加法或者减法，基于此来减少所需的算术操作次数。它通过辨识0和1的序列来工作，并且在辨识的同时执行移位。找出这个算法的描述，并详细地分析它是如何工作的。

3.6.5 [30] <3.3>使用Booth算法，逐步给出A和B相乘的结果。假设A和B都是8位补码整数，以十六进制的格式存储。

3.6.6 [60] <3.3>写出MIPS汇编程序，用来执行A和B的相乘，乘法采用Booth算法。

习题3.7

让我们更详细地了解除法。我们使用下表中的八进制数。

	A	B
a.	50	23
b.	25	44

3.7.1 [20] <3.4>用图3-9中的硬件结构计算A除以B，并给出一个类似于图3-11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。

3.7.2 [30] <3.4>用图3-12中的硬件结构计算A除以B，并给出一个类似于图3-11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3-10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3-12中暗示的那个余数寄存器既可右移也可左移的事实。）

3.7.3 [60] <3.4>用图3-9中的方法计算A和B相除，写出MIPS汇编程序。假设A和B都是6位无符号整数。

下表给出两组八进制数。

	A	B
a.	55	24
b.	36	51

3.7.4 [30] <3.4>用图3-9中的硬件结构计算A除以B，并给出一个类似于图3-11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位有符号整数并以符号－数值的形式存储。要包括如何计算商和余数的符号。

3.7.5 [30] <3.4>用图3-12中的硬件结构计算A除以B，并给出一个类似于图3-11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位有符号整数并以符号－数值的形式存储。要包括如

何计算商和余数的符号。

3.7.6 [60] <3.4>用图3-12中的方法计算A和B相除，写出MIPS汇编程序。假设A和B都是有符号整数。

习题3.8

图3-10描述了恢复除算法。因为将除数从余数中减去时可能产生负结果，所以需要将除数加回到余数（从而恢复了数值）。然而，有些算法可以去掉这个多余的加法，可以在网上进行相关的搜索。我们用下表中的两组八进制数来分析此类算法。

	A	B
a.	75	12
b.	52	37

3.8.1 [30] <3.4>用不恢复除算法（nonrestoring division）来计算A除以B，并给出一个类似于图3-11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。

3.8.2 [60] <3.4>用不恢复除算法计算A除以B，写出MIPS汇编程序。假设A和B都是6位有符号（补码）整数。

3.8.3 [60] <3.4>怎么比较恢复和不恢复除法的性能？通过计算每种A除以B的算法所需要的步数来论证。假设A和B都是6位有符号（符号－数值）整数。可以写一个执行恢复和不恢复除法的程序。下表再给出两组八进制数。

	A	B
a.	17	14
b.	70	23

3.8.4 [30] <3.4>用不执行除算法（nonperforming division）来计算A除以B，并给出一个类似于图3-11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。

3.8.5 [60] <3.4>用不执行除算法计算A除以B，写出MIPS汇编程序。假设A和B都是6位有符号补码整数。

3.8.6 [60] <3.4>怎么比较不执行和不恢复除法的性能？通过计算每种A除以B的算法所需要的步数来论证。假设A和B都是6位有符号整数并以符号－数值的形式存储。可以写一个采用不执行和不恢复除法的程序。

习题3.9

除法是很耗时的和困难的，因此《CRAY T3E Fortran优化指南》讲到："对除法最好的策略就是尽可能地避免使用它。"这个练习是检查下面两个执行除法的不同策略。

a.	恢复除法
b.	SRT除法

3.9.1 [30] <3.4>详细地描述算法。

3.9.2 [60] <3.4>用流程图（或者一段高级语言）来描述算法的工作原理。

3.9.3 [60] <3.4>写一段MIPS汇编程序，用来完成使用这种算法的除法。

习题3.10

在冯·诺依曼体系结构中，一组数本身是没有任何含义的。位模式代表的含义完全依赖于如何使用它们。下表以十六进制给出两个位模式。

a.	0x24A60004
b.	0xAFBF0000

3.10.1 [5] <3.5> 如果是补码整数，则这些位模式代表的十进制是多少？如果是无符号整数呢？

3.10.2 [10] <3.5> 如果这些位模式放在指令寄存器中，那么将执行什么 MIPS 指令？

3.10.3 [10] <3.5> 如果是浮点数，则这些位模式代表的十进制数是多少？使用 IEEE 754 标准。

下表给出十进制数。

a.	-1609.5
b.	-938.8125

3.10.4 [10] <3.5> 写出这些十进制数的二进制表达。设采用 IEEE 754 单精度格式。

3.10.5 [10] <3.5> 写出这些十进制数的二进制表达。设采用 IEEE 754 双精度格式。

3.10.6 [10] <3.5> 写出这些十进制数的二进制表达。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）。

习题 3.11

在 IEEE 754 浮点标准中，指数采用“偏阶”（也叫“余 - N”）的格式存储。选用这个方法的原因是我们想要全零的模式能够尽可能地接近零。因为使用了隐含 1，所以如果我们用补码来表示指数，则一个全零的模式可能就是 1！（记住，任何数的 0 次方为 1，故 $1.0^0 = 1$。）IEEE 754 标准中有许多特性来帮助浮点单元运算得更快。然而，在许多老式的机器中，浮点计算采用软件处理，因此也有其他的格式被使用。下表给出十进制数。

a.	5.00736125×10^5
b.	$-2.691650390625 \times 10^{-2}$

3.11.1 [20] <3.5> 写出二进制位模式。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。

3.11.2 [20] <3.5> NVIDIA 采用一种“半”格式，其类似于 IEEE 754 但只有 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。

3.11.3 [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，作为一种有趣的交叉，指数以符号 - 数值的形式存储且符号位在最右端！写出这种格式的二进制位模式。没有陷含 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。

下表给出两组十进制数。

	A	B
a.	-1278×10^3	-3.90625×10^{-1}
b.	2.3109375×10^1	$6.391601562 \times 10^{-1}$

3.11.4 [20] <3.5> 手算 A 和 B 的和，设 A 和 B 以习题 3.11.2 中提到的 16 位 NVIDIA 格式存储（也可以看书中的描述）。假设有保护位、舍入位和粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤。

3.11.5 [60] <3.5>写出计算 A 和 B 的和的 MIPS 汇编程序，设 A 和 B 以习题 3.11.2 中提到的 16 位 NVIDIA 格式存储（也可以看书中的描述）。假设有保护位、舍入位和粘贴位，并采用向最靠近的偶数舍入的模式。

3.11.6 [60] <3.5>写出计算 A 和 B 的和的 MIPS 汇编程序，设 A 和 B 以习题 3.11.1 中提到的格式存储。修改求和程序，设采用习题 3.11.3 中提到的格式。哪种格式程序员更容易处理？和 IEEE 754 格式相比呢？（在这个问题中不要担忧粘贴位。）

习题 3.12

浮点乘比起浮点加更复杂和更有挑战性，但两者和浮点除相比又差了许多。

	A	B
a.	5.66015625×10^0	8.59375×10^0
b.	6.18×10^2	5.796875×10^1

3.12.1 [20] <3.5>手算 A 和 B 的积，设 A 和 B 以习题 3.11.2 中提到的 16 位 NVIDIA 格式存储（也可以看书中的描述）。假设有保护位、舍入位和粘贴位，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，而不用习题 3.4 到 3.6 中描述的技术。注明是否上溢或者下溢。分别以 16 位模式和十进制数写出你的答案。你的结果精确吗？和你用计算器取得的结果相比呢？

3.12.2 [60] <3.5>写出计算 A 和 B 的积的 MIPS 汇编程序，设 A 和 B 以 IEEE 754 格式存储。注明是否上溢或者下溢。（记住，IEEE 754 假设有保护位、舍入位和粘贴位，并采用向最靠近的偶数舍入的模式。）

3.12.3 [60] <3.5>写出计算 A 和 B 的积的 MIPS 汇编程序，设 A 和 B 以习题 3.11.1 中提到的格式存储。修改求积程序，设采用习题 3.11.3 中提到的格式。哪种格式程序员更容易处理？和 IEEE 754 格式相比呢？（在这个问题中不要担忧粘贴位。）

下表再给出两组十进制数。

	A	B
a.	3.264×10^3	6.52×10^2
b.	-2.27734375×10^0	1.154375×10^2

3.12.4 [60] <3.5>手算 A 除以 B。给出必要的步骤。假设有保护位、舍入位和粘贴位，并在必要时使用。以 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。

Livermore Loops 是从劳伦斯·利弗莫尔实验室的科学计算程序中摘出的浮点密集型核心组件。下表给出单独的核心程序。你可以在 http://www.netlib.org/benchmark/livermore 找到它们。

a.	Livermore Loop 1
b.	Livermore Loop 7

3.12.5 [60] <3.5>用 MIPS 汇编程序写出循环。

3.12.6 [60] <3.5>详细地描述用数字计算机执行浮点除的技术。一定要给出你所使用信息来源的参考文献。

习题 3.13

定点整数的操作正如人所希望的那样——满足交换律、结合律和分配律。但是在处理浮点数时却并不总是满足。首先看一下结合律。下表给出几对十进制数。

	A	B	C
a.	-1.6360×10^{4}	1.6360×10^{4}	1.0×10^{0}
b.	2.865625×10^{1}	4.140625×10^{-1}	1.2140625×10^{1}

3.13.1 [20] <3.2, 3.5, 3.6>手算 (A + B) + C，设 A、B、C 都以习题 3.11.2 中提到的 16 位 NVIDIA 格式存储（书中也有介绍）。假设有一位保护位、一位舍入位和一位粘贴位，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。

3.13.2 [20] <3.2, 3.5, 3.6>手算 A + (B + C)，设 A、B、C 都以习题 3.11.2 中提到的 16 位 NVIDIA 格式存储（课文中也有介绍）。假设有一位保护位、一位舍入位和一位粘贴位，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。

3.13.3 [10] <3.2, 3.5, 3.6>基于习题 3.13.1 和习题 3.13.2 中的答案，看看是否 (A + B) + C = A + (B + C)？

下表再给出几对十进制数。

	A	B	C
a.	4.8828125×10^{-4}	1.768×10^{3}	2.50125×10^{2}
b.	4.721875×10^{1}	2.809375×10^{1}	3.575×10^{1}

3.13.4 [30] <3.3, 3.5, 3.6>手算 (A × B) × C，设 A、B、C 都以习题 3.11.2 中提到的 16 位 NVIDIA 格式存储（书中也有介绍）。假设有一位保护位、一位舍入位和一位粘贴位，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。

3.13.5 [30] <3.3, 3.5, 3.6>手算 A × (B × C)，设 A、B、C 都以习题 3.11.2 中提到的 16 位 NVIDIA 格式存储（书中也有介绍）。假设有一位保护位、一位舍入位和一位粘贴位，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。

3.13.6 [10] <3.3, 3.5, 3.6>基于习题 3.13.4 和习题 3.13.5 中的答案，看看是否 (A × B) × C = A × (B × C)？

习题 3.14

处理浮点数时并不只是交换律得不到满足。还有其他古怪的事情发生。下表给出几对十进制数。

	A	B	C
a.	1.5234375×10^{-1}	2.0703125×10^{-1}	9.96875×10^{1}
b.	-2.7890625×10^{1}	-8.088×10^{3}	1.0216×10^{4}

3.14.1 [30] <3.2, 3.3, 3.5, 3.6>手算 A × (B + C)，设 A、B、C 都以习题 3.11.2 中提到的 16 位 NVIDIA 格式存储（书中也有介绍）。假设有一位保护位、一位舍入位和一位粘贴位，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。

3.14.2 [30] <3.2, 3.3, 3.5, 3.6>手算 (A × B) + (A × C)，设 A、B、C 都以习题 3.11.2 中提到的 16 位 NVIDIA 格式存储（书中也有介绍）。假设有一位保护位、一位舍入位和一位粘贴位，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。

3.14.3 [10] <3.2, 3.3, 3.5, 3.6>基于习题 3.14.1 和习题 3.14.2 中的答案，看看是否 (A × B) + (A × C) = A × (B + C)？

下表给出两组数，每组包含一个分数和一个整数。

	A	B
a.	1/3	3
b.	−1/7	7

3.14.4 [10] <3.5>按照 IEEE 754 浮点格式，写出 A 的位模式。你能精确表示 A 吗？

3.14.5 [10] <3.2，3.3，3.5，3.6>如果将 A 自加 B 次得到多少？A × B 是多少？它们相同吗？它们应该是多少？

3.14.6 [60] <3.2，3.3，3.4，3.5，3.6>如果将 B 开方根，再加上自身，得到多少？应该得到多少？分别用单精度和双精度浮点数做。（写一个程序来做这些计算。）

习题 3.15

在尾数域中使用二进制编码，但并非必须如此。例如，IBM 在他们的浮点中使用了基数为 16 的数字格式。也有其他一些可能的方法，每种都有自己的优点和缺点。下表给出了两个分数，用各种浮点格式来表达。

a.	1/2
b.	1/9

3.15.1 [10] <3.5，3.6>写出尾数的位模式，其浮点格式采用二进制编码的尾数（就是本章用到的）。假设有 24 位，并且不需要进行规格化。这种表达精确吗？

3.15.2 [10] <3.5，3.6>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？

3.15.3 [10] <3.5，3.6>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 ~ E。）假设有 24 位，并且不需要进行规格化。这种表达精确吗？

3.15.4 [10] <3.5，3.6>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。）假设有 20 位，并且不需要进行规格化。这种表达精确吗？你能看出这种方法有什么优点吗？

小测验答案

3.2 节　C。

3.5 节　C。

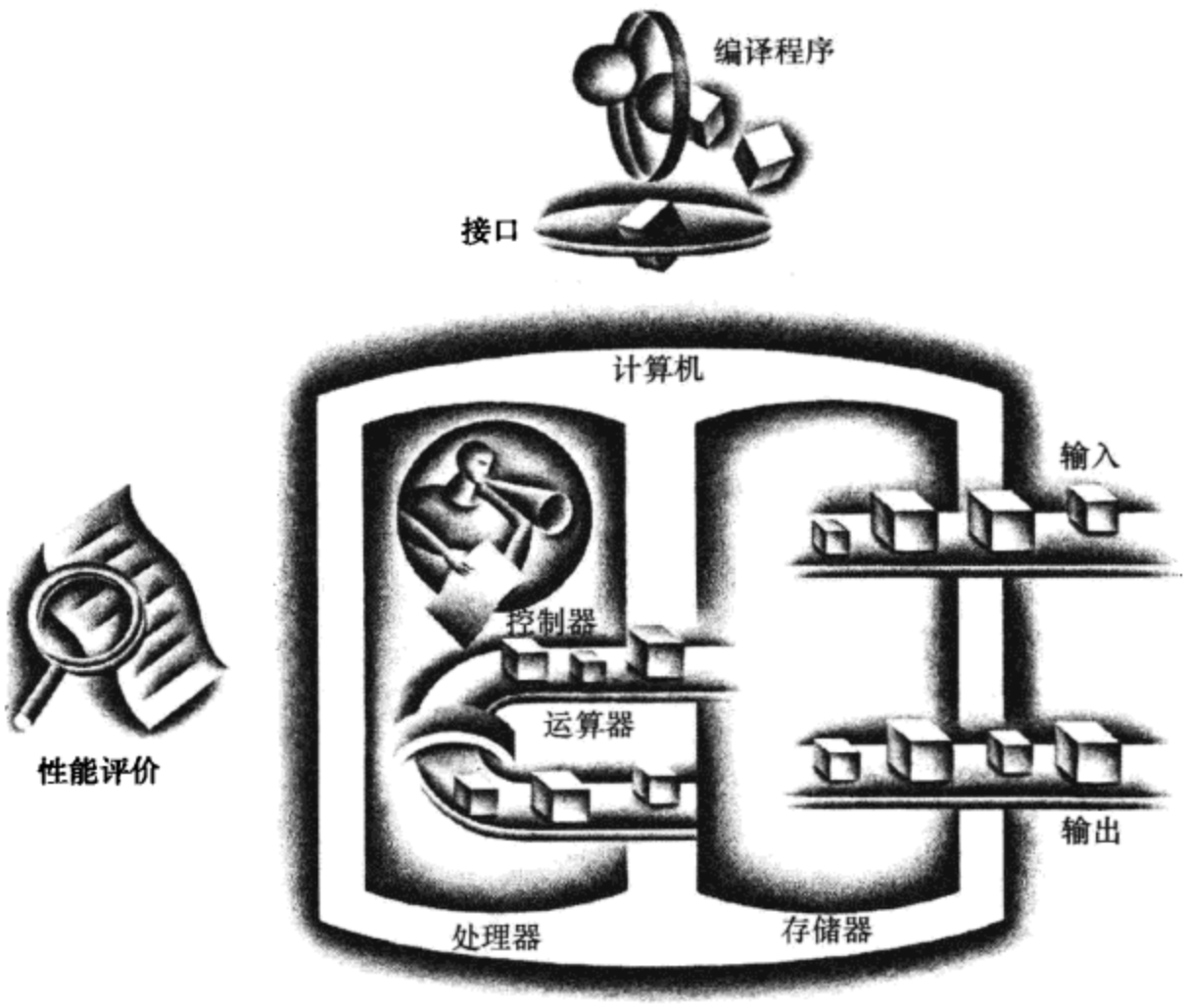
第4章

Computer Organization and Design: The Hardware/Software Interface, 4E

处 理 器

在关键问题上，没有什么细节是小事。

——法国谚语



计算机的五大经典部件

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知识。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，4.1节和4.5节给出了流水线的基本解释。4.10节介绍了最近的趋势。4.11节列举了最新的AMD Opteron X4（Barcelona）微处理器作例子。这几节提供了在高层次理解流水线概念的必要背景知识。

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 CD 中的 4.12 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。

4.1.1 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）和存储字（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。

这个子集没有包含所有的整数指令（如不包含乘、除指令和移位指令等），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通路时的关键原理，并可以在此基础之上实现其他指令。

在学习此实现方式时，我们将能够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 2 章 2.2 节和 2.3 节介绍的许多关键设计原理，如“加速执行常用操作”和“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。

4.1.2 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。算术逻辑指令或装载指令将来自 ALU 或存储器的数据写入寄存器。对分支指令，我们需要基于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。

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。光盘中的附录 C 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。

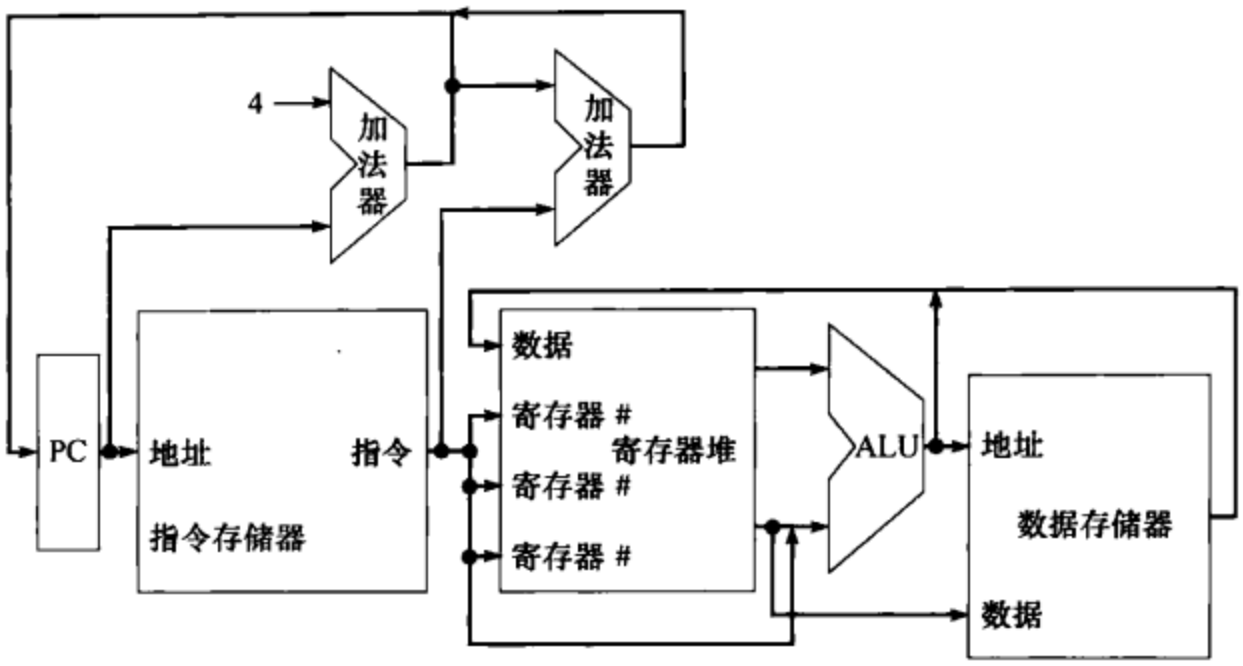


图 4-1 一个 MIPS 子集实现的抽象视图，描述了主要功能单元及其连接

所有指令都开始于使用程序计数器获得的指令存储器中指令的地址。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（对于整数算术逻辑类指令），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（在其中 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。

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，装载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，就像我们在第 2 章中提到的那样。（光盘中的附录 C 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，该多选器根据用来执行 beq 指令比较的 ALU 的 Zero 输出设置。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。

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哪几个？

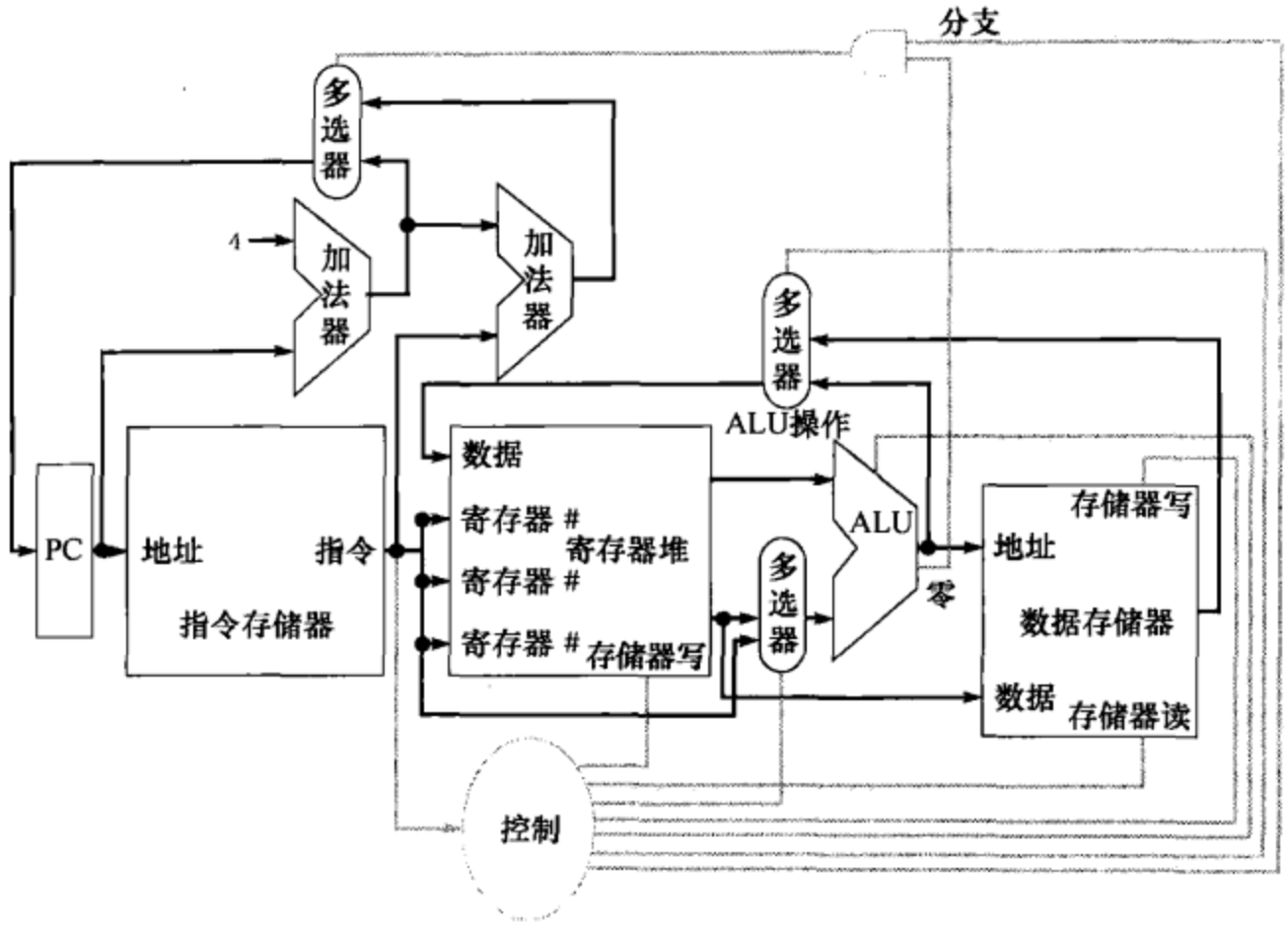


图 4-2 一个 MIPS 子集的基本实现，其中包含了必要的多选器和控制信号

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（装载指令时）。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、数据存储器读写和寄存器堆写入等。控制信号在图中用灰色线标识出来。

4.2 逻辑设计惯例

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，看一看光盘中的附录 C 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 中并在光盘的附录 C 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。

设计中的其他单元不是组合的，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。

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（参见光盘中的附录 C），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，

⊖ 组合单元（combinational element）：一个操作单元，如与门或 ALU。
⊖ 状态单元（state element）：一个存储单元，如寄存器或存储器。

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光盘的附录 C 中有详细论述。

我们将使用术语**有效**㊀（asserted）。表示信号为逻辑高，**无效**㊁（deasserted）表示信号为逻辑低。

时钟方法

时钟方法㊂（*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是很重要的，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，计算机的设计中不能允许这样的不确定性。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**㊃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。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。

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**㊄。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。写控制信号和时钟都是输入信号，只有当写控制信号有效并且时钟边沿到来时，状态单元才改变状态。

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并使之经过一些组合逻辑，同时将别的值写入该寄存器。选择在时钟的上升沿还是下降沿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。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。在光盘的附录 C 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。

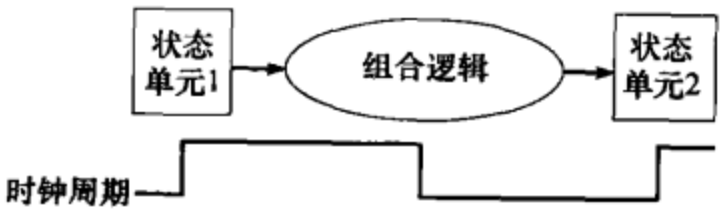


图 4-3 组合逻辑、状态单元和时钟周期的关系

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）都是边沿触发的。

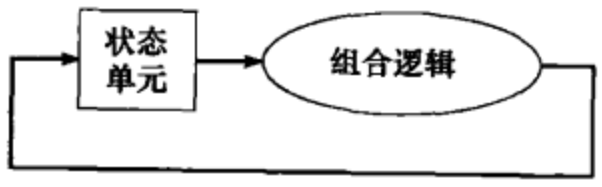


图 4-4 一种边沿触发方式，支持状态单元在同一个时钟周期内同时读写而不会因竞争而出现中间数据

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。

㊀ 有效（asserted）：信号为逻辑高或真。
㊁ 无效（deasserted）：信号为逻辑低或假。
㊂ 时钟方法（clocking methodology）：用来确定数据相对于时钟何时稳定和有效的方法。
㊃ 边沿触发的时钟（edge-triggered clocking）：一种所有的状态改变发生于时钟沿的时钟机制。
㊄ 控制信号（control signal）：用来决定多选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。

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。

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。另外，我们之所以要使用术语“有效”和“无效”，是因为数字1有时表示逻辑高，有时表示逻辑低。

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**[⊖]。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。

图4-5a展示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（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增加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C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，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。

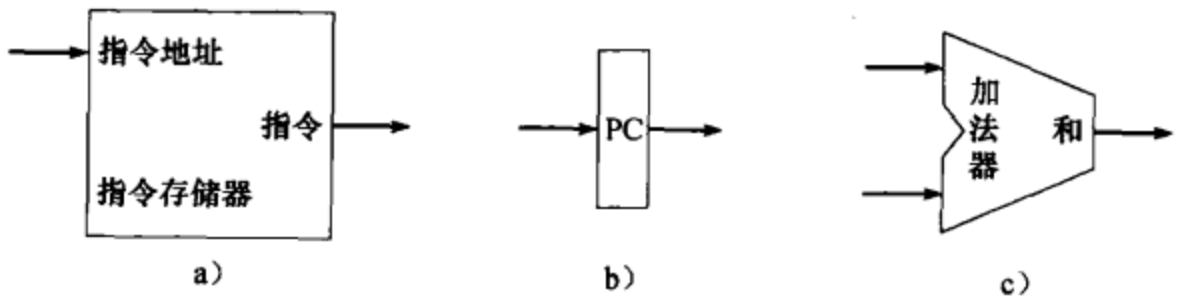


图4-5 存取指令需要的两个状态单元，以及计算下一条指令地址所需要的加法器

a）指令存储器；b）程序计数器；c）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。（在装载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。

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个字节。此时的数据通路如图4-6所示，使用了图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。

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3章介绍的`add`、`sub`、`AND`、`OR`和`slt`指令。回忆一下，此类指令的典型

⊖ 数据通路部件（datapath element）：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。

形式是 add $t1，$t2，$t3，它将读取 $t2 和 $t3，并将结果写回 $t1。

处理器的 32 个通用寄存器位于一个叫做**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。

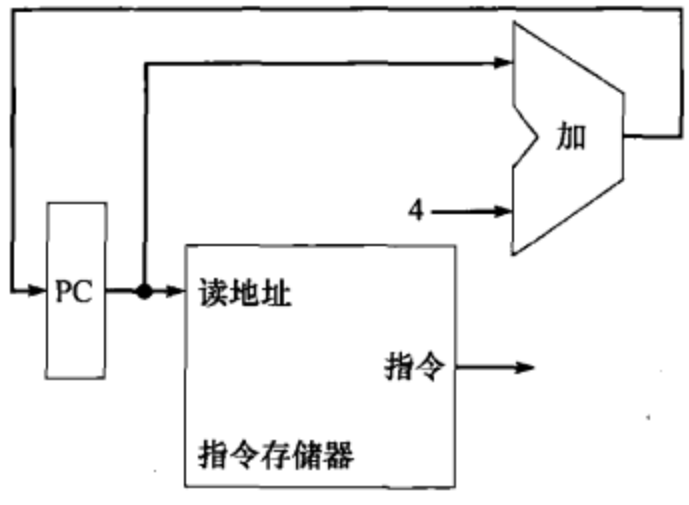


图 4-6 用于取指和程序计数器自增的数据通路部分

取出的指令被数据通路的其他部分所使用。

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 位控制信号在光盘的附录 C 中有详细的描述。

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；如果是装载指令，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。

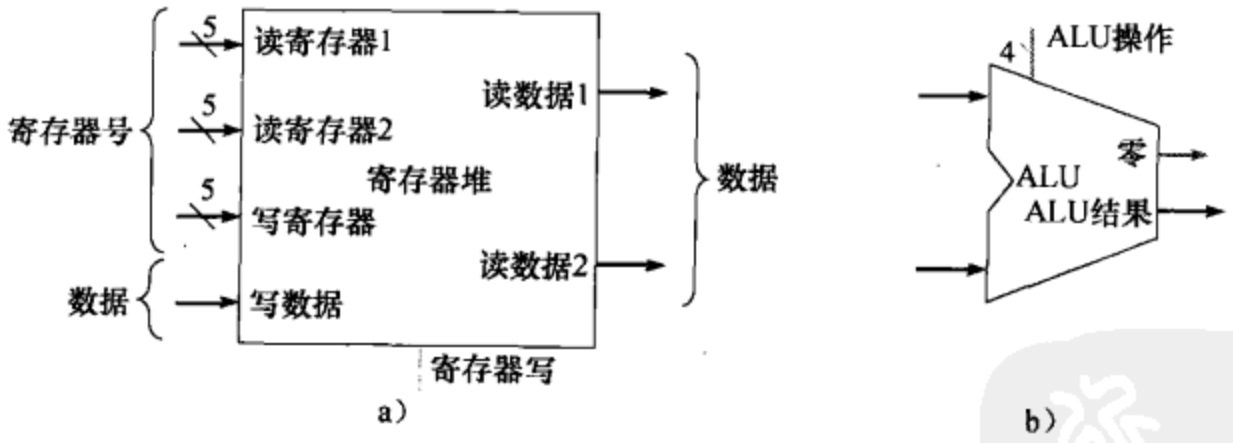


图 4-7 实现 R 型指令的 ALU 操作所需的两个单元——寄存器堆和 ALU

a）寄存器推；b）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 C（见光盘）的 C.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 C（见光盘）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。

⊖ 寄存器堆（register file）：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。

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。图4-8中给出了这两个单元。

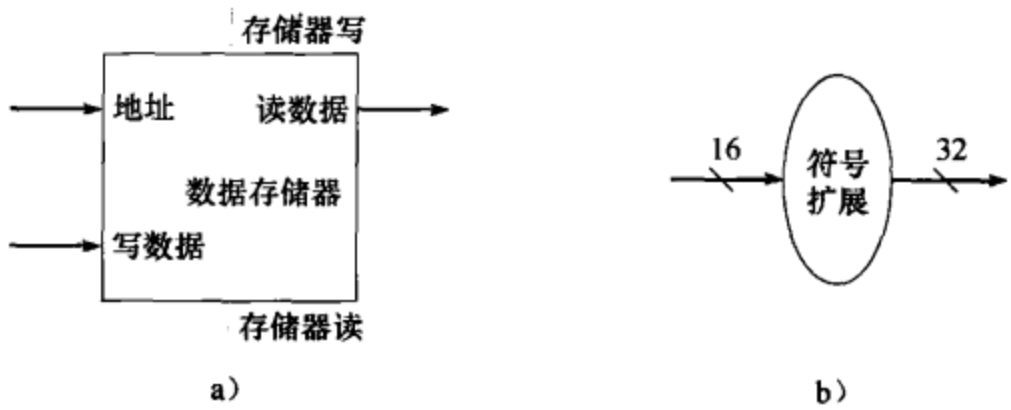


图4-8 除了图4-7中的寄存器堆和ALU，存储指令和装载指令还需要两个单元——数据存储器单元和符号扩展单元

a）数据存储器单元；b）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，见光盘中附录C的C.8节。

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**[⊜]。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**[⊝]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**[⊠]。

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，分支指令还需要改变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，分支目标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

⊖ 符号扩展（sign-extend）：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。

⊜ 分支目标地址（branch target address）：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（branch taken）：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（branch not taken）：分支条件不满足而PC变为分支指令的下一条指令地址。

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，比较可由在光盘中的附录C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。

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）。

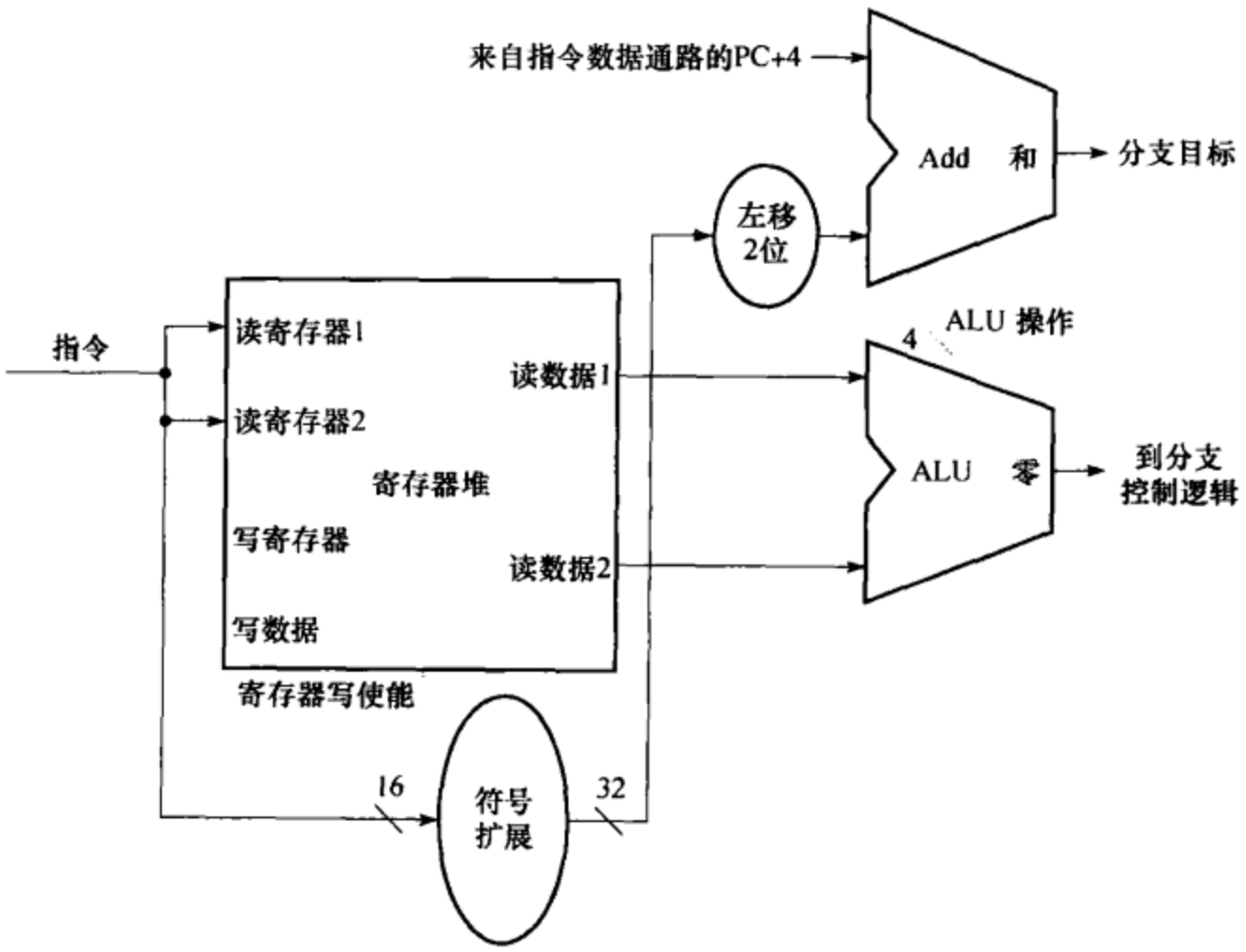


图4-9 在分支指令的数据通路中，用ALU计算分支条件是否成立，用另外的加法器将自增后的PC值与符号扩展后左移两位的指令低16位（分支偏移量）相加，得到分支目标地址

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。

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。

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每个时钟周期执行一条指令。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

⊖ 延迟的分支（delayed branch）：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。

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。

举例 **建立一个数据通路**

算术逻辑指令（或 R 型指令）的数据通路与存取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（对装载操作而言）。

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。

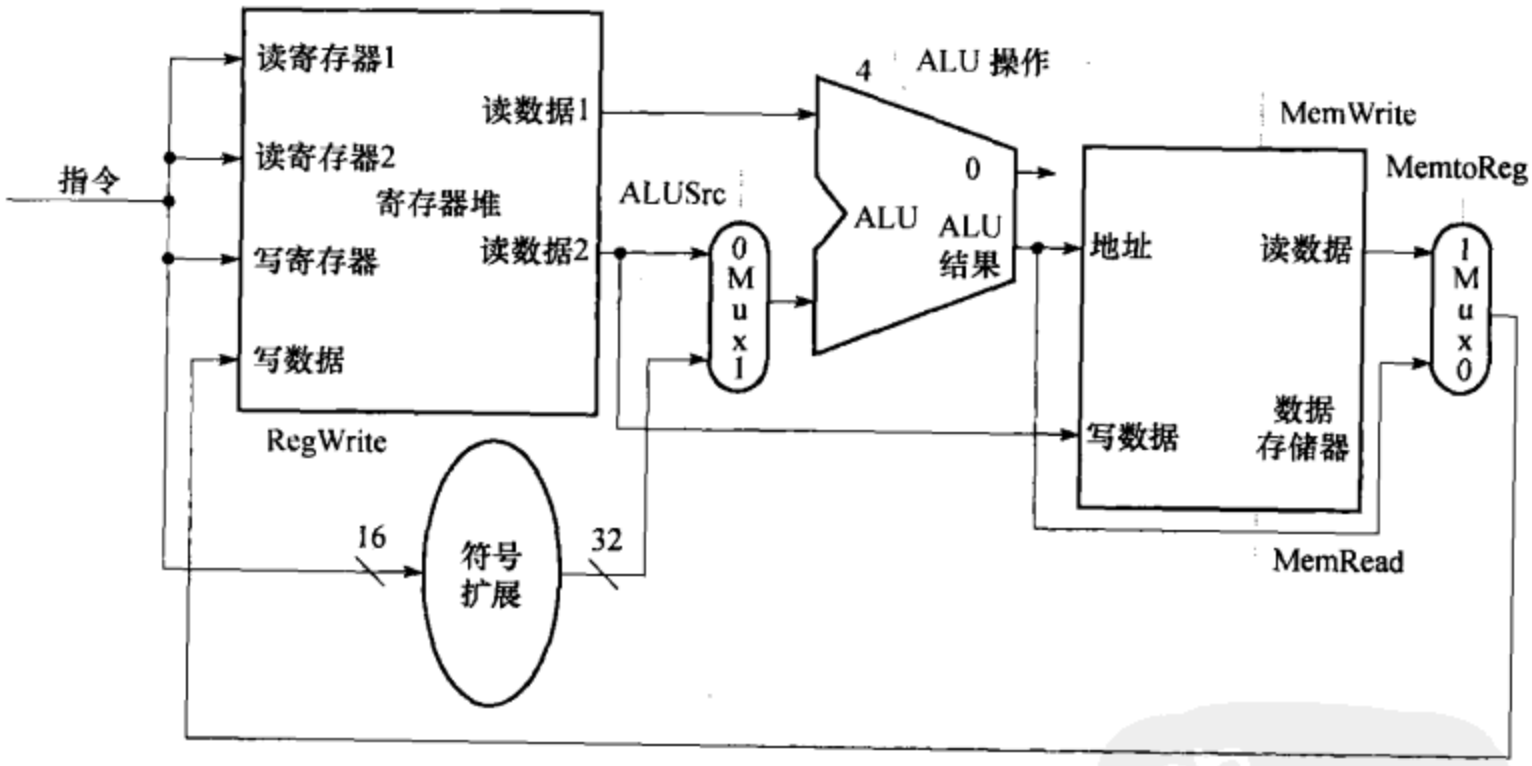


图 4-10 存储指令和 R 型指令数据通路的合并

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 型指令和存储指令数据通路，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。

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。

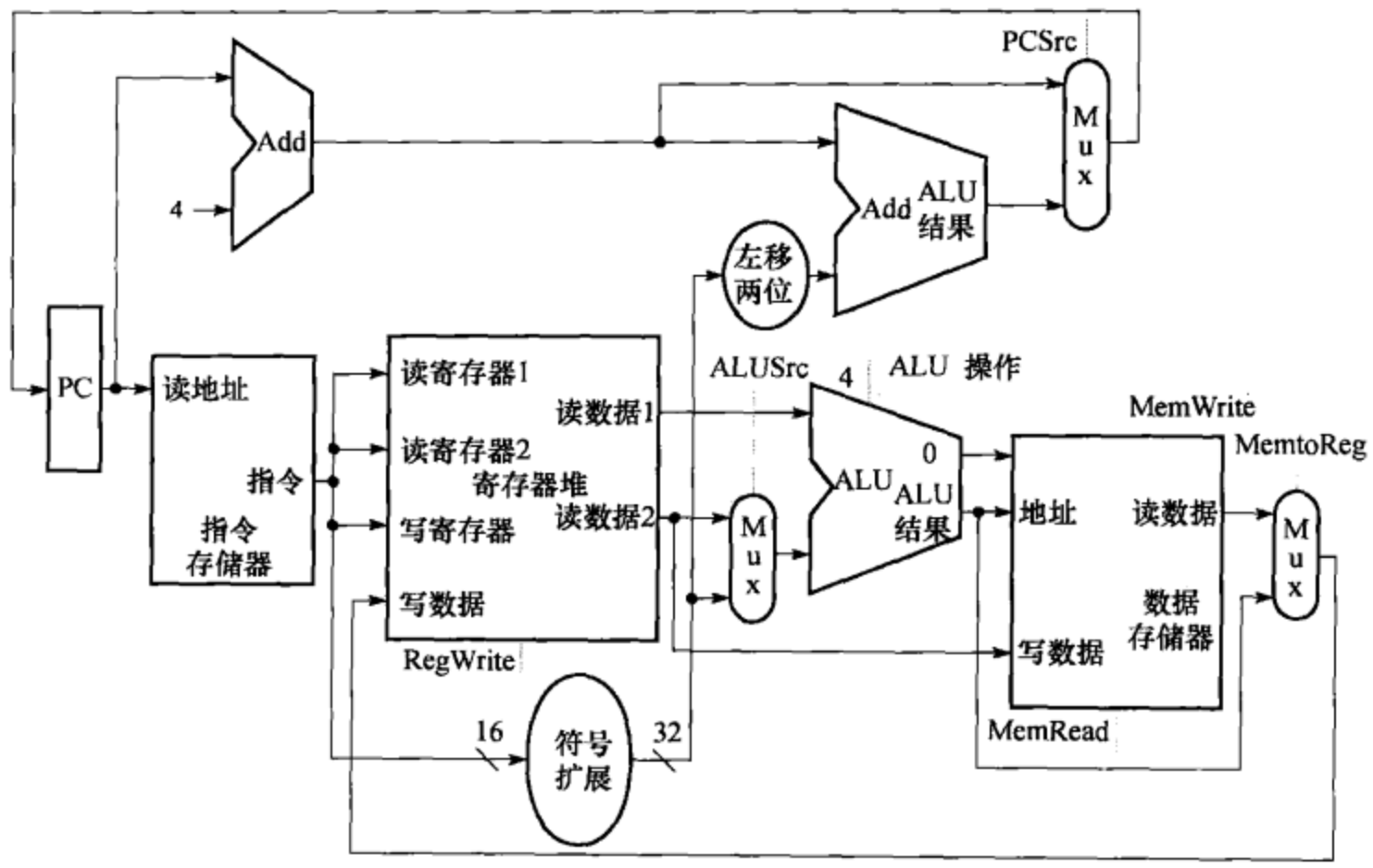


图 4-11 将不同类型指令所需的功能单元合并在一起实现的一个简单 MIPS 系统结构数据通路

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。

小测验

1）对装载指令来说，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. 对装载指令而言，MemtoReg 信号线的设置无关紧要。

2）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）、存储字（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）和小于则设置（set on Less than），后面我们还将实现跳转指令（j）。

4.4.1 ALU 控制

光盘附录 C 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 1
0010	加	1100	或非

根据指令类型的不同，ALU 将执行上述五种功能中的一种。（或非操作在我们目前实现的 MIPS 子集中暂时无用。）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 执行五种操作中的一种（与、或、减、加、小于则置 1）。对相等则分支指令，ALU 执行减法操作。

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 字段决定（10）。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。

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
存储字	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 1	101010	小于则置 1	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。所有的编码以二进制给出。注意，当 ALUOp 为 00 或 01 时，输出字段不依赖于 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。详情见光盘附录 C。

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减少时钟周期非常关键。

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。

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），我们并不是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（这样做的缺点在光盘附录 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**[⊜]。真值表中的无关项（在输入列表中用 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 C

⊖ 真值表（truth table）：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（don't-care term）：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。

（见光盘）中有更多的讨论。

ALUOp		funct 字段						操作
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	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。

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（见光盘）中的 D.2 节讨论。

4.4.2　主控制单元的设计

我们已经描述了如何使用 funct 和 ALUOp 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。

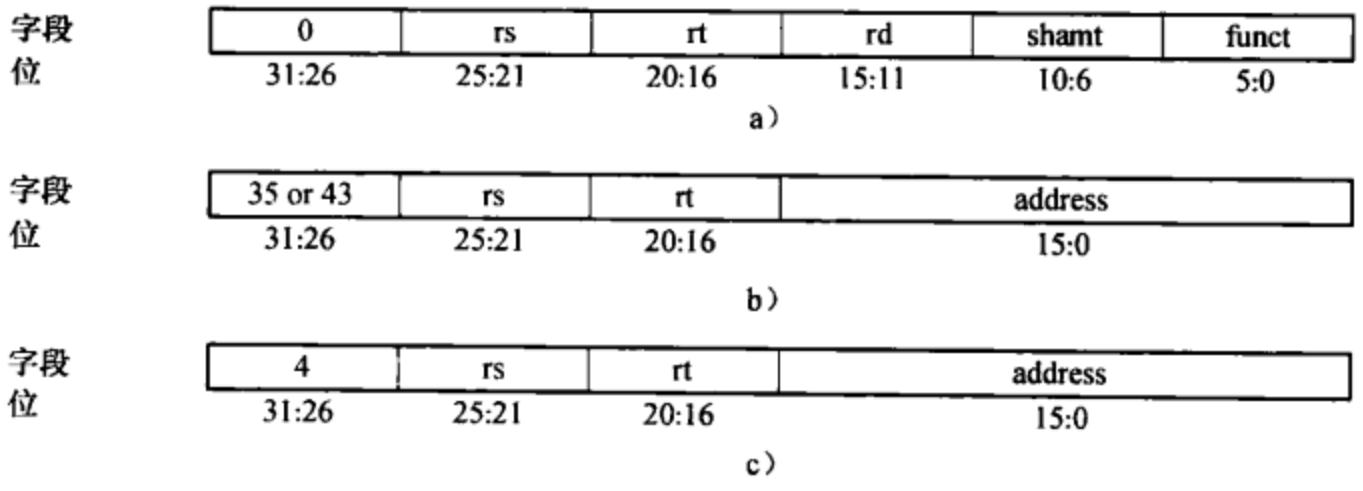


图 4-14　三种指令类型（R 型、存取和分支）使用的两种指令格式

a）R 型指令；b）存取指令；c）分支指令

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）装载指令（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。对装载指令，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 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，亦称**操作码**[⊖]，总是为 31:26 位。我们将用 Op［5:0］来表示。

⊖ 操作码（opcode）：指示指令操作和格式的字段。

- 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。
- 存取指令的基址字段在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。对装载指令为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。

从第2章得到的第一个设计原则——简单导致规整——在这里就体现出来了。

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），如图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写信号。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信线。

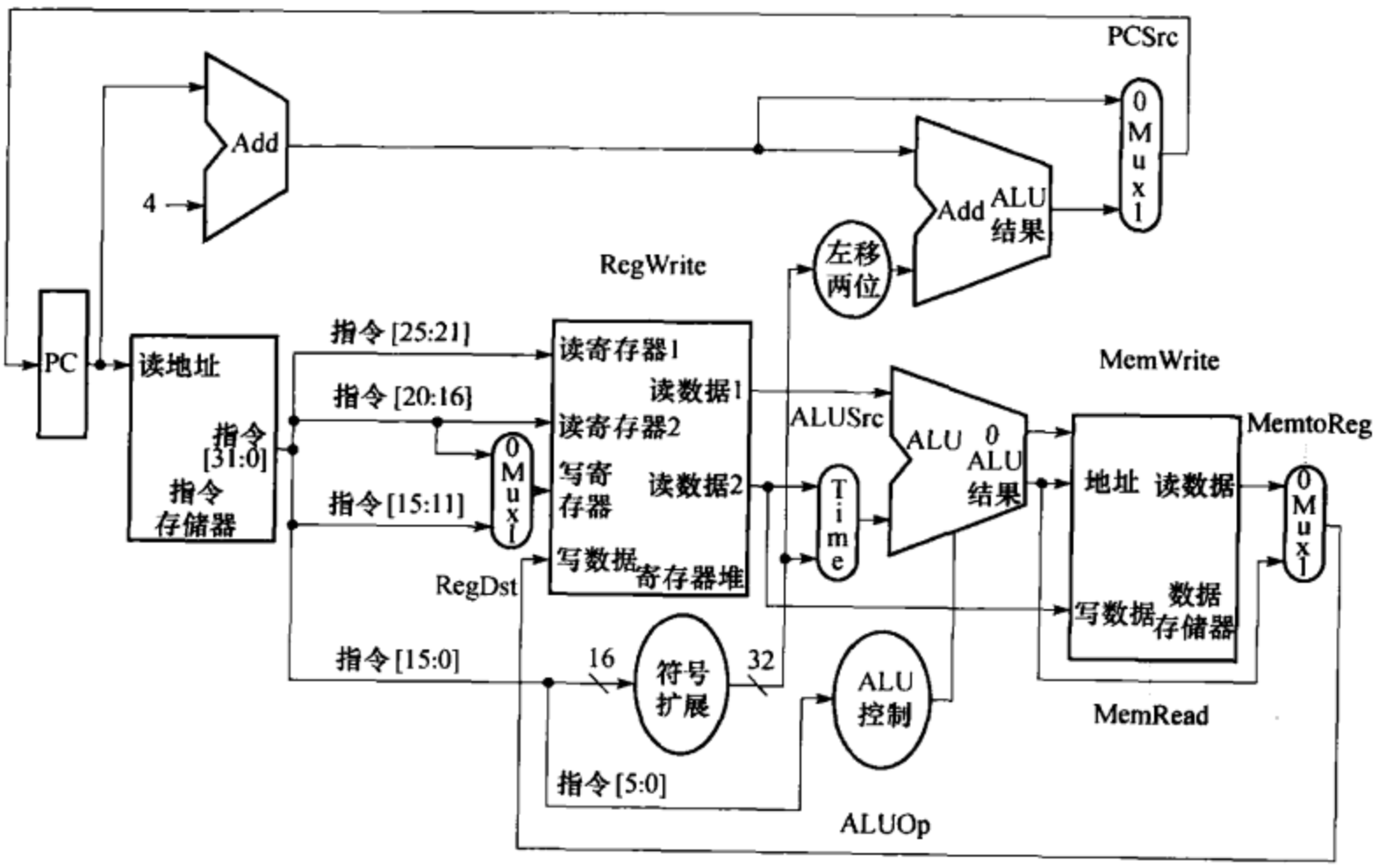


图4-15 在图4-12的数据通路上增加了所有必需的多选器并标识出了所有的控制信号

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。

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，最好非正式地定义一下其他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。

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”。

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。

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 位的符号扩展
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	数据存储器写使能有效
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

当两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——时钟信号，且用于写控制中。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（光盘的附录 C 中对此问题有进一步的讨论。）

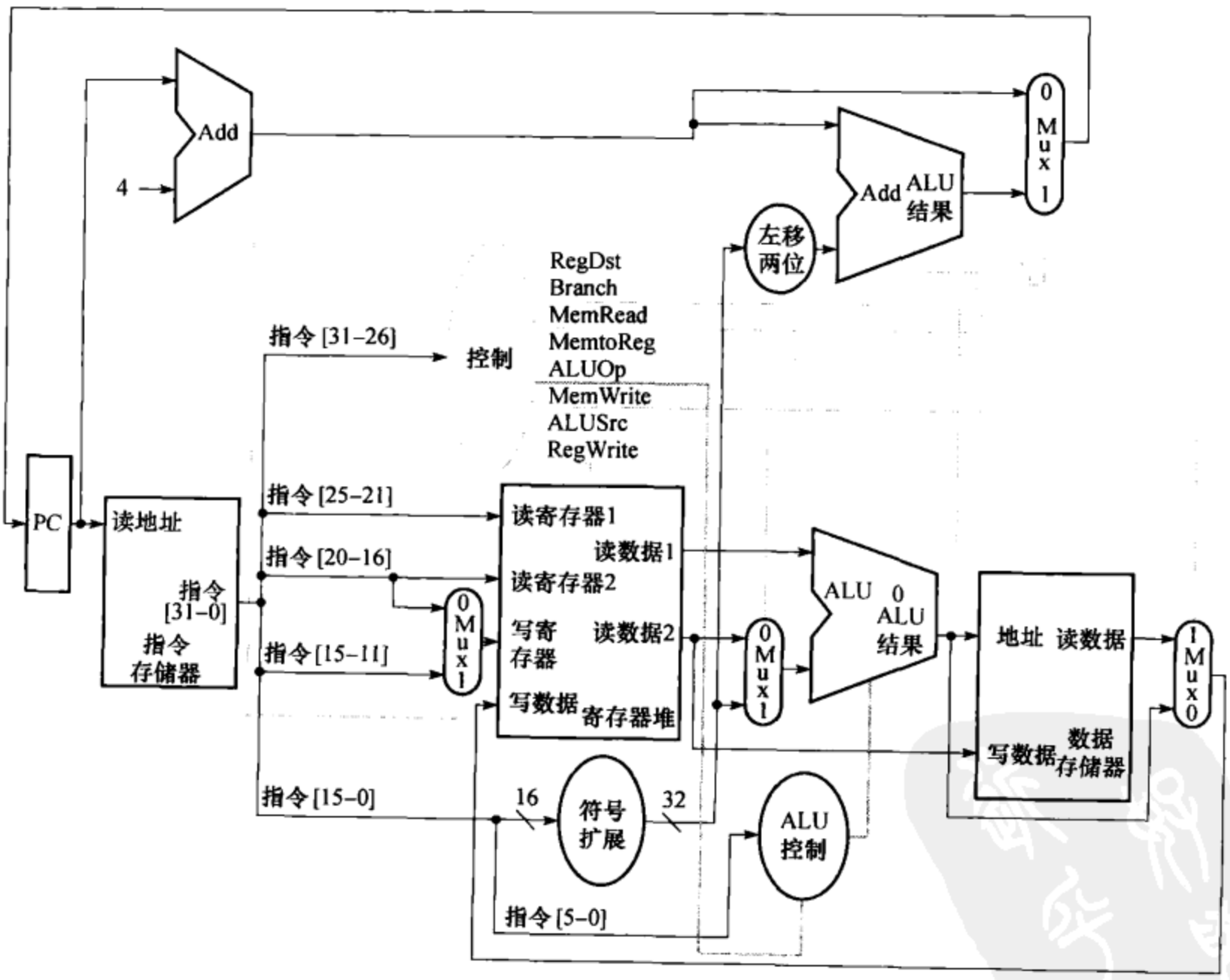


图 4-17　包含控制单元的简单数据通路

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信号（Branch），和一个 ALU 的 2 位控制信号（ALUOp）。分支控制信号 Branch 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。

在设计控制单元之前，这里先非正式地定义一下控制功能。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。

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。

4.4.3 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 4-19 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）。

我们可以用和图 4-19 类似的方式描述装载指令（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。可以考虑将装载指令的执行分为 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。

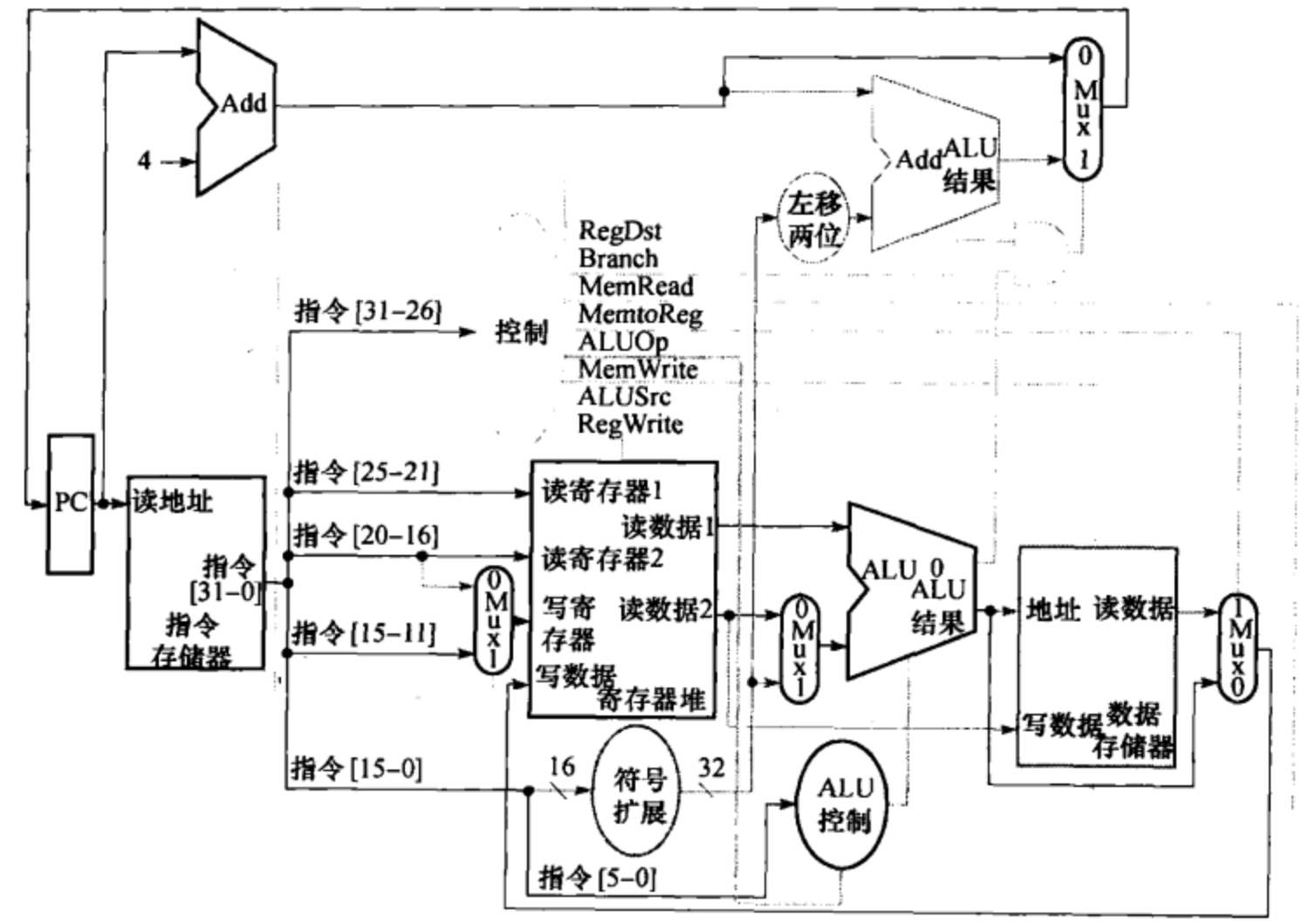


图 4-19　执行 R 型指令（如 add $t1，$t2，$t3）时数据通路的操作

操作中用到的控制信号、功能单元和连接均用灰色线显示。

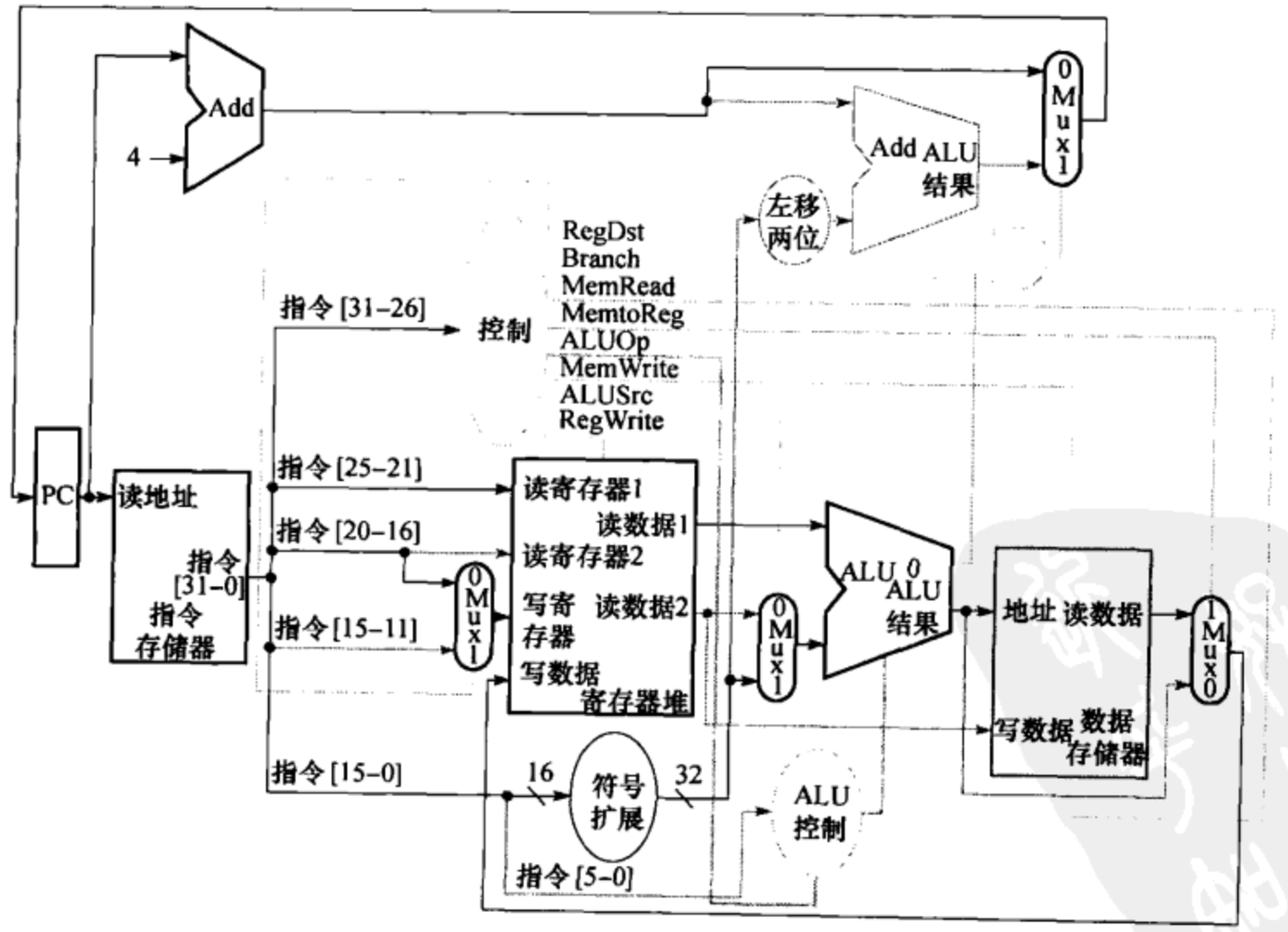


图 4-20　执行装载指令时数据通路的操作

操作中用到的控制信号、功能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。

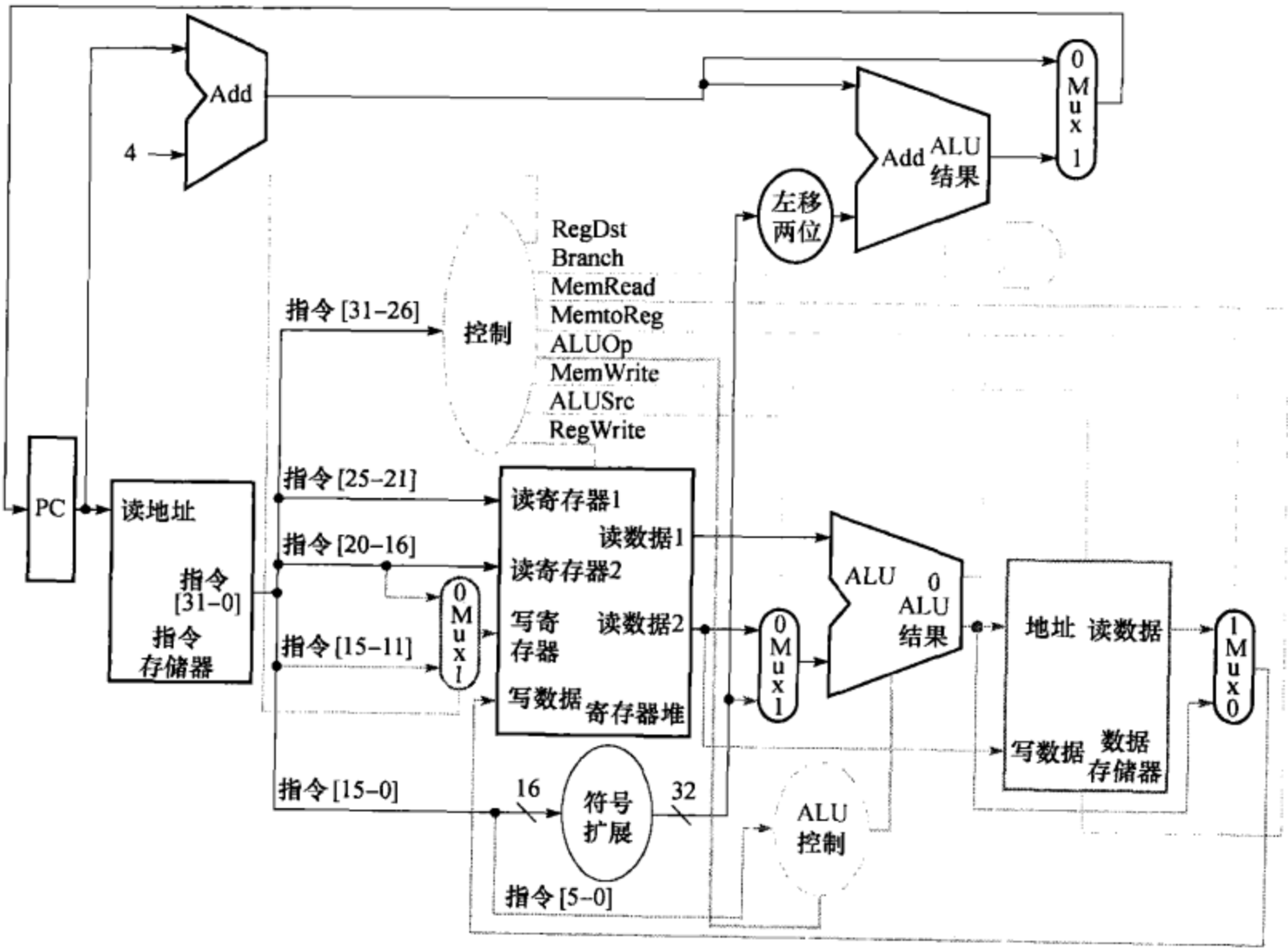


图4-21 执行相等则分支指令时数据通路的操作

在用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。

4.4.4 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（见光盘）的D.2节对此进行了描述。

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**[⊖]，在此基础之上我

⊖ 单周期实现（single-cycle implementation）：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。

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。

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
输出	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0P	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 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]）的四种组合。表的下半部分为四种组合的输出。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四个操作码，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$\overline{Op5} \cdot \overline{Op2}$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。

举例 **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。

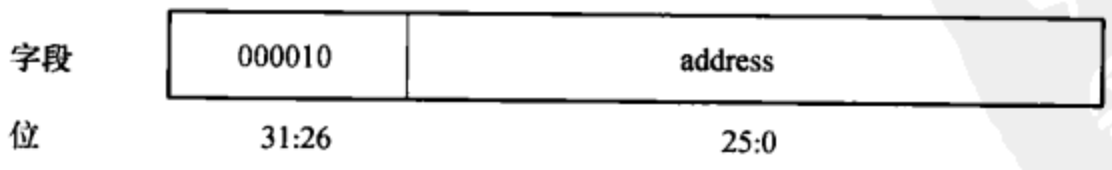


图 4-23 跳转指令的格式（操作码 =2）

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 位地址连结，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。

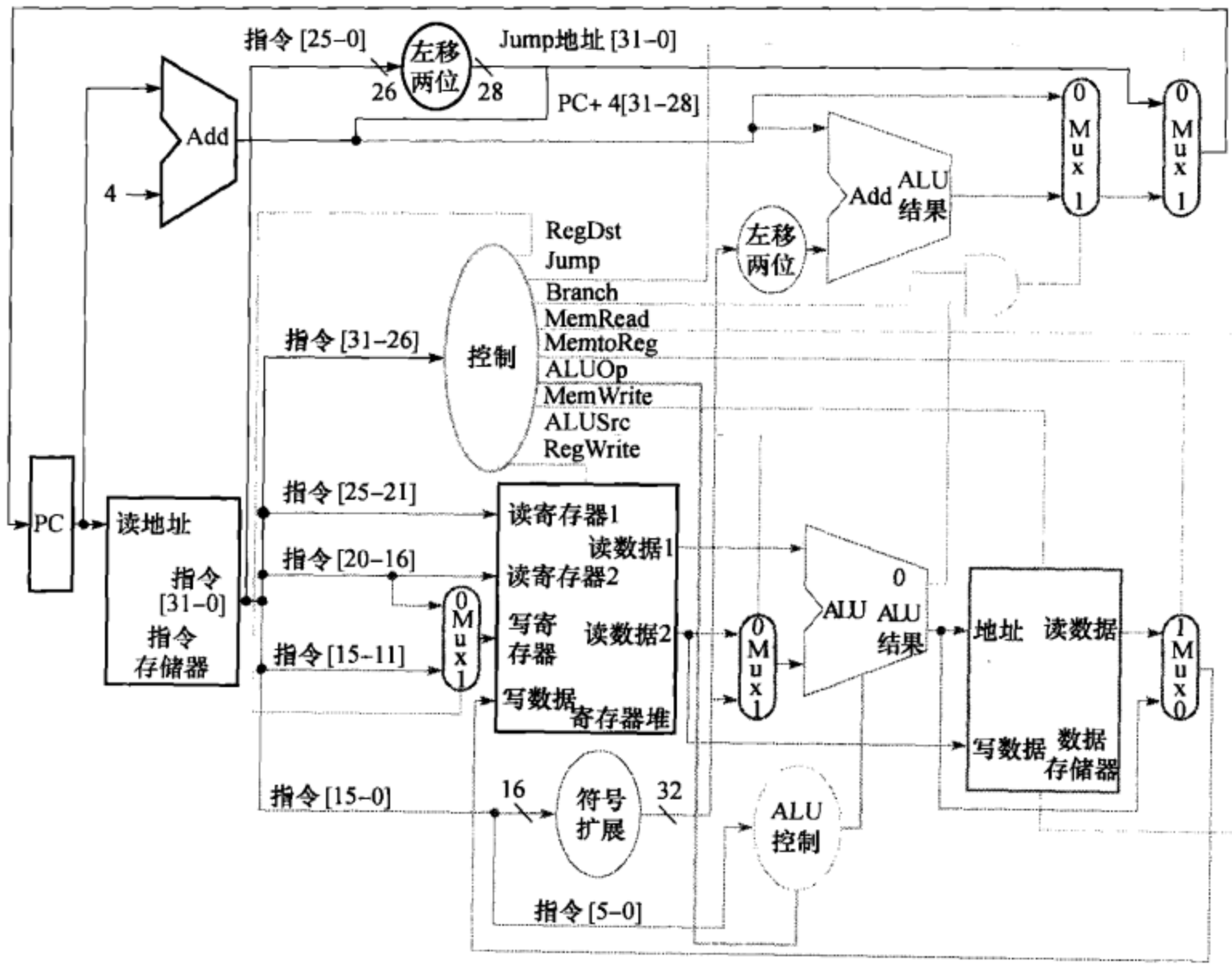


图 4-24 扩展后能处理跳转指令的简单控制和数据通路

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。

4.4.5 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。这条指令几乎肯定是装载指令，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。

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 2 章中加快常见情况这一设计原则。

在 4.5 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，流水线技术的使用是普遍的。

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概的了解，可以集中精力看完本节，然后直接跳到4.10节和4.11节学习在最近的处理器（AMD Opteron X4、Intel Core）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，将之从烘干机中取出，然后将衣服放在桌子上叠起来。

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

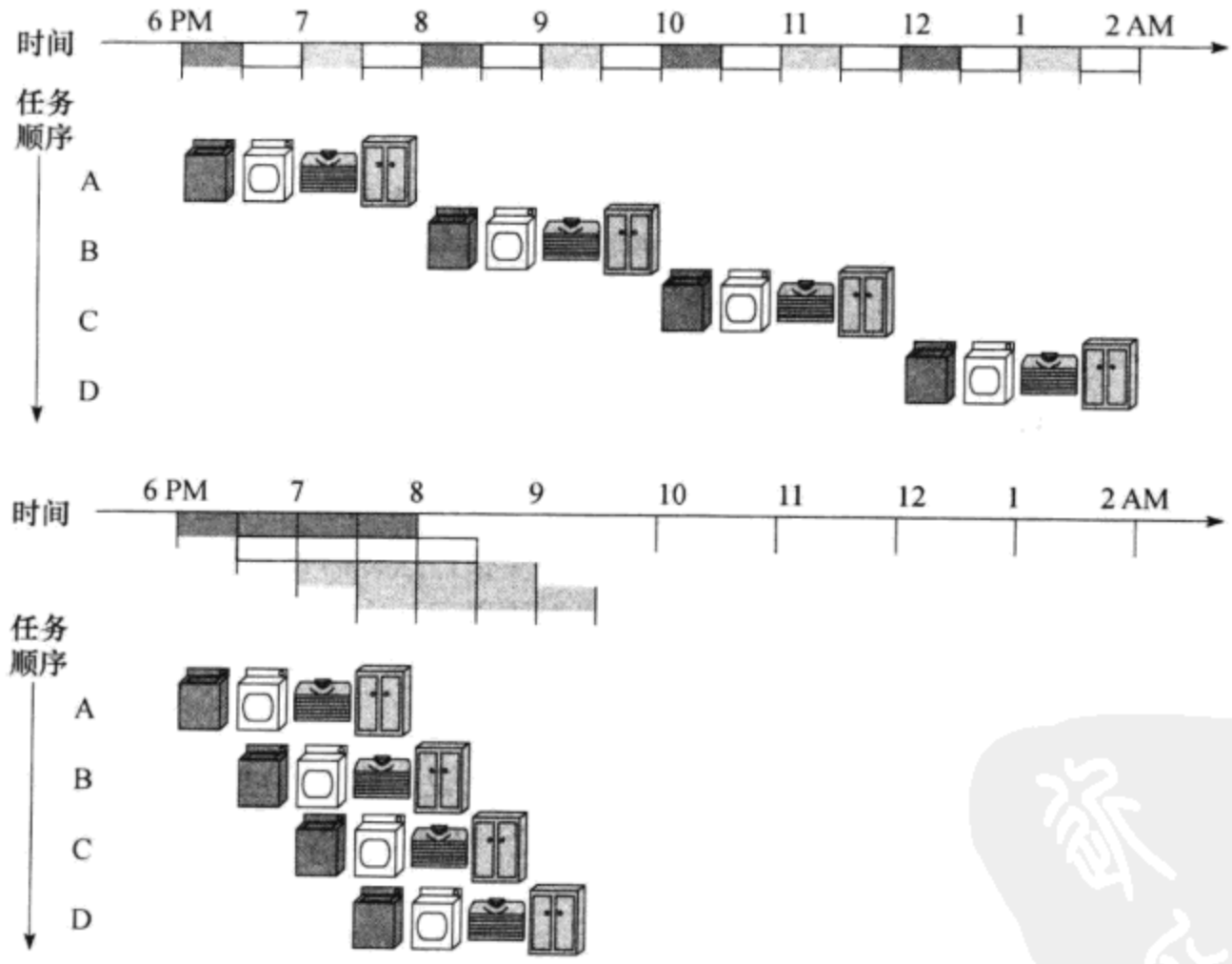


图4-25 以洗衣店为例类比流水线的动作过程

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”每个都需要三十分钟来完成各自的任务。顺序的洗涤方法将花费8个小时的时间洗完四批衣服，而流水线的洗涤方法只需要花费3.5小时。

⊖ 流水线（pipelining）：一种实现多条指令重叠执行的技术，与生产流水线类似。

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三批衣服，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：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四批衣服的处理过程。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是很大时，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指令包含如下五个处理步骤：

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。

举例 **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）、存储字（sw）、加（add）、减（sub）、与（AND）、或（OR）、小于则置1（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：200 ps；ALU操作：200 ps；寄存器堆的读写：100 ps。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。

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800 ps。与图4-25类似，图4-27比较了三条装载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 × 800 ps = 2400 ps。

指令类型	指令预取	读寄存器	ALU 操作	数据存取	写寄存器	总时间
取字（lw）	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
存储字（sw）	200 ps	100 ps	200 ps	200 ps		700 ps
R 型（add、sub、AND、OR、slt）	200 ps	100 ps	200 ps		100 ps	600 ps
分支（beq）	200 ps	100 ps	200 ps			500 ps

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。

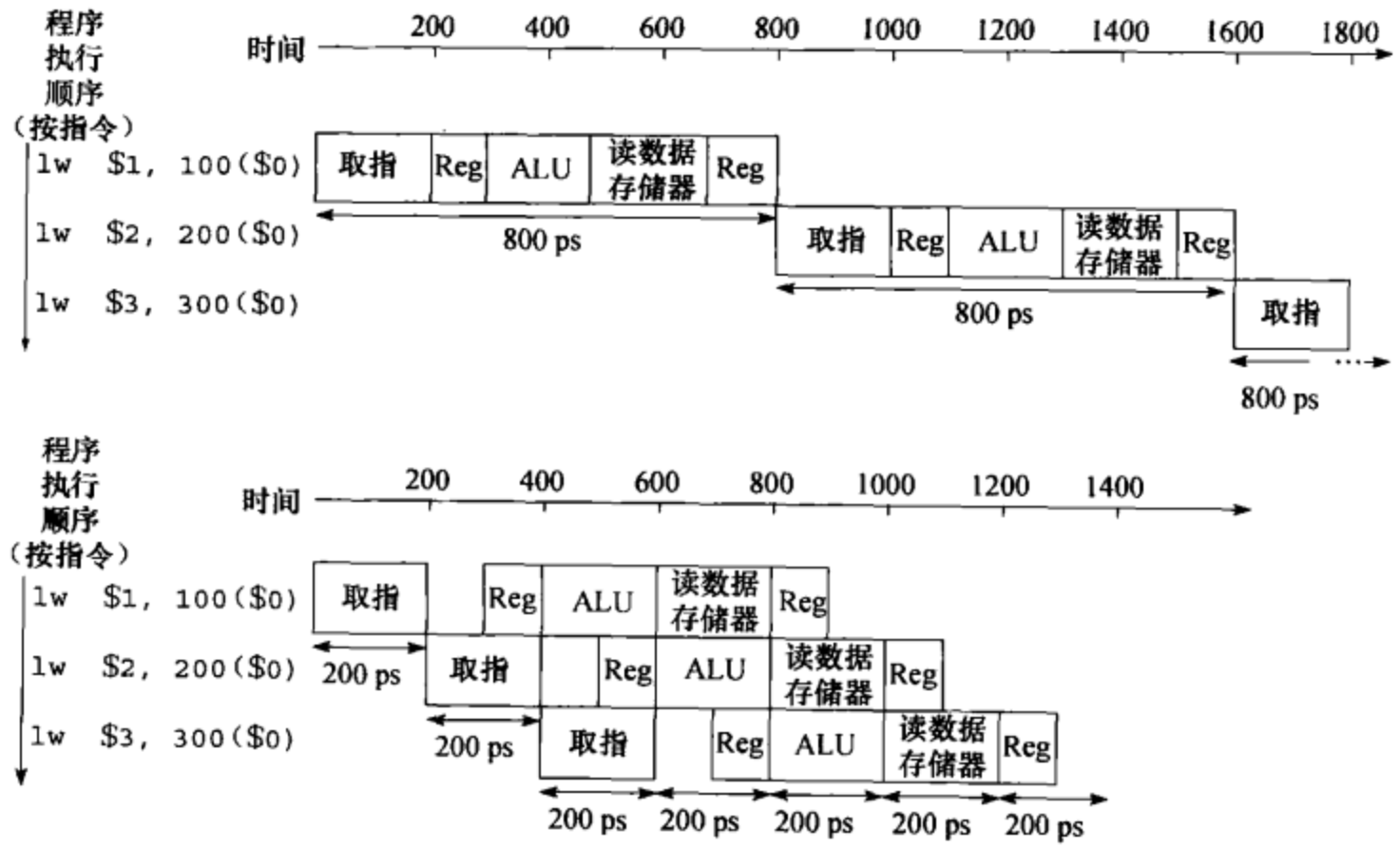


图 4-27 单周期、非流水线的指令执行过程（上图）与流水线的指令执行过程（下图）

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 800 ps 降到了 200 ps。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 500 ps，但它必须选择在最坏情况下的 800 ps 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 200 ps 而不是有些步骤可以达到的 100 ps。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 × 200 ps = 600 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）

指令执行时间（流水线）= 指令执行时间（非流水线）/流水线级数

即在理想情况和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 800 ps 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 160 ps 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2400 ps/1400 ps，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200 ps，因此，整个的执行时间就变成1 000 000 × 200 ps + 1400 ps，即200 001 400 ps。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800 ps，因此整个的执行时间为1 000 000 × 800 ps + 2400 ps，即800 002 400 ps。在这种理想的条件下，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

$$800\,002\,400\ \text{ps}/200\,001\,400\ \text{ps} \approx 800\ \text{ps}/200\ \text{ps} \approx 4.00$$

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17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3级与第四级将会扩展为地址计算、存储访问和执行3级。

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

第一种冒险叫做**结构冒险**[⊖]。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。

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

⊖ 结构冒险（structural hazard）：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。

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[⊖]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搁置一边。

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）。举个例子来说，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观察：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**转发**[⊖]或者**旁路**。

举例 **两条指令间的转发**

对于上述的两条指令，说明如何使用转发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。

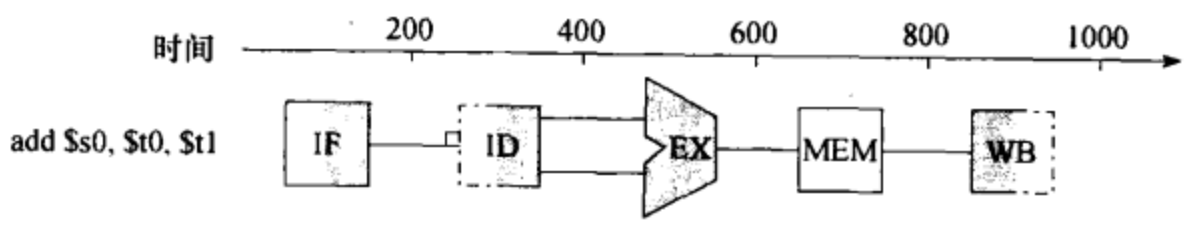


图 4-28 指令流水线的图形表示，其与图 4-25 中的洗衣店流水线类似

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 指令执行的输入的转发连接。

⊖ 数据冒险（data hazard）：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。

⊖ 转发（forwarding）：也称为旁路（bypassing）。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。

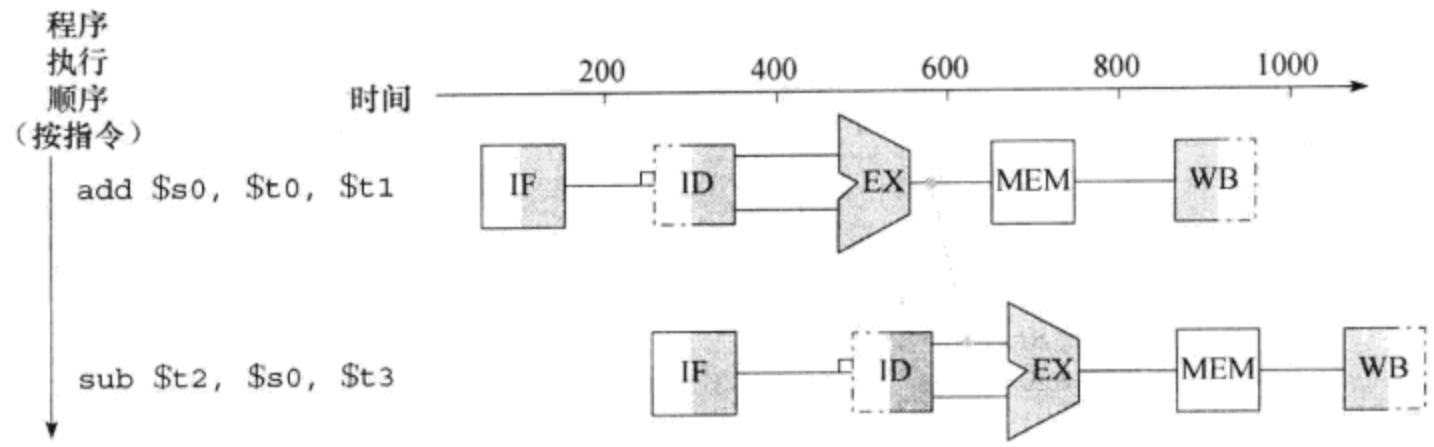


图 4-29 转发的图形表示

图中的连接表示从 add 指令的 EX 操作输出到 sub 指令的 EX 操作输入的转发路径，从而替换掉在 sub 的第二步从寄存器 $s0 读取的值。

在图 4-29 中，只有当目标步骤在时间上晚于源步骤时转发的路径才有效。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转发，因为那样的话将意味着时间的倒流。

转发可以工作得很好，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的第四级完成之后才能生效，这对于 sub 指令的第三级输入来说就太迟了。因此，如图 4-30 所示，即使采用了转发机制，在遇到**装载 - 使用型数据冒险**[⊖]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**[⊖]，但是它经常被昵称为气泡（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。

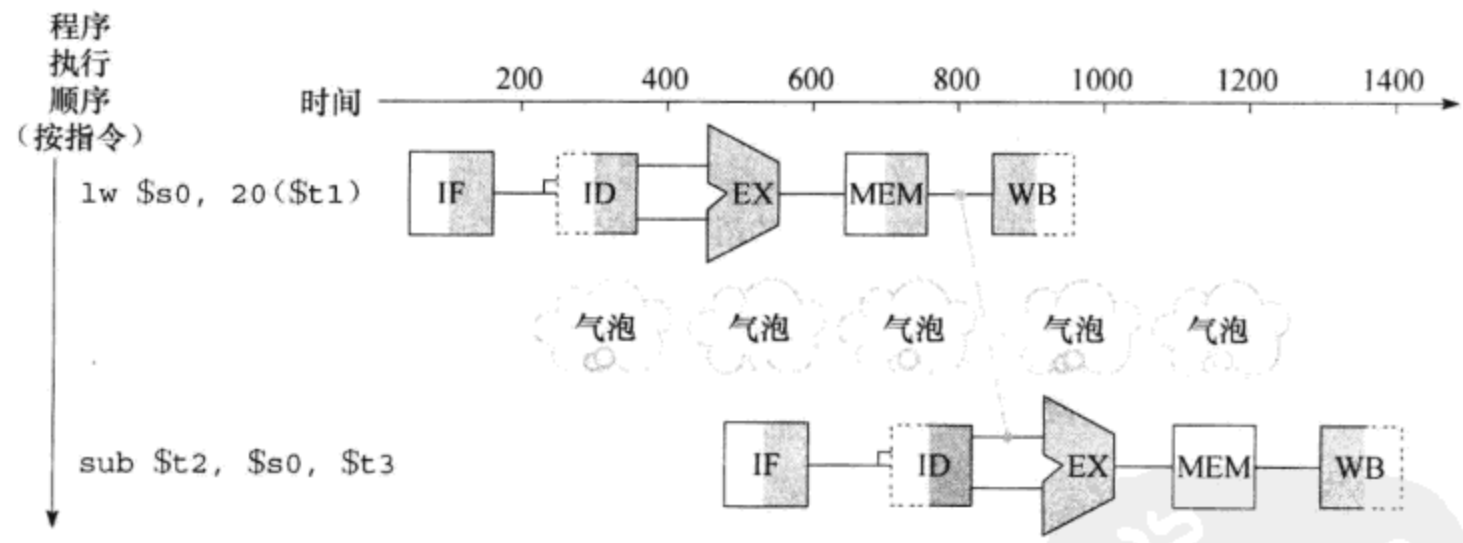


图 4-30 当一条 R 型指令之后紧跟着一条需要使用其结果的装载指令时，即使使用了转发机制，仍然会产生一次阻塞

如果不进行一次阻塞的话，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。

举例 **重新安排代码以避免流水线阻塞**

考虑下面这段 C 代码：

⊖ 装载 - 使用型数据冒险（load-use data hazard）：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（pipeline stall）：也称为气泡（bubble）。为了解决冒险而实施的一种阻塞。

```
a = b + e;
a = 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$st0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)
```

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。

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，通过转发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)
```

在一个具有转发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体系结构指令集的四个原则，由转发可以得到设计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转发设计要复杂得多。

精解：“转发”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

第三种冒险叫做**控制冒险**[⊖]。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，而其正在执行中。

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。

⊖ 控制冒险（control hazard）：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。

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。如果分支未实现应执行的 lw 指令，会被阻塞一个 200 ps 的额外时钟周期。

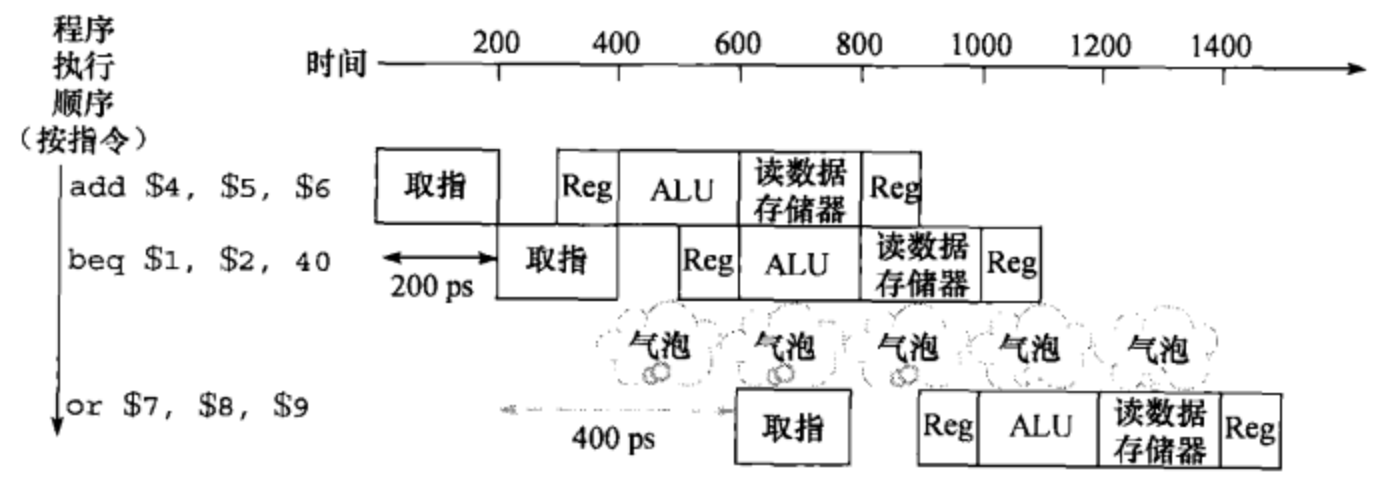


图 4-31 在每一个条件分支上阻塞是避免流水线控制冒险的一种解决方法

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。

举例 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。

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。

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：

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。

一种更加成熟的**分支预测**[⊖]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（branch prediction）：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。

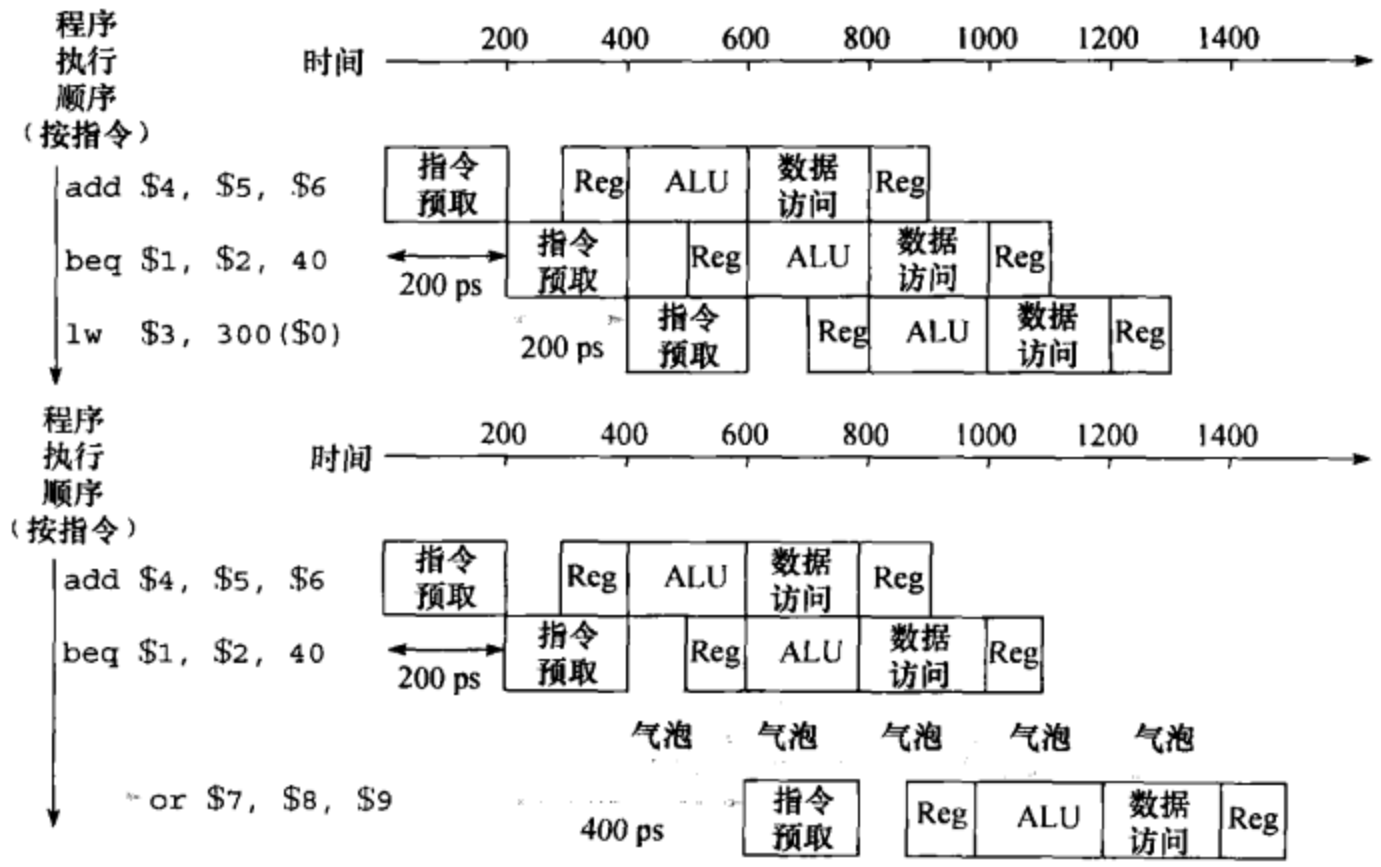


图 4-32 预测分支未发生是一种避免流水线控制冒险的解决方法

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。

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指令的行为，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，这种硬件预测分支的方式能够达到 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。

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 的汇编程序员们是透明的。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。

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节中学习转发和阻塞的实现。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，哪个只使用转发就可以避免阻塞，而哪个既不需要阻塞也不需要转发就可以执行。

指令序列1	指令序列2	指令序列3

lw $t0,0($t0)
add $t1,$t0,$t0
	add $t1,$t0,$t0
	addi $t2,$t0,#5
	addi $t4,$t1,#5
		addi $t1,$t0,#1
		addi $t2,$t0,#2
		addi $t3,$t0,#2
		addi $t3,$t0,#4
		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。与之相比，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调度来减少数据间的依赖。

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**[⊖]。例如，一个五级流水线仍然需要五个周期来完成一条指令。用第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**。

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、转发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

看起来东西很多，其实不然。

——Tallulah Bankhead，remark to Alexander Woollcott，1922

⊖ 延迟（latency）：流水线的级数或者顺序执行过程中两条指令间的级数。

图 4-33 是摘自 4.4 节的一个单时钟周期的数据通路。将指令划分为五个阶段意味着一个流水线采用五级，也就意味着在任何一个单时钟周期内，最多会执行五条指令。因此必须把数据通路分为五个部分，每一部分用与之对应的指令执行阶段来命名。

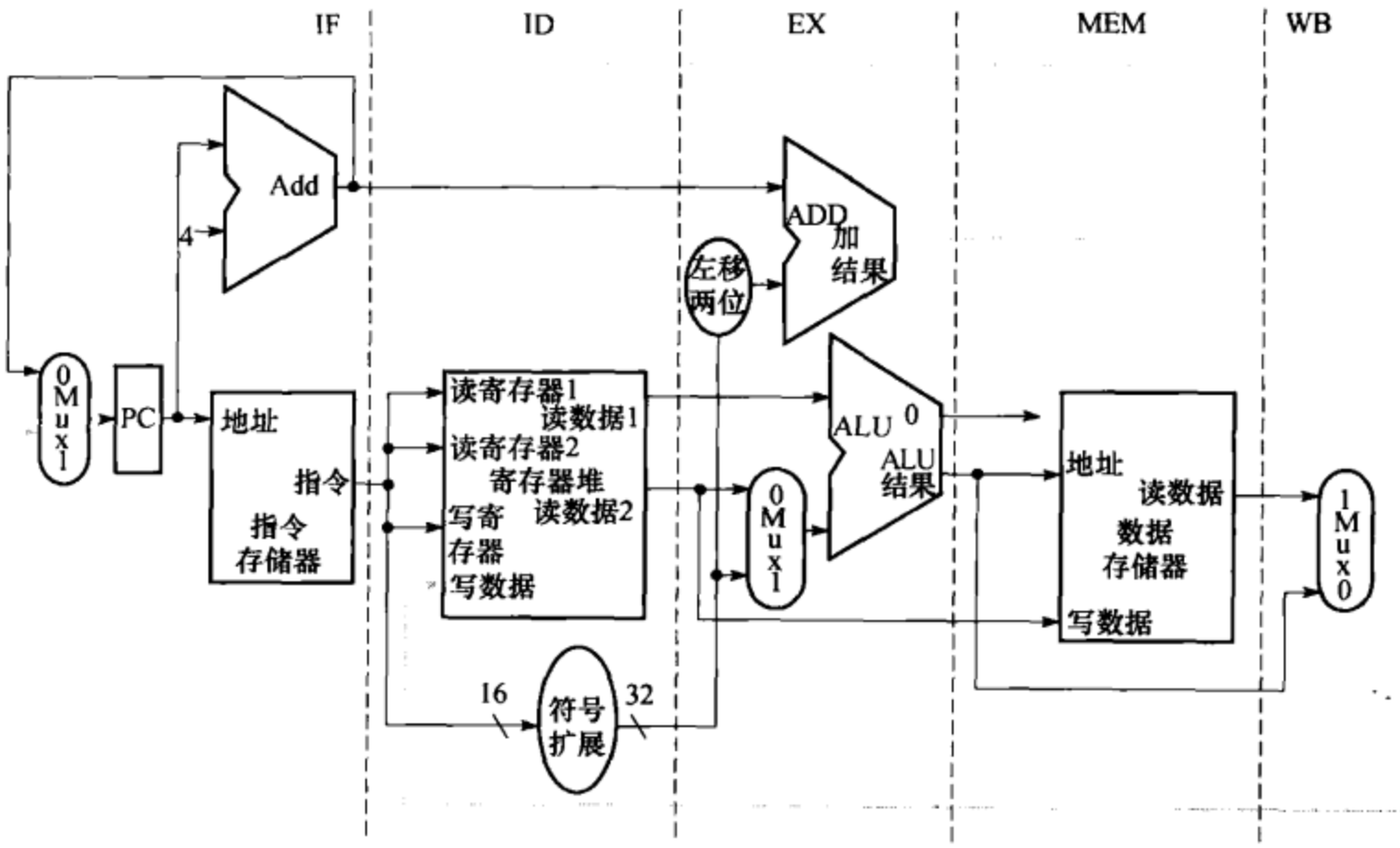


图 4-33　4.4 节中的单时钟周期数据通路（与图 4-17 类似）

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：存储器访问

5）WB：写回

图 4-33 的五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。

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五个步骤中的一步中用到，因此我们允许

它在其他四步中被其他的指令共享。为了在其他四步中保持指令的值，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。

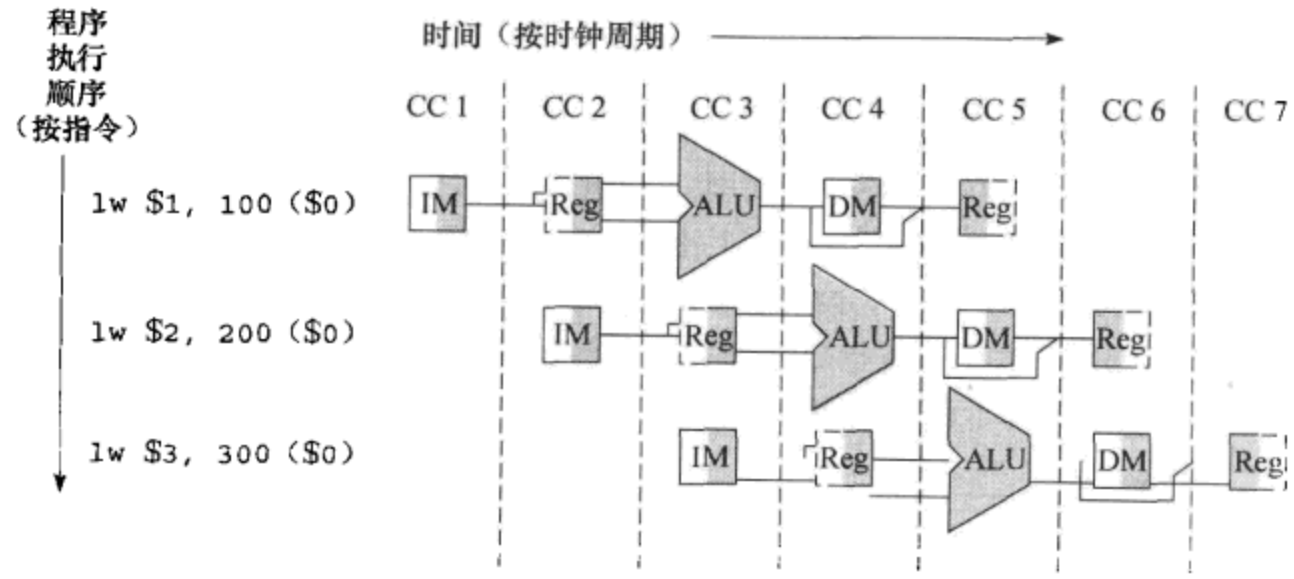


图4-34 按图4-33中的单时钟周期数据通路执行的指令（假定以流水线方式执行）

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，依此类推。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做IF/ID。

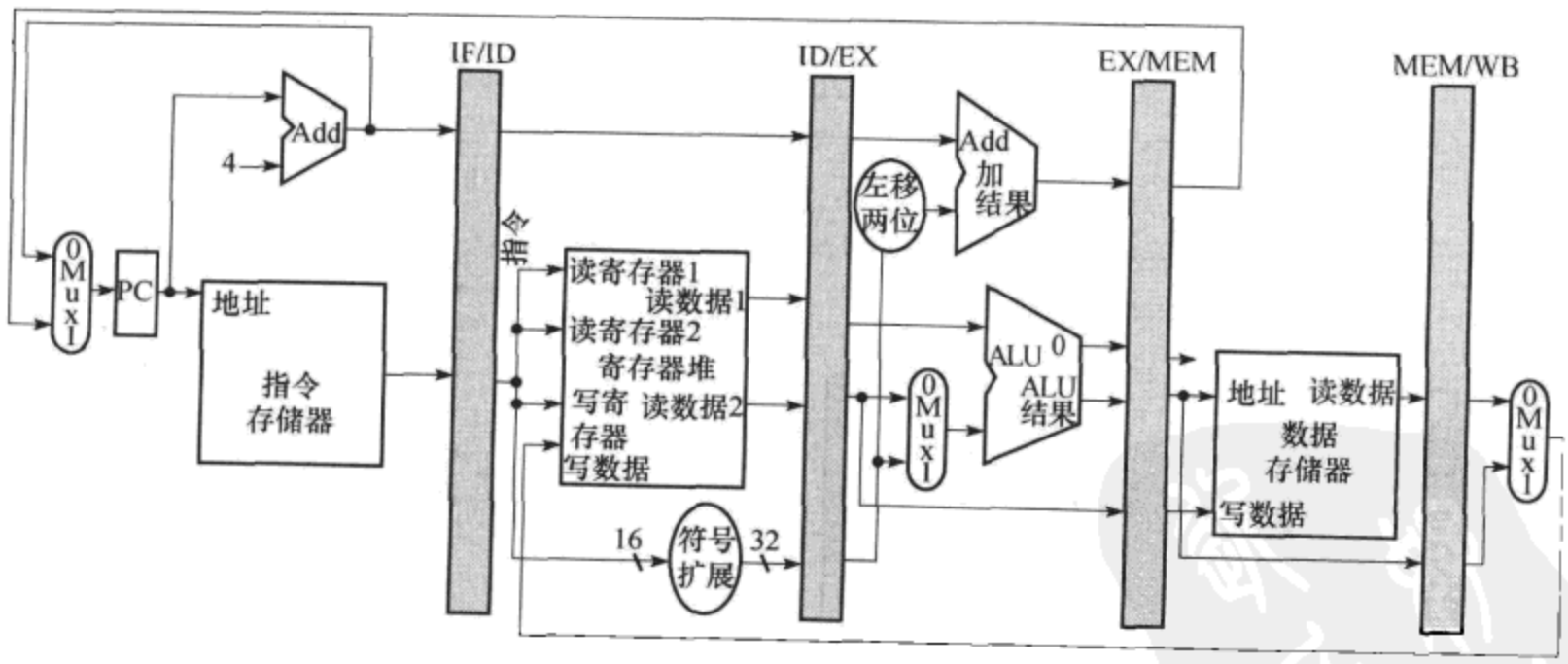


图4-35 图4-33数据通路的流水线版本

流水线寄存器（以灰色标识）将流水线的各部分分开。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提取出来的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。

图4-36～图4-38表示了装载指令在通过流水线的5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5级。正如图4-28～图4-30所显示的那样，当寄存器

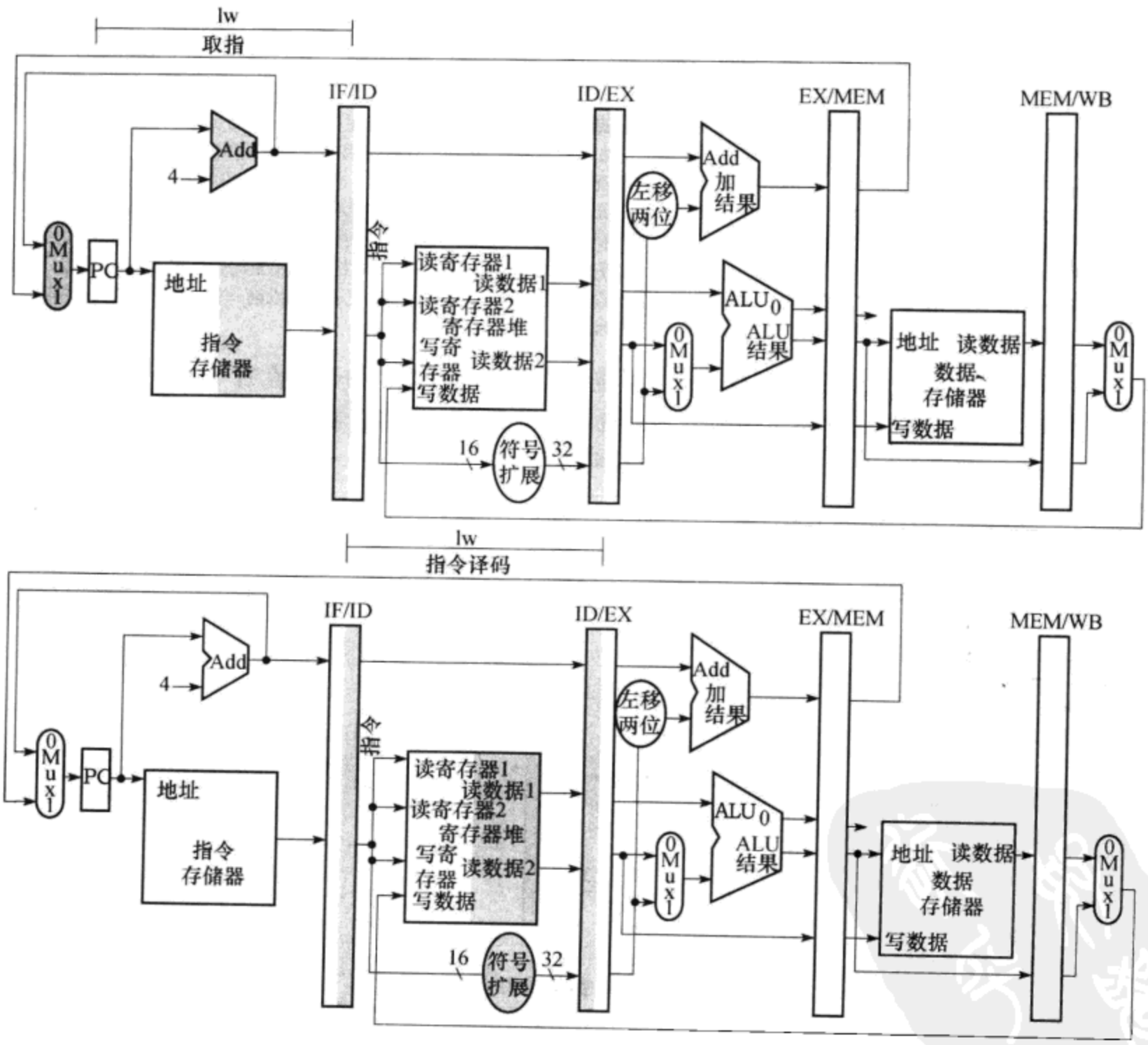


图4-36 IF和ID：lw指令在流水线中的第一、二步，图4-35中活动的数据通路部件用灰色表示

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。

或存储器被读取时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。

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，并用 ALU 将它们相加，和值存入 EX/MEM 流水线寄存器中。

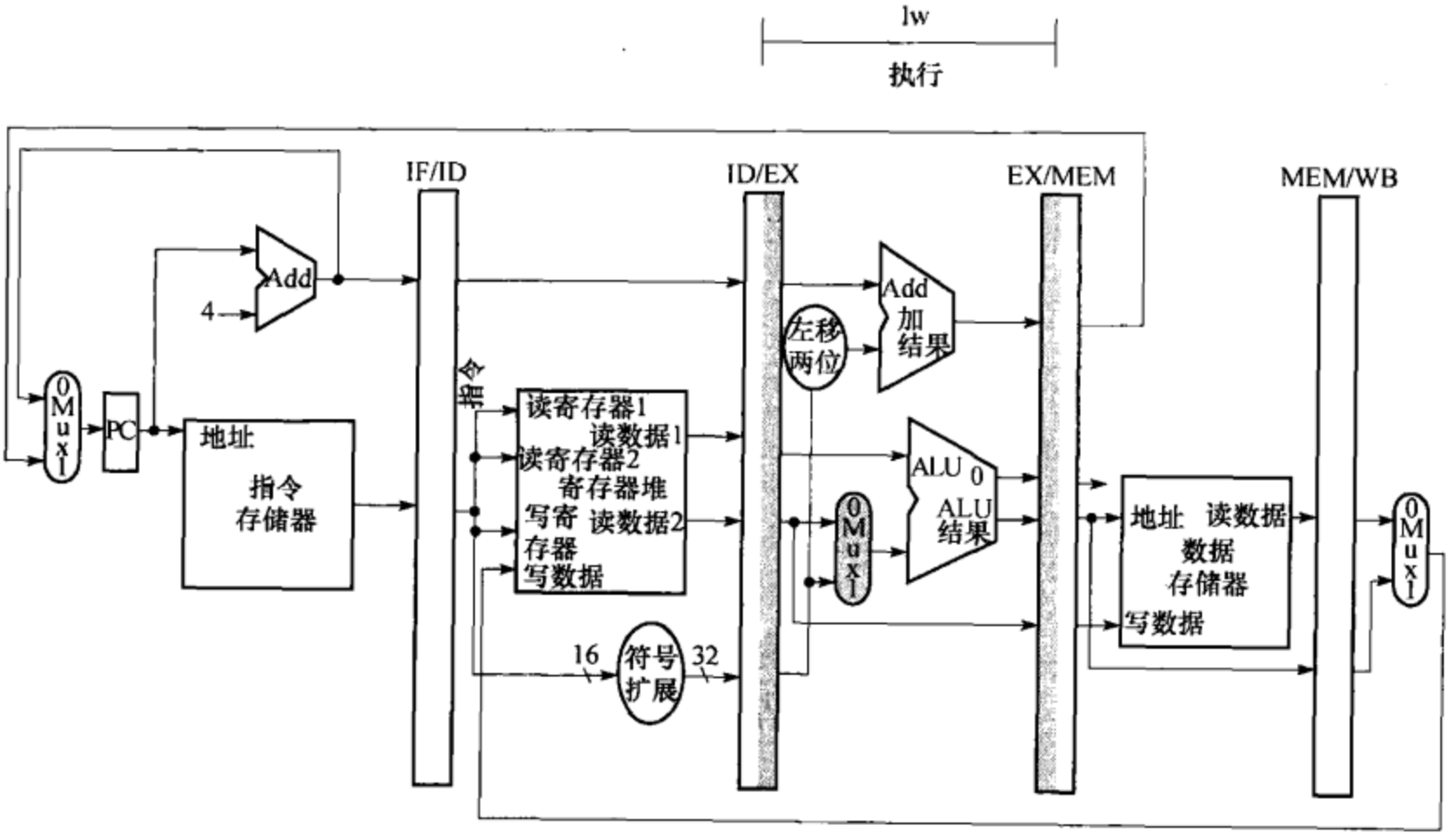


图 4-37 EX：lw 指令在流水线中的第三步，图 4-35 中活动的数据通路部件用灰色表示

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。

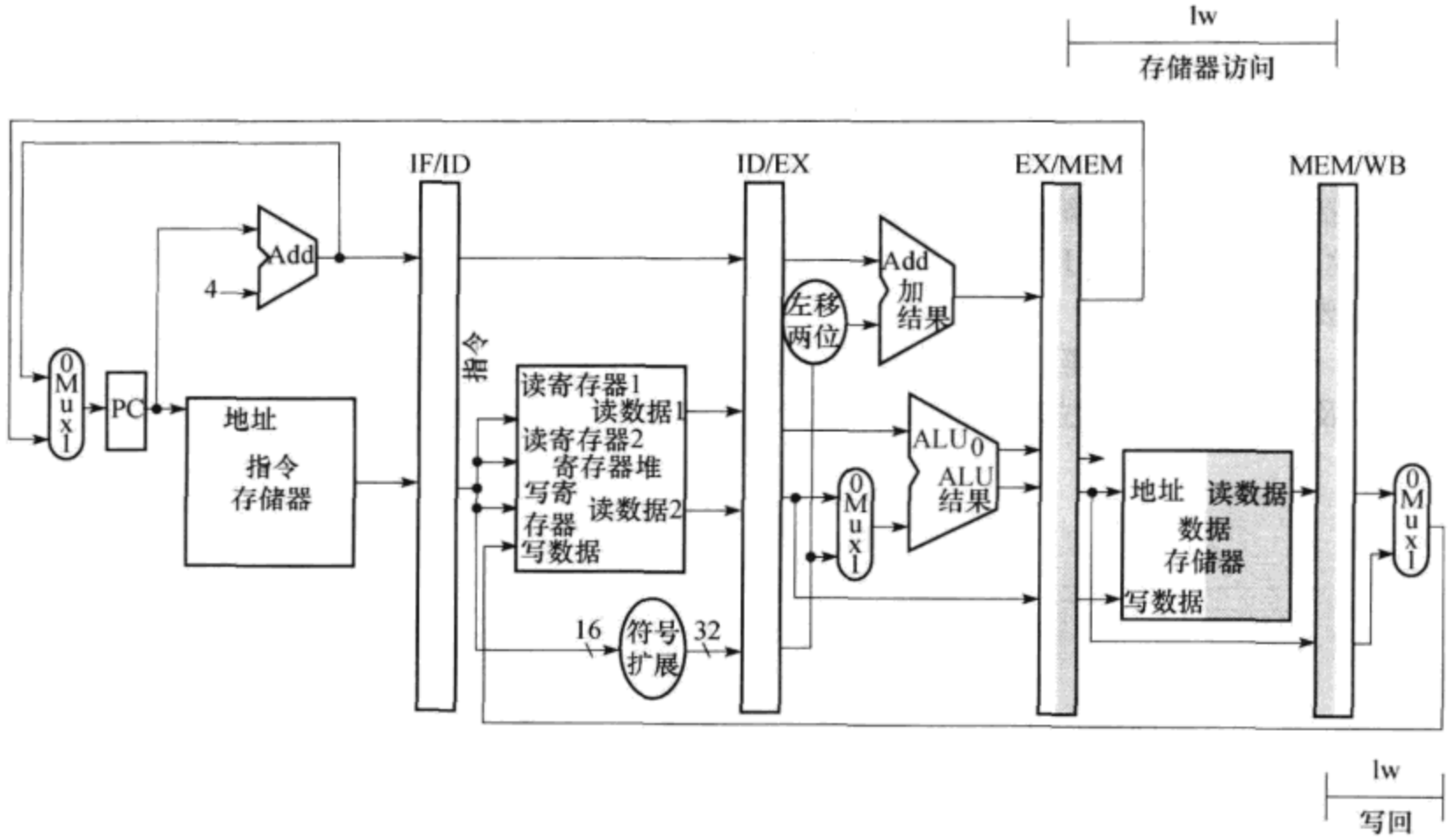
4）存储器访问：图 4-38 的顶端表示装载指令使用从 EX/MEM 流水线寄存器中得到的地址读取数据存储器，并将数据存入 MEM/WB 流水线寄存器中。

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寄存器堆。

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数据必须通过流水线寄存器传递。存储指令也是如此。下面是存储指令的五个执行步骤：

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。

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，因此所有指令的执行这两个步骤都相同。



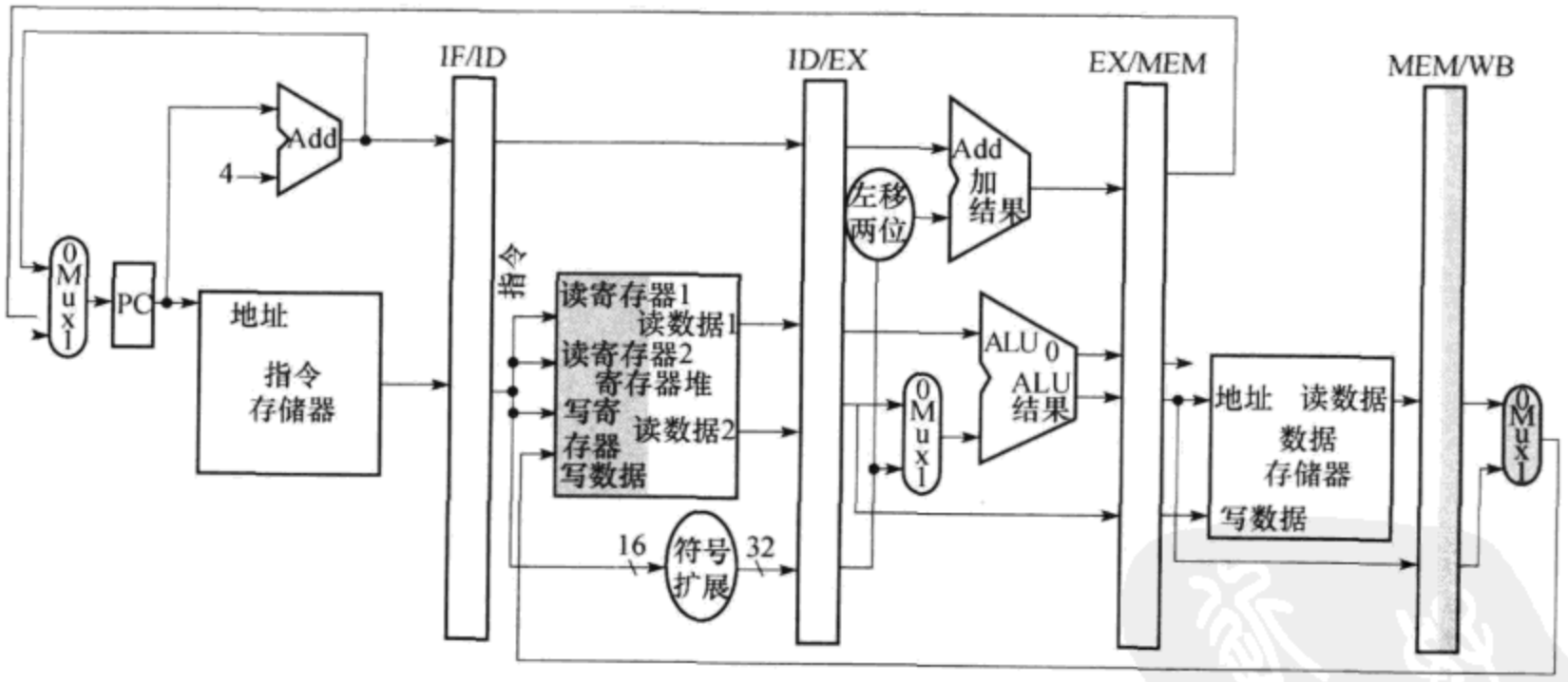


图 4-38　MEM 和 WB：lw 指令在流水线中的第四步和第五步，图 4-35 中活动的数据通路部件用灰色表示

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到 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。

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。

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 中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。

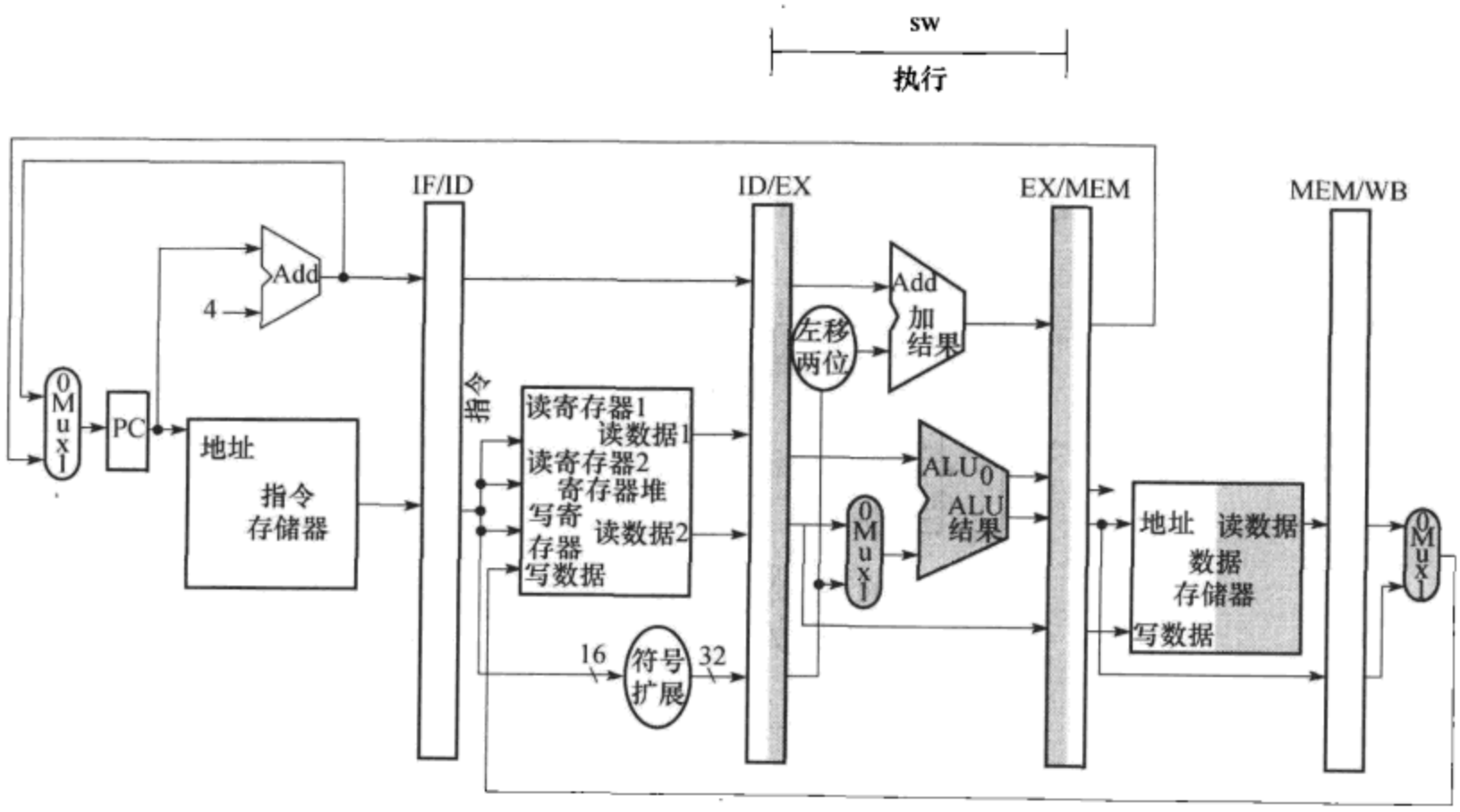


图 4-39 EX：sw 指令在流水线中的第三步

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。

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。

装载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功能单元（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）。所以这些功能单元可以和一个流水级相联系。

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 装载指令在所有五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。

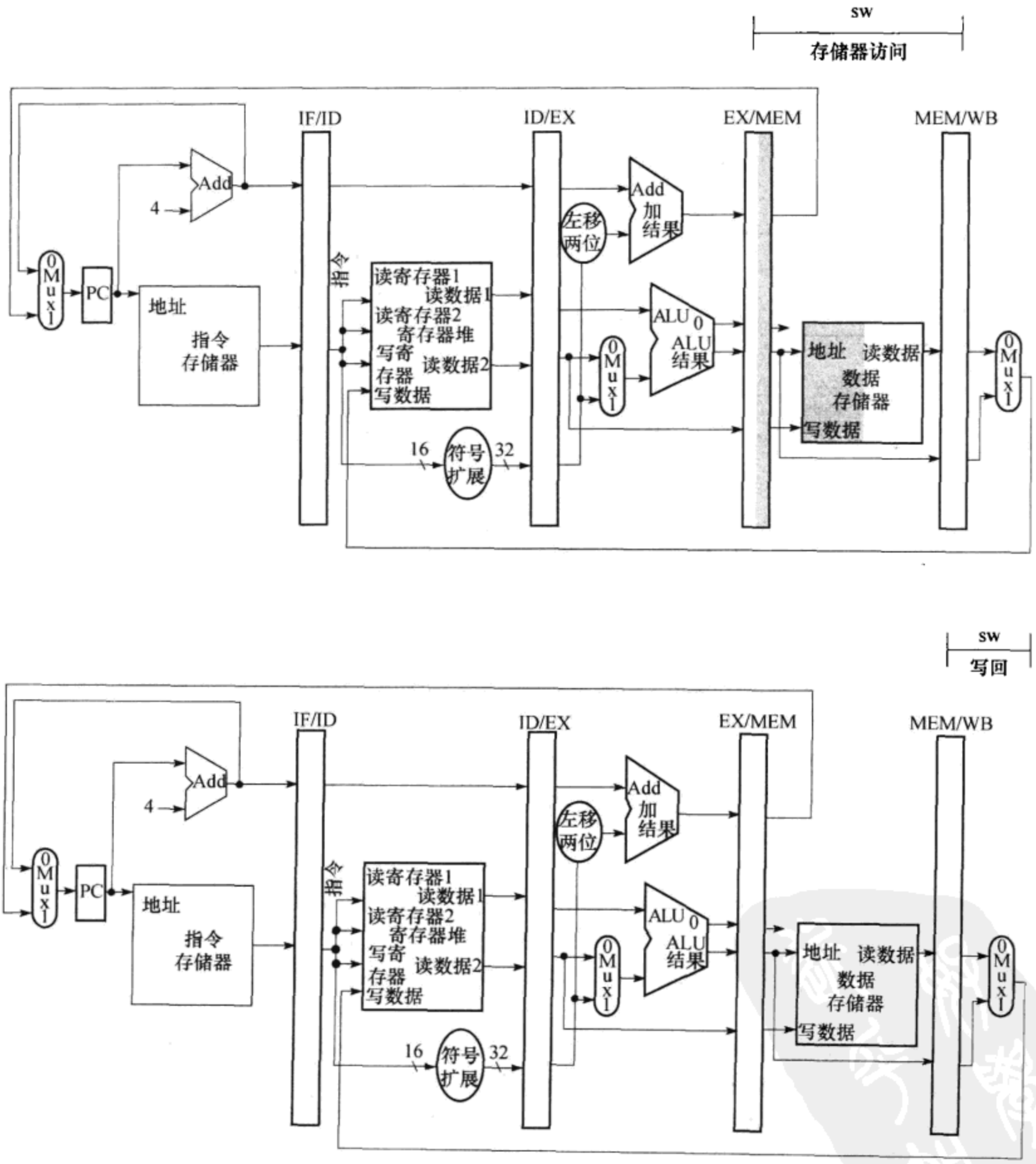


图 4-40 MEM 和 WB：*sw* 指令在流水线中的第四步和第五步

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。

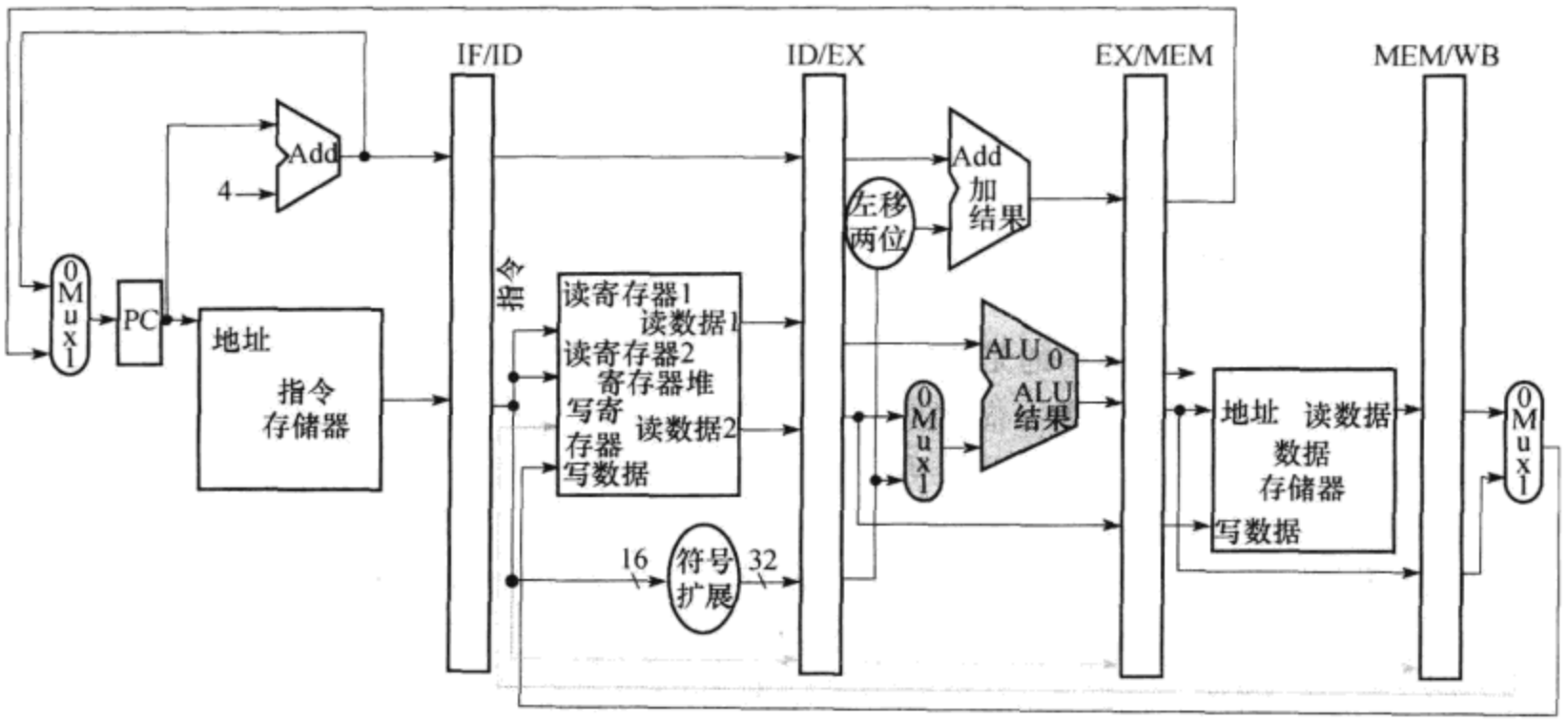


图 4-41 可正确执行装载指令的流水线数据通路

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 流水级一直传送到 MEM/WB 流水线寄存器。新的路径以灰色线标识。

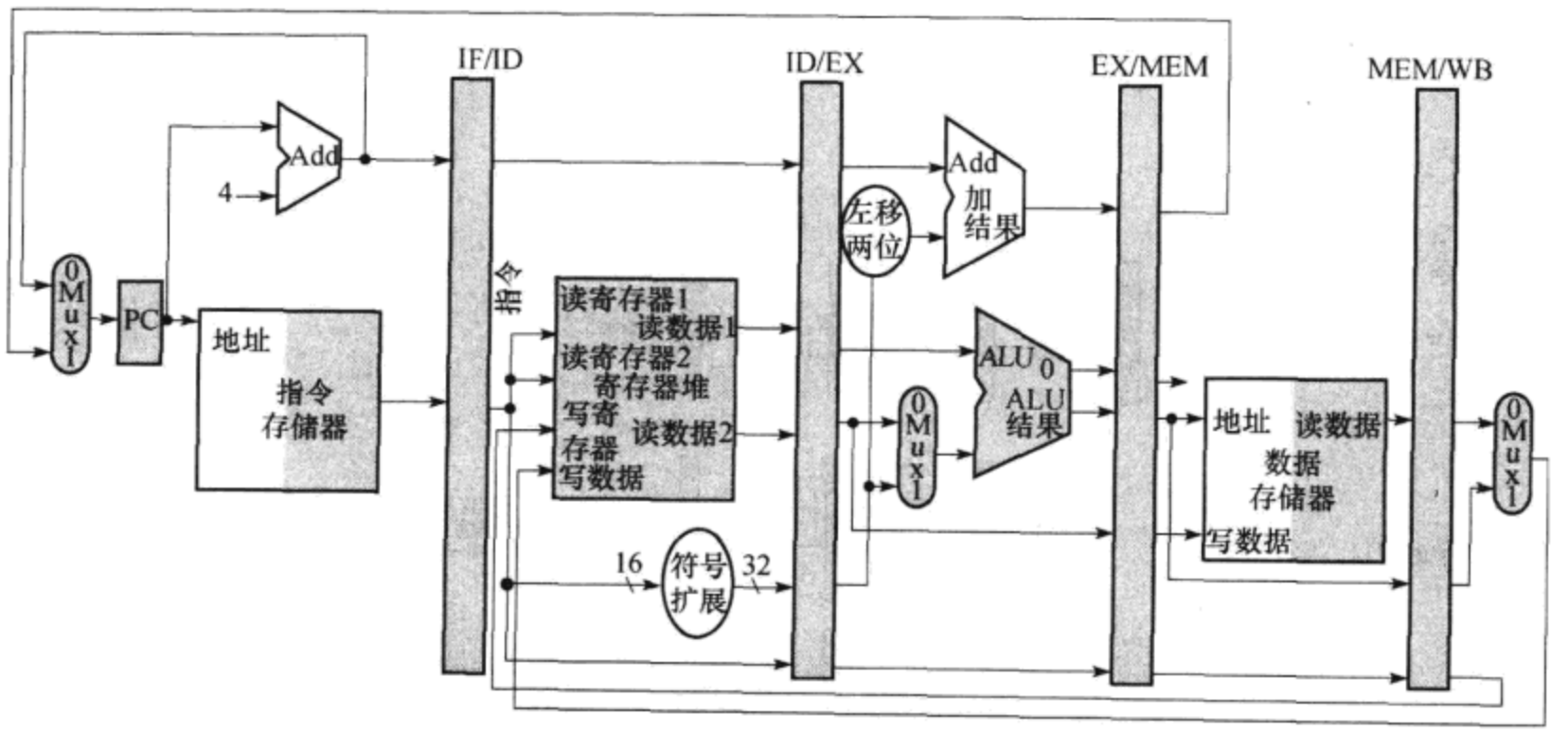


图 4-42 图 4-41 中在装载指令的五级流水线中用到的全部数据通路

4.6.1 图形化表示的流水线

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，有两种基本的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）。多时钟周期虽然简单但不包括所有的细节。下面以这五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，通常所有五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 4.12 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 的第五个时钟周期。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。

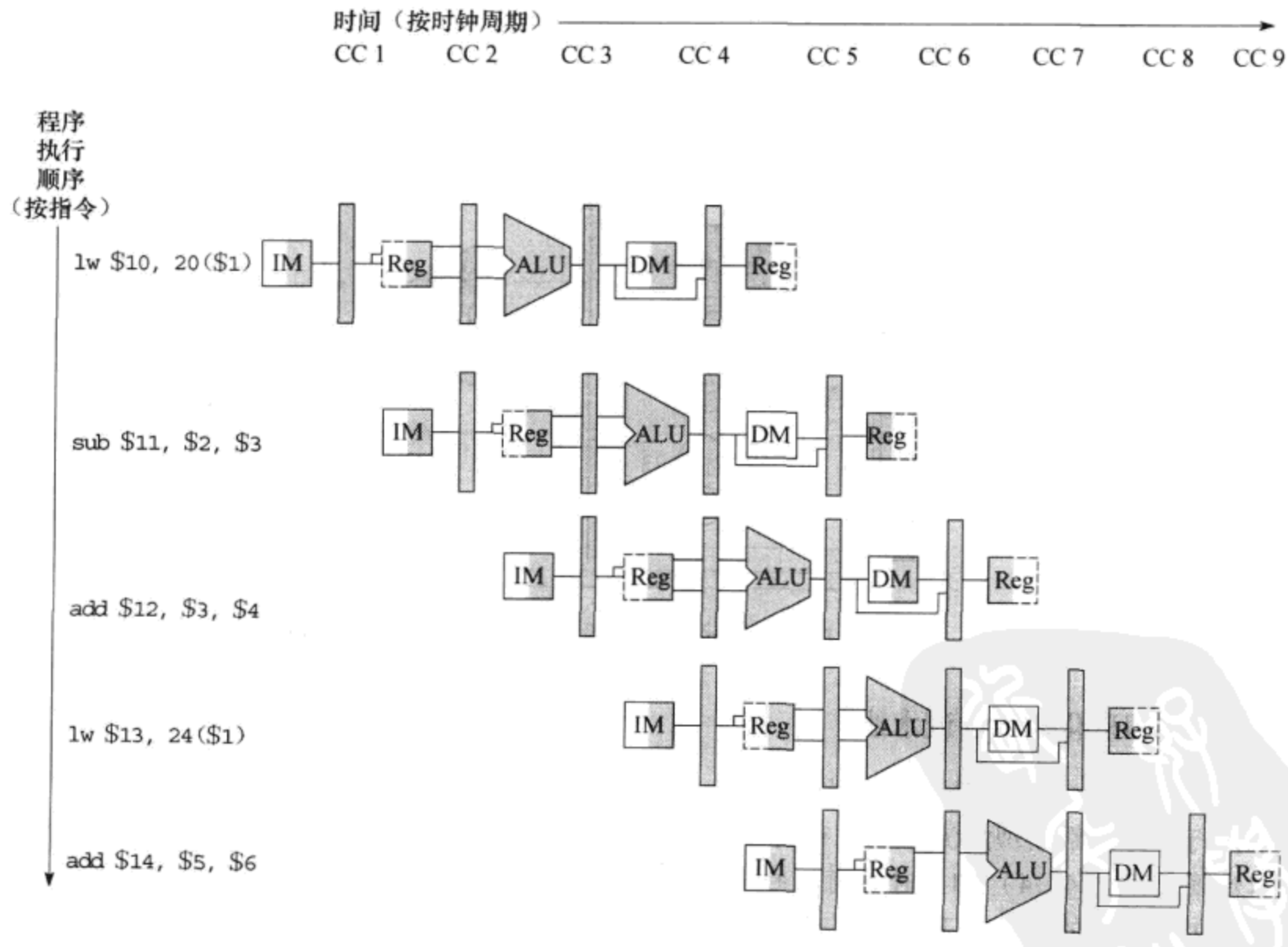


图 4-43 五条指令的多时钟周期流水线图

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。

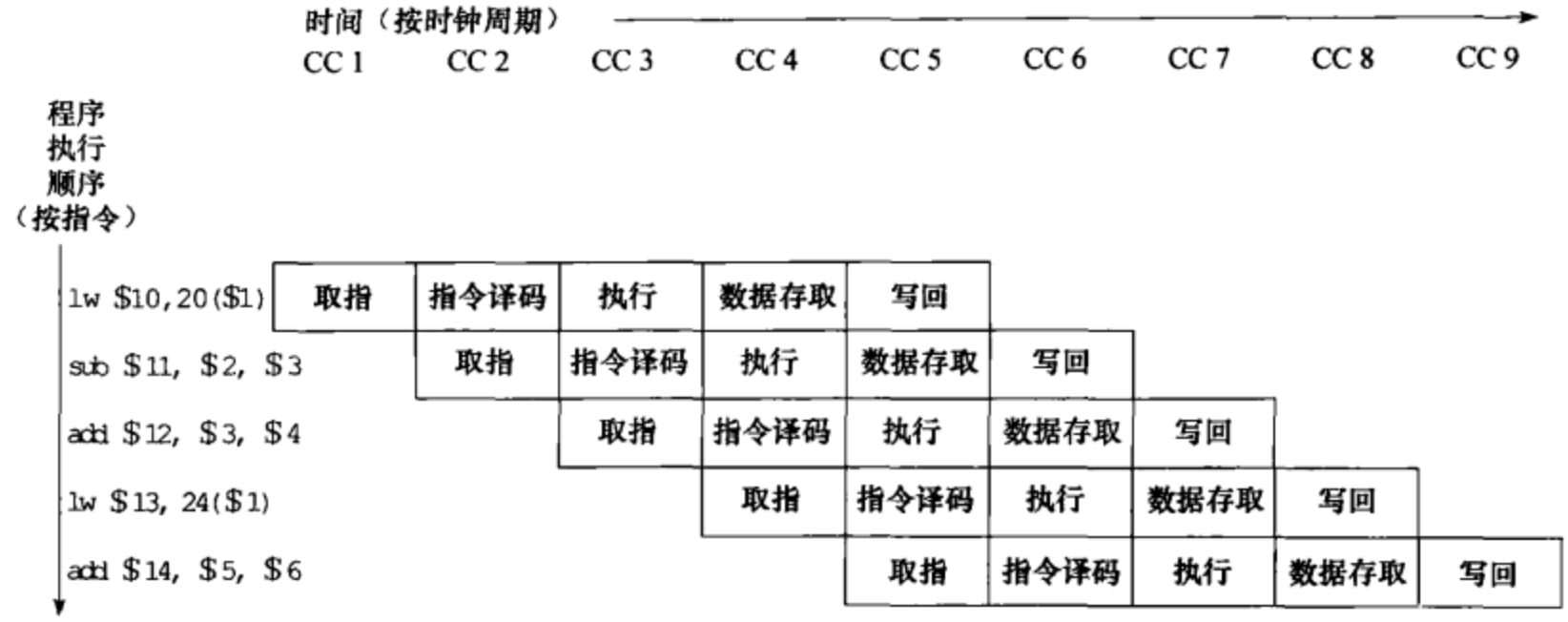
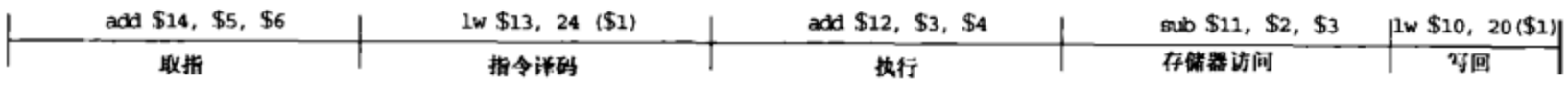


图 4-44 相对图 4-43 更为传统的多时钟周期流水线图



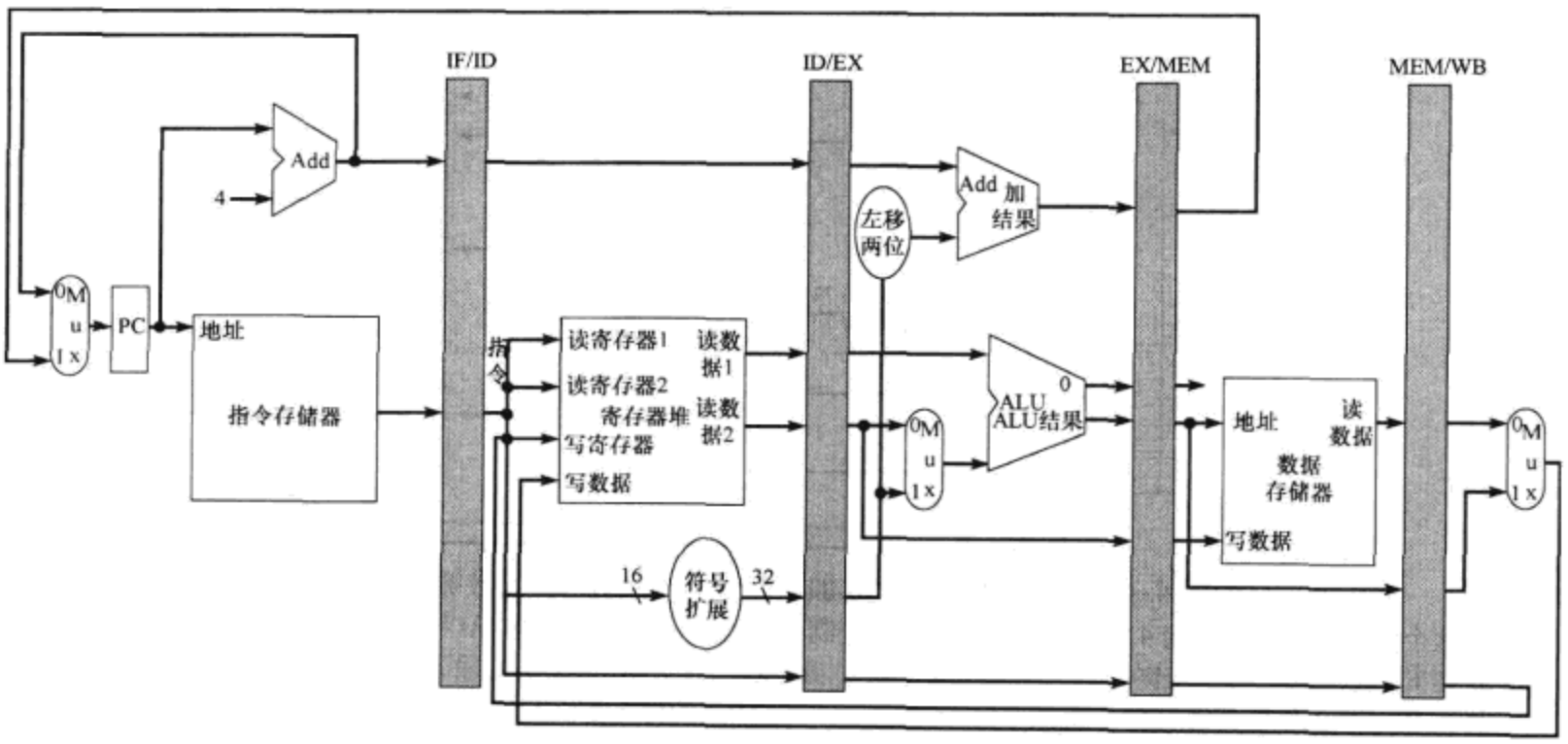


图 4-45 对应图 4-43 和图 4-44 的流水线第 5 个时钟周期的单时钟周期流水线图

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？

A. 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。

B. 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。

C. 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。

D. 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

相对以前的任何计算机，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 节介绍了在简单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，然后在 4.7 节 ~4.9 节中逐步去掉这些限制条件。

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。

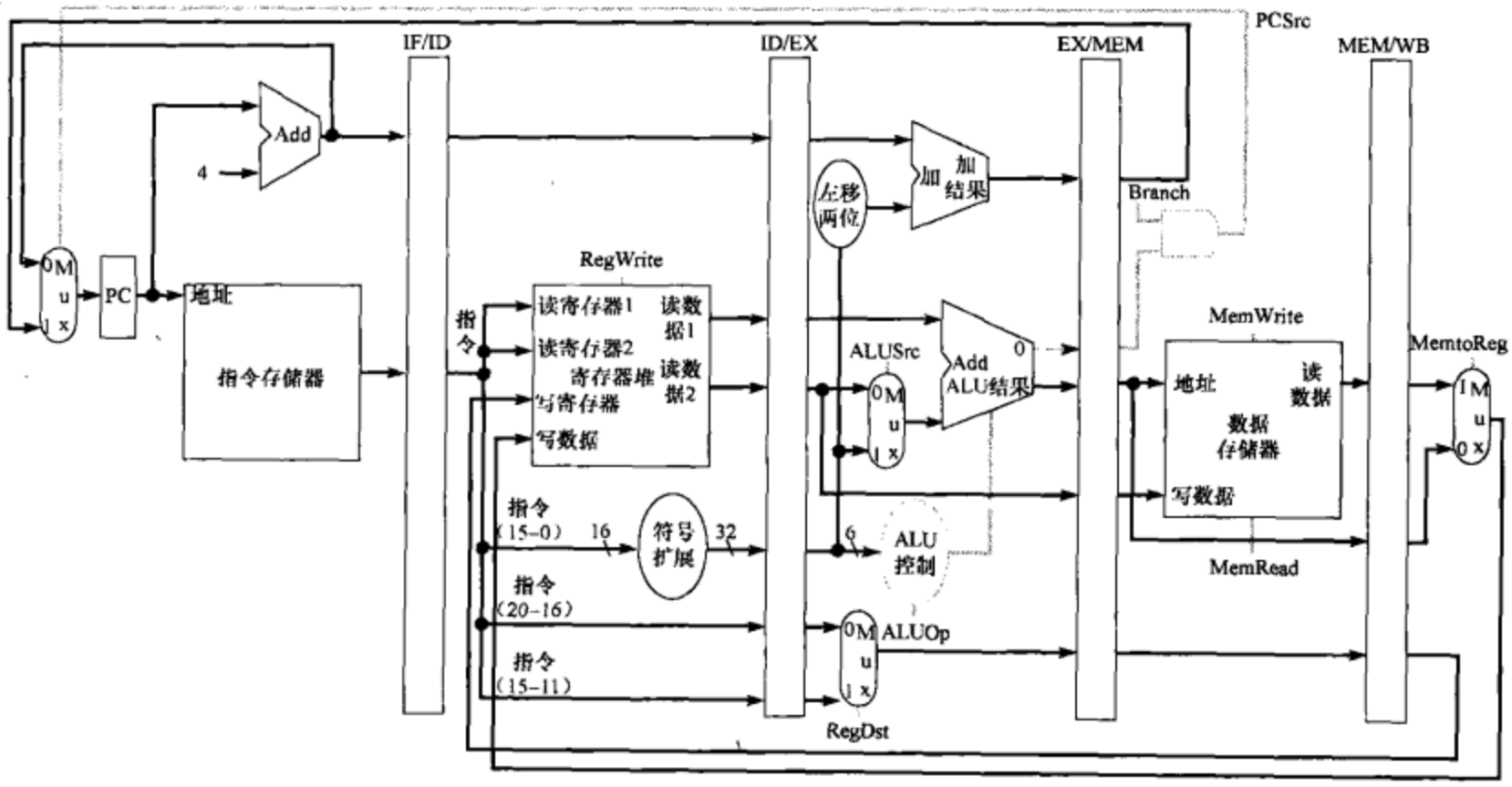


图 4-46 在图 4-41 上增加了控制信号的流水线数据通路

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。

指令操作码	ALUOp	指令操作	功能码	ALU 操作	ALU 控制信号
LW	00	取字	XXXXXX	加	0010
SW	00	存储字	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 1	101010	小于则置 1	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。

⊖ 需要保持副本与正本完全相同。——译者注

信号名	置无效时的效果（0）	置有效时的效果（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

图中定义了七个控制信号的功能。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				存储器存取阶段的控制信号			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

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五级将控制信号分成五组：

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）。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。除非控制电路断言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。

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，其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，4.12 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。）

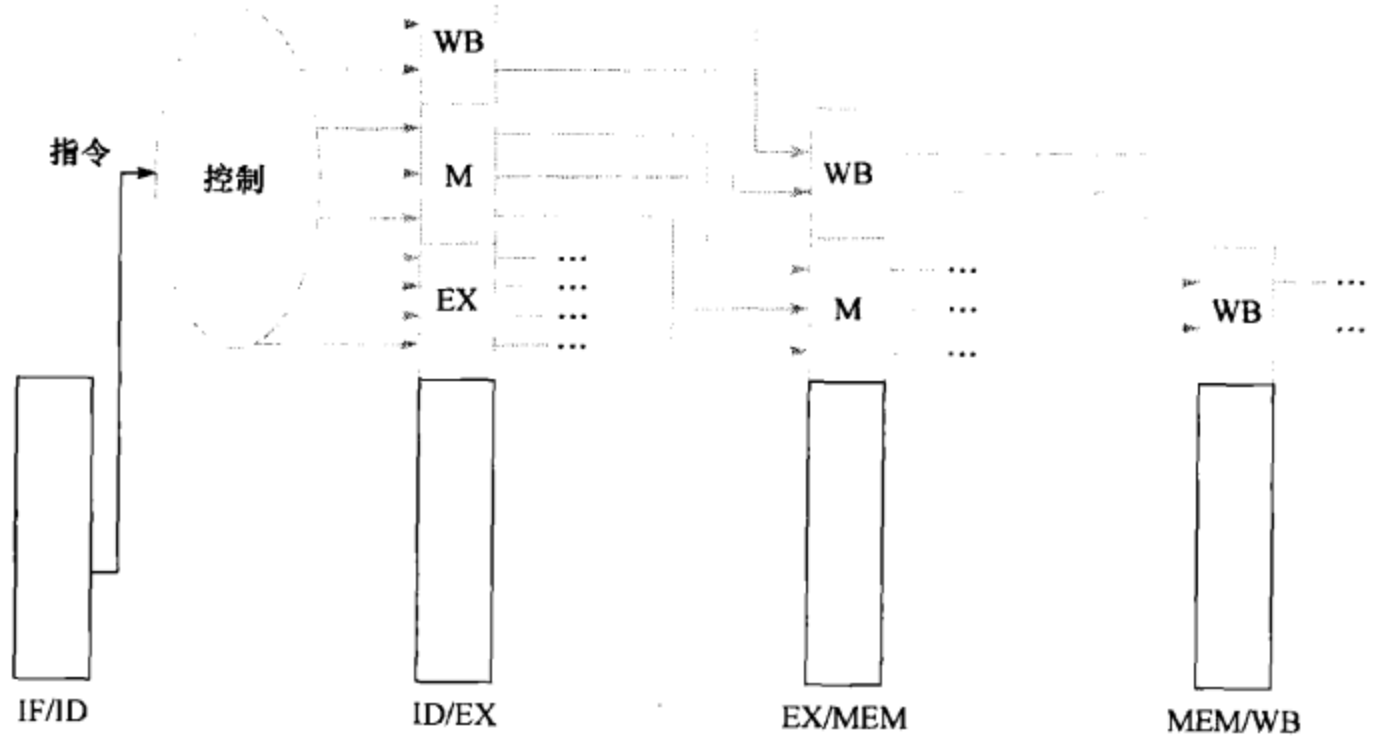


图 4-50 流水线最后三级的控制信号

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。

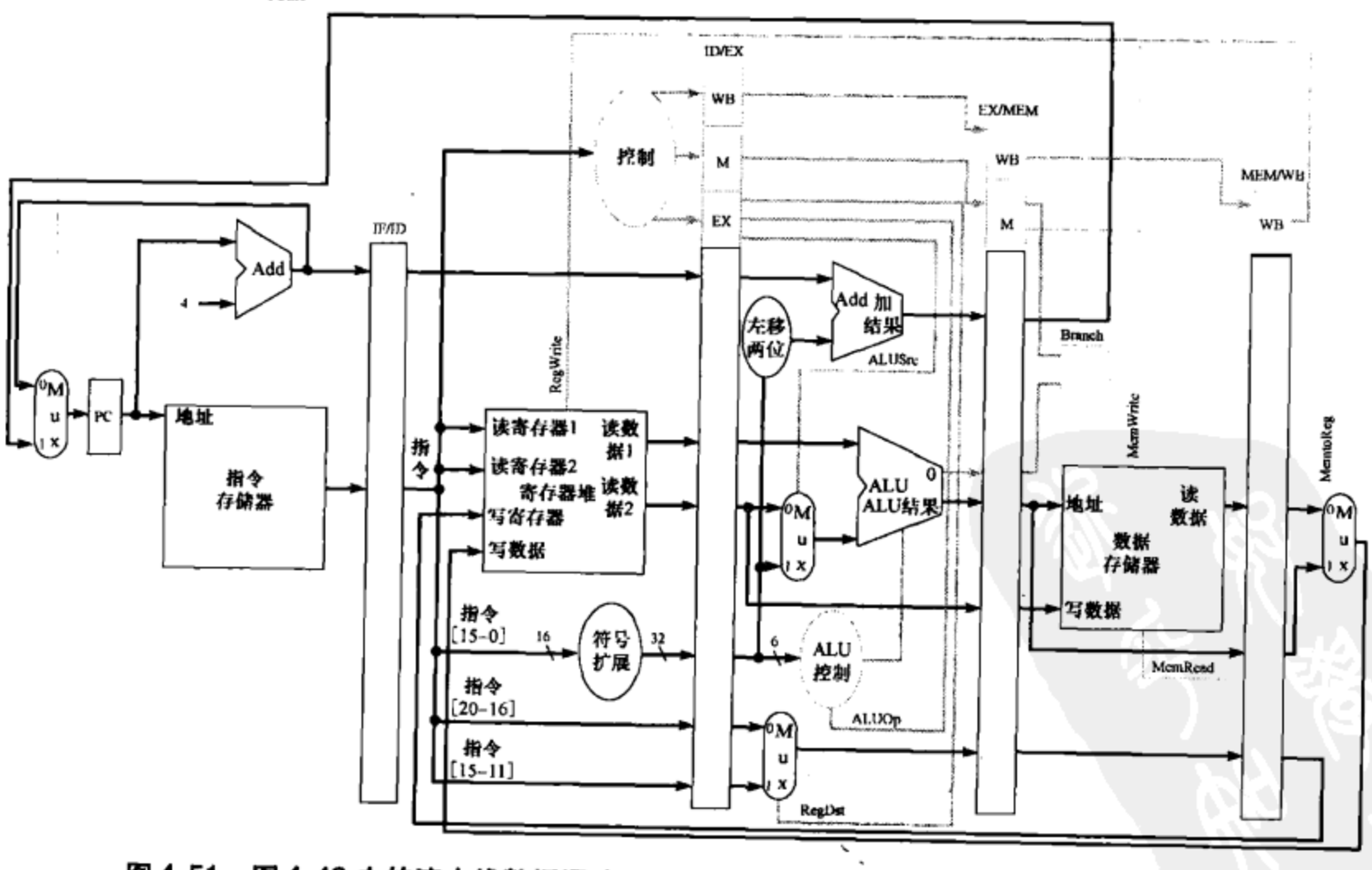


图 4-51 图 4-46 中的流水线数据通路，已将控制信号连接到流水线寄存器的控制部分

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： 转发与阻塞

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障碍之一。

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
```

后四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，程序员认为后四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。

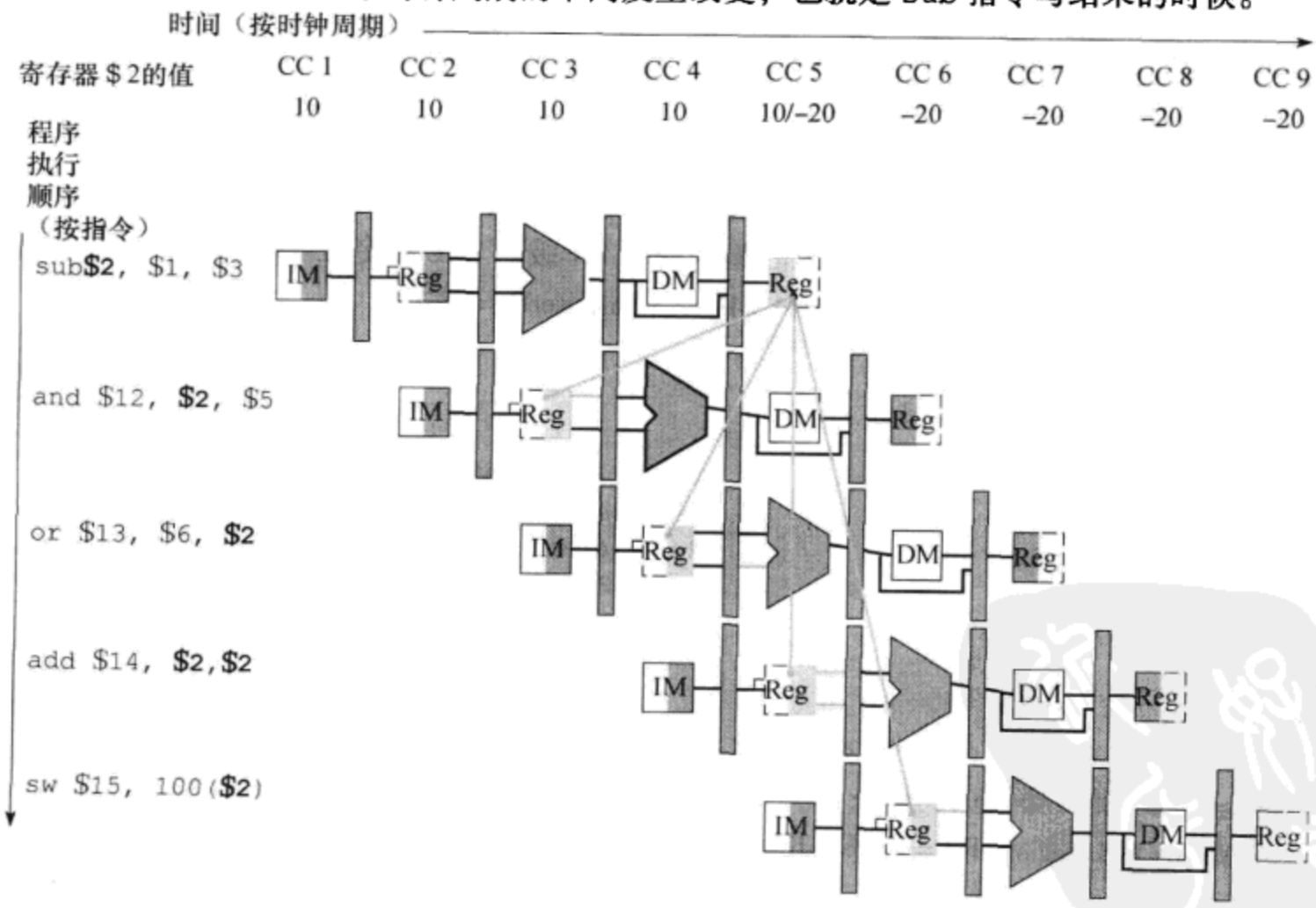


图 4-52 使用多时钟周期流水线图表示流水线中指令序列的相关性

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，后四条指令读寄存器 $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线表示。那些导致时间后退的依赖就是流水线数据冒险。

最后一个潜在的数据冒险可以通过设计相应的寄存器堆硬件解决。如果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 sub2 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转发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。

转发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。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

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。

举例 **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) & 2nd($2) set by sub
sw      $15,100($2)         #Index($2) set by sub
```

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，直接采用总是转发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转发。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 是否被设置。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 按非零结果转发，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，但仍然需要解决转发数据策略的问题。

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 级写操作的寄存器堆。由于流水线寄存器保存了需要转发的数据，因此后面的指令能够获得相应的数据。

时间（按时钟周期）	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
寄存器 $2的值:	10	10	10	10	10/–20	–20	–20	–20	–20
EX/MEM的值:	X	X	X	–20	X	X	X	X	X
MEM/WB的值:	X	X	X	X	–20	X	X	X	X

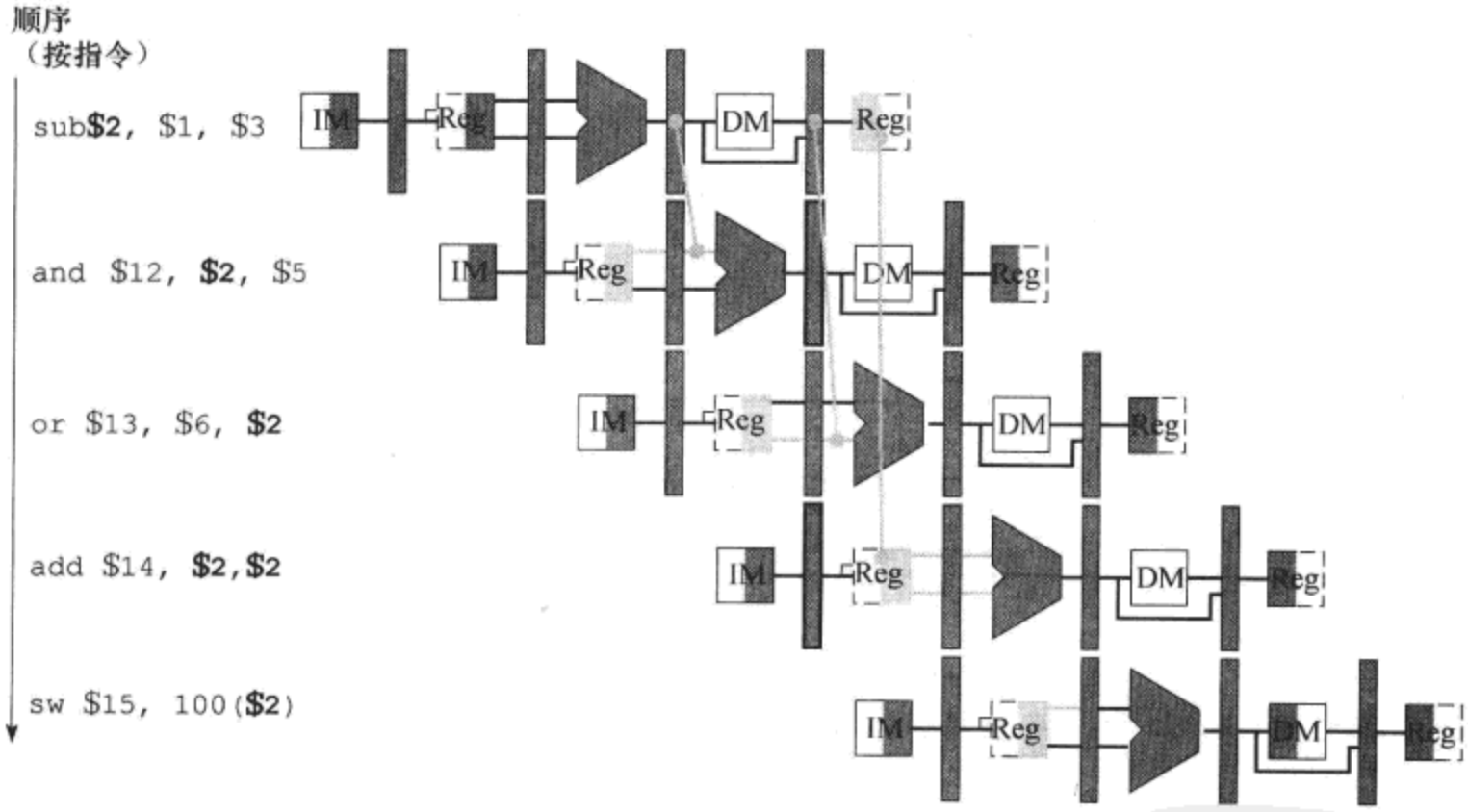
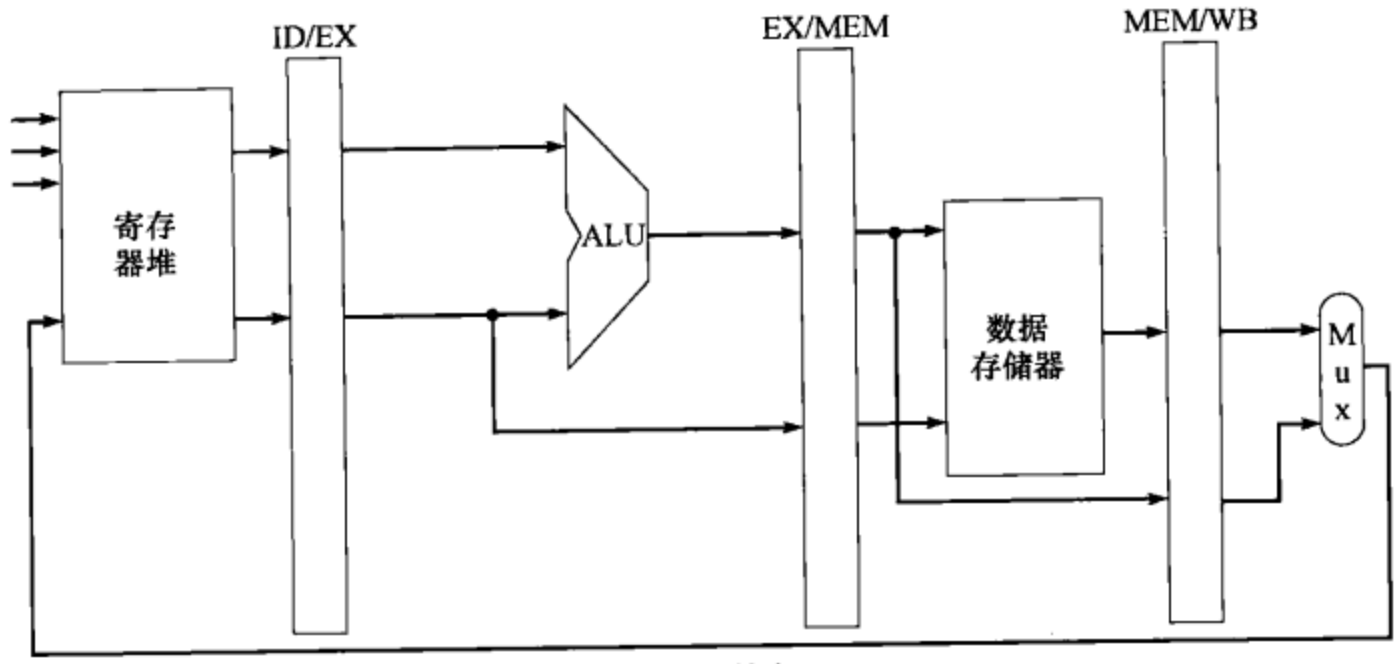


图 4-53 流水线寄存器和 ALU 间的相关性。通过转发流水线寄存器中保存的结果就有可能提供 AND 指令和 OR 指令所需的 ALU 输入

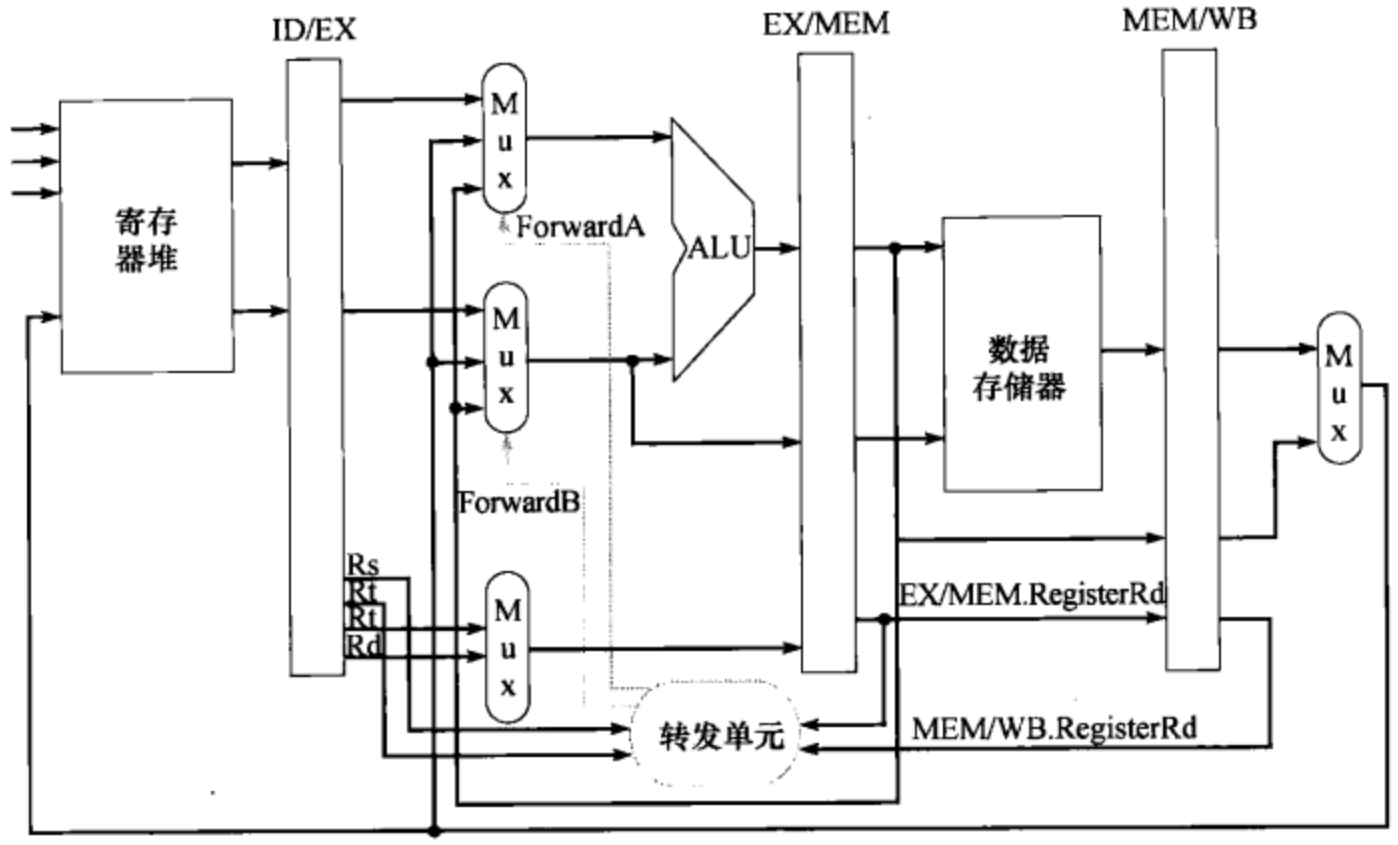
流水线寄存器存有相应的值，在数据写入寄存器堆之前就已经有效了。如果寄存器堆可在同一时钟周期内转发要读写的数据，add 指令就不用阻塞了。这种寄存器堆的转发的值不是来自于流水线寄存器而是来自寄存器堆。它使得寄存器 $2 中的值在第五个时钟周期的开始是 10，而在周期结束时是 –20，即在这一时钟周期里读操作读到的值是写操作写入的值。在本节下面的部分，我们将处理所有的转发（除了存储指令要存的数值之外）。

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，那么就可以转发所需的数据。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，假设需要转发的指令只有四个 R 型指令：add、sub、AND 和 OR。图 4-54 给出了在加入转发机制前后 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转发的数值间进行选择的 ALU 多选器控制信号的值。



a）不转发



b）转发

图 4-54　加入转发机制前后的 ALU 和流水线寄存器

下图使用多选器增加了转发路径，并标识了转发单元。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，一根连接到转发单元，但实际上它是一个信号。如前所述，这里还忽略了转发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。

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 运算结果转发获得
ForwardA = 01	MEM/WB	第一个 ALU 操作数从数据存储器或者前面的 ALU 结果中转发获得
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 运算结果转发获得
ForwardB = 01	MEM/WB	第二个 ALU 操作数由数据存储器或者前面的 ALU 结果转发获得

图 4-55　图 4-54 中转发多选器的控制信号

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。

因为 ALU 转发多选器在 EX 中，所以转发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，以决定是否转发相应的值。我们已经有了 rt 字段（20～16 位）。在支持转发前，ID/EX 流水线寄存器未保存 rs 字段。因此，为支持转发，rs（25～21 位）被加入 ID/EX 流水线寄存器中。

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 域是 ALU 指令（来自 Rd 域）或装载指令（来自 Rt 域）的目标寄存器号。

这种情况是将前一条指令的结果转发到任何一个 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转发，但这种转发只发生在寄存器堆内部。

更为复杂的潜在数据冒险发生在 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
……
```

在这种情况下，由于 MEM 级的结果是最新的，因而结果是由 MEM 级转发得到。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≠LD/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给出了为了支持转发EX级结果所增加的必要硬件设备。注意，图中EX/MEM.RegisterRd域是一条ALU指令（来自Rd域）或装载指令（来自Rt域）的目标寄存器。

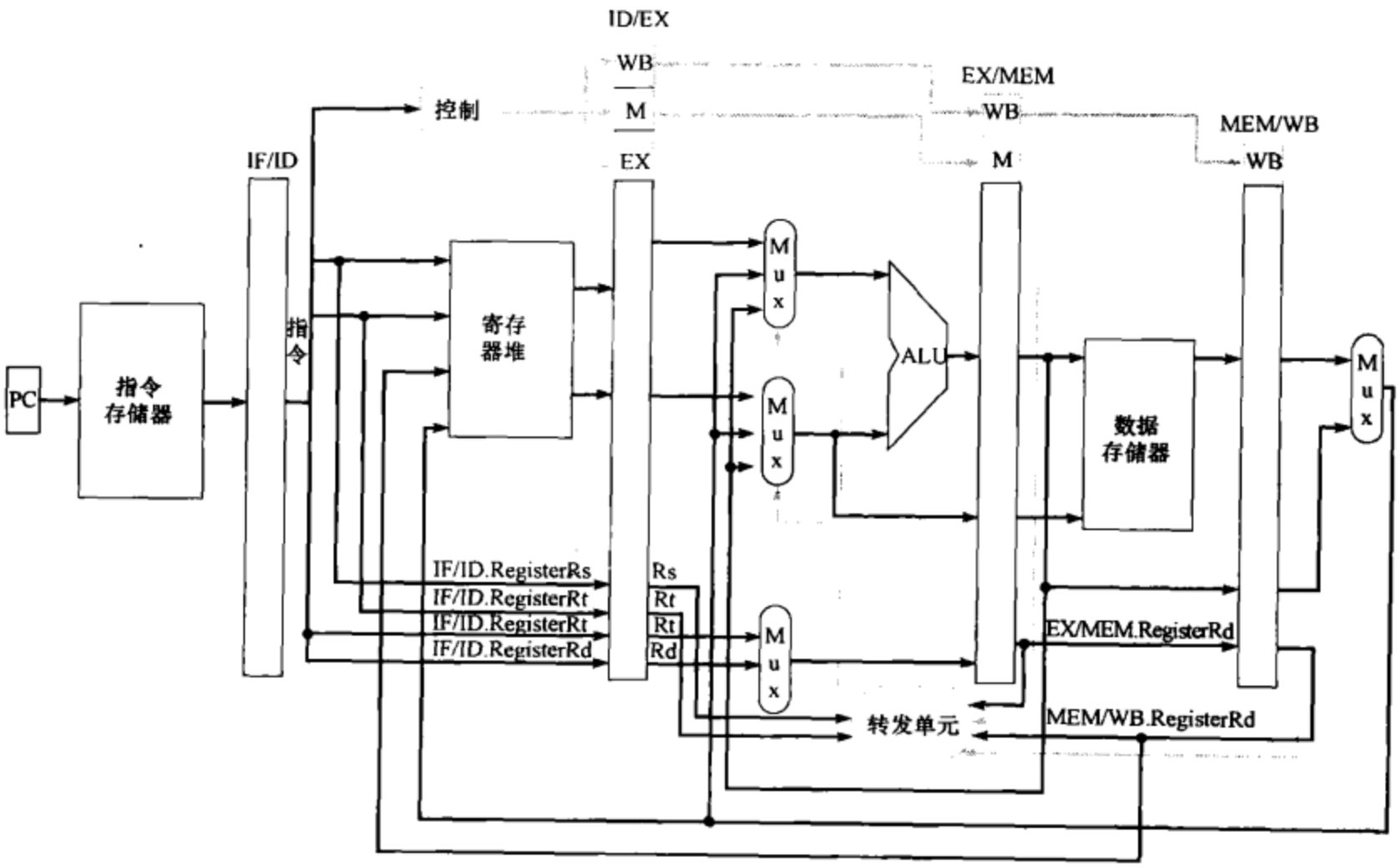


图4-56 通过转发解决冒险的数据通路

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。

在CD中的4.12节给出了两段MIPS代码，其中存在需要使用转发解决的冒险，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：转发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，所以转发应当容易。但在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转发硬件。如果我们重画图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转发。我们将如何对其修改作为练习题留给读者。

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，而且转发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。

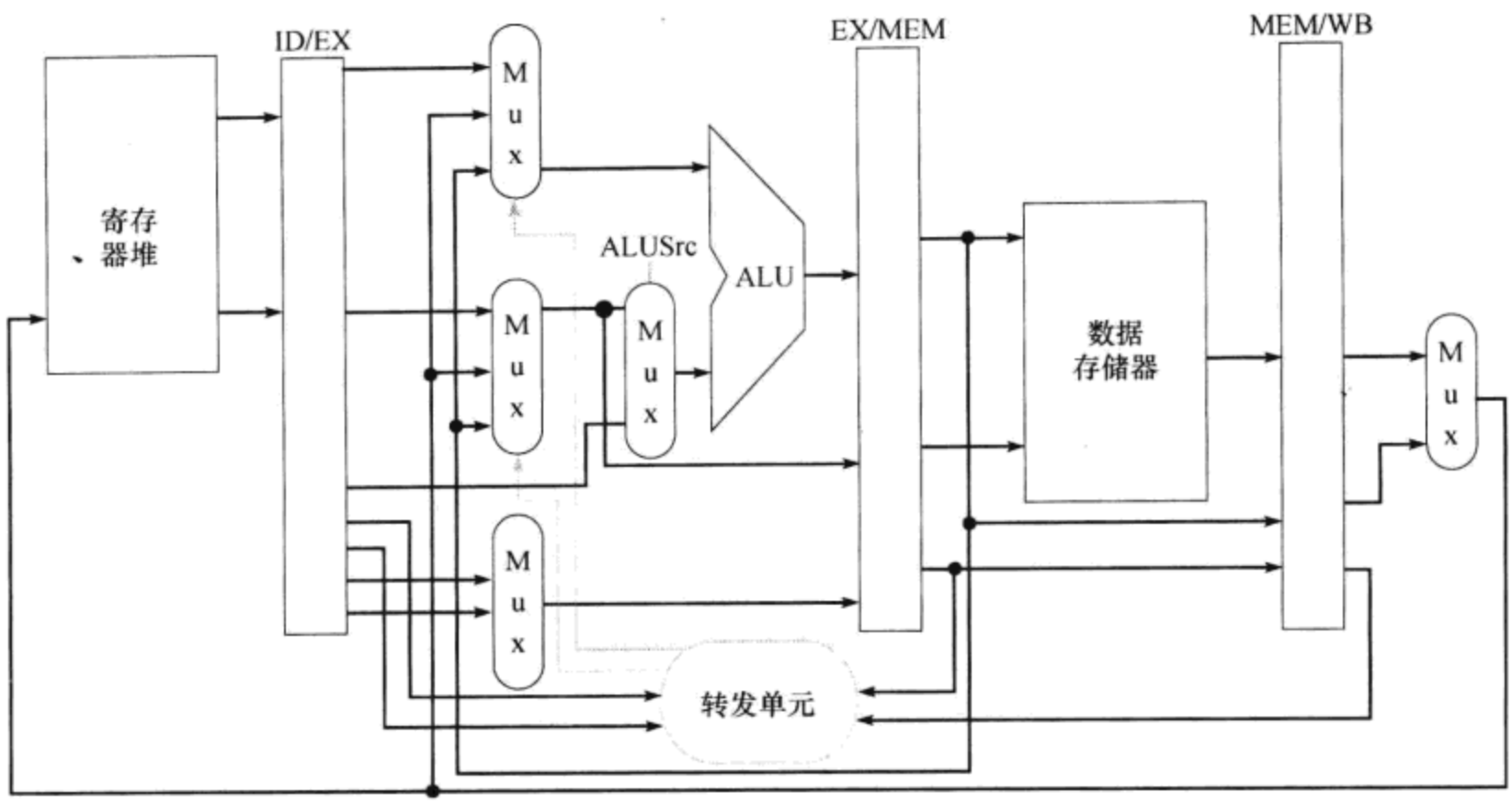


图 4-57 在图 4-54 中的数据通路加入了一个 2:1 的多选器，用以选择带符号立即数作为 ALU 的输入

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，就无法使用转发解决冒险了。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

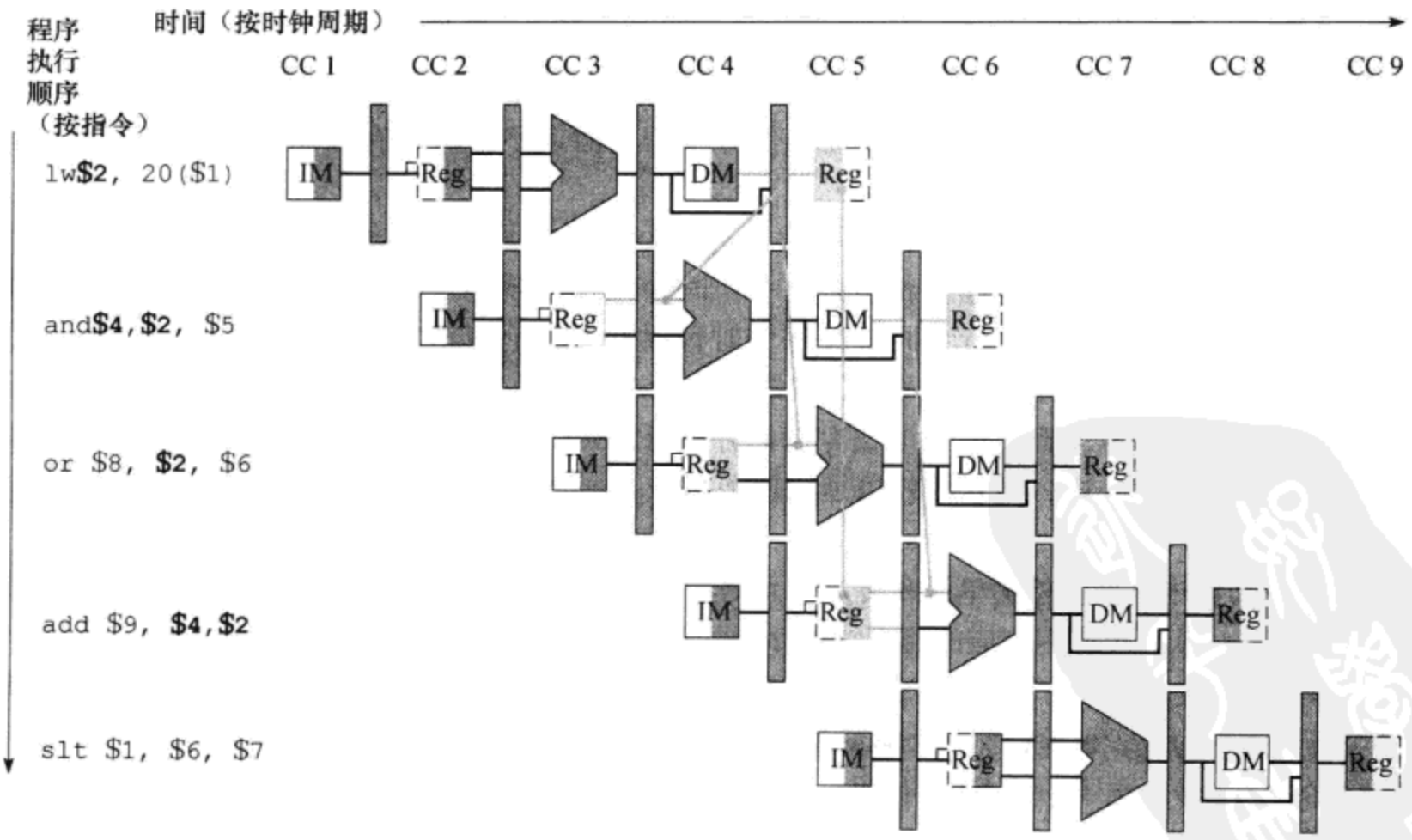


图 4-58 一个指令序列的多时钟周期流水线图

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转发来解决。因此，这类指令组合导致冒险检测单元产生阻塞。

四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。

因此，除了一个转发单元以外，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，转发逻辑就可以处理相关性并继续执行程序了（如果没有采用转发，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洗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[⊖]。

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 的话，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 4-60 给出了冒险检测单元和转发单元的流水线连接。和前面的介绍一样，转发单元控制 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的话，CD 中的 4.12 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。

⊖ 空指令（nop）：一种不进行任何操作或不改变任何状态的指令。

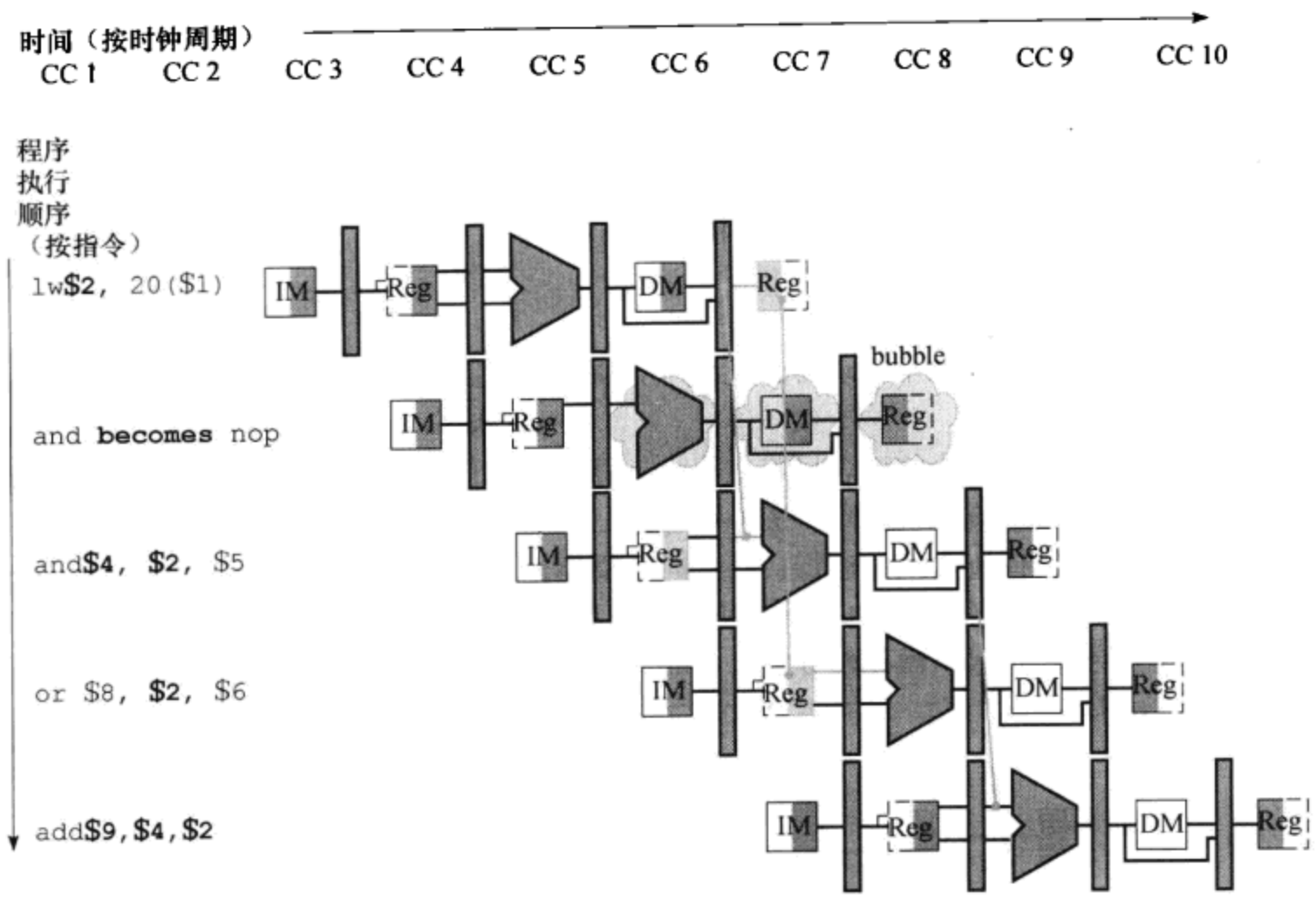


图 4-59 在流水线中插入阻塞的方法

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，所有的相关性沿时间前进，冒险不再发生。

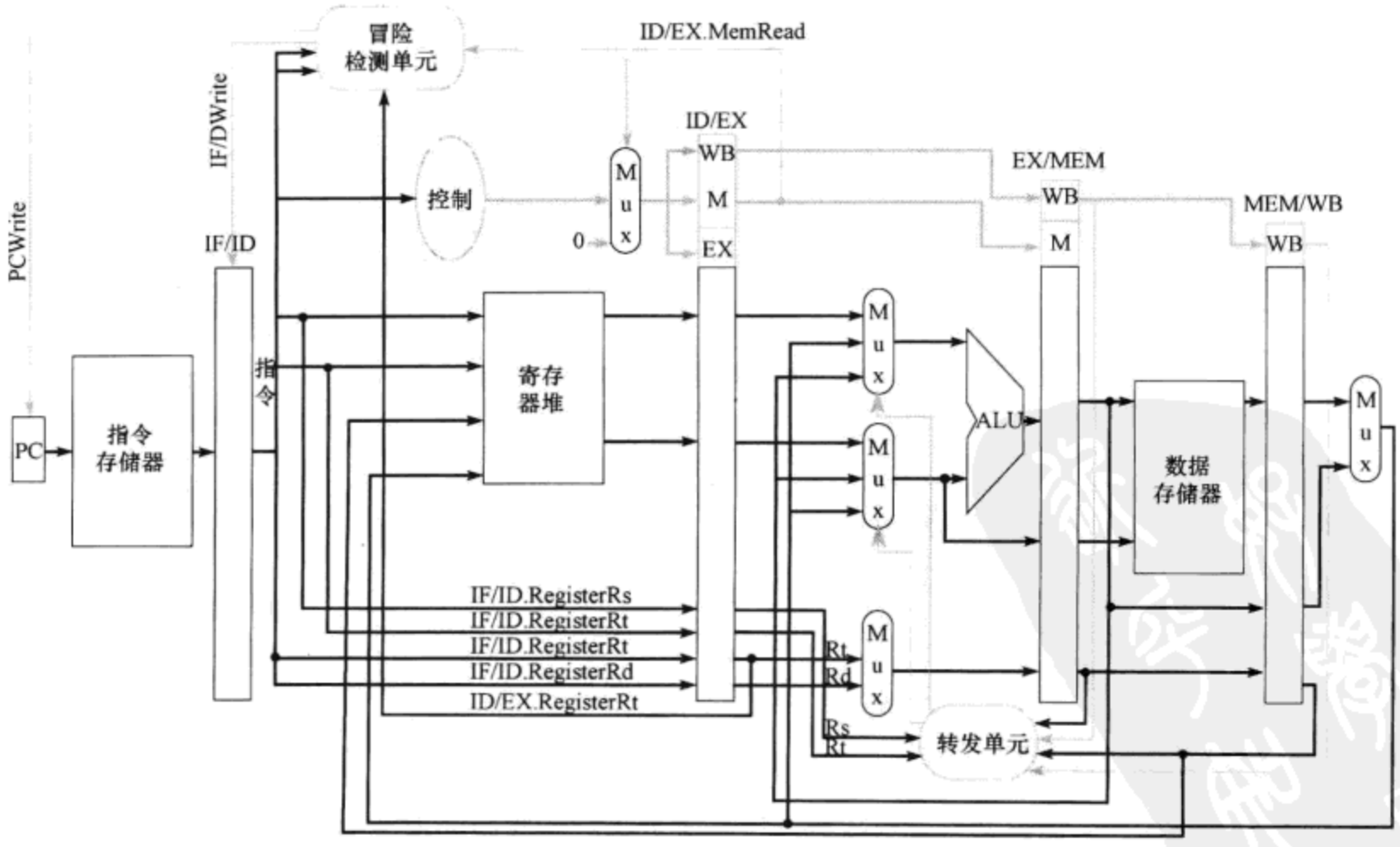


图 4-60 流水线控制概述，其中包括两个转发多选器、一个冒险检测单元和一个转发单元

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），但本图说明了转发和阻塞的基本硬件支持。

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

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做控制冒险（control hazard）或分支冒险（branch hazard）。

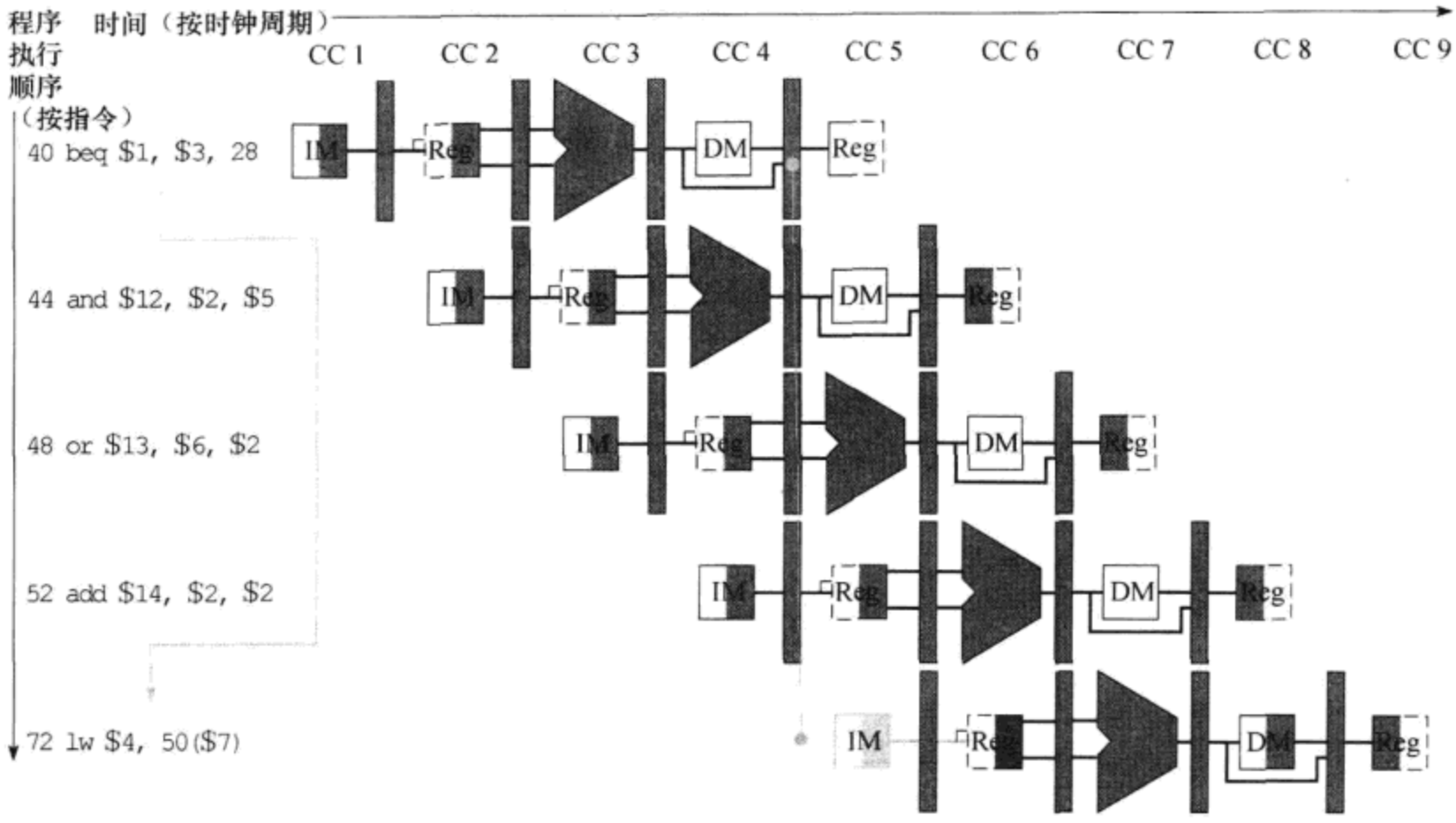


图4-61 分支指令对流水线的影响

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。如果不加干涉的话，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。）

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，而且与采用转发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分支冒险。因此，这一节关于控制冒险的讨论要比前一节的数据冒险要短得多。本节将介绍两种解决控制冒险的方案，并进行了优化。

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。如果分支发生的话，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

令的代价很小的话，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**[⊖]。

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，还需要额外的转发和冒险检测硬件，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），我们需要转发结果至ID级进行相等检测。这里有两个比较复杂的因素：

1）在ID级指令译码后，决定是否需要将所需数据转发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。转发分支指令的操作数以前是由ALU转发单元来完成的，但ID级相等检测单元的引入需要一个新的转发单元。必须注意的是，需要转发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。举例来说，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。下面的例题对转发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。清空寄存器的结果是将预取到的指令转变成为空指令。

举例 **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
```

⊖ 清除（flush）：因发生了意外而丢弃流水线中的指令。

```
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上只有一个流水线气泡。

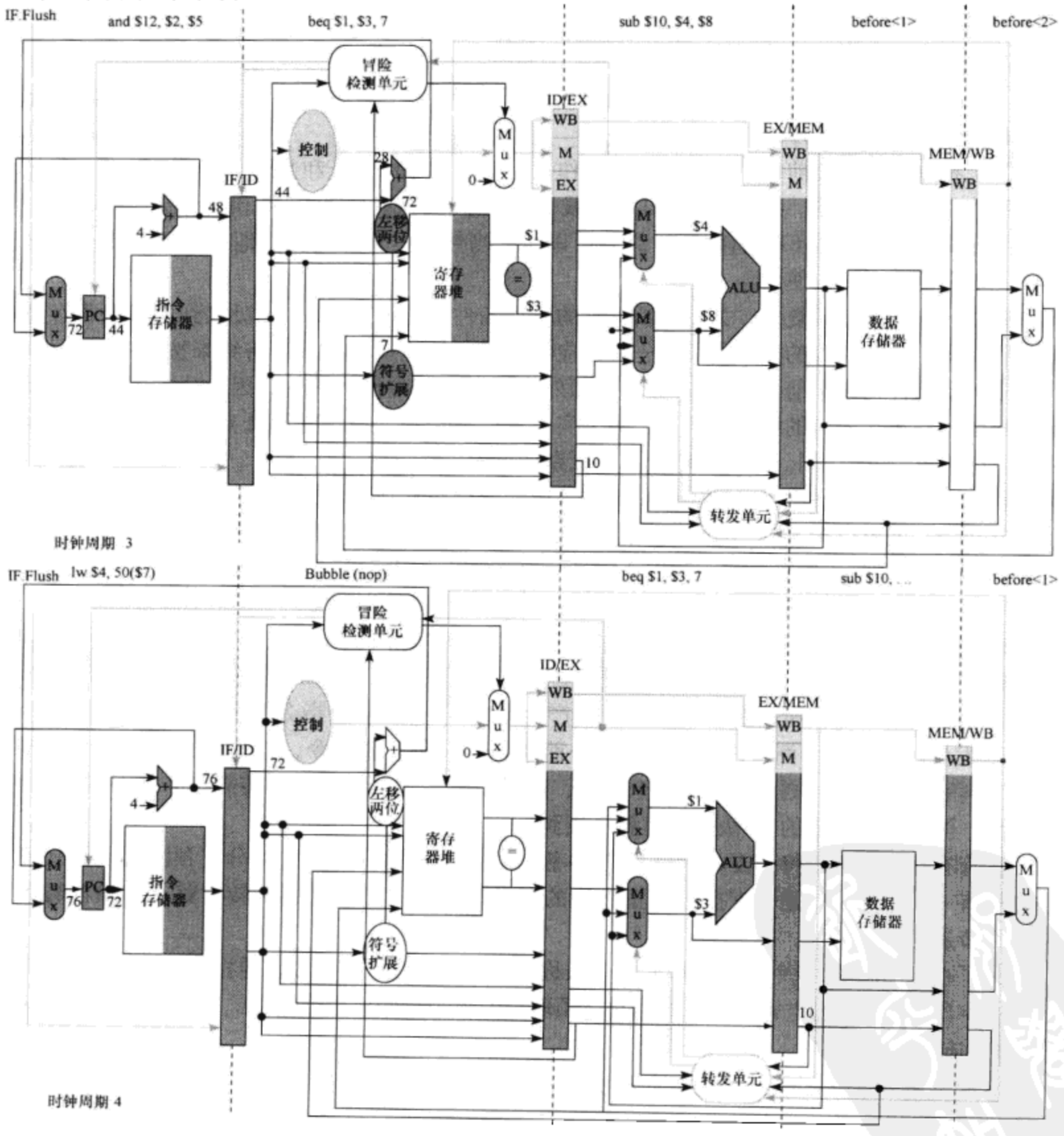


图 4-62 在第三个时钟周期 ID 级确定分支发生，因此地址 72 被选为下一个 PC 地址，同时将为下一个时钟周期预取的指令置为 0

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）。

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**[⊖]。

这种策略的一种实现方法就是采用**分支预测缓存**[⊜]或分支历史记录表。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。

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，而且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二次（而不是一次）预测错误。下面的例子说明了这种情况。

举例 **循环与预测**

让我们看一个循环分支，它在一行代码上的分支发生了九次，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九次，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，八次正确预测）。

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，再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。

精解：如4.5节所述，在五级流水线中，通过重新定义分支，我们可以将控制冒险转化为一种可用的特性。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。

⊖ 动态分支预测（dynamic branch prediction）：根据运行信息在运行中进行分支预测。

⊜ 分支预测缓存（branch prediction buffer）：也称为分支历史记录表（branch history table）。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。

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**分支延迟时间片**[⊖]。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片的方法。

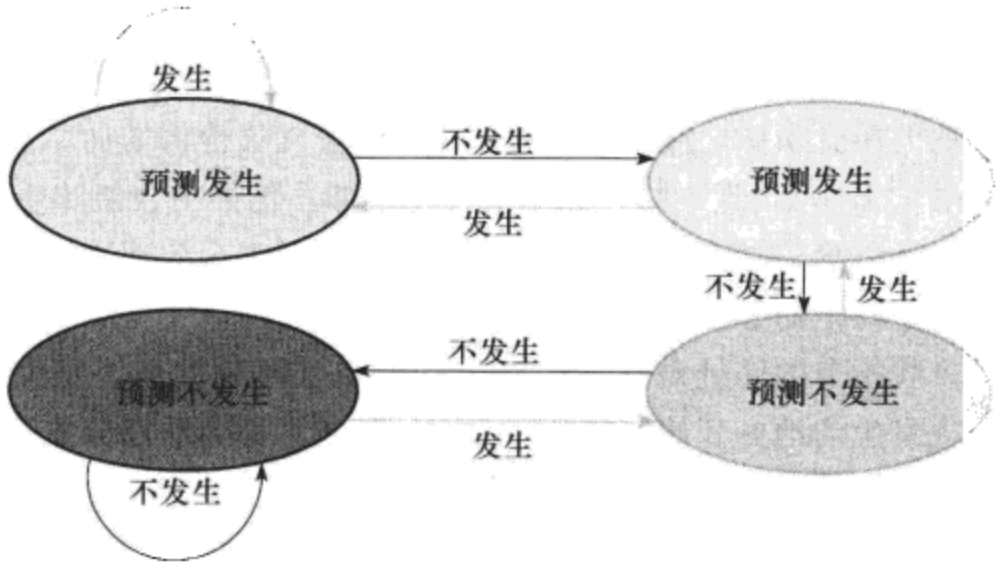
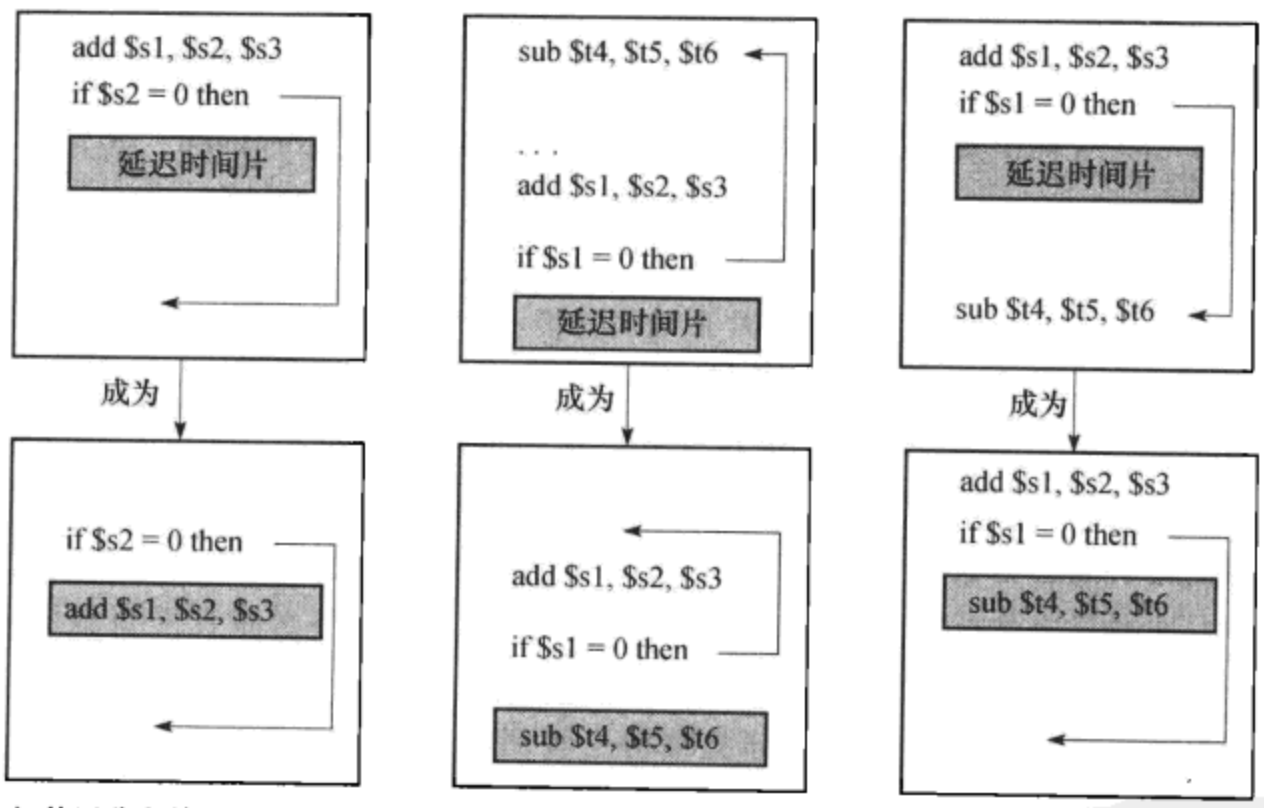


图 4-63 两位预测位机制的状态图

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四种状态。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。



a）使用分支前的指令填充　　b）以分支目标指令填充　　c）以分支不发生时的下一条指令填充

图 4-64 分支延迟时间片的调度

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 `$1`）移入分支延迟时间片。方案 b 中的分支延迟时间片是按照分支目标地址调度的。由于目标指令可以通过其他路径访问到，通常需要将它们进行复制。当分支发生的可能性比较大时，一般选择方案 b，如循环分支。最后，也可能采用方案 c 预测分支不发生的下一条指令进行调度。为了使方案 b 和方案 c 中的优化合法，sub 指令必须在分支预测错误时也能“正常”执行。“正常”意味着虽然有些工作是多余的，但程序依然能够正确执行。例如，当分支预测错误且 `$t4` 是未被使用的临时寄存器时，就是这种情况。

⊖ 分支延迟时间片（branch delay slot）：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。

延迟分支调度的限制在于：①对能够被调度到分支延迟时间片中的指令的限制；②在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，单芯片上晶体管数目的增加使动态预测的成本相对更低。

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**[⊖]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**[⊜]。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**竞赛预测器**[⊜]。竞赛预测器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。目前竞赛预测器的预测是最准确的。典型的竞赛预测器对每个分支地址有两个预测：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，其操作类似于一位或两位的预测器。一些最新的微处理器使用了这种预测器。

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的值为非零的话，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。

ARM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、转发路径、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。

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？

A. 分支发生概率为5%。

B. 分支发生概率为95%。

C. 分支发生概率为70%。

⊖ 分支目标缓存（branch target buffer）：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（correlating predictor）：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。

⊜ 竞赛预测器（tournament branch predictor）：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。

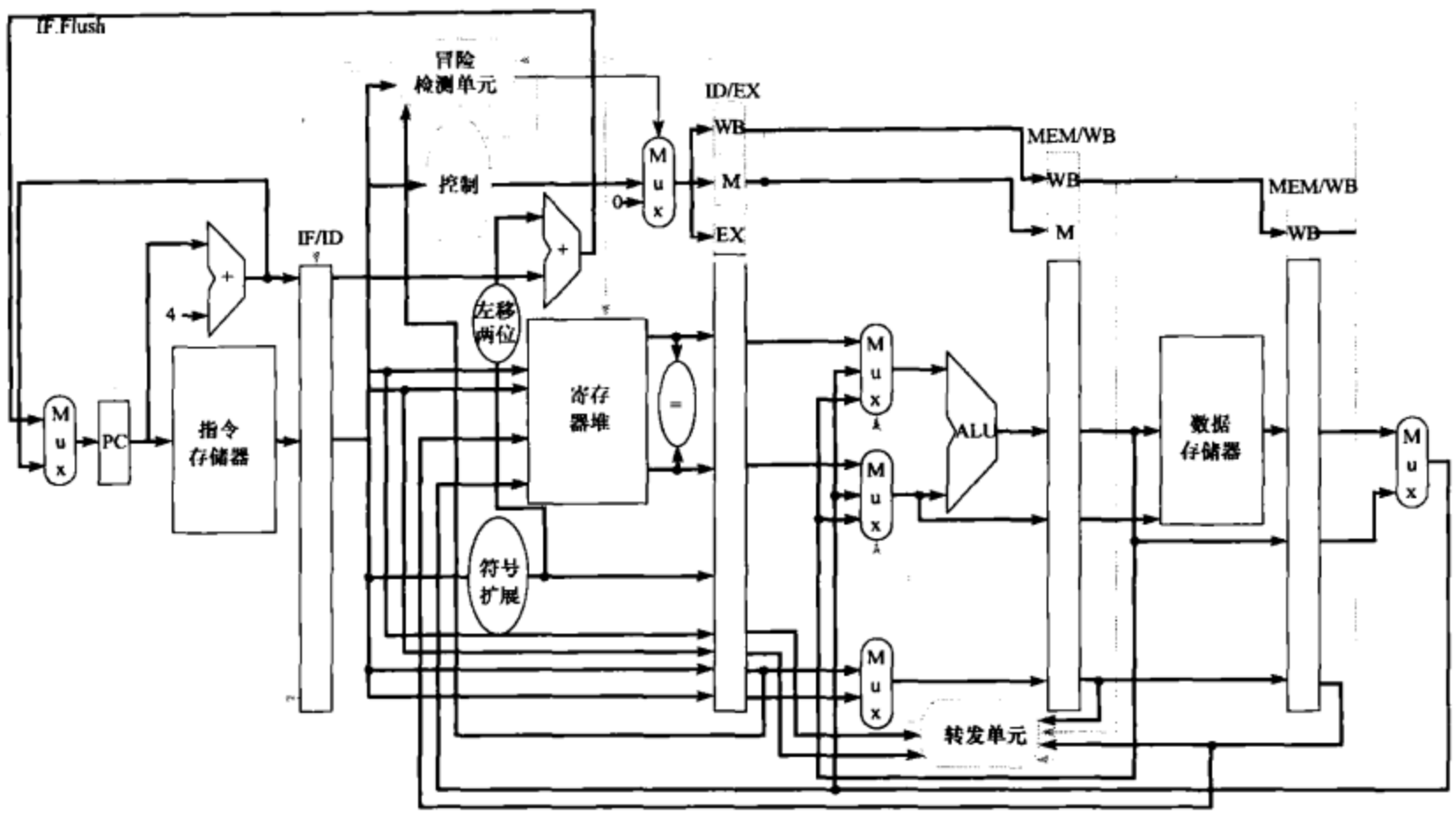


图 4-65 本章最终的数据通路与控制

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。

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**[⊖]和**中断**[⊖]——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 6 章中我们将看到，它们也可用于 I/O 部件与处理器的通信。

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。下面五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

⊖ 异常（exception）：也称为中断（interrupt），指打断程序正常执行的突发事件，例如检测溢出等。

⊖ 中断（interrupt）：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。

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章讨论存储层次和在第6章讨论I/O时，我们将讨论这个话题，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 异常在MIPS体系结构中的处理

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（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。

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**[⊜]。在向量中断中，控制权被转移到由异常原因决定的地址处。（操作系统通过异常向量地址得知异常原因。）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：

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，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），需要给数据通路加上两个寄存器：

- 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。假定使用一个五位的域对前面两种异常原因进行编码：未定义指令=10，数据溢出=12。

⊖ 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

⊜ 向量中断（*vectored interrupt*）：由异常原因决定中断控制转移地址的中断。

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。

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 $8000\ 0180_{16}$ 传递到 PC。图 4-66 具体描述了这种变化。

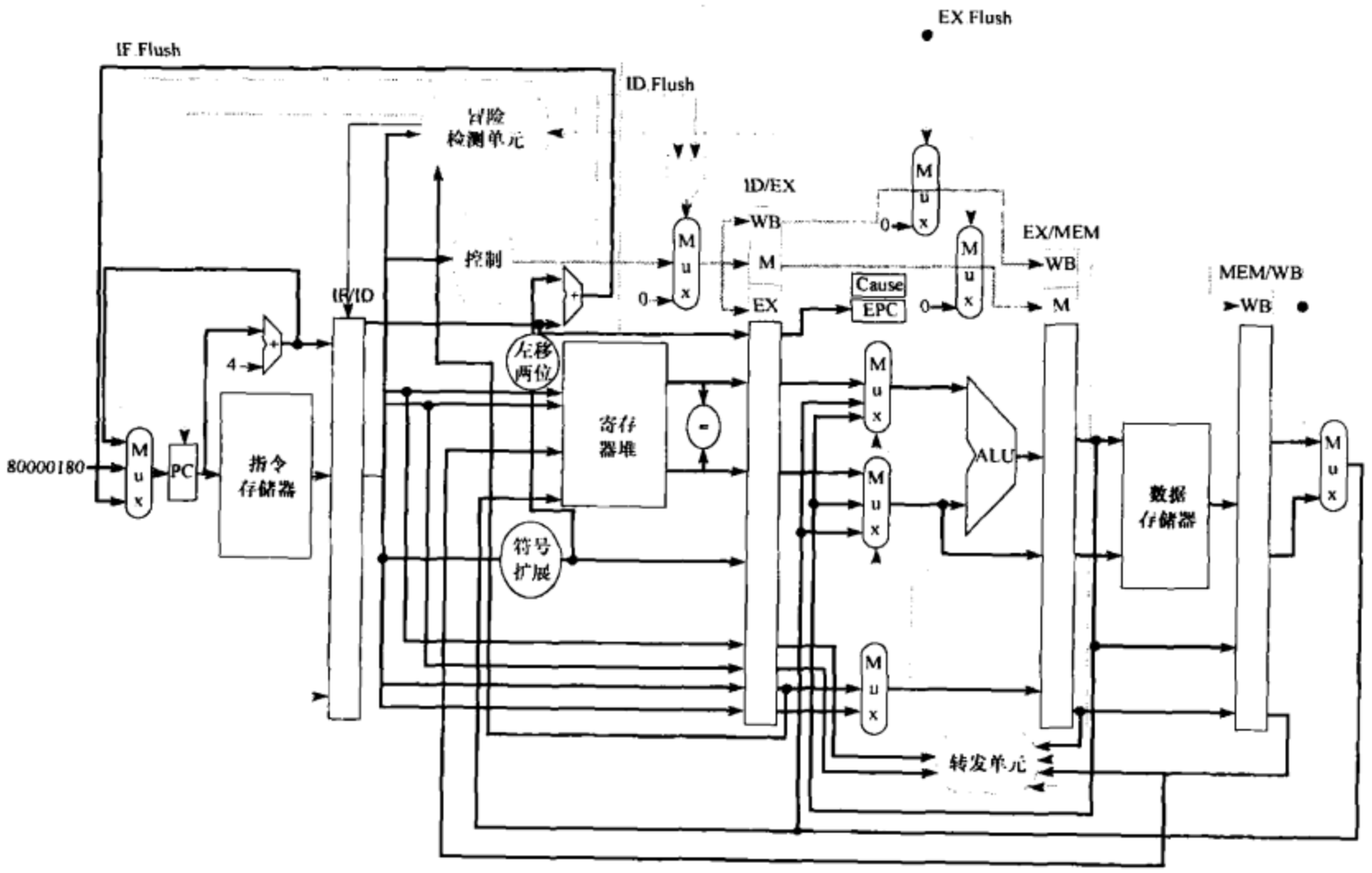


图 4-66 处理异常的数据通路与控制

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。

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。

举例 **流水线处理器中的异常**

给出以下指令序列：

```
40₁₆  sub  $11,$2,$4
44₁₆  and  $12,$2,$5
48₁₆  or  $13,$2,$6
4C₁₆  add  $1,$2,$1
50₁₆  slt  $15,$6,$7
54₁₆  lw  $16,50($7)
……
```

假定异常处理程序的开始部分如下：

```
80000180₁₆  sw  $25,1000($0)
80000184₁₆  sw  $26,1004($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（$4C_{16}+4=50_{16}$）。

在前面我们曾提到五个异常的例子，在第 5 章和第 6 章我们还会看到其他的例子。任何时钟周期流水线中都有五条活动的指令，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描述异常发生的原因，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。

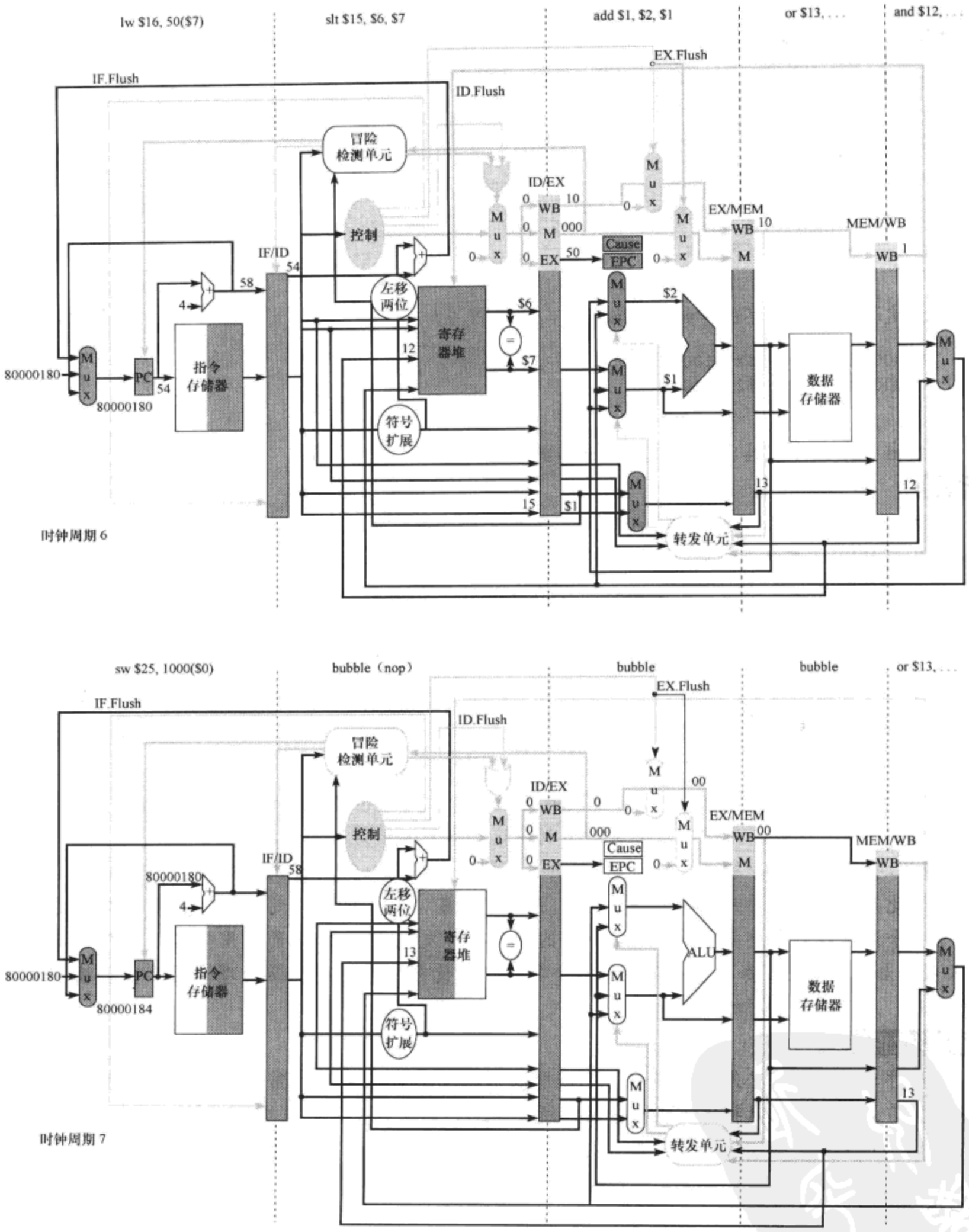


图 4-67　add 指令算术溢出导致的异常

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。

精解：在流水线处理器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**[⊖]或者**非精确异常**。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**[⊜]或**精确异常**（我们将在第 5 章中看到，原因之一是为了支持虚拟存储器）。

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？

```
A. add $1, $2, $1          #算术溢出
B. XXX $1, $2, $1          #未定义指令
C. sub $1, $2, $1          #硬件错误
```

4.10 并行和高级指令级并行

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，可以参考我们的另一本教材：《Computer Architecture：A Quantitative Approach》第 4 版。本节大约十几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**（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。

另一种方法是复制处理器内部部件的数量，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**[⊞]。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。还需要雇佣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。

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。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 4 GHz 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。然而，一般来说对于能同时执行的指令，肯定是存在很多约束的。

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。由于分工方法不同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），

⊖ 非精确中断（imprecise interrupt）：也称为非精确异常（imprecise exception）。流水线处理器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（precise interrupt）：也称为精确异常（precise exception）。流水线处理器中的中断或异常与导致中断或异常的指令精确地关联。

⊝ 指令级并行（instruction-level parallelism）：指令间的并行性。

⊞ 多发射（multiple issue）：一种单时钟周期内发射多条指令的机制。

所以这两种方式有时也被称为**静态多发射**[㊀]和**动态多发射**[㊁]。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**[㊂]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。

4.10.1 推测的概念

推测是一种寻找和挖掘更多 ILP 的最重要的方法。**推测**[㊃]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。推测技术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。实现这种回滚能力增加了额外的复杂性。

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本不存在的异常。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。

㊀ 静态多发射（static multiple issue）：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作出的。
㊁ 动态多发射（dynamic multiple issue）：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作出的。
㊂ 发射槽（issue slot）：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。
㊃ 推测（speculation）：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**[⊖]。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，实际上也确实可以这么理解。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**（VLIW）[⊖]。

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			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

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。特别是使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。

⊖ 发射包（issue packet）：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（Very Long Instruction Word，VLIW）：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域。

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。

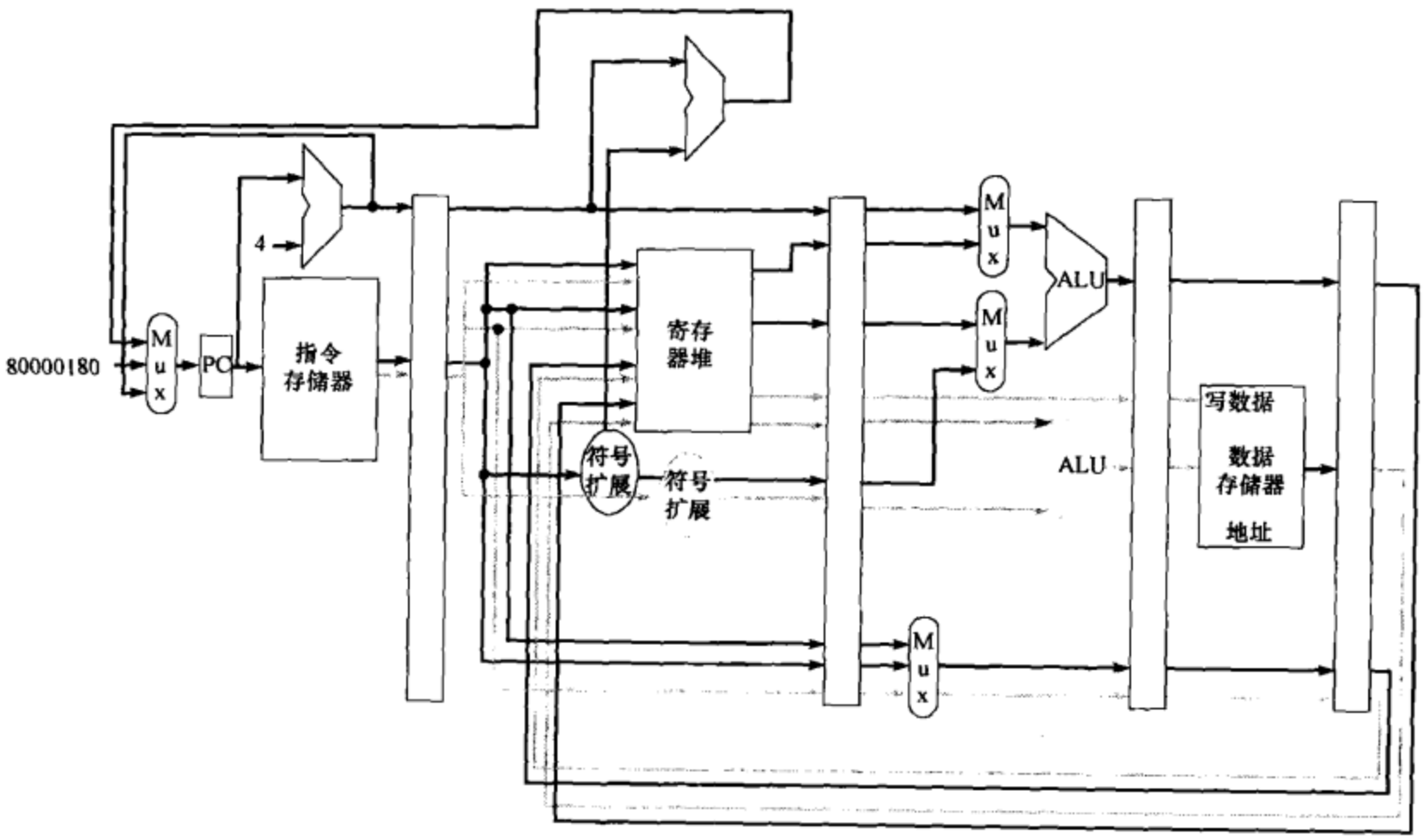


图 4-69 一个静态双发射的数据通路

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。

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**[⊖]，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，原本在简单的一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。

举例 **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
```

⊖ 使用延迟（use latency）：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。

重排序该指令序列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**[⊖]。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）。

举例 **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

⊖ 循环展开（loop unrolling）：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。

存器重命名[㊀]，目的是为了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 $t0 展开的代码是什么样子的。在指令 t0,4($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 寄存器的使用的话，这些指令应该是完全无关的，即一个指令对与下一个指令对之间没有任何数据流动。这就是**反相关**[㊁]，也被称为名字相关，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。

在循环展开的过程中重命名寄存器允许编译器随后移动这些无关的指令以更好地调度。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接近两倍，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四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**[㊂]。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重新编译。

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**[㊃]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。如果内存很慢的话（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。

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 2008 年的高端处理

㊀ 寄存器重命名（register renaming）：由编译器或硬件对寄存器进行重命名以消除反相关。

㊁ 反相关（antidependence）：也被称为名字相关（name dependence），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。

㊂ 超标量（superscalar）：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。

㊃ 动态流水线调度（dynamic pipeline scheduling）：对指令进行重排序以避免阻塞的硬件支持。

器中有10个或更多）和一个**提交单元**㊀。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**㊁），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。结果被得到后，它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**㊂，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转发逻辑。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。

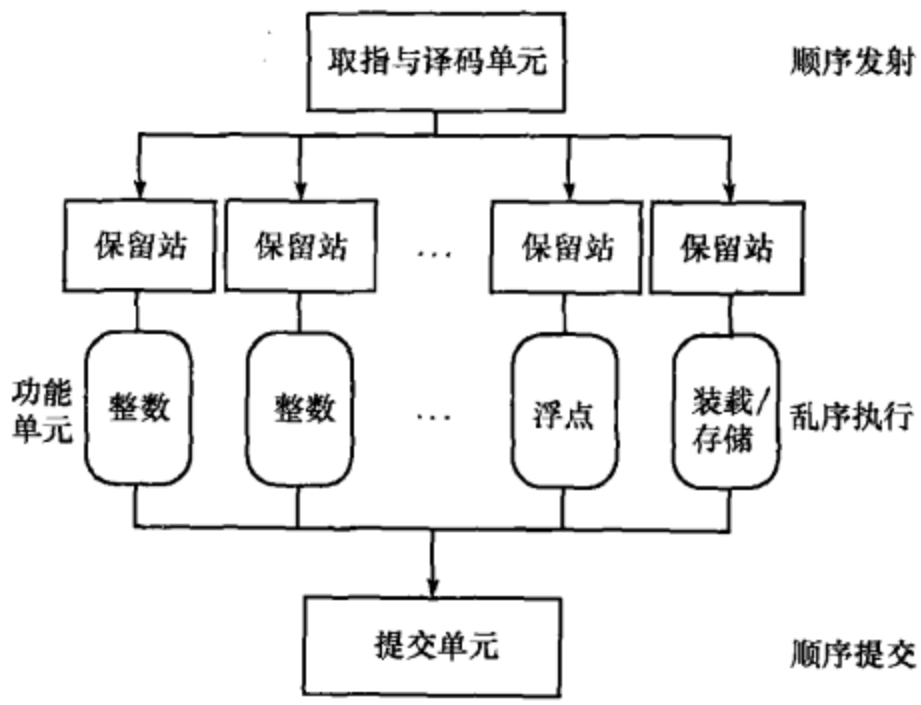


图4-72 动态调度流水线中的三个主要单元
最后一个更新状态的步骤也被称为退休或毕业。

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。

2）如果操作数不在寄存器或重排序缓冲区中，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**㊃，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**㊄。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。

㊀ 提交单元（commit unit）：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。

㊁ 保留站（*reservation station*）：功能单元的缓冲区，用来保存操作数和操作。

㊂ 重排序缓冲区（reorder buffer）：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。

㊃ 乱序执行（out-of-order execution）：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。

㊄ 顺序提交（in-order commit）：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 AMD Opteron X4（Barcelona）处理器的动态调度流水线设计与推测机制。

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 缺失会导致不可预测的阻塞（参见第 5 章）。动态调度使得处理器可以调度并执行其他无关的指令以掩盖阻塞。

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，由于流水线延时和发射宽度根据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时存在着密切关系。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。它还会影响编译器进行寄存器重命名的过程。动态调度使得硬件将这些细节屏藏起来。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。同样的，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。

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，往往有更大的存在隐式相关的可能。反之，数组访问由于有更大的规则性使得编译器可以推测出没有相关存在的情况。同样的，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，存储系统中的缺失同样会使流水线难以满负荷运转（这是第 5 章的主题）。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。

功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，但是在同样的功耗下却能得到更高的性能。所以当设计的约束更多来自功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。

微处理器	年份	时钟频率	流水线级数	发射宽度	乱序/推测执行	每芯片核数	功耗	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	1	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Sun UltraSPARC III	2003	1950 MHz	14	4	No	1	90	W
Sun UltraSPARC T1 (Niagara)	2005	1200 MHz	6	1	No	8	70	W

图 4-73 Intel 和 Sun 公司一些微处理器的指标

其中，Pentium 4 的流水线级数没有包括提交级，如果加上提交级的话，Pentium 4 的流水线级数会更深一些。

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拷贝直到更新该寄存器的指令不再是靠推测得出的。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中（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 操作不依赖于预测的分支时写内存。

精解：非阻塞 cache（*nonblocking cache*）在 cache 访问缺失时能够继续提供 cache 访问服务（参见第 5 章），它有利于存取存储器。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。

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。

A. 分支预测
B. 多发射
C. 超长指令字（VLIW）
D. 超标量
E. 动态调度
F. 乱序执行
G. 推测机制
H. 重排序缓冲区
I. 寄存器重命名

4.11 实例：AMD Opteron X4（Barcelona）流水线

如同绝大多数的现代微处理器一样，x86 微处理器也使用了成熟的流水线技术。如第 2 章中

所述，这些处理器依然面临实现复杂的 x86 指令集的挑战。AMD 和 Intel 都将取到的指令在内部翻译成类 MIPS 指令，AMD 称之为 RISC 操作（RISC operation，Rops），而 Intel 称之为微操作（microoperation）。在 AMD Opteron X4（Barcelona）中，这些 RISC 操作被一个复杂的动态调度、推测流水线执行，并能维持每个时钟周期执行 3 个 RISC 操作的速度。本节关注的就是这个 RISC 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**[⊖]来描述处理器内部体系结构的细节。图 4-74 给出了 X4 的微体系结构，我们重点关注用来执行 RISC 操作的结构。

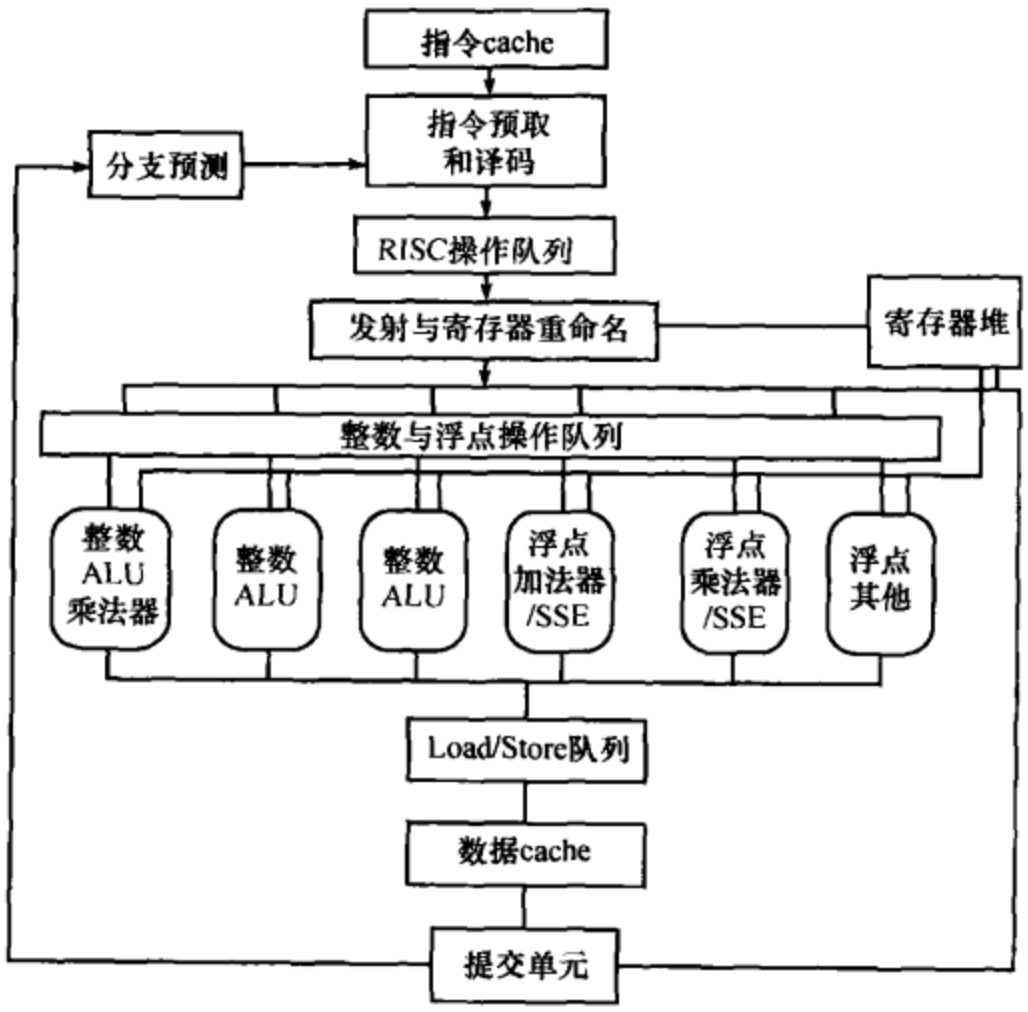


图 4-74 AMD Opteron X4 的微体系结构

延长的队列允许最多 106 条 RISC 操作处于未完成状态：包括 24 个整数操作、36 个浮点或 SSE 操作，44 个存/取操作。存取单元实际上分为两部分：第一部分处理整数 ALU 单元的地址计算；第二部分处理实际的存储器引用。在功能单元间有一个扩展的旁路网络。因为流水线是动态而不是静态的，为了使结果产生时能与队列中需要该结果的指令匹配，旁路是通过标记结果和跟踪源操作数完成的。

分析 X4 的另一种方法是观察一条典型的指令执行将经过哪些流水级。图 4-75 给出了流水线的结构和一般在每一步骤中花费的时钟周期数。当然，实际的时钟周期数会因动态调度特性和不同 RISC 操作的需求而有所变化。

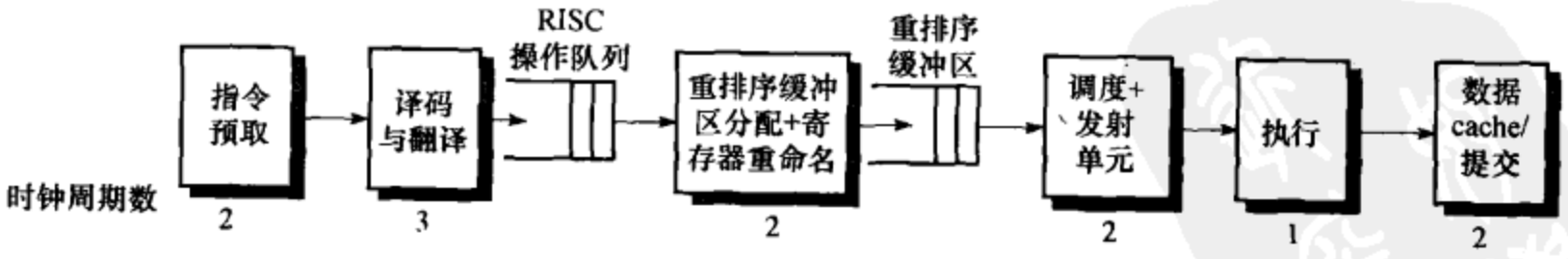


图 4-75 Opteron X4 的流水级，显示了一条典型的指令执行过程及 12 级整数 RISC 操作流水线的主要步骤及其花费的时钟周期数

浮点执行队列的长度为 17 级。图中也给出了用于 RISC 操作等待的主要缓存。

精解： Opteron X4 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

⊖ 微体系结构（microarchitecture）：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。

术显式地将处理器中的**体系结构寄存器**[⊖]（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（在X4中是72个）。X4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，要能指出哪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。

小测验

判断下列表述的正误。

A. Opteron X4多发射流水线直接执行x86指令。

B. X4使用了动态调度但没有使用推测执行。

C. X4微体系结构中的寄存器比x86所要求的更多。

D. X4的流水线级数比早期Pentium 4 Prescott的一半还少（见图4-73）。

理解程序性能

Opteron X4同时使用一个12级的流水线和贪婪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呢？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RISC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数据cache缺失——这会导致阻塞。
- 因为存取存储器（见第5章）导致的处理器阻塞。

4.12 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CD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转发、数据冒险和分支冒险所做的改变。

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指令时流水线的工作细节。

4.13 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Opteron X4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

⊖ 体系结构寄存器（architectural register）：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。

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。

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和精确异常的实现复杂化。这个问题已经解决，最初在20世纪80年代后期的DEC VAX 8500中，采用了微流水线的方案，正如今天Opteron X4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。
- 复杂的寻址方法可能引起很多问题。更新寄存器的寻址方法会使冒险的检测复杂化。而需要多次访问存储器的寻址方法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。

4.14 本章小结

智慧十之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**[⊖]）；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平均时钟周期数（CPI）。

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。在软硬件上都使用调试和推测执行，是降低相关带来影响的主要手段。

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7章的主题）。这种趋势也迫使设计者们对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。

⊖ 指令延迟（instruction latency）：执行一条指令所真正花费的时间。

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——内存系统。

4.15 拓展阅读

这一部分放在光盘中，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。

4.16 练习题

习题 4.1

在基本的单周期实现中不同的指令使用不同的硬件单元。根据如下指令回答下列 3 个问题。

	指令	解释
a.	add Rd,Rs,Rt	Reg[Rd] = Reg[Rs] + Reg[Rt]
b.	lw Rt,Offs(Rs)	Reg[Rt] = Mem[Reg[Rs] + Offs]

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？

不同单元有不同的延迟时间。在图 4-2 中有七种主要单元。对一条指令而言，关键路径（产生最长延迟的那条路径）上各单元的延迟时间决定了该指令的最小延迟。假设各单元的延迟时间如下表所示，回答下列 3 个问题。

	指令存储器	加	多选器	ALU	寄存器堆	数据存储器	控制
a.	400 ps	100 ps	30 ps	120 ps	200 ps	350 ps	100 ps
b.	500 ps	150 ps	100 ps	180 ps	220 ps	1000 ps	65 ps

4.1.4 [5] <4.1> 对一条 MIPS 的与指令（AND）而言，关键路径是什么？

4.1.5 [5] <4.1> 对一条 MIPS 的装载指令（LD）而言，关键路径是什么？

4.1.6 [10] <4.1> 对一条 MIPS 的相等则分支指令（BEQ）而言，关键路径是什么？

习题 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复杂度。对下表中的新指令而言，试回答下列 3 个问题。

	指令	解释
a.	add3 Rd,Rs,Rt,Rx	Reg[Rd] = Reg[Rs] + Reg[Rt] + Reg[Rx]
b.	sll Rt,Rd,Shift	Reg[Rd] = Reg[Rt] << Shift 左移

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？

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？

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、数据寄存器和控制单元的延迟分别为

400 ps、100 ps、30 ps、120 ps、200 ps、350 ps 和 100 ps，相应的成本分别为 1000、30、10、100、200、2000 和 500。试根据表中的改进分别回答下列问题。

	改进	延迟	成本	优势
a.	更快的加法器	加法单元 – 20 ps	每个加法单元 +20	把已有的加法器用更快的加法器替代
b.	更大的寄存器堆	寄存器堆 + 100 ps	寄存器堆 +200	需要更少的 load 和 store 指令。这将导致指令数减少 5%

4.2.4 [10] <4.1> 改进前后的时钟周期分别是多少？

4.2.5 [10] <4.1> 改进后将获得多大的加速比？

4.2.6 [10] <4.1> 比较改进前后的性价比。

习题 4.3

根据下表中的逻辑单元分别回答下列问题。

	逻辑单元
a.	含 4 个 8 位字的小指令存储器
b.	含 2 个 8 位寄存器的小寄存器堆

4.3.1 [5] <4.1，4.2> 这个逻辑单元只包含组合逻辑还是只包含寄存器？还是两者都包含？

4.3.2 [20] <4.1，4.2> 仅使用与门、或门、非门和 D 触发器实现这个逻辑单元。

4.3.3 [10] <4.1，4.2> 当用的与门和或门必须都是二输入时，重做习题 4.3.2。

数字逻辑的延迟和成本取决于构成它的基本单元（门）。根据下表的两种情况分别回答下列问题。

	非门		二输入与门/二输入或门		与/或门每增加一个输入		D 触发器	
	延迟	成本	延迟	成本	延迟	成本	延迟	成本
a.	20 ps	1	30 ps	2	+0 ps	+1	40 ps	6
b.	50 ps	1	100 ps	2	+40 ps	+1	160 ps	2

4.3.4 [5] <4.1，4.2> 习题 4.3.2 中你的实现延迟是多少？

4.3.5 [5] <4.1，4.2> 习题 4.3.2 中你的实现成本是多少？

4.3.6 [20] <4.1，4.2> 试着重新设计以最小化延迟，再重新设计以最小化成本，最后比较这两种设计。

习题 4.4

当用数字电路实现逻辑表达式时，经常需要使用可用的逻辑门实现不可用的逻辑门的功能。试根据下表的两种情况分别回答下列问题。

	控制信号 1	控制信号 2
a.	(((A OR B) OR C) OR (A AND C)) OR (A AND B)	(A OR B) OR C
b.	(((A OR B) XOR B) OR (A OR C)) OR (A AND B)	A AND B

4.4.1 [5] <4.2> 使用非门和二输入与门、或门和异或门实现控制信号 1。注意不要试着“优化”原表达式。

4.4.2 [10] 假设所有门的延迟相同。习题 4.4.1 中你的电路关键路径有多长（几个门）？

4.4.3 [10] <4.2> 当需要实现多个逻辑表达式时，通过在多个逻辑表达式中使用同一信号来减小实现代价是有可能的。重做习题4.4.1，但这次要实现控制信号1和控制信号2，试着在实现两个表达式时共享某些信号。

假设可以使用非门、二输入与、二输入或和二输入异或门。试根据下表的两种延迟和成本情况分别回答下列问题。

	非门		二输入与门		二输入或门		二输入异或门	
	延迟	成本	延迟	成本	延迟	成本	延迟	成本
a.	20 ps	1	30 ps	2	34 ps	3	40 ps	6
b.	50 ps	1	100 ps	2	120 ps	2	150 ps	2

4.4.4 [10] <4.2> 习题4.4.3中你的电路关键路径长度为多少？

4.4.5 [10] <4.2> 习题4.4.3中你的电路成本为多少？

4.4.6 [10] <4.2> 习题4.4.3中同时实现两个控制信号和单独实现相比，能节约多少成本？

习题4.5

这个习题的目的是帮助读者熟悉时序逻辑电路的设计和操作。试根据下表两个ALU操作回答以下两个问题。

	ALU操作
a.	加1（X+1）
b.	左移2位（X<<2）

4.5.1 [20] <4.2> 设计一个电路串行实现该操作，该电路的数据输入与输出均为1位，从最低有效位开始。在这个串行实现中，电路一位一位地处理输入，也一位一位地生成输出。例如，一个串行的与电路就是一个简单的与门。在第N个周期输入操作数的第N位，生成结果的第N位。除了数据输入，该电路还有一个时钟输入（Clk）和一个“开始”信号输入，“开始”信号为1时，表示当前为该操作的第1个周期。在设计中只可以使用D触发器、非门、与门、或门和异或门。

4.5.2 [20] <4.2> 重做习题4.5.1，这次要求一次对两位进行操作。

假设下表中的数字逻辑单元可用，试根据两种不同的延迟和成本情况回答以下四个问题。

	非门		与门		或门		异或门		D触发器	
	延迟	成本	延迟	成本	延迟	成本	延迟	成本	延迟	成本
a.	20 ps	1	30 ps	2	20 ps	2	30 ps	4	40 ps	6
b.	40 ps	1	50 ps	2	60 ps	2	80 ps	3	80 ps	12

D触发器的延迟指的是建立时间。D触发器的数据输入必须在时钟沿到来之前就准备好要存入D触发器的值。

4.5.3 [10] <4.2> 习题4.5.1中你的设计的时钟周期是多少？完成一次32位的操作需要多长时间？

4.5.4 [10] <4.2> 习题4.5.2中你的设计的时钟周期是多少？对一次32位的操作而言，这个设计比习题4.5.1中的设计要快多少？

4.5.5 [10] <4.2> 分别计算习题4.5.1和习题4.5.2中设计的成本。

4.5.6 [5] <4.2> 分别计算习题4.5.1和习题4.5.2中设计的性价比。其中性能可以用完成一次32位操作所需时间的倒数来计算。

习题4.6

下表给出了实现处理器数据通路的逻辑单元延迟。试根据下表的两种情况分别回答下列问题。

	指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
a.	400 ps	100 ps	30 ps	120 ps	200 ps	350 ps	20 ps	2 ps
b.	500 ps	150 ps	100 ps	180 ps	220 ps	1000 ps	90 ps	20 ps

4.6.1 [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？

4.6.2 [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？

4.6.3 [10] <4.3> 重做习题 4.6.2，但这次假设只需处理有条件相对跳转指令。

根据下表的两种数据通路的逻辑单元分别回答下列问题。

	单元
a.	加 4（对 PC）
b.	数据存储器

4.6.4 [10] <4.3> 哪些类型的指令需要该单元？

4.6.5 [20] <4.3> 对哪些类型的指令而言，该单元位于关键路径上？

4.6.6 [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。

习题 4.7

本题讨论数据通路中不同的单元延迟对整个数据通路时钟周期的影响，以及指令如何利用不同的数据通路单元。根据下面的两种延迟情况，分别回答下列问题。

	指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
a.	400 ps	100 ps	30 ps	120 ps	200 ps	350 ps	20 ps	0 ps
b.	500 ps	150 ps	100 ps	180 ps	220 ps	1000 ps	90 ps	20 ps

4.7.1 [10] <4.3> 如果仅需支持 ALU 类指令（如 add、and 等），处理器的时钟周期是多少？

4.7.2 [10] <4.3> 如果仅需支持 lw 指令，时钟周期是多少？

4.7.3 [20] <4.3> 如果必须支持 add、beq、lw 和 sw 指令，时钟周期是多少？

假设各类型指令所占比例如下表所示，试根据下表的两种情况分别回答下列问题。

	add	addi	not	bgq	lw	sw
a.	30%	15%	5%	20%	20%	10%
b.	25%	5%	5%	15%	35%	15%

4.7.4 [10] <4.3> 数据存储器平均用了多少时钟周期？

4.7.5 [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？

4.7.6 [10] <4.3> 如果可以将数据通路上某个单元的延迟减少 10%，应该减少哪个单元的延迟？改进后整个处理器的加速比是多少？

习题 4.8

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，某些线上的信号为常值（如电源线），该线附近的线也被固定为 0（stuck-at-0）或 1（stuck-at-1）。试根据下表的两种缺陷（信号来自图 4-24）分别回答下列问题。

	有问题的信号
a.	指令存储器，输出指令第7位
b.	控制单元，输出信号 MemtoReg

4.8.1 [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？

4.8.2 [10] <4.3，4.4>重做习题4.8.1，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。

4.8.3 [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。

根据下表的缺陷分别回答下列问题。

	缺陷
a.	固定为1
b.	如果指令的第31～26位全为0，则固定为0，否则无缺陷

4.8.4 [10] <4.3，4.4>重做习题4.8.1，这次检测控制信号 MemRead 是否存在该缺陷？

4.8.5 [10] <4.3，4.4>重做习题4.8.1，这次检测控制信号 Jump 是否存在该缺陷？

4.8.6 [40] <4.3，4.4>使用习题4.8.1中描述的测试方案，可以一次对几个不同的信号进行测试，但一般来说不可能同时测试到所有信号。试着设计一系列方案对所有多选器输出的该缺陷进行测试（五个多选器输出的每一位都要测试到）。尽量使用较少的测试方案。

习题4.9

本习题讨论特定指令在单周期数据通路中的操作。根据下表中的 MIPS 指令分别回答下列问题。

	指令
a.	`lw $1,40($6)`
b.	`Label: bne $1,$2,Label`

4.9.1 [10] <4.4>该指令字的值是多少？

4.9.2 [10] <4.4>提供给寄存器堆“读寄存器1”端口的寄存器号是多少？该寄存器真的被读了吗？对于“读寄存器2”呢？

4.9.3 [10] <4.4>提供给寄存器堆“写寄存器”端口的寄存器号是多少？该寄存器真的被写了吗？

不同的指令需要设置数据通路上不同的控制信号。根据下表的两种控制信号情况分别回答下列问题（参考图4-24）。

	控制信号1	控制信号2
a.	RegDst	MemRead
b.	RegWrite	MemRead

4.9.4 [20] <4.4>对该指令而言，这两个控制信号的值应该是多少？

4.9.5 [20] <4.4>对图4-24中的数据通路而言，画出控制单元中实现第一个信号的部分电路图。假设我们仅需支持 lw、sw、beq、add 和 j（jump）指令。

4.9.6 [20] <4.4>重做习题4.9.5，这次两个信号都要实现。

习题 4.10

本题讨论处理器时钟周期与控制单元设计之间的相互影响。根据下表的两种数据通路单元延迟情况分别回答下列问题。

	指令存储器	加法器	多选器	ALU	寄存器堆	数据寄存器	符号扩展	左移两位	ALU 控制
a.	400 ps	100 ps	30 ps	120 ps	200 ps	350 ps	20 ps	0 ps	50 ps
b.	500 ps	150 ps	100 ps	180 ps	220 ps	1000 ps	90 ps	20 ps	55 ps

4.10.1 [10] <4.2, 4.4> 为了避免增加图 4-24 中数据通路的关键路径长度，留给控制单元产生 MemWrite 信号的时间有多少？

4.10.2 [20] <4.2, 4.4> 图 4-24 中哪个控制信号最不关键，控制单元需要在多长时间内产生该信号以避免其成为关键路径？

4.10.3 [20] <4.2, 4.4> 图 4-24 中哪个控制信号最关键，控制单元需要在多长时间内产生该信号以避免其成为关键路径？

假设控制单元产生控制信号的时间如下表所示，试根据表中的两种情况回答下列问题。

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
a.	720 ps	730 ps	600 ps	400 ps	700 ps	200 ps	710 ps	200 ps	800 ps
b.	1600 ps	1600 ps	1400 ps	500 ps	1400 ps	400 ps	1500 ps	400 ps	1700 ps

4.10.4 [20] <4.4> 处理器的时钟周期为多少？

4.10.5 [20] <4.4> 如果你可以加速控制信号的产生，但加快一个控制信号 5 ps 的代价是处理器成本增加 1$。那么为了最大化性能你会加速哪个控制信号？这种性能改进的代价是多少钱？

4.10.6 [30] <4.4> 如果一个处理器的成本已经很高，那么我们需要在维持处理器性能的同时降低其成本，而不是像习题 4.10.5 中所作的那样为提高它的性能而买单。如果你可以使用更慢的逻辑来实现对信号的控制，并且单个控制信号每减慢 5 ps，处理器成本就可以节省 1 美元，那么在保持处理器性能的同时，你会减慢哪些控制信号，并且减慢多少来降低成本？

习题 4.11

本题讨论单周期数据通路中指令的执行细节。根据表中的两种指令字情况回答下列问题。

	指令字
a.	10001100010000110000000000010000
b.	00010000001000110000000000001100

4.11.1 [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？

4.11.2 [10] <4.4> 对该指令字而言，ALU 控制单元的输入是什么？

4.11.3 [10] <4.4> 该指令执行后的新 PC 值是什么？在图 4-24 中粗线决定该新 PC 值的路径。

下列问题假设数据存储器中的值是全零并且寄存器堆中的初值如下表所示，试根据表中的两种情况回答下列问题。

	$0	$1	$2	$3	$4	$5	$6	$8	$12	$31
a.	0	1	2	3	−4	5	6	8	1	−32
b.	0	−16	−2	−3	4	−10	−6	−1	8	−4

4.11.4 [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。

4.11.5 [10] <4.4> 对给定的指令字和寄存器堆初值，给出 ALU 和两个加法器数据输出的值。

4.11.6 [10] <4.4> 对给定的指令字和寄存器堆初值，给出寄存器堆所有输入信号的值。

习题 4.12

本习题讨论流水线对处理器时钟周期的影响。表中给出了数据通路中不同阶段延迟的两种情况，试根据这两种情况分别回答下列问题。

	IF	ID	EX	MEM	WB
a.	300 ps	400 ps	350 ps	500 ps	100 ps
b.	200 ps	150 ps	120 ps	190 ps	140 ps

4.12.1 [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？

4.12.2 [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？

4.12.3 [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？

假设处理器执行的指令比例如下表两种情况所示，试根据每种情况分别回答下列问题。

	ALU	beq	lw	sw
a.	50%	25%	15%	10%
b.	30%	25%	30%	15%

4.12.4 [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）？

4.12.5 [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？

4.12.6 [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。

习题 4.13

本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。试根据下表的两种指令序列情况分别回答下列问题。

	指令序列		指令序列

a.	lw $1,40($6)
	add $6, $2, $2
	sw $6,50($1)
		b.	lw $5, -16($5)
			sw $5, -16($5)
			add $5, $5, $5

4.13.1 [10] <4.5> 指出指令序列中存在的相关及其类型。

4.13.2 [10] <4.5> 假设该流水线处理器没有转发，指出指令序列中存在的冒险并加入 nop 指令以消除冒险。

4.13.3 [10] <4.5> 假设该流水线处理器中有充分的转发。指出指令序列中存在的冒险并加入 nop 指令以消除冒险。

根据下表的两种时钟周期情况，分别回答下列问题。

	无转发	充分的转发	仅 ALU 至 ALU 的转发
a.	300 ps	400 ps	360 ps
b.	200 ps	250 ps	220 ps

4.13.4 [10] <4.5> 该指令序列在无转发和有充分的转发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?

4.13.5 [10] <4.5> 如果仅有 ALU 至 ALU 的转发（没有从 MEM 到 EX 的转发），如何加入 nop 指令以消除可能的冒险?

4.13.6 [10] <4.5> 该指令序列在仅有 ALU 至 ALU 的转发时总执行时间分别是多少? 与无转发的情况相比，加速比是多少?

习题 4.14

本习题讨论结构冒险、控制冒险和 ISA 设计如何影响流水线执行。根据下表的两个 MIPS 代码片段分别回答下列问题。

	指令序列		指令序列

a.	`lw $1,40($6)`
	`beq $2,$0,Label ;Assume $2 == $0`
	`sw $6,50($2)`
	`Label: add $2,$3,$4`
	`sw $3,50($4)`
		b.	`lw $5,-16($5)`
			`sw $4,-16($4)`
			`lw $3,-20($4)`
			`beq $2,$0,Label ;Assume $2!= $0`
			`add $5,$1,$4`

4.14.1 [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的话就会发生结构冒险。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?

4.14.2 [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?

4.14.3 [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?

根据下表中不同流水级的两种延迟情况，试分别回答下列问题。

	IF	ID	EX	MEM	WB
a.	100 ps	120 ps	90 ps	130 ps	60 ps
b.	180 ps	100 ps	170 ps	220 ps	60 ps

4.14.4 [10] <4.5> 在给定的流水级延迟下，重做习题 4.14.2，考虑可能的时钟周期变化。如果 EX 级和 MEM 级重叠起来，大部分时候它们可以并行工作。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 20 ps。

4.14.5 [10] <4.5> 在给定的流水级延迟下，重做习题 4.14.3，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 10 ps。

4.14.6 [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 20 ps 而 MEM 级的延迟不变。

习题 4.15

本习题讨论指令集对流水线设计的影响。试根据下表的两条新指令回答下列问题。

a.	bezi(Rs),Label	if Mem[Rs]=0 then PC=PC+Offs
b.	swi Rd,Rs(Rt)	Mem[Rs+Rt]=Rd

4.15.1 [20] <4.5>为了将这条新指令增加到 MIPS 指令集，必须对流水线数据通路做什么改动？

4.15.2 [10] <4.5>需要在习题 4.15.1 的数据通路上增加哪些控制信号？

4.15.3 [20] <4.5，4.13>对新指令的支持是否会引入新的冒险？已有冒险导致的阻塞是否会更加严重？

4.15.4 [10] <4.5，4.13>给出一种能用上新指令的情况和一个能用该指令替代的 MIPS 指令序列。

4.15.5 [10] <4.5，4.11，4.13>假设这条指令已经存在于原指令集中，试解释其在新处理器如 AMD Barcelona 中将如何执行。

本练习的上一个问题假定每条新指令的使用都替换一定数量的原有指令，在给定数量的原有指令中仅会进行一次替换，并且每次新指令执行时，给定数量的额外停顿周期都将计入程序的执行时间。试根据表中的两种情况分别回答下列问题。

	替代的指令数	发生替换的频率	额外的阻塞周期
a.	2	20	1
b.	3	60	0

4.15.6 [10] <4.5>加入新指令后取得的加速比为多少？假设原程序（不含新指令）的 CPI 为 1。

习题 4.16

试根据表中的两条 MIPS 指令分别回答下列问题。

	指令
a.	lw $1,40($6)
b.	add $5,$5,$5

4.16.1 [5] <4.6>指令执行时，两级流水线之间的寄存器中的内容是什么？

4.16.2 [5] <4.6>哪些寄存器是需要读的？实际上读了哪些寄存器？

4.16.3 [5] <4.6>这条指令在 EX 级和 MEM 级分别做了什么？

根据下表的两个循环分别回答下列问题。假设分支被完美地预测（没有因控制冒险导致的阻塞），没有延迟时间槽，而且流水线有完全的转发支持，并且循环在退出前运行了很多次。

	循环		循环

a.	Loop: lw $1,40($6)
	add $5,$5,$8
	add $6,$6,$8
	sw $1,20($5)
	beq $1,$0,Loop
		b.	Loop: add $1,$2,$3
			sw $0,0($1)
			sw $0,4($1)
			add $2,$2,$4
			beq $2,$0,Loop

4.16.4 [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结束。给出这段时间内流水线中的所有指令。

4.16.5 [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？

4.16.6 [10] <4.6>在第三次循环的首条指令被取指时，IF/ID 寄存器中的内容是什么？

习题 4.17

假设流水线处理器执行指令的比例如下表所示，试根据每种情况分别回答下列问题。

	add	beq	lw	sw
a.	50%	25%	15%	10%
b.	30%	15%	35%	20%

4.17.1 [5] <4.6> 假设不发生阻塞且所有条件分支有60%发生，EX级的分支加法器生成的值有多少时间被真正用到（以百分比表示）？

4.17.2 [5] <4.6> 假设不发生阻塞，同一周期用到寄存器堆的三个端口（两个读端口和一个写端口）的时候有多少（以百分比表示）？

4.17.3 [5] <4.6> 假设不发生阻塞，用到数据存储器的时候有多少（以百分比表示）？

图4-33中不同的流水级有不同的延迟，并且流水级间的寄存器也会引入额外的延迟。试根据下表给出的两种流水线延迟情况回答下列问题。

	IF	ID	EX	MEM	WB	流水级间的寄存器
a.	100 ps	120 ps	90 ps	130 ps	60 ps	10 ps
b.	180 ps	100 ps	170 ps	220 ps	60 ps	10 ps

4.17.4 [5] <4.6> 假设没有阻塞，流水线相对于单周期数据通路的加速比是多少？

4.17.5 [10] <4.6> 我们可以把所有存取指令的寻址方式变为仅基于寄存器（不考虑偏移），这样存取存储器操作就可以与ALU操作并行了。在单周期数据通路和流水线数据通路中进行这种改变后，时钟周期分别是多少？假设新的EX/MEM级的延迟等于EX级和MEM级的较大者。

4.17.6 [10] <4.6> 习题4.17.5要求许多现存的 `lw/sw` 指令变换为包含2条指令的指令对。假设有50%的存取指令需要这种变换，那么从五级流水线变成四级流水线（EX和MEM级并行）的加速比是多少？

习题4.18

下表给出了流水线（参见图4-51）中执行的指令、时钟周期、ALU延迟和Mux延迟的两种情况。试根据表中两种情况分别回答下列问题。

	指令	时钟周期	ALU 延迟	多选器延迟
a.	`add $1, $2, $3`	100 ps	80 ps	10 ps
b.	`slt $2, $1, $3`	80 ps	50 ps	20 ps

4.18.1 [10] <4.6> 每个流水级的控制信号值是多少？

4.18.2 [10] <4.6, 4.7> 控制单元需要在多长时间内产生控制信号ALUSrc？与单周期实现进行比较。

4.18.3 对这条指令而言，PCSrc控制信号的值应是多少？这个信号在MEM级中产生（仅使用了一个与门），为什么不在EX级才产生这个信号？

下表给出两个信号的两种情况，试根据每种情况分别回答下列问题。

	信号1	信号2
a.	RegDst	RegWrite
b.	MemRead	RegWrite

4.18.4 [5] <4.6> 这两个控制信号分别是在哪个流水级生成，又在哪个流水级使用的？

4.18.5 [5] <4.6> 对哪些MIPS指令，这两个信号都设置为1？

4.18.6 [10] <4.6> 这两个信号中哪一个沿流水线反向传输？这是一个时间旅行悖论吗？为什么？

习题4.19

本习题讨论流水线处理器中转发的成本/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 2nd”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	EX to1st and 2nd	仅 EX to 2nd	MEM to 1st
a.	10%	10%	5%	25%
b.	15%	5%	10%	20%

4.19.1 [10] <4.7> 如果不使用转发，会有百分之多少的时间周期因为数据冒险阻塞？

4.19.2 [5] <4.7> 如果使用完全的转发（转发所有可以转发的结果），会有百分之多少的时间周期因为数据冒险阻塞？

4.19.3 [10] <4.7> 假设不能提供三输入多选器（对完全的转发是必需的），我们必须确定从 EX/MEM 流水线寄存器转发（转发 1 个周期）还是从 MEM/WB 流水线寄存器转发（转发 2 个周期）更好？哪种方法会产生更少的数据阻塞？

下表给出了各级流水线延迟的两种情况。其中 EX 级给出了不同转发情况下的延迟。试根据两种情况分别回答下列问题。

	IF	ID	EX（无转发）	EX（完全的转发）	EX（仅有来自 EX/MEM 的转发）	EX（仅有来自 MEM/WB 的转发）	MEM	WB
a.	100 ps	50 ps	75 ps	110 ps	100 ps	100 ps	100 ps	60 ps
b.	250 ps	300 ps	200 ps	350 ps	320 ps	310 ps	300 ps	200 ps

4.19.4 [10] <4.7> 对给定的冒险概念和流水级延迟，完全的转发相对于无转发的加速比是多少？

4.19.5 [10] <4.7> 如果加入能消除所有数据冒险的时间旅行转发，其相对于完全转发的加速比是多少？假设在完全转发的基础上加入这个还没发明的时间旅行转发的代价是增加 100 ps 的延迟。

4.19.6 [20] <4.7> 重做习题 4.19.3，这次问哪种方法会产生更小的 CPI。

习题 4.20

试根据下表的两个指令序列分别回答下列问题。

	指令序列		指令序列

a.	`lw $1,40($2)`
	`add $2,$3,$3`
	`add $1,$1,$2`
	`sw $1,20($2)`
		b.	`add $1,$2,$3`
			`sw $2,0($1)`
			`lw $1,4($2)`
			`add $2,$2,$1`

4.20.1 [5] <4.7> 找出指令序列中的数据相关。

4.20.2 [10] <4.7> 分别对有转发和无转发的五级流水线找出指令序列中的冒险。

4.20.3 [10] <4.7> 为减少时钟周期，考虑将 MEM 级划分成两级。对这个六级流水线重做习题 4.20.2。

假设数据存储器中的初值为 0 且寄存器 0～3 号的值如下表所示，试根据两种情况分别回答下列问题。

	$0	$1	$2	$3
a.	0	1	31	1000
b.	0	-2	63	2500

4.20.4 [5] <4.7>第一个转发的值是多少？它覆盖掉的值是多少？

4.20.5 [10] <4.7>假如当我们设计冒险检测单元时忘了实现转发单元，指令序列执行完成后最终的寄存器值是多少？

4.20.6 [10] <4.7>假如忘了实现转发单元（如习题4.20.5描述的设计），该怎么加入nop指令以保证指令序列的正确执行？

习题4.21

本习题讨论转发、冒险检测和指令集设计之间的关系。分别根据下表的两个指令序列回答下列问题。假设其在一个五级流水线上执行。

	指令序列		指令序列

a.	lw $1,40($6)
	add $2,$3,$1
	add $1,$6,$4
	sw $2,20($4)
	and $1,$1,$4
		b.	add $1,$5,$3
			sw $1,0($2)
			lw $1,4($2)
			add $5,$5,$1
			sw $1,0($2)

4.21.1 [5] <4.7>如果没有转发或冒险检测电路，请插入nop指令以保证正确执行。

4.21.2 [10] <4.7>重做习题4.21.1，这次仅当通过改变或重排序指令都也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。

4.21.3 [10] <4.7>如果处理器中存在转发，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？

4.21.4 [20] <4.7>如果存在转发，在执行指令序列的前五个时钟周期，通过图4-60中的冒险检测和转发单元指出每个周期中哪些信号是有效的。

4.21.5 [10] <4.7>如果没有转发，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。

4.21.6 [20] <4.7>对习题4.21.5中新的冒险检测单元来说，给出执行时前五个时钟周期中每个周期设置的控制信号（使有效）。

习题4.22

本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、有完全转发和预测分支总发生的处理器上运行。试根据下表的两个指令序列分别回答下列问题。

a.	Label1:	lw $1,40($6)	
		beq $2,$3,Label2	;分支发生
		add $1,$6,$4	
	Label2:	beq $1,$2,Label1	;分支不发生
		sw $2,20($4)	
		and $1,$1,$4	
b.		add $1,$5,$3	
	Label1:	sw $1,0($2)	
		add $2,$2,$3	
		beq $2,$4,Label1	;分支不发生
		add $5,$5,$1	
		sw $1,0($2)	

4.22.1 [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。

4.22.2 [10] <4.8>重做习题4.22.1，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。

4.22.3 [20] <4.8> 不同于需要一个 ALU 操作，另一种提前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 是临时寄存器，并且可以使用 R 型指令 seq（set if equal，相等置 1）。

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，并且需要额外的转发逻辑和冒险检测。

4.22.4 [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 级执行分支应加入的冒险检测逻辑。该逻辑需要检测什么类型的冒险？

4.22.5 [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。

4.22.6 [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 级执行分支应加入的转发。比较新转发单元与图 4-62 中转发的复杂度。

习题 4.23

一个好的分支预测器有多重要取决于条件分支指令的多少，它与分支预测器的精度共同决定误预测分支导致的阻塞时间长短。下表给出了不同类型指令所占比例及对应分支预测器精度的两种情况，试根据不同情况回答下列问题。

	R 型	beq	jmp	lw	sw
a.	50%	15%	10%	15%	10%
b.	30%	10%	5%	35%	20%

	分支总发生	分支总不发生	2 位预测器
a.	40%	60%	80%
b.	60%	40%	95%

4.23.1 [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。

4.23.2 [10] <4.8> 重做习题 4.23.1，这次改为分支总不发生预测器。

4.23.3 [10] <4.8> 重做习题 4.23.1，这次改为 2 位分支预测器。

4.23.4 [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。

4.23.5 [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。

4.23.6 [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？

习题 4.24

本习题讨论不同分支预测器对固定分支模式（如循环）的预测精度。下表给出了两种分支模式（其中 T 表示分支发生，NT 表示分支未发生），试根据不同的分支模式回答下列问题。

	分支模式
a.	T, T, NT, T
b.	T, T, T, NT, NT

4.24.1 [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？

4.24.2 [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）。

4.24.3 [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？

4.24.4 [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。

4.24.5 [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，那么在习题4.24.4中你设计的预测器对这个分支的准确率是多少？

4.24.6 [20] <4.8> 重做习题4.24.4，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。

习题4.25

本习题讨论异常处理对流水线设计的影响。根据下表的两种情况（每种情况包含两条指令）分别回答下列问题。

	指令1	指令2
a.	`add $0, $1, $2`	`bne $1, $2,Label`
b.	`lw $2,40($3)`	`nand $1, $2, $3`

4.25.1 [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被检测到。

4.25.2 [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？假设设计处理器时已知每个异常处理程序的地址。

4.25.3 [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（见习题4.25.1）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。

下表给出了异常处理程序地址的两种情况，试根据每种情况分别回答下列问题。

	溢出	无效数据地址	未定义指令	无效指令地址	硬件故障
a.	0xFFFFF000	0xFFFFF100	0xFFFFF200	0xFFFFF300	0xFFFFF400
b.	0x00000008	0x00000010	0x00000018	0x00000020	0x00000028

4.25.4 [5] <4.9> 习题4.25.3中异常处理程序的地址是多少？如果指令存储器中该地址处是一条无效指令会怎样？

4.25.5 [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。重做习题4.25.3，这次使用支持向量异常处理的流水线。

4.25.6 [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，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。

习题4.26

本习题讨论异常处理对控制单元设计和处理器时钟周期的影响。根据下表两条引发异常的指令分别回答下列问题。

	指令	异常
a.	`add $0, $1, $2`	算术溢出
b.	`lw $2,40($3)`	无效的数据存储器地址

4.26.1 [10] <4.9>对每个流水级，确定指令流经该级时图4-66中异常相关控制信号的值。

4.26.2 [5] <4.9>一些在ID级生成的控制信号被保存在ID/EX流水线寄存器中，还有一些直接就进入了EX级。用该指令为例解释其原因。

4.26.3 [10] <4.9>如果我们在异常条件出现的下一级再处理异常，可以让EX级更快一些。以该指令为例，指出这种方法的主要缺点。

根据下表的两种流水级延迟情况，分别回答下列问题。

	IF	ID	EX	MEM	WB
a.	300 ps	320 ps	350 ps	350 ps	100 ps
b.	200 ps	170 ps	210 ps	210 ps	150 ps

4.26.4 [10] <4.9>如果每100 000条指令发生一次溢出异常，那么将溢出检查移到MEM级带来的总体加速比是多少？假设这个改变将使EX级的延迟减少30 ns并且无异常时流水线处理器的IPC为1。

4.26.5 [20] <4.9>可以在EX级而非ID级产生异常控制信号吗？为什么？使用“`bne $4, $5,Label`”指令和各流水级延迟为例进行说明。

4.26.6 [10] <4.9>假设每个多选器的延迟是40 ps，那么控制单元应在什么时间内产生清除信号？其中哪个信号最关键？

习题4.27

本习题讨论异常处理与分支和存取指令如何相互影响。下表给出了两种分支指令和对应延迟时间槽的情况，试根据每种情况分别回答下列问题。

	分支与延迟时间槽
a.	`beq $1, $0,Label` `sw $6,50($1)`
b.	`beq $5, $0,Label` `nor $5, $4, $3`

4.27.1 [20] <4.9>假设分支发生被正确地预测，但是标号“Label”处的指令是一条未定义指令。说明每个周期每级流水级中的情况，从分支指令被译码开始到异常处理程序的第一条指令被取指为止。

4.27.2 [10] <4.9>重做习题4.27.1，这次假设延迟时间槽中的指令也将在MEM级导致一个硬件错误异常。

4.27.3 [10] <4.9>如果分支发生但延迟时间槽中的指令引起了一个异常，那么EPC中的值是多少？异常处理程序完成后将发生什么情况？

试根据下表的两条存储指令分别回答下列问题。

	存储指令
a.	`sw $6,50($1)`
b.	`sw $5,60($3)`

4.27.4 [10] <4.9> 如果分支发生且标号“Label”处是一条无效的指令，而且异常处理程序的第一条指令是表中的 sw 指令且其访问了一个无效地址，这时会发生什么情况？

4.27.5 [10] <4.9> 如果存取指令的地址计算可能溢出，是否可以将这个溢出检测延迟到 MEM 级？以表中给出的 sw 指令为例进行说明。

4.27.6 [10] <4.9> 对调试来说，能够检测特定的存储器地址中是否被写入了一个特定的值非常有用。为了实现该功能，首先要加入两个寄存器：WADDR 和 WVAL。当 WADDR 中的地址被写入 WVAL 中的值时处理器产生一个异常。应该怎样设计流水线以实现该功能？在这个数据通路中 sw 指令将如何处理？

习题 4.28

本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。根据下表的两段 C 代码分别回答下列问题。

	C 代码
a.	`for(i=0;!=j;i++)` `  b[i]=a[i];`
b.	`for(i=0;a[i]!=a[i+1];i++)` `  a[i]=0;`

在写 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	c	空闲
a.	$1	$2	$3	$4	$5	$6，$7，$8
b.	$4	$5	$6	$7	$8	$1，$2，$3

4.28.1 [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。

4.28.2 [10] <4.10> 如果循环仅执行两次后就退出，画出习题 4.28.1 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）。

4.28.3 [10] <4.10> 重排序习题 4.28.1 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。

4.28.4 [10] <4.10> 重做习题 4.28.2，但这次使用习题 4.28.3 中的 MIPS 代码。

4.28.5 [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习题 4.28.1 的代码，假设循环执行 1 000 000 次。与习题 4.28.2 相同，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。

4.28.6 [10] <4.10> 重做习题 4.28.5，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。

习题 4.29

本习题讨论静态调度超标量处理器上循环的执行。为简化起见，假设一个周期内可以同时执行任意类型的指令组合。例如，在一个三发射超标量处理器上，同时执行三条 ALU 指令/分支指令/存取指令，或是任意这些指令的组合。注意，这个简化仅去掉了资源约束，数据相关和控制相关仍然需要考虑。根据下表的两个循环分别回答下列问题。

	循环		循环

a.	Loop: lw $1,40($6)
	add $5,$5,$1
	sw $1,20($5)
	addi $6,$6,4
	addi $5,$5,-4
	beq $5,$0,Loop
		b.	Loop: add $1,$2,$3
			sw $0,0($1)
			addi $2,$2,4
			beq $2,$0,Loop

4.29.1 [10] <4.10>如果循环执行很多次（如1 000 000次），在双发射静态超标量处理器中有多少周期（以百分比表示）需要使用寄存器堆的所有读端口？

4.29.2 [10] <4.10>如果循环执行很多次（如1 000 000次），在三发射静态超标量处理器中有多少周期（以百分比表示）需要使用寄存器堆的所有读端口？与习题4.29.1中的结果相比较。

4.29.3 [10] <4.10>如果循环执行很多次（如1 000 000次），在三发射静态超标量处理器中有多少周期（以百分比表示）需要使用寄存器堆的两到三个写端口？

4.29.4 [20] <4.10>展开循环一次并在双发射静态超标量处理器上调度。假设循环总是执行偶数次。在调度代码时可以使用寄存器$10 ~ $20来消除相关。

4.29.5 [20] <4.10>在双发射静态超标量处理器上，习题4.29.4的代码相对于原代码的加速比是多少？假设循环执行很多次（如1 000 000次）。

4.29.6 [10] <4.10>在单发射处理器上，习题4.29.4的代码相对于原代码的加速比是多少？假设循环执行很多次（如1 000 000次）。

习题4.30

本习题有一系列的假设条件。首先，假设一个N发射超标量处理器一个周期内能执行任意类型的N条指令。其次，假设每条指令是独立选择的，不考虑指令的位置。再次，假设没有因数据相关引起的阻塞，不使用延迟时间槽，并且分支方向在流水线的EX级确定。最后，假设执行各类型指令的比例如下表所示。试根据下表的两种情况回答下列问题。

	ALU	正确预测的分支	错误预测的分支	lw	sw
a.	50%	18%	2%	20%	10%
b.	40%	10%	5%	35%	10%

4.30.1 [5] <4.10>该程序在双发射静态超标量处理器上能取得的CPI是多少？

4.30.2 [10] <4.10>对双发射静态超标量处理器来说，分支预测器一个周期能预测两个分支相对于只能预测一个分支的加速比是多少？假设分支预测器不能处理分支阻塞。

4.30.3 [10] <4.10>对双发射静态超标量处理器来说，寄存器堆有两个写端口相对于只有一个写端口的加速比是多少？

4.30.4 [5] <4.10>对双发射静态超标量处理器（假设是经典的5级流水线）来说，完美的分支预测相对于实际的分支预测的加速比是多少？

4.30.5 [10] <4.10>重做习题4.30.4，这次是4发射处理器。当处理器的发射宽度增加时，关于一个良好的分支预测的重要性你能得出什么结论？

4.30.6 <4.10>重做习题4.30.5，这次是具有50级流水线的4发射处理器。假设经典的5级流水线中每级都被划分为更小的10级，并且分支方向确定是在新的10个EX级中的第一级。当处理器的流线深度增加时，关于一个良好的分支预测的重要性你能得出什么结论？

习题4.31

该习题基于下表中的循环，它分别以x86和MIPS两种指令形式给出。假设循环在结束前执行了很多

次。这意味着在计算性能时仅需计算“稳定”状态下的性能，不必考虑循环开始和结束时的情况。而且，假设有完全的转发支持和完美的分支预测，没有延迟时间槽，这意味着仅需考虑资源冒险和数据冒险。注意这个问题中的绝大多数 x86 都有两个操作数。指令的最后一个操作数（一般是第二个）同时是第一个源操作数和目的操作数。例如，“sub (edx), eax”读寄存器 edx 索引的存储器中的值，减去寄存器 eax 中的值，再把结果放入寄存器 eax 中。

	x86 指令	对应的 MIPS 指令

a.	Label: mov -4(esp),eax
	add (edx),eax
	
	mov eax,-4(esp)
	add 1,ecx
	add 4,edx
	cmp esi,ecx
	jl Label
		Label: lw $2,-4($sp)
		lw $3,0($4)
		add $2,$2,$3
		sw $2,-4($sp)
		addi $6,$6,1
		addi $4,$4,4
		slt $1,$6,$5
		bne $1,$0,Label

b.	Label: add eax,(edx)
	
	mov eax,edx
	add 1,eax
	jl Label
		Label: lw $2,0($4)
		add $2,$2,$5
		sw $2,0($4)
		add $4,$5,$0
		addi $5,$5,1
		slt $1,$5,$0
		bne $1,$0,Label

4.31.1 [20] <4.11>循环的 MIPS 版本在静态调度 5 级流水线单发射处理器上运行的 CPI 为多少？

4.31.2 [20] <4.11>循环的 x86 版本在静态调度 7 级流水线单发射处理器上运行的 CPI 为多少？流水级的各级分别是 IF、ID、ARD、MRD、EXE 和 WB。IF 和 ID 级与 5 级 MIPS 流水线是类似的。ARD 计算读存储器的地址；MRD 执行存储器读；EXE 执行操作；WB 将结果写回寄存器堆或存储器。数据存储器有一个读端口（对 MRD 级的指令）和一个写端口（对 WB 级的指令）。

4.31.3 [20] <4.11>如果循环的 x86 版本在这样一个处理器上运行，其内部将 x86 指令转变成类 MIPS 的微操作，然后在静态调度的单发射 5 级流水线上执行微操作，其能取得的 CPI 是多少？注意计算 CPI 时用的是 x86 指令的数量。

4.31.4 [20] <4.11>循环的 MIPS 版本在动态调度的单发射处理器上运行的 CPI 为多少？假设处理器不进行寄存器重排名，所以只能重排序无数据相关的指令。

4.31.5 [30] <4.10, 4.11>假设有许多空闲的寄存器可用，对循环的 MIPS 版本进行重命名以尽可能消除循环内的数据相关。使用这个重命名后的代码重做习题 4.31.4。

4.31.6 [20] <4.10, 4.11>重做习题 4.31.4，但这次假设处理器在每条指令译码时分配给其一个新名字，然后重命名后续指令使用的寄存器以使用正确的值。

习题 4.32

本习题假设分支指令占所有执行指令的比例和分支预测准确率如下表所示。假设处理器不会因数据相关和资源相关阻塞，例如，在没有控制冒险的情况下，处理器总是每个周期取和执行最大数量的指令。对控制相关来说，处理器使用分支预测并沿预测的方向继续取指。如果分支预测错误，分支结果确定时将丢弃因为误预测分支而取的那些指令，下个周期处理器将沿正确的方向开始取指。

	所有执行指令中分支指令的比例	分支预测准确率
a.	20	90%
b.	20	99.5%

4.32.1 [5] <4.11> 在检测到一次分支误预测到检测到下一次分支误预测之间，估计有多少条指令被执行？根据下表的流水线深度和确定分支方向的级（从第1级算起）的两种情况，分别回答下列问题。

	流水线深度	在哪一级确定分支方向
a.	12	10
b.	25	18

4.32.2 [5] <4.11> 在具有上述流水线参数的4发射处理器中，在任意给定周期内估计有多少条分支指令在“进行中”（已经取指但还没提交）？

4.32.3 [5] <4.11> 在4发射处理器中一次误预测将导致误取多少条指令？

4.32.4 [10] <4.11> 8发射处理器相对于4发射处理器的加速比是多少？假设8发射处理器与4发射处理器仅在每周期发射指令数上不同，其他方面（流水线深度、分支确定级等）都相同。

4.32.5 [10] <4.11> 4发射处理器中提前一级确定分支方向带来的加速比是多少？

4.32.6 [10] <4.11> 8发射处理器中提前一级确定分支方向带来的加速比是多少？与习题4.32.5的结果进行比较。

习题4.33

本习题讨论分支预测对深流水线多发射处理器性能的影响。根据下表流水线深度和发射宽度的两种情况分别回答下列问题。

	流水线深度	发射宽度
a.	10	4
b.	25	2

4.33.1 [10] <4.11> 为避免资源冒险处理器的寄存器堆需要多少个读端口？

4.33.2 [10] <4.11> 如果不存在分支误预测和数据相关，该处理器性能比经典的5级流水单发射处理器要好多少？假设时间周期的减少与流水线级数成比例。

4.33.3 [10] <4.11> 重做习题4.33.2，但这次假设每条指令都与紧跟其后的指令存在RAW（Read After Write，写后读）数据相关。假设转发可以允许连续的指令背靠背执行，不需要阻塞。

根据下表的分支指令比例、确定分支方向级、分支预测器精度和分支误预测时性能损失的两种情况，分别回答下列问题。

	所有执行指令中分支指令的比例	确定分支方向的级	分支预测器精度	性能损失
a.	30%	7	95%	10%
b.	15%	8	97%	2%

4.33.4 [10] <4.11> 在给定的分支指令比例和预测精度下，有多少周期（以百分比表示）完全用于错误分支方向上的取指？忽略表中的性能损失。

4.33.5 [20] <4.11> 如果想限制误预测导致的阻塞不造成表中给出的性能损失程度，分支预测准确率应该是多少？忽略表中的分支预测准确率。

4.33.6 [10] <4.11> 如果想让性能达到具有理想分支预测处理器性能的一半，分支预测准确率应该是多少？

习题4.34

本习题讨论4.13节的“流水线是简单的”误解。根据下表的两条MIPS指令分别回答下列问题。

	指令	解释
a.	add Rd,Rs,Rt	Reg[Rd] = Reg[Rs] + Reg[Rt]
b.	lw Rt,Offs(Rs)	Reg[Rt] = Mem[Reg[Rs] + Offs]

4.34.1 [10] <4.13> 给出仅需支持该指令的流水线数据通路。假设执行的所有指令都是该指令的不同实例。

4.34.2 [10] <4.13> 给出习题 4.34.1 中数据通路所需的转发和冒险检测单元。

4.34.3 [10] <4.13> 为了支持未定义指令异常需要给习题 4.34.1 的数据通路增加哪些电路？注意未定义指令异常应该在处理器遇到任何其他类型指令时触发。

根据下表的两条 MIPS 指令分别回答下列问题。

	指令	翻译
a.	beq Rs,Rt,Label	if Reg[Rs] == Reg[Rt] PC = PC + Offs
b.	and Rd,Rs,Rt	Reg[Rd] = Reg[Rs] & Reg[Rt]

4.34.4 [10] <4.13> 说明如何扩展习题 4.34.1 的数据通路以支持该指令。扩展后的数据通路应仅支持这两条指令的实例。

4.34.5 [10] <4.13> 对习题 4.34.4 扩展后的数据通路重做习题 4.34.2。

习题 4.35

本习题讨论指令集设计与流水线之间的关系。假设我们有一个多发射流水线处理器，其流水线深度、发射宽度、确定分支方向所在级、分支预测器精度等如下表所示，试根据每种情况分别回答下列问题。

	流水线深度	发射宽度	确定分支方向的级	分支预测器精度	所有指令中分支指令的比例
a.	10	4	7	80%	20%
b.	25	2	17	92%	25%

4.35.1 [5] <4.18, 4.13> 控制冒险能通过加入分支延迟时间槽消除。如果想消除该处理器的所有控制冒险，每个分支后应有几个分支延迟时间槽？

4.35.2 [10] <4.8, 4.13> 通过使用 4 个分支延迟时间槽，可以给处理器带来多大的加速比？假设指令间没有数据相关，并且所有 4 个延迟时间槽中都填满了有用的指令。为了简化计算，可以假设误预测的分支指令总是一个周期内取的最后一条指令，例如，当在误预测的路径上取出分支指令时同流水级中没有其他指令。

4.35.3 [10] <4.8, 4.13> 重做习题 4.35.2，但这次假设执行分支的 10% 有 4 个延迟时间槽且其中都填充了有用的指令，20% 的分支有 3 个延迟时间槽填充了有用的指令（第四个延迟时间槽中是 nop），30% 的分支有 2 个延迟时间槽填充了有用的指令，剩下 40% 的分支在其延迟时间槽中没有有用的指令。

根据表中的两个 C 循环代码分别回答下列问题。

a.
```
for(i=0;i!=j;i++){
  b[i]=a[i];
}
```

b.
```
for(i=0;a[i]!=a[i+1];i++){
  c++;
}
```

4.35.4 [10] <4.8, 4.13> 将 C 版本的循环代码转化为 MIPS 版本的，假设我们的指令集对每个分支要求一个延迟时间槽。尽可能在延迟时间槽中填充非 nop 指令。假设变量 a、b、c、i 和 j 分别被保存在寄存器 $1、$2、$3、$4 和 $5 中。

4.35.5 [10] <4.7, 4.13> 重做习题 4.35.4，这次每个分支有 2 个分支延迟时间槽。

4.35.6 [10] <4.10, 4.13>可以同时有多少个习题4.35.4中的循环在处理器的流水线中“处理”？我们所谓的“处理”是指至少循环中的一条指令被取指但还没有被提交。

习题4.36

本习题讨论4.13节提到的最后一个陷阱——指令集设计时没有考虑流水线。根据下表的两条新MIPS指令分别回答下列头4个问题。

	指令	解释
a.	`lwinc Rt,Offset(Rs)`	`Reg[Rt] = Mem[Reg[Rs] + Offset]` `Reg[Rs] = Reg[Rs] + 4`
b.	`addr Rt,Offset(Rs)`	`Reg[Rt] = Mem[Reg[Rs] + Offset] + Reg[Rt]`

4.36.1 [10] <4.11, 4.13>将该指令翻译成MIPS微操作。

4.36.2 [10] <4.11, 4.13>为了支持新指令所需的微操作，需要对原有的5级MIPS流水线做什么改动？

4.36.3 [20] <4.13>如果要把这条指令加入MIPS指令集，讨论应如何修改流水线（在哪一级修改？修改哪些结构？）以直接（不是以微操作的形式）支持该指令。

4.36.4 [10] <4.13>你认为这条指令可能用的概率有多大？你认为把这条指令加入MIPS指令集合适吗？

本练习剩下的两个问题，是关于向ISA中添加一条新的addm指令的。在添加了addm的处理器中，按照在该周期完成了哪条指令（或哪种停顿阻止了指令的完成），假定这些问题对时钟周期进行如下分类：

	add	beq	lw	sw	addm	控制阻塞	数据阻塞
a.	35%	20%	20%	10%	5%	5%	5%
b.	25%	10%	25%	10%	10%	10%	10%

4.36.5 [10] <4.13>如果用三条指令组成的序列（lw、add和sw）代替addm指令能取得多少的加速比？假设addm指令以某种方式被经典的5级流水线支持且不会产生结构冒险。

4.36.6 [10] <4.13>重做习题4.36.5，但这次假设addm指令的支持需要增加一级流水线。如果addm被翻译成三条指令执行，就可以不需要这一级流水线，这意味着可以消除一半数据阻塞。注意，数据阻塞的消除仅对addm翻译前存在的阻塞有效，对addm翻译本身产生的阻塞无效。

习题4.37

本习题讨论流水线设计的一些折中考虑，如时钟周期和硬件资源的利用。根据表中的两段MIPS代码分别回答下列问题。假设处理器不使用延迟时间槽。

a.	`lw $1,40($6)`
	`beq $1, $0,Label ;假设 $1 == $0`
	`sw $6,50($1)`
	`Label: add $2, $3, $1`
	`sw $2,50($1)`

b.	`lw $5, -16($5)`
	`sw $5, -16($5)`
	`lw $5, -20($5)`
	`beq $5, $0,Label ;假设 $5 != $0`
	`add $5, $5, $5`

4.37.1 [5] <4.3, 4.14>基本的单周期数据通路的哪些单元会被所有这些指令用到？哪些部件是最少用到的？

4.37.2 [10] <4.6, 4.14> 怎样使用数据存储器的读端口和写端口?

4.37.3 [10] <4.6, 4.14> 假设已经有一个单周期的设计。实现流水线设计需要的流水线寄存器的总位数是多少?

根据下表各单元延迟的两种情况分别回答下列问题。

	指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
a.	400 ps	100 ps	30 ps	120 ps	200 ps	350 ps	20 ps	0 ps
b.	500 ps	150 ps	100 ps	180 ps	220 ps	1000 ps	90 ps	20 ps

4.37.4 [10] <4.3, 4.5, 4.14> 比较单周期实现和5级流水线实现的时钟周期。

4.37.5 [10] <4.3, 4.5, 4.14> 重做习题4.37.4, 但这次假设数据仅需支持ADD指令。

4.37.6 [20] <4.3, 4.5, 4.14> 如果减少数据通路中某单元延迟1 ps的代价是$1, 那单周期实现和流水线实现的时钟周期减少20%所需的代价分别是多少?

习题4.38

本习题讨论性能与功耗的关系。假设数据通路各单元的功耗如下表所示, 其他单元的功耗可以忽略。试根据下表的两种情况分别回答下列问题。

	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
a.	100 pJ	60 pJ	70 pJ	120 pJ	100 pJ
b.	200 pJ	90 pJ	80 pJ	300 pJ	280 pJ

4.38.1 [10] <4.3, 4.6, 4.14> 在单周期实现和流水线实现中执行一条加法指令的功耗分别是多少?

4.38.2 [10] <4.6, 4.14> 功耗消耗最大的MIPS指令是哪一条? 执行这条指令的功耗是多少?

4.38.3 [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` 指令的功耗是多少?

假设数据通路各单元的功耗如下表所示, 其他单元的功耗可以忽略。试根据下表的两种情况分别回答下列问题。

	指令存储器	控制	寄存器读或写	ALU	数据存储器读或写
a.	400 ps	300 ps	200 ps	120 ps	350 ps
b.	500 ps	400 ps	220 ps	180 ps	1000 ps

4.38.4 [10] <4.6, 4.14> 如果像习题4.38.3中那样设计流水线, 其对性能会造成多大的影响?

4.38.5 [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?

4.38.6 [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?

习题4.39

本习题假设程序执行时, 处理器的时钟周期是这样消耗的: 一个周期被“消耗”在一条指令上仅当在该周期完成这种类型的指令时。一个周期被“消耗”在阻塞上仅当处理器在该周期因阻塞没有完成一条指令时。

	add	beq	lw	sw	控制阻塞	数据阻塞
a.	35%	20%	20%	10%	10%	5%
b.	25%	10%	25%	10%	20%	10%

本题还假设各流水级的延迟和功耗如下表所示。各流水线在给定延迟内完成工作必须消耗相应的功耗。注意，无存储器存取时 MEM 级是没有功耗的。类似的，没有寄存器写时，WB 级也是没有功耗的。试根据表中的两种情况分别回答下列问题。

	IF	ID	EX	MEM	WB
a.	300 ps/120 pJ	400 ps/60 pJ	350 ps/75 pJ	500 ps/130 pJ	100 ps/20 pJ
b.	200 ps/150 pJ	150 ps/60 pJ	120 ps/50 pJ	190 ps/150 pJ	140 ps/20 pJ

4.39.1 [10] <4.14>处理器的性能（每秒执行指令数）如何?

4.39.2 [10] <4.14>处理器的功耗（每秒消耗多少焦）如何?

4.39.3 [10] <4.6，4.14>在不影响处理器时钟周期的前提下，哪个流水级可以慢一些，最多慢多少?

4.39.4 [20] <4.6，4.14>为了减少功耗经常牺牲一些电路的速度。假设我们可以在增加延迟至 X 倍的同时减少功耗至 1/X。试对各流水级进行调整以在不影响处理器总体性能的前提下尽量减少功耗。对调整后的处理器重做习题 4.39.2。

4.39.5 [10] <4.6，4.14>重做习题 4.39.4，但这次的目标是最小化每指令功耗但时钟周期的增加不超过 10%。

4.39.6 [10] <4.6，4.14>重做习题 4.39.5，但这次假设可以在增加延迟至 X 倍的同时减少功耗至 $1/X^2$。相对于习题 4.39.2 的结果而言这种方法能节省多少功耗?

小测验答案

4.1 节 控制器、数据通路、存储器。少了输入和输出。

4.2 节 错。边沿触发状态单元可以同时进行读写。

4.3 节 1）A；2）C。

4.4 节 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 节 1）因为 lw 的结果而阻塞；2）转发第一个加法器的结果写入 `$t1`；3）不需要阻塞或旁路。

4.6 节 B 和 D 正确，其余错误。

4.8 节 A. 预测不发生；B. 预测发生；C. 动态预测。

4.9 节 A。因为在逻辑上它最先执行。

4.10 节 A. 都有；B. 都有；C. 软件；D. 硬件；E. 硬件；F. 硬件；G. 都有；H. 硬件；I. 都有。

4.11 节 前两个错误，后两个正确。

4.12 节 光盘页 6.7-3，第一种和第三种情况正确。

4.12 节 光盘页 6.7-7，只有第三种情况完全正确。

第5章

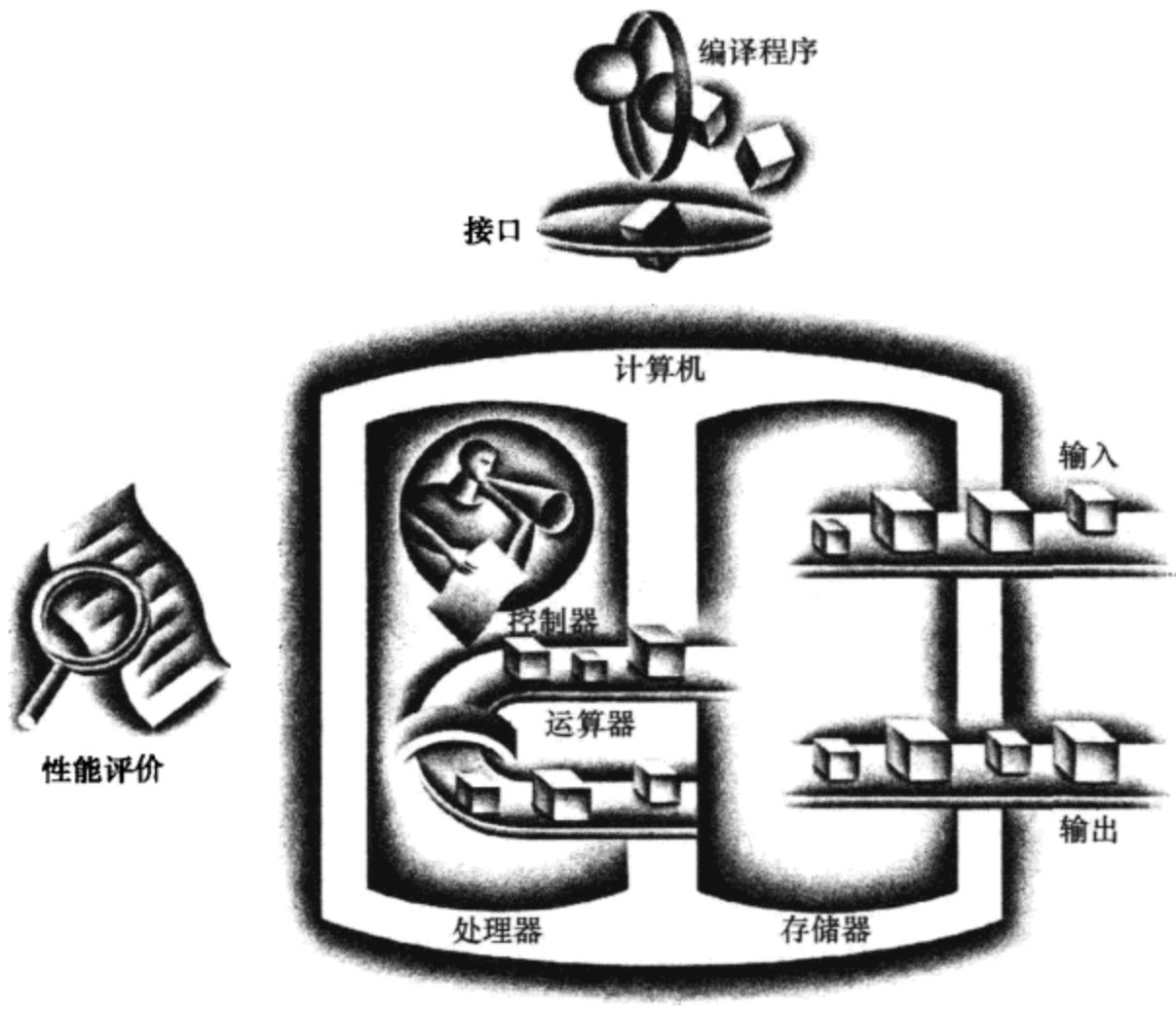
Computer Organization and Design: The Hardware/Software Interface, 4E

大容量和高速度：开发存储器层次结构

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……字都可以立刻得到……在实际中，我们需要构建一个存储器的层次结构，其中的每一层都比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



计算机的五大经典部件

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无限大容量的虚拟快速存储器。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。

假如一个学生正在完成一份关于计算机硬件重要历史性发展的论文，他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。这样，所需要的相关资料就可能在这些书中找到，并且在很长一段时间内，只需阅读摆在书桌上的书而无需返回到书架前。当然不排除会出现其间要从书架上增补部分所需资料到书桌上的情况。但与每次从书架取一本书到书桌，并不断地回到书架前还书而后取另一本书相比，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

像你不会一次以相同的概率查阅图书馆中的每一本书那样，一个程序也不会一次以相同的概率访问它全部的代码或数据。否则，让存储器访问速度既快且容量又大是不可能的，就好比把图书馆中所有的书放在书桌上，还要保持快速查找一样是不可能的。

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**㊀（时间上的局部性）：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，那么很快再次查阅它的可能性是很大的。
- **空间局部性**㊁（空间上的局部性）：如果某个数据项被访问，与它地址相邻的数据项很快可能也将被访问。例如，当查找到一本关于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同样有所需的材料。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层次结构。

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**㊂。存储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。

目前，构建存储器层次结构主要有三种技术。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 C（见光盘）的 C.9 节中介绍。第三种技术是磁盘（disk），它通常是存储层次结构中容量最大且速度最慢的一层。在很多嵌入式设备中，常用闪存（flash memory）来替代磁盘，详见 6.4 节。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 2008 年的典型数据）。

存储器技术	典型存取时间	2008 年每 GB 的价格
SRAM	0.5 ~ 2.5 ns	$2 000 ~ $5 000
DRAM	50 ~ 70 ns	$25 ~ $75
磁盘	5 000 000 ~ 20 000 000 ns	$0.02 ~ $2

由于价格和访问时间的不同，构建存储器的层次结构是有好处的。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。

数据同样被组织成层次化结构：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。

㊀ 时间局部性（temporal locality）：某个数据项在被访问之后可能很快被再次访问的特性。
㊁ 空间局部性（spatial locality）：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。
㊂ 存储器层次结构（memory hierarchy）：一种由多存储层次组成的结构，存储器的容量和访问时间随着与处理器距离的增加而增加。

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且访问速度更快，这是因为它采用了成本更高的技术来实现的。如图 5-2 中所示，我们将一个两级层次结构中存储信息的最小单元称为**块**（block）或**行**（line）[一]，就像在图书馆中，一个信息块就是一本书。

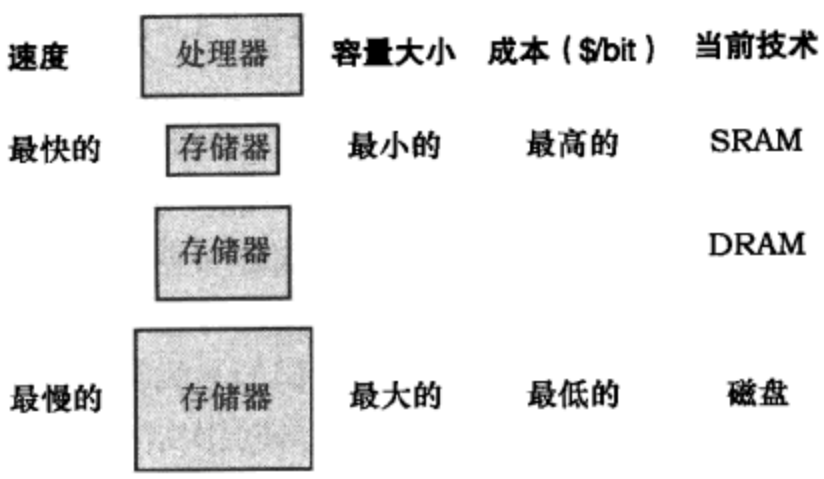


图 5-1　存储器层次的基本结构

存储系统采用层次结构后，用户对于存储器的认识就是：它的容量和层次结构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。在很多嵌入式系统中，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层次中引入新的一层；见 6.4 节。

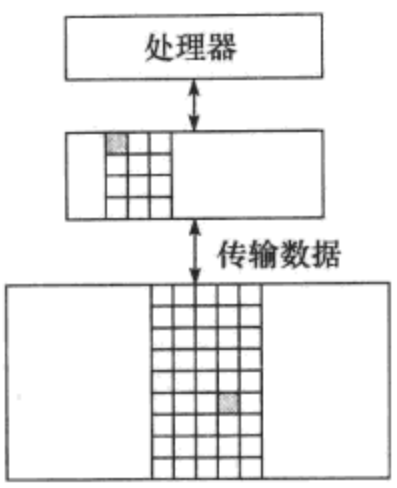


图 5-2　存储器层次结构中的每两个层次可以被认为一个是高层次，一个是低层次

在每一层中，那些存储信息的最小单元被称为块或者行。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**[二]，或命中比率，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**[三]（1 – 命中率）则是数据在高层存储器中没有找到的存储访问比例。

追求高性能是我们使用存储器层次结构的主要目的，因而命中和缺失的执行时间就显得尤为重要。**命中时间**[四]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（浏览书桌上书籍所花费的时间）。**缺失代价**[五]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（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）。

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也将影响到一台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平台随机访问存储设备，而现在必须理解存储层次结构如何工作才能获得良好的性能。稍后我们将举例来说明其重要性（见图 5-18）。

㊀ 块（block）或行（line）：可存在于或不存在于 cache 中的信息的最小单元。
㊁ 命中率（hit rate）：在高层存储器中找到目标数据的存储访问比例。
㊂ 缺失率（miss rate）：在高层存储器中没有找到目标数据的存储访问比例。
㊃ 命中时间（hit time）：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。
㊄ 缺失代价（miss penalty）：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、将数据逐层传输、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。

由于存储系统对性能至关重要，计算机设计人员花费了大量精力在这些系统上，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，简化和抽象了许多概念。

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能被很快处理。而缺失后，需要访问容量大但速度慢的低层存储层次。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。

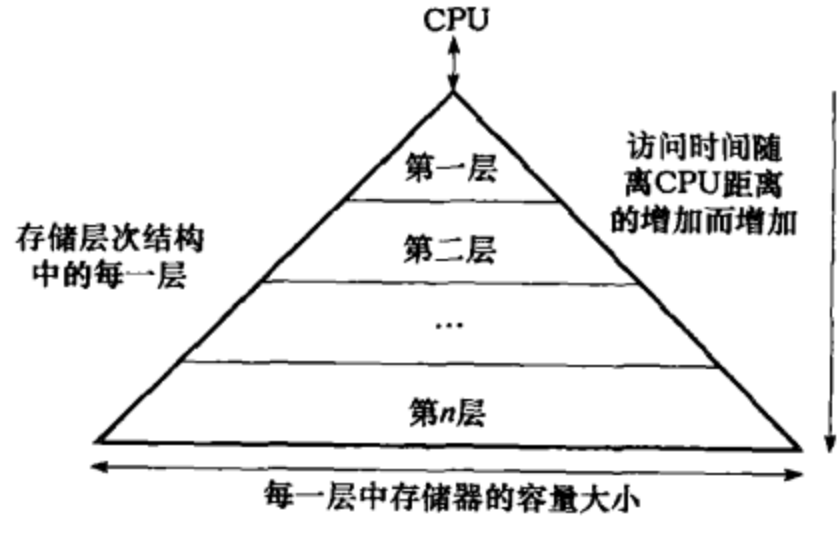


图 5-3　这幅图说明了存储器层次结构：离处理器越远，容量越大

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一层来决定，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。

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 层中存在。

小测验

下面哪些表述通常是正确的？

A. cache 利用了时间局部性。

B. 在一次读操作中，返回的值取决于哪些块在 cache 中。

C. 存储器层次结构的大部分成本在于最高一层。

D. 存储器层次结构的大部分容量处于最低一层。

5.2　cache 的基本原理

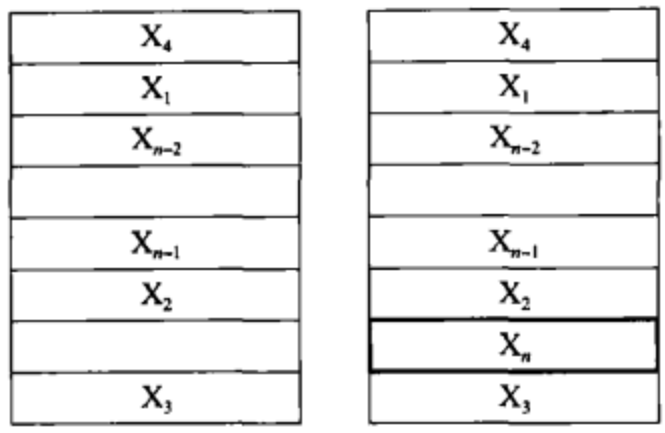
cache：一个隐藏或者存储信息的安全场所。

——《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。

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 5.3 节）。图 5-4 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 5-4　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，并强制 cache 从存储器中取回 X_n，随后将 X_n 放入 cache 中。

观察图 5-4 中的情景，有两个问题需要解决：我们怎样知道一个数据项是否在 cache 中？此外，如果数据项在 cache 中，我们如何找到它？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**[㊀]。每个主存地址对应到 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）

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 5-5 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。

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**[㊁]，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 5-5 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。

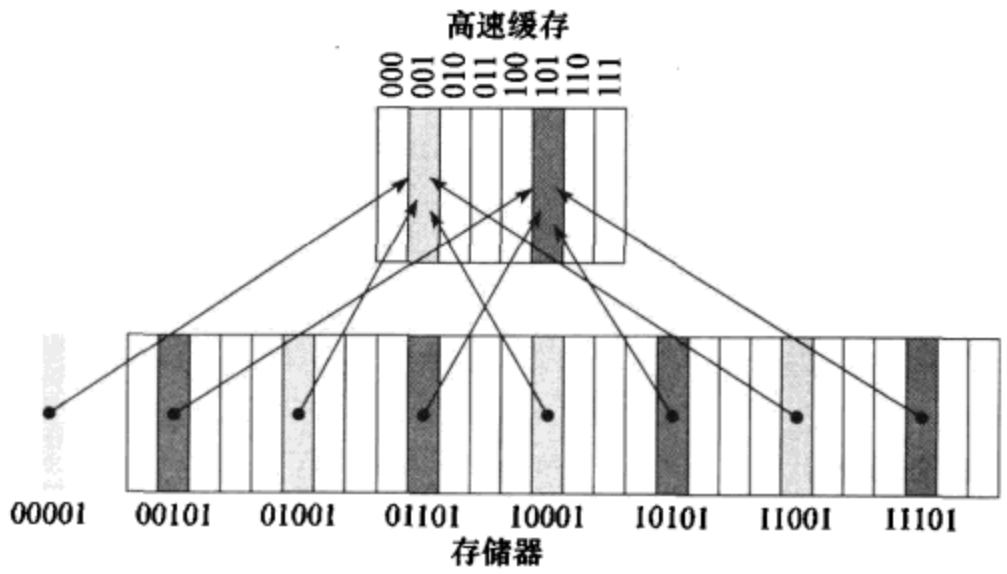


图 5-5　主存地址 0 ~ 31 被映射到 cache 中的相同位置，该 cache 中有 8 个字

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 $\log_2(8) = 3$ 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。

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 中的一些块依然为空，如图 5-4 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**[㊂]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。

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

㊀ 直接映射（direct mapped）：一种 cache 结构，其中每个主存地址仅仅对应到 cache 中的一个位置。

㊁ 标记（tag）：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。

㊂ 有效位（valid bit）：表中的一个字段，用来标识一个块是否含有一个有效数据。

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。

5.2.1　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。

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（7.6b）	$(10110_2 \bmod 8) = 110_2$
26	11010_2	缺失（7.6c）	$(11010_2 \bmod 8) = 010_2$
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（7.6d）	$(10000_2 \bmod 8) = 000_2$
3	00011_2	缺失（7.6e）	$(00011_2 \bmod 8) = 011_2$
16	10000_2	命中	$(10000_2 \bmod 8) = 000_2$
18	10010_2	缺失（7.6f）	$(10010_2 \bmod 8) = 010_2$
16	10000_2	命中	$(10000_2 \bmod 8) = 000_2$

图5-6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。

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）

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）

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）

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）

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）

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）

图 5-6　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）

a）启动后 cache 的初始状态；b）处理地址（10110_2）缺失后；c）处理地址（11010_2）缺失后；d）处理地址（10000_2）缺失后；e）处理地址（00011_2）缺失后；f）处理地址（10010_2）缺失后 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）。这些图指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。

由于 cache 初始为空，第一次访问的一些数据都会发生缺失。图 5-6 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。

上述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 5-7 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。

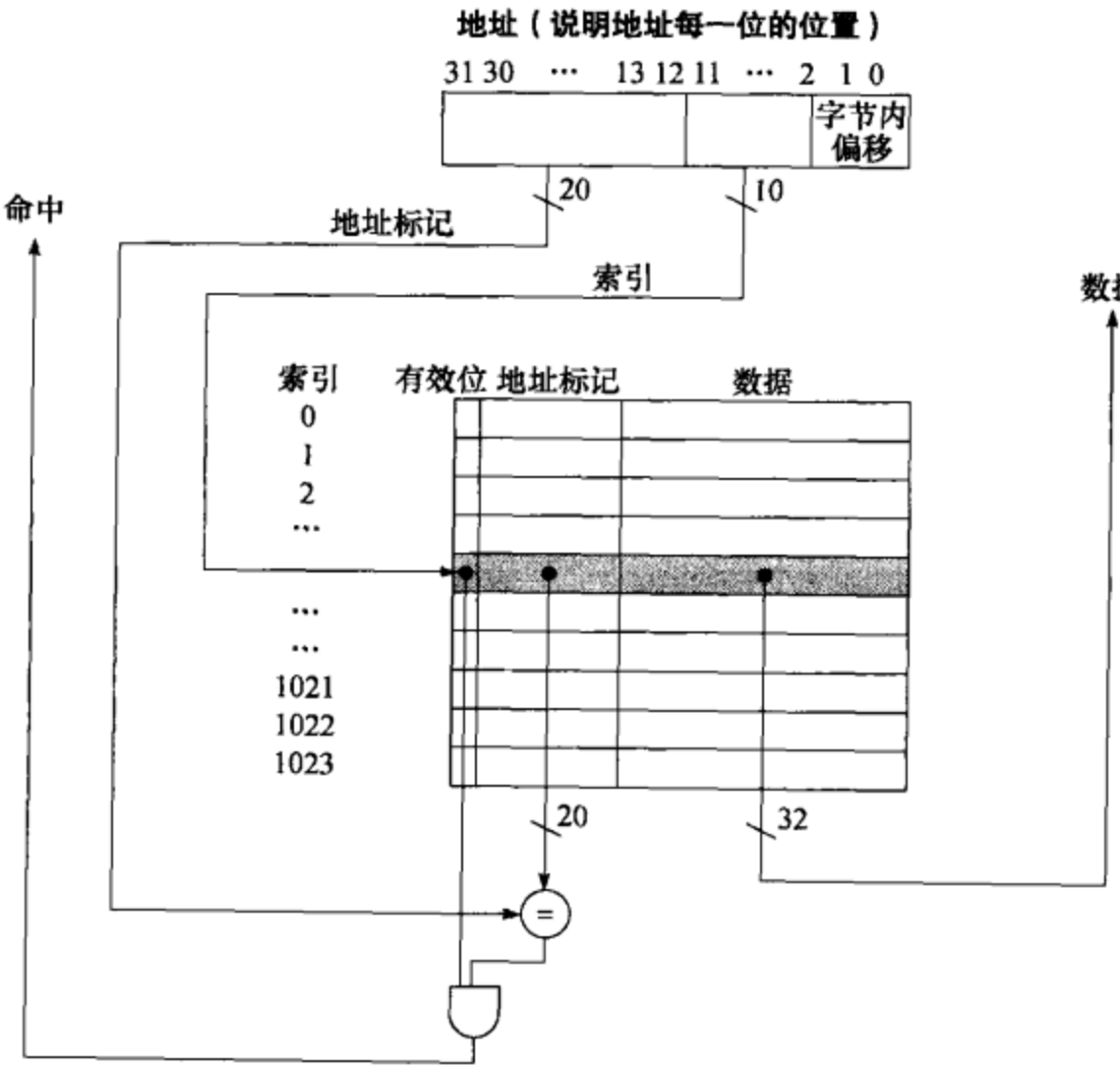


图 5-7　对这个 cache，地址的低位用来选择由数据字和标记组成的一个 cache 项

这个 cache 中有 1024 个字或者 4 KB。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}（1024）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。否则就发生缺失。

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 不仅存储数据而且存储标记位，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 位字节地址。

- 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}个字节），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

$$2^n \times (\text{块大小} + \text{标记域大小} + \text{有效位域大小})$$

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 5-7 中是一个 4 KB 的 cache。

举例　cache 中的位数

假设一个直接映射的 cache，有 16 KB 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 16 KB 是 4 K（2^{12}）字，块大小是 4 个字（2^2），那么就有 1024（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

$$2^{10} \times (4 \times 32 + (32-10-2-2)+1) = 2^{10} \times 147 = 147 \text{ Kbits}$$

即能装 16 KB 数据的 cache 总共需要 18.4 KB 的容量。这个 cache 的总位数是数据存储量的 1.15 倍。

举例　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，那么字节地址为 1200 将被映射到 cache 中的哪一块？

答案

块由下面公式给出：

$$(\text{块地址}) \bmod (\text{cache 中的块数})$$

其中块地址为

$$\text{字节地址}/\text{每块字节数}$$

注意，这个块地址包含了所有在

$$\left\lfloor \frac{\text{字节地址}}{\text{每块字节数}} \right\rfloor \times \text{每块字节数}$$

和

$$\left\lfloor \frac{\text{字节地址}}{\text{每块字节数}} \right\rfloor \times \text{每块字节数} + (\text{每块字节数} - 1)$$

之间的地址。

因此，由于每个块有 16 个字节，字节地址 1200 对应的块地址为

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

对应于 cache 中的块号（75 mod 64）＝11。事实上，地址 1200 和 1215 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 5-8 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。

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。

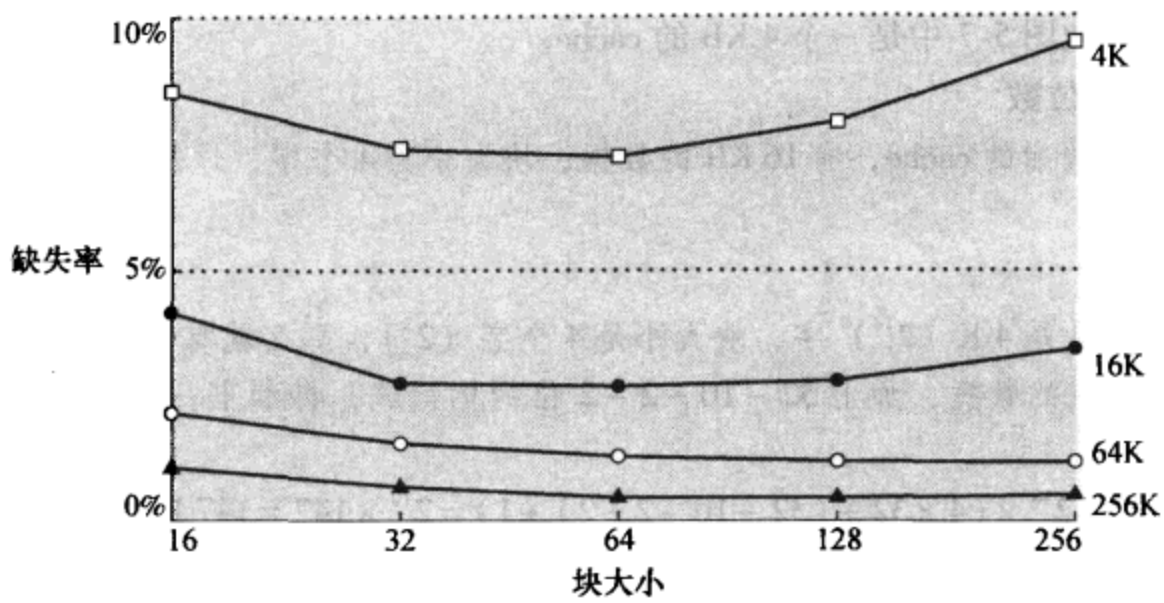


图 5-8　缺失率与块大小

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（图中没有考虑关联度，稍后讨论）。可惜的是，如果包括块大小，那么 SPEC2000 追踪信息将花费太长的时间，因此这些数据都基于 SPEC92。

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），即当块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 时效率要低一些，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，会受限于同样的问题。

5.2.2　cache 缺失处理

在研究一个真实系统中的 cache 之前，让我们来看一下控制部件是如何处理 **cache 缺失**[⊖]的。（在 5.7 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

⊖ cache 缺失：由于数据不在 cache 中而导致被请求的数据不能满足。

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。

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，临时寄存器和程序员可见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。

让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，程序计数器加了一个增量，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）重新返回指令执行第一步，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。

5.2.3　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，这种方法被认为**写直达法**[⊖]。

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 $1.0 + 100 \times 10\% = 11$，性能降低 10 倍多。

这个问题的一种解决方法是采用**写缓冲**[⊖]。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

⊖ 写直达法（write-through）：也译为写通过。写操作总是同时更新主存和 cache，以保持二者一致性的一种方法。

⊖ 写缓冲（write buffer）：一个保存等待写入主存数据的缓冲队列。

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要慢，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要慢，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**写回机制**[⊖]。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。

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 5.5 节，我们会对写操作进行更详细的介绍。

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填零一样。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。

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到 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。

5.2.4 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 AMD Opteron X4（Barcelona）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 5-9 给出了内置 FastMATH 处理器数据 cache

---

⊖ 写回机制（write-back）：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。

的结构。

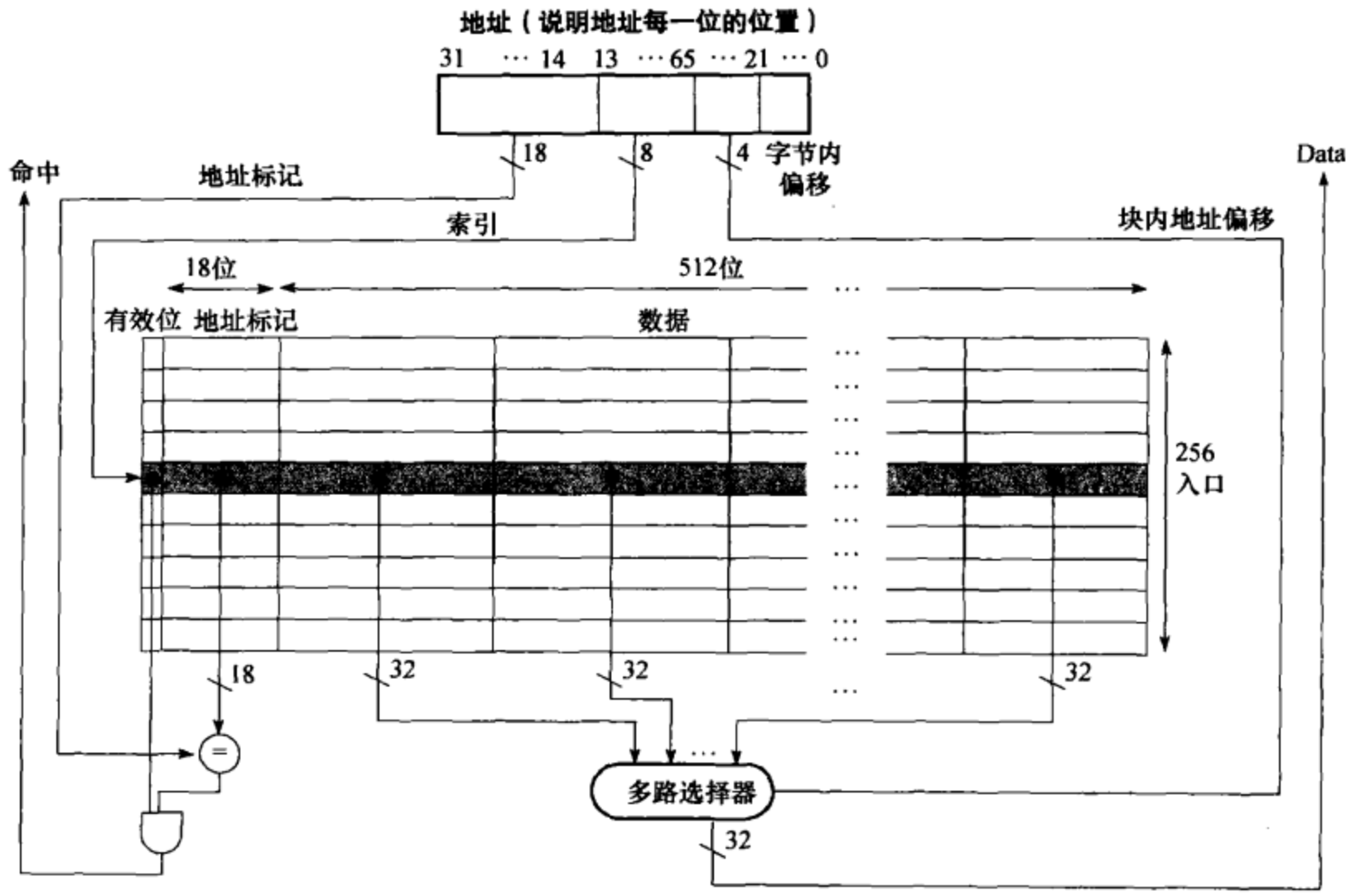


图 5-9　内置 FastMATH 处理器的 16 KB 的 cache，cache 中有 256 块，每块 16 个字

标记域是 18 位，索引域是 8 位，另有一个 4 位（5 ~ 2 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内偏移提供。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，就像在第 4 章中讨论的一样。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。为了满足流水线不阻塞的需求，使用了分离的指令 cache 和数据 cache。每个 cache 容量为 16 KB，即 4 K 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 5-9 底部所示），从检索到的块中选择 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 5-10 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问的

实际缺失率。

指令缺失率	数据缺失率	实际综合缺失率
0.4%	11.4%	3.2%

图 5-10　内置 FastMATH 处理器执行 SPEC2000 测试程序时指令和数据的近似缺失率

综合缺失率是将 16 KB 的指令 cache 和 16 KB 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**[⊖]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 没有将指令用的条目数与数据用的条目数严格区分出来。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 5.5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：32 KB。
- 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些。

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 5.3 节所示。

5.2.5　设计支持 cache 的存储系统

cache 缺失时从主存中取数，而主存由 DRAM 构成。在 5.1 节，我们看到 DRAM 设计最初的重点在成本和密度上。尽管减少从存储器取出第一个字的延迟比较困难，但是我们可以通过增加存储器和 cache 之间的带宽来降低缺失代价。这使我们在使用较大容量块的同时，仍然保持与较小容量块相近的低缺失代价。

通常来说，处理器通过总线与存储器相连（就像我们将在第6 章所见，传统在变革，但是实际上互联技术并没有变化，因此我们依然使用总线）。总线的时钟频率通常比处理器要慢很多。总线的速度影响了缺失代价。

为了理解不同存储器组织结构的影响，我们定义了一组存储器访问时间，假定：

- 发送地址需要 1 个存储器总线时钟周期；
- 每次 DRAM 访问需要 15 个存储器总线时钟周期；
- 传送一个数据字需要 1 个存储器总线时钟周期。

如果一个 cache 块中有 4 个字，DRAM 的存储区为一个字宽，缺失代价为 $1 + 4 \times 15 + 4 \times 1 = 65$ 个存储器总线时钟周期。因此，对每次单独缺失，每个总线时钟周期传送的字节数为

$$4 \times 4/65 = 0.25$$

图 5-11 显示了存储系统设计的三种选择。第一种选择延续了上面的假设：存储器是一个字宽，所有的访问都是顺序执行。第二种选择通过加宽存储器以及处理器和存储器之间的总线来增加存储器带宽；这种方法允许并行地访问块中多个字。第三种选择没有加宽互联总线，仅仅通

⊖ 分离 cache（split 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。

过加宽存储器来增加带宽。因此，我们仍然需要花费时间传输每一个字，但可以避免不止一次的访问延迟时间。让我们来看后面两种方法如何改进图 5-11a 所示的第一种方法中的 65 个时钟周期的缺失代价。

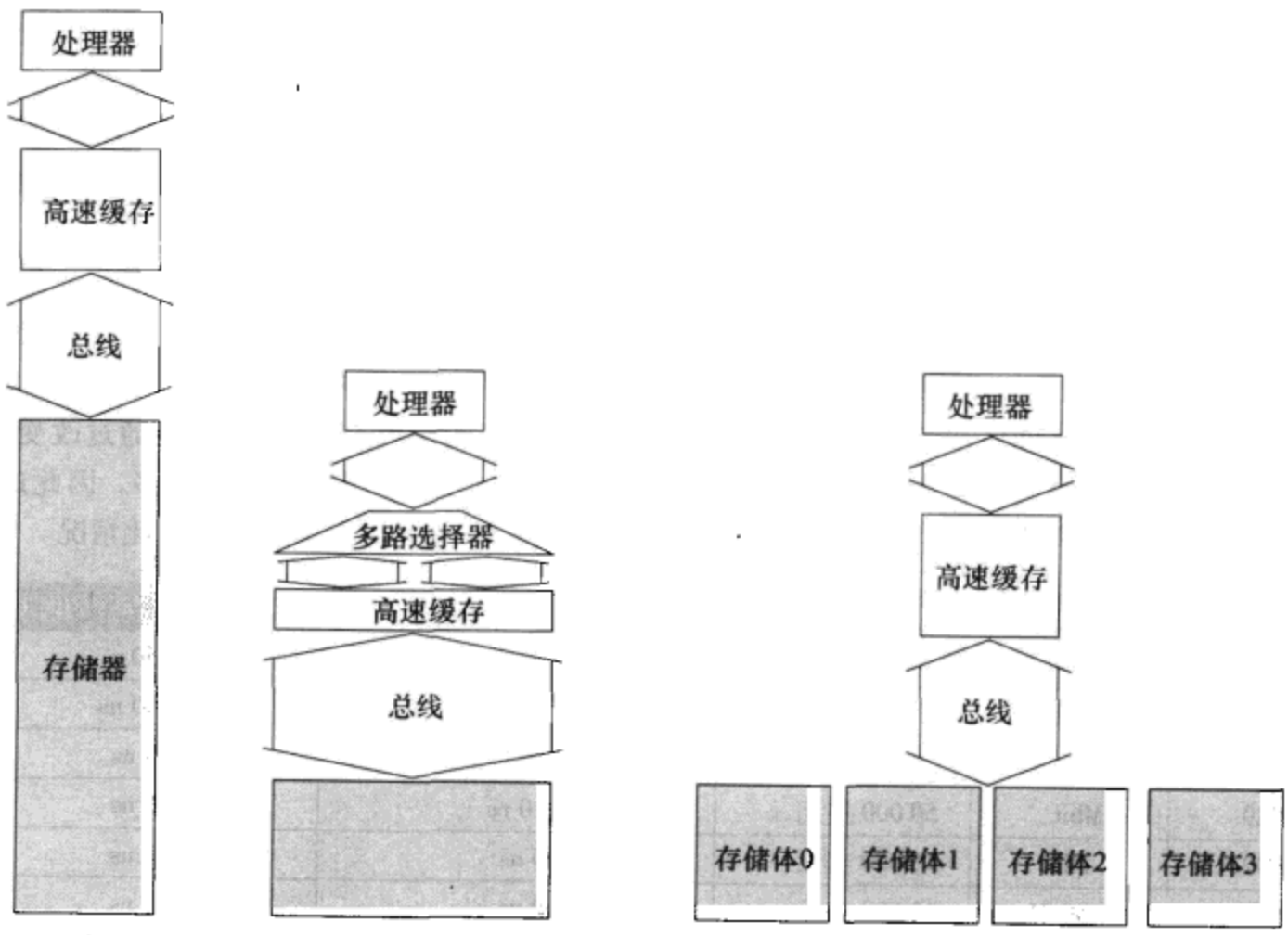


图 5-11　获得高存储器带宽的主要方法是增加存储系统的物理或逻辑宽度

在本图中，通过两种方法来增加存储器带宽。最简单的设计是 a，它使用一个所有部件都是一个字宽的存储器；b 显示了一个更宽的存储器、总线和 cache；c 使用一个窄的总线，以及一个交叉存取的存储器。在 b 中，cache 和处理器之间的逻辑包括一个用于读操作的多路选择器以及用于写操作更新 cache 中相应字的控制逻辑。

增加存储器和总线的位宽会相应地增加存储器的带宽，减少存取时间和传输时间，从而降低缺失代价。当主存位宽为两个字，缺失代价从 65 个存储器总线时钟周期降为 $1 + (2 \times 15) + 2 \times 1 = 33$ 个存储器总线时钟周期。对于一个宽度为 2 个字的存储器来说，一次缺失的带宽是 0.48 字节/总线时钟周期（几乎是 1 字宽的 2 倍）。这种性能提高的主要代价是使用了宽总线，以及在处理器和 cache 之间使用多路选择器和控制逻辑后带来潜在的 cache 访问时间增加。

相对于加宽存储器和 cache 之间的整个路径，存储器芯片可以被组织成多个存储体（bank），这样每次访问就可以同时读/写多个字而不是每次只能读/写一个字。每个存储体为一个字宽，这样，总线和 cache 的宽度就无需改变，但是要把一个地址传到多个存储体则需要允许它们同时读。这种技术称为交叉存取（interleaving），它保留了整个存储器只有一次延迟的好处。例如，存储器有 4 个存储体，取 1 个 4 字的块的时间包括：1 个时钟周期用来传送地址和读请求到各个存储体，15 个时钟周期用于 4 个存储体访问存储器，4 个时钟周期用来把 4 个字送入 cache。一次缺失代价为：$1 + (1 \times 15) + 4 \times 1 = 20$ 个存储器总线时钟周期。这样每次缺失后每时钟周期的有效带宽为 0.80 字节，大约是一个字宽的存储器和总线的带宽的 3 倍。存储体对写操作也很有效。每个存储体可以独立执行写操作，使得写操作时带宽为原先的 4 倍，并减少了在写直达 cache 中遇到的阻塞的情况。我们将看到，对写操作使用另一种策略会使交叉存取技术更具吸引力。

由于 cache 无处不在以及人们对大数据块的渴望，DRAM 制造商规定了对 DRAM 中一连串顺序地址的突发访问。最新的发展是*双倍速率动态随机存储器*（Double Data Rate DRAM）。其名字的意思是在时钟的上升沿和下降沿各传输一次数据，因此，得到的带宽是基于时钟频率和数据宽度所能得到的带宽的两倍。为了分配如此高的带宽，内部的 DRAM 就被组织成为多体交叉访问存储体结构。

这些优化的好处在于它们保留了大多已经在 DRAM 中存在的电路，只是稍微增加了一些系统成本，带宽就获得了明显的提高。DRAM 的内部结构以及这些优化是如何实现的将在附录 C（见光盘）的 C.9 节予以说明。

精解： 存储器芯片组织起来有很多输出位，通常为 4 ~ 32 位，在 2008 年最常见的是 16 位。我们用 $d \times w$ 来描述这种 RAM 的结构，其中，d 是可寻址位置的数目（也就是深度），w 是输出位数（每个位置的宽度）。逻辑上，DRAM 被组织成矩形阵列，访问时间被分为行访问时间和列访问时间。DRAM 可以缓冲一行。突发传输可以重复访问缓冲区而不占用行访问时间。缓冲区的作用类似于 SRAM，通过改变列地址，可以随机访问缓冲区的任一位，直到访问下一行。由于对行中每一位的访问时间减少很多，因此这种方法明显改善了访问时间。图 5-12 说明了近年来 DRAM 的存储密度、成本以及访问时间的变化情况。

推出年份	芯片地址空间	每 GB 的价格	访问新的一行/列的时间	访问现存行的时间
1980	64 Kbit	1 500 000 美元	250 ns	150 s
1983	256 Kbit	500 000 美元	185 ns	100 ns
1985	1 Mbit	200 000 美元	135 ns	40 ns
1989	4 Mbit	50 000 美元	110 ns	40 ns
1992	16 Mbit	15 000 美元	90 ns	30 ns
1996	64 Mbit	10 000 美元	60 ns	12 ns
1998	128 Mbit	4 000 美元	60 ns	10 ns
2000	256 Mbit	1 000 美元	55 ns	7 ns
2004	512 Mbit	250 美元	50 ns	5 ns
2007	1 Gbit	50 美元	40 ns	1.25 ns

图 5-12　在 1996 年之前，DRAM 的容量每三年大概就增为原来的 4 倍，此后增长速度就很慢了

访问时间的改进尽管很慢但是却从未停止，并且价格几乎紧随着存储密度的提高而改变，尽管价格也受其他因素影响，如供应量和需求量，但每 GB 的价格并没有因通货膨胀而调整。

为了改进处理器的接口，DRAM 增加了时钟，*严格意义上被称为同步动态随机存取存储器*（SDRAM）。SDRAM 的优点在于使用了统一的时钟从而减少了存储器和处理器同步的时间。

精解： 一种评估 cache 下层的存储系统性能的方法是使用流基准测试程序［McCalpin，1995］。它评估了长向量操作的性能。它们不具备时间局部性，并且比起被测试计算机的 cache，它们访问的是更大的阵列。

精解： DDR 存储器所使用的突发模式同样被应用于存储器总线，如 Intel Duo Core 的前端总线。

5.2.6　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。

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。

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？

A. 存储器延迟越短，cache 块越小。

B. 存储器延迟越短，cache 块越大。

C. 存储器带宽越高，cache 块越小。

D. 存储器带宽越高，cache 块越大。

5.3　cache 性能的评估和改进

在这一节中，我们首先探讨评测和分析 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 美元的高端计算机中，此后该技术被广泛应用于台式计算机中，而售价已不到 500 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 + 存储器阻塞的时钟周期）× 时钟周期

我们假设存储器阻塞的时钟周期主要来自于 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 = 读操作引起阻塞的时钟周期 + 写操作引起阻塞的时钟周期

读操作阻塞的时钟周期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 5.2.3 节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为这两种情况阻塞的时钟周期之和。

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，说明它设计得不够好；设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 5.5 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：

$$\text{存储器阻塞时钟周期} = (\text{存储器访问次数}/\text{程序数}) \times \text{缺失率} \times \text{缺失代价}$$

也可以表示如下：

$$\text{存储器阻塞时钟周期} = (\text{指令数}/\text{程序数}) \times (\text{缺失数}/\text{指令}) \times \text{缺失代价}$$

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。

举例　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%。

根据指令计数器（I），由指令缺失引起的时钟周期损失数为

$$\text{指令缺失时钟周期} = \text{I} \times 2\% \times 100 = 2.00 \times \text{I}$$

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：

$$\text{数据缺失时钟周期} = \text{I} \times 36\% \times 4\% \times 100 = 1.44 \times \text{I}$$

总的存储器阻塞时钟周期为 $2.00 \times \text{I} + 1.44 \times \text{I} = 3.44 \times \text{I}$，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

$$\frac{\text{有阻塞的 CPU 执行时间}}{\text{配置理想 cache 的 CPU 执行时间}} = \frac{\text{I} \times \text{CPI}_{\text{阻塞}} \times \text{时钟周期}}{\text{I} \times \text{CPI}_{\text{理想}} \times \text{时钟周期}} = \frac{\text{CPI}_{\text{阻塞}}}{\text{CPI}_{\text{理想}}} = \frac{5.44}{2}$$

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，那样又会发生什么？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 3 m^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，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：

$$\text{AMAT} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

举例　计算平均存储器访问时间

处理器时钟周期的时间为 1 ns，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。

每条指令的平均存储器访问时间为

$$\begin{aligned}\text{AMAT} &= \text{命中时间} + \text{缺失率} \times \text{缺失代价}\\ &= 1 + 0.05 \times 20\\ &= 2\ \text{个时钟周期}\end{aligned}$$

即 2 ns。

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 5.11 节中我们将给出其他的例子。

5.3.1　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*（direct mapped），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。

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**[⊖]，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**[⊜]。在组相联 cache 中，每个块可被放置的位置数是固定的（至少两个）。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，组相联映像将直接映射和全相联映像结合起来：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 5-13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。

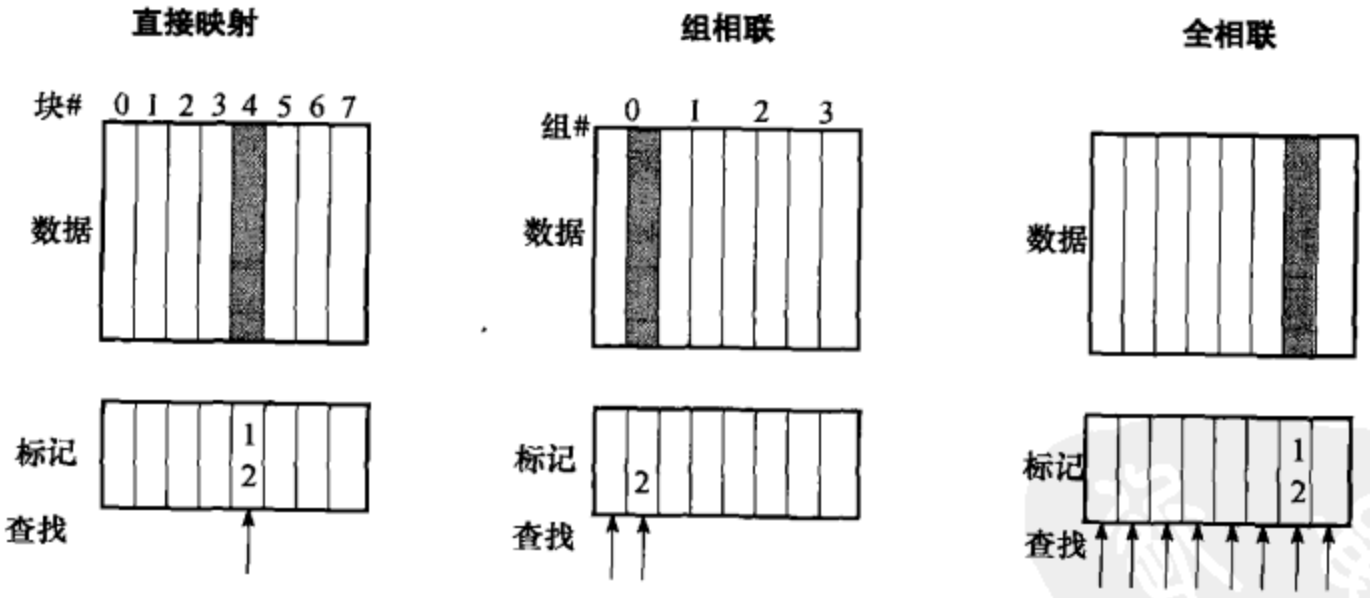


图 5-13　地址为 12 的主存块在 cache 中的位置，cache 容量为 8 块，采用直接映射、组相联以及全相联机制

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（12 mod 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。

回想直接映射的 cache，一个存储块的位置是这样给出的：

⊖ 全相联 cache（fully associative 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（set-associative 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）。

（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 5-14 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 1 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。

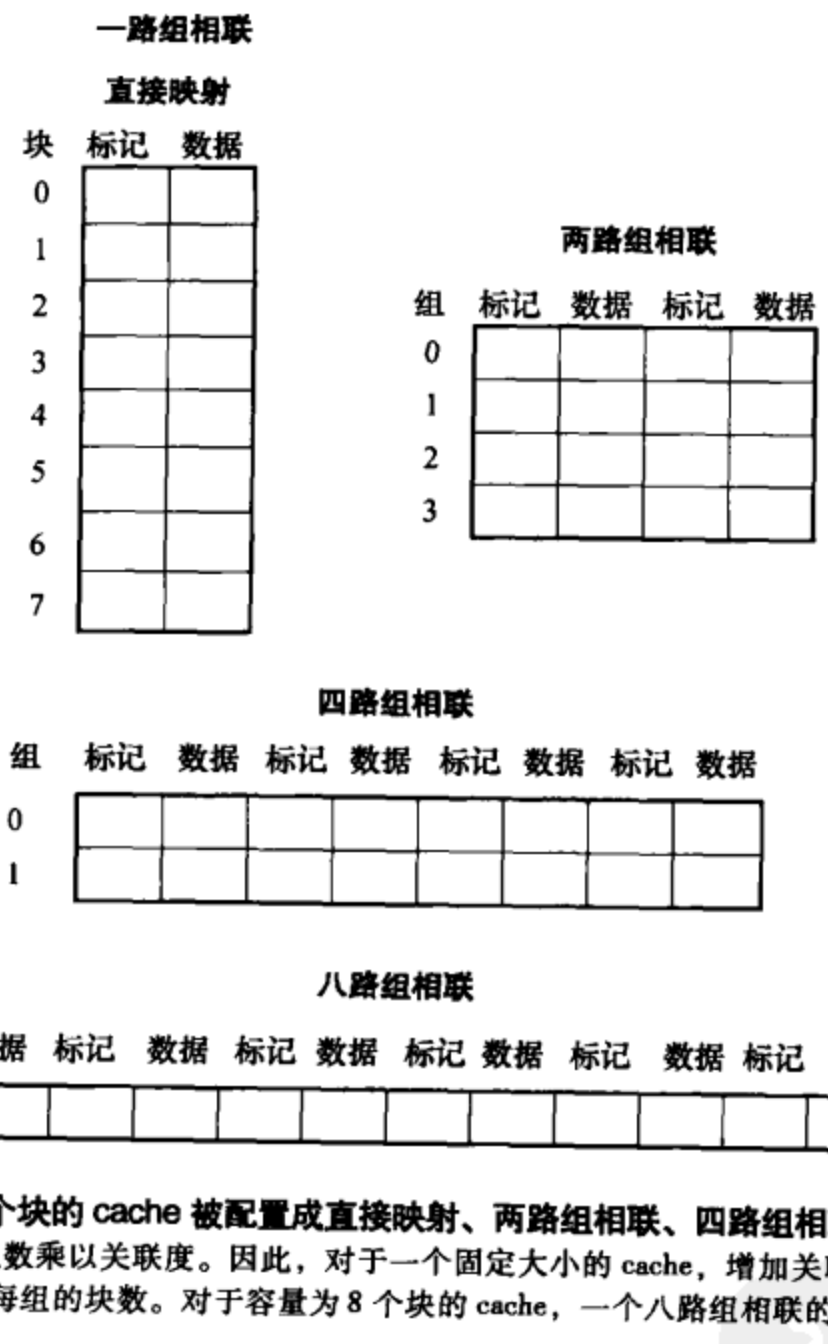


图 5-14　一个拥有 8 个块的 cache 被配置成直接映射、两路组相联、四路组相联以及全相联结构

cache 中块的总数等于组数乘以关联度。因此，对于一个固定大小的 cache，增加关联度的同时也就减少了组数，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。

提高关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。

举例　**cache 的缺失与关联度**

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到 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

由于当缺失时，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被使用的块；也就是说，在过去最久的时间被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：

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 0	块 1	块 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［8］	**主存［6］**	
8	命中	主存［0］	主存［6］	主存［6］	

对于这一系列的访问，三次缺失是可能得到的最好结果，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 容量和关联度不能分开考虑。

关联度能使缺失率下降多少呢？图 5-15 显示了一个容量为 64 KB，块大小为 16 字的数据 cache，当关联度从直接映射到 8 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%，但是更高的关联度对缺失率的改善就很小了。

5.3.2　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被检测，从而判断是否和来自处理器的块地址相匹配。图 5-16 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查看。由于速度是最根本的，被选中的组中所有块的标记是并行检测的。就像在全相联 cache 中一样，组相联 cache 使用顺序检测将使得命中时间太长。

关联度	数据缺失率
1	10.3%
2	8.6%
4	8.3%
8	8.1%

图 5-15　使用与内置 FastMATH 处理器相似的 cache 结构，关联度从一路到八路，采用 SPEC2000 基准测试程序测出的数据 cache 缺失率

10 个 SPEC2000 测试程序的结果来自 Hennessy 和 Patterson［2003］。

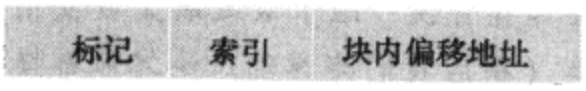


图 5-16　组相联或者直接映射 cache 中地址的三个组成部分

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，块内偏移地址是块中被请求数据的地址。

如果 cache 总容量保持不变，提高关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：关联度每增加到两倍就会使每组块数加倍而使组数减半。相应地，关联度每增加到两倍，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，除了块内偏移地址，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 5-17 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。

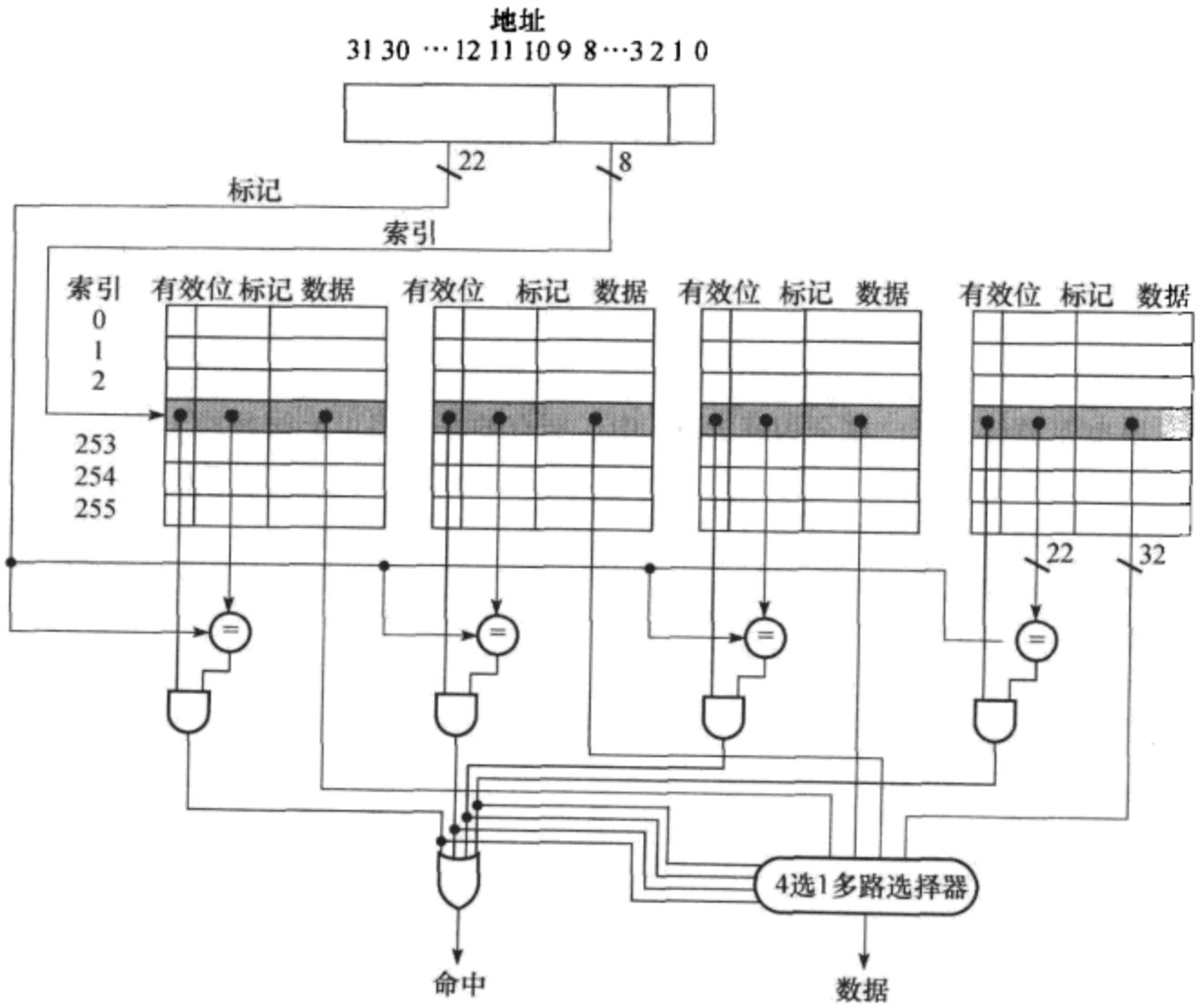


图 5-17　实现一个四路组相联的 cache 需要 4 个比较器和一个 4 选 1 的多路选择器

比较器用来判断被选中的组中哪一个单元（如果有的话）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 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，需要在缺失代价和关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 那样根据地址读数据，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行的索引。CAM 的出现意味着设计者能提供更高的关联度，这比在 SRAM 和比较器之外还需要构建硬件才能实现的关联度还要高。在 2008 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。

举例　**标记位大小与组相联**

提高关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 K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 $\log_2(4\text{ K})=12$，因此有 12 位是索引位；因此总的标记位数是 $(28-12)\times 4\text{ K}=16\times 4\text{ K}=64\text{ K}$ 位。

关联度每增加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 K个组，总的标记位数为（28 - 11）×2 × 2 K = 34 × 2 K = 68 K位。而四路组相联中组数为1 K，那么总的标记位数为（28 - 10）×4 × 1 K = 72 × 1 K = 72 K位。

对于全相联cache，只有一个有4 K个块的组，标记位是28位，因此总的标记位数是28 × 4 K × 1 = 112 K位。

5.3.3　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。在关联的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**最少使用法（LRU）**[⊖]，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）。

LRU替换算法的实现是通过追踪每一块的相对使用情况。对于一个两路组相联cache，追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。当关联度提高时，LRU的执行就变得困难些；在第5.5节中，我们将会讨论另一种替换机制。

5.3.4　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。

举例　**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4 GHz。假设主存访问时间为100 ns，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5 ns，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

$$\frac{100\ \text{ns}}{0.25\ \frac{\text{ns}}{\text{时钟周期}}} = 400\ \text{个时钟周期}$$

只有一级cache的处理器的有效CPI由下列公式给出：

总的CPI = 基本CPI + 每条指令的存储器阻塞时钟周期

因此，在本例中，只有一级cache时，

总的CPI = 1.0 + 每条指令的存储器阻塞时钟周期 = 1.0 + 2% × 400 = 9

⊖ 最近最少使用法（LRU）（least recently used）：一种替换策略，总是替换很长时间没有被使用的块。

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

$$\frac{5\ \text{ns}}{0.25\ \frac{\text{ns}}{\text{时钟周期}}} = 20\ \text{个时钟周期}$$

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

$$= 1 + 2\% \times 20 + 0.5\% \times 400 = 1 + 0.4 + 2.0 = 3.4$$

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；而必须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**[㊀]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 更高的关联度以减少缺失率。

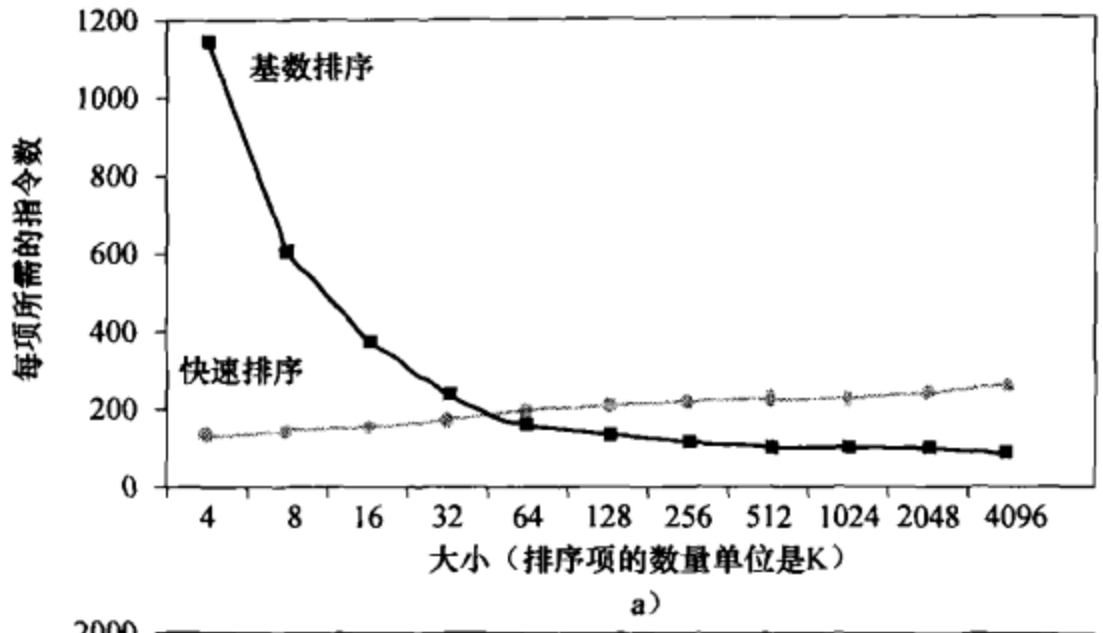
理解程序性能

我们用尽一切方法对冒泡排序（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 5-18a 说明了使用基数排序和快速排序时，指令执行的情况。果然，对于大的数组，在操作次数上，基数排序比快速排序要有优势。图 5-18b 是每项平均所需的时间，而不是执行的指令数。我们可以看到开始两条曲线的轨迹与图 5-18a 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 5-18c 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。

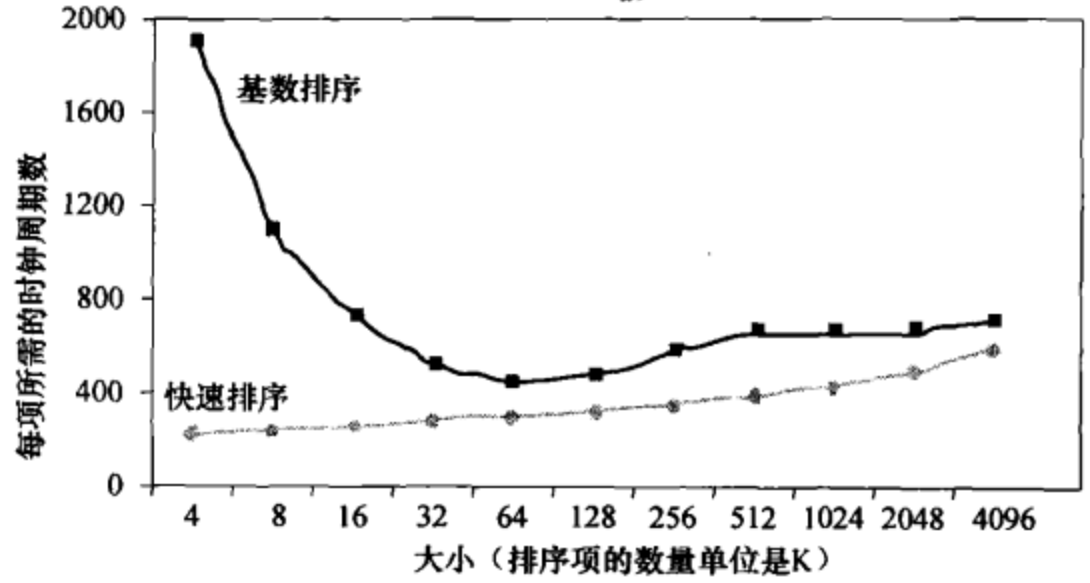
标准算法分析通常会忽视存储器层次结构的影响，正如更快的时钟频率和摩尔定律让体系结构设计者从指令流中获取所有的性能，合理地使用存储器层次结构是获得高性能的关键。如我们在概述中所说的，理解存储器层次结构的行为对于理解当今计算机的程序性能是十分关键的。

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“cache 的缺失与关联度”的例子中，我们看见了一级 cache 缺失率以及**全局缺失率**[㊁]，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**[㊂]。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“cache 的缺失与关联度”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。

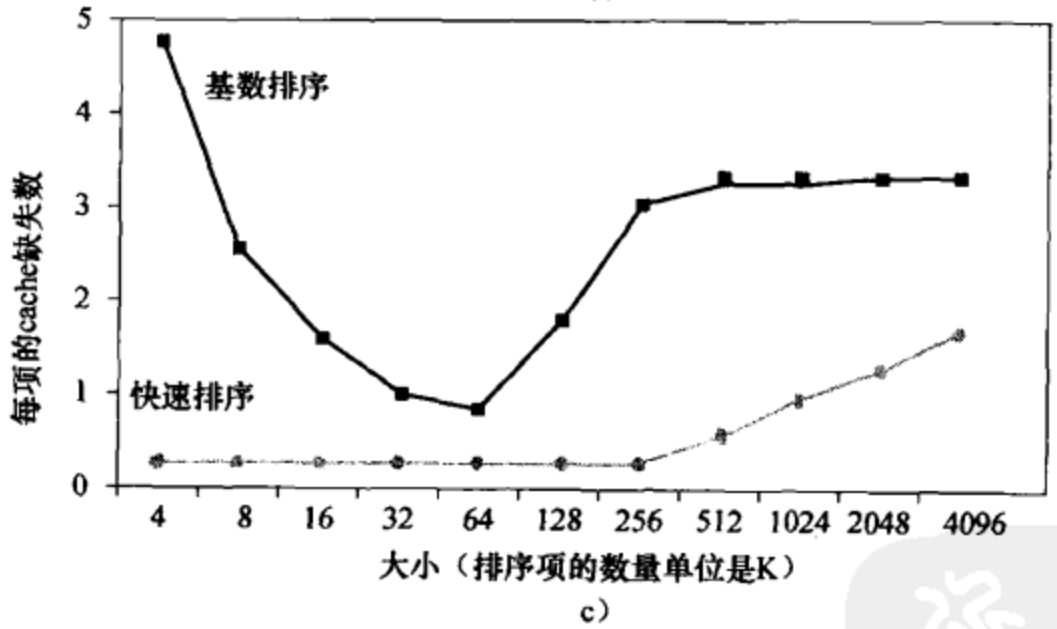
㊀ 多级 cache（*multilevel cache*）：存储系统由多级 cache 组成，而不仅仅只有主存和一个 cache。
㊁ 全局缺失率（global miss rate）：在多级 cache 的所有级中都缺失的那部分访问。
㊂ 局部缺失率（local miss rate）：在多级 cache 中，某一级 cache 的缺失率。



a）



b）



c）

图 5-18　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。尽管在更新的处理器上数值会有变化，但是结论不变。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 5.11 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。

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：

$$\frac{\text{存储器阻塞周期数}}{\text{指令数}} = \frac{\text{缺失数}}{\text{指令数}} \times (\text{总的缺失延迟} - \text{重叠的缺失延迟})$$

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，但是却很少隐藏二级 cache 的缺失代价。

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、关联度、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。为了复制这些变化，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）。

小测验

有关多级 cache 的设计，下面哪些是正确的？

A. 一级 cache 更关注命中时间，二级 cache 则更关注缺失率。

B. 一级 cache 更关注缺失率，二级 cache 则更关注命中时间。

5.3.5　小结

在这一节中，我们集中讨论了三个主题：cache 性能、利用关联度来降低缺失率、利用多级 cache 结构来降低缺失代价。

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。在第 5.5 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，我们对关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 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 是不切实际的。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。使用何种关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。

最后，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）。和关联度考虑相似，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。

5.4　虚拟存储器

……这样的系统已经被设计：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”**[⊖]。从历史观点来说，构造虚拟存储器有两个主要动机：允许在多道程序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。40 年后，第一条变成主要设计动机。

考虑一系列程序在一台计算机上同时运行的情况。当然，为了允许多个程序共享同一个存储器，我们必须保护每个程序，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，局部性原理也造就了虚拟存储器，虚拟存储器使得我们能更有效地共享处理器和主存。

⊖ 虚拟存储器（virtual memory）：一种将主存用作辅助存储器高速缓存的技术。

在编译的时候，我们不知道哪些程序将和其他程序共享存储器。事实上，当程序在执行的时候，程序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）——存储器中只能由该程序访问的独立的一连串地址。虚拟存储器实现程序地址空间到**物理地址**[㊀]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**。[㊁]

使用虚拟存储器的第二个动机就是允许单用户程序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*覆盖*（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覆盖，并且装载的覆盖不会超过存储器的总容量。传统的覆盖被构造成模块，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，它自动管理由主存（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**[㊂]。在虚拟存储器中，处理器产生一个**虚拟地址**[㊃]，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 5-19 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**[㊄]。如今，由虚拟存储器控制的两级存储器层次结构是 DRAM 和磁盘（见第 1 章）。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。

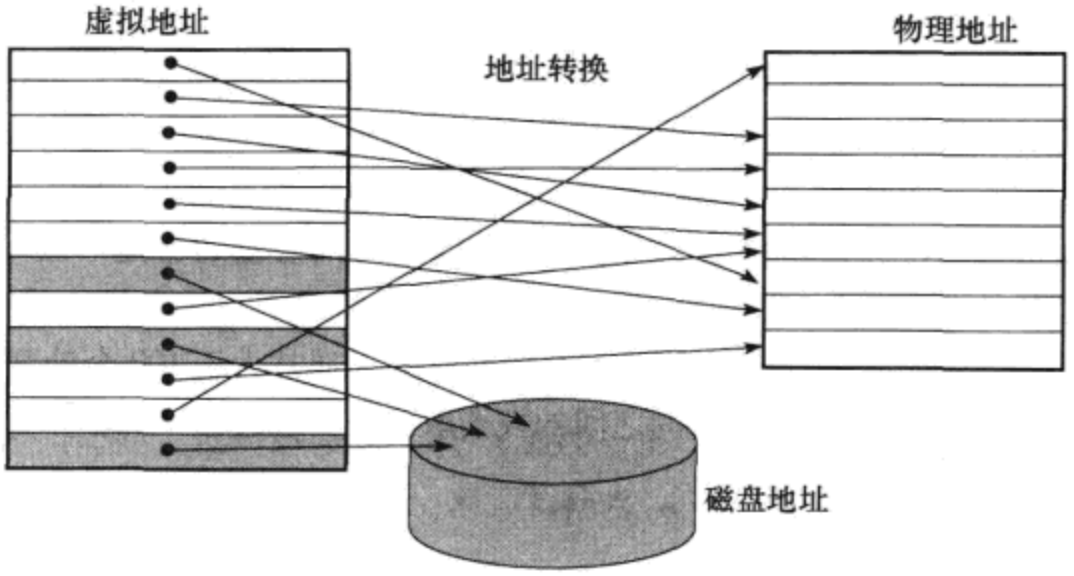


图 5-19　在存储器中，主存中的块（称为页）从一组地址（称为虚拟地址）映射到另一组地址（称为物理地址）

访问主存使用物理地址，而处理器产生虚拟地址。虚拟地址和物理地址都被划分成页，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

㊀ 物理地址（physical address）：主存储器的地址。

㊁ 保护（protection）：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。

㊂ 缺页（page fault）：访问的页不在主存储器中。

㊃ 虚拟地址（virtual address）：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。

㊄ 地址转换（address translation）：也称为地址映射（address mapping）。在访问内存时将虚拟地址映射为物理地址的过程。

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）和页内偏移（page offset）。图 5-20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，而页内偏移是不变的，构成物理地址的低位部分。页内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。

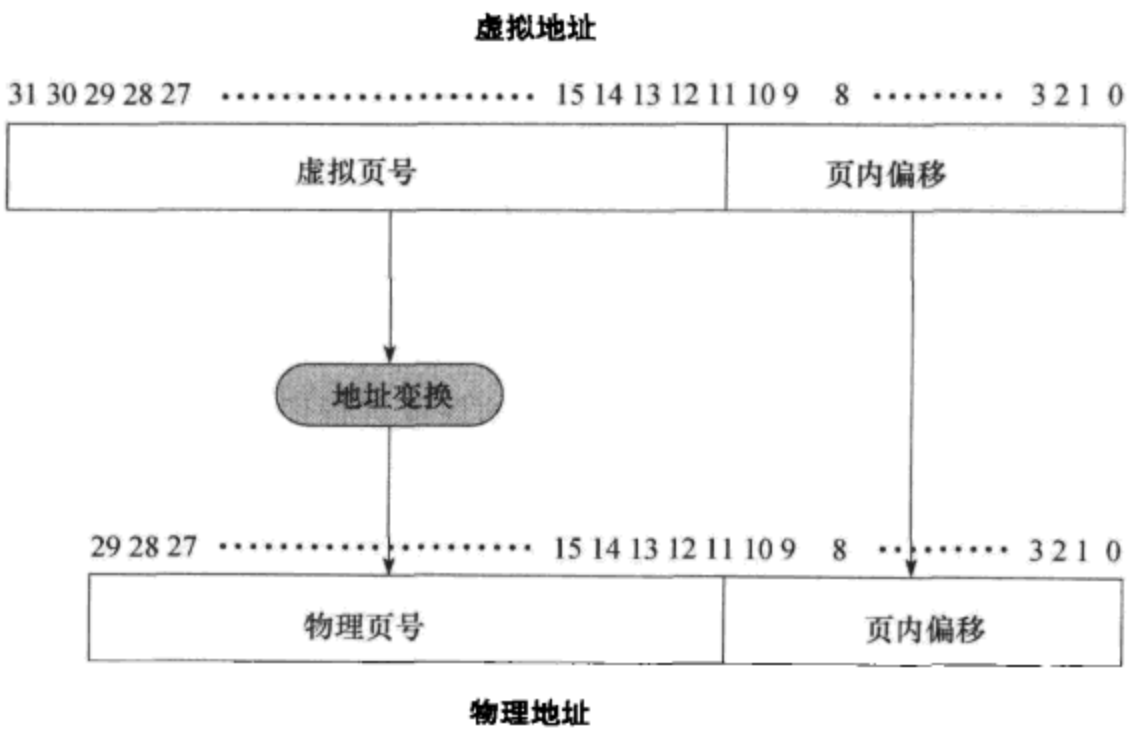


图 5-20　虚拟地址到物理地址的映射

页大小为 2^{12} = 4 KB。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 1 GB 的主存，而虚拟地址空间为 4 GB。

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（5.1 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 KB 到 16 KB。能支持 32 KB 到 64 KB 页的新型台式计算机和服务器正在被研发，但是新的嵌入式系统走的是相反的方向，页大小为 1 KB。
- 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。

下面几节将把这些因素融入到虚拟存储器的设计中去。

精解：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时候，相反的情况也会出现。单个程序不会受益，但是一组程序同时执行就可能因无需交换到主存，或者在

并行处理器上执行而受益。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。

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**[⊖]。在段式存储中，地址由两部分组成：段号和段内偏移。段寄存器被映射到物理地址，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。

5.4.1　页的存放和查找

由于缺页的代价高得惊人，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追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。

正如在 5.3 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**[⊜]，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）。现在假定页表存在于存储器中一个固定的连续区域内。

硬件 软件接口

页表、程序计数器以及寄存器，确定了一个程序的状态（state）。如果我们想让另一个程序使用处理器，我们必须保存该状态。随后，在恢复了该状态之后，程序就可以继续执行。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

⊖ 段式（segmentation）：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。

⊜ 页表（page table）：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。

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。

图 5-21 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 0，该页就不在主存中，就发生一次缺页。如果该位为 1，表明该页在主存中，并且该项包含有物理页号。

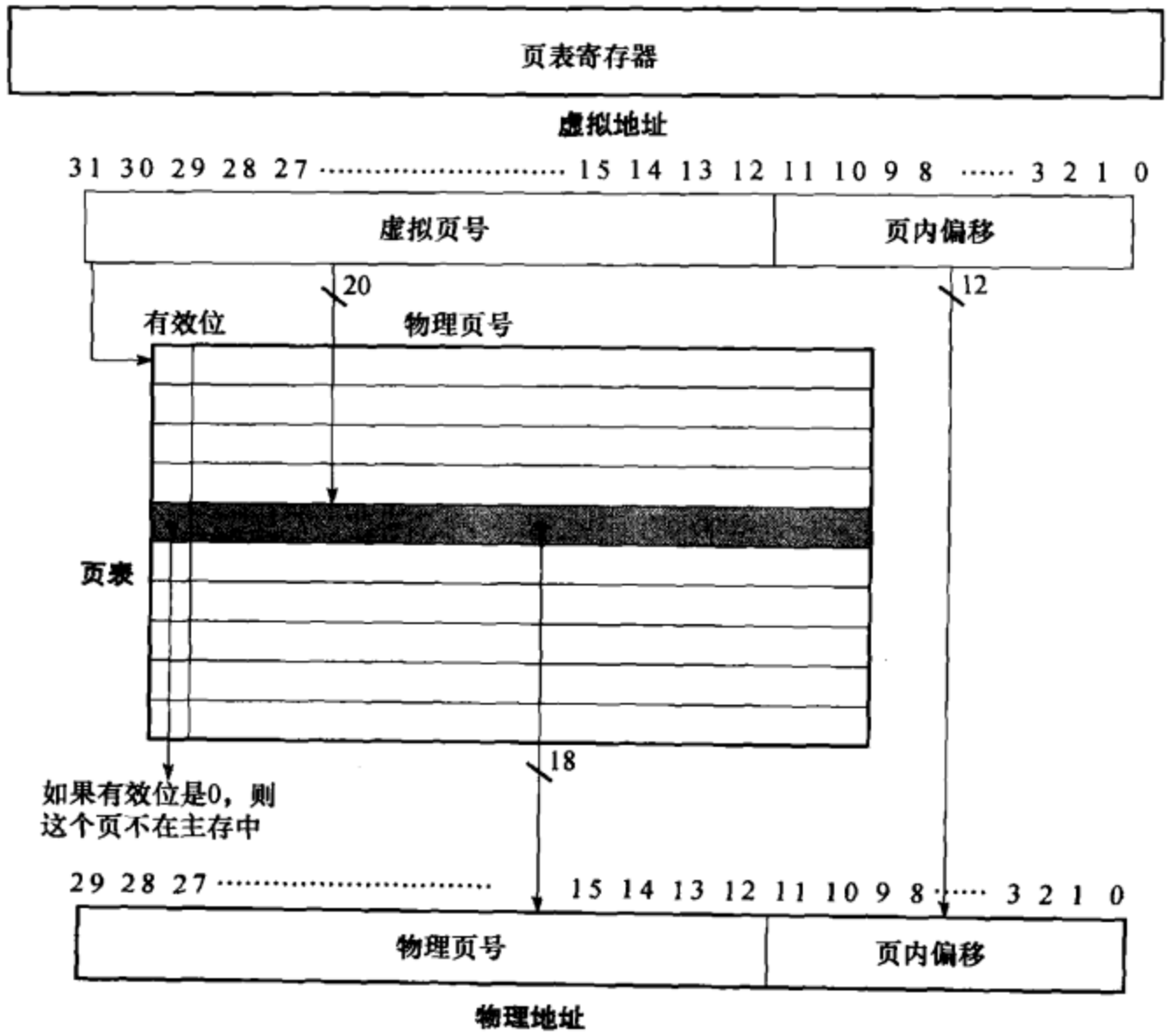


图 5-21　用虚拟页号来索引页表以获得对应的物理地址部分

假定地址为 32 位。页表的首地址由页表指针给出。在本图中，页大小为 2^{12} 字节，即 4 KB，虚拟地址空间为 2^{32} 字节，即 4 GB，物理地址空间为 2^{30} 字节，可以支持高达 1 GB 的主存。页表项数为 2^{20}，即 100 万项。每一项的有效位指出了映射是否合法。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要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。

5.4.2　缺页

如果虚拟页的有效位关闭，就会发生缺页。操作系统获得控制权。控制的转移由异常机制完成，这点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层次（通常是磁盘）中找到该页然后决定将请求页放到主存的什么位置。

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，我们必须保持追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**[⊖]。同时，它也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 5-22 是一个包含物理页号或磁盘地址的单个表的结构。

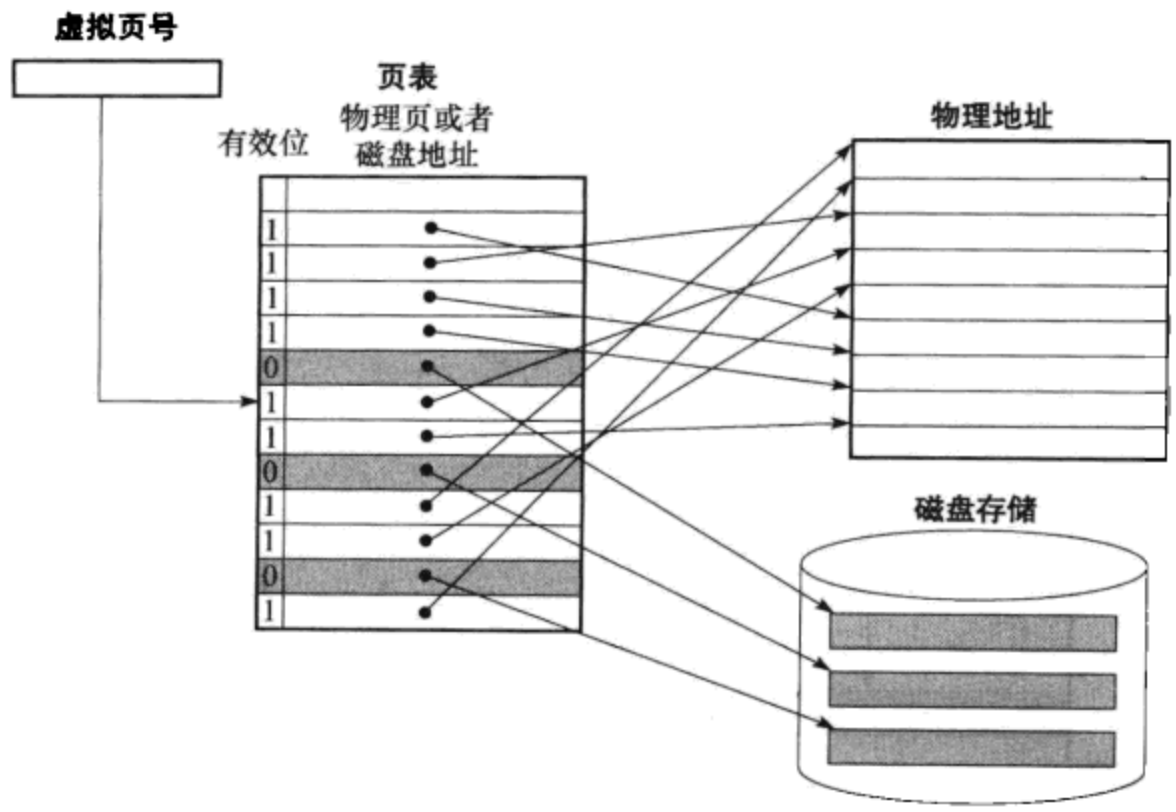


图 5-22　页表将虚拟存储器中的每一页映射到主存中的一页或者存储结构的下一层（磁盘上的一页）

虚拟页号用来检索页表。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地址）。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，它们逻辑上是一个表，但是保存在两个独立的数据结构中。使用双表在某种程度上是正确的，因为我们必须保存所有页的磁盘地址，即使有些页当前不在主存中。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追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，操作系统就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，操作系统遵循我们在 5.3 节中提到的最近最少使用替换策略（LRU）。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。

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追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**[⊜]或者称为**使用位**，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。

⊖ 交换区（swap space）：为进程的全部虚拟地址空间所预留的磁盘空间。

⊜ 引用位（reference bit）：也称为使用位（use bit）。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4 KB，页表每一项为4个字节，我们可以计算总的页表容量为

$$\text{页表项数} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{页表容量} = 2^{20}\text{个页表项} \times 2^2 \frac{\text{字节}}{\text{页表项}} = 4\text{ MB}$$

也就是说，每个程序在执行的任何时候都需要4 MB的存储器空间。对单个程序来说，这个大小并不差。但是如果计算机中同时有成百上千的程序同时执行时，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。下面五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，每一段可以有地址空间的一半大。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5.4节的第二个精解中讨论的段，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稀疏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。

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页。这些大的块有时候被称为段，而第一级的映射表有时被称为段表，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，多数系统也允许页表分页。尽管听起来这很复杂，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。如何克服这些问题都描述地很细节化并且一般因机器而异。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。

5.4.3　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，这种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写，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重写位（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，重写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）。

5.4.4　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：一次访存以获得物理地址，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。当一个转换的虚拟页号被使用时，它可能在不久的将来再次被使用到，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 以追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（TLB）**[⊖]（将其称为地址变换高速缓存更精确）。TLB 就相当于记录卡片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。

如图 5-23 所示，TLB 的每个标记项存放虚拟页号的一部分，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，如重写位和引用位。

每次访问，我们都要在 TLB 中查找虚拟页号。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，重写位同样要被置位。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 缺失会比缺页要频繁得多。

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重写位，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

⊖ 快表（translation-lookaside buffer，TLB）：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。

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 TLB 缺失率有望较低。一些系统使用其他技术来近似引用位和重写位，以消除除了缺失后装入新表项之外写 TLB 的必要。

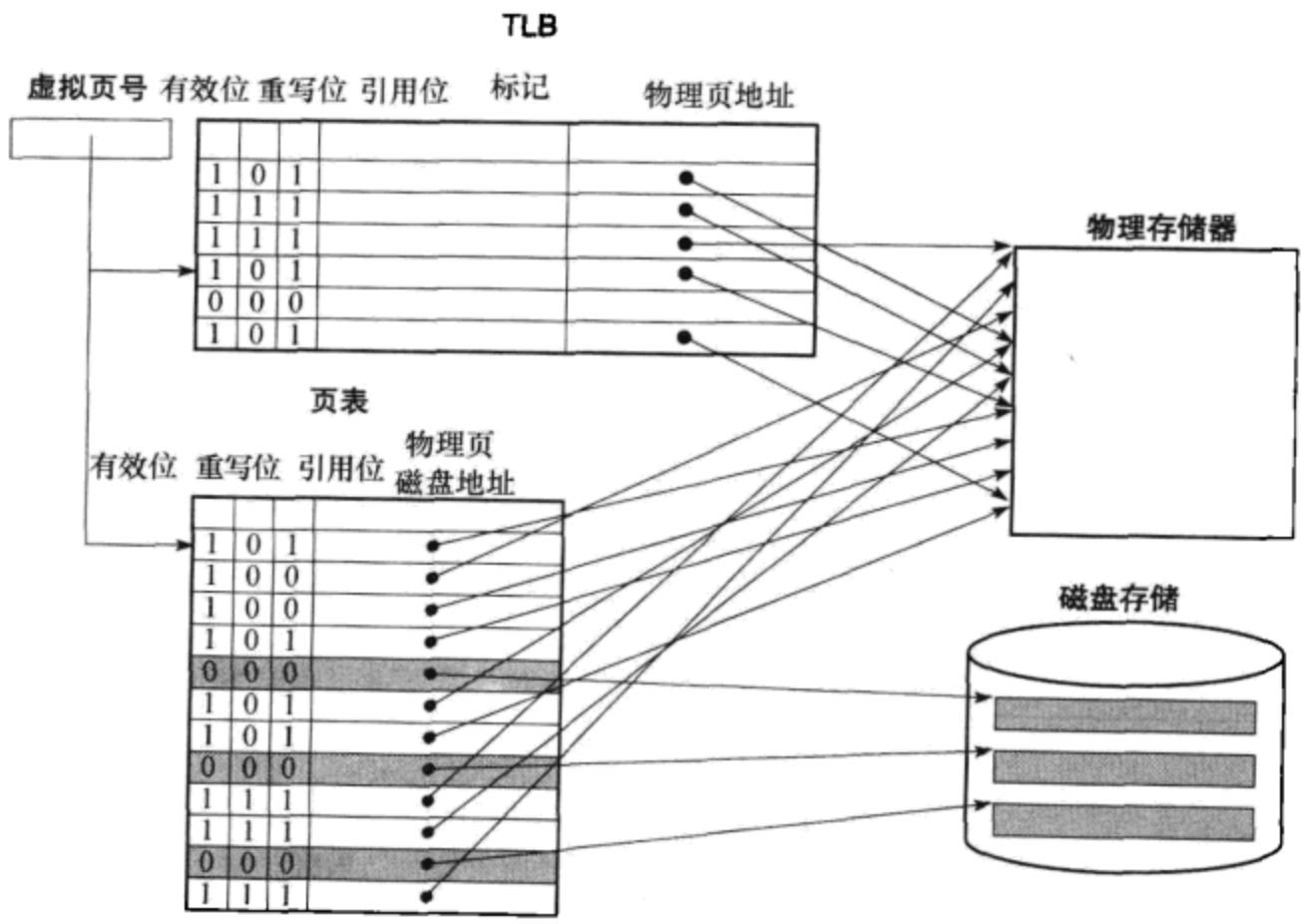


图 5-23　TLB 作为页表的 cache，用于存放映射到物理页中的那些项

TLB 包含了页表中虚页到物理页映射的一个子集。TLB 映射以粗线显示。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。

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 个字节）。
- 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 设计中对关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关联度低且容量大的 TLB。在全相联映射的方式下，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的软件算法。所以很多系统都支持随机地选择替换表项。在 5.5 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 4 KB，地址空间为 32 位，因此，虚拟页号长为 20 位，如图 5-24 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、一个重写位以及一些其他管理操作位。

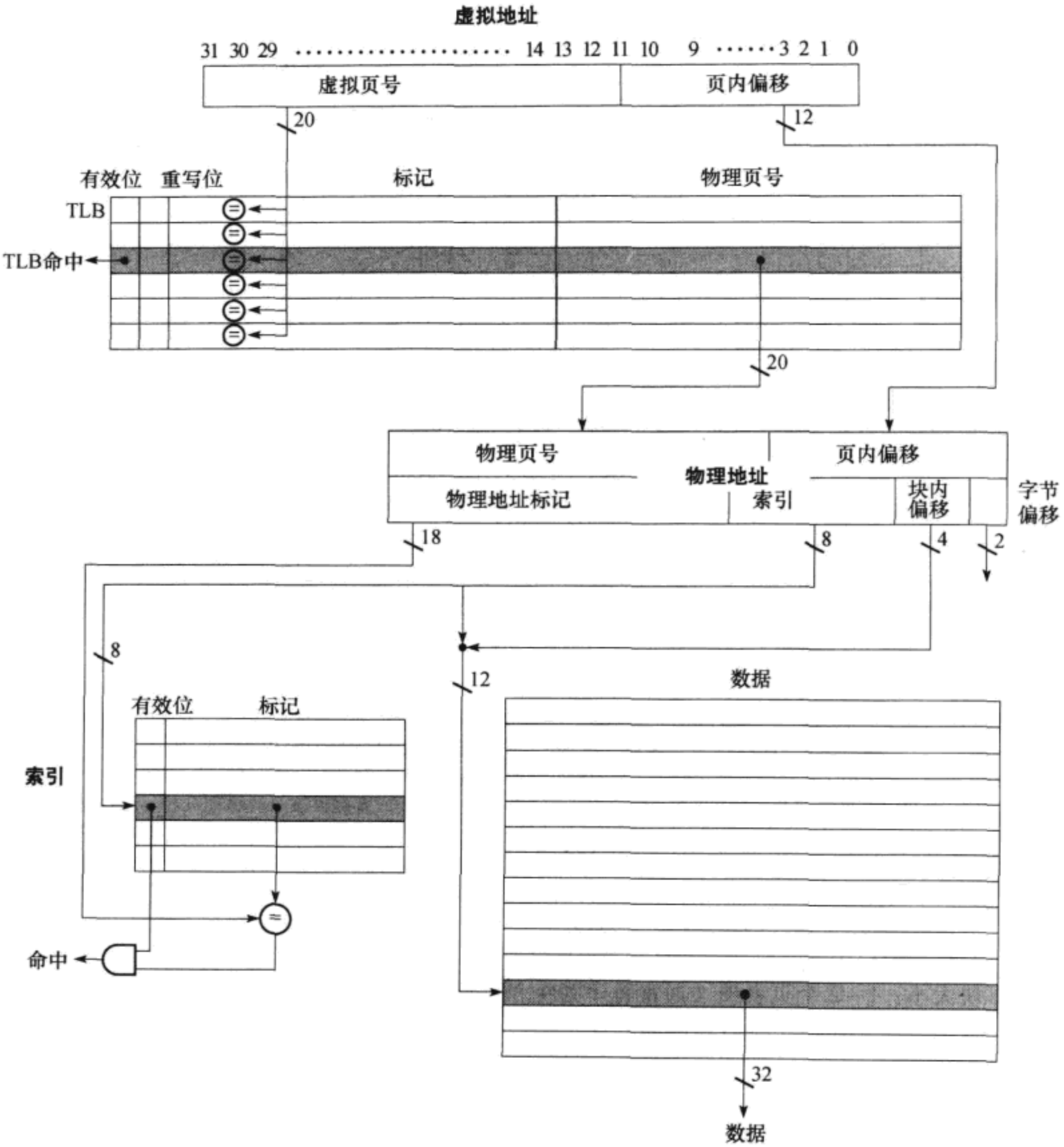


图 5-24　内置 FastMATH 中 TLB 和 cache 实现了从虚拟地址到数据项的转换过程

本图描述了 TLB 和数据 cache 的结构，这里假设页大小是 4 KB。本图主要研究读操作，图 5-25 则描述了如何处理写操作。注意到不同于图 5-9，标记和数据 RAM 是分开的。用 cache 索引和块内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 标记都与虚拟页号进行比较（参考 5.3.2 节精解的内容）。如果匹配表项的有效位为 1，那么 TLB 访问命中，物理页号与页内偏移中的位共同形成访问 cache 的索引。

图 5-24 是 TLB 和一个 cache，图 5-25 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 5.4.7 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。

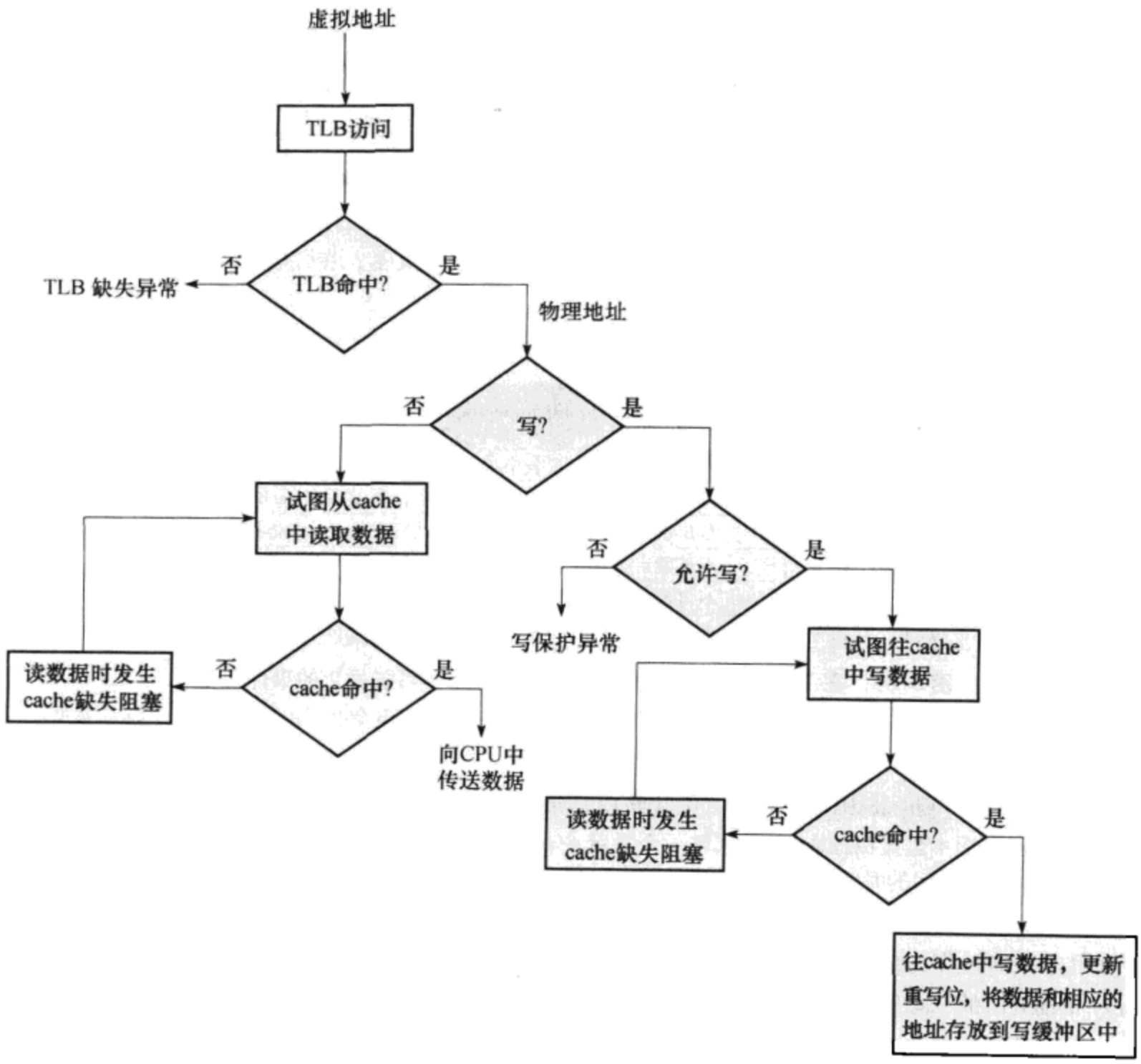


图5-25　在内置 FastMATH 的 TLB 和 cache 中处理读或者写直达操作

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 的重写位置位，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。

5.4.5　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--页移到磁盘上去时，就

从 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。

举例　存储器层次结构的全部操作

在类似于图 5-24 的由一个 TLB 和一个 cache 组成的存储器层次结构中，一次存储器访问可能遭遇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 5-26 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	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 中必找不到数据
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 5-26　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，有三种是不可能的，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）。

精解：图 5-26 假定在访问 cache 之前，所有存储器地址都被转换成物理地址。在这样一种结构中，cache 是物理寻址（physically indexed）并且物理标记（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）。在这样一个系统中，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[⊖]，它使用虚拟地址作为标记；因此这种 cache 是虚拟寻址（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，并且程序之间共享页（程序可能使用不同的虚拟地址访问页）时，就可能有**别名**[⊜]。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。这种多义性就产生一个问题，由于这种页上的一个字可能存在于 cache 中的两个不同位置，每个对应用不同的虚拟地址。这就允许一个程序写数据，而另一个程序并不知道数据已经改变。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能是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内偏移部分，由于没有被转换，因此实际上是物理地址——但使用物理地址标记。这些采用虚拟索

⊖ 虚拟寻址 cache（virtually addressed cache）：一种使用虚拟地址而不是物理地址访问的 cache。

⊜ 别名（aliasing）：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地址时。

引而物理标记的设计，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**[⊖]的简单结构。例如，在这种情况下就没有别名的问题。图 5-24 假定的页大小为 4 KB，但实际上有 16 KB，因此内置 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 大小以及关联度之间进行谨慎的权衡。

5.4.6　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。

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**[⊖]进程、核心进程或者主管进程。

2）提供一部分处理器的状态，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。操作系统用只能在管理态下可用的特殊指令对它们进行写操作。

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**[⊖]异常处理完成的，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。

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

⊖ 物理寻址 cache（physically addressed cache）：使用物理地址寻址的 cache。

⊖ 超级用户管理模式（supervisor mode）：也称作管态、核心模式（kernel mode）。运行操作系统进程的模式。

⊖ 系统调用（system call）：将控制权从用户模式转换到管理员模式的特殊指令，触发进程中的一个异常机制。

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。

精解：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**[⊖]，或者进程切换）时，它必须保证 P2 不能访问 P1 的页表，因为那样不利于保护。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。遗憾的是，P1 随后发现它所有的表项都不见了，因此不得不通过 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。

通常另一种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，由于在进程切换的时候，cache 包含正在执行的进程的数据。对物理寻址和虚拟地址寻址的 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。

5.4.7　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。

怎么知道这两种情况中的哪一种发生了呢？当我们处理缺失时，需要查找页表项并且取回 TLB。如果匹配的页表项的有效位是关闭的，那么对应的页就不在主存中，发生缺页，而不仅仅是 TLB 缺失。如果有效位开启，只需取回所需的表项。

TLB 缺失可以由软件处理，也可以由硬件处理，这是因为只需要短短几步操作就能将一个有效的页表项从存储器中复制到 TLB 中。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，异常程序计数器（*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

⊖ 上下文切换（context switch）：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。

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**[⊖]来避免这种错误的发生。当异常第一次发生时，处理器设置一个位，禁止其他异常的发生；这可以与处理器设置管理态位同时进行。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正确恢复。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 5-27 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。

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 5-27　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。

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，因此它可以很随意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，同样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。

⊖ 使能异常：（*exception enable*）：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。

要使指令**可重新启动**[⊖]，这样异常可以被处理，指令稍后也能继续执行，这在类似于 MIPS 的结构中实现相对简单。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。

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。

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 $80\,000\,000_{16}$，TLB 缺失**处理程序**[⊖]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:
 mcf0     $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
```

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 $8000\ 0180_{16}$，与 TLB 缺失处理程序不相同的地址。这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。由于异常处理程序不经常使用浮点寄存器，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。

图 5-28 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。

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。

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**非映射的**[⊜]。操作系统将异常入口点代码和异常堆栈存

⊖ 可重启指令（restartable instruction）：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。
⊖ 处理程序（handler）：用于“处理”异常或中断的软件程序的名字。
⊜ 非映射的（unmapped）：地址空间中的一个部分，在这个区域不会导致缺页异常。

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，虚拟地址的高位被忽略不计，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。

保存状态			
保存 GPR	addi	$k1, $sp, -XCPSIZE	#堆栈中用于保存状态的空间
	sw	$sp,XCT_SP($k1)	#将 $sp 保存到堆栈
	sw	$v0,XCT_V0($k1)	#将 $v0 保存到堆栈
	…		#将 $v1, $ai, $si, $ti…保存到堆栈
	sw	$ra,XCT_RA($k1)	#将 $ra 保存到堆栈
保存 hi,lo	mfhi	$v0	#复制 Hi
	mflo	$v1	#复制 L0
	sw	$v0,XCT_HI($k1)	#将 Hi 的值保存到堆栈中
	sw	$v1,XCT_LO($k1)	#将 Lo 的值保存到堆栈中
保存异常寄存器	mfc0	$a0, $cr	#复制引发寄存器
	sw	$a0,XCT_CR($k1)	#将 $cr 寄存器的值保存到堆栈中
	…		#保存 $V1 的值
	mfc0	$a3, $sr	#保存状态寄存器
	sw	$a3,XCT_SR($k1)	#将 $sr 保存到堆栈
设置 sp	move	$sp, $k1	#将 sp-XCPSIZE 赋值给 sp 寄存器
允许嵌套异常			
	andi	$v0, $a3,MASK1	#$v0 = $sr&MASK1,异常允许
	mtc0	$v0, $sr	#将 sr 的值设置为允许异常
调用 C 异常处理程序			
设置 gp	move	$gp,GPINIT	#设置 gp,使得它指向堆区域
调用 C 代码	move	$a0, $sp	#arg1 为指向异常堆栈的指针
	jal	xcpt_deliver	#调用 c 代码处理异常
恢复状态			
恢复大多数 GPR，以及 hi、lo 寄存器的值	move	$at, $sp	#寄存器的临时值
	lw	$ra,XCT_RA($at)	#从堆栈中恢复 ra 的值
	…		#恢复 t0…a1 的值
	lw	$a0,XCT_A0($k1)	#恢复 a0 的值
恢复状态寄存器	Iw	$v0,XCT_SR($at)	#从堆栈中读取旧的 sr 值
	li	$v1,MASK2	#使用屏蔽来禁止异常
	and	$v0, $v0, $v1	#设置 $v0 = $sr & MASK2,禁止异常
	mtc0	$v0, $sr	#设置状态寄存器
异常返回			
恢复 $sp 的值，将剩下的 GPR 作为临时寄存器	lw	$sp,XCT_SP($at)	#从堆栈中恢复 sp
	lw	$v0,XCT_V0($at)	#从堆栈中恢复 v0
	lw	$v1,XCT_V1($at)	#从堆栈中恢复 v1
	lw	$k1,XCT_EPC($at)	#从堆栈中复制旧的 epc 值
	lw	$at,XCT_AT($at)	#从堆栈中恢复 at 的值
恢复 ERC 的值，异常返回	mtc0	$k1, $epc	#恢复 epc 的值
	eret	$ra	#恢复被中断的指令

图 5-28　异常时保存状态和恢复状态的 MIPS 码

精解：图5-28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`。

精解：对于有着更为复杂指令的处理器来说，可能会访问主存中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。处理一条指令可能在指令中间产生多次缺页。例如，x86处理器有能访问成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，稍后从执行中间处继续。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。

5.4.8　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，支持多个同时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，这是由于缺页的代价很高。下面一些技术被用来降低缺失率：

1）增大页的容量以便利用空间局部性并降低缺失率。

2）由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。

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追踪一页是否更改过（采用重写位）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。进程之间受控制地共享页可以在操作系统的帮助下实现，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，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。

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，但磁盘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主存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在一个有着更大主存的计算机上运行，或者为你的计算机增加主存。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64×4 KB＝0.25 MB。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 4 KB 的标准页面，MIPS 硬件还支持 16 KB、64 KB、256 KB、1 MB、4 MB、16 MB、64 MB 和 256 MB 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。

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。

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：

A. 一级 cache	a. cache 的 cache
B. 二级 cache	b. 磁盘的 cache
C. 主存	c. 主存的 cache
D. TLB	d. 页表项的 cache

5.5　存储器层次结构的一般架构

到目前为止，我们已经知道了不同类型的存储层次结构共用的许多原理。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 5-29出示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的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四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	15 000 ~ 50 000	16 000 ~ 250 000	40 ~ 1024
以 KB 计量的总容量	16 ~ 64	500 ~ 4 000	1 000 000 ~ 1 000 000 000	0.25 ~ 16
块的字节数	16 ~ 64	64 ~ 128	4000 ~ 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 ~ 1000
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 5-29　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 2008 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。

5.5.1　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	$\frac{\text{cache 中的块数}}{\text{关联度}}$	关联度（一般为 2 ~ 16）
全相联	1	cache 中的块数

增加关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 5-30 显示了不同的 cache 容量，在关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，关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，并且由关联度引起的缺失率的绝对改进明显减少。如前所述，关联度增加的潜在缺点是增加了代价以及访问时间。

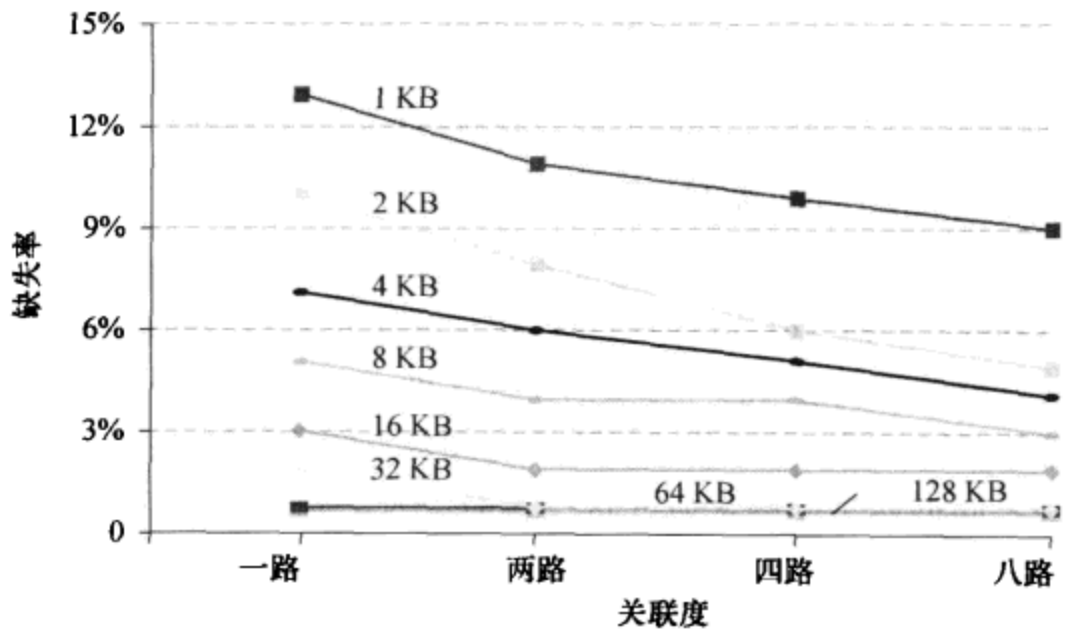


图 5-30　当关联度增加时，8 种不同容量数据 cache 各自的缺失率

从一路（直接映射）到两路组相联变化时获益明显，进一步增加关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，因此从关联度所获得的好处就很明显。图 5-15 解释了这些数据是如何收集的。

5.5.2　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：

关联度	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	关联的度
全相联	查找所有 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关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 允许实现更高的关联度，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 访问时间对决定处理器时钟周期的重要性。

5.5.3　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，在关联度不低（典型的是两路到四路）的层次结构中实现 LRU 的代价太高了，这是因为追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）。

对于更高的关联度，或者用近似的 LRU 算法，或者采用随机替换策略。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。由于缺失的代价是如此高，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。

5.5.4　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）。5.2 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 5.4 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）的延迟很长，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，并且可以通过对 4 个问题的各自解答来理解。

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：有四种方法：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？

答：层次结构中的每一层都可以使用写直达或者写回策略。

5.5.5　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）**[㊀]：

- **强制缺失**[㊁]：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 misses）。
- **容量缺失**[㊂]：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**[㊃]：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）。

图 5-31 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，因此提高关联度就

㊀ 3C（three Cs model）：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。因其三类名称的英文单词首字母均为 c 而得名。

㊁ 强制缺失（compulsory miss）：也称为冷启动缺失（cold - start miss）。对没有在 cache 中出现过的块第一次访问时产生的缺失。

㊂ 容量缺失（capacity miss）：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。

㊃ 冲突缺失（conflict miss）：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。

可以减少冲突缺失。然而，提高关联度会延长访问时间，导致整个性能的降低。

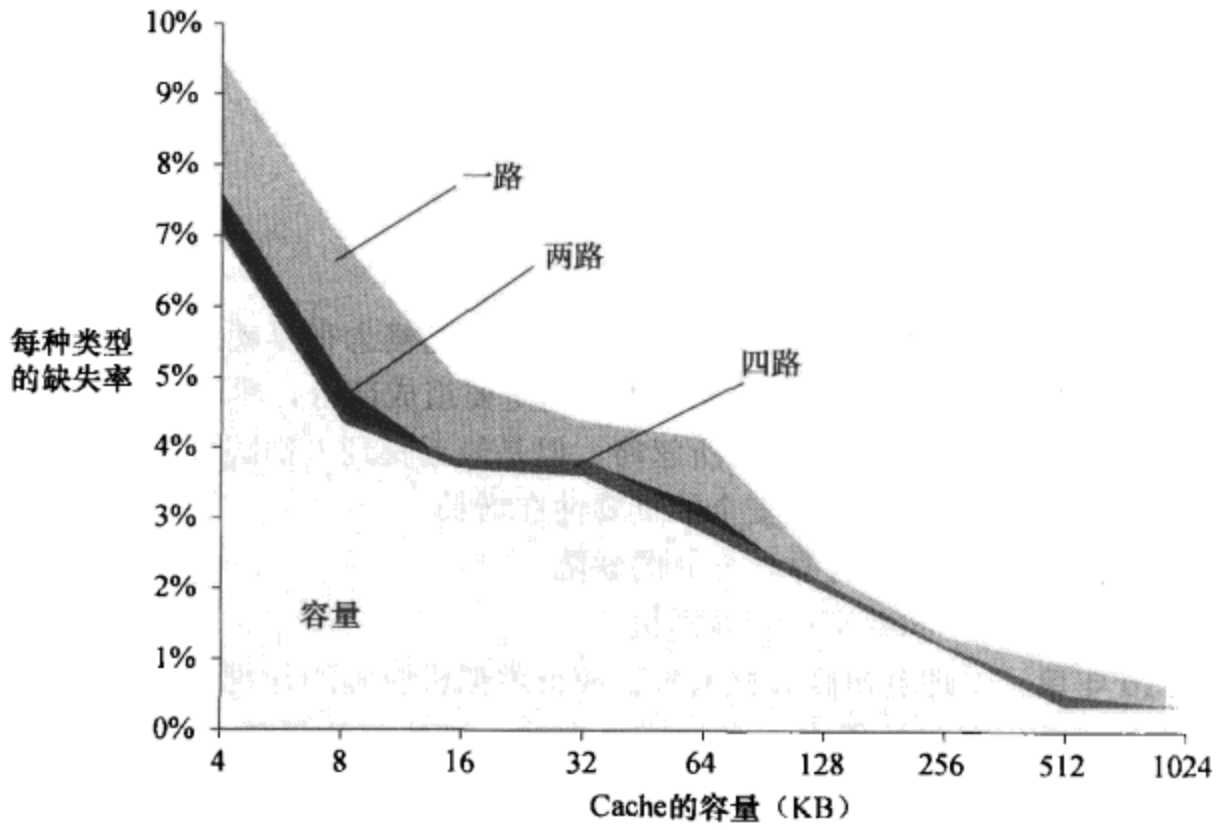


图 5-31　根据缺失原因缺失率被分成三种

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 5-30 出自同一来源，都是由 SPEC2000 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关联度，又取决于 cache 额容量，图中给出了关联度从一路到八路的冲突缺失率。在每种情况下，当关联度从下一个更高度变化到标记地方的关联度时，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 关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。

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，尽管一级 cache 也在增大，但是增大地非常缓慢。

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。

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一些缺失率的组成部分。尽管有这些缺点，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 5-32 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
提高关联度	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 5-32　存储器层次结构设计面临的挑战

小测验

下面哪些表述（如果有的话）是正确的？

A. 没有减少强制缺失的方法。

B. 全相联 cache 中没有冲突缺失。

C. 在减少缺失方面，关联度比容量更为重要。

5.6　虚拟机

和虚拟存储器一样，虚拟机（Virtual Machine，VM）的概念很早就出现了。虚拟机最早出现于 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的缺陷。
- 在多个不相关的用户间共享一台计算机。
- 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匹配硬件。这样的虚拟机被称为（操作）*系统虚拟机*（system virtual machine），如 IBM VM/370、VMware ESX Server 以及 Xen。

系统虚拟机让用户觉得自己在使用整个计算机，包括操作系统的副本。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。底层的硬件平台被称为*主机*（host），它的资源被*客户端*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，虚拟机也提供了其他两个优势：

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。

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。

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“虚拟化”处理器，VMM 必须能控制一切——访问特权状态、地址转换、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，在计时器中断的情况下，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。依赖计时器中断的客户虚拟机会有由 VMM 提供的一个虚拟计时器和模拟的计时器中断。

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。和上述的页式虚拟存储器相似，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，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 和 MIPS 都是如此。

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/写一个页表指针——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系统结构和操作系统，但是随着虚拟存储器的广泛使用，系统结构设计者需要对指令集系统结构中一些不方便使用的细节进行修改。例如，为了在 IBM 370 中支持虚拟存储器，系统结构设计者不得不改变了仅发布了 6 年的 IBM 360 指令集系统结构。如今为了适应虚拟机，也需要做相应的调整。

例如，x86 的指令集 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。系统结构和 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。

精解：除了虚拟化指令集，另一个难题是虚拟存储器的虚拟化，令运行在每个虚拟机上的每个客户操作系统管理各自的页表。为此，VMM 将*实际存储器*（real memory）和*物理存储器*（physical memory）的概念区分开来（经常被认为是同义的），并单独将实际存储器作为虚拟存储器和物理存储器中间一层（有些人使用*虚拟存储器*、*物理存储器*和*机器存储器*的概念来命名这三层）。客户操作系统通过使用页表将虚拟存储器映射到实际存储器，VMM 页表又将用户的实际存储器映射到物理存储器。这种虚拟存储器系统结构或者由页表来定义，如 IBM VM/370 和 x86；或者由 TLB 结构定义，如 MIPS。

VMM 维持一个*影子页表*，以用来将客户虚拟地址空间直接映射到硬件物理地址空间，这样就不需要在每次间接访问存储器时花费额外的开销。通过检查对客户页表的所有修改，VMM 可以确保被硬件用作页表转换的影子页表项和客户操作系统环境之间的对应关系，除了用正确的物理页代替客户页表中实际页的情况。因此，VMM 必须中断任何试图通过客户操作系统改变页表或者访问页表的指针。这通常由写保护客户页表来实现，并且 trap 中断任何通过客户操作系统来访问页表指针的操作。如前文所述，如果访问页表指针是一个特权操作，那么后面发生的 trap 中断是自然的。

系统结构中最后需要虚拟化的部分是 I/O。由于附属于计算机的 I/O 数量增加，并且 I/O 设备类型增加，因此，到目前为止，它是系统虚拟化最难的部分。另一个难点是在多个虚拟机之间共享一个实际设备。然而还有支持多种设备驱动需求的问题，特别是在同一个虚拟机系统上支持不同的客户操作系统。虚拟机可以这样理解：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。

5.7　使用有限状态机来控制简单的 cache

现在我们可以实现对 cache 的控制，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。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。CD 中的 5.9 节用一种新的硬件描述语言更深入地介绍了 cache 和控制器。

5.7.1　一个简单的 cache

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 16 KB，因此它能容纳 1024 个块。
- 32 位字节地址。
- cache 中每个块包含一个有效位和重写位。

根据5.2节，我们可以计算出cache的地址域：

- cache索引位为10位。
- 块内偏移为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。

注意到这是一个阻塞式cache，因此处理器必须等到cache处理完请求之后才能继续执行。

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位的数据，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。

5.7.2　有限状态机

为了给单周期的数据通路设计控制部件，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**[⊖]，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**[⊖]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。

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

⊖ 有限状态机（finite-state *machine*）：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（next-state function）：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。

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 C（见光盘）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 C（见光盘）。

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 5-33 显示了可能的实现效果图。附录 D（见光盘）详细介绍了使用这个结构如何实现有限状态机。在 C.3（见光盘）节中，一个有限状态机的组合逻辑由 ROM（read-only memory，只读存储器）和 PLA（programmable logic array，可编程逻辑阵列）来实现。（附录 C（见光盘）中对这些逻辑单元也进行了描述。）

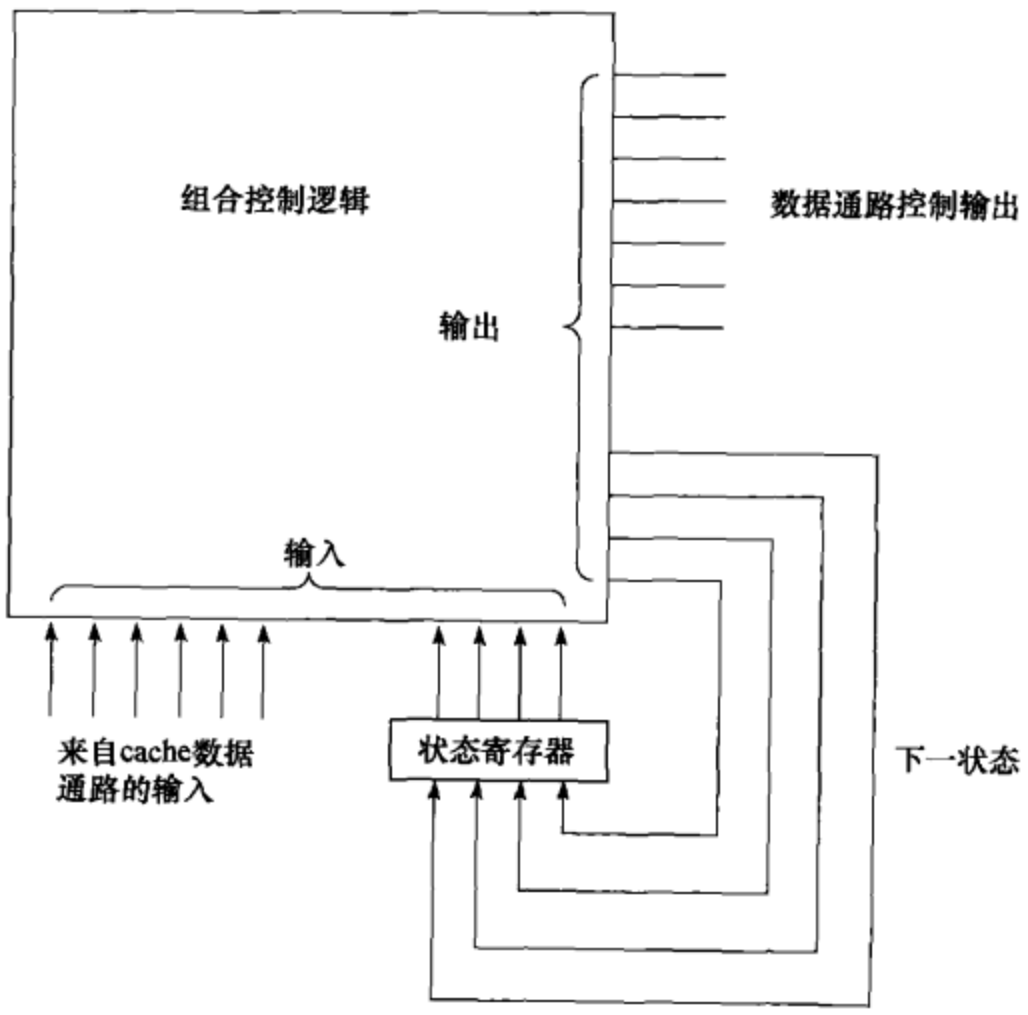


图 5-33　典型的有限状态机控制器由一个组合逻辑和一个保存当前状态的寄存器来实现

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。在这种情况下，输入就是指令寄存器的操作码位。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，精解更详细地进行了解释。

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 C（见光盘）中，用逻辑门实现了这种有限状态机，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。

5.7.3　一个简单的 cache 控制器的有限状态机

图 5-34 是简单 cache 控制器的四个状态。

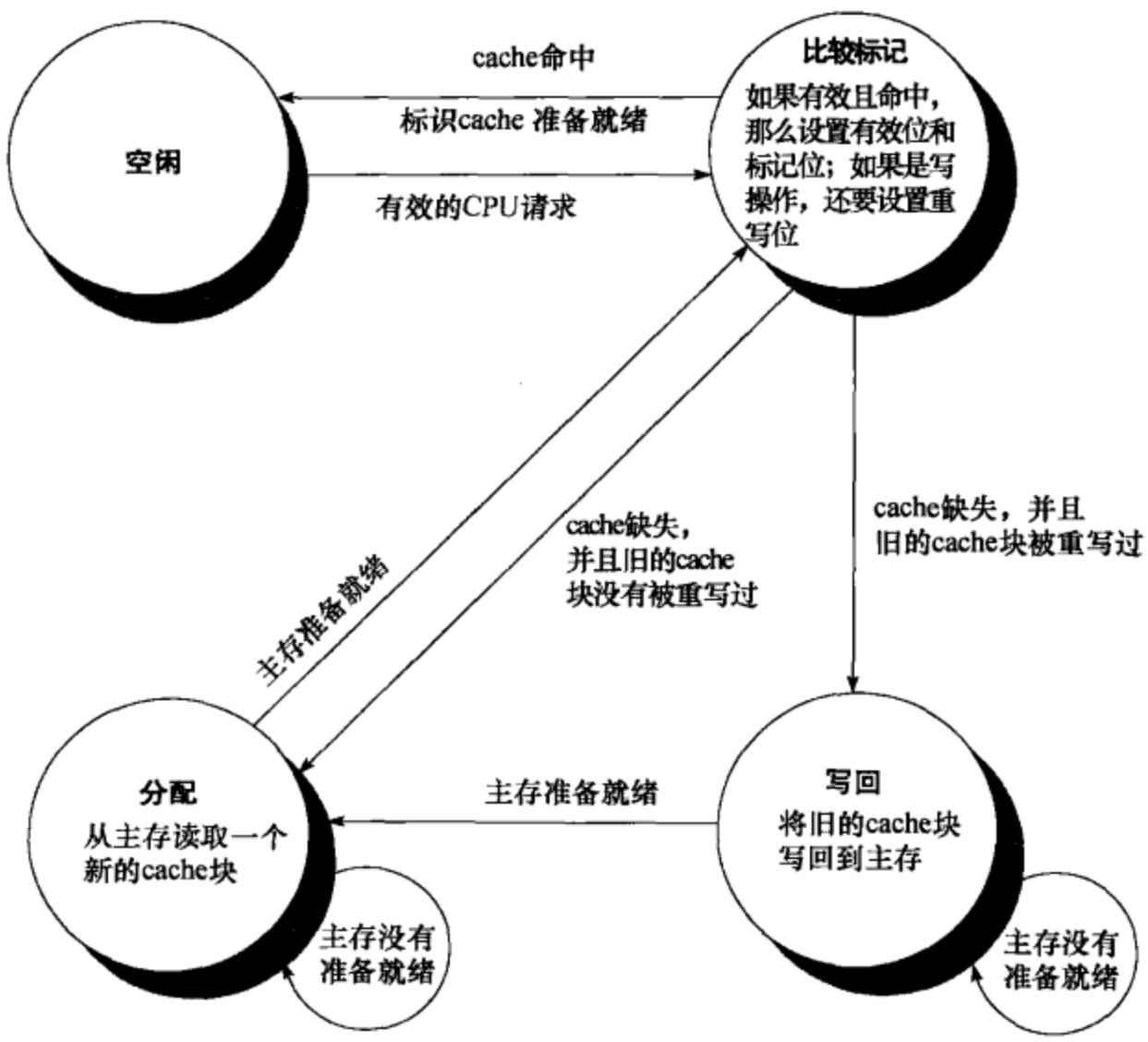


图 5-34　简单控制器的 4 个状态

- 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。
- 标记比较：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，发生命中。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，还要将重写位置为 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，改变重写位时，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。一次缺失首先要更新 cache 标记，随后，如果这个位置的块的重写位为 1，则转入写回状态；如果重写位为 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。我们继续停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。

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，这样我们就可以保存脏块，然后先读出新的块。这样，当一个脏块缺失时，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 CD 中的 5.9 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。

5.8　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 5-35 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 5-35　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做了更新，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 5-35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 5-35 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 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具有相同的顺序，这个性质称为写串行化（write serialization）。

5.8.1　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。

在下一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。

5.8.2　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5-36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。

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 5-36　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能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**[⊖]：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。

精解：尽管前面的三个属性已经能充分保证一致性，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 5-35 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。假设，例如，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的数据，因为在那一刻，被写的数据可能甚至还没有离开处理器。连贯性模型详细定义了写数据何时能被读操作看见。

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（没有完成时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访问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 之后再写位置 Y，那么，任何处理器在看到 Y 的新值时也必须看见 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，但是强制处理器以程序执行的顺序完成写操作。

精解：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。不同于 I/O，多个数据副本的情况很少——只要有就应该避免——程序运行在多个处理器上时，通常在一些 cache 中都有相同数据的副本。

精解：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的实现开销略高一些，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。

5.9　高级内容：实现 cache 控制器

本节内容在 CD 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

⊖ 假共享（false sharing）：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。

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。

5.10　实例：AMD Opteron X4（Barcelona）和 Intel Nehalem 的存储器层次结构

在这一节，我们将看一下两种现代微处理器的存储器层次结构：AMD Opteron X4（Barcelona）处理器和 Intel Nehalem。图 5-37 所示是 Intel Nehalem 的芯片照片，而第 1 章的图1-9 则是 AMD Opteron X4 的芯片照片。在主处理器芯片内两者都支持二级 cache 和三级 cache。这种集成减少了对较低级 cache 的访问时间，同时减少了芯片的管脚数，因为不需要连接片外二级 cache 的总线了。两款处理器都支持片内的主存控制器，这减少了与主存通信的延迟。

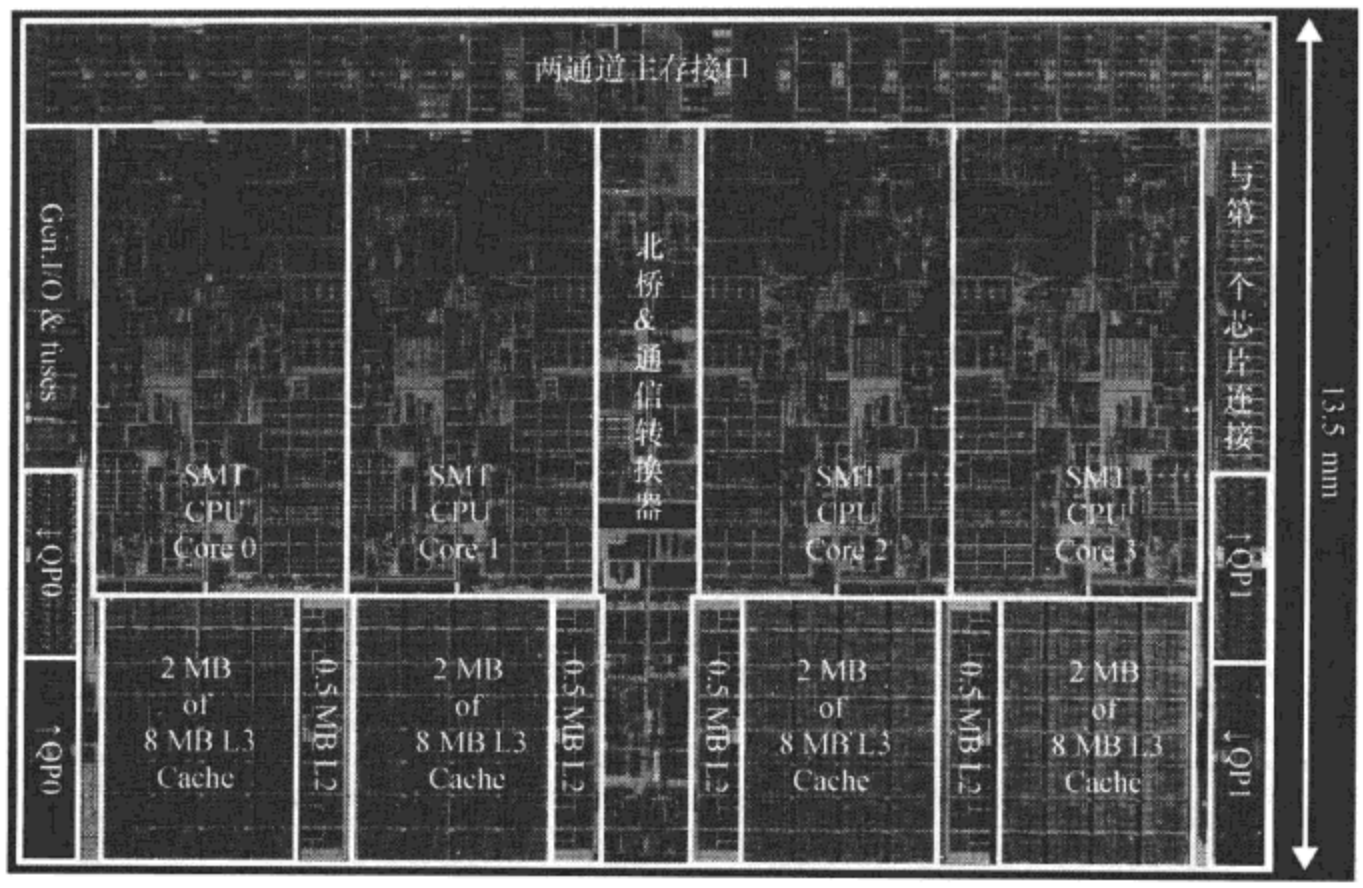


图 5-37　带元件标签的 Intel Nehalem 处理器芯片

这款 13.5 mm × 19.6 mm 的芯片有 731 百万个晶体管。它包含了 4 个处理器，每个处理器都有私有的 32 KB 的指令 cache 和 32 KB 的数据 cache，以及一个 512 KB 的二级 cache。四个核心共享一个 8 MB 的三级 cache。两个 128 位的存储器通道连接着 DDR3 DRAM。每个核还支持一个两级的 TLB。存储器控制器在片上，因此不像 Intel Clovertown 那样有独立的北桥芯片。

5.10.1　Nehalem 和 Opteron 的存储器层次结构

图 5-38 总结了两个处理器的地址大小和 TLB。注意到 AMD Opteron X4（Barcelona）有 4 个 TLB，并且虚拟地址和物理地址也不用和字大小匹配。X4 只使用了虚拟地址空间 64 位中的 48 位，64 位物理地址空间中的 48 位。Nehalem 有 3 个 TLB，虚拟地址是 48 位，物理地址是 44 位。

图 5-39 是这两个处理器的 cache。X4 中的每个处理器都有 64 KB 的一级指令 cache 和数据 cache，以及 512 KB 的二级 cache。四个处理器共享一个 2 MB 的三级 cache。Nehalem 有类似的结构，每个处理器有 32 KB 的一级指令 cache 和数据 cache，以及 512 KB 的二级 cache，四个处理器共享一个 8 MB 的三级 cache。

特征	Intel Nehalem	AMD Opteron X4（Barcelona）
虚拟地址	48 位	48 位
物理地址	44 位	48 位
页大小	4 KB，2/4 MB	4 KB，2/4 MB
TLB 组织结构	每个核都有一个指令 TLB 和一个数据 TLB 一级 TLB 都是四路组相联，LRU 替换算法 二级 TLB 是四路组相联，LRU 替换算法 一级指令 TLB 小页表有 128 项，大页表每线程有 7 项 一级数据 TLB 小页表有 64 项，大页表有 32 项 二级 TLB 有 512 项 硬件处理 TLB 缺失	每个核都有一个一级指令 TLB 和一个一级数据 TLB 一级 TLB 都是全相联，LRU 替换算法 每个核有一个二级指令 TLB 和一个二级数据 TLB 二级 TLB 都是四路组相联，循环替换算法 一级 TLB 均有 48 项 二级 TLB 均有 512 项 硬件处理缺失

图 5-38　Intel Nehalem 和 AMD Opteron X4 的地址变换和 TLB 硬件

字大小确定了虚拟地址的上限，但是处理器并不需要用到所有的位。两个处理器都支持大页，这可以用于操作系统或者映射帧缓冲区。大页机制避免了为映射一个总是存在的对象而浪费大量的表项。Nehalem 每个核心支持两个硬件支持的线程（参见第 7 章的 7.5 节）。

特征	Intel Nehalem	AMD Opteron X4（Barcelona）
一级 cache 组织结构	分离的指令和数据 cache	分离的指令和数据 cache
一级 cache 大小	每个核心私有 32 KB 指令/数据 cache	每个核心私有 64 KB 指令/数据 cache
一级 cache 关联度	指令 cache：4 路组相联 数据 cache：8 路组相联	2 路组相联
一级 cache 替换策略	近似的 LRU 替换策略	LRU 替换策略
一级 cache 块大小	64 字节	64 字节
一级 cache 写策略	写回，写分配	写回，写分配
一级 cache 命中时间 load-use	不可用	3 个时钟周期
二级 cache 组织结构	每个核心私有统一的指令和数据 cache	每个核心私有统一的指令和数据 cache
二级 cache 大小	256 KB（0.25 MB）	512 KB（0.5 MB）
二级 cache 关联度	8 路组相联	16 路组相联
二级 cache 替换策略	近似的 LRU 替换策略	近似的 LRU 替换策略
二级 cache 块大小	64 字节	64 字节
二级 cache 写策略	写回，写分配	写回，写分配
二级 cache 命中时间	不可用	9 个时钟周期
三级 cache 组织结构	统一的指令和数据 cache	统一的指令和数据 cache
三级 cache 大小	8192 KB（8 MB），共享	2048 KB（2 MB），共享
三级 cache 关联度	16 路组相联	32 路组相联
三级 cache 替换策略	不可用	替换被最少核共享的块
三级 cache 块大小	64 字节	64 字节
三级 cache 写策略	写回，写分配	写回，写分配
三级 cache 命中时间	不可用	38（?）个时钟周期

图 5-39　Intel Nehalem 和 AMD Opteron X4 2356（Barcelona）的一级、二级和三级 cache

图 5-40 显示了在 Opteron X4 上运行 SPECint 2006 基准测试程序时的 CPI、一级 cache 和二级 cache 中每 1000 条指令的缺失率，以及每 1000 条指令 DRAM 的访问次数。注意到 CPI 和 cache 缺

失率紧密相关。CPI 和每 1000 条指令的一级 cache 缺失次数之间的关联系数为 0.97。尽管我们没有实际的三级 cache 缺失，但是我们可以通过对二级 cache 缺失时减少的 DRAM 访问来推断三级 cache 的有效性。少数程序从 2 MB 的三级 cache 中明显获益——h264avc、hmmer 以及 bzip2——但是大部分程序并未获益。

名称	CPI	一级数据 cache 缺失/1000 条指令	二级数据 cache 缺失/1000 条指令	DRAM 访问/1000 条指令
perl	0.75	3.5	1.1	1.3
bzip2	0.85	11.0	5.8	2.5
gcc	1.72	24.3	13.4	14.8
mcf	10.00	106.8	88.0	88.5
go	1.09	4.5	1.4	1.7
hmmer	0.80	4.4	2.5	0.6
sjeng	0.96	1.9	0.6	0.8
libquantum	1.61	33.0	33.1	47.7
h264avc	0.80	8.8	1.6	0.2
omnetpp	2.94	30.9	27.7	29.8
astar	1.79	16.3	9.2	8.2
xalancbmk	2.70	38.0	15.8	11.4
Median	1.35	13.6	7.5	5.4

图 5-40　Opteron model X4 2356（Barcelona）存储系统中运行 SPECint 2006 时的 CPI、缺失率以及 DRAM 访问

在这款芯片上，三级 cache 缺失计数器并没有工作，因此我们只能通过 DRAM 访问次数来推测三级 cache 的效能。注意到这个图中的使用系统和基准测试程序和第 1 章图 1-20 中的一样。

5.10.2　减少缺失代价的技术

Nehalem 和 Opteron X4 都有其他的优化措施来降低缺失代价。首先是在缺失时先返回被请求的字，如 5.2.5 节的“精解”所描述的。这两款处理器都允许在 cache 缺失期间继续执行访问数据 cache 的指令。这个技术称为**非阻塞 cache**[⊖]，经常被设计者用在乱序处理器上来隐藏 cache 缺失延迟。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。

要重叠多个未解决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。在台式计算机系统中，存储器只能有限地获得这项功能的益处，但是大型服务器和多处理器通常拥有能并行处理不止一个缺失的存储系统。

这两款微处理器都预取指令，并且采用内嵌的硬件预取机制来访问数据。它们观察数据缺失的模式，并使用该信息尝试在缺失发生前预测下一个取数的起始地址。这种技术的效果通常在循环访问数组时最好。

cache 设计者面临的最严峻的一个挑战是，支持像 Nehalem 和 Opteron X4 那样的每个时钟周期可以执行不止一条存储器指令的处理器。两种不同的技术可以支持一级 cache 中的多个请求。cache 可以是多端口的，允许对同一个 cache 块的多个访问同时进行。然而，多端口 cache 通常很昂贵，因为多端口存储器中的 RAM 单元要比单端口中的大得多。另一种方案则是把 cache 分成不同的组，并允许多个互相独立的存取操作对两个不同的组进行访问。这种技术类似于主存的

⊖ 非阻塞 cache（nonblocking cache）：在处理器处理前面的 cache 缺失时仍可正常访问的 cache。

交叉存取（参见图 5-11）。Opteron X4 的一级数据 cache 每个时钟周期支持两个 128 位的读操作，它有 8 个组。

在存储层次中，Nehalem 和绝大多数其他的处理器都遵循包含策略。这意味着，在较高层次 cache 中所有数据的副本均可在较低层次的 cache 中找到。与之相反，AMD 处理器在第一、二级 cache 中，遵循互斥策略，意味着 cache 块仅能在第一级或第二级 cache 中找到，而不是在两者中都能找到。因此，由于 L1 缺失而将块从 L2 取入 L1 时，L1 中被替换的块被发回 L2cache。

这些芯片的复杂存储器层次结构以及芯片中很大一部分都用于 cache 和 TLB 的事实说明：为了缩短处理器周期时间和存储器延迟之间的距离，设计者付出了非常大的努力。

精解：Opteron X4 中的共享三级 cache 并不总是独占的。三级 cache 中的数据块可以被多个处理器所共享，如果没有其他处理器共享数据块，该数据块只能被移出三级 cache。因此，三级 cache 协议知道某个数据块是否被共享或者只被一个处理器所使用。

精解：正如 Opteron X4 不遵从常规的包容属性一样，它在存储器层次结构的各级之间有一种新的关系。不同于存储器向二级 cache 提供数据，然后二级 cache 再向一级 cache 提供数据那样，这里的二级 cache 只保存那些从一级 cache 中逐出的数据。因此，这种二级 cache 被称为牺牲 cache（victim cache），因为它只存储那些从一级 cache 中替换出来的块（“victims”）。同样，三级 cache 是二级 cache 的牺牲 cache，只包含从二级 cache 中移出的块。如果一级 cache 缺失没有在二级 cache 中得到处理，但是在三级 cache 中找到所需的数据，那么三级 cache 直接向一级 cache 提供数据。因此，一级 cache 缺失可以被二级 cache 命中、三级 cache 命中或者主存来处理。

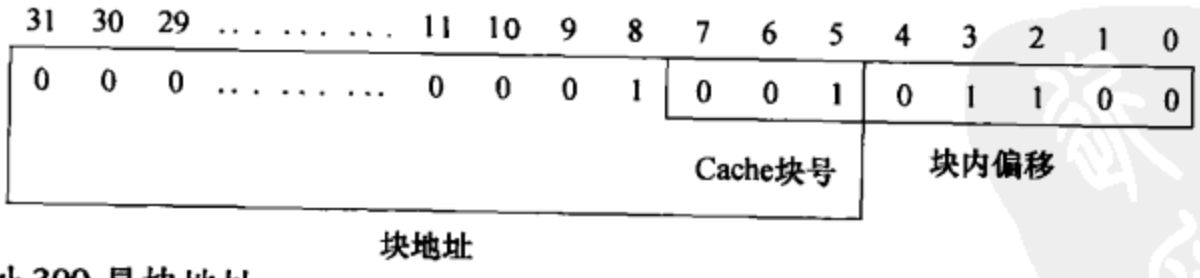
5.11　谬误和陷阱

作为计算机系统结构中的定量原则，存储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，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（9 mod 8）= 1。另一方面，如果地址 36 是字地址，那么它就映射到块（36 mod 8）= 4。要保证清楚地说明基准地址。

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

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

cache 中的块数是

$$\left\lfloor \frac{256}{32} \right\rfloor = 8$$

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：在写程序或者编译器生成代码时忽略了存储系统的行为。

这个也可以写作："在写代码时，程序员可能忽略了存储器层次结构。"我们可以通过一个使用矩阵乘法完成图5-18中的排序比较的例子来说明这个错误。

下面是第3章中矩阵乘法版本的内部循环代码：

```
for ( i =0; i! =500; i =i +1)
    for (j =0; j! =500; j =j +1)
        for (k=0; k! =500; k =k +1)
            x[i][j] =x[i][j] +y[i][k] * z[k][j];
```

当输入是500×500的双精度数的矩阵时，上述循环运行在一个拥有1 MB大小二级cache的MIPS CPU上，从CPU执行时间来看，其速度仅为将循环次序改为k，j，i（i在最里层）时的一半！唯一的区别就是程序如何访问存储器以及对存储器层次结构接下来的影响。我们使用一种称为分块（blocking）的技术进一步优化编译，可以使这段代码的运行时间再减少3/4。

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^n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^n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5.3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：通过在未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。

陷阱：在不为虚拟化设计的指令集系统结构上实现虚拟机监视器。

在20世纪70年代和80年代，很多计算机系统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在这些系统结构上存在问题，包括x86，这里我们就以它为例。

图5-41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）。
- 检查分段的系统结构所需的保护，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） 存储局部描述符表寄存器（SLDT） 存储中断描述符表寄存器（SIDT） 存储机器状态字（SMSW） 标志入栈（PUSHF，PUSHFD） 标志出栈（POPF，POPFD）
在用户模式下访问虚拟存储机制时，x86 保护检查指令失效	从段描述符读取访问权限（LAR） 从段描述符读取段的边界（LSL） 如果段描述符可读，进行读校验（VERR） 如果段描述符可写，进行写校验（VERW） 段寄存器出栈（POP CS，POP SS，…） 段寄存器入栈（PUSH CS，PUSH SS，…） 远调用不同的特权级（CALL） 远返回至不同的特权级（RET） 远跳转至不同的特权级（JMP） 软中断（INT） 存储段选择寄存器（STR） 移入/移出段寄存器（MOVE）

图 5-41　虚拟化产生问题的 18 条 x86 指令的概述［Robin 和 Irvine，2000］

上面一组的前五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。只有移入段寄存器操作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 都提出通过新的模式扩展系统结构。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系统结构定义、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。这种技术称为泛虚拟化（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。

5.12　本章小结

无论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。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。5.11 节的讨论说明了对循环结构即使只作简单的改变也会非常有效。

还有一种方法是**预取**[⊖]。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。

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 7 章的 7.10 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 7 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。

5.13　拓展阅读

本节光盘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。

5.14　练习题

本章习题由惠普公司的 Jichuan Chang、Jacob Leverich、Kevin Lim 和 Parthasarathy Ranganathan 提供。

习题 5.1

在这个练习中，我们研究适用于多种应用的存储器层次结构，应用在下表中列出：

a.	网络浏览
b.	在线银行

5.1.1 [10] <5.1> 假定客户和服务都被包含在进程中，首先命名客户和服务系统。那么 cache 放置在何处可以加速进程？

5.1.2 [10] <5.1> 为系统设计一个存储器层次结构。说明在层次结构中不同层的典型大小和延迟。在 cache 容量和访问延迟之间的关系是什么？

5.1.3 [15] <5.1> 在层次结构中，数据传输的单元是什么？数据的位置、数据的大小和传输延迟之间的关系是什么？

5.1.4 [10] <5.1，5.2> 当设计一个存储器层次结构时，通信带宽和服务处理带宽是需要考虑的两个重要因素。在这里哪一种带宽是限制因素？如何改进？改进的代价是什么？

5.1.5 [5] <5.1，5.8> 现在考虑多个客户同时访问服务器，这种情况能改进空间和时间局部性吗？

5.1.6 [10] <5.1，5.8> 请举例说明 cache 可以提供过时的数据。如何减少或者避免这个问题。

习题 5.2

在这个练习中，我们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。

⊖ 预取（prefetching）：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。

a.	for(I =0;I <8000;I ++)
	  for(J =0;J <8;J++)
	  A[I][J] =B[J][0] +A[J][I];

b.	for(J =0;J <8;J++)
	  for(I =0;I <8000;I ++)
	  A[I][J] =B[J][0] +A[J][I];

5.2.1 [5] <5.1>16 字节的 cache 行中可以存放多少 32 位的整数?

5.2.2 [5] <5.1>访问哪些变量会显示出时间局部性?

5.2.3 [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。同样的计算机可以用下面的 Matlab 来写，不同于 C，同一列的矩阵元素是连续存放的。

a.	for I =1:8000
	  for J =1:8
	  A(I,J) =B(J,0) +A(J,I);
	  end
	end

b.	for J=1:8
	  for I =1:8000
	  A(I,J) =B(J,0) +A(J,I);
	  end
	end

5.2.4 [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 行?

5.2.5 [5] <5.1>访问哪些变量会显示出时间局部性?

5.2.6 [5] <5.1>访问哪些变量会显示出空间局部性?

习题 5.3

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。

a.	1，134，212，1，135，213，162，161，2，44，41，221
b.	6，214，175，214，6，84，65，174，64，105，85，215

5.3.1 [10] <5.2>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。

5.3.2 [10] <5.2>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。

5.3.3 [20] <5.2，5.3>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。下表列出了对于不同的直接映射 cache 设计的参数。

	cache 数据量	cache 块大小	cache 访问时间
a.	64 KB	1 个字	1 个周期
b.	64 KB	2 个字	2 个周期

5.3.4 [15] <5.2> 假定 32 位的地址，计算表中列出的 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。

5.3.5 [20] <5.2，5.3> 在一个 2 KB 的组相联 cache 上产生一系列读请求时的缺失率要比在表中 cache 上执行读请求的缺失率低。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 2 KB cache 的缺失率。讨论这种解决方案的优点和缺点。

5.3.6 [15] <5.2> 5.2 节的第一个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 cache 可能需要做的一些改动。如果不可以，请解释原因。

习题 5.4

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
a.	31 ~ 10	9 ~ 4	3 ~ 0
b.	31 ~ 12	11 ~ 15	4 ~ 0

5.4.1 [5] <5.2> cache 行大小是多少（单位为字）？

5.4.2 [5] <5.2> cache 有多少项？

5.4.3 [5] <5.2> 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	1024	30	140	3100	180	2180

5.4.4 [10] <5.2> 有多少块被替换？

5.4.5 [10] <5.2> 命中率是多少？

5.4.6 [20] <5.2> 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。

习题 5.5

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。

	一级 cache	二级 cache
a.	写回，写分配	写直达，写不分配
b.	写回，写分配	写直达，写分配

5.5.1 [5] <5.2，5.5> 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。

5.5.2 [20] <5.2，5.5> 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。

5.5.3 [20] <5.2，5.5> 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。

	每1000条指令中数据读	每1000条指令中数据写	指令 cache 缺失率	数据 cache 缺失率	块大小（字节）
a.	200	160	0.20%	2%	8
b.	180	120	0.20%	2%	16

5.5.4 [5] <5.2，5.5>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）？

5.5.5 [5] <5.2，5.5>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？

5.5.6 [5] <5.2，5.5>如果要实现 CPI = 1.5 的性能，所需的最小带宽是多少？

习题 5.6

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 512 KB 的工作集的情况，地址流如下：

0，4，8，12，16，20，24，28，32，…

5.6.1 [5] <5.5，5.3>假设有一个 64 KB 的直接映射 cache，cache 行大小为 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 如何随之变化？根据 3C 模型，这些缺失如何被分类？

5.6.2 [5] <5.5，5.1>当 cache 行大小分别为 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？

5.6.3 [10] <5.10>“预取”是一种技术：当一个特殊 cache 行被访问时，预测地址模式并推测地取回其他 cache 行。预取的一个例子是流缓冲区，当一个特定的 cache 行被取回时，将与其相邻的 cache 行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 行。假设一个流缓冲区有两项，并且假设 cache 延迟满足：在先前 cache 行的计算完成之前可以加载下一个 cache 行。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，请找出最优的 cache 块大小。

	8	16	32	64	128
a.	8%	3%	1.8%	1.5%	2%
b.	4%	4%	3%	1.5%	2%

5.6.4 [10] <5.2>缺失延迟为 20 × B 个周期时，最佳的块大小是多少？

5.6.5 [10] <5.2>缺失延迟为 24 + B 个周期时，最佳的块大小是多少？

5.6.6 [10] <5.2>缺失延迟为恒定值时，最佳的块大小是多少？

习题 5.7

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 70 ns，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
a.	P1	1 KB	11.4%	0.62 ns
	P2	2 KB	8.0%	0.66 ns
b.	P1	8 KB	4.3%	0.96 ns
	P2	16 KB	3.4%	1.08 ns

5.7.1 [5] <5.3> 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？

5.7.2 [5] <5.3> P1 和 P2 各自的 AMAT（平均存储器访问时间）分别是多少？

5.7.3 [5] <5.3> 假定基本的 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
a.	512 KB	98%	3.22 ns
b.	4 MB	73%	11.48 ns

5.7.4 [10] <5.3> 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？

5.7.5 [5] <5.3> 假定基本的 CPI 为 1.0，增加二级 cache 后，P1 的总的 CPI 是多少？

5.7.6 [10] <5.3> 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？

习题 5.8

这个练习研究了不同 cache 设计的效果，特别将关联的 cache 与 5.2 节中的直接映射的 cache 进行比较。练习中使用的是习题 5.3 中的地址流表。

5.8.1 [10] <5.3> 使用习题 5.3 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、块偏移位，以及当前访问是命中还是缺失。

5.8.2 [10] <5.3> 使用习题 5.3 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。

5.8.3 [15] <5.3> 使用习题 5.3 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，请问缺失率是多少？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
a.	2.0	3 GHz	125 ns	5%	15 周期	3.0%	25 周期	1.8%
b.	2.0	1 GHz	100 ns	4%	10 周期	4.0%	20 周期	1.6%

5.8.4 [10] <5.3> 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？

5.8.5 [10] <5.3> 拥有比两级更多的 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？

5.8.6 [20] <5.3> 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 512 KB 的片外二级 cache 的全局缺失率为 4%。如果 cache 每增加 512 KB 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？

习题 5.9

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 10 ms，传输率为 10 MB/s，最优的页大小是 16 K。

页大小（KB）	页的使用/B-tree 深度（保存的磁盘访问次数）	索引页的访问开销（ms）	效用/代价
2	6.49 或 $\log_2$（2048/16 × 0.7）	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

5.9.1 [10] <5.4> 如果项数为 128 字节，最佳的页大小是多少？

5.9.2 [10] <5.4> 根据习题 5.9.1，如果页处于半满状态，最佳的页大小是多少？

5.9.3 [20] <5.4> 根据习题 5.9.2，如果使用的是最新的磁盘，延时 3 ms，而传输率为 100 MB/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？

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_MB/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（$/MB）	页大小（KB）	磁盘开销（$/disk）	磁盘访问率（访问/秒）
1987	5000	1	15 000	15
1997	15	8	2 000	64
2007	0.05	64	80	83

5.9.4 [10] <5.1，5.4> 对于这三种技术时代，重用时间阈值是多少？

5.9.5 [10] <5.4> 如果我们保持使用相同的 4 K 大小的页，重用时间阈值是多少？这里趋势是什么？

5.9.6 [20] <5.4> 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。

习题5.10

如5.4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4 KB的页，一个四项的全相联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。

a.	4095，31 272，15 789，15 000，7193，4096，8912
b.	9452，30 964，19 136，46 502，38 110，16 653，48 480

TLB

有效位	标记位	物理页号	有效位	标记位	物理页号
1	11	12	1	3	6
1	7	4	0	4	9

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

5.10.1 [10] <5.4>已知表中的地址流，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。

5.10.2 [15] <5.4>重复习题5.10.1，但是这次使用16 KB的页来代替4 KB的页。使用更大的页的好处有哪些？缺点又有哪些？

5.10.3 [15] <5.3，5.4>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
a.	32位	4 KB	4字节
b.	64位	16 KB	8字节

5.10.4 [5] <5.4>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。

5.10.5 [10] <5.4>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个字节，计算所需的最小和最大的内存容量。

5.10.6 [10] <5.4>一名cache设计者要将一个4 KB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16 KB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？

习题 5.11

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）
a.	32	4 GB	8 KB	4
b.	64	16 GB	4 KB	8

5.11.1 [10] <5.4> 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大小？

5.11.2 [10] <5.4> 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？

5.11.3 [15] <5.4> 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？

下表是一个有四项内容的 TLB。

项的 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

5.11.4 [5] <5.4> 在什么样的情况下第二项的有效位被置为 0？

5.11.5 [5] <5.4> 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？

5.11.6 [5] <5.4> 当一条指令写入虚拟地址页 xxx 时，会发生什么情况？

习题 5.12

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。你会发现画表（就像 5.3.1 节的例子那样）对于解决习题中的问题很有帮助，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	被逐出的块	引用后 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］
…						

下表是地址序列。

	地址序列
a.	0，2，4，0，2，4，0，2，4
b.	0，2，4，2，0，2，4，0，2

5.12.1 [5] <5.3，5.5> 假定使用 LRU 替换算法，在这组地址序列中有多少次命中？

5.12.2 [5] <5.3，5.5>假定使用 MRU（最近最常使用）替换算法，在这组地址序列中有多少次命中？

5.12.3 [5] <5.3，5.5>通过掷硬币来模拟随机替换算法。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？

5.12.4 [10] <5.3，5.5>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？

5.12.5 [10] <5.3，5.5>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？

5.12.6 [10] <5.3，5.5>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？

习题 5.13

为了支持多虚拟机，需要两级存储器虚拟化。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）或被称为扩展页表（extended page table）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。

5.13.1 [10] <5.4，5.6>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？

5.13.2 [10] <5.4，5.6>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？

5.13.3 [15] <5.4，5.6>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？

下表是影子分页系统的参数。

每 1000 条指令 TLB 缺失数	嵌套页表 TLB 缺失延迟	每 1000 条指令缺页数	影子页缺失代价
0.2	200 个周期	0.001	30 000 个周期

5.13.4 [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？

5.13.5 [10] <5.6>使用什么技术可以减少影子页表所带来的开销？

5.13.6 [10] <5.6>使用什么技术可以减少嵌套页表所带来的开销？

习题 5.14

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）
a.	2	100	20 个周期	150 个周期	20	1000 个周期
b.	1.5	110	25 个周期	160 个周期	10	1000 个周期

5.14.1 [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？

5.14.2 [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。

5.14.3 [30] <5.4，5.6> 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。

5.14.4 [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？

习题 5.15

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 5-34 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 5.7.1 节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。

5.15.1 [10] <5.5，5.7> 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？

5.15.2 [10] <5.5，5.7> 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？

5.15.3 [30] <5.5，5.7> 设计一个有限状态机使得写缓冲器能被使用。

习题 5.16

cache 一致性考虑的是多个处理器看到的同一个 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）。

	P1	P2
a.	`X[0]++;  X[1] =4;`	`X[0] =2;  X[1]++;`
b.	`X[0]++;  X[1] + =3;`	`X[0] =5;  X[1] =2;`

5.16.1 [15] <5.8> 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。

5.16.2 [15] <5.8> 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。

5.16.3 [10] <5.8> 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.	`A=1;B =2;A++;B++;`	`C =B;D =A;`
b.	`A=1;B+ =2;A++;B =4;`	`C =B;D =A;`

5.16.4 [15] <5.8> 若使用 5.8 节开始部分列出的保证一致性协议的假设，请列出 C 和 D 的值。

5.16.5 [15] <5.8> 如果假设不成立，那么至少列出一对 C 和 D 可能的值。

5.16.6 [15] <5.2，5.8> 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？

习题 5.17

Barcelona 和 Nehalem 都是单芯片多处理器（chip multiprocessor，CMP），即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
a.	6	12	120
b.	8	20	120

5.17.1 [15] <5.10> 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。

5.17.2 [15] <5.10> 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。

5.17.3 [10] <5.10> 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；如果还有片上三级 cache，请重新考虑这些问题。

5.17.4 [15] <5.10> 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 提升到 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？

5.17.5 [10] <5.10> 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 2012 年的处理器需要多少片下存储器带宽？

5.17.6 [15] <5.10> 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？

习题 5.18

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 ;              //运程 IP 地址
    char URL[128] ;          //请求 URL (例如. , "GET index. html")
    long long refTime ;      //访问时间
    int status ;             //连接状态
    char browser[64] ;       //客户浏览器名称
} log [NUM_ENTRIES] ;
```

一些日志上的处理函数如下：

a.	topk_sourceIP ( );
b.	peak_hour(int status) ; //给定状态的峰值

5.18.1 [5] <5.11> 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？

5.18.2 [10] <5.11> 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？请给出结构代码定义。

5.18.3 [10] <5.11> 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”（www.cs.wisc.edu/multifacet/misc/spec2000cache-data/），如下表所示。

a.	apsi/facerec
b.	perlbmk/ammp

5.18.4 [10] <5.11> 64 KB 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？

5.18.5 [10] <5.11> 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 64 KB。如果一级 cache 是直接映射的，那么为 1 MB 的二级 cache 选择组相联度。

5.18.6 [20] <5.11> 请列举一个缺失率表的例子说明较高的关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。

小测验题参考答案

5.1 节 A 和 D。（C 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 2008 年开销最高的通常是 DRAM。）

5.2 节 A 和 D。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。

5.3 节 A。

5.4 节 A 和 a，B 和 c，C 和 b，D 和 d。

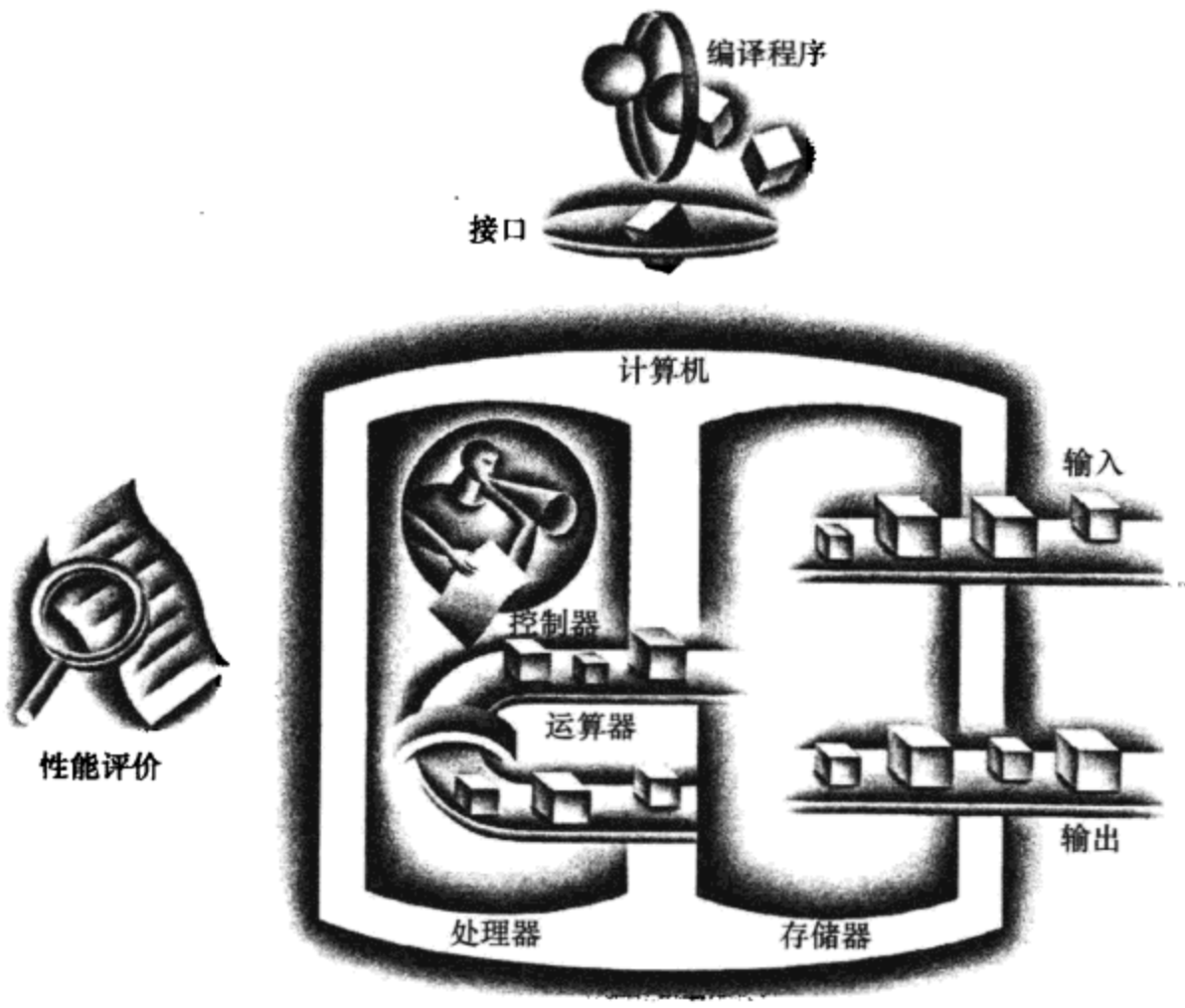
5.5 节 B。（大容量的块和预取都能降低强制缺失，因此 A 是错误的。）



存储器和其他 I/O 主题

有效结合传输带宽与存储器资源……可以快速而可靠地访问日益增长的宝贵信息…无论是从不断扩容的磁盘空间还是从有无尽宝藏的因特网。

——George Gilder,《The End Is Drawing Nigh》, 2000



计算机的五大经典部件

6.1 引言

如果用户的电脑死机了而不得不重启，他们会感到沮丧；但是如果存储系统崩溃了而丢失信息，情况就会更糟。因此，存储的可信度标准要大大高于计算。网络也为处理通信错误准备了一些方案，包括几套检测错误并从中恢复的机制。因此，输入与输出系统（I/O）通常把重点放在可信度与成本上，而处理器和内存则把重点放在性能与成本上。

I/O 系统还必须考虑设备的可扩展性和多样性，而这是处理器所不关心的。可扩展性涉及存储设备的容量，这是 I/O 系统的又一个设计参数；对完成它们的任务而言，系统可能需要拥有较低限的存储容量。

尽管 I/O 的性能重要性相对较低，但它的表现更为繁杂。比如，对某些设备来说，我们主要关心存储延时，而对另外一些设备来说，吞吐率才是至关重要的因素。进一步来说，性能的好坏取决于系统的很多方面，如设备特性、设备与系统中其他部分的连接、存储器层次结构和操作系统等。系统所有的组成部分，从单个 I/O 设备到处理器，再到系统软件，都会

影响到包括 I/O 操作在内的任务可信度、可扩展性和性能。图 6-1 描述的就是一个带有 I/O 设备的简单系统的结构。

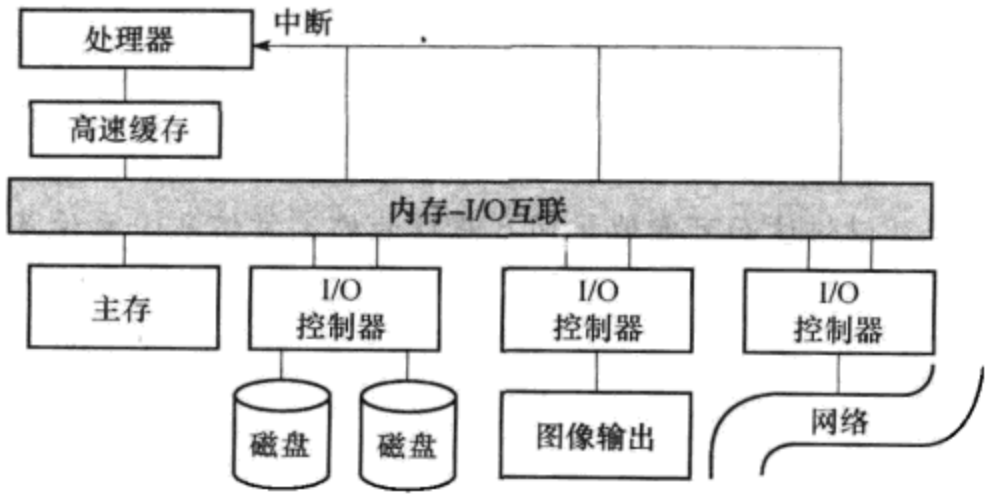


图 6-1　I/O 设备典型的连接

I/O 设备与处理器以及内存之间的连接线路通常被称作总线，虽然这个术语意味着共享的并行连线，但如今很多 I/O 连接与专用串行总线很相似。本章将介绍设备与处理器之间使用这些互联协议以及中断来通信。图 6-9 给出了一个桌面 PC 机的例子。

I/O 设备有令人难以置信的多样性。由此带来的众多特性中，有三类极为重要：

- 行为（behavior）：输入（只读）、输出（只写，不能被读）或者存储（可以被重读或者重写）。
- 合作者（partner）：I/O 设备的另一端是人还是机器，是在输入端输入数据还是在输出端读数据。
- 数据速率（data rate）：数据在 I/O 设备与主存或者处理器之间传输的峰值速率（peak rate）。它可用来了解在设计一个 I/O 系统时，设备能产生的最大需求。

例如，键盘是一个由人使用的输入（input）设备，具有大约每秒 10 个字节的峰值数据速率。图 6-2 列出了一些连接到计算机上的 I/O 设备。

设备	行为	合作者	数据速率（Mbit/s）
键盘	输入	人	0.0001
鼠标	输入	人	0.0038
语音输入设备	输入	人	0.2640
声音输入设备	输入	机器	3.0000
扫描仪	输入	人	3.2000
语音输出设备	输出	人	0.2640
声音输出设备	输出	人	8.0000
激光打印机	输出	人	3.2000
图像显示器	输出	人	800.0000 ~ 8000.0000
调制解调器	输入或者输出	机器	0.1280 ~ 6.0000
网络/局域网	输入或者输出	机器	100.0000 ~ 10 000.0000
网络/无线局域网	输入或者输出	机器	11.0000 ~ 54.0000
光盘	存储	机器	80.0000 ~ 220.0000
磁带	存储	机器	5.0000 ~ 120.0000
快闪式存储器	存储	机器	32.0000 ~ 200.0000
磁盘	存储	机器	800.0000 ~ 3000.0000

图 6-2　各种 I/O 设备

可以根据以下条件来区别 I/O 设备：首先，它们是用于输入、输出还是存储的设备；其次，它们的合作者（人还是其他的计算机）；再次，它们的峰值通信速率。数据速率快慢跨度达八个数量级。注意网络可以作为输入设备或者输出设备，但是不能用作存储设备。设备的传输速率总是以基 10 来表示，所以10 Mbit/s = 10 000 000 bit/s。

在第 1 章，我们简要讨论了四个重要的、具备代表性的 I/O 设备：鼠标、图像显示器、磁盘

以及网络。本章，我们将更加深入地研究存储及相关问题。光盘中关于网络的高级话题，在作者另外一本书中进行了详细介绍。

我们如何评测 I/O 性能常常取决于应用。在某些环境下，我们注重系统的吞吐量。在这种情况下，I/O 带宽将是最重要的。但即便是 I/O 带宽也有两种不同的方法来测量：

1）在某一段时间内，我们能够从系统中传送多少数据？

2）在某个单位的时间内，我们能够做多少个 I/O 操作？

选取何种性能测量方法取决于实际的工作环境。例如，在很多媒体应用中，大部分的 I/O 请求用于处理长的数据流，因此传送带宽是重要的指标。在另外一些情况下，我们可能希望 I/O 设备进行多个小数据量的、不相关的存取。这里有一个来自美国国家税务局（NITS）的税收办公室的例子。美国国家税务局通常需要在某一段特定的时间内处理一大批表格，各税务表格分别存储且相当小。一个面向大文件的传输系统或许也能满足要求，但是一个能够支持小文件同时传输的 I/O 系统对于处理这些数以百万计的税务表格来说也许更廉价和更便捷。

在其他一些应用中，我们更重视响应时间，即为完成特定任务总共需要的时间。当 I/O 请求[⊖]非常多的时候，响应时间严重依赖于带宽，但是很多情况下大多数访问的数据量很小，所以单个访问操作延迟时间最短的 I/O 系统将获得最快的响应时间。对于台式机和笔记本电脑这样的单用户机器，响应时间是关键的性能指标。

大多数应用，特别是在广大商务计算机市场中，要求同时具备高吞吐量和快速的响应时间。自动取款机（ATM）、订货库存跟踪系统、文件服务器、Web 服务器都属于这种情况。在这些系统中，我们关心每个任务耗费多少时间以及每秒可以处理多少任务。如果每次应答都需要 15 分钟，那么每小时能处理的 ATM 请求数量将没有任何意义，你将不可能再有任何用户。同样，如果你对每个请求都能快速处理，但是一次只能同时处理少量的请求，那么也无法支持多个 ATM，或者每台 ATM 的计算机成本会很高。

总之，三种类型的计算机（台式机、服务器以及嵌入式计算机）对于 I/O 可信度和成本是敏感的。台式机和嵌入式系统更注重响应时间和 I/O 设备的多样性，而服务器更关心吞吐量和 I/O 设备的可扩展性。

6.2　可信度、可靠性和可用性

用户渴望得到可以信赖的存储器，但是如何来定义它呢？在计算机界，这比查字典难多了。经过大量的辩论，下面的描述可以被认为是标准的定义（Laprie，1985）：

计算机系统的可信度指它所提供的服务质量达到了有理由给予信任的水平。系统提供的服务是指通过其他系统与本系统的用户交互，其他系统观察到的本系统的实际行为。每个模块均有指定的理想行为，其中服务需求是对期望行为广泛认同的描述。当实际的行为与指定的行为偏离时，系统发生错误。

这样，为了确定可信度，需要给出期望行为的一个参考规范。这样用户在观察系统时，会看到系统在参考规范描述的两种服务状态之间改变：

1）服务实现（service accomplishment），按照预定方式提供服务。

2）服务中断（service interruption），提供不同于预定的服务。

从状态 1 到状态 2 的转换由故障引起，从状态 2 到状态 1 的转换称为恢复（restoration）。故障可以是永久性的，也可以是偶发性的。偶发性的故障更难诊断，因为系统在两个状态之间振荡；相比较而言，永久性故障的诊断容易些。以上定义导致两个相关术语：可靠性（reliability）

⊖ 对 I/O 设备的读或写。

和可用性（availability）。

可靠性是实现连续服务的度量——或者说从服务开始的测量点至出现故障时的时间度量。因此，图 6-5 的磁盘平均故障时间（mean time to failure，MTTF）就是一个可靠性的度量。一个相关的术语是年失效率（annual failure rate，AFR）。AFR 可理解为在给定的 MTTF 下，设备在一年内可能发生故障的概率。服务中断使用平均修理时间（mean time to repair，MTTR）作为度量。故障之间的平均时间（mean time between failures，MTBF）刚好是 MTTF 与 MTTR 的和。虽然 MTBF 被广泛使用，但 MTTF 通常是一个更合理的度量。

当服务在上述两个状态之间变化时，可用性是服务实现的一个度量。可用性可以表示为

$$\text{Availability} = \text{MTTF}/(\text{MTTF} + \text{MTTR})$$

注意，可靠性和可用性实际上都是可度量的，而不仅是可信度的同义词。

引起故障的原因有哪些？图 6-3 总结了许多关于计算机系统和通信系统的故障原因的文章。很明显，人为操作是一个重要的故障来源。

操作员	软件	硬件	系统	搜集数据的年份
42%	25%	18%	数据中心（Tandem）	1985
15%	55%	14%	数据中心（Tandem）	1989
18%	44%	39%	数据中心（DEC VAX）	1985
50%	20%	30%	数据中心（DEC VAX）	1993
50%	14%	19%	美国公用电话网	1996
54%	7%	30%	美国公用电话网	2000
60%	25%	15%	因特网服务	2002

图 6-3　故障原因的研究总结

虽然很难收集数据来确定操作员是不是导致故障的原因（因为通常是操作员来记录故障的原因），但是这些研究确实找到了操作员作为故障原因的数据。常常还有其他类型的原因，如环境因素，但这些通常都微不足道。图中最上面两行来自于 Jim Gray［1990］的一篇经典论文，自收集数据后的 20 年来仍被广泛引用。接下来两行来自 Murphy 和 Gent 的一篇论文，他们研究不同时期 VAX 系统的出错原因［“Measuring system and software reliability using an automated data collection process”，《Quality and Reliability Engineering International 11：5》，1995 年 10～12 月，341～353］。第 5 行和第 6 行是 Kuhn 和 Patty Enriquez 对美国公用交换电话网络 FCC 错误数据的研究［“Sources of failure in the public switched telephone network”，《IEEE Computer 30：4》，1997 年 4 月，31～36］。最近的关于三项因特网服务的研究来自于 Oppenheimer、Ganapath 和 Patterson［2003］。

为了提高 MTTF，可以提高部件的质量或者设计一种可以在一些部件发生故障时能继续工作的系统。因此，需要根据实际情况定义故障（failure）。某个部件的故障不一定引起系统故障。为了明确这种区别，术语 fault（错误）用来表示一个部件的故障。有三种提高 MTTF 的途径：

1）错误避免（fault avoidance）：通过结构设计来防止错误的发生。

2）错误容忍（fault tolerance）：利用冗余技术允许服务在错误发生时仍然能正常地工作，这里的错误主要指的是硬件错误。6.9 节中描述的 RAID 方法就是通过错误容忍使得存储设备变得可靠。

3）错误预测（fault forecasting）：预测错误的存在和产生，将这种预测使用到硬件错误和软件错误中，可以达到在一个部件出现故障之前替换掉它的目的。

缩短 MTTR 可以和增大 MTTF 一样达到提高可用性的目的。例如，错误检测、错误诊断以及错误修复工具均有助于减少修复错误（无论是人为的，还是软、硬件引起的）的时间。

小测验

关于可信度，下面哪些论述是正确的？

A. 如果系统上电，那么它的所有组成部分都在完成它被期望的服务。
B. 可用性是系统可以取得所期望服务的时间百分比，是一个可量化的指标。
C. 可靠性是系统可以实现连续服务的一个可量化指标。
D. 如今，软件是故障的一个主要源头。

6.3　磁盘存储器

如第 1 章所述，磁盘依赖带磁介质表面的旋转盘片并使用一个可以移动的读/写磁头来访问。磁盘存储是**非易失性的**[一]——当停止供电后，存储设备还能够保持数据。磁盘可由一叠盘片（1 ~ 4个）构成，每张盘片有两个可记录数据的磁盘面。盘片以 5 400 ~ 15 000 RPM（转/分）的速度旋转，盘片直径从 1 ~ 3.5 英寸不等。每一个盘面被分成许多**磁道**[二]。每一个磁盘面一般有 10 000 ~ 50 000 条磁道。每个磁道又被分成一些记录信息的**扇区**[三]；每条磁道可以有 100 ~ 500 个扇区；扇区的大小通常有 512 ~ 4 096 个字节。在磁介质中一般顺序存放着扇区号、间隔、存储在扇区的带有纠错码的信息（参见光盘中的附录 C）、间隔和下一个扇区的扇区号等。

早期磁盘的所有磁道具备相同数目的扇区，因此具备相同数目的位数，但是随着 20 世纪 90 年代早期区域位记录 ZBR（*zone bit recording*）的出现，磁盘变成每个磁道拥有可变数目的扇区（位数也是如此），而保持位之间的距离不变。ZBR 增加了外围磁道的位数，因此增加了磁盘可用容量。

在第 1 章我们看到，为了读/写信息，磁头必须移动到正确的位置上。每个磁盘面的磁头都被连在一起并且一起移动，因此每个磁头会处在每个面的相同磁道上。*柱面*（cylinder）这个术语是指：当磁头定位在盘面的某个给定位置，磁头下所有的盘面对应的磁道所形成的柱面。

为了访问数据，操作系统必须通过一个分三个阶段完成的进程来指挥磁盘。第一步，把磁头定位到正确的磁道上，这个操作叫做**寻道**[四]，磁头找到正确磁道的时间被称为*寻道时间*（seek time）。

磁盘制造商在使用手册中报告了最小寻道时间、最大寻道时间和平均寻道时间。前两种很容易测量，但是平均寻道时间由于与寻道距离有关而有多种解释。工业界用所有可能的寻道时间总和除以可能发生的寻道次数来计算平均寻道时间。平均寻道时间一般宣传为 3 ~ 13 ms，但是依赖于应用和对磁盘请求的调度，由于磁盘访问的局部性，实际的平均寻道时间仅仅为上述数值的 25% ~ 33%。这种局部性的出现一方面是由于同一个文件的连续访问操作，另一方面由于操作系统尽量把相邻地址的操作安排在一起。

磁头一旦到达正确的磁道，我们必须等待正确的扇区旋转到读/写磁头下面，这段时间被称为**旋转时间**[五]。取得所要的信息的平均延时是磁盘旋转半周所需的时间。因为磁盘旋转速度是 5 400 ~ 15 000 RPM，所以平均旋转延时是在

$$\text{平均旋转延时} = \frac{0.5\ \text{转}}{5\,400\ \text{RPM}} = 0.0056\ \text{s} = 5.6\ \text{ms}$$

与

㊀ 非易失性的（nonvolatile）：当断电时，数据仍保留的存储设备。
㊁ 磁道（track）：磁盘面上的一个同心圆为一个磁道。
㊂ 扇区（sector）：磁道上的一段弧称为扇区，一个扇区是磁盘中被读或者写的最小信息块。
㊃ 寻道（seek）：在一个读或者写操作中，把磁头定位到合适的磁道的过程。
㊄ 旋转时间（rotational latency）：也称为旋转延迟（rotational delay），是使得合适的扇区旋转到读/写头下的时间。

$$\text{平均旋转延时} = \frac{0.5\ \text{转}}{15\,000\ \text{RPM}} = 0.002\,0\ \text{s} = 2.0\ \text{ms}$$

之间。

磁盘存取的最后一个要素是传输时间，即传输一块数据所需的时间。传输时间是扇区大小、旋转速度和磁道记录密度的函数。在 2008 年，传输速率处于 70 MB/s 和 125 MB/s 之间。一个使得问题复杂化的因素是大部分的磁盘控制器中都有一个内置缓存来保持存储访问过的扇区；而通过这个缓存的数据传输速率通常都相当高，2008 年可以达到 375 MB/s，即 3 Gbit/s。如今，大多数磁盘以多个扇区为单位进行传输。

磁盘控制器（disk controller）通常用来具体地控制磁盘以及磁盘与内存之间的数据传输。控制器导致磁盘存取时间又多了一项——控制器时间（controller time），它是执行 I/O 存取操作时控制器带来的开销。执行 I/O 操作的平均时间将由这四段时间组成，此外还有因其他程序使用磁盘而带来的等待时间。

举例　读磁盘时间

对于一个旋转速度为 15 000 RPM 的磁盘，读或者写一个 512 字节的扇区需要多少时间？手册上宣称的平均寻道时间是 4 ms，传输速率是 100 MB/s，控制器开销是 0.2 ms。假设这个磁盘是空闲的，也就是没有等待时间。

答案

平均磁盘存取时间 = 平均寻道时间 + 平均旋转延迟 + 传输时间 + 控制器开销。将手册中公布的平均寻道时间代入，答案是

$$4.0\ \text{ms} + \frac{0.5\ \text{转}}{15\,000\ \text{RPM}} + \frac{0.5\ \text{KB}}{100\ \text{MB/S}} + 0.2\ \text{ms} = 4.0 + 2.0 + 0.005 = 6.2\ \text{ms}$$

如果测量到的平均寻道时间是手册中所公布的平均寻道时间的 25%，那么，答案就变为

$$1.0\ \text{ms} + 2.0\ \text{ms} + 0.005\ \text{ms} + 0.2\ \text{ms} = 3.2\ \text{ms}$$

注意，当我们考虑测量到的平均寻道时间时，旋转延时成为访问时间中最大的部分，和手册中所说的平均寻道时间是访问时间中最大的部分相矛盾。

磁盘的密度持续增加已经超过 50 年。如图 6-4 表示，磁盘的密度提高了，物理体积却在不断缩小的这种综合影响令人吃惊。磁盘设计者的不同目的导致在不同时期都出现了具有不同特点的磁盘。图 6-5 显示了四种不同磁盘的特性。在 2008 年，来自同一厂商的磁盘售价每 GB 为 0.3 ~ 5 美元。在市场上，根据尺寸、接口和性能的不同，一般的价格范围为每GB 0.2 ~ 2 美元。

在可以预见的未来，应用磁盘仍然是主流，但访问数据块的方法会不断改变。扇区 - 磁道 - 柱面（sector-track-cylinder）模型是这样的：邻接的块处在同一个磁道上；处于同一个柱面的块将花费比较少的时间来寻找，因为不需要寻道时间；邻近磁道的寻道时间更少。这种模型失败的原因在于接口层次的上升。如 ATA[⊖]和 SCSI[⊜]这类高级智能接口需要在磁盘中设置微处理器，来进行性能的优化。

为了加速顺序的传输，这些高级接口将磁盘组织成类似磁带设备，而非随机访问设备。逻辑块按照类似单一平面上的蛇形线结构来组织，并试图捕捉所有以相同的位密度存储的扇区。因此，顺序的块可能会处于不同的磁道上。我们将会在图 6-19 中看到这种基于传统的扇区 - 磁道 - 柱面模型的磁盘的缺陷。

⊖ ATA（Advanced Technology Attachment）：在 PC 中很流行的一种被用作 I/O 设备标准的指令集。

⊜ SCSI（Small Computer Systems Interface）：一种 I/O 设备的标准指令集。

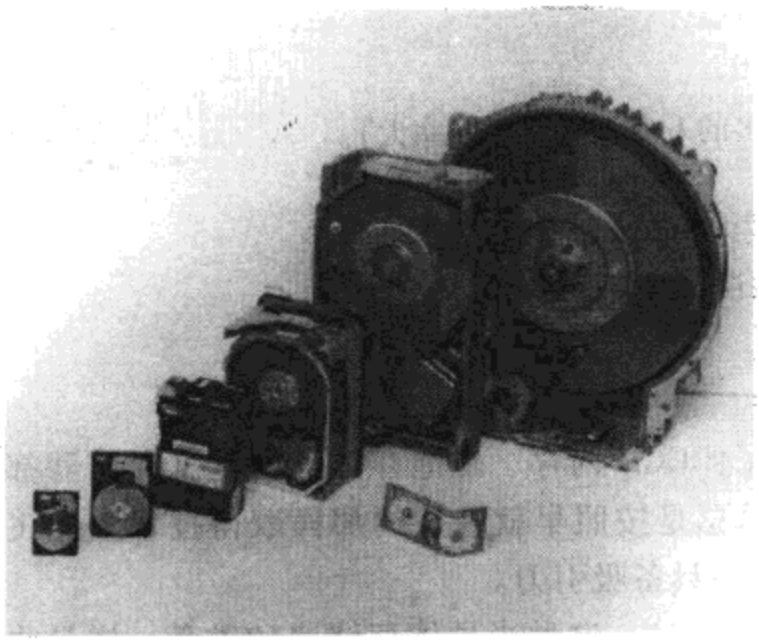


图 6-4　直径从 14 英寸减到 1.8 英寸的 6 种磁盘

图中的磁盘是十几年以前推出的，因此并不指望代表同尺寸现代磁盘的最好性能。然而这幅图精确地描绘了它们的相对物理尺寸。最大的磁盘是 DEC 公司的 R81，配有 4 张直径为 14 英寸的盘片，存储容量为 456 MB，于 1985 年制造。直径 8 英寸的磁盘来自日本富士通公司，这种 1984 年出品的磁盘在 6 张盘片上存储了 130 MB 的容量。Micropolis 公司的 RD53 有 5 张 5.25 英寸的盘片，存储容量为 85 MB。IBM 公司的 0361 也有 5 张盘片，但是它们的直径只有 3.5 英寸。这种 1988 年的磁盘容量为 320 MB。在 2008 年，密度最高的 3.5 英寸磁盘有 2 张 3.5 英寸的盘片，磁盘容量为 1TB，以相同的空间，在密度上提高了 3000 倍！Conner CP 2045 具有 2 个 2.5 英寸的盘片，具有 40 MB 容量，1990 年制造。图中最小的磁盘是 Integral 公司的 1820。这种 1.8 英寸的单片磁盘容量为 20 MB，1992 年制造。

精解：这些高级接口允许磁盘控制器带有内置的缓存。最近的因 CPU 请求而传送的数据可以保存在缓存中，以便再次访问时获得更快的存取速度。这种磁盘采用写直达策略，这样写缺失时不需要更新数据，同时带有预取算法用于预测请求。控制器也使用一个指令队列，允许磁盘来决定执行这些指令的顺序，以便在得到正确行为的前提下达到最大的性能。当然，当比较磁盘的性能时，这种特点使得测量磁盘性能的工作变得更为复杂，也使得工作负载的选择越发重要。

特性	Seagate ST33000655SS	Seagate ST31000340NS	Seagate ST973451SS	Seagate ST9160821AS
磁盘直径（英寸）	3.50	3.50	2.50	2.50
格式化数据容量（GB）	147	1000	73	160
磁面的数量（磁头）	2	4	2	2
旋转速度（RPM）	15 000	7 200	15 000	5400
内部磁盘高速缓存的大小（MB）	16	32	16	8
外部接口，带宽（MB/s）	SAS，375	SATA，375	SAS，375	SATA，150
支持的传输速率（MB/s）	73～125	105	79～112	44
最小寻道时间（读/写）（ms）	0.2/0.4	0.8/1.0	0.2/0.4	1.5/2.0
平均寻道时间（读/写）（ms）	3.5/4.0	8.5/9.5	2.9/3.3	12.5/13.0
平均故障时间 MTTF（小时）	1 400 000@25℃	1 200 000@25℃	1 600 000@25℃	—
年平均故障率 AFR（百分比）	0.62%	0.73%	0.55%	—
接触，启动－停止周期	—	50 000	—	>600 000
保质期（年）	5	5	5	5
不可恢复的每位读错误	$<1/10^{16}$ 扇区	$<1/10^{15}$ 扇区	$<1/10^{16}$ 扇区	$<1/10^{14}$ 扇区
温度，震动（操作）	5～55℃，60 G	5～55℃，63 G	5～55℃，60 G	0～60℃，350 G
大小：尺寸（英寸），重量（磅）	1.0″×4.0″×5.8″，1.5lbs	1.0″×4.0″×5.8″，1.4lbs	0.6″×2.8″×3.9″，0.5lbs	0.4″×2.8″×3.9″，0.2lbs
功率：操作/空闲/等待（瓦特）	15/11/—	11/8/1	8/5.8/—	1.9/0.6/0.2
GB/英寸^3，GB/W	6 GB/英寸^3，10 GB/W	43 GB/英寸^3，91 GB/W	11 GB/英寸^3，9 GB/W	37 GB/英寸^3，84 GB/W
2008 年的售价，美元/GB	~250 美元，~1.70 美元/GB	~275 美元，~0.30 美元/GB	~350 美元，~5.00 美元/GB	~100 美元，~0.60 美元/GB

图 6-5　2008 年产自同一厂商的 4 种型号磁盘的特性

左边 3 个磁盘用于服务器和台式机，最右边的用于笔记本电脑。注意，第三个磁盘的直径仅有 2.5 英寸，但它是高性能的磁盘，具有最高的可靠性和最快的寻道时间。这里展示的磁盘或者具有 SCSI（SAS）接口（用于多种系统的标准 I/O 总线），或者具有 ATA（SATA）接口（用于 PC 机的标准 I/O 总线）。缓存的传输率比磁盘盘面的传输率快 3～5 倍。SATA 的 3.5 英寸磁盘每 GB 的价格低廉的主要原因是 PC 市场的高强度竞争。但 SATA 在传输率上与 SAI 存在很大差距，因为 SAS 具有更快的转速和寻道时间。这些磁盘的服务时间是 5 年。注意，这里的 MTTF 是假定在正常功耗和温度环境下的值。如果温度和震动没有很好地控制，磁盘的寿命将大幅缩短。在 www.seagate.com 网站可以得到关于这些磁盘更多的信息。

小测验

以下关于磁盘的说法，哪一个是对的？

A. 3.5 英寸的磁盘会比 2.5 英寸的磁盘每秒钟执行更多的 I/O 操作（吞吐率大）。

B. 2.5 英寸的磁盘每瓦功耗提供最高的数据率。

C. 对于一个高容量的磁盘，顺序地读其内容将需要好几个小时。

D. 对于一个使用具有 512 字节扇区的随机扇区磁盘来说，读数据需要好几个月。

6.4　快闪式存储器

有很多取代磁盘技术的发明，但是不幸都失败了：CCD 内存、磁泡存储器、全息存储器都曾如此。每次当一种新的技术即将推出时，磁盘技术总是按照早就期望的那样取得技术上的飞跃，造价相应下降，使与之竞争的产品变得在市场上不具备吸引力。

第一种成气候的挑战者是快闪式存储器（flash storage）。这种半导体存储器和磁盘一样是非易失性的，但是延时却只是磁盘的 1‰ ~ 1%，而且它尺寸小，功耗低，抗震性更好。同样重要的是，由于快闪式存储器在移动电话、数码相机、MP3 播放器中被大量应用，于是就市场来说，值得投入财力推动快闪式存储器技术的发展。最近，快闪式存储器的价格每年每 GB 下降近 50%。在 2008 年，每 GB 的快闪式存储器的价格降到了 4 ~ 10 美元，或者说大概比磁盘贵 2 ~ 40 倍，是 DRAM 价钱的 1/5 ~ 1/10。图 6-6 将三类基于快闪式存储器的产品做了比较。

特性	Kingston SecureDigital (SD) SD4/8 GB	Transend Type I CompactFlash TS16GCF133	RiDATA Solid State Disk 2.5 英寸 SATA
格式化数据容量（GB）	8	16	32
每扇区的字节数	512	512	512
数据传输速率（读/写 MB/s）	4	20/18	68/50
操作功耗/等待（W）	0.66/0.15	0.66/0.15	2.1/—
大小：高 × 宽 × 深度（英寸）	0.94 × 1.26 × 0.08	1.43 × 1.68 × 0.13	0.35 × 2.75 × 4.00
重量（克）	2.5	11.4	52
平均故障间隔时间（小时）	>1 000 000	>1 000 000	>4 000 000
GB/英寸^3，GB/W	84 GB/英寸^3，12 GB/W	51 GB/英寸^3，24 GB/W	8 GB/英寸^3，16 GB/W
最高售价（2008）	~30 美元	~70 美元	~300 美元

图 6-6　三类快闪式存储器产品特性的比较

CompactFlash 标准封装是 1994 年 Sandisk 公司为便携式 PC 的 PCMCIA-ATA 卡而推出的。因为遵循 ATA 接口，它模拟了一个磁盘的接口，包括寻道指令、逻辑磁道等。RiDATA 产品模拟 SATA 的 2.5 英寸接口。

尽管每 GB 快闪式存储器的价格比磁盘要贵，移动设备中快闪式存储器还是很受欢迎的，部分原因在于它的尺寸小。导致的结果是，直径为 1 英寸的磁盘从一些嵌入式市场上消失了。例如，2008 年 Apple iPod Shuffle 的具有 1 GB 容量的 MP3 播放器售价为 50 美元，同时最小的具有 4 GB 容量的磁盘售价比它还要贵。

快闪式存储器是一种电可擦写、可编程的只读存储器（EEPROM），第一种快闪式存储器被称作 NOR 快闪式存储器，因为它的存储单元和一个标准的或非门很相似，成为其他 EEPROM 存储器的直接竞争者并且像所有的存储器一样，是可随机寻址的。几年后，NAND 快闪式存储器具备更高的存储密度，但是存储器仅仅能够以块的形式被读写，因为那些为随机访问而设计的线被去掉了。NAND 快闪式存储器的每个 GB 相当便宜，因而比 NOR 快闪式存储器更受欢迎。

图 6-6中的所有产品都是使用的 NAND 快闪式存储器。图 6-7 比较了 NOR 和 NAND 快闪式存储器的关键特性。

特性	NOR 快闪式存储器	NAND 快闪式存储器
典型的应用	BIOS 存储器	USB key
最小的访问规模（字节）	512 字节	2048 字节
读时间（us）	0.08	25
写时间（us）	10.00	1500 到擦写 +250
读带宽（MB/s）	10	40
写带宽（MB/s）	0.4	8
使用次数（每个单元的写次数）	100 000	10 000 ~ 100 000
最高售价（2008）	65 美元	4 美元

图 6-7　比较 2008 年间 NOR 和 NAND 快闪式存储器的特性
无论它们的访问规模如何，这些器件都可以按字节和 16 位字进行读操作。

与磁盘和 DRAM 不同，但类似别的 EEPROM 技术，快闪式存储器的位可能会被用坏（见图 6-7）。为了应对这种局限性，大多数 NAND 快闪式存储器产品使用一个控制器采用重映射块来分布写操作，目的是将写次数多的块转移到写次数少的块。这项技术被称作磨损测量（wear leveling）。使用该技术，移动电话、数码相机、MP3 播放器或者密码狗（memory key）这样的消费类产品就不太可能超过快闪式存储器的写限制。这种控制器降低了快闪式存储器的潜在性能但确实必要，除非采用高层软件监控块的磨损。然而，控制器也能通过屏蔽那些在制造中损坏的存储单元来提高产品的良好率。

写限制是快闪式存储器在桌面系统以及服务器中使用不广泛的一个原因。然而在 2008 年，销售了第一批采用快闪式存储器替代硬盘的笔记本电脑，它们尺寸更小、电池时间更长、启动时间更快，且价格相当实惠。如图 6-6 所示，快闪式存储器也成为标准磁盘的一部分。结合两种思路，混合硬盘可包括一个 GB 级的快闪式存储器，这样笔记本电脑就可以更快地启动，而且通过让磁盘更多地保持空闲而节省能量。

未来几年，快闪式存储器将在很多基于电池供电的设备中战胜磁盘。随着容量的增加，以及价格的持续下降，也许会看到快闪式存储器在桌面系统和服务器市场中因为高性能和低功耗而取得机会。

小测验

关于快闪式存储器，下面哪些说法是正确的？

A. 和 DRAM 一样，快闪式存储器也是一种半导体存储器。
B. 和磁盘一样，如果掉电，快闪式存储器不会丢失信息。
C. NOR 快闪式存储器读数据的访问时间和 DRAM 差不多。
D. NAND 快闪式存储器的读带宽和磁盘的差不多。

6.5　连接处理器、内存以及 I/O 设备

在计算机系统中，不同的子系统必须具备连接其他子系统的接口。例如，内存和处理器需要通信，同样，处理器和 I/O 设备也需要通信。多年以来，这些工作都是由总线来担当的。总线是一种共享的通信链接，它使用一组连线来连接多个子系统。总线结构的两个主要优点是功能多和成本低。通过定义一种互联方案，新的设备就很容易被添加进来，外围设备也可以在使用同类总线的计算机系统之间移动。而且，因为同一组连线被多个通信路径共享，所以总线具备较好的成本效益。

总线的主要缺点是它会产生通信瓶颈，这就限制了 I/O 设备的最大吞吐量。当 I/O 数据传输必须通过一个总线时，这条总线的带宽就会限制 I/O 的最大吞吐量。设计一个能够满足处理器的

要求，又能够连接大量的 I/O 设备的总线系统是一个巨大的挑战。

传统上总线被划分为**处理器 - 内存总线**[㊀]和 I/O 总线，处理器 - 内存总线比较短，通常是高速总线，用来连接内存系统以便得到最大的内存 - 处理器带宽。相反，I/O 总线比较长，可以在其上挂接多种类型的设备，而且各类设备的数据带宽差异很大。I/O 总线通常不会直接连接内存，而是要么通过处理器 - 内存总线，要么通过**底板总线**[㊁]来使用内存。其他总线如图形总线，因为特殊的功能而具备不同的特性。

总线的设计非常困难的原因之一是最高总线速度在很大程度上被物理因素所限制：总线的长度和设备的数量。这些物理上的限制使得我们不能以任意快的速度运行总线。原因之二，为了支持一系列延时和数据传输率各不相同的设备也使得总线的设计极富挑战性。

由于时钟偏斜（clock skew），以及信号反射的问题（见光盘中的附录 C），高速的并行线路很难实现，工业界从并行的共享总线转变到带开关的点对点高速串行互联。这样的 I/O 网络正在逐渐替代系统中的 I/O 总线。

由于这种转变的出现，本节在这一版的修订中，强调了连接 I/O 设备、处理器以及内存的基本问题，而不再仅仅关心总线。

6.5.1　互联基础

我们来考虑一个典型的 **I/O 事务**[㊂]。一个事务包括两个部分：发送地址和接收/发送数据。总线事务的界定取决于对内存所进行的操作。读事务是从内存传出数据（或者到处理器或者到一个 I/O 设备）；写事务则是向内存写入数据。为了避免歧义，我们使用输入和输出两个术语，它们从处理器的角度来定义：一次输入操作指的是从设备输入数据到内存，使得处理器可以读取；而一次输出操作是指把处理器写到内存的数据输出到设备。

I/O 互联总线是一种扩展机器和连接新外设的途径。为了简便起见，计算机业界发明了几套标准。这些标准作为一种规格为计算机制造商和外围设备制造商提供了操作规范。对于计算机设计者，标准保证了外设对于新的机器是可用的。而且，标准使得外围设备建造者相信用户能够将新设备挂接到计算机系统中。图 6-8 总结了五种流行的 I/O 标准的关键特性：Firewire，USB，PCI Express（PCIe），串行 ATA（SATA），串行连接的 SCSI（SAS）。它们把各种外设连接到台式机上，如键盘、相机和磁盘等。

传统的总线是**同步的**[㊃]，这就意味着总线的控制线中包含了一个时钟，以及一个和时钟相关的固定的通信协议。例如，处理器 - 内存总线为了执行读内存操作，需要在第一个时钟周期传输地址和读操作指令，并且使用控制线来指明请求的类型。内存可能被要求在第五个时钟周期以提供数据字的方式来做出响应。这种类型的协议使用小型的有限状态机很容易实现。由于这种协议是预先确定的，涉及很少的控制逻辑，总线可以运行得很快而且接口上的逻辑电路也会很小。然而，同步总线有两大缺点：第一，总线上的每个设备必须按照相同的时钟频率运行；第二，由于时钟偏斜（skew），如果总线相当快的话，总线不能做得太长（见光盘中的附录 C）。

这些问题导致了**异步互联**[㊄]的产生，这样的互联不需要时钟。由于它们没有时钟，所以异步总线可以满足很多不同设备的需要，而且异步总线可以延长，而不用担心时钟偏斜（skew）或

㊀ 处理器 - 内存总线（processor-memory bus）：连接处理器和内存的总线。通常比较短，速度高，和内存系统匹配，这样可以使得处理器和内存间的带宽达到最大。

㊁ 底板总线（backplane bus）：一个用来连接处理器、内存和 I/O 设备的单一总线。

㊂ I/O 事务（I/O transaction）：在一个互联上的一系列操作，包含了一个请求以及可能的回答，它们均可能包含数据。一个事务由一个请求发起，可能包含很多独立的总线操作。

㊃ 同步总线（synchronous bus）：这样的总线的控制线中包含了时钟信号，以及依赖这个时钟的固定通信协议。

㊄ 异步互联（asynchronous interconnect）：使用一个握手协议来协调而不用时钟；可以适应速度相异的不同设备。

者其他的同步问题。所有图 6-8 中的例子都是异步的。

特性	Firewire（1394）	USB 2.0	PCI Express	串行 ATA	串行连接的 SCSI
使用目的	外部使用	外部使用	内部使用	内部使用	外部使用
每通道的设备数	63	127	1	1	4
基本数据线宽度（信号数）	4	2	每通道 2 个	4	4
理论上的峰值带宽	50 MB/s（Firewire 400）或者 100 MB/s（Firewire 800）	0.2 MB/s（低速）、1.5 MB/s（全速）或 60 MB/s（高速）	每个通道 250 MB/s（1×）；PCIe 卡为 1×、2×、4×、8×、16× 或 32×	300 MB/s	300 MB/s
热插拔	是	是	依赖于格式参数	是	是
最大总线长度（铜线）	4.5 m	5 m	0.5 m	1 m	8 m
标准的名称	IEEE 1394，1394b	USB 实现论坛	PCI-SIG	SATA-IO	T10 协会

图 6-8　五种领域的 I/O 标准的关键特性

这些总线标准可用于计算机之间的互联电缆，或者是计算机内部的互联短线，或者是印制电路板上的互联。PCIe 可以支持同时的读和写，所以一些出版物在假定读和写带宽各半的情况下把它的带宽加倍。

为了协调发送者和接收者之间的数据传输，异步总线采用**握手协议**[⊖]。握手协议由一系列步骤组成，只有当发送者和接收者之间达成一致才能进行下一步。这种协议通过一套附加的控制线路来实现。

6.5.2　x86 处理器的 I/O 互联

图 6-9 展示的是一个传统的 PC 的 I/O 系统。处理器通过两个主要的芯片来连接外设。处理器旁边的那块芯片就是内存控制器集线器，通常称为北桥（north bridge），另一块与之相连的就是 I/O 控制器集线器，称为南桥（south bridge）。

北桥本质上是一个 DMA 控制器，把处理器连接到内存、图形卡和南桥芯片。南桥将北桥和各种 I/O 总线连接。Intel、AMD、NVIDIA 以及其他公司提供了很多将处理器和外界连接起来的芯片组。

图 6-10 展示了三个芯片组的例子。注意，AMD 处理器的 Opteron 版本及其后续产品包含了北桥芯片。因此外部世界连接到处理器上的芯片的数量减少了。

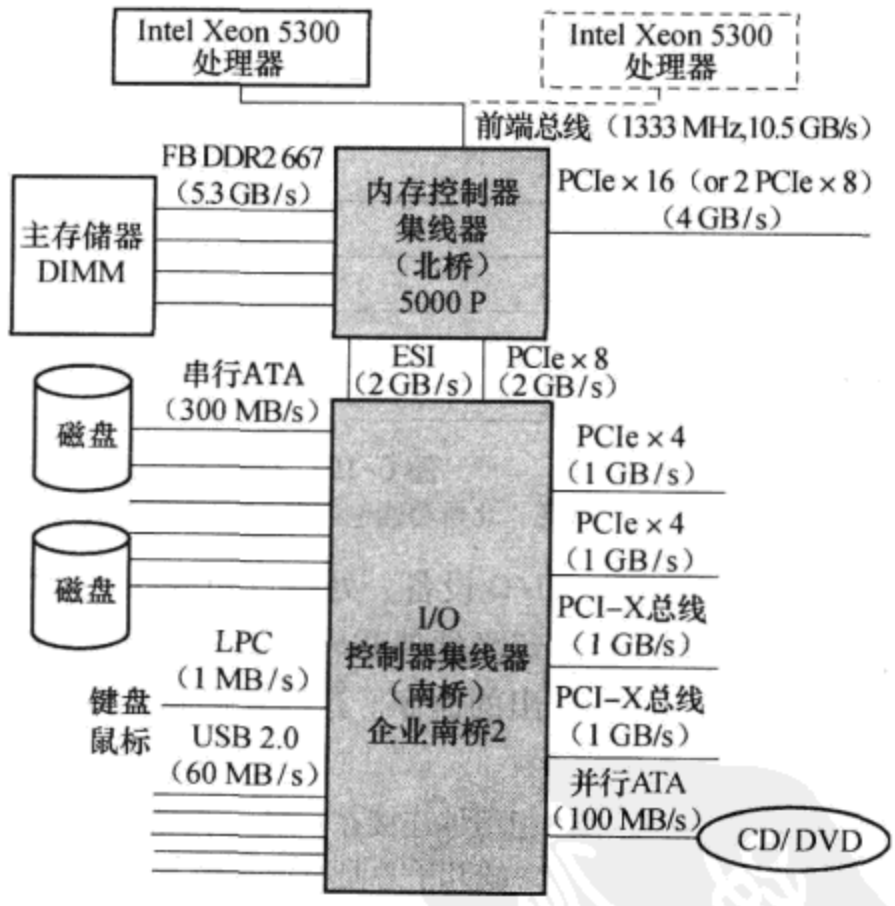


图 6-9　使用 Intel 5000P 芯片组的 Intel 服务器的 I/O 系统的结构

如果假定读操作和写操作各自占有一半的通信量，可以在 PCIe 中得到双倍的带宽。

随着摩尔定律的延续，很多 I/O 控制器曾经作为连接到 I/O 总线的可选卡，如今已经被集成到芯片组中。例如，AMD 的 Opteron X4 处理器和 Intel 的 Nehalem 处理器内部包含了北桥，而且 Intel 975 处理器的南桥芯片包含了一个 RAID 控制器（见 6.9 节）。

⊖ 握手协议（hand shaking protocol）：是为了协调异步总线传输而使用的一系列步骤，在这些步骤中，发送者和接受者只有在当彼此都对当前的步骤确认时，才可以进行下一个步骤。

	Intel 5000P 芯片组	Intel 975X 芯片组	AMD 580X CrossFiret
目标段	服务器	性能 PC	服务器/性能 PC
前端总线（64 位）	1066/1333 MHz	800/1066 MHz	—
内存控制器集线器（“北桥”）			
产品名称	Blackbird 5000P MCH	975X MCH	
管脚数目	1432	1202	
内存类型，速度	DDR2 FBDIMM 667/533	DDR2 800/667/533	
内存总线，宽度	4 ×72	1 ×72	
DIMM 的数目，DRAM/DIMM	16，1 GB/2 GB/4 GB	4，1 GB/2 GB	
内存容量的最大数量	64 GB	8 GB	
是否纠正内存错误	是	否	
PCIe/ 外部图形接口	1 PCIe ×16 或者 2 PCIe ×8	1 PCIe ×16 或者 2 PCIe ×8	
南桥接口	PCIe ×8，ESI	PCIe ×8	
I/O 控制器集线器（“南桥”）			
产品名称	6321 ESB	ICH 7	580X CrossFire
封装尺寸，管脚数目	1284	652	549
PCI-bus：宽度、速度	两个 64 位，133 MHz	32 位，33 MHz，6 主机	—
PCI Express 端口	三个 PCIe ×4		两个 PCIe ×16，4 PCI ×1
以太网 MAC 控制器，接口	—	1000/100/10 Mbit	—
USB 2.0 端口，控制器	6	8	10
ATA 端口，速度	一个 100	两个 100	一个 133
串行 ATA 端口	6	2	4
AC-97 音频控制器，接口	—	是	是
I/O 管理	SMbus 2.0，GPIO	SMbus 2.0，GPIO	ASF 2.0，GPIO

图 6-10　来自 Intel 和 AMD 的两组 I/O 芯片组

注意，北桥功能包含在 AMD 的微处理器中，就像 Intel Nehalem 处理器那样。

I/O 互联提供了 I/O 设备、处理器和内存之间的电气互联，同时定义了最低级的通信协议。在这个基础上，必须定义硬件和软件协议，用来控制 I/O 设备和内存之间的数据传输，详细说明处理器给 I/O 设备发出的命令。这些主题将在下一节中阐述。

小测验

网络和总线都可以把组件连接在一起。下列关于它们的说法中哪些是正确的？

A. I/O 网络和 I/O 总线几乎总是标准化的。

B. I/O 网络和 I/O 总线几乎总是同步的。

6.6　为处理器、内存和操作系统提供 I/O 设备接口

总线或者网络协议定义一个字或者数据块在一组线路上应当如何通信。为了能把数据从一个设备传输到用户程序的内存地址空间，还必须执行其他几个任务。本节主要讨论这些任务，并且回答如下问题：

- 如何将用户的 I/O 请求转换成设备指令并和该设备进行通信？
- 数据到底是如何从内存单元输入输出的？
- 操作系统扮演的是什么角色？

在回答这些问题的过程中，我们会了解到操作系统在处理 I/O 操作中扮演主要角色，它是请求 I/O 操作的程序和硬件之间的接口。

操作系统的职责由 I/O 系统的三个特性决定：

1）多个程序通过使用处理器来共享 I/O 系统。

2）I/O 系统通常使用中断（外部产生的异常）来传输和 I/O 操作有关的信息。因为中断导致向内核态或超级用户态的转变，中断必须由操作系统（OS）来处理。

3）I/O 设备的低级控制相当复杂，因为它需要管理一组并发事件，而且正确的设备控制通常具有非常详细的需求。

硬件 软件接口

以上列出的 I/O 系统的三个特性导致 OS 必须提供以下几种不同的功能：

- OS 要保证用户程序只能访问 I/O 设备中用户有权力访问的那个部分。如果某个文件的所有者没有给某个程序访问权，OS 不允许该程序对磁盘上的这个文件进行读或者写操作。在一个有共享 I/O 设备的系统中，如果用户程序能够直接执行 I/O 操作，那么就没有办法提供这种保护了。
- OS 通过提供处理低级设备操作的例程为访问设备提供一种抽象。
- OS 就像它处理一个程序产生的异常一样，处理由 I/O 设备产生的中断。
- OS 设法对共享的 I/O 资源提供公平的访问，为了提高系统吞吐量而对访问进行调度。

为了代表用户程序实现这些功能，操作系统必须能够与 I/O 设备通信并且阻止用户程序直接与 I/O 设备通信。这要求实现以下三类通信：

1）OS 必须能给 I/O 设备提供指令。这些指令不但包括像读和写这样的操作，而且包括像磁盘寻道之类的其他设备操作。

2）当 I/O 设备已经完成一个操作或者遇到一个错误时它必须能够通知 OS。例如，当磁盘完成一次寻道时，它就会通知 OS。

3）数据必须能在内存和 I/O 设备之间传输。例如，从磁盘中读取的数据块必须能够传送到内存中去。

在接下来的几节中，我们将看到这些通信是如何实现的。

6.6.1　给 I/O 设备发送指令

为了给 I/O 设备发送命令，处理器必须能够寻址这个设备并且能提供一个或者多个命令字。有两种方法用来寻址设备：内存映射 I/O 和特殊 I/O 指令。在内存映射 I/O[⊖]中，地址空间的一部分分配给 I/O 设备。对这些地址的读和写被解释成 I/O 设备的指令。

例如，写操作能用来向 I/O 设备发送数据，在这种情形下，这个数据将被解释成一个命令。当处理器把地址和数据放到内存总线上时，内存系统将忽略这个操作，因为这个地址指明了这个内存空间的部分是用于 I/O 的。但是设备控制器识别这个操作，会把数据记录下来，并把它作为命令传送到设备中。用户程序不能直接发布 I/O 操作，因为 OS 没有为分配给 I/O 设备的地址空间提供访问权限，因此这些地址就能受到地址变换机制的保护。内存映射 I/O 也能够通过对选定地址的写或者读操作来传输数据。设备使用地址来确定命令的类型，而且数据可以由一个写操作提供或者由一个读操作获得。不管在哪种情况下，这些地址的编码包含设备标识、处理器和设备之间的传输类型。

⊖ 内存映射 I/O（memory-mapped I/O）：一种 I/O 策略，地址空间的一部分被分配给 I/O 设备，而且读和写这些地址被解释为 I/O 设备的指令。

为了完成一个程序的请求，执行数据的读或者写操作通常需要几个独立的 I/O 操作。此外，处理器可能不得不在这些独立的指令之间执行询问设备的状态的操作，目的是确定指令是否成功完成。例如，一台简单的打印机有两个 I/O 设备寄存器——一个用来存储状态信息，另一个用来存储被打印的数据。状态寄存器包括一个完成位（done bit），每打印完一个字符，这个位由打印机负责设置；还有一个错误位（error bit），用来指明打印机被阻塞或者缺纸。要打印的每一个字节都存放在数据寄存器。处理器必须等到打印机设置好完成位，才能把下一个字符存放到缓存区。处理器还必须检查错误位以便确定是否出现了问题。每一个这样的操作都需要一个单独的 I/O 设备访问。

精解：代替内存映射 I/O 的另一种方法是在处理器中采用专用的 **I/O 指令**[⊖]。这些 I/O 指令能够指定设备号以及命令字（或者命令字在内存中的位置）。处理器和设备通过一组 I/O 总线的线路来实现通信。实际的命令能够在总线的数据线上传输。带有 I/O 指令的计算机例子是 Intel x86 和 IBM 370 计算机。通过规定在非内核态或非管态下执行的 I/O 指令非法而实达到阻止用户程序直接访问设备的目的。

6.6.2　与处理器通信

如上面的例子，处理器通过周期性地检查状态位以便决定能否执行下一个 I/O 操作，这个方法被称为**轮询**[⊜]。轮询是 I/O 设备与处理器通信的最简单方式。I/O 设备把信息放到一个状态寄存器中，处理器必须访问而获得这个信息。处理器完全控制和执行所有工作。

轮询以几种不同的方式使用。实时嵌入式应用使用轮询，因为其 I/O 的速度是预定的，I/O 开销更容易预测，从而对实时有帮助。我们将看到，即使 I/O 速率在某种程度上稍高一些，仍然可以使用轮询。

轮询的缺点是它浪费了大量的处理器时间，因为处理器的速度比 I/O 设备快很多。处理器可能已经读取状态寄存器多次了，却发现设备还没有完成相对较慢的 I/O 操作，或者自上次轮询以来鼠标还没有移动。当设备完成一次操作时，我们仍然得读取状态以便确定本次操作是否成功。

很久以前轮询接口的巨大开销就引起了关注，这导致了“中断”的发明。中断用来通知处理器，何时 I/O 设备需要处理器的注意。**中断驱动 I/O**[⊝]方式几乎被所有的系统使用，至少设备使用 I/O 中断通知处理器某个 I/O 设备需要引起注意。当一个设备想要通知处理器它已经完成了某种操作或者需要引起注意时，这件事就会引起处理器中断。

一个 I/O 中断类似于在第 4 章和第 5 章中见到的异常，但是有两点重要的区别：

1）对指令执行来说，I/O 中断是异步的。也就是说，中断与任何指令不相关，且不阻止指令的执行。它和缺页中断或者算术溢出那样的异常有很大的差别。我们的控制单元仅仅需要在开始执行一条新的指令时检查是否有挂起的 I/O 中断。

2）除了 I/O 中断产生之外，还要给出像确认哪个设备产生的中断这类更进一步的信息。而且，中断所代表的设备有不同的优先级。这些设备的中断请求具备不同的紧急程度。

为了向处理器传递诸如引起中断的设备标识这类信息，系统采用矢量中断或者异常原因寄存器。当处理器确认了中断，设备就把中断矢量的地址或者某些状态域发送给异常原因寄存器。结果，当 OS 取得控制权之后，就能知道引起中断的设备标识并且能够立即查询该设备。中断机制消除了处理器轮询设备的需求，从而使得处理器能集中“精力”运行程序。

⊖ I/O 指令（I/O instructions）：一种专用指令，用来给 I/O 设备发送指令，而且指定了设备号，以及指令字（或者内存中的指令字的地址。）。

⊜ 轮询（polling）：周期性地检查 I/O 设备的状态寄存器的过程，目的是确定是不是需要为设备服务。

⊝ 中断驱动 I/O（interrupt-driven I/O）：一种 I/O 策略，利用中断来指示处理器某个设备需要被关注。

6.6.3　中断优先级

为了对付 I/O 设备的不同优先级，大多数中断机制拥有多个优先级；UNIX 操作系统使用 4～6个级别。这些优先级指明了处理器处理中断的顺序。内部产生的异常和外部的 I/O 中断都有优先级；通常 I/O 中断优先级要比内部异常低。I/O 中断优先级也可能有多个，速度越高的设备拥有越高的优先级。

为了支持中断优先级，MIPS 提供了操作系统实现该策略的原语，这与 MIPS 处理 TLB 缺失类似。图 6-11 给出了关键寄存器，附录 B.7 中给出了细节。

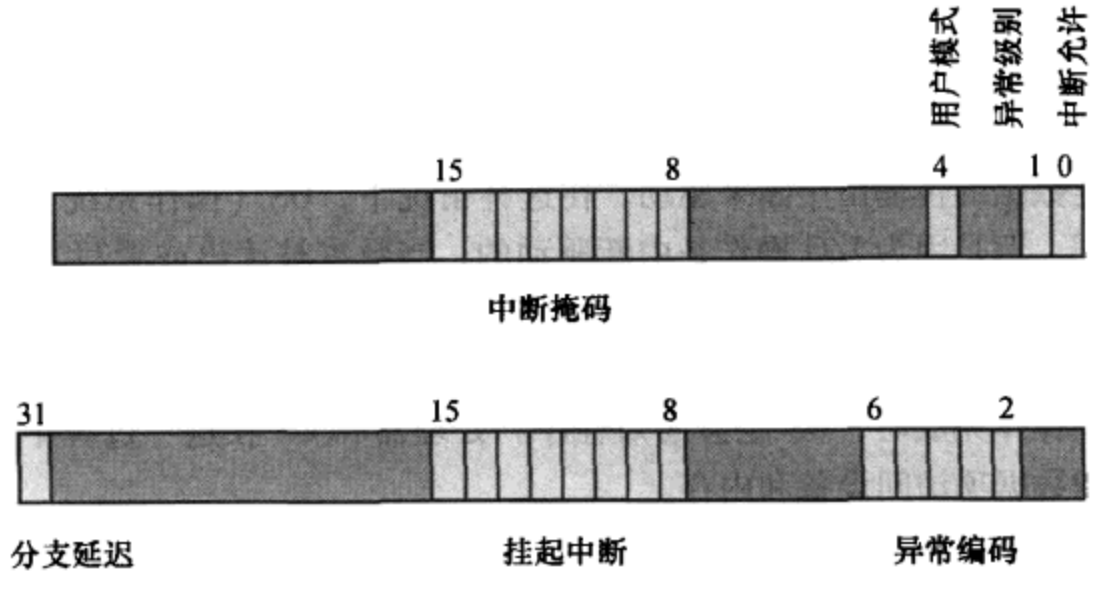


图 6-11　原因与状态寄存器

这个版本的中断原因寄存器对应的是 MIPS-32 体系结构。早期的 MIPS I 体系结构具有 3 组嵌套的核心态/用户和中断允许位，可以支持中断嵌套。在附录 B 的 B.7 节对这些寄存器有详细的介绍。

状态寄存器决定哪个设备能中断计算机。如果中断允许位（interrupt enable bit）为 0，那么谁都不能产生中断，中断掩码域（interrupt mask field）给出了更加细致的中断阻塞方案。原因寄存器中的挂起中断域（pending interrupt field）的每一位，都对应到掩码域中的一位。要允许相应的中断，掩码域中的相应位必须为 1。一旦发生中断，操作系统可以从状态寄存器的异常编码域（exception code field）找出中断发生的原因：0 表示中断发生，其他值表示第 5 章提及的异常。

下面是处理中断必须的步骤：

1）对挂起中断域和中断掩码域做逻辑与操作，看哪些被允许的中断发生了。用 mfc0 指令获得这两个寄存器的备份。

2）在这些中断中选择出优先级比较高的中断。软件惯例是最左边的有最高的优先级。

3）保存状态寄存器的中断掩码域。

4）改写中断掩码域，屏蔽所有与之相等或更低优先级的中断。

5）保存处理中断所需的处理器状态。

6）为了允许更高优先级的中断发生，将原因寄存器的中断允许位置 1。

7）调用适当的中断处理例程，执行之。

8）在恢复状态前，把原因寄存器的中断允许位置 0，这便于恢复中断掩码域。

在附录 B 中展示了一个简单 I/O 任务的异常处理例程。

中断优先级（interrupt priority level，IPL）是如何与这些机制对应起来的？IPL 是操作系统的发明。它保存在进程的内存中，每个进程都被赋予一个 IPL。在最低的 IPL 下，允许所有中断。相反，在最高的 IPL 下，则阻止所有中断。升高或降低 IPL 涉及状态寄存器的中断掩码域的改变。

精解：挂起中断和中断掩码域的最低两位是给软件中断用的，它们具有更低优先级。高优先级的中断

使用这些位确定中断的直接原因，把留下的任务交给更低优先级中断。一旦高优先级中断完成，低优先级中断就会被通知而且被处理。

6.6.4　在设备与内存之间传输数据

我们已经知道设备和处理器通信有两种不同的方法——轮询和 I/O 中断，它们形成了 I/O 设备和内存之间进行数据传输的基础。这两种技术在带宽比较低的设备中都工作得非常好，在这些设备中我们对于减少设备控制器的成本和接口的成本要比提供高带宽的传输更感兴趣。轮询和中断驱动把数据搬移和管理数据传输的负担加在处理器上。讨论完这两种方案之后，我们将介绍一种更合适高性能设备或很多设备的方案。

基于轮询，我们能利用处理器在设备与内存之间传输数据。在实时应用中，处理器从 I/O 设备寄存器装入数据再将它们存入内存。

另一种机制是让数据的传输由中断来驱动。在这种情况下，OS（操作系统）仍然从设备输入和输出少量的数据。但是因为 I/O 操作是中断驱动的，当数据被读出或者写入设备时，OS 完全可以从事其他任务。当 OS 识别出从设备发出的中断时，它首先读取状态位检查是否发生错误。如果没有错误，OS 就能够提供下一段数据，如一系列存储映射的写操作。当 I/O 请求的最后字节传输完和 I/O 操作完成时，OS 能够通知程序。处理器和 OS 在这一过程中完成所有的工作，为每一个传输的数据项访问设备和内存。

中断驱动 I/O 将处理器从不得不等待每一个 I/O 事件中解放出来，尽管如此，如果我们用这种方法从硬盘传出或者写入数据，开销仍然无法接受，因为使用磁盘传输会花费处理器很多时间。对于像硬盘这种高带宽的设备，传输主要由相对大块的数据组成（成百上千个字节）。因而计算机的设计者发明了一种不需要处理器的机制，让设备控制器直接往内存或从内存传输数据而不需要处理器干涉。这种机制叫做直接内存访问（DMA）[⊖]。设备和处理器通信时仍然用到中断机制，但仅限于 I/O 传输完成或出现错误时。

DMA 由一种独立于处理器，能在 I/O 设备和内存之间直接传输数据的专用控制器实现。DMA 控制器是**总线控制器**[⊜]，指挥自己和内存之间进行的读写操作，在一次 DMA 传输中有三个步骤：

1）处理器通过提供以下参数设置 DMA：设备标识、设备要执行的操作、内存中传输数据的源或目的地址、传输的字节数。

2）DMA 启动设备上的操作，执行互联仲裁。当数据可用时（数据来自内存或设备）就传输数据。DMA 设备提供数据读写的内存地址。如果某个请求需要传输多次数据，那么 DMA 单元就产生下一个内存地址并初始化下一次传输。采用这种机制，一个 DMA 单元能够完成一次长度可能为数千个字节的完整传输，而不需打扰处理器。很多 DMA 控制器包含一些存储单元，使它们能灵活地应对在传输中或者等待总线控制权时产生的延时。

3）一旦 DMA 传输完成，控制器向处理器发出中断，然后处理器通过询问 DMA 设备或检查内存决定整个操作是否成功完成。

在一个计算机系统中可能有多个 DMA 设备。例如，在一个采用单条处理器 - 内存总线和多条 I/O 总线的系统中，每个 I/O 总线控制器通常包含 DMA 处理器，用于处理在 I/O 总线上设备和内存之间的任何传输。

与轮询方式或中断驱动 I/O 不同，DMA 可用作硬盘的接口，这样不用在单次 I/O 操作中消

⊖ 直接内存访问（direct memory access，DMA）：一种提供设备控制器的机制，具备能够从内存传输数据，而不需要处理器介入的能力。

⊜ 总线控制器（master）：处于 I/O 互联中的一个单元，能够发起传输请求。

耗全部的处理器周期。当然，在处理器也要使用内存而内存正忙于处理一次 DMA 传输时，操作将被延迟。采用高速缓存，在大部分时间里处理器能避免内存访问，从而可以把大部分内存带宽留给 I/O 设备。

精解：一次中断请求包含多个 I/O 操作，处理它占用了处理器的时间，为了进一步减少中断对处理器的需求，I/O 控制器可以做得更智能。智能控制器通常叫做 I/O 处理器，也称 I/O 控制器或通道控制器（channel controller）。这些专用的处理器专门负责执行一系列 I/O 操作，称为 I/O 程序。这个程序可以驻留在 I/O 处理器中，或存放在内存里并能被 I/O 处理器读取。当采用 I/O 处理器时，操作系统通常建立一个 I/O 程序来指明所做的 I/O 操作以及所有读写操作的大小和传输地址。然后 I/O 处理器从 I/O 程序中取出这些操作，并且只在整个 I/O 程序完成时才向处理器发出中断。DMA 处理器本质上是专用处理器（通常是单片的，且不能编程）；而 I/O 处理器通常使用通用微处理器来实现，可以运行专用的 I/O 程序。

6.6.5　直接存储器访问和内存系统

当 DMA 被集成到 I/O 系统时，内存系统和处理器的关系就发生了变化。没有 DMA 时，所有对内存系统的访问都来自处理器，需要经过地址转换和高速缓存的访问，就好像是处理器自己发出的这些请求。采用 DMA 方式，就有了另外一条到内存系统的路径——一条不需要通过地址转换机制或高速缓存（cache）层次的路径。这种区别导致虚拟内存系统和采用高速缓存的系统产生了一些问题。这些问题通常用硬件技术和软件支持相结合的方法解决。

在虚拟内存系统中使用 DMA 困难的原因是页（page）有物理和虚拟两种地址。在有高速缓存的系统中使用 DMA 也会产生问题，因为一个数据项可能同时会有两个副本：一个在高速缓存中，一个在内存中。因为 DMA 处理器是直接向内存发出请求而不是通过高速缓存访问数据，所以 DMA 部件所看到的内存地址中的值可能和处理器看到的值不一样。思考如下这种情况：DMA 部件从磁盘中读数据并且直接存入内存。如果被 DMA 写入的数据的地址也在高速缓存中，那么当处理器从高速缓存中读取数据时就获得旧值。同样，如果高速缓存采用写回策略，当高速缓存中已更新数据但是尚未写回内存的时候，DMA 直接从内存读取的值就不是正确的数据了。这称为*失效数据问题*（stale data problem）或者*一致性问题*（coherence problem，见第 5 章）。

我们已经知道，传输 I/O 与内存之间的数据有三种不同的方式。从轮询方式到中断驱动再到 DMA 接口，我们将管理 I/O 操作的负担从处理器逐渐转移到一个更加智能的 I/O 控制器。这些方法具有解放处理器的优点，缺点是增加了 I/O 系统的成本。因此，在一个给定的计算机系统中，需要为连接到系统的 I/O 设备选择这三种方法中适合的一种。

在讨论 I/O 系统的设计之前，下一节将简要介绍 I/O 系统的性能度量。

小测验

关于三种 I/O 操作方式的排名，下面的论断哪些是正确的？

A. 如果要获得 I/O 设备中 I/O 操作的最低延迟，排序顺序为：轮询、DMA 以及中断驱动。

B. 按照单个 I/O 设备对处理器的使用率影响最小来排序，排序顺序是：DMA、中断驱动和轮询。

硬件/软件接口

在采用虚拟内存的系统中，DMA 是应该采用虚拟地址还是物理地址？采用虚拟地址最明显的问题是 DMA 部件需要将虚拟地址转换成物理地址。使用物理地址在 DMA 传输中的主要问题是传输不容易跨越一个存储页的边界。一个 I/O 请求如果跨域了存储页的边界，它传输的内存位置在虚拟内存中就不一定连续。因此，如果使用物理地址，DMA 传输必须被限制在一个页面之内。

使 DMA 工作在虚拟地址上的一种方法是允许系统能启动跨越页边界的 DMA 传输。在这样一个系统中，DMA 部件有少量的映射项用于保存传输中的虚拟地址到物理地址的映射。当 I/O 启动时，由操作系统来提供这些映射。使用这种映射，DMA 部件不需要关注被传输的虚拟页面的位置。

另外一种方法是将 DMA 传输切断为由操作系统执行的一系列的传输，它们中的每一个操作都限制在单一的物理页面中，最后将这些传输链接起来，交给 I/O 处理器或者智能 DMA 部件处理，由它们执行整个序列的传输；或者由操作系统逐一请求这些传输。

无论采用以上提及的哪一种方法，当包含某个页面的 DMA 传输正在进行时，操作系统都不能对该页进行重映射。

硬件 软件接口

避免 I/O 数据的不一致问题，可以使用下面三种方法之一。第一种方法是将高速缓存与 I/O 操作关联，这种方法确保了读操作能看到最新值而写操作会更新高速缓存中的数据。将高速缓存和所有的 I/O 相关联的成本很高，对处理器也存在潜在的负面影响，因为 I/O 数据很少立刻被用到，而且这可能会将运行程序所需的有用数据从 cache 中挤出去。第二种方法是让 OS 选择性地将高速缓存中与 I/O 读操作相关的数据项设置为无效；对于 I/O 写操作，强迫高速缓存写回内存（通常称为高速缓存刷新）。这种办法需要一些少量的硬件来支持，如果软件能够方便而有效地执行这个功能，采用软件可能会更加有效。因为这种大块高速缓存刷新的操作只在 DMA 进行块的访问时发生，所以它的相对出现次数不多。第三种方法是提供一种硬件机制，选择性地刷新高速缓存一些块（或者使之无效）。用硬件失效机制来确保高速缓存一致性的技术在多处理器系统中很常见，这种技术也被用在 I/O 中；在第 5 章对这些有详细的讨论。

6.7 I/O 性能度量：磁盘和文件系统的例子

如何比较不同的 I/O 系统的性能？这是一个复杂的问题，因为 I/O 性能取决于系统的很多方面，不同的应用侧重于 I/O 系统的不同方面。而且，设计在响应时间和吞吐量之间做权衡，这就要求我们不能只测量某一方面的性能。例如，尽量早处理请求通常会最小化响应时间，而把相关的请求合起来处理，则可以达到更大的吞吐量。相应地，我们把访问地址相近的请求放在一起，就可以增加磁盘的吞吐量。这样的策略对某些请求会增加响应时间，可能导致响应时间发生巨大变化。尽管吞吐量会更高，但有些基准程序对请求的最大响应时间有约束，导致一些优化方案可能会出现问题。

在本节，我们给出度量存储系统性能的一些实例。这些基准程序受一些系统特性的影响，包括磁盘技术、磁盘链接的方式、内存系统、处理器以及操作系统所提供的文件系统。

在讨论基准程序之前，需要澄清在术语和单位方面易混淆的问题。I/O 系统的性能取决于系统传输数据的速率。时钟频率的单位是 GHz = 10^9 周期每秒。传输速率取决于时钟频率，通常单位是 GB/s。在 I/O 系统中，GB 以 10 为底计算（1 GB = 10^9 = 1 000 000 000 字节），而不像主存以 2 为底计算（1 GB = 2^{30} = 1 073 741 824 字节）。除了增加混乱以外，这种差异还需要在 10 为底（1 K = 1000）和 2 为底（1 K = 1 024）的数之间进行转换，因为许多 I/O 访问中传输的数据块是以 2 为底来计算的。精确地转换度量方式会使得我们的例子复杂化，与之不同，我们在这里注明了这种差别并将这两种度量作为单位来解释会引入小的误差的事实。我们将在 6.12 节举例说明引入的误差。

6.7.1 事务处理 I/O 基准程序

事务处理[⊖]（TP）应用软件包含了对响应时间的要求和基于吞吐量的性能要求。另外，大多数的 I/O 访问的数据量都很小。因此，TP 应用主要关心 **I/O 速率**[⊖]，即每秒磁盘访问次数，而

⊖ 事务处理（transaction processing）：这是一种应用，包含了处理小的、短的操作（事务），这些事务通常需要 I/O 和计算。事务处理应用通常具有响应时间的需求和基于事务吞吐量的性能度量。

⊖ I/O 速率（I/O rate）：单位时间的 I/O 性能尺度，例如，每秒钟读操作数目。

不是**数据速率**[⊖]，即每秒钟传送数据的字节数。TP应用一般包括对大型数据库的修改操作，因此系统有响应时间的要求，同时还要能很好地处理某些种类的故障。这些应用非常关键，对成本也极度敏感。例如，银行一般都是用TP系统，因为要考虑一系列特性。这些特性包括确保事务不丢失，迅速处理事务和确保处理每个事务的开销最小。尽管面对故障可靠性是这类系统的根本要求，但是响应时间和吞吐量对构造性价比最优的系统非常关键。

现在已经开发出很多事务处理基准程序。其中最为著名的是，由事务处理委员会（Transaction Processing Council，TPC）开发的一套基准程序。

TPC-C，最初创建于1992年，模拟了一个复杂的查询环境。TPC-H对特殊决策支持进行建模——查询是不相关的，而且过去查询的信息不能用来优化未来的查询；所以查询的执行时间会很长。TPC-W是基于Web的事务基准程序，它模拟了一个面向事务型的Web服务器的活动，运行了数据库系统和潜在的Web服务器软件。TPC-App是一个应用程序服务器和网络服务基准程序。最新的TPC-E模拟了经纪商事务处理器负载。关于TPC的基准程序在http：//www.tpc.org中有描述。

所有的TPC基准程序按每秒的事务数来测量性能。另外，它们包含一个响应时间要求，因而只有当响应时间达到要求时才测量吞吐量性能。为了对真实世界的系统进行模拟，更高的事务率往往和更大的系统相关联，系统的大小既指用户数的多少，也指处理事务数据库的大小。因此，存储容量必须随着性能而扩展。最后，基准测试系统的系统开销也必须包含在内，以便准确地比较开销与性能之比的值。

6.7.2 文件系统和Web I/O的基准程序

除了给处理器提供基准程序，SPEC还提供文件服务器基准程序（SPECSFS）以及Web服务器基准程序（SPECWeb）。SPECSFS是使用文件服务器请求脚本来测量网络文件系统（Network File System，NFS）性能的基准程序；它也测试I/O系统的性能，包括磁盘、网络I/O和处理器。SPECSFS是针对吞吐量的基准程序，但是对响应时间也有重要的要求。SPECWeb是Web服务器基准程序，模拟多个客户向服务器请求静态和动态的页面和模拟客户向服务器传送数据的情况（见第1章）。

最近SPEC致力于功耗测量的测试程序开发。SPECPower测量小型服务器的功耗以及性能特性。

Sun公司最近公布了*filebench*，这是一个文件系统基准测试程序框架，它没有采用标准的负载，而是提供了一种语言来让你描述想在文件系统上运行的负载。然而，存在使用文件负载模拟常见文件系统的应用。

小测验

下面哪些是正确的？和处理器基准程序不同，I/O基准程序

A. 比起延迟，更关注吞吐量。

B. 为了达到要求的性能，需要扩展数据集合或者扩展用户数目。

C. 经常报告造价和性能。

6.8 设计I/O系统

设计者在I/O系统中会遇到两种主要的约束：时延约束和带宽约束。对于这两点，对通信模式的认知将影响整个设计和分析。

时延约束确保完成一次I/O操作的延迟时间被限制在某个数量范围内。一种简单的情况是认

⊖ 数据速率（data rate）：单位时间字节的性能尺度，例如，GB/s。

为系统是无负载的，设计者必须确保满足某些时延约束，这是因为这种限制对应用程序非常重要，或者设备为了防止某种错误必须接受某些有保证的服务。同样，在一个无负载系统中计算延迟相对比较容易，因为只用跟踪 I/O 操作的路径并累加单个延迟时间即可。

在有负载的情况下，得到平均时延（或者时延分布）是一个复杂得多的问题。这些问题可以通过排队论（当工作量请求的行为和 I/O 服务次数能够通过简单的分布来近似时）或者模拟（当 I/O 事件的行为很复杂时）的方法解决。这两个主题在本书讨论的范围之外。

给定一个工作负载，设计一个满足一组带宽约束的 I/O 系统是设计者需要面对的另一个典型的问题。换句话说，给定一个部分配置好的 I/O 系统，要求设计者平衡系统，以维持该系统预配置部分规定的可能到达的最大带宽。后面的这个设计问题是前者的简化版本。

设计这样一个系统的一般方法如下：

1）找出 I/O 系统中效率最低的链路，一般是 I/O 路径中限制设计的部件。依赖于工作负载，该部件可以存在于任何地方，包括处理器、内存系统、I/O 控制器或外设。工作负载和配置限制会决定这个效率最低的部件到底在哪儿。

2）配置这个部件以保持所需的带宽。

3）确定系统中其他部分的需求，配置它们以支持这个带宽。

理解这个方法最简单的方式是举一个例子。我们将在 6.10 节对 Sun Fire x4150 服务器做 I/O 系统的简单性能分析。

6.9　并行性与 I/O：廉价磁盘冗余阵列

第 1 章的 Amdahl 定律提醒我们在并行革命中忽略 I/O 是相当愚蠢的事情。下面的例子就会证明这一点。

举例　**I/O 对系统性能的影响**

假设存在一个运行时间为 100 s 的基准程序，其中 90 s 是 CPU 时间，剩余的是 I/O 占用的时间。如果 CPU 的数目每两年增长一倍，但是处理器的速度保持不变，I/O 时间保持不变，那么 6 年后运行该程序要耗费多少时间？

答案

我们知道

$$\begin{aligned}\text{耗费时间} &= \text{CPU 时间} + \text{I/O 时间}\\ 100 &= 90 + \text{I/O 时间}\\ \text{I/O 时间} &= 10\ \text{s}\end{aligned}$$

下面的表计算新的 CPU 时间和运行时间。

第 n 年以后	CPU 时间	I/O 时间	总耗费时间	I/O 时间（%）
0	90 s	10 s	100 s	10%
2	90/2 = 45 s	10 s	55 s	18%
4	45/2 = 23 s	10 s	33 s	31%
6	23/2 = 11 s	10 s	21 s	47%

6 年后，处理器速度的提高：90/11 = 8。

然而，运行速度的提高只有：100/21 = 4.7。

I/O 时间在运行时间中所占的比例从 10% 变为 47%。

因此，不但计算需要并行革命，I/O 也需要并行革命，否则，只要程序需要做 I/O 操作，实际上每个程序都需要有 I/O 操作，花费在并行化上的代价可能就会被浪费掉。

加速 I/O 性能是磁盘阵列的一个最初的动机（见光盘中的 6.14 节）。在 20 世纪 80 年代后期，高性能的存储器就是又大又昂贵的磁盘，如图 6-4 中的那些大磁盘。有一种观点是使用很多小的磁盘来替代少量的大磁盘，性能会得到提高，因为这样会得到更多的读磁头。这种转变对于多处理器来说是一个好的匹配，因为很多的读/写磁头意味着存储系统可以支持更多的独立访问，以及将大的传输遍布到很多个磁盘中。也就是说，你可以在每一秒钟得到高的 I/O 率以及高的数据传输率。除了有较高性能的优势，在价格、功耗以及面积方面这个策略也具备优势，因为小一些的磁盘通常比大一些的磁盘在每 GB 上更有效。

这种观点的缺点在于磁盘阵列导致可靠性变得更加糟糕。这些小而且便宜的磁盘和大磁盘相比具有低的 MTTF。而且更为重要的是，比如说，通过 50 个小磁盘来替换一个大磁盘，故障率将会至少上升 50 倍！

解决问题的办法是增加冗余，以便系统能够不丢失信息地处理磁盘故障。和少量的大磁盘的可靠性的解决策略相比，很多小的磁盘使用额外的冗余提高可靠性的代价相当小。因此，如果要构建一个廉价磁盘冗余阵列的话，可靠性是可以负担起的。这个研究导致了它的名字：redundant arrays of inexpensive disk[⊖]简称为 RAID。

回顾一下，尽管这个发明的目的是性能，但是可靠性成为 RAID 流行起来的关键原因。并行革命为 RAID 观点中原有的性能方面铺平了道路。本节的剩余部分总结了可靠性的几种途径以及各种途径对于性能和价格的影响。

那么需要多少冗余呢？需要额外信息来找到错误吗？组织磁盘上的数据和检查磁盘上的额外校验信息是否重要？定义该术语的文章对这些问题，从最简单但最昂贵的解决方案开始，给出了一个渐进式的回答。图 6-12 给出了演变的过程，并按额外校验磁盘数量来计算开销。为了使过程明了，作者给 RAID 的各个阶段编了号，这在今天仍然使用。

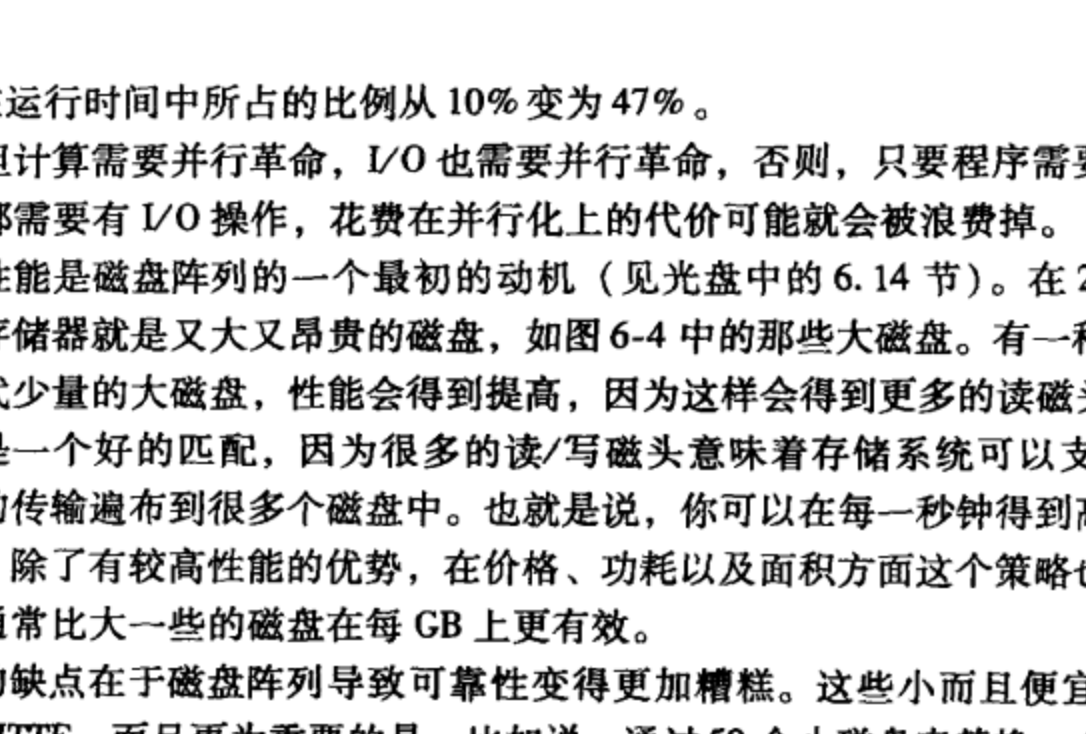


图 6-12　以 4 个数据磁盘为例列出每级 RAID 所需的额外的校验盘以及使用各级 RAID 的对应公司

图 6-13 和 6-14 说明了 RAID3、RAID4、RAID5 之间的区别。

⊖ 一种磁盘的组织形式，使用小而便宜的磁盘组成阵列，目的是提高性能和可靠性。

6.9.1　无冗余（RAID 0）

仅仅把数据分散到多个磁盘，称为条带化[⊖]，自动把访问强制分布到几个磁盘上。在一组磁盘上进行条带化使得这一组磁盘对于软件来说是一个大磁盘，从而简化了存储管理。而且对多个同时的访问来说有利于改进性能，因为多个磁盘可以同时操作。例如，视频编辑系统经常对它们的数据进行条带化，不需要像数据库那样关心可靠性问题。

RAID 0 的称谓有些不妥，因为它根本没有冗余。然而，RAID 的级别通常由操作员在创建系统时设置，而 RAID 0 经常被列为其中一个选项。因此，RAID 0 的说法就被广泛使用了。

6.9.2　镜像（RAID 1）

这种传统的容忍磁盘失效的方法，被称为镜像[⊜]或者影像（shadowing），使用比 RAID 0 多一倍的磁盘数。数据写入某个盘时，同样的数据会写入其冗余盘，因此始终存在信息的两份副本。如果一个磁盘出现故障，系统就转向其“镜像”读取内容以获得所需信息。镜像是最昂贵的 RAID 方案，因为它需要最多的磁盘。

6.9.3　错误检测和纠错码（RAID 2）

RAID 2 借用了主存常用的错误校验和恢复技术（参见光盘中的附录 C）。RAID 2 已经不再使用了，因此我们这里不做介绍。

6.9.4　位交叉奇偶校验（RAID 3）

增加可用性的开销可以减至 $1/n$，这里 n 为保护组[⊝]内磁盘的数目。我们不再为每个磁盘做一个原始数据的完全备份，而只需要加入足够的冗余信息以便在出错的时候恢复丢失的信息。读写操作在组内所有磁盘上进行，一个额外的磁盘存有校验信息以防错误的发生。RAID 3 在使用大数据集的应用（如多媒体和科学计算）中很流行。

奇偶校验（parity）就是这样的一个策略。不熟悉奇偶校验的读者可以把冗余磁盘想象成保存有其他磁盘所有数据的和。当一个磁盘出错时，用奇偶校验盘减去正常磁盘的数据的和；余数就是丢失的信息。奇偶校验就是模 2 下的求和。

与 RAID 1 不同，RAID 3 必须读很多磁盘才能确定丢失的数据。该技术背后的假设就是用更长的时间来恢复错误而用更少的冗余存储得到一个好的平衡。

6.9.5　块交叉奇偶校验（RAID 4）

RAID 4 使用同 RAID 3 数目比率一样大的数据磁盘和校验盘，但是访问数据的方式不同。奇偶校验码以块为单位存储，和一组数据块相关。

在 RAID 3 中，每次访问都用到所有磁盘。然而，某些应用偏重于较小的数据访问，允许并行地发生多个独立访问。这就是发明 RAID 4 ~ RAID 6 的目的。由于读操作需要校验每个扇区的错误检测信息来判断数据正确与否，只要少量的访问数据仍为同一个扇区，各磁盘上这些“小数据量的读操作”就可以独立地进行。在 RAID 环境中，小数据量访问在保护组中的一个磁盘发生，而大数据量访问需要用到保护组中的所有磁盘。

写操作是另外一个问题。看上去似乎每一次小数据量的写操作都需要访问其他磁盘信息，使用这

⊖ 条带化（striping）：将逻辑上连续的数据块分布到不同的磁盘上，得到比单个磁盘更高的性能。
⊜ 镜像（mirroring）：将相同的数据写到多个磁盘上，目的是增加数据的可用性。
⊝ 保护组（protection group）：共享一个公共校验磁盘的数据磁盘组或者数据块。

些信息重新计算新的奇偶校验值，如图 6-13 所示。一次“小数据量的写操作”需要读取旧数据和旧奇偶校验，添加新信息，接着把新的奇偶校验和写入校验盘，把新的数据写入数据盘。

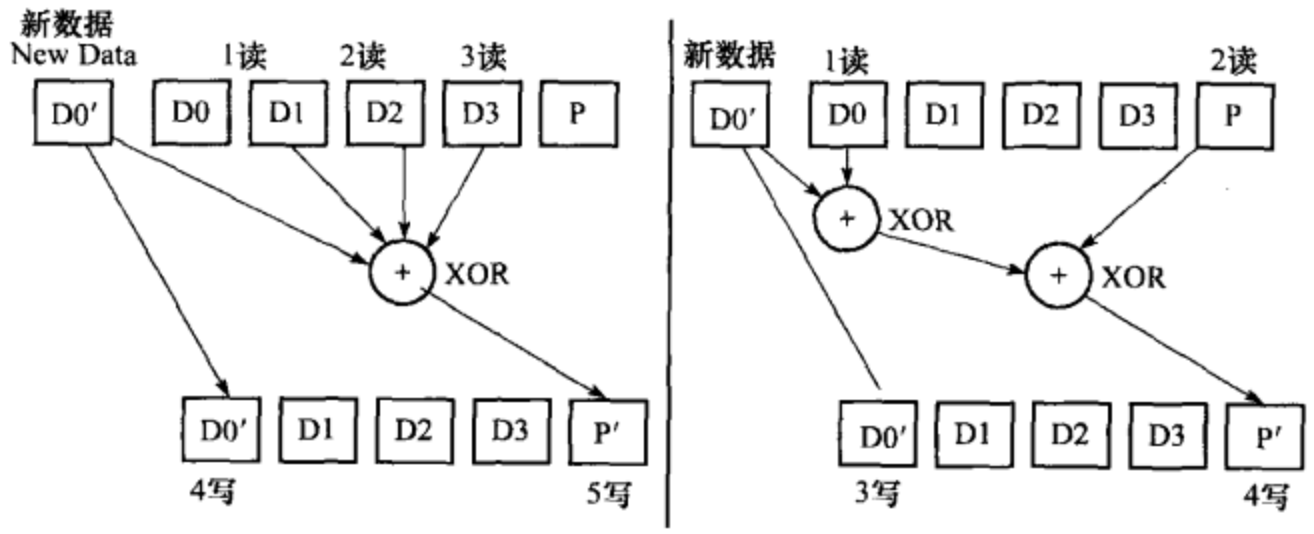


图 6-13　小数据量写更新在 RAID 3 和 RAID 4 上的比较

对小数据量写操作的优化减少了磁盘访问的数量，也减少了占用磁盘空间的数量。本图假设有 4 块数据和 1 块校验码。图左侧的 RAID 3 校验计算在加入块 D0′之前要读数据块 D1、D2 和 D3 才能计算新校验码 P′。（需要注意的是，新数据 D0′直接来自 CPU，所以不需要读磁盘来获取。）图右侧的 RAID 4 优化方法是读取旧值 D0 并与新值 D0′比较看是否改变。然后读取旧校验码 P，修改对应的位，形成新校验码 P′。使用或逻辑操作即可实现。图中把三次读磁盘（D1、D2、D3）和两次写磁盘（D0′，P′）替换为两次读磁盘（D0，P）和两次写磁盘（D0′，P′），前者访问了所有磁盘，而后者仅访问其中的两个磁盘。随着校验组大小的增加将使得优化的效果更加明显。RAID 5 亦使用同样的方式。

减小开销的关键在于校验码不过是信息的一个总和；通过观察写入新信息后哪些位发生了变化，我们只需改变校验盘上的对应位的信息即可。图 6-13 的右图说明了该方法。我们必须从要写的磁盘读取旧数据，用旧数据和新数据比较，看哪些位发生了变化。读旧奇偶校验和，改变对应的位，然后写入新数据以及新的校验和。这样，一次小数据量的写操作包含对两个磁盘的 4 次访问，而不是访问所有的磁盘。这种组织结构就是 RAID 4。

6.9.6　分布式块交叉奇偶校验（RAID 5）

RAID 4 有效地支持了大数据量读、大数据量写和小数据量读、小数据量写的混合操作。它的缺点是每次写操作都要更新校验盘，从而校验盘成为连续写的瓶颈。

为了解决校验-写瓶颈，校验信息可以分布到所有盘上，使得写操作不存在单一的瓶颈。这种分布式的奇偶校验组织方式就是 RAID 5。

图 6-14 展示了数据在 RAID 4 和 RAID 5 上是如何分布的。右图展示的是 RAID 5 组织方式，其中数据块每行的校验信息不再限定在单个磁盘。只要校验块不在相同的磁盘上，这种组织方式就使得多

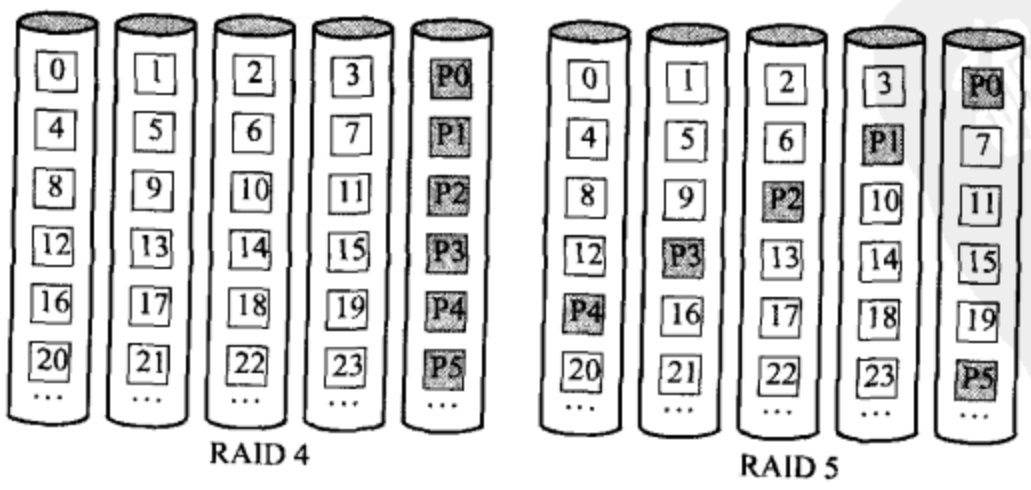


图 6-14　块交叉奇偶校验（RAID 4）与分布式块交叉奇偶校验（RAID 5）

通过将校验块分布到所有磁盘，一些小数据量的写操作可以并行进行。

个写操作可以同时发生。例如，右侧第 1 个写操作是向第 8 块写数据，需要同时访问 P2 中的奇偶值，从而需要访问第 1 个和第 3 个磁盘。右侧第 2 个写操作对第 5 块进行写，意味着要更新其校验块 P1，从而需要访问第 2 个和第 4 个磁盘，所以它可以和写第 8 个数据并发进行。对于左侧的组织结构来说，同样的写操作则需要修改第 5 个磁盘上的 P1 和 P2，这就构成了瓶颈。

6.9.7　P + Q 冗余（RAID 6）

基于奇偶校验的机制可使系统免受单个可自动识别的错误的破坏。当单个错误纠正机制不足以保护系统时，可利用奇偶校验对数据和另一个校验盘的信息进行二次计算。二次校验块可使系统从二次错误中恢复过来。因此，它的存储开销是 RAID 5 的两倍。图 6-13 中的小数据方法还能成立，只是现在更新 P 和 Q 信息需要访问 6 个盘而不是访问 4 个盘。

6.9.8　RAID 小结

RAID 1 和 RAID 5 广泛用于服务器；一项估计是服务器中 80% 的磁盘都使用了某种 RAID。

RAID 系统的弱点是修复。首先，为了避免在修复时数据不可用，阵列必须设计为不必关闭系统就能替换出错盘。RAID 拥有足够的冗余性以保证不间断的操作，但是**热交换**[⊖]磁盘对阵列和磁盘接口的物理及电路设计提出了要求。其次，修复中可能出现另外的错误，这样修复时间会影响丢失数据的概率：修复时间越长，另一错误引起丢失数据的概率越大。某些系统并不用等待操作员来装上好的磁盘，它们包含**应急备用**[⊜]，这样一旦检测出错误，数据就可以立即重建。操作员就可轻松地更换出错磁盘。最后，操作人员最终决定撤掉哪个磁盘。如图 6-3 所示，注意，操作员是人，因此他们有时候会撤掉好的磁盘导致不可恢复的磁盘错误。

除了设计可以修复的 RAID，还存在一些如何随着磁盘技术变化的问题。尽管磁盘厂商标称他们的产品具有很高的 MTTF，但是这些数据是在假设的情况下得到的。如果某个特定磁盘阵列遭遇了由于空调系统故障、糟糕的磁盘架设计、构建或者安装引起震动而引起温度周期变化，出错率将大大增加，增加 3 ~ 6 倍（见 6.12 节）。RAID 可靠性的计算假设多个磁盘失效之间是独立的，但实际上这些失效可能是相关的，因为环境引起的损伤可能会发生在阵列中的所有磁盘上。另一个问题是磁盘的带宽相对磁盘的容量变化得越来越慢，在一个 RAID 系统中修复一个磁盘的时间变得越来越长，这一点反过来增加第二次故障出现的概率。例如，在假设没有干扰时，一个 1000 GB SATA 磁盘可能需要花费 3 个小时来顺序读。假设这个损坏的 RAID 很可能被继续用来提供数据，重建过程就会被延长很多。除了增加时间外，另一个问题是在重建过程中一次读很多数据将意味着增加不可恢复的读媒体故障发生的概率，而不可恢复的故障将导致数据丢失。其他关于同时发生多个故障的看法是增加阵列中的磁盘数目以及使用 SATA 磁盘，这样比传统的商用磁盘慢一些，但具有更高的容量。

因此，这些趋势导致对防止系统免受多重故障的研究兴趣大大增加。所以 RAID 6 成为一种可选项，在实际中被使用。

小测验

下列关于 RAID 1、3、4、5 和 6 级的说法，哪些是对的？

A. RAID 系统依靠冗余来取得高可用性。

⊖ 热交换（hot-swapping）：系统运行的时候，替换一个硬件模块。

⊜ 应急备用（standby spares）：使用预留的硬件资源立即替换发生故障的模块。

B. RAID 1（镜像）的校验盘开销最大。

C. 对于小数据量的写操作，RAID 3（比特交叉奇偶校验）吞吐量最差。

D. 对于大数据量写操作，RAID 3、RAID 4 和 RAID 5 拥有同样的吞吐量。

精解：一个问题是镜像和条带化之间如何互相影响。假设要存储 4 个磁盘的数据，有 8 个磁盘可以使用。你会先把磁盘组成四对——以 RAID 1 的组织方式——然后把数据带状分布吗？还是创建两组 4 磁盘系列——以 RAID 0 为组织方式——之后在镜像中写入？RAID 术语把前者称为 RAID 1 +0 或 RAID 10（“带状镜像”），而把后者称为 RAID 0 +1 或者 RAID 01（“镜像带状分布”）。

6.10　实例：Sun Fire x4150 服务器

我们不但见证了微处理器构架的创新，也见证了软件交付的革新。不像传统的软件模型那样使用一个 CD 来销售，或者通过互联网络安装到你的计算机中，而是使用另外一种方式：软件作为一种服务。也就是说，使用自己的计算机，通过网络来工作，运行你所需要的软件来给你提供需要的服务。最流行的例子恐怕是 Web searching（网络搜索），但是还有那些提供图像编辑或者存储的服务、文本处理的服务、数据库存储以及虚拟世界等。如果你仔细看，就会发现你的个人计算机上所使用的每一个程序都能找到它们的服务版本。

这种转变导致了构建大的数据中心来保存计算机和磁盘中运行的成千上万的外部用户的服务。如果计算机被设计成数据中心，计算机应该是个什么模样？这些计算机肯定没有显示器和键盘。很明显，假设数据中心具有 10 000 个这样的机器，除需要考虑价格和性能外，空间效率以及功耗效率对于数据中心来说是相当重要的。

一个相关的问题是，这样的一个数据中心的存储该是什么样？虽然有很多的可选方案，一个流行版本是在每个处理器和内存中包含磁盘，使其作为构建一个大系统的基本单元。为了克服可靠性问题，应用程序自身具有很多的副本，而且应用程序负责保持各个副本之间的一致性并从错误中恢复。

IT 工业在为数据中心而设计的计算机在物理设计方面很大程度上达成了一致，尤其对于数据中心用来放计算机的机架来说形成了一些标准。最流行的是 19 英寸的机架，这些机架是 19 英寸宽（482.6 mm）。为这些机架而设计的计算机，被自然地标记为机架固定件（rack mount），但是也被称为子机架或者被称为架子或搁板（shelf）。由于机架中传统的格挡为匹配 1.75 英寸的（44.45 mm）搁板是分离的，这些间隔通常被称作机架单元或者单元（U）。最流行的 19 英寸的机架是 42U 高，即 42x1.75 或者 73.5 英寸高。搁板的深度各式各样。

因此，最小的机架固定件（计算机）是 19 英寸宽，1.75 英寸高，通常被称为 1U 计算机或者 1U 服务器。鉴于它们的尺寸，给它们取名为“比萨盒子”。图 6-15 说明了一个标准的具有 42 个 1U 服务器的机架。

图 6-16 所示为 Sun Fire x4150，1U 服务器的例子。最

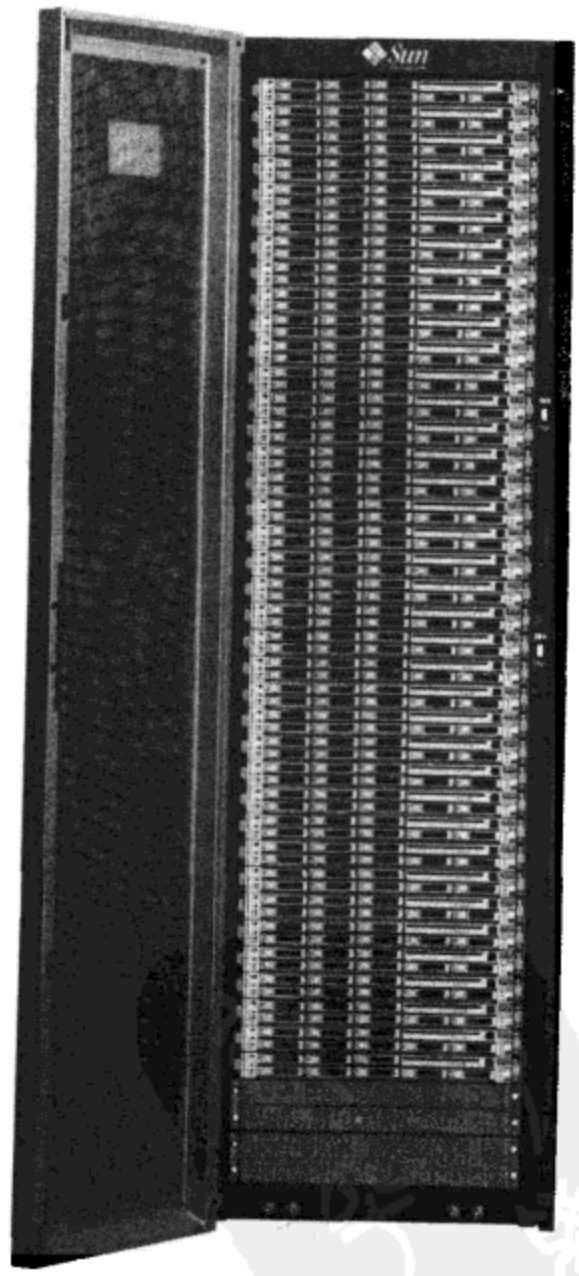


图 6-15　一个标准的具有 42 1U 服务器的 19 英寸的机架

这个机架具有 42 个 1U“比萨盒子”服务器。资料来源：http://gchelpdesk.ualberta.ca/news/07mar06/cbhd_news_07mar06.php。

大配置的 1U 盒子包含以下内容：

- 8 个 2.66 GHz 处理器，遍及 2 个插槽（2 个 Intel Xeon 5345）
- 64 GB 的 DDR2-667 DRAM，遍及 16 个 4 GB FBDIMM
- 8 个 15 000 RPM 73 GB SAS 2.5 英寸磁盘驱动器
- 1 个 RAID 控制器（支持 RAID 0、RAID 1、RAID 5 和 RAID 6）
- 4 个 10/100/1000 以太网口
- 3 个 PCI Express x8 接口
- 4 个外部和 1 个内部 USB 2.0 接口

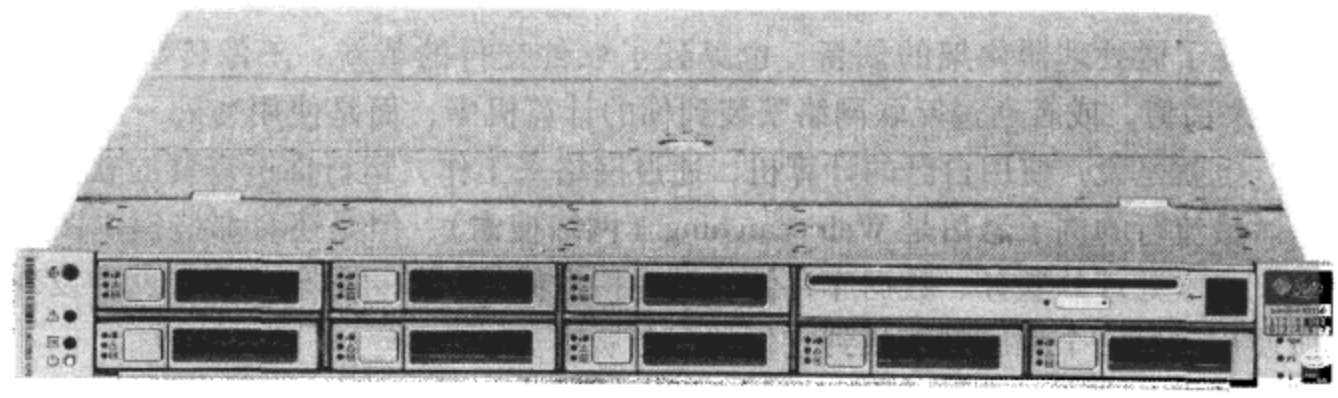


图 6-16　Sun Fire x4150 1U 服务器的前面和后面

尺寸是 1.75 英寸高乘于 19 英寸宽。8 个 2.5 英寸的磁盘可以从驱动器的前端放进去。右上角是一个 DVD 和两个 USB 接口。图下方标识着服务器后面的一些条目。服务器具有冗余的电源以及风扇，以便服务器在一个模块出现故障时可以持续工作。

图 6-17 展示了主板上芯片的连接以及带宽。图 6-9 和 6-10 描述了 Intel 5345 的 I/O 芯片组，图 6-5 描述了 Sun Fire x4150 中的 SAS 磁盘。

为了解释 6.8 节中的 I/O 系统设计中提到的几点建议，我们实施了一个性能评价的简单实例，为一个假设的应用找瓶颈。

举例　**I/O 系统的设计**

对于 Sun Fire x4150 作以下假设：

- 用户程序的每个 I/O 操作需要使用 200 000 条指令。
- 每个 I/O 操作，操作系统平均需要 100 000 条指令。
- 负载由 64 KB 的读组成。
- 每秒钟，每个处理器执行 10 亿条指令。

为一个满负载的 Sun Fire x4150 中的随机读和顺序读，计算最大的可以得到的 I/O 率。如果存在以下磁盘：假设读操作总是在一个空闲的磁盘上进行（例如，忽略磁盘的冲突），而且 RAID 控制器不成为瓶颈。

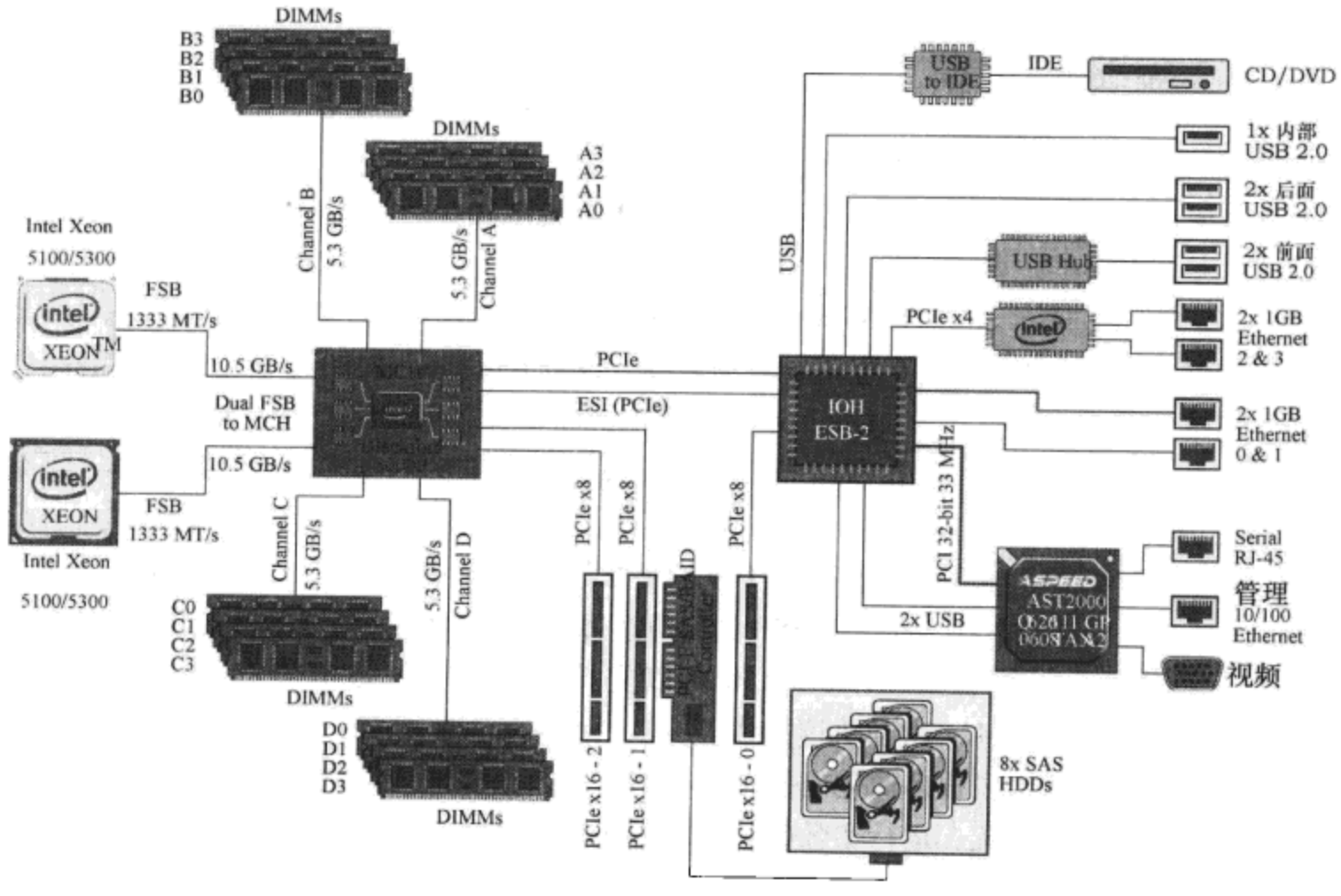


图 6-17　Sun Fire x4150 服务器的逻辑连接以及模块之间的带宽

3 个 PCIe 连接器允许 x16 板子被插入，但是仅仅为 MCH 提供 8 路的带宽。

资料来源："SUN FIRE™ X4150 AND X4450. SERVER ARCHITECTURE" 中的图 5（参见 www.sun.com/servers/x64/x4150/）。

答案

首先找到单个处理器的 I/O 率。每个 I/O 具有 200 000 条用户指令而且有 100 000 条 OS 指令。所以

$$\text{单个处理器的最大 I/O 率} = \text{指令执行速率} / \text{每个 I/O 的指令} = \frac{1 \times 10^9}{(200 + 100) \times 10^3} = 3333(\text{I/O/s})$$

单个 Intel 5345 插槽具有 4 个处理器，它可以执行 13 333 个 IOPS。两个插槽具有 8 个处理器可以执行 26 667 个 IOPS。

对于图 6-5 中的 2.5 英寸磁盘 SAS 中随机读和顺序读，我们来计算每个磁盘的 IOPS。假设很多情况下（见 6.3 节），不使用磁盘生产商的平均寻道时间，仅仅使用这个值的 1/4，单个磁盘的随机读所需要的时间是

$$\begin{aligned}\text{每个磁盘 I/O 的访问时间} &= \text{寻道} + \text{旋转时间} + \text{传输时间} \\ &= \frac{2.9}{4}\ \text{ms} + 2.0\ \text{ms} + \frac{64\ \text{KB}}{112\ \text{MB/s}} \\ &= 3.3\ \text{ms}\end{aligned}$$

因此，每个磁盘可以完成 1000 ms/3.3 ms 或者每一秒完成 303 个 I/O。而且每秒钟 8 个磁盘执行 2424 个随机读操作。

对于顺序读操作，传输大小被磁盘的带宽分割如下：

$$(112\ \text{MB/s})/64\ \text{KB} = 1750\ \text{IOPS}$$

8 个磁盘可以执行 14 000 个顺序的 64 KB 读操作。

我们需要研究，是否从磁盘到内存和处理器的路径是个瓶颈。从 RAID 卡到北桥芯片的 PCI Express 之间的连接开始，PCIe 的每个线都是 250 MB/s，所以 8 线宽的可达到 2 GB/s。

$$\text{PCIe x8 的最大 I/O 率} = \frac{\text{PCI 带宽}}{\text{每 I/O 的字节数}} = \frac{2 \times 10^9}{64 \times 10^3} = 31\,250(\text{I/O/s})$$

8 个磁盘顺序传输仅使用 PCIe x8 链路的一半的带宽。

一旦数据到达 MCB，需要被写入 DRAM。一个 DDR2667 MHz 的 FBDIMM 的带宽是 5 336 MB/s。一个 DIMM 可以执行

$$(5336\ \text{MB/s})/64\ \text{KB} = 83\,375\text{IOPS}$$

即使仅仅有一个 DIMM，内存也不会成为瓶颈，况且 Sun Fire x4150 中有 16 个 DIMM。

这个链路的最后一站是连接到 Intel 5345 插槽的北桥集线器的前端总线（Front Side Bus）。它的峰值带宽是 10.6 GB/s，但是由 7.10 节知道，可以得到的带宽不会高于峰值的一半。每个 I/O 的传输率是 64 KB，所以

$$\text{FSB 的最大 I/O 速率} = \text{总线带宽}/\text{每个 I/O 的字节数} = 5.3 \times 10^9/(64 \times 10^3) = 81\,540\ (\text{I/O/s})$$

每个插槽有一个前端总线，所以双 FSB 的峰值超过 150 000 IOPS，所以 FSB 还是不会成为瓶颈。

因此，一个完全配置的 Sun Fire x4150 可以支持 8 个磁盘的峰值带宽，支持磁盘中每秒 14 000次顺序读操作或者 2424 次随机读操作。

注意，这个例子作出了很多简化的假设。在实际中，很多这些简化对于 I/O 密集型的应用程序来说是不成立的。鉴于此，运行实际的负载或者相关的基准测试通常是评价 I/O 性能的唯一现实的途径。

在本节一开始就提到，很多新数据中心关心功耗、面积，也关心造价和性能。图 6-18 展示了一个完全配置的 Sun Fire x4150 服务器所需要的峰值功耗和空闲功耗。我们试着寻找能够节省 Sun Fire x4150 功耗的其他方案。

	部件			系统			
项目	空闲	峰值	数目	空闲		峰值	
单个 intel 2.66Ghz E5345 插槽，Intel 5000 MCB/IOH 芯片组，以太网控制器，电源，风扇…	154 W	215 W	1	154 W	37%	215 W	39%
另一个 Intel 2.66 GHz E5345 插槽	22 W	79 W	1	22 W	5%	79 W	14%
4 GB DDR2 – 667 5300 FBDIMM	10 W	11 W	16	160 W	39%	176 W	32%
73 GB SAS 15 K 磁盘驱动器	8 W	8 W	8	64 W	15%	64 W	12%
PCIe x8RIAD 磁盘控制器	15 W	15 W	1	15 W	4%	15 W	3%
总计	—	—	—	415 W	100%	549 W	100%

图 6-18　一个全面配置的 Sun Fire x4150 服务器的空闲功耗和峰值功耗

使用 29 个不同的配置来运行 SPECJBB 做这些实验，这样峰值功耗可能在运行不同的应用程序时表现出不同（资料来源：www.sun.com/servers/x64/x4150/calc）

举例　**I/O 系统的功耗评价**

重新配置 Sun Fire x4150 以减小功耗，假设上一个例子中的负载是 1U 服务器中唯一的活动。

答案

为了得到像前一个例子中那样每秒有 2424 次随机的 64 KB 的读操作，8 个磁盘和 PCI RAID

控制器我们全部需要。通过上面的计算，一个 DIMM 可以支持超过 80 000 IOPS，所以我们可以节省内存的功耗。Sun Fire x4150 最少有两个 DIMM 内存，所以我们可以节省 14 个 4 GB 的 DIMM 的功耗（以及造价）。一个插槽可以支持 13 333 IOPS，所以我们可以节省一个 Intel E5345 插槽。使用图 6-18 中的数据，系统的总体功耗现在是：

$$\text{空闲功耗(随机读)} = 154 + 2 \times 10 + 8 \times 8 + 15 = 253\text{W}$$

$$\text{峰值功耗(随机读)} = 215 + 2 \times 11 + 8 \times 8 + 15 = 316\text{W}$$

功耗减少了 1.6 ~ 1.7 倍。

原始系统每一秒完成 14 000 次 64 KB 顺序读操作。我们仍然需要所有的磁盘以及磁盘控制器，而且相同数目的 DIMM 可以处理更高的负载。这个工作负载超过了单个 Intel E5345 插槽的处理功耗，所以我们需要第二个插槽。

$$\text{空闲功耗(随机读)} = 154 + 22 + 2 \times 10 + 8 \times 8 + 15 = 275\text{W}$$

$$\text{峰值功耗(随机读)} = 215 + 79 + 2 \times 11 + 8 \times 8 + 15 = 395\text{W}$$

功耗减少了 1.4 ~ 1.5 倍。

6.11　高级主题：网络

不像其他 I/O 设备，网络越来越受到人们的关注，有很多关于网络的书和课程。对于没有学过网络方面课程也没有看过网络方面图书的读者，光盘中的 6.11 节给出了有关主题和术语的一个快速浏览，包括互联网络、OSI 模型、协议族（比如 TCP/IP）、远程网络（如 ATM）、局域网（如以太网），以及无线网络（如 IEEE 802.11）。

6.12　谬误与陷阱

谬误：磁盘的平均故障时间（MTTF）为 1 200 000 小时，即近 140 年，所以磁盘实际上不会有故障。

现今的磁盘厂商的市场策略会误导用户。这样的 MTTF 是如何计算出来的？早期，制造商要把上千块磁盘放到屋子里，让其运行几个月，然后计算出现故障磁盘的数目。他们用所有磁盘运行的总小时数的累加和除以出现故障的磁盘的数目来计算 MTTF。

问题在于这一数值远远超过了磁盘的使用寿命，一般预计为 5 年或者 43 800 小时。要使这么长的 MTTF 站得住脚，磁盘厂商辩解说该计算相当于用户购买磁盘，然后 5 年（设计的磁盘使用寿命）更换一次。这一声明表示，如果很多顾客（以及他们的曾孙）在接下去的一个世纪里都这样做，平均更换 27 次才可能出现故障，即大约 140 年。

一个更有用的测量办法是一年中出现故障磁盘所占的百分比，称为年失效率（annual failure rate，AFR）。假设有 1000 个 MTTF 为 1 200 000 小时的磁盘，每天 24 小时都在使用。如果你用可靠性相同的新磁盘来替代发生故障的磁盘，每年（8760 小时）出现故障的磁盘数为

$$\text{故障磁盘数} = (1000\text{ 个磁盘} \times 8760\text{ 小时} / \text{磁盘}) / (1\,200\,000\text{ 小时} / \text{故障}) = 7.3$$

换个说法，AFR 的值是 0.73%。磁盘生产商开始引用 AFR 的同时，也为用户提供 MTTB，给用户留下他们的产品优良的印象。

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 ATA 以及 SCSI 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 ATA 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］。

谬误：GB/s 的互联网络可以每秒传输 1 GB 的数据。

首先，你不可能 100% 利用任何计算机资源。对于总线来说，能获得峰值带宽的 70% ~80% 就算是幸运的。传输地址的时间、应答信号的时间、等待繁忙总线而阻塞的时间都是你不能 100% 利用总线的原因。

其次，存储器件的单位 GB 和带宽每秒 G 字节的定义不一致。像在 6.7 节中讨论过的一样，I/O 带宽的测量通常是以 10 为底的（也就是，1 GB/s = 10^9 字节/秒），而 1 GB 的数据通常是以 2 为底来测量的（也就是，1 GB = 2^{30}字节）。这种区别到底有多大呢？如果我们能 100% 使用带宽传输时间，那么 1 GB/s 的总线传输 1 GB 的数据的时间实际上等于

$$2^{30}/10^9 = 1\,073\,741\,824/1\,000\,000\,000 \approx 1.07 \text{ s}$$

陷阱：试图仅仅提供互联网络的特性，而忽视端到端的情形。

涉及的问题是，假设低层次的特性只能够在最高层次上得到，因此仅仅满足部分的通信需求。Saltzer、Reed 和 Clark［1984］给出了端到端问题（end to end argument），如下：

只有获得处于通信系统的另一端的应用程序的相关知识以及应用程序的帮助，被讨论的（被请求的）功能才能够被完全而且正确地描述。因此，那种假设被请求的功能作为通信系统自身一部分的特性是不可能的。

关于这一陷阱的一个实例是 MIT 大学中使用几个网关的一个互联网络，每个网关从另一个网关得到校验和给另一个网关。应用程序的编程人员假定校验和能保证正确性，错误地以为每个网关内存中保存的消息被保护得很好。当一个网关产生了瞬间故障：故障表现为每传输 100 万字节，其中的一对字节交换。随着时间的推移，一个操作系统的源代码通过这个网关被重复传输，因此破坏了代码。唯一的解决方法是：通过与程序清单列表对比，手工修复代码，纠正被破坏的代码！要是在终端系统中运行程序，计算校验和以及做校验，安全性就会得到保证。

要使中间状态的校验是有用的，得假设端到端的校验是可行的。端到端的校验能查出两个节点之间出错了，但是却不知道问题出在哪里。中间的校验可以发现哪个模块出错了。这两种都需要修复错误。

陷阱：把 CPU 的功能向 I/O 处理器转移，没有经过仔细的分析就期望提高性能。

尽管合理使用 I/O 处理器的确能提高性能，还是有很多误导人的陷阱。这个谬误的一个常见实例是使用智能 I/O 接口，由于启动一次 I/O 请求需要较高的开销，这会使它比处理器直接控制的 I/O 活动具有更长的时延（尽管这样将使处理器充分空闲，系统吞吐量可能还会有所提高）。更为普遍的情况是，I/O 处理器比主处理器性能低，系统性能会下降。结果是少量的主处理器时间被大量的 I/O 处理器时间所替代。工作站的设计者经常看到这两种现象。

Myer 和 Sutherland［1968］写了一篇经典文章讨论 I/O 控制器的复杂性和性能之间的平衡。借用宗教概念里的“转世轮回”，他们意识到他们陷入一个循环：不断增加 I/O 处理器的能力，直到它自己又需要一个简单的协处理器：

我们首先采用了一个简单的策略，然后添加一些我们觉得可能提高机器性能的命令和特性。逐渐地，［显示］处理器变得越来越复杂……最后显示处理器变成一个完整的带有一些特别图像功能的羽翼丰满的计算机。然后奇怪的事情发生了。我们被迫给处理器加上了辅助的处理器，而辅助处理器本身也变得越来越复杂。这时我们才发现烦人的真相。设计一个显示处理器变成了不会结束的循环过程。事实上，我们发现该过程是如此令人沮丧，于是我们称其为“转世轮回”。

陷阱：使用磁带来备份磁盘。

这既是一个谬误也是一个陷阱。磁带作为计算机系统的一个部分这个历史和磁盘的历史一

样长，因为它使用和磁盘类似的技术。因此历史上也有相同的改进密度。历史上著名的磁盘和磁带的性价比不同在于，一个密封的、旋转的磁盘访问时延比顺序的磁带短，但是磁带上可卸下来的磁带轴意味着许多磁带可以由一个读写头来使用，磁带可以很长，因此容量很大。所以，过去一个磁带就可以容纳许多磁盘的容量，而且它比磁盘每个 GB 便宜 10 ~ 100 倍，因此磁带是一个很有用的备份媒介。

一个主张是磁带必须紧跟磁盘，因为磁盘的创新能够帮助磁带的创新。这个主张很重要，因为磁带的市场太小，无力支持单独的研究和开发代价。市场小的一个原因则是台式机的用户通常不把磁盘内容备份到磁带上，所以台式机是目前磁盘的最大市场，却是磁带的小市场。

更广阔的市场使得磁盘的发展比磁带快得多。从 2000 年到 2002 年，容量最大的磁盘已经超过了容量最大的磁带。同时，ATA 磁盘的每 GB 价格也要低于磁带。磁带有兼容性的要求，而磁盘不存在这个问题；磁带机需要读写新版本和旧版本的磁带，支持最近四代的格式。而磁盘是密闭的系统，磁头只需要读磁盘所装入的盘片，这一优势解释了为什么磁盘发展得如此之快。

今天，一些组织完全放弃了磁带，他们使用网络和远程磁盘来复制数据。实际上，许多公司都以服务的形式提供了软件，该服务使用便宜的组件，在多个站点间的应用层复制数据。选择复制的地点保证了不会两边同时发生灾难，这就使得及时的数据恢复成为可能。（磁带的串行性特点导致的另一个严重的缺点是恢复时间长。）要使得这种方案要在经济上可行，需依赖于磁盘容量和网络带宽的发展，而这两个已经获得了越来越多的投资，因此近来磁盘比磁带获得了更好的发展成就。

陷阱：操作系统是调度磁盘访问的最好地方。

如 6.3 节所提到的，像 ATA 和 SCSI 这样的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 可以得到的最好的性能是将逻辑块的地址按照递增的顺序排序。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	9987	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	9987	1

依赖于数据在磁盘中的相对位置，如图 6-19 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：采用 I/O 系统一部分的传输峰值来表示性能或进行性能比较。

I/O 系统中有很多部件，从设备到控制器再到总线，都有它们自己的峰值带宽。在实践中，这些峰值带宽往往无法实现，这是因为这些峰值带宽的测量通常基于对系统不切实际的假定，或者其他的系统局限导致峰值带宽不可能实现。例如，在提到总线性能时，峰值传输速度的计算通常建立在一个不可能实现的内存系统之上。对于网络系统，启动通信的软件开销被忽略掉了。

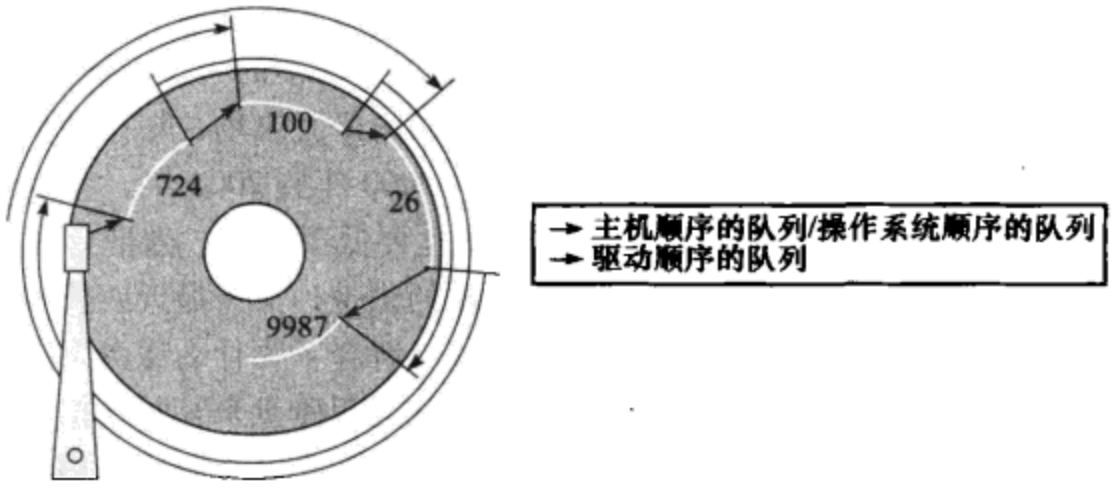


图 6-19　给出了 OS 调度与磁盘调度访问的例子，标记为宿主顺序和驱动顺序
前者完成 4 个读操作需要 3 个旋转周期，而后者完成 4 个读操作仅仅在一个 3/4 的旋转周期即可完成（资料来源：Anderson［2003］）。

一个 32 位、33 MHz 的 PCI 总线具有大约 133 MB/s 的峰值带宽。实际上，即使是比较长的传输，实际的内存系统也很难维持超过大约 80 MB/s 的速度。

Amdahl 法则提醒我们 I/O 系统的吞吐量将被 I/O 路径上最低性能的部件所限制。

6.13　本章小结

采用几个不同的特性评估 I/O 系统性能：可靠度；所支持 I/O 设备的多样性；I/O 设备的最大数目；成本；性能延迟时间和吞吐量。这些目标导致提供 I/O 设备接口的方案多样化。在低端和中端系统中，采用缓冲区的 DMA 是主流的传输机制。而在高端系统中，时延和带宽同样重要，成本是其次。具有有限的缓冲的多通道 I/O 设备通常被刻画为高端 I/O 系统。典型地，随着系统的增大，能在任何时间访问 I/O 设备上的数据（高可用性）变得越来越重要。结果是当我们扩大系统时，冗余和纠错机制变得越来越普遍。

人们对存储器和网络的需求正以史无前例的速度增长，部分原因是人们对实时信息的需求日趋增大。一项估计认为 2002 年产生的信息总量为 5 千兆兆字节——相当于美国国会图书馆拥有的文本的 500 000 份副本，而且每 3 年世界的信息总量翻一番［Lyman 和 Varian，2003］。

I/O 的未来的发展方向包括扩大有线网络和无线网络的范围，使几乎每个设备都拥有一个 IP 地址，在存储系统中扩展快闪式存储器的角色。

理解程序性能

I/O 系统的性能，无论是通过带宽还是时延来度量，都取决于设备和内存之间路径上的所有要素，包括产生 I/O 命令的操作系统。总线的带宽、内存和设备决定了进出设备的最大传输速度。同样，延迟时间由设备的延迟时间、内存系统或者总线带来的延迟决定。有效带宽和响应时延依赖于路径中其他可能引起资源冲突的 I/O 请求。最后，操作系统变成一个瓶颈。在某些情况下，OS 需要很多时间才能把 I/O 请求从用户程序转交给 I/O 设备，从而导致很大的延时。在其他情况下，操作系统在很大程度上限制了 I/O 带宽，因为操作系统能够支持并发 I/O 操作的数量是有限的。

记住，尽管性能有助于 I/O 系统的销售，但是用户必然关心 I/O 系统的可靠性和容量。

6.14　拓展阅读

I/O 系统的历史是引人入胜的。光盘中的 6.14 节给出了磁盘、RAID 闪存、数据库、因特网和万维网的一个简短的历史，以及以太网如何持续压制它的挑战者。

6.15　练习题

习题 6.1

图 6-2 根据设备的行为、合作者和数据速率，描述了很多 I/O 设备，然而，这些分类通常没有提供一个系统完整的数据流图。研究下面的这些设备该如何分类。

a.	视频游戏
b.	手持 GPS

6.1.1 [5] <6.1> 对于上表中的设备，指出 I/O 接口，并根据设备的行为和合作者将它们分类。

6.1.2 [5] <6.1> 对于这些在上一个问题中被指出的接口，估计它们的数据速率。

6.1.3 [5] <6.1> 对于这些在上一个问题中被指出的接口，确定是否数据速率或者操作速率是最好的性能尺度。

习题 6.2

故障间平均时间（MTBF）、替换平均时间（MTTR）和平均失效时间（MTTF），这些对于评价存储资源的可靠性以及可用性来说都是很有用的度量。研究这些概念，并且使用这些度量回答关于设备的问题。

	MTTF	MTTR
a.	5 年	1 周
b.	10 年	5 天

6.2.1 [5] <6.1，6.2> 为表中的每个设备计算 MTBF。

6.2.2 [5] <6.1，6.2> 为表中的每个设备计算可用性。

6.2.3 [5] <6.1，6.2> 在实际情况下，当 MTTR 接近 0.1 s 时，可用性如何？

6.2.4 [5] <6.1，6.2> 随着 MTTR 变得很高，例如一个设备很难修理，可用性如何？是不是意味着设备的可用性很低？

习题 6.3

在存储设备中，读操作和写操作的平均时间和最小时间是用来比较设备的常见手段。使用第 6 章的技术，根据下面表中罗列的特性，计算读磁盘以及写磁盘有关的时间。

	平均寻道时间	RPM	磁盘传输速率	控制器传输速率
a.	11 ms	7200	34 MB/s	480 MB/s
b.	9 ms	7200	30 MB/s	500 MB/s

6.3.1 [10] <6.2，6.3> 为表中的每个磁盘计算读或者写 1024 个字节需要的平均时间。

6.3.2 [10] <6.2，6.3> 为表中的每个磁盘计算读或者写 2048 个字节需要的最小时间。

6.3.3 [10] <6.2，6.3> 对于表中的每个磁盘，确定性能的主导因素。具体地讲，如果你能够从磁盘的某个方面提高磁盘的性能，你将选择哪种途径？如果没有主导因素，解释原因。

习题 6.4

最终，存储系统的设计需要考虑磁盘尺寸以及使用环境。不同的情形下需要不同的尺寸。让我们系统性地评价一下磁盘系统。探索不同的存储系统应当如何被评价，通过根据下面的应用，回答相应的问题。

a.	在线 NASA 卫星数据库
b.	视频游戏系统

6.4.1 [5] <6.2，6.3> 对于每种应用，在读和写的时候，减少扇区大小是否会提高性能？解释你的答案。

6.4.2 [5] <6.2，6.3> 对于每种应用，增大磁盘器的旋转速度会提高性能？解释你的答案。

6.4.3 [5] <6.2，6.3> 对于每种应用，假设 MTTF 下降，增大旋转速度将提高系统的性能？解释你的答案。

习题 6.5

快闪式存储器是传统磁盘驱动的真正竞争者之一。根据下面的应用，研究快闪式存储器的一些观点，回答问题。

a.	在线 NASA 卫星数据库
b.	视频游戏系统

6.5.1 [5] <6.2，6.3，6.4> 当我们向使用快闪式存储器构建的固态驱动转移时，假设数据传输速率保持恒定，磁盘的读次数会发生什么样的变化？

6.5.2 [10] <6.2，6.3，6.4> 假定造价是一个设计因素，每种应用程序能够从固态快闪式存储器驱动获益吗？

6.5.3 [10] <6.2，6.3，6.4> 假定造价不是一个设计因素，对固态快闪式存储器驱动来说，每个应用程序是不合适的？

习题 6.6

根据下面这些与快闪式存储器相关的性能，研究快闪式存储器的本质。

	数据传输速率	控制器传输速率
a.	34 MB/s	480 MB/s
b.	30 MB/s	500 MB/s

6.6.1 [10] <6.2，6.3，6.4> 计算表中的每个快闪式存储器读取或者写入 1024 个字节需要的平均时间。

6.6.2 [10] <6.2，6.3，6.4> 计算表中的每个快闪式存储器读取或者写入 512 个字节需要的最小时间。

6.6.3 [10] <6.2，6.3，6.4> 图 6-6 中展示的快闪式存储器读和写的访问时间随着快闪式存储器容量的增大而增加。根据计算，是这样的吗？是什么因素导致这样的结果？

习题 6.7

I/O 的执行可以是同步的也可以是异步的，关于下面的外设，研究这两种方式的不同，回答相应的问题。

a.	鼠标
b.	内存控制器

6.7.1 [5] <6.5> 对于表中的每个外设来说，CPU 和外设之间通信使用哪种总线最合适，是异步的总线，

还是同步的总线?

6.7.2 [5] <6.5> 对于表中的每个外设来说，CPU 和外设之间采用长的、同步总线连接将会引起什么问题?

6.7.3 [5] <6.5> 对于表中的每个外设来说，CPU 和外设之间采用异步总线连接将会引起什么问题?

习题 6.8

如今最常使用的总线类型是：FireWire（IEEE1394）、USB、PCI 以及 SATA。尽管这四种总线都是异步的，它们根据各自的特性以不同的方式实现。根据下面的外设，研究这些不同总线的结构，回答问题。

a.	外部硬件驱动器（external hard drive）
b.	键盘

6.8.1 [5] <6.5> 为表中的外设选择一种合适的总线（FireWire、USB、PCI、或者 SATA），并且解释为什么所选的总线是合适，（见图 6-8，参照每种总线的关键特性）

6.8.2 [20] <6.5> 使用网络或者图书馆的资源，总结出每种类型总线的通信结构。指出总线的控制器是什么? 以及物理的控制在哪里?

6.8.3 [15] <6.5> 总结每种总线的限制，解释为什么必须在使用时考虑总线的这些限制?

习题 6.9

通过联合使用轮询、中断处理、内存映射以及特设的 I/O 指令来实现与 I/O 设备的通信。联合使用这些技术，回答下面关于 I/O 系统通信的问题。

a.	视频游戏控制器
b.	计算机显示器

6.9.1 [5] <6.6> 描述设备的轮询。表中的哪种应用对于通信中的轮询技术是合适的? 为什么?

6.9.2 [5] <6.6> 描述一下中断驱动通信。对于表中每一种应用，如果轮询是不合适的，解释中断驱动的技术为什么是合适的。

6.9.3 [10] <6.6> 对于表中每一种应用，概述内存映射的通信。指出内存保留区域以及概述它们的内容。

6.9.4 [10] <6.6> 对于表中每一种应用，概述一个实现指令驱动通信的设计中的那些指令。指出这些指令以及指令和设备之间的交互。

6.9.5 [5] <6.6> 对于 I/O 子系统来说，使用内存映射以及指令驱动通信是不是合适的? 请解释原因。

习题 6.10

第 6.6 节定义了一个 8 阶段的中断处理步骤。中断原因寄存器以及中断状态寄存器一起使用为中断产生的原因以及为中断处理系统提供了信息。通过回答以下关于中断的问题，研究中断处理。

a.	掉电	过热	以太网控制器数据
b.	过热	重启	鼠标控制器

6.10.1 [5] <6.6> 当中断被检测出来时，状态寄存器被保存，除了优先级最高的中断，其他的中断都被禁止了。为什么低优先级的中断被禁止? 为什么状态寄存器在禁止中断之前被保存?

6.10.2 [10] <6.6> 对表中每一行设备的中断按照优先级排序。

6.10.3 [10] <6.6> 对于表中每种设备出现的中断，概述该如何处理?

6.10.4 [5] <6.6> 当一个中断被处理时，原因寄存器的中断允许位没有被设置将会发生什么? 为了得到相同效果，中断掩码该是什么值?

6.10.5 [5] <6.6> 很多中断处理系统在操作系统中得以实现。为了使得中断处理更有效，应该增加什么硬件支持？将你对潜在的硬件支持的解决方案与功能调用（软件实现）相比较。

6.10.6 [5] <6.6> 在一些中断处理的实现中，一个中断引起一个到中断矢量的立即跳转。不是通过中断原因寄存器为每个中断设置相应的位，而是每个中断具有自己的中断向量。可以通过相同的途径在系统中实现相同优先级的中断系统吗？这种方法是否具备某种优势？

习题 6.11

直接存储器访问（DMA）允许设备直接访问存储器而不是通过 CPU 来访问。这样可以极大地加速外围设备的性能，但是也增加了内存系统设计的复杂度。关于以下外围设备，回答问题，研究 DMA 的实现。

a.	图形卡
b.	声卡

6.11.1 [5] <6.6> 当 DMA 活动时，CPU 是否释放对内存的控制权？例如，一个外围设备可以直接和内存系统通信，从而完全避免 CPU 吗？

6.11.2 [10] <6.6> 上表中的外设中，哪种设备能够利用 DMA？如果 DMA 是合适的，设备的什么特性（准则）决定了使用 DMA 是合适的？

6.11.3 [10] <6.6> 上表中的外设中，哪个设备将会导致和 cache 内容相关的一致性问题的出现？如果一致性问题必须被解决，由什么特性（准则）决定？

6.11.4 [5] <6.6> 将 DMA 和虚拟内存混合起来，描述一下会发生什么问题。表中的哪些设备将会引发问题？这些问题该如何避免？

习题 6.12

根据应用的不同，I/O 的性能尺度就有很大的不同。在一些情形下，所处理的事务主导了性能，数据吞吐量主导了另外一些应用的性能。根据下面的应用，通过回答下面的问题研究 I/O 性能的评价。

a.	网络搜索
b.	声音编辑

6.12.1 [10] <6.7> 对于表中的每个应用，I/O 性能是否主导了系统的性能？

6.12.2 [10] <6.7> 对于表中的每个应用，原始数据吞吐量是不是最好的 I/O 性能测量尺度？

6.12.3 [5] <6.7> 对于表中的每个应用，使用被处理的事务的数目是不是最好的 I/O 性能测量尺度？

6.12.4 [5] <6.7> 前两个问题中的性能尺度与是使用轮询还是中断驱动的通信之间是否有关系？与选择采用内存映射或指令驱动的 I/O 是不是也有关系？

习题 6.13

基准程序扮演着评价和选择外设的重要角色。为了使得基准程序有用，基准程序需要呈现出和一个正常使用着的设备一样的特性。关于下面的应用，回答问题，研究基准程序以及设备的选择。

a.	网络搜索
b.	声音编辑

6.13.1 [5] <6.7> 对表中的每个应用，当评估 I/O 子系统时，定义一组基准程序的特性？

6.13.2 [15] <6.7> 使用在线或者图书馆资源，为表中的应用指定一组标准的基准程序。为什么基准程序是有用的？

6.13.3 [5] <6.7> 在一个大系统的外面评价其中的 I/O 子系统是否有意义？如何评价 CPU？

习题 6.14

在存储系统中，RAID 是实现并行性以及冗余性最受欢迎的途径。RAID 的名字暗示了关于它的几件事情。结合下面的活动来研究 RAID 的这些方面。

a.	在线数据库服务
b.	声音编辑

6.14.1 [10] <6.9> RAID 0 使用跳步（striping）在多个磁盘间实现并行访问。为什么跳步可以提高磁盘的性能？对于表中的每个活动，跳步将更好地帮助实现它们的目标吗？

6.14.2 [5] <6.9> RAID 1 在多个磁盘上镜像数据。假设便宜的磁盘具有比贵磁盘低的 MTBF，那么使用冗余廉价磁盘如何在系统中取得低的 MTBF？使用 MTBF 的数学定义来解释你的回答。对于表中的每项活动，RAID 1 是否可以更好地帮助它们实现目标？

6.14.3 [5] <6.9> RAID1、RAID 3 提供更高的数据可用性（availability），解释 RAID 1 和 RAID 3 之间的折中。表中的每个应用是不是从 RAID 3 得到的益处都比从 RAID 1 得到的更多一些？

习题 6.15

RAID 3、RAID 4 和 RAID 5 都使用了奇偶校验系统来保护数据块。具体地讲，将一个奇偶块和一些数据块关联起来。下面表中每一行是数据块和奇偶块的值，正如图 6-13 中所描述的。

	新 D0	D0	D1	D2	D3	P
a.	FEFE	00FF	A387	F345	FF00	4582
b.	AB9C	F457	0098	00FF	2FFF	A387

6.15.1 [10] <6.9> 为 RAID 3 计算新的奇偶校验值 P′。

6.15.2 [10] <6.9> 为 RAID 4 计算新的奇偶校验值 P′。

6.15.3 [5] <6.9> 是 RAID 3，还是 RAID 4 效率更高？是否有什么原因使得 RAID 3 比 RAID 4 更好一些？

6.15.4 [5] <6.9> RAID 4 和 RAID 5 基本使用相同的机制来计算和存储数据块的奇偶校验和。对于什么应用来说，RAID 5 效率更高些，而且 RAID 5 和 RAID 4 之间的不同如何表现？

6.15.5 [5] <6.9> 根据 RAID 3，随着被保护的数据块尺寸的增大，RAID 4 和 RAID 5 的速度随之增加，为什么是这样的？是否在一种情形下 RAID 4 以及 RAID 5 没有 RAID 3 有效？

习题 6.16

因为金融、在线存储以及通信应用而产生的网路服务器使得磁盘服务器成为关键应用。可用性以及速度成为磁盘服务器众所周知的属性，但是功耗变得越来越重要。使用下面的参数，回答关于配置和评估磁盘服务器的问题。

	应用程序的指令中 I/O 操作的数目	OS 指令中 I/O 操作的数目	负载（KB）	处理器速度（每秒的指令数）
a.	250 000 个	50 000 个	128	40 亿
b.	100 000 个	50 000 个	64	40 亿

6.16.1 [10] <6.8，6.10> 为随机读和写找出可以得到的最大 I/O 速率。忽略磁盘的冲突，而且假设 RAID 控制器不是一个瓶颈。使用和 6.10 节中相同的方法，并且所做的假设相同。

6.16.2 [10] <6.8，6.10> 假设我们配置一个和 6.10 节所描述的一样的 Sun Fire x4150 服务器。确定如果配置 8 个磁盘，I/O 会不会成为瓶颈？当把配置设置为 16、4 以及 2 个磁盘时，情况会如何？

6.16.3 [10] <6.8，6.10>确定如果 PCI 总线、DIMM，或者前端总线（front side bus）成为 I/O 瓶颈，使用和 6.10 节中相同的参数以及相同的假设计算。

6.16.4 [5] <6.8，6.10>解释，为什么实际系统倾向于使用基准程序或者实际程序来评估真实的性能？

习题 6.17

使用一个相对完整的数据来确定一个服务器的性能是个比较简单的任务。然而，当使用不同的供应商提供的不同数据来比较不同的服务器时，在其中选出需要的服务器却是困难的。研究如何评价服务器，根据下面的应用来回答问题。

Web 服务器

6.17.1 [15] <6.8，6.10>对于上表中的应用，指出一个操作系统的运行时特性。选择出支持和习题 6.16 中类似的评价性能的特性。

6.17.2 [15] <6.8，6.10>对于上表中的应用，找到在市场上可用的服务器，这个服务器你认为对于运行这样的服务是合适的。在评价这个服务器之前，指出为什么选择它。

6.17.3 [20] <6.8，6.10>使用第 6 章和习题 6.16 中使用过的类似的尺度（测量），将习题 6.17.2 中的服务器和习题 6.16 中被评价的 Sun Fire x4150 做比较。你将会选择哪个？是不是你的分析结果使你很惊讶？具体地说，你是否得到了与设想不同的选择？

6.17.4 [15] <6.8，6.10>指出一个标准基准程序组，它将对于比较习题 6.17.2 中的服务器和 Sun Fire x4150 服务器是有用的。

习题 6.18

必须小心地解释供应商所提供的存储器的度量手段和统计数据，以便得到关于系统行为的有意义的预测。下表为不同的磁盘驱动提供了一些数据。

	驱动的数目	小时/驱动	小时/故障
a.	1000	8760	1 000 000
b.	1000	10 512	1 500 000

6.18.1 [10] <6.12>为表中的每个磁盘计算年故障率（AFR）。

6.18.2 [10] <6.12>假设上表中的 AFR 在磁盘的生命周期中是变化的。具体地说，假设 AFR 在使用后第一个月是 3 倍，而且第 5 年开始，每一年变成 2 倍。计算 7 年之后需要更换多少个磁盘？10 年之后又如何？

6.18.3 [10] <6.12>假设磁盘越昂贵，故障率就越低。具体地说，比较贵的磁盘的可用性将在 8 年而不是 5 年后故障率开始加倍。如果你的数据想要在磁盘中保持 7 年，你将需要消费多少个磁盘？如果数据需要保存 10 年又如何？

习题 6.19

对于习题 6.18 中的磁盘，假设你的供应商提供了 RAID 0 配置，整个配置将会使得存储系统增加 70% 的吞吐量，而且 RAID 1 配置将会使磁盘的 AFR 下降 2 倍。假设每个方案的造价将是原始造价的 1.6 倍。

6.19.1 [5] <6.9，6.12>假设原始系统的参数不变，你是否会推荐升级到 RAID 0 或者 RAID 1，假设单个磁盘参数和习题 6.18 中的表保持一致？

6.19.2 [5] <6.9，6.12>假设你的公司使用了大的磁盘场来运作一个全球搜索引擎，依靠一些广告，升级到 RAID 0 或者 RAID 1 是否使你的收入模型变得经济和有意义？

6.19.3 [5] <6.9，6.12>重做习题 6.19.2，通过运作大的磁盘场实现一个在线的后备公司。基于你的服务器的可靠性，升级到 RAID 0 或者 RAID 1 是否使你的收入模型变得经济和有意义？

习题 6.20

计算机系统每天的评价和维护工作包含了很多在第 6 章中讨论过的概念。通过回答下面的问题，研究评价系统的本质特性。

6.20.1 [20] <6.10，6.12> 配置 Sun Fire x4150，使得它能为具有 1 000 个处理器的处理器阵列，提供 10T 字节的存储容量，以便运行生物信息仿真的服务器。你的配置应当最小化功耗，同时解决磁盘的吞吐量以及可用性问题。当你执行你的配置时，确保考虑到最大仿真的属性。

6.20.2 [20] <6.10，6.12> 为习题 6.20.1 的磁盘阵列推荐一个后备以及数据历史系统。对比磁盘、磁带和在线后备容量。使用网络以及图书馆资源指出潜在的服务器。使用第 6 章描述的参数描述评估成本和应用程序的维护性。对于指定的需求，比较应用的特性，选择参数。

6.20.3 [15] <6.10，6.12> 假设在习题 6.20.2 中所选择的系统竞争供应商允许你在网上评价他们的系统。指出你为了选择出最好的系统所使采用的基准程序。指出为了做决定而搜集数据，你需要的时间。

小测验答案

6.2 节　B 和 C 正确。

6.3 节　B 和 C 正确。

6.4 节　全部正确（假设 40 MB/s 与 100 MB/s 是可比的）。

6.5 节　A 正确。

6.6 节　A 和 B 都正确。

6.7 节　A、B 和 C 均错误，由于大多数 TPC 基准都包含开销。

6.9 节　全部正确。

第7章

Computer Organization and Design: The Hardware/Software Interface, 4E

多核、多处理器和集群

在大海里有比我们曾经捕获的更多更好的鱼。

——爱尔兰谚语



多处理器或集群的结构

“在月球的山脉上，沿着阴影笼罩的山谷，前进，勇敢地前进!”阴影回应道，——“如果你在寻找理想国!”

——埃德加·爱伦·坡，《理想国》，第4节，1849

7.1 引言

计算机架构师一直在寻求计算机设计的理想境地：只需将现有的多个较小计算机简单地连接在一起，即可构成功能强大的计算机。这就是**多处理器**[㊀]产生的根源。在理想情况下，用户可以按照其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。通过将大型低效能处理器替换为许多小型高效能处理器，在软件可以有效地使用每个处理器的情况下，可在单位瓦特或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。

由于多处理器软件支持可变数量的处理器，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6章）。

高性能意味着处理独立作业的高吞吐量，这被称作**作业级并行或进程级并行**[㊁]。并行作业是互相独立的应用程序，这在并行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**[㊂]来表示同时运行在多个处理器上的单一

㊀ 多处理器（multiprocessor）：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。

㊁ 作业级并行（job-level parallelism）或进程级并行（process-level parallelism）：通过同时运行独立程序的方法来利用多处理器。

㊂ 并行处理程序（parallel processing program）：同时运行在多个处理器上的单一程序。

程序。

在过去的数十年里，很多科学问题都需要更快的计算机，同时这些问题也被用于评价新型的并行计算机。本章将介绍其中的几个问题。这些问题有些处理起来很简单，使用由封装在不同独立服务器或 PC 上的多个微处理器组成的**集群**[⊖]即可完成计算。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。

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，而是借助于在单芯片内集成更多的处理器。为了避免名称上的冗长，称之为**多核微处理器**[⊖]而不是多处理器微处理器（multicore microprocessor）。处理器在多核芯片内一般称为核（core）。核的数量预计每两年翻一番。所以，注重性能的程序员必须成为并行程序员，因为顺序程序等同于慢速程序。

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呈几何级数缩放。

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，因而会有很多关于术语及其内涵的困惑。图 7-1 试图澄清串行（serial）、并行（parallel）、顺序（sequential）和并发（concurrent）等术语之间的差异。该图中的列代表固有顺序或并发的软件，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括词法分析、句法分析、代码生成和优化等是顺序完成的。与之相比，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	运行于 Intel Pentium 4 上的用 Matlab 编写的矩阵乘	运行于 Intel Pentium 4 上的 Windows Vista 操作系统
	并行	运行于 Intel Xeon e5345（Clovertown）上的用 Matlab 编写的矩阵乘	运行于 Intel Xeon e5345（Clovertown）上的 Windows Vista 操作系统

图 7-1　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 7-1 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 Xeon e5345 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 Xeon e5345 上运行。也许你会认为并行的唯一挑战是如何将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。

7.2 节分析了创建高效的并行处理程序的困难所在。7.3 节和 7.4 节描述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）。7.5 节描述了术语*多线程*（multithreading），它经常和多处理（multiprocessing）产生混淆，部分是因为它们都依赖

⊖ 集群（cluster）：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。

⊖ 多核微处理器（multicore microprocessor）：在单一集成电路上包含多个处理器（“核”）的微处理器。

于程序的并发性。7.6 节描述了一种比图 7-1 更古老的分类方法，以及支持顺序应用程序在并行硬件上运行的两种指令集类型：SIMD 和向量机。7.7 节描述了一种来自图形硬件处理领域的相对较新的计算机，称为 GPU（graphics processing unit，图形处理单元）。附录 A 给出了 GPU 的具体细节。7.9 节首先描述了寻找并行基准测试程序的难点，然后描述了一种新的简单但却深刻的性能模型，可用于辅助应用程序及体系结构的设计。利用该性能模型，我们在 7.11 节评估了 4 种最近的多核计算机在两种应用程序内核上的性能。本章最后解析了一些常见谬误和陷阱，并进行了总结。

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：结合律
- 第 4 章 4.10 节：并行和高级指令级并行
- 第 5 章 5.8 节：并行与存储器层次结构：cache 一致性
- 第 6 章 6.9 节：并行性与 I/O：廉价磁盘冗余阵列

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。

7.2　创建并行处理程序的难点

并行的难点不在于硬件，目前只有极少量重要的应用程序经过重新编写能在多处理器上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是困难的，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以开发呢？

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和效率；否则，应该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，都要面临如下挑战：调度、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，使用更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。

举例　**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：

$$\text{改进后的执行时间} = \frac{\text{受改进影响的执行时间}}{\text{改进量}} + \text{未受改进影响的时间}$$

使用加速比的形式重新表示 Amdahl 定律：

$$\text{加速比} = \frac{\text{改时前的执行时间}}{(\text{改进前的执行时间} - \text{受影响的执行时间}) + \dfrac{\text{受影响的执行时间}}{100}}$$

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：

$$\text{加速比} = \frac{1}{(1 - \text{受影响的执行时间比例}) + \dfrac{\text{受影响的执行时间比例}}{100}}$$

将加速比 90 的目标替换到上述公式中：

$$90 = \frac{1}{(1 - \text{受影响的执行时间比例}) + \dfrac{\text{受影响的执行时间比例}}{100}}$$

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。

举例　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，一个加法是一对 10×10 二维矩阵的求和。使用 10 个和 100 个处理器达到的加速比分别是多少呢？如果矩阵维数是 100×100 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

$$\text{改进后的执行时间} = \frac{100t}{10} + 10t = 20t$$

所以使用 10 个处理器的加速比是 $110t/20t = 5.5$。使用 100 个处理器的执行时间是

$$\text{改进后的执行时间} = \frac{100t}{100} + 10t = 11t$$

所以使用 100 个处理器的加速比是 $110t/11t = 10$。

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 100 个处理器仅达到了潜在加速比的 10%。如果矩阵增加到 100×100，那么顺序程序的执行时间为 $10t + 10\,000t = 10\,010t$。10 个处理器的执行时间是

$$\text{改进后的执行时间} = \frac{10\,000t}{10} + 10t = 1\,010t$$

所以 10 个处理器的加速比是 $10\,010t/1\,010t = 9.9$。100 个处理器的执行时间是

$$\text{改进后的执行时间} = \frac{10\,000t}{100} + 10t = 110t$$

所以 100 个处理器的加速比是 $10\,010t/110t = 91$。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 99% 的潜在加速比，使用 100 个处理器获得了超过 90% 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**[⊖]指保持问题规模固定所测得的加速比。**弱比例缩放**[⊖]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。

取决于不同的应用程序，可以使用不同的比例缩放方法。例如，第6章的TPC-C借贷数据库基准测试程序需要按比例增加客户数量，从而提高每分钟内的事务处理次数。这是因为不能由于银行安装了更快的计算机而假定客户每天使用100次ATM，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处理次数提高100倍，应当在顾客数量提高100倍的情况下进行实验。

最后一个例子说明了负载均衡的重要性。

举例　**加速比的挑战：负载均衡**

在上个例子中，我们使用100个处理器在较大问题规模中实现了加速比91，其中假定了负载是完全均衡的。也就是说，100个处理器中每一个都完成1%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2%和5%工作量时的加速比。

答案

如果一个处理器负责2%的并行负载，那么它需要完成2%乘以10000即200次加法，其他的99个处理器分担剩余的9800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。

$$\text{改进后的执行时间} = \text{Max}\left(\frac{9800t}{99}, \frac{200t}{1}\right) + 10t = 210t$$

加速比降低至 $10\,010t/210t = 48$。如果一个处理器完成5%的负载，它必须执行500次加法：

$$\text{改进后的执行进间} = \text{Max}\left(\frac{9500t}{99}, \frac{500t}{1}\right) + 10t = 510t$$

加速比进一步降低至 $10\,010t/510t = 20$。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，加速比几乎降低了一半；一个处理器的负载是其他处理器的五倍时，加速比几乎降低到了原来的五分之一。

小测验

是非判断：强比例缩放不遵守Amdahl定律。

7.3　共享存储多处理器

在给出了重写原有程序运行于并行硬件上的难点之后，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，以便程序不必考虑它们在哪里运行，只要知道它们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间都是可见的。另一种方法是每个处理器采用独立的地址空间，共享必须是显式的；我们将在下节描述这种情况。当物理地址空间公用时（这一般发生在多核芯片中），通常由硬件提供cache一致性，以便保证共享存储器的一致性（参见第5章5.8节）。

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**共享存储多处理器**[⊜]

⊖ 强比例缩放（strong scaling）：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（weak scaling）：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。

⊜ 共享存储多处理器（shared memory multiprocessor，SMP）：具有单一地址空间的并行处理器，存取时采用隐式通信的方式。

(SMP)，尽管更加准确的术语应该是共享地址多处理器（shared-address multiprocessor）。需要注意，此类系统虽然全部共享同一物理地址空间，但是依然可以在它们自己的虚地址空间中运行独立的作业。处理器通过存储器中的共享变量互相通信，所有处理器都能访问任何存储器位置。图 7-2 给出了 SMP 的典型组成。

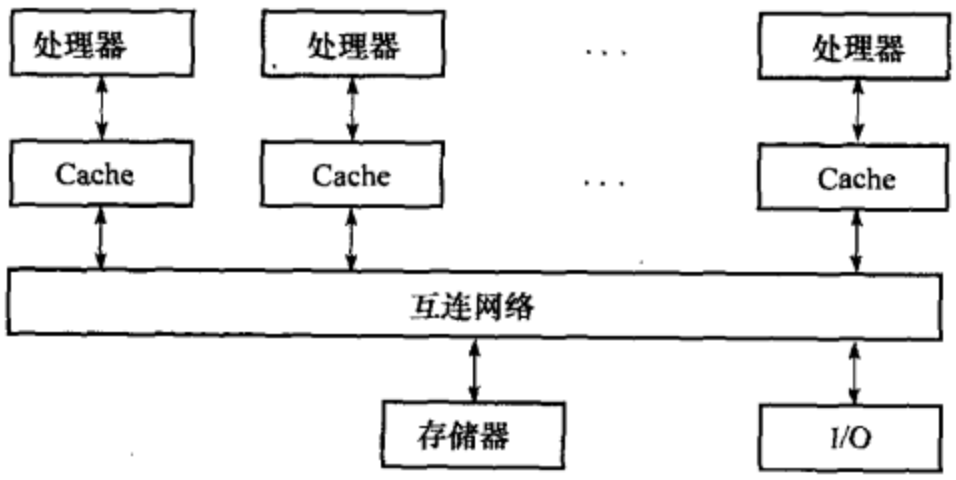


图 7-2　一个共享存储多处理器的典型组成

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大致相同，无论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**（UMA）[⊖]多处理器。在另一种类型中，某些存储访问更快，这取决于哪个处理器请求访问哪个字。这类机器称为**非统一存储访问**（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。

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**[⊜]。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**[⊛]。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。

举例　**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 100 的共享存储多处理器计算机上对 100 000 个数求和，该计算机具有统一的存储器访问时间。

答案

需要完成的第一步依然是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 到 99 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 =i +1)
     sum[Pn] =sum[Pn] +A[i]; /*  对指定的区域求和 * /
```

⊖ 统一存储访问（uniform memory access，UMA）：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（nonuniform memory access，NUMA）：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。

⊝ 同步（synchronization）：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（lock）：一个时刻仅允许一个处理器访问数据的同步装置。

下一步是将这些部分和加起来，称为**约简**[㊀]。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 7-3 对约简的过程进行了说明。

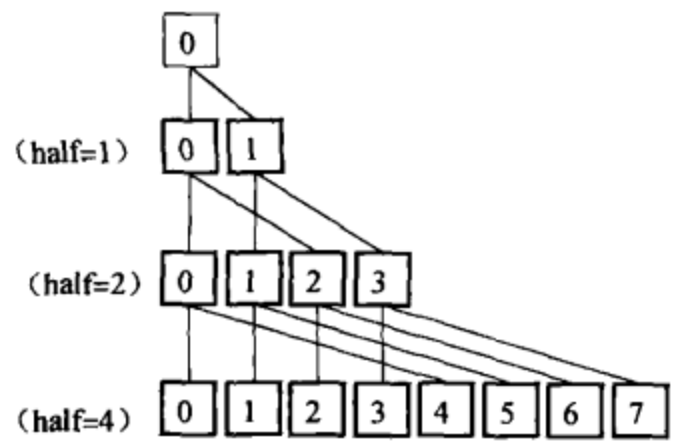


图 7-3　自底向上的最后 4 级求和过程

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。

在该例子中，“消费者”处理器在读取由“生产者”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 = 100; /* 在该多处理器中有 100 个处理器 */
repeat
      synch(); /* 等待部分和的计算完成 */
      if (half% 2! =0 && Pn = =0)
          sum[0] = sum[0] + sum[half-1];
          /* 当 half 为奇数时需要进行此步求和* /
      half = half/2; /* 完成求和工作处理器的分界线* /
      if (Pn < half)
          sum[Pn] = sum[Pn] + sum[Pn + half];
until (half = =1);
```

小测验

是非判断：共享存储多处理器不能利用作业级并行。

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，但为了向程序员提供一个实用的共享存储器抽象，它的开销过大。

7.4　集群和其他消息传递多处理器

相对于共享地址空间，另一种方法是每个处理器具有自己私有的物理地址空间。图 7-4 给出了具有多个私有地址空间的典型组成。这种多处理器必须通过显式的这**消息传递**[㊁]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**[㊂]和**接收消息例程**[㊃]，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。

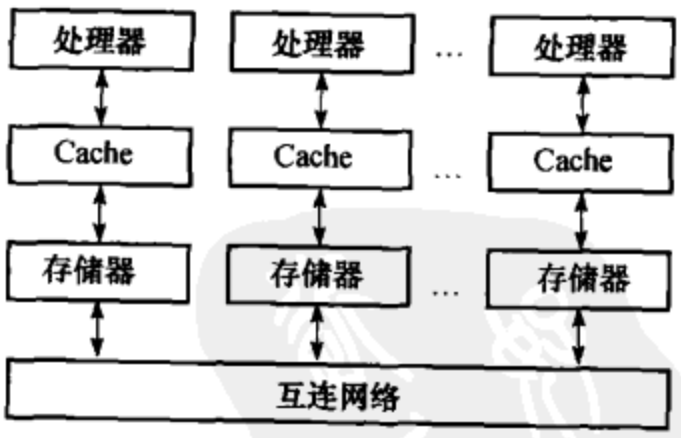


图 7-4　具有多个私有地址空间的多处理器的组成，传统上称为消息传递多处理器

与图 7-2 中的 SMP 不同，互连网络不是在 cache 和存储器之间，而是在处理器 - 存储器的节点之间。

无论采用共享地址还是消息传递，一些并发的应用程序在并行硬件上都可以运行得很好。特别是对一些作业级并行和几乎不需通信的应用程序（如 Web 搜索、邮件服务器和文件服务器），即使不需要共享地址也可以运行得很好。

㊀ 约简（reduction）：处理一个数据结构并返回单一值的函数。
㊁ 消息传递（message passing）：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。
㊂ 发送消息例程（send message routine）：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。
㊃ 接收消息例程（receive message routine）：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。

曾经有过几次基于高性能消息传递网络构建高性能计算机的尝试。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，但成本过高。很少有应用程序能够为更高的性能支付更多的成本。因此，**集群**[⊖]已经成为目前使用最为广泛的消息传递并行计算机。集群通常是采用标准网络交换机和线缆互连的一组商用计算机。每台计算机运行操作系统的一个不同备份。目前，几乎所有 Internet 服务都依赖于由商用服务器和交换机构成的集群。

集群的一个缺点是管理由 n 台机器构成的集群的成本几乎与管理 n 台独立机器的成本相同，而管理一个含有 n 个处理器的共享存储多处理器的成本几乎与管理一台机器的相同。

这一缺点是促使虚拟机（见第 5 章）日益普及的原因之一，因为虚拟机相对集群更加容易管理。例如，虚拟机支持程序的自动关闭或启动，这简化了软件的升级。虚拟机甚至不需要停止程序就可将其从集群中的一台计算机迁移到另一台上，这使得程序可以从失效的硬件上迁移。

集群的另一个缺点是集群中的处理器通常借助每台计算机的 I/O 进行互连，而多处理器中的核通常借助计算机的存储器进行互连。存储器互连具有更高的带宽和更低的延迟，因而能够提供更高的通信能力。

集群的最后一个缺点是存储器划分的开销：n 台机器构成的集群具有 n 个独立的存储器和 n 份操作系统备份，但是共享存储多处理器允许一个程序使用计算机中几乎全部存储器，并且只需要操作系统的一份备份。

举例　存储器效率

假定某个共享存储多处理器有 20 GB 主存，而另一个由 5 台计算机构成的集群中每台计算机有 4 GB 主存。如果一个操作系统占 1 GB 主存，请问共享存储多处理器用户的可用存储容量超过集群用户多少呢？

答案

共享存储多处理器相对集群的程序可用存储器大小比例是

$$\frac{20-1}{5\times(4-1)}=\frac{19}{15}\approx 1.25$$

因此共享存储多处理器可提供超过集群 25% 的更多主存空间。

下面我们重新完成前面的求和示例，从中可以看出使用多个私有存储器和显式通信所带来的影响。

举例　一个采用消息传递的简单并行处理程序

假设我们需要在一个消息传递多处理器上完成对 100 000 个数的求和，处理器数量为 100，每个处理器都有自己的私有存储器。

由于该计算机具有多个地址空间，因而第一步是将 100 个子集分布到 100 个处理器的局部存储器上。拥有 100 000 个数的处理器负责把子集发送给 100 个处理器 - 存储器节点。

下一步是获得每个子集的和。这一步只需在每个处理器上执行下面的一个循环即可：从局部存储器上读一个字，并将其加到一个局部变量上：

```
sum = 0;
for (i=0; i<1000; i=i+1) /* 在每个数组上循环* /
    sum=sum+AN[i]; /* 对局部数组求和* /
```

最后一步是对 100 个部分和求和。难点在于每个部分和都在不同的处理器上。因此，我们必须使用互连网络发送部分和以便完成最终计算。如果将所有部分和都发送到同一处理器上，这会导致部分和的相加顺序完成，所以我们仍旧采用分而治之的方法。

⊖ 集群（cluster）：一组计算机通过 I/O 接口与标准网络交换机连接而形成的消息传递多处理机。

首先，一半处理器将自己的部分和发送给另外一半处理器，并由另外一半处理器完成两个部分和的相加。然后四分之一的处理器（一半的一半）发送新的部分和到另外四分之一的处理器（另外一半的一半），以完成下一轮的求和。对分、发送和接收将一直持续直到获得最后的结果。令 Pn 表示处理器的编号，send(x, y)是将值 y 通过互连网络发送给处理器编号为 x 的例程，receive()是从互连网络接收数据的函数。下面是相应的代码：

```
limit=100; half=100; /* 100个处理器 * /
repeat
     half=(half+1)/2; /* 发送和接收的分界线* /
     if (Pn >=half && Pn <limit) send(Pn-half, sum);
     if (Pn <(limit/2)) sum=sum+receive();
     limit=half; /* 发送者的上界* /
until (half==1) /* exit with final sum* /
```

这段代码将所有处理器分成发送者或接收者，并且每个接收处理器只收到一个消息，我们可以假定接收处理器在收到消息之前一直阻塞。因此，发送和接收除了用于通信之外，还可以用作同步原语，因为处理器需要等待传输的数据。

如果节点的数量为奇数，那么中间的节点不参加发送和接收。界限的设定将使其成为下轮计算的最高节点。

精解：这个例子假定消息传递速度和加法一样快。实际上，消息发送和接收是非常慢的。让少数节点从其他处理器接收多个部分和是另一个能够更好地平衡计算与通信的优化方案。

硬件/软件接口

对硬件设计者来说，基于消息传递的计算机比需要维持缓存一致性的共享存储计算机更加容易设计（见第 5 章 5.8 节）。对程序员来说，消息传递的优点是显式通信，这意味着与共享存储的隐式通信相比性能提升较少；缺点是难以将一个顺序程序移植到消息传递计算机中，因为每次通信必须提前标识出来，否则程序将无法工作。缓存一致的共享存储允许硬件判断哪些数据需要通信，这使得移植相对简单。现在对于如何最快地获得高性能有不同的观点，隐式通信也有大量的支持者和反对者。

尽管集群的内存互相独立是一个缺点，但从另一方面讲，内存独立提高了系统的可用性。由于集群是由互相独立的计算机通过局域网互连构成的，因此它相对 SMP 来说，不需要关闭集群系统即可替换其中的一台机器。从根本上说，共享地址结构意味着在操作系统不能提供特别辅助时，很难隔离一个处理器并将其进行替换。由于集群软件是运行在每台计算机局部操作系统之上的一层，因而断开并替换一台失效的机器要简单得多。

如果集群是由整个计算机和规模可变的独立网络构成的，隔离使得扩展系统更加容易，不需要关闭运行在集群之上的应用程序即可进行扩展。

低成本、高可用性、高功效以及快速、增量式的扩展性使得集群对互联网服务提供商具有很大吸引力。数百万用户每天使用的搜索引擎即依赖这项技术。eBay、Google、Microsoft、Yahoo 和其他公司都拥有许多由集群构成的数据中心，每个集群包含数以万计的处理器。显然，多处理器在 Internet 服务商中的应用已经获得了巨大的成功。

精解：大规模计算的另一种形式为网格计算，它的计算机分布在更广泛的区域，运行在多台计算机之间的程序通过广域网通信。网格计算最流行的独特形式是 SETI@ home 项目率先提出的。研究人员发现，在任何时候都有数以百万的 PC 处于空闲状态没有工作可做，如果有人能在这些计算机上开发相应软件，并将整个问题独立的一部分交由每台计算机来完成，那么这些计算机的计算能力就得到了充分利用。第一个这样的应用是 SETI（Search for ExtraTerrestrial Intelligence）。超过 200 多个国家的 500 多万计算机用户参与了 SETI@ home，并共同贡献了超过 190 亿小时计算处理时间。截至 2006 年年底，SETI@ home 网格达到了

257 TeraFLOPS。

小测验

1）是非判断：和 SMP 类似，消息传递计算机依赖锁机制实现同步。

2）是非判断：与 SMP 不同的是，消息传递计算机需要并行处理程序和操作系统的多份备份。

7.5　硬件多线程

硬件多线程[㊀]支持多个线程以重叠方式共享处理器的功能单元。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。特别需要指出的是，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。

硬件多线程主要有两种实现方法。**细粒度多线程**[㊁]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延迟执行。

粗粒度多线程[㊂]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，如二级缓存缺失。这种改变降低了线程切换的开销，并且几乎不会降低单个线程的执行速度，因为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，流水线重新填充时间和阻塞时间相比是可以忽略的。

同时多线程(SMT)[㊃]是硬件多线程的一个变种，它使用多发射动态调度处理器的资源来挖掘线程级并行，并同时保持指令级并行。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，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。

既然 SMT 是依赖于现有的动态机制，SMT 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 7-5 说明了开发超标量资源方式不同时处理器能力的差别。上面的部分表示四个线程如何在不支持多线程的超标量处理器上独立运行。下面的部分表示四个线程如何以三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

㊀ 硬件多线程（hardware multithreading）：在线程阻塞时处理器可切换到另一线程的实现。

㊁ 细粒度多线程（find-grained multithreading）：硬件多线程的一种形式，其建议每条指令执行之后都进行线程切换。

㊂ 粗粒度多线程（coarse-grained multithreading）：硬件多线程的一种形式，其建议仅在一些重要事件（如缓存缺失）之后进行线程切换。

㊃ 同时多线程（simultaneous multithreading，SMT）：多线程的一种形式，其利用多发射、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。

- 支持同时多线程的超标量

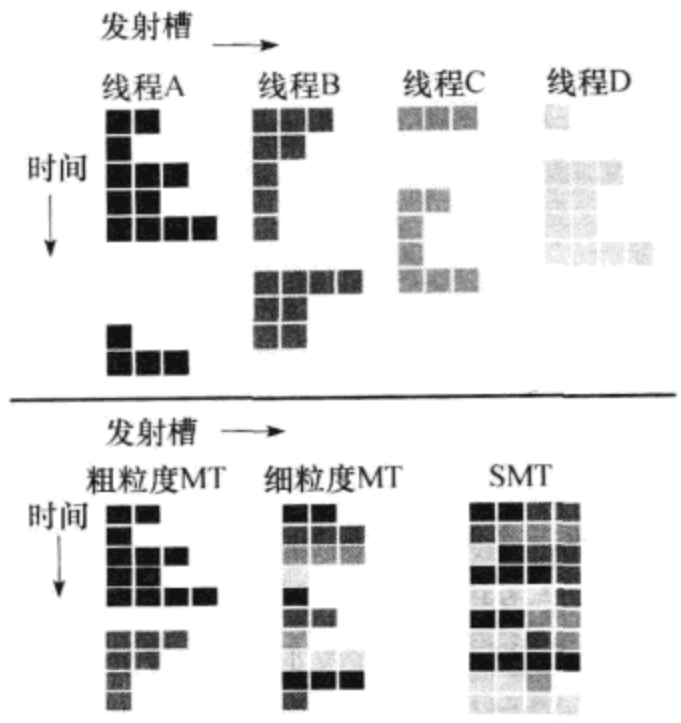


图 7-5　四个线程如何以不同方式利用超标量处理器中的发射槽

上面的四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四个不同线程。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示，但其会导致更多的吞吐量损失。

在不支持硬件多线程的超标量处理器中，缺乏指令级并行时发射槽的利用受到限制。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 7-5 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。例如，最近的 Intel Nehalem 多核处理器支持两个线程的 SMT，目的是提高核的利用率。

最后总结以下三点：首先，从第 1 章我们知道，功耗墙驱使处理器芯片设计朝着简单并且有效功耗的方向发展。在乱序处理器中可能需要精简未被充分利用的资源，并且应当使用多线程的简单形式。例如，7.11 节中的 Sun UltraSPARC T2（Niagara 2）微处理器是一个回归简约的微体系结构，因此其使用了细粒度多线程。

第二，容忍缓存缺失所引起的延迟是提高性能的关键。细粒度计算机（如 UltraSPARC T2）在缓存缺失时切换到另一线程，相对于 SMT 试图填充未使用发射槽的方法，在隐藏存储器延迟方面可能更加有效。

第三，硬件多线程的目标是通过在不同任务之间共享资源，从而更加有效地使用硬件。多核设计也是共享资源。例如，两个处理器可能共享一个浮点单元或一个三级缓存。相对于使用更多非多线程的核，这类共享会减少多线程所带来的好处。

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：同时多线程使用线程提高动态调度的乱序处理器的资源使用率。

7.6　SISD、MIMD、SIMD、SPMD 和向量机

20 世纪 60 年代提出了并行硬件的另一种分类方法，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 7-6 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[㊀]和 MIMD[㊁]

		数据流	
		一个	多个
指令流	一个	SISD：Intel Pentium 4	SIMD：x86 的 SSE 指令
	多个	MISD：目前没有实例	MIMD：Intel Xeon e5345（Clovertown）

图 7-6　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，并使用条件语句控制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**（SPMD）[㊂]，它是 MIMD 计算机编程的正常方式。

很难找到可以归类为多指令流单数据流（MISD）的有效实例，反过来 SIMD 要有意义得多。SIMD[㊃]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。

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 7-1，一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程序存储器的大小——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 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**[㊄]。SIMD 在使用 case 或 switch 语句时效率最低，因此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将被摒弃，而带有正确数据的执行单元将继续执行。在这种情况下系统将以 $1/n$ 的性能运行，其中 n 为 case 的分支数量。

启发了 SIMD 分类的向量处理器正逐渐成为历史（见 CD 上的 7.14 节），但是直到现在 SIMD 的两种表示依然并存。

7.6.1　在 x86 中的 SIMD：多媒体扩展

SIMD 目前使用最广泛的变种几乎在每个微处理器中都可以找到，并且已经成为 x86 微处理器中数百条 MMX 和 SSE 指令的基础（见第 2 章）。引入这些指令的目的是提高多媒体程序的性能。这些指令使得硬件具有许多并行工作的 ALU，或者说是将一个很宽的 ALU 分布到许多并行

㊀ SISD（Single Instruction stream，Single Data stream）：单指令流单数据流的单处理器。
㊁ MIMD（Multiple Instruction streams，Multiple Data streams）：多指令流多数据流的多处理器。
㊂ SPMD（Single Program，Multiple Data streams）：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。
㊃ SIMD（Single Instruction stream，Multiple Data streams）：单指令流多数据流。同样的指令在多个数据流上操作，和向量处理器或阵列处理器一样。
㊄ 数据级并行（data-level parallelism）：操作独立的数据所获得的并行。

工作的小 ALU 上。例如，一个硬件单元可以是一个 64 位 ALU，也可以是两个 32 位 ALU、四个 16 位 ALU 甚至八个 8 位 ALU。存储器操作位宽与 ALU 是相同的，这样无论是传输一个 64 位数据元或还是传递两个 32 位数据元、四个 16 位数据元或八个 8 位数据元，程序员都可以认为数据传输指令是相同的。

这种对于位宽较小的整数非常低成本的并行是在 x86 中引入 MMX 指令的最初想法。随着摩尔定律的发展，更多硬件已经增加到多媒体扩展中来，现在 SSE2 能够支持一对 64 位浮点数的同时执行。

操作和寄存器的位宽编码到了多媒体指令的操作码中。随着操作和寄存器位宽的变长，多媒体指令的操作码数量也在增加，现在已经有数百条 SSE 指令，可以进行各种有效的组合（见第 2 章）。

7.6.2　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 Cray 公司制造的计算机。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。

举例　**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 的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)      ;读入标量 a
        addiu   r4,  $s0,#512   ;读入的上界
loop:   l.d     $f2,0($0)       ;读入 x(i)
        mul.d   $f2,$f2,$f0     ;a×x(i)
        l.d     $f4,0($s1)      ;读入 y(i)
        add.d   $f4,$f4,$f2     ;a×x(i)+y(i)
        s.d     $f4,0($s1)      ;写回 y(i)
        addiu   $s0,$s0,#8      ;递增 x 的索引
```

```
        addiu       $ s1, $ s1, #8          ;递增 y 的索引
        subu        $ t0, r4, $ s0          ;计算边界
        bne         $ t0, $ zero, loop      ;检查是否完成
```

针对 DAXPY 的向量 MIPS 代码是：

```
l.d                 $ f0,a($ sp)            ;读入标量 a
lv                  $ v1,0($ s0)            ;读入向量 x
mulvs.d             $ v2, $ v1, $ f0        ;向量与标量乘法
lv                  $ v3, 0($ s1)           ;读入向量 y
addv.d              $ v4, $ v2, $ v3        ;将 y 加到乘积上
sv                  $ v4, 0($ s1)           ;写回结果
```

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，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功耗。

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，每次 add.d 必须等待 mul.d，并且每次 s.d 必须等待 add.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，而不是每次向量数据元操作时发生一次。在这个例子中，MIPS中的流水线阻塞频率大约比VMIPS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。

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。后面的处理过程被称作条状开采法（strip mining）。

7.6.3　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：

- 一条向量指令就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功耗降低。
- 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都是地址连续的，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功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。

7.6.4　向量与多媒体扩展的对比

与 x86 SSE 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，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。向量同时支持跨越存取（strided access）和变址存取（indexed access），前者硬件每隔 n 个存储器中的数据元读取一次，后者按照数据项地址读取到向量寄存器中。

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 位数据上操作。

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；相对多媒体扩展，向量机更加容易随时间推移而得到不断改进。

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。

精解：在给出了向量体系结构如此之多的优点之后，考虑为何向量机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。然而，最近来自 Intel 的声明建议向量机将发挥更重要的作用。Intel 的先进向量指令（Advanced Vector Instruction，AVI）将 SSE 寄存器从 128 位扩展到 256 位，并且最终将扩展到 1024 位。1024 位等价于 16 个双精度浮点数的宽度。除此之外，Intel 将在 2010 年进入 GPU 市场——代码为“Larrabee”——据称包含向量指令。

精解：向量和多媒体扩展的另外一个优点是易于扩展一个标量指令集体系结构，使其提高数据并行操作的性能。

7.7　图形处理单元简介

在现有体系结构中增加 SIMD 指令的一个主要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，当摩尔定律增加了微处理器设计中可用晶体管数量时，提高图形处理能力就变得有必要了。

正如摩尔定律使得 CPU 提高了图形处理能力一样，它也使得视频图形控制芯片增加了加速 2D 和 3D 图形处理的功能。而且，由 Silicon Graphics 提供的高端昂贵显卡甚至可以加到工作站中，用来生成摄影级质量的图像。这些高端显卡普遍用于创建计算机生成的图像，后来进入到电视广告和电影领域。因此，随着处理资源的不断增加，视频图形控制器有着明确的应用目标，正如超级计算机拥有丰富的资源一样，微处理器可以通过请求超级计算机获取更高的性能。

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让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。

考虑到图形和游戏社区与微处理器开发社区有着不同的目标，它采用了自己的一套制程和术语。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。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，因为可以让 CPU 在必要的时候完成。所以 CPU-GPU 的组合是异构多处理（heterogeneous multiprocessing）的一个实例，其中并非所有处理器都是相同的。（另外一个实例是 7.11 节的 IBM Cell 体系结构，也是被设计来加速 2D 和 3D 图形。）

- GPU 的编程接口是高层次的应用程序接口（application programming interface，API），如 OpenGL 和 Microsoft 的 DirectX，并与高层次图形绘制语言紧密结合，如 NVIDIA 的 Cg（C for Graphics）和 Microsoft 的 HLSL（High Level Shader Language）。这些语言的编译器的目标代码是业界标准的中间语言，而不是机器指令。GPU 驱动软件会产生针对特定 GPU 优化的机器指令。这些 API 和语言发展迅速，能够包含摩尔定律带来的更多 GPU 新资源，同时 GPU 设计者不必考虑二进制指令的向后兼容性，可以不断开发新的体系结构，而永远无需因实验失败承担责任而担心。这种环境使得 GPU 的创新速度超过了 CPU。
- 图形处理主要包括绘制 3D 几何基元（如线和三角形）顶点、几何基元中像素片元（pixel fragment）的着色或渲染。例如在视频游戏中，需要绘制 20 ~ 30 倍的像素和顶点。
- 每个顶点可以独立绘制，并且每个像素片元可以独立渲染。为了快速渲染一帧中的数百万个像素，GPU 并行执行许多来自顶点和像素渲染程序的线程。
- 图形数据的类型是顶点（由（x，y，z，w）坐标构成）和像素（由（red，green，blue，alpha）颜色成分构成）。（参见附录 A 了解有关顶点和像素的更多内容。）GPU 将每个顶点对象用一个 32 位浮点数表示。四个像素对象中的任何一个最初都用一个 8 位无符号整数表示，但是最近的 GPU 开始用一个 0.0 到 1.0 之间的单精度浮点数表示。
- 工作集可以达到上亿字节，它所显示的时间局部性与主流应用程序中的数据并不相同。而且，在这些任务中存在大量的数据级并行。

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 不依赖像 CPU 一样的多级缓存来隐藏到存储器的长延迟。事实上，GPU 依赖足够的线程数量来隐藏到存储器的延迟。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。
- GPU 借助广泛的并行来获得高性能，在其内部实现了许多并行处理器和并发线程。
- GPU 的主存是面向带宽的而不是面向延迟的。甚至有面向 GPU 的分离 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 2008 年，GPU 一般有不超过 1 GB 的存储器，而 CPU 一般在 2 ~ 32 GB 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 是通过许多线程的联合来获取高存储器带宽，GPU 可以提供许多并行处理器也可以提供许多线程。因此，每个 GPU 处理器都是高度多线程的。
- 在过去，GPU 借助异构专用处理器提供图形应用程序所需的性能。最近的 GPU 正在朝着和通用处理器一样的方向发展，在编程方面提供更多的灵活性，使得 GPU 更像主流计算中的多核设计一样。
- 考虑到图形数据的类型具有四个元素，GPU 在历史上采用像 CPU 一样的 SIMD 指令。然而，最近的 GPU 更加专注于标量指令，以便提高编程性和效率。
- 与 CPU 不同的是，GPU 一直不支持双精度浮点算术，因为在图形应用程序中不需要这类运算。在 2008 年，第一个支持硬件双精度的 GPU 问世了。无论如何，单精度操作将继续比双精度操作快 8 ~ 10 倍，即使是在这些新的 GPU 上。

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。为区别这种使用 GPU 的风格，有些人称之为通用 GPU（General Purpose GPU，GPGPU）。在厌倦了使用图形 API 和图形绘制语言描述问题之后，他们开发了类 C 编程语言，可以直接在 GPU 上编程。Brook 是其中一种，它是一种面向 GPU 的流语言。NVIDIA 的 CUDA（Compute Unified Device 的可编程性 Architecture）无论在硬件和编程语言方面都得到了进一步的提高，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。GPU 在并行计算方面的应用随着可编程性的提高正在不断增长。

7.7.1　NVIDIA GPU 体系结构简介

附录 A 深入介绍了 GPU 和最新的 NVIDIA GPU 体系结构——Tesla。因为 GPU 在自己的应用领域不断改进，它们不仅具有如上所述的不同体系结构，而且使用不同的术语。在了解这些 GPU 术语之后，你会看到与前面章节介绍方法的相似性，如细粒度多线程和向量。

为了帮助你转换到新的术语上，我们对 Tesla GPU 体系结构中的理念和术语以及 CUDA 编程环境进行简要介绍。

一个分离的 GPU 芯片位于一个独立的卡上，该卡通过 PCI-Express 接口连接到标准 PC 中。所谓的板载 GPU（motherboard GPU）是集成到主板芯片组（如北桥或南桥，见第 6 章）的 GPU。

GPU 通常提供一系列性价比不等的芯片，所有软件均可互相兼容。基于 Tesla 的 GPU 芯片可以提供 1 ~ 16 个节点，NVIDIA 称之为多处理器（multiprocessor）。在 2008 年早期，最大版本是 GeForce 8800 GTX，内含 16 个多处理器，时钟频率为 1.35 GHz。每个多处理器包含 8 个多线程单精度浮点单元和整数处理单元，NVIDIA 称之为流处理器（streaming processor）。

由于该体系结构包括一个单精度浮点乘加指令，因此 8800 GTX 的单精度乘加的最高性能是：

$$16\ \text{MPs} \times \frac{8\ \text{SPs}}{\text{MP}} \times \frac{2\ \text{FLOPs/instr}}{\text{SP}} \times \frac{1\text{instr}}{\text{clock}} \times \frac{1.35 \times 10^9 \text{clocks}}{\text{second}} = \frac{16 \times 8 \times 2 \times 1.35\ \text{GFLOPs}}{\text{second}}$$

$$= \frac{345.6\ \text{GFLOPs}}{\text{second}}$$

GeForce 8800 GTX 的 16 个多处理器中的每一个都有软件管理的 16 KB 局部存储器和 8192 个 32 位寄存器。8800 GTX 的存储器系统由六片 900 MHz Graphics DDR3 DRAM 构成，每个 DRAM 为 8 字节宽度，容量为 128 MB。因此存储器总大小为 768 MB。GDDR3 的最大存储器带宽是：

$$6 \times \frac{8\ \text{Bytes}}{\text{transfer}} \times \frac{2\ \text{transfers}}{\text{clock}} \times \frac{0.9 \times 10^9 \text{clocks}}{\text{second}} = \frac{6 \times 8 \times 2 \times 0.9\ \text{GB}}{\text{second}} = \frac{86.4\ \text{GB}}{\text{second}}$$

为了隐藏存储器延迟，每个流处理器都有硬件支持的线程。32 个线程构成一组，称之为 warp。warp 是调度的基本单位，一个 warp 内最多有 32 个活动线程以 SIMD 的方式并行执行。该多线程体系结构通过允许线程选择不同分支路径的方式来支持条件语句。当 warp 中的线程遇到分支路径时，warp 会使用一些非活动线程顺序执行两条分支，这使得活动线程的执行速度减缓。一旦分支路径已经完成，硬件会将线程返回到一个完全活动的状态。为了获得最佳性能，一个 warp 中的 32 个线程需要同时并行执行。以类似的方式，硬件也在不断查找来自不同线程的地址流，以便将独立的请求尽可能合并为数量较少但是长度较大的存储器块传输，从而提高存储器性能。

图 7-7 将这些特征汇总在一起，并将 Tesla 多处理器与 Sun UltraSPARC T2 核（见 7.5 节和 7.11 节）进行了比较。两者都是硬件多线程，以按时间调度线程的方式，在纵轴上显示。横轴显示出每个 Tesla 多处理器由 8 个流处理器构成，每周期都在执行 8 个并行线程。如上所述，当

一个 warp 中 32 个线程都在以类似 SIMD 的方式一起执行时可达到最高性能，Tesla 体系结构称之为单指令多线程（single-instruction multiple-thread，SIMT）。SIMT 可以动态发现 warp 中哪些线程能够一起执行相同指令，哪些线程在该周期处于空闲状态。T2 核仅包含一个多线程处理器。每个周期它执行来自一个线程的一条指令。

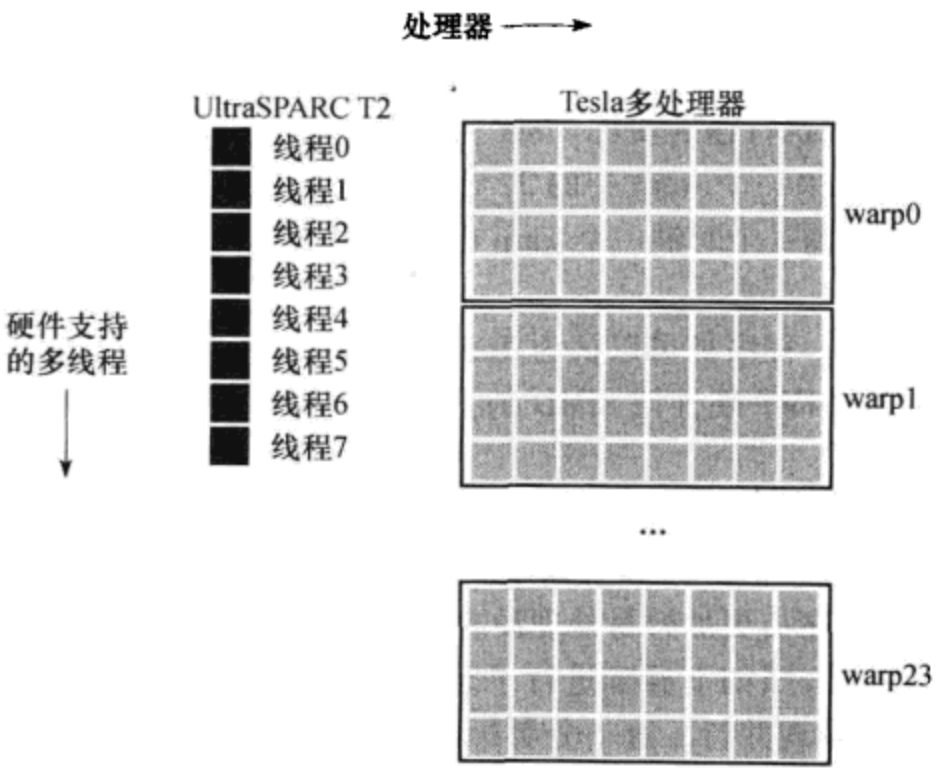


图 7-7　Sun UltraSPARC T2（Niagara 2）中一个核与一个 Tesla 多处理器的比较

T2 核是一个单一的处理器，使用硬件支持的多线程技术，线程数量为 8。Tesla 多处理器包含 8 个流处理器，也使用硬件支持的多线程技术，内含 24 个 warp，每个 warp 内含 32 个线程（8 个处理器乘上 4 个时钟周期）。T2 能在每个时钟周期切换线程，而 Tesla 每两个或四个周期切换一次 warp。比较两者的一种方法是 T2 仅能随时间实现多线程，而 Tesla 可以随时间和空间实现多线程；也就是说，8 个流处理器可以按 4 个时钟周期分段实现随空间变化的多线程。

Tesla 多处理器使用细粒度硬件多线程随时间调度 24 个 warp，调度时将 4 个时钟周期作为一个时间块，如图中的纵轴方向所示。与此类似，UltraSPARC T2 随时间调度 8 个硬件支持的线程，纵轴方向显示出每个周期执行一个线程。因此，正如 T2 硬件在不同线程之间切换保持 T2 核一直处于忙碌状态一样，Tesla 硬件也是在不同 warp 之间切换以保持 Tesla 多处理器一直忙碌。主要的区别是 T2 核仅有一个处理器，因此能够每个时钟周期都切换线程，而在 Tesla 微处理器中切换 warp 的最小单位是两个时钟周期切换 8 个流核。由于 Tesla 的定位是面向具有大量数据级并行的程序，设计者认为每两个或四个时钟周期切换一次相对于每时钟周期切换一次的性能差别很小，而限制切换频率极大简化了硬件设计。

CUDA 编程环境也有自己的术语。一个 CUDA 程序是用于异构 CPU 和 GPU 的一个统一的 C/C ++ 程序。它在 CPU 上执行，并将并行任务分派到 GPU 上。这项工作由来自主存的数据传输和线程分配（thread dispatch）构成。线程是 GPU 程序的一段代码。由程序员指定线程块（thread block）中的线程数量，以及希望在 GPU 上开始执行的线程块的数量。由程序员指定线程块的原因在于线程块中的所有线程都会调度到同一多处理器执行，它们全部共享同一局部存储器。因此它们可以通过存取操作而不是消息机制实现通信。CUDA 编译器为每个线程分配寄存器，约束是每线程的寄存器数量乘上每线程块的线程数量不超过每个多处理器包含的 8192 个寄存器。

一个线程块最多有 512 个线程。线程块中的每 32 个线程构成一组，封装为 warp。大的线程块相对小的线程块效率更高，但是线程块也可以小到只有一个线程。如上所述，小于 32 个线程的线程块和 warp 的执行效率没有填满的 warp 高。

硬件调度器总是试图调度每个多处理器中尽可能多的线程块。如果调度成功，调度器也负责将 16 KB 局部存储器在不同线程块之间进行动态划分。

7.7.2　深入理解 GPU

像 NVIDIA Tesla 体系结构一样的 GPU 不能完全归于前面所讲到的计算机分类中（见图 7-6）。显

然，内含 16 个 Tesla 多处理器的 GeForce 8800 GTX 是一个 MIMD。问题是 Tesla 多处理器和构成 Tesla 多处理器的 8 个流处理器该如何归类?

回顾我们前面谈到的 SIMD，它适合 for 循环语句编程，但是不适合 case 和 switch 语句编程。Tesla 的目标是为数据级并行提供高性能计算能力，同时让程序员易于处理独立的线程级并行。Tesla 允许程序员认为该多处理器是由 8 个流处理器构成的多线程 MIMD，但是当同一 warp 中的多个线程可以一起执行时，硬件又试图将 8 个流处理器结合在一起以 SIMT 的方式工作。当若干线程独立工作并沿着独立的执行路径时，它们的执行速度要比 SIMT 方式低得多，因为一个 warp 中的 32 个线程是共享一个指令取指单元的。如果一个 warp 中的 32 个线程都执行独立指令，每个线程都会以峰值性能的 1/16 在运行。如果每隔 4 个时钟一个 warp 都有 32 个线程可以执行在 8 个流处理器上，那么系统就达到峰值性能。

因此，每个独立的线程都有自己的有效 PC，程序员可以认为 Tesla 多处理器是 MIMD，但是程序员必须谨慎处理控制转移语句，以便允许 SIMT 硬件按 SIMD 的方式执行 CUDA 程序，从而获得预期的性能。

与向量体系结构相比，向量体系结构需要借助向量化编译器在编译时识别数据级并行，并产生相应的向量指令；而 Tesla 体系结构的硬件实现可以在运行时发现线程间的数据级并行。因此，Tesla GPU 不需要向量化编译器，并使得程序员更加容易处理没有数据级并行的那部分代码。为了深入理解这一独特的方法，图 7-8 对 GPU 按照如下标准进行了分类：是指令级并行还是数据级并行，是在编译时发现并行还是在运行时发现并行。这一分类标志着 Tesla GPU 为计算机体系结构开辟出一片新天地。

	静态：编译时发现	动态：运行时发现
指令级并行	VLIW	超标量
数据级并行	SIMD 或向量机	Tesla 多处理器

图 7-8　基于静态/动态和 ILP/DLP 的处理器体系结构的硬件分类及实例

小测验

是非判断：GPU 借助图形 DRAM 芯片来降低存储器延迟，从而提高图形应用程序的性能。

7.8　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。本节讨论不同多处理器网络的优点与缺点。

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，某些核可能是相邻的，而其他核可能在芯片另一端。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，在这芯片设计受功耗限制的时代里，不同组织结构具有不同功效，功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，图中的每条弧表示通信网络中的一条链路。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。在本节中，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 - 存储器节点和其他开关的链接。网络相对总线的第一个改进就是网络将若干节点组成的序列连接到了一起。



该拓扑叫做环（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。

和总线不同的是，环可以同时进行多个传输。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**[⊖]，它是每条链路带宽与链路数量的乘积。该度量表示网络最好情况下的性能。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，也就是该链路带宽的两倍。

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**[⊜]。它的计算是通过将机器分割为两个部分，每一部分都包含一半节点。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。

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。由于这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**[⊜]，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 $P \times (P-1)/2$，而切分带宽是 $(P/2)^2$。

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，很大程度上依赖于机器上所运行的并行程序负载的通讯特征。

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 7-9 给出了两种常见拓扑。在实际的机器中常常在这些简单拓扑中增加一些额外的链路以提高性能和可靠性。

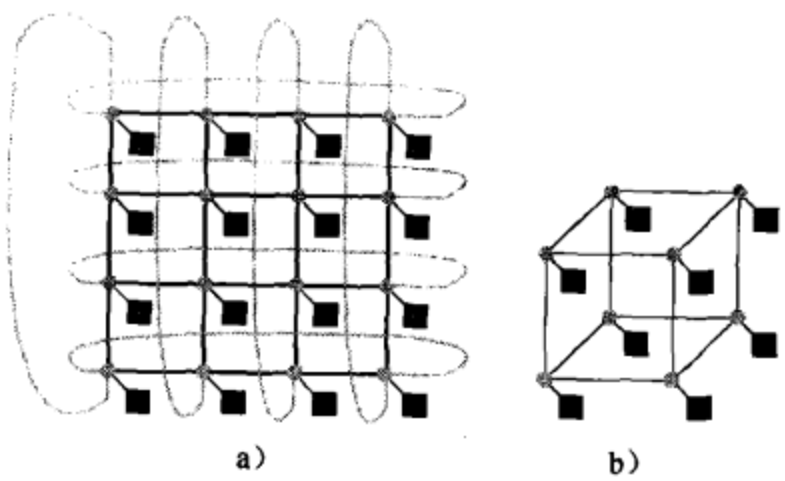


图 7-9　已经出现在商业并行处理器中的网络拓扑
a）16 个节点的 2 - D 网格；b）8 个节点的 n 维立方体
其中灰色圆形表示开关，而黑色方块表示处理器 - 存储器节点。尽管一个开关可以有多个链路，但是通常只有一个连接到处理器。布尔 n 维立方体拓扑是一个使用 2^n 个节点构成的 n 维互连，每个开关需要 n 个链路（并加上一个处理器链路），因而存在 n 个最近相邻节点。这些基本拓扑常常会补充一些额外链路，从而提高性能和可靠性。

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

⊖ 网络带宽（network bandwidth）：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共同的传输速度。

⊜ 切分带宽（bisection bandwidth）：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。

⊜ 全连接网络（fully connected network）：通过专用通信链路连接所有处理器 - 存储器节点的网络。

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**[⊖]，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 7-10 给出了两种常见的多级结构。全连接网络或**交叉开关网络**[⊜]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 7-10 中的 Omega 网络在从 P0 向 P6 发送信息的同时，不能从 P1 向 P7 发送信息。

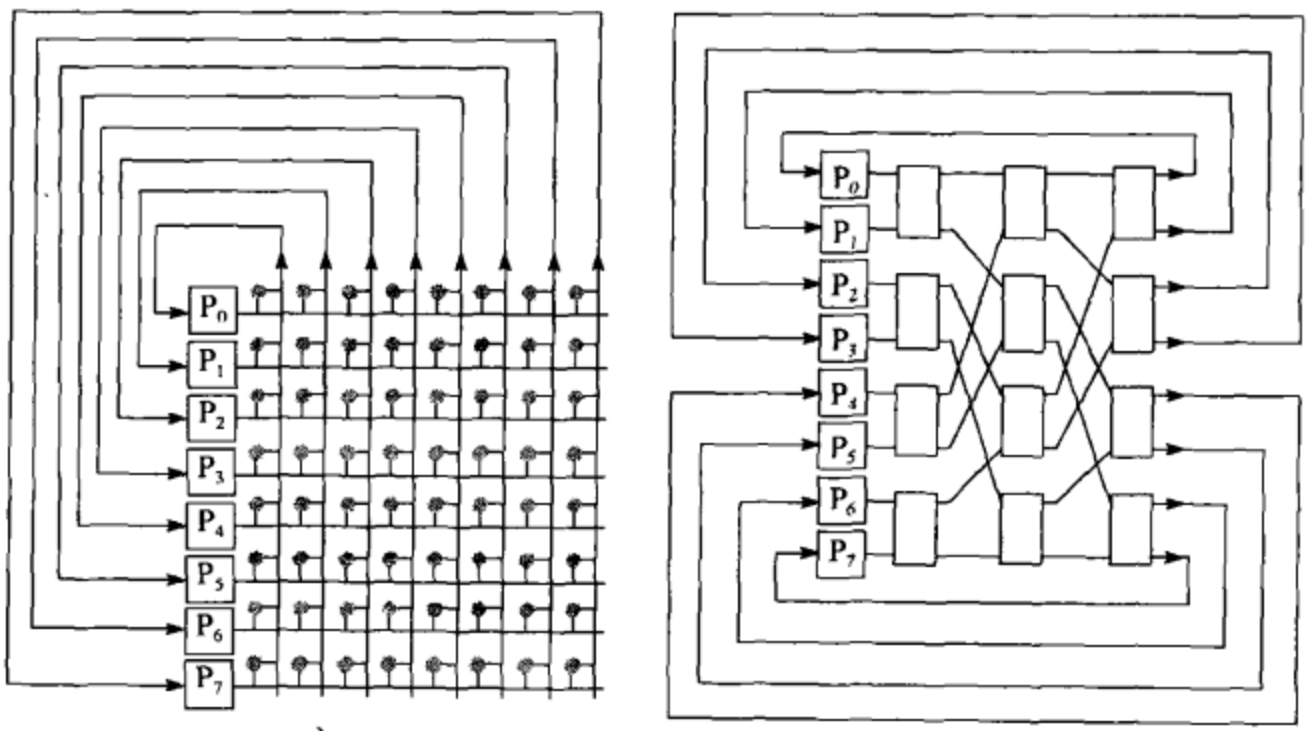


图 7-10　常见的八节点多级网络拓扑

a）交叉开关；b）Omega 网络；c）Omega 网络的开关盒

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；数据从底部进入，从右边的链路退出。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

在本节对所有网络简单分析的时候，忽略了一些在网络构建时需要考虑的一些实际因素。在高速时钟下，链路的距离影响通信成本——一般来说，距离越长，在高速时钟下的成本越大。较短的距离也会使得更加容易地将更多的连线增加到同一链路中，因为芯片内连线越短，

⊖ 多级网络（multistage network）：每个节点提供一个小开关的网络。

⊜ 交叉开关网络（crossbar network）：任何一个需一次即可与其他任意一个节点通信的网络。

驱动连线的功耗就会越低。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功耗。例如，功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，在使用硅工艺制造时可能是不切实际的。

7.9　多处理器基准测试程序

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伎俩，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，但即便如此，在比较不同问题规模的程序结果时仍要小心。

图7-11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DAXPY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，它允许使用者以任何形式和任何语言重写Linpack，只要保持计算结果的正确性。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，它主要测量作业级并行，因为这些作业之间没有通信。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。该程序需要强比例缩放，尽管它提供了两组数据集。
- 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。它由五个核心构成，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，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）的多线程程序组成。它们主要专注于新兴市场，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。

⊖ Pthread：创建和操作线程的一个UNIX API。它使用一个库提供。

⊜ OpenMP：在C、C++或Fortran中用于共享内存多处理编程的API，可以运行于UNIX和Microsoft平台。它包括编译器指示、一个库和运行时指示。

基准测试程序	比例缩放?	重编程?	说明
Linpack	弱	是	稠密矩阵线性代数［Dongarra，1979］
SPECrate	弱	否	独立作业并发［Henning，2007］
SPLASH 2［Woo 等，1995］	强（尽管提供两种问题规模）	否	复杂一维 FFT 分块 LU 分解 分块稀疏楚列斯基因式分解 整数基数排序 Barnes-Hut 算法 自适应快速多极子算法 海洋模拟 种源光能辐射 光线追踪 体绘制程序 带有空间坐标数据结构的水模拟 不带有空间坐标数据结构的水模拟
NAS 并行基准测试程序［Bailey 等，1991］	弱	是（仅支持 C 或 Fortran）	EP：高度并行 MG：简化的多重网格 CG：用于共轭梯度方法的非结构化网格 FT：使用 FFT 对 3D 偏微分方程求解 IS：超大整数排序
PARSEC 基准测试程序集［Bienia 等，2008］	弱	否	Blackscholes：使用 Black-Schole 偏微分方程的期权定价 Bodytrack：人体追踪 Canneal：用于优化路由选择的缓存感知模拟退火算法 Dedup：支持重复数据删除的下一代压缩算法 Facesim：面部表情的模拟 Ferret：内容相似度搜索服务器 Fluidanimate：使用 SPH 方法的流体动力学演示 Freqmine：频繁项集挖掘 Streamcluster：对一个输入流的在线聚类 Swaptions：期权组合定价 Vips：图形处理 x264：H. 264 视频编码
Berkeley 设计模式［Asanovic 等，2006］	强或弱	是	有限状态机 组合逻辑 图遍历 结构化网格 稠密矩阵 稀疏矩阵 波谱法（FFT） 动态编程 多体问题 MapReduce 限界回溯算法 图模型推理 非结构化网格

图 7-11　并行基准测试程序的实例

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。

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年代的前五年。但是，这些准则在变革开始之后就不合时宜了。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。他们识别了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创新。因此，具有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。我们将在7.11节看到此类基准测试程序的实例。

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。

7.10　Roofline：一个简单的性能模型

本节基于Williams和Patterson 2008年的一篇论文。近几年，计算机体系结构中普遍认同的传统观点导致了微处理器设计间的相似性。几乎每台桌面计算机或服务器都使用缓存、流水线、超标量指令槽、分支预测和乱序执行。尽管指令集不同，但是所有微处理器的设计如出一辙。

多核时代的到来可能标志着微处理器日趋多样化，因为目前尚无一个公认的体系结构能以最简单的方式支持编写并行处理程序并有效运行，且能随着时间推移根据处理器核数按比例缩放。而且，由于每芯片核的数量确实在不断增加，一家制造商希望同时提供每芯片内包含不同数量核的产品以满足不同价位的要求。

考虑到不断增加的多样性，如果我们能拥有一个简单的模型将是十分有益的，可以用其分析不同设计的性能。这个模型不需要是完美的，只要有所见地就行。

第5章的3C模型并不是一个完美的模型，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，缓存缺失的原因在一个设计中可能是因为容量，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20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。

为了找到这样一个模型，让我们从图7-11中的13个Berkeley设计模式开始。设计模式的想法是：一个给定的应用程序性能是实现这些设计模式若干核心的加权和。我们将在这里评估每个核心，但是需要注意的是，实际的应用程序是许多核心的组合。

尽管有不同数据类型的许多版本，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：

$$\frac{\text{浮点操作数}/\text{秒}}{\text{浮点操作数}/\text{字节}} = \text{字节}/\text{秒}$$

存储器每访问一字节所包含的浮点操作比例被称作**算术密度**[⊖]。它的计算可以用程序中总的

⊖ 算术密度（arithmetic intensity）：一个程序中浮点操作数量与访问主存字节数量的比值。

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 7-12 给出了图 7-11 中几种 Berkeley 设计模式的算术密度。

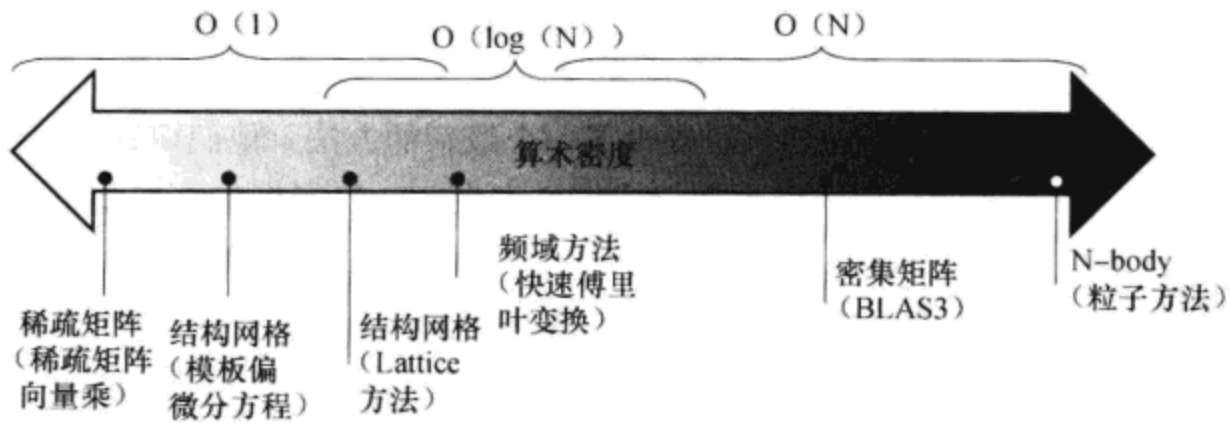


图 7-12　算术密度，计算方式为用运行程序中总的浮点操作数除以访问主存总的字节数［Williams，Patterson，2008］。

一些核心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。

7.10.1　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 Patterson，2008］。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 5.2.5 节的相关说明）。

图 7-13 给出了针对一台计算机的模型，注意不是针对每个核心的模型。纵轴 Y 表示浮点性能，从 0.5 到 64.0 GFLOPs/秒。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。

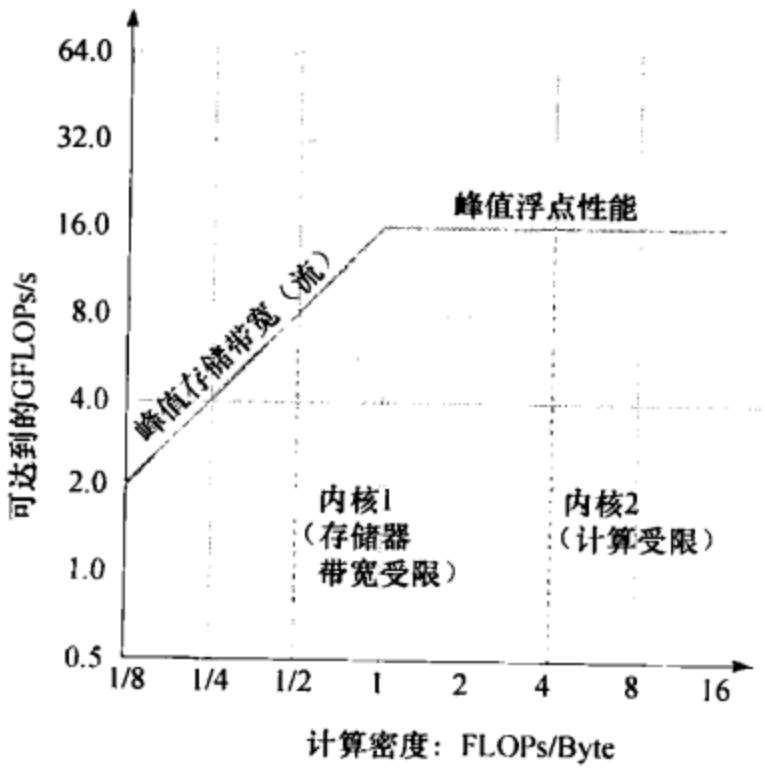


图 7-13　Roofline 模型［Williams，Patterson，2008］

本例具有 16 GFLOPS/s 的峰值浮点性能和 16 GB/s 的峰值存储带宽，该数据来自流测试程序（由于流实际上是四次测量，图中的线是四次的均值）。左边的彩色点垂线标识内核 1，其计算密度为 0.5 FLOPs/byte。在 Opteron X2 上，受限于低于 8 GFLOPS/s 的存储器带宽。右边的点垂线标识内核 2，计算密度为 4 FLOPs/byte。它只受限于 16 GFLOPS/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 2 GHz 的双核）。

对给定的核心，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，因为这是一个硬界限（hardware limit）。

我们如何画出峰值存储性能呢？由于 X 轴是 FLOPs/字节，Y 轴是个 GFLOPs/秒，所以字节/秒只是图中一条 45°的对角线。因此，我们画出第三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 7-13 中画出该线：

可达到的 GFLOPs/秒 = 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在不同算术密度下的性能上界。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；它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 7-13 中，核心 2 属于前者，而核心 1 属于后者。在有了一台计算机的 Roofline 模型之后，你可以重复利用它，因为它不会受核心影响。

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才能获得最大性能。如果它过于靠左，那么几乎所有核心都可以达到最大性能。对于这两种情况我们都将会给出相应实例。

7.10.2　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 2.2 GHz，Opteron X4 为 2.3 GHz——所以 Opteron X4 的峰值浮点性能是 Opteron X2 的 4 倍还多，而两者 DRAM 带宽完全相同。Opteron X4 还有 2 MB 的三级缓存，而 Opteron X2 没有。

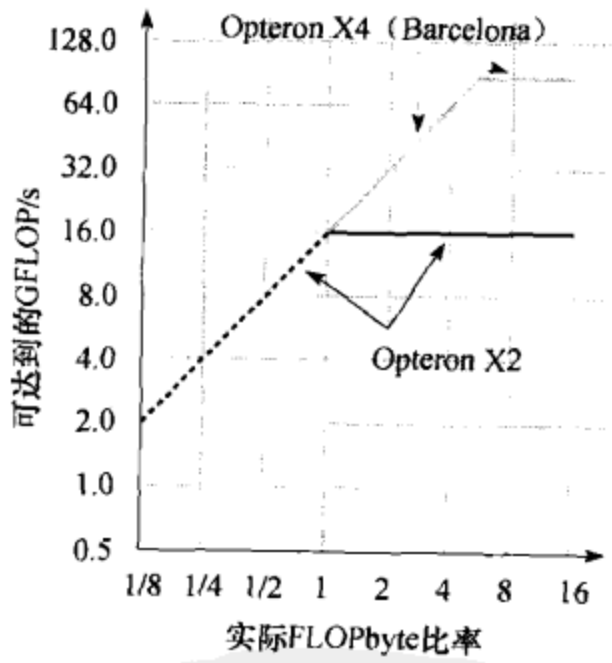


图 7-14　两代 Opteron 的 Roofline 模型

Opteron X2 的屋顶线与图 7-11 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在 Opteron X4 中可能是存储性能受限。

图 7-14 比较了两个系统的 Roofline 模型。正如我们所期望的那样，脊点从 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，核心的算术密度必须大于 1，或者核心的工作集必须适合 Opteron X4 的缓存。

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：

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 3.5.5 节的精解），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指令混合是浮点操作和非整数指令。

2）提高指令级并行并应用 SIMD。对超标量体系结构，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（见第 4 章）。这里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。对 x86 体系结构而言，一个单一的 SIMD 指令可以对一对双精度操作数进行操作，因此它们应该被尽量使用。

为了克服存储瓶颈，可以采用下面的两种优化：

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，这使得执行软件预取指更加容易，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）。大多数现在的微处理器都在片内包含了内存控制器。如果系统中含有多颗芯片，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访问对于其他芯片是本地的 DRAM。后者会降低性能。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，而存储带宽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针对该计算机优化的先后次序。

图 7-15 相对于图 7-13 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。

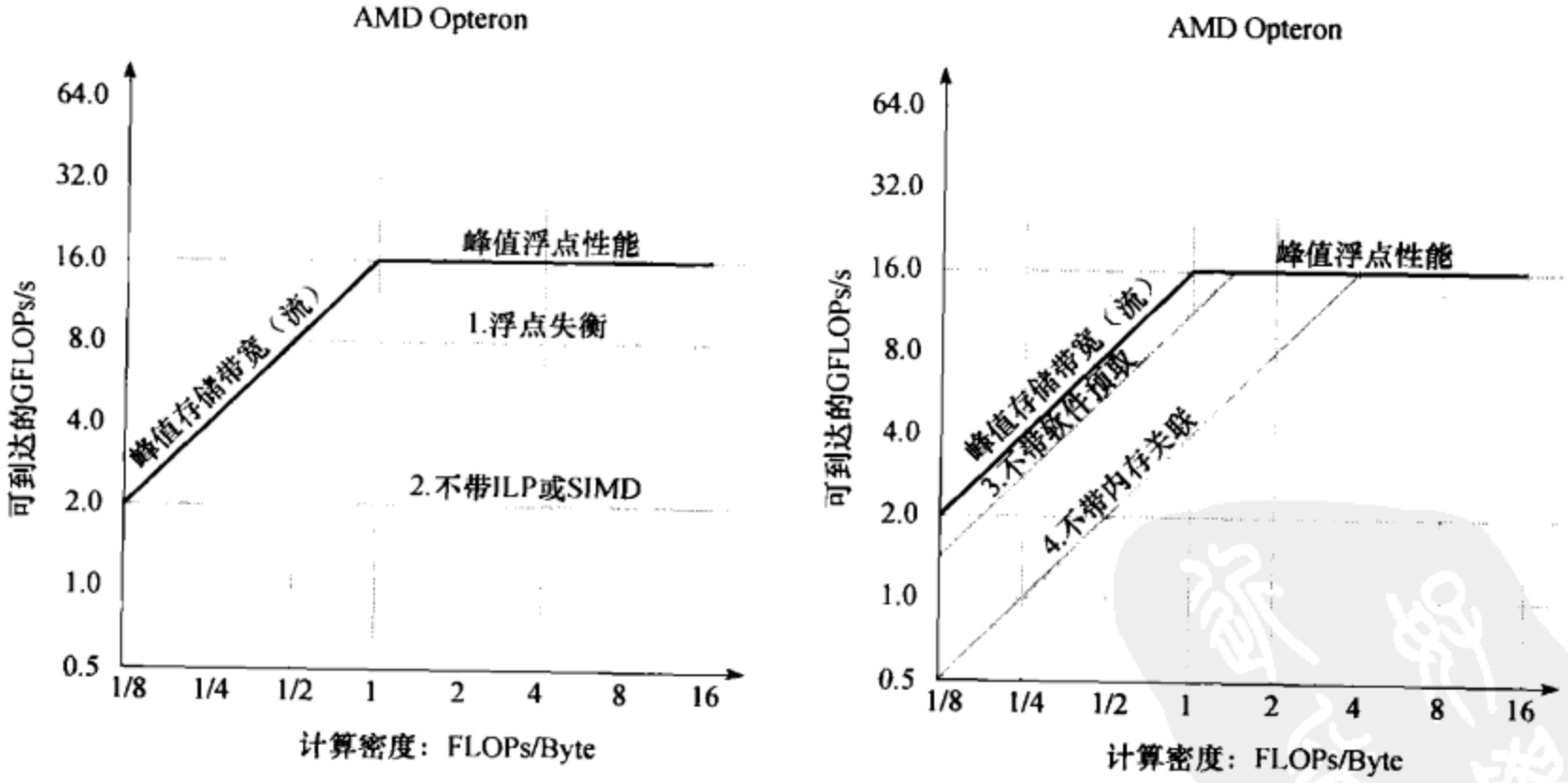


图 7-15　带有天花板的 Roofline 模型

其中左图表示计算性能的“天花板”，其中 1 表示浮点操作混合失衡情况下性能为 8 GFLOPs/秒，2 表示同时未使用 ILP 和 SIMD 下的性能为 2 GFLOPs/秒。右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 11 GB/秒，4 表示同时没有优化内存关联的带宽为 4.8 GB/秒。

天花板之间间隙的厚度和下一个更高的限制表示优化之后的收益。因此，图 7-15 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 7-16 将图 7-15 中的天花板整合到一张图中。核心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 7-16 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 2 落在右边灰色梯形区域，表示只工作在计算优化上。核心 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 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。

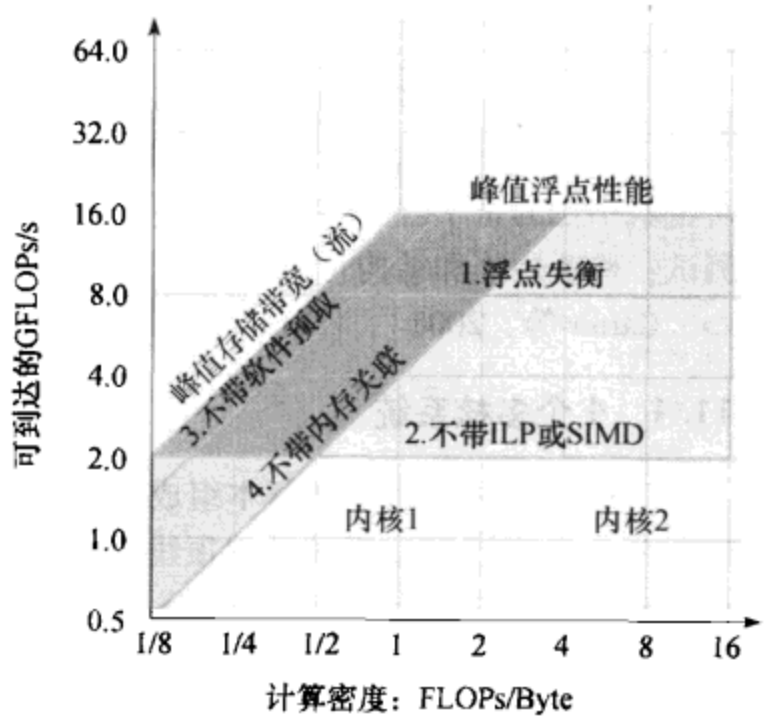


图 7-16　将图 7-13 中两图重叠的 Roofline 模型
算术密度处于右边灰色梯形区域的核心应当着重于计算优化，而处于浅灰色三角形区域的核心应当着重于存储带宽优化。处于灰色和浅灰色平行四边形区域的核心两种优化都应当考虑。例如核心 1 落在中间的平行四边形中，可尝试优化 ILP 和 SIMD、内存关联、软件预取指等。核心 2 落在右边的梯形区域，可尝试优化 ILP 和 SIMD 以及浮点操作均衡等。

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 7-12）。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，缓存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改善时间局部性，并将使用相近地址的语句分组到一起。许多计算机提供特殊的缓存指令，可以先将数据分配到缓存中但是先不从存储器中填充，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。这种右移会使核心移到一个不同的优化区域。

下一节会使用屋顶线模型来分析四种最近的多核微处理器运行两个实际应用程序核心的差异。上面的这些例子不仅显示了该模型如何帮助程序员提高性能，而且还可以帮助体系结构人员从哪里着手优化硬件，从而提高那些他们认为重要的核心性能。

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，就会存在一些含糊不清的地方。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 DRAM 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：上图的两轴分别是每秒钟的浮点操作次数和访问主存的算术密度。屋顶线模型也可用于其他核心和计算机，使用不同的性能度量。

例如，如果工作集采用该计算机的二级缓存，那么屋顶线模型中斜线部分将表示二级缓存带宽而不再是主存带宽，并且 X 轴的算术密度将基于每访问二级缓存一个字节时的 FLOPs。二级缓存的斜切线将上移，脊点也很可能左移。

再举一个例子，如果核心变成排序，那么 Y 轴上每指令的浮点操作次数将会变成每秒钟完成排序的记录数量，算术密度也会变成每访问 DRAM 一个字节所操作的记录数量。

屋顶线模型甚至用于 I/O 密集型的核心。Y 轴将变成每秒钟的 I/O 操作次数，X 轴将变成每次 I/O 操作的平均指令数，屋顶线将表示峰值 I/O 带宽。

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。

7.11　实例：使用屋顶线模型评估四种多核处理器

由于在这场并行革新中最好的方法尚不明确，所以当我们见到不同多核芯片都采用不同设计时也就不足为奇了。在本节中，我们将使用图 7-11 中的两种设计模式核心对 4 种多核系统进行测试：稀疏矩阵和结构化网格。（本节中的相关内容出自［Williams、Oliker 等，2007］，［Williams、Carter 等，2008］，［Williams 和 Patterson，2008］。）

7.11.1　4 个多核系统

图 7-17 给出 4 个系统的基本组成，图 7-18 给出了本节实例的关键特性。这些系统都是双插槽的。图 7-19 给出每个系统的屋顶线模型。

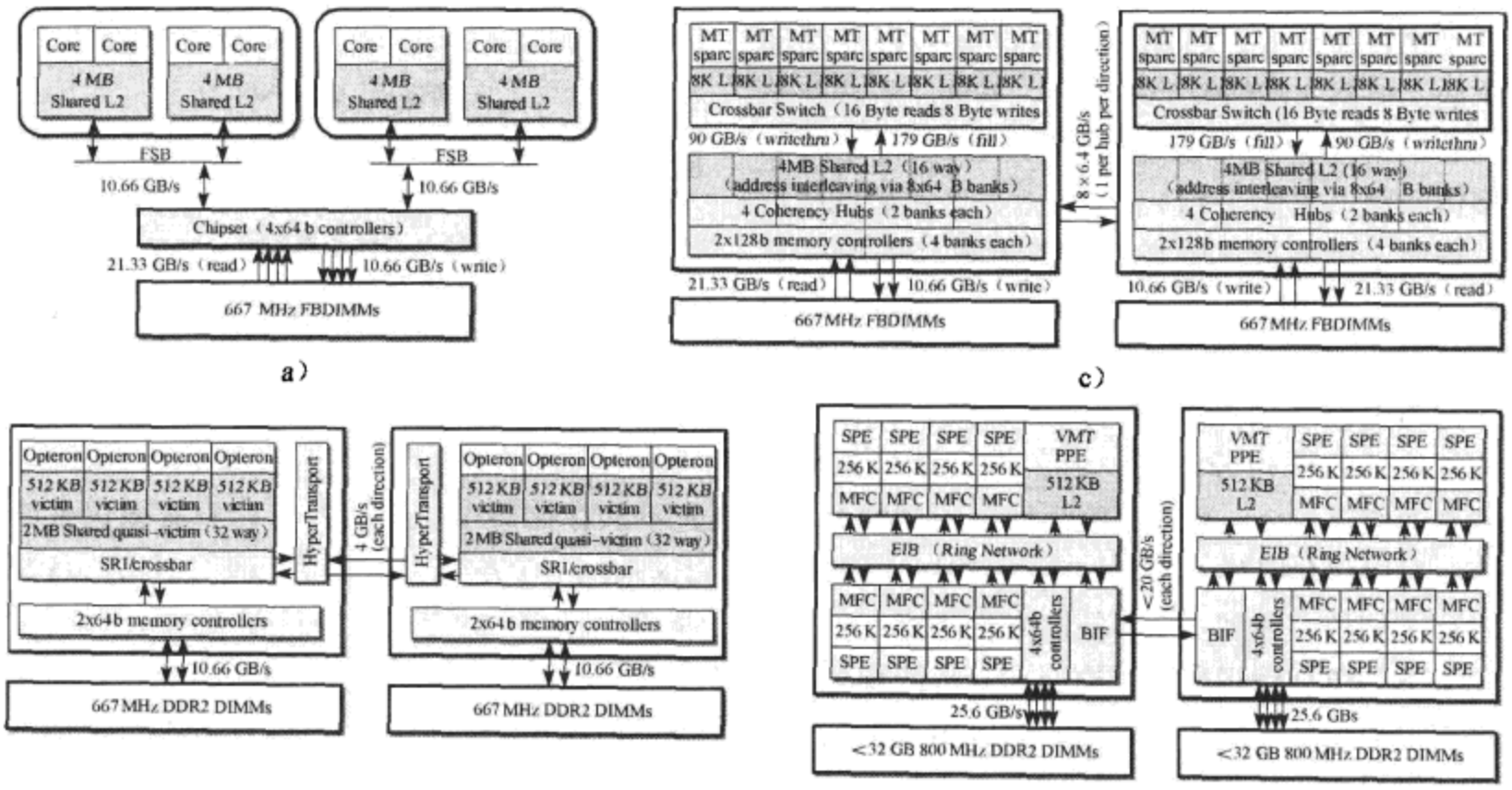


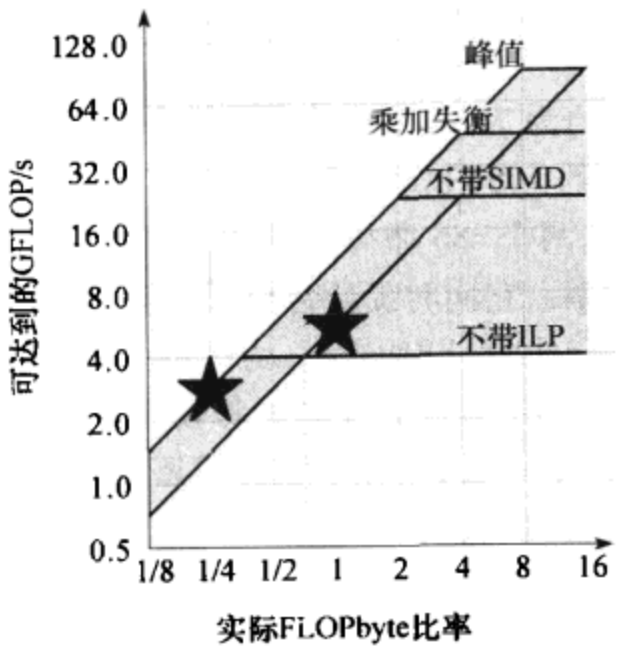
图 7-17　四种最近的多处理器，每个使用双插槽

a）Intel Xeon e5345（Clovertown）；b）AMD Opteron X4 2356（Barcelona）；
c）Sun UltraSPARC T2 5140（Niagara 2）；d）IBM Cell QS20
注意 Intel Xeon e5345（Clovertown）有一个独立的北桥芯片，而其他没有。

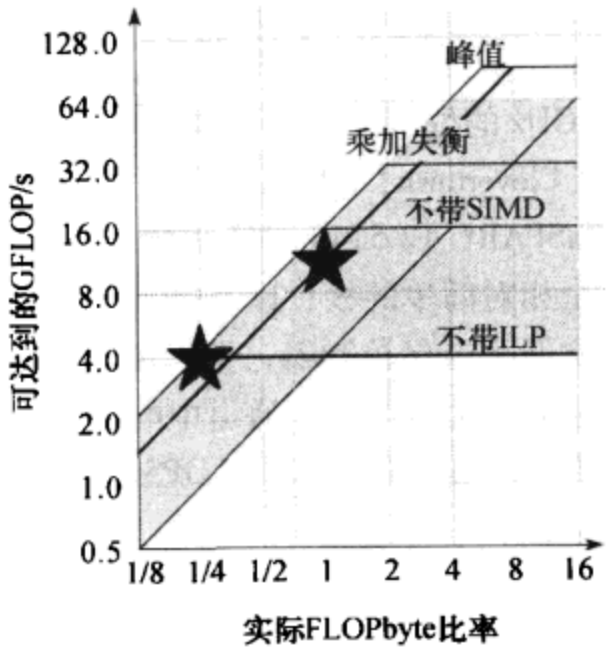
MPU 类型	ISA	线程数	核数	插槽数	时钟 GHz	峰值 GFLOP/s	DRAM：峰值 GB/s、时钟频率及类型	
Intel Xeon e5345（Clovertown）	x86/64	8	8	2	2.33	75	FSB：2×10.6	667 MHz FBDIMM
AMD Opteron X4 2356（Barcelona）	x86/64	8	8	2	2.30	74	2×10.6	667 MHzDDR2
Sun UltraSPARC T2 5140（Niagara 2）	Sparc	128	16	2	1.17	22	2×21.3（读） 2×10.6（写）	667 MHz FBDIMM
IBM Cell QS20	Cell	16	16	2	3.20	29	2×25.6	XDR

图 7-18　四种最近多核处理器的关键特性

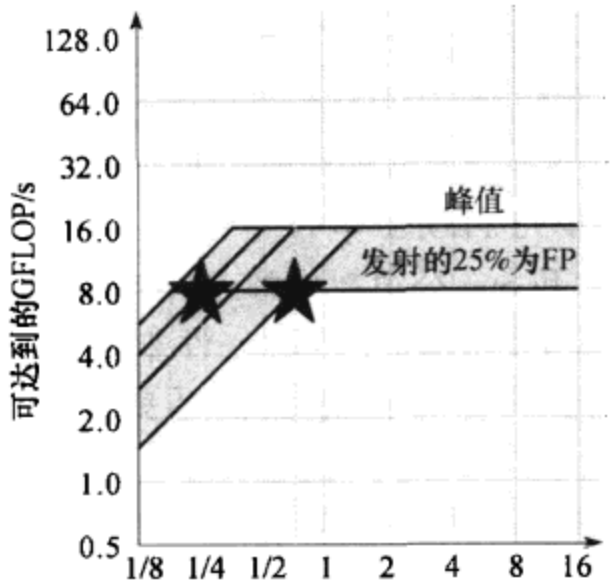
尽管 Xeon e5345 和 Opteron X4 使用相同速度的 DRAMs，但是 Stream 基准测试程序显示出 Opteron X4 具有较高的实际存储带宽，因为 Xeon e5345 的前端总线不是很有效。



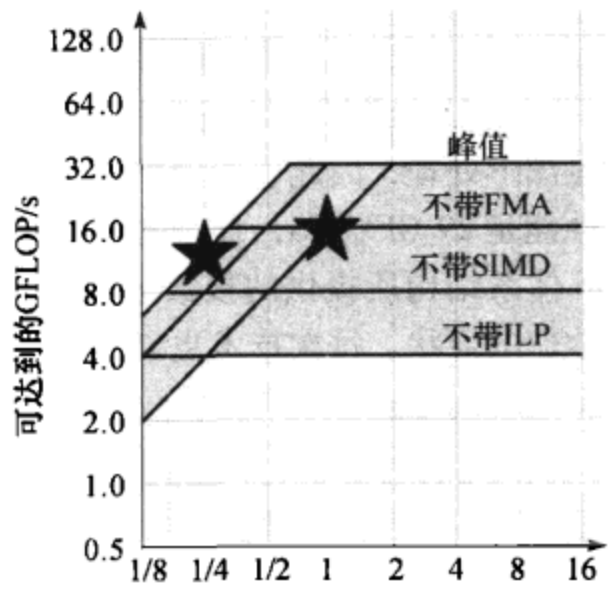
a）



b）



c）



d）

图7-19　图7-15中多核多处理器的Roofiine模型

a）Intel Xeon e5345（Clovertown）；b）AMD Opteron X4 2356（Barcelona）；
c）Sun UltraSPARC T2 5140（Niagara 2）；d）IBM Cell QS20

最上面与图7-13相同。从左上角开始，计算机依次是：（a）Intel Xeon e5345（Clovertown），（b）AMD Opteron X4 2356（Barcelona），（c）Sun UltraSPARC T2 5140（Niagara 2）和（d）IBM Cell QS20。注意，这四个微处理器的脊点与x轴的交叉点的计算密度分别为6、4、1/3利3/4。点垂线是针对本部分的两个内核的，星标表示这些核进行所有的优化之后所获得的性能。SpMV用左侧的一对点垂线表示。它之所以有两条线是因为在寄存器阻塞优化的基础上，它的算术密度可从0.166提升到0.255。LBHMD是右边的点垂线。它在（a）和（b）中有一对线，是因为当处理器可以向整个块中写入新数据时，cache优化会针对缺失而跳过填充cache块。这种优化使算术密度从0.70增加到1.07。（c）中是一条单线，因为UltraSPARC T2不提供cache优化。（d）中是一条位于1.07的单线，因为Cell的局部存储由DMA装入，因而程序不用像cache那样取进不必要的数据。

Intel Xeon e5345（代号Clovertown）每个插槽包含四个核，通过封装可将两个核集成到一个插槽上。这两个芯片共享一个前端总线，并连接到一个分离的北桥芯片组上（见第6章）。该北桥芯片组支持两个前端总线，因此支持两个插槽。它包含支持667 MHz的全缓冲DRAM DIMM（FB-DIMM）内存控制器。该双插槽系统使用2.33 GHz的处理器时钟频率，在四个系统中具有最高的峰值性能：75 GFLOPS。然而，从图7-19中的屋顶线模型可以看到，这只能在算术密度大于等于8时才能达到。其原因是使用了双前端总线进行两者的接口，这只能为程序提供较低的存储带宽。

AMD Opteron X4 2356（Barcelona）每颗芯片包含四个核，并且每个插槽都有一颗芯片。每

颗芯片都有一个板上的内存控制器以及一条独立的667 MHz DDR2 DRAM通道。两个插槽通过专用的Hypertransport链路进行通信，这使得构建“无缝”连接多芯片系统成为可能。该双插槽系统采用2.30 GHz的处理器时钟频率，峰值性能大约为74 GFLOPS。图7-19显示脊点相对Intel Xeon e5345（Clovertown）偏左，对应的算术密度大约为5 FLOPS/字节。

Sun UltraSPARC T2 5140（代号Niagara 2）与这两个x86微体系结构大不相同。它在每颗芯片上使用八个相对简单的核，以及低得多的时钟频率。它同时提供每核八线程的细粒度多线程。一颗芯片包含四个内存控制器，可以驱动四组667 MHz的FBDIMM。为了将两颗UltraSPARC T2芯片连接在一起，把四个内存通道中的两个连接到一起，其他两个留给每颗芯片自己使用。该双插槽系统的峰值性能大约为22GFLOPS，而脊点低得令人吃惊，对应的算术密度只有1/3 FLOPS/字节。

IBM Cell QS20与上面三种也不相同。它是一个异构设计，内含一个相对简单的PowerPC核和八个支持SIMD类型指令的SPE（*Synergistic Processing Element*）。每个SPE也有自己的局部存储器而不是缓存。一个SPE必须将数据从主存转移到自己的局部存储器才能对其操作，而在操作完成之后再将数据写回主存。它使用DMA，与软件预取指有些类似。两个插槽通过多芯片通信的专用链路进行连接。该系统的时钟频率是四个多核中最高的，达到了3.2 GHz，而且它使用XDR DRAM芯片，这一般用在游戏机中。XDR DRAM的特点是带宽高但是容量低。考虑到Cell的主要应用是图形处理，因此它的单精度性能要远远超过双精度性能。该双插槽系统中SPE的峰值双精度性能是29 GFLOPS，脊点的算术密度是0.75 FLOPS/字节。

两种x86体系结构所提供的核数远远小于IBM和Sun公司在2008年早期所能提供的核数，这就是目前他们的现状。每次技术更新换代会使核的数量加倍，所以一件非常有趣的事情就是不知道是x86体系结构会缩短两者核数的差距，还是IBM和Sun会继续加入更多的核，尽管它们专注的领域一个是桌面计算机，一个是服务器。

需要注意的是，这些系统采用了完全不同的存储器系统。Xeon e5345使用了常规的私有一级缓存，二级缓存在一对处理器之间共享。然后通过两条总线使用片外内存控制器连接到常规内存上。相比之下，Opteron X4每颗芯片都有独立的内存控制器和内存，并且每个核都有私有的一级和二级缓存。UltraSPARC T2每颗芯片都有片上内存控制器和四个独立的DRAM通道，所有处理器核共享二级缓存，并且二级缓存分为四组以提高带宽。它在多核设计之上的细粒度多线程使得它可以保持许多存储器操作同时进行。Cell是最为激进的一个设计。它在每个SPE都有局部存储器，并且使用DMA在DRAM与局部存储器之间传输数据。它通过包含很多核以及每个核上的许多DMA传输来承受许多存储器操作同时进行。

下面来看四个多核在两种核心上的性能对比。

7.11.2　稀疏矩阵

第一个核心是稀疏矩阵计算的设计模式SpMV（Sparse Matrix-Vector multiply）。SpMV在科学计算、经济学建模和信息检索中很常见。但遗憾的是，常规的实现方式最多只能发挥单处理器峰值性能的10%。不规则的存储器访问是原因之一，例如使用稀疏矩阵工作的核心就是如此。假设需要完成以下计算：

$$y = A \times x$$

其中A是一个稀疏矩阵，x和y各是一个稠密向量。我们从不同真实应用中选择了14个稀疏矩阵对SpMV性能进行了评估，但是只给出了一个居中的性能结果。算术密度在采用寄存器阻塞优化之前是0.166，在优化之后是0.250 FLOPS/字节。

首先将算法并行化以利用全部处理器核。假定SpMV的算术密度低于图7-19中所有四个多

核处理器的脊点，那么大多数优化都是涉及存储系统的：

- *预取指*：为了挖掘存储系统的最大性能，软件取指和硬件取指都要使用。
- *内存关联*：有三个系统包含局部存储器，该优化手段可以降低访问连接到另外一个插槽的 DRAM 存储器的次数。
- *压缩数据结构*：既然很可能是存储带宽限制了性能，该优化手段使用更小的数据结构来增加性能——例如，将 16 位索引改为 32 位索引，使用空间更加有效的方法来表示稀疏矩阵中每一行的非零值。

图 7-20 展示了在具有不同核数的四个系统上，SpMV 的性能。（同样的结果也出现在图 7-19 中，但使用对数标尺时很难进行性能比较。）请注意，尽管拥有图 7-18 中的最高峰值性能和最高单核性能，Intel Xeon e5345 的四核交付性能却是最低的。Opteron X4 的性能是其性能的 2 倍。Xeon e5345 的瓶颈在于双前端总线。尽管 Sun UltraSPARC T2 时钟频率是最低的，但其具有的大量的简单核却胜过了两个 x86 处理器。IBM Cell 在四个处理器中性能最高。请注意，除 Xeon e5345 外，所有处理器都能对核数进行很好的扩展，尽管 Opteron X4 在四核或更多核时，扩展的更加缓慢。

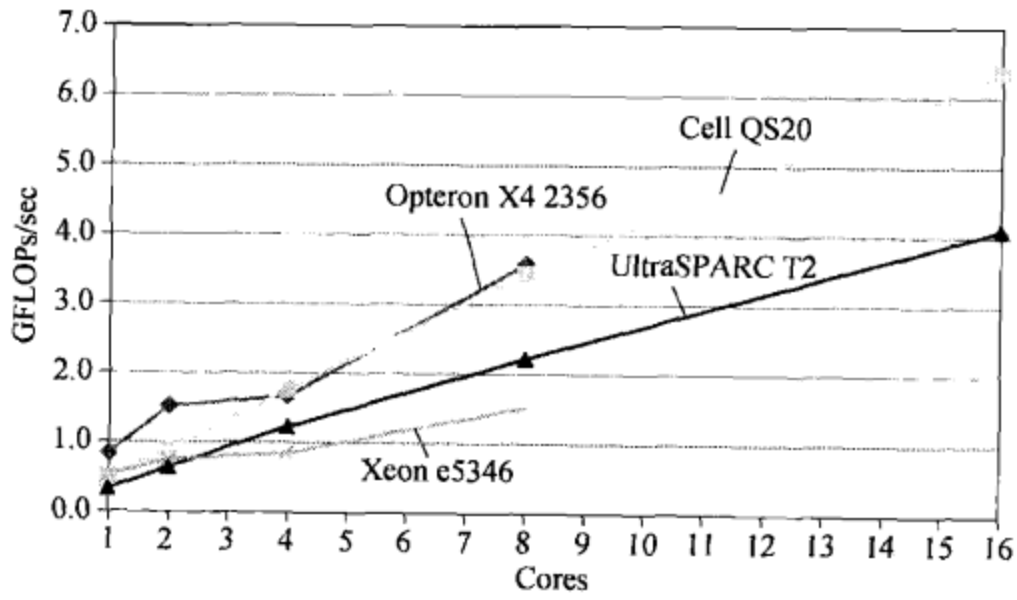


图 7-20　SpMV 在 4 种多核上的性能

7.11.3　结构化网格

第二个核心是一个结构化网格设计模式的实例。LBMHD（Lattice-Boltzmann Magneto-Hydrodynamic）在计算流体动力学中很常见；它是带有一系列时间步的结构化网格。

每个点都要读写大约 75 个双精度浮点数，并执行大约 1300 个浮点操作。和 SpMV 类似，LBMHD 在单处理器上也只能达到峰值性能的很小一部分，因为 LBMHD 的数据结构复杂，且存储访问模式不规则。LBMHD 的算术密度为 0.70 FLOPS/字节，高于 SpMV 的 0.25。在发生缓存写缺失，且程序将重写整个缓存块时，不从存储器填充整个缓存块，那么可将算术密度提高到 1.07。除了 UltraSPARC T2（Niagara 2）之外的所有多核都提供这种缓存优化机制。

从图 7-19 中可以看到，除了 UltraSPARC T2 之外，LBMHD 的算术密度已经足够高了，计算优化和存储带宽优化都可以使用。UltraSPARC T2 的脊点低于 LBMHD，因此只能使用计算优化。

除了将代码并行化以充分利用多核之外，LBMHD 还可以选用下面的优化：

- *内存关联*：该优化也是有效的，原因同上。
- *TLB 缺失最小化*：为了大幅度降低 LBMHD 中的 TLB 缺失，可以使用数组结构将一些循环组合在一起，而不是像常规方法那样使用结构数组。

- 循环展开和重排序：为了尽可能地挖掘并行性并提高缓存利用率，需要将循环展开，并将使用邻近地址的语句重排序到同一组中。
- SIMD 化：这两种 x86 系统的编译器不能产生较好的 SSE 代码，因此必须使用汇编语言手工编写。

图 7-21 给出了 4 个系统在不同核数下的 LBMHD 性能。和 SpMV 一样，Intel Xeon e5345 的可扩展性最差。尽管 Opteron X4 只有 UltraSPARC T2 核数的一半，但是因为其单核性能超过 T2 的简单核，所以性能依然超过 T2。和 SpMV 相同，IBM Cell 仍是最快的系统。除了 Xeon e5345 之外所有系统性能都随核数按比例增加，尽管 T2 和 Cell 的增长比 Opteron X4 更加平缓。

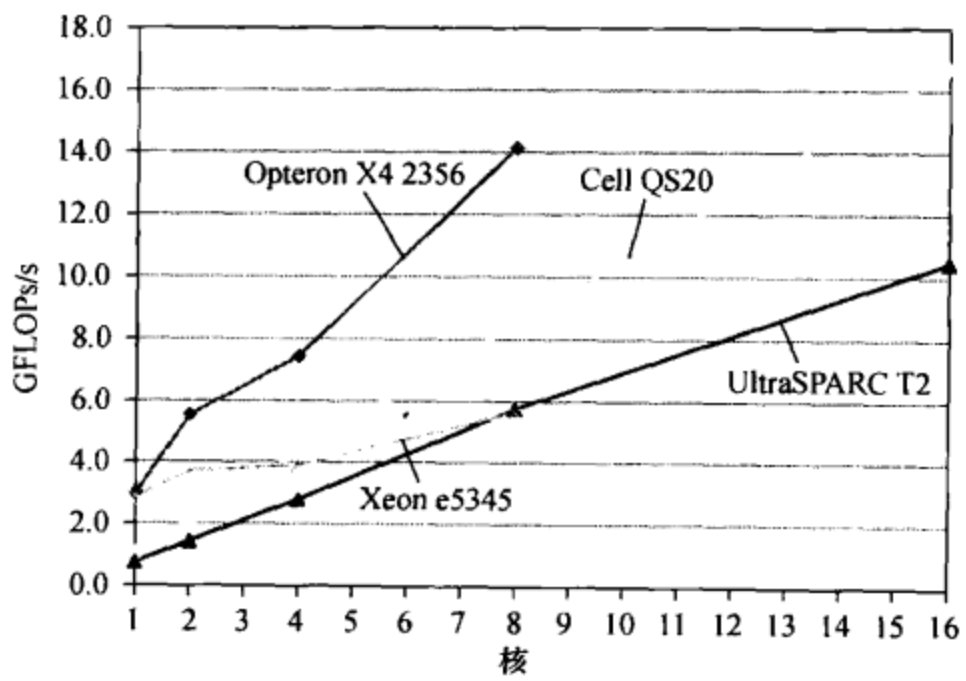


图 7-21　4 种多核上的 LBMHD 性能

7.11.4　生产率

除了性能之外，并行计算革新的另外一个重要问题是生产率，或者说是获得性能的编程难度。为了比较，图 7-22 给出四个系统在这两个核心上的最初性能和最优性能。

MPU 类型	核心	最初的性能 GFLOPs/s	优化后的性能 GFLOPs/s	最初性能相对优化后性能的百分比
Intel Xeon e5345（Clovertown）	SpMV LBMHD	1.0 4.6	1.5 5.6	64% 82%
AMD Opteron X4 2356（Barcelona）	SpMV LBMHD	1.4 7.1	3.6 14.1	38% 50%
Sun UltraSPARC T2（Niagara 2）	SpMV LBMHD	3.5 9.7	4.1 10.5	86% 93%
IBM Cell QS20	SpMV LBMHD	— —	6.4 16.7	0% 0%

图 7-22　4 个多核在两个核心上最初性能与全优化后性能的对比

注意，Sun UltraSPARC T2（Niagara 2）的最初性能达到全优化后性能的百分比很高。这里没有给出 IBM Cell 的基准性能，因为不能将代码移植到没有缓存的 SPES 上。如果你在 PowerPC 核上运行该代码，那么性能相对 SPES 将降低一个数量级，因此我们在本图中忽略了这种情况。

最容易的是 UltraSPARC T2，因为它的存储带宽很大，核理解起来也容易。对于这两个核心在 UltraSPARC T2 中的建议是应该从编译器获得执行有效的代码，并使用尽可能多的线程。对于

其他核心一个注意的地方是，想当然地认为 UltraSPARC T2 的组相关应与硬件线程数相匹配，事实上并非如此（见 5.11 节）。UltraSPARC T2 每个芯片支持 64 个硬件线程，但是二级缓存是四路组相连的。这种不匹配需要将循环语句重新组织以减少冲突导致的缓存缺失。

Xeon e5346 的优化非常困难，因为难以理解双前端总线的存储器行为；难以理解硬件预取是如何工作的；难以从编译器上获得高质量的 SIMD 代码。它和 Opteron X4 的 C 代码在内嵌 SIMD 指令之后会获得更高的性能。

Opteron X4 可受益于大多数类型的优化，因此相对于 Xeon e5345 它需要更多的努力，尽管它的存储器行为相对于 Xeon e5345 更容易理解。

Cell 提出了两类挑战。首先，SPE 所使用的 SIMD 指令是编译器所难以处理的，因此你需要帮助编译器将汇编语言指令嵌入到 C 代码中。第二，存储系统变得更加有趣。由于每个 SPE 都有使用独立地址空间的局部存储器，不能简单地移植代码并直接在 SPE 上运行。因此，在图 7-22 中 IBM Cell 没有给出基准性能，并且需要改变程序以发射 DMA 命令在局部存储器和主存之间来回传递数据。一个好消息是 DMA 在缓存中扮演软件预取指的角色，并且 DMA 易于使用并获得高的存储器性能。Cell 能够为这些核心提供存储带宽“屋顶线近”90% 的性能，而其他多核只能提供到 50% 甚至更少。

7.12　谬误与陷阱

十多年来，一直有人在争论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三个。

谬误：Amdahl 定律不适用于并行计算机。

在 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，因此必然有一个合算的处理器数量上界——比如说是 100。通过给出使用 1000 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，但是没有意识到当问题规模增大时算法是如何按比例改变的。

谬误：峰值性能可代表实际性能。

例如，7.11 节指出 Intel Xeon e5345 在四个微处理器中具有最高的峰值性能，但是在完成两个核心计算时反而是最慢的。

超级计算机业界在市场中使用该度量方法，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。

在很长的时间里软件一直落后于并行处理器，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！

在为单处理器设计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 操作系统最初通过一个锁来保护页表，假定页分配是不频繁的。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。

页表操作的串行化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。该性能瓶颈甚至在作业级并行中也存在。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。

7.13　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转折点。……这不是一场竞争。这是计算的翻天覆地的变化……”

——Paul Otellini，Intel 总裁，Intel 开发者论坛，2004 年

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就可以构建计算机。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。CD 中的 7.14 节讨论了这些多处理器的历史。

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**[⊖]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。不难想象无论是数据中心的数量还是每个数据中心内服务器的数量都会持续增长。毋庸置疑，此类数据中心会采用多核设计，因为数据中心已经在应用中使用了数以千计的处理器。
- 在科学计算和工程计算等领域中并行处理的使用是非常普遍的。此类应用领域对计算能

⊖ 软件即服务（software as a service）：软件不再是安装运行在客户自己的计算机上，而是运行在远程计算机上，通过 Internet 来使用，典型情况是通过 Web 接口为客户服务。然后根据使用情况向客户收费。

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 2007 Linpack 报告，集群在 500 台最快的计算机中占据了超过 80% 的份额。虽然如此，为这些应用编程也并非易事，如何进行并行处理器编程仍是一项挑战。此类应用也必然采用多核芯片，因为它们已经采用成百上千个处理器。

- 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。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。
- 相对于多芯片设计，同一芯片上的多处理器可提供完全不同的通信速率，具有更低的延迟和更高的带宽。这些改进可以使得高性能更易获得。
- 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，如果单线程性能随增加硅面积的开方增长，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。目前并行正在片上以多核的形式实现，我们可以使用已经获得成功的传统微处理器来获得亚线性的性能提升。
- 运行时编译技术的成功使得软件更容易适应处理器核数量的增长，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，在改变旧有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。

软硬件接口上的变革也许是近五十年来所面临的最大挑战。它在 IT 界内外提供了大量研究和商业机遇，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。也许你就会抓住其中的机会，成为创新者中的一员。

7.14　拓展阅读

本节在 CD 上主要给出了近 50 年来多处理器的发展历史。

7.15　练习题

由美国东北大学的 David Kaeli 提供。

习题 7.1

首先写一个每周你通常需要完成的日常活动的列表。例如，你可能会起床、淋浴、穿衣服、吃早饭、弄干头发、刷牙等。确保列表中至少包含 10 项活动。

7.1.1 [5] <7.2> 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齿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。

7.1.2 [5] <7.2> 接下来考虑哪些活动可以并发执行（例如吃早餐和听新闻）。对每个活动都分析哪些活动可以与其配对并发执行。

7.1.3 [5] <7.2> 对习题 7.1.2，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？

7.1.4 [5] <7.2> 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。

习题 7.2

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。

7.2.1 [10] <7.2> 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//找到
        }
        return -1//未找到
}
```

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。

7.2.2 [5] <7.2> 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？

习题 7.3

请看下面的 C 代码片段：

```
for (j=2;j<1000;j++)
    D[j]=D[j-1]+D[j-2];
```

与之对应的 MIPS 代码如下所示：

```
          DADDIU  r2,r2,999
loop:     L.D     f1, -16(f1)
          L.D     f2, -8(f1)
          ADD.D   f3, f1, f2
          S.D     f3, 0(r1)
          DADDIU  r1, r1, 8
          BNE     r1, r2, loop
```

每种指令的延迟如下（以周期为单位）：

ADD.D	L.D	S.D	DADDIU	BNE
3	5	1	1	3

7.3.1 [10] <7.2> 执行一次循环内所有的指令需要多少周期？

7.3.2 [10] <7.2> 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，我们会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。

7.3.3 [10] <7.2> 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 7.4 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 4 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。

7.3.4 [10] <7.2> 互连网络的延迟是决定消息传递系统效率的重要因素之一。请问为了让习题 7.3.3 中的分布式系统能获得任意加速比，互连网络需要提供多快的速度？

习题 7.4

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 m 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
```

7.4.1 [10] <7.2> 假设 MergeSort 运行在具有 Y 个核的多核处理器上，且 Y 远远小于 m（长度）。请问预期的加速比是多少？请画图表示。

7.4.2 [10] <7.2> 接下来，假设 Y 与 m（长度）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？

习题 7.5

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：

烤箱预热至 160℃（325℉）。在烤盘上抹黄油和一层薄薄的面粉。

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。

最后慢慢加入蓝莓，将蛋糕均匀地放在烤盘中，烘烤约 60 分钟。

7.5.1 [5] <7.2> 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。

7.5.2 [5] <7.2> 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？

7.5.3 [5] <7.2> 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。这些将对习题 7.5.1 中的计划有何改变？

7.5.4 [5] <7.2> 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。

习题 7.6

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，欲与一个 $n \times p$ 的矩阵 B 相乘。乘法结果为一个 $m \times p$ 的矩阵 AB。如果令 $C = AB$，c_{ij} 代表在位置 (i, j) 处 C 的值，则

$$c_{i,j} = \sum_{r=1}^{n} a_{i,r} b_{r,j} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j} + \cdots + a_{i,n} b_{n,j}$$

其中 $1 \leqslant i \leqslant m$ 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：$a_{1,1}, a_{2,1}, a_{3,1}, a_{4,1}, \cdots\cdots$

7.6.1 [10] <7.3> 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。

7.6.2 [10] <7.3> 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），重新计算习题 7.6.1 中的问题。

7.6.3 [10] <7.3> 有什么办法消除可能出现的伪共享问题？

习题 7.7

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 + 1;`

核 4：`z = x + y;`

7.7.1 [10] <7.3> 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。

7.7.2 [5] <7.3>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。

习题7.8

在CC-NUMA（cache-coherent nonuniform-memory access）共享存储器系统中，CPU和物理内存被划分到不同计算节点上。每个CPU有自己的局部cache。为了维护存储一致性，我们可为每个cache块增加状态位，或者引入存储目录。基于存储目录，每个节点使用一个专用硬件表来管理每个存储块的状态。目录的大小是与CC-NUMA共享空间大小相关的函数。为节点中本地存储器的每个块提供一个入口，如果将一致性信息存储在cache中，应当将其存储在每个系统的每个cache中（也就是说，存储空间的数量是所有cache中可用cache块数量的函数）。

在下面的问题中，假定所有节点具有相同数量的CPU和相同数量的存储器（也就是说，CPU和存储器在CC-NUMA机器中各个节点之间平均分配）。

7.8.1 [15] <7.3>假定在CC-NUMA系统中，P个CPU分布在T个节点上，每个CPU有C个存储块，每个cache块上维护1字节的一致性信息。请问为了维持一致性，系统中一个节点的cache中需要保存多少存储器内容？写出相应的计算公式。公式中不用考虑数据存储时实际的空间占用大小，只需考虑存储一致性信号所需要的空间即可。

7.8.2 [15] <7.3>假定CC-NUMA系统总的存储块为S，总的节点数为T。如果每个目录项都为每个CPU保持1个字节的信息，请给出可计算每个目录中需要多少存储空间的公式。

习题7.9

在习题7.8描述的CC-NUMA系统中，假定节点数量为4，每个节点只有1个CPU（每个CPU有自己的一级数据cache和二级数据cache）。一层数据cache是写直达的，而二级数据cache是写回的。假定系统当前的工作负荷是让每个CPU写入存储器的某地址，其他CPU都读取所写入的数据。还假定写入的地址在开始时仅存储在存储器中而不在任何一个局部cache中，在写之后更新的块只存在于执行写操作的处理器的一级和二级cache中。

7.9.1 [10] <7.3>对使用基于cache的块状态来维护一致性的系统，请描述以下情况下节点之间的通信情况：某个节点对一个地址进行写操作时；其他三个节点读取该地址的数据之后。

7.9.2 [10] <7.3>对基于目录的一致性机制，重新回答习题7.9.1中的问题。

7.9.3 [20] <7.3>假定每个CPU均为四核处理器，每个核都有一级数据cache，四核共享二级数据cache。一个核执行该写操作，其他15个核执行读操作。在这种情况下重新回答习题7.9.1和习题7.9.2中的问题。

7.9.4 [10] <7.3>在习题7.9.3的基础上，假定每个核对同一cache块中的两个不同字节进行写操作，请问这对总线上的通信有何影响？

习题7.10

在CC-NUMA系统中，对非局部存储器的访问开销限制了多处理的效率。下表给出了访问局部存储和非局部存储中数据的开销，以及我们的应用程序中局部存储访问所占的百分比。

局部存取（周期）	非局部存取（周期）	局部存取所占百分比
20	100	50

请回答下面的问题。假定存储访问在应用程序中是均匀分布的，从而我们在进行存储访问时继续进行计算（不存在真相关）。还假定在任何一个周期内仅执行一个存储器操作。请说明本地和非本地存储器操作排序的所有假设。

7.10.1 [10] <7.3>如果平均每隔75个周期访问一次存储器，请问对应用程序的影响如何？

7.10.2 [10] <7.3> 如果平均每隔 50 个周期访问一次存储器，请问对应用程序的影响如何？

7.10.3 [10] <7.3> 如果平均每隔 100 个周期访问一次存储器，请问对应用程序的影响如何？

习题 7.11

哲学家就餐问题是一个经典的同步和并发问题。该问题假设就座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。

7.11.1 [10] <7.4> 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？

7.11.2 [10] <7.4> 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。

7.11.3 [10] <7.4> 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）。

7.11.4 [10] <7.4> 如果我们让请求周期性地重试直到资源变为可用，这样是否就解决了习题 7.11.3 中的问题？请给出原因。

习题 7.12

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 指令支持指令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：需两个周期执行	B1：无相关性
A2：需要 A1 的结果	B2：与 B1 使用的一个功能单元冲突
A3：与 A2 使用的一个功能单元冲突	B3：无相关性
A4：需要 A2 的结果	B4：需要 B2 的结果

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。

7.12.1 [10] <7.5> 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？

7.12.2 [10] <7.5> 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？

7.12.3 [10] <7.5> 如果使用一个 SMT 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？

习题 7.13

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。

7.13.1 [30] <7.5>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。

7.13.2 [15] <7.5>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？

习题7.14

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;
```

7.14.1 [10] <7.6>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？

7.14.2 [20] <7.6>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。

习题7.15

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讯。

7.15.1 [10] <7.6>分析脉动阵列的各种实现机制（可以在互联网或出版物中查找相关资料），然后使用MISD模型对习题7.14中的循环进行编程，并对遇到的问题进行讨论。

7.15.2 [10] <7.6>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。

习题7.16

假定我们要执行本章讲述NVIDIA 8800 GTX GPU时提到的DAXP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SAXP）。假定指令的执行周期数如下所示。

Loads	Stores	Add.s	Mult.s
4	1	2	5

7.16.1 [20] <7.7>请问在一个八核处理器中如何构建warp来完成SAXP循环？

习题7.17

从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，并确认它们运行在仿真器上。

7.17.1 [90] <7.7>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。

7.17.2 [90] <7.7> 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 的计算时间，并解释其中的原因。

习题 7.18

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能够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。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 互连技术。

7.18.1 [25] <7.7> 比较这两种互连技术的带宽和延迟。

习题 7.19

参照图 7-9b 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。

7.19.1 [10] <7.8> 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接？请写出计算公式。

7.19.2 [10] <7.8> 比较 n 维立方体和全互连网络在节点数量相同时的可靠性。画图比较两种拓扑分别在多少连接损坏时导致连接失效。

习题 7.20

基准测试程序（benchmark）用于在指定的计算平台上运行代表性的工作负荷，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 7.11 节中描述的各个系统上。

7.20.1 [60] <7.9> 两种工作负载运行在这些多核系统上的本质区别是什么？

7.20.2 [60] <7.9，7.10> 使用 Roofline Model 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？

习题 7.21

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 Sparse Matrix Format。它使用 3 个一位数组存储维数 $m \times n$ 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。第三个数组 JA 包含 A 中每个元素的列号，因此它的长度也为 R。

7.21.1 [15] <7.9> 分析下面的稀疏矩阵 X，并编写 C 程序将其存储为 Yale Sparse Matrix Format。

```
Row 1[0,0,0,0,10]
Row 2[0,0,0,0,0]
Row 3[8,0,0,0,6]
Row 4[0,1,8,7,0]
```

```
Row 5[7,0,0,0,0]
```

7.21.2 [10] <7. 9 > 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。

7.21.3 [15] <7. 9 > 执行下面给出的矩阵 X 和矩阵 Y 的矩阵乘。

[9, 8, 7, 100, 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。

7.21.4 [15] <7. 9 > 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）？

习题 7.22

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速多核 CPU（具有浮点单元）。

CPU B——每周期可执行单条指令的快速单核整型 CPU（例如，无浮点单元）。

CPU C——每周期可执行同样指令的多个拷贝的慢速向量 CPU（具备浮点能力）。

假定我们的处理器在下面的频率运行：

CPU A	CPU B	CPU C
1.5 GHz	3 GHz	500 MHz

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 均可执行整型算术，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 1024 * 1024 的浮点元素。输出结果应是指示矩阵 X 中何处的值比矩阵 Y 中的值大的一系列数。

7.22.1 [10] <7. 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。

7.22.2 [10] <7. 11 > 你会向向量 CPU C 中增加哪类指令，以获得更好的性能？

习题 7.23

假定一个四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。同时假定，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。

Average transaction latency	Maximum transaction processing rate
1 ms	5000/sec
2 ms	5000/sec
1 ms	10 000/sec
2 ms	10 000/sec

（average transaction latency：平均事务延迟；Maximum transaction precessing rate：最大事务处理速率）

对于表中的每一对数据，回答如下问题：

7.23.1 [10] <7. 11 > 在任意给定的瞬间，平均有多少请求被处理？

7.23.2 [10] <7. 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

机每秒处理多少事务)?

7.23.3 [10] <7. 11 > 讨论为什么通过简单地增加核的数量，我们很少获得这种加速?

小测验参考答案:

7.1 节　错误。作业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。

7.2 节　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。

7.3 节　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。

7.4 节　1. 错误。发送和接收消息是隐含同步，和共享数据一样。2. 正确。

7.5 节　1. 正确。2. 正确。

7.6 节　正确。

7.7 节　错误。图形 DRAM DIMM 因其更高的带宽而被赞扬。

7.9 节　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，以赢得工业界在并行计算上所下的赌注。

图形和计算 GPU

John Nickolls
NVIDIA 体系结构总监

David Kirk
NVIDIA 首席科学家

想象力比知识更重要。

——阿尔伯特·爱因斯坦，《on Science》，1930

A.1 引言

本附录主要讨论 **GPU**[⊖]——一种普遍存在于 PC、笔记本、桌面计算机和工作站中的**图形处理单元**。在它的大多基本形式中，GPU 产生 2D 和 3D 的图形、图像和视频，以支持基于窗口的操作系统、图形用户界面、视频游戏、可视化图像应用和视频播放。我们在此描述的现代 GPU 是为**可视化计算**[⊜]而优化的高度并行、多线程处理器。为了通过图形、图像和视频，提供与计算模型的实时、可视交互能力，GPU 具有一个统一的图形计算结构，同时也是可编程的图形处理器和标量并行处理平台。个人计算机和游戏主机将 GPU 与 CPU 组合形成一个**异构系统**[⊝]。

A.1.1 GPU 发展简史

大约 15 年前，GPU 还没有出现，PC 上的图形操作由视频图形阵列（VGA）控制器完成。简单地说，VGA 控制器由连接到一定容量 DRAM 上的存储控制器和显示产生器构成。在 20 世纪 90 年代，半导体技术获得了充分的发展；更多的功能可以加入到 VGA 控制器中。到 1997 年，VGA 控制器开始具有一些三维（3D）加速功能，包括用于三角形生成、光栅化（将三角形切割成单独的像素）、纹理贴图和阴影（为像素使用贴纸或图案并混合颜色）。

在 2000 年，一个单片图形处理器集成了传统高端工作站图形流水线的几乎每一个细节，因此，应当在 VGA 控制器之外取一个新的名字。术语 GPU 用来表示图形设备已经变成了一个处理器。

随着时间的推移，GPU 的可编程性愈发强大，其作为可编程处理器取代了固定功能的专用逻辑，同时保持了基本的 3D 图形流水线组织。另外，随着时间的推移，计算变得更为精确，从索引算术、整型和定点发展到单精度浮点，再到近来的双精度浮点。GPU 已经变成具有上百个核和上千个线程的大规模并行可编程处理器。

近来，GPU 增加了处理器指令和存储器硬件，以支持通用编程语言，并且创立了一种编程环境，以允许使用熟悉的语言（包括 C/C++）对 GPU 进行编程。这种革新使 GPU 成为了一个

⊖ 图形处理单元（graphics processing unit，GPU）：一种面向 2D 和 3D 图形、视频、可视化计算和显示优化的处理器。

⊜ 可视化计算（visual computing）：图形处理和计算的混合体，使得用户可以通过图形、图像和视频可视化与计算对象进行交互。

⊝ 异构系统（heterogeneous system）：由不同类型处理器组成的系统，如 PC 是 CPU 和 GPU 组成的异构系统。

完全通用的、可编程的多核处理器，具备一些独特的优势，也有一定的局限性。

GPU 的趋势

GPU 及其相关驱动实现了图形处理中的 OpenGL 和 DirectX 模型。OpenGL 是一个大多数计算机使用的用于 3D 图形编程的开放标准。DirectX 是一系列微软多媒体编程接口，包括用于 3D 图形的 Direct3D。由于这些**应用程序接口**[⊖]具有定义明确的行为，通过这些 API，为图形处理功能构建高效的硬件加速器成为可能。这是每隔 12 ~ 18 个月就推能出一款使现有应用程序执行性能翻倍的新 GPU 的原因之一（除了增加器件密度之外）。

GPU 性能经常性地成倍提升使得过去不可能实现的新应用成为可能。图形处理和并行计算的交叉催生了图形的一种新范例，就是我们所知道的可视化计算。它以几何学可编程元素、顶点和像素例程取代了大部分传统的顺序硬件图形流水线模型。现代 GPU 中的可视化计算以一种革新的方式组合了图形处理和并行计算，允许实现新的图形算法，并打开了通向在普适高性能 GPU 上进行全新的并行处理应用的大门。

A.1.2　异构系统

尽管论证表明 GPU 是典型 PC 中具有最多并行性和最强大性能的处理器，但它的确不是唯一的处理器。现在是多核，很快将会是众核的 CPU 作为补充，将是主要的串行处理器，是与大规模并行众核 GPU 共同工作的处理器。这两种类型的处理器一起组成了异构微处理器系统。

众多应用程序的最好性能来源于同时使用 CPU 和 GPU。本附录将帮助你理解如何及何时在这两个并行度都在增加的处理器中最好地划分工作。

A.1.3　GPU 发展成了可扩展的并行处理器

GPU 的功能已经从硬连线、能力有限的 VGA 控制器发展为可编程的并行处理器，从逻辑（基于 API）图形流水线发展到混合可编程元素。最终，GPU 使得将异构的可编程流水线元素融合成一个统一的众核可编程处理器阵列具有了意义。

在 GeForce 8 系列 GPU 中，几何、顶点和像素处理均在相同类型的处理器上运行。这种统一带来了显著的可扩展性。更多的可编程处理器核心增加了系统的总吞吐量。统一的处理器在负载平衡方面也非常有效，因为任何处理功能均可使用整个处理器阵列。另一方面，现在可以使用非常少的处理器来构建处理器阵列，因为所有的功能都能在同样的处理器上运行。

A.1.4　为什么使用 CUDA 和 GPU 计算

这个统一、可扩展的处理器阵列使得 GPU 产生了一个新的可编程模型。GPU 处理器阵列所拥有的强大浮点处理能力在解决非图形问题方面具有很大的吸引力。处理器阵列在图形处理方面所具有的强大并行性和可扩展性，使得为通用计算而使用的编程模型可以直接表达大规模并行性，并且允许扩展执行。

GPU 计算[⊜]是通过并行编程语言和 API 使用 GPU，而不是用传统的图形 API 和图形流水线模型，进行计算而产生的术语。这是为了与早前使用图形 API 和图形流水线进行非图形任务的**GPU 上的通用计算**[⊝]方法进行区别。

⊖ 应用程序接口（Application Programming Interface，API）：函数和数据结构定义的集合，为函数库提供一个接口。

⊜ GPU 计算（GPU computing）：通过并行编程语言和 API 使用 GPU 进行计算。

⊝ GPU 上的通用计算（General Purpose computation *on* GPU，GPGPU）：通过传统的图形 API 和图形流水线进行通用目的的计算。

统一计算设备架构[⊖]是 GPU 和其他并行处理器的一个可扩展的编程模型和软件平台，允许程序员跳过 GPU 中的图形 API 和图形接口，仅用 C 或 C ++ 进行编程。CUDA 编程模型具有 SPMD（单程序多数据）软件特征，即程序员为单个线程编写的程序，在 GPU 的多核中由众多线程实例化然后并行执行。实际上，CUDA 也为多 CPU 核提供了一个编程工具，因此 CUDA 是一个为整个异构计算机系统编写并行程序的环境。

A. 1. 5　GPU 统一了图形和计算

随着 GPU 性能的增加又产生了 CUDA 和 GPU 计算，这使得用 GPU 同时作为图形处理器和计算处理器，并在可视化计算应用中组合这些用途成为可能。GPU 内在的处理器架构在两个地方体现出来：首先，可编程的图形 API 的实现；其次，在 CUDA 模型下使用 C/C ++ 语言可编程的大规模并行处理器。

尽管 GPU 的内部处理器是统一的，但让所有的 SPMD 线程程序都相同却没有必要。GPU 可在 GPU 图形方面运行图形阴影程序，处理几何、顶点和像素，同时也在 CUDA 中运行线程程序。

GPU 是一个真正的通用多处理器结构，支持大量的处理任务。GPU 在图形和可视化计算方面是性能卓越的，因为它专门面向这些应用设计。GPU 在众多与图形处理类似的通用应用上也是性能卓越的，因为这些应用执行大量的并行操作，并具有规整的问题结构。一般而言，它们与数据并行问题非常匹配（参见第 7 章），特别是大数据量的问题，但在规则性不强、数据量较小的问题方面表现较差。

A. 1. 6　GPU 可视化计算的应用

可视化计算包括了传统的图形应用，并加入了许多新的应用。最初 GPU 的应用范围仅限于做“和像素有关”的事情，而现在包括了很多与像素无关，却有着规则计算和数据结构的问题。GPU 在 2D 和 3D 图形处理方面是高效的，因为 GPU 正是为此设计的。在该应用性能方面的失败将是致命的。2D 和 3D 图形以“图形模式”使用 GPU，通过图形 API、OpenGL™和 DireceX™使用 GPU 强大的处理能力。游戏是基于 3D 图形处理能力构建的。

除了 2D 和 3D 图形，图像处理和视频对于 GPU 来说也是重要的应用。这些可以在计算模式下，使用 CUDA 对 GPU 进行编程，使用图形 API 或计算程序来实现。使用 CUDA，图像处理仅仅是另一种数据并行阵列程序。对于数据访问规则且有良好局部性的应用来说，程序将极为高效。事实上，图像处理对于 GPU 来说是非常好的应用。视频处理，尤其是编解码（根据一些标准算法进行压缩和解压缩）是非常有效的。

GPU 上可视化计算应用的最大特点在于“打破了图形流水线”。尽管先前的 GPU 具有非常高的性能，但仅实现了特定的图形 API。如果 API 支持你想要的操作，将会是绝妙的；如果不支持，GPU 就不能加速你的任务，因为早期的 GPU 功能是不可变的。现在，随着 GPU 计算和 CUDA 的出现，仅通过编写 CUDA 程序描述所需的计算和数据流，就可以对这些 GPU 编程以实现不同的虚拟流水线。因此，所有的应用现在都是可能的，这将刺激新的可视计算方法。

A. 2　GPU 系统架构

在本节中，我们概述当前普遍使用 GPU 的系统结构。我们讨论系统配置、GPU 功能和服务、标准编程接口和基本的 GPU 内部结构。

⊖ 统一计算设备架构（Compute Unified Device Architecture，CUDA）：一个基于 C/C ++ 语言的可扩展并行编程模型。它是面向 GPU 和多核 CPU 的一个并行编程平台。

A.2.1　异构 CPU-GPU 系统架构

一个使用 GPU 和 CPU 的异构计算机系统架构，可以在较高的层次上使用两种主要的特征进行描述：首先，使用了多少功能子系统或芯片，它们的互联技术和拓扑是什么；其次，什么存储子系统对这些功能子系统是可用的。PC 的 I/O 系统和芯片组的背景知识可以参考第 6 章。

历史上的 PC（1990 年前后）

图 A-2-1 是 1990 年前后遗留下来 PC 机的高层模块图。北桥（参看第 6 章）包含了高带宽接口，将 CPU、存储器和 PCI 总线连接起来。南桥包含了早期接口和设备：ISA 总线（音频、局域网）、中断控制器，DMA 控制器，定时器/计数器。在这个系统中，显示器由连接到 PCI 总线上的一个简单帧缓存子系统（我们所知的 VGA，视频图形阵列）驱动。具有内建处理元素的图形子系统（GPU）在 1990 年的 PC 环境中是不存在的。

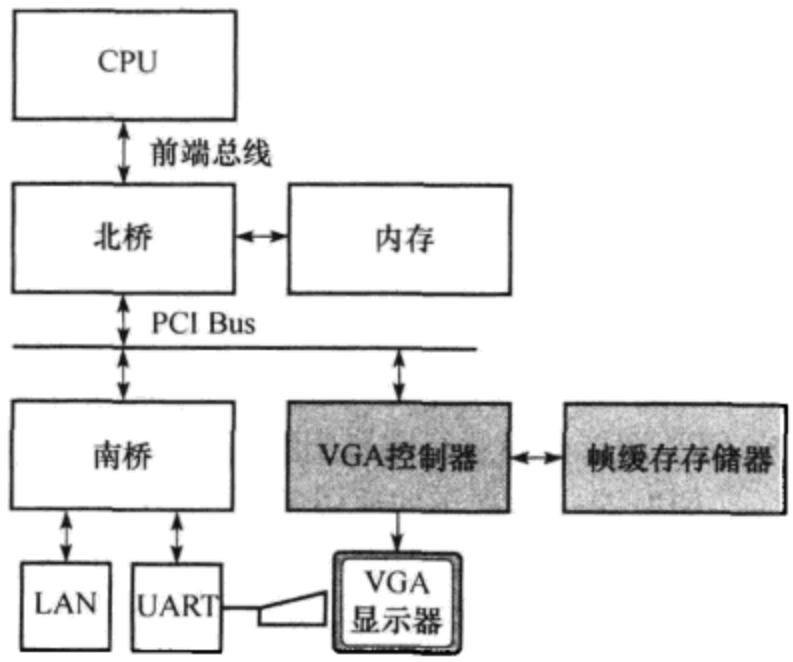


图 A-2-1　历史上的 PC
VGA 控制器从帧缓存存储器驱动图形显示器。

图 A-2-2 给出了目前普遍使用的两种配置。它们的特征为具有独立 GPU（离散 GPU）和 CPU，并有各自的存储子系统。在图 A-2-2a 中，我们可以看到，GPU 通过 16 通道的 **PCI-Express**[⊖] 2.0 插槽与 Intel CPU 连在一起，可提供 16 GB/s 传输速率（每个方向各为 8 GB/s）。与此类似，在图 A-2-2b 中，GPU 首先连接到芯片组，然后再通过具有同样可用带宽的 PCI-Express 与 AMD 的 CPU 相连。在上面的两种情况下，GPU 和 CPU 均可访问对方的存储器，尽管带宽比访问它们直接连接的存储器要小。在上面 AMD 的情形中，北桥或内存控制器与 CPU 集成到了同一个芯片中。

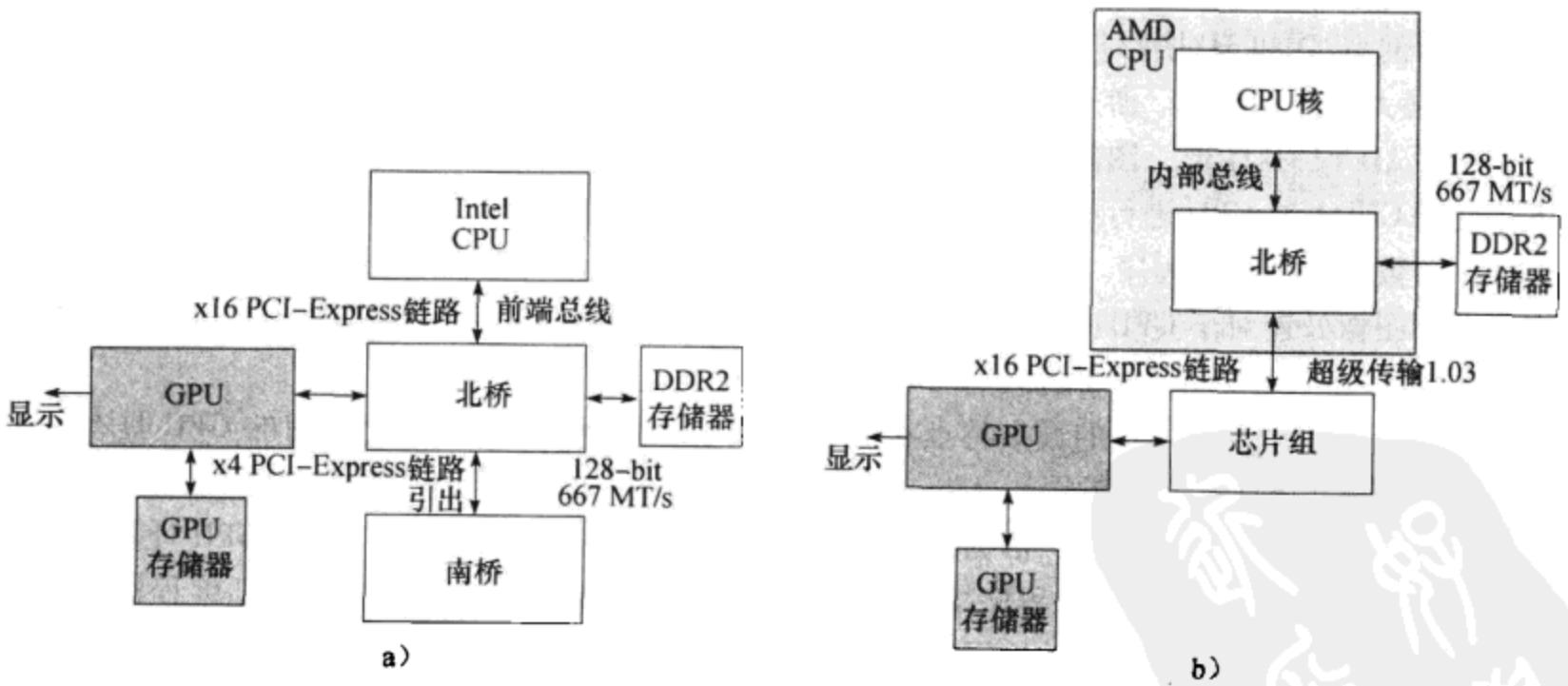


图 A-2-2　具有 Intel 和 AMD CPU 的当代 PC
图中组件和互联的解释可以参考第 6 章。

这些系统的一种低成本变种是**统一存储结构**[⊜]系统，即仅使用 CPU 存储器，而省略 GPU 存

⊖ PCI-Express（PCIe）：一种使用点对点链路的系统标准 I/O 互连，其链路具有可配置的槽数和带宽。
⊜ 统一存储结构（unified memory architecture，UMA）：一种 CPU 和 GPU 共享系统存储器的体系结构。

储器。这些系统使用性能相对较低的 GPU，因为它们可获得的性能受限于可用系统存储带宽和增加的存储器访问延迟，而专用的 GPU 存储器具有高带宽和低延时。

一个高性能变种是使用多个相连的 GPU（典型的是使用 2 ~ 4 个）并行工作，使用菊花链将它们组织起来。该系统的一个实例是 NVIDIA SLI（可扩展链路交互）多 GPU 系统，主要面向高性能游戏和工作站设计。

下一类系统是将 GPU 和北桥集成（Intel）或将 GPU 和具有或没有专用的存储器的芯片组（AMD）集成。

第 5 章解释了在共享地址空间中，cache 怎样维持一致性。有了 CPU 和 GPU，就有了多个地址空间。GPU 可以访问它们自己的物理局部存储器，并通过 GPU 中 MMU 转换出的虚拟地址对 CPU 系统中物理存储器进行访问。操作系统内核管理 GPU 的页表。系统物理页可以使用连贯的或非连贯的 PCI-Express 事务进行访问，由 GPU 页表中的一个属性决定。CPU 可以通过 PCI Express 地址空间的一个地址范围（也称为缝隙）访问 GPU 的局部存储器。

游戏控制机

游戏控制机系统，如 Sony 的 PlayStation 3 和 Microsoft 的 Xbox 360 类似于先前描述的 PC 系统结构。控制机系统设计的目标是，产品出售时保证在五年或更长的生命周期中，都具有同样的性能和功能。在这期间，一个系统或许被重新实现很多次，以采用更先进的硅片制造工艺，在保持性能不变的情况下降低成本。控制系统不需要使其子系统像 PC 系统那样不断地膨胀和升级，因此主要的内部系统总线趋向于自定义而不是标准化。

A.2.2　GPU 接口和驱动

今天的 PC 中，GPU 通过 PCI-Express 连接到 CPU。先前的 PC 代系使用 **AGP**[⊖]。图形应用调用 OpenGL［Segal 和 Akeley，2006］或 Direct3D［Microsoft DirectX Specification］API 功能，将 GPU 作为协处理器使用。API 通过面向特殊 GPU 优化的图形设备驱动向 GPU 发送命令、程序和数据。

A.2.3　图形逻辑流水线

在 A.3 节将对图形逻辑流水线进行描述。图 A-2-3 说明了主要处理阶段，并对重要的可编程阶段用灰色框表示（顶点、几何和像素渲染阶段）。

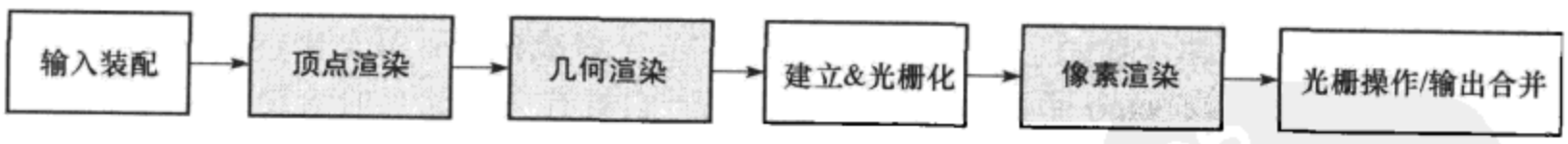


图 A-2-3　图形逻辑流水线

可编程图形渲染阶段用灰色表示，固定功能块用白色表示。

A.2.4　将图形流水线映射到统一的 GPU 处理器

图 A-2-4 显示了如何将由分立、独立的可编程阶段组成的逻辑流水线映射到处理器的物理分布的阵列上。

⊖ AGP：一个 PCI I/O 总线的扩展版本，为一个单一的卡槽提供高达 8 倍于原始 PCI 总线的带宽。它的主要目的是将图形子系统连接到 PC 系统。

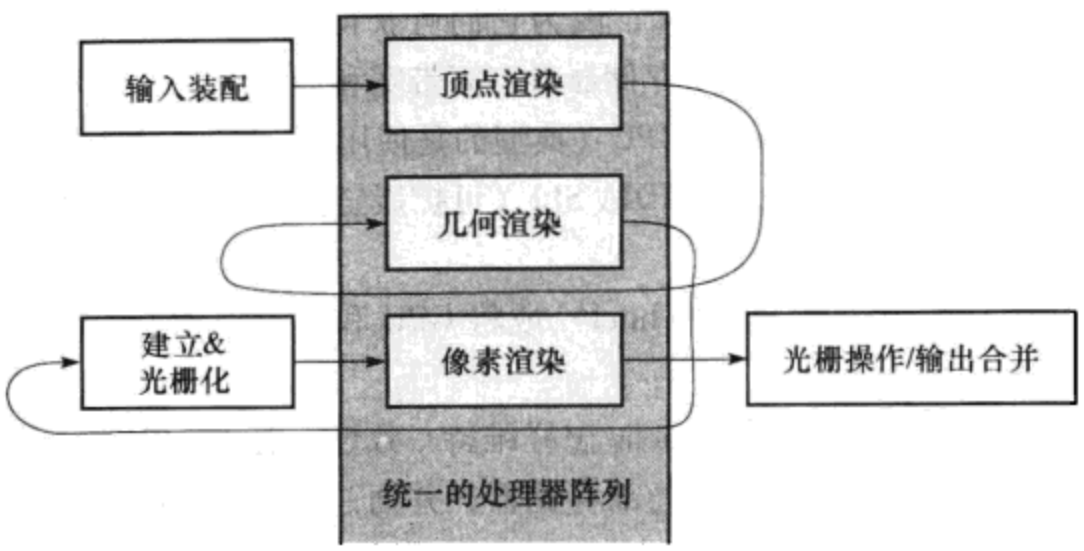


图 A-2-4　逻辑流水线映射到物理处理器

可编程的渲染阶段在统一的处理器阵列上执行，逻辑图形流水线的数据流通过处理器不断循环。

A.2.5　基本的统一 GPU 结构

统一 GPU 结构是以多个可编程处理器组成的并行阵列为基础的。它们在同样的处理器上统一了顶点、几何、像素渲染处理和并行计算，而不像早期的 GPU，对于每种处理类型有专用的分立处理器。可编程处理器阵列与用于纹理滤波、光栅化、光栅操作、别名消除、压缩、解压、显示、视频解码和高清视频处理的固定功能处理器紧密集成。尽管固定功能处理器在由面积、成本和功耗开销约束的绝对性能方面要明显优于更为通用的可编程处理器，我们在此仍以可编程处理器为重点。

与多核 CPU 相比，众核 GPU 具有不同的结构设计出发点，它着力于在众多的处理器核上有效地执行众多的并行线程。通过使用众多的简单核，并在线程组间对数据并行进行优化，提高了单片晶体管用于计算的比例，并降低了片上 cache 和开销。

处理器阵列

一个统一 GPU 处理器阵列包含众多的处理器核，典型地使用多线程多处理的方式组织。图A-2-5展示了一个 GPU，它具有 112 个流处理器（SP）核阵列，并组织成 14 个多线程流处理多核处理器（SM）。每个 SP 核是高度多线程的，并通过硬件管理 96 个并发线程和它们的状态。这些处理器通过内部互联网络与 4 个 64 位宽的 DRAM 分区相连。每个 SM 具有 8 个 SP 核，两个专用功能单元（SFU），指令和常量 cache，一个多线程指令单元和一个共享存储器。这是由 NVIDIA GeForce 8800 实现的基本 Tesla 结构。它具有一个统一的结构，在该结构下，传统的对于顶点、几何和像素渲染的图形应用可以在统一的 SM 和它们的 SP 核上运行，且计算程序在同样的处理器上运行。

通过缩放多处理器的个数和存储器分区的数量，处理器阵列结构的规模可以缩放到较小或更大的 GPU 配置上。图 A-2-5 展示了具有 2 个 SM 的 7 个簇，共享纹理单元和纹理 L1 cache。纹理单元将过滤后的结果传递给 SM，并以纹理图的形式给出坐标集合。因为对于连续的纹理请求来说，支持的过滤区域常常是重叠的，一个小的流化的 L1 纹理 cache 对减少存储器系统请求是有效的。处理器阵列通过一个 GPU 宽度的内部互联网络与光栅操作处理器（ROP）、L2 纹理 cache、外部 DRAM 存储器和系统存储器相连。处理器数量和存储器数量可以根据不同的性能和市场区域进行缩放，以设计均衡的 GPU 系统。

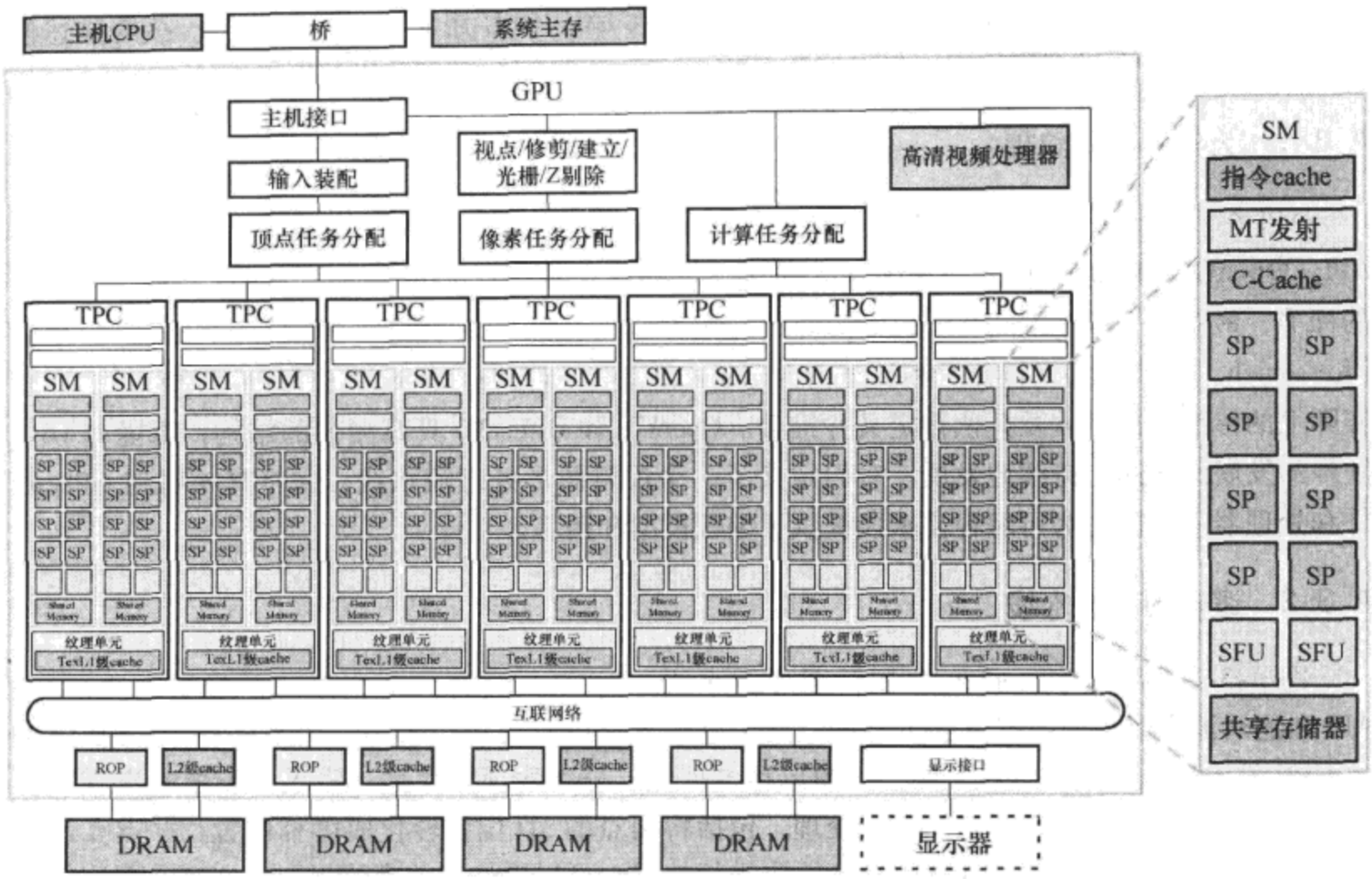


图 A-2-5　基本的统一 GPU 结构

示例 GPU 具有组织成 14 个流多核的 112 个流处理器核；这些核是高度多线程的。该 GPU 具有 NVIDIA GeForce 8800 的基本 Tesla 结构。处理器通过内部互联网络连到 4 个 64 位宽的 DRAM 分区。每个 SM 具有 8 个 SP 核，两个特殊功能单元，指令和常量 cache，一个多线程指令单元和一个共享存储器。

A.3　可编程 GPU

可编程的多处理 GPU 和可编程的其他多处理器，如多核 CPU，在本质上是不同的。GPU 提供的线程和数据并行是 CPU 的 2～3 个数量级，规模在上百个处理器核和数万并发线程（在 2008 年）。集成电路密度的增加和体系结构效率的提高，使得 GPU 的并行度得到了持续的增加，每 12～18 个月翻一倍。为了在不同的市场区域中覆盖较广的价格和性能范围，不同的 GPU 实现在处理器和线程数量方面有很大的变动。然而用户期望游戏、图形、图像和计算应用在 GPU 上运行，而不用考虑它运行了多少并发线程和它具有多少并发的核，用户还希望越贵的 GPU（具有更多线程和核）运行应用时越快。因此，GPU 编程模型和应用程序设计以透明扩展的方式支持到大范围的并行。

实时图形性能是 GPU 采用大量并行线程和核的驱动力，例如在高分辨率下需要以至少 60 帧/秒的速率渲染复杂 3D 场景。相应地，设计了图形渲染语言的可扩展编程模型，如 Cg（C for graphics）和 HLSL（high-level shading language），通过众多独立并行线程并扩展到任意数量的处理器核上，开发更大的并行度。CUDA 的可扩展并行编程模型，同样使得通用并行计算应用可利用大量的并行线程，而且可以对应用透明的方式扩展到任意数量的并行处理器核上。

在这些可扩展的编程模型中，程序员为单线程编写代码，而 GPU 则以并行方式运行无数的线程实例。这样，程序透明地扩展到很大范围的硬件并行。这个简单的范例是由描述如何渲染一个顶点或像素的图形 API 和渲染语言引发的。自 20 世纪 90 年代后期以来，这个一直作为 GPU 快速提高其并行度和性能的一个范例。

本节简要描述使用图形 API 和可编程语言，将可编程 GPU 用于实时图形应用。接着描述使用 C 语言和 CUDA 可编程语言，将可编程 GPU 用于可视计算和通用并行计算应用。

A.3.1　为实时图形编程

API 在 GPU 和处理器的快速、成功开发方面扮演着重要的角色。有两种主要的标准图形 API：**OpenGL**[⊖]和 **Direct3D**[⊜]（Microsoft DirectX 多媒体编程接口中的一个）。OpenGL 是一个开放的标准，最初由 Silicon Graphics Incorporated 提出并定义。OpenGL 标准［Segal 和 Akeley，2006］，［Kessenich，2006］的现行开发和扩充由 Khronos 和工业界管理。Direct3D［Blythe，2006］，一个实际上的标准，由微软及其伙伴定义并推动。OpenGL 和 Direct3D 具有相似的结构，都随着 GPU 硬件的发展而快速、持续地发展。它们定义了一个逻辑图形处理流水线，并将其映射到 GPU 硬件和处理器上，同时为可编程流水段提供了编程模型和语言。

A.3.2　逻辑图形流水线

图 A-3-1 展示了 Direct3D 10 逻辑图形流水线。OpenGL 具有一个相似的图形流水线结构。API 和逻辑流水线为流数据流提供基础，并为可编程渲染阶段提供管道，用黑体标出。3D 应用向 GPU 发出以几何原语（点、线、三角形和多边形）分组的顶点序列。输入装配器收集顶点和原语。顶点渲染程序对每个顶点进行处理，包括将顶点的 3D 位置转化成屏幕位置，并点亮顶点以决定它的颜色。几何渲染程序对每个原语进行处理，可以增加和放弃原语。建立和光栅化单元建立由几何原语覆盖的像素段（段是像素的潜在属性）。像素渲染程序对每个段进行处理，为每段添加参数、纹理和颜色。像素渲染大量使用采样和过滤，使用插值浮点坐标在称为**纹理**[⊜]的 1D、2D 或 3D 数组中进行查找。渲染为映射、功能、贴花、图像和数据进行纹理访问。光栅操作处理阶段执行 Z-buffer 深度测试和模板测试，在这两种测试中，可以丢弃隐藏像素段或用段的深度取代像素深度，并执行颜色合成操作，将段颜色和像素颜色进行组合并将合成后的颜色写入像素点。

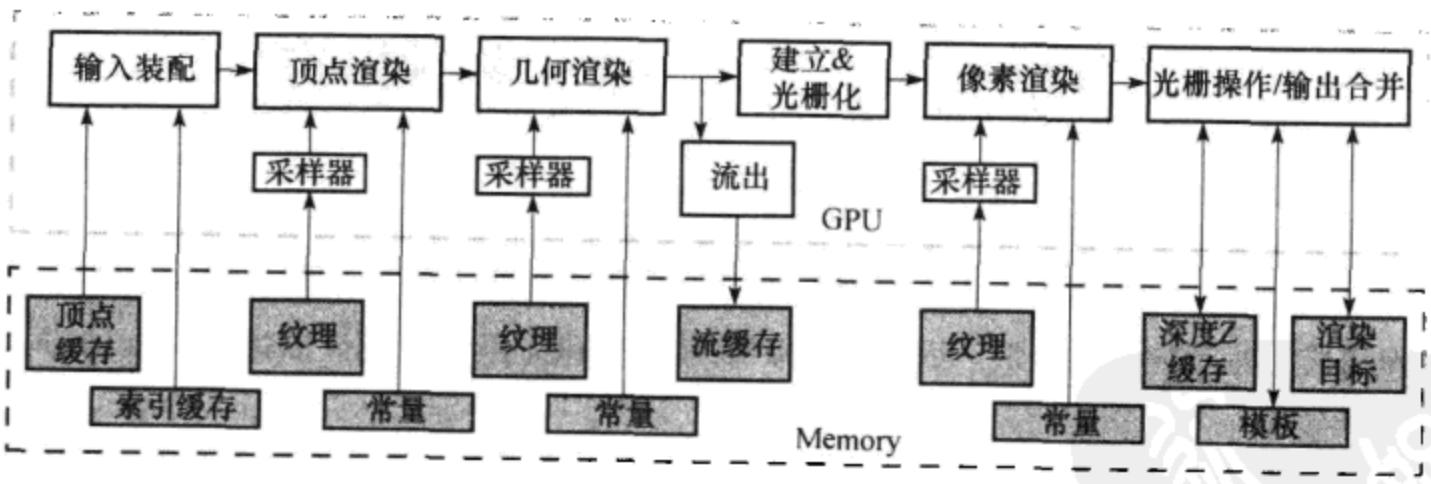


图 A-3-1　Direct3D 10 图形流水线

每一个逻辑流水线对应于 GPU 硬件或者 GPU 处理器。可编程渲染阶段由深灰色表示，固定功能块由白色表示，存储用灰色表示。每级以流数据流模式处理一个顶点、几何原语或像素。

图形 API 和图形流水线为对每个顶点、原语和像素段进行处理的渲染程序提供输入、输出、存储对象和基础。

⊖ OpenGL：一个开放的标准图形 API。

⊜ Direct3D：由微软及其合作伙伴定义的一个图形 API。

⊜ 纹理（texture）：支持使用插值坐标进行采样和过滤的一个 1D、2D 或 3D 的数组。

A.3.3　图形渲染程序

实时图形应用使用很多不同的**渲染程序**[⊖]对灯光与不同材质的相互影响进行建模，并渲染（render）复杂的灯光和阴影。**渲染语言**[⊖]基于逻辑图形流水线相应的数据流或流编程模型。顶点渲染程序将三角顶点的位置映射到屏幕上，变换它们的位置、颜色或方位。典型地，顶点渲染线程接收一个浮点类型的（x，y，z，w）顶点位置，计算后得到一个浮点类型的（x，y，z）屏幕位置。几何渲染程序对由多个顶点定义的几何原语（如线和三角形）进行操作，对它们进行变换或产生额外的原语。像素段渲染对每个像素进行“渲染”，计算一个浮点的红、绿、蓝和 alpha（RGBA）颜色，用于在它的像素采样（x，y）图像位置显示图像。在屏幕上显示复杂的光、阴影和高动态范围时，会碰到极端范围的像素贡献值计算，渲染（和 GPU）使用浮点算数计算像素的颜色，以消除可见的锯齿。对于全部这三种图形渲染，由于每一个都使用独立的数据，产生独立的结果，并且没有副作用，因而很多程序实例可以作为独立的并行线程并行地运行。独立的顶点、原语和像素进一步使得在不同规模的 GPU 上运行的相同的图形程序可以并行地处理不同数量的顶点、原语和像素。图形程序因而透明地扩展到了具有不同并行度和性能的 GPU 上。

所有这三种逻辑图形线程的用户程序具有一个公共的目标高层语言。通常使用 HLSL（high-level shading language）和 Cg（C for graphics）。它们具有和 C 语言类似的语法和一系列丰富的用于矩阵操作、三角学、插值、纹理访问和过滤的库函数，但它们远非通常的计算语言：当下它们缺乏常规的存储器访问、指针、文件 I/O 和递归。HLSL 和 Cg 假定程序存活于一个逻辑图形流水线中，因此 I/O 是隐含的。例如，一个像素段渲染器等待几何法线和多个纹理坐标（该法线和坐标由上游的固定功能段使用顶点值插值产生），并为颜色的输出参数赋值，将其传送到下游，与一个隐含（x，y）位置的像素混合。

GPU 硬件为每个顶点、每个原语和每个像素段，产生一个新的独立线程执行顶点、几何或像素渲染程序。在视频游戏中，大量的线程执行像素渲染程序，因为视频游戏中，有比顶点多 10～20 倍或者更多的像素段、复杂的灯光和阴影，这要求顶点渲染线程中具有更大的像素比例。图形渲染程序模型驱使 GPU 体系结构在众多并行处理器核上高效地执行数千个独立细粒度线程。

A.3.4　像素渲染示例

考虑下面实现“环境映射”展示渲染技术的 Cg 像素渲染程序。对于每个像素线程，这个渲染程序传递 5 个参数，包括采样表面颜色时需要的 2D 浮点纹理图像坐标和一个反映离开表面的视点方向的 3D 浮点向量。另外三个“统一”参数从一个像素实例（线程）到另一个不发生变化。渲染程序在两幅纹理图像中查找颜色：2D 纹理，用来进行表面颜色访问；3D 纹理，访问立方体映射（对应于立方体六个面的六幅图像）以获取对应反射方向的外部世界颜色。然后，最终的四种成分（红、绿、蓝和 alpha）浮点颜色由称为“lerp”的加权平均或线性插值函数计算得出。

```
    Void reflection(
    float 2                texCoord      :TEXCOORD0,
    float 3                reflection_dir:TEXCOORD1,
    out float4             color         :COLOR,
    uniform float              shiny,
    uniform sampler2D          surfaceMap,
    uniform samplerCUBE envMap)
{
```

⊖ 渲染程序（shader programme）：对图形数据，如顶点或像素段，进行操作的一个程序。

⊖ 渲染语言（shading language）：一种图形描述语言，通常具有一个数据流或流编程模型。

```
// 从纹理中提取表面颜色
    float4 surfaceColor = tex2D(surfaceMap,texCoord);

// 通过采样立方体映射获取反射颜色
    float4 reflectedColor = texCUBE(environmentMap,reflection_dir);

// 输出是这两个颜色的加权平均
    color = lerp(surfaceColor,reflectedColor,shiny);
}
```

尽管这个渲染程序只有 3 行的长度，但它激活了很多 GPU 硬件。对每一个纹理的取操作，GPU 纹理子系统进行多次存储器访问，以采样采样坐标附近的图像颜色，然后使用浮点过滤算法插值产生最终结果。多线程 GPU 并行执行数千计的这种轻量级的 Cg 像素渲染线程，深度交叉执行以掩盖纹理取操作和存储器延时。



图 A-3-2　GPU 渲染图像

为给出皮肤的可视深度和半透明效果，像素渲染程序建模了三个分立的皮肤层，每个都具有单独的表面散射行为。它执行 1400 条指令以渲染每个皮肤像素段的红、绿、蓝和 alpha 颜色成分。

Cg 使程序员的目光聚集到一个单一顶点、原语或像素，这些在 GPU 中作为一个单一的线程实现；渲染程序透明地扩展到可用的处理器上以开发线程的并行度。作为面向具体应用的语言，Cg 提供了一系列丰富的数据类型、函数库和语言结构以表达不同的渲染技术。

图 A-3-2 展示了使用分段像素渲染渲染过的皮肤。由于在重新显现之前周围光照的多次反射，真实的皮肤与油画表现出很大的不同。在这个复杂的渲染中，建模了三个分立的皮肤层，每个都有唯一的表面散射行为，以使皮肤具有可视的深度和半透明效果。散射可以通过在一个平面纹理空间中，使用比绿色多的红色模糊、比绿色少的蓝色模糊的模糊卷积进行建模。经过编译的 Cg 渲染执行 1400 条指令以计算一个皮肤像素的颜色。

随着 GPU 具有超高的浮点性能和很高的流存储带宽用于实时图形运算，除传统的图形应用外，它们还引起了其他具有很高并行性的应用的注意。首先，仅在将一个应用表达成图形渲染算法时才可以使用这种强大的能力，但这种 GPGPU 方法常常是难用并且是具有局限的。现在，CUDA 编程模型提供了使用 C 编程语言开发 GPU 可扩展高性能浮点和存储带宽的一个更为简单的方法。

A. 3. 5　并行计算应用编程

CUDA、Brook 和 CAL 是着眼于数据并行计算而非图形的 GPU 编程接口。CAL（Compute Abstraction Layer）是 AMD GPU 的一个低级汇编语言接口。Brook 是一个适合 GPU 的流语言，由 Buck 等 2004 提出。CUDA 由 NVIDIA［2007］开发，是用于众核 GPU 和多核 CPU 的可扩展并行编程的 C 和 C++ 语言的扩展。下面根据 Nickolls、Buck、Garland 和 Skadron［2008］的一篇文章对 CUDA 编程模型进行描述。

有了新的模型，GPU 在数据并行和吞吐量计算方面胜过“他人”，除图形应用外，它还执行高性能计算应用。

数据并行问题分解

为了将大的计算问题有效地映射到较高并行度的处理结构上，程序员或编译器将问题分解成可以并行求解的很多小问题。例如，程序员将一个大的结果数据数组划分成块儿，并进一步将

每个块儿划分成元素，这样，结果块儿可以独立地并行计算，且每块中的元素也可以并行计算。图 A-3-3 展示了将结果数据数组分解成块的 3 ×2网格，每个块又进一步分解成 5 ×3 的元素数组。两级的并行分解自然地映射到 GPU 结构上：并行多处理器计算结果块和并行线程计算结果元素。

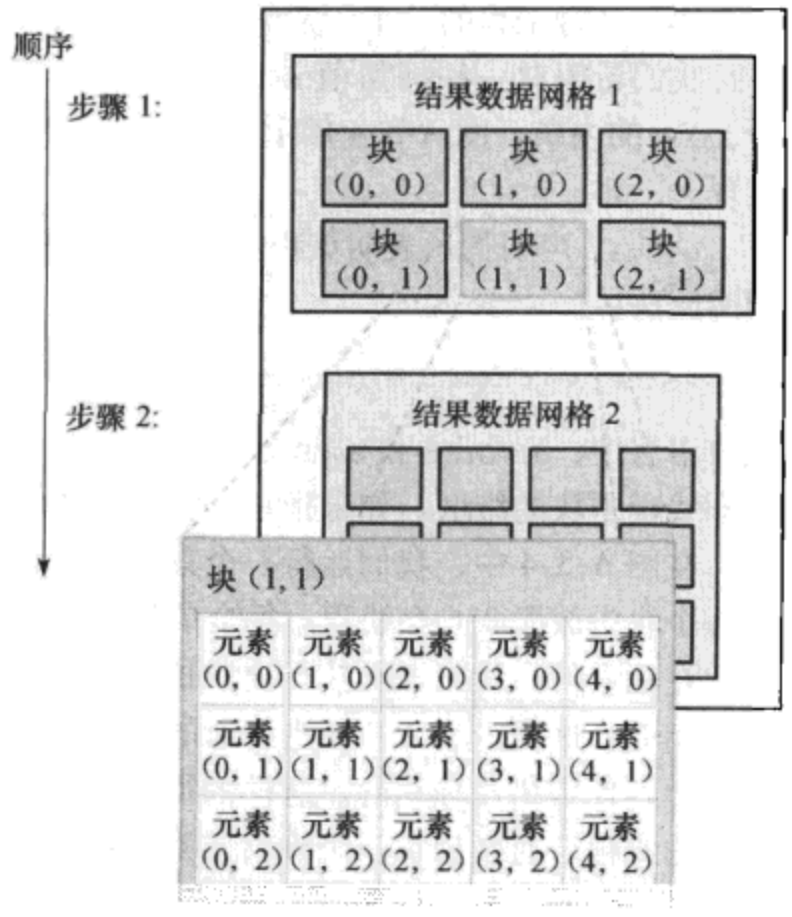


图 A-3-3　将结果数据解耦合到网格的块中以便进行并行计算

程序员编写计算一系列结果数据网格的程序，将每个结果网格划分成可独立并行计算的粗粒度的结果块。程序使用一个细粒度并行线程数组对每个结果块进行计算，在线程中划分工作，以使每个线程计算一个或多个结果元素。

A. 3. 6　使用 CUDA 进行可扩展并行编程

CUDA 可扩展编程语言模型扩充了 C 和 C ++ 语言，以便在高并行多处理器，尤其是 GPU 上，为通用应用开发巨大的并行度。CUDA 的早期经验显示，很多复杂的程序可使用一些易理解的抽象进行表示。自从 NVIDIA 在 2007 年发布了 CUDA，开发者迅速在广泛的应用范围内开发了可扩展的并行程序，包括地震数据处理、可计算化学、线性代数、稀疏矩阵求解、排序、查找、物理建模和可视计算。这些应用透明地扩展到上百处理器核和上千并发线程上。具有 Tesla 统一图形和计算架构（在 A. 4 节和 A. 7 节描述）的 NVIDIA GPU 运行 CUDA C 程序，并可广泛地应用于笔记本、PC、工作站和服务器。CUDA 模型在其他共享内存并行处理结构，包括多核 CPU［Stratton，2008］上也是可用的。

CUDA 提供了三个关键的抽象——层次线程组、共享的内存和栅障同步——为一个层次的线程的传统 C 代码提供了清晰的并行结构。粗粒度数据并行和任务并行中的多级线程、存储器和同步提供了细粒度的数据并行和线程并行。这种抽象引导程序员将问题划分成可以并行独立求解的粗粒度子问题，然后划分成可以并行解决的细粒度片断。可编程模型透明地扩展到数量巨大的处理器核上：一个经过编译的 CUDA 程序在任意数量的处理器上执行，并且仅运行时系统需要知道物理处理器的数量。

CUDA 范例

CUDA 是 C 和 C ++ 编程语言的一个小的扩展。程序员编写一个串行程序，调用并行**内核**[㊀]，这个串行程序可以是一个简单的函数或整个程序。内核在一系列线程间并行交叉执行。程序员将这些线程组织成一个线程块层次结构和线程块网格。**线程块**[㊁]是一系列可以通过阻塞同步和通过私有块存储空间共享存取进行相互协作的并发线程。**网格**[㊂]是一系列可以独立执行并因此并行执行的线程块。

调用一个内核时，程序员指定每块中的线程数和组成网格的块数。每个线程在它的线程块中给定一个唯一的线程号 `threadIdx`，用 `0,1,2,…,blockDim-1` 标记，每个线程块在其网格中给定一个唯一的块 ID 号 `blockIdx`。CUDA 支持线程块包含多达 512 个线程。为方便起见，线程块和网

㊀ 内核（kernel）：适合于一个线程的程序或函数，设计用来被众多线程执行。
㊁ 线程块（thread block）：执行相同线程程序并相互协作以计算结果的一系列并发线程。
㊂ 网格（grid）：执行相同内核程序的一系列线程块。

格可以是 1，2 或 3 维，通过 .x、.y 和 .z 索引域访问。

作为并行编程的一个非常简单的示例，假定我们具有两个向量 x 和 y，每个都有 n 个浮点数，并且我们希望对一些标量值 a 计算 $y = ax + y$ 的结果。这是由 BLAS 线性几何库定义的，被称为 SAXPY的内核。图 A-3-4 展示了在串行处理器和使用 CUDA 的并行两种情况下该计算的 C 代码实现。

_global_声明指示表明该程序是一个内核入口点。CUDA 程序运行的并行内核具有扩展函数调用语法：

```
kernel <<< dimGrid, dimBlock >>> (…parameter list…);
```

在该语法中，dimGrid 和 dimBlock 是类型为 dim3 的三元素向量，dim3 分别指定了块中网格的维度和线程中块的维度。如未指定，默认的维度为 1。

在图 A-3-4 中，我们运行一个具有 n 个线程的网格，为向量中的每个元素分配一个线程，并在每个块中放置 256 个线程。每个单独的线程计算一个元素索引，该索引来自它的线程和块 ID，然后在相应的向量元素上执行要求的计算。比较这个代码的串行和并行版本，我们发现它们惊人地相似。这代表了一个相当普通的模式。串行代码由一个循环组成，在这个循环中，每次迭代与其他迭代是相互独立的。这样的循环可以被机械地转化到并行内核：每个循环迭代变成一个独立的线程。通过为每一个输出元素分配一个单一线程的方法，我们避免了在将结果写回存储器时在线程间需要的任何同步的需求。

用串行循环计算 $y = ax + y$：

```
void saxpy_serial (int n, float alpha, float *x, float *y)
{
        for (int i = 0; i < n; ++i)
          y[i] = alpha *x[i]+ y[i];
}
// 唤起 SAXPY 核
saxpy_serial (n, 2.0, x, y);
```

用 CUDA 并行计算 $y = ax + y$：

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}
// 唤起并行 SAXPY 核 (每块256个线程)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

图 A-3-4　串行 C 代码（上）与并行 CUDA SAXPY 代码（下）（参照第 7 章）

CUDA 并行线程取代 C 串行循环——每个线程与一个循环重复计算的结果相同。并行代码使用 n 个线程计算出 n 个结果，这 n 个线程以 256 个线程的块大小进行组织。

CUDA 内核的文本是一个简单的用于顺序执行线程的 C 函数。因此，CUDA 内核文本通常书写起来直截了当，与为向量操作编写并行代码相比，更为简单。并行度被清楚、直接地决定，通过在运行一个内核时指定一个网络的维度和它的线程块。

并行执行和线程管理是自动的。所有的线程创建、调度和终止由底层系统为程序员处理。实际上，一个 Tesla 架构的 GPU 直接由硬件对线程进行管理。一个块中的多个线程同时执行，并且

或许会在**同步栅障**[㊀]处通过调用__syncthreads()原语进行同步。这保证了在块中的所有线程均到达栅障之前，块中没有可以继续执行的线程。在经过栅障之后，仍可以保证这些线程可以看到由块中栅障之前的线程进行的所有存储器写操作。因此，一个块中的线程可以通过在同步栅障处读、写每个块中共享存储器的方法彼此进行通信。

由于一个块中的线程可以共享存储器并通过栅障进行同步，它们将存在于同一个物理处理器或多处理器上。线程块的数量甚至大大超过处理器的数量。CUDA 线程编程模型对处理器进行了虚拟，并为程序员提供了并行化的灵活性，该并行化可在任何最便利的粒度上进行。虚拟成线程和线程块允许直觉问题的分解，因为块的数量可以通过被处理的数据大小而不是系统中处理器的数量进行指定。同样的 CUDA 程序可以扩展到在很大范围内变动数量的处理器核上。

为管理这个处理元素的虚拟化并提供可扩展性，CUDA 要求线程块可以独立地执行。必须能够在并行或串行方式下，以任意次序执行块。不同的块没办法直接通信，尽管它们可以调整自己的行为，通过在对所有线程可见的全局存储器上使用**原子存储器操作**[㊁]，例如，通过原子地增加队列的指针。这个独立的要求允许线程块可以以任意的次序在任何数量的核上进行调度，使得 CUDA 模型可扩展到任意数量的核上，也可以扩展到许多并行体系结构上。并且它对避免可能的死锁也有帮助。一个应用可能独立或相关地执行多个网格。给定充足的硬件资源的话，独立的网格可以同时执行。相关的网格顺序地执行，在它们之间具有一个隐含的内核间的栅障，这样保证了第一个网格中的所有的块在第二个、相关网格的任何块开始执行之前都已经结束。

线程在它们执行期间，可能会对多个存储器空间中的数据进行访问。每个线程具有一个私有的**局部存储器**[㊂]。CUDA 用局部存储器存储那些不适合在线程寄存器中存储的线程私有变量，也用来存储堆栈帧和进行寄存器溢出处理。每个线程块具有一个**共享存储器**[㊃]，对块中的所有线程可见，且与块具有相同的生命周期。最后，所有的线程均可以访问同样的**全局存储器**[㊄]。程序在共享和全局存储器中声明变量，使用__shared__和__device__限定词。在一个 Tesla 结构的 GPU 上，这些存储器空间对应物理上分离的存储器：每块的共享存储器是一个低延时的片上 RAM，而全局存储器存在于显卡上的高速 DRAM 中。

我们希望共享存储器是靠近每个处理器的低延时存储器，很像一级 cache。因此，它可以在线程块的线程间提供高性能通信和数据共享。由于它具有与相应的线程块同样的生命周期，内核代码将在共享变量中初始化数据，使用共享变量进行计算，并将共享存储器中的结果复制到全局存储器中。顺序相关网格中的线程块通过全局存储器通信，使用全局存储器读输入并写出结果。

图 A-3-5 显示了关于线程、线程块和线程块网格的嵌套级别。并进一步说明了相应的存储器共享级：局部、共享和全局存储器分别与每个线程、每个线程块、每个应用数据共享相对应。

一个程序通过调用 CUDA 运行时的函数，如 cudaMalloc()和 cudaFree()，来管理对内核可见的全局存储器空间。内核可能在一个物理分离的设备上执行，与在 GPU 上运行内核的情况类似。从而，应用程序必须使用 cudaMemcpy()在分配的空间和主机系统存储器之间复制数据。

CUDA 编程模型在网格上与我们熟悉的**单程序多数据**[㊅]（SPMD）模型相似——它显式地加速了并行性，并且每个内核在固定数量的线程上执行。然而，CUDA 与 SPMD 的大多数实现相

㊀ 同步栅障（synchronization barrier）：线程在同步栅障处等待，直到线程块中的所有线程到达该栅障。
㊁ 原子存储器操作（atomic memory operation）：一个存储器读、修改、写操作序列，它们直到执行完成不被任何访问打断。
㊂ 局部存储器（local memory）：每个线程的局部存储器，被该线程私有。
㊃ 共享存储器（shared memory）：每个线程块存储器，被块中的所有线程共享。
㊄ 全局存储器（global memory）：每个应用存储器，被所有的线程共享。
㊅ 单程序多数据（single-program multiple data，SPMD）：并行编程模型的一种网格，其所有的线程执行同样的程序。典型地，SPMD 线程通过栅障同步进行协调。

比，更为灵活，因为每个内核调用动态地创建一个新的网格，该网格对该应用的执行步骤来说，具有合适数量的线程块和线程。程序员可以针对每个内核选用适当的并行度，而不必将计算的所有阶段设计成使用相同数量的线程。图 A-3-6 展示了一个类 SPMD 的 CUDA 代码序列示例。它的第一个实例 kernelF 是在一个 3 × 2 的块组成的 2D 网格上，2D 的线程块由 5 × 3 的线程组成。紧接着的一个实例 kernelG 在一个 1D 的网格上，该网格有 4 个 1D 的线程块，每个 1D 的线程块有 6 个线程。由于 kernelG 依赖于 kernelF 的结果，它们被一个内核间的同步栅障所隔离。

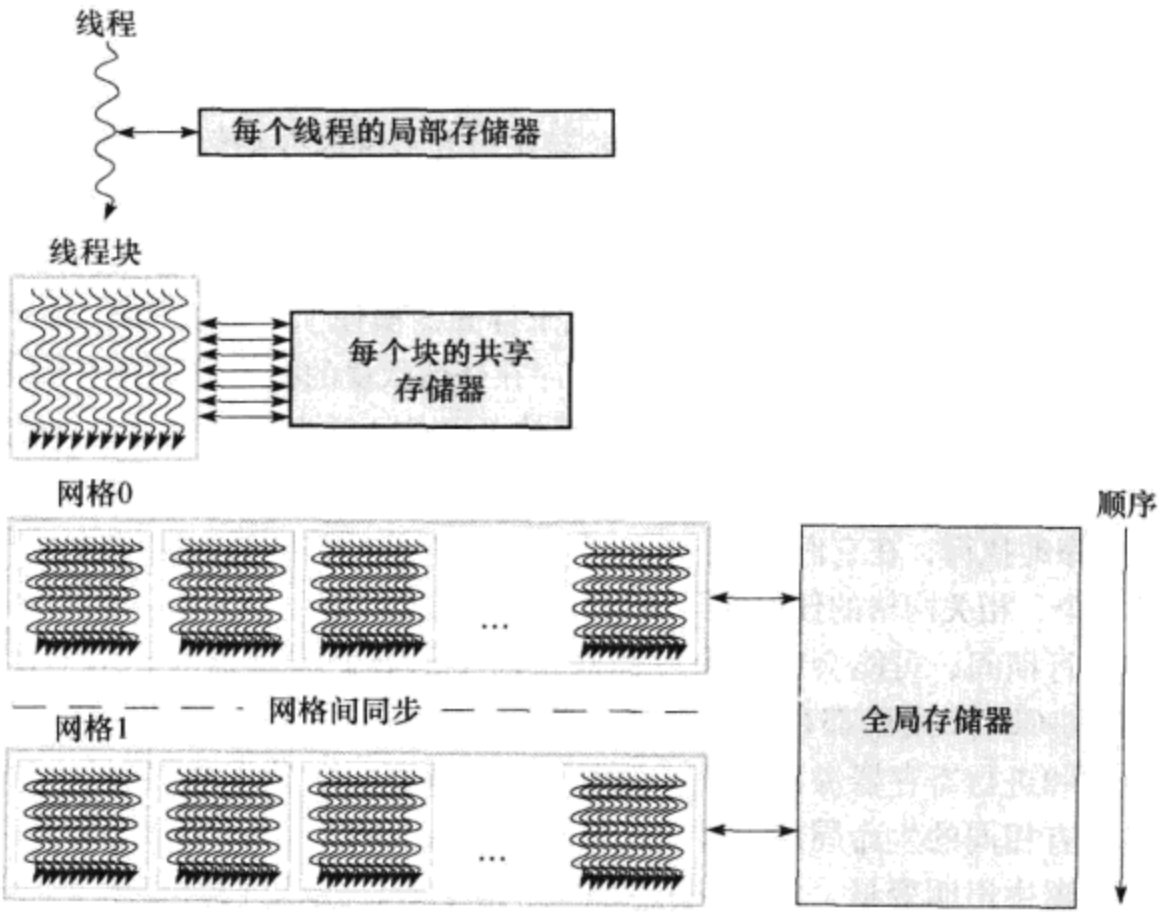


图 A-3-5　Nested 粒度级——线程、线程块和网格——具有通信存储共享级——局部、共享和全局

每个线程的局部存储器为该线程所私有。每个块的共享存储为块中的所有线程所共享。每个应用全局存储为所有的线程所共享。

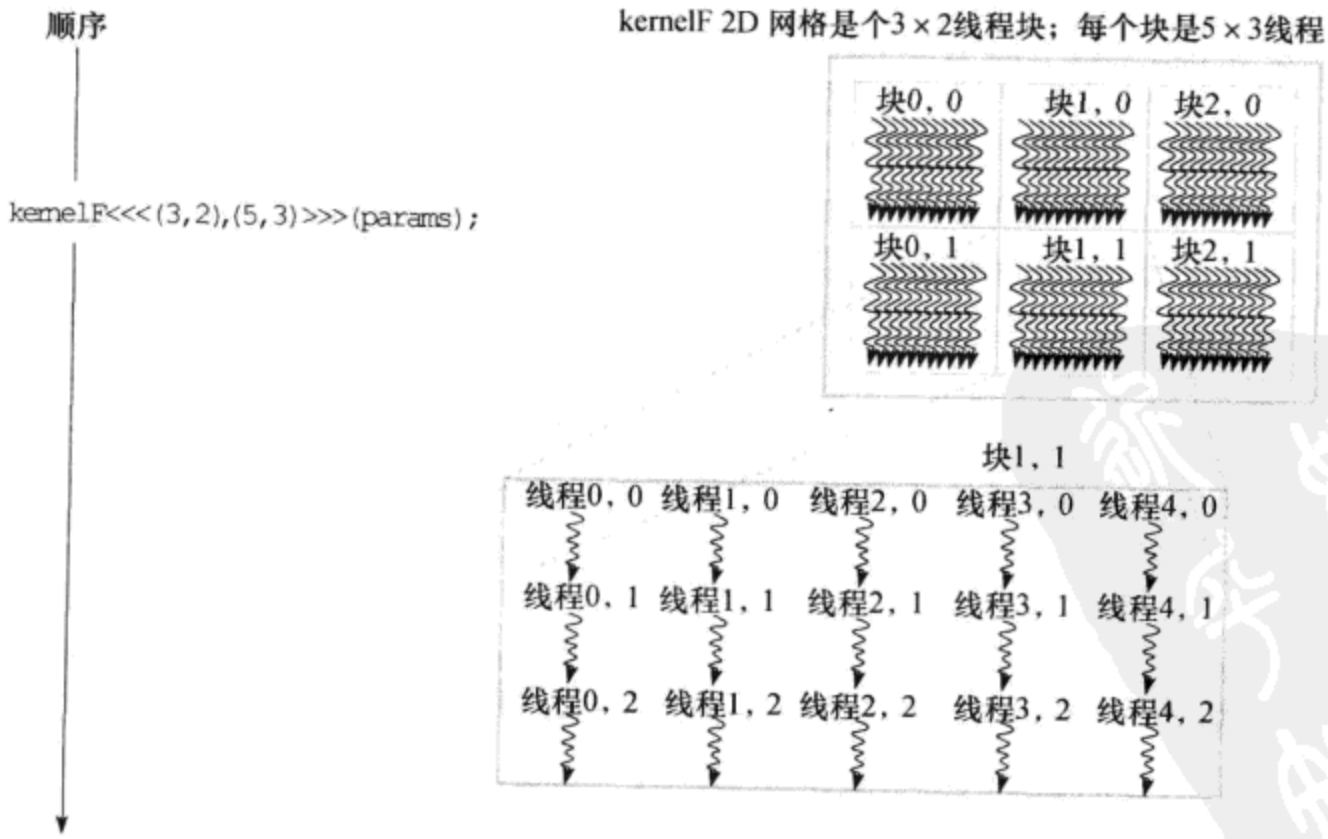


图 A-3-6　kernelF 序列在 2D 线程块的一个 2D 网格上的实例化，一个内核的同步栅障，后面紧跟一个 1D 线程块的 1D 网格上的 kernel G

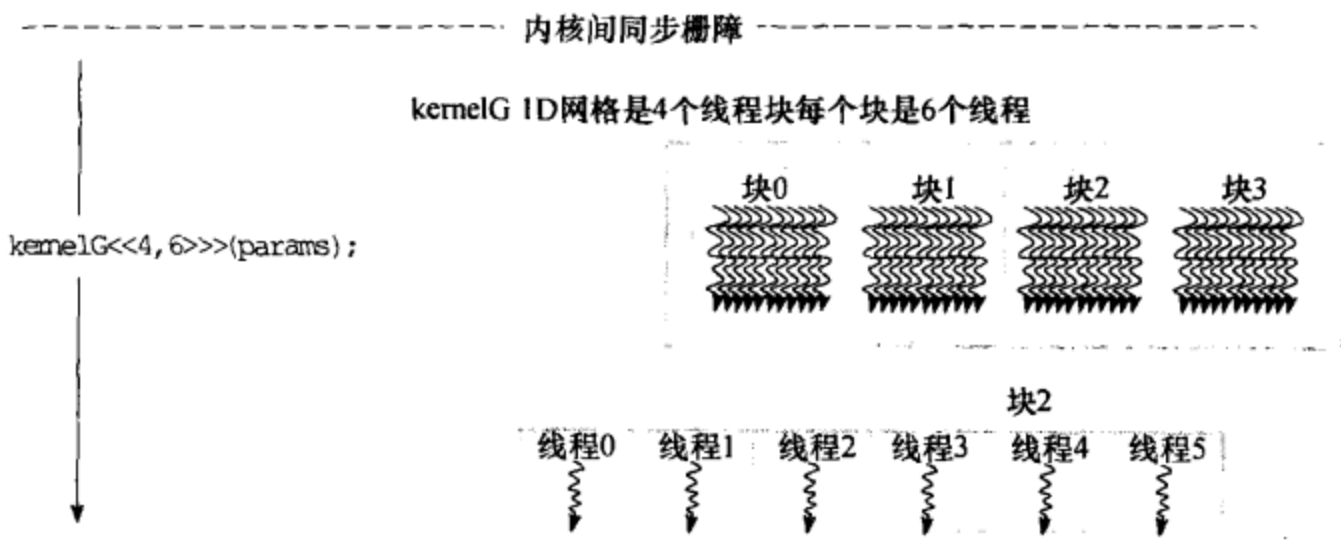


图 A-3-6　（续）

线程块中的并行线程表达了细粒度的数据并行和线程并行。网格中独立的线程块表达了粗粒度的数据并行。独立网格表达了粗粒度任务并行。一个内核是层次结构中一个线程的简单的 C 代码。

A.3.7　一些限制

考虑到效率，且为了简化实现，CUDA 编程模型具有一些限制。线程和线程块仅能通过调用一个并行内核创建，而不能在一个并行内核内部创建。这和需要的独立线程块一起，使得使用一个引入较小运行时开销的简单调度器执行 CUDA 程序成为可能。实际上，Tesla GPU 架构实现了线程和线程块的硬件管理和调度。

任务级并行可以在线程块级表示，而很难在线程块中表示，因为线程同步栅障在块中的所有线程上操作。为能使 CUDA 程序可以在任意数量的处理器上运行，同一个内核网格中的线程块间不允许存在依赖——块必须独立地执行。由于 CUDA 要求线程块独立，并能以任意的次序执行，组合由多个块产生的结果通常必须通过在一个新的线程块的网格上运行第二个内核来完成（尽管线程块可能在对所有线程可见的全局存储器上使用原子存储器操作协调它们的行为——例如，通过原子地增加队列指针）。

当前，递归函数调用在 CUDA 内核中是不允许的。在大规模并行内核中，递归是无吸引力的，因为数以万计的线程提供的栈空间可能被激活，这需要足够内存。通常使用递归表示的串行算法，如快速排序，可以使用嵌套数据并行很好地实现，而不直接使用递归。

为支持集成了一个 CPU 和一个 GPU，且都有各自的存储系统的异构系统结构，CUDA 程序必须在主机存储器和设备存储器之间复制数据和结果。通过使用 DMA 块传输引擎和快速互联，将 CPU-GPU 交互和数据传输的开销降到了最小。需要 GPU 执行的足够大的计算密集型问题比数据量小的问题在加速分摊开销方面要好。

A.3.8　体系结构隐含的问题

图形和计算的并行编程模型使得 GPU 体系结构与 CPU 体系结构相比变得更为不同。驱动 GPU 处理器体系结构改变的 GPU 程序的主要方面有：

- *细粒度数据并行的广泛使用*：渲染程序描述怎样处理一个单一像素点或顶点，CUDA 程序描述怎样计算一个单独的结果。
- *高的线程编程模型*：一个渲染线程程序处理一个单一像素或顶点，一个 CUDA 线程程序可能产生一个单一结果。在每秒 60 帧的情况下，一个 GPU 必须为每帧创建并执行数以

百万计的这样的线程程序。

- 可扩展性：在额外获得了处理器后，一个程序必须自动地提高它的性能，而不用重新编译。
- 密集的浮点（或整数）计算。
- 支持高吞吐量计算。

A.4　多线程的多处理器架构

为面向不同的市场区域，GPU 实现了可扩展数量的多处理器——实际上，GPU 是由多个处理器组成的多处理器。此外，每个多处理器是高度多线程的，高效地执行很多细粒度顶点和像素渲染线程。比较基础的 GPU 具有两到四个多处理器，而一个游戏狂热者的 GPU 或计算平台则具有数十个多处理器。本节着眼于这样一个多线程多处理器的体系结构，一个简化版的 NVIDIA Tesla 流多处理器（SM）在 A.7 节描述。

为什么使用一个多处理器，而不是几个单独的处理器？多处理器内部的并行性提供了高性能的局部性和对 A.3 节中描述的细粒度并行编程模型进行支持的扩展多线程。一个线程块中的多个单个线程在同一个多处理器内一起执行以共享数据。我们在此描述的多线程多处理器具有 8 个标量处理器核，使用紧耦合结构，最多可执行高达 512 个线程（在 A.7 中描述的 SM 可以执行高达 768 个线程）。出于面积和功耗效率的考虑，多处理器在 8 个处理器核间共享面积较大的复杂单元，包括指令 cache、多线程指令单元和共享存储器 RAM。

A.4.1　大规模多线程

GPU 处理器是高度多线程的，以达到几个目标：

- 掩盖从存储器装入和从 DRAM 中纹理预取的延时。
- 支持细粒度并行图形渲染编程模型。
- 支持细粒度并行计算编程模型。
- 将物理处理器虚拟化成线程和线程块以提供透明的扩展性。
- 简化并行编程模型以为一个线程编写一个串行程序。

存储器和纹理读取延迟需要数百个处理器时钟，由于典型的 GPU 具有小的流 cache 而不像 CPU 那样具有大的工作集 cache。一个取请求通常需要一整个 DRAM 访问延迟加上互联和缓冲延迟。多线程使用有用的计算从而掩盖了延迟——在一个线程等待一个载入或纹理提取完成时，处理器可以执行其他的线程。细粒度并行编程模型提供了数千个可保持众多处理器忙碌的独立线程，尽管对于单个线程来说还可以看到很长的存储器延迟。

一个图形顶点或像素渲染程序是一个处理单个顶点或像素的单一线程的程序。相似地，一个 CUDA 程序是为一个单一线程计算结果的一个 C 程序。图形和计算程序例化了众多并行线程以渲染复杂图像并计算大的结果数组。为动态地平衡轮换的顶点和像素渲染线程负载，每个多处理器同时执行多个不同的线程程序和不同类型的渲染程序。

为支持独立的顶点、原语、图形渲染语言的像素编程模型和 CUDA C/C++ 的单线程编程模型，每个 GPU 线程有它自己的私有寄存器、私有存储器、程序计数器和线程执行状态，并且可以执行一个独立的代码路径。为有效地执行数百个并发轻量级线程，GPU 多处理器是硬件多线程的——它使用硬件管理和执行数百个并发线程，没有调度开销。线程块中的并发线程可以在一个栅障处使用一条简单的指令进行同步。轻量级的线程创建、零开销线程调度和快速的栅障同步有效地支持极细粒度的并行。

A.4.2　多处理器体系结构

一个标准的图形和计算多处理器执行顶点、几何和像素段渲染程序，还有并行计算程序。如图 A-4-1 所示，多处理器的例子有 8 个标量处理器（SP）核，每个核都有一个大的多线程寄存器文件（RF），两个特殊功能单元（SFU），一个多线程指令单元，一个指令 cache，一个只读常量 cache 和一个共享存储器。

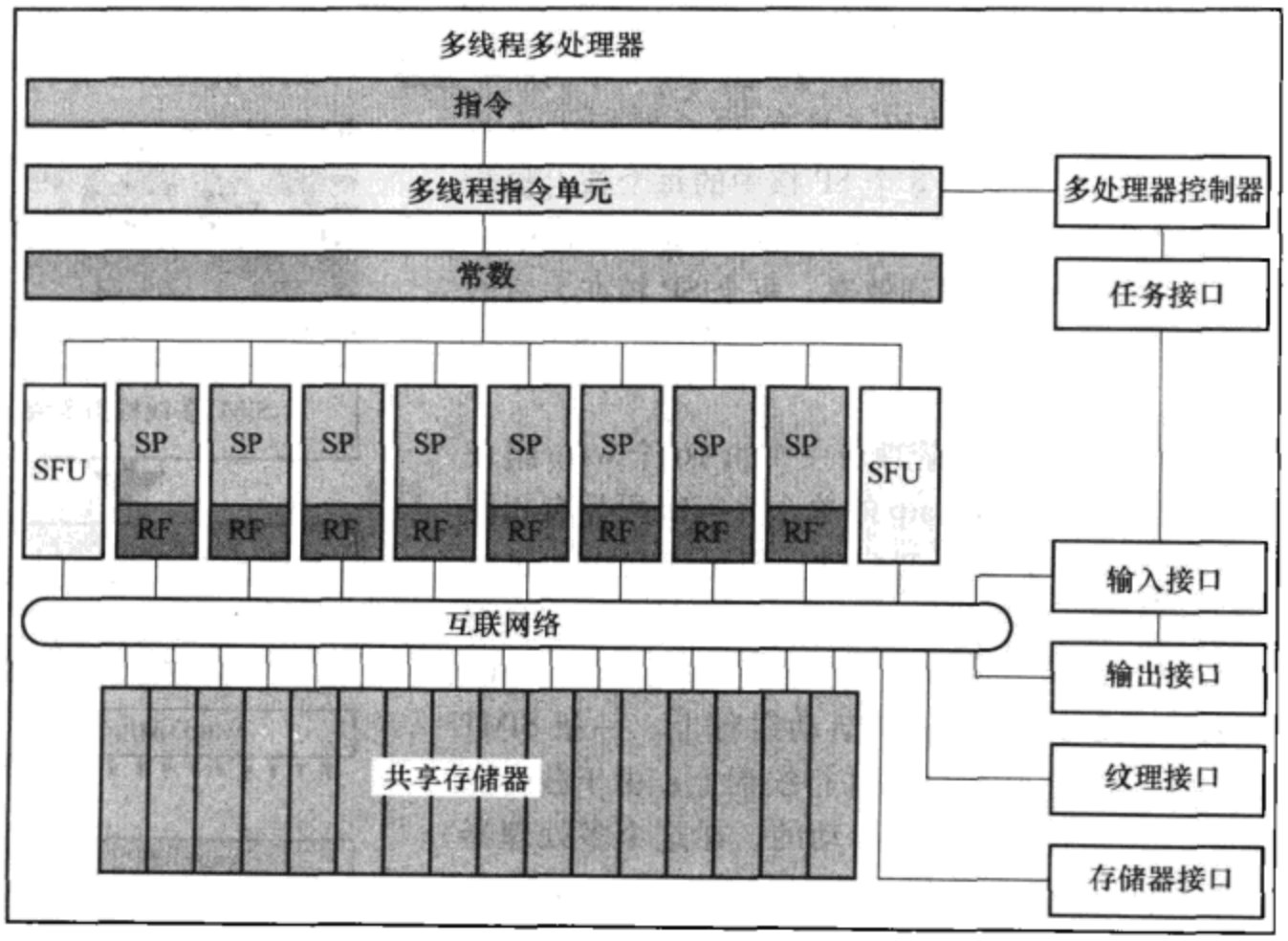


图 A-4-1　具有 8 个标量处理器（SP）核的多线程多处理器

8 个 SP 核每个都有一个大的多线程寄存器文件（RF），并共享一个指令 cache，多线程指令发射单元，常量 cache，两个特殊功能单元（SFU），互联网络和一个多体共享存储器。

16 KB 的共享存储器保存着图形数据缓存和共享计算数据。使用__shared__声明的 CUDA 变量存放在共享存储器中。为映射逻辑图形流水线负载多次通过多处理器，如 A.2 节说明的那样，顶点、几何和像素线程具有独立的输入和输出缓存，且负载的到达和离开独立于线程的执行。

每个 SP 核包含标量整数和浮点算数单元，这些单元执行了大部分的指令。SP 是硬件多线程的，支持高达 64 个线程。每个流水化的 SP 核中，每个线程在一个时钟周期中执行一条标量指令，在不同的 GPU 产品中，时钟范围为 1.2 ~ 1.6 GHz。每个 SP 核都有一个大的寄存器文件（RF），含有 1024 个通用 32 位寄存器，根据它所分配的线程进行划分。程序声明它们的寄存器需求，典型地为每线程分配 16 ~ 64 个标量 32 位寄存器。SP 可以同时运行许多使用减少寄存器的线程，或较少使用较多寄存器的线程。编译器优化寄存器分配以在溢出寄存器和减少线程的代价之间进行平衡。像素渲染程序常常使用 16 个或更少的寄存器，使每个 SP 运行高达 64 个像素渲染线程以掩盖长延迟纹理预取。编译 CUDA 程序通常每个线程需要 32 个寄存器，将每个 SP 限制到 32 个线程，这又限制了在该例的多处理器中每个线程块中这样的内核程序只能有 256 个线程，而不是它的最大数量，512 个线程。

流水的 SFU 执行完成特殊功能的线程指令，并从原始顶点属性插值像素属性。这些指令可以与该 SP 上的指令同时执行。SFU 将在稍后描述。

多处理器通过纹理接口在纹理单元上执行纹理预取指令，并使用存储器接口进行外部存储器的装入、存储和原子访问指令。这些指令可以与该 SP 上的指令同时执行。共享存储器的存取，通过 SP 处理器和共享存储体间的一个低延时的互联网络进行。

A.4.3　单指令多线程（SIMT）

为管理和执行数百个线程以高效地运行多个不同的程序，多处理器使用**单指令多线程（SIMT）**[⊖]架构。它通过调用 warp 创建、管理、调度和执行并行线程组中的并发线程。术语 **warp**[⊜]源于第一个并行线程技术 weaving。图 A-4-2 中的照片通过一个织布机展示了并行线程的一个 warp。这个示例多处理器使用了具有 32 个线程大小的 SIMT warp，在 4 个时钟内，在 8 个 SP 核中的每个核上执行 4 个线程。A.7 节中描述的 Tesla SM 多处理器也使用具有 32 个并行线程大小的 warp，考虑到效率，每个 SP 核在大量的像素线程和计算线程中执行 4 个线程。线程块由一个或多个 warp 组成。

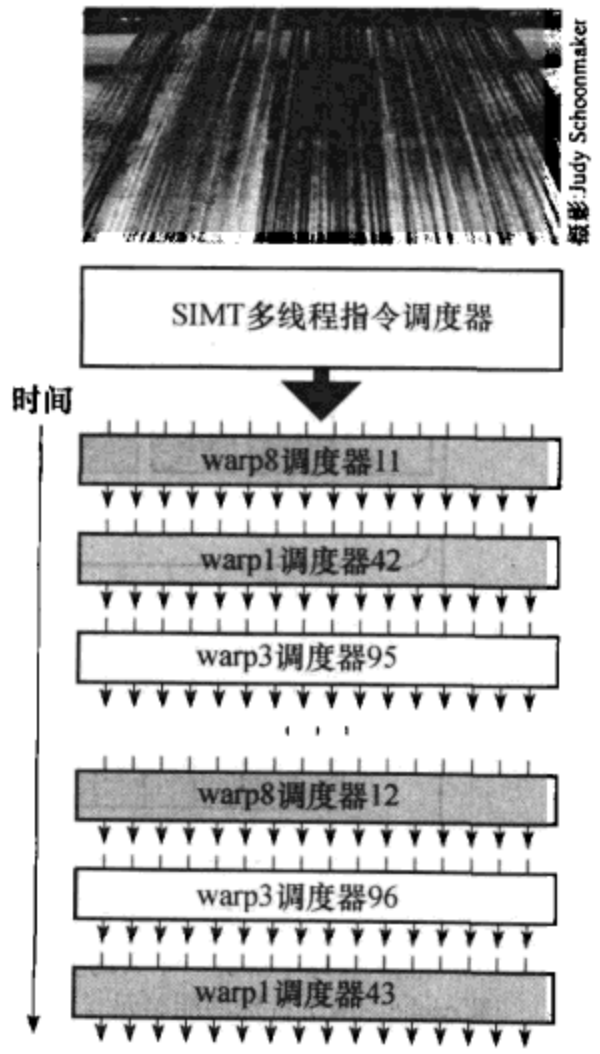


图 A-4-2　SIMT 多线程 warp 调度
调度器选择一个就绪 warp 并同步地向组成 warp 的并行线程发射一条指令。由于各个 warp 是独立的，调度器每次可以选择一个不同的 warp。

这个示例 SIMT 多处理器管理着一个由 16 个 warp 组成的池，共 512 个线程。组成 warp 的单个并行线程具有相同的类型，在同一个程序地址一起启动，但可以自由地进行分支并独立地执行。在每条指令的发射时间，SIMT 多线程指令单元选择一个准备好执行它的下一条指令的 warp，然后将那条指令发射到该 warp 的活动线程上。一条 SIMT 指令被同时广播到 warp 中的活动并行线程上；由于独立的分支或预测，单个线程或许是非活动的。在这个多处理器中，每个 SP 标量处理器核使用 4 个时钟为一个 warp 中的 4 个单独线程执行一条指令，反映了 warp 线程和核的 4:1 比例。

SIMT 处理器结构与单指令多数据设计类似，都是允许一条指令在多个数据通道上执行，但不同的是，SIMT 应用一条指令到多个独立的并行线程上，而不仅是到多个数据通路。SIMD 处理器的一条指令控制一个多条数据通路一起组成的向量，而 SIMT 处理器的一条指令控制一个单独的线程，出于对效率的考虑，SIMT 指令单元向独立并行线程组成的 wrap 发射一条指令。SIMT 处理器运行时在线程间挖掘数据级并行，与超标量处理器运行时在指令间挖掘指令级并行类似。

当 warp 中的所有线程具有相同的执行路径时，一个 SIMT 处理器就获得了全部的效率和性能。如果 warp 的线程通过一个依赖数据的条件分支运行于不同的执行路径，则按照每个分支路径顺序执行，当全部路径完成时，线程回到相同的执行路径。对于等长的路径，运行 if-else 代码块不同路径的效率为 50%。多处理器使用一个分支同步栈管理独立的分支和聚集线程。不同的 warp 以全速独立执行，而不考虑它们是在执行普通的还是分离的代码路径。结果就是，SIMT

⊖ 单指令多线程（single-instruction multiple-thread，SIMT）：应用一条指令到多个独立的并行线程上的一种处理器结构。

⊜ warp：在 SIMT 结构下，一起执行相同指令的并行线程集。

GPU 在分支代码方面与早期的 GPU 相比，明显地更具效率和灵活性，因为它们的 warp 比先前 GPU 中的 SIMD 宽度要窄得多。

与 SIMD 向量结构相对的是，SIMT 使程序员能够为单独的独立线程编写线程级并行代码，也可以为众多同样的线程编写数据并行代码。为了正确地编程，程序员可以从本质上忽略 warp 的 SIMT 执行属性；然而，只要留意到代码很少要求 warp 中的线程进行分支，就可以获得实质的性能提升。实际上，这与传统代码中的缓存行是类似的：在为正确性而进行设计时，缓存行可以被安全地忽略，但在为峰值性能进行设计时，就必须在代码结构中进行考虑。

A.4.4　SIMT warp 执行和分支

独立打包的 SIMT 调度方法比先前 GPU 结构的调度方法更具灵活性。一个 warp 由相同类型的并行线程组成：顶点、几何、像素或计算。像素段渲染处理的基本单元是被作为 4 个像素渲染线程实现的 2×2 的像素方块。多处理器控制器将像素块打包到 warp 中。同样，多组的顶点和原语打包到多个 warp 中，而将计算线程打包到一个 warp 中。一个线程块包含一个或多个 warp。SIMT 设计在一个 warp 的多个并行线程间有效地共享取指和发射单元，但需要激活全部 warp 中的线程以获得最高的效率。

这种统一的多处理器调度和多 warp 类型的同时执行，使得可以同时执行顶点和像素 warp。由于每个处理器核有四个线程通道，它的 warp 调度以比处理器时钟频率稍低的频率运行。在每个调度周期期间，它选择一个 warp 执行一条 SIMT warp 指令，如图 A-4-2 中所示。一条已发射的 warp 指令被打包成四组八个线程，在四个处理器周期上执行。处理器流水线完成每条指令需要几个时钟的延迟。如果活动 warp 的数量是每个 warp 执行流水线延时的数倍，程序员可以忽略流水线延时。对于这样的多处理器，八个 warp 的罗宾环，在同一 warp 的连续指令间的调度周期为 32 个时钟周期。如果程序可以在每个多处理器上保持 256 个线程活动，多至 32 个时钟周期的指令延迟可以从单个顺序线程中隐藏。然而，具有较少活动 warp 的话，处理器的流水线深度将变得很明显，并且可能导致处理器停顿。

一个设计中的挑战性问题是实现不同 warp 程序和程序类型动态混合的零开销 warp 调度。指令调度器必须每四个时钟选择一个 warp，以便每个时钟为每个线程发射一条指令，相当于每个核的 IPC 是 1。由于 warp 是独立的，仅相同 warp 的顺序指令间存在依赖。调度器使用一个依靠寄存器的记分板来描述 warp，warp 的活动线程准备好执行一条指令。它按优先级区分所有这些就绪的线程，并为发射选择一个优先级最高的。划分优先级必须考虑 warp 的类型、指令类型和要求，保证对所有的活动 warp 是公平的。

A.4.5　管理线程和线程块

多处理器控制器和指令单元管理线程和线程块。控制器接收工作请求、输入数据并负责共享资源访问的仲裁，包括纹理单元，存储器访问路径和 I/O 路径。对于图形负载，它创建并同时管理三种类型的图形线程：顶点、几何和像素。每个图形任务类型具有独立的输入和输出路径。它累积每个这样的输入工作类型，将其打包成执行同样线程程序的并行线程的 SIMT warp。它分配一个空闲 warp，为 warp 线程分配寄存器，并启动 warp 在多处理器中的执行。每个程序声明各自的程序寄存器命令；仅在控制器可以为一个 warp 线程分配所请求的寄存器数量时，控制器才启动该 warp。当 warp 的所有线程退出时，控制器取出结果并释放 warp

的寄存器和资源。

控制器创建**协作线程组（CTA）**[⊖]，它以一个或多个并行线程 warp 的形式实现了 CUDA 线程块。当控制器可以创建所有 CTA warp 并分配所有 CTA 资源时，它创建一个 CTA。除线程和寄存器外，一个 CTA 还需要为其分配共享存储器和栅障。程序声明它所需要的空间，控制器将进行等待，直到在运行 CTA 前，它可以分配这些空间。然后它以 warp 的调度速率创建 CTA warp，以便 CTA 程序以多处理器全部的性能立即开始执行。控制器监控何时一个 CTA 的所有线程均已退出，并释放 CTA 共享资源和它的 warp 资源。

A.4.6　线程指令

SP 线程处理器为单个线程执行标量指令，不像早前的 GPU 向量指令结构，为每个顶点或像素渲染程序执行由四部分组成的向量指令。顶点程序通常计算（x，y，z，w）位置向量，而像素渲染程序计算（红，绿，蓝，alpha）颜色向量。然而，渲染程序正变得更长，使用更多标量，完全占据、甚至占用较早 GPU 的四部分组成的向量结构中的两个正变得更为困难。实质上，SIMT 结构使 32 个独立的像素线程并行，而非一个像素中的四个向量组成部分并行。在 CUDA C/C++程序中，每个线程都具有占主导地位的标量代码。先前的 GPU 使用向量打包（例如，组合工作中的子向量以提高效率），但这使得调度硬件和编译器都变得复杂。标量指令更为简单，并且易于编译。纹理指令仍旧是基于向量的，接受一个源坐标向量并返回一个过滤后的颜色向量。

为支持具有不同二进制微指令格式的多样 GPU，高层图形和计算语言编译器产生中间汇编级指令（如，Direct3D 向量指令或 PTX 标量指令），然后对其进行优化并将其转化为二进制 GPU 微指令。NVIDIA PTX（并行线程执行）指令集定义［2007］为编译器提供一个稳定的目标 ISA，并且以改进的二进制伪指令集架构在几代的 GPU 上提供兼容性。优化器很容易地将 Direct3D 向量指令扩展到多条标量二进制微指令。PTX 标量指令与标量二进制微指令的转化几乎是一对一的，尽管一些 PTX 指令扩展到多条标量二进制微指令，多条 PTX 指令可以折叠进一条二进制微指令。由于中间汇编级指令使用虚拟寄存器，优化器分析数据依赖并分配真实寄存器。优化器消除了无效代码，在可行时将多条指令折叠到一起，并优化 SIMT 分支转移和汇集点。

A.4.7　指令集架构（ISA）

这里描述的线程指令集架构是一个简化版的 Tesla 结构 PTX 的 ISA，一个基于寄存器的标量指令集，包含浮点、整数、逻辑、转化、特殊功能、流控制、存储器访问和纹理操作。图 A-4-3 列出了基本的 PTX GPU 线程指令；详细资料可参考 NVIDIA PTX 说明［2007］。指令格式是：

```
opcode.type d,a,b,c;
```

其中 `d` 是目的操作数，`a`、`b`、`c` 是源操作数，`.type` 是以下之一：

类型	指定器
无类型的位 8,16,32和 64 位	.b8,.b16,.b32,.b64
无符号的整型 8,16,32和 64 位	.u8,.u16,.u32,.u64
有符号的整型 8,16,32和 64 位	.s8,.s16,.s32,.s64
浮点 16,32和 64 位	.f16,.f32,.f64

⊖ 协作线程组（cooperative thread array，CTA）：一组执行相同的线程程序且可以协作计算出一个结果的并发线程。一个 GPU CTA 实现一个 CUDA 线程块。

基本PTX GPU线程指令

组	指令	举例	意义	说明
算术	arithmetic *.type* = .s32, .u32, .f32, .s64, .u64, .f64			
	add.*type*	add.f32 d, a, b	d = a + b;	
	sub.*type*	sub.f32 d, a, b	d = a − b;	
	mul.*type*	mul.f32 d, a, b	d = a * b;	
	mad.*type*	mad.f32 d, a, b, c	d = a * b + c;	乘-加
	div.*type*	div.f32 d, a, b	d = a / b;	除法微指令
	rem.*type*	rem.u32 d, a, b	d = a % b;	整数求余
	abs.*type*	abs.f32 d, a	d = |a|;	
	neg.*type*	neg.f32 d, a	d = 0 − a;	
	min.*type*	min.f32 d, a, b	d = (a < b)? a:b;	正常浮点数选择
	max.*type*	max.f32 d, a, b	d = (a > b)? a:b;	正常浮点数选择
	setp.*cmp*.*type*	setp.lt.f32 p, a, b	p = (a < b);	比较和设置断言
	numeric *.cmp* =eq, ne, lt, le, gt, ge; unordered *cmp* = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.*type*	mov.b32 d, a	d = a;	移动
	selp.*type*	selp.f32 d, a, b, p	d = p? a: b;	用断言选择
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	从a类型转换成d类型
特殊功能	special *.type* = .f32 (some .f64)			
	rcp.*type*	rcp.f32 d, a	d = 1/a;	倒数
	sqrt.*type*	sqrt.f32 d, a	d = sqrt(a);	平方根
	rsqrt.*type*	rsqrt.f32 d, a	d = 1/sqrt(a);	平方根倒数
	sin.*type*	sin.f32 d, a	d = sin(a);	正弦
	cos.*type*	cos.f32 d, a	d = cos(a);	余弦
	lg2.*type*	lg2.f32 d, a	d = log(a)/log(2)	以2为底的对数
	ex2.*type*	ex2.f32 d, a	d = 2 ** a;	二进制指数
逻辑	logic.*type* = .pred, .b32, .b64			
	and.*type*	and.b32 d, a, b	d = a & b;	
	or.*type*	or.b32 d, a, b	d = a | b;	
	xor.*type*	xor.b32 d, a, b	d = a ^ b;	
	not.*type*	not.b32 d, a, b	d = ~a;	1的补码
	cnot.*type*	cnot.b32 d, a, b	d = (a==0)? 1:0;	C逻辑非
	shl.*type*	shl.b32 d, a, b	d = a << b;	左移
	shr.*type*	shr.s32 d, a, b	d = a >> b;	右移
存储器访问	memory *.space* = .global, .shared, .local, .const; *.type* = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.*space*.*type*	ld.global.b32 d, [a+off]	d = *(a+off);	从存储器*space*装入
	st.*space*.*type*	st.shared.b32 [d+off], a	*(d+off) = a;	存储到存储器*space*
	tex.*nd*.*dtyp*.*btype*	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	纹理查询
	atom.*spc*.*op*.*type*	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = *op*(*a, b); }	原子读-改-写操作
	atom *.op* = and, or, xor, add, min, max, exch, cas; *.spc* = .global; *.type* = .b32			
控制流	branch	@p bra target	if (p) goto target;	条件分支
	call	call (ret), func, (params)	ret = func(params);	调用函数
	ret	ret	return;	从函数调用返回
	bar.sync	bar.sync d	wait for threads	栅障同步
	exit	exit	exit;	中止线程执行

图 A-4-3　基本 PTX GPU 线程指令

源操作数是寄存器中 32 位或 64 位的标量值，一个立即数或一个常量；断言操作数是 1 位的布尔值。除写存储器以外，目的操作数是寄存器。使用@p 或@!p 前缀对指令进行断言，其中 p 是一个断言寄存器。存储器和纹理指令传输由 2～4 个部分组成的最高可达 128 位的标量或向量。PTX 指令指定了一个线程的行为。

PTX 算术指令在 32 位或 64 位浮点、有符号整型和无符号整型类型上操作。近来的 GPU 支持 64 位双精度浮点，参考 A.6 节。在当前的 GPU 中，PTX 64 位整型和逻辑指令被转换成

两条或更多的执行 32 位操作的二进制微指令。GPU 特殊功能指令仅限于 32 位的浮点。线程控制流指令是条件 branch、函数 call 和 return，线程 exit 和 bar.sync（barrier synchronization）。条件分支指令@p bra target 使用断言寄存器 p（或!p）确定线程是否进行分支，断言寄存器由先前通过比较和设置断言指令 setp 设置。其他指令同样可以根据断言寄存器是真还是假进行断言。

存储器访问指令

tex 指令获取并过滤通过纹理子系统的从存储器中 1D、2D 和 3D 纹理矩阵采样得到的纹理。纹理读取通常使用插值产生的浮点坐标来定位一个纹理。一旦一个图形像素渲染线程计算出它的像素段颜色，光栅操作处理器将该颜色与指派的（x，y）像素位置的像素颜色进行混合，并将最终颜色写入存储器。

为了支持计算和 C/C ++语言的需要，Tesla PTX ISA 实现了存储器存取指令。它们使用由寄存器寻址的整型字节加上偏移地址，使传统的编译代码优化变得容易。存储器存取指令在处理器中是很普遍的，但在 Tesla 架构 GPU 中却是一种重要的新能力，因为之前的 GPU 仅提供图形 API 需要的纹理和像素访问。

为进行计算，存取指令访问三个可读写的存储器空间，这三个空间实现了 A.3 节中对应的 CUDA 存储器空间：

- 局部存储器，存储每个线程可私有寻址的临时数据（在外部 DRAM 中实现）。
- 共享存储器，存储被同一个 CTA/线程块中协作线程所共享的低延时访问数据（在片上的 SRAM 中实现）。
- 全局存储器，存储被一个计算应用中的所有线程所共享的大批数据（在外部 DRAM 中实现）。

存储器存取指令 ld.global、st.global、ld.shared、st.shared、ld.local 和 st.local 访问全局、共享和局部存储器空间。计算程序使用快速栅障同步指令 bar.sync 来同步 CTA/线程块中通过共享和全局存储器互相通信的线程。

为提高存储器带宽和减小开销，当地址落进同一块中且符合对齐规则时，局部和全局存取指令把同一 SIMT warp 中的单独的并行线程请求合并成一个存储块请求。合并存储器请求与单独线程的分立请求相比性能得到了显著提升。多处理器巨大的线程数量，加之对众多 load 请求的高效支持，帮助掩盖了在外部 DRAM 中实现的局部和全局存储器的 load-to-use 延迟。

最近的 Tesla 架构 GPU 通过 atom.op.u32 指令，也为存储器提供了有效的原子存储器操作，包括整型操作 add、min、max、and、or、xor、exchange 和 cas（compare-and-swap）操作，使并行简化和并行数据结构管理变得容易。

用于线程通信的栅障同步

快速栅障同步允许 CUDA 程序通过简单的调用__syncthreads()；作为每个内部线程通信步骤的一部分，利用共享存储器和全局存储器频繁地通信。同步的内在函数产生单一的一条 bar.sync指令。可是，在每个 CUDA 线程块所具有的最高 512 个线程之间实现快速的栅障同步是一个挑战。

将线程划分成具有 32 个线程的 SIMT warp 以一个 32 的因子减小了同步的难度。线程在 SIMT 线程调度器中的栅障处等待，因此它们在等待时不消耗任何处理器周期。当一个线程执行 bar.sync指令时，它增加栅障的线程到达计数器，且调度器将线程标记成等待栅障的。一旦所有的 CTA 线程到达，栅障计数器与期待的 terminal 数值匹配，调度器释放所有在栅障处等待的线程并继续执行线程。

A.4.8　流处理器（SP）

多线程流处理（SP）核是多处理器中主要的线程指令处理器。它的寄存器文件（RF）提供1024 个标量 32 位寄存器，可用于最多 64 个线程。它执行所有的基本浮点操作，包括 add.f32、mul.f32、mad.f32（浮点乘加）、min.f32、max.f32 和 setp.f32（浮点比较并设置断言）。浮点加和乘操作与单精度浮点数的 IEEE 754 标准兼容，包含非数值（NaN）和无穷值。SP 核也实现了图 A-4-3 中的 32 位和 64 位整数算术、比较、转换和逻辑 PTX 指令。

浮点 add 和 mul 操作使用 IEEE 的取最接近偶数作为默认的舍入模式。mad.f32 浮点乘加操作执行带截断的乘法，接着进行使用取最接近偶数的加法。SP 冲刷输入的非正常操作数到保留符号零。目标输出指数范围下溢的结果在舍入后被冲刷到保留符号零。

A.4.9　特殊功能单元（SFU）

某些线程指令可以在 SFU 上执行，同时其他线程指令在 SP 上执行。SFU 实现了图 A-4-3 中的特殊功能指令，这些指令计算 32 位浮点近似值的倒数、倒数平方根和关键超越函数。也实现了像素渲染的 32 位浮点平面属性插值，提供精确的属性插值，如颜色、深度和纹理坐标。

每个流水 SFU 每时钟周期产生一个 32 位的浮点特殊功能结果；每个多处理器中的两个 SFU 执行特殊功能指令的速度，是八个 SP 执行简单指令速度的四分之一。与八个 SP 并发，SFU 同样执行 mul.f32 乘指令，用合适的指令混合增加了线程 50% 的峰值运算速度。

为进行功能评价，Tesla 架构 SFU 使用基于加强鞍点近似的二次插值，用于近似倒数、倒数平方根、$\log_2 x$、2^x 和 sin/cos 功能。功能评估精度从 22 ~ 24 位尾数位。关于 SFU 算术的更多详细信息可参考 A.6 节。

A.4.10　与其他多处理器的比较

与 SIMD 向量架构的处理器（如 x86 SSE）比较，SIMT 多处理器可以独立地执行单个线程，而不是总是在一个同步组中一起执行它们。SIMT 硬件在独立线程中寻找数据并行性，而 SIMD 硬件则需要软件在每条向量指令中显式地表达数据并行性。一个 SIMT 机器，可以在 warp 中的 32 个线程具有相同的执行路径时，同时执行它们，也可以在它们出现分支时独立地执行每个线程。由于 SIMT 程序和指令仅仅描述了一个单独独立线程的行为，而不是 SIMD 具有四个或更多数据通路的数据向量，这种优势是显著的。然而 SIMT 多处理器具有类似于 SIMD 的效率，将一条指令单元的面积和成本分摊到一个 warp 中的 32 个线程上和 8 个流处理器核上。SIMT 提供 SIMD 的性能和多线程的生产力，避免了为边界条件和部分分支显式地编码 SIMD 向量的需要。

SIMT 多处理器引入了较小的开销，由于它是具有硬件栅障同步的硬件多线程。这允许图形渲染和 CUDA 线程表达非常细粒度并行性。图形和 CUDA 程序使用线程表达单线程程序中的细粒度数据并行，而不是强迫程序员使用 SIMD 向量指令表示它。开发标量单线程代码比向量代码更为简单且更有效率，且 SIMT 多处理器执行这些代码具有类似 SIMD 的效率。

将八个流处理器核紧密地连接到一起，组成一个多处理器，然后实现可扩展数量的这种多处理器就构成一个两级的由多处理器组成的多处理器。CUDA 编程模型，通过为细粒度并行计算提供单独线程，和为粗粒度并行操作提供线程块网格，对这两级层次进行了开发。相同的线程程序可提供细粒度和粗粒度这两种操作。相反，使用 SIMD 向量指令的 CPU 必须使用两种不同的编程模型以提供细粒度和粗粒度操作：在不同核上的粗粒度并行线程和用于细粒度数据并行的 SIMD 向量指令。

A.4.11　多线程多处理器总结

示例中基于 Tesla 构架的 GPU 多处理器是高度多线程的，同时执行总数可达 512 的轻量级线程以支持细粒度像素渲染和 CUDA 线程。它使用在 SIMD 架构和多线程上的一种变种，称为 SIMT（单指令多线程），高效地将一条指令广播到一个 warp 中的 32 个并行线程中，同时允许每个线程独立地进行分支和执行。每个线程在八个流处理器（SP）核中的一个执行它的指令流，流处理器是多线程的，最高可达 64 线程。

PTX ISA 是一个基于寄存器的存/取标量 ISA，描述一个单线程的执行。由于 PTX 指令被优化且被转换成一个特定 GPU 的二进制微指令，硬件指令可以快速地形成，而无需打断编译器和产生 PTX 指令的软件工具。

A.5　并行存储系统

在 GPU 本身之外，存储子系统是图形系统性能的最重要的决定因素。存储器的读和写，图形负载都要求非常高的传输速率。像素写和混合（读—修改—写）操作、深度缓存读和写、纹理映射读、产生和拒绝顶点以及属性数据读，组成了存储器通信的大部分。

现代 GPU 是高度并行的，如图 A-2-5 所示。例如，在 600 MHz 情况下，GeForce 8800 每个时钟周期可处理 32 个像素。典型地，每个像素需要一个颜色读和写，和一个 4 个字节像素的深度读和写。通常，平均读取每 4 个字节中 2 到 3 个纹理像素，来产生那个像素的颜色。因此对于一个典型的情况，每时钟周期需要 28 字节乘以 32 像素 = 896 字节。显然，对存储系统的带宽需求是巨大的。

为满足这些需求，GPU 存储系统具有以下特点：

- 它们是宽的，意味着具有大量的管脚用于 GPU 和它的存储设备之间的数据传输，且存储器阵列本身由许多 DRAM 芯片组成以提供完全的数据总线宽度。
- 它们是快的，意味着采用了快速的信号技术以最大化每个管脚上的数据速率（位/秒）。
- GPU 寻找每一个可用的时钟周期进行数据传输，读写存储阵列。特别地，为了达到这个目标，GPU 并非旨在最小化存储系统的延迟。高吞吐量（使用效率）和低延迟从根本上来说是冲突的。
- 同时采用了有损（程序员必须知道）和无损（对应用不可见，且只能看时机进行）压缩技术。
- 采用 cache 和运行相关结构减少片外通信需求的总量，并保证用于搬运数据的时钟周期尽可能满地使用。

A.5.1　DRAM 的考虑

GPU 必须考虑 DRAM 独有的特性。DRAM 芯片在内部被安排成多体（典型的为 4 ~ 8），每个体包含 2 的幂次数量的行（典型的为 16 384 左右），且每行包含 2 的幂次数量的位（典型的为 8192）。DRAM 在它们的控制处理器上引入了多种时间需求。例如，激活一行需要数十个时钟周期，但一旦激活，使用一个新的列地址每四个时钟周期可以完成该行中任意位的访问。双数据率（DDR）同步 DRAM 在接口时钟的上升沿和下降沿都传输数据（参考第 5 章）。因此，使用 1 GHz 时钟的 DDR DRAM 每秒每数据管脚传送 2 G 位。图形 DDR DRAM 通常具有 32 个双向数据管脚，因此每时钟周期可从 DRAM 读或写 8 个字节。

GPU 内部有大量的存储传输发生器。每个逻辑图形流水线的不同段都有它们自己的请求流：命令和定点属性读取，渲染纹理读取和 load/store，像素深度和颜色读写。在每个逻辑段，常有

多个独立单元来传送并行吞吐量。其中的每一个都是独立的存储器请求者。当从存储系统的观点出发，有大量无关联的请求在同时执行。这天生与 DRAM 首选的引用模式不匹配。GPU 存储控制器的一个解决方案是为流向不同 DRAM 体的通信维持独立的堆，一直等待，直到一个特定 DRAM 行有足够的通信量在等待，这时激活该行，并一次传输所有的通信量。要注意的是，积聚等待的请求，虽然对 DRAM 行局部性和因此而来的数据总线的利用率来说是有益的，但在那些花费时间等待其他请求的请求者看来，导致了更长的平均延迟。设计中必须考虑到，没有特殊的请求等待过长，否则一些处理单元因等待数据而饥饿，并最终导致相邻的处理器处于空闲状态。

GPU 存储子系统被安排成多存储器分区，每个都有一个完全独立的存储控制器和一两个 DRAM 设备，这些 DRAM 设备被这些分区完全和排他地所有。为达到最好的负载平衡并因此接近 n 分区的理论性能，地址是细致交织的，均匀地穿过所有的存储器分区。典型的分区交织步幅是一个由几百字节组成的块。设计存储器分区的数量，以平衡处理器和其他存储请求者的数量。

A.5.2　cache

典型的 GPU 工作负载具有庞大的工作集——将近数百兆字节用来产生一个单一的图形帧。不像 CPU，构建足够大的片上 cache，用来存放图形应用中接近整个工作集的所有数据是不现实的。然而 CPU 可以假定非常高的 cache 命中率（99.9% 甚至更高），GPU 通常的命中率接近 90% 并因此必须处理众多飞行缺失。CPU 可以适度地设计为在等待较少的 cache 缺失时暂停，GPU 则需要将处理缺失和命中混合。我们称之为一种流 cache 结构。

GPU cache 必须为它们的客户提供非常高的带宽。考虑一个纹理 cache 的情形。一个典型的纹理单元，每个时钟周期可以为每四个像素计算两个双线性插值，且一个 GPU 可能有很多这样独立的操作纹理单元。每个双线性插值需要四个分立的纹理像素，每个纹理像素可能是一个 64 位的值。四个 16 位的成分是有代表性的。因此，总带宽是 $2 \times 4 \times 4 \times 64 = 2048$ 位每时钟周期。每个分立的 64 位纹理像素是独立寻址的，因此每时钟周期 cache 需要处理 32 个唯一地址。这天生地适合多体和/或多端口组织的 SRAM 阵列。

A.5.3　MMU

现代 GPU 具有将虚拟地址转化成物理地址的能力。在 GeForce 8800 中，所有的处理单元在一个 40 位的虚拟地址空间中产生存储器地址。为进行计算，load 和 store 线程指令使用 32 位的字节地址，该地址通过加上一个 40 位的偏移扩展成一个 40 位的虚拟地址。存储管理单元完成虚到实的地址转换；硬件从局部存储器中读取页表，以对在处理器核和渲染引擎上扩散的转换查找旁路缓冲层次行为上的缺失做出反应。除物理页位之外，GPU 页表条目为每一页指定了压缩算法。页表大小在 4 ~ 128 K 字节之间变化。

A.5.4　存储器空间

如 A.3 节介绍的那样，CUDA 展示了不同的存储器空间以允许程序员用最优性能的方式保存数据。为方便下面的讨论，假设使用 NVIDIA Tesla 构架 GPU。

A.5.5　全局存储器

全局存储器存储在外部 DRAM 中；对于任何一个物理流多处理器（SM）来说它都不是局部的，因为它意欲用来在不同网格中的不同 CTA（线程块）间进行通信。实际上，访问全局存储器中一个位置的众多 CTA 或许不会在 GPU 上同时执行；通过设计，在 CUDA 中程序员不知道 CTA 执行的相对顺序。由于地址空间在所有存储器分区间是均匀分布的，从任意流多处理器到

任意 DRAM 分区之间必须有一个读/写路径。

不同线程（和不同处理器）对全局存储器的访问不能保证具有顺序一致性。线程程序看到的是一个松弛的存储器顺序模型。在一个线程内部，对相同地址的存储器的读写次序是保持的，但对不同地址的访问次序可能是没有保持的。不同线程的存储器读写请求是无序的。在一个 CTA 内部，栅障同步指令 bar.sync 可以用来在 CTA 的线程间获得严格的存储器次序。membar 线程指令提供一个存储器 barrier/fence 操作，提交先前的存储器访问并使它们在处理之前对其他线程可见。线程也可以使用 A.4 节中描述的存储器原子操作在它们所共享的存储器上协调工作。

A.5.6　共享存储器

每个 CTA 共享存储器仅对属于该 CTA 的那些线程可见，共享存储器仅在 CTA 创建到它终止这段时间占用存储。共享存储器因此可以驻留在片上。这种方法有很多好处。首先，共享存储器流量不需要竞争访问全局存储器所需要的片外带宽。其次，在片上构建超高带宽存储器结构，以支持每个流多处理器的读写需求是实用的。事实上，共享存储器与流多处理器是紧耦合的。

每个流多处理器包含八个物理线程处理器。在一个共享存储器时钟周期里，每个线程处理器可以处理两个线程相当的指令，因此在每个时钟周期，必须处理 16 个线程相当的共享存储器请求。由于每个线程可以产生它自己的地址，地址是典型唯一的，共享存储器使用 16 个独立编址的 SRAM 体构建。对于通常的访问模式，16 个体对维持吞吐量来说是足够的，但不合理的情况是可能的；例如，所有的 16 个线程可能恰好访问一个 SRAM 体的不同地址。必须可以将一个请求从任意一个线程通路路由到 SRAM 的任意体，因此需要一个 16 乘 16 的互联网络。

A.5.7　局部存储器

每线程的局部存储器是私有存储器，仅对单一线程可见。局部存储器在结构上比线程寄存器文件要大，且一个程序可以将地址计算进局部存储器。为支持局部存储器的大的分配（回收总的分配是每线程分配的数量乘上活动线程的数量），局部存储器位于外部 DRAM 中。

尽管全局和每线程的局部存储器驻留在片外，它们很适于在片上进行缓存。

A.5.8　常量存储器

常量存储器对一个在 SM 上运行的程序来说是只读的（可以通过 GPU 命令写入）。它存在于外部 DRAM 中，且在 SM 中进行了高速缓存。由于一般地一个 SIMT warp 中的大多或所有线程从常量存储器的相同地址进行读取，每个时钟一个单独的地址查找是足够的。常量 cache 设计以向每个 warp 中的线程广播标量值。

A.5.9　纹理存储器

纹理存储器保存有巨大的只读数据阵列。用于计算的纹理具有与用于 3D 图形的纹理相同的属性和容量。尽管纹理一般是二维图像（2D 像素值数组），1D（直线）和 3D（体积）纹理也是可用的。

一个计算程序使用一条 tex 指令引用一个纹理。操作数包括一个命名纹理的标示符，和基于纹理维度的 1、2 个或 3 个坐标。浮点坐标包含小数部分，该小数部分在纹理像素位置中指定了一个采样位置。在结果返回程序之前，非整数坐标调用一个对四个最接近值（对于 2D 纹理来说）的双线性权重插值。

纹理读取在一个流 cache 层次中缓存，该流 cache 层次设计用来在上千个同时线程中优化纹理读取的吞吐量。一些程序使用纹理读取作为缓存全局存储器的一种方式。

A. 5. 10　表面

表面是一个用于一维、二维或三维像素值数组和相关格式类的术语。定义了多种格式：例如，一个像素可以被定义成四个 8 位 RGBA 整数元素，或四个 16 位浮点元素。程序内核不需要知道表面的类型。tex 指令根据表面的格式，将其结果重新计算成浮点。

A. 5. 11　load/store 访问

具有整数字节寻址的 load/store 指令使得使用传统语言如 C 和 C ++ 编写和编译程序成为可能。CUDA 程序使用 load/store 指令进行存储器访问。

为提高存储器带宽和减小开销，当地址落进同一块中且符合对齐规则时，局部和全局 load/store 指令将从同一 warp 中来的单独的并行线程请求合并成一个存储块请求。将独立的小存储器请求合并成大块的请求，与分立请求相比提供了显著的性能提升。巨大的线程数量，加之支持众多未完成的 load 请求，帮助掩盖了在外部 DRAM 中实现的局部和全局存储器的 load-to-use 延迟。

A. 5. 12　ROP

如图 A-2-5 所示，NVIDIA Tesla 构架的 GPU 包含一个可扩展的流处理器阵列（SPA），它执行所有的 GPU 可编程计算；和一个可扩展的存储器系统，它由外部 DRAM 控制和固定功能光栅操作处理器（ROP）组成，其中 ROP 直接在存储器中执行颜色和深度帧缓存操作。每个 ROP 单元都有一个特定的存储器分区与之配对。ROP 分区通过一个内部互联网络由 SM 供给数据。每个 ROP 负责深度和模板测试和更新，还负责颜色混合。ROP 和存储控制器协作实现颜色和深度的无损压缩（最高达 8:1），以减少外部带宽需求。ROP 单元也在存储器上执行原子操作。

A. 6　浮点算术

今天的 GPU 使用 IEEE 754 兼容的单精度 32 位浮点操作（参考第 3 章），在可编程处理器中核执行大多数算术操作。早期 GPU 的定点算术被 16 位、24 位和 32 位浮点，然后是 IEEE 754 兼容的 32 位浮点所超越。GPU 中的一些固定功能逻辑，如纹理过滤硬件，继续使用私有数字格式。近来的 GPU 也提供 IEEE 754 兼容的双精度 64 位浮点指令。

A. 6. 1　支持的格式

IEEE 754 标准［2008］具体说明了浮点算术的基本原理和存储格式。GPU 使用基本格式中的两个进行计算，32 位和 64 位二进制浮点，一般称为单精度和双精度。该标准也指定了一个 16 位二进制存储浮点格式——**半精度**[⊖]。GPU 和 Cg 渲染语言使用窄 16 位半数据格式进行有效的数据存储和移动，而维持了高的动态范围。GPU 在纹理过滤单元和光栅操作单元中，用半精度执行了许多纹理过滤和像素混合计算。由 Industrial Light and Magic［2003］制定的 OpenEXR 高动态范围图像文件格式，在图像计算和运动图片应用中，颜色元素值使用同样的半格式。

A. 6. 2　基本算术

一般的 GPU 可编程核上的单精度浮点操作包括加、乘、**乘加**[⊖]、最小值、最大值、比

⊖ 半精度（half precision）：一个 16 位二进制浮点格式，有 1 位符号位、5 位指数、10 位小数和隐含的整数位。

⊖ 乘加（multiply-add，MAD）：执行一个先乘后加组合操作的单精度浮点指令。

较、设置断言和在整数与浮点数之间的转化。浮点指令通常为非数和绝对值提供源操作数修饰符。

今天，大多 GPU 的浮点加和乘操作与单精度浮点数的 IEEE 754 标准兼容，包含非数值（NaN）和无穷值。浮点加和乘操作使用 IEEE 的取最接近偶数作为默认的舍入模式。为增加浮点指令的吞吐量，GPU 通常使用一个混合的乘加指令（mad）。乘加操作执行截断的浮点乘，接着进行取最近偶数的浮点加。它在一个发射周期中提供两个浮点操作，无需指令调度器分派两条独立的指令，但在加操作之前，计算没有融合且结果没有截断。这与第 3 章和本节后面要讨论的融合的乘加指令有所不同。典型地，GPU 冲刷非正常的源操作数到保留符号零，且目标输出指数范围下溢的结果在舍入后被冲刷到保留符号零。

A.6.3　专用算术

GPU 提供硬件来加速特殊功能计算、属性插值和纹理过滤。特殊功能指令包括余弦、正弦、2 的指数幂、2 的对数、倒数和倒数平方根。属性插值指令提供高效的像素属性的产生，源于平面等式的计算。A.4 节中介绍的**特殊功能单元（SFU）**[⊖]计算特殊功能和插值平面属性［Oberman 和 Siu，2005］。

使用硬件计算特殊功能有几种方法。已经显示，基于增强的最小逼近的二次插值是用硬件实现逼近功能的一种非常有效的方法，包括倒数、倒数平方根、$\log_2 x$、2^x、sin 和 cos。

我们可以总结 SFU 二次插值的方法。对于一个具有 n 位有效数字的二进制输入操作数 X，有效数字被划分为两部分：X_u 是高 m 位，X_l 是低 $n-m$ 位。高 m 位 Xu 用来查询一系列三查找表，以返回三个限定词系数 C_0、C_1、C_2。每个函数的近似需要唯一的一组表。这些系数用于一个给定函数 f(X) 在范围 $X_u <= X < X_u + 2^{-m}$ 的近似，通过计算表达式：

$$f(X) = C_0 + C_1 X_l + C_2 X_l^2$$

每个函数估计的精确度从 22 到 24 有效数字位变化。示例函数统计在图 A-6-1 中给出。

功能	输入间隔	精度（取整）	ULP* error	% exactly 取整	单调
1/x	[1, 2)	24.02	0.98	87	Yes
1/sqrt(x)	[1, 4)	23.40	1.52	78	Yes
2^x	[0, 1)	22.51	1.41	74	Yes
$\log_2 x$	[1, 2)	22.57	N/A**	N/A	Yes
sin/cos	[0, π/2)	22.47	N/A	N/A	No

*ULP: unit in the last place. **N/A: not applicable.

图 A-6-1　特定功能近似统计

适合于 NVIDIA GeForce 8800 特殊功能单元（SFU）。

IEEE 754 标准为除法和平方根指定了精准的舍入要求，然而，对许多 GPU 应用来说，准确的遵循是不需要的。对于那些应用来说，与末尾位的精确度相比，更高的计算吞吐量是更为重要的。对于 SFU 特殊功能，CUDA 数学库既提供了完全精确函数，也提供了具有 SFU 指令精度的快速函数。

GPU 中另外的专用算术操作是属性插值。通常为原始顶点指定的关键属性组成需渲染的一幅场景。示例属性是颜色、深度和纹理坐标。由于需要在每个像素位置决定属性值的缘故，这些

⊖ 特殊功能单元（special function unit，SFU）：一个用于计算特殊功能和插值平面属性的硬件单元。

属性必须在（x，y）屏幕空间中进行插值。在一个（x，y）平面的一个给定属性 U 的值可以使用如下平面方程表达：

$$U(x,y) = A_u x + B_u y + C_u$$

其中 A、B、C 是与每个属性 U 关联的插值参数。插值参数 A、B、C 均使用单精度浮点数表示。

在像素渲染处理器中，函数求解器和属性插值器的需求确定后，即可进行 SFU 的设计，以简单有效地执行这两种功能。这两种函数使用乘积操作的和插值产生结果，两个函数中进行求和的项数非常相似。

纹理操作

纹理映射和过滤是 GPU 中另一组重要的专用浮点算术操作。用于纹理映射的操作包括：

1）为当前平面像素（x，y）接收纹理地址（s，t），其中 s 和 t 为单精度浮点数。

2）计算细节的级别以鉴别正确的纹理 **MIP-map**[⊖]级别。

3）计算三线条插值片段。

4）为选定的 MIP-map 级缩放纹理地址（s，t）。

5）访问存储器并取回要求的纹理像素（纹理元素）。

6）在纹理像素上执行过滤操作。

纹理映射需要相当数量的浮点计算用于全速操作，这些浮点计算大多在 16 位的半精度上执行。例如，GeForce 8800 除它的常规 IEEE 单精度浮点指令外，为纹理映射指令，极限情况下可产生大约 500 GFLOPS 的私有格式浮点计算。关于纹理映射和过滤的更多细节，参考 Foley 和 van Dam［1995］。

A.6.4　性能

浮点加和乘算术硬件是完全流水的，对延迟进行了优化，以在延迟和面积间进行权衡。流水以后，特定功能的吞吐量比浮点加和乘操作要少。在现代 GPU 中，一个 SFU 被四个 SP 核所共享，典型的性能是，特殊功能的速度是吞吐量的四分之一。相反，对于相似功能，CPU 典型地具有相当低的吞吐量，如除法和平方根，即使具有更为精确的结果。通常，属性插值硬件是完全流水的，以全速地进行像素渲染。

A.6.5　双精度

较新的 GPU，如 Tesla T10P，也以硬件的方式支持 IEEE 754 64 位双精度操作。双精度下的标准浮点算术操作包括加、乘和不同浮点和整数格式间的转化。2008 版的 IEEE 754 标准包含了融合的乘加操作（FMA）规范，如第 3 章中描述。FMA 操作执行一个浮点乘，然后跟着一个加操作，使用单精度舍入。融合乘和加操作在中间计算中保持全精度。这种行为使得更为精确的浮点计算，包括乘积累加、带小数点乘、矩阵乘和多项式求值成为可能。FMA 指令也使得软件高效地实现准确舍入除法和平方根，而无需硬件除或平方根单元成为可能。

双精度硬件 FMA 单元实现 64 位加、乘、转化和 FMA 操作本身。双精度 FMA 单元的结构，提供了在输入和输出上进行全速非规格数处理的支持。图 A-6-2 展示了一个 FMA 单元的块图。

⊖ MIP-map：拉丁短语 multum in parvo 的缩写，即小而丰富。一个 MIP-map 包含预先计算好的不同分辨率的图像，用于增加渲染速度并减少处理。

如图 A-6-2 所示，A 和 B 的有效数字相乘形成一个 106 位的乘积，以进位保留形式保存结果。并行地，53 位加数 C 有条件地反转并对齐到 106 位乘积。106 位乘积的和与进位结果通过一个 161 位宽的进位保留加法器（CSA），与一个对齐的加数相加。在一个进位传播加法器中，进位保留输出这时加到一起，以产生一个未舍入、无冗余、二的补码形式的结果。结果再有条件地求补，以便返回一个符号数值形式的结果。对求补的结果进行规格化，然后进行舍入以匹配目标格式。

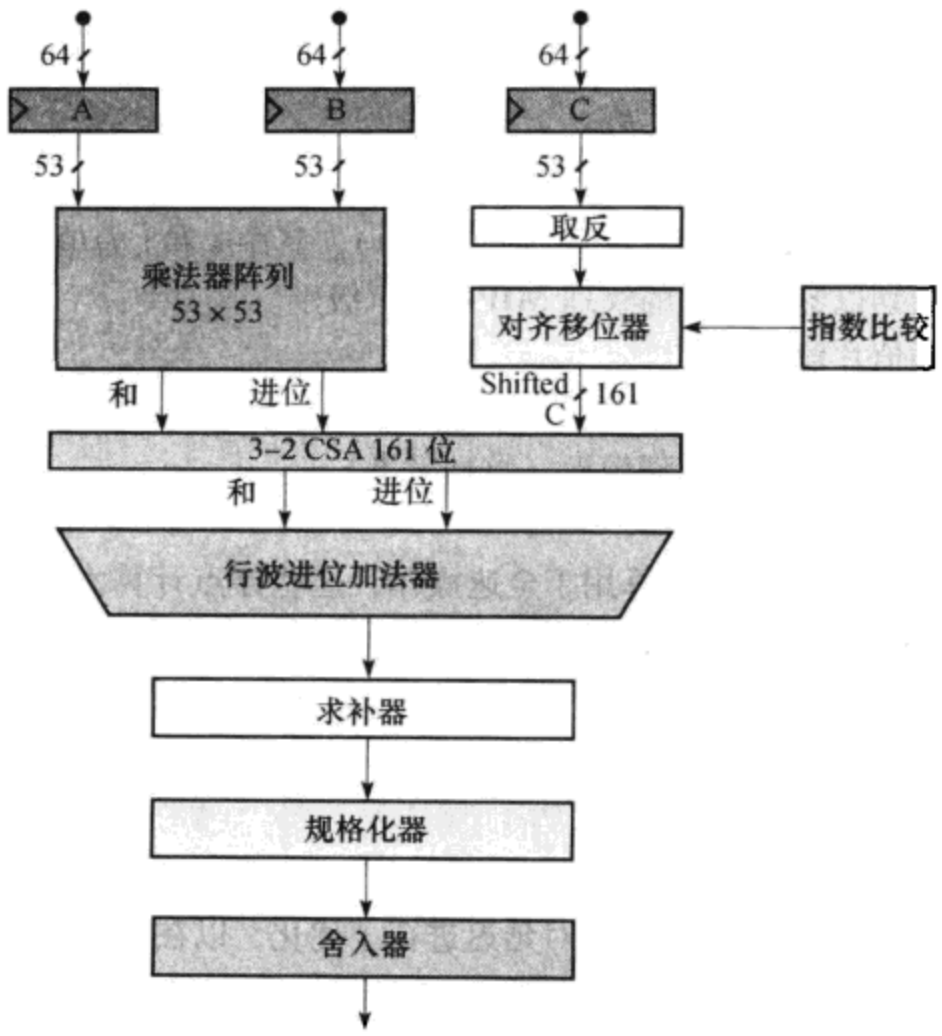


图 A-6-2　双精度融合乘加（FMA）单元
硬件实现双精度浮点 A × B + C。

A.7　资料：NVIDIA GeForce 8800

NVIDIA GeForce 8800 于2006 年11 月上市，是一个统一的顶点和像素处理器设计，也支持使用 CUDA 并行编程模型编写的 C 语言的并行计算应用。它是 A.4 节和 Lindholm、Nickolls、Oberman 和 Montrym［2008］中描述的 Tesla 统一图形和计算架构的首次实现。一系列 Tesla 架构 GPU 满足了笔记本电脑、桌面电脑、工作站和服务器的不同需求。

A.7.1　流处理器阵列（SPA）

图 A-7-1 中展示的 GeForce 8800 GPU 包含组织成 16 个流多处理器（SM）的 128 个流处理器（SP）核。每个纹理/处理器集群（TPC）中的两个 SM 共享一个纹理单元。一个由 8 个 TPC 组成的阵列构成流处理器阵列（SPA），SPA 执行所有图形渲染程序和计算程序。

主机接口单元通过 PCI-Express 总线与主机 CPU 通信，检查命令的连贯性，并完成上下文切换。输入装配单元收集几何原语（点、线和三角形）。工作分配模块分发顶点、像素和计算线程阵列到 SPA 中的 TPC。TPC 执行顶点、几何渲染程序和计算程序。输出集合数据被发送到视点/修剪/建立/光栅/Z 剔除模块，光栅化成像素段，之后重新分发到 SPA 中，以执行像素

渲染程序。渲染后的像素被 ROP 单元通过内部互联网络发送以进行处理。网络也将从 SPA 来的纹理存储器读请求路由到 DRAM，并将从 DRAM 通过一个 2 级 cache 读出的数据路由回 SPA。

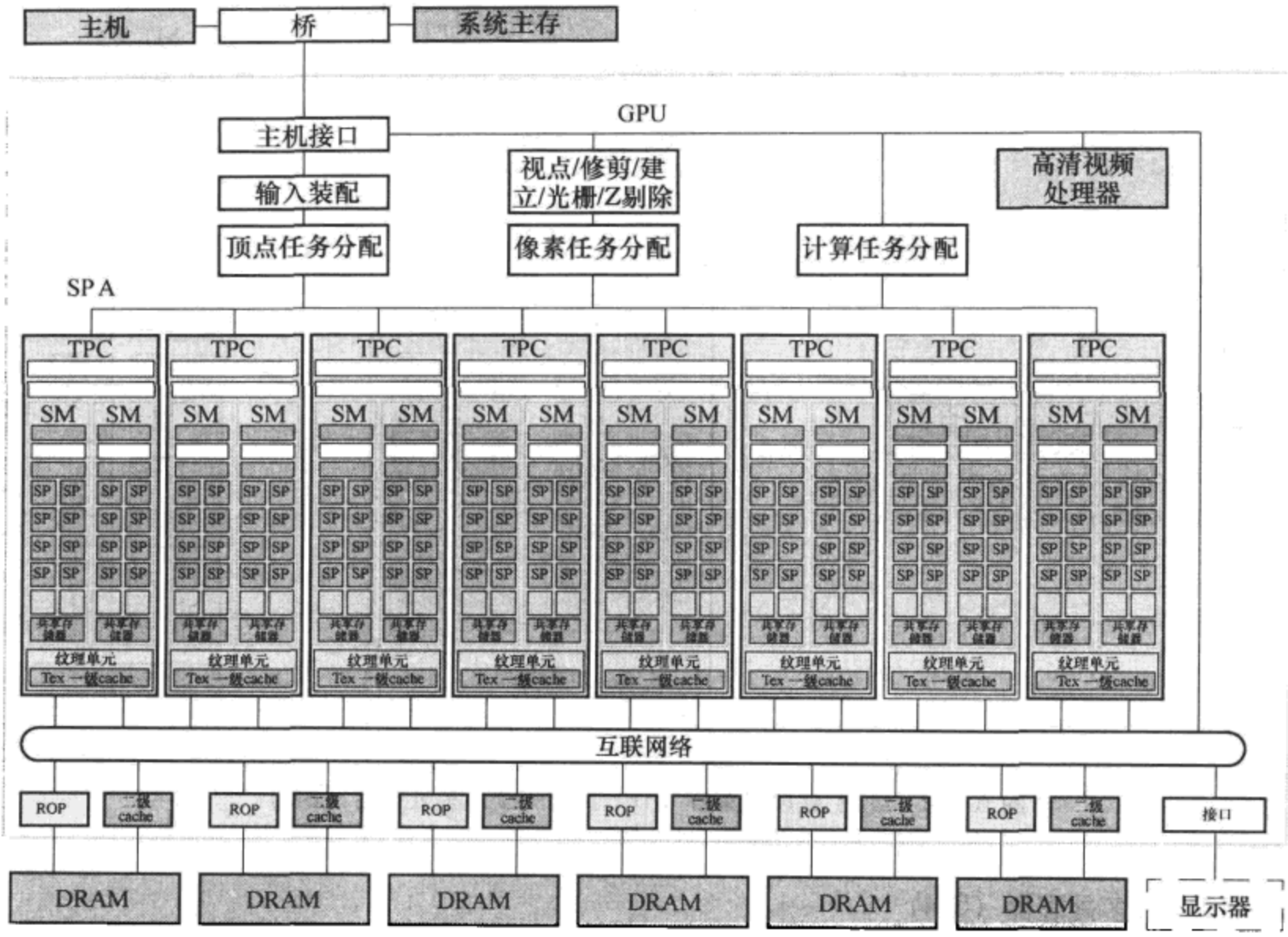


图 A-7-1　NVIDIA Tesla 统一的图形和计算 GPU 构架

GeForce 8800 具有 128 个流处理器（SP）核在 16 个流多处理器（SM）中，被组织成 8 个纹理/处理器簇（TPC）。处理器通过一个内部互联网络与 6 个 64 位宽的 DRAM 分区相连。其他的 GPU 改变 SP 核、SM、DRAM 分区和其他单元的数量，实现 Tesla 架构。

A.7.2　纹理/处理器簇（TPC）

每个 TPC 包含一个几何控制器、一个 SM 控制器（SMC）、两个流多处理（SM）和一个纹理单元，如图 A-7-2 所示。

几何控制器通过指挥 TPC 中所有原语和顶点属性和拓扑流，将逻辑图形顶点流水线映射成物理 SM 上的不断循环。

SMC 控制多个 SM，仲裁共享纹理单元、load/store 路径和 I/O 路径。SMC 同时服务三个图形工作负载：顶点、几何和像素。

纹理单元每时钟周期为一个顶点、几何、像素方块或四个计算线程处理一条纹理指令。纹理指令的源是纹理坐标，输出是加权的采样值，典型地是一个 4 元素（RGBA）浮点颜色。纹理单元是深度流水的。尽管它包含一个流 cache 以捕捉过滤的局部性，它的流命中与无停顿的缺失混合。

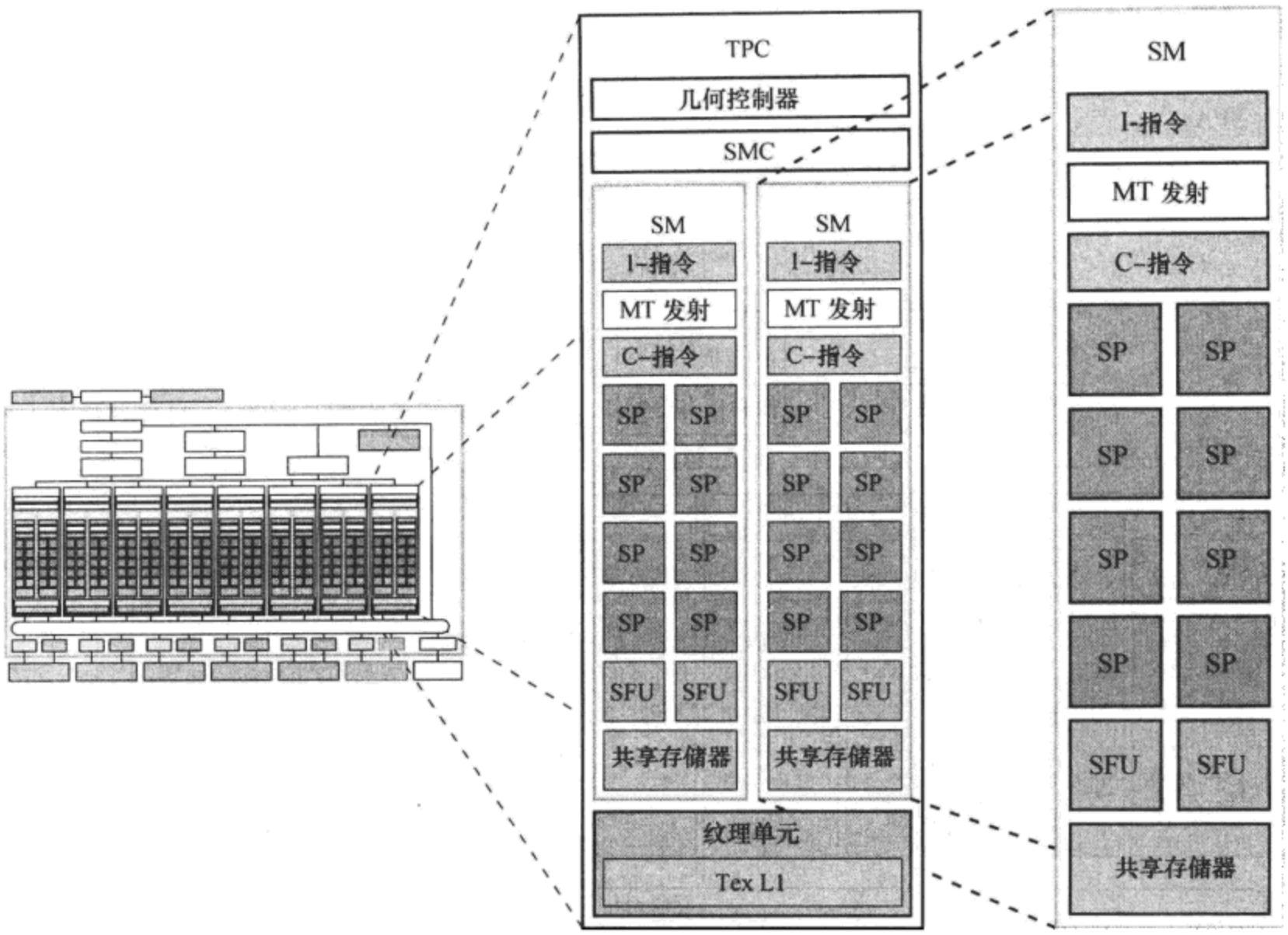


图 A-7-2　纹理/处理器集群（TPC）和一个流多处理器（SM）

每个 SM 有 8 个流处理器（SP）核、两个 SFU 和一个共享存储器。

A.7.3　流多处理器（SM）

SM 是一个统一的图形和计算多处理器，执行顶点、几何、像素段渲染程序和并行计算程序。SM 由 8 个 SP 线程处理器核，两个 SFU，一个多线程取指和发射（MT 发射），一个指令 cache，一个只读常量 cache 和一个 16 KB 读写共享存储器组成。它为单独线程执行标量指令。

GeForce 8800 极端情况下使 SP 核和 SFU 运行在 1.5 GHz，可达到每 SM 36 GFLOPS 的峰值速度。为了对功耗和面积进行有效的优化，一些 SM 的非数据路径单元以 SP 时钟频率的半数运行。

在运行几个不同的程序时，为了有效地执行数百个并行程序，SM 是硬件多线程的。它用硬件管理和执行最高达 768 个并发线程，调度开销为 0。每个线程拥有它自己的线程执行状态，可以以独立的代码路径执行。

一个 warp 由最多 32 个相同类型的线程组成，类型可为顶点、几何、像素或计算。SIMT 设计，曾在 A.4 节中描述，有效地在 32 个线程间共享 SM 取指和发射单元，但需要活动线程的一个满的 warp，以获得全部的性能效率。

SM 同时调度和执行多个 warp。每个发射周期，调度器从 24 个 warp 中选择一个，执行一条 SIMT warp 指令。一条发射后的 warp 指令以 4 组 8 线程在 4 个处理器周期上执行。SP 和 SFU 单元独立地执行指令，并通过在轮流的时钟周期上在它们之间发射指令，调度器可以保证它们均满负荷工作。一个记分板用来限定每周期中每个 warp 的发射。指令调度器对所有的就绪 warp 按优先级排序，并选择一个具有最高优先级的进行发射。优先级依据 warp 类型、指令类型和对 SM 中所有的 warp 执行“公平”进行设置。

SM 以多个并发 warp 的方式执行协作线程组（CTA），它们访问一个为 CTA 动态申请的共享

存储器区域。

A.7.4　指令集

线程执行标量指令，而不像先前的 GPU 向量指令架构。标量指令简单且易于编译。纹理指令仍旧是基于向量的，接受一个源坐标向量并返回一个经过滤的颜色向量。

基于寄存器的指令集包括所有的浮点和整数算术、超越、逻辑、流控制、存储器 load/store 和图 A-4-3 PTX 指令表中列出的纹理指令。存储器 load/store 指令使用整数字节寻址，使用寄存器加偏移地址算法。为了进行计算，load/store 指令访问三个可读写存储器空间：每个线程的局部存储器、私有和临时数据；用于 CTA 中的线程进行低延时每 CTA 数据共享的共享存储器；用于在所有线程间进行数据共享的全局存储器。计算程序使用快速的栅障同步指令 `bar.sync` 同步 CTA 中的线程，这些线程通过共享和全局存储器互相通信。最近的 Tesla 架构 GPU 实现了 PTX 原子存储器操作，这使得并行简化和并行数据结构管理更为便利。

A.7.5　流处理器（SP）

如 A.4 节中介绍的那样，多线程 SP 核是主要的线程处理器。它的寄存器文件为最多可达 96 个线程（比 A.4 节中的示例 SP 更多的线程）提供 1024 个标量 32 位寄存器。它的浮点加和乘操作与单精度浮点数的 IEEE 754 标准兼容，包括非数值（NaN）和无穷值。加和乘操作使用 IEEE 的最接近偶数舍入作为默认的舍入模式。该 SP 核也实现了所有的 32 位和 64 位整数算术、比较、转换和图 A-4-3 中的逻辑 PTX 指令。该处理器是完全流水的，且对延时进行了优化以平衡延迟和面积。

A.7.6　特殊功能单元（SFU）

SFU 支持超越函数和平面属性插值两者的计算。如 A.6 节中所述，它使用基于增强最小逼近的二次插值，以每时钟周期一个结果的速度，来近似倒数、倒数平方根、$\log_2 x$、2^x、sin 和 cos 函数。SFU 也支持以每时钟周期四个采样的像素属性插值，如颜色、深度和纹理坐标。

A.7.7　光栅化

从 SM 来的几何原语以它们最初的轮询输入次序到达视点/修剪/建立/光栅/Z 剔除模块。视点和修剪单元将原语修剪成视点平截头体，和任何使能的用户修剪平面，然后将顶点转换成屏幕（像素）空间。

留下的元素这时前往建立单元，在那儿为光栅产生边缘方程。一个粗光栅化阶段产生所有至少部分位于原语内部的像素瓦片。Z 剔除单元维持一个层次化的 z 表面，如果确定一个像素瓦片被先前绘制的像素所遮蔽，则拒绝它们。拒绝速率最高可达每时钟 256 像素。从 Z 剔除中保留下来的像素这时进入一个细光栅化阶段产生详细的覆盖信息和深度值。

深度测试和更新可以在段渲染之前或之后进行，根据当前状态决定。SMC 聚集保留下来的像素到 warp 中，由一个运行着当前像素渲染的 SM 处理。SMC 然后将保留下来的像素和相关的数据发送到 ROP。

A.7.8　光栅操作处理器（ROP）和存储系统

每个 ROP 都有一个特定的存储器分区与之相配。对于每一个由像素渲染程序发射的像素段，ROP 执行深度和模板测试与更新，同时进行颜色混合与更新。采用无损颜色压缩（最高达8:1）和深度压缩（最高达 8:1）减小 DRAM 带宽需求。每个 ROP 峰值速度为每时钟 4 个像素，并支持 16 位浮点和 32 位浮点 HDR 格式。ROP 支持在禁止颜色写时双速率的深度处理。

反混淆支持包括高达 16 × 的多采样和超采样。覆盖采样反混淆（CSAA）算法计算并存储高

达 16 个采样的布尔覆盖，并压缩冗余颜色、深度和模板信息到 4 或 8 采样的存储器空间和带宽，以提高性能。

DRAM 存储器数据总线宽度为 384 针，安排成 6 个独立的分区，每个 64 针。每个分区支持在高达 1.0 GHz 频率上，双数据率 DDR2 和面向图形的 GDDR3 协议，产生的带宽约为每分区 16 GB/s或 96 GB/s。

存储控制器支持较宽范围的 DRAM 时钟频率、协议、器件密度和数据总线宽度。纹理和 load/store 请求可以在任何 TPC 和任何存储分区发生，因此有一个互联网络用于路由请求和回应。

A.7.9　可扩展性

Tesla 统一架构是为可扩展而设计的。改变 SM、TPC、ROP、cache 和存储器分区的数量，为 GPU 市场区域的不同性能和成本提供了合适的均衡。可扩展连接互联（SLI）连接多个 GPU，提供进一步的可扩展性。

A.7.10　性能

GeForce 8800 在极限情况下，为达到 576 GFLOPS 的理论操作峰值，以 1.5 GHz 的时钟驱动 SP 线程处理器核和 SFU。GeForce 8800 GTX 具有 1.35 GHz 的处理器时钟和相应的 518 GFLOPS 的峰值。

接下来的三节，以三个不同的应用——密集线性代数、快速傅里叶变换和排序比较 GeForce 8800 GPU 和一个多核 CPU。GPU 程序和库是编译过的 CUDA C 代码。CPU 代码使用单精度多线程 Intel MKL 10.0 库来补充 SSE 指令集和多核。

A.7.11　密集线性代数性能

密集线性代数计算在众多应用中都是基本的计算。Volkov 和 Demmel［2008］给出了在单精度密集矩阵-矩阵乘法（SGEMM 例程）和 LU、QR 和 Cholesky 矩阵分解方面，GPU 和 CPU 的性能结果。图 A-7-3 对 GeForce 8800 GTX GPU 和 4 核 CPU，在 SGEMM 密集矩阵-矩阵乘法上的 GFLOPS 速率进行了比较。图 A-7-4 对一个 GPU 和一个 4 核 CPU 在矩阵分解上的 GFLOPS 速率进行了比较。

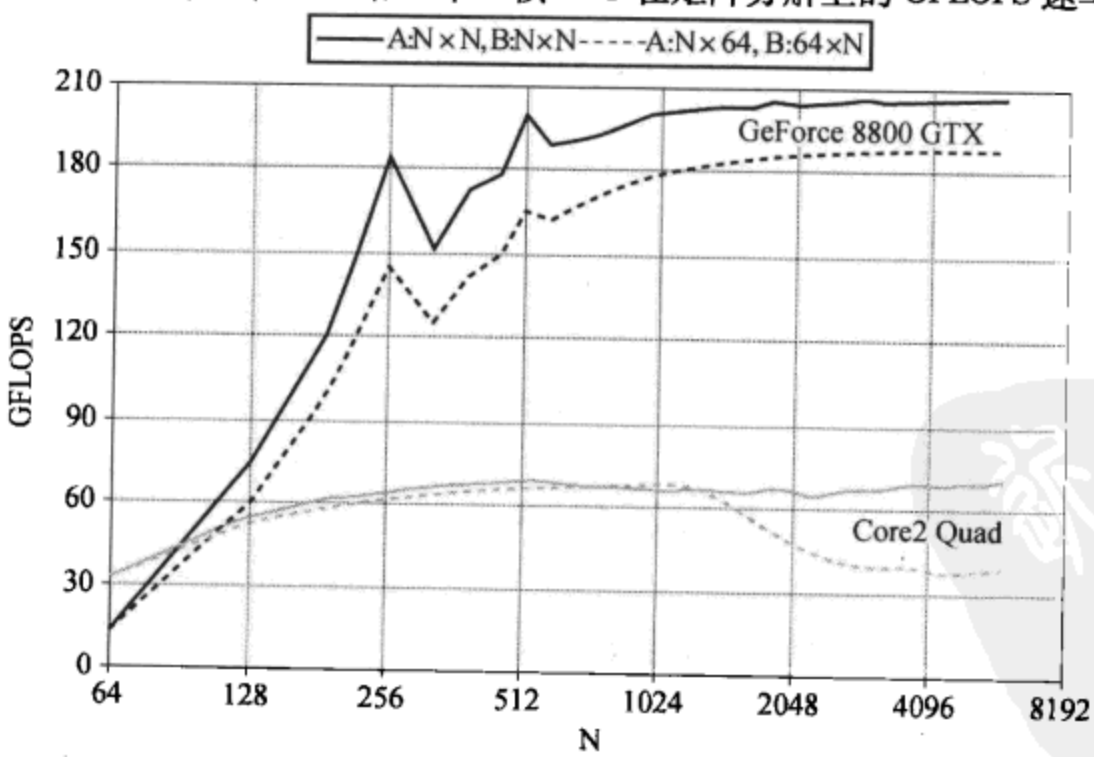


图 A-7-3　SGEMM 密集矩阵-矩阵乘法性能速率

图中展示了单精度正方 N×N 矩阵乘（实线）和稀疏 N×64 和 64×N 矩阵（点线）的 GFLOPS 速率。根据 Volkov 和 Demmel［2008］的图 6 改编。黑线是 1.35 GHz GeForce 8800 GTX，使用 GPU 存储器中的 Volkov 的矩阵 SGEMM 代码（现在在 NVIDIA CUBLAS 2.0 中）。灰线是 4 核 2.4 GHz Intel Core2 Quad Q6600，64 位 Linux，使用 CPU 存储器中 Intel MKL 10.0 矩阵代码。

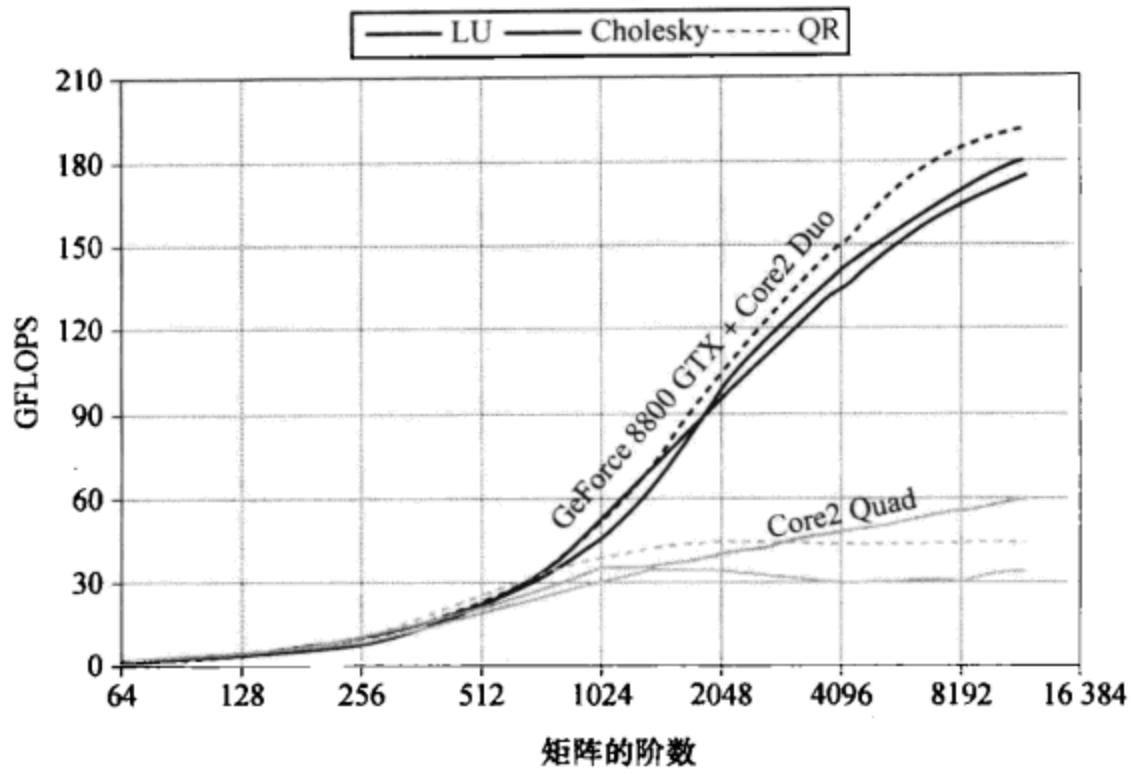


图 A-7-4　密集矩阵分解性能速率

图中展示了单独使用 GPU 和 CPU 在矩阵分解方面获得的 GFLOPS 性能。根据 Volkov 和 Demmel［2008］的图 7 改编。黑线是 1.35 GHz NVIDIA GeForce 8800 GTX，CUDA 1.1，Windows XP，连接到一个 2.67 GHz Intel Core2 Duo E6700 Windows XP，包括所有的 CPU-GPU 数据传输时间。灰线是 4 核 2.4 GHz Intel Core2 Quad Q6600，64 位 Linux，Intel MKL 10.0。

由于 SGEMM 密集矩阵-矩阵乘法和类 BLAS3 例程都是大批的矩阵分解中的工作，它们的性能设定了因子分解速率的上限。由于矩阵阶数的增加超过了 200 ~ 400，因子分解问题变得足够大，以至于 SGEMM 可以补充 GPU 的并行度，并克服 CPU-GPU 系统和复制开销。Volkov 的 SGEMM 矩阵-矩阵乘法获得了 206 GFLOPS，大约 60% 的 GeForce 8800 GTX 峰值乘加速率，而 QR 因子分解达到了 192 GFLOPS，约为 4 核 CPU 的 4.3 倍。

A.7.12　FFT 性能

在众多应用中都用到了快速傅里叶变换。大的变换和多维度变换被分解成一批小的 1D 变换。图 A-7-5 对 1.35 GHz GeForce 8800 GTX（2006 年下半年生产）和 2.8 GHz 4 核 Intel Xeon

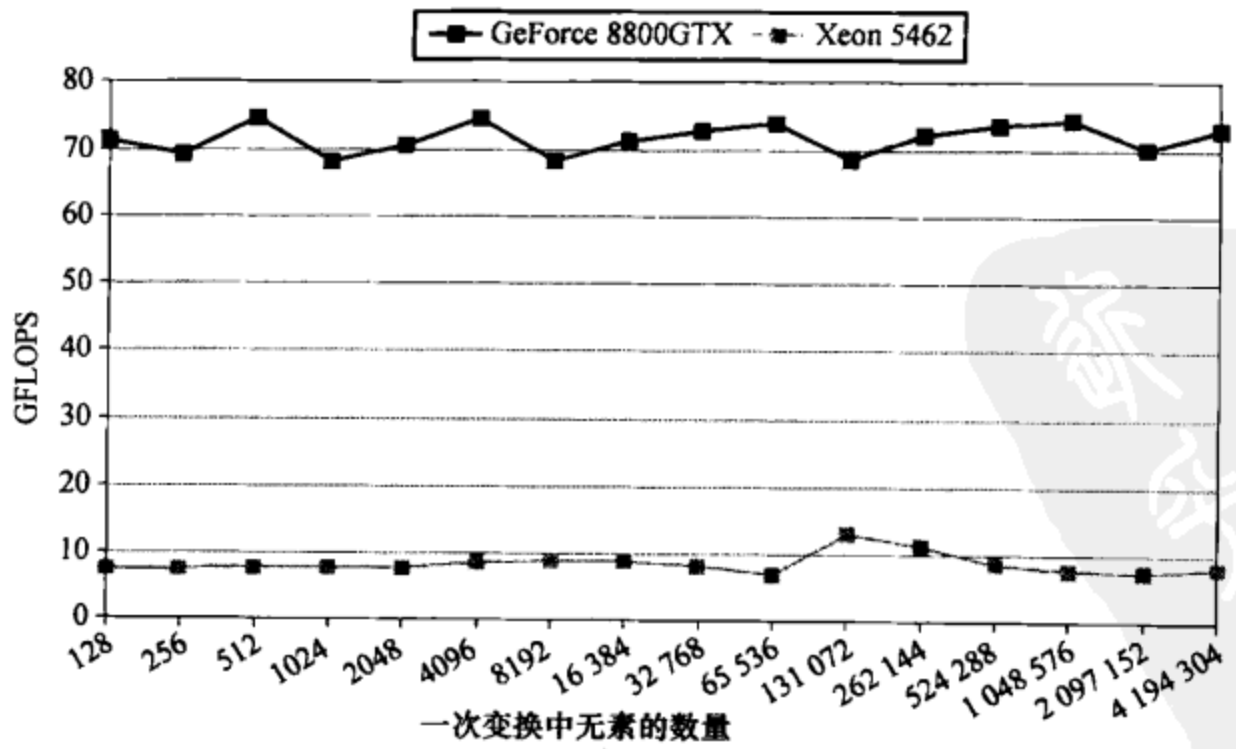


图 A-7-5　快速傅里叶变换吞吐量性能

图中对批量一维复杂 FFT 变换在 1.35 GHz GeForce 8800 GTX 和 4 核 2.8 GHz Intel Xeon E5462 系列（代号“Harpertown”），6 MB 二级缓存，4 GB 内存，1600 FSB，红帽子 Linux，Intel MKL 10.0 上的性能进行了比较。

E5462 系列（代号“Harpertown”，2007 年下半年生产）的原地 1D 复杂单精度 FFT 性能进行了比较。CPU 性能使用四线程的 Intel 数学核心库（MKL）10.0 FFT 进行测量。GPU 性能使用 NVIDIA CUFFT 2.1 库和一批十六进制频分 FFT 进行测量。CPU 和 GPU 的吞吐量性能均使用批量 FFT，一批的大小为 $2^{24}/n$，其中 n 是转换的大小。因此，每个转换大小的负载是 128 MB。为决定 GFLOPS 速率，每次转换的操作数采用 $5n\log_2 n$。

A.7.13　排序性能

与刚刚讨论的应用相反，在并行线程中排序需要更多的真实坐标，且并行扩展相应更难获得。不过，多种著名的排序算法可以被高效地并行化，以在 GPU 上良好地运行。Satish 等[2008] 详细说明了 CUDA 中排序算法的设计，且他们报告的基数排序结果概括如下。

图 A-7-6 对 GeForce 8800 Ultra 和 8 核 Intel Cloverdown 系统的并行排序性能进行了比较，它们都是 2007 年上半年生产的。CPU 核分布于两个物理插座之间。每个插座包含一个具有双 Core2 芯片的多片模块，且每片具有 4 MB 的二级缓存。所有的排序例程都按键-值对进行排序设计，且键和值两者都是 32 位整数。被研究的主要算法是基数排序，尽管在比较中也包括了 Intel 线程构建块提供的基于快速排序的 `parallel_sort()` 程序。基于 CPU 的两个基数排序代码中的一个，仅使用标量指令集，其余利用手工保守优化的汇编例程，这些汇编例程利用 SSE2 SIMD 向量指令。

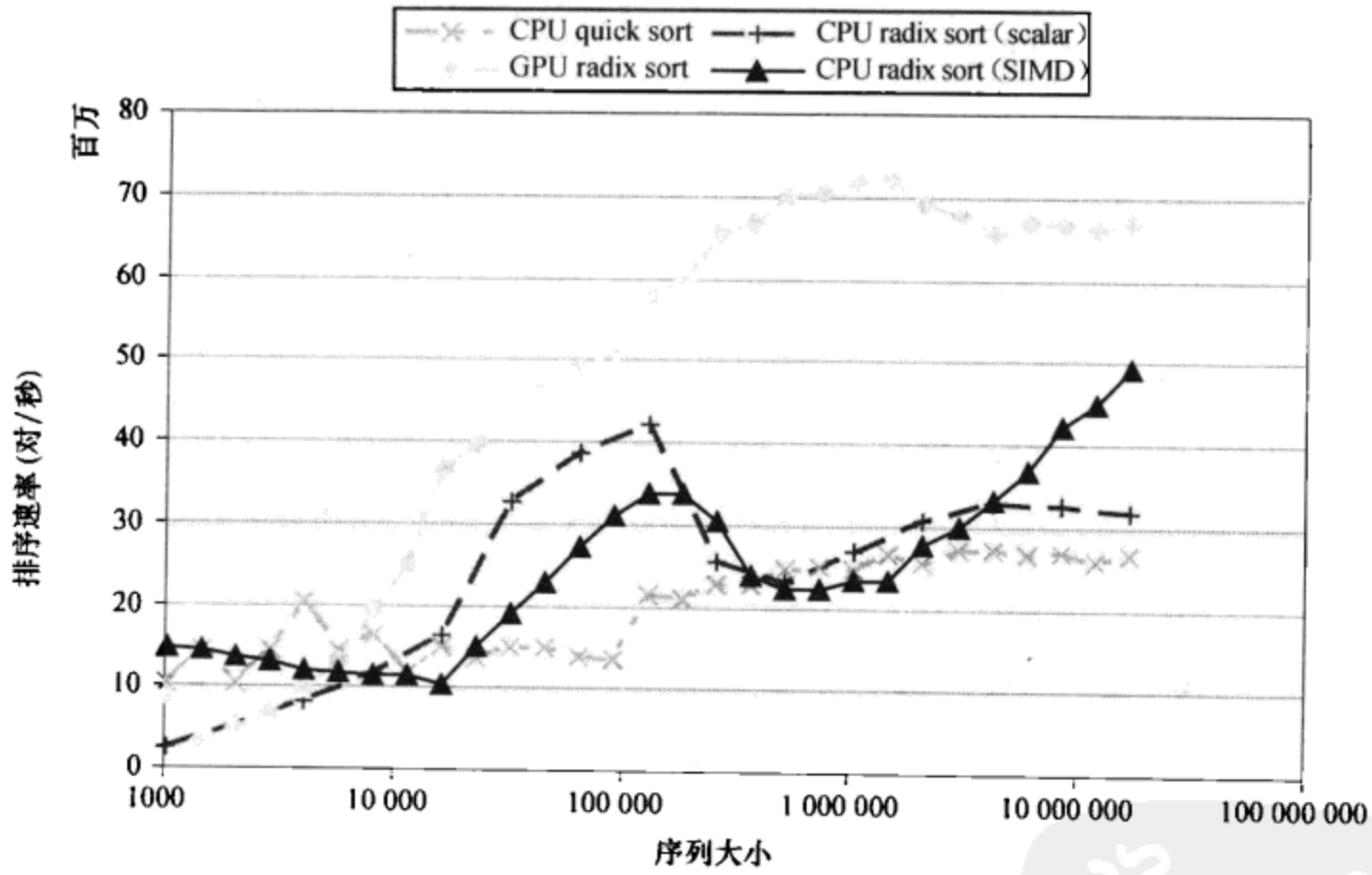


图 A-7-6　并行排序性能

该图对在 GeForce 8800 Ultra 和 8 核 2.33 GHz Intel Core2 Xeon E5345 系统上实现的并行基数排序的性能进行了比较。

图形说明了进行一系列排序的速率——定义成排序的元素数除以排序时间。从图 A-7-6 可以直观地看出，对于所有 8 K 和更大的元素序列，GPU 基数排序获得了最高的排序速率。在这个范围中，与使用 8 个可用的 CPU 核相比，平均比快速排序例程快 2.6 倍，大约比基数排序例程快 2 倍。CPU 基数排序性能变化很大，似乎是其全局交换、cache 局部性较差的原因。

A.8　资料：将应用映射到 GPU

多核 CPU 和众核 GPU 的到来意味着主流处理器芯片现在都是并行系统。而且，它们的并行

性仍以摩尔定律持续扩展。挑战是开发主流视觉计算和高性能计算应用程序，透明地扩展它们的并行度以利用增加的处理器核数量，像许多3D图形应用透明地扩展它们的并行度到核心数量广泛变化的GPU上。

本节给出了使用CUDA映射可扩展并行计算应用到GPU上的例子。

A.8.1　稀疏矩阵

很多种类的并行算法可以使用CUDA以相当直截了当的方式编写，甚至当包含的数据结构不是简单规则的网格时也可以。稀疏矩阵向量乘法（SpMV）是重要的数字构建块中一个很好的例子，通过CUDA提供的抽象能力，这些主要的数字构建块可以被相当直接地并行化。我们下面讨论的核心是，当与提供的CUBLAS向量例程结合时，使循环求解的编写（如共轭梯度方法）变得更为直接。

$n \times n$ 的稀疏矩阵是指，矩阵中非零元素的个数 m 仅占总元素个数的一小部分。稀疏矩阵表示法寻求存储一个矩阵的非零元素。由于相当典型，一个 $n \times n$ 稀疏矩阵将仅包含 $m = O(n)$ 个非零元素，这表明可在存储空间和处理时间上有实质的节省。

一般无结构稀疏矩阵的最普通表示方法之一是压缩的稀疏行（CSR）表示法。矩阵 A 中 m 个非零元素以行优先的顺序存储到数组 Av 中。第二个数组 Aj 为 Av 的每个条目记录相应的列索引。最后，一个具有 $n+1$ 元素的数组 Ap 记录前面数组每行的范围；Aj 和 Av 中行 i 的条目从索引 Ap[i]向上扩展得到，但却不包括索引 Ap[i+1]。这意味着 Ap[0]将总是0，Ap[n]将总是矩阵中非零元素的个数。图 A-8-1 给出了一个简单矩阵的CSR表示的例子。

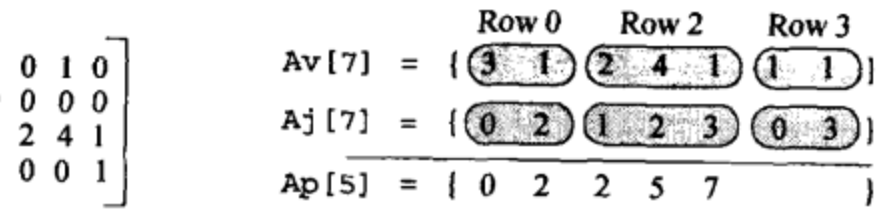


图 A-8-1　压缩稀疏行（CSR）矩阵
a）示例矩阵 A；b）示例矩阵的 CSR 表示

给定一个CSR形式的矩阵 A 和一个向量 x，使用图 A-8-2 中展示的 multiply_row()程序，我们可以计算一个单行乘积 $y = Ax$。那么计算完全乘积仅仅是在所有行上循环并使用 multiply_row()计算那行的结果，multiply_row()的串行C代码在图 A-8-3 中给出。

```
float multiply_row(unsigned int rowsize,
                   unsigned int *Aj, // 列索引
                   float *Av,        // 非零项
                   float *x)         // RHS向量
{
    float sum = 0;

    for(unsigned int column=0; column<rowsize; ++column)
        sum += Av[column] * x[Aj[column]];

    return sum;
}
```

图 A-8-2　一个单行稀疏矩阵向量乘的串行C代码

```
void csrmul_serial(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
    for(unsigned int row=0; row<num_rows; ++row)
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                              Av+row_begin, x);
    }
}
```

图 A-8-3 稀疏矩阵向量乘的串行代码

该算法可以很容易地转化成一个并行 CUDA 内核。我们仅将 csrmul_serial()中的循环展开到众多并行线程上。每个线程将恰好计算输出向量 y 中的一行。该内核的代码在图 A-8-4 中给出。注意，它看上去与 csrmul_serial()程序中的串行循环非常相似，但实际上有两点不同。首先，每线程的行索引从为每线程分配的块和线程索引计算得到，消除了 for 循环；其次，我们有一个条件，计算一行的乘积仅当行索引在矩阵的界限内（这是必要的，因为行数 n 无需是运行内核时块大小的倍数）。

```
__global__
void csrmul_kernel(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
    unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;

    if( row<num_rows     )
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                              Av+row_begin, x);
    }
}
```

图 A-8-4 稀疏矩阵向量乘的 CUDA 版本

假定矩阵数据结构已经复制到了 GPU 的设备存储器中，运行该内核将会如下：

```
unsigned int blocksize = 128;    // 最多 512 的任何大小
unsigned int nblocks = (num_rows + blocksize - 1) / blocksize;
csrmul_kernel <<<nblocks.blocksize>>>(Ap, Aj, Av, num_rows, x, y);
```

我们在此看到的模式是非常普通的一个。最初的串行算法是一个循环，它的迭代彼此独立。这样的循环可以很容易地并行化，通过简单地指派循环中的一个或多个迭代到每个并行线程。CUDA 提供的编程模型使得表示该类的并行极为直接。

这个将计算分解成独立工作块的普通策略，和更加明确的分解独立循环迭代，并不是 CUDA 所独有的。这是在包括 OpenMP 和 Intel 的线程构建块的多种并行编程系统中，以一种形式或其

他形式使用的公共方法。

A.8.2　在共享存储器中进行缓存

上面概述的 SpMV 算法是相当简化的。在 CPU 和 GPU 代码中都可以进行多种优化，以提高性能，包括循环展开、矩阵重排和寄存器分块。并行内核也可以根据 Sengupta 等［2007］提出的数据并行扫描操作重新实现。

CUDA 所揭露的一个重要的体系结构特征是，每块共享存储的存在，该共享存储是一个小的具有极低延迟的片上存储器。利用该存储器的优势，可以获得实质的性能提升。这样做的一个普通方法是将共享存储器作为一个软件管理的 cache 来保持频繁重用的数据。使用共享存储的修改在图 A-8-5 中说明。

```
__global__
void csrmul_cached(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   const float *x, float *y)
{
    // Cache the rows of x[] corresponding to this block.
    __shared__ float cache[blocksize];

    unsigned int block_begin = blockIdx.x * blockDim.x;
    unsigned int block_end   = block_begin + blockDim.x;
    unsigned int row         = block_begin + threadIdx.x;

    // Fetch and cache our window of x[].
    if( row<num_rows)  cache[threadIdx.x] = x[row];
    __syncthreads();

    if( row<num_rows     )
     {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];
        float sum = 0, x_j;

        for(unsigned int col=row_begin; col<row_end; ++col)
        {
            unsigned int j = Aj[col];

            // Fetch x_j from our cache when possible
            if( j>=block_begin && j<block_end )
                x_j = cache[j-block_begin];
            else
                x_j = x[j];

            sum += Av[col] * x_j;
         }

        y[row] = sum;
    }
}
```

图 A-8-5　稀疏矩阵向量乘法的共享存储器版本

在稀疏矩阵乘法的上下文中，我们观察到 A 的数行可能使用一个特殊的矩阵元素 x[i]。在许多普通场合，特别地在矩阵重排后，使用 x[i] 的行将是与行 i 接近的行。我们可以因此实现一个简单的缓存方案，并期待获得一些性能好处。处理行 i 所有 j 的线程块将把 x[i]行所有 x[j] 装进共享存储器。我们将展开 multiply_row()循环并在任何可能的时候从 cache 中预取 x 的元素。

结果代码如图 A-8-5 所示。共享存储器也可以用来进行其他优化，如从一个临近线程取 Ap[row+1] 而不是再次从存储器中取。

由于 Tesla 架构提供显式管理的片上共享存储器，而不是隐含激活硬件 cache，增加此类优化是相当普遍的。尽管这可能给程序员增加了一些额外的开发负担，但其负担相对较小，却能得到实质的性能提升。在上面的示例中，甚至简单使用共享存储器在表示由 3D 表面网格得来的矩阵上就获得了大约 20% 的性能提升。显式管理的存储器有取代隐式 cache 的可能，并且它还具有缓存和预取策略可根据应用需求进行裁剪的优点。

这些是相当简单的内核，目的是阐明编写 CUDA 程序的基本技术，而不是怎样获得最大的性能。众多可能的优化方法都是可用的，Williams 等［2007］在少数不同的多核结构上，对其中的一些进行了探索。然而，调查这些即便是简化内核的可比较的性能仍具有指导意义。在一个 2 GHz Intel Core2 Xeon E5335 处理器上，对于一组从使用三角形绘制的 3D 表网格获得的拉普拉斯矩阵，csrmul_serial()核大约以每秒 2.02 亿非零处理的速度运行。使用 Intel 线程构建块提供的 parallel_for结构将该内核并行化，在 2、4 和 8 核的机器上可分别获得 2.0、2.1 和 2.3 倍的加速。在一个 GeForce 8800 Ultra 上，csrmul_kernel()和 csrmul_cached()内核获得大约每秒 772 和 92 000 万非零的处理速率，对应的并行加速为单核 CPU 核上串行性能的 3.8 和 4.6 倍。

A.8.3　扫描和归约

并行扫描，也被称为并行前缀总和，是数据并行算法［Blelloch，1990］最重要的构建块之一。给定一个 n 元素的序列 a：

$$[a_0, a_1, \cdots, a_{n-1}]$$

和一个二进制联合操作⊕，扫描函数计算如下序列：

$$\text{scan}(a, \oplus) = [a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})]$$

作为一个例子，如果我们将⊕当成通常的加操作，这时对输入数组应用扫描

$$a = [3\ 1\ 7\ 0\ 4\ 1\ 6\ 3]$$

将产生部分和序列：

$$scan(a, +) = [3\ 4\ 11\ 11\ 15\ 16\ 22\ 25]$$

在输出序列元素 i 和输入元素 a_i 联合的场景中，这个 scan 操作是一个包含 scan。仅与之前的元素联合会产生一个排外的 scan 操作，也被称为前缀总和操作。

这个操作的串行实现是非常简单的。它仅是一个在整个序列上迭代一次的循环，如图 A-8-6 所示。

```
template<class T>
__host__ T plus_scan(T *x, unsigned int n)
{
    for(unsigned int i=1; i<n; ++i)
        x[i] = x[i-1] + x[i];
}
```

图 A-8-6　串行加扫描模板

乍一看，该操作看来可能是固有串行的。然而，实际上它可以用并行高效地实现。关键在于，由于加操作是联合的，我们可以自由地改变元素加到一起的次序。例如，我们可以想象并行地加上多对连续元素，然后，将这些部分和加在一起，等等。

实现的一个简单方案来自 Hillis 和 Steele［1989］。他们算法的一个 CUDA 实现在图 A-8-7 中

给出。假定输入数组 x[]恰好包含每个线程块中线程的一个元素。它完成一个循环的 $\log_2 n$ 次迭代以将部分和聚集到一起。

```
template<class T>
__device__ T plus_scan(T *x)
{
    unsigned int i = threadIdx.x;
    unsigned int n = blockDim.x;

    for(unsigned int offset=1; offset<n; offset *= 2)
    {
        T t;

        if(i>=offset)  t = x[i-offset];
        __syncthreads();

        if(i>=offset)  x[i] = t + x[i];
        __syncthreads();
    }
    return x[i];
}
```

图 A-8-7　CUDA 的并行加扫描模板

为了理解这个循环的行为，仔细考虑图 A-8-8，它阐明了 $n=8$ 线程和元素的一个实例。图表的每级表示循环中的一步。线条指示数据取出的位置。对于输出的每一个元素（例如，图中的最后一行），我们将在输入元素上构建一个总和树。边沿粗体线说明最后元素的这个总和树的形式。该树的叶子都是初始元素。从任何输出元素向上回溯说明它联合了之上所有的输入值并包括它自身。

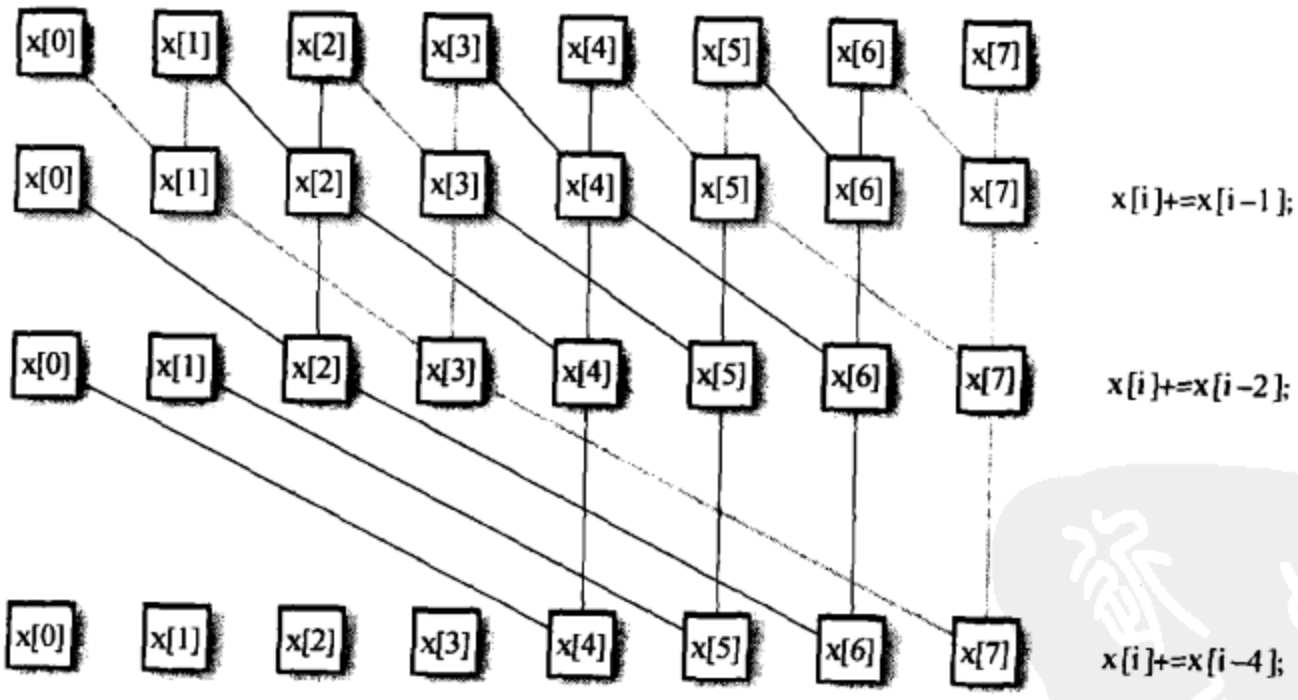


图 A-8-8　基于树的并行扫描数据参考

尽管简单，该算法没有我们所希望的那样有效。调查串行实现，我们看到，它完成 $O(n)$ 次加法。相反，并行实现完成 $O(n \log n)$ 次加法。由于这个原因，它不能高效地工作，因为计算同样的结果，它要做比串行实现更多的工作。幸运的是，还有其他高效的技术来实现 scan。更为高效的实现技术细节和这个多块数组块程序的扩展由 Sengupta 等［2007］提供。

在某些场合，我们可能仅对计算一个数组中所有元素的总和感兴趣，而不是由 scan 返回的

所有前缀总和序列。这就是*并行归约*问题。我们可以仅使用一个扫描算法来完成这个计算，但归约通常可以比扫描更为高效地实现。

图 A-8-9 展示了使用加法计算一个归约的代码。在该例中，每个线程仅装入输入序列的一个元素（例如，它最初对一个长度为 1 的序列求和）。在归约的最后，我们希望线程 0 保持最初被块中线程装入的所有元素的总和。这个内核中的循环隐含地在输入元素上构建了一个总和树，很像上面的扫描算法。

```
__global__
void plus_reduce(int *input, unsigned int N, int *total)
{
    unsigned int tid = threadIdx.x;
    unsigned int i   = blockIdx.x*blockDim.x + threadIdx.x;

    // Each block loads its elements into shared memory, padding
    // with 0 if N is not a multiple of blocksize
    __shared__ int x[blocksize];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    // Every thread now holds 1 input value in x[]
    //
    // Build summation tree over elements.
    for(int s=blockDim.x/2; s>0; s=s/2)
     {
        if(tid < s)  x[tid] += x[tid + s];
        __syncthreads();
     }

    // Thread 0 now holds the sum of all input values
    // to this block. Have it add that sum to the running total
    if( tid == 0 )  atomicAdd(total, x[tid]);
}
```

图 A-8-9 加归约的 CUDA 实现

在这个循环的最后，线程 0 保持着被该块装入的所有值的总和。如果我们希望由 total 指向的位置的值包含数组中所有元素的总和，我们必须结合网格中所有块的部分和。实现的一种策略是使每块将其部分和结果写入一个第二数组，然后再次运行归约内核，重复该过程直到我们将序列归约成了一个单个值。Tesla GPU 架构支持的，一个更有吸引力的替代办法是，使用 atomicAdd()原语，一个由存储器子系统支持的高效读—修改—写原语。这消除了额外临时数组和反复运行内核的需要。

对于并行编程、突出每块共享存储器的重要性和在线程间进行高效协作的低成本栅障来说，并行归约都是一个必需的原语。如果在片外的全局存储器中做的话，线程间这种程度的数据混合将具有高得惊人的代价。

A.8.4 基数排序

扫描原语的一个重要应用是排序例程的实现。图 A-8-10 中的代码在一个单独的线程块上实现了整数的基数排序。它接收一组为块中的每个线程包含一个 32 位整数的 values 作为输入。出

于对效率的考虑，这个数组应该被存储在每个线程块的共享存储器中，但对于排序的正确运转来说这是不需要的。

```
__device__ void radix_sort(unsigned int *values)
{
    for(int bit=0; bit<32; ++bit)
    {
        partition_by_bit(values, bit);
        __syncthreads();
    }
}
```

图 A-8-10　基数排序的 CUDA 代码

这是基数排序的一个相当简单的实现。它假定过程 partition_by_bit()可用，该过程对给定的数组进行划分，以使指定位为 0 的所有数出现在指定位为 1 的所有数之前。为了产生正确的输出，这个划分必须是稳定的。

实现划分程序是扫描的一个简单应用。线程 i 持有值 x_i，且必须计算正确的输出索引，指示向哪儿写入该值。这样，需要计算：①指派位是 1 的，线程 $j<i$ 的数量；②指派位是 0 的位的总数。partition_by_bit()的 CUDA 代码在图 A-8-11 中给出。

```
__device__ void partition_by_bit(unsigned int *values,
                                 unsigned int bit)
{
    unsigned int i    = threadIdx.x;
    unsigned int size = blockDim.x;
    unsigned int x_i  = values[i];
    unsigned int p_i  = (x_i >> bit) & 1;

    values[i] = p_i;
    __syncthreads();

    // Compute number of T bits up to and including p_i.
    // Record the total number of F bits as well.
    unsigned int T_before = plus_scan(values);
    unsigned int T_total  = values[size-1];
    unsigned int F_total  = size - T_total;
    __syncthreads();

    // Write every x_i to its proper place
    if( p_i    )
        values[T_before-1 + F_total] = x_i;
    else
        values[i - T_before] = x_i;
}
```

图 A-8-11　按位划分数据的 CUDA 代码，是基数排序的一部分

一个相似策略可以用来实现一个基数排序内核，该内核对一个大长度数组进行排序，而不仅是一块数组。基本步骤保持了扫描过程，尽管计算在多个内核间划分时，我们必须使用双缓冲存储数组的值而不是在原地进行划分。在大数组上高效执行基数排序的细节由 Satish、Harris 和 Garland［2008］提供。

A.8.5 GPU 上的 N-Body 应用[⊖]

Nyland、Harris 和 Prins [2007] 描述了一种简单但具有极高性能且很有用的计算内核——all-pairs N-body 算法。对很多科学应用来说，它是一个耗时的元素。N-body 仿真计算多体系统的演化，在该多体系统中，每个体持续地与每个天其他体相互作用。一个例子是天体物理仿真，其中每个体代表一个独立的天体，并且每个天体用引力相互吸引。其他例子是蛋白质折叠，其中 N-body 仿真用来计算静电和 van der Waals 力；液体湍流仿真；计算机图形中的全局照明。

all-pairs N-body 算法通过计算系统中的每个 pair-wise 力，并对每个体求和，计算系统中每个体上的力的总和。许多科学家认为这种方法是最精确的，仅存在来自浮点硬件操作的精度损失。缺点是它的 $O(n^2)$ 计算复杂度，对于具有多于 10^6 体的系统来说，太大了。为克服这个大的开销，提出了几个复杂度为 $O(n \log n)$ 和 $O(n)$ 的简化算法；例子是 Barnes-Hut 算法、快速多极方法和 Particle-Mesh-Ewald 求和。所有的快速方法仍旧依赖于 all-pairs 方法，作为短范围里精确计算的一个内核；因此，它依然很重要。

N-Body 数学

对于引力仿真，使用基本物理学计算 body-body 力。在两个用 i 和 j 索引的天体之间，3D 力向量是：

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{||r_{ij}||^2} \times \frac{r_{ij}}{||r_{ij}||}$$

力的大小以左边的形式计算，而方向以右边的形式计算（单位向量从一个天体指向另一个）。

给定一系列相互作用的天体（整个系统或一个子集），计算是简单的：对所有的相互作用对，为每个天体计算力和总和。一旦算出了力的总和，就用它来更新每个天体的位置和速度（基于之前的位置和速度）。力的计算复杂度是 $O(n^2)$，而更新的复杂度是 $O(n)$。

串行力-计算代码使用两个嵌套的 for 循环在多对体上迭代。外部循环对体进行选择，以其计算力的总和，内部循环在所有的体上迭代。内部循环调用一个计算 pair-wise 力的函数，然后将力加到一个连续的总和中。

为了并行地计算多个力，我们为每个体分配一个线程，因为在每个体上的力的计算与其他体上的计算是独立的。一旦所有的力都已算出，就可以对天体的位置和速度进行更新。

串行和并行版本的代码在图 A-8-12 和图 A-8-13 中给出。串行版本有两个嵌套的 for 循环。到 CUDA 的转化，像许多其他例子一样，将串行外部循环转化成每线程的内核，每个线程计算一个单独天体上力的总和。CUDA 内核为每个线程计算一个全局的线程 ID，替换串行外部循环的迭代变量。两个内核以将加速度的总和存入一个全局数组而结束，该全局数组用于以并行步的方式计算新的位置和速度。

```
void accel_on_all_bodies()
{
    int i, j;
    float3 acc(0.0f, 0.0f, 0.0f);
    for (i = 0; i < N; i++)   {
        for (j = 0; j < N; j++)  {
            acc = body_body_interaction(acc, body[i], body[j]);
        }
        accel[i] = acc;
    }
}
```

图 A-8-12 在 N 个体上计算所有 pair-wise 力的串行代码

⊖ 根据 Nyland、Harris 和 Prins [2007]《GPU Gems 3》第 31 章，“Fast N-Body Simulation with CUDA” 改编。

```
__global__ void accel_on_one_body()
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j;
    float3 acc(0.0f, 0.0f, 0.0f);

    for (j  = 0; j < N; j++) {
        acc = body_body_interaction(acc, body[i], body[j]);
    }
    accel[i] = acc;
}
```

图 A-8-13　在一个单个体上计算力的总和的 CUDA 线程代码

外部循环被运行 N 个线程，每个线程对应一个体的 CUDA 内核网格所代替。

GPU 执行优化

上述的 CUDA 代码功能上是正确的，但却不是高效的，因为它忽略了关键的架构特征。可以使用三个主要的优化获得更好的性能。首先，共享存储器可以用来避免线程间的同一存储器读；其次，在各个体上使用多线程以提高小值 N 的性能；再次，循环展开以减少循环开销。

使用共享存储器

共享存储器可以保存体位置的一个子集，很像一个 cache，消除了线程间的多余全局存储器请求。我们对上面展示的代码进行优化，使一个线程块中 p 个线程的每一个装到共享存储器的一个位置（共 p 个位置）。一旦所有的线程均已装入一个值到共享存储器中，由 __syncthreads() 保证，每个线程这时可以进行 p 次交互（使用共享存储器中的数据）。对于每一个体，这个过程重复 N/p 次以完成力的计算，这一因子 p（典型地在 32 ~ 128 的范围中）减少了请求存储器的数量。

称为 accel_on_one_body() 的函数需要少许改动以支持这个优化。修改后的代码在图 A-8-14 中给出。

```
__shared__ float4 shPosition[256];
...
__global__ void accel_on_one_body()
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j, k;
    int p = blockDim.x;
    float3 acc(0.0f, 0.0f, 0.0f);
    float4 myBody = body[i];

    for (j = 0; j < N; j += p) {  // Outer loops jumps by p each time
        shPosition[threadIdx.x] = body[j+threadIdx.x];
        __syncthreads();
        for (k = 0; k < p; k++) { // Inner loop accesses p positions
            acc = body_body_interaction(acc, myBody, shPosition[k]);
        }
        __syncthreads();
    }
    accel[i] = acc;
}
```

图 A-8-14　在每个体上计算力的总量的 CUDA 代码，使用共享存储器以提高性能

之前在所有体上迭代的循环现在被块的维数 p 所跳过。外部循环的每次迭代装入 p 个连续位置到共享存储器中（每个线程一个位置）。多个线程同步进行，然后每个线程计算 p 个力的计算。需要进行第二次同步，以确保新值在所有线程使用当前值完成力的计算之前没有被装入共享存储器。

使用共享存储器将存储器带宽需求减少到小于 GPU 可以维持的总带宽的 10%（使用少于 5 GB/s）。这个优化使应用忙于执行计算而不是等待存储器访问，因为它或许无需使用共享存储器。改变 N 值的性能在图 A-8-15 中给出。

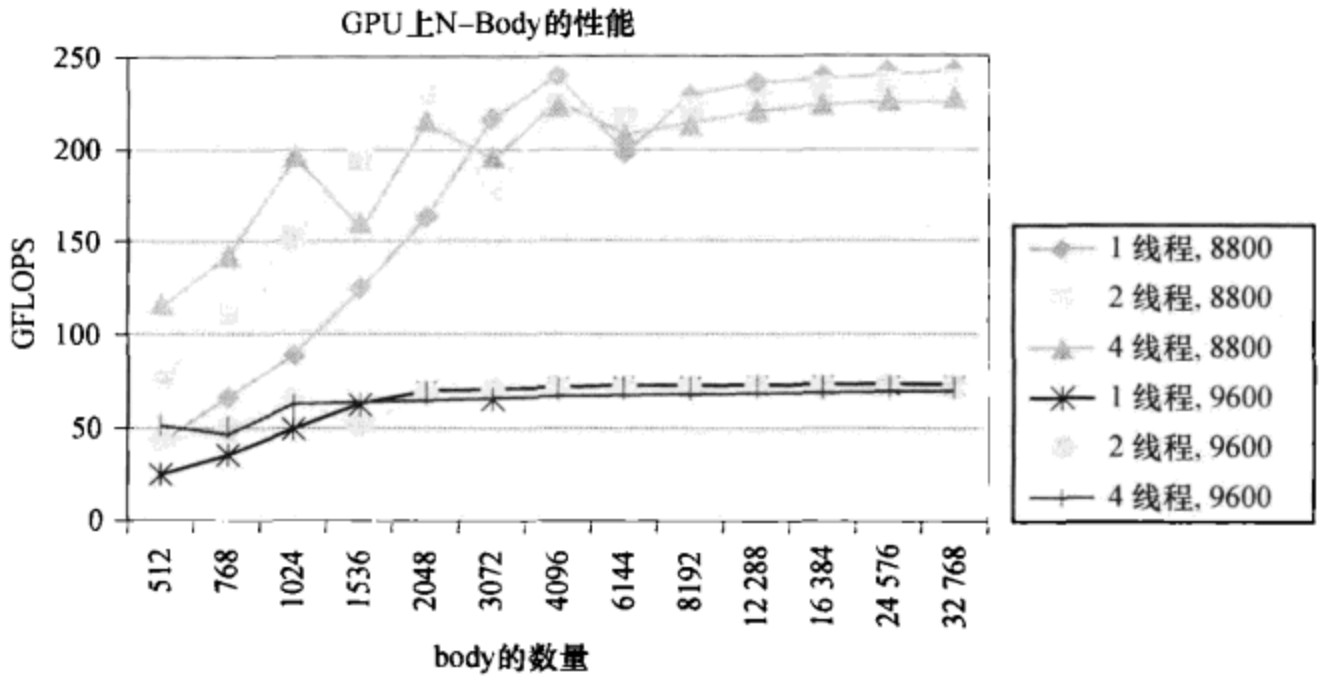


图 A-8-15　在 GeForce 8800 GTX 和 GeForce 9600 上 N-body 应用的性能测量
8800 具有 128 个运行在 1.35 GHz 的流处理器，而 9600 具有 64 个运行在 0.80 GHz（大约是 8800 的 30% 的流处理器）。峰值性能是 242 GFLOPS。对于一个具有更多处理器的 GPU 来说，为获得完全的性能，问题将会更大（9600 的峰值大约为 2048 体，而 8800 直到 16 384 体才达到它的峰值性能）。对于较小的 N，每体多于一个的线程显著地提高性能，但由于 N 的增长，最终将造成性能损失。

在每个体上使用多线程

图 A-8-15 展示了在 GeForce 8800 GTX 上的小 N 值（$N < 4096$）性能退化问题。许多研究成就依赖于集中于小 N（为进行长时间的仿真）的 N-body 计算，使得它成为我们优化努力的目标。我们用来解释较低性能的假定是，在 N 较小时，没有足够的工作保持 GPU 忙碌。解决办法是为每个体分配更多的线程。我们将线程块的维数从（p，1，1）改变到（p，q，1），其中 q 个线程将一个单一体中的工作划分成相等的部分。通过分配相同线程块中的额外线程，部分结果可以被存储到共享存储器中。当所有力的计算都完成时，q 个部分结果可以被收集到一起并相加来计算最终的结果。每个体使用两个或四个线程导致了小 N 的很大改进。

作为一个例子，当 $N = 1024$ 时，8800 GTX 的性能跳跃到 110% 附近（一个线程达到 90 GFLOPS，四线程达到 190 GFLOPS）。在大 N 上，性能仅有轻微退化，因此我们仅对小于 4096 的 N 使用该优化。一个具有 128 个处理器和一个具有 64 个处理器以三分之二时钟速度运行的更小 GPU 的性能提升在图 A-8-15 中给出。

性能比较

N-body 代码的性能在图 A-8-15 和图 A-8-16 中给出。在图 A-8-15 中，展示了高性能和中性能 GPU，连同在每体上使用多线程获得的性能提升。最快 GPU 的性能在从 90 到仅低于 250 GFLOPS 的范围内。

图 A-8-16 展示了在 Intel Core2 CPU 运行的几乎同样的代码（C ++ 与 CUDA）。CPU 的性能大约是 GPU 的 1%，在 0.2 ~2 GFLOPS 的范围内，在很宽的问题大小范围内几乎保持不变。

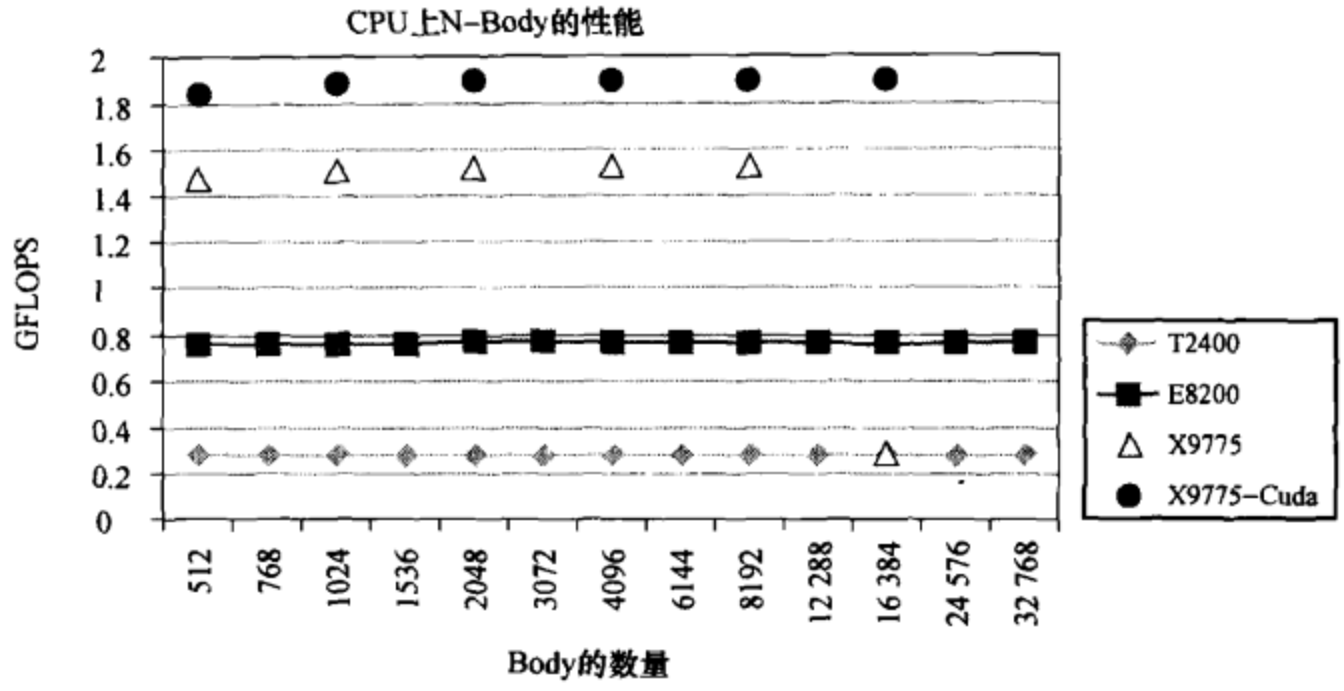


图 A-8-16　在一个 CPU 上的 N-body 代码的性能测量

图中说明了使用 Intel Core2 CPU 的单精度 N-body 性能，由它们的 CPU 模型号码指示。注意 GFLOPS 性能的显著下降（在 y 轴上以 GFLOPS 显示），证明了与 CPU 相比，GPU 有多快。CPU 的性能通常独立于问题的规模，除了当 $N \approx 16\,384$ 在 X9775 CPU 上的反常低性能之外。图形也说明了在一个单核 CPU 上运行 CUDA 版代码（使用面向 CPU 的 CUDA 编译器）的结果，它胜过 C++代码 24%。作为一种编程语言，CUDA 暴露了编译器可以开发的并行性和局部性。Intel CPU 是 3.2 GHz Extreme X9775（代号为“penryn”），2.66 GHz E8200（代号为“Wolfdale”），前 Penryn 桌面 CPU 和 1.83 GHz T2400（代号为“Yonah”），一个 2007 笔记本电脑 CPU。Core2 架构的 Penryn 版本以其 4 位除法器对 N-body 计算相当感兴趣，允许除法和平方根操作执行比先前的 Intel CPU 快 4 倍。

图 A-8-16 也说明了编译用于 CPU CUDA 版代码的结果，其中性能提升了 24%。CUDA 作为一种编程语言，提供了并行性，允许编译器在单核上更好地使用 SSE 向量单元。N-body 代码的 CUDA 版本也自然地映射到多核 CPU（使用块的网格），它因而在一个 8 核系统上，$N = 4096$ 时获得了近乎完美的扩展（在 2、4 和 8 核上，对应的比率为 2.0、3.97 和 7.94）。

结果

以适度的努力，我们开发了一种计算内核，与多核 CPU 相比，可以通过乘上一个因子来提高 GPU 的性能，该因子最高可达 157。与以 44 Hz 帧率在 GeForce8800 GPU 运行的同样代码相比，N-body 代码在近来 Intel CPU（3.2 GHz 的单核 Penryn X9775）上的执行时间，每帧占用时间要长 3 秒。在前 Penryn CPU 上，该代码需要 6～16 秒，在稍老 Core2 处理器和奔腾 IV 处理器上，时间大约为 25 秒。我们必须将明显的性能增加划分成一半，因为 CPU 仅需要半数计算量来计算相同的结果（使用在一对体上的力与强度和对应的方向等价地优化）。

GPU 是如何以这么大的数量加速代码的？答案需要检查体系结构细节。pair-wise 力的计算需要 20 个浮点操作，主要由加和乘指令组成（一些可以使用乘加指令组合），但也有用于向量标准化的除和平方根指令[⊖]。Intel CPU 使用很多周期完成单精度除和平方根指令，尽管在最近的 Penryn 系列 CPU 中，以更快的 4 位除法器[⊜]对其进行了改进。另外，寄存器容量的限制导致了 x86 代码中众多的 MOV 指令（大概是 to/from 一级 cache）。相反，GeForce 8800 使用 4 个周期执行一个倒数平方根线程指令；专用功能的精确性可参考 A.6 节。它有一个更大的寄存器文件（每线程）和共享存储器，可以作为指令操作数访问。最后，与来自多种 x86 CPU 编译器的多于

⊖ 没有考虑 x86 SSE 指令倒数平方根（RSQRT*）和倒数（RCP*），因为它们的精确度太低以致不具可比性。

⊜ 《Intel Corporation, Intel 64 and IA-32 Architectures Optimization Reference Manual》，2007 年 11 月。序列号：248966-016。登录 www3.intel.com/design/processor/manuals/248966.pdf 也可以得到。

40 条的指令比较，CUDA 编译器为循环的一个迭代发射 15 条指令。更大的并行、复杂指令的更快执行、更多的寄存器空间和高效的编译器，所有这些组合，解释了 CPU 和 GPU 之间，N-body 代码性能显著提升的原因。

在 GeForce 8800 上，all-pairs N-body 算法产生高于 240 GFLOPS 的性能，与之相比，在近来的串行处理器上仅能获得不到 2 GFLOPS 的性能。在 CPU 上编译和执行 CUDA 版的代码证明了问题可以很好地扩展到多核 CPU 上，但它仍然要比单个 GPU 要慢得多。

我们将 GPU N-body 仿真与运动图形显示结合，可以以 44 帧每秒的速率交互地显示 16K 的体。这允许天体物理和生物物理以交互的速率显示和操作。另外，我们可以用参数表示很多设置，例如噪声衰减、阻尼和融合技术，立即动态地显示它们对系统的效果。这为科学家提供了极好的可视影像，推进了他们在其他不可视系统（太大或太小，太快或太慢）上的洞察力，允许他们为物理现象创造更好的模型。

图 A-8-17 展示了 16 K 体的天体物理仿真的一个时间序列显示，每体充当一个星系。初始配置是一个围绕 z 轴旋转的球体外壳。天体物理学家感兴趣的一个现象是聚集，它发生、并同星系随时间推移的合并一起存在。对于感兴趣的读者，这个应用的 CUDA 代码可从 www. nvidia. com/CUDA 中的 SDK 获得。

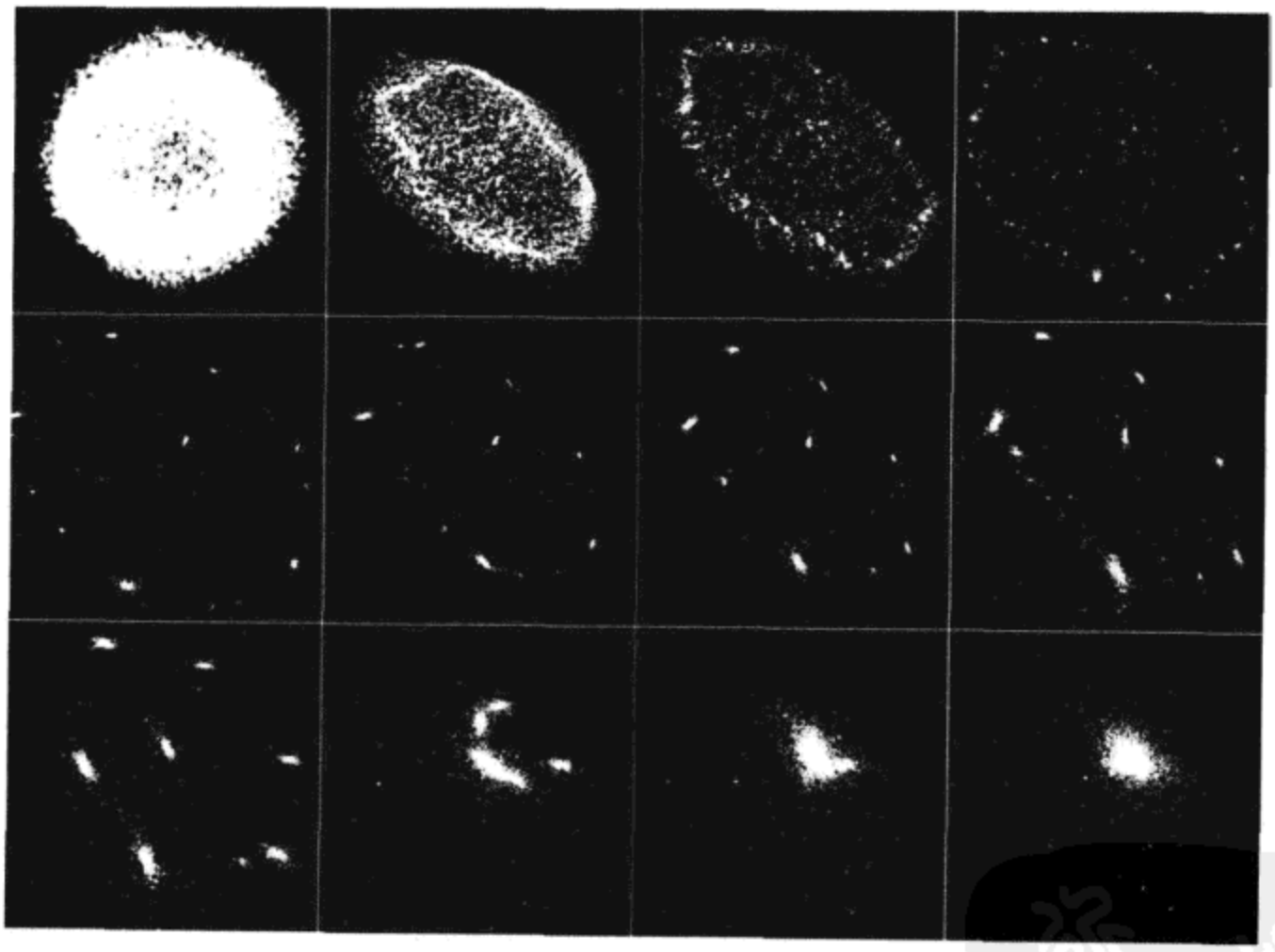


图 A-8-17　在一个具有 16 384 体的 N-body 系统演变过程中捕捉到的 12 幅图像

A. 9　谬误与陷阱

GPU 的发展和变化如此之快，以至于出现了许多谬误和易犯的错误。我们在此讨论典型的几个。

谬误：GPU 仅仅是 SIMD 向量多处理器。

很容易得出，GPU 仅是 SIMD 向量多处理器的错误结论。GPU 确实有一个 SPMD 风格的编程模型，在该模型中，程序员可以编写一个可以在多个线程实例中以多数据执行的单个程序。尽管

如此，这些线程的执行不是纯粹的 SIMD 或向量；它是 A.4 节中描述的单指令多线程（SIMT）。每个 GPU 线程有它自己的标量寄存器、线程私有存储器、线程执行状态、线程 ID、独立的执行和分支路径和高效的程序计数器，并且可以独立地对存储器寻址。尽管执行线程的 PC 相同时，一组线程（例如，一个 32 线程的 warp）执行得更为高效，但这不是必需的。因此，多处理器不是纯粹的 SIMD。线程执行模型是具有栅障同步和 SIMT 优化的 MIMD。如果单个线程 load/store 存储器访问能聚合成块访问，执行也会变得更为高效。然而，严格来说这个不是必须的。在一个纯粹的 SIMD 向量架构中，不同线程的存储器/寄存器访问必须以一个规整的向量模式对齐。GPU，对于寄存器或存储器访问，没有这样的限制；然而，如果线程 warp 访问局部数据块的话，执行将更为高效。

SIMT GPU 可以同时执行多于一个的线程 warp，进一步违反了纯粹的 SIMD 模型。在图形应用中，或许有多组顶点程序、像素程序和几何程序同时运行在多处理阵列中。计算程序也可以同时执行不同 warp 中的不同程序。

谬误：GPU 的性能不能比摩尔定律增长得更快。

摩尔定律仅是一个速率。对于任何其他速率来说，它不是“光速”极限。摩尔定律描述了一个随时间推移的展望，由于半导体技术的进步和晶体管变得更小，每晶体管的制造成本将以指数下降。以另外一种方式，给定一个不变的制造成本，晶体管的数量将指数增加。Gordon Moore［1965］预测，这种进步每年将为同样的成本提供大约两倍数量的晶体管，后来修正为每两年翻倍。尽管摩尔在每个集成电路仅有 50 个元件的 1965 年，做出了最初的预测，它仍被证明极为一致。从历史上说，晶体管尺寸的减小还有其他的好处，例如每个晶体管更低的功耗和不变功耗下更快的时钟速度。

这个晶体管增加的馈赠被系统架构师用来构建处理器、存储器和其他组件。曾经一段时间，CPU 设计者使用额外的晶体管以近似于摩尔定律的速度增加处理器的性能，性能提高了那么多以至于许多人认为处理器速度以摩尔定律的速度每 18 ~ 24 个月翻一番。实际上，不是这样的。

微处理器设计者花费了一些新的晶体管到处理器核上，改进架构和设计，并采用流水以获得更快的时钟速率。剩下的新晶体管用来提供更多的 cache，以使存储器访问更快。相反，GPU 设计者几乎不使用新的晶体管提供更多的 cache；大多数晶体管都被用在改进处理器核和增加更多的处理器核上。

通过四种机制 GPU 变得更快。第一，GPU 设计者收获了摩尔定律的馈赠，直接通过应用以指数增多的晶体管构建更多的并行、更快的处理器。第二，GPU 设计者可以随时间推移改进架构，增加处理的效率。第三，摩尔定律假定成本不变，因此摩尔定律速度可以无疑地被超过，通过为具有更多晶体管的更大的芯片花费更多。第四，通过使用更快的存储器、更宽的存储器、数据压缩和更好的 cache，GPU 存储系统，以几乎可与处理速度相比的速度，增加了它们的有效带宽。这四种方法的组合历史性地允许 GPU 性能有规律地翻倍，大约 12 ~ 18 个月翻倍。这种速度超过了摩尔定律的速度，已经在图形应用方面被证明了将近十年，且没有显示有明显下降的信号。最具挑战的速率限制因素看来是存储系统，但具有竞争性的革新也同样快速地提升着它。

谬误：GPU 仅能渲染 3D 图形；它们不能进行通用计算。

GPU 被创建来渲染 3D 图形、2D 图形和视频。为了满足图形软件开发者在图形 API 中表达的接口、性能/特征需求的要求，GPU 变成了大规模并行可编程浮点处理器。在图形领域，这些处理器通过图形 API 和鲜为人知的编程语言（OpenGL 和 Direct3D 中的 GLSL、Cg 和 HLSL）编程。尽管如此，没什么能够阻挡 GPU 架构师向程序员暴露不使用图形 API 或鲜为人知的图形语言的并行处理器核。

实际上，Tesla 架构系列的 GPU 通过一个被称为 CUDA 的软件环境暴露处理器，CUDA 允许

程序员使用 C 语言和后来的 C ++ 开发通用应用程序。GPU 是完全的图灵处理器，因此它们可以运行 CPU 可以运行的任何程序，尽管可能不太好。也可能更快。

谬误：GPU 不能快速地运行双精度浮点程序。

在过去，GPU 完全不能运行双精度浮点程序，除非通过软件模拟。那一点也不快。GPU 已经取得了进步，从索引算术表示法（颜色查找表）到每颜色元素 8 位整数，到定点算术，到单精度浮点，到近来新增的双精度浮点。现代 GPU 事实上以单精度 IEEE 浮点算术执行所有计算，另外，也开始使用双精度。

付出较小的额外开销，GPU 即可支持双精度浮点，如同支持单精度浮点一样。今天，双精度要比单精度运行慢很多，大约慢 5 ~ 10 倍。增加额外的开销，双精度性能可以相对于单精度分阶段的增加，如同很多应用要求的那样。

谬误：GPU 不能正确地进行浮点计算。

GPU，至少在 Tesla 构架系列的处理器中，以 IEEE 754 浮点标准指定的级别，执行单精度浮点处理。因此，在精度方面，GPU 与任何其他 IEEE 754 兼容的处理器是相当的。

今天，GPU 没有实现标准中描述的一些特殊特征，像处理非规格化数和提供精确浮点异常。尽管如此，近来面市的 Tesla T10P GPU 提供了完整的 IEEE 舍入、合并的乘加和双精度的非规格化数支持。

陷阱：仅仅使用更多的线程来掩盖更长的存储器延迟。

CPU 核典型的设计来全速运行一个单线程。为全速运行，每条指令和它的数据，在那条指令运行的时候，需要可用。如果下一条指令没有准备好或者它所需要的数据不可用，指令不能运行，且处理器停顿。外部存储器距处理器较远，因此从存储器中取数据浪费了很多周期。因此，CPU 需要很大的本地 cache，保持运行避免停顿。存储器延迟很长，因此努力在 cache 中运行以避免存储器访问。在某个位置，程序的工作集需求可能比任何 cache 都大。一些 CPU 使用多线程来容忍延迟，但每核的线程数通常限制在一个较小的数量上。

GPU 的策略是不同的。GPU 核设计用来同时运行众多线程，但每次仅执行任意线程的一条指令。表述这个的另一种方法是，GPU 缓慢地运行每个线程，但总体上高效地运行这些线程。每个线程可以容忍一些数量的存储器延迟，因为其他线程可以运行。

这样的不利之处是需要多路——众多多路线程——来掩盖存储器延迟。另外，如果存储器访问是分散的或在线程间不相关，存储系统为适应单独的请求将逐渐变慢，甚至多线程将不能掩盖延迟。因此，易犯的错误是“仅使用更多的线程”策略来掩盖延迟，你必须有足够的线程，且线程在存储器访问的局部性方面必须有较好的行为。

谬误：$O(n)$ 算法是难以加速的。

无论 GPU 在处理数据方面有多快，传输数据到设备和从设备传出数据的步骤可能会限制复杂度为 $O(n)$（每个数据具有较小的工作量）的算法的性能。使用 DMA 传输且仅有较少的非 DMA 传输时，PCIe 总线的最高传输率接近 48 GB/s。相反，CPU 典型的系统存储器访问速度为 8 ~ 12 GB/s。举例说明问题，如向量加，将被到 GPU 的输入和计算的返回输出传输所限制。

有三种方法克服传输数据的开销。首先，尽力将数据保留在 GPU 中，能有多久就多久，而不是为一个复杂算法的不同步骤向后或向前移动数据。CUDA 有意地将数据在运行期间单独地留在 GPU 中，以支持该方法。

其次，GPU 支持 copy-in、copy-out 和计算的同时操作，因此在它进行计算时，数据可以流入和流出设备。该模型对任何到达即可处理的数据流来说是有用的。例子是，视频处理、网络路由、数据压缩/解压，甚至简单的计算如大向量数学。

第三种提议是将 CPU 和 GPU 放在一起使用，通过为各自分配工作的一个子集来提升性能，

将系统作为一个异构计算平台看待。CUDA 编程模型支持在不使用线程（通过异步 GPU 函数）的情况下，将工作分配到一个或多个 GPU，并持续使用 CPU，因此，保持所有 GPU 和一个 CPU 同时工作来解决这个问题，甚至会更快，相对是比较简单的。

A.10　小结

GPU 是大规模并行处理器，且被广泛应用，不仅在 3D 图形方面，而且在众多其他应用方面。图形设备到可编程处理器的演变使得这个广泛的应用成为可能。GPU 的图形应用编程模型通常是一个 API，像 DirectX™ 或者 OpenGL™。为了更多通用目的计算，CUDA 编程模型使用 SPMD（单程序多数据）风格，执行一个具有众多并行线程的程序。

GPU 的并行性将继续以摩尔定律扩展，主要通过增加处理器的数量。只有可以容易地扩展到数百个处理器核和数千个线程上的并行编程模型，在支持众核 GPU 和 CPU 方面，才会获得成功。同样，只有那些具有众多大量独立并行的任务才能被大规模并行众核架构所加速。

GPU 的并行编程模型正变得更为灵活，可用于图形和并行计算。例如，CUDA 正朝着完全 C/C ++ 功能方向快速发展。图形 API 和编程模型将可能根据 CUDA 的并行计算能力和模型改编。它的 SPMD 风格的线程模型对于表达大量并行来说，是可扩展的，并且方便、简洁、易学。

被编程模型中的这些改变所驱动，GPU 架构变得更为灵活和更具编程性。GPU 固定功能单元正变得可被通用程序访问，沿着 CUDA 程序，使用 GPU 纹理指令和纹理单元，已经使用纹理内在函数执行纹理查找的路线。

GPU 架构将不断地适应图形和其他应用程序员的使用模型。GPU 将继续扩展以包含更多的处理能力，通过增加处理器核、增加线程和程序可用的存储器带宽。另外，编程模型必须改进以包含可编程异构众核系统（同时包含 GPU 和 CPU）。

致谢

本附录是 NVIDIA 几个作者的工作成果。我们非常感谢 Michael Garland、John Montrym、Doug Voorhies、Lars Nyland、Erik Lindholm、Paulius Micikevicius、Massimiliano Fatica、Stuart Oberman 和 Vasily Volkov 的突出贡献。

A.11　拓展阅读

本节在 CD 中给出，综述了可编程的实时图形处理单元（GPU）的历史，从 20 世纪 80 年代到今天，价格降低了两个数量级，性能提高了两个数量级。追溯了 GPU 的发展，从固定功能流水线到具有 GPU 计算前景的可编程图形处理器、到统一的图形和计算处理器、到可视计算和可扩展 GPU。

附录 B

Computer Organization and Design: The Hardware/Software Interface, 4E

汇编器、链接器和 SPIM 仿真器

James R. Larus
微软研究院

对恶意中伤的恐惧，不能成为阻止言论和集会自由的借口。

Louis Brandeis、Whitney v. 加利福尼亚，1927

B.1 引言

编码指令作为一种二进制数字对计算机来说是自然而且有效的。然而人类理解和处理这些数字有很大的困难。人们读和写这些符号（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**㊀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和数据的内存字。

一个被称为**汇编器**㊁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新度。例如，B.2 节讨论的**宏**㊂允许程序员通过定义新操作来扩展汇编语言。

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 B-1-1 说明了如何构建一个程序。很多程序由多个文件组成——也被称为调用模块——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以及库中定义的数据。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**㊃全部解决时才能执行。另一个工具被称为**链接器**㊄，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。

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，这个程序计算而且打

㊀ 二进制表示，用来和计算机系统通信。
㊁ 将符号版本的指令翻译二进制版本的一段程序。
㊂ 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。
㊃ 一个需要从外部源代码获取更多信息才能完成的引用。
㊄ 也被称为链接编辑器。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。

印出 0 ~ 100 的整数的平方和。图 B-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 B-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。

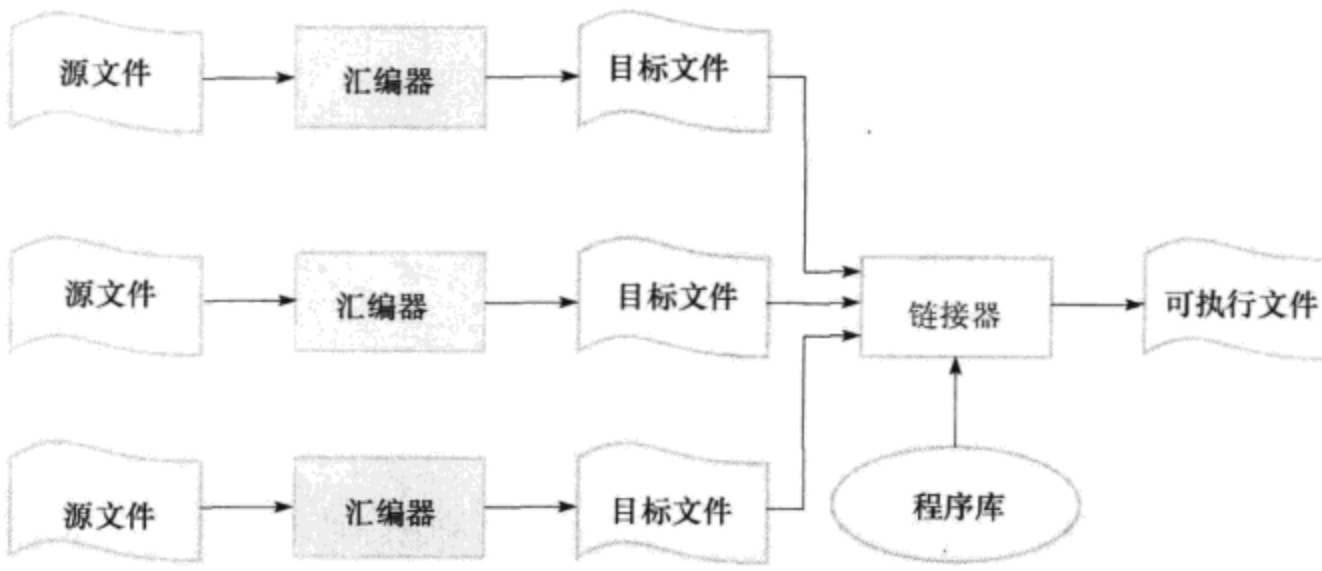


图 B-1-1　产生可执行文件的过程

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链接组成可执行文件。

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

图 B-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

```
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 B-1-3　同一个程序的汇编语言版

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 B-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**[⊖]，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一样具有可读性（除了没有耀眼的注释），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。

⊖　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，通常它以空白开始。

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($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      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra

        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

图 B-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

以空点开始的指令是汇编指令（见 B.10）。.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。

对比之下，图 B-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。其他形式的程序都是 C 编译器和汇编器产生的。

```
   #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 B-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 B-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。高级语言被称为**源语言**[⊖]，而编译器的输出是目标语言。

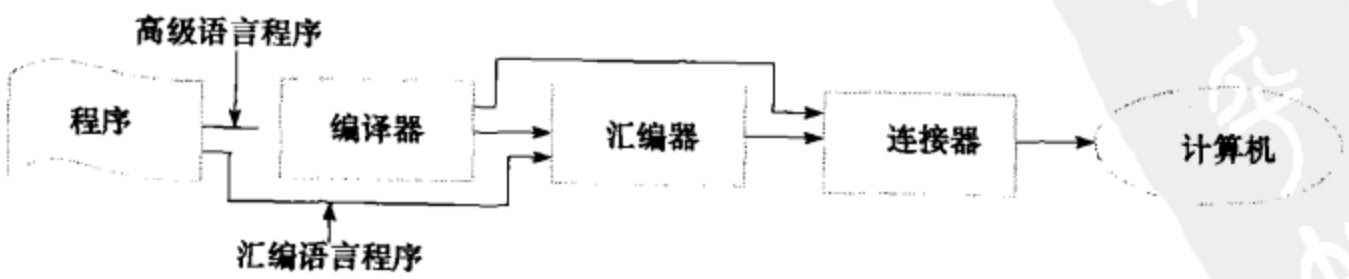


图 B-1-6　汇编语言由程序员编写或由编译器输出

⊖ 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，汇编语言仍然是很重要，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。

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。

B. 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，它在速度和代码体积方面具有优势，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。一个汇编语言程序员，在另一方面，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）。

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，甚至跨域程序边界，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。

B. 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要更长。例如，图 B-1-5 的 C 程

序仅有 11 行，但是图 B-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。

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 B-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？

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的更快。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。

精解：尽管有这些考虑，一些嵌入式应用使用高级语言编写。很多这些应用的程序很大，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埋单的国防部门开发 Ada——一种编写嵌入式系统的新高级语言。

B.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（labels）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。就如图 B-1-1 所示，汇编器产生一个输出文件，叫做目标文件，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，这个**标签是外部的**[⊖]（被称为全局的）。标签是局部的，如果仅仅能在定义它的文件内部被引用。很多的汇编器，默认标签是局部的，而且必须是显示声明为全局的。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**[⊜]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。

举例局部和全局标签

考虑图 B-1-4 中的程序。子程序具有一个外部标签（全局）`main`，它包含了两个局部标签——`loopt` 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 B-1-4 中的标签能否从另一个文件引用？

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。

⊖ 也被称为全局标签（external label，global label），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（local label）：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。

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 B-1-4 中，标签 str 在定义之前就被 la 指令使用。一个**前向引用**[⊖]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。

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。这些部分叫做*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**[⊜]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个指令在哪里。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。

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被定义。

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。

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[⊝]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，反向修补技术能加速汇编的原因在于：汇编器对输入只读一次。然而，它需要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。

B.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 B-2-1）：

⊖ 前向引用（forward reference）：一个标签在被定义前之就被使用。

⊜ 符号表（symbol table）：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。

⊝ 反向修补（backpatching）：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。



图 B-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。

- 目标文件头描述了文件中其他段的大小和位置。
- **代码段**[㊀]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**[㊁]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**[㊂]指明指令和数据字依赖于**绝对地址**[㊃]。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（symbol table）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（debugging information）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。

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以相同的地址开始（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。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。

B. 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 B-1-4 中，指令

```
.asciiz "The sum 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。汇编器能够比人更快而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 B. 10 节描述。

㊀ 代码段（text segment）：UNIX 目标文件的一个段，源文件中程序的机器语言代码。
㊁ 数据段（data segment）：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。
㊂ 重定位信息（relocation information）：UNIX 目标文件的一个段，根据绝对地址来区别数据字和指令。
㊃ 绝对地址（absolute address）：内存中标量或者程序的实际地址。

举例　**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。宏，如同子程序，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。

举例　**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
lnt_str:    .asciiz“% d”
            .text
            la $a0,int_str      #加载字符串的地址到第一个参数
            mov $a1, $7         #将值加载到第二个参数
            jal printf          #调用 printf 例程
```

.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
lnt_str:    .asciiz“% d”
            .text
            .macro print_int( $arg)
            la $a0,int_str      #将字符串的地址加载到第一个参数
            mov $a1, $arg       #加载宏的参数( $arg)到第二个参数
            jal printf          #调用 printf 例程
            .end_macro
print_int( $7)
```

宏有一个**形式参数**[⊖] $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
```

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？

⊖ 形式参数（formal parameter）：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。

```
la   $a0,int_str
mov $a1,$a0
jal printf
```

答案

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。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融合成一个文件。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，这部分代码可排除在外。

如果宏和条件汇编有用的话，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。

B.3　链接器

单独编译[⊖]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。

将多个文件融合在一起的工具叫做链接器（linker）（见图 B-3-1）。它执行三个任务：

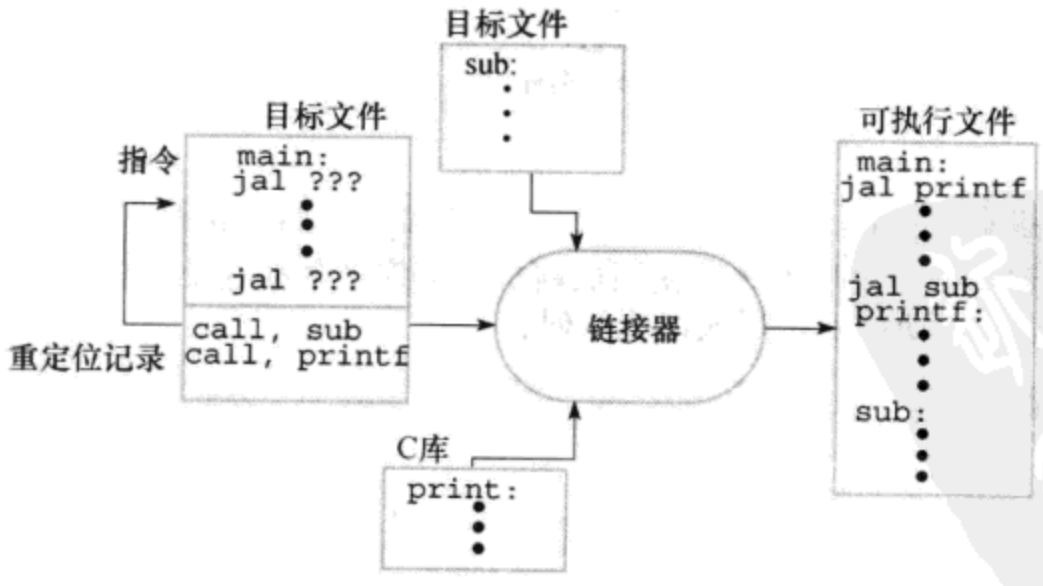


图 B-3-1　链接器搜查一组目标文件和程序库，寻找在程序中使用的非局部函数，将其合并成一个可执行文件，而且解决不同文件间的程序的引用

⊖ 单独编译（separate compilation）：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。

- 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连接器的目标代码中。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。

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。因为文件在汇编上独立的，汇编器不知道一个模块的指令或者数据相对别的文件放在哪里。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。典型地，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。

B.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 B.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 B.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。

B.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 系统上的一组规程。很多情况下，硬件不会影响这些规程。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，这些规程代表了编程人员为了遵循一些相同地规定的一种约定。

基于 MIPS 的系统通常将内存分割成三个部分（见图 B-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[⊖]（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。

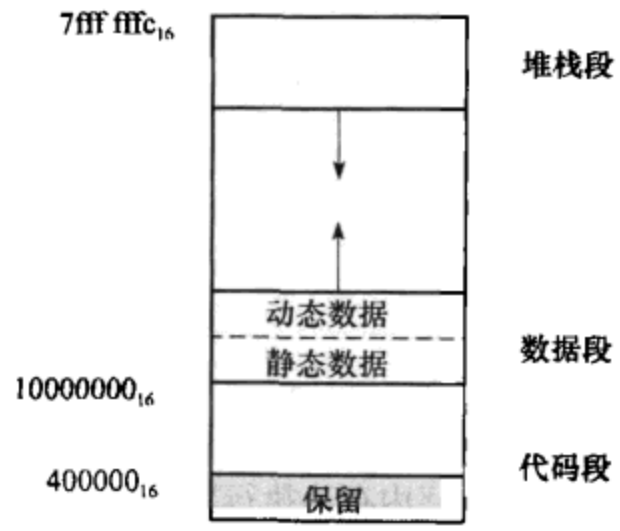


图 B-5-1　内存布局

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的范围来满足这个需求。如图 B-5-1 中向上的箭头所指示的，malloc 通过使用系统调用 sbrk 扩展了动态内存，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 5.4 节）。

第三部分，程序**堆栈段**[⊜]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。

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} 的字到寄存器 $v0 需要两个指令：

```
lui $s0,0x1001       #0x1001 是十六进制数
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 $32\,768_{10}$。）

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 64 KB。拥有这个全局指针，我们可以将以上例子改写为一个指令：

```
lw $v0,0x8020($gp)
```

当然，一个全局指针寄存器使得寻址 $10000000_{16} \sim 10010000_{16}$ 比别的堆地址定位要快。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。

B.6　过程调用规范

当程序中的过程（procedure）是被分别编译时，管理寄存器使用的规范是必要的。为了编译一个某个过程，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

⊖ 静态数据（static data）：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。
⊜ 堆栈段（stack segment）：程序用来保存过程调用帧的那段内存。

留。寄存器使用的规则被称作**寄存器使用**[⊖]或者过程调用规范。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。

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` ~ `$t9`（8 ~ 15，24，25）是**调用者保存的寄存器**[⊜]，被用来保存临时变量，这些值在调用的时候不需要保存（见第 2 章 2.8 节）。
- 寄存器 `$s0` - `$s7`（16 ~ 23）被称为**被调用者保存的寄存器**[⊝]，保存长期存活值，这些值应当在程序调用时保存。
- 寄存器 `$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。指令 `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 B-6-1 罗列了这些寄存器及其用途。

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	保存临时值（过程调用预留）
$at	1	为汇编器保留	$s1	17	保存临时值（过程调用预留）
$v0	2	表达式求值以及函数的结果	$s2	18	保存临时值（过程调用预留）
$v1	3	表达式求值以及函数的结果	$s3	19	保存临时值（过程调用预留）
$a0	4	参数 1	$s4	20	保存临时值（过程调用预留）
$a1	5	参数 2	$s5	21	保存临时值（过程调用预留）
$a2	6	参数 3	$s6	22	保存临时值（过程调用预留）
$a3	7	参数 4	$s7	23	保存临时值（过程调用预留）
$t0	8	临时（不为过程调用预留）	$t8	24	临时（不为过程调用预留）
$t1	9	临时（不为过程调用预留）	$t9	25	临时（不为过程调用预留）
$t2	10	临时（不为过程调用预留）	$k0	26	为 OS 内核保留
$t3	11	临时（不为过程调用预留）	$k1	27	为 OS 内核保留
$t4	12	临时（不为过程调用预留）	$gp	28	全局区域的指针
$t5	13	临时（不为过程调用预留）	$sp	29	堆栈指针
$t6	14	临时（不为过程调用预留）	$fp	30	帧指针
$t7	15	临时（不为过程调用预留）	$ra	31	返回地址（函数调用使用）

图 B-6-1　MIPS 寄存器和使用规则

⊖ 寄存器使用（register use convention）：或者称为 procedure call convention：管理过程（调用）使用寄存器的软件协议。
⊜ 调用者保存的寄存器（caller-saved register）：调用程序保存的寄存器。
⊝ 被调用者保存的寄存器（callee-saved register）：被调用者程序保存的寄存器。

B.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**[⊖]。这段内存被用作以下目的：

- 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。
- 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被称作堆栈帧。

图 B-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，所以帧的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`：

```
lw $v0,0($fp)
```

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：

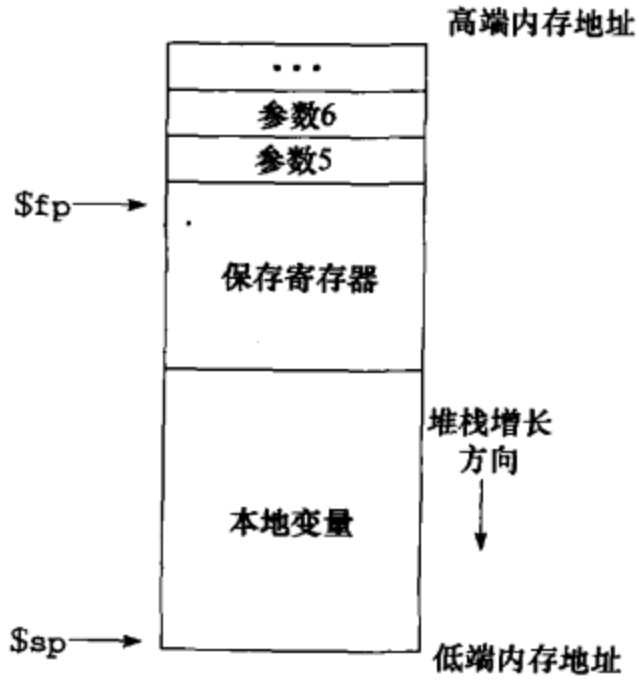


图 B-6-2　堆栈帧的示意图

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 个参数被传递到寄存器中，所以第五个参数成为栈中第一个被保存的参数

1）传递参数。根据规范，第一批的 4 个参数被传递到寄存器 `$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。

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。

⊖ 过程调用帧（procedure call frame）：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。

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**[⊖]——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。

B.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)
{
   lf (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，$10! = 10 \times 9 \times \cdots \times 1$）。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 $ra。一个帧的大小比两个寄存器大，因为调用程序所需要的一个堆栈帧的最小大小是 24 字节。最小的帧可以容纳 4 个寄存器参数（$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      #  栈的帧大小是 32 字节
        sw      $ra,20($sp)      #  保存返回地址
        sw      $fp,16($sp)      #  保存旧的栈帧指针
        addiu   $fp, $sp,28      #  创建栈指针
```

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，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。

⊖ 递归过程（recursive procedure）：就是指某个过程能通过调用链直接或间接地调用自己。

```
li      $a0,10      # 将参数(10)放到地址 $a0
jal     fact        # 调用阶乘函数
la      $a0, $LC    # 将格式化串保存到地址 $a0 处
move    $a1, $v0    # 将 fact 函数计算的阶乘结果保存到地址 $a1
jal     printf      # 调用打印函数
```

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)     # 保存返回地址
    lw      $fp,16($sp)     # 保存帧指针
    addiu   $sp,$sp,32      # 弹出堆栈帧
    jr      $ra             # 返回到调用者
    .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      # 栈的帧大小是 32 字节
    sw      $ra,20($sp)     # 保存返回地址
    sw      $fp,16($sp)     # 保存帧指针
    addiu   $fp,$sp,28      # 创建帧指针
    sw      $a0,0($fp)      # 保存参数(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)      #加载 n
    bgtz    $v0,$L2         # 分支,如果 n>0
    li      $v0,1           #返回 1
    jr      $L1             #跳转到返回的代码处
$L2:
    lw      $v1,0($fp)      #加载 n
    subu    $v0,$v1,1       #计算 n-1
    move    $a0,$v0         # 将值保存到 $a0(地址 a0)
    jal     fact            # 调用阶乘函数
    lw      $v1,0($fp)      #加载 n
    mul     $v0,$v0,$v1     #计算阶乘 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:                        # 结果保存到 $v0
    lw      $ra,20($sp)     # 恢复 $ra 中的值
    lw      $fp,16($sp)     # 保存 $fp 中的值
    addiu   $sp,$sp,32      #弹出栈
    jr      $ra             # 返回到调用者
```

举例 **递归过程中的栈**

图 B-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？

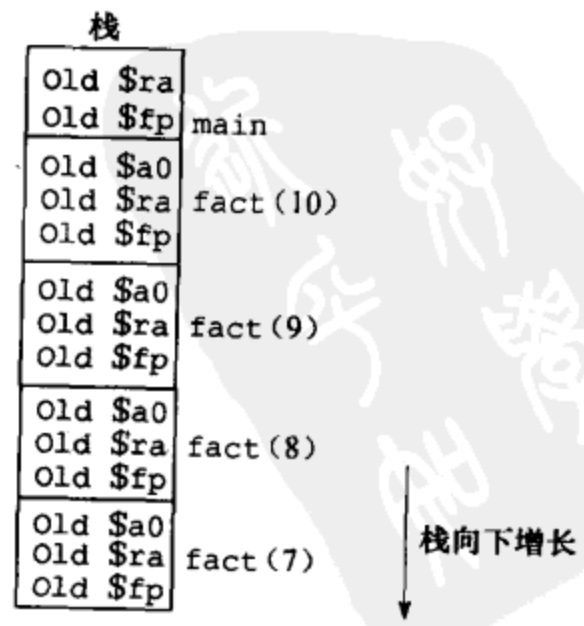
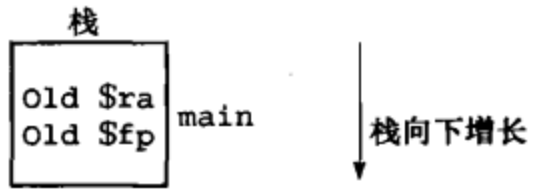


图 B-6-3 调用 fact(7)过程中的栈帧

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。这种改变节省了过程调用和序列返回的一对指令。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。

B.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)
{
    lf(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被调用保存的寄存器中，由于例程或许会调用那些需要使用寄存器 $a0 ~ $a2 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)      # 临时
```

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  #如果(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，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    $a0, $s3
move    $a1, $s0
move    $a2, $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［（function epilogue（跋，后记）］。

```
L1:
    move  $v0, $s2
```

以下的代码是函数末尾，它恢复被保存的寄存器而且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
```

B.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[⊖]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：

寄存器名称	寄存器编号	使用
BadVAddr	8	在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

这 7 个寄存器是 coprocessor 0 处理器的寄存器组的一部分。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 B-7-1 展示了 MIPS 仿真器 SPIM 实现的状态寄存器域。中断掩码域为 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。

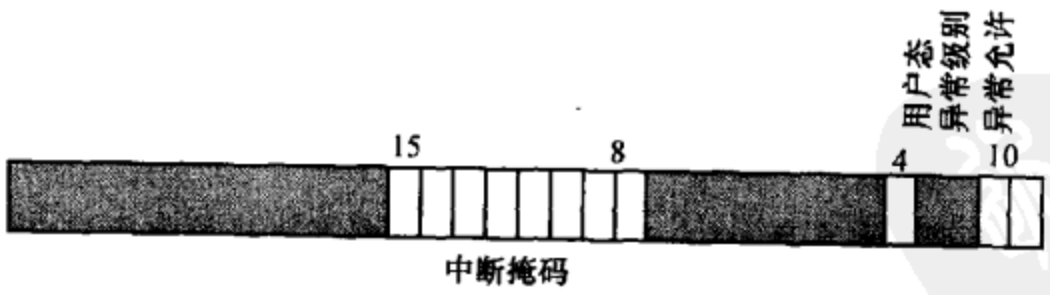


图 B-7-1　状态寄存器

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，

⊖ 中断处理（interrupt handler）：一段代码，作为异常或者中断的执行结果。

⊖ 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-1 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。

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 B-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：

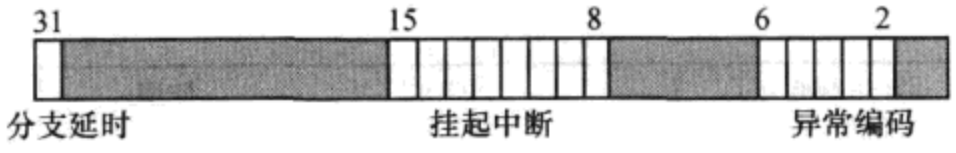


图 B-7-2　原因寄存器

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	取指令的总线错误
7	DBE	加载数据或者存储数据的总线错误
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	算术上溢异常
13	Tr	陷阱
15	FPE	浮点

异常和中断导致 MIPS 处理器跳转到一段代码，地址是 80000180_{16}（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（handler），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。

举例　**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        #保存 $at 寄存器
sw      $a0, save0      #程序不是,不能用
```

```
sw      $a1,save1      #堆栈来保存 $a0, $a1
                       #不需要保存 $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）。如果是这样的话，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。

```
mfc0    $k0, $13       #将原因寄存器中的值传递到 $k0
srl     $a0, $k0,2     #提取 ExcCode 域
andi    $a0, $a0,0xf
bgtz    $a0,done       #分支,如果 ExcCode 是 Int(0)
mov     $a0, $k0       #将原因寄存器中的值传递到 $a0
mfc0    $a1, $14       #将 EPC 寄存器中的值传递到 $a1
jal     print_excp     #打印异常错误信息
```

在运行之前，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       #保存 EPC
        addiu   $k0, $k0,4     #错误、有问题的指令不需要重新执行
        mtc0    $k0, $14       #EPC
        mtc0    $0, $13        #清除原因寄存器
        mtc0    $k0, $12       #固定状态寄存器
        andi    $k0,0xfffd     #清除 EXL 位
        ori     $k0,0x1        #中断允许
        mtc0    $k0, $12
        lw      $a0,save0      #恢复寄存器
        lw      $a1,save1
        mov     $at, $k1
        eret                   #返回到 EPC
        .kdata
save0:  .word 0
save1:  .word 0
```

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跳转到 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。

B.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（console），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 程序可以读取你键入的字符。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（一个例子，因为你键入 control-C，或者因为程序遇到一个断点）时，终端被重新连接到 SPIM，这样你就可以键入 SPIM 的指令了。

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（flag）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。发射者在终端（控制台）显示字符。两个单元完全独立。这意味着，例如，从键盘键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 B-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 $ffff0000_{16}$，实际仅用到它的两个位。位 0 被称作“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。当字符从键盘键入时，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。

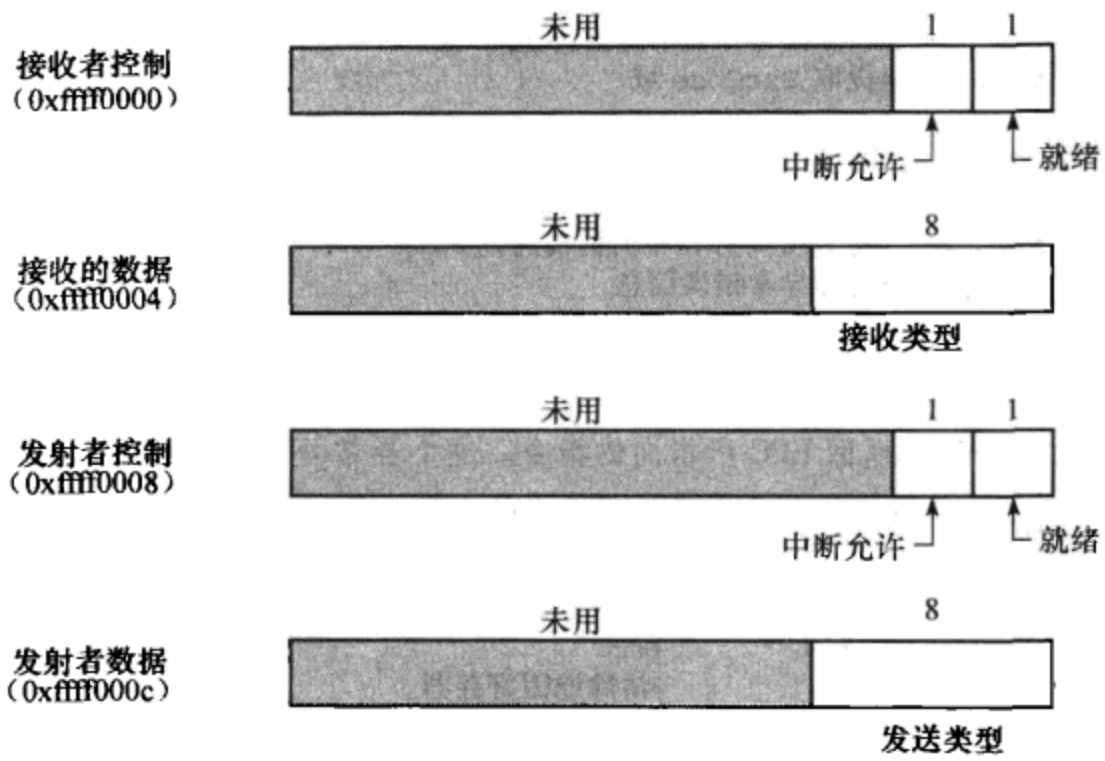


图 B-8-1　终端被 4 个设备寄存器控制着，每个寄存器在给定地址作为一个内存地址仅仅这些寄存器的一些位实际上被使用。别的位通常被读成 0，对它们的写被忽略。

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，无论何时字符被键入，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（见第 B.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 位包含着从键盘键入的最后字符，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键入时才改变。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 位被称作“ready”，而且是只读的。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，终端在硬件 0 级请求一个一个中断，无论发送者是否准备好一个新的字符，而且 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 $ffff000c_{16}$）。当一个值被写入到这个地址处时，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。

B.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 可以读而且可以立即执行汇编语言文件。

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 运行的快得多时，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的更好的环境，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，而不是硅。所以对于添加新指令，构建像多处理器这样的新系统，或者收集数据这样的事情，使用仿真器容易验证也容易修改。

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）。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 章的第 4.2 节）。

MIPS 一般通过汇编语言实现的**虚拟机**[⊜]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。

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

⊖ SPIM 的早期版本（7.0 以前的版本）实现了 MIPS-1 的体系结构，使用原始的 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。

⊜ 虚拟机（virtual machine）：一种虚拟机计算机，它分支和取数指令没有延迟，且指令集比实际硬件更丰富。

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硅实现的机器那样使用。

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：键入一行文本，按回车键，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 CD 上。xspim、pcSpim、spim 的教程和 **SPIM 命令行选项**都在 CD 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 CD 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 CD 上的 **xspim** 教程

令人惊讶的特性

尽管如实的仿真了 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，而这将在实际机器上将导致异常。

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。

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。

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 B.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"`：

服务	系统调用代码	参数	结果
print_int	1	$a0 = integer 整数	
print_float	2	$f12 = float 浮点	
print_double	3	$f12 = double 浮点	
print_string	4	$a0 = string 字符串	
read_int	5		integer (in $v0)
read_foat	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), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

图 B-9-1　系统服务

```
    .data
str:
    .asciiz"the anser = "
    .text
li      $v0,4       #print_str 系统调用代码
la      $a0,str     #打印的字符串的地址
Syscall             #打印字符串
li      $v0,1       #print_int 系统调用代码
li      $a0,5       #打印整数
Syscall             #打印它
```

给 print_int 系统调用传递一个整数并在终端上打印出来。print_float 打印一个浮点数；print_double打印出来一个双精度度数；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。

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 B.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 以及 close 是 UNIX 的标准库调用。

B.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 B-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。

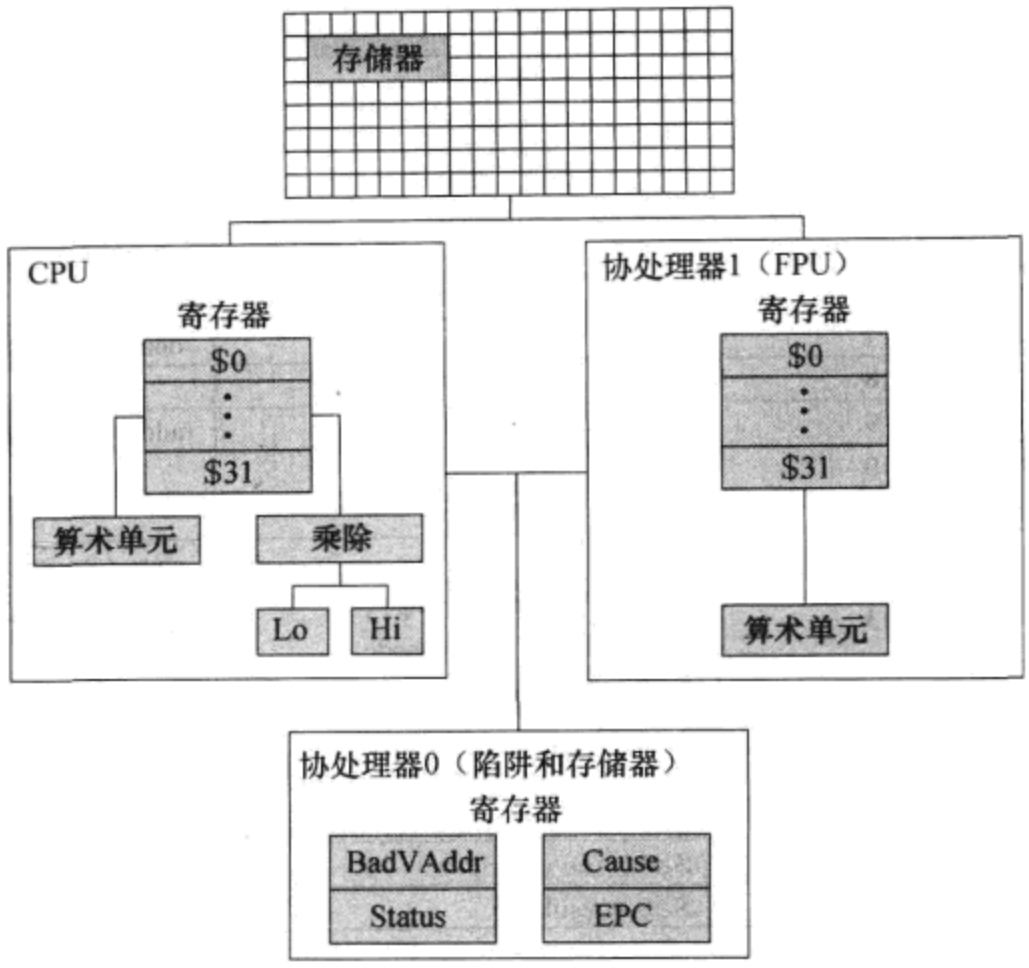


图 B-10-1　MIPS R2000 CPU 和 FPU

B. 10. 1　寻址方式

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
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）。

精解： MIPS 汇编器（SPIM 也一样）通过在对数据存取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方式。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方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。

B.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:   lw            $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 语言惯例：

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 个字节分界。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。

.asciiz str　　将字符串 str 存入主存，并以空字符结束。

.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>　　将后续项放入核心正文段。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 and .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 个字节（SPIM 中则必须为数据段）。

.text <addr>　　将后续项送入用户正文段中。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）。

B.10.3　MIPS 指令编码

图 B-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。

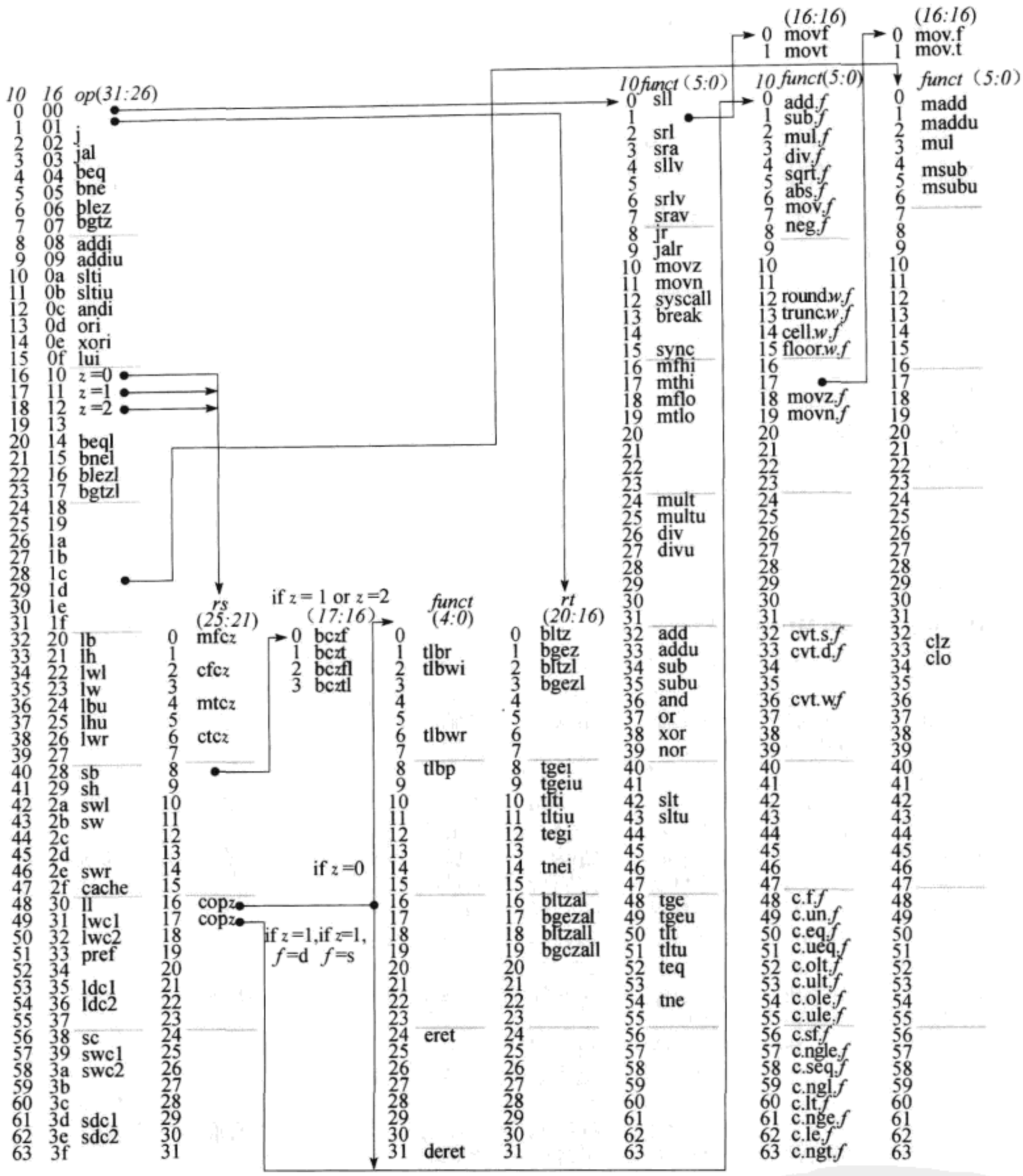


图 B-10-2　MIPS 操作码图

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，由指针进行识别。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“z”分别表示：“0”，“1”，“2”，“3”。如果 rs = 16，则操作由别处定义：如果 z = 0，则操作在第四个字段中定义（4 ~ 0 位）；如果 z = 1，则操作在最后的字段中，且 f = s。如果 rs = 17 且 z = 1，则操作在最后的字段中，且 f = d。

B. 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 rd, rs, rt
```

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

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。

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`）。

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`

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

加法（不带溢出位）

`addu rd, rs, rt`

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

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`

8	rs	rt	imm
6	5	5	16

立即数加（不带溢出位）

`addiu rt, rs, imm`

9	rs	rt	imm
6	5	5	16

将寄存器 rs 与立即数之和存入寄存器 rt 中。

逻辑与

`and rd, rs, rt`

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

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`

0xc	rs	rt	imm
6	5	5	16

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`

0x1c	rs	0	rd	0	0x21
6	5	5	5	5	6

统计起始为 0 的个数

`clz rd, rs`

0x1c	rs	0	rd	0	0x20
6	5	5	5	5	6

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`

0	rs	rt	0	0x1a
6	5	5	10	6

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。

除法（带溢出位）

```
div rdest, rsrc1, src2
```

伪指令

除法（不带溢出位）

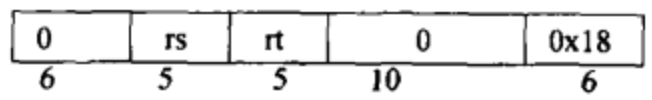
```
divu rdest, rsrc1, src2
```

伪指令

将寄存器 rsrc1 和 src2 的商存入寄存器 rdest。

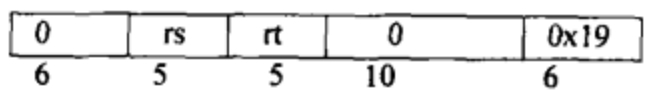
乘法

```
mult rs, rt
```



无符号数乘法

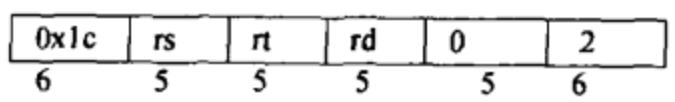
```
multu rs, rt
```



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）

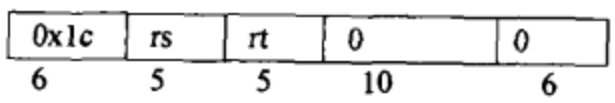
```
mulou rdest, rsrc1, src2
```

伪指令

将寄存器 rsrc1 和 src2 的乘积结果的低 32 位存入寄存器 rdest。

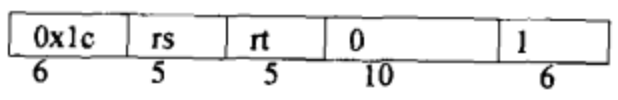
乘加

```
madd rs, rt
```



无符号乘加

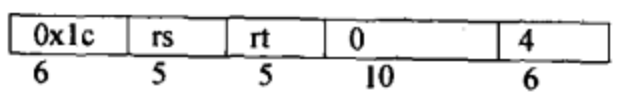
```
maddu rs, rt
```



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。

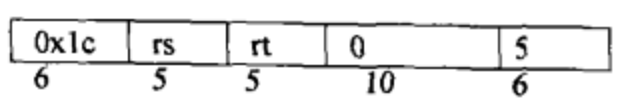
乘减

```
msub rs, rt
```



无符号乘减

```
msub 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）

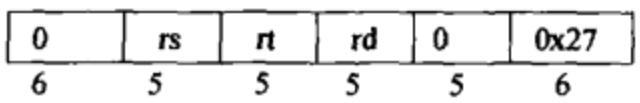
```
negu rdest, rsrc
```

伪指令

将寄存器 rsrc 的相反数存入寄存器 rdest。

异或

nor rd, rs, rt



将寄存器 rs 和 rt 的异或结果存入寄存器 rd。

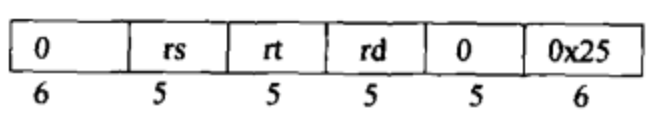
取反

not rdest, rsrc　　伪指令

将寄存器 rsrc 逐位取反存入寄存器 rdest。

逻辑或

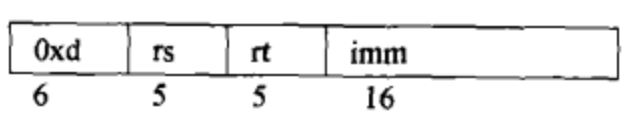
or rd, rs, rt



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

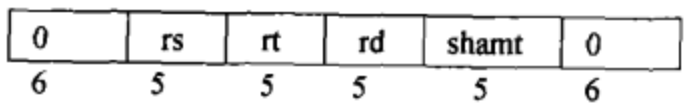
求无符号数的余数

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。

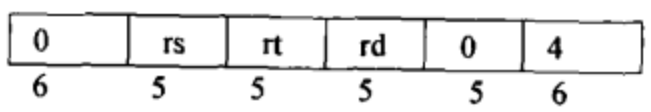
逻辑左移

sll rd, rt, shamt



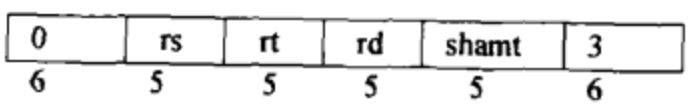
逻辑左移变量

sllv rd, rt, rs



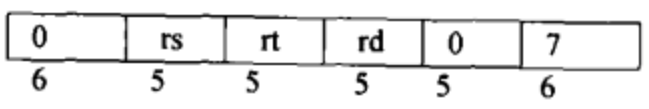
算术右移

sra rd, rt, shamt



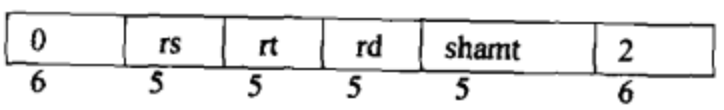
算术右移变量

srav rd, rt, rs



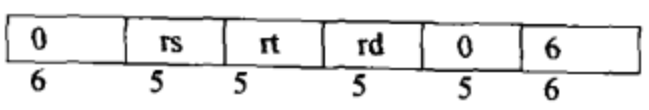
逻辑右移

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

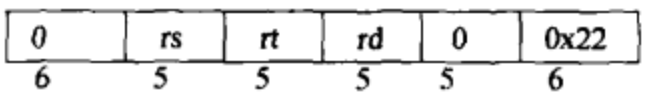
循环右移

`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`。

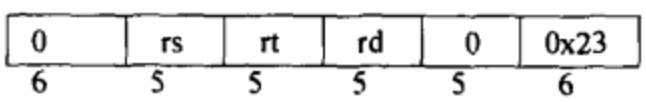
减法（带溢出位）

`sub rd, rs, rt`



减法（不带溢出位）

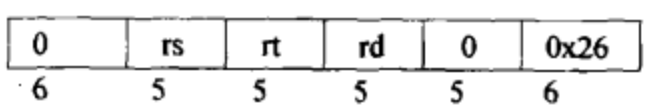
`subu rd, rs, rt`



将寄存器 `rs` 减去寄存器 `rt` 并将结果存入寄存器 `rd`。

异或

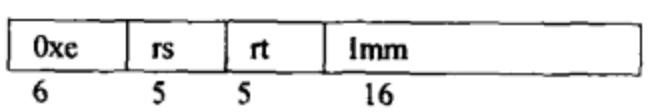
`xor rd, rs, rt`



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`

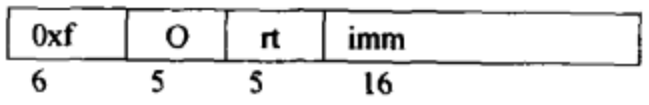


将寄存器 `rs` 和 0 扩展立即数按位逻辑异或的结果存入寄存器 `rt`。

B.10.5 常数操作指令

立即数高位取指令

`lui rt, imm`



将立即数 `imm` 的低半字位存入寄存器 `rt` 的高半字位地址，并将寄存器的低位值置为 0。

取立即数

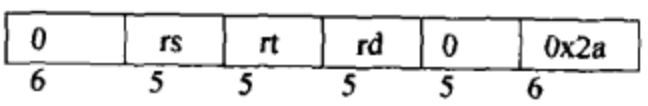
`li rdest, imm`　　伪指令

将立即数 `imm` 存入寄存器 `rdest`。

B.10.6 比较指令

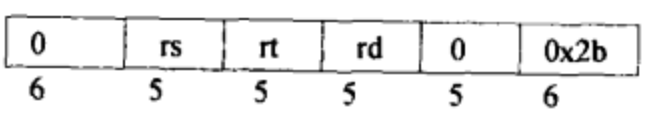
小于指令

`slt rd, rs, rt`



小于无符号数指令

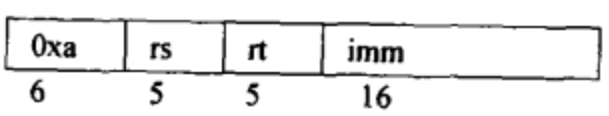
`sltu rd, rs, rt`



若寄存器 `rs` 比 `rt` 小，则将寄存器 `rd` 置为 1；否则，将 `rd` 置为 0。

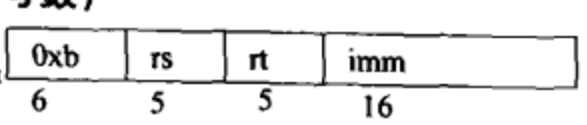
小于立即数

`slti rt, rs, imm`



小与立即数（无符号数）

`sltiu rt, rs, imm`



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`　　伪指令

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`　　伪指令

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

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。

B.10.7　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 215 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，偏移量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。

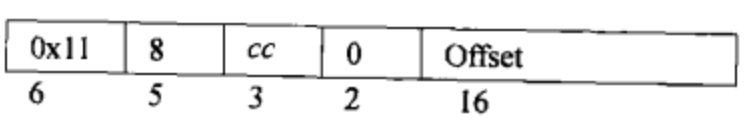
分支指令

`b label`　　伪指令

无条件转移到标记的指令。

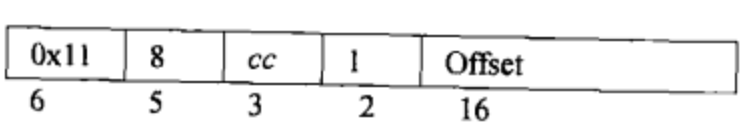
分支协处理器假

`bclf cc label`



分支协处理器真

`bclt cc label`

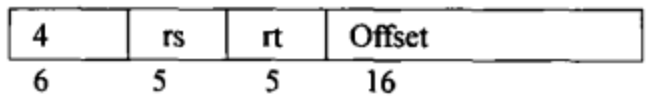


如果浮点协处理器条件标记 *cc* 为假（真），条件转移的指令数由偏移量所指定。如果 cc 被

指令所忽略，条件码标记为0。

相等分支

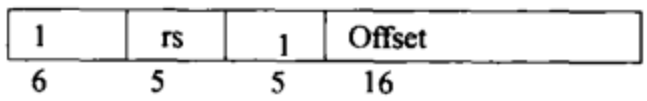
```
beq rs, rt, label
```



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

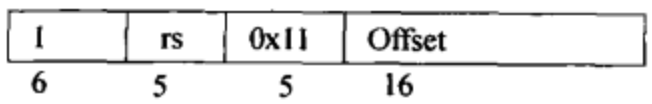
```
bgez rs, label
```



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

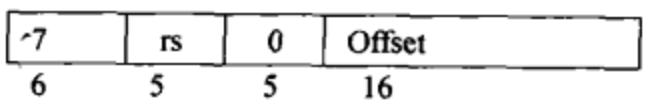
```
bgezal rs, label
```



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

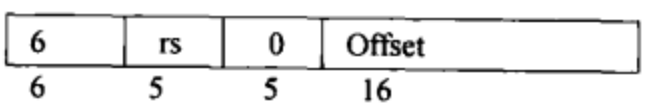
```
bgtz rs, label
```



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

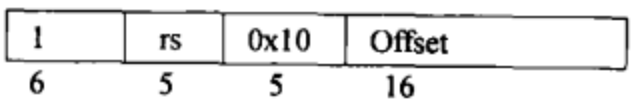
```
blez rs, label
```



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

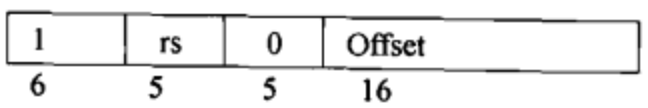
```
bltzal rs, label
```



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

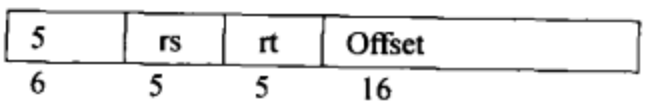
```
bltz rs, label
```



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, rsrc2, label　　伪指令

小于等于无符号数分支

bleu rsrc1, rsrc2, 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

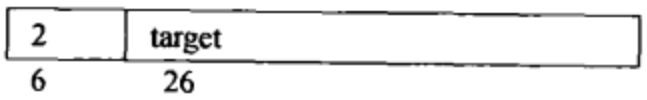
bnez rsrc, label　　伪指令

如果寄存器 rsrc 的值不等于 0，条件转移到标记的指令那里。

B. 10. 8　跳转指令

跳转

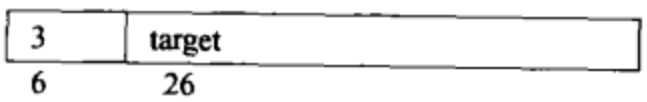
j target



无条件跳转到目标指令。

跳转并链接

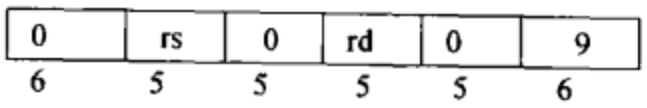
jal target



无条件跳转到目标指令，并将下一条指令地址保存到 $ra 中。

跳转并链接到寄存器

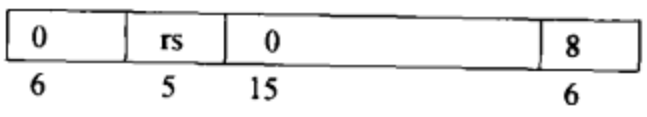
jalr rs, rd



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

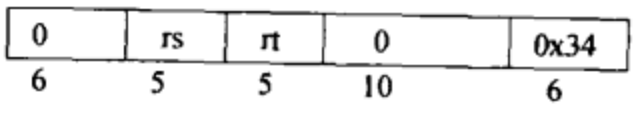


无条件跳转到由寄存器 rs 指定的指令。

B. 10. 9　陷阱指令

等于陷阱

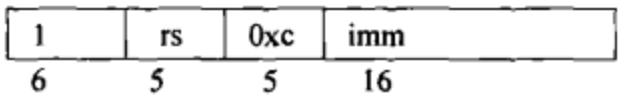
teq rs, rt



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

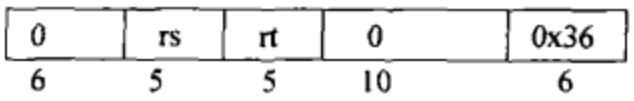
teqi rs, imm



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

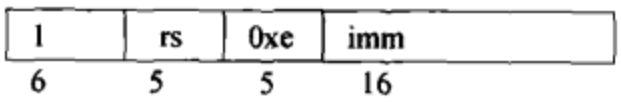
teq rs, rt



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

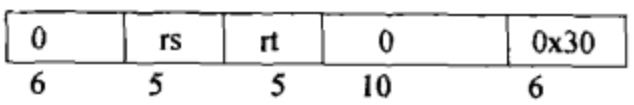
teqi rs, imm



如果寄存器 rs 的值不等于符号扩展值——imm，引发陷阱异常。

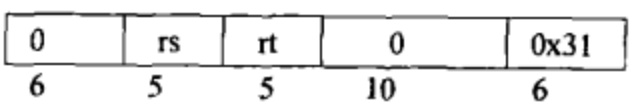
大于等于陷阱

tge rs, rt



大于等于无符号数陷阱

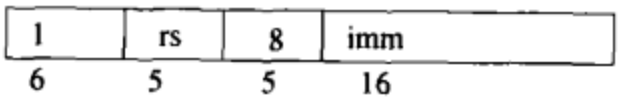
tgeu rs, rt



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。

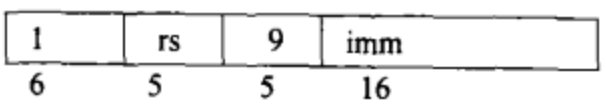
大于等于立即数陷阱

tgei rs, imm



大于等于无符号立即数陷阱

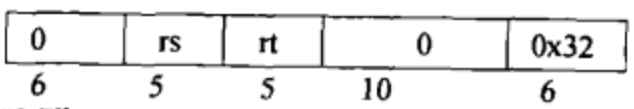
tgeiu rs, imm



如果寄存器 rs 的值大于等于符号扩展值——imm，引发陷阱异常。

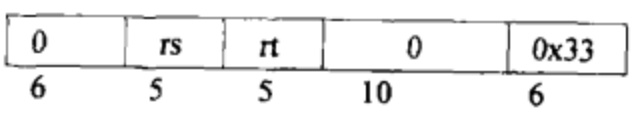
小于陷阱

tlt rs, rt



小于无符号数陷阱

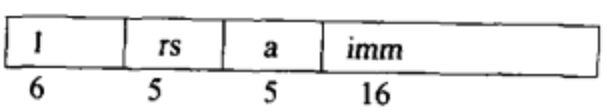
tltu rs, rt



如果寄存器 rs 的值小于寄存器 rt 的值，引发陷阱异常。

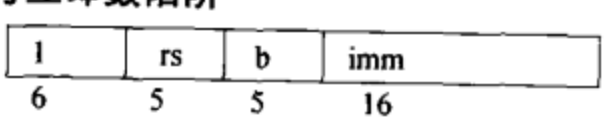
小于立即数陷阱

tlti rs, imm



小于等于无符号立即数陷阱

tltiu rs, imm



如果寄存器 rs 的值小于等于符号扩展值——imm，引发陷阱异常。

B. 10. 10　取数指令

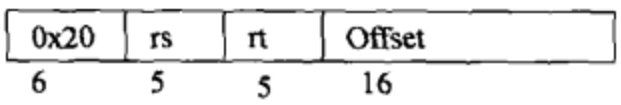
取地址

`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。

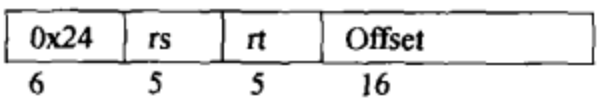
取字节

`lb rt, address`



取字节（无符号）

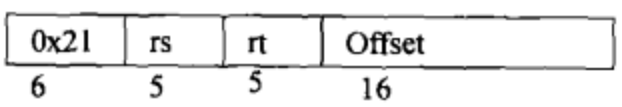
`lbu rt, address`



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`。

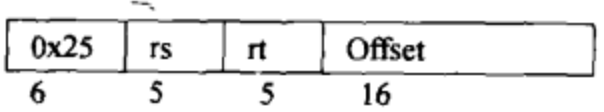
取半字

`lh rt, address`



取半字（无符号）

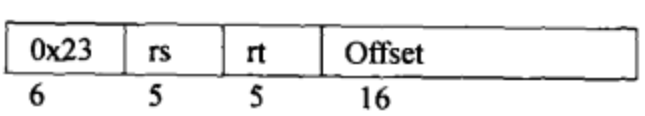
`lhu rt, address`



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`。

取字

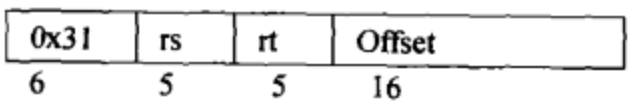
`lw rt, address`



将地址 address 中 32 位数值（字）存入寄存器 `rt` 中。

协处理器 1 取字

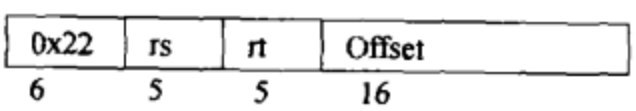
`lwcl ft, address`



将地址 address 中的字以浮点单元的形式存入寄存器 `ft` 中。

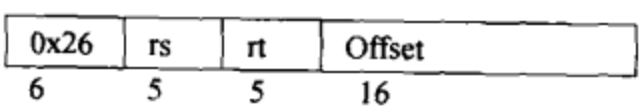
取左半字

`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

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。

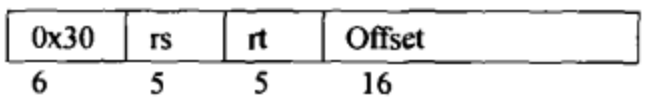
非对齐地址中取字（无符号）

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

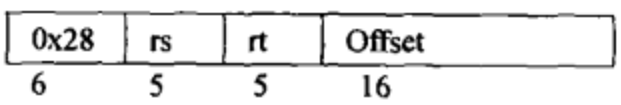


将 address 中 32 位数值存入寄存器 rt 中，并且开始执行原子读—修改—写操作。该操作由条件存指令（sc）来完成，但如果其他处理器对包含有被取字的块进行写操作时，该操作将失败。由于 SPIM 不能模拟多处理器，因而条件存操作总是可以成功执行的。

B.10.11　保存指令

存字节

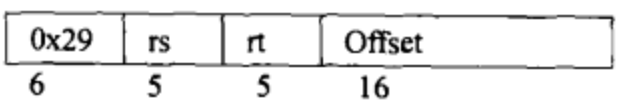
sb rt, address



将寄存器 rt 的低字节保存到地址 address 中。

存半字

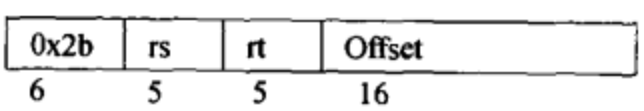
sh rt, address



将寄存器 rt 的低 16 位值（半字）保存到地址 address 中。

存字

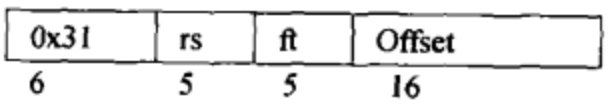
sw rt, address



将寄存器 rt 中的字保存到地址 address 中。

协处理器 1 存字

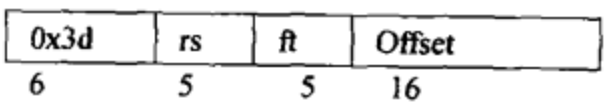
swc1 ft, address



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

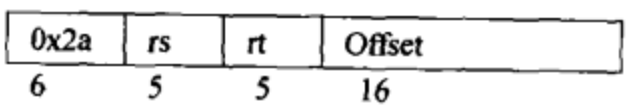
sdc1 ft, address



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。

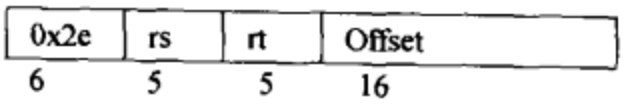
存左半字

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

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`

0x38	rs	rt	Offset
6	5	5	16

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。

B. 10. 12　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

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。

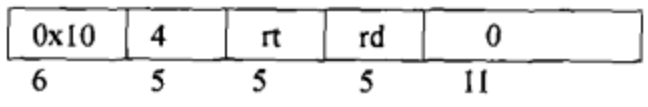
从协处理器 1 中传送双字

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。

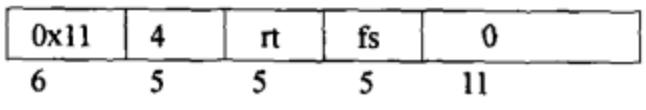
传送到协处理器 0

mtc0 rd, rt



传送到协处理器 1

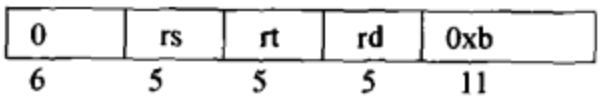
mtc1 rd, fs



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

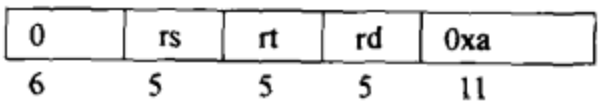
movn rd, rs, rt



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

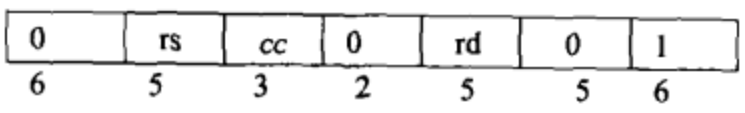
movz rd, rs, rt



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

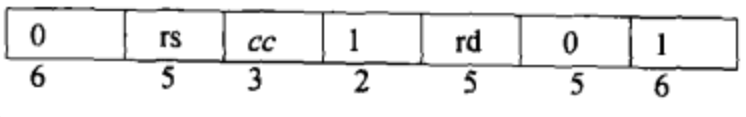
movf rd, rs, cc



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



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。

B. 10. 13　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
6	5	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

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

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

`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。

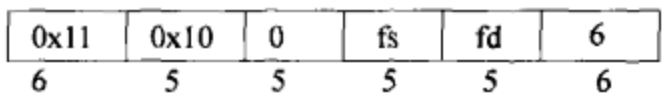
双精度浮点数的传送

`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

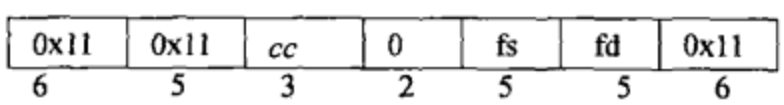
`mov.s fd, fs`



将寄存器 `fs` 中的双精度（单精度）浮点数传送到寄存器 `fd` 中。

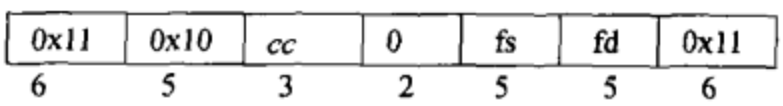
条件为假时双精度浮点数传送

`movf.d fd, fs, cc`



条件为假时单精度浮点数传送

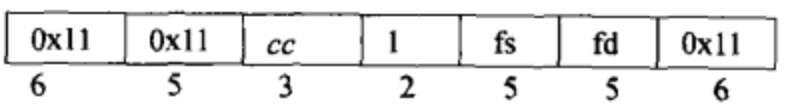
`movf.s fd, fs, cc`



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。

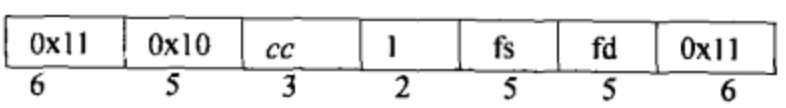
条件为真时双精度浮点数传送

`movt.d fd, fs, cc`



条件为真时单精度浮点数传送

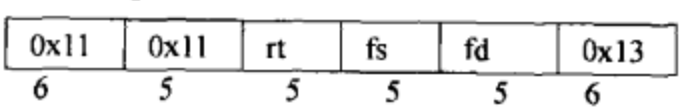
`movt.s fd, fs, cc`



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。

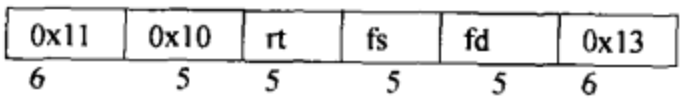
非零条件双精度浮点数传送

`movn.d fd, fs, rt`



非零条件单精度浮点数传送

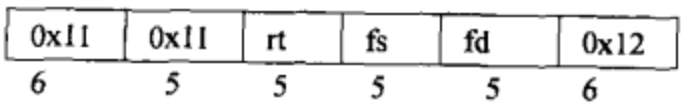
`movn.s fd, fs, rt`



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。

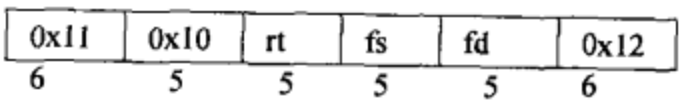
等于零条件双精度浮点数传送

`movz.d fd, fs, rt`



等于零条件单精度浮点数传送

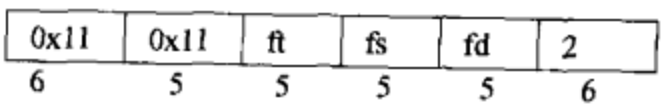
`movz.s fd, fs, rt`



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。

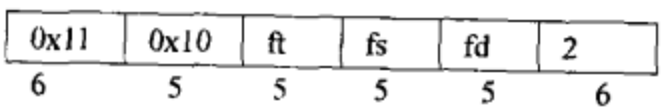
双精度浮点乘

`mul.d fd, fs, ft`



单精度浮点乘

`mul.s fd, fs, ft`



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

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	0f	sf	d	0xc

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

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

B. 10. 14　异常和中断指令

异常返回

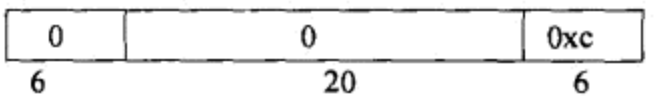
eret

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

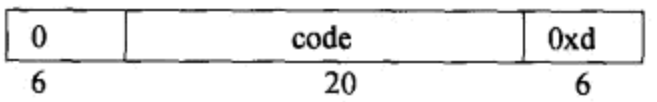
syscall



寄存器 $v0 中保存了 SPIM 提供的系统调用的个数（见图 B-9-1）。

跳出

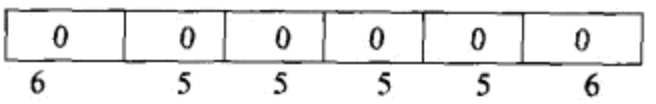
break code



产生异常码，异常 1 为调试程序保留。

空操作

nop



不做任何操作。

B. 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。

B. 12　参考文献

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*）

B. 13　练习题

B. 1 [5] <B. 5 > B. 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。

B.2 [20] <B.6> 用更少的指令重写 `fact` 程序。

B.3 [5] <B.7> 用户程序使用寄存器 `$k0` 或 `$k1` 时总是安全的么?

B.4 [25] <B.7> B.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。

B.5 [15] <B.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。这样的话，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。

B.6 [5] <B.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 B.9 节介绍的 SPIM 系统调用。

B.7 [5] <B.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 B.9 节介绍的 SPIM 系统调用。你可以任意中断程序。

B.8 [5] <B.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”。

B.9 [25] <B.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。

在 SPIM 上验证你的程序。

B.10 [10] <B.6, B.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 到 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;
}
```

华章计算机科学丛书经典推荐



深入理解计算机系统（原书第2版）
(美) Randal E. Bryant
David R. O'Hallaron 著
ISBN 978-7-111-32133-0
定价：99.00元

深入理解计算机系统（英文版第2版）
(美) Randal E. Bryant
David R. O'Hallaron 著
ISBN 978-7-111-32631-1
定价：128.00元



软件工程：实践者的研究方法（原书第7版）
(美) Roger S. Pressma 著
ISBN 978-7-111-33581-8
定价：79.00元



软件工程：实践者的研究方法（英文版第7版）
(美) Roger S. Pressma 著
ISBN 978-7-111-31871-2
定价：75.00元



软件工程：面向对象和传统的方法（英文版第8版）
(美) Stephen R. Schach 著
ISBN 978-7-111-34196-3
定价：79.00元

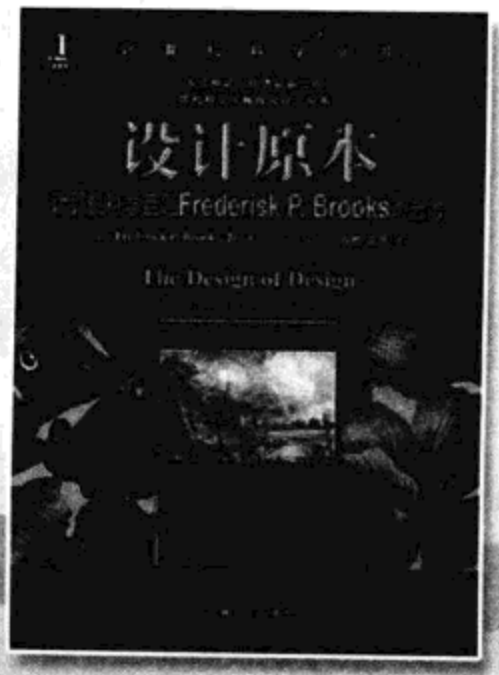


计算机组织与结构：性能设计（原书第8版）
(美) William Stallings 著
ISBN 978-7-111-32878-0
定价：79.00元



操作系统：精髓与设计原理（原书第6版）
(美) William Stalling 著
ISBN 978-7-111-30426-5
定价：69.00元

- Stallings，大师级的人物，涉猎计算机安全、网络、体系结构等多方面，堪称计算机界的全才。
- 本书是其经典著作之一，得到全球计算机教育界和工程技术人员的好评！
- 以当代最流行的操作系统—Windows Vista、UNIX和Linux为例，全面清楚地展现了当代操作系统的本质和特点，具有先进性和适应性。



设计原本 ISBN 978-7-111-32557-4 定价：55.00



Java语言程序设计 基础篇（原书第8版）
(美) Y. Daniel Liang 著
ISBN 978-7-111-34081-2
定价：75.00元

Java语言程序设计 进阶篇（原书第8版）
(美) Y. Daniel Liang 著
ISBN 978-7-111-34236-6
定价：79.00元

教师服务登记表

尊敬的老师：

您好！感谢您购买我们出版的_____教材。

机械工业出版社华章公司为了进一步加强与高校教师的联系与沟通，更好地为高校教师服务，特制此表，请您填妥后发回给我们，我们将定期向您寄送华章公司最新的图书出版信息！感谢合作！

个人资料（请用正楷完整填写）

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

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

样书申请			
已出版著作		已出版译作	
是否愿意从事翻译/著作工作 □是 □否		方向	
意见和建议			

填妥后请选择以下任何一种方式将此表返回：（如方便请赐名片）

地 址：北京市西城区百万庄南街1号 华章公司营销中心 邮编：100037

电 话：(010) 68353079 88378995 传真：(010)68995260

E-mail:hzedu@hzbook.com marketing@hzbook.com 图书详情可登录http://www.hzbook.com网站查询