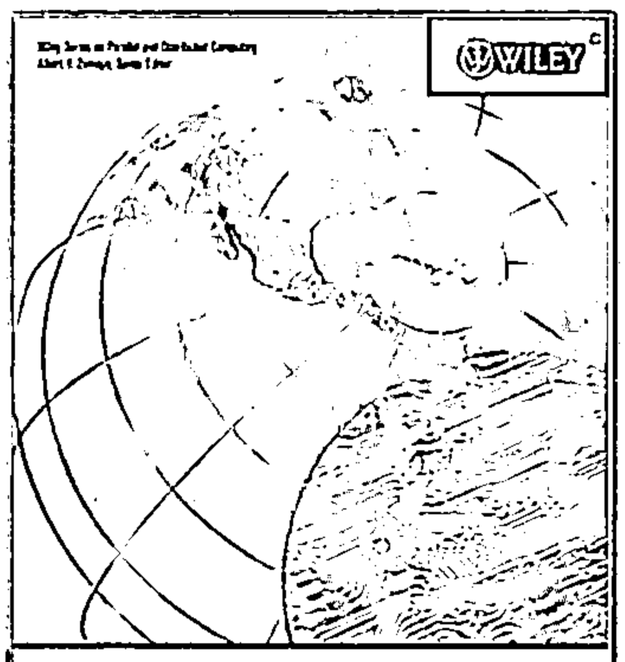
国外计算机科学教材系列

计算机组成与体系结构

Fundamentals of Computer Organization and Architecture



[科] Mostafa Abd-El-Barr
[美] Hesham El-Rewini
著

陆鑫达 翁楚良 林新华 译

電子工業出版社
Publishing House of Electronics Industry
http://www.phei.com.cn

国外计算机科学教材系列

计算机组成与体系结构

Fundamentals of Computer Organization and Architecture

［科］ Mostafa Abd-El-Barr

［美］ Hesham El-Rewini 著

陆鑫达　翁楚良　林新华　译

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

本书主要论述计算机组成和计算机体系结构中的基础问题，主要内容包括：计算机系统导论，指令集体系结构和设计，汇编语言程序设计，计算机算术，处理部件设计，存储系统设计，输入/输出系统设计及组成结构，流水线设计技术，RISC 计算机，以及多处理器导论。

本书取材合理，内容精练，结构清晰，可作为计算机专业及电气工程专业高年级本科生的教材，也可作为相关工程师和程序设计人员的参考书。

Mostafa Abd-El-Barr, Hesham El-Rewini: **Fundamentals of Computer Organization and Architecture.**
ISBN 0-471-46741-3
Copyright © 2005, John Wiley & Sons, Inc.
All Rights Reserved. Authorized translation from the English language edition published by Wiley Publishing, Inc.
No part of this book may be reproduced in any form without the written permission of Wiley Publishing, Inc.
Simplified Chinese translation edition Copyright ©2005 by Publishing House of Electronics Industry.

本书中文简体字翻译版由 Wiley Publishing, Inc 授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2004-5757

图书在版编目（CIP）数据

计算机组成与体系结构 /（科）巴尔（Barr, M. A.）等著；陆鑫达等译.
北京：电子工业出版社，2005.12
（国外计算机科学教材系列）
书名原文：Fundamentals of Computer Organization and Architecture
ISBN 7-121-01955-8

Ⅰ.计… Ⅱ.①巴… ②陆… Ⅲ.①计算机体系结构－教材 Ⅳ.TP303

中国版本图书馆 CIP 数据核字（2005）第 135153 号

责任编辑：贺瑞君　　特约编辑：吴　颖
印　　刷：北京市顺义兴华印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
经　　销：各地新华书店
开　　本：787 × 1092　1/16　　印张：12.25　　字数：345 千字
印　　次：2005 年 12 月第 1 次印刷
定　　价：23.00 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

出 版 说 明

21世纪初的5至10年是我国国民经济和社会发展的重要时期，也是信息产业快速发展的关键时期。在我国加入WTO后的今天，培养一支适应国际化竞争的一流IT人才队伍是我国高等教育的重要任务之一。信息科学和技术方面人才的优劣与多寡，是我国面对国际竞争时成败的关键因素。

当前，正值我国高等教育特别是信息科学领域的教育调整、变革的重大时期，为使我国教育体制与国际化接轨，有条件的高等院校正在为某些信息学科和技术课程使用国外优秀教材和优秀原版教材，以使我国在计算机教学上尽快赶上国际先进水平。

电子工业出版社秉承多年来引进国外优秀图书的经验，翻译出版了“国外计算机科学教材系列”丛书，这套教材覆盖学科范围广、领域宽、层次多，既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。这些教材涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。同时，我们也适当引进了一些优秀英文原版教材，本着翻译版本和英文原版并重的原则，对重点图书既提供英文原版又提供相应的翻译版本。

在图书选题上，我们大都选择国外著名出版公司出版的高校教材，如Pearson Education培生教育出版集团、麦格劳－希尔教育出版集团、麻省理工学院出版社、剑桥大学出版社等。撰写教材的许多作者都是蜚声世界的教授、学者，如道格拉斯·科默（Douglas E. Comer）、威廉·斯托林斯（William Stallings）、哈维·戴特尔（Harvey M. Deitel）、尤利斯·布莱克（Uyless Black）等。

为确保教材的选题质量和翻译质量，我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本系列教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师、博士，也有积累了几十年教学经验的老教授和博士生导师。

在该系列教材的选题、翻译和编辑加工过程中，为提高教材质量，我们做了大量细致的工作，包括对所选教材进行全面论证；选择编辑时力求达到专业对口；对排版、印制质量进行严格把关。对于英文教材中出现的错误，我们通过与作者联络和网上下载勘误表等方式，逐一进行了修订。

此外，我们还将与国外著名出版公司合作，提供一些教材的教学支持资料，希望能为授课老师提供帮助。今后，我们将继续加强与各高校教师的密切联系，为广大师生引进更多的国外优秀教材和参考书，为我国计算机科学教学体系与国际教学体系的接轨做出努力。

电子工业出版社

教材出版委员会

主　任　　杨芙清　　北京大学教授
中国科学院院士
北京大学信息与工程学部主任
北京大学软件工程研究所所长

委　员　　王　珊　　中国人民大学信息学院院长、教授

胡道元　　清华大学计算机科学与技术系教授
国际信息处理联合会通信系统中国代表

钟玉琢　　清华大学计算机科学与技术系教授
中国计算机学会多媒体专业委员会主任

谢希仁　　中国人民解放军理工大学教授
全军网络技术研究中心主任、博士生导师

尤晋元　　上海交通大学计算机科学与工程系教授
上海分布计算技术中心主任

施伯乐　　上海国际数据库研究中心主任、复旦大学教授
中国计算机学会常务理事、上海市计算机学会理事长

邹　鹏　　国防科学技术大学计算机学院教授、博士生导师
教育部计算机基础课程教学指导委员会副主任委员

张昆藏　　青岛大学信息工程学院教授

译 者 序

本书是将“计算机组成”和“计算机体系结构”内容组合在一起的一本教材。这样做的最大优点是，可避免过去分开编写“计算机组成”和“计算机体系结构”这两门教材时必不可少的内容重复和交叉，从而导致两者不一致的缺陷，应该说这是本书的特色之一。此外，本书在指令集体系结构和设计这一章之后，专门叙述了有关汇编语言程序设计的内容，使读者对汇编语言程序设计的基本概念和具体的编程方法有了初步的掌握，这是本书的另一特色。本书的内容包括：指令集体系结构和设计，汇编语言程序设计，计算机算术，处理部件设计，存储系统设计，输入/输出设计和组成结构，流水线技术，RISC 计算机，以及多处理机系统导论。对计算机发展的历史背景也做了较全面和公正的论述。本书中对关键概念的叙述，往往通过实例来加以说明，尽可能地展示了这些概念的实际应用。可以说这是本书的又一特色。此外，本书的每一章都有小结、适量的习题，以及相应的参考文献。

本书各章节组织的逻辑性较强，叙述浅显易懂，从而能使读者较快地掌握计算机专业的关键知识点。本书适合作为计算机科学、计算机工程和电气工程本科生的计算机专业课教材，对在职的工程师和程序员也是一本很有价值的参考书。

本书的翻译工作由陆鑫达教授主持，并翻译了目录、前言、第 1～3 章、第 11 章以及索引。翁楚良讲师翻译了第 4～7 章，林新华助教翻译了第 8～10 章。译稿全文由陆鑫达教授进行了统一和校对。

由于翻译时间较紧，译文中若有错误或不妥之处，敬请广大读者不吝指正。

最后，要感谢电子工业出版社的有关策划和编辑人员，正是他们的大力支持才使本书得以顺利出版。

前　言

本书主要面向计算机工程、计算机科学和电气工程等专业的学生，适合作为一学期的“计算机组成和汇编语言”课程及一学期的“计算机体系结构”课程的教材，并且本书假定准备学习“计算机组成”或“计算机体系结构”的学生已经学习过“数字逻辑设计基础”课程和“高级计算机语言导论”等课程。

本书作者长期从事计算机组成和计算机体系结构课程的教学工作，本书正是他们15年教学经验的结晶。书中所用到的大部分材料已在大学本科课堂中使用过。本书着眼于两个基本的计算机视角。第一个是程序员的视角，而第二个是计算机整体结构和功能的视角。前一个视角着眼于通常在低年级计算机组成和汇编语言课程中所教授的内容，而第二个视角则涉及了通常在高年级计算机体系结构课程中所教授的内容。下面，我们将逐章对本书中的内容进行介绍。这样做的目的在于向教师、学生以及实践工程师、科学工作者提供足够多的信息，以便帮助他们选择其中合适的一章或若干章节进行阅读或研究。

第1章是后续各章的基础。在这一章中，我们的论述将从对计算机系统发展简短历史的回顾开始，其目的在于让读者了解影响计算机性能的因素，并希望能预测计算机的未来发展。此外还将引入有关通用和专用机的一般性问题。计算机系统可以通过处于若干抽象层间的接口加以定义，其中每一层将为它的上一层提供功能支持。处于应用程序和高级语言之间的接口称为语言体系结构。指令集体系结构定义了基本指令集和运行时间及I/O控制之间的接口。另一个有关计算机体系结构的不同定义建立在四个基本概念之上，它们是结构、组成、实现和性能。结构定义了各个硬件部件之间的互连；组成定义了各个部分之间的动态相互影响和管理；实现定义了硬件部分的详细设计，而性能说明了计算机系统的行为。第1章中还包括了体系结构的发展和风格。第1章的最后一部分将专门讨论不同CPU性能的评测。

第2章和第3章将介绍指令集体系结构和汇编语言编程。第2章涉及指令集体系结构与设计中的基本原理。我们将从向存储器存储信息和从存储器中检索信息这一问题开始，接着讨论若干不同的寻址方式。此外还将用一定篇幅解释指令的执行和排序。这一章在编写完成简单编程任务的示例代码段时将展示所提及的寻址方式的应用，以及一些指令特征。在第2章内容的基础上，第3章将考虑有关汇编语言的编程问题。我们将介绍由程序员所看到的假想机以及用来表示机器模型中不同指令的助记符和语法。此后将讨论汇编程序的执行和X86 Intel CISC系列汇编语言示例。

第4章和第5章涉及算术电路的设计和分析，以及中央处理器(CPU)的设计。第4章将向读者介绍有关算术操作以及用于支持计算机中计算的电路的基本问题。这里首先将介绍数的表示、基数转换和整数算术，特别要介绍用来完成整数加法、减法、乘法和除法的一些算法以及硬件方法。对于浮点算术，我们将介绍浮点表示、浮点操作以及浮点硬件的设计。第5章所涉及的主要问题是有关CPU组成和设计的。CPU的主要功能是执行存储在计算机存储器中的一组指令。一个简单的CPU由一组寄存器、算术逻辑部件(ALU)和控制部件(CU)组成。其中首先介绍了取指令的读取－执行周期以及CPU寄存器组设计的基本原理。这些基本原理已应用于实际机器的设计，如80x86和MIPS系列，此外还详细讨论了典型CPU数据通路和控制器的设计。

第6章和第7章专门讨论存储器系统设计。一个典型的存储器层次结构从一个小而贵但相当快速的、称为高速缓存存储器(cache)的部件开始。层次结构中在cache之下的是较大、、稍便宜但相对较慢的主存储器部件。高速缓存存储器和主存储器由固体半导体材料构成。在它们之下的是一个通常由磁盘(硬盘)和磁带组成的磁存储器,该磁存储器很大,但较便宜且速度很慢。第6章首先分析了影响成功构成一台计算机的存储器层次结构的因素,这一章的其余部分则叙述了高速缓冲存储器的设计和分析。有关主存储器和虚拟存储器的设计和分析则在第7章中论述。第7章还将对不同只读存储器的实现进行讨论。

在任何现代计算机系统中,输入/输出(I/O)都起着关键的作用。对I/O操作、部件和接口基本原理的清晰理解和正确评价是极其重要的。第8章的重点是研究I/O的设计和组成。我们将涉及有关可编程和中断驱动I/O的基本问题。此外将对80x86和MC9328MX1/MXL AITC实际机器中的中断体系结构进行解释。然后我们将讨论直接存储器访问(DMA)、总线(同步和异步)及其仲裁方法。第8章的内容将以对I/O接口的讨论结束。

提高一个处理器的指令执行速率存在两种基本技术,即增加时钟速率,以便减少指令的执行时间,或是增加可同时执行的指令数。流水线技术和指令级并行是后一技术的例子。流水线技术是第9章讨论的重点。它的基本思想是使处理器同时处理多条指令。这可以通过将一条指令的执行分解到若干子部件(级)上来完成,每一个子部件完成所需操作的一部分,即取指令、指令译码、取操作数和存结果。我们将介绍流水线处理器的性能衡量,讨论造成指令流水冲突的主要问题及某些可能的解决方法。此外,我们还将描述算术流水线的概念,以及在设计这种流水线时将会遇到的问题。我们将通过介绍ARM 1026EJ-S和UltraSPARC-III这两个流水线处理器来结束这一章。

第10章专门介绍精简指令集计算机(RISC)。这类机器代表了计算机体系结构的重要转变。RISC强调合理分配资源来增强体系结构,从而使得执行最常用和最耗时的指令变得最为有效。基于RISC的机器可以由一些共同属性描述,比如简单而精练的指令集、固定的指令格式、一个机器周期一条指令、流水线的指令读取和执行部件、数量充足的通用寄存器(或优化的编译器代码生成),载入/存储式的存储器操作和硬连线的控制器设计。这一章中,我们的讨论从RISC体系结构的发展开始。然后讨论RISC发展中的一个基本概念:重叠寄存器窗口。最后介绍了应用基本RISC原理制造的机器,如BerkeleyRISC、Stanford MIPS,Compaq Alpha和SUN UltraSparc。

在讨论了多处理器设计和分析的基本问题并指出单流机器的主要限制之后,第11章介绍多处理器系统的基本概念 。在多处理器系统中,一些(两个或多个)处理器以某种方式相连,从而允许它们同时参与一个单一任务。使用多处理器的优点是只需简单连接许多较小的现有计算机就可以创建出功能强大的计算机。此外,由许多单处理器系统组成多处理机系统比构造一台高性能的单处理器系统更为经济。我们介绍了许多连接多个处理器时所使用的拓扑结构、计算机体系结构的不同分类方案和基于拓扑的互连网络的分类。对MIMD(多指令多数据)的两种存储器组成方案也做了介绍,即共享存储器和消息传递。这一章以关于多处理器的分析和性能指标的内容结束。感兴趣的读者可以参阅《先进计算机体系结构与并行处理》① 一书。

从以上对本书各章的介绍中我们可以清楚地看到本书的各章在很大程度上是相互独立、自成体系的。我们相信这样的方式有助于教师针对课程目的有选择性地讲授部分章节。然而,经验告诉我们,第1章到第5章以及第8章的内容适用于“计算机工程”及“电气工程”学生的“计算

① 该书与本书同期出版,电子工业出版社——编者注。

机组成和汇编语言”课程，而第 1 章、第 6 章、第 7 章、第 9 章到第 11 章适用于“计算机体系结构”课程。实践工程师和科学工作者可以通过本书各章节的预览有选择性地参阅单独的一章或某些章节。例如，如果想了解更多存储器设计的内容，可以参考第 6 章和第 7 章的内容。

致谢

我们要向所有协助完成此书的人表示由衷的感谢。他们是 Saskatchewan，SMU；KFUPM 大学“计算机组成和计算机结构”课程的学生；Kuwait 大学使用了本书一些章节的草稿并提出了宝贵的反馈意见和建议，对他们表示感谢。我们的同事 Donald Evan，Fatih Kocan，Peter Seidel，Mitch Thornton，A. Naseer，Habib Ammari 和 Hakki CanKaya 提出了积极的意见和很好的建议，使得本书在形式和表述方面得到了明显的改善。在 John Wiley 公司的安排下我们还接受了许多不知名书评家的建议和指正。此外，还要特别感谢这套丛书的编辑 Albert Y. Zomaya 和 Jonh Wiley 公司的 Val Moliere，Kirsten Rohstedt 和 Christine Punzo，是他们使这本书最终成型。当然，此书还存在许多错误和不足之处，欢迎读者批评指正。最后，我们要感谢我们的家人，感谢他们在本书编写过程中给予我们的支持和鼓励。

Mostafa Abd-El-Barr

Hesham El-Rewini

目　录

第 1 章　计算机系统导论 …… 1

1.1　历史背景 …… 1

1.2　体系结构的发展和风格 …… 3

1.3　技术进展 …… 3

1.4　性能衡量 …… 4

1.5　小结 …… 8

习题 …… 8

参考文献和推荐读物 …… 10

参考网站 …… 10

第 2 章　指令集体系结构和设计 …… 11

2.1　存储单元和操作 …… 11

2.2　寻址方式 …… 13

2.3　指令类型 …… 19

2.4　程序设计举例 …… 22

2.5　小结 …… 24

习题 …… 24

参考文献和推荐读物 …… 25

第 3 章　汇编语言程序设计 …… 26

3.1　一台简单机器 …… 26

3.2　指令助记符和句法 …… 28

3.3　汇编器指令和命令 …… 30

3.4　程序汇编和执行 …… 31

3.5　举例:x86 系列 …… 34

3.6　小结 …… 40

习题 …… 40

参考文献和推荐读物 …… 41

第 4 章　计算机算术运算 …… 42

4.1　数字系统 …… 42

4.2　整数运算 …… 44

4.3　浮点运算 …… 53

4.4　小结 …… 56

习题 …… 56

参考文献和推荐读物 …… 57

第 5 章　处理单元设计 …… 58
5.1　CPU 基础 …… 58
5.2　寄存器组 …… 59
5.3　数据通路 …… 62
5.4　CPU 指令周期 …… 64
5.5　控制器 …… 66
5.6　小结 …… 73
习题 …… 74
参考文献和推荐读物 …… 75

第 6 章　存储系统设计 I …… 76
6.1　基本概念 …… 76
6.2　高速缓冲存储器 …… 77
6.3　小结 …… 93
习题 …… 93
参考文献和推荐读物 …… 94
参考网站 …… 95

第 7 章　存储系统设计 II …… 96
7.1　主存 …… 96
7.2　虚拟存储器 …… 101
7.3　只读存储器 …… 111
7.4　小结 …… 112
习题 …… 113
参考文献和推荐读物 …… 114
参考网站 …… 114

第 8 章　输入/输出的设计与组成 …… 115
8.1　基本概念 …… 115
8.2　编程 I/O …… 117
8.3　中断驱动 I/O …… 119
8.4　直接存储器访问 …… 125
8.5　总线 …… 127
8.6　I/O 接口 …… 129
8.7　小结 …… 130
习题 …… 131
参考文献和推荐读物 …… 131
参考网站 …… 132

第 9 章　流水线设计技术 …… 133
9.1　一般概念 …… 133
9.2　指令流水线 …… 134
9.3　流水线处理器实例 …… 144

9.4 指令级并行 …… 148
9.5 算术流水线 …… 149
9.6 小结 …… 152
习题 …… 152
参考文献和推荐读物 …… 153
参考网站 …… 154

第 10 章 精简指令集计算机 …… 155
10.1 RISC/CISC 进化周期 …… 155
10.2 RISC 设计原则 …… 155
10.3 重叠寄存器窗口 …… 157
10.4 RISC 对 CISC …… 157
10.5 先驱 RISC 机器 …… 159
10.6 高级 RISC 机器实例 …… 162
10.7 小结 …… 166
习题 …… 166
参考文献和推荐读物 …… 167
参考网站 …… 167

第 11 章 多处理机系统导论 …… 168
11.1 引言 …… 168
11.2 计算机体系结构的分类 …… 169
11.3 SIMD 方式 …… 174
11.4 MIMD 方式 …… 176
11.5 互连网络 …… 179
11.6 分析和性能指标 …… 181
11.7 小结 …… 181
习题 …… 181
参考文献和推荐读物 …… 182

第1章　计算机系统导论

计算机工业的技术进展是两支主力军做出一系列巨大努力的结果。这两支主力军就是以大学研究中心为代表的学术界和以计算机公司为代表的工业界。但是,公平地讲,当今计算机工业的技术进展应归功于大学研究中心的始创。为了了解当前计算机工业中的技术进展,人们不得不对计算机的历史和它的发展进行追溯。对历史回顾旨在了解影响今天的计算机性能的因素,并希望能预测计算机的未来发展。我们日常生活中所使用的绝大多数计算机都是通用机。这些机器在构造时并不考虑专门的应用,而是考虑能完成许多不同应用所需的性能。这些通用机不同于为特殊用途服务(制造)的专用机。在 1.1 节中,我们将对计算机诞生和发展的历史背景进行简短的叙述。

计算机系统已通过在若干抽象层上的接口加以定义,其中每一层将为它的上一层提供功能的支持。这些层包括应用程序、高级语言和机器指令集。根据系统不同级间的接口,就可以定义许多计算机体系结构。处于应用程序和高级语言之间的接口称为语言体系结构,而指令集体系结构则定义了基本指令集与运行时间及 I/O 控制之间的接口。计算机体系结构的另一个不同定义建立在四个基本概念 之上,它们是结构、组成、实现和性能。结构定义了各个硬件部件之间的互连,组成定义了各个部件之间的动态相互影响和对它们的管理,实现定义了硬件部件的详细设计,而性能说明了计算机系统的行为。有关体系结构的发展和风格将在 1.2 节中叙述。

1.3 节中将论述许多技术的发展情况。本章的最后将详细讨论 CPU 性能的衡量。

1.1　历史背景

在这一节中,我们将提供一个计算机工业发展的历史背景。我们在一开始就要指出建造计算机并不是源出一处。有充分理由相信建造第一台计算机的尝试存在于地球上不同的地方,我们也坚定地相信建造一台计算机需要合作。因此,我们只提及机器和它的所在地,而不把它们与某个研究者的名字联系在一起。

较为公平地说,曾经建造过的第一台程序控制(机械)计算机是 Z1(1938 年)。接着在 1939 年出现了第一台定点算术操作程序控制计算机 Z2。在大学中,第一次有记载的建造计算机的尝试可以追溯到 20 世纪 40 年代早期的美国依阿华州立大学。该大学的研究者们建造了一台小型的专用电子计算机,但这台机器从未真正运行过。几乎在同时,即 1941 年,在德国有报告说完成了功能完全可编程的专用机器 Z3 的设计。有迹象表明由于缺少资金,该设计无法得以实现。历史还记载在进行这两个建造专用电子计算机的尝试时,来自世界上不同地区的研究者们通过对正在从事该项研究工作的实验室和研究所的访问获得了第一手经验。可以设想正是这种访问和思想的交流,使得访问者们在返回他们自己的实验室后开始了类似的项目研究。

就通用机而言,记载表明宾夕法尼亚大学于 1944 年主持建造了 ENIAC(电子数字积分器和计算器)机器。这是第一台用真空管建造的、实际运行的通用机。在第二次世界大战中,这台机器主要用来帮助计算火炮轨迹表。借助手动的开关设置和电缆的插接可对该机器进行编程。按照今天的标准这台机器是很慢的,它只有很有限的存储容量和原始的可编程性。该大学还提出了一个改进的 ENIAC 版本,称为 EDVAC(电子离散变量自动计算机),该机器试图改进程序的输入方法,启发了开发存储程序式计算机的探索。直到 1952 年,EDVAC 的研究项目才完成。受 ENIAC 实现思想的启发,普林斯顿高级研究所(IAS)的研究者们建造了 IAS 机器(1946 年),它的速度比 ENIAC 快了约 10 倍。

1946 年,当 EDVAC 研究项目尚在进行中时,剑桥大学启动了一个类似的研究项目。该项目试图建造一台存储程序计算机,称为电子延迟存储自动计算器(EDSAC)。1949 年,EDSAC 成为了世界上第一台完整的、存储程序的全运行计算机。由 EDSAC 带来的副产品是在哈佛建造了一系列的机器。该系列机包括 MARK I,II,III 和 IV。后两台机器引入了分离的指令存储器和数据存储器。后人对这些机器赋予了“哈佛体系结构”(Harvard Architecture)的概念,以表明使用了分离的存储器。注意,“哈佛体系结构”今天则用来描述具有分离的指令高速缓存和数据高速缓存。

1951 年,第一台通用的商用计算机——UNIVAC(通用自动计算机)I 面市了,它是对 1949 年建造的 BINAC(二进制自动计算机)的改进。IBM 公司在 1952 年发布了它的第一台计算机 IBM 701。但在 20 世纪 50 年代早期,计算机工业发展遭遇减速。此后,在 1964 年,IBM 公发布了 IBM 360 系列机。该系列机包括许多价格和性能不同的型号。该系列机的出现导致了 DEC 推出第一台小型计算机 PDP-8,它当时被认为是价格相当低廉的机器。而 Intel 公司则于 1971 年推出了第一台微处理器 Intel 4004。当苹果计算机系列于 1977 年首次推出时,世界上第一台个人计算机诞生了。1977 年,DEC 公司又推出了 VAX-11/780。此后,Intel 又推出了 80x86 微处理器系列。

1977 年,由 Altair, Processor Technology, North Star, Tandy, Commodore, Apple 以及许多其他公司推出的个人计算机(PC),在许多行业都提高了终端用户的生产率。由 Compaq, Apple, IBM, Dell 以及许多其他公司生产的个人计算机很快得到了普及,并改变了计算的面貌。

与小型机同时出现的是超级计算机。由 Control Data 公司于 1961 年推出的 CDC 6600 是第一台这类超级计算机,而 Cray 研究公司则在 1976 年推出了具有最好性价比的超级计算机 Cray-1。

20 世纪 80 年代和 90 年代,业界推出了许多具有多处理器的商用并行计算机。它们一般可分为两大类:(1)共享存储器系统;(2)分布式存储器系统。在单机中所包含的处理器数从在共享存储器计算机中的几个到在大规模并行系统中的几百至几千个。这一时代的并行机例子包括 Sequent Symmetry, Intel iPSC, nCUBE, Intel Paragon, Thinking Machines(CM-2, CM-5), MsPar(MP), Fujitsu(VPP500),以及其他机器。

计算发展的一个明显趋势是集中式的服务器将为计算机网络所替代。这些由网络连接的、廉价但功能强大的台式机能形成极强的计算能力。约在 1990 年,功能强大的个人计算机和工作站局域网(LAN)开始替代大型机和小型机。这些单独的台式机很快就用广域网(WAN)连接成更大的计算联合体。

因特网的普及使网络计算以及近期的网格计算变得更加引人入胜。网格是一种地域分布式计算平台,它们将能提供可信赖、一致而且廉价的对高端计算设施的访问。

表 1.1 是 Lawrence Tesler(1995)所提出的表的修改版本。该表列出了自 20 世纪 60 年代开始与每个计算年代相关的不同计算模式的主要特征。

表 1.1　计算的 4 个年代

特征	批处理	分时	台式机	网络
年代	20 世纪 60 年代	20 世纪 70 年代	20 世纪 80 年代	20 世纪 90 年代
场所	计算机房	终端室	桌面	移动
用户	专家	专业人员	个人	小组
数据	字母数字	文本,数字	字体,图形	多媒体
对象	计算	访问	显示	通信
界面	穿孔卡片	键盘和 CRT 显示器	观看和指点	询问和告知
操作	处理	编辑	布局	协调
连接性	无	外设电缆	局域网	因特网
拥有者	公司计算中心	分区的 IS 商店	部门终端用户	每个人

CRT,阴极射线管; LAN,局域网。

1.2　体系结构的发展和风格

计算机体系结构设计师总是尽力增强他们所设计的体系结构的性能。为达到这一目的他们已采用了许多方法,其中的一个设计思想是在一条指令中做尽可能多的工作,这样就可用较少的指令数完成相同的作业。这一思想的直接结果是只需较少的存储器读/写操作,而获得最终的运算加速。这一设计思想的另一个优点是,增加指令的复杂性和寻址方式具有理论上的优点,那就是可以减小高级语言指令和低级(机器)语言指令间的"语义差别"。用一条(机器)指令将几个二进制编码的十进制数(BCD)转换成二进制数是一个很好的例子,它可以说明某些指令要完成的动作有多复杂。众多数量的寻址方式(在 VAX 机器中多达 20 多种)进一步增加了指令的复杂性。追随这一设计思想的机器称为复杂指令集计算机(CISC)。CISC 机器的例子包括 Intel Pentium,Motorola MC 68000 以及 IBM & Macintosh Power PC。

注意,将更多的能力加到处理器会增加支持更高时钟速率的难度,而这本来是能够做到的。难度增加是因为增加了在单个时钟周期内的计算复杂性。从 20 世纪 70 年代中期到 20 世纪 80 年代早期,许多研究也指明了在典型的程序中,所执行的指令 80%以上是那些使用赋值语句、条件转移和过程调用的指令。此外,简单赋值语句占据了这些操作的几乎 50%。这些发现导致了另一种不同设计思想的出现。在减少指令复杂性和寻址方式的同时,该设计思想通过加速那些最常用的操作来促进体系结构的优化。采用这一设计思想的机器称为精简指令集计算机(RISC)。RISC 机器的例子包括 Sun SPARC 和 MIPS 机器。

以上体系结构设计的两种设计思想已导致了无法调和的争论,即哪一种体系结构风格为"最好"。但是,应该注意的是,研究已经表明 RISC 体系结构的确可以使程序更快地执行。大多数当代的微处理器芯片均追随 RISC 模式。本书将介绍 CISC 和 RISC 机器各自的突出特点以及相应的示例。

1.3　技术进展

计算机技术正以空前的速度演进,包括处理器和存储器的发展。的确,技术改进刺激了计算机工业的发展。在单个芯片上集成的晶体管数(一个晶体管是一个可控的开关)已从几百个增加到以百万计,这种令人难以置信的增加得益于晶体管制造工艺的进步。

集成的规模已从小规模(SSI)、中规模(MSI)、大规模(LSI),发展到了超大规模(VLSI)乃至最近的晶片规模(WSI)。表 1.2 给出了各种工艺中每个芯片上的典型器件数。

表 1.2 芯片上的器件数

集成度	工艺	典型器件数	典型功能
SSI	双极	10 ~ 20	门和触发器
MSI	双极和 MOS	50 ~ 100	加法器和计数器
LSI	双极和 MOS	100 ~ 10 000	ROM 和 RAM
VLSI	CMOS(大多数)	10 000 ~ 5 000 000	处理器
WSI	CMOS	> 5 000 000	数字信号处理和专用

应该提及的是,最小器件特征尺寸的持续减小导致了芯片上器件数量的持续增长,而这又促成了更多方面的进展。其中在 RAM 存储器中器件数的增长又有助于设计者平衡存储器的大小和速度。特征尺寸大小的改进为引入改进的设计风格提供了极好的机会。

1.4 性能衡量

在这一节中,我们将论述评估计算机性能的重要问题,并着重讨论用来评估计算机的许多性能衡量方法。让我们在一开始就明确:计算机的性能可以反映在不同方面。例如,用户衡量计算机的性能是基于执行一个给定作业(程序)所需的时间,而实验室工程师衡量其系统的性能则是基于在给定的时间内所完成的总工作量。用户将程序的执行时间作为性能的衡量标准,而实验室工程师则认为吞吐量是更为重要的性能衡量标准。评估计算机性能的度量标准有助于比较不同的设计。

性能分析应有助于回答这样的问题,即使用给定的计算机,程序能执行得多快? 为了回答这一问题,我们需要确定由一台计算机执行一个给定作业所需的时间。我们定义时钟周期时间为一个周期时钟信号(参见图 1.1)的两个连续上升(下降)沿间的时间。时钟周期能用来为单位计算计数,因为计算结果的存储是与上升(下降)时钟沿同步的。计算机执行一个作业所需的时间通常以时钟周期数来表示。

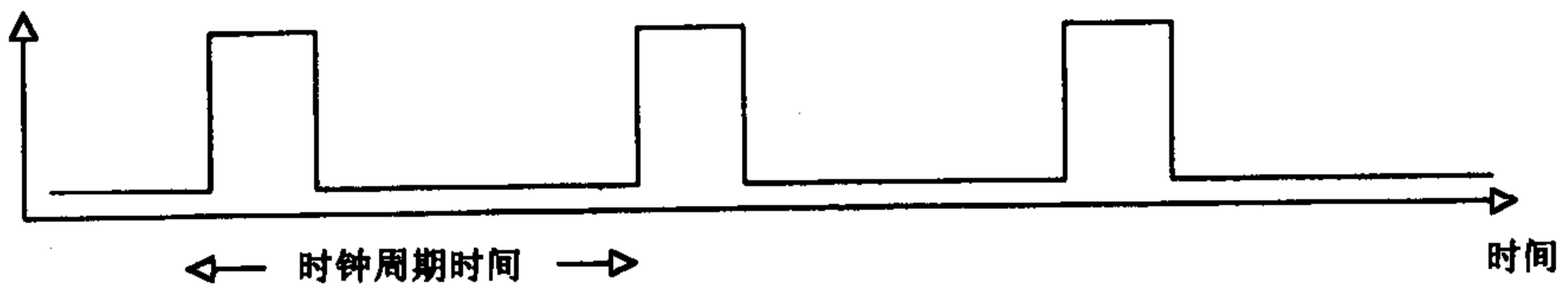


图 1.1 时钟信号

我们用 *CC*(cycle count)表示执行一个作业所需的 CPU 时钟周期数,用 *CT*(cycle time)表示周期时间,并用 $f = 1/CT$ 表示时钟频率。则由 CPU 执行一个作业所需的时间可表达为

$$\text{CPU 时间} = CC \times CT = CC/f$$

计算在一个给定的程序中所执行的指令数,比计算执行该程序所需的 CPU 时钟周期数可能更为容易。为此,每条指令所需的平均时钟周期数(*CPI*,clock cycles per instruction)已作为另一种性能衡量的指标。下面的方程说明了如何计算 *CPI*。

$$CPI = \frac{\text{程序的 CPU 时钟周期}}{\text{指令数}}$$

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

$$= \frac{\text{指令数} \times CPI}{\text{时钟速率}}$$

已知一台给定机器的指令集是由 ALU(简单的赋值、算术及逻辑指令)、load(装载)、store(存储)和 branch(转移)等指令类型所组成。则在已知每一种指令 *CPI* 的情况下,总 *CPI* 就可计算如下:

$$CPI = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{\text{指令数}}$$

其中,I_i 是程序中执行指令类型 i 的次数,而 CPI_i 则表示执行这类指令所需的平均时钟周期数。

例 1.1　根据执行一组基准测试程序所记录的以下性能测量值(如表 1.3 所示),计算机器 A 的总 *CPI*。假定 CPU 的时钟速率为 200 MHz。

表 1.3　例 1.1 表

指令类型	出现的百分比	每条指令的周期数
ALU	38	1
Load 和 store	15	3
Branch	42	4
其他	5	5

假定执行 100 条指令,则总 CPI 可计算如下:

$$CPI_A = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{\text{指令数}} = \frac{38 \times 1 + 15 \times 3 + 42 \times 4 + 5 \times 5}{100} = 2.76$$

注意,*CPI* 反映的是处理器的组成和指令集的体系结构,而指令数反映的是指令集的体系结构和所使用的编译技术。这两个参数是相辅相成的,所以在评估一台给定的计算机或比较两台机器的性能时,必须同时考虑 *CPI* 和指令数两者。

近年来受到较多关注的另一个性能衡量指标是 *MIPS*(每秒百万指令,即单位时间的指令执行速率),它定义为

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

例 1.2　假设在另一台机器 B 上执行前面所提及的相同基准测试程序组,所记录的测量值如表 1.4 所示。

表 1.4　例 1.2 表

指令类型	出现的百分比	每条指令的周期数
ALU	35	1
Load 和 store	30	2
Branch	15	3
其他	20	5

现假定时钟速率为 200 MHz,则前一例中机器 A 和本例中机器 B 的 *MIPS* 各为多少?

$$CPI_A = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{\text{指令数}} = \frac{38 \times 1 + 15 \times 3 + 42 \times 4 + 5 \times 5}{100} = 2.76$$

$$MIPS_A = \frac{\text{时钟速率}}{CPI_A \times 10^6} = \frac{200 \times 10^6}{2.76 \times 10^6} = 70.24$$

$$CPI_B = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{\text{指令数}} = \frac{35 \times 1 + 30 \times 2 + 20 \times 5 + 15 \times 3}{100} = 2.4$$

$$MIPS_B = \frac{\text{时钟速率}}{CPI_A \times 10^6} = \frac{200 \times 10^6}{2.4 \times 10^6} = 83.67$$

因此，$MIPS_B > MIPS_A$。

有趣的是，虽然 *MIPS* 已用来衡量机器的性能，但用它比较有不同指令集的机器时必须非常小心，这是因为 *MIPS* 并不涉及执行时间。例如，考虑在两台不同机器上运行一组给定的基准测试程序所得到的测量结果，如表 1.5 所示。

表 1.5 在两台不同机器上运行一组给定基准测试程序所得的测量结果

指令类型	指令数(百万)	每条指令的周期数
机器(A)		
ALU	8	1
Load 和 store	4	3
Branch	2	4
其他	4	3
机器(B)		
ALU	10	1
Load 和 store	8	2
Branch	2	4
其他	4	3

$$CPI_A = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{\text{指令数}} = \frac{(8 \times 1 + 4 \times 3 + 4 \times 3 + 2 \times 4) \times 10^6}{(8 + 4 + 4 + 2) \times 10^6} \cong 2.2$$

$$MIPS_A = \frac{\text{时钟速率}}{CPI_A \times 10^6} = \frac{200 \times 10^6}{2.2 \times 10^6} \cong 90.9$$

$$CPU_A = \frac{\text{指令数} \times CPI_A}{\text{时钟速率}} = \frac{18 \times 10^6 \times 2.2}{200 \times 10^6} = 0.198\,\text{s}$$

$$CPI_B = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{\text{指令数}} = \frac{(10 \times 1 + 8 \times 2 + 4 \times 4 + 2 \times 4) \times 10^6}{(10 + 8 + 4 + 2) \times 10^6} = 2.1$$

$$MIPS_B = \frac{\text{时钟速率}}{CPI_A \times 10^6} = \frac{200 \times 10^6}{2.1 \times 10^6} = 95.2$$

$$CPU_B = \frac{\text{指令数} \times CPI_A}{\text{时钟速率}} = \frac{20 \times 10^6 \times 2.1}{200 \times 10^6} = 0.21\,\text{s}$$

$MIPS_B > MIPS_A$ 且 $CPU_B > CPU_A$

该例子说明虽然 B 比 A 有更高的 *MIPS*，但却需要更长的 CPU 时间来执行相同的基准测试程序集。

每秒百万次浮点运算(*MFLOPS*,单位时间浮点指令执行速率)也已被用来衡量机器的性能。它定义为

$$MFLOPS = \frac{\text{程序中的浮点操作数}}{\text{执行时间} \times 10^6}$$

与 *MIPS* 测量平均指令速率不同,*MFLOPS* 只对浮点指令的子集加以定义。反对 *MFLOPS* 的一个论点基于如下事实,即浮点操作集对不同机器可能是不一致的,所以确切的浮点操作将随机器不同而不同。另一个反对论点则基于这样的事实,即对于给定的程序,由 *MFLOPS* 衡量所得到的性能无法通用化,只能为机器提供一个单一的性能指标。

大多数读者可能对只涉及一个具体程序的机器性能不感兴趣。综合有关更大程序集(如基准测试程序集)性能的最流行方法是使用算术和几何平均。它们的定义如下:

$$\text{算术平均} = \frac{1}{n}\sum_{i=1}^{n} \text{执行时间}$$

$$\text{几何平均} = \sqrt[n]{\prod_{i=1}^{n} \text{执行时间}}$$

其中,执行时间 i 是第 i 个程序的执行时间,而 n 是基准测试程序集中的总程序数。

表 1.6 给出了计算这些指标的一个例子。

表 1.6　CPU 时间计算示例

项	计算机 A 的 CPU 时间(s)	计算机 B 的 CPU 时间(s)
程序 1	50	10
程序 2	500	100
程序 3	5000	1000
算术平均	1835	370
几何平均	500	100

我们以对著名的 Amdahl(阿姆达尔)定律进行讨论来结束本节的内容,该定律认为是由于功能增强而导致加速($SU_\circ$)。在这种情况下,我们将加速作为一个指标,以衡量一台机器在实现某种功能增强后的性能相对于其原始性能的比值。下面的关系公式化了阿姆达尔定律:

$$SU_\circ = \frac{\text{增强后的性能}}{\text{增强前的性能}}$$

$$\text{加速} = \frac{\text{增强前的执行时间}}{\text{增强后的执行时间}}$$

例如,考虑对机器功能的一种可能增强,它将使某些基准测试程序的执行时间从 25 s 减少到 15 s。我们就说由于执行时间的减少导致的加速为 $SU_\circ = 25/15 = 1.67$。

在所给定的形式中,阿姆达尔定律所说明的情况是,功能的改进可作用到指令执行时间。但是,有时可能获得的性能增强仅出现在部分时间 Δ 中。在这种情况下需要推导一个新公式,以使由于部分时间 Δ 增强而获得的加速 SU_Δ 与由于整个时间的增强而获得的加速 $SU_\circ$ 相关。此关系式可表示为

$$SU_\circ = \frac{1}{(1-\Delta) + (\Delta/SU_\Delta)}$$

注意,当 $\Delta = 1$ 时,即增强在所有时间都可能时,则 $SU_\circ = SU_\Delta$。

例如,考虑一台机器在实施增强措施后加速达到 30。若在某些条件下,该增强仅在 30% 的时间中有效,则由于这种部分增强可获得的加速为多少?

$$SU_o = \frac{1}{(1-\Delta)+(\Delta/SU_\Delta)} = \frac{1}{(1-0.3)+\frac{0.3}{30}} = \frac{1}{0.7+0.01} = 1.4$$

注意,上面的公式可以通用化为如下所示的情况,即许多不同的独立增强可以分别以不同的部分时间 $\Delta_1, \Delta_2, \cdots, \Delta_n$ 实施,从而导致各自的加速 $SU_{\Delta_1}, SU_{\Delta_2}, \cdots, SU_{\Delta_n}$。

$$SU_o = \frac{1}{[1-(\Delta_1+\Delta_2+\cdots+\Delta_n)]+\frac{(\Delta_1+\Delta_2+\cdots+\Delta_n)}{(SU_{\Delta_1}+SU_{\Delta_2}+\cdots+SU_{\Delta_n})}}$$

1.5 小结

在本章中,我们简要介绍了计算机系统发展的历史背景,即从 1938 年有记录的第一台尝试建造的计算机 Z1 开始,到 CDC 6600 和 Cray 超级计算机,并以今天的现代高性能机器结束。然后,我们讨论了 RISC 和 CISC 体系结构风格以及它们对机器性能的影响。接着是对技术发展及其对计算性能影响的简短讨论。本章的论述以对有关评估计算机性能问题的详细分析结束。特别是我们介绍了许多性能衡量标准,如 *CPI*, *MIPS*, *MFLOPS* 和算术/几何性能平均,它们中没有一个能对一台机器的性能给以一致的定义。有一些方法可以衡量一台机器的给定部分改进或整体改进而导致的加速,本章末尾我们对这些方法进行了讨论。

习题

1. 从下面的一些观点叙述计算的发展趋向:
   (a) 硬件成本;
   (b) 存储器大小;
   (c) 硬件速度;
   (d) 处理单元数目;
   (e) 系统部件的地理位置。
2. 叙述近 20 年来的计算发展趋向,并描述你对未来计算的预测。
3. 给出以下术语的含义:
   (a) 机群计算;
   (b) 网格计算;
   (c) 量子计算;
   (d) 纳米技术。
4. 假定如晶体管那样的一个开关元件的开关时间为 0,我们假设用这种开关元件来构造一个圆盘形的计算机芯片,惟一的限制是将电子信号从芯片一边发送到另一边所需的时间。为简单起见,假设电子信号以 300 000 km/s 的速度传送。假设芯片的时钟速率为 1 GHz。若要使任何计算结果可在芯片上任何一处加以使用,则应限制此圆形芯片的直径为多少? 若芯片的时钟速率为 1 THz = 10^{12} Hz,则此时圆形芯片的直径应限制为多少? 制造这样的芯片可行吗?

5. 从以下各个方面比较单处理器系统和多处理机系统：
   (a) 编程是否容易；
   (b) 是否需要同步；
   (c) 性能评估；
   (d) 运行系统。
6. 考虑一个程序，它在时钟速率为 500 MHz 的计算机 A 上运行需 50 s。若另一台机器 B 运行相同程序所需的时钟周期数是机器 A 的 2.5 倍，若想在机器 B 上以 20 s 的时间运行相同的程序，则机器 B 的时钟速率必须为多少 MHz?
7. 假定我们有两种实现相同指令集体系结构的方案。机器 A 的时钟周期时间为 50 ns，且针对某一程序的 *CPI* 为 4.0，而机器 B 的时钟周期时间为 65 ns，而针对同一程序的 *CPI* 为 2.5。问哪一台机器更快？快多少？
8. 一个编译程序设计者正试图为一台特定的机器判断采用两个代码序列中的哪一个。硬件设计者已提供了如下信息：

指令类	指令类的 *CPI*
A	1
B	3
C	4

对于一种特定的高级语言，编译程序的编写者要考虑需要以下指令数的两个代码序列：

代码序列	指令数(百万)		
	A	B	C
1	2	1	2
2	4	3	1

每个代码序列的 *CPI* 为多少？哪一个代码序列更快？快多少？

9. 一台机器有 3 类指令，所测得的 *CPI* 如下：

指令类	指令类的 *CPI*
A	2
B	5
C	7

假定我们对一个给定程序的代码用两个编译程序加以测量并获得了以下数据：

代码序列	指令数(百万)		
	A	B	C
编译程序 1	15	5	3
编译程序 2	25	2	2

假定机器的时钟速率为 500 MHz。就 *MIPS* 而言，哪一个代码序列将执行得更快？若按执行时间，则哪一个代码序列将执行得更快？

10. 一台新机器有以下 3 种增强加速的方案：加速(a) = 30，加速(b) = 2，加速(c) = 15。假定对某一程序集，增强(a)使用的百分比为 25%，增强(b)使用的百分比为 30%，而增强(c)

使用的百分比为45%。若只能实现一种增强方案,则为获得最大的加速应选择哪一种方案?若可以实现两种增强方案,则应该选择哪两种来获得最大的加速?

参考文献和推荐读物

J.-L. Baer, Computer architecture, *IEEE Comput.*, 17(10), 77 – 87, (1984).

S. Dasgupta, *Computer Architecture: A Modern Synthesis*, John Wiley, New York, 1989.

M. Flynn, *Some computer organization and their effectiveness*, *IEEE Trans Comput.*, C-21, 948-960 (1972).

D. Gajski, V. Milutinovic, H. Siegel, and B. Furht, *Computer Architecture: A Tutorial*, Computer Society Press, Los Alamitos, Calif, 1987.

W. Giloi, Towards a taxonomy of computer architecture based on the machine data type view, Proceedings of the 10th International Symposium on Computer Architecture, 6-15, (1983).

W. Handler, The impact of classification schemes on computer architecture, Proceedings of the 1977 International Conference on Parallel Processing, 7 – 15, (1977).

J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, 1996.

K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, 2nd ed., McGraw-Hill, New York, 1996.

D. J. Kuck, *The Structure of Computers and Computations*, John Wiley, New York, 1978.

G. J. Myers, *Advances in Computer Architecture*, John Wiley, New York, 1982.

L. Tesler, Networked computing in the 1990s, reprinted from the Sept. 1991 Scientific American, The Computer in the 21st Century, 10 – 21, (1995).

P. Treleaven, Control-driven data-driven and demand-driven computer architecture (abstract), *Parallel Comput.*, 2, (1985).

P. Treleaven, D. Brownbridge, and R. Hopkins, Data drive and demand driven computer architecture, *ACM Comput. Surv.*, 14(1), 95 – 143, (1982).

参考网站

http://www.gigaflop.demon.co.uk/ ~ wasel.

第2章　指令集体系结构和设计

在这一章中，我们将考虑指令集体系结构及其基本设计原理。我们将从对存储器单元和地址的讨论开始。我们将提出一个主存储器的抽象模型，在该模型中主存储器将被视为是一个单元序列，每个单元能存储 n 位。然后我们将讨论存储器的存入和读取问题。将信息存入存储器和从存储器读取都需要选择地址。因此本章中讨论的下一个主题便是众多不同的选择存储单元的方法(寻址方法)。一个程序由许多按一定顺序访问的指令所组成，这就促使我们要较为详细地解释指令执行和排序的问题。接下来，我们将展示所介绍的寻址方式和指令特征在编写能完成简单程序设计任务的示例代码段中的应用。

计算机存储器的一个独特特征是它应该以层次方式构成。在这种层次中，较大和较慢的存储器将用来补充较小和较快的存储器。一个典型的存储器层次结构从一个容量小、价格贵但速度相对较快的称为高速缓存(cache)的模块开始。在该高速缓存下面的层次是较大、稍便宜和相对较慢的主存储器部分。高速缓存和主存储器是用半导体材料构成的。在它们之后的存储层次是更大、较便宜但却慢得多的由(硬)磁盘和磁带所组成的磁存储器。第6章和第7章将详细讨论影响成功构成计算机存储层次结构的因素。本章将专注于由程序员所观察到的(主)存储器，特别会侧重讨论信息存入存储器和从存储器读取的方法。

2.1　存储单元和操作

(主)存储器可被模型化为数百万个相邻单元的一个阵列，每个单元能存储一个二进制数字(位)，其值为1或0。这些单元以固定单元数(如 n 个)组的形式构成，它将被视为一个原子实体。由8位(bit，b)所组成的实体称为一个字节(byte，B)。在许多系统中，由 n 位组成的、能用一个基本存储器操作向存储器存入和取出的实体称为一个字(最小可寻址实体)。一个字的典型大小是16位到64位。但是我们习惯用字节来表示存储器的大小。例如，通常一台个人计算机存储器的大小为256兆字节(256 Mbyte，256 MB)，即 $256 \times 2^{20} = 2^{28}$ B。

为了能将一个字移入和移出存储器，必须为每个字分配一个不同的地址，用该地址来确定要被存储的给定字在存储器中的位置，这称为存储器的写操作。类似地，地址也用来确定从存储器中读取一个字的位置，这称为读操作。

在一个存储器中为区别 M 个字所需的不同地址位 l 由 $l = \text{lb}\ M$ 给定。例如，若存储器的大小为64 M(读做64兆字)，则地址位数应是 $\text{lb}(64 \times 2^{20}) = \text{lb}(2^{26}) = 26$ 位。换言之，若地址位数为 l，则最大的存储器规模(按照使用 l 位能寻址的字数确定)为 $M = 2^l$。图2.1说明了上面所解释的存储器字和字地址的概念。

如前所述，有两个基本的存储器操作，它们是存储器写和存储器读操作。在一个存储器写操作中，一个字将被存放到指定地址的存储单元中。在一个存储器读操作中，一个字将从指定地址的存储单元中读出。通常，存储器读和写操作是由中央处理器(CPU)来完成的。

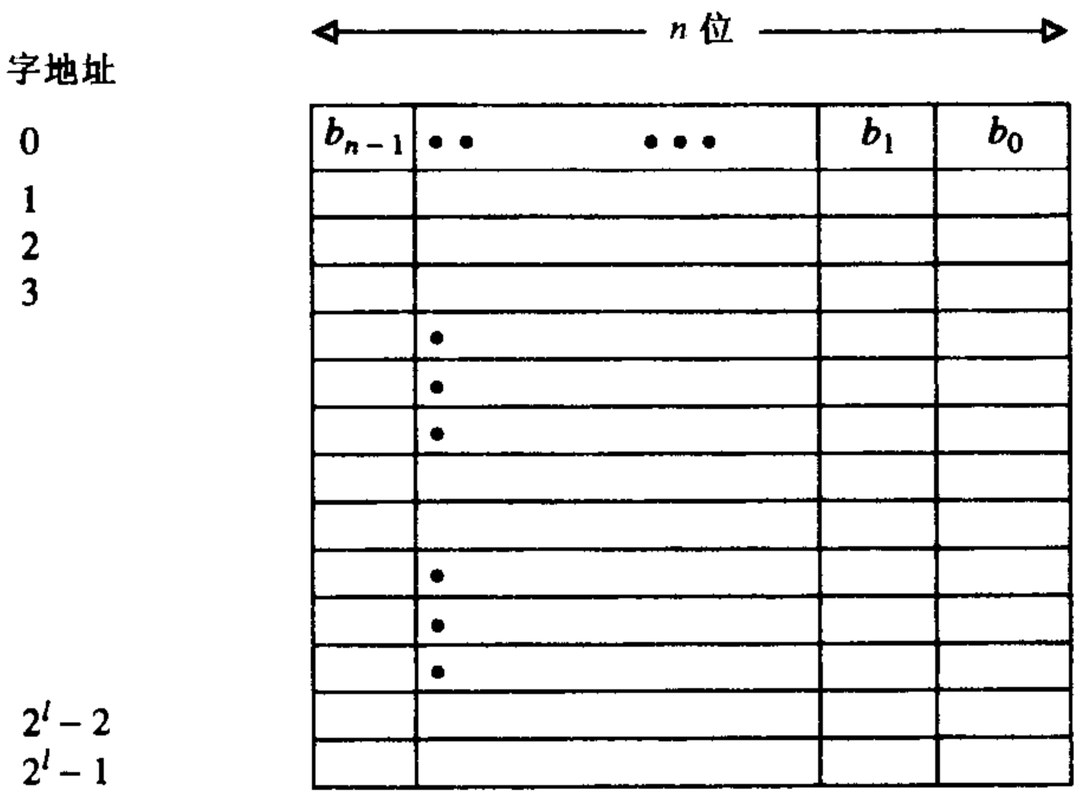


图 2.1　主存储器寻址的说明

要使 CPU 向一个指定的存储单元完成写操作需要经历 3 个步骤：

1. 首先由 CPU 将要存放到存储单元的字装载到一个指定的存储器数据寄存器(MDR)中；
2. 由 CPU 将要存放字的存储单元地址装载到一个指定的存储器地址寄存器(MAR)中；
3. 由 CPU 发出一个称为 write(写)的信号，表明存储在 MDR 中的字将被存放到已装载到 MAR 的存放地址所指定的存储单元中。

图 2.2 说明了将字 7E(十六进制)写入到地址为 2005 的存储单元的写操作。如图 2.2(a)所示为在执行写操作之前与写操作有关的寄存器和存储单元的状态。如图 2.2 (b)所示为在执行写操作之后的状态。

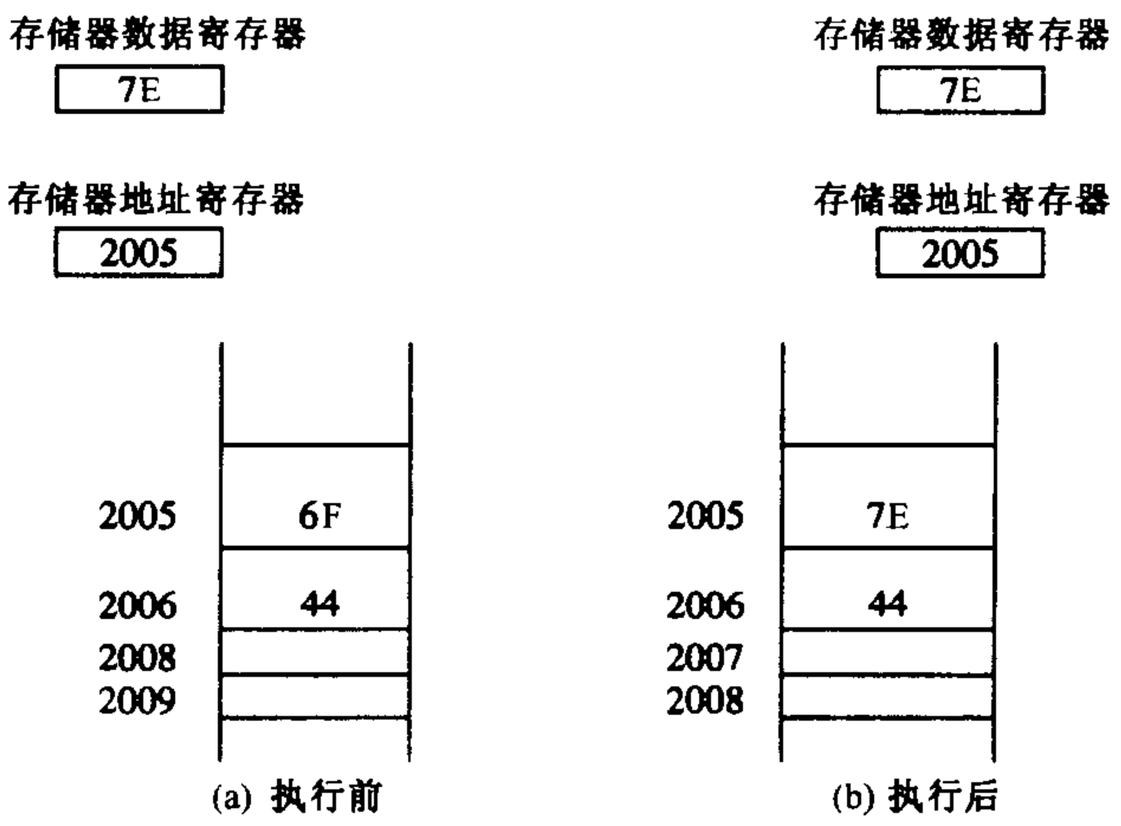


图 2.2　存储器写操作的说明

值得提及的是，MDR 和 MAR 是 CPU 专用的寄存器，程序员是不可访问的。

类似于写操作，为完成存储器读操作也需要 3 个基本步骤：

1. 将要读出的存储单元的地址装载到 MAR 中；
2. 由 CPU 发出一个称为 read(读)的信号，表明存储在 MAR 中的地址所指定存储单元包含的字将被读到 MDR 中；

3. 经过读指定字的存储器的一段延迟时间后，所需的字将由存储器装载到 MDR 中，以供 CPU 使用。

图 2.3 说明了读出存储在地址为 2010 的存储单元中的字的操作。图 2.3 (a)所示为在执行读操作之前与读操作有关的寄存器和存储单元的状态。图 2.3(b)所示为在执行读操作之后的状态。

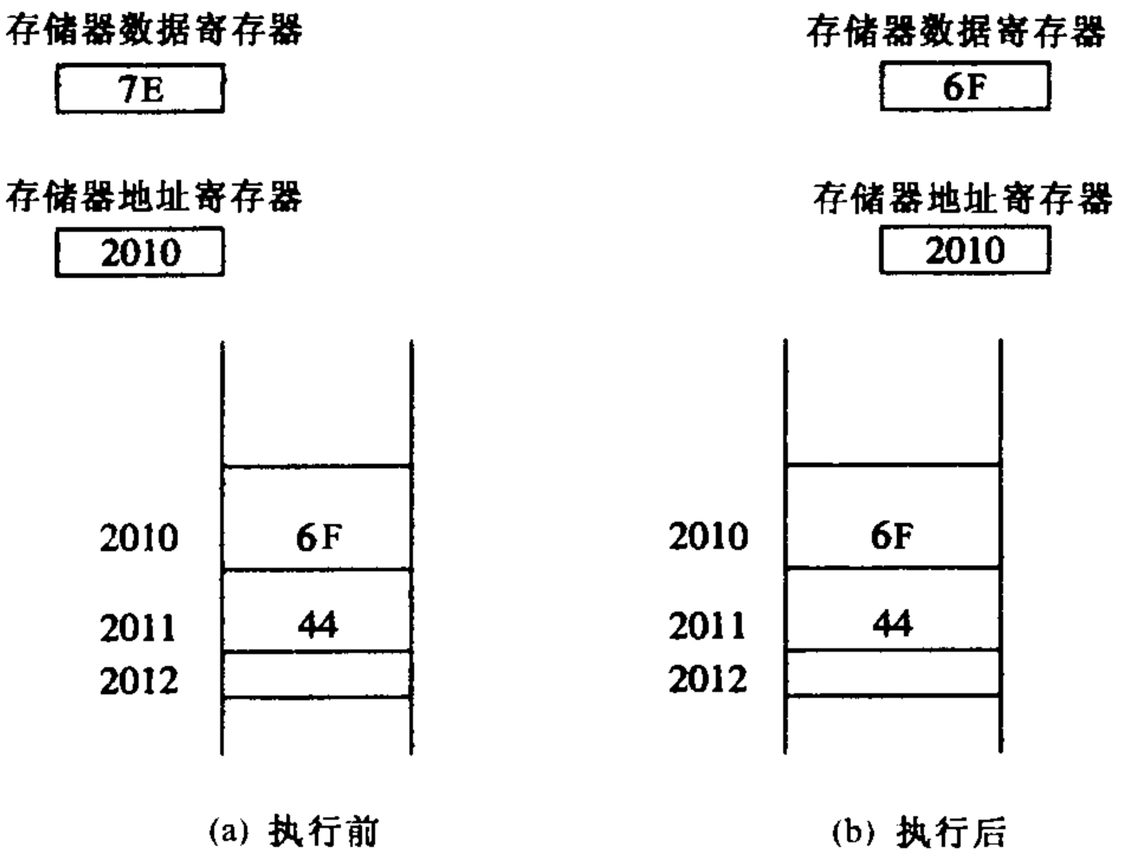


图 2.3　存储器读操作的说明

2.2　寻址方式

由 CPU 完成的任何操作中的信息都需要寻址，计算机术语中称这种信息为操作数(operand)。所以由处理器发动的任何指令必须至少具有两种类型的信息，这就是要完成的操作，编码后称为操作码(op-code)，以及要进行操作的操作数的地址信息，编码后称为地址(address)字段。

根据操作数的数目，可将指令分为：三地址、二地址、一个半地址、单地址以及零地址。在下面几个段落中，我们将解释这些寻址类型以及它们的简单例子。应该注意的是在介绍这些例子时我们将使用操作、源、目的(operation，source，destination)的约定来表示任何指令。在这个约定中，operation 代表要完成的操作，如加、减、写或读。Source 字段代表源操作数，它可以是一个常数、寄存器中的一个值、或存储器中的一个值。Destination 字段表示存储操作结果的地方，例如一个寄存器或一个存储单元。

三地址的指令用如下形式表示：operation add-1，add-2，add-3。其中 add-1，add-2 和 add-3 各代表一个寄存器或一个存储单元。例如，对于指令 ADD R_1，R_2，R_3，它表明要完成的操作是加法，同时它也指明要加的数存储在寄存器 R_1 和 R_2 中，而结果存放在寄存器 R_3 中。涉及存储单元的三地址指令的例子可以用以下形式表示：ADD A，B，C。该指令把存储单元 A 的内容加到存储单元 B，并将结果存入存储单元 C。

二地址的指令用如下形式表示：operation add-1，add-2。其中 add-1 和 add-2 各代表一个寄存器或一个存储单元。例如，对于指令 ADD R_1，R_2，该指令把寄存器 R_1 加到寄存器 R_2，并将结果存放到寄存器 R_2。操作后寄存器 R_2 中原先的内容将丢失，而寄存器 R_1 中的内容保持不变。该指令等价于三地址的指令形式 ADD R_1，R_2，R_2。使用存储单元而不是寄存器的类似指令可用如下形式 ADD A，B 表示，此时将存储单元 A 的内容加到存储单元 B，其结果覆盖存储单元 B 原来的内容。

由三地址指令 ADD A,B,C 完成的操作可用两条二地址指令 MOVE B,C 和 ADD A,C 来完成。这是因为第 1 条指令将存储单元 B 的内容移到存储单元 C,而第 2 条指令将存储单元 A 的内容加到存储单元 C(其中为存储单元 B 的内容),并将结果存入存储单元 C。

单地址指令的形式为 ADD R_1。在这种形式中,指令隐含地涉及一个称为累加器(Accumulator),R_{acc}的寄存器,该指令使得累加器的内容加到寄存器 R_1,并将结果写回累加器 R_{acc}。若操作数是存储单元而不是寄存器,则使用 ADD B 的形式,该指令将累加器中的内容与存储单元 B 中的内容相加,然后把结果存到累加器中,指令 ADD R_1 等价于三地址指令 ADD R_1,R_{acc},R_{acc}或二地址指令 ADD R_1,R_{acc}。

在二地址和单地址指令之间,有一种一个半地址指令。例如,对于指令 ADD B,R_1,它将寄存器 R_1 的内容加到存储单元 B,并把结果存入寄存器 R_1。由于该指令实际上使用了两种类型的寻址,即一个寄存器寻址和一个存储单元寻址,因而它称为一个半地址指令。这是因为寄存器寻址比存储器寻址所需的位数要少。

很有趣的是存在零地址指令,这些指令使用堆栈操作(stack operation)。堆栈是一种数据构造机制,最后存入的数据项将被首先取出。在堆栈上可完成两个特定的操作,进栈(push)和退栈(pop)。图 2.4 说明了这两个操作。

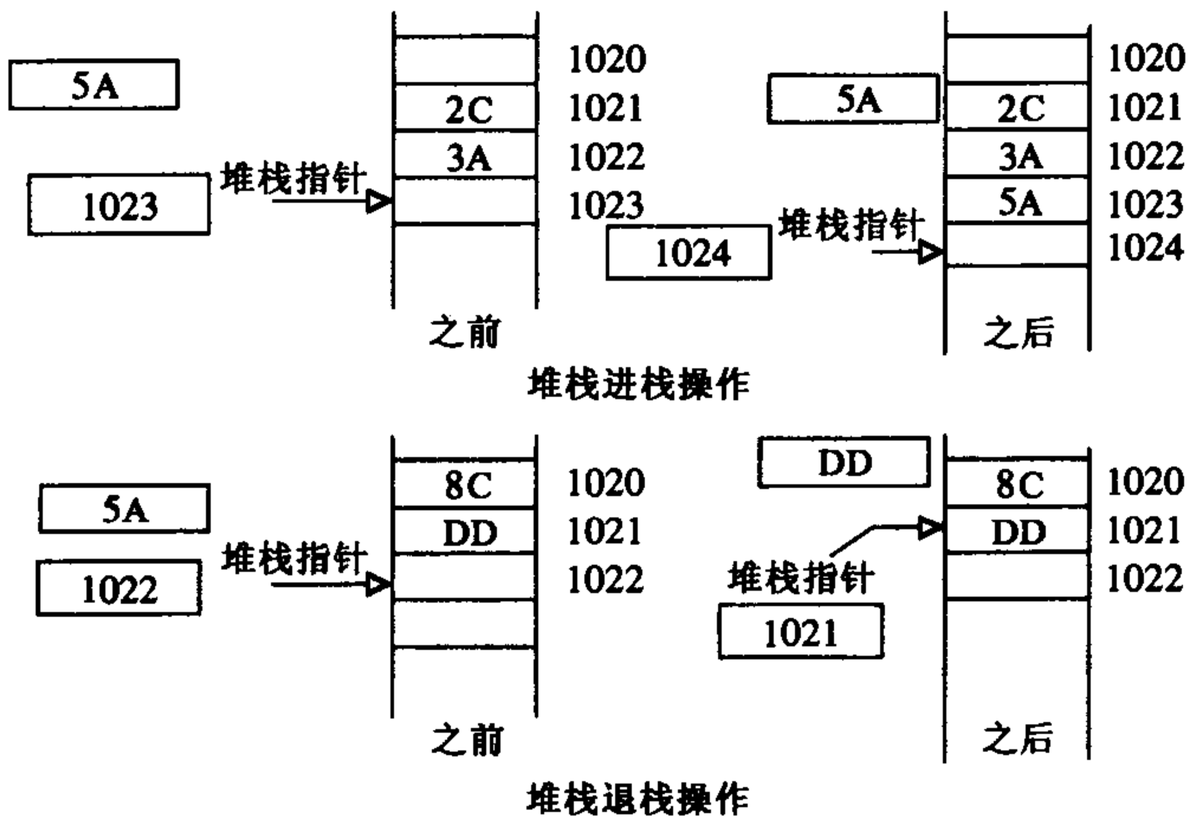


图 2.4 堆栈的进栈和退栈操作

如图所示,我们使用一个称为堆栈指针(SP,stack pointer)的特定寄存器来指示可被寻址的堆栈单元。在堆栈的进栈操作中,SP 值用来指明存放值(5A)的堆栈单元(称为栈顶,这里是单元 1023)。在存储该值(进栈)后 SP 被加 1,指向 1024 单元。在堆栈的退栈操作中,SP 值先被减 1,成为 1021。然后取出(弹出)存储在该单元的值(这里是 DD),并存放到所指示的寄存器中。

使用堆栈的结构可完成不同的操作。例如,对于指令 ADD (SP) + ,(SP),它将 SP 所指定的堆栈单元的内容加到由 SP + 1 所指定的堆栈单元,并将结果存到由当前 SP 值所指定的堆栈单元。图 2.5 说明了这个加法操作。表 2.1 对上面所讨论的指令分类做了总结。

对操作数寻址的不同方法称为寻址方式。寻址方式的不同在于对操作数地址信息说明的不同。最简单的寻址方式是将操作数本身包含在指令中,即不需要地址信息。这称为立即数寻址。一种更为复杂的寻址方式是通过加一个常数值到一个寄存器来计算操作数的地址。这称为变址寻址。在这两种寻址方式之间存在许多其他的寻址方式,包括绝对寻址、直接寻址和间接寻址。下面介绍这些不同的寻址方式。

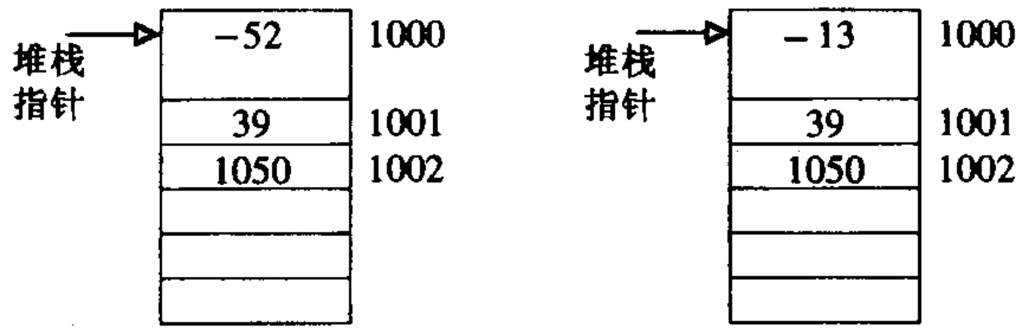


图 2.5　用堆栈进行加法

表 2.1　指令分类

指令类型	例
三地址	ADD R_1, R_2, R_3
	ADD A, B, C
二地址	ADD R_1, R_2
	ADD A, B
一个半地址	ADD B, R_1
单地址	ADD R_1
零地址	ADD(SP) + , (SP)

2.2.1　立即数方式

按照这种寻址方式、操作数的值在指令本身中就可(立即)使用。例如考虑装载十进制数值 1000 到寄存器 R_i 的情况。该操作可用如下指令加以完成:LOAD # 1000, R_i。在该指令中,要完成的操作是装载一个值到一个寄存器。源操作数(立即数)给定为 1000,而目的寄存器是 R_i。应该注意的是为了提示在指令中提及的数值 1000 是操作数本身而不是它的地址(立即方式),习惯上在操作数前加上一个专用符号(#)。显而易见,立即数寻址方式的使用是很简单的。但立即数寻址的使用将导致不尽如人意的程序设计实践。这是因为操作数值的改变需要在每一条使用该操作数的指令中改变立即数值。下面说明一种更为灵活的寻址方式。

2.2.2　直接(绝对)方式

按照这种寻址方式,指令中含有操作数所在的存储单元的地址。例如,考虑将存储在地址为 1000 的存储单元中的值装载到寄存器 R_i 的情况。该操作可用指令 LOAD 1000, R_i 完成。该指令中,源操作数为存储在地址为 1000 的存储单元中的值,R_i 是目标寄存器。注意,值 1000 前不带有任何专用字符表明这是源操作数的(直接或间接)地址。如图 2.6 所示为对直接寻址方式的说明。例如,若地址为 1000 的存储单元的内容在指令 LOAD 1000, R_i 执行时为(- 345),则该指令的执行结果是将值(- 345)装载到寄存器 R_i。

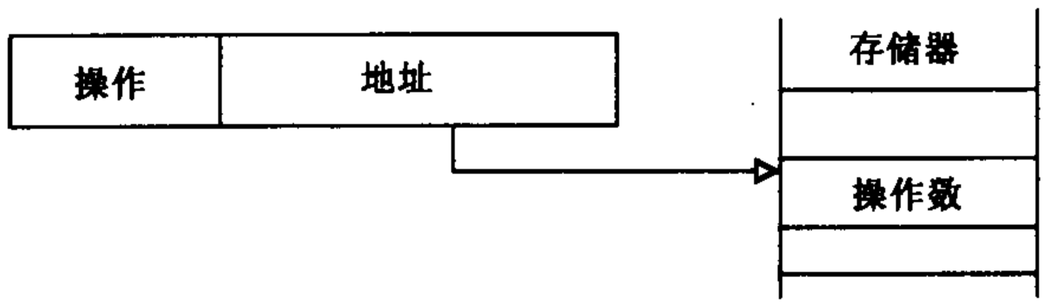


图 2.6　直接寻址方式的说明

与立即数寻址方式相比,直接(绝对)寻址方式提供了更多的灵活性。但是它需要在指令中显式包含操作数的地址。通过使用间接寻址方式可提供更灵活的寻址机制,下面将对它进行说明。

2.2.3 间接方式

在间接方式中,指令中包含的不是操作数的地址,而是含有操作数(有效)地址的寄存器或存储单元的名字。为了表明在指令中使用间接寻址,习惯上将寄存器或存储单元的名字放在括号中。例如对于指令 LOAD(1000),R_i,该指令中存储单元 1000 处于括号中,从而指明这是间接寻址。该指令的意义是将其地址存放在存储器地址 1000 中的存储单元的内容装载到寄存器 R_i。因为间接寻址可以通过一个寄存器或是一个存储单元实现,所以我们能区别两种类型的间接寻址,即寄存器间接寻址(它使用一个寄存器存放操作数的地址)和存储器间接寻址(它使用一个存储单元存放操作数的地址)。图 2.7 说明了这两种类型的间接寻址。

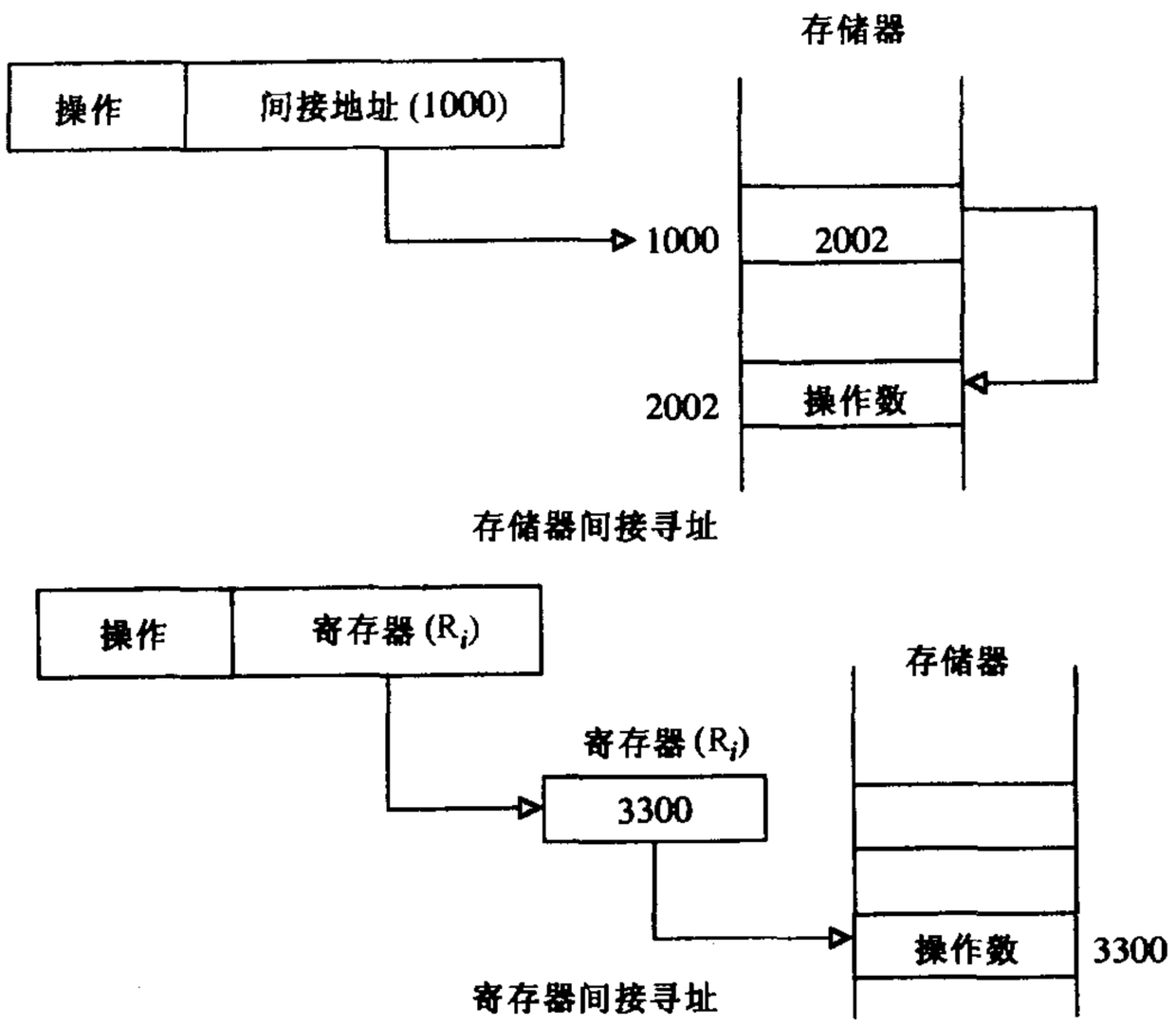


图 2.7 间接寻址方式的说明

2.2.4 变址方式

在这种寻址方式中,通过加一个常数到一个变址寄存器的内容来得到操作数的地址。例如,对于指令 LOAD X(R_{ind}),R_i。该指令将地址为寄存器 R_{ind} 的内容和值 X 之和的存储单元中的内容装载到寄存器 R_i。在指令中,变址寻址是通过下述方法加以表明的,即将变址寄存器名放在括号中并使用符号 X 指明要相加的常数。图 2.8 说明了变址寻址。可见,变址比寄存器间接寻址方式更加复杂。

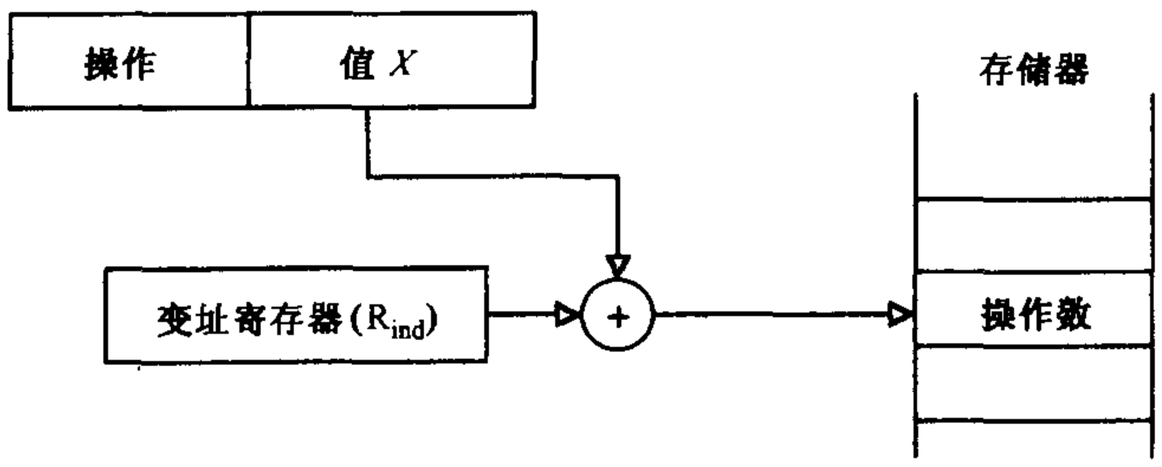


图 2.8 变址寻址方式的说明

2.2.5　其他方式

上面所介绍的寻址方式代表了在大多数处理器中最常用的方式。它们向程序员提供了足够多的方法来处理大多数一般的程序设计任务。但是还有一些其他的寻址方式已经使用在一些处理器中以方便特定的程序设计任务的执行。这些寻址方式与上面已叙述的相比要更为复杂。在这些寻址方式中相对方式、自动加 1 和自动减 1 是最为著名的。下面我们对它们进行解释。

相对方式。请回忆在变址寻址中使用的一个变址寄存器 R_{ind}。相对寻址除了用程序计数器(PC)代替变址寄存器以外与变址寻址是相同的。例如,指令 LOAD X(PC),R_i 是将地址为程序计数器(PC)的内容与值 X 之和的存储单元的内容装载到寄存器 R_i。图 2.9 说明了相对寻址方式。

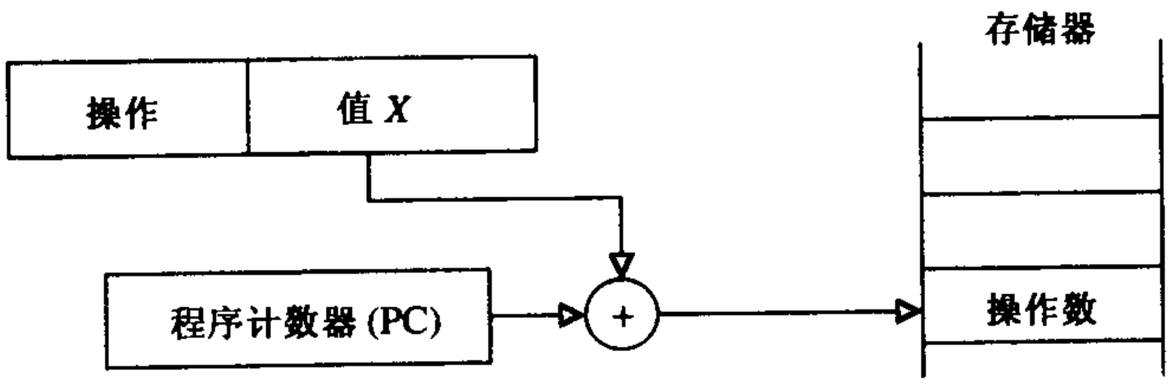


图 2.9　相对寻址方式的说明

自动加 1 方式。这一寻址方式类似于寄存器间接寻址方式,即操作数的有效地址是包含在指令中的一个称为自动加 1 寄存器的内容。但是,这里的自动加 1 是指在访问操作数之后使寄存器的内容自动加 1。如同前面一样,借助括号中的自动加 1 寄存器指明间接方式。例如,对于指令 LOAD(R_{auto}) + ,R_i,该指令将地址为寄存器 R_{auto} 内容的操作数装载到寄存器 R_i。在将操作数装载到寄存器 R_i 后,就将寄存器 R_{auto} 的内容加 1,从而指向项表中的下一项。图 2.10 说明了自动加 1 寻址方式。

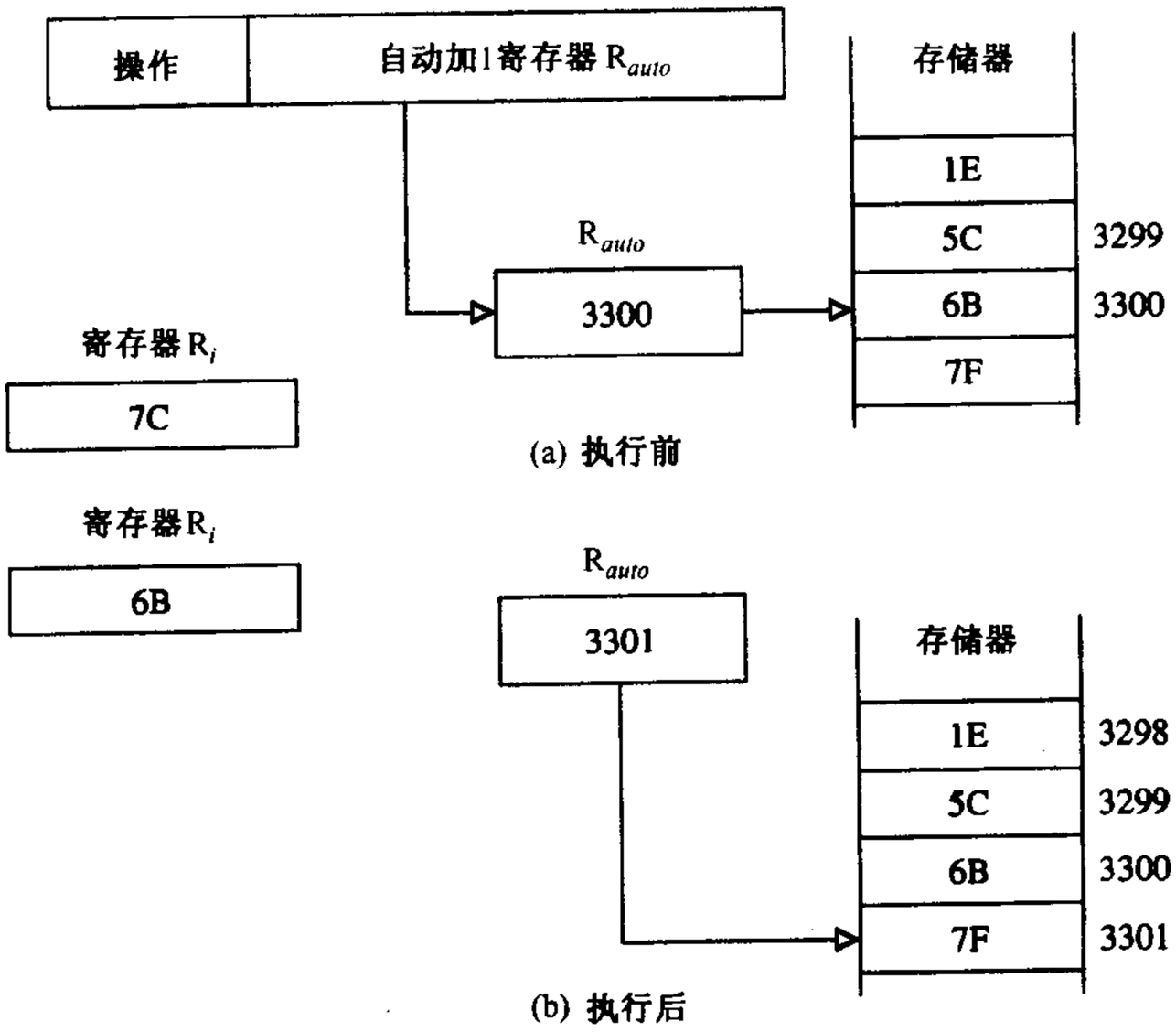
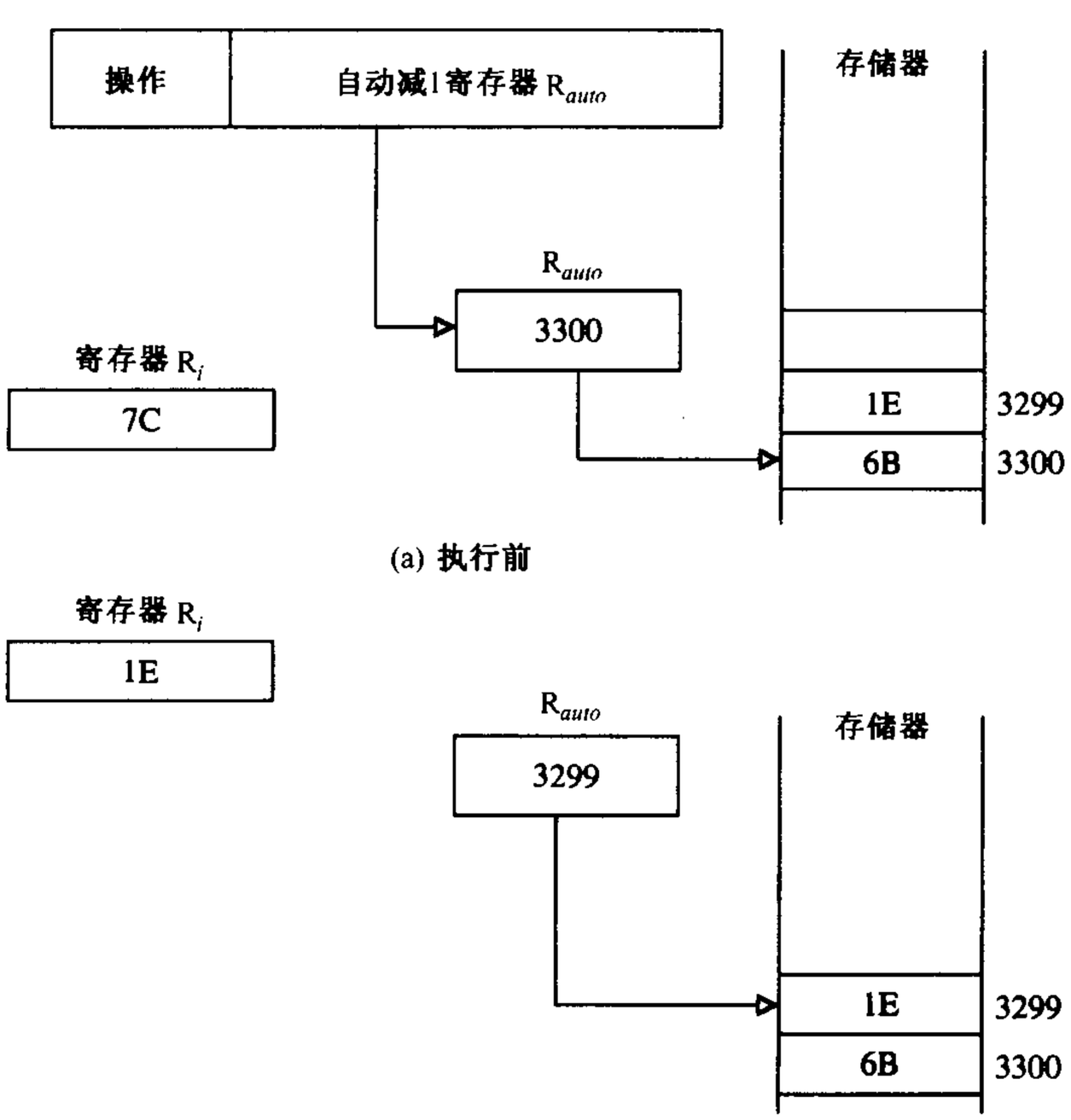


图 2.10　自动加 1 寻址方式的说明

自动减 1 方式 类似于自动加 1,自动减 1 方式使用一个寄存器来保存操作数的地址。但是,在这里自动减 1 寄存器的内容是先行减量,然后使用新的内容作为操作数的有效地址。为了反映自动减 1 寄存器内容在访问操作数之前先进行减 1 这一事实,在间接括号前放置一个(-)。例如,考虑指令 LOAD -(R_{auto}),R_i,该指令先将寄存器 R_{auto} 内容减 1,然后使用新内容作为装载到寄存器 R_i 中的操作数的有效地址。图 2.11 说明了自动减 1 的寻址方式。



(a) 执行前

(b) 执行后

图 2.11 自动减 1 寻址方式的说明

表 2.2 对上述的 7 种寻址方式做了总结。表中对每一种寻址方式给出了其名称、定义,以及说明这种方式的一个通用例子。

表 2.2 寻址方式的总结

寻址方式	定义	举例	操作
立即数	指令中含有操作数值	load #1000,R_i	R_i←1000
直接(绝对)	指令中含有操作数的地址	load 1000,R_i	R_i←M[1000]
寄存器间接	在指令所指明的寄存器中含有操作数所在存储单元的地址	load (R_j),R_i	R_i←M[R_j]
存储器间接	在指令所指明的存储单元中含有操作数所在存储单元的地址	load (1000),R_i	R_i←M[1000]
变址	操作数地址是一个变址值和一个变址寄存器内容的和	load X(R_{ind}),R_i	R_i←M[R_{ind} + X]
相对	操作数地址是一个变址值和程序计数器内容的和	load X(PC),R_i	R_i←M[PC + X]
自动加 1	操作数地址在一个寄存器中,在取操作数后其值加 1	load (R_{auto}) + ,R_i	R_i←M[R_{auto}] R_{auto}←R_{auto} + 1
自动减 1	操作数地址在一个寄存器中,其值在取操作数之前减 1	load - (R_{auto}),R_i	R_{auto}←R_{auto} - 1 R_i←M[R_{auto}]

在介绍这些不同的寻址方式时，为说明起见我们使用了 LOAD(装载)指令。但是应该认识到在给定的机器中还有其他类型的指令。在下面一节中我们将详细说明通常构成一个给定机器指令集的不同类型的指令。

2.3　指令类型

形成一台机器指令集的指令类型是对机器底层体系结构功能的反映。指令分类如 2.3.1 节到 2.3.4 节所述。

2.3.1　数据移动指令

使用数据移动指令可以将数据在机器的不同部件中进行移动。其中最让人耳熟能详的应该是在 CPU 的不同寄存器间移动数据的那些指令。简单的寄存器到寄存器的数据移动可使用如下指令

$$\text{MOVE } R_i, R_j$$

该指令移动寄存器 R_i 的内容到寄存器 R_j。该指令的结果是覆盖(目的)寄存器 R_j 的内容，但不改变(源)寄存器 R_i 的内容。数据移动指令包括那些将数据从存储器到寄存器(或相反方向)的移动指令。这些指令通常各自称为 LOAD(装载)和 STORE(存储)指令。这两种指令的例子为

$$\text{LOAD } 25838, R_j$$

$$\text{STORE } R_i, 1024$$

第一条指令将地址为 25838 的存储单元的内容装载到目的寄存器 R_j。执行 LOAD 指令时，存储单元的内容不会发生变化。STORE 指令将源寄存器 R_i 的内容存入存储单元 1024。在执行 STORE 指令时，源寄存器内容不会发生变化。表 2.3 给出了一些常见的数据传输操作以及它们的意义。

表 2.3　一些常见的数据移动操作

数据移动操作	意义
MOVE	将数据(字或块)从一个指定源(寄存器或存储器)移动到一个指定目的地
LOAD	将数据从存储器装载到寄存器
STORE	将数据从寄存器存储到存储器
PUSH	将数据从寄存器存储到堆栈
POP	从堆栈读取数据到寄存器

2.3.2　算术和逻辑指令

算术和逻辑指令用来对寄存器和存储器内容进行算术和逻辑的处理。算术指令的例子包括 ADD(加)和 SUBTRACT(减)指令。例如：

$$\text{ADD } R_1, R_2, R_0$$

$$\text{SUBTRACT } R_1, R_2, R_0$$

第一条指令将源寄存器 R_1 和 R_2 的内容相加，并将结果存入目的寄存器 R_0。第二条指令将源寄存器 R_1 和 R_2 的内容相减，并将结果存入目的寄存器 R_0。执行 ADD 和 SUBTRACT 指令不改变源

寄存器的内容。除了 ADD 和 SUBTRACT 指令外,某些机器有 MULTIPLY(乘法)和 DIVIDE (除法)指令。这两种指令实现的代价很高,但它们可以用重复的加和减来替代。大多数现代体系结构的指令集中不具有 MULTIPLY 和 DIVIDE 指令。表 2.4 给出了一些常见的算术操作和它们的意义。

表 2.4 一些常见的算术操作

算术操作	意义
ADD	对两个操作数求算求和
SUBTRACT	对两个操作数求算求差
MULTIPLY	对两个操作数求算求积
DIVIDE	两个操作数相除
INCREMENT	寄存器内容加 1
DECREMENT	寄存器内容减 1

逻辑指令用来完成逻辑操作,如 AND,OR,SHIFT,COMPARE 和 ROTATE。如它们的名字所示,这些指令对寄存器或存储器各自进行与,或,移位,比较和旋转操作。表 2.5 给出了一些逻辑操作。

表 2.5 一些常见的逻辑操作

逻辑操作	意义
AND	对两个操作数求逻辑与
OR	对两个操作数求逻辑或
EXOR	对两个操作数求异或
NOT	对操作数取补
COMPARE	对两个操作数进行逻辑比较,并设置相应的标志
SHIFT	对寄存器内容进行逻辑移位(左或右)
ROTATE	对寄存器内容进行循环逻辑移位(左或右)

2.3.3 排序指令

控制(排序)指令用来改变指令执行的顺序。它们有以下的一些形式:条件转移(条件跳转),无条件转移(跳转),以及调用指令。这些指令的一个共同特性是它们的执行将改变程序计数器(PC)的值。改变 PC 的值可以是无条件的,例如无条件转移或跳转指令。在这种情况下,PC 原来的值将丢失,程序的执行将从指令说明的新值开始,例如,对于指令 JUMP NEW-ADDRESS,该指令的执行将用 NEW-ADDRESS 表示的存储单元装载 PC,从而执行存放在该新地址中的指令。另一方面,也可基于一个指定标志的值有条件地由转移指令改变 PC 中的值。这些标志的例子包括 Negative(N,表示负),Zero(Z,表示零),Overflow(V,表示溢出)以及 Carry(C,表示进位)。这些标志表示一个称为条件码(CC)寄存器(CONDITIONAL CODE REGISTER)的指定寄存器的各个相应位。相应标志位的值根据执行不同指令的结果设置。表 2.6 给出了每个标志位的意义。

表 2.6 条件标志举例

标志名	意义
Negative(N)	若最近的操作结果为负,就设置为 I,否则为 0
Zero(Z)	若最近的操作结果为零,就设置为 1,否则为 0
Overflow(V)	若最近的操作结果发生溢出,就设置为 1,否则为 0
Carry(C)	若最近的操作导致进位,就设置为 1,否则为 0

例如,考虑下面的指令组

```
        LOAD          #100, R1
Loop:   ADD           (R2)+, R0
        DECREMENT     R1
        BRANCH-IF-GREATER-THAN Loop
```

第 4 条指令是一条条件转移指令,它表示若寄存器 R_1 内容减 1 的结果大于 0,即若 Z 标志位未被设置,则下一条要执行的指令是标号为 Loop 的指令。要注意的是,条件转移指令能用来执行程序循环(如上所示)。

CALL(调用)指令用来使程序的执行转移到一个子程序。从使用一个新值装载 PC,并由此执行下一条指令这一点来看,CALL 指令与 JUMP 有相同的效果。但是,使用 CALL 指令时,将把加 1 后的 PC 值(在指令序列中指向下一条指令)压入堆栈。在子程序中执行一条 RETURN 指令将用退栈值装载 PC。它所起的效果将是从转移到子程序的那一点继续程序的执行。

图 2.12 给出了使用 CALL 指令的一个程序段。该程序段对 N 个值求和,并将结果存入存储单元 SUM。待加的值存储在从 NUM 开始的一连串存储单元中,称为 ADDITION 的子程序用来完成实际的值相加操作,而主程序将结果存储到 SUM。

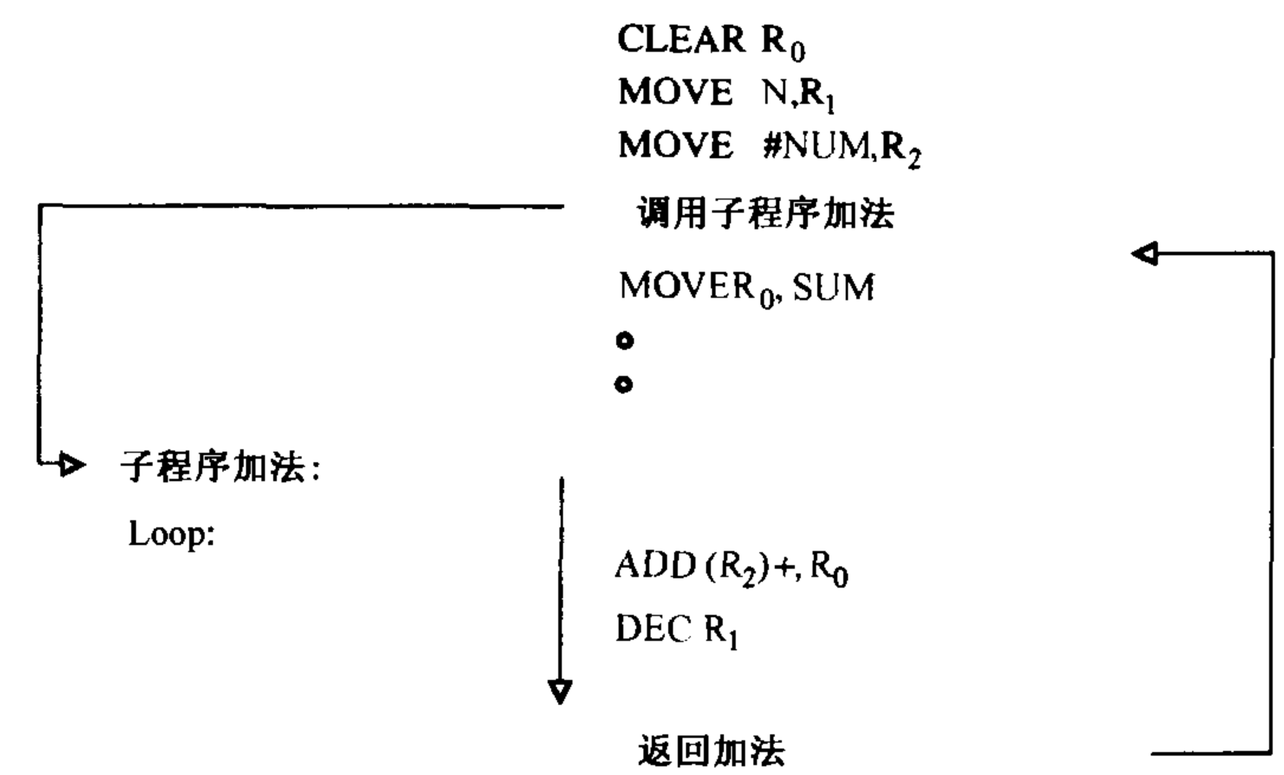


图 2.12　使用子程序的一个程序段

表 2.7 给出了一些常用的控制转移操作。

表 2.7　一些控制转移操作

控制转移操作	意义
BRANCH-IF-CONDITION	若条件为真,控制就转移到新地址
JUMP	控制无条件转移
CALL	控制转移到子程序
RETURN	控制程序转移到子程序调用者(返回)

2.3.4　输入/输出指令

输入/输出指令(I/O 指令)用来在计算机和外围部件之间传输数据。两条基本I/O 指令是 INPUT 和 OUTPUT 指令。INPUT 指令将数据从一个输入部件传送到处理器。输入部件的例子包括键盘或鼠标。输入部件通过专用的输入端口与计算机交互。计算机能使用专用的地址访问这些端口。假设一个键盘通过专用地址为 1000 的输入端口与计算机连接,因此,指令 INPUT

1000 的执行,将使存储在键盘和计算机之间接口中的指定寄存器(称为输入数据寄存器)内的数据移到计算机中另一个指定的寄存器(称为累加器)中。类似地,指令 OUTPUT 2000 的执行,将使存放在累加器中的数据移动到输出部件中地址为 2000 的数据输出寄存器中。也就是说,计算机能以通常寻址存储单元的方法访问这些端口。在这种情况下,计算机通过执行一条指令 MOVE R_{in},R_0 就能从一个输入部件输入数据。该指令将寄存器 R_{in} 的内容移到寄存器 R_0。类似地,指令 MOVE R_0,R_{in} 就将寄存器 R_0 中的内容移到寄存器 R_{in},即是完成了一个输出操作。后一种方案称为存储器映射输入/输出。存储器映射输入/输出的优点除了不需要专用的 I/O 指令外,它还能够在 I/O 部件中的寄存器上执行许多存储器专用指令。它的主要缺点是需要为 I/O 部件留出存储器地址空间的专用部分。

2.4 程序设计举例

在介绍了寻址方式和指令类型后,我们现在转而通过一些程序设计例子来说明对这些概念的使用。在介绍这些例子时将使用通用的助记符。这样做的动机在于让读者理解如何使用不同的寻址方式以完成不同的操作,而与所使用的机器无关。在第 3 章中将使用实际的机器例子来介绍类似原理的应用。

例 2.1 在本例中,我们想展示完成加 100 个数的一个程序段,这 100 个数存储在起始位置为 1000 的连续存储单元中。结果将存放到存储单元 2000。

```
        CLEAR R0;              R0 ← 0
        MOVE # 100, R1;        R1 ← 100
        CLEAR R2;              R2 ← 0
LOOP:   ADD 1000 (R2),R0;      R0 ← R0 + M (1000 + R2)
        INCREMENT R2;          R2 ← R2 + 1
        DECREMENT R1;          R1 ← R1 − 1
        BRANCH-IF > 0 LOOP;    GO TO LOOP if contents of R1 > 0
        STORE R0, 2000;        M(2000) ← R0
```

本例中使用了立即数(MOVE # 100,R_1)和变址寻址[ADD 1000(R_2),R_0]。

例 2.2 在这个例子中,将使用自动加 1 寻址来完成例 2.1 中相同的任务。

```
        CLEAR R0;              R0 ← 0
        MOVE # 100, R1;        R1 ← 100
        CLEAR R2;              R2 ← 0
LOOP:   ADD 1000 (R2)+,R0;     R0 ← R0 + M (1000 + R2) & R2 ← R2 + 1
        DECREMENT R1;          R1 ← R1 − 1
        BRANCH-IF > 0 LOOP;    GO TO LOOP if contents of R1 > 0
        STORE R0, 2000;        M(2000) ← R0
```

可见,对于一个给定的任务可以使用多种程序设计的方法加以完成。程序员所使用的方法依赖于他/她的经验,以及机器所使用的指令集的丰富性。还应注意,在例 2.2 中使用自动加 1 的寻址导致了完成同样任务所需指令数的减少。

例 2.3 该例子使用一个子程序 SORT,以升序方式排序 *N* 个值,如图 2.13 所示。这些数最初存放在从单元 1000 开始的一个表中。排序后的值也存放在同一表中,并仍以单元 1000 为起

始点。子程序使用著名的“冒泡排序”技术排序数据。在每次循环末尾检测寄存器 R_3 的内容以判断该表是否已排序完毕。

```
MAIN PROGRAM
        MOVE #999, R1          ; R1 ← 999
        MOVE N, R0             ; R0 ← N
        CALL SUBROUTINE SORT

SUBROUTINE SORT
Again:  CLEAR R3               ; R3 ← 0
Next:   COMPARE (R1), (R1+1)   ; 比较当前和下一值
        BRANCH_IF_LOWER Skip   ; 若 R1 < R1 + 1 则转移
        MOVE (R1), R2          ; 否则，交换当
        MOVE (R1 + 1), (R1)    ; 前值与
        MOVE R2 , (R1 + 1)     ; 下一值
        MOVE #1, R3            ; 指明交换
Skip:   INCEMENT R1            ; R1 ← R1 + 1
        DECREMENT R0           ; R0 ← R0 - 1
        BRANCH_IF_GREATER Next ; 转到下一值
        COMPARE R3, 1          ; 还有无交换?
        BRANCH_IF_EQUAL Again  ; 重复过程
        RETURN SORT
```

图 2.13　SORT(排序)子程序

例 2.4　该例说明如何使用子程序 SEARCH 查找 *N* 项表中的值 *VAL*，如图 2.14 所示。假定表开始时未排序，因此使用穷举查找方法。在这种查找中，值 *VAL* 从顶到底与表中每个元素进行比较。用寄存器 R_3 的内容指示是否找到值 *VAL*。表的第一个元素位于地址 1000。

```
MAIN PROGRAM
        MOVE #1000, R1         ; R1 ← 1000
        MOVE N, R0             ; R0 ← N
        CALL SUBROUTINE SEARCH

SUBROUTINE SEARCH
        CLEAR R3               ; R3 ← 0, 找到值时将该寄存器置1

        MOVE #1000, R1         ; R1 ← 1000
        MOVE N, R0             ; R0 ← N
Next:   COMPARE (R1)+,VAL      ; 使用自动加 1 方式
        BRANCH_IF_EQUAL Found  ; 若R1 = VAL就转移
        DECREMENT R0           ; R0 ← R0 - 1
        BRANCH_IF > 0 Next
        COMPARE R0, 0          ; 是否已检查表中所有值
        BRANCH_IF_EQUAL Not_Found
Found:  MOVE #1, R3            ; 在R1 -1位置发现值 VAL
Not_Found:

        RETURN SEARCH
```

图 2.14　SEARCH(搜索)子程序

例 2.5　该例说明如何使用子程序 SEARCH 查找 *N* 项表中的值 *VAL*(与例 2.4 一样)，如图 2.15 所示。但使用堆栈传送参数 *VAL* 和 *N*。

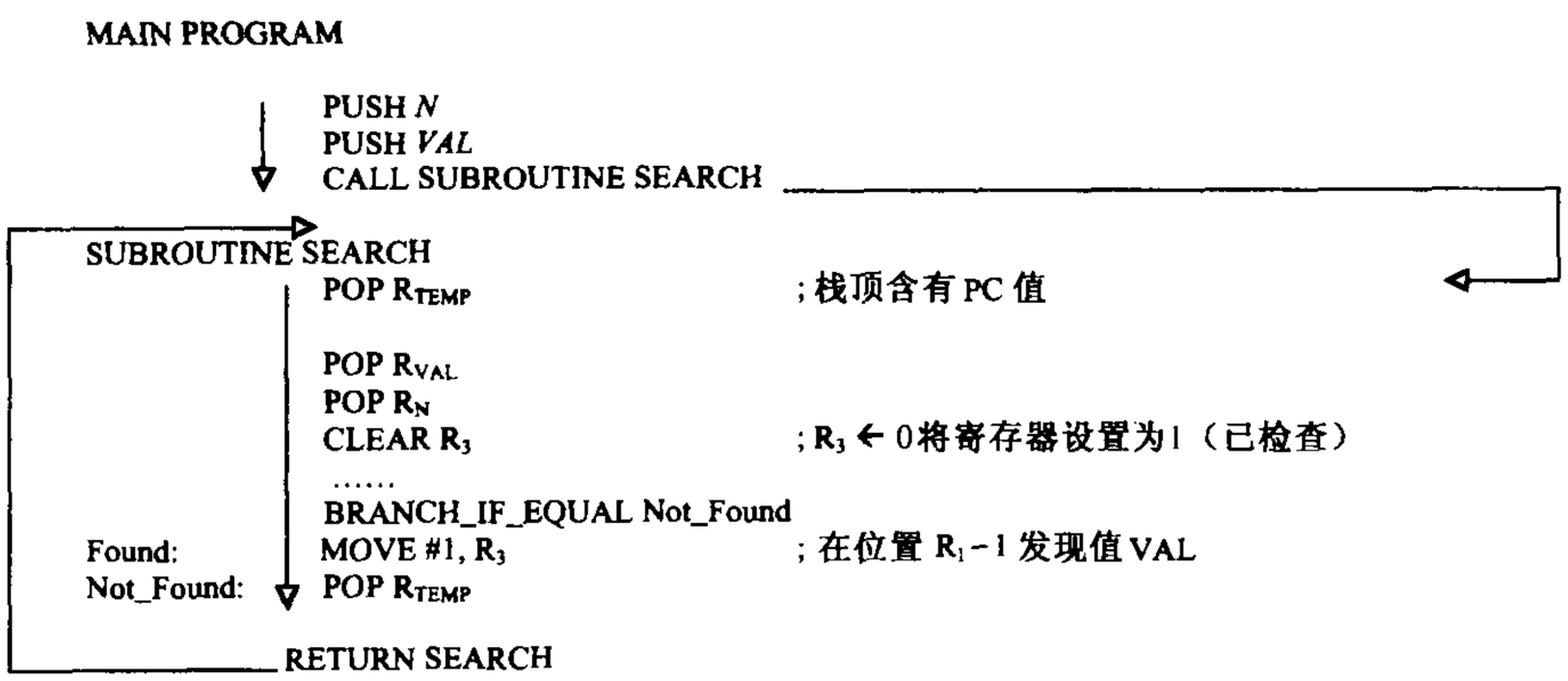


图 2.15 使用堆栈传递参数 *VAL* 和 *N* 的 SEARCH(搜索)子程序

2.5 小结

在这一章中我们论述了有关指令集设计和特征的主要问题。我们介绍了一个主存储器的模型,在该模型中存储器被抽象为一个单元序列,每个单元能存储 n 位。还介绍了许多寻址方式,它们包括立即数、直接、间接、变址、自动加 1 和自动减 1。然后我们举例说明了如何使用这些寻址方式。我们也对指令类型进行了讨论,这些指令类型包括数据移动、算术/逻辑、指令排序以及输入/输出。在本章的最后,我们通过一些例子来说明如何使用在这一章中所论述的原理和概念进行编程,以求解一些问题。在下一章,我们将介绍使用汇编语言编程求解实际生活问题时所用到的一些概念。

习题

1. 使用 2.2 节和 2.3 节中介绍的寻址方式和指令类型,编写一个程序使存储在 16 位寄存器 R_0 中的二进制位次序颠倒。
2. 考虑一个计算机有多个寄存器,其中 $R_0 = 1500, R_1 = 4500, R_2 = 1000$。请指出下列每条指令中存储器的有效地址和寄存器的内容(假定所有的数均为十进制数)。
   (a) ADD$(R_0)+, R_2$
   (b) SUBTRACT$-(R_1), R_2$
   (c) MOVE $500(R_0), R_2$
   (d) LOAD $\#5000, R_2$
   (e) STORE $R_0, 100(R_2)$
3. 假定程序中堆栈的栈顶由寄存器 SP 指定。请编写程序段以完成如下任务(假定只有如下寻址方式可用:变址,自动加 1,自动减 1)。
   (a) 从栈顶退出 3 个元素,将它们相加,并将结果压回堆栈。
   (b) 从栈顶退出 2 个元素,将它们相减,并将结果压回堆栈。
   (c) 向栈顶压入 5 个元素(每次一个)。
   (d) 从栈顶移走最顶上的 5 个元素。
   (e) 从栈顶复制第 3 个元素到寄存器 R_0。

4. 要求编写一个能完成操作 $C \leftarrow A + B$ 的程序段，其中 A 和 B 各自表示 100 个存储单元的集合，每个存储单元中存放一个值。由 A 代表的值集合存储的起始单元为 1000，而由 B 代表的值集合存储的起始单元为 2000。存储操作结果的起始单元为 3000。使用以下每一种指令类完成以上的操作。
   (a) 具有单地址指令的机器；
   (b) 具有一个半地址指令的机器；
   (c) 具有两地址指令的机器；
   (d) 具有三地址指令的机器；
   (e) 具有零地址指令的机器。
5. 要求使用习题 4 中所指明的每一种指令类编写一个能完成操作 $C \leftarrow C + A \times B$ 的程序段。假定 A，B 和 C 是存储器地址。
6. 假定对 50 个学生的班级连续进行 5 次测验。每次测验学生所得的分数存放在起始地址分别为 1000，2000，3000，4000 和 5000 的连续存储单元中。要求你编写一个程序计算每个学生 5 次测验所得的平均分数，并将其存放到起始地址为 6000 的连续存储单元中。每个学生用他/她的学生 ID 加以识别。假设学生的 ID 是连续的。
7. 假定所使用的存储器是以字节寻址的，且每个分数占用 32 位，重做习题 6。
8. 假定学生的 ID 是非连续的，即每个学生的 ID 用于指向他/她测验分数的指针，重写习题 6 中的相同程序。
9. 重做上述的习题 6，假定学生的分数存储在一个数组 S(50,5)中，即每一行存有一个学生所得的分数(每个分数在同一行的一列中)以及数组的第一个元素，即 S(0,0)存放在存储单元 4000 中。以行为主存放分数，即一行接一行。每个学生的平均分数将存放在由他/她的 ID 所指向的一个存储单元中。
10. 重做上述的习题 9，假定你的工作是编写一个子程序完成习题 9 中同样的任务。假设学生数、测验次数以及数组中的第 1 个元素各自作为在寄存器 R_1，R_2 和 R_3 中的参数被传递到子程序。

参考文献和推荐读物

C. M. Gilmore, *Microprocessors : Principles and Applications*, 2nd ed., McGraw-Hill, New York, 1996.

V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed., McGraw-Hill, New York, 2002

A. D. Patterson, J. L. Hennessy, *Computer Organization & Design; The Hardware/Software Interface*, Morgan Kaufmann, San Mateo, CA, 1994

B. Wilkinson, *Computer Architecture : Design and Performance*, 2nd ed., Prentice-Hall, Hertfordshire, UK, 1996.

第3章　汇编语言程序设计

在第2章中，我们介绍了涉及机器指令系统设计的基本概念和原理。本章将讨论有关汇编语言程序设计的问题。虽然多年来我们见证了高级语言和编译技术的巨大进步，但汇编语言在某些情况下仍是必需的。汇编程序设计可使所生成的机器代码比由高级语言的编译器所生成的要小得多也快得多。在某些嵌入式和可移动的应用中，小和快的代码可能是非常关键的，因为在那种场合资源可能是很有限的。在这些场合下，频繁使用的一小部分程序就可用汇编语言编写。对于本书的读者来讲，学习汇编语言和编写汇编代码对于理解计算机组成和体系结构是非常有用的。

计算机程序可以在不同的抽象级上表示。程序可以用与机器无关的高级语言如 Java 或 C++ 编写。但一台计算机只能执行以机器语言表示的程序，该机器语言对它的体系结构来讲是专用的。给定体系结构的机器语言程序是以二进制形式表示的机器指令的集合。以任何高于机器语言级别编写的程序，在一台计算机能执行它们之前，必须转换成二进制的表示形式。汇编语言程序是机器语言程序的符号表示。机器语言是纯二进制代码，而汇编语言是二进制代码直接映射成为人们更易了解和管理的符号形式。将符号表示转换成机器语言是由称为汇编器的专用程序来完成的。汇编器是这样的一种程序，它接受一个符号语言程序(源)，并产生等价的机器语言(目标)。在将一个程序转换成二进制代码时，汇编器用数字地址替换符号地址，用机器操作码替换符号操作码，为指令和数据预定存储器，并且将常数翻译成机器表示。

本章的目的是使读者对汇编语言和它的程序设计有一个总的了解。它不是任何特定体系结构的汇编语言手册。本章的论述将从一个简单的、贯穿于整章的假想机开始。该假想机只有5个寄存器，且它的指令系统只有10种指令。我们用这个简单的机器定义一组相当简易的汇编语言，旨在易于理解并有助于解释汇编程序设计中的主要问题。我们将介绍指令的助记符和它的句法以及汇编器指令和命令。然后我们对汇编程序的执行进行讨论。我们以展示 Intel x86 CISC 系列汇编语言的实际例子结束本章。

3.1　一台简单机器

机器语言是某一特定处理器的固有语言。由于汇编语言是机器语言的符号形式，因此不同类型的处理器有其自身独特的汇编语言。在对给定处理器的汇编语言进行研究之前，我们首先需要了解该处理器的细节。我们需要知道存储器的大小和它的组成、处理器的寄存器、指令格式，以及整个的指令系统。在本节中，我们将提出一个非常简单的假想机，它将在整章中用来说明汇编语言的不同论题。

我们的简单机器是一个基于累加器的处理器，它有5个16位的寄存器：程序计数器(PC)，指令寄存器(IR)，地址寄存器(AR)，累加器(AC)和数据寄存器(DR)。PC 中含有下一条要执行指令的地址。IR 含有要执行指令的操作码部分。AR 含有要执行指令的地址(如果有)部分。AC 用做隐含的数据源和目的。DR 用来保存数据。存储器部件由4096个字的存储部件组成，字长为16位。如图3.1所示为一台简单机器。

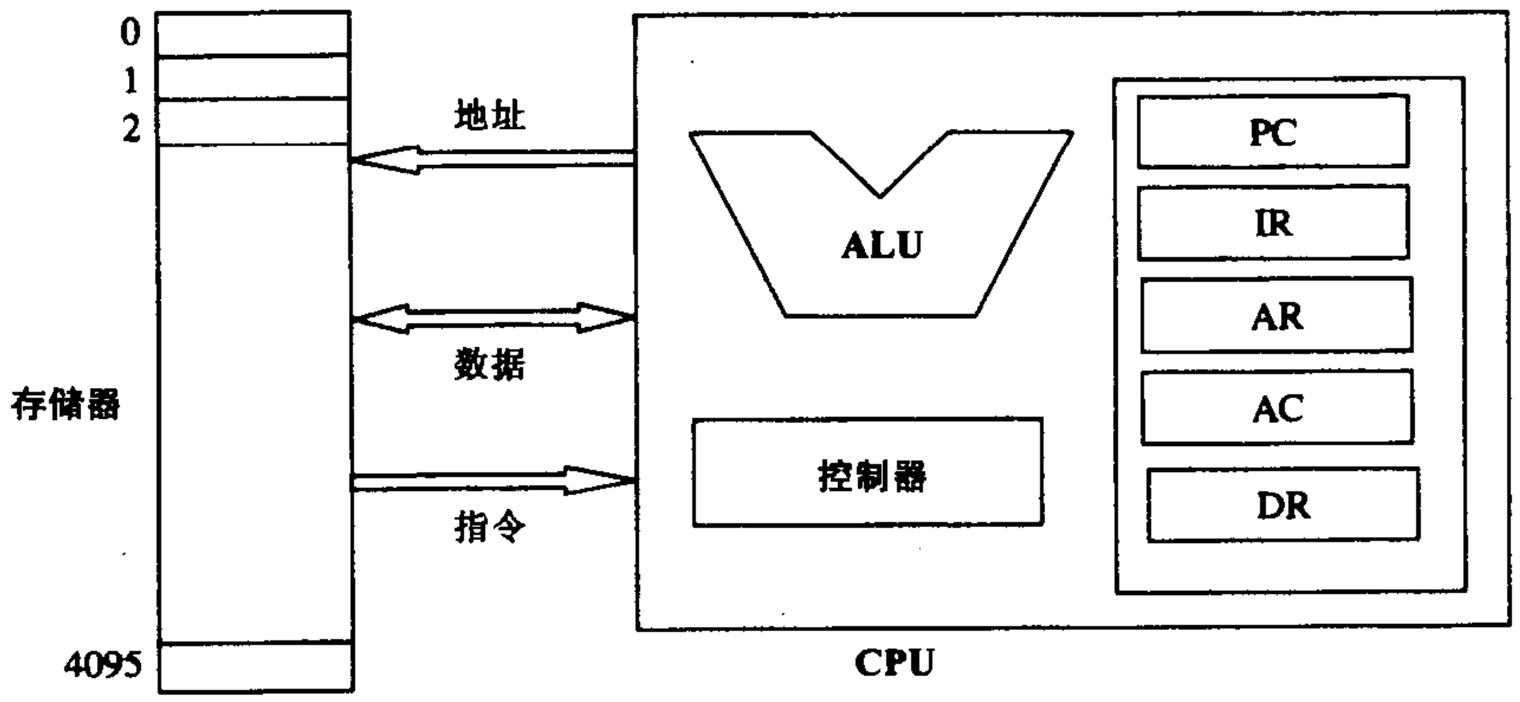


图 3.1　一台简单机器图

我们假设该简单机器支持 3 种类型的指令:数据传输、数据处理和程序控制。数据传输操作即寄存器 AC 和 DR 之间的存、取和数据移动。数据处理指令是加、减、与和非。程序控制指令为跳转和条件跳转。表 3.1 中总结了该处理器的指令系统。指令长度为 16 位,其中 4 位为操作码,12 位为地址(当使用时)。

表 3.1　简单处理器的指令集

操作码	操作数	指令意义
0000		停止执行
0001	adr	将操作数从存储器(单元地址)装载到 AC
0010	adr	存 AC 内容到存储器(单元地址 adr)
0011		将 AC 内容复制到 DR
0100		将 DR 内容复制到 AC
0101		加 DR 到 AC
0110		从 AC 减 DR
0111		DR 与 AC 按位与
1000		AC 内容取补
1001	adr	跳转到地址为 adr 的指令处
1010	adr	若 AC = 0,就跳转到地址为 adr 的指令处

例 3.1　让我们编写一个机器语言程序,它将把初始内容为 350 的存储单元 12(00C,十六进制)与初始内容为 96 的存储单元 14(00E,十六进制)相加,并把结果存入到初始内容为 0 的存储单元 16(010,十六进制)。

表 3.2 给出了以二进制指令表示的程序。第 1 栏中给出的是以二进制表示的每条指令和操作数的存储单元。第 2 栏中列出的是存储单元的内容。例如,单元 0 的内容是操作码为 0001 且操作数地址为 0000 0000 1100 的指令。请注意当遇到指令不需要操作数的情况时,指令的操作数部分应写为 0。在执行时,该程序应存放在自单元 0 开始的存储单元中。若程序要被存放在不同的存储单元中,则需要改写某些指令的地址以反映新的位置。

很显然,用二进制代码编写的程序是很难理解的,当然,也是很难调试的。用十六进制表示指令可将数字位减少为每条指令只需 4 个。表 3.3 给出了同一程序的十六进制表示。

表 3.2 以二进制表示的简单机器语言程序(例 3.1)

存储单元(字节)	二进制指令	描述
0000 0000 0000	0001 0000 0000 1100	将单元 12 的内容装载到 AC
0000 0000 0010	0011 0000 0000 0000	将 AC 中内容移到 DR
0000 0000 0100	0001 0000 0000 1110	将单元 14 的内容装载到 AC
0000 0000 0110	0101 0000 0000 0000	将 DR 加到 AC
0000 0000 1000	0010 0000 0001 0000	将 AC 的内容存到单元 16
0000 0000 1010	0000 0000 0000 0000	停止
0000 0000 1100	0000 0001 0101 1110	数据值 350
0000 0000 1110	0000 0000 0110 0000	数据值 96
0000 0001 0000	0001 0000 0000 1100	数据值 0

表 3.3 以十六进制表示的简单机器语言程序(例 3.1)

存储单元(字节)	十六进制指令
000	100C
002	3000
004	100E
006	5000
008	2010
00A	0000
00C	015E
00E	0060
010	0000

3.2 指令助记符和句法

汇编语言是符号形式的机器语言。汇编程序以称为助记符的简短缩写符编写。一个助记符是代表实际机器指令的缩写符。汇编语言程序设计是用助记符形式编写机器指令,其中每条机器指令(二进制或十六进制)用一个助记符替代。显然,使用助记符比起十六进制或二进制的值更有意义,它将使低级的程序设计更加容易,也更易管理。

一个汇编程序由一串汇编语句组成,其中每行写一条语句。汇编程序的每一行分为以下的 4 个字段:标号,操作码(opcode),操作数,注释。图 3.2 给出了一条汇编指令的 4 栏格式。

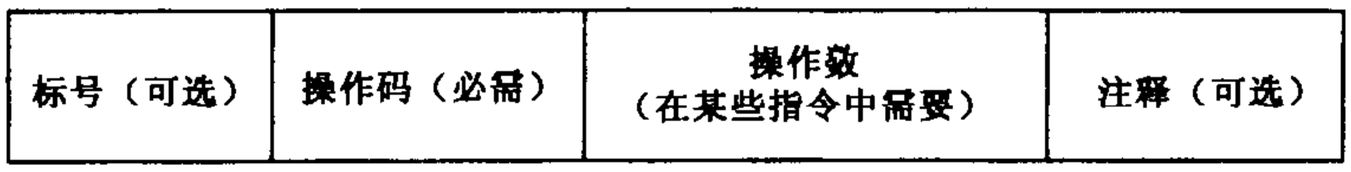


图 3.2 汇编指令格式

汇编语言使用标号为存储器地址提供符号名。在程序行前使用的标号是一个识别符,以便转移到该标号行。不同汇编语言的最大标号长度是不同的。有些长度可达 32 个字符,而有些可能限制为 6 个字符。某些处理器的汇编语言在每个标号后需要一个冒号,而有些则不需要。例如,SPARC 汇编在每个标号后需要一个冒号,但 Motorola 汇编就不需要。Intel 的汇编在代码标号后需要冒号,但数据标号后就不需要冒号。

操作码(opcode)字段含有一个给定操作的符号缩写。操作数字段则由操作码所需的附加信息或数据组成。操作数字段可用来说明常数、标号、立即数、寄存器或一个地址。注释字段提供书写文档空间,解释要完成什么动作以方便调试和维护。

对于前一节所描述的简单处理器,我们假定它的标号字段(可能是空的)最多可为 6 个字符。在每个标号后不需要冒号。在注释前将使用“ \ ”。表 3.4 概括了表 3.1 中 10 条二进制指令的简单助记符。

表 3.4　简单处理器的汇编语言

助记符	操作数	指令意义
STOP		停止执行
LD	x	从存储器(单元 x)装载操作数到 AC
ST	x	将 AC 内容存到存储器(单元 x)
MOVAC		复制 AC 内容存到 DR
MOV		复制 DR 内容存到 AC
ADD		加 DR 到 AC
SUB		从 AC 减 DR
AND		按位与 DR 和 AC
NOT		将 AC 内容取补
BRA	adr	跳转到地址为 adr 的指令
BZ	adr	若 AC = 0,跳转到地址为 adr 的指令

让我们考虑以下的汇编指令:

```
START    LD X    \ copy the contents of location X into AC
```

指令 LD X 的标号是 START,其意义是它是该指令的地址。在程序中该标号可用做访问标记,如下面指令所示:

```
BRA START    \ go to the statement with label START
```

跳转指令将使处理器跳转到地址标号为 START 的存储器地址,这样在 BRA 指令后就直接执行指令 LD X。

除了程序指令以外,一个汇编程序也可能包含伪指令或汇编器指令。汇编器指令是汇编程序能理解的命令,而不对应于实际的机器指令。例如,能请求汇编器分配存储器。在我们简单的处理器中,我们假定能使用伪指令 W 在存储器中预留一个字(16 位)。例如,下面的伪指令为标号 X 预留一个字,并将其初始化为十进制值 350:

```
X    W    350    \ reserve a word initialized to 350
```

同样,伪指令 W 350 的标号是 X,这表示它是该值的存储器地址。以下是前一节中例 3.1 机器语言程序的汇编代码:

```
     LD X          \ AC ← X
     MOVAC         \ DR ← AC
     LD Y          \ AC ← Y
     ADD           \ AC ← AC + DR
     ST Z          \ Z ← AC
     STOP
X    W    350     \ reserve a word initialized to 350
Y    W    96      \ reserve a word initialized to 96
Z    W    0       \ result stored here
```

例 3.2 在此例中,我们将编写一个完成乘法操作的汇编程序:Z←X × Y,其中 X ,Y 和 Z 为存储单元。

如我们所知,简单机器的汇编不包括乘法操作。我们将通过使用多次加法操作来计算乘积。为了加 Y 到本身 X 次,我们使用 N 作为计数器,并将它初始化为 X,在完成每一次加法后就将其减 1。BZ 指令用来测试 N 到达 0 的情况。我们将使用一个存储单元存储 N,但需在 BZ 指令执行前将它装载到 AC。我们还将用一个存储单元 ONE 存储常数 1。存储单元 Z 用来存储部分乘积和最终结果。

以下所列的是使用我们简单处理器汇编语言所编写的程序。我们假设 X 和 Y 值足够小,以允许它们的乘积能以单字存储,由于这个原因,我们将 X 和 Y 分别初始化为 5 和 15。

```
        LD X          \ Load X in AC
        ST N          \ Store AC (X original value) in N
LOOP    LD N          \ AC ← N
        BZ EXIT       \ Go to EXIT if AC = 0 (N reached 0)
        LD ONE        \ AC ← 1
        MOVAC         \ DR ← AC
        LD N          \ AC ← N
        SUB           \ subtract 1 from N
        ST N          \ store decrements N
        LD Y          \ AC ← Y
        MOVAC         \ DR ← AC
        LD Z          \ AC ← Z (partial product)
        ADD           \ Add Y to Z
        ST Z          \ store the new value of Z
        BRA LOOP
EXIT    STOP
X       W       5     \ reserve a word initialized to 5
Y       W       15    \ reserve a word initialized to 15
Z       W       0     \ reserve a word initialized to 0
ONE     W       1     \ reserve a word initialized to 1
N       W       0     \ reserve a word initialized to 0
```

3.3 汇编器指令和命令

在前一节中,我们向读者介绍了汇编和机器语言。我们用简单的机器模型编写了几个汇编代码段。在为一个特定的体系结构编写汇编语言程序时,必须考虑如下一些实际问题:

- 汇编器指令;
- 符号的使用;
- 合成操作的使用;
- 汇编器句法;
- 与操作系统的交互。

汇编器指令(directives,也称为伪操作,pseudo-operation)的使用在编写汇编语言程序时是一个重要的问题。汇编器指令是由汇编器解读的命令,它并不对应实际的机器指令。汇编器指令会影响汇编器将汇编代码转换成机器代码的方法。例如,可以使用特殊的汇编器指令使汇编器以合适的对准方式存放数据项。为有效实现体系结构需要在存储器中对准数据。为了使数据正

确地对准，n 字节宽的数据必须以可被 n 整除的地址存放，例如 2 字节宽的一个字必须存放在地址可被 2 整除的单元中。

在汇编语言程序中，用符号表示数，例如立即数。这样做就可使代码更易读、易理解和易调试。然后由汇编器将符号翻译成相应的数字值。

合成操作的使用有助于程序员使用不为体系结构直接支持的指令。然后由汇编器将这些合成操作翻译成为体系结构所定义的一组指令。例如，汇编器允许在没有定义增量指令的体系结构上通过使用某些其他指令(如加法指令)来使用(合成的)增量指令。

汇编器常常在涉及硬件的成分如寄存器和存储单元时施加某些规约。其中之一是在立即数值前冠以特殊符号(如 #)或在寄存器名前冠以符号(如%)。

某些机器的底层硬件不能由程序员直接访问。操作系统(OS)会起到访问如存储器和 I/O 设备的媒介作用。与操作系统的交互能以代码形式发生，该代码将导致属于 OS 一部分的函数的执行。这些函数称为系统调用(system call)。

3.4　程序汇编和执行

现在你可能已经知道，用汇编语言编写的程序在它执行之前需要翻译成二进制机器语言。本节将学习如何从编写一个汇编程序过渡到执行阶段。图 3.3 给出了在汇编和执行过程中的 3 个步骤。首先，汇编器阅读由汇编语言编写的源程序并生成二进制的目标程序。然后，汇编器将目标程序传送给连接程序(linker)，连接程序将检查目标文件以调用连接库中的过程。连接程序接着将所需连接库中的过程与目标程序组合在一起并生成可执行的程序。之后，装载程序(loader)将可执行的程序装入存储器并使 CPU 转移到起始地址。此后程序就开始执行。

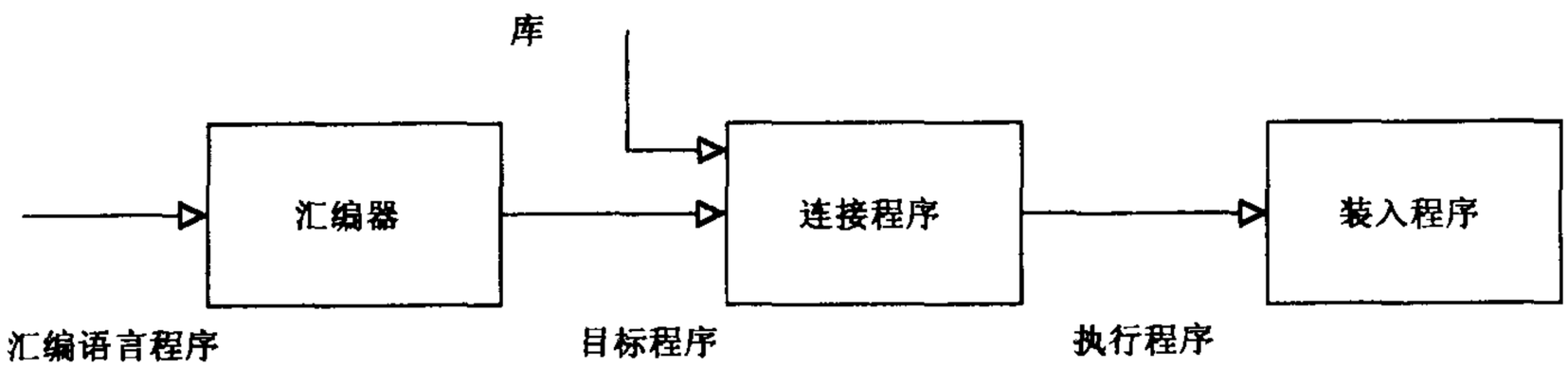


图 3.3　汇编和执行过程

3.4.1　汇编器

汇编器是将以汇编语言编写的源代码程序转换为机器代码指令的程序。汇编器用数字地址替换符号地址，用机器操作代码替换符号操作代码，为指令和数据保留存储器，并将常数翻译成机器表示。

汇编器的功能通过扫描汇编语言程序并将它的指令映射成与它们等价的机器代码指令加以完成。由于符号在指令中可以先使用后定义，因此对程序的一次扫描可能无法完成映射。一个简单的汇编器将扫描(scan)整个汇编程序两次，这里每次扫描称为一遍(pass)。在第一遍扫描中，汇编器产生包括所有符号和它们的二进制值的一张表，该表称为符号表。在第二遍扫描中，汇编程序将使用该符号表以及其他表以生成目标程序，并输出连接程序所需的一些信息。

3.4.2 数据结构

汇编器为完成它的功能至少需使用三张表:符号表、操作码表和伪指令表。在第一遍扫描中生成的符号表对应程序中的每个符号。与每个符号有关的是它的二进制值和其他信息。表 3.5 中给出了例 3.2 乘法程序段中的符号表。我们假定指令起始于存储器中的 0 号单元。由于每条指令为两个字节,故符号 LOOP 的值为 4(十六进制为 004)。而符号 N 将存储在十进制单元 40(十六进制为 028)。其他符号的值可以用类似方法得到。

表 3.5 乘法段的符号表(例 3.2)

符号	值(十六进制)	其他信息
Loop	004	
EXIT	01E	
X	020	
Y	022	
Z	024	
ONE	026	
N	028	

操作码表提供有关操作码的信息。与表中每个符号操作码有关的是它的数字值和有关其类型、指令长度及其操作数的其他信息。表 3.6 中给出了在 3.1 节中所描述的简单处理器的操作码表。作为一个例子我们解释与操作码 LD 有关的信息。它有一个操作数,是一个存储器地址,它的二进制值为 0001。LD 的指令长度为 2 B,它的类型是存储器访问。

表 3.6 简单处理器汇编语言的操作码表

操作码	操作数	操作码值(二进制)	指令长度(字节)	指令类型
STOP	—	0000	2	控制
LD	Mem-adr	0001	2	访问存储器
ST	Mem-adr	0010	2	访问存储器
MOVAC	—	0011	2	访问寄存器
MOV	—	0100	2	访问寄存器
ADD	—	0101	2	访问寄存器
SUB	—	0110	2	访问寄存器
AND	—	0111	2	访问寄存器
NOT	—	1000	2	访问寄存器
BRA	Mem-adr	1001	2	控制
BZ	Mem-adr	1010	2	控制

伪指令表中的项是伪指令符号。每一项使汇编器去访问一个过程以处理在程序中遇到的伪指令。例如,若遇到 END 就终止翻译过程。

为了了解指令位置的动态情形,汇编器维护一个称为指令定位计数器(ILC, instruction location counter)的变量。ILC 含有分配给正在处理的指令或操作数的存储单元的值。ILC 初始化为 0,并在每一条指令处理后进行增量。ILC 使用正被处理的指令长度进行增量,或是以所分配的字节数进行增量(作为一条数据分配伪指令的结果)。

图 3.4 和图 3.5 给出了两遍扫描汇编器的第一遍扫描和第二遍扫描的简化流程图。记住第一遍扫描的主要功能是建立符号表，而第二遍扫描的主要功能是生成目标代码。

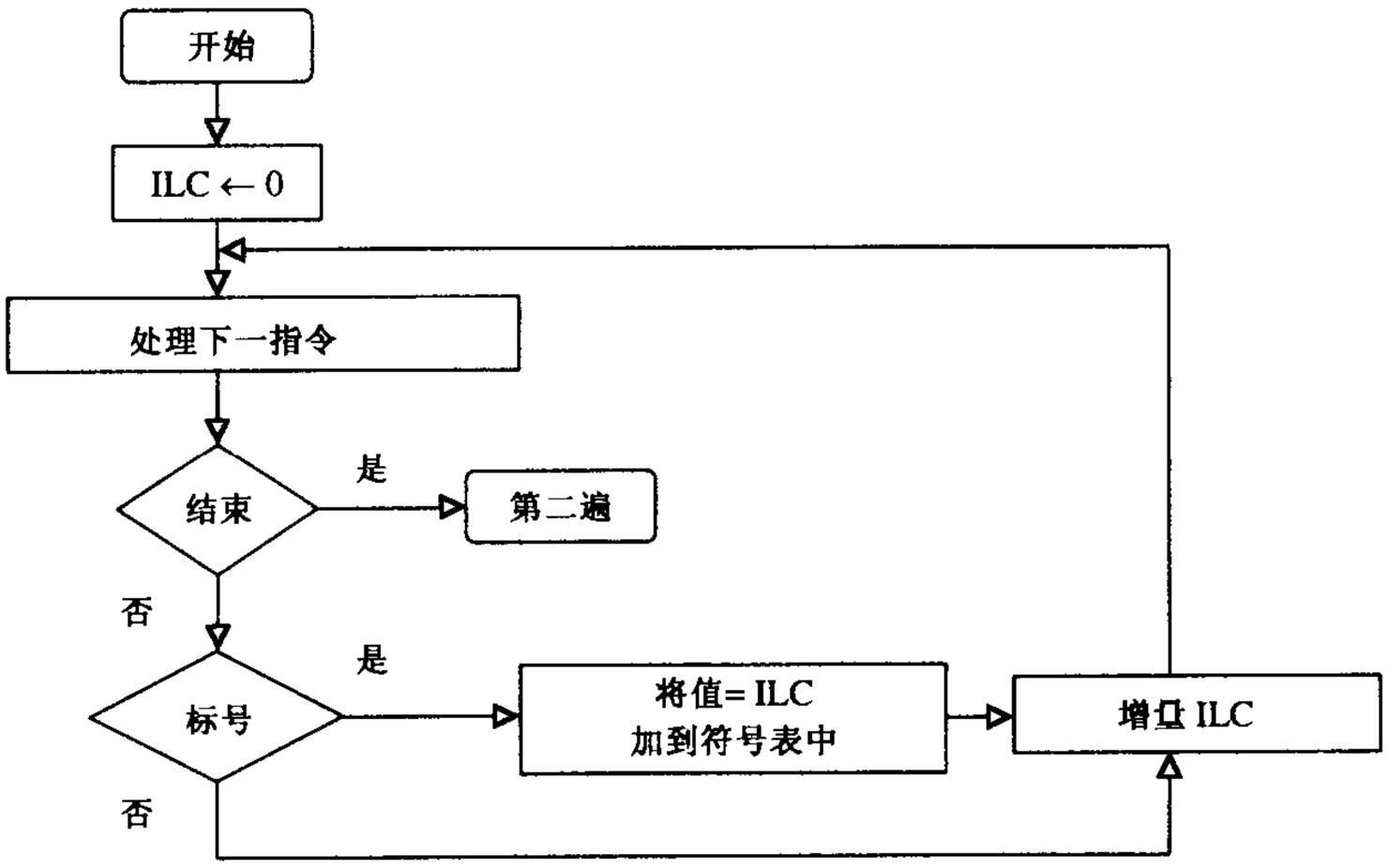


图 3.4　两遍汇编器中简化的第一遍

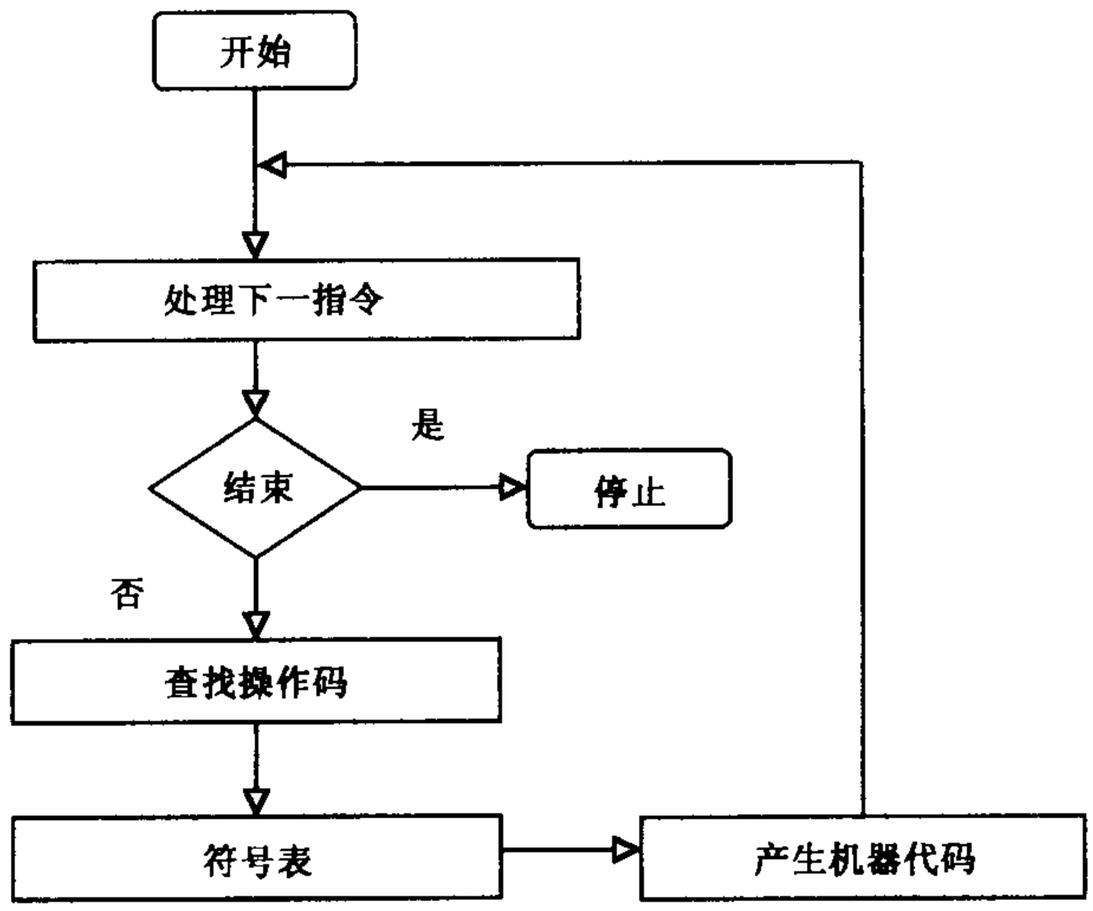


图 3.5　两遍汇编器中简化的第二遍

3.4.3　连接程序和装载程序

连接程序是能够将多个目标模块组合在一起的实体，这些目标模块是由于分别汇编多个汇编模块形成的。装载程序是操作系统实用程序，它将可执行代码读入存储器并启动执行。

总之，在汇编模块被翻译成目标模块后，连接程序和装载程序的功能是准备程序的执行。这些功能包括将目标模块组合在一起，明确在汇编时间未知的地址，分配存储器并在最后执行程序。

3.5 举例:x86 系列

在这一节中我们将讨论汇编语言的特征以及 x86 系列的使用。我们将给出系统的基本结构特征、基本的程序设计模型、寻址方式和所使用的不同指令类型的样例,并在最后展示如何在编程解决实际生活问题时使用系统的汇编语言。

在 20 世纪 70 年代后期,Intel 公司推出了它的第一个 16 位微处理器 8086,该处理器有一条 16 位外部总线。由 8086 演化出一系列速度更快、功能更强的处理器,从 80286 开始到奔腾(Pentium)结束。奔腾是在 1993 年推出的。Intel 处理器系列通常称为 x86 系列。表 3.7 总结了该系列中主要成员的主要特征。

表 3.7 Intel x86 微处理器系列的主要特征

特性	8086	286	386	486	奔腾
推出日期	1978	1982	1985	1991	1993
数据总线	8 b	16 b	32 b	32 b	64 b
地址总线	20 b	24 b	32 b	32 b	32 b
运算速度	5,8,10 MHz	6,8,10, 12.5,16,20 MHz	16,20,25,33, 40,50 MHz	25,33,50 MHz	50,60, 66,100 MHz
指令高速缓存大小	NA	NA	16 B	32 B	8 KB
数据高速缓存大小	NA	NA	256 B	8 KB	8 KB
物理存储器	1 MB	16 MB	4 GB	4 GB	4 GB
数据字大小	16 b	16 b	16 b	32 b	32 b

Intel 奔腾处理器大约有 300 万个晶体管,它的计算能力是它的前一代处理器 80486 的 2 到 5 倍。奔腾处理器中引入了许多新的特征,其中之一是融入了双流水线超标量体系结构,它能在一个时钟周期内处理多于一条的指令。

图 3.6 中给出了 386,486 以及奔腾的基本程序设计模型。模型由 3 组寄存器组成,它们是通用寄存器组,段寄存器组以及指令指针(程序计数器)和标志寄存器组。第一组通用寄存器组包括通用寄存器 A,B,C,D,SI(源索引)、DI(目的索引)、SP(堆栈指针)和 BP(基指针)。应该注意的是在命名这些寄存器时,我们使用 X 表示扩展(extended)。第二组段寄存器组由 CS(代码段)、SS(堆栈段)以及 4 个数据段寄存器 DS,ES,FS 和 GS 组成。第三组由指令指针(程序计数器)和标志(状态)寄存器组成。图 3.7 中给出了标志寄存器,其中前 5 个与早期 8085 8 位微处理器中引入的那些位是相同的,而后面的 6 ~ 11 位与 8086 引入的那些位是一样的,标志中的 12 ~ 14 位是在 80286 中引入的,而 16 ~ 17 位由 80386 引入。第 18 位标志位是由 80486 引入的。表 3.8 给出了这些标志的意义。

在 x86 系列中,一条指令可对一个或两个操作数进行操作。在二操作数指令中,第二个操作数可以是 2 补码格式的立即数。数据传输、算术和逻辑指令可作用于立即数、寄存器或存储单元。

在 x86 系列中可以使用直接和间接存储器寻址。直接寻址中使用由一个 8,16 或 32 位的字所组成的位移地址作为逻辑地址。将逻辑地址加上移位后的段寄存器(段基地址)内容就可得到一个物理地址。图 3.8 说明了直接寻址的过程。

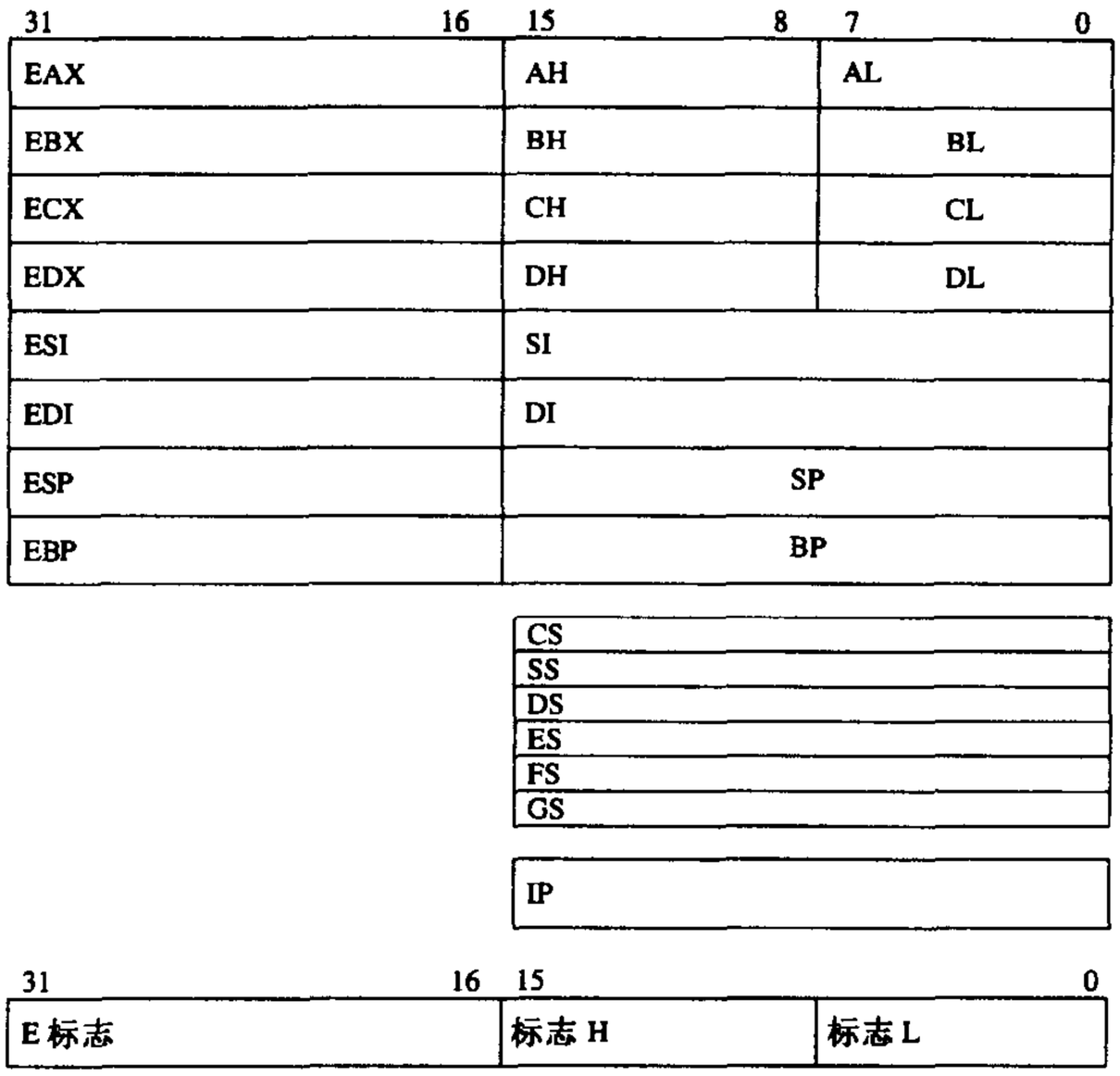


图 3.6　x86 程序设计模型的基本寄存器集

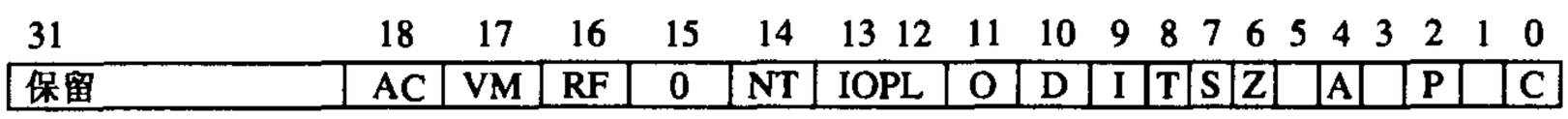


图 3.7　x86 的标志寄存器

表 3.8　x86 的状态标志

标志	意义	处理器	标志	意义	处理器
C	进位	所有	P	奇偶校验	所有
A	辅助	所有	Z	0	所有
S	符号	所有	T	陷阱	所有
I	中断	所有	D	方向	所有
O	溢出	所有	IOPL	I/O 优先级	286
NT	嵌套任务	286	RF	继续	386
VM	虚拟方式	386	AC	对准检查	486

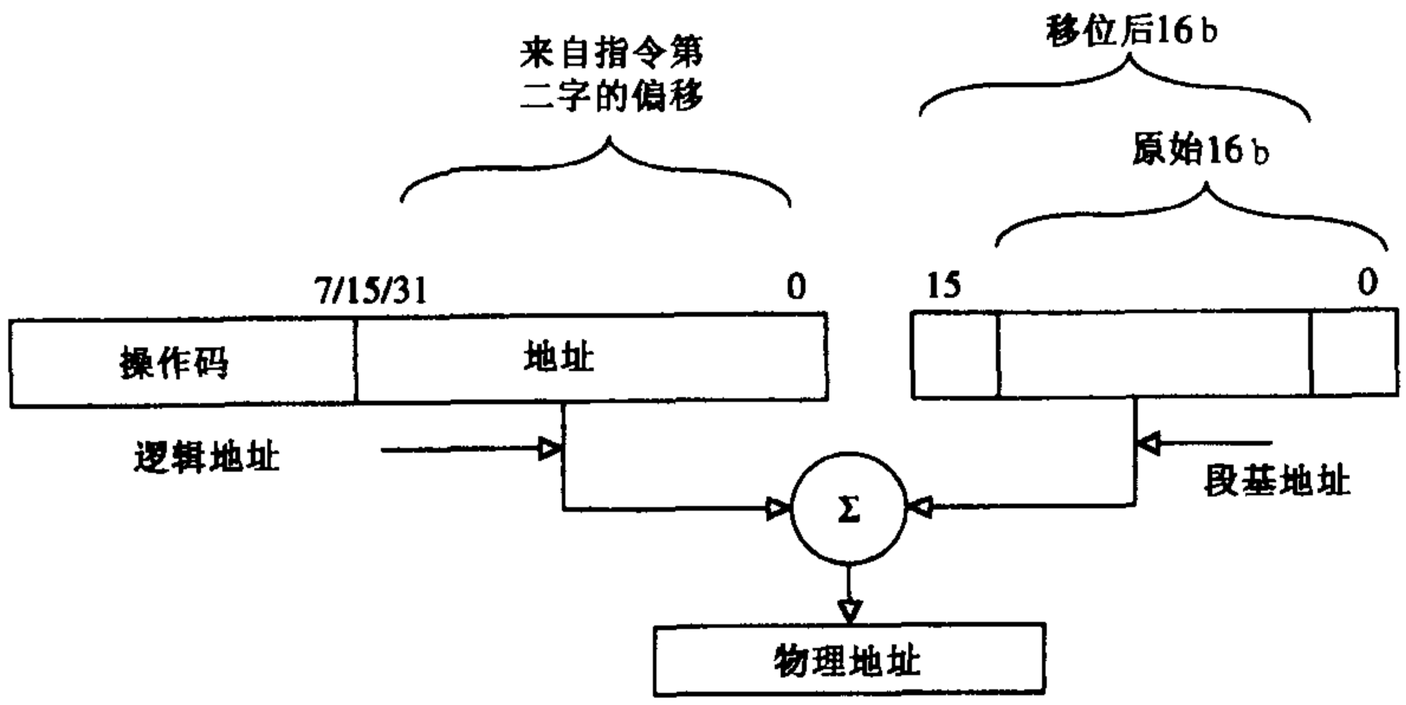


图 3.8　x86 系列中的直接寻址

x86 系列中的间接地址可通过使用基指针寄存器(BPR)内容、变址寄存器内容或基寄存器与变址寄存器的和得到。图 3.9 说明了使用 BPR 进行间接寻址的过程。

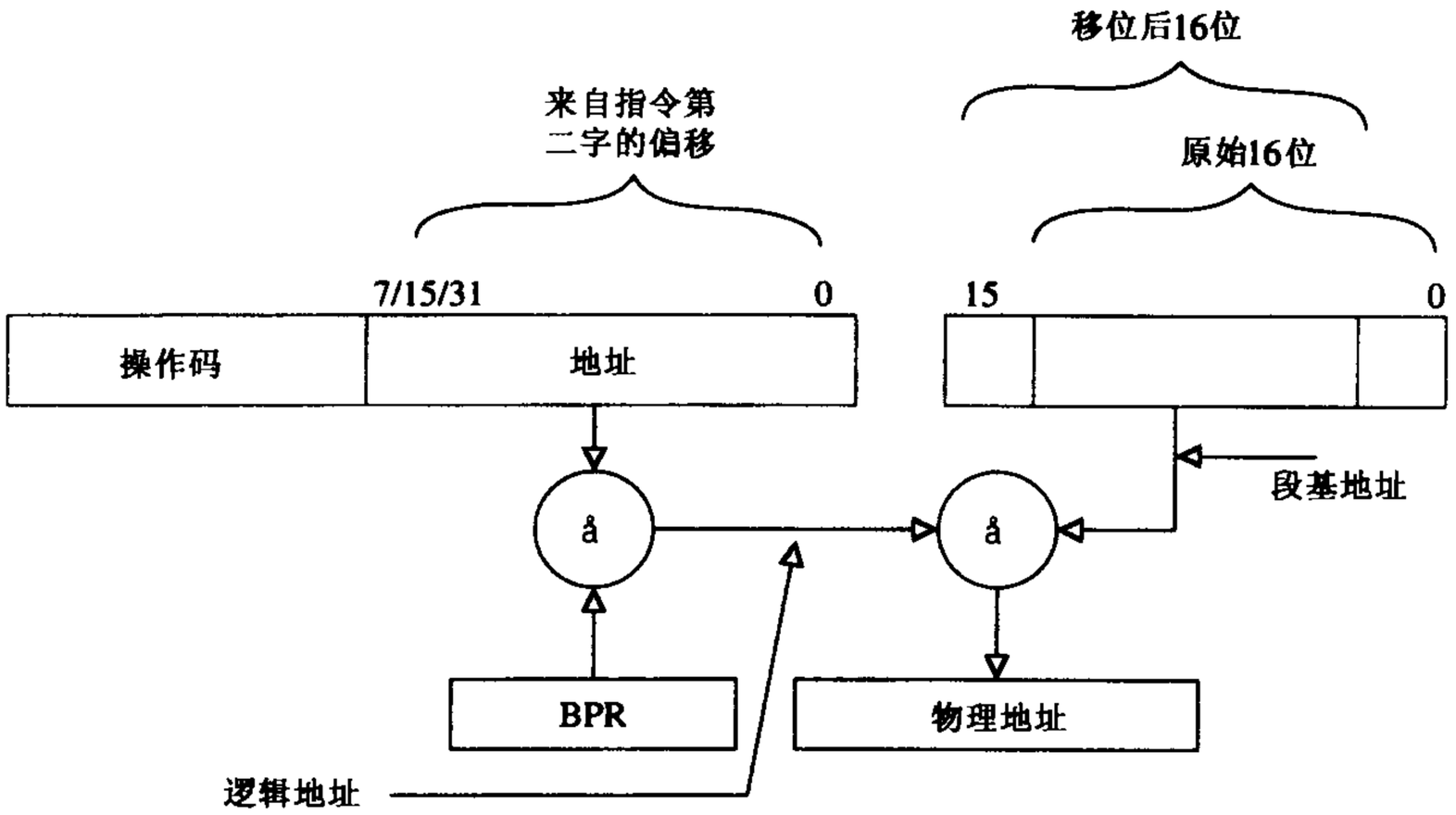


图 3.9 x86 系列中使用 BPR 的间接寻址

x86 处理器系列中定义了许多指令类型,符合前面所介绍的命名约定。这些指令类型是数据移动、算术和逻辑,以及排序(控制转移)。另外,在 x86 系列中还定义了如字符串处理、位处理和支持高级语言的其他指令类型。

在 x86 系列中的数据移动指令主要包括 4 种子类型,它们是通用、累加器 - 专用、地址 - 对象和标志指令。表 3.9 给出了这些指令的一些例子。

表 3.9 x86 数据移动指令的实例

助记符	操作	子类型
MOV	移动源到目的	通用
POP	源出栈	通用
POPA	全出栈	通用
PUSH	源进栈	通用
PUSHA	全进栈	通用
XCHG	交换源与目的	通用
IN	输入到累加器	累加器
OUT	从累加器输出	累加器
XLAT	翻译字节的表查找	累加器
LEA	装载有效地址到寄存器	地址 - 对象
LMSW	装载机器状态字	地址 - 对象
SMSW	存储机器状态字	地址 - 对象
POPF	标志出栈	标志
PUSHF	标志进栈	标志

在 x86 系列中的算术和逻辑指令主要包括 5 种子类型,它们是加、减、乘、除以及逻辑指令。表 3.10 给出了算术指令的一些例子。

表 3.10　x86 算术指令的举例

助记符	操作	子类型
ADD	加源到目的	加法
ADC	带进位加源到目的	加法
INC	操作数加 1	加法
SUB	从源减目的	减法
SBB	带进位从源减目的	减法
DEC	操作数减 1	减法
MUL	不带符号源乘目的	乘法
IMUL	带符号源乘目的	乘法
DIV	不带符号源除累加器	除法
IDIV	带符号源除累加器	除法

逻辑指令包括典型的 AND,OR ,NOT,XOR 和 TEST。TEST 指令对源和目的进行逻辑比较并相应地设置标记。另外，x86 系列有一组移位和旋转指令。表 3.11 给出了这些指令的一些例子。

表 3.11　x86 移位和旋转指令的举例

助记符	操作
ROR	右旋转
ROL	左旋转
RCL	带进位左旋转
RCR	带进位右旋转
SAR	算术右移
SAL	算术左移
SHR	逻辑右移
SHL	逻辑左移

在 x86 系列中的控制转移指令主要包括 4 种子类型，它们是条件、迭代、中断和无条件转移。表 3.12 给出了这些指令的一些例子。

表 3.12　x86 控制转移指令的举例

助记符	操作	子类型
SET	根据条件将字节设置为真或伪	条件
JS	若符号为真就跳转	条件
LOOP	若 CX 不等于 0 就循环	迭代
LOOPE	若 CX 不等于 0 且 ZF = 1 就循环	迭代
INT	中断	中断
IRET	中断返回	中断
JMP	无条件跳转	无条件转移
RET	从过程返回	无条件转移

在 x86 系列中的处理器控制指令主要包括 3 种子类型，它们是外同步、标志处理和通用控制指令。表 3.13 给出了这些指令的一些例子。

表 3.13 x86 处理器控制指令的举例

助记符	操作	子类型
HLT	停止	外同步
LOCK	锁定总线	外同步
CLC	清除进位标志	标志
CLI	清除中断标志	标志
STI	设置中断标志	标志
INVD	使数据高速缓存无效	通用控制

在介绍了 x86 处理器系列指令集的基本特征后,我们现在给出一些程序设计例子来说明如何使用该指令集。这里给出的例子与在第 2 章末给出的例子是相同的。

例 3.3 将起始地址为 1000、连续存放的 100 个数相加,并将结果存入存储单元 2000。LIST 定义为一个有 *N* 个元素的数组,每个元素的大小为 1 B。FLAG 是一个存储器变量,用来指明表是否已排序。寄存器 CX 用做 Loop 指令的计数器。Loop 指令将使 CX 寄存器减 1,且若结果不为 0 就发生转移。用来访问数组 List[BX + 1]的寻址方式称为基址寻址方式。注意,由于我们使用 BX 和 BX + 1,以值 999 装载 CX 计数器时就不会越表界[①]。

```
        MOV CX, 1000 - 1        ; Counter = CX ← (1000 - 1)
        MOV BX, Offset LIST     ; BX ← pointer to LIST
        CALL SORT
        .....
SORT PROC NEAR
Again:  MOV FLAG, 0             ; FLAG ← 0
Next:   MOV AL, [BX]
        CMP AL, [BX + 1]        ;Compare current and next values
        JLE Skip                ;Branch if current < next values
        XCHG AL, [BX + 1]       ;If not, Swap the contents of the
        MOV [BX + 1], AL        ;current location with the next one
        MOV FLAG, 1             ;Indicate the swap
Skip:   INC BX                  ; BX ← BX + 1
        LOOP Next               ;Go to next value
        CMP FLAG, 1             ;Was there any swap
        JE Again                ;If yes Repeat process
        RET
SORT ENDP
```

例 3.4 这里我们用 8086 指令集来实现 SEARCH 算法。LIST 定义为一个有 *N* 个元素的数组,每个元素的大小为一个字。FLAG 是一个存储器变量,用来指明表是否已排序。寄存器 CX 用做 Loop 指令的计数器。Loop 指令将使 CX 寄存器减 1,且若结果不为 0 就发生转移。用来访问数组 LIST[BX + 1]的寻址方式称为基址寻址方式。

```
        MOV CX, 1000        ; Counter = CX ← 1000
        MOV  BX,  Offset    ; BX ← pointer to LIST
        LIST
        MOV SI, 0           ; SI used as an index
        MOV AX, VAL         ; AX ← VAL
```

① 该例有错,原意是对 100 个元素求和,但后面说明是对元素进行排序,调用 SORT 例程——译者注。

```
CALL SEARCH
                        ; Test FLAG to check whether value found
.....
SEARCH PROC NEAR
                MOV FLAG, 0          ; FLAG ← 0
Next:           CMP AX, [BX + SI]    ;Compare current value to VAL
                JE Found             ;Branch if equal
                ADD SI, 2            ; SI ← SI + 2, next value
                LOOP Next            ;Go to next value
                JMP Not_Found
Found:          MOV FLAG, 1          ;Indicate value found
                MOV POSITION, SI     ;Return index of value in List
Not_Found:      RET
SEARCH ENDP
```

例 3.5　该例与例 3.4 相同,但使用 x86 的堆栈特性。

```
PUSH DS                     ;See Table 3.9
MOV CX, 1000                ;Counter = CX ← 1000
MOV BX, OFFSET LIST         ;Point to beginning of LIST
PUSH BX
PUSH VAL                    ;VAL is a word variable
CALL SEARCH
                            ;Test FLAG to check whether value found
                            ;If found get index from SI register
                             using
                            POP SI
.....
SEARCH PROC NEAR
POP TEMP                ;Save IP
POP AX                  ;AX ← VAL. Value to search for
POP SI                  ;SI ← OFFSET LIST and let BX = SI
                POP ES              ;Make ES = DS (See Table)
                CLD                 ;Set auto-increment mode
                REPNE SCASW         ;Scan LIST for value in AX if not
                                     found; increment SI by 2,
                                     decrement CX and if; not zero
                                     scan next location in LIST.
                                    ;If occurrence found Zero flag
                                     is set
                JNZ Not_Found       ;If value not branch to
                                     Not_Found?
                MOV FLAG, 1         ;Yes
                SUB SI, BX
                PUSH SI             ;Save position
Not_Found:      PUSH TEMP           ;Restore IP
                RET
SEARCH ENDP
```

注意,在上例中,当启动调用过程时,IP 寄存器是压入栈顶的最近值。所以必须小心,以免改变 IP 寄存器的值。为此,在过程入口处将栈顶内容保存到临时变量 TEMP,并在退出之前将其恢复。

3.6 小结

机器语言是以 0 和 1 表示的机器指令的集合。汇编语言提供了易于使用的符号表示,使用等价于机器语言的字母数字表示。汇编语言语句和机器指令间存在一一对应关系。汇编器是接受符号语言程序(源程序)并生成它的等价机器语言(目标程序)的程序。虽然与高级语言程序设计相比汇编语言程序设计较难,但学习汇编仍是很重要的。在某些应用中,若一个程序的一小部分被大量使用,就需要用汇编语言来编写。用汇编进行程序设计可以使所生成的机器代码比由高级语言编译器所生成的机器代码更小和更快。汇编程序员不得不使用所有目标机的硬件特性,而这对高级语言程序员来讲是不需要的。另外,学习汇编语言对理解计算机组成和体系结构底层细节有很大的帮助。在本章中我们对汇编语言和它的程序设计向读者提供了总体的评述。我们也从程序员的视角简略介绍了 Intel x86 微处理器系列的一些特性。然后举例展示了如何使用 x86 指令集编写类似于第 2 章中给出的那些举例程序。

习题

1. 以下每一对术语的差别是什么?
   - 编译器(编译程序)和汇编器;
   - 源代码和目标代码;
   - 助记符和十六进制表示;
   - 伪指令和指令;
   - 标号和地址;
   - 符号表和操作码表;
   - 程序计数器(PC)和指令定位计数器(ILC)。
2. 使用 3.1 节中简单处理器的汇编语言编写处理以下操作的汇编代码段:
   - 交换两个数;
   - 逻辑 OR(或);
   - 求反(negation)。
3. 将输入/输出指令加到 3.1 节中简单处理器的指令集中,并编写一个寻找 Fibonacci 级数的汇编程序。
4. 根据 3.2 节中给出的乘法汇编程序,写出它的机器语言代码。
5. 随着高级语言和编译器技术的快速发展,有些人认为汇编语言已不再重要。对这一观点给出某些支持和反对的论据。
6. 用 x86 系列的指令编写计算 $\sum_{i=1}^{200} X_i Y_i$ 的程序段,其中 X_i 和 Y_i 为带符号的 8 位数。假定不会发生溢出。
7. 用 x86 指令编写一个在不同代码段中可由主程序调用的子程序。该子程序将 AL 中一个带符号位的 8 位数与 CX 中一个带符号位的 16 位数相乘。主程序将调用这一子程

序,并将结果存放到 2 个连续的存储器字中,然后停止。假定 SI 和 DI 各自含有带符号位的 8 位数和 16 位数。

8. 用 x86 指令编写一个程序,比较长度为 100 字的源字符串和目的字符串。源字符串在 DS 中用 2000H 偏移量指定,而目的字符串在 DS 中用 4000H 偏移量指定。
9. 用 x86 指令编写生成 Fibonacci 级数前 10 个数的程序,即生成级数 1,1,2,3,5,8,13,21,34。
10. 用 x86 指令编写一个程序,将一个英文单词中的字母从大写转换到小写。假定该单词由存放在以单元 START 开始和以单元 FINISH 结束的连续存储单元中的 ASCII 字符所组成。

参考文献和推荐读物

C. M. Gilmore, *Microprocessors : Principles and Applications*, 2nd ed., McGraw-Hill, New York, 1996.

V. C. Hamacher, Z. G. Vranesic and S. G. Zaky, *Computer Organization*, 5th ed., McGraw-Hill, New York, 2002.

J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, New York, 1998.

V. Heuring, and H. Jordan, *Computer Systems Design and Architecture*, Addison Wesley, NJ, 1997.

K. R. Irvine, *Assembly Language for Intel-Based Computers*, 4th ed., Prentice Hall, NJ, 2003.

A. D. Patterson and J. L. Hennessy, *Computer Organization & Design; The Hardware/ Software Interface*, Morgan Kaufmann, San Mafeo, 1994.

W. Stallings, *Computer Organization and Architectures : Designing for Performance*, 4th ed., Prentice-Hall, NJ, U. S. A, 1996.

A. Tanenbaum, *Structured Computer Organization*, 4th ed., Prentice Hall, Paramus, NJ, U. S. A, 1999.

J. Uffenbeck, *The 80 × 86 Family, Design, Programming, and Interfacing*, 3rd ed., Prentice Hall, Essex, UK, 2002.

B. Wilkinson, *Computer Architecture : Design and Performance*, 2nd ed., Prentice-Hall, Hertfordshire, UK, 1996.

第4章　计算机算术运算

本章重点讨论计算机算术运算，向读者介绍关于算术运算操作的基本问题，以及支持计算的计算机电路。首先介绍数字系统，具体来讲，即介绍数字表示和基本变换。接下来讨论整数算术运算，这部分介绍用于实现整数加、减、乘和除等算术运算的算法和硬件电路。最后讨论浮点数的算术运算，具体来讲，介绍浮点数的表示、运算和硬件电路。IEEE 浮点数标准将在本章结尾讨论。

4.1　数字系统

一个数字系统采用一个特定的基数。在数字系统中，基数通常采用 2 的幂，包括二进制（以 2 为基数）、四进制（以 4 为基数）、八进制（以 8 为基数）和十六进制（以 16 为基数）。其中，二进制在计算机系统中占主导地位。

一个无符号整数 A 可以由以 b 为基数的 n 位数字表示：$A=(a_{n-1}a_{n-2}\cdots a_2a_1a_0)_b$。在表达式（称为位置表示）中，每个数字 a_i 取值为 $0\leqslant a_i\leqslant(b-1)$。采用位置表示方法，无符号整数 A 的十进制数值为 $A=\sum_{i=0}^{n-1}a_i\times b^i$。例如，十进制数 $A=106$ 的位置表示，若采用以 2 为基数的 8 位数表示，可以表示为 $A=0\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+1\times2^1+0\times2^0$。

采用 n 位数字，一个无符号整数 A 的最大值为 $A_{max}=b^n-1$。例如，以 2 为基数的 4 位数可以表示的最大无符号数是 $2^4-1=15$。在这个例子中，对应十进制数的范围是 0 到 15（对应于二进制从 0000 到 1111）。类似地，以 4 为基数的 4 位数可以表示的最大无符号数是 $4^4-1=255$。在这个例子中，可以表示的十进制数的取值范围是从 0 到 255（对应于四进制从 0000 到 3333）。

考虑采用 n 位数字表示以 b 为基数的实数 X 的情况，高位有效 k 位数表示其整数部分，而低位有效 m 位数则表示其小数部分，则 X 的值可以表示为

$$X=\sum_{i=-m}^{k-1}x_i\times b^i=x_{k-1}b^{k-1}+x_{k-2}b^{k-2}+\cdots+x_1b^1$$

$$+x_0b^0+x_{-1}b^{-1}+\cdots+x_{-m}b^{-m}$$

设想实数 $X=25.375$ 的表示法。采用二进制的表示方法如下（其中 $k=5,m=3$）：

$$X=\sum_{i=-3}^{4}x_i\times b^i=x_4\times2^4+x_3\times2^3+x_2\times2^2+x_1\times2^1$$

$$+x_0\times2^0+x_{-1}\times2^{-1}+x_{-2}\times2^{-2}+x_{-3}2^{-3}$$

其中，$x_4=1,x_3=1,x_2=0,x_1=0,x_0=1,x_{-1}=0,x_{-2}=1,x_{-3}=1$。

通常需要将一个数从一种进制转换为另一种进制，如从二进制转换为十进制。这种转换可以通过一些方法（算法）来实现。在这些算法中一个重要的工具是分解方法，分解方法的基本思想是将一个整数 a 通过另外一个整数 c 以及一个基数 b 加以表示，基本关系为 $a=c\times q+r$，其中，q 是商，r 是余数，$0\leqslant r\leqslant b-1$，$q=\lfloor a/c\rfloor$。基数之间的转换将在下面讨论。

4.1.1　基数转换算法

基数转换算法用于将基数 r_1 的数字表示转换为另一种基数 r_2 的数字表示。考虑数字 X 的整数部分 X_{int} 的转换。整数部分 X_{int} 可以表示为

$$X_{\text{int}} = \{[\cdots(x_{k-1}r_2 + x_{k-2})r_2 + \cdots + x_2)r_2 + x_1]\}r_2 + x_0$$

X_{int} 除以 r_2 得到商 $X_q = \{[\cdots(x_{k-1}r_2 + x_{k-2})r_2 + \cdots + x_2)r_2 + x_1]\}$，以及余数 $X_{\text{rem}} = x_0$。重复对商进行除运算，保留余数为需要的数字，直至商为零，则得到 X_{int} 在以 r_2 为基数时的对应表示。

同理，针对 X 的小数部分 X_{f} 重复乘以 r_2，保留获得的整数部分作为需要的数字，可以得到在以 r_2 为基数时对应小数部分的表示。值得注意的是，小数部分的转换不同于整数部分的转换，即通过有限的重复乘之后不一定能终止。因此，这个过程在执行一定的步数后会被终止，从而会引入一定程度上的近似。

例4.1　考虑十进制数 67.575 转换为二进制的情况。其中，$r_1 = 10$，$r_2 = 2$，$X_{\text{int}} = 67$，$X_{\text{f}} = 0.575$。对于整数部分 X_{int}，重复除以 2 得到如下的商和余数：

商	33	16	8	4	2	1	0
余数	1	1	0	0	0	0	1

因此在以 2 为基数的情况下的整数部分为 $X_{\text{int}} = (1000011)$。可以通过类似方法获得其小数部分的表示(通过重复乘)：

小数部分	0.150	0.300	0.600	0.200	0.400	0.800	0.600	0.200	…
进位位	1	0	0	1	0	0	1	1	…

小数部分 $X_{\text{f}} = (.10010011\cdots)$。因此，数 67.575 的二进制合成表示为(1000011.10010011…)。

4.1.2　负整数的表示

针对负整数的表示存在一些方法，包括符号 - 数值、补码、反码等，将在下面简要说明。

4.1.3　符号 - 数值

在这种表示方法中，(用于表示数的 n 位的)最高位用于表示数的符号，其中，最高位为“1”时表示是一个负数，而最高位为“0”时表示是一个正数，余下的 $(n-1)$ 位则用来表示数值。例如，采用符号 - 数值格式，以 2 为基数以及 6 位表示，负数(-18)可以表示为(110010)，而(+18)可以表示为(010010)。尽管表示简单，但符号 - 数值表示法在进行算术运算时却很复杂，特别是符号位需要同数值位分开加以处理。例如，数 +18(010010)和数 -19(110011)用符号 - 数值表示法进行加法运算，由于两数的符号不相同，计算结果的符号应采用绝对值大的数的符号，在这个例子中是(-19)的符号，余下的 5 位数进行减法运算(10011 - 10010)得(00001)，即结果是(-1)。

4.1.4　补码

在补码表示法中，正数的表示方法与符号 - 数值表示法相同，但负数的表示采用 b 的补码表示(针对以 b 为基数)。例如，采用 2 的补码表示数(-19)。在这个例子中，数 19 首先被表示为(010011)，接着对每位数字取反，得到(101100)，最后在最低位上加“1”得到(101101)。再来考

虑数(+18)的2补码表示,由于它是正数,因此与符号-数值的表示法相同,可以表示为(010010)。现在考虑这两个数的相加,在这种情况下,直接对相应的位数进行相加,而不用对符号位做专门的处理,两数相加的结果为(111111),这正是(-1)的2补码的表示。2补码表示法的主要优点在于不需要针对数的符号做专门的处理;其另一个特点是执行算术运算时最高位的进位可以忽略,而不影响计算结果的正确性。例如,-19(101101)和+26(011010)相加,结果是(1)(000111),在进位忽略后,正确的计算结果是(+7)。

4.1.5 反码

这种表示法与补码表示法相类似,需要针对数的各位取补,不同之处在于,不需要在取反后的最低位上加"1"。根据这种数字系统表示方法,(-19)可以表示为(101100),而(+18)可以表示为(010010)。若对这两个数进行相加,则得到(111110),这即是(-1)的1反码。反码表示法的主要缺点是执行算术运算时无论最高位是否有进位,均需要修正因子。例如,-3(111100)和+18(010010)相加得到(1)(001110),若将进位加到结果的最低位上,则得到(001111),即是正确的计算结果(+15)。

表4.1对比了8位数 x 的2补码和1补码的表示。

表4.1 一个8位数的2补码和1补码的表示

数	表示	例子
2补码		
$x=0$	0	0(00000000)
$0<x<256$	x	77(01001101)
$-128\leqslant x<0$	$256-\lvert x\rvert$	-56(11001000)
1补码		
$x=0$	0或255	(11111111)
$0<x<256$	x	77(01001101)
$-127\leqslant x<0$	$255-\lvert x\rvert$	-56(11000111)

4.2 整数运算

这部分将介绍利用补码表示法进行整数算术运算的相关技术。以下讨论主要是针对以2为基数的二进制表示法。

4.2.1 2补码表示

为了以2补码的形式表示一个数,需要执行以下两步:

1. 针对每位执行布尔反(包括符号位)。
2. 对最低位加1(将其作为无符号二进制整数),即 $-A=\bar{A}+1$。

例4.2 考虑以2的补码表示(-22)。

$$
\begin{array}{r}
22 = 00010110 \\
\Downarrow \\
11101001\ (1\text{补码}) \\
+\qquad 1 \\
\hline
11101010\ (-22)
\end{array}
$$

4.2.2　2 补码的算术运算

加。以 2 补码表示的两个 n 位数加法可以通过 n 位加法器实现，只要加法结果在 -2^{n-1} 至 $+2^{n-1}-1$ 之间，进位可以忽略，且不影响结果的正确性。

例 4.3　考虑两个以 2 补码表示的数 (-7) 和数 $(+4)$ 的加法。相加为 $(-7)+(+4)=-3$，即 $1001+(0100)=1101$，即是以 2 补码表示的 (-3)。

结果在 -2^{n-1} 至 $+2^{n-1}-1$ 之间的这个条件非常重要，因为结果超出这个范围，将会导致溢出，因此是一个错误的结果。简而言之，若给定运算的计算结果超出可表示数字的范围，则会发生溢出。考虑下面的两个例子。

例 4.4　考虑以 2 补码表示的数 $(+7)$ 和数 $(+6)$ 相加的情况。相加过程为 $(+7)+(+6)=+13$，即 $0111+(0110)=1101$，这是一个错误的结果，这是因为计算结果超过了最大值 $(+7)$。

例 4.5　考虑以 2 补码表示的数 (-7) 和数 (-4) 相加的情况。相加过程为 $(-7)+(-4)=-11$，即 $1001+(1100)=0101$，这也是一个错误的结果，这是因为计算结果小于了最小值 (-8)。请注意原始的两个数是负数，但结果却是正数。

从这两个例子可以看出：当两个数（同为正或同为负）相加时，当且仅当结果的符号与被加数的相反时，溢出才可能被检测出来。

减。在 2 补码中，减法执行过程与加法相同。例如，$B-A=B+\overline{A}+1$，即从 B 中减 A 等同于将 A 的补码加到 B 中。

例 4.6　考虑减法 $2-7=-5$。这可以如下执行：$2+\overline{7}+1=0010+1000+0001=1011(-5)$。

值得注意的是，前面加法情况下的溢出现象同样适用于减法，这是因为减法等同于补码的加法。考虑以下实例。

例 4.7　考虑减法 $7-(-7)=14$。这可以如下执行：$7+\overline{7}+1=0111+1000+0001=(1)0000$，这是一个错误的结果（结果 >7）。

符号数加减法的硬件结构。2 个 n 位数 A 和 B 相加需要一个基本的硬件电路，以接收 3 个输入，即 a_i，b_i 和 c_{i-1}。这 3 位分别表示数 A 和数 B 当前的位（位置 i），以及来自前一位（位置 $i-1$）的进位。电路需要产生两个输出，即 s_i 和 c_i。根据下面的真值表，它们分别表示和与进位。

a_i	0	0	0	0	1	1	1	1
b_i	0	0	1	1	0	0	1	1
c_{i-1}	0	1	0	1	0	1	0	1
s_i	0	1	1	0	1	0	0	1
c_i	0	0	0	1	0	1	1	1

输出逻辑函数是 $s_i=a_i \oplus b_i \oplus c_{i-1}$，以及 $c_i=a_ib_i+a_ic_{i-1}+b_ic_{i-1}$。用于实现这两个函数的电路称为全加器（FA），如图 4.1 所示。

两个 n 位的数 A 和 B 相加可以通过 n 个连续的 FA 完成，这种排列方式称为行波进位通过加法器（CRT，carry-ripple through adder），如图 4.2 所示。

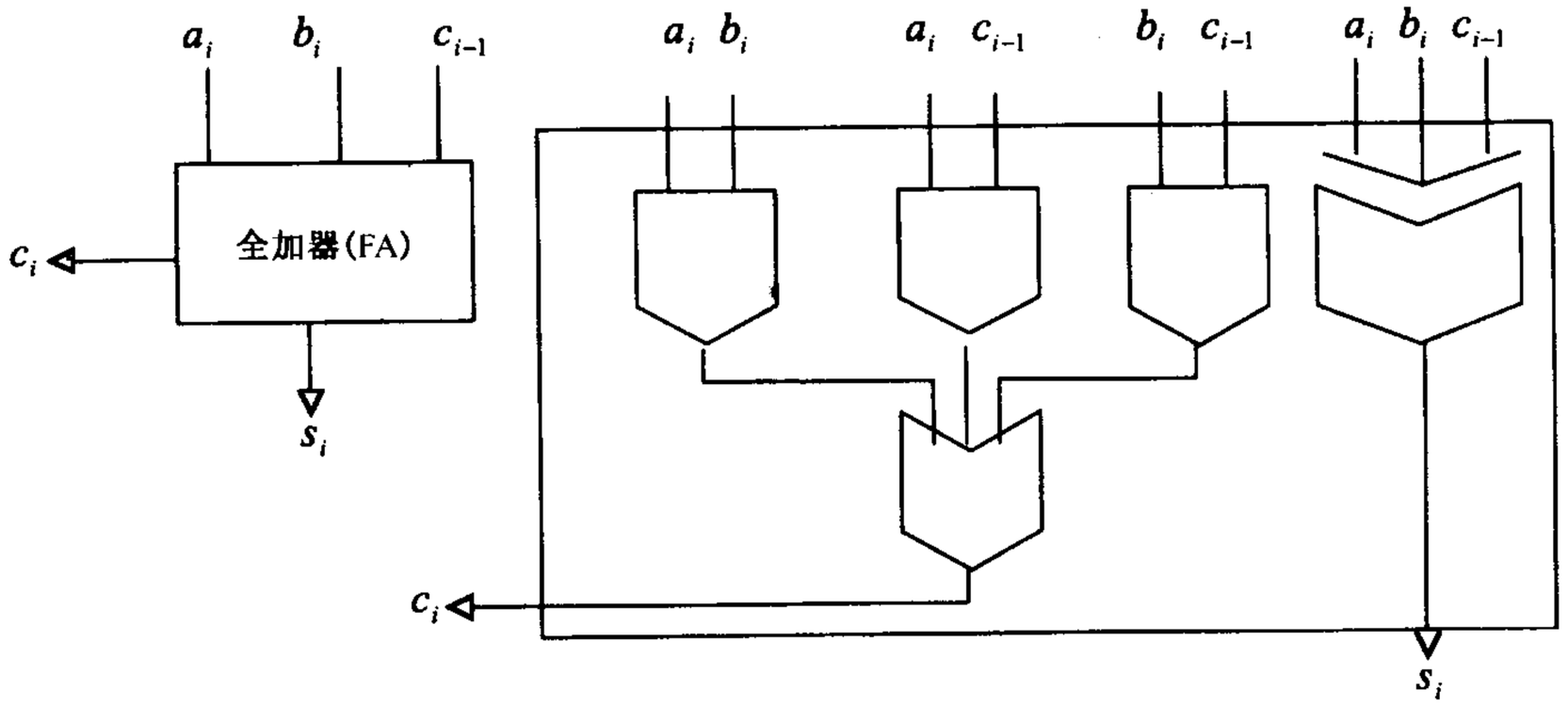


图 4.1 全加器(FA)电路

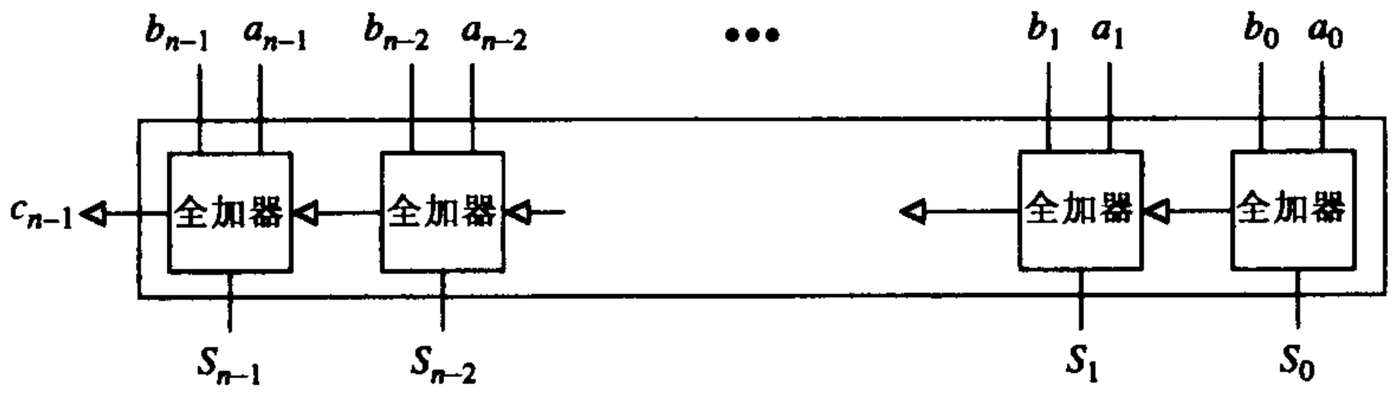


图 4.2 n 位行波进位通过加法器(CRT)

图 4.2 所示的 n 位 CRT 加法器可用于执行以 2 补码表示的数 A 和数 B 的加法，其中，b_{n-1} 和 a_{n-1} 表示符号位。通过关系式 $B - A = B + \bar{A} + 1$，同样的电路可以用于执行减法，二进制加/减法的逻辑网络结构如图 4.3 所示。

在该图中，2 个输入 A 和 B 表示用于加法或减法的参数，控制输入用于确定执行加法或减法操作，当控制输入为 0 时则执行加法操作，而当控制输入为 1 时则执行减法操作。一个实现图 4.3 所示加/减模块的 4 位输入简单电路如图 4.4 所示。

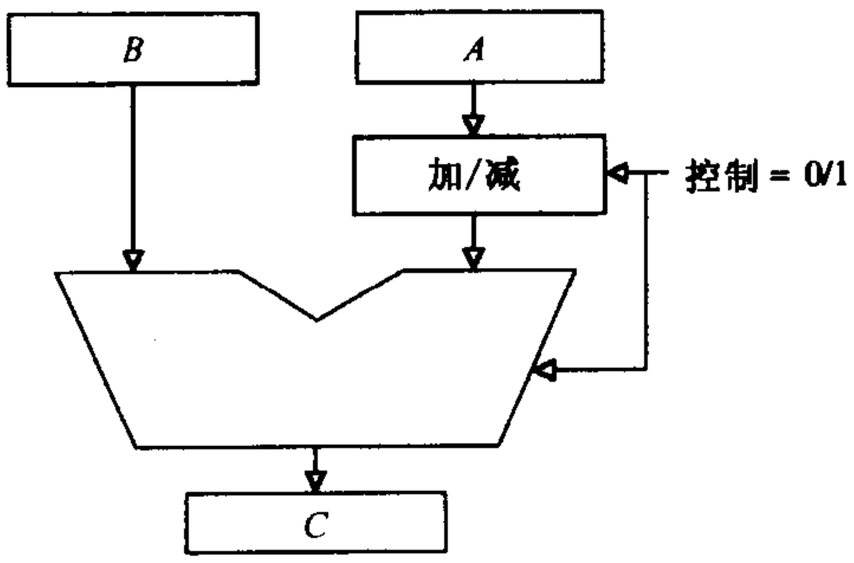


图 4.3 加/减法的逻辑网络结构

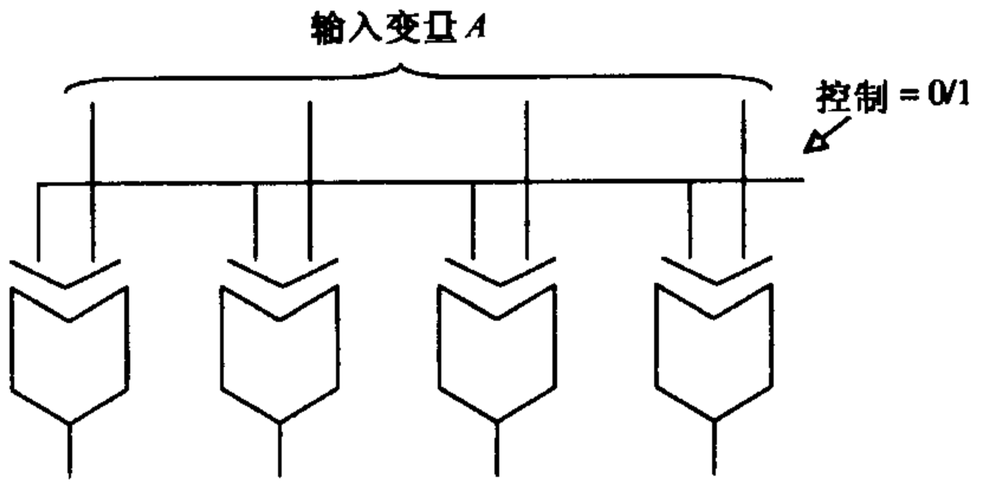


图 4.4 加/减电路

CRT 电路的一个主要缺点是输入提交到电路的时间和最终输出获得的时间之间的期望延迟较长，这是因为每一级运算与前一级的进位输出相关。这种相关链使得 CRT 加法器的延迟是 $O(n)$，其中 n 是加法器的级数。为了加速加法过程，有必要引入快速加法电路，在其中加法级之间的相关链被打破。参考文献中介绍了一些快速加法电路，其中先行进位加法器(CLA)尤为著名。下面介绍 CLA 加法器。

考虑 CRT 加法器电路。两个逻辑函数是 $s_i = a_i \oplus b_i \oplus c_{i-1}$ 和 $c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1}$。这两个函数可以用两个新的子函数进行重写，进位生成 $G_i = a_i b_i$ 和进位传递 $P_i = a_i \oplus b_i$。利用这两个新的子函数，可以重写任意级上进位输出的逻辑方程为 $c_i = G_i + P_i c_{i-1}$。现在可以写出不同级上一系列的进位输出如下：

$$
\begin{aligned}
c_0 &= G_0 + P_0 c_{-1} \\
c_1 &= G_1 + P_1 c_0 = G_1 + P_1(G_0 + P_0 c_{-1}) = G_1 + G_0 P_1 + P_0 P_1 c_{-1} \\
c_2 &= G_2 + P_2 c_1 = G_2 + P_2(G_1 + G_0 P_1 + P_0 P_1 c_{-1}) \\
&= G_2 + G_0 P_1 P_2 + G_1 P_2 + P_0 P_1 P_2 c_{-1} \\
c_3 &= G_3 + P_3 c_2 = G_3 + P_3(G_2 + G_1 P_2 + G_0 P_1 P_2 + P_0 P_1 P_2 c_{-1}) \\
&= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + P_0 P_1 P_2 P_3 c_{-1} \\
&\vdots
\end{aligned}
$$

进位输出序列表明不同进位之间完全独立。图 4.5 表示 4 位 CLA 加法器的完整体系结构。在 1 个 CLA 中有 3 个基本部分，第一部分用于产生 G_i 和 P_i，第二部分用于生成所有的进位，第三部分用于生成所有和输出。不管在 CLA 中有多少位，第一部分的延迟等于一个一级门电路延迟，第二部分的延迟等于一个两级门电路延迟，第三部分的延迟等于一个一级门电路延迟。在图 4.5 中显示了一些进位以及和输出的生成，希望读者能够补全设计(见本章的习题)。

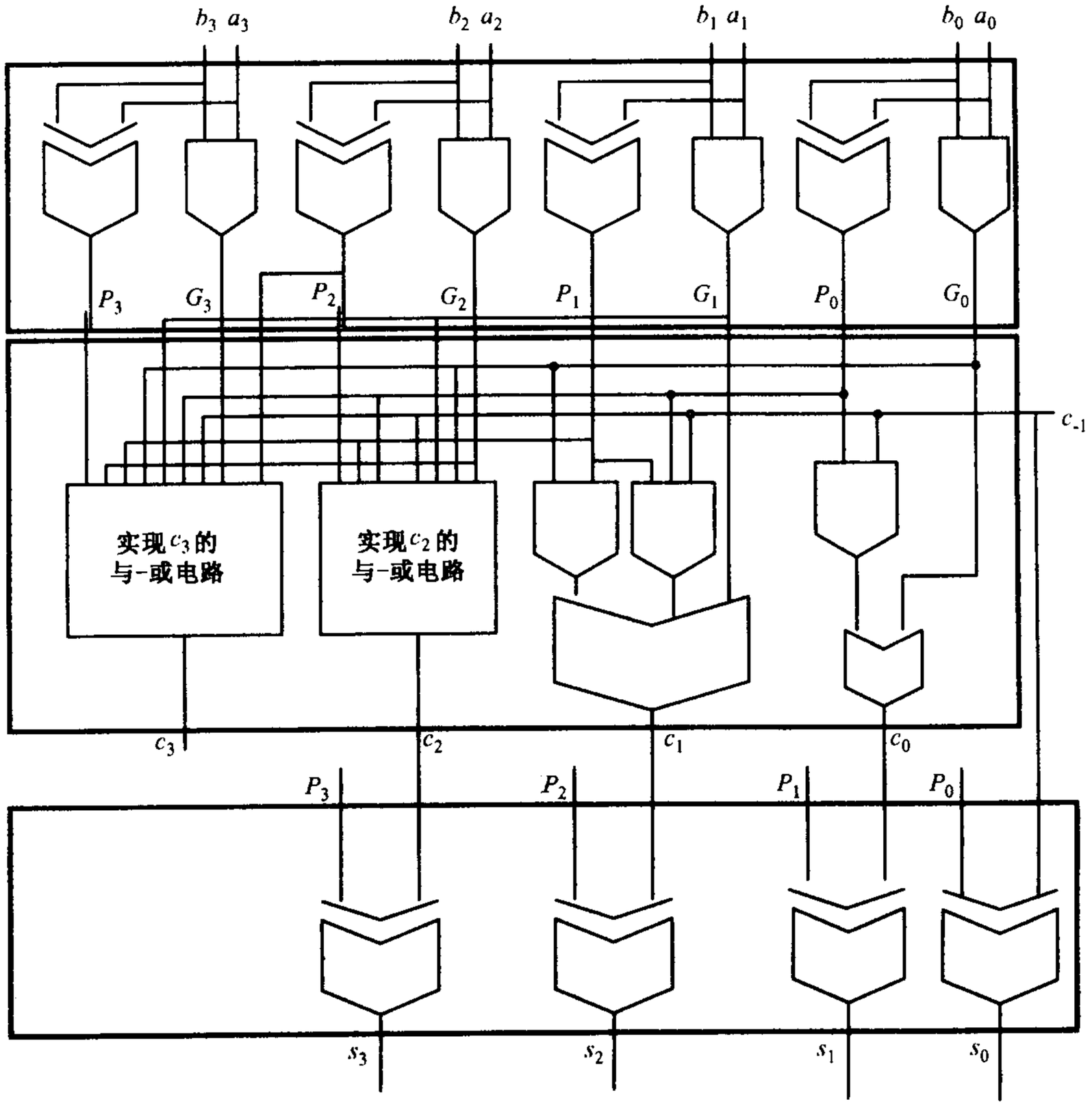


图 4.5　一个 4 位 CLA 加法器结构

乘。在讨论乘法时，首先设定两个参数形式为乘数 $Q = q_{n-1}\,q_{n-2}\cdots q_1\,q_0$ 和被乘数 $M = m_{n-1}\,m_{m-2}\cdots m_1\,m_0$。已有一些方法用于执行乘操作，下面讨论其中某些方法。

纸和铅笔方法(无符号数)。这是两个无符号数乘法的最简单方法，可以通过下面的例子加以说明。

例 4.8 考虑两个无符号数(14)和(10)的乘法，这两个数的二进制表示的乘法过程如下所示：

```
1110(14)        被乘数(M)
1010(10)        乘数(Q)
--------
  0000          (部分积)
 1110           (部分积)
0000            (部分积)
1110            (部分积)
= = = = = = =
10001100(140)最终结果(P)
```

上面的乘法可以通过由 FA 和 AND 组成的单元阵列完成，每个单元计算指定部分的乘积。图 4.6 表示一个乘法器阵列的基本单元和一个 4 位数 × 4 位数排列示例。

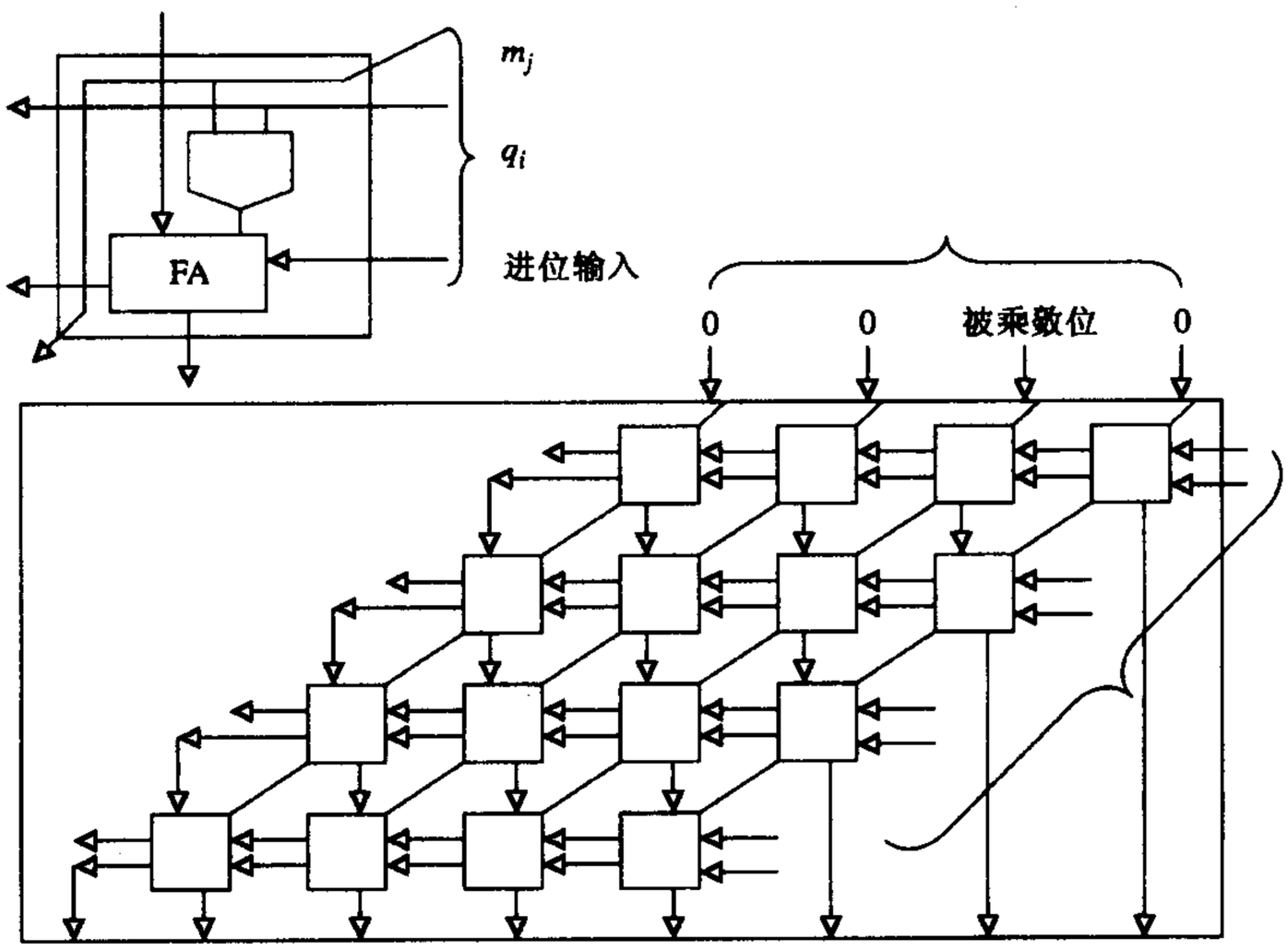


图 4.6 无符号二进制数的阵列乘法器

这种方法的特征是需要将这 n 个部分的乘积进行求和，而不用考虑这个乘数位的值。值得注意的是，若乘数的给定位为 0，则无需计算相应部分的乘积。下面的方法即是基于这个观察。

加 - 移位法。在这种情况下，乘法由一系列的(n)条件加和移位操作完成，若乘数给定位是 0，则只有移位操作；若乘数给定位是 1，则针对这部分乘积执行求和操作和移位操作。该方法如下例所示。

例 4.9 考虑两个无符号数 11 和 13 的乘法。这个过程如下表所示。其中，A 是一个 4 位的寄存器，并初始化为 0，C 是最高有效位的进位。这个过程重复 $n = 4$(乘数 Q 的位数)次。若乘

数位是“1”,则 $A \leftarrow A + M$ 且 AQ 串接向右移动一位。另一方面,若该位是“0”,仅针对 AQ 执行一个移位操作。用于执行这种操作的结构如图 4.7 所示,在该图中,控制逻辑用于根据 Q 中的最低有效位确定执行的操作。一个 n 位加法器用于对寄存器 A 和 M 的内容进行求和。

M	*C*	*A*	*Q*		
1011	0	0000	1101	初始值	
1011	0	1011	1101	加	第一个周期
1011	0	0101	1110	移位	
1011	0	0010	1111	移位	第二个周期
1011	0	1101	1111	加	第三个周期
1011	0	0110	1111	移位	
1011	1	0001	1111	加	第四个周期
1011	0	1000	1111	移位	

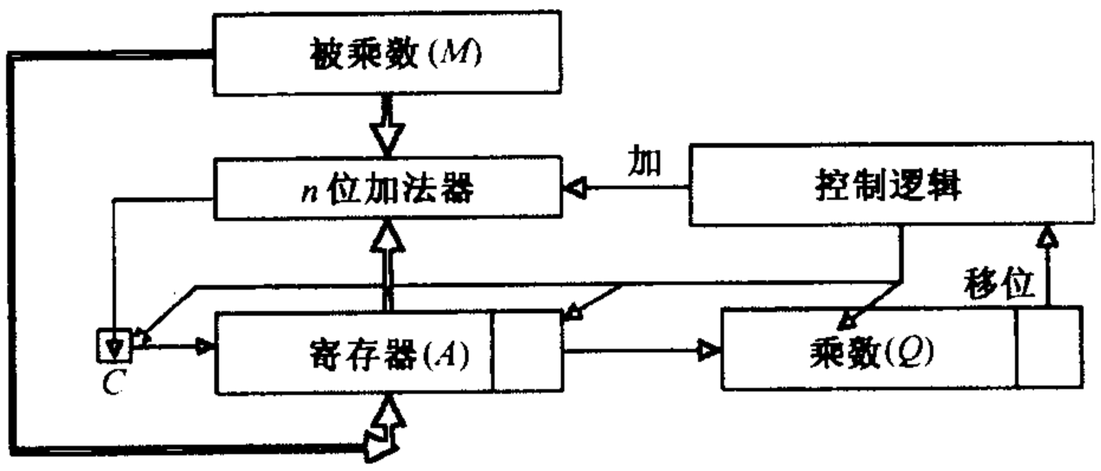


图 4.7　加 – 移位乘法的硬件结构

为了加速乘法操作,我们采用了一些其他技术。这些技术基于这样的现象:连续的 0 和 1 数量越多,则产生的部分积越少。乘数中一组连续 0 不需要产生新的部分积,乘数中一组 k 个连续 1 产生的部分积少于 k。Booth 算法即是利用这个现象。下面讨论 2 位 Booth 算法。

Booth 算法。同时检测乘数的两位 $Q(i)Q(i-1)$,$0 \leqslant i \leqslant n-1$,根据这两位的二进制值确定其行为,即若这两个值是 01,则 $A \leftarrow A + M$;若这两个值是 10,则 $A \leftarrow A - M$;若值是 00 和 11,则无操作。在所有 4 种情况中,对 AQ 串接执行算术右移操作。整个过程重复 n 次(n 是乘数的位数)。Booth 算法要求在乘过程的初始阶段,位 $Q(-1)=0$ 作为乘数 Q 的最低有效位。该算法如图 4.8 所示。

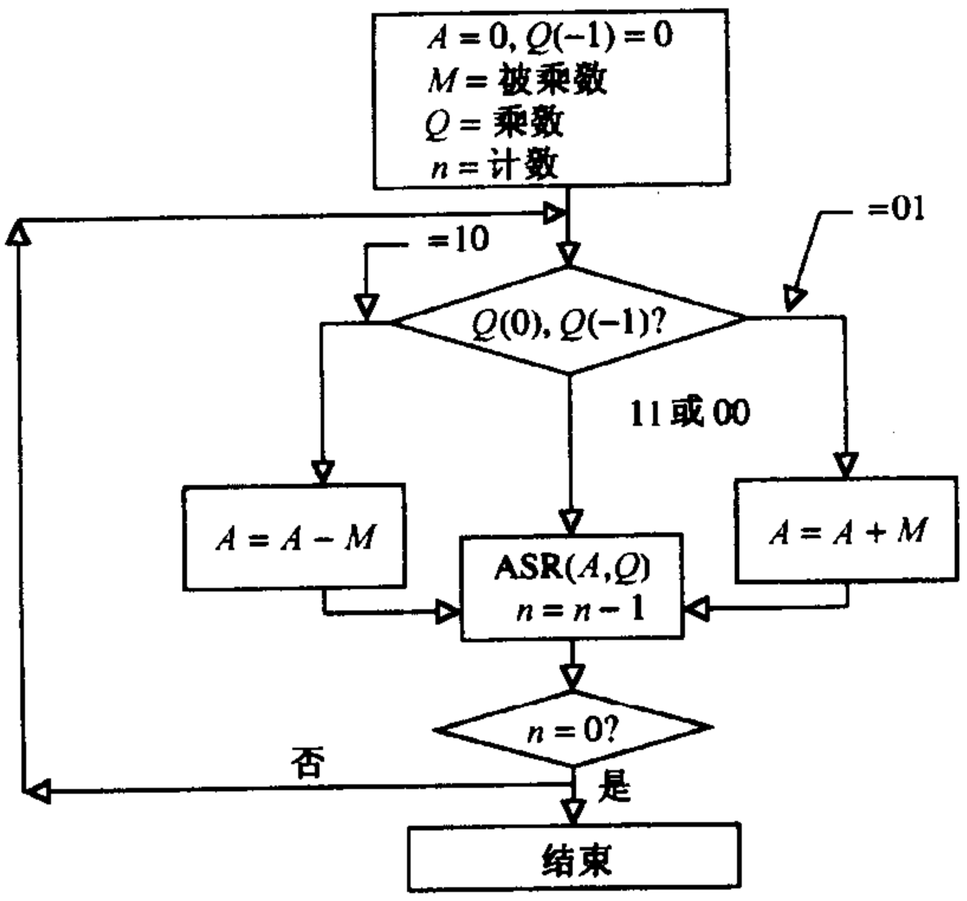


图 4.8　Booth 算法

下面的例子说明 Booth 算法的步骤。

例 4.10 考虑两个正数 $M=0111(7)$和 $Q=0011(3)$的乘法,设定 $n=4$,步骤列表如下:

M	A	Q	$Q(-1)$		
0111	0000	0011	0	初始值	
0111	1001	0011	0	$A=A-M$	
0111	1100	1001	1	ASR	结束循环 1
0111	1110	0100	1	ASR	结束循环 2
0111	0101	0100	1	$A=A+M$	
0111	0010	1010	0	ASR	结束循环 3
0111	0001	0101	1	ASR	结束循环 4

+21(正确的结果)

例 4.11 考虑两个数 $M=0111(7)$和 $Q=1011(-3)$的乘法,设定 $n=4$,步骤列表如下:

M	A	Q	$Q(-1)$		
0111	0000	1101	0	初始值	
0111	1001	1101	0	$A=A-M$	
0111	1100	1110	1	ASR	结束循环 1
0111	0011	1110	1	$A=A+M$	
0111	0001	1111	0	ASR	结束循环 2
0111	1010	1111	0	$A=A-M$	
0111	1101	0111	1	ASR	结束循环 3
0111	1110	1011	1	ASR	结束循环 4

−21(正确的结果)

图 4.9 所示的硬件结构可以用于执行 Booth 算法所需的操作,它由一个 ALU 组成,该 ALU 根据 $Q(0)Q(-1)$两位执行加/减操作。一个控制电路用来执行 $ASR(AQ)$,并发出合适信号用于控制循环次数。

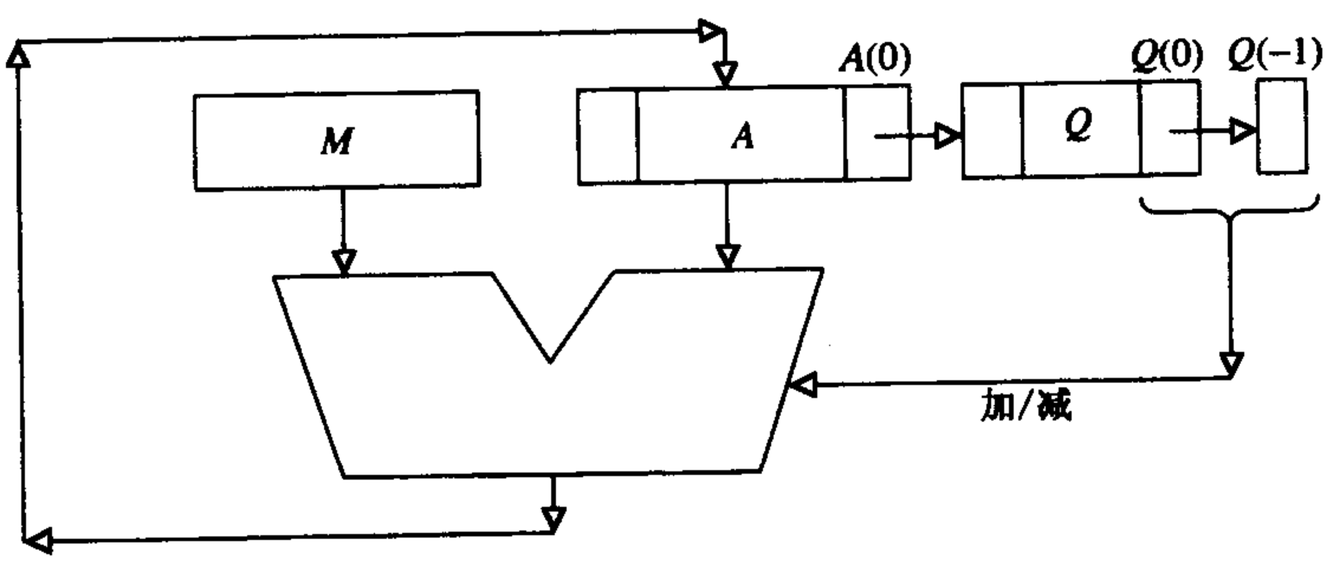


图 4.9 实现 Booth 算法的硬件结构

Booth 算法的主要缺点是，当 Q 中位模式是重复地在 0(1)后为 1(0)的情况下，加/减操作次数的易变性以及算法的低效率。若同时检测 3 个位，而不是 2 个位，可以改进后一种情况。

除。在 4 种基本算术操作中，除被认为是最复杂、同时也是最费时的。在最简单的情况下，整数除操作需要两个参数，被除数 X 和除数 D，产生两个输出，即商 Q 和余数 R。这 4 个量满足关系 $X = Q \times D + R$，其中 $R < D$。处理除法时会导致许多复杂性，最显而易见的是 $D = 0$。另外一个棘手问题是要求得到的商不得超过其所用的寄存器容量。若 $Q < 2^{n-1}$，则可以满足这一要求，其中 n 是商所在寄存器的位数。这也意味着关系 $X < 2^{n-1} D$ 需得到满足。上述条件中任意一条不能满足将会导致溢出情况的发生。

开始我们说明除算法时，将假设包括被除数、除数、商和余数在内所有数值都是分数，但如后面所示，这个过程同样适用于整数数值情况。

除操作通过重复一系列减和移位以获得正分数商 $Q = 0q_1 q_2 \cdots q_{n-1}$。在每一步中，比较余数和除数 D，若余数大，则商位设为 1，否则商位设为 0。可以通过下面的方程表示这一过程：$r_i = 2r_{i-1} - q_i \times D$，其中 r_i 和 r_{i-1} 分别是当前的商以及前一步的商，$r_0 = X$，$i = 1, 2, \cdots, n-1$。

例 4.12　考虑以下除法，其中，被除数 $X = 0.5 = (0.100000)$。除数 $D = 0.75 = (0.1100)$。这个过程如下表所示。

$r_0 = X$			0	∘	1	0	0	0	0	0	初始值
$2r_0$		0	1	∘	0	0	0	0	0		设 $q_1 = 1$
$-D$	+	1	1	∘	0	1	0				
$r_1 = 2r_0 - D$		0	0	∘	0	1	0	0	0		
$2r_1$		0	0	∘	1	0	0	0			设 $q_2 = 0$
$r_2 = 2r_1$		0	0	∘	1	0	0	0			
$2r_2$		0	1	∘	0	0	0				设 $q_3 = 1$
$-D$	+	1	1	∘	0	1	0				
$r_3 = 2r_2 - D$		0	0	∘	0	1	0				

得到商 $Q = (0.101) = (5/8)$，余数 $R = (1/32)$。由于 $X = QD + R = (5/8)(3/4) + 1/32 = 1/2$，因此这些值是正确的。

现在来证明上述过程对于整数值的正确性。在这种情况下，方程 $X = QD + R$ 可以重写为 $2^{2n-2} X_f = 2^{n-1} Q_f \times 2^{n-1} D_f + 2^{n-1} R_f$，得到 $X_f = Q_f \times D_f + 2^{-(n-1)} R_f$，其中，$X_f$，$D_f$，$Q_f$ 和 R_f 均是分数。为清晰起见，下面给出一个示例。

例 4.13　考虑被除数 $X = 32 = (0100000)$ 和除数 $D = 6 = (0110)$ 的除法。过程如下表所示。

$r_0 = X$			0	1	0	0	0	0	0	初始值
$2r_0$		0	1	0	0	0	0	0		设 $q_1 = 1$
$-D$	+	1	1	0	1	0				
$r_1 = 2r_0 - D$		0	0	0	1	0	0	0		
$2r_1$		0	0	1	0	0	0			设 $q_2 = 0$
$r_2 = 2r_1$		0	0	1	0	0	0			
$2r_2$		0	1	0	0	0				设 $q_3 = 1$
$-D$	+	1	1	0	1	0				
$r_3 = 2r_2 - D$		0	0	0	1	0				
$2r_3$		0	0	1	0					设 $q_4 = 0$

其商 $Q = 0101 = (5)$，余数 $R = 0010 = (2)$。这些都是正确的值。

二进制除法的硬件结构如图 4.10 所示。在图中,除数(D)和寄存器 A 的值利用($n+1$)位加法器进行求和。一个控制逻辑用于执行必要的左移操作(见习题)。

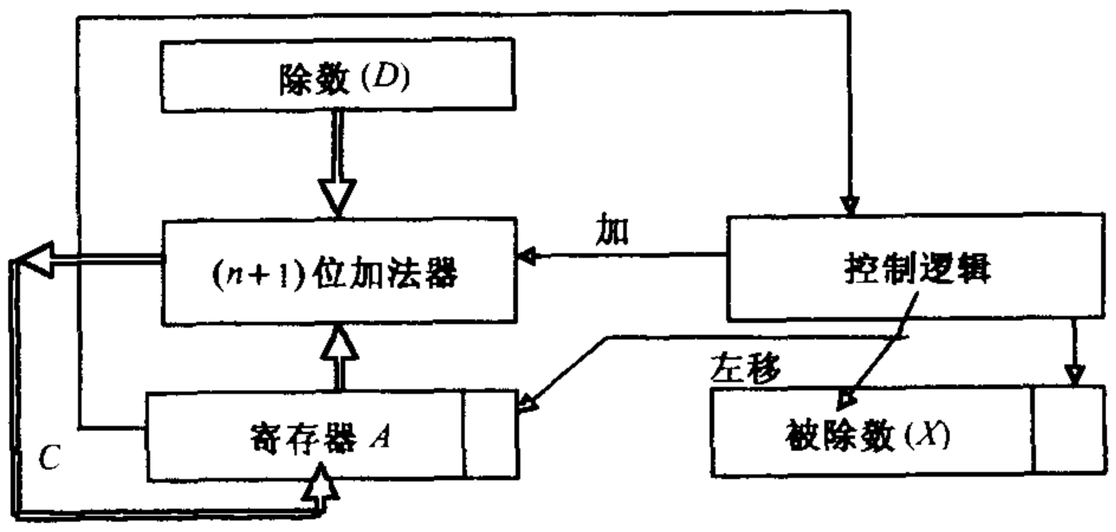


图 4.10 二进制除法结构

对余数和除数进行大小比较被认为是除法过程中最困难的一步。上述执行这个比较的方法是从 $2r_{i-1}$ 中减去 D,若结果是负数,则设 $q_i=0$,这个操作需要通过加回已减去的值来复原原有的值(复位除法)。

替代方法是采用非复位除法:

步骤 1: 执行下面的步骤 n 次

1. 若 A 的符号是 0,则左移 AQ,并从 A 中减去 D;否则左移 AQ,将 D 加到 A 上。
2. 若 A 的符号是 0,设 $q_0=1$,否则设 $q_0=0$。

步骤 2: 若 A 的符号是 1,则将 D 加到 A 上。

例 4.14 考虑采用非复位算法的被除数为 $X=8=(1000)$和除数为 $D=3=(0011)$的除法。过程如下表所示。

初始值	0	0	0	0	0	1	0	0	0	
	0	0	0	1	1					
移位	0	0	0	0	1	0	0	0		第一次循环
减	1	1	1	0	1					
设置 x_0	1	1	1	0		0	0	0	**0**	
移位	1	1	1	0	0	0	0	0		第二次循环
加	0	0	0	1	1					
设置 x_0	1	1	1	1	1	0	0	0	**0**	
移位	1	1	1	1	0	0	0	0		第三次循环
加	0	0	0	1	1					
设置 x_0	0	0	0	0	1	0	0	0	**1**	
移位	0	0	0	1	0	0	0	1		第四次循环
减	1	1	1	0	1					
设置 x_0	1	1	1	1	1	0	0	1	**0**	
							商			
加	1	1	1	1	1					
	0	0	0	1	1					复原余数
	0	0	0	1	0					
				余数						

4.3　浮点运算

在讨论了整数表示和运算后，我们将在本节介绍浮点表示和运算。

4.3.1　浮点表示

浮点(FP)数可以以下面的形式表示：$\pm m \times b^e$，其中 m 称之为尾数，表示这个数的小数部分，通常由带符号的二进制小数表示，e 表示指数，b 表示指数的基数。

例 4.15　图 4.11 表示一个浮点数，其中 $m = 23$ 位，$e = 8$ 位，S(符号位) = 1。若 S 中的值为 0，则该数是正数，若 S 中的值是 1，则该数是负数。

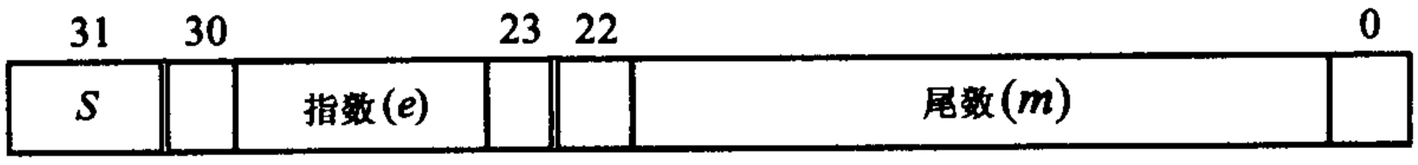


图 4.11　浮点数的表示

上例中的指数表示正数范围是 0 至 255，为了表示正的和负的指数，一个固定值，即偏移量值，需要从指数字段中减去以获得真实的指数。在上例中设定偏移量为 128，则可以表示的指数范围是 - 128(指数字段为 0)至 + 127(指数字段为 255)。基于这种表示，指数 + 4 是通过在指数字段中存储 132 表示的，而指数 - 12 则是在指数字段中存储 116 表示的。

设定 $b = 2$，则浮点数 1.75 可以以图 4.12 中的任一形式表示。

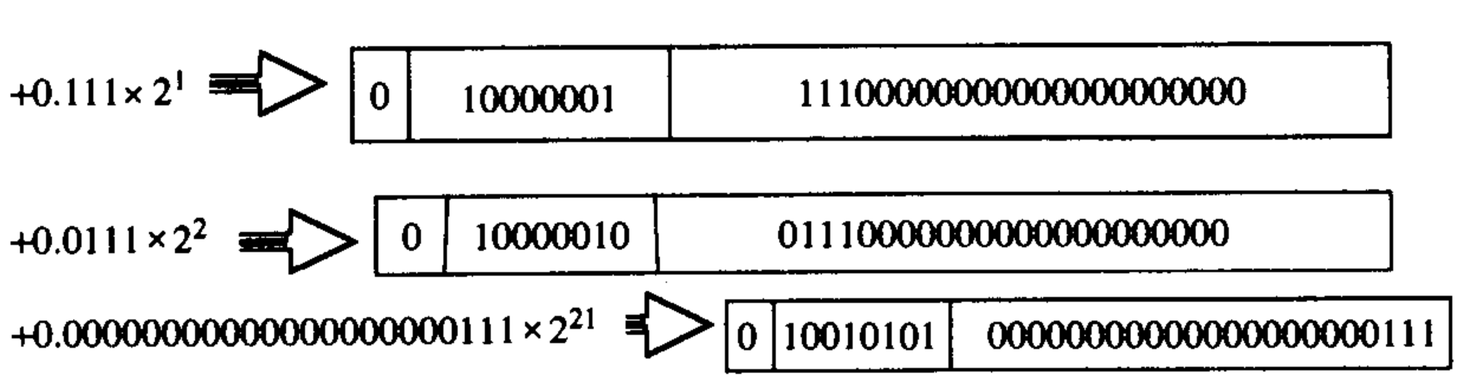


图 4.12　一个浮点数的不同表示

为了简化浮点数的操作并增加其精度，它们总以规格化形式表示。若尾数的最左边位是 1，则称该浮点数是规格的。因此，在 1.75 的上述 3 种可能表示中，第 1 种表示是规格的，我们予以采用。

由于在规格化浮点数中最高有效位(MSB)总是 1，因此这一位常不被存储，被认为是小数点左边的隐含位，即存储的尾数是 1.m。因此非零规格化数可以表示的值为 $(-1)^s \times (1.m) \times 2^{e-128}$。

浮点加/减运算。对两个浮点数相加的困难在于它们可能具有不同的指数，因此在相加之前，必须使它们的指数相等，即必须校正(对齐)具有较小指数值的尾数。

两个浮点数加/减的步骤

1. 比较两个数的指数量级，校正指数量级较小的数。
2. 执行加/减。
3. 规格化操作，即对计算结果的尾数进行移位，并调整计算结果的指数。

例 4.16 考虑两个浮点数 1.1100×2^4 和 1.1000×2^2 相加。

1. 校正:1.1100×2^2 被校正为 0.0110×2^4。
2. 相加:两数相加得到 10.0010×2^4。
3. 规格化:最终规格化的结果是 0.1000×2^6(设定小数点后保留 4 位)。

两个浮点数加/减的示意图如图 4.13 所示。

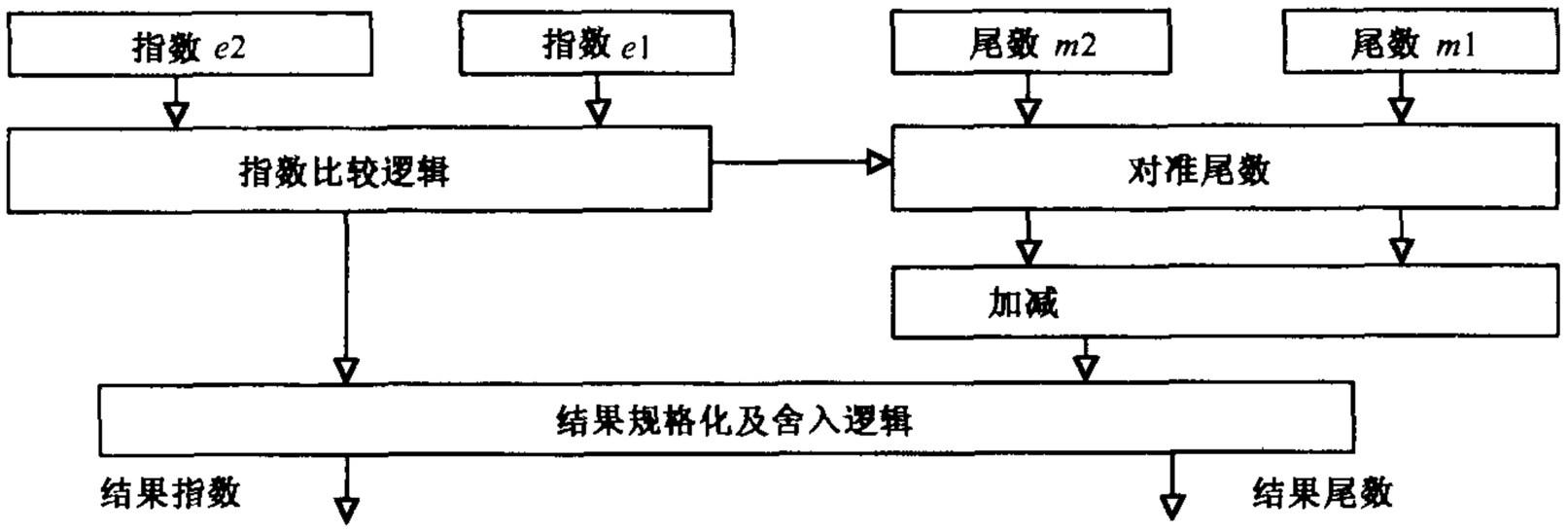


图 4.13 浮点数的加/减

乘。一对浮点数 $X = m_x \times 2^a$ 和 $Y = m_y \times 2^b$ 的乘法可以表示为 $X \times Y = (m_x \times m_y) \times 2^{a+b}$。浮点数乘法的通用算法由以下 3 个基本步骤组成,它们是

1. 计算乘积的指数部分,即指数相加。
2. 求两个尾数的乘积。
3. 规格化和舍入最终的乘积。

例 4.17 考虑两个浮点数 $X = 1.000 \times 2^{-2}$ 和 $Y = -1.010 \times 2^{-1}$ 的乘法。

1. 指数相加:$-2 + (-1) = -3$。
2. 尾数相乘:$1.000 \times -1.010 = -1.010000$。

乘积是 -1.0100×2^{-3}。

两个浮点数相乘的示意图如图 4.14 所示。

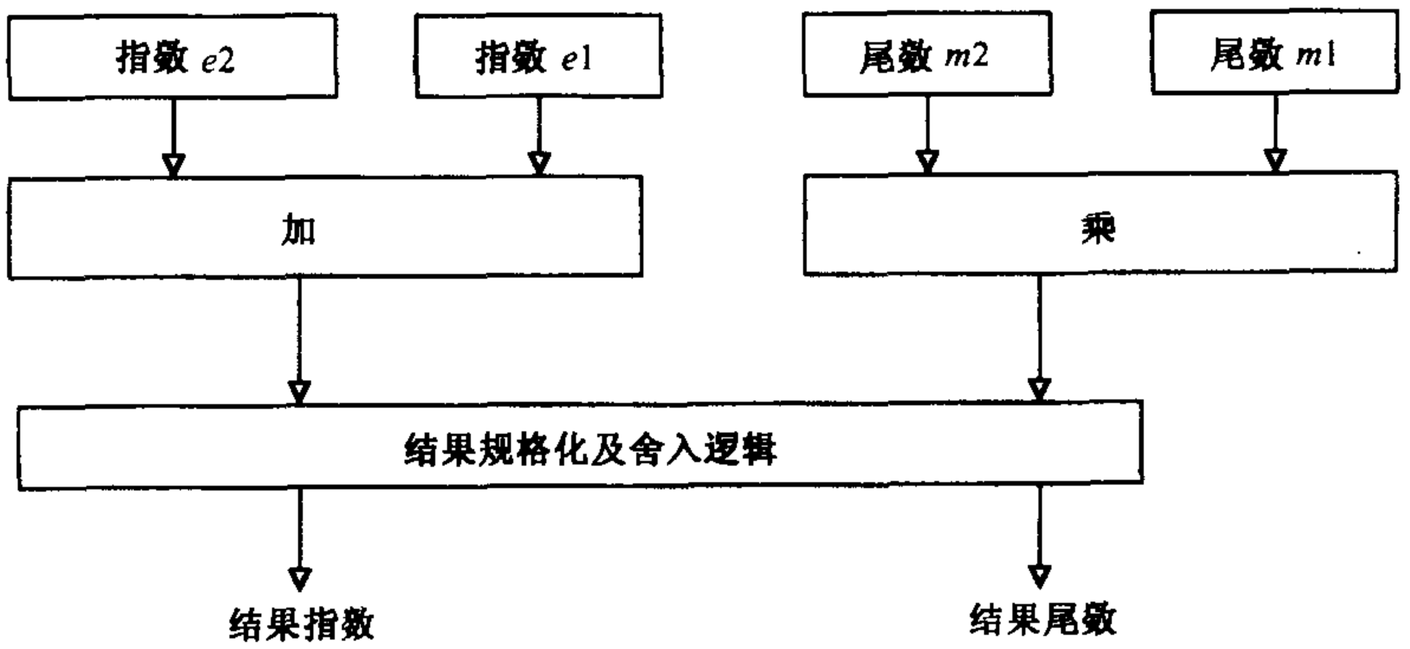


图 4.14 浮点数的乘

除。一对浮点数 $X = m_x \times 2^a$ 和 $Y = m_y \times 2^b$ 的除法可以表示为 $X/Y = (m_x/m_y) \times 2^{a-b}$。浮点数除法通用算法由以下 3 个基本步骤组成:

1. 通过指数相减,得到结果的指数部分。
2. 尾数相除,并确定结果的符号。
3. 必要情况下规格化并四舍五入计算结果。

例 4.18　考虑两个浮点数 $X = 1.0000 \times 2^{-2}$和 $Y = -1.0100 \times 2^{-1}$的除法。

1. 指数部分相减:$-2-(-1) = -1$。
2. 尾数相除:$1.0000 \div (-1.0100) = -0.1101$。
3. 结果是 -0.1101×2^{-1}。

两个浮点数除法的示意图如图 4.15 所示。

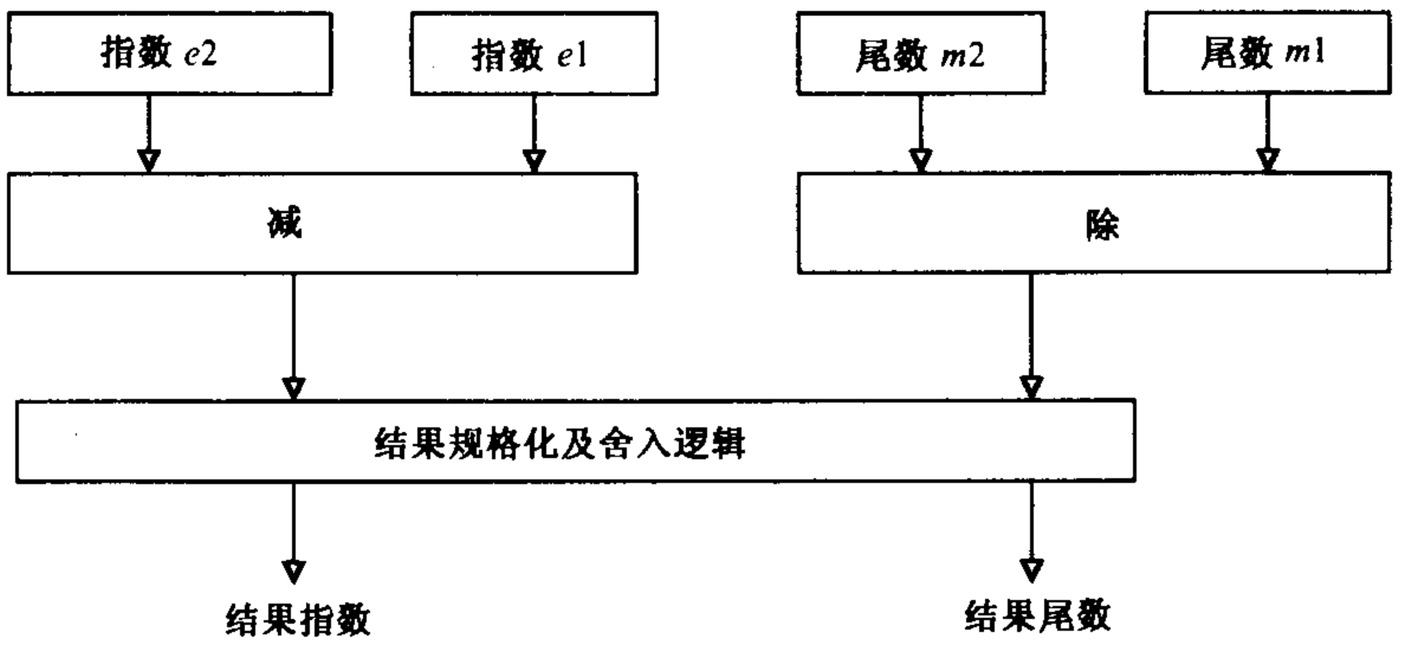


图 4.15　浮点数除

4.3.2　IEEE 浮点数标准

实际上存在两种 IEEE 标准的浮点数格式,它们是基本格式和扩展格式。而在每种格式中,IEEE 定义了两种格式,即单精度格式和双精度格式。单精度格式是 32 位,而双精度是 64 位。单精度扩展格式至少是 44 位,而双精度扩展格式至少是 80 位。

在单精度格式中,以 2 为基数,允许采用隐含位。指数字段的长度是 8 位。IEEE 单精度表示方法如图 4.16 所示。

图 4.16　IEEE 单精度表示

8 位指数可以表示 256 种组合情况,其中,保留两种组合情况以用于特定值:

1. $e = 0$ 用于表示 0(小数部分 $m = 0$)和非规格化数(小数部分 $m \neq 0$);
2. $e = 255$ 用于表示 $\pm \infty$(小数部分 $m = 0$)和非数(NaN)(小数部分 $m \neq 0$)。

	$m = 0$	$m \neq 0$
$e = 0$	0	非规格化
$e = 255$	$\pm \infty$	NaN

IEEE 单精度扩展格式将指数字段从 8 位扩展到 11 位,尾数部分从 23 + 1 位扩展到 32 位或更多位(其中没有隐含位),故总长度至少为 44 位。单精度扩展格式在计算中间结果时用到。

4.3.3 双精度 IEEE 格式

在这里指数字段是 11 位,有效位字段是 52 位,格式如图 4.17 所示。

图 4.17 双精度表示

类似于单精度格式,e 的极值(0 和 2047)以同样目的予以保留。

IEEE 单精度和双精度格式的特征属性如表 4.2 所示。

表 4.2 IEEE 单精度和双精度浮点格式的特征

特征	单精度	双精度
长度(位)	32	64
小数部分(位)	23	52
隐含(位)	1	1
指数长度位	8	11
偏置	127	1023
近似范围	$2^{128} \approx 3.8 \times 10^{38}$	$2^{1024} \approx 9.0 \times 10^{307}$
最小规格化数	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$

4.4 小结

本章讨论了一些关于计算机算术运算的问题。首先介绍了数的表示和基数转换技术。然后讨论了整数算术运算,特别是讨论了四种主要操作,即加、减、乘和除。在每种情况下都说明了基本体系结构和组织。最后讨论了浮点数的表示和运算。同时说明了执行诸如加、减、乘和除等基本浮点操作所需的基本体系结构。本章结尾介绍了 IEEE 浮点数的表示。

习题

1. 用下面的二进制格式表示带符号十进制数 26 和 – 123:
   (a) 符号 – 数值。
   (b) 2 补码。
2. 若给定的数是无符号整数,计算二进制数 1011 1101 0101 0110 对应的十进制数的值;若给定的数是 2 补码,重新计算其值;若给定的数是符号 – 数值整数,重新计算其值。
3. 下列加减运算中的 6 位二进制数是带符号 2 补码表示,计算每个算式,指出是否会发生溢出。

010110	011001	110111	100001	111111	011010
+001001	+010000	+101011	−011101	−000111	−100010

4. 用 2 位 Booth 算法对下列每对带符号 2 补码表示的数进行相乘。

$M = 010111$	$M = 110011$	$M = 110101$	$M = 1111$
$Q = 110110$	$Q = 101100$	$Q = 011011$	$Q = 1111$

5. 采用复位和非复位算法对下列每对带符号 2 补码表示的数进行相除。

$X = 010111$	$X = 110011$	$X = 110101$	$X = 1111$
$D = 110110$	$D = 101100$	$D = 011011$	$D = 1111$

6. 针对下列浮点数，说明如何进行加、减、乘和除。

$A =$	0	10001	011011
$B =$	1	01111	101010

这些数采用 12 位格式，其中，基数 $b = 2$，5 位指数字段 e，偏置 = 16，6 位规格化的尾数。

7. 设定两个 4 位输入数是 $A = a_4 a_3 a_2 a_1$ 和 $B = b_4 b_3 b_2 b_1$，针对所有进位 c_1, c_2, c_3, c_4 采用先行进位技术，说明一个 4 位加/减法器的完整设计（采用逻辑公式）。
8. 利用一个 4 位二进制加法器和最少的逻辑门设计一个 BCD 加法器，加法器输入是两个 4 位数 A 和 B，输出是 4 位和以及一个进位输出。
9. 利用图 4.5 所示的 4 位 CLA 部件，说明一个 16 位 CLA 的设计，计算其延迟和面积（以所需的逻辑门数量表示）。
10. 已知一个 32 位加法器，在其多层体系结构中采用了 4 位 CLA 加法部件，另一个是 32 位 CRT 加法器，比较这两个 32 位加法器从输入至输出间的最长路径延迟。假定一个门延迟由 T_g 表示。
11. 将下列十进制数转换为 IEEE 单精度浮点数表示。
    (a) −76
    (b) 0.92
    (c) 5.3125
    (d) −0.000 072
    (e) 8.04×10^{21}
12. 将下列 IEEE 单精度浮点数转换为十进制数。
    (a) 6589 00000
    (b) 807B 00000H
    (c) CDEF 0000H
13. 完成图 4.6 所示的阵列乘法器的逻辑设计。
14. 设计图 4.7 所示的控制逻辑。
15. 为图 4.10 所示的控制逻辑提供一个完整的逻辑设计。

参考文献和推荐读物

C. Hamacher, Z. Vranesic and S. Zaky, *Computer Organization*, 5th ed., McGraw-Hill, New York, 2002.

V. Heuring and H. Jordan, *Computer Systems Design and Archiecture*, Addison Wesley Longman, NJ, USA, 1997.

K. Israel, *Computer Arithmetic Algorithms*, 2nd ed., A. K. Peters, Ltd., Massachusetts, 2002.

W. Stallings, *Computer Organization and Architectures: Designing for Performance*, 4th ed., Prentice-Hall, NJ, USA, 1996.

B. Wilkinson, *Computer Architecture: Design and Performance*, 2nd ed., Prentice-Hall, Hertfordshire, UK, 1996.

第5章　处理单元设计

在前几章，我们学习了计算机系统的历史以及有关存储单元、寻址模式、汇编语言和计算机算术运算等基本问题。本章将重点关注计算机系统的主要部件，即中央处理器(CPU)。CPU 的主要功能是执行存放在计算机存储器中的一组指令。一个简单的 CPU 包括一组寄存器、一个算术及逻辑部件(ALU)和一个控制器(CU)。下面将向读者介绍 CPU 的组成和它的主要操作。

5.1　CPU 基础

一个典型的 CPU 包括三个主要部件：(1)寄存器组；(2)算术及逻辑部件；(3)控制器。不同计算机体系结构的寄存器组互不相同，它通常是通用和专用寄存器的组合。通用寄存器可用于任何目的，因此称之为通用。专用寄存器在 CPU 内部有特定的功能，例如，程序计数器(PC)是一个专用寄存器，用于保存下一条要执行指令的地址；另一个专用寄存器的例子是指令寄存器(IR)，用于保存当前正在执行的指令。ALU 提供电路以执行指令集中所需的算术、逻辑和移位操作。第 4 章已介绍了一些算术操作，以及支持 ALU 中这些计算的电路。控制器负责从主存中取出将被执行的指令，进行译码，然后执行该指令。CPU 的主要组成部分以及它与存储器系统和输入输出设备之间的交互如图 5.1 所示。

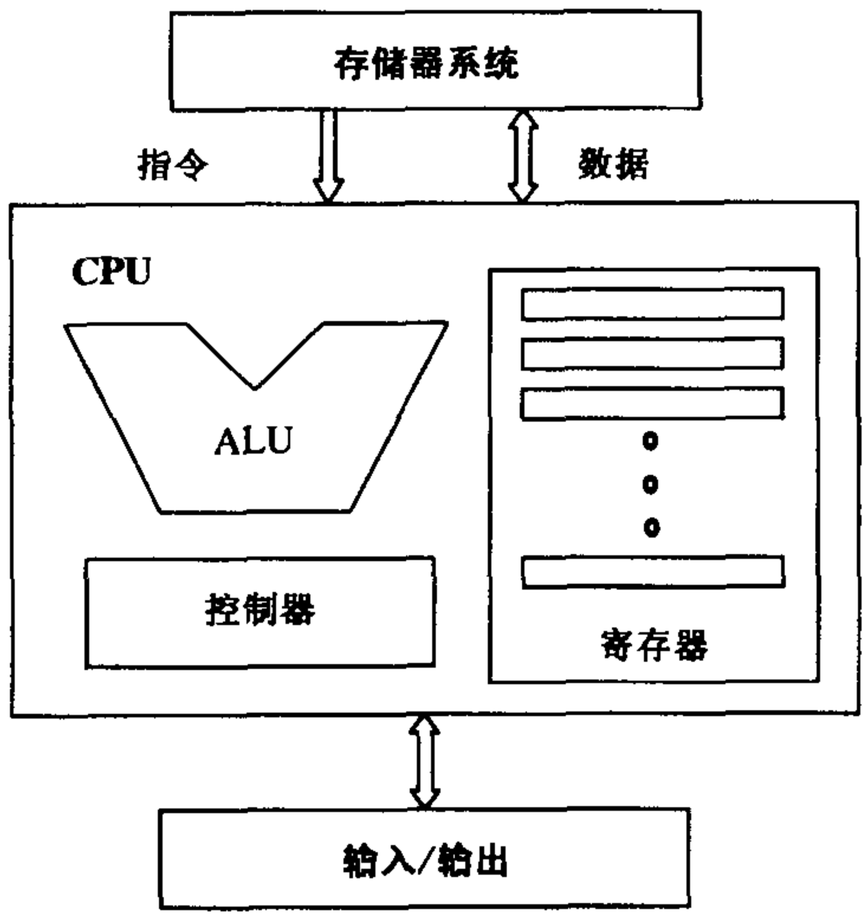


图 5.1　中央处理器的主要组成部分及其与存储器和 I/O 的交互

CPU 从存储器取出指令，对存储器进行数据的读和写，并对输入/输出设备执行数据的传送。一个简单、典型的执行周期可以归纳如下：

1. 从 PC 中获取下一条要执行指令的地址，从存储器中取出该指令并存入到 IR 中；
2. 译码该指令；
3. 根据需要，从存储器中取出操作数，并存放到 CPU 寄存器中；

4. 执行指令；

5. 根据需要，将结果从 CPU 寄存器传送到存储器中。

只要有待执行的指令，执行周期将一直重复。挂起中断检查通常也包括在这个周期中。中断的例子包括 I/O 设备请求、算术溢出或页缺失(见第 7 章)。当有一个中断请求时，就会转移到一个中断处理例程。被调用的中断处理例程是收集当前执行程序状态、标出中断原因并恢复程序状态的程序。

一个执行周期中 CPU 的动作是由控制器发出的微指令进行确定的。这些微指令是单独的控制信号，通过专用的控制线路发出。例如，假定执行一条将寄存器 X 中的内容移到寄存器 Y 中的指令，同时设定这两个寄存器通过数据总线 D 相连，控制器将发出一个控制信号，告诉寄存器 X 将其内容发到数据总线 D 上，在一些延迟之后，另外一个控制信号将发出以告诉寄存器 Y 从数据总线 D 上读取数据，控制信号的触发是由硬连线控制或微程序设计确定的。这些概念将在本章的稍后部分进行解释。

本章的后续部分是这样组织的：5.2 节介绍寄存器组，解释不同类型的寄存器及其功能；5.3 节介绍数据通路和控制；5.4 节和 5.5 节分别介绍 CPU 指令周期和控制器。

5.2　寄存器组

寄存器是 CPU 内部特别快速的存储单元，用来创建和保存 CPU 操作和其他计算的结果。不同的计算机具有不同的寄存器组，不同之处在于寄存器的数量、寄存器类型和每个寄存器的容量。它们的不同之处还在于每个寄存器的用途。通用寄存器可用于多种目的，由程序员分配多种功能，专用寄存器仅限于其特定的功能。在某些情况下，一些寄存器仅用于保存数据，而不能用于操作数地址的计算。数据寄存器必须具有足够的容量，以保存大多数数据类型的值。某些机器允许两个邻接的寄存器用于保存双倍长度的值。地址寄存器可以专用于特定的寻址模式，也可以用于通用地址。地址寄存器必须足够长以保存最大的地址。体系结构中的寄存器数量影响了指令集的设计，寄存器数量非常少将会导致存储器访问的增加。另一种寄存器用于保存处理器状态位或标志，这些标志位由 CPU 根据一个操作执行的结果而设定，这些状态位可在随后的其他操作中进行测试。

5.2.1　存储器访问寄存器

在存储器写和读操作中两个寄存器是必需的：存储器数据寄存器(MDR)和存储器地址寄存器(MAR)。MDR 和 MAR 由 CPU 专用，而不能由程序员直接访问。

为了对指定的存储单元执行写操作，MDR 和 MAR 的用法如下：

1. 将要保存到存储单元的字首先由 CPU 装入 MDR 中；
2. 用于保存该字的存储单元的地址由 CPU 装入到 MAR 中；
3. CPU 发出一个写信号。

类似地，为了执行一个存储器读操作，MDR 和 MAR 的用法如下：

1. 将要读取的字对应的存储单元地址装入 MAR 中；
2. CPU 发出一个读信号；
3. 需要的字由存储器装入到已就绪的 MDR 中以供 CPU 使用。

5.2.2 取指令寄存器

在取指令的过程中用到两个寄存器:程序计数器(PC)和指令寄存器(IR)。PC 是保存即将取出的下一条指令地址的寄存器,而已取出的指令则被装入到 IR 中以用于执行。在取指令成功之后,PC 更新以指向将被执行的下一条指令,在转移操作的情况下,在解析该转移之后,即确定目标地址之后,更新 PC 以指向转移目标指令。

5.2.3 条件寄存器

条件寄存器或标志用于保存状态信息,某些体系结构含有专用的程序状态字(PSW)寄存器,PSW 中包含一些位,由 CPU 设定其值以指示一个运行程序的当前状态。这些指示位一般用于算术操作、中断、存储器保护信息或处理器状态中。

5.2.4 专用地址寄存器

变址寄存器。正如第 2 章所述,在变址寻址中,操作数地址是通过对一个寄存器值加上一个常数获得,这个寄存器即变址寄存器。变址寄存器保存着一个地址偏移量,通过在指令中包含变址寄存器名字(圆括号内)并利用标识符 X 标明待加的常量,以表示变址寻址。

段指针。正如第 6 章所述,为了支持分段,处理器发出的地址由段号(基地址)和在该段内的偏移量(位移量)组成,段寄存器保存了这个段的基地址。

堆栈指针。如第 2 章所述,堆栈是一种数据组织机制,即最后存储的数据项是最先提取的数据项。堆栈两个特有的操作是压入和弹出操作,一个特定的寄存器,即堆栈指针(SP),用于指示可进行寻址的堆栈位置。在堆栈压入操作中,SP 值用于指示位置(即栈顶),在存储(压入)这个值之后,增加 SP 的值(在一些体系结构,如 x86 中,SP 的值是随着堆栈在存储器中的位置向下延伸而减小)。

5.2.5 寄存器

如第 3 章所述,Intel 386,486 和奔腾的基本编程模型包含三个寄存器组,它们分别是通用寄存器、段寄存器、指令指针(程序计数器)和标志寄存器。

图 5.2(同图 3.6)说明了这三组寄存器。第一组包括通用寄存器 A,B,C,D,SI(源变址)、DI(目的变址)、SP(堆栈指针)和 BP(基指针)。第二组寄存器包括 CS(代码段)、SS(堆栈段)以及 4 个数据段寄存器 DS,ES,FS 和 GS。第三组寄存器包括指令指针(程序计数器)和标志(状态)寄存器。在这些状态位中,前 5 个状态位等同于早在 8085 8 位微处理器中引入的那些位,接下来的 6 ~ 11 位等同于在 8086 中引入的那些位,而 12 ~ 14 位的标志在 80286 中引入,16 ~ 17 位在 80386 中引入,第 18 位的标志在 80486 中引入。

5.2.6 MIPS 寄存器

MIPS CPU 有 32 个通用寄存器,编号为 0 ~ 31,寄存器 x 由 $x 表示,$zero 寄存器总是存有硬连线值 0。表 5.1 列出了这些寄存器并描述了它们的用法,寄存器 $at(1),$k0(26)和 $k1(27)预留给汇编程序和操作系统使用,寄存器 $a0 ~ $a3(4 – 7)用来将前 4 个参数传送给例程(其他参数通过堆栈方式传递),寄存器 $v0 和 $v1(2,3)用于函数返回值,寄存器 $t0 ~ $t9(8 ~ 15,24,25)是调

用者保存寄存器,用于那些不需要在调用后进行保存的临时量。寄存器$s0 ~ $s7(16 ~ 23)是被调用者保存寄存器,用于保存长期存在的值,这些值应该在调用过程进行保存。

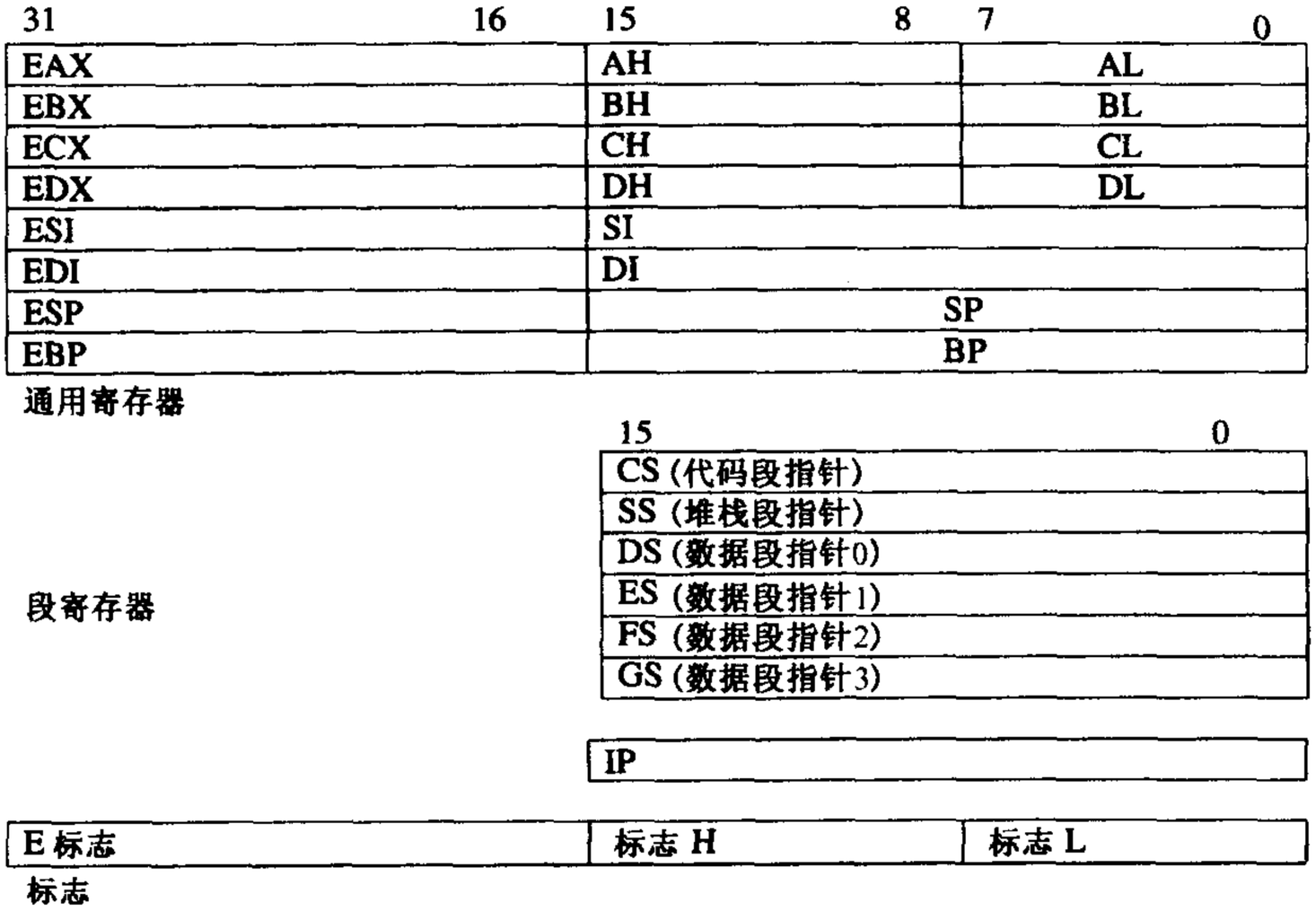


图 5.2　80x86 中的主要寄存器组(80386 及以上则扩展了除段寄存器之外的所有 16 位寄存器)

表 5.1　MIPS 通用寄存器

寄存器名	序号	用途	寄存器名	序号	用途
zero	0	常数	s0	16	临时保留(调用后保留)
at	1	预留给汇编程序	s1	17	临时保留(调用后保留)
v0	2	表达式求值	s2	18	临时保留(调用后保留)
v1	3	函数结果	s3	19	临时保留(调用后保留)
a0	4	参数 1	s4	20	临时保留(调用后保留)
a1	5	参数 2	s5	21	临时保留(调用后保留)
a2	6	参数 3	s6	22	临时保留(调用后保留)
a3	7	参数 4	s7	23	临时保留(调用后保留)
t0	8	临时(调用后不保留)	t8	24	临时(调用后不保留)
t1	9	临时(调用后不保留)	t9	25	临时(调用后不保留)
t2	10	临时(调用后不保留)	k0	26	预留给操作系统内核
t3	11	临时(调用后不保留)	k1	27	预留给操作系统内核
t4	12	临时(调用后不保留)	gp	28	全局指针
t5	13	临时(调用后不保留)	sp	29	堆栈指针
t6	14	临时(调用后不保留)	fp	30	帧指针
t7	15	临时(调用后不保留)	ra	31	返回地址(用于函数调用)

寄存器$sp(29)是堆栈指针,用来指示堆栈当前位置。寄存器$fp(30)是帧指针,寄存器$ra(31)用来保存函数调用的返回地址。寄存器$gp(28)是一个全局指针,指向堆(heap)中 64 K 存储块的中部,堆用来保存常量和全局变量,该堆中的对象可以通过单一的存取指令快速访问。

5.3 数据通路

CPU 可以分为数据部分和控制部分。数据部分即数据通路,包括寄存器和 ALU。数据通路能对数据项执行某些操作。控制部分基本上是控制器,用来向数据通路发出控制信号。在 CPU 内部,数据在两个寄存器间及 ALU 和寄存器之间移动,内部数据的移动是通过本地总线完成的,该总线可以传送数据、指令和地址。在外部,数据通常通过系统总线从寄存器移动至存储器或 I/O 设备。在寄存器间以及 ALU 和寄存器间的内部数据移动可以采用不同的结构,包括单总线、双总线和三总线。在数据传输频繁的两个部件之间也可以采用专用数据通路。例如在每个指令周期开始时 PC(程序计数器)中的值传送到 MAR(存储器地址寄存器)中以取一条新的指令,因此从 PC 到 MAR 之间的专用数据通路可以用来加速指令中这部分的执行。

5.3.1 单总线结构

采用单总线即 CPU 寄存器和 ALU 采用单一总线以移动输出和输入的数据。由于一条总线在一个时间周期只能处理一次数据移动,则对于 ALU,两操作数的操作需要两个周期,同时也需要额外寄存器用来给 ALU 缓存数据。这种总线结构是最简单的,也是花费最小的,但是限制了在同一时间周期内数据传输的数量,从而会降低总体的性能。如图 5.3 所示为一个单总线的数据通路,包括一组通用寄存器、一个存储器地址寄存器(MAR)、一个存储器数据寄存器(MDR)、一个指令寄存器(IR)、一个程序计数器(PC)和一个 ALU。

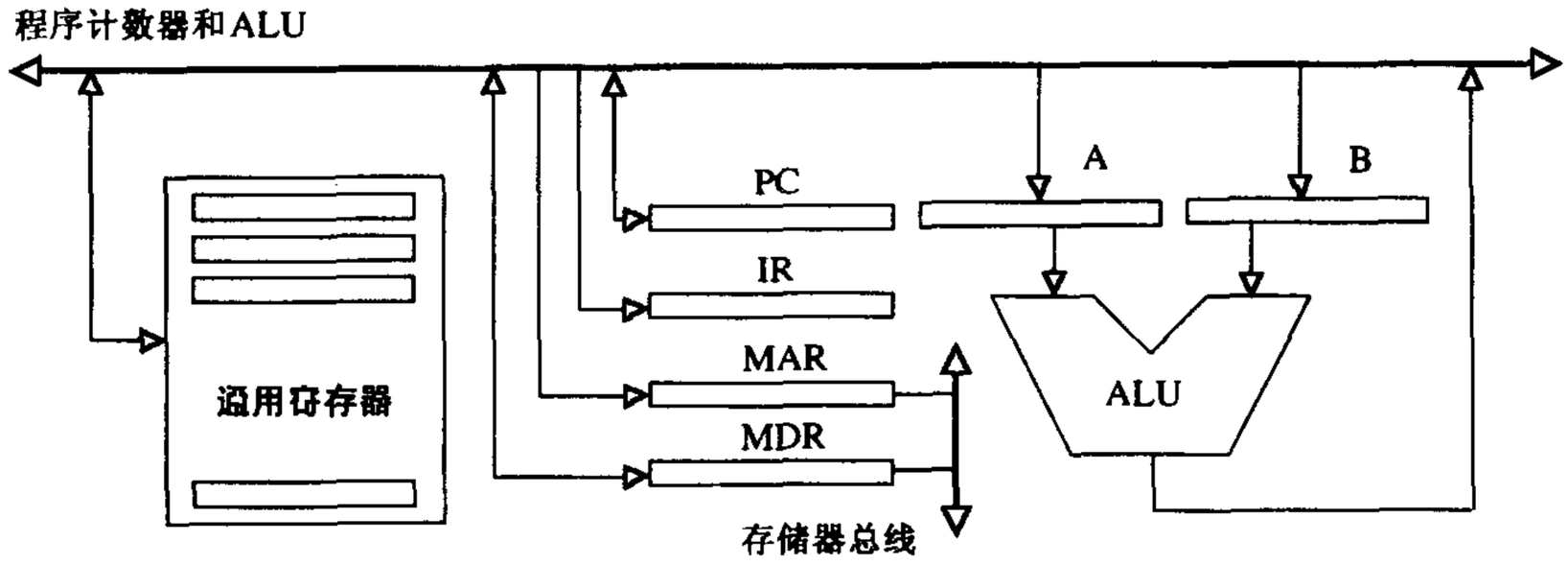


图 5.3 单总线的数据通路

5.3.2 双总线结构

相对于单总线结构而言,采用双总线是一种较快的解决方案。在双总线情况下,通用寄存器与两条总线均相连,数据可以同时从两个不同的寄存器传送到 ALU 的输入点,因此双操作数的操作可以在同一时间周期内取得两个操作数。当两条总线忙于传送两个操作数时,需要额外缓冲寄存器用来保存 ALU 的输出。如图 5.4(a)所示为一个双总线结构,在某些情况下,其中一条总线专门用来将数据移入到寄存器中(入总线),另一条总线专门用来将数据从寄存器中移出(出总线)。在这种情况下,需要额外缓冲寄存器作为 ALU 的输入,用来保存其中之一的操作数。ALU 的输出可以直接连到入总线上,该入总线将结果传送到一个寄存器中。如图 5.4(b)所示为一个具有入总线和出总线的双总线结构。

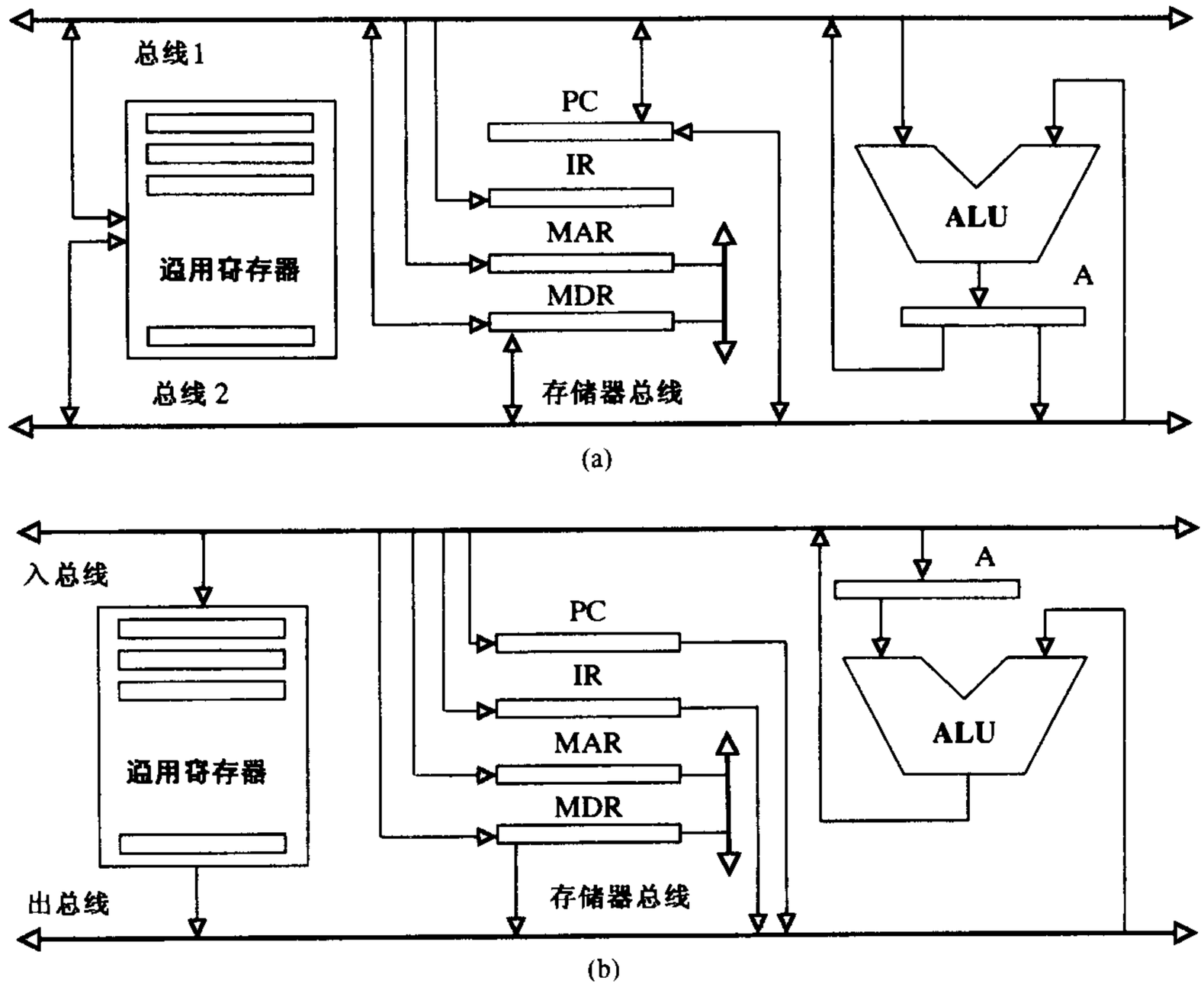


图 5.4　双总线组成

(a)双总线数据通路举例;(b)具有入总线和出总线的双总线数据通路的另一个示例

5.3.3　三总线结构

在三总线结构中,两条总线作为源总线,而第三条总线作为目的总线。源总线从寄存器中移出数据(出总线),目的总线将数据移入寄存器(入总线)。两个出总线均连接到 ALU 的输入点,ALU 的输出直接连接到入总线上。可想而知,采用越多的总线,在单一时间周期内可以移动的数据就越多。然而,增加总线数目也会增加硬件的复杂性。如图 5.5 所示是一个三总线数据通路的例子。

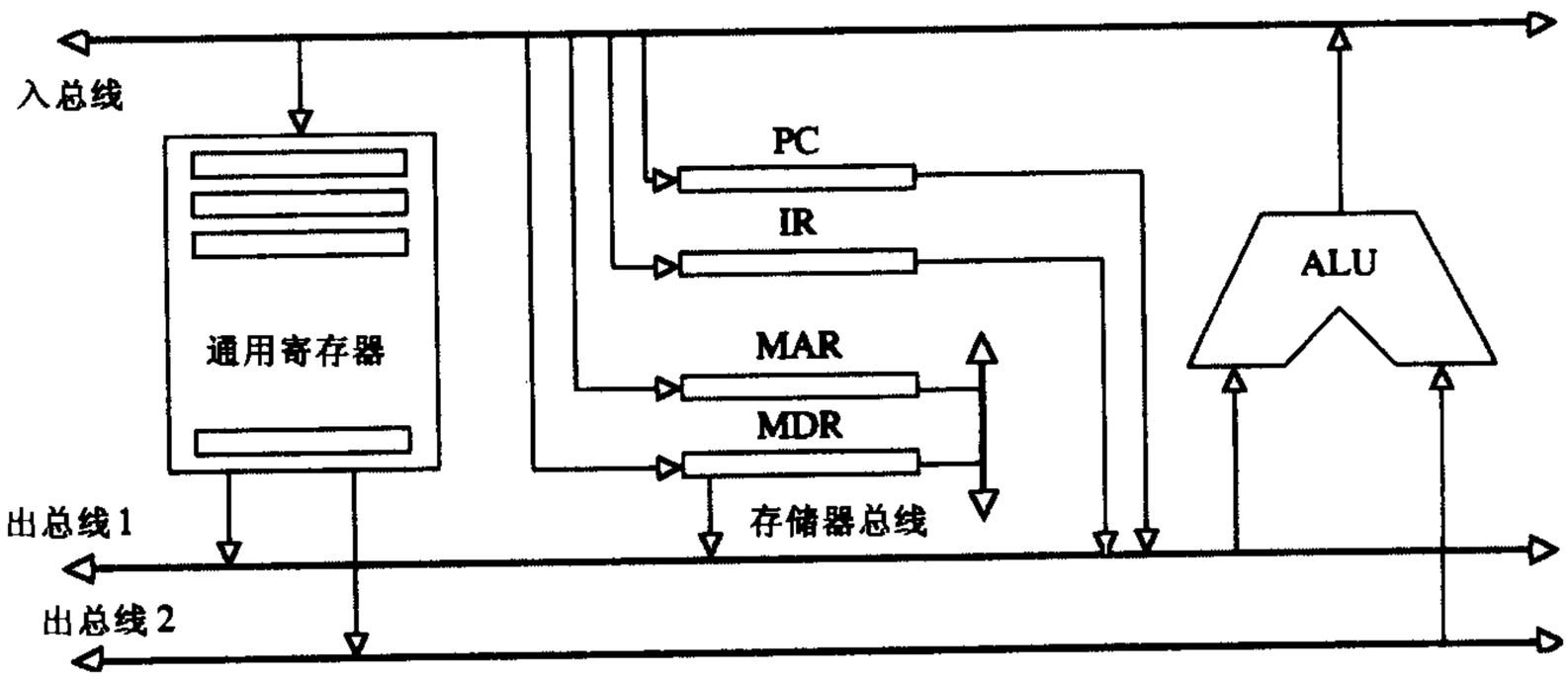


图 5.5　三总线数据通路

5.4 CPU 指令周期

CPU 执行指令的一系列操作如图 5.6 所示，只要还有指令待执行，下一条指令就将从主存中提取。执行指令是根据该指令中译码字段所说明的操作，在指令执行结束时，进行一个测试以确定是否已发生了一个中断，若发生中断则调用一个中断处理例程。取指令、执行指令或者处理中断的基本动作是由一系列微操作定义的。一组控制信号必须按照规定顺序进行启动以触发一个微操作的执行。本节将说明如何通过微操作实现取指令、执行简单算术指令和中断处理。

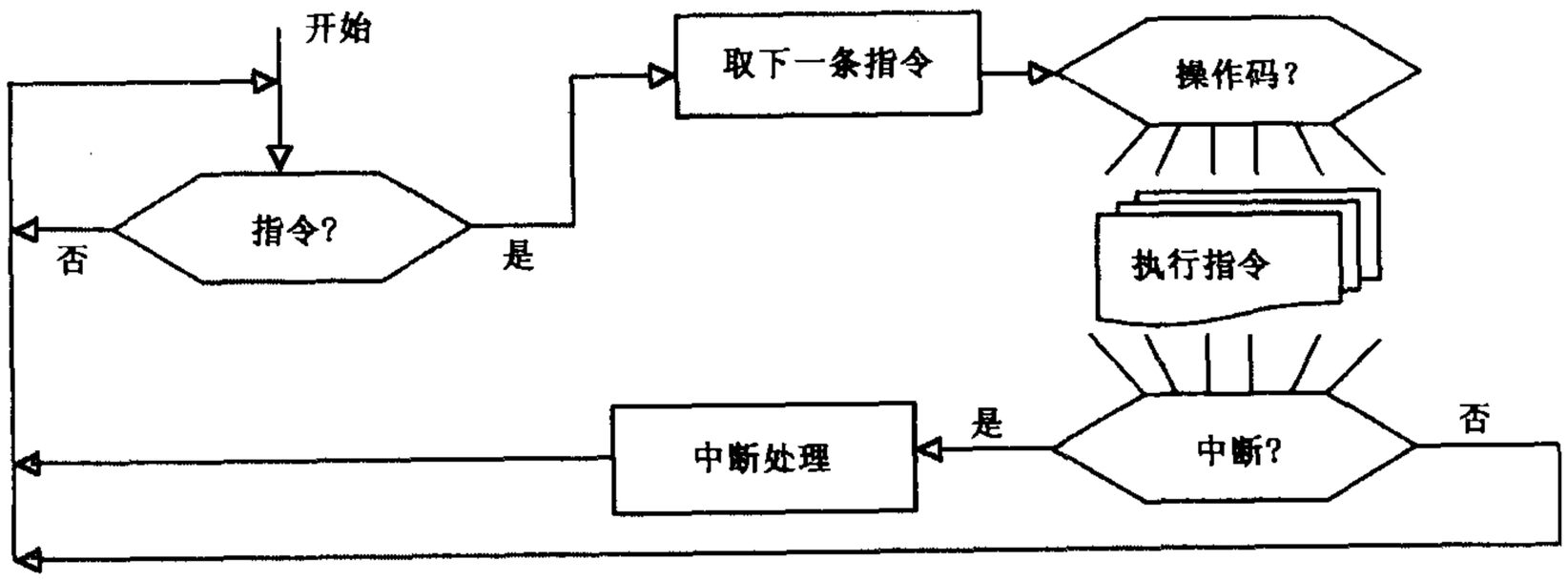


图 5.6 CPU 的功能

5.4.1 取指令

取一条指令的过程可以分为以下 4 步：

1. PC 中的值被装入到 MAR 中；
2. 增量 PC 中的值(这个操作可以与存储器访问并行进行)；
3. 通过存储器读操作，将指令装入 MDR 中；
4. 将 MDR 中的值装入到 IR 中。

如图 5.3 所示的单总线数据通路结构中，可以看到取操作可以通过下表所示的 3 步完成，其中 $t_0 < t_1 < t_2$。注意，多个操作由“;”间隔开，表示它们可以并行完成。

步骤	微操作
t_0	MAR ← (PC); A ← (PC)
t_1	MDR ← Mem[MAR]; PC ← (A) + 4
t_2	IR ← (MDR)

采用如图 5.5 所示的三总线数据通路，所需的步骤如下表所示。

步骤	微操作
t_0	MAR ← (PC); PC ← (PC) + 4
t_1	MDR ← Mem[MAR]
t_2	IR ← (MDR)

5.4.2 执行简单的算术操作

Add R_1, R_2, R_0。这个指令即是将源寄存器 R_1 和 R_2 的值相加，并将结果存到目的寄存器 R_0。相加过程如下所示：

1. 从 IR 中析取寄存器 R_1，R_2，R_0；

2. 将 R_1，R_2 中的值传送到 ALU 以进行相加；

3. 将 ALU 的输出传送到 R_0。

采用如图 5.3 所示的单总线数据通路，该加操作可以采用如下表所示的 3 步，其中，$t_0 < t_1 < t_2$。

步骤	微操作
t_0	A ← (R_1)
t_1	B ← (R_2)
t_2	R_0 ← (A) + (B)

采用如图 5.4(a)所示的双总线数据通路，该加操作可以采用如下表所示的两步，其中，$t_0 < t_1$。

步骤	微操作
t_0	A ← (R_1) + (R_2)
t_1	R_0 ← (A)

采用如图 5.4(b)所示的具有入总线和出总线的双总线数据通路，该加操作可以采用如下表所示的两步，其中，$t_0 < t_1$。

步骤	微操作
t_0	A ← (R_1)
t_1	R_0 ← (A) + (R_2)

采用如图 5.5 所示的三总线数据通路，该加操作可以采用如下表所示的一步完成。

步骤	微操作
t_0	R_0 ← (R_1) + (R_2)

Add X，R_0。该指令将存储单元 X 中的值加到寄存器 R_0，并将结果存储到寄存器 R_0。该加操作执行过程如下：

1. 从 IR 中析取存储单元 X，并装入到 MAR；

2. 通过存储器读操作，将存储单元 X 中的值装入到 MDR 中；

3. 将 MDR 中的值加到寄存器 R_0 中。

采用如图 5.3 所示的单总线数据通路，该加操作可以采取如下表所示的 5 步，其中，$t_0 < t_1 < t_2 < t_3 < t_4$。

步骤	微操作
t_0	MAR ← X
t_1	MDR ← Mem[MAR]
t_2	A ← (R_0)
t_3	B ← (MDR)
t_4	R_0 ← (A) + (B)

采用如图 5.4(a)所示的双总线数据通路，该加操作可以采取如下表所示的 4 步，其中，$t_0 < t_1 < t_2 < t_3$。

步骤	微操作
t_0	MAR ← X
t_1	MDR ← Mem[MAR]
t_2	A ←(R_0) + (MDR)
t_3	R_0←(A)

采用如图 5.4(b)所示的具有入总数和出总数的双总线数据通路,该加操作可以采用如下表所示的 4 步,其中,$t_0 < t_1 < t_2 < t_3$。

步骤	微操作
t_0	MAR ← X
t_1	MDR ← Mem[MAR]
t_2	A ← (R_0)
t_3	R_0← (A) + (MDR)

采用如图 5.5 所示的三总线数据通路,该加操作可以采取如下表所示的 3 步,其中,$t_0 < t_1 < t_2$。

步骤	微操作
t_0	MAR ← X
t_1	MDR ← Mem[MAR]
t_2	R_0← R_0 + (MDR)

5.4.3 中断处理

在执行一条指令之后,需要检测是否存在挂起的中断,若有等待的中断请求,则执行下列的步骤:

1. 将 PC 中的值装入到 MDR 中(加以保存);
2. 向 MAR 装入地址,即 PC 中的值将要被保存的地址;
3. 向 PC 装入中断处理例程第一条指令的地址;
4. 将 MDR 中的值(PC 中的原值)存入到存储器。

这个过程如下表所示,其中,$t_1 < t_2 < t_3$。

步骤	微操作
t_1	MDR ← (PC)
t_2	MAR ← address1(保存 PC 中的原值)
	PC ← address2(中断处理例程)
t_3	Mem[MAR] ← (MDR)

5.5 控制器

控制器是通过向数据通路发送控制信号以指挥系统操作的主要部件,这些控制信号控制着 CPU 内部、CPU 和外部单元之间的数据流,其中外部单元是指存储器和 I/O 设备等。控制总线通常以时钟驱动方式传送在控制器和其他 CPU 部件之间的信号,系统时钟以特定的持续时间和频

率产生一个连续系列的脉冲，一系列(时间)步 $t_0, t_1, t_2, \cdots, (t_0 < t_1 < t_2 < \cdots)$用于执行一条指令。通过对一条取出指令的操作码字段(op-code field)进行译码，可以向控制信号生成器提供该指令执行所需的相关信息，由逻辑电路模块产生的步信息以及其他输入信息用来产生控制信号。信号生成器可以通过一组布尔方程进行简要说明，在这组方程中输出是基于输入表达的。如图 5.7 所示为一个方块图，描述了如何在控制信号产生过程中运用定时。

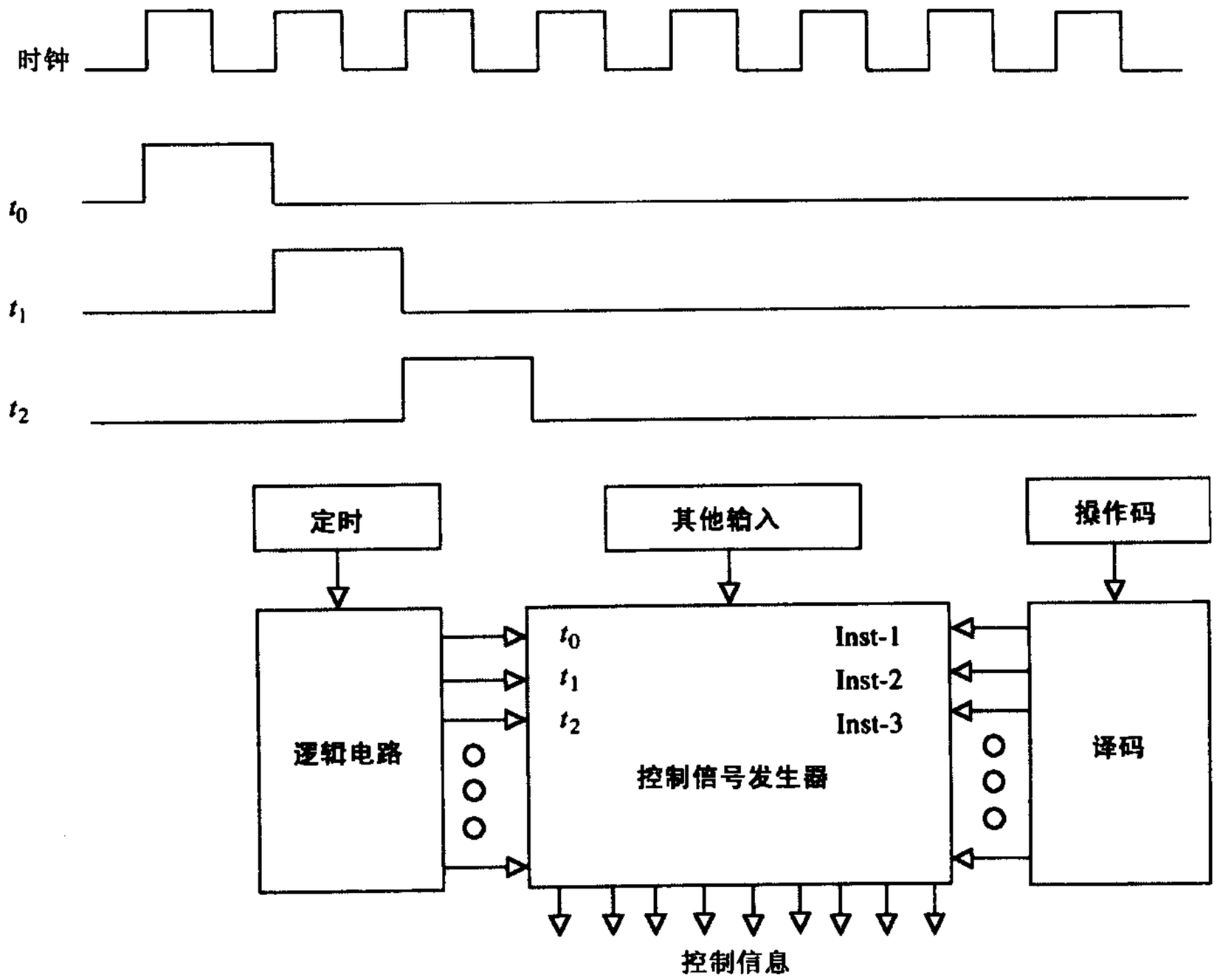


图 5.7　控制信号的定时

控制器主要有两种不同类型：微程序控制和硬连线控制。在微程序控制中，与操作相关的控制信号以控制字方式存储在特定的存储区域，而这个存储区域对于程序员而言是不可访问的。一个控制字是一条说明一个或多个微操作的微指令，我们称一系列微指令为一个微程序，它被存储在称为控制存储器 CM 的 ROM 或 RAM 中。

在硬连线控制中，针对布尔表达式直接采用相应固定逻辑电路产生控制信号。很明显，硬连线控制速度快于微程序控制。然而，对于复杂系统，硬连线控制将会很昂贵而且复杂，控制器越小采用硬连线控制就越经济。同时需要注意的是，微程序控制易于适应系统设计的变化，我们可以方便地增加新指令而不用改变硬件。而当系统发生任何变化时，硬连线控制则需要对整个系统进行重新设计。

例 5.1　重新考虑加操作：将源寄存器 R_1，R_2 中的值相加后将结果保存到目的寄存器 R_0 中。在先前的图 5.5 中已给出了采取三总线数据通路时该操作只需一步就可完成。

考察完成该加操作在 t_0 步时需要的控制序列，设定当前指令的操作码字段被译码为 Inst-x 类型。首先需要确定源寄存器和目的寄存器，接着确定 ALU 即将执行 Add(加)操作，所需的步骤和控制序列如下表所示。

步骤	指令类型	微操作	控制
t_0	Inst-x	$R_0 \leftarrow (R_1) + R_2$	选定 R_1 作为出总线 1 上的源 1(R_1 out-bus1) 选定 R_2 作为出总线 2 上的源 2(R_2 out-bus2) 选定 R_0 作为入总线上的目的(R_0 in-bus) 选定 ALU 的加功能(Add)

如图 5.8 所示为产生的信号,该信号用来执行 t_0 时期的 Inst-x。AND(与门)确保当操作码字段译码为 Inst-x 时及在 t_0 时刻能够发出这些信号。信号(R_1 out-bus1)、(R_2 out-bus2)、(R_0 in-bus)和(Add)分别选定 R_1 作为出总线 1 上的源、R_2 作为出总线 2 上的源、R_0 作为入总线上的目的和选定 ALU 的加功能。

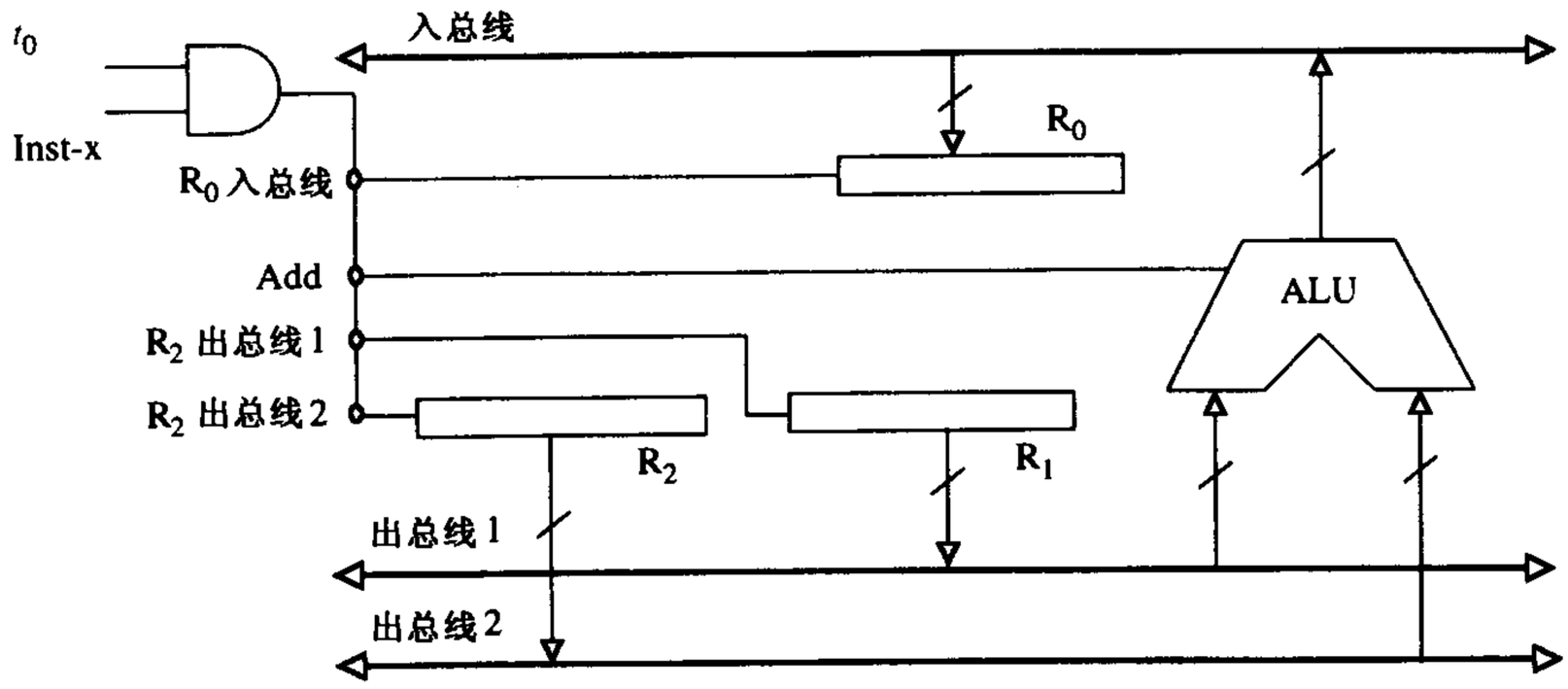


图 5.8 三总线数据通路上 t_0 周期内为执行 Inst-x 所产生的信号

例 5.2 采用如图 5.3 所示的单总线数据通路重复上述例子的操作,前面已经说明了采用单总线数据通路时该操作执行需要三步。设当前指令的操作码字段被译码为 Inst-x 类型,所需的步骤和控制序列如下表所示。

步骤	指令类型	微操作	控制
t_0	Inst-x	$A \leftarrow (R_1)$	选定 R_1 作为源(R_1 out)选定 A 作为目的(A in)
t_1	Inst-x	$B \leftarrow (R_2)$	选定 R_2 作为源(R_2 out)选定 B 作为目的(B in)
t_2	Inst-x	$R_0 \leftarrow (A) + (B)$	选定 ALU 的加功能(Add)选定 R_0 作为目的(R_0 in)

执行 t_0, t_1, t_2 周期 Inst-x 的信号如图 5.9 所示。AND 确保在 op-code 被译码为 Inst-x 时和其他时刻适当的信号能够发出。在 t_0 周期,发出信号(R_1 out)和(A in)将 R_1 中的值移入到 A 中。类似地,在 t_1 周期,发出信号(R_2 out)和(B in)将 R_2 中的值移入到 B 中。最后,在 t_2 周期,发出信号(R_0 in)和(Add),将 A 和 B 中的值相加并将结果移入到 R_0 中。

5.5.1 硬连线实现

硬连线控制通过逻辑电路直接实现。如图 5.7 所示,对于每个控制线必须根据控制信号产生器的输入确定其布尔表达式,可以通过下面的简单实例说明其实现。设一台机器的指令系统有以下 3 条指令:Inst-x,Inst-y 和 Inst-z;A,B,C,D,E,F,G 和 H 均是控制线。下表说明了在 t_0,t_1,t_2 步时钟这 3 条指令需要激活控制线的情况。

步骤	Inst-x	Inst-y	Inst-z
t_0	D,B,E	F,H,G	E,H
t_1	C,A,H	G	D,A,C
t_2	G,C	B,C	

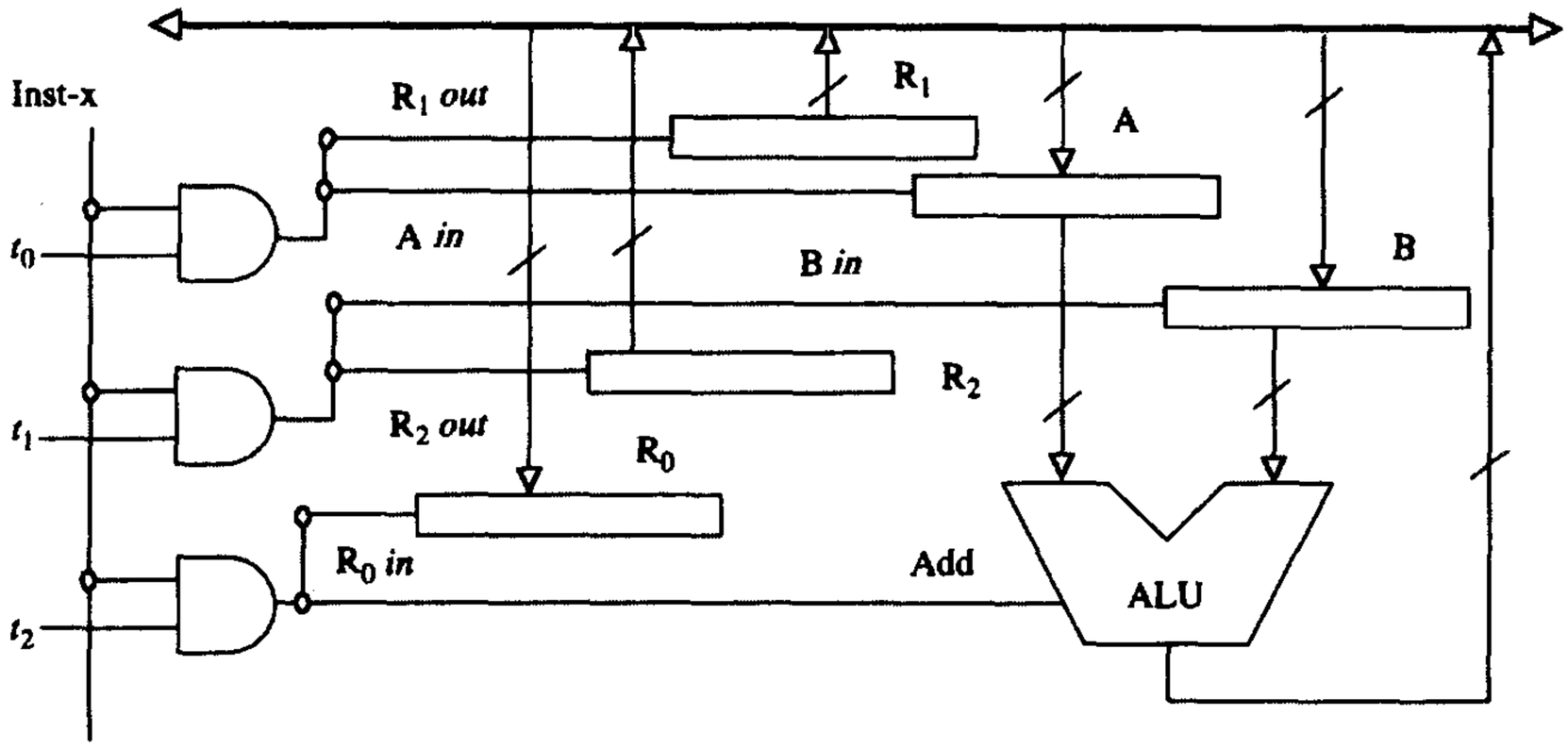


图 5.9　单总线数据通路在 t_0, t_1, t_2 周期为执行 Inst-x 所产生的信号

控制线 A,B 和 C 的布尔表达式如下：

$$A = \text{Inst-x} \cdot t_1 + \text{Inst-z} \cdot t_1 = (\text{Inst-x} + \text{Inst-z}) \cdot t_1$$

$$B = \text{Inst-x} \cdot t_0 + \text{Inst-y} \cdot t_2$$

$$C = \text{Inst-x} \cdot t_1 + \text{Inst-x} \cdot t_2 + \text{Inst-y} \cdot t_2 + \text{Inst-z} \cdot t_1$$
$$= (\text{Inst-x} + \text{Inst-z}) \cdot t_1 + (\text{Inst-x} + \text{Inst-y}) \cdot t_2$$

如图 5.10 所示为这些控制线的逻辑电路，其余控制线的布尔表达式可以通过类似方式获得。如图 5.11 所示为这些指令执行周期的状态图。

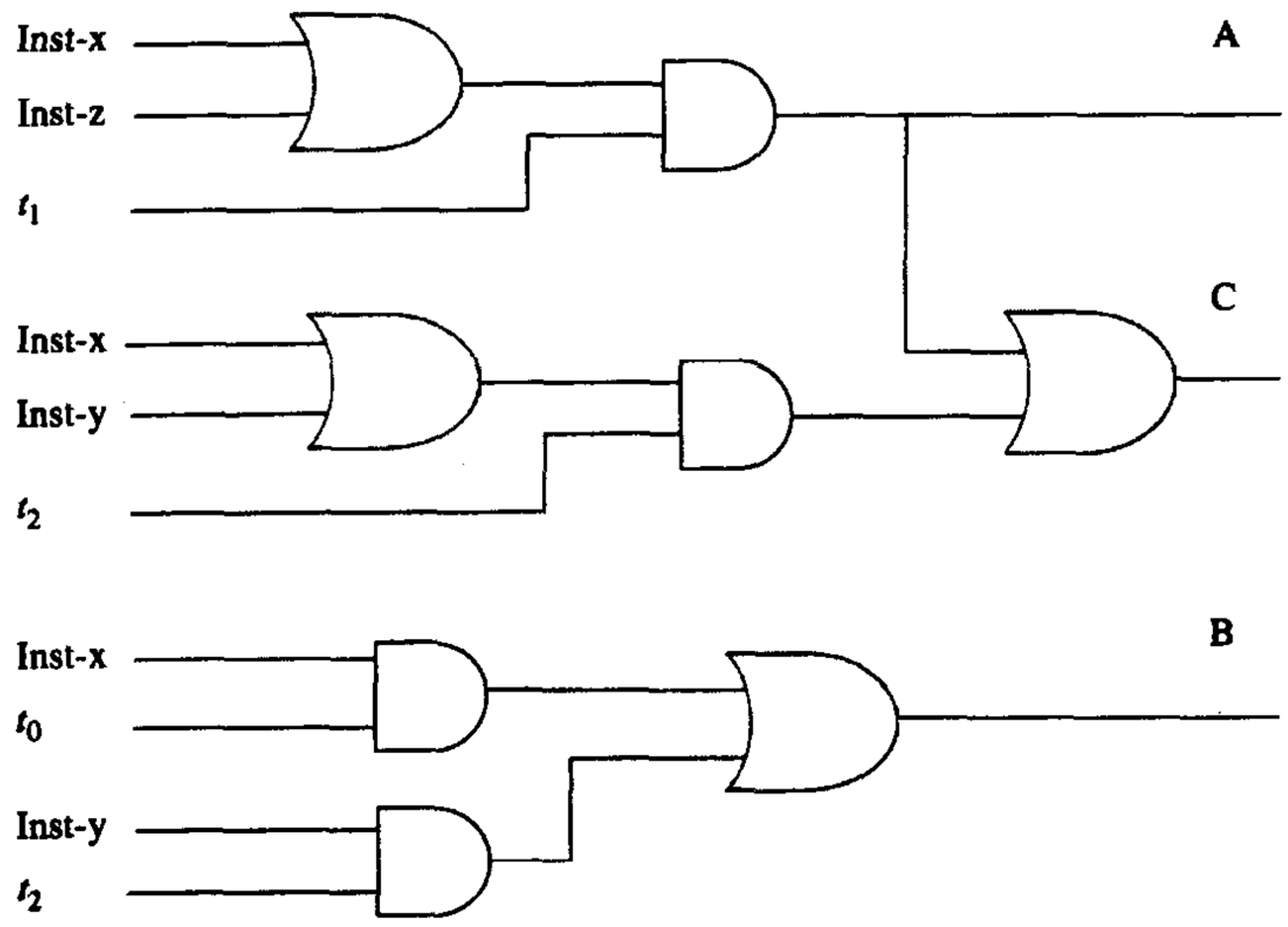


图 5.10　控制线 A,B 和 C 的逻辑电路

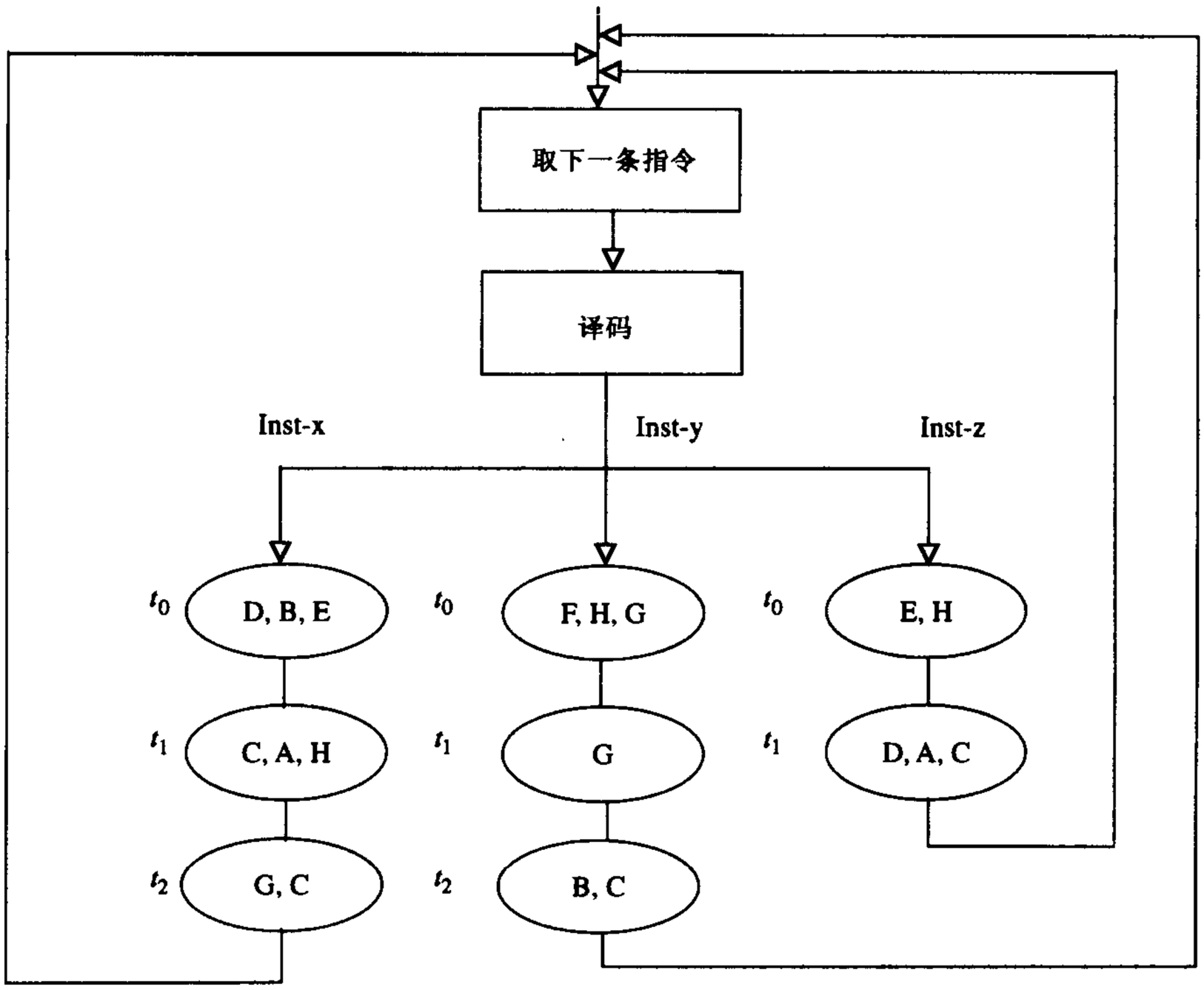


图 5.11　指令执行状态图

5.5.2　微程序控制器

微程序控制器的思想是由 M. V. Wilkes 在 20 世纪 50 年代早期提出的，为降低硬连线控制的复杂性而促成了微程序方法。正如前面分析的，一条指令可以用一组微操作加以实现。而与每一个微操作相关的是一组控制线，要执行相应的微操作必须激活这些控制线。微程序控制的思想是，将与一条给定指令实现相关的控制信号，作为一个微程序存储在一个特定的存储器，即控制存储器(CM)中。一个微程序包括一系列的微指令，一条微指令是一个位向量，其中每个位是一个控制信号，一个条件代码或下一条微指令地址。从 CM 中取微指令的过程与从主存中取程序指令的方式相同，如图 5.12 所示。

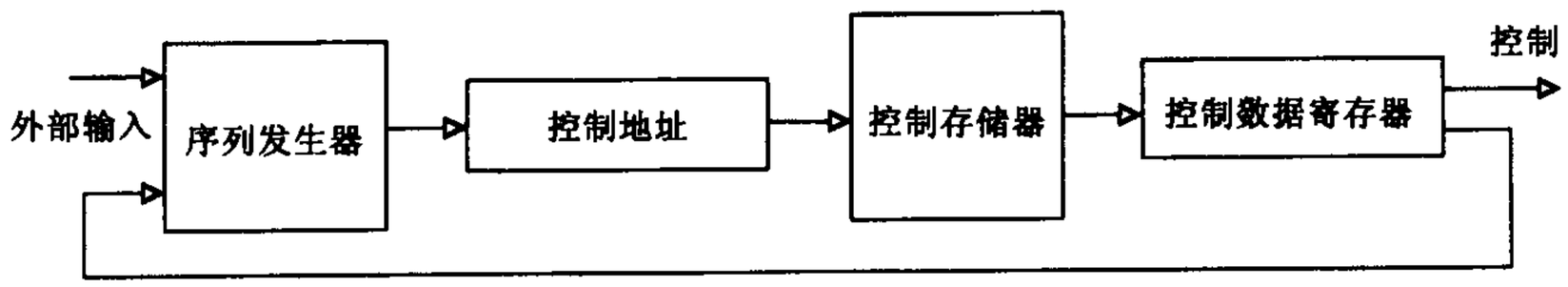


图 5.12　取微指令(控制字)

当从存储器中取一条指令时，指令操作码字段确定了将执行何种微程序。换言之，即将操作码映射到控制存储器中的微指令地址，微指令处理器根据该地址取出微程序第一条微指令。在取得每条微指令后，便能操作相应的控制线。对应于“1”位的控制线将接通，而对应于“0”位的控制线将关闭。在执行完一条微指令之后，一条新的微指令将被取出并执行。若条件代码判断必须执行一个转移，则要在当前微指令的地址位指明下一条微指令，否则将取出序列中的下一条微指令并加以执行。

一条微指令的长度取决于该微指令中所指明的微操作的数目、控制位的解释方式，以及下一条微指令地址的获取方式。一条微指令可指定一个或多个同时被激活的微操作。微指令的长度将随每条微指令并行微操作数目的增加而增长。此外，当微指令中每个控制位恰好对应一条控制线时，微指令的长度则会变得更长。若将控制线在微指令的特殊字段进行编码，则微指令的长度可以缩短，但需要译码器将每个字段映射到各条单独的控制线。显而易见，采用译码器将会降低需被同时激活的控制线数目。在微指令长度和并行度之间需要进行平衡。缩短微指令长度将降低成本和对控制存储器的访问时间，这一点很重要；但使更多的微操作可以并行执行以及更多的控制线能够同时激活，也是我们所希望的。

水平微指令与垂直微指令。微指令可以分为水平微指令和垂直微指令。在水平微指令中每一位对应于一条控制线，由于每一位控制一个控制线，因此水平微指令较长，但允许最大的并行性。在垂直微指令中，将控制线编码到微指令的特定字段，采用译码器将 k 位的字段映射到 2^k 种可能的控制线组合。例如，在微指令中一个 3 位的字段可以用来指定 8 种可能控制线中的任一种。由于进行编码，垂直微指令比水平微指令短，但编码在同一个字段的控制线不能同时被激活，因此垂直微指令只能允许有限的并行性。值得注意的是，在水平微指令中无需译码，而在垂直微指令中则是必须的。

例 5.3　考虑如图 5.5 所示的三总线数据通路。除了 PC，IR，MAR 和 MDR 之外，设定有 16 个通用寄存器，其编号为 $R_0 \sim R_{15}$。另外，设 ALU 支持 8 种功能(加、减、乘、除、AND、OR、左移和右移)。考虑加操作 Add R_1，R_2，R_0，就是将源寄存器 R_1，R_2 中的值相加，并将结果保存到目的寄存器 R_0。在这个例子中，我们将分析在水平组织方式下微指令的格式。

在采用的水平微指令中，对应于每条控制线有一个控制位，微指令格式需要为以下操作设置控制位：

- ALU 操作；
- 输出至出总线 1 的寄存器(源 1)；
- 输出至出总线 2 的寄存器(源 2)；
- 从入总线输入的寄存器(目的)；
- 未列出的其他相关操作。

ALU、源 1、源 2 和目的寄存器所需的位数如下表所示。

目的	位数目	说明
ALU	8 位	8 种功能
源 1	20 位	16 个通用寄存器 + 4 个专用寄存器
源 2	16 位	16 个通用寄存器
目的	20 位	16 个通用寄存器 + 4 个专用寄存器

在三总线数据通路上，Add R_1，R_2，R_0 对应的微指令如图 5.13 所示。

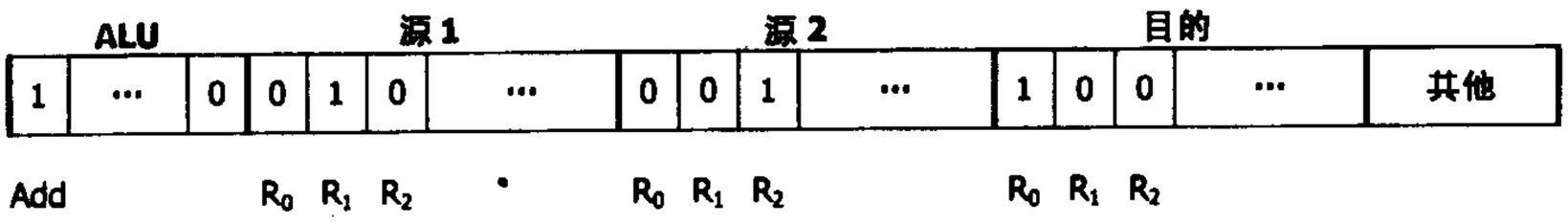


图 5.13　Add R_1，R_2，R_0 的(水平)微指令

例 5.4　这个例子采用垂直微指令，其中用到译码器，也将用到如图 5.5 所示的三总线数据通路，设有 16 个通用寄存器，ALU 支持 8 种功能。ALU 功能、连接于出总线 1 的寄存器(源 1)、连接于出总线 2 的寄存器(源 2)和连接于入总线的寄存器(目的)，它们的编码如下面 3 个表所示。

目的	位数	说明
ALU	4 位	8 种功能 + 无
源 1	5 位	16 个通用寄存器 + 4 个专用寄存器 + 无
源 2	5 位	16 个通用寄存器 + 无
目的	5 位	16 个通用寄存器 + 4 个专用寄存器 + 无

编码	ALU 功能
0000	无(ALU 将连接出总线 1 到入总线)
0001	加
0010	减
0011	乘
0100	除
0101	与
0110	或
0111	左移
1000	右移

编码	源 1	目的	编码	源 2
00000	R_0	R_0	00000	R_0
00001	R_1	R_1	00001	R_1
00010	R_2	R_2	00010	R_2
00011	R_3	R_3	00011	R_3
00100	R_4	R_4	00100	R_4
00101	R_5	R_5	00101	R_5
00110	R_6	R_6	00110	R_6
00111	R_7	R_7	00111	R_7
01000	R_8	R_8	01000	R_8
01001	R_9	R_9	01001	R_9
01010	R_{10}	R_{10}	01010	R_{10}
01011	R_{11}	R_{11}	01011	R_{11}
01100	R_{12}	R_{12}	01100	R_{12}
01101	R_{13}	R_{13}	01101	R_{13}
01110	R_{14}	R_{14}	01110	R_{14}
01111	R_{15}	R_{15}	01111	R_{15}
10000	PC	PC	10000	None
10001	IR	IR		
10010	MAR	MAR		
10011	MDR	MDR		
10100	NONE	NONE		

在垂直组织方式下采用三总线数据通路，Add R_1，R_2，R_0 对应的微指令如图 5.14 所示。

ALU				源 1					源 2					目的						
0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	其他

图 5.14　Add R_1，R_2，R_0 的(垂直)微指令

例 5.5　采用与例 5.4 相同的编码，确定取指令所使用的垂直微指令。

MAR←PC。首先，将源 1 设为“10000”，这样选定 PC 作为源 1。类似地，在目的域中设“10010”，选定 MAR 作为目的。同时需要将 ALU 字段设为“0000”，其译码为“NONE”，如 ALU 编码表(例 5.4)所示，“NONE”表示出总线 1 连接到入总线。源 2 字段设为“10000”，即不选定寄存器。微指令如图 5.15 所示。

ALU				源 1					源 2					目的						
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	其他

图 5.15　MAR←PC 的微指令

存储器读和写。存储器操作可以很容易进行调节，即为读和写各额外加一个位，如图 5.16 所示的两条微指令分别执行存储器读和写。

					R	W	
ALU	源 1	源 2	目的		1	0	其他

MDR ← Mem[MAR]

					R	W	
ALU	源 1	源 2	目的		0	1	其他

Mem[MAR] ← MDR

图 5.16　存储器读和写的微指令

取指令。采用如图 5.17 所示的 3 条微指令可以执行取指令操作。

	ALU				源 1					源 2					目的				R	W	
MAR ← (PC)	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	其他
MDR ← Mem[MAR]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	其他
IR ← MDR	0	0	0	0	1	0	0	1	1	1	0	0	0	0	1	0	0	0	1	0	0	其他

图 5.17　取指令的微指令

第一条、第二条微指令已在上面进行了说明。第三条微指令将 MDR 中的值移入到 IR(IR ← MDR)。在源 1 字段用“10011”将 MDR 选做源 1，类似地，在目的字段用“10001”将 IR 选做目的。另外，需要在 ALU 字段用“0000”(“NONE”)，将出总线 1 连接到入总线。源 2 字段需要设为“10000”，即表示不选定任何寄存器。

5.6　小结

CPU 作为计算机的主要组成部分，负责解释和执行包含在程序里的指令。CPU 的主要部件

是寄存器组、ALU 和控制器。寄存器文件包括通用寄存器和专用寄存器，通用寄存器可用于保存操作数和中间结果，专用寄存器用于存储器访问、排序、保存状态信息或者在译码和执行期间用于保存取出的指令等。算术和逻辑操作是在 ALU 中进行的，在 CPU 内部，数据可以从一个寄存器移到另一个寄存器，或者在寄存器和 ALU 之间进行移动。数据还可以在 CPU，以及如存储器和I/O这样的外围部件之间移动。控制器是控制指令周期状态的部件。只要还有指令需要执行，则下一条指令将从主存中取出。指令是根据在该指令操作码字段中规定的操作执行的。控制器产生信号用来控制 CPU 内部的数据流，以及 CPU 和如存储器和 I/O 那样的外围单元之间的数据流。控制器的实现可以采用硬连线技术或微程序技术。

习题

1. 指出下列指令所需的指令位：
   (a) 在一个有 64 个通用寄存器的机器中的两个操作数寄存器和一个结果寄存器；
   (b) 在有 64 KB 主存的机器中的 3 个存储地址。
2. 分别采用下列系统，说明 load，store，jump 这 3 个操作的微操作：
   (a)单总线系统；
   (b)双总线系统；
   (c)三总线系统。
3. 在 5.4 节的所有表格中加入控制信号。
4. 在 CPU 内部进行数据移动可以采用几种不同的方式。对比下列方法的优点和不足：
   (a) 专用连接；
   (b) 单总线数据通路；
   (c) 双总线数据通路；
   (d) 三总线数据通路。
5. 为下表中的微指令确定一种编码方法，使得控制位数最小，同时保持微操作之间原有的并行性。

微指令	激活的控制信号
I_1	a,b,c,d,e
I_2	a,d,f,g
I_3	b,h
I_4	c
I_5	c,e,g,i
I_6	a,h,j

6. 设一台机器的指令集中有 3 条指令：Inst-1，Inst-2 和 Inst-3；A，B，C，D，E，F，G，H 是控制线。针对 3 步骤 T0，T1，T2 时的 3 条指令，需要激活控制线的情况如下表所示。

步骤	Inst-1	Inst-2	Inst-3
T0	D,B,E	F,H,G	E,H
T1	C,A,H	G	D,A,C
T2	G,C	B,C	

(a) 硬连线方法：

(i) 对所有控制线 A-G，写出其布尔表达式。

(ii) 为每条控制线画出其逻辑电路。

(b) 微程序方法：

(i) 采用水平表示，为指令 Inst-1 写出其微程序，并指出微指令的大小。

(ii) 若同时采用水平表示和垂直表示，最好的分组方式是什么？微指令大小为多少？写出 Inst-1 的微程序。

7. 某一处理器有一种微指令格式，它包括 10 个不同的控制字段 $C_0 \sim C_9$。每个 C_i 可以激活 n_i 个不同控制线的任意一种，其中 n_i 的规定如下所示。

i:	0	1	2	3	4	5	6	7	8	9
n_i:	4	4	3	11	9	16	7	1	8	22

(a) 表示这 10 个控制字段所需控制位的最小值是多少；

(b) 若对于所有控制信息都采用纯水平格式，则所需控制位的最大值是多少。

8. 说明下列两者之间的主要不同之处。

(a) 垂直微指令和水平微指令；

(b) 微程序和硬连线控制。

9. 针对条件转移指令，采用单总线体系结构，以适当的次序(用最少的微指令数)产生必要的控制信号。

10. 分别采用单总线数据通路和双总线数据通路，写出取指令的微程序。

参考文献和推荐读物

R. J. Baron and L. Higbie, *Computer Architecture*, Addison Wesley, Canada, 1992.

M. J. Flynn, *Computer Architecture*, Jones and Barlett, MA, USA, 1995.

J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, New York, 1998.

J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, *CA*, 2003.

V. P. Heuring and H. F. Jordan, *Computer Systems Design and Architecture*, Addison Wesley, NJ, 1997.

M. Murdocca and V. Heuring, *Principles of Computer Architecture*, Prentice Hall, NJ, USA, 2000.

D. Patterson and J. Hennessy, *Computer Organization and Design*, Morgan Kaufmann, San Mateo, CA, 1998.

W. Stallings, *Computer Organization and Architecture: Designing for Performance*, NJ, 1996.

A. S. Tanenbaum, *Structured Computer Organization*, Prentice Hall, NJ, USA, 1999.

第6章　存储系统设计Ⅱ

本章将讨论计算机存储系统。第3章已说明,若没有存储器,则无法在计算机里存储或检索信息。有趣的是,早在1946年Burks,Goldstine和Von Neumann已经认识到计算机存储系统必须以层次方式进行组织。在这个层次结构中,较大较慢的存储器辅助较小较快的存储器,这一论断从那时起被证明在构造一个计算机存储器中是至关重要的。若撇开CPU寄存器组(CPU内的一级存取信息设施,参见第5章),则在典型的存储器层次中,首先是小的、昂贵的、相对较快的部件,即高速缓存。在层次结构中高速缓存之后是较大、较廉价且相对较慢的主存部件。高速缓存和主存采用固态半导体材料制造。层次结构中的下一级是更大的、较廉价的、更慢的磁性存储器,包括(硬)磁盘和磁带。本章首先讨论决定计算机存储器层次的特征和因素,然后是高速缓存的设计和分析,(主)存储部件将在第7章讨论,在第7章还要讨论与虚拟存储器相关的问题,同时还将简要介绍只读存储器(ROM)的不同实现。

6.1　基本概念

本节将介绍与计算机存储器层次相关的一些基本概念。

6.1.1　存储器层次结构

正如上面所述,在一个典型的存储器层次结构中,首先是小而昂贵且相对快速的部件,即高速缓存,其次是较大且较便宜的、相对较慢的主存部件。高速缓存和主存储器采用固态半导体材料制成(一般是CMOS晶体三极管),通常称这种快速存储器为主存。固态存储器之后是较大且便宜的慢速磁存储器,一般包括(硬)磁盘和磁带,通常称磁盘为二级存储器,而磁带则称为三级存储器。设计存储器层次结构的目标是,建立一个存储器系统,其性能如同该系统全部由最快的单元组成,而其费用却是由最慢单元的费用主导。

存储器层次可以通过一些参数进行描述,这些参数是:访问类型、存储器容量、周期时间、时延、带宽和成本。术语访问(access)是指在读或写操作期间所发生的物理动作,存储器层次结构的容量通常以字节度量,周期时间是指从一个读操作开始至下一个读操作之间的时间,时延则是指从请求信息到访问该信息第一位之间的时间间隔,带宽是对每秒访问位数的度量,而一个存储器层次结构的成本一般是以美元/MB来衡量的。图6.1描述了一个典型的存储器层次结构,表6.1所示是存储器层次结构参数的典型取值。

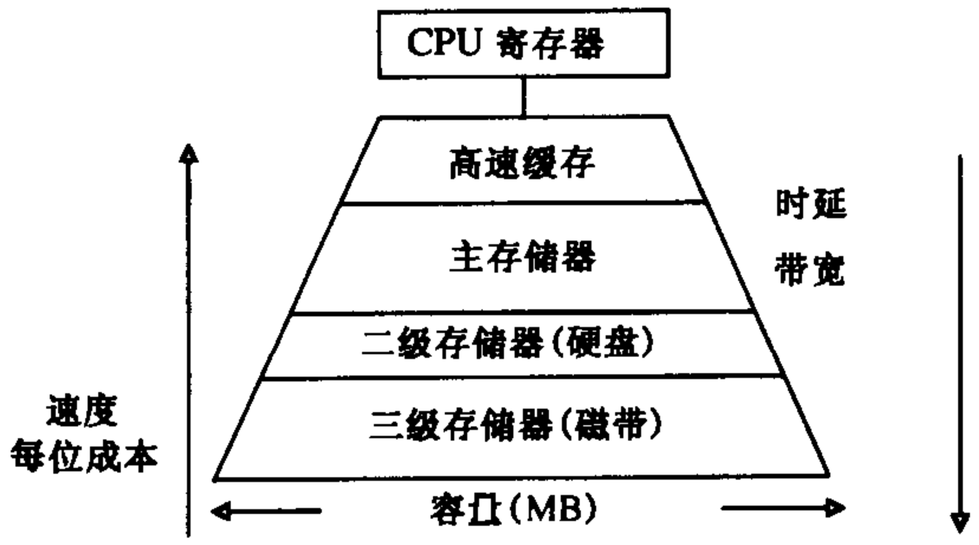


图6.1　典型的存储器层次结构

表 6.1　存储器层次结构的参数

	访问类型	容量	时延	带宽	成本/MB
寄存器	随机	64 ~ 1024 B	1 ~ 10 ns	系统时钟速率	高
高速缓存	随机	8 ~ 512 KB	15 ~ 20 ns	10 ~ 20 MB/s	500 美元
主存储器	随机	16 ~ 512 MB	30 ~ 50 ns	1 ~ 2 MB/s	20 ~ 50 美元
硬盘存储器	直接	1 ~ 20 GB	10 ~ 30 ms	1 ~ 2 MB/s	0.25 美元
磁带存储器	顺序	1 ~ 20 GB	30 ~ 10 000 ms	1 ~ 2 MB/s	0.025 美元

术语随机访问是指这样一个情况，对任何存储单元进行任意访问所需要的时间是相同的，而与实际存储单元所在位置和访问序列无关。例如，若对存储单元 100 进行一次写操作需要 15 ns，且该操作之后是对存储单元 3000 的一个读操作，则该读操作也将需要 15 ns。与之相对的是顺序访问，若对单元 100 处的访问需要 500 ns，紧接其后对单元 101 的访问需要 505 ns，而访问单元 300 则需要 1500 ns，这是因为存储器需要从单元 100 转动到单元 300，而每转动一个单元需要 5 ns。

存储器层次结构的有效性取决于下面的原则，即并不频繁地将信息移入到快速存储器，在以新信息更新它之前，对其进行频繁的访问。这个原则是源于所谓的局部性现象，即在一个给定的时间里，程序倾向于对存储器相对固定的字段进行重复访问。存在两种形式的局部性：空间局部性和时间局部性。空间局部性是指，当一个给定地址被访问时，则该地址邻近的地址在短时间内极可能被访问，如顺序程序中的连续指令。而时间局部性是指一旦一个特定存储项被访问之后，则该存储项极有可能在下一次被访问，如程序循环中的一条指令。

当处理器请求一个数据项时发生的事件序列如下。首先，该项在存储器层次结构的一级存储器中进行搜索，在一级中找到该请求项的概率称为命中率 h_1。在存储器层次的第一级中没有找到(缺失)该请求项的概率称为缺失率：$(1-h_1)$。当请求项引起“缺失”时，将在下一级存储器中进行搜索。在二级存储中找到该请求项的概率，即是二级命中率 h_2，二级存储的缺失率即是 $(1-h_2)$。重复该过程直至该项被找到，一旦找到请求项，该项即被取出并送到处理器。在一个由三级组成的存储层次中，平均存储器访问时间可以表示如下：

$$\begin{aligned} t_{av} &= h_1 \times t_1 + (1-h_1)[t_1 + h_2 \times t_2 + (1-h_2)(t_2 + t_3)] \\ &= t_1 + (1-h_1)[t_2 + (1-h_2)t_3] \end{aligned}$$

定义对某级存储器的平均访问时间为访问该级一个字所需的时间，在这个方程中，t_1，t_2，t_3 分别表示这三级的访问时间。

6.2　高速缓冲存储器

高速缓冲存储器是由 Wilkes 早在 1965 年提出的，在那时，Wilkes 将主存储器分为两种类型：常规存储器和从存储器。在 Wilkes 的术语中，从存储器是一个由非常规高速存储器构成的二级存储，对应于当今的高速缓冲存储器(cache，本意是隐藏或保存东西的安全地方)。

采用高速缓存作为存储器层次中的第一级，也就是将 CPU 频繁使用的信息保存在高速缓存(毗邻于 CPU 的一小块高速存储)中。最终结果是在任意给定时刻，主存储器中某一活跃部分被复制到高速缓存中。因此，当处理器请求某一存储引用时，该请求首先在高速缓存中进行搜索。若请求对应的元素目前正处在高速缓存中，则称之为缓存命中。若对应于请求的元素当前不在

缓存中,则称之为缓存缺失。高速缓存命中率 h_c 定义为在高速缓存中找到请求元素的概率。高速缓存缺失率$(1-h_c)$则定义为在高速缓存中没有找到请求元素的概率。

当在高速缓存中没有找到请求的元素时,该请求元素将需要从存储器层次的下一级存储器中取出。设该元素存在于下一级存储器,即主存储中,则该元素将被取出并放置到高速缓存中。预期下一请求元素将处于当前请求元素的邻近区域(空间局部性),于是一旦发生高速缓存缺失,则实际从主存储器中取出的是一个元素块,该元素块包含了所请求元素。若用一个主存访问时间可以传送这样一个元素块,则从主存储器传送一个块至高速缓存的优势是最显著的。通过增加主存储器和高速缓存之间信息传送的速率可以达到这一目标,一种增加带宽的技术是使用存储器交叉存取。为了达到最佳效果,设定当发生高速缓存缺失时,从主存储器传送到高速缓存的块是由存储在不同存储器模块上的元素组成的,即连续存储地址处在相邻的存储器模块上。如图 6.2 所示是一个由 8 个存储器模块组成的主存储器的简单示例,在这个例子中一个块是由 8B 组成的。

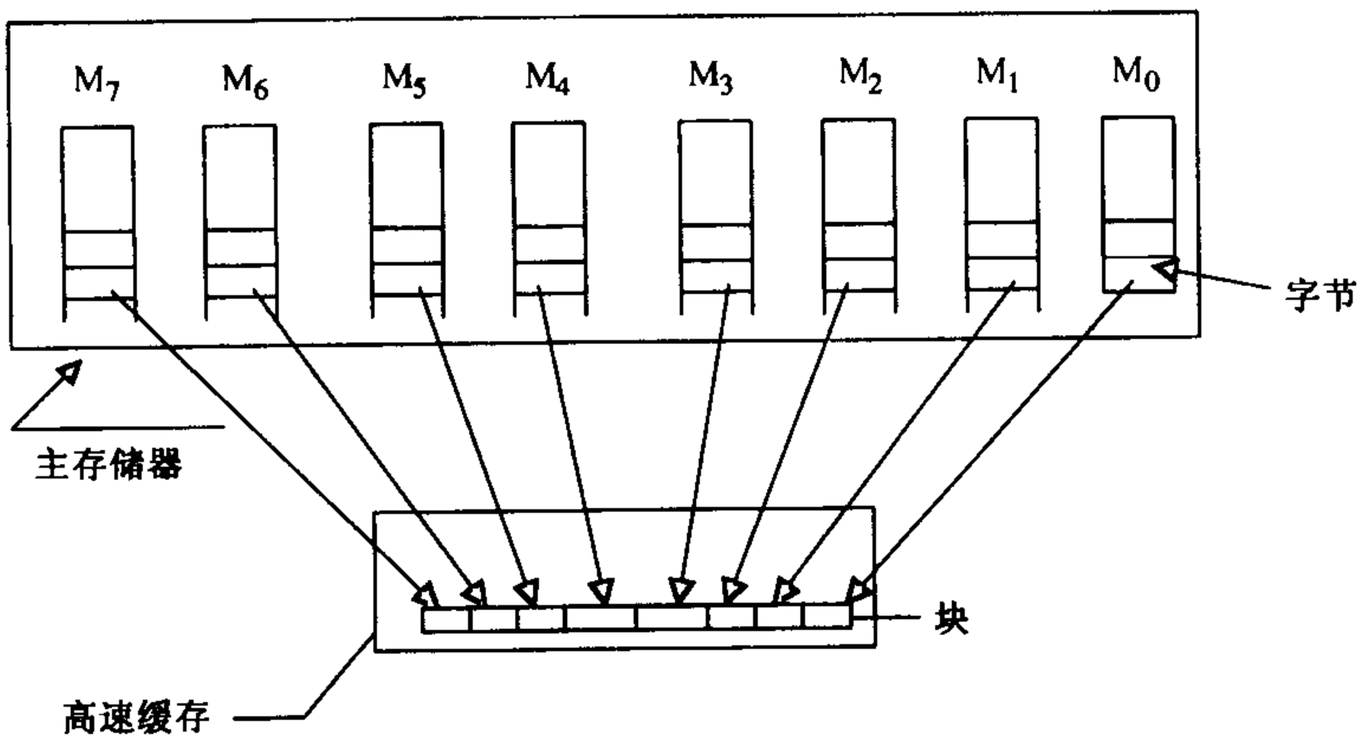


图 6.2 采用 8 个模块的存储器交叉

在介绍了高速缓存使用的基本思想之后,我们将对空间和时间局部性对存储器层次性能的影响进行评估。为了便于分析,我们将限定考虑一种二级体系结构,即高速缓存和主存储器。设主存访问时间是 t_m,高速缓存访问时间是 t_c,以平均访问时间测定局部性的影响,定义平均访问时间为在这样一个二级体系结构中,访问一个处理器请求元素(字)所需的平均时间。

6.2.1 时间局部性影响

这里设在程序循环中需要被执行多次的指令(如为 n 次),一旦加载到高速缓存中,则在被新指令替换前会被使用多次。平均访问时间 t_{av} 由下式给出:

$$t_{av} = \frac{nt_c + t_m}{n} = t_c + \frac{t_m}{n}$$

在推导上面表达式时,设定所请求的存储元素产生了一个高速缓存缺失,因此需要时间 t_m 传送一个主存块。之后,对该请求元素进行 n 次访问,每次需要 t_c。上式表明,随着重复访问次数 n 的增加,平均访问时间将降低,这正是存储器层次的期望特征。

6.2.2 空间局部性影响

在这种情况下,设在发生一次高速缓存缺失时,从主存储器向高速缓存传送的块的大小为

m 个元素，同时还假设由于空间局部性，所有 m 个元素都会被处理器请求，每次一个。根据上面的这些假设，平均访问时间 t_{av} 由下式给出：

$$t_{av} = \frac{mt_c + t_m}{m} = t_c + \frac{t_m}{m}$$

在推导上式时，设定请求的存储元素产生了一次高速缓存缺失，需要时间 t_m 传送主存储器块，该块由 m 个元素组成。之后进行了 m 次访问，每次访问该块的一个元素。上式表明随着块中元素数目 m 的增加，平均访问时间在降低，这正是存储器层次的期望特征。

6.2.3　时间和空间局部性组合的影响

这里假设由于处理器请求的元素产生了一次高速缓存缺失，需要向高速缓存中传送（需要时间 t_m）一个由 m 个元素组成的块。由于空间局部性，处理器每次一个分别请求组成该块的这 m 个元素（需要时间 mt_c）。之后，最初请求的元素被访问（$n-1$）次（时间局部性），即对该元素进行了总数为 n 的访问。基于上面的假定，平均访问时间 t_{av} 可由下式给出：

$$t_{av} = \frac{\left(\frac{mt_c + t_m}{m}\right) + (n-1)t_c}{n} = \frac{t_c + \left(\frac{t_m}{m}\right) + (n-1)t_c}{n} = \frac{t_m}{nm} + t_c$$

若假设 $t_m = mt_c$ 就可将上面表达式进一步简化为

$$t_{av} = \frac{mt_c}{nm} + t_c = t_c + \frac{t_c}{n} = \frac{n+1}{n}t_c$$

上式表示随着重复访问次数 n 的增加，平均访问时间将接近 t_c，这将显著提高性能。

从上面的讨论可以清楚地看到，若越多的请求项不在高速缓存中（缓存缺失），就需要将越多的块调入到高速缓存中。这将会引入两个问题：调入的主存块将放在高速缓存的何处；若缓存已填满，调入的主存块将替换哪个缓存块。调入块的放置以及现有块的替换是根据具体的协议（算法）进行操作的，这些协议与高速缓存的内部组织密切相关，高速缓存的内部组织将在下面的几节中讨论。然而，在讨论高速缓存组织之前，让我们首先介绍一下高速缓存映射函数。

6.2.4　高速缓存映射函数

为不失一般性，我们针对在存储器层次中两个相邻级之间的接口，提出缓存映射函数，其中该相邻级分别是一级和二级。若关注的是高速缓存和主存之间的接口，则高速缓存是一级，而主存是二级，这个原则适用于体系结构中任意两个存储级之间的接口。在下面的讨论中，我们重点分析高速缓存和主存之间的接口。

需要注意的是，对存储单元访问的请求是由处理器通过发出请求单元地址进行的。处理器发出的地址可能对应于当前处在高速缓存中的某一单元（高速缓存命中）；否则，它就对应于当前处在主存中的某一单元。因此，需要通过地址转换确定请求单元所在的位置，这是存储器管理部件（MMU）要完成的一个功能，地址映射函数框图如图 6.3 所示。

在图 6.3 中，系统地址即是由处理器发出的请求单元地址，它由 MMU 中的地址转换函数进行处理。若地址转换发现所请求的地址对应于当前高速缓存中的某一单元，则该单元对处理器就为可用；若该单元当前不在高速缓存中，则它将从主存中取出（作为一个块的部分），并放置到高速缓存中，此时该请求单元就变为对处理器可用。

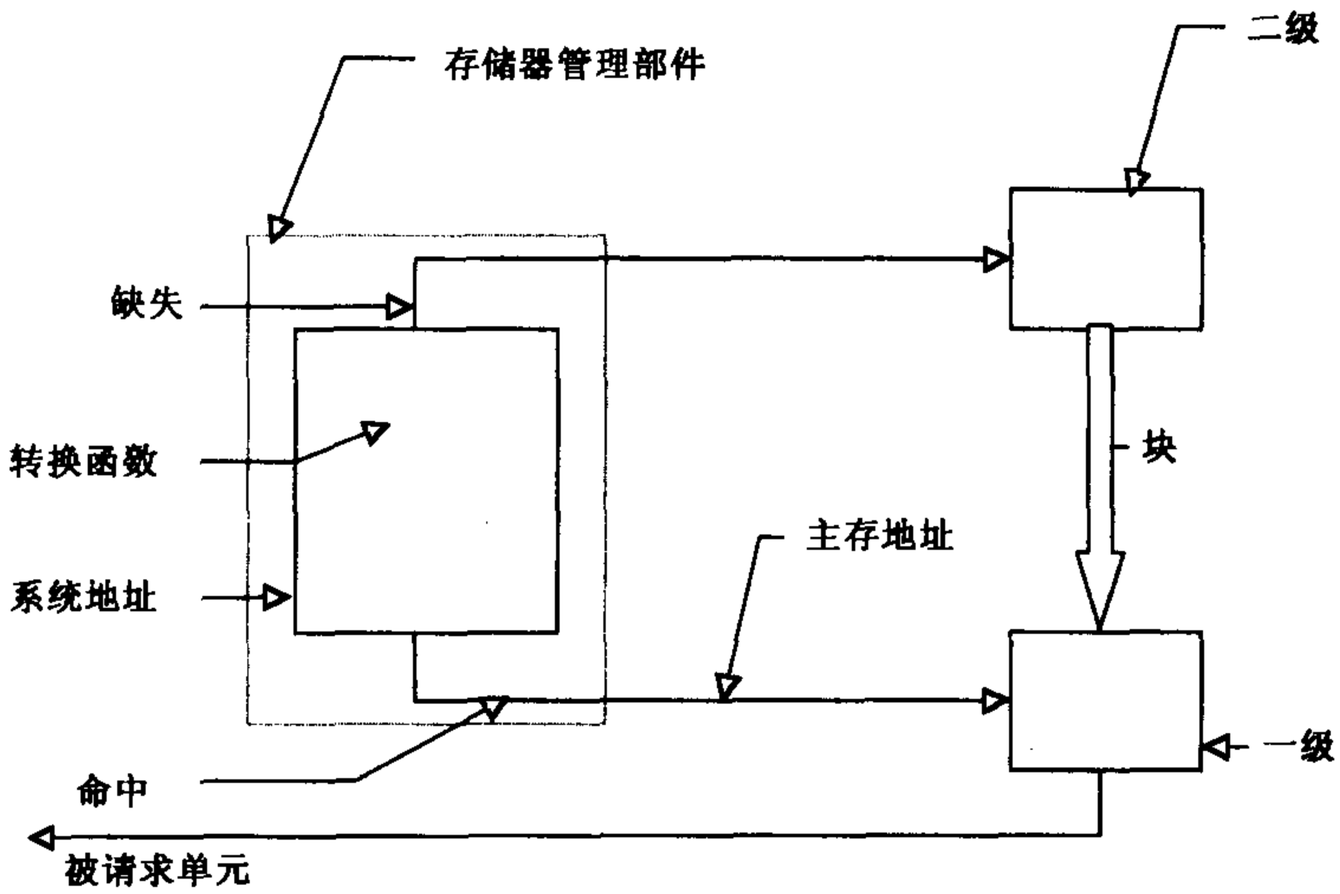


图 6.3　地址映射操作

6.2.5　高速缓存组成

对于高速缓存,存在三种主要的不同组成技术,分别在下面进行讨论,这些技术的不同之处主要表现在以下两个方面:

1. 从主存调入的块在高速缓存中存放的准则;

2. 调入块替换高速缓存块的准则(在高速缓存已满的情况下)。

直接映射。这是三种技术中最简单的一种,它的简单性源于它将主存调入块放置在一个特定的固定高速缓存块位置,这种放置是基于以下三者的固定的关系:调入块编号 i、高速缓存块编号 j 和高速缓存块总数 N,其关系为如下:

$$j = i \bmod N$$

例 6.1　考虑如下例子,即主存由 4 K 个块组成,高速缓存由 128 个块组成,块的大小为 16 个字。图 6.4 显示了根据直接映射高速缓存技术对主存和高速缓存进行划分的情况。

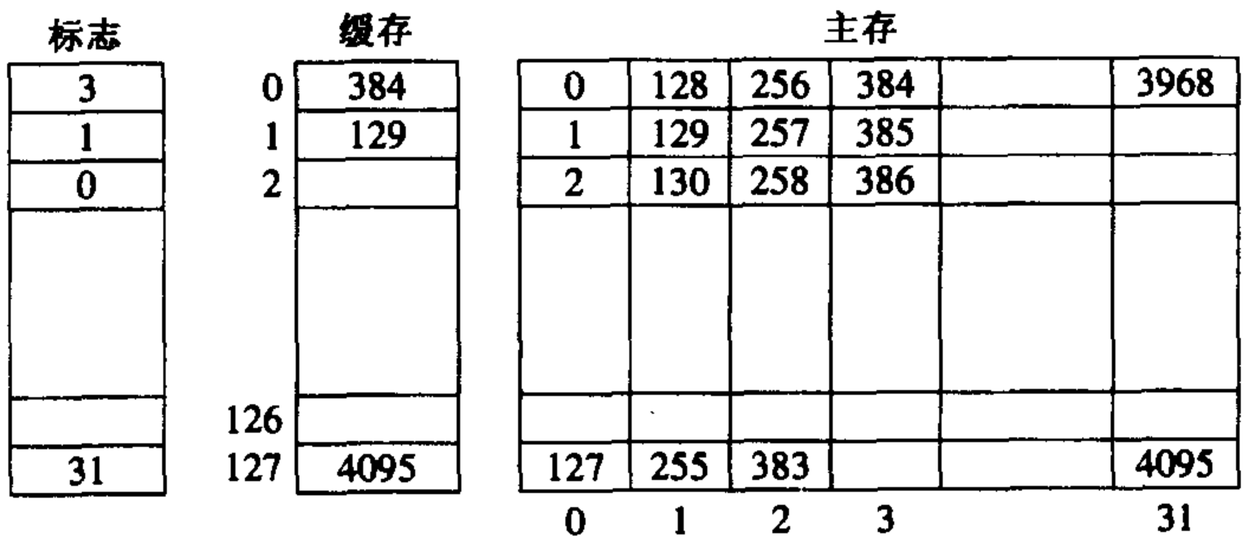


图 6.4　映射主存块至高速缓存块

如图所示,每 32 个主存块映射到同一个给定的高速缓存块,例如,主存块 0,128,256,384,…,3968 均映射到高速缓存块 0。因此,称这种直接映射技术为多对一映射技术。直接映射技术的优点在于它确定调入主存块在高速缓存中放置位置的简单性,它的不足在于对高速缓存利用

率的不足。这是因为根据该项技术，多个主存块可能对应同一给定的高速缓存块，因而发生冲突，即使还有其他空闲的高速缓存块。该不足导致了高速缓存命中率的低下。

根据直接映射技术，MMU 将处理器发出的地址解释为如图 6.5 所示的三个字段，图中每个字段的位长如下：

1. 字字段 = lb B，其中 B 是以字为单位时块的大小；
2. 块字段 = lb N，其中 N 是以块为单位时高速缓存的大小；
3. 标志字段 = lb(M/N)，其中 M 是以块为单位时主存的大小；
4. 主存地址的位数 = lb($B \times M$)。

值得注意的是，以上前三项计算得到的位总和应等于主存地址的长度，这个性质可以作为对计算正确性的检查。

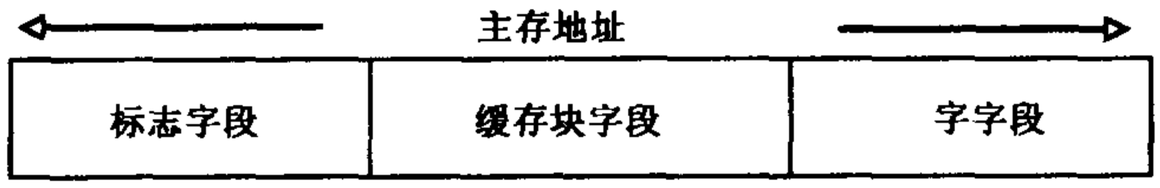


图 6.5　直接映射地址字段

例 6.2　计算例 6.1 中所对应的上述 4 个参数：

字字段 = lb$B$ = lb 16 = lb 2^4 = 4 位

块字段 = lb$N$ = lb 128 = lb 2^7 = 7 位

标志字段 = lb(M/N) = lb($2^2 \times 2^{10}/2^7$) = 5 位

主存地址的位数为 lb($B \times M$) = lb($2^4 \times 2^{12}$) = 16 位

在说明主存地址划分之后，现在说明 MMU 所采用的协议，通过该协议以响应处理器访问给定单元的请求，在说明该协议时要用到上面例子中的参数，如图 6.6 所示。

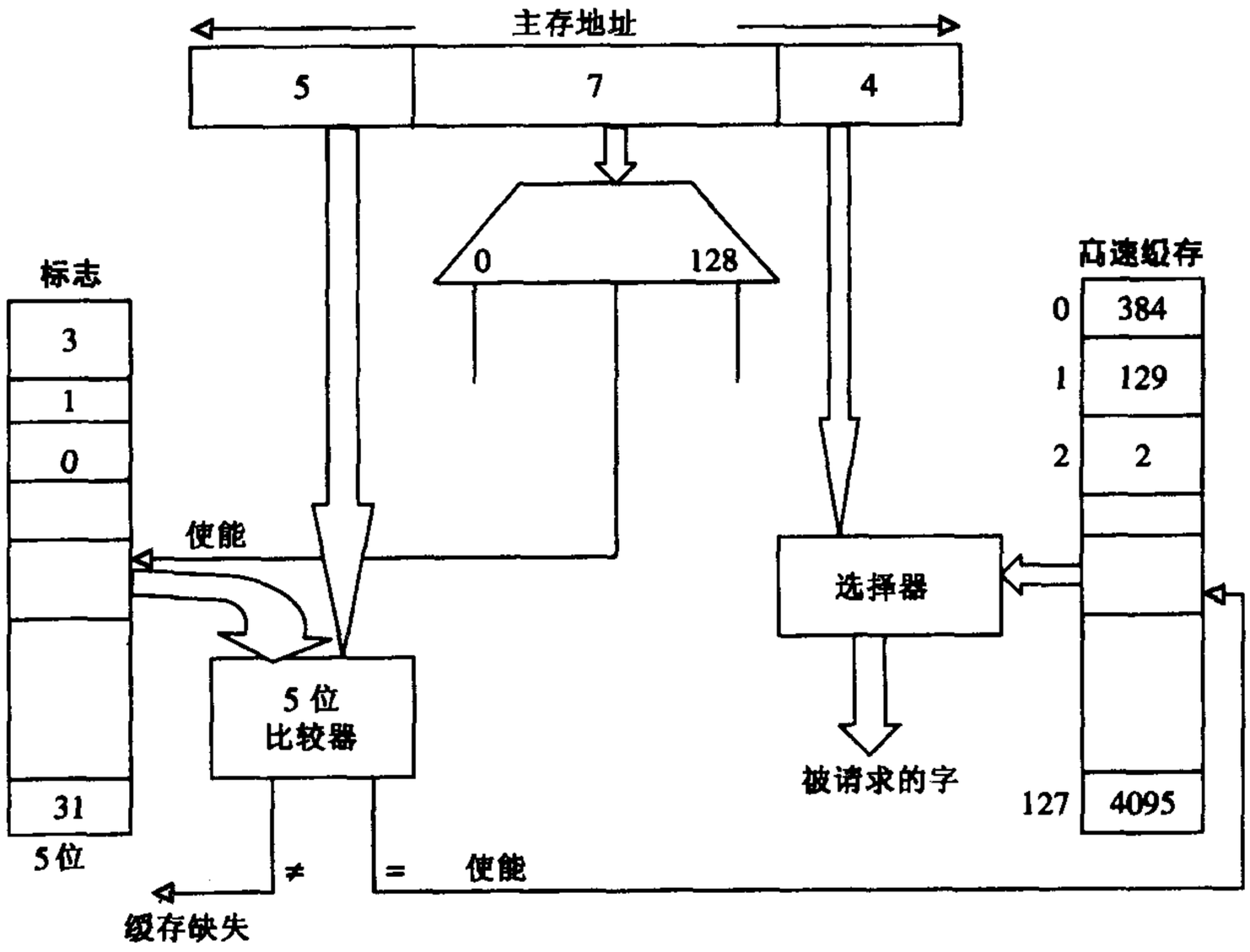


图 6.6　直接映射地址转换

协议过程如下：

1. 通过块字段确定处理器请求的单元所在的高速缓存块。因为块字段直接用来确定高速缓存块搜索，因此这项技术称为直接映射；
2. 检查相应的标志存储区，以确定其内容是否与标志字段相匹配。两者相匹配说明步骤 1 中确定的目标高速缓存块当前正保存了处理器请求的主存单元，即高速缓存命中；
3. 通过字字段在确定的高速缓存块中选定目标单元；
4. 若步骤 2 中没有找到匹配项，则高速缓存缺失。因此，请求的块需要从主存中调入到高速缓存中，并将目标单元设为对处理器可用。同时，标志存储区和高速缓存块存储区需要进行相应的更新。

直接映射技术不仅解决了高速缓存问题中调入主存块的放置位置，同时还解决了替换问题。当高速缓存已填满，而此时又必须调入一个新的主存块时，块的替换将直接由方程 $j = i \bmod N$ 确定。

直接映射技术的主要优点在于它直接确定高速缓存块的简洁性，无需搜索，在替换机制上也简单。直接映射的主要缺点是高速缓存预期利用率差，主要体现在给定高速缓存块命中率上，需要经常替换一些高速缓存块，而其余的高速缓存块却没有使用。考虑下面的例子，处理器请求的单元序列保存在主存块 1,33,65,97,129 和 161 上，若高速缓存大小是 32 个块，则所有上述块均映射到高速缓存块 1 上，因此这些块将竞争同一高速缓存块，而其余 31 个高速缓存块却没有使用。

直接映射技术高速缓存预期利用率差的原因主要是由于对调入主存块在高速缓存中放置位置的限制(多对一的属性)。若放松该限制，即可以将调入主存块放置到任意空闲(可用)的高速缓存块中，则所导致的相应技术的灵活性就较好，从而使高速缓存的高效利用成为可能。这种灵活的技术称为相联映射技术，将在下面介绍。

全相联映射。根据该技术，一个调入主存块可以放置到任意的高速缓存块上。因此，处理器发出的地址只包含两个字段，它们是标志和字字段。第一个字段惟一标明块，同时该字段也驻留在高速缓存中，第二个字段标明处理器请求块中的单元。MMU 通过将处理器发出的地址分为两部分进行解释，这两个字段如图 6.7 所示。图中每个字段的位长如下所示：

1. 字字段 = lb$B$，其中 B 是以字为单位时块的大小；
2. 标志字段 = lb$M$，其中 M 是以块为单位时主存的大小；
3. 主存地址的位数为 lb($B \times M$)。

需要注意的是，上述前两式计算得到的位数总和必须等于主存地址的长度，该性质可以用来检查计算时的正确性。

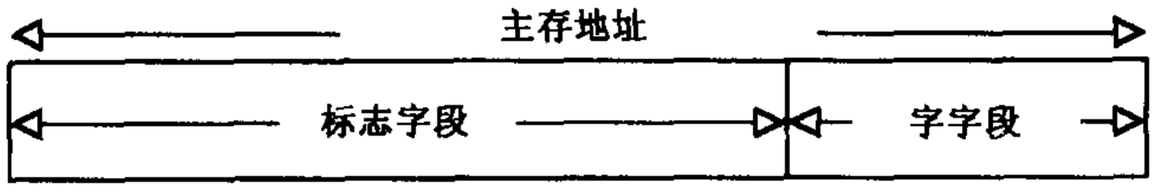


图 6.7　相联映射地址字段

例 6.3　为了确定一个存储系统的上述三个参数，需要做如下指定：主存大小为 4 K 块，高速缓存大小为 128 块，每个块的大小是 16 个字。设该系统采用相联映射，则：

字字段 = lb$B$ = lb 16 = lb 2^4 = 4 位

标志字段 = $\mathrm{lb}M = \mathrm{lb}(2^2 \times 2^{10}) = 12$ 位

主存地址位数为 $\mathrm{lb}(B \times M) = \mathrm{lb}(2^4 \times 2^{12}) = 16$ 位

在说明了主存地址的划分之后，进一步介绍针对处理器请求访问一给定单元，MMU 所采用的协议，在说明该协议时用到了上面例子中的参数(见图 6.8)，该协议的步骤如下：

1. 根据标志字段在标志存储区进行搜索，以确定一个匹配的存储标志；
2. 标志存储区中存在一个匹配，表明由步骤 1 确定的对应目标高速缓存块当前保存了处理器所求的主存单元，即高速缓存命中；
3. 在高速缓存块包含的单元中，通过字字段选定目标单元；
4. 若步骤 2 中没有匹配，即发生高速缓存缺失。因此需要从主存中调入请求块，并放置到第一个可用的高速缓存块中，且将目标单元(字)设为对处理器可用，同时标志存储区和高速缓存块存储区需要进行相应的更新。

需要注意的是，上述步骤 1 中的搜索需要将地址中的标志字段与标志存储区中的每一项进行匹配，若串行执行该搜索，则会导致相当长的延时。为此，标志被存放在相联存储区中(按内容访问)，从而可以对标志存储区中的所有项进行并行搜索(相联式地)，因此这种技术称之为相联映射。

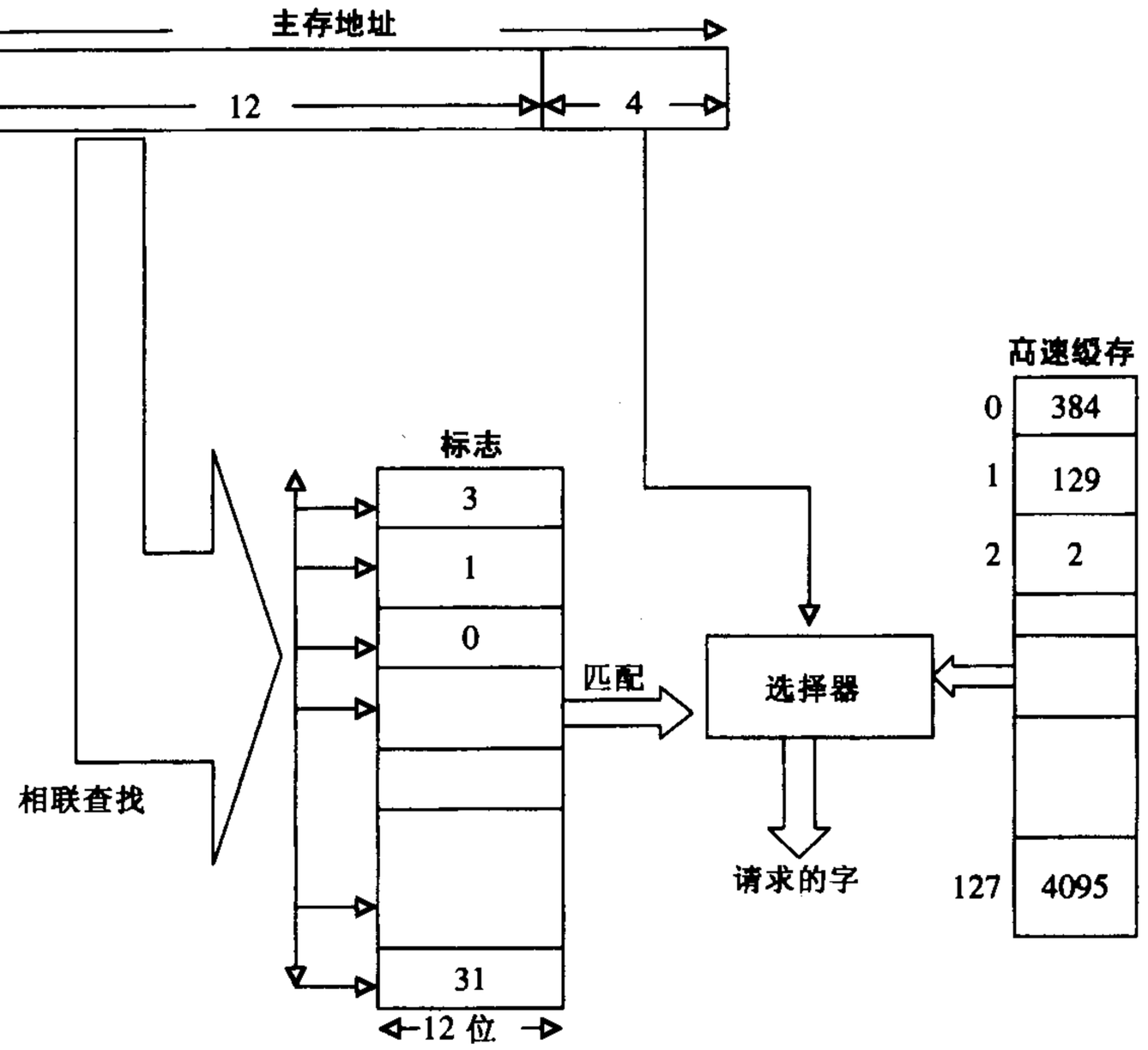


图 6.8　相联映射地址转换

还需要注意的是，尽管用到了高速缓存组织，但还需要一种机制以确保所访问的高速缓存块中包含的是有效信息。高速缓存块中信息的有效性可以通过对为每个高速缓存块所设置的、称为有效位的一位进行检查。高速缓存块有效位是以下面的方式进行更新的，即若有效位为 1，则相应的高速缓存块包含了有效信息，否则，高速缓存块中的信息为无效。当计算机系统启动时，所有有效位均设为 0，表示它们包含的是无效信息。当数据块调入到高速缓存中后，它们的状态会发生相应的变化，以反映它们所含信息的有效性。

相联映射技术的主要优点在于能高效地利用高速缓存,这主要是因为它对放置调入主存块没有限制,任何没有被占用的高速缓存块均可用来接收那些调入的主存块。它的主要缺点是执行相联查找所需的硬件开销,即上面讨论的在标志字段和标志存储区之间的匹配查找。

在简单但效率不高的直接高速缓存组织和烦琐但高效的相联高速缓存组织之间的一种折中方法是,在高速缓存的局部范围进行搜索,同时提前知道一个调入主存块在缓存中放置的位置,这就是下面要介绍的组相联映射技术的基础。

组相联映射。在组相联映射技术中,高速缓存被分为一些组,每个组包含一定数量的块。一个给定的主存块基于下面公式被映射到一个特定的高速缓存组上,该公式即 $s = i \bmod S$,其中,S 是高速缓存中组的数目,i 是主存块的编号,s 是块 i 被映射的特定高速缓存组。然而,一个调入的块可以被映射到给定高速缓存组中的任一块上。因此,处理器发出的地址被分为三个不同的字段,它们是标志字段、组字段和字字段。组字段用来惟一地指明目标块所对应的特定高速缓存组,标志字段用来惟一指明在高速缓存组中的目标块,字字段则在高速缓存块中指明处理器所请求的单元(字)。根据组相联映射技术,MMU 通过将处理器发出的地址分为三个字段进行解释,这三个字段如图 6.9 所示,图中每个字段的位长度如下所示:

1. 字字段 = lb B,其中 B 是以字为单位时块的大小;
2. 组字段 = lb S,其中 S 是高速缓存中组的数目;
3. 标志字段 = lb(M/S),其中,M 是以块为单位时的主存大小,$S = N/B_s$,其中,N 是高速缓存块的数量,而 B_s 是每个组包含的块数;
4. 主存地址的位数为 lb($B \times M$)。

需要注意的是,上述三式计算得到的位应与主存地址位的长度相等,这可以用来检查计算的正确性。

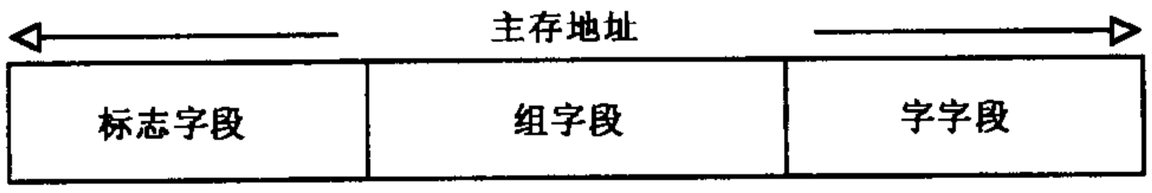


图 6.9 组相联映射地址字段

例 6.4 为计算存储系统的上述 3 个参数(字、组和标志)需要做如下指定:主存大小是 4 K 个块,高速缓存大小是 128 个块,每个块的大小是 16 个字。设系统采用组相联映射,其中每个组有 4 个块。

$$S = \frac{128}{4} = 32 \text{ 组}$$

1. 字字段 = lb B = lb 16 = lb 2^4 = 4 位
2. 组字段 = lb 32 = 5 位
3. 标志字段 = lb($4 \times 2^{10}/32$) = 7 位

主存地址位 = lb($B \times M$) = lb($2^4 \times 2^{12}$) = 16 位。

在说明主存地址的划分之后,将介绍 MMU 所采用的协议,通过该协议响应处理器发出的访问某一单元的请求,在说明该协议时用到了上面例子中的相关参数,如图 6.10 所示,这个协议的步骤如下:

1. 用组字段(5 位)确定(直接地)一个特定的组(32 个组中的一个)。
2. 用标志字段在已确定的组中找到一个匹配块(4 个块中的一个),标志存储区的匹配表明步骤 1 所确定的特定组中当前包含了目标块,即高速缓存命中。
3. 在命中的高速缓存块所包含的 16 个字(单元)中,用字字段选定所请求的字。
4. 若步骤 2 中没有找到匹配项,则说明高速缓存缺失,因此,请求块需要从主存中调入,并先将其存在指定组中,然后将目标单元(字)设为对处理器可用。与此同时,高速缓存标志存储区和高速缓存块存储区必须进行相应的更新。

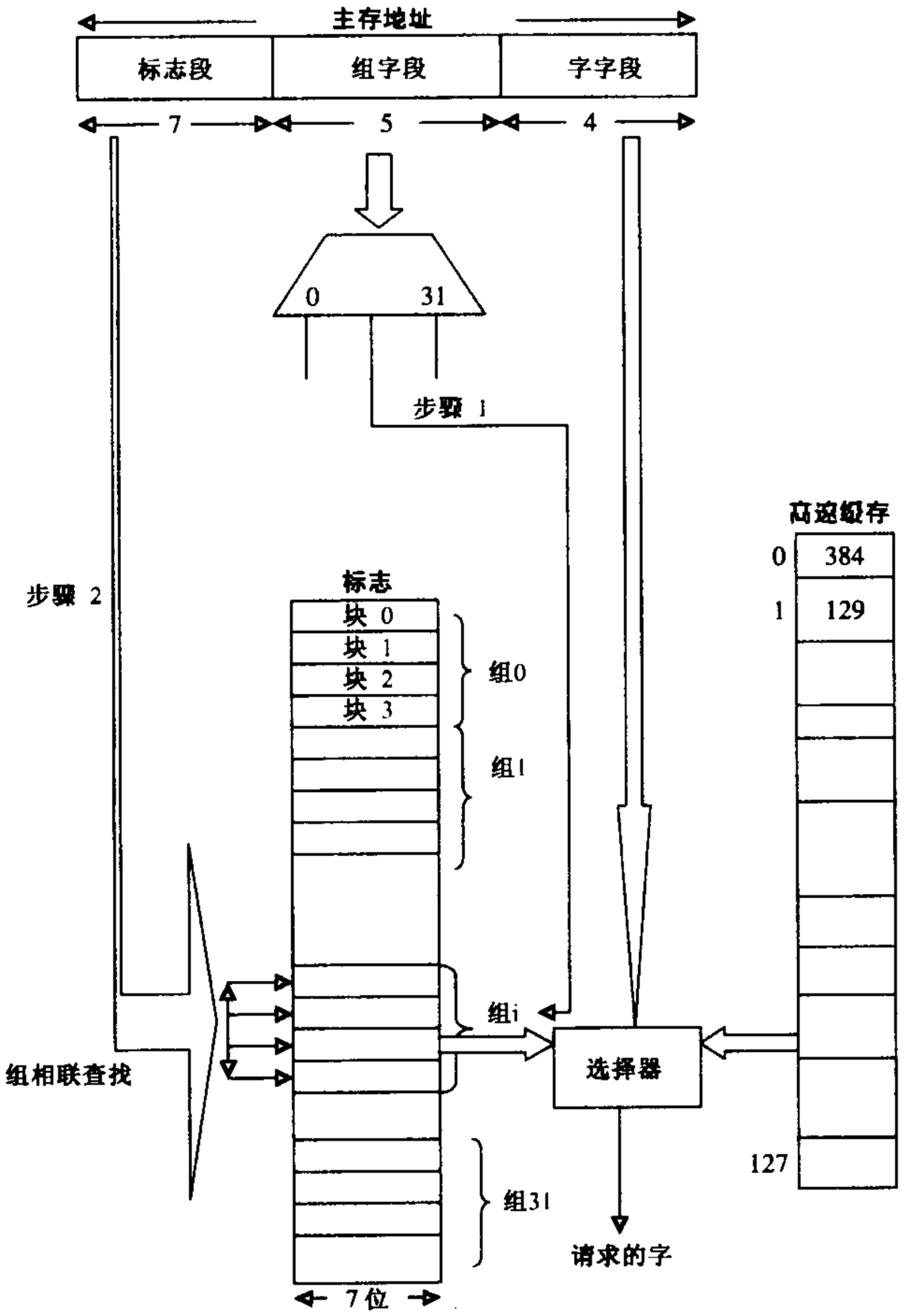


图 6.10　组相联映射地址转换

需要注意的是,在上述步骤 2 中的搜索需要将地址中的标志字段与特定组标志存储区中的每一项进行匹配,这个搜索在该组内并行进行(相联地),因此称之为组相联映射。在一个组内执行相联搜索,以在标志字段和标志存储区间找到一个匹配,但其所需的硬件开销没有全相联技术那样复杂。

组相联映射技术是希望产生一个较高效的高速缓存利用率,即没有全相联技术那样高效,但不会像直接映射技术那样差。然而,这种技术承袭了直接映射技术在确定目标组时的简洁性。

表 6.2 对这三种映射技术进行了全面的定性比较,由于组相联技术中等程度的复杂性和较好的高速缓存利用率,该项技术已在 Intel 奔腾系列处理器中采用。

表 6.2 高速缓存映射技术之间的定性比较

映射技术	简易性	相联标志查找	预期的高速缓存利用率	替换技术
直接	有	无	低	不需要
全相联	无	有	高	是
组相联	中等	中等	中等	是

上面的讨论中已说明了相联映射和组相联技术如何将调入的主存块放置到高速缓存中,在开始讨论高速缓存时涉及的另一个重要问题是替换。具体来讲,当高速缓存已被填满,同时一个新的主存块需要调入,哪个块需要选定以替换掉,这将在下面讨论。

6.2.6 替换技术

已有的一些替换技术包括随机选取块(随机选取)、在高速缓存中时间最长的块(先进先出,FIFO)、在高速缓存中最少被使用的块(最近最少使用,LRU)。

假设当计算机系统接通电源时,一个随机数字生成器开始产生 0 至($N-1$)之间的数字。正如名字所示,这种方法随机地选取一个高速缓存块加以替换,完全是根据替换时随机数字生成器的输出进行。这种技术简单且无需额外的开销,然而,它的主要缺点是没有考虑局部性。随机方法已被证实有足够的效率,首先由 Intel 应用在它的 iAPX 微处理器系列中。

FIFO 技术将一个块在高速缓存中存在的时间作为替换衡量标准,在高速缓存中存在最久的块将被选为替换对象,而不管当前对该块的访问情况。这种技术需要对高速缓存块的生命周期进行跟踪,因此不会像随机选择技术那样简单。直观上来讲,FIFO 技术适用于顺序执行程序,因为它不关心引用的局部性。

根据 LRU 替换技术,最近最少使用的高速缓存块将被替换。在这三种替换技术中,LRU 技术是最高效的,这是因为它考虑了块利用的历史(作为替换标准)。LRU 算法需要用到高速缓存控制电路以跟踪所有高速缓存驻留块的引用,已有一些实现方法。在这些实现方法中,可以采用计数器法。在这种方法里,为每个高速缓存块分配一个计数器,发生高速缓存命中时,相应块的计数器被置为 0,所有其他比命中高速缓存计数器中原值小的计数器均加 1,而那些已有更大值的计数器则保持其值不变。而发生高速缓存缺失时,计数器值最大的高速缓存块被替换,同时该计数器被置为 0,而所有其他计数器均加 1。

在介绍了上述 3 种高速缓存映射技术之后,我们将通过下面的例子说明关于这三种技术的主要观察结果。

例 6.5 考虑一个 4×8 两维数组 A。设数组中的每个元素占一个字,数组元素按列为主的顺序保存在主存的 1000 至 1031 单元,高速缓存由 8 个块组成,每个块正好 2 个字。此外还假设总是采用 LRU 替换策略。设处理器将按下列顺序对数组元素加以请求,我们将考察分别采用上述 3 种映射技术时在高速缓存内的变化。

$$a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}, a_{0,4}, a_{0,5}, a_{0,6}, a_{0,7}$$
$$a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}, a_{1,5}, a_{1,6}, a_{1,7}$$

解决方案。数组元素在主存中的分布如图 6.11 所示,该图同时也给出了在上述请求之前的高速缓存状态。

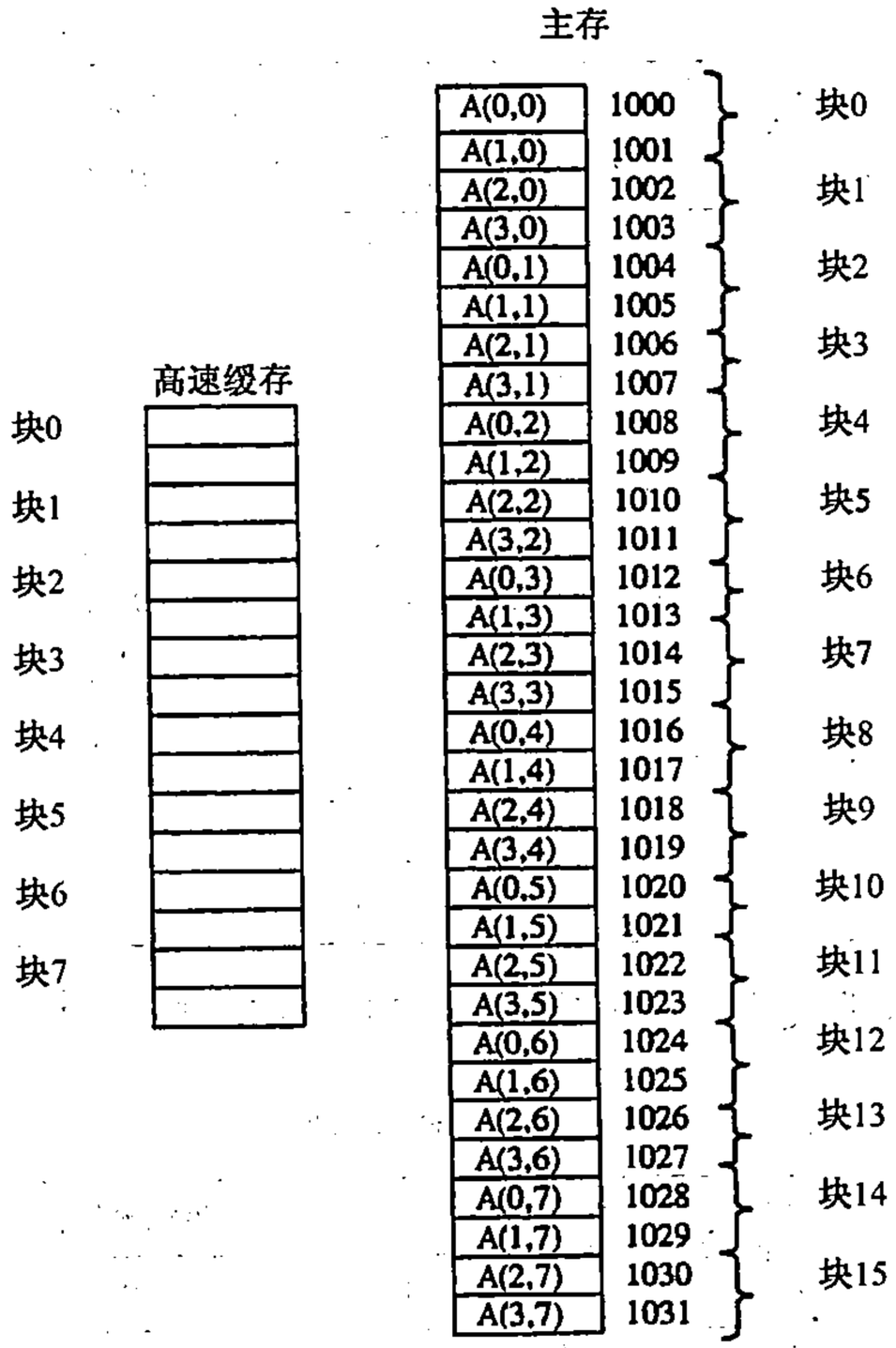


图 6.11　主存中的数组元素

直接映射。表 6.3 表明发生了 16 次高速缓存缺失(非单次高速缓存命中),替换数为 12(阴影显示的)。同时表明在这 8 个可用的高速缓存块中,仅有 4 个(0,2,4,6)被用到,而其余的 4 个在所有时间全部是非活动的,这表明高速缓存利用率为 50%。

表 6.3　直接映射

请求	高速缓存命中/缺失	MM 块号 (i)	高速缓存块号 (j)	高速缓存状态 BL0	BL1	BL2	BL3	BL4	BL5	BL6	BL7
A(0,0)	缺失	0	0	0 1 0 0							
A(0,1)	缺失	2	2	0 1 0 0		0 1 1 1					
A(0,2)	缺失	4	4	0 1 0 0		0 1 1 1		0 1 2 2			
A(0,3)	缺失	6	6	0 1 0 0		0 1 1 1		0 1 2 2	0 1 3 3		
A(0,4)	缺失	8	0	[illegible]		0 1 1 1		0 1 2 2	0 1 3 3		
A(0,5)	缺失	10	2	0 1 4 4		[illegible]		0 1 2 2	0 1 3 3		
A(0,6)	缺失	12	4	0 1 4 4		0 1 5 5		[illegible]	0 1 3 3		
A(0,7)	缺失	14	6	0 1 4 4		0 1 5 5		0 1 6 6	[illegible]		

续表

请求	高速缓存命中/缺失	MM块号(i)	高速缓存块号(j)	高速缓存状态 BL0	BL1	BL2	BL3	BL4	BL5	BL6	BL7
A(1,0)	缺失	0	0	0 1 0 0		0 1 5 5		0 1 6 6		0 1 7 7	
A(1,1)	缺失	2	2	0 1 0 0		0 1 1 1		0 1 6 6		0 1 6 6	
A(1,2)	缺失	4	4	0 1 0 0		0 1 1 1		0 1 2 2		0 1 6 6	
A(1,3)	缺失	6	6	0 1 0 0		0 1 1 1		0 1 2 2		0 1 3 3	
A(1,4)	缺失	8	0	0 1 4 4		0 1 1 1		0 1 2 2		0 1 3 3	
A(1,5)	缺失	10	2	0 1 4 4		0 1 5 5		0 1 2 2		0 1 3 3	
A(1,6)	缺失	12	4	0 1 4 4		0 1 5 5		0 1 6 6		0 1 3 3	
A(1,7)	缺失	14	6	0 1 4 4		0 1 5 5		0 1 6 6		0 1 7 7	

全相联映射。表 6.4 表明发生了 8 次高速缓存命中(所有请求的 50%),没有发生替换,即 100%的高速缓存利用率。

表 6.4 全相联映射

请求	高速缓存命中/缺失	MM块号(i)	高速缓存块号	高速缓存状态 BL0	BL1	BL2	BL3	BL4	BL5	BL6	BL7
A(0,0)	缺失	0	0	0 1 0 0							
A(0,1)	缺失	2	1	0 1 0 1 0 0 1 1							
A(0,2)	缺失	4	2	0 1 0 1 0 1 0 0 1 1 2 2							
A(0,3)	缺失	6	3	0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3							
A(0,4)	缺失	8	4	0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4							
A(0,5)	缺失	10	5	0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5							
A(0,6)	缺失	12	6	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6							
A(0,7)	缺失	14	7	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7							
A(1,0)	命中	0	0	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7							
A(1,1)	命中	2	1	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7							
A(1,2)	命中	4	2	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7							
A(1,3)	命中	6	3	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7							
A(1,4)	命中	8	4	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7							
A(1,5)	命中	10	5	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7							
A(1,6)	命中	12	6	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7							
A(1,7)	命中	14	7	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7							

组相联映射(每组两个块)　表 6.5 表明共发生了 16 次高速缓存缺失(非单次的高速缓存命中),替换数是 12(阴影显示的)。同时显示,4 个可用的高速缓存组中只用到了 2 个组,而另外 2 个组在所有时间全部处于非活动状态,这表明高速缓存利用率是 50%。

表 6.5　组相联映射

请求	高速缓存命中/缺失	MM 块号 (i)	高速缓存块号	缓存状态 Set # 0 BL0	Set # 0 BL1	Set # 1 BL2	Set # 1 BL3	Set # 2 BL4	Set # 2 BL5	Set # 3 BL6	Set # 3 BL7
A(0,0)	缺失	0	0	0	1						
				0	0						
A(0,1)	缺失	2	2	0	1					0	1
				0	0					1	1
A(0,2)	缺失	4	0	0	1	0	1			0	1
				0	0	2	2			1	1
A(0,3)	缺失	6	2	0	1	0	1			0	1
				0	0	2	2			1	1
A(0,4)	缺失	8	0	0	1	0	1			0	1
				4	4	2	2			1	1
A(0,5)	缺失	10	2	0	1	0	1			0	1
				4	4	2	2			5	5
A(0,6)	缺失	12	0	0	1	0	1			0	1
				4	4	6	6			5	5
A(0,7)	缺失	14	2	0	1	0	1			0	1
				4	4	6	6			5	5
A(1,0)	缺失	0	0	0	1	0	1			0	1
				0	0	6	6			5	5
A(1,1)	缺失	2	2	0	1	0	1			0	1
				0	0	6	6			1	1
A(1,2)	缺失	4	0	0	1	0	1			0	1
				0	0	2	2			1	1
A(1,3)	缺失	6	2	0	1	0	1			0	1
				0	0	2	2			1	1
A(1,4)	缺失	8	0	0	1	0	1			0	1
				4	4	2	2			1	1
A(1,5)	缺失	10	2	0	1	0	1			0	1
				4	4	2	2			5	5
A(1,6)	缺失	12	0	0	1	0	1			0	1
				4	4	6	6			5	5
A(1,7)	缺失	14	2	0	1	0	1			0	1
				4	4	6	6			4	4

6.2.7　高速缓存写策略

在讨论了高速缓存映射技术和替换策略之后,我们进一步讨论一个相关的重要问题,即高速缓存一致性。在任何时候任何情况下,都要保证高速缓存字与其在主存中副本的一致性,已有一些策略(技术)用于向主存块中执行写操作,而这些块又同时驻留在高速缓存中。这些策略决定了高速缓存字与其在主存中对应部分的一致性程度。下面的段落将讨论这些写策略。将特别讨论两种主要情况:高速缓存命中时的高速缓存写策略和高速缓存缺失时的高速缓存写策略。同时,我们还将讨论高速缓存缺失时的高速缓存读策略,而高速缓存命中时的高速缓存读是很简单的。

高速缓存命中时的高速缓存写策略。在高速缓存命中时存在两种可能的写策略,即通写和回写。在通写策略中,在对高速缓存进行写操作的同时,也向主存中写入。在回写策略中,所有

写操作是针对高速缓存的,对主存的写操作被延迟至当替换发生时。为每个高速缓存块分配一位,称之为脏位,以指示对驻留在高速缓存中的该高速缓存块已至少进行了一次写操作。在替换时,检查该脏位,若该位已被置位,则将该块写回到主存中,否则直接用调入块覆盖它。通写策略为了保持高速缓存块和它们在主存中对应部分的一致性,相应的写主存操作需要额外时间,从而增加了其平均访问时间。回写策略可以消除平均访问时间这方面的开销,但是一致性仅在替换时得到保证。

高速缓存缺失时的高速缓存写策略。有两种模式可以使用。写 - 分配模式将主存块调入到高速缓存后再将其更新,而写 - 非分配模式则将缺失的主存块在主存中更新,但不调入高速缓存。

通常情况下,通写高速缓存采用写 - 非分配策略,而回写高速缓存则采用写 - 分配策略。

高速缓存缺失时的高速缓存读策略。有两种可用的策略。其一是,缺失主存块被调入到高速缓存中同时被请求的字尽可能快地立即发送到 CPU;其二是,缺失主存块先整体调入到高速缓存中,然后再将请求的字发送到 CPU。

在讨论了高速缓存设计与分析的相关问题后,针对不同高速缓存写策略下存储器层次的平均访问时间,这里简要地给出相关的公式。

情况 1:高速缓存通写策略

写 - 分配。在这种情况下,存储系统的平均访问时间如下:

$$t_a = t_c + (1 - h)t_b + w(t_m - t_c)$$

其中,t_b 表示传送一个块至高速缓存所需的时间,$(t_m - t_c)$表示由于写操作所需的额外时间,w 是写操作所占的比例。需要注意的是,若数据通路和组织结构允许,则有 $t_b = t_m$,否则 $t_b = Bt_m$,其中 B 是以字为单位时块的大小。

写 - 非分配。在这种情况下,平均访问时间如下:

$$t_a = t_c + (1 - w)(1 - h)t_b + w(t_m - t_c)$$

情况 2:高速缓存回写策略

采用回写策略时系统的平均访问时间为:$t_a = t_c + (1 - h)t_b + w_b(1 - h)t_b$,其中 w_b 是高速缓存驻留块需要回写(即已被改变)的概率。

6.2.8 实际高速缓存结构分析

Intel 奔腾 IV 处理器高速缓存。Intel 奔腾 IV 处理器采用两级高速缓存结构,如图 6.12 所示。在图 6.12 中,L1 表示一个 8 KB 的数据高速缓存,是 4 路的组相联,块的大小是 64 B。考虑下面的例子(用于奔腾 L1 高速缓存之后)。

例 6.6 下表为一个 Intel 奔腾 IV 处理器的配置参数:

高速缓存结构	组相联
主存大小	16 MB
高速缓存 L1 大小	8 KB
每个组中的高速缓存块数	4
CPU 寻址	字节可寻址

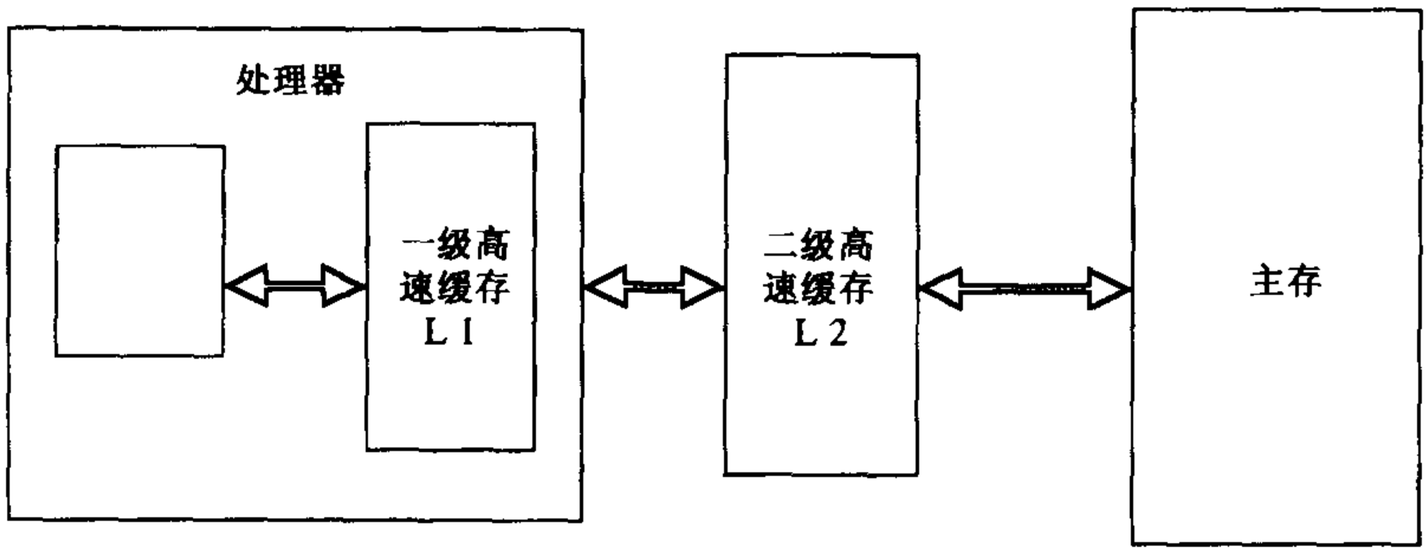


图 6.12　奔腾 IV 的两级高速缓存

主存地址被分为 3 个字段:字、组和标志,如图 6.13 所示,每个字段的长度计算如下:

主存块数 $M = 2^{24}/2^6 = 2^{18}$ 块
高速缓存块数 $N = 2^{13}/2^6 = 128$ 块
$S = 128/4 = 32$ 组
组字段 = lb 32 = 5 位
字字段 = lb B = lb 64 = lb 2^6 = 6 位
标志字段 = lb($2^{18}/2^5$) = 13 位
主存地址 = lb($B \times M$) = lb($2^6 \times 2^{18}$) = 24 位

在图 6.12 中的二级高速缓存是 L2,称为先进传递高速缓存,为 8 路组相联高速缓存,总大小为 256 KB,每个块为 128 B,和上面 L1 级步骤类似,有:

主存块数 $M = 2^{24}/2^7 = 2^{17}$ 块
高速缓存块数 $N = 2^{18}/2^7 = 2^{11}$ 块
$S = 2^{11}/2^3 = 2^8$ 组
组字段 = lb 2^8 = 8 位
字字段 = lb B = lb 128 = lb 2^7 = 7 位
标志字段 = lb($2^{17}/2^8$) = 9 位

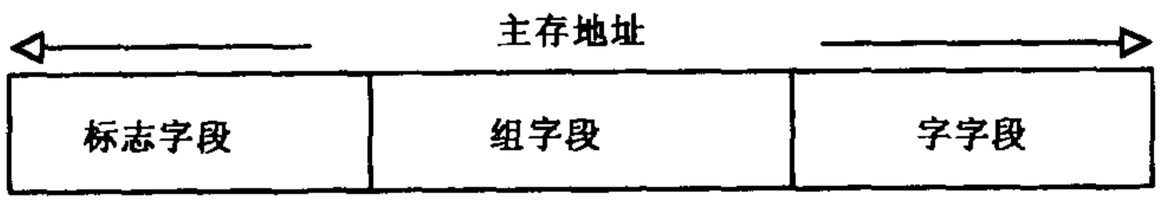


图 6.13　主存地址的划分

下表对奔腾 IV 的 L1 和 L2 两级高速缓存在高速缓存命中率和高速缓存延时这两方面的性能做了一个小结:

CPU	L1 命中率	L2 命中率	L1 时延	L2 时延	平均时延
奔腾 IV 1.5 GHz	90%	99%	1.33 ns	6.0 ns	1.8 ns

PowerPC 604 处理器高速缓存。PowerPC 高速缓存分为数据高速缓存和指令高速缓存,称为哈佛结构。指令高速缓存和数据高速缓存均是 16 KB 的 4 路组相联,下表是 PowerPC 604 高速缓存基本特征的小结。

高速缓存结构	组相联
块大小	32 字节
主存大小	4 GB(M = 128 兆块)
高速缓存大小	16 KB(N = 512 块)
每个组中的块数	4
高速缓存组数(S)	128 组

主存地址可以分为 3 个字段:字、组和标志(见图 6.13),每个字段长度的计算情况如下:

主存块数 $M = 2^{32}/2^5 = 2^{27}$ 块

高速缓存块数 $N = 2^{14}/2^5 = 512$ 块

$S = 512/4 = 128$ 组

组字段 = lb 128 = 7 位

字字段 = lb B = lb 32 = lb 2^5 = 5 位

标志字段 = lb($2^{27}/2^7$) = 20 位

主存地址 = lb($B \times M$) = lb($2^5 \times 2^{27}$) = 32 位

PMC-Sierra RM7000A 64 位 MIPS RISC 处理器。RM7000 采用的高速缓存结构与 Intel 和 PowerPC 不同。在该高速缓存结构中,采用了 3 个分离的高速缓存。

1. 一级指令高速缓存:16 KB、4 路组相联高速缓存,每个块的大小是 32 B(8 条指令);
2. 一级数据高速缓存:16 KB、4 路组相联高速缓存,每个块的大小是 32 B(8 个字);
3. 二级高速缓存:256 KB、4 路组相联,可用于指令和数据。

除这 3 个片上高速缓存之外,RM7000 还提供了一个专用的三级高速缓存接口,支持的三级高速缓存大小是 512 KB,2 MB 和 8 MB,只有在二级高速缓存缺失之后才能访问三级高速缓存。

一级高速缓存需要一个周期时间访问,这些一级高速缓存均有 64 位的读数据通路和 128 位的写数据通路,两者可同时进行访问,提供超过 4 GB/s 的总带宽。二级高速缓存有 64 位的数据通路,只能在一级高速缓存缺失之后才能被访问,缺失损失为 3 个周期。由于 RM7000 非同寻常的高速缓存结构,其采用了两种高速缓存访问模式,如下所示。

非阻塞高速缓存。在这种模式下,高速缓存在发生缺失时并不会停顿,相反处理器在一级高速缓存之外继续操作,直至下列事件之一发生:

1. 两次高速缓存缺失未处理,且在指令总线上出现了第三条 load/store 指令。
2. 后继指令需要从产生高速缓存缺失的任一指令中请求数据。

即使在高速缓存发生缺失后,仍然允许对高速缓存继续进行操作,故这种非阻塞高速缓存可提高整体性能。

高速缓存锁定。在这种模式中,关键代码或数据段被锁定在一级和二级高速缓存中,锁定的块可以在写命中时更新,但在发生高速缓存缺失时不能选作替换对象。RM7000 允许其 3 种高速缓存分别独立进行锁定,但是,每种高速缓存 4 个高速缓存组中只有两个能够被锁定。具体来讲,RM7000 允许最大 128 KB 的数据或代码能被锁定在二级高速缓存中,最多 8 KB 的代码能够被锁定在指令高速缓存中,最多 8 KB 的数据能够被锁定在数据高速缓存中。

6.3　小结

本章讨论了存储器层次中第一级的设计和分析，即高速缓存，并讨论了局部性问题以及其对平均访问时间的影响。对三种高速缓存映射技术，即直接映射全相联映射和组相联映射进行了分析，并比较了它们的性能。同时，还介绍了三种替换技术：随机、FIFO 和 LRU 替换，并分析了这三种技术对高速缓存命中率的影响。介绍和分析了高速缓存写策略。最后介绍了三个实际例子的高速缓存结构和特征：奔腾 IV 处理器、PowerPC 处理器和 PMC-Sierra RM7000 处理器。第 7 章将讨论与主存内部和外部设计相关的话题，同时还会讨论虚拟存储器。

习题

1. 假设所给定的主存访问时间为 100 ns，高速缓存访问时间为 20 ns，则为使平均访问时间少于 60 ns，存储器交叉因数应为多少？合成系统的平均访问时间是多少？
2. 若一个系统有三级存储，高速缓存、半导体主存和磁盘二级存储，若它们的存储器访问时间分别为 20 ns，100 ns 和 1 ms，高速缓存命中率是 90%，主存命中率是 95%，则系统的平均访问时间为多少？
3. 一个计算机系统主存大小为 16 MB，32 位字，高速缓存为 8 KB，假设计算机采用字节可寻址机制，确定下列方式的地址中每个字段的位数：
   (a) 直接映射，块大小为 1 个字；
   (b) 直接映射，块大小为 8 个字；
   (c) 相联映射，块大小为 8 个字；
   (d) 组相联映射，每组 4 块，块大小为 1 个字。
4. 考虑下面关于数组 A 的程序段执行情况。

```
For i: = 0 to 7 do
SUM: = 0
    For j: = 0 to 7 do
        SUM: = SUM + A(i,j)
    End for
AVE(i): = SUM/8
End for
```

   设主存被分为 8 个交叉存储块，每个高速缓存块由 8 个元素组成，同时设高速缓存访问时间为 10 ns，主存访问时间是高速缓存访问时间的 10 倍。计算数组 A 中元素的平均访问时间。
5. 考虑在数组 A 上的程序段执行情况。4×10 的两维数组 A 以列主序列存储在主存中，设在高速缓存中有 8 个块，每个块正好一个字，替换策略采用 LRU。针对 3 种不同的高速缓存结构，即直接映射、全相联和组相联映射，针对下标 j 和 k 的不同值，说明高速缓存块内容的变化记录，针对获得的结果加以评论。

```
SUM: = 0
For j: = 0 to 9 do
    SUM: = SUM + A(0,j)
End for
AVE: = SUM/10
For k: = 0 to 9 do
    A(0,k): = A(0, k)/AVE
End for
```

6. 考虑 4×8 的两维数组,设数组的每个元素占一个字,数组元素以行为主方式存放在主存的 1000 单元至 1031 单元。高速缓存由 8 个块组成,每个块由 4 个字组成。设替换策略采用 LRU 方式。若采用上述 3 种映射技术以下面的序列请求数组元素,考察高速缓存中的变化情况。

$a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}, a_{0,4,}, a_{0,5}, a_{0,6}, a_{0,7}$

$a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}, a_{1,5}, a_{1,6}, a_{1,7}$

$a_{2,0}, a_{2,1}, a_{2,2}, a_{2,3}, a_{2,4}, a_{2,5}, a_{2,6}, a_{2,7}$

$a_{3,0}, a_{3,1}, a_{3,2}, a_{3,3}, a_{3,4}, a_{3,5}, a_{3,6}, a_{3,7}$

说明给定请求发生前后高速缓存状态的变化、替换次数以及高速缓存利用率的估计值。

7. 设计算机系统主存包括 1 M 的 16 位字,高速缓存大小为 4 K 字,组织方式为块组相联方式,每个组有 4 个块,每个块为 64 字,高速缓存初始状态为空。设 CPU 需要从单元 0,1,2,…,4351(按顺序)取 4352 个块,并重复取操作 9 次。若高速缓存比主存快 10 倍,估算由于采用高速缓存性能提高的比例。设定若从主存中取出一个块,同时高速缓存相应组已填满,新块将替换该组中最近最少使用的块。再考虑下列情况,若采用最近最多使用替换技术,即若高速缓存已满,新块将替换高速缓存中最近使用最多的块。注:该例引自参考文献 #3。

参考文献和推荐读物

S.D. Burd, *Systems Architecture*, 3rd ed., Thomson Learning Ltd, Boston, USA, 2001.

H. Cragon, *Memory Systems and Pipelined Processors*, Jones and Bartlett: Sudbury, MA, 1996.

V. C. Hamacher, Z.G. Vranesic, and S.G. Zaky, *Computer Organization*, 5th ed., McGraw-Hill, New York, 2002.

J. L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 1996.

V. P. Heuring, and H.F. Jordan, *Computer Systems Design and Architecture*, Addison-Wesley, NJ, USA, 1997.

D. A. Patterson, and J.L. Hennessy, *Computer Organization & Design: The Hardware/ Software Interface*, Morgan Kaufmann, San Mateo, CA, 1994.

H. S. Stone, *High-Performance Computer Architecture*, Addison-Wesley, Amsterdam, Netherlands, 1987.

M. Wilkes, Slave memories and dynamic storage allocation, IEEE Trans. Electron. Comput., EC-14(2), 270 – 271, (1965).

B. Wilkinson, *Computer Architecture: Design and Performance*, Prentice-Hall, Hertfordshire, UK, 1996.

参考网站

http://www.sysopt.com

http://www.intel.com

http://www.AcerHardware.com

http://www.pmc-sierra.com/products/details/rm7000a

http://physinfo.ulb.ac.be/divers_html/PowerPC_Programming_Info/into_to_ppc/pc2_hardware.html

第7章　存储系统设计 III

在第6章中我们介绍了存储器层次结构的概念。我们还介绍了存储器层次结构访问局部性的特征及其对平均访问时间的影响。然后，我们又讨论了关于存储器层次结构中的第一级，即高速缓存(建议读者认真复习第6章之后，再开始本章的学习)的不同问题。在本章，我们将继续讨论存储器层次结构中其他级的情况。具体来讲，首先讨论有关(主)存储器部件的设计和分析问题，然后讨论有关虚拟存储器的设计，本章最后讨论不同只读存储器(ROM)的实现。

7.1　主存

正如名字所示，主存提供了计算机的主要存储设施。图7.1表示主存和CPU之间的一个典型接口，采用两个CPU寄存器作为CPU对主存的接口，它们分别是存储地址寄存器(MAR，memory address register)和存储数据寄存器(MDR，memory data register)。MDR用于保存将被存储到主存单元或从主存单元中取回的数据，而主存单元的地址则保存在MAR中。

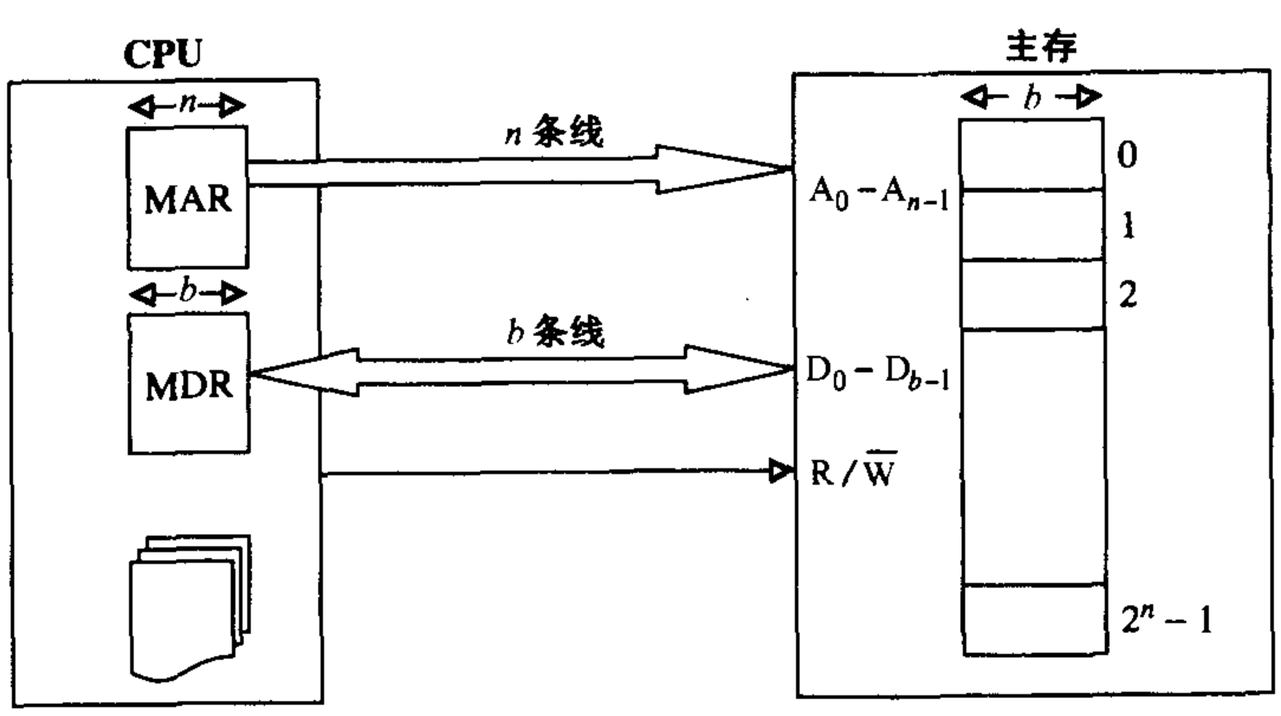


图7.1　CPU与主存的典型接口

典型的主存内部结构可以视为由多行多列的基本单元(cell)所构成，每个单元能够存储一位的信息。如图7.2所示是一个存储器芯片的概念性内部结构，在该图中，一个给定行上的单元构成了一个给定字的位，地址线 A_{n-1} $A_{n-2}\cdots\cdots A_1$ A_0 作为地址译码器的输入，从而能够产生字选择线 $W_{2^n-1}\cdots W_1 W_0$。同一行上的所有存储单元共用一条字选择线，在任一给定时刻，地址译码器只会激活一条字选择线，同时使其他选择线无效。一条字选择线使同一行上的所有单元能够被读或写。数据(位)线用于输入或输出单元里的内容，每个存储单元与两条数据线相连，一条给定数据线为某一指定列中的所有单元共有。

在静态CMOS技术中，每个主存单元由6个晶体管组成，如图7.3所示。这6个晶体管静态CMOS存储单元构成了两个背靠背的反相器。需要注意的是，存储单元可处于两种稳定状态中的一种。例如，在图7.3中，$A=1$，则晶体管 N_2 被打开，而点 $B=0$，它反过来又使得晶体管 P_1 打开，从而使得点 $A=1$，这即是单元的一种稳定状态，称为状态1。类似地，若 $A=0$，则 $B=1$，

这表示了单元的另一种稳定状态，称为状态 0。晶体管 N_3 和 N_4 用于连接单元至两条数据(位)线上。正常情况下(若字选择没有被激活)，这两个晶体管不接通，因此可以保护单元免受数据线上的信号影响。当字选择线被激活后，这两个晶体管被接通，在这两个晶体管被接通后，单元状态的变化就取决于所采取的存储操作，如下所示。

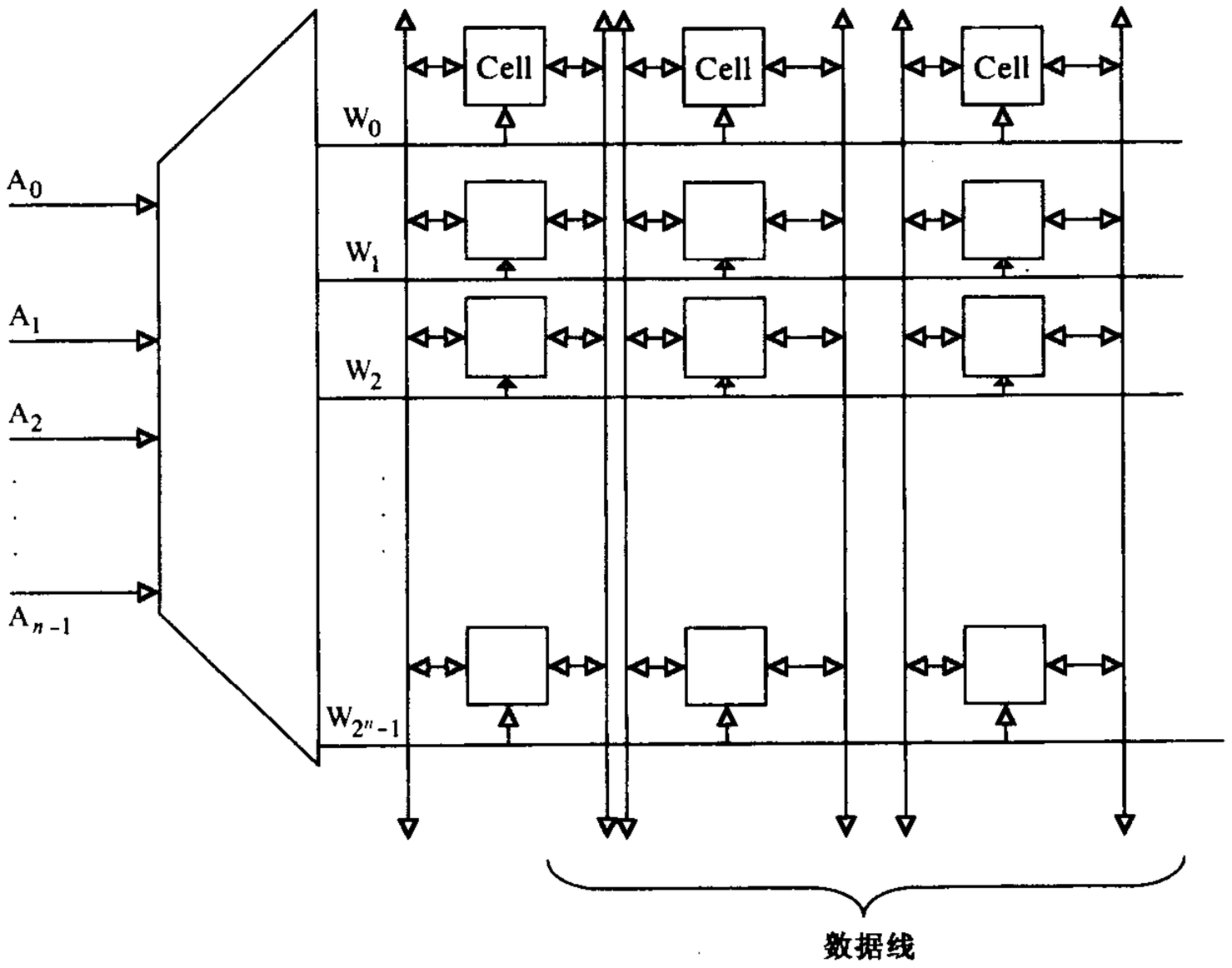


图 7.2　存储器芯片概念性内部结构图

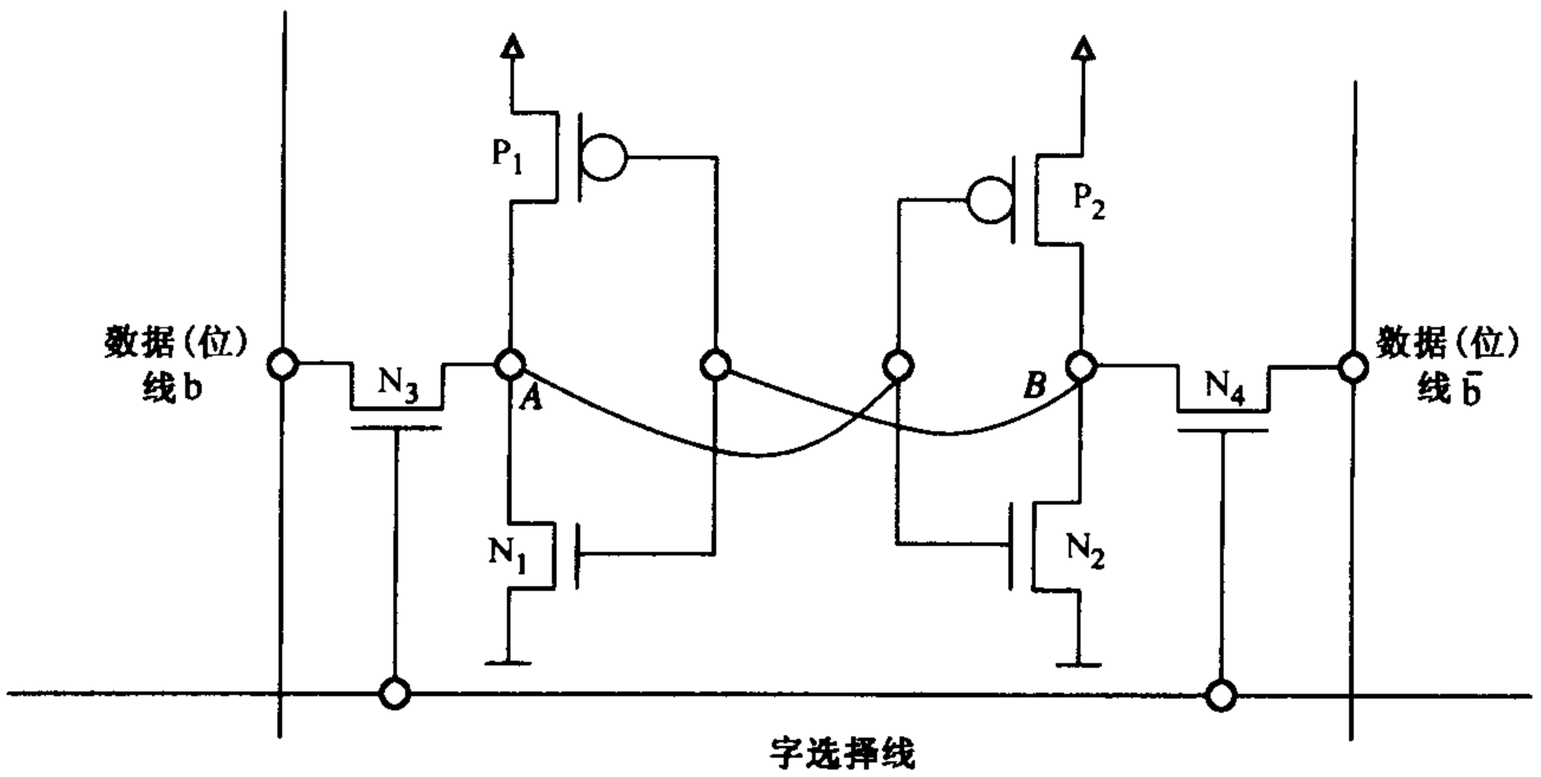


图 7.3　静态 CMOS 存储单元

读操作。

1. 线 b 和 $\bar{b}$ 预充电至高电位；
2. 字选择线被激活，接通晶体管 N_3 和 N_4；
3. 根据存储单元中已存储的内部值，点 A(B)将会使得线 b($\bar{b}$)放电。

写操作

1. 位线预充电，使得 $b(\bar{b}) = 1(0)$；
2. 字选择线激活，接通晶体管 N_3 和 N_4；
3. 预充电为 0 的位线，将迫使点 $A(B)$由 1 变为 0。

存储阵列内部结构需要满足一个重要的存储器设计因素，即存储器芯片的高效利用。例如，一个 1 K×4 的存储芯片，采用如图 7.2 所示的结构，存储阵列组织为 1 K 行的单元，每行由 4 个单元组成。芯片将需要有 10 个地址引脚、4 个数据引脚。但是，这并不一定是芯片面积最好的利用方式。存储单元阵列的另一种可能组织是，阵列结构为 64 行，每行 64 个单元。在这种情况下，需要 6 根地址线(形成行地址)以从 64 行中选取一行，另外 4 条地址线(列地址)用于从 64 位的一行中选取合适的 4 个位，其结构如图 7.4 所示。

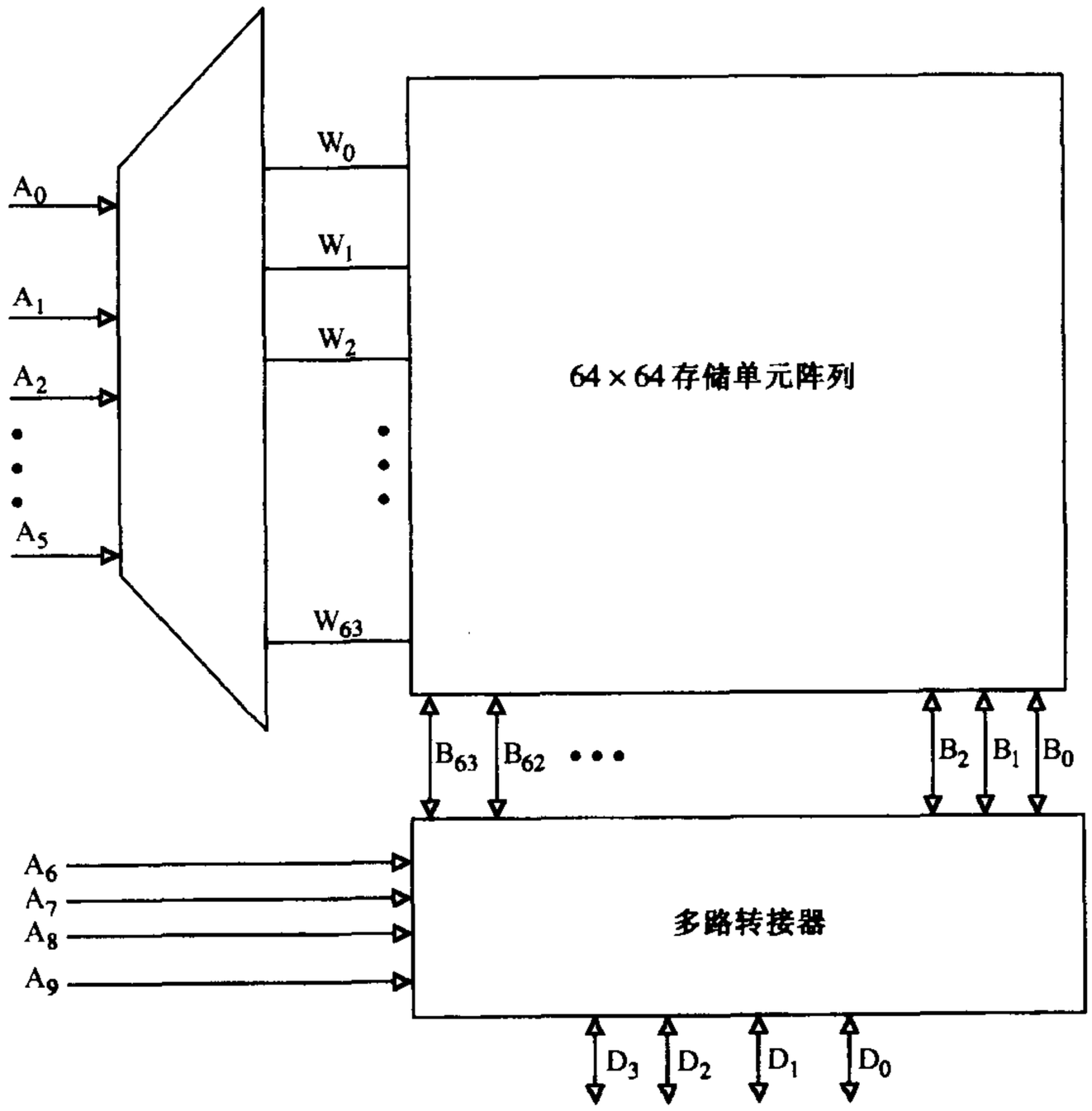


图 7.4 存储芯片的高效内部结构

与主存储子系统设计相关的另一个重要因素是在一个集成电路中所需的芯片引脚数。例如，一个存储子系统的容量为 4 Kb，同一存储容量的不同结构将会导致所需芯片引脚数量的不同，表 7.1 说明了这一现象。可以很清楚地看出，随着每个可寻址单元中位数的增加，集成电路所需的引脚数也在增加。

表 7.1 采用不同结构对引脚数的影响

组织	所需地址线数	所需数据线数
4 K×1	12	1
1 K×4	10	4
512×8	9	8
256×16	8	16

设计主存储器子系统的另一个相关因素是所需存储器芯片的总数。可用的单个存储器芯片容量是设计存储子系统的一个限制因素。例如,采用 1 Mb 的芯片设计一个 4 MB 的主存储子系统,需要的芯片数为 32 个,需要注意的是 4 MB 系统所需的地址线数为 22,数据线数为 8。如图 7.5 所示为一个框图,说明了所设计的存储子系统以及构造这样的子系统所需的构件块。

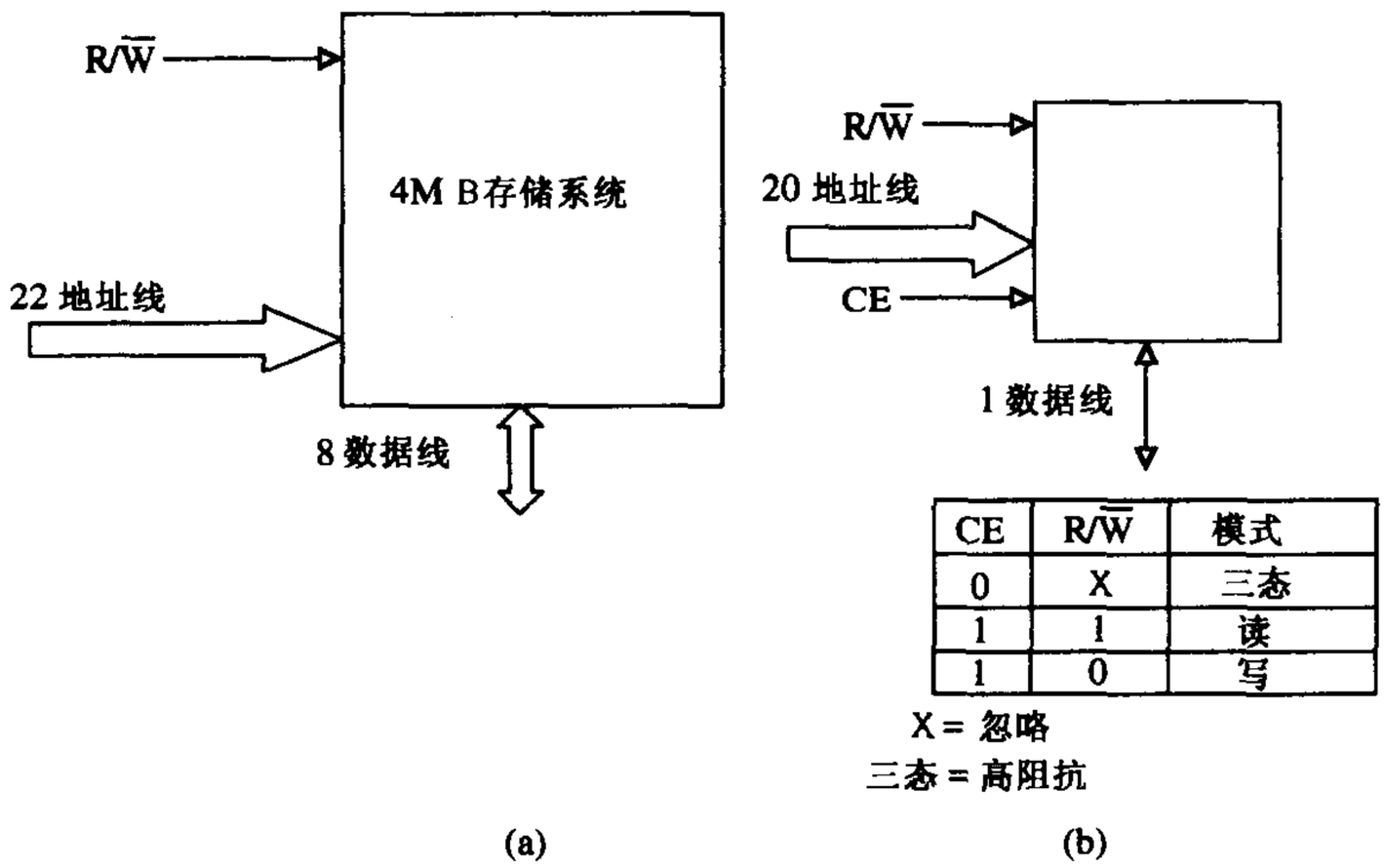


图 7.5　一个存储系统的框图及其构件块
(a)预期的存储系统;(b)基本存储器的构造块

存储子系统可以排列成 4 行,每行有 8 个芯片,这种排列的框图如图 7.6 所示。在该图中,最低有效的 20 根地址线 $A_{19} \sim A_0$ 用于构件块,即 1 Mb 的单位芯片的寻址,高位的 2 根地址线 $A_{21} \sim A_{20}$ 用做一个 2－4 译码器的输入,以产生 4 位使能线,每根线均连到一行 8 个芯片的 CE 线上。

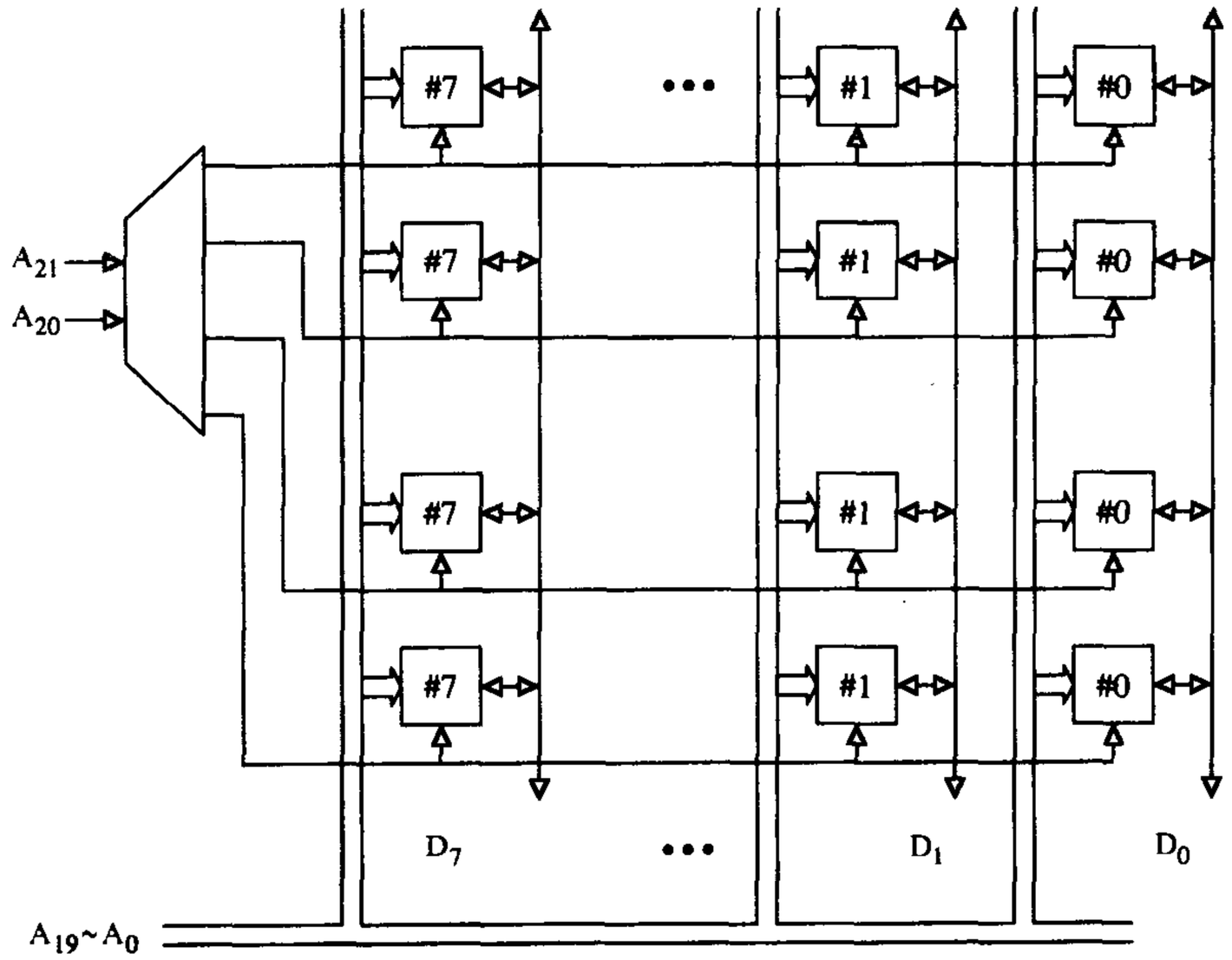


图 7.6　采用 1M 1 位存储芯片构成 4M 8 位存储器的结构

在针对主存储器系统的上述讨论中假定采用了一个 6 晶体管的静态随机单元,但也可以采用 1 晶体管的动态单元。动态存储器取决于所保存的逻辑值,它采用一个电容和一个作为开关的晶体管。采用动态存储器可以节省芯片面积,但是,由于已存储值的衰减性(电容上存储电能的泄露),动态存储器需要周期地进行刷新,以恢复所存储的逻辑值。如图 7.7 所示为动态存储器阵列结构,图 7.7 中读/写电路起到感应位线值、放大该值并刷新保存在电容里值的作用。

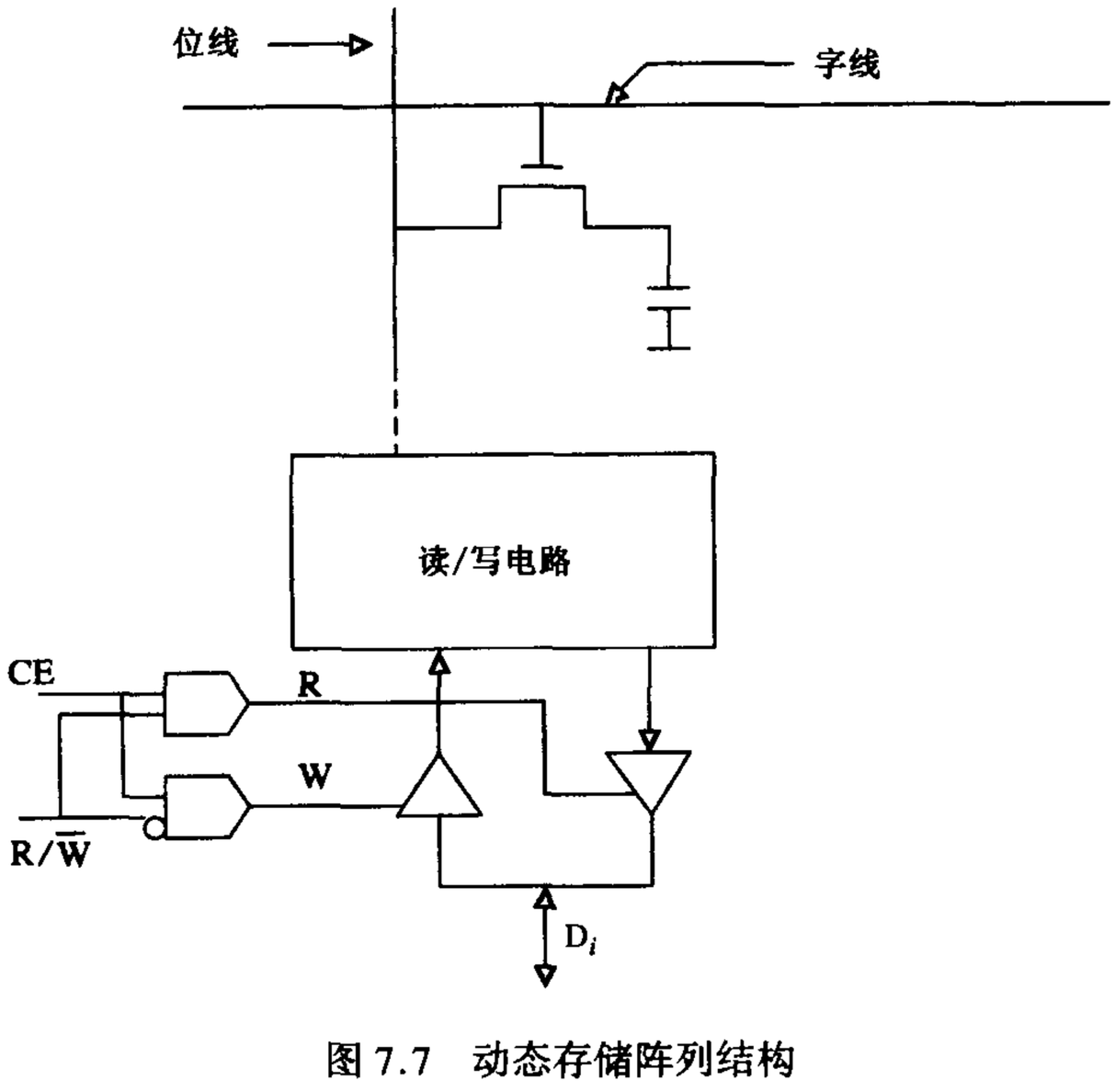


图 7.7 动态存储阵列结构

为了执行读操作,位线预充电至高位(与静态存储器相同),并且激活字线,这将使得保存在电容里的值出现在位线上,即出现在数据线 D_i 上。可以看出,读操作带有破坏性,即电容被充到位线电位。因此,每个读操作之后要进行一个相同值的写操作。

为了执行写操作,要将待写入的值加载到位线上,同时激活写线。若待写入的值是 1,则电容进行充电,若值是 0,则电容进行放电。表 7.2 是对控制电路操作的小结。

表 7.2 控制电路的操作

CE	R/W	操作
0	x	无
1	1	读
1	0	写

x = 忽略

如前所述,存储器子系统合适的内部组织有利于节省所需的 IC 引脚,这是 IC 设计的一个重要考虑因素。为了降低一个给定动态存储子系统所需的引脚数,通常的做法(与静态存储器的情况相同)是将地址线分为行地址线和列地址线。另外,行地址线和列地址线可以在相同的引脚上进行传输,即先传输一种地址线,然后再传输另一种地址线,这种模式称为时间复用传输(time-multiplexing)方式,这种方式可以将地址引脚数减少一半。由于采用地址线的时间复用方式,因此有必要引入额外的两条控制线,即行地址选通($\overline{RAS}$)和列地址选通($\overline{CAS}$)。这两根控制

线分别用于向存储芯片指明行地址线有效时间以及列地址线有效时间。例如,设计一个 1 M × 1 的动态存储器子系统,如图 7.8 所示是其一种可能内部结构,它是一个存储单元阵列,阵列组织方式为 1024 × 1024。

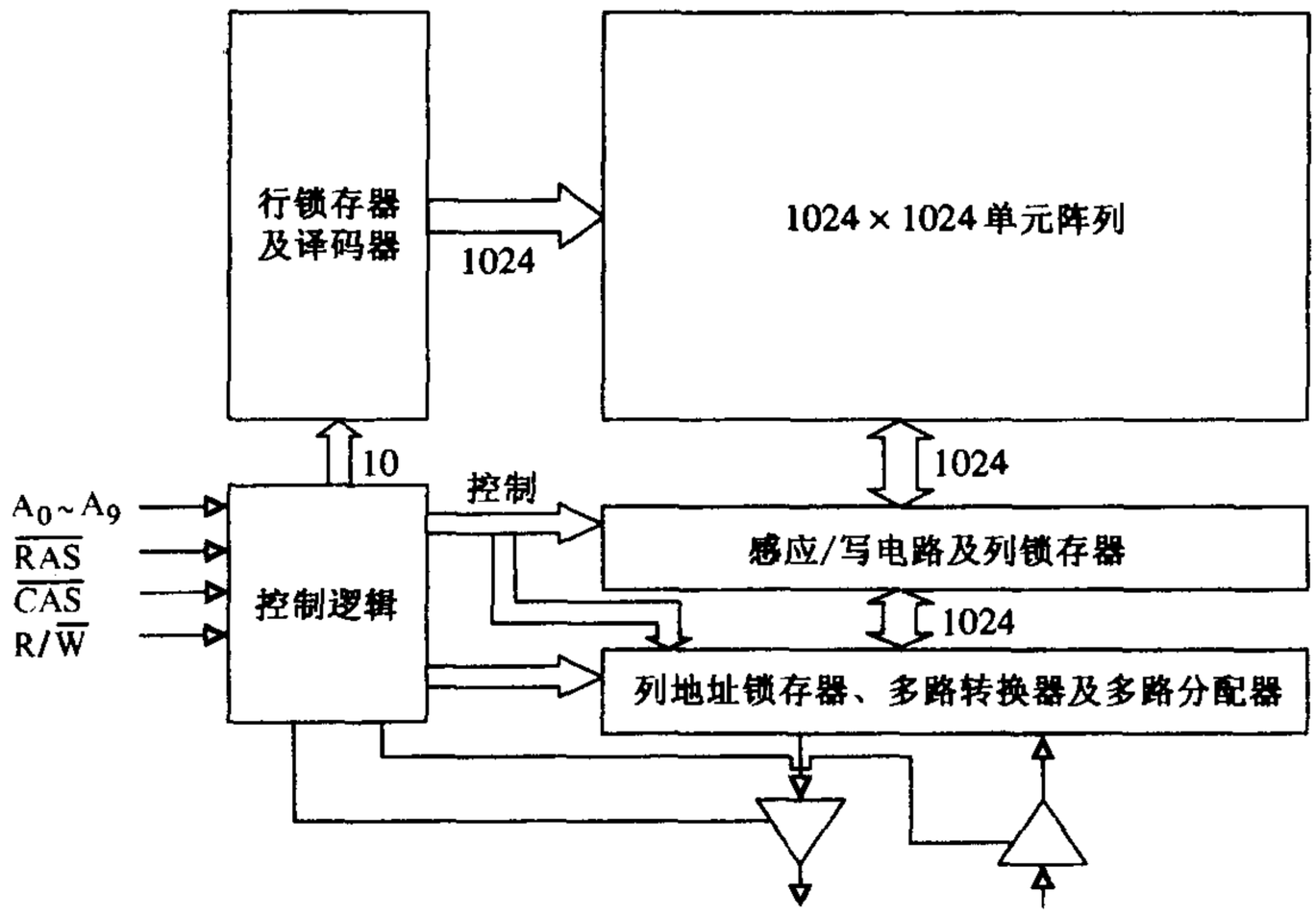


图 7.8　一个存储阵列结构

需要注意的是,这里只用了 10 根地址线,它们为 10 根行地址线和 10 根列地址线所复用。行锁存器和列锁存器用于保存一个存储周期的行地址和列地址,在这种情况下,一个存储访问由一个$\overline{RAS}$和行地址,以及其后的$\overline{CAS}$和列地址所组成。

7.2　虚拟存储器

虚拟存储器的概念在原则上类似于 6.2 节中的高速缓存概念。虚拟存储系统意在利用硬盘(低速)优化主存(高速)的使用。实际上,虚拟存储器是一种利用二级存储扩展存储器可用空间至超出其有限物理容量的技术,通常可用的物理存储空间不足以驻留一个给定活动程序的所有部分,程序中的活动部分被调入到主存,而非活动部分将保存在磁盘上。若程序段含有处理器正请求的、但不在主存中的字,则该段将会从硬盘中调入到主存中。应用在虚拟存储器设计中的基本原则与那些应用在高速缓存中的原则一样,最为相关的原则是将活动部分保存在高速主存中,而将非活动部分移入硬盘中。

在硬盘和主存之间移动数据的形式是页,页是存储字的集合,当处理器请求访问其中的字时,页能够从硬盘移到主存。现代计算机中,页的典型大小是 2 KB 至 16 KB。当处理器请求字的所在页不在主存中时,将发生页缺失,此时必须将该页从硬盘调入。在主存和硬盘之间移动程序或数据页,对应用程序员来讲是完全透明的。数据和程序的这种移动由操作系统负责。

值得提及的是,虽然高速缓存和虚拟存储器基于类似的原则,但二者之间存在一个显著的差别,即高速缓存缺失引起的时间开销是高速缓存命中时的 5 至 10 倍,而页缺失所需的开销则是页命中时的 1000 倍。因此,不可能在页缺失时让处理器等待将页传输到主存中,这是因为在页传输过程中,现代的处理器可以执行几千条指令。

处理器发出访问一个指定字的地址并不对应于一个物理存储空间,因此,这些地址称之为

虚拟(逻辑)地址。存储器管理部件(MMU)负责将虚拟地址转换为它们相应的物理地址。地址转换技术有三种,它们是直接映射、相联映射和组相联映射。在这些技术中,主存单元和相应虚拟页的信息保存在一个表中,称为页表,页表保存在主存中。保存在页表中的其他信息有一位用于指示页的有效性、一位用于指示页的修改,还有一位用于指示页的访问权限。若页已装入主存中,则就设置其对应的有效位(valid bit)。在计算机启动后,复位所有页的有效位。保存在页表中的另一控制位是脏位(dirty bit),若驻留在主存中的页被改动,则设置该位。若驻留在主存中的给定页没有修改过,则将其脏位重置。这有助于确定是否将一个页中的内容写回硬盘(在替换时),或只是用其他的页来覆盖该页的内容。下面关于地址转换技术的讨论中将用到这些不同类型的控制位。

7.2.1 直接映射

图 7.9 说明了直接映射技术中的虚拟地址转换过程。在这种情况下,处理器发出的虚拟地址分为两个字段:虚拟页码字段和偏移字段。若虚拟页字段的位长为 N,则页表项的总数为 2^N。

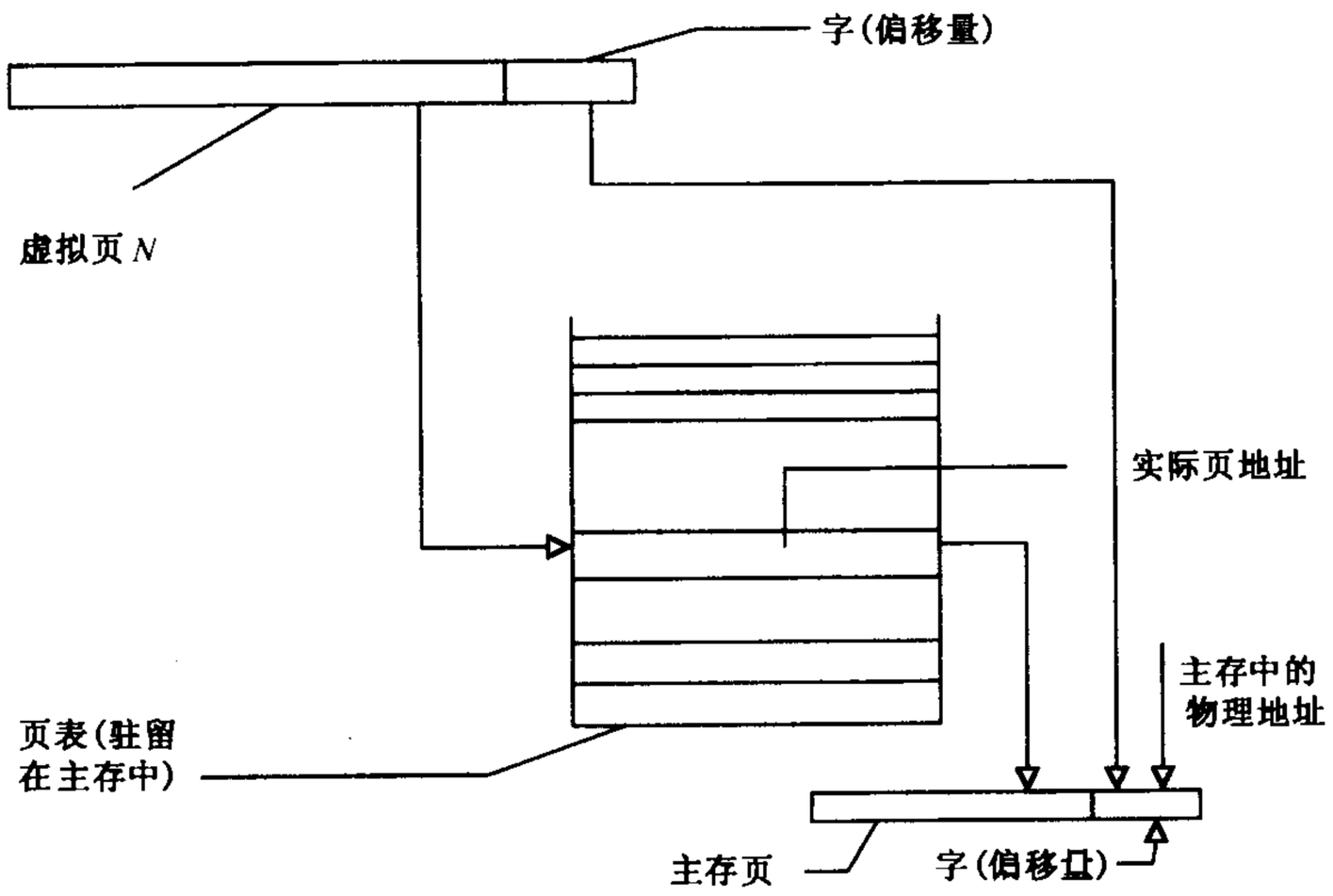


图 7.9 直接映射的虚拟地址转换

虚拟页码字段直接用来对页表的项进行编址,若相应的页是有效的(由有效位指示),则指定页表项中的内容即对应于物理页的地址。物理页地址与偏移量字段一起可以形成处理器请求字的物理地址。而若页表中的指定项不包含有效的物理页码,则表示一个页缺失。在这种情况下,MMU 将会从硬盘中把该页调至主存,同时标明页的有效性,此后的转换过程与前面讲述的过程相同。

直接映射技术的主要优点在于它的简单性,即直接对页表项进行编址。其主要不足则是需要巨大的页表。为了克服对大页表的需要,可以使用下面将要介绍的相联映射技术。

7.2.2 相联映射

如图 7.10 所示为相联映射技术的地址转换技术,该技术与直接映射类似,即处理器发出的虚拟地址被分为两个字段:虚拟页码和偏移量字段。但是,相联映射页表远比直接映射页表短,页表中的每个项可以分为两个部分:虚拟页码和物理页码。匹配操作是在地址的虚拟页码字段

和保存在页表中的虚拟页码之间进行搜索。若找到一个匹配，则取出保存在页表中的相应物理页码，与偏移量一起产生处理器请求字的物理地址。若没有找到匹配，则表示存在一个页缺失。在这种情况下，MMU 将从硬盘中把相应的页调入主存，并标明该页的有效性，此后的转换过程与前面讲述的过程相同。

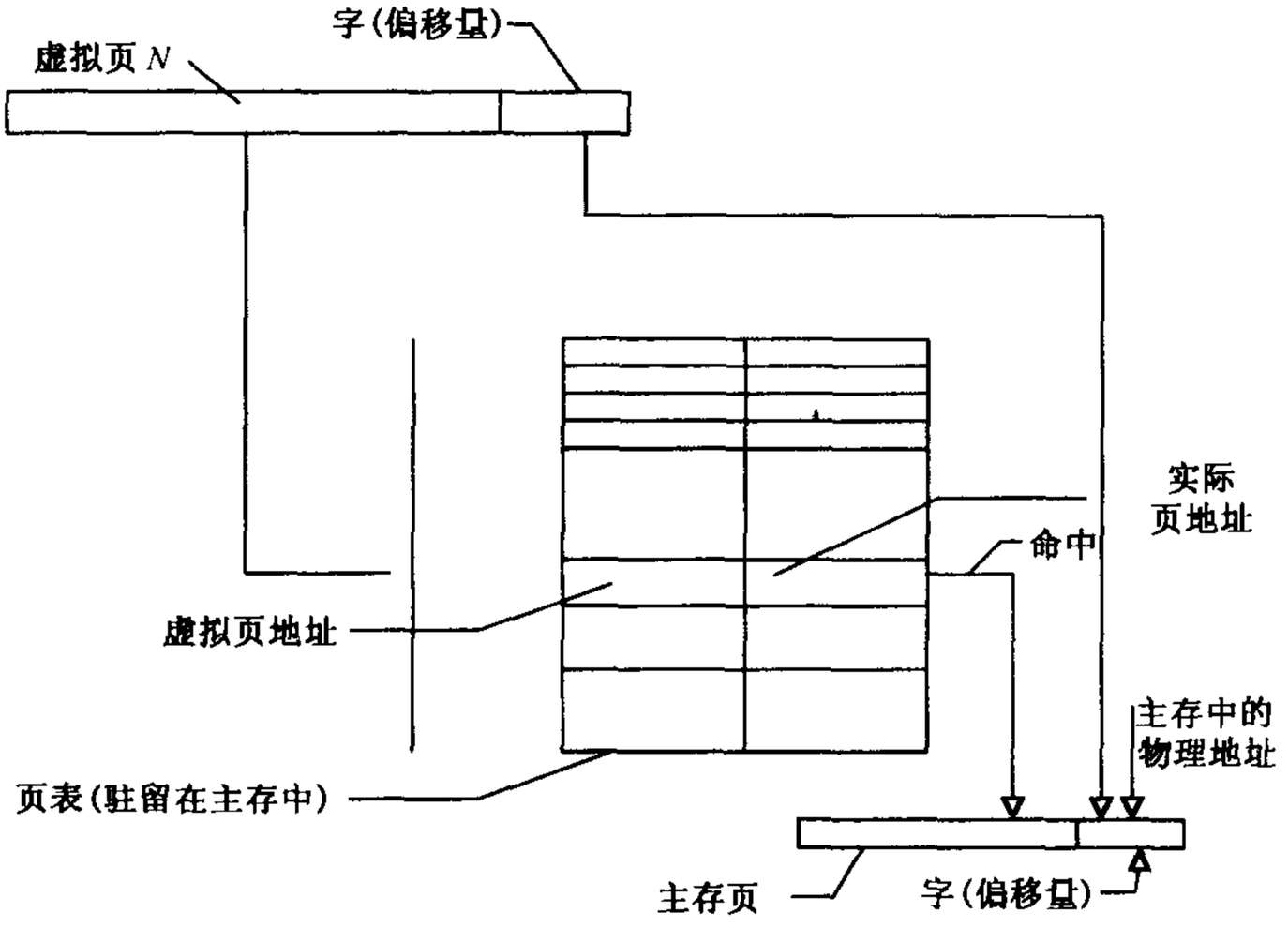


图 7.10　相联映射的地址转换

相联映射技术的主要优点是转换过程所需的页表较短(相对于直接映射技术)，其主要不足是需要将虚拟页码字段与保存在页表中的所有虚拟页码进行匹配，尽管这种搜索是全相联的操作，但需要额外的硬件开销。

相联映射的复杂性和直接映射简单性之间的可能折中是组相联技术，这种混合技术将在下面进行讨论。

7.2.3　组相联映射

图 7.11 说明了组相联映射中的地址转换。其中，处理器发出的虚拟地址被分为三个字段：标识字段、下标和偏移量。组相联映射中的页表被分为多个组，每个组由一定数量的项组成。页表中的每项由一个标志和相应的物理页地址组成。类似于直接映射，用索引直接确定将要执行搜索的组。若索引字段的位数为 S，则页表中的组数为 2^S。在确定组之后(类似于相联映射)将标志字段与指定组的所有项进行匹配。若找到匹配，则相应的物理页地址被提取出来，与偏移量一起产生处理器请求字的物理地址。另一方面，若没有找到匹配，则是页缺失，在这种情况下，MMU 将从硬盘中将相应的页调至主存中，更新相应的组，标明页的有效性，此后的转换过程与前面讲述的过程相同。

组相联技术是直接映射在页表利用上的低效以及相联映射的过量硬件开销之间的一种折中，它同时吸收了两种技术的优点：直接映射的简单性和相联映射的高效性。

需要指出的是，在所有上述这些地址转换技术中，为了访问页表，均需要额外的主存访问。若将一小部分页表的副本保存在 MMU 中，可以节省这些额外的主存访问。这部分页表由最近访问过的那些页的页表项所组成，在这种情况下，在执行地址转换之前，要先行在这部分页表中

进行搜索以确定在虚拟地址字段中的虚拟页码是否可以找到匹配。这一小部分页表保存在MMU的地址转换旁路缓冲器(TLB)中,这将在下面进行说明。

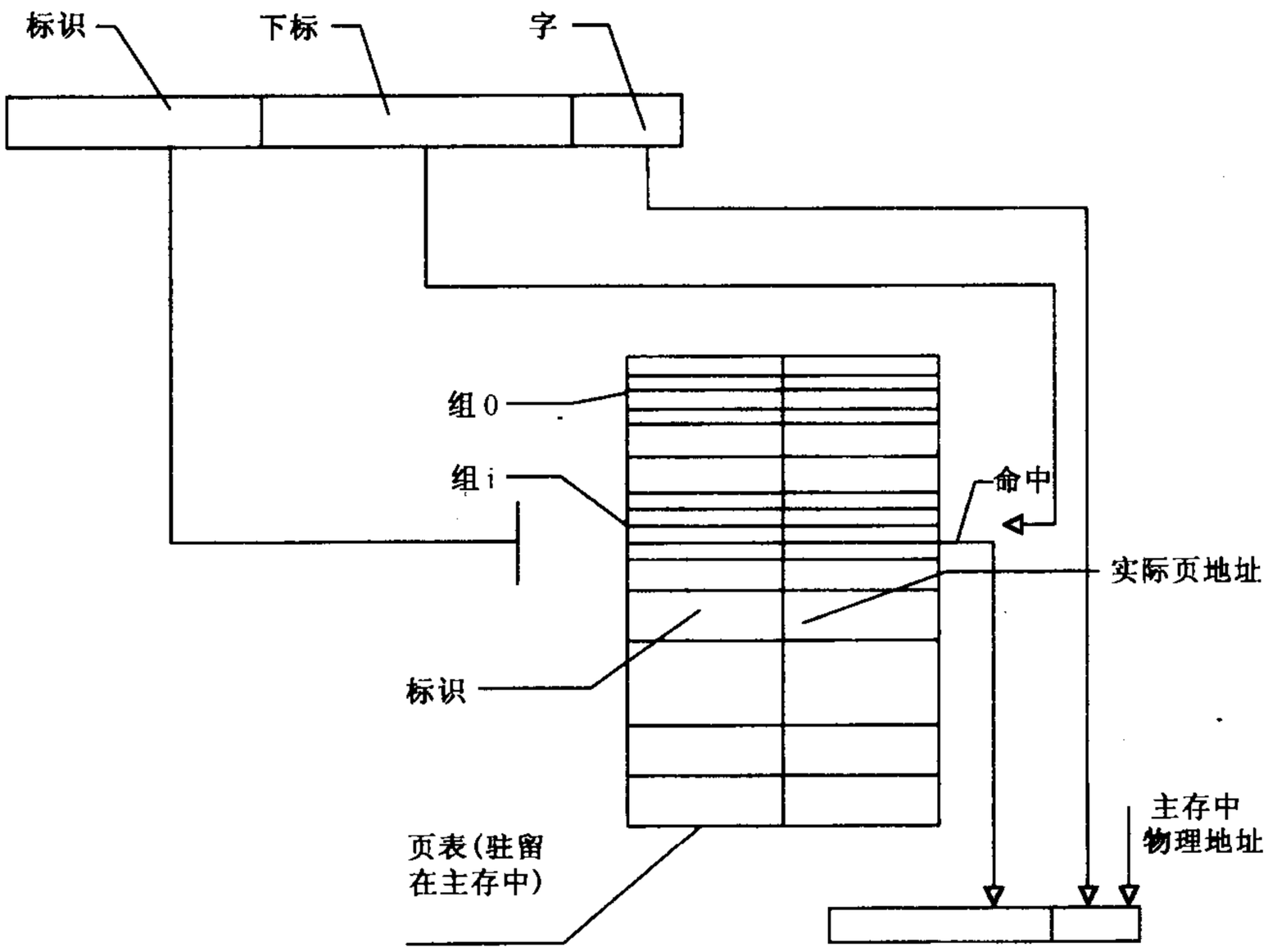


图 7.11 组相联映射的地址转换

7.2.4 地址转换旁路缓冲器

在现代计算机系统中,一小部分页表的副本被保存在处理器芯片中,这部分页表由最近访问页的页表项组成,这一小部分页表保存在地址转换旁路缓冲器(TLB)中。在对页表搜索之前首先对 TLB 进行搜索,因此,虚拟页字段首先与 TLB 中的项进行检查以期找到匹配。TLB 的命中可以产生处理器请求字的物理地址,因此可以节省访问页表所需的额外主存访问。需要注意的是,TLB 缺失并不等于页缺失。图 7.12 说明了在虚拟地址转换过程中 TLB 的用法,TLB 的典型大小范围是从 16 项至 64 项。对于这样小的 TLB 大小,命中率完全有可能在 90%以上。由于它的有限大小,因此在 TLB 中可进行相联搜索,从而降低了所需的搜索时间。

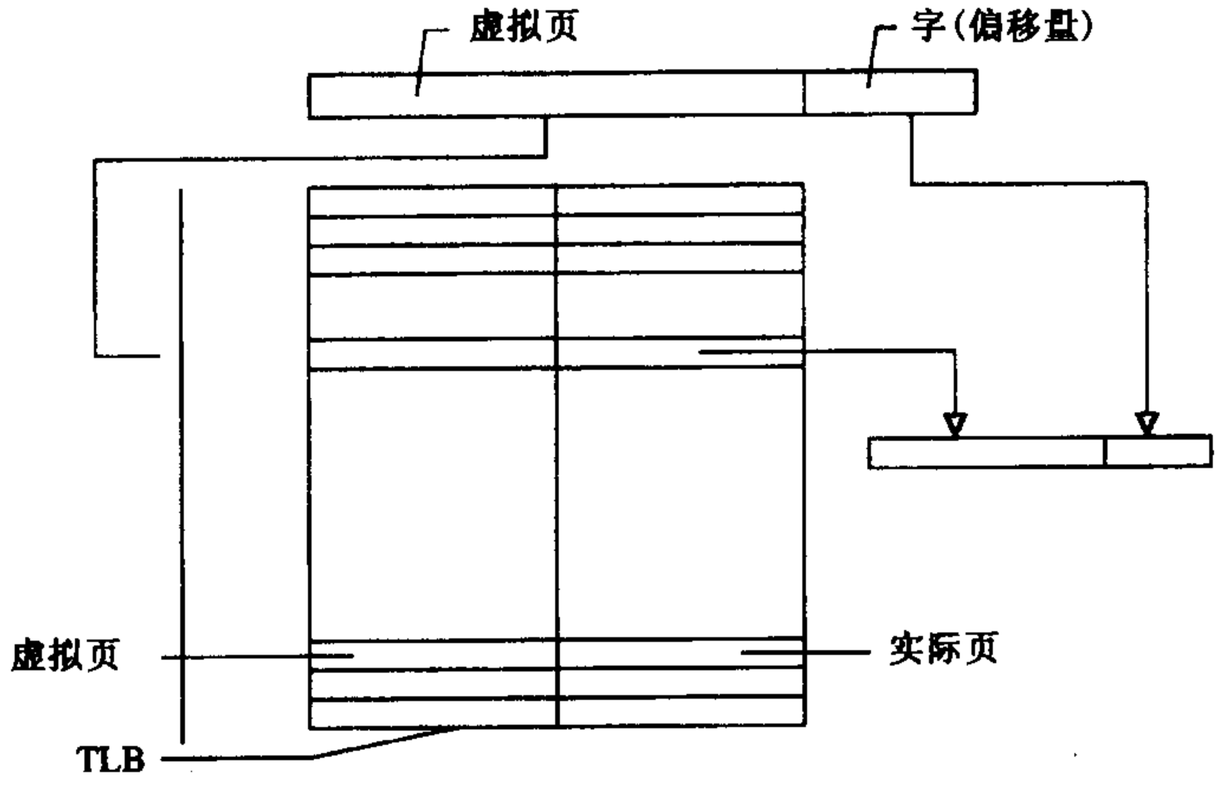


图 7.12 在虚拟地址转换中采用 TLB

为了说明采用 TLB 的有效性,可以考虑在虚拟存储系统中采用 TLB 时用到如下表所示参数时的情况。

TLB 中项的数目 = 16	TLB 相联搜索时间 = 10 ns
主存访问时间 = 50 ns	TLB 命中率 = 0.9

平均访问时间 $= 0.9(10 + 50) + 0.1(10 + 2 \times 50) = 0.9 \times 60 + 0.1 \times 110 = 65$ ns,与没有 TLB 时需要的 100 ns 访问时间形成了对比。需要注意,为了简单起见,这里忽略了上面说明中高速缓存的存在。

从上面的讨论中可以清楚地看到,请求的不在主存中的页(页缺失)越多,则需要从硬盘调入主存中的页也就越多,最终将会导致填满整个主存。将任何新的页从硬盘调入至已完全填满的主存中,将会引起这样一个问题:为了给调入的页提供空间,应去掉(替换) 哪个主存页? 替换算法(策略)将在下面进行说明。

值得注意的是,奔腾 IV 处理器有 36 根地址线,因此它的最大主存容量为 64 GB,根据 Intel 的性能规格,虚拟存储器为 64 TB(65 528 GB),这就使处理器的访问空间由 2^{36}B 增至 2^{46}B。与之相比,PowerPC 604 有两个 12 项、两路组相联地址的转换旁路缓冲器(TLB):一个用于指令,另一个用于数据,因此虚拟存储空间为 2^{52} = 4 PB (Peta-bytes)。

7.2.5　替换算法

虚拟存储器实现的基础是请求分页的概念,也就是依赖于操作系统,而不是程序员,根据运行进程的请求控制页面换入或换出主存。当一个进程需要非驻留页时,操作系统需要确定请求页替换哪一个驻留页,在虚拟存储中用来确定替换的技术称为替换算法(策略)。

已有一些可用的替换机制,这些机制的主要目标是确定近期预计无需引用的页为删去页。

随机替换。根据该替换策略,任选一页用于替换,这是最简单的替换机制,可以通过伪随机数生成器为所有页帧生成相应的随机数加以实现。在替换时,产生的随机数将确定需要被替换的页帧。由于其简单性,这项技术难以对主存进行高效利用,即命中率 h 低。随机替换在 RISC 处理器 Intel i860 系列中使用。

先进先出(FIFO)替换。根据该替换策略,主存中最早装入的页将被替换。页替换的基础是一个给定页驻留主存的时间,而与该页使用模式无关。该技术同样简单,若处理器引用页是按严格的顺序方式,该技术可以获得较好的性能,即较好的主存命中率。为了说明 FIFO 机制,请看下面的例子。

例 7.1　处理器引用页的序列如下:6,7,8,9,7,8,9,10,8,9,10。具体考虑以下两种情况:(a)主存中分配的页帧为 2;(b)分配的页帧为 3。图 7.13 显示了这两种情况下引用序列的轨迹,从该图中可以看到,当页帧数为 2 时,共有 11 次页缺失(黑体字显示);当页帧数增加至 3 时,页缺失降至 5。由于 5 页需要引用,因此这是最优的情况。当引用序列是按增加页码引用的严格次序时,FIFO 策略可以得到最优(最少)的页缺失。

最近最少使用(LRU)替换。根据该技术,页替换是根据主存中给定页的使用模式,而与该页在主存中的时间无关。最久没有被引用的主存页被替换,LRU 技术符合大多数程序的特征,因此在主存命中率上预计有可能获得最好的性能,但相对其他技术而言也较为复杂。为了说明 LRU 机制的用法,请看下面的例子。

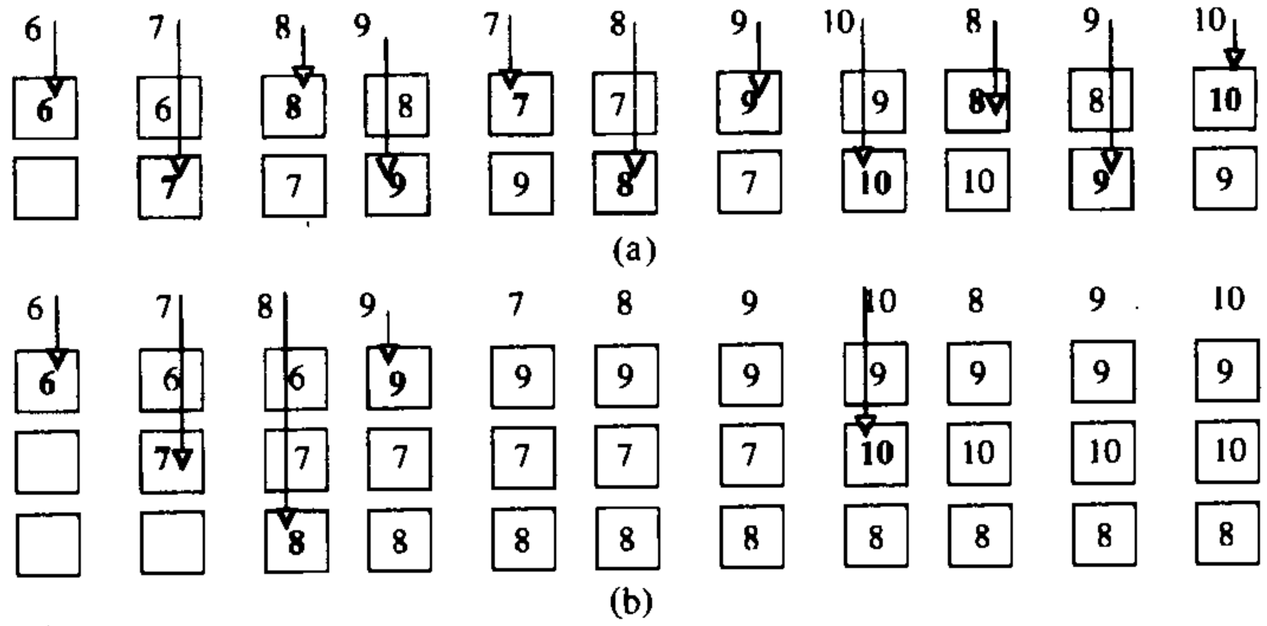


图 7.13 FIFO 替换技术

(a)页帧数为 2 时的 FIFO 替换(页缺失数为 11);(b)页帧数为 3 时的 FIFO 替换(页缺失次数为 5)

例 7.2 处理器请求页的引用序列如下:4,7,5,7,6,7,10,4,8,5,8,6,8,11,4,9,5,9,6,9,12,4,7,5,7。设在主存中分配的页帧数为 4,计算产生的页缺失数,主存内容轨迹如图 7.14 所示,页缺失数 = 17[①]。

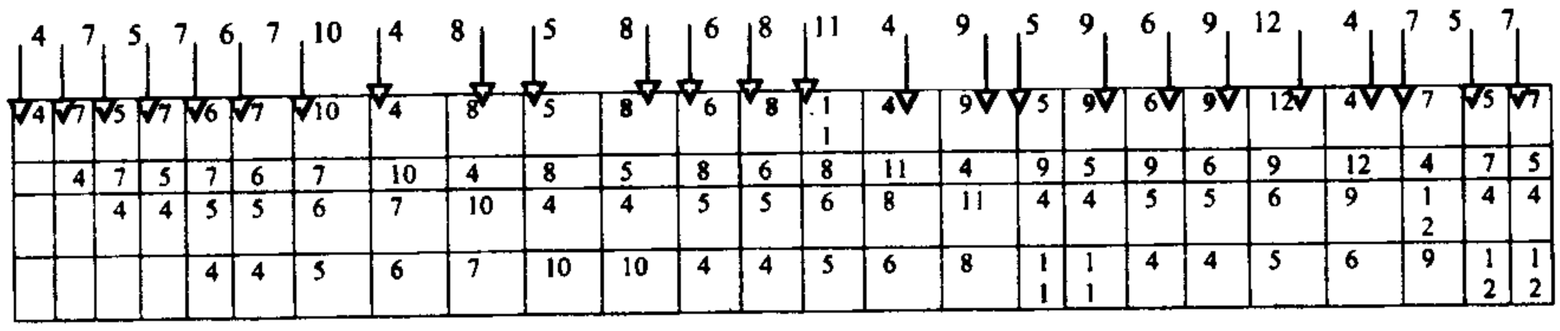


图 7.14 LRU 替换技术

LRU 有一个特殊的实现,称为基于堆栈的 LRU。在该实现中,最近被访问的页由最上面的页方框表示。与 FIFO 框图不同,该方框不表示某一特定的页帧,因此每次引用产生的页缺失是处于栈顶。值得注意的是,分配更多的页给程序,列中的页引用不会发生变化,只是页缺失的数目在变化,因此 n 个页帧的主存页集合是 $n+1$ 个页帧时页集合的子集。实际上,该框图可以看做是一个堆栈数据结构,堆栈的深度表示页帧数,若页不在堆栈中(即找到该页的深度大于页帧数),则产生一个页缺失。

例 7.3 考虑一个二维 8×8 的数组 A,数组按行主方式存储,有 3 个页帧,对于下面的数组初始化循环,计算将产生多少次页缺失。假设采用 LRU 替换算法,并且所有帧初始为空,页的大小为 16。

```
for I = 0 to 7 do
        for J = 0 to 7 do
          A[I, J] = 0;
        End for
End for
```

数组元素在二级存储上的排列如图 7.15 所示。在开始的 2 个外部循环执行中数组元素的请求序列如下所示:

I = 0

J = 0, 1, 2, 3, 4, 5, 6, 7

$a_{00}, a_{01}, a_{02}, a_{03}, a_{04}, a_{05}, a_{06}, a_{07}$ 页缺失数 = 1

① 原书有误,应为 18——译者注。

```
I = 0
    J = 0, 1, 2, 3, 4, 5, 6, 7
        a00, a01, a02, a03, a04, a05, a06, a07        页缺失数 = 1
```

根据上面的分析,可以清楚地看到,每个外部循环执行有一次缺失,因此总的缺失数为 8[①]。需要注意的是,若数组以列主方式存储,则每个外部循环将要产生 8 个页缺失,因此缺失总数将变为 64。

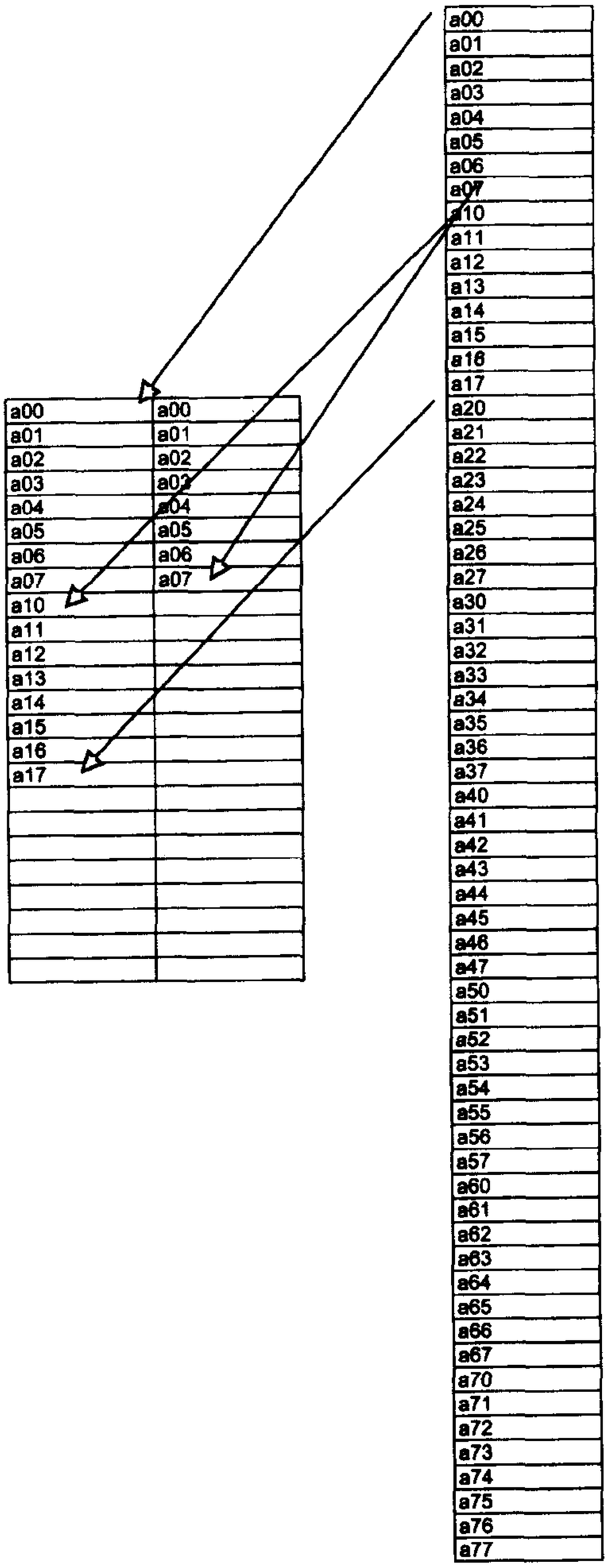


图 7.15　数组元素在辅存和主存中的排列

① 这里每个元素为 2 个字节——译者注。

时钟替换算法。这是一种改进的 FIFO 算法,既要考虑驻留主存页所花的时间(类似于 FIFO),同时又考虑页的使用模式(类似于 LRU),因此该技术有时称为先进不用者先出(FINUFO, First-In-Not-Used-First-Out)。为了保存时间和利用率的轨迹,该技术采用一个指针指向调入页的存放位置,以及一个已用位(used bit)标明指定页是否已使用过。下面的 3 步可以说明该项技术:

1. 若已用位 = 1,则复位该位,增量指针并重复;
2. 若已用位 = 0,则替换相应的页,增量指针;
3. 若页在初始装入后被引用,便置位已用位。

例 7.4 考虑下面的请求,如图 7.16 所示,其中,主存中有 3 个页帧,采用 FINUFO 技术:2,3,2,4,6,2,5,6,1,4,6,估计命中率。估计的命中率 = 4/11。

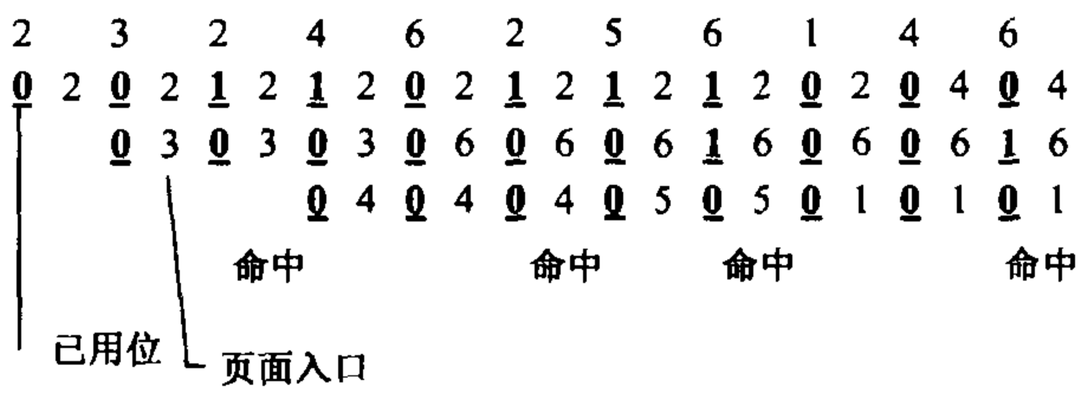


图 7.16 FINUFO 替换技术

7.2.6 带有高速缓存的虚拟存储系统

典型的计算机系统包含一个高速缓存、一个虚拟存储器和一个 TLB。当从处理器取得虚拟地址后,根据请求项在缓存、主存和二级存储系统中的可用性,会有一些不同的情形,图 7.17 所示为这些不同情形的通用流程图。

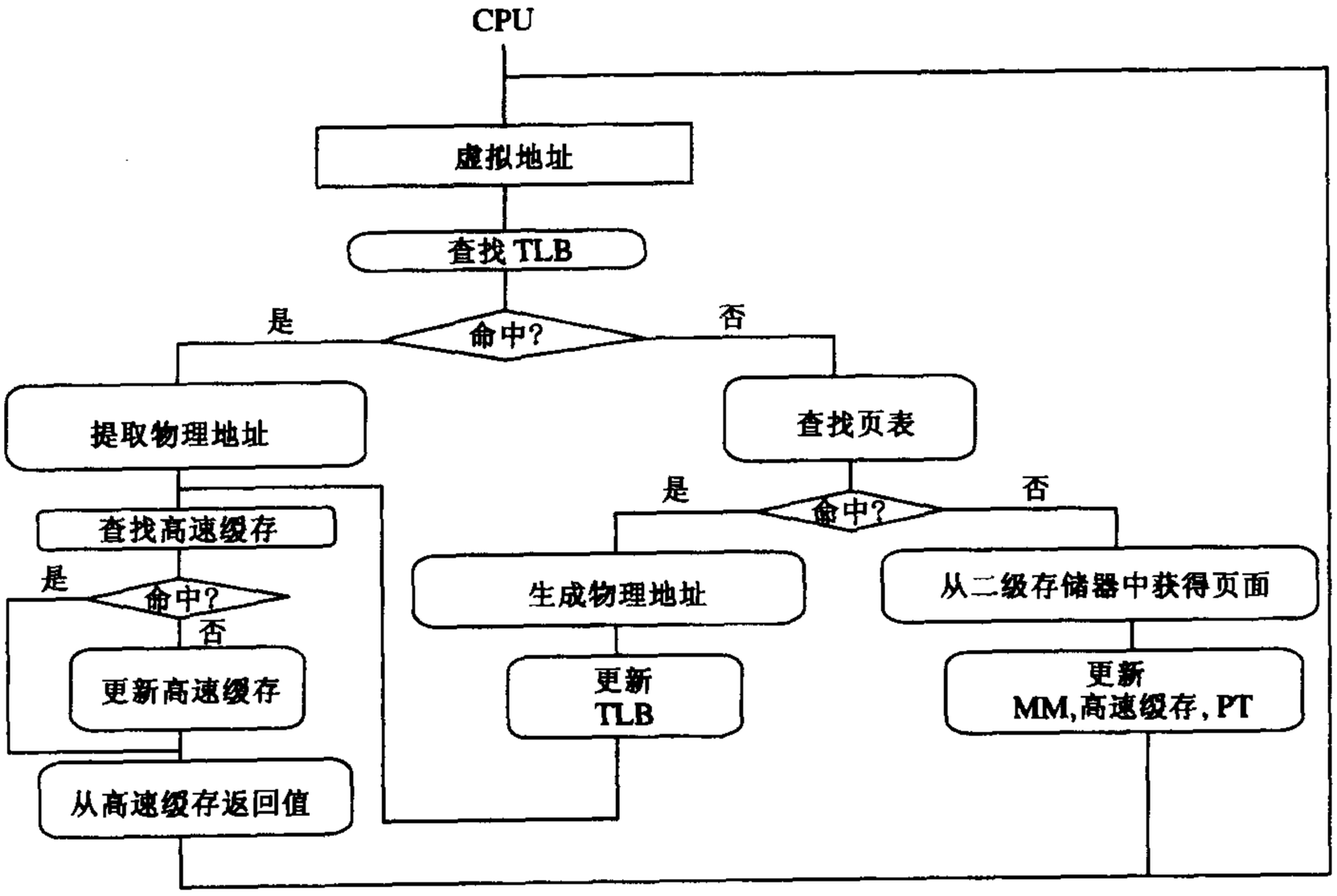


图 7.17 存储器层次结构的访问情形

第一级地址转换在接收到的虚拟地址和保存在 TLB 中的虚拟地址之间检查匹配，若找到匹配(TLB 命中)，则可获得相应的物理地址，这个物理地址可以用来访问高速缓存。若找到匹配(高速缓存命中)，则处理器请求的单元可以从高速缓存中送到处理器。而如果高速缓存缺失，则包含有目标单元的块从主存中复制到高速缓存(如前面讨论的)，然后请求的单元被送到处理器。

上面的情况是基于 TLB 命中的假设，若 TLB 缺失，则搜索页表(PT)以确定主存中包含目标元素的页是否存在。若 PT 命中，则生成相应的物理地址，并搜索包含请求元素的块(如前面所讨论的)，并且需要更新 TLB。另一方面，若 PT 缺失(表示一个页缺失)，则包含目标元素的页需要从硬盘中复制到主存中，然后将含有请求元素的一个块复制到高速缓存中，同时该元素被送到处理器。最后需要更新页表、主存和高速缓存。处理器对该虚拟地址的后续请求导致 TLB 的更新。

7.2.7　分段

一个段是一个可变大小的连续单元块，操作系统利用段在主存和硬盘空间中重新定位完整的程序，多个程序间可以共享段，有相应的机制用来保护非授权的访问和执行。若访问没有被专门允许，就不可能从一个段进入另一个段。数据段和代码段被分开，在取指令时不能改变代码段中的信息，同时也不可能执行数据段中的数据。

7.2.8　段地址转换

为了支持分段，处理器发出的地址需要包括段号(基址)和段内移位(或偏移量)。地址转换是直接通过段表进行的，目标段的起始地址是通过将段号加到段表指针的内容而得到的。段表中一个重要的值是物理段基地址，将后者加到偏移量上可以得到所需的物理地址。如图 7.18 所示为段地址转换过程。

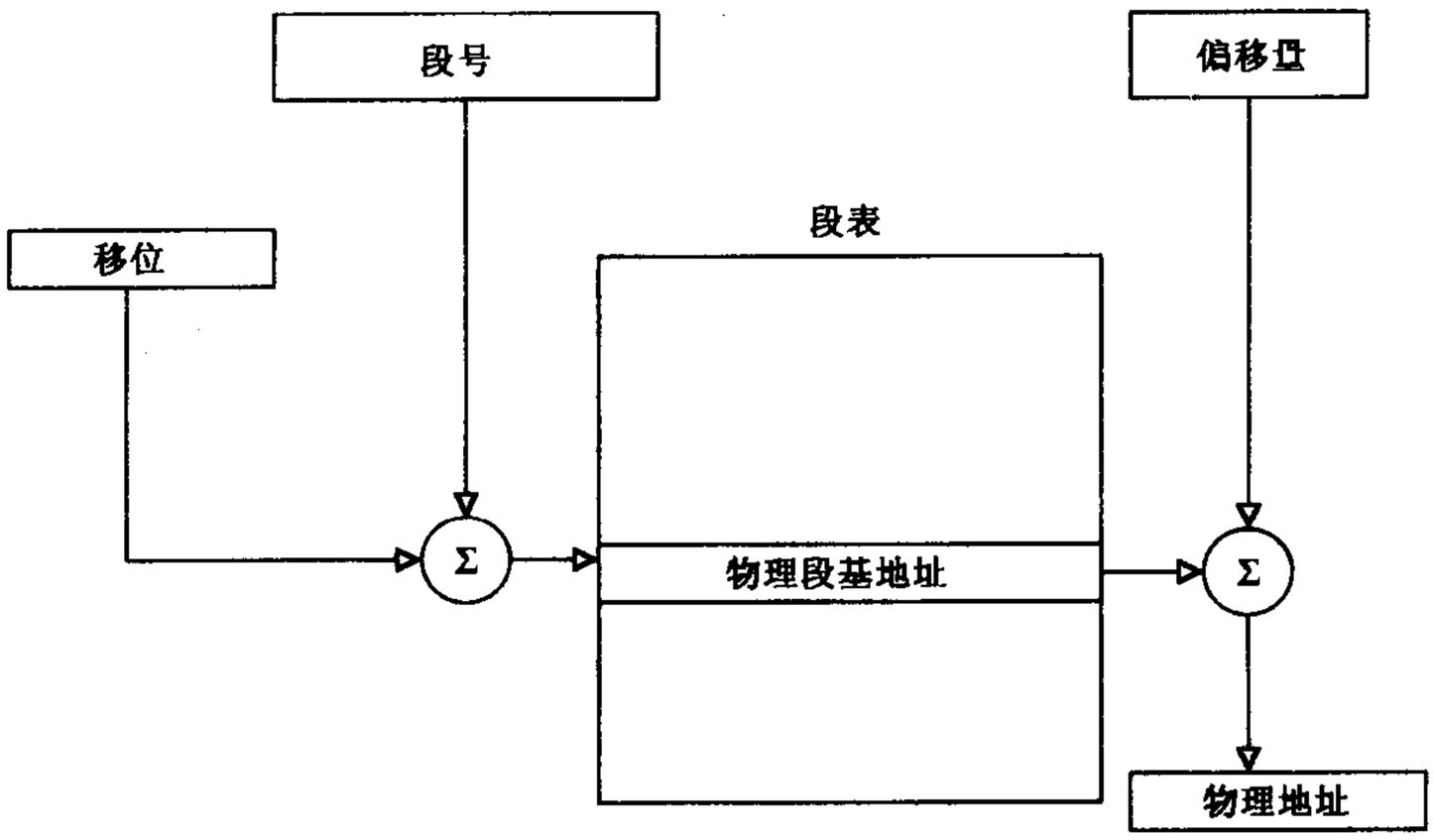


图 7.18　段地址转换

在段表中可能还包括的其他信息如下：

1. 段的长度；
2. 存储保护(只读、只执行或系统级操作等)；
3. 替换算法(类似于分页系统的方法)；

4. 布局算法(在主存中找一个适合的位置用于保存调入的段),例如:

(a) 首次适合;

(b) 最佳适合;

(c) 最坏适合。

7.2.9 页式分段

许多系统结合了分段和分页,每个段被分成一定数量、等大小的页,在主存和硬盘之间传输数据的基本单位是页,即在任一给定时刻,主存可能包括了不同段的一些页。在这种情况下,虚拟地址被分为段号、页号和页内移位。地址转换同上面讲述的一样,只不过是将从段表中获得的物理段基地址加到虚拟页号上从而得到页表中对应的入口,从页表中得到的是页的物理地址,再与虚拟地址中的字字段一起形成物理地址,如图 7.19 所示为这种页式分段地址转换的情形。

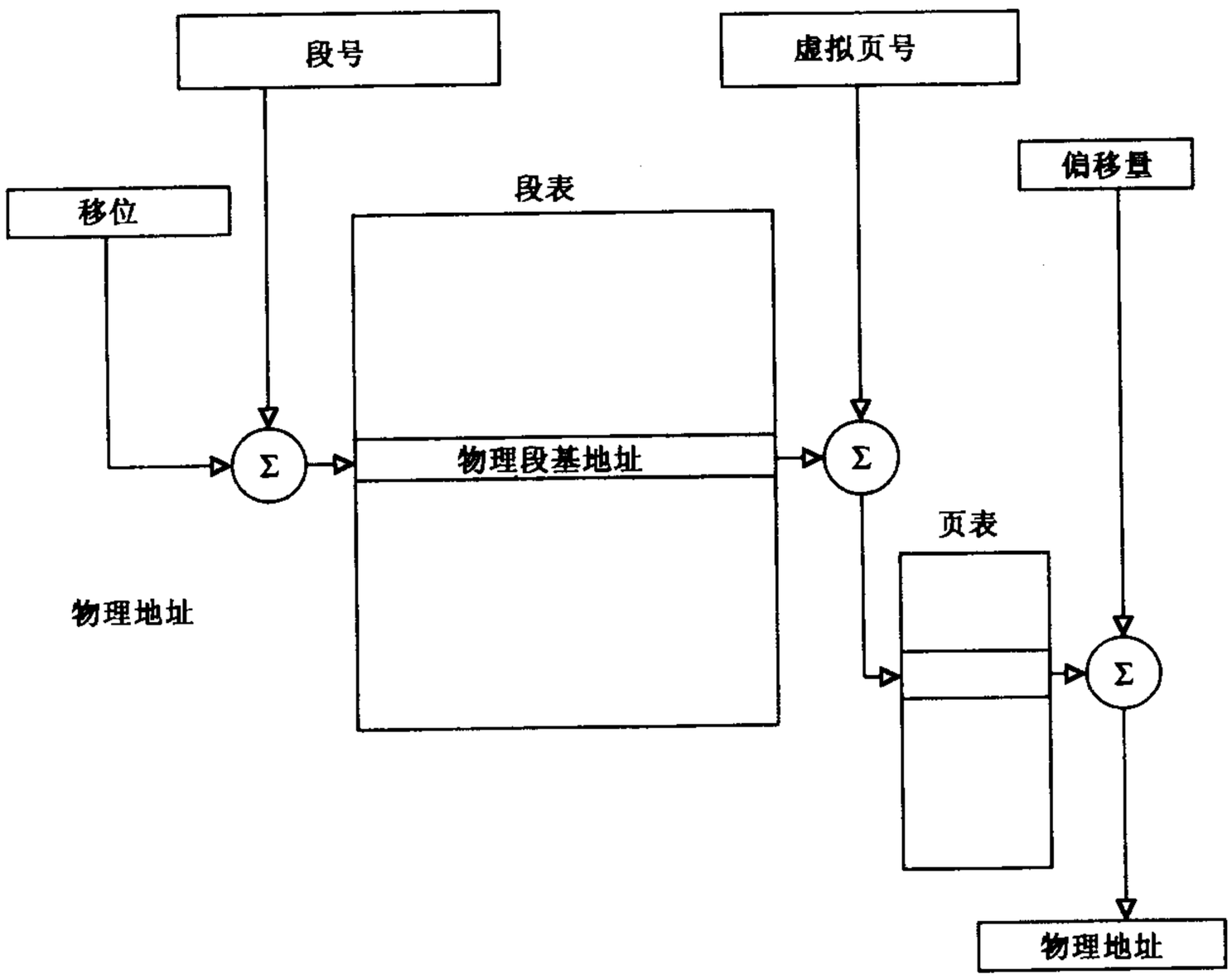


图 7.19 页式分段地址转换

7.2.10 奔腾存储管理

在奔腾处理器中,分段和分页可以单独使用,也可以设为不可用,存储器有 4 种不同的存储形式:

1. 不分段不分页式存储;

2. 不分段分页式存储;

3. 分段不分页式存储;

4. 分段分页式存储。

对于分段,16 位的段号(其中 2 位用于保护)和 32 位的偏移量产生一个段式的虚拟地址空间,其大小为 2^{46} = 64 TB。虚拟地址空间被分为两部分:一半是 8 K × 4 GB,是全局的且被所有进程共享,另一半是局部的,分别给不同的进程。

对于分页，奔腾处理器采用了一个二级表查找分页系统。第一级是一个有 1024 个项的页目录，即 1024 个页组，每个页组有其自身的页表，且每个组的长度为 4 MB。每个页表包含 1024 项，每项对应于一个 4 KB 的页。

7.3　只读存储器

随机访问存储器和高速缓冲存储器都是易失性(volatile)存储器，易失性存储器可以定义为关掉电源即丢失其内容的存储器，而非易失性存储器即使在关掉电源后，也能保存存储的信息。计算机需要易失性存储器，也需要非易失性存储器，需要用到非易失性存储器的计算机应用有计算机引导程序、微代码控制和视频游戏盒等，一种很典型的非易失性存储器，只读存储器(ROM)，还可以用来实现组合逻辑功能。

ROM 技术的实现也发展了许多年，ROM 早期的实现称为掩模编程 ROM。在这种情况下，根据用户提供的特定编码模式对定制的 ROM 进行一次性的编程，一个 4×4 CMOS ROM 芯片结构如图 7.20 所示。

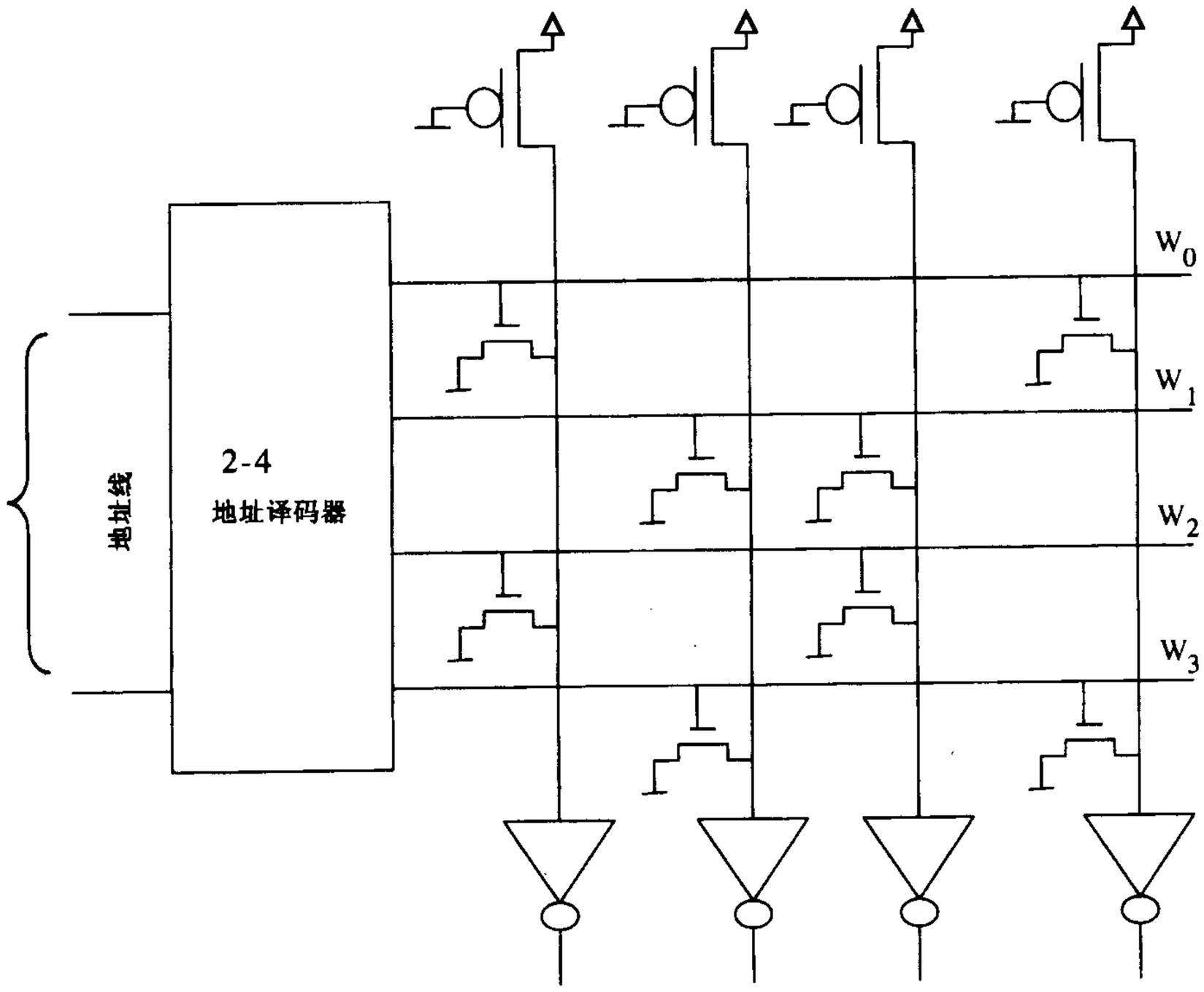


图 7.20　一个 4×4 CMOS ROM 芯片的示例

在该图中，n 型晶体管中保存值 1，一个 2-4 地址译码器用来产生 4 个字线，每条字线用来激活一行晶体管。当 1 出现在字线上时，相应的晶体管导通，于是相应的位线被置为 0，与之对应的位线输出端的反向器输出 1。表 7.3 说明了这 4 个 ROM 单元所存储的模式。

掩模编程 ROM 起初是用来保存机器微代码、桌面系统引导程序和视频游戏盒程序。由于只能由厂家进行一次性的编程，因此掩模编程 ROM 缺乏灵活性。若用户希望就地对他们的

ROM 进行编程,则可采用另一种不同类型的 ROM,称为可编程 ROM(PROM)。在这种情况下,采用熔断器替代晶体管放置在字线和位线的交叉位置,用户通过有选择地熔断合适的熔断器对 PROM 进行编程。在这些特定熔断器中通过高电流,就可使得它们熔断,从而实现编程,这一过程称为“烧 ROM”。

表 7.3 4 个 ROM 单元所存储的模式

地址线	激活的字线	输出模式
00	W_0	1001
01	W_1	0110
10	W_2	1010
11	W_3	0101

虽然 PROM 有一定的灵活性,但是仍然非常有限,因为它只能(由用户)进行一次编程。第三种 ROM,即可擦写 PROM(EPROM),是可以多次编程的,即它可以允许擦去保存的数据进而保存新的数据。为了提供这一灵活性,EPROM 采用特种晶体管制成,这些晶体管有两种状态,即正常状态和失效状态。一个失效的晶体管可以视为一个处于关闭状态的开关,而一个正常晶体管可以被编程为全打开状态,这是通过感应一定量的电荷使其在门电路下被捕获实现的。失效晶体管可以通过消除感应电荷从而变为正常的,这需要将这些晶体管暴露在紫外线下。将 EPROM 芯片暴露在紫外线下将会导致整个芯片内容被清除,这被认为是 EPROM 的主要缺点。PROM 和 EPROM 均用于中等大小系统的原型机中。

闪速 EPROM(FEPROM)是 EPROM 强有力的竞争对手,这是因为 FEPROM 相对于 EPROM 而言更紧凑、更快、更好清除,FEPROM 的清除时间远少于 EPROM。

另一种不同类型的 ROM 可以克服 EPROM 的缺点,称为 EEPROM。EEPROM 的清除可以有选择性地加电完成,即只是清除被选择的单元内容,而让其他单元内容保持不变。FEPROM 和 EEPROM 可以用在需要经常更新信息的应用中,例如可编程的电视机,VCR(录像机)和汽车中。

表 7.4 总结了上面讨论的各种不同类型 ROM 的主要特征。

表 7.4 不同 ROM 实现的特征

类型	成本	可编程性	典型应用
掩模编程 ROM	很便宜	制造一次性	微代码
PROM	便宜	就地一次性	原型机制造
EPROM	适中	多次	原型机制造
FEPROM	昂贵	多次	VCR & TV
EEPROM	很昂贵	多次	VCR & TV

7.4 小结

本章讨论的内容是第 6 章的延续。具体来讲,本章主要讨论了与主存内部和外部结构相关的设计问题;介绍了静态 RAM 单元设计,并重点讨论了读写操作。在虚拟存储方面,本章首先讨论了地址转换相关的内容,讨论和比较了 3 种地址转换技术,它们分别是直接、相联和组相联技术。本章说明了采用 TLB 可以提高平均访问时间;介绍了 3 种替换技术,分别是 FIFO,LRU 和时

钟替换;还介绍了分段分页系统。对虚拟存储的讨论以阐述奔腾 IV 处理器的虚拟存储器作为结束。本章最后涉及了一些有关 ROM 实现的内容。

习题

1. 一个计算机系统使用高速缓存和分页虚拟存储器,如图 7.21 所示。可以将其分成 5 种访问的组合,请说明这 5 种组合。根据表所示的信息,确定每种情况的概率和访问时间,再计算总体的平均访问时间。

TLB 地址转换和搜索	25 ns
为确定地址是否在缓存中的缓存搜索时间	25 ns
高速缓存访问时间	25 ns
主存访问时间	250 ns
硬盘访问时间	100 ms
TLB 命中率	0.9
高速缓存命中率	0.95
主存命中率	0.8

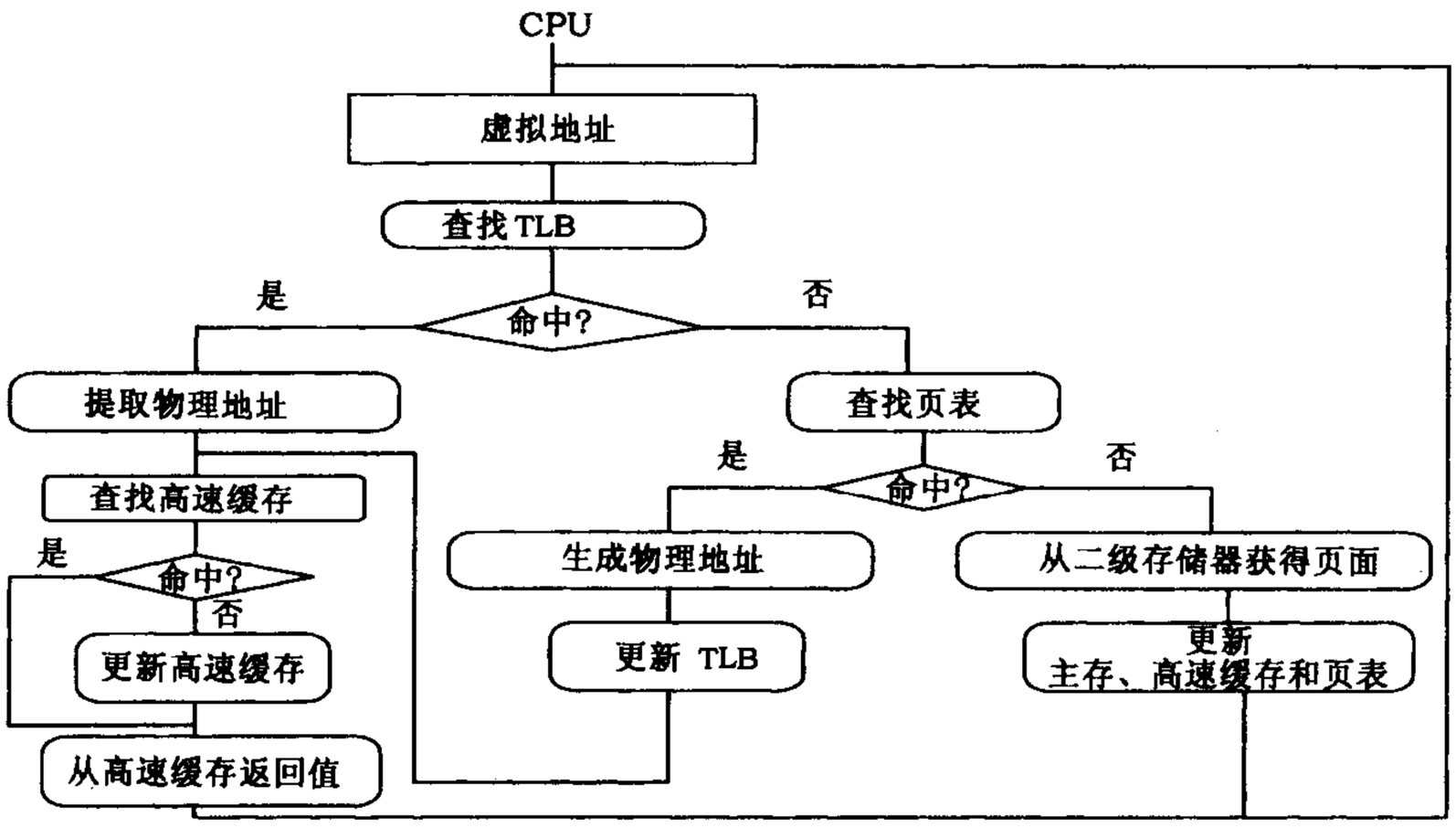


图 7.21　带有高速缓存、使用分页虚拟存储的计算机系统

2. 一个 64 × 64 字(元素)的数组需要进行“规则化”,即对每行确定其最大的元素,并使该行的所有元素除以这个最大元素。设虚拟存储器中的页包含 64 个字,在计算中分配 2 K 字用来保存其数据。当发生页缺失时,从硬盘中加载一页到主存所需的时间设为 100 ms。
   (a) 编写一个能够执行上述任务的简单代码段(用符号格式)。
   (b) 若数组元素按列主方式保存在虚拟存储器中,将发生多少次页缺失?
   (c) 若数组元素按行主方式保存在虚拟存储中,将发生多少次页缺失?
   (d) 针对情况(b)和(c),估计执行这种规则化所需的总时间。
3. 采用一定数量的 16 M × 1 位静态 RAM 芯片设计一个 64 M × 8 位存储器。设单个芯片有一条芯片选择线($\overline{CS}$)和一条读/写线(R/$\overline{W}$),计算所需的芯片数,并画出设计存储器的完整连接示意图。

4. 考虑下面的页请求流:1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,设主存由 4 个页帧构成。针对下面每种页替换算法,说明主存中页帧状态的轨迹,并估计命中率。
   (a) FIFO;
   (b) LRU;
   (c) FINUFO。
5. 考虑一个 20 × 20 的 2 维数组 A,数组以列主方式存储,主存有 5 个页帧,针对下面的初始化循环,计算产生了多少次页缺失。设采用 LRU 替换算法,所有页帧初始值为空,页的大小为 40 个元素。

```
for I = 1 to 20 do
for J = 1 to 20 do
A[I, J] = 0;
```

6. 该习题要求选择一种实际的计算机存储系统,该系统采用了高速缓存和虚拟存储机制,运用本章中所涉及的存储系统设计和分析方面的知识,描述并分析所挑选的系统。挑选的系统不限于奔腾 IV,PowerPC 和 Alpha AXP 21064 等。尽可能具体地通过例子和图表论证分析的结果,记住重点放在其中的分析和基本设计问题上,你可以采用任何参考资料,但要确保在参考列表中列出它们。

参考文献和推荐读物

S. D. Burd, *Systems Architecture*, 3rd ed., Thomson Learning Ltd, Boston, 2001.

H. Cragon, *Memory Systems and Pipelined Processors*, Jones and Bartlett: Sudbury, MA, 1996.

V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed., McGraw-Hill, NY, 2002.

J. L. Hennessy, and Patterson, D. A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 1996.

V. P. Heuring, and H. F. Jordan, *Computer Systems Design and Architecture*, Addison- Wesley, NJ, USA, 1997.

D. A. Patterson, and J. L. Hennessy, *Computer Organization & Design: The Hardware/ Software Interface*, Morgan Kaufmann, San Mateo, CA, 1994.

H. S. Stone, *High-Performance Computer Architecture*, Addison-Wesley, Amsterdam, Netherlands, 1987.

B. Wilkinson, *Computer Architecture: Design and Performance*, Prentice-Hall, Hertfordshire, UK, 1996.

参考网站

http://www.sysopt.com

http://www.intel.com

http://www.AcerHardware.com

http://www.pmc-sierra.com/products/details/rm7000a

http://physinfo.ulb.ac.be/divers_html/PowerPC_Programming_Info/into_to_ppc/pc2_hardware.html

第8章　输入/输出的设计与组成

在了解了指令集设计、汇编语言程序设计、处理器设计和存储器设计这些基础概念之后，我们现在将注意力转向与输入/输出(I/O)设计和组成相关的问题。需要在一开始强调的是输入/输出在任何一个现代计算机系统中都扮演着重要的角色。因此对输入/输出的操作、设备和接口的基础有清楚地了解和评价是非常重要的。

各种输入/输出设备的特征有很大的差异。输入设备(同样也在输出设备)之间的一个区分因素是数据处理速度，它是指一台设备每秒钟处理字符的平均数量。例如，键盘的数据处理速度大约为每秒 10 字符(字节)，而扫描仪发送数据的速度则可以达到大约每秒 200 000 字符(字节)。激光打印机输出数据的速度可以达到大约每秒 100 000 字符(字节)，而图形显示器输出数据的速度可以达到大约每秒 30 000 000 字符(字节)。

敲击计算机键盘上的一个按键会将该字符(以 ASCII 码的形式)发送到计算机中。在下一个字符发送到计算机之前所经过的时间取决于用户打字的技巧甚至是思考的速度。通常情况下用户知道要输入什么，但有时他们需要在敲击键盘的下一个按键前进行思考。因此键盘输入的速度缓慢而且具有自然的突发性。把计算机宝贵的时间消耗在等待一个低速输入设备的输入是在浪费时间。因而需要有一个机制，当设备准备好后它向处理器发出中断请求以获得注意，这称为计算机和输入/输出设备间的中断驱动(interrupt-driven)通信(参见 8.3 节)。

考虑磁盘的情况。典型的磁盘必须具备以每秒超过几百万字节的速度传输数据的能力。以逐字节(8 位)甚至逐字(32 位)的方式传输数据是很费时的，因此通常数据是以块的形式，即整个程序的形式传输。同时还需要提供一个机制，允许磁盘在没有 CPU 干预的情况下传输海量数据。这就使得在磁盘和存储器间传输海量数据时，CPU 能处理其他有用的操作。这就是将在 8.4 节中讨论的直接存储器访问(DMA)机制的本质。

我们的讨论将从 8.1 节中所提供的一些基本概念开始。

8.1　基本概念

如图 8.1 所示为单总线系统，即在一个给定计算机系统中，将处理器和存储器与输入设备(比如键盘)、输出设备(比如图形显示器)以相对简单的方法相连接。一个由所需地址线、数据线和控制线组成的单一总线被用来连接图 8.1 中系统的各个组件。

处理器与存储器交换数据的方法已经在第 6 章和第 7 章中做了解释。这里我们所关心的是处理器与输入/输出(I/O)设备交换数据的方法。本章的引言部分已经指出处理器处理信息的速度与输入/输出设备的速度之间存在很大的差异。一个能适应这种速度差异的简单方法是让输入设备，比如键盘，将用户敲击的字符存放到寄存器(输入寄存器)中，表示该字符对处理器可用。当该字符被处理器取走时，就表示输入设备可以继续输入下一个字符，等等。类似地，当处理器有一个字符要输出(显示)时，就将这个字符存放到一个特定的、专门用来与图形显示器通信的寄存器(输出寄存器)中。当这个字符被图形显示器取走时，表示处理器可以继续输出下一个字符，等等。这种处理器和 I/O 设备间通信的简单方法，称为 I/O 协议，它依赖于输入和输出寄存器的可用性。在典型计算机系统中，有一些输入寄存器，每个都属于某个特定的输入设备。

也有一些输出寄存器,每个都属于某个特定的输出设备。而且必须采用关于处理器能对哪些输入和输出寄存器进行寻址的机制。存在不止一种方式能满足上面所提到的需求。下面解释其中两种方法。

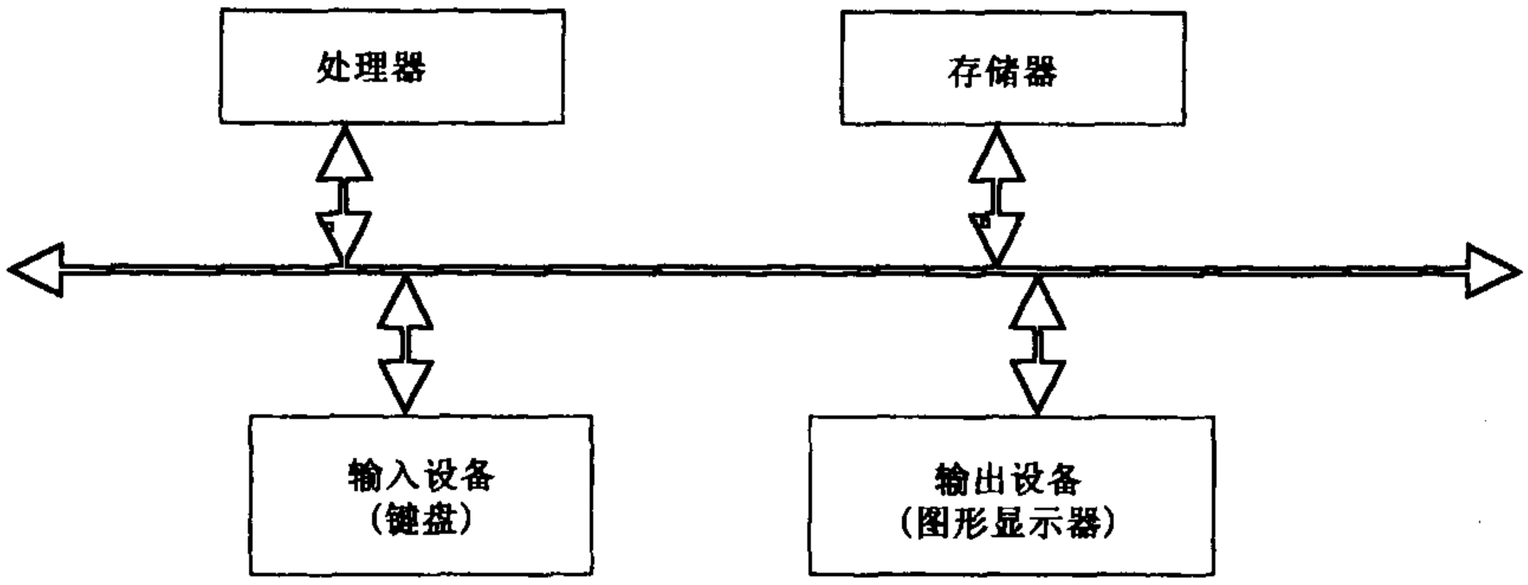


图 8.1 单总线系统

第一种方式给 I/O 设备分配了专门的地址,与分配给存储器的地址空间隔离。执行在输入设备地址上的一条输入指令,将使存储在该设备的输入寄存器中的字符传送到 CPU 中某个特定的寄存器。类似地,执行在输出设备地址上的一条输出指令将使存储在 CPU 中某个特定寄存器中的字符传送到输出设备的输出寄存器。这种方式如图 8.2 所示,称为共享 I/O。在这种情况中,来自 CPU 的地址线和数据线可以在存储器和 I/O 设备间共享,但必须使用单独的控制线,以满足执行输入和输出指令的需要。典型的计算机系统中存在不止一个输入和输出设备。因此有必要使用地址译码电路来识别设备,也有必要为每个输入和输出设备提供状态寄存器。输入设备的状态,不论它是否准备好向处理器发送数据,都应该存储在该设备的状态寄存器中。类似地,输出设备的状态,不论是否准备好从处理器接收数据,都应该存储在属于该设备的状态寄存器中。输入(输出)寄存器、状态寄存器和地址译码电路是 I/O 接口(模块)的主要组件。

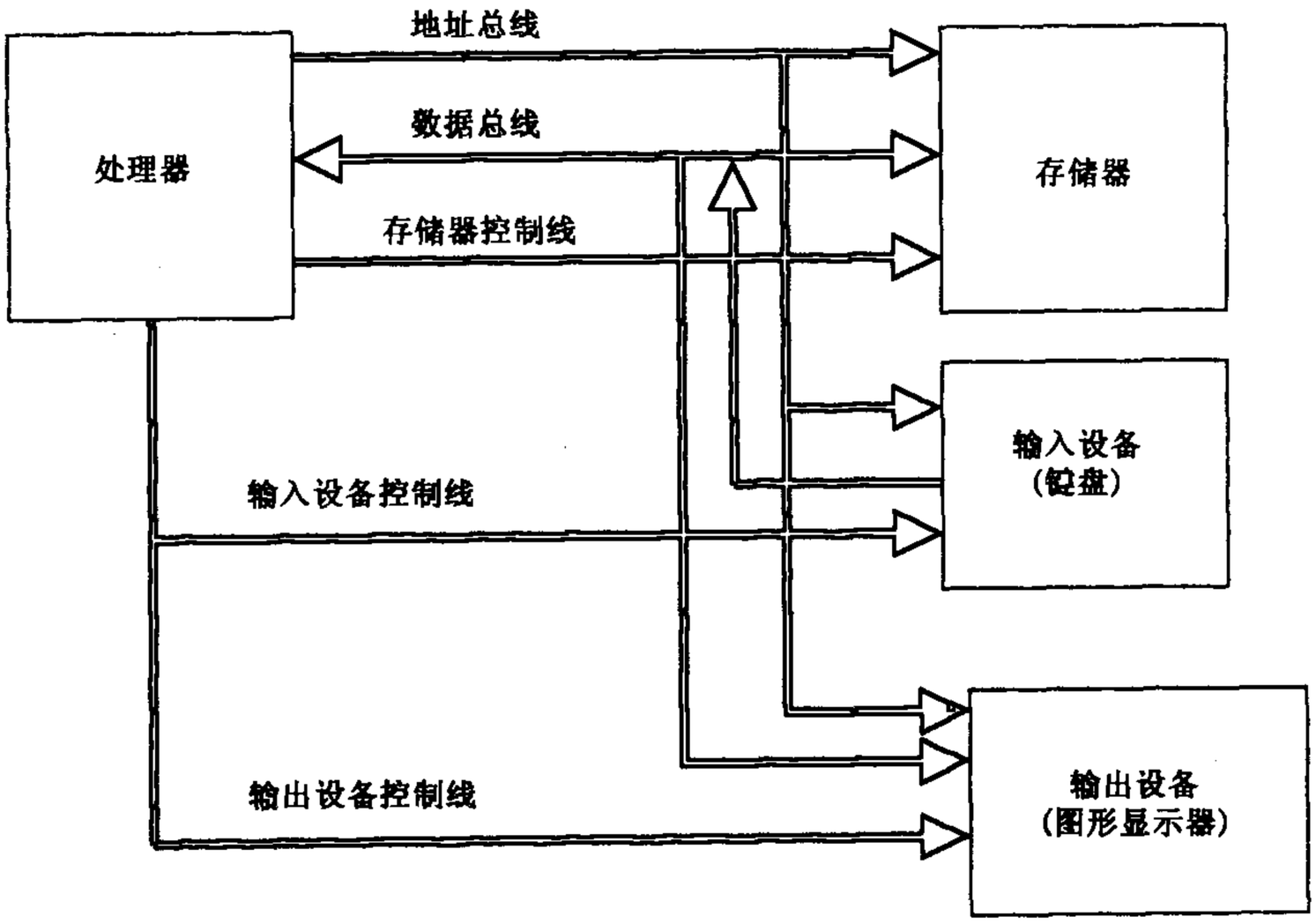


图 8.2 共享 I/O 方式

共享 I/O 方式的主要优点是分离了存储器地址空间和 I/O 设备地址空间。它的主要缺点是需要在处理器指令集中加入特殊的输入和输出指令。共享 I/O 方式主要是 Intel 在使用。

第二种 I/O 方式是将输入和输出寄存器像通常存储单元一样处理。在这里，对应于一个输入设备的输入寄存器地址的读入操作，比如 Read Device 6，相当于从 Device #6 的输入寄存器执行一个输入操作。类似地，对应于一个输出设备的输出寄存器地址的写入操作，比如 Write Device 9，相当于对 Device #9 的输出寄存器执行一个输出操作。这种方式称为存储器映射 I/O，如图 8.3 所示。

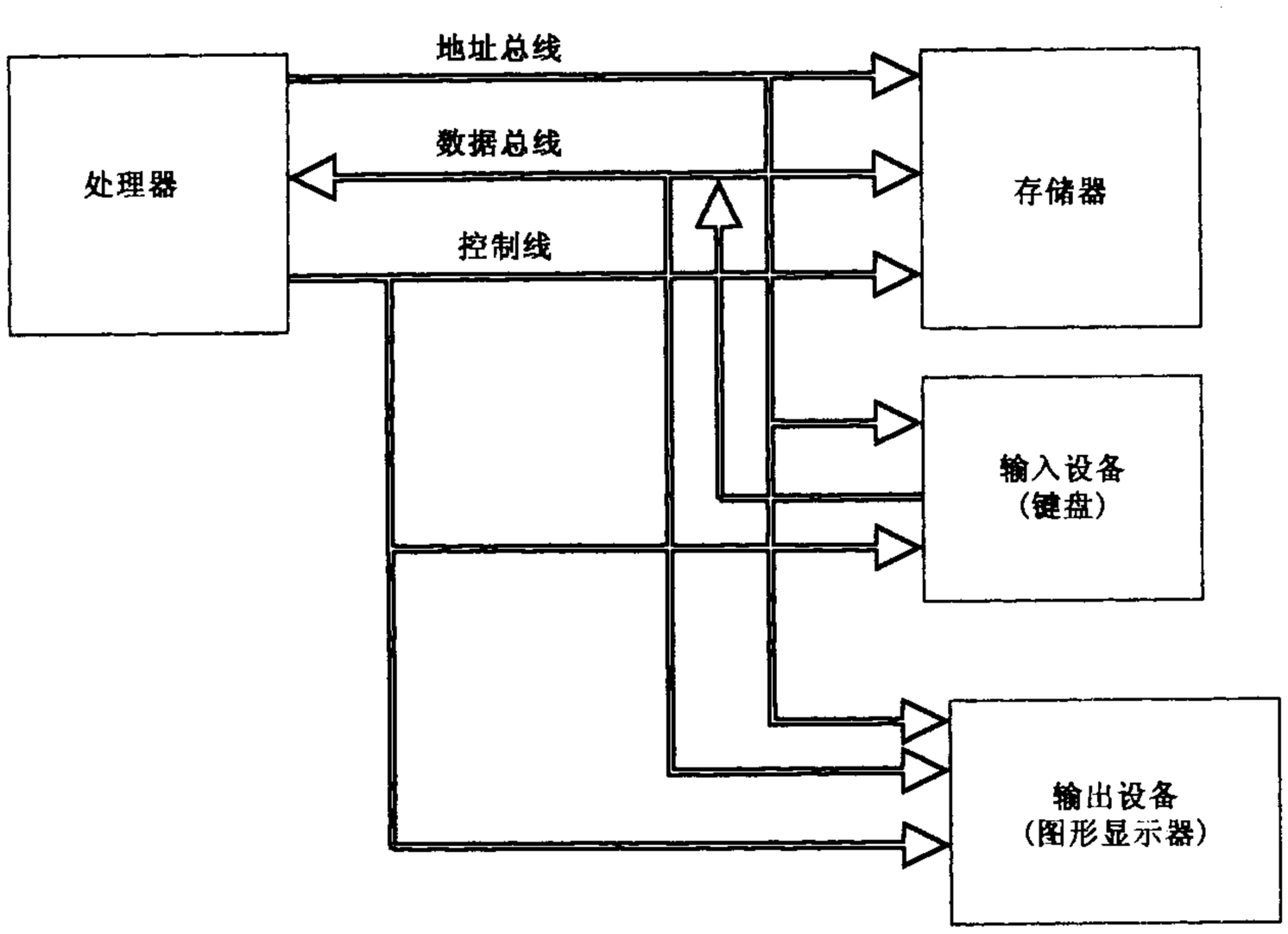


图 8.3　存储器映射 I/O 方式

存储器映射 I/O 的主要优点是分别使用处理器的读和写指令来执行输入和输出操作。它消除了使用特殊 I/O 指令的需要。它的主要缺点是需要为 I/O 设备的寻址保留一部分存储器地址，也就是说，减少了可用的存储器地址空间。存储器映射 I/O 主要是 Motorola 在使用。

8.2　编程 I/O

本节中，我们介绍处理器和 I/O 设备之间通信所需的主要硬件组件，也会提到有关通信发生(协议)的方法。协议必须以运行在 CPU 控制之下的例程的形式来编程。例如，考虑在共享 I/O 方式中，一个来自设备 6(可能是键盘)的输入操作。同时我们也假设在这个例子中，有 8 个不同的输入/输出设备连接到该处理器，如图 8.4 所示。

I/O 必须遵循下列的协议步骤(程序)：

1. 处理器执行来自设备 6 的一条输入指令，比如 INPUT 6。执行该指令的结果是发送设备编号到每个输入设备所属的地址译码电路，以识别所涉及的特定输入设备。在这里，Device #6 所属的译码器输出激活，而所有其他译码器输出则失效；
2. 在指定输入设备(Device #6)中保存数据的缓冲器(在图中我们假设有 8 个这样的缓冲器)将被地址译码电路输出激活；

3. 被激活缓冲器的数据输出对数据总线可用；
4. 指令译码将会把数据总线上可用的数据装入到 CPU 的寄存器输入中，通常是累加器。

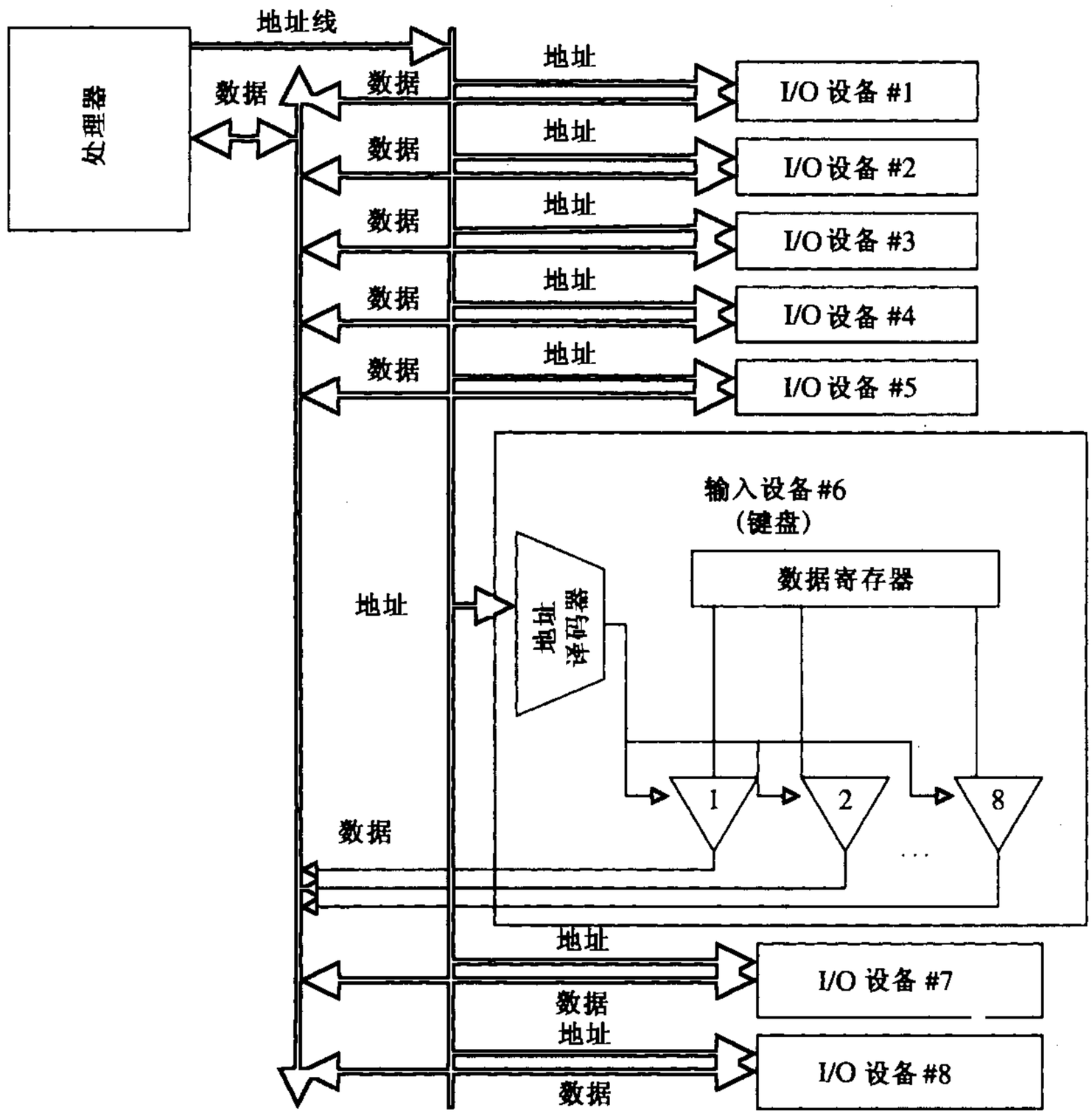


图 8.4　8 个 I/O 设备连接到处理器的例子

输出操作能以与前面解释过的输入操作类似的方式执行。惟一的区别是数据传输的方向。数据将从指定的 CPU 寄存器传输到指定输出设备的输出寄存器。以这种方式执行的 I/O 操作称为编程 I/O，它们在 CPU 的控制下执行。每个输入和输出操作都将执行一个完整的指令周期：取指、译码和执行。编程 I/O 在每次传送一个字符的情况中特别有用，比如键盘和字符模式的打印机。虽然简单，但是编程 I/O 的速度比较缓慢。

在上面关于编程 I/O 的描述中忽略了一点，就是如何处理 I/O 设备和处理器之间的巨大速度差距。必须采用一种机制来确保一个发送到输出设备（比如屏幕）的输出寄存器中的字符不会（由于处理器的高速度）在显示前被处理器改写，同样，键盘的输入寄存器中一个字符只能被处理器读取一次。这引入了 I/O 设备的状态问题。可以实现一种机制，它需要每个输入设备接口的状态位（B_{in}）可用，而且每个输出设备接口的状态位（B_{out}）可用。当输入设备，比如键盘有一个字符在它的输入寄存器中可用时，它通过设定 $B_{in}=1$ 来表示。处理器中的程序能不间断地监视 B_{in}。当程序检查到 $B_{in}=1$ 时，它会认为该设备的输入寄存器中有一个字符可用。通过执行上面所解释的协议来读入这个字符。一旦字符被读入，程序将 B_{in} 重置为 0，这样避免了多次读入同样的字符。类似地，只有当 $B_{out}=0$ 时，处理器才能够将一个字符放到输出设备比如屏幕的输出寄存器中。当屏幕显示了这个字符后，它会将 B_{out} 设置为 1，向监控 B_{out} 的程序表示屏幕

已准备好接受下一个字符。检查 I/O 设备的状态,以确定它们是否准备好接受和/或发送字符的过程称为软件 I/O 轮询。硬件 I/O 的轮询方式如图 8.5 所示。

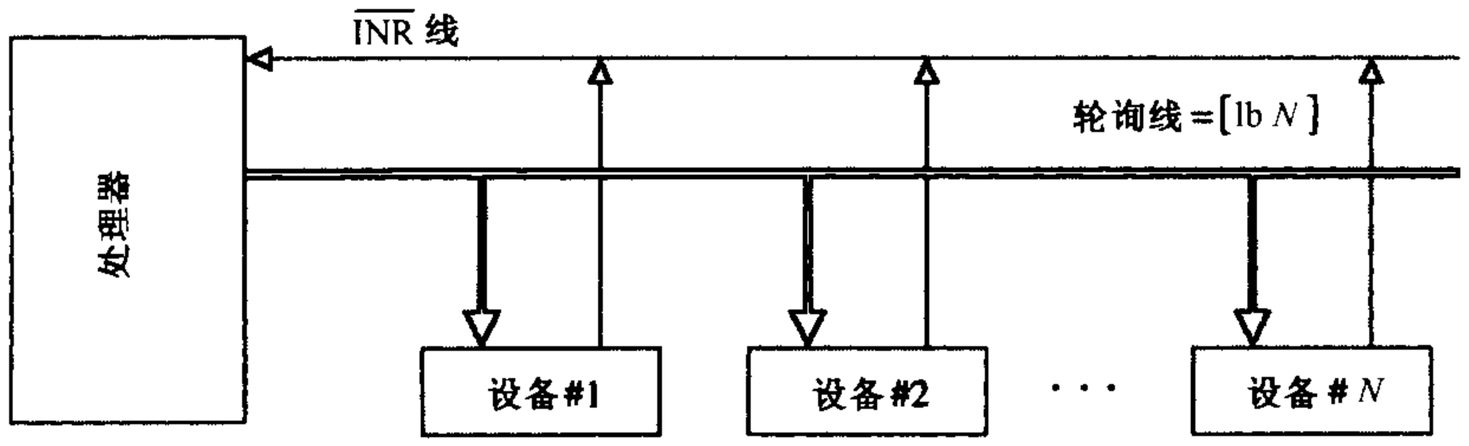


图 8.5　硬件轮询模式

在图中,N 个 I/O 设备访问中断线$\overline{\text{INR}}$。在辨认出到达的请求(称为中断请求)后,处理器轮询各个设备以确定发送请求的设备。这可以通过 lb N 条轮询线实现。请求设备的优先级将决定放置到轮询线中地址的顺序。先放置最高优先级设备的地址,接着是次优先级的,这样一直下去,最后放入的是最低优先级设备的地址。

除 I/O 轮询之外,另外两种机制也能够用来执行 I/O 操作。它们是中断驱动 I/O 和直接存储器访问(DMA)。它们将在下面两节中加以讨论。

8.3　中断驱动 I/O

通常需要有一个中断程序的正规流程来应对非正常事件,比如电源故障。中断也可以用来确认一个特定过程行为的结束,比如打印机通知计算机,它已经结束对输入寄存器中字符的打印,准备好接受其他字符。中断也可以用在分时系统中在多个不同程序间分配 CPU 时间。现代 CPU 的指令集通常包括了模拟硬件中断的指令。

当 CPU 被中断时,它需要暂停当前活动,处理中断条件(服务中断),然后从暂停处继续原来的活动。暂停处理器的当前活动需要结束当前指令的执行,保存处理器状态(大都采用将寄存器中的值压入堆栈的方式),传递控制(跳转)到所谓的中断服务例程(ISR)。对中断提供的服务依赖于中断源。例如,若是电源故障的中断,采取的行动是保存处理器所有寄存器和指针的值,这样在电源恢复后能保证继续正确操作。在 I/O 中断中,中断服务意味着执行所请求的数据传输。在完成中断服务后,处理器能从堆栈中弹出相关的值从而恢复到原先的状态。一旦处理器恢复到原来的状态,中断源能再次被激活。

上面描述的情形中忽略的一个要点是服务多个中断的问题,例如,当处理器正在服务一个中断时,另外又有一个中断发生了。对新到达中断的反应将依赖于新到达中断的优先级和正在服务的中断的优先级谁高谁低。若新到达中断的优先级比正在服务中的低或是相等,则它将等待直到处理器完成现有中断服务。另外一种情况,若新到达中断的优先级比正在服务中的高,例如在服务 I/O 中断时发生电源故障中断,则处理器必须将它的状态压入到堆栈中,并为具有更高优先级的中断服务。正是由于使用了压栈和退栈的操作以保存和恢复正确的处理器状态,才能保证正确地处理多个中断。例如为了服务第一个中断,将 STATUS 1(状态 1)压入堆栈,在收到第二个中断时,STATUS 2 将被压入堆栈。完成第二个中断服务后,STATUS 2 退栈,而在完成第一个中断服务后,再使 STATUS 1 退栈。

中断设备可以通过在中断请求后发送代码的方式,向处理器表明自己身份。由给定 I/O 设备发送的代码可以表示它的 I/O 地址或是该设备 ISR 的起始存储器地址位置。这种模式称为向量中断。

8.3.1 中断硬件

在上面的讨论中,我们假设处理器在服务中断前已经识别出中断的产生。计算机以连接处理器的专用中断线的形式提供了中断硬件的能力。这些线用来向处理器发送中断信号。在 I/O 的情况中,由于存在不止一个的 I/O 设备,处理器必须提供一种机制以处理并发的中断请求,并识别中断设备。有两种基本方法能够完成这个任务。第一种称为菊花链总线仲裁(DCBA),第二种称为独立源总线仲裁(ISBA)。

按照图 8.6(a)所示的 DCBA 方式,I/O 设备向中断请求线$\overline{\text{INR}}$发送它们的中断请求(与轮询方式相似)。当识别出一个到达的中断请求后,处理器通过菊花链许可线(GL)发送它的许可到发出请求的设备以开始其与处理器的通信。许可线遍历所有的设备,从与处理器最接近的设备开始,然后到达下一个设备,再下一个,直到到达最后一个设备(Device # N)。若设备 # 1 发出过请求,则它将保留许可信号,然后开始与处理器通信。另一种情况,若设备 # 1 没有发出过中断请求,它将把许可信号传递到设备 # 2,这样的过程将重复多次。在多个请求的情况中,DCBA 方式的安排给物理上最靠近处理器的设备以最高的优先级,而离处理器最远的设备以最低的优先级。

对于图 8.6(b)所示的 ISBA 方式,每个 I/O 设备有自己的中断请求线,通过它能够发送中断请求,且与其他设备独立。类似地,每个设备有自己的许可线,通过它接收到对请求的许可信号,这样它就能开始与处理器通信。ISBA 中的 I/O 设备优先级与设备位置无关,但它需要优先级仲裁电路来处理并发的中断请求。

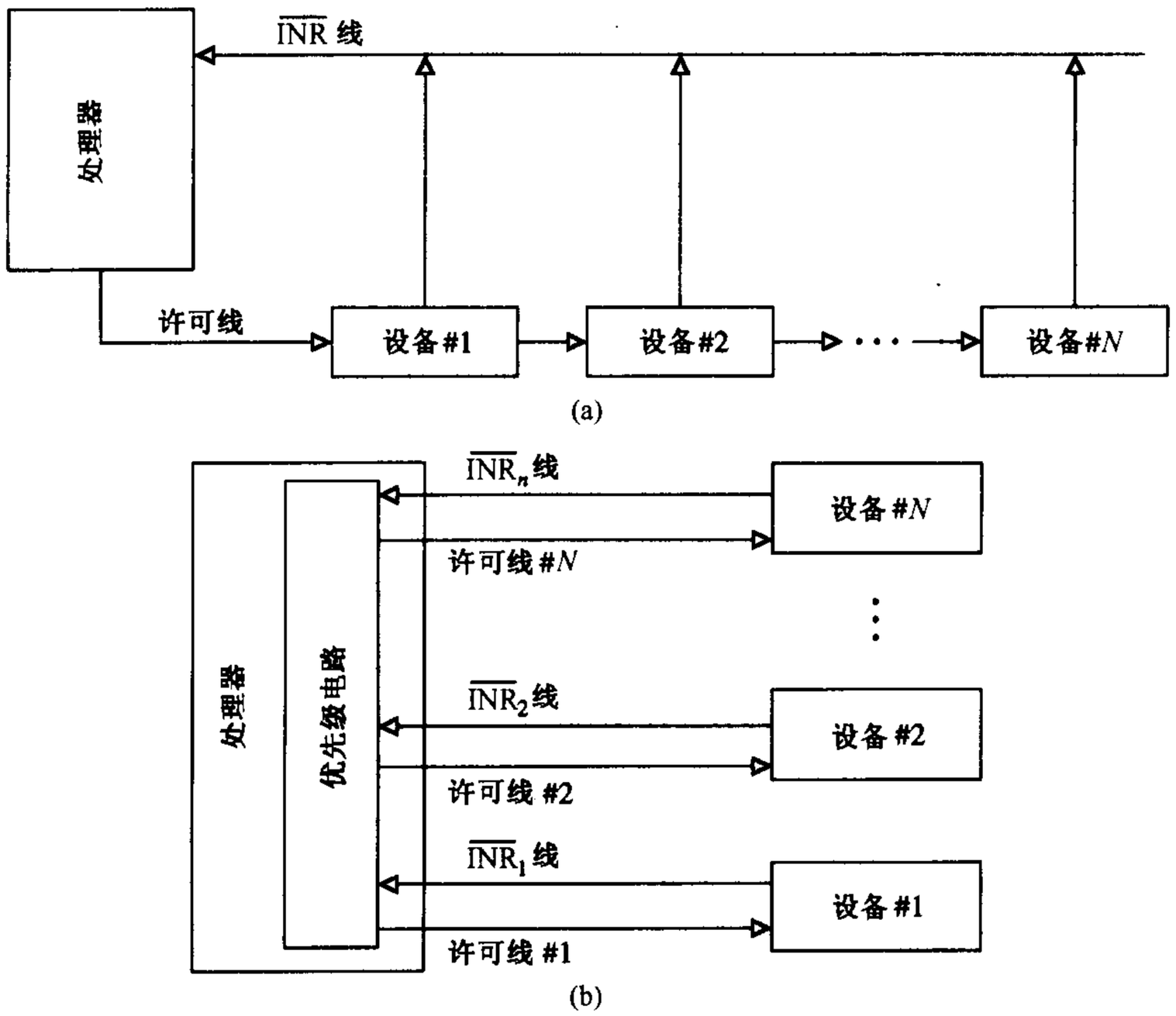


图 8.6 中断硬件模式:(a)菊花链中断方式;(b)独立中断方式

8.3.2　操作系统中的中断

当一个中断发生时,操作系统获得控制权。操作系统保存了被中断进程的状态,分析中断,然后传递控制权到合适的例程来处理中断。中断可分为几种类型,其中包括 I/O 中断。I/O 中断通知操作系统 I/O 设备已经结束或是挂起它的操作,现在需要某种 CPU 的服务。为了处理一个中断,必须保存当前进程的上下文,同时调用中断处理例程。这个过程称为上下文切换。一个进程的上下文分为两个部分:处理器上下文和存储器上下文。处理器上下文是指 CPU 寄存器的状态,包括程序计数器(PC)、程序状态字(PSW)和其他寄存器。存储器上下文是指程序存储器的状态,包括程序和数据。中断处理程序是一个处理不同类型中断的例程。

操作系统必须为程序提供保存上下文的区域,同时也必须提供一个有组织的方法为被中断的进程分配和解除分配存储器。当中断处理例程完成处理中断后,CPU 要么分配到原先被中断的进程,要么分配给优先级最高、已经就绪的进程。这依赖于被中断的进程是抢占式的还是非抢占式的。若进程是非抢占式的,它将会再次得到 CPU。先恢复上下文,然后将控制交还给被中断的进程。

如图 8.7 所示为 I/O 操作中涉及的软件层次。首先是程序通过 I/O 调用发出一个 I/O 请求。这个请求被传递到 I/O 设备。当设备完成 I/O 后,发送一个中断,然后调用中断处理程序。最后将控制返回给发起 I/O 的进程。

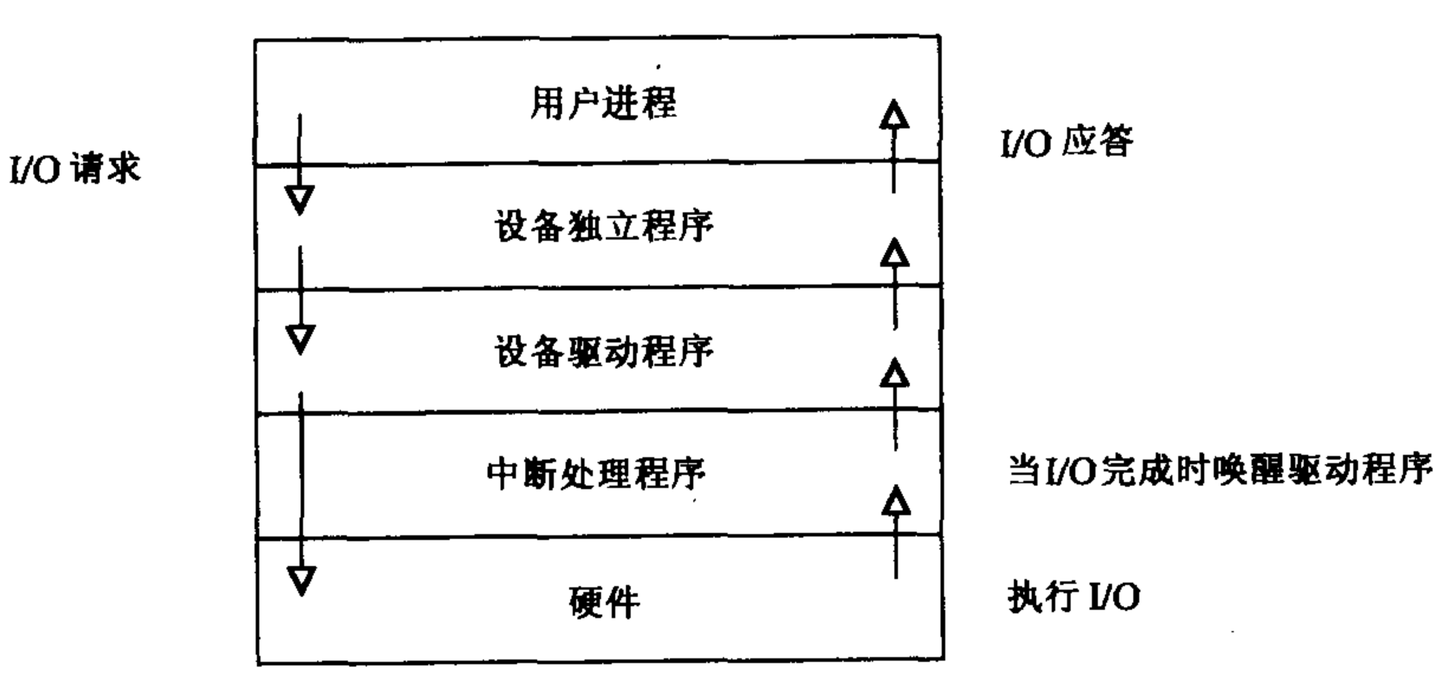


图 8.7　分层的 I/O 软件

例 8.1　80x86 的中断体系结构。80x86 处理器有两个硬件中断针脚。它们被标号为 INTR 和 NMI。NMI 是一个非可屏蔽中断,这表示它不能被阻塞,处理器必须对它进行响应。NMI 输入通常是为重要系统功能保留的。INTR 输入是位于 CPU 和可编程中断控制器(8259A PIC)之间的可屏蔽中断请求线。INTR 上的中断可以分别使用指令 STI(设置中断标志)和 CLI(清除中断标志)加以激活和使其失效。

中断处理程序又称为中断服务例程(ISR)。每个中断服务例程的地址都保存在中断向量表(IVT)的四个连续的存储器单元中。IVT 保存了指向每种中断类型 ISR 的指针。当一个中断发生时,将向处理器提供一个 8 位的类型编号,用来识别表中适当的项。

当一个中断由设备生成后,它进入到可编程中断控制器(PIC)中。多个中断可以同时生成,但是它们都被 PIC 所缓冲。PIC 将决定这些中断中的哪一个应该被转送给 CPU。为了提醒 CPU 有一个未完成的中断正在等待被处理,PIC 向 CPU 发送一个中断请求(INTR),CPU 然后在适当时

间以中断确认(INTA)加以回复。与此同时 PIC 将发送一个与设备有关的 8 位中断类型编号到总线上,这样 CPU 就能识别应该调用哪个中断处理程序。当出现多个中断等待的情况时,PIC 只有在收到来自当前 ISR 的中断命令的结束信号后,它才会发送下一个中断请求给 CPU。如图 8.8 所示为用来决定调用哪个 ISR 的简单协议。

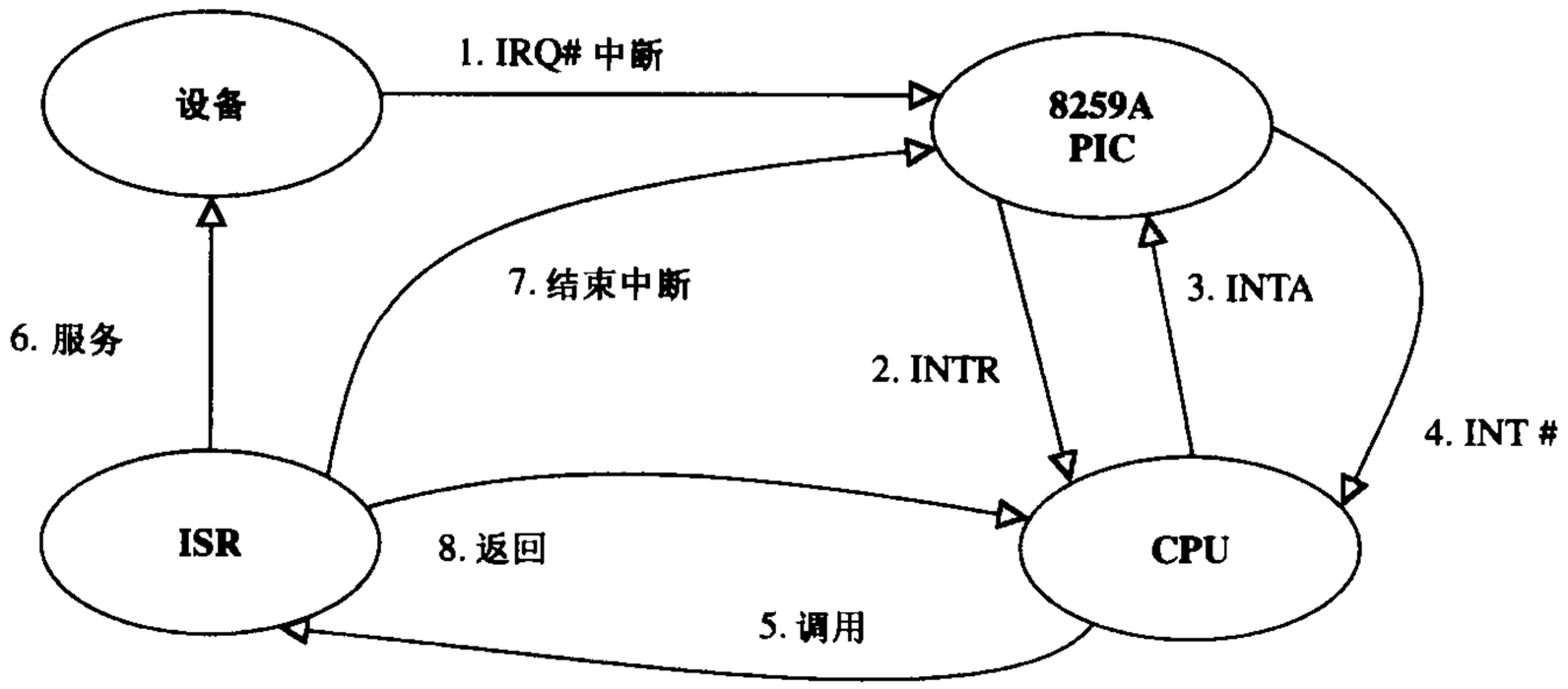


图 8.8 80x86 中的中断处理

在使用单个 PIC 的计算机设计(PC 和 XT)中,允许有 8 个不同的中断请求(IRQ0 ~ IRQ7)。表 8.1 列出了典型设备的标准中断类型编号。在设计 AT 时,增加了第二个 PIC,从而使中断输入的编号增加到 15。图 8.9 显示了两个串接的 PIC 连线。一个是主 PIC,另外一个是从 PIC。如图中所示,所有来自 PIC 的中断是通过主 PIC 的 IRQ1 输入的。一般而言,一个 PIC 可以连接 8 个不同的从 PIC。

表 8.1 标准 IBM-PC 典型设备的中断类型编号

设备	IRQ 编号	中断类型编号
可编程的间隔定时器	0	08H
键盘	1	09H
串接到第二个 PIC	2	保留
串行通信口(COM2)	3	0BH
串行通信口(COM1)	4	0CH
硬盘控制器	5	0DH
软盘控制器	6	0EH
并行打印机控制器	7	0FH

例 8.2 ARM 的中断体系结构。ARM 是 Advanced RISC Machine 的缩写。ARM 是 16/32 位体系结构,由于它的低功耗和相对较高的性能而主要用于便携式设备。称为 ATIC 的中断控制器收集并控制向 ARM 核心发出的中断请求。它提供了对核心的接口且能收集到 64 个中断请求。

通常中断事件的顺序如下:中断先在源(比如外设)处激活,然后在中断控制器处激活,最终在核心激活。当一个中断出现在源处时,它的信号将路由到中断控制器,然后到 ARM 核心。中断控制器能激活到核心的中断或使之失效,并为其分配一个优先级级别。一旦中断请求到达核心,它将使核心暂停正常的例程处理,以使中断请求能得到服务。

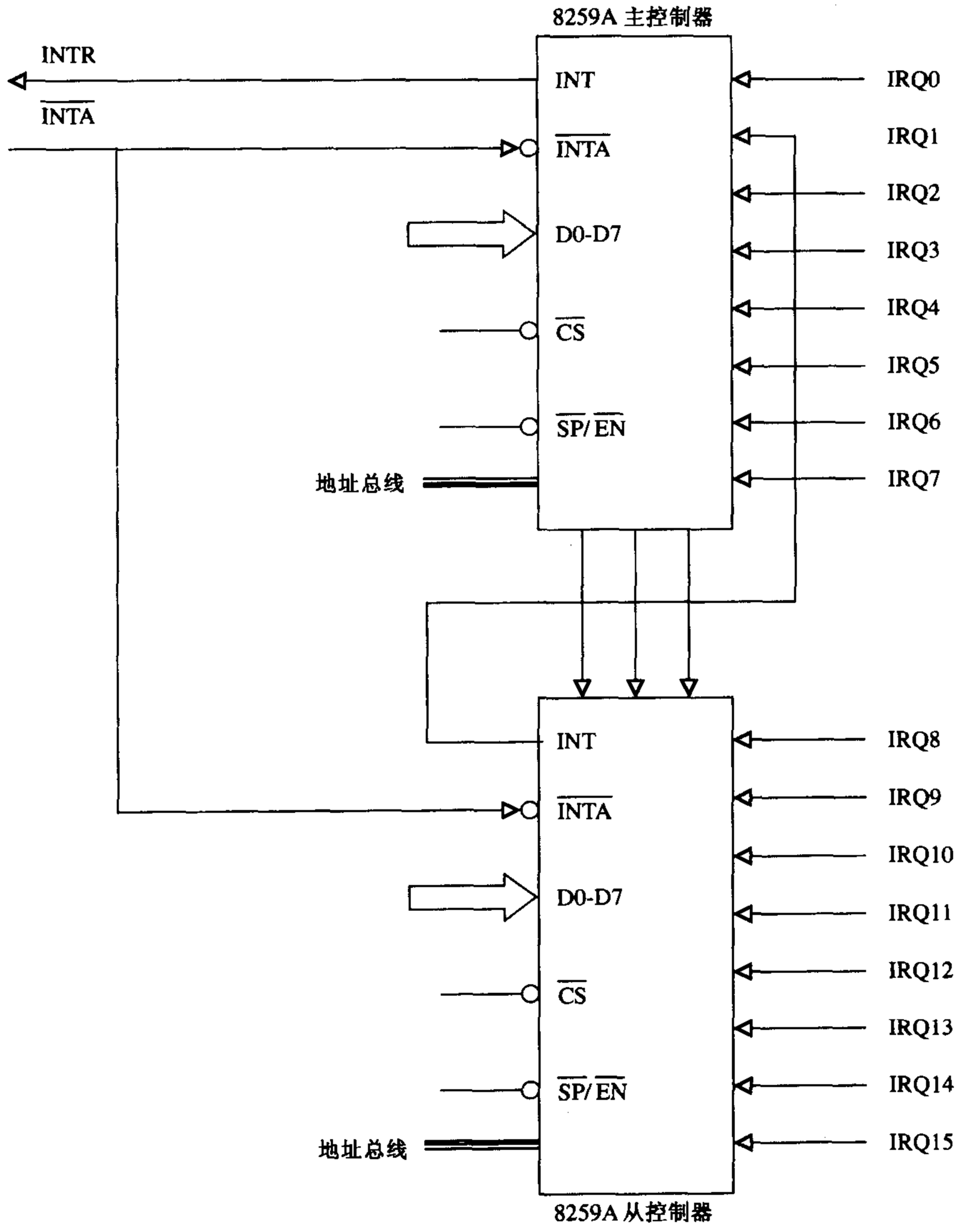


图 8.9　串接的两个 PIC 支持 15 个不同的中断

在不同中断请求中，ARM 核心能处理的是 IRQ 和 FIQ 请求。IRQ(正常中断请求)用于通用的中断处理。与 FIQ(快速中断请求)相比它的优先级较低而且在当 FIQ 序列进入时它将被屏蔽。FIQ 用来支持高速数据传输或是通道处理。

与 80x86 类似，中断处理程序的地址存放在一个向量表中，如表 8.2 所示。例如当一个 IRQ 被核心检测到时，它就访问向量表地址 0×18，然后执行装载在该地址中的指令。通常在向量表 0×18 中找到的指令形式如下：LDR PC，IRQ _ Handler(将 IRQ 中断处理程序的地址装载到程序计数器 PC 中)。当一个 FIQ 被核心检测到时，它访问向量表地址 0×1C，然后执行该地址中保存的指令。通常在向量表 0×1C 中找到的指令形式如下：LDR PC，FIQ _ Handler。

当中断发生时，在核心中发生以下这些动作：

1. 将 CPSR(当前程序状态寄存器)复制到正在进入的模式的 SPSR(被保存的程序状态寄存器);
2. 设置 CPSR 位与正在进入的模式相适应,设置核心为 ARM 状态,同时设置相关中断失效标志;
3. 保存分组寄存器的适当集合;
4. 存放返回地址到链接寄存器中(以相关的方式);
5. 将程序计数器 PC 设置为相关向量地址。

表 8.2 中断向口表

异常类型	模式	地址
复位	超级用户	0×00000000
未定义指令	未定义	0×00000004
软件中断(SWI)	超级用户	0×00000008
预取异常中止	异常中止	0×0000000C
数据异常中止	异常中止	0×00000010
IRQ(正常中断)	IRQ	0×00000018
FRQ(快速中断)	FIQ	0×0000001C

例如,当检测到 IRQ 中断时,ARM 核心就激活 SPSR_irq 作为 CPSR,通过设置 CPSR 的模式位为 10010 进入 IRQ 模式,通过设置 CPSR 的 I 位使普通中断失效,保存下一指令 R14_irq 的地址,将 0×18 装载到 PC 中。在地址 0×18,指令将载入中断处理程序的地址到 PC 中。类似地,当检测到 FIQ 中断,ARM 核心将激活 SPSR_fiq 作为 CPSR,通过设置 CPSR 的模式位为 10001 进入 FIQ 模式,通过设置 CPSR 的 I 位和 F 位使普通中断和快速中断失效,保存下一指令 R14_fiq 的地址,将 0×1C 装载到 PC 中。在地址 0×1C,指令将载入中断处理程序的地址到 PC 中。

MC9328MX1/MXL AITC。MC9328MX1/MXL AITC 包含了 26 个 32 位寄存器,如表 8.3 中所示。通过使用这些寄存器,AITC 允许一个等待的中断源对核心产生普通中断(IRQ)还是快速中断(FIQ)进行选择。这可以通过 INTTYPEH 和 INTTYPEL 寄存器实现。这些寄存器的每一位均对应于系统中可用的中断源。设置这一位将选择对应的中断源为快速中断,而清除这一位将选择对应的中断源为普通中断。在 INTTYPEL 寄存器中,位 0 对应于中断源 0,位 1 对应于中断源 1,这样一直到位 31 对应于中断源 31。在 INTTYPEH 寄存器中,位 0 对应于中断源 32,位 1 对应于中断源 33,这样一直到位 31 对应于中断源 63。

在决定了等待中断的类型之后,下一步是激活中断。这可以通过 INTENABLEH 和 INTENABLEL 寄存器实现。为了激活等待进入核心的中断,必须设置在 INTENABLEH 或 INTENABLEL 寄存器中对应中断源的一位。同样,使中断失效就清除该位。在 INTENABLEL 寄存器中,位 0 对应于中断源 0,位 1 对应于中断源 1,这样一直到位 31 对应于中断源 31。在 INTENABLEH 寄存器中,位 0 对应于中断源 32,位 1 对应于中断源 33,这样一直到位 31 对应于中断源 63。例如选择中断源位 15 为普通中断,首先清除 INTTYPEL 寄存器的位 15。然后为了激活该中断,设置 INTENABLEL 寄存器的位 15。同样,选择中断源 45 为快速中断,首先设置 INTTYPEH 寄存器的位 13。然后为了激活该中断,设置 INTENABLEH 寄存器的位 13。AITC 也允许程序员以 16 个不同优先级来划分等待中的普通中断源。这可以通过 NIPRIORITY[7:0]寄存器实现。

表 8.3　AITC 的寄存器

寄存器	描述
INTCNTL	设置 AITC 特定的控制函数
NIMASK	控制普通中断屏蔽级。所有普通中断优先级级位于或低于 NIMASK 寄存器中编程的级将会被屏蔽。普通中断优先级借助 NIPRIORITY[7:0]寄存器进行编程
INTENNUM	提供激活中断的硬件加速。这可以通过对寄存器和想要激活的中断源编程实现。这样做将马上激活(设置)INTENABLEH/L 寄存器的中断源位
INTDISNUM	提供使中断失效的硬件加速。这可以通过对寄存器和想要激活的中断源编程实现。这样做将马上清除 INTENABLEH/L 寄存器的中断源位
INTENABLEH	用来激活等待进入核心的中断源位[63 ~ 32]
INTENABLEL	用来激活等待进入核心的中断源位[31 ~ 0]
INTTYPEH	用来选择一个已经激活且在等待的中断源位[63 ~ 32]对核心产生普通中断或是快速中断
INTTYPEL	用来选择一个已经激活且在等待的中断源位[31 ~ 0]对核心产生普通中断或是快速中断
NIPRIORITY[7:0]	提供普通中断的软件优先级
NIVECSR	提供最高等待普通中断的优先级,同时提供最高等待普通中断的来源编码
FIVECSR	提供最高等待快速中断源号
INTSRCH	反映中断请求输入(来源 63 ~ 32)的状态到中断控制器
INTSCRL	反映中断请求输入(来源 31 ~ 0)的状态到中断控制器
INTFRCH	许可中断源为 63 ~ 32 的中断的软件生成
INTFRCL	许可中断源为 31 ~ 0 的中断的软件生成
NIPNDH	为中断源 63 ~ 32 反映等待中的普通中断请求源的编号
NIPNDL	为中断源 31 ~ 0 反映等待中的普通中断请求源的编号
FIPNDH	为中断源 63 ~ 32 反映等待中的快速中断请求源的编号
FIPNDL	为中断源 31 ~ 0 反映等待中的快速中断请求源的编号

8.4　直接存储器访问

直接存储器访问(DMA)的主要思想是使外部设备在数据传输中取代 CPU 的“中间人”角色。它允许外部设备直接从/向存储器传输数据而不需要 CPU 的干预。让外部设备直接访问存储器将允许 CPU 做其他的工作,这就提高了系统的性能,特别是在海量传输的情况下。

DMA 控制器是一块控制一个或多个外部设备的硬件。它允许设备在没有处理器帮助的情况下从/向系统存储器传输数据。在典型的 DMA 传输中,由某个事件通知 DMA 控制器,数据需要从存储器传输或是传输到存储器。DMA 和 CPU 共用存储器总线,因此在同一时刻两者中只有一个能使用存储器。首先 DMA 控制器向 CPU 发送请求以获得使用存储器总线的许可。CPU 返回确认给 DMA 控制器,授权它访问总线。此时 DMA 获得了对总线的控制权,可以独立地进行存储器传输。当传输结束时,DMA 将总线的控制权交还给 CPU。支持 DMA 的处理器提供一个或多个输入信号以确保总线请求者获得对总线的控制,同时也提供一个或多个输出信号以确保 CPU 已经放弃了对总线的控制。如图 8.10 所示为 DMA 控制器如何共享 CPU 的存储器总线。

DMA 控制器需要由 CPU 初始化。典型的参数设置包括源区域的地址、目标区域的地址、块的长度和一旦块传输结束 DMA 控制器是否会生成一个处理器中断。DMA 控制器含有地址寄存

器、字计数寄存器和控制寄存器。地址寄存器包括了指定所需传送数据所在存储器单元的地址。通常使 DMA 控制器在每个字传输后自动增加地址寄存器的值,这样下次传输将从下一个存储器单元开始。字计数寄存器保存了所需传送字的数量,在每个字传输后字计数将减 1。而控制寄存器则用来指定传输模式。

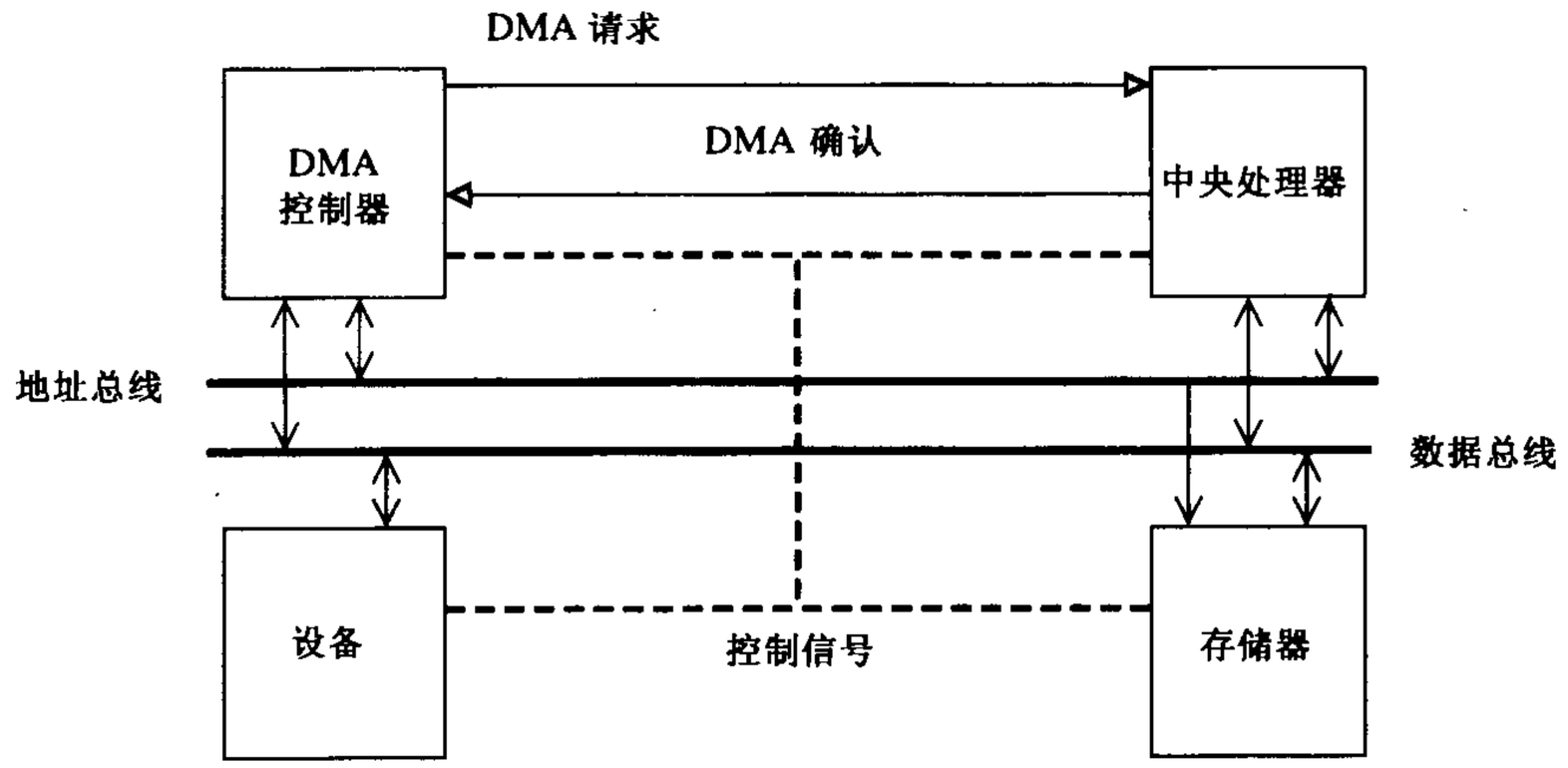


图 8.10 DMA 控制器共享 CPU 的存储器总线

DMA 的数据传输能以突发模式或单周期模式执行。在突发模式中,DMA 控制器保持对总线的控制,直到所有的数据从/向存储器传送到/从外部设备。快速设备需要这种传输模式,这样数据能不间断地传送直到整个传输完成。在单周期模式(周期窃取)中,DMA 控制器在每次传输完一个数据字之后放弃总线。这就可以使 DMA 控制器阻止 CPU 控制总线的总时间最小化,不过它需要总线每个单周期都执行请求/应答序列。这个开销降低了性能。单周期模式适用于下列情况:若系统不能容忍超过几个周期的额外中断延迟,或是若外部设备能缓冲很大数量的数据,从而导致 DMA 控制器在过多的时间内一直占用总线。

下列步骤总结了 DMA 的操作:

1. DMA 控制器启动数据传输;
2. 移动数据(增加存储器中的地址,减少所需移动字的计数值);
3. 当字计数为 0 时,DMA 以中断的方式通知 CPU 移动数据过程已结束;
4. CPU 重新获得对存储器总线的访问权。

DMA 控制器可以有多个通道。相应于每个通道有一个地址寄存器和一个计数寄存器。为了启动数据传输,设备驱动程序需要设置 DMA 通道的地址、计数寄存器和数据传输的方向,读或是写。在传输进行时,CPU 可以做其他事情。当传输结束时,CPU 被中断。

各个设备驱动程序不能共享 DMA 通道,一个设备驱动程序必须能决定使用哪个 DMA 通道。某些设备有固定的 DMA 通道,而另外一些则比较灵活,它的设备驱动程序能简单地从空闲 DMA 通道中选择一个使用。

Linux 利用向量 dma_chan 数据结构来追踪 DMA 通道的使用情况(每个 DMA 对应一个向量)。dma_chan 数据结构只有两个字段,一个是指针,由它指向描述 DMA 通道所有者的字符串,另一个是标志位,用来表示 DMA 通道是否已经被分配。

8.5　总线

总线在计算机术语中表示一个物理连接,用来将信号从一点传送到另一点。总线传送的信号可能表示地址、数据、控制信号或电源。一般而言,总线由一些运行在一起的连接器(connection)组成。每个连接器称为总线线路(bus line)。总线线路通常用编号识别,而相关的总线线路组通常用名字识别。比如在一个给定的计算机系统中,总线线路为 1 到 16 的组可能用来传送存储器单元的地址,因此称为地址线。至少存在 4 种基于信号传送的总线:地址、数据、控制和电源总线。地址总线传送地址数据,数据总线传送数据,控制总线传送控制信号,电源总线传送电源/接地电压。地址、数据和控制总线的大小(线数)在不同的系统中是不同的。例如在一个给定系统中,连接 CPU 和存储器的总线称为 CPU 总线。这个系统存储器的大小是 512 兆字,每个字是 32 位。在这样的系统中,地址总线的大小应该是 $\mathrm{lb}(512 \times 2^{20}) = 29$ 条线,数据总线的大小应该是 32 条线,此外至少有一条控制线($\overline{R}/W$)。

除了传送控制信号之外,控制总线也能传送定时信号。这些信号可以用来确定数据从总线传出或是数据传送到总线的确切时间。也就是它们决定什么时候一个给定计算机的系统组件,比如处理器、存储器或 I/O 设备,能够释放数据到总线,什么时候能够从总线接受到数据。总线可以是同步的——若总线上数据的传输由总线时钟控制。时钟为所有总线信号提供了定时基准。总线也可以是异步的——若总线上数据的传输是基于数据的可用性而不是时钟信号。数据在异步总线上传输所使用的技术称为握手。同步总线和异步总线的操作将在下面解释。

为了清楚地了解同步和异步之间的区别,我们考虑当主设备(比如 CPU 或是 DMA)是所需传送数据的来源时,将数据传送到从设备,比如 I/O 设备的情况。下面是涉及主从设备的事件序列。

1. 主设备:发送使用总线的请求;
2. 主设备:请求被认可,总线被分配给主设备;
3. 主设备:将地址/数据放到总线上;
4. 从设备:选择从设备;
5. 主设备:信号数据传送;
6. 从设备:取数据;
7. 主设备:释放总线。

8.5.1　同步总线

在同步总线中,数据传输的步骤发生在固定时钟周期。每件事情都与总线时钟同步,而主设备和从设备都可获得时钟信号。总线时钟是一个方波信号。一个周期从时钟的一个上升边缘开始,到下一个上升边缘结束,同时也是下一个周期的开始。传送可能需要多个总线周期,这依赖于总线的速度参数和传送的两端。

某个特定情况可能是在第一个时钟周期,主设备将地址放到地址总线上,将数据放到数据总线上,并激活相应的控制线。同时从设备在地址总线上识别出它的地址,在第二个周期从总线读取新的值。

同步总线简单而且容易实现。但当将多个不同速度的设备连接到同步总线时,最慢的设备将决定总线的速度。此外,为了避免时钟偏差的问题,同步总线的长度也有限制。

8.5.2 异步总线

异步总线中没有固定的时钟周期,而是使用握手。如图 8.11 所示为握手协议。主设备在见到一个数据接收信号时就激活数据就绪线(见图中点 1 处)。当从设备看到数据就绪的信号时,它就激活数据接收线(见图中点 2 处)。数据接收线的上升导致数据就绪线的下降,同时将数据从总线中移出。数据就绪线的下降(见图中点 3 处)导致数据接收线的下降(见图中点 4 处)。这种握手(称为完全互锁)将一直重复直到数据传送完成。异步总线适合不同速度的设备。

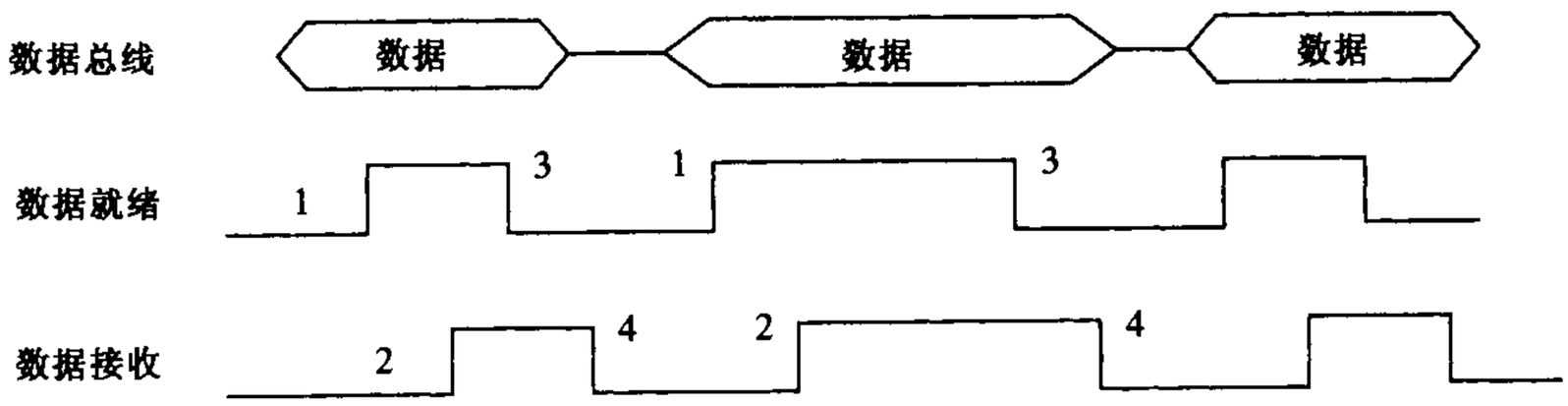


图 8.11 使用握手协议的异步总线的定时

8.5.3 总线仲裁

在当两个或更多设备同时希望成为总线主设备时需要总线仲裁。简短地说,仲裁是在多个候选者中挑选下一个总线主设备的过程。冲突能够在集中式或分布式机制中基于公平性或优先级解决。

集中式仲裁。在集中式仲裁方案中,使用单个仲裁者挑选下一个主设备。一种简单形式的集中式仲裁使用了总线请求线、总线许可线和总线忙碌线。这些线被以菊花链式串接的候选主设备所共享。如图 8.12 所示为这种简单的集中式仲裁方案。

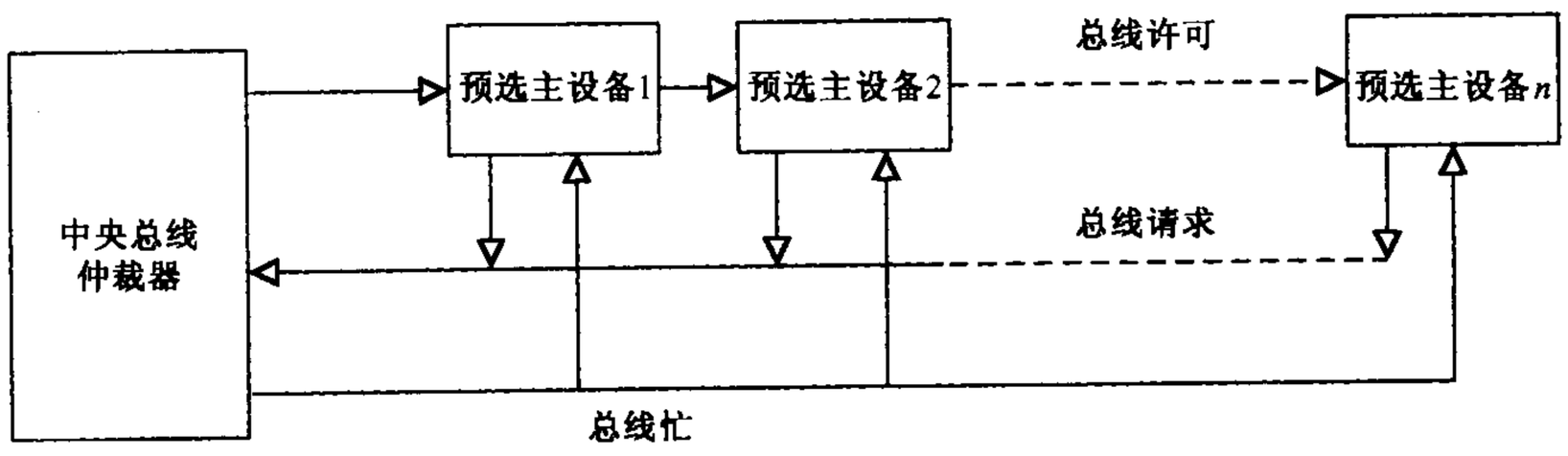


图 8.12 菊花链方案的集中式仲裁器

在图中,每个候选主设备可以在任何时刻提交总线请求。这里以从左向右的方式在主设备间设置了固定优先级。当中央的总线仲裁器收到一个总线请求时,它通过激活总线许可线发出总线许可。当最靠近仲裁器的候选主设备(候选主设备 1 号)看到总线许可信号时,它将查看一下它是否发送过总线请求。若发送过,它占用总线并停止总线许可信号的进一步传送。若它没有发送过请求,它将简单地转发总线许可信号给右边的下一个主设备(候选主设备 2 号),如此继续。当事务处理完成时,就取消忙碌线的激活状态。

除了使用共享的请求和许可线,也可以使用多条总线请求线和许可线。如图 8.13 所示的一种方案中,每个主设备将拥有自己独立的请求和许可线。中央仲裁器能使用任何基于优先级或是基于公平性的仲裁方案。另一种方案允许主设备有多个优先级级别。每个优先级级别有各

自的总线请求和许可线。在每个优先级级别中使用了菊花链。在这种方案中,每个设备均附到某个优先级级别的菊花链上。若仲裁者收到多个不同优先级级别的总线请求,它将许可总线授予最高优先级级别。菊花链用在该级别的设备之间。如图 8.14 所示为包括 2 个优先级级别的 4 个设备的例子。候选主设备 1 号和 3 号在级别 1 的菊花链中,而候选主设备 2 号和 4 号则在级别 2 的菊花链中。

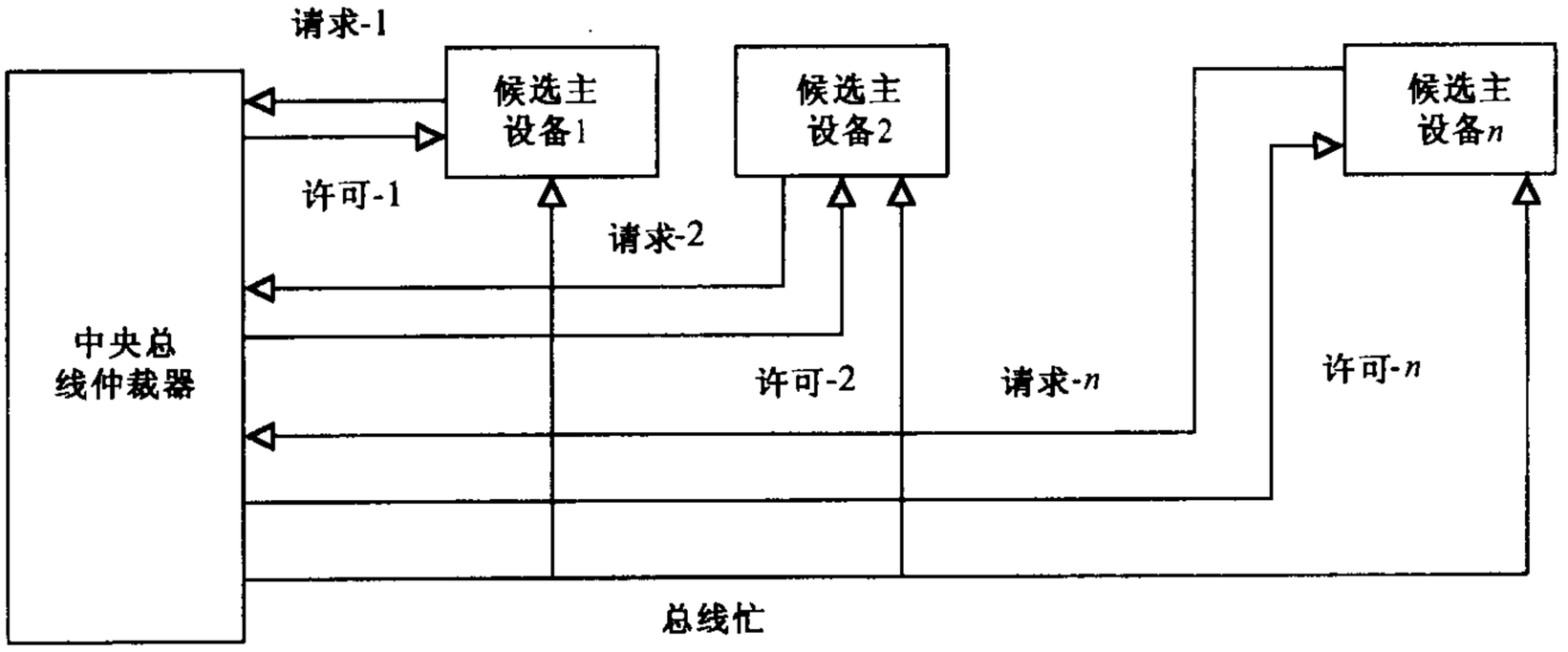


图 8.13　有独立请求和许可线的集中式仲裁器

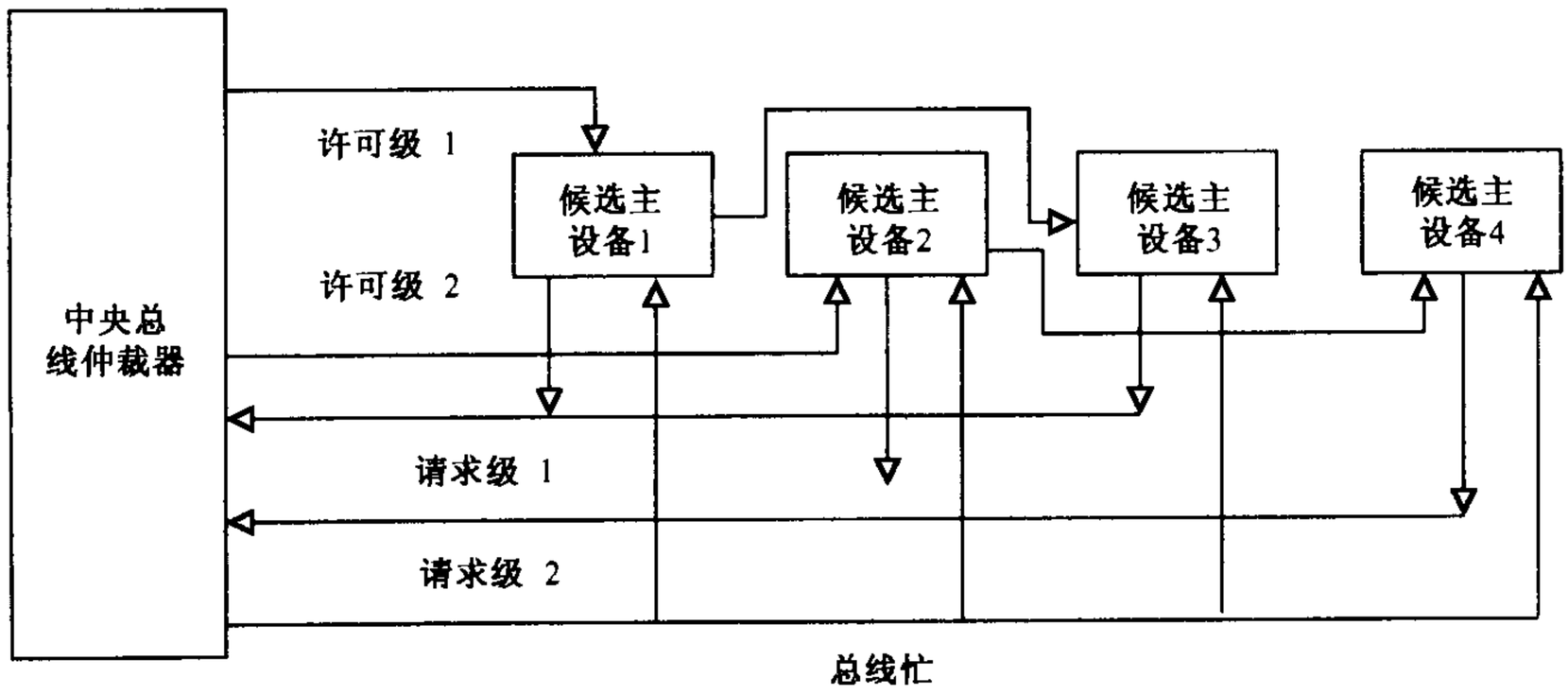


图 8.14　有 2 个优先级级别(4 设备)的集中式仲裁器

分布式仲裁。在分布式仲裁方案中,基于优先级的仲裁通常被用在分布式方式中。每个候选主设备有惟一的仲裁号,它用来解决多个请求提交时的冲突。例如冲突总能以有利于最高仲裁号的设备的方式解决。现在的问题是如何确定哪个设备拥有最高仲裁号?一个方法是请求设备将它的惟一仲裁号提供给所有其他设备。每个设备将这个号与自己的仲裁号对比。数字较小的设备被去除。最终具有最高仲裁号的请求者被保留下来,并被许可访问总线。

8.6　I/O 接口

在计算机系统中接口是两个分离设备间的数据通路。总线的接口能够基于在给定时间内所传送的位数而分类成串行接口或并行接口。在串行接口中,一次只能传送一位数据。鼠标和调制解调器通常连接到串行接口。并行接口允许一次传送多于 1 位的数据。打印机是连接到并行接口的最常用的外部设备。表 8.4 汇总了个人计算机中所使用的各种不同的总线和接口。

表 8.4 个人计算机中所使用的总线和接口的说明

总线/接口	说明
PS/2	一种可以用来连接鼠标和键盘到计算机的端口(接口)类型,PS/2 口有时又称为鼠标口
工业标准体系结构(ISA)	ISA 最初是 8 位总线,后来在 1984 年扩展成 16 位总线。1993 年,Intel 和微软提出了一种即插即用的 ISA 总线,它允许计算机自动检测和设置计算机的 ISA 外部设备,比如调制解调器和声卡
扩展工业标准体系结构(EISA)	EISA 是 ISA 的增强形式,它支持 32 位数据传输,同时又保持对 8 位和 16 位扩展板的支持。但是它的总线速度和 ISA 一样只有 8 MHz。由于昂贵的价格和复杂的设计,EISA 并没有被广泛使用
微通道体系结构(MCA)	MCA 在 1987 年由 IBM 提出。它在 ISA 的基础上提供一些特性,比如 32 位总线,可以支持更高效率的自动设置卡和总线。它与 EISA 相比稍占优势,但是很少有扩展板能适应 MCA 的规范
VESA(视频电子标准协会)本地总线(VLB)	VESA 是 NEC 创建的一个非赢利组织,它在 1992 年发布了 VLB 标准。它是 32 位总线,能以通常的 486(33/40 MHz)CPU 的速度直接访问系统存储器。VLB 2.0 稍后在 1994 年发布,有一个 64 位总线,总线速度是 50 MHz
外部设备互连总线(PCI)	PCI 是 Intel 在 1992 年提出的,在 1993 年修订成 2.0 版本,后来又在 1995 年修订成 PCI 2.1。它是 32 位总线,现在也有了 64 位总线。许多现代扩展板都可以连接到 PCI 槽
图形加速端口(AGP)	AGP 是 Intel 在 1997 年提出的。AGP 是针对 3D 图形的高要求所设计的 32 位总线。AGP 有一条到存储器的直接连线,它允许 3D 元素存储在系统存储器中,而不是在视频存储器中。AGP 的设计是为了适应数据密集的图形卡需要,比如 3D 加速卡。它的设计允许数据以每秒 266 MB 的速度进行吞吐
通用串行总线(USB)	USB 是由 Intel、康柏、DEC、IBM、微软、NEC 和北方电信联合开发的外部总线。1996 年与 Intel 430HX Triton II 主板一起发布。USB 每秒能传输 12 Mb 数据,支持多达 127 种设备。许多设备能够连接到 USB 端口,它支持即插即用
火线(FireWire,IEEE 1394)	火线是外部总线的一种,能支持非常高的传输速率:每秒 400 Mb。因此火线适合连接视频设备(比如 VCR)到计算机
小型计算机系统接口(SCSI)	SCSI 是并行接口一种,通常用于海量存储器设备。SCSI 能够以每秒 4 MB 的速率传送数据。另外,有些 SCSI 变种能支持更快的速度:如 Fast SCSI(每秒 10 MB)、Ultra SCSI 和 Fast Wide SCSI(每秒 20 MB)、以及 Ultra Wide SCSI(每秒 40 MB)
集成驱动器电路(IDE)	IDE 是常用的硬盘设备驱动器和 CD-ROM 驱动器的接口。它比 SCSI 便宜,但提供的性能稍差
增强型集成驱动器电路(EIDE)	EIDE 是 IDE 的改进版本,提供了比标准 SCSI 更好的性能。它提供的传输速率在每秒 4 MB 到 16.6 MB 之间
扩展外部设备互连总线(PCI-X)	PCI-X 是一种高性能总线,它是设计来满足日益增长的 I/O 的技术需求,比如光纤通道、千兆以太网和 Ultra3 SCSI
通信和网络插卡(CNR)	CNR 是 Intel 在 2000 年提出的。它是支持音频、USB 和局域网络接口的核心逻辑芯片集的规范

8.7 小结

计算机系统的一个主要特性是它能与其他设备交换数据,并允许用户与系统交互。本章主要关注 I/O 系统和在计算机系统中处理器与 I/O 设备交换数据的方法。本章描述了三种 I/O 的

组织方式:编程 I/O、中断驱动 I/O 和直接存储器访问。在编程 I/O 中,CPU 负责处理寄存器和设备间的数据传输。在中断驱动 I/O 中,CPU 处理数据传输时,I/O 模块也在并发运行。在直接存储器访问中,数据在没有 CPU 干预的情况下,在存储器与 I/O 设备间传输。我们也研究了两种同步方法:轮询和中断。在轮询中,处理器在等待 I/O 结束的过程中轮询设备。显然处理器周期在这种方式中被浪费了。使用中断,处理器可以在 I/O 过程中自由地切换到其他任务。当 I/O 结束时,设备将激活中断。中断会产生一些延时开销。我们还介绍了两个中断处理的实例:80x86 系列和 ARM。本章最后介绍了总线和接口。并对个人计算机中广泛使用的一些总线和接口进行了总结。

习题

1. 在因特网上搜索关于 I/O 设备的资料,准备一张表格将不同设备分为不同的类别。例如输入、输出、基于字符的、基于块的,等等。为表格中每个条目注明速度、接口和类别。
2. 列举共享 I/O 和存储器映射 I/O 各自的优缺点。
3. 说明在以下三种 I/O 方案中数据如何从磁盘传输到存储器:编程 I/O,中断驱动 I/O,直接存储器访问。说出各方案中所需执行的步骤。
4. 若一次中断需要 50 ms 的开销,而轮询一个设备则需要 5 ms 的开销,描述这两种方式各自占优势的情况。
5. 在计算机系统中设备驱动程序需要与哪些实体通信?设备驱动程序的功能是什么?列举所有的操作。
6. 哪些类型的操作是用来加速 DMA 的?
7. DMA 模块使用周期窃取技术从传输速率为每秒 19 200 位的一个设备向存储器传输数据。CPU 的速度是 3 MIPS。问采用 DMA 传输对 CPU 性能的影响有多大。
8. 描述同步总线优/劣于异步总线的情形。
9. 讨论本章中不同总线仲裁策略的优点和缺点。准备一张对照表,从实现和操作两方面比较各种仲裁技术。

参考文献和推荐读物

I. Englander, *The Architecture of Computer Hardware and System Software*, John Wiley, New York, 1996.

C. Hamacher, Z. Vranesic, and S. Zaky, *Computer Organization*, 5th ed., McGraw-Hill, New York, 2002.

V. Heuring, and H. Jordan, *Computer Systems Design and Architecture*, Addison Wesley, NJ, USA, 1997.

S. Shiva, *Computer Design and Architecture*, Harper Collins, MA, USA, 1991.

A. Tanenbaum, *Structured Computer Organization*, 4th ed., Prentice Hall, NJ, USA, 1999.

J. Uffenbeck, *The 80 × 86 Family, Design, Programming, and Interfacing*, 3rd ed., Prentice Hall, Essex, UK, 2002.

参考网站

PCI Local Bus Specifications, www.pcisig.com/developers

SCSI-3 Architecture Model(SAM), www.ansi.org

Universal Serial Bus Specification, www.usb.org/developers

www.arm.com

www.motorola.com/semiconductors

第9章　流水线设计技术

有两种基本技术可以提高处理器指令的执行速率。一种是增加时钟速率,从而减少指令执行的时间。另一种是增加同时执行的指令数量。流水线和指令级并行是后一种技术的例子。流水线的概念源于汽车装配线,其理念是处理器同时执行多条指令。与装配线类似,流水线的成功依赖于将指令执行划分成在几个子部件(级)间进行,每级执行所需操作的一部分。一种可能的划分是将指令的读取(F)、指令的译码(D)、操作数的读取(F)、指令的执行(E)和结果的保存(S)视为执行指令所必需的子任务。在这种情况下,有可能最多将五条指令同时放到流水线中,从而减少指令执行的等待时间。在本章中,我们将讨论设计指令流水线所涉及的基本概念,介绍度量流水线性能的方法,讨论导致指令流水线冒险的主要问题,还介绍一些可能的解决办法。同时我们还将介绍算术流水线的概念和设计这种流水线所涉及的问题。最后我们将讨论最近出现的一些流水线处理器。

9.1　一般概念

流水线是指将一个给定任务划分成一些需顺序执行的子任务所使用的技术。每个子任务由一个给定的功能单元执行。这些单元以串行的方式连接,而且所有的单元同时操作。与传统的顺序执行任务相比,使用流水线提高了性能。如图 9.1 所示为使用流水线和顺序处理来执行一条给定指令的 4 个子任务(在这里是读取 F,译码 D,执行 E 和写结果 W)之间的基本区别。

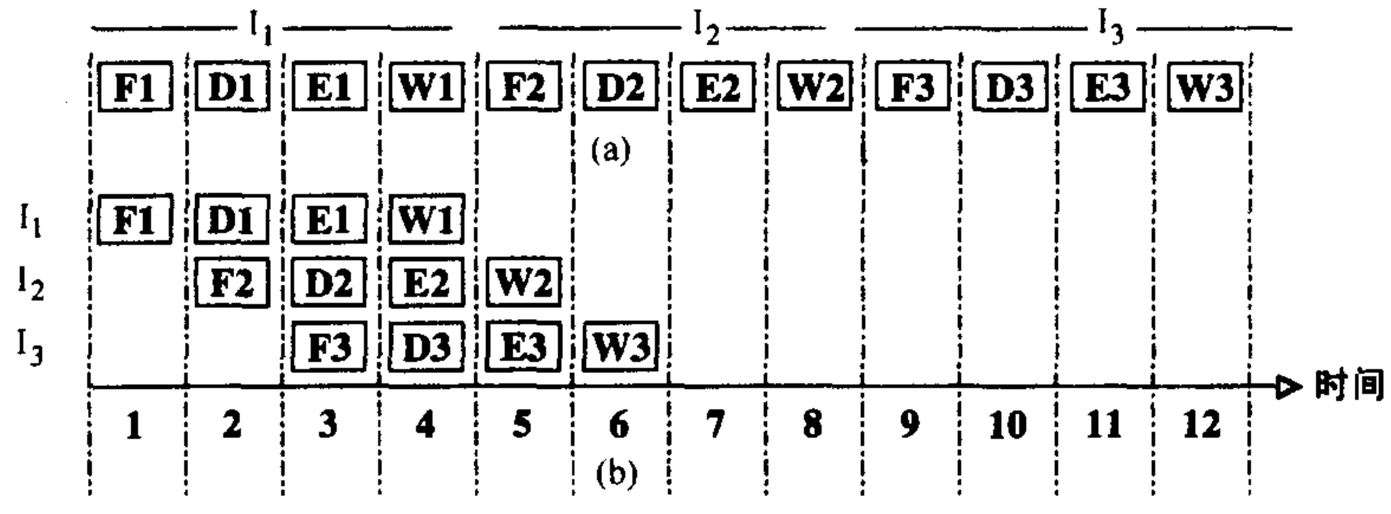


图 9.1　流水线与顺序处理:(a)顺序处理;(b)流水线

从图 9.1 中可以清楚地看出,若使用 4 级流水线处理 3 条指令(I_1,I_2,I_3)总共需时为 6 个时间单位,而若使用顺序处理则需 12 个时间单位。使用流水线执行这 3 条指令可以节省将近 50% 的执行时间。为了简洁地描述流水线在处理串行任务时的一些性能度量,我们使用了时空图(也称为 Gantt 图)。该图能显示流水线中子任务的连续性和对应的时间。图 9.2 就是 Gantt 图。在该图中,纵轴表示子部件(这里是 4 个),横轴表示时间(根据每个部件执行任务所需的时间单位来度量)。在绘制 Gantt 图时,我们假设每个子部件执行任务的时间(T)是一样的,我们称之为单位时间。

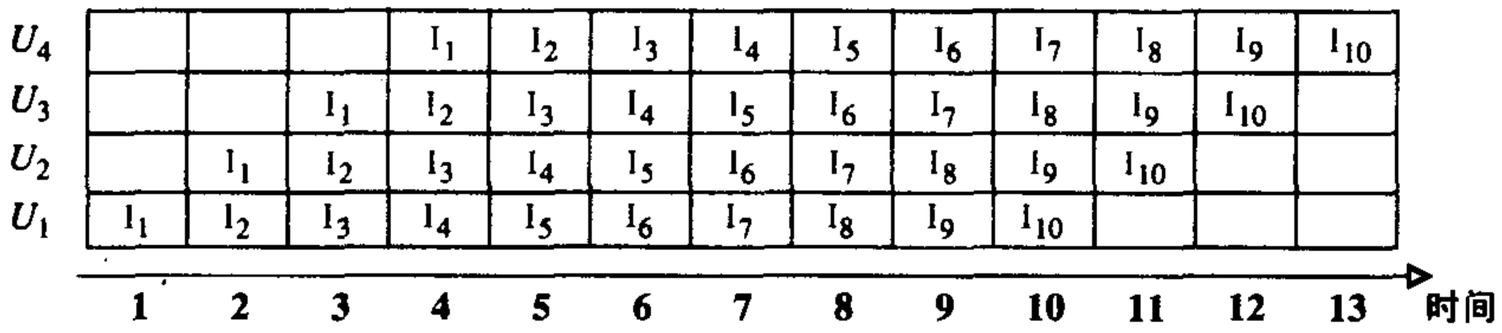


图 9.2 时空图(Gantt 图)

从图 9.2 中可以看出,完成 10 条指令(I_1 到 I_{10})的执行需要 13 个时间单位。与之相比,若使用顺序处理则需要 40 个时间单位(10 条指令每条需要 4 个时间单位)。

在下面的分析中,我们提供了 3 种度量流水线性能优势的方式。分别是加速 $S(n)$、吞吐量 $U(n)$和效率 $E(n)$。需要注意的是在这个分析中我们假设单位时间 $T = t$ 单位。

1. **加速 $S(n)$**。考虑使用 n 级(单元)流水线执行 m 个任务(指令)。可以看到,完成 m 个任务需要 $n + m - 1$ 个时间单位。

$$\text{加速 } S(n) = \frac{\text{使用串行处理的时间}}{\text{使用流水线处理的时间}} = \frac{m \times n \times t}{(n + m - 1) \times t}$$

$$= \frac{m \times n}{n + m - 1}$$

$$\lim_{m \to \infty} S(n) = n \text{(也就是说,速度增长 } n \text{ 倍在理论上是可能的)}$$

2. **吞吐量 $U(n)$**。

$$\text{吞吐量 } U(n) = \text{每个单位时间执行的任务数量} = \frac{m}{(n + m - 1) \times t}$$

$$\lim_{m \to \infty} U(n) = 1 \quad \text{假设 } t = 1 \text{ 个单位时间}$$

3. **效率 $E(n)$**。

$$\text{效率 } E(n) = \text{实际加速与最大加速的比率} = \frac{\text{加速}}{n} = \frac{m}{n + m - 1}$$

$$\lim_{m \to \infty} E(n) = 1$$

9.2 指令流水线

9.1 节中的简单分析忽略了一个能影响流水线性能的重要方面,即流水线停顿。若一个部件(阶段)需要更多的时间来执行它的功能,从而强制其他级空闲,这样的流水线操作称为停顿。例如考虑读取指令时遇到高速缓存缺失的情况。同时假设高速缓存缺失需要 3 个额外的时间单位。如图 9.3 所示为指令 I_2 遇到高速缓存缺失的情形(假设执行 10 条指令 I_1 到 I_{10})。

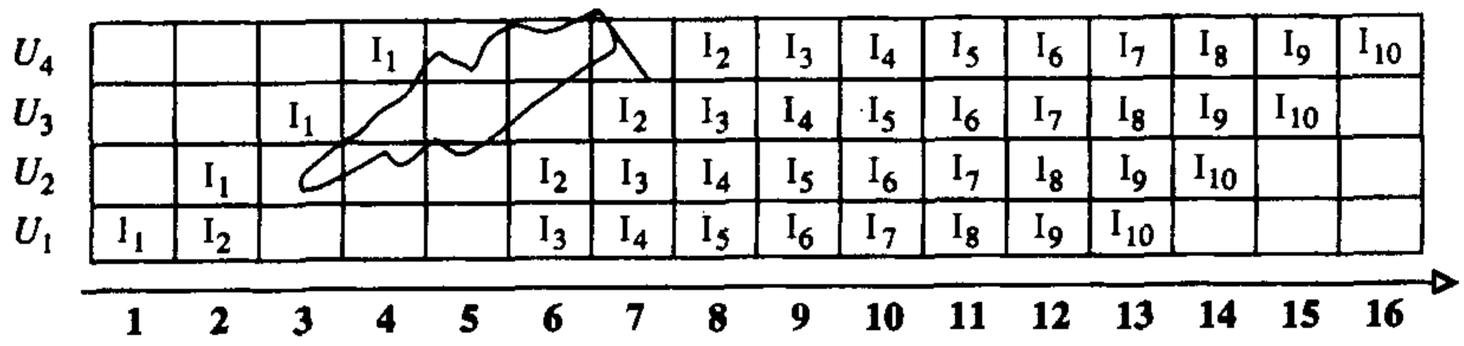


图 9.3 流水线中高速缓存缺失的影响

图 9.3 中显示由于读取指令 I_2 所需的额外时间单位，流水线停顿了，也就是延迟了读取指令 I_3 和它以后的指令。这种情况产生了所谓的流水线气泡(或流水线冒险)。流水线气泡的产生导致单位时间的浪费，从而导致完成执行给定数量指令所需的时间单位在总体上增加。如图 9.3 中所示，执行 10 条指令所需的时间单位数现在是 16，而若没有高速缓存缺失则是 13 个时间单位。

流水线冒险会因为一些其他原因产生，其中主要是指令相关和数据相关，将在下文解释。

9.2.1　指令相关导致的流水线“停顿”

流水线的正确操作需要一级执行的操作绝对不能依赖于另一级执行的操作。指令相关是指一条指令的读取依赖于前一条指令执行结果的情况。指令相关在执行条件转移指令时出现。例如考虑“若(条件)为负，转移”指令。在这里，直到知道“若(条件)为负，转移”指令的执行结果，所需读取的下一个指令才会被知道。在下面的讨论中，我们假设条件转移指令后的指令直到知道(存储)执行转移指令的结果才会被读取。下面的例子说明指令相关对流水线的影响。

例 9.1　考虑在一个由 IF(指令读取)、ID(指令译码)、IE(指令执行)和 IS(指令结果存储)组成的 4 级流水线上，执行 10 条指令 I_1 到 I_{10}。假设指令 I_4 是条件转移指令，当它执行时，转移没有执行，也就是没有满足转移条件。同时假设当读取转移指令时，流水线停顿，直到知道转移指令的执行结果。要求显示流水线中指令的连续性，也就是显示 Gantt 图。如图 9.4 所示为所需的 Gantt 图。图中清楚地显示了由于流水线停顿而产生的气泡。

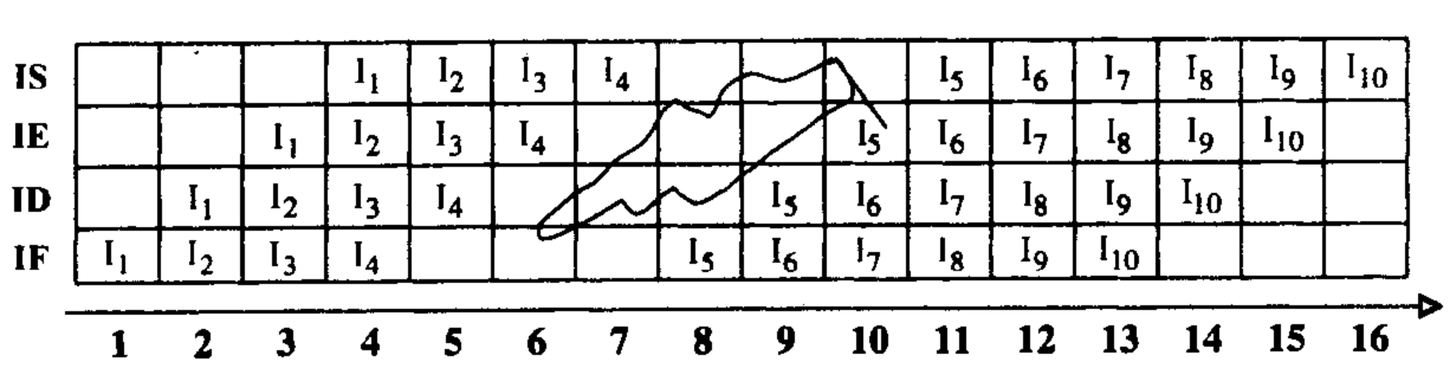


图 9.4　指令相关对流水线的影响

9.2.2　数据相关导致的流水线“停顿”

当指令 I_i 的一个源操作数依赖于先前指令 $I_j(i > j)$的执行结果时，在流水线中将产生数据相关。必须注意虽然可以读取指令 I_i，但它的操作数在 I_j 的结果被存储前将一直不可用。下面的例子显示了数据相关对流水线的影响。

例 9.2　考虑执行下面这段代码：

```
•
ADD    R1, R2, R3;     R3 ← R1 + R2
SL     R3;             R3 ← SL(R3)
SUB    R5, R6, R4;     R4 ← R5 − R6
•
```

在这段代码中，第一条指令称为 I_i，它将寄存器 R_1 和 R_2 的内容相加，并存储结果到寄存器 R_3。第二条指令称为 I_{i+1}，它将 R_3 的内容左移一位，并存储结果回 R_3。第三条指令称为 I_{i+2}，它将寄存器 R_5 和 R_6 内容相减，并存储结果到寄存器 R_4。为了显示这种数据相关的影响，我们假设

流水线由 5 级组成:IF,ID,OF,IE 和 IS。在这里,OF 阶段表示操作数读入阶段。剩下 4 级的功能和前面解释过的相同。图 9.5 显示了这段代码的 Gantt 图。如图 9.5 所示,虽然指令 I_{i+1} 在时间单位 $k+2$ 时被成功译码,但这条指令在时间单位 $k+3$ 内不能处理到 OF 级。这是因为在时间单位 $k+3$ 时 I_{i+1} 读取的操作数应该是寄存器 R_3 的内容,但它刚被指令 I_i 修改。R_3 修改后的值在时间单位 $k+4$ 结束前将一直不可用。这将需要指令 I_{i+1}(在 ID 部件的输出处)一直等到时间单位 $k+5$。注意指令 I_{i+2} 也必须(在 IF 部件的输出处)一直等待指令 I_{i+1} 处理 ID。最终结果是由于指令 I_i 和 I_{i+1} 之间的数据相关,流水线发生停顿。

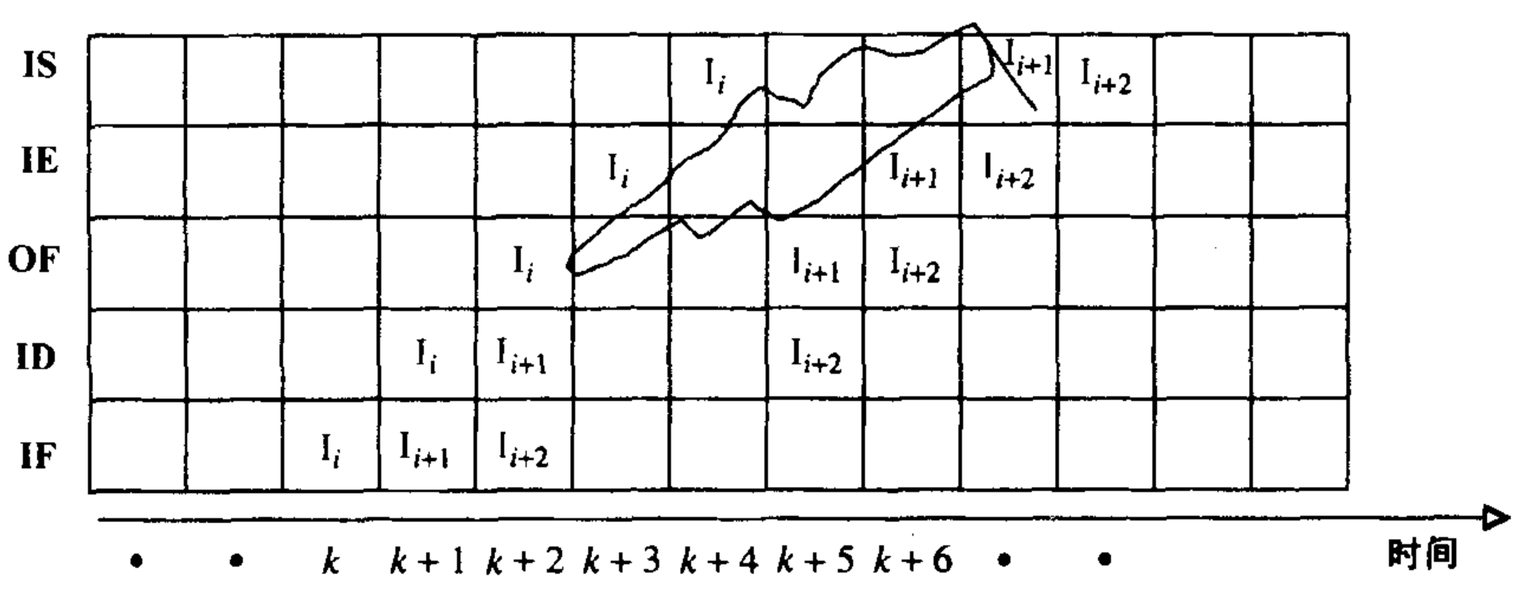


图 9.5 先写后写的数据相关

上面例子中表现的数据相关是由于寄存器 R_3 同时是指令 I_i 和 I_{i+1} 的目的地址。这称为先写后写数据相关。考虑任何寄存器都能够被写入(或读出),这样总共存在 4 种不同的可能性,包括先写后写情况。另外 3 种情况分别是先写后读、先读后写和先读后读。在这 4 种情况中,先读后读不会导致流水线停顿。这是因为寄存器的读操作并不改变寄存器的内容。在剩下 3 种情况中,先写后写(见前面的例子)和先写后读会导致流水线停顿。下面的这段代码显示了先写后读的情况:

```
o
ADD     R1, R2, R3;     R3 ← R1 + R2
SUB     R3, 1, R4;      R4 ← R3 − 1
o
```

在这里,第一条指令修改了寄存器 R_3 的内容(通过写操作),而第二条指令使用寄存器 R_3 修改后的内容(通过读操作)来载入一个值到寄存器 R_4。当这两条指令在流水线中处理时,必须要小心第二条指令读入寄存器 R_3 的值是执行前一条指令后更新的值。如图 9.6 所示为这种情况的 Gantt 图,假设第一条指令为 I_i,第二条指令为 I_{i+1}。

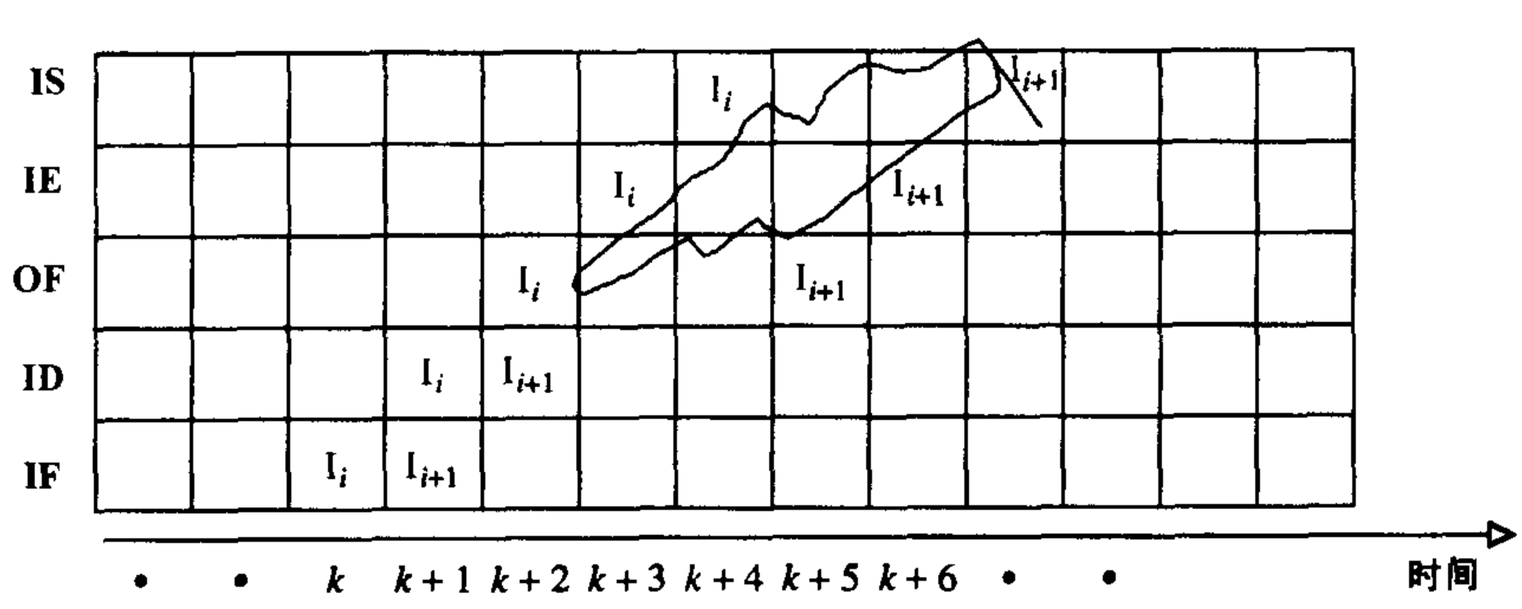


图 9.6 先写后读的数据相关

显然,第二条指令的操作数不能在时间单位 $k+3$ 时读取,它必须延迟到时间单位 $k+5$。这是因为寄存器 R_3 修改后的内容在时间单位 $k+5$ 前一直不可用。在时间单位 $k+3$ 时读取第二条指令的操作数将导致不正确的结果。

例 9.3　考虑在一个由 IF,ID,OF,IE 和 IS 所组成的 5 级流水线上,执行下列指令序列。说明流水线中的连续性:

$I_1 \rightarrow$ Load　$-1, R_1;$　　$R_1 \leftarrow -1;$

$I_2 \rightarrow$ Load　$5, R_2;$　　$R_2 \leftarrow 5;$

$I_3 \rightarrow$ Sub　$R_2, 1, R_2$　　$R_2 \leftarrow R_2 - 1;$

$I_4 \rightarrow$ Add　$R_1, R_2, R_3;$　　$R_3 \leftarrow R_1 + R_2;$

$I_5 \rightarrow$ Add　$R_4, R_5, R_6;$　　$R_6 \leftarrow R_4 + R_5;$

$I_6 \rightarrow$ SL　R_3　　$R_3 \leftarrow SL(R_3)$

$I_7 \rightarrow$ Add　$R_6, R_4, R_7;$　　$R_7 \leftarrow R_4 + R_6;$

在这个例子中,我们可以观察到下表所示的数据相关:

指令	数据相关类型
I_3 和 I_2	先写后读和先写后写(W-W)
I_4 和 I_1	先写后读(R-W)
I_4 和 I_3	先写后读(R-W)
I_6 和 I_4	先写后读和先写后写(W-W)
I_7 和 I_5	先写后读(R-W)

如图 9.7 所示为在考虑了上面提到的数据相关后,这些指令在流水线中的流动过程。图 9.7 的 Gantt 图中所做的假设是依赖于前面指令执行结果的操作数读取,将被延迟到该操作数可用,即结果被保存。在考虑不同指令间的数据相关之后,执行给定的 7 条指令现在总共需要 16 个时间单位。

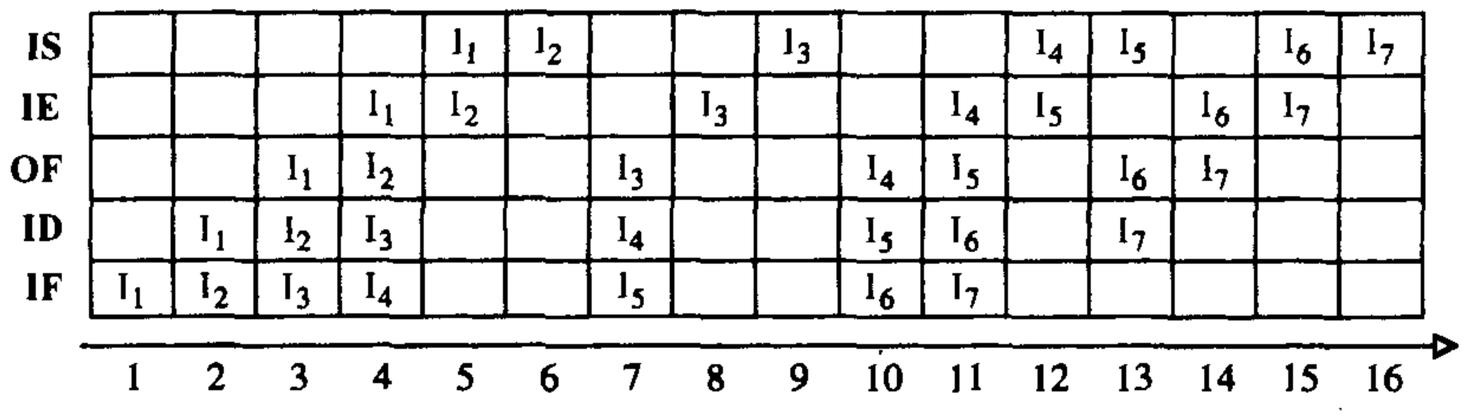


图 9.7　例 9.3 的 Gantt 图

基于上面得到的结果,我们可以计算出执行例 9.3 中给定代码段的加速和吞吐量:

$$\text{加速 } S(5) = \frac{\text{使用串行处理的时间}}{\text{使用流水线处理的时间}} = \frac{7 \times 5}{16} = 2.19$$

$$\text{吞吐量 } U(5) = \text{每单位时间执行的任务数} = \frac{7}{16} = 0.44$$

从上面对由于指令和数据相关而导致流水线停顿的讨论,可以得出与这些相关导致的问题有关的 3 个主要观点,它们是:

1. 指令和数据相关都将导致流水线中额外的时间延迟;
2. 指令相关会导致读取错误指令;

3. 数据相关会导致读取错误操作数。

有一些方法可以解决由指令和数据相关导致的问题。其中一些方法尝试阻止读取错误指令或错误操作数,而另外一些尝试减少由于在流水线中存在指令或数据相关而产生的时间延迟。这些方法将在下面介绍。

用来阻止读取错误指令或错误操作数的 NOP 方法。NOP 方法在指令相关中用来阻止读取错误指令,在数据相关中用来阻止读取错误操作数。回忆例 9.1,在一个由 IF,ID,IE 和 IS 4 级组成的流水线中,执行 10 条指令 I_1 到 I_{10}。为了显示这些指令在流水线中的执行,我们假设当读取转移指令时,流水线会停顿,直到存储转移指令的执行结果。这个假设是用来阻止在读取转移指令后读取错误指令。在实际环境中,需要一个机制来确保在正确的时间读取正确的指令。插入"NOP(空操作)"指令将有助于完成这个任务。"NOP"是一条对处理器状态不产生任何影响的指令。

例 9.4 考虑在一个由 IF,ID,IE 和 IS 4 级组成的流水线上,执行 10 条指令 I_1 到 I_{10}。假设指令 I_4 是条件转移指令。当它被执行时,转移不被执行,即没有满足转移条件。

为了执行这个指令集合,同时又阻止读取错误指令,我们假设插入了一定数量的 NOP 指令,比如在指令 I_4 之后、指令 I_5 之前。图 9.8 显示的 Gantt 图表示了新指令序列的执行(在插入 NOP 指令后)。图中显示在指令 I_4 之后插入 3 个 NOP 指令将确保在 I_4 之后读取正确的指令。在这里,I_5 将只有在时间单位 8 被读取,而那时 I_4 执行的结果已经保存了,转移的条件已经知道了。

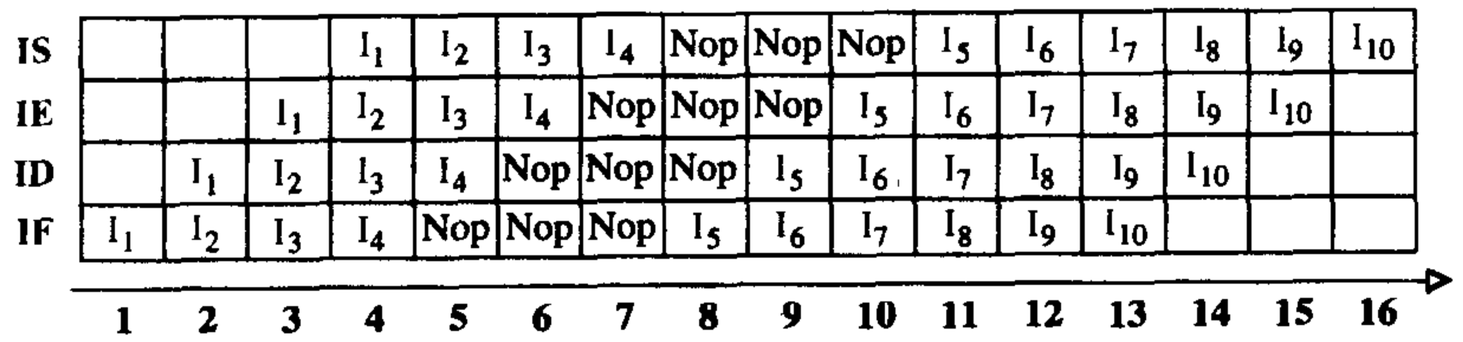


图 9.8 使用 NOP 指令

必须注意所需 NOP 指令的数量等于$(n-1)$,n 表示流水线级数。

例 9.4 表明在指令相关中使用 NOP 指令来阻止读取错误指令。相同的方法可以用在数据相关中阻止读取错误操作数。考虑在一个 5 级(IF,ID,OF,IE 和 IS)流水线上,执行下列代码段:

```
ADD   R1, R2, R3;     R3 ← R1 + R2
SUB   R3, 1, R4;      R4 ← R3 − 1
MOV   R5, R6;         R6 ← R5
```

注意前 2 条指令间的先写后读形式的数据相关。在存储第一条指令的结果前,将不能读取第二条指令的操作数,也就是读取 R_3 的内容。为了达到这个目的,在第一条和第二条指令之间插入 NOP 指令,如下所示:

```
ADD   R1, R2, R3;     R3 ← R1 + R2
NOP
NOP
SUB   R3, 1, R4;      R4 ← R3 − 1
MOV   R5, R6;         R6 ← R5
```

执行修改后的指令序列如图 9.9 所示。图中显示使用 NOP 确保了指令 I_2 在时间单位 6 读取 R_3 的正确值。这个值作为指令 I_1 的执行结果在时间单位 5 存储。

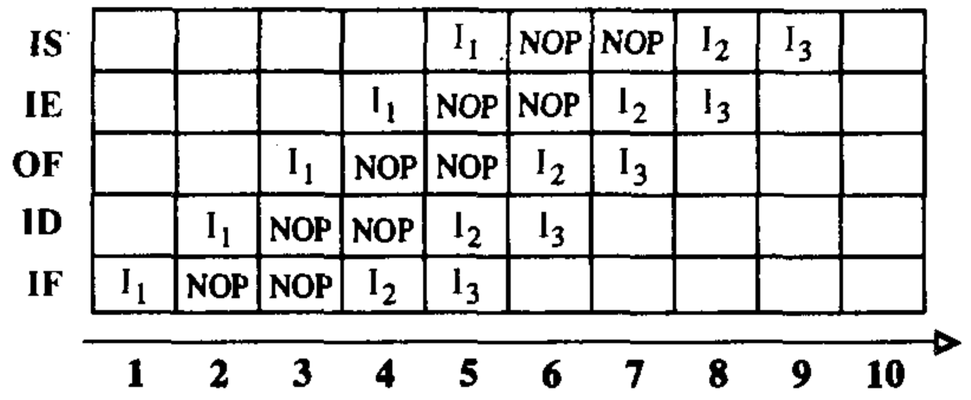


图 9.9　在数据相关中使用 NOP

用来减少由于指令相关导致的流水线停顿的无条件转移指令方法。为了能够减少由于无条件转移导致的流水线停顿，有必要在读取错误指令前尽早识别无条件转移，也有可能通过重排指令序列来减少停顿。这些方法将在下面解释。

指令重排。重排指令的序列使正确的指令能发送到流水线，同时又确保重排后的指令集合能产生正确的结果。例如考虑在一个由 IF，IE 和 IS 3 级组成的流水线上，执行下列指令集合 I_1，I_2，I_3，I_4，I_5 . . ，I_j，I_{j+1}，. . . . 。在这个指令集合中，I_4 是无条件转移指令，其目标指令是 I_j。以给定顺序执行这个指令集合将导致在指令 I_4 后不正确地读入指令 I_5。但是考虑执行重排的序列 I_1，I_4，I_2，I_3，I_5 . . I_j，I_{j+1}，. . 。如图 9.10 所示，使用 3 级流水线执行这个重排的序列。

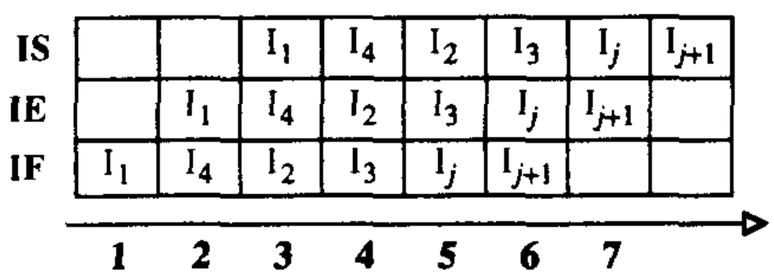


图 9.10　指令重排

图 9.10 中显示了指令的重排使指令 I_j 将在时间单位 5 被读取，也就是执行指令 I_4 后。“聪明”的编译器能够实现指令重排，它能扫描代码序列，决定指令的正确重排顺序，从而在产生正确的最终结果的同时，最小化由于指令相关而损失的时间单位数量。为了使重排指令的方法能产生正确的结果，必须满足一个重要条件，即与转移指令交换的指令集合中不存在任何数据和/或指令相关。

*在读取部件中使用专用硬件。*在这里，读取部件被假设与一个专用硬件部件有联系，它能识别无条件转移指令并尽可能快地计算出转移的目标地址。例如考虑执行上面所述的相同指令序列。同时假设读取部件有一个专用硬件部件，它能够在不需要额外时间单位的情况下，识别无条件转移指令并计算出转移的目标地址。如图 9.11 所示为这个指令序列的 Gantt 图。图中显示其执行了正确的指令序列而且不需要额外单位时间。

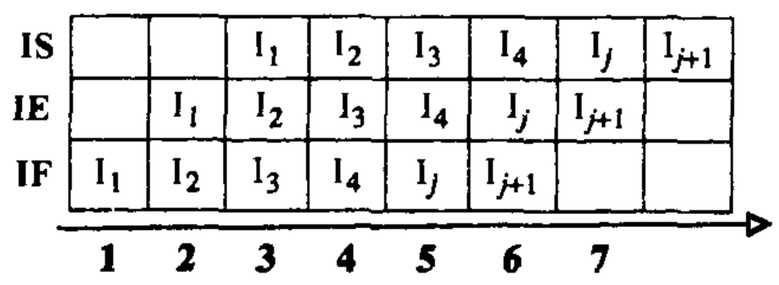


图 9.11　使用额外硬件部件来识别转移指令

不需要额外时间单位就能识别转移指令并计算出目标转移地址的假设毕竟是不现实的。在通常的情况中,读取部件增加的硬件部件将需要额外的时间单位来识别转移指令并计算目标转移地址的任务。在硬件部件执行所需的额外时间单位时,若能执行其他指令,并就有可能减少额外时间单位甚至完全消除额外的时间单位代价。这就是下面要介绍的方法的本质。

转移的预计算和指令的重排。这种方法被认为是前面所述方法的组合。在这里,专用硬件(用来识别转移指令和计算目标转移地址)与其他指令并发执行。例如考虑上面提到的相同指令序列,同时假设专用硬件部件需要一个时间单位来执行它的任务。在这里重排后的指令变成 $I_1, I_2, I_4, I_3, I_5 . , . I_j, I_{j+1} . , . ,$ 它将产生正确的结果而不会导致额外的时间单位损失。使用 Gantt 图,如图 9.12 所示。注意时间单位 4 被专用硬件单元用来计算目标地址,同时用来并发读取指令 I_3。

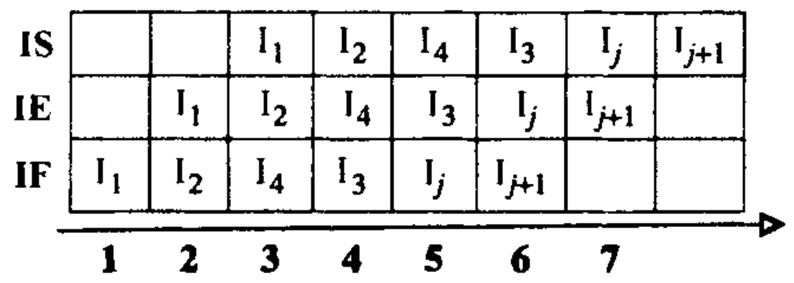


图 9.12 转移折叠

必须注意这种方法的成功依赖于在执行可用的指令同时,专用硬件能并发计算目标转移地址。在上面的情况中,假设指令的重排能提供那些能与计算目标转移地址并发执行的指令。但是若这样的重排行不通,则使用指令队列和指令预取两者就能帮助提供所需的条件,这将在下面解释。

指令预取。这种方法需要指令能够在需要它们之前被读取,并存储在指令队列中。这种方法也要求读取部件有所需的硬件来识别转移指令并计算目标转移地址。若由于数据相关导致的流水线停顿而没有新指令被读取到流水线中,则读取部件能够使用这样的时间继续读取指令并把它们添加到指令队列中。另一方面,若读取指令时发生时间延迟,例如由于指令相关,则那些在指令队列中的预读取指令能够为流水线提供新的指令,从而消除一些由于指令相关而导致的时间单位损失。由指令队列提供合适的指令到流水线中通常使用所谓的"分派单元"来完成。这种由于指令相关导致流水线停顿时预取指令和执行指令的技术称为"转移折叠"(branch folding)。

用来减少由于指令相关导致的流水线停顿的条件转移指令方法。上面讨论的在无条件转移指令情形中的技术在条件转移指令中可能不适用。这是因为在转移指令执行完成前,条件转移的目标转移地址将不会被知道。由于条件转移带来的指令相关导致了时间单位的损失,有一些技术能用来最小化损失的数量。

延迟转移。延迟转移是指有可能将条件转移指令后的位置(称为转移延迟槽)用可执行的有用指令填满,直到知道目标地址。例如考虑在一个由读取(F)和执行(E)2 级组成的流水线上,执行下列程序循环。

```
I1 →  Again:  Load 5, R1;          R1 ← 5;
I2 →          Sub  R2;             R2 ← R2 − 1;
I3 →          Bnn  Again;          Branch to Again if result is Not Negative;
I4 →          Add  R4, R5, R3;     R3 ← R4 + R5;
```

必须注意的是在第一个循环的结尾，根据指令 I_3 的执行结果，必须读取指令 I_1 或是指令 I_4 中的一个。处理这种情况的方法是延迟读取下一个指令，直到知道指令 I_3 的执行结果。这将导致流水线中额外的时间延迟，但是这里额外的延迟可以通过如下面所示的重排指令序列避免：

```
Again:   Sub   R2;          R2 ← R2 − 1;
         Load  5, R1;        R1 ← 5;
         Bnn   Again;        Branch to Again if result is Not Negative;
         Add   R4, R5, R3;   R3 ← R4 + R5;
```

如图 9.13 所示为在进入循环时 $R_2 = 3$ 的情况下，执行修改后代码段的 Gantt 图。

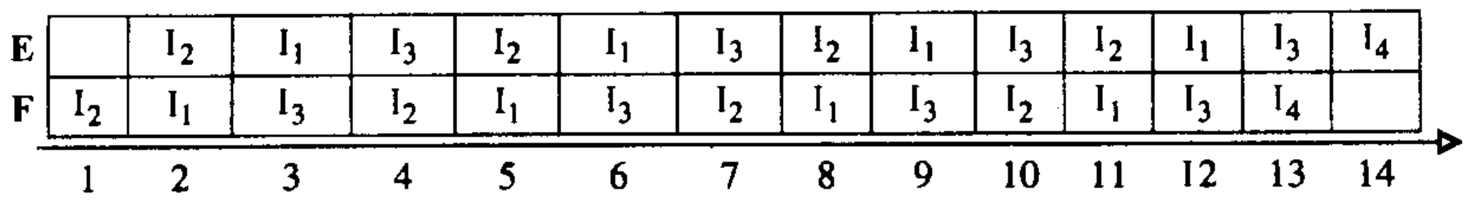


图 9.13　延迟转移

图中表明转移发生在实际位置(即转移指令在原先指令序列中出现的位置)后一个指令，因此称为“延迟转移”。从图 9.13 中还可以清楚地看出，通过重排指令序列，有可能使用有用的指令填充转移延迟时间槽，从而消除流水线中额外的时间延迟。一些研究表明有些“聪明”的编译器在 80%的情况下能够利用转移延迟时间槽。通过使用“聪明”的编译器，充分利用转移延迟时间槽，可以提高处理器的加速和吞吐量。

读取下一指令的预测。这种方法通过在读取条件转移指令后预测下一条读取指令，尝试减少由于指令相关导致的可能损失的时间单位。这种方法的基础是若转移的结果是随机的，则有可能减少 50%的损失时间。实现这个技术的简单方法是假设只要遇到条件转移，系统就预测转移将不会执行(或者是将会执行)。在这个方法中，可以通过读取顺序地址序列的指令继续(或者通过读取目标转移指令起始处的指令继续)。当完成执行转移指令时，将会知道结果，必须做出决定判断根据假设转移不被执行(或是被执行)而已经执行的指令是否是想要的正确指令序列。这个决定的结果有两种可能。若预测正确，则执行将继续而无需浪费额外时间单位。另一方面，若预测错误，则必须小心地将机器状态(通常以存储器和寄存器的内容加以衡量)回退到如同没有发生过猜测执行(speculative execution)一样。

基于上面模式的预测将导致在每次遇到给定指令时，做出相同的转移预测决定，因此称为静态转移预测。这种最简单的转移模式在编译时实现。

另外一种能够用在转移预测中的技术是动态转移预测。在这里，预测在运行时完成而不是在编译时。当遇到一个转移时，就检查记录，参考以前是否遇到过同样的转移，如果遇到过，看当时做出的决定是什么，也就是转移是否执行，然后做出一个执行或是不执行转移的运行时决定。在做出这样的决定时，可以使用两状态算法，“可能要执行”(LTK)或“可能不要执行”(LNK)。若当前状态是 LTK，而转移执行，则算法将保持 LTK 状态；否则它将转换到 LNK。而若当前状态是 LNK，而转移没有执行，则算法将保持 LNK 状态；否则它将转换到 LTK 状态。这个简单的算法能够很好地工作，特别是当转移向后时，例如在循环执行的过程中。但是当最后一次循环时，它将会导致错误预测。更健壮的算法使用了 4 个状态，ARM 11 微体系结构(见下文)就使用了这种算法。

有意思的是,动态和静态转移预测技术的组合能提高性能。一开始先尝试使用动态转移预测,若不行,则系统采用静态预测技术。

例如考虑 ARM 11 微体系结构(ARMv6 指令集体系结构的第一个实现),它使用了动态/静态转移预测的组合。64 项 4 状态转移目标地址高速缓存(BTAC)中的记录,用来帮助动态转移预测找出一个给定的转移在以前是否出现过。若该转移出现过,记录将显示它是否是经常执行转移还是不执行转移。若 BTAC 显示以前曾遇到过这个转移,则基于以前的结果做出一个预测。这 4 个状态是强执行、弱执行、强不执行和弱不执行。

在这里,若不能找到转移的记录,则使用静态转移预测过程。静态转移预测过程研究该转移从而找出它是向后还是向前。向后的转移被假设是循环的一部分,转移被假设为执行。向前的转移则不执行。ARM 11 使用了 8 级流水线。每个正确转移的预测通常将节省 5 个处理器时间周期。在 ARM 11 体系结构中,由于使用了动态/静态预测,大约 80% 的转移预测是正确的。ARM 11 的流水线特性将在后面解释。

UltraSPARC III RISC 的 14 级流水线处理器使用的转移预测技术基于 16 K 项的转移历史记录。错误预测的影响,特别是由于转移预测错误而损失的时钟周期数可以通过下面的方法加以减少。在预测转移将被执行,以及在读取转移目标指令的同时,通过 4 项转移缺失队列(BMQ)并行地发射"转移不发生"时所需的指令。这将预测错误的损失减少到了 2 个周期。UltraSPARC III 在转移预测中达到了 95% 的成功率。UltraSPARC III 的流水线特性将在下一小节解释。

用来减少由于数据相关导致流水线停顿的硬件操作数转发方法。硬件操作数转发使得一个算术逻辑部件操作的结果在下一个周期对另一算术逻辑部件操作可用。考虑下面 2 条指令:

$$\begin{aligned} &\text{ADD} \quad R_1, R_2, R_3; \qquad R_3 \leftarrow R_1 + R_2 \\ &\text{SUB} \quad R_3, 1, R_4; \qquad R_4 \leftarrow R_3 - 1 \end{aligned}$$

很容易注意到在这 2 条指令之间存在先写后读的数据相关。在一个 5 级(IF,ID,OF,IE 和 IS)流水线中正确执行这个序列将导致第二条指令在译码后停顿,直到第一条指令的结果存储在 R_3 中。只有在那个时候,第二条指令的操作数(也就是存储在 R_3 中更新的值)才能被第二条指令读取。但是有可能在将第一条指令的结果存储到 R_3 的同时,将结果转发到算术逻辑部件,从而减少停顿时间。如图 9.14 所示。

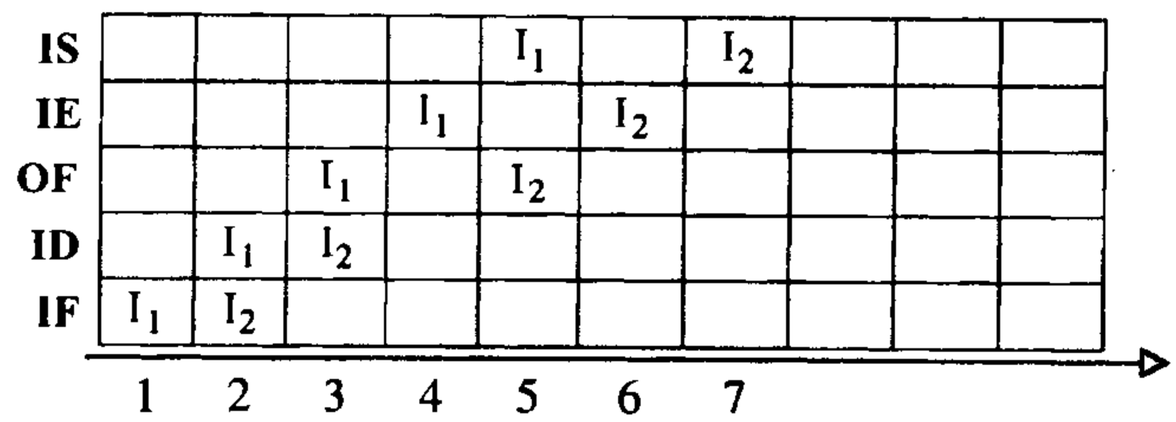


图 9.14 硬件转发

假设第二条指令的操作数能够在它可用后在存储到 R_3 的同时马上转发,其前提是需要修改数据通路,例如创建附加的反馈通路来转发操作数。在图 9.15 中用虚线标出了这样的修改。应注意实现硬件操作数转发所需的修改是昂贵的,而且需要小心地发射控制信号。同时也要注意若有可能在相同时间单位同时执行指令译码和操作数读取将不会有时间单位损失。

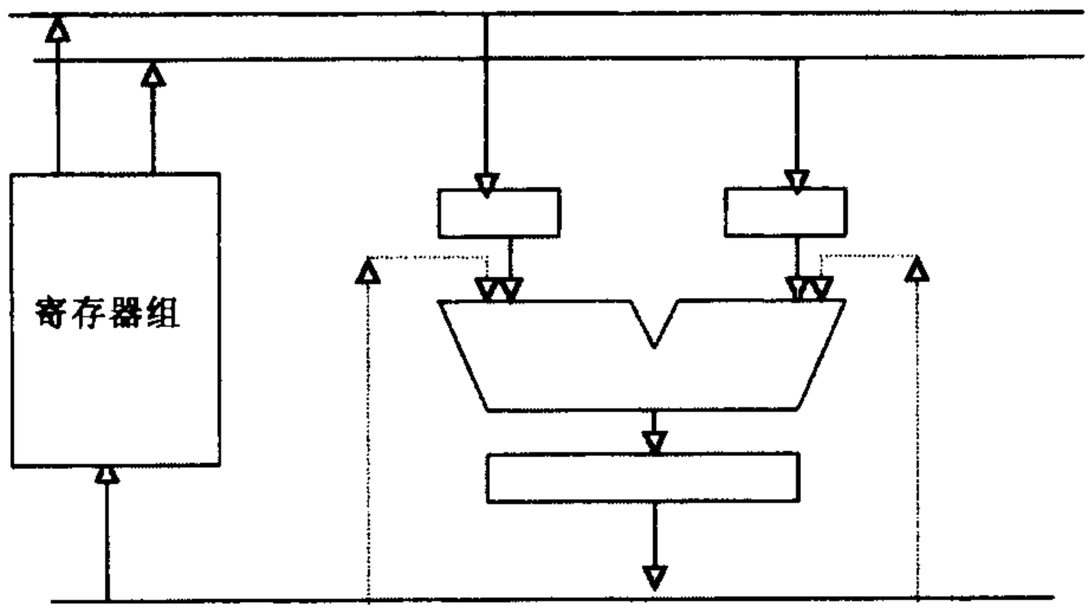


图 9.15　硬件转发

用来减少由数据相关导致流水线停顿的软件操作数转发方法。操作数转发也能够由编译器以软件的方式执行。在这里，编译器必须足够“聪明”以使一些指令的执行结果(比如操作数)迅速地对接下来的指令可用。这个期望的特性需要编译器执行数据相关分析，以决定有可能对接下来指令可用(转发)的操作数，从而减少停顿时间。这种数据相关分析需要识别三种基本格式。接下来将通过简单的例子来解释。

存储 – 读取。这种情况表示由于上一条指令的结果存储在存储器中，而下一条指令请求读取同样的结果而产生的数据相关。考虑下面 2 条指令序列：

Store　R_2, (R_3);　　$M[R_3] \leftarrow R_2$
Load　(R_3), R_4;　　$R_4 \leftarrow M[R_3]$

在这个序列中，第二条指令所需的操作数(存储器单元中的内容，其地址存储在寄存器 R_3 中)已经在寄存器 R_2 中可用，因此能够迅速地(转发)移动到寄存器 R_4 中。

当“聪明”的编译器认出这种数据相关时，它能将上面的序列替换成下面的序列：

Store　R_2, (R_3);　　$M[R_3] \leftarrow R_2$
Load　R_2, R_4;　　$R_4 \leftarrow R_2$

读取 – 读取。这种情况表示的数据相关即上一条指令存储的数据也是下一条指令所需的操作数。考虑下面的指令序列：

Load　(R_3), R_2;　　$R_2 \leftarrow M[R_3]$
Move　(R_3), R_4;　　$R_4 \leftarrow M[R_3]$

在这个序列中，第一条指令所需的操作数(存储器单元中的内容，其地址存储在寄存器 R_3 中)同时也是第二条指令所需的操作数。因此操作数能够迅速(转发)移动到寄存器 R_4 中。当“聪明”的编译器认出这种数据相关时，它能将上面的序列替换成下面的序列：

Load　(R_3), R_2;　　$R_2 \leftarrow M[R_3]$
Move　R_2, R_4;　　$R_4 \leftarrow R_2$

存储 – 存储。这种情况表示的数据相关是上一条指令存储的数据被下一条指令改写。考虑下面的指令序列：

Store　R_2, (R_3);　　$M[R_3] \leftarrow R_2$
Store　R_4, (R_3);　　$M[R_3] \leftarrow R_4$

在这个序列中,第一条指令写入的结果(将寄存器 R_2 的内容写到存储器单元中,其地址存储在寄存器 R_3 中)被第二条指令以寄存器 R_4 的内容改写。假设这2条指令顺序执行,第一条指令写入的结果对输入/输出操作而言是不需要的,例如 DMA 操作,则这2条指令的序列可以替换成下面的一条指令。

Store R_4, (R_3); $M[R_3] \leftarrow R_4$

9.3 流水线处理器实例

在本节中,我们简单地介绍两个流水线处理器,它们使用了本章中所描述的不同流水线技术。我们介绍的重点是这些体系结构的流水线特性。这两个处理器是 ARM 1026EJ-S 和 UltraSPARC III。

9.3.1 ARM 1026EJ-S 处理器

ARM 1026EJ-S 处理器是 ARM 公司设计的 RISC 处理器系列的一个成员。这个系列是为高性能、低成本和低功耗的嵌入式应用而设计的。ARM 1026EJ-S 整数核心有多个执行部件,因此在相同流水线阶段允许存在多条指令,它还允许并发执行指令。ARM 1026EJ-S 能达到每周期一条指令的峰值吞吐量。整数核心由下列部件组成。

1. 预读取部件:该部件负责指令的读取。只要有可能,它也预测转移的结果;
2. 整数部件:该部件负责对来自预读取部件的指令译码。该部件包括桶形位移器、ALU 和乘法器。它执行的指令例如 MOV,ADD 和 MUL。整数部件帮助载入/存储部件执行载入和存储指令。它也帮助执行一些协处理器指令;
3. 载入/存储部件:该部件能每周期载入或存储2个寄存器(64位)。

ARM 1026EJ-S 是一个流水线处理器,其 ALU 由以下6级组成。

1. 读取级:访问指令高速缓存并为已经读取的指令进行转移预测;
2. 发射级:进行初始的指令译码;
3. 译码级:进行最终的指令译码,为 ALU 操作读寄存器,进行操作数转发以及初始的互锁分解;
4. 执行级:进行数据访问地址的计算,数据处理移位,移位和饱和处理,ALU 操作,第一阶段乘法,标志位设置,条件代码检查,转移错误预测的侦测,以及存储读入的数据寄存器;
5. 存储器级:进行数据高速缓存访问,第二阶段乘法和饱和处理;
6. 写入级:进行寄存器写和指令退出。

在这种布局中,读取级使用了能保存3条指令的先入先出(FIFO)缓冲器。发射和译码级能同时容纳一条预测的转移指令和另一条指令。执行、存储器和写入级能同时容纳下列任意的指令:

1. 一条预测的转移指令;
2. 一条 ALU 或乘法指令;
3. 正在进行的多条乘法载入或存储指令;
4. 正在进行的多周期协同处理器指令。

预读取部件在读取级操作,而且它能从指令端高速缓冲中每周期读取64位。但是它每周期只能发射一条32位指令到整数部件。预读取部件将等待中的指令放置到预读取缓冲器中。当一条

指令在预读取缓冲器中时,转移预测逻辑能译码该条指令,以检测它是否是可预测的转移。预读取缓冲器最多能保存 3 条指令,且能使预读取部件进行如下操作:

1. 在读取级前检测出转移指令;
2. 预测那些可能要执行的转移;
3. 移去那些不可能执行的转移。

若转移被预测为执行,则重定向该指令地址到转移目标地址。若转移被预测为不执行,则读取下一个指令。在没有足够时间完成转移处理时,仍然重定向读取地址,从而减少转移预测错误的损失。

整数部件执行不可预测转移。为了迅速得到所需地址,使用了专用的快速转移加法器。这样做是为了避免通过桶形移位器传递。

预读取缓冲器在下列情况中将进行刷新:

1. 进入一个异常处理序列;
2. 载入程序计数器(PC);
3. PC 的算术处理;
4. 执行不可预测转移;
5. 检测错误预测转移。

执行的预测转移是惟一不会导致自动刷新预读取缓冲器内容的情况。错误转移和执行不可预测的转移将导致 3 个周期的损失。

9.3.2　UltraSPARC III 处理器

UltraSPARC III 基于 SUN SPARC-V9 RISC 体系结构规范。下面所列是 SPARC-V9 特征的一部分:

1. 少量而简单的指令格式。所有指令都是 32 位。存储器访问只使用载入和存储操作;
2. 少量的寻址模式。存储器寻址只有两种模式,寄存器 + 寄存器和寄存器 + 直接模式;
3. 三元寄存器操作数。大部分指令操作有两个寄存器操作数或是 1 个寄存器操作数和 1 个常量。这些结果存储到第三个寄存器中;
4. 巨大的窗口寄存器文件。

UltraSPARC III 处理器使用了 6 个独立的部件,如图 9.16 所示。它们是:

1. 指令发射部件(IIU)。该部件预测程序流程,从存储器中读取预测的路径,发送读取的指令到执行流水线。指令被转发到 IEU 或 FPU。IIU 与一个 4 路组相联指令高速缓存、一个地址转换缓冲器和一个 16 K 项转移预测器协同工作;
2. 整数执行部件(IEU)。该部件执行所有整数操作,包括整数载入和存储、整数算术、逻辑运算和转移指令。IEU 每周期能并发执行多达 4 条的整数指令;
3. 数据高速缓存部件(DCU)。该部件包括 3 个不同的一级(L1)数据高速缓存和 1 个数据地址转换缓冲器。3 个数据高速缓存分别是:请求读取(64 KB 的 4 路组相联,块大小为 32 字节)、预读取高速缓存(2KB 的 4 路组相联,块大小为 64 字节)和写高速缓存(2KB 的 4 路组相联,块大小为 64 字节);
4. 浮点部件(FPU)。该部件执行所有浮点和图形指令;

5. 外部存储器部件(EMU)。该部件控制对2个芯片外存储器模块的访问。这2个芯片外模块分别是2级(L2)数据高速缓存和主存;
6. 系统接口部件(SIU)。该部件在微处理器和外部系统之间提供通信接口,这里的外部系统是指主存、输入/输出设备和多处理机系统中的其他处理器。

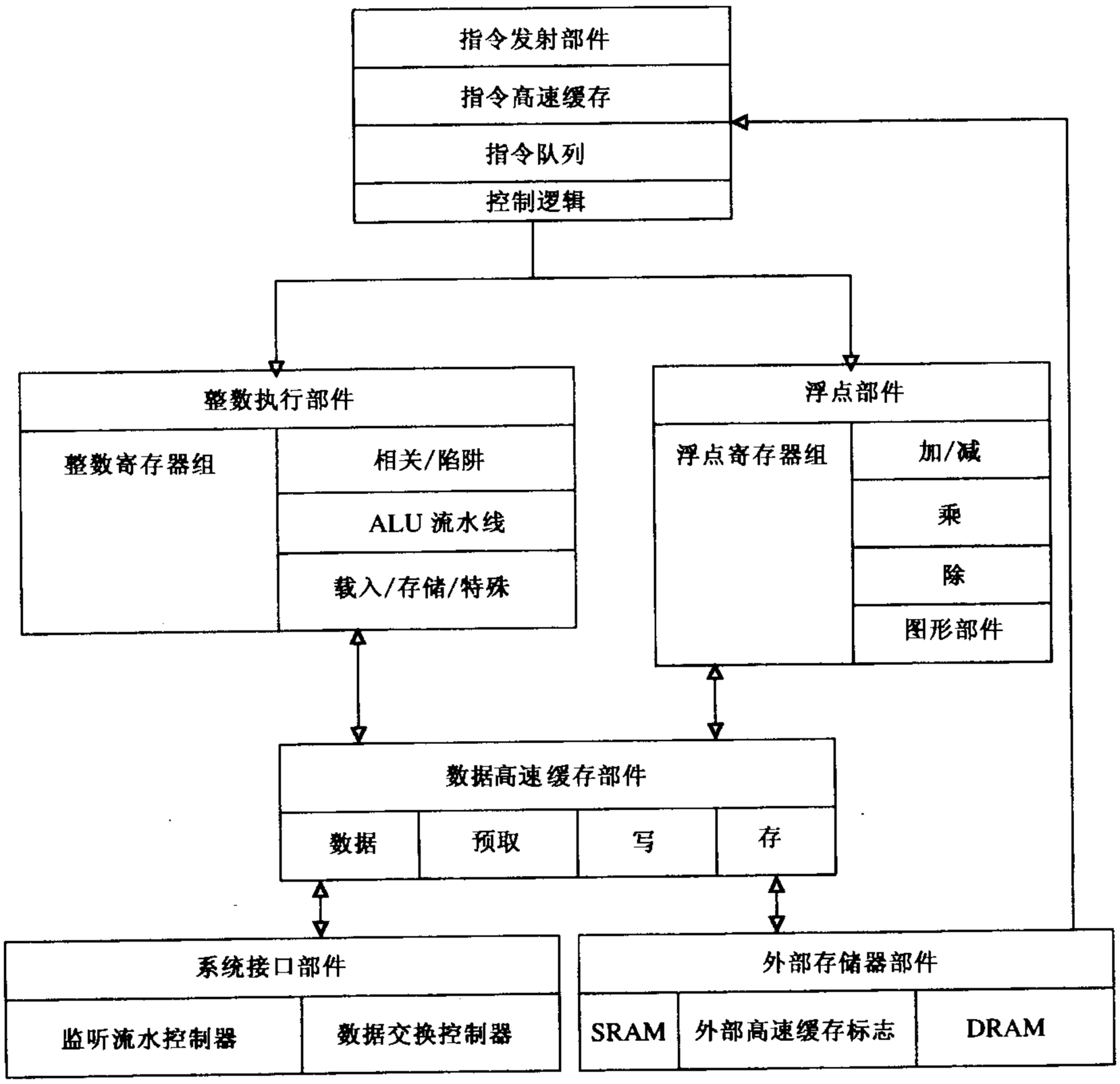


图 9.16 UltraSPARC III 的功能部件

UltraSPARC III 有一个 14 级的流水线,各级介绍如下:

1. 地址生成部件(A):该部件生成指令读取地址;
2. 指令预读取部件(P):该部件从高速缓存中读取指令的第二周期,访问转移预测的第一个周期;
3. 指令读取部件(F):该部件从高速缓存中读取指令的第二周期,访问转移预测的第二周期。该部件也执行虚拟地址到物理地址的转换;
4. 转移目标计算部件(B):该部件计算转移的目标地址,译码指令的第一周期;
5. 指令译码部件(I):该部件译码指令的第二周期,发送它们到队列;
6. 指令控制部件(J):该部件发送指令到合适的执行部件。整数指令被发送到整数执行部件,而浮点数和图形指令被发送到浮点部件;
7. 寄存器组读入部件(R):该部件读入整数寄存器组的操作数;
8. 整数执行部件(E):该部件执行整数指令。

9. 数据高速缓存访问部件(C):该部件访问数据高速缓存的第二周期,按字和双字转发载入的数据,载入和执行浮点指令的第一周期;
10. 存储器旁路部件(M):该部件为半字和字节的载入进行数据对齐,载入和执行浮点指令的第二周期;
11. 工作寄存器组写部件(W):该部件对整数寄存器文件执行写操作,执行浮点指令的第三周期;
12. 流水线扩展部件(X):该部件为精确浮点陷阱扩展了整数流水线,执行浮点指令的第四周期;
13. 陷阱部件(T):该部件在陷阱发生后报告它们;
14. 完成部件(D):该部件写体系结构的寄存器组。

UltraSPARC III 中有两项主要的技术是用来处理转移的,它们将在下面解释。

转移预测。UltraSPARC III 使用的转移预测技术融合了前面提到的静态和动态转移预测技术。在这里,转移预测发生在 IIU 部件。它使用转移预测表和动态预测算法的硬件实现。

转移预测表。转移预测表(BPT, Branch Prediction Table)是二位有限状态机(FSM, Finite State Machine)表的硬件实现。它是一个饱和式的增 - 减计数器。当转移发生时,转移目标地址和/或转移历史用来找出发现转移预测位置的表索引。转移条件若对应于两种 FSM 状态的一种:强不执行或弱不执行。转移条件若对应于两种 FSM 状态的一种:弱执行或强执行。计数器在每次转移执行时增加,否则减少,所以称为增 - 减计数器。若计数器达到强执行状态(11 状态),只要转移执行它将一直保持该状态;若它达到强不执行状态(00 状态),只要转移不执行它将一直保持该状态,所以称为饱和。UltraSPARC III 中的 BPT 由 16 K 的项(16 K 二位饱和增 - 减计数器)组成。

全局共享动态预测算法。全局共享(gshare)算法使用两级转移历史信息来动态预测转移的方向。第一级记录了最近遇到的 k 个转移的历史,表示全局转移行为。它通过一个全局转移历史寄存器实现。这是一个移位寄存器,每次遇到执行转移就输入 1,每次遇到不执行转移就输入 0。转移历史信息的第二级记录了 k 个转移中符合指定模式转移的最近 s 次出现。这个信息保存在转移预测表中。全局共享算法通过读取转移目标地址的低位,使用历史寄存器异或它们,来得到需要在预测表中使用的索引。

UltraSPARC III 使用了全局共享算法的一个改进版本。这个改进需要预测器经历流水线的两个阶段,也就是若使用原来的全局共享算法,预测器被程序计数器的一个旧备份来索引。改进的全局共享算法在每次访问预测器时,读取 8 个计数器,在 B 流水线阶段使用程序计数器寄存器的低 3 位挑选其中一个。

指令缓冲器(队列)。UltraSPARC III 指令发射部件(IIU)与两个指令缓冲队列协同工作。它们是转移指令队列(BIQ)和转移缺失队列(BMQ)。

转移指令队列(BIQ)。一个 20 项的队列,允许读取和执行部件独立地操作。读取部件预测执行路径,且连续填充 BIQ。当遇到一个执行转移时,将损失两个读取周期来填充 BIQ。

转移缺失队列(BMQ)。在损失的两个周期中,串行指令已经被访问并缓冲到一个 4 项的 BMQ 中。若发现转移预测错误,BMQ 中的指令将被直接发送到执行部件。

9.4 指令级并行

与流水线技术相比,指令级并行(ILP)是基于多发射处理器(MIP)的设计思想。多发射处理器具有多个用来执行指令的流水线数据通路。每个流水线每周期能发射和执行 1 条指令。如图 9.17 所示为有 3 条流水线的处理器情况。出于比较的目的,我们在图中同时给出了顺序处理和单流水线的情况。从图中可以清楚地看出,就每条指令所需周期(CPI)数的限制而言,单流水线中的 CPI = 1,而 MIP 能达到 CPI < 1。

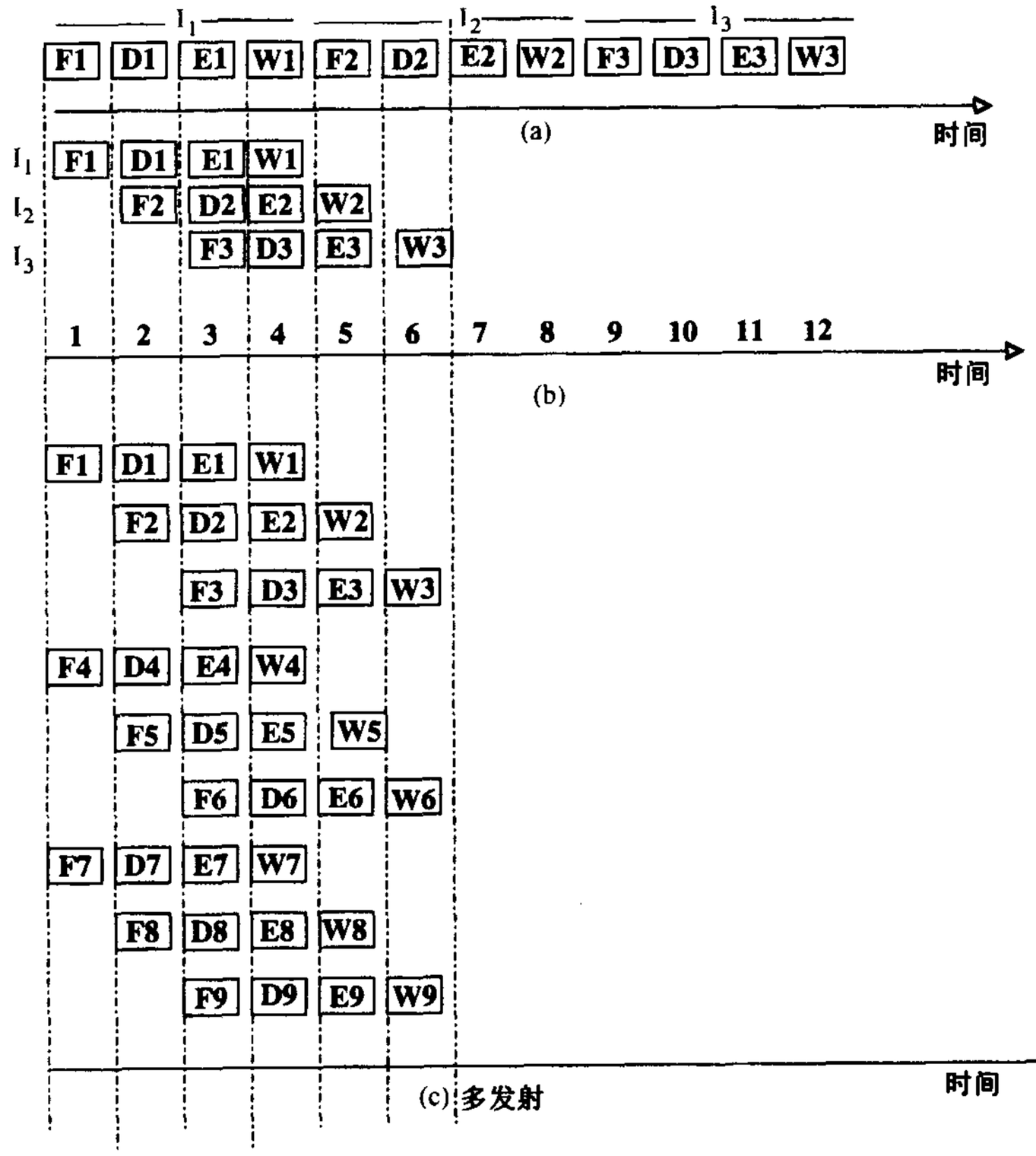


图 9.17 多发射、流水线及顺序处理:(a)顺序处理;(b)流水线;(c)多发射

为了充分利用指令级并行,必须进行分析以识别给定程序中存在的指令和数据相关。该分析可以保证指令组的合理调度,使得在发射它们的同时保持程序的正确性。静态调度配合超长指令字(VLIW)体系结构的使用,动态调度配合超标量体系结构的使用。

在 VLIW 中,一条指令代表需要同时发射的一束多个操作。编译器负责检查所有的相关,对操作进行合理的编组/调度。超标量体系结构则不同,它完全依赖于硬件调度指令。

超标量体系结构。标量机每次只能执行一个算术操作。超标量体系结构(SPA, superscalar architecture)能同时对多条指令进行读取、译码、执行和结果存储。为了同时执行多条指令,它将静态、串行的指令流转变成动态、并行的指令流。当指令完成时,SPA 再应用原来的串行指令序列,使指令能以原来的顺序结束。

SPA 指令的处理包括读取、译码、发射和提交阶段。在读取阶段,多个指令被同时读取。转移预测和猜测执行也在读取阶段中完成。这是为了在转移和跳转指令后继续读取指令。

译码分两步实现。预译码负责识别转移指令,在主存和高速缓存间执行。真正的译码用来为每条指令决定:(1)要执行的操作数;(2)操作数所在的位置;(3)要保存的结果位置。发射阶段识别出那些已分派且能开始执行的指令。在提交阶段,生成的值/结果被写到它们的目的寄存器。

SPA 中处理指令的最重要步骤是相关分析。这种分析的复杂性与指令字的规模成平方关系。这就限制了 SPA 所能达到的并行度,比如并行度超过 4 是不现实的。超过这个限制,相关分析和调度必须由编译器来实现。这是 VLIW 方法的基础。

超长指令字(VLIW)。在这个方法中,编译器执行相关分析,决定操作的合理编组/调度。能同时执行的操作被编组成一个超长指令字。因此为了保证最大的可能并行度,指令字被设计得足够长。例如 IBM DAISY 机器有 8 个操作长度的指令字,因此称为 8 发射机器。

在 VLIW 中,资源绑定能够通过将指令字的每个字段分别专用于一个惟一的功能单元的方式实现。但是这种安排限制了每周期能发射的混合指令数。更灵活的做法是允许给定指令的字段被不同类型的操作占用。例如 Philips TriMedia 机器是个 5 发射机器,有 27 个映射到 5 发射槽的功能部件。在 IBM DAISY 中,每条指令实现多通路选择模式。在这种情况中,VLIW 的前 72 位称为头,并含有树形信息、条件测试和转移目标的信息。头后面是 8 个 23 位的包,每个译码为一个操作。为了解决向大量功能部件提供操作数的问题,IBM DAISY 保存了同一寄存器组的 8 个副本,每个对应 8 个功能部件的 1 个。

9.5　算术流水线

指令流水线中使用的原则也能用来提高执行算术操作的速度,比如加、减和乘的计算机性能。在这里,这些原则将用来实现 ALU 中的算术电路。在本节中,我们将详细阐述如何使用算术流水线的方法加速算术操作。我们从定点算术操作开始,然后讨论浮点操作。

9.5.1　定点算术流水线

在 ALU 中执行的基本定点算术操作是将两个 n 位操作数 $A = a_{n-1}\,a_{n-2}\cdots a_2\,a_1\,a_0$ 和 $B = b_{n-1}\,b_{n-2}\cdots b_2\,b_1\,b_0$ 相加。这两个操作数的相加可使用许多技术来完成。这些技术在两个基本属性上有所区别:复杂性和能达到的速度。这两个属性在某种程度上互相矛盾,也就是说简单的实现可能得到较慢的电路,而复杂的实现可能得到较快的电路。例如考虑行波进位通过加法器(CRTA)和先行进位加法器(CLAA)。CRTA 简单但是缓慢,而 CLAA 复杂但是快速。

可以将一些操作数对在加法器中执行,即以流水线的方式改进 CRTA,从而提高 CRTA 中加法的整体速度。如图 9.18 所示为改进的 4 位 CRTA 的一个例子。在这里,两个操作数 A 和 B 通过使用同步元件,比如时钟控制锁存器,提供给 CRTA。这些锁存器将确保在 CRTA 中部分进位值的移动将与后续高位加法器的输入同步。例如通过使用锁存器,就可使第一位的进位输出(c_0)和第二个位对(a_1 和 b_1)的到达在第二个全加器(从低顺序位计数到高顺序)的输入处得到同步。

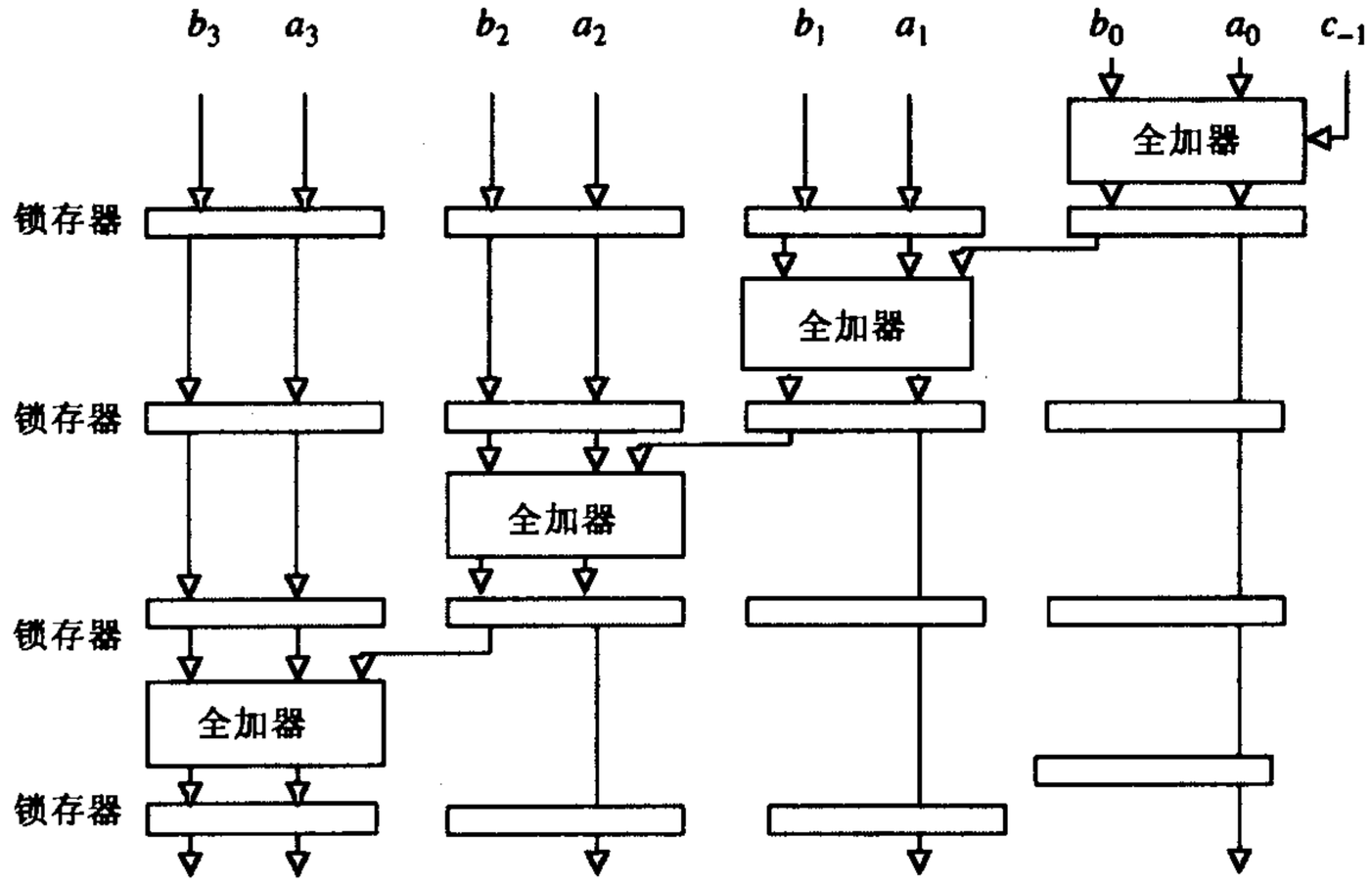


图 9.18 改进的 4 位 CRTA

虽然改进 CRTA 的操作在原则上保持一致,即进位以行波的方式通过加法器,但使用锁存器则有可能向加法器同时提供多对操作数。例如考虑 M 对操作数进行加法的情况,其中每对操作数是 n 位。不使用流水线的 CRTA 执行这 M 对加法的时间是 $T_{np} = M \times n \times T_a$,其中 T_a 是执行一位加法的时间。这可与与使用流水线的 CTRA 进行同样计算所需的时间 $T_{pp} = (n + M - 1) \times T_a$ 加以比较。例如 $M = 16, n = 64$ 位,则我们有 $T_{np} = 1024 \times T_a$ 和 $T_{pp} = 79 \times T_a$,从而算出加速大约是 13。在极端的例子中,即可能同时提供无数对操作数(M)给 CRTA,此时加速将达到 64,即每个操作数的位数。

9.5.2 浮点算术流水线

使用相似的方法,有可能将浮点(FP)加法/减法流水线化。在这里,流水线必须围绕执行浮点加法所需的操作来组织。浮点加法所需的主要操作是指数比较(EC)、指数对齐(EA)、加法(AD)和规格化(NZ)。因此一个可能的流水线组织是 4 级流水线,每个执行 EC,EA,AD 和 NZ 中的一个操作。如图 9.19 所示为浮点流水线加法器的模式,可以同时在加法器中处理多个浮点操作数的集合,从而减少浮点加法所需的总时间。和前面一样,需要同步锁定器来同步浮点加法器中给定级输入的操作数。

9.5.3 使用进位 – 保留加法的流水线乘法

和前面介绍的一样,加法的一个主要问题是事实上进位必须以行波的方式从一级传送到下一级。可以使用一种称为进位 – 保留的方法来取消进位的行波方式逐级传送。考虑对 44,28,32 和 79 做加法的情况,一种将这些数相加而无需使进位以行波方式逐级传送的方法如图 9.20 所示。这个想法是在加法最后一步前,推迟对由中间级产生的进位执行加法,而只在最后阶段使进位以行波方式逐级传送。

进位-保留加法能作为构件块实现流水线乘法。例如考虑对两个 n 位操作数 A 和 B 进行乘法。乘法操作可如图 9.21 所示转换成加法。如图 9.21 所示为两个 8 位的操作数 A 和 B 进行乘法的情况。基于进位 – 保留的乘法模式如图 9.22 所示,它使用图 9.21 显示的原则。该模式基于先产生所需的部分结果集合,再使用进位 – 保留加法模式相加的方法。

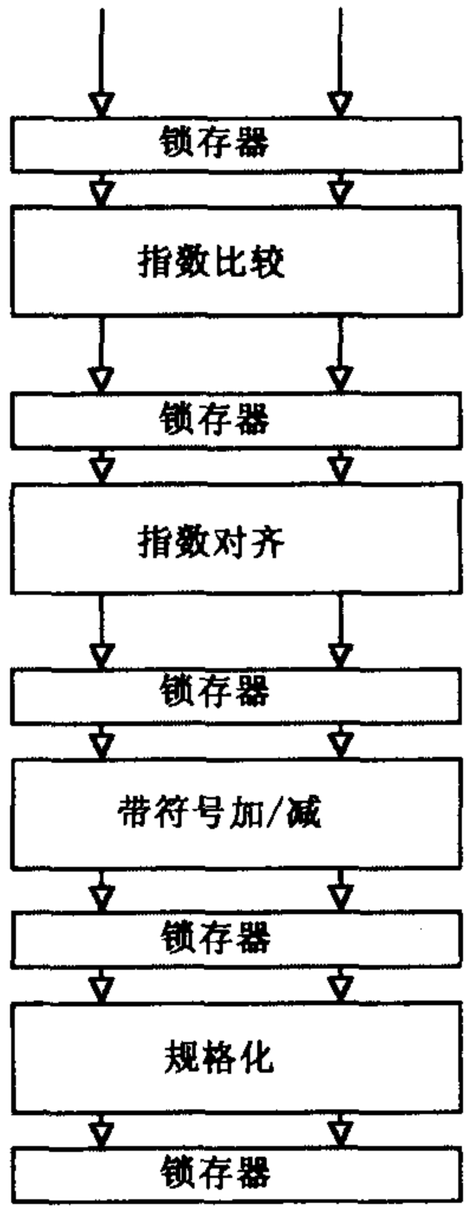


图 9.19　流水线浮点加法器的原理图

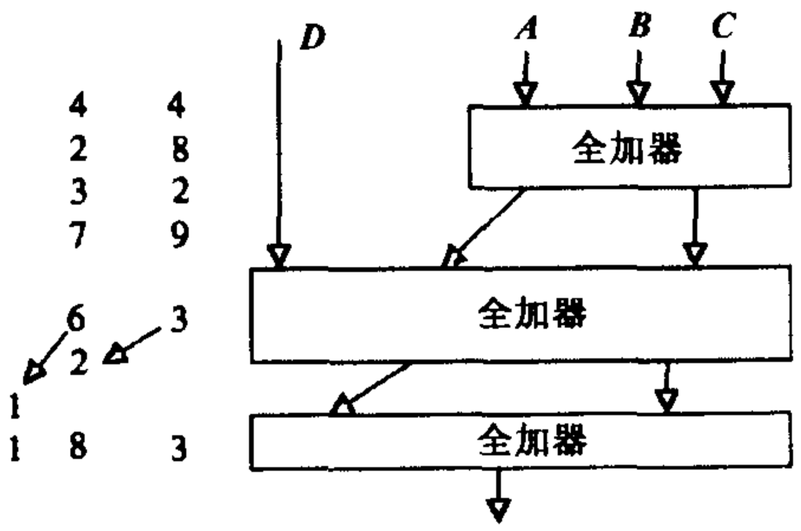


图 9.20　进位 – 保留加法

$$
\begin{aligned}
P &= A \times B \\
&= A \times (B_7 \times 2^7 + B_6 \times 2^6 + B_5 \times 2^5 + \cdots + B_0 \times 2^0) \\
&= \sum_{i=0}^{7} A \times B_i \times 2^i \\
&= \sum_{i=0}^{7} S_i \\
S_i &= A \times B_i \times 2^i \text{ 表示16位的部分乘积}
\end{aligned}
$$

图 9.21　基于进位 – 保留的两个 8 位操作数 A 和 B 的乘法

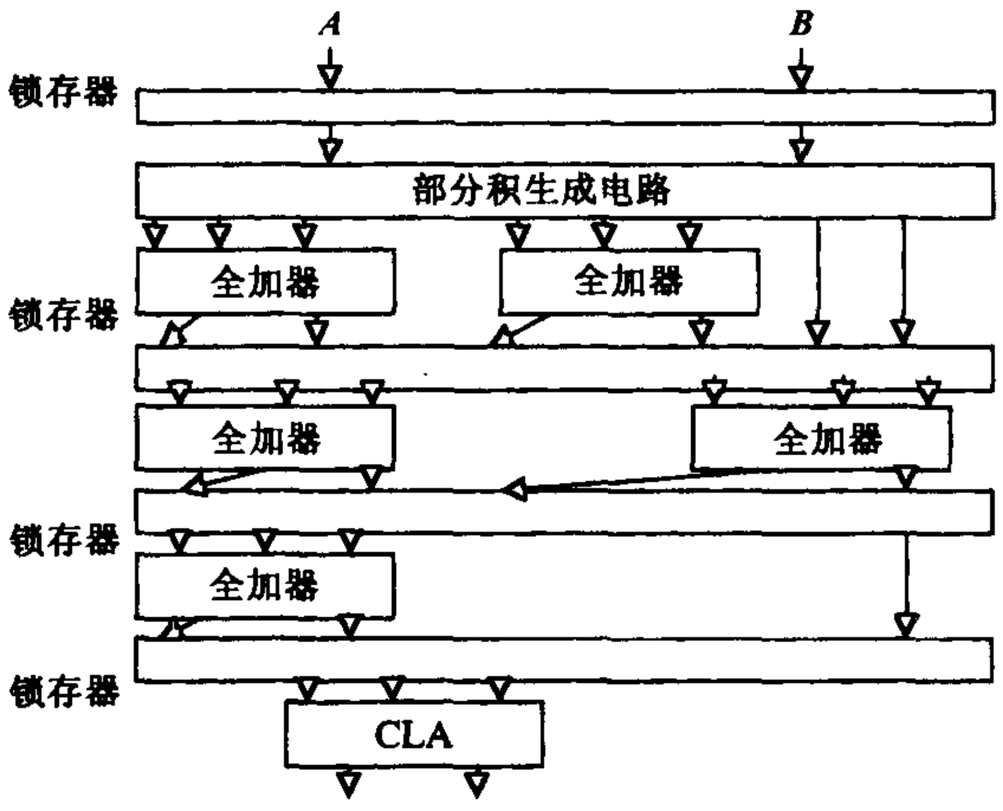


图 9.22　基于进位 – 保留加法的乘法模式

9.6 小结

在本章中,我们讨论了设计流水线体系结构所涉及的基本原理。我们的介绍从一些可以用来度量流水线性能优势的指标开始,然后我们对设计流水线体系结构所需要考虑的一些主要问题进行了概要的讨论。特别地,我们考虑了两个主要问题:指令相关和数据相关,还详细描述了这两个问题对流水线性能的影响;介绍了一些可以用来减少指令和数据相关影响的技术;提到了最新流水线体系结构的两个例子:ARM 11 微体系结构和 UltraSPARC III 处理器。最后我们介绍了一些可用来实现算术流水线体系结构的思想。

习题

1. 考虑在一个 5 级流水线机器上执行 500 条指令。给定一条指令是转移指令的概率是 $p=0.3$,计算由于使用流水线而产生的加速? 若加速要达到 4,问 p 必须是多少? 转移指令的数目应是多少? 若加速要达到 5,问 p 必须是多少? 假设每级均使用一个周期来执行任务。
2. 假设一台 RISC 机器,若不执行转移指令,则每个时钟周期执行 1 条指令。延迟转移的使用需 3 个时钟周期。考虑在这台机器上执行 1000 条指令,其中 30% 是转移指令的两种情况。第一种情况是使用了入门级编译器,它不能减少由于转移指令而浪费的额外时钟周期。第二种情况使用了较智能的编译器,它能利用 85% 的额外时钟周期。计算这两种情况下每周期执行指令的平均数。同时计算由于使用较智能的编译器而获得的性能改善的百分比。
3. 一个计算机系统有一个 4 级流水线,它由指令读取部件(F)、指令译码部件(D)、指令执行部件(E)和写部件(W)组成。计算流水线在执行由 20 条指令组成的代码段时的加速比 $P(4)$、吞吐量 $U(4)$和效率 $\zeta(4)$,其中转移发生顺序给定如下:I_3,I_9,I_{10},I_{15},I_{20}。假设当一条转移指令被读取时,流水线将停顿,直到知道下一条要读取的指令。计算使用二路交叉存取存储器来执行同样 20 条指令的时间,若 F,E,W 部件执行的功能需要使用存储器,问在这两种情况中每条指令的平均周期数是多少? 使用时空图来计算时间单位的数量。
4. 考虑对两个 16 位数字 M 和 Q 做整数乘法,得到结果 P。该操作可以表示为

$$P=\sum_{i=0}^{15} P_i$$

其中,$P_i = M \times Q_i \times 2^i$,表示 32 位的部分结果。

设计一个流水线部件来执行上述操作,使用最少数量的进位 - 保留加法器和 1 个先行进位加法器。同时再设计一条执行浮点加法/减法的流水线,并给出数值例子来证明你的设计。

5. 一个计算机系统有一个 3 级流水线,它由读取部件(F)、译码部件(D)、执行部件(E)组成。利用时空图计算使用二路交叉存取存储器来执行 20 条串行指令的时间,假设这 3 个部件需要同时使用存储器。

6. 使用时空图和分析模型,计算在 5 级流水线中执行 36 条指令的平均处理时间,若条件转移发生顺序如下:I_5,I_7,I_{10},I_{25},I_{27}。

7. 一个计算机系统有一个 5 级流水线,每级执行为一个周期。这 5 级分别是指令读取(IF)、指令译码(ID)、操作数读取(OF)、指令执行(IE)和操作数存储(OS)。考虑在该计算机上执行下列代码序列。

```
        Load −1, R1;          R1 ← −1;
        Load 5, R2;           R2 ← 5;
Again:  Sub  R2, 1, R2        R2 ← R2 − 1;
        Add  R1, R2, R3;      R3 ← R1 + R2;
        Bnn  Again;           branch to Again if result is Not Negative;
        Add  R4, R5, R6;      R6 ← R4 + R5;
        Add  R6, R4, R7;      R7 ← R4 + R6;
```

a. 分析上面代码片段的执行,计算不使用流水线执行时上面的代码需要多少周期? 假设每条指令实际需要 5 个周期来执行。

b. 若使用了上面描述的流水线,计算(使用 Gantt 图)执行上面代码需要多少周期? 假设没有转发硬件,且当读取转移指令时,流水线将停顿,直到计算出目标地址并做出转移决定。忽略任何的数据相关。

c. 其他都一样,若考虑数据相关,重做上面的(b)。

d. 计算(b)和(c)中由于使用流水线而提高的性能百分比。

参考文献和推荐读物

An Overview of UltraSPARC III Cu, Version 1.1 September 2003, A White Paper, Sun Microsystems, 1 – 18.

D. Brash, The ARM Architecture Version 6 (ARMv6), ARM Ltd., January 2002, White Paper, 1 – 15.

A. Clements, *The Principles of Computer Hardware*, 3rd ed., Oxford University Press, New York, 2000.

D. Cormie, The ARTM11 Microarchitecture, ARM Ltd., April 2002, A White Paper, 1 – 9.

K. Ebcioglu, J. Fritts and S. Kosonocky, An eight-issue tree VLIW processor for dynamic binary translation, *IEEE Proc.*, ICCD, (1998).

M. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design*, Jones and Bartlett Publisher, New York, 1995.

S. Furber, *ARM System-on-Chip Architecture*, Addison-Wesley, MA, USA, 2000.

G. Goldman and P. Tirumalai, UltraSPARC-III: The advancement of Ultra Computing, *Proc. IEEE COMPCON' 97*, p.417.

C. Hamacher, Z. Vranesic, S. Zaky, *Computer Organization*, 5th ed., McGraw-Hill, New York, 2002.

V. Heuring and H. Jordan, *Computer Systems Design and Architecture*, Prentice-Hall, New Jersey, 1997.

S. Hily and A. Seznec, Branch prediction and simultaneous multireading, *Proc. IEEE PACT' 96*, p.170.

J. Hoogerbrugge and L. Augusteijn, Instruction scheduling for TriMedia, *J. Instruction-Level Parallelism* (1999).

W.-M. Hwu, Introduction to predicted execution, *IEEE Comput.*, Vol. 31, No. 1, 49 – 50(1998).

D. Jaggar, ARM architecture and systems, *IEEE Micro*, 17(4), 9 – 11 (1997).

G. Lauthbatch and T. Horel, UltraSPARC-III: Designing third generation 64-bit performance, *IEEE Micro*, May-June, 73 – 85(1999).

R. Nair, Optimal 2-bit branch predictors, *IEEE Trans. Comput.*, 698, 1995.

R. Oehler and R. Groves, IBM RISC system/6000 processor architecture, *IBM J. Res. Dev.*, 34; 23 – 36 (1990).

B. Rau and J. Fisher, Instruction-level parallel processing: history, overview and perspective, *J. Supercomput.*, 7; 9 – 50(1993).

A. Scott, K. Burkhart, A. Kumar, R. Blumberg and G. Ranson, Four-way superscalar PA-RISC processors, *Hewlett-Packard J.*, August (1997).

J. Smith and G. Sohi, The microarchitecture of superscalar processors, *Proc. IEEE*, Vol. 83, No. 12, 1609 – 1624 (1995).

M. Tremblay, Increasing work, pushing the clock, *IEEE Comput.*, Vol. 31, No. 1, 40 – 41(1998).

B. Wilkinson, *Computer Architecture: Design and Performance*, 2nd ed., Prentice-Hall, Hert-fordshire, UK, 1996.

参考网站

http://www.ar.com

http://www.arm.com/support/White_Papers

http://www.sun.com/ultrasparc

第 10 章　精简指令集计算机

本章专门研究精简指令集计算机(RISC)。这种计算机体现了计算机体系结构领域中一个值得注意的转变。这种模式提倡简单而不是复杂。RISC 方法的有效性已被一些研究所证实。这些研究显示在一个典型的基准测试程序集中,赋值语句、条件转移和过程调用/返回在所有操作中占了超过 90%,而复杂操作(比如长除法)只占 2%。这些研究还表明在所有的操作中,过程调用/返回最耗时。基于这些结果,RISC 方法提倡使用一定的资源来增强体系结构,从而使得执行最常用和最耗时的指令变得最为有效。RISC 方法萌芽于 20 世纪 70 年代中期,而最初的实际机器是加州大学伯克利分校的 RISC-I 和斯坦福大学的 MIPS 机器,它们是在 20 世纪 80 年代中期制造的。当今,基于 RISC 的机器早已步入实际应用,而且它们可以被一些共同的属性所描述,比如简单而精练的指令集、固定的指令格式、一个机器周期一条指令、流水线的指令读取和执行部件、数量充足的通用寄存器(或采用优化编译器的代码生成)、载入/存储的存储器操作和硬连线的控制单元设计。本章中我们的介绍从对 RISC 体系结构的进化讨论开始。然后我们对一些导致采用 RISC 模式的性能研究进行简短的讨论。接着我们讨论 RISC 发展中的一个基本概念,重叠寄存器窗口。本章后半部分,我们提供了一些基于 RISC 体系结构的详细信息,包括加州大学伯克利分校的 RISC、斯坦福大学的 MIPS、康柏的 Alpha 和 SUN 的 UltraSparc。

10.1　RISC/CISC 进化周期

术语 RISC 表示精简指令集计算机。它最初是作为一个理论观念被引入的,表示计算机系统结构能够以每个时钟周期一条指令的快速度执行。RISC 在 20 世纪 70 年代中期作为一种理念开始,最终导致了第一台 RISC 机器的开发,即 IBM 801 小型机。RISC 理念的推出宣告了在计算机体系结构设计中一种新模式的开始。这种模式在计算机体系结构设计中提倡简单。它尤其倡导返回到基本层面而不是为高级语言提供额外的硬件支持。这种模式改变了与所谓的语义差别之间的关系。语义差别用来度量高级语言(HLL)与计算机体系结构所能提供的操作之间的不同。

通常认为语义差别越宽,不理想的后果数目也就越多。这些后果包括:(a)执行效率低;(b)过大的机器程序大小;(c)增加编译器的复杂性。因为这些预计的后果,计算机体系结构师们通常的解决办法是为更新的体系结构增加复杂的层次。这包括增加指令的数量和复杂度,增加寻址模式的数量。这种通过“增加更多复杂性”而产生的体系结构现在被认为是复杂指令集计算机(CISC)。但是情况很快就明朗了,复杂指令集有许多缺点。包括复杂的指令译码方式、控制部件的增大和逻辑延迟的增加。这些缺点促使一些计算机体系结构师们采用“少实际上是多”的原则。为此他们进行了复杂性对性能影响的许多研究,我们将在下面加以讨论。

10.2　RISC 设计原则

拥有最少数量指令的计算机的一个缺点是,即使实现一个简单的功能也需要执行大量的指令,从而导致速度劣势。另一方面,拥有大量指令的计算机有译码复杂的缺点,从而也导致速度

劣势。人们很自然地相信拥有经过精心挑选的精简指令集的计算机会在上述两个设计方案的选择中获得平衡。现在问题变成了精心挑选的精简指令集应该由什么组成？为了得到这个问题的答案，有必要对计算的许多方面进行深入的研究。这些方面应该包括：(a)运行典型(基准测试)程序时执行频率最高的操作；(b)最耗时的操作；(c)使用频率最高的操作数类型。

为了找出运行基准测试程序时操作的典型瓶颈，人们进行了一些早期的研究。各种操作的估计分布如表 10.1 所示。

表 10.1 操作的估计分布

操作	估计百分比
赋值语句	35
循环	5
过程调用	15
条件转移	40
无条件转移	3
其他	2

仔细查看操作执行的估计百分比就会发现赋值语句、条件转移和过程调用在整个操作的执行中占了 90%，而其他的操作，不管有多复杂，只占了剩下的 10%。

除了以上的发现，针对操作的时间 - 性能特性所进行的研究发现，在所有的操作中，过程调用/返回是最耗时的。关于典型计算中所使用的操作数类型，发现大部分的引用(不少于 60%)使用了简单的标量变量，且不少于 80%的标量是本地变量(对过程而言)。

以上关于典型程序行为的观察导致了下列结论：

1. 数据的简单移动(表现为赋值语句)——而不是复杂操作——数量更多，应该被优化；
2. 条件转移是主要的操作，因此对指令的顺序必须给予特殊的关注。特别是在已知流水线必须使用时更为如此；
3. 过程调用/返回是最耗时的操作，因此必须设计一种机制来实现调用过程和被调用过程之间参数的传递，而使所需执行的指令为最少；
4. 优化的一个主要候选方案是存取本地标量变量的机制。

上述结论导致了一个观点，即与其将指令集体系结构更靠近高级语言，不如对典型高级语言程序中最耗时的特性进行优化。这显然是提倡将体系结构简单化而不是复杂化。请记住，复杂的操作(比如长除法)在一个典型计算执行的操作中只占到很小一部分(少于 2%)。人们会问这样的问题：我们如何实现它？答案是：(a)将访问频率最高的操作数放置在 CPU 寄存器中；(b)最小化寄存器到存储器的操作。

以上两个原则可以通过使用下列机制加以实现：

1. 使用大量的寄存器来优化操作数引用并减少处理器与存储器间的通信；
2. 优化指令流水线的设计，以实现最少的编译器代码生成(见第 8 章)；
3. 使用简化的指令集，省去那些复杂、多余的代码。

下列两种方法被认为可以实现以上三种机制：

1. 软件方法。使用编译器以最大化寄存器的使用，将寄存器分配给那些在给定时间段内使用次数最多的变量(这是斯坦福大学的 MIPS 中所采用的设计思想)；

2. 硬件方法。使用充足的 CPU 寄存器使得更多的变量可以在更长时间段内被保存在寄存器中(这是加州大学伯克利分校的 RISC 机器中所采用的设计思想)。这种硬件方法需要使用一种新的寄存器组织,称为重叠寄存器窗口(overlapped register window)。这将在下面加以解释。

10.3　重叠寄存器窗口

使用重叠寄存器窗口的一个主要思想是最小化对存储器的访问。为了达到这个目的,需要大量的 CPU 寄存器。例如早先的 SPARC 机器(最早的 RISC 机器之一)上可用的通用 CPU 寄存器数目为 120 个。但是人们想要的是,在任意给定的时刻,只有这些寄存器的一个子集是可见的和可被寻址的,就如同它们是可用寄存器惟一集合。因此 CPU 寄存器被划分为多个小集合,每个被分配给不同的过程。一个过程调用将自动切换 CPU 去使用不同的大小固定的寄存器窗口。为了最小化调用过程和被调用过程间参数的实际移动,每个寄存器集合被划分为 3 个子集:参数寄存器、本地寄存器和临时寄存器。当一个过程调用启动后,一个新的重叠窗口将被建立,使得调用过程的临时寄存器和被调用过程的参数寄存器在物理上相同。这种重叠允许参数在过程间传递而无需数据的实际移动,如图 10.1 所示。

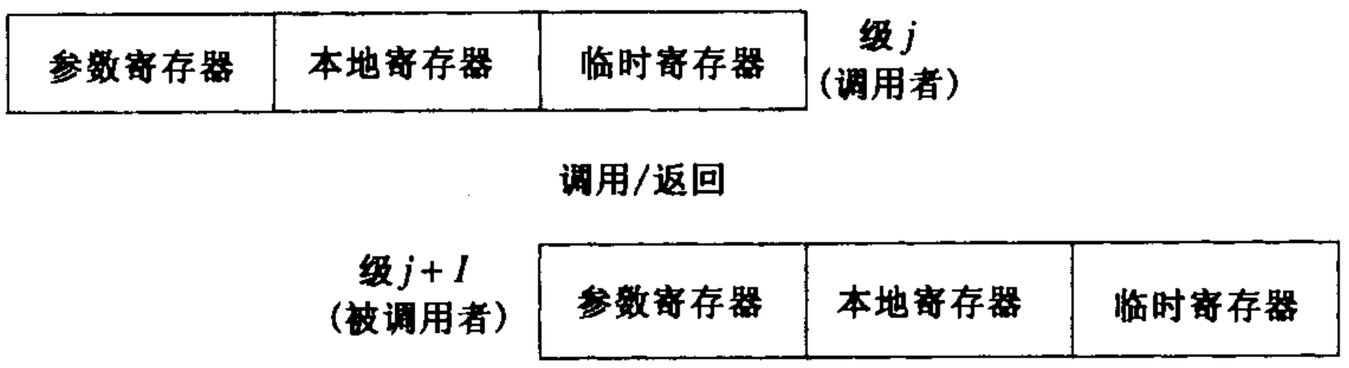


图 10.1　寄存器窗口重叠

此外,一组数量固定的 CPU 寄存器被用做全局寄存器,可供所有过程使用。例如在 SPARC 体系结构中,引用寄存器 0 到 7 将指向惟一的全局寄存器,而引用寄存器 8 到 31 表示寄存器在当前窗口。当前窗口通过通常所说的当前窗口指针(CWP)指向。当所有窗口填满后,寄存器窗口头尾相连,就像"循环缓冲器"一样动作。表 10.2 列出了一些体系结构的窗口数量和大小。

表 10.2　不同寄存器窗口特征

体系结构	窗口数目	窗口大小
加州大学伯克利分校的 RISC-I	8	16
金字塔(Pyramids)	16	32
SPARC	32	32

值得注意的是,在 1985 年进行的一项研究,其目的是找出在加州大学伯克利分校的 RISC 机器上使用寄存器窗口对性能的影响。这项研究中使用了这种机器的两个版本。第一个设计时使用了寄存器窗口,而第二个是假想的加州大学伯克利分校 RISC 机器实现,不使用寄存器窗口。研究结果表明使用寄存器窗口将存储器通信减少 2 到 4 倍(依赖于特定的基准测试程序)。

10.4　RISC 对 CISC

选择 RISC 或 CISC 完全基于计算机设计者必须要考虑的因素。这些因素包括大小、复杂性和速度。RISC 体系结构必须要执行更多指令来完成与 CISC 体系结构相同的功能。为了弥补这

个缺陷,RISC 体系结构必须将由于不需要进行复杂指令译码而省下来的芯片空间,用于提供大量的 CPU 寄存器、附加执行部件和指令高速缓存。使用这些资源导致处理器与存储器间的通信减少。另一方面,拥有更丰富更复杂指令的 CISC 体系结构与 RISC 相比只需更少的指令。但是,CISC 体系结构需要复杂的译码模式,从而导致逻辑延迟。因此有理由相信 RISC 和 CISC 模式的主要不同是在于所采用的折中不同设计因素的策略。

很难相信一个能让 RISC 体系结构提高性能的想法会在 CISC 体系结构上失败,反之亦然。例如 RISC 开发中一个主要方法是使用优化编译器来减少硬件的复杂度,并优化 CPU 寄存器的使用。同样的想法也可以应用在 CISC 编译器中。增加 CPU 寄存器的数量也能够显著提高 CISC 机器的性能。这可以作为找不到一台纯 RISC(或者 CISC)商业机器的原因。经常可看到 RISC 的机器使用复杂的浮点指令(具体看下一节的 SPARC 体系结构),同样也可以看到 CISC 机器使用 RISC 寄存器窗口。事实上有些研究表明 CISC 机器,比如摩托罗拉 680xx 在使用寄存器窗口后使存储器通信减少了 2 到 4 倍。这是 RISC 体系结构,比如加州大学伯克利分校的 RISC,使用寄存器窗口后所能达到的相同倍数。

但应注意大多数处理器生产商(除了 Intel 和它的合作者)选择了 RISC 处理器。计算机系统生产商,比如 Sun 公司在它们的产品中使用了 RISC 处理器。但是为了与个人计算机市场兼容,这些公司还在生产基于 CISC 的产品。

表 10.3 和表 10.4 分别针对性能和特征在 RISC 和 CISC 样机间进行了有限的比较。

表 10.3 RISC 与 CISC 性能对比

应用	MIPS CPI (RISC)	VAX CPI (CISC)	CPI 比	指令数比
Spice 2G6	1.80	8.02	4.44	2.48
Matrix300	3.06	13.81	4.51	2.37
Nasa 7	3.01	14.95	4.97	2.10
Espresso	1.06	5.40	5.09	1.70

表 10.4 RISC 与 CISC 特征对比

特征	VAX-11 (CISC)	加州大学伯克利分校的 RISC-1 (RISC)
指令数	303	31
指令大小(位)	16 ~ 456	32
寻址模式	22	3
通用寄存器数	16	138

一些商用 RISC 和 CISC 机器间更加详细的对比,如表 10.5 所示。

表 10.5 一些 RISC 和 CISC 机器特性的总结

	摩托罗拉 88110	Alpha AXP 21264	奔腾	Power PC 601
公司	摩托罗拉	康柏(DEC)	Intel	IBM
体系结构	RISC	RISC	CISC	RISC
# 寄存器(I)	32	80	64	32
# 高速缓存 I/D	8/8 KB	64/64 KB	8/8 KB	32
# 寄存器(GP/FP)	32/32	31/31	8/8	32/32
# 每周期的指令数	2	1	2	3
# 流水线(I/FP)	NS	4/2	5/8	4/6
多处理支持	否	是	是	是

值得注意的是,在 RISC 机器中存在下列观察到的共同特征集合:

1. 定长的指令;
2. 有限的指令数(128 或更少);
3. 有限的简单寻址方式(最少是两个:变址和 PC 相关);
4. 所有运算在寄存器中执行;没有存储器操作;
5. 只有两个存储器操作:载入和存储;
6. 流水线式的指令执行;
7. 大量通用寄存器或使用先进编译器技术优化寄存器的使用;
8. 一个时钟周期一条指令;
9. 硬连线的控制器设计,而不用微程序设计。

10.5　先驱 RISC 机器

在本节中,我们对两个先驱(由大学开发的)RISC 机器的主要体系结构特性进行简单描述。第一台机器是加州大学伯克利分校的 RISC,第二台是斯坦福大学的 MIPS 机器。对这些机器进行说明是为了展示原来 RISC 机器的样子,同时也是为了使读者了解 RISC 机器的开发自诞生以来的进展。

10.5.1　加州大学伯克利分校的 RISC

有两种加州大学伯克利分校的 RISC 机器:RISC-I 和 RISC-II。除非另有说明,我们接下来讨论的 RISC 是指 RISC-I。RISC 是 32 位载入/存储的体系机构,共有 138 个 32 位的寄存器 $R_0 \sim R_{137}$ 可供用户使用。前面 10 个寄存器 $R_0 \sim R_9$ 是全局寄存器(为所有进程可见)。寄存器 R_0 用来合成机器不直接提供的寻址方式和操作。寄存器 $R_{10} \sim R_{137}$ 被划分成重叠寄存器窗口模式,使得有 32 个寄存器在任何时刻可见。称为当前窗口指针(CWP)的一个 5 位变量,用来指向当前寄存器集。

所有 RISC 指令都占用一个全字(32 位)。RISC 指令集分为 4 类,即 ALU(总共 12 条指令)、载入/存储(总共 16 条指令)、转移与调用(总共 7 条指令)和特殊指令(总共 4 条指令)。一些 RISC 指令的例子如下:

1. ALU:ADD R_s,S,R_d;$R_d \leftarrow R_s + S$
2. 载入/存储:LDXW(R_x)S,R_d;$R_d \leftarrow M[R_x + S]$
3. 转移与调用:JMPX COND,(R_x)S;PC$\leftarrow R_x$ + S;COND 是一个条件
4. 特殊指令:GETPSW R_d; $R_d \leftarrow$PSW

所有的算术和逻辑指令有 3 个操作数,且有相同的格式 Destination := source1 op source2(目的操作数 = :源操作数 1　操作符　源操作数 2),如图 10.2 所示。Load(载入)和 Store(存储)指令可以使用如图 10.2 所示格式中的一种,其中 DST(目的)是用来载入或是存储的寄存器。指令低端的 19 位用来决定有效地址。

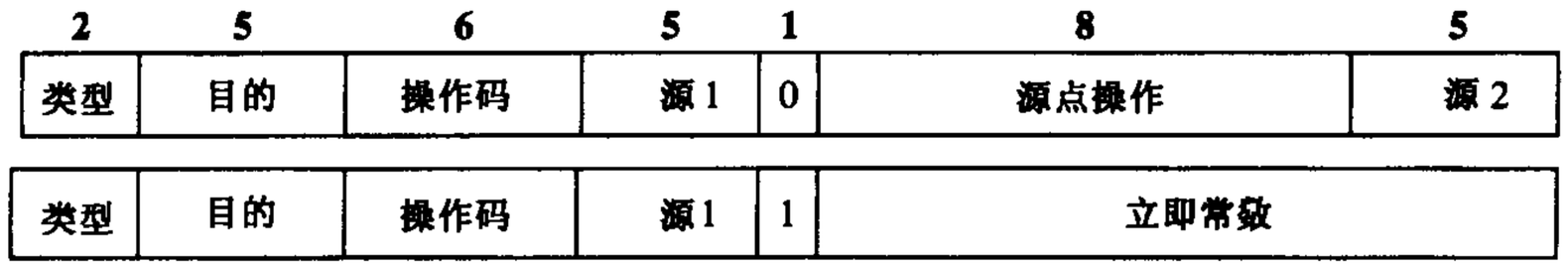


图 10.2　RISC 中使用的 3 个操作数指令格式

指令载入和存储8,16,32和64位数到32位寄存器中。调用过程有两种方法。CALL指令使用了30位的程序计数器相对偏移,如图10.3所示。

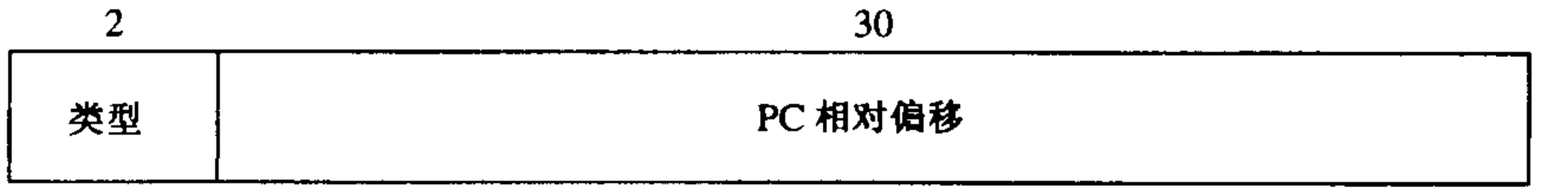


图10.3 RISC的过程调用指令

JMP指令可以使用算术和逻辑操作的任一指令格式,允许将返回地址放置到任意寄存器中。

RISC使用三地址指令格式,也有一些可用的两地址和单地址的指令。只有两种寻址方式,它们是变址方式和程序计数器PC相关方式。变址方式能用来合成三种其他方式。它们是基址-绝对(直接)、寄存器间接和线性字节数组变址模式。RISC使用静态两级流水线:取指令和执行。

浮点计算部件(FPU)包括了32个32位寄存器,用来存储32个单精度(32位)浮点操作数、16个双精度(64位)浮点操作数或8个扩展精度(128位)浮点操作数。FPU可以使用算术指令的第一种格式执行大约20条浮点指令,大部分使用单精度、双精度或是扩展精度。除了载入和存储FPU寄存器的指令,CPU也能测试FPU寄存器,也可以根据结果的条件进行转移。RISC使用一个传统的MMU(存储器管理部件)以支持分页的32位单一地址空间。RISC的四总线组成如图10.4所示。

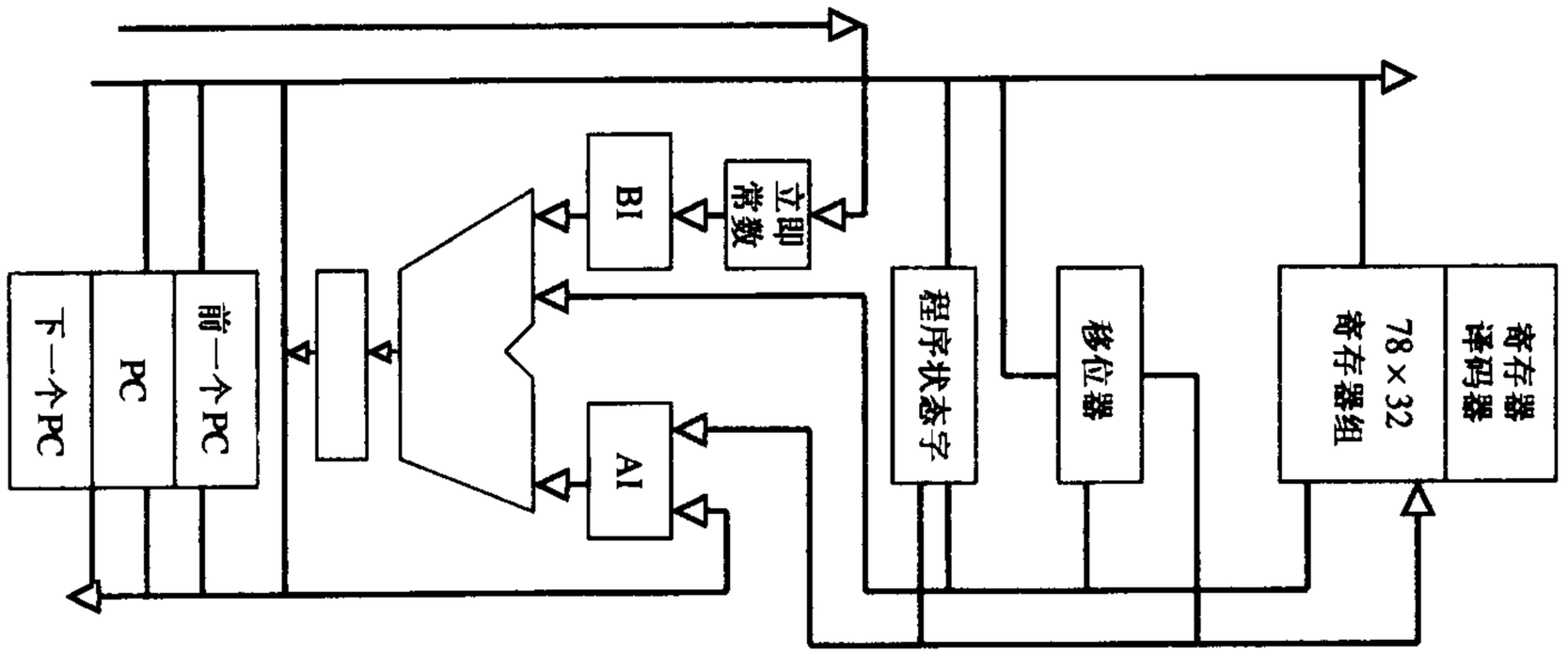


图10.4 RISC的四总线组成

10.5.2 斯坦福大学的MIPS

MIPS是32位流水线式载入/存储机器。它使用五级流水线,包括指令读取(IF)、指令译码(ID)、操作数译码(OD)、操作数存储/执行(OS/EX)和操作数读取(OF)。前三级分别执行指令读取、指令译码和操作数译码。OS/EX级在保存指令时发送操作数到存储器,或在执行指令时使用ALU。OF级在载入指令时接收操作数。MIPS使用了一种称为流水线互锁(pipeline interlock)的机制,用来阻塞一条指令继续执行直到所需操作数可用。

不像加州大学伯克利分校的RISC,MIPS只有一组16个32位的通用寄存器。MIPS编译器在编译当前程序时选择最优方式优化寄存器的使用。除了16个通用寄存器以外,MIPS提供了4个附加的寄存器,用来保存4个以前的PC(程序计数器)值(以支持遇到错误时的回退和重启)。第5个寄存器用来保存将来的PC值(以支持转移指令)。

MIPS中使用了4种寻址方式。它们是立即数、变址、带偏移基址和移位基址。MIPS中区分

了 4 类指令。它们是 ALU, Load/Store, Control 和特殊指令。ALU 指令共 13 种,包括了所有寄存器对寄存器的 2 或 3 操作数格式,如图 10.5 所示。一共提供了 10 种 16 位或 32 位的 Load/Store 指令。在 32 位的情况中,变址地址可以通过加一个带符号的 16 位常量到使用图 10.5 中第二种格式的寄存器中来使用。一共提供了 6 种控制流指令,包括 jumps, relative jumps 和 compare 指令。只提供了 2 个特殊的流指令,它们支持过程和中断的链接。一些 MIPS 指令的例子如下:

1. ALU: Add src_1, src_2, dst; dst←src_1 + src_2
2. Load/Store: Ld[src_1 + src_2], dst; dst←M[src_1 + src_2]
3. Control: Jmp dst; PC←dst
4. 特殊功能指令: SavePC A; M[A]←PC

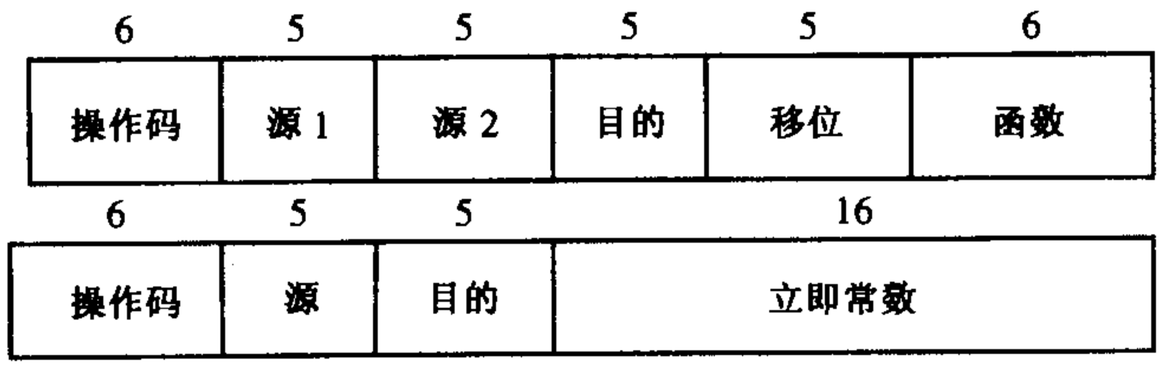


图 10.5　MIPS 中使用的三操作数指令格式

MIPS 没有对浮点操作提供直接的支持。浮点操作由一个专用的协处理器完成。有趣的是,一些非 RISC 指令比如 MULT 和 DIV 也包含了进来,它们使用专用的功能部件。两个寄存器的内容能够进行乘法或相除,64 位结果将保存在两个专用的寄存器 LO 和 HI 中。

过程调用可以通过 JUMP 指令来实现,如图 10.6 所示。该指令使用了 26 位跳转目标地址。

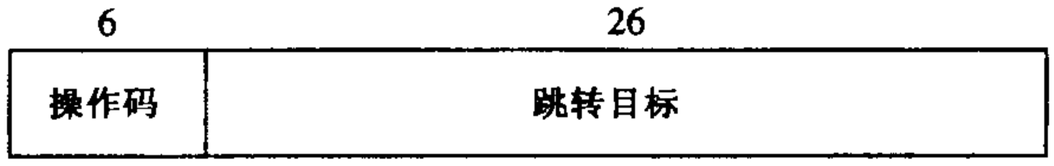


图 10.6　MIPS 中使用的跳转指令格式

MIPS 的虚拟地址有 32 位,因此允许增长到 4 G 字的虚拟地址空间。一个虚拟地址分成一个 20 位的虚拟页面号和一个 12 位的页面偏移量。MIPS 的实际实现受封装约束的限制,只允许有 24 个地址引脚。也就是说,真正的物理地址是 2^{24} = 16 M 字(每字 32 位)。MIPS 提供了对芯片外地址翻译 TLB 的支持。其结构如图 10.7 所示。

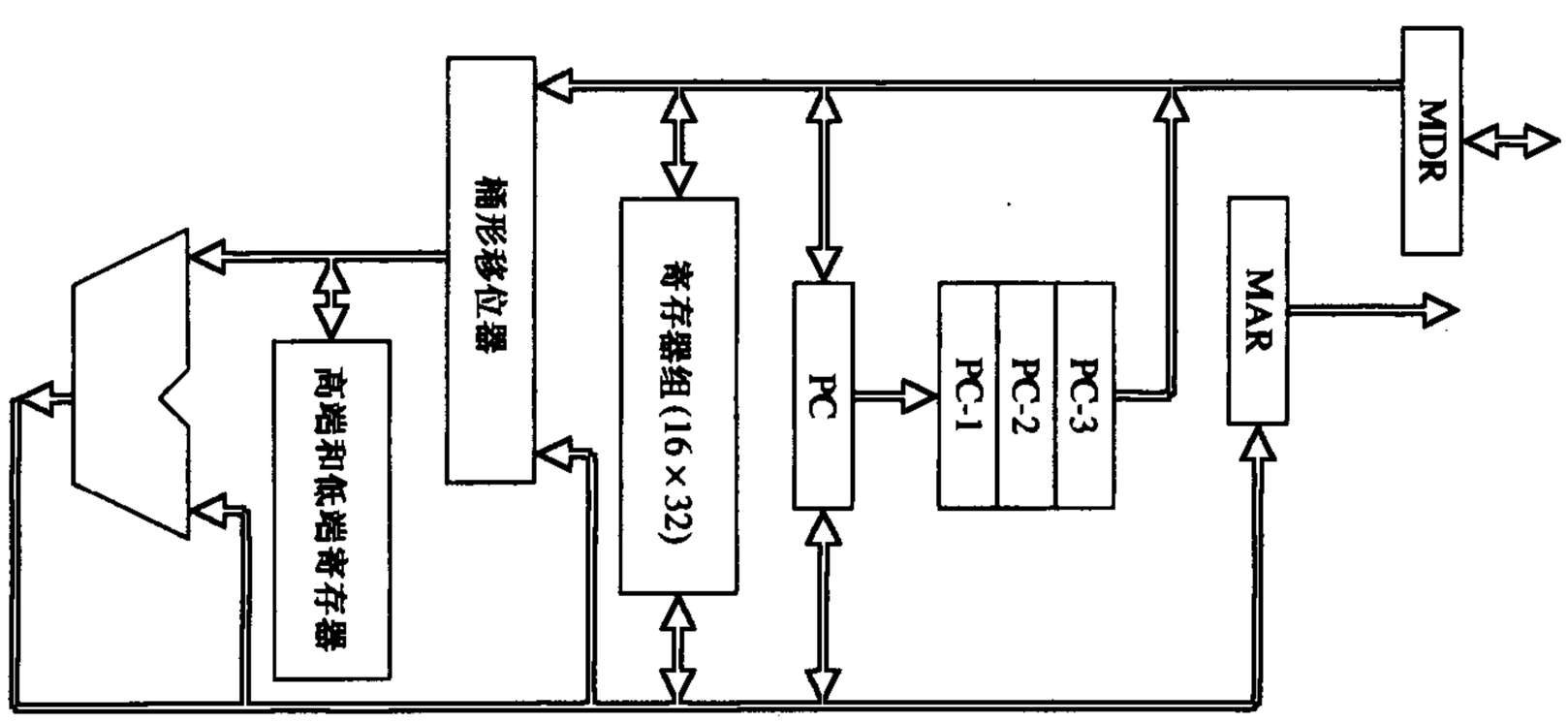


图 10.7　MIPS 的结构

10.6 高级 RISC 机器实例

在本节中,我们将介绍两个具有代表性的高级 RISC 机器。在这里我们强调的是流水线特征和所使用的转移处理机制。

10.6.1 康柏 Alpha 21264

Alpha 21264(EV6)是康柏[①] 的第三代 RISC 超标量处理器。它是完全的 64 位处理器。21264 有一个 80 项的整数寄存器组和一个 72 项的浮点寄存器组。它使用了 2 级高速缓存。L1 数据和指令高速缓存各有 64 KB。它们以二路组相联的方式组成。L2 数据高速缓存大小可以在 1 MB 到 16 MB 之间(为指令和数据所共享),通过直接映射组成。块大小是 64 位。数据高速缓存能够接受整数执行流水线每个周期两个载入或存储的任意组合。这相当于有一个 64 KB 芯片内数据缓存每周期发送 16 B,从而加倍了处理器的时钟速度。21264 存储器系统能够支持到 32 个并发载入、32 个并发存储和 8 个并发(64 B)高速缓存块填充和 8 个高速缓冲缺失。它有一个 64 KB 二路组相联(既是数据又是指令)高速缓存。它还可以支持多至 2 个的乱序操作,如图 10.8 所示。

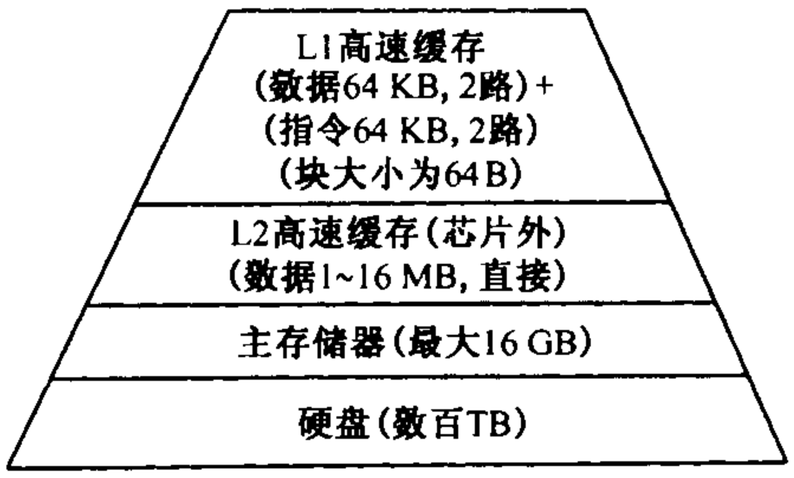


图 10.8 21264 的存储器层次

10.6.2 Alpha 21264 流水线

Alpha 21264 的指令流水线如图 10.9 所示,由 7 级组成,分别是:读取(Fetch)、槽分配(Slot Assignment)、重命名(Rename)、发射(Issue)、寄存器读入(Register Read)、执行(Execute)和存储器级(Memory)。

图 10.9 21264 的指令流水线

读取阶段每周期能够读取和执行多至 4 条的指令。读取阶段的框图如图 10.10 所示。这一级使用了惟一的“块和组”(block and set)预测技术。根据这一技术,接下来 4 条指令的位置和所属的组(共有 2 个组)可以被预测。“块和组”预测技术结合了直接映射高速缓存的速度优势和二路组相联高速缓存的低缺失率。这个技术达到了 85%以上的命中率。错误预测的损失是一个单独的周期。21264 使用了 2 级猜测转移预测方案,它以对转移所显示的本地和全局关联为依据。本地关联利用了转移过去的行为。另一方面,全局关联利用所有以前转移的过去行为。在 21264 中使用的混合本地/全局预测使转移行为模式与本地转移历史(也就是在惟一的程序计数

① 康柏现已被惠普并购,它的前身是 DEC——编者注。

器 PC 位置上一条转移指令的执行)和全局转移历史(也就是之前所有转移的执行)相关联。该方案动态地在本地和全局转移历史之间进行选择,如图 10.11 所示。

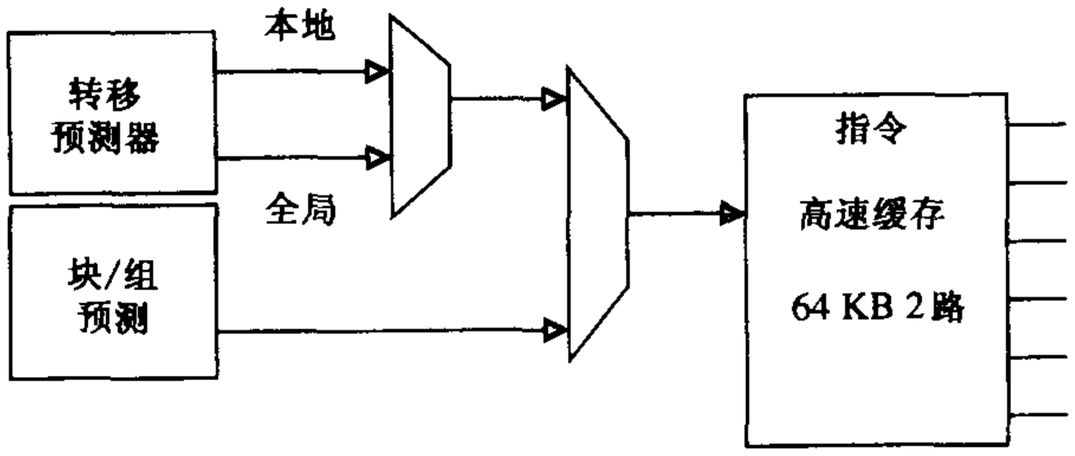


图 10.10　21264 的读取级

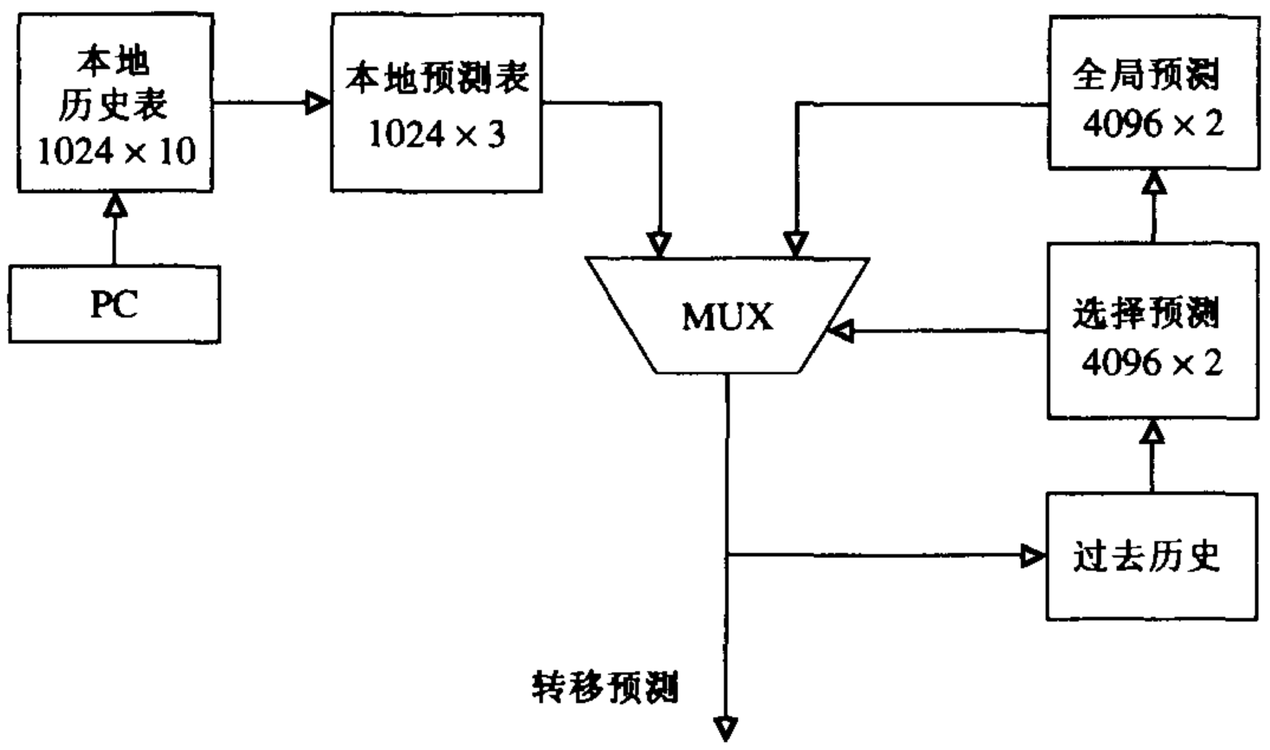


图 10.11　21264 的选择转移预测器

本地转移预测器有两张表。第一张是 1024 × 10 的本地历史表,表中每项保存了最近执行的选择转移的 10 位本地历史。本地历史表通过指令地址(使用程序计数器 PC)进行索引。第二张是 1024 × 3 的本地预测表,表中每项保存了用来预测转移结果的 3 位饱和计数器。当退出转移后,21264 使用正确的转移方向和引用计数器更新本地历史表。这种方法提高了正确预测的可能性,称为“预测器训练”。

全局转移预测器有一张 4096 × 2 的全局预测表,表中每项保存了 2 位饱和计数器。它用来追踪最近 12 次转移的全局历史。全局转移预测表通过一个 4096 × 2 的选择预测表加以索引。当退出转移后,21264 更新索引的全局预测计数器,以提高正确预测的可能性。

本地预测在一个给定转移中交替地出现转移/不转移序列的情况中特别有用。在这种情况中,转移的本地历史将最终解析出一个 10 个交替为 0 和 1 的模式,以表明转移成功或失败的交替出现。由于转移将执行多次,将使对应于本地历史值的预测计数器达到饱和,从而使预测变得正确。

全局预测在当转移的结果能从以前转移的方向推理出来时特别有用。比如考虑两个转移重复调用的情况。当第一个转移检查一个值等于 1001 成功时,第二个转移检查同样的值为奇数也必然成功。全局历史预测器能从这两个转移的重复调用中学习这种模式。

4096 × 2 选择预测器是一张每项保存了 2 位饱和计数器、用来实现选择(比赛)方案的预测表。当本地和全局预测器的预测结果不同时,21264 会更新选中的选择预测项以支持正确的预测器。在退出转移时 21264 更新选择预测表。

槽赋值级(S#2)简单地将指令分配给那些与整数和浮点数队列关联的槽。

21264 中的乱序发射逻辑每个周期接受 4 条已读取的指令,然后重命名并重新映射寄存器(以避免不必要的寄存器相关),并将指令放入队列,直到操作数和/或功能部件可用。它每周期最多可动态地发射 6 条指令,4 条整数和 2 条浮点指令。寄存器重命名表示将指令虚拟寄存器映射到内部物理寄存器。一共有 31 个整数和 31 个浮点寄存器对用户可见。这些寄存器在执行过程中被重命名为内部寄存器。只有当指令完成(退出)后这些内部寄存器才会被重命名回可见的寄存器。寄存器重命名消除了先写后写和先读后写的数据相关。但是,它保留了所有正确计算所必须的先写后读的相关。

待执行指令的列表是由乱序队列逻辑维护的。在每个周期,整数和浮点数队列选择那些已经准备好执行的指令。这个选择是基于一个重命名寄存器的记分牌做出的。这个记分牌通过跟踪单周期、多周期、可变周期指令的过程,保存重命名寄存器的状态。当功能部件或载入数据结果可用后,记分牌部件通知所有队列中的指令关于所需寄存器值的可用情况。每个队列每个周期选择等待时间最久的数据就绪和功能部件就绪的指令加以执行。21264 整数队列静态地将指令分配到 4 个管道中的 2 个,上面的或是下面的,如图 10.12 所示。

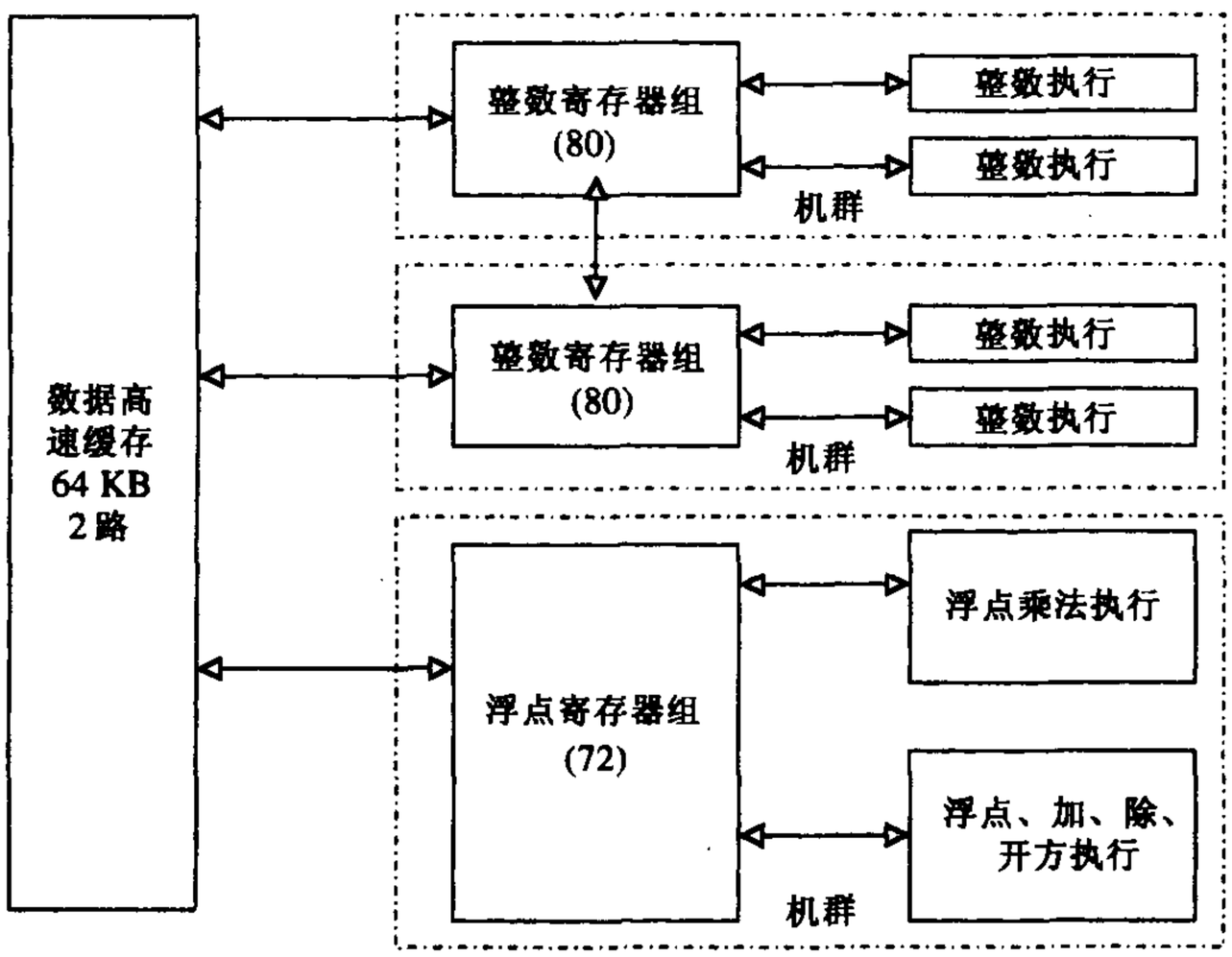


图 10.12 21264 的执行管道

Alpha 21264 有 4 条整数流水线和 2 条浮点数流水线。这就允许处理器在同一周期动态发射 6 条指令。发射(或者队列)级保存了一张详细目录,通过它处理器能动态选择发射最多为 6 条的指令。有一个 20 项的整数发射队列和一个 15 项的浮点数发射队列。指令发射的重新排序在发射级进行。

21264 使用 2 个整数文件,每个 80 项,用来保存寄存器内容的副本。两个管道访问一个单独文件从而形成一个簇(cluster)。两个簇形成一个 4 路整数指令执行。结果从一个簇广播到其他簇。指令由整数发射队列动态选择送到一个给定的流水线中执行。一条指令可以启发式地加以选择以在同一个簇上执行产生结果。21264 有一个 72 项的浮点寄存器组。这个组和两条指令执行流水线一起形成一个簇。图 10.12 显示了寄存器读入/执行流水线。

最后要注意我们应该指出的是 21264 在第 2 级高速缓存中使用了一个写无效的高速缓存一

致性机制,以提供对共享存储器的多重处理的支持。它支持以下高速缓存状态:修改的(modified)、拥有的(owned)、共享的(snared)、独占的(exclusive)和无效的(invalid)。

10.6.3　SUN 的 UltraSPARC III

UltraSPARCIII 是一个高性能超标量 RISC 处理器,它实现了 64 位 SPARC-V9 RISC 体系结构。SPARC III 处理器有不少种实现,包括 UltraSPARC IIIi 和 UltraSPARC III Cu。本节中我们所涉及的内容原则上与任何具体实现无关。但在合适时,我们会涉及具体的实现。

UltraSPARC III 是第三代的 64 位 SPARC RISC 微处理器。它支持 64 位虚拟地址空间和 43 位物理地址空间。UltraSPARC III 采用了多级高速缓存体系结构。例如,UltraSPARC IIIi(和 UltraSPARC III Cu)体系结构有一个 32 KB 4 路组相联 L1 指令高速缓存、一个 64 KB 4 路组相联 L1 数据高速缓存、一个 2 KB 预取高速缓存和一个 2 KB 写高速缓存。UltraSPARC IIIi 支持 1MB 4 路组相联指令/数据合一的片内 L2 高速缓存。在 UltraSPARC IIIi 中高速缓存块大小为 64 字节。而 UltraSPARC III Cu 体系结构支持一个 1 MB、4 MB 或 8 MB 2 路组相联指令/数据合一的外部高速缓存。UltraSPARC III Cu 中的高速缓存块大小从 64 字节(为 1 MB 高速缓存使用)到 512 字节(为 8 MB 高速缓存使用)不等,如图 10.13 所示。

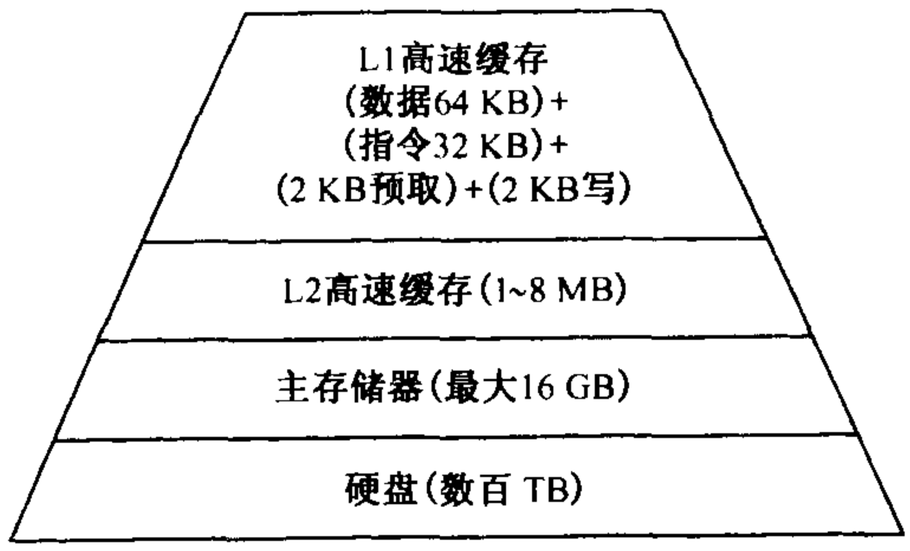


图 10.13　UltraSPARC III 的存储器层次

UltraSPARC III 使用了可被并行访问的 2 个指令 TLB 和 3 个数据 TLB。2 个指令 TLB 以 16 项全相联的方式组成,用来保存 8 KB,64 KB,512 KB 和 4MB 页面大小的项。一个 128 项 2 路组相联的 TLB 只能使用 8 KB 的页面大小。3 个数据 TLB 以 16 项全相联的方式组成,用来保存 8 KB,64 KB,512 KB 和 4 MB 页面大小的项。2 个 512 项 2 路组相联 TLB 能够被编程,用来在给定段时间内保存任意一个页面大小。UltraSPARC III 使用了一种写分配、回写的高速缓存写策略。

UltraSPARC III 的流水线已在第 9 章介绍过,这里不再赘述。

最后,值得注意的是 UltraSPARC III 是设计来支持 1 到 4 路多重处理。为了这个目的,它使用了 JBus,它能支持一个小规模的多处理机系统。JBus 能够为网络和嵌入式系统应用提供所需的高带宽,通过 JBus 处理器能够与一致共享的总线相连,而无需粘合逻辑,如图 10.14 所示。

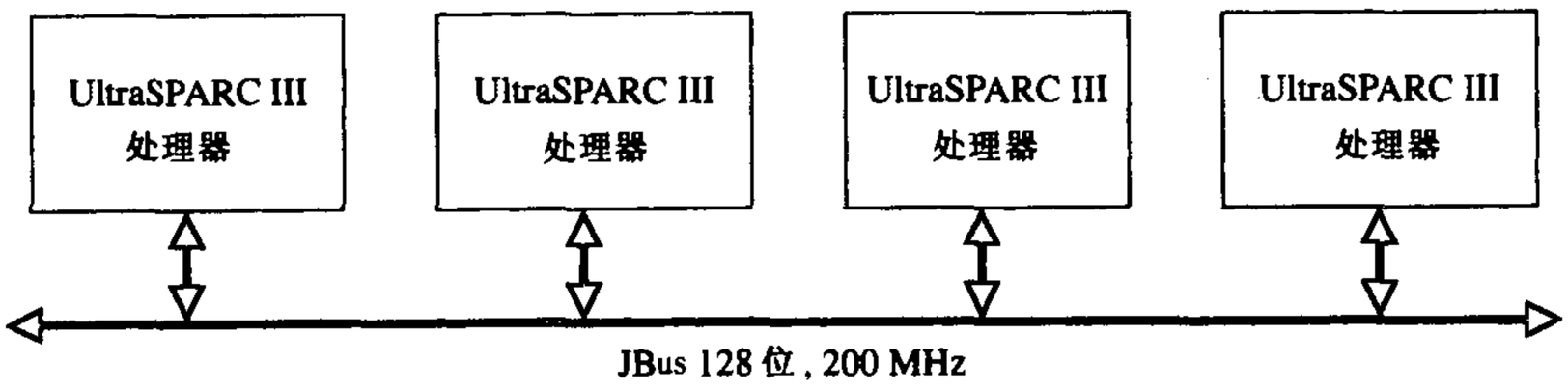


图 10.14　4 路 UltraSPARC III 多处理机系统的配置

10.7 小结

RISC 体系结构节省了 CISC 体系结构用来译码和执行复杂指令所需的额外芯片空间。省下来的芯片空间则用来提供一个芯片内的高速缓存,它能够用来减少处理器与存储器之间的指令通信。大多数 RISC 设计所共享的通用特征是:有限而简单的指令集、大量的通用寄存器和/或使用编译器技术以优化寄存器的使用和指令流水线。RISC 设计思想精髓之一是将最频繁的访问操作数放在寄存器里,并最小化寄存器 - 存储器操作。这可以用两种方法中的一种来实现:软件方法,使用编译器最大化寄存器的使用,将寄存器分配给那些在一段给定时间内使用次数最多的变量(这是斯坦福大学的 MIPS 中采用的设计思想);硬件方法,使用更多的寄存器使更多的变量可以在更长的时间内保存在寄存器中(这是加州大学伯克利分校的 RISC 机器中采用的设计思想)。寄存器窗口是多个寄存器的小集合,每个被分配给不同的过程。一个过程调用自动地切换 CPU 去使用不同的大小固定的寄存器窗口,而不是在调用时将寄存器保存在存储器中。任何时候,只有一个寄存器窗口是可见和可寻址的,就像它是寄存器的惟一集合。窗口重叠需要某一层的临时寄存器在物理上就是下一层的参数寄存器。这种重叠允许在不实际移动数据的情况下传递参数。

值得注意的是,将处理器分类为纯 RISC 或是纯 CISC 将变得越来越不确切,或者说不恰当。人们真正关心的是使用了一个特定设计风格的要素之后,可以获得多大的性能提高。大部分现代处理器使用了经过精心计算的两种设计风格要素的组合。使用每种设计风格的某些要素的决定性因素是基于所需增加性能和预计增加成本之间的折中而获得的。一些被分类为 RISC 的处理器却采用了一些 CISC 的特性,比如整数/浮点除法指令。同样,一些被分类为 CISC 的处理器也采用了一些 RISC 的特性,比如流水线。

习题

1. 哪些是用来构造一台 RISC 机器的主要原则?
2. 对比 RISC 机器中用来减少存储器操作的两种方法(软件和硬件)。
3. 用例子来解释寄存器窗口和窗口重叠的概念。提出一种不同的方法以得到相同的结果,就像使用存器窗口和窗口重叠实现一样。
4. 为了回答这个问题,你需要挑选一台近期的 RISC 处理器作为研究对象。提交一份总结主要流水线所使用的特性和主要 RISC 特性的短报告(长度不少于 5 页但不多于 10 页)。所介绍内容的程度应该适合一位大四的本科生。请确保你的介绍准确而简洁。有需要时可以使用简单的例子,提供精确和有意义的图片和表格。你的介绍需要覆盖流水线处理器和 RISC 机器的所有方面。
5. RISC 与 CISC 之间的争论从来没有停息过。假设你作为 RISC 方法的拥护者,写出不少于一页对 CISC 方法的批评,指出它的缺点,同时说明 RISC 方法的优点。你可以使用实例机器的性能作为你支持 RISC 设计思想的依据。
6. 重复习题 5,假设你是 CISC 设计思想(方法)的拥护者。

参考文献和推荐读物

An Overview of UltraSPARC III Cu, Version 1.1 September 2003, A White Paper, Sun Microsystems, 1 – 18.

R. Colewell *et al*. Computers, complexity, and controversy, *IEEE Comput*., 18(9), 8 – 19(1985).

Z. Cvetanovic and R. Kessler, Performance analysis of the Alpha 21264-based Compaq E40 system, *Proc. 27th Annual International Symposium on Computer Architecture*, Vancou-ver, British Columbia, Canada, 192 – 202.

Exploring Alpha Power for Technical Computing. A Compaq Report on Compag High Performance Technical Computing, November 1999, pp. 1 – 28.

G. Goldman and P. Tirumalai, UltraSPARC – III: The advancement of ultra computing, *Proc. IEEE COMPCON' 97*, p.417.

J. Hennessy, VLSI processor architecture, *IEEE Trans. Comput*., C-33(11), 1221 – 1246 (1984).

J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann: San Mateo, San Francisco, CA, 1996.

R. Kessler, The Alpha 21264 Microprocessor, *IEEE Micro*, Vol. 19, issue 2, 24 – 36 (1999).

R. Kessler, E. McLellan and D. Webb, The Alpha 21264 Microprocessor Architecture, International Conference on Computer Design, Oct. 88, pp. 96 – 102.

G. Lauthbatch and T. Horel, UltraSPARC-III: Designing third generation 64-bit performance, *IEEE Micro*., 73 – 85 (1999).

R. Nair, Optimal 2-bit branch predictors, *IEEE Trans. Comput*., 698 (1995).

D. Patterson, Reduced instruction set computers, *Commun. ACM*, 28(1), 8 – 21 (1985).

D. Patterson and R. Ditzel, The case for the reduced instruction set computer, *Comput. Architecture News*, 8(6), 25 – 33 (1980).

D. Patterson and C. Sequin, A VLSI RISC, *IEEE Comput*., 15(9), 8 – 21 (1982).

G. Radin, The 801 minicomputer, *IBM J. Res. Develop*., 27(3), 237 – 246 (1983).

R. Sherburne, M. Katevenis, D. Patterson and C. Sequin, A 32-bit NMOS processor with a large register file, *IEEE J. Solid-State Circuits*, Sc-19(5), 682-689 (1984).

A. Tanenbaum, *Structured Computer Organization*, 3rd ed., Prentice-Hall: Englewood Cliffs, New Jersey.

参考网站

http://www.sun.com/processors/UltraSPARC-IIIi

http://www.sun.com/processors/whitepapers

第 11 章　多处理机系统导论

在讨论了单处理器系统设计和分析的基本问题并指出单流机器的主要限制之后，在本章中我们将探讨多处理器问题。在多处理机系统中，一些(两个或多个)处理器以某种方式连接，从而允许它们同时执行一个任务。使用多处理器的主要出发点是希望通过连接许多较小的现成计算机就可创建功能强大的计算机。显然一台多处理机系统将能达到比最快的单处理器系统还要快的速度。另外，由许多单处理器组成的多处理机系统比起构造一台高性能单处理器系统来更为经济有效。由 n 个处理器构成多处理机系统的另一个好处是，若有一个处理器失效，则其余未失效的 $n-1$ 个处理器仍能继续提供性能降级的服务。本章的内容将从所使用的通用概念和术语这一小节开始，然后讨论为连接多处理器所使用的不同拓扑，接着介绍和分析计算机体系结构的不同分类方案。此后介绍的是基于拓扑的互连网络的分类。对 MIMD(多指令多数据)的两种存储器组成方案也做了介绍。本章以关于多处理器分析和性能指标的内容而告结束。感兴趣的读者可参阅我们所写的“先进计算机体系结构与并行处理”① 一书中第 2 章和第 3 章中有关多处理器更详细的讨论。

11.1　引言

由两个或多个处理器组成的多处理机系统需要连接多个处理器以允许这些处理器能同时(并行)执行给定计算任务。并行处理已经是构造高性能计算机系统的一种很有效的方法。为了有效使用这些处理器，不可避免地要满足两个基本要求。这两个要求是：(1)在执行一个给定任务时处理器间有较低的通信开销；(2)任务具有内在的并行度。

在多处理器网络中存在许多通信类型。广义上它们可按通信模型(CM)或物理连接(PC)加以分类。按照 CM，网络可进一步分为多处理器(单地址空间或共享存储器计算)或多计算机(多地址空间或消息传递计算)。按照 PC，网络可进一步分为基于总线或基于网络的多处理机系统。表 11.1 中总结了这些系统的典型规模。

表 11.1　某些多处理机系统的典型规模

类	子类	处理器数
通信模型	多处理器	2 ~ 256
	多计算机	8 ~ 256
物理连接	基于总线	2 ~ 32
	基于网络	8 ~ 256

用来连接处理器的互连网络对多处理机系统的组成和性能有很大影响。可以使用单共享总线作为多处理器的互连网络，也可以使用交叉开关作为互连网络。虽然前一种技术代表了一种简单、易扩展的拓扑，但它在性能上受到一定的限制，因为在任何给定时间它不允许有多于一个的处理器/存储器传送。而交叉开关则可提供所有处理器/存储器的不同连接，但却比较昂贵。

① 原书名为“Advanced Computer Architecture and Parallel Processing”，中译本由电子工业出版社出版，与本书同期——编者注。

多级互连网络(MIN)力求在单共享总线系统的局限和基于交叉开关系统的昂贵两者之间达成一种平衡。在 MIN 中,同时可建立多于一个的处理器/存储器连接,而 MIN 的成本又低于交叉开关,特别当使用大量处理器和/或存储器时。使用多总线连接多处理器和多存储器模块也已被提议作为单总线和昂贵交叉开关两者之间的折中。图 11.1 说明了上述 4 种互连网络的类型。感兴趣的读者可参阅我们写的"先进计算机体系结构与并行处理"一书。

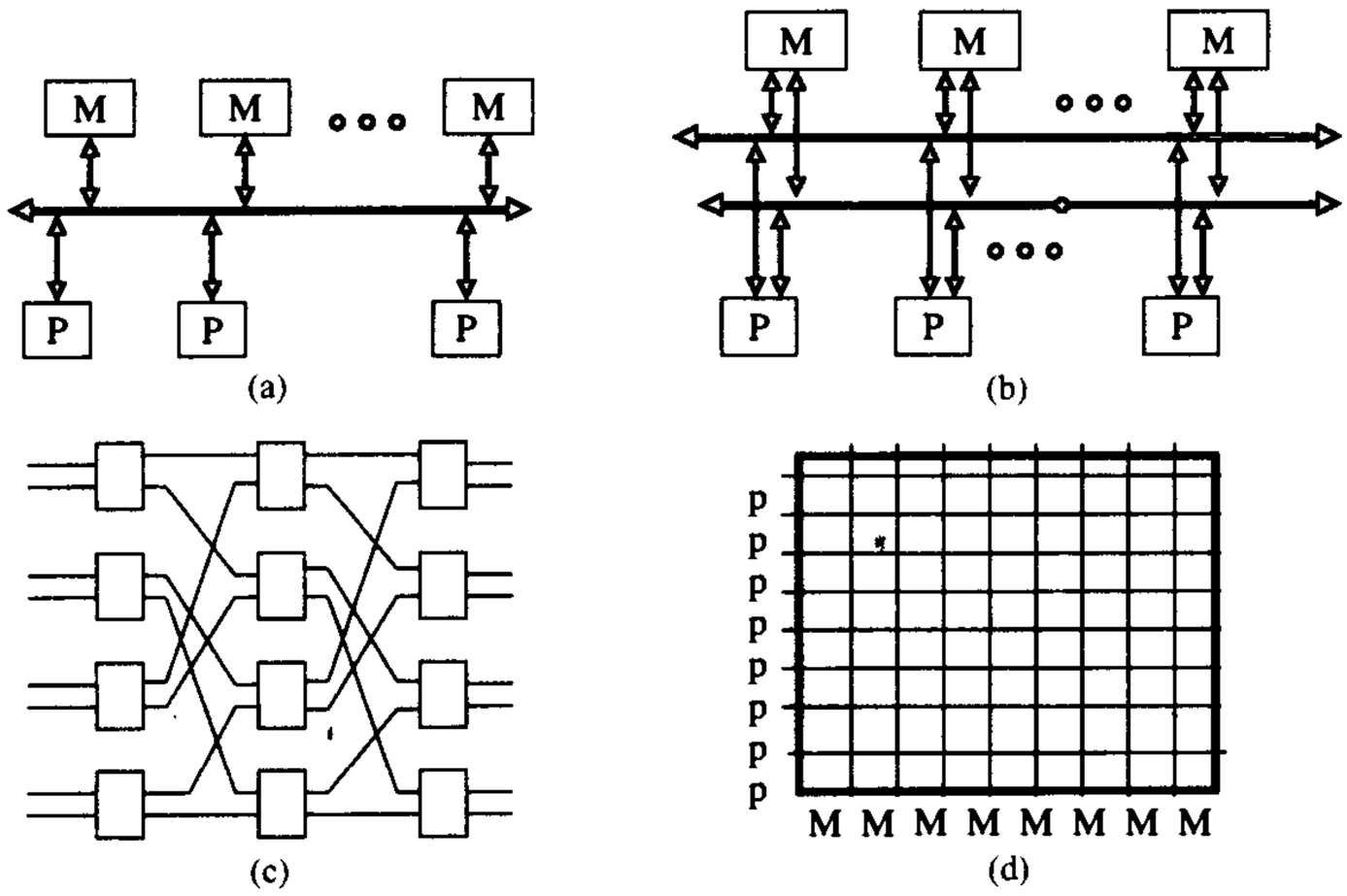


图 11.1　多处理器互连网络的 4 种类型

(a)单总线系统;(b)多总线系统;(c)多级互连网络;(d)交叉开关系统

11.2　计算机体系结构的分类

分类意味着对众多对象的按类整理,每一类有一些共同特征,其中某些特征间还存在某种关系。就这一点而论,计算机体系结构的一种分类方案的目的在于将那些有共同特征的体系结构分为一类,而不同的类代表不同的体系结构组。其次,计算机体系结构的一种分类方案应提供对给定体系结构进行信息整理和性能预测的基础。

对计算机体系结构进行分类存在两种主要的方案。第一种是基于体系结构的外部(形态)特征,第二种基于体系结构的进化特征。第一种方案着重的是体系结构已形成的形式,而第二种方案着重的是一种体系结构从其前驱者演化而来的方法和它对后继者的影响。形态分类为预测性能提供了基础,而演化分类为更好地理解体系结构提供了基础。考察一种分类方案在何种程度上适合它所说明的对象,就可评估该分类方案的优缺点。

过去的 30 年中已提出了许多分类方案,其中包括弗林分类(1966),Kuck 分类(1978),Hwang 和 Briggs 分类(1984),Erlangen 分类(1981),Giloi 分类(1983),Skillicorn 分类(1988)以及 Bell 分类(1992)。下面将对其中的一些进行简短的介绍。

11.2.1　弗林分类

弗林(Flynn)的分类方案基于识别计算机中存在的两个正交流,即指令流和数据流。指令流定义为由计算机所完成的指令序列,而数据流定义为存储器和处理部件间的数据通信。按照弗林分类,指令流和数据流可以是单的也可以是多的。从而形成 4 种不同类型的计算机体系结构:

1. 单指令单数据流(SISD);

2. 单指令多数据流(SIMD);

3. 多指令单数据流(MISD);

4. 多指令多数据流(MIMD)。

如图 11.2 所示为弗林分类的正交流结构。

如图 11.3 所示为由弗林分类而生成的 4 种体系结构的图解。表 11.2 列出了分属于 4 类中每一类的商品化机器。

		数据流	
		单	多
指令流	单	SISD	SIMD
	多	MISD	MIMD

图 11.2 弗林分类

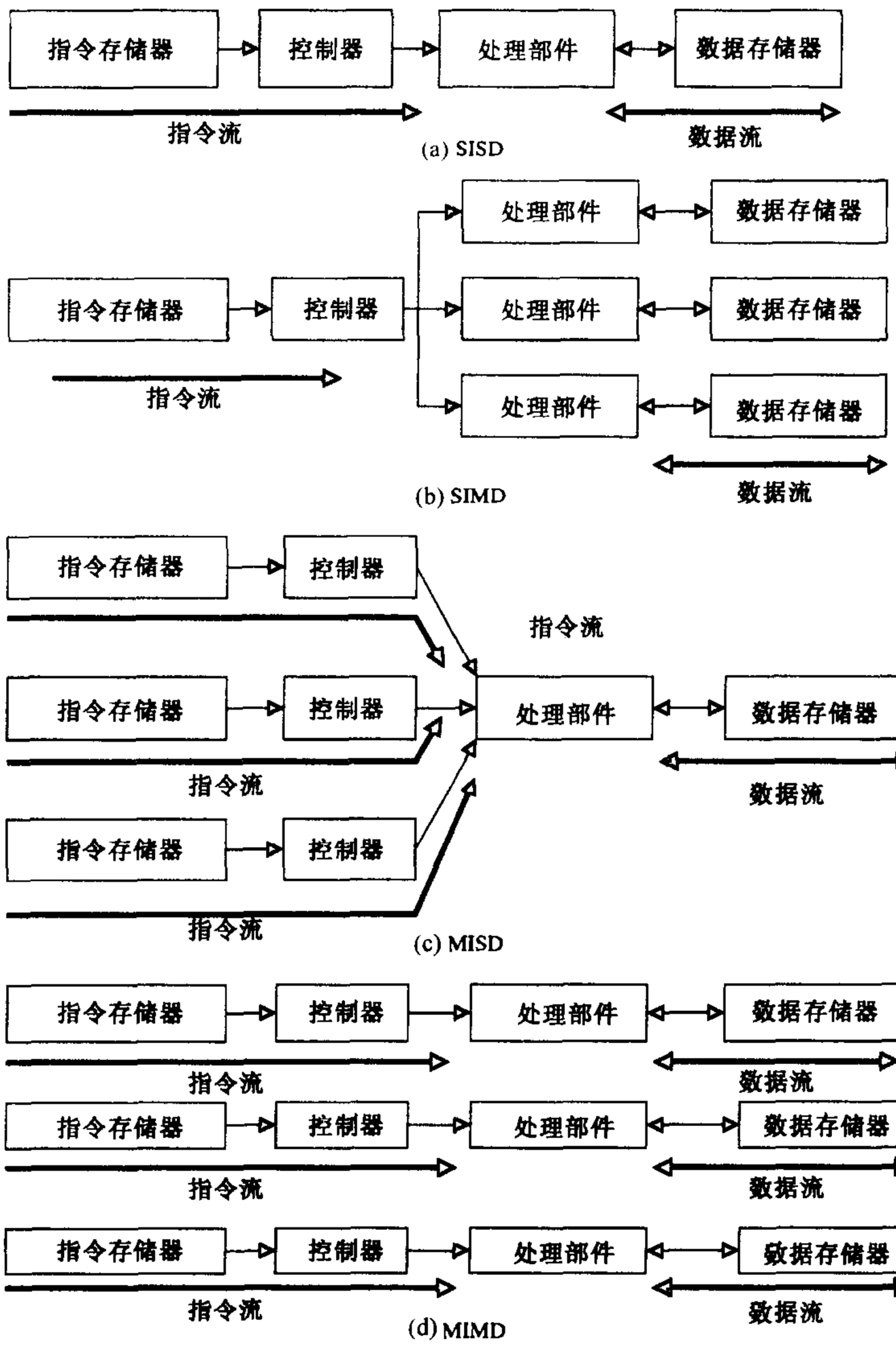


图 11.3 由弗林分类而得到的 4 种体系结构:(a)SISD;(b)SIMD;(c)MISD;(d)MIMD

表 11.2　机器实例和它们的弗林分类

分类类型	机器实例
SISD	IBM 704, VAX 11/780, CRAY-1
SIMD	ILLIAC-IV, MPP, CM-2, STARAN
MISD	参见下面对弗林分类的评论 5
MIMD	Cm*, CRAY XMP, IBM 370/168M

对弗林分类的评论

1. 弗林分类是这些分类中最先引入的一个,因此必定启发了后来的分类;
2. 该分类有助于分类过去已有的和那些后来提出的体系结构。例如,该分类中的 SIMD 和 MIMD 机器模型必定启发了体系结构学者提出这些新的机器模型;
3. 弗林分类强调存储器 – 处理器级的体系结构关系。其他的体系结构级全部忽略;
4. 弗林分类强调体系结构的外部(形态)特征。不含有属于同类体系结构革新关系的信息;
5. 由于它的纯抽象性,没有实际可行的由该分类所提出的 MISD 模型的例子(至少到目前为止是如此)。但应注意某些体系结构学者认为流水线机器(以及脉动阵列计算机)可作为 MISD 的例子;
6. 弗林分类中所缺少的一个重要方面是机器性能问题。虽然该分类给人们的印象是 SIMD 和 MIMD 会胜过它们的 SISD 和 MISD 对手,但它没有任何关于 SIMD 和 MIMD 机器相对性能的信息。

11.2.2　Kuck 分类方案

被认为是通用分类方案的弗林分类方案已被许多计算机体系结构学者做了扩展。这些扩展中的一个是 D. J. Kuck 于 1978 年提出的。Kuck 的分类中将指令流进一步扩展成单(标量和数组)流和多(标量和数组)流,并将数据流称为执行流(execution stream),且也被扩展成包含单(标量和数组)流和多(标量和数组)流。这些流的组合形成了总共 16 类的体系结构,如表 11.3 所示。

表 11.3　由 Kuck 分类而得到的 16 种体系结构分类

指令流	执行流			
	单		多	
	标量	数组	标量	数组
单				
标量	单处理器	单处理器	SIMD	
数组		ILLIAC-IV		
多				
标量		NYU Ultracomputer	Cray X MP	
数组				

我们认为弗林和 Kuck 这两个分类覆盖了整个体系结构的空间。与弗林分类着重的是在指令集级对体系结构的描述不同,Kuck 分类着重的是在硬件级对体系结构的描述。

11.2.3　Hwang 和 Briggs 分类方案

Hwang 和 Briggs 对分类的主要新贡献是他们引入了类的概念。这是对弗林分类的进一步精化。例如,按照 Hwang 和 Briggs,SISD 类被进一步细分为两个子类:单功能部件 SISD(SISD-S)和多

功能部件 SISD(SISD-M)。MIMD 类被进一步细分为松耦合 MIMD(MIMD-L)和紧耦合 MIMD(MIMD-T)。SIMD 类被进一步细分为字片(word-sliced)处理 SIMD(SIMD-W)和位片(bit-sliced)处理 SIMD(SIMD-B)。所以 Hwang 和 Briggs 的分类为机器的分类层次加了一级,从而使给定机器首先分类为 SISD,SIMD,MIMD,然后进一步加以分类。

按照 Hwang 和 Briggs 的分类方案,预测一个 SISD-M 对同一任务将比一个 SISD-S 性能更好总是正确的。但是,有关 SIMD-W 和 SIMD-B 这样的预测就存有疑虑。例如,已经证明用最大潜在并行度来度量性能时,ILLIAC-IV 机器(SIMD-W)要比 MPP 机器(SIMD-B)差。对 Hwang 和 Briggs 分类的最后评论是,共享存储器系统(可参阅我们写的"先进计算机体系结构与并行处理"一书的第 4 章)能自然映射到 MIMD-T 类型,而非共享存储器系统则能自然映射到 MIMD-L 类型。

11.2.4 Erlangen 分类方案

将 Erlangen 分类方案的最简单形式与弗林分类方案相比,该方案只是增加了一层计算机内部结构的细节。特别是该分类方案除了控制(CNTL)和处理(ALU)部件外,考虑了第 3 个称为基本逻辑部件(ELU)的子部件,它被用来描述一个给定计算机体系结构的特征。ELU 代表在 ALU 中完成位级处理所需的电路。一个体系结构特征可用一个三元系统(k, d, w)加以描述,其中 k = CNTL 数,d = 与一个控制部件相关的 ALU 数,以及 w = 每个 ALU 中的 ELU 数(一个单数据字的宽度)。例如,在它的一个模型 ILLIAC-IV 中,由 64 个 64 位的 ALU 构成一个网格阵列,并由 Burroughs 公司的 B6700 计算机加以控制。按照 Erlangen 分类,ILLIAC-IV 模型应描述为(1,64,64)。

假设流水线处理在硬件的所有三级上都存在,则该分类就含有另外 3 个参数。它们是 w' = 每个 ALU 中的流水级数,d' = 每个 ALU 中的功能部件数,以及 k' = 形成控制部件的 ELU 数。对 3 个硬件处理级中的每一级赋予所期待的多部件性质,就可用一个更为通用的 6 元系统来描述一个体系结构如下:$(k \times k', d \times d', w \times w')$。图 11.4 为 Erlangen 分类系统的示意图。

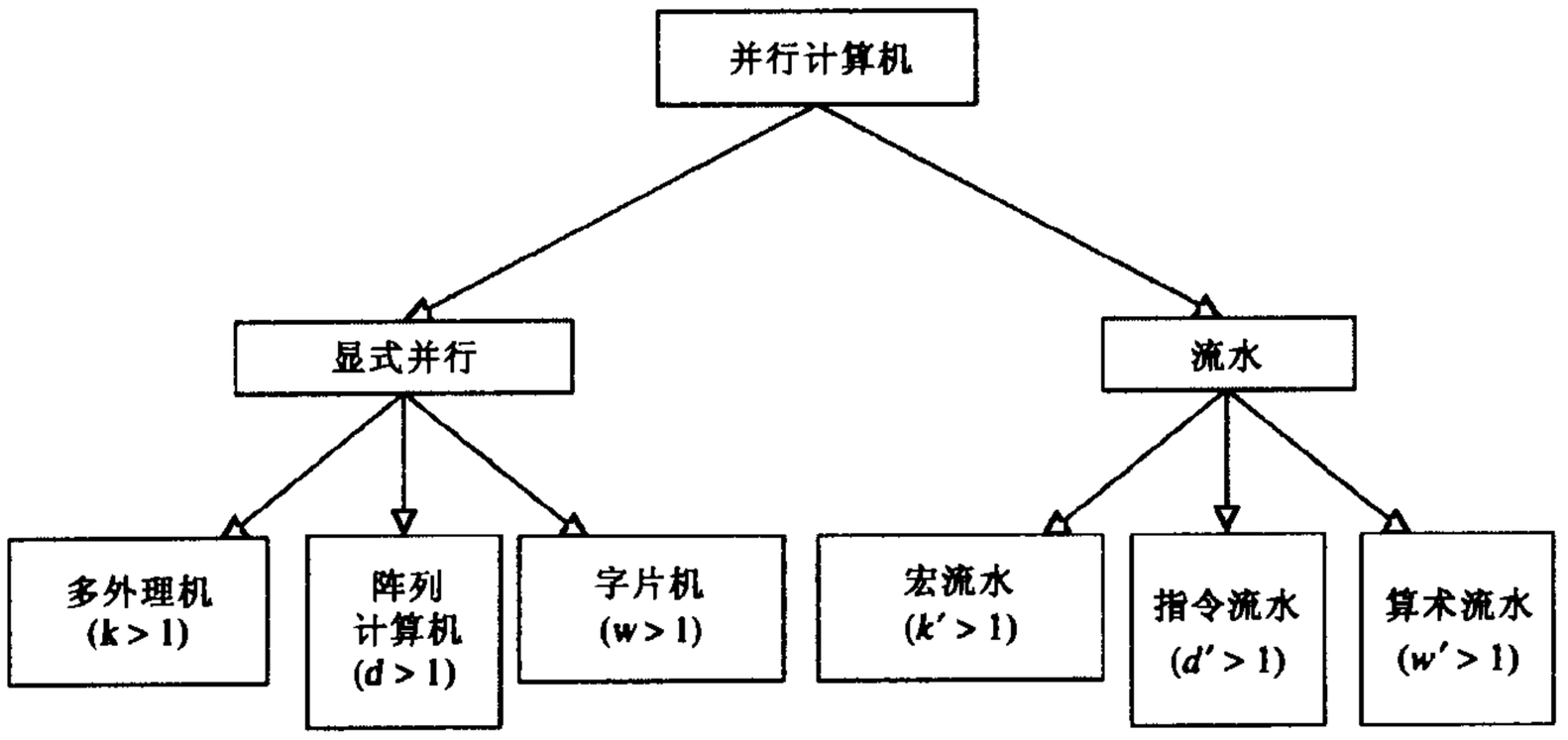


图 11.4 Erlangen 分类方案

借助使用两个附加的算子,即用 × 表示的 AND(与)算子和用 V 表示的 ALTERNATIVE(选择)算子,Erlangen 分类系统就可描述更为复杂的系统。例如,一个由两个计算子部件所组成的体系结构,每个子部件各自有一个 6 元组$(k_0 \times k_0', d_0 \times d_0', w_0 \times w_0')$和$(k_1 \times k_1', d_1 \times d_1', w_1 \times w_1')$,就可以用两者的子部件以$(k_0 \times k_0', d_0 \times d_0', w_0 \times w_0') \times (k_1 \times k_1', d_1 \times d_1', w_1 \times w_1')$加以描述,而若一个体系结构可以表示成两个子部件中的任一个,则可用$(k_0 \times k_0', d_0 \times d_0', w_0 \times w_0') \vee (k_1 \times k_1', d_1 \times d_1', w_1 \times w_1')$加以描述。

例如,一个稍后设计的 ILLIAC-IV 由两个 DEC PDP-10 作为前端控制器,其中数据每次只能由一个 PDP-10 接收。这个 ILLIAC-IV 版本可以描述成(2,1,36) × (1,64,64)。现在由于该 ILLIAC-IV 也能以半字方式工作,此时就有 128 个 32 位处理器而不是 64 个 64 位处理器,因而 ILLIAC-IV 的总体特性就可由(2,1,36) × [(1,64,64)V (1,128,32)]给定。

由此可见,该分类方案可视为是一个层次分类,它更着重处理硬件的内部结构。但该分类不提供计算机体系结构分类和/或组合的基础,特别是该分类忽略了不同部件间的互连。

11.2.5　Skillicorn 分类方案

由于弗林分类的固有性质,对于具有类似体系结构特征但却有不同功能性的计算机系统,弗林分类将把它们组合成一类。这也成为了 1988 年提出 Skillicorn 分类的主要灵感。按照这一分类,一个抽象的冯·诺依曼机器将模型化成如图 11.5 所示的那样。从图中可见,除了指令处理器(IP)和数据处理器(DP)之外,该抽象模型含有两个存储器子部件,指令存储器(IM)和数据存储器(DM)。在利用这一分类方案时,将考虑以下可能的互连关系:(IP - DP),(IP - IM),(DP - DM)和(DP - IP)。互连方案考虑了数据处理器、数据存储器、指令处理器和指令存储器之间的连接类型和数目。它们之间可能没有连接,可能存在一对多连接,也可能存在多对多连接。表 11.4 说明了可由该分类识别的不同连接方式。

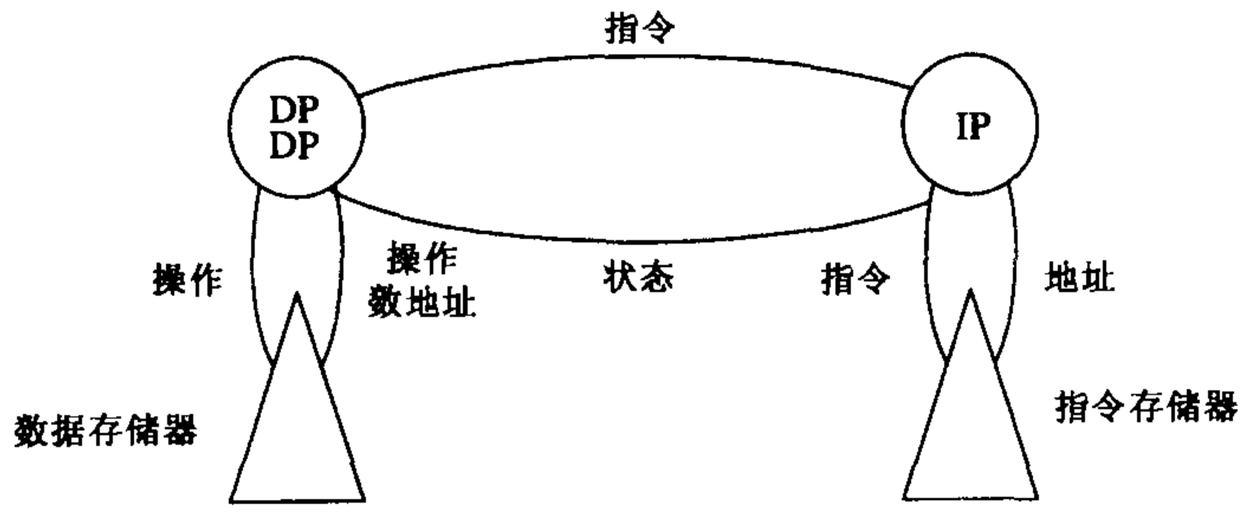


图 11.5　简单机器的抽象模型

表 11.4　可能的连接方案

连接类型	含义
1 - 1	两个单一部件间的连接
1 - n	一个单一部件和 n 个其他部件间的连接
n - n	n 个(1 - 1)的连接
$n \times n$	n 个(1 - n)的连接

使用给定的连接方案,Skillicorn 获得了 28 种不同的类型。如表 11.5 所示为样例类型。表中最右边的一列是相应的弗林类型。图 11.6 说明了按照这一分类而得到的 4 个实例类。

Skillicorn 分类的主要优点包括:(1)简单性;(2)对部件间连接性的恰当考虑;(3)灵活性;(4)能表示最现代计算机系统的能力。但是,该分类不包含如流水线那样的操作情况,所以也难于预测属于同一类、但又不清楚该类中互连方式的机器的相对能力。

多处理机系统可被进一步分为紧耦合和松耦合的。在紧耦合系统中,所有处理器平等地访问一个全局存储器。另外,每个处理器也可以有自己的局部或高速缓存存储器。在松耦合系统中,存储器被处理器所分割,从而每个处理器有附属于自己的存储器。但是处理器仍共享相同的存储器地址空间。任何处理器都能直接访问任何远程的存储器。紧耦合多处理器的例子包

括 CMU C.mmp,Encore Computer Multimax,以及 Sequent 公司的 Balance 系列。松耦合多处理器的例子包括 CMU Cm*,BBN Butterfly,以及 IBM RP3。

表 11.5 连接类型示例

类	IP	DP	IP-DP	IP-IM	DP-DM	DP-DP	描述	弗林分类
1	1	1	1-1	1-1	1-1	无	冯·诺依曼单处理器	SISD
2	1	N	1-n	1-1	n-n	$n \times n$	类型 1 阵列处理器	SIMD
3	1	N	1-n	1-1	$n \times n$	无	类型 2 阵列处理器	SIMD
4	N	N	n-n	n-n	n-n	$n \times n$	松耦合冯·诺依曼	MIMD
5	N	N	n-n	n-n	n-n	无	紧耦合冯·诺依曼	MIMD

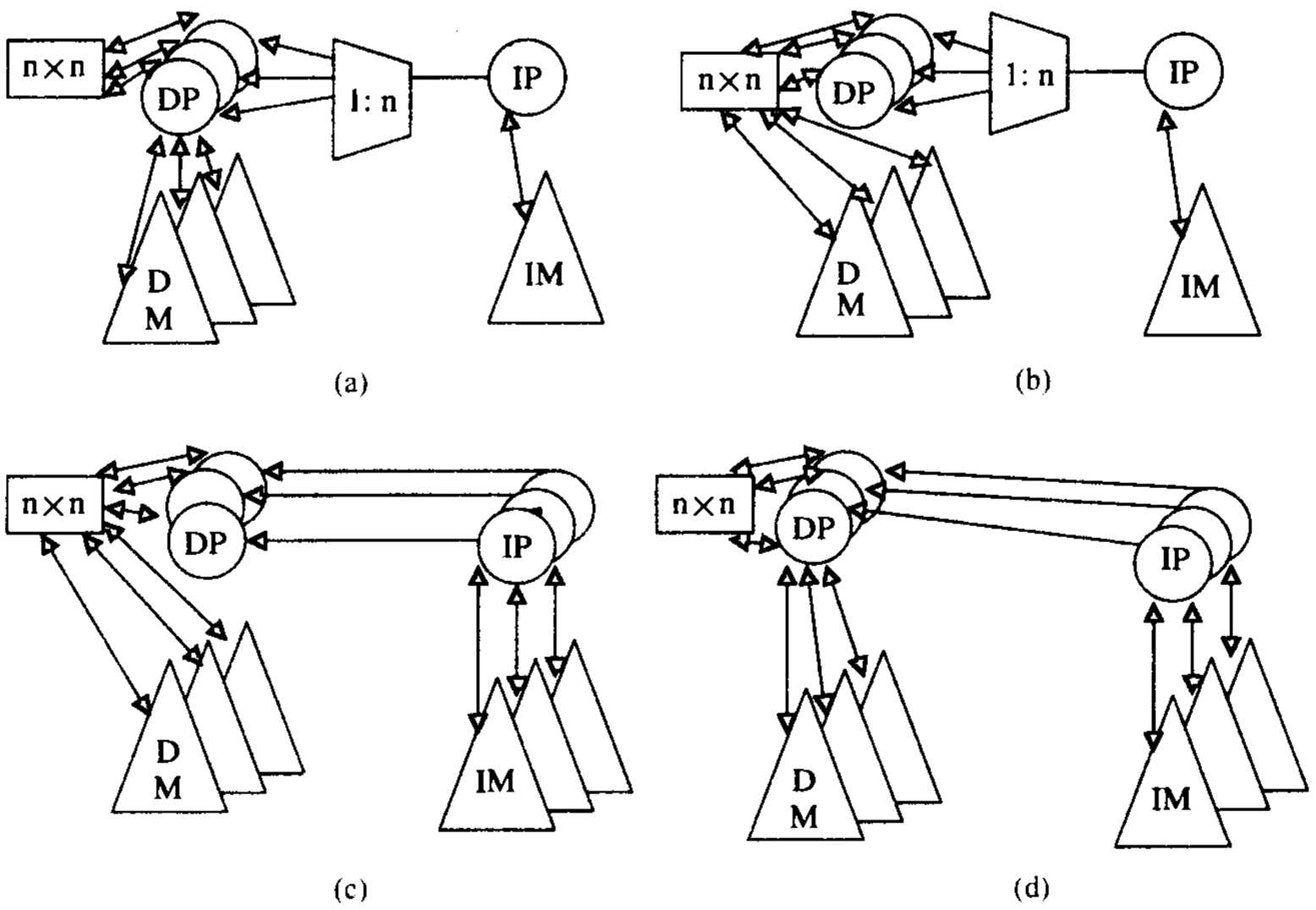


图 11.6 连接类型示例:(a)阵列 1 类;(b)阵列 2 类;(c)紧耦合多处理器;(d)松耦合多处理器

11.3 SIMD 方式

请回忆弗林分类中的 4 种基本体系结构,其中,SIMD 和 MIMD 通常用来构造并行体系结构。在本节中,我们将提供有关 SIMD 方式的基本信息。重要的是在开始时我们必须指出 SIMD 主要是用来开发在如图像处理应用中所需的矩阵(数组)操作中所出现的固有并行性。商品化的著名实际机器有 ILLIAC-IV(1972),STARAN(1974)和 MPP(1982)。

在实际的机器中有两种主要的配置,如图 11.7 所示。

在第一种方案中,每个处理器有自己的局部存储器。处理器可以通过互连网络相互进行通信。若互连网络在一对给定的处理器间未提供直接的连接,则这对处理器可通过一个中间处理器进行数据交换。ILLIAC IV 就使用这种互连方案。ILLIAC IV 中的互连网络允许每一个处理器在一个 8×8 的矩阵模式中直接与 4 个相邻的处理器通信,也就是说第 i 个处理器能直接与第 $(i-1),(i+1),(i-8)$和$(i+8)$个处理器进行通信。

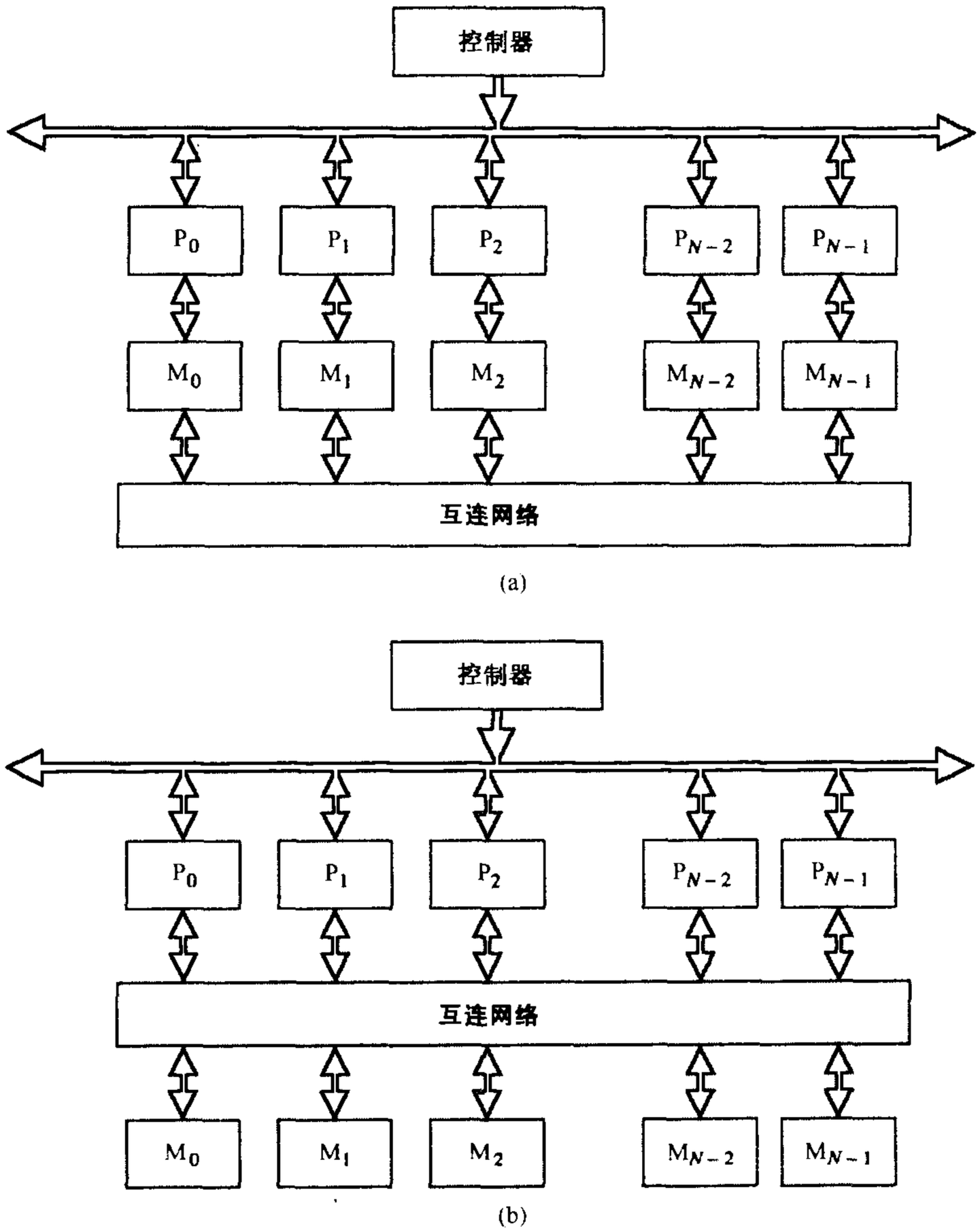


图 11.7　两种 SIMD 方式:(a)SIMD 方式 1;(b)SIMD 方式 2

在第二种 SIMD 方案中,处理器和存储器模块间的通信是通过互连网络进行的。两个处理器可通过中间的存储器模块进行相互通信,也可通过中间的处理器进行通信。例如,假定处理器 i 与存储器模块$(i-1)$,i 和$(i+1)$连接,此时,处理器 1 就能通过中间的存储器模块 2,3 和 4 与处理器 5 进行通信。BSP(Burroughs Scientific Processor)使用 SIMD 的第二种方案。

为了说明 SIMD 在处理数组操作时的有效性,让我们考虑两个一维数组 A 和 B 相应元素相加并将结果存入第三个一维数组 C 的操作。再假定使用 SIMD 的方式 1。若使三个数组的元素分布成在 M_0 中含有元素 $A(0)$,$B(0)$和 $C(0)$,M_1 中含有元素 $A(1)$,$B(1)$和 $C(1)$,…,以及 M_{N-1}中含有元素 $A(N-1)$,$B(N-1)$和 $C(N-1)$,则在一步中可完成 N 个加法。在这种情况下,所有的处理器将同时执行 $C \leftarrow A+B$ 的一个加法指令。在所有处理器执行完这一单步后,结果数组 C 的元素将跨存储器模块进行存储以使 M_0 存储 $C(0)$,M_1 存储 $C(1)$,…,以及 M_{N-1}存储$C(N-1)$。

习惯上我们用 5 元(N,C,I,M,F)形式表示 SIMD 机器。其中每个参量的含义给定如下:

1. N 是处理单元的数目$(N=2^k, k\geqslant 1)$;

2. *C* 是控制部件使用的控制指令集，如 do，for，step；
3. *I* 是由处理部件执行的指令集；
4. *M* 是有效处理单元的子集；
5. *F* 是确定处理单元间通信链路的互连函数集。

11.4 MIMD 方式

MIMD 机器使用自己拥有存储器的处理器集协同执行给定的任务。一般而言，MIMD 系统根据它的存储器组成可分类成共享存储器和消息传递体系结构。这两者间的选择依赖于应用中的通信(与计算有关的)代价和应用问题的负载平衡程度。

11.4.1 共享存储器组成

近期以来业界对分布式共享存储器系统的兴趣在不断增长。这是因为共享存储器为进程间的交互提供了一个有吸引力的概念性模型，尽管底层的硬件不提供直接的支持。在一个共享存储器模型中，处理器间的通信是通过读、写为所有处理器平等访问的一个共享存储器中的单元来完成的。每个处理器可以有寄存器、缓冲器、高速缓存和局部存储器体作为附加的存储器资源。

在设计共享存储器系统时必须考虑一些基本问题，其中包括访问控制、同步、保护和安全。访问控制决定了哪个进程可以访问哪些资源。访问控制模型将根据访问控制表的内容对每一个由处理器向存储器发出的访问请求加以必要的检查。访问控制表中含有决定每一次访问企图合法性的标志。若存在访问资源的企图，则直到所期望的访问完成之前，将阻塞所有不被允许访问的企图以及不合法的进程。来自共享进程的请求在执行过程中可以改变访问控制表的内容。访问控制的标志以及同步规则决定了系统的功能。同步约束限制了共享进程对共享资源的访问时间。恰当的同步保证了信息的正确流动并保证了系统的功能性。保护是系统的一个特征，以防止进程随意访问属于其他进程的资源。共享和保护两者是不兼容的；共享允许访问，而保护则限制访问。

在两个处理器上运行同一程序的两个副本，由于对共享存储器的竞争将降低相对于单处理器系统的性能。当 3 个、4 个或更多个程序副本同时执行时将进一步使性能降低。

共享存储器系统由一组独立的处理器，一组存储器模块，以及一个互连网络所组成。最简单的共享存储器系统由一个可被两个处理器 P_a 和 P_b 访问的存储器模块(M)和这两个处理器所组成，如图 11.8 所示。访问请求通过存储器的两个端口到达存储器。存储器模块中的仲裁部件将到达的请求送往存储器控制器。若存储器模块不忙，且仅有一个请求到达，则仲裁部件就将该请求送往存储器控制器，并满足该请求。当一个存储器模块正在服务时，模块就被设置为忙状态。当存储器正忙于对前一个请求服务时，若有新的请求到达，则存储器模块就通过存储器控制器向提出新请求的处理器发送一个等待信号。作为响应，请求处理器可能在线保持其请求直至存储器变为空闲或是稍后再重复其请求。若仲裁部件接收到两个请求，它就选择一个并将其传送给存储器控制器。同样，被拒绝的请求或是保留等待下一次服务或是稍后再重复其请求。

采用仲裁部件构成由两个处理器使用的存储器模块可能是不合适的。主要问题来自两个处理器对存储器访问的交互排序。考虑由两个处理器 P_a 和 P_b 访问同一存储单元 M(1000)而引起的以下两种情景，如图 11.9 所示。让我们进一步假定存于存储单元 M(1000)中的初始值为

150。注意在这两种情况中,由每个处理器所完成的指令序列是相同的。两种情形的惟一差别在于两个处理器更新 M(1000)中的值的相对时间。对两种情形加以细致观察就可看到存于单元 M(1000)中的值在第一种情形下将为 151,而在第二种情形下将为 152。

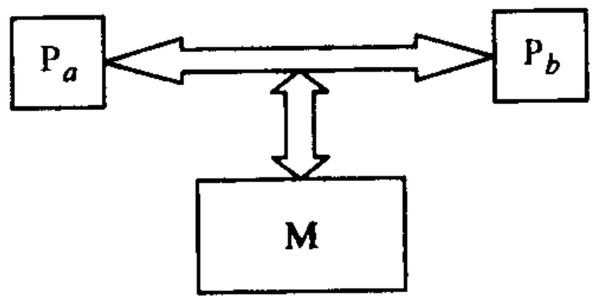


图 11.8　简单的共享存储器方式

周期	处理器 P_a	处理器 P_b
1	a ← M(1000);	
2		b ← M(1000);
3	a ← a + 1;	
4		b ← b + 1;
5	M(1000) ← a;	
6		M(1000) ← b;

情形 1

周期	处理器 P_a	处理器 P_b
1	a ← M(1000);	
2	a ← a + 1;	
3	M(1000) ← a;	
4		b ← M(1000)
5		b ← b + 1;
6		M(1000) ← b;

情形 2

图 11.9　潜在的共享存储器问题

以上的说明性举例呈现了这个简单共享存储器系统非功能性行为的情况。该例展示了要使这类系统成功工作的基本要求。这些要求是:

1. 解决竞争处理器间冲突的机制;
2. 说明排序约束的技术;
3. 强制排序规约的机制。

满足这些基本要求的方法在我们的“先进计算机体系结构与并行处理”一书第 4 章中做了介绍。

在共享存储器多处理机系统中不同互连网络的使用将导致系统具有以下的特征之一:

1. 具有均匀存储器访问(UMA)性质的共享存储器体系结构;
2. 全高速缓存(COMA)存储器体系结构;
3. 非均匀存储器访问(NUMA)的分布式共享存储器体系结构。

如图 11.10 所示为以上所述的 3 种共享存储器体系结构的典型组成。在 UMA 系统中,一个共享存储器可为所有处理器通过互连网络加以访问,就如同一个单处理器访问它的存储器一样。所以所有处理器对任何存储单元有相同的访问时间。用于 UMA 中的互连网络可以是单总线、多总线、交叉开关或多端口存储器。

在 NUMA 系统中,每个处理器拥有附属的部分共享存储器。存储器具有单一的地址空间。所以,任一处理器都能直接使用它的实际地址访问任何存储器单元。但是,对模块的访问时间则依赖于与处理器的距离。这就导致非均匀的存储器访问时间。有许多体系结构以 NUMA 方式将处理器互连到存储器模块。其中有树和层次总线网络(可参阅“先进计算机体系结构与并行处理”一书的第 2 章)。

与 NUMA 类似,在 COMA 中每个处理器具有部分共享存储器,但此时的共享存储器是由高速缓存组成的。COMA 系统需要将数据迁移到请求它的处理器中。

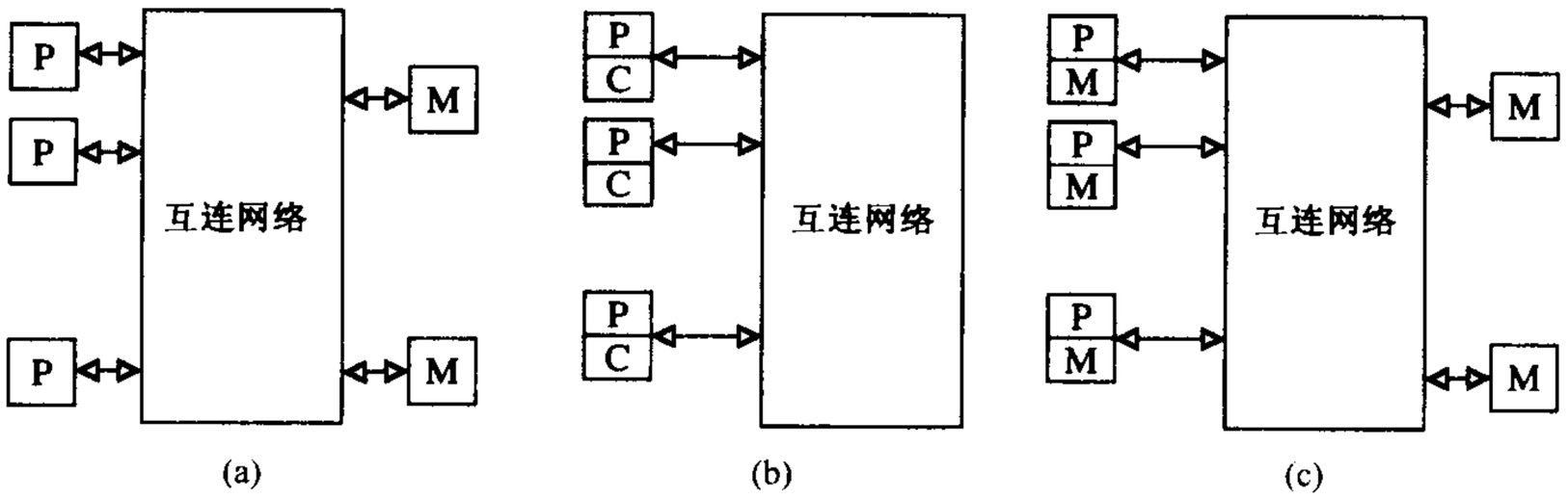


图 11.10 共享存储器体系结构的三个实例:(a)UMA;(b)COMA;(c)NUMA

11.4.2 消息传递组成

消息传递代表了另一种多处理器间数据通信和移动的方法。在处理器间使用本地而不是全局存储器进行消息的通信。一个消息定义为一组在处理器间的直接链路上传播的相关信息。有许多的消息传递模型。消息传递系统的实例包括 cosmic 立方体,工作站机群及 transputer。

于 1983 年提出的 transputer 系统 T212 宣告了第一台消息传递多处理机系统的诞生。此后,1985 年 T414 诞生,1986 年 Inmos 公司提出了 VLSI 的 transputer 处理器。在此之后,又提出了两个后继的 transputer 产品 T800(1988)和 T9000(1990)。Cosmic 立方体消息传递多处理器是在 1981 – 1985 年期间于加州理工学院(Caltech)设计的。它代表了第一个实际工作的超立方体多处理机系统。在消息传递中的虫孔路由是 1987 年作为传统的存储 – 转发路由的另一种选择而提出的,意在减少所需的缓冲器大小并降低消息的时延。在虫孔路由中,一个包被分成称为片(flits,flow control bits,流控制位)的更小单位,并使片以流水方式移动,由包的头片引导到目的结点。当网络发生拥塞时,头片就被阻塞,从而其余的片也就被阻塞(更多的细节请参阅“先进计算机体系结构与并行处理”一书的第 5 章)。

取消大的全局存储器(这通常是使整个系统变慢的一个原因)以及它的异步性质,使得消息传递方式胜过共享存储器方式。与共享存储器多处理器类似,应用程序被分成更小部分;每个部分能由一个单独的处理器以并发的状态加以执行。

如图 11.11 所示为消息传递多处理器体系结构的一个简单例子。从图中可以看到,处理器使用本地总线(内部通道)与它的本地存储器进行通信,而与其他处理器的通信则通过互连网络(外部通道)进行。运行在给定处理器上的进程使用内部通道相互交换消息,而运行在不同处理器上的进程则使用外部通道相互交换消息。这样的方案提供了很大的灵活性以适应大量的处理器,且容易扩展。注意,进程以及执行它的处理器被认为是两个分离的实体。进程的大小由程序员确定,并可用粒度进行如下描述:

$$\text{粒度} = \frac{\text{计算时间}}{\text{通信时间}}$$

粒度可以用三种类型加以区别,它们是:

1. 粗粒度。每个进程拥有大量的顺序指令,并花费大量的时间加以执行;
2. 中粒度。由于进程的通信开销随粒度减小而增加,因此中粒度代表了减少通信开销的一个中间地带,使得每个结点的通信能耗费较少的时间;
3. 细粒度。每个进程只含有几条顺序指令(最少时仅有一条指令)。

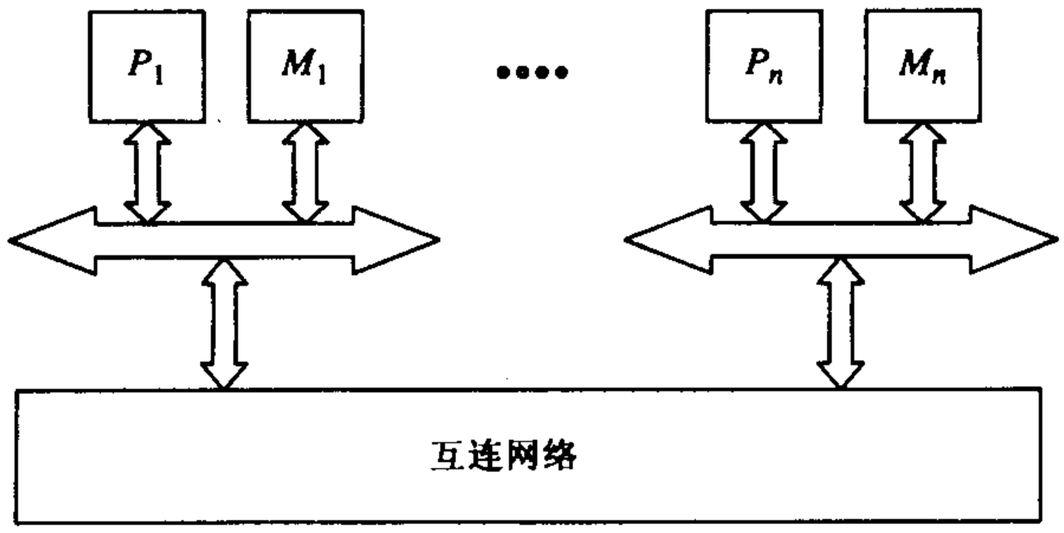


图 11.11　消息传递多处理器体系结构示例

消息传递多处理机系统绝大多数使用中粒度或粗粒度。

消息传递多处理机系统使用各种静态网络进行局部通信。其中尤其是超立方体网络一直得到了特殊的关注。在消息传递多处理机系统中还同样可使用与超立方体网络最邻近的二维和三维网格网络。两个重要的因素导致超立方体和网格网络适合用于消息传递网络。这两个因素是方便 VLSI 的实现和适合于二维和三维的应用。

为消息传递系统设计互连网络必须考虑两个重要的设计因素，这就是链路带宽(bandwidth)和网络时延(latency)。链路带宽定义为单位时间内能传输的位数(b/s)。网络时延定义为完成一个消息传输所需的时间。例如，链路可以是单向或双向的，每次可传送一位或多位。要评估网络时延，我们首先必须确定通路的建立时间，它依赖于通路上的结点数，而依赖于消息大小的实际传输时间也必须加以考虑。

从一个给定源通过网络传输信息可按以下两种方式进行：

1. 电路交换网络。在这种类型的网络中，每个结点中不需要缓冲器。首先要在源和目的间建立通路，沿该通路的所有链路被预留。消息传送完成后，预留的链路被释放以供其他消息使用。电路交换网络最突出的特征是它有最小的延迟。电路交换网络的主要缺点是链路使用效率低下。因此，电路交换网络仅在传输较大消息时才有优势；
2. 包交换网络。在这种网络中，消息在结点间传输之前被分割成更小的称为包的部分。每个结点必须有足够的缓冲器以在传输包之前保持所接收到的包。在传输开始时，从源到目的的完整通路可能还不能使用。一旦链路变为可用，包便从一个结点移动到另一个结点，直至到达目的地。该技术也称为存储 - 转发包交换技术。

虽然存储 - 转发包交换网络消除了在开始传输时所需的完整通路，但它会增加整个网络的时延。这是因为包将被存储在结点的缓冲器中等待输出链路可用。为了减少所需的缓冲器大小并降低网络的时延，人们提出了虫孔路由(wormhole routing)。

在涉及了某些基于弗林分类的机器类型后，我们现在来介绍在这些机器中所使用的互连网络。我们撰写的“先进计算机体系结构与并行处理”一书的第 2 章中对多处理器的互连网络做了详细的论述。

11.5　互连网络

互连网络(IN)有许多分类准则。以下是这些分类准则。

11.5.1 操作方式

按照操作方式,互连网络可分为同步或异步的。在同步操作方式中,系统中的所有部件使用单一的全局时钟,因此整个系统以锁步方式运行。另一方面,异步操作方式则不需要全局时钟,而是使用握手信号来协调异步系统的操作。虽然与异步系统相比同步系统显得较慢,但却无竞争和冒险。

11.5.2 控制策略

按照控制策略,互连网络可分为集中或分散的。在集中式控制系统中,用单个中央控制部件监视和控制系统部件的操作。而在分散式控制中,控制功能分布在系统内的不同部件中。在集中式控制系统中,中央控制部件的功能和可靠性可能成为瓶颈。交叉开关是集中式系统,而多级互连网络则是分散式系统。

11.5.3 交换技术

按照交换机制,互连网络可分为电路交换网络及包交换网络。在电路交换机制中,在通信开始前,源和目的结点间必须先建立一条完整的通路,且在整个通信期间所建立的通路必须保持。在包交换机制中,源和目的间的通信是借助被划分为更小的、称为包的消息实体进行的。在到达目的地途中,包可以以存储 - 转发方式由一个结点送往另一个结点,直至到达目的地。与电路交换相比,虽然包交换显得能更有效地使用网络资源,但却有可变包延迟的缺点。

11.5.4 拓扑

按照它们的拓扑,互连网络可被分为静态和动态的。在动态互连网络中,使用开关元件在输入和输出间建立连接。根据交换开关的设置可建立不同的互连。在静态互连网络中,结点间存在有固定的直接链路,静态互连网络中没有开关元件(结点)。

在介绍了互连网络的通用分类准则后,我们现在可以为互连网络引入一个可能的基于拓扑的分类法。图 11.12 提供了这样一个分类方法。

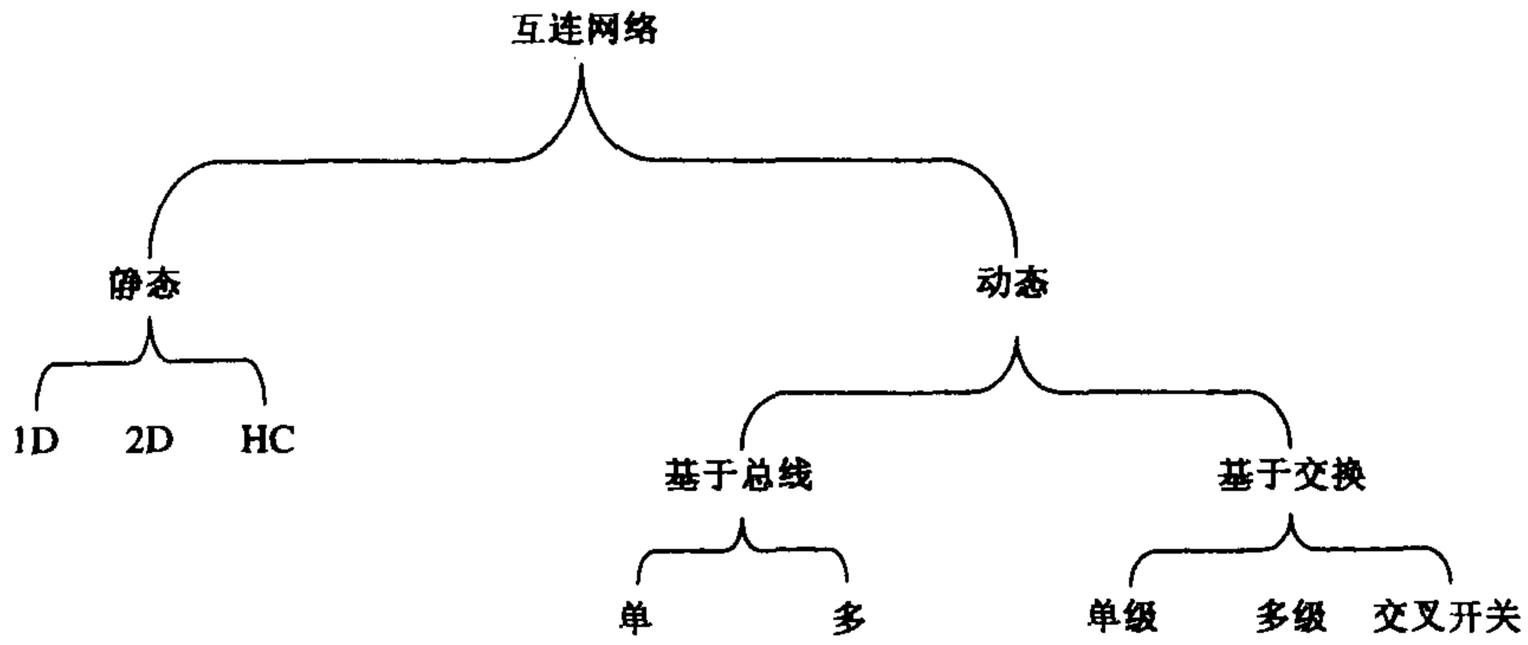


图 11.12 基于拓扑的互连网络分类方法

按照所示的分类法,互连网络可被分为静态和动态的。按它们的互连模式,静态互连网络可进一步被分为一维(1D),二维(2D)或超立方体(HC)。另一方面,动态网络按互连方案可进一步被分为基于总线或基于交换开关的。基于总线的互连网络可被分为单总线或多总线的。基

于交换开关的动态互连网络按照互连网络的结构可进一步分为单级(SS),多级(MS)或交叉开关网络。

我们撰写的“先进计算机体系结构与并行处理”一书的第 2 章中对多处理器的互连网络做了详细的解释。

11.6　分析和性能指标

在介绍了多处理器体系结构后,我们现在来叙述在多处理器中有关性能问题的某些基本思想。对详情感兴趣的读者可参阅我们撰写的“先进计算机体系结构与并行处理”一书的第 3 章。

这里一个基本问题是与单处理器系统相比使用多处理器求解一个给定问题到底能快多少?这一问题可用以下的加速公式加以定义:

$$\begin{aligned} S(n) &= \text{加速} \\ &= \text{由于使用 } n \text{ 个处理器的多处理机系统而增加的速度} \\ &= \frac{\text{单处理器系统的执行时间}}{n \text{ 处理机系统的执行时间}} \end{aligned}$$

一个相关的问题是 n 个处理器中的每一个是如何被有效利用的。该问题可用以下的公式加以定义:

$$E(n) = \text{效率} = \frac{S(n)}{n} \times 100\%$$

在使用多处理机系统执行任务(程序)时,可以假设一个给定的任务可被分割成 n 个相等的子任务,而其中的每一个子任务可由一个处理器执行。因此所期望的加速将由 $S(n) = n$ 给定,而效率 $E(n) = 100\%$。这种一个给定的任务可被分割成 n 个相等子任务,每一个子任务由一个处理器执行的假设是不现实的。我们撰写的“先进计算机体系结构与并行处理”一书的第 3 章中将详细描述和分析有意义的计算模型。在同一章中还介绍和分析了许多其他的性能指标。

11.7　小结

在本章中我们论及了有关多处理器问题的许多概念和系统配置,特别是我们提出了在多处理器环境中使用的通用概念和术语。我们介绍和分析了一些多处理器的分类法,对两种存储器的组成也做了介绍,它们是共享存储器和消息传递系统。另外,我们还介绍了用来互连多处理器的不同拓扑。我们撰写的“先进计算机体系结构与并行处理”一书的第 2 章中对有关互连网络的性能指标有更多的论述,而该书的第 4 章和第 5 章又各自对共享存储器和消息传递体系结构做了解释。

习题

1. 考虑在本章中所讨论的体系结构的 5 种分类方案。请列表指出每一种分类的优缺点,需要考虑每一种分类满足分类所要达到目的的程度。
2. 从 5 种已有的分类中推导出一种新的分类,它将胜过 5 种分类的每一个。以表格的形式列出所提出新分类的新增优点和消除的缺点。

3. 列表比较 SIMD 和 MIMD 机器的主要优缺点。
4. 列表比较共享存储器和消息传递模式的优缺点。
5. 列举 3 个所熟悉的工程应用,说明哪一个是 SIMD 最能有效使用的,同样列举另外 3 个,说明哪一个是 MIMD 最能有效使用的。
6. 考虑使用图 11.1 中的各种互连网络连接 N 个处理器和 N 个存储器模块的情况。假定 T 是一个处理器访问存储器模块中一项所需要的时间,并假定所有处理器对不同的存储器模块发出访问请求。试计算在 4 种互连网络中每一种可能出现的最坏情况下的延迟。
7. 文中已经提及可以用 5 元形式(N, C, I, M, F)对一个给定的 SIMD 机器加以描述。要求选择 3 种最近的不同 SIMD 机器,并以表格形式列出它们 5 元中的每一个。
8. 假设 2 个元素的简单加法需要一个单位时间,现在要求使用以下的每一种方式计算完成 40×40 元素数组的加法所需的执行时间:
   (a) 一个有 64 个处理单元,以最近邻居方式连接的 SIMD 系统,认为每个处理器只有自己的局部存储器。
   (b) 一个有 64 个处理单元,通过互连网络连接到一个共享存储器的 SIMD 系统,通信时间忽略不计。
   (c) 一个有 64 个独立处理单元,通过互连网络访问共享存储器的 MIMD 系统,通信时间忽略不计。
   (d) 若通信时间需要 2 个时间单位,重做(b)和(c)。
9. 针对图 11.1 中 4 种不同的互连网络,就互连网络的 4 个属性(操作方式,控制策略,交换机制和拓扑)中每一个的适用性进行精确的讨论。证实你对一个给定互连网络的给定属性适用性的评论是有充分理由的。
10. 考虑一个由 N 个处理器所组成的一个多处理机系统,假定每个处理器执行一个给定临界区所需的时间为 t,f 代表能并行执行的操作部分所占的比例。再假定单个处理器执行相同任务需时为 T。请证明使用 N 个处理器执行该任务所需的总时间将由下式给定:

$$T_N = (1-f) \times T + \frac{f \times T}{N} + t$$

为了获得最少的执行时间 T_N 所需的处理器数 N 应为多少?

参考文献和推荐读物

S. Abraham and K. Padmanabhan, Performance of the direct binary n-cube network for multiprocessors, *IEEE Trans. Comput.*, 38(7), 1000 – 1011(1989).

P. Agrawal, V. Janakiram and G. Pathak, Evaluating the performance of multicomputer configurations, *IEEE Trans. Comput.*, 19(5), 23 – 27 (1986).

G. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin Cummings, Redwood City, CA, USA, 1989.

K. Al-Tawil, M. Abd-El-Barr and F. Ashraf, A survey and comparison of wormhole routing techniques in mesh networks, *IEEE Network*, 11 (2), 38 – 45(1997).

L. Bhuyan, Q. Yang and D. Agrawal, Performance of multiprocessor interconnection networks, *IEEE Comput.*, 22(2), 25 – 37 (1989).

W.-T. Chen and J.-P. Sheu, Performance analysis of multiple bus interconnection networks with hierarchical requesting model, *IEEE Trans. Comput.*, 40(7), 834 – 842 (1991).

S. Dasgupta, *Computer Architecture: A Modern Synthesis*, Vol. 2: Advanced Topics, John Wiley, New York, 1989.

A. Decegama, *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Hardware Volume 1*, Prentice-Hall, NJ, 1989.

J. Dongarra, *Experimental Parallel Computing Architectures*, North-Holland, Amsterdam, 1987.

A. Goyal and T. Agerwala, Performance analysis of future shared storage systems, *IBM J. Res. Devel.*, 28(1), 95 – 107 (1984).

J.-Y. Juang and B. Wah, A contention-based bus-control scheme for multiprocessor systems, *IEEE Trans. Comput.*, 40(9), 1046 – 1053 (1991).

T. Lewis and H. El-Rewini, Introduction to Parallel Computing, Prentice-Hall, Englewood Cliffs, NJ, 1992.

D. Linder and J. Harden, An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes, *IEEE Trans. Comput.*, 40(1), 2 – 12 (1991).

L. Ni and P. McKinely, A survey of wormhole routing techniques in direct networks, *IEEE Comput.*, 26 (2), 62 – 76 (1993).

J. Patel, Performance of processor – memory interconnections for multiprocessor computer systems, *IEEE Trans.*, 28(9), 296 – 304 (1981).

D. Reed and R. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, MA, USA, 1987.

H. El-Rewini and T. Lewis, Distributed and Parallel Computing, Manning & Prentice Hall, 1998.

H. El-Rewini and M. Abd-El-Barr, Advanced Computer Architecture and Parallel Processing, John Wiley, Hoboken, NJ, USA, 2005.

E. Sima, T. Fountain and P. Kacsuk, *Advanced Computer Architectures: A Design Space Approach*, Addison Wesley, MA, USA, 1996.

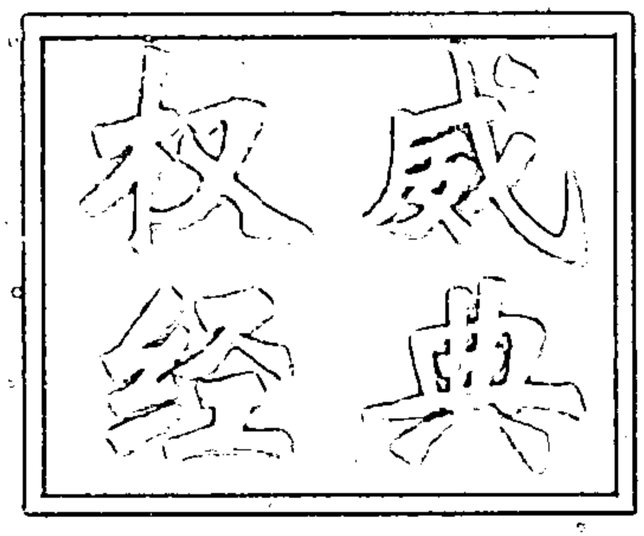
H. Stone, *High Performance Computer Architecture*, 3rd ed., Addison Wesley, MA, USA, 1993.

B. Wilkinson, *Computer Architecture: Design and Performance*, 2nd ed., Prentice-Hall, Hertfordshire, UK, 1996.

Q. Yang and S. Zaky, Communication performance in multiple-bus systems, *IEEE Trans. Comput.*, 37 (7), 848 – 853 (1988).

H. Youn and C. Chen, A comprehensive performance evaluation of crossbar networks, *IEEE Trans. Parallel Distribute Syst.*, 4(5), 481 – 489 (1993).

M. Zargham, *Computer Architecture: Single and Parallel Systems*, Prentice-Hall, NJ, USA, 1996.



计算机组成与体系结构

Fundamentals of Computer Organization and Architecture

本书作者在计算机组成与体系结构方面具有多年的教学和科研经验，他们用浅显平实的语言将多年的理解和心得转述给读者，力求使读者能快速、到位地理解计算机组成与体系结构中的重要概念和方法。本书结构清晰，语言浅显易懂，选取的例子紧密结合实践，非常便于教师整理讲课材料。本书覆盖的内容要点有：

- 指令集体系结构和设计
- 汇编语言编程
- 处理单元设计
- 存储系统设计
- 输入输出设计和组织
- 流水线设计技术
- 精简指令集计算机
- 多处理机系统介绍

Mostafa Abd-El-Barr

博士，现在是科威特大学信息科学系的教授和系主任。他是另外两本专业书的作者，已发表120余篇论文，曾是许多国际会议和专题讨论会的主席。

Hesham El-Rewini

博士，专业工程师，美国南卫理公会大学（SMU）计算机科学与工程系的正教授和系主任。他是另外数本著作的合作作者，在学术期刊和会议录上发表了许多研究论文，并曾担任过许多国际会议的主席。

ISBN 7-121-01955-8

9 787121 019555 >

传播教育信息 共享教育资源
华信教育资源网
www.hxedu.com.cn
为您提供优质教学资源

WILEY PHEI

责任编辑：贺瑞君
特约编辑：吴 颖
责任美编：毛惠庚

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

ISBN 7-121-01955-8 定价：23.00元