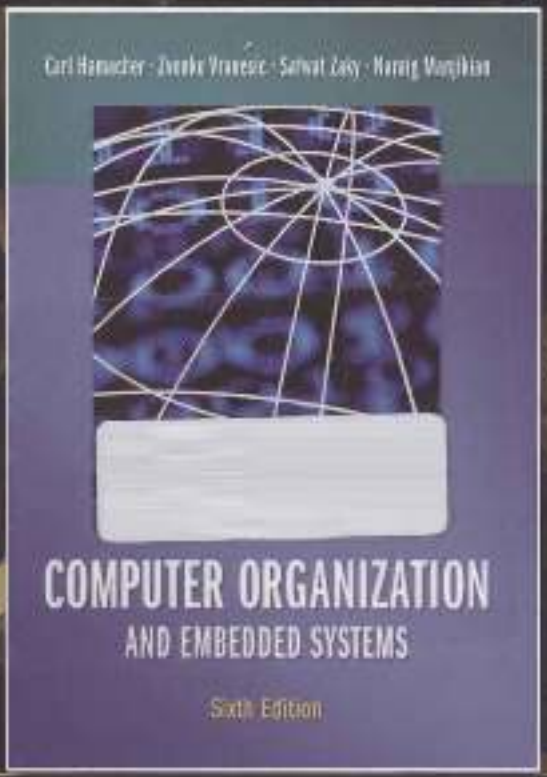
计 算 机 科 学 丛 书

原书第6版

# 计算机组成与嵌入式系统

(加) Carl Hamacher 女皇大学 Zvonko Vranesic 多伦多大学 Safwat Zaky 多伦多大学 Naraig Manjikian 女皇大学 著 王国华

Computer Organization and Embedded Systems Sixth Edition




机械工业出版社
China Machine Press

# 计算机组成与嵌入式系统（原书第6版）

## Computer Organization and Embedded Systems Sixth Edition

本书是一本经典的计算机组成教材，自1978年问世以来，已被多所世界知名大学选为教材。本书知识结构合理，知识点全面完整，基本概念广泛而新颖。书中不仅介绍了硬件设计的原理，说明了硬件设计如何受软件需求影响，而且以流行的商用处理器作为实例，描述了各种基本知识和基本概念的应用方法和应用过程，具有很强的实用性。此外，本书还涵盖了当今许多先进的技术和设计思想。

### 本书特色

- 系统地介绍了现代计算机硬件系统的各个组成部分，包括处理器、输入/输出、存储器和互连标准等。
- 以Nios II、ARM、ColdFire和Intel IA-32等商用处理器为例来阐释基本概念，侧重于讨论RISC设计风格的处理器（如MIPS），同时也介绍了CISC设计风格的处理器（如应用比较广泛的商用处理器Intel IA-32）。

### 作者简介

**Carl Hamacher** 女皇大学电子与计算机工程系荣誉退休教授，曾担任女皇大学应用科学系主任，多伦多大学电子工程及计算机科学系教授、计算机系统研究所所长、工程科学部主席。他的研究兴趣是多处理器和多计算机，侧重于网络互连。

**Zvonko Vranesic** 多伦多大学电子与计算机工程系荣誉退休教授，曾参与Altera公司多伦多技术中心的研究和开发工作。他代表加拿大参加过多次国际象棋比赛，拥有国际象棋大师的头衔。他的研究兴趣是计算机体系结构、现场可编程VLSI技术和多值逻辑系统。

**Safwat Zaky** 多伦多大学电子与计算机工程系荣誉退休教授，并且曾担任该系系主任。他的研究兴趣是计算机体系结构、数字电路设计和电磁兼容性。

**Naraig Manjikian** 女皇大学电子与计算机工程系副教授，他的研究兴趣是计算机体系结构、多处理器系统、现场可编程VLSI技术和并行处理应用。



英文影印版
书号：978-7-111-37721-4
定价：69.00元


Mc Graw Hill Education
www.mheducation.com

客服电话：(010) 88378991 88361066
购书电话：(010) 68326294 88379649 68995259
投稿电话：(010) 88379604

版权所有：www.hzmedia.com.cn
华章网站：www.hzbook.com
网上购书：www.china-pub.com



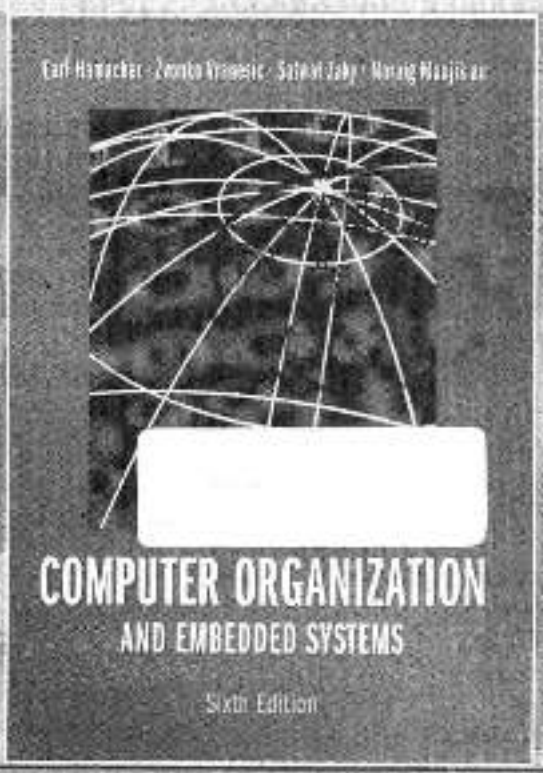

计 算 机 科 学 丛 书

原书第6版

# 计算机组成与嵌入式系统

(加) Carl Hamacher 女皇大学 Zvonko Vranesic 多伦多大学 Safwat Zaky 多伦多大学 Naraig Manjikian 女皇大学 著 王田华 等译

Computer Organization
and Embedded Systems Sixth Edition




机械工业出版社
China Machine Press

**图书在版编目（CIP）数据**

计算机组成与嵌入式系统（原书第6版）/（加）哈马克（Hamacher，C.）等著；王国华等译．北京：机械工业出版社，2013.8
（计算机科学丛书）
书名原文：Computer Organization and Embedded Systems, Sixth Edition

ISBN 978-7-111-43865-6

Ⅰ.计… Ⅱ.①哈… ②王… Ⅲ.①计算机组成原理 ②微型计算机－系统设计 Ⅳ.①TP301 ②TP360.21

中国版本图书馆CIP数据核字（2013）第207203号

**版权所有·侵权必究**
封底无防伪标均为盗版
本书法律顾问 北京市展达律师事务所

**本书版权登记号：图字：01-2011-2302**

Carl Hamacher, Zvonko Vranesic, Safwat Zaky, and Naraig Manjikian: Computer Organization and Embedded Systems, Sixth Edition (978-0407-338865-0)

Copyright © 2012 by McGraw-Hill Education.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation edition is jointly published by McGraw-Hill Education (Asia) and China Machine Press. This edition is authorized for sale in the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan.

Copyright © 2013 by The McGraw-Hill Asia Holdings (Singapore) PTE. LTD and China Machine Press.

版权所有。未经出版人事先书面许可，对本出版物内任何部分不得以任何方式或途径复制或传播，包括但不限于复印、录制、录音，或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳-希尔（亚洲）教育出版公司和机械工业出版社合作出版。此版本经授权仅限在中华人民共和国境内（不包括香港特别行政区、澳门特别行政区和台湾）销售。

版权© 2013 由麦格劳-希尔（亚洲）教育出版公司与机械工业出版社所有。

本书封面贴有 McGraw-Hill Education 公司防伪标签，未标签者不得销售。

本书是计算机组成的入门级教程，全面地介绍了计算机组成结构、操作、性能的基本概念。还介绍了有关外围设备、处理器系列模型以及嵌入式系统的一些基本内容。书中知识具有很强的实用性，并涵盖了当今许多先进的技术和设计思想。

本书特别适合作为电子和计算机专业的本科生关于计算机组成与嵌入式系统方面的入门教程。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：顾 谦 迟振春
北京瑞德印刷有限公司印刷
2013年10月第1版第1次印刷
185mm×260mm·30.5印张
标准书号：ISBN 978-7-111-43865-6
定 价：79.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379001 88361066 购书热线：（010）68326294 88379649 68995259 投稿热线：（010）88379604
读者信箱：hzjg@hzbook.com

本书适用于电子工程、计算机工程、计算机科学专业有关计算机组成和嵌入式系统方面的初级课程。本书的知识结构是相对独立的，假定读者已具备了计算机高级语言程序设计的基本知识。由于多数学习计算机组成的学生都已经学习了数字逻辑电路这一入门课程，因此，本书的主体内容没有包含这一部分知识，但是我们为有需要的读者提供了逻辑电路方面的详尽附录。

本书融入了作者在教授电子与计算机工程、计算机科学以及工程科学专业的本科生时所积累的丰富经验。我们总是从实践的角度进行计算机组成原理的教学，因此在形成本书内容时的一个关键考虑是使用从商用计算机中提取的实例来详细解释主要原理。本书中主要的商业实例来源于：Altera 的 Nios II、Freescale 的 ColdFire、ARM 以及 Intel 的 IA-32 体系结构。

读者必须清楚地认识到，数字系统的设计并不是应用最佳设计算法的简单过程。许多设计决策取决于大量经验性的判断和经验。这包括在一系列选择方案中进行成本/性能、硬件/软件的权衡。我们的目标就是把这些思想传达给读者。

本书可以作为工程学或计算机科学专业一个学期的课程用书，也适用于软件和硬件方向的学生。尽管本书侧重于硬件，我们仍阐述了大量有关软件方面的问题。

McGraw-Hill 建立了一个包含本书辅助材料的网站：http://www.mhhe.com/hamacher。

## 本书的内容

前 3 章介绍了计算机的基本结构，在机器指令级执行的操作，以及程序员可见的输入/输出方法。第 4 章讲述了将汇编语言和高级语言编写的程序翻译成机器语言以及管理其执行过程所需要的系统软件。后 8 章讨论了包括嵌入式系统在内的现代计算机中硬件设备的组织结构、互连和性能。

本书还提供了 5 个详尽的附录。附录 A 涵盖了数字逻辑电路。其后的 4 个附录分别描述了 4 种流行的商业指令集体系结构——Altera 的 Nios II、Freescale 的 ColdFire、ARM 以及 Intel 的 IA-32。

第 1 章对计算机硬件给出了总体描述，并对后续章节中将会深入讨论的术语进行了概括性介绍。该章介绍了基本功能部件以及它们相互作用组成一个完整计算机系统的方法，讨论了数和字符的表示以及基本的算术运算，还介绍了性能问题以及计算机的发展简史。

第 2 章系统地介绍了机器指令、寻址技术和指令序列。该章使用通用的汇编语言表示的机器指令级别的程序示例来讨论循环、子程序和堆栈等概念。在介绍这些概念时，使用了 RISC 风格的指令集体系结构。此外还包括了与 CISC 风格指令集的比较描述。

第 3 章从程序员的角度介绍了基本的输入/输出技术。该章说明了如何使用轮询法进行程序控制 I/O，以及在 I/O 传输中如何使用中断。

第 4 章介绍了系统软件。该章说明了编译器、汇编程序、连接程序和装载程序执行的任务，描述了跟踪和显示程序执行结果的实用程序，也描述了管理用户程序的执行以及包括中断处理在内的输入/输出操作的类作系统程序。

第 5 章探讨了 RISC 风格处理器的设计。该章说明了提取和执行不同类型机器指令所需的

处理步骤序列，然后详细阐述了实现这些处理步骤所需的硬件组织，同时也考虑了 CISC 风格处理器的不同需求。

第 6 章介绍了流水线和多个执行部件在高性能处理器设计中的使用。该章利用第 5 章中描述的 RISC 风格处理器设计的流水线版本来阐明流水线，探讨了编译器的作用以及流水线执行与指令集设计之间的关系，还对超标量处理器进行了讨论。

第 7 章探讨了输入/输出硬件。该章讨论了包括总线结构在内的互连网络，说明了同步和异步操作，也介绍了包括 USB 和 PCI Express 在内的互连标准。

第 8 章讨论了半导体存储器，包括 SDRAM、Rambus 和闪存（Flash memory）的实现。该章介绍了可增加存储器带宽的高速缓存（cache），从性能建模等细节上对其进行了讨论，还介绍了虚拟存储器系统、存储器管理和快速地址转换技术，并将磁盘和光盘作为存储器层次结构的一部分进行了讨论。

第 9 章探讨了计算机中算术部件的实现。该章描述了对补码数进行定点加、减、乘、除操作的硬件逻辑设计，解释了超前进位加法器和快速乘法器，并描述了 Booth 乘数重编码和进位保留加法技术，还介绍了 IEEE 标准中浮点数的表示与操作。

今天，越来越多的处理器被用于嵌入式系统而不是通用计算机中。第 10 章和第 11 章针对嵌入式系统进行了讨论。首先，在第 10 章中介绍了系统集成的基本内容、部件（component）互连以及实时操作，还对微控制器的使用进行了讨论。然后，第 11 章集中讨论片上系统（SoC）的实现，其中单一的芯片上集成了满足特定应用需要的数据处理、存储器，I/O 和定时器功能，并通过一个详尽的例子说明了如何在这样的环境中使用 FPGA 和现代设计工具。

第 12 章讨论了并行处理和性能。该章介绍了用于增强单处理器功能的硬件多线程和向量处理，描述了共享存储器的多处理器以及高速缓存一致性的问题，还介绍了多处理器互连网络。

附录 A 详细介绍了数字逻辑电路知识，可供没有修过电路设计课程的读者参考。

附录 B、C、D 和 E 说明了第 2 章和第 3 章中介绍的指令集概念在 4 种商用处理器 Nios II、ColdFire、ARM 以及 Intel IA-32 中的实现。Nios II 和 ARM 处理器说明了 RISC 的设计风格。ColdFire 采用了相对容易理解的 CISC 设计，而 IA-32 的 CISC 体系结构代表了最成功的商业设计。每个处理器的介绍都包括第 2 章和第 3 章中的汇编语言例子在相应处理器中的实现。这些附录中给出的细节对于理解本书的主体内容来说不是必需的，只涵盖其中一个附录就足以认识商用处理器的指令集了。选择哪个处理器作为例子很可能受到实际实验室中设备的影响，教师不妨使用多个处理器来说明不同的设计方法。

## 第6版的变化

本书的第 6 版对内容和结构安排作了重大改动。主要包括：

- 用 RISC 方法介绍指令集体系结构的基本概念，接着对 RISC 方法与 CISC 方法进行了比较。
- 处理器设计的讨论着重于 RISC 风格体系结构的实现，从而自然地过渡到流水线操作的介绍。
- 用两章介绍了嵌入式系统的内容：一章介绍了嵌入式系统的基本结构和微控制器的使用，另一章则探讨了片上系统的实现。

- 附录给出了4种商用处理器的例子。每个附录包含了给定处理器指令集体系结构的基本信息。
- 在每一章和附录的末尾增加了一个新的小节"问题解析"，给学生提供了一些典型问题的预期解决方案。

## 习题的难度等级

每一章和附录末尾的习题分为以下几类：简单（E）、中等（M）或偏难（D），这些分类的解释如下：

- 简单——直接应用本书所介绍的具体信息便可以在几分钟内得到解决方案。
- 中等——通常不能直接按照本书所介绍例子中采用的方法来解决问题。在某些情况下，解决方案可能是某个例子的一般情况，但比简单问题要花更长的时间。
- 偏难——解决这些问题需要一些额外的洞察力。如果一个解决方案需要编写程序，则其底层的算法或结构可能跟本书给出的任何程序示例都不同。如果需要硬件设计，它所涉及的基本逻辑电路部件的布局和互连可能跟本书所示的任何设计示例都不同。如果要进行性能分析，它可能需要推导代数表达式。

## 课程安排

本书适合作为大学计算机组成入门课程一学期的教材。

书中提供了多于一个学期课程所要讲授的内容，第1章至第9章给出了关于计算机组成和有关软件问题的核心内容。未学习过逻辑电路课程的学生，应该在学习第5章之前学习附录A的内容。

针对嵌入式系统的课程应该包括第1、2、3、4、7、8、10和11章。

感兴趣的学生可在教师的指导下在相关的硬件实验室中实践附录B到E中商用处理器例子的内容。

## 致谢

在此，向许多在第6版筹备期间提供帮助的朋友们表示衷心的感谢。

我们的同事——南巴黎大学的Daniel Etiemble和多伦多大学的Glenn Gulak，提供了许多有助于内容成型的建设性的意见和建议。

Blair Fort和Dan Vranesic提供了一些有价值的程序实例。

罗切斯特理工学院的Warren R. Carithers、北德州大学的Krishna M. Kavi和中西部州立大学的Nelson Luiz Passos对本书第5版和第6版的内容提供了书评。

对本书第5版的内容提供过书评的还有以下人员：多媒体大学的Goh Hock Ann、科罗拉多大学丹佛分校的Joseph E. Beaini、尼赫鲁科技大学的Kalyan Mohan Goli、示范工程学院（埃尔纳古勒姆）的Jaimon Jacob、安娜大学（哥印拜陀）的M. Kumaresan、香港城市大学的Kenneth K. C. Lee、技术教育和研究协会的Manoj Kumar Mishra、马来西亚理科大学的Junita Mohamad-Saleh、布巴内斯瓦尔工程技术学院的Prashanta Kumar Patra、"国立台湾科技大学"的Shanq-Jang Ruan、G. B. Pant农业技术大学的S. D. Samantaray、阿克伦大学的Shivakumar

Sastry、米兰理工大学的 Donatella Sciuto、巴特那国家技术研究所的 M. P. Singh、阿肯色大学的 Albert Starling、加州大学欧文分校的 Shannon Tauro、孔古工程学院的 R. Thangarajan、鲁尔克拉国家技术研究所的 Ashok Kumar Turuk 和辛辛那提大学的 Philip A.Wilsey。

最后，我们衷心感谢 McGraw-Hill 的 Raghothaman Srinivasan、Peter E. Massar、Darlene M. Schueller、Lisa Bruflodt、Curt Reynolds、Brenda Rolwes 和 Laura Fuller 的支持。

Carl Hamacher
Zvonko Vranesic
Safwat Zaky
Naraig Manjikian

# 目　录

Computer Organization and Embedded Systems, Sixth Edition


出版者的话
译者序
前言
作者简介

第 1 章　计算机的基本结构 …… 1
1.1　计算机的类型 …… 1
1.2　功能部件 …… 2
1.2.1　输入设备 …… 3
1.2.2　存储器 …… 3
1.2.3　运算器 …… 4
1.2.4　输出设备 …… 4
1.2.5　控制器 …… 4
1.3　基本操作概念 …… 4
1.4　数的表示及算术运算 …… 6
1.4.1　整数 …… 6
1.4.2　浮点数 …… 11
1.5　字符表示 …… 11
1.6　性能 …… 12
1.6.1　技术 …… 13
1.6.2　并行性 …… 13
1.7　发展历程 …… 13
1.7.1　第一代计算机 …… 14
1.7.2　第二代计算机 …… 14
1.7.3　第三代计算机 …… 14
1.7.4　第四代计算机 …… 14
1.8　结束语 …… 15
1.9　问题解析 …… 15
习题 …… 16
参考文献 …… 17
第 2 章　指令集体系结构 …… 18
2.1　存储单元和地址 …… 18
2.1.1　按字节寻址能力 …… 19
2.1.2　大端和小端分配 …… 20
2.1.3　字的对齐 …… 20
2.1.4　访问数和字符 …… 20
2.2　存储器操作 …… 21
2.3　指令和指令序列 …… 21
2.3.1　寄存器传送标记 …… 21
2.3.2　汇编语言符号 …… 22
2.3.3　RISC 和 CISC 指令集 …… 22
2.3.4　RISC 指令集介绍 …… 23
2.3.5　指令执行和线性序列 …… 24
2.3.6　转移 …… 25
2.3.7　生成存储器地址 …… 26
2.4　寻址方式 …… 26
2.4.1　变量和常数的实现 …… 27
2.4.2　间接和指针 …… 28
2.4.3　变址和数组 …… 29
2.5　汇编语言 …… 32
2.5.1　汇编指示 …… 33
2.5.2　程序的汇编和执行 …… 35
2.5.3　数的表示 …… 36
2.6　堆栈 …… 36
2.7　子程序 …… 38
2.7.1　子程序嵌套及处理器堆栈 …… 39
2.7.2　参数传递 …… 39
2.7.3　堆栈的结构 …… 42
2.8　其他指令 …… 44
2.8.1　逻辑指令 …… 44
2.8.2　移位和循环移位指令 …… 45
2.8.3　乘法和除法 …… 47
2.9　处理 32 位的立即值 …… 47
2.10　CISC 指令集 …… 48
2.10.1　其他寻址方式 …… 49
2.10.2　条件码 …… 50
2.11　RISC 和 CISC 风格 …… 51
2.12　实例程序 …… 52
2.12.1　向量点积程序 …… 52
2.12.2　字符串搜索程序 …… 53
2.13　机器指令的编码 …… 54
2.14　结束语 …… 56
2.15　问题解析 …… 56
习题 …… 59

第3章 基本输入/输出 …… 63
3.1 访问I/O设备 …… 63
3.1.1 I/O设备接口 …… 64
3.1.2 程序控制I/O …… 64
3.1.3 一个RISC风格的I/O程序示例 …… 67
3.1.4 一个CISC风格的I/O程序示例 …… 67
3.2 中断 …… 68
3.2.1 中断的允许与禁止 …… 70
3.2.2 处理多台设备 …… 71
3.2.3 控制I/O设备行为 …… 72
3.2.4 处理器控制寄存器 …… 73
3.2.5 中断程序示例 …… 74
3.2.6 异常 …… 78
3.3 结束语 …… 79
3.4 问题解析 …… 79
习题 …… 83
第4章 软件 …… 86
4.1 汇编过程 …… 86
4.2 装载及执行目标程序 …… 87
4.3 连接程序 …… 88
4.4 库 …… 88
4.5 编译器 …… 88
4.5.1 编译器优化 …… 89
4.5.2 组合不同语言编写的程序 …… 89
4.6 调试器 …… 90
4.7 使用高级语言实现输入/输出任务 …… 90
4.8 汇编语言与C语言的交互 …… 92
4.9 操作系统 …… 95
4.9.1 引导程序 …… 96
4.9.2 管理应用程序的执行 …… 96
4.9.3 中断在操作系统中的使用 …… 97
4.10 结束语 …… 99
习题 …… 99
参考文献 …… 100
第5章 基本处理部件 …… 101
5.1 一些基本概念 …… 101
5.2 指令的执行 …… 103
5.2.1 Load指令 …… 103
5.2.2 算术及逻辑运算指令 …… 104
5.2.3 Store指令 …… 104
5.3 硬件组件 …… 105
5.3.1 寄存器文件 …… 105
5.3.2 ALU …… 106
5.3.3 数据通路 …… 106
5.3.4 取指令部分 …… 108
5.4 指令的读取和执行步骤 …… 109
5.4.1 转移 …… 111
5.4.2 条件存储器 …… 113
5.5 控制信号 …… 114
5.6 硬件控制 …… 116
5.6.1 数据通路控制信号 …… 117
5.6.2 存储器延迟的处理 …… 118
5.7 CICS风格的处理器 …… 118
5.7.1 使用总线实现互连 …… 119
5.7.2 微程序控制 …… 121
5.8 结束语 …… 122
5.9 问题解析 …… 122
习题 …… 124
第6章 流水线 …… 127
6.1 基本概念——理想情况 …… 127
6.2 流水线结构 …… 128
6.3 流水线问题 …… 129
6.4 数据依赖性 …… 129
6.4.1 操作数转发 …… 129
6.4.2 用软件处理数据依赖性 …… 130
6.5 存储器延迟 …… 131
6.6 转移延迟 …… 132
6.6.1 无条件转移 …… 132
6.6.2 条件转移 …… 133
6.6.3 转移延迟槽 …… 133
6.6.4 转移预测 …… 134
6.7 资源限制 …… 136
6.8 性能评估 …… 137
6.8.1 停顿和时间代价的影响 …… 137
6.8.2 流水线的级数 …… 138
6.9 超标量操作 …… 139
6.9.1 转移和数据依赖性 …… 140
6.9.2 无序执行 …… 141
6.9.3 执行完成 …… 141

6.9.4 调度操作 …… 142
6.10 CISC 处理器中的流水线 …… 143
6.10.1 ColdFire 处理器中的流水线 … 144
6.10.2 Intel 处理器中的流水线 …… 144
6.11 结束语 …… 144
6.12 问题解析 …… 145
习题 …… 146
参考文献 …… 148
第 7 章 输入/输出组织结构 …… 149
7.1 总线结构 …… 149
7.2 总线操作 …… 150
7.2.1 同步总线 …… 150
7.2.2 异步总线 …… 152
7.2.3 电气考虑 …… 154
7.3 总线仲裁 …… 154
7.4 接口电路 …… 156
7.4.1 并行接口 …… 156
7.4.2 串行接口 …… 159
7.5 互连标准 …… 161
7.5.1 通用串行总线 …… 162
7.5.2 火线 …… 164
7.5.3 PCI 总线 …… 165
7.5.4 SCSI 总线 …… 168
7.5.5 SATA …… 169
7.5.6 SAS …… 169
7.5.7 PCI Express …… 169
7.6 结束语 …… 170
7.7 问题解析 …… 170
习题 …… 172
参考文献 …… 173
第 8 章 存储器系统 …… 174
8.1 基本概念 …… 174
8.2 半导体随机存储器 …… 175
8.2.1 存储器芯片的内部组织结构 … 176
8.2.2 静态存储器 …… 177
8.2.3 动态随机存储器 …… 178
8.2.4 同步动态随机存储器 …… 180
8.2.5 大容量存储器的结构 …… 182
8.3 只读存储器 …… 184
8.3.1 ROM …… 184
8.3.2 PROM …… 185
8.3.3 EPROM …… 185
8.3.4 EEPROM …… 185
8.3.5 闪存 …… 186
8.4 直接存储器访问 …… 186
8.5 存储器层次结构 …… 188
8.6 高速缓存 …… 189
8.6.1 映射功能 …… 190
8.6.2 替换算法 …… 193
8.6.3 映射技术的例子 …… 194
8.7 性能因素 …… 196
8.7.1 命中率和失效开销 …… 196
8.7.2 处理器芯片上的高速缓存 …… 198
8.7.3 其他改进 …… 198
8.8 虚拟存储器 …… 200
8.9 存储器管理需求 …… 203
8.10 辅助存储器 …… 204
8.10.1 磁盘 …… 204
8.10.2 光盘 …… 208
8.10.3 磁带系统 …… 211
8.11 结束语 …… 212
8.12 问题解析 …… 212
习题 …… 215
参考文献 …… 217
第 9 章 算术运算 …… 218
9.1 有符号数加减法 …… 218
9.2 快速加法器设计 …… 220
9.3 无符号数乘法 …… 224
9.3.1 阵列乘法器 …… 224
9.3.2 顺序电路乘法器 …… 225
9.4 有符号数乘法 …… 227
9.5 快速乘法 …… 229
9.5.1 乘数位倒置编码 …… 229
9.5.2 求和项的进位保留加法 …… 229
9.5.3 使用 3-2 简化器的求和项
加法树 …… 231
9.5.4 使用 4-2 简化器的求和项
加法树 …… 233
9.5.5 快速乘法总结 …… 234
9.6 整数除法 …… 234
9.7 浮点数及其运算 …… 236
9.7.1 浮点数算术运算 …… 238

9.7.2 保护位与截取 …… 239
9.7.3 浮点操作的实现 …… 240
9.8 十进制数到二进制数的转换 …… 242
9.9 结束语 …… 242
9.10 问题解析 …… 243
习题 …… 245
参考文献 …… 248
第 10 章 嵌入式系统 …… 249
10.1 嵌入式系统实例 …… 249
10.1.1 微波炉 …… 249
10.1.2 数码照相机 …… 251
10.1.3 家用通讯技术 …… 252
10.2 嵌入式应用中的微控制器芯片 …… 252
10.3 一个简单的微控制器 …… 253
10.3.1 并行 I/O 接口 …… 253
10.3.2 串行 I/O 接口 …… 256
10.3.3 计数器/定时器 …… 256
10.3.4 中断控制机制 …… 258
10.3.5 编程实例 …… 258
10.4 反应定时器——一个完整的实例 …… 261
10.5 传感器与执行器 …… 264
10.5.1 传感器 …… 264
10.5.2 执行器 …… 266
10.5.3 应用实例 …… 266
10.6 微控制器系列 …… 267
10.6.1 基于 Intel 8051 的微控制器 …… 268
10.6.2 Freescale 微控制器 …… 268
10.6.3 ARM 微控制器 …… 269
10.7 设计问题 …… 269
10.8 结束语 …… 271
习题 …… 271
参考文献 …… 273
第 11 章 片上系统——案例研究 …… 274
11.1 FPGA 的实现 …… 274
11.1.1 FPGA 器件 …… 275
11.1.2 处理器的选择 …… 275
11.2 计算机辅助设计工具 …… 275
11.3 闹钟示例 …… 279
11.3.1 系统的用户视图 …… 279
11.3.2 系统的定义和生成 …… 279
11.3.3 电路实现 …… 281
11.3.4 应用软件 …… 281
11.4 结束语 …… 287
习题 …… 287
参考文献 …… 288
第 12 章 并行处理及性能 …… 289
12.1 硬件多线程 …… 289
12.2 向量（SIMD）处理 …… 290
12.3 共享存储器的多处理器 …… 292
12.4 高速缓存一致性 …… 295
12.4.1 直接写协议 …… 295
12.4.2 写回协议 …… 296
12.4.3 监听高速缓存 …… 296
12.4.4 基于目录的高速缓存一致性 …… 297
12.5 消息传递多计算机 …… 298
12.6 多处理器并行编程 …… 298
12.7 性能建模 …… 300
12.8 结束语 …… 301
习题 …… 301
参考文献 …… 302
附录 A 逻辑电路 …… 303
附录 B Altera Nios II 处理器 …… 344
附录 C ColdFire 处理器 …… 371
附录 D ARM 处理器 …… 397
附录 E Intel IA-32 体系结构 …… 431
索引 …… 461

第1章

Computer Organization and Embedded Systems, Sixth Edition

# 计算机的基本结构

## 本章目标

在本章中你将学习以下内容：

- 计算机的不同类型
- 计算机的基本结构与操作
- 机器指令及其执行
- 数与字符的表示
- 二进制数的加法与减法
- 计算机系统中的基本性能问题
- 计算机发展简史



本书讲述的是计算机的组成结构。书中描述了数字计算机中用于存储和处理信息的各个部件的功能和设计，还介绍了与计算机相连的从外部设备接收信息的输入部件和向外部指定设备传送计算结果的输出部件。输入、存储、处理和输出操作由组成程序的一系列指令管理。

本书的大部分内容是专门针对计算机硬件（computer hardware）和计算机体系结构（computer architecture）的。计算机硬件由电子电路、磁性和光存储介质、显示器、电气机械设备以及通信设施等构成，计算机体系结构包含具体的指令集和用于执行这些指令的硬件设备的功能行为。

本书还介绍了计算机系统中有关程序设计和软件组件方面的许多知识。要想对计算机系统有一个好的认识，重要的是对各个计算机部件设计中的硬件和软件都要有所认识。

## 1.1 计算机的类型

自20世纪40年代数字计算机发明以来，计算机已逐步分化为在其大小、成本、计算能力和使用目的上有着很大不同的许多类型。现代计算机大致可以分为四大类：

- 嵌入式计算机（embedded computer）集成在一个较大的设备或系统中，用以自动监控与控制物理过程或环境。它们被用于特定的目的，而不是通用的任务处理。其典型应用包括工业和家庭自动化、家电、通信产品和交通工具。用户甚至可能并不知道计算机在这类系统中发挥了作用。
- 个人计算机（personal computer）在家庭、教育机构以及商业与工程办公环境中广泛使用，但主要用于个人用途。个人计算机支持各种各样的应用，如通用计算、文档编制、计算机辅助设计、视听娱乐、人际交流和互联网浏览。如今有很多种对个人计算机进行分类的方法。台式计算机（desktop computer）可以满足一般的需求，并占用较少的工作空间。工作站计算机（workstation computer）为工程和科学计算提供更高的计算能力和更强大的图形显示能力。便携式计算机（portable computer）和笔记本电脑（notebook computer）提供了个人计算机的基本功能，它们可以使用电池操作以提供一定的移动性。
- 服务器（server）和企业系统（enterprise system）是能被大量用户共享的大型计算机，这些用户通常会从某种形式的个人计算机上通过公有或私有网络发起访问。这些计算

机可包含大型的数据库，为政府机构或商业组织提供信息处理服务。

- 超级计算机（super computer）和网格计算机（grid computer）通常提供最高的性能，它们是最昂贵的以及物理上最大型的计算机。超级计算机用于天气预报、工程设计与仿真以及科研等对计算要求极高的领域中。因为超级计算机需要很高的成本，所以出现了更经济的网格计算机。网格计算机把大量的个人计算机和磁盘存储单元结合在一个物理上分散的高速网络中，这称为网格，它被当成一个整体计算资源来管理。通过在网格中均匀地分配计算工作量，网格计算机可以在数值计算和信息检索等大型应用上获得很高的性能。

计算机界的一个新趋势是云计算（cloud computing）。个人计算机用户为了自己的计算需求去访问广泛分布的计算和存储服务器资源。互联网提供了必要的通信设施。云端软硬件服务提供商把云作为一种工具进行运作，基于按使用付费的模式进行收费。

## 1.2 功能部件

计算机包括五个功能相对独立的主要部分：输入设备、存储器、算术逻辑部件、输出设备和控制器，如图1-1所示。输入设备接收来自使用键盘等设备的操作员或者通过数字通信线路连接的其他计算机上的编码信息。所接收的信息存储在计算机的存储器中供以后使用，或者立即被算术逻辑部件处理。这个处理步骤是由同样存储在存储器中的一个程序指定的。最后，处理的结果通过输出设备送回到外部设备中。所有这些动作都是由控制器控制和协调的。互连网络提供了这些功能部件间交换信息和协调动作的方法。后面的章节将会详细讨论单个部件以及它们之间的互连。前面提到的算术逻辑电路与主控制电路结合构成了处理器（processor）。输入设备和输出设备通常整体称为输入/输出（input-output, I/O）设备。




图1-1 计算机的基本功能部件

现在仔细观察一下计算机所处理的信息。把信息划分成指令和数据以便于讨论。指令（instruction），或称为机器指令（machine instruction），是具体的命令，它们：

- 控制计算机与I/O设备之间以及计算机内部的信息传送。
- 指定要执行的算术和逻辑运算。

程序（program）是执行任务的指令序列。程序存储在存储器中。处理器从存储器中一条接一条地取出程序指令，然后完成所需要的操作。除了可能有来自于操作员或与计算机连接的I/O设备上的外部中断以外，计算机由所存储的程序控制。数据（data）是指作为指令操作数的数字和字符。数据也存储在存储器中。

计算机处理的指令和数据必须按照适当的格式进行编码。当今的大多数硬件使用的是只有两个稳定状态的数字电路。指令、数字或字符都被编码成叫做位（bit）的二进制数字中，每个位只能从表示两个稳定状态的0和1中取一个。数字通常用按位二进制记数法表示，这将在1.4节中讨论。字母数字字符也可以用二进制码表示，这将在1.5节中讨论。

### 1.2.1　输入设备

计算机通过输入设备接收编码后的信息。最常见的输入设备就是键盘。当一个键被敲击时，相应的字母或数字就会自动转换成相应的二进制码并传送到处理器中。

还有许多其他类型的用于人机交互的输入设备，包括触摸板、鼠标、操纵杆和轨迹球。这些设备经常作为图形输入装置与显示器一起使用。麦克风可以用来捕获音频输入，然后对其取样并转换为数字编码进行存储和处理。同样，摄像头可以用来捕获视频输入。

数字通信设施，例如互联网，也可以从其他计算机或数据库服务器向一台计算机提供输入。

### 1.2.2　存储器

存储器的功能是存储程序和数据。它分为主存储器和辅助存储器两种。

#### 1. 主存储器

主存储器（primary memory），也称为主存（main memory），是以电子速度运行的高速存储器。程序在执行时必须要存储在主存中。主存由大量的半导体存储单元（cell）组成，每一个存储单元能够存储一位二进制信息。这些单元很少被单独地读取或写入，而是按固定大小的组进行处理，这个组称为字（word）。主存这样组织以便于在一个基本操作中可以存储或检索一个字。每个字所包含的位数称为计算机的字长（word length），典型的字长有 16、32 或 64 位。

为了能方便地访问主存中的任何一个字，每个字单元都与一个不同的地址（address）相关联。地址是从 0 开始的，是用来识别逐个单元的连续数字。指明一个特定字的地址，然后向主存发出一条开始进行存储或检索过程的控制命令，就能够对该字进行访问了。

在处理器的控制下，指令和数据既可以写入主存也可以从主存中读出。在主存中能够尽快地访问到任意位置的字是非常重要的。其中的任何单元在指明了地址后都能在一个很短的固定时间内访问到的存储器，叫做随机访问存储器（random-access memory, RAM），访问一个字所需要的时间叫做存储器访问时间（memory access time）。这个时间与所访问的字的位置无关。当今 RAM 设备的访问时间通常在几纳秒（ns）至 100 纳秒之间。

#### 2. 高速缓存

作为主存储器的辅助手段，一个更小、更快的被称为高速缓存（cache）的 RAM 设备可用于存储目前正在执行的程序段以及所有相关的数据。高速缓存与处理器紧密耦合，它们通常被装在同一个集成电路芯片上。高速缓存的目的是为了提高指令的执行速率。

在程序开始执行的时候，高速缓存是空的。所有程序指令和任何所需的数据都存储在主存中。随着执行过程的推进，指令被读取到处理器芯片中，并且在高速缓存中存放每一条指令的副本。当指令的执行需要主存中的数据时，数据被取出并同时被拷贝到高速缓存中。

现在，假设许多指令在一段很短的时间内重复执行，比如程序中的循环语句。如果这些指令可从高速缓存中获得，那么它们可以在重复使用时被很快地取出来。类似地，如果相同的数据单元被反复访问，而这些内容的副本可从高速缓存中获得，那么它们就可以被很快地取出来。

#### 3. 辅助存储器

虽然主存储器是必需的，但它的价格也是昂贵的，而且断电后无法保存信息。因此在需要存储大量数据和程序，尤其是那些不经常访问的信息时，就会使用比较便宜的永久性辅助存储器（secondary storage）。辅助存储器的访问时间比主存储器的长。辅助存储设备有很多种，包括磁盘（magnetic disk）、光盘（optical disk，DVD 和 CD）和闪存设备（flash memory device）等。

### 1.2.3 运算器

大多数计算机的操作是由处理器的算术逻辑部件（arithmetic and logic unit，ALU）或运算
5 器执行的。任意的算术或逻辑运算，比如加、减、乘、除或比较大小，都是通过将所需的操作数送至由 ALU 执行运算的处理器中开始的。例如，如果将主存中放置的两个数求和，那么它们被送入处理器中，然后由 ALU 执行加法操作。所求得的和可能存储在主存中或保留在处理器中以便直接使用。

当操作数被送入处理器时，它们被存储在叫做寄存器（register）的高速存储单元中。每个寄存器可以存储一个字的数据。寄存器的访问时间比处理器芯片上的高速缓存的访问时间更短。

### 1.2.4 输出设备

输出设备与输入设备相对应。它的功能是向外界输出处理结果。这类设备中一个常见的例子是打印机（printer）。大多数打印机使用激光打印机中的复印技术或是喷墨流完成打印过程。这样的打印机每分钟至少可以打印 20 页。但是，打印机是机械设备，这样的速度和处理器的电子速度相比仍然是很慢的。

一些设备，比如图形显示器，既有显示文字与图形这样的输出功能，又有通过触摸技术实现的输入功能。这也是很多情况下对这种具有双重作用的设备使用单一名称输入/输出设备的原因。

### 1.2.5 控制器

存储器、运算器和输入/输出设备对信息进行存储和处理，然后执行输入和输出操作。这些设备的操作必须按照一定的方式互相协调，这就是控制器的职责。控制器是高效的中枢系统，它将控制信号传送到其他设备并检测它们的状态。

由输入和输出操作构成的 I/O 传输是由程序指令控制的，程序指令识别相关的设备和需要传输的信息。控制电路负责产生控制传输和决定何时发生规定动作的时序信号（timing signal）。处理器和存储器之间的数据传送也是由控制器通过时序信号控制的。于是有理由将控制器看成一个意义明确而且物理上完全独立的设备，它和计算机的其他部分相互作用。但在实际中情况却恰恰相反。很多控制电路分布于整个计算机中。大量的控制线（缆线）传递着所有部件中事件的时序和同步信号。

一台计算机的操作可以归纳如下：

- 计算机通过输入设备以程序和数据的形式接收信息，然后将其存储在存储器中。
- 在程序的控制下，存储在存储器中的信息被取出，然后送入运算器中进行处理。
- 经过处理的信息由输出设备送出计算机。
- 6 计算机内的所有活动都由控制器控制。

## 1.3 基本操作概念

在 1.2 节中，我们介绍了计算机的活动是由指令控制的。为了执行一个给定的任务，要在存储器中存储一个包含一连串指令的相应程序。完成特定操作的指令从存储器中取出，然后送入处理器中，用作指令操作数的数据也存储在存储器中。

一条典型的指令如下：

```
Load    R2, LOC
```

这条指令将读取地址标签 LOC 所指向的存储单元的内容，然后将其装入处理器寄存器 R2 中。LOC 单元中原来的内容被保存了下来，而寄存器 R2 中的原始内容被覆盖了。这条指令需要执行若干步。首先，指令从存储器中取出并被送入处理器中。然后，由控制器确定将要执行的操作，从存储器中取出 LOC 单元的操作数并送入处理器。最后，将操作数存储到寄存器 R2 中。

在操作数已经从存储器装入处理器寄存器后，就可以对它们进行算术或逻辑运算了。例如，指令

```
Add    R4, R2, R3
```

把寄存器 R2 与 R3 的内容相加，然后将它们的和放入寄存器 R4 中。R2 与 R3 中的操作数并未发生改变，但 R4 中先前的值被计算结果覆盖了。

完成所需的操作后，计算结果还在处理器寄存器中。可以通过使用如下指令将计算结果传送到存储器中

```
Store    R4, LOC
```

这条指令将寄存器 R4 中的操作数复制到存储单元 LOC 中。单元 LOC 中的原始内容被覆盖，但是 R4 的原始内容被保存了下来。

对 Load 和 Store 指令而言，存储器和处理器之间的传送从发送所需要的存储单元地址给存储器并发出适当的控制信号开始，然后将数据送入或送出存储器。

图 1-2 显示了存储器和处理器是如何连接在一起的，也显示了一些还没有讨论过的处理器部件。这些部件间的互连并没有明确表示出来，因为到此为止我们只讨论了它们的功能特性。第 5 章将详细描述作为处理器结构一部分的互连的细节。

除 ALU 和控制电路外，处理器中还包含许多用于不同目的的寄存器。指令寄存器（instruction register, IR）保存当前正在执行的指令。它的输出结果可由控制电路使用，以产生控制指令执行过程中不同处理部件的时序信号。程序计数器（program counter, PC）是另一个专用的寄存器。它包含下一条即将被读取和执行的指令的存储器地址。在一条指令的执行过程中，PC 中的内容相应地更新为下一条将被执行指令的地址。习惯上说 PC 指向（point）下一条将从存储器中取出的指令。除了 IR 和 PC，图 1-2 还给出了通用寄存器（general-purpose register）$R_0$ 到 $R_{n-1}$。通常也被称作处理器寄存器。它们的功能有很多，包括保存从存储器中载入的待处理的操作数。通用寄存器的作用将在第 2 章中详细解释。

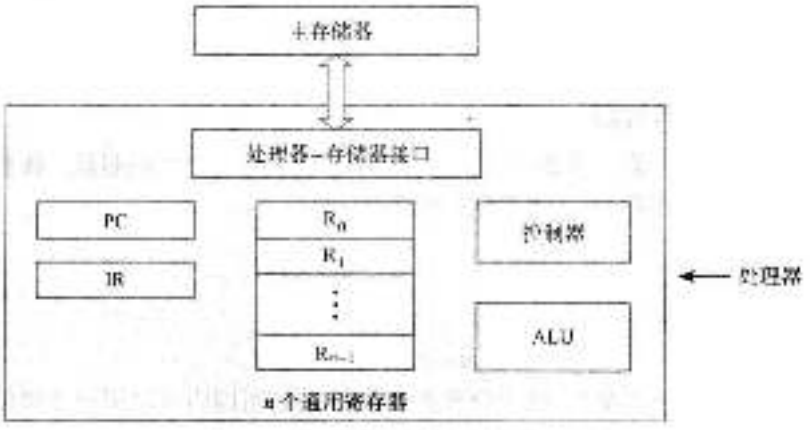



图 1-2　处理器和主存储器之间的连接

处理器-存储器接口是管理主存储器和处理器之间数据传输的电路。如果需要从存储器中读取一个字，该接口会向存储器发送该字的地址，同时送出一个读控制信号。接口等待要取回的字，然后将它传输到适当的处理器寄存器中。如果一个字需要被写入到存储器中，接口会向存储器传输该字的存储地址和内容，同时送出一个写控制信号。

现在来看一下典型的操作步骤。为了便于执行，程序必须存储在主存储器中，而且通常
8 是通过输入设备从辅助存储器传输到主存储器的。当 PC 指向程序的第一条指令时，程序开始执行。PC 中的内容连同一个读控制信号一起传送到存储器中。当被寻址的字（在这里是程序的第一条指令）从存储器中读出时，它被装入寄存器 IR 中。此时，指令便可以进行解释并执行了。

Load、Store 和 Add 等指令执行数据传输与算术操作。如果指令需要一个保存在存储器中的操作数，就需要通过将它的地址发送给存储器并启动一个读操作来获取。当这个操作数已经从存储器中读出时，它将被传送到处理器寄存器中。当操作数按照这种方式取出后，ALU 就可以对处理器寄存器中的值执行所需要的算术运算了，比如加法。运算结果会送到一个处理器寄存器中。如果这个结果将通过 Store 指令被写入存储器中，它会被从处理器寄存器传输到存储器中，存储该结果的单元地址也将送往存储器，然后开始一个写操作。

在每条指令执行过程中的某个点上，PC 中的内容递增，以便使 PC 指向下一条要执行的指令。这样，一旦当前的指令执行完毕，处理器就可以读取新的指令了。

除了在存储器和处理器之间传送数据外，计算机还从输入设备接收数据以及向输出设备输出数据，为此提供了一些处理 I/O 传送的机器指令。

当一些设备需要紧急服务时，程序的正常执行可能会被中断。例如，在一个计算机控制的工业流程中，监视器可能观察到了一个危险的情况。为了立即做出响应，必须暂停当前程序的执行。为此，设备发出一个中断（interrupt）信号，向处理器提出服务请求。处理器通过执行中断服务程序（interrupt-service routine）来提供所请求的服务。因为这样的转变可能会改变处理器的内部状态，所以必须在处理中断请求之前将处理器的状态保存在存储器中。通常，要保存的信息包括 PC 中的内容、通用寄存器的内容以及一些控制信息。当中断服务程序完成时，处理器的状态就从存储器中恢复，从而使得被中断的程序可以继续执行。

这一节对计算机的操作进行了简略介绍，在后面的章节中我们还将详细讨论这些概念。第 2、3、4 章将首先从程序员的角度来详细描述这些概念，而后面的章节将从硬件设计师的角度来详细描述这些概念。

## 1.4 数的表示及算术运算

在计算机系统中表示数的最基本方法是使用一串位，即一个二进制数。我们首先描述整
9 数的二进制表示及算术运算，然后将简单介绍浮点数的表示。

### 1.4.1 整数

考虑一个 $n$ 位的向量

$$B = b_{n-1} \cdots b_1 b_0$$

对于 $0 \leq i \leq n-1$，$b_i = 0$ 或 1。这个向量在 0 到 $2^n-1$ 的范围内可以用一个无符号整数 $V(B)$ 表示，这里

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

我们需要表示正数和负数，有三种编码系统可以用来表示这些数：

- 原码
- 反码
- 补码

在这三种编码体系中，最左边一位为“0”时表示正数，为“1”时表示负数。图 1-3 用 4 位数举例说明了在这三种体系中数字的表示方法。在所有的编码体系中正数的表示法相同，而负数有着不同的表示方法。在原码（sign-and-magnitude）系统中，负数值是将相应正数值的向量 $B$ 中的最高有效位（图 1-3 中的 $b_3$）由 0 变为 1 来表示的。例如：+5 表示为 0101，−5 表示为 1101。

在反码（1's-complement）表示中，负数值是通过将相应正数中的每一位求反而获得的。因此，−3 的表示就是通过对向量 0011 的各位求反得到了 1100。运用同样的按位求反操作也可以把一个负数转换成相应的正数。这种转换方法就称作给一个特定的数取反。对于 $n$ 位数，为其生成反码的操作相当于从 $2^n-1$ 中减去这个数，在图 1-3 中的 4 位数情况下也就是从 $2^4-1=15$ 或者二进制的 1111 中减去该数。

最后，在补码（2's-complement）系统中，一个 $n$ 位数的补码是从 $2^n$ 中减去这个数而得到的。因此，一个数的补码可以用这个数的反码加 1 而获得。

| $B$ | 所表示的值 | | |
|---|---|---|---|
| $b_3b_2b_1b_0$ | 原码 | 反码 | 补码 |
| 0 1 1 1 | +7 | +7 | +7 |
| 0 1 1 0 | +6 | +6 | +6 |
| 0 1 0 1 | +5 | +5 | +5 |
| 0 1 0 0 | +4 | +4 | +4 |
| 0 0 1 1 | +3 | +3 | +3 |
| 0 0 1 0 | +2 | +2 | +2 |
| 0 0 0 1 | +1 | +1 | +1 |
| 0 0 0 0 | +0 | +0 | +0 |
| 1 0 0 0 | −0 | −7 | −8 |
| 1 0 0 1 | −1 | −6 | −7 |
| 1 0 1 0 | −2 | −5 | −6 |
| 1 0 1 1 | −3 | −4 | −5 |
| 1 1 0 0 | −4 | −3 | −4 |
| 1 1 0 1 | −5 | −2 | −3 |
| 1 1 1 0 | −6 | −1 | −2 |
| 1 1 1 1 | −7 | −0 | −1 |

图 1-3　二进制有符号整数的表示

值得注意的是，在原码和反码系统中，“+0”和“−0”都有着不同的表示，而在补码系统中“0”只有一种表示。如图 1-3 所示，对 4 位数来说，在补码系统中能表示数值 −8，而在其他系统中却不能。原码系统看起来是最自然的，因为在笔算中我们就是用原码系统来处理十进制数的。反码系统很容易与这个系统关联起来，而补码系统可能显得有点不自然。但是，我们将会说明补码系统为实现加法和减法运算提供了最有效的方法，它是现代计算机中最常用的方法之一。

### 1. 无符号整数的加法

图 1-4 说明了 1 位数的加法。1 和 1 相加的和是一个 2 位的向量 10，表示数值 2。我们说

这个和（sum）是0，进位输出（carry-out）是1。为了完成多位数相加，我们使用一种类似于十进制数笔算中使用的方法。从这个位向量的最低位（右边）开始进行两位相加，并将进位传递到它的高位（左边）上去。某位上两个数相加所得的进位输出作为左边下一位的进位输入（carry-in）。进位输入必须与那个位置的两位相加，产生和以及进位输出。例如，如果某位的两个数都是1，并且进位输入也是1，则和为1，进位输出也为1，即表示数值3。

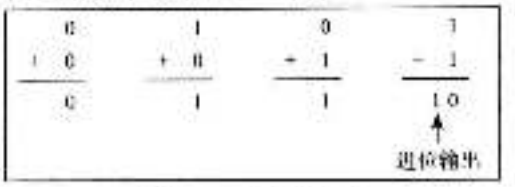



图1-4 1位数的加法

11

2. 有符号整数的加法和减法

我们介绍了表示正数和负数（或简称为有符号数（signed number））的二种系统。这些系统的不同之处仅在于对负数的表示方式。从是否容易执行算术运算的观点来看它们的优劣可以归纳如下。原码系统在表示上是最简单的，但是它对于执行加法和减法的运算是最不便的。反码表示法稍微好一些，补码系统对于执行加法和减法运算是最有效的。

为了理解补码的算术运算，考虑以$N$为模的加法（简写成mod $N$）。描述无符号整数mod $N$加法的有效图示方式是使用一个圆，沿着它的圆周标记上值0到$N-1$，如图1-5a所示。考虑当$N=16$的情况，如图1-5b所示。十进制数0到15可以用圆外侧的4位二进制数0000到1111表示。就十进制而言，(7+5) mod 16运算得到的值是12。我们使用图形执行这个运算，在圆外侧找到7(0111)的位置，然后按顺时针方向移动5个单位就到达了答案12(1100)处。类似地，(9 − 14) mod 16 = 7，即在圆上找到9(1001)，然后按顺时针方向移动14个单位越过0的位置就到达答案7(0111)处。对于任何无符号数$a$和$b$来说，这种图形技术对计算（$a+b$）mod 16都是有效的，即为了完成加法，定位$a$，然后按顺时针方向移动$b$个单位就可以得到$(a+b)$ mod 16的结果。

现在考虑对于模数为16的圆的另一种不同解释。我们按照圆内侧所示的补码表示将圆外侧的二进制向量重新解释成从−8到−7的有符号数。

将mod 16的加法技术应用在将−7加到−3的例子上。这两个数的补码表示分别是0111和1101。为了完成两数相加，在图1-5b的圆周上找到0111，然后按顺时针方向移动1101 (13)步到达了0100处，得到了+4的正确答案。值得注意的是，−3的补码表示被解释成一个无符号数以表示移动的步数。

如果采用从右到左按位加的方法完成这个加法，我们得到：

$$\begin{array}{cccccc} & & 0 & 1 & 1 & 1 \\ + & & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 0 & 0 \end{array}$$

进位

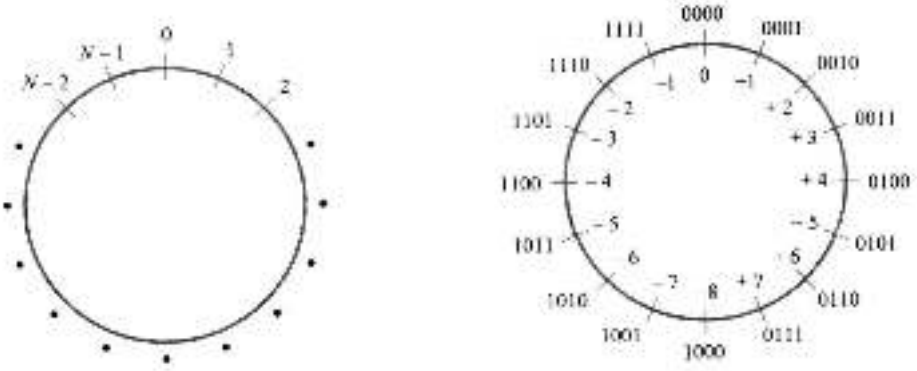
如果在这个加法中忽略第四位上的进位，就得到了一个正确的答案。实际上就是这样做的。忽略这个进位是对$N$取模运算的自然结果。当我们在图1-5b中绕着圆移动时，1111值的下一个值通常应该是10000，而我们回到了0000值。

12

使用补码表示系统的$n$位有符号数的加法和减法规则可以描述如下：

- 两个数相加（add）时，它们的$n$个表示位相加，忽略最高有效位（MSB）上的进位位。如果实际结果是在$-2^{n-1}$到$2^{n-1}-1$的范围之内，那么它们的和将是用补码表示的代数运算的正确值。
- $X$和$Y$两个数相减（subtract），也就是执行$X-Y$时，求出$Y$的补码形式，然后使用加

法规则将它加到 $X$ 中。同样，如果实际结果是在 $-2^{n-1}$ 到 $+2^{n-1}-1$ 的范围内，那么这个结果将是用补码表示的代数运算的正确值。




a）模数模 $N$ 的圆的表示　　b）二进制补码数的模为 16 的系统

图 1-5　模数系统和补码系统

在图 1-6 中给出了一些补码系统中加法和减法的例子。在所有这些 4 位数的例子中，结果都在 −8 到 +7 的表示范围。当结果没有在表示范围之内时，便发生了算术溢出（arithmetic overflow）。这种情况将在后面的小节中讨论。图 1-6 中 a 到 d 四个加法运算符合加法规则，e 到 j 六个减法运算符合减法规则。减法运算需要形成减数（算式中下面的值）的补码形式，不论减数为正数还是负数，这个运算都是以完全相同的方式完成的。要得到一个数的补码形式，需要对该数按位取反，然后加 1。

利用补码表示法简化了有符号数的加法和减法运算，这就是现代计算机采用补码表示法的理由。看起来似乎反码表示法与补码表示法一样有效，但是尽管数的求反很容易，可是加法运算后得到的结果却不能保证总是正确的。进位 $c_n$ 不能被忽略。如果 $c_n = 0$，得到的结果是正确的，如果 $c_n = 1$，必须对这个结果加上 1，以保证它的正确性。这个必需的修正操作意味着加法和减法运算在反码系统中不能像在补码系统中那样方便地实现。

**3. 符号扩展**

我们经常需要用更多的位数来表示一个用一定位数给出的值。对于一个正数，用左边加 0 的方法实现。对于一个用补码表示法表示的负数，最左边的位（表示数的符号）等于 1，具有同样值的较长位数的数可以通过在左边根据需要多次重复符号位来实现。为了理解为什么这样做是正确的，考查图 1-5b 中的模为 16 的圆。将它与比较大的圆，即模为 32 或模为 64 的情况做比较，值 −1，−2 等的表示完全相同，只是用若干个 1 加到了左边。概要地说，在补码形式中为了用更多的位数来表示一个有符号数，可以按照需要在左边多次重复它的符号位。这种操作称为符号扩展（sign extension）。

**4. 整数算术运算中的溢出**

使用补码表示法，$n$ 位可表示的值在 $-2^{n-1}$ 到 $+2^{n-1}-1$ 之间。例如，使用 4 位可以表示的数的范围是 −8 到 +7，如图 1-3 所示。当算术运算的实际结果超出了这个表示范围时，就发生了算术溢出。

当无符号数相加时，最高有效位上的进位输出 1 表明发生了溢出。但是当对有符号数相加时，这种方法并不总是正确的。比如，当使用补码表示 4 位有符号数时，如果我们将 +7 和 +4 相加，和向量是 1011，它表示数值 −5，是一个不正确的结果。在这种情况下，从 MSB（最

高有效位）位置上得到的进位输出位是0。如果我们将 −4 和 −6 相加，得到0110 = +6，也是一个错误的结果。在这种情况下进位输出位为1。因此，从符号位得到的进位输出位的值不能说明发生了溢出。显然，只有当两个加数有相同的符号时，才有可能产生溢出。具有不同符号的数相加时不会产生溢出，因为结果总会保持在可以表示的范围之内。

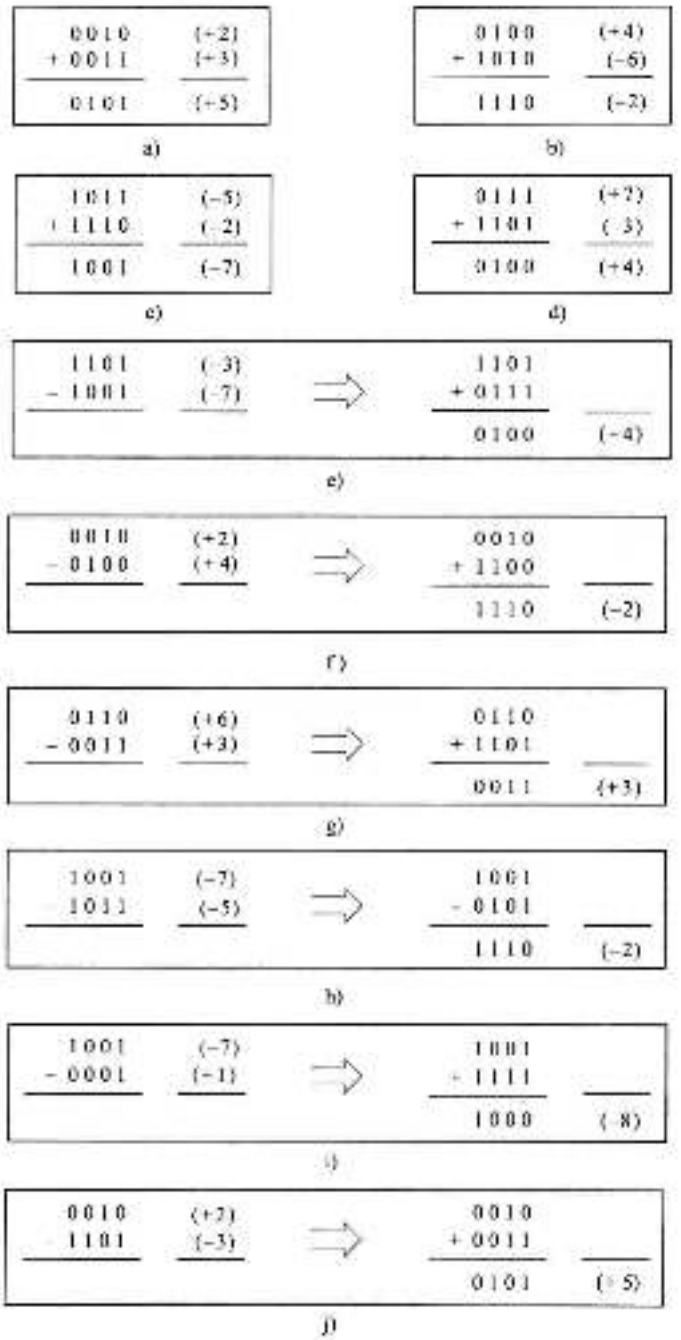



图 1-6 补码的加法和减法运算

这样我们可以得出以下对两个补码表示的数相加时检测溢出的方法。测试两个加数的符

号以及结果的符号。当两个加数具有相同的符号，而和的符号与加数的符号不同时就发生了溢出。

对两个数做减法的时候，检测器出的测试方法要做相应的修改，但是这个方法还是很简单的，参见习题 1.10。

14
15

## 1.4.2　浮点数

到目前为止我们只考虑了整数，它有一个隐含的二进制小数点在数的最右端，也就是 $b_0$ 位的后面。在字长为 32 位的计算机中，如果我们用一个全字去表示一个补码形式的有符号整数，数的表示范围是 $-2^{31}$ 到 $+2^{31}-1$。在十进制中，这个范围比 $-10^{10}$ 到 $+10^{10}$ 要小一些。

如果假设隐含的二进制小数点正好在符号位的右方，也就是 32 位表示中左端的位 $b_{31}$ 到 $b_{30}$ 之间，那么同样的 32 位模型还能表示在 $-1$ 到 $+1-2^{-31}$ 范围内的小数。在这种情况下，可以表示的最小的小数大约为 $10^{-10}$。

这两种定点（fixed-point）数表示法的表示范围对于许多科学和工程计算来说是不够的。为方便起见，我们希望能有一种二进制数表示法，这种方法能够容易地将非常大的整数和非常小的小数包含进来。要做到这一点，二进制小数点的位置应该是可变的，并且随着计算的进行可以自动调整，而计算机必须能够以这种形式表示数字并对其进行操作。在这种情况下，我们说二进制小数点是浮动（float）的，并称数字为浮点数（floating-point number）。

因为浮点数中二进制小数点的位置是可以变化的，所以在浮点表示中必须明确指出小数点的位置。例如，在我们熟悉的十进制科学记数法中，数字可以记为 $6.0247 \times 10^{23}$，$3.7291 \times 10^{-47}$，$-1.0341 \times 10^{2}$，$-7.3000 \times 10^{-4}$ 等。我们说这些数具有 5 位有效数字（significant digit）的精度。它们的比例因子（scale factor）$10^{23}$，$10^{-47}$，$10^{2}$ 和 $10^{-4}$ 指示了相对于有效数字的十进制小数点的实际位置。同样的方法也可以用来表示计算机中的二进制浮点数，只不过要用 2 来做比例因子的基数。因为基数是固定的，所以并不需要在表示中给出。指数可以是正数也可以是负数。

总的来说，二进制浮点数可以表示为：

- 数的符号
- 一些有效位
- 有符号的比例因子指数（隐含的基数为 2）

已经制定的表示 32 位浮点数的国际 IEEE（电气和电子工程师协会）标准使用 1 个符号位、23 个有效位以及 8 位比例因子的有符号指数（隐含的基数为 2）。在十进制中，所表示的数的范围大约在 $\pm 10^{-38}$ 到 $\pm 10^{38}$ 范围内，这对于大部分科学与工程计算来说足够了。IEEE 标准还定义了 64 位的表示，以提供更多的有效位和更多的有符号指数位，从而得到更高的精度和更大的取值范围。

浮点数表示和浮点数的算术运算将在第 9 章中详细介绍。附录 B 到 E 中描述的一些商用处理器在它们的指令集中包含了对浮点数的操作，并有一些专门用于保存浮点数的处理器寄存器。

16

## 1.5　字符表示

最常见的字符编码方案是 ASCII（美国信息交换标准代码）。字母数字字符、运算符、标点符号以及控制字符可以用表 1-1 所示的 7 位编码来表示。用一个 8 位的字节（byte）来表示和存储一个字符是很方便的。ASCII 编码占据较低的 7 位，而高位通常设置为 0。注意，当字

母与数字字符的编码被解释为无符号二进制数时，它们是按递增顺序排列的，这有利于字母和数字数据的排序操作。

十进制数字 0 到 9 的 ASCII 码的低 4 位是二进制数字系统的前十个值，这 4 位编码称为二进制编码的十进制（binary-coded decimal，BCD）码。

表 1-1 7 位 ASCII 码

| 位位置 | 位位置 654 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3210 | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0000 | NUL | DLE | SPACE | 0 | 表 | P | ' | p |
| 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0010 | STX | DC2 | " | 2 | B | R | b | r |
| 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 0100 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 0111 | BEL | ETB | ' | 7 | G | W | g | w |
| 1000 | BS | CAN | ( | 8 | H | X | h | x |
| 1001 | HT | EM | ) | 9 | I | Y | i | y |
| 1010 | LF | SUB | * | : | J | Z | j | z |
| 1011 | VT | ESC | + | ; | K | [ | k | { |
| 1100 | FF | FS | , | < | L | / | l | \| |
| 1101 | CR | GS | - | = | M | ] | m | } |
| 1110 | SO | RS | . | > | N | ^ | n | ~ |
| 1111 | SI | US | / | ? | O | — | o | DEL |

| | | | |
|---|---|---|---|
| NUL | 空 / 空闲 | SI | 移进 |
| SOH | 头部开始 | DLE | 数据连接出口 |
| STX | 文本码开始 | DC1-DC4 | 设备控制 |
| ETX | 文本的结尾 | NAK | 非应答 |
| EOT | 传输结束 | SYN | 同步空闲 |
| ENQ | 询问 | ETB | 传送的块结尾 |
| ACK | 应答 | CAN | 取消（数据错误） |
| BEL | 差听信号 | EM | 媒介结尾 |
| BS | 退格 | SUB | 专用序列 |
| HT | 水平制表符 | ESC | 出口 |
| LF | 换行 | FS | 文件分隔符 |
| VT | 垂直制表符 | GS | 组分隔符 |
| FF | 换页 | RS | 记录分隔符 |
| CR | 回车 | US | 单元分隔符 |
| SO | 移出 | DEL | 撤除 / 空闲 |

编码格式的位位置 - 6 5 4 3 2 1 0

## 1.6 性能

衡量一台计算机性能最主要的因素是看它执行程序的速度有多快。一台计算机执行程序的速度受到指令集、硬件和软件（包括操作系统和实现硬件的技术）设计的影响。由于程序通常是使用高级程序设计语言编写的，所以性能也会受到将程序转换成机器语言的编译器的影响。本书不描述编译器或操作系统的细节，但是第 4 章对软件进行了概述，包括编译器与操作系统的作用的介绍。本书只关注指令集、存储器、处理器与 I/O 硬件的设计，以及小型和大型

计算机的组织结构。1.2.2 节描述了高速缓存是如何提高存储器性能的。在第 2 章中还会讨论指令集性能方面的内容。在这一节里，我们将概括介绍技术，以及处理器和系统的组织结构是如何影响性能的。

### 1.6.1　技术

用于在单一芯片上制造处理器电子电路的超大规模集成电路（Very Large Scale Integration，VLSI）技术是影响机器指令执行速度的关键因素。在逻辑电路中 0 和 1 状态之间切换的速度很大程度上取决于组成电路的晶体管尺寸，小的晶体管切换速度更快。过去几十年中，制造技术的进步使得晶体管的尺寸大大减小。这有两个好处：可以使指令更快地执行；在一个芯片上可以放置更多的晶体管，从而实现更多的逻辑功能以及更大的存储器容量。

17
18

### 1.6.2　并行性

并行执行多项操作可以提高性能。并行性可以在不同的级别上实现。

**1. 指令级并行性**

在处理器中执行一个指令序列最简单的方法是在下一条指令的步骤开始之前完成当前指令的所有步骤。如果我们把连续指令的步骤重叠执行，那么总的执行时间将会减少。例如，在对当前指令的寄存器操作数进行算术运算的同时，可以从存储器中提取下一条指令。这种形式的并行称为流水线（pipelining）。流水线技术将在第 6 章中详细讨论。

**2. 多核处理器**

单个芯片上可以放置多个处理单元。在技术文献中，术语核心（core）用于表示芯片上的每一个处理单元，而术语处理器则指的是整个芯片。因此，用双核（dual-core）、四核（quad-core）和八核（octo-core）处理器等术语来分别表示有两个，四个和八个核心的芯片。

**3. 多处理器**

计算机系统中可能包含有许多个处理器，每个处理器又可能包含有多个核心。这种系统称为多处理器（multiprocessor）。这些系统可以并行执行许多不同的应用任务或者并行执行一个大型任务中的各个子任务。在这样的系统中，所有处理器通常都有权访问这个系统中的所有存储器，并且经常使用共享存储器多处理器（shared-memory multiprocessor）这一术语来明确表示这一点。这些多处理器系统的高性能使得复杂性和成本大大增加，这是由于使用了多个处理器和存储器单元以及更复杂的互连网络。

和多处理器系统相比，我们也可以将一组完整的计算机互连起来从而获得较高的总计算能力。这些计算机通常只访问它们自己的存储器，当它们正在执行的任务需要共享数据时，用通信网络交换消息（message）的方式来实现。这一特点将它们与共享存储器的多处理器区别开来，因此将它们命名为消息传递多计算机（message-passing multicomputer）。

多处理器和多计算机将在第 12 章中进行描述。

## 1.7　发展历程

自 20 世纪 40 年代以来，已开发出我们现在所知道的电子数字计算机。在电子计算机被发明之前有一段漫长的机械计算设备的发展过程。这里，我们只简要地描述一下计算机发展的历程。在 Hayes[1] 中可以找到更多的内容。

19

在 20 世纪中叶之前的 300 年里，一系列由齿轮、杠杆和滑轮构成的越来越复杂的机械装

置被用来执行基本的加、减、乘、除运算。穿孔卡片上的孔被机器感知后自动控制一系列的计算，这是当时提供的主要编程能力。这些设备可以计算完整的对数表和使用多项式近似的三角函数表，输出结果被穿孔在卡片上或打印在纸上。在20世纪30年代末40年代初，机电继电器设备，比如那些在早期电话转换系统中使用的设备，提供了计算机构造中执行逻辑功能的方法。

在第二次世界大战期间，第一台电子计算机在宾夕法尼亚大学设计制造完成，它使用了为无线电和军事雷达设备开发的真空管技术。真空管电路用来执行逻辑运算和存储数据。这项技术开创了电子数字计算机的新纪元。

计算机的处理器、存储器和I/O设备的制造技术的发展分为四代：第一代为1945～1955年；第二代为1955～1965年；第三代是1965～1975年；第四代是从1975年至今。

### 1.7.1 第一代计算机

程序存储的核心概念是在第一台电子数字计算机发明的时候提出的。程序和程序中使用的数据被放在同一个存储器中，就像今天这样。这使得改变现有的程序与数据或者准备和装入新的程序与数据之类的工作变得非常容易。汇编语言用来准备程序并被转换成机器语言以便于执行。

如果使用真空管技术实现逻辑功能，基本算术运算只需要执行几毫秒。这比早期的机械和机电技术的速度要快100～1000倍。最初使用的是水银延迟线存储器，而I/O功能是由类似打字机的设备完成的，还开发出了磁心存储器和磁带存储设备。

### 1.7.2 第二代计算机

在20世纪40年代后期，AT&T贝尔实验室（AT&T Bell Laboratories）发明了晶体管，并迅速用它取代了真空管来实现逻辑功能。这一基本技术的转变标志着第二代计算机的开始。在第二代计算机中广泛使用了磁心存储器和磁鼓存储设备，磁盘存储设备也在这一时期被开发出来。还开发出了最早的高级语言，例如Fortran，使得应用程序的制作更为容易。编译器也被开发出来，它将高级语言程序翻译成汇编语言程序，然后汇编语言程序再被翻译成可执行的机器语言形式。在这一时期，IBM公司成为主要的计算机制造商。



### 1.7.3 第三代计算机

德州仪器（Texas Instruments）与仙童半导体（Fairchild Semiconductor）公司发明了在一个独立的硅芯片上建造许多晶体管的技术，称之为集成电路技术，利用该项技术可以建造速度更快、成本更低的处理器和存储元件。集成电路存储器开始取代磁心存储器。这一技术的发展标志着第三代计算机的开始。在这个时期还发展了其他技术，诸如微程序设计、并行性和流水线技术。操作系统软件可以使若干个用户程序有效地分享计算机系统。还开发出了高速缓存和虚拟存储器技术。高速缓存使得主存储器看起来比实际要快，而虚拟存储器使得主存储器看起来比实际要大。IBM公司的System 360大型机和数字设备（Digital Equipment）公司的PDP系列小型机主宰了第三代计算机的商用产品市场。

### 1.7.4 第四代计算机

到20世纪70年代初，集成电路制造技术已经达到小型计算机中的整个处理器和大部分

主存储器都可以在单个芯片上实现的程度。这标志着第四代计算机的开始。数以万计的晶体管可以放置在一个芯片上，超大规模集成电路（VLSI）这一名词描述了这项技术。一个完整的处理器可以在一个芯片上制造，这就是微处理器。一些公司如英特尔（Intel）、国家半导体（National Semiconductor）、摩托罗拉（Motorola）、德州仪器（Texas Instruments）和超微半导体（Advanced Micro Devices，AMD）是这一技术的推动者。目前的 VLSI 技术能够使多个处理器（核心）和高速缓存集成到单个芯片上。

超大规模集成电路技术的一种特殊形式——现场可编程门阵列（Field Programmable Gate Array，FPGA），使系统开发者能够在单一芯片上设计和实现处理器、存储器和 I/O 电路，以满足特定应用的要求。尤其是在嵌入式计算机系统中。先进的计算机辅助设计工具使人们可以迅速地开发基于 FPGA 的产品。Altera 和 Xilinx 等公司提供了这项技术以及所需要的软件开发系统。

嵌入式计算机系统、便携式笔记本电脑以及多功能的移动电话现在被广泛使用。台式个人计算机和工作站通过有线的或无线的局域网和因特网互连起来，可以访问数据库服务器和搜索引擎，从而提供了多种强大的计算平台。

当第四代计算机成熟时，诸如并行性、分级存储器等组织结构的概念被用在了当今的高性能计算系统的生产中。在高性能计算高端领域的超级计算机和网络计算机，被用于天气预报、科学与工程计算以及仿真等应用中。

## 1.8　结束语

这一章介绍了计算机结构和操作方面的基本概念，简要地描述了机器指令和程序，并对二进制数的加法和减法进行了解释。对许多与这些主题有关的术语都给出了定义。后面的章节将对这些术语和概念做出详细的解释，并将重点阐述体系结构和硬件。

## 1.9　问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。

**例 1.1**

**问题：** 按照图 1.2 中所示的部件之间的数据传输以及一些简单的控制命令，列出执行如下机器指令所需的步骤：

```
Load   R2, LOC
```

1.3 节中给出了所需步骤的概述。假设包含该指令的存储单元的地址最初是在寄存器 PC 中。

**解答：** 所需的步骤是：

- 将指令字的地址从寄存器 PC 发送到存储器中，并发出一个读控制命令。
- 等待所请求的字从存储器中取回，然后把它装入寄存器 IR 中，它在 IR 中被控制电路解释（或译码）以便确定所需执行的操作。
- 递增寄存器 PC 的内容以指向存储器中的下一条指令。
- 将地址值 LOC 从寄存器 IR 中的指令发送到存储器并发出一个读控制命令。
- 等待所请求的字从存储器中取回，然后将其装入寄存器 R2 中。

**例 1.2**

**问题：** 一个程序总共有 500 条指令，其中有一个包含 100 条指令的循环需要执行 25 次。在这种情况下，估算一下使用高速缓存对性能所产生的影响。计算没有高速缓存与有高速缓存时程序执行时间的比值，这个比值称为加速比（speedup）。

假设访问主存储器需要 10 个单位的时间，而访问高速缓存需要 1 个单位的时间。为了方便说明使

用高速缓存的优势，我们还做出如下的假设以简化计算：

- 程序的执行时间与从主存储器或高速缓存中提取指令所需的总时间量成比例。忽略访问操作数数据的时间。
- 最初，所有指令都存储在主存储器中，高速缓存是空的。
- 高速缓存的容量足以包含所有的循环指令。

**解答**：没有高速缓存时的执行时间为：

$$T = 400 \times 10 + 100 \times 10 \times 25 = 29\,000$$

有高速缓存时的执行时间为：

$$T_{cache} = 500 \times 10 + 100 \times 1 \times 24 = 7\,400$$

所以，加速比为

$$T/T_{cache} = 3.92$$

**例 1.3**

**问题**：将以下各对十进制数转换成5位补码数，然后对每一对进行加法和减法运算。指出每一种情况中是否会发生溢出。

(a) 7 和 13

(b) −12 和 9

**解答**：转换与运算如下：

(a) $7_{10} = 00111_2$，$13_{10} = 01101_2$。

把这两个正数相加我们得到 10100，这是一个负数，因此发生了溢出。

要把这两个数相减，我们首先需要求出 01101 的补码 10011。然后再将其与 00111 相加得到 11010，也就是 $-6_{10}$，这是正确的答案。

(b) $-12_{10} = 10100_2$，$9_{10} = 01001_2$。

把这两个数相加，得到 $11101 = -3_{10}$，这是正确的答案。

要把这两个数相减，我们首先需要求出 01001 的补码 10111，然后再对两个负数 10100 和 10111 进行加法运算得到 01011，这是一个正数，因此发生了溢出。



## 习题

[E] 1.1 对在 1.3 节中讨论的机器指令

```
Add   R4, R2, R3
```

重复例 1.1。

[E] 1.2 对在 1.3 节中讨论的机器指令

```
Store   R4, LOC
```

重复例 1.1。

[M] 1.3 (a) 对任务“将存储单元 A 和 B 中的内容相加，并将结果放到单元 C 中”给出一个机器指令的短序列。指令

```
Load   Ri, LOC
```

及

```
Store   Ri, LOC
```

是用于在存储器和通用寄存器之间传递数据的唯一可用指令。Add 指令在 1.3 节中进行了描述。不要改变单元 A 和 B 中的内容。

(b) 假设 Move 和 Add 指令的可用格式为：

```
Move   Location1, Location2
```

和

```
Add   Location1, Location2
```

这些指令将在第二个位置 Location2 中的操作数副本传送或添加到第一个位置 Location1 中，

覆盖第1个位置中原有的操作数内容。两个操作数或其中的一个可以在存储器中也可以在通用寄存器中。是否可以使用较少的这种类型的指令完成（a）部分中的任务？如果可以，请给出指令序列。

[M] 1.4（a）某程序共有300条指令，其中有一个包含50条指令的循环需要执行15次。处理器有一个高速缓存，如1.2.2节所描述。在主存储器中取出与执行一条指令需要20个时间单位。如果在高速缓存中能找到该指令，则取出与执行该指令只需要2个时间单位。忽略访问操作数数据的时间，计算没有高速缓存与有高速缓存情况下程序执行时间的比值。由于使用了高速缓存，所以这个比值称为加速比。假设高速缓存最初是空的，且它的容量足以包含这个循环的所有指令，而程序开始时所有指令都存储在主存储器中。

（b）把（a）部分中的常数300、50、15、20和2用变量w、x、y、m和c替换，写出加速比的表达式。

（c）如果w=300，x=50，m=20，c=2，y取什么值可以使加速比为5？ 24

（d）考虑（b）部分中得到的加速比表达式的形式，当循环迭代的次数y越来越大时，加速比的上限是多少？

[M] 1.5（a）在1.2.2节中讨论过处理器高速缓存。假设一个程序的执行时间与指令提取时间成比例，再假设从高速缓存中取出一条指令要花费1个时间单位，但从主存中取出一条指令要花费10个时间单位。此外，假设所请求的指令在高速缓存中找到的概率为0.96。最后，假设如果一条指令在高速缓存中没有找到，必须首先将它从主存储器中读取出来送到高速缓存中，然后再从高速缓存中取出来执行。计算没有高速缓存与有高速缓存情况下程序执行时间的比值。由于使用了高速缓存，所以这个比值称为加速比。

（b）如果高速缓存的大小加倍，同时假设在其中找不到所请求指令的概率减半。对于双倍大小的高速缓存重复（a）部分中的问题。

[E] 1.6 扩充图1-4，使图中所示的4种情况都能包含进位输入的两种可能性（0或1）。列举出8种新情况的和位与进位输出位。

[M] 1.7 将以下各对十进制数转换成5位补码数，然后将它们相加。判断在每种情况中是否会发生溢出。

（a）4和11

（b）6和14

（c）−13和12

（d）−4和8

（e）−2和−9

（f）−9和−14

[M] 1.8 重复习题1.7的描述，对各对十进制数做减法运算，即从每对数的第一个数中减去第二个数。判断在每种情况中是否会发生溢出。

[M] 1.9 一个字节存储单元中包含模式01010011，当这个模式被解释为二进制整数时，它所表示的十进制数值是多少？如果被解释为ASCII码它又表示什么？

[M] 1.10 1.4.1节末尾给出了当两个补码数相加时的一种检测溢出的方法。描述一下当这两个数相减时如何对溢出进行检测。

## 参考文献

1. J. P. Hayes, *Computer Architecture and Organization*, 3rd Ed., McGraw-Hill, New York, 1998.

25 / 26

第2章

Computer Organization and Embedded Systems, Sixth Edition

# 指令集体系结构

## 本章目标

在本章中你将学习以下内容：

- 机器指令和程序的执行
- 访问寄存器和存储器操作数的寻址方式
- 用于表示机器指令、数据和程序的汇编语言
- 堆栈和子程序



这一章我们从机器指令集的角度来考虑程序在计算机中的执行方法。第1章已经介绍了存储在存储器中的程序指令和数据操作数的一般概念，在本章中我们将讨论指令是如何组成的，学习指令序列从存储器传递到处理器并完成给定任务的方法，还将介绍用来访问存储单元和处理器寄存器中操作数的常用寻址方式。

这里强调的是基本概念。我们用一种通用的方式来描述机器指令和商用处理器中典型的操作数寻址方式。本章将介绍足够的指令和寻址方式，以便于能够给出一个针对简单任务的完整且真实的程序。这些通用程序用汇编语言进行了说明，其中机器指令和操作数寻址信息用符号名来表示。包括操作数寻址方式在内的完整的指令集通常被称为处理器的指令集体系结构（instruction set architecture，ISA）。在本章中我们对基本概念进行讨论，不需要定义完整的指令集，我们也不会去尝试这样做，而是给出足够的例子来说明一种典型指令集的功能。

在本章和第3章介绍的概念涉及输入/输出技术，这些概念对于理解计算机的功能是非常必要的。我们选择了一种通用的表示方式使得内容易于阅读和理解。同时，这种方式允许进行一般性的讨论而不受特定处理器特性的限制。

这些概念是怎样在真实的计算机上实现的呢？这很有趣而且非常重要，因此在第2章和第3章中，我们会提供四个流行的商用处理器的例子。这些处理器在附录B到E中进行介绍。附录B讨论Altera公司的Nios II处理器。附录C介绍Freescale半导体公司的ColdFire处理器。附录D讨论ARM公司的ARM处理器。附录E介绍Intel公司的处理器基本体系结构。在第2章和第3章出现的通用程序在每一个附录中都会用具体的指令集表示出来。

读者可以选择一种处理器来学习相应附录中的内容，以了解商用的ISA设计。但是，这些附录中的知识对于本书主体内容的理解来说不是必要的。

大多数的程序是用高级语言编写的，比如用C、C++或Java。为了在处理器上执行高级语言程序，这个程序必须首先被翻译成该处理器上的机器语言，这由编译程序完成。汇编语言是机器语言的一种易读的符号表示形式。在本书里我们会大量使用汇编语言，因为这是描述计算机工作方式的最好方法。

本章中，我们将首先讨论指令和数据是如何存储在存储器中的以及如何访问它们以进行处理。

## 2.1 存储单元和地址

我们首先来考虑计算机的存储器是如何构成的。存储器是由几百万个存储单元（cell）构成的，其中每个单元可以存储一位（bit）具有0值或1值的信息。由于单独的一位只表示信

息中一个非常小的量，所以很少单独对位进行处理。常用的方法是按固定大小的组对位进行处 28
理。为此，存储器被组织成适合在一个基本操作中对 $n$ 位一组的信息进行存储或检索的形式。每一个 $n$ 位组称为一个字（word），$n$ 称为字长（word length）。计算机的存储器可以用图表方式表示成字的集合，如图 2-1 所示。

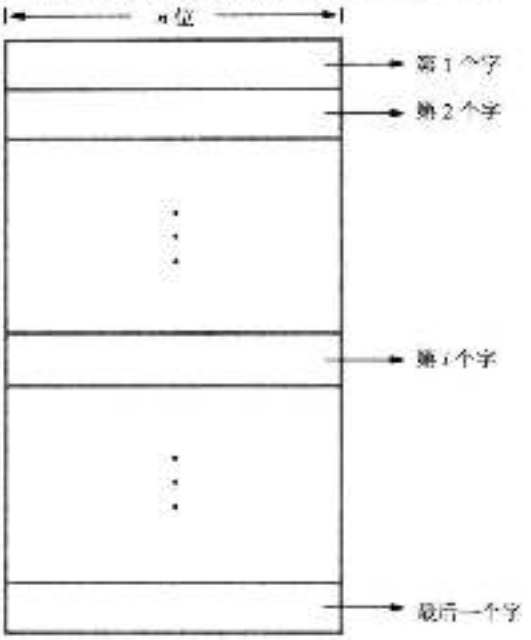



图 2-1　存储器中的字

现代计算机的字长范围一般是从 16 位到 64 位。如果一台计算机的字长是 32 位，那么如图 2-2 所示，一个单独的字就能够存储一个 32 位的有符号数或是四个各占 8 位的 ASCII 编码字符。一个 8 位的单元叫做一个字节（byte）。机器指令可能需要用一个或多个字来表示。在描述了汇编语言级的指令后，在后面一节中我们将讨论机器指令是如何被编码到存储器的字中的。

为了存储或检索一个信息项需要访问存储器，该信息项无论是一个字或是一个字节，对于每一项的位置都要有具体的名字或是地址（address）。习惯上我们用从 0 到 $2^k-1$（$k$ 取某个适当的值）的数字作为存储器连续单元的地址。因此存储器可以有高达 $2^k$ 个可寻址单元。$2^k$ 个地址构成了计算机的地址空间（address space）。例如，一个 24 位的地址生成一个具有 $2^{24}$ (16 777 216) 个存储单元的地址空间。这个数通常写成 16M，这里 1M 表示数 $2^{20}$(1 048 576)。一个 32 位的地址创建出具有 $2^{32}$ 或者 4G 个存储单元的地址空间，这里 1G 表示的是 $2^{30}$。其他 29
的惯用表示法就是用 K 来表示数 $2^{10}$(1 024)，用 T 表示数 $2^{40}$。

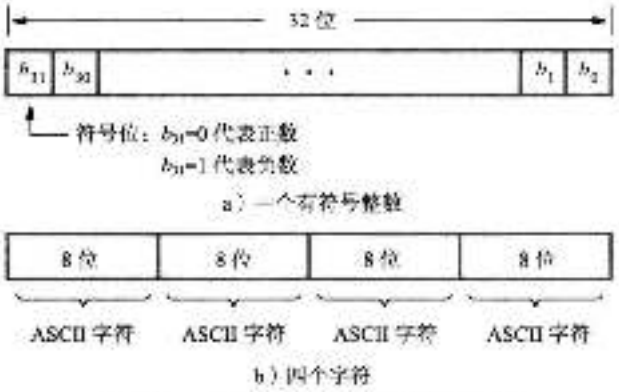



图 2-2　一个 32 位字中编码信息的例子

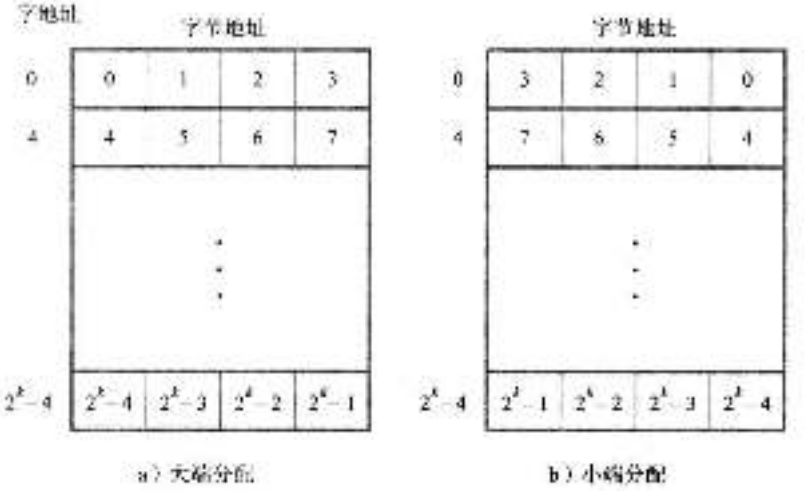
## 2.1.1　按字节寻址能力

现在有三种基本的信息处理量：位、字节和字。一个字节通常为 8 位，但字长通常的范围是从 16 位到 64 位。为存储器中的每一位分配不同的地址是不切实际的做法。实际上大部分的分配方法是将连续的地址对应于存储器中连续的字节单元。这是大多数现代计算机中使用的分

配方法。按字节寻址存储器（byte-addressable memory）就使用了这种分配方法。字节单元具有地址 0，1，2，…，这样，如果机器的字长为 32 位，那么连续的字被分配到地址 0，4，8，…中，其中每个字包含四个字节。

### 2.1.2 大端和小端分配

如图 2-3 所示，有两种在字中分配字节地址的方法。当低字节地址作为一个字中的最高有效字节（最左边字节）时采用大端（big-endian）方法。而小端（little-endian）方法用于相反的次序中，其中低字节地址作为一个字中的最低有效字节（最右边字节）。"最高有效"和"最低有效"是指当这个字表示一个数时它们在分配位中所占的权重（以 2 为权）。小端和大端两种
30 分配法都在商业计算机中使用。在这两种情况，都将字节地址 0，4，8，… 作为 32 位字长计算机的存储器中连续字的地址。这些地址也可以用作访问存储器以存储或检索一个字时的地址。




a）大端分配　　b）小端分配

图 2-3　字节和字的寻址

在一个字中除了指明字节地址排序外，还需要说明每一位在一个字节或一个字中的标记。最常用的也是我们在本书中采用的方法在图 2-2a 中给出，它是数字型数据编码最常用的排序方法。这种排序法也可以用于在字节中标记位，即从左到右为 $b_7$，$b_6$，…，$b_0$。

### 2.1.3 字的对齐

在 32 位字长的情况下，如图 2-3 所示，自然字的边界发生在地址 0，4，8，…上。如果字位置的开始处在一个字节地址上，这个地址又是一个字中字节数的整数倍，那么我们说这些字的位置具有对齐（aligned）地址。因为实际中会涉及二进制码地址的操作，一个字的字节数是 2 的幂次方，因此，如果字长是 16 位（2 个字节），对齐字从字节地址 0，2，4，… 开始，而如果字长是 64 位（$2^3$ 个字节），对齐字从字节地址 0，8，16，… 开始。

没有一个基本原则性的约定规定字不能从任意一个字节地址开始。如果一个字可以从任意的字节地址开始，这些字就是不对齐（unaligned）地址。但是，在大多数情况下使用的是对
31 齐的地址，我们将会在第 8 章中看到，这会使得存储器操作数的访问操作更加高效。

### 2.1.4 访问数和字符

一个数通常占用一个字，在存储器中可以通过指明字的地址对其进行访问。同样，对于

单个字符也可以通过它们的字节地址进行访问。

为了编程方便，用不同的方法在程序指令中指定地址是非常有益的。我们将会在 2.4 节中讨论这个主题。

## 2.2　存储器操作

程序的指令和数据操作数都存储在存储器中。要执行一条指令，处理器控制电路必须要将包含这条指令的单个字（或多个字）从存储器传送到处理器中。操作数和操作结果也必须能够在存储器和处理器之间传送。因此，有关存储器的两个基本操作是必需的，即 Read（读）和 Write（写）。

Read 操作是指将一个指定存储单元中的内容拷贝传送到处理器中。存储器中的内容保持不变。在开始 Read 操作时，处理器向存储器发送所需单元的地址并且要求读取它的内容。存储器读出存储在那个单元中的数据并将其发送给处理器。

Write 操作是从处理器向一个指定的存储单元传送一条信息，它将覆盖这个单元中原有的内容。为了初始化 Write 操作，处理器向存储器发送一个所要求的单元地址，同时发送将要写入这个单元中的数据。然后存储器使用这个地址和数据来完成写操作。

有关这些操作的硬件实现细节将在第 5 章和第 6 章中论述。在本章中我们从 ISA 的角度来考虑所有的操作，集中讨论指令和操作数的逻辑处理。

## 2.3　指令和指令序列

一个计算机程序所执行的任务是由一系列小的执行步骤构成的，比如像两个数相加、测试特定的条件，从键盘上读一个字符或是发送一个字符到显示屏上去显示等。一台计算机必须具备能够执行以下四种类型操作的指令：

- 存储器和处理器寄存器之间的数据传送
- 数据的算术和逻辑运算
- 程序序列化和控制执行
- 输入 / 输出传送

我们首先讨论前两种类型的指令。为了便于讨论，需要先介绍一些标记符号。　[32]

### 2.3.1　寄存器传送标记

我们需要描述信息在计算机中从一个单元传送到另一个单元。在这种传送中涉及的单元可以是存储器中的单元、处理器寄存器或是 I/O 子系统中的寄存器。大多数情况下，我们用方便的符号名来识别这些单元的位置。例如，代表存储单元地址的名字可能是 LOC、PLACE、A 或者 VAR2。处理器寄存器的预定义名可能是 R0 或者 R5，而 I/O 子系统中的寄存器可能由像 DATAIN 或者 OUTSTATUS 这样的名字来识别。为了描述信息的传送，任何一个单元中的内容用它的名字外加方括号来表示。因此表达式

$$R2 \leftarrow [LOC]$$

表示存储器单元 LOC 中的内容被传送到处理器寄存器 R2 中。

另一个例子是将寄存器 R2 和 R3 的内容相加，并将它们的和放到寄存器 R4 中。这个动作表示为：

$$R4 \leftarrow [R2] + [R3]$$

这种标记方式就是所谓的寄存器传送标记（Register Transfer Notation，RTN）。要注意的是，在

RTN 表达式的右边总是表示一个值，而左边是存放这个值的单元名，将用这个值覆盖该单元中的原有内容。

在计算机术语中，“传送”和“移动”常用于表示“复制”的意思。从源（source）单元 A 传输数据到目的（destination）单元 B 意味着读取单元 A 的内容，然后将其写到单元 B 中。在这个操作中，只有目的单元的内容被改变，源单元的内容将保持不变。

### 2.3.2 汇编语言符号

我们需要用另一种标记符号来表示机器指令和程序。为此，我们使用汇编语言（assembly language）。例如，将存储单元 LOC 的内容传送到处理器寄存器 R2 中，产生这种传送操作的通用指令将使用下面的语句来说明：

```
Load   R2, LOC
```

执行这条指令后 LOC 的内容不改变，但寄存器 R2 中原有的内容被覆盖了。名字 Load 对于这条指令是恰当的，因为从存储单元中读取的内容被装入（load）到处理器的寄存器中。

第二个例子是将处理器寄存器 R2 和 R3 中的两个数相加并将得到的和放到 R4 中，该例子可以用汇编语言的语句描述为：

```
Add    R4, R2, R3
```

33 在本例中，寄存器 R2 和 R3 保存着源操作数，而 R4 则为目的寄存器。

一条指令（instruction）指定了一个将要执行的操作及其包含的操作数。在上面的例子中，我们用英语单词 Load 和 Add 表示所需的操作。在实际的（商用）处理器的汇编语言指令中，这样的操作是用助记符（mnemonic）来定义的，助记符通常是描述这些操作的单词的缩写。例如，Load 操作可能写成 LD，而将一个字从处理器寄存器传送到存储器的 Store 操作，则可能被写成 STR 或 ST。对于给定的操作，不同处理器的汇编语言通常使用不同的助记符。为了避免在这个较早的阶段了解特定汇编语言的细节，在本章中我们将会继续使用英语单词来描述，而不使用特定处理器的助记符。

### 2.3.3 RISC 和 CISC 指令集

区分不同计算机的一个最重要的特征是其指令的本质。在现代计算机的指令集设计中，有两种根本不同的方法。其中一种流行的方法所依据的前提是，如果每条指令恰好占据存储器中的一个字，则可以获得更高的性能，并且执行指令指定的算术或逻辑运算所需要的所有操作数都已经在处理器的寄存器中。这种方法有利于处理单元的实现，其中处理指令序列所需要的各种操作可以用流水线的方式重叠执行，从而减少程序的总执行时间，这将在第 6 章中讨论。每条指令必须恰好放入一个单字中的限制降低了这些指令的复杂度，减少了计算机指令集中所包含的不同指令类型的数目。这样的计算机被称为精简指令集计算机（Reduced Instruction Set Computer，RISC）。

相对于 RISC 方法的另一种方法是使用更复杂的指令，这些指令可能跨越多个存储器字，但可以指定更复杂的操作。这种方法在 20 世纪 70 年代 RISC 方法推出之前是很盛行的。尽管复杂指令的使用最初没有确定任何特定的称呼，但是基于这个思想的计算机后来被称为复杂指令集计算机（Complex Instruction Computer，CISC）。

由于 RISC 风格的指令集比较简单，并且易于理解，所以我们将首先介绍 RISC 风格的指令集，然后再讨论 CISC 风格的指令集并说明这两种方法之间的主要区别。

### 2.3.4　RISC 指令集介绍

RISC 指令集的两个关键特性是：

- 每条指令为一个字长
- 使用 load/store 体系结构（load/store architecture），其中
  - 只能通过 Load 和 Store 指令来访问存储器操作数。
  - 算术或逻辑运算中涉及的所有操作数都必须在处理器寄存器中，或者其中一个操作数在指令字中被明确地给出。

34

在程序运行之初，程序中用到的所有指令和数据都存储在计算机的存储器中。此时，处理器寄存器中并没有包含有效的操作数。如果希望操作数在指令使用之前就已经在处理器的寄存器中，那么就有必要先把这些操作数放到寄存器中。这个任务由 Load 指令完成，Load 指令会将存储单元中的内容复制到处理器寄存器中。Load 指令的格式为：

Load　目的操作数，源操作数

或者更具体一点：

Load　处理器寄存器，存储单元

存储单元可以用若干种方法指定。寻址方式（addressing mode）就是指能够实现这一目的的不同方式，我们将在 2.4 节中讨论。

现在我们来考虑一个典型的算术运算。两个数相加的操作是任何一台计算机的基本功能。高级语言程序中的语句

C = A + B

指示计算机将变量 A 和 B 中的当前值相加，并将得到的和赋值给第三个变量 C。当包含这个语句的程序被编译时，三个变量 A、B、C 被分配到了存储器的不同单元中。为简单起见，我们将这些单元的地址分别称作 A、B 和 C。这些单元的内容表示这三个变量的值。因此，上述高级语言语句需要在计算机中完成的动作是：

C ← [A] + [B]

为了执行这个动作，从存储器中取出存储单元 A 和 B 中的内容并将其传送到处理器中，在那里对它们进行求和计算。然后将计算结果传送回存储器并存储在单元 C 中。

上述语句所需的动作可以通过一系列简单的机器指令来完成。我们选择使用寄存器 R2、R3、R4 以及四条指令来执行这个任务。

```
Load    R2, A
Load    R3, B
Add     R4, R2, R3
Store   R4, C
```

我们称 Add 为三操作数（three-operand）指令或三地址（three-address）指令且具有以下格式：

Add　目的操作数，源操作数 1，源操作数 2

Store 指令具有以下格式：

Store　源操作数，目的操作数

其中源操作数为处理器寄存器，目的操作数为存储单元。需要注意的是，Store 指令中源操作数和目的操作数的定义顺序与 Load 指令相反，这是一般的使用惯例。

35

注意，如果其中一个源操作数寄存器也同时用作存放结果的目的寄存器，那么只需要使用两个寄存器 R2 和 R3 就可以完成所需的加法。在这种情况下加法可以按如下方式执行：

Add　R3, R2, R3

最后一条指令将变为

Store R3, C

### 2.3.5 指令执行和线性序列

在前面的小节中，我们使用任务 C = A + B 作为例子，C = A + B 实现为 C ← [A] + [B]。图 2-4 给出了完成这个任务的程序段存放在计算机存储器中的情况。假定计算机的字长为 32 位并且存储器是按字节寻址的。程序中的四条指令放在连续的字单元中，起始单元是 $i$。由于每条指令是 4 字节长，所以第二、第三和第四条指令的起始地址分别为 $i + 4$、$i + 8$ 和 $i + 12$。为了简单起见，假定所需的存储器地址都能够在 Load 和 Store 指令中直接说明，虽然这在包含一个完整的 32 位地址时是不可能的。我们将在 2.4 节中解决这个问题。

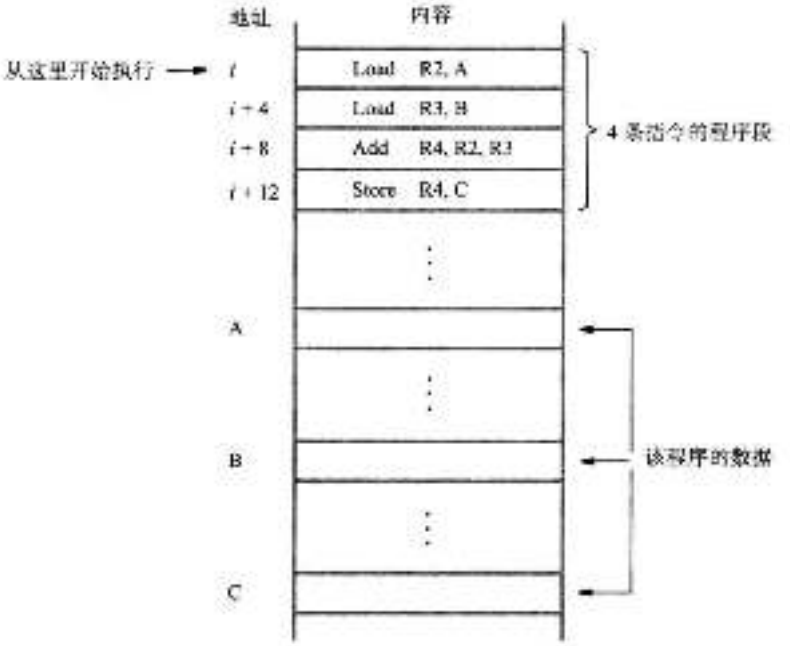



图 2-4 一个完成 C ← [A] + [B] 的程序

我们来考虑这个程序将如何执行。处理器中包含一个称为程序计数器（program counter，PC）的寄存器，它保存着将要执行的下一条指令的地址。当开始执行一个程序时，它的第一条指令地址（在本例中为 $i$）必须被放入 PC 中。然后，处理器的控制电路利用 PC 中的这个信息，按照地址递增的顺序一次一条地取出并执行指令。这种执行方式叫做线性序列（straight-line sequencing）。在每条指令执行时，PC 每次递增 4 以指向下一条指令。这样当 $i + 12$ 单元中的 Store 指令被执行后，PC 中包含的值是 $i + 16$，它是下一个程序段的第一条指令的地址。

执行一条给定的指令可以分成两个阶段。第一个阶段叫做取指令（instruction fetch）阶段，在该阶段中，根据 PC 中的地址从存储单元中取出该指令。这条指令被放入处理器的指令寄存器（instruction register，IR）中。第二个阶段叫做指令执行（instruction execute）阶段，在该阶段开始时，对 IR 中的指令进行检查以确定将要执行哪种操作。然后处理器执行这个指定的操作。这个执行过程涉及少量步骤，如从存储器或处理器寄存器中提取操作数，执行一个算术或逻辑运算，然后将结果存放到目的单元中。在这个两阶段过程中的某个点上，PC 的内容被递增以指向下一条指令。当一条指令的执行阶段完成时，PC 中包含着下一条指令的地址，并且一条新指令的读取阶段又可以开始了。

### 2.3.6　转移

考虑对一个有 $n$ 个数的列表相加的任务。图 2-5 中的程序是对图 2-4 中程序的推广，包含这 $n$ 个数的存储单元的地址用符号 NUM1、NUM2、…、NUM$n$ 给出，并且使用单独的 Load 指令和 Add 指令将每个数累加到寄存器 R2 中，当所有的数都加完以后，结果被存放在存储单元 SUM 中。

如果不使用图 2-5 所示的一长串 Load 和 Add 指令，则可实现一个程序循环，在循环中指令读出列表中的下一个数并将其加到当前的总和中。为了将所有的数相加，循环必须执行与列表中数的数量相等的次数。图 2-6 展示了所需程序的结构。这个循环体是一个重复执行的程序指令序列。它的开始点在 LOOP 单元处，结束点在指令 Branch_if_[R2] > 0 处。每经过一轮循环，列表中下一个元素的地址就被确定下来，然后将这个元素装入 R5 并加到 R3 中。操作数的地址可以用多种方式指出，这些将在 2.4 节中介绍。现在，我们把注意力集中在如何创建并控制程序的循环上。

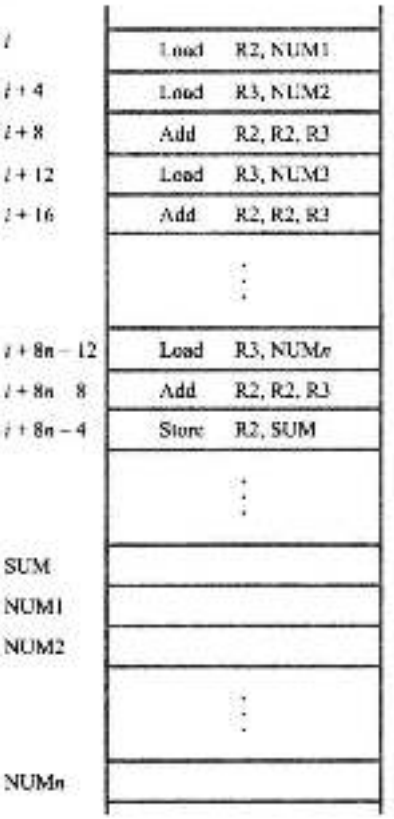



图 2-5　一个将 $n$ 个数相加的程序




图 2-6　用一个循环将 $n$ 个数相加

假设这个列表中元素的数量为 $n$，$n$ 被存储在存储单元 N 中，如图 2-6 所示。寄存器 R2 被当作一个计数器，用来确定执行循环的次数。因此，单元 N 中的内容在程序开始时被装入寄存器 R2 中。然后在循环体内部，每经过一轮循环，指令

Subtract　R2, R2, #1

就对 R2 的内容减 1（我们将在 2.4.1 节中解释符号“#”的意义）。只要 R2 的内容大于 0，就重复执行这个循环。

现在我们来介绍转移（branch）指令。这类指令将一个新的地址加载到程序计数器中，因此，处理器按这个新地址（该地址被称为转移目标（branch target））取出指令并执行，而不是执行在连续的地址顺序上紧接着转移指令的那个单元中的指令。条件转移（conditional branch）指令只有在给出的条件满足时才产生一个转移。如果条件不满足，PC按正常的方式进行递增，连续地址顺序的下一条指令被取出并被执行。

在图2-6的程序中，指令

Branch_if_[R2] > 0　LOOP

是一个条件转移指令，如果寄存器R2中的内容是大于0的，就跳转去执行LOOP单元中的指令。这意味着只要在列表中还有元素可以被加到R3中，这个循环就要重复执行。在$n$遍循环过后，Subtract指令在R2中产生了一个0值，因此转移不再发生。替代它的是Store指令被取出并被执行。它将最后的结果从R3中移到存储单元SUM中。

这种首先测试条件然后从一组可选方式中选择一种继续执行计算的能力比仅仅完成循环控制的方式有更大的应用价值。这种能力在所有计算机的指令集中都可以找到，并且它对于大多数重要任务的程序设计也是必需的。

36 / 39

一种实现条件转移指令的方法是比较两个寄存器的内容，如果比较的结果满足指定的要求，则跳转到目标指令。例如，一条实现了如下动作的指令

Branch_if_[R4]>[R5]　LOOP

可用通用的汇编语言写作

Branch_greater_than　R4, R5, LOOP

或使用实际的助记符写作

BGT　R4, R5, LOOP

它将寄存器R4和R5的内容进行比较，且没有改变其中任一个寄存器的内容。如果R4的内容大于R5的内容，则产生一个到LOOP的跳转。

另一种实现转移指令的方法使用了条件码的概念，我们将在2.10.2节中讨论。

### 2.3.7 生成存储器地址

我们回到图2-6中，从LOOP开始的指令块的目的是在每次循环时，将列表中连续的数相加。因此，在每次循环中，这个块中的Load指令必须访问一个不同的地址。怎样来指明这个地址呢？存储器操作数地址无法在循环中用一条单独的Load指令直接给出。否则，它将需要在每次循环中进行修正。作为一种可能情况，假设处理器寄存器R1用来保存一个操作数的存储器地址。如果在进入循环之前，R1用NUM1作为初始装载地址，以后每次循环后用4进行递增，这样就可以提供所需要的地址了。

在这种情况以及其他很多类似的情况下，更多的是根据需要用灵活的方式去指明一个操作数的地址。一台计算机的指令集通常提供了多种这样的方法，称之为寻址方式（addressing mode）。尽管各计算机之间的具体情况不同，但是其基本概念是相同的。我们将在下一小节中讨论这些问题。

## 2.4 寻址方式

现在我们已经看到了一些汇编语言程序的简单例子。一般来说，一个程序是对存储在计算机存储器中的数据进行操作的，这些数据可以用多种方式组织，而这些方式能够反映出信息的本质以及怎样使用信息。程序员使用数据结构（data structure）如链表和数组去组织用

于计算的数据。

程序通常是用高级语言编写的，在高级语言中程序员可以方便地描述将要在各种数据结构上执行的操作。当把一个高级语言程序翻译成汇编语言时，编译器生成适当的低级指令序列来实现所需的操作。指明指令操作数位置的不同方法称为寻址方式。在这一节中，我们给出 RISC 风格处理器中的基本寻址方式，在表 2-1 中给出了一个简要的描述，同时表中包含了每一种方式中我们将会用到的汇编语法。汇编语法定义了指定指令及其操作数寻址方式的方法，这将在 2.5 节中讨论。

表 2-1　RISC 风格的寻址方式

| 名称 | 汇编语法 | 寻址功能 |
|---|---|---|
| 立即方式 | 值 | 操作数 = 值 |
| 寄存器方式 | Ri | EA=Ri |
| 绝对方式 | LOC | EA=LOC |
| 寄存器间接方式 | (Ri) | EA=[Ri] |
| 变址方式 | X(Ri) | EA=[Ri]+X |
| 带变址的基址方式 | (Ri, Rj) | EA=[Ri]+[Rj] |

EA= 有效地址

Value= 有符号数

X= 变址值

40

### 2.4.1　变量和常数的实现

变量几乎在任何一个计算机程序中都会存在。在汇编语言中，一个变量通过分配一个保存该变量值的寄存器或是存储单元来表示。这个值可以根据需要使用适当的指令来改变。

在图 2-5 的程序中只使用了两种寻址方式访问变量。我们通过指明寄存器的名称或是操作数被装入的存储单元地址来访问操作数。这两种方式的具体定义是：

*寄存器方式*（register mode）——操作数是一个处理器寄存器中的内容，在指令中给出寄存器的名称。

*绝对方式*（absolute mode）——操作数在一个存储单元中，指令中明确地给出了这个单元的地址。

由于在 RISC 风格的处理器中，一条指令必须放到一个单字中，所以能用来指定绝对地址的位的数量是有限的，如果字长为 32 位则这个数量通常就是 16 位。为了生成 32 位的地址，通常通过将位 $b_{15}$ 的值复制到位位置 $b_{31-16}$ 中（符号扩展）来将这个 16 位的值扩展为 32 位。这意味着可以用这种方式指定一个绝对地址，但只能得到一个有限范围的完整地址空间。我们将在 2.9 节中讨论关于如何指定完整的 32 位地址的问题。为了保持例子的简洁，现在我们假设程序中所有存储单元的地址都以 16 位来指定。

41

指令

```
Add   R4, R2, R3
```

的三个操作数都使用了寄存器寻址方式。寄存器 R2 和 R3 保存着两个源操作数，而 R4 则为目的寄存器。

绝对寻址方式可以在程序中表示全局变量。例如在高级语言程序中有这样一个声明：

```
Integer   NUM1, NUM2, SUM;
```

它将导致编译器为变量 NUM1、NUM2 和 SUM 分别分配一个存储单元。每当它们在以后的程序中被引用时，编译器就产生使用绝对方式去访问这些变量的汇编语言指令。

绝对寻址方式用于指令

```
Load   R2, NUM1
```

该指令把存储单元 NUM1 中的值加载到寄存器 R2 中。

表示数据或地址的常数几乎在任何一个计算机程序中都会存在。这样的常数在汇编语言中可以用立即寻址方式来表示。

立即方式 (immediate mode)——操作数在指令中被明确地给出。

例如指令

$$\text{Add} \quad R4, R6, 200_{\text{immediate}}$$

表示将数值 200 与寄存器 R6 的内容相加，并把结果存入寄存器 R4 中。在汇编语言中使用下标来说明立即方式是不合适的。常用的约定是在这个值的前面使用符号“#”来说明这个值是作为立即操作数使用的。因此，我们用以下方式写出上面那条指令：

$$\text{Add} \quad R4, R6, \#200$$

在下面的寻址方式中，指令没有明确地给出操作数或者其地址。取而代之的是，指令中提供了一些信息，当指令被执行的时候，处理器可以从这些信息中推导出有效地址（effective address，EA），然后再利用有效地址去访问操作数。

### 2.4.2 间接和指针

图 2-6 中的程序需要在每次循环时能够修改存储器操作数的地址。提供这一功能的一个好方法是用一个处理器寄存器保存操作数的地址。在每次遍历时，改变（递增）这个寄存器的内容，以提供列表中下一个将要被访问的数的地址。这个寄存器担当着列表指针（pointer）的角色，我们称列表中的每个数据项是通过使用寄存器中的地址间接（indirectly）访问的，所需的功能是由间接寻址方式提供的。

间接方式（indirect mode）——操作数的有效地址是一个寄存器中的内容，而这个寄存器在指令中给出。

我们通过在指令中使用圆括号的方法来表示间接寻址，在圆括号中放置寄存器的名称，如图 2-7 和表 2-1 中给出的示例那样。

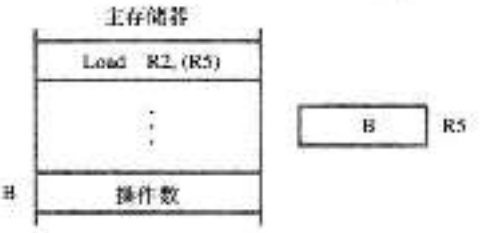



图 2-7 寄存器间接寻址

为了执行图 2-7 中的 Load 指令，处理器将寄存器 R5 中的值作为操作数的有效地址。它请求一个读操作从存储器中取出单元 B 的内容。这个从存储器中读出的值是所需要的操作数。处理器将它加载到寄存器 R2 中。间接寻址也可以通过一个存储单元完成，但只有在 CISC 风格的处理器中才会出现。

间接方式和指针的使用在程序设计中是一个重要且强大的概念。它们允许使用相同的代码来对不同的数据运行操作。例如，图 2-7 中的寄存器 R5 作为 Load 指令的一个指针，用来把存储器中的操作数加载到寄存器 R2 中。有一段时间 R5 可能指向存储器中的单元 B。然后，程序可能改变 R5 的内容使其指向不同的单元，此时相同的 Load 指令将会把那个单元中的值加载到 R2 中。因此，只需要改变指针的值，就可以方便地重用包含该 Load 指令的程序段。

现在我们回到图 2-6 对一个列表中的数进行加法操作的程序中。间接寻址可以用来访问列表中的连续的数，程序的修改结果在图 2-8 中给出。寄存器 R4 被当作指向列表中的数的指针，操作数通过 R4 被间接地访问。程序的初始化部分从存储单元 N 中将计数器的值 $n$ 加载到 R2 中。然后使用 Clear 指令将 R3 的内容清为 0。下一条指令使用立即寻址方式将地址值 NUM1 放置到 R4 中，NUM1 是列表中第一个数的地址。需要注意的是，我们不能使用 Load 指令去加载所需要的立即值，因为 Load 指令只能对存储器源操作数进行操作。取而代之，我们使用 Move 指令

$$\text{Move} \quad R4, \#NUM1$$

```
        Load          R2, N          加载列表的大小
        Clear         R3             将和初始化为 0
        Move          R4, #NUM1      获取第一个数的地址
LOOP:   Load          R5, (R4)       获取下一个数
        Add           R3, R3, R5     把这个数加到和中
        Add           R4, R4, #4     递增指向列表的指针
        Subtract      R2, R2, #1     递减计数器
        Branch_if_[R2]>0  LOOP       如果没完成就返回到循环
        Store         R3, SUM        存储最终的和
```

图 2-8　在图 2-6 的程序中使用间接寻址

在很多 RISC 风格的处理器中，有一个通用寄存器专用于保存常量值 0。通常，这个寄存器是 R0。它的内容不能通过程序指令改变。在我们关于 RISC 风格处理器的讨论中，假设 R0 就是以这种方式使用的。然后上面的 Move 指令可以这样实现：

Add　R4, R0, #NUM1

通常情况下，为了程序员方便，Move 作为一条伪指令（pseudoinstruction）使用，但是实际上是由 Add 指令实现的。

图 2-8 中循环体里的前三条指令实现了图 2-6 中从 LOOP 开始的未说明的指令块。第一次循环时，指令

Load　R5,(R4)

从单元 NUM1 中取出操作数并将其装入 R5 中。第一条 Add 指令把这个数加到寄存器 R3 的总和中。第二条 Add 指令将指针 R4 的内容加上 4，这样在第二次通过循环执行 Load 指令时，R4 中将包含着地址值 NUM2。

作为另一个指针的例子，再来看 C 语言语句

A = * B;

这里 B 是一个指针变量，符号 "*" 是间接访问的操作符。这个语句使得 B 所指向的存储单元的内容被装入存储单元 A 中。这个语句可以被编译成

Load　R2，B
Load　R3，(R2)
Store　R3，A

通过寄存器进行间接寻址已经被广泛应用。图 2-8 中的程序展示了它所提供的灵活性。 44

## 2.4.3　变址和数组

下一个我们将要讨论的寻址方式提供了访问操作数不同方式的灵活性。它对于列表和数组的处理很有用。

变址方式 (index mode)——操作数的有效地址是通过将一个寄存器中的内容加上一个常数值而生成的。

为了方便起见，我们将这种方式中使用的寄存器称为变址寄存器（index register）。通常，变址寄存器就是一个通用寄存器。我们表示变址方式的符号是

X (Ri)

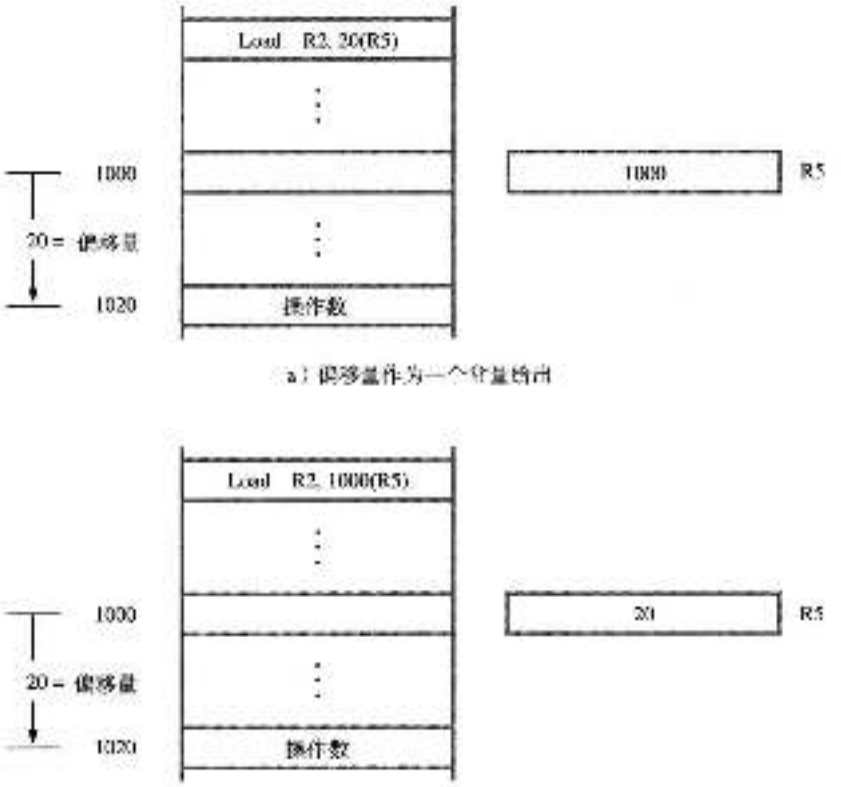
这里 X 表示在指令中包含的有符号操数的常数值，而 Ri 是相关寄存器的名字。该操作数的有效地址用下面的式子给出：

EA = X + [Ri]

寄存器的内容在生成有效地址的过程中不能被改变。

在汇编语言程序中，每当需要一个常数如 X 时，都可以直接给出或用一个表示数值的符号名给出。在这种方法中，一个符号名是与一个特定的数值相关的，这将在 2.5 节中讨论。当指令被转换成机器码时，常数 X 作为指令的一部分被给出并且被限制为比计算机的字长要少的位数。因为 X 是一个有符号的整数，所以在将它与寄存器中的内容相加之前，必须对它进行符号扩展使其与寄存器的字长相同（参看 1.4 节）。

图 2-9 说明了两种使用变址的方法。在图 2-9a 中，变址寄存器 R5 中包含一个存储单元的地址，数值 X 定义了一个从这个地址到操作数位置的偏移量（也称为位移量）。在图 2-9b 中具体说明了所使用的另一种方法。这里，常数 X 相当于一个存储器地址，而变址寄存器中的内容定义了一个到操作数的偏移量。无论在哪种情况下，有效地址都是两个值的和，其中一个值是在指令中明确给出的，另一个值是保存在一个寄存器中的。




a）偏移量作为一个常量给出

b）偏移量在变址寄存器中

图 2-9 变址寻址

为了理解变址寻址的作用，考虑一个有关学生选修课程成绩表的简单例子。假设这个成绩表开始的位置在 LIST 处，它的结构如图 2-10 所示。4 个字长的存储块构成一个记录，记录中存储着与每个学生相关的信息。每个记录由学生的标识号（ID）和学生在三次测验中所得的分数构成。在这个班里有 $n$ 个学生，这个 $n$ 值被存储在单元 N 中，这个单元 N 紧接在这张表的前面。按照图中学生的 ID 号和考试分数给出的地址，假设该存储器是按字节编址的并且字长是 32 位的。

我们应该注意到，图 2-10 中的这张表代表着一个 $n$ 行 4 列的二维数组。每行包含一个学

生的数据项，而每列给出了 ID 号和考试分数。

假设我们想要计算在每次考试中所有得分的总和，并将这三个和存储在存储单元 SUM1、SUM2 和 SUM3 中。图 2-11 中给出了一个可以完成这项任务的程序。在循环体内，程序按照图 2-9a 中描述的方式使用变址寻址方式去访问一个学生记录中三个得分中的每一个分数。寄存器 R2 被用作变址寄存器。在进入循环前，R2 被设置成指向第一个学生记录（其地址为 LIST）的 ID 单元。

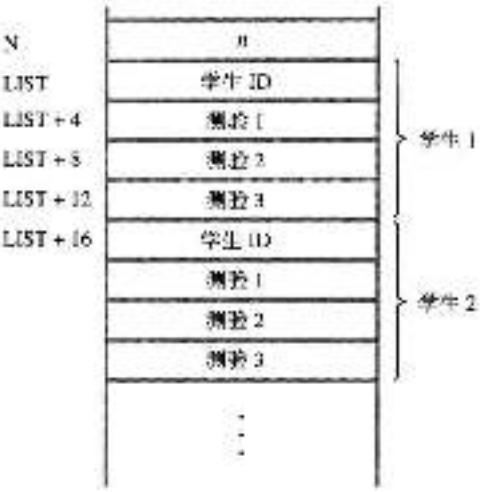



图 2-10　学生成绩表

在第一次循环时，第一个学生的考试得分被加到了保存在寄存器 R3、R4 和 R5 的和中，这些值在初始化时被清成了 0。这些得分可以使用变址寻址方式 4 (R2)、8 (R2) 和 12 (R2) 进行访问。然后变址寄存器 R2 用 16 进行增值，指向第二个学生的 ID 单元。寄存器 R6 初始化时的内容为 $n$ 值，每一次循环结束时它的内容减 1。当 R6 的内容减到 0 时，所有学生的记录就都被访问到了，并且循环结束。在那之前，条件转移指令将控制返回到循环的开始处去处理下一个记录。最后三条指令将寄存器 R3、R4 和 R5 中的累加和分别传送到存储单元 SUM1、SUM2 和 SUM3 中。

```
        Move      R2, #LIST     送学生地址 LIST
        Clear     R3
        Clear     R4
        Clear     R5
        Load      R6, N         加载 n 值
LOOP:   Load      R7, 4(R2)     将下一个学生的测验 1 成绩
        Add       R3, R3, R7    加到部分和上
        Load      R7, 8(R2)     将这个学生的测验 2 成绩
        Add       R4, R4, R7    加到部分和上
        Load      R7, 12(R2)    将这个学生的测验 3 成绩
        Add       R5, R5, R7    加到部分和上
        Add       R2, R2, #16   递增指针
        Subtract  R6, R6, #1    递减计数器
        Branch_if_[R6]>0  LOOP  如果没完成就返回到前面
        Store     R3, SUM1      保存测验 1 的总和
        Store     R4, SUM2      保存测验 2 的总和
        Store     R5, SUM3      保存测验 3 的总和
```

图 2-11　用于访问图 2-10 中学生成绩表的变址寻址方式

这里要强调的是，当变址寄存器 R2 在变址寻址方式中用来访问分数时，其内容是不能够改变的。R2 的内容只能被循环中的最后一条 Add 指令改变，这样就可以从一个学生记录转向下一个学生记录。

一般来说，变址方式有助于访问那些特定的操作数，这些操作数的存储单元是用相对于保存操作数的数据结构中的一个参照点来定义的。在刚刚给出的这个例子中，连续的学生记录的 ID 单元是参考点，考试分数是使用变址寻址方式访问的操作数。

我们已经介绍了变址寻址的最基本的形式，即用寄存器 Ri 和偏移常量 X。在实际编程中还有一些这类基本形式的变形（尽管在一些处理器中可能并不包含它们），它们对存储器操作

数的访问也是非常有效的。例如，可以使用第二个寄存器 Rj 来保存偏移量 X，在这种情况下我们可以将变址方式写成：

(Ri,Rj)

其有效地址是寄存器 Ri 和 Rj 内容的和。第二个寄存器通常称作基址（base）寄存器。这种变址寻址方式在访问操作数方面提供了更多的灵活性，因为有效地址的两部分都可以被改变。

变址方式还有一种使用两个寄存器加上一个常数的版本，这可以表示为：

X (Ri,Rj)

在这种情况下，有效地址是常数 X 与寄存器 Ri 及 Rj 内容的和。这种新增加的灵活性在访问一条记录内部的每一项中的多个部分时是有用的，这里一个项的开始是由寻址方式的（Ri，Rj）部分来指明的。

最后，我们应该注意到在基本的变址方式

X(Ri)

中，如果寄存器的内容等于 0，那么有效地址就恰好等于 X 的符号扩展值。这与绝对方式有同样的效果。如果寄存器 R0 总是包含值 0，那么绝对方式可以简单实现为

X(R0)

## 2.5 汇编语言

45 / 48

机器指令用 0 和 1 模式来表示。这种模式在讨论或准备程序时是很不便的。因此，我们使用符号名来表示这些模式。到目前为止，已经使用了自然单词如 Load、Store、Add 以及 Branch 作为指令操作去表示相应的二进制码模式。当为一台指定的计算机编写程序时，这些单词通常使用称为助记符（mnemonic）的缩写形式来代替，比如 LD、ST、ADD 和 BR。在识别寄存器的时候，用简写符号也是很有用的，例如用 R3 表示寄存器 3。最后，可能需要定义像 LOC 这样的符号来表示一个特定的存储单元。这些符号名以及使用规则的完整集合构成了一种程序设计语言，通常叫做汇编语言（assembly language）。使用助记符以及描述完整的指令和程序的规则集合称为这种语言的语法（syntax）。

使用汇编语言编写的程序可以被一个叫做汇编程序（assembler）的程序自动翻译成机器指令的序列。这个汇编程序是实用程序集中的一个，它是计算机系统软件的一部分。汇编程序与任何其他的程序一样，作为一个机器指令序列被存储在计算机的存储器中。一个用户程序通常是通过键盘输入到计算机中，并存储在存储器或是磁盘上的。在这点上，用户程序被简单地看作是字符数字流的一个集合。当汇编程序执行时，它读取用户程序做分析，然后生成所需要的机器语言程序。而机器语言程序中包含着将要被计算机执行的用 0 和 1 模式说明的指令。按照原始的字符数字文本形式组成的用户程序叫做源程序 (source program)，汇编后的机器语言程序叫做目标程序（object program）。我们将在 2.5.2 节和第 4 章中讨论汇编程序是如何工作的。首先给出一些汇编语言本身的外部特征。

一台给定计算机的汇编语言可能是大小写敏感的也可能不是，也就是说，它可以区别也可以不区别大写和小写字母。在本节的例子中，为了提高文本的易读性，我们使用大写字母表示所有的名称和标号。例如，将 Store 指令写成：

ST R2, SUM

助记符 ST 表示这条指令所执行操作的二进制模式，或叫做操作（OP）码。汇编程序将这个助记符翻译成这台计算机能够识别的二进制 OP 码。

OP 码助记符后面最少跟随一个空格字符或制表符，再后面的信息给出了具体的操作数

在上面的 Store 指令中，源操作数在寄存器 R2 中。这个信息后面跟随的是目的操作数的具体说明，用一个逗号与源操作数隔开。目的操作数在存储单元中，该存储单元使用名字 SUM 来表示它的二进制地址。

因为对于具体操作数的单元可以用多种寻址方式来说明，所以每条汇编语言指令必须说明使用的是哪一种方式。例如，一个数字的值或者一个单独使用的名字，就像前面指令中的 SUM，可以用来表示绝对方式。数字符号通常表示一个立即操作数，这样，指令

ADD R2,R3,#5

表示对寄存器 R3 的内容加上数字 5 并将结果放到寄存器 R2 中。这个数字符号不是表示立即寻址方式的唯一方法。在有些汇编语言中，立即寻址方式在 OP 码的助记符中指出。例如，前面的 Add 指令可以写成 49

ADDI R2, R3, 5

助记符 ADDI 中的后缀 I 表示第二个源操作数是用立即寻址方式给出的。

间接寻址通常由放置在圆括号中的代表指向操作数的指针的名字或符号来表示。例如，如果寄存器 R2 包含存储器中一个数的地址，那么可以通过如下指令将这个数加载到寄存器 R3 中：

LD R3, (R2)

### 2.5.1 汇编指示

除了为在程序中表示指令提供一种机制以外，汇编语言还允许程序员说明将源程序翻译成目标程序时所需要的其他信息。我们已提到过需要给程序中使用的任何一个名字分配数值。假设名字 TWENTY 用来表示值 20，这个情况可以通过下面的标等（equate）语句来传递到汇编程序中：

TWENTY EQU 20

这个语句在目标程序运行时不代表一条将要被执行的指令，实际上，它甚至不出现在目标程序中。它只是简单地通知汇编程序，名字 TWENTY 无论出现在程序的任何地方都要用值 20 来取代。这样的语句叫做汇编指示（或汇编命令），它在汇编程序将源程序翻译成目标程序时使用。

为了进一步说明汇编语言的使用，我们重新考虑图 2-8 中的程序。为了在计算机中运行这个程序，需要按照汇编语言的要求来编写源代码，具体说明生成相应目标程序所需要的所有信息。假设它的每条指令和每个数据项都占据存储器中的一个字。另外假设存储器是按字节寻址的并且它的字长是 32 位。还假定这个目标程序将被装载到主存储器中，如图 2-12 所示。图中给出了该程序为了运行而被装载后，机器指令和所需数据项占据的存储器地址。如果汇编程序按这种分配方式产生目标程序，它需要知道：

- 怎样去解释名字。

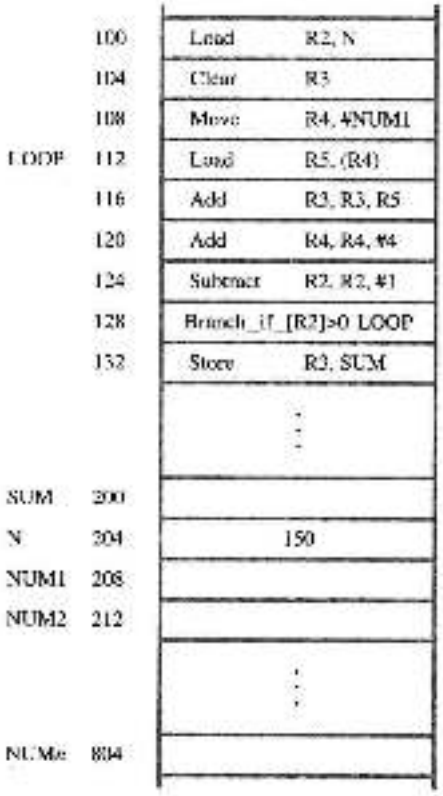



图 2-12 图 2-8 中程序的存储器分配

- 在存储器的什么位置存放指令。
- 在存储器的什么位置存放数据操作数。

为了提供这些信息，源程序可以写成如图 2-13 所示的形式。这个程序以汇编指示 ORIGIN 开始，ORIGIN 告诉汇编程序在存储器的什么位置存放后面的指令。它指明了目标程序的指令将要被装载到存储器中起始地址为 100 的连续单元中。跟在后面的是用适当的助记符和语法编写的源程序指令。注意到我们使用了语句

BGT R2, R0, LOOP

来表示一条执行下面这个操作的指令：

Branch_if_[R2]>0 LOOP

第二个汇编指示 ORIGIN 告诉汇编程序在存储器的什么位置存放后面的数据块。在本例中，被指明的单元地址是 200。这是要用来存放最后总和的单元。通过使用汇编指示符 RESERVE 来为总和保留 4 个字节的空间。在地址 204 中的下一个字，必须包含列表中条目的数量 150。DATAWORD 指示符是用来告诉汇编程序这个要求的。下一个 RESERVE 指示符声明预留一个 600 字节的存储块来保存数据。这个指示符并不能将任何数据加载到这些单元里。只有使用输入程序，才能把数据装入该存储器中，就像我们将在第 3 章中解释的那样。源程序中最后一个语句是汇编指示 END，它告诉汇编程序这里是源程序正文的结束点。

| | 存储器地址标号 | 操作 | 寻址或数据信息 |
|---|---|---|---|
| 汇编指示 | | ORIGIN | 100 |
| 生成机器指令的语句 | LOOP: | LD | R2, N |
| | | CLR | R3 |
| | | MOV | R4, #NUM1 |
| | | LD | R5, (R4) |
| | | ADD | R3, R3, R5 |
| | | ADD | R4, R4, #4 |
| | | SUB | R2, R2, #1 |
| | | BGT | R2, R0, LOOP |
| | | ST | R3, SUM |
| | | 下一条指令 | |
| 汇编指示 | | ORIGIN | 200 |
| | SUM: | RESERVE | 4 |
| | N: | DATAWORD | 150 |
| | NUM1: | RESERVE | 600 |
| | | END | |

图 2-13 图 2-12 中程序的汇编语言表示形式

我们先前已经描述了 EQU 指示符是如何将一个指定的值（可能是一个地址）与一个特定的名字关联起来的。在图 2-13 中阐明了一种把地址与名字或标号关联起来的不同方法。我们可以为任何指令和存储单元中的数据指定一个存储器地址标号。汇编程序自动为这个标号分配该单元的地址。举个例子，对于跟在第二个 ORIGIN 指示符之后的数据块，我们用标号 SUM、N 和 NUM1 来标记。因为 ORIGIN 指示符之后的第一个 RESERVE 语句被指定了标号 SUM，所以就为名字 SUM 分配值 200。每当在程序中遇到 SUM，它都被替换成这个值。这种将 SUM 当作一个标号的方式等价于使用以下的汇编指示：

SUM EQU 200

类似地，标号 N 和 NUM1 被分别分配了值 204 和 208，因为它们表示两个紧跟在地址为 200 的字单元之后的字单元地址。

大多数汇编语言要求源程序中的语句按以下方式编写：

标号: 操作 操作数 注释

这四个字段（field）采用适当的定界符，可能用一个或多个空格或者制表符进行分隔。标号可以是任意与存储器地址相关的名字，该语句所生成的机器语言指令将被装入这个地址中。标号也可以与数据项的地址相关联。在图 2-13 中有四个标号：LOOP、SUM、N 和 NUM1。

操作字段中包含着一个汇编指示符或所需指令的 OP 码助记符。操作数字段包含着用于访问操作数的寻址信息。注释字段将被汇编程序忽略，它是使程序更容易理解的文档。

我们已经介绍的仅仅是汇编语言最基本的特性。不同计算机在这些语言的细节和复杂程度上是不相同的。

### 2.5.2 程序的汇编和执行

用汇编语言编写的源程序在能够被执行之前必须被汇编成机器语言的目标程序。这是由汇编程序来完成的，它将用在机器指令中使用的二进制码替换所有表示操作的符号以及寻址方式，并且将所有的名字和标号用它们的实际值替换。

汇编程序为指令和数据块分配地址，起始地址由汇编指示符 ORIGIN 给出。它还插入一些可能是在 DATAWORD 命令中给出的常数，并按 RESERVE 命令的要求保留存储空间。

汇编处理的一个关键部分是确定那些将要用来替代名字的值。在某种情况下，名字的值是由 EQU 指示符说明的，这是一种简单的方式。在另一些情况下，名字是由一条给定指令的标号字段定义的，用这种名字表示的值是由汇编后的目标程序中这条指令所在的位置决定的。因此，汇编程序必须在生成连续指令的机器码时时刻注意地址的值。例如，图 2-13 程序中的名字 LOOP 和 SUM 将分别被分配的值是 112 和 200。

在有些情况下，汇编程序不直接用一个地址的实际值去取代表示这个地址的名字。例如，在一条转移指令中，用于指明将要转入的那个位置（转移目标）的名字不用实际的地址替换。转移指令通常是在机器码中用指定转移目标的方法来实现的，转移目标被指定为从程序计数器中的当前地址到目标指令之间的距离（以字节为单位）。汇编程序计算转移偏移量（branch offset），偏移量可正可负，并将它放进机器指令中。我们将在 2.13 节中展示转移指令是如何实现的。 53

汇编程序将目标程序存储在计算机中可用的辅助存储设备中，通常是磁盘。在目标程序开始执行之前，它必须被装入计算机的主存储器中。为了完成这项工作，另一个叫做装载程序（loader）的实用程序必须已经在存储器中了。执行这个装载程序就是运行一系列的输入操作，将机器语言程序从磁盘中传送到存储器中指定的位置上。装载程序必须知道这个程序的长度和将要存放它的存储器地址。汇编程序通常把这些信息放在目标代码的头部。装载完目标代码后，装载程序就转移到将要执行的第一条指令处开始执行目标程序，将要执行的第一条指令可能由一个地址标号如 START 来识别。汇编程序把这个地址放到目标代码的头部供装载程序在执行时使用。

当目标程序开始执行后，它会一直进行到结束，除非在程序中有逻辑性错误。用户必须能够容易地发现错误，汇编程序只能够监测并报告语法错误。为了帮助用户发现其他的程序设计错误，系统软件通常提供一个调试（debugger）程序。这个程序使用户能够在一些感兴趣的

点上停止目标程序的运行，去检查各个处理器寄存器和存储单元的内容。

在这一小节中，我们介绍了一些关于程序汇编和运行的重要问题。第4章将会更详细地讨论这些问题。

### 2.5.3 数的表示

在处理数值的时候，使用熟悉的十进制记数法通常是很方便的。当然，这些值是用二进制方式存储在计算机中的。有些情况下，直接指定二进制模式是比较方便的。大多数汇编程序允许数值在汇编语法定义的约定下有多种表示方式。例如，考虑数字93，它用8位二进制数表示为01011101。如果这个值被当作立即操作数使用，它可以作为一个十进制数给出，就像如下的指令：

```
ADDI   R2, R3, 93
```

或者用一个汇编程序指定的前缀符号（比如百分号）识别成二进制数，比如：

```
ADDI   R2, R3, %01011101
```

二进制数可以写成比较紧凑如十六进制（hexadecimal或hex）方式的数，这时数的四位可以用一位十六进制数字表示。前十个模式0000、0001、0010、…、1001，称为BCD（binary-coded decimal，二进制编码的十进制）码，用数0、1、…、9表示，其余的六个4位模式1010、1011、…、1111用字母A、B、…、F表示。在十六进制表示法中，十进制数93变成5D。在汇编语言中，常常用一个0x（像C语言中一样）或者美元符号做前缀来表示一个十六进制数，这样前面的指令应写成：

54

```
ADDI   R2, R3, 0x5D
```

## 2.6 堆栈

被程序操作的数据可以用多种方法来组织。我们已经接触了像表这样的数据结构，现在来考虑另一种重要的数据结构——堆栈（简称栈）。堆栈（stack）是一个数据元素表，其中的元素通常是字，它具有的访问约束是数据元素只能从表的一端移进或移出。这一端叫做该栈的栈顶，另一端为栈底。这种结构有时也称为下推（pushdown）栈。设想自助餐厅里的一叠盘子，客户从这叠盘子的顶部拿走新盘子，而清洗干净的盘子又被添加到这叠盘子的顶部。后进先出（last-in-first-out，LIFO）栈也是用来描述这种类型的存储机制的，即表示最后放入堆栈中的数据项在检索开始时是第一个被取出的。术语压入（push）和弹出（pop）分别用来描述向栈中放入一个新的数据项和从栈顶取出一个数据项。

在现代计算机中，栈是用主存储器中的一部分来实现的。一个叫栈指针（stack pointer，SP）的处理器寄存器被用来指向一个特定的称为处理器栈（processor stack）的栈结构，它的用法稍后会解释。

数据可以存储在一个栈中，其中连续的元素占据连续的存储单元。假设第一个元素放在BOTTOM位置上，当新的元素被压进栈时，它们被放置在连续的低地址单元中。在我们的讨论中，使用了一个按照减小存储器地址方向进行增长的栈，这是一种惯用的方式。

图2-14给出了一个保存字数据项的堆栈示例。它所保存的数值以43为栈底，-28为栈顶。栈指针（SP）用来跟踪栈中元素地址，它在任何时候都指向栈顶。假设一个按字节寻址的存储器具有32位的字长，压栈操作可以这样实现：

```
Subtract      SP, SP, #4
Store         Rj, (SP)
```

这里 Subtract 指令从 SP 的内容中减去 4，再把结果放回到 SP 中。假设这个将要被压入栈中的新项在处理器寄存器 $R_j$ 中，则 Store 指令会把这个值放到栈中。这两条指令将 $R_j$ 中的字拷贝到该栈的栈顶上。在存储（压入）操作之前对栈指针减 4。弹出操作可以这样实现：

```
Load     Rj, (SP)
Add      SP, SP, #4
```

这两条指令从栈中将栈顶值加载（弹出）到寄存器 $R_j$ 中，然后将栈指针加 4，便其指向新的栈顶元素。图 2-15 给出了在图 2-14 的栈中完成这些操作的效果。 55

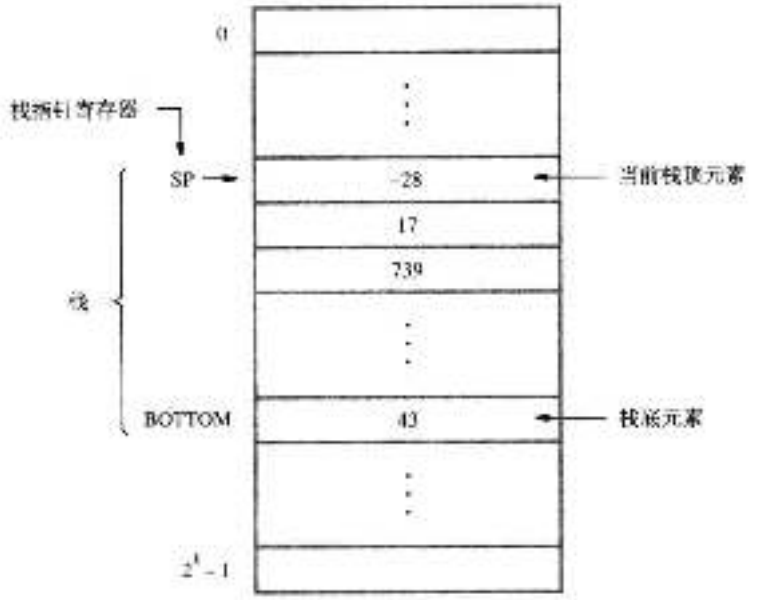



图 2-14　存储器中的一个字堆栈

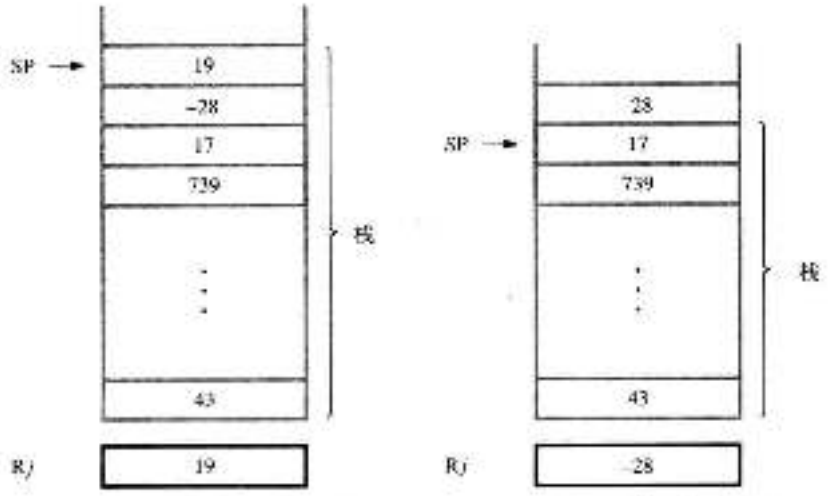



图 2-15　对图 2-14 中的栈进行栈操作的效果

## 2.7 子程序

在一个给定的程序中，常常需要对不同的数据多次执行同一个特定的任务。比较明智的做法是把这个任务实现为一个指令块，每次要执行任务的时候就运行该指令块。这样的指令块通常叫做子程序（subroutine）。例如，一个子程序可以求一个数学函数的值或者将一列数值按照递增或递减的顺序排序。

虽然可以在程序中任何需要的地方复制构成子程序的指令块，但是为了节省空间，我们只需在存储器中存放一份子程序的拷贝。当任何程序需要使用该子程序时，只要简单地跳转到这个子程序的起始位置即可。当一个程序将控制转移到一个子程序时，我们称其为调用（call）子程序。执行这个转移操作的指令叫做调用（Call）指令。

一个子程序被执行后，调用它的程序必须能够恢复执行紧跟在这条调用指令后面的指令。我们将此称为子程序返回（return）到调用它的程序中，这可以通过在子程序中执行一条Return指令的方式来完成。因为子程序可能在一个调用程序的任何位置上被调用，所以必须准备好返回到适当的位置上。调用程序重新恢复执行的位置是当执行Call指令时被修改的程序计数器（PC）所指出的位置。因此，为了能正确地返回到调用程序中，Call指令必须保存PC中的内容。

这种能够在计算机中调用子程序并从子程序中返回的方法称为子程序链接（subroutine linkage）法。最简单的子程序链接法是将返回地址存储在一个指定的单元中，这个单元可以是一个专门用于这种功能的寄存器，这种寄存器被称为链接寄存器（link register）。当子程序完成了它的任务时，Return指令通过链接寄存器的间接跳转返回到调用程序中。

Call指令是一条特殊的转移指令，它执行以下操作：

- 将PC中的内容存储到链接寄存器中。
- 转移到由Call指令指定的目标地址中。

Return指令也是一条特殊的转移指令，它执行的操作是：

- 转移到链接寄存器所保存的地址中。

56
/
57

图2-16说明了Call和Return指令是如何影响PC和链接寄存器的。

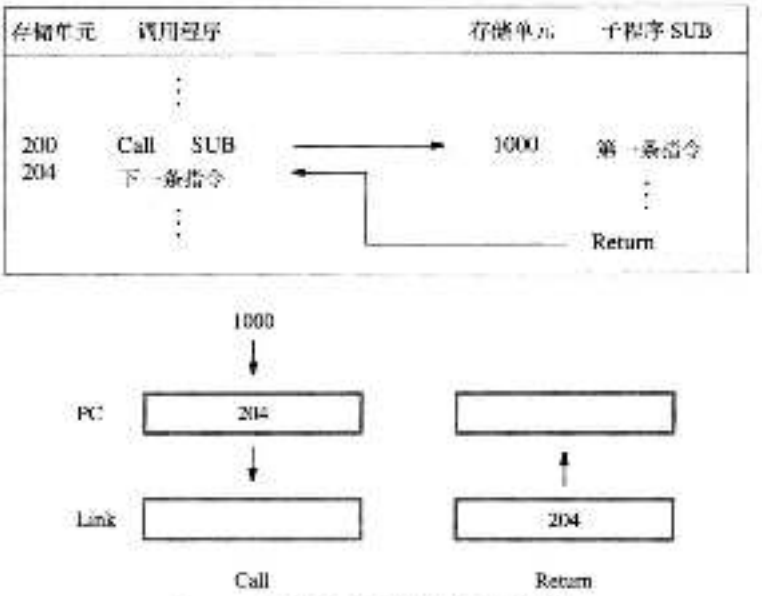



图2-16 使用链接寄存器的子程序链接

### 2.7.1　子程序嵌套及处理器堆栈

一个子程序调用另一个子程序叫做子程序嵌套（subroutine nesting）。在这种情况下，第二个调用的返回地址也被存储在链接寄存器中，这会覆盖它原有的内容。因此，在调用其他子程序之前，把链接寄存器中的内容存储到其他的单元里是十分重要的，否则第一个子程序的返回地址将会丢失。

子程序的嵌套可以达到任何深度。最终，最后一个被调用的子程序完成它的计算并返回到调用它的子程序中。第一次返回所需要的返回地址，是嵌套调用序列中最后一个生成的。也就是说，返回地址的生成和使用是按照后进先出的顺序进行的。这意味着，与子程序调用有关的返回地址应该被压入处理器栈中。

如果一个给定的子程序 SUB1 在调用另一个子程序 SUB2 之前，把返回地址保存到栈中的链接寄存器中（该链接寄存器是通过栈指针 SP 访问的），那么就可以实现正确的嵌套调用顺序。然后，在执行它自己的 Return 指令之前，子程序 SUB1 必须从栈中弹出所保存的返回地址，并将其装入链接寄存器中。

58

### 2.7.2　参数传递

当调用一个子程序时，程序必须要给子程序提供参数，也就是那些将在计算中使用的操作数或是它们的地址。然后，子程序返回另外的一些参数，通常是计算的结果。这种在调用程序和子程序之间的信息交换称为参数传递（parameter passing）。参数传递能够用多种方法实现。参数可以放在寄存器或存储单元中，这些地方是能够被子程序访问到的。或者，可以将这些参数放在处理器栈中。

通过处理器寄存器进行参数传递是简单有效的。图 2-17 给出了图 2-8 中将一列数字相加的程序是如何作为一个子程序（LISTADD），通过寄存器传递参数来实现的。存储在存储单元

```
调用程序

            Load        R2, N          参数 1 为列表长度
            Move        R4, #NUM1      参数 2 为列表位置
            Call        LISTADD        调用子程序
            Store       R3, SUM        保存结果
            ⋮

子程序

LISTADD:    Subtract    SP, SP, #4
            Store       R5, (SP)       将 R5 的内容保存到栈中
            Clear       R3             将和初始化为 0
LOOP:       Load        R5, (R4)       获取下一个数
            Add         R3, R3, R5     把这个数加到和中
            Add         R4, R4, #4     将指针增加 4
            Subtract    R2, R2, #1     递减计数器
            Branch_if_[R2]>0 LOOP
            Load        R5, (SP)       恢复 R5 的内容
            Add         SP, SP, #4
            Return                     返回到调用程序
```

图 2-17　将图 2-8 中的程序写成一个子程序：通过寄存器传递参数

N 中的列表大小 $n$ 和第一个数的地址 NUM1 分别通过寄存器 R2 和 R4 传递。由子程序计算出来的和通过寄存器 R3 传回给调用程序。在图 2-17 中，前四条指令组成了调用程序的相关部分。其中的前两条指令把 $n$ 和 NUM1 分别装入 R2 和 R4 中。Call 指令转移到子程序的起始位置 LISTADD 处，并将返回地址（即这个调用程序的 Store 指令的地址）保存到链接寄存器中。子程序计算出和并将它放入 R3 中。子程序执行 Return 指令后，调用程序将 R3 中的和存储到存储单元 SUM 中。

59

除了用来传递参数的寄存器 R2、R3 和 R4 外，子程序还用到了寄存器 R5。由于调用程序可能也用到了 R5，所以需要在子程序的入口处将它的值也压入处理器栈中保存起来，并在返回调用程序前恢复。

如果涉及许多参数，那么可用的通用寄存器就有可能不能满足向子程序传递参数的需要。处理器栈提供了一种方便灵活的机制来传递任意数量的参数。图 2-18 给出了由图 2-8 的程序重写成的子程序 LISTADD，这个子程序用处理器栈传递参数。该列表中第一个数的地址以及列表中的项数被压入由寄存器 SP 指向的处理器栈中。然后子程序被调用。在返回调用程序之前，计算出的和被放到栈中。

```
假设栈顶在图 2-19 的第一级中
                Move            R2, #NUM1       把参数压入栈中
                Subtract        SP, SP, #4
                Store           R2, (SP)
                Load            R2, N
                Subtract        SP, SP, #4
                Store           R2, (SP)
                Call            LISTADD         调用子程序（栈顶
                                                在第二级）
                Load            R2, 4(SP)       从栈中获取结果并
                Store           R2, SUM         保存到 SUM 中
                Add             SP, SP, #8      恢复栈顶（栈顶在
                                                第一级）
                :

LISTADD:        Subtract        SP, SP, #16     保存寄存器
                Store           R2, 12(SP)
                Store           R3, 8(SP)
                Store           R4, 4(SP)
                Store           R5, (SP)        （栈顶在第二级）
                Load            R2, 16(SP)      将计数器初始化为 n
                Load            R4, 20(SP)      初始化指针使其指向列表
                Clear           R3              将和初始化为 0
LOOP:           Load            R5, (R4)        获取下一个数
                Add             R3, R3, R5      把这个数加到和中
                Add             R4, R4, #4      将指针增加 4
                Subtract        R2, R2, #1      递减计数器
                Branch_if_[R2]>0 LOOP
                Store           R3, 20(SP)      把结果放到栈中
                Load            R5, (SP)        恢复寄存器
                Load            R4, 4(SP)
                Load            R3, 8(SP)
                Load            R2, 12(SP)
                Add             SP, SP, #16     （栈顶在第二级）
                Return                          返回到调用程序
```

图 2-18 将图 2-8 中的程序写成一个子程序：在堆栈上传递参数

图 2-19 展示了这个例子中栈的表项。假设在子程序被调用之前，栈顶在第一级上。调用程序将地址 NUM1 和数值 $n$ 压入栈中，并调用了程序 LISTADD。现在栈顶是在第二级上。子程序在运行的时候使用了四个寄存器，因为这些寄存器中可能包含属于调用程序的有效数据，所以在子程序开始的时候，它们的内容应该被压入栈中进行保存。现在栈顶是在第三级上。子程序使用变址寻址方式从这个堆栈中访问参数 $n$ 和 NUM1，其中偏移量的值是相对于新的栈顶（第三级）而言的。注意这一操作不改变栈指针，因为有效的数据项仍然是在这个栈的栈顶。数值 $n$ 被装入 R2 中作为计数的初始值，并且地址 NUM1 被装入 R4 中作为扫描这个列表表项的一个指针。

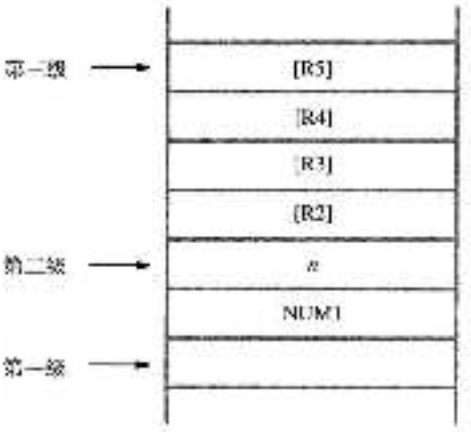



图 2-19　图 2-18 程序中栈的内容

在计算结束时，寄存器 R3 中包含着计算出来的和。在子程序返回到调用程序之前，R3 的内容被插入到这个栈中，替换了参数 NUM1，因为这个参数不再需要了。然后子程序所使用的这四个寄存器的内容从栈中恢复。同样，增加栈指针以指向子程序被调用时已存在的栈顶，即第二级中的参数 $n$。子程序返回后，调用程序将结果存在单元 SUM 中，再通过将 SP 增加 8 的方式来将栈顶降到它原来的级别上。

观察图 2-18 中的子程序 LISTADD 可以发现，我们没有使用这对指令

```
Subtract        SP, SP, #4
Store           Ri, (SP)
```

去把每个寄存器的内容压到栈中。由于我们必须保存四个寄存器，所以需要八条指令。而通过立即调整 SP 使其指向四个寄存器都保存后即生效的栈顶，我们则只需要五条指令。然后，用变址方式来保存寄存器的内容。在从子程序返回前恢复寄存器的时候，我们使用同样的优化方式。

我们还应该注意到一些计算机有特殊的指令用于装载和保存多个寄存器。例如，可以使用如下指令将图 2-18 中的四个寄存器保存在栈中：

```
StoreMultiple   R2-R5, -(SP)
```

源寄存器通过范围 R2-R5 说明。记号 -(SP) 说明了栈指针必须相应地调整。前面的减号表明在把每个寄存器的内容放到栈中之前，必须将 SP 减 4。

同样，指令

```
LoadMultiple    R2-R5, (SP)+
```

以相反的顺序将保存在栈中的值加载到寄存器 R2、R3、R4 和 R5 中。记号 (SP)+ 表明在将每个值装入相应的寄存器之后，必须将栈指针增加 4。我们将在 2.9.1 节中详细讨论 -(SP) 和 (SP)+ 所表示的寻址方式。

**按值和按引用的参数传递**

注意在图 2-17 和图 2-18 中两个参数 NUM1 和 $n$ 的实际值被传递到了子程序中，子程序的目的是对一个列表中的数字做加法。调用程序不传递实际的列表项，而传递这个列表中第一个数的地址，这种技术叫做按引用传递（passing by reference）。第二个参数是按值传递（passing by value），也就是将实际的表项数 $n$ 传递到子程序中。

### 2.7.3 堆栈的结构

现在我们来看在图 2-18 和 2-19 的例子中存储空间是如何作为栈来使用的。在子程序执行期间，栈顶的 6 个单元包含了子程序需要的条目。这些单元构成了子程序的一个私有工作空间，该空间在进入子程序时分配，当子程序将控制交回调用程序时释放。这种空间叫做栈结构（stack frame）。如果子程序需要更多的空间用于存储局部变量，也可以在栈中为这些变量分配空间。

图 2-20 给出了一个常用的栈结构中的信息布局示例。除了栈指针 SP 外，通常还需要另一个指针寄存器：结构指针（frame pointer，FP），以便于访问传递到子程序中的参数，以及子程序中使用的局部变量。在该图中，我们假设有四个参数被传递到子程序中，有三个局部变量要在子程序内部使用，还有寄存器 R2、R3 和 R4 需要进行保存，因为它们也要在子程序中使用。就像我们会在下面的例子中看到的一样，当使用嵌套子程序的时候，调用程序的栈结构还将包括返回地址。

63

如图 2-20 所示，FP 寄存器指向所存储参数正上方的单元，我们可以使用变址寻址方式很容易地访问这些参数和局部变量。参数可以用地址 4(FP)、8(FP)、…访问。局部变量可以用地址 −4(FP)、−8(FP)、…访问，FP 中的内容在子程序执行期间保持不变，这一点不同于栈指针 SP，SP 必须始终指向栈中当前的栈顶元素。

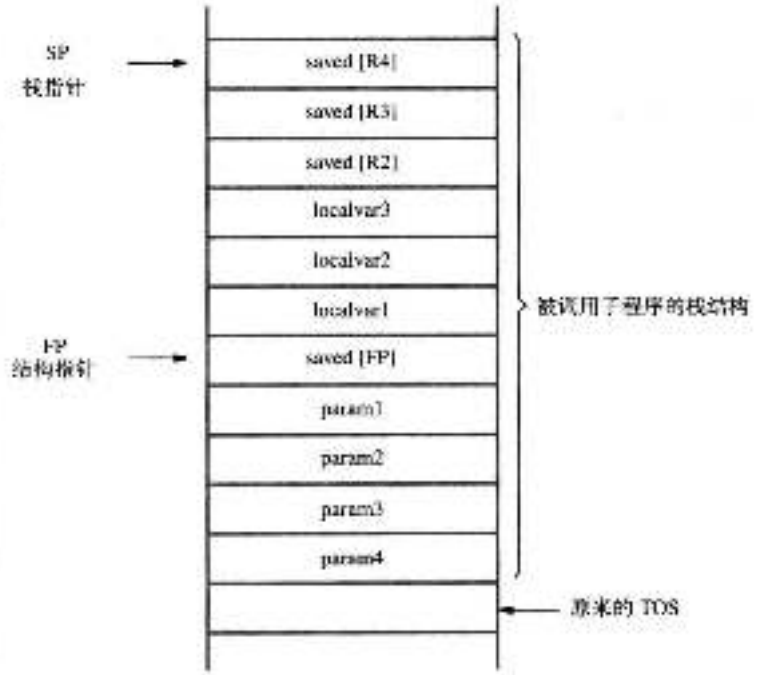



图 2-20 一个子程序栈结构示例

现在我们来看看当调用一个子程序，对它的栈结构进行分配、使用和释放时，指针 SP 和 FP 是如何操作的。我们首先假定 SP 指向图 2-20 中原来的栈顶（TOS）元素。在子程序被调用之前，调用程序将四个参数压入栈中。然后执行 Call 指令。这时，SP 指向最后一个被压入栈中的参数。如果子程序要使用结构指针，它应该先把 FP 的内容压入栈中保存起来，因为 FP 通常是一个通用寄存器，它可能包含用于调用程序的信息。此时，SP 指向保存着 FP 值的单元。然后将 SP 的值拷贝到 FP 中。

这样一来，在子程序中执行的前三条指令是：

```
Subtract    SP, SP, #4
Store       FP, (SP)
Move        FP, SP
```

Move 指令将 SP 的值复制到 FP 中。这些指令执行以后，SP 和 FP 都指向了保存后的 FP 的内容。现在通过执行以下指令在栈中为三个局部变量分配空间：

```
Subtract    SP, SP, #12
```

最后，处理器寄存器 R2、R3 和 R4 的内容被压入栈中进行保存。此时，这个栈结构已经被设置成如图 2-20 所示的状况。

子程序现在开始执行它的任务。当这个任务完成时，子程序弹出 R4、R3 和 R2 的保存值放回到这些寄存器中，执行以下指令从栈结构中释放局部变量：

Add　SP, SP, #12

并且弹出所保存的 FP 旧值放回到 FP 中。这时候，SP 指向了放在栈中的最后一个参数，接着执行返回指令 Return，将控制交还给调用程序。

调用程序负责从栈结构中释放这些参数，其中一些可能是子程序传递回来的结果。在释放参数之后，栈指针指向原来的 TOS（栈顶），于是我们又回到了调用前的状态。

**用于子程序嵌套的栈结构**

当使用嵌套子程序的时候，需要保证返回地址都被正确地保存起来。当调用程序调用一个子程序 SUB1 的时候，返回地址被保存在链接寄存器中。现在，如果 SUB1 要调用另一个子程序 SUB2，它必须在调用 SUB2 之前保存链接寄存器的当前内容。保存返回地址的合适位置是在 SUB1 的栈结构中。如果 SUB2 接着又调用 SUB3，那么它必须将链接寄存器的当前内容保存到与 SUB2 相关联的栈结构中，以此类推。

我们来看一个例子，主程序调用了第一个子程序 SUB1，然后 SUB1 又调用了第二个子程序 SUB2，该过程由图 2-21 给出。这两个嵌套子程序所对应的栈结构如图 2-22 所示。所有与这个例子相关的参数都被传递到栈

| 存储单元 | | 指令 | | 注释 |
|---|---|---|---|---|
| **主程序** | | | | |
| | | ⋮ | | |
| 2000 | | Load | R2, PARAM2 | 将参数放置到栈中 |
| 2004 | | Subtract | SP, SP, #4 | |
| 2008 | | Store | R2, (SP) | |
| 2012 | | Load | R2, PARAM1 | |
| 2016 | | Subtract | SP, SP, #4 | |
| 2020 | | Store | R2, (SP) | |
| 2024 | | Call | SUB1 | 调用子程序 |
| 2028 | | Load | R2, (SP) | 保存结果 |
| 2032 | | Store | R2, RESULT | |
| 2036 | | Add | SP, SP, #8 | 恢复栈的级数 |
| 2040 | | 下一条指令 | | |
| **第一个子程序** | | | | |
| 2100 | SUB1: | Subtract | SP, SP, #24 | 保存寄存器 |
| 2104 | | Store | LINK_reg, 20(SP) | |
| 2108 | | Store | FP, 16(SP) | |
| 2112 | | Store | R2, 12(SP) | |
| 2116 | | Store | R3, 8(SP) | |
| 2120 | | Store | R4, 4(SP) | |
| 2124 | | Store | R5, (SP) | |
| 2128 | | Add | FP, SP, #16 | 初始化结构指针 |
| 2132 | | Load | R2, 8(FP) | 获取第一个参数 |
| 2136 | | Load | R3, 12(FP) | 获取第二个参数 |
| | | ⋮ | | |
| | | Load | R4, PARAM3 | 将参数放置到栈中 |
| | | Subtract | SP, SP, #4 | |
| | | Store | R4, (SP) | |
| | | Call | SUB2 | |
| | | Load | R4, (SP) | 从SUB2获取结果 |
| | | Add | SP, SP, #4 | |
| | | ⋮ | | |
| | | Store | R5, 8(FP) | 将答案放入栈中 |
| | | Load | R5, (SP) | 恢复寄存器 |
| | | Load | R4, 4(SP) | |
| | | Load | R3, 8(SP) | |
| | | Load | R2, 12(SP) | |
| | | Load | FP, 16(SP) | |
| | | Load | LINK_reg, 20(SP) | |
| | | Add | SP, SP, #24 | 返回到主程序 |
| | | Return | | |
| **第二个子程序** | | | | |
| 3000 | SUB2: | Subtract | SP, SP, #12 | 保存寄存器 |
| 3004 | | Store | FP, 8(SP) | |
| | | Store | R2, 4(SP) | |
| | | Store | R3, (SP) | |
| | | Add | FP, SP, #8 | 初始化结构指针 |
| | | Load | R2, 4(FP) | 获取参数 |
| | | ⋮ | | |
| | | Store | R3, 4(FP) | 将 SUB2 的结果放入栈中 |
| | | Load | R3, (SP) | 恢复寄存器 |
| | | Load | R2, 4(SP) | |
| | | Load | FP, 8(SP) | |
| | | Add | SP, SP, #12 | |
| | | Return | | 返回到子程序 1 |

图 2-21　嵌套子程序

中，这两个图中只显示了在主程序和两个子程序中的控制流和数据。实际的计算方法并没有给出。

执行的流程如下：主程序将两个参数 param2 和 param1 按顺序压进栈中，然后调用 SUB1。第一个子程序负责计算一个结果并使用栈将这个结果传回给主程序。在其计算过程中，SUB1 调用了第二个子程序 SUB2，用于执行一些其他的子任务。SUB1 给 SUB2 传递了一个单独的参数 param3，并且通过栈中相同的位置把结果传回给 SUB1。当 SUB2 执行了 Return 指令以后，SUB1 把这个结果装入寄存器 R4 中。然后 SUB1 继续计算，最后用堆栈方式将所需要的答案传回给主程序。当 SUB1 执行返回指令返回到主程序时，主程序将这个答案存放在存储单元 RESULT 中，恢复栈的级数，然后在地址为 2040 的下一条指令处继续它的计算过程。要注意返回到调用程序的地址 2028 是如何存储到图 2-22 中 SUB1 的栈结构中的。

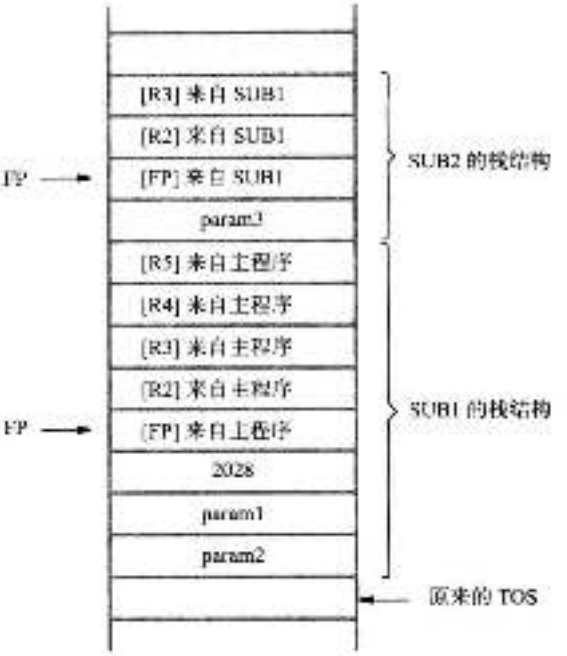



图 2-22 图 2-21 的栈结构

在图 2-21 中的注释给出了如何管理这个执行流程的细节。每个子程序执行的第一个动作是将所有在子程序中用到的寄存器的内容保存到栈中，包括结构指针和链接寄存器（如果需要的话）。接着初始化结构指针。SUB1 使用了 R2 到 R5 四个寄存器，而 SUB2 使用了 R2 和 R3 两个寄存器。这些寄存器、结构指针和 SUB1 用到的链接寄存器只有在执行 Return 指令之前才会被恢复。

使用与结构指针寄存器 FP 有关的变址寻址方式，可以从栈中加载参数以及将答案放回到栈中。这些操作中使用的字节偏移量通常是 4、8、…，如同在图 2-20 中讨论的一般栈结构一样。最后要注意的是，每个调用程序要负责从栈中删除自己的参数。这是由 Add 指令来完成的，Add 指令可以降低栈顶。

## 2.8 其他指令

到目前为止，我们已经介绍了下列指令：Load、Store、Move、Clear、Add、Subtract、Branch、Call 和 Return。使用这些指令和表 2-1 中的寻址方式，我们已经可以编写程序来说明机器指令的执行序列了，包括转移和子程序的链接。在这一节中，我们再介绍几条指令，这些指令是在大部分指令集中都存在的。

### 2.8.1 逻辑指令

按位 AND（与）、OR（或）和 NOT（非）等逻辑操作，是数字电路的基本构件，附录 A 中对数字电路进行了介绍。逻辑操作也可以在软件中执行，用一些指令将这些操作独立或是并行地施加于一个字或一个字节的所有位上。例如，指令

```
And   R4, R2, R3
```

对寄存器 R2 和 R3 中的操作数进行按位 AND（与）运算，并且将结果存在 R4 中。这条指令的立即数形式可能是：

```
And    R4, R2, #Value
```

在这里 Value 本来是一个 16 位的逻辑值，通过向其 16 个最高有效位填充 0 来将其扩展到 32 位。

下面考虑这条逻辑指令的应用。假设有四个 ASCII 字符被保存在 32 位的寄存器 R2 中。在某个任务中，我们希望确定最右边的字符是否为 Z，如果是 Z，就执行条件转移到 FOUNDZ。从第 1 章的表 1-1 中我们找到了字符 Z 的 ASCII 码是 01011010，用十六进制记数法表示为 5A。这三条指令的序列

```
And                  R2, R2, #0xFF
Move                 R3, #0x5A
Branch_if_[R2]=[R3]  FOUNDZ
```

65 - 67

实现了所期望的动作。And 指令将 R2 中最左边三个字符的所有位都清 0，而使最右边的字符保持不变。这是由于使用了一个右端 8 位是 1，左端 24 位是 0 的立即操作数。Move 指令将十六进制值 5A 加载到 R3 中。由于 R2 和 R3 最左边的 24 位都为 0，所以 Branch 指令将 R2 中右端的剩余字符与字符 Z 的二进制表示形式进行比较，如果相匹配，就产生一个到 FOUNDZ 的跳转。

### 2.8.2　移位和循环移位指令

有很多应用需要将一个操作数的位向右或向左移动指定数量的位。这个移动过程如何被执行，取决于操作数是一个有符号数还是一般的二进制码信息。对于一般的操作数，我们使用逻辑移位。对于有符号数，我们使用算术移位，算术移位可以保持该数的符号不变。

#### 1. 逻辑移位

有两条逻辑移位指令，一条是左移（LShiftL），另一条是右移（LShiftR）。这些指令将一个操作数移动一定数量的位，这个移动数量值由该指令中的一个计数操作数指定。逻辑左移指令的一般形式为：

68

```
LShiftL    Ri, Rj, count
```

该指令将寄存器 $R_j$ 的内容左移计数操作数所给定的位数，并把结果放到寄存器 $R_i$ 中，而 $R_j$ 的内容没有改变。计数操作数可以作为一个立即操作数给出，也可以包含在一个处理器寄存器中。为了完善左移操作的描述，我们需要指明目的操作数右边空出位的值，并确定在左端移出的位上发生了什么。空出的位置用 0 填充。在不使用条件码标志的计算机中，移出的位被简单地丢弃。在使用条件码标志的计算机中（将在 2.10.2 节中讨论），这些位经过进位标志 C，然后再被丢弃。将 C 标志一起参与移位，这在对占用多个字的大数执行算术运算的时候是很有用的。图 2-23a 给出了一个例子，它将寄存器 R3 的内容左移两位。逻辑右移指令 LShiftR 除了向右移以外，其他工作方式与逻辑左移指令是相同的。图 2-23b 说明了这种操作。

#### 2. 数字打包示例

考虑以下的小任务，它说明了移位操作和逻辑操作的用法。假设两个 ASCII 码表示的十进制数存储在存储器的字节单元 LOC 和 LOC+1 中。我们希望将这些十进制数用 4 位 BCD 码表示，并将两个数的 BCD 码都存放到一个单独的字节单元 PACKED 中。上述的结果被称为是打包 BCD 码（packed-BCD）格式的。第 1 章的表 1-1 给出了对应于十进制数 BCD 码的 ASCII 码最右边的四位数。因此，我们需要做的就是在 LOC 和 LOC+1 中提取出低 4 位的内容，然后

将它们拼接成一个单字节放在 PACKED 中。

图 2-24 所示的指令序列完成这个任务，它用寄存器 R2 作为指向存储器中 ASCII 码字符的指针，并用寄存器 R3 和 R4 来生成 BCD 数字码。程序使用 LoadByte 指令，从存储器中加载一个字节到 32 位处理器寄存器最右边的 8 位，并将剩下的高位清为 0。StoreByte 指令将源寄存器中最右边的字节写入到指定的目的单元中，但是不影响其他的字节单元。And 指令中的值 0xF 用于将 R4 中除最右边四位以外的其他所有位清 0。注意，立即源操作数被写成 0xF，将它被解释为一个 32 位的模式，其最高有效位上有 28 个 0。

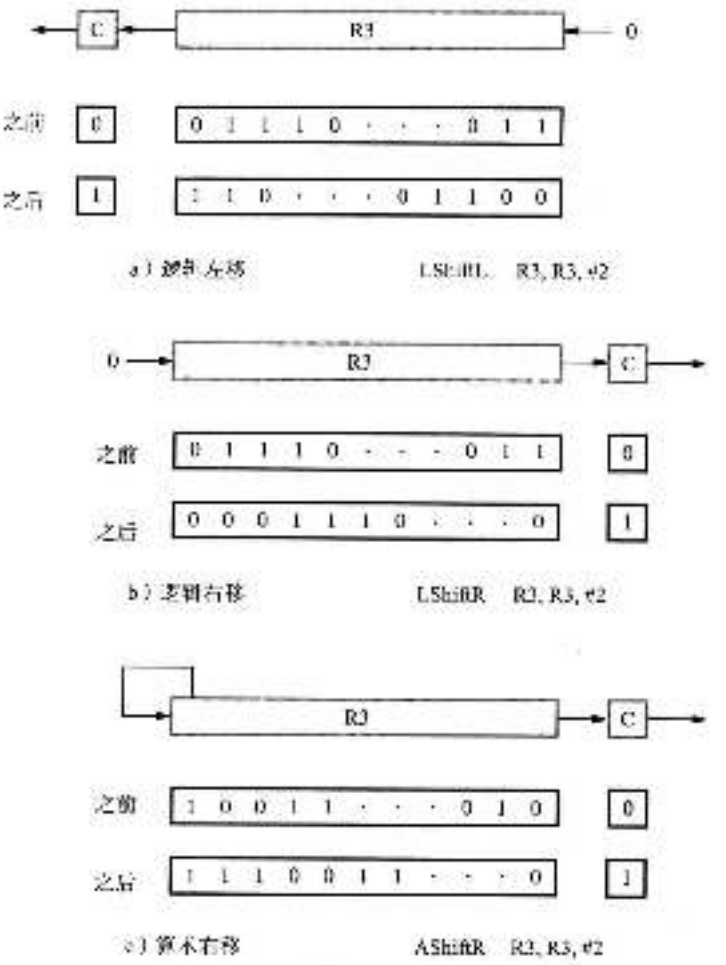



图 2-23 逻辑和算术移位指令

```
Move      R2, #LOC      R2 指向数据
LoadByte  R3, (R2)      将第一个字节正载到 R3 中
LShiftL   R3, R3, #4    左移 4 位
Add       R2, R2, #1    递增指针
LoadByte  R4, (R2)      将第二个字节加载到 R4 中
And       R4, R4, #0xF  将高位清 0
Or        R3, R3, R4    连接 BCD 数
StoreByte R3, PACKED    保存结果
```

图 2-24 一个将两个 BCD 数打包到一个字节中的程序

### 3. 算术移位

在算术移位中，将被移位的位模式解释成一个有符号数。图 1-3 中补码的二进制数表示法表明将一个数向左移动一位就相当于将这个数乘以 2；而向右移动一位相当于将这个数除以 2。当然，在左移时可能会产生溢出，而右移时余数可能会被丢失。另一个要注意的是，由于数的补码表示法的要求，在右移中必须在空位上重写符号位进行填充。在右移时这种要求将算术移位与逻辑移位区别开来，在逻辑移位中总是用 0 对空位进行填充，没有这种要求的话，这两种移位是一样的。在图 2-23c 中给出了算术右移指令 AShiftR 的一个例子，算术左移与逻辑左移是完全相同的。

69

### 4. 循环移位操作

在移位操作中，除了最后一个移出位保存在进位标志 C 中以外，操作数的其他移出位都被丢掉了。针对希望保留所有位的情况，可以使用循环移位指令。这些指令将操作数一端移出的位移动到操作数的另一端。通常循环左移和循环右移都有两种版本的指令。在一种版本里，

操作数的位被简单地进行循环。在另一种版本里，循环中包括了 C 标志。图 2-25 给出了在循环中包含或不包含 C 标志的循环左移和循环右移操作。要注意的是，当循环中不包括 C 标志时，C 标志中仍然保留着寄存器尾部的最后一个移出位。OP 码 RotateL、RotateLC、RotateR 和 RotateRC 表示执行循环移位操作的指令。

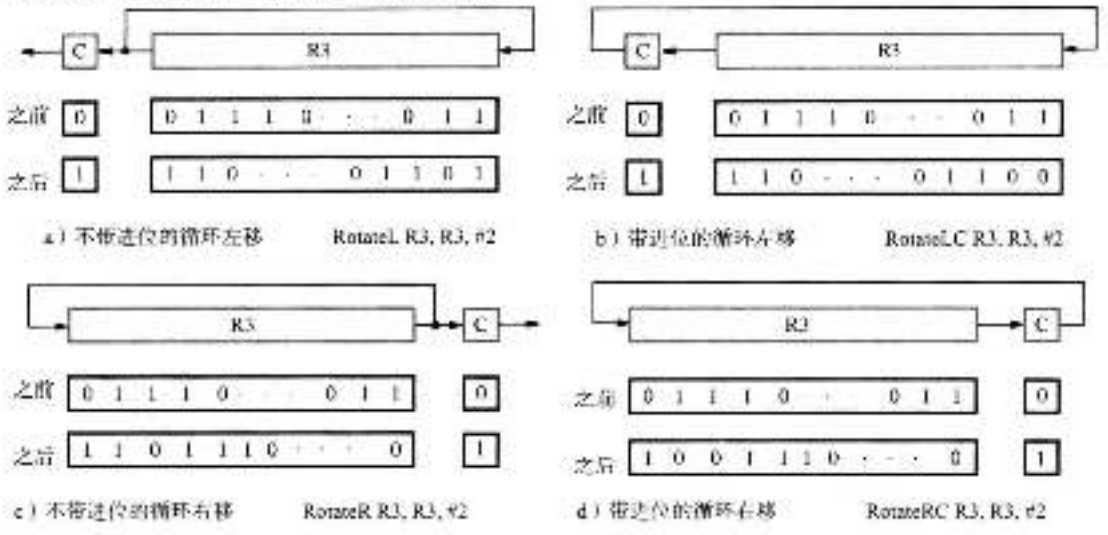



图 2-25　循环移位指令

### 2.8.3　乘法和除法

与我们之前看到过的 Add 指令的格式一样，两个有符号整数可以用机器指令进行乘和除。指令

Multiply　R*k*, R*i*, R*j*

执行的操作为：

R*k* ← [R*i*] × [R*j*]

两个 $n$ 位数的乘积可以是 $2n$ 位大的数。因此，乘积肯定不能放进寄存器 R*k* 中。许多指令集中都有一条乘法（Multiply）指令，它计算乘积的低 $n$ 位并把它放在寄存器 R*k* 中，就像说明的那样。如果已知在一些特殊的应用任务中所有乘积都将是不超过 $n$ 位的数，这是足够的。为了适应一般的 $2n$ 位乘积的情况，一些处理器将乘积生成在两个寄存器中，通常是相邻的寄存器 R*k* 和 R($k+1$)，其中高位部分放在寄存器 R($k-1$) 中。

一个指令集还可以提供一条有符号整数的除法（Divide）指令

Divide　R*k*, R*i*, R*j*

它执行的操作是：

R*k* ← [R*j*] / [R*i*]

把得到的商放到 R*k* 中。余数可以放在 R($k+1$) 中，或者可以丢掉。

对于没有乘法（Multiply）和除法（Divide）指令的计算机，可以用一些基本的指令序列比如加（Add）、减（Subtract）、移位（Shift）和循环移位（Rotate）来完成乘除操作以及其他的算术操作。当我们在第 9 章中描述算术运算的实现时，这些将会变得更清楚。

## 2.9　处理 32 位的立即值

在 2.4.1 节对寻址方式的讨论中，我们提出了一个问题，即一个代表常数或存储器地址的

32位值是如何被装入处理器寄存器中的。RISC风格处理器中的立即寻址和绝对寻址方式将操作数的大小限制为16位。因此，一个32位的值不能在单条指令中明确地给出，因为单条指令必须要放入一个32位的字中。

为了这个目的，一种可能的解决办法是用两条指令。RISC风格的处理器中有一种方法是使用两种不同的逻辑或（OR）操作指令。指令

Or Rdst, Rsrc, #Value

向16位立即操作数的高位补0使其扩展成32位数，然后将其与寄存器Rsrc中的内容进行或（OR）操作。如果Rsrc的值是0，那么Rdst就正好是那个扩展的32位数的值。另一条指令

OrHigh Rdst, Rsrc, #Value

将16位的立即操作数作为高位并在低位补0而形成一个32位的值。然后该值与Rsrc的内容进行或（OR）操作。通过使用这些指令，并假设R0中包含的值为0，我们可以按照下面的方式将32位值0x20004FF0装入寄存器R2中：

```
OrHigh      R2, R0, #0x2000
Or          R2, R2, #0x4FF0
```

为了更容易地编写程序，RISC风格的指令集可能包含伪指令，伪指令可以表示一个需要多条机器指令才能完成的动作。这些伪指令会被汇编程序替换为相应的机器指令序列。例如，伪指令

MoveImmediateAddress R2, LOC

可以用于将符号LOC所表示的32位地址装入寄存器R2中。在汇编程序中，它就会被两条使用16位值的指令替换掉，像上面的一样。

除了用两条指令之外还有另一个方法可以将一个32位的地址加载到寄存器中，即用多个字表示一条指令。在这种情况下，可以用一条双字指令在第一个字中给出OP码和寄存器说明，而在第二个字中含一个32位的值。这种方法在CISC风格的处理器中使用。

73

最后，注意到在前面的章节中我们总是假设单条Load和Store指令可以访问符号名所代表的存储单元。这使得示例程序更简单，也更容易阅读。如果所需要的存储器地址可以用16位指定，那么程序就会正确地运行。如果包含更长的地址，那么就必须使用上面描述的构建32位地址的方法。

## 2.10 CISC指令集

在前面的章节中，我们介绍了RISC风格的指令集。现在我们将研究复杂指令集计算机（Complex Instruction Set Computer，CISC）的一些重要特征。

一个关键的不同之处是，CISC指令集不受限于load/store体系结构（load/store architecture），在load/store体系结构中只可以对处理器寄存器中的操作数执行算术和逻辑运算。另一个关键的不同之处是指令不需要放在一个单字中。有些指令可能占据一个单字，其他的可能跨越多个字。

现代CISC处理器中的指令通常不使用三地址指令格式。大多数算术和逻辑指令使用二地址指令格式：

操作 目的操作数，源操作数

这种类型的Add指令为：

Add B, A

该指令对存储器操作数执行B ← [A] + [B]操作。当这个和被计算出来时，结果就被送到存储

器中并存储在单元 B 中，替换了这个单元中原来的内容。这意味着存储单元 B 既是源操作数又是目的操作数。

再考虑一下将两个数字相加的任务：

$$C = A + B$$

其中三个操作数可能都在存储单元中。显然，这不能通过一条单独的二地址指令完成。可以通过增加另一条二地址指令来执行这个任务，该二地址指令将一个存储单元中的内容拷贝到另一个单元中，就像下面的这条指令：

```
Move    C, B
```

该指令执行 C ← [B] 操作，同时保留单元 B 中的内容不变。于是 C ← [A] + [B] 操作现在可以用两条指令的序列来完成：

```
Move    C, B
Add     C, A
```

可以观察到，通过使用这个指令序列，单元 A 和 B 的内容都没有被覆盖。 74

在一些 CISC 处理器中，其中一个操作数可能在存储器中，但另一个必须在寄存器中。在这种情况下，完成所需任务的指令序列可以是：

```
Move    Ri, A
Add     Ri, B
Move    C, Ri
```

Move 指令的一般形式为：

```
Move    destination, source
```

其中源操作数和目的操作数都可以是一个存储单元或者是一个处理器寄存器。Move 指令包含了 Load 和 Store 指令的功能，Load 和 Store 指令我们曾在前面 RISC 风格处理器的讨论中用到过。在 Load 指令中，源操作数是一个存储单元而目的操作数是一个处理器寄存器。在 Store 指令中，源操作数是一个寄存器而目的操作数是一个存储单元。Load 和 Store 指令被限定在存储器与处理器寄存器之间移动操作数，而 Move 指令则有更广泛的适用范围。它可以用来移动立即操作数还可以在两个存储单元或者两个寄存器之间传递操作数。

### 2.10.1 其他寻址方式

大多数 CISC 处理器都有所有的五种基本寻址方式：立即方式，寄存器方式，绝对方式，间接方式和变址方式。在 CISC 处理器中通常还有三种另外的寻址方式。

**1. 自动增量和自动减量方式**

对于访问存储器中连续单元中的数据项和栈的实现来说，有两种方式特别方便。

*自动增量方式*（autoincrement mode）——操作数的有效地址是在指令中指定的一个寄存器的内容。在访问该操作数后，这个寄存器的内容就自动增加，指向存储器中的下一个操作数。

我们在一个特定的寄存器外加括号表示这个寄存器中的内容要用作有效地址，然后在后边写一个加号表示访问过操作数后递增寄存器的值。这就是自动增量方式的表示形式，写作：

$$(Ri)+$$

为了在一个 32 位字长的按字节寻址存储器中访问连续的字，这个增量必须是 4。对于具有自动增量方式的计算机，它可以用与被访问的操作数大小相符的值来递增寄存器中的内容。因此，对于一个字节大的操作数其增量是 1，对于 16 位的操作数其增量是 2，对于 32 位的操作

数其增量是4。因为操作数的大小通常是指令操作码的一部分，所以用 (Ri)+ 来表示自动增量
75 方式就足够了。

与自动增量方式相对应，还有一种逆序访问存储单元的方式：

自动减量方式（autodecrement mode）——指令中指定的寄存器的内容首先做自动减量操作，然后作为这个操作数的有效地址使用。

我们在一个特定寄存器外加括号，然后在前边写一个减号，表示在把寄存器的值读取出来用作有效地址之前，要先递减寄存器的值。这就是自动减量方式的表示形式，写作：

$$-(R_i)$$

在这种方式中，操作数按照递减地址的顺序进行访问。

读者可能会感到奇怪，为什么在自动减量方式中地址在使用之前要先做递减，而在自动增量方式中地址在使用之后才做递增。这样做的主要原因是为了更容易地利用这些方式去实现一个栈结构。我们不需要使用下面的两条指令：

```
Subtract    SP, #4
Move        (SP), NEWITEM
```

来将一个新项压入栈中，而只需要使用下面这一条指令来完成该操作：

```
Move    -(SP), NEWITEM
```

同样，也不需要使用两条指令：

```
Move        ITEM, (SP)
Add         SP, #4
```

来从栈中弹出一个项，我们可以只使用如下的一条指令来完成该操作：

```
Move    ITEM, (SP)+
```

**2. 相对方式**

我们已经用处理器的通用寄存器定义了变址方式。一些计算机还在这种方式中用程序计数器 PC 代替通用寄存器进行变址寻址。这时，可以使用 X(PC) 来对一个存储单元进行寻址，该存储单元离程序计数器当前指向的位置有 X 个字节的距离。由于被寻址的单元是相对于程序计数器而确定的，而程序计数器总是指向一个程序中的当前执行位置，所以相对方式的名字由此而来。

相对方式（relative mode）——在变址寻址方式中用程序计数器代替通用寄存器 Ri 来获得
76 有效地址。

## 2.10.2 条件码

处理器执行的运算通常会产生如正数、负数或 0 这样的结果。处理器可以维护着这些结果信息以便随后的条件转移指令使用。这可以通过在单独的位中记录所需要的信息来实现。这些位通常被称为条件码标志（condition code flags）。这些标志通常被集中保存在一个叫做条件码寄存器（condition code register）或是状态寄存器（status register）的特殊处理器寄存器中。每个单独的条件码标志可以根据所执行操作的结果，被设置为 1 或清除为 0。

四个常用的标志是：

N（负数） 如果结果是负数则置为 1；否则清除为 0。

Z（零） 如果结果是 0 则置为 1；否则清除为 0。

V（溢出） 如果发生算术溢出则置为 1；否则清除为 0。

C（进位） 如果运算结果有一个进位输出则置为 1；否则清除为 0。

N 和 Z 标志记录算术或逻辑运算的结果是否为负数或零。在一些计算机中，它们还可能受 Move 指令中操作数的值的影响。这使得以后的条件转移指令可以根据被移动的操作数的符号和值来进行转移。有些计算机还提供一条特殊的 Test 指令，它检查寄存器或存储器中的一个值但不修改这个值，并根据检查的情况相应地设置或清除 N 和 Z 标志。

V 标志表明是否发生了溢出。就像在 1.4 节解释的那样，当一个算术运算的结果超出了操作数可用位数所能表示的值的范围时，就产生了溢出。处理器对 V 标志进行设置，以允许程序员去测试是否已经发生了溢出并转移到一个适当的程序中去处理这个问题。像 Branch_if_overflow 这样的指令通常就是为这一目的而提供的。

如果在算术运算中从最高有效位产生一个进位，C 标志就被置为 1。这个标志使得在执行算术运算中的操作数可以比处理器的字长还要长。这种操作可以用在多精度的算术运算中，这些将在第 9 章中讨论。

考虑一下图 2-6 中的 Branch 指令。如果使用条件码，只要寄存器 R2 的内容仍然大于 0，那么 Subtract 指令将会使得 N 和 Z 标志都被清为 0。所需的转移可以简单地指定为：

Branch>0 LOOP

而不用在条件测试中指明所涉及的寄存器。如果 N 和 Z 都不是 1，也就是说 Subtract 指令产生的结果既不为负数又不等于 0，那么这条指令就会引起一个转移。计算机的指令集中提供了很多条件转移指令，以满足各种条件的测试。这些条件被定义为包含条件码标志的逻辑表达式。

为了说明条件码的使用，我们再来考虑图 2-8 中的程序，该程序是用 RISC 风格的指令将一个列表中的数相加。如果使用 CISC 风格的指令，则可以用更少的指令来实现这个任务，如图 2-26 所示。Add 指令利用指针寄存器（R4）去访问列表中连续的数，并将它们加到寄存器 R3 的和中。因为使用了自动增量寻址方式来指定源操作数，所以在访问了源操作数之后，处理器自动增加该指针。Subtract 指令用来设置条件码，然后该条件码会被 Branch 指令使用。

77

```
        Move      R2, N          加载列表的大小
        Clear     R3             将和初始化为 0
        Move      R4, #NUM1      加载第一个数当加址
LOOP:   Add       R3, (R4)+      把下一个数加到和中
        Subtract  R2, #1         递减计数器
        Branch>0  LOOP           如果没有完成就循环回到前面
        Move      SUM, R3        存储最终的和
```

图 2-26　图 2-8 中程序的 CISC 版本

## 2.11　RISC 和 CISC 风格

RISC 和 CISC 是两种不同风格的指令集。我们首先介绍了 RISC，因为它更简单，更容易理解。在看到这两种风格指令集的基本特征后，我们来总结一下它们的主要特点：

RISC 风格指令集的特点是：

- 寻址方式简单。
- 每一条指令都能放到一个单独的字中。
- 由于寻址方式简单，所以指令集中的指令数较少。
- 只能对处理器寄存器中的操作数执行算术和逻辑运算。
- Load/store 体系结构不允许从一个存储单元直接传输数据到另一个单元，这样的传输必须要通过一个处理器寄存器。

- 指令简单，这有助于处理单元使用如在第6章中所介绍的流水线技术来快速地执行指令。
- 程序的大小往往较大，因为执行复杂的任务需要较多但较简单的指令。

CISC风格指令集的特点是：

- 寻址方式比较复杂。
- 指令比较复杂，一条指令可能跨多个字。
- 有很多用来实现复杂任务的指令。
- 对存储器中的操作数以及处理器寄存器中的操作数都可以执行算术和逻辑运算。
- 通过使用Move指令可以从一个存储单元传输数据到另一个单元。
- 程序的大小往往较小，因为执行复杂的任务需要较少但较复杂的指令。

在20世纪70年代以前，所有计算机都是CISC风格的。这样做的一个重要目的是通过让硬件执行非常复杂的任务，从而简化软件的开发，换言之，就是把复杂性从软件层面转移到硬件层面。这样有助于使程序变得更简短，在计算机存储器比较小并且比较昂贵时，这是很重要的。而如今，存储器价格低廉，大多数计算机都有大容量的存储器。

RISC风格的设计试图通过简单的硬件来实现非常高的性能，这样就可以用第6章中将要讨论的流水线模式快速地执行指令。这使得复杂性从硬件层面转移到了软件层面。因此较复杂的编译器就被开发了出来，用来优化由简单指令组成的代码。随着存储器容量的增加，代码的大小已变得不那么重要了。

虽然RISC和CISC风格看起来像是定义了两种显著不同的方法，但现代的处理器往往是这两种方法的一个折中。例如，为了减少所执行指令的数量，可以在RISC处理器中加入一些非RISC的指令，只要这些新指令能被很快地执行，这将是很有吸引力的。我们将在第6章中讨论流水线概念的同时，深入探讨性能方面的问题。

## 2.12 实例程序

在这一节里，我们给出两个例子来进一步说明机器指令的使用。这些例子代表了数字的和非数字的应用。

### 2.12.1 向量点积程序

第一个例子是一个关于数字的应用程序，它是前面那个将数字相加的程序的扩展形式。在计算中包括了向量和矩阵，这对于计算两个向量的点积通常是必要的。假设A和B是两个长度为$n$的向量。它们的点积被定义为：

$$\text{点积} = \sum_{i=0}^{n-1} A(i) \times B(i)$$

图2-27和图2-28分别给出了计算点积并将其存储到存储单元DOTPROD中的RISC和CISC风格的程序。每个向量的第一个元素$A(0)$和$B(0)$被存储到存储单元AVEC和BVEC中，其余的元素就存储在随后的字单元中。

这种累加乘积和的任务常出现在许多信号处理应用中。在这种情况下，其中的一个向量由输入到信号处理单元的连续时序信号中取出$n$个最近的信号样本组成。另一个向量是$n$个权重的集合。这$n$个信号样本乘以这些权重，再将这些乘积相加就组成了一个输出信号样本。

有些计算机指令集将图2-27和图2-28程序中使用的Multiply和Add指令操作结合起来，成为一条单独的MultiplyAccumulate指令。这将在附录D中介绍的ARM处理器中说明。

```
        Move        R2, #AVEC       R2 指向向量 A
        Move        R3, #BVEC       R3 指向向量 B
        Load        R4, N           R4 作为计数器
        Clear       R5              R5 累加点积
LOOP:   Load        R6, (R2)        获取向量 A 的下一个元素
        Load        R7, (R3)        获取向量 B 的下一个元素
        Multiply    R8, R6, R7      计算下一对元素的积
        Add         R5, R5, R8      累加到前面的和中
        Add         R2, R2, #4      增加指向向量 A 的指针
        Add         R3, R3, #4      增加指向向量 B 的指针
        Subtract    R4, R4, #1      递减计数器
        Branch_if_[R4]>0  LOOP      如果没有完成再次循环
        Store       R5, DOTPROD     将点积保存到存储器中
```

图 2-27　计算两个向量点积的 RISC 风格的程序

## 2.12.2　字符串搜索程序

作为非数字应用的一个例子，我们来考虑一下字符串搜索的问题。假设有两个由 ASCII 编码字符组成的字符串，一个长字符串 $T$ 和一个短字符串 $P$，我们要确定目标 $T$ 中是否包含模式 $P$。由于可能会在 $T$ 中的多个地方找到 $P$，所以当从左往右搜索 $T$ 时，我们将只关心 $P$ 在 $T$ 中的第一次出现，从而简化我们的任务。假设 $T$ 和 $P$ 分别包含 $n$ 和 $m$ 个字符，其中 $n>m$。字符存储在存储器中连续的字节单元中。假设所需的数据按以下规定放置：

```
        Move        R2, #AVEC       R2 指向向量 A
        Move        R3, #BVEC       R3 指向向量 B
        Move        R4, N           R4 作为计数器
        Clear       R5              R5 累加点积
LOOP    Move        R6, (R2)+
        Multiply    R6, (R3)+       计算下一组元素的积
        Add         R5, R6          累加到前面的和中
        Subtract    R4, #1          递减计数器
        Branch>0    LOOP            如果没有完成再次循环
        Move        DOTPROD, R5     将点积保存到存储器中
```

图 2-28　计算两个向量点积的 CISC 风格的程序

- T 为 $T(0)$ 的地址，其中 $T(0)$ 为字符串 $T$ 的第一个字符。
- N 是一个值为 $n$ 的 32 位字的地址。
- P 为 $P(0)$ 的地址，其中 $P(0)$ 为字符串 $P$ 的第一个字符。
- M 是一个值为 $m$ 的 32 位字的地址。
- RESULT 是将要存储搜索结果的字的地址。如果在 $T$ 中找到了子字符串 $P$，那么 $T$ 中相应位置的地址就会被存放在 RESULT 中；否则，就会在 RESULT 中存放值 −1。

字符串搜索是一个重要且研究很充分的问题。已经开发了很多算法。由于我们的主要目的是说明汇编语言指令的用法，所以我们将使用最简单的算法，即所谓的 brute-force 字符串搜索算法。它在图 2-29 中给出。

```
for   i ← 0 to n − m do
      j ← 0
      while j < m and P[j] = T[i + j] do
        j ← j + 1
      if j = m return i
return −1
```

图 2-29　用 brute-force 字符串搜索算法搜索字符串

在 RISC 风格的计算机中，这个算法可以如图 2-30 所示的那样去实现。图中的注释说明了各种处理器寄存器的用法。注意，在搜索失败的情况下，立即值 −1 将使得 R8 的内容等于 0xFFFFFFFF，即 −1 的补码。

图 2-31 展示了这个算法是如何在 CISC 风格的计算机中实现的。观察到 LOOP2 中的第一条指令将一个字符从字符串 $T$ 中装入寄存器 R8 中，接下来的一条指令将这个字符与字符串 $P$ 中的字符比较。读者可能会感到奇怪，为什么不能使用这样的单条指令

```
CompareByte   (R6)+, (R7)+
```

80
|
81

来达到相同的效果。CISC 风格指令集允许操作中包含存储器操作数，它们通常这样要求：如果一个操作数在存储器中，另一个操作数就必须在处理器寄存器中。一个常见的例外是 Move 指令，它可能包含两个存储器操作数。这就提供了一种在不同的存储单元中移动数据的简单方法。

```
        Move          R2, #T        R2 指向字符串 T
        Move          R3, #P        R3 指向字符串 P
        Load          R4, N         获取值 n
        Load          R5, M         获取值 m
        Subtract      R4, R4, R5    计算 n-m
        Add           R4, R2, R4    T(n-m) 的地址
        Add           R5, R3, R5    P(m) 的地址
LOOP1:  Move          R6, R2        用 R6 遍历字符串 T
        Move          R7, R3        用 R7 遍历字符串 P
LOOP2:  LoadByte      R8, (R6)      比较字符串 T 和 P 中的
        LoadByte      R9, (R7)      一对字符
        Branch_if_[R8]≠[R9]  NOMATCH
        Add           R6, R6, #1    指向 T 中的下一个字符
        Add           R7, R7, #1    指向 P 中的下一个字符
        Branch_if_[R5]>[R7]  LOOP2  如果没有完成再次循环
        Store         R2, RESULT    存储 T(i) 的地址
        Branch        DONE
NOMATCH: Add          R2, R2, #1    指向 T 中的下一个字符
        Branch_if_[R4]>[R2]  LOOP1  如果没有完成再次循环
        Move          R8, #-1       写入 -1 表明没有发现匹配
        Store         R8, RESULT
DONE:   next instruction
```

图 2-30 一个 RISC 风格的字符串搜索程序

```
        Move          R2, #T        R2 指向字符串 T
        Move          R3, #P        R3 指向字符串 P
        Move          R4, N         获取值 n
        Move          R5, M         获取值 m
        Subtract      R4, R5        计算 n-m
        Add           R4, R2        T(n-m) 的地址
        Add           R5, R3        P(m) 的地址
LOOP1:  Move          R6, R2        用 R6 遍历字符串 T
        Move          R7, R3        用 R7 遍历字符串 P
LOOP2:  MoveByte      R8, (R6)+     比较字符串 T 和 P 中的一对
        CompareByte   R8, (R7)+     字符
        Branch≠0      NOMATCH
        Compare       R5, R7        检查是否在 P(m) 中
        Branch >0     LOOP2         如果没有完成再次循环
        Move          RESULT, R2    存储 T(i) 的地址
        Branch        DONE
NOMATCH: Add          R2, #1        指向 T 中的下一个字符
        Compare       R4, R2        检查是否在 T(n-m) 中
        Branch ≥ 0    LOOP1         如果没完成再次循环
        Move          RESULT, #-1   没发现匹配
DONE:   next instruction
```

图 2-31 一个 CISC 风格的字符串搜索程序

## 2.13 机器指令的编码

在这一章中，我们已经介绍了各种有用的指令和寻址方式。我们使用了汇编语言的一般形式去强调基本概念，而避免使用特定处理器的首字母缩写词或助记符。汇编语言指令以符号

的形式表示必须由处理器电路执行的操作。就像在 2.5 节讨论的那样，为了在处理器中执行，汇编语言指令必须由汇编程序转换成机器指令，这些机器指令是按紧凑的二进制模式编码的。

现在我们来研究机器指令的几种可能的格式。Add 指令

Add　Rdst, Rsrc1, Rsrc2

表示的是操作数都在处理器寄存器中的三操作数指令。寄存器 Rdst、Rsrc1 和 Rsrc2 包含目的操作数和两个源操作数。如果处理器有 32 个寄存器，那么这就需要在这样的指令中用 5 个位去指定每一个寄存器。如果每条指令是在 32 位字中实现的，那么剩下的 17 位可以用来指定 OP 码，去明将要执行的操作。在图 2-32a 中展示了一种可能的格式。




图 2-32　可能的指令格式

现在考虑一类指令，其中一个操作数使用立即寻址方式给出，如

Add Rdst, Rsrc, #Value

32 个可用位中，需要 10 位来指定两个寄存器。剩下的 22 位必须给出 OP 码以及立即操作数的值。立即操作数最常用的大小为 32 位、16 位和 8 位。由于不可能为立即操作数分配 32 位，所以一个较好的选择是给它分配 16 位。这就留下了 6 位来指定 OP 码。图 2-32b 中给出了一种可能的格式。这个格式也可用于 Load 和 Store 指令，在这些指令中变址寻址方式使用 16 位的字段去指定与变址寄存器中的内容相加的偏移量。

图 2-32b 中的格式也可以用于对 Branch 指令进行编码。考虑图 2-12 中的程序。如果寄存器 R0 的内容为 0，那么在存储器地址 128 中的 Branch-greater-than 指令可以用特定的汇编语言写成

82 / 83

BGT　R2, R0, LOOP

寄存器 R2 和 R0 可以在图 2-32b 中的两个寄存器字段中指定。6 位的 OP 码需要用来标记 BGT 操作。16 位的立即数字段用来提供确定转移目标地址所需要的信息，即指令中带有 LOOP 标记的位置。目标地址通常由 32 位组成。由于没有 32 位的空间，BGT 指令利用立即数字段给出了该指令在程序中的位置到所要求的转移目标的偏移量。在 BGT 指令被执行的时候，程序计数器 PC 的值已被增加至指向下一条指令，即在地址 132 中的 Store 指令。因此，转移偏移量为 132 − 112 = 20。由于处理器是通过将 PC 的当前内容与转移偏移量相加来计算目标地址的，所以在这个例子中所需的偏移量为负数，即 −20。

最后，我们应该来考虑一下用来调用子程序的 Call 指令。它只需要指定 OP 码和一个用来确定子程序第一条指令地址的立即值。如果使用 6 位来指定 OP 码，那么剩下的 26 位可以用来表示立即值。在图 2-32c 中给出了这种格式。

在本节中，我们介绍了机器指令编码的基本概念。不同的商用处理器具有不同实现细节

84 的指令集。附录 B 到 E 中我们选择了四种处理器指令集作为例子来进行介绍。

## 2.14 结束语

这一章从程序员的角度介绍了汇编与机器级的指令和程序的表示与执行。在讨论中强调了寻址技术和指令序列的基本原则。程序设计示例说明了使用现代计算机指令集实现操作的基本类型。介绍了常用的寻址方式。对于程序的概念和实现子程序所需要的指令也进行了讨论。在本章的讨论中，我们还对比了两种不同的机器指令集（RISC 和 CISC）的设计方法。

## 2.15 问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。

**例 2.1**

**问题**：假设有一个由 ASCII 编码字符组成的字符串存放在存储器中起始地址为 STRING 的连续单元中。这个字符串以回车（CR）符结束。写一个 RISC 风格的程序来确定字符串的长度并将其存储在 LENGTH 单元中。

**解答**：图 2-33 给出了一个可能的程序。字符串中的字符都与 CR（ASCII 码为 0x0D）进行比较，递增计数器直到到达该字符串的末尾。

```
        Move            R2, #STRING     R2 指向字符串的开始处
        Clear           R3              R3 是计数器并被清为 0
        Move            R4, #0x0D       回车符的 ASCII 码
LOOP:   LoadByte        R5, (R2)        获取下一个字符
        Branch_if_[R5]!=[R4] DONE       如果字符为 CR 则结束
        Add             R2, R2, #1      递增字符串指针
        Add             R3, R3, #1      递增计数器
        Branch          LOOP            如果没有完成，就循环回到前面
DONE:   Store           R3, LENGTH      将计数值存放在单元 LENGTH 中
```

85 图 2-33 例 2.1 的程序

**例 2.2**

**问题**：我们想在一个 32 位正整数的列表中找出最小的数。在找到最小的数后，把它的值存放在地址为 1000 的字中。下一个字存放列表中的项数 $n$。接下来的 $n$ 个字保存列表中的数。程序在地址 400 处开始。写一个 RISC 风格的程序来找出最小的数。程序中需包含按指定方式组织程序和数据所需的汇编指示符。虽然程序必须能够处理不同长度的列表，但是在你的代码中只需包含由七个整数组成的样本数据列表。

**解答**：图 2-34 中的程序可以完成所需的任务。程序中的注释说明了该任务是如何执行的。

**例 2.3**

**问题**：写一个 RISC 风格的程序将一个 $n$ 位的十进制整数转换成二进制数。这个十进制数以 $n$ 个 ASCII 编码字符的形式给出，就像在键盘中键入数字的情况一样。存储单元 N 中存放值 $n$，ASCII 字符串从 DECIMAL 处开始，转换后的数字存放在 BINARY 中。

**解答**：考虑一个 4 位的十进制数，$D=d_3d_2d_1d_0$。这个数的值为 $((d_3\times 10+d_2)\times 10+d_1)\times 10+d_0$。该数字的这种表示形式是图 2-35 中程序所使用的转换技术的基础。注意，每一个 ASCII 编码字符在它用于计算之前都被转换为一个二进制编码的十进制（Binary Coded Decimal，BCD）数字。这里假设转换后的值可以用不超过 32 个位来表示。

**例 2.4**

**问题**：考虑一个数组 $A(i,j)$，其中行索引从 $i=0$ 到 $n-1$，而列索引从 $j=0$ 到 $m-1$。这个数组一行接一行地存放在计算机存储器中，每行的元素占用 $m$ 个连续的字单元。假设存储器是按字节寻址的且字长

为 32 位。写一个 RISC 风格的子程序，将第 $x$ 列与第 $y$ 列逐个元素地相加，将和元素存放在第 $y$ 列中。索引 $x$ 和 $y$ 通过寄存器 R2 和 R3 传递给子程序。参数 $n$ 和 $m$ 通过寄存器 R4 和 R5 传递给子程序。元素 A(0,0) 的地址通过寄存器 R6 传递。

```
LIST      EQU                 1000          列表的起始地址

          ORIGIN              400
          Move                R2, #LIST     R2 指向列表的开始处
          Load                R3, 4(R2)     R3 是一个计数器，初始化为 n
          Add                 R4, R2, #8    R4 指向第一个数
          Load                R5, (R4)      R5 保存着目前为止找到的最小的数
LOOP:     Subtract            R3, R3, #1    递减计数器
          Branch_if_[R3]=0    DONE          如果 R3 等于 0 则结束
          Add                 R4, R4, #4    递增列表指针
          Load                R6, (R4)      获取下一个数
          Branch_if_[R5]≤[R6] LOOP          检查是否已找到最小的数
          Move                R5, R6        更新已找到的最小的数
          Branch              LOOP
DONE:     Store               R5, (R2)      将最小的数存到 SMALL 中

          ORIGIN              1000
SMALL:    RESERVE             4             找到的最小的数的空间
N:        DATAWORD            7             列表中的项数
ENTRIES:  DATAWORD  4,5,3,6,1,8,2           列表项
          END
```

图 2-34　例 2.2 的程序

```
          Load                R2, N         初始化计数器 R2 为 n
          Move                R3, #DECIMAL  R3 指向 ASCII 数字
          Clear               R4            R4 存放二进制数
LOOP:     LoadByte            R5, (R3)      获取下一个 ASCII 数字
          And                 R5, R5, #0x0F 形成 BCD 数字
          Add                 R4, R4, R5    加到中间结果中
          Add                 R3, R3, #1    递增数字指针
          Subtract            R2, R2, #1    递减计数器
          Branch_if_[R2]=0    DONE
          Multiply            R4, R4, #10   乘以 10
          Branch              LOOP          如果没有完成循环回到前面
DONE:     Store               R4, BINARY    将结果存到单元 BINARY 中
```

图 2-35　例 2.3 的程序

**解答**：图 2-36 给出了一个可能的程序。我们假设值 $x$、$y$、$n$ 和 $m$ 存放在存储单元 X、Y、N 和 M 中。同样，数组的元素也存放在以 ARRAY 单元开始的连续的字中。ARRAY 单元就是元素 A(0,0) 的地址。程序中的注释说明每条指令的目的。

**例 2.5**

**问题**：我们想要对存放在存储器中的一个字符列表进行排序。这个列表包含 $n$ 个字节，且每个字节都包含从 A 到 Z 的字母集合中一个字符的 ASCII 码。在第 1 章介绍过的 ASCII 码中，字母 A、B、…、Z 用具有递增数值的 7 位二进制模式表示。当一个 ASCII 码字符存储在一个字节单元中时，习惯上会将其最高有效位置为 0。利用这种编码，我们可以通过将字符的编码（将它们作为正数）按数字递增的顺序排序以此来将字符列表按字母顺序进行排序。

假设这个列表存储在从 LIST 到 LIST + $n$ − 1 的存储单元中，$n$ 为存储在地址 N 处的 32 位值。排序要在适当的位置进行，也就是说，排序后的列表要与原始列表占用相同的存储单元。

86
-
87

```
        Load        R2, X          加载值 x
        Load        R3, Y          加载值 y
        Load        R4, N          加载值 n
        Load        R5, M          加载值 m
        Move        R6, #ARRAY     加载 A(0,0) 的地址
        Call        SUB
        next instruction
        :
SUB:    Subtract    SP, SP, #4
        Store       R7, (SP)       保存寄存器 R7
        LShiftL     R5, R5, #2     确定一列中连续的元素之间的
                                   距离（以字节为单位）
        Subtract    R3, R3, R2     计算 y - x
        LShiftL     R3, R3, #2     计算 4(y - x)
        LShiftL     R2, R2, #2     计算 4x
        Add         R6, R6, R2     R6 指向 A(0,x)
        Add         R7, R6, R3     R7 指向 A(0,y)
LOOP:   Load        R2, (R6)       获取第 x 列中的下一个数
        Load        R3, (R7)       获取第 y 列中的下一个数
        Add         R2, R2, R3     将这些数相加并存储和
        Store       R2, (R7)
        Add         R6, R6, R5     递增指向第 x 列的指针
        Add         R7, R7, R5     递增指向第 y 列的指针
        Subtract    R4, R4, #1     递减行计数器
        Branch_if_[R4]>0, LOOP     如果没有完成则循环回到前面
        Load        R7, (SP)       恢复 R7
        Add         SP, SP, #4
        Return                     返回到调用程序
```

图 2-36 例 2.4 的程序

我们可以利用直接选择排序算法对列表进行排序。首先，找到最大的数并将其放到列表末尾的单元 LIST + $n$ − 1 中。然后将剩下的 $n-1$ 个数的子列表中最大的数放到子列表末尾的单元 LIST + $n$ − 2 中。重复这个过程直到列表被排序完毕。在图 2-37 中展示了这个排序算法的 C 语言程序，在该程序中列表被视为一个从 LIST(0) 到 LIST($n-1$) 的一维数组。对每一个从 LIST($j$) 到 LIST(0) 的子列表，LIST($j$) 中的数都要与子列表中每一个其他的数进行比较。每当在子列表中找到一个更大的数，就将它与 LIST($j$) 中的数进行交换。

```
for   (j = n-1; j > 0; j = j - 1)
      {for (k = j-1; k > = 0; k = k - 1)
         {if    (LIST[k] > LIST[j])
                { TEMP = LIST[k];
                  LIST[k] = LIST[j];
                  LIST[j] = TEMP;
                }
         }
      }
```

图 2-37 实现排序的 C 语言程序

注意，C 语言程序是向后遍历列表的。当编写机器语言程序时，这种遍历顺序将简化循环的终止，因为当索引减到 0 时就退出循环。

写一个 CISC 风格的程序，实现该排序任务。

**解答：** 图 2-38 给出了一个可能的程序。

```
        Move        R2, #LIST    加载 LIST 到基址寄存器 R2 中
        Move        R3, N        初始化外部循环的索引寄存器 R3
        Subtract    R3, #1       为 j=n-1
OUTER:  Move        R4, R3       初始化内部循环的索引寄存器 R4
        Subtract    R4, #1       为 k=j-1
        MoveByte    R5, (R2,R3)  加载 LIST(j) 到 R5 中，R5 保存
                                 着当前子列表中的最大值
INNER:  CompareByte (R2,R4), R5  如果 LIST(k) ≤ [R5]，则不交换
        Branch ≤0   NEXT
        MoveByte    R6, (R2,R4)  否则，将 LIST(k) 和 LIST(j) 交换，
        MoveByte    (R2,R4), R5  并将新的最大值装入 R5 中
        MoveByte    (R2,R3), R6
        MoveByte    R5, R6       寄存器 R6 作为 TEMP
NEXT:   Decrement   R4           递减变址寄存器 R4 和 R3，它们也
        Branch >0   INNER        作为循环计数器，在循环没有完成
        Decrement   R3           时就跳转回前面
        Branch>0    OUTER
```

图 2-38　一个字节排序程序

88
~
89

## 习题

[E] 2.1 在一些存储单元中给出一个二进制模式，是否能说明这个模式表示的是一条机器指令还是一个数字？

[E] 2.2 考虑一台计算机，它有一个按字节寻址的存储器。该存储器按照大端策略组织为 32 位的字的集合。一个程序读取在键盘上输入的 ASCII 字符并将它们存储到从单元 1000 开始的连续字节单元中。在输入单词 "Computer" 后，说明单元 1000 和 1004 中两个存储字的内容。

[E] 2.3 针对小端策略重复习题 2.2 中的问题。

[E] 2.4 在使用以下每种寻址方式去访问一个存储器操作数之前，寄存器 R4 和 R5 包含着十进制数 2000 和 3000。在每一种情况中有效地址（EA）是多少？

(a) 12 (R4)

(b) (R4,R5)

(c) 28 (R4,R5)

(d) (R4)+

(e) -(R4)

[E] 2.5 写一个 RISC 风格的程序，计算表达式 SUM = 580 + 68 400 + 80 000。

[E] 2.6 写一个 CISC 风格的程序完成习题 2.5 中的任务。

[E] 2.7 写一个 RISC 风格的程序，计算表达式 ANSWER = A × B × C × D。

[E] 2.8 写一个 CISC 风格的程序完成习题 2.7 中的任务。

[M] 2.9 重写图 2-8 中的加法循环，使得列表中的数字可以以相反的顺序被访问；也就是说，第一个被访问的数是列表中的最后一个，最后一个被访问的数在存储单元 NUM1 中。尝试实现最有效的确定循环终止的方法。你的循环是否比图 2-8 中的循环执行得更快？

[M] 2.10 将图 2-10 所示的学生成绩列表修改为每个学生包含有 $f$ 项测验分数。假设有 $n$ 个学生。写一个 RISC 风格的程序，计算每项测验的分数和，并将这些和存储在存储器中地址为 SUM、SUM+4、SUM+8、…的字单元中。由于测验的数量 $f$ 比处理器中的寄存器数量要大，所以图 2-11 所示的用于 3 项测验的程序已不能被使用。应该使用两个嵌套循环。内部循环累加每项特定测验的和，而外部循环遍历测验的数目 $f$。假设用来存放和的存储区域最初已被清除为零。

[M] 2.11 写一个 RISC 风格的程序，在包含 $n$ 个 32 位整数的列表中找出负数的个数，并将计数结果存储在单元 NEGNUM 中。值 $n$ 存储在存储单元 N 中，列表中第一个整数存储在单元 NUMBERS

90 中，程序要包含必要的汇编指示符和一个含有六个数的样本列表，其中有一些数是负数。

[E] 2.12 以下两个语句段都可以将值 300 存储在单元 1000 中，但是却是在不同的时间完成的。

```
ORIGIN    1000
DATAWORD  300
```

以及

```
Move   R2, #1000
Move   R3, #300
Store  R3, (R2)
```

请解释这种差异。

[E] 2.13 参照图 2-13 的风格为图 2-11 中的程序写一个汇编语言程序。假设采用图 2-10 中的数据布局。

[E] 2.14 写一个 CISC 风格的程序完成例 2.1 中的任务。一条指令最多可以有一个操作数在存储器中。

[M] 2.15 写一个 CISC 风格的程序完成例 2.2 中的任务。一条指令最多可以有一个操作数在存储器中。

[M] 2.16 写一个 CISC 风格的程序完成例 2.3 中的任务。一条指令最多可以有一个操作数在存储器中。

[M] 2.17 写一个 CISC 风格的程序完成例 2.4 中的任务。一条指令最多可以有一个操作数在存储器中。

[M] 2.18 写一个 RISC 风格的程序完成例 2.5 中的任务。

[E] 2.19 在一个程序中寄存器 R5 用来指向一个包含 32 位数的栈的栈顶。使用变址、自动增量和自动减量寻址方式写一个指令序列，执行以下的各个任务：

（a）弹出栈顶部的两项内容，将它们相加，然后将结果压入栈中。

（b）从栈顶将第五项内容拷贝到寄存器 R3 中。

（c）从栈中删除栈顶部的 10 项内容。

对于每一种情况，假设栈都包含 10 个或更多的元素。

[M] 2.20 在图 2-18 的程序中，下列每一条指令执行之后，写出此时处理器栈的内容和栈指针 SP 的内容。假设在调用程序开始执行之前，[SP]=1000，且在第一级上。

（a）子程序中第一条 Store 指令

（b）子程序中最后一条 Load 指令

（c）调用程序中最后一条 Store 指令

[M] 2.21 一个子程序的返回地址可保存在：

（a）处理器的寄存器中

（b）一个与调用有关的存储单元中，因此当子程序从不同的地方被调用时将使用不同的单元

91 （c）一个堆栈中

在这些可能性中哪一种支持子程序嵌套？哪一种支持子程序递归（也就是子程序调用它自身）？

[M] 2.22 除处理器栈之外，在一些程序中使用另外一个栈可能会更加方便。通常在存储器中为第二个栈分配固定大小的空间。在这种情况下，当栈已达到其最大大小时，需要避免再把一个项压入栈中。同样也要避免从一个空栈中弹出一个项。这可能是由一个编程错误引起的。写两个简短的 RISC 风格的程序，分别叫做 SAFEPUSH 和 SAFEPOP，用于压入和弹出这个栈结构，同时避免这两个可能出现的错误。假设将被压入或弹出的元素位于寄存器 R2 中，寄存器 R5 作为这个用户栈的栈指针。如果栈中最顶层的元素存放在单元 TOP 中，那么这个栈就满了。如果最后弹出的元素存放在单元 BOTTOM 中，那么这个栈就是空的。如果发生错误，程序就应该相应地跳转到 FULLERROR 和 EMPTYERROR 中。假设所有的元素都是一个字的大小，并且栈向较低编号的地址单元增长。

[M] 2.23 重复习题 2.22 中的问题，写一个 CISC 风格的程序，可以使用自动增量和自动减量寻址方式。

[D] 2.24 另外有一个类似于栈的有用的数据结构称为队列（queue）。将数据存储到队列中和从队列中取出数据都是基于先进先出（FIFO）方法的。因此，如果我们假设队列在存储器中朝地址增加的方向增长（这是普遍的做法），新的数据就被加到队列的后面（高地址端），而从队列的前面（低地址端）取出数据。

栈和队列的实现有两个重要的差异。栈的一端是固定的（底部），另一端随着数据的压入和弹出而升高和降低。在任何时候都只需要一个指向栈的顶部的指针。而另一方面，随着数据从后面加入和从前面移出，队列的两端都会移向更高的地址，所以需要两个指针来跟踪队列的两端。

一个 FIFO 的字节队列是在存储器中实现的，它占用 K 个字节的固定区域。所需的指针有 IN 指针和 OUT 指针。IN 指针跟踪下一个将被加入到队列后面的字节的位置，而 OUT 指针跟踪包含下一个将要从队列前面移出的字节的位置。

（a）随着数据项加入到队列中，它们被加到连续的高地址中，直至到达存储区域的末端。那么当一个新的项要加入到队列中的时候，接下来会发生什么呢？

（b）为 IN 指针和 OUT 指针选择一个合适的定义，指出它们在这个数据结构中指向什么地方。用一个简单的图说明你的答案。

（c）如果队列的状态只用两个指针描述，说明队列完全满和完全空这两种情况是难以区分的。

（d）为了解决（c）部分中的问题需要增加什么条件？

（e）给出一个程序，使其可以操作两个指针 IN 和 OUT 来向队列增加数据项和从队列中删除数据项。

[92]

[M] 2.25 考虑习题 2.24 中描述的队列结构，写出 APPEND 和 REMOVE 程序使得数据可以在处理器寄存器和队列之间传输。每当试图或正在执行一个操作时，都要仔细检查和更新队列和指针的状态。

[M] 2.26 在 2.12.1 节中讨论了点积计算。这种类型的计算可以用于以下信号处理的任务。一个输入信号时间序列 IN（0），IN（1），IN（2），IN（3），…被一个三元素的权重向量 WT（0），WT（1），WT（2））=（1/8，1/4，1/2）进行处理，产生一个如下的输出信号时间序列 OUT（0），OUT（1），OUT（2），OUT（3），…：

$$OUT(0) = WT(0) \times IN(0) + WT(1) \times IN(1) + WT(2) \times IN(2)$$
$$OUT(1) = WT(0) \times IN(1) + WT(1) \times IN(2) + WT(2) \times IN(3)$$
$$OUT(2) = WT(0) \times IN(2) + WT(1) \times IN(3) + WT(2) \times IN(4)$$
$$OUT(3) = WT(0) \times IN(3) + WT(1) \times IN(4) + WT(2) \times IN(5)$$
$$\cdots$$

所有的信号和权重值都是 32 位的有符号数。权重、输入和输出分别被存放在存储器中以 WT、IN 和 OUT 为起始的单元中。写一个 RISC 风格的程序计算并存储前 $n$ 个输出的值，其中 $n$ 存储在单元 N 中。（提示：算术右移可以用来做乘法。）

[M] 2.27 写一个子程序 MEMCPY 将主存中一个区域的字节序列复制到另一个区域中。这个子程序可以接收三个在寄存器中的输入参数，分别表示源（from）地址，目的（to）地址和将要复制的序列的长度（length）。这两个区域可能会有重叠。除了这一种情况以外，在其他所有情况下，这个子程序都应该按地址递增的顺序复制这些字节。但是，在目的地址落在将要复制的字节序列中，即目的地址 to 在 from 到 from+length−1 之间的情况下，为了防止覆盖还没有被复制的字节，子程序必须从将要复制的字节序列的末端开始按地址递减的顺序复制这些字节。

[M] 2.28 写一个子程序 MEMCMP 对主存中的两个字节序列逐个字节地进行比较。这个子程序可以接收三个在寄存器中的输入参数，分别表示第一个（first）序列的地址，第二个（second）序列的地址和将要比较的序列的长度（length）。还需要使用一个寄存器来返回比较中不匹配的次数。

[M] 2.29 写一个子程序 EXCLAM，它接收一个在寄存器中的参数，该参数表示一个保存在连续字节中的 ASCII 码字符串在主存中的起始地址 STRING。这个字符串可以代表任意的句子集合，并以控制字符 NUL（值 0）结尾。这个子程序需要从地址 STRING 开始扫描字符串，将每一个出现的句号（"."）替换成感叹号（"!"）。

[93]

[M] 2.30 写一个子程序 ALLCAPS，它接收一个在寄存器中的参数，该参数表示一个保存在连续字节中的 ASCII 码字符串在主存中的起始地址 STRING，并以控制字符 NUL（值 0）结尾。这个子程序需要从地址 STRING 开始扫描字符串，将每一个出现的小写字母（"a" ~ "z"）替换成相应的大写字母（"A" ~ "Z"）。

[M] 2.31 写一个子程序 WORDS，它接收一个在寄存器中的参数，该参数表示一个保存在连续字节中的 ASCII 码字符串在主存中的起始地址 STRING，并以控制字符 NUL（值 0）结尾。这个字符串表示单词间有空格字符的英文文本。这个子程序必须确定字符串中单词的个数（除了标点符号）。它必须通过寄存器将结果返回给调用程序。

[D] 2.32 写一个子程序 INSERT，将一个数字放到一个按值升序存储的正数列表中正确的位置。需要通过寄存器向这个子程序传递三个输入参数，分别表示排序数字列表的起始地址、列表的长度和将被插入列表中的新值。该子程序需要在列表中为新值找到适当的位置，然后将所有较大的数字向上移动一个位置，在列表中创建空间来存储新值。

[D] 2.33 写一个子程序 INSERTSORT，重复使用习题 2.32 中的 INSERT 子程序，将一个无序的数字列表创建成一个将这些数字按递增顺序排列的新列表。这个子程序需要接收三个在寄存器中的输入参数，分别表示无序数字序列的起始地址 OLDLIST、列表的长度和有序数字序列的起始地址 NEWLIST。

94

# 基本输入 / 输出

## 本章目标

在本章中你将学习以下内容：

- 处理器与输入 / 输出（I/O）设备之间的数据传输
- 从程序员的角度看 I/O 传输
- 如何使用轮询方式执行程序来控制 I/O
- 如何在 I/O 传输中使用中断



计算机的一个基本特性是具有与其他设备交换信息的能力。这一通信能力使操作员可以完成很多操作，例如用键盘和显示屏幕来处理文本和图形。实际中，我们广泛地使用计算机通过 Internet 与其他计算机通信并访问全球信息。此外，在其他应用中，计算机虽然不是很明显，但仍有着同等重要的作用。它们是构成家用电器、制造设备、运输系统、银行和销售点终端所必需的一部分。在这些应用中，计算机的输入可能来自传感器开关、数字照相机、麦克风或火警报警器；输出可能是发送给扬声器的声音信号或者是用来改变发动机速度、打开阀门或使机器人按指定方式移动的数字编码指令。简而言之，计算机应具有在不同环境下与多种设备交换数字信息或模拟信息的能力。

在本章中，我们将从程序员的角度来分析计算机的输入 / 输出（I/O）功能。我们只介绍所有计算机都提供的基本 I/O 操作，这将使读者可以在典型的教学实验环境中的设备上完成有趣并有用的练习。更复杂的 I/O 方案，以及实现 I/O 功能所需要的硬件将在第 7 章中讨论。

## 3.1 访问 I/O 设备

计算机系统的部件之间通过互连网络进行通信，如图 3-1 所示。互连网络包含了在处理器、存储器和一些 I/O 设备之间进行信息传输所需的电路。

在第 2 章中，我们描述了地址空间的概念以及处理器如何访问地址空间中独立的存储单元。Load 和 Store 指令使用寻址方式来产生识别所需单元的有效地址。这种使用地址来访问存储器中不同单元的思想也可以扩展到 I/O 设备的处理上。为此，每个 I/O 设备在处理器看来必须包括一些可寻址的单元，就像存储器一样。处理器将地址空间中的一些地址分配给这些 I/O 单元，而不是主存储器。这些单元通常以寄存器的形式组织成位存储电路（触发器），习惯上把它们称为 I/O 寄存器（I/O register）。由于 I/O 设备和存储器共享同一个地址空间，所以这种方式被称为存储器映射 I/O（memory-mapped I/O）。大多数计算机都使用了这种组织方式。

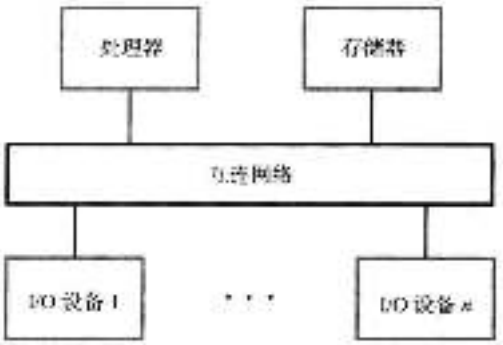



图 3-1 计算机系统



使用存储器映射 I/O 时，任何可以访问存储器的机器指令也都可以用来与 I/O 设备进行数据传送。比如，如果 DATAIN 是输入设备中一个寄存器的地址，那么指令

Load R2, DATAIN

将从DATAIN寄存器中读取数据并将其装入寄存器R2中。类似地，指令

Store R2, DATAOUT

将寄存器R2的内容发送到DATAOUT单元中。DATAOUT单元是输出设备中的一个寄存器。

### 3.1.1 I/O设备接口

一个I/O设备通过一个称为设备接口（device interface）的电路连接到互连网络中，这个电路提供了便于数据传输和管理设备操作所需的数据传输方法以及状态和控制信息的交换方法。这个接口包括一些可以被处理器访问的寄存器，一个寄存器可作为数据传输的缓冲区，另一个可以保存关于设备当前状态的信息，还有一个则可以存储控制设备操作行为的信息。这些数据（data）、状态（status）和控制（control）寄存器是通过程序指令来访问的，就好像它们是存储单元一样。典型的信息传输是在I/O寄存器和处理器寄存器之间发生的。图3-2说明了从软件的角度来看，键盘和显示设备是如何连接到处理器上的。

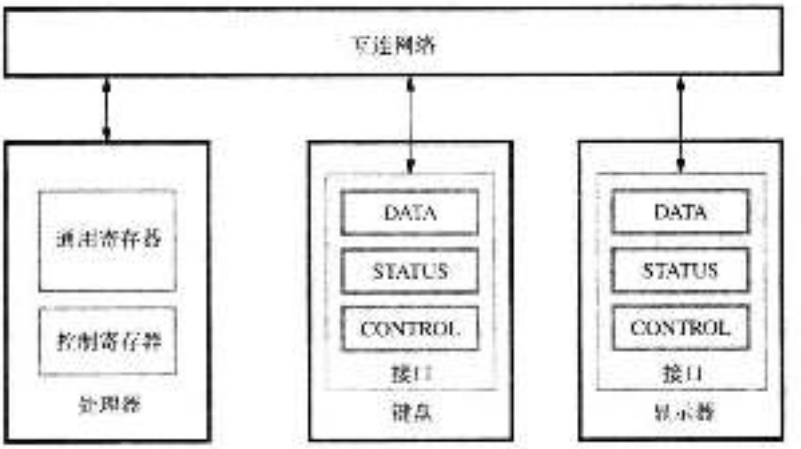



图3-2 处理器、键盘和显示器之间的连接

### 3.1.2 程序控制I/O

让我们从两个基本的人机交互I/O设备——键盘和显示器开始讨论输入/输出方面的问题。考虑这样一个任务：读取键盘上键入的字符，并将这些数据存储在存储器中，然后再在显示屏幕上显示这些字符。实现这个任务的一种简单方法就是写一个程序，让它来执行实现所需动作的所有功能，这种方法被称为程序控制I/O（program-controlled I/O）。

除了把每个字符从键盘传输到存储器中然后再传输到显示器之外，还需要保证这一切都要在正确的时刻发生。在一个键被按下的时候必须读取输入的字符。对于输出来说，只有当显示设备能够接受一个字符的时候才能将该字符发送给显示器。从键盘到计算机的数据传输速度是受用户打字速度的限制的，打字速度不可能超过每秒几个字符。而从计算机到显示器的输出传输速度要高得多，输出速度是由字符被传输到显示设备并显示出来的速度决定的，显示速度通常是每秒几千个字符。但是，这还是比每秒可执行数十亿条指令的处理器速度要慢得多。这种处理器和I/O设备之间的速度差异使得在它们之间进行数据传输需要有同步机制的支持。

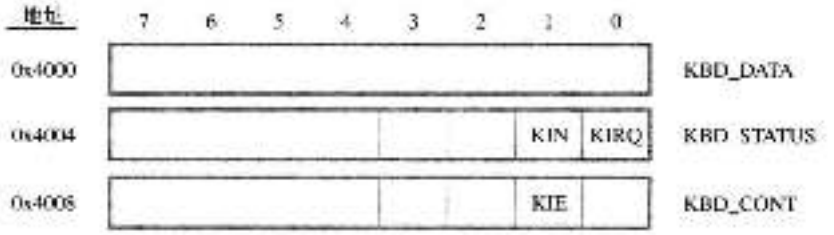
该问题的一种解决方案是利用信号协议。在输出时，处理器发送第一个字符，然后就等待来自于显示器的表示可以发送下一个字符的信号，然后处理器再发送第二个字符，依次类

准。从键盘获得输入字符使用类似的方法。处理器等待着一个信号，该信号表示一个键被按下，并且相应字符的二进制码已经保存在一个与键盘相关联的 I/O 寄存器中，然后处理器开始读取该二进制码。

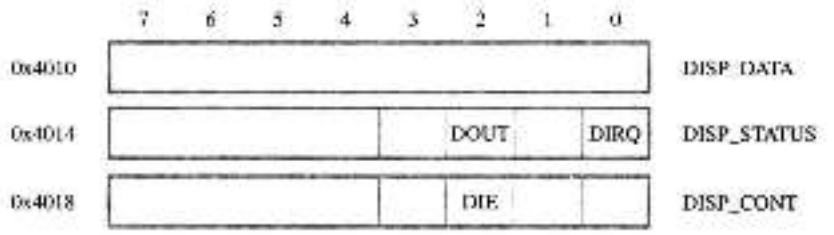
在一个键被按下的时候，键盘中的一个电路会对其进行响应，为对应字符产生计算机所使用的代码。我们假定计算机使用的是 ASCII 码（表 1-1），其中每个字符代码占用一个字节。令 KBD_DATA 表示一个用于存放生成字符的 8 位寄存器的地址标签。另外，假设有一个信号，可以通过将一个称为 KIN 的触发器置为 1 来表示一个键被按下，其中 KIN 是一个 8 位状态寄存器 KBD_STATUS 中的一部分。处理器可以读取状态标志（status flag）KIN 来确定一个字符代码是否已被放置在 KBD_DATA 内。当处理器读取状态标志以确定其状态的时候，我们说处理器正在轮询（poll）该 I/O 设备。

显示器则包含一个叫做 DISP_DATA 的 8 位寄存器，用于从处理器接收字符。而且它还必须能够表明它已准备好接收下一个字符。这可以通过一个叫做 DOUT 的状态标志来完成，DOUT 是一个状态寄存器 DISP_STATUS 中的一位。

97 / 98




a）键盘接口




b）显示器接口

图 3-3　键盘和显示器接口中的寄存器

图 3-3 说明了这些寄存器是如何组织的。每个设备的接口中还包括一个控制寄存器，我们将在 3.2 节中讨论。我们只列举了寄存器中与本章所讨论的内容有关的一些位。寄存器中的其他位可以用于其他用途，或者可能被简单地忽略掉。

如果需要像访问存储单元一样访问 I/O 接口中的寄存器，则必须给每个寄存器分配一个能被接口电路识别的特定地址。在图 3-3 中，我们分别为键盘和显示器分配了十六进制数 4000 和 4010 作为其基本地址，这两个数是数据寄存器的地址。状态寄存器的地址比数据寄存器的地址高 4 个字节，控制寄存器的地址则比数据寄存器的地址高 8 个字节。这使得一台 32 位字长的计算机中的所有地址都可以实现字对齐，在实际的应用中通常也是这么做的。以这种方式给寄存器分配地址使得在处理器执行的程序中可以访问 I/O 寄存器。这也是从程序员角度所看到的设备。

我们需要一个程序来执行下面的任务：读取键盘所产生的字符，并把这些字符存储在存储

器中，最后将它们发送到显示器上。为了执行 I/O 传输，处理器必须执行检查状态标志和在处
99 理器与 I/O 设备之间传输数据的机器指令。

我们研究一下输入过程的细节。当一个键被按下时，键盘电路把 ASCII 编码的字符放到 KBD_DATA 寄存器中。在同一时间，键盘电路将 KIN 标志设置为 1。同时处理器正在执行一个 I/O 程序，它会不断地检查 KIN 标志的状态。当它检测到 KIN 被设置为 1 时，它就把 KBD_DATA 的内容传送到一个处理器寄存器中。一旦 KBD_DATA 的内容被读出，KIN 就必须被清除为 0，这通常是由接口电路自动完成的。如果在键盘上输入第二个字符，KIN 再次被置成 1，并重复上述过程。所需的动作可通过执行以下操作来实现：

```
READWAIT   读取 KIN 标志
           当 KIN = 0 时转移到 READWAIT
           将数据从 KBD_DATA 传输到 R5
```

这些操作将字符读到处理器寄存器 R5 中。

当字符从处理器传送到显示器时，会发生类似的处理过程。当 DOUT 等于 1 时，表示显示器已经准备好接收一个字符了。在程序的控制下，处理器监控 DOUT，并且当 DOUT 等于 1 时，处理器就将一个 ASCII 编码字符传送到 DISP_DATA 中。字符被传送到 DISP_DATA 后 DOUT 被清为 0。当显示设备准备好接收第二个字符时，DOUT 再次被置成 1。这可以通过执行以下操作来实现：

```
WRITEWAIT  读取 DOUT 标志
           当 DOUT = 0 时转移到 WRITEWAIT
           将数据从 R5 传输到 DISP_DATA
```

重复执行该等待循环，直到状态标志 DOUT 被显示器置成 1，这表示显示器此时是空闲的，可以接收一个字符了。然后 R5 中的字符被传输到 DISP_DATA 中准备显示，这个操作也将把 DOUT 清除为 0。

我们假设 KIN 的初始状态是 0，DOUT 的初始状态是 1。在接通电源的时候，通常由设备控制电路来执行这个初始化操作。

在使用存储器映射 I/O 的计算机中，其中一些地址被用来表示 I/O 接口中的寄存器，这样就可以使用像 Load、Store 和 Move 这样的指令在寄存器和处理器之间传输数据。例如，用下面这条指令可以将键盘字符缓冲区 KBD_DATA 中的内容传送到处理器寄存器 R5 中：

```
LoadByte R5, KBD_DATA
```

类似地，使用下面的指令可以将寄存器 R5 中的内容传输到 DISP_DATA 中：

```
StoreByte R5, DISP_DATA
```

区别于作用在字操作数上的 Load 和 Store 操作码，LoadByte 和 StoreByte 操作码表示操作数的
100 大小是一个字节。

上面描述的读操作可以用如下 RISC 风格的指令来实现：

```
READWAIT:  LoadByte              R4, KBD_STATUS
           And                   R4, R4, #2
           Branch_if_[R4]=0      READWAIT
           LoadByte              R5, KBD_DATA
```

And 指令用来测试 KIN 标志，也就是从寄存器 KBD_STATUS 中读取到 R4 中的状态信息的位 $b_1$。只要 $b_1 = 0$，AND 操作的结果就会使得 R4 的值等于零，该 READWAIT 循环就会继续执行。

同样，写操作可以用如下指令实现：

```
WRITEWAIT: LoadByte          R4, DISP_STATUS
           And               R4, R4, #4
           Branch if [R4]=0  WRITEWAIT
           StoreByte         R5, DISP_DATA
```

我们可以看到，在这种情况下，And 指令使用立即值 4 来测试显示器的状态位 $b_2$。

### 3.1.3　一个 RISC 风格的 I/O 程序示例

如图 3-4 所示，我们现在有了一个可以执行典型的 I/O 任务的完整程序。该程序使用了上面描述的程序控制 I/O 的方法来读取、存储以及显示在键盘上键入的一行字符。当字符一个个地被读入时，它们被存储在存储器中，然后回显（echo）到显示器上。当遇到回车符 CR 时，这个程序结束。存储器中将要存储该行字符的第一个字节单元的地址是 LOC。寄存器 R2 用来指向这一部分的存储器。程序中第一条指令用地址 LOC 对其进行了初始化设置。每读取和显示一个字符，R2 就递增一次。

```
        Move              R2, #LOC      初始化指针寄存器 R2，使其指向主存
                                        中存储字符的第一个单元的地址

        MoveByte          R3, #CR       将回车符为 ASCII 码输入 R3 中
READ:   LoadByte          R4, KBD_STATUS  等待一个字符被键入
        And               R4, R4, #2    检查标志 KIN
        Branch_if_[R4]=0  READ
        LoadByte          R5, KBD_DATA  从 KBD_DATA 中读取字符（这将把 KIN
                                        清为 0）
        StoreByte         R5, (R2)      将字符写入到主存中并递增指向主存的
        Add               R2, R2, #1    指针
ECHO:   LoadByte          R4, DISP_STATUS  等待显示器准备就绪
        And               R4, R4, #4    检查 DOUT 标志
        Branch_if_[R4]=0  ECHO
        StoreByte         R5, DISP_DATA 将刚读入的字符移到显示器缓冲寄存器
                                        中（这将 DOUT 清为 0）
        Branch_if_[R5]≠[R3] READ        检查刚读入的字符是否为回车符，如果
                                        不是，则转移回去读取另一个字符
```

图 3-4　一个读取并显示一行字符的 RISC 风格的程序

### 3.1.4　一个 CISC 风格的 I/O 程序示例

我们使用 CISC 风格的指令来完成同样的任务。在 CISC 指令集中我们可以直接对存储器中的操作数执行一些算术和逻辑运算。所以可能会有如下的指令：

```
TestBit   destination, #k
```

这条指令测试目的操作数的 $b_k$ 位，如果 $b_k$=0，则将条件标志 Z（Zero）设置为 1，反之则设置为 0。因为操作数可以在一个存储单元中，所以我们可以使用指令

```
TestBit   KBD_STATUS, #1
```

来测试键盘接口中 KIN 标志的状态。然后可以使用一条能够检查 Z 标志状态的 Branch（转移）指令来跳转到等待循环的开始部分。

图 3-5 给出了一个读取并显示一行字符的 CISC 风格的程序。我们可以看到，第一条 MoveByte 指令将每个字符从 KBD_DATA 直接传输到 R2 指向的存储单元。一条 Compare（比

较）指令

Compare destination, source

将目的操作数的内容减去源操作数的内容来进行比较，然后根据比较的结果来设置状态标志。这个操作并不会改变源或者目的操作数的内容。注意图 3-5 中的 CompareByte 指令使用了自动增量的寻址方式，这种寻址方式会在比较操作结束后自动递增指针 R2 的值。而在图 3-4 中 RISC 风格的程序中，指针必须使用一条单独的 Add（加法）指令来进行递增。

```
        Move        R2, #LOC        初始化指针寄存器R2，使其指向主存
                                    中存储字符的第一个单元的地址
READ:   TestBit     KBD_STATUS, #1  等待一个字符被输入到键盘缓冲区
        Branch=0    READ            KBD_DATA中
        MoveByte    (R2), KBD_DATA  将字符从KBD_DATA传送到主存中
                                    （这将KIN清为0）
ECHO:   TestBit     DISP_STATUS, #2 等待显示器准备就绪
        Branch=0    ECHO
        MoveByte    DISP_DATA, (R2) 把输入的字符移到显示器缓冲寄存器
                                    中（这将DOUT清为0）
        CompareByte (R2)+, #CR      检查刚读入的字符是否为CR（回车
                                    符）。如果不是，则转移到去读取另一个
        Branch≠0    READ            字符。同时，递增指针以存储下一个字符
```

图 3-5 一个读取并显示一行字符的 CISC 风格的程序

我们已经讨论了在大多数计算机中所使用的存储器映射 I/O 方案。在一些处理器中还可以找到一种替代方案，这些处理器中存在特殊的 In 和 Out 指令来执行 I/O 传输。在这种情况下，存在一个只被这些指令使用的单独的 I/O 地址空间。当搭建使用这类处理器的计算机系统时，设计者可以选择将 I/O 设备连接起来以使用特殊的 I/O 地址空间，或简单地将它们整合为存储器地址空间的一部分。

101 ; 102

程序控制 I/O 方式需要处理器连续地参与到 I/O 活动中。在图 3-4 和 3-5 的程序中几乎所有的执行时间都花费在等待一个键被按下或显示器变为可用的循环中了。为了避免处理器的执行时间浪费在这些等待循环上，我们可以使用中断的概念。

## 3.2 中断

在图 3-4 和图 3-5 的例子中，当程序进入等待循环后，不停地重复测试设备状态。这期间，处理器不进行任何有效的计算。但在很多情况下，处理器在等待 I/O 设备就绪期间可以执行其他的任务。为此，我们安排 I/O 设备在准备就绪时主动通知处理器，这可以通过发送一个称为中断请求（interrupt request）的硬件信号给处理器来实现。既然处理器不再需要连续地轮询 I/O 设备的状态，那么便可以利用等待时间来执行其他有用的任务。实际上，使用中断可以很理想地消除这些等待时间。

**例 3.1**

我们来看一项任务，它需要进行大量连续的计算，并将结果在显示设备上显示出来。显示结果必须每 10 秒更新一次。10 秒的时间间隔可以由一个简单的定时器电路来确定，该电路产生一个合适的信号。处理器将定时器电路作为一个输入设备，它产生可被查询的信号。如果这通过轮询的方式进行，处理器将大量的时间浪费在检查信号的状态上。一个更好的解决方案是让定时器电路每 10 秒钟产生一个中断请求。作为响应，处理器显示最新的结果。

103

该任务可以使用一个包括两个子程序 COMPUTE 和 DISPLAY 的程序来实现。处理器连续地执行 COMPUTE 子程序。当它从定时器接收到中断请求时，将暂停 COMPUTE 子程序的执行，转去执行将最新结果发送到显示设备的 DISPLAY 子程序。DISPLAY 子程序完成后，处理器恢复 COMPUTE 子程序的执行。由于将结果发送到显示设备所需的时间与 10 秒钟的时间间隔相比是非常小的，所以实际上处理器几乎所有的时间都花费在执行 COMPUTE 子程序上。

这个例子说明了中断的概念。响应中断请求时执行的程序称为中断服务程序（interrupt-service routine），该例子中的 DISPLAY 程序便是中断服务程序。中断与子程序调用十分相似。假设在执行图 3-6 中的指令 $i$ 时有一个中断请求到达。那么处理器先执行完指令 $i$，然后将中断服务程序的第一条指令地址装入程序计数器中。这里，我们不妨假设该地址就在处理器中。执行完中断服务程序后，处理器返回到指令 $i+1$ 处。因此，当发生中断时，当前 PC 的内容指向第 $i+1$ 条指令，这个值必须暂时保存到一个已知区域中。在中断服务程序结束后，其末尾的 Return-from-interrupt（中断返回指令）将从暂时的存储区域取出存储的内容并重新装入 PC 中， 104
处理器便可从第 $i+1$ 条指令处恢复执行。返回地址（return address）必须被保存在一个指定的通用寄存器中或者处理器堆栈中。

应该注意的是，作为中断处理的一部分，处理器必须通知设备它的请求已经被识别，使得该设备撤销它的中断请求信号。这可以是一个通过互连网络发送给设备的专门的控制信号来实现，该控制信号被称为中断响应（interrupt acknowledge）信号。另一种方法是通过在处理器与 I/O 设备接口之间传输数据来达到相同的目的。在中断服务程序中，当执行访问设备接口的状态或数据寄存器的指令时，也就相当于通知该设备其中断请求已经被识别了。

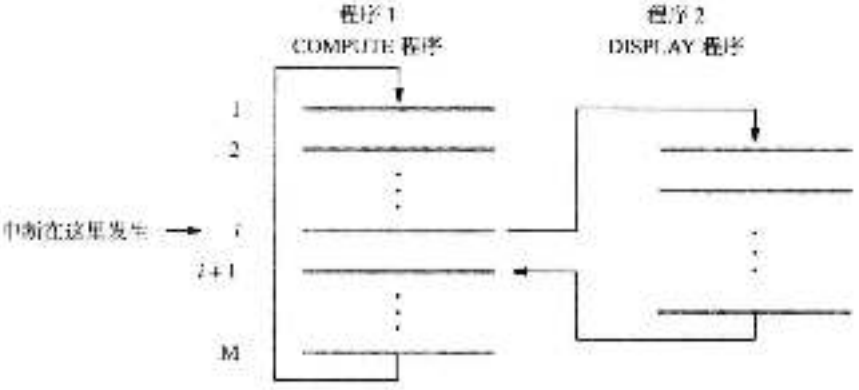



图 3-6 使用中断方式转移控制

从目前来看，对中断程序的处理和子程序十分相似。但需要注意的是二者有着重要的区别。子程序被调用时执行的是调用程序请求的功能，因此，对状态信息和寄存器内容的潜在改变是可以预测的。而中断服务程序可能与收到中断请求时正在执行的程序毫不相关。因此在开始执行中断服务程序之前，必须将该中断服务程序执行期间可能会改变的状态信息和处理器寄存器的内容都保存起来。这些保存的信息在被中断的程序重新开始执行前必须被重新装入。这样，原来的程序就可以继续执行，除了时间延迟以外不受中断的任何影响。

保存和恢复信息的任务可以由处理器自动完成，也可以由程序指令完成。大部分现代处理器只保存能保持程序完整执行的最小信息量。这是因为保存和恢复寄存器的过程中包括存储器传递，它将增加总的执行时间，因而增加执行开销。并且，保存寄存器也会增加从接收中断请求到开始执行中断服务程序之间的时间延迟。这一延迟被称为中断等待（interrupt latency）。

在一些应用中，过长的中断等待是不可接受的。所以，当收到中断请求时，应尽量减少处理器自动保存的信息量。通常，处理器只保存程序计数器和处理器状态寄存器的内容。任何需要保存的额外信息都必须由显式的指令在中断服务程序开始时保存起来，在中断服务程序结束时进行恢复。在一些早期的处理器中，尤其是在那些只有很少寄存器的处理器中，收到中断请求时所有寄存器都被处理器硬件自动保存起来。而中断返回指令的执行会将这些保存的数据恢复到各自的寄存器中。

有一些计算机提供了两种类型的中断，其中一种需要保存所有寄存器的内容，而另一种不需要。一个特定的I/O设备可以选择任何一种类型的中断，这取决于它对响应时间的要求。还有一种方式是复制处理器的寄存器组。在这种方式中，中断服务程序可以使用另一个不同的寄存器组，不需要保存和恢复寄存器。这些重复的寄存器有时候也被称为影子寄存器（shadow register）。

中断要比简单的程序控制能更好地协调I/O传输。从一般意义上说，中断能够由计算机外
105 部的事件引起控制权从一个程序到另一个程序的转移。被中断的程序在中断服务程序执行完后接着继续执行。在操作系统和很多其他控制应用中都使用了中断的概念，在这些应用中，特定程序的运行必须与外部事件严格同步，这通常称为实时处理（real-time processing）。

### 3.2.1 中断的允许与禁止

计算机所提供的设施必须使程序员能够完全控制程序执行期间发生的事件。当外部设备的中断请求到达时，处理器将中止一个程序的执行转而去启动另一个程序。由于随时都可能出现，所以中断有可能改变程序员预先设定的事件顺序。因此，必须谨慎处理程序执行中的中断。计算机的一个基本功能就是能按照需要允许和禁止中断。

在很多情况下处理器需要忽略中断请求。例如，例3.1的定时器电路，只有在COMPUTE子程序执行的时候才能提出中断请求，在其他任务正在执行的时候应当阻止其发出中断请求。在有些情况下，可能需要保证特定的指令序列没有中断地执行下去，因为中断服务程序可能会改变序列中某些指令所需要的数据。出于这些原因，我们必须提供一些允许和禁止中断的方法供程序员使用。

在处理器和I/O设备两端都可以很方便地允许和禁止中断。处理器可以接受或忽略中断请求，I/O设备也可以被允许或被禁止提出中断请求。实现这一点的一种常用机制是使用一些可以被程序指令访问的寄存器控制位。

处理器中有一个状态寄存器（status register, PS），其中包含有关其当前操作状态的信息。该寄存器的一个位IE被分配用于允许/禁止中断。然后程序员可以设置或清除IE位来执行所需的动作。当IE = 1时，I/O设备的中断请求被处理器接受并处理。当IE= 0时，处理器就简单地忽略所有I/O设备的中断请求。

I/O设备的接口包括一个控制寄存器，其中包含可以管理设备操作模式的信息。该寄存器的一个位可以专门用于中断控制。只有在该位被设置为1时I/O设备才能被允许发出中断请求。我们将在3.2.3节中讨论这个方法。

现在让我们考虑只有一台设备的单一中断请求的特殊情况。当一台设备激活中断请求信号时，它保持这个信号直到确认处理器已经接收了该中断请求。这就是说中断请求信号在中断服务程序执行期间将保持有效，也许要持续到访问该设备的指令到达。另一方面也要保证这个
106 有效的请求信号不会导致连续的中断，而使系统进入一个无法恢复的死循环。

一个不错的方法是使处理器在开始执行中断服务程序之前自动禁止中断。处理器保存程

序计数器和处理器状态寄存器的内容。将PS寄存器的内容保存后，此时IE位的1也被保存了下来，之后处理器将PS寄存器中的IE位清0，从而禁止新的中断发生。然后处理器开始执行中断服务程序。当执行中断返回指令时，PS寄存器恢复之前保存的内容，IE位被重置为1。这样，中断被重新允许。

在继续学习更复杂的中断内容前，我们总结一下处理来自单一设备的中断请求时所包含的事件序列。假设在处理器和设备端都是允许中断的，下面是一个一般的过程：

1）设备发出一个中断请求。

2）处理器中止当前正在执行的程序并保存PC与PS寄存器中的内容。

3）将PS寄存器中的IE位清零来禁止中断。

4）中断服务程序执行中断所请求的操作。在此期间，通知设备其中断请求已经被识别，然后设备撤销中断请求信号。

5）中断服务程序执行结束后，PC与PS寄存器恢复之前保存的内容（IE位随着PS的恢复也被恢复为1而允许中断），被中断的程序继续执行。

### 3.2.2 处理多台设备

现在我们考虑一个情况，处理器上连接了多台能够发出中断请求的设备。因为在操作上这些设备是相互独立的，所以没有固定的中断发生顺序。例如，正在处理设备Y引起的中断时设备X可能又发出中断请求，或者几台设备都刚好在同一时间发出中断请求。这导致了一系列的问题：

1）处理器如何确定哪个设备发出了中断请求？

2）如果不同的设备要求不同的中断服务程序，在每一种情况下处理器怎样获取适当的程序起始地址？

3）当有中断正在被处理时，另一台设备是否可以中断处理器？

4）怎样处理两个或更多个同时产生的中断请求？

不同的计算机处理这些问题的方法各不相同，采用何种方法解决上述问题对于确定计算机与特定应用是否匹配是非常重要的。

当收到一个中断请求时，需要识别是哪一台设备发出的中断请求。而且，如果有两台设备同时发出中断请求，必须选择其中一个进行处理。执行完选中设备的中断服务程序后，处理器才可以响应第二个中断请求。

读取设备的状态寄存器中的信息可以确定一个设备是否正在请求中断。当设备发出中断请求后，它的状态寄存器中的一位被置成1，这一位称为IRQ位。最简单的识别中断设备的方法就是用中断服务程序去轮询系统中所有的I/O设备。查询到的第一个IRQ位被置1的设备就是发出中断请求的设备，然后调用相应的处理子程序来提供所请求的服务。

轮询方式非常容易实现。它的主要缺点是：设备即使没有发出任何服务请求，系统也要花费时间去检查其IRQ位。另一种可供选择的方法是使用向量中断，我们将在下面进行描述。

#### 1. 向量中断

为了减少轮询过程所花费的时间，请求中断的设备可以直接向处理器标明它自己。然后，处理器就可以立即开始执行相应的中断服务程序。术语向量中断（vectored interrupt）指的就是基于这种方法的中断处理方式。

如果请求中断的设备有它自己的中断请求信号，就可以利用中断请求信号来标识自己，或者它可以通过互连网络向处理器发送一个专门的代码来标识自己。处理器的电路会确定所请

求的中断服务程序的存储器地址。一个常用的方案是在存储器中永久地分配一个区域来保存中断服务程序的地址。这些地址通常被称为中断向量（interrupt vector），而中断向量构成一个中断向量表（interrupt-vector table）。例如，可以分配128个字节来保存一个包含32个中断向量的表。通常情况下，中断向量表会保存在最低的地址范围内。中断服务程序则可能位于存储器中的任何位置。当一个中断请求到达时，发出请求的设备所提供的信息用来作为中断向量表中的一个指针，相应的中断向量地址会被自动地加载到程序计数器中。

2. 中断嵌套

在3.2.1节中我们提到在执行中断服务程序期间应该禁止中断，以保证同一设备的中断请求不会导致多次中断。这一方式还常用于多台设备的情况，在这种情况下一个特定的中断服务程序一旦开始执行，处理器就必须等到该服务程序执行完毕才能接受第二个设备的中断请求。但中断服务程序一般都很短，所导致的延迟对于大部分简单设备来说都是可以接受的。

然而，对于某些设备，响应中断请求时过长的延迟将会导致错误的操作。例如，有一台使用实时时钟记录日常时间的计算机。实时时钟在固定的时间间隔内向处理器发送中断请求。对每一个请求，处理器执行一个简短的中断服务程序来增加存储器中的一组计数器，这些计数器用秒、分等来记录时间。正确的操作要求响应实时时钟中断请求的延迟必须小于两个连续请
108 求之间的时间间隔。为了确保在存在其他中断设备的情况下满足这一条件，需要在执行其他设备的中断服务程序时接收时钟的中断请求。这就是中断嵌套。

这个例子表明I/O设备需要按照一定的优先级结构组织起来。在处理低优先级设备的中断请求时，高优先级设备的中断请求应该被接受。

多优先级结构意味着在执行一个中断服务程序期间，一些设备的中断请求可以被接受，而其他一些设备的请求不可以被接受，这取决于设备的优先级。为了实现这种方式，我们可以给处理器分配一个优先级，它可以在程序的控制下改变。处理器的优先级就是当前正在执行程序的优先级。处理器只接受优先级高于它自己的设备的中断。在某台设备的中断服务程序开始执行时，处理器的优先级自动地或者由专门的指令提升为该设备的优先级。这一操作将禁止来自同一优先级或较低优先级设备的中断，而来自更高优先级设备的中断请求将继续被接受。处理器的优先级可以编码在处理器状态寄存器的某一些位中。在某些处理器中会使用这种方法，在后面的例子中我们将使用一种更简单的方法。

最后我们需要指出，如果允许嵌套的中断，则每一个中断服务程序都必须把程序计数器和状态寄存器中的内容保存到堆栈中。这必须在中断服务程序把状态寄存器中的IE位设置为1以允许嵌套之前完成。

3. 同时请求

我们还需要考虑两台或多台设备的中断请求同时到达的问题，处理器必须能够决定哪个请求最先被服务。轮询I/O设备的状态寄存器是最简单的方法。这种方式下，优先级由轮询设备的顺序来决定。当使用向量中断时，必须保证只有一台设备被选定发送它的中断向量代码。这是在硬件中通过使用仲裁电路来完成的，我们将在第7章中介绍仲裁电路。

### 3.2.3 控制I/O设备行为

确保中断请求只来自那些处理器愿意处理的I/O设备是非常重要的。因此，我们需要在每台设备的接口电路中建立一种机制，来控制该设备是否允许中断处理器。这一控制在设备的接口电路中通常以中断允许位（interrupt-enable，IE）的形式提供。

I/O设备的复杂性各不相同，从简单到复杂。简单的设备，如键盘，只需要很少的控制。

复杂的设备可能有很多种可能的操作模式需要进行控制。一种常用的方法是在设备接口中提供一个控制寄存器，其中包含控制设备行为所需的信息。就像我们之前讨论过的数据寄存器和状态寄存器一样，该寄存器也作为一个可寻址的单元进行访问。寄存器中的一位作为中断允许位 IE。当中断允许位被一条将新信息写到控制寄存器的指令置为 1 时，只要该设备已为 I/O 传输做好准备它就可以随时中断处理器。 109

图 3-3 展示了在键盘和显示设备的接口中使用的寄存器。由于这些设备传输基于字符的数据，每次只处理一个字符，所以在这里使用 8 位的数据寄存器是合适的。我们假定状态寄存器和控制寄存器的长度也是 8 位的。这些寄存器中只需要一位或两位来处理 I/O 传输。剩余的位可以用来指定设备操作的其他方面，如果不需要的话就可以忽略掉。键盘状态寄存器包括 KIN 位和 KIRQ 位。我们已经在 3.1.2 节中讨论了 KIN 位的用途。如果中断请求已经发出但尚未被处理，则 KIRQ 位被置为 1。只有当其控制寄存器中的中断允许位 KIE 被置为 1 时，键盘才可以发出中断请求。因此，当 KIE 位与 KIN 位都等于 1 时，键盘发出中断请求，并且 KIRQ 位被置为 1。同样，显示器接口的状态寄存器中的 DIRQ 位可以表示显示器是否已经发出中断请求。该接口的控制寄存器中的 DIE 位用来表示是否允许中断。可以观察到我们已经把 KIN 位和 KIE 位放置在位位置 1 上，把 DOUT 和 DIE 位放置在位位置 2 上。这种设置不是绝对的，只是为了使接下来的程序实例更容易理解。

### 3.2.4　处理器控制寄存器

我们已经讨论了处理器中设置状态寄存器的必要性。为了处理中断，引入一些其他的控制寄存器是非常有用的。图 3-7 描述了一种可能的方案，其中有 4 个处理器控制寄存器。状态寄存器 PS 中包含中断允许位 IE，除此之外还有一些其他的状态信息。只有当 IE 位被置为 1 时，处理器才能接受中断。当收到一个中断请求并接受该请求时，IPS 寄存器将自动保存 PS 的内容。在中断服务程序结束时，通过把 IPS 的内容传输到 PS 中，处理器就会自动恢复为以前的状态。由于只有一个寄存器被用于存储以前的状态信息，所以，如果允许中断嵌套的话，那么在堆栈中保存 IPS 的内容将会变得很有必要。 110

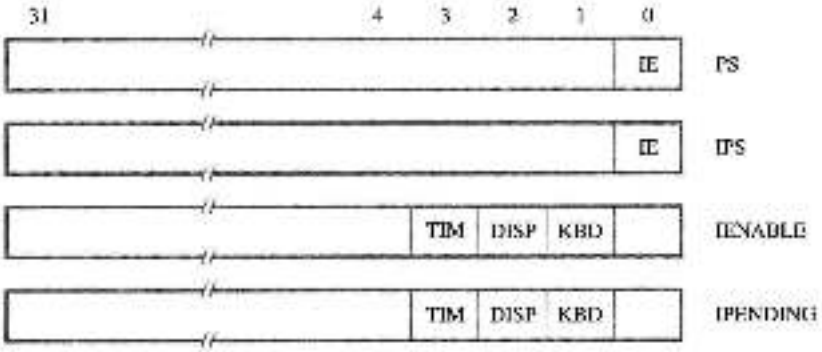



图 3-7　处理器中的控制寄存器

IENABLE 寄存器可以使处理器有选择地响应每个 I/O 设备。可以在 IENABLE 寄存器中为每个设备分配一位，如图中所示的为键盘、显示器以及我们将在后面的例子中使用的定时器电路各分配了一位。当某位被置为 1 时，处理器将接受所对应设备的中断请求。IPENDING 寄存器指示活跃的中断请求。当多个设备同时发出请求时这将是非常方便的。然后，程序可以决定哪个中断应该首先被服务。

在一个32位的处理器中，控制寄存器的长度为32位。使用图3-7中的结构，将可以用一种直接的方式来表示32个I/O设备。

汇编语言指令可以用图3-7中那样的名字来使用处理器控制寄存器。但是不能使用访问通用寄存器的方式来访问这些寄存器。它们不能被算术和逻辑指令访问，也不能被使用图2-32c所示编码格式的Load和Store指令访问，因为在这些指令中用一个5位的字段来指定源寄存器或目的寄存器，这使得它只能够指定32个通用寄存器。可以提供专门的指令或专门的寻址方式来访问处理器控制寄存器。在RISC风格的处理器中，专门的指令可以是如下类型的

MoveControl R2, PS

这个指令把程序状态寄存器的内容装入寄存器R2中，而指令

MoveControl IENABLE, R3

则把R3的内容放入IENABLE寄存器中。这些指令完成控制寄存器与通用寄存器之间的数据传输操作。

### 3.2.5 中断程序示例

我们已经描述了中断的基本知识，现在给出一些说明性的例子。我们将使用具有图3-3所示寄存器结构的键盘和显示设备来展示这些例子。

**例3.2**

我们再来看一下这个任务：读取键盘上键入的一行字符，并把字符存储在主存中，然后在显示设备上显示这些字符。在图3-4和3-5中，我们使用轮询法来检测I/O设备是否做好数据传输的准备，以此来展示这个任务是如何执行的。现在，我们将对键盘使用中断方式，而对显示器使用轮询方式。

111

现在假设一个特定的存储单元ILOC专门用于处理中断，并假设它包含了中断服务程序的第一条指令。每当一个中断请求到达处理器，并且处理器中断被允许时，处理器将自动完成以下工作：

- 将程序计数器的内容存储在用于保存返回地址的处理器寄存器中或者存储在处理器堆栈中。
- 将状态寄存器PS的内容转移到IPS寄存器中保存起来，并清除PS中的IE位。
- 将地址ILOC装入程序计数器中。

假设我们希望在主程序中从键盘读取一行字符，并存储到存储器中从LINE单元开始的连续字节区域中。另外，假设中断服务程序已经装入存储器中从ILOC单元开始的区域中。则主程序必须按如下步骤来初始化中断过程：

1）将地址LINE装入存储单元PNTR中。中断服务程序把该单元作为一个指针来将输入字符存储到存储器中。

2）通过将KBD_CONT寄存器中的KIE位置1，以允许键盘中断。

3）通过将控制寄存器IENABLE中的KBD位置1，以使处理器能够接受键盘中断。

4）通过将处理器状态寄存器PS中的IE位置1，以使处理器能够响应中断。

该初始化过程完成后，从键盘输入一个字符就会导致键盘接口产生一个中断请求。此时正在执行的程序将被中断，转去执行键盘输入中断服务程序。该中断服务程序必须执行以下任务：

1）从键盘的输入数据寄存器中读取输入字符。然后，键盘接口电路撤销中断请求。

2）将字符存储到PNTR指向的存储单元中，并递增PNTR。

3）使用轮询方式显示该字符。

4）当达到行尾时，禁止键盘中断并通知主程序。

5）从中断返回。

执行这些任务的 RISC 风格的程序如图 3-8 所示。程序中的注释已经解释了相关的细节。当检测到输入行的末端时，中断服务程序将 KBD_CONT 寄存器中的 KIE 位清除，表示不期望更多的输入了。同时也将变量 EOL（End Of Line）置为 1，它在初始时被清为 0。假设主程序用周期性的查询来确定输入行是否准备好进行处理。EOL 变量提供了一种在主程序和中断服务程序之间发送信号的方法。 [112]

```
中断服务程序
ILOC:   Subtract        SP, SP, #8          保存寄存器
        Store           R2, 4(SP)
        Store           R3, (SP)
        Load            R2, PNTR            载入地址指针
        LoadByte        R3, KBD_DATA        从键盘读入字符
        StoreByte       R3, (R2)            将字符写入存储器
        Add             R2, R2, #1          递增指针
        Store           R2, PNTR            更新存储器中的指针
ECHO:   LoadByte        R2, DISP_STATUS     等待显示器准备就绪
        And             R2, R2, #4
        Branch_if_[R2]=0    ECHO
        StoreByte       R3, DISP_DATA       显示刚读入的字符
        Move            R2, #CR             回车符的 ASCII 码
        Branch_if_[R3]≠[R2] RTRN            如果不是 CR，则返回
        Move            R2, #1
        Store           R2, EOL             两个行的结束
        Clear           R2                  禁止键盘中断
        StoreByte       R2, KBD_CONT
RTRN:   Load            R3, (SP)            恢复寄存器
        Load            R2, 4(SP)
        Add             SP, SP, #8
        Return-from-interrupt

主程序
START:  Move            R2, #LINE
        Store           R2, PNTR            初始化缓冲区指针
        Clear           R2
        Store           R2, EOL             清除行结束指示变量
        Move            R2, #2              允许键盘中断
        StoreByte       R2, KBD_CONT
        MoveControl     R2, IENABLE
        Or              R2, R2, #2          在处理器控制寄存器中允许
        MoveControl     IENABLE, R2         键盘中断
        MoveControl     R2, PS
        Or              R2, R2, #1
        MoveControl     PS, R2              设置 PS 中的中断允许位
        next instruction
```

图 3-8　一个使用中断方式读取一行字符，并使用轮询方式显示该行字符的 RISC 风格的程序 [113]

观察可知主程序的最后二条指令被用来将 PS 中的中断允许位置 1。由于只有 MoveControl 指令可以访问控制寄存器的内容，所以 PS 的内容被装入通用寄存器 R2 中，修改后又写回到 PS 中。使用 Or 指令修改内容只会影响 IE 位，而 PS 中的其余位则保持不变。

当多个 I/O 设备提出中断请求时，需要确定是哪个设备请求了中断。这可以通过使用软件去检查 IPENDING 控制寄存器中的信息，并选择应执行的中断服务程序来实现。

**例 3.3**

在例 3.2 中，我们只对键盘使用了中断方式，但显示设备也可以使用中断方式。假设一个程序需要显示存储在存储器中的一页文本。这可以通过如下的方式来完成：当显示器接口通过

一个中断请求来表示它已准备就绪时处理器就发送一个字符。假设该程序使用显示器和键盘，并且允许两者发出中断请求。使用图 3-3 和 3-7 中的寄存器结构、中断的初始化和中断请求的处理都可以用图 3-9 所示的方法来完成。

```
中断处理程序
ILOC:       Subtract        SP, SP, #12         保存寄存器
            Store           LINK_reg, 8(SP)
            Store           R2, 4(SP)
            Store           R3, (SP)
            MoveControl     R2, IPENDING        检查 IPENDING 的内容
            And             R3, R2, #4          检查显示器是否发出请求
            Branch_if_[R3]=0 TESTKBD            如果没有，则检查键盘是否发出请求
            Call            DISR                调用显示器中断服务程序（ISR）
TESTKBD:    And             R3, R2, #2          检查键盘是否发出请求
            Branch_if_[R3]=0 NEXT               如果没有，则检查下一个设备
            Call            KISR                调用键盘中断服务程序（ISR）
NEXT:       . . .                               检查其他中断
            Load            R3, (SP)            恢复寄存器
            Load            R2, 4(SP)
            Load            LINK_reg, 8(SP)
            Add             SP, SP, #12
            Return-from-interrupt
主程序
START:      . . .                               为 ISR 建立参数表
            Move            R2, #2              允许键盘中断
            StoreByte       R2, KBD_CONT        允许显示器中断
            Move            R2, #4
            StoreByte       R2, DISP_CONT
            MoveControl     R2, IENABLE
            Or              R2, R2, #6          在处理器控制寄存器中允许中断
            MoveControl     IENABLE, R2
            MoveControl     R2, PS
            Or              R2, R2, #1
            MoveControl     PS, R2              设置 PS 中的中断允许位
            下一条指令
键盘中断服务程序
KISR:       . . .
            .
            Return
显示器中断服务程序
DISR:       . . .
            .
            Return
```

图 3-9 一个初始化并处理中断的 RISC 风格的程序

主程序必须初始化中断服务程序所需的任何变量，如存储器缓冲区指针。然后主程序允许键盘和显示器中断。接下来主程序在处理器控制寄存器 IENABLE 中允许中断。注意，装入这个寄存器的立即值 6 将 KBD 和 DISP 位都置为 1。最后通过将处理器状态寄存器 PS 中的 IE 位置为 1，使处理器能够响应中断。

我们再假设当一个中断请求到达时，处理器将自动保存程序计数器（PC）的内容，然后将地址 ILOC 装入 PC 中。它同时也将状态寄存器（PS）中的内容转移到 IPS 寄存器中保存起来，并禁止中断。与只有一个设备能够提出中断请求的例 3.2 不同，现在我们不能直接进入所需的中断服务程序。首先，需要确定发出中断的设备。我们可以在处理器控制寄存器

IPENDING 中找到所需要的信息。由于中断服务程序在这个过程中使用寄存器 R2 和 R3，所以这两个寄存器的内容必须被保存在堆栈中以便以后能够恢复。子程序链接寄存器 LINK_reg 的内容也需要保存，因为当某个子程序正在执行时可能会发生一个中断，而该中断的中断服务程序可能调用另一个子程序。检测中断的电路将 IPENDING 中与每一个待处理的请求相对应的位置为 1。在图 3-9 中，IPENDING 的内容被装入通用寄存器 R2 中，然后再检查它来确定哪些中断正在等待处理。如果显示器有一个待处理的中断，那么它的中断服务程序将被执行。如果没有，则再检查键盘。接下来可能还将检查任何可能有待处理请求的其他设备。在同时请求的情况下，IPENDING 中的位的检查顺序，确立了中断设备的优先级。

114
:
115

处理中断请求并为提出请求的设备提供相应服务的程序通常称为中断处理程序（interrupt handler）。注意，当中断处理程序在一个固定的地址 ILOC 处开始时，单独的中断服务程序就只是可以放在存储器中任意位置的子程序而已。

在图 3-9 中，我们使用软件的方法来确定发出中断请求的设备。在使用向量中断的处理器中，检测中断请求的电路会为每一个在中断向量表中分配了具体位置的中断自动加载一个不同的地址到程序计数器中。对于每个待处理的请求，会有一个单独的中断服务程序一直执行到结束，即使有多个中断请求在同一时间提出。

**CISC 风格的中断示例**

使用与前面类似方法的 CISC 风格的指令也可以实现上述任务。主要区别在于某些操作，如测试 I/O 寄存器中的某个位，可以直接执行。例 3.2 和例 3.3 中的任务可分别用图 3-10 和图 3-11 中的程序实现。TestBit 指令用来测试状态标志。SetBit 和 ClearBit 指令分别用来将 I/O 寄存器中的某位置为 1 或 0。程序中的注释解释了如何实现所需的任务。

计算机系统中所包含的输入 / 输出操作通常要比这些简单例子中的多得多。正如我们将在第 4 章中描述的，计算机操作系统将代表用户程序执行这些操作。在第 7 章中，我们将详细讨论 I/O 操作中所使用的硬件。

```
中断服务程序
ILOC:   Move        -(SP), R2       保存寄存器
        Move        R2, PNTR        装入地址指针
        MoveByte    (R2), KBD_DATA  将字符写入存储器并递增指针
        Add         PNTR, #1
ECHO:   TestBit     DISP_STATUS, #2 等待显示器准备就绪
        Branch=0    ECHO
        MoveByte    DISP_DATA, (R2) 显示刚读入的字符
        CompareByte (R2), #CR       检查刚读入的字符是否为回车符（CR）
        Branch≠0    RTRN            如果不是 CR 则返回
        Move        EOL, #1         指示行的结束
        ClearBit    KBD_CONT, #1    禁止键盘中断
RTRN:   Move        R2, (SP)+       恢复寄存器
        Return-from-interrupt

主程序
START:  Move        PNTR, #LINE     初始化缓冲区指针
        Clear       EOL             清除行结束指示变量
        SetBit      KBD_CONT, #1    允许键盘中断
        Move        R2, #2          在处理器控制寄存器中允许键盘中断
        MoveControl IENABLE, R2
        MoveControl R2, PS
        Or          R2, #1
        MoveControl PS, R2          设置 PS 中的中断允许位
        下一条指令
```

图 3-10　使用中断方式读取一行字符并使用轮询方式显示该行字符的 CISC 风格的程序

```
中断处理程序
ILOC:       Move          -(SP), R2          保存寄存器
            Move          -(SP), LINK_reg
            MoveControl   R2, IPENDING       检查 IPENDING 的内容
            TestBit       R2, #2             检查显示器是否发出请求
            Branch=0      TESTKBD            如果没有，则检查键盘是否发出请求
            Call          DISR               调用显示器的中断服务程序（ISR）
TESTKBD:    TestBit       R2, #1             检查键盘是否发出请求
            Branch=0      NEXT               如果没有，则检查下一个设备
            Call          KISR               调用键盘的中断服务程序（ISR）
NEXT:       . . .                            检查其他中断

            Move          LINK_reg, (SP)+    恢复寄存器
            Move          R2, (SP)+
            Return-from-interrupt

主程序
START:      . . .                            为 ISR 建立参数表
            SetBit        KBD_CONT, #1       允许键盘中断
            SetBit        DISP_CONT, #2      允许显示器中断
            MoveControl   R2, IENABLE
            Or            R2, #6             在处理器控制寄存器中允许中断
            MoveControl   IENABLE, R2
            MoveControl   R2, PS
            Or            R2, #1
            MoveControl   PS, R2             设置 PS 中的中断允许位
            下一条指令
键盘中断服务程序
KISR:       . . .
            :
            Return
显示器中断服务程序
DISR:       . . .
            :
            Return
```

图 3-11 初始化并处理中断的 CISC 风格的程序

### 3.2.6 异常

中断是导致一个程序的执行被中止、另一个程序的执行开始的事件。到目前为止，我们处理的中断只是由那些与 I/O 数据传输有关的事件引起的。然而，中断机制还可以用在许多其他的情形下。

术语异常（exception）通常用来指任何引起中断的事件。因此，I/O 中断是异常的一种。现在我们来讲述几种其他类型的异常。

#### 1. 错误恢复

计算机使用大量的技术来确保所有硬件都能正确工作。例如，许多计算机在主存中都包含有错误校验码，它可以对存储的数据进行错误检测。如果出现了错误，控制硬件就会检测到并产生中断通知处理器。

当处理器正在执行某个程序的指令时，如果检测到错误或不正常的情况，它也会中断该

程序。例如，一条指令的OP码字段有可能不对应任何合法指令，或一条算术指令试图除以0。

当这些错误引发异常时，处理器按照与处理I/O中断请求相同的方式进行处理，中止正在执行的程序并启动异常服务程序，该异常服务程序进行适当的操作，尽可能从错误中恢复过来，或将该错误通知给用户。回想一下I/O中断时的情形，我们假设处理器在接受中断前必须完成正在执行的指令。然而，当中断是由与当前指令有关的错误引起的时候，通常不等那条指令执行完，处理器就立即开始异常处理。

2. 调试

另一种非常重要的异常是用于帮助调试程序的。系统软件通常包括一个调试器（debugger）程序，它用来帮助程序员找出程序中的错误。调试器使用异常来提供两项重要的功能：跟踪模式和断点。这些功能将在第4章中详细描述。 116 | 118

3. 异常在操作系统中的使用

操作系统（OS）软件的职责是协调计算机内部的活动。它使用异常来与用户程序通信并控制用户程序的执行，使用硬件中断来执行I/O操作。这部分内容将在第4章中进行讨论。

## 3.3 结束语

在这一章中，我们讨论了两种基本的I/O传输方式。最简单的技术是程序控制I/O，在这种技术中，处理器在程序指令的直接控制下执行所有必要的功能。第二种方式是基于中断的，这种机制可以中断程序的正常执行，转去服务需要迫切关注的、具有更高优先级的请求。虽然所有的计算机都有处理这种情况的机制，但是不同计算机的中断处理方案的复杂程度各不相同。

我们是从程序员的角度来处理I/O问题的。在第7章中，我们将介绍硬件方面的内容以及一些常用的I/O标准。

## 3.4 问题解析

本节将介绍一些可能要求学生解决的典型问题实例，并分析说明如何解决这样的问题。

**例3.4**

**问题**：假定一个存储单元BINARY包含一个32位的模式，要求在具有图3-3所示接口的显示设备上把这些位显示为8个十六进制数字。写一个程序完成这个任务。

**解答**：首先，需要把32位的模式转换成可表示为ASCII编码字符的十六进制数字。一个简单的转换方法是使用查表（table-lookup）法。建立一个包含16个表项的表，为每一个可能的16进制数字提供ASCII码。然后，我们将BINARY中的模式分成4位一组的段，可以在表中查出相应的字符，并将其存储在从HEX单元开始的存储器字节块中。最后，将从HEX开始的8个字符发送给显示器。

图3-12和图3-13分别给出了完成所需任务的RISC风格和CISC风格的程序。程序中的注释详细描述了每行代码的行为。 119

**例3.5**

**问题**：思考例3.1中描述的任务。假设定时器电路包括一个32位的可上/向下计数器，它由100MHz的时钟驱动。该计数器可以设定为从一个特定的初始计数值开始计数。定时器的I/O接口如图3-14所示，其中包括4个寄存器：

- TIM_STATUS指示定时器当前的状态，其中：
  - 当计数器运行时TON位被置为1。
  - 当计数器达到零时ZERO位被置为1。
  - 当计数器内容达到零而且定时器中断被允许时，定时器发出中断请求，此时，TIRQ位被置为1。

```
        Load        R2, BINARY      载入二进制数
        Move        R3, #8          R3 是一个被设置为 8 的数字计数器
        Move        R4, #HEX        R4 指向十六进制数字
LOOP:   RotateL     R2, R2, #4      把高位数字循环移位到低位

        And         R5, R2, #0xF    提取下一个数字
        LoadByte    R6, TABLE(R5)   获取数字的 ASCII 码
        StoreByte   R6, (R4)        将其存储在 HEX 单元中
        Subtract    R3, R3, #1      递减数字计数器
        Add         R4, R4, #1      递增指向十六进制数字的指针
        Branch_if_[R3]>0 LOOP
DISPLAY: Move       R3, #8          如果不是最后一位数字则返回循环
        Move        R4, #HEX        等待显示器准备就绪
DLOOP:  LoadByte    R5, DISP_STATUS 检查 DOUT 标识
        And         R5, R5, #4
        Branch_if_[R5]=0 DLOOP
        LoadByte    R6, (R4)        获取下一个 ASCII 字符
        StoreByte   R6, DISP_DATA   将其发送到显示器上
        Subtract    R3, R3, #1      递减计数器
        Add         R4, R4, #1      递增了指针
        Branch_if_[R3]>0 DLOOP      循环，直到所有字符都显示完毕
        下一条指令

        ORIGIN      1000
HEX:    RESERVE     8               ASCII 编码数字的存储空间
TABLE:  DATABYTE    0x30,0x31,0x32,0x33 ASCII 码转换表
        DATABYTE    0x34,0x35,0x36,0x37
        DATABYTE    0x38,0x39,0x41,0x42
        DATABYTE    0x43,0x44,0x45,0x46
```

图 3-12 例 3.4 的 RISC 风格的程序

```
        Move        R2, BINARY      载入二进制数
        Move        R3, #8          R3 是一个被设置为 8 的数字计数器
        Move        R4, #HEX        R4 指向十六进制数字
LOOP:   RotateL     R2, #4          把高位数字循环移位到低位

        Move        R5, R2
        And         R5, #0xF        提取下一个数字
        MoveByte    (R4)+, TABLE(R5) 获取数字的 ASCII 码并将其存储在
                                    HEX 单元中
        Subtract    R3, #1          递减数字计数器
        Branch>0    LOOP            如果不是最后一位数字则返回循环
DISPLAY: Move       R3, #8
        Move        R4, #HEX
DLOOP:  TestBit     DISP_STATUS, #2 等待显示器准备就绪
        Branch=0    DLOOP
        MoveByte    DISP_DATA, (R4)+ 将下一个字符发送到显示器
        Subtract    R3, #1          递减计数器
        Branch>0    DLOOP           循环，直到所有字符都显示完毕
        下一条指令

        ORIGIN      1000
HEX:    RESERVE     8               ASCII 编码数字的存储空间
TABLE:  DATABYTE    0x30,0x31,0x32,0x33 ASCII 码转换表
        DATABYTE    0x34,0x35,0x36,0x37
        DATABYTE    0x38,0x39,0x41,0x42
        DATABYTE    0x43,0x44,0x45,0x46
```

图 3-13 例 3.4 的 CISC 风格的程序

读取这个状态寄存器的动作会自动将 ZERO 和 TIRQ 位清 0。

- TIM_CONT 控制操作模式，其中：
  - UP 位被置为 1 时，计数器递增计数；当 UP 位被清为 0 时，计数器递减计数。
  - FREE 位被置为 1 时，计数器进入持续运行模式，此时，当实际计数达到 0 时会自动将初始计数值重新装入计数器中。
  - RUN 位被置为 1 时，计数器开始计数；RUN 位被清 0 时计数器停止计数。
  - TIE 位被置为 1 时，允许定时器中断。
- TIM_INIT 保存初始计数值。
- TIM_COUNT 保存当前的计数值。

写一个程序来实现所需的任务，使用图 3-7 中所描述的处理器控制寄存器。




图 3-14　定时器接口中的寄存器

**解答：**为了每 10 秒钟能获得一个中断请求，需要计数 $10^8$ 个时钟周期。可以这样实现：将这个值写入 TIM_INIT 寄存器，然后递减计数器的计数，当计数达到 0 时发出中断请求。$10^8$ 这个值可以用十六进制数 3B9ACA00 来表示。为了实现所需的操作，FREE、RUN 和 TIE 位必须被置为 1，而 UP 位则必须被置为 0。

使用图 3-9 中概述的方法，我们可以用图 3-15 所示的 RISC 风格的程序来实现所需的任务。注意，初始计数值是一个 32 位的立即值，可以用 2.9 节中给出的方法将其装入 R2 中。

图 3-16 给出了用图 3-11 概述的方法实现所需任务的 CISC 风格的程序。在这里，32 位的立即数操作数可以在一条单一的指令中指定。

**例 3.6**

**问题：**数字系统中有一种常用的输出设备七段显示器，如图 3-17 所示。该设备包括七个独立的数码管段，给它们加上电信号的时候它们可以被点亮。假设每一段在施加逻辑值 1 的时候被点亮。图中给出了显示数字 0 到 9 所需的位模式。

写一个程序，显示用 ASCII 编码字符表示的数。该 ASCII 编码字符存储在地址为 0x800 的存储单元 DIGIT 中。假设显示器有一个 I/O 接口，它包含一个 8 位的数据寄存器 SEVEN。其中 $a$ 段至 $g$ 段分别连接到 SEVEN 的第 6 至第 0 位上。设 SEVEN 的第 7 位等于 0。另外，假设寄存器 SEVEN 的地址为 0x4030。如果 DIGIT 单元中的 ASCII 码表示的字符不是 0 到 9 范围内的数，则显示器是空白的，所有的段都被关闭。

**解答：**可以使用一个查找表 TABLE 来保存与数字 0 到 9 相对应的七段位模式。通过使用 AND（"与"）操作，可以将 ASCII 编码的数字转换为可作为表索引的 4 位数。此外，需要检查 ASCII 码的高 4 位是否是 0011。注意，DIGIT、SEVEN 和 TABLE 这三个地址可以用 16 位来表示。

图 3-18 和图 3-19 分别给出了可能的 RISC 风格和 CISC 风格的程序。

```
中断处理程序
ILOC:   Subtract    SP, SP, #8          保存寄存器
        Store       LINK_reg, 4(SP)
        Store       R2, (SP)
        MoveControl R2, IPENDING        检查 IPENDING 的内容
        And         R2, R2, #8          检查是否是定时器的请求
        Branch_if_[R2]=0 NEXT
        LoadByte    R2, TIM_STATUS      清除 TIRQ 和 ZERO 位
        Call        DISPLAY             调用 DISPLAY 程序
NEXT:   · · ·                           检查其他中断

        Load        R2, (SP)            恢复寄存器
        Load        LINK_reg, 4(SP)
        Add         SP, SP, #8
        Return from interrupt

主程序
START:  · · ·                           为 ISR 建立参数表
        OrHigh      R2, R0, #0x3B9A     带各初始计数值
        Or          R2, R2, #0xCA00
        Store       R2, TIM_INIT        设置初始计数值
        Move        R2, #7              将定时器设置为持续运行
        StoreByte   R2, TIM_CONT        模式，并允许中断⊖
        MoveControl R2, IENABLE
        Or          R2, R2, #8          在处理器控制寄存器中允
        MoveControl IENABLE, R2         许定时器中断
        MoveControl R2, PS
        Or          R2, R2, #1          设置 PS 中的中断允许位
        MoveControl PS, R2
COMPUTE: 下一条指令
```

图 3-15 例 3.5 的 RISC 风格的程序

```
中断处理程序
ILOC:   Move        -(SP), R2           保存寄存器
        Move        -(SP), LINK_reg
        MoveControl R2, IPENDING        检查 IPENDING 的内容
        TestBit     R2, #3              检查是否是定时器的请求
        Branch=0    NEXT
        MoveByte    R2, TIM_STATUS      清除 TIRQ 和 ZERO 位
        Call        DISPLAY             调用 DISPLAY 程序
NEXT:   · · ·                           检查其他中断

        Move        LINK_reg, (SP)+     恢复寄存器
        Move        R2, (SP)+
        Return-from-interrupt

主程序
START:  · · ·                           为 ISR 建立参数表
        Move        TIM_INIT, #0x3B9ACA00   设置初始计数值
        MoveByte    TIM_CON, #7         将定时器设置为持续运行模
                                        式，并允许中断⊖
        MoveControl R2, IENABLE
        Or          R2, #8
        MoveControl IENABLE, R2         在处理器控制寄存器中允许
        MoveControl R2, PS              定时器中断
        Or          R2, #1
        MoveControl PS, R2              设置 PS 中的中断允许位
COMPUTE: 下一条指令
```

图 3-16 例 3.5 的 CISC 风格的程序

⊖ 即将 TIM_CONT 寄存器的 FREE、RUN 和 TIE 位都置为 1。——译者注




| 数字 | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 3 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 6 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

图 3-17 七段显示器

```
DIGIT   EQU             0x800           ASCII 编码数字的位置
SEVEN   EQU             0x4030          七段显示器的地址
        LoadByte        R2, DIGIT       载入 ASCII 编码的数字
        And             R3, R2, #0xF0   提取 ASCII 码的高位
        And             R2, R2, #0x0F   提取十进制数
        Move            R4, #0x30       检查 ASCII 码的高位是否是
        Branch_if_[R3]=[R4] HIGH3       0011
        Move            R2, #0x0F       如果不是数字，显示空白
HIGH3:  LoadByte        R5, TABLE(R2)   获得七段模式
        StoreByte       R5, SEVEN       显示数字

        ORIGIN          0x1000
TABLE:  DATABYTE        0x7E,0x30,0x6D,0x79     包含七段模式的表
        DATABYTE        0x33,0x5B,0x5F,0x70
        DATABYTE        0x7F,0x7B,0x00,0x00
        DATABYTE        0x00,0x00,0x00,0x00
```

图 3-18 例 3.6 的 RISC 风格的程序

```
DIGIT   EQU             0x800           ASCII 码数字的位置
SEVEN   EQU             0x4030          七段显示器的地址
        Move            R2, DIGIT       载入 ASCII 码的数字
        Move            R3, R2
        And             R3, #0xF0       提取 ASCII 码的高位
        And             R2, #0x0F       提取十进制数
        CompareByte     R3, #0x30       检查 ASCII 码的高位是否是 0011
        Branch=0        HIGH3
        Move            R2, #0x0F       如果不是数字，显示空白
HIGH3:  MoveByte        SEVEN, TABLE(R2) 显示数字

        ORIGIN          0x1000
TABLE:  DATABYTE        0x7E,0x30,0x6D,0x79     包含七段模式的表
        DATABYTE        0x33,0x5B,0x5F,0x70
        DATABYTE        0x7F,0x7B,0x00,0x00
        DATABYTE        0x00,0x00,0x00,0x00
```

图 3-19 例 3.6 的 CISC 风格的程序

123
|
125

## 习题

[E] 3.1 一旦输入数据寄存器被读取，接口电路中的输入状态位就会被清零。为什么要这么做？

[E] 3.2 写一个程序，在显示设备的一行上以 16 进制形式显示主存中 10 个字节区内容。这 10 个字节从存储器的 LOC 单元开始。每个字节含有两个 16 进制字符。显示时，连续的字节用空格隔开。

[E] 3.3 子程序与中断服务程序的区别是什么？

[E] 3.4 在图 3-4 中的第一条 And 指令中，在检查 KIN 标识的时候使用了立即值 2，但在图 3-5 中的第一条 TestBit 指令中，检查 KIN 标识时却使用了立即值 1。解释两者的区别。

[D] 3.5 有一台计算机，要求它从 20 台终端的键盘输入端接收字符。指针 PNTRn 指向用于存储每一台终端的数据的主存区域，n 从 1 到 20。当另一个程序 PROG 正在执行时，终端的输入数据必须被收集起来。这可以由如下两种方法实现：

（a）每 T 秒钟程序 PROG 调用一个轮询子程序 POOL。这个子程序按顺序查询所有 20 台终端的状态并将所有输入字符传送到存储器中，然后返回到 PROG。

（b）当任何终端的接口缓冲区中有就绪的字符时，产生一个中断请求，然后中断服务程序 INTERRUPT 被执行。INTERRUPT 查询状态寄存器以找出第一个就绪的字符，传送该字符，然后返回到 PROG。

编写程序 POOL 和 INTERRUPT。假设所有终端的最大字符输入速率为每秒 c 个字符，平均速率等于 rc，其中 $r \leq 1$。在方法（a）中，能够保证不丢失输入字符的 T 的最大值是多少？在方法（b）中，等价的 T 值又是多少？估计一下，在 c=100，r 分别为 0.01、0.1、0.5 和 1 时，方法（a）和（b）中服务终端的平均时间百分比是多少。假设 POLL 花费 800 纳秒去查询所有 20 台设备，处理设备的中断需要 200 纳秒。

[E] 3.6 在图 3-9 中，主程序的 START 段最后才设置 PS 中的中断允许位，这是为什么？该顺序对 START 中之前的操作是否有影响？为什么？

[E] 3.7 即使有多个中断请求等待处理，图 3.9 中每次进入 ILOC 时仍只能处理一个请求。以上说法是正确还是错误？说明理由。

[E] 3.8 一个用户程序可以在每次除法操作前立刻检查除数是否为 0，然后采取适当的动作而不去调用操作系统。给出理由说明为什么在用户程序中除数为 0 的实际情况下，允许产生一个异常中断是/不是最好的。

[126]

[M] 3.9 假设一个存储单元 BINARY 包含一个 16 位的模式。要求在具有图 3-3 所示接口的显示设备上把这些位显示为 0 和 1 组成的字符串。写一个 RISC 风格的程序完成这个任务。

[M] 3.10 写一个 CISC 风格的程序完成习题 3.9 中的任务。

[E] 3.11 如果 TABLE 的地址是 0x10100，修改图 3-18 中的程序。

[E] 3.12 如果 TABLE 的地址是 0x10100，修改图 3-19 中的程序。

[M] 3.13 使用图 3-17 中的七段显示器以及图 3-14 中的定时器接口寄存器，写一个 RISC 风格的程序，显示十进制数字的重复序列 0、1、2、…、9、0、…，每一个数字显示一秒钟。假设定时器电路中的计数器是由 100MHz 的时钟驱动的。

[M] 3.14 写一个 CISC 风格的程序完成习题 3.13 中的任务。

[D] 3.15 使用两个图 3-17 所示的七段显示器以及图 3-14 中的定时器接口寄存器，写一个 RISC 风格的程序，显示十进制数的重复序列 0、1、2、…、98、99、0、…，每个数显示一秒钟。假设定时器电路中的计数器是由 100MHz 的时钟驱动的。

[D] 3.16 写一个 CISC 风格的程序完成习题 3.15 中的任务。

[D] 3.17 写一个 RISC 风格的程序，计算并以小时（0 到 23）和分钟（0 到 59）的形式显示时间。显示器包括 4 个图 3-17 所示的七段显示器。还有一个具有图 3-14 所示接口寄存器的定时器电路，其计数器是由 100MHz 的时钟驱动的。

[D] 3.18 写一个 CISC 风格的程序完成习题 3.17 中的任务。

[M] 3.19 写一个 RISC 风格的程序，倒序显示用户的名字。该程序应能显示一个提示符，要求在键盘上输入用户名的字符，并以回车符（CR）结束。该程序还应该能接受字符序列并将它们保存在主存中。然后它应显示一条消息，来表示该用户的名字将被倒序显示，紧接着就以相反的顺序显示用户名的字符。

[M] 3.20 写一个 CISC 风格的程序完成习题 3.19 中的任务。

[M] 3.21 写一个 RISC 风格的程序，确定用户在键盘上输入的单词是否是回文。回文是指其字符以正写

顺序或者相反顺序写出时都相同的单词。该程序应能显示一个提示符，要求用户在键盘上输入任意一个单词的字符，并以回车符（CR）结束。该程序还应该能读取这些字符并将它们保存在主存中。然后分析该单词以确定它是否是一个回文。最后，该程序应显示一条消息，以说明分析的结果。

[M] 3.22 写一个 CISC 风格的程序完成习题 3.21 中的任务。 127

[D] 3.23 写一个 RISC 风格的程序，在一个标准的 80 个字符的行上水平居中显示一个字符串，并将其封闭在一个盒子中，如下所示。

```
+-------------+
| sample text |
+-------------+
```

该字符串存储在以地址 STRING 开始的主存中。字符串的末尾有一个 NUL 控制字符（值为 0）。如果该字符串含有 78 个以上的字符（包括空格），则程序应将显示的字符串截断为 78 个字符。可以将例 2.1 中确定字符串长度的程序改写为本问题中的程序所使用的一个子程序。假设显示设备具有图 3-3 所示的接口。

[D] 3.24 写一个 CISC 风格的程序完成习题 3.23 中的任务。

[D] 3.25 写一个 RISC 风格的程序，显示一个 ASCII 字符编码的文本长序列。为了适应每行 80 个字符，该文本序列应能自动换行。在显示下一个单词前，该程序必须确定本行剩下的空间是否足以显示这个单词。如果不够，该单词应该在下一行的开头显示。当到达要显示的字符序列末端的 NUL 控制字符（值为 0）时，显示过程结束。假设该字符序列不使用除了末端的 NUL 字符之外的控制字符，因此单词间仅用一个空格字符隔开。假设显示设备具有图 3-3 所示的接口。

[D] 3.26 写一个 CISC 风格的程序完成习题 3.25 中的任务。 128

第4章
Computer Organization and Embedded Systems, Sixth Edition

# 软　件

## 本章目标

在本章中你将学习以下内容：

- 准备及运行程序所需的软件
- 汇编程序
- 装载程序
- 连接程序
- 编译器
- 调试器
- 汇编语言与C语言之间的交互
- 操作系统

[129]

第2章介绍了计算机的指令集并阐述了如何用汇编语言编写程序。第3章中又描述了如何编写可以执行输入/输出操作的程序。在本章中，我们将概述准备及运行程序所需的软件。

汇编语言程序是使用程序员易于理解的符号表示来编写的，而这些程序必须在翻译成机器语言代码之后才能在计算机上运行，就像在2.5节中所解释的那样。这是由汇编程序来完成的，它解释表示机器指令的助记符以及表示数据声明的汇编指示。

在阐述了如何编写汇编语言程序之后，现在我们将讨论程序运行前的整个准备过程，我们将描述：

- 由汇编语言编写的源程序是如何被翻译成包含二进制形式的机器指令及数据的目标程序的？
- 目标程序是如何装入计算机的存储器中的？
- 程序的执行是如何开始及结束的？
- 一个较大的程序是如何由若干个相关的程序连接而成的？
- 程序执行期间是如何识别程序设计错误的？

然后，我们将探讨用高级语言（如C）准备程序时的一些相关问题。最后还将介绍操作系统软件在管理和协调计算机资源的使用方面的作用。

## 4.1　汇编过程

为了准备源程序，程序员可以通过使用一个称为文本编辑器（text editor）的实用程序在键盘上输入源程序语句并将其保存在一个文件（file）中。包含源程序的文件是一个二进制编码的字符序列。文件是由用户选择的名字来标识的，通常将文件存储在辅助存储设备中，如磁盘。

源文件准备就绪后，程序员使用另一个称为汇编程序(assembler)的实用程序来将汇编语言编写的源程序翻译成由机器指令组成的目标程序，这一过程通常称为汇编(assemble)一个程序。汇编程序还可将汇编语言表示的数据转化为二进制模式，它也是目标程序的一部分。在将源文件从磁盘装入存储器并将其翻译成目标程序之后，汇编程序将把目标程序存储在磁盘上一个单独的文件中。

源程序使用助记符来表示机器指令中的操作（OP）码。有一组语法规则来管理这些指令中数据操作数的寻址方式规范。汇编程序为操作码和其他的指令字段生成二进制编码。

汇编程序可以识别用于指明数字和字符的汇编指示以及为数据又分配存储空间的汇编指示。程序员可以使用 EQU（等于）指示来定义常量的名字，然后，这些名字可以被当作指令中的操作数在源程序中直接使用。同样这些名字还可以被定义为转移目标的地址标签，子程序 130
的入口指针，或者存储器中的数据单元。地址标签可根据它们相对于汇编后程序的开头位置来赋值。

当汇编程序扫描一个源程序时，它将所有名字及其对应的值都记录在一个符号表（symbol table）中。每当出现一个名字，就用它在表中的值替换它。

### 二遍扫描汇编程序

当一个名字在定义它的值之前作为一个操作数出现时，便会引发一个问题。例如，当需要向前转移到在后面的程序中才会出现的一个地址标签时就会发生这种情况。正如在 2.5.2 节中所讨论的，汇编程序使用转移目标地址来计算转移的偏移量。向前转移的情况下，汇编程序不能确定转移目标的地址，因为地址标签的值还没有被记录到符号表中。

解决这个问题的常用方法是让汇编程序对源程序做二遍扫描。在第一遍扫描期间，它建立一个符号表。对于 EQU 指示，每个名字和它定义的值被记录在符号表中。对于地址标签，汇编程序用每个名字相对于源程序开始的位置来确定它的值，也就是把定义名字的语句出现之前所处理过的所有机器指令的大小相加来确定。在第一遍扫描结束时，源程序中出现的所有名字都已经在符号表中被分配了数值。然后汇编程序对源程序做第二遍扫描，在符号表中查找它遇到的每一个名字，并替换相应的数值。这样的二遍扫描（two-pass）汇编程序会产生一个完整的目标程序。

## 4.2 装载及执行目标程序

汇编程序所产生的目标程序存储在磁盘上的文件中。为了执行一个特定的目标程序，首先要将其从磁盘装入内存中，然后再把将要执行的第一条指令的地址装入程序计数器中。用来执行这些操作的实用程序为装载程序（loader）。

当用户输入一个命令来执行存储在磁盘上的目标程序时，装载程序将被调用。用户命令指明了目标文件的名字，这使得装载程序可以从磁盘上找到该文件。然后装载程序将目标程序从磁盘传送到内存的指定位置上。它必须知道这个程序的长度和将要存放它的内存地址。汇编程序通常把这些信息放在目标文件的头部，即目标程序的机器指令和数据之前。

用户可以通过键盘来输入命令，也可以使用一种更常用的方法，即图形用户界面（graphical user interface，GUI）来输入命令。在这种情况下，用户使用鼠标来选择所需的目标文件。然后，GUI 软件将目标文件在磁盘上的位置信息传递给装载程序。 131

一旦目标程序被装入内存中，装载程序就转移到将要执行的第一条指令处开始执行。在源程序中，程序员用一个特定的地址标签，如 START，来指示第一条指令。汇编程序将该地址标签的值包含在目标程序的头部。

当目标程序执行完它的任务时，它的执行必须以一种明确的方式结束。这将收回包含目标程序的内存空间，并允许用户输入一个新的命令来执行另一个目标程序。这些问题通常由操作系统（OS）软件来处理。我们将在 4.9 节讨论操作系统软件。

## 4.3 连接程序

在前面的章节中，我们假设一个特定程序中的所有指令及数据都在一个单独的源文件中指定，汇编程序从该源文件产生一个目标程序。而在许多情况下，程序员可能希望调用其他程序员所编写的子程序。但是将所有需要的子程序从多个不同的源文件收集到一个单独的源文件中来由汇编程序处理是不方便或不实际的。

然而，有一个常见的过程可以对每一个源文件单独汇编。在这种情况下，每一个单独的输出文件将不会是一个完整的目标程序。每个程序可能包含外部名（external name）的引用，外部名是在其他源文件中定义的地址标签。当汇编程序处理一个源文件时，它将识别出这样的外部引用，并建立一个关于这些名字和引用它们的指令的列表，然后汇编程序将该列表包含在它从源程序产生的目标文件中。

一个被称为连接程序（linker）的实用程序可用来将不同目标文件的内容合并到一个目标程序中。它使用每个目标文件中记录的信息来解析外部名的引用。连接程序需要源文件中所定义地址标签的相对地址，从而当它合并不同的目标文件时可以确定地址标签的绝对地址值。为达到这个目的，必须从每个源文件中导出（export）可能在其他源文件中引用的地址标签相关信息。通常程序员需要标记出将要被导出的特定标签。汇编程序将导出的名字与程序中用到的外部名列表和引用外部名的指令一起包含在它所产生的每一个目标文件中。

连接程序使用每个目标文件中的信息和机器语言程序的已知长度来构造最终合并的目标文件的存储器映像。一旦所有单独的目标文件被收集在一起并在内存中给它们分配了最终的存储单元，被导出的地址标签所对应的最终值就可以被确定。这时候就可以解决外部名的引用问题了。由连接程序确定的最终地址值将代入包含外部引用的特定指令中。一旦解决了所有的外部引用问题，就生成了最终的目标程序。

程序员可以在目标文件中明确地确定一些指令和数据的地址。这可以通过在汇编语言源程序中使用类似 ORIGIN 这样的指示语句来完成。在这种情况下，程序员必须确保不同目标文
132 件中的指令和数据不能在内存中重叠。另一种更灵活的方法是不使用 ORIGIN 指示，而是让连接程序自由选择目标程序的起始地址并相应地分配绝对地址。连接程序要确保不同的目标文件不会彼此重叠或与特定的存储单元（如中断向量）重叠。

## 4.4 库

为某个应用程序所编写的子程序可能也会被其他的应用程序使用。一个普遍的做法是将包含这些子程序的目标文件聚集到一个存储在磁盘上的库（library）文件中。然后，库文件中的子程序可以与其他任何应用程序的目标文件进行链接。库文件是由称为归档程序（archiver）的实用程序用来创建的。库文件包含连接程序用来解决调用库例程的程序中外部名引用问题所需的信息。

当调用连接程序时，程序员需要指定所需的库文件。然后连接程序从库文件中提取相关的目标文件并将其包含到最终的目标程序中。

## 4.5 编译器

用汇编语言编程需要了解特定机器的细节知识，因为不同计算机的细节各不相同。而用 C、C++ 或 Java 这样的高级语言编程则不需要这些知识。用高级语言编写的程序在一台计算机上运行之前，它必须先被翻译成该计算机的汇编语言，然后再被翻译成机器语言。一个被称为

编译器（compiler）的实用程序用来执行第一个任务。程序员用高级语言准备源程序，并将其存储在磁盘上。编译器从这个源文件生成汇编语言指令和指示，并将其写入一个输出文件中。然后，编译器调用汇编程序对该文件进行汇编。

将一个高级语言编写的源程序划分为多个文件，并根据相关的任务将子程序分类组合是非常方便的。在每个源文件中，必须声明其他文件中的外部子程序名和数据变量名。这就需要编译器来检查数据类型并检测错误。针对每一个源文件，编译器都会生成一个汇编语言文件，然后调用汇编程序来产生一个目标文件。连接程序合并所有的目标文件，包括库例程，来创建最终的目标程序。

用高级语言编程的一个重要好处就是编译器会自动完成用汇编语言编程时程序员必须做的许多繁琐的任务。例如，在生成子程序的汇编语言表示形式时，编译器将执行所有跟堆栈帧的管理有关的任务。 133

### 4.5.1 编译器优化

如果编译器使用一个简单的方法将高级语言编写的源程序编译成汇编语言程序，那么所产生的程序在执行时间或大小方面可能不会是最有效的。如果编译器使用一些技术，如重排序指令，就会获得较高的性能。具有这种功能的编译器被称为优化（optimiz）编译器。

因为一个程序的大部分执行时间都花费在了循环上，所以编译器可以特别对循环进行有效的优化。例如，一个高级语言源程序使用一个内存变量作为循环计数器。每次循环都需要读出该变量，递增其值后再写回。用汇编语言实现该任务时可使用 Load、Add 和 Store 指令来完成循环。一种更好的实现方法是让编译器意识到，在执行循环时可以把计数器的值放入一个寄存器中维护。在这种情况下，循环中就不再需要 Load 和 Store 指令。只需在进入循环前使用一条 Load 指令来将初始值放入到寄存器中，在退出循环后，用 Store 指令来记录计数器的最终值即可。

### 4.5.2 组合不同语言编写的程序

4.3 节描述了连接程序，它可以将多个目标文件链接起来生成目标程序。在某些情况下，程序员可能希望将用高级语言编写的源文件所生成的目标文件以及用汇编语言编写的源文件所生成的目标文件组合起来。例如，程序员可以编写特定的汇编语言子程序，这些子程序被精心编写以实现高性能。然后，一个高级语言源程序可以调用这些汇编语言子程序。类似地，汇编语言程序也可以调用高级语言子程序。

图 4-1 给出了从多个源文件和库例程生成目标程序的完整流程。

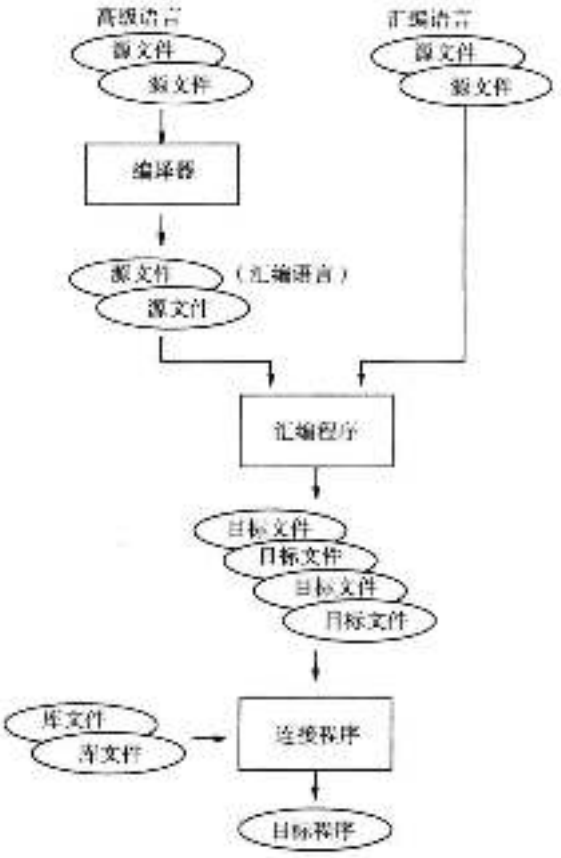



图 4-1 生成目标程序的总体流程图

## 4.6 调试器

当在程序的源文件中没有语法错误或未知的名称时，就可以成功地生成目标程序。汇编程序、编译器、连接程序可以检测并报告这些问题。然后程序员再对源程序进行必要的修改。

然而，当目标程序执行时，由于存在难以查出的编程错误或 bug，所以可能会产生错误的结果。为了帮助程序员找出这些错误，可以使用一个被称为调试器（debugger）的实用程序。它使程序员能够在某些感兴趣的点上停止执行目标程序，检查各种处理器寄存器和存储单元的
134 内容。程序员可以用这种方式将任何执行点的计算值与期望的结果进行比较，以此来确定可能
135 存在程序错误的地方。有了这些信息，程序员就可以修改源文件的错误。

为支持调试器的功能，处理器通常需要有特定的操作模式以及特定的中断。调试功能的两个实例是跟踪模式和断点。

### 1. 跟踪模式

当处理器运行在跟踪（trace）模式下时，每执行完一条指令就产生一次中断。每当发生这样的中断，调试器程序中的中断服务程序就会被调用。该中断服务程序允许调试器对执行过程进行控制，可以使用户输入命令来检查寄存器和存储单元的内容。当用户输入一条命令来恢复目标程序的执行时，中断返回指令被执行。接着执行被调试程序的下一条指令，然后用另一个中断再次转到调试器中。当进入调试器程序时，跟踪模式的中断将被自动禁止，当返回到目标程序时再重新允许中断。

### 2. 断点

断点（breakpoint）提供了与跟踪相似的基于中断的调试功能，只是被调试的目标程序只在程序员指定的特定点发生中断。例如，程序员可以设置一个断点来确定目标程序究竟会不会运行一个特定的子程序。如果运行了就通过一个中断转到调试器中。然后程序员可以检查此时的处理状态。使用断点的好处是程序的执行可以全速进行直到遇到断点。

断点通常通过一条被称为陷阱或软件中断的特殊指令来实现。该指令的执行会产生与接收到一个硬件中断请求时相同的动作。当调试器有执行控制的权限时，它允许用户在目标程序的指令 $i$ 前设置一个断点来中断程序的执行。调试器将指令 $i$ 保存到一个临时位置，并用一条软件中断指令替换它。然后用户输入命令来恢复目标程序的执行，调试器执行一条中断返回指令。接着，目标程序的指令会被正常处理，直到再次遇到软件中断指令。那时，中断处理会导致再次转入到调试器中，允许用户检查处理状态。

当用户输入命令来恢复目标程序的执行时，调试器必须执行几个任务，而不是仅仅执行指令 $i$，它还将再次设置同样的断点。它必须首先将指令 $i$ 放回到它在程序中的原始位置。当程序恢复执行时，指令 $i$ 将成为第一条要执行的指令。然后，调试器必须重新设置断点。这需要在指令 $i$ 执行后安排第二次中断的发生。为实现此目的，如果可能的话可以使用跟踪模式，或者，也可以在指令 $i+1$ 的位置上设置一个临时断点，然后继续执行被调试的程序。当指令 $i$ 执行完毕，由于指令 $i+1$ 处设置了临时中断，因此发生第二次中断。此时，调试器恢复指令
136 $i+1$，在指令 $i$ 处重新设置断点，并继续执行被中断的程序。

## 4.7 使用高级语言实现输入 / 输出任务

编译器、汇编程序以及连接程序为程序员提供了相当大的灵活性。源程序可以完全用汇编语言，完全用高级语言或者用几种语言的组合来编写。在大多数应用中，人们更喜欢使用高级语言，因为程序开发时间较短，并且所需的代码易于生成和维护。在本节及下一节中，我们

将给出一些使用C语言实现I/O任务的实例程序来说明这种方法。

考虑下面的I/O任务。一个程序使用轮询法来从键盘上读取用户输入的8位字符并将它们发送到显示器上。第3章介绍过此类设备的存储器映射接口实例。图4-2显示了一个实现该I/O任务的汇编语言程序，该程序使用了图3-3所示的接口。

```
KBD_DATA        EQU             0x4000          键盘数据寄存器（8位）
KBD_STATUS      EQU             0x4004          键盘状态寄存器（第一位为KIN标志）
DISP_DATA       EQU             0x4010          显示器数据寄存器（8位）
DISP_STATUS     EQU             0x4014          显示器状态寄存器（第二位为DOUT标志）

                Move            R2, #KBD_DATA   键盘设备接口指针
                Move            R3, #DISP_DATA  显示设备接口指针

KBD_LOOP:       LoadByte        R4, 4(R2)       检查是否有来自键盘的字符
                And             R4, R4, #2
                Branch_if_[R4]=0 KBD_LOOP
                LoadByte        R5, (R2)        读取接收到的字符
DISP_LOOP:      LoadByte        R4, 4(R3)       检查显示器是否准备好显示一个字符
                And             R4, R4, #4
                Branch_if_[R4]=0 DISP_LOOP
                StoreByte       R5, (R3)        将接收到的字符写到显示器
                Branch          KBD_LOOP
```

图4-2 将字符从键盘传送至显示器的汇编语言程序

图4-3给出了一个执行同样任务的C语言程序。在C语言中，指针可以被设置为指向任意的存储单元，包括存储器映射I/O单元。此指针的值是我们所要访问单元的地址。如果该单元的内容被当作是一个字符，那么指针就应该被声明为字符类型的。这会将该内容定义为一个字节长度，也就是图3-3中I/O寄存器的大小。图4-3中的define语句用于将所需的地址常量与指针的符号名关联起来。这些语句与图4-2中的EQU语句达到同样的目的。它们使得编译器将程序中的符号名替换为数值。define语句同样也指明了指针的数据类型。然后编译器可以用已知值和正确的数据大小来生成汇编语言指令。 137

```
/* 定义寄存器地址 */
#define   KBD_DATA      (volatile char *) 0x4000
#define   KBD_STATUS    (volatile char *) 0x4004
#define   DISP_DATA     (volatile char *) 0x4010
#define   DISP_STATUS   (volatile char *) 0x4014

void main()
{
    char ch;

    /* 传输字符 */
    while (1) {                                   /* 开始循环 */
        while ((*KBD_STATUS & 0x2) == 0);         /* 等待新字符 */
        ch = *KBD_DATA;                           /* 从键盘读取字符 */
        while ((*DISP_STATUS & 0x4) == 0);        /* 等待显示器准备就绪 */
        *DISP_DATA = ch;                          /* 传输字符到显示器 */
    }
}
```

图4-3 与图4-2中汇编语言程序执行同样任务的C程序

注意，指针KBD_STATUS和DISP_STATUS被定义为volatile类型。这是必需的，因为程序只读取相应单元的内容，而没有数据会被写到这些单元中。优化编译器可能会删除那些看起

来没有影响的程序语句，包括那些引用被读取过但从未写入过的存储单元的语句。因为存储器映射寄存器 KBD_STATUS 和 DISP_STATUS 的内容是随程序外部的影响而变化的，所以必须告知编译器这一事实。编译器将不会删除那些包含被声明为 volatile 类型的指针或其他变量的语句。

对于包含高速缓存的计算机来说，一些编译器对 volatile 类型的指针或变量有额外的解释。高速缓存是一个容量小、速度快的存储器，它保持了主存中数据的副本。当数据可在高速缓存中找到时，可以更快速地执行那些引用存储单元的指令。然而，存储器映射 I/O 寄存器的数据是不能保存在高速缓存中的，因为这些寄存器的内容会随外部的影响而变化。这样，对这些单元的引用就应该绕过（或通开，bypass）高速缓存，而直接访问 I/O 寄存器。将这些单元的指针声明为 volatile 类型可以通知编译器不必进行不必要的优化，可产生绕过高速缓存的存储器访问指令。

在图 4-3 中，我们将表示位位置特定值的数值常量包含到两个状态寄存器中。例如，下列语句

```
while ((*KBD_STATUS & 0x2) == 0);
```

[138] 其中的常量 0x2 用于检测 KBD_STATUS 寄存器中的 b1 位是否被设置过。这里使用的方法使其更容易将给定值与图 3-3 中的设备接口规范进行比较。在编写 C 程序时，一种更通常的方法是包含 define 语句来将有意义的名称与这样的常量值关联起来，然后在程序的其他部分使用该名称。

## 4.8 汇编语言与 C 语言的交互

一个程序可能偶尔需要访问处理器中的控制寄存器。例如，中断服务程序的初始化就需要访问处理器中的控制寄存器。编译器不能根据高级语言中的一个语句来生成访问处理器中控制寄存器的汇编语言指令。因为达到此目的需要使用汇编语言指令，所以编译器允许将汇编语言指令直接包含在高级语言程序中。本节将介绍这种方法。

考虑一个从键盘传输字符到显示器的 I/O 任务。假设使用中断从键盘接口接收字符。为使例子简单，假设中断服务程序将每一个接收到的字符直接发送到显示器接口，而不需要轮询显示器接口的状态，但前提是字符能以足够低的速率被接收，以便于显示器处理。

此程序中该任务的初始化需要访问 I/O 寄存器和处理器控制寄存器。图 3-3 中的 I/O 接口需要配置为当 KEN=1 时可以提出中断请求，KBD_CONT 寄存器中相应的中断允许位 KIE 必须置为 1。同时，需要将图 3-7 中处理器状态（PS）寄存器的 IE 位和 IENABLE 控制寄存器的 KBD 位置为 1 以允许处理器中断。

第 3 章中描述了几种不同的方法识别当一个特定的中断发生时所要执行的中断服务程序的首地址。其中向量中断法对不同的中断源使用一些预定的存储单元来保存相应中断服务程序的地址。在这一节中，为简单起见，我们将假设对于所有的中断只存在一个地址为 0x20 的中断向量 IVECT，该向量必须用中断服务程序的地址来进行初始化。

图 4-4 显示了一个使用中断来从键盘上读取字符的汇编语言程序。主程序将中断服务程序的地址装入 IVECT 单元中，并将键盘接口中控制寄存器的 KIN 位置为 1，同时将处理器中 IENABLE 和 PS 寄存器的中断允许位也置为 1。在每一次键盘中断时，中断服务程序就读取输入字符，并将其发送到显示器上。

现在考虑使用 C 语言来实现该 I/O 任务。像 C 这样的高级语言是不能直接处理诸如中断之类的硬件功能的。编写使用中断的 C 程序我们需要解决如下两个问题：

```
IVECT       EQU     0x20        中断服务程序向量
KBD_DATA    EQU     0x4000      键盘数据寄存器（8位）
KBD_STATUS  EQU     0x4004      键盘状态寄存器（第一位是 KIN 标志）
KBD_CONT    EQU     0x4008      键盘控制寄存器（第一位是 KIE 标志）
DISP_DATA   EQU     0x4010      显示器数据寄存器（8位）
DISP_STATUS EQU     0x4014      显示器状态寄存器（第二位是 DOUT 标志）

主程序
MAIN:       Move    R2, #KBD_DATA   键盘接口指针
            Move    R3, #0x2
            StoreByte R3, 8(R2)     设置键盘以产生中断
            Move    R2, #IVECT      向量指针
            Move    R3, #INTSERV    中断服务程序的开始
            Store   R3, (R2)        设置中断向量
            Move    R2, #0x2        允许处理器识别键盘中断
            MoveControl IENABLE, R2
            Move    R2, #0x1        为处理器设置中断允许位
            MoveControl PS, R2
LOOP:       Branch  LOOP            等待等待循环

中断服务程序
INTSERV:    Subtract SP, SP, #8     保存寄存器
            Store   R2, 4(SP)
            Store   R3, (SP)
            Move    R2, #KBD_DATA   键盘接口指针
            LoadByte R3, (R2)       读取下一个字符
            Move    R2, #DISP_DATA  显示器接口指针
            StoreByte R3, (R2)      将接收到的字符写入显示器
            Load    R2, 4(SP)       恢复寄存器
            Load    R3, (SP)
            Add     SP, SP, #8
            Return-from-interrupt
```

图 4-4 使用中断传输字符的汇编语言程序

- 如何访问处理器控制寄存器？
- 如何编写中断服务程序？

中断方法的初始化要求设置 IENABLE 和 PS 寄存器中的控制位。图 4-3 所示的基于指针的方法可用于访问存储器映射 I/O 寄存器，但不能用于访问控制寄存器 IENABLE 和 PS，因为这两个寄存器没有地址。相反，可以通过在 C 程序中直接嵌入适当的汇编语言指令来访问这些寄存器。我们可通过一个特殊的指示符来告知编译器实现此功能。例如，语句

139
-
140

```
asm ("MoveControl PS, R2");
```

使得 C 编译器将引号之间的汇编语言指令插入到编译后的代码中。由于寄存器 R2 可能已经被编译器生成的指令所使用，所以任何插入的汇编语言指令都不能破坏它的内容。一个简单的解决方法是在 R2 被 MoveControl 指令使用并修改前将 R2 的内容保存到堆栈中，然后在该指令结束后再将它们恢复。本例中我们将使用这种方法。但是，应该注意的是，编译器提供了更复杂的方法来管理

```
#define   KBD_DATA   (volatile char *) 0x4000
#define   DISP_DATA  (volatile char *) 0x4010

void main()
{
        .
        .
        .
}

void intserv()
{
   *DISP_DATA = *KBD_DATA;   /* 传输一个字符 */
}
```

图 4-5 在 C 程序中将中断服务程序表示为函数

asm 指示符中所指定的寄存器的使用方法。

第二个问题是中断服务程序。C 语言中需要将该中断服务程序编写为一个函数。然而，编译器会将所有的 C 函数实现为以 Return-from-subroutine（从子程序返回）指令结束的子程序。图 4-5 给出了一个例子。主函数中执行了一些未指定的任务。名为 intserv 的函数将一个字符从键盘传送到显示器。函数 intserv 经过编译后生成的代码为：

```
<保存寄存器>
LoadByte R2, 0x4000(R0)
StoreByte R2, 0x4010(R0)
<恢复寄存器>
Return-from-subroutine
```

因为 I/O 寄存器的地址在 16 位以内，所以编译器可以使用绝对寻址方式，如 2.4.3 节所讨论的，寄存器 R0 始终包含值 0。

为了将 intserv 函数当作中断服务程序使用，它必须以 Return-from-interrupt 指令结束。这条指令用来将程序计数器和处理器状态寄存器的内容恢复到它们在中断发生之前的值。我们可
141 以通过使用语句

```
asm ("Return-from-interrupt");
```

在程序中插入 Return-from-interrupt 指令作为 intserv 函数的最后一个语句。使用该语句，函数编译后的代码为：

```
<保存寄存器>
LoadByte R2, 0x4000(R0)
StoreByte R2, 0x4010(R0)
Return-from-interrupt
<恢复寄存器>
Return-from-subroutine
```

如同对所有的函数那样，编译器还是在末尾包含了恢复寄存器的代码和 Return-from-subroutine 指令。然而，编译器在函数中包含 Return-from-interrupt 指令意味着在其之后的代码将永远不会被执行。因为中断可以在程序中的任何位置发生，所以如果不能成功地对在函数中修改过的寄存器恢复其原始值的话，会导致程序的后续执行出错。更关键的是，如果不能恢复堆栈指针的正确值的话，会导致嵌套了程序的堆栈帧被破坏。

有两种方法可以在高级语言（如 C）中正确地支持中断。第一种方法需要扩展高级语言的语法，增加一个可以标识出中断服务程序的特殊关键字。例如，C 编译器可能会识别出在函数定义开始处的 interrupt 关键字，比如将图 4-5 中的函数改写为：

```
interrupt void intserv() {…}
```

该关键字指示编译器将 Return-from-subroutine 指令用 Return-from-interrupt 指令替换。寄存器还是像以前一样被保存和恢复。但是，不是所有的 C 编译器都提供这种功能。

第二种方法是使用汇编语言来编写中断处理程序并使用连接程序将其连接到 C 程序中。这种情况下，中断处理程序必须首先保存链接寄存器的值，因为主程序调用一个子程序后可能会产生中断。在保存了链接寄存器的值之后，中断处理程序可以调用一个 C 语言子程序来提供中断服务。这样的话，在高级语言源程序中就不需要特殊的关键字了。从子程序返回后，中断处理程序恢复链接寄存器的值并执行 Return-from-interrupt 指令。

现在我们可以编写一个使用中断来从键盘发送字符到显示器的 C 程序了。图 4-6 给出了

一个等同于图 4-4 的程序。我们使用了基于 C 编译器中特殊关键字的方法，因为它允许整个程序都在单独的一个高级语言源文件中。注意，存储器映射 I/O 寄存器的指针是字符类型的，因为它们指向与设备接口中 8 位寄存器相对应的单元。指针 IVECT 是无符号整数类型的，因为它指向一个存储 4 字节中断向量的存储单元。 [142]

```
#define   IVECT          (volatile unsigned int *) 0x20
#define   KBD_DATA       (volatile char *) 0x4000
#define   KBD_CONT       (volatile char *) 0x4008
#define   DISP_DATA      (volatile char *) 0x4010
#define   DISP_STATUS    (volatile char *) 0x4014

interrupt void intserv();   /* 前置声明 */

void main()
{
   /* 基于中断的字符传输的初始化 */
   *KBD_CONT = 0x2;                          /* 允许键盘中断 */
   *IVECT = (unsigned int) &intserv;         /* 设置中断向量 */
   asm ("Subtract    SP, SP, #4");           /* 保存寄存器 R2 */
   asm ("Store    R2, (SP)");
   asm ("Move    R2, #0x2");                 /* 允许处理器识别键盘中断 */
   asm ("MoveControl    IENAble, R2");
   asm ("Move    R2, #0x1");                 /* 允许处理器中断 */
   asm ("MoveControl    PS, R2");
   asm ("Load    R2, (SP)");                 /* 恢复寄存器 R2 */
   asm ("Add    SP, SP, #4");

   while (1)                                 /* 连续循环 */
   {
                                             /* 使用中断服务程序传输字符 */
   }
}

interrupt void intserv()   /* 关键字指示编译器将函数当作中断程序 */
{
   *DISP_DATA = *KBD_DATA;   /* 传输一个字符 */
   /* 编译器将在函数末尾插入 Return-from-interrupt 指令 */
}
```

图 4-6 使用中断实现字符传输的 C 程序

## 4.9 操作系统

前面几节描述了如何在多个实用程序的帮助下准备及执行应用程序。操作系统（operating system，OS）将使本章中描述的所有任务都可以很方便地实现，它是大多数计算机中一个关键的软件组件，负责协调计算机中的所有活动。操作系统通常包含一些常驻内存的基本例程，以 [143]
及存储在磁盘上、在需要的时候会被加载到存储器中执行的各种实用程序。

在程序执行期间，操作系统管理计算机的处理、存储器及输入/输出资源。它可以解释用户的命令、分配内存和磁盘空间、在内存和磁盘之间传送信息、处理 I/O 操作等。它还可以使用户使用文本编辑器、编译器、汇编程序和连接程序来准备应用程序。装载程序通常是操作系统的一部分，当用户输入执行一个应用程序的命令时装载程序将被调用。在这一节中，我们的

目标是对操作系统所执行的重要功能进行一个基本的描述。更深入的讨论超出了本书的范围（参见参考文献[1]）。

### 4.9.1 引导程序

通用计算机的操作系统是一个很大且很复杂的软件集合。操作系统的所有部分，包括常驻内存的部分，通常都被存储在磁盘上。引导程序（boot-strapping process）是用来将操作系统中常驻内存的部分加载到内存中，使操作系统可以开始执行，并对计算机的资源进行控制管理。

当打开计算机的时候，引导程序开始工作，处理器从一个预定的单元提取第一条指令。那个单元必须在内存的一个永久部分中，在关闭计算机的时候也能保留其内容。放置在该单元的一个小程序使得处理器能够将操作系统中更大的部分逐渐从磁盘传送到内存的非永久部分中。在这个引导序列中执行的每个程序将操作系统的更多部分从磁盘传送到内存中，并对计算机内存和I/O设备进行必要的初始化。最后，装载程序和操作系统中负责处理用户命令的部分被传送到内存中。这使得操作系统可以开始接收命令来加载并执行存储在磁盘文件中的应用程序了。

### 4.9.2 管理应用程序的执行

为了理解操作系统的基本原理，我们来讨论一下一台由处理器及键盘、显示器、磁盘和打印机等I/O设备组成的计算机。我们首先讨论一个应用程序的运行步骤。然后，再探讨操作系统是如何管理多个应用程序的执行的。

为了执行一个存储在磁盘文件中的应用程序，用户输入一个命令使得装载程序将该文件传输到内存中。当传输结束后，程序开始执行。假定该程序的任务包括将一个数据文件从磁盘读取到内存中，对数据进行某些计算后再打印结果。当程序执行到某处需要读取数据文件时，它请求操作系统来将数据文件从磁盘传送到内存。数据传送完毕后，操作系统将执行控制权传回
144 给应用程序，应用程序继续执行所需的计算。当计算完成后，存储在内存中的结果已经准备就绪等待打印，应用程序再发送一个请求给操作系统。然后执行一个操作系统例程来打印结果。

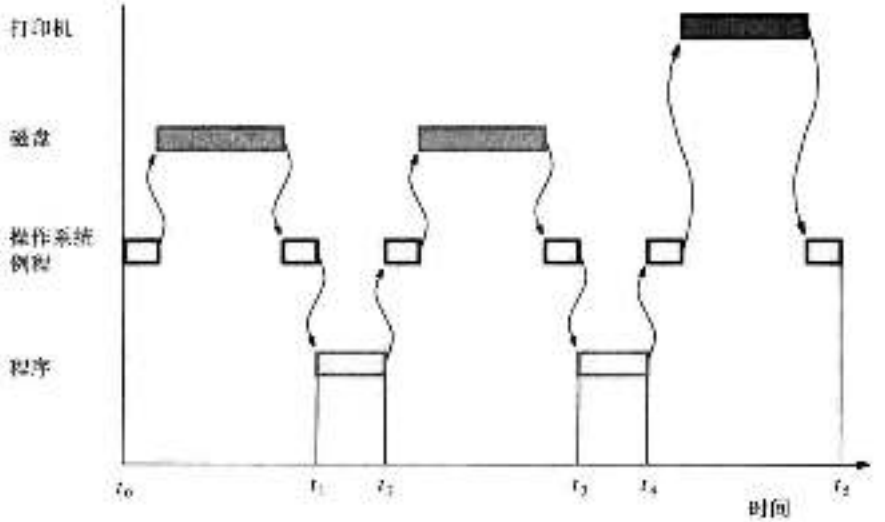



图4-7 用户程序与操作系统例程间转换执行控制权的时间线

执行控制权在应用程序和操作系统例程之间来回传递，它们共享处理器来执行各自的任

务。说明此活动的一种简便方法是使用时线图，如图 4-7 所示。在 $t_0$ 到 $t_1$ 时间段中，装载程序将目标程序从磁盘传送到内存；在 $t_1$ 时刻，操作系统将执行控制权传递给应用程序，然后应用程序开始运行直到它需要读取磁盘上的数据；在 $t_2$ 到 $t_3$ 时间段中，操作系统传输所请求的数据；最后，在 $t_4$ 到 $t_5$ 时间段中，操作系统打印存储在内存中的运算结果。

计算机的资源在同时执行多个应用程序时可以被更高效地利用。注意，图 4-7 中 $t_4$ 到 $t_5$ 时间段的大部分时间磁盘和处理器都处于闲置状态。如果在此期间允许用户输入命令，则操作系统可以在打印机进行打印的时候装载并开始执行另一个应用程序。当两个程序不竞争访问计算机中的同一资源时，它们的计算过程和 I/O 请求是并行处理的。操作系统负责管理几个应用程序的并行执行，以尽可能地充分利用计算机的所有资源。这种并行执行的方法被称为多道程序（multiprogram）或多任务（multitask）。在这种操作模式中，处理器以交错的时间顺序来执行多个程序，并与不同 I/O 设备执行的任务重叠执行。 [145]

### 4.9.3 中断在操作系统中的使用

操作系统在执行 I/O 操作、与程序通信和控制程序执行时大量地使用了中断。中断机制使操作系统可以分配优先级、从一个程序切换到另一个程序、终止程序、实现安全和保护功能以及协调 I/O 活动等。我们将简单讨论一下这些方面的内容，以说明如何使用中断。

操作系统将所有与计算机相连的并且可以发出中断的设备的中断服务程序组织在一起。在一台安装了操作系统的通用计算机中，应用程序本身并不直接执行 I/O 操作。当应用程序需要进行输入或输出操作时，它指向需要传输的数据，并请求操作系统来执行该操作。应用程序的请求通常是通过一个库子程序发出软件中断并进入操作系统例程来完成的。操作系统暂停执行发出请求的程序，然后开始执行所请求的 I/O 操作。当 I/O 操作结束后，通常通过一个硬件中断来告知操作系统这一状况。然后操作系统允许暂停的程序继续执行。操作系统和应用程序之间的控制权传递是使用软件中断来实现的。

操作系统为应用程序提供各种各样的应用服务。为了便于实现这些服务，处理器可能有几条不同的软件中断指令，其中每一条都有自己的中断向量。根据所请求的服务，它们可以被用来调用操作系统的不同部分；或者，处理器可能只有一条软件中断指令，该指令用一个立即操作数来指明所需的服务。

操作系统必须确保应用程序的执行能正确地终止。在应用程序的末尾执行一条适当的软件中断指令可以告知操作系统获取控制权并终止程序。你应该还记得，在目标程序的头部包含了程序在内存中的起始地址和长度等有关信息。操作系统可以使用这些信息来回收分配给程序的空间，然后再将回收的空间用于其他应用程序。

为了实现多任务，操作系统需要在任何时候都能接收用户的新命令。当发出请求的程序所需要的所有资源都可用时，操作系统装载并开始执行该程序。

**多任务示例**

为了说明应用程序和操作系统之间的相互作用，我们来看一个多任务的例子。实现多任务的一种常见技术被称为时间片技术（time slice）。每个程序只运行一段很短的时间 $\tau$，称为时间片，然后另一个程序接着运行它的时间片，如此继续下去。周期 $\tau$ 由连续运行的硬件定时器确定，这个定时器每 $\tau$ 秒钟产生一个中断。

图 4-8 描述了在多任务环境下实现一些基本功能所需的例程。操作系统启动时要执行一个初始化例程，在图 4-8a 中称为 OSINIT。此外，初始化例程还对存储器中的中断向量区域进行设置。写入到向量区域的值对应于不同中断的中断服务程序起始地址。例如，OSINIT 将 [146]

SCHEDULER 例程的起始地址装入定时器中断的中断向量中。因此，在每个时间片结束时，定时器中断将执行该例程。

```
OSINIT        设置中断向量
                定时器中断 ← SCHEDULER
                软件中断 ← OSSERVICES
                I/O 中断 ← IODATA
                .
                .
                .
OSSERVICES    检查堆栈或处理器寄存器以确定所请求的操作

              调用适当的例程
SCHEDULER     保存当前运行进程的程序状态
              选择另一个就绪进程
              恢复新选择进程以前保存的程序状态
              中断返回
```

a）操作系统初始化，服务程序和调度程序

```
IOINIT        设置请求进程的状态为阻塞状态
              初始化内存缓冲区地址指针和计数器
              调用设备驱动来初始化设备，并在设备接口中允许中断

              从子程序返回

IODATA        轮询设备，确认中断源
              调用适当的驱动程序
              如果 END=1，则设置 I/O 阻塞进程的状态为就绪状态
              中断返回
```

b）I/O 例程

```
KBDINIT       允许中断
              从子程序返回

KBDDATA       检查设备状态
              如果准备就绪使传输字符

              如果字符 =CR，则（设置 END=1；禁止中断），否则设置 END=0

              Return from subroutine：从子程序返回
```

c）键盘驱动程序

图 4-8 操作系统例程示例

将程序与所有描述其当前执行状态的信息视为一个整体，称为进程 (process)。一个进程可以处于以下三种状态之一：运行、就绪和阻塞。运行状态指的是程序当前正在被执行。如果程序已经准备好并且在等待被选定执行，那么此时进程就处于就绪状态。第三种状态，阻塞，指的是程序因为一些原因而没有准备好开始执行。比如，它可能正在等待之前所请求的 I/O 操作的完成。

假设程序 A 在一个给定的时间片内处于运行状态。在该时间片结束时，定时器中断这个程序的运行并开始执行 SCHEDULER。SCHEDULER 是一个操作系统例程，它决定哪个用户程序在下个时间片运行。该例程首先保存以后继续执行程序 A 所需要的全部信息。这些保存的信息包括程序计数器和处理器状态寄存器等一些寄存器的内容。必须保存寄存器的内容是因为中断发生时它们可能包含一些正在进行的计算的中间结果。程序计数器指向以后可以继续执行的位置。处理器状态寄存器反映了当前的程序状态。

然后，SCHEDULER 选择执行另一程序 B，程序 B 在早些时候被暂停并且正处于就绪状态。SCHEDULER 将程序 B 暂停时保存的所有信息恢复，包括程序计数器和状态寄存器的内

容，然后执行一条中断返回指令。于是，程序B就重新开始执行τ秒钟，在这个时间片结束时定时器再次产生一个中断，并发生到另一个就绪进程的上下文切换（context switch）。

假设程序A当前正在执行，并且需要从键盘读入一行字符。它并不亲自执行这个操作，而是向操作系统请求I/O服务。它使用堆栈或处理器寄存器向操作系统传递信息来描述请求的操作、I/O设备和程序数据区域中用来存放从键盘输入字符的缓冲区地址。然后它发出一个软件中断，相应的中断向量指向图4-8a中的OSSERVICES例程。OSSERVICES例程检查堆栈或寄存器中的信息并调用适当的操作系统例程来启动请求的操作。在此例中，它调用图4-8b中的IOINIT例程，它是一个负责启动I/O操作的通用例程。

当进行I/O操作时，请求该操作的程序不能继续执行。因此，IOINIT例程将与程序A相关联的进程设置成阻塞状态。IOINIT例程执行I/O操作需要的所有准备工作，如初始化缓冲指针和字节计数，然后调用一个例程对请求I/O操作的特定设备进行初始化。

设计操作系统时通常将属于特定I/O设备的所有软件封装成一个独立的模块，称为设备驱动程序（device driver）。这样的模块可以很容易地添加到操作系统中或从操作系统中删除。我们假设键盘的设备驱动程序由两个例程组成：KBDINIT和KBDDATA，如图4-8c所示。IOINIT例程调用KBDINIT，KBDINIT用来执行设备或其接口电路所需要的初始化操作。同时KBDINIT还在接口电路中设置控制寄存器的相应位来允许中断，然后返回到IOINIT，IOINIT再返回到OSSERVICES。至此键盘接口已准备好参与数据传输操作了。每当一个键被按下时，它就产生一个中断请求。

147
|
148

返回到OSSERVICES后，SCHEDULER例程选择另一个用户程序运行。当然，调度例程不会选择程序A，因为它已经请求了一个I/O操作，现正处于阻塞状态。而是会选择程序B或其他处于就绪状态的程序。中断返回指令不仅能使被选定的用户程序开始执行，同时还因为将处理器状态寄存器恢复为原来保存的内容而重新允许处理器中断。这样，由键盘接口产生的中断请求将被接受，该中断的中断向量指向一个称为IODATA的操作系统例程。因为可能会有几台设备请求中断，所以IODATA首先测试这些设备，以识别出请求服务的设备，然后调用适当的设备驱动程序来为该请求服务。在我们的例子中，所调用的驱动程序是KBDDATA，它将传输一个字符的数据。如果是回车符，它还将END标志置为1来通知IODATA所请求的I/O操作已经完成。同时，IODATA例程将进程A的状态由阻塞变为就绪，从而调度例程便可以在将来的某个时间片中选择进程A执行。

## 4.10 结束语

软件是决定一台计算机的通用性和实用性的关键因素。实用程序使得用户可以创建、执行和调试应用软件。程序员可以通过编译器、汇编程序和连接程序来灵活地组合高级语言源文件、汇编语言源文件和库文件，以生成目标程序。必要时，还可以在高级语言源文件中嵌入汇编语言指令。

操作系统软件大大增强了计算机的功能，它可以管理和协调所有的活动。操作系统的多任务处理使得多个应用程序可以并行执行不同的活动，因而能够充分地利用计算机。

## 习题

[E] 4.1 编写一个C程序实现习题3.2中描述的任务。

[M] 4.2 编写一个C程序实现习题3.9中描述的任务。

[M] 4.3 编写一个C程序实现习题3.13中描述的任务。

149 [D] 4.4 编写一个 C 程序实现习题 3.15 中描述的任务。

[D] 4.5 编写一个 C 程序实现习题 3.17 中描述的任务。

[D] 4.6 编写一个 C 程序实现习题 3.17 中描述的任务，使用与定时器相关的中断服务程序。

[D] 4.7 假定处理器的指令集包含指令

MultiplyAccumulate Ri, Rj, Rk

它使用处理器寄存器 Ri，Rj 和 Rk 来执行 Ri ← [Ri] + [Rj] × [Rk] 操作。这条指令在 2.12.1 节中描述过。

假定编译器在生成汇编语言输出的时候不使用这条指令，且在 C 程序中定义了 X、Y 和 Z 三个全局变量。用 C 语言编写一个 mult_acc_XYZ 函数，使用 MultiplyAccumulate 指令来计算 X = X + Y × Z。注意，该函数和调用程序经过编译器编译后生成的汇编语言指令可能会使用处理器寄存器来保存数据。

[D] 4.8 4.9.2 节讨论了图 4-7 所示的一系列程序的输入和输出步骤是如何重叠执行以减少总的执行时间的。设 6 个操作系统例程中每一个的执行时间是 1 个单位的时间，每个磁盘操作需要 3 个单位的时间，打印需要 3 个单位的时间，每个程序的执行需要 2 个单位的时间。计算这一系列程序的最佳重叠时间与非重叠时间的比率。忽略启动和结束的时间。

[D] 4.9 4.9.2 节指明程序计算可与输入或输出操作或两者重叠起来。忽略执行操作系统例程所需的相对较短的运行时间，为完成一系列程序的执行，最佳重叠时间与非重叠时间的比率是多少？其中每个程序的输入、计算和输出活动都大致均衡。

[M] 4.10 在 4.9.3 节对进程三种状态的讨论中，我们描述了从就绪到运行、就绪到阻塞、阻塞到就绪的状态转换。这些状态间还有哪些直接的转换是可能的？哪些是不可能的？请简述原因。

## 参考文献

1. A. Silberschatz, P. B. Gavin, and G. Gagne, *Operating System Concepts*, 8th ed., John Wiley and Sons,
150 Hoboken, New Jersey, 2008.

# 基本处理部件

## 本章目标

在本章中你将学习以下内容：

- 处理器如何执行指令
- 处理器的功能部件以及这些部件如何互连
- 用于产生控制信号的硬件
- 微程序控制

[151]

在本章中我们将主要讨论处理部件，该部件的主要功能是执行机器语言指令以及协调计算机中其他部件的活动。我们将查看其内部结构，并说明它是如何完成指令的读取、译码和执行的。该处理部件通常称为中央处理器（CPU）。因为现在的计算机通常包含多个处理部件，所以跟过去相比，“中央”一词在今天就有些欠妥了。我们将在本章的讨论中使用处理器（processor）一词。

随着技术的不断发展以及人们对计算机性能要求的不断提高，处理器的结构多年来也在不断地发生变化。为了实现更高的性能，一个明智的做法就是使处理器的各种功能部件尽可能地并行操作。这种处理器一般都含有流水线（pipeline）结构，这使得处理器可以在前一条指令执行结束之前就开始执行下一条指令。另外一种方法称为超标量（superscalar）操作，可以同时读取并开始执行多条指令。我们将在第6章中讨论流水线以及超标量方法。本章将主要对各种处理器都通用的基本思想进行讨论。

## 5.1 一些基本概念

一个典型的计算任务是由构成程序的机器语言指令序列所指定的一系列操作组成的。处理器每次取出一条指令并执行指定的操作。除非遇到转移指令或跳转指令，否则指令都从连续的存储单元读取。处理器使用程序计数器（Program Counter，PC）来跟踪将要取出和执行的下一条指令的地址。取出指令以后，PC的内容将被更新以指向序列中的下一条指令。而转移指令则可能会将一个不同的值装入PC中。

当一条指令被取出时，它被处理器的控制电路放置在指令寄存器（Instruction Register，IR）中，指令就是在IR中被解释或译码的。IR将一直保存该指令直到它执行完毕。

例如，一台32位的计算机，它的每条指令都被包含在存储器的一个字中，就像RISC风格的指令集体系结构那样。为了执行一条指令，处理器必须执行以下步骤：

1）取出PC指向的存储单元的内容。该单元中的内容是将要执行的指令，因此它们被装入IR中。这个动作使用寄存器传输符号形式可以表示为：

$$IR \leftarrow [[PC]]$$

2）递增PC的值使其指向下一条指令。假设存储器是按字节寻址的，那么PC就增加4，即

$$PC \leftarrow [PC] + 4$$

3）执行IR中指令所指定的操作。

[152]

取出一条指令并将其装入IR通常称为取指令阶段（instruction fetch phase），执行指令所指定的操作称为指令执行阶段（instruction execution phase）。

除了少数例外情况，指令所指定的操作都可以通过执行以下一个或多个动作来实现：

- 读取给定存储单元的内容，并将其装入处理器寄存器中。
- 从一个或多个处理器寄存器中读取数据。
- 执行算术或逻辑运算，并将运算结果放到处理器寄存器中。
- 将处理器寄存器中的数据保存到指定的存储单元中。

执行这些动作所需的硬件组件如图 5-1 所示。处理器通过“处理器－存储器”接口与存储器进行通信，该接口在读和写操作期间从存储器接收数据或向存储器传送数据。每条指令取出以后，指令地址发生器都会更新 PC 的内容。寄存器文件是一个存储器部件，它的存储单元被组织成为处理器的通用寄存器。在执行过程中，算
153 术或逻辑运算指令所指定的寄存器的内容被发送到算术逻辑部件（ALU）。由 ALU 执行所需的计算。计算结果则存储在寄存器文件中的一个寄存器中。

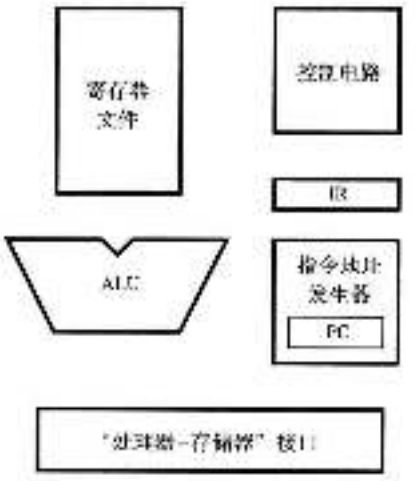


图 5-1 处理器的主要硬件组件

在我们深入研究这些部件以及它们之间的相互关系之前，了解一下数据处理系统的总体结构是很有帮助的。

**数据处理硬件**

一个典型的计算会对存储在寄存器中的数据进行操作。组合电路（如加法器）对这些数据进行处理，并将结果放到一个寄存器中。图 5-2 说明了这种结构。这里，我们使用一个时钟信号来控制数据传输的时序。寄存器由边沿触发的触发器构成。新数据就是在时钟的工作沿被装入的。在本章中，我们假定时钟的上升沿为工作沿。时钟周期是指两个连续的上升沿之间的时间，它必须足够长以确保组合电路产生正确的结果。

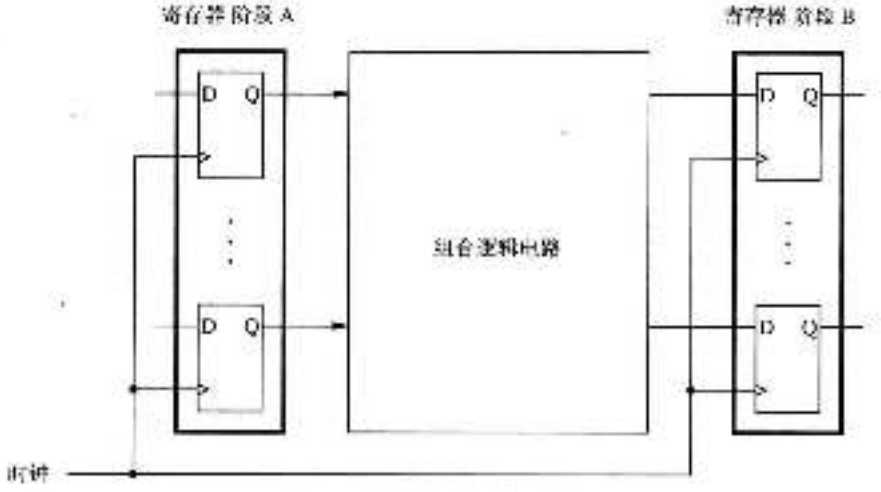


图 5-2 数据处理的基本结构

图 5-2 中的组合模块所执行的操作可能会相当复杂。通常，我们可以把这个操作分解成几个简单的步骤，其中，每个步骤都是由原始电路的一个子电路去完成的。然后，这些子电路可以级联起来形成一个多阶段结构，如图 5-3 所示。那么，如果使用 $n$ 个阶段的结构，则完成一个操作需要 $n$ 个时钟周期。由于这些组合子电路比原始电路小得多，它们可以用较少的时间完

成其操作，因此可以使用更短的时钟周期。多阶段结构的一个主要优势在于它适合流水线操作，这将在第 6 章中讨论。这种结构特别适用于实现 RISC 风格指令集的处理器。在本章的其余部分，我们将主要讨论使用这种多阶段结构的处理器。在 5.7 节中，我们还将介绍一种适合于 CISC 风格处理器的更加传统的方法。

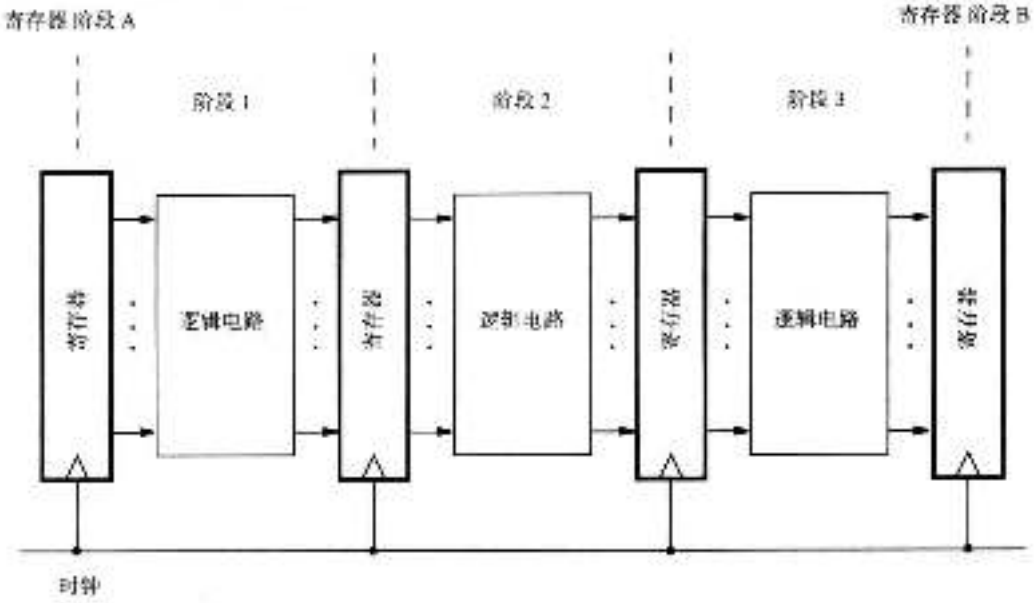



图 5.3　具有多个阶段的硬件结构

## 5.2　指令的执行

现在，让我们来看看一下取出及执行指令时所涉及的动作。我们将用几条有代表性的 RISC 风格的指令来具体说明这些动作。

### 5.2.1　Load 指令

考虑下面的指令：

Load　R5, X(R7)

它使用变址寻址方式来将存储单元 X + [R7] 中一个字长的数据装入寄存器 R5 中。执行这条指令包括以下几个动作：

- 从存储器中取出指令。
- 递增程序计数器的值。
- 对指令进行译码以确定将要执行的操作。
- 读取寄存器 R7。
- 将立即值 X 与 R7 的内容相加。
- 将 X + [R7] 的结果作为源操作数的有效地址，并读取存储器中相应单元的内容。
- 将从存储器中读取的数据装入目的寄存器 R5 中。

154
|
155

根据硬件的组织方式，上述动作中有一些是可以在同一时间执行的。在接下来的讨论中，我们假设处理器有 5 个硬件阶段。这是 RISC 风格的处理器常用的结构。每条指令的执行分为 5 步，这样，每一步可由一个硬件阶段完成。在这种情况下，上述 Load 指令的读取及执行就

可以按如下步骤完成：

1）取出指令并递增程序计数器的值。

2）对指令进行译码并从寄存器文件中读取寄存器 R7 的内容。

3）计算有效地址。

4）读取存储器中的源操作数。

5）将操作数装入目的寄存器 R5 中。

### 5.2.2 算术及逻辑运算指令

涉及算术或逻辑运算的指令可以使用类似的步骤执行。它们与 Load 指令的不同之处在于以下两个方面：

- 有两个源寄存器，或者有一个源寄存器以及一个立即源操作数。
- 不需要访问存储器操作数。

这种类型的典型指令是：

Add R3, R4, R5

完成该指令需要以下几个步骤：

1）取出指令并递增程序计数器的值。

2）对指令进行译码并读取源寄存器 R4 和 R5 的内容。

3）计算和：[R4] + [R5]。

4）将计算结果装入目的寄存器 R3 中。

由于 Add 指令并不需要访问存储器中的操作数，因此可以在四个步骤内完成，而不像 Load 指令那样需要五个步骤。然而，在下一章中我们将会看到，对尽可能多的指令使用相同的多阶段处理硬件是非常有利的。要做到这一点，我们需要使每一条指令的执行都花费相同的步骤数。为此，以 Load 指令的执行步骤为模板，需要将 Add 指令的执行扩展为五个步骤。由于 Add 指令不需要访问存储器操作数，所以我们可以在步骤 3 与步骤 4 之间插入一个不发生任何动作的空步骤。这样，Add 指令就将按如下步骤执行：

1）取出指令并递增程序计数器的值。

2）对指令进行译码并读取源寄存器 R4 和 R5 的内容。

3）计算和：[R4] + [R5]。

156 4）空操作。

5）将计算结果装入目的寄存器 R3 中。

如果指令使用一个立即操作数，如指令

Add R3, R4, #1000

在指令中给出了立即值。一旦该指令被装入 IR 中，立即值就可以用于加法运算中了。这样就同样可以使用上述五步动作，其中，步骤 2 和步骤 3 修改为：

2）对指令进行译码并读取寄存器 R4 的内容。

3）计算和：[R4] + 1000。

### 5.2.3 Store 指令

用于 Load 指令以及 Add 指令的五步动作也同样适用于 Store 指令，只是最后一步将结果装入目的寄存器是不需要的。负责这一步的硬件阶段不执行任何动作。例如，指令

Store R6, X(R8)

将寄存器 R6 的内容存储到存储单元 X + [R8] 中。其实现步骤如下：

1）取出指令并递增程序计数器的值。

2）对指令进行译码并读取寄存器 R6 和 R8 的内容。

3）计算有效地址：X + [R8]。

4）将寄存器 R6 的内容存入存储单元 X + [R8] 中。

5）空操作。

在第 2 步读取寄存器 R8 的内容之后，第 3 步中使用指令寄存器 IR 中的立即值 X 计算存储器地址。在第 4 步中，R6 的内容被发送到存储器中存储。第 5 步不执行任何动作。

总之，图 5-4 所示的五步动作序列对 RISC 风格指令集中的所有指令都适用。正如我们在第 2 章中所提到的，RISC 风格的指令为一个字长，而且只有 Load 指令和 Store 指令才会访问存储器中的操作数。其他执行计算的指令所使用的数据不是存储在通用寄存器中，就是在指令中以立即数的形式给出。

| 步骤 | 动作 |
|---|---|
| 1 | 取出指令并递增程序计数器的值 |
| 2 | 对指令进行译码并从寄存器文件中读取寄存器的内容 |
| 3 | 执行 ALU 运算 |
| 4 | 如果指令包含存储器操作数，则对存储器数据进行读或写操作 |
| 5 | 如果需要，将结果写入目的寄存器 |

图 5-4　取出并执行指令的五步动作序列

上述这五步动作序列适用于所有的 Load 和 Store 指令，这是因为这些指令中使用的寻址方式都是变址方式的特殊情况。大多数 RISC 风格的处理器都会提供一个通用寄存器，通常是 R0，其中总是包含数值 0。当把 R0 用作变址寄存器时，操作数的有效地址就是立即值 X，这就相当于绝对寻址方式。另外，如果把偏移量 X 设为 0，那么有效地址就是变址寄存器 Ri 中的内容，这就相当于间接寻址方式。因此，只需要实现一种寻址方式，即变址方式。这样，就显著地简化了处理器的硬件。选择 R0 为变址寄存器或者将 X 设为 0 的任务是由汇编程序或编译器来完成的。这与 RISC 的理念是一致的，即以更高的编译器复杂度和更长的编译时间为代价来换取硬件的简单和快速。由于程序的执行往往要比程序的编译频繁得多，所以这样做的结果就是节省了在计算机上执行各种任务所需的总时间。 157

## 5.3　硬件组件

上述的讨论表明，我们可以使用图 5-4 中的五步动作序列来执行 RISC 风格处理器的所有指令。因此，可以把处理器的硬件组织成五个阶段，每个阶段负责执行一个步骤所需的动作。现在我们将研究图 5-1 中的组件，并解释它们是如何被组织成图 5-3 的多阶段结构的。

### 5.3.1　寄存器文件

寄存器文件是一个小型而快速的存储区。通常情况下，通用寄存器是以寄存器文件的方式实现的。寄存器文件包含了一组存储元件，以及可以使数据被读出或写入任何寄存器的接入电路。接入电路的设计使其可以同时读取两个不同的寄存器，并令它们的内容出现在两个独立的输出端 A 和 B 上。寄存器文件有两个地址输入端，可以选择两个寄存器进行读取。这些输入端与 IR 中指定源寄存器的字段相连接，以便读取所需的寄存器。寄存器文件还有一个数据输入端 C 以及一个对应的地址输入端，这个地址输入端用来选择将被写入数据的寄存器，并与 IR 中指定目的寄存器的字段相连接。

通常，我们把任何存储器部件的输入和输出称为输入和输出端口（port）。一个拥有两个
158 输出端口的存储器部件被称为是双端口（dual-ported）的。图5-5显示了实现双端口寄存器文件的两种方式。一种方式是只使用一组寄存器，利用双重的数据通路以及接入电路，实现在同一时间读取两个寄存器的内容。另一种方式则是用两个存储区，每个存储区包含寄存器文件的一个副本。每当要向一个寄存器写入数据时，就将数据写入该寄存器的两个副本中。因此，这两个文件的内容是完全相同的。当一条指令需要读取两个寄存器中的数据时，就可以在每个文件中访问一个寄存器。实际上，这两个寄存器文件一起充当了一个单独的双端口寄存器文件。

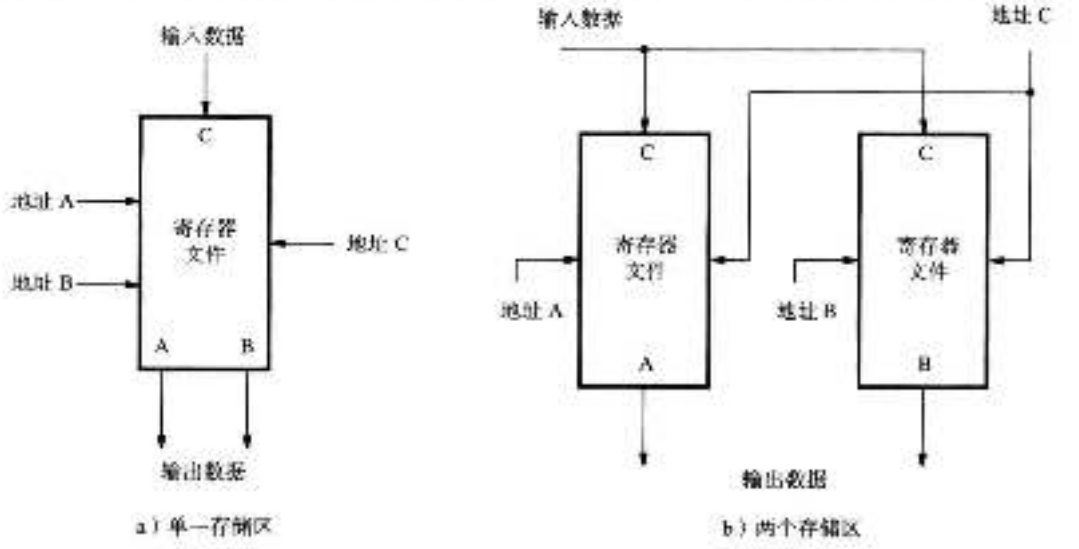



图5-5 双端口寄存器文件的两种实现方式

## 5.3.2 ALU

算术逻辑部件用于操纵数据。它可以执行加法和减法等算术运算，也可以执行AND（与）、OR（或）和XOR（异或）等逻辑运算。从概念上讲，寄存器文件和ALU可以按图5-6所示的方式连接起来。当要执行一条算术或逻辑运算指令
159 时，就从寄存器文件中读出指令中所指定的两个
160 寄存器的内容，并使其出现在输出端A和B上。输出端A与ALU的第一个输入端InA直接相连，输出端B则连接到多路复用器MuxB上。多路复用器选择寄存器文件的输出端B或者IR中的立即值，并将其连接到ALU的第二个输入端InB上。ALU的输出端与寄存器文件的数据输入端C相连，以便可以将计算结果装入目的寄存器中。

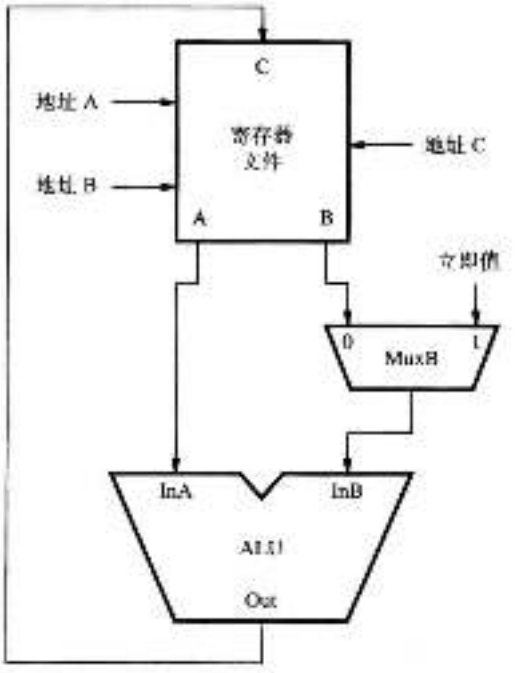



图5-6 计算所需部件的概念视图

## 5.3.3 数据通路

指令处理包括两个阶段：取指令阶段和执行

阶段。所以把处理器硬件划分为两个相应的部分会很方便。其中一个部分负责取指令，另一个部分则负责执行指令。取指令的部分同时也负责对指令进行译码，并负责为执行阶段将发生的相应动作生成控制信号。执行部分读取指令中指定的数据操作数、执行所需的计算，并存储结果。

现在，我们需要将硬件组织成一个类似于图 5-3 中的多阶段结构，其中的五个阶段分别与图 5-4 中的五个步骤相对应。图 5-7 给出了一种可能的结构。这五个阶段中每个阶段所执行的动作都要在一个时钟周期内完成。硬件阶段 1 在第 1 步中取出一条指令并将其放入 IR 中。在第 2 步中，该指令被译码，并读取其源寄存器的内容。IR 中的信息被用来为后续的步骤生成控制信号。因此，IR 必须一直保存该指令直到它执行完毕。

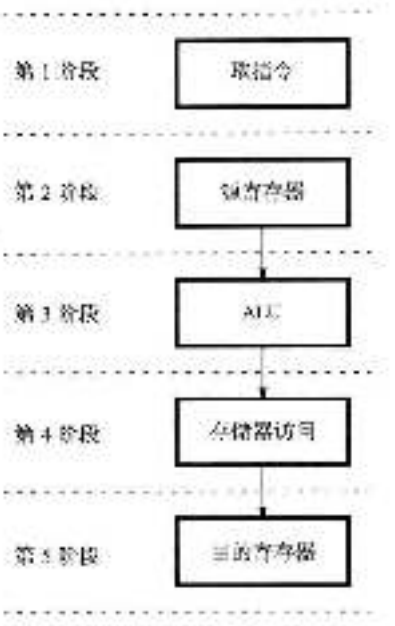


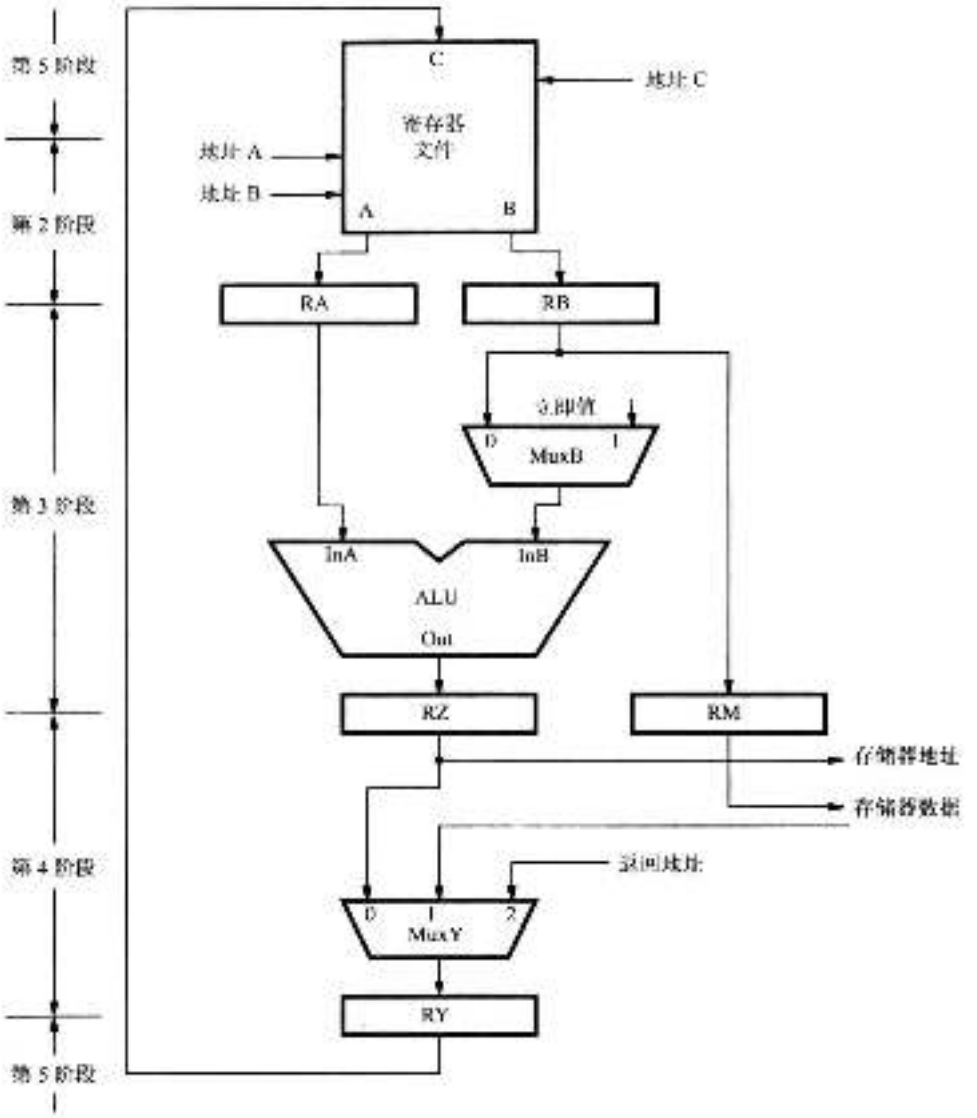
图 5-7　一个五阶段的结构

我们还需要在相邻阶段之间插入一些寄存器。这些段间寄存器保存相应阶段产生的结果，这样，在下一个时钟周期中，它们可被用作下一阶段的输入。这就引出了图 5-8 的结构。该图中的硬件通常被称为数据通路（datapath）。它对应于图 5-7 中的第 2 到第 5 阶段。从寄存器文件中读取的数据被放置到寄存器 RA 和 RB 中。寄存器 RA 为 ALU 的输入端 InA 提供数据。多路复用器 MuxB 则将寄存器 RB 中的内容或者 IR 中的立即值转发给 ALU 的第二输入端 InB。ALU 构成第 3 阶段，其计算结果被放置到寄存器 RZ 中。

回想一下之前的计算型指令，如 Add 指令，在第 4 步中没有发生任何处理动作。在这一步中，图 5-8 中的多路复用器 MuxY 选择寄存器 RZ 以将计算结果传送给 RY。在第 5 步中，RY 中的内容被传送给寄存器文件，并被装入目的寄存器中。为此，寄存器文件同处于第 2 和第 5 阶段。说它是第 2 阶段的一部分是因为它包含了源寄存器，而说它是第 5 阶段的一部分则是因为它包含了目的寄存器。

对于 Load 和 Store 指令而言，ALU 在第 3 步中计算出存储器操作数的有效地址并将其装入寄存器 RZ 中。在第 4 阶段，该地址从 RZ 发送至存储器。对于 Load 指令，多路复用器 MuxY 选择从存储器中读取的数据并将其放置在寄存器 RY 中，以便在下一个时钟周期将其传送给寄存器文件。对于 Store 指令来说，数据是在第 2 阶段从寄存器文件中读出的并被放置到寄存器 RB 中。由于存储器访问是在第 4 阶段进行的，所以需要在多阶段结构中增加一个段间寄存器以保持数据流的正确性。引入寄存器 RM 就是出于这个目的。在第 3 步中，将需要存储的数据从 RB 移到 RM 中，继而在第 4 步中再将其存入存储器中。在这种情况下，第 5 步无需采取任何动作。

2.7 节中介绍的子程序调用指令把返回地址保存在一个通用寄存器中，为了便于引用，我们称之为 LINK。类似地，3.2 节中描述的中断处理也需要保存返回地址。假设我们为此使用了另一个通用寄存器 IRA。这两个动作都要求将程序计数器的内容发送到寄存器文件中。为此，多路复用器 MuxY 就有了第三个输入端，通过该输入端就可以将返回地址传送至寄存器 RY 中，然后再从 RY 发送到寄存器文件中。其中，返回地址是由指令地址发生器产生的，稍后我们将进行讲解。




161
/
163

图 5-8 处理器中的数据通路

### 5.3.4 取指令部分

图 5-9 给出了处理器中取指令部分的组织结构。在取指令时，访问存储器的地址来自于程序计数器 PC；而在访问指令操作数时，访问存储器的地址则来自于数据通路中的寄存器 RZ。多路复用器 MuxMA 会选择其中的一个并将其发送到“处理器—存储器”接口。PC 被包含在一个更大的模块——指令地址发生器中，该发生器会在取出每条指令后对 PC 的内容进行更新。从存储器中读取的指令被装入 IR 中，IR 会一直保存该指令直到其执行完毕并且下一条指令被取出。

控制电路可以通过检查 IR 的内容来生成控制处理器所有硬件所需的信号。被标记为立即数的模块也会用到 IR 的内容。正如第 2 章所述，一些指令中可能会包含立即值。16 位的立即值会被扩展为 32 位，然后，扩展值可以直接作为操作数使用，也可以用来计算操作数的有效

地址。对于一些指令，例如执行算术运算的指令，立即值是用符号扩展的；而对于其他的指令，例如逻辑运算指令，立即值则是用零来填充的。图 5-9 中的立即数模块负责生成扩展值，并将其转发给图 5-8 中的多路复用器 MuxB，以便在 ALU 计算中使用。同时它生成的扩展值也可用于计算转移指令的目标地址。

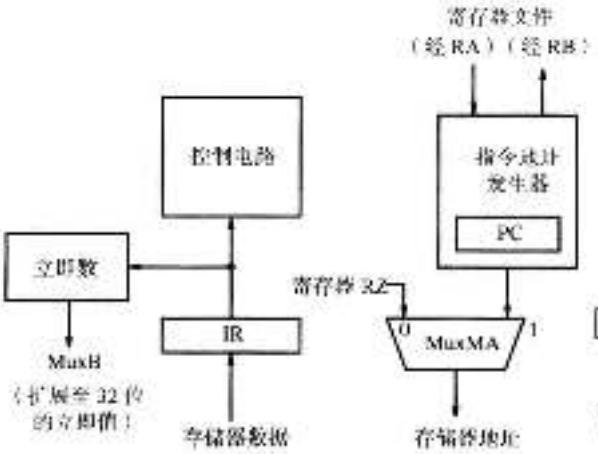



图 5-9　图 5-7 的取指令部分

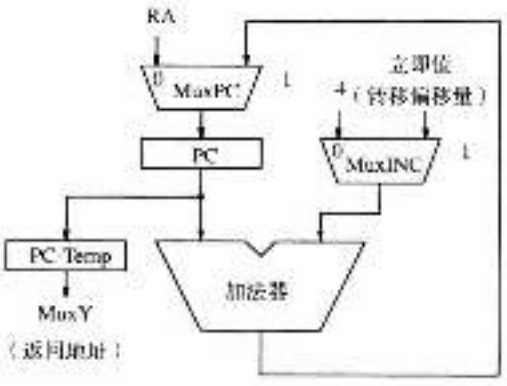



图 5-10　指令地址发生器

地址发生器的电路如图 5-10 所示。在程序的线性执行（无跳转）过程中，加法器用于将 PC 的值递增 4。当执行转移指令和子程序调用指令时，它也可以用于计算一个将要被装入 PC 中的新值。加法器的一个输入端与 PC 相连，另一个输入端则连到多路复用器 MuxINC 上，该多路复用器选择将常数 4 或者转移偏移量加到 PC 上。转移偏移量是由 IR 中的立即数字段给出的，并且被图 5-9 中的立即数模块符号扩展为 32 位。加法器的输出通过第二个多路复用器 MuxPC 传送到 PC 中，多路复用器 MuxPC 在加法器的输出和寄存器 RA 的输出之间进行选择，后者在执行了程序链接指令时用到。在保存了程序或中断的返回地址的过程中，我们需要用寄存器 PC-Temp 来临时存放 PC 的内容。

## 5.4　指令的读取和执行步骤

现在我们利用图 5-8 中的数据通路对取指令和执行指令的过程进行更深入的研究。再次考虑指令

Add　R3, R4, R5

图 5-11 给出了读取和执行该指令的步骤。假设我们用图 2-32（此处复制为图 5-12）中的格式对该指令进行编码。在从存储器中取出该指令并将其放入到 IR 后，源寄存器地址就在 $IR_{31\ldots27}$ 和 $IR_{26\ldots22}$ 字段中了。这两个字段与寄存器文件中端口 A 和 B 的地址输入端相连。因此，在第 2 步的末尾，可以去读取寄存器 R4 和 R5 的内容并将其分别装入寄存器 RA 和 RB 中。在下一步中，控制电路通过设置 MuxB 来选择输入端 0，从而将寄存器 RB 连接到 ALU 的输入端 InB 上。同时，这也使得 ALU 执行加法运算。由于寄存器 RA 与输入端 InA 相连接，所以 ALU 就会产生 [RA] + [RB] 的和，并在第 3 步末尾将结果装入寄存器 RZ 中。

164 / 165

在第 4 步中，多路复用器 MuxY 选择输入端 0，从而使得寄存器 RZ 的内容被传送到 RY 中。控制电路则将 Add 指令的目标地址字段 $IR_{21\ldots17}$ 连接到寄存器文件端口 C 的地址输入端上。在第 5 步中，控制电路向寄存器文件发送一个 Write 命令，使得寄存器 RY 的内容被写入到寄存器 R3 中。

| 步骤 | 动作 |
|---|---|
| 1 | 存储器地址 ← [PC]，读取存储器，IR ← 存储器数据，PC ← [PC] + 4 |
| 2 | 对指令进行译码，RA ← [R4]，RB ← [R5] |
| 3 | RZ ← [RA] + [RB] |
| 4 | RY ← [RZ] |
| 5 | R3 ← [RY] |

图 5-11 读取和执行指令 Add R3, R4, R5 所需的动作序列

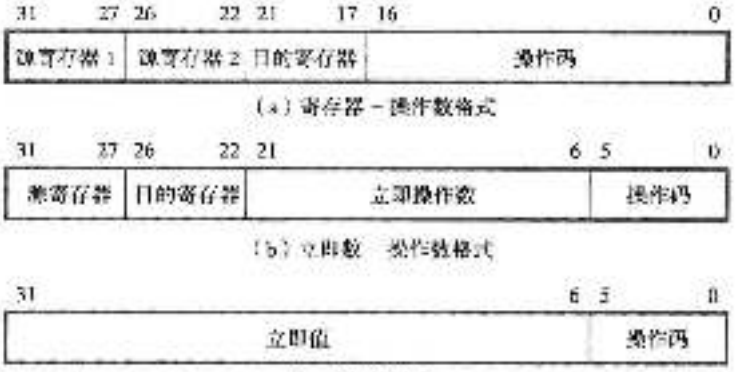



图 5-12 指令编码格式

| 步骤 | 动作 |
|---|---|
| 1 | 存储器地址 ← [PC]，读取存储器，IR ← 存储器数据，PC ← [PC] + 4 |
| 2 | 对指令进行译码，RA ← [R7] |
| 3 | RZ ← [RA] + 立即值 X |
| 4 | 存储器地址 ← [RZ]，读取存储器，RY ← 存储器数据 |
| 5 | R5 ← [RY] |

图 5-13 读取并执行指令 Load R5, X(R7) 所需的动作序列

| 步骤 | 动作 |
|---|---|
| 1 | 存储器地址 ← [PC]，读取存储器，IR ← 存储器数据，PC ← [PC] + 4 |
| 2 | 对指令进行译码，RA ← [R8]，RB ← [R6] |
| 3 | RZ ← [RA] + 立即值 X，RM ← [RB] |
| 4 | 存储器地址 ← [RZ]，读取存储器 ← [RM]，写入存储器 |
| 5 | 空操作 |

图 5-14 读取并执行指令 Store R6, X(R8) 所需的动作序列

Load 和 Store 指令都以类似的方式执行。在这种情况下，字段 $IR_{26-22}$ 给出了目的寄存器的地址。控制硬件把该字段连接到寄存器文件输入端 C 相对应的地址输入端。图 5-13 和图 5-14 给出了这些指令执行的步骤。这两个例子中，存储器地址都以变址方式指定，其中变址值 X 是在指令中以立即值的方式给出的。在第 3 步中，MuxB 选择 IR 中的立即数字段，由图 5-9 中的立即数模块进行适当扩展后，再与寄存器 RA 的内容相加，所得到的和就是操作数

的有效地址。

**一些需要注意的问题**

在上述的讨论中，我们假设存储器读写操作可以在一个时钟周期内完成。问题是，这个假设现实吗？一般而言，对计算机主存储器进行访问所需的时间要远远超过对寄存器文件中寄存器的内容进行读取的时间。然而，大多数现代处理器都使用高速缓存，我们将在第 8 章中详细讨论高速缓存。高速缓存比主存储器要快得多。它通常与处理器放在同一芯片上，这使得它可以达到与寄存器文件相当的速度。因此，当所需数据存放在高速缓存中时，存储器的读或写操作就可以在一个时钟周期内完成。当一个操作需要访问主存储器时，处理器必须等待该操作完成。我们将在 5.4.2 节中讨论如何处理速度较慢的存储器访问。

166 - 167

在第 2 步中我们还假设处理器会读取指令中的源寄存器，然而那时处理器仍然在对刚被装入 IR 中的指令的操作码进行译码。这两个操作可能在同一步中完成吗？在指令译码完成之前，控制硬件又如何知道要去读取哪个寄存器？其实这是可以做到的，因为在所有的指令中，源寄存器的地址都是由相同的位位置指定的。一旦指令被装入 IR 中，硬件就会读取由这些位位置中的地址所指定的寄存器。它们的内容在第 2 步末尾被装入寄存器 RA 和 RB 中。如果指令需要这些数据，就可以在第 3 步中使用它们。否则，后续的硬件阶段将会忽略这些数据。

需要注意的是，图 5-11、图 5-13 和图 5-14 描述的动作并没有说明在任何情况下都需要在第 2 步中读取两个寄存器。为了避免混淆，即使总是要读取两个寄存器，也只有图中所描述的特定指令所需要的寄存器才会被提及。

### 5.4.1　转移

在程序的线性执行过程中，指令是从存储器的连续字单元中取出的。每取出一条指令，处理器就将程序计数器 PC 递增 4 以指向下一个字。这种执行方式继续下去，直到转移指令或子程序调用指令将一个新的地址装入 PC 中。其中，子程序调用指令需要保存返回地址，以便于其返回到调用程序时使用。这一节，我们就来研究实现这些指令所需要的动作。I/O 设备以及软件中断指令引起的中断都以类似的方式处理。

转移指令指定了相对于 PC 的转移目标地址。在指令中，以立即值形式给出的转移偏移量被加到 PC 的当前内容中。表示该偏移量的位数比计算机的字长要少得多。这是因为指令中还需要空间来指定操作码以及转移条件。因此，一条转移指令能够到达的地址范围是有限的。

子程序调用指令可以到达更大的地址范围，这是因为这些指令中没有包含条件，这样就可以使用更多的位来指定目标地址。此外，大多数 RISC 风格的计算机中都有 Jump 和 Call 指令，它们使用一个通用寄存器来指定一个完整的 32 位地址。正如附录 B 到 E 中所介绍的处理器示例那样，每一台计算机的具体细节都有所不同。

**1. 转移指令**

图 5-15 给出了实现无条件转移指令的步骤序列。和之前一样，在第 1 步中取出指令并将 PC 的值递增。在第 2 步中，指令被译码之后，多路复用器 MuxINC 在第 3 步选择把 IR 中的转移偏移量加到 PC 中，从而得到用于取出下一条指令的地址。转移指令的执行在第 3 步完成。第 4 步和第 5 步则没有执行任何动作。

168

我们在 2.13 节中解释过，转移偏移量是转移目标与转移指令之后的存储单元之间的距离。其原因在图 5-15 中是显而易见的。在第 1 步中，PC 递增 4，此时取出了转移指令。然后，在第 3 步中，通过将转移偏移量与 PC 中更新后的内容相加来计算转移目标地址。

| 步骤 | 动作 |
| --- | --- |
| 1 | 存储器地址 ← [PC]，读取存储器，IR ← 存储器数据，PC ← [PC] + 4 |
| 2 | 对指令进行译码 |
| 3 | PC ← [PC] + 转移偏移量 |
| 4 | 空操作 |
| 5 | 空操作 |

图 5-15 读取并执行无条件转移指令所需的动作序列

我们可以很容易地修改图 5-15 所示的序列来实现条件转移指令。在那些不使用条件码标志的处理器中，转移指令会指定一个比较 - 测试操作来确定转移条件。例如，指令

Branch_if [R5]=[R6] LOOP

在寄存器 R5 和 R6 的内容相同时，会引起一个转移。在执行该指令时，会比较这两个寄存器中的内容，如果相等，则会转移到位置 LOOP 处。

图 5-16 显示了该指令是如何执行的。和之前一样，在第 2 步中读取寄存器 R5 和 R6 的内容，并在第 3 步中进行比较。其中，比较操作可以通过在 ALU 中进行减法运算 [R5] - [R6] 来完成。ALU 会生成相应的信号来指示减法运算的结果是正的、负的还是 0。ALU 还会生成信号来表示是否发生了算术溢出以及是否产生了进位。控制电路会检查这些信号以测试转移指令中给出的条件。在上面的例子中，它将检测减法运算的结果是否为零。如果等于零，就将转移目标地址装入 PC，用于读取下一条指令。否则，PC 的内容保持在第 1 步中递增后的值不变，并继续以线性方式执行。

| 步骤 | 动作 |
| --- | --- |
| 1 | 存储器地址 ← [PC]，读取存储器，IR ← 存储器数据，PC ← [PC] + 4 |
| 2 | 对指令进行译码，RA ← [R5]，RB ← [R6] |
| 3 | 比较 [RA] 和 [RB]，如果 [RA] = [RB]，则 PC ← [PC] + 转移偏移量 |
| 4 | 空操作 |
| 5 | 空操作 |

图 5-16 读取并执行指令 Branch_if_[R5]=[R6] LOOP 所需的动作序列

根据图 5-16 所示的步骤序列，比较寄存器的内容和对结果进行测试这两个动作都在第 3 步中完成。因此，时钟周期必须足够长，以保证这两个动作能依次完成。由于这个原因，比较操作理应尽快完成。ALU 中的减法运算很耗时，不过在这里我们也不需要进行减法运算。用一个简单而快速的比较器电路就可以比较寄存器 RA 和 RB 的内容并产生指示大于、等于或是小于等条件的条件信号。图 5-8 中并没有单独画出比较器，因为它可以是 ALU 模块的一部分。例 5.3 展示了如何设计一个比较器电路。

**2. 子程序调用指令**

子程序调用和返回的实现方式与转移指令相似。子程序的地址可能会使用指令中给出的立即值计算得到，也可能会由一个通用寄存器完全给出。图 5-17 给出了下面指令的动作序列：

Call_Register R9

该指令调用一个子程序，其地址在寄存器 R9 中。在第 2 步中，读取该寄存器的内容并将其装入 RA 中。在第 3 步中，多路复用器 MuxPC 选择其输入端 0，从而将寄存器 RA 中的数据装入 PC 中。

| 步骤 | 动作 |
| --- | --- |
| 1 | 存储器地址 ← [PC]，读取存储器，IR ← 存储器数据，PC ← [PC] + 4 |
| 2 | 对指令进行译码，RA ← [R9] |
| 3 | PC-Temp ← [PC]，PC ← [RA] |
| 4 | RY ← [PC-Temp] |
| 5 | 寄存器 LINK ← [RY] |

图 5-17　读取并执行指令 Call_Register R9 所需的动作序列

169
~
170

假设了程序的返回地址，即之前 PC 中的内容，被保存在寄存器文件的一个通用寄存器 LINK 中。在第 5 步中，数据被写入寄存器文件中。因此，不可能在第 3 步直接将返回地址发送给寄存器文件。为了在这个五阶段结构中保持数据流的正确性，处理器将返回地址保存在一个临时寄存器 PC-Temp 中。在第 4 步中，返回地址从 PC-Temp 传送到寄存器 RY 中，然后在第 5 步中再传送到寄存器 LINK 中。地址 LINK 被内置于控制电路中。

子程序返回指令将寄存器 LINK 中保存的值再传回到 PC 中。Return-from-subroutine 指令的编码使得寄存器 LINK 的地址出现在位 $IR_{3...27}$ 中。该字段与寄存器文件的地址 A 相连。因此，一旦指令被取出，就会去读取寄存器 LINK 并将其内容放到 RA 中，然后通过图 5-10 中的 MuxPC 将它们再从 RA 传送到 PC 中。Return-from-interrupt 指令以类似的方式进行处理，除了它是使用一个不同的寄存器来保存返回地址。

### 5.4.2　等待存储器

处理器–存储器接口电路的作用是控制处理器和存储器之间的数据传输。前面我们曾指出，现代处理器使用快速的片上高速缓存。大多数时候，存储器的读写操作所引用的指令或数据都可以在高速缓存中找到，此时，读写操作可以在一个时钟周期内完成。当所请求的信息不在高速缓存中并且不得不从主存储器中读取时，则需要花费几个时钟周期。此时，接口电路必须将这种情况通知给处理器的控制电路，以便将后续的执行步骤延迟直到存储器操作完成。

假设处理器–存储器接口电路产生一个叫做存储器功能完成（Memory Function Completed，MFC）的信号，它会在所请求的存储器读写操作完成时发出该信号。当处理器发出一个存储器读写请求时，处理器的控制电路会检测该信号，以确定它何时可以进入下一步。当所请求的数据在高速缓存中时，接口电路就会在发出存储器请求的时钟周期结束之前发出一个 MFC 信号。因此，指令继续执行而不会中断。如果需要访问主存储器，接口电路就延迟发送 MFC 信号直到该操作完成。在这种情况下，处理器的控制电路必须把执行步骤的持续时间延长为所需要的多个时钟周期，直到 MFC 信号发出。我们使用命令 Wait for MFC 来表示必须延长一个给定的执行步骤，如果有必要，一直延长到存储器操作完成。当接收到 MFC 信号时，表明该步骤所指定的动作已经完成，处理器继续执行序列中的下一步。

任何指令，在其执行序列的第 1 步都需要从存储器中读取指令。因此，必定会包含一条 Wait for MFC 指令，如下所示：

存储器地址← [PC]，读取存储器，Wait for MFC，
IR ← 存储器数据，PC ← [PC] + 4

171

图 5-13 以及图 5-14 中 Load 和 Store 指令的第 4 步也需要 Wait for MFC 指令。大多数时候，所请求的信息会在高速缓冲中找到，因此会很快产生 MFC 信号，这样该步骤可以在一个时钟周期内完成。当需要访问主存储器时，MFC 响应会被延迟，从而该步骤会被延长至多个时

钟周期。

## 5.5 控制信号

处理器硬件组件的操作是由控制信号控制的。这些信号决定多路复用器要选择哪一个输入端、ALU 要执行什么样的操作，等等。在这一节中，我们将讨论控制图 5-8 至图 5-10 中所示组件的操作所需要的信号。

为了便于理解，我们先回顾一下数据是如何流经数据通路的四个阶段的，如 5.3.3 节所述。在每个时钟周期中，发生在某个阶段的动作所得到的结果被存储在段间寄存器中，以便于在下一个时钟周期中被下一个阶段使用。由于数据在每个时钟周期都要从一个阶段传送到下一个阶段，所以段间寄存器总是处于启用状态。寄存器 RA、RB、RZ、RY、RM 以及 PC-Temp 都是这种情况。其他寄存器中的内容，即 PC、IR 和寄存器文件，在每个时钟周期内都必须保持不变。只有在一个特定的处理步骤中调用新数据时，新数据才会被装入这些寄存器中，这些寄存器也只有在那种情况下才处于启用状态。

多路复用器的作用是可以在任何给定的阶段选择将要操作的数据。例如图 5-8 中第 3 阶段的 MuxB，当指令中使用了一个立即源操作数时，它会选择 IR 中的立即数字段；当指令使用立即数作为偏移量计算存储器操作数的有效地址时，它也会选择 IR 中的立即数字段；否则，它会选择寄存器 RB。MuxB 选择的数据供 ALU 使用。仔细观察图 5-11、图 5-13 和图 5-14，我们发现 ALU 只在第 3 步中用到，因此 MuxB 的选择也只在这一步中才起作用。为了简化所需的控制电路，MuxB 可以在所有的执行步骤中保持相同的选择。对于 MuxY 也可以做同样的处理。然而，图 5-9 中的 MuxMA 在不同的执行步骤中必须改变其选择。在第 1 步中取出新的指令时，它选择 PC 作为存储器地址。而在 Load 和 Store 指令的第 4 步中，它选择寄存器 RZ，该寄存器中包含存储器操作数的有效地址。

所需的控制信号如图 5-18 ~图 5-20 所示。寄存器文件有三个 5 位的地址输入端，可以访问 32 个通用寄存器。其中两个输入端，地址 A 和地址 B，确定了将要读取的寄存器。它们与指令寄存器中的字段 $IR_{18-14}$ 和 $IR_{23-19}$ 相连。第三个地址输入端，地址 C，用于选择目的寄存器，端口 C 上的输入数据会被写入到该寄存器中。多路复用器 MuxC 用于选择地址 C 的来源。如图 5-12 所示，我们已经作出假设：三寄存器指令使用 $IR_{13-9}$ 而其他指令使用 $IR_{26-22}$ 来指定目的寄存器。多路复用器 MuxC 的第三个输入端是于程序链接指令中使用的链接寄存器的地址。只有当控制信号 RF_write 发出后，新数据才会被装入所选择的寄存器中。

172
/
173

多路复用器受信号的控制选择让哪个输入数据出现在多路复用器输出端。例如，当 B_select 等于 0 时，MuxB 选择将寄存器 RB 的内容传送到 ALU 的输入端 InB。需要注意的是，控制 MuxC 和 MuxY 的信号需要两个位，因为这两个多路复用器都需要从三个输入端中选择一个。

一个 $k$ 位的控制码 ALU_op 确定了 ALU 要执行的运算，该控制码可以表示 $2^k$ 个不同的运算，如 Add、Subtract、AND、OR 以及 XOR。当一条指令要求比较两个值的大小时，一个比较器会执行指定的比较运算，正如我们之前提到的那样。比较器将产生指示比较结果的条件信号。在执行条件转移指令时，控制电路会检查这些信号以确定转移条件是否成立。

处理器和存储器之间的接口以及与指令寄存器相关的控制信号如图 5-19 所示。MEM_read 和 MEM_write 这两个信号用于初始化存储器读或写操作。当请求的操作完成时，该接口发出 MFC 信号。指令寄存器通过控制信号 IR_enable 将一条新的指令装入寄存器。在取指令阶段，

该信号必须在 MFC 信号发出后才能被激活。

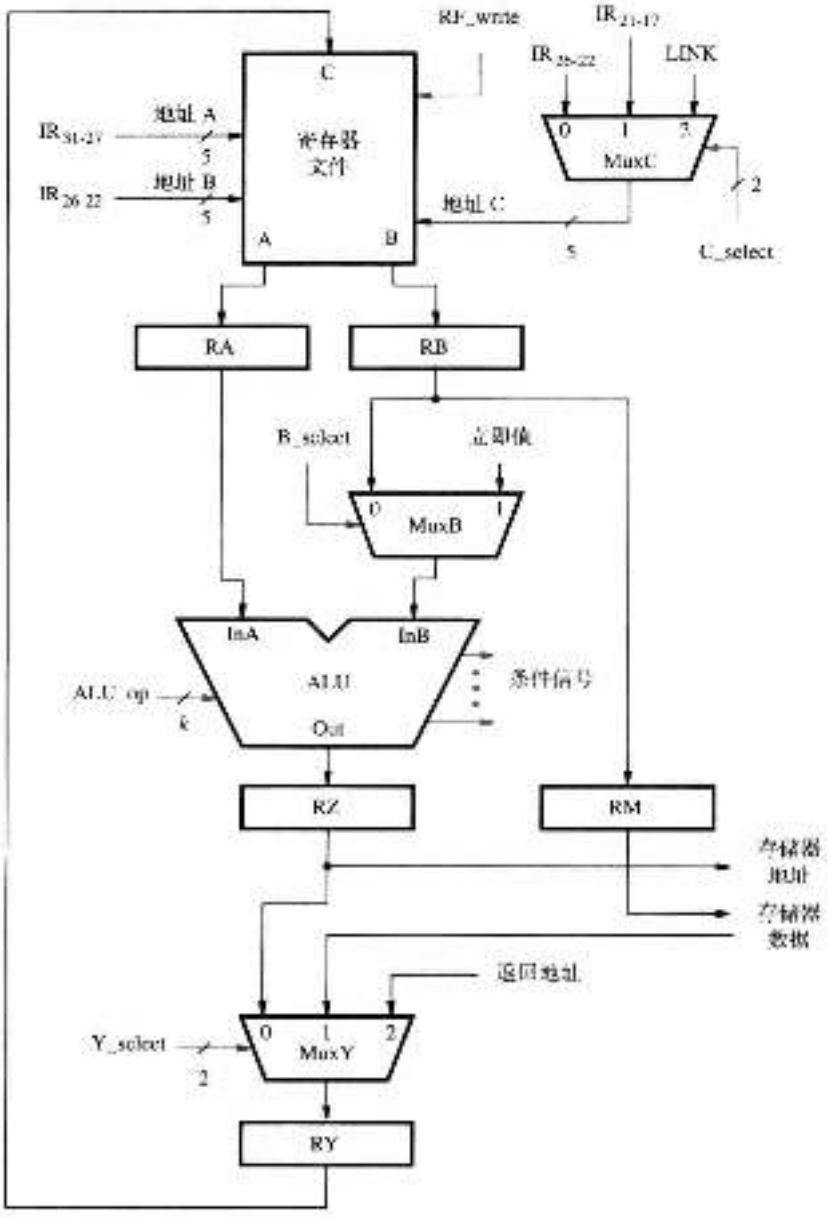



图 5-18　数据通路的控制信号

我们已经假设立即数模块可以处理三种可能的立即值形式：16 位符号扩展值，16 位无符号扩展值和 26 位以特殊方式处理的值（参见习题 5.14）。因此，其控制信号 Extend 包含了两位。

控制指令地址发生器操作的信号如图 5-20 所示。其中 INC_select 信号用来选择加到 PC 中的值是常数 4 还是指令中指定的转移偏移量。当控制信号 PC_enable 被激活时，PC_select 信号会选择已更新的地址或者寄存器 RA 的内容并将其装入 PC 中。

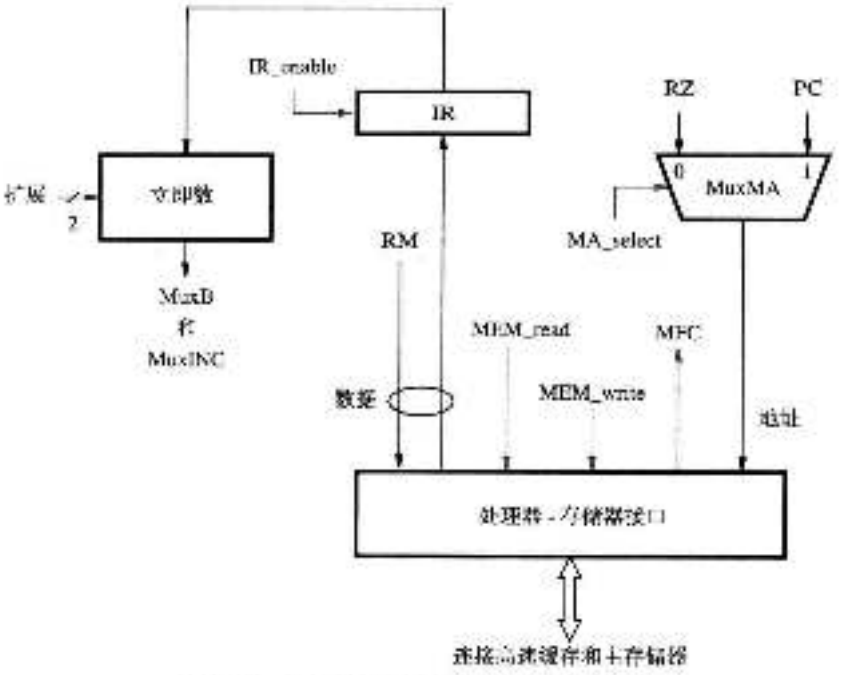



图 5-19 处理器-存储器接口和 IR 控制信号

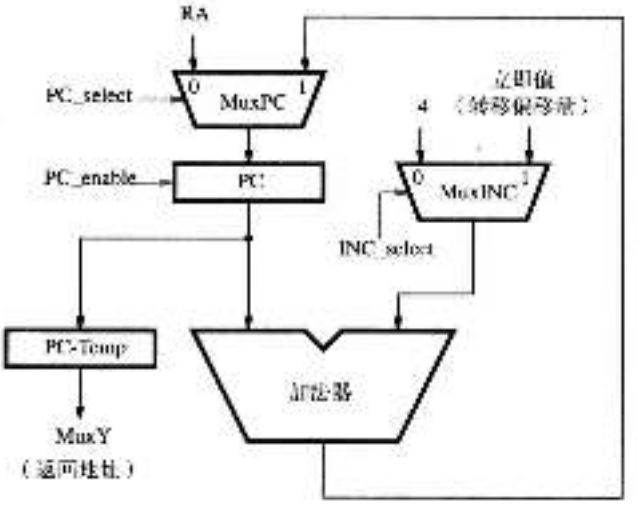



图 5-20 指令地址发生器的控制信号

## 5.6 硬件控制

前几节讨论了取指令和执行指令所需的动作。现在我们来研究处理器是如何产生所需的控制信号以使得这些动作能够在合适的时间以正确的顺序发生。有两种基本的方法：硬件控制和微程序控制。本节讨论硬件控制方法。

174
~
175

指令按一定的步骤序列执行，该序列中的每一步都需要在一个时钟周期内完成。因而可以使用一个步计数器来跟踪执行的进度。在每一步内可以执行多个动作，这具体取决于正在执

行的指令。在某些情况下，比如转移指令，要执行的动作取决于计算结果或者比较操作的结果。而外部信号（如中断请求）也有可能会影响到将要执行的动作。因此，控制信号的设置取决于：

- 步计数器的内容。
- 指令寄存器的内容。
- 计算结果或比较操作的结果。
- 外部输入信号，比如中断请求。

生成控制信号的电路可以按图 5-21 所示的方式组织。其中指令译码器负责解释操作码以及 IR 中的寻址方式信息，并将相应的 INS*i* 输出置为 1。在每个时钟周期中，步计数器输出端 T1 到 T5 中的一个被置为 1，以表明当前正在执行指令的那一步。由于所有的指令都在五个步骤内完成，所以我们可以使用一个模 5 计数器。控制信号发生器实际上是一个组合电路，它根据所有的输入来产生所需的控制信号。控制信号所需的设置信息由实现每条指令（由信号 INS1 到 INS*m* 表示）的动作序列确定。

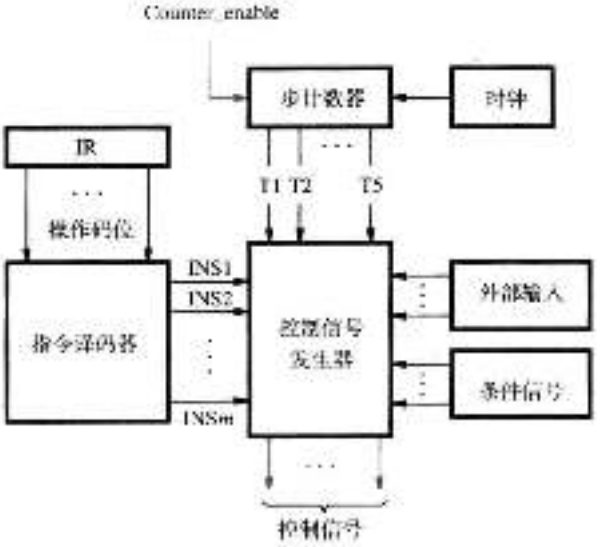



图 5-21　控制信号的生成

作为一个例子，考虑指令执行过程的第 1 步，在该步中，一条新指令从存储器中取出。我们通过发出信号 T1 来识别这一步。在该时钟周期内，图 5-19 中的 MA_select 信号被置为 1 以选择 PC 作为存储器地址，MEM_read 信号被激活以初始化存储器读操作。当存储器发出响应信号 MFC 时，IR_enable 信号被激活以将从存储器中取出的数据装入 IR 中。同时，图 5-20 中的 INC_select 被置为 0、PC_select 被置为 1，以使 PC 的值增加 4。最后，激活 PC_enable 信号，以将新的 PC 值在标志 T1 步结束的时钟上升沿装入 PC 中。

### 5.6.1　数据通路控制信号

处理数据的指令包括 Load、Store 以及所有计算型的指令。它们通过处理器的数据通路进行各种数据传输和操纵类作，该数据通路的控制信号如图 5-18 和图 5-19 所示。一旦有指令装入 IR 中，指令译码器便解释其内容并确定所需的动作。同时，读取源寄存器，并将它们的内容放到寄存器文件的两个输出端 A 和 B 上。前面我们提到，段间寄存器 RA、RB、RZ、RM 和 RY 一直都处于启用状态。这就意味着在每个时钟信号的工作沿，数据会自动从数据通路的一个阶段流向下一个阶段。

我们通过检查每条指令的每一个执行步中所发生的动作，来决定各种控制信号所需的设置。例如，在执行一条将数据写入寄存器文件的指令时，会在 T5 步中把 RF_write 信号置为 1。该信号可以通过下列逻辑表达式产生：

$$RF\_write = T5 \cdot (ALU + Load + Call)$$

其中 ALU 表示所有执行算术或逻辑运算的指令，Load 表示所有的 Load 指令，Call 表示所有

176

的子程序调用以及软件中断指令。RF_write 信号是指令和时序信号的函数。但是，正如我们前面提到的，一些多路复用器的设置并不需要在各个时序步中做任何改变。这种情况下，多路复用器的选择信号就可以只用指令的函数来实现。例如：

$$B\_select = Immediate$$

其中，Immediate 表示所有在 IR 中使用立即值的指令。我们鼓励读者研究其他控制信号，并根据各种指令的执行步骤得出恰当的逻辑表达式。

### 5.6.2 存储器延迟的处理

随着步计数器的递增，时序信号 T1 到 T5 会按顺序发出。大多数时候，步计数器在每个时钟周期结束时递增。然而，如果在某一步中发出了 MEM read 和 MEM_write 命令，则直到 177
发出 MFC 信号表明请求的存储器操作已经完成之后，该步才会结束。

为了将一个执行步骤的持续时间延长到一个时钟周期以上，我们需要禁用步计数器。假设只有当 Counter_enable 控制信号启用步计数器时，它才会递增。我们用控制信号 WMFC 来表示正在等待存储器操作完成。在任何执行步中，只要发出 Wait for MFC 命令，WMFC 就会被激活。未发出 WMFC 信号时，应该将 Counter_enable 信号置为 1。另外，在发出 MFC 信号时，应将它置为 1。这就是说：

$$Counter\_enable = \overline{WMFC} + MFC$$

在图 5-20 中的 PC_enable 信号被激活的情况下，在每个时钟周期结束时，一个新值会被装入 PC 中。当把执行步延长为多个时钟周期时，我们必须确保 PC 的值只被递增一次。因此，在取指令时，只有在收到 MFC 信号后 PC 才会被启用。在引发转移的指令的第 3 步中，PC 也会被启用。我们用 BR 表示所有引发转移的指令。那么，PC_enable 可以用如下方式实现：

$$PC\_enable = T1 \cdot MFC + T3 \cdot BR$$

## 5.7 CICS 风格的处理器

在前面的章节中，我们知道 RISC 风格的指令集有助于实现多阶段结构的处理器。所有的指令都可以使用相同的五阶段硬件来变成一的方式执行。因此，其硬件结构简单而且适于流水线操作。同时，控制信号也比较容易生成。

CISC 风格的指令集则更为复杂，因为它们使指令操作数的访问具有更大的灵活性。与只有 Load 和 Store 指令能够访问存储器数据的 RISC 风格指令集不同，CISC 指令可以直接对存储器操作数进行操作。此外，它们也不受一个字长的限制。正如我们在 2.10 节所描述的，一条指令可以用几个字来指定操作数地址和所要执行的动作。因此，CISC 风格的指令需要不同的处理器硬件结构。

图 5-22 给出了一个可能的处理器结构。这个结构与之前讨论的五阶段结构之间的主要区别在于互连模块，该模块将其他各个模块相互连接，其本身并不规定数据流的特定结构或模式。它提供路径，使得在完成指令所需的任意两个组件之间可以进行数据传输。图 5-8 的多阶段结构使用了暂存寄存器，如 RZ 和 RY。图 5-22 的结构中不需要这些寄存器。然而，为了保存指令执行过程中的中间结果，也需要一些寄存器。图中所示的临时寄存器模块就是为了这个目的而设计的。它包含了两个临时寄存器，Temp1 和 Temp2。通过后面的例子，我们可以清晰 178
地了解这两个寄存器的必要性。

实现互连的一种传统方法是使用总线。总线（bus）是由连接多个设备的一组线路组成的，使得数据可以在任意两个设备之间传输。我们把在总线线路上发送信号的逻辑门称为总线驱动

器（bus driver）。由于每一个与总线相连的设备都可以发送数据，我们必须确保在任何时刻只有一个设备在驱动总线。为此，总线驱动器被设计为一个特殊类型的逻辑门，称为三态门（tri-state）。它包括一个可以将其打开或关闭的控制输入端。当打开三态门时，三态门根据其输入的值将逻辑信号 0 或 1 放置到总线上。当关闭时，三态门在电路上与总线断开连接，如附录 A 所述。

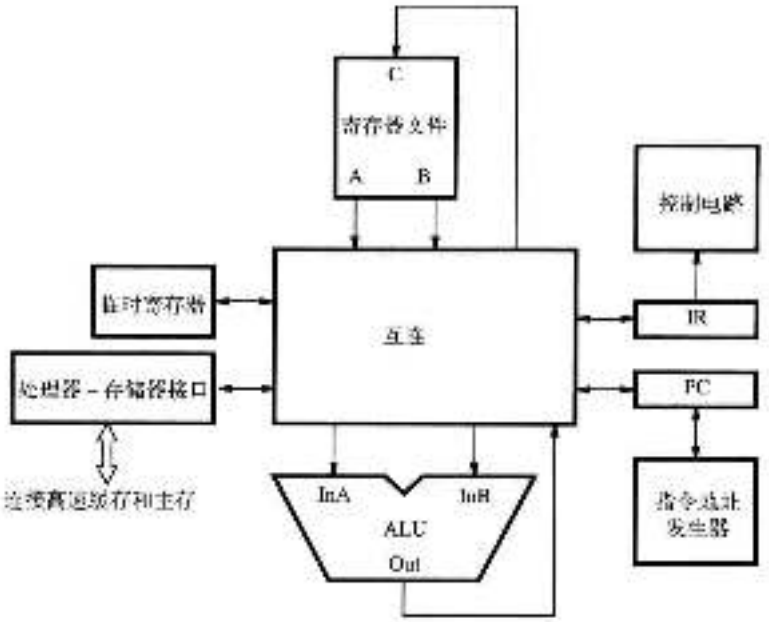



图 5-22　CICS 风格处理器的结构

图 5-23 显示了形成数据寄存器中一个位的触发器是如何连接到总线上的。有两个控制信号 $R_a$ 和 $R_{out}$，当 $R_a$ 等于 1 时，多路复用器将总线线路上的数据装入触发器中。将 $R_a$ 置 0 会使得触发器保持其当前值。触发器的输出端通过一个三态门连到总线线路上，当发出信号 $R_{out}$ 时，三态门就被打开。在其他时间，三态门是关闭的，从而可以允许其他的组件驱动总线线路。

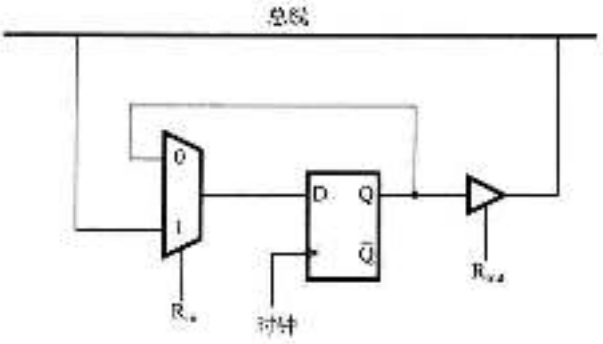



图 5-23　一个寄存器位的输入/输出门

179

### 5.7.1　使用总线实现互连

图 5-22 中的互连模块可以使用一个或多个总线实现。图 5-24 给出了一种三总线的实现方案。假定所有的寄存器都是边沿触发的，也就是说，当寄存器处于启用状态时，在时钟周期结束时的时钟工作沿将数据装入其中。控制模块为寄存器文件的三个端口提供地址。为了简洁起见，我们没有在图中画出这些连接。同样也没有画出将 IR 连到总线 B 上的立即数模块，该模块的电路将 IR 中的立即操作数扩展为 32 位。

考虑双操作数指令

Add R5, R6

该指令执行的操作为

R5 ← [R5] + [R6]

用图 5-24 中的硬件取出并执行该指令的操作可以在二步内完成，如图 5-25 所示。每一步，只要不涉及存储器访问，都可以在一个时钟周期内完成。在第 1 步中，我们使用总线 B 将 PC 的内容发送至处理器—存储器接口，该接口再将 PC 的内容发送到存储器地址线上并初始化存储器读操作。从存储器中获得的数据，即要执行的指令，通过总线 C 发送到 IR 中。5.4.2 节已经解释过存储器访问时间可能会超过一个时钟周期，所以在这里我们插入一个 Wait for MFC 命令来进行调整。在第 2 步中，对指令进行译码，同时控制电路开始读取源寄存器 R5 和 R6。然而，直到第 3 步，寄存器的内容才被放到寄存器文件的 A、B 输出端上。随即通过总线 A 和 B 将它们发送至 ALU。由 ALU 执行加法运算，其结果通过总线 C 送回至 ALU，并在该时钟周期结束时将其写入寄存器 R5。

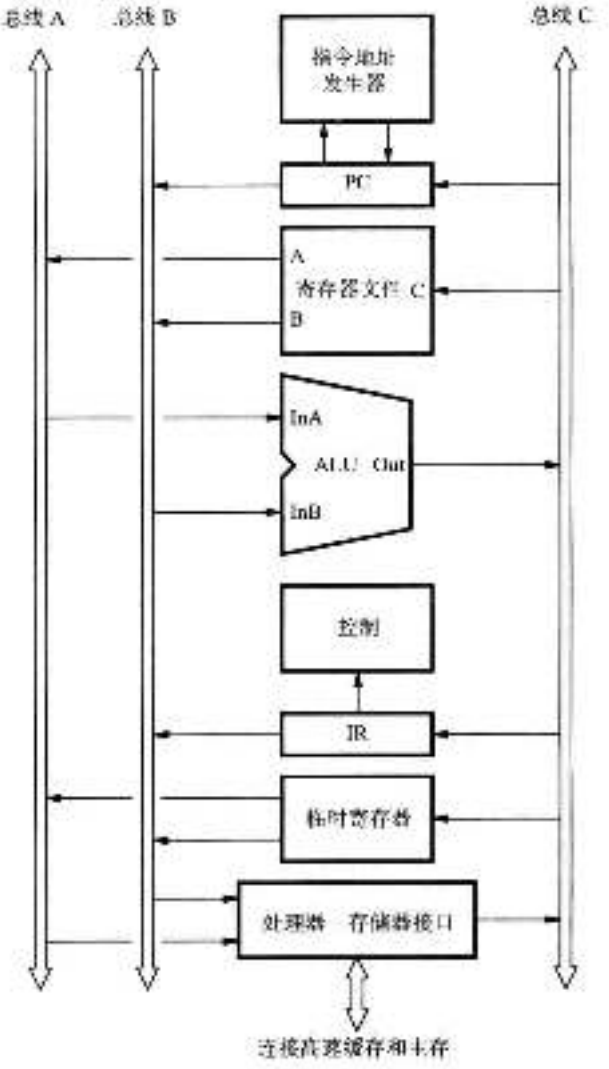



图 5-24 三总线的 CISC 风格的处理器结构

我们注意到，在图 5-11 中，读取源寄存器的操作是在第 2 步完成的。那时，读取寄存器的操作与指令译码操作并行进行，因为 RISC 风格的指令中包含寄存器地址的位字段位置是已知的。由于 CISC 风格的指令并不总是使用同一个指令字段来指定寄存器地址，因此在指令至少被部分译码之前，读取源寄存器的操作是不会开始的。因此，读取源寄存器的操作就可能无法在第 2 步内完成。

180
181

| 步骤 | 动作 |
|---|---|
| 1 | 存储器地址 ← [PC]，读取存储器；Wait for MFC，IR ← 存储器数据<br>PC ← [PC] + 4 |
| 2 | 对指令进行译码 |
| 3 | R5 ← [R5] + [R6] |

图 5-25 读取并执行指令 Add R5, R6 所需的动作序列

接下来，考虑指令

And X(R7), R9

该指令对寄存器 R9 和存储单元 X + [R7] 的内容进行逻辑 AND 运算，并把结果存回到 X +[R7] 中。假设变址偏移量 X 是 32 位的值，由指令中的第二个字给出。为了执行该指令，需要对存储器进行四次访问。首先，取出操作码。然后，当指令译码电路识别出变址方式时，取出变址偏移量 X。接着，读取存储器操作数并进行 AND 运算。最后，再将结果存回到存储器中。

| 步骤 | 动作 |
|---|---|
| 1 | 存储器地址 ← [PC], 读取存储器，Wait for MFC，IR ← 存储器数据，<br>PC ← [PC] + 4 |
| 2 | 对指令进行译码 |
| 3 | 存储器地址 ← [PC], 读取存储器，Wait for MFC，Temp1 ← 存储器数据<br>PC ← [PC] + 4 |
| 4 | Temp2 ← [Temp1] + [R7] |
| 5 | 存储器地址 ← [Temp2], 读取存储器，Wait for MFC，Temp1 ← 存储器数据 |
| 6 | Temp1 ← [Temp1] AND [R9] |
| 7 | 存储器地址 ← [Temp2], 读取存储器 ← [Temp1], 写入存储器，Wait for MFC |

图 5-26　读取并执行指令 And X(R7), R9 所需的动作序列

图 5-26 给出了执行该指令的步骤。在第 2 步中对指令进行译码。然后在第 3 步中读取指令的第二个字。所得到的数据（即偏移量 X）被临时存储在寄存器 Temp1 中，以便在下一步 [182]
中用来计算存储器操作数的有效地址。在第 4 步中，寄存器 Temp1 和 R7 的内容通过总线 A 和 B 发送到 ALU 的输入端，计算得到有效地址并将其装入寄存器 Temp2 中。在第 5 步中使用该地址读取操作数并用寄存器 Temp1 保存这个从存储器中接收到的数据操作数。在第 6 步中开始进行计算，其结果会被放回到寄存器 Temp1 中。在最后一步中，再将结果发送至地址仍存储在寄存器 Temp2 中的操作数所指向的存储单元中存储。

图 5-25 和图 5-26 中的两个例子说明了 CISC 风格指令的执行步数是可变的。与 5.2 节中所述的 RISC 风格指令不同，这里不存在一个适合所有指令的统一动作序列。

### 5.7.2　微程序控制

控制图 5-22 和图 5-24 中操作操作的控制信号可以通过使用 5.6 节中的硬件方法生成。但是，在过去还流行着另外一种有趣的控制方式，下面我们将会介绍。

根据 IR 中的指令，我们为每一个执行步生成相应的控制信号。在硬件控制方式中，这些信号是由解释 IR 内容的电路以及步计数器的时序信号产生的。如果不采用这样的电路，我们还可以使用一种“软件”的方法，即由存储在特殊存储器中的一个程序来决定每一步中控制信号所需的设置。为了将其与处理器执行的程序区别开来，我们把该控制程序称为微程序（microprogram）。微程序存储在处理器芯片上一个小型而快速的存储器中，该存储器被称为微程序存储器（microprogram memory）或控制存储器（control store）。

假设我们需要 $n$ 个控制信号。每一个控制信号可以用 $n$ 位字中的一位来表示，该 $n$ 位字通常被称为控制字（control word）或微指令（microinstruction），其中的每一位都指定了执行流中特定步的相应信号的设置。对于指令执行序列中的每一步，都对应有一个控制字存储在微程序存储器中。例如，从存储器中读取指令或数据操作数的动作分别需要使用 5.5 和 5.6.2 节所介绍的 MEM read 信号和 WMFC 信号。在图 5-26 的第 1、3、5 步中，通过将控制字的相应位置为 1，即可发出这些信号。当从控制存储器中读取一条微指令时，每个控制信号的值就是

其对应位的值。

与给定的机器指令相对应的微指令序列构成了实现该指令的微例程（microroutine）。图 5-25 和图 5-26 的前两步指定了取指令和指令译码的操作。这些操作适用于所有指令。而从第 3 步开始，微例程才与给定的机器指令相对应。

图 5-27 描述了微程序控制所需硬件的典型结构。其中包括一个微指令地址发生器，它可生成用于从控制存储器
185 中读取微指令的地址。该地址发生器使用了微程序计数器（microprogram counter）μPC，以便于在从连续的单元中读取微指令时跟踪控制存储器的地址。在图 5-25 和图 5-26 的第 2 步中，微指令地址发生器对 IR 中的指令进行译码，以便得到相应微例程的起始地址，并将该地址装入 μPC 中。该地址将用于在下一个时钟周期读取第 3 步中相应的控制字。随着执行的推进，微指令地址发生器将 μPC 递增以便从控制存储器的连续存储单元中读取微指令。微指令中有一个称为 End 的位，用来标记给定微例程的最后一条微指令。当 End 等于 1 时（图 5-25 的第 3 步和图 5-26 的第 7 步就是这种情况），地址发生器会返回到对应于第 1 步的微指令，这样就可以取出一条新的机器指令。

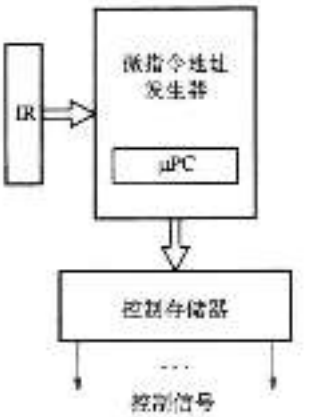


图 5-27 微程序控制部件的结构

我们可以把微程序控制看成是在主处理器中还有一个控制处理器。微指令的取出和执行与机器指令非常相似。它们的作用就是确定在每个执行步中需要激活哪些控制信号以指引主处理器硬件组件的操作。

微程序控制很容易实现且能够在控制机器指令的执行方面提供相当大的灵活性。但是，它比硬件控制要慢。同时，它所提供的灵活性在 RISC 风格的处理器中是不需要的。正如本章
184 所讨论的，实现 RISC 风格指令所需的控制信号是很容易生成的。由于逻辑电路的成本已经不再是一个重要因素，所以硬件控制已经成为最佳选择。

## 5.8 结束语

在本章中，我们介绍了处理器的基本结构及其执行指令的方式。现代处理器采用多阶段的组织结构，该结构非常适合于流水线操作。每个阶段实现指令的一个执行步所需的动作。我们给出了一个五步序列，在该序列中，每一步都在一个时钟周期内完成。这种方法常用于 RISC 风格的指令集处理器中。

在本章的讨论中，我们假设在取出下一条指令前，前一条指令已经执行完毕。随着指令的执行在每个时钟周期中从一个阶段移动到下一阶段，在任意时刻，五个硬件阶段中只有一个在使用。在下一章中我们将会说明，将多条连续指令的执行步重叠起来是有可能的，从而可以带来更好的性能。这就是流水线结构。

## 5.9 问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。

**例 5.1**

**问题：** 图 5-11 给出了执行 Add 指令的五个步骤，但是在第 4 步中并没有发生任何处理操作。如果我们想要去除这一步，应该怎样修改图 5-8 中的数据通路？

**解答：** 将图 5-8 中 ALU 的输出直接发送至寄存器 RY，可以跳过第 4 步。我们可以给多路复用器

MuxY 增加一个输入端，并将该输入端连接到 ALU 的输出端来实现。这样，在 ALU 输出端的计算结果就会在第 3 步结束时装入寄存器 RZ 和 RY 中。对于 Add 指令，或者任意其他的计算型指令，我们可以在第 4 步中激活寄存器文件控制信号 RF_write 来将 RY 的内容装入寄存器文件中。

**例 5.2**

**问题**：假设在一个使用 1GHz 时钟的处理器中，所有的存储器访问操作都在一个时钟周期内完成。如果在一个程序中，Load 和 Store 指令占动态指令数的 20%，那么存储器访问操作的频率是多少？（动态指令数是指实际执行的指令数量，包括可能会使得一些指令被执行多次的程序循环。）假定所有的指令都在 5 个时钟周期内执行完成。

**解答**：读取每一条指令时都会涉及一次存储器访问。还有 20% 的指令需要对存储器进行第二次访问以便读取或写入存储器操作数。平均而言，在 5 个时钟周期里，每条指令会进行 1.2 次存储器访问。因此，存储器访问的频率是 $(1.2/5) \times 10^9$，即每秒 2.4 亿次。

**例 5.3**

**问题**：请推导出比较两个无符号数 $X = x_2x_1x_0$ 和 $Y = y_2y_1y_0$ 的电路的逻辑表达式，并生成三个输出 $XGY$，$XEY$ 和 $XLY$，每其中的一个输出置为 1 来分别表示 $X$ 大于、等于或小于 $Y$。

**解答**：为了比较两个无符号数，我们需要从最高有效位开始，依次比较各个位。如果 $x_2 = 1$ 且 $y_2 = 0$，那么 $X$ 大于 $Y$。如果 $x_2 = y_2$，那么我们就需要比较下一位，以此类推。因此，三个输出的逻辑表达式可以表示为：

$$XGY = x_2\overline{y}_2 + \overline{(x_2 \oplus y_2)} \cdot (x_1\overline{y}_1 + \overline{(x_1 \oplus y_1)} \cdot x_0\overline{y}_0)$$

$$XEY = \overline{(x_2 \oplus y_2)} \cdot \overline{(x_1 \oplus y_1)} \cdot \overline{(x_0 \oplus y_0)}$$

$$XLY = \overline{XGY + XEY}$$

**例 5.4**

**问题**：请给出 RISC 风格处理器中从子程序返回（Return-from-subroutine）指令的动作序列。假设存储了子程序返回地址的通用寄存器的地址 LINK 在与寄存器文件的地址 A 相连的指令字段中给出（$IR_{31-27}$）。

**解答**：每当指令被装入 IR 中时，我们都会读取位 $IR_{31-27}$ 所指定的通用寄存器的内容，并将其装入寄存器 RA 中（见图 5-18）。因此，从子程序返回指令会使得寄存器 LINK 的内容被读取，并装入寄存器 RA 中。其执行过程如下：

1）存储器地址 ← [PC]，读取存储器，Wait for MFC，IR ← 存储器数据，PC ← [PC] + 4

2）对指令进行译码，RA ← [LINK]

3）PC ← [RA]

4）空操作

5）空操作

**例 5.5**

**问题**：某处理器的中断结构如下：当收到一个中断时，会将该中断的返回地址保存到通用寄存器 IRA 中，处理器状态寄存器 PS 的当前内容，会被保存在一个特殊的寄存器 IPS 中，该寄存器不是通用寄存器。中断服务程序从地址 ILOC 开始。

假设处理器在每条指令的最后一个执行步中进行中断检测。如果出现中断请求，并且处理器允许中断，则该中断请求被接受。处理器会保存 PC 和 PS 并跳转至 ILOC，而不是去读取下一条指令。请给出执行这些操作的相应步骤序列。为了支持中断处理，还需要在图 5-18 到图 5-20 中增加哪些硬件？

**解答**：指令执行的前两步，取指令和指令译码，在中断的情况下是不需要的。这两步可以被跳过，或者为空操作以保持 5 步序列。我们可以用与子程序调用指令完全相同的方式来保存 PC。图 5-18 中的多路复用器 MUXC 需要增加一个输入端，并将其与寄存器 IRA 的地址相连。为了将中断服务程序的起始地址装入 PC 中，我们需要为图 5-20 中的 MuxPC 增加一个输入端与 ILOC 值相连。寄存器 PS 与 IPS 应该直接互连，这样可以使数据在两者之间传输。所需的执行步骤如下：

3. PC-Temp ← [PC]，PC ← ILOC，IPS ← [PS]，禁止中断

4. RY ← [PC-Temp]

5. IRA ← [RY]

对 Return-from-interrupt 指令而言，上述操作是逆向进行的。参见习题 5.8。

**例 5.6**

**问题**：例 5.5 说明了当接收到中断请求时 PC 和 PS 的内容是如何保存的。如 3.2 节所述，为了支持中断嵌套，中断服务程序需要将这些寄存器保存到处理器堆栈中。要做到这一点，就需要在接收中断时将保存在寄存器 IPS 中的 PS 的内容移到一个通用寄存器中，然后再从那里保存到堆栈中。假设两条特殊的指令

MoveControl Ri, IPS

和

MoveControl IPS, Ri

可分别用来保存和恢复 IPS 的内容。请对图 5-8 和图 5-10 中的硬件进行修改以实现这些指令。

**解答**：图 5-28 给出了一种可能的结构。为了保存 IPS 的内容，我们将其输出端与 MuxY 新增的输入端相连。而在恢复其内容时，MuxIPS 会选择寄存器 RA。

187

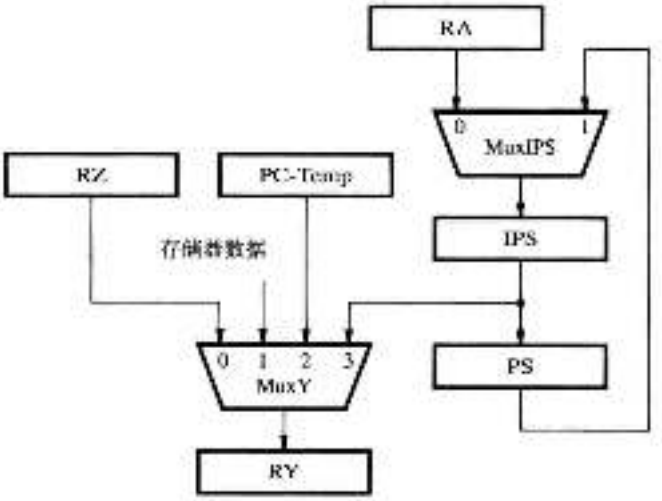



图 5-28 例 5.6 中 IPS 的连接

## 习题

[M] 5.1 图 5-2 中，组合电路的传播延迟为 600 ps（皮秒；$10^{-12}$ 秒）。寄存器的准备时间需要 50 ps，从时钟输入端到 Q 输出端的最大传播延迟为 70 ps。

（a）如果该电路正确操作，则其最小时钟周期是多少？

（b）假设该电路被重新组织为图 5-3 中的三个阶段，这样该组合电路在每个阶段的延迟是 200 ps。请问这种情况下最小的时钟周期是多少？

[M] 5.2 在读取指令

Load R6, 1000（R9）

时，R6 和 R9 中的值分别为 4200 和 85320。存储单元 86320 中的值是 75900。请给出在该指令五个执行步的每一步中，图 5-8 中段间寄存器的内容。

[E] 5.3 图 5-12 给出了不同指令域其寄存器地址的位字段分配情况。为什么所有的指令要使用相同的字段位置？

[M] 5.4 在程序执行的某个时刻，寄存器 R4，R6 和 R7 中的值分别为 1000，7500 和 2500。请给出在读取并执行指令 188

Subtract R6, R4, R7

的第 3 步至第 5 步以及读取下一条指令的第 1 步时，寄存器 RA、RB、RZ、RY 和 R6 的内容。

[M] 5.5 指令

And R4, R4, R8

存放在存储单元 0x37C00 处。在读取该指令时，寄存器 R4 和 R8 的值分别为 0x1000 和 0xB2500。请给出图 5-8 和图 5-10 中该指令执行的每个时钟周期以及下一条指令的第一个时钟周期中，寄存器 PC、R4、RA、RM、RZ 和 RY 的值。

[D] 5.6 修改例 5.3 中的表达式，以比较两个用补码形式表示的 4 位有符号数。

[E] 5.7 第 2 章所描述的子程序调用指令总是使用相同的通用寄存器 LINK 来保存返回地址。因此，指令中不包含该寄存器的地址。然而，在子程序返回指令中，地址 LINK 却包含在位 $IR_{31-27}$ 中（参见 5.4.1 节和例 5.4）。为什么这两条指令会有这样的区别？

[M] 5.8 对于具有例 5.5 所示中断结构的处理器，请给出其 Return-from-interrupt 指令的执行步骤序列。假设寄存器 IRA 的地址在指令的位 $IR_{31-27}$ 中给出。

[D] 5.9 考虑一个指令集，其指令的编码方式使得不同指令的寄存器地址并不总在相同的位位置中。请问这会对指令的执行步骤造成什么影响？在这种情况下，应该怎样做才能保持五步的执行序列？假设使用与图 5-8 相同的硬件结构。

[M] 5.10 假设立即操作数占据指令的 $IR_{21-6}$ 位。该立即值在算术运算指令中被符号扩展为 32 位，如 Add 指令；在逻辑运算指令中被用零填充为 32 位，如 Or 指令。请为图 5-9 中的立即数模块设计一种合适的实现方式。

[M] 5.11 一个使用图 5-4 中五步序列的 RISC 处理器是由一个 1 GHz 的时钟驱动的。一个大型程序中的指令统计结果如下：

| | |
|---|---|
| Branch | 20% |
| Load | 20% |
| Store | 10% |
| 计算型指令 | 50% |

请预测在下面每一种情况中指令执行的速率：

（a）存储器访问始终在 1 个时钟周期内完成。

（b）90% 的取指令操作在一个时钟周期内完成，另外 10% 则需要 4 个时钟周期。而 Load 或 Store 指令中的数据操作数访问则平均在 3 个时钟周期内完成。 189

[E] 5.12 计算型指令的执行按照图 5-11 中 Add 指令的模式进行，其中在第 4 步中没有执行任何操作。考虑一个程序，其指令统计结果在习题 5.11 中给出。请估测在取消第 4 步的情况下，指令执行速率的提升效果。假设所有的执行步都在一个时钟周期内完成。

[D] 5.13 图 5-16 显示，条件转移指令的第 3 步可能会使得一个新值被装入 PC 中。在流水线处理器中，我们希望在执行序列中尽早确定条件转移的结果。请问需要对硬件进行什么样的改变才可以把第 3 步的操作移到第 2 步中？分析这两步中的每一个操作，并说明哪些操作可以并行执行，哪些操作必须顺序执行。

[M] 5.14 某计算机指令的编码方式如图 5-12 所示。当指令中给出一个立即值的时候，它必须被扩展为 32 位。假设该立即值有以下三种不同的使用方式：

（a）16 位的值，被符号扩展，用于算术运算；

（b）16 位的值，左边用零填充，用于逻辑运算；

（c）26 位的值，右边填充 2 个零，左边用 PC 的 4 个高位进行扩展，用于子程序调用指令。

请给出图 5-19 中立即数模块的一种实现方式以完成上述所需的扩展操作。

[E] 5.15 我们已经知道 RISC 风格的指令可以在图 5-8 的多阶段操作中用图 5-4 所示的步骤来执行。RISC

风格的指令集中不包括自动增量和自动减量寻址方式。请解释指令

Load R3,(R5)+

为什么不能在图 5-8 的硬件中执行。

[E] 5.16 2.9 节描述了如何使用 Or 和 OrIHigh 两条指令来将一个 32 位的值装入寄存器中。请问为了实现 OrIHigh 指令，还需要在处理器的数据通路中新增哪些功能？请给出读取及执行该指令的动作序列。

[E] 5.17 在指令处理的第 1 步中，启动一个存储器读操作以读取在存储单元 0x46000 处的指令。然而，由于在高速缓存中没有找到该指令，所以该读操作被延迟，MFC 信号直到第四个时钟周期才被激活。假设该延迟按 5.6.2 节中所述的方式进行处理。请分别给出在第 1 步的四个时钟周期中以及第 2 步中 PC 的内容。

[M] 5.18 请给出读取并执行例 5.6 中所使用的两条特殊指令

MoveControl R/, IPS

和

MoveControl IPS, R/

190 的步骤序列。

[D] 5.19 图 5-8 和图 5-22 的硬件结构之间的本质区别是什么？通过确定在图 5-8 所示的硬件中执行指令

Subtract LOC, R5

时可能会遇到的困难来说明你的答案。该指令执行以下操作：

LOC ← [LOC]− [R5]

其中，LOC 是一个存储单元，其地址由一条双字指令的第二个字给出。

[M] 5.20 考虑执行 5.4.1 节给出的指令所需的动作。对于这些指令，请推导出生成图 5-18 和图 5-19 中的 C_select、MA_select 和 Y_select 信号的逻辑表达式。

[E] 5.21 为什么在 5.6.2 节所给出的 Counter_enable 的逻辑表达式中需要同时包含 WMFC 和 MFC 两个信号？

[E] 5.22 请说明如果将 5.6.2 节所给出的 PC_enable 表达式中的 MFC 变量省略掉，会发生什么情况？

[M] 5.23 请推导出生成图 5-20 所示的 PC_select 和 INC_select 信号的逻辑表达式，考虑执行以下指令所需的操作：

Branch：所有转移指令，其中每条指令都包括一个 16 位的转移偏移量

Call_register：子程序调用指令，其子程序的地址在一个通用寄存器中给出

其他：所有不包含转移的其他指令

[M] 5.24 一个微程序处理器具有如下参数。生成一条指令的微例程起始地址需要 2.1ns，从控制存储器中读取一条微指令需要 1.5ns。执行一次 ALU 运算最多需要 2.2ns，访问高速缓存需要 1.7ns。假设所有的指令和数据都在高速缓存中。

(a) 请确定图 5-26 中的每一步所需要的最小时间。

(b) 忽略所有其他延迟，该处理器可使用的最小时钟周期是多少？

[M] 5.25 请给出在图 5-24 所示的处理器中读取并执行指令

Load R3,(R5)−

的步骤序列。假设操作数是 32 位。

191 / 192 [M] 5.26 一个 CISC 风格的处理器将子程序的返回地址保存在处理器堆栈中，而不是在预先定义的寄存器 LINK 中。请给出在图 5-24 所示的处理器中执行 Call_register 指令的动作序列。

# 流 水 线

## 本章目标

在本章中你将学习以下内容：

- 通过重叠执行机器指令来提高性能的流水线方式
- 限制流水线处理器性能提升的冲突（hazard）以及降低冲突影响的方法
- 流水线中的硬件和软件含义
- 流水线对指令集设计的影响
- 超标量处理器

[193]

第5章介绍了一次执行一条指令的处理器结构。在本章中，我们将讨论流水线的概念，它重叠执行连续的指令，以获取更高的性能。我们首先概述流水线的基本知识以及它是如何提高性能的。然后研究导致性能下降的冲突以及减轻它们对性能影响的技术。我们讨论优化编译器（optimizing compiler）的作用，它可以重新排列指令序列以使流水线的效益最大化。为进一步提高性能，我们还考虑在超标量（superscalar）处理器中复制硬件设备，这样多个流水线可以同时操作。

## 6.1 基本概念——理想情况

程序执行的速度受到许多因素影响。提高性能的一种方法是利用快速电路技术来实现处理器和主存储器，另一种可能是对硬件进行合理地安排，使其能同时执行多项操作。采用后一种方法，即使执行任何一个操作所需的时间不变，但是每秒钟执行操作的数目也会增加。

在计算机系统中，流水线是组织并发活动的一种非常有效的方法。它的基本思想很简单。在制造工厂里经常会看到流水线，那里的流水线通常是作为装配线操作的。毫无疑问，读者对汽车制造中用到的装配线是很熟悉的。装配线的第一站准备汽车底盘，第二站增加车身，下一站安装发动机等。当一组工人在一辆汽车上安装发动机时，另一组人在第二辆汽车的底盘上安装车身，还有一组人为第三辆汽车准备新的底盘。虽然完成一辆汽车可能会花费几小时或几天的时间，但是装配线操作使得每隔几分钟就有一辆新车从装配线的末端开出来。

让我们考虑一下流水线的思想如何能用于计算机中。图5-7中的五阶段处理器结构以及图5-8中相应的数据通路允许一次提取并执行一条指令。完成每条指令的执行需要花费五个时钟周期。我们不需要等到每条指令都执行完成才去提取和执行其他指令，而是可以用图6-1所示的流水线方式来提取并执行指令。这五个阶段对应于图5-7中的五个阶段，标记为取指（Fetch）、译码（Decode）、计算（Compute）、访存（Memory）和写回（Write）。在第一个周期中，提取指令 $I_1$，它将在接下来的周期中经过剩下的阶段。在第二个周期中，提取指令 $I_{i+1}$，而指令 $I_i$ 现在处于译码阶段，同时它的操作数正在从寄存器文件中读取。在第三个周期中，提取指令 $I_{i+2}$，而指令 $I_{i+1}$ 现在处于译码阶段，指令 $I_i$ 处于计算阶段，正在对它的操作数进行一个算术或逻辑运算。最理想的是，这种重叠执行的模式对所有指令都是可能的。虽然任何一条指令需要花费五个周期才能完成它的执行，但是指令是按每个周期一条的速率完成的。

[194]

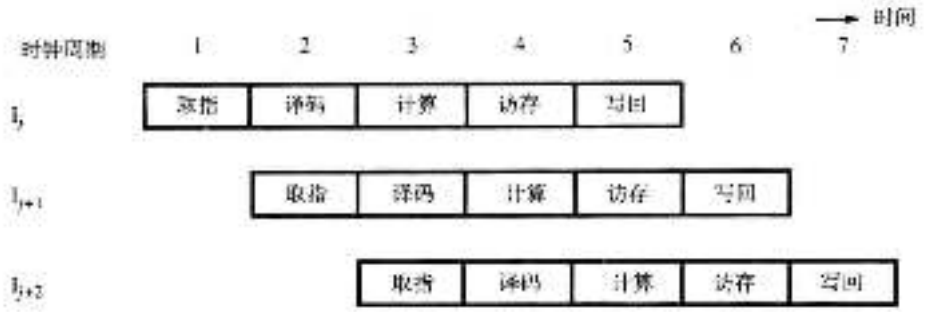



图 6-1 指令的流水线执行——理想情况

## 6.2 流水线结构

图 6-2 显示了如何对图 5-7 和图 5-8 中的五阶段结构实现流水线。在流水线的第一个阶段，程序计数器（PC）用于提取一条新的指令。当提取其他指令时，前一条指令通过后续的阶段继续执行。在任何时间，流水线的每个阶段都在处理不同的指令。随着每条指令从一个阶段进行到下一个阶段，诸如寄存器地址、立即数据和将要执行的操作之类的信息必须也被传送通过流水线。这些信息被保存在段间缓冲器（interstage buffer）中，这些缓冲器包括图 5-8 中的寄存器 RA、RB、RM、RY 和 RZ，图 5-9 和图 5-10 中的 IR 和 PC-Temp 寄存器，以及辅助存储器。段间缓冲器使用如下：

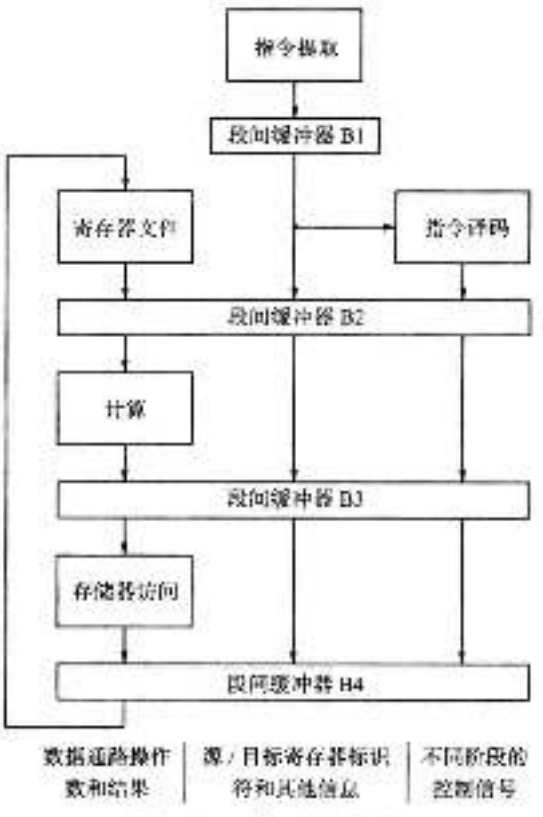



图 6-2 一条 5 段流水线

- 段间缓冲器 B1 向译码阶段提供一条新提取的指令。
- 段间缓冲器 B2 向计算阶段提供从寄存器文件中读取的两个操作数、源/目标寄存器标识符、来自指令的立即值、用作子程序调用返回地址的递增后的 PC 值，以及由指令译码器确定的控制信号的设置值。控制信号的设置值穿过流水线以确定 ALU 运算、存储器操作以及一个可能的写入寄存器文件的操作。
- 段间缓冲器 B3 保存 ALU 运算的结果，它可能是即将写入寄存器文件的数据或者是送入访存阶段的一个地址。在对存储器进行写访问时，缓冲器 B3 里保存着要写入的数据，这些数据是在译码阶段从寄存器文件中读取的。缓冲器中还保存从前一阶段传递过来的递增后的 PC 值，以防需要它作为子程序调用指令的返回地址。
- 段间缓冲器 B4 向写回阶段提供一个即将写入寄存器文件的值，这个值可能是计算阶段的 ALU 运算结果，也可能是存储器访问阶段的结果，或者是用作子程序调用指令返回

地址的递增后的 PC 值。

## 6.3 流水线问题

图 6-1 描述了三条连续指令的理想重叠情况。但是，有些时候不可能每个周期都有一条新的指令进入流水线。考虑两条指令 $I_j$ 和 $I_{j+1}$ 的情况，其中指令 $I_j$ 的目标寄存器是指令 $I_{j+1}$ 的一个源寄存器。指令 $I_j$ 的结果直到第 5 个周期才写入寄存器文件，但它早在第 3 个周期为指令 $I_{j+1}$ 读取源操作数时就需要。如果指令的执行按图 6-1 所示的那样进行，那么指令 $I_{j+1}$ 的结果将是不正确的，因为它将使用其源寄存器中的旧值执行算术运算。为获得正确的结果，需要等到指令 $I_j$ 将新值写入寄存器中。因此，指令 $I_{j+1}$ 直到第 6 个周期才能读取它的操作数，这意味着它必须在译码阶段被暂停（stall）3 个周期。当指令 $I_{j+1}$ 被暂停时，指令 $I_{j+2}$ 和所有后续指令也同样被延迟。新的指令不能进入流水线，总的执行时间也将增加。

195 196

我们称这种引起流水线暂停的情况为冲突（hazard）。我们刚刚描述的是一个数据冲突（data hazard）的例子，如果一条指令的一个源操作数的值在需要的时候不能获得就会引起这种冲突。还有其他由存储器延迟、转移指令和资源限制产生的冲突。接下来的几节我们将详细描述这些冲突以及降低它们对性能影响的技术。

## 6.4 数据依赖性

考虑图 6-3 中的两条指令：

```
Add         R2, R3, #100
Subtract    R9, R2, #30
```

Add 指令的目标寄存器 R2 是 Subtract 指令的一个源操作数。在这两条指令间存在数据依赖性（data dependency），因为寄存器 R2 将数据从第一条指令传输到第二条指令。图 6-3 描述了这两条指令的流水线执行情况。Subtract 指令被暂停了 3 个周期，直到第 6 个周期可以获得新数据时才延迟读取寄存器 R2 的内容。

我们现在详细解释流水线停顿。当控制电路在第 3 个周期译码 Subtract 指令时，它必须先识别出数据依赖性，这可以通过将保存在段间缓冲器 B1 中的 Subtract 指令的源寄存器标识符与保存在段间缓冲器 B2 中的 Add 指令的目标寄存器标识符进行比较来识别。然后，在第 3 个周期到第 5 个周期期间，Subtract 指令必须保持在段间缓冲器 B1 中。与此同时，Add 指令继续通过剩下的流水线阶段。在第 3 个周期到第 5 个周期中，随 Add 指令向前移动，可以在段间缓冲器 B2 中设置一条隐含的 NOP（空操作）指令控制信号，它不会修改存储器或寄存器文件。每条 NOP 指令通过计算、访存和写回阶段到达流水线终端时，就产生了一个空闲时间的时钟周期，称为气泡（bubble）。

197

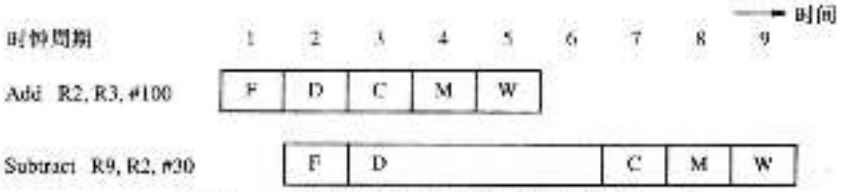



图 6-3 由于数据依赖性而产生的流水线停顿

### 6.4.1 操作数转发

由于数据依赖性而产生的流水线停顿可以通过使用操作数转发（operand forward）技术来

缓解。考虑上面讨论的那对指令，流水线被暂停了 3 个周期后 Subtract 指令才可以使用寄存器 R2 中的新值。然而，实际上所需的值在第 3 个周期的末尾就可用了，那时 ALU 已经完成了 Add 指令的操作。这个值被装入图 5-8 中的寄存器 RZ 中，RZ 是段间缓冲器 B3 的一部分。硬件可以将这个值从寄存器 RZ 中转发至第 4 个周期中需要它的地方，也就是 ALU 的输入端，而不必延迟 Subtract 指令。图 6-4 显示了实现操作数转发时指令的流水线执行情况。箭头表示第 3 个周期的 ALU 结果被作为第 4 个周期中 ALU 的一个输入端使用。

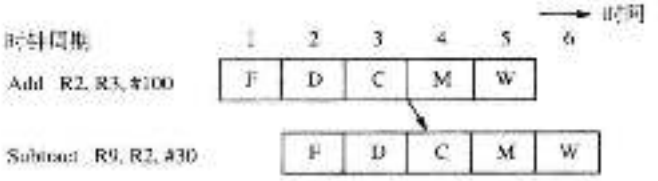



图 6-4 使用操作数转发来避免流水线停顿

图 6-5 对图 5-8 中的数据通路进行必要的修改，以支持这种转发机制。一个新的多路复用器 MuxA 插入到了 ALU 的输入端 InA 之前，而已有的多路复用器 MuxB 扩展出了另一个输入端。这两个多路复用器要么选择按正常方式从寄存器文件中读取的值，要么选择寄存器 RZ 中的可用值。

引申一下，图 5-8 中寄存器 RY 所保存的结果可能也需要进行转发。比如在下列指令序列中就可以通过这种方式进行处理：

| | |
|---|---|
| Add | R2, R3, #100 |
| Or | R4, R5, R6 |
| Subtract | R9, R2, #30 |

当 Subtract 指令处于流水线的计算阶段时，Or 指令处于访存阶段（不执行任何操作），Add 指令处于写回阶段。由 Add 指令产生的寄存器 R2 的新值或在寄存器 RY 中，将这个值从寄存器 RY 转发到 ALU 的输入端 InA 则可以避免暂停流水线。为此 MuxA 需要另一个 RY 值作为输入，同样，MuxB 也用另一个输入端进行扩展。

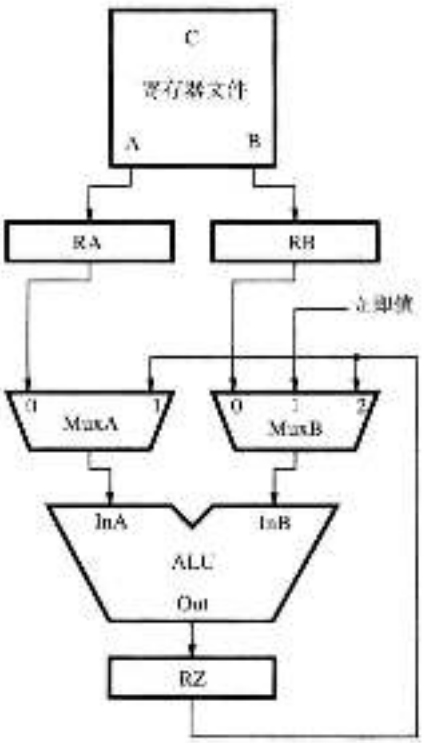



图 6-5 修改图 5-8 中的数据通路，以支持数据从寄存器 RZ 转发至 ALU 的输入端

### 6.4.2 用软件处理数据依赖性

图 6-3 和图 6-4 显示了通过处理器硬件来处理数据依赖性的方法：暂停流水线或者转发数据。另一种方法是将检测和处理数据依赖性的任务留给编译器。当编译器识别出在指令 $I_i$ 和 $I_{i+1}$ 之间存在数据依赖性时，它会在这两条指令之间插入 3 条显式的 NOP（空操作）指令。NOP 指令引入了必要的延迟，使得指令 $I_{i+1}$ 能够在寄存器文件被写入后再从其中读取新的值。对于图 6-4 中的指令，编译器将生成图 6-6a 中的指令序列，从图 6-6b 可以看出 3 条 NOP 指令在执行时间上与图 6-3 中的流水线停顿有相同的效果。

198 / 199

利用编译器识别依赖性并插入 NOP 指令简化了流水线的硬件实现。但是，代码长度增加了，并且执行时间没有像使用操作数转发时那样减少。编译器可以尝试优化（optimize）代码以提高性能，并通过重新排序指令，将有用的指令移到 NOP 槽中来减少代码长度。这样做时，编译器必须考虑指令间的数据依赖性，这限制了 NOP 槽可以被有效填充的程度。

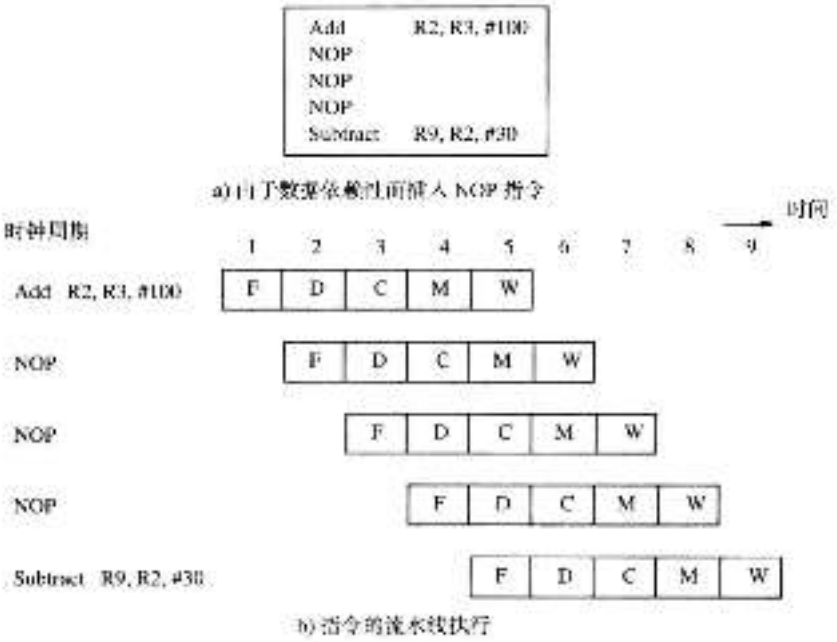



图 6-6 使用 NOP 指令从软件上处理数据依赖性 [200]

## 6.5 存储器延迟

由存储器访问而产生的延迟是流水线停顿的另一个原因。例如，一条 Load 指令可能需要多个时钟周期来从存储器中获取它的操作数。当所请求的指令或数据在高速缓存中没有找到，即高速缓存失效（cache miss）时就会发生这种情况。图 6-7 显示了在流水线执行过程中访问存储器中数据所产生延迟的影响。一次存储器访问可能需要花费 10 个或更多的周期，为简单起见，图中只显示了 3 个周期。高速缓存失效会导致所有后续的指令被延迟，取指令时高速缓存失效会引起类似的延迟。

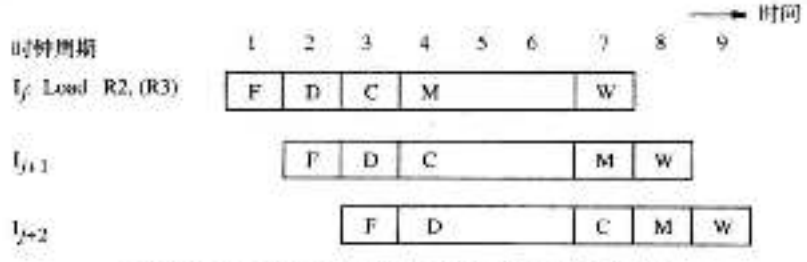



图 6-7 Load 指令中由存储器访问延迟引起的停顿

还有另外一种与存储器有关的停顿，当存在涉及 Load 指令的数据依赖性时会发生。考虑指令：

```
Load        R2, (R3)
Subtract    R9, R2, #30
```

假设 Load 指令的数据可以在高速缓存中找到，那么只需要一个周期来访问操作数。而 Load 指令的目标寄存器 R2 是 Subtract 指令的一个源寄存器。这里不能用图 6-4 所示的同样的方式来进行操作数转发，因为从存储器（在本例中是高速缓存）中读取的数据直到它们在第 5 个周期的开始处被装入寄存器 RY 时才可用。因此，Subtract 指令必须被暂停一个周期，以推迟 ALU 操作，如图 6-8 所示。在第 5 个周期存储器操作数已装入寄存器 RY 中之后才可以将其转发到 ALU 的输入端。

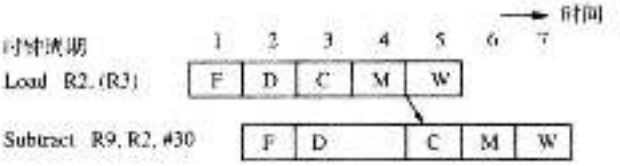



图 6-8 为了向跟在 Load 指令之后的指令转发操作数，需要引入停顿

编译器可以为这种类型的数据依赖性消除这一个周期的停顿，对指令进行重新排序，在 Load 指令和依赖于存储器数据的指令之间插入一条有用的指令即可。这条插入的指令将填充本来会产生的那个气泡。如果编译器找不到一条有用的指令，那么硬件将自动引入一个周期的
201 停顿。如果处理器硬件不能处理依赖性，那么编译器就必须插入一条显式的 NOP 指令。

## 6.6 转移延迟

在理想的流水线执行中，每个周期提取一条新指令，而前面一条指令还在译码中。转移指令会改变执行的顺序，但它们必须先被执行，以确定是否转移以及转移到哪里。我们现在研究转移指令的影响以及用于减轻转移指令对流水线执行的影响的技术。

### 6.6.1 无条件转移

图 6-9 显示了一个指令序列的流水线执行，该序列从一条无条件转移指令 $I_i$ 开始。接下来的两条指令 $I_{i+1}$ 和 $I_{i+2}$ 存储在 $I_i$ 之后的连续存储器地址中。转移目标是指令 $I_k$。根据图 5-15，在第 1 个周期提取转移指令，然后在第 2 个周期对其进行译码，在第 3 个周期计算目标地址。因此，在用目标地址更新程序计数器后，在第 4 个周期提取指令 $I_k$。在流水线执行中，在转移指令被译码、确定目标地址之前，第 2 和第 3 个周期就已分别提取了指令 $I_{i+1}$ 和 $I_{i+2}$。这两条指令必须被丢弃，而由此产生的两个周期延迟构成了转移代价（branch penalty）。

转移指令会经常出现。实际上，转移指令大约占大多数程序中动态指令数的 20%（动态数是指实际执行的指令数量，考虑到程序中有些指令由于循环而执行很多次的情况）。由于转移代价有两个周期，所以如果程序中转移指令的频率较高的话会使程序的执行时间增加高达 40%。因此，找到方法来减轻其对性能的影响非常重要。

为了减少转移代价，应该在流水线中较早地计算出转移目标地址。在译码阶段是有可能确定目标地址并更新程序计数器的，而不需要等到计算阶段。这样，指令 $I_k$ 可以提前一个时钟周期提取，将转移代价减少到一个周期，如图 6-10 所示。这时，只有一条指令 $I_{i+1}$ 被错误地提
202 取，因为目标地址是在译码阶段确定的。

为了实现这个变化，必须对图 5-10 中的硬件进行修改。图中的加法器需要在每个周期增加 PC 的值。在译码阶段需要第二个加法器为每条指令计算转移目标地址。当指令译码器确定

指令确实是一条转移指令时，第二个加法器在这个周期结束之前就可计算出目标地址。然后在下个周期就可以使用该目标地址去提取目标指令。

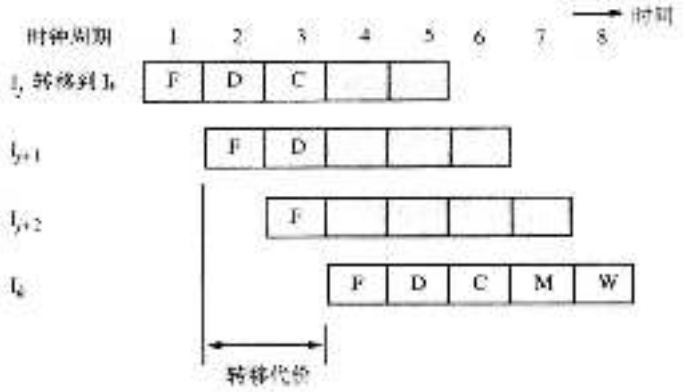



图 6-9 在流水线的计算阶段确定目标地址时的转移代价

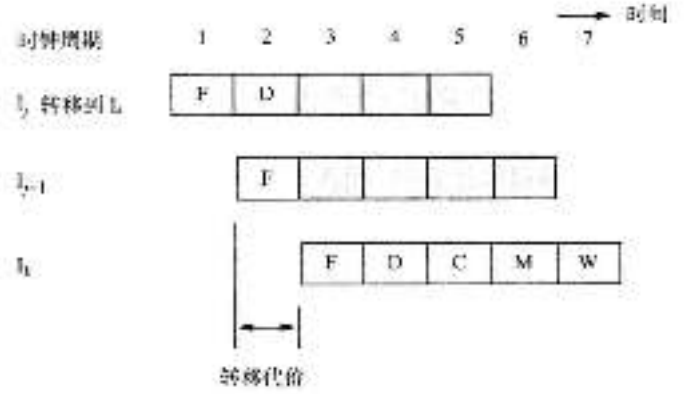



图 6-10 在流水线的译码阶段确定目标地址时的转移代价 203

### 6.6.2 条件转移

考虑一条条件转移指令，比如：

Branch_if_[R5] = [R6] LOOP

这条指令的执行步骤如图 5-16 所示。在第三步中的比较结果决定是否进行转移。

对于流水线来说，必须尽可能早地测试转移条件以减少转移代价。我们刚刚描述了一条无条件转移指令如何在译码阶段确定目标地址。同样，测试转移条件的比较器也可以移到译码阶段，使得在确定目标地址的同时做出条件转移决策。在这种情况下，比较器直接使用寄存器文件的输出端 A 和 B 的值。

将转移决策移到译码阶段保证了所有转移指令都只有一个周期的转移代价。在接下来的两节中，我们将介绍另外的技术来进一步减轻转移对执行时间的影响。

### 6.6.3 转移延迟槽

考虑图 6-11a 所示的程序片段。假设转移目标地址和转移决策在译码阶段确定，同时提取指令 $I_{i+1}$。在评估转移条件后，转移指令可能会导致指令 $I_{i+1}$ 被丢弃。如果条件为真，则在提取正确的目标指令 $I_k$ 前有一个周期的转移代价。如果条件为假，则指令 $I_{i+1}$ 被执行，并且没有转

移代价。在这两种情况下，紧跟在转移指令之后的指令总是被提取。基于这一观察，我们介绍一种减少转移指令代价的技术。

跟在转移指令后面的单元被称为转移延迟槽（branch delay slot）。我们不是有条件地丢弃延迟槽中的指令，而是可以合理安排以使得无论转移是否发生，流水线始终执行这条指令。在延迟槽中的指令不能是 $I_{i+1}$，因为根据转移条件它可能被丢弃。相反，编译器试图找到一条合适的指令占据延迟槽，即使发生转移，这条指令也需要被执行。这可以通过将转移指令之前的一条指令移到延迟槽中来完成。当然，只有在保持被移动指令的数据依赖性的前提下才可以这样做。如果能找到一条有用的指令，那么将不会有转移代价。如果因为数据依赖性的限制而没有有用的指令可以放到延迟槽中，那么必须放一条 NOP 指令在这里。在这种情况下，不管是否发生转移都将有一个周期的时间代价。

```
        Add                 R7, R8, R9
        Branch if [R3]=0    TARGET
        I_{i+1}
        :
TARGET: I_k
```

a）包含条件转移指令的原始指令序列

```
        Branch if [R3]=0    TARGET
        Add                 R7, R8, R9
        I_{i+1}
        :
TARGET: I_k
```

b）将 Add 指令放到转移延迟槽中，它将始终被执行

图 6-11 用一条有用的指令加充转移延迟槽

对于图 6-11a 中的指令，Add 指令可以安全地移到转移延迟槽中，如图 6-11b 所示。即使发生转移，Add 指令也始终被提取并执行。指令 $I_{i+1}$ 只有在不发生转移时才被提取。逻辑上，程序的执行继续进行就像转移指令被放在 Add 指令之后一样。也就是说，与转移指令在指令序列中出现的位置相比，转移的发生要晚一条指令。这种技术称为延迟转移（delayed branch）。

延迟转移技术的有效性取决于编译器重排指令有效填充延迟槽的频率。从许多程序收集到的实验数据表明，70% 以上的情况下，编译器都可以填充一个延迟槽。

### 6.6.4 转移预测

上面的讨论表明在执行转移指令的第 2 个周期做出转移决策可以减少转移代价。但是，即使如此，紧跟在转移指令之后的那条指令仍然在第 2 个周期被提取，并可能被丢弃。提取这条指令的决策实际上是在第 1 个周期做出的，那时 PC 被递增，而转移指令本身正在被提取。因此，为了进一步减少转移代价，处理器需要预期即将取出的指令可能是转移指令，并预测（predict）其结果以确定应在第 2 个周期取出哪条指令。在这一节中，我们首先描述转移预测的不同方法，然后，我们讨论如何在第 1 个周期当转移指令正在被提取的时候做出预测。

204
,
205

#### 1. 静态转移预测

转移预测最简单的形式是假设转移不会发生，并按连续的地址顺序提取下一条指令。如果预测是正确的，则提取的指令就可以完成，并且没有时间代价。但是，如果预测错误，则已提取的指令将被丢弃，正确的转移目标指令被取出。预测错误会引起转移代价。这种简单的方法是静态转移预测（static branch prediction）的一种形式。每次遇到一个条件转移都采用相同的选择（假设转移不会发生）。

如果转移结果是随机的，那么将有一半的条件转移会发生。在这种情况下，始终假设转移不会发生将有 50% 的预测准确率。然而，循环末尾的反向转移大部分时间会发生。对于这样的转移，预测它可能会发生则可以达到更高的准确性。因此，只要一知道转移目标地址就用它来提取指令。同样，对于循环开始处的正向转移，预测转移不会发生将会产生较好的预测准

确性。处理器可以通过检查转移偏移量的符号来做出转移发生或者不发生的静态预测。另外，转移指令的机器编码中可能包含一个位，用来表示应该将该转移预测为发生还是不发生。这个位的设置可以由编译器指定。

**2. 动态转移预测**

为了进一步提高预测准确性，我们可以使用实际的转移行为来影响预测。这就是动态转移预测（dynamic branch prediction）。处理器硬件通过跟踪一条转移指令每次执行时的转移决策来判断该转移发生的可能性。

最简单的形式是动态预测算法可以使用转移指令最近执行的结果。处理器假设下一次执行该指令时，转移决策可能跟上一次相同。此算法可以用图 6-12a 中的 2 状态机来描述。这两个状态是：

LT：可能发生转移

LNT：可能不发生转移

假设算法从状态 LNT 开始。当转移指令执行并且发生转移时，状态机移到状态 LT。否则，保持在状态 LNT 上。下一次遇到相同指令时，如果状态机在状态 LT，预测为会发生转移，否则预测为不发生转移。

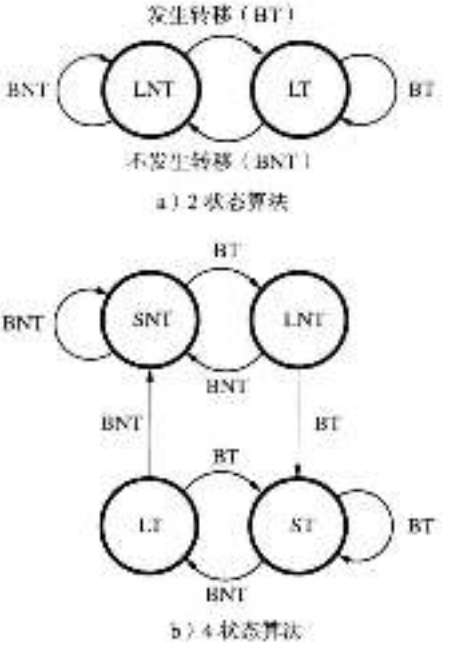



图 6-12 转移预测算法的状态机表示

这种简单的方案在程序循环内部会很好的运行，它只需要用一个位来表示转移指令的执行历史。一旦进入循环，控制循环的转移指令的结果除了循环的最后一遍之外始终是相同的。因此，除了最后一遍，转移指令的每一次预测都是正确的。最后一遍的预测将是错误的，并且转移历史状态机也会改变到相反的状态上。不幸的是，这意味着下一次进入同样的循环时（假设存在多遍循环），第一遍循环状态机会产生错误的预测。因此，同一循环的重复执行在第一遍和最后一遍会产生预测错误。

对执行历史保留更多信息可以获得更好的预测准确性。采用 4 状态的算法如图 6-12b 所示。这 4 个状态是：

ST：极有可能发生

LT：有可能发生

LNT：可能不发生

SNT：极有可能不发生

206
·
207

再假设算法的初始状态设置为 LNT。执行转移指令后，如果确实发生转移，那么状态变化到 ST；否则，状态变化到 SNT。随着程序的执行，多次遇到同样的转移指令时，预测算法的状态会如图所示发生变化。如果状态是 ST 或者 LT，那么预测为会发生转移，否则，预测为不会发生转移。

我们再考虑一下当执行程序循环时所发生的情况。假设转移指令位于循环末尾并且处理器将算法的初始状态设置为 LNT。在循环的第一遍，预测（不发生转移）将是错误的，从而状态会变化到 ST。在后续所有遍中，预测都将是正确的，除了最后一遍。那时，状态会变化到 LT。当再一次进入循环时，在第一遍，预测会发生转移，如果有多次迭代，该预测会是正确的。因此，同一循环的重复执行现在只在最后一遍会产生一次预测错误。

### 3. 动态预测的转移目标缓冲区

在前面的讨论中，我们指出，转移目标地址和转移决策都可以在流水线的译码阶段，也就是指令执行的第2个周期确定。在同一周期中要提取的指令可能是也可能不是转移指令之后要执行的那条。它可能必须被丢弃，在这种情况下，正确的指令将在第3个周期提取。如何利用转移预测来获取更好的性能？

提高性能的关键是提高在第2个周期提取指令的正确率。这只有当转移预测在第1个周期与提取转移指令同时发生时才可以实现。为了做到这一点，处理器需要保留更多的执行历史信息。这些信息通常保存在一个被称为转移目标缓冲区（branch target buffer）的小型、快速的存储器中。

转移目标缓冲区根据指令地址来识别转移指令。随着每条转移指令的执行，处理器在缓冲区中记录指令的地址和转移决策的结果。信息被组织成一个查找表的形式，其中每个表项包括：

- 转移指令的地址
- 转移预测算法的一个或两个状态位
- 转移目标地址

有了这些信息，处理器就能够识别转移指令，并根据所提取指令的地址来获得相应的转移预测状态位。

每次处理器提取了一条新的指令，它就会在转移目标缓冲区查找包含相同指令地址的表项。如果找到含有那个地址的表项，这意味着要提取的指令是一条转移指令。然后处理器能使用状态位去预测是否可能发生转移。同时，目标地址也得到了。这些信息在处理器于第1个周期提取转移指令时就能获得。在第2个周期，处理器使用该转移的预测结果去提取下一条指令。当然，它还必须确定实际的转移决策和目标地址以判断预测值是否正确。如果正确，程序继续执行，没有时间代价。否则，刚已提取到的指令会被丢弃，并在第3个周期提取正确的指令。转移目标缓冲区的主要价值在于转移预测所需的状态信息和转移指令的目标地址都能够在提取转移指令的同时获得。

208

大型程序有许多转移指令。一个有足够存储空间来容纳所有转移指令信息的转移目标缓冲区将是很大的，对它进行快速搜索就会很难。出于这个原因，查找表的大小是有限的，只包含最近执行的转移指令的信息。当其他的转移指令执行时，表中的表项被替换。通常情况下，这个表包含大约1024个表项。

## 6.7 资源限制

流水线使得指令可以重叠执行，但是当没有足够的硬件资源以允许所有的动作同时进行时，流水线便会暂停。在同一时钟周期中，如果两条指令需要访问相同的资源，那么必须暂停一条指令以允许另一条指令使用资源。这种情况可以通过提供额外的硬件来避免。

在一台只有一个高速缓存且每个周期只能访问一次该高速缓存的计算机中会发生这种停顿。如果流水线的取指和访存阶段都连接到高速缓存上，那么不可能同时进行两个阶段的活动。通常情况下，取指阶段在每个周期都会访问高速缓存。然而，当在访存阶段有一条Load或Store指令也需要访问高速缓存时，这个取指阶段的活动必须被暂停一个周期。如果所有被执行的指令中有25%是Load或Store指令，那么这些停顿会使执行时间增加25%。对指令和数据使用分离的高速缓存可以使取指阶段和访存阶段同时进行而没有停顿。

## 6.8 性能评估

对于非流水线处理器，拥有动态指令数 $N$ 的程序其执行时间 $T$ 由公式

$$T=\frac{N\times S}{R}$$

给出，其中 $S$ 是提取并执行一条指令所用时钟周期的平均数，$R$ 是时钟频率（以每秒周期数为单位）。这通常被称为基本性能公式（basic performance equation）。一个有用的性能指标是指令吞吐量（instruction throughput），它是每秒钟执行的指令数量。对于非流水线执行，吞吐量 $P_{np}$ 由公式

$$P_{np}=\frac{R}{S}$$

给出。第5章介绍的处理器使用5个周期来执行所有指令。因此，如果没有高速缓存失效，那么 $S$ 就等于5。

流水线通过重叠执行连续的指令来提高性能，它增加了指令吞吐量，尽管单条指令的执行周期数仍然相同。对于本章描述的5段流水线，每条指令在5个周期内执行，但是理想情况下每个周期都有一条新的指令进入流水线。因此，当不存在停顿时，$S$ 等于1，使用流水线技术的理想吞吐量是

$$P_p=R$$

5段流水线可使吞吐量提高5倍。通常，$n$ 段流水线有可能将吞吐量提高 $n$ 倍。因此，似乎 $n$ 值越高，得到的性能就越强。这会引起两个问题：

- 指令吞吐量的潜在提高实际上能真正实现多少？
- $n$ 值最好为多少？

只要流水线出现暂停或者指令被丢弃，指令吞吐量就会降低到理想值以下。因此，流水线的性能大大受到诸如因指令间的数据依赖性而产生的停顿和因转移而产生的时间代价等因素的影响。高速缓存失效会进一步增加执行时间。我们先讨论这些问题，然后再讨论应该采用多少段流水线的问题。

### 6.8.1 停顿和时间代价的影响

前几节已经定性地研究了停顿和时间开销的影响。现在我们将定量地考虑一下这些影响。

5段流水线包含取指和访存阶段的存储器访问操作和计算阶段的ALU操作。具有最长延迟的操作决定了周期时间和时钟速率 $R$。对于具有片上高速缓存的处理器来说，当所需的指令或数据能在高速缓存中找到时，存储器访问操作的延迟很小，所以通过ALU的延迟可能是关键的参数。如果这个延迟是2 ns，那么 $R$=500 MHz，理想的流水线指令吞吐量是 $P_p$ = 500 MIPS（每秒百万条指令）。

考虑一个具有6.4.1节中所述的操作数转发功能的处理器。这意味着除了Load指令之外，没有因数据依赖性而产生的时间代价。为了评估与高速缓存失效无关的停顿的影响，我们可以考虑一下在Load指令后面紧跟着一条使用存储器访问结果的指令的频率。6.5节解释了在这样的情况下，一个周期的停顿是必需的。虽然理想的流水线执行中 $S$=1，但由于这种Load指令而产生的停顿会将 $S$ 增加 $\delta_{ald}$。例如，假设Load指令占动态指令数的25%，并假设这些Load指令中有40%后面跟着一条依赖的指令。在这样的情况下，需要一个一周期的停顿。因此，将理想情况 $S$=1 增加了

$$\delta_{ald}=0.25\times 0.40\times 1=0.10$$

也就是说，执行时间 $T$ 被增加了 10%，吞吐量被减少到

$$P_p = \frac{R}{1+\delta_{ds1}} = \frac{R}{1.1} = 0.91R$$

编译器可以通过减少 Load 指令后面紧跟一条相关指令的次数来提高性能。每当编译器可以将附近的一条指令安全地移到 Load 指令和依赖的指令之间的位置上时就能消除一个停顿。

现在，考虑一下程序执行期间由于错误预测转移而产生的时间代价。当转移决策和转移目标地址都在流水线的译码阶段确定时，转移代价是一个周期。假设转移指令占程序动态指令数的 20%，并且转移指令的平均预测准确率是 90%。换句话说，所有执行的转移指令中有 10% 会因为预测错误而产生一个周期的时间代价。由于转移代价而导致每条指令的平均周期数增加了

$$\delta_{branch\_penalty} = 0.20 \times 0.10 \times 1 = 0.02$$

较高的预测准确度在限制这个代价对性能的负面影响方面是有利的。

跟 Load 指令有关的停顿和转移预测错误的代价是相互独立的。因此，它们对性能的影响是相加的。$\delta_{ds1}$ 和 $\delta_{branch\_penalty}$ 的和决定了周期数 $S$ 的增加、执行时间 $T$ 的增加以及吞吐量的 $P_p$ 减少。

高速缓存失效对性能的影响可以通过考虑其发生的频率来进行评估。每次发生高速缓存失效时，访问速度较慢的主存储器的时间是暂停流水线 $p_m$ 个周期的代价。所有被提取的指令中有一小部分 $m_i$ 会引起高速缓存失效。所有指令中有一小部分 $d$ 是 Load 或 Store 指令，这些指令中有一部分 $m_d$ 会引起高速缓存失效。由于高速缓存失效而导致理想情况 $S=1$ 增加了

$$\delta_{miss} = (m_i + d \times m_d) \times p_m$$

假设所有被提取指令中有 5% 会引起高速缓存失效，而所有执行指令中有 30% 是 Load 或 Store 指令，并且这些指令中 10% 的数据操作数访问会引起高速缓存失效。假设由于高速缓存失效而访问主存储器的时间代价是 10 个周期。在这种情况下，由于高速缓存失效而导致理想情况 $S=1$ 的增加由

$$\delta_{miss} = (0.05 + 0.30 \times 0.10) \times 10 = 0.8$$

给出。

与数据依赖性的 $\delta_{ds1}$ 和转移预测错误的 $\delta_{branch\_penalty}$ 相比，在这个例子中，由于高速缓存失效而访问慢速的主存储器的影响更为显著。当将所有的因素结合起来时，$S$ 会从理想值 1 增加到 $1 + \delta_{ds1} + \delta_{branch\_penalty} + \delta_{miss}$。高速缓存失效的影响往往是其中占主导地位的一个因素。

211

## 6.8.2 流水线的段数

$n$ 段流水线可以使指令吞吐量提高 $n$ 倍的事实促使我们使用大量的流水线段。然而，随着流水线段数的增加，会有更多的指令并发地执行。因此，指令间会存在更多的可能引起流水线停顿的潜在依赖性。此外，如果较长的流水线将转移决策移到后面的阶段，那么转移代价可能大于一个周期。由于这些原因，$n$ 值的增加所带来的吞吐量增益开始减小，更多段的流水线的成本可能就不再合理了。

另一个重要因素是处理器执行的基本操作中的固有延迟，其中最重要的是 ALU 延迟。在许多处理器中，处理器将完成一项 ALU 操作的时间当作一个时钟周期的大小。其他的操作，包括访问高速缓存的操作，通常分为几步，其中每一步需要的时间大约与完成一个 ALU 操作的时间相同。如果使用流水线 ALU，有可能进一步减少时钟周期时间。最近的一些处理器实现中使用 20 或更多的流水线段来大大减少周期时间。利用现代技术实现这种长流水线可以达

到几 GHz 的时钟速率。

## 6.9 超标量操作

流水线处理器的最大吞吐量是每个时钟周期完成一条指令。一种更先进的方法是使处理器配备有多个执行部件，每个部件都可以被流水，来提高处理器并行处理多条指令的能力。有了这种方案，多条指令可以在同一个时钟周期但在不同的执行部件上开始执行。这种处理器被称为使用了多发操作（multiple-issue）。这种处理器的指令执行吞吐量可以达到每个周期执行多条指令，它们通常被称为超标量（superscalar）处理器。许多现代的高性能处理器就是采用这种方式。

为了实现多发操作的执行，超标量处理器有一个很复杂的取指部件（fetch unit），在需要一条指令之前，取指部件每个周期提取两条或两条以上的指令，并将它们放到指令队列中。一个称为调度部件（dispatch unit）的独立部件，从队列的头部取出两条或两条以上的指令，对它们进行译码，并将它们发送到适当的执行部件中。在流水线的末端，另一个部件负责将结果写入到寄存器文件中。图 6-13 给出了一个具有这种结构的超标量处理器。它包含两个执行部件，一个用于算术指令，另一个用于 Load 和 Store 指令。算术运算一般只需要一个周期，因此第一个执行部件比较简单。因为 Load 和 Store 指令在每次存储器访问之前需要为变址方式计算地址，所以 Load/Store 部件有一个 2 段的流水线。

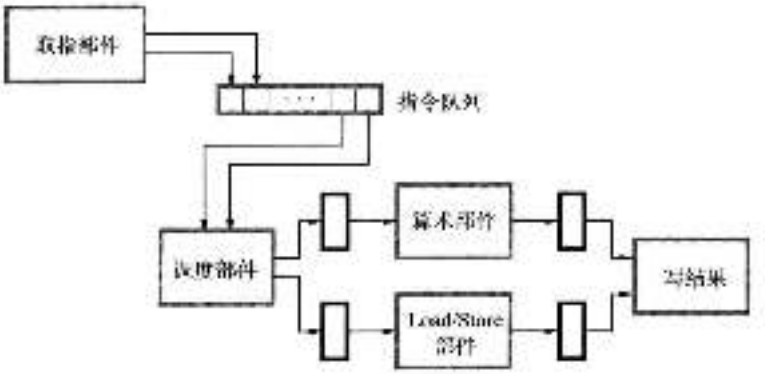



图 6-13 含有两个执行部件的超标量处理器

图 6-13 的结构为寄存器文件提出了一些重要的启示。当一条算术指令和一条 Load 或 Store 指令在同一个周期中被发送给两个执行部件时，它们必须从寄存器文件中获得它们的操作数。现在寄存器文件必须有 4 个输出端口而不是简单流水线中所需要的 2 个输出端口。同样，当一条算术指令和一条 Load 指令在同一个周期中完成时，它们必须将其结果写入寄存器文件中。因此，寄存器文件现在必须有 2 个输入端口而不是简单流水线中的一个输入端口。还有一个潜在的问题是，两条指令可能同时竞争一个寄存器作为写入结果的目标寄存器。通过指令调度，防止两条指令同时写同一个寄存器有可能避免这个问题。否则，必须暂停一条指令，以确保按程序的原指令序列顺序将结果写入目标寄存器。

为了说明图 6-13 中处理器的超标量执行，考虑下面的指令序列：

```
Add         R2, R3, #100
Load        R5, 16(R6)
Subtract    R7, R8, R9
```

```
Store          R10, 24(R11)
```

图 6-14 显示了这些指令的执行方式。取指部件每个周期提取两条指令。在下一个周期中这两条指令被译码，并读取它们的源寄存器。然后，指令被发送给算术部件和 Load/Store 部件。算术运算每个周期都可以开始。Load 或 Store 指令也可以每个周期都开始，因为 Load/Store 部件中的 2 段流水线将 Load 或 Store 指令的地址计算与前面一条 Load 或 Store 指令的存储器访问重叠起来。当指令分别在算术部件和 Load/Store 部件中完成执行时，寄存器文件允许在同一个周期中写入两个结果，因为它们的目标寄存器不同。

212 / 213

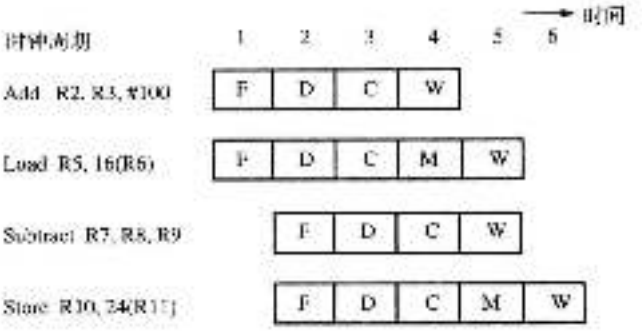



图 6-14 图 6-13 所示处理器中的指令流例子

### 6.9.1 转移和数据依赖性

当不存在任何转移指令以及指令间不存在任何数据依赖性时，可以将能同时发送给不同执行部件的指令进行交错来使吞吐量最大化。然而，程序中可能会包含转移指令和指令间的数据依赖性，超标量处理器必须确保指令能按正确的顺序执行。此外，由于高速缓存失效引起的存储器延迟有时可能会暂停指令的提取和调度。所以，实际吞吐量通常低于理论的最大值。转移指令和数据依赖性所带来的挑战可以用额外的硬件来解决。我们首先考虑转移指令，然后再考虑由数据依赖性所产生的问题。

取指部件在决定将哪些指令放入调度队列中时处理转移指令。它必须确定转移决策和每条转移指令的目标。转移决策可能依赖于一条较早的还在排队的指令或者刚刚被调度的指令的结果。将取指部件暂停直到结果可用会显著降低吞吐量，因此这不是一种可取的方法。因而，最好是采用转移预测的方法。由于我们的目标是实现高吞吐量，所以还可以将转移预测与一种称为推测执行（speculative execution）的技术结合起来。在这种技术中，对基于未经证实的预测选择的后续指令进行提取、调度，甚至执行，但是它们被标记为推测的，以便于当预测不正确时可以将这些指令及其结果丢弃。这个技术需要额外的硬件来维护有关推测执行指令的信息，确保寄存器或存储单元在预测的有效性被证实之前不能被修改。同时还需要额外的硬件来确保后一预测错误，正确的指令也能够被提取和调度。

214

指令间的数据依赖性对指令序列施加了排序约束。一种简单的方法是按顺序将相依赖的指令发送给同一执行部件，在该部件中维持这些指令的顺序。然而，有些时候相依赖的指令可能被发送到不同的执行部件。例如，一条被发送到图 6-13 中 Load/Store 部件的 Load 指令，其结果可能被一条被发送到算术部件的 Add 指令所需要。因为这些部件独立操作，并且可能已经将其他的指令发送到这些部件中了，所以没有办法保证 Add 指令需要的结果是由 Load 指令产生的。这就需要一种机制来确保一条依赖指令会等待它的操作数变得可用。当一条指令被发送给一个执行部件时，它被缓存起来，直到所有必需的来自其他指令的结果都产生为止。这样

的缓冲区被称为保留站（reservation station），它们用于保存与每条被调度指令有关的信息和操作数。每个执行部件的结果被广播到所有的保留站，其中每个结果都被打上了寄存器标识符的标签。这使得保留站可以识别一条被缓冲指令所依赖的结果。当有配对的标签时，硬件将结果拷贝到包含该指令的保留站中。只有当被缓冲的指令得到了它所有的操作数时，控制电路才开始执行它。

在使用多发操作的超标量处理器中，争领所带来的负面影响比在单发操作的流水线处理器中更加明显。编译器能够通过合理的选择和指令排序来避免许多停顿。例如，对于图 6-13 中的处理器，编译器应该尽量将算术指令和存储器指令交错开来。这使得调度部件能够保持两个部件在大部分时间都处于工作状态。

### 6.9.2 无序执行

图 6-14 中的指令按照它们在程序中出现的顺序进行调度。然而，指令的执行可能是无序完成的。例如，Subtract 指令写入寄存器 R7 的操作与之前的 Load 指令写入寄存器 R5 的操作在同一个周期中。如果 Load 指令的存储器访问需要多个周期完成，那么 Subtract 指令的执行将在 Load 指令之前完成。这种情况会带来问题吗？

我们已经讨论过指令之间的依赖性所产生的问题。例如，如果指令 $I_{j+1}$ 依赖指令 $I_j$ 的结果，但是当需要该结果的时候如果它还无法得到，那么 $I_{j+1}$ 的执行就被延迟。只要这种依赖性能够被正确处理，就不必延迟无关指令的执行。如果一对指令间没有依赖性，则指令完成的顺序就无关紧要。

然而，在考虑到指令可能会引起异常时就又出现了新的困难。例如，图 6-14 中的 Load 指令可能会试图进行一次非法的未对齐的存储器访问来获取其数据操作数。当识别到这个非法操作的时候，Load 指令之后的 Subtract 指令可能已经修改了它的目标寄存器。现在程序的执行就会出现不一致的状态。在原序列中发现了导致异常的指令，但是该序列中的一条后续指令已经执行完成。如果允许这种状况的发生，就说该处理器有不精确异常（imprecise exception）。

另一种称为精确异常（precise exception）的方法需要额外的硬件。为了保证异常发生时状态的一致性，指令的执行结果必须严格按程序顺序写入到目标位置。这意味着必须延迟图 6-14 中 Subtract 指令写入寄存器 R7 的操作，直到 Load 指令的寄存器 R5 被更新之后 Subtract 指令才写入寄存器 R7。这样，图 6-13 中的算术部件必须保留 Subtract 指令的结果，或者将结果缓存在一个临时寄存器中，直到前面的指令已经写入了它们的结果。如果在指令执行期间发生异常，那么所有后续指令及其被缓存的结果都将被丢弃。

在外部中断的情况下很容易提供精确异常。当接收到一个外部中断时，调度部件停止从指令队列中读取新的指令，丢弃队列中剩余的指令。所有未执行完的指令继续执行。此时，处理器和其所有的寄存器都处于一致的状态，可以开始对中断进行处理了。

### 6.9.3 执行完成

为了提高性能，应该允许执行部件执行其操作数已在其保留站中就绪的任何指令。这可能导致指令的无序执行。然而，指令必须按程序顺序完成以允许精确异常。这两个看上去冲突的要求可以通过使用临时寄存器解决。允许指令无序执行但是结果要写到临时寄存器中，随后临时寄存器的内容被按正确的程序顺序传送到永久寄存器中。最后这一步通常被称为提交（commitment）步，因为从这点之后指令的影响不能再被恢复。如果一条指令导致异常，由于任何已经执行的后续指令的结果此时仍然保存在临时寄存器中，所以可以被安全地丢弃。通

常，需要正常写入存储器中的结果也会被临时缓冲，并且它们也可以被安全地丢弃。

为保存指令结果而分配的临时寄存器承担永久寄存器的作用，其中缓存着永久寄存器应该保存的数据。在此期间，临时寄存器的内容被转发给任何指向原来的永久寄存器的后续指令。这项技术被称为寄存器重命名（register renaming）。临时寄存器可能和永久寄存器一样多，也可能比永久寄存器少，它们可以根据需要分配，与不同的永久寄存器联系起来。

当允许无序执行时，需要一个专门的控制部件来保证指令按正确的顺序提交。这个部件被称为提交部件（commitment unit）。它使用一个称为重排序缓冲器（reorder buffer）的单独队列来决定下一次应该提交哪条（哪些）指令。随着指令不断地被调度执行，指令严格地按照程序顺序进入队列。当一条指令到达队首并且该指令的执行已经完成时，相应的结果从临时寄存
216 器传送到永久寄存器中，指令从队列中移除。释放所有分配给该指令的资源，包括临时寄存器。这时就说该指令已经被释放（retire）了。因为只有当指令位于队首时才会被释放，在它之前调度的所有指令也一定都被释放了。因此，指令可以按照无序方式结束执行，但必须按照程序的顺序来释放。

### 6.9.4 调度操作

我们现在讨论调度操作。当作出调度决策时，调度部件必须确保已经具备了指令执行所需要的所有资源。例如，由于指令的执行结果可能需要写到临时寄存器中，所以应该有一个可用的寄存器，并且保留出来用作那条指令进行调度操作的一部分。另外，在相应执行部件的保留站中必须有可用的空间。最后，在重排序缓冲器中也必须为该指令留出以后提交结果的位置。当分配完所需要的所有资源时，便可以调度这条指令了。

指令可以按照无序方式进行调度吗？例如，考虑图 6-14 中的 Load 指令，先前被调度指令的高速缓存失效会使得 Load/Store 部件的保留站中没有空间，从而导致 Load 指令的调度被延迟。那么此时可以调度 Subtract 指令吗？理论上这是可以的，只要为 Load 指令保留它需要的所有资源，包括在重排序缓冲器中的位置。这对确保所有指令最终按正确的顺序释放并且不会发生死锁来说是非常重要的。

死锁（deadlock）是当两个部件 A 和 B 使用同一共享资源时出现的一种状态。假设 B 部件要等到 A 部件完成后才能完成它的操作。同时，A 部件所需要的资源已经分配给了 B 部件。如果这种情况发生，那么两个部件都不能完成其操作。A 部件等待它所需要的资源，而该资源正被 B 部件占有。同时，B 部件在完成其操作并释放该资源之前要等待 A 部件完成。

当按无序方式调度指令时，考虑只有一个临时寄存器的超标量处理器作为死锁的例子。当图 6-14 中的 Subtract 指令在 Load 指令之前被调度时，将临时寄存器保留给 Subtract 指令使用。Load 指令由于正在等待同一个临时寄存器而不能被调度，而该临时寄存器直到 Subtract 指令释放后才会空闲。又由于 Subtract 指令不能在 Load 指令之前释放，所以产生死锁。

为了预防死锁，调度部件必须考虑许多因素。因此，无序地发出指令可能会大大增加调度部件的复杂度。这也意味着可能需要更多的时间作出调度决策。有序地调度指令可以避免这种复杂性。在这种情况下，指令被调度的时刻和被释放的时刻都是按照它们在程序中的顺序进行的。在这两个时刻之间，只要没有指令间的内部依赖性的影响，多个执行部件中几条指令的执行都可以无序地进行。

超标量处理器的最后一个问题是关于执行部件的数量。图 6-13 中的处理器有一个算术部件和一个 Load/Store 部件。为了获得更高的性能，现代的超标量处理器通常都有两个用于整数
217 操作的算术部件，以及一个独立的用于浮点操作的算术部件。浮点部件有它自己的寄存器文

件。许多处理器还包含一个用于整数或浮点运算的向量部件，它通常可以并行执行2到8个操作。这样的部件也可能有其专用的寄存器文件。一个Load/Store部件通常支持整数、浮点或向量部件的所有存储器访问。为了使许多执行部件保持忙碌，现代处理器可以同时提取4条或更多的指令放到指令队列的尾部，同样，可以从指令队列的头部调度4条或更多的指令发送给执行部件。

## 6.10 CISC处理器中的流水线

RISC处理器的指令集使得流水线相对容易实现。所有的指令都是一个字的长度，操作数信息通常位于不同指令字中相同的位置。没有指令需要多个存储器操作数。只有Load和Store指令会访问存储器操作数，通常只使用变址寻址方式。所有其他的指令都对寄存器操作数进行操作。本章所描述的5段流水线就是迎合RISC风格指令的这些特征设计的。

CISC处理器中，由于指令的长度可变、有多个存储器操作数和复杂的寻址方式，以及条件码的使用，使得在CISC处理器中实现流水线出现困难。占多个字的指令可能需要多个周期来提取。此外，指令长度和格式的可变使得译码和操作数访问跟超标量处理器中调度队列的管理一样复杂。

执行指令时，更复杂的寻址方式（如自动增量方式或自动减量方式）会产生副作用（side effect）。当除了目标操作数单元之外的另一个单元也受到影响时，就会发生副作用。例如，指令

```
Move   R5, (R8)+
```

有副作用。不仅目标寄存器R5受影响，源寄存器R8也受自增操作的影响。如果后面的指令依赖于寄存器R8中的值，这种依赖性必须像对待涉及目标寄存器R5的依赖性一样用额外的硬件来处理。它可能需要暂停流水线或者转发新的值。在超标量处理器中，这种依赖性需要使用6.9.3节中讨论的临时寄存器和寄存器重命名来处理。

条件码也会产生副作用。例如，在下列的指令序列中

```
Compare        R7, R8
Branch>0       TARGET
```

Compare指令的结果对条件码标志的影响是一个副作用，而Branch指令隐含地依赖于这个副作用。条件码寄存器可以相对容易地包含在如图6-2所示的简单流水线中，因为在任何周期中只执行一个ALU操作。然而，在含有多个执行部件的超标量处理器中，许多指令可能会处于 218
执行的不同阶段，每个周期可能会执行两个或两个以上的ALU操作。与条件码有关的副作用所产生的依赖性需要使用额外的临时寄存器和寄存器重命名来处理。

最后，考虑下列CISC风格的指令序列：

```
Move           (R2), (R3)
Move           (R4), R5
```

第一条Move指令需要对存储器进行两次操作数访问，而第二条Move指令只需要一次。如图6-2所示的流水线中执行这些指令需要额外的硬件来暂停第二条Move指令，以便于第一条Move指令能完成它对存储器的两次操作数访问。在如图6-13所示的超标量处理器中，Load/Store部件也同样必须暂停它的内部流水线。

CISC风格的指令使流水线变得复杂。这是开发RISC方法的主要原因之一。尽管如此，在流水线开始广泛使用前就引入的CISC风格指令集也已经有为其实现了流水线的处理器。附录C和附录E中讨论了基于ColdFire和Intel指令集的处理器的例子。ColdFire处理器主要用

于嵌入式应用，而Intel处理器提供通用的需求。因此，ColdFire处理器中使用流水线的程度低于Intel处理器。

### 6.10.1 ColdFire处理器中的流水线

V1和V2版本的ColdFire处理器实现中有两个流水线，它们之间由一个先进先出（FIFO）缓冲区相连。2段的指令提取流水线将指令预取到缓冲区中，然后缓冲区向执行指令的2段流水线提供指令。只包含寄存器操作或者寄存器–存储器操作的指令通过这两个执行阶段一次。包含存储器–寄存器操作或者存储器–存储器操作的指令必须通过这两个执行阶段两次。

更高版本的ColdFire处理器实现中在两个流水线间使用类似的缓冲区结构，但为了更高的性能，这些版本中包含了各种增强的功能。例如，V4版本中的指令提取流水线被扩展为4个流水段，并包含转移预测。执行流水线被扩展为5个流水段，其中前边的流水段用于地址计算，后边的流水段用于算术/逻辑运算。这种功能的分离提供了一种受限的超标量处理的形式。比如在某些情况下，Move指令和另一条指令可以在同一个周期内被分发到执行流水线上。V5版本的处理器实现中有两个独立的基于V4结构的执行流水线，它们提供真正的超标量处理。

### 6.10.2 Intel处理器中的流水线

219 Intel处理器使用超标量执行和深流水线来实现高性能。例如，Core 2和Core i7体系结构有4条指令的多发宽度和一个14段的流水线。使用了转移预测、寄存器重命名、无序执行和其他的技术。

为了减少内部的复杂性，CISC风格的指令由硬件动态地转换成简单的RISC风格的微操作。然后这些微操作被分发到执行部件以完成原来CISC风格指令所指定的任务。这种方法保留了代码的兼容性，同时使其能够使用为RISC风格指令集开发的可以大大提升性能的技术。在某些情况下，为了更有效地处理，可将一些微操作融合为宏操作。例如，在一个包含原来CISC风格指令的程序中，一条影响条件码的比较指令后面往往是一条转移指令。硬件可能首先将比较指令和转移指令转换成单独的微操作，但之后又将它们融合成一个综合的比较-转移操作。该操作的功能反映了RISC风格指令集中的典型特征。

## 6.11 结束语

本章介绍了性能提升的两个重要特性：流水线和指令多发。流水线可使处理器的指令吞吐量达到每个时钟周期一条指令。与流水线结合的指令多发使得指令吞吐量为每个时钟周期多条指令的超标量操作成为可能。

只有认真解决以下三个方面的问题，才可以实现性能的提高：

- 处理器的指令集
- 流水线硬件的设计
- 相关编译器的设计

认识到三者之间的相互作用是非常重要的。在一个处理器的设计过程中，对这三者之间相互作用的认识程度决定性地影响着处理器的性能水平。特别适用流水线执行的指令集是现代处理器的关键特征。

关于本章所讨论主题的更多细节请参考其他材料，参考文献[1]涵盖了流水线的内容，参考文献[2]涵盖了超标量处理器的内容。

## 6.12 问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。 220

**例 6.1**

问题：考虑下列指令序列的流水线执行：

| | |
|---|---|
| Add | R4, R3, R2 |
| Or | R7, R6, R5 |
| Subtract | R8, R7, R4 |

寄存器 R2 和 R3 的初始值分别为 4 和 8，寄存器 R5 和 R6 的初始值分别为 128 和 2。假设流水线提供从寄存器 RY 和 RZ（参见图 5-8）到 ALU 的转发通路。在第 1 个周期提取第 1 条指令，在后续的周期中提取剩下的指令。

假设处理器使用操作数转发，画出类似于图 6-1 的图，显示这些指令的流水线执行。然后，参考图 5-8，描述第 4 到第 7 个周期中寄存器 RY 和 RZ 的内容。

**解答：**此例中存在涉及寄存器 R4 和 R7 的数据依赖性。在这两个寄存器的新值被写入寄存器文件之前 Subtract 指令就要用到它们。因此，当 Subtract 指令处于流水线的计算阶段时，需要将这两个值转发到 ALU 的输入端。图 6-15 给出了带转发的流水线执行情况。其中一个箭头表示从寄存器 RZ 转发的寄存器 R7 的新值，另一个箭头表示从寄存器 RY 转发的寄存器 R4 的新值。

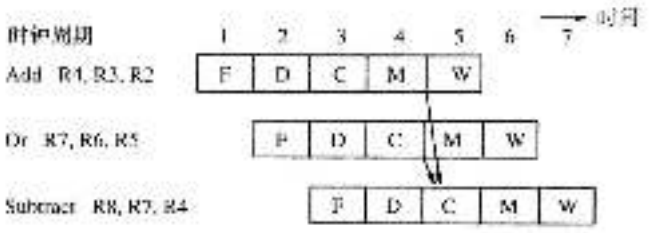



图 6-15 例 6.1 中指令的流水线执行

至于第 4 到第 7 个周期中寄存器 RY 和 RZ 的内容，下面的描述提供了答案。

- Add 指令使用寄存器 R2 和 R3 的初始值，在第 3 个周期产生结果 12。在第 4 个周期中，可在寄存器 RZ 中得到这个结果。在第 4 个周期中，寄存器 RY 的值是 Add 指令之前一条未指定指令的结果。
- 在第 4 个周期中，Or 指令产生结果 130。该结果被放入寄存器 RZ 中，以便于在第 5 个周期中使用。而在第 5 个周期，Add 指令的结果 12 在寄存器 RY 中。
- 在第 5 个周期中，Subtract 指令处于计算阶段。为了产生正确的结果，使用转发来提供寄存器 RY 中的值 130 和寄存器 RZ 中的值 12。ALU 的结果是 130-12=118。第 6 个周期可在寄存器 RZ 中 221
  得到这个结果。第 6 个周期 Or 指令的结果 130 在寄存器 RY 中。
- 在第 6 个周期中，Subtract 指令处于访存阶段。Subtract 指令后面的未指定指令正在计算阶段产生结果。在第 7 个周期中，未指定指令的结果在寄存器 RZ 中，Subtract 指令的结果在寄存器 RY 中。

**例 6.2**

问题：假设一个程序所执行的动态指令中有 20% 是转移指令。没有由于数据依赖性而产生的流水线延迟。使用静态的转移预测，并假设不会发生转移。

（a）当发生 30% 的转移和发生 70% 的转移时，确定这两种情况的执行时间。

（b）确定一种情况相对于另一种情况的加速比。将加速比表示为相对于 1 的百分比。

**解答：**6.8.1 节描述了考虑转移代价影响的 $\delta_{stall\_pmb}$ 计算。

（a）对于第一种情况，$\delta_{stall\_pmb}$ 的值是 $0.20 \times 0.30 = 0.06$。对于第二种情况，$\delta_{stall\_pmb}$ 的值是 $0.20 \times 0.70 = 0.14$。使用 $S = 1/\delta_{stall\_pmb}$，第一种情况的执行时间是 $(1.06 \times N)/R$，第二种情况的执行时间是 $(1.14 \times N)/R$。

（b）因为第一种情况的执行时间比较少，所以性能改善的加速百分比是

$$\left(\frac{1.14}{1.06}-1\right)\times 100\text{–}7.5\%$$

## 习题

[M] 6.1 考虑下列在存储器中给定地址处的指令：

```
1000    Add         R3, R2, #20
1004    Subtract    R5, R4, #3
1008    And         R6, R4, #0x3A
1012    Add         R7, R2, R4
```

222

寄存器 R2 和 R4 的初始值分别为 2000 和 50。这些指令在一台含有图 6-2 所示的 5 段流水线的计算机上执行。在第 1 个时钟周期取第 1 条指令，在后续的周期中提取剩下的指令。

（a）画出类似于图 6-1 的图，表示通过流水线的指令流。描述在第 1 到第 8 个时钟周期中每一个流水段所执行的操作。

（b）参考图 5-8 和图 5-9，描述第 2 到第 8 个周期中流水线中的寄存器 IR，PC，RA，RB，RY 和 RZ 的内容。

[M] 6.2 针对下列程序：

```
1000    Add         R3, R2, #20
1004    Subtract    R5, R4, #3
1008    And         R6, R3, #0x3A
1012    Add         R7, R2, R4
```

重复习题 6.1 中的问题。假设流水线提供从图 5-8 中的寄存器 RY 和 RZ 到 ALU 的转发通路，并假设处理器使用操作数转发。

[M] 6.3 考虑图 2-8 所示程序中的循环。假设它在一个 5 段的流水线上执行，该流水线具有从图 5-8 中的寄存器 RY 和 RZ 到 ALU 的转发通路。假设流水线使用静态转移预测，并假设不会发生转移。画一个类似于图 6-1 的图，表示该循环连续两次迭代的执行情况。

[D] 6.4 重复习题 6.3 中的问题，但像编译器那样首先重新排列指令以优化性能。

[D] 6.5 对于使用包含有一个延迟槽的延迟转移技术的流水线，重复习题 6.3 中的问题。根据需要重新排列指令以提高性能。

[M] 6.6 图 6-5 中的转发通路允许寄存器 RZ 的内容直接在 ALU 运算中使用。运算的结果存到寄存器 RZ 中，并替换它先前的内容。本题要求在多个周期中跟踪寄存器 RZ 的内容。考虑两条指令

```
I1: Add       R3, R2, R1
I2: LShift1   R3, R3, #1
```

当在第 1 个周期提取指令 $I_1$ 时，一条先前提取的指令正在执行 ALU 运算，给出一个结果 17。然后，当在第 2 个周期对指令 $I_1$ 进行译码时，另一条先前提取的指令正在执行 ALU 运算，给出一个结果 198。还是在第 2 个周期，寄存器 R1，R2 和 R3 的值分别为 30，100 和 45。使用这些信息，画一个时序图，显示在第 2 到第 5 个周期中寄存器 RZ 的内容。

[M] 6.7 假设一个程序所执行的动态指令中有 20% 是转移指令。使用包含有一个延迟槽的延迟转移技术。

223

假设没有其他因素造成的延迟。首先，如果所有的延迟槽都用 NOP 指令填充，推导出执行时间（以周期为单位）的表达式。然后，如果 70% 的延迟槽被优化编译器填充了有用的指令，推导出另一个反映执行时间的表达式。从这些表达式中，确定编译器对性能增强的影响，表示成加速百分比的形式。

[D] 6.8 对于含有两个转移延迟槽的流水线处理器，重复习题 6.7 中的问题。优化编译器的输出使得第一

个延迟槽70%的时间用有用的指令填充，但第二个延迟槽仅10%的时间用有用的指令填充。将这种情况下编译器优化后的执行时间与习题6.7中编译器优化后的执行时间进行比较。假设两个处理器具有相同的时钟速率。指出哪一个处理器/编译器组合比较快，并指出速度较快的那个组合的加速百分比。

[D] 6.9 假设一个程序所执行的动态指令中有20%是转移指令。再假设75%的转移指令实际上发生了转移。该程序在两个具有相同时钟速率的不同处理器上执行。一个处理器使用假设转移不成功方法的静态转移预测。另一个处理器使用基于图6-12a中状态的动态转移预测。以6.6.4节中描述的方式使用转移目标缓冲区。

(a) 没有其他原因产生流水线延迟时，使用动态转移预测的处理器与使用静态转移预测的处理器表现相同时其预测准确率的最小值必须是多少？

(b) 如果动态预测准确率实际为90%的话，那么相对于使用静态预测的加速比是多少？

[M] 6.10 如图6-5那样转发寄存器RZ的值需要在流水线中增加额外的控制逻辑。该额外的逻辑必须检查什么具体条件来确定在流水线的计算阶段向ALU输入端提供数据的多路复用器的设置？

[M] 6.11 对于将图5-8中寄存器RY的内容转发给向ALU输入端提供数据的多路复用器，重复习题6.10中的问题。

[D] 6.12 作为习题6.10和6.11的延续，考虑下列的指令序列：

```
Add         R3, R2, R1
Subtract    R5, R5, R4
Or          R8, R3, #1
```

描述在这种情况下处理转发的方式。应该如何修改习题6.10和习题6.11中的条件？

[M] 6.13 有一个程序，由4条存储器访问指令和4条算术指令组成。假设指令间没有数据依赖性。该程序的两个版本在图6-13所示的超标量处理器上执行。第一个版本将4条存储器访问指令按顺序排列，随后是4条算术指令。第二个版本将存储器访问指令与算术指令交错开来。画两个类似于图6-14的图来比较该程序两个版本的执行情况。 224

[E] 6.14 假设一个程序不包含转移指令，它在图6-13所示的超标量处理器上执行。如果程序指令由75%的算术指令和25%的存储器访问指令组成，那么预期的最佳执行时间是多少个周期？如果使用相同的时钟，将这个时间与图6-2中简单处理器上的最佳执行时间进行比较。

[M] 6.15 假设程序指令由15%永远不会发生转移的转移指令、65%的算术指令和20%的存储器访问指令组成，重复习题6.14中的问题，为图6-2和图6-13中的处理器找出可能的最佳执行时间。假设所有转移指令的预测准确率为100%。

[E] 6.16 有一个处理器，使用图6-12b所示的转移预测方案。该处理器的指令集增加了一个功能，就是可以使编译器为每条转移指令指定初始预测状态为LT或者LNT。处理器在执行程序时，如果在转移目标缓冲区中没有找到有关转移指令的信息时就使用这个初始状态。当为下列两种情况生成代码时，讨论编译器如何使用这项功能。

(a) 有一个循环，在其末尾有一条转移到循环开始处的条件转移指令。

(b) 有一个循环，在其开始处有一条转移到循环出口的条件转移指令，在其末尾有一条转移到循环开始处的无条件转移指令。

[M] 6.17 假设一个处理器具有习题6.16所描述的功能，可以为转移指令指定初始预测状态。考虑一个语句格式

```
IF A > B THEN A = A + 1 ELSE B = B + 1
```

(a) 为上面的语句生成汇编语言代码。

(b) 在没有任何其他信息的情况下，讨论编译器如何用汇编代码为转移指令指定初始预测状态。

(c) 对含有上述语句程序的执行行为的研究表明，变量A的值通常大于变量B的值。如果编译

器可以得到这些信息，讨论这些信息如何影响该转移指令的初始预测状态。

[M] 6.18 考虑一个语句形式

IF A > B THEN A = A + 1 ELSE B = B + 1

（a）考虑一个具有图 6-2 所示流水线结构的处理器，它使用转移不成功假设的静态转移预测。为上述语句编写汇编语言代码。画出类似于图 6-1 的图，以显示不同转移决策时指令的流水线执行情况，并确定执行时间（以周期为单位）。

（b）现在假设使用延迟转移技术，为上述语句编写汇编语言代码。画图显示采用不同转移决策时指令的流水线执行情况，并将执行时间（以周期为单位）最能一种情况中的时间进行比较。

225

## 参考文献

1. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th edition, Morgan Kaufmann, Burlington, Massachusetts, 2009.

2. J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill, New York, 2005.

226

# 输入/输出组织结构

## 本章目标

在本章中你将学习以下内容：

- 访问I/O设备所需的硬件
- 同步与异步总线操作
- 接口电路
- 商业标准，如USB、SAS和PCI Express



计算机的一个基本特性是具有与I/O设备进行数据传送的能力。这一通信能力使操作员可以完成很多操作，例如，用键盘和显示屏幕来处理文本和图形。实际中，我们广泛地使用计算机通过Internet与其他计算机通信并访问全球信息。此外，在嵌入式应用中，计算机虽然不是很直观，但仍有着同等重要的作用。它们是家用电器、测量设备、车辆系统、手机、银行和销售点终端的集成部分。在这些应用中，计算机的输入可能来自触摸板、传感器开关、数字照相机、麦克风或火警报警器；输出可能是需要显示的字符或数字，发送给扬声器的声音信号或用来改变发动机速度、打开阀门或使机器人按指定方式移动的数字编码命令。

计算机应该具备与各种各样的设备交换信息的能力。在很多情况下，处理器全程参与到这些交换活动中。但是，数据传输也可以直接在I/O设备（比如磁盘）和主存之间进行。处理器只需最低限度地参与数据传输过程。这种可能性将在下一章存储器系统中进行探讨。

第3章介绍了如何从程序员的角度看待在处理器与I/O设备接口寄存器之间发生的输入/输出数据传输操作。在本章中，我们将讨论为实现此类传输所需的硬件细节。

互连网络被用来在处理器、存储器和I/O设备之间传输数据。下面我们将描述一种常用的称为总线（bus）的互连网络。

## 7.1 总线结构

图7-1所示的总线是实现图3-1中互连网络的一种简单结构。在任一时刻都只能有一对源/目的单元使用该总线来传输数据。

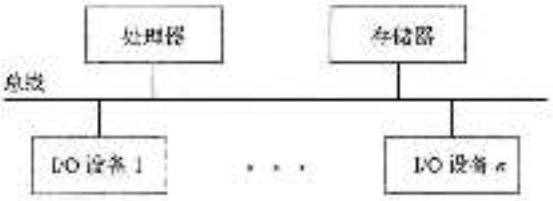



图7-1 单总线结构

总线由三组线路组成，分别用来传输地址、数据和控制信号。I/O设备接口连接到这些线路上，如图7-2所示的输入设备与这些线路的连接。每一个I/O设备接口中的寄存器都被分配了唯一的一组地址。当处理器将一个特定的地址放置到地址线上时，总线上所有设备的地址译码器都会检查这个地址。识别出这个地址的设备就会对控制线上的命令做出响应。处理器使用控制线来请求一个读或者写操作，所请求的数据则在数据线上传输。



如3.1节所述，当I/O设备和存储器共享同一个地址空间时，这种方式被称为存储器映射I/O（memory-mapped I/O）。任何能够访问存储器的机器指令也都可以用来与I/O设备进行数据传输。例如，如果图7-2中的输入设备是一个键盘，且DATAIN是键盘的数据寄存器，则指令

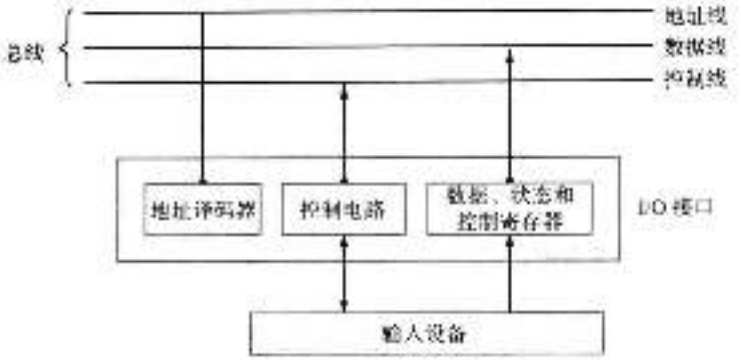



图 7-2 输入设备的 I/O 接口

Load R2, DATAIN

将从 DATAIN 中读取数据并将其存储到处理器寄存器 R2 中。类似地，指令

Store R2, DATAOUT

将寄存器 R2 的内容发送到 DATAOUT 中，DATAOUT 可能是显示设备接口的数据寄存器。状态寄存器与控制寄存器包含与 I/O 设备操作有关的信息。地址译码器、数据寄存器和状态寄存器以及协调 I/O 传输所需的控制电路构成了设备的接口电路。

## 7.2 总线操作

总线需要一组通常被称为总线协议（bus protocol）的规则，用来管理各种设备如何使用总线。总线协议决定设备何时可以将信息放到总线上，何时可以将总线上的数据装入设备的寄存器中，等等。这些规则将通过控制信号来实现，控制信号指示什么时候可以采取什么样的
[229] 动作。

一根控制线，通常标记为 R/W，指定要执行读还是写操作。如同这个标签所指示的，当它被置为 1 时表示进行读操作，被置为 0 时表示进行写操作。在允许传输不同长度的数据时，如字节、半字或字，所需的长度由其他的控制线来表示。总线的控制线还要传送时序信息。这些信息详细说明处理器和 I/O 设备何时可以将数据放到数据线上或从数据线上接收数据。目前已设计出多种数据传输时序的方式，大致可以分为同步和异步两种。

在任何数据传输操作中都有一台设备扮演主控器或主控设备（master）的角色。它通过在总线上发送读或写命令来启动数据传输。通常，处理器是总线主控器，但在 7.3 节中我们将看到，其他设备也可以成为总线主控器。被主控设备寻址的设备称为从动设备（slave）。

### 7.2.1 同步总线

在同步（synchronous）总线上，所有设备都从一根叫做总线时钟（bus clock）的控制线上获取时序信息，如图 7-3 的顶部所示。该控制线上的信号有两个相位（phase）：一个高电平和一个紧跟着的低电平。两个相位组成了一个时钟周期（clock cycle）。时钟周期的前半部分，即电平的低到高转换与高到低转换之间的那部分通常被称为时钟脉冲。

在图 7-3 中的地址线和数据线显示好像它们可以同时传送高电平和低电平信号。有些线上
[230] 是高电平有些线上是低电平。这是一种通用的约定表示方式，具体依赖于正在传输的特定地址和数据值。交点表示的是这些模式发生变化的时间。处于高低电平之间的中间电平的信号线表示信号不可靠的时段，所有设备必须忽略这段时间的信号。

我们来看一下在一次输入（读）操作期间信号事件发生的顺序。在 $t_0$ 时刻时，总线主控器将设备地址放到地址线上并在控制线上发送一条表示读操作的命令。这个命令还可以指明将要读取的操作数的长度。信息在总线上的传输速度取决于总线的物理和电气特性。时钟脉冲的宽度 $t_1 - t_0$ 必须大于总线上的最大传播延迟。而且，该时间要长到保证所有设备对地址和控制信号进行译码，从而使得被寻址的设备（从动设备）能够在时间 $t_1$ 进行响应，将请求的输入数据放到数据线上。该时钟周期结束时，也就是在时间 $t_2$ 时，主控设备将数据线上的数据装入它的某一个寄存器中。要使数据能够正确地装入寄存器中，数据的有效时间必须大于寄存器的准备时间（参见附录 A）。因此，$t_2 - t_1$ 这段时间必须大于总线上的最大传输时间与主控设备寄存器的准备时间之和。

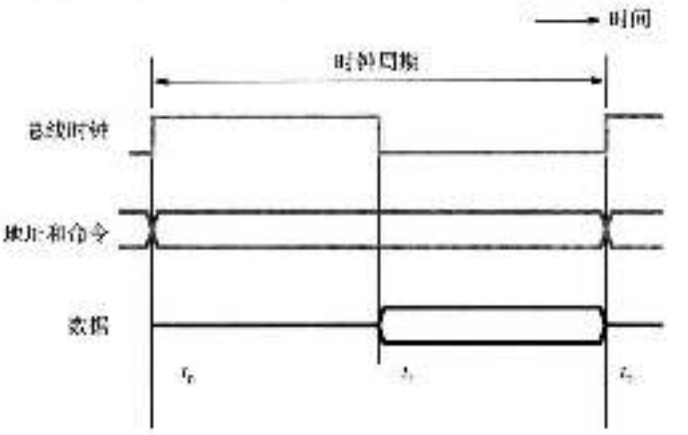



图 7-3　同步总线上的输入传输时序

写操作也有类似的过程。主控设备在发送地址和命令信息的同时也将输出数据放置到数据线上。在时间 $t_2$，被寻址的设备将数据装入自己的数据寄存器中。

图 7-3 中的时序图是总线线路上所发生动作的一种理想化表示。实际中信号变换状态的确切时间与图中显示的有些差别，因为在总线线路和设备电路上存在着传输延迟。图 7-4 给出了一个更接近实际情况的图。在这幅图中除了时钟以外，每个信号都显示了两个图像。因为信号要花费时间从一台设备传送到另一台设备，所以不同的设备见到同一信号跳变的时间不同。上面的图像显示的是主控设备见到的信号，而下面的图像显示的则是从动设备见到的信号。我们假设连接到总线上的所有设备见到时钟变换的时间相同。系统设计者需要花费相当大的精力来确保时钟信号能满足这一要求。

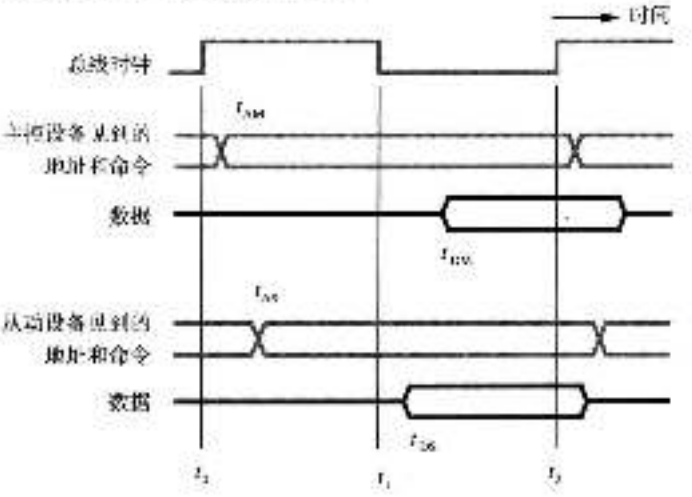



图 7-4　对应图 7-3，更详细的输入传输时序图

主控设备在时钟周期开始的时钟上升沿（$t_0$）发送地址和命令信号。但由于从主控设备到总线线路的电子电路输出端的延迟，在 $t_{AM}$ 之前，这些信号并不真正出现在总线上。很短的时间之后，在 $t_{AS}$ 时刻，信号到达从动设备。从动设备对地址信号进行译码，并在 $t_1$ 时刻发送所请求的数据。同样，数据信号直到 $t_{DS}$ 时刻才出现在总线上。它们向主控设备方向传输，在 $t_{DM}$ 时刻到达。在 $t_2$ 时刻，主控设备将数据装入它的寄存器中。因此，$t_2 - t_{DM}$ 这段时间必须大于主控设备寄存器的准备时间。数据在 $t_2$ 后必须保持有效一段时间，这段时间等于该寄存器的保

持时间（保持时间参见附录A）。

时序图常常只给出图7-3中的简化图，尤其是在介绍数据是如何传输的基本思想时。但是，实际信号总是包含有图7-4所示的延迟。

**多周期数据传输**

按照上面描述的方式我们可以得到设备接口的一种简单设计方式。但是，这种方式有它的局限性。因为每次传输必须在一个时钟周期内完成，时钟周期 $t_3 - t_0$ 必须满足总线上最长的时间延迟和最慢的设备接口。这将迫使所有设备都以最慢设备的速度工作。

此外，处理器也不能确定被寻址的设备是否已经响应。在 $t_3$ 时刻，它简单地假定读操作中的输入数据已在数据线上有效，或写操作中的输出数据已经被I/O设备接收。但如果由于故障，设备不能正常运行，则无法检测到该错误。

为克服这些局限性，大部分总线都有表示设备响应的控制信号。这些信号通知主控设备从动设备已经识别了它的地址并准备好参与数据传输操作了。也可以调整数据传输周期的脉冲宽度来匹配不同设备的响应速度。通常我们会让一个完整的数据传送操作跨越多个时钟周期来实现这一点。因此，不同设备传输数据所需的时钟周期数就有可能不同。

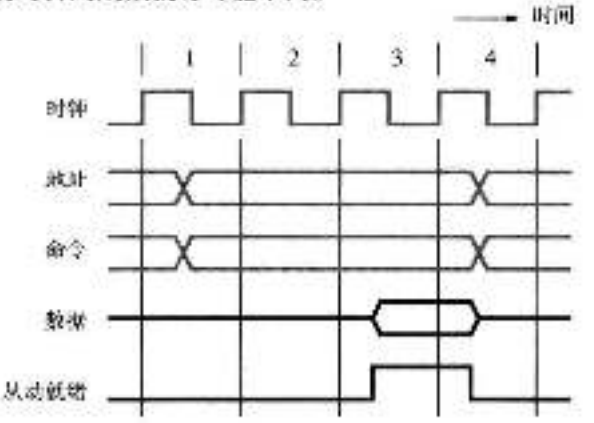



图7-5 多时钟周期的输入传输

图7-5所示的是多时钟周期传输的一个例子。在时钟周期1期间，主控设备将地址和命令信息发送到总线上请求一次读操作。从动设备接收到这个信息并将其译码，并在时钟周期2开始时的时钟上升沿开始访问被请求的数据。我们已经假设在获取数据的过程中存在延迟，所以从动设备不能立即响应。在时钟周期3期间数据已准备好并被放置到总线上。同时，从动设备发出从动就绪控制信号。在该时钟周期结束时，一直在等待这个信号的主控设备把数据装入它的寄存器中。在时钟周期3结束时，从动设备从总线上撤销数据信号并将从动就绪控制信号返回到低电平。至此总线传送操作完成。主控设备可以在时钟周期4发送新的地址和命令信号来启动一次新的传送过程。

从动就绪信号是从动设备对主控设备的应答，它确认被请求的数据已经被放置到总线上了。从动就绪信号还使得不同设备的总线传输脉冲宽度可以不同。在图7-5的例子中，从动设备在周期3响应。其他不同的设备可能在较早或稍后的周期中响应。如果被寻址的设备根本不响应，主控设备等待预先确定的最大时钟周期数后就放弃这次操作。这可能是由于地址错误或设备故障导致的。

现在我们将介绍一种不使用时钟信号的方法。

## 7.2.2 异步总线

控制总线上数据传输的另一种方式是基于主控设备和从动设备之间的握手（handshake）协议。握手是在主控设备和从动设备之间交换命令和响应信号。它是图7-5中从动就绪信号使用方式的泛化。一根被称为主控就绪的控制线由主控设备启动，表示它已经准备好开始数据传输了，从动设备则通过启动从动就绪信号来响应。

握手协议控制的数据传送按照如下步骤进行。主控设备将地址和命令信息放到总线上。然后它激活主控就绪线来通知所有设备它已经将地址和命令信息放到总线上了。这使得所有设备对该地址进行译码。被选定的从动设备执行请求的操作，并激活从动就绪线来通知处理器它已经准备好。主控设备等待从动就绪线有效后，将自己的信号从总线上撤销。在读操作中，主控设备还需要将数据装入它的寄存器中。

图 7-6 给出了一个基于握手协议的输入数据传输操作的时序实例，它描述了如下的事件序列：

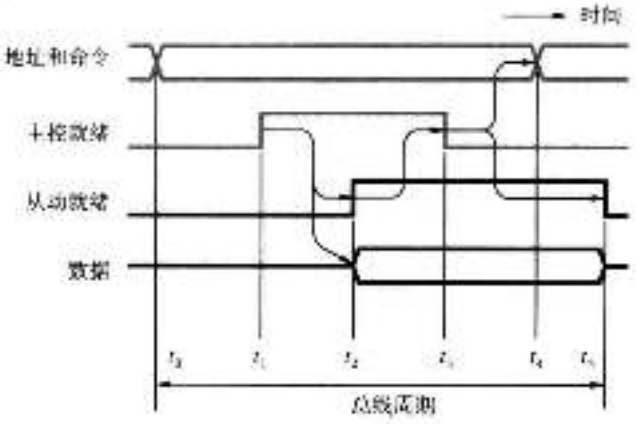



图 7-6　输入操作中数据传输的握手控制

$t_0$——主控设备将地址和命令信息放到总线上，总线上的所有设备对该信息进行译码。

$t_1$——主控设备将主控就绪线置为 1，通知所有设备地址和命令信息已经放到总线上了。$t_1 - t_0$ 的延迟用来补偿总线上出现的相位偏移（skew）。当同一信号源同时发送的两个信号不同时到达目的地时，就会出现相位偏移。这是由总线中不同线路的传播速度不同导致的。因此，为了保证主控就绪信号不在地址和命令信息之前到达任何设备，延迟 $t_1 - t_0$ 应该长于最大的总线相位偏移。（注意：在同步情况下，总线相位偏移是最大传播延迟的一部分。）要保证设备接口电路有足够的时间进行地址译码，这个延迟也应该包含在 $t_1 - t_0$ 内。 |234

$t_2$——被选定的从动设备在对地址和命令信息译码后，将它的数据放到数据线上来执行请求的输入操作。同时，将从动就绪信号置为 1。如果在将数据放到总线之前接口电路产生了额外延迟，从动设备必须相应地延迟从动就绪信号。$t_2 - t_1$ 这段时间的大小依赖于主控设备和从动设备之间的距离以及从动设备电路所产生的延迟。

$t_3$——从动就绪信号到达主控设备，表示输入数据已经在总线上可用。主控设备必须允许总线相位偏移。此外，还必须允许主控设备寄存器所需的准备时间。在延迟了最大总线相位偏移和最小准备时间之后，主控设备将数据装入它的寄存器中，然后撤销主控就绪信号，表示它已经收到数据。

$t_4$——主控设备从总线上撤销地址和命令信息。$t_3$ 与 $t_4$ 之间的延迟也是用来补偿总线相位偏移的。因为如果设备在总线上见到的地址发生变化，而主控就绪信号仍然等于 1，就会产生错误的寻址。

$t_5$——当设备接口收到主控就绪信号从 1 到 0 的跳变后，就从总线上撤销数据和从动就绪信号。此时输入传输结束。

图 7-7 显示的是输出操作的时序，实质上与输入操作相同。在输出

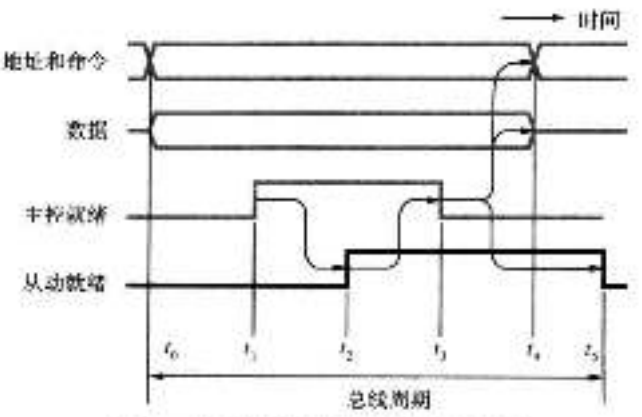



图 7-7　输出操作中数据传输的握手控制

操作中，主控设备在将输出数据放置到数据总线的同时发送地址和命令信息。选定的从动设备收到主控就绪信号后将数据装入它的数据寄存器中，并将从动就绪信号置1来表示它已经接收到数据了。这个周期的其余部分与输入操作相同。

图7-6和图7-7中的握手信号被认为是完全互锁（fully interlocked）的，因为一个信号的变化通常是对另一个信号变化的响应。因此，这种方式被称为完全握手（full handshake）方式。它提供了最高程度的灵活性和可靠性。

讨论

在商业计算机中使用了许多与上述类似的总线协议。具体设计的选择需要权衡如下因素：

- 接口电路的简单性。
- 适应具有不同延迟的设备接口的能力。
- 总线传输需要的总时间。
- 由于寻址不存在的设备或者由于接口故障所导致的错误检测能力。

235

异步总线的主要优点是握手协议不需要发布一个所有设备能够在同一时间见到的单一时钟信号，从而简化了时序设计。接口电路或在总线线路上进行传播所产生的延迟都可以很容易地解决。这些延迟可能随着设备的不同而不同，但数据传输的时序可以自动地进行调整。而对于同步总线，时钟电路必须仔细设计以确保正确的时序，并且延迟也必须严格控制在一定范围之内。

由握手协议控制的异步总线上的数据传送，由于每一次传送都要包括两个来回的延迟（四个端到端的延迟），所以传输速率受到很大的限制。从图7-6和图7-7中我们可以看出，从动就绪信号必须等到主控就绪信号的转变到达后才能转变，反之亦然。在同步总线中，时钟周期只需满足一个来回的延迟即可。因此，同步总线可以得到更高的传输速率。对于较慢的设备，只需像前面讲过的那样使用附加的时钟周期即可。今天我们使用的大部分高速总线都是使用同步方法。

### 7.2.3 电气考虑

总线是若干设备之间相互连接的媒介。因此有必要保证在任何给定的时间只能有一个设备可以将数据放置在总线上。在总线上放置数据的逻辑门被称为总线驱动器（bus driver）。除正在发送数据的设备之外，所有连接到总线上的其他设备都必须将其总线驱动器关闭。一个被称为三态门（tri-state gate）的特殊类型的逻辑门可以用于此目的。三态门有一个控制输入端，

236

用来将三态门打开或关闭。当三态门被打开或者启用时，它将根据输入信号的值用1或0驱动总线。当三态门被关闭或者禁用时，它将有效地与总线断开连接。从电气的角度来说，三态门的输出进入高阻抗状态，所以不会影响总线上的信号。

## 7.3 总线仲裁

有些时候两个或多个设备会竞争使用计算机系统中的某个资源。例如，两个设备可能同时需要访问一个给定的从动设备。在这种情况下，需要决定哪个设备可以首先访问从动设备。这通常由仲裁者（arbiter）电路通过执行一个仲裁过程来作出决定。每个设备发出使用共享资源的请求（request）时启动仲裁过程。仲裁者给每一个请求分配一个优先级。如果仲裁者同时接收到两个请求，它将把从动设备的使用权优先授予（grant）给拥有较高优先级的设备。

为了说明仲裁过程，我们考虑如下情况：一条单一的总线是共享资源，在总线上启动数据传输请求的设备是总线主控设备。在7.2节的讨论中仅涉及一个总线主控设备，也就是处理

器。但是有可能的是计算机系统中的多个设备都想成为总线主控设备来传输数据。例如，I/O 设备想成为总线主控设备来直接与计算机的存储器进行数据传输。因为总线是一个单一的共享设备，所以总线主控设备必须有次序地访问总线。

想要使用总线的设备发送请求给仲裁者。当多个请求同时到达的时候，仲裁者选择一个请求并把总线使用权授予给相应的设备。对于某些设备来说，获得总线使用权的延迟会导致错误发生，所以必须给这些设备分配较高的优先级。如果没有特别紧急的请求，则仲裁者会使用一种简单的轮转法把总线使用权轮流授予给各个设备。

图 7-8 显示了包含两个总线主控设备的总线仲裁情况。有两根总线请求线 BR1 和 BR2，以及两根总线授权线 BG1 和 BG2 将仲裁者与主控设备连接起来。总线主控设备通过激活它的总线请求线来请求使用总线。如果只有一根总线请求线被激活，则仲裁者激活相应的总线授权线。对于选定的主控设备来说这意味着现在它可以使用总线传输数据了。当传输结束时，主控设备释放其总线请求线，仲裁者也释放该主控设备的总线授权线。 [237]

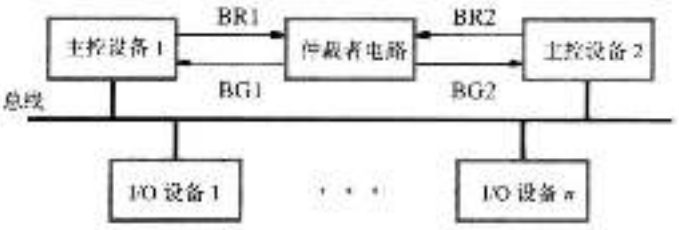



图 7-8　总线仲裁

图 7-9 说明了包含三个总线主控设备时可能发生的事件序列。假设主控设备 1 拥有最高的优先级，接下来是主控设备 2，而主控设备 3 的优先级最低。主控设备 2 首先发送一个使用总线的请求。因为此时没有其他请求，所以仲裁者启用 BG2 将总线授权给该主控设备。当主控设备 2 完成数据传输操作时，它通过释放 BR2 来释放总线。在那同时，主控设备 1 和 3 都启用了它们的总线请求线。因为设备 1 拥有更高的优先级，所以仲裁者在释放 BG2 之后就激活了 BG1，从而将总线授权给主控设备 1。之后，当主控设备 1 通过释放 BR1 来释放总线时，仲裁者释放 BG1 并激活 BG3 以把总线授权给主控设备 3。注意，即使主控设备 3 在主控设备 1 之前激活请求线，总线也会先授权给主控设备 1。

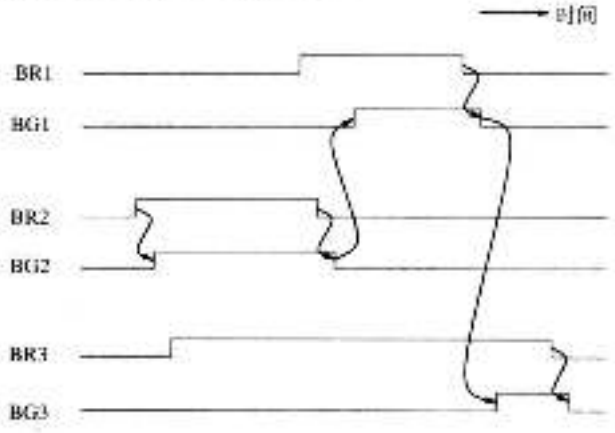



图 7-9　基于优先级的总线使用授权

## 7.4 接口电路

设备的I/O接口是由将设备连接到总线上的电路组成的。在接口的一侧有总线地址线、
[238] 数据线和控制线。在另一侧有接口与I/O设备之间传输数据所需的连接。这一侧称为端口（port），它可以是并行或串行的端口。并行端口每次可以同时传输多位数据。而串行端口每次只能发送或接收1位数据。与处理器的通信对于两种方式来说都是一样的，并行格式和串行格式之间的转换在接口电路中完成。

在介绍具体的电路实例之前，首先回顾一下I/O接口的功能。根据3.1节的内容，I/O接口：

1）提供一个临时存储数据的寄存器。

2）包含一个状态寄存器，其中含有可被处理器访问的状态信息。

3）包含一个控制寄存器，其中保存着管理接口行为的信息。

4）包含地址译码电路以确定何时被处理器寻址。

5）产生所需要的时序信号。

6）执行所有在处理器和I/O设备之间传送数据所需的格式转换，如串行端口下的并/串转换。

### 7.4.1 并行接口

现在我们用几个例子来解释一下接口设计的主要内容。首先，描述一下用于连接简单输入设备（如键盘）的8位输入端口的接口电路。然后我们再描述用于连接输出设备（如显示器）的8位输出端口的接口电路。假定这些接口电路是连接到一个32位处理器上的，该处理器使用存储器映射I/O寻址以及图7-6与图7-7所描述的异步总线协议。

**1. 输入接口**

图7-10显示了一个可以用来将键盘连接到处理器的电路。这个电路中的寄存器对应于图3-3中给出的寄存器。假设不使用中断，那也就不需要控制寄存器。因此只需要两个寄存器，一个数据寄存器KBD_DATA和一个状态寄存器KBD_STATUS。后者包含有键盘状态标志KIN。

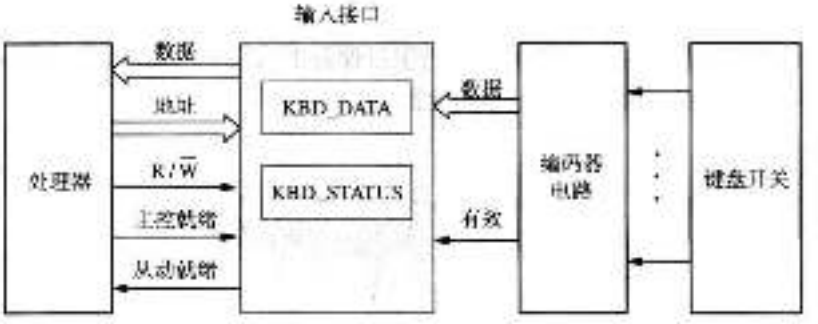



图7-10 键盘与处理器的连接

一般的键盘由机械开关组成，这些开关通常是断开的。当一个键被按下时，它的开关闭合形成一条电信号通路。该信号可以被编码器电路检测到然后产生相应字符的ASCII码。这种机械按钮开关的一个难点是当一个键被按下后的接触反弹（bounce），这会导致开关稳定在闭合位置之前电气连接的多次形成与断开。即使反弹只持续1～2毫秒，这也足以使计算机将
[239] 一次按键行为错误地解释为按下和松开了多次。可以使用一个简单的消除反弹的电路来消除反

弹的副作用，该电路可以是键盘硬件的一部分，也可以合并到编码器电路中。另外，开关反弹也可以用软件来处理。软件检测到键盘状态标志 KIN 被设置为 1 时就表明有某个键被按下。然后 I/O 程序在读取输入缓冲区 KBD_DATA 的内容之前可以引入足够长的延时来确保反弹已经结束。当采用硬件来消除反弹时，I/O 程序在检测到 KIN 等于 1 的时候就可以读取输入字符了。

图 7-10 中编码器的输出由表示编码字符的一个字节数据和一个有效（Valid）控制信号组成。当一个键被按下时，有效信号从 0 变为 1，并将相应字符的 ASCII 码装入 KBD_DATA 寄存器，同时将状态标志 KIN 置 1。处理器读取 KBD_DATA 寄存器的内容后将状态标志清 0。所示的接口电路与一条异步总线相连，该总线使用图 7-6 的握手信号“主控就绪”和“从动就绪”控制传输。此外该总线还有另外一根控制线 R/W，它等于 1 时表示读操作。

图 7-11 展示了一个输入接口电路。接口中有两个可寻址的单元，KBD_DATA 和 KBD_STATUS。它们在地址空间中占用相邻的字单元，如图 3-3 所示。实际上状态寄存器中只有 $b_1$ 这一位包含有用的信息，也就是键盘状态标志 KIN。当处理器读取状态寄存器时，其他的位都被认为是 0。

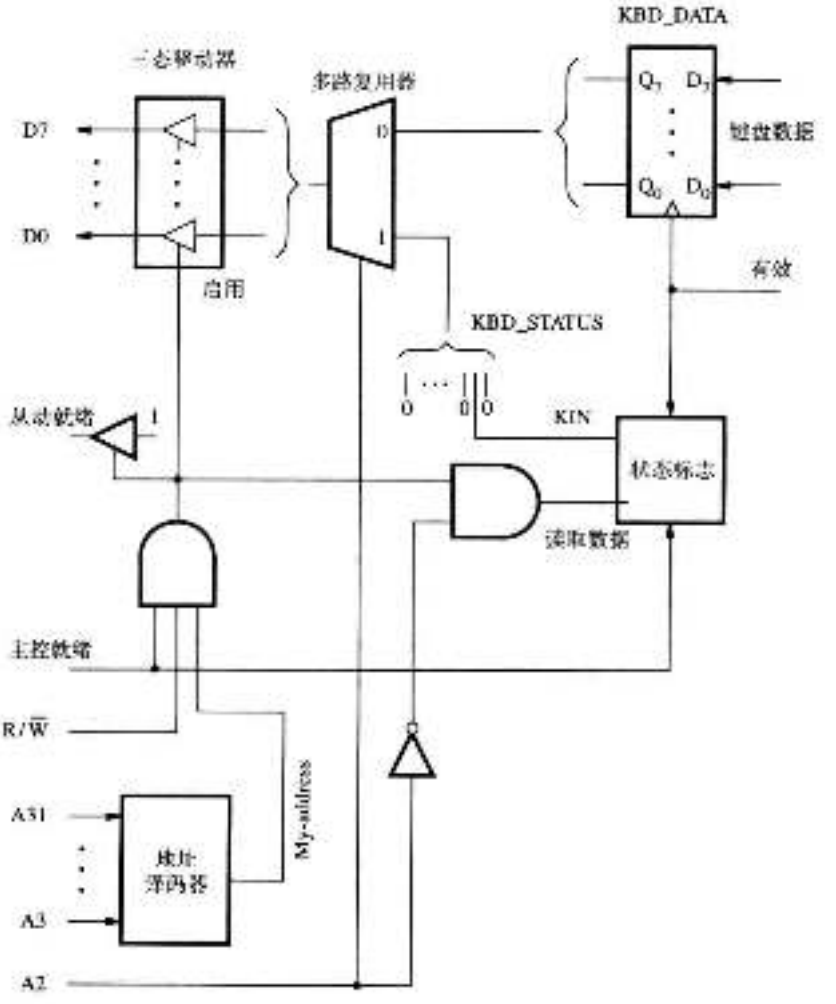



图 7-11　输入接口电路

当处理器请求一个读操作时，处理器将相应寄存器的地址放置到总线的地址线上。当KBD_DATA和KBD_STATUS这两个寄存器其中一个正在被寻址时，接口电路中的地址译码器会检查 $A_{31-3}$ 位，并发出其输出 My-address。$A_2$ 位决定两个寄存器的哪一个参与到过程中来。因此，可以使用一个多路复用器来根据地址位 $A_2$ 选择连接到总线上的寄存器。两个最低有效地址位 $A_1$ 和 $A_0$ 并没有使用，因为我们已经假设所有地址都是字对齐的。

多路复用器的输出通过一组三态门连接到总线的数据线上。只有当主控就绪、My-address和R/W这三个信号都等于1时接口电路才会将三态门打开，以表示读操作。从动就绪信号也同时被启用，以通知处理器所请求的数据或者状态信息已经被放置在数据线上了。当地址位 $A_2$ 等于0时，读取数据（Read-data）信号也被启用，这个信号用来重置KIN标志。

图7-12给出了状态标志电路的一种可行实现方案，KIN标志是所连接的NOR锁存器的输出，如图7-12所示。触发器将被"有效"信号线上的上升沿置1。这将改变NOR锁存器的状态，从而将KIN置为1，但这仅仅是在主控就绪信号为低电平时才会发生。该附加条件是为了确保KIN在被处理器读取的时候不能改变锁存器的状态。当"读取数据"信号变为1，表明KBD_DATA在被读取时，触发器和锁存器都被重置成0。

240
·
241

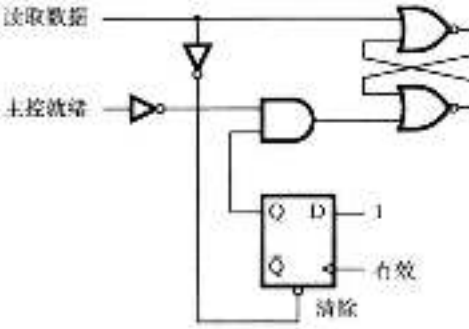



图7-12 图7-11中状态标志模块的电路

图7-11和图7-12所示的电路说明了接口电路需要实现的各种功能。利用现代计算机辅助设计工具的设计人员会使用硬件描述语言（如VHDL或者Verilog）来详细说明这些功能。设计出来的电路可能会因为所使用技术的不同而与这两幅图所示的电路有所不同。

### 2. 输出接口

现在让我们再来看一下图7-13所示的输出接口，它可以用来连接一个输出设备，如显示器。假设显示器使用两个握手信号：新数据（New-data）和就绪（Ready），使用方式与主控就绪和从动就绪两个总线信号之间的握手方式类似。当显示器准备好接收一个字符时，它就启动它的"就绪"信号，这使得DISP_STATUS寄存器中的DOUT标志被置为1。当I/O程序检查到DOUT等于1时，它就发送一个字符到DISP_DATA。这将会把DOUT标志清为0并把"新数据"信号置为1。显示器进行响应，将"就绪"信号设为0，接收并显示DISP_DATA中的字符。当准备好接收另一个字符时，显示器再次启动"就绪"信号，并重复上述过程。

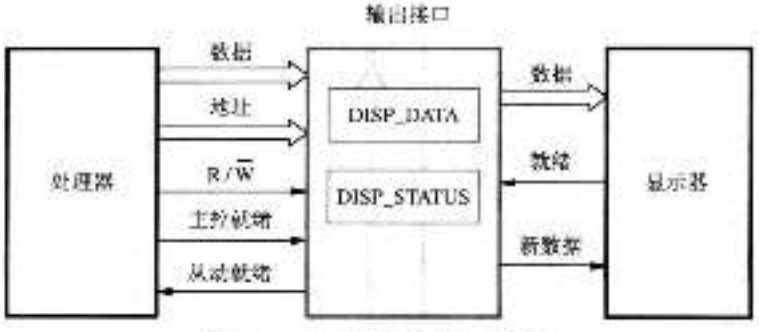



图7-13 显示器与处理器的连接

图 7-14 显示了输出接口的一种实现。该接口除了对读和写操作都会进行响应之外，它的操作与图 7-11 中输入接口的类似。当 $A_2$=0 时进行写操作，将一个字节的数据装入 DISP_DATA 寄存器中。当 $A_2$=1 时进行读操作，读取状态寄存器 DISP_STATUS 的内容。在这种情况下，只有 DOUT 标志，也就是状态寄存器的 $b_2$ 位会被接口发送。DISP_STATUS 中剩余的位都不会被使用。状态标志的状态由握手控制电路决定。本章最后的例 7.4 给出了描述这个电路行为的状态图。

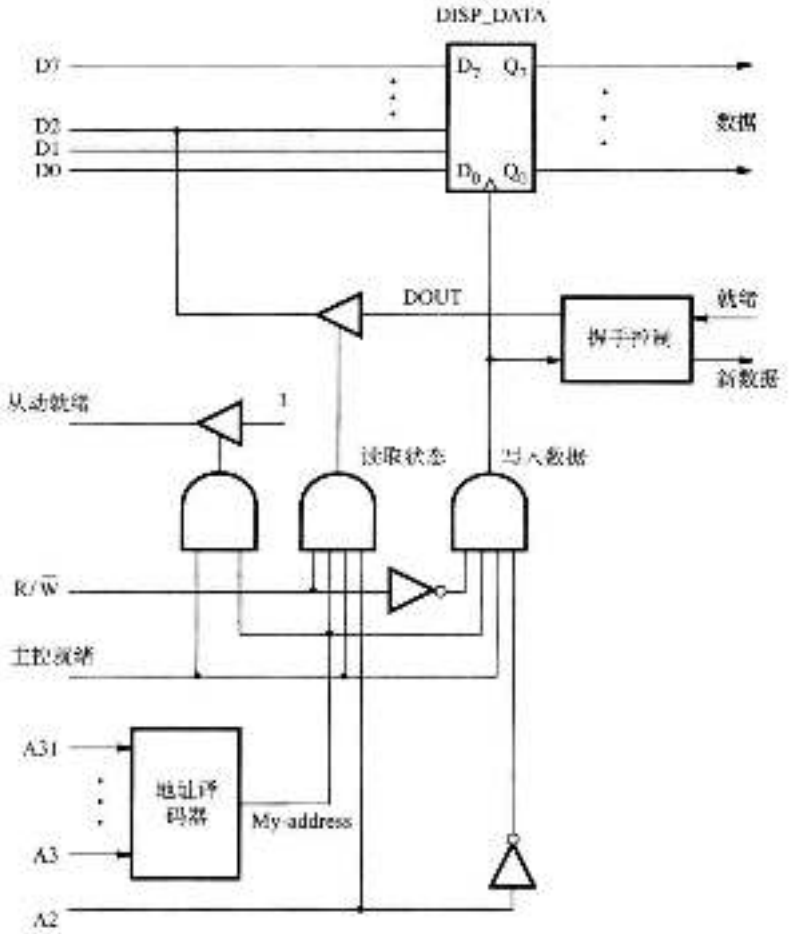



图 7-14　输出接口电路

## 7.4.2　串行接口

串行接口用于连接处理器与那些每次只传输一位数据的 I/O 设备。数据在设备端以串行位的形式传输，而在处理器端则以并行位的形式传输。并行与串行之间的转换由移位寄存器完成。移位寄存器有并行存取的能力。图 7-15 是一个典型的串行接口模块图。输入移位寄存器接收来自 I/O 设备的串行位输入。当 8 位数据都接收完后，移位寄存器的内容并行装入 DATAIN 寄存器中。相似地，DATAOUT 寄存器中的输出数据也被传输到输出移位寄存器中，在输出移位寄存器中数据被逐位移出并发送到 I/O 设备上。

这种接口中处理总线的部分与前面讲述的并行接口相同。两个状态标志，我们将其称为

SIN 和 SOUT，由状态与控制模块进行管理。当新数据从移位寄存器装入 DATAIN 后 SIN 标志被置为 1，这些数据被处理器读取后 SIN 标志被清为 0。SOUT 标志指出 DATAOUT 寄存器是否可用，处理器将新数据写入 DATAOUT 寄存器后 SOUT 被置为 0，数据从 DATAOUT 寄存器传送到输出移位寄存器后 SOUT 被置为 1。

图 7-15 中输入 / 输出通路中使用的双缓冲非常重要。如果将 DATAIN 和 DATAOUT 本身实现为移位寄存器的话，虽然不再需要独立的移位寄存器，但是这将会给 I/O 设备的操作带来不便。从串行线上接收一个字符后，接口只有等到处理器读取了 DATAIN 的内容后才能够开始接收下一个字符。这样，两个字符之间就需要一个间隔以使处理器有时间来读取输入数据。使用双缓冲技术的话，第二个字符的传送在第一个字符从移位寄存器装入 DATAIN 寄存器后就可以开始。因此，如果处理器在第二个字符的串行传输完成前读取了 DATAIN 寄存器的内容，接口就可以在串行线上接收连续的输入数据流。在接口的输出通路中也存在类似的情况。

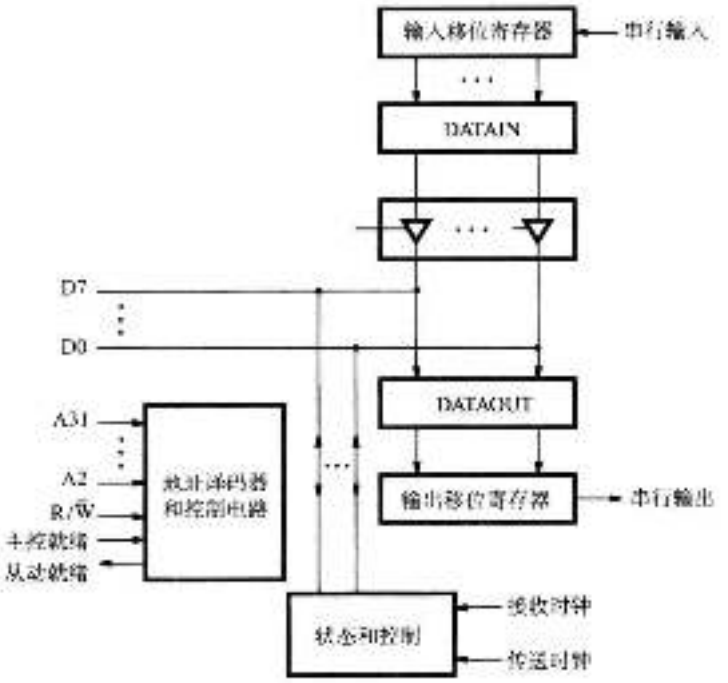



图 7-15 串行接口

在串行传输期间，接收器需要知道什么时候将每个位移到其输入移位寄存器中。因为没有单独的线路把时钟信号从发送器传送到接收器，所以必须使用一种编码方案将所需的时序信息嵌入到所传输的数据中。有两种基本的方法。第一种方法被称为异步传输，因为接收器使用的时钟与发送器的时钟不同步。在第二种方法中，接收器能够产生一个与发送器时钟同步的时钟，因此这种方法被称为同步传输。下面将简要描述这两种方法。

**1. 异步传输**

异步传输使用了一种称为起止（start-stop）传输的技术。数据被组织成带有明确定义的起点和终点的 6 到 8 位的小组。典型情况下，用 8 位编码的字符按照图 7-16 所示进行传输。连接发送器与接收器的线路在闲置时的状态为 1，传输一个 0 位作为起始位，后面紧接着的是 8 个数据位和 1 或 2 个终止位，终止位的逻辑值为 1。起始位开始处的 1 到 0 跳变提醒接收器数

242 / 245

据传输即将开始。接收器使用它自己的时钟来确定下一个 8 位数据的位置，并将这 8 位数据装入它的输入寄存器中。跟在所传输字符之后的终止位等于 1，确保能识别出下一个字符的起始位。当传输结束时，线路保持 1 状态，直到另一个字符开始传输。

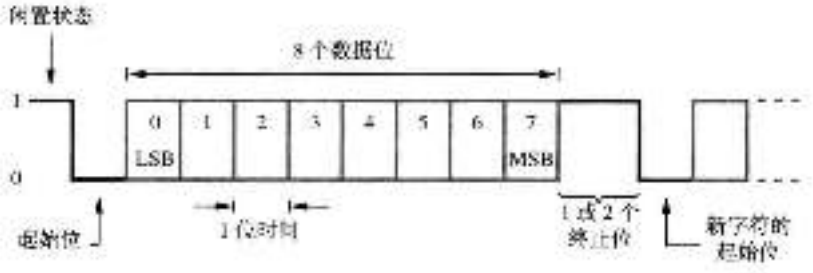



图 7-16　异步串行字符传输

为了保证接收器能够正确地接收数据，接收器在对输入数据进行采样的时候需要尽可能地靠近每一位的中心。这可以通过使用一个频率 $f_g$ 远远高于传输时钟 $f_t$ 的时钟信号来完成。通常，$f_g = 16f_t$。这意味着每一个数据位间隔期间可以产生 16 个本地时钟脉冲。这个时钟用来使一个模 16 的计数器递增，当检测到起始位的前沿时该计数器被清 0。计数值为 8 时就达到了起始位的中间位置。这时，输入线的状态再次被采样以确认它是一个有效的起始位（0），同时计数器被清 0。从这一点往后，每当计数达到 16 就对输入的数据信号采样一次，此时应当接近每一个输入位的中间位置。因此，只要 $f_g/16$ 足够接近 $f_t$，接收器都可以正确加载输入字符中的每一位。

2. 同步传输

上面描述的起止方式中，起始位的开始有一个 1 到 0 的跳变，如图 7-16 所示。这个跳变的位置是获取正确时序信息的关键。只有在传输速度足够低以及图中所示的方波形状在传输链路上保持不变的情况下，这种方式才是可用的。为了能高速地传输数据，接收器需要一个更可靠的方法来恢复时序信息。

在同步传输中，接收器通过检测所收到信号中连续的 1 到 0 和 0 到 1 跳变来产生一个与发送器时钟同步的时钟。接收器将时钟工作沿的位置调整到位位置的中心。有很多的编码方式可以用来确保发生足够的信号跳变以使得接收器能产生一个同步的时钟并能保持同步。一旦实现同步，数据传输就可以无限期地继续下去。编码数据通常是以块为单位进行传输的，每一块包括几百到几千比特。在每一块的开始和结尾都使用了适当的代码来标记，块内的数据是按照 246
一组约定的规则组织的。同步传输能实现非常高的数据传输率。

## 7.5　互连标准

一台典型的台式机或笔记本电脑拥有多个端口，这些端口可以用来连接鼠标、存储键或磁盘驱动器等 I/O 设备。如今已经开发出一些标准接口，使得 I/O 设备可以使用与任何特定的处理器无关的接口来与计算机连接。例如，一个具有 USB 连接器的存储键可以用在任何带有 USB 端口的计算机中。在这一节中，我们将简要描述一些广泛使用的互连标准。

大部分标准是由很多公司共同协作开发的。很多情况下，电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）会进一步开发这些标准并将它们作为 IEEE 标准进行发布。

### 7.5.1 通用串行总线

通用串行总线（Universal Serial Bus，USB）[1]是最广泛使用的互连标准。很多设备都带有一个USB连接器，包括鼠标、存储键、磁盘驱动器、打印机、摄像头及其他很多的设备。USB在商业上的成功是因为它的简单性及低成本。原来的USB规范支持低速（1.5Mb/s）和全速（12Mb/s）这两种速度的操作。后来，引入了称为高速USB的USB 2，它可使数据传输速度高达480Mb/s。当I/O设备继续发展到对速度有了更高的要求时就开发出了USB 3（称为超高速），它能支持高达5Gb/s的数据传输率。

USB是为了满足如下几个主要目标而设计的：

- 提供一个简单、廉价、使用方便的互连系统。
- 能够满足多种I/O设备和比特率，包括Internet连接和音频视频应用。
- 采用“即插即用”的操作模式，使用户操作更加方便。

在讨论USB的技术细节之前，我们先详细阐述一下这些设计目标。

**1. 设备特性**

连接到计算机上的各种设备其功能范围非常广泛，设备的数据传送速度、容量和时序限制相差都非常大。

在键盘上，每按下一个键就会产生一个字节的数据，而且随时都可能发生。这些数据应该迅速地传送给计算机。因为按键事件不与计算机系统中的任何其他事件同步，所以键盘产生
247 的数据是异步（asynchronous）的，并且又受操作人员速度的限制，所以产生数据的速率非常低，大概为10 byte/s，少于100 bit/s。

各种可能连接到计算机上的简单设备其产生数据的过程都有类似的特性——低速和异步。计算机鼠标和在视频游戏中使用的一些控制器和操纵杆就是很好的例子。

现在看一下另一种类型的数据源。许多计算机都有外带的或内置的麦克风。由麦克风接收到的声音产生了一个模拟电信号，该信号必须被转换成数字形式才能被计算机处理。周期性地对模拟信号进行采样可以完成这个转换。对每一个样本，模/数（A/D）转换器产生一个$n$位的数表示样本的量级。位的数量$n$是根据表示每个样本所需的精度来选择的。后来，当这些数据被送到扬声器时，再使用一个数/模（D/A）转换器将数字信号转换为原来的模拟信号。类似的方法也可以用来处理来自摄像头的视频信息。

采样过程将产生一个连续的数字化样本流，这些数字化样本按固定的时间间隔到达，与采样时钟同步。这种数据流被称为等时（isochronous）的，指的是连续的事件被相同的时间段隔开了。信号必须以足够快的速度进行采样，以捕获它的最高频率。一般来说，如果抽样频率为每秒$s$个样本，采样过程捕捉到最高频率就是$s/2$。例如，用8kHz的采样频率可以充分捕捉到人的语音信息，这将记录高达4kHz频率的声音信号。对于更高品质的声音，如在音乐系统中，需要使用更高的采样频率。数字声音的一种标准采样频率是44.1kHz。每一个样本用4个字节的数据表示以满足高品质声音再现所需的较大音量范围（动态范围）。这将产生大约为1.4Mb/s的数据传输速率。

在处理采样的声音和音乐信号时，有一个很重要的要求就是在采样和回放过程中保持精确的时序。高度的抖动（样本时序的变化）是不允许的。因此，计算机和音乐系统之间的数据传送机制必须保持一致的样本间延迟。否则就需要引入复杂的缓冲和重定序电路。另一方面，偶尔的错误或样本丢失是可以接受的。听者可能完全注意不到，或者只产生不引人注意的滴答声。并不需要复杂的机制来确保完全正确的数据传输。

图像或视频数据的传输也有相似的要求，但要求更高的数据传输率。为了保证商用电视

的图像质量，一张图片需要用 160KB 来表示，并且每秒钟发送 30 帧，再加上控制信息，这将产生 44Mb/s 的总比特率。高品质的图像，如 HDTV（高清晰度电视）会要求更高的传输速率。

大容量存储设备如磁盘和光盘的要求则不同。这些设备是计算机存储器层次结构的一部分，将在第 8 章中介绍。它们到计算机的连接需要至少 40 或 50Mb/s 的数据传输带宽。磁盘机械装置中机械部件的运动所产生的延迟大约为 1 毫秒。因此，向计算机发送或从计算机中接收数据时产生的较小额外延迟可以忽略，也不用考虑抖动。但是传输机制必须保证数据的正确性。 [248]

**2. 即插即用**

当一个 I/O 设备连接到一台计算机上时，操作系统需要设备的一些信息。它需要知道设备是什么类型的，以便于它能使用合适的设备驱动程序。操作系统还需要知道设备接口中的寄存器地址以便于能和设备接口进行通信。USB 标准定义了能与其通信的 USB 硬件和软件。它的即插即用（plug-and-play）特性指的是一台新设备连接到计算机上时，系统能够自动检测到它的存在。该软件可以确定设备的种类、如何与它通信以及它可能具有的任何特殊的要求。其结果是，用户只需简单地插入 USB 设备，就可以开始使用它，而不必参与其中任何细节。

USB 也是可热插拔的，这指的是在电源开启的情况下设备可以插入 USB 端口或者从 USB 端口移除。

**3. USB 体系结构**

USB 使用了点对点的连接以及串行传输格式。当多个设备被连接起来的时候，它们被组织成图 7-17 所示的树形结构。树的每个结点都有一个称为集线器（hub）的设备，它是主计算机与 I/O 设备之间的中间传输点。在树的根部有一个根集线器（root hub）将整棵树连接到主计算机上。树的叶子是 I/O 设备，如鼠标、键盘、打印机、Internet 连接器、摄像头或扬声器。树形结构使得它可以使用简单的点到点串行链接来连接许多设备。

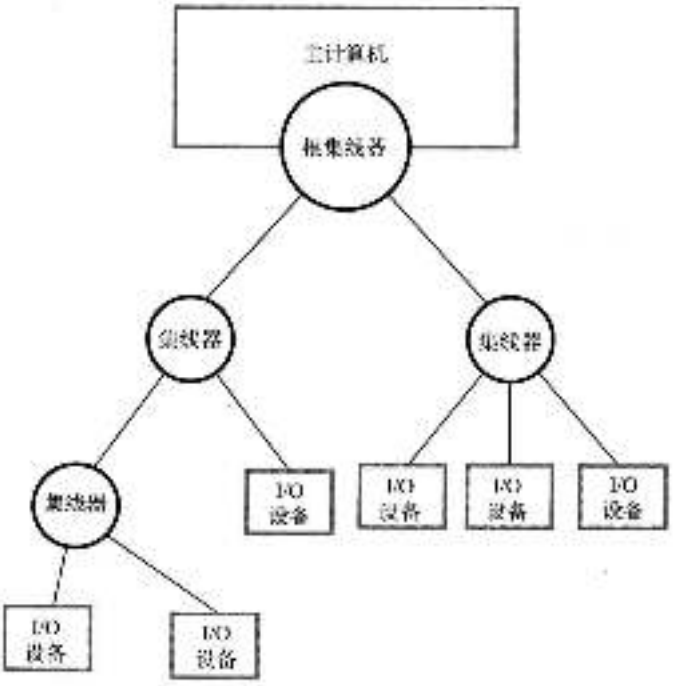



图 7-17　通用串行总线（USB）的树形结构

如果I/O设备可以在任何时候发送消息，那么两条消息可能在同一时间到达集线器并相互干扰。因为这个原因，USB的操作是严格基于轮询法的。设备只有在响应主处理器的轮询时才发送消息。因此，没有两台设备可以同时发送消息。这个限制使得我们可以采用非常简单和廉价的集线器。

USB上的每一台设备，不管是集线器还是I/O设备，都分配有一个7位的地址。这个地址是USB树的局部地址，并且与处理器的地址空间没有任何关系。连接到处理器上的USB的根集线器看起来就像是一台单一的设备一样。主机软件通过将信息发送到根集线器来与单个设备进行通信，根集线器再将信息转发给USB树中的适当设备。

当一台设备第一次连接到集线器上或被启动时，它的地址为0。主机周期性地轮询每一个集线器来收集状态信息，了解是否有新增加或删除的设备。当主机得知一台新设备已连接后，它便从设备USB接口的一个特殊存储器中读取信息以了解该设备的功能，然后为该设备分配一个唯一的USB地址并把这个地址写到设备的某个接口寄存器中。这就是使USB设备具备即插即用功能的初始连接过程。

4. USB上的等时传输

USB的一个主要特性是它能够支持以简单的方式传输等时数据。正如前面所提到的，等时数据需要在精确定时的固定时间间隔内传输。为了适应这种类型的传输，根集线器每毫秒在USB树内发送一个唯一可识别的位序列。这个位序列被称为起始帧字符，用来标识该字符之后所传输的等时数据的开始。因此数字化的音频和视频信号可以用一种定期而精确定时的方式进行传输。

5. 电气特性

USB连接由4根电线组成，其中两根用来连接电源+5V和地，另外两根用于传输数据。因此，对电源需求不大的I/O设备可以直接从USB获得能源，这消除了像存储键和鼠标这样的简单设备对独立电源的需要。

在USB电缆中发送数据可以使用两种方法。在低速发送数据时，相对于地线而言的高电压在两根数据线中的一根上传输时表示发送0，在另一根上传输时表示发送1。在两种情况中地线都传送返回电流。这种在相对于地线的线路上传输信号的方法被称为单终（single-ended）传输模式。

249
|
250

数据在任何电缆中的传输速度都受到电噪声数量的限制。噪声（noise）是指任何干扰所需的数据信号从而可能会导致错误的信号。单端传输模式很容易受到噪声的影响。地线上的电压对于连接到计算机上的所有设备来说是共同的。一个设备发送的信号可能引起地线电压的微小波动，因此会干扰其他设备发送的信号。某根线受邻近线上噪声的影响也可能会产生干扰。

高速USB使用了另一种被称为差分信号（differential signaling）的方法。数据信号在绞在一起的两根数据线（即双绞线）上传输。地线不参与传输过程。接收器不参考地线，直接感知两根信号线间的电压差。这种方法对于消除接收器所收到的噪声是非常有效的，因为任何输入到双绞线中某一根线上的噪声也会同时输入到另外一根线上。由于接收器只对两根线之间的电压差敏感，所以噪音成分也就被消除了。地线扮演了屏障的角色，防止双绞线上的数据受到邻近线路的干扰。与单端信号法相比，差分信号法可以使用更低的电压和更高的速度进行数据传输。

## 7.5.2 火线

火线是另一种流行的互连标准。它最初是由苹果公司开发的，现在已被采纳为IEEE1394

标准 [2]。像 USB 一样，它使用了差分点对点串行连接。下面是火线与 USB 之间显著的不同点：

- 火线总线上的设备被组织成菊花链的形式，而不是 USB 的树形结构。第一个设备连接到计算机上，第二个设备连接到第一个设备上，第三个设备连接到第二个设备上，以此类推。
- 火线适用于连接音频和视频设备。它可以运行在等时模式下，该模式被高度优化以实现高速等时传输。
- 连接到 USB 的 I/O 设备与主计算机进行通信。如果数据从一个设备传输到另一个设备，例如从摄像头传输到显示器或打印机，则数据首先被主计算机读取然后再将其发到显示器或打印机上。而火线则不同，它支持一种点对点（peer-to-peer）的操作模式。这指的是数据可以从一个 I/O 设备直接传输到另一个 I/O 设备，而不需要主计算机的参与。
- 基本的火线连接器有 6 个引脚。有两对数据线，其中一对用于在每个方向上传输数据，另一对用于电源和接地。高速版本的火线使用 9 个引脚的连接器，增加了三根地线来屏蔽对数据线的干扰。
- 火线总线可以提供比 USB 更大的功率。因此它可以支持中等功率要求的设备。

火线被广泛应用于音频和视频设备中。例如，大多数摄像机有一个火线端口。现有的几个版本的火线标准中，其工作的速度范围从 400Mb/s 到 3.6Gb/s。

### 7.5.3　PCI 总线

PCI（外围部件互连，Peripheral Component Interconnect）总线 [3] 是一种廉价的且独立于处理器的总线。它位于计算机的主板上，用来连接各种设备的 I/O 接口。连接到 PCI 总线上的设备在处理器看来就好像它是直接连接到处理器总线上的一样。它的接口寄存器在处理器的地址空间内分配地址。

我们将首先描述 PCI 总线如何工作，然后再讨论 PCI 总线的一些特性。

#### 1. 总线结构

图 7-18 说明了 PCI 总线在计算机系统中的使用。PCI 总线通过一个被称为桥的控制器连接到处理器总线上。桥有一个特殊的端口用来连接计算机的主存储器。它还可能有另外一个特殊的高速端口用来连接图形设备。桥将一条总线上的命令和应答进行翻译并转发到另一条总线上，并在两条总线之间传输数据。例如，当处理器发送一个读请求给一个 I/O 设备时，桥将命令和地址转发给 PCI 总线。当桥接收到设备的应答时，它再使用处理器总线将数据转发给处理器。I/O 设备可能通过使用以太网、USB、SATA、SCSI 或 SAS 等标准的端口连接到 PCI 总线上。

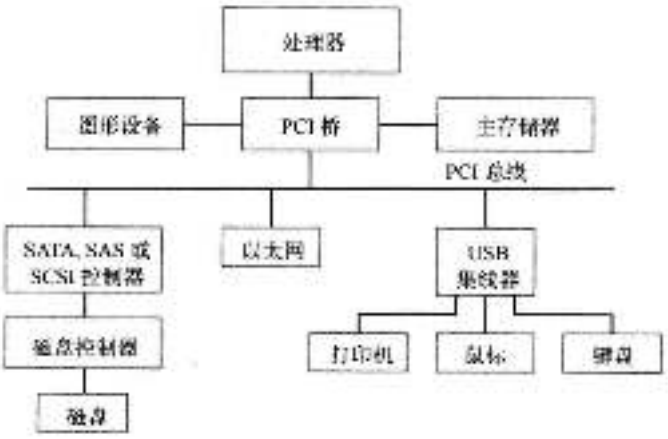



图 7-18　PCI 总线在计算机系统中的使用

PCI总线支持三个独立的地址空间：存储器、I/O和配置。即使处理器有独立的I/O地址空间可用，系统设计者也可能会选用存储器映射I/O方式。实际上，PCI标准就是建议使用这种方法以获得更广泛的兼容性。配置空间是用来支持PCI即插即用功能的，稍后我们将简要说明。随同地址一起发送的一个4位命令将指明在特定的数据传输操作中使用的是三个地址空间中的哪一个。

计算机总线上的数据传输往往以数据块的形式操作，而不是单个的字。存储在连续存储单元中的字在存储器和I/O设备（如磁盘或以太网连接）之间直接传输。数据传输由I/O设备的接口作为总线主控设备发起。这种直接在存储器与I/O设备之间传输数据的方法将在第8章中进行详细的讨论。PCI总线设计的主要目的是支持多字传输。单个字的读或写操作被简单地视为长度为1的块操作。

PCI总线上的信号约定类似于图7-5中使用的方式，但有一个重要的区别。PCI总线使用同一线路来传输地址与数据。在图7-5中，我们假设主控设备将在总线上保持地址信息直到数据传输结束。但这并不是必须的。地址信息只需保持到从动设备被选定就足够了，然后就可以释放线路，以便于它在后续的时钟周期中发送数据。在多个字的传输中，从动设备可以将地址存储在一个内部寄存器中，并递增它以访问连续的地址单元。这种方式可以显著降低成本，因为总线的线路数量是影响计算机系统成本的一个重要因素。

**2. 数据传送**

为理解PCI总线的操作和它的各种特性，下面我们将分析一个典型的总线事务。总线主控设备，也就是通过发出读写命令来启动数据传输的设备，在PCI术语中被称为启动设备（initiator）。响应这些命令的被寻址的设备被称为目标设备（target）。表7-1中列出了用于传输数据的主要的总线信号。有32或64根线使用与图7-5中类似的同步信号方法来传输地址和数据。目标就绪信号TRDY#相当于图7-5中的从动就绪信号。另外，PCI总线还使用了启动就绪信号IRDY#来支持块传输。我们将简要介绍这些信号，以让读者了解总线的主要特点。

PCI总线上一次完整的传送操作，包括一个地址和一块数据，被称为事务（transaction）。让我们来看一个启动设备从存储器中读取4个连续的32位字的总线事务。图7-19显示了总线上的事件顺序。所有信号的跳变都由时钟的上升沿触发。与图7-5一样，通过显示信号在这个
[253] 时钟周期中稍后才发生变化来表示它们遇到的延迟。名字以#结尾的信号表示低电平有效。

表7-1 PCI总线上的数据传送信号

| 名 称 | 功 能 |
|---|---|
| CLK | 33MHz或66MHz的时钟 |
| FRAME# | 由启动设备发送以指示传输的持续时间 |
| AD | 32位地址/数据线，可以增加到64位 |
| C/BE# | 4位命令/字节允许线（64位总线时为8位） |
| IRDY#, TRDY# | 启动就绪和目标就绪信号 |
| DEVSEL# | 来自设备的响应，表示它已经识别了地址并准备好了数据传送事务 |
| IDSEL# | 初始化设备选择 |

在时钟周期1，总线主控设备作为启动设备启动信号FRAME#表示事务开始。同时，它将地址发送到AD线上，将命令发送到C/BE#线上。在本例中，命令指明了这是一次读操作请求并且使用的是存储器地址空间。

在时钟周期2，启动设备撤销地址信号，将它的驱动器从AD线上断开，并启动信号IRDY#来表示它已准备好接收数据。被选定的目标设备启动信号DEVSEL#来表示它已经辨识

出它的地址并准备好做回应了。同时，被选定的目标设备将它的驱动器连接到 AD 线上，以便在随后的周期中它可以发送数据给启动设备。时钟周期 2 用来满足当启动设备关闭其驱动器而目标设备启用它的驱动器时，转换 AD 线的延迟。目标设备在时钟周期 3 启动 TRDY# 信号并开始发送数据。它将保持 DEVSEL# 信号直到事务结束。

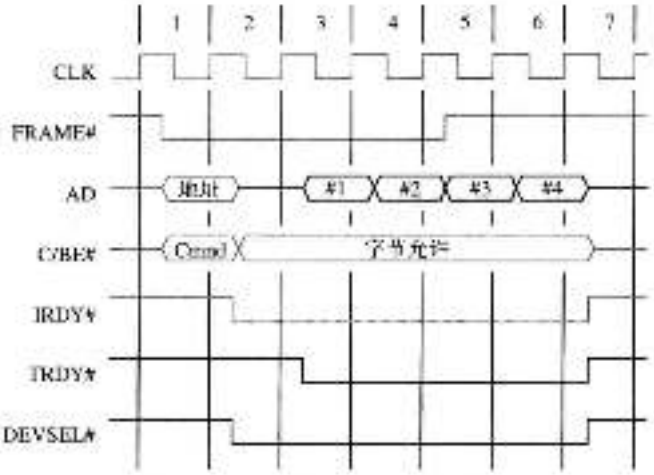



图 7-19　PCI 总线上的一次读操作

我们已经假定目标设备在时钟周期 3 已经准备好发送数据了。如果没有准备好，目标设备会推迟启动 TRDY# 信号直到它做好准备。整个数据块不需要在连续的时钟周期中发送。启动设备或目标设备可能会通过撤销其就绪信号来引入一个暂停，然后当设备准备好恢复数据传输时再启动就绪信号。

C/BE# 线在时钟周期 1 用来发送总线命令，在其余周期中用作其他目的。这四根线中的每一根都与 AD 线上的一个字节相关联。启动设备启动 C/BE# 线中的一根或几根来表示哪几根字节线将被用来传送数据。

启动设备使用 FRAME# 信号表示脉冲的持续时间。它在传送倒数第 2 个字时撤销这个信号。在图 7-19 中，启动设备在时钟周期 5，也就是在它接收第 3 个字的周期撤销 FRAME# 信号。作为响应，目标设备在时钟周期 6 再发送一个字并，然后停下来。发送完第 4 个字后，目标设备撤销 TRDY# 和 DEVSEL# 信号，并将它的驱动器与 AD 线断开。

### 3. 设备配置

将一台 I/O 设备连接到计算机上时，需要对与它进行通信的设备接口和软件进行配置操作。像 USB 一样，PCI 也具有即插即用的功能，这极大程度地简化了这一配置过程。事实上，即插即用这个特性最早是由 PCI 标准引入的。PCI 接口包括一个较小的配置 ROM 存储器来保存与其连接的设备的有关信息。所有设备的配置 ROM 在配置地址空间内都可以被访问，它们在系统启动或复位时由 PCI 的初始化软件读取。通过读取配置 ROM 中的信息，初始化软件可以确定该设备是打印机、摄像头、以太网接口还是磁盘控制器，还可以得知各种设备选项和特性方面的信息。

连接到 PCI 总线的设备并没有在其 I/O 接口硬件中分配永久的地址。相反的，设备的地址是在初始化配置过程中由软件分配的。也就是说当电源接通时，不能以通常的方式使用它们的地址来访问设备，因为还没有给它们分配任何地址。此时需要使用另一种不同的机制来选择 I/O 设备。

254
|
255

PCI 总线最多可以有 21 个可供 I/O 设备接口卡插入的连接器。每个连接器都有一个被称为初始化设备选择（IDSEL#）的引脚。这个引脚被连接到高位的 21 根地址 / 数据线 AD11 到 AD31 中的一根。如果设备接口的 IDSEL# 输入端被启用，那么它会响应配置命令。配置软件扫描所有 21 个位置来确定 I/O 设备接口安装在哪里。对于每一个位置，配置软件使用一个地址来发布一条配置命令，在这个地址中对应于那个位置的 AD 线被置为 1，其余的 20 根线被

置为0。如果某个设备接口产生回应，则给它分配一个地址，并将这个地址写入它为此目的所指定的寄存器中。使用相同的编址机制，处理器读取设备的配置ROM并进行任何必要的初始化工作。它使用低地址位AD0到AD10来访问配置ROM中的位置。这个自动化的过程意味着使用者只需插上接口板并打开电源即可，其余的工作全部由软件来完成。

PCI总线已经非常流行，尤其是在PC领域。它也用在许多其他的计算机中，这主要得益于它可以为大范围的I/O设备提供PCI接口。现在有32位和64位两种配置可用，它们分别使用33MHz或者66MHz的时钟。一种称为PCI-X的高性能版本也投入使用了，它是以133MHz时钟频率运行的64位总线。而更高性能版本的PCI-X则能以高达533MHz的时钟频率运行。

### 7.5.4 SCSI总线

缩写SCSI代表小型计算机系统接口[4]。它是由美国国家标准组织（ANSI）定义的一种总线标准。SCSI总线可以用来将各种设备连接到计算机上，而它特别适合用于磁盘驱动器。我们往往可以在使用了很多磁盘驱动器的机构数据库或电子邮件系统中发现SCSI总线的踪影。

在最初的SCSI标准规范中，设备通过50根线的电缆连接到计算机中，电缆最长可达25米，数据传输速率最高可达5MB/s。SCSI总线标准经历了多次修改，它的数据传输能力增长得非常快。已经定义的SCSI-2和SCSI-3标准每一种都有好几个选项。使用高达80MHz的时钟频率时，数据可以并行传输8位或者16位。同时也有好几种电信号方式可供选择。SCSI总线可以使用单端传输，此时每一个信号使用一根线，所有信号使用一根共用的地线进行返回。在另一种选择中，使用了差分信号，每个信号使用一对线路。

**数据传输**

连接到SCSI总线的设备不属于处理器地址空间的一部分，这与连接到处理器总线或PCI总线的设备是一样的。SCSI总线可能会直接连接到处理器总线上，但更有可能通过SCSI控制器连接到PCI之类的其他标准I/O总线上。数据和命令会以多字节数据包的方式进行传输。为了向设备发送命令或数据，处理器在存储器中组装信息，然后指示SCSI控制器把它传送给设备。同样，当从设备读取数据时，SCSI控制器将数据传送到存储器中，然后通过中断的方式通知处理器。

[256]

为了说明SCSI总线的操作，我们来看一下它是如何与磁盘驱动器一起工作的。与磁盘驱动器通信跟与主存通信在本质上是完全不同的。数据是存储在磁盘的扇区（sector）中的，每个扇区包括几百个字节。当一个数据文件被写入磁盘时，它并不总是存储在相邻的扇区中，因为磁盘中一些扇区可能已经包含了先前存储的数据，还有一些可能有缺陷必须忽略。所以，一次读或写请求访问的可能是几个不相邻的磁盘扇区。由于磁盘机械移动的限制，在到达第一个需要传送数据的扇区前可能会有大约几毫秒的较长延迟，然后一块数据被高速传送。接着又会发生另一个延迟以达到下一个扇区，再传送另一块数据。一次读或写请求可能包括好几块这样的数据。SCSI协议有助于这种操作模式的实现。

我们分析一个完整的读操作例子。下面是简化的高级描述，忽略了细节与信号约定。假设处理器要从磁盘驱动器读取一块数据时，这些数据存储在两个不相邻的磁盘扇区中。处理器发送一个命令给SCSI控制器，将发生如下事件序列：

1）SCSI控制器竞争SCSI总线的控制权。

2）当SCSI控制器在仲裁过程中获胜后，它向磁盘控制器发送命令，指明所要求的读操作。

3）磁盘控制器不能马上开始传输数据。它必须先把磁头移动到所需的扇区。因此，它发送一个消息给SCSI控制器，表明它将暂时中断它与SCSI控制器的连接。SCSI总线现在是空

闲的，可以被其他的设备使用。

4）磁盘控制器发送一个命令给磁盘驱动器，移动磁头到读操作中的第一个扇区。然后，读取存储在扇区里的数据并保存到数据缓冲区中。当准备好开始传送数据时，磁盘控制器请求总线控制权。仲裁获胜后，它重新建立与 SCSI 控制器的连接，传送数据缓冲区中的内容，然后再次中断连接。

5）重复这个过程，读取并传输第二个磁盘扇区的内容。

6）SCSI 控制器将所请求的数据传输到主存储器中，并发送一个中断给处理器表示数据现在是可用的。

这一过程表明 SCSI 总线上交换的消息要比处理器总线上交换的消息更高级一些。消息指的是可能需要好几步才能完成的更复杂的操作，这取决于具体的设备。处理器和 SCSI 控制器都不需要了解磁盘操作的细节，也不需要知道它是如何从一个扇区移动到另一个扇区的。

SCSI 总线标准定义了大量的控制消息。这些消息可以用来操作不同类型的 I/O 设备。还定义了用来处理在设备操作或数据传送期间可能发生的各种错误或故障情况的消息。 257

### 7.5.5　SATA

在个人计算机的早期日子里，流行的 IBM 计算机的总线 AT 成为一个行业标准，它是基于 Intel 8080 微处理器总线的。它被命名为 ISA，是工业标准体系（Industry Standard Architecture）的缩写。一个增强版本，其中包括了用来支持磁盘驱动器的基本软件定义，后来被命名为 ATA，是 AT 附件（AT Attachment）总线的缩写。这种体系体系结构的串行版本就是人们熟知的 SATA[5]，如今已经广泛应用在磁盘接口中。像所有标准一样，现在已经开发出了几个版本的 SATA，增加了新的特性并提高了速度。原始的并行版本已经被重新命名为 PATA，但是在新设备中已经不再使用了。

基本的 SATA 连接器有 7 个引脚，连接了两对双绞线及三根地线。使用了差分传输，时钟频率的范围从 1.5Gb/s 到 6Gb/s。最近的一些版本还提供了等时传输特性以支持音频和视频设备。

### 7.5.6　SAS

SAS 是 SCSI 总线的串行实现，因此它的名字叫做串行连接 SCSI（Serially Attached SCSI）[6]。主要用于连接磁盘和 CD、DVD 驱动器。它使用了类似于 SATA 的串行点对点连接。SAS 连接可以同时在两个方向上传输数据，最高速度可以达到 12Gb/s。在软件层面上，SAS 与 SCSI 完全兼容。

### 7.5.7　PCI Express

I/O 互连的需求日益增加。互联网连接、复杂的图形设备、流媒体视频及高清晰度电视是涉及高速数据传输的典型应用实例。PCI Express 互连标准（通常被称为 PCIe）[7] 就是为满足这些需求而开发的，且随着新应用的引入，又不可避免地需要进一步提高数据传输速率。

PCI Express 使用串行的点对点连接，经由交换器相互连接形成一个树形结构，如图 7-20 所示。树的根节点称为根联合体（Root complex），被连接到处理器总线上。根联合体有一个连接主存储器的特殊端口。从根联合体发出的所有其他连接都是连到 I/O 设备上的串行连接。这其中的某些连接可能会连到一个交换器上以扩展出更多的串行分支，如图中所示。交换器可能还会连接到支持其他标准（如 PCI 或 USB）的桥接接口上。例如，树形结构的一个分支可能是一条 PCI 总线，以利用现有的具有 PCI 接口的多种设备。

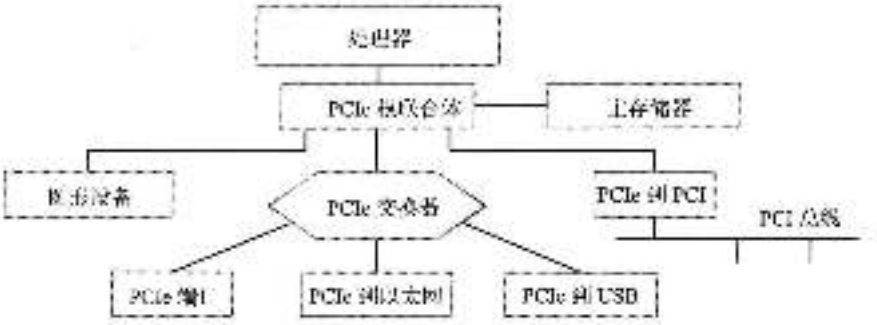



图 7-20 PCI Express 连接

基本的 PCI Express 连接包括两对双绞线，每一对用于一个方向上的数据传输。数据在每一对双绞线上以 2.5Gb/s 的速度传输，并使用 7.5.1 节中所描述的差分信号法。数据可以同时在两个方向上传输。另外，因为在 PCI 或 SCSI 中没有共享总线，所以连到不同设备的连接可以同时传输数据。此外，一个连接在每个方向上可能会使用多对双绞线。每个方向使用一对双绞线的基本结构被称作一个通道（lane），也被称为 X1 连接。一个连接可能会使用 2、4、8 或 16 个通道，分别被称为 X2、X4、X8 或者 X16 连接。

同步传输连接上的接收器必须与发送器的时钟同步，如 7.4.2 节中所述。为了做到这一点，对所传输的数据进行编码以确保 0 到 1 和 1 到 0 的跳变可以发生的足够频繁。在 PCIe 标准中，每一个 8 位的数据用 10 位进行编码。插入到数据流中的其他位是为了执行各种控制功能，如描述地址和数据信息。在添加了附加位后，数据传输率为 2.5Gb/s 的一对双绞线实际上每秒钟只传输了 1.6Gb 或者 200MB 的有效信息。一个 X16 连接可以在每个方向上以 3.2Gb/s 的速度传输数据。相比之下，一个工作频率为 64MHz 的 64 位 PCI 总线峰值时的总数据传输率是 512MB/s。PCI Express 的另外一个优点是使用了少量的线路，因而成本较低。

258 ~ 259

PCI Express 协议与 PCI 协议完全兼容。例如，它们使用了相同的初始化配置过程。因此，一台使用 PCI Express 的计算机可以使用为基于 PCI 总线的系统所开发的操作系统及应用软件。

## 7.6 结束语

本章从硬件的角度介绍了计算机的 I/O 结构。以连接到总线上的 I/O 设备为例，来说明数据传输的同步与异步方法。

由于人们对高速的数据传输、低廉的成本和便利的功能（如即插即用）等方面的需求不断地增长，输入/输出设备的互连网络体系结构已经成为一个主要的发展领域。本章简要描述了几种 I/O 标准，并阐明了用于实现这些目标的方法。目前的趋势是从并行总线转移到串行的点对点连接。串行连接成本较低并且可以高速地传输数据。

## 7.7 问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。

**例 7.1**

**问题**：计算机的 I/O 总线使用了图 7-4 所示的同步协议。该总线的最大传播延迟为 4ns。总线主控设备将地址放到地址线上需要花费 1.5ns。从动设备需要 3ns 来对地址进行译码，另外需要最长 5ns 的时间来将所请求的数据放置在数据线上。连接到总线上的输入寄存器需要的最小建立时间为 1ns。假设总线时钟有 50% 的占空比，即时钟的高相位与低相位的持续时间相等。请问该总线最大的时钟频率是多少？

**解答**：对钟高相位的最小时间等于地址到达从动设备并被译码的时间，也就是 1.5 + 4 + 3 = 8.5 ns。时钟低相位的最小时间则等于从动设备将数据放到总线上的时间加上主控设备把数据装入寄存器的时间，也就是 5 + 4 + 1 = 10 ns。因此，最小的时钟周期为 2 × 10 = 20 ns，可以得出最大的时钟频率为 50MHz。

**例 7.2**

**问题**：一个仲裁者接收到了三个请求信号 R1、R2 和 R3，并产生三个授权信号 G1、G2 和 G3。请求信号 R1 无先级最高，R3 的无先级最低。图 7-9 给出了这个仲裁者操作的例子。请给出一个描述该仲裁者行为的状态图。

**解答**：图 7-21 给出了一个状态图。其中仲裁者从空闲状态 A 开始。当一个或多个请求信号发出时，仲裁者会移动到 B、C、D 三个状态中的某一个，这取决于哪一个活跃请求具有最高的优先级。当仲裁者进入到新的状态时，它启动相应的授权信号。仲裁者一直保持在这个状态中，直到被服务的设备取消其请求，此时仲裁者返回状态 A。当仲裁者返回状态 A 的时候，它会对当时任何活跃的请求作出响应，或者等待一个新的请求信号发出。

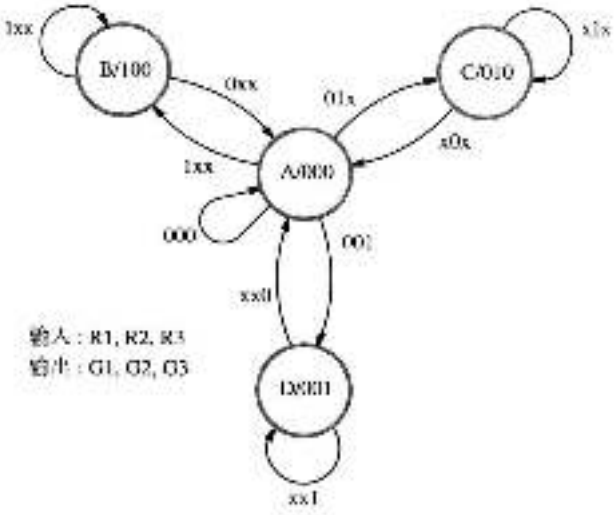



图 7-21　例 7.2 的状态图

**例 7.3**

**问题**：为使用图 7-4 中协议的同步总线设计一个输出接口电路。当数据被写入这个接口电路的数据寄存器时，接口在“新数据”（New-data）线上发出一个宽度为一个时钟周期的脉冲。这个脉冲使得连接到该接口上的输出设备可以获知新数据已经可用。

**解答**：同步总线电路中的所有事件都由一个时钟信号驱动。图 7-22 显示了一种可能的接口电路。“写入数据”（Write-data）信号启用数据寄存器，而后在该时钟周期末尾的时钟边沿将数据装入该数据寄存器中。同时，“新数据”触发器被置为 1。该触发器 Q 输出端的反馈连接在接下来的时钟边沿上将触发器清 0。

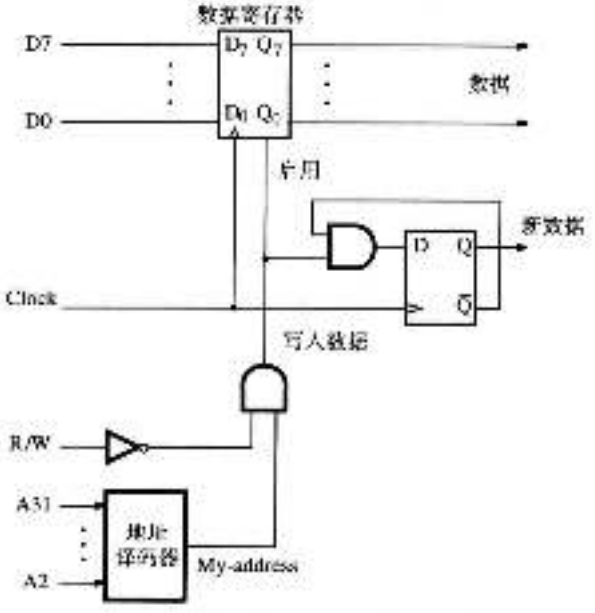



图 7-22　例 7.3 的同步输出接口电路

**例 7.4**

**问题**：画出表示图 7-14 中握手控制电路行为的有穷状态机（FSM）的状态图。

**解答**：图 7-23 给出了一个状态图。其中，电路从状态 A 开始，此时显示设备已准备好接收新数据。因此，New-data = 0，DOIT = 1。一个写操作会使得“写入数据”（Write-data）信号变为 1。这将导致状态机移动到状态 B，并且其输出变成 10。状态机保持在状态 B 中，直到“就绪”（Ready）信号变为 0，表示显示设备已经知道新数据可用了。这时，状态机移动到状态 C，等待显示设备再次做好准备。如果“写入数据”信号还没有变为 0，状态机也必须等待该信号变为 0。

|260|

261
↓
262

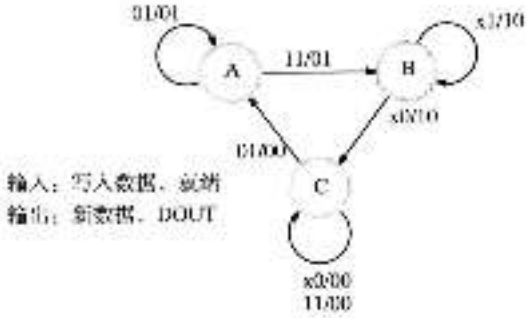



图 7-23 例 7.4 的状态图

## 习题

[E] 7.1 一旦输入数据寄存器被读取，接口电路中表示新数据可用的输入状态位就被清除。为什么要这么做？

[E] 7.2 某计算机有 16 根地址线，$A_{15}$ ~ $A_0$。如果分配给一台设备的十六进制地址是 7CA4，并且该设备的地址译码器忽略 $A_1$ 和 $A_0$ 两根线，这台设备将对哪些地址进行响应？

[M] 7.3 一个处理器有 7 级中断请求线，INTR1 到 INTR7。INTR7 的优先级最高，而 INTR1 的优先级最低。设计一个优先级编码电路，使其可以产生表示最高优先级请求的 3 位代码。

[M] 7.4 图 7-4 ~ 图 7-6 展示了在主控设备与从动设备之间传输数据的三种协议。在读操作期间，如果被寻址的设备因为故障而没有响应，访问在三种不同协议下会发生什么情况？这将会导致什么问题？有什么可行的补救措施？

[E] 7.5 在图 7-5 的时序图中，处理器在总线上保持地址信号，直到接收到设备的响应信号。这是必需的吗？如果处理器只在一个周期内发送地址，则设备端需要什么附加设施？

[E] 7.6 当处理器与 I/O 设备之间的距离增加时，图 7-6 中的时序图会受到什么影响？在图 7-4 中，如何才能适应距离的增大？

[263]

[E] 7.7 有一条同步总线按照图 7-5 中的时序图工作。总线和连接到总线上的接口电路有如下的参数：

总线驱动延迟 2ns
总线传输延迟 5 到 10ns
地址译码器延迟 6ns
提取请求数据的时间 0 到 25ns
准备时间 1.5ns

（a）该总线能够工作的最大时钟速度是多少？

（b）完成一次输入操作需要多少个时钟周期？

[M] 7.8 考虑图 7-6 所示的异步总线协议。使用与习题 7.7 相同的参数，完成一次总线传输的最小和最大时间是多少？假设总线相位偏移为 1ns。

[M] 7.9 图 7-6 中的异步总线协议使用了完全握手协议，其中主控设备在“主控就绪”线上保持一个启动信号，直到它接收到“从动就绪”信号；从动设备保持“从动就绪”线为启动状态，直到“主控就绪”信号变为无效，以此类推。考虑另外一种协议，其中每个信号是一个固定宽度为 4ns 的脉冲。设备只在脉冲的上升沿做出动作。使用与习题 7.7 相同的参数，完成一次总线传输需要的最小和最大时间是多少？

[M] 7.10 在图 7-9 所描述的仲裁者协议示例中，主控设备接收到总线授权信号后保持它的请求线为启动状态，直到它准备放弃总线的控制权。假设有一条公共的线路“忙碌”（Busy），它由当前正在使用总线的主控设备启动。仲裁者只有在“忙碌”信号未被启动的情况下才能对总线授权。一旦主控设备接收到授权信号，它就启动“忙碌”信号并撤销它的请求，作为回应，仲裁者也撤销授权信号。当主控设备使用总线完成工作时就撤销“忙碌”信号。请为这种操作模式画一个与

图 7-9 类似的时序图。

[M] 7.11 针对习题 7.10 中所描述的操作模式，修改列 7.2 中给出的状态图。

[D] 7.12 例 7.2 中的仲裁者控制对公共资源的访问。它不允许抢占，也就是说如果一个高优先级的请求在一个已获得授权的低优先级请求之后到达，则它必须等待直到当前正在使用公共资源的设备完成操作。在某些情况下，允许抢占会更加合适，以便能更快地为高优先级的设备提供服务。这种系统中的设备应能够在仲裁者的要求下停止并放弃对公共资源的使用。这必须在一个安全的方式下完成。正在使用该资源的设备必须能够到达服务可以终止的安全点，然后设备会通知仲裁者它已经停止使用该资源。

（a）给出对该信号协议的一个修改方案，使得正在运行的服务能够安全地终止。

（b）修改仲裁者的状态图，以实现改进后的协议。 264

[E] 7.13 仲裁者控制对公共资源的访问。它使用轮转优先级的方法来对 R1 到 R4 线上的请求进行响应。起初，R1 的优先级最高，R4 的优先级最低。某条线路上的请求获得服务后，那条线路的优先级变成最低，而序列中的下一条线路则获得最高优先级。举例来说，当 R2 被服务后，优先级顺序从最到低依次为 R3，R4，R1，R2。请求序列 R3、R1、R4、R2 的授权序列是什么？假设最后三个请求在第一个请求正在被服务的时候到达。

[E] 7.14 有一个仲裁者使用了习题 7.13 中描述的优先级方法，如果一个设备反复请求服务会发生什么情况？将该仲裁者的行为与使用固定优先级方法的仲裁者进行比较。

[E] 7.15 给出能识别 16 位十六进制地址 FA68 的地址译码器的逻辑表达式。

[M] 7.16 一家工厂使用几个传感器来监控温度、压力和其他因素。每一个传感器都包含一个开关，当相应的参数超过预先设定的限制时会将开关移动到 ON 的位置。需要 8 个这种类型的传感器连接到一台 16 位计算机的总线上。设计一个合适的接口，使得 8 个开关的状态可以作为一个单独的字节被同时读取。假设总线是同步的，并使用图 7-4 中的时序。

[E] 7.17 图 7-4 中的总线协议指定了从动设备只能在时钟的第二个相位中发送数据。

（a）有可能某个设备会识别出它的地址并准备好尽快发送数据。为什么设备不能这么做？处理器是否会接收到错误的数据？

（b）是否还会发生其他的问题？

[M] 7.18 数据存储在输入接口中的一个小存储器中，该接口连接到使用图 7-5 所示协议的同步总线上。总线上的读和写操作由命令线 $R/\overline{W}$ 指示。从存储器中读取数据需要两个时钟周期。设计一个电路以生成该接口的从动就绪响应信号。

[E] 7.19 图 7-19 中 PCI 协议的两个信号 DEVSEL# 和 TRDY# 各自代表一个来自启动设备的响应信号。这两个信号的功能有何不同？

[E] 7.20 考虑图 7-19 所示的 PCI 总线的数据传输操作。如果目标设备在第 2 和第 3 个字之间需要两个时钟周期的延迟，该总线协议将如何处理这种情况？

[E] 7.21 向连接到 PCI 总线上的一个输出设备传送 3 个字，请画出该操作的时序图。 265

## 参考文献


1. *Universal Serial Bus Specification*, available at www.usb.org/developers.
2. *IEEE Standard for a High-Performance Serial Bus*, IEEE Std. 1394-2008, October 2008.
3. Specifications and other information about the *PCI Local Bus and PCI Express* are available at www.pcisig.com/developers.
4. *SCSI-3 Architecture Model (SAM)*, ANSI Standard X3.270, 1996. This and other SCSI documents are available on the web at www.ansi.org.
5. *SATA* specifications and related material are available at www.serialata.org.
6. Information about the *Serial SCSI (SAS)* standard is available at www.scsita.org.
7. A. Wilen, J. Schade, and R. Thornburg, *Introduction to PCI Express, A Hardware and Software Developer's Guide*, Intel Press, 2003.


266

第 8 章

Computer Organization and Embedded Systems, Sixth Edition

# 存储器系统

## 本章目标

在本章中你将学习以下内容：

- 基本的存储器电路
- 主存储器的组织结构
- 存储器技术
- 一种 I/O 机制——直接存储器访问
- 减少有效存储器访问时间的高速缓存
- 增加主存储器表现容量的虚拟存储器
- 用于辅助存储的磁盘和光盘

[267]

程序和程序操作的数据保存在计算机的存储器中。在这一章我们将讨论计算机的这个重要部分是如何运作的。到目前为止，读者已经了解到程序的执行速度很大程度上依赖于指令和数据在处理器与存储器之间传输的速度。此外，足够的存储器对于加快拥有大量数据的大型程序的执行速度也是十分重要的。

从理想的角度考虑，存储器应该是高速度、大容量、而且很廉价的，但不幸的是不可能同时满足这三个要求。在增加速度和容量的同时会导致成本的增加。人们做了很多工作去开发一些结构，这些结构能在合理的成本范围内，提高存储器的有效速度和有效容量。

如第 1 章所述，计算机的存储器构成了一个层次结构，包括高速缓存、主存储器和辅助存储器。在本章中，我们将描述用于实现这些部件的最常用的组件和组织结构。引入直接存储器访问作为在 I/O 设备（如磁盘）和主存储器之间传输数据的机制，而处理器只需最低限度地参与传输过程。我们将分析存储器的速度，并讨论如何通过使用高速缓存的方法来减少存储器数据的存取时间。接着我们将介绍利用辅助存储设备的大存储容量来增加存储器有效容量的虚拟存储器概念。我们首先介绍一些基本概念，以扩充第 1 章和第 2 章中所讨论的内容。

## 8.1 基本概念

任何一台计算机中能使用的存储器最大容量取决于寻址方式。例如产生 16 位地址的计算机能寻址 $2^{16}$ = 64K（kilo）个存储单元。能产生 32 位地址的机器能使用包含 $2^{32}$ = 4G（giga）个单元的存储器，而 64 位地址的机器能访问 $2^{64}$ = 16E（exa）= $16 \times 10^{18}$ 个单元。单元的数目表示计算机地址空间的大小。

存储器通常被设计成按字存储和检索数据。例如，考虑一台能产生 32 位地址的按字节编址的计算机，当处理器给存储器发送一个 32 位地址时，其中高 30 位的地址决定哪一个字被访问。如果只是访问一个字节，那么最低的两位地址指定哪一个字节被访问。

处理器和存储器之间的连接由地址线、数据线和控制线组成，如图 8-1 所示。处理器使用地址线指明数据传输操作中的存储单元，使用数据线传输数据。同时，控制线包含指示读或写操作以及传输一个字节还是一个字的命令。控制线还提供必要的时序信息，也可以被存储器用来指明它何时完成所请求的操作。当处理器—存储器接口接收到存储器的响应，它发出如图 5-19 所示的 MFC 信号。这是处理器的内部控制信号，用来指示所请求的存储器操作已经完

成。当发出该信号后，处理器继续进行执行序列中的下一步。 [268]

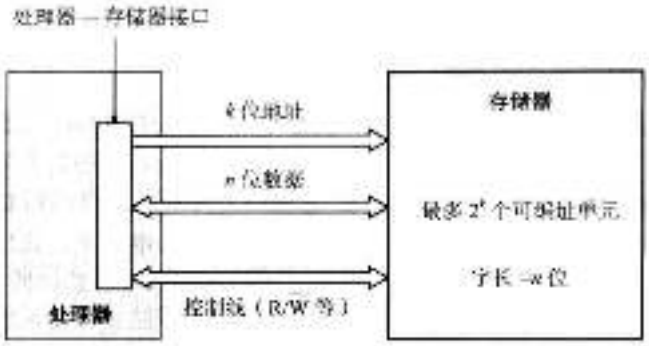



图 8-1　存储器到处理器的连接

对存储器速度的一个有用测量标准是从开始传输一个数据字的操作到结束传输所用的时间，这被称为存储器访问时间（memory access time）。另一个重要的衡量标准是存储器周期时间（memory cycle time），它是指两个连续的存储器操作开始时刻之间的最小时间延迟，例如两个连续的读操作之间的时间。一般来说，周期时间比存取时间稍长，这与存储器的实现细节有关。

如果任何一个存储单元的访问时间都相同，而且与存储单元的地址无关，那么这个存储器就称为随机访问存储器（random-access memory，RAM）。这个定义把随机访问存储器与连续访问的存储设备或部分连续访问的存储设备，例如磁盘和光盘区分开来，后者的访问时间依赖于数据的地址或位置。

实现计算机存储器的技术是使用半导体集成电路，后面的章节将介绍一些关于随机存储器的内部结构和操作的基本情况，然后我们再讨论增加存储器有效速度和容量的技术。

### 1. 高速缓存和虚拟存储器

计算机处理器处理指令和数据的速度通常比从主存储器中获取指令和数据的速度快，因此，存储器访问时间是整个系统的瓶颈。减少存储器访问时间的一个办法是使用高速缓存（cache memory）。高速缓存是一个容量小、速度快的存储器，它在容量大、速度慢的主存与处理器之间，保存当前活动的程序部分和数据。

虚拟存储器（virtual memory）是关于存储器结构的另一个重要概念。使用这种技术，只需将程序的活动部分存放在主存储器中，剩余部分存放在较大容量的辅助存储设备中。程序段在主存储器和辅助存储设备之间来回地传输，这对应用程序来说是透明的。因此，应用程序所 [269]
看到的存储器比计算机中的物理主存储器大得多。

### 2. 块传输

上面的讨论表明数据经常在主存和高速缓存之间、主存和磁盘之间移动。这些传输不是以每次一个字的方式发生的，数据通常是以包含几十、几百甚至几千个字的邻近块的方式进行传输的。主存和图形显示器或以太网接口这样的高速设备之间的数据传输也包含大块的数据。因此，主存性能的一个关键参数是其高速读写数据块的能力。这是我们在讨论存储器技术和存储器系统的结构时会反复遇到的一个重要考虑因素。

## 8.2　半导体随机存储器

半导体随机存储器（RAM）可以在很广泛的速度范围内使用。它们的周期时间可以从

100ns 到小于 10ns。在这一节中，我们将讨论这类存储器的主要特性。首先介绍存储器单元在芯片内的组织方法。

### 8.2.1 存储器芯片的内部组织结构

存储器单元通常按阵列的形式构成，其中每个单元存储一位（bit）信息。图 8-2 描述了一种可能的组织方式。每一个单元行组成存储器中的一个字，一行中的所有单元通过一根公共的线连接到一起，这根线称为字线（word line），它是由芯片的地址译码器驱动的。每列的单元通过两条位线（bit line）连接到一个读出/写（Sense/Write）电路上。读出/写电路连接到芯片的数据输入/输出线上。在读操作期间，这些电路读出通过字线选择的单元所存储的信息，并把这些信息放到输出数据线上。在写操作期间，读出/写电路接收输入数据，并把它们存储到选定的字对应的单元中。

图 8-2 是一个非常小的存储器电路的例子，它包含 16 个字，每个字包括 8 位。这被称为 $16 \times 8$ 结构。每一个读出/写电路的数据输入和数据输出连接到一条双向数据线上，该双向数据线可以连接到计算机的数据线上。还有两条控制线 R/W 和 CS。R/$\overline{W}$（Read/$\overline{Write}$）输入指定请求的操作，而 CS（芯片选择）输入可在多芯片存储器系统中选择一个给定的芯片。

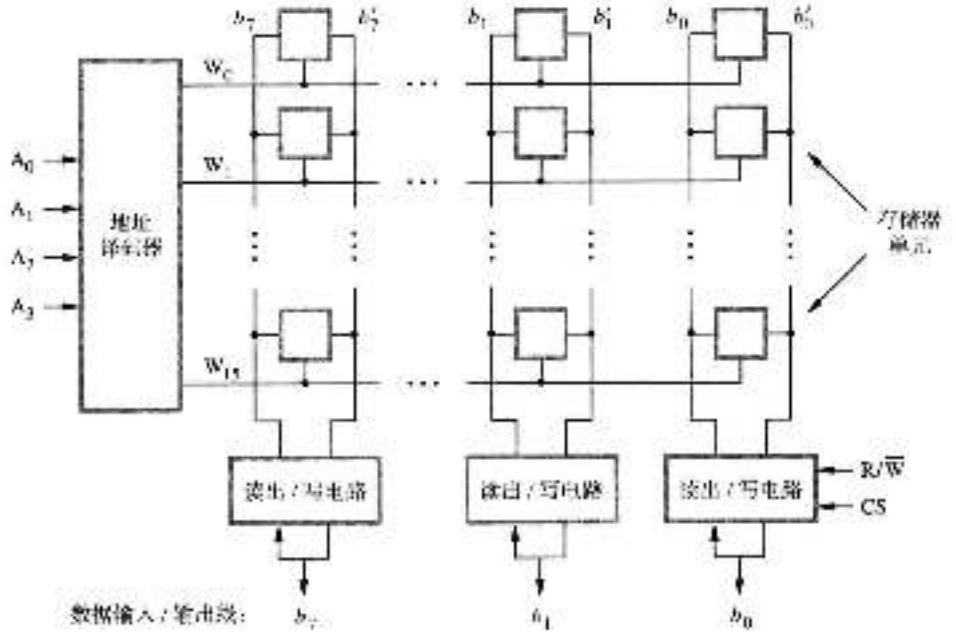



图 8-2 存储器芯片内部位单元的组织结构

图 8-2 中的存储器电路存储了 128 位数据，需要为地址、数据和控制提供 14 条外部连接线。它还需要两条线用于电源支持和接地。现在来考虑一个稍微大一些的存储器电路，它包含 1K（1024）个存储器单元。这个电路可以被组织成 $128 \times 8$ 的存储器，总共需要 19 根外部连接线。换一种方式，我们还可以把这个电路组织成 $1K \times 1$ 的形式。在这种情况下，需要 10 位地址，但只需要一根数据线，因此共需要 15 根外部连接线。图 8-3 显示了这种组织结构。需要的 10 位地址被分成两组，每组 5 位，分别构成单元阵列的行地址和列地址。行地址选择一个行，包含 32 个存储器单元，它们被并行访问。但是，根据列地址，这些单元中只有一个连接到外部数据线上。

商业上可用的存储器芯片所包含的存储器单元数比图 8-2 和图 8-3 显示的例子要多得多，我们使用小的例子可以使得图更容易理解。大规模芯片的组织结构从本质上来说与图 8-3 所示的一样，只不过使用更大的存储器单元阵列，并且有更多的外部连接线。例如，一个 1G 芯片的组织结构可能是 256M × 4，这时需要 28 位地址，可向或从芯片传输 4 位数据。

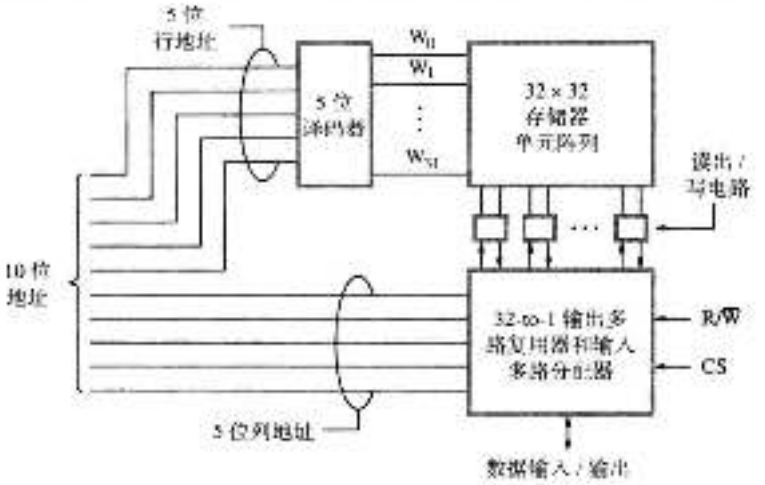



图 8-3　1K × 1 的存储器芯片组织结构

## 8.2.2　静态存储器

有的电路只要不停止供电，就能一直保持它的状态，由这样的电路组成的存储器称为静态存储器（static memory）。图 8-4 描述了如何实现静态随机存储器（SRAM）单元。两个反相器交叉耦合形成一个锁存器，锁存器通过晶体管 $T_1$ 和 $T_2$ 连接到两根位线上，这些晶体管起到开关的作用，可以在字线的控制下打开或关闭。当字线在低电平时，晶体管断开，锁存器保持它的状态。例如，如果 $X$ 点的逻辑值为 1 且 $Y$ 点的逻辑值为 0，只要字线的信号处于低电平，这个状态将一直保持下去。假设这个状态代表值 1。

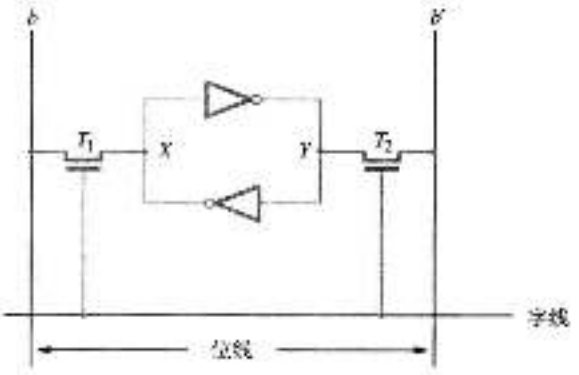



图 8-4　静态随机存储器单元

270
/
272

### 1. 读操作

为了读取 SRAM 单元的状态，字线被激活以闭合开关 $T_1$ 和 $T_2$。如果单元处于状态 1，那么位线 $b$ 上的信号是高电平，而位线 $b'$ 上的信号是低电平。如果单元的状态为 0，则正好相反。因此，$b$ 和 $b'$ 总是互补的。位于这两根位线末端的读出 / 写电路监测它们的状态，并相应地设置对应的输出。

### 2. 写操作

在写操作期间，读出 / 写电路驱动位线 $b$ 和 $b'$，而不是读出它们的状态。它把适当的值放到位线 $b$ 上，并把它的补值放到 $b'$ 上，激活字线，这样就迫使单元进入相应的状态，当字线失效时，单元就一直保持这个状态。

### 3. CMOS 单元

实现图 8-4 中存储器单元的 CMOS 单元在图 8-5 中给出。晶体管对 ($T_3$, $T_5$) 和 ($T_4$, $T_6$) 形成锁存器中的反相器（见附录 A）。单元的状态就像刚才解释的那样被读写。例如，位于状态 1 时，$X$ 点保持在高电平，此时晶体管 $T_3$ 和 $T_6$ 是打开的，而 $T_4$ 和 $T_5$ 是关闭的。如果 $T_1$ 和 $T_2$ 打开，位线 $b$ 和 $b'$ 将分别具有高电平和低电平信号。

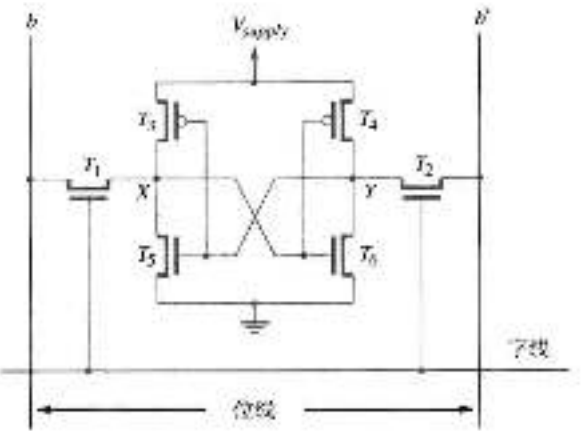



图 8-5 CMOS 存储器单元的例子

存储器单元需要持续的电压供应来维持它的状态。如果电源中断了，单元中的内容就会丢失。当电源恢复时，单元中的锁存器进入一个稳定状态，但这个状态不一定与电源中断前单元的状态相同。因为存储的内容在电源中断后会丢失，所以 SRAM 被称为易失性（volatile）存储器。

CMOS 静态随机存储器的一个主要优点是它们的功耗非常低，因为只有被访问时电流才流入存储器单元。此外，$T_1$，$T_2$ 和每个反相器中的一个晶体管都是关闭的，这样保证在 $V_{supply}$ 和地之间没有持续的电气通路。

静态随机存储器的访问速度非常快，商用的芯片中已经有访问时间大约为几纳秒的芯片。SRAM 通常用于速度至关重要的应用场合中。

## 8.2.3 动态随机存储器

静态随机存储器速度很快，但是它们的存储器单元需要多个晶体管。使用更简单的存储器单元可以实现更廉价的、更高密度的随机存储器。但是，这些更简单的单元不能长期地保持它们的状态，除非它们被经常访问来进行读或写操作。使用这种单元的存储器被称为*动态随机存储器*（DRAM）。

信息以电容中电荷的形式保存在动态存储器单元中，但这些电荷只能保持几十毫秒。因为我们需要存储器单元在更长的时间内保存信息，因此必须通过把电容中的电荷恢复成满值来定期*刷新*（refresh）存储器单元中的内容。当存储器单元中的内容被读出或者当新的信息写入存储器单元的时候会对存储器单元进行刷新。

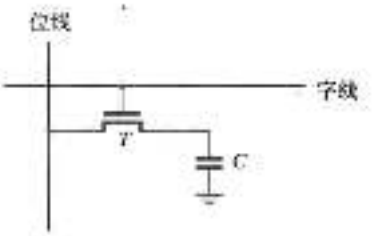



图 8-6 单晶体管动态存储器单元

图 8-6 显示了一个由电容 $C$ 和晶体管 $T$ 组成的动态存储器单元的例子。为了在单元中存储信息，晶体管 $T$ 打开，并将一个适当的电压加到位线上，这使得一定数量的电荷被存储到电容中。

晶体管关闭后，电荷仍然存储在电容中，但是保存时间不长，电容就开始放电，这是由于晶体管关闭后，仍会有微弱的电流通过，这个电流只有几皮安。因此只有在电容中的电荷降到低于某个阈值之前，才能正确读出存储器单元中存储的信息。在读操作期间，选中单元的晶体管会打开，连接到位线上的一个传感放大器检测电容中所存储的电荷是否高于或低于某个阈值。如果电荷高于阈值，那么传感放大器把位线上的电压提高成满电压以表示逻辑值 1。结果，电容被重新充满电荷，使它对应逻辑值 1。如果传感放大器检测到电容中的电荷低

于阈值，它就把位线拉成低电平，使电容完全放电。因而，读取存储器单元内容的时候会自动刷新它的内容。因为字线对于一行中的所有单元来说是共用的，所以选定行中所有的单元被同时读取和刷新。

图 8-7 显示了一个 256M 位的 DRAM 芯片，被配置成 32M × 8 的形式。存储器单元被组织成 16K × 16K 的阵列。每行的 16384 个单元被分成 2048 组，每组 8 个单元，从而形成 2048 字节的数据。选择一行需要 14 位地址，在一个选定的行中指定一个 8 位的组还需要 11 位地址。因此，在这个存储器中访问一个字节总共需要 25 位地址，高 14 位和低 11 位分别构成一个字节的行地址和列地址。为了减少外部连接的引脚数，行地址和列地址多路复用 14 根引脚。在读或写操作期间，行地址首先被加载。芯片响应被称为行地址选通（RAS）的输入控制线上的脉冲信号，把行地址装入行地址锁存器中。这将导致一个读操作开始，选定行中的所有单元被读取和刷新。

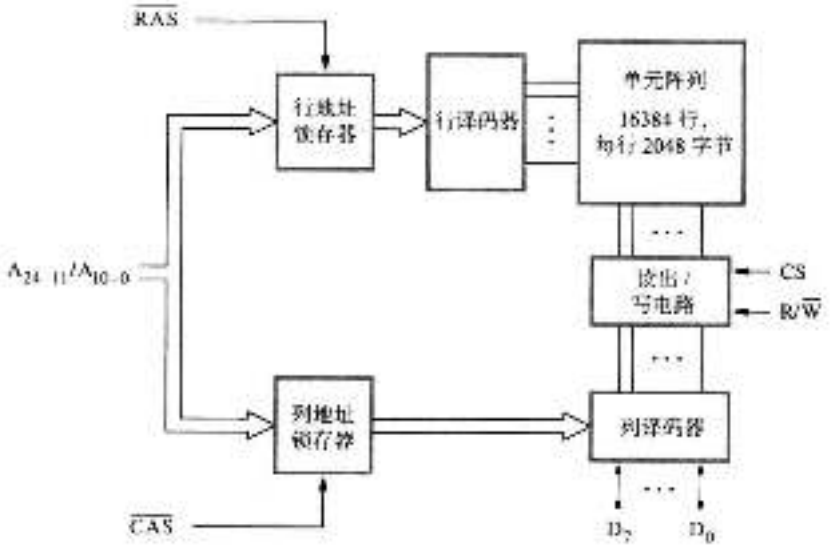



图 8-7　32M × 8 的动态存储器芯片的内部组织结构

275

行地址被装载后，列地址立刻被加载到地址引脚上，然后在第二根被称为列地址选通（CAS）的控制线的控制下，列地址被装入列地址锁存器中。列地址锁存器中的信息被译码，该列地址对应的一个 8 位组的读出/写电路被选中。如果 R/$\overline{W}$ 控制信号指示这是一个读操作，那么选定电路的输出值被传输到数据线 $D_{7..0}$ 上。如果是一个写操作，$D_{7..0}$ 线上的信息被传输给选定的电路，然后用这些信息覆盖选定的 8 列存储器单元的内容。我们应该注意，在商业 DRAM 芯片中，RAS 和 CAS 控制信号是低电平有效，因此当这些信号从高电平变成低电平时，地址被锁存。为了表示这种情况，在图中这些信号被表示成 $\overline{RAS}$ 和 $\overline{CAS}$。

上面描述的 DRAM 操作的时序是由 RAS 和 CAS 信号控制的。当处理器发出一个读或写命令时，这些信号由芯片外部的一个存储控制器电路产生。在读操作期间，在一个相当于存储器访问时间的延迟后，输出数据被传输到处理器。这种存储器称为异步动态随机存储器（asynchronous DRAM）。存储控制器还负责对存储在存储器芯片中的数据进行刷新，我们稍后将描述。

**快速页模式**

当图 8-7 所示的 DRAM 被访问时，选中行的 16384 个单元的内容都被读取，但是只有 8

位信息被放在数据线 $D_{i,0}$ 上，这个字节通过列地址位 $A_{m,0}$ 来选择。可以对电路做一个简单的修改，使得在访问同一行中其他字节的时候不需要重新进行行选择。每个传感放大器也可用作一个锁存器，当加载行地址的时候，选定行中所有单元的内容都被装入相应的锁存器中。于是，只要加载不同的列地址就能把该行的其他字节放到数据线上。

这种方式将会导致一个非常有用的功能，即，在连续的 CAS 信号控制下通过加载连续的列地址序列就可以按顺序传输选定行中的所有字节。因此，一块数据的传输要比使用随机地址传送的速率快得多。这种块传输能力被称为快速页模式（fast page mode）。（一大块数据通常被称为一页。）

以前我们就指出过，绝大多数的主存储器事务都涉及块传输。使用快速页模式获得的高速率特性使得动态随机存储器特别适合这种环境。

### 8.2.4 同步动态随机存储器

20 世纪 90 年代初，随着存储器技术的发展，出现了操作与时钟信号同步的动态随机存储器，这种存储器被称为同步动态随机存储器（synchronous DRAM，SDRAM）。其结构如图 8-8 所示。它的单元阵列与异步动态随机存储器一样。SDRAM 的显著特点是使用时钟信号，这使得它可以在芯片上包含控制电路，提供许多有用的功能。例如，SDRAM 有内置的刷新电路，用一个刷新计数器来提供要刷新的行地址。因此，这些存储器芯片的动态特性对用户来说几乎
276 是不可见的。

SDRAM 的地址和数据连接可以如图 8-8 中所示的那样通过寄存器进行缓冲。同异步 DRAM 一样，读出/写放大器用作锁存器。读操作使得选中行中所有单元的内容被装入锁存器。锁存器中选中列的数据被传输到数据寄存器中，然后就可以在数据输出引脚上获得这些数据。当以很快的速度传输大块数据时，缓冲寄存器是非常有用的。通过将外部连接与芯片的内部电路分离开来，在向或从寄存器传输数据时，就有可能开始一个新的数据访问操作。

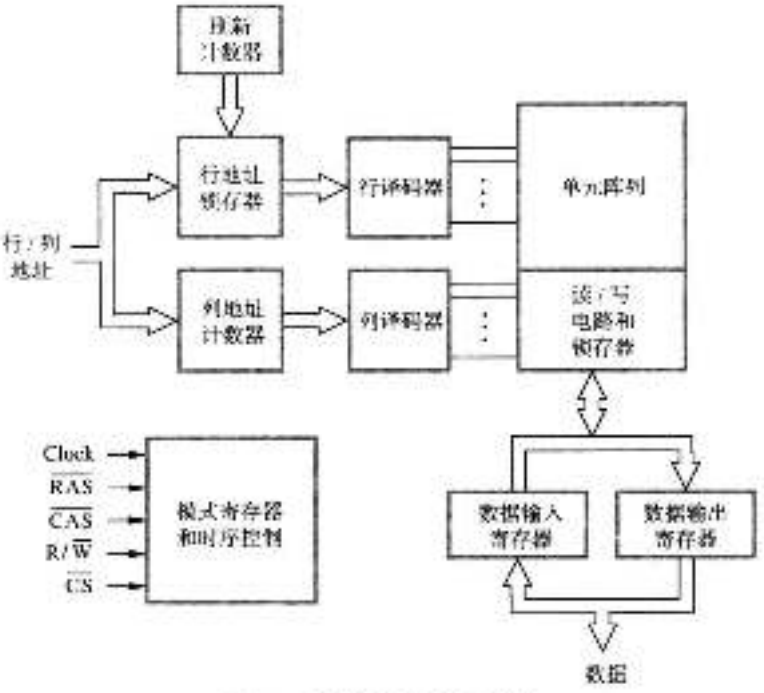



图 8-8 同步动态随机存储器

SDRAM 有几种不同的操作模式，可以通过向模式（mode）寄存器中写入控制信息来选择这些模式，例如可以指定不同长度的块操作。选择连续的列不需要在 CAS 线上提供外部产生的脉冲，芯片内部使用列计数器和时钟信号就能产生所需要的控制信号。新数据在每个时钟脉冲的上升沿被放到数据线上。

图 8-9 是一个典型的长度为 4 的读脉冲串时序图。首先，行地址在 RAS 信号的控制下被锁存。存储器通常使用 5 个或 6 个时钟周期（为简单起见，在图中我们使用 2 个）激活选中的 277
行。然后，列地址在 CAS 信号的控制下被锁存。一个时钟周期的延迟后，第一组数据被放到数据线上。然后 SDRAM 自动增加列地址去访问选定行中后面的三组数据，在随后的 3 个时钟周期中把这三组数据放到数据线上。

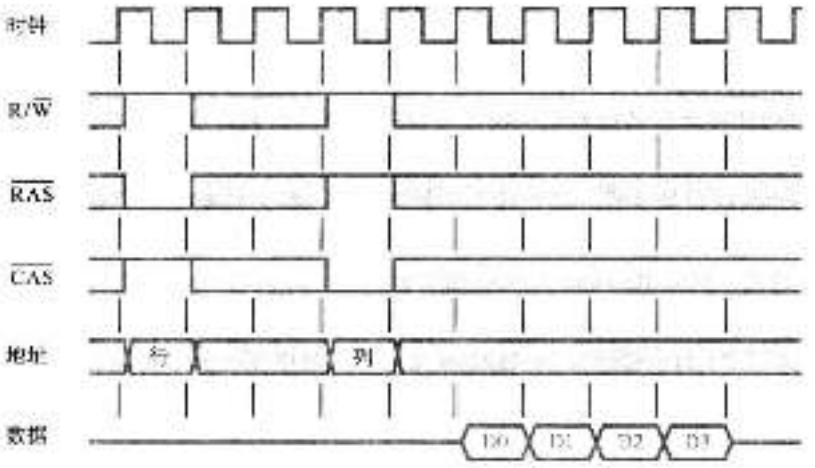



图 8-9　在 SDRAM 中长度为 4 的读脉冲串

同步 SDRAM 可以以非常快的速度传送数据，因为所需要的所有控制信号都是在芯片内部产生的。20 世纪 90 年代设计出了最初的商业 SDRAM，其时钟频率高达 133MHz。随着技术的发展，开发出了更快速的 SDRAM 芯片。现在的 SDRAM 可以在超过 1GHz 的时钟速度下工作。

1. 延迟与带宽

向或从主存储器传输的数据通常包括数据块。这些传输的速度对计算机系统的性能影响很大。在传输数据块时，前面定义的存储器访问时间就不足以描述存储器的性能了。在块传输期间，存储器延迟（memory latency）是传输块的第一个字所花费的时间。传输一个完整的块所需要的时间还依赖于后继字的传输速度以及块的大小。传输块中后继字的时间比传输第一个字所需要的时间要短得多。例如，在图 8-9 所示的时序图中，当 RAS 信号出现时，访问周期开始，第一个字在五个时钟周期后传输，因此延迟是五个时钟周期。如果时钟频率是 500MHz，那么延迟是 10ns。剩下的二个字在后续的时钟周期中传输，每 2ns 传输一个字。

上面的例子表明，在块传输期间，我们需要除了存储器延迟之外的另一个参数来描述存储器的性能。一个有用的性能衡量标准是一秒钟之内能传输的位或字节的数量，这个衡量标准 278
通常被称为存储器带宽（bandwidth）。它依赖于访问存储数据的速度和能并行访问的位数。向或从存储器传输数据的速率依赖于系统互连的带宽。为此，所使用的互连需要始终确保处理器和存储器之间的数据传输的可用带宽是非常高的。

2. 双倍数据速率 SDRAM

通过对提高性能的不断探索，SDRAM 的更快版本已被开发出来。除了采用更快的电路外，新的组织和操作特点使它有可能在块传输期间获得高速的数据传输率。其核心思想是利用存储器在加载一个行地址时，会同时访问芯片内大量的位这个特点。可使用各种技术将这些位快速传输到芯片的引脚上。为了充分利用可用的时钟速度，在时钟的上升沿和下降沿都传输数据。因此，使用这种技术的存储器称为双倍数据速率 SDRAM（double-data-rate SDRAM，DDR SDRAM）。

已经开发出了多个版本的 DDR 芯片。最早的版本被称为 DDR。后来的版本 DDR2、DDR3 和 DDR4 具有增强的功能。它们提供了更大的存储容量、更低的功率以及更快的时钟速度。例如，DDR2 和 DDR3 可分别在 400MHz 和 800MHz 的时钟频率下工作。因此，它们分别使用 800MHz 和 1600MHz 的有效时钟速度传输数据。

3. Rambus 存储器

在存储器和处理之间传输数据的速率是存储器的带宽和存储器到处理器的连接的带宽的函数。Rambus 是一种可以获得高速的数据传输率的存储器技术，它通过在存储器和处理器之间提供一个高速的接口来实现。增加这个连接带宽的一种方法是使用更宽的数据通路。然而，这需要更大的空间和更多的引脚，从而增加了系统的成本。另一种方法是使用较少的线路但具有更高的时钟速度。这是 Rambus 公司采用的方法。

Rambus 技术的关键特性是在向或从存储器芯片传输数据时使用了差分信号技术。差分信号的基本思想在 7.5.1 节中描述。在 Rambus 技术中，使用在一个参考值上下 0.1V 浮动的微小电压来传输信号。该标准已经开发出了多个版本，时钟速度高达 800MHz，数据传输率达到每秒数千兆字节。

Rambus 技术直接与 DDR SDRAM 技术竞争，它们各自有各自的优点和缺陷。一个非技术因素是，DDR SDRAM 的规范是一个开放的标准，可以免费使用。而另一方面，Rambus 是一个私有的方案，必须由芯片生产商授权才能使用。

### 8.2.5 大容量存储器的结构

我们已经讨论了存储器电路的基本组织结构，它们可以在一个单独的芯片上实现，下面
[279] 来分析存储器芯片如何连接到一起形成更大的存储器。

1. 静态存储器系统

考虑一个由 2M 个 32 位的字组成的存储器，图 8-10 显示了如何使用 512K × 8 的静态存储器芯片来实现这个存储器。图中每一列实现了一个字中一个字节的位置，用四个芯片提供 2M 个字节。四列合起来实现了所需要的 2M × 32 的存储器。每一个芯片有一个控制输入，被称为芯片选择（Chip Select）。当这个输入被置成 1 时，允许芯片从数据线上接收数据或把数据放到数据线上。每个芯片的数据输出都是 7.2.3 节中所描述的三态类型的，只有选中的芯片才能把数据放到数据输出线上，所有其他芯片的输出都与数据线断开连接。选择存储器中一个 32 位的字需要 21 个地址位。地址的高 2 位被译码，用来决定选择 4 行中的哪一行。其余的 19 个地址位用来在选中行的每个芯片内指定访问哪个字节位置。所有芯片的 R/W 输入连到一起提供一个共用的 Read/Write 控制线（没有在图中显示）。

2. 动态存储器系统

现代计算机使用非常大的存储器。即使是小型的个人计算机也可能至少有 1G 字节的内存，典型的台式计算机可能有 4G 字节或更大的内存。大容量的内存会带来更好的性能，因为

更多当前正被处理的程序和数据能被保存在内存中，从而能降低访问辅助存储设备的频率。

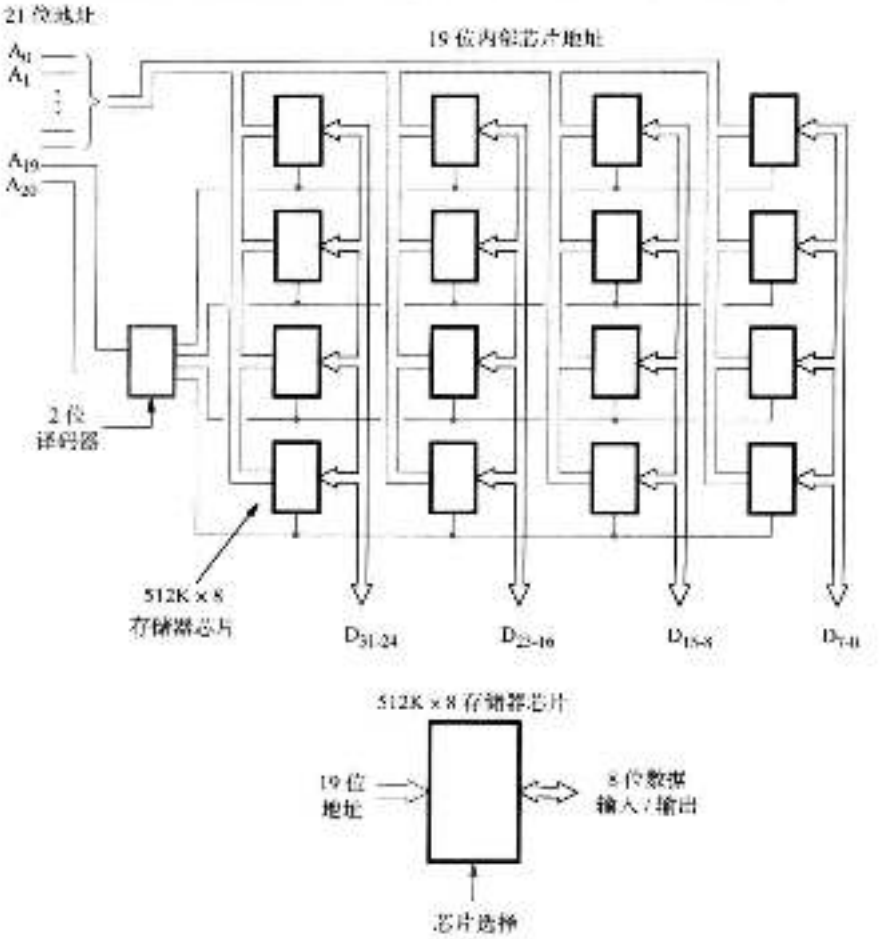



图 8-10　使用 512K × 8 静态存储器芯片的 2M × 32 存储器模块的组织结构

因为位密度高，成本低，动态随机存储器（大多是同步类型的）被广泛应用在计算机的存储器中。它们比静态随机存储器慢，但它们使用较少的电量，有相当低的位价格。可用的芯片有高达 2G 位的容量，甚至更大容量的芯片也正在开发中。为了减少给定计算机中所需要的存储器芯片的数量，存储器芯片可被组织成能并行读或写很多位的形式，如图 8-7 所示的情况。芯片被生产成不同的组织结构，这样在设计存储器系统时可以提供足够的灵活性。例如，1G 位的芯片可以被组织成 256M × 4 或者 128M × 8。

封装因素导致了被称为存储器模块的组装技术的发展。每一个这样的模块将许多存储器芯片（通常为 16 到 32 个）放置到一块小板上，然后再把小板插入到计算机主板的一个槽中。根据引脚的配置，存储器模块通常被称为 SIMM（Single In-line Memory Module，单列直插存储器模块）或 DIMM（Dual In-line Memory Module，双列直插存储器模块）。不同容量的模块被设计成使用相同的槽，例如，128M × 64 位、256M × 64 位和 512M × 64 位的 DIMM 都使用 240 针的槽。这样，通过把使用相同槽的小模块替换成大模块的方式，存储器的总容量很容易进行扩展。

### 3. 存储控制器

正如前面所解释的那样，加载到动态随机存储器芯片上的地址被分成两个部分，高地址位用来选择存储器单元阵列的行，它们首先被提供，并在 RAS 信号的控制下锁存在存储器芯片中。低地址位用来选择列，它们随后通过相同的地址引脚提供，并在 CAS 信号的控制下锁存。因为典型的处理器同时发出所有的地址位，所以需要一个多路复用器。这种多路复用功能通常由存储控制器（memory controller）电路完成。控制器在请求（request）信号的控制下从处理器接收一个完整的地址和 R/W 信号，请求信号表示需要一个存储器访问操作。控制器把
280 / 281
R/W 信号和行地址、列地址传给存储器，并产生具有正确时序的 RAS 和 CAS 信号。当存储器包含多个模块时，可根据地址的高位部分选择其中的一个模块。存储控制器将这些高地址位进行译码，生成选择适当模块的芯片选择信号。处理器和存储器之间的数据线是直接连接的。

动态随机存储器必须定期刷新。发起刷新周期所需的电路是同步 DRAM 内部控制电路的一部分。然而，异步 DRAM 需要芯片外部的控制电路来发起周期性的读周期以刷新其存储器单元，存储控制器可以提供这种功能。

### 4. 刷新开销

当内部刷新操作正在进行时，动态随机存储器不能响应读或写请求。这些请求被延迟，直到刷新周期结束。但是，为完成刷新操作所损失的时间是非常小的。例如，考虑一个 SDRAM，其中每一行需要每隔 64ms 刷新一次。假设访问两行之间的最短时间为 50ns，刷新操作要被安排成可使得芯片的所有行在 8K（8192）个刷新周期中被刷新。因此，需要 $8192 \times 0.050 = 0.41$ms 来刷新所有行。刷新开销是 $0.41 / 64 = 0.0064$，它小于访问存储器可用总时间的 1%。

### 5. 技术的选择

为给定的应用选择随机存储器芯片要考虑多个因素，其中最重要的是成本、速度、功耗和芯片容量。

静态随机存储器的特征是它们的高速操作。但是，它们的成本和位密度受实现基本单元所需电路的复杂程度的影响。静态随机存储器大部分用于小容量的但速度非常快的存储器中。而另外，动态随机存储器具有较高的位密度和较低的位价格。同步动态随机存储器是实现主存储器的主要选择。

## 8.3 只读存储器

静态随机存储器和动态随机存储器芯片都是易失的，这意味着只有当电源打开时，它们才能保持信息。有很多应用要求存储器设备在电源关闭后仍能保留它所存储的信息。例如，第 4 章描述了需要在这样的存储器中存储一个小程序，用于启动引导过程，来将操作系统从硬盘装入主存中。第 10 章和第 11 章描述的嵌入式应用是另一个重要的例子。许多嵌入式应用不使用硬盘而需要使用非易失性的存储器来存储它们的软件。

现在已经开发出了不同类型的非易失性存储器。通常，可以采用上面讨论的易失性存储器所用的方法来读取非易失性存储器的内容。但是，把信息存入非易失性存储器需要一个特殊的写过程。因为它们一般的操作只涉及读取所存储的数据，因此这种类型的存储器称为只读存储器（read-only memory，ROM）。
282

### 8.3.1 ROM

信息只在存储器生产时被写入一次，这样的存储器称为只读存储器，或者 ROM。图 8-11

显示了一种可能的 ROM 单元结构，如果晶体管在 $P$ 点接地，那么单元存储的逻辑值为 0，否则为 1。位线通过一个电阻连到电源线上。要读取单元的状态，需要激活字线以闭合晶体管开关。因此，如果晶体管接地的话，位线上的电压会降到接近 0 电位；如果没有接地的话，位线上的电压仍然保持高电位，表示逻辑值 1。位线末端的传感电路产生正确的输出值。每个单元中的接地状态是在芯片制造时通过使用一个掩码来决定的，该掩码表示要存入信息的模式。

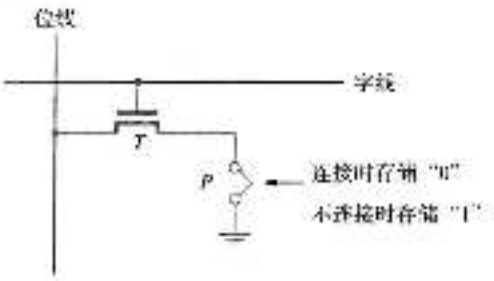



图 8-11　只读存储器单元

### 8.3.2　PROM

有些 ROM 设计成由用户装入数据，即可编程只读存储器（programmable ROM，PROM）。可编程能力是通过在图 8-11 中的 $P$ 点插入一根熔丝实现的。在编程之前，存储器包含的内容全是 0。用户可以通过使用高电流脉冲把所需位置的熔丝烧断，使那些位置的值为 1。当然，这个过程是不可逆的。

PROM 提供了 ROM 不具备的灵活性和方便性。为准备存储特定信息需要的掩码所付出的成本使得 ROM 仅在大批量生产时比较划算。另一种 PROM 技术提供了一种更方便并且非常廉价的方法，因为存储器芯片可以直接由用户编程。

283

### 8.3.3　EPROM

另一种 ROM 芯片提供了更高层次的方便性，它允许擦除存储的数据，并写入新的数据。这种可擦除（erasable）、可再编程的 ROM 通常被称为可擦除可编程只读存储器（EPROM），它在数字系统的开发阶段提供了相当大的灵活性。因为 EPROM 能在很长的时间内保存数据，所以当软件正在被开发的时候，可以用它们来替换 ROM 或 PROM。通过这种方法，可以很容易地实现存储信息的改变和升级。

EPROM 单元的结构与图 8-11 中的 ROM 单元相似。但是，$P$ 点是通过一个特殊的晶体管接地的，这个晶体管通常是关闭的，形成一个打开的开关。可以通过向晶体管注入电荷来打开晶体管，注入的电荷在晶体管内部被捕获。这样，EPROM 单元可以用前面讨论的 ROM 单元所用的方法来构建存储器。擦除需要驱散组成存储器单元的晶体管所捕获的电荷，这可以通过把芯片暴露在紫外线下照射来完成，紫外线可擦除芯片上所有的内容。为了做到这一点，EPROM 芯片被安装在有透明窗口的外壳中。

### 8.3.4　EEPROM

EPROM 重新编程时必须从电路上物理地移除。并且，所存储的信息不能有选择地擦除，当芯片暴露在紫外线下照射时，它所有的内容都会被擦除。另一种可擦除的 PROM 可以用电来编程、擦除和再编程，这样的芯片叫做电可擦除可编程只读存储器（electrically erasable PROM，EEPROM），擦除时不需要把它从电路上移除，而且，还可以有选择地擦除存储器单元中的内容。EEPROM 的一个不足是擦除数据、写数据和读数据时需要用不同的电压，这增加了电路的复杂性。然而，这个不足被 EEPROM 的许多优点超过，在实践中它们已经取代了 EPROM。

### 8.3.5 闪存

一种与 EEPROM 技术类似的方法出现了，它就是闪存（flash memory）设备。闪存单元基于一个由捕获电荷控制的单独晶体管，很像 EEPROM 单元。在闪存中可以读取单个单元的内容，这跟 EEPROM 也很类似。关键的不同是，在闪存中只能写入整块单元的内容。在写之前，这个块以前存储的内容被擦除。闪存有更高的密度，这可以带来更高的容量和更低的位成本。它们只需要单一的电压，而且在操作中消耗更少的能量。

低功耗使得闪存在便携式、电池供电的设备应用中有很强的吸引力。典型的应用包括手持计算机、蜂窝电话、数码相机和 MP3 音乐播放器。在手持计算机和蜂窝电话中，闪存保存操作设备所需要的软件，这样就免除了对磁盘的需求。在数码相机中，闪存用来保存图片数
284 据。在 MP3 播放器中，闪存存储表示声音的数据。蜂窝电话、数码相机和 MP3 播放器是很好的嵌入式系统的例子，这些将在第 10 章和第 11 章讨论。

单独的闪存芯片不能为上面提到的应用提供足够的存储容量，可以使用由多个芯片组成的更大的存储器模块。实现这样的模块时有两种常见的选择：闪存卡和闪存驱动器。

#### 1. 闪存卡

构建更大模块的一个方法是把闪存芯片安装到一个小卡片上，这样的闪存卡有一个标准的接口，这使得它们可以用在不同的产品上。卡片只需要简单地插入到一个可以很方便接触到的槽上。具有 USB 接口的闪存卡被广泛地使用，通常被称为存储键。它们有不同的容量，较大的闪存卡可以存储多达 32G 字节的数据。使用 MP3 编码格式时，一分钟的音乐可以存储在大约 1M 字节的存储器中，因此，一个 32G 字节的闪存卡可以存储大约 500 小时的音乐。

#### 2. 闪存驱动器

更大容量的闪存模块已经被开发出来，可以替换硬盘驱动器，因此被称为闪存驱动器。它们完全仿效硬盘设计，可以装入标准的磁盘驱动器架中。但是闪存驱动器的存储容量太低了，目前闪存驱动器的容量大约为 64G 到 128G 字节。相比之下，硬盘的容量已超过 1T 字节。另外，磁盘驱动器每位的成本非常低。

闪存驱动器是没有可移动部分的固态电子设备，这为它们提供了一些超过磁盘驱动器的重要优点。它们有更短的访问时间，从而有更短的响应时间。它们受震动的影响很小，并且有更低的功耗，这使得它们在便携式的、电池驱动的应用中有很大的吸引力。

## 8.4 直接存储器访问

主存储器和 I/O 设备（比如磁盘）之间经常进行数据块传输。这一节将讨论一种控制这种传输的技术，不需要处理器频繁地进行程序控制干预。

第 3 章中主要讨论的是处理器和 I/O 设备之间的单字或单字节数据传输。将数据从 I/O 设备传输到存储器，首先要使用一条指令如：

```
Load   R2, DATAIN
```

将数据从 I/O 设备读出来装入处理器的寄存器中；然后，将读出的数据存放到存储单元中。将数据从存储器传输到 I/O 设备就进行一个相反的过程。传送输入或输出数据的指令只有在处理器确定 I/O 设备准备就绪时才能执行，处理器通过查询 I/O 设备的状态寄存器，或等待一个中断请求来确定设备是否就绪。但是这两种方式都会导致相当大的开销，因为每传送一个字大小
285 的数据必须执行好几条包含很多存储器访问的程序指令。当传输一个数据块时，需要使用指令来增加存储器地址和记录字数。中断的使用涉及操作系统例程，这些例程将会导致由于保存和

恢复处理器寄存器、程序计数器和其他状态信息而产生的额外开销。

另一种可行的方法是直接在主存和 I/O 设备（如磁盘）之间传输数据块。该方法提供了一个专门的控制部件来管理传输过程，而不需要处理器做连续干预。这种方法被称为直接存储器访问（direct memory access，DMA）。控制 DMA 传输的部件被称为 DMA 控制器（DMA controller）。它可能是 I/O 设备接口的一部分，或者也可能是一个被一些 I/O 设备共享的独立部件。访问主存时，DMA 控制器执行本来由处理器执行的功能。对每一个传输的字，它提供存储器地址，产生需要的所有控制信号。它为连续的字增加存储器地址并累计已传输的字数。

虽然 DMA 控制器不需要处理器的参与就可以传输数据，但它的操作必须由处理器执行的程序控制。这个程序通常是一个操作系统例程。为了初始化一个字块的传输，处理器必须向 DMA 控制器发送起始地址、块内字数和传输方向等信息。然后 DMA 控制器就开始执行所请求的操作。当整块数据传输完成后，它产生一个中断信号通知处理器。

图 8-12 显示了 DMA 控制器中的寄存器，处理器访问这些寄存器来初始化数据传输操作。其中两个寄存器用来存储起始地址和字数。第三个寄存器保存着状态和控制标志。R/W 位决定数据传输的方向，当它被程序指令置为 1 时，控制器执行读操作，即将数据从存储器传输到 I/O 设备。否则，控制器执行写操作。此外，处理器也会传输一些 I/O 设备可能需要的额外信息。例如，就磁盘来说，处理器会向磁盘控制器提供一些用以识别数据在磁盘上位置的信息（对于磁盘的详细信息，请参阅 8.10.1 节）。

当控制器传输完一块数据并准备好接收另一个命令时，将 Done 标志置 1。第 30 位是中断允许标志 IE。当这个标志被置为 1 时，控制器在传输完一块数据后将产生一个中断。最后，在请求中断后控制器将 IRQ 位置 1。

图 8-13 显示了如何在图 7-18 所示的计算机系统中使用 DMA 控制器。其中一个 DMA 控制器将高速的以太网连接到计算机的 I/O 总线上（图 7-18 中的 PCI 总线）。磁盘控制器控制两个磁盘，同时具有 DMA 功能，并提供两个 DMA 通道。它能够执行两个独立的 DMA 操作，就像每个磁盘都有自己的 DMA 控制器一样。它有两套寄存器来存放存储器地址、字数等信息，因此每个磁盘可以使用一套。

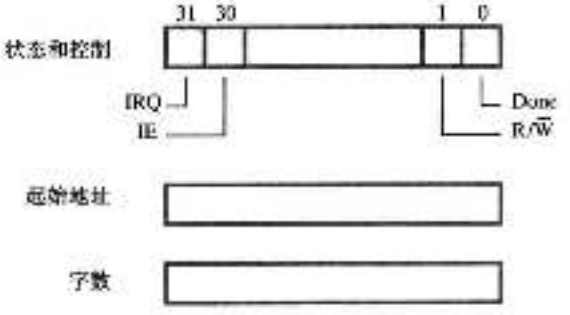



图 8-12　DMA 控制器中的典型寄存器

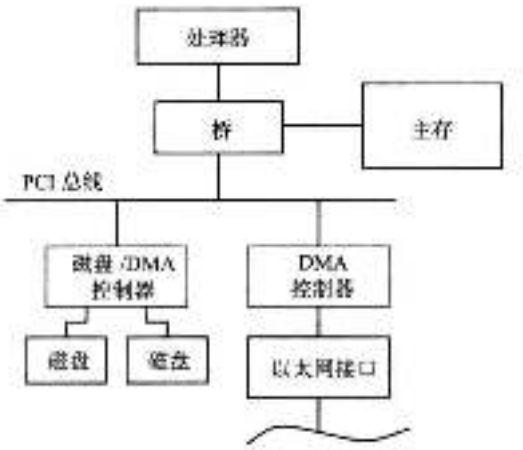



图 8-13　DMA 控制器在计算机系统中的使用

要启动一次 DMA 传输，将一块数据从主存传送到其中一个磁盘，操作系统例程需要将地址和字数信息写入到磁盘控制器的寄存器中。DMA 控制器独立执行指定的操作。当 DMA 传输完成后，通过设置 Done 位在 DMA 通道的状态和控制寄存器中记录该事件。同时，如果 IE

286 ~ 287

位被置位，控制器将发送一个中断请求到处理器并设置 IRQ 位。状态寄存器也可以用来记录其他信息，比如传输是正确的还是错误的。

## 8.5 存储器层次结构

我们已经说过，理想的存储器应该是高速度、大容量和低价格的。从 8.2 节的讨论中我们已经清楚地了解到用静态随机存储器芯片可以实现非常快的存储器。但是这些芯片不适合用来实现大容量的存储器，因为它们的基本单元大小及功耗都比动态随机存储器的单元要大。

虽然动态存储器部件可以用合理的成本实现数千兆字节的容量，但是可提供的容量与有庞大数据的大程序要求相比还是很小的。一个解决办法是使用辅助存储设备来提供所需的存储器空间，这种辅助存储设备主要是磁盘。磁盘可以用合理的成本得到，而且它们在计算机系统中被广泛地使用，但要比半导体存储器部件慢得多。总的来说，划算的海量存储可以由磁盘来提供。一个容量较大、速度相当快但价格合理的主存使用动态随机存储器技术来构建。价格更贵、速度更快的静态随机存储器技术用在更小的部件中，这些部件中速度极其重要，例如高速缓存。

所有这些不同类型的存储器部件都有效地在计算机系统中使用，计算机的整个存储器可以看作在图 8-14 中描述的分层结构。最快的访问是对保存在处理器寄存器中数据的访问，因此，如果把处理器寄存器看作存储器层次结构的一部分的话，那么就访问速度而言，处理器寄存器位于最顶端。当然，寄存器只提供了所需存储器的一个很小部分。

288

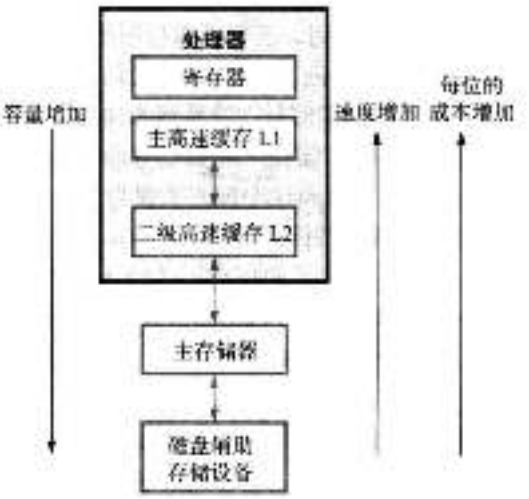



图 8-14 存储器层次结构

层次结构中的下一层是一个相对来说容量较小的存储器，它可以直接在处理器芯片内部实现。这个存储器被称为处理器高速缓存（processor cache），用来保存存储在外部更大存储器中的指令和数据的副本。高速缓存的概念在 1.2.2 节中介绍，在 8.6 节中有详细的分析。通常有两级或两级以上的高速缓存。主高速缓存总是放在处理器芯片内部，这个高速缓存很小，其访问时间是可以跟处理器寄存器相比的。主高速缓存被称为一级（L1）高速缓存。一个更大的、速度有点慢的辅助高速缓存放在主高速缓存和其余存储器之间，它被称为二级（L2）高速缓存，通常二级高速缓存也被放置在处理器芯片内部。

有些计算机除了一级和二级高速缓存外，还有一个容量更大的三级（L3）高速缓存。三级高速缓存也是用 SRAM 技术实现的，它跟处理器和一级、二级高速缓存可能在同一个芯片中，也可能不是。

层次结构中再往下一层是主存储器（main memory），这是用动态存储器部件实现的一个很大的存储器，通常组装成存储器模块的形式，比如 8.2.5 节中描述的 DIMM。主存比高速缓存大很多，但是也要慢得多。在一台处理器时钟为 2GHz 或更高的计算机中，主存的访问时间比一级高速缓存的访问时间长达 100 倍。

磁盘设备提供廉价的大容量存储，它们被广泛用作计算机系统中的辅助存储。与主存相比，它们是非常慢的。它们代表存储器层次结构中的最底层。

在程序执行期间，存储器访问速度是至关重要的。管理图 8-14 中分层存储器系统操作的关键是把即将要使用的指令和数据放到离处理器尽可能近的地方，这是使用我们接下来要讨论的高速缓存的主要目的。

## 8.6　高速缓存

高速缓存是一个很小的但速度很快的存储器，它被插入到处理器和主存之间。它的目标是使主存对处理器而言比实际上表现得更快一些。这种方法的有效性是以计算机程序的引用局部性（locality of reference）特征为基础的。对程序的分析显示它们大部分的执行时间花在例程中，其中有很多指令被重复执行。这些指令可能包括一个简单的循环、嵌套的循环或一些反复互相调用的过程。指令序列的具体模式并不重要，关键是在一段时间内程序局部区域的很多指令被反复执行。这种行为表现在两个方面：时间和空间上。时间方面意味着最近执行的指令可能很快又被执行到，而空间方面意味着与最近执行指令邻近的指令也可能很快被执行到。 289

从概念上说，高速缓存的操作非常简单。存储器控制电路利用引用局部性特征而设计。时间局部性指出只要信息项（指令或数据）是第一次需要，那么它就应该被放入高速缓存，因为可能很快又需要它。空间局部性指出不要每次只从主存中取一项放到高速缓存中，取邻近地址中的多个项是很有用的。术语高速缓存块（cache block）是指一定大小的一组邻近地址单元。另一个经常用来指明高速缓存块的术语是高速缓存行（cache line）。

考虑图 8-15 中的简单布局。当处理器发出一个读请求时，包含指定单元的一块存储器字的内容被传到高速缓存。随后，当程序引用这个块中的任何单元时，所需内容直接从高速缓存中读取。通常高速缓存在给定时刻能保存合理数量的块，但这个块数与主存中所有的块数相比还是要小得多。主存块与高速缓存块的对应关系由映射功能（mapping function）指定。当高速缓存已经满了，而只高速缓存外的一个存储器字（指令或数据）被引用时，高速缓存控制硬件必须决定移出哪一个块，为包含所引用字的新块腾出空间。做出这种决定的规则集合构成了高速缓存的替换算法（replacement algorithm）。




图 8-15　高速缓存的使用

### 1. 高速缓存命中

处理器不需要明确地知道高速缓存的存在，它简单地使用指向存储单元的地址发出读写请求。高速缓存控制电路判断所请求的字当前是否存在于高速缓存中。如果在，那么读或写操作直接对恰当的高速缓存单元执行，这时称为发生了读或写命中（read or write hit）。当一个读操作在高速缓存中命中时，就不会涉及主存。对于写操作，系统可以按两种方法之一来处理。第一种技术称为直接写（write-through）协议，高速缓存单元和主存单元都更新。第二种技术只更新高速缓存单元，并将包含该单元的块打上一个关联的标志位，这个标志位称为脏位（dirty bit）或修改位（modified bit）。这个字的主存单元将在以后更新，它的更新是在当包含这个标记字的块为新块腾出空间而被移出高速缓存时进行的。这个技术被称为写回（write-back 或 copy-back）协议。 290

直接写协议比写回协议简单，但是当一个给定的高速缓存字在高速缓存期间被多次更新时，直接写会造成对主存的不必要的写操作。写回协议也可能包含不必要的写操作，因为当这个块在高速缓存中时即使只有一个字被修改过，那么块中所有的字最终都会被写回。写回协议是最常用的，它利用了数据块能被高速传输到存储器芯片的优势。

2. 高速缓存失效

对一个不在高速缓存中的字进行读操作，会造成一次读失效（read miss）。它将使得包含所请求字的那个块从主存拷贝到高速缓存中。每个块装进高速缓存后，所请求的这个字传送给处理器。一种替代的方法是字一旦从主存中读出，就立刻送到处理器中。后一种方法称为直接装入（load-through）或早重启（early restart），它在一定程度上减少了处理器等待的时间，但却以更复杂的电路为代价。

在一台使用直接写协议的计算机中发生写失效（write miss）时，信息会直接写入主存。对于写回协议，包含所寻址的字的块首先被装入高速缓存，然后高速缓存中对应的字再被新信息覆盖。

回顾一下6.7节中描述的流水线处理器中的资源限制可能会导致指令的执行被暂停一个或几个周期。如果一条Load或Store指令请求访问存储器中的数据，而同时要提取随后的一条指令时，就会发生这种情况。当发生这种情况时，指令的提取将被延迟，直到数据访问操作完成为止。为了避免暂停流水线，许多处理器使用独立的指令和数据高速缓存，这使得这两种操作可以并行进行。

### 8.6.1 映射功能

有几种用于确定存储器块被放入高速缓存中哪个位置的方法，这里将使用一个具体的小例子来描述这些方法。考虑一个包含128个块的高速缓存，每个块有16个字，总容量是2048（2K）个字。假设主存用16位地址编址。主存有64K个字，我们把它看作4K个16个字的块。为简便起见，假设连续的地址对应连续的字。

1. 直接映射

决定主存块在高速缓存中位置的最简单方法是直接映射（direct-mapping）技术。在这项技术中，主存块$j$映射到高速缓存中$j$模128的块上。如图8-16中所示。这样，只要是第0、128、256、…个主存块要被装入高速缓存，它们都被存储在高速缓存的第0块，第1、129、257、…个主存块都被存储在高速缓存的第1块，依次类推。因为多个主存块被映射到一个指定的高速缓存块位置，所以即使高速缓存没有满，在该位置上也可能引起冲突。例如，一个程序的指令可能从第1块开始，接着可能在一个转移语句后跳到第129块。当这个程序执行时，这两个块都被传输到高速缓存的第1块。可以通过允许新块覆盖已存在的块来解决这种冲突。

291

使用直接映射技术，其替换算法的价值是不大的。一个块在高速缓存中的放置由其主存地址决定，主存地址可以分成三个字段，如图8-16所示。低4位从16个字的块中选择一个字。当新块进入高速缓存的时候，7位的高速缓存块字段决定这个块存入高速缓存中哪个位置。块的主存地址的高5位存储在与它在高速缓存中的位置相关联的5个标志（tag）位中。标志位用来确定映射到这个位置的32个主存块中哪一块当前正在高速缓存中。在执行过程中，由处理器产生的每个地址的7位高速缓存块字段指向高速缓存中特定的块位置。地址的高5位和与高速缓存位置相关联的标志位做比较。如果匹配，那么所需的字就在那个高速缓存块中；如果不匹配，则包含所需字的块必须首先从主存中读出并装入高速缓存中。直接映射技术很容易实现，但是不是很灵活。

2. 相联映射

图8-17给出了一种最灵活的映射方法，通过它可以把主存块放置到高速缓存中任何一个位置上。在这种情况下，当一个主存块在高速缓存中的时候，需要12个标志位去标记它。从处理器接收的地址中的标志位与高速缓存中每个块的标志位做比较，看所需要的块是否存在。

这被称为相联映射（associative-mapping）技术。它可以完全自由地选择在哪个高速缓存位置上放置主存块，从而可以更有效地利用高速缓存的空间。当一个新块需要放入高速缓存时，只有高速缓存满了时它才替换出一个已存在的块。在这种情况下，需要一个算法选择出哪个块将要被换出。这里有多种替换算法进行选择，我们将在 8.6.2 节中讨论。相联映射高速缓存的复杂性比直接映射高速缓存的要高，因为它需要检索所有 128 个标志的值，以判断给定的块是否在高速缓存中。为了避免长时间的延迟，标志必须被并行地检索。这种检索被称为相联检索（associative search）。

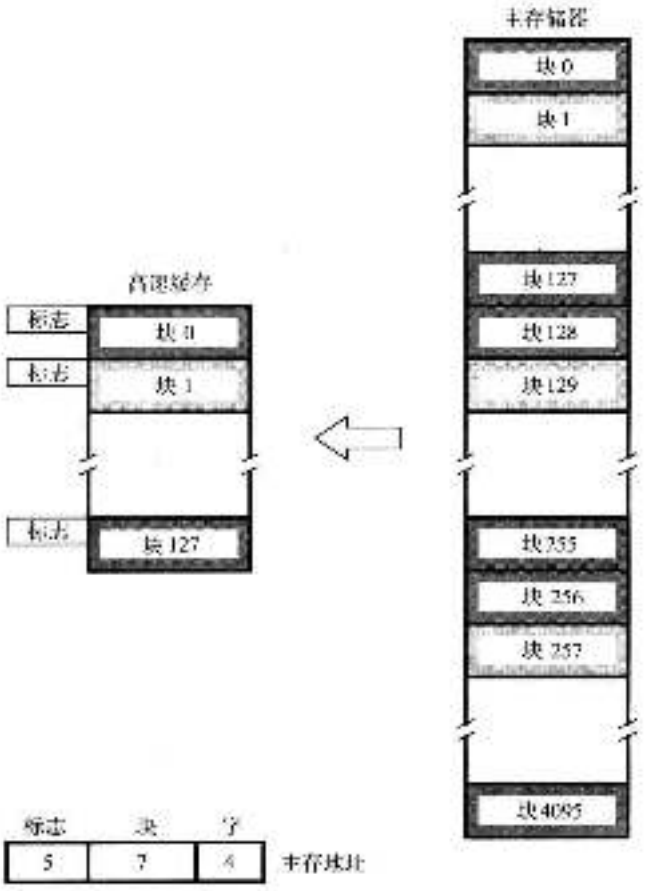



图 8-16　直接映射高速缓存

### 3. 组相联映射

另一种方法可以将直接映射技术和相联映射技术结合起来使用。高速缓存的块被分成组，这种映射允许主存块可以位于特定组中的任何一个块中。这样，放置块时可以有多个选择，从而使直接映射方法的冲突问题得到缓解。同时，通过减少相联检索的大小可以降低硬件成本。图 8-18 给出了一个组相联映射（set-associative-mapping）技术的例子，其中高速缓存中每个组有两个块。这样，第 0、64、128、…、4032 个主存块映射到高速缓存的第 0 组，它们可以占用这个组的两个位置中的任意一个。有 64 个组时，就意味着地址中的 6 位组字段决定高速缓存中的哪个组可能包含所需的块。然后地址中的标志字段与这个组中两个块的标志域相联比较，以检查所需要的块是否存在。这种两路相联检索实现起来很简单。

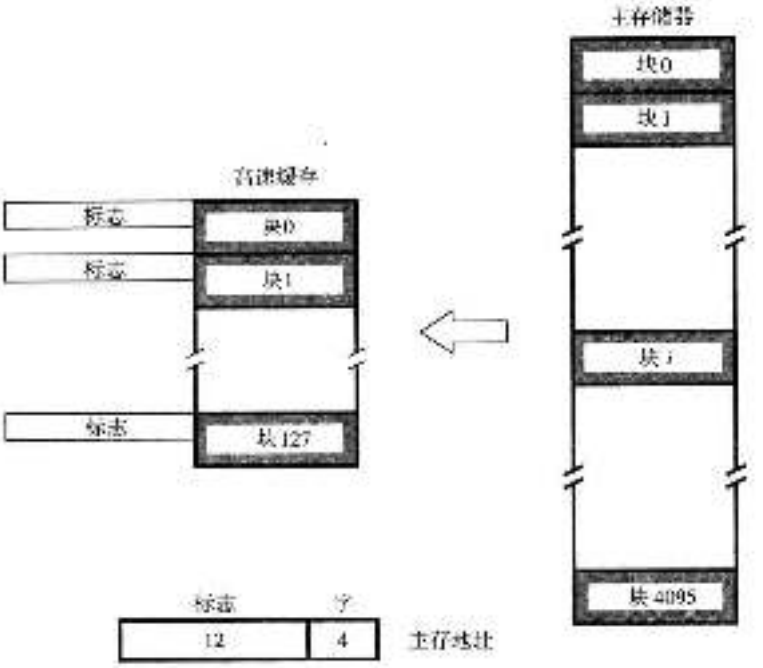



图 8-17 全相联映射高速缓存

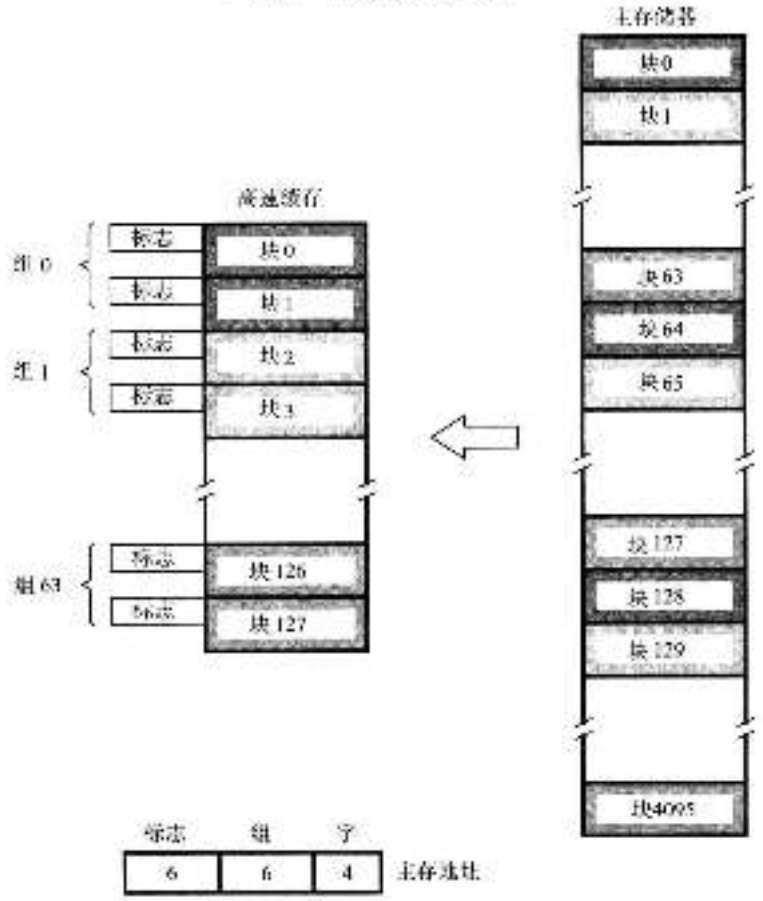



图 8-18 每组两个块的组相联映射高速缓存

每组的块数可以作为一个参数，根据特定计算机的需求进行选择。对于图 8-18 中的主存和高速缓存的容量来说，5 位的组字段可以满足每组 4 个块、4 位的组字段可以满足每组 8 个块，依次类推。每组 128 个块的极端情况不需要组字段，它对应全相联技术，有 12 个标志位。另一种每组只有一个块的极端情况就是直接映射方法。每组有 $k$ 个块的高速缓存被称为 $k$ 路组相联高速缓存。

4. 过时数据

当电源第一次开启时，高速缓存中不包含任何有效的数据。必须为每个高速缓存块提供一个控制位，通常被称为有效位（valid bit），来表示那个块中的数据是否是有效的。这个位不应该与前面提到的修改位或脏位混淆起来。当系统电源刚接通时，所有高速缓存块的有效位都被置成 0。当新的程序或数据从磁盘装入主存时，一些有效位被置为 0。使用 DMA 机制从磁盘传输到主存的数据通常绕过高速缓存，直接装入主存中。如果正在进行更新的主存块目前在高速缓存中，那么相应高速缓存块的有效位被置为 0。随着程序的执行，当一个主存块被装入一个给定的高速缓存块中时，该高速缓存块的有效位被置为 1。处理器只在高速缓存块的有效位等于 1 时才从该高速缓存块中提取数据。按这种方式使用有效位可以保证处理器不会从高速缓存中提取过时（stale）的数据。

在使用写回协议的系统中，需要采取一种类似的预防措施。在这种协议下，写入高速缓存的新数据不同时写入主存中。因此，主存中的数据并不总是反映出对高速缓存中的副本所进行的修改。重要的是要确保主存中的这种过时数据不会被传输到磁盘上。一种解决方法就是转储冲洗（flush）高速缓存，在执行传输操作之前迫使所有的脏块都写回到主存中。操作系统可以通过在初始化传输数据到磁盘的 DMA 操作之前发送一条命令给高速缓存来做到这一点。转储清除高速缓存对性能的影响并不大，因为这样的磁盘传输发生的频率不高。保证两个不同的实体（这里指处理器和 DMA 子系统）使用相同数据备份的需求被称为高速缓存一致性（cache-coherence）问题。

292 ~ 295

## 8.6.2　替换算法

在直接映射高速缓存中，每个块的位置是由它的地址事先决定的，因此替换策略是没有价值的。在相联和组相联高速缓存中存在一定的灵活性。当一个新块需要放入高速缓存而所有它可能占用的位置都已经满了时，高速缓存控制器必须决定那一个原有的块将被覆盖。这是一个很重要的问题，因为这个决定是系统性能的决定性因素。通常，我们的目标是将那些最近可能会被引用的块保留在高速缓存中。但是决定哪一个高速缓存块将被引用并不是一件容易的事。程序的引用局部性特征为我们一个合理的策略提供了线索。因为程序的执行通常在一个局部区域内保留相当长的时间，所以最近引用的块将会有很高的概率被再次引用。因此，如果有一个块要被覆盖，那么覆盖最长时间没有被访问的那个块是比较合理的。这个块被称为最近最少使用（least recently used，LRU）的块，这项技术被称为 LRU 替换算法（LRU replacement algorithm）。

要使用 LRU 算法，高速缓存控制器必须在进行计算的过程中跟踪所有块的引用情况。假设需要在每组四块的组相联高速缓存中跟踪 LRU 块，可以为每个块使用一个 2 位的计数器。命中时，被引用块的计数器被置成 0，原来比被引用块的计数器值低的计数器都增加 1，其他的保持不变。当发生高速缓存失效且组不满时，与从主存中装入的新块相关联的计数器被置成 0，所有其他计数器的值增 1。当发生高速缓存失效且组已经满了时，计数器值为 3 的块被移出，新块放到被移出块的位置上，并将它的计数器置成 0，其他三个块的计数器增 1。很容易

证明被占用块的计数器值总是不同的。

LRU 算法被广泛地使用。虽然它在很多访问模式下表现得很好，但是在某些情况中会导致很差的性能。例如，在顺序访问一个比较大的、无法全部装入高速缓存的数组元素时，它将产生令人失望的结果（参见 8.6.3 节和习题 8.11）。可以通过在决定哪个块被替换时引入少量随机性来提高 LRU 算法的性能。

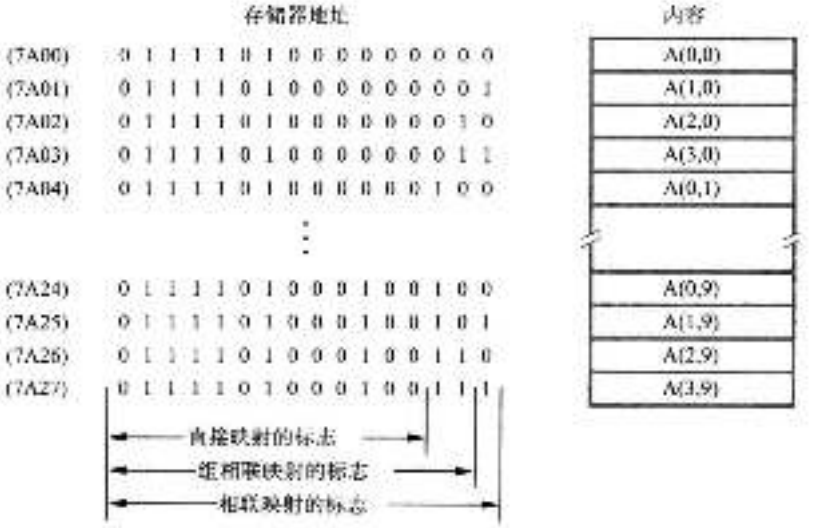
实践中也使用一些其他的替换算法。直觉上一个合理的规则应该是当装入新块时从一个已经满了的组中移出最旧的块。但是，因为这个算法没有考虑高速缓存块的最近访问情况，所以在选择最适合被移出的块时通常不如 LRU 算法有效。最简单的算法是随机选择一个块覆盖。
[296] 有趣的是，人们发现这个简单的算法在实践中非常有效。

### 8.6.3 映射技术的例子

现在来考虑一个详细的例子，来说明不同的高速缓存映射技术的效果。假设处理器有独立的指令高速缓存和数据高速缓存。为了使例子简单，假设数据高速缓存只有能容纳 8 块数据的空间，同时假设每个块只由一个 16 位字的数据组成，且存储器是按字编址的，它有 16 位地址（这些参数在实际计算机中并不真实，但是却可以让我们更清楚地描述映射技术）。最后，假设在高速缓存中替换块时使用 LRU 替换算法。

让我们来分析一下运行下面程序引起的数据高速缓存表项的变化：一个 $4 \times 10$ 的数字阵列，每个数字占一个字，阵列存储在主存单元 7A00 到 7A27 处（十六进制）。这个阵列 A 的元素按列的顺序存储，如图 8-19 所示。这个图还说明了不同高速缓存映射技术的标志是如何从存储器地址中得到的。注意这里不需要像图 8-16 到图 8-18 那样用一些位来标识块中的字，因为我们已经假设每个块中只有一个字。程序用 A 的第一行元素的平均值来标准化这一行元素。因此，我们需要计算这一行元素的平均值，然后用这个平均值去除每一个元素。所需要的任务可以表示为：

$$A(0, i) \leftarrow \frac{A(0, i)}{\left(\sum_{j=0}^{9} A(0, j)\right)/10} \qquad i = 0, 1, \cdots, 9$$




[297] 图 8-19 存储在主存中的一个阵列

图 8-20 给出了对应于这项任务的程序结构。我们使用变量 SUM 和 AVE 来分别保存总和

与平均值。这些变量，以及索引变量 $i$ 和 $j$，在计算过程中都保存在处理器的寄存器中。

```
SUM := 0
for j := 0 to 9 do
    SUM := SUM + A(0,j)
end
AVG := SUM/10
for i := 9 downto 0 do
    A(0,i) := A(0,i)/AVG
end
```

图 8-20　8.6.3 节中示例的任务

### 1. 直接映射高速缓存

在一个直接映射的数据高速缓存中，高速缓存内容的变化如图 8-21 所示。表中的各列表示图 8-20 中程序的两个循环各次执行完成后高速缓存中的内容。例如，第一个循环执行完两次后（$j=1$），高速缓存中保存元素 A (0, 0) 和 A (0, 1)。这些元素在块位置 0 和 4 中，由地址的最低 3 位决定。在下次循环中，元素 A (0, 0) 被 A (0, 2) 替换，因为 A (0, 2) 也被映射到相同的块位置。注意，所需要的元素只被映射到高速缓存的两个位置中，其他的六个位置无论在标准化任务开始之前的内容是什么，都保持不变。

第一个循环在执行第九次和第十次期间（$j=8, 9$），元素 A (0, 8) 和 A (0, 9) 被装入高速缓存。第二个循环颠倒了元素的处理顺序。这个循环的前两次（$i=9, 8$）在高速缓存中找到需要的数据。当 $i=7$ 时，元素 A (0, 9) 被 A (0, 7) 替换；当 $i=6$ 时，元素 A (0, 8) 被 A (0, 6) 替换，依次类推。因此在执行第二个循环时，八个元素都被替换了。所以在执行这个任务的过程中总共只有两次命中。

| | 每次循环后数据高速缓存中的内容： | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 块位置 | $i=1$ | $j=3$ | $j=5$ | $j=7$ | $j=9$ | $i=6$ | $i=4$ | $i=2$ | $i=0$ |
| 0 | A(0,0) | A(0,2) | A(0,4) | A(0,6) | A(0,8) | A(0,6) | A(0,4) | A(0,2) | A(0,0) |
| 1 | | | | | | | | | |
| 2 | | | | | | | | | |
| 3 | | | | | | | | | |
| 4 | A(0,1) | A(0,3) | A(0,5) | A(0,7) | A(0,9) | A(0,7) | A(0,5) | A(0,3) | A(0,1) |
| 5 | | | | | | | | | |
| 6 | | | | | | | | | |
| 7 | | | | | | | | | |

图 8-21　直接映射数据高速缓存的内容

读者应该记住要在高速缓存中为每个块保存标志，为保持图的简单，我们没有在图中显示它们。

### 2. 相联映射高速缓存

图 8-22 显示了相联映射高速缓存情况下高速缓存内容的变化。假设高速缓存最开始是空的，那么在第一个循环的前八次循环中，元素被放入连续的块位置中。在第 9 次循环（$j=8$）中，LRU 算法选择 A (0, 0) 被 A (0, 8) 覆盖。$j$ 循环的下一次也是最后一次循环中，元素 A (0, 1) 被 A (0, 9) 替换。现在，第二个循环的前八次循环（$i=9, 8, \cdots, 2$）需要的所有元素都可在高速缓存中找到。当 $i=1$ 时，需要的元素是 A (0, 1)。所以它替换最近最少使用的元素 A (0, 9)。在最后一次循环中，A (0, 0) 替换 A (0, 8)。

| | 每次循环后数据高速缓存中的内容： | | | | |
|---|---|---|---|---|---|
| 块位置 | $j=7$ | $j=8$ | $j=9$ | $i=1$ | $i=0$ |
| 0 | A(0,0) | A(0,8) | A(0,8) | A(0,8) | A(0,0) |
| 1 | A(0,1) | A(0,1) | A(0,9) | A(0,1) | A(0,1) |
| 2 | A(0,2) | A(0,2) | A(0,2) | A(0,2) | A(0,2) |
| 3 | A(0,3) | A(0,3) | A(0,3) | A(0,3) | A(0,3) |
| 4 | A(0,4) | A(0,4) | A(0,4) | A(0,4) | A(0,4) |
| 5 | A(0,5) | A(0,5) | A(0,5) | A(0,5) | A(0,5) |
| 6 | A(0,6) | A(0,6) | A(0,6) | A(0,6) | A(0,6) |
| 7 | A(0,7) | A(0,7) | A(0,7) | A(0,7) | A(0,7) |

图 8-22　相联映射数据高速缓存的内容

这种情况下，当第二个循环执行时，只有两个元素不能在高速缓存中找到。在直接映射的情况中，第二个循环执行期间有八个元素需要重新装入。显然，相联高速缓存受益于可以完全自由地把主存块映射到高速缓存中的任何位置。在这两种情况下，通过在程序的第二个循环中顺序处理元素可以更好地利用高速缓存。如果第二个循环处理元素的顺序与第一个循环一样的话将会怎样呢？考虑这个问题很有意思。使用直接映射或 LRU 算法，在第二个循环中，所有元素都会在使用前被覆盖（参见习题 8.10）。

### 3. 组相联映射高速缓存

在这个例子中，假设组相联数据高速缓存组织成两个组，每组能保存四个块。这样，地
298 址的最低有效位决定一个主存块映射到哪个组中，但该主存块的数据可以放在这个组中四个块
299 的任何一块中。地址的高 15 位组成标志。

图 8-23 描述了高速缓存内容的变化。因为所有需要的块都是偶数的地址，所以它们映射到第 0 组中。在这种情况下，在第二个循环执行期间，有六个元素被重新装入。

虽然这是一个简单的例子，但是它说明了一般情况下，相联映射性能最好，组相联映射次之，直接映射最差。但是相联映射实现代价太高，所以组相联映射是一种很好的折中的实用方法。

| | 每次循环后数据高速缓存中的内容： | | | | | |
|---|---|---|---|---|---|---|
| | $j=5$ | $j=7$ | $j=9$ | $i=4$ | $i=2$ | $i=0$ |
| 第0组 | A(0,0) | A(0,4) | A(0,8) | A(0,4) | A(0,4) | A(0,0) |
| | A(0,1) | A(0,5) | A(0,9) | A(0,5) | A(0,5) | A(0,1) |
| | A(0,2) | A(0,6) | A(0,6) | A(0,6) | A(0,2) | A(0,2) |
| | A(0,3) | A(0,7) | A(0,7) | A(0,7) | A(0,3) | A(0,3) |
| 第1组 | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

图 8-23 组相联映射数据高速缓存的内容

## 8.7 性能因素

计算机在商业上取得成功的两个关键因素是性能和成本，其目标是在给定的成本上获得最高的性能。衡量成效的一个通用标准是性价比（price/performance ratio）。性能依赖于机器指令能以多快的速度送入处理器，以及它们能以多快的速度执行。第 6 章说明了流水线是如何提高程序执行速度的。在这一章，我们把重点放在存储器子系统上。

在 8.5 节中描述的存储器层次结构源自于对更好性价比的探索，这个层次结构的主要目的是创建一个从处理器角度来看访问时间短且容量大的存储器。使用高速缓存时，当所引用存储单元的数据在高速缓存中时，处理器能够快速地访问指令和数据。因此，高速缓存提高性能的程度依赖于所请求的指令和数据能在高速缓存中找到的频繁程度。在这一节中，我们将量化地
300 分析这个问题。

### 8.7.1 命中率和失效开销

衡量存储器体系结构的具体实现效率的一个很好指标是在这个层次结构的各个层上访问信息的成功率。前面提到在高速缓存中一次成功的数据访问称为命中。命中次数比上访问总次数称为命中率（hit rate），而失效次数比上访问总次数称为失效率（miss rate）。

理想情况下，整个存储器层次结构对于处理器来说表现得就像一个单独的存储器部件那样，具有处理器芯片上的高速缓存的访问速度，还具有磁盘的容量。我们距离这个理想目标有多远，很大程度上取决于层次结构中不同层上的命中率。超过 0.9 的高命中率对于高性能计算机来说是必需的。

失效发生时需要采取的措施会对性能产生不利影响。由于把数据块从存储器层次结构中

的慢速部件放入高速部件需要额外的时间，从而会产生性能开销。在此期间，处理器被暂停以等待指令或数据。等待时间取决于高速缓存操作的具体细节，例如，它取决于是否使用了直接装入法。我们将失效发生时处理器所看到的总访问时间称为失效开销（miss penalty）。

考虑一个只有一级高速缓存的系统，在这种情况下，失效开销几乎全部由访问主存中数据块的时间组成。令 $h$ 表示命中率，$M$ 表示失效开销，$C$ 表示访问高速缓存中的信息需要的时间。这样，处理器的平均访问时间是

$$t_{ave} = hC + (1-h)M$$

下面的例子将说明这些参数的值是如何影响平均访问时间的。

**例 8.1**

考虑一台有以下参数的计算机，访问高速缓存和主存的时间分别为 $\tau$ 和 $10\tau$。在发生高速缓存失效时，一个 8 个字的块从主存传输到高速缓存中。传输该块的第一个字需要花费 $10\tau$ 的时间，剩下的 7 个字以每 $\tau$ 秒传输一个字的速度传输。失效开销还包括初始访问高速缓存失效时一个 $\tau$ 的延迟时间，以及当数据块装入高速缓存后再传输其中的一个字给处理器所需要的一个 $\tau$ 的延迟时间（假设不采用直接装入法）。这样，该计算机的失效开销可由下面的式子给出：

$$M = \tau + 10\tau + 7\tau + \tau = 19\tau$$

假设在一个具有代表性的程序中 30% 的指令需要执行读写操作，这意味着每执行 100 条指令会有 130 次存储器访问。假设高速缓存命中率对于指令是 0.95，对于数据是 0.9，再进一步假设读写访问的失效开销是相等的，那么使用高速缓存所带来的存储器性能的提高可以按下面的公式粗略估算： [301]

$$\frac{\text{无高速缓存时的时间}}{\text{有高速缓存时的时间}} = \frac{130 \times 10\tau}{100(0.95\tau + 0.05 \times 19\tau) + 30(0.9\tau + 0.1 \times 19\tau)} = 4.7$$

这个结果表明高速缓存使得存储器表现出来的速度比它实际的速度快差不多五倍。性能提高的倍数随着高速缓存相对于主存的速度的提高而提高。例如，如果主存的访问时间是 $20\tau$，则性能提高的倍数变为 7.3。

高命中率对于高速缓存有效减少存储器访问时间来说是非常重要的。命中率取决于高速缓存的容量，设计以及所执行程序的指令和数据访问模式。考虑这个例子中的高速缓存与命中率为 100% 的理想高速缓存的相比效果是很有益的。当高速缓存的命中率为 100% 时，所有的存储器引用都只耗费 1 个 $\tau$ 的时间，这样，由于高速缓存失效导致存储器访问时间增加的估算由下式给出：

$$\frac{\text{实际高速缓存时的时间}}{\text{理想高速缓存时的时间}} = \frac{100(0.95\tau + 0.05 \times 19\tau) + 30(0.9\tau + 0.1 \times 19\tau)}{130\tau} = 2.1$$

换言之，100% 的高速缓存命中率可以使得存储器表现出来的速度比使用实际的命中率时快两倍。

如何才能提高命中率呢？一个可能是使用更大的高速缓存，但是这必定要增加成本。另一个可能是在保持高速缓存总容量不变的前提下增加高速缓存块的大小，以发挥空间局部性的优势。如果在一个较大的块中所有的项都是计算中所需要的，那么最好是在一次失效中将所有这些项都装入高速缓存，而不是在多次失效后分别装入，每次只装入一个较小的块。在块传输期间可获得的高数据率是这个优势的主要原因。在达到一定的大小之前，增大块的大小是有效的，超过这个大小，命中率的提高会被抵消，因为有些项还没有被访问，该块就已经被换出了。并且，较大的块需要较长的时间传输，因而增加了失效开销。由于计算机的性能受命中率

增加的正面影响，还受失效开销增加的负面影响，因此块的大小既不能太小也不能太大。在实际中，最常用的块大小范围是16字节到128字节。

最后，我们注意到如果在新块装入高速缓存时使用直接装入法，那么失效开销可以降低。这样，所需要的字一旦被装入高速缓存处理器就可以重新开始执行，而不是必须等待整个块传输完毕。

### 8.7.2 处理器芯片上的高速缓存

当信息在不同的芯片间传输时，芯片上的驱动器和接收器会产生不可忽略的延迟。所以
[302] 最好在处理器芯片上实现高速缓存。大多数处理器芯片包括至少一个一级（L1）的高速缓存。通常使用两个独立的一级（L1）高速缓存，一个用于指令，另一个用于数据。

在高性能处理器中，通常使用两级高速缓存，独立的一级（L1）指令高速缓存和数据高速缓存，还有一个更大的二级（L2）高速缓存。这些高速缓存通常在处理器芯片上实现。在这种情况中，一级高速缓存的速度必须很快，因为它们决定了处理器所能看到的存储器访问时间。二级高速缓存可以慢一些，但是它应该比一级高速缓存大得多，以保证高命中率。它的速度不是那么关键，因为它只影响一级高速缓存的失效开销。一台典型的计算机可能有容量为数十K字节的一级高速缓存和数百K字节甚至数兆字节的二级高速缓存。

包含一个二级高速缓存能进一步减小主存速度对计算机性能的影响。可以观察到，二级高速缓存的平均访问时间是任何一个一级高速缓存的失效开销，我们以此来评估二级高速缓存的效果。为简单起见，我们将假设指令和数据的命中率是相等的。因此，在这种系统中，处理器的平均访问时间是：

$$t_{ave} = h_1C_1 + (1-h_1)(h_2C_2 + (1-h_2)M)$$

其中，

$h_1$是一级高速缓存的命中率；

$h_2$是二级高速缓存的命中率；

$C_1$是一级高速缓存的信息访问时间；

$C_2$是将信息从二级高速缓存传输到一个一级高速缓存的失效开销；

$M$是将信息从主存传输到二级高速缓存的失效开销。

在处理器对存储器的所有引用中，二级高速缓存的失效次数由$(1-h_1)(1-h_2)$给出。如果$h_1$和$h_2$都是在高于90%这个范围内，那么二级高速缓存的失效次数将小于所有存储器访问次数的1%。这使得$M$的值以及主存的速度都不那么重要了。对这个问题的定量分析参见习题8.14。

### 8.7.3 其他改进

除了刚才讨论的主要设计问题，还存在其他几种改进性能的可能性。在这一节我们将讨论其中三种。

1. 写缓冲区

当使用直接写协议时，每个写操作导致一个新数据被写入主存。如果处理器必须等待存储器功能完成，像我们之前假设的那样，那么所有的写请求会导致处理器变慢。但是处理器通
[303] 常并不需要立即访问写操作的结果，所以它不必等待写请求的完成。为了提高性能，可以引入一个写缓冲区（write buffer）来临时保存写请求。处理器把每个写请求放到这个缓冲区中，然后继续执行后面的指令。当存储器没有可响应的读请求时，就把保存在写缓冲区中的写请求发给主存。快速处理读请求是很重要的，因为在接收到从存储器中读取的数据前，处理器通常不

能继续执行。因此，这些请求的优先级比写请求高。

写缓冲区可以保存一定数量的写请求。因此，后面的读请求引用的数据可能仍在写缓冲区中。为了保证正确的操作，从存储器中读取的数据的地址总是要与写缓冲区中数据的地址做比较，如果匹配，就使用写缓冲区中的数据。

如果使用写回协议，会发生类似的情况。这时，处理器发出写命令对高速缓存中的字进行写。当读失效导致新的数据块被放入高速缓存时，可能会替换出一个现有的包含一些脏数据的块。脏块必须被写到主存中。如果先执行所需的写回操作，那么处理器在把新块读入高速缓存前，必须等待这个写回操作完成。更精明的做法是先读入新块，这就需要把即将从高速缓存中换出的脏块临时存储在写缓冲区中，并且在读取新块时让一直保存在那里。然后，再把写缓冲区中的内容写进主存。因而，写缓冲区对于写回协议也是有效的。

2. 预取

在前面讨论的高速缓存机制中，我们假设新数据在第一次需要时被装入高速缓存。读失效后，处理器必须暂停，直到新数据到达，这样就引发了失效开销。

为了避免处理器暂停，可以在需要数据之前把它们预取到高速缓存中。实现这一点的最简单方法是使用软件。处理器的指令集中可能提供一条特殊的预取（prefetch）指令。执行这条指令将使得被寻址的数据被装入高速缓存，就像读失效时的情况一样。在程序中插入预取指令，使得数据在程序需要之前就已经被装入了高速缓存中。然后，处理器将不必像读失效时那样等待引用的数据。最好在处理器执行不会引发读失效的指令时进行预取，这样主存访问就能与处理器运算重叠起来。

预取指令可以由程序员或者编译器插入到程序中。对很多应用程序来说，编译器能很成功地插入这些指令。软件预取肯定有一定的开销，因为包含预取指令会增加程序的长度。此外，一些预取操作可能会把那些后续指令不用的数据装入高速缓存。如果其他数据引发的读失效把已预取的数据从高速缓存中换出，这种情况就会发生。但是，软件预取对性能的总体效果还是有利的，并且许多处理器都有支持这个特性的机器指令。关于软件预取的全面讨论见参考文献 [1]。

384

预取也能用硬件实现，这需要使用试图发现存储器引用模式并根据这个模式预取数据的电路。为了这个目标已经提出了很多方案，在参考文献 [2] 和参考文献 [3] 中对这些方案进行了描述。

3. 无锁定高速缓存

如果软件预取对指令的正常执行产生很大干扰的话，就不会产生好的效果。如果预取操作在预取完成前阻止其他访问高速缓存的操作就会出现这样的情况。在响应失效时，我们称高速缓存被锁定了。这个问题可以通过修改高速缓存的基本结构来解决，允许处理器在高速缓存响应失效时仍能访问它。这种情况下，可能会有多个未响应的失效，硬件必须适应这样的情况发生。

能支持多个未响应的失效的高速缓存称为无锁定（lockup-free）高速缓存。这样的高速缓存必须引入用于跟踪所有未响应的失效的电路。这可以通过把与失效相关的信息保存在特定的寄存器中来实现。无锁定高速缓存在 20 世纪 80 年代早期最先用于由 Control Data 公司 [4] 生产的 Cyber 系列计算机上。

我们已经以软件预取为动机描述了在读失效时不锁定高速缓存的必要性。一个更重要的原因是在流水线处理器中，多条指令的执行是重叠的，一条指令引起的读失效会暂停其他指令的执行。无锁定高速缓存减少了这种停顿的可能性。

## 8.8 虚拟存储器

在大多数现代计算机系统中，物理主存没有处理器的地址空间大。例如，一个产生 32 位地址的处理器有 4G 字节的可寻址空间，而具有 32 位处理器的典型计算机中主存容量的范围可能从 1G 到 4G 字节。如果一个程序还没有完全装入主存，它当前未执行的部分存放在辅助存储设备上，通常是磁盘。因为程序执行中需要这些部分，所以它们必须预先装入主存，可能会替换已经在主存中存在的另外部分。这些操作由操作系统使用一种称为虚拟存储器（virtual memory）的方案自动完成。应用程序员不需要考虑可用主存带来的限制，他们使用处理器的整个地址空间来准备程序。

在虚拟存储器系统中，程序，其实也就是处理器，引用了一个与可用物理主存空间无关的地址空间中的指令和数据。这个由处理器生成的指令或数据的二进制地址称为虚拟地址（virtual address）或逻辑地址（logical address）。这个地址由硬件和软件协作转换成物理地址。如果一个虚拟地址引用了当前位于物理主存中的程序空间或数据空间的一部分，那么主存中对应位置的内容可以立即被访问。否则，引用地址的内容必须在被使用之前被放入主存中恰当的
305 位置上。

图 8-24 给出了实现虚拟存储器的典型组织结构。一个称为存储器管理部件（Memory Management Unit，MMU）的特殊硬件跟踪哪部分虚拟地址空间在物理主存中。当需要的数据或指令在主存中时，MMU 把虚拟地址转换成相应的物理地址。然后，所请求的存储器访问按通常的方式继续进行。如果数据不在主存中，那么 MMU 通知操作系统把数据从磁盘传送到主存，这种传送使用 8.4 节中讨论的 DMA 方式进行。

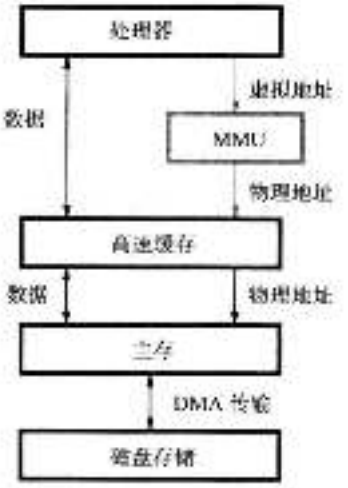


图 8-24 虚拟存储器组织结构

### 地址转换

把虚拟地址转换成物理地址的一种简单方法是假设所有的程序和数据都是由称为页（page）的固定长度单元组成的，每个页包含主存中连续单元组成的一个字块。页的长度范围通常是从 2K 到 16K 字节。当 MMU 确定需要进行一次传送时，页构成了在主存和磁盘之间传送信息的基本单位。页不能太小，
306 因为磁盘的访问时间（几毫秒）比主存的访问时间长得多，这是由于磁盘需要花费相当长的时间对数据进行定位，一旦定位，数据就能以每秒几兆字节的速度传输。另一方面，如果页太大，那么页中很大一部分数据可能没有被用到，这些不需要的数据会占用主存的有用空间。

这个讨论与 8.6 节中在高速缓存中引入的概念类似。高速缓存在处理器与主存之间的速度沟壑上搭起一座桥梁，它使用硬件实现，而虚拟存储器机制是在主存和辅助存储设备之间的容量和速度沟壑上架起一座桥梁，它通常部分地使用软件技术实现。从概念上看，高速缓存技术和虚拟存储器技术非常相似，它们的区别主要在实现细节上。

基于固定长度页的虚拟存储器地址转换方法如图 8-25 所示。处理器产生的每个虚拟地址，不管是用于取指令还是存取操作数，都被解释成虚拟页号（virtual page number）（高地址位）后边跟一个用来在一个页内指定一个特定字节（或字）位置的偏移量（offset）（低地址位）。每个页在主存中的位置信息保存在页表（page table）中。这个信息中包括页所在的主存地址和当前的状态。主存中能保存一个页的区域称为页帧（page frame）。页表的起始地址保存在页表基址寄存器（page table base register）中。把虚拟页号与这个寄存器的内容相加就能得到这个

页在页表中相应表项的地址。如果该页当前位于主存中，那么这个页表单元的内容指出这个页的起始地址。

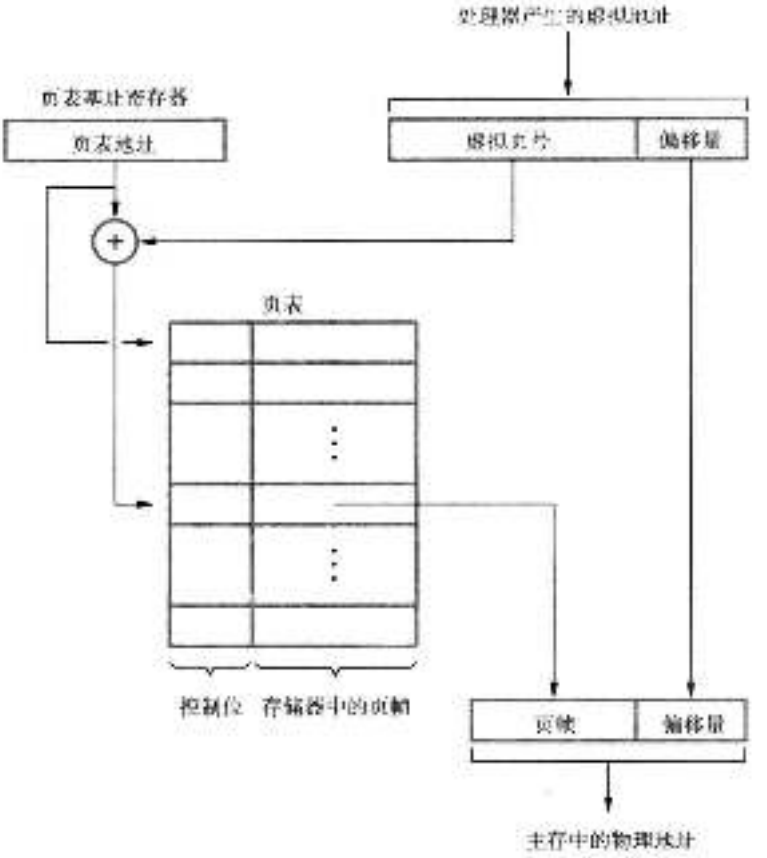



图 8-25　虚拟存储器地址转换

页表中的每个表项还包含一些控制位，用来描述该页在主存中时的状态。其中一个位表示页的有效性，即这个页是否确实在主存中。它允许操作系统将一个页标志为无效，而不用实际移除这个页。另一个位表示页在主存期间是否被修改过。与高速缓存一样，这个信息用于决定在为其他页腾出空间而从主存中移出这个页之前是否需要把它的内容写回到磁盘中。其他的控制位表示对访问这个页所施加的不同约束。例如，一个程序可能有完全的读写权限，也可能被限制为只能进行读访问。

**1. 转换监视缓冲区**

MMU 在每一个读写访问中都使用页表信息，所以从理想角度考虑，页表应该放在 MMU 内部。不幸的是，页表可能很大。由于 MMU 通常实现为处理器芯片的一部分，所以不可能把整个表放到 MMU 内。不过，可以将页表的一小部分拷贝到 MMU 内，而整个表保存在主存中。保存在 MMU 内部的这部分包括那些最近访问的页所对应的表项。它们被保存到一个通常称为转换监视缓冲区（Translation Lookaside Buffer，TLB）的小表中。TLB 对主存中的页表来说相当于一个高速缓存，TLB 中的每个表项包含页表相应表项中信息的副本。此外，它还包含页的虚拟地址，为一个特定的页搜索 TLB 时需要使用该虚拟地址。图 8-26 显示了 TLB 的一种可能的组织结构，它使用了相联映射技术。商业产品中也有使用组相联映射技术的 TLB。

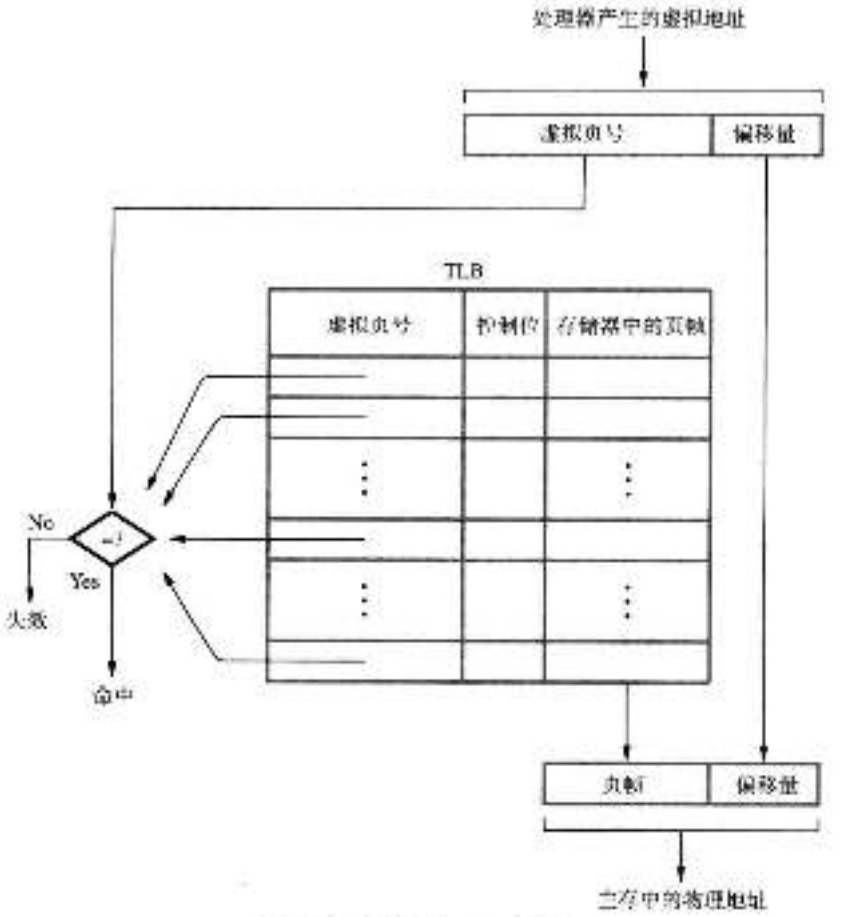



图 8-26 相联映射 TLB 的使用

接下来是地址转换处理。给定一个虚拟地址，MMU 在 TLB 中搜索引用的页。如果这个页的页表表项在 TLB 中，则可以立即获得物理地址。如果 TLB 失效，那么就从主存中的页表获取所需要的表项，并同时更新 TLB。

必须保证 TLB 中的内容与主存页表的内容始终是相同的。当操作系统更改页表中的内容时，它必须同时把 TLB 中对应的表项置成无效。TLB 中有一个控制位用于这个目的。当一个表项被置为无效时，TLB 将从主存中的页表获取新的信息。这是 MMU 对访问失效进行响应的其中一个步骤。

2. 页故障

当一个程序发出访问一个不在主存中的页的请求时，我们说发生一次页故障（page fault）。必须把整个页从磁盘放入主存之后才能对一个页进行访问。当 MMU 检测到一个页故障时，它通过产生一个异常（中断）来请求操作系统进行处理。这时，产生页故障的程序的执行被中断，控制权转移到操作系统。操作系统把请求的页从磁盘拷贝到主存中。因为这个过程需要很长的延迟，所以操作系统可能开始执行另一个页在主存中的程序。当页传输完成后，被中断的程序恢复执行。

307 ~ 309

当 MMU 产生一个中断来表明页故障时，请求存储器访问的那条指令可能已经被执行了一部分。必须保证被中断的程序在恢复执行后能正确地继续执行。这就有两种选择，被中断的

指令要么从中断点继续执行，要么重新开始。一个具体处理器的设计决定了使用的是两种方法中的哪一种。

当主存已满的时候，如果要从磁盘读出一个新的页，那么必须替换一个已经存在的页。选择哪个页被换出的问题与在高速缓存中一样重要，程序把大多数时间花费在一些局部区域的规律在这里也同样适用。因为主存要比高速缓存大得多，所以应该能把程序中相对更多的部分保存在主存中，这就减少了向或从磁盘中传输数据的频率。与 LRU 替换算法类似的概念也可以用于页替换中，页表表项中的控制位可以用来记录使用历史。有种基于一个控制位的简单方案是，当相应的页被引用（访问）时，这个控制位被置成 1。操作系统会定期地清除所有页表表项中的这个位，这样就提供了一个简单的方法来判断那些页最近没有被使用过。

一个修改过的页在移出主存之前需要写回到磁盘中。注意在高速缓存体系结构中非常有用的直接写协议不适合虚拟存储器，这一点很重要。磁盘的访问时间太长，经常访问它来写入少量的数据是没有意义的。

查找 TLB 的表项会引入一些延迟，这减慢了 MMU 的操作。这里我们再利用引用的局部性特征，多个连续的 TLB 地址转换很可能只涉及同一个程序页上的地址。这种情况在取指令时尤为可能。这样，我们可以将最近使用的 TLB 表项保存在一些可以快速访问的特殊寄存器中，从而减少地址转换时间。

## 8.9　存储器管理需求

在对虚拟存储器概念的讨论中，我们隐含地假设只有一个大程序在执行。如果不能把这个程序全部装入可用的物理存储器中，那么它的一部分（一些页）在执行时将从磁盘移到主存中。虽然我们间接地提到了管理程序段移动所需要的软件程序，但是并没有指出它们的细节内容。

存储器管理例程是计算机操作系统的一部分。将操作系统例程集中到一个虚拟地址空间中会很方便，这个空间称为*系统空间*（system space），它与用户应用程序所在的虚拟空间独立，后者称为*用户空间*（user space）。实际上，可能存在多个用户空间，每个用户一个，这可以通过为每个用户程序提供一个单独的页表来实现。MMU 使用页表基址寄存器来判断在转换过程中使用的页表的地址，因此，通过修改这个寄存器的内容，操作系统就可以从一个空间转 310
换到另一个空间。于是，物理主存由系统空间和多个用户空间的活动页共同分享，但是，在任何给定的时刻，只有属于其中一个空间的页能被访问。

在任何允许独立的用户程序在主存中共存的计算机系统中，必须解决保护（protection）的问题。任何程序都不能破坏主存中其他程序的指令和数据。保护可以通过几种方法来提供。让我们首先来考虑最基本的保护形式。大部分处理器可以在管理模式（supervisor mode）或者用户模式（user mode）下工作。当执行操作系统例程时，处理器通常处于管理模式，而执行用户程序时，处理器处于用户模式。在用户模式下，有一些机器指令不能被执行，这些指令是特权指令（privileged instruction），它们包括修改页表基址寄存器的指令，这种指令只能在处理器处于管理模式时才能执行。因为用户程序是在用户模式下执行的，所以它就不能访问其他用户的页表或者系统空间的页表。

有时候一个应用程序要求访问属于另一个程序的某些页，操作系统通过使这些页在两个空间中都可见来实现这一点，因而这些共享的页在两个不同的页表中都有表项。每个页表表项中的控制位都可以用来控制相应程序的访问权限。例如，对于一个给定的页，一个程序可能被允许读和写，而另一个程序可能只被允许读访问。

## 8.10 辅助存储器

前面章节讨论的半导体存储器并不能提供计算机需要的全部存储能力，其主要限制是每位存储信息的成本。大多数计算机系统的大量存储需求是使用更经济的磁盘和光盘的形式实现的，它们通常被称为辅助存储设备。

### 8.10.1 磁盘

在磁盘系统中存储介质是由安装在一个轴上的一个或多个磁盘盘片组成。在每个盘片上覆盖着一层很薄的磁性薄膜，通常两面都有。该装配放在一个能使它以恒定的速度旋转的驱动器上。磁性表面在读/写磁头附近移动，如图 8-27a 所示。数据存储在同心磁道上，读写磁头沿径向方向移动以访问不同的磁道。

每个读写磁头包括一个磁轭和一个磁性线圈，如图 8-27b 所示。通过向磁性线圈施加适当极性的电流脉冲可以将数字信息存储到磁性薄膜上，这导致磁头正下方的薄膜区域的磁化方向与施加的磁场同向。这个磁头还可以用于读取存储的信息，此时，磁性薄膜相对磁轭的运动导致磁头附近的磁场发生变化，这会在磁性线圈中感应出电压，这时这个线圈用作读出线圈。控制电路检测该电压的极性，以判断薄膜的磁化状态。在读操作时，只有磁头下的磁场发生变化才能被感应到。因此，如果二进制状态 0 和 1 用两个相反的磁化状态表示，那么只有在位流中 0 变成 1 或 1 变成 0 时，磁头才能感应出电压，一长串的 0 或 1 只有在这个串的开头和结尾才能产生感应电压。因此，为了判断所存储的连续的 0 或 1 的个数，必须用一个时钟提供同步信息。

311

在早期的设计中，时钟被存储在一个单独的磁道的磁性中。这个磁道的磁性在每个位周期中都必须改变。以这个时钟信号为参照，存储在其他磁道中的数据就能被正确读出了。现代的方法是把时钟信息与数据结合起来。现在已经开发出几种不同的技术用于这种编码。一种简单的方案在图 8-27c 中给出，它被称为相位编码（phase encoding）或曼彻斯特编码（Manchester encoding）。在这种方案中，每个数据位都有磁性变化发生，如图 8-27 所示。磁性变化可在每个位周期的中点发生，以此来提供时钟信息。曼彻斯特编码的缺点是它的位存储密度较低，表示一位需要的空间必须足够大，以容纳磁性的两个变化。我们用曼彻斯特编码的例子来阐述如何实现一个自同步时钟（self-clocking）方案，因为它比较好理解。此外，人们还开发了一些更紧凑的编码，它们更有效，并能提供更高的存储密度，但同时也需要更复杂的控制电路。对这些编码的讨论超出了本书的范围。

读写磁头必须与转动的盘面保持一个很近的距离，以此来获得较高的位密度和可靠的读写操作。当磁盘以一个稳定的速度旋转时，在盘面和磁头之间会产生一个空气压力，迫使磁头远离盘面。这个力可以由一个弹簧装置抵消，这个装置把磁头压向盘面。在磁头和它的支撑臂之间有灵活的弹簧连接，这允许磁头在离开盘面所需距离的位置上悬置，可以免受磁盘表平面微小起伏的影响。

在多数现代磁盘部件中，盘体和读写磁头被放置在一个密封的、对空气进行过滤的外壳中，这种方法被称为温切斯特技术（Winchester technology）。在这样的部件中，读写磁头可以在距离磁化轨迹表面更近的位置上工作，因为里面不存在灰尘微粒，而灰尘微粒是未密封组装中的难题。磁头离磁道表面越近，数据就能以越高的密度沿着磁道存放，磁道之间的距离也就越近。因此，在给定物理尺寸的条件下，温切斯特磁盘的容量比未密封部件更大。温切斯特技术的另一个优势是在密封部件中，存储介质没有暴露在污染环境中，因而数据有更好的完整性。

磁盘系统中的读写磁头是可移动的。每个盘面有一个磁头，所有的磁头都安装在一个梳状支撑臂上，这个支撑臂可以横跨磁盘堆径向移动，从而实现对单个磁道的访问，如图 8-27a

所示。要在一个给定的磁道上读或写数据，读写磁头必须首先定位到该磁道上。

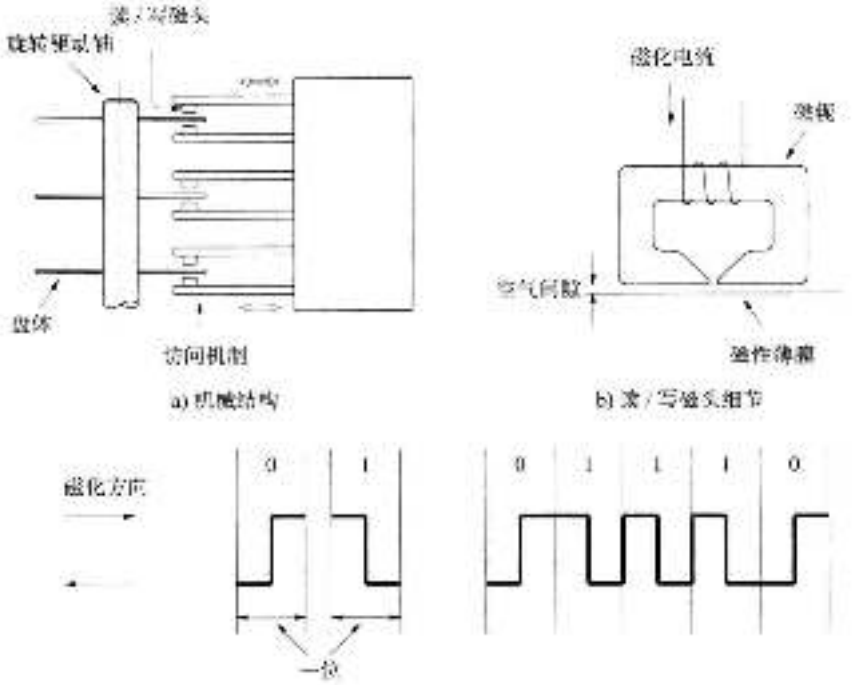



图 8-27　磁盘原理

磁盘系统包括三个关键部分。第一个部分是装配的磁盘盘片，通常称为磁盘（disk）。第二个部分是电机装置，用来旋转磁盘和移动读写磁头，这被称为磁盘驱动器（disk drive）。第三个部分是磁盘控制器（disk controller），它是控制系统运作的控制电路。磁盘控制器可以实现成一个单独的模块，也可以放在包含整个磁盘系统的外壳中。我们已经注意到通常把磁盘驱动器与磁盘合起来称为磁盘，在后面的章节中如果没有歧义，我们也使用这种叫法。

312
,
313

**1. 磁盘数据的组织结构和访问**

磁盘上数据的组织结构如图 8-28 所示，每个盘面都被分成同心的磁道（track），每个磁道又被分成扇区（sector）。磁盘堆所有盘面上相同磁道的集合形成逻辑上的柱面（cylinder），不需移动读写磁头就可以访问一个柱面中的所有磁道。数据通过指定盘面号、磁道号和扇区号来访问，读写操作总是在扇区边界处开始。

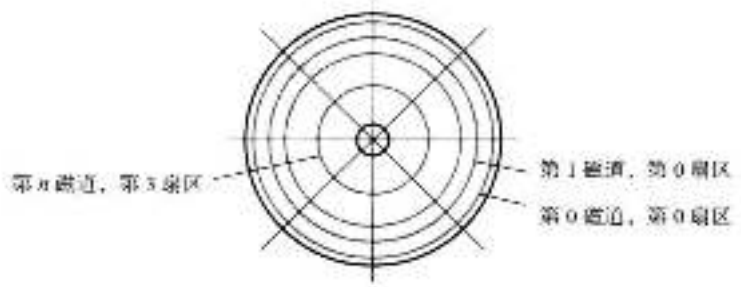



图 8-28　磁盘一个表面的结构

数据位顺序存放在每个磁道中。每个扇区可能包含512或者更多字节的数据。数据的前面有一个扇区头（sector header），它包含标志（寻址）信息，这些信息用来在选定的磁道上找到所需要的扇区。在数据后面有一些组成纠错码（error-correcting code，ECC）的附加位。ECC位用来检测和纠正在读写这些数据字节时可能发生的错误。扇区间存在一个小的扇区间隙（inter-sector gap），可使得磁盘控制电路很容易地区别两个连续的扇区。

一个未格式化磁盘的磁道上没有任何信息。格式化过程将写入一些标记信息，用来把磁盘分成磁道和扇区。在这个过程中，磁盘控制器可能会发现一些有缺陷的扇区甚至整个磁道，磁盘控制器保存这些缺陷的记录，并在使用时排除这些扇区。格式化信息包括扇区头、ECC位和扇区间隙。除掉格式化信息的开销后，格式化磁盘的容量是磁盘存储容量的很好的指标。格式化后，磁盘被分成逻辑分区。

图8-28说明每个磁道有相同数目的扇区，这意味着所有的磁道有相同的存储容量。在这种情况下，所存储的信息在内层磁道上比外层磁道放得更密。我们也可以通过在周长更长的外层磁道上放置更多的扇区来增加存储密度，这将需要使用更加复杂的访问电路。

314

2. 访问时间

从磁盘接收地址到实际的数据传输开始之间的时间延迟包含两个部分。第一个部分称为寻道时间（seek time），是把读写磁头移动到恰当磁道所需的时间，它依赖于开始时磁头到该地址所指定磁道的相对位置。寻道时间的平均值在5～8毫秒的范围内。第二个部分是旋转延迟（rotational delay），也称为等待时间（latency time），它是将读写磁头定位到正确的磁道后再到达所寻址的扇区所花费的时间。平均起来，这是磁盘转动半周所花的时间。这两个延迟的总和被称为访问时间（access time）。如果在一次操作中只有少数扇区的数据被访问，那么访问时间至少比传输数据所花费的时间高一个数量级。

3. 数据缓冲区/高速缓存

一个磁盘驱动器可以使用一些标准的互连方案，例如SCSI或SATA，连接到计算机系统的其他部分上。这些互连硬件传输数据的速率通常要比从磁盘的磁道读数据的速率快得多，处理传输速率差异的一个有效方法是在磁盘部件中引入一个数据缓冲区（data buffer）。这个缓冲区是一个半导体存储器，能存储几兆字节的数据。所请求的数据在磁盘磁道和缓冲区之间传输，传输速率依赖于磁盘的转速。然后在数据缓冲区和主存之间的传输能以它们之间的互连所允许的最大速度进行。

磁盘控制器中的数据缓冲区还能用来为磁盘提供高速缓存机制。当一个读请求到达磁盘时，控制器可以先检查所需要的数据是否已经在缓冲区中。如果在，那么可以在微秒级内将数据传输到存储器中，而不是毫秒级。如果不在，那么使用通常的方法把数据从磁盘磁道上读出，存入缓冲区中，然后再传输到存储器中。由于引用的局部性，后面的请求很可能会引用当前请求所指定数据的后续数据。在对将来请求的预测中，磁盘控制器可以读取比需要数据更多的数据，并把它们放入缓冲区中。当缓冲区用作高速缓存时，它通常足够大，能放下整个磁道的数据。所以一个可能的策略是读写磁头一定位到所需磁道上就开始将这个磁道的内容传输到数据缓冲区中。

4. 磁盘控制器

磁盘驱动器的操作由磁盘控制器电路控制，此外这个电路还提供磁盘驱动器和计算机系统其他部分之间的接口。一个磁盘控制器可以用来控制多个驱动器。

直接跟处理器相连的磁盘控制器包括可由操作系统读写的多个寄存器。于是操作系统和磁盘控制器之间的通信可以使用与I/O接口相同的方法实现，就像在第7章中讨论的那样。磁

盘控制器使用 DMA 方案在磁盘和主存之间传输数据。实际上，数据是从或向数据缓冲区进行传输的，数据缓冲区作为磁盘控制器的一部分来实现。操作系统通过发出读或写请求来启动一次传输，这需要把所需的寻址和控制信息装入控制器的寄存器中。通常，这些信息包括： |315

- 主存地址：传输中所涉及字块的第一个主存单元的地址。
- 磁盘地址：包含所需字块起始位置的扇区单元。
- 字数：所传输的块中的字数。

操作系统发出的磁盘地址是逻辑地址，它与磁盘上对应的物理地址可能不一样。例如，磁盘在格式化时检测到了坏扇区，磁盘控制器记录这些扇区，维护逻辑地址和物理地址之间的映射关系。通常，在每个磁道或同一个柱面的其他磁道中保留了一些空闲扇区，用作坏扇区的替代品。

从磁盘驱动器的角度来看，控制器的主要功能是：

- 寻道：使磁盘驱动器把读写磁头从当前位置移动到所需要的磁道上。
- 读：初始化一个读操作，从磁盘地址寄存器指定的地址开始读。从磁盘中顺序读出的数据被组装成字，并被放入数据缓冲区，用来向主存传输。字数由字计数寄存器决定。
- 写：使用类似于读操作的控制方法把数据传输到磁盘上。
- 错误检查：计算从指定扇区所读出数据的纠错码（ECC）值，并将其与从磁盘读出的相应的 ECC 值进行比较。不匹配时，如果可能，它就纠正这个错误，否则它就产生一个中断，通知操作系统发生了一个错误。在写操作期间，控制器计算要写入数据的 ECC 值，并把这个值存到磁盘上。

### 5. 软盘

上面讨论的磁盘叫做硬盘部件。软盘（floppy disk）是更小、更简单、更廉价的磁盘部件，它包含一个柔韧的可移动的塑料磁盘，这个磁盘覆盖着一层磁性材料。磁盘装在一个塑料外壳中，这个外壳有一个开口，可以从开口处定位读写磁头。磁盘驱动器的旋转轴可以插入磁盘中心的一个孔中转动磁盘。

软盘的主要特点是低成本，并可以方便地移动。但是与硬盘相比，它们的存储容量小得多，访问时间更长，并且失败率更高。近年来，它们已经大量地被 CD、DVD 和作为便携式存储介质的闪存卡所取代。 |316

### 6. RAID 磁盘阵列

处理器速度已经显著地增加，同时，由于机械运动的限制，磁盘驱动器的访问时间仍在毫秒级。减少访问时间的一种方法是使用多个磁盘并行操作。1988 年，加州大学伯克利分校的研究者提出了这样一个存储系统 [5]，他们称之为 RAID，表示廉价磁盘冗余阵列（因为现在所有的磁盘都很便宜，所以 RAID 这个缩写后来被重新解释为独立磁盘冗余阵列（Redundant Array of Independent Disk））。使用多个磁盘还可以提高整个系统的可靠性。研究者们提出了几种不同的配置结构，此后又开发出了更多的配置结构。

基本的配置结构被称为 RAID 0，它是非常简单的。一个大文件被分成一些小块，把这些小块存到不同的磁盘上，从而实现把这个大文件存储到多个不同的磁盘上，这称为数据条带化（data striping）。当对这个文件进行读操作访问时，所有的磁盘可以并行访问它们的那部分数据。因此，数据的传输速率等于单个磁盘的数据传输速率乘以磁盘的数量，但是访问时间，也就是用于在每个磁盘上定位数据起点的寻道时间和旋转延迟并没有减少。由于每个磁盘独立操作，所以访问时间各不一样。数据的各个块被缓冲，以便能重组成完整的文件，并把它作为一

个单一的实体传送给存储器。

各种不同的 RAID 配置形成一个层次结构，其中每一个层次都提供额外的特性。例如，RAID 1 用来提供更高的可靠性，它把数据的相同备份存放到两个磁盘上，而不只是一个磁盘上。这两个磁盘相互被称为彼此的镜像。如果一个磁盘出错了，所有的读写操作都指向它的镜像。层次结构中的其他层次通过不同的奇偶校验方案来获得更高的可靠性，它们不需要整个磁盘的副本。其中一些还具有错误恢复能力。

RAID 概念已经被商家接受，许多制造商都制造了 RAID 系统以用于各种操作系统中。

### 8.10.2 光盘

存储设备也可以使用光学方法实现。人们熟知的用于音频系统的光盘（Compact Disk，CD）是这项技术的第一个实际应用。后来光学技术很快就用于计算机环境，以提供高容量的只读存储介质，被称为只读光盘（CD-ROM）。

第一代 CD 在 20 世纪 80 年代中期由索尼和飞利浦公司开发出来。这个技术利用了可以使用数字来表示模拟的声音信号的可能性。为了提供高质量的声音记录和再现，人们使用了 16 位的模拟声音信号采样，频率为每秒采样 44 100 次。最初，CD 被设计成可以保存 75 分钟的音乐，这需要大约 $3 \times 10^9$ 位（3G 位）的存储量。从那以后，更高容量的设备被开发出来。

#### 1. CD 技术

用于 CD 系统的光学技术利用激光可以聚焦于一个很小的点上这一特性。一个激光束直射到旋转的光盘上，光盘表面上有排列成长螺旋轨道的微小凹痕。这些凹痕把射来的光束反射到一个光电探测器上，用它来探测所存储的二进制模式。

[317]

激光器发出一束相干光，它准确地聚焦到盘面上。相干光由具有相同波长的同步波组成。如果一束相干光与另一束相同类别的光混合，并且这两束光同相位，那么就会产生一束更亮的光。但是，如果这两束光相位差 180 度，那么它们会互相抵消。因此，光电探测器能用来探测光束，在第一种情况下它会看到一个亮点，在第二种情况下它会看到一个暗点。

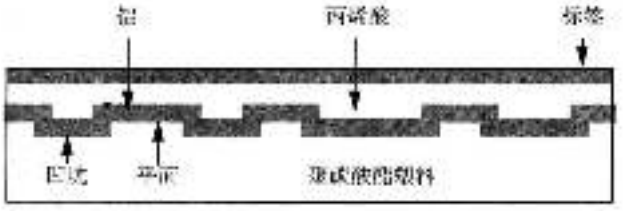
图 8-29a 显示了 CD 一小部分的横切面。最底层是由透明的聚碳酸酯塑料制成的，起到透明盘基的作用。塑料的表面可以通过刻出凹坑（pit）来编程存储数据。未刻的部分称为平面（land）。然后在已编程的盘上覆上一个很薄的铝反射层，再在铝反射层外覆上一个丙烯酸保护层。最后，再覆上最外层并印上标签。光盘的总厚度是 1.2 毫米，这几乎都是聚碳酸酯塑料的厚度，其他层都非常薄。

激光源和光电探测器被放在聚碳酸酯塑料的下方，发出的光线穿过塑料层，在铝反射层上被反射回来，然后向回传播到光电探测器。注意从激光这边来看，凹坑实际上是一个高于平面的凸起。

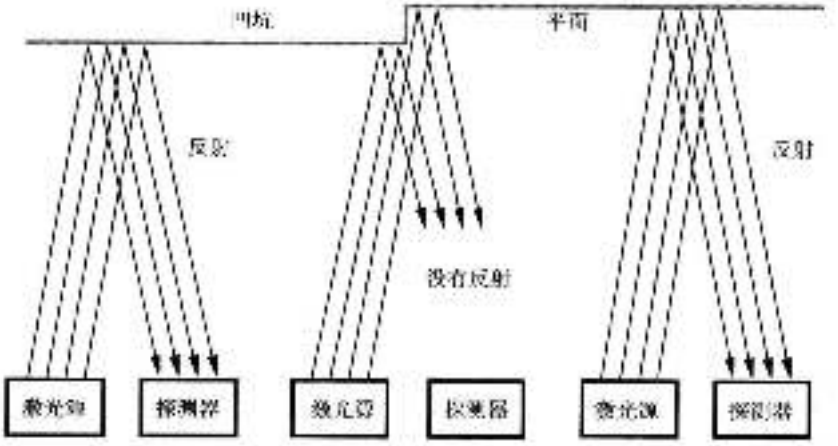
图 8-29b 显示了在激光束沿着盘面扫描并遇到一个从凹坑到平面的转变时发生的情况，图中显示了激光源和探测器的三个不同位置，它们随着光盘的转动而出现。当光只是在凹坑或平面上反射时，探测器会看到反射的光，它将探测到一个亮点。但是当光线移过凹坑和相邻平面之间的边时，就会发生不同的情况。凹坑接近激光源波长的 1/4，因此从凹坑和相邻的平面反射的光线相位差 180 度，它们互相抵消。于是探测器在凹坑－平面和平面－凹坑的转换处看不到反射的光线，它将探测到一个暗点。

图 8-29c 描述了在平面和凹坑之间的一些转换。如果每一个被探测为暗点的转换表示二进制值 1，平的部分表示 0，那么探测到的二进制模式就如图所示。这个模式不直接表示所存储的数据，CD 使用一个复杂的编码方案来表示数据。每字节的数据表示成 14 位的编码，它提供

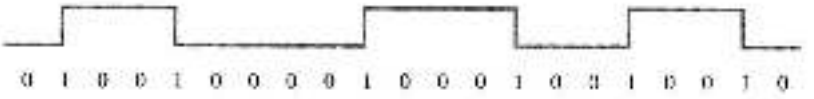
了相当强的错误检测能力。我们不会深入研究这个编码的细节。




a) 横截面




b) 从凹坑到平面的转变




c) 存储的二进制模式

图 8-29　光盘

凹坑在盘面的一条长轨道上排列，这条轨道从盘的中间螺旋伸展到外边缘。但是，习惯上我们把每个跨越 360 度的圆形路径看作一个单独的轨道，这与磁盘使用的技术类似。CD 的直径是 120 毫米，在中心有一个 15 毫米的孔。轨道覆盖了从半径 25 毫米到 58 毫米的区域。轨道间的间隔是 1.6 微米。凹坑 0.5 微米宽，0.8 到 3 微米长。一张盘上有超过 15 000 个的轨道。如果整个轨道螺旋被展开，它的长度将超过 5 千米。

2. CD-ROM

因为 CD 以二进制的形式存储信息，所以它们也适合用作计算机系统的存储介质。主要的挑战是要保证所存储数据的完整性。因为凹坑很小，所以很难完全正确地实现每一个凹坑。在音频和视频应用中，数据中的一些错误是可以容忍的，因为它们对再现的声音和图像产生的影响不会被察觉到。然而，这样的错误在计算机应用中是不可接受的。由于物理上的缺陷是不可避免的，所以需要使用附加的位来提供错误检测和纠正能力。用来存储计算机数据的 CD 被称为只读光盘（CD-ROM），因为像半导体只读存储器芯片一样，它们的内容只能被读取。

318
!
319

数据在CD-ROM的轨道上按块的形式组织，这些块被称为扇区（sector）。扇区有几种不同的格式。一种格式称为模式1，它使用2352字节的扇区。每个扇区有一个16字节的头，包含用于检测扇区起点的同步字段和用于标志扇区的寻址信息。后面跟着的是2048字节的存储数据。在扇区的末尾还有288字节，用于实现错误纠正方案。每个轨道的扇区数各不相同。较长的外侧轨道上有更多的扇区。使用模式1这种格式，CD-ROM有大约650M字节的存储容量。

错误检测和纠正在多个层次上进行。正如前面提到的，存储在CD上的每个字节信息使用14位的代码来编码，它有一定的纠错能力，能纠正单个位的错误。在短脉冲中产生的错误会影响多个位，这可以使用扇区末尾处的错误检查位来检测和纠正。

CD-ROM驱动器以很多种不同的旋转速度工作。基本速度称为1X，是每秒75个扇区，如果使用模式1格式，这将提供153 600字节/秒（150K字节/秒）的数据传输率。更高速度的CD-ROM驱动器用相对基本速度的形式标注。因此，一个56X的CD-ROM的数据传输率是1X的CD-ROM的56倍，或者大约6M字节/秒。这个传输速率要比硬盘的传输速率低得多，硬盘的传输速率在每秒几十兆字节的范围内。另一个显著的性能区别是寻道时间，在CD-ROM中寻道时间可能是几百毫秒。所以，就性能而言，CD-ROM显然要劣于磁盘。它们的吸引力在于其物理尺寸小，成本低，还有容易作为可拆装和可移动的海量存储介质。因此，它们广泛应用于软件发布、教科书、应用程序和视频游戏中，等等。

3. 可刻录CD

以上描述的CD是只读的设备，信息在制造的时候存储。首先，使用高功率激光在需要凹坑的位置烧出洞，以此来生产一个母盘。然后根据母盘生产一个模具，它在有洞的地方会有凸起。通过把熔化的聚碳酸酯塑料注入模具来进行复制，就可以生产出与母盘凹坑模式相同的CD了。这个过程显然只适合包含相同信息的CD的批量生产。

20世纪90年代末有一种新类型的CD被开发出来，计算机用户可以很容易地把数据刻到它上面，它被称为可刻录光盘（CD-Recordable，CD-R）。在生产过程中，在光盘上做出了一条覆盖着有机染料的闪亮的螺旋轨道。然后，在CD-R驱动器中使用激光在有机染料上烧出凹坑来。被烧的点变得不透明了，当读取CD的时候，这些点比闪光的区域反射的光要少。这个过程是不可逆转的，这意味着写入的数据被永久地存储。光盘未使用的部分可以在以后用来存储其他数据。

320

4. 可擦写CD

最灵活的CD是那些可以由用户多次写入的CD，它们被称为可擦写光盘（CD-ReWritable，CD-RW）。

CD-RW的基本结构与CD-R类似。但是在刻录层没有使用有机染料，而是使用银、铟、锑和碲的合金。这种合金在被加热和冷却时发生的反应非常有趣，也非常有用。如果它被加热到超过熔点（500摄氏度）然后被冷却，就变成非晶体状态，此时它吸收光线。但是如果它只被加热到200摄氏度左右，然后保持一段时间，就会发生称为退火（annealing）的过程，这将导致合金变成晶体状态，此时它允许光线穿过。如果用晶体状态来表示平面区域，那么可以通过把选定点加热到超过熔点来创建凹坑。存储的数据可以使用退火过程来擦除，这使合金回到统一的晶体状态。反射材料放在刻录层的上面，在光盘被读取的时候用来反光。

CD-RW驱动器使用三种不同功率的激光。最高功率用来刻录凹坑；中等功率用来把合金变成晶体状态，它被称为擦除功率；最低功率用来读取所存储的信息。

为读写CD-RW光盘而设计的CD驱动器通常可以与其他光盘介质一起使用，它能读取

CD-ROM，能读写 CD-R。它按标准互连接口的要求设计，例如 SATA 和 USB。

CD-RW 光盘提供了低成本的存储介质。它们适合于信息的档案存储，这些信息的范围可以从数据库到图形图像。也可以用于信息的小批量发布，与 CD-R 一样，还可以用于备份。CD-RW 技术已经使得 CD-R 不那么重要了，因为它以稍高一点的成本提供了非常好的性能。

5. DVD 技术

CD 技术的成功和对更大存储容量的不断寻求导致了 DVD（Digital Versatile Disk，数字多功能光盘）技术的发展。第一个 DVD 标准是由一个企业联盟在 1996 年制定的，目标是能在 DVD 盘的一面上存储一整部电影。

DVD 光盘的物理尺寸与 CD 相同，盘体 1.2 毫米厚，直径 120 毫米。通过改变下面几项设计使得它的存储容量比 CD 高得多：

- 使用波长为 635nm 的红色激光器代替 CD 中使用的波长为 780nm 的红外激光器。较短的波长可以把光线聚焦到一个更小的点上。
- 凹坑更小，最小长度为 0.4 微米。
- 轨道更接近，轨道间的距离是 0.74 微米。

使用了这些改进使得 DVD 的容量可达到 4.7G 字节。

使用两层或两面的盘可以进一步增加容量。单层单面的盘在标准中定义为 DVD-5，它的结构与图 8-29a 中的 CD 几乎一样。双层盘使用两个层，每层都刻有轨道。第一层是一个透明的盘基，与 CD 盘一样。但是它没有使用铝来反射，而是在这层的平面和凹坑上覆盖一层半透明的材料，把它作为一个半反射体。然后在这层材料的表面上也刻上凹坑来编程以存储数据。在第二层上的凹坑和平面上再放上一层反射材料。这种盘通过把激光束聚焦到需要的层上来实现读取。当激光束聚焦到第一个层上时，半透明材料反射回足够的光来检测所存储的二进制模式；当激光束聚焦到第二个层上时，反射材料返回的光显示了这一层所存储的信息。在这两种情况下，没有被激光束聚焦的层反射回少量的光，它被探测器电路当作噪声消除了。两个层的总存储容量是 8.5G 字节，在标准中这种盘被称为 DVD-9。 [321]

两个单面盘可以放在一起形成一个类似三明治的结构，其中上面的那张盘要翻个面。这可以使用单层盘来实现，像在 DVD-10 中规定的，组合成的盘的容量是 9.4G 字节。它也可以使用双层盘来实现，像在 DVD-18 中规定的，可产生 17G 字节的容量。

DVD 驱动器的访问时间与 CD 驱动器相似，但是，当 DVD 盘以相同的速度旋转时，由于它的凹坑密度更高，所以它的数据传输率要高得多。人们还开发出了可擦写的 DVD 设备，可以提供很大的存储容量。

### 8.10.3　磁带系统

磁带适合用于大量数据的离线存储，它们通常用于备份和归档存储。磁带记录时使用了与磁盘相同的原理，主要的区别是磁带的磁性薄膜是镀在一条很细的宽度为 0.5 到 0.25 英寸的塑料带上。在磁带的宽度方向上平行记录了 7 或 9 位（对应一个字符）。它与磁带的运动方向垂直。磁带上每个位的位置都有一个单独的读写磁头，所以一个字符的所有位能并行读出或写入。字符中有一个位用作奇偶位。

磁带上的数据按记录（record）的形式组织，记录由间隙分开，如图 8-30 所示。磁带的运动只有在记录间隙位于读写磁头下方时才会停止。记录的间隙足够长，允许磁带在到达下一个记录的起点之前能达到正常速度。如果使用类似于图 8-27c 中的编码方案在磁带上记录数据，那么记录间隙可以表示为没有磁性变化的区域。这允许记录间隙的检测独立于所记录的数

据。为了帮助用户组织大量的数据，把一组相关的记录称为文件（file）。文件的起点通过文件标记（file mark）来识别，如图 8-30 所示。文件标记是一个特殊的单字符或多字符记录，在它前面通常有一个比记录间间隙长一些的间隙。文件标记后的第一个记录可以用作文件的文件头
322 （header）或文件标识符（identifier），这允许用户在包含大量文件的磁带中搜索特定的文件。

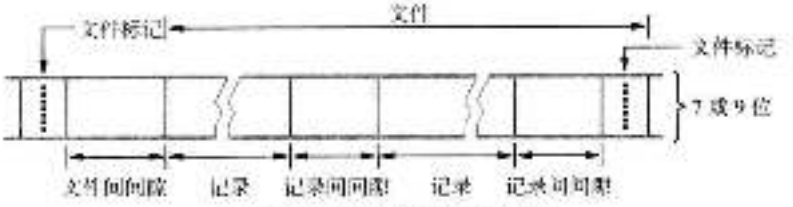



图 8-30 磁带上的数据组织结构

**盒式磁带系统**

磁带系统已经被开发用来为在线磁盘存储做备份。其中有一种这样的系统，它使用 8 毫米的视频格式磁带，磁带位于一个盒子中。这种部件被称为盒式磁带，它们的存储容量为 2G 到 5G 字节，能以每秒几百 K 字节的速度传输数据。可通过一个螺旋扫描系统横跨磁带进行操作来实现读写，这与盒式录像带驱动器类似。位密度能达到每平方英寸几千万比特。现在已经有能自动装盒和卸盒的多盒系统，所以数十 G 字节的在线存储备份可以不需要人来干涉。

## 8.11 结束语

存储器层次结构的设计对计算机系统的性能来说是非常重要的。现代操作系统和应用程序对存储器的容量和速度都有很高的要求。在本章中，我们介绍了存储器系统中最重要的技术和组织结构细节，以及它们如何演变以满足这些要求。

半导体技术的发展使得存储器芯片的速度和容量有了显著的提高，每位的成本也大幅下降。计算机存储器的性能可通过使用存储器层次结构来进一步提高。今天，大容量且价格合理的主存都是使用动态存储器芯片实现的。系统中还总是会提供一级或多级的高速缓存，高速缓存的引入大大减少了处理器所看到的有效存储器访问时间。而虚拟存储器则使得主存看起来比物理内存要大。

磁盘依然是辅助存储的主要技术。它们提供了极大的存储容量，在单个驱动器上可达到或超过万亿字节，并且每位的成本也很低。但是，闪存半导体技术也已开始在某些应用中具有
323 一定的竞争能力。

## 8.12 问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。

**例 8.2**

**问题：** 对于一个使用 512K × 8 存储器芯片的 8M × 32 存储器，描述一种类似于图 8-10 的结构。

**解答：** 所需的结构本质上与图 8-10 中的一样，只是需要 16 行，每行有 4 个 512K × 8 的芯片。地址线 $A_{22}$ 需要连接到所有的芯片上，地址线 $A_{21:20}$ 连接到一个 4 位的译码器上，以选择 16 行中的一行。

**例 8.3**

**问题：** 一个计算机系统使用 32 位的存储器地址。它的主存有 1G 字节。它有一个 4K 字节的高速缓存，按组相联的形式组织，每组 4 个块，每块 64 个字节。

（a）计算主存地址中的 Tag，Set 和 Word 字段的位数。

（b）假设高速缓存初始为空，处理器从位置 0 开始的连续字单元中取 1088 个字，每个字 4 个字节，然后它再重复这个取操作序列 9 次。如果高速缓存比主存快 10 倍，估算一下使用高速缓存后性能提高的倍数。假设在块替换时使用 LRU 算法。

**解答：**连续的地址是针对字节来说的，也就是每个字节对应一个地址，即按字节编址。

（a）一个块有 64 个字节，因此 Word 字段为 6 位，每组有 4 × 64 = 256 字节，所以有 4K / 256 = 16 组，Set 字段需要 4 位，剩下 32 − 4 − 6 = 22 位作为 Tag 字段。

（b）1088 个字构成 68 个块，占用主存的块 0 到块 67。高速缓存有 64 块的空间，因此，在第一遍中将块 0，1，2，…，63 从主存读到高速缓存后，高速缓存就满了。接下来的编号为 64 到 67 的那 4 块映射到第 0，1，2 和 3 组中，每一块将替换所在组中最近最少使用的高速缓存块，就是块 0。在第二遍中，主存块 0 将被重新装入高速缓存的第 0 组中，因为它已经被块 64 覆盖了。它将被放到第 0 组在那个时候最近最少使用的块中，就是块 1。接下来，主存块 1，2 和 3 将分别替换高速缓存中第 1，2 和 3 组中的块 1。主存块 4 到 15 将在高速缓存中找到。在第 0 到第 3 组中块 1 位置上的主存块 16 到 19 现在被覆盖，并将被重新装入这些组中块 2 的位置上。 [324]

随着执行的运行，16 个高速缓存组中前两个组的所有主存块在下一遍使用它们之前都会被覆盖。主存块 0，16，32，48 和 64 不断地彼此替换，因为它们竞争高速缓存中第 0 组的 4 个块位置。在第 1 组（主存块 1，17，33，49，65），第 2 组（主存块 2，18，34，50，66）和第 3 组（主存块 3，19，35，51，67）中也会发生同样的事情。最后 12 个组（第 4 组到第 15 组）中的主存块在第一遍中被取出一次，然后在接下来的 9 遍中一直保持在高速缓存中。

总之，在第一遍中，所有 68 个块都被从主存中取出，在后 9 遍的每一遍中，48 个块能在高速缓存的第 4 组到第 15 组中找到，剩下的 20 个块必须从主存中读取。假设 $c$ 为高速缓存的访问时间，则

$$\text{性能提高的倍数} = \frac{\text{无高速缓存时的时间}}{\text{有高速缓存时的时间}}$$

$$= \frac{10 \times 68 \times 10c}{1 \times 68 \times 11c + 9(20 \times 11c + 48c)}$$

$$= 2.15$$

这个例子说明了在执行程序循环期间 LRU 算法的缺点。这种情况下另一种算法的性能参见习题 8.9。

**例 8.4**

**问题：**假设一台计算机的处理器有两个一级高速缓存（一个用于指令，另一个用于数据）和一个二级高速缓存。假设 $c$ 为两个一级高速缓存的访问时间，将一个块从二级高速缓存传输到一级高速缓存的失效开销大约为 $15c$，从主存传输到二级高速缓存大约为 $100c$。针对该问题，我们假设指令和数据的命中率相同，并且一级高速缓存和二级高速缓存的命中率分别为 0.96 和 0.80。

（a）在一级和二级高速缓存中访问都失效，因而需要访问主存的比例是多少？

（b）处理器所看到的平均访问时间是多少？

（c）假设二级高速缓存的理想命中率为 1，处理器所看到的平均存储器访问时间将被减少多少倍？

（d）考虑下面对存储器层次结构的改变：二级高速缓存被移走，两个一级高速缓存的容量被增加，因而它们的失效率减少了一半。在这种情况下，处理器所看到的平均存储器访问时间是多少？ [325]

**解答：**只具有一个高速缓存级别的平均存储器访问时间在 8.7.1 节中给出，如下：

$$t_{avg} = hC + (1 - h)M$$

具有一级和二级高速缓存的平均存储器访问时间在 8.7.2 节中给出，如下：

$$t_{avg} = h_1C_1 + (1 - h_1)(h_2C_2 + (1 - h_2)M)$$

（a）在一级和二级高速缓存中访问都失效时的存储器访问比例为

$$(1 - h_1)(1 - h_2) = (1 - 0.96)(1 - 0.80) = 0.008$$

（b）使用两个高速缓存级别的平均存储器访问时间为

$$t_{avg} = 0.96\tau + 0.04(0.80 \times 15\tau + 0.20 \times 100\tau)$$
$$= 2.24\tau$$

（c）如果在二级高速缓存中没有失效，我们得到：

$$t_{av}(\text{理想}) = 0.96\tau + 0.04 \times 15\tau = 1.56\tau$$

因此，

$$\frac{t_{av}(\text{实际})}{t_{av}(\text{理想})} = \frac{2.24\tau}{1.56\tau} = 1.44$$

（d）如果使用更大的一级高速缓存，并将二级高速缓存移除，则访问时间为

$$t_{avg} = 0.98\tau + 0.02 \times 100\tau = 2.98\tau$$

**例 8.5**

**问题**：一个由 32 位的数组成的 1024 × 1024 数组按以下方法标准化。对每一列，找到最大的元素，然后将这个列中的所有元素都除以这个元素的值。假设虚拟存储器中每页包含 4K 字节，并且在这个计算中主存的 1M 字节被分配用来存储数组数据。假定当发生页故障时从磁盘装载一个页到主存需要 10 毫秒。

（a）假设一次处理该数组的一列，如果数组元素按列顺序存储在虚拟存储器中，会发生多少次页故障？需要多长时间才能完成这个标准化过程？

（b）假设元素按行顺序存储，重复（a）的问题。

（c）当数组按行顺序存储在存储器中时，提出另一种处理这个数组的方法，以减少页故障的次数。
326 估算页故障的次数以及你的解决方案所需要的时间。

**解答**：每一个 32 位的数构成 4 字节，因此，每页容纳 1024 个数。在计算中被分配用来存储数据的那 1M 字节的主存部分有 256 页的空间。

（a）每一列存储在一页中，将每一列放到主存中有一次页故障，共 1024 次页故障。

$$\text{处理时间} = 1024 \times 10 \text{ 毫秒} = 10.24 \text{ 秒}$$

（b）每一列的处理需要两遍，第一遍为了找到最大的元素，第二遍为了执行标准化。当处理第一列时，每个元素的访问导致一次页故障，会将相应行的所有元素都装入主存中。256 个元素被检查后，主存已满。接下来 256 个元素的访问导致页故障，替换主存中所有的数据，然后重复这个过程。因此，数组中每一个元素的每一次访问都会发生一次页故障。

$$\text{处理时间} = 2 \times 1024 \times 1024 \times 10 \text{ 毫秒} = 20\,972 \text{ 秒} = 5.8 \text{ 小时}$$

（c）对于数据的这种布局，另一种更有效的方法是在第一遍中先完成每一列的 1/4，然后再处理第二个 1/4，等等。第二遍采用相同的方法处理。在这种情况下，通过数组的每一遍导致 1024 次页故障，共 2048 次。

$$\text{处理时间} = 2048 \times 10 \text{ 毫秒} = 20.48 \text{ 秒}$$

这个例子说明了当主存容量不足以存放应用程序的情况下，页故障的次数是如何显著增加的。这种行为被称为系统颠簸（thrashing）。

**例 8.6**

**问题**：考虑一个磁盘访问的长序列，磁盘的平均寻道时间为 6 毫秒，平均旋转延迟为 3 毫秒，所访问块的平均大小为 8K 字节，从磁盘传输数据的速率为 34M 字节 / 秒。

（a）假设这些数据块随机分布在磁盘上，估算一下寻道操作和旋转延迟所占总时间的平均百分比。

（b）重新排列这些磁盘访问，使得在 90% 的情况中下一次访问的数据块与这次访问的数据块在同一个柱面上，在这种情况下重复第一问。

**解答**：传输一个数据块需要花费 8K / 34M = 0.23 毫秒。

（a）访问每个数据块所需的总时间为 6 + 3 + 0.23 = 9.23 毫秒。寻道和旋转延迟所占时间的比例为
327 9 / 9.23 = 0.97 = 97%。

（b）90% 的情况下只需要旋转延迟。因此，访问一个数据块的平均时间是 0.9 × 3 + 0.1 × 9 + 0.23 = 3.89 毫秒。寻道和旋转延迟所占时间的比例为 3.6 / 3.89 = 0.92 = 92%。

## 习题

[M] 8.1 考虑图 8-6 中的动态存储器单元。假设 C = 30 飞法（$10^{-15}$ 法），通过晶体管的泄漏电流大约是 0.25 皮安（$10^{-12}$ 安），当电容充满电荷时，它两端的电压为 1.5 伏。这个单元必须在电压降到 0.9 伏之前进行刷新。估算最小刷新率。

[M] 8.2 考虑由 SDRAM 芯片构造的主存，数据以脉冲串的形式传输，如图 8-9 所示，只是脉冲长度为 8。假设并行传输 32 位数据，如果使用 400MHz 的时钟，传输下面数据需要多少时间：

（a）32 字节数据

（b）64 字节数据

每种情况的延迟是多少？

[E] 8.3 对于一个使用 1M × 4 存储器芯片的 16M × 32 存储器，描述它的类似图 8-10 的结构。

[E] 8.4 分析下面这句话为什么不对："使用更快的处理器芯片可以相应地提高计算机的性能，尽管主存的速度保持不变。"

[M] 8.5 一台计算机的主存是按字节编址的，字长为 32 位。一个程序包含两个嵌套的循环——较小的内部循环和较大的外部循环。程序的一般结构在图 P8-1 中给出。图中十进制的主存地址显示了两个循环的位置和整个程序的起点和终点。在程序不同部分（8–52、56–136、140–240 等）的所有存储单元都包含按顺序执行的指令。这个程序在按直接映射方法（参见图 8-16）组织指令高速缓存的计算机上运行，并且各个参数如下：

高速缓存大小　　1K 字节

块大小　　128 字节

指令高速缓存的失效开销是 80*t*，其中 *t* 是高速缓存的访问时间。计算在图 P8-1 中的程序执行期间取指令需要的总时间。

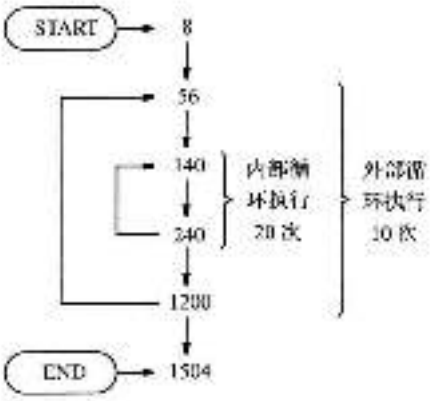



图 P8-1　习题 8.5 的程序结构

328

[M] 8.6 一台字长为 16 位的计算机有一个直接映射的高速缓存，用于指令和数据。主存地址是 16 位长，且主存是按字节编址的。高速缓存的容量很小，它只包含 4 个 16 位的字，每个字构成一个高速缓存块，并有一个相联的 13 位的标志，如图 P8-2a 所示。使用地址的低 3 位来访问高速缓存中的字。当读指令或数据操作数期间发生一次失效时，把请求的字从主存读出，然后送到处理器中，同时它被拷贝到高速缓存中，并且其块号被存到相联的标志中。考虑下面的短循环，其中所有的指令都是 16 位长。

```
LOOP:   Add         R0, (R1)+
        Decrement   R2
        BNE         LOOP
```

假设在进入循环前，寄存器 R0、R1 和 R2 包含的内容分别为 0、054E 和 3。再假设主存中包含图 P8-2b 所示的数据，其中所有的入口都以 16 进制给出。循环在 LOOP=02EC 处开始。在 Add 指令中使用自动增量寻址方式去访问一个列表（包含 3 个数）中的连续的数，并把它们加到寄存器 R0 中。计数寄存器 R2 递减直到它到达 0，这时从循环中退出。

（a）开始时高速缓存是空的，写出每执行完一次循环后高速缓存中的内容，包括标志位。

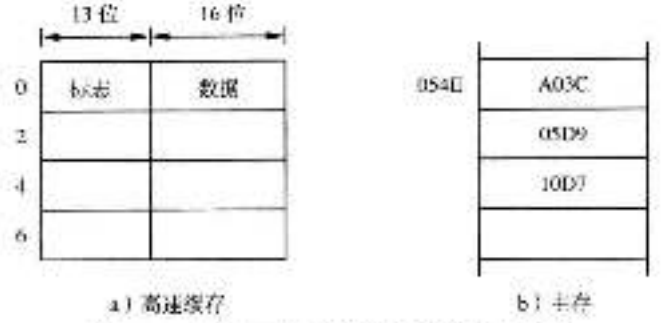
（b）假设高速缓存和主存的访问时间分别是 *t* 和 10*t*，计算每次循环的执行时间，只计算存储器访问时间。

329

[M] 8.7 重复习题 8.6，假设在高速缓存中只存储指令，数据操作数直接从主存中获取，并且不拷贝到高速缓存中。为什么这种选择比把指令和数据都装入高速缓存中执行速度更快？

[E] 8.8 一个组相联高速缓存共包含 64 个块，每组 4 块。主存有 4096 个块，每块有 32 个字。假设有一

个32位字节编址的地址空间。在每个Tag、Set和Word字段中各有多少位？




a）高速缓存　　b）主存

图P8-2　习题8.6中高速缓存和主存中的内容

[M] 8.9 考虑例8.3中的高速缓存，假设每当从主存中读出一个新块，并且它在高速缓存中对应的组已经满了，新块就替换这个组中最近使用的块。推算出第二问在这种情况下的解决方案。

[D] 8.10 8.6.3节用图8-20中的程序描述了不同高速缓存映射技术的效果。假设这个程序的第二个循环中元素处理的顺序改成与第一个循环相同，也就是使用如下语句控制第二个循环

for $i := 0$ to 9 do

推算出图8-21到图8-23针对这个程序的等价形式。从这个习题可以得出什么结论？

[M] 8.11 一台按字节编址的计算机有一个能容纳8个32位字的小数据高速缓存，每个高速缓存块有1个32位的字。当执行一个给定的程序时，处理器按下列十六进制地址顺序读取数据：

200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4

这个模式重复四遍。

（a）假设高速缓存初始为空。如果使用直接映射方式，写出每次循环结束时高速缓存中的内容，并计算命中率。

330

（b）如果换成采用LRU替换算法的相联映射高速缓存，重复第一问。

（c）如果是4路组相联高速缓存，重复第一问。

[M] 8.12 假设每个高速缓存块包含两个32位的字，重复习题8.11。在第三问中，使用2路组相联高速缓存，它采用LRU替换算法。

[E] 8.13 在很多计算机中，高速缓存块的大小在32字节到128字节之间的范围内。使用更大或更小的高速缓存块的主要优点和缺点各是什么？

[M] 8.14 一台计算机有L1和L2两级高速缓存。对于$h_2$=0.75和$h_2$=0.85这两个值，绘制两张平均存储器访问时间（$y$轴）相对于命中率$h_1$（$x$轴）的图，$h_1$使用值0.90、0.92、0.94和0.96。假设L1和L2高速缓存的失效开销分别是15$\tau$和100$\tau$，其中$\tau$是L1高速缓存的访问时间。

[E] 8.15 考虑例8.4中描述的两级高速缓存，该例子第二问的解决方案中给出了平均访问时间是2.24$\tau$。如果所有其他的参数都跟该例子中的相同，要将$t_{ave}$减少为1.5$\tau$，需要$h_1$的值为多少？通过提高二级（L2）高速缓存的命中率能否达到相同的结果？

[E] 8.16 考虑下面对高速缓存概念的类比。一个修理员到一个住宅修理加热系统。他带了一个工具箱，里面有他最近在类似工作中使用的工具。他反复使用这些工具，直到需要其他的工具。可能在屋外的卡车上有他需要的工具，但是，如果卡车上没有，他就必须回店铺中去拿。

假设我们把工具箱、卡车和店铺对应成一级（L1）高速缓存、二级（L2）高速缓存和计算机主存，那么这个类比是否恰当？讨论它正确和不正确的地方。

[E] 8.17 使用一个二级（L2）高速缓存的目的是为了减少一级（L1）高速缓存的失效开销，进而减少处理器所看到的存储器访问时间。另一种方法是增加一级（L1）高速缓存的容量来提高它的命中率。限制该方法效用的因素是什么？

[M] 8.18 分析8.7.1节的例8.1中给出的假设“失效开销对读访问和写访问都是相同的”为什么不对。在

阐述你的答案时考虑 8.6 节所描述的直接写和写回这两种情况。

[M] 8.19 考虑一个计算机系统，其物理内存的可用页在几个应用程序中分配。操作系统监视页的传送活动并动态地调整分配给不同程序的页数量。给出一个能被操作系统用来最小化总体页传送率的合适策略。

[M] 8.20 在一台有虚拟存储器系统的计算机中，一条指令的执行可能被页故障中断。为了能使这条指令在以后能继续执行，需要保存哪些状态信息？注意把一个新页装入主存涉及 DMA 传输，它需要执行其他的指令，放弃被中断的指令，以后再把它完全重新执行一遍是否更简单？重新执行一遍能实现吗？ 331

[E] 8.21 当一个程序产生一个对不在物理主存中的页引用时，这个程序的执行会被挂起，直到所请求的页从磁盘装载到主存。当一个页中一条指令的操作数在另一个页中时，会有什么困难发生？为了处理这种情况，处理器必须拥有何种能力？

[M] 8.22 一个磁盘部件有 24 个记录面，它共有 14 000 个柱面，平均每个磁道有 400 个扇区，每个扇区包含 512 字节的数据。

（a）这个磁盘部件最多能存储多少字节？

（b）以 7200rpm 的速度旋转时，数据传输率是每秒多少字节？

（c）如果使用 32 位的字，给出一个指定磁盘地址的合理方案。

[M] 8.23 考虑一个磁盘访问的长序列，该磁盘的平均寻道时间是 8 毫秒，平均旋转延迟是 3 毫秒，数据传输率是 60M 字节 / 秒。所访问块的平均大小是 64K 字节。假设每个数据块存储在连续的扇区中。

（a）假设这些块随机分布在磁盘上，估算寻道操作和旋转延迟所占总时间的平均百分比。

（b）假设从邻近的柱面按顺序传输 20 个块，寻道时间减少为 1 毫秒。如果这些块随机分布在这些柱面上，那么总的传输时间是多少？

[M] 8.24 磁盘系统的平均寻道时间和旋转延迟分别是 6ms 和 3ms，向或从磁盘传输数据的速率是 30M 字节 / 秒，并且所有的磁盘访问都是对存储在连续扇区上的 8K 字节数据的访问。数据块存储在磁盘上的任意位置。磁盘控制器有 8K 字节的缓冲区。磁盘控制器、处理器和主存都连到一个单一的总线上，总线数据宽度为 32 位，向或从主存的一次总线传输花费 10 纳秒。

（a）能同时向或从主存传输数据的磁盘部件的最大数目是多少？

（b）在一个长时间段内要传输一系列独立的 8K 字节，在这段时间内平均有百分之几的主存访问被磁盘部件使用？

[M] 8.25 在大多数虚拟存储器系统中使用磁盘作为辅助存储设备，用来存储程序和数据文件，哪些磁盘参数将对页大小的选择产生影响？

## 参考文献


1. T.C. Mowry, "Tolerating Latency through Software-Controlled Data Prefetching," *Tech. Report CSL-TR-94-628*, Stanford University, Calif., 1994.
2. J.L. Baer and T.F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proceedings of Supercomputing '91*, 1991, pp. 176–186. 332
3. J.W.C. Fu and J.H. Patel, "Stride Directed Prefetching in Scalar Processors," *Proceedings of the 24th International Symposium on Microarchitecture*, 1992, pp. 102–110.
4. D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981, pp. 81–85.
5. D.A. Patterson, G.A. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988, pp. 109–166. 333 / 334

# 算术运算

## 本章目标

在本章中你将学习以下内容：

- 加法器和减法器电路
- 基于超前进位逻辑电路的高速加法器
- 有符号数乘法的 Booth 算法
- 基于进位保留加法的高速乘法器
- 除法运算逻辑电路
- 符合 IEEE 标准的浮点数算术运算

335

两个数的加和减是所有计算机中机器指令层次的基本运算，同其他的算术和逻辑运算一样，这些运算是在处理器的算术逻辑部件（ALU）中实现的。本章将介绍实现算术运算的逻辑电路。执行加法或减法运算所需要的时间会影响处理器的性能。与加减运算相比，乘除运算需要更复杂的电路，并且也会影响处理器的性能。我们将介绍几种现代计算机中使用的高速执行算术运算的技术，也将对浮点数运算进行描述。

我们在第 1 章的 1.4 节中描述了有符号二进制数的表示，还指出了对于执行加减运算来说，补码是最好的表示方法。在图 1-6 的例子中我们看到，两个 $n$ 位有符号数可以使用 $n$ 位二进制加法相加，对符号位用与其他位一样的方法进行处理。换句话说，计算无符号二进制数加法的逻辑电路也可以用来计算补码表示的有符号数的加法。本章的前两节将介绍加法与减法的逻辑电路。

## 9.1 有符号数加减法

图 9-1 显示了将两个数 $X$、$Y$ 中同等权重的两位 $x_i$ 和 $y_i$ 相加时，求和函数与进位输出函数的真值表。图中还给出了这两个函数的逻辑表达式，以及两个 4 位无符号数 7 和 6 相加的例子。请注意加法过程中的每一步都必须包含一个进位输入位。我们用 $c_i$ 表示第 $i$ 步的进位输入，同时它也是第 $(i-1)$ 步的进位输出。

图 9-1 中 $s_i$ 的逻辑表达式可以用一个三输入端的异或门实现，在图 9-2a 中它是单步二进制加法所需逻辑的一部分。如图中所示，进位输出函数 $c_{i+1}$ 使用与 - 或电路实现。图中还使用了一个方便的符号——全加器（full adder，FA）来表示单步加法所需的全部逻辑电路。

图 9-2b 显示了 $n$ 个全加器部件的级联，可以用来计算两个 $n$ 位数的加法。由于进位要在级联中像水波一样传播下去，所以这种结构被称为行波进位加法器（ripple-carry adder）。

最低有效位（least-significant-bit，LSB）位置上的进位输入 $c_0$ 提供了为数字加 1 的简单方法。例如，为得到一个数的补码需要对该数的反码加 1。进位信号还可以用来将 $k$ 个加法器互连起来，从而形成可以处理 $kn$ 位输入数字的加法器，如图 9-2c 所示。

### 加法 / 减法逻辑部件

图 9-2b 中的 $n$ 位加法器可以用来计算补码数 $X$ 和 $Y$ 的加法，其中 $x_{n-1}$ 与 $y_{n-1}$ 位为符号位。

336

进位输出位 $c_n$ 并不是加法结果的一部分。1.4 节讨论了算术溢出。当两个操作数的符号相同，

但计算结果的符号不同时会发生溢出。因此，可以为 $n$ 位加法器添加执行以下逻辑表达式的溢出检测电路：

$$溢出 = x_{n-1}y_{n-1}\overline{s}_{n-1} + \overline{x}_{n-1}\overline{y}_{n-1}s_{n-1}$$

| $x_i$ | $y_i$ | 进位输入 $c_i$ | 和 $s_i$ | 进位输出 $c_{i+1}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i\overline{y}_i\overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

示例：

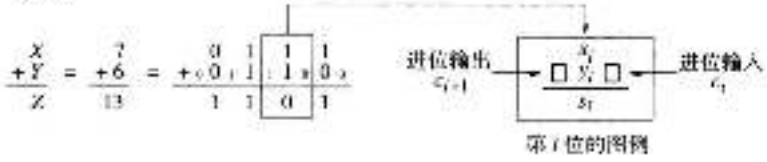
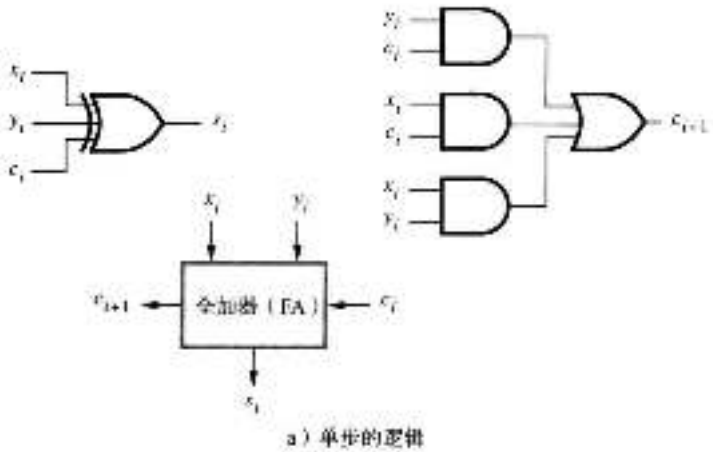



图 9-1 单步二进制加法的逻辑说明

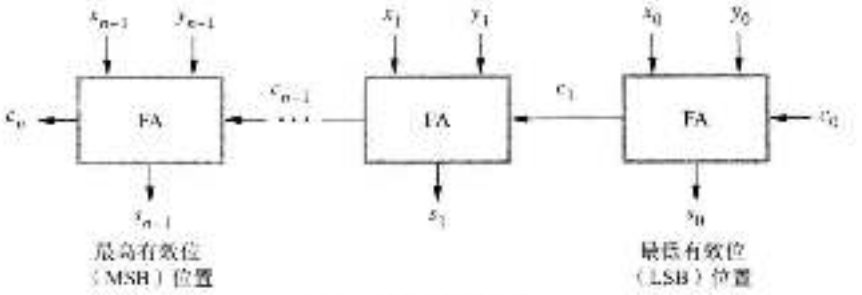
还可以证明当进位位 $c_n$ 与 $c_{n-1}$ 不同时就发生了溢出（参见习题 9.5）。因此，使用一个异或门执行表达式 $c_n \oplus c_{n-1}$ 就可以得到更简单的溢出检测电路。

为了执行补码数 $X$ 与 $Y$ 的减法操作 $X-Y$，我们先得到 $Y$ 的补码再将其与 $X$ 相加。图 9-3 所示的逻辑电路可以根据加法/减法输入控制线上的值来执行加法或减法操作。执行加法时将控制线置 0，将 $Y$ 不加改变地送至加法器的一个输入端，并使进位输入信号 $c_0$ 为 0。当将加/减控制线置 1 时，将 $Y$ 通过异或门形成反码（即按位取反），并且将 $c_0$ 置 1 从而得到 $Y$ 的补码。回顾一下，负数补码的计算方法与正数是完全相同的。可以在图 9-3 中添加一个异或门来检测溢出条件 $c_n \oplus c_{n-1}$。 337

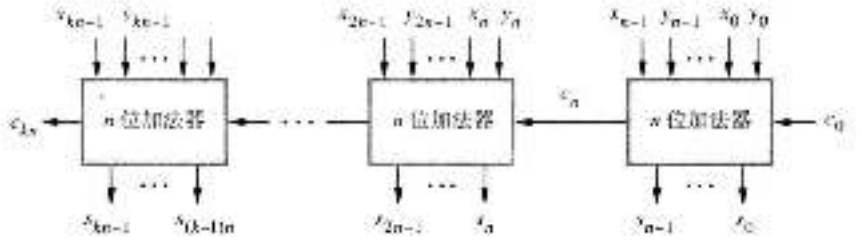



a）单步的逻辑

图 9-2 二进制数的加法逻辑




b) $n$ 位行波进位加法器




c) $k$ 个 $n$ 位加法器的级联

338

图 9.2 （续）

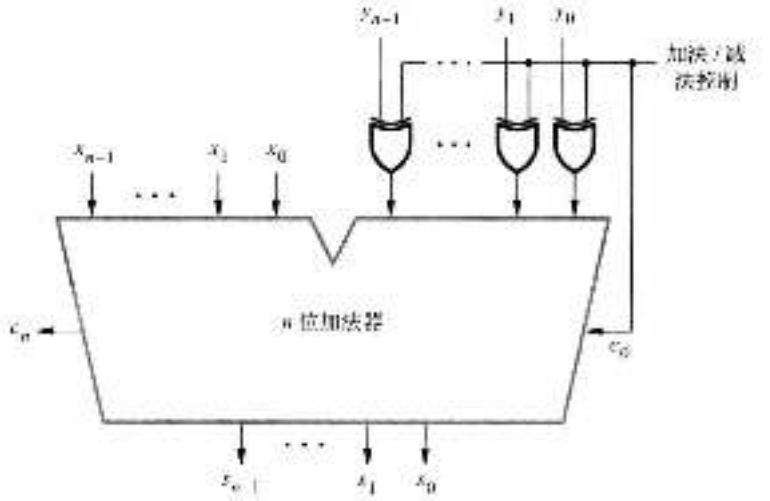



图 9-3 二进制加法/减法逻辑电路

## 9.2 快速加法器设计

如果图 9-3 中的加法/减法电路使用 $n$ 位行波进位加法器，可能需要经历很长的延迟才能得到输出 $s_1$ 至 $s_{n-1}$ 以及 $c_n$。这个延迟是否可以接受只取决于其他处理器部件的速度以及寄存器和高速缓存的数据传输时间的要求。一个逻辑门网络所需的延迟依赖于构造该网络的集成电路的电子工艺，以及从输入至输出所经历的逻辑门的数目。通过任何一个以特定工艺制造的门构成的组合电路所需的延迟，是该电路中最长信号传输路径上所有逻辑门延迟的总和。对于

$n$ 位行波进位加法器，最长路径是从最低有效位 LSB 位置上的输入 $x_0$、$y_0$ 及 $c_0$ 到最高有效位（most-significant-bit，MSB）位置上的输出 $c_n$ 与 $s_{n-1}$。

使用图 9-2a 所示的实现电路，$c_{n-1}$ 在 $2(n-1)$ 个门延迟后得到，$s_{n-1}$ 需要再经过一个异或门延迟后得到。而最终的进位输出 $c_n$ 需要经历 $2n$ 个门延迟后才能得到。因此，如果使用行波进位加法器实现图 9-3 所示的加法 / 减法部件，可以在 $2n$ 个门延迟后得到所有的求和位，包括 $Y$ 输入端经历的异或门延迟。如果使用 $c_n \oplus c_{n-1}$ 来检测溢出，则检测结果可以在 $2n+2$ 个门延迟后得到。

有两种方法可以减少加法器的延迟。第一种方法是使用最快的电子工艺。第二种方法是使用一种称为超前进位网络的逻辑门网络，下面我们将描述超前进位网络。 339

### 超前进位加法

快速加法器电路必须加速进位信号的产生。第 $i$ 步的 $s_i$（和）与 $c_{i+1}$（进位输出）的逻辑表达式为（参见图 9-1）

$$s_i = x_i \oplus y_i \oplus c_i$$

和

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

将第二个等式因式分解为

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

因此我们可以写出

$$c_{i+1} = G_i + P_i c_i$$

其中

$$G_i = x_i y_i \text{ 以及 } P_i = x_i + y_i$$

表达式 $G_i$ 和 $P_i$ 分别称为第 $i$ 步的生成（generate）函数与传播（propagate）函数。如果第 $i$ 步的生成函数等于 1，则 $c_{i+1} = 1$，而与进位输入 $c_i$ 无关。这种情况发生在 $x_i$ 与 $y_i$ 都为 1 时。传播函数意味着当 $x_i$ 或者 $y_i$ 其中之一为 1 时，进位输入就会产生进位输出。所有的 $G_i$ 与 $P_i$ 函数都可以在 $X$ 与 $Y$ 操作数加载到 $n$ 位加法器输入端后独立并行地经一个逻辑门延迟生成。每一位段包括一个与门以生成 $G_i$，一个或门以生成 $P_i$，以及一个三输入端的异或门以生成 $s_i$。仔细观察我们发现使用 $P_i = x_i \oplus y_i$ 足以实现传播函数，这样会得出更简单的电路。这个公式只在 $x_i = y_i = 1$ 时与 $P_i = x_i + y_i$ 不同。但这时 $G_i = 1$，所以 $P_i$ 是 0 是 1 无关紧要。于是，可以使用两个两输入端异或门的级联来实现 $s_i$ 的三输入端异或功能，图 9-4a 中基本的 B 单元可以用于每一位段的计算。

使用以 $i-1$ 为下标的变量扩展 $c_i$，并将其代入 $c_{i+1}$ 表达式中，我们得到

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

继续这种扩展，任一进位变量的最终表达式为

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i P_{i-1} \cdots P_0 c_0 \qquad (9\text{-}1)$$

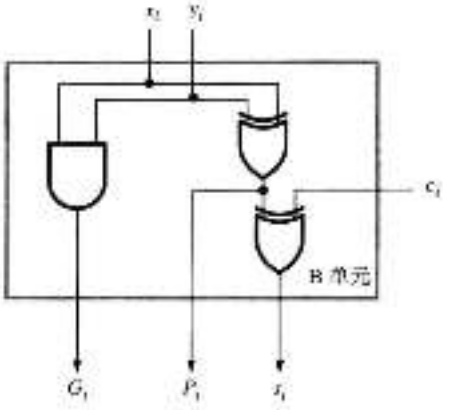
这样，所有的进位都可以在加载输入操作数 $X$、$Y$ 及 $c_0$ 三个门延迟后得到，因为生成所有 $P_i$ 与 $G_i$ 信号只需要一个门延迟，再加上生成 $c_{i+1}$ 的与 - 或电路中的两个门延迟。再经过一个异或门延迟，将得到所有的求和位。因此，$n$ 位加法过程总共只需要四个门延迟，而与 $n$ 无关。 340

我们现在考虑一下 4 位加法器的设计。其进位可以实现为

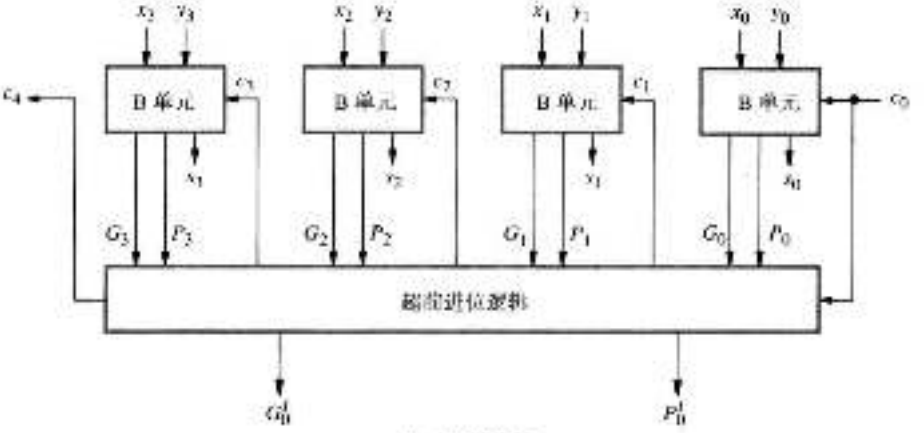
$$c_1 = G_0 + P_0 c_0$$

$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$
$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$




a）位段单元




b）4位加法器

341

图 9-4 4 位超前进位加法器

图 9-4b 显示了完整的 4 位加法器。进位在标记为超前进位逻辑的部件中产生。以这种形式实现的加法器被称为超前进位加法器（carry-lookahead adder）。该加法器对所有的进位位来说有 3 个门延迟，对所有的求和位来说有 4 个门延迟。对比一下 4 位行波进位加法器，$s_3$ 需要 7 个门延迟，而 $c_4$ 需要 8 个门延迟。

如果想要扩展图 9-4b 的超前进位加法器设计，使它能够对更长的操作数进行加法运算，就会遇到门的扇入（fan-in，即输入端数目）限制问题。由表达式 9-1，我们看到最后的与门和或门生成 $c_{i+1}$ 时需要的扇入为 $i+2$。在 4 位加法器中，生成 $c_4$ 需要的扇入为 5。这已经接近实际门的极限了。所以图 9-4b 中的加法器设计不能简单地进行扩展以处理长操作数。但是，可以将几个 4 位加法器级联起来构造较长的加法器，如图 9-2c 所示。

8 个 4 位超前进位加法器可以如图 9-2c 那样连接起来构成一个 32 位加法器。级联中高端的 4 位加法器生成求和位 $s_{31}$、$s_{30}$、$s_{29}$、$s_{28}$ 以及进位位 $c_{32}$ 所需的延迟计算如下：低端加法器的进位输出 $c_4$ 在输入操作数 $X$、$Y$ 及 $c_0$ 加载到 32 位加法器 3 个门延迟后获得。之后，第二个

加法器的输出 $c_4$ 再经过 2 个门延迟后获得，$c_{12}$ 需要再经过 2 个门延迟，依次类推。最后，高端 4 位加法器的进位输入 $c_{16}$ 可以在 $(6 \times 2) + 3 = 15$ 个门延迟后获得。这样，$c_{32}$ 和高端加法器内的所有进位还需要 2 个门延迟后才能得到，而 4 个求和位需要再经历 1 个门延迟才能获得，一共是 18 个门延迟。作为对比，如果使用行波进位加法器，$s_{31}$ 与 $c_{32}$ 的总延迟数分别为 63 和 64。

下面将对刚才讨论的级联结构加以改进，从而进一步减少延迟。关键的思想是并行地生成进位 $c_4$、$c_8$ 等，就像在 4 位超前进位加法器中并行地生成 $c_1$、$c_2$、$c_3$ 和 $c_4$ 一样。

**高层的生成与传播函数**

在刚才讨论的 32 位加法器中，进位 $c_4$、$c_8$、$c_{12}$ …行波传递进入各个 4 位加法器部件，每个部件两个门延迟。这与行波进位加法器中单个进位行波传递进入每个位段的方式非常相像。通过使用高层的生成与传播函数，就有可能使用超前进位方法并行地生成进位 $c_4$、$c_8$、$c_{12}$、…。

图 9-5 显示了由四个 4 位加法器部件构成的 16 位加法器。这些部件提供了新的输出函数 $G_k'$ 与 $P_k'$，其中 $k = 0$ 对应于第一个 4 位部件，$k = 1$ 对应于第二个 4 位加法器部件，依次类推，如图 9-4b 和图 9-5 所示。在第一个部件中，

$$P_3' = P_3P_2P_1P_0$$

和

$$G_3' = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

第一层的 $G_i$ 与 $P_i$ 函数决定了位段 $i$ 是否生成或传播进位，而第二层的 $G_k'$ 与 $P_k'$ 函数决定了部件 $k$ 是否生成或传播进位。有了这些新函数，就没有必要等待进位行波传送进入 4 位加法器部件了。进位 $c_{16}$ 由图 9-5 所示的超前进位电路按以下公式生成： [342]

$$c_{16} = G_3' + P_3'G_2' + P_3'P_2'G_1' + P_3'P_2'P_1'G_0' + P_3'P_2'P_1'P_0'c_0$$

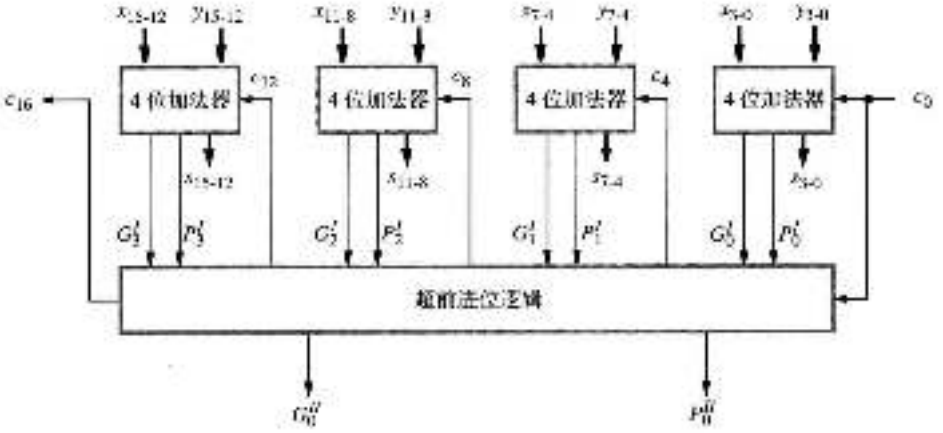



图 9-5 由 4 位加法器（见图 9-4b）构造的 16 位超前进位加法器

4 位部件的进位输入也由相似的简单表达式并行地生成。$c_{16}$、$c_{12}$、$c_8$ 及 $c_4$ 的表达式形式分别与图 9-4b 超前进位电路所实现的 $c_4$、$c_3$、$c_2$ 及 $c_1$ 相同，只是变量名不同。因此，图 9-5 中超前进位电路的结构与图 9-4b 是完全相同的。但是，由 4 位加法器部件内部产生的进位 $c_4$、$c_8$、$c_{12}$、$c_{16}$ 在图 9-5 中并不需要，因为在这里它们是由高层的超前进位电路生成的。

现在，我们考虑 16 位超前进位加法器产生输出所需的延迟。超前进位电路产生进位所需

的延迟比生成 $G_i'$ 和 $P_i'$ 函数多两个门延迟。而生成 $G_i'$ 和 $P_i'$ 在生成 $G_i$ 和 $P_i$ 之后分别需要两个和一个门延迟。因此，超前进位电路在 $X$、$Y$ 及 $c_0$ 加载到输入端五个门延迟后即可得到所有的进位。进位 $c_{15}$ 是在图 9-5 中高端的 4 位加法器部件内部产生的，它比 $c_{32}$ 晚两个门延迟，之后 $s_{15}$ 又晚了一个门延迟。因此，$s_{15}$ 将在八个门延迟后得到。如果采用级联 4 位超前进位加法器部件的方法构造 16 位加法器，生成 $c_{16}$ 和 $s_{15}$ 的延迟分别为九个和十个门延迟，而在图 9-5 中只需要五个和八个门延迟。

两个 16 位加法器部件可以级联构成一个 32 位加法器。这时，低端部件的输出 $c_{16}$ 成为高端部件的进位输入。其延迟要比以前讨论过的 32 位加法器少得多，那时我们通过级联八个 4 位加法器来构造 32 位加法器。回顾一下，在该加法器中，$s_{31}$ 在 18 个门延迟后得到，$c_{32}$ 在 17
[343] 个门延迟后得到。而级联两个 16 位加法器的延迟分析如下：我们刚刚讨论过，低端部件的进位输出 $c_{16}$ 在五个门延迟后得到。然后，高端部件的 $c_{28}$ 和 $c_{32}$ 在随后的两个门延迟后得到，$c_{31}$ 在 $c_{28}$ 之后的两个门延迟后得到。因此 $c_{31}$ 在总共九个门延迟后得到，$s_{31}$ 在十个门延迟后得到。也就是说，$s_{31}$ 和 $c_{32}$ 分别在十个和七个门延迟后获得。作为比较，如果使用八个 4 位加法器级联构成的 32 位加法器，获得相同的结果则分别需要 18 和 17 个门延迟。

从第一层的 $G_i$ 和 $P_i$ 函数生成第二层的 $G_i'$ 和 $P_i'$ 函数所用的推理可以用来从 $G_i'$ 与 $P_i'$ 函数生成第三层的 $G_i''$ 和 $P_i''$ 函数。这两个第三层的函数在图 9-5 中被表示为超前进位逻辑的输出。使用四个图 9-5 中的 16 位加法器，再加上产生进位 $c_{16}$、$c_{32}$、$c_{48}$ 和 $c_{64}$ 的超前进位逻辑电路，就可以构成一个 64 位加法器。使用上面对 16 位加法器推理的扩展，可以证明这个加法器的延迟为 $s_{63}$ 需要 12 个门延迟，$c_{64}$ 需要 7 个门延迟。（参见习题 9.7。）

## 9.3 无符号数乘法

图 9-6a 显示了二进制系统中手工进行整数乘法的普通算法。两个 $n$ 位无符号数的积小于等于 $2n$ 位，因此如图所示，本例中两个 4 位数的积为 8 位。在二进制系统中，被乘数与乘数的一位相乘是很容易的。如果乘数位为 1，则在适当移位的位置上放入被乘数。如果乘数位为 0，则放入 0，如示例的第三行。然后从右至左将各位列相加，每次计算一位，并在各位列之间传递进位值，从而得到乘积。

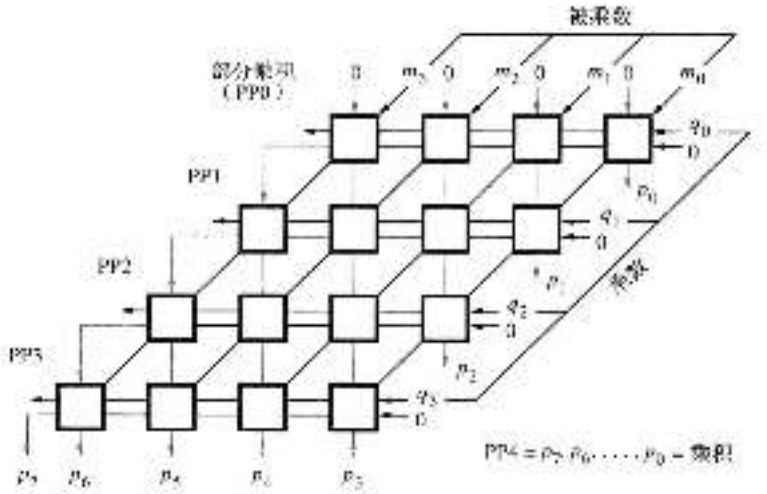
### 9.3.1 阵列乘法器

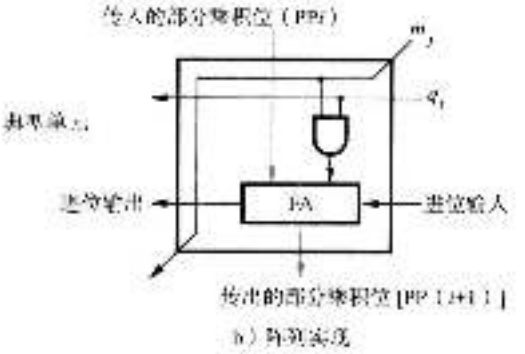
无符号操作数的二进制乘法可以用二维组合逻辑阵列实现，4 位操作数的乘法如图 9-6b 所示。每个单元的主要组成部分是一个全加器 FA。每个单元的与门根据乘数位 $q_i$ 的值决定被乘数位 $m_j$ 是否要加到传入的部分乘积位上。如果 $q_i = 1$，每一行 $i$（其中 $0 \leq i \leq 3$）将（经过适当移位的）被乘数加到传入的部分乘积 PP$i$ 上，以生成传出的部分乘积 PP$(i+1)$。如果 $q_i = 0$，则 PP$i$ 就不经改变地垂直向下传出。PP0 为全 0，而 PP4 为乘积结果。被乘数沿着倾斜的信号传输路径每行左移一个位置。我们注意到，不同于前面描述的逐列进行的普通手工加法，阵列电路中的加法是逐行进行的。

最坏情况下的信号传输延迟路径是从阵列的右上角到阵列左下角的高位乘积位。这条关
[344] 键路径包括每行右端的两个单元和最底行的所有单元，呈现出阶梯状模式。假设从全加器部件 FA 的输入到输出需要两个门延迟，则对 $n \times n$ 阵列而言，该关键路径的总延迟为 $6(n-1)+1$ 个门延迟，包括所有单元中初始的与门延迟。（参见习题 9.8）阵列的第一行不需要全加器，因为传入的部分积 PP0 为 0。在推导延迟表达式时已经考虑了这一点。

```
          1  1  0  1      (13) 被乘数 M
      ×   1  0  1  1      (11) 乘数 Q
    ---------------
          1  1  0  1
       1  1  0  1
    0  0  0  0
 1  1  0  1
---------------------
1  0  0  0  1  1  1  1      (143) 乘积 P
```

a）手工乘法运算




b）阵列实现

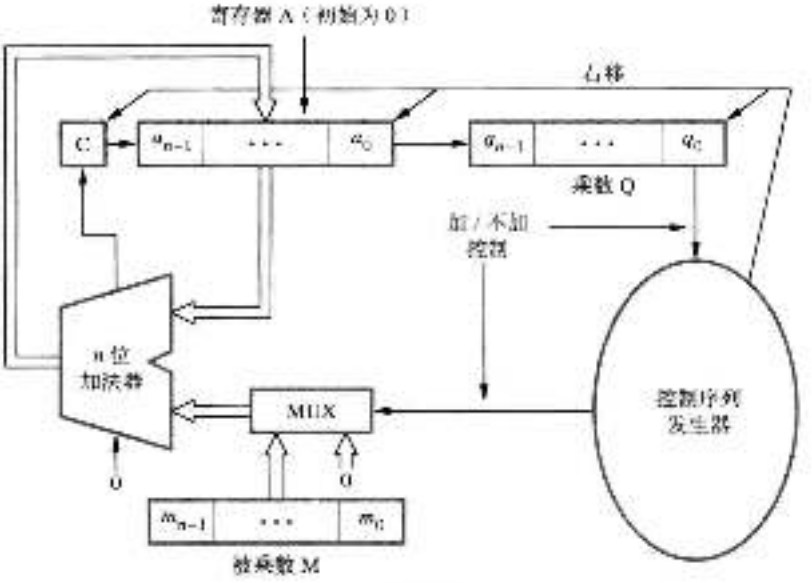
图 9-6 无符号二进制操作数的阵列乘法

## 9.3.2 顺序电路乘法器

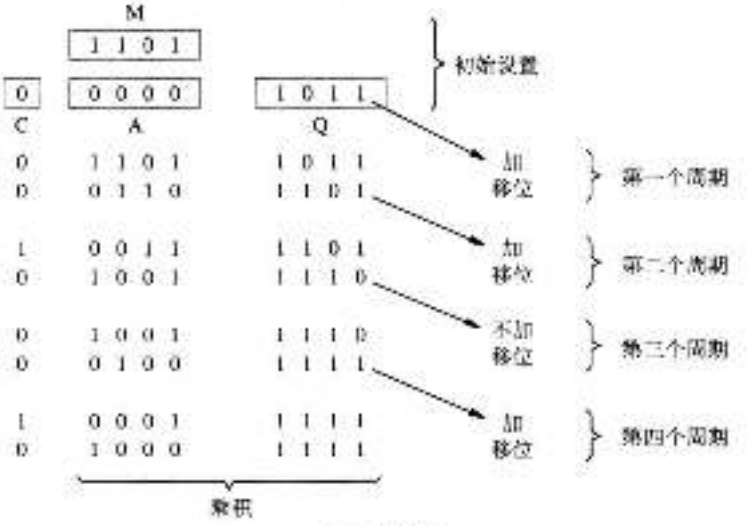
刚刚所描述的组合阵列乘法器在计算实际大小的数字（如 32 位或 64 位数字）时，使用了很多的逻辑门。两个 $n$ 位数的乘法还可以在一个使用单个 $n$ 位加法器的顺序电路中实现。

图 9-7a 的框图显示了顺序乘法电路的硬件组成。该电路通过使用一个 $n$ 位加法器进行 $n$ 次运算来实现图 9-6b 中 $n$ 行行波进位加法器所执行的空间加法。寄存器 A 和 Q 是移位寄存器，如图中所示连接起来，它们一起容纳部分积 PP$i$，而乘数位 $q_0$ 产生“加 / 不加”信号。该信号使得多路复用器 MUX 选择 0（当 $q_0$=0 时）或被乘数 M（当 $q_0$=1 时）加到 PP$i$ 上，从而生成 PP$(i+1)$。乘积需要 $n$ 个周期计算。从初始向量 PP0（寄存器 A 中的 $n$ 个 0）开始，在每一周期中部分积 PP$i$ 的长度都增长一位。加法器的进位输出存储在触发器 C 中，它在寄存器

A 的左侧。开始时，乘数被装载到寄存器 Q 中，被乘数被装载到寄存器 M 中，而 C 和 A 被清零。在每一周期结束时，C、A 和 Q 右移一位以容纳部分积的增长，同时乘数移出寄存器 Q。由于这种移位，寄存器 Q 的最低有效位（LSB）位置上的乘数位 $q_i$ 才可以在正确的时刻生成“加 / 不加”信号，比如第一个周期的 $q_0$ 到第二个周期的 $q_1$ 等等。这些乘数位使用过后就被右移操作丢弃了。注意，加法器的进位输出是 FF $(i+1)$ 的最左位，它必须保存在触发器 C 中，并随 A 和 Q 的内容一起右移。$n$ 个周期之后，乘积的高位部分保存在寄存器 A 中，而低位部分保存在寄存器 Q 中。图 9-7b 显示了以这种硬件安排执行图 9-6a 中的乘法示例。




a）寄存器配置




b）乘法示例

图 9-7 二进制顺序乘法电路

## 9.4 有符号数乘法

我们现在讨论补码操作数的乘法，它生成长度加倍的乘积。总的策略依然是由乘数位选择被乘数累加部分积的方法。

首先，考虑正乘数和负被乘数的情况。当我们将负的被乘数加到部分积时，必须将被乘数符号位的值扩展到乘积的最左边。图 9-8 给出了一个例子，其中 5 位有符号操作数 −13 为被乘数，乘以 +11 得到了 10 位的乘积 −143。被乘数的符号扩展用灰色显示。如果前面讨论过的硬件能被扩大以提供部分积的符号扩展，那么它就可以接受负的被乘数。

345 / 347

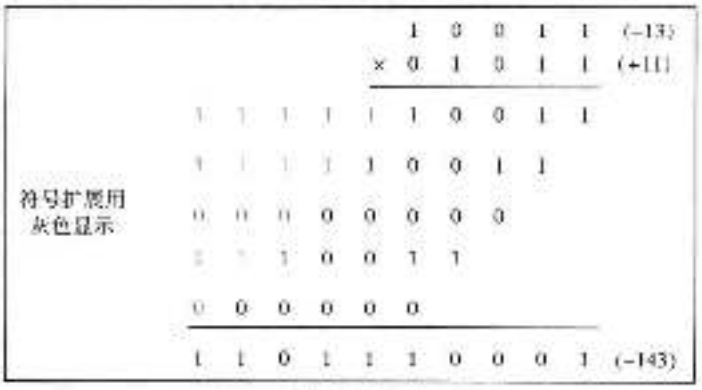



图 9-8 负被乘数的符号扩展

对于负乘数，直接的解决办法就是形成乘数与被乘数的补码，并按照正乘数的情况进行运算。可以这样做是因为对两个操作数都求补并不改变乘积的值或符号。下面我们将讨论一种对负乘数和正乘数效果同样好的技术，称为 Booth 算法。

### Booth 算法

Booth 算法 [1] 生成 $2n$ 位的乘积，而且它以统一的方式对待正负补码形式的 $n$ 位操作数。为了理解该算法的基本原理，考虑一个乘法运算，其中乘数为正数而且有唯一的一块区域为全 1，例如 0011110。为了得到乘积，在标准程序中需要将四个适当移位的被乘数相加。但是，通过将该乘数看作是两个数的差可以减少所需操作的数目：

$$\begin{array}{rll} & 0100000 & (32) \\ - & 0000010 & (2) \\ \hline & 0011110 & (30) \end{array}$$

这意味着将被乘数的 $2^5$ 倍与 $2^1$ 倍的补码相加即可得到乘积。为了方便，我们可以将前面的乘数重新编码为 0 +1000 −10，并以此描述所需操作的顺序。

一般而言，在 Booth 算法中，从右至左扫描乘数，由 0 变为 1 时将移位的被乘数乘以 −1，由 1 变为 0 时将移位的被乘数乘以 +1。图 9-9 演示了刚才所讨论例子的普通算法与 Booth 算法。很明显，Booth 算法可以扩展到包含任意块数 1 的乘数，包括将单独的一个 1 视为一块的情况。图 9-10 展示了对乘数进行重编码的另一个例子。当乘数的最低有效位为 1 时，假设其右方隐含地包括一个 0。Booth 算法还可以直接应用于负乘数的情况，如图 9-11 所示。

348

为了证明 Booth 算法对负乘数的正确性，我们使用补码系统中负数表示的以下性质：假设负数 $X$ 中最左方的 0 在位位置 $k$ 处，即

$$X = 11\cdots10x_{k-1}\cdots x_0$$

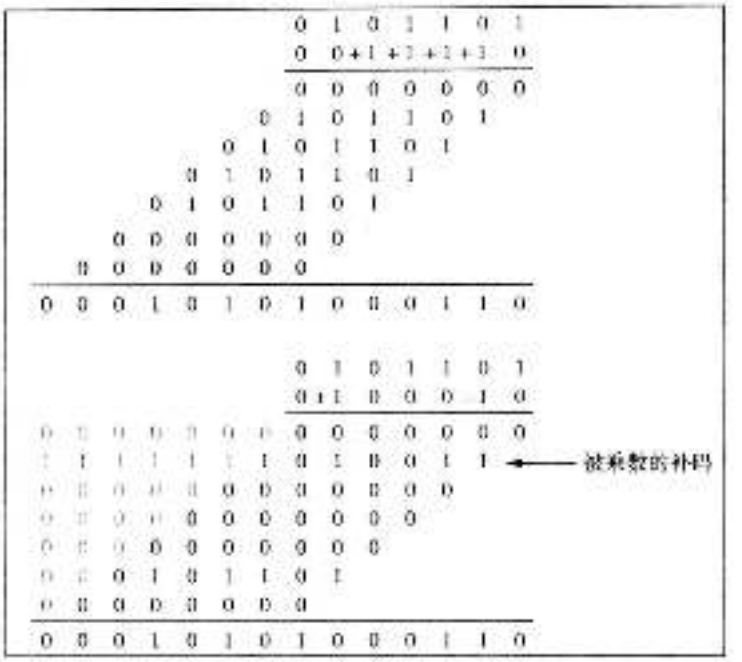



图 9-9 普通乘法与 Booth 乘法方案

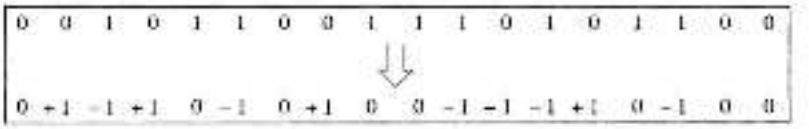


图 9-10 乘数的 Booth 重编码

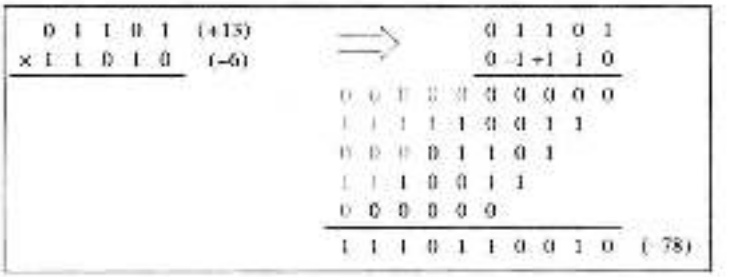


图 9-11 负乘数的 Booth 乘法

则 $X$ 的值为

$$V(X) = -2^{k+1} + x_{k-1} \times 2^{k-1} + \cdots + x_0 \times 2^0$$

$V(X)$ 表达式的正确性可以这样证明，即将 $X$ 看作以下两个数之和：

$$\begin{array}{rl} & 11\cdots100000\cdots0 \\ + & 00\cdots00x_{k-1}\cdots x_0 \\ \hline X = & 11\cdots10x_{k-1}\cdots x_0 \end{array}$$

其中上面的数是 $-2^{k+1}$ 的补码表示。现在，重编码的乘数的一部分就是和式中下面的数，并且其 $k+1$ 位为 $-1$。例如，乘数 110110 可以重编码为 0−1+10−10。

图 9-12 总结了对乘数重编码的 Booth 技术。由 011…110 至 +100…0−10 的转换称为规过

1 位（skipping over 1s）。这个术语是由于将乘数分解成几个连续全 1 的块区域的情况得来的。只需将被乘数（求和项）几种移位的变体相加就可得到乘积，因此加速了乘法运算。但是，在最坏情况下，即乘数中的 1 和 0 交替时，乘数的每一位都将选择一个求和项。实际上，它产生的求和项比不使用 Booth 算法时还要多。图 9-13 显示了最坏情况、普通情况和较好情况下的 16 位的乘数。

| 乘数 | | 第 $i$ 位选择的被乘数变体 |
|---|---|---|
| 第 $i$ 位 | 第 $i-1$ 位 | |
| 0 | 0 | $0\times M$ |
| 0 | 1 | $+1\times M$ |
| 1 | 0 | $-1\times M$ |
| 1 | 1 | $0\times M$ |

图 9-12 Booth 乘数重编码表

Booth 算法有两个吸引人的特征。第一，它以统一的方式处理正负乘数；第二，当乘数中包含很多大块的全 1 区域时，它的效率很高。

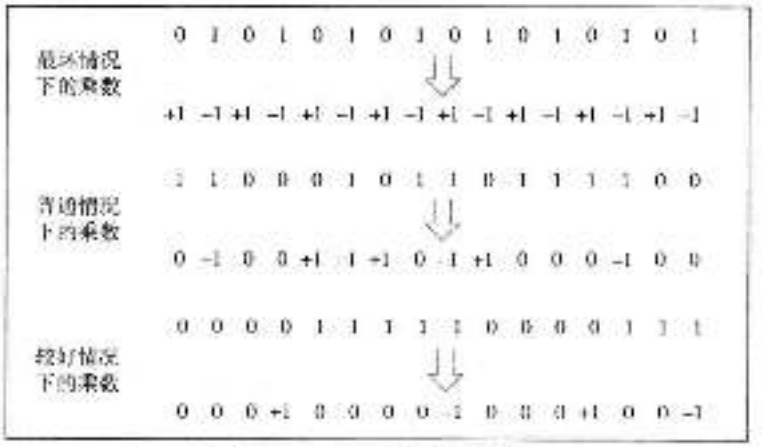



图 9-13 Booth 重编码的乘数

## 9.5 快速乘法

现在我们讲述两种提高乘法运算速度的技术。第一种技术保证对于 $n$ 位操作数，参与加法的求和项（被乘数的不同变体）的最大数目为 $n/2$。第二种技术是将求和项并行相加。

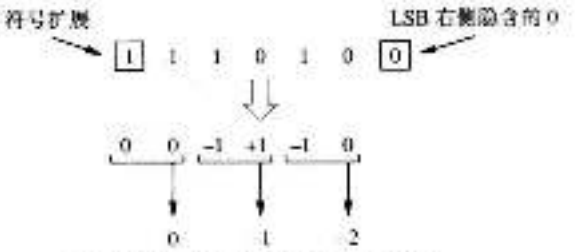
349 ↓ 351

### 9.5.1 乘数位偶重编码

乘数位偶重编码（bit-pair recoding）技术可将每对乘数位用最多一个求和项来表示。它直接来源于 Booth 算法。将 Booth 算法重编码的乘数位组成对时，可以观察到：对 (+1 −1) 与对 (0 +1) 是等价的。也就是说，不是将移位位置 $i$ 上 −1 倍的被乘数 M 与位置 $i+1$ 上 +1 倍的 M 相加，而是在位置 $i$ 加入 $+1\times M$，也可以得到相同的结果。其他的例子有：(−1 0) 与 (0 +2) 等价，(−1 +1) 与 (0 −1) 等价，等等。这样，如果从右边开始对 Booth 算法重编码过的乘数每次检查两位，那么乘数可以重写为另一种形式，该形式对于每一对乘数位最多只需要将一个被乘数加到部分积中。图 9-14a 显示了对图 9-11 中乘数的位偶重编码。而图 9-14b 显示了在所有可能的情况下被乘数的选择决策表。图 9-15 使用乘数的位偶重编码重新计算了图 9-11 中的乘法。

### 9.5.2 求和项的进位保留加法

在乘法中需要将几个求和项相加，一种称为进位保留加法（carry-save addition，CSA）的技术可以用来加速这一过程。考虑图 9-16a 中的 4×4 乘法阵列。这个结构使用的是图 9-6 所示的阵列形式，它的第一行只包含产生四个输入 $m_3q_0$、$m_2q_0$、$m_1q_0$ 和 $m_0q_0$ 的与门。




a) 基于 Booth 重编码的位组重编码示例

| 乘数位组 | | 右侧乘数位 | 位置 $i$ 选择的被乘数 |
|---|---|---|---|
| $i+1$ | $i$ | $i-1$ | |
| 0 | 0 | 0 | $0\times M$ |
| 0 | 0 | 1 | $+1\times M$ |
| 0 | 1 | 0 | $+1\times M$ |
| 0 | 1 | 1 | $+2\times M$ |
| 1 | 0 | 0 | $-2\times M$ |
| 1 | 0 | 1 | $-1\times M$ |
| 1 | 1 | 0 | $-1\times M$ |
| 1 | 1 | 1 | $0\times M$ |

b) 被乘数选择决策表

图 9-14 乘数位组重编码

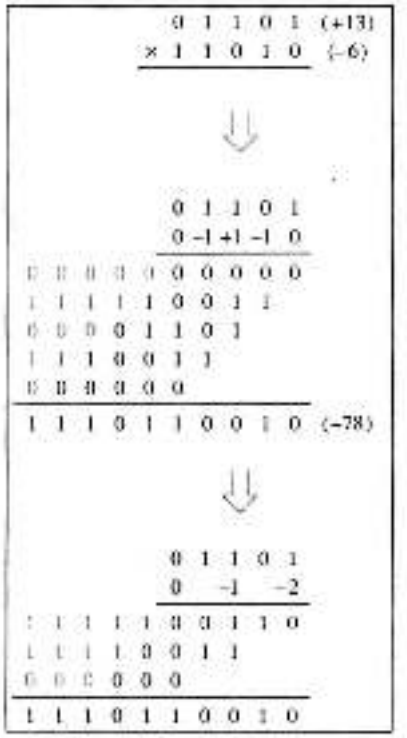
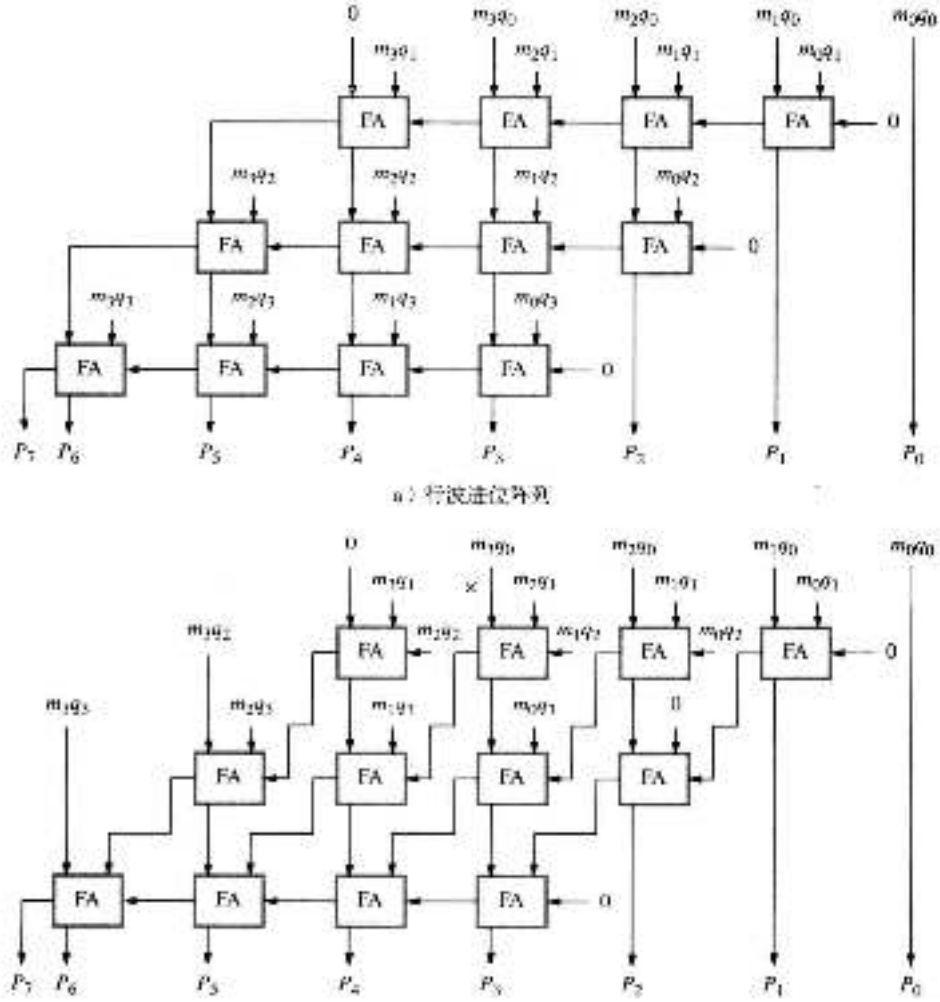



图 9-15 只需 $n/2$ 个求和项的乘法

我们可以不让进位在各行中波动，而是将它们“保存”起来，并在正确的权重位置上将它们引入到下一行中，如图 9-16b 所示。这释放了第一行前三个全加器中每个全加器的一个输入端。这些输入端用来引入第三个求和项的位 $m_2q_2$、$m_1q_2$、$m_0q_2$。现在，第二行前二个全加器中每个全加器的两个输入端装载了来自第一行的和与进位输出。第三个输入端用来引入第四个求和项的位 $m_3q_3$、$m_2q_3$、$m_1q_3$。第三和第四个求和项的高端位 $m_3q_2$ 与 $m_3q_3$ 被引入到第二与第三行左侧其余空闲的全加器输入端中。来自第二行的保留进位位及求和位现在加到第三行的行波进位加法器中，从而得到最终的乘积位。

352 ~ 353

进位保留阵列的延迟比行波进位阵列的延迟要小一些。这是因为每一行的输出向量 $S$ 和 $C$ 是在一个全加器延迟中并行生成的。延迟的减少量将在习题 9.15 中考虑。




a）行波进位阵列

b）进位保留阵列

图 9-16 一个 4×4 乘法器的行波进位与进位保留阵列

### 9.5.3 使用 3-2 简化器的求和项加法树

当处理比图 9-16 中所考虑的操作数更长的操作数时，有可以更显著地减小延迟的方法。我们可以将三个求和项分为一组，并对每个组并行地执行进位保留加法，从而在一个全加器延

迟中得到一系列 $S$ 与 $C$ 向量。这里我们将全加器电路简称为加法器。然后将所有的 $S$ 和 $C$ 向量分为三个一组，对它们执行进位保留加法，在一个加法器延迟中生成更深一层的 $S$ 与 $C$ 向量。我们继续这个过程，直到只剩下两个向量。三个求和项中每个位位置上的加法器被称为 3-2 简化器（3-2 reducer），将求和项数目减为 2 的逻辑电路结构被称为 CSA 树（CSA tree），如 Wallace 在参考文献 [2] 中所描述的。最后两个 $S$ 和 $C$ 向量可以使用超前进位加法器相加，产生最终的乘积。

354
↓
355

考虑图 9-17 所示的例子，在两个 6 位的无符号数相乘且乘数的全部位均为 1 时，对六个被乘数的移位变体求和。图 9-18 使用进位保留加法对六个求和项 $A$，$B$，…，$F$ 求和。两幅图中的“方框”指示了相同的操作数位，并显示了如何对它们使用进位保留加法生成求和位与进位位。示例中执行了三个层次的进位保留加法，如图 9-19 所示。图中显示了在六个输入求和项加载到第一层上之后经过三个加法器延迟得到最终的两个向量 $S_4$ 与 $C_4$。将 $S_4$ 与 $C_4$ 相加生成最终乘积的普通加法操作可以使用超前进位加法器来完成。

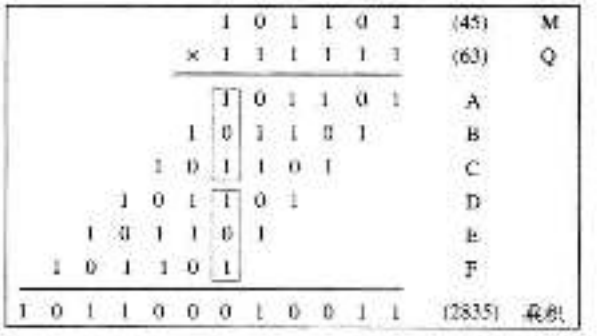



图 9-17 展示图 9-18 中进位保留加法的乘法示例

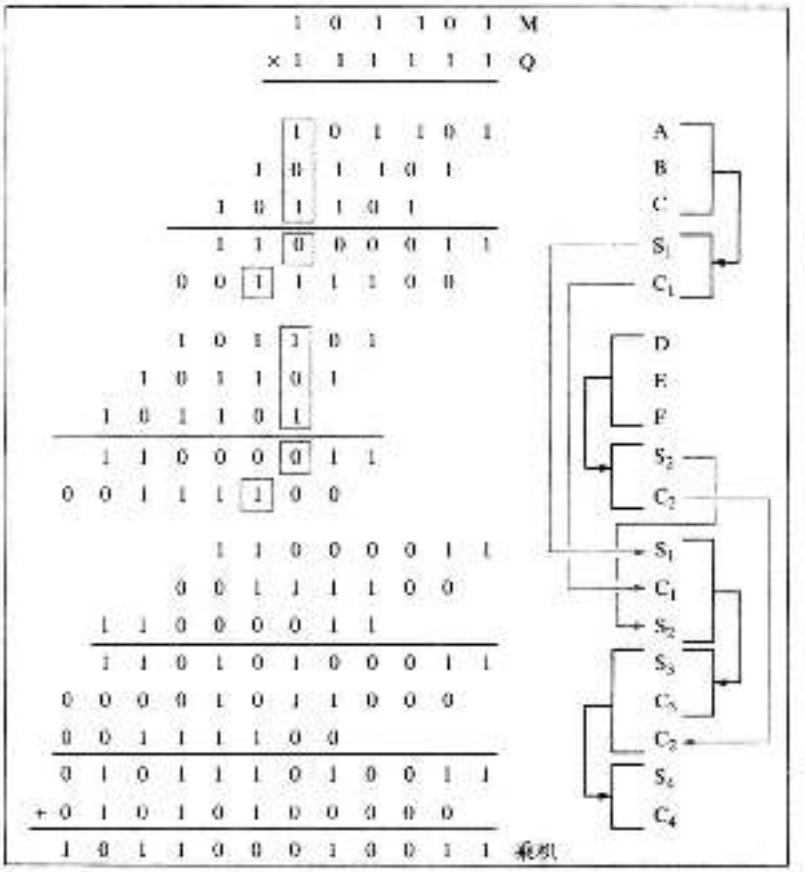



图 9-18 使用进位保留加法计算图 9-17 中的乘法示例

使用图 9-19 所示的树结构时乘法器的延迟比使用图 9-16b 所示的阵列结构时要少。当求和项的数目很大时，延迟的减少是比较显著的。例如，按照图 9-19 的模式将 32 个求和项相加，在最终的加法操作前只需要八层的 3-2 简化操作。一般而言，可以证明将 $k$ 个求和项降为最终求和以生成乘积的两个向量大约需要 $1.7\log_2 k - 1.7$ 层的 3-2 简化操作（参见 9.10 节中的例 9.3）。

我们应该注意，当使用有符号数乘法以及乘数的 Booth 重编码技术时，可能会涉及负的求和项，这就需要在求和项进入简化树之前对它们进行符号扩展。同时，如果使用了乘数的位偶重编码技术，那么需要相加的求和项的数目将会减少。

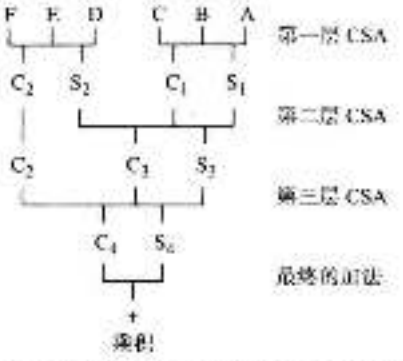



图 9-19 图 9-18 中进位保留加法操作的图示

3-2 简化器不是构建简化树的唯一逻辑电路，还可以使用 4-2 简化器和 7-3 简化器。下一小节将描述 4-2 简化器，7-3 简化器将在习题 9.17 中探讨。

### 9.5.4 使用 4-2 简化器的求和项加法树

从图 9-19 中我们可以发现，使用 3-2 简化器的 CSA 树，其各层间的互连模式是不规则的。通过使用 4-2 简化器[3]，我们可以得到一个结构更加规则的树，当求和项的数目是 2 的幂时则更是如此。处理器的 ALU 在进行乘法运算时通常就是采用这种方式。例如，如果在每个简化层上使用 4-2 简化器将 32 个求和项缩减到 2 个，那么只需要四层即可。树的结构很规则，四层的输出依次为 16、8、4 和 2 个求和项。而如果使用 3-2 简化器，则需要八层，并且各层间的线路连接都很不规则。在实现 VLSI 电路时，规则的树结构可以使逻辑电路和线路的布局更加便利。

356 ↓ 357

让我们来考虑一下在参考文献[3]中提出的 4-2 简化器的设计。对四个求和项中四个相同权重的位 $w$、$x$、$y$ 和 $z$ 进行加法计算，会产生一个在 0 到 4 范围内的值。这个值无法用一个求和位 $s$ 和一个单独的进位位 $c$ 来表示。然而，我们可以使用第二个进位 $c_{out}$（与 $c$ 的权重相同），与 $s$ 和 $c$ 一起来表示 0 到 5 范围内的任意值。而这可以满足我们这里的要求。

我们不希望向下一个简化层发送三个输出位，那样就是 4-3 简化器了，它将比 3-2 简化器提供更少的简化。解决方法是将 $c_{out}$ 横向发送给同一简化层中权重较高的相邻位位置上的 4-2 简化器。这样，每一个 4-2 简化器必须有第五个输入 $c_{in}$，它是同一简化层中权重较低的相邻位位置上的 4-2 简化器的 $c_{out}$ 输出。

设计 4-2 简化器的最后一个要求是 $c_{out}$ 的值不能依赖于 $c_{in}$ 的值。这是一个关键的要求，如果没有这个要求，进位将会在一个简化层中横向波动，这样就完全背离了在一个较短的固定延迟时间内并行地简化求和项的目的。4-2 简化器部件如图 9-20 所示。

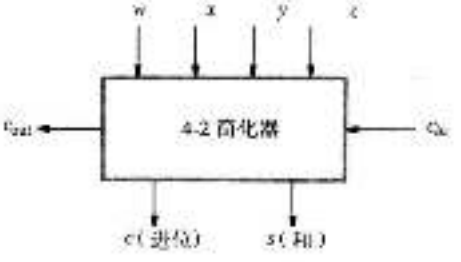



图 9-20 4-2 简化器部件

综上所述，一个 4-2 简化器的具体设计要求如下：

- $s$、$c$ 和 $c_{out}$ 这三个输出代表了五个输入的算术和，即

$$w + x + y + z + c_{in} = s + 2(c + c_{out})$$

其中所有的运算符都是算术运算符。

- 输出 $s$ 是通常的和变量；也就是说，$s$ 是这五个输入变量的异或函数。
- 横向进位 $c_{out}$ 必须独立于 $c_{in}$。它只是 $w$、$x$、$y$ 和 $z$ 这四个输入变量的函数。

满足上述条件的两个进位输出有很多组，这里我们给出一组便于描述的进位输出。首先，当输入变量 $w$、$x$、$y$ 和 $z$ 中的两个或两个以上都等于1时，将横向的进位输出 $c_{out}$ 赋值为1。然后，为了满足算术条件，另一个进位输出 $c$ 也确定了。满足这些条件的完整真值表如图9-21所示。这个表的表示形式与附录A中使用的普通形式不同，其中，$w$、$x$、$y$ 和 $z$ 这四个输入不是按照二进制数字的顺序列出的，而是按组的方式列出，每组中值为1的输入个数相同。这样我们可以很容易地看出如何指定输出以满足给定的条件。根据该表，我们可以推导出一个逻辑门网络。

358

| | | | | $c_{in}=0$ | | $c_{in}=1$ | | |
|---|---|---|---|---|---|---|---|---|
| $w$ | $x$ | $y$ | $z$ | $c$ | $s$ | $c$ | $s$ | $c_{out}$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

图9-21 4-2简化器真值表

### 9.5.5 快速乘法总结

现在我们总结一下快速乘法技术。首先，由Booth算法发展而来的乘数位偶重编码技术可将求和项的数目减少为原来的1/2。然后，利用一个简化层数目相对较少的简化树中进一步将求和项的数目降为2。最终的乘积可以通过使用超前进位加法器进行一次加法运算得到。高性能处理器的设计者已经在各种不同的组合中使用了所有这三种技术——乘数的位偶重编码、求和项的并行简化和超前进位加法——来减少执行乘法操作所需的时间。

## 9.6 整数除法

在9.3节讨论无符号数乘法时，我们将手工计算和逻辑电路计算乘法操作的方法联系了起来。在这里将使用同样的方法讨论整数除法。我们将详细地讨论无符号数除法，然后再对有符号操作数的情况作一般性的说明。

```
      21                          10101
13 ) 274                  1101 ) 100010010
     26                          1101
      14                          10000
      13                           1101
       1                             1110
                                     1101
                                        1
```

图9-22 笔算除法示例

图9-22显示了同一数值的十进制除法与二进制除法的示例。先来看一下十进制除法的情况。商中的2是由以下推理确定的：首先，试着用13去除2，但这不行。然后，试着用13去除27。我们试探着将13乘以2得到26，而且27−26＝1小于13，所以上商2并进行所需的减法。被除数的下一位4被放下来，最后我们用13去除14，余数为1。也可以用类似的方式来讨论二进制除法，而且由于商中的各位只可能为0或1，所以二进制的除法运算更简单。

用这种笔算方法实现除法的电路操作如下：它先将除数与被除数适当地对齐并执行减法。如果余数为0或为正，则可确定商位为1，而余数用被除数的下一位进行扩展，除数被重新定位，然后再执行下一次减法。如果余数为负，则可确定商位为0，被除数加回除数以恢复原值，然后除数被重新定位，执行下一次减法。这被称为恢复除法（restoring division）算法。

### 1. 恢复除法

图 9-23 显示了实现刚刚讨论的恢复除法算法的逻辑电路装置。注意它与图 9-7 所示的乘法电路非常相似。操作开始时，$n$ 位正除数被加载到寄存器 M 中，而 $n$ 位正被除数被加载到寄存器 Q 中。寄存器 A 被清零。除法结束后，$n$ 位的商将存放在寄存器 Q 中，而余数则存放在寄存器 A 中。所需的减法操作可以使用补码运算方便地完成。A 和 M 左边的附加位位置在减法过程中存放符号位。下面的算法执行了恢复除法。

359 · 360

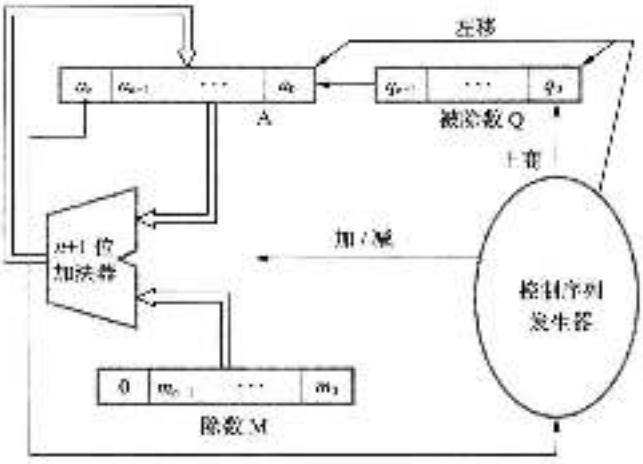



图 9-23 二进制除法的电路装置

执行以下三步 $n$ 次：

1）将 A 与 Q 左移一位。

2）从 A 中减去 M，并将结果放回 A。

3）如果 A 的符号为 1，上商 $q_0$ 为 0，并将 M 加回到 A 中（即恢复 A）；否则上商 $q_0$ 为 1。图 9-24 显示了使用图 9-23 中的电路进行 4 位除法运算的例子。

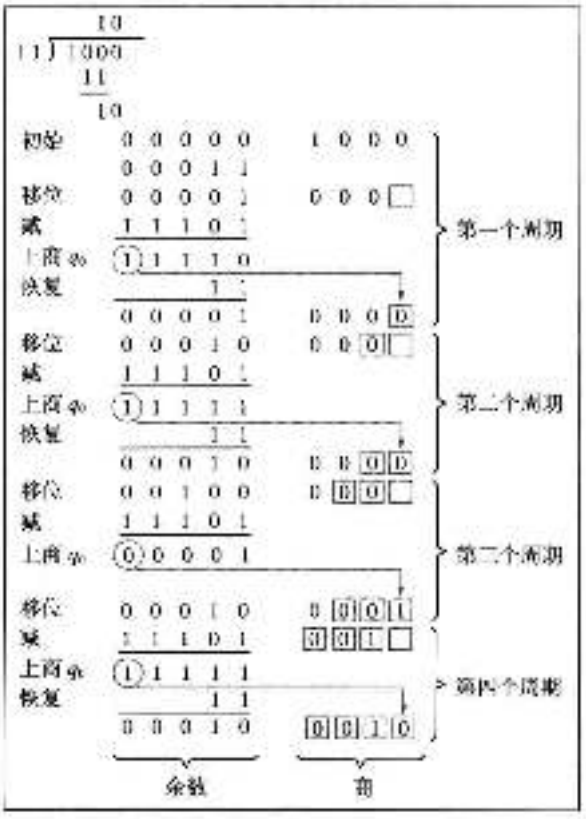



图 9-24 恢复除法示例

### 2. 不恢复除法

在不成功的减法之后避免恢复 A 的值可以提高恢复除法算法的效率。如果结果为负，则减法被说成是不成功的。考虑前面算法中减法操作之后所发生的操作步骤。如果 A 为正，左移并减去 M，也就是执行 2A−M。如果 A 为负，就执行 A + M 来恢复它，然后再左移并减去 M。这跟执行 2A − M 是等价的。正确的操作执行后，$q_0$ 位就被恰当地置为 0 或 1。我们可以用下面的算法对不恢复除法（non-restoring division）作一个总结。

第 1 阶段：执行以下两个步骤 $n$ 次。

1）如果 A 的符号为 0，将 A 和 Q 左

移一位，并用A减去M；否则，将A和Q左移，并将M加到A上。

2）现在，如果A的符号为0，上商 $q_0$ 为1；否则，上商 $q_0$ 为0。

第2阶段：如果A的符号为1，将M加到A上。

在执行第1阶段 $n$ 个周期之后，需要执行第2阶段以便在A中设置适当的正余数。图9-23的逻辑电路也可以用来执行这个算法，除了不再需要恢复操作之外。第1阶段的每个周期只执行一次加法或减法操作，第2阶段可能还要执行最后一次加法操作。图9-25显示了使用不恢复除法算法计算图9-24中示例的情况。

对有符号操作数进行除法运算并没有与有符号数乘法相对应的简单算法。在除法中，可以将操作数转变为正数。在使用了上面讨论的一种算法之后，如果有需要再将商和余数的符号进行调整。

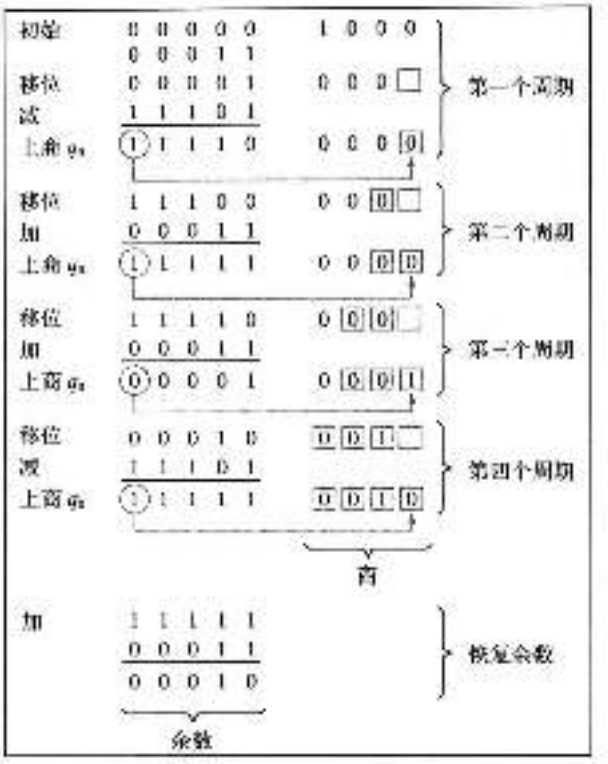



图9-25 不恢复除法示例

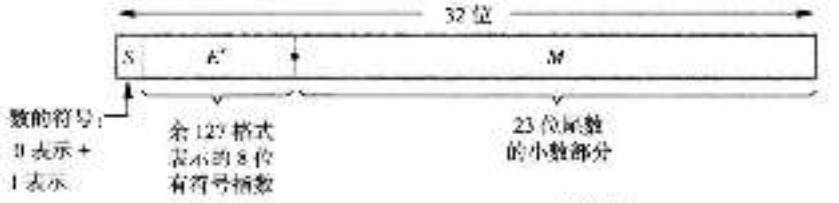
## 9.7 浮点数及其运算

361
|
363

第1章描述了使用浮点数的动机，并指出了如何将它们表示成32位的二进制格式。在本章中，我们将详细描述浮点数的表示格式以及对浮点数的算术运算。这里提供的这些描述是基于2008版本的IEEE（Institute of Electrical and Electronics Engineers，电气和电子工程师协会）754标准，标记为754-2008[4]。

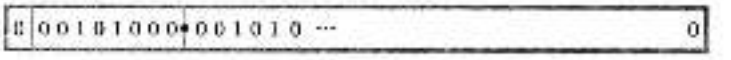
回顾一下第1章中曾描述过一个二进制浮点数可以表示为：

- 数的符号
- 一些有效位
- 有符号的比例因子指数（隐含的基数为2）




表示的数值 $= \pm 1.M \times 2^{E-127}$

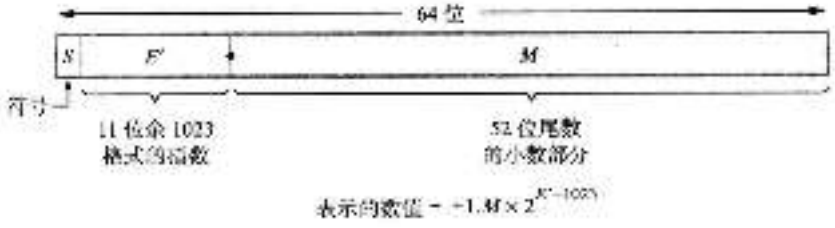
a）单精度




表示的数值 $= 1.001010 \cdots 0 \times 2^{-87}$

b）单精度数示例

图9-26 IEEE标准浮点数格式




c）双精度

图 9-26 （续）

基本的 IEEE 格式是一种 32 位表示，如图 9-26a 所示。最左边的位表示数的符号 $S$。接下来的 8 位 $E'$ 表示比例因子的有符号指数（有一个隐含的基数 2），剩下的 23 位 $M$ 是有效位的小数部分。有效位的整个 24 位串 $B$ 被称为尾数（mantissa），其前导位始终为 1，并有一个二进制小数点紧靠在其右边。因此，尾数

$$B = 1.M = 1.b_{-1}b_{-2}\cdots b_{-23}$$

的值为

$$V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-23} \times 2^{-23}$$

习惯上，当二进制小数点位于第一个有效位的右方时，我们说这个数是规格化（normalized）的。注意比例因子的基数 2 以及尾数的前导位 1 都是固定的，它们不需要在浮点数的表示中明确地出现。

指数域中存储的值并不是实际的有符号指数 $E$，而是一个无符号整数 $E' = E + 127$。这被称为余 127（excess-127）格式。这样，$E'$ 的取值范围为 $0 \leqslant E' \leqslant 255$。这个范围的端点值 0 和 255 用来表示特殊值，我们在后面会讲到。因此 $E'$ 的正常取值范围为 $1 \leqslant E' \leqslant 254$。这意味着实际指数 $E$ 的取值范围为 $-126 \leqslant E \leqslant 127$。指数使用余 127 表示简化了两个浮点数相对大小的比较。（参见习题 9.23。）

图 9-26a 中的 32 位标准表示称为单精度（single-precision）格式，因为它占用了一个 32 位的字。比例因子的范围为 $2^{-126}$ 到 $2^{+127}$，这大致接近于 $10^{\pm 38}$。24 位尾数所提供的精度与 7 位十进制值相仿。图 9-26b 中显示了一个单精度浮点数的例子。

为了提高浮点数的精度与取值范围，IEEE 标准还指定了双精度（double-precision）格式，如图 9-26c 所示。双精度格式增大了指数与尾数的取值范围。11 位余 1023 指数 $E'$ 的正常取值范围为 $1 \leqslant E' \leqslant 2046$，0 和 2047 用来指示特殊值。这样，实际指数 $E$ 的取值范围为 $-1022 \leqslant E \leqslant 1023$，它所提供的比例因子为 $2^{-1022}$ 到 $2^{1023}$（接近 $10^{\pm 308}$）。53 位尾数所提供的精度与 16 位十进制数字相当。

计算机必须至少提供单精度表示以符合 IEEE 标准。双精度表示是可选的。标准还规定了这两种表示的几种可选的扩展形式。这些扩展形式为计算过程中的中间值表示提供更高的精度和更大的指数范围。使用扩展格式有助于减小计算所需结果过程中累积的舍入误差。例如，两个向量的点积可以通过累加乘积的和得到。输入向量分量以标准精度表示，即单精度或双精度，最终的计算结果（点积）也截取为同样的精度。所有的中间计算都应该使用扩展的精度格式进行，以减少误差的累积。扩展格式还可以提高初等函数如正弦、余弦等的精确度。除了四种基本的算术运算，标准还要求提供余数、平方根以及二进制与十进制表示的转换这三种额外的运算。

请注意浮点数操作的两个基本方面。第一，如果一个数没有规格化，那么可以通过

移位二进制小数点和调整指数将它转化为规格化格式。图 9-27 显示了一个非规格化的数 $0.0010110\cdots\times 2^{5}$，以及它的规格化形式 $1.0110\cdots\times 2^{2}$。因为比例因子的形式为 $2^{i}$，将尾数向右或向左移动一位可以分别通过对指数加 1 或减 1 来补偿。第二，计算过程中可能会产生正常数字表示范围之外的数字。对于单精度，这意味着该数的规格化表示中的指数需要小于 −126 或者大于 −127。在第一种情况中，我们说发生了下溢（underflow），而在第二种情况中，我们说发生了上溢（overflow）。




图 9-27 IEEE 单精度格式浮点数规格化

1. 特殊值

余 127 指数 $E'$ 的端点值 0 和 255 被用来表示特殊值。当 $E'=0$ 且尾数的小数部分 $M$ 也为 0 时，表示的是值 0。当 $E'=255$ 且 $M=0$ 时，表示的是值 $\infty$，这里 $\infty$ 是用 0 去除一个正常数的结果。这些表示中仍然使用符号位，所以存在 +0 和 −0、+∞ 和 −∞ 的表示。

当 $E'=0$ 且 $M\neq 0$ 时，表示的是非规格化（denormal）数，其值为 $\pm 0.M\times 2^{-126}$。因此它们比最小的规格化数还要小。这时在二进制小数点的左方没有隐含的 1，而 $M$ 是任何非零的 23 位小数。引入非规格化数的目的是实现逐级下溢（gradual underflow），可以对正常表示的数值范围进行扩展，在处理某些场合下所需要的极小数值时非常有用。当 $E'=255$ 且 $M\neq 0$ 时，表示的值称为非数（Not a Number，NaN）。NaN 表示执行非法操作的结果，例如 0/0 或 $\sqrt{-1}$。

366

2. 异常

为了符合 IEEE 标准，执行操作时如果发生以下条件，处理器必须设置异常（exception）标志：下溢、上溢、除 0、不精确和非法。我们已经讲过前三种条件。不精确（inexact）描述的是结果需要进行舍入才能用规范格式表示的情况。非法（invalid）异常在试图执行 0/0 或 $\sqrt{-1}$ 操作时发生。当一个异常发生时，计算结果就被设置为一个特殊值。

如果允许对某个异常标志中断，当相关的异常发生时就会进入系统或用户定义的例程。或者，如果必要，应用程序可以检测异常的发生，并决定接下来需要做什么。

### 9.7.1 浮点数算术运算

本节我们将概述浮点数的加、减、乘、除运算的基本步骤。下面所给出的规则适用于单精度 IEEE 标准格式。这些规则只说明了执行四则运算的主要步骤；例如，我们没有讨论可能发生的上溢或下溢。还有，尾数与指数的中间结果需要的位数可能分别比 24 和 8 位更多。在设计符合标准的算术部件时，必须认真考虑运算中的这些问题及其他问题。尽管在规则说明中没有给出全部的细节，我们还是考虑了实现中的一些方面，包括后面章节将会讨论的舍

入问题。

当对浮点数进行加或减运算时，如果它们的指数不同，则尾数必须根据对方进行移位操作。考虑一个十进制的例子，将 $2.9400 \times 10^2$ 与 $4.3100 \times 10^4$ 相加。我们将 $2.9400 \times 10^2$ 重写为 $0.0294 \times 10^4$，再进行尾数加法从而得到 $4.3394 \times 10^4$。加减法的规则如下：

**1. 加/减法规则**

1）选取指数较小的数，将其尾数右移，右移的步数等于两指数之差。

2）将结果的指数设为与较大的指数相等。

3）对尾数进行加/减运算，并确定结果的符号。

4）如果必要，对结果的值进行规格化。

乘除法比加减法要简单些，因为不需要对齐尾数。

**2. 乘法规则**

1）将指数相加并减去 127 以保持余 127 表示。

2）将尾数相乘并确定结果的符号。

3）如果必要，对结果的值进行规格化。 [367]

**3. 除法规则**

1）将指数相减并加上 127 以保持余 127 表示。

2）将尾数相除并确定结果的符号。

3）如果必要，对结果的值进行规格化。

### 9.7.2 保护位与截取

现在我们来考虑一下实现上述算法步骤时的一些重要方面。尽管初始操作数和最终结果的尾数都限制在 24 位（包括隐含的前导 1），在中间步骤中保留一些附加位，通常称为保护（guard）位，是非常重要的。它们使最终结果具有最高的精确度。

生成最终结果时去除保护位要求对扩展的尾数进行截取（truncate），从而获得接近扩展尾数的 24 位数。在其他一些情况下也会进行这一操作，比如将十进制数转换成二进制时。应当指明舍入也可以用于截取操作，但是在这里我们将舍入更严格地定义为一种特定格式的截取。

截取有几种方法。最简单的方法是将保护位去除，而对剩余的各位不加改变。这种方式称作截断（chopping）。假设我们要使用这种方法将一个小数从六位截取到三位。区间 $0.b_{-1}b_{-2}b_{-3}\ 000$ 到 $0.b_{-1}b_{-2}b_{-3}\ 111$ 内的所有小数都将被截取为 $0.b_{-1}b_{-2}b_{-3}$。三位结果的误差为 0 到 0.000111。换句话说，截断误差从 0 到接近剩余位最低有效位置上的 1，在上面的例子中，该位置就是 $b_{-3}$ 的位置。截断的结果是有偏（biased）近似，因为误差区间并不关于 0 对称。

另一种最简单的截取方法是冯·诺依曼舍入（von Neumann rounding）。如果要去除的各位全为 0，就简单地去除它们，并保持其他位不变。但是如果要去除的位中有任何一位为 1，就将剩余位的最低有效位置为 1。在六位到三位的截取例子中，所有 $b_{-1}b_{-2}b_{-3}$ 不等于 000 的小数都被截取为 $0.b_{-1}b_{-2}1$。这种截取方法的误差在剩余位 LSB 位置的 −1 到 +1 之间。尽管这种技术的误差范围比截断要大，但误差的最大值相同，而且其近似是无偏（unbiased）的，因为误差区间关于 0 对称。

如果在生成结果时涉及许多操作数和运算，那么无偏近似是有利的，因为随着计算的进行正误差会与负误差相互抵消。在统计上，复杂计算的结果更精确。

第三种截取方法是舍入（rounding）过程。舍入的结果最接近于被截取的数，而且是无偏近似。其过程如下：如果去除位的 MSB 位置上为 1，则在保留位的 LSB 位置上加 1。这样，

$0.b_{-1}b_{-2}b_{-3}1\cdots$就被舍入成$0.b_{-1}b_{-2}b_{-3}+0.001$，而$0.b_{-1}b_{-2}b_{-3}0\cdots$则被舍入为$0.b_{-1}b_{-2}b_{-3}$。这种方法提供了我们需要的近似，除了去除位为10…0的情况。这是一个中间状态；符截取的数字位
|368 于两个最近的截取表示的正中间。为了无偏地解决该问题，一种可能是将保留位设置为最接近的偶数。对于六位的例子，数值$0.b_{-1}b_{-2}0100$将被截取为$0.b_{-1}b_{-2}0$，而数值$0.b_{-1}b_{-2}1100$则被截取为$0.b_{-1}b_{-2}1+0.001$。"在陷入僵局时舍入到最近的数或最近的偶数"这句话有时就是用来描述这种截取技术。误差区间大致在保留位LSB位置的−1/2到+1/2。很明显，这是最好的方法。但是，它也是最难实现的，因为它要求一次加法操作以及可能的再规格化。IEEE浮点标准将这种舍入技术指定为截取操作的默认模式。标准还规定了其他的截取方法，并将所有这些方法称为舍入模式。

这里对通过截取去除保护位引入误差的讨论只考虑了单独的截取操作。当对浮点数进行一系列很长的计算时，确定最终结果误差区间或范围的分析是一项复杂的研究。除了对IEEE浮点标准中保护位和舍入的处理方式作一些讨论之外，我们将不再进一步讨论数值计算等方面。

根据标准，单步操作的结果必须精确到LSB位置上单位的一半。这意味着必须采用舍入作为截取方法。实现舍入只需要在执行一个操作的中间步骤中添加三个保护位。其中前两位是将被去除的尾数部分中的两个最高有效位。第三位是在尾数的完整表示中在以上两位之后的所有低位的逻辑或。这一位在所执行操作的中间步骤中比较容易维护。它应当初始化为0。如果在对齐尾数时有一个1从该位置移出，则将这一位置为1，并保持该值；因此，它通常被称为*粘着位*（sticky bit）。

### 9.7.3 浮点操作的实现

浮点操作的硬件实现涉及大量的逻辑电路。这些操作也可以用软件例程实现。不论使用哪种方式，计算机必须能够将用户十进制表示的数转换为所需要的输入格式，并将输出转换为十进制表示。很多通用处理器在机器指令级提供浮点操作，并用硬件实现。

图9-28显示了一个浮点操作实现的例子。这是具有图9-26a所示格式的32位浮点操作数加减操作的硬件实现框图。按照9.7.1节中的加/减法规则，我们看到第1步是比较指数以确定对具有较小指数的数的尾数进行移位的次数。移位次数$n$由图中左上角的8位减法器电路决定。$E'_A-E'_B$的差值$n$被传送到SHIFTER（移位器）部件。如果$n$大于操作数的有效位数目，则结果实际上就是较大的操作数（不考虑舍入中的保护位与粘着位），而且可以采用便捷的方
|369 法产生结果。对此我们不进行详细讨论。

比较指数所得的差的符号决定对哪个数的尾数移位。因此在第1步中，这个符号被传入图9-28右上角的SWAP（交换器）网络。如果符号为0，则$E'_A \geqslant E'_B$，尾数$M_A$与$M_B$直接通过SWAP网络。这使得$M_B$被送入SHIFTER，并被右移$n$位。另一个尾数$M_A$则被直接送入尾数加法器/减法器。如果符号为1，则$E'_A < E'_B$，在尾数送至SHIFTER之前需要将它们交换。

第2步由图中靠近左下角的多路复用器MUX完成。结果的指数$E'$暂时由第1步中指数比较所得的差的符号确定，如果$E'_A \geqslant E'_B$则$E'=E'_A$，如果$E'_A < E'_B$则$E'=E'_B$。

第3步涉及主要的组件——图中部的尾数加法器/减法器。CONTROL（控制）逻辑尾数是相加还是相减。这由操作数的符号（$S_A$与$S_B$）和操作数上执行的操作（加或减）确定。CONTROL逻辑还决定结果的符号$S_R$。例如，如果$A$为负（$S_A=1$），$B$为正（$S_B=0$），且操作为$A-B$，则需要对尾数执行加法且结果的符号为负（$S_R=1$）。另一方面，如果$A$和$B$都为正，且操作为$A-B$，则需要将尾数相减。现在结果的符号$S_R$依赖于尾数的相减操作。例如，如果

$E'_A > E'_B$，则 $M = M_A -$（移位的 $M_B$），结果为正。如果 $E'_A < E'_B$，则 $M - M_B -$（移位的 $M_A$），结果为负。从这个例子可以看出，指数比较所得的符号也需要作为 CONTROL 网络的输入。当 $E'_A = E'_B$ 且尾数相减时，尾数加法器/减法器输出的符号决定了结果的符号。读者现在应该可以为 CONTROL 网络构造完整的真值表了（参见习题 9.26）。

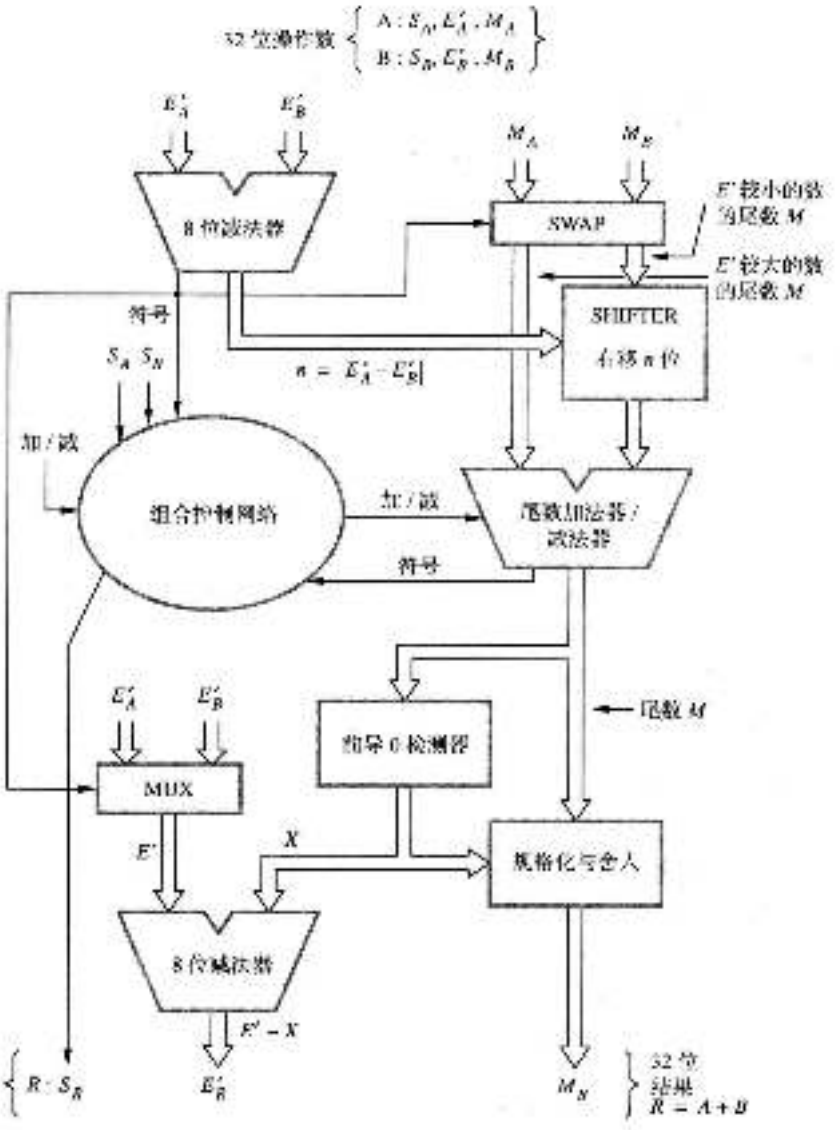



图 9.28 浮点加法/减法部件

加/减法规则的第 4 步是通过适当地左移或右移 $M$ 来对第 3 步的结果进行规格化。$M$ 中前导 0 的数目决定了对 $M$ 的移位次数 $X$。对规格化的值舍入从而生成结果的 24 位尾数 $M_R$。临时的结果指数 $E'$ 也需要减去值 $X$，生成真正的结果指数 $E'_R$。注意，可能只需要右移一位就可以对结果规格化。两个形式为 1.xx…的尾数相加时就会发生这种情况。这时 $M$ 向量具有 1x.xx…的形式。

对于中间尾数值必须附带的保护位，我们没有给出任何细节。在 IEEE 标准中，如前所

述，生成结果中的 24 位规格化尾数只需要几个保护位即可。

让我们考虑一下实现 9-28 中的框图所需要的实际硬件。如前所述，两个 8 位减法器和尾数加法器/减法器可以使用组合逻辑实现，因为它们必须以原码的形式输出，因此我们需要对前面的讨论做一些修改。反码运算与原码表示的组合经常用到。SHIFTER 和输出规格化操作的实现允许有极大的灵活性。这些操作可以使用移位寄存器来实现。然而，为了达到高性能，也可以将它们构造为组合逻辑部件。

370
371

## 9.8 十进制数到二进制数的转换

在第 1 章和本章中，涉及十进制数的例子都使用了比较小的值，根据二进制位位置的权重 1，2，4，8，16 等将十进制数转换为二进制表示是非常容易的。但是，如果有一种将十进制数转换为二进制表示的通用方法，将会是很有用的。

定点的无符号二进制数

$$B = b_n b_{n-1} \cdots b_0 . b_{-1} b_{-2} \cdots b_{-m}$$

有 $n$ 位整数部分和 $m$ 位小数部分。它的值 $V(B)$ 由下式给出：

$$V(B) = b_n \times 2^n + b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m}$$

将一个定点的十进制数转换为二进制数时，整数部分和小数部分要分别计算。整数部分的转换先将其除以 2，余数 0 或 1 是 $B$ 中整数部分的最低有效位 $b_0$。商再除以 2，余数是 $B$ 中整数部分的次低位 $b_1$。重复这个过程直至商为 0（包括商为 0 这一步）。

小数部分的转换先将其乘以 2。乘积中十进制小数点左边的部分 0 或 1 是 $B$ 中小数部分的位 $b_{-1}$。乘积的小数部分再乘以 2，得到 $B$ 中小数部分的下一位 $b_{-2}$。重复此过程直到乘积的小数部分变成 0 或者达到要求的精确度为止。

图 9-29 给出了一个将十进制数 927.45 转换为二进制的例子。注意，整数部分的转换总是精确的，并且当商为 0 的时候结束。但一个给定的十进制小数可能不存在精确的二进制小数。例如，图 9-29 中的十进制小数 0.45 并没有精确的二进制等价值。这从图中可以明显地看出来。在这种情况下，二进制小数按所需的精度水平取值。当然，有些十进制小数有精确的二进制值。例如，十进制小数 0.25 有等价的二进制值 0.01。

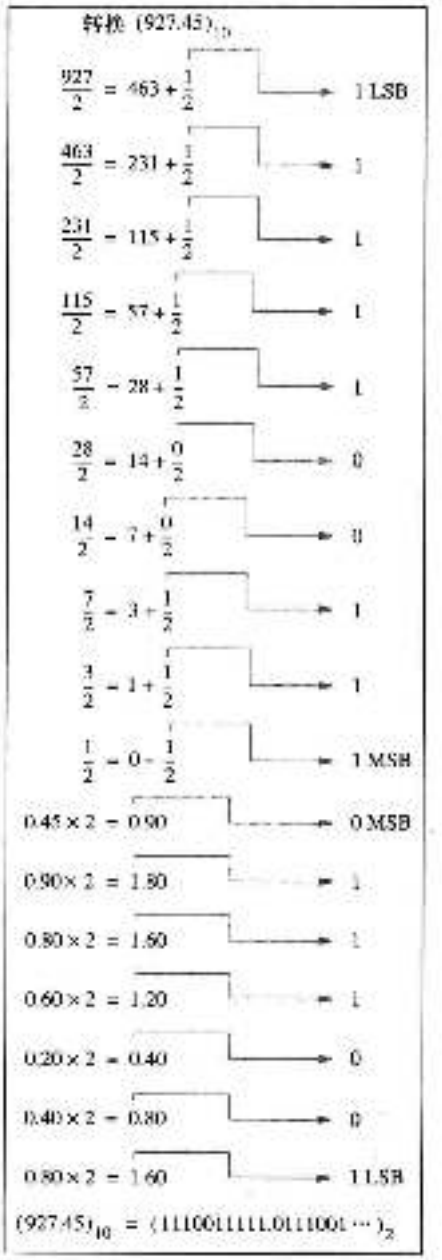


图 9-29 十进制数到二进制数的转换

## 9.9 结束语

计算机的算术运算涉及几个非常有趣的逻辑设计问题。本章讨论了一些在二进制算术部件设计中有实用价

值的技术。超前进位技术是高性能加法器设计的主要思想之一。在快速乘法器的设计中，由 Booth 算法发展而来的乘数位偶重编码减少了生成乘积所需的求和项的数目。使用进位保留简化树的求和项并行加法显著地减少了将求和项相加所需的时间。本章还介绍了浮点数的 IEEE 表示标准，以及执行浮点数四则运算的规则。

372 / 373

## 9.10 问题解析

本节将介绍一些可能要求学生解决的类型问题，并分析说明如何解决这样的问题。

**例 9.1**

问题：构建图 9-4 所示的 4 位超前进位加法器需要多少逻辑门？

解答：如图 9-4a 所示，每个 B 单元需要 3 个门，因此四个 B 单元需要 12 个门。

根据 9.2.1 节中的四个逻辑表达式可知，超前进位逻辑产生的进位 $c_1$、$c_2$、$c_3$ 和 $c_4$ 分别需要 2、3、4 和 5 个门。如 9.2.1 节所述，超前进位逻辑还使用 4 个门产生 $G'_0$，使用 1 个门产生 $P'_0$。因此，实现超前进位逻辑总共需要 19 个门。

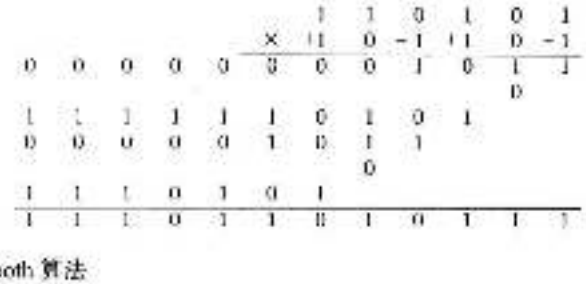
完整的 4 位加法器需要 12+19=31 个门，且最大的扇入为 5。

**例 9.2**

问题：假设使用 6 位的补码数表示，按照表 9-15 的格式，使用普通的 Booth 算法和位偶重编码 Booth 算法计算被乘数 A=110101 与乘数 B=011011 的乘积。

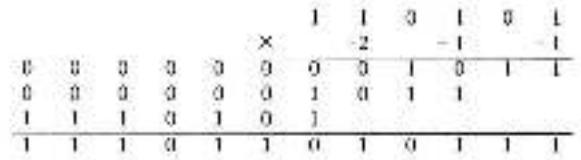
解答：乘法的执行如下：

（a）普通的 Booth 算法



374

（b）位偶重编码 Booth 算法



**例 9.3**

问题：在简化树中，要将 $k$ 个求和项降为 2 个需要多少层 4-2 简化器？如果使用 3-2 简化器，那么将需要多少层？

解答：假设层数为 $L$。

对于 4-2 简化器，有

$$k(1/2)^L = 2$$

对该等式两边取以 2 为底的对数，可以得到

$$\log_2 k - L = 1$$

或者

$$L = \log_2 k - 1$$

对于 3-2 简化器，有

$$k(2/3)^L = 2$$

同上，两边取以 2 为底的对数，得到

$$\log_2 k + L(\log_2 2 - \log_2 3) = \log_2 2$$

$$\log_2 k + L(1 - 1.59) = 1$$

$$L = (1 - \log_2 k)/(-0.59)$$

$$L = 1.7\log_2 k - 1.7$$

如果每层的输入求和项数目不是 4（4-2 简化时）或者 3（3-2 简化时）的倍数，那么这些表达式只能是近似的。

**例 9.4**

问题：将十进制小数 0.1 转换为二进制小数。如果转换不精确，使用 9.7.2 节中讨论的二种截取方法将二进制小数近似为二进制小数点后 8 位。

375

解答：使用 9.8 节给出的转换方法。如图 9-29 所示，将十进制小数 0.1 乘以 2，所得乘积的小数部分再乘以 2，不断重复这一过程，每次乘积中小数点左边的部分生成位序列 0，0，0，1，1，0，0，1，1，0，0，1，1，…，该序列将无限期地继续下去，重复模式 0，0，1，1。因此，该转换是不精确的。

- 通过截断方法截取得到 0.00011001
- 通过马·诺依曼舍入方法截取得到 0.00011001
- 通过舍入方法截取得到 0.00011010

**例 9.5**

问题：为了便于完成下面的数字练习，考虑一种 12 位的浮点数表示格式。第 1 位是数的符号，接下来的 5 位表示比例因子（隐含的基数为 2）的余 15 格式的指数。最后 6 位表示尾数的小数部分，其二进制小数点的左边有一个隐含的 1。

对如下的类作数 $A$ 和 $B$ 执行减法和乘法操作：

| | | | |
|---|---|---|---|
| $A$= | 0 | 10001 | 0.1011 |
| $B$= | 1 | 01111 | 101010 |

它们分别表示数

$$A = 1.011011 \times 2^1$$

和

$$B = -1.101010 \times 2^0$$

解答：所需的类作执行如下：

- 减法

根据 9.7.1 节中的加/减法规则，执行下列四个步骤：

1）将 $B$ 的尾数右移两位，得到 0.01101010。

2）将结果的指数设为 10001。

3）因为 $B$ 为负数，所以通过尾数相加将 $A$ 的尾数减去 $B$ 的尾数，得到

```
  1 . 0 1 1 0 1 1 0 0
+ 0 . 0 1 1 0 1 0 1 0
-------------------
  1 . 1 1 0 1 0 1 1 0
```

并将结果的符号设为 0（正）。

4）结果是规格化的形式，但尾数的小数部分需要被截取为 6 位。如果使用舍入方法进行截取，要去除的两位表示中间状态，所以我们通过加 1 将保留位舍入到最近的偶数，得到结果尾数 1.110110。结果为

376

| | | | |
|---|---|---|---|
| $A - B$= | 0 | 10001 | 110110 |

- 乘法

根据 9.7.1 节中的乘法规则，执行下列二个步骤：

1）将指数相加并减去 15，得到 10001 作为结果的指数。

2）将尾数相乘，得到 10.010110101110 作为结果的尾数，结果的符号设为 1（负）。

3）将结果尾数右移一位对其进行格式化，然后将指数加1得到10010作为结果的指数。通过舍入方法将尾数的小数部分截取为6位，得到结果

$A \times B =$

| 0 | 10010 | 001011 |
|---|---|---|

## 习题

[M] 9.1 半加器（half adder）是一个组合逻辑电路，它有两个输入 $x$ 与 $y$，以及两个输出 $s$ 与 $c$，分别是 $x$ 与 $y$ 相加所得的和与进位输出。

（a）采用二级与或电路设计一个半加器。

（b）说明如何使用两个半加器以及必要的外部逻辑门实现图 9.2a 中的全加器。

（c）比较（b）部分的网络和图 9-2a 中的加法器网络的最长逻辑延迟路径。

[M] 9.2 反码与补码二进制表示方法是以 $b$ 为基数的数字系统中（$b-1$）进制补码和 $b$ 进制补码表示技术的特例。例如，考虑十进制系统。以原码形式表示的数值 1526，−526，+70 和 −70 在两个补码系统中都有 4 位原码表示，如图 P9-1 所示。数字的每一位对 9 取补就构成了九进制补码；对九进制补码加 1 就构成了十进制补码。在这两种表示中，正数的最左位为 0，负数的最左位为 9。 377

| 表示 | 示例 | | | |
|---|---|---|---|---|
| 原码 | +526 | −526 | +70 | −70 |
| 9 进制补码 | 0526 | 9473 | 0070 | 9929 |
| 10 进制补码 | 0526 | 9474 | 0070 | 9930 |

P9-1 习题 9.2 中以 10 为基数的有符号数

现在考虑基数为 3 的系统（三进制系统），其中 5 位无符号数 $a_4a_3a_2a_1a_0$ 的值为 $a_4\times3^4+a_3\times3^3+a_2\times3^2+a_1\times3^1+a_0\times3^0$，$0\le a_i\le2$。将原码表示的三进制数 −11011、−10222、+2120、−1212、+110 和 −201 表示为三进制补码系统中的 6 位有符号三进制数。

[M] 9.3 将十进制数 56、−37、122 和 −123 表示为三进制补码格式的 6 位有符号数，对所有可能的配对进行加减操作，并说明所执行的每个操作是否出现了算术溢出。（参见习题 9.2 对三进制数系统的定义，并使用类似于 9.8 节中十进制到三进制转换的技术。）

[M] 9.4 对 BCD 数求和需要模 10 加法器，两个 BCD 数 $A=A_3A_2A_1A_0$ 和 $B=B_3B_2B_1B_0$ 的模 10 加法执行如下：将 $A$ 加到 $B$ 上（二进制加法）。之后，如果结果为大于或等于 $10_{10}$ 的非法编码，则加 $6_{10}$。（忽略本次加法的溢出。）

（a）什么时候输出进位等于 1？

（b）证明本算法对于以下数值可以给出正确结果：

（1）$A=0101$　$B=0110$

（2）$A=0011$　$B=0100$

（c）使用 4 位二进制加法器以及必要的外部逻辑门设计一个 BCD 数加法器。输入为 $A_3A_2A_1A_0$，$B_3B_2B_1B_0$ 与进位输入。输出为和数 $S_3S_2S_1S_0$ 与进位输出。级联这样的组件可以构成行波进位 BCD 加法器。

[E] 9.5 使用适当的真值表证明，在补码溢出加法中，逻辑表达式 $c_n \oplus c_{n-1}$ 正确地指示了溢出的发生。

[E] 9.6 使用例 9.1 中解决方案的适当部分计算构建图 9-5 中的 16 位超前进位加法器需要多少逻辑门。

[M] 9.7 本题研究的是超前进位加法器及其延迟问题。

（a）使用四个图 9-5 所示的 16 位超前进位加法器以及附加逻辑电路设计一个 64 位加法器，从图中的 $c_{15}$、$G_3''$ 和 $P_3''$ 产生生成 $c_{31}$、$c_{32}$、$c_{48}$ 和 $c_{64}$。附加电路与图中超前进位逻辑电路有什么关系？

（b）证明 9.2.1 节末尾的结论，通过 64 位加法器的延迟对 $s_{63}$ 来说为 12 个门延迟，对 $c_{64}$ 来说为 7 个门延迟。

（c）比较（a）部分中 64 位加法器与 9.2.1 节讨论的使用两个 16 位加法器级联构成的 32 位加法器生成 $s_{31}$ 和 $c_{32}$ 的门延迟。

378

[M] 9.8 证明 9.3.1 节的结论，通过图 9-6b 中的 $n \times n$ 阵列最坏情况的延迟为 $6(n-1)+1$ 个门延迟。

[E] 9.9 使用 Booth 算法计算以下各对有符号补码数的乘法。假设 $A$ 为被乘数，$B$ 为乘数。

（a）$A = 010111$　$B = 110110$

（b）$A = 110011$　$B = 101100$

（c）$A = 001111$　$B = 001111$

[M] 9.10 使用乘数位倒重编码重复习题 9.9。

[M] 9.11 概括地说明怎样修改图 9-7a 中的电路图，使其可以使用 Booth 算法实现 $n$ 位补码数的乘法。明确指出控制序列发生器（Control sequencer）的输入和输出，以及对加法器和寄存器 $A$ 所作的必要改变。

[M] 9.12 将图 9-14b 中的表扩展为 16 行，说明如何对 3 位乘数位 $i+2$、$i+1$ 和 $i$ 重编码。是否可以通过对被乘数 M 进行移位和/或求反来生成位置 $i$ 选择的所有被乘数变体？如果不可以，那么哪些被乘数变体不能以这种方式生成？什么情况下需要这些被乘数变体？

[M] 9.13 如果两个 $n$ 位补码数的乘积可以用 $n$ 位表示，则图 9-6a 中的手工相乘算法就可以直接使用，这时将符号位与其他位同等对待。用这种方式对下面的两对 4 位有符号数进行乘法计算：

（a）被乘数 = 1110　乘数 = 1101

（b）被乘数 = 0010　乘数 = 1110

为什么这时的计算是正确的？

[D] 9.14 利用一个能够计算 16 位无符号数加法和乘法的整数运算部件计算两个 32 位无符号数的乘法。所有的操作数、中间结果和最终结果存储于标号为 $R_0$ 到 $R_{15}$ 的 16 位寄存器中。硬件乘法器将 $R_i$（被乘数）与 $R_j$（乘数）的内容相乘，并将 32 位的乘积存入寄存器 $R_i$ 和 $R_{i+1}$ 中，其中低位部分存入 $R_i$ 中。当 $j = i-1$ 时，乘积将覆盖两个操作数。硬件加法器将 $R_i$ 与 $R_j$ 的内容相加并将结果存入 $R_i$ 中。加法操作的进位输入为 0，而有进位加法（Add-with-carry）操作的进位输入为进位标志 C 的内容。加法器的进位输出始终保存在 C 中。

说明计算寄存器 $R_1$、$R_0$ 和 $R_3$、$R_2$ 中两个 32 位操作数（高位部分在前）乘法的步骤，将 64 位的乘积放入寄存器 $R_{15}$、$R_{14}$、$R_{13}$ 和 $R_{12}$ 中。如果需要，$R_{11}$ 到 $R_4$ 的任何寄存器都可以用来保存中间结果。过程中的每一步可以是乘法、加法或寄存器传输操作。

[M] 9.15 本题研究的是乘法阵列中的延迟问题。

（a）计算图 9-16 中每个 $4 \times 4$ 乘法阵列产生乘积位 $p_i$ 的延迟，即全加器部件的延迟。忽略开始时生成所有 $m_i q_j$ 乘积的与门延迟。

（b）作为问题（a）部分的延伸，为图 9-16 中的每个阵列写出其在 $n \times n$ 情况下的延迟表达式。然后使用这些表达式计算每个矩阵在 $32 \times 32$ 情况下的延迟。

379

[M] 9.16 本题分析的是进位保留简化中的时延问题。

（a）使用与图 9-19 相似的模式，将 16 个求和项降为两个需要多少层 3-2 简化操作？

（b）将 32 个求和项降为两个，重复（a）部分的问题，以证明 9.5.3 节中八层的结论是正确的。

（c）将（a）部分与（b）部分中的精确答案与使用 9.10 节中例 9.3 所生成的近似值面得到的结果进行比较。

[M] 9.17 9.5.3 节和 9.5.4 节分别描述了使用 3-2 和 4-2 简化器对求和项进行简化的树结构。还可以在每个简化层上进行 7-3 简化操作。当只剩下 3 个求和项时，执行 3-2 简化操作，然后对最终的两个求和项执行加法运算。

（a）将 32 个求和项降为 3 个需要多少层 7-3 简化操作？将其与使用 3-2 简化时将 32 个求和项降为 3 个所需要的 7 层进行比较。

（b）9.10 节中例 9.3 证明了在简化树中将 $k$ 个求和项降为 2 个需要 $\log_2 k - 1$ 层 4-2 简化操作，那么将 $k$ 个求和项降为 3 个需要多少层 7-3 简化操作？

[M] 9.18 说明如何使用两个 3-2 简化器实现一个 4-2 简化器，该实现的真值表与图 9-21 中的不同。

[M] 9.19 用手工方法对 5 位无符号数 $A = 10101$ 和 $B = 00101$ 计算 $A \times B$ 与 $A \div B$ 操作。

[M] 9.20 创建与图 9-7b 和图 9-25 类似的列表，说明习题 9.19 中的乘法和除法操作是如何使用图 9-7a 和图 9-23 的硬件执行的。

[D] 9.21 在 9.7 节中，我们使用了 32 位的 IEEE 标准格式来表示浮点数。这里使用一种缩短格式，它保持了全部相关的概念，但是便于完成数字练习。考虑图 P9-2 中的 12 位格式表示浮点数。比例因子有一个隐含的基数 2 和 5 位的余 15 指数，其中两个端点值 0 和 31 分别用来表示数值 0 和无穷大。6 位尾数使用 IEEE 格式规格化，其二进制小数点的左边有一个隐含的 1。

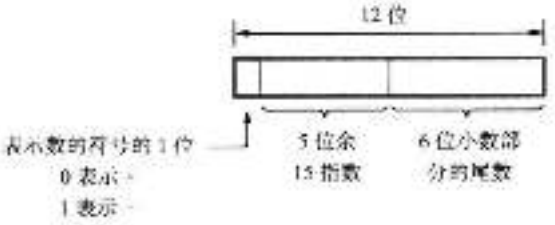



图 P9-2 习题 9.21 中的浮点数格式

（a）将数字 +1.7、−0.012、+19 和 1/8 表示为这种格式。

（b）这种格式所能表示的最小和最大值是多少？

（c）将（b）部分计算的区间范围与 12 位有符号整数和 12 位有符号小数的表示范围进行比较。

（d）对如下的操作数 A 和 B 执行加减乘除运算：

| | | | |
|---|---|---|---|
| $A=$ | 0 | 10000 | 011011 |
| $B=$ | 1 | 01110 | 101010 |

[380]

[D] 9.22 考虑用类似习题 9.21 格式表示的 16 位浮点数。它具有 6 位指数和 9 位尾数小数。比例因子的基数为 2，指数采用余 31 格式表示。

（a）将以下格式的 A 和 B 相加：

| | | | |
|---|---|---|---|
| $A=$ | 0 | 100001 | 111111110 |
| $B=$ | 1 | 011111 | 001010101 |

将计算结果转换为规格化格式。记住二进制小数点的左方有一个隐含的 1 但并不包含在 $A$ 和 $B$ 的格式中。在生成最终的尾数时使用舍入法作为截取方法。

（b）使用十进制数 $w$、$x$、$y$ 和 $z$ 表示前面规格化浮点数格式所能表示的最大值与最小（非零）值。使用下面的格式

$$\text{最大值} = w \times 2^{x}$$
$$\text{最小值} = y \times 2^{-z}$$

[M] 9.23 在图 9-26a 的浮点数表示中，指数的余 $x$ 表示为比较两个浮点数的相对大小提供了什么便利？（提示：假设有一个组合逻辑网络可以比较两个 32 位无符号整数的相对大小。使用这一网络以及必要的外部逻辑门设计比较浮点数所需的网络。）

[D] 9.24 在习题 9.21（a）中，简单十进制数到二进制浮点数的转换是直接的。但是，如果十进制数也用浮点格式表示的，转换就不是直接的了，这是因为我们不能单独地转换尾数与比例因子的指数，因为 $10^{x} - 2^{y}$ 一般并不能保证 $x$ 和 $y$ 都为整数。假设计算机中存储着一个表，表中的二进制浮点数 $t_i$ 和 $s_i$ 的关系为 $t_i = 10^{s_i}$。请说明将给定的十进制浮点数转换为二进制浮点格式的一般步骤。可以使用计算机中的整数和浮点数指令。 [381]

[D] 9.25 构建一个实例，说明当两个正数相减时为获得正确的结果需要三个保护位。

[M] 9.26 为图 9-28 中组合控制网络的输出 Add/Sub（加/减）和 $S_2$ 推导逻辑表达式。

[M] 9.27 如果门的扇入为 4，如何用组合电路实现图 9-28 中的 SHIFTER（移位器）？

[M] 9.28 画出实现图 9-28 中的多路复用器 MUX 的逻辑门网络。

[M] 9.29 将图 9-28 中的 SWAP（交换器）网络结构与习题 9.28 的答案联系起来。

[M] 9.30 如何用组合电路实现图 9-28 中的前导 0 检测器？

[M] 9.31 图 9-28 中的反码加法器/减法器对正的无符号二进制小数操作，而且必须以原码的形式给出结果。在对图 9-28 的讨论中我们说过对于输入和输出操作数所需的格式，反码运算是非常方便的。当两个反码表示的有符号数相加时，必须将符号位的进位输出加到结果中才能得到正确答案。这被称为循环进位校正（end-around carry correction）。图 P9-3 中的两个示例使用 4 位有符号反码对操作数和答案编码，并展示了加法操作。当需要生成原码形式的结果时，反码运算系统是非常方便的，因为以反码形式表示的负数可以通过对符号位右方的各位求补转换为原码形式。而使用补码运算，将负数转换为原码形式时需要加上 −1。如果使用超前进位加法器，可以将反码运算所需的循环进位操作结合到超前逻辑当中。请以上述讨论为指导，给出图 9-28 中反码加法器/减法器的完整设计。

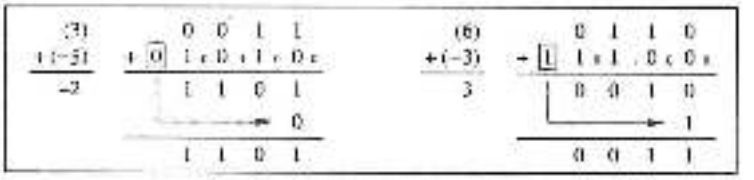



图 P9-3 习题 9.31 中的反码加法

[M] 9.32 1.4.2 节中讨论了补码表示的有符号二进制小数。

（a）将十进制数值 0.5，−0.123，−0.75 和 −0.1 表示为 6 位有符号的小数。（参阅 9.8 节中的十进制到二进制小数的转换。）

382

（b）当二进制小数点后使用 5 位有效数字时，最大表示误差 $e$ 是多少？

（c）分别计算表示误差 $e$ 小于 0.1、0.01 或者 0.001 时二进制小数点后所需的有效数字位数。

[E] 9.33 习题 9.32（a）中的四个 6 位答案哪些是不精确的？对每一个不精确的答案，给出与 9.7.2 节定义的三种截取方法相对应的三个 6 位值。

## 参考文献


1. A. D. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 2, part 2, 1951, pp. 236-240.
2. C. S.Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, February 1964, pp. 14-17.
3. M. R. Santoro and M. A. Horowitz, "SPIM: A Pipelined 64 × 64-bit Iterative Multiplier," *IEEE Journal of Solid-State Circuits*, vol. 24, No.2, April 1989, pp. 487-493.

383
384

4. Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-2008, August 2008.

# 嵌入式系统

## 本章目标

在本章中你将学习以下内容：

- 嵌入式应用
- 嵌入式系统中的微控制器
- 传感器与执行器
- 使用 C 语言控制 I/O 设备
- 设计问题



前几章我们讨论了通用计算机系统中使用的一些概念。现在我们将集中讨论旨在为特定的应用提供服务的系统。用于对特定目标进行计算机控制而不是用于通用计算的实际系统，称为嵌入式系统（embedded system）。我们将展示如何在这样的系统中应用前面介绍的一般概念。

为嵌入式系统编写的软件的一个重要方面是它必须与硬件紧密交互。术语反应系统（reactive system）通常用来描述这样的情况：执行各种例程的时间点是由处理器外部的事件确定的，比如一个开关的闭合，或者在一个输入端口上到达了新的数据。软件设计人员必须决定如何实现这种交互。第 3 章中介绍的基于轮询和中断的输入/输出技术将用于此目的。

微处理器控制现在普遍应用于照相机、手机、可视电话、销售点终端、厨房电器、汽车和许多玩具中。在这些应用中，低成本和高可靠性是基本的要求，小尺寸和低功耗往往也是至关重要的。为实现所有的这些要求，通常我们不仅仅将处理器电路，还将一些存储器、输入/输出接口、定时器电路和其他功能放置到一个单一的芯片上，以便于使用很少的芯片来实现一个完整的计算机控制系统。这种类型的微处理器芯片通常被称为微控制器（microcontroller）。在这一章中，我们将探讨基于微控制器的嵌入式系统的主要特点。在第 11 章中，我们将讨论使用现场可编程门阵列（FPGA）技术来实现这种系统的片上系统方法。

## 10.1 嵌入式系统实例

在本节中我们给出三个嵌入式系统的实例，说明在一个典型的嵌入式应用中所需要的处理过程和控制功能。

### 10.1.1 微波炉

许多家用电器使用计算机控制去管理它们的操作，典型的例子就是微波炉。这种用具基于一个磁控管发生器产生的微波去加热在一个封闭空间中的食物。当磁控管打开时，它产生最大的功率输出，较低的功率级别是通过依据受控的时间间隔开关磁控管的方式达到的。有了控制功率和总加热时间的功能，就可以应对烹调操作中各种用户的选择。

微波炉的具体规格说明可能包括以下的烹调操作：

- 手动选择功率级别和烹调时间。
- 手动选择不同烹调步骤的次序。
- 在用户指明了食物类型（例如：肉、蔬菜或爆米花）以及食物的重量时自动操作；然后，一个合适的功率和时间便由控制器计算出来。

386

- 根据指定食物重量自动解冻。

炉中包含一个显示器，它可以显示：

- 每日时钟定时
- 烹饪过程中时钟定时器递减
- 给用户的提示信息

以蜂鸣声形式发出的声音报警信号，用于表示烹饪操作结束。还需要提供一个排风扇和一个炉灯信号。作为一种安全措施，如果炉子的门被打开，门锁必须关闭磁控管。所有这些功能可以用微控制器控制。

与用户有关的输入/输出功能必须包括：

- 输入键，它包括 0～9 的数字键以及 Reset（重置）、Start（开启）、Stop（暂停）、Power Level（功率级别）、Auto Defrost（自动解冻）、Auto Cooking（自动烹饪）、Clock Set（时钟设置）及 Fan Control（风扇控制）这样的功能键。
- 液晶显示器形式的可视化输出，类似于图 3-17 所示的七段显示器。
- 一个小型扬声器用来产生蜂鸣声。

由微控制器执行的用来控制微波炉的计算任务非常简单，仅包括维护每天的时钟时间、确定各种烹饪操作中所需的动作、产生打开或关闭磁控管和电风扇这类设备的控制信号以及生成显示信息。实现所需动作的程序非常小，它存储在一个不易失的只读存储器中，这样当关闭电源时才不会丢失。还需要有一个小型的 RAM 用于在运算过程中执行和保存用户的输入数据。微控制器中最重要的需求就是，针对所有的输入键、显示器和输出控制信号，微控制器要具备足够的 I/O 能力。并行 I/O 端口提供了一套与外部的输入和输出信号进行交互的便利方法。

图 10-1 给出了微波炉的一种可能的组成结构。一个带有小容量 ROM 和 RAM 部件的简单处理器就足够用了。一些基本的输入/输出接口用来连接该系统的其他部分。在一个小的微控制器芯片中实现这些电路中的大部分功能是可行的。

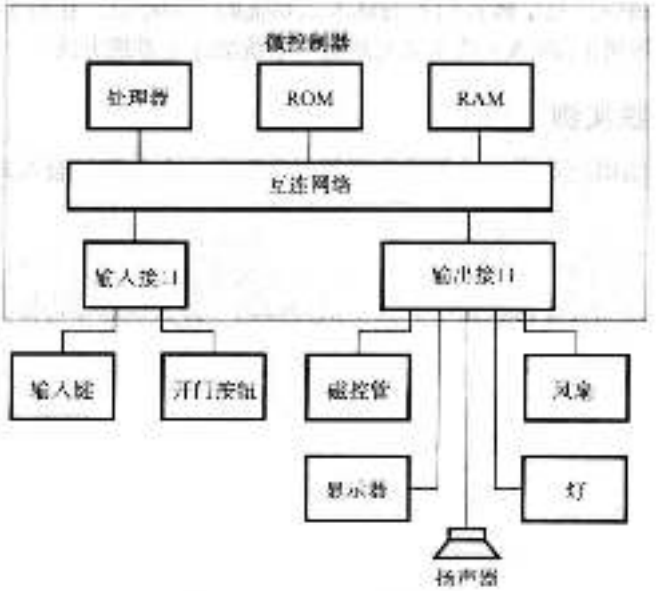



图 10-1 一个微波炉的框图

### 10.1.2　数码照相机

数码照相机给出了一个在小型封装设备中成熟的嵌入式系统的优秀范例。图 10-2 列出了数码照相机中的主要部分。

传统照相机使用胶片去捕获图像。在数码照相机中，用一个光传感器阵列去捕获图形。这些传感器可以将光转换成电荷。光的强度决定产生电荷总量的大小。商业产品中使用了两种不同类型的传感器。一种类型是著名的电荷耦合器件（charge-coupled device，CCD）。最早的数字照相机中就是用的这种传感设备。它渐渐被改进成可提供高质量图像的设备。最近基于 CMOS 技术的传感器已经开发出来。

每个传感器产生一个对应于一个像素（pixel）的电荷，它是图片图像中的一个点。像素的数量决定了可以记录和显示的图片的质量。电荷是一个模拟量，它被数模（analog-to-digital，A/D）转换电路转换成数字表示方式。A/D 转换可以产生一个图像的数字表示形式，其中每个像素的颜色和亮度使用若干个位来表示。然后，图像的这种数字形式就可以像使用标准计算机电路处理的任何其他数据那样进行操作了。

图 10-2 中的处理器和系统控制器模块包含着需要与系统其他部分连接的接口电路。处理器控制照相机的操作，它对从 A/D 电路中接收的原始图像数据进行处理，生成用标准格式表示的图像。这些标准格式适合用于计算机、打印机和显示设备。主要使用的格式有：用于无压缩图像的 TIFF（标签图像文件格式），和用于压缩图像的 JPEG（联合图像专家组）。处理过的图像存储在一个大型图像存储设备中，在 8.3.5 节中描述的闪存卡（Flash memory card）就是一种用于图像存储的流行设备。

387
!
388

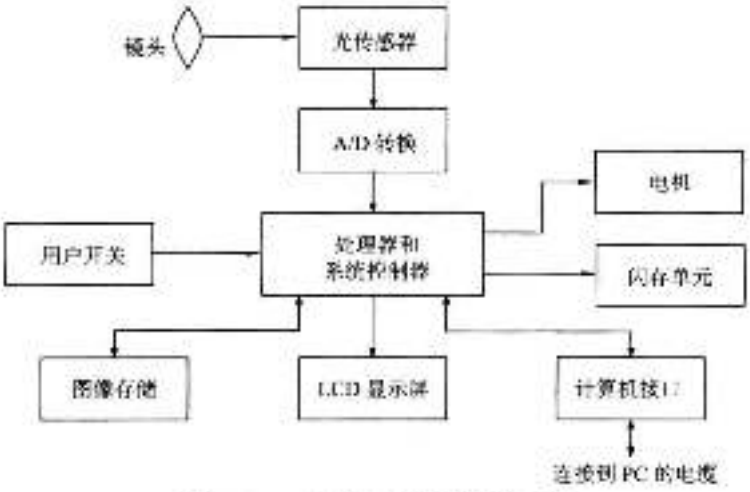



图 10-2　一个简化的数码照相机框图

获取并处理过的图像可以在嵌入在照相机中的液晶显示器（LCD）的屏幕上进行显示。这允许用户决定哪个图像值得保存。可以保存的图像的数量与图像存储器的大小有关，同时也依赖于所选择的图像质量，即每个图像中像素的个数以及压缩程度（JPEG 格式）。

标准接口提供将图像传递到计算机或打印机中的机制。这通常由一根 USB 线来完成。如果使用了闪存卡，图像也可以通过物理地传递闪存卡的形式进行传递。

系统控制器还生成控制调焦装置和闪光部件操作所需要的信号。有一些输入信息是由于用户对开关的操作而形成的。

389

数码照相机中所需要的处理器比之前描述的微波炉应用中的处理器要强大许多，该处理器必须执行复杂的信号处理功能。而且，重要的是该处理器不能消耗太多的能量，因为照相机是一种用电池做电源的设备。通常处理器消耗的电能比照相机中显示器和闪光灯所消耗的电能要少。

### 10.1.3 家用遥测技术

微控制器被用在家庭中大量的嵌入式应用中。在10.1.1节中我们考虑了微波炉的例子，在其他的设备中还可以找到类似的例子，像洗衣机、干燥机、洗碗机、炊具、火炉以及空调。另一个值得关注的例子是可视电话，在电话中嵌入处理器使得电话可以具有多种有用的特征。除了标准的电话特征外，带有嵌入式微控制器的电话可以对家庭中的其他设备进行远程访问。

使用这种电话可以远程执行以下功能：

- 与一个计算机控制的家庭安全系统通信。
- 对火炉或空调保持的温度进行调整，设定一个合适的温度。
- 对提前已放入微波炉中的食品设定启动时间、烹调时间及加热温度。
- 读电表、气表和水表，取代过去服务公司派遣雇员到家中去读这些表的过程。

如果每一个这种类型的设备都是由微控制器控制的，那么所有这些功能都很容易实现。因为只需要在设备中的微控制器与电话中的微处理器之间提供一条有线的或无线的连接即可。用远距离的信号去观察并控制设备的状态通常称为遥测技术（telemetry）。

## 10.2 嵌入式应用中的微控制器芯片

一个微控制器芯片应该是通用的，以便于能够为各种各样的应用提供服务。图10-3给出了一个典型芯片的模块图。其中主要的部分是处理器核（processor core），它是应用微处理器的基础版本。选择在实际中已经证明被广泛接受的微处理器体系结构是较稳妥的，因为针对这样的处理器已经有许多CAD工具、优秀的实例以及大量有助于新产品设计的经验和知识。

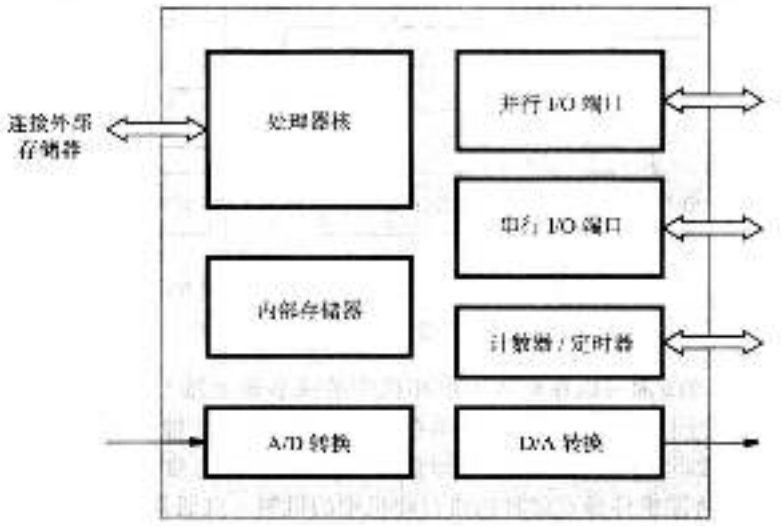



图10-3 微控制器框图

在芯片中包含一些存储器是非常有用的，你会发现这些存储器完全能够满足小型应用中的存储要求。这些存储器中有些必须是RAM类型的，用于存储那些在计算中发生变化的数

据；有些应该是只读类型的，用来存储软件，因为嵌入式系统中通常不包括磁盘。为了满足低容量应用中的成本效益，需要有一种现场可编程类型的 ROM 存储器，而实现这种存储的流行选择是 EEPROM 和闪存。

有几种 I/O 端口可以同时提供并行和串行的接口，这些接口可以容易地实现标准的 I/O 连接。在许多的应用中，需要以可编程的时间间隔生成控制信号。如果微控制器芯片中包含一个定时器电路，这类任务就很容易完成。因为定时器是一个时钟脉冲计数电路，所以它也可以用作事件计数，比如对一个移动的机械臂或旋转轴所产生的脉冲个数进行计数。

在一个嵌入式系统中可能包含一些模拟设备，为了能与这样的设备进行交互，就需要能够将模拟信号转换成数字表示形式，反之亦然。如果嵌入式控制器中包含有 A/D 和 D/A 转换电路，这些也是很容易做到的。

许多嵌入式处理器芯片在市面上可以得到，一些比较好的芯片有：Freescale 的 68HC11 及 68K/ColdFire 系列，Intel 的 8051 和 MCS-96 系列，它们都有 CISC 风格的处理器核，而 ARM 微控制器具有 RISC 风格的处理器。处理器核的特性在本章不是重要的讨论内容，我们强调的是嵌入式应用系统方面的内容，并说明如何将前几章介绍的概念结合起来用到一个完整的嵌入式计算机系统的设计中去。

390
↓
391

## 10.3　一个简单的微控制器

微控制器的输入/输出结构必须有足够的灵活性，以满足不同应用的需求，并能充分利用芯片上的管脚。例如，一个并行端口既可以配置为输入也可以配置为输出。

在这一节中我们讨论一个简单微控制器的基本结构，说明它的一些典型特征。图 10-4 给出了它的框图，其中包含一个处理器核和一些片上存储器。因为片上存储器可能满足不了所有潜在的应用，因此在该芯片的管脚上提供了处理器的总线连接，以使可以添加外部存储器。

它有两个 8 位的并行接口（称为端口 A 和端口 B）以及一个串行接口。该微控制器还包含一个 32 位的计数器/定时器电路，该电路可以用来在设定的时间间隔内产生内部中断，还可以用来作为系统的定时器，对输入线上的脉冲个数计数，生成方波输出信号等。

### 10.3.1　并行 I/O 接口

嵌入式系统的应用在输入/输出接口方面需要相当大的灵活性。所涉及的设备的性质，以及它们如何连接到微控制器上，可以通过考虑图 10-1 所示的微波炉中的一些组件来理解。当门打开时，需要一个传感器来产生一个值为 1 的信号，这个信号被发送到微控制器输入接口的一个引脚上。微波炉前面板的按键也是如此。这些简单的设备每一个产生一位信息。

输出设备也可以用类似的方式来控制。磁控管是由能将它打开或关闭的单一输出线来控制的。风扇和灯也是如此。扬声器也可以通过单一的输出线连接，处理器在该输出线上发送一个具有适当音调频率的方波信号。另一方面，液晶显示器需要并行发送多个数据位。

微控制器输入/输出接口的设计目标之一是尽可能地减少对外部电路的需要。微控制器可能被连接到简单的设备上，其中许多设备只需要一根输入或输出信号线。在大多数情况下，不需要编码或解码。

图 10-4 中每个并行端口都有一个相关联的 8 位数据定向寄存器，可用于将独立的数据线配置为输入或输出。图 10-5 说明了对端口 A 上的某一位的双向控制。如果数据定向触发器中的值为 0，端口引脚 $PA_i$ 就被当作一个输入。在这种情况下，对控制信号 Read_Port 的激活操作会将逻辑值放在处理器总线中数据线 $D_i$ 的端口引脚上。如果数据定向触发器的值被设置为

1，端口引脚作为输出使用。在 Write_Port 信号的控制下，加载到输出数据触发器中的值被放置在引脚上。

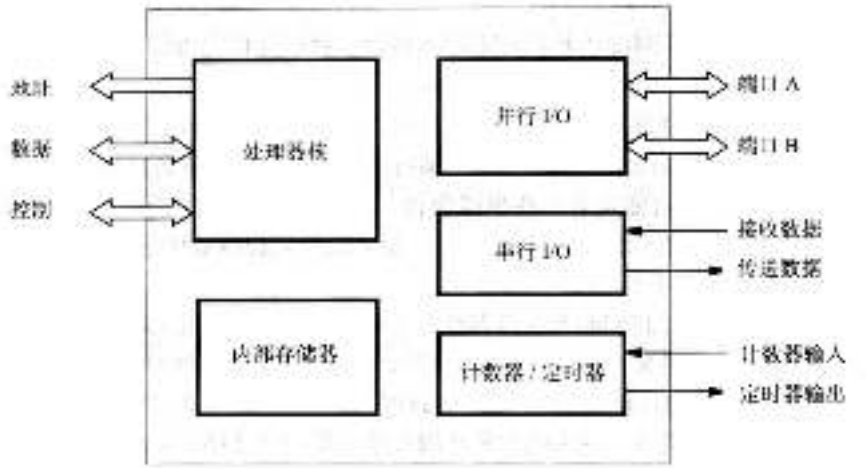



图 10-4 一个微控制器的例子

图 10-5 只显示了控制数据传输方向的接口部分，在输入数据的路径上，没有触发器捕获并保存由连接到相应引脚上的设备所提供的数据信号的值。一个通用的并行接口可能包括两种可能性：一种是直接从引脚读取输入数据，另一种则是将输入数据存储在图 7-11 所示接口的寄存器中，可通过在接口的控制寄存器中设置一位来选择这两种方式中的一种。

392
393

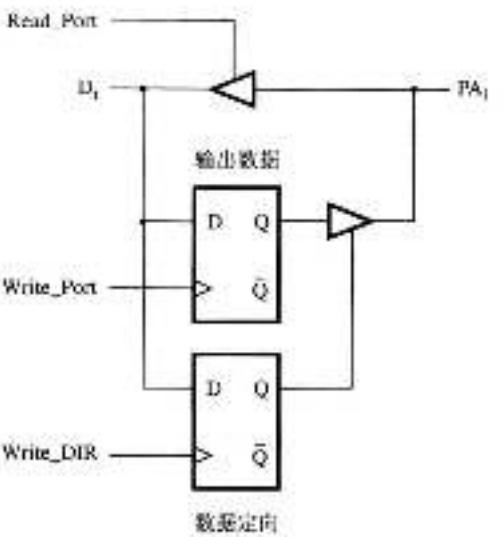



图 10-5 访问图 10-4 的端口 A 中的一位

图 10-6 描述了并行接口中的所有寄存器，以及分配给它们的地址。这里我们任意选择了 32 位地址范围中的高端地址。

状态寄存器 PSTAT 包含着状态标志。当端口 A 上有新数据时，PASIN 标志被置成 1。当处理器通过读取 PAIN 寄存器接收了该数据时，它又被清为 0。当寄存器 PASOUT 中的数据被所连接的设备接收时，PASOUT 标志被置成 1，表示处理器现在可以向 PAOUT 中装入新数据了。接口使用一根单独的控制线（将在下文描述）来向所连接的设备发送新数据可用的信号。当处理器将数据写入 PAOUT 时，PASOUT 标志被清为 0。PBSIN 和 PBSOUT 标志对端口 B 执行同样的功能。

状态寄存器还包含了四个中断标志。当某个中断被允许并且相应的 I/O 动作发生时，就有一个中断标志（比如 IAIN）被设置成 1。中断允许位保存在控制寄存器的 PCONT 中。PCONT 中的某个允许位被设置成 1 就可以允许相应的中断发生。例如，如果 ENAIN=1 并且 PASIN=1，则中断标志 IAIN 被置成 1 并且产生一个中断请求。因此，

$$IAIN = ENAIN \cdot PASIN$$

一个单独的中断请求信号用于接口中所有的端口。作为对某个中断请求的响应，处理器必须检

查中断标志以确定该中断请求的实际来源。

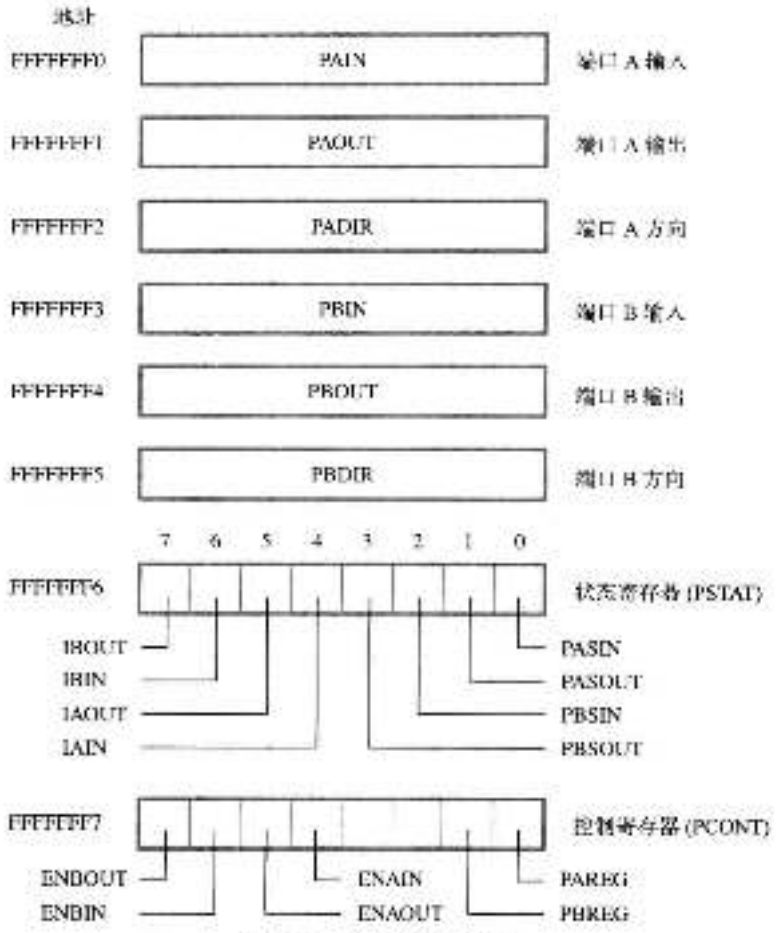



图 10-6　并行接口寄存器

状态和控制寄存器中的信息用于控制向或从连接到端口 A 和端口 B 上的设备传递数据。端口 A 有两条控制线 CAIN 和 CAOUT，它们可以在接口和所连接的设备间提供自动的信号机制。对于输入传递，设备将新数据放置在端口引脚，并通过激活 CAIN 一个时钟周期来表示该动作。当接口电路观测到 CAIN=1 时，它将状态位 PASIN 置成 1。稍后，当处理器读出了这个输入数据时该位被清为 0。这个动作还引发该接口向 CAOUT 连线发送一个脉冲，通知该设备可以向接口发送新的数据了。对于输出传递，处理器将数据写入 PAOUT 寄存器中，接口对其作出响应，将 PASOUT 位清为 0 并向 CAOUT 连线发送一个脉冲以通知设备可以得到新的数据了。当设备接收该数据时，会向 CAIN 连线发送一个脉冲，这又将 PASOUT 置成 1。当一个端口上的所有数据引脚有着同样的方向时，也就是当端口都作为输入或输出时，才可使用这种信号机制。如果选择一些引脚作为输入而其他的作为输出，就不能使用自动机制了，控制线以及状态和控制寄存器中也均不包含有用的信息。在这种情况下，输入数据直接从引脚读取。

控制寄存器的 PAREG 和 PBREG 位分别用来选择端口 A 和 B 的输入操作模式。如果两者都置为 1，就用寄存器来存储输入数据；否则，就要像图 10-5 所示的那样使用引脚连出来的直接通路。作为一个使用直接通路的例子，考虑图 10-1 所描述的微波炉的操作。微控制器将磁控管打开，开始烹饪操作，但是只有当炉门关上时它才可以这样做。一个简单的传感器开关通

过提供一个信号来指示炉门是否是打开的，该信号可以被读取为一个比特的数据。传感器连接到被控制器接口的一个引脚上，使被控制器可以直接读取这个输入的逻辑值，从而确定炉门的状态。

394 / 395

### 10.3.2 串行 I/O 接口

串行接口基于7.4.2节中描述的方案提供了UART（Universal Asynchronous Receiver/Transmitter，通用异步收发器）功能来传递数据。在发送与接收路径中都使用了双缓冲，如图10-7所示。这种缓冲用来正确处理I/O传递中的突发数据流。

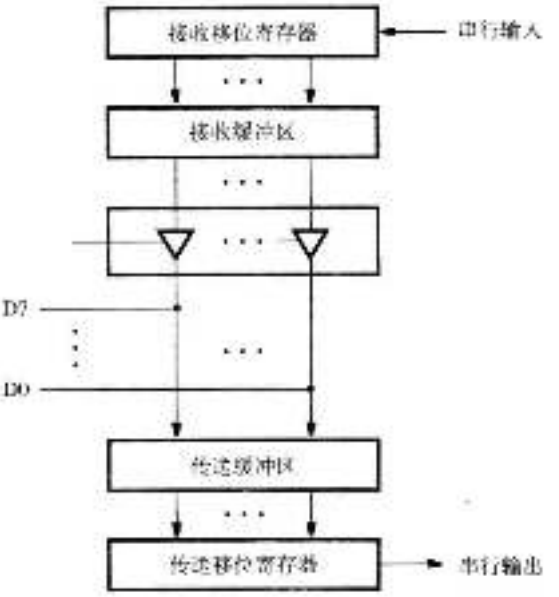



图10-7 串行接口中的接收和传送结构

图10-8给出串行接口的可编址寄存器。输入数据被从8位的接收缓冲区中读出，输出数据被装入8位的传送缓冲区中。状态寄存器SSTAT提供有关接收和传送单元的当前状态信息。当在接收缓冲区中存在有效的数据时，$SSTAT_2$位被置成1，当对接收缓冲区进行读访问时，它被自动清为0。当传送缓冲区为空并可以装入新数据时，$SSTAT_1$位被置成1。这些位与在3.1节中讨论的状态标志KIN和DOUT起着相同的作用。如果在接收过程中产生了一个错误，$SSTAT_2$就被置成1。例如，如果接收缓冲区中的字符在被处理器读出之前被后面接收的字符覆盖，就产生一个错误。状态寄存器中还包含有中断标志。当接收缓冲区已满并且接收区中断是允许的，$SSTAT_4$位被置成1。类似地，当传送缓冲区变为空并且传送区中断是允许的，$SSTAT_5$位被置成1。如果$SSTAT_4$或是$SSTAT_5$等于1，串行接口就引发一个中断。当$SSTAT_2$=1并且错误条件中断是允许时的，$SSTAT_6$被置成1，此时串行接口也引发一个中断。

控制寄存器SCONT用来保存中断允许位。设定$SCONT_{5-4}$位为1或0，分别表示允许或禁止相应的中断发生。该寄存器中还说明如何生成传送时钟，如果$SCONT_3$=0，则传送的时钟与系统（处理器）时钟相同；如果$SCONT_3$=1，则用时钟除法电路获得低频率的传送时钟。

396

串行接口中的最后一个寄存器是时钟除数寄存器DIV。这个32位的寄存器与一个计数器电路相关联。该计数器电路对系统时钟信号进行分频生成串行传送时钟。生成的时钟信号的频率等于系统时钟频率除以该寄存器的内容所得到的频率。装入该寄存器的值被传递到计数器中，然后用系统时钟进行递减计数。当计数减为零时，该计数器将DIV寄存器中的值重新装入。

### 10.3.3 计数器/定时器

32位的递减计数器电路可以作为计数器或定时器使用。该电路的基本操作包括将一个起始值加载到计数器中，然后使用内部系统时钟或是外部时钟信号递减计数器中的内容。这个电路可被编程为当计数器中的内容达到0时引发一个中断。图10-9给出了与计数器/定时器电路相关联的寄存器。计数器/定时器的寄存器CNTM可以被装入一个初值，然后将它传递到计数器电路中。计数器中的当前内容可以通过访问内存地址FFFFFFD4来读出。控制寄存器

CTCON 用于指明计数器 / 定时器电路的操作方式。它提供了一种机制用于开始和停止计数过程以及当计数器内容递减为 0 时允许中断。状态寄存器 CTSTAT 反映该电路的状态。

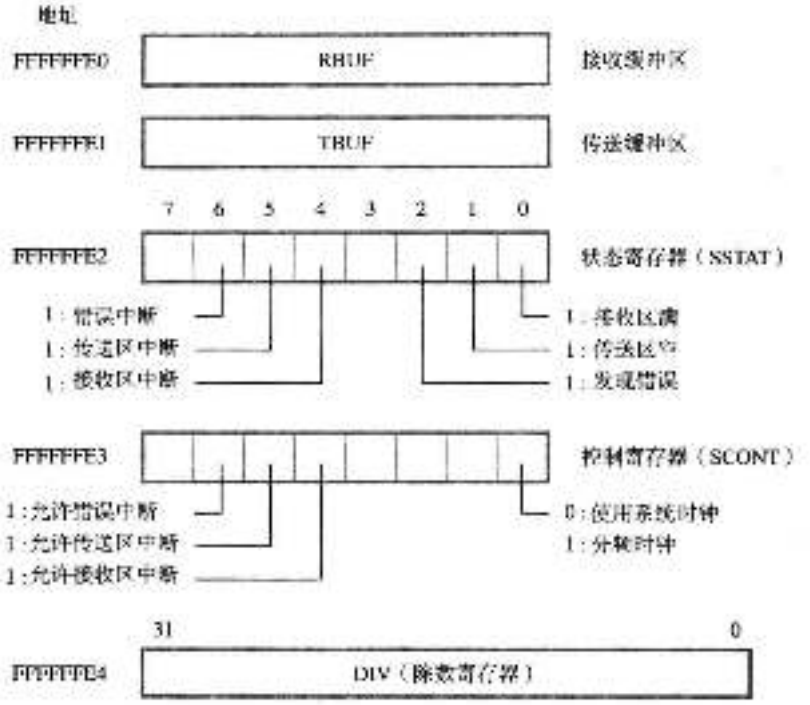



图 10-8　串行接口寄存器

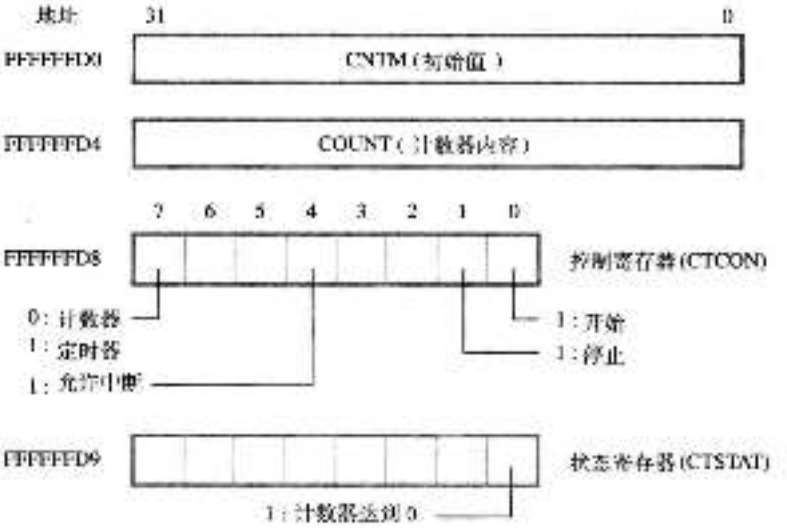



图 10-9　计数器 / 定时器寄存器

**1. 计数器模式**

设定 $CTCON_0$ 位为 0 时选择该电路为计数器模式，将起始值写入寄存器 CNTM 来将其装入计数器中。当 $CTCON_0$ 位被程序指令设置成 1 时计数过程开始。一旦计数开始，$CTCON_0$ 位自动清为 0。计数器随图 10-4 中计数器输入（Counter_in）线上的脉冲递减。当达到 0 时，

计数器电路将状态标志 CTSTAT$_0$ 置成 1，如果此时相应的中断允许位已经被置成 1，就引发一个中断。在下一个时钟脉冲到来时计数器重新装入保存在寄存器 CNTM 中的起始值，然后计数过程继续。当 CTCON$_1$ 位被设置为 1 时计数过程停止。

2. 定时器模式

设定 CTCON$_1$ 位为 1 时选择该电路为定时器模式。这种模式可用于产生周期性的中断，也适合于在图 10-4 中的输出线 Timer_out 上生成一个方波信号。这个过程的开始如同上面对计数器模式的解释一样，当计数器进行递减计数时，输出线上的值保持不变。当减到 0 时，计数器自动用起始值进行重新加载，这时连线上的输出信号反向。因此，输出信号的周期是起始计数值与控制时钟脉冲周期乘积的两倍。在定时器模式下，计数器用系统时钟作递减操作。

397
398

### 10.3.4 中断控制机制

在微控制器例子中的处理器有两个中断请求输入 IRQ 和 XRQ。IRQ 输入用于由微控制器内部 I/O 接口引发的中断；XRQ 输入用于由外部设备引发的中断。如果 IRQ 输入有效，并且中断是被允许的，则处理器执行一个中断服务程序，它用轮询方式确定中断请求源。通过检查状态寄存器 PSTAT、SSTAT 及 CTSTAT 中的标志位来完成这个过程。XRQ 中断的优先级比 IRQ 中断的优先级高。

处理器中的状态寄存器 PSR 有两个中断允许位。如果 PSR$_6$=1，则 IRQ 中断被允许；如果 PSR$_7$=1，则 XRQ 中断被允许。当处理器接收一个中断时，它将在中断服务程序执行前清除相应的 PSR 位，禁止同一优先级上的其他中断产生。这里要用到向量中断方法，其中对应于 IRQ 和 XRQ 中断的向量分别存放在存储单元 0x24 和 0x28 中。每个向量中包含着相应的中断服务程序的第一条指令的地址。这个地址会自动装入程序计数器 PC 中。

在处理器中有一个链接寄存器 LR，它如同在 2.7 节中讲述的那样，用于子程序的链接。一条子程序调用指令将使程序计数器中修改后的内容（即需要的返回地址）在转移到子程序的第一条指令前被保存到 LR 中。还有一个寄存器 IRA，它在接收中断请求时保存返回地址。这时，除了将返回地址保存在 IRA 中以外，处理器状态寄存器 PSR 中的内容还应保存在处理器的寄存器 IPSR 中。

从子程序返回是通过执行一条 ReturnS 指令完成的，该指令将 LR 中的内容传递到 PC 中。从中断中返回是通过执行 ReturnI 指令完成的，该指令将 IRA 和 IPSR 中的内容分别传递到 PC 和 PSR 中。因为只有一个 IRA 和 IPSR 寄存器，嵌套式中断可以通过在中断服务程序中使用指令，将这些寄存器中的内容保存在堆栈中的方法来实现。注意，如果中断服务程序调用一个子程序，那么中断服务程序必须保存 LR 的内容，因为中断可能会发生在处理器执行另一个子程序的时候。

### 10.3.5 编程实例

前面已经介绍了微控制器的硬件，现在我们来考虑当微控制器的接口连接到 I/O 设备时所引起的一些软件问题。程序可以用汇编语言或高级语言来编写，后一种选择在大部分应用中更可取，因为这样所需要的编码更容易生成和维护，而且开发时间较短。在本章的例子中，我们将使用 C 编程语言。

399

本节中的例子是一些最基本的应用，旨在说明实现方法的可能性。在 10.4 节中，我们将给出一个完整且比较复杂的应用程序实例。

**例 10.1**

考虑以下任务，使用微控制器监测一些机械设备的状态。这些状态信息由四根电线上提供的二进制信号获得。该状态的 16 个可能值将在图 3-17 所示的七段显示器上显示为一个 16 进制数字。

使用图 10-4 到 10-6 所示的并行接口可以实现所需要的操作，4 根输入线连接到端口 A 的引脚上，7 根数据线连接到端口 B 的七段显示器上。然后，数据定向寄存器 PADIR 和 PBDIR 必须将端口 A 和 B 分别配置为输入和输出，输入数据就可从端口 A 的引脚上直接读出。

图 10-10 给出了一个可能的程序。Define 语句用来将需要的地址常量与指针的符号名关联起来。注意，PAIN 指针被声明为可变的（volatile）。这样做是因为程序只能读相应地址的内容，而不能向这个地址写任何数据，也不能将这个地址与一个特定的值相关联。一个优化的编译器可能会删除那些看起来没有影响的程序语句，这些语句中包含一些变量，但它们的值从未被改变过。因为寄存器 PAIN 的内容改变是受程序外部因素影响的，所以有必要将这个事实告诉编译器。编译器不会删除那些包含可变变量的语句。

```
/* 定义寄存器地址*/
#define   PAIN     (volatile unsigned char *) 0xFFFFFFF0
#define   PADIR    (volatile unsigned char *) 0xFFFFFFF2
#define   PBOUT    (volatile unsigned char *) 0xFFFFFFF4
#define   PBDIR    (volatile unsigned char *) 0xFFFFFFF5
#define   PCONT    (volatile unsigned char *) 0xFFFFFFF7

/* 十六进制到七段的转换表*/
unsigned char table[16] = { 0x40, 0x79, 0x24, 0x30, 0x19, 0x12,
    0x02, 0x78, 0x00, 0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E };
unsigned int current_value;

void main()
{
    /* 初始化端口 A 和 B */
    *PADIR = 0x0;                          /* 将端口 A 配置为输入端口*/
    *PBDIR = 0xFF;                         /* 将端口 B 配置为输出端口*/
    *PCONT = 0x0;                          /* 直接从引脚读取输入 */

    /* 读取并显示数据 */
    while (1)                              /* 连续循环 */
    {
        current_value = *PAIN & 0x0F;      /* 从端口 A 读取输入 */
        *PBOUT = table[current_value];     /* 将字符发送到端口 B */
    }
}
```

图 10-10　例 10.1 的 C 程序

将一个十六进制数字转换成相应的七段模式时要使用一个表。本程序使用了一个连续的循环去读取和显示新的数据。但在实际的应用中，不可能以这种方式使用循环，因为系统中还可能包含其他的任务。我们此处使用连续的循环仅仅是为了使例子简单化。

**例 10.2**

现在考虑一个能以位串格式发送信息的 I/O 设备的情况，这种位串格式的信息可以由图 10-7 和图 10-8 中所描述的串行接口处理。该设备发送 8 位的信息，将在两个图 3-17 那样的七段显示器上显示为两个十六进制数字。我们将该 I/O 设备连接到串行接口上，七段显示器连接到并行端口 A 和 B 上。因此，A 和 B 都必须被配置为输出端口。

图 10-11 给出了一个可能的程序。使用轮询法来确定串行接口的接收缓冲区中的新数据什么时候可用。$SSTAT_1$ 位作为轮询的标志位，如 10.3.2 节中所描述的，当从缓冲区中读取数据时这个位被清零。

```
/* 定义寄存器地址 */
#define   RBUF     (volatile unsigned char *) 0xFFFFFFE0
#define   SSTAT    (volatile unsigned char *) 0xFFFFFFE2
#define   PAOUT    (volatile unsigned char *) 0xFFFFFFF1
#define   PADIR    (volatile unsigned char *) 0xFFFFFFF2
#define   PBOUT    (volatile unsigned char *) 0xFFFFFFF4
#define   PBDIR    (volatile unsigned char *) 0xFFFFFFF5

/* 十六进制到七段的转换表 */
unsigned char table[16] = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12,
    0x02, 0x78, 0x00, 0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E};
unsigned int current_value, low_digit, high_digit;

void main()
{
    /* 初始化并行端口 */
    *PADIR = 0xFF;                          /* 将端口 A 配置为输出端口 */
    *PBDIR = 0xFF;                          /* 将端口 B 配置为输出端口 */

    /* 读取并显示数据 */
    while (1)                               /* 连续循环 */
    {
        while (*SSTAT & 0x1) == 0);         /* 等待新数据 */
        current_value = *RBUF;              /* 读取 8 位的值 */
        low_digit = current_value & 0x0F;
        high_digit = (current_value >> 4) & 0x0F;
        *PAOUT = table[low_digit];          /* 将两个数字查表 */
        *PBOUT = table[high_digit];         /* 到七段显示器上 */
    }
}
```

图 10-11 例 10.2 的 C 程序，使用轮询方式读取输入数据

图 10-12 显示了用中断的方式访问新数据的程序。回顾 10.3.4 节，IRQ 中断向量在存储单元 0x20 中，中断服务程序的地址被装入这个单元。将 $SCONT_4$ 位置为 1 以允许接收区中断。为了使处理器响应 IRQ 中断，处理器状态寄存器 PSR 的第 6 位必须被置为 1。因为 PSR 不是可寻址空间中的一个单元，所以需要使用 asm 指示符将下列汇编语言指令嵌入到 C 程序中：

```
MoveControl   PSR, #0x40
```

此外，需要在目标程序中包含中断返回指令，以保证能够正确返回到被中断的程序。编译器将插入这条指令，因为 inserv 函数的定义中包含中断（interrupt）这个关键词。在高级语言中处理中断的方式在 4.6 节中作了介绍。

```
#define   RBUF     (volatile unsigned char *) 0xFFFFFFE0
#define   SCONT    (volatile unsigned char *) 0xFFFFFFF3
#define   PAOUT    (volatile unsigned char *) 0xFFFFFFF1
#define   PADIR    (volatile unsigned char *) 0xFFFFFFF2
#define   PBOUT    (volatile unsigned char *) 0xFFFFFFF4
#define   PBDIR    (volatile unsigned char *) 0xFFFFFFF5
#define   IVECT    (volatile unsigned int *) 0x20
```

图 10-12 例 10.2 的 C 程序，使用中断方式读取输入数据

```
/* 十六进制到七段的转换表 */
unsigned char table[16] = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12,
    0x02, 0x78, 0x00, 0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E};
unsigned int current_value, low_digit, high_digit;

interrupt void intserv();

void main()
{
    /* 初始化并行端口 */
    *PADIR = 0xFF;                          /* 将端口 A 配置为输出端口 */
    *PBDIR = 0xFF;                          /* 将端口 B 配置为输出端口 */

    /* 初始化中断机制 */
    *IVECT = (unsigned int *) &intserv;     /* 设置中断向量 */
    asm ("MoveControl    PSR, #0x40");      /* 响应 IRQ 中断 */
    *SCONT = 0x10;                          /* 允许接收区中断 */

    while (1);                              /* 连续循环 */
}

/* 中断服务程序 */
interrupt void intserv()
{
    current_value = *RBUF;                  /* 读取 8 位的值 */
    low_digit = current_value & 0x0F;
    high_digit = (current_value >> 4) & 0x0F;
    *PAOUT = table[low_digit];              /* 将两个数字发送 */
    *PBOUT = table[high_digit];             /* 到七段显示器上 */
}
```

图 10-12　（续）

## 10.4　反应定时器——一个完整的实例

前面已经介绍了微控制器的基本特征。现在说明如何将它用在一个简单的嵌入式系统中，该系统实现了一个容易理解的、体现了反应系统（reactive system）特征的任务。我们需要设计一个“反应定时器”，它可以用于测量人们对于视觉刺激的反应速度。设计思想是需要微控制器去打开一盏灯，然后测量被测者按下按钮键关掉灯所经历的反应时间。该系统的细节和操作描述如下：

- 有两个手动的按钮键 Go 和 Stop、一个发光二极管（LED）和一个三位数的七段显示器。
- 按下 Go 键，该系统被激活。
- 激活后，七段显示器被设置成 000，并且 LED 是关闭的。
- 经过三秒的延时后，LED 被打开并且计时过程开始。
- 当 Stop 键被按下时，计时过程停止，LED 关闭，并且将经历的时间显示在七段显示器上。
- 经历的时间被计算出来，并按百分之一秒的格式显示。由于显示器只有三位数，所以假定经历的时间将少于 10 秒。

400 / 403

图 10-13 描绘了能实现所需反应定时器的硬件。这里的微控制器提供了除输入键和输出显示以外的所有硬件组件。跟上一节中的例子不同，我们假设每一个七段显示装置都与一个 BCD-七段译码器电路关联起来，从而使微控制器只需为每个显示的数字发送一个四位的 BCD 码。我们的微控制器没有足够的并行端口来同时向三个显示器发送编码的七段信号。

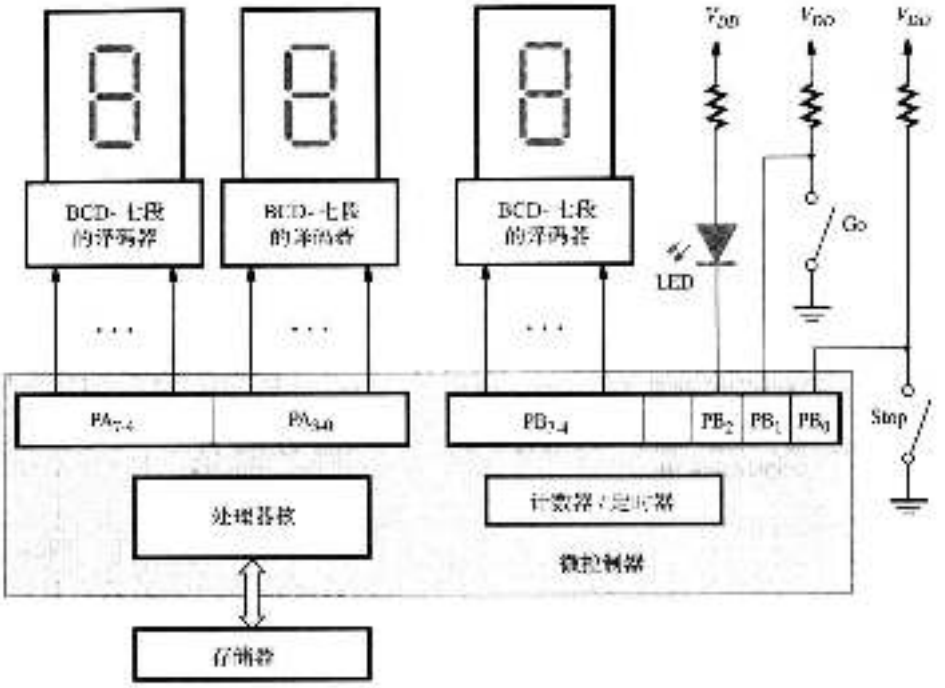



图 10-13 反应定时器电路

我们将使用两个并行端口 A 和 B 来完成所有的输入/输出功能。所显示时间的两个最高有效 BCD 位连接到端口 A 上，最低有效位连接到端口 B 的高四位上。按键和 LED 连接到端口 B 的最低三位上。计数器/定时器电路用于测量经历的时间，它由系统时钟驱动，我们假定
404 系统时钟具有 100MHz 的频率。

实现这个任务的程序可以基于以下方法完成：

- 用户想开始测试的意图可通过一个等待循环来监视，在循环中反复查询 Go 键的状态。
- 当观察到 Go 键已被按下，也就是发现 $PB_1 = 0$ 时，再延时三秒钟之后打开 LED。
- 计数器设置成初始值 0xFFFFFFFF，在每个时钟脉冲中递减计数的过程开始。
- 用等待循环去查询 Stop 键的状态，探测用户是何时按下它来做出反应的。
- 当 Stop 键被按下时，LED 关闭，停止计数并计算经历的时间。
- 将测量出的延迟时间转换成一个 BCD 数，并发送到七段显示器上。

微控制器中各种 I/O 寄存器的地址如图 10-6 到图 10-9 所示，程序必须将端口 A 和 B 配置成图 10-13 所示连接所需要的那样。端口 A 的所有位和端口 B 的高四位被配置成输出。端口 B 的低三位中的 $PB_0$ 和 $PB_1$ 用做输入，而 $PB_2$ 是一个输出。在两个端口中不需要使用控制信号，因为输入设备是由直接驱动端口线的按钮键构成的，而输出设备是一个显示器，当驱动显示器的端口引脚上的信号有任何改变时，该显示器就会显示出来。

我们将说明如何用 C 程序语言来实现所需的应用。程序执行下列任务。当 Go 键被按下后，用定时器实现三秒钟的延时。由于计数器/定时器电路的时钟为 100MHz，计数器被初始化为十六进制值 11E1A300，它相当于十进制值 300 000 000。当 $CTCONT_0$ 位被置成 1 时递减计数过程开始。当计数值达到 0 时，LED 被打开，开始反应时间测试，并且计数器被置成 0xFFFFFFFF。当检测到 Stop 键被按下时，设置 $CTCONT_1 = 1$，停止计数过程。总计数值是这样计算的：

总计数值 − 0xFFFFFFFF − 当前计数值

因为这是时钟周期的总数，按百分之一秒计算的实际时间是：

实际时间＝总计数值 / 1000000

这个二进制数可以转换成一个十进制数：首先用这个二进制整数除以 100，以生成最高位有效数。余数再除以 10，生成下一位有效数，最后的余数是最低位有效数。

图 10-14 给出了一个可能的程序，按要求将端口 A 和 B 进行配置并关闭显示器和 LED 后，该程序不断地查询引脚 $PB_1$ 上的值。在 Go 键被按下，$PB_1$ 变成 0 后，插入一个三秒钟的延时。然后 LED 被打开并且反应计时过程开始。另一个轮询操作用于等待 Stop 键被按下。当这个键被按下时，LED 关闭，计数器停止，并且读出计数器中的内容。就像上面解释的那样完成经历时间的计算，并将其转换成十进制数。由此产生的三个 BCD 数字，根据图 10-13 所 405
描述的结构，被写入端口数据寄存器。

```
/* 定义寄存器地址 */
#define PAOUT (volatile unsigned char *) 0xFFFFFFF1
#define PADIR (volatile unsigned char *) 0xFFFFFFF2
#define PBIN (volatile unsigned char *) 0xFFFFFFF3
#define PBOUT (volatile unsigned char *) 0xFFFFFFF4
#define PBDIR (volatile unsigned char *) 0xFFFFFFF5
#define CNTM (volatile unsigned int *) 0xFFFFFFD0
#define COUNT (volatile unsigned int *) 0xFFFFFFD4
#define CTCON (volatile unsigned char *) 0xFFFFFFD8
void main()
{
    unsigned int counter_value, total_count;
    unsigned int actual_time, seconds, tenths, hundredths;

    /* 初始化并行端口 */
    *PADIR = 0xFF;                  /* 配置端口 A*/
    *PBDIR = 0xF4;                  /* 配置端口 B*/
    *PAOUT = 0x0;                   /* 关闭显示器 */
    *PBOUT = 0x4;                   /* 和 LED*/

    /* 开始测试 */
    while (1)                       /* 连续循环 */
    {
        while ((*PBIN & 0x2) != 0);     /* 等待 Go 键被按下 */

        /* 等待二秒钟然后打开 LED */
        *CNTM = 0x11E1A300;         /* 设置定时器值为 300 000 000*/
        *CTCONT = 0x1;              /* 启动定时器 */
        while ((*CTSTAT & 0x1) == 0);   /* 等待直到定时器达到 0*/
        *PBOUT = 0x0;               /* 打开 LED*/

        /* 初始化计数过程 */
        counter_value = 0;
        *CNTM = 0xFFFFFFFF;         /* 设置起始计数器的值 */
        *CTCONT = 0x1;              /* 开始计数 */

        while ((*PBIN & 0x1) != 0);     /* 等待 Stop 键被按下 */

        /* Stop 键被按下，停止计数 */
        *CTCONT = 0x2;              /* 停止计数器 */
        *PBOUT = 0x4;               /* 关闭 LED*/
        counter_value = *COUNT;     /* 读取计数器的内容 */
```

图 10-14　用于反应定时器的 C 程序

```
        /* 计算器的计数值 */
        total_count = (0xFFFFFFFF - counter->value);

        /* 将计数转换为时间 */
        actual_time = total_count/1000000;      /* 以 1/100 秒为单位的时间 */
        seconds = actual_time/100;
        tenths = (actual_time - seconds * 100)/10;
        hundredths = actual_time - (seconds * 100 + tenths * 10);

        /* 显示经过的时间 */
        *PAOUT = ((seconds << 4) | tenths);
        *PBOUT = ((hundredths << 4) | 0x4);   /* 保持 LED 关闭 */
    }
}
```

图 10-14 （续）

## 10.5 传感器与执行器

嵌入式计算机与其所处的环境紧密交互。到目前为止，我们已经使用开关和简单的显示设备来说明这种相互作用。在实际的应用中会使用多种其他的设备。为了控制一个机械系统，需要使用能在机械领域和计算机中使用的数字电子领域之间提供接口的设备。它能够使计算机感知或者监控被控制系统的状态，还会引起由计算机控制的操作的执行。这样的设备通常被称为传感器（sensor）和执行器（actuator）。它们被统称为换能器（transducer）。在这一节中，我们将展示几个传感器的例子来说明一些基本原则。

### 10.5.1 传感器

根据控制系统的性质，嵌入式计算机可能需要监控很多参数。考虑汽车中的巡航控制系统。它的目的是将速度维持在尽可能地接近所需值的水平，无论道路是水平的还是上坡或者下坡。仅简单地将汽油门保持在一个固定的位置是不够的。控制系统必须不断地测量汽车的速度并调整油门开度使其维持所需的速度。因此，需要一个传感器来测量汽车的速度并用适用于计算机的数字形式表示出来。然后计算机能够确定油门开度是否需要调整，并向执行器发送命令以促使其发生。汽车中还有许多其他的传感器，包括测量各种液体（如汽油、石油和发动机冷却液）水平的设备。传感器可以监控冷却液的温度、轮胎的气压和电池的电压。

406
|
407

有一些传感器是比较简单的，例如那些检测是否有人或动物沿着关闭的车库门行走的传感器。其他的传感器可能会非常复杂。下面给出几个传感器的例子。

#### 1. 位置传感器

考虑一台嵌入式计算机，它控制机器人的手部运动。手部位置是由肩、肘和腕关节的角度位置来确定的。为了监控手部的位置，计算机必须要测量这些关节中每一个关节的角度位置。

测量转轴相对于其外壳的角度位置的一种简单传感器是电位计。它包括一个环绕在连接到外壳上的圆形底座周围的电阻，和一个连接到转轴上的滑块触点。转轴的位置是被监控的。当转轴旋转时，在电阻上的接触点会改变。当如图 10-15 所示的那样连接到一个电源上时，接触点任一侧的两部分电阻构成一个分压器。这个电路的输出电压由下式给出：

$$V_{out} = \frac{R_2 V_{in}}{R_1 + R_2}$$

这个电压被送到一个 A / D 转换电路中，以便将模拟值转换成数字表示形式。图 10-3 中的微控

制器包含一个可用于这种情况中的 A / D 转换器。

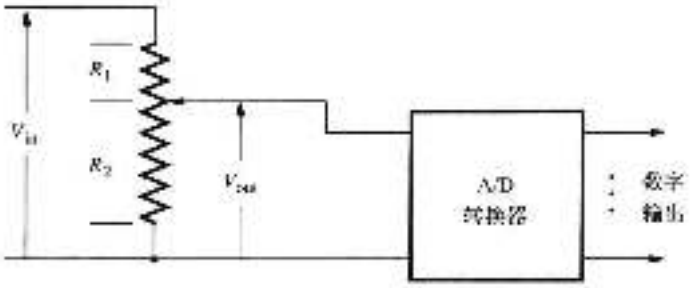



图 10-15　使用分压器的传感器

有一些传感器可直接产生一个数字输出。例如，一个编码的盘可能连接到一个转轴上，该转轴的位置被监控着。盘上的编码是以透明和不透明区域的形式组织在同心的环形区域中，如图 10-16 所示。盘被定位在发光二极管和光电探测器之间，每一个环形区域有一对发光二极管和光电探测器。每一个光电探测器连接到一个电路上，该电路在检查到光时就产生一个逻辑信号 1 否则就产生 0。这样，当盘旋转时，图中所示的两个光电探测器就产生 2 位的二进制数 00，01，10 和 11，表示该转轴的角度位置。 408

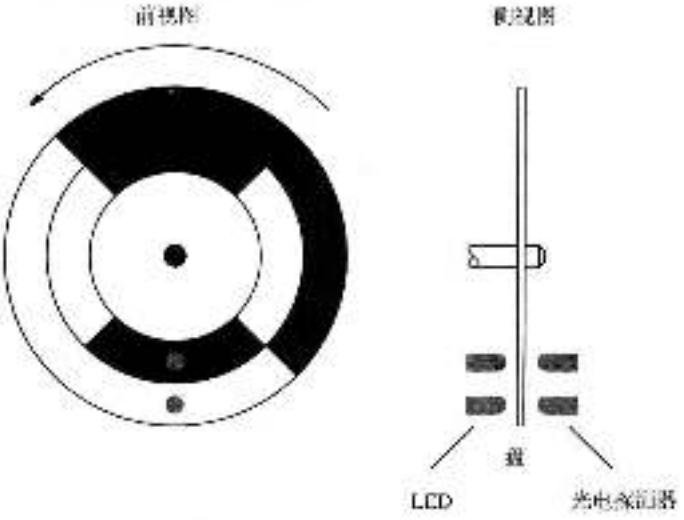



图 10-16　一种光学位置传感器

### 2. 温度传感器

许多类型的传感器利用材料在温度或湿度变化时其特性发生的变化，或者由于拉伸、压缩或弯曲造成变形的结果来探测。一个导体的电阻随温度的变化而变化，其效果在某些材料中比其他材料更明显。因此，使用一个类似于图 10-15 中的分压器，用对温度变化具有较高敏感度的材料制成电阻 $R_2$，并将其暴露在待测量温度的环境中，由此可以构造一个温度传感器。当温度改变时，电压 $V_{out}$ 发生改变。这样，在将 $V_{out}$ 转换为数字表示形式后，该值可以被计算机读取以作为对温度的测量值。其他类型的温度传感器使用电容器作为传感元件。电容的电容量随着构造电容所使用的电介质（绝缘材料）的特性变化而变化。一些陶瓷材料也很适合用于温度传感器中。

### 3. 压力传感器

应变计是一种广泛使用的传感器，它是由一个放在软膜上的电阻组成，用于测量电阻中
409 的变化。当薄膜被拉伸时，电阻值发生改变。

假设我们希望监控一个封闭容器内（如汽车轮胎）的空气压力。压力传感器可以构造成一个其一端连接有一个膜片的小圆筒以及一个安装在膜片上的应变计的形式。将圆筒插入到容器中，这样膜片阻止空气流出这个容器。当该容器被加压时，膜片拉伸，应变计的电阻改变。可以用类似于图10-15中的电路来测量电阻的改变。

### 4. 速度传感器

将一个小的发电机连接到旋转轴上，能产生一个与旋转速度成正比的电压。发电机的输出电压可以转换成数字形式，并被计算机读取以作为对速度的测量值。这种装置通常被称为转速计。

另一种方法是当转轴旋转时产生电脉冲。例如，将齿形盘连接到转轴上。当盘旋转时，其每个齿通过一个适当放置的电磁或光传感器都会产生一个电脉冲。通过对一个固定的时间周期中所产生的脉冲数量进行计数，或者测量连续脉冲之间的时间延迟即可确定旋转速度。

## 10.5.2 执行器

执行器是一种在接收到命令后使得机器部件移动的装置。图10-17说明了构造各种执行器的基本原则。电线被缠绕在圆筒周围形成一个线圈，圆筒中包含一个叫做电枢（armature）的可动铁芯。这种结构被称为螺线管。电枢可使用像铁这样的磁性材料来制造，通过一个弹簧将其保持在部分插入圆筒中的位置，如图所示。当有电流流经线圈时，它产生一个磁场，将电枢拉入圆筒中。当电流停止流动时，电枢被弹簧拉回其静止位置。电枢的运动可以用来打开水阀，或关闭电机的开关。这种结构还可用在汽车的起动电机中。

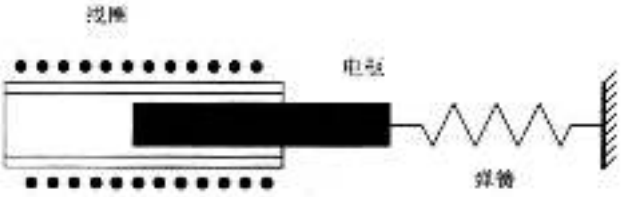


图10-17 螺线管执行器

电机和传感器组合起来，可用于将一个对象移动到所需的位置。当电机打开，这个对象
410 就开始移动。然后计算机就可以使用传感器来重复检测是否已到达所需的位置。在具有自动对焦功能的照相机中，电机使调焦装置旋转。与此同时，计算机使用图像分析算法来检测图像是否被正确地对焦。当达到最佳的焦点时，它就停止电机。

另一种有用的执行器是步进电机，该电机能响应脉冲形式的命令使转轴旋转。每一个脉冲使电机轴旋转一个预定量，如旋转几度。步进电机在需要精确控制对象（如机器人的手臂）位置的应用中非常有用。

上面介绍的传感器和执行器构成了计算机及其所嵌入的物理环境之间的接口。它们向计算机提供了监控各种系统组件的状态和促使所需的动作发生的能力。传感器为计算机提供了关于系统状态的一些信息，而执行器则接收命令并执行这些命令所指定的功能。计算机的任务是实现应用所需的控制算法，它同时也接收来自用户的命令并将信息显示给用户。

## 10.5.3 应用实例

现在我们简要地介绍两个应用，来说明如何使用传感器和执行器。

1. 家庭供暖控制

家中的暖气和空调是由一个叫做恒温器的装置来控制的。让我们来看一下供暖炉的控制。在恒温器中加入一个微控制器，可以实现像在一天的不同时间或周末自动改变所需的温度这样的功能。这样的恒温器包含了：

- 一个温度传感器
- 产生输出信号使供暖炉打开或关闭的电路
- 一个用来跟踪一天时间和一周时间的定时器
- 可供用户输入所需设置值的按钮键
- 一个显示屏

在这里我们将只讨论温度控制机制。假设我们使用了 10.5.1 节中描述的电阻温度传感器，并假设其输出电压连接到微控制器中的 A/D 转换器上。螺线管用来激活一个能将炉子打开或关闭的开关。控制算法由微控制器中的一个程序实现。假设所需的温度被设置为 $T$ 度，并允许房间的实际温度偏离该设置值一个较小的偏移量 $\Delta T$（高于或低于所需的值）。然后，可通过反复执行以下动作来实现控制任务：

1）对温度传感器进行采样以获得表示温度的输入电压。

2）如果温度低于 $T-\Delta T$，则打开供暖炉。

3）如果温度高于 $T+\Delta T$，则关闭供暖炉。

411

该计算机程序的其余部分处理这些任务：使用定时器来跟踪时间，通过按钮键接收用户的输入，将信息发送到显示器上。

2. 巡航控制

许多汽车中的巡航控制系统需要以下组件：

- 一个测量汽车驱动轴速度的传感器
- 一个控制汽油门位置的执行器
- 驾驶员用来激活系统和设置所需速度的输入按钮

这个应用与家庭供暖系统的不同点在于它所控制的参数——油门的位置——是在完全关闭或者完全打开这两个极端之间连续变化。在上一个例子中使用的简单开/关控制算法在这里是不合适的。当控制一个像油门位置这样的变化量时，需要更复杂的算法。该算法的讨论已经超过了本书的范围。然而，所需的动作可简单描述如下。计算机反复测量汽车的速度并与原来设置的值进行比较。比较的差值为误差 $e$，它可能为正或者为负。根据 $e$ 的值，计算机对油门位置作微小调整。我们的目标是维持一个尽可能接近 0 的 $e$ 值。实际上，计算机模仿了手动操控汽车并保持恒定速度的驾驶员的决策和动作。

## 10.6　微控制器系列

我们已经在 10.3 节给出了一个微控制器的例子，现在来简要地讨论一些商用的微控制器芯片。许多嵌入式应用不需要功能很强的处理器，显然，10.1.1 节中讨论的微波炉就不需要功能强大的微控制器，因为它需要的计算相当简单，并且对反应时间的要求不高。针对这样的应用最好是使用这样一种芯片，它包括一个简单的处理器，但同时还含有足够的存储器和 I/O 资源以实现所有的控制器功能。在 10.1.2 节中讨论的数码照相机就有高得多的计算需求，因此它就需要使用一个功能比较强的处理器。

处理器的性能可以用访问存储器数据时能够并行处理多少个数据位来衡量。目前最强大的微控制器可能是基于一个 32 位的处理器构成的，它具有 32 位宽的数据总线。某些基于

ARM 体系结构的微控制器就是这样的。微控制器也可能有一个 32 位内部结构的处理器，但只有 16 位宽的数据总线连到存储器上。微控制器 Freescale 68K/ColdFire 系列就是这样的例子。一些最流行的微控制器是 8 位的芯片。它们非常便宜，但是其功能足以满足大量嵌入式应用的
412 需要。还有更小的 4 位的芯片，由于它们的简单性和极低的成本，所以也极具吸引力。

### 10.6.1 基于 Intel 8051 的微控制器

在 20 世纪 80 年代初，Intel 公司推出了一款称为 8051 的微控制器芯片。这个芯片具有 Intel 8080 微处理器系列的基本体系结构，它使用 8 位的芯片，可用于通用的计算应用中。8051 芯片得到了快速的普及，并已成为在实际中使用最为广泛的芯片之一。它有四个 8 位的 I/O 端口、一个 UART 和两个 16 位的计数器 / 定时器电路。它还包含有 4K 字节的 ROM 和 128 字节的 RAM 存储器。该芯片的 EPROM 版本中包含有 4K 字节的 EPROM 而不是 ROM，它被命名为 8751。

还有许多基于 8051 体系结构的芯片；它们进行了不同程度的改进。例如，8052 芯片有 8K 字节的 ROM 和 256 字节的 RAM，还有一个附加的计数器 / 定时器电路。它的 EPROM 版本被称为 8752。

8051 体系结构由 Intel 公司开发。后来，许多其他半导体制造商生产的芯片要么与 8051 系列的芯片相同，要么有一些增强的功能，但在其他方面与 8051 完全兼容。

### 10.6.2 Freescale 微控制器

在 20 世纪 80 年代，Motorola 在微处理器芯片的制造商中占据着主导地位。它们最流行的 8 位微处理器成为微控制器的基础。Freescale 半导体公司是 Motorola 在这一领域的继任者。基于不同的处理器核，Freescale 生产了各种各样的微控制器。

#### 1. 68HC11 微控制器

Motorola 最流行的 8 位微处理器是 6800 和 6809。后来推出的 68HC11 微控制器芯片实现了 6800 指令集的超集。它有五个可以用于各种目的的 I/O 端口。I/O 结构中包括两个串行接口。它还有能够在若干不同模式下进行操作的计数器 / 定时器电路。

68HC11 芯片中的存储器总量范围从原来芯片中一个 8K 字节的 ROM、一个 512 字节的 EEPROM 和一个 256 字节 RAM 到之后芯片中一个 12K 字节的 ROM、一个 512 字节的 EEPROM 和一个 512 字节 RAM。

#### 2. 68K 微控制器

68K 系列的微控制器是基于 32 位的 68000 处理器核构成的。为了减少引脚的数量，外部数据总线只有 16 位宽。这一系列的芯片包含有并行和串行端口、计数器 / 定时器以及 A/D 转换电路。片上存储器的总量因芯片的不同而不同。比如，68376 芯片有一个 8K 字节的 EEPROM 和一个 4K 字节的 RAM。

#### 3. ColdFire 微控制器

68000 指令集体系结构为附录 C 中介绍的 32 位 ColdFire 处理器和被称为 ColdFire 嵌入式处理器的 MCF5xxx 微控制器提供了基础。它们的显著特点是使得性能得到大大提高的流
413 水线结构。外部数据总线的宽度可以是 16 位或者 32 位的，这取决于所选择的微控制器芯片。ColdFire 处理器核也是为了用于片上系统（system-on-a-chip）环境而设计的，片上系统将在第 11 章中讨论。

4. PowerPC 微控制器

Freescale 的高端 32 位微处理器系列被称为 PowerPC，它是基于 RISC 风格的体系结构建立的。目前也可以见到以这种处理器体系结构建立的微控制器，比如包括 MPC5xx 系列在内的芯片。

### 10.6.3　ARM 微控制器

在附录 D 中给出的 ARM 体系结构对于嵌入式系统是具有吸引力的，因为嵌入式系统需要可靠的计算能力，并且成本和功耗相对比较低。ARM 处理器设计的一个主要目标是使其适合于片上系统的环境。ARM 微控制器也可以作为一个单独的芯片使用。

已经有一系列的 ARM 处理器核用于了嵌入式应用中，包括 ARM6，ARM7，ARM9、ARM10 和 ARM Cortex。基本的 ARM 体系结构使用 32 位的组织和一个所有指令都是 32 位长的指令集。还存在另一种版本，称为 Thumb（拇指）型，它使用 16 位的指令和 16 位的数据传输。Thumb 版本使用的是 ARM 指令的子集，该子集被编码成适合 16 位的格式。它包含的寄存器也比 ARM 体系结构少一些。Thumb 的优势是只需要使用相当小的存储器来存储由高度编码的 16 位指令构成的程序。在执行时，每个 Thumb 指令被扩展成正常的 32 位 ARM 指令。因此，一个 Thumb 型的 ARM 核中除了正常的电路外，还包含有一个 Thumb 解压缩装置（Thumb decompressor）。

## 10.7　设计问题

一个嵌入式系统的设计者必须做出许多重要的决定。对实际应用或是将要设计的产品的特性提出特定的要求和限制，在本节中我们将考虑设计者要面对的一些最重要的问题。

1. 成本

在许多嵌入式应用中电子设备的成本必须是低廉的，如果单个芯片能实现所有必需的功能，则可实现成本最低的解决方案。只有当该芯片能提供足够的 I/O 能力来满足应用的需求以及有充足的片上存储器来存放必需的程序和数据时，做到这一点才是可能的。

2. I/O 能力

微控制器芯片提供多种 I/O 资源，从简单的并行和串行端口到计数器、定时器以及 A/D 和 D/A 转换电路。可用的 I/O 线数是重要的，没有足够的 I/O 线时需要使用外部电路去补充它。这已经在图 10-13 的反应定时器例子中作了说明，其中外部译码器电路用于驱动由微控制器提供的 4 位 BCD 信号的七段显示器。如果微控制器有四个而不是两个并行端口，那么就可以将每个七段显示器与一个端口相连。然后控制程序就可以直接地生成每个显示器中驱动七个单独的段所需要的七位信号。

3. 规格

控制器芯片有各种不同的规格，如果一个应用用 8 位的微控制器就足够处理了，就没有必要使用可能价格比较昂贵、尺寸比较大并且消耗更多功率的 16 位或 32 位的芯片。实际中大多数的应用可以使用相对较小的芯片进行处理。

4. 功耗

功耗在所有的计算机应用中是一个重要的考虑因素。在高性能的系统中功耗很高，需要增加一些机制来驱散产生的热量。在许多嵌入式应用中所消耗的功率很低，所以散热就不是一个问题。但是，这些应用通常是电池供电的产品，所以电池的寿命（取决于功耗）是主要的因素。

5. 片上存储器

在微控制器芯片中包含存储器会使得简单的嵌入式应用可以使用单个芯片来实现。存储器的大小和类型有很大的差异。相对小容量的RAM可能足够存储计算中的数据。存储程序需要一个较大的只读存储器，这个存储器可以是ROM、PROM、EPROM、EEPROM或闪存。对于大容量的产品，最为经济的选择是采用带ROM的微控制器。但是，这也是一种最不灵活的选择，因为ROM中的内容在芯片制造时就被永久设置了。最灵活的应用是由EEPROM和闪存提供的存储方式，它们可以被多次编程。

对于有更大存储要求的应用，需要使用外部存储器。有些微控制器不包含任何的片上存储器，它们通常用于所需要的存储总量非常大，不能在微控制器芯片内部实现的更复杂的应用中。

6. 性能

当微控制器在家用设备和玩具这样的应用中使用时，性能通常不是一个重要的考虑因素。在这种情况下可以选择小型并且便宜的芯片。但是，在数码相机、移动电话和一些手持视频游戏机这样的应用中就需要有更高的性能。高性能就需要比较强大的芯片，并会导致更高的费用和更大的功耗。由于这些应用通常是用电池供电的，所以降低功耗也是很重要的。需要进行多
415 方面的权衡以满足这些相互冲突的目标。

7. 软件

使用高级计算机语言编写应用程序有许多优势，它们使程序开发过程变得简单并且使将来的软件维护和修改工作更加容易。但是，在有些情况下求助于汇编语言可能会更加理想或有必要。一个精心设计的汇编语言程序生成的目标代码可能会比由编译程序生成的代码压缩10%～20%（就需要的存储总量来讲）。如果一个嵌入式应用是基于具有有限的片上存储器的微控制器而建立的，那么如果所需的代码能放进芯片上提供的存储器中，就避免了外部存储器的使用，这将是一个主要的优势。

对于系统设计者来说，片上RAM的有限容量是一个主要的考虑因素。这种存储器通常用于存储动态数据，就像一个临时缓冲区，或是用于实现堆栈。当用高级语言（如C）编写一个应用程序时，必须小心，以确保代码和数据的总量不超过可用存储器的容量。

8. 指令集

另一个重要的问题是处理器所使用指令集的特性。CISC风格的指令比RISC风格的指令产生更紧凑的代码。因此，处理器的选择对代码的大小也有影响。ARM体系结构的Thumb版本提供了解决这个问题的方法的一个有趣例子，其中为32位处理器设计的RISC风格的指令集已经被修改成了使用16位指令的更高度的编码形式，这在10.6.3节中讨论过。为Thumb版本编写的程序比为完全ARM体系结构编写的程序紧凑了30%。

9. 开发工具

数字系统的设计者对开发工具的依赖性很强。这些开发工具包括计算机辅助设计（CAD）软件包、编译器、汇编程序及处理器的模拟器。开发工具的范围和可用性通常依赖于所选择的嵌入式处理器。有第三方支持也是非常具有吸引力的，因为在第三方那里有可替代的工具和文档。良好的文档和来自于制造商的有益的建议（如果需要的话）都是非常宝贵的。

10. 可测试性和可靠性

印刷电路板通常是很难测试的，尤其是当板上组装着密集的芯片时。如果将整个系统设计成容易测试的方式，那么测试过程就会大大简化。一个微控制器芯片可以包含一些电路，这些电路可以使包含该芯片的印刷电路板比较容易测试。例如，在有些微控制器中包含有一个测试访问端口（test access port），它与用于可测试结构的IEEE 1149.1标准兼容，该标准被称为

测试访问端口和边界扫描结构标准 [1]。

嵌入式应用要求具有稳健性和可靠性。一个典型产品的生命周期希望最少可达到 5 年以上。这一点与个人计算机不同，个人计算机容易在短期内被淘汰。

## 10.8　结束语

这一章介绍了嵌入式计算机系统的设计。我们没有使用某个具体的商用微控制器进行讨论，因为所介绍的原则是通用的并且处理的也是嵌入式系统设计者要面对的核心问题。

理解硬件和软件间密切的相互作用是非常重要的。设计选择可能包括对轮询 I/O 和中断的权衡、不同的指令集间根据功能和代码紧凑性的权衡、功耗和性能间的权衡，等等。

在这一章中我们讨论了使用微控制器实现嵌入式系统，在下一章中，我们将介绍如何使用 FPGA 芯片来实现这样的系统。特别地，我们将集中讨论片上系统方法，试图在单个芯片上实现整个系统。

最后，我们讨论了嵌入式系统不同于通用计算机的特点。基本原则是相同的。嵌入式系统的设计者必须很熟悉计算机的组织结构，对包括指令系统、程序的执行、输入 / 输出技术、存储器结构以及与系统中可能包含的各种设备进行交互的接口方案都要有一个透彻的理解。那么，有什么不同呢？

通用计算机执行任意的应用程序，包括那些用来创建或修改其他程序的应用程序。由于处理算法或者用户的控制，每个程序通常运行一段有限的时间。在处理器芯片外部需要有一个大的主存，以满足一些应用程序的潜在的大容量需求。也需要大容量的辅助存储设备来存放包含程序和数据的文件。一个复杂的操作系统用于控制计算机系统中的所有资源。

相比之下，嵌入式系统通常执行一个不太可能被修改的单一的应用程序。当系统启动时，这个程序会自动开始执行，并且程序的执行是连续不断的，直到系统关闭。存储器的需求往往是很小的，通常有可能使用一个具有足够的片上存储器的微控制器。不需要功能强大的操作系统软件。在很多情况下，是没有操作系统软件的。这个单一的应用程序连续不断地执行，并直接访问所有的处理器、存储器和输入 / 输出资源。

我们的讨论集中在低端的嵌入式系统，其计算需求相对较小。这种系统体现了嵌入式应用的主要原则。但是，也有很多高端的嵌入式系统，比如那些用于飞机和高速列车中的系统，它们需要相当强大的计算能力，往往使用多个处理器来实现。第 12 章我们会讲解与多处理器系统有关的问题。

## 习题

[M] 10.1 在例 10.2 中，我们假设 I/O 设备能以位串方式发送 8 位数据。现在考虑一个使用八根线的类似设备，它并行发送数据。这样，微控制器中的一个并行端口必须被用来接收数据，这只留下一个并行端口来显示表示数据的两个十六进制数字。因此，只能使用一个图 3-17 所示的七段显示设备。为了传达接收到的信息，可以让这一个设备依次地显示数字。先使最高有效位数字显示一秒钟，随后是一秒钟的空白期，然后第二位数字显示一秒钟。显示第二位数字后，显示器将会显示两秒钟的“—”（破折号）。这个序列将被不断地重复显示，以反映收到的最新数据。使用图 10-9 中的定时器，并假设它被一个 100MHz 的时钟驱动。给出必要的硬件连接，并写一个程序来实现所需的任务。使用轮询方式来检查定时器的状态。

[M] 10.2 使用定时器的中断能力来解决习题 10.1 中的问题。

[M] 10.3 使用 10.3 节描述的微控制器来控制包含一个能在两种速度（快速和慢速）下运行的电机的系统。跟电机相关联的传感器使用一根信号线来指示电机的当前速度。对于快的速度，传感器传

输一个频率为 100kHz 的、连续的方波信号，对于慢的速度，它传输 50kHz 的信号。如果电机不运行，它就不传送信号（即恒定的逻辑值 0）。微控制器使用图 3-17 所示的七段显示器来将电机的状态显示为 F、S 或者 0。给出必要的硬件连接，并写一个程序来实现所需完成的任务。使用图 10-9 中的定时器来生成测试传感器所发出信号的频率所需的时间间隔。假设定时器由 100MHz 的时钟驱动，使用轮询方式来检查定时器的状态。

[M] 10.4 使用定时器的中断能力来解决习题 10.3 中的问题。

[D] 10.5 使用 10.3 节中的微控制器在它的串行端口上接收十进制数。每个数由两个 ASCII 码字符的数字编码组成，为了区分连续的两位数字的数，使用 H 作为定界符。这样，如果两个连续的数是 43 和 28，那么接收到的序列将是 H43H28。每个数将被显示在连接到并行端口 A 和 B 上的两个七段显示器上。定界符不应该被显示出来。只有当下一个数的两位数字都接收到时显示的数才会被改变。给出完成这种功能所需要的连接，如图 3-17 所示的那样标记显示器单元的段。写一个程序执行所需完成的任务。使用图 10-8 中的串行接口，并使用轮询方式检测每个 ASCII 码字符是否到达。

[D] 10.6 通过使用中断方式检测每个 ASCII 码字符是否到达来解决习题 10.5 中的问题。

[D] 10.7 使用 10.3 节中的微控制器在它的串行端口上接收十进制数，每个数由四个 ASCII 码字符的数字编码组成。为了区分连续的四位数字的数，使用 H 作为定界符。这样，如果两个连续的数是 2143 和 6292，那么接收到的序列将是 H2143H6292。每个数将被显示到四个七段显示器单元上。假设每个显示器单元有一个 BCD-七段译码器电路与其相连，如图 P10-1 所示。给出到微控制器的必要连接，写一个程序执行所需完成的任务。使用图 10-8 中的串行接口，并使用轮询方式检测每个 ASCII 码字符是否到达。

418

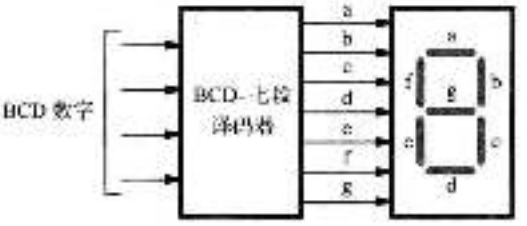
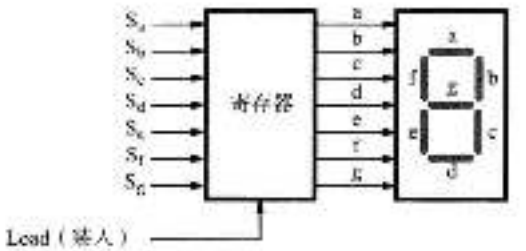



图 P10-1 使用 BCD 译码器的七段显示器

[D] 10.8 通过使用中断方式检测每个 ASCII 码字符是否到达来解决习题 10.7 中的问题。

[D] 10.9 重复习题 10.7 中的问题，但假定每个七段显示器单元有一个 7 位的寄存器与其相连，而不是 BCD-七段译码器。该寄存器有一个控制输入端 Load，当 Load=1 时，7 个数据位被装入寄存器中。寄存器中的每一位可以驱动所连接的显示器单元的一个段。图 P10-2 给出了寄存器显示器方案。排列数控制器的输出连接以使并行端口 A 为所有的四个显示器单元提供数据。




419

图 P10-2 使用寄存器的七段显示器

[D] 10.10 通过使用中断方式检测每个 ASCII 码字符是否到达来解决习题 10.9 中的问题。

[E] 10.11 假定被测试人总是在一秒之内做出响应，修改 10.4 节中的反应定时器。这样，经历的反应时

间可以只用两个表示百分之一秒的数字显示。将两个七段显示器单元连接到端口 A 上并修改图 10-14 中的程序来实现所需要的操作。

[M] 10.12 在图 10-13 中，每个数字的七段显示器单元都包含着一个 BCD-七段译码器；因此微控制器为每个将要被显示的数字提供一个 4 位 BCD 码。假设不使用译码器，每个七段单元有一个带有控制输入端 Load 的 7 位寄存器，当 Load = 1 时，7 个数据位被装入寄存器中。寄存器中的每一位驱动所连接的显示器单元的一个段。图 P10-2 给出了寄存器显示器方案。修改图 10-14 中的程序使其能够适用于这个寄存器显示器电路。

[M] 10.13 使用 10.3 节中的微控制器生成一个“日计时”时钟。这个时间（小时和分钟）将被显示到四个七段显示器单元上。假定每个显示器单元有一个 BCD-七段译码器与其相连，如图 P10-1 所示。同时假设使用一个 100MHz 的时钟。给出所需要的硬件连接并写出相应的程序。

[M] 10.14 假定每个七段显示器单元有一个寄存器与其相连，如图 P10-2 所示，重复习题 10.13 中的问题。

[E] 10.15 在一个用单芯片实现的系统中，处理器和主存储器驻留在同一个芯片上。在这个系统中是否需要高速缓存？请解释。

## 参考文献

1. *Test Access Port and Boundary-Scan Architecture*, IEEE Standard 1149.1, May 1990.

第 11 章

Computer Organization and Embedded Systems, Sixth Edition

# 片上系统——案例研究

## 本章目标

在本章中你将学习以下内容：

- 设计一个可在 FPGA 芯片上实现的系统
- 使用 CAD（计算机辅助设计）工具
- 使用参数化模块
- 典型的设计要求

421

在第 10 章，我们讨论了嵌入式系统中微控制器的使用。在一个嵌入式应用中，我们希望使用的芯片数量尽可能的少。理想的情况是用单个芯片来实现整个系统。术语片上系统（System-on-a-Chip，SOC）用来描述该技术。在比较简单的应用中，用那些可获得的商用微控制器可以实现所有需要的功能。但是在比较复杂的应用中就不太可能了。

为一个复杂的嵌入式应用设计一个微控制器并将其以定制芯片的形式实现不仅富有挑战性，而且费用高昂。同时，这也会耗费大量的时间。但是，大部分消费品的开发时间又必须要尽量短。如果设计者可以利用一些预先设计的易于使用的电路模块，那么对于一个给定的应用，实现其整个系统的芯片就可以在较短的时间内开发出来。处理器电路就是所需的模块之一。这种电路在技术文献中被称为处理器核（processor core）。通过使用权限许可协议，我们可以从多家企业获得各种处理器核。同时也可以获得其他模块来实现存储器、输入/输出接口、模/数（A/D）和数/模（D/A）转换电路或 DSP（数字信号处理）电路。然后系统开发者使用这些可用模块并设计出特定应用所需电路的剩余部分，从而完成设计。

处理器核和其他模块的提供商是在销售他们的设计而不是芯片。他们将知识产权（intellectual property，IP）提供给其他人使用，以设计他们自己的芯片。为了方便基于 IP 产品的开发，可以使用各种计算机辅助设计（CAD）工具。

成本是实现一个定制芯片的主要因素。制造这种芯片的费用很高，尽管它们能够提供更好的性能和更低的功耗，但仅当需要大量的专用芯片时，其成本才是合理的。一种可能的替代方式是采用现场可编程门阵列（FPGA）技术。

## 11.1 FPGA 的实现

FPGA 为在单个芯片上实现系统提供了一个有吸引力的平台。它不像市面上销售的微处理器芯片那样为设计者提供一组预先定义好的功能单元，FPGA 器件允许设计者完全自由地进行设计。设计者可以使用合适的 IP 模块，然后按照需求构造系统的其余部分。这可以相对容易地完成。一旦完成设计并经过测试，就可以立即将其在 FPGA 器件中实现。

FPGA 的功能已经有了突飞猛进的增强。单个 FPGA 芯片可以实现一个包含数十万个逻辑门的系统。这样大的芯片已经足以实现复杂的嵌入式应用所需要的微控制器和其他电路的典型功能。

在这一章中，我们将研究在嵌入式环境中使用 FPGA 所涉及的问题。为了使讨论尽可能地实用，我们将考虑 Altera 公司提供的技术，该公司是 FPGA 器件的主要供应商之一，并支持

422

CAD 软件。

### 11.1.1　FPGA 器件

附录 A 中对 FPGA 的基本结构进行了说明。FPGA 器件包含了大量的逻辑元件以及用于将它们互连的通用布线资源。通常它们也包含相当数量的存储器，当存储器容量需求不太大时可以用这些存储器来实现嵌入式系统中的 RAM 和 ROM 部件。许多 FPGA 也包含乘法器电路，该电路在 DSP 应用中尤其有用。

我们必须对 FPGA 器件进行编程以实现特定的设计。逻辑元件包含了可编程的开关，我们必须对这些开关进行设置以实现所需的逻辑功能。通常情况下，可以对一个逻辑元件进行编程以实现包含四到六个变量的逻辑函数。逻辑元件同时也包含一个触发器，该触发器使得寄存器和有限状态机的实现成为可能。互连布线也包含可编程的开关，这些开关可以用来将逻辑元件互连起来实现所需的电路。设置这些开关的编程过程被称作配置（configure）FPGA 器件。

在大多数的 FPGA 器件中，每个可编程开关的状态都是在 8.2.2 节中所讨论的 SRAM 单元中进行设置的。由于 SRAM 单元只有在打开 FPGA 电源的情况下才能保持其状态，所以这种 FPGA 是易失性的。如果关闭电源，我们必须在再次接通电源时重新配置该器件。为了配置 FPGA，必须将配置信息装入器件中。这通常是使用另一个叫做配置器件（configuration device）的芯片来完成的，该器件将所需的信息保存在闪存类型的存储器中。只要接通电源，配置器件就自动对 FPGA 进行编程。

通常情况下，配置器件中的闪存容量很大，不仅可以容纳在 FPGA 中所实现电路的配置数据，而且还可以容纳一些由数据或代码组成的额外信息。如果处理器核是在 FPGA 中实现的，那么就可以在配置器件中存储一些处理器将要执行的代码。

### 11.1.2　处理器的选择

任何一个片上系统的关键部件是处理器核。基于 FPGA 的系统存在两种不同的选择。其中一种包含一个处理器，该处理器是由软件定义的并在 FPGA 中以与其他电路相同的方式来实现的。另一种则包含一个专用的 FPGA 芯片，其处理器核是在制造时实现在芯片上的。

#### 1. 软处理器核

最灵活的解决方案是提供一个用硬件描述语言（如 Verilog 或者 VHDL）编写的软件模块，该模块指定了一个参数化的处理器。嵌入式系统的设计者可以设置参数使得处理器拥有目标应用所需的合适功能。例如，高速缓存配置有一个参数，其可能的选择如下：　423

- 没有高速缓存。
- 有指令高速缓存，但没有数据高速缓存。
- 既有指令高速缓存又有数据高速缓存。

另一个参数可能与处理器中是否包含乘法和除法电路有关。乘法和除法操作可以用硬件实现，但也可以用软件方式实现。以硬件方式实现乘法和除法操作会使用更多的 FPGA 资源，但这也大大地提高了性能。

#### 2. 硬处理器核

软处理器核的一种替代方法是直接在芯片上以硬件模块的方式实现处理器，这样就可以构造一个专用的 FPGA，从而可以实现更高性能的系统。但是这种 FPGA 的成本要高于普通 FPGA 器件的成本。

## 11.2　计算机辅助设计工具

FPGA 器件的制造商提供了强大的 CAD 工具，这使得嵌入式系统的设计变得相对简单了。

各种预先定义的模块都以参数化的形式提供。设计者可以通过包含这些模块并指定其参数来构建一个满足应用要求的系统。这样的模块有：

- 处理器核
- 存储器模块及其接口
- 并行 I/O 接口
- 串行 I/O 接口
- 定时器 / 计数器电路

这些模块可能足以实现一个特定的嵌入式系统所需要的所有功能。如果还不够，那么我们必须设计额外的专用电路并将其包含在系统中。

通常情况下，我们会首先确定一个包含处理器核和其他参数化模块的子系统。使用 CAD 工具可以生成一个实现该子系统的模块。该模块由硬件描述语言定义。然后在总体设计时将该模块与任何已创建的其他专用电路一起实例化。最后，我们使用另一个不同的 CAD 工具来合成并实现总体设计，该设计形式可用于配置 FPGA 器件。

除了 FPGA 器件，还需要将完成系统所需要的外部组件也包括进来，如交换机、显示器和额外的存储器芯片。这些部件必须连接到 FPGA 的适当引脚上。我们将在 11.3 节的设计实例中讨论这些问题。

424

为了向读者提供一个 CAD 工具和 FPGA 器件模块的具体例子，我们将简要地介绍 Altera 公司提供的工具。有关其技术和工具的所有信息都可以在 Altera 公司的网站上得到 [1]。

## ALTERA 公司的 CAD 工具

Altera 公司主要的 CAD 工具被称为 Quartus II 软件，其中包括了在 FPGA 器件上设计并实现一个数字系统所需的全部工具。其中有一个工具叫做 SOPC Builder，可以用来设计包含一个处理器核的系统。该工具包括了可用于所设计系统的多个参数化模块。为了说明它们的特性，我们将考虑其中的四个模块。

### 1. Nios II 处理器

附录 B 中描述了 Nios II 处理器。对于在 FPGA 中的实现，它提供了三个版本：经济版、标准版和快速版。经济版是最简单的且实现起来花费最少（从使用 FPGA 资源的角度上看），其性能也是最低的。它不包含任何高速缓存，是非流水线的，而且不使用转移预测。标准版有着较好的性能，它包括一个指令高速缓存，是流水线的，并且使用静态转移预测。快速版的性能是最好的，它同时包括指令高速缓存和数据高速缓存，并且使用动态转移预测。

设计者可以指定多个参数，包括指令以及数据高速缓存的大小。Nios II 处理器核很小，它只占 FPGA 器件的很小一部分。在一个相对小的 FPGA 中，有可能可以实现多达 10 个 Nios II 核。

### 2. 存储器

FPGA 器件的存储块可用于实现高速缓存和一部分主存。用这种方式实现的那部分主存被称为片上存储器（on-chip memory）。该片上存储器可以用多种方式对其进行配置。它可以被实现为 RAM 或 ROM 类型的存储器。在设计阶段，我们可以指定其大小和字长。

如果片上存储器不够大，无法容纳嵌入式应用中所需的软件，那么就有必要通过使用外部存储器芯片来提供额外的存储空间。使用 SOPC Builder 可以很容易地生成将 FPGA 上的系统连接到各种外部存储器组件（如 SRAM、SDRAM 和闪存设备）所需的控制器和接口。

### 3. 并行 I/O 接口

并行接口，被称为 PIO，是一个可同时用于输入和输出的参数化模块。在设计时，我们可以选择其数据端口将其用作：

- 输入端口
- 输出端口
- 双向端口

425

如果选择了双向端口选项，那么 PIO 数据线必须与具有三态功能的 FPGA 引脚相连。




图 11-1　PIO 接口中的寄存器

处理器将 PIO 作为一个存储器映射接口来访问，可以采用第 3 章中描述的方式与其进行通信。图 11-1 显示了 PIO 中的寄存器。寄存器的大小 $n$ 是一个在设计阶段指定的参数，其范围在 1 到 32 之间。这些寄存器的使用如下：

- 数据（data）寄存器保存了在处理器和 PIO 接口之间传输的 $n$ 位数据。
- 在实现双向端口时，方向（direction）寄存器确定了 $n$ 条数据线中每一条的传输方向（输入或输出）。
- 中断屏蔽（interrupt-mask）寄存器用于激活来自与 PIO 相连的输入线上的中断。在 $n$ 条输入线的每一条上都可以发出单独的中断请求。
- 边沿捕获（edge-capture）寄存器通过检测与 PIO 相连的输入线上的信号，指示其中逻辑值的变化。所检测到的边沿类型（上升或下降）将在设计阶段指定。

我们可以单独配置将 PIO 连接至一个 I/O 设备的每条线路。如果 PIO 只是作为一个输入端口，那么在设计阶段我们会将所有的 $n$ 条线路都配置为输入。类似地，对于一个输出端口，所有的线路都被配置为输出。在这些情况下，方向寄存器是不需要的，同时它也不会在最终生成的电路中实现。对于一个双向端口，就需要方向寄存器了；当其第 $k$ 位的值等于 1（0）时，端口的第 $k$ 条线路就起着向（从）所连接的 I/O 设备输出（输入）信息的作用。

当把 PIO 作为一个输入端口使用时，其数据寄存器包含当前输入线上的逻辑值。通过使用边沿捕获寄存器我们可以检测输入线上逻辑值的变化。在设计时，我们可以指定：当输入信号出现一个边沿的时候将该寄存器中的位设置为 1。该边沿可以被指定为：上升、下降或二者均可。通过一条程序指令将 0 写入寄存器，就可以将边沿捕获寄存器中的位清零。

426

中断屏蔽寄存器允许激活和禁用中断。通过将 1 写入寄存器的第 $k$ 位，就可以激活由输入线 $k$ 上的活动引起的中断。该中断可以是：

- 电平敏感的，在这种情况下，当任意一条被激活的输入线路上的信号值是 1 的时候，就会发出中断请求。
- 边沿敏感的，在这种情况下，当边沿捕获寄存器中任意一个被激活的位的值等于 1 的

时候，就会发出中断请求。

注意，不论长度 $n$ 是多少，图 11-1 中 PIO 寄存器的地址都偏移 4 个字节。这样，寄存器地址都是字对齐的。

**4. 间隔定时器**

定时器模块的功能与 3.4 节中描述的定时器类似。其关键部件是一个计数器，计数器的内容在每个时钟周期中减 1。该计数器可以被指定为 32 或 64 位长。在我们的讨论中，我们将假设计数器是 32 位的。间隔定时器接口中的寄存器如图 11-2 所示。每个寄存器都是 16 位长的。在状态寄存器中，只使用了 2 位：

- 当计数器运行的时候，RUN 等于 1，否则就等于 0。该位不受处理器写入状态寄存器操作的影响。
- TO 是超时位。它在计数器到达 0 时被置为 1，并且它会一直保持值 1 直到处理器通过写入 0 将其清零。




427

图 11-2 间隔定时器接口中的寄存器

在控制寄存器中，用到了 4 个位：

- 通过将 STOP 置为 1 来停止计数器。
- 通过将 START 置为 1，使计数器开始运行。
- 当计数器到达 0 时，CONT 决定计数器此时的行为。如果 CONT=0，计数器会在它到达 0 时停止运行。如果 CONT=1，则计数器重新装入初始计数值并继续运行。
- ITO 被置为 1 时允许中断。

初始计数值必须在两个 16 位的写操作内装入寄存器中。计数器快照寄存器用于在计数器运行时对其内容进行快照。对任意一个快照寄存器的写操作都会导致一个快照操作，也就是说计数器的当前内容会被装入快照寄存器中。然后，我们就可以用通常的方式去读取这些寄存器。

除了可以使用初始计数值来定义一个超时周期外，我们也可以在设计时指定一个默认的超时周期。当初始计数值为零的时候，我们就使用该默认的超时周期。

当 TO=1 时会发出一个中断请求并将 ITO 置为 1。为了清除该请求，处理器必须向 TO 中写入 0。

在下一节中，我们将在一个完整的设计示例中使用上面所介绍的模块。

## 11.3　闹钟示例

在这一节中，我们将介绍一个嵌入式系统的详细示例。我们将说明如何使用 FPGA 技术来实现一个闹钟。该闹钟，如图 11-3 所示，具有以下的要求：

- 通过四个七段显示器用小时和分钟来显示时间；
- 使用一个 on/off 滑动开关来启用闹铃功能；
- 使用两个滑动开关来设置实际时间和闹铃时间；
- 使用两个按键开关来设置小时和分钟；
- 一个 LED 灯用来显示闹铃已被设置；
- 用两个垂直排列的 LED 灯，组成一个冒号来分隔小时和分钟；
- 当闹铃滑动开关被激活并且到达设置的闹铃时间时，发出嗡嗡的声音。

### 11.3.1　系统的用户视图

图 11-3 显示了用户所看到的闹钟。图 3-17 所描述的七段显示器用来显示时间。时间的显示范围为 00:00 到 23:59。该时钟的操作如下：




图 11-3　闹钟的用户视图

- 当接通电源时，当天时间和闹铃时间都被清为 0。
- 当天时间的设置方法：激活“设置实际时间（set-actual-time）”的滑动开关，然后通过按下小时（Hour）和分钟（Minute）按键来设置时间。每按一次按键，显示的时间就增加 1。
- 在激活“设置闹铃时间（set-the-alarm-time）”开关后，用同样的方式设置闹铃时间。
- 通过激活“闹铃－开 / 关（alarm-on/off）”开关来开启闹铃。这会使得对应的 LED 灯被点亮。
- 在启动闹铃开关的情况下，扬声器会在到达闹铃时间时发出嗡嗡声。

### 11.3.2　系统的定义和生成

我们的目标是利用 FPGA 芯片以及包括滑动开关、按键开关、七段显示器、LED 灯和扬声器在内的外部组件来实现闹钟。图 11-4 描述了所需的系统，其中的处理器是 Nios II 处理器。即便是在一个很小的 FPGA 芯片上，其片上存储器的容量也足以满足我们应用的需要。我们通过 PIO 接口来连接外部组件。系统包含两个定时器，其中一个旨在提供一分钟的时间间隔，用于更新一天的时间。另一个则用于生成方波来产生嗡嗡声。通过使用 11.2.1 节中描述的间隔定时器模块，就可以实现定时器。我们将假设滑动开关被激活时会产生一个逻辑信号 1。按键开关是防反弹的，当它们被按下时会产生一个逻辑信号 0。

该系统是通过使用 Quartus II 软件在 FPGA 器件中实现的。SOPC Builder 用来实现图 11-4 中阴影区域的块。其中 PIO 块的配置如下：

- PIO1 是一个 3 位宽的输入端口。它的数据输入是电平敏感的。
- PIO2 是一个 2 位宽的输入端口。它的输入是下降沿敏感的，这样当相应的按键被按下时，边沿捕获寄存器的一位就会被置为 1。
- PIO3 是一个 32 位宽的输出端口。它的每一个字节都与一个七段显示器相连，每个字节的 0 到 6 位中的每一位用来驱动七段显示器的一段，而不使用第 7 位。低字节将与

显示器上显示分钟的低位数字先连，而高字节则与显示小时的高位数字相连。

- PIO4 是一个 3 位宽的输出端口。
- PIO5 是一个 1 位宽的输出端口。

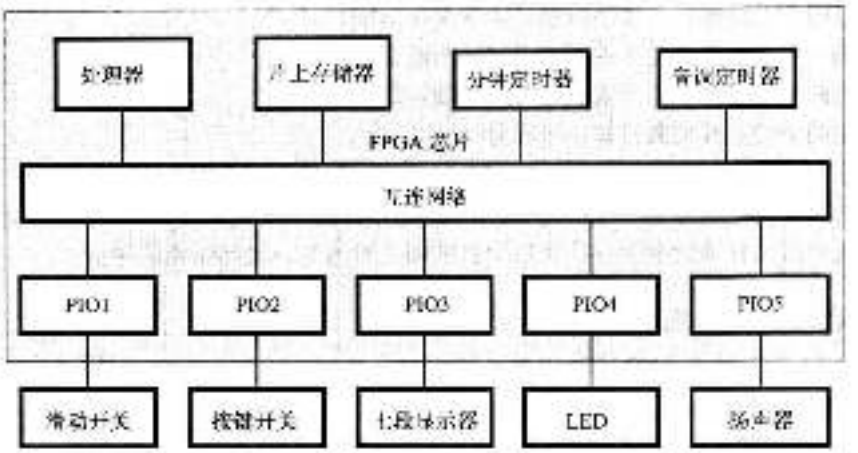



图 11-4 闹钟的框图

通过使用 SOPC Builder，该系统被指定为如图 11-5 所描述的那样。注意，PIO 的名称表示了它们的功能。SOPC Builder 为系统中的各种组件分配地址。片上存储器占用的地址范围是 0 到 0x3FFF，而定时器和 PIO 接口的起始地址为 0x5000。如果需要的话设计者也可以指定不同的地址。还要注意的是定时器和按键开关可以发出中断请求（IRQ）。图中的最后一列给出了 Nios II 控制寄存器 ctl3 和 ctl4 中与这些中断源的中断相关的位。SOPC Builder 通过产生一个用 Verilog 或 VHDL 硬件描述语言描述系统的模块生成指定的系统。




图 11-5 使用 SOPC Builder 设计的 FPGA 片上系统

### 11.3.3　电路实现

Quartus II 软件包括一个编译器，该编译器可以接受用硬件描述语言描述的数字系统规范。它合成实现系统的电路，并确定如何在 FPGA 芯片上实现该电路。

由于外部设备必须连接到 FPGA 的引脚上，设计者必须指定所需的连接。这被称为引脚分配（pin assignment）。编译器会以一个配置文件的形式生成最终的实现结果，这个配置文件用于将配置信息下载到 FPGA 器件中。

图 11-4 并没有给出实现完整闹钟所需要的所有部件。缺少的部件有：电源，时钟信号发生器以及对 FPGA 芯片编程的配置设备。我们将假设使用一个 100MHz 的外部时钟信号。FPGA 的引脚通常是不能直接连接到如开关、七段显示器和 LED 等外部设备上的。每个设备都有它自己的电气特性，这通常意味着我们必须使用电阻等元件，如图 10-13 所示。

在我们的设计中，设置小时和分钟的按键分别与相应 PIO 的 $b_1$ 和 $b_0$ 位相连。滑动开关“设置实际时间”（set-actual-time）、“设置闹铃时间”（set-alarm-time）和“闹铃一开/关”（alarm-on/off）分别与相应 PIO 的 $b_2$、$b_1$ 和 $b_0$ 位相连。分钟定时器的超时周期为 60 秒，而音调定时器的超时周期则为 1 毫秒。

### 11.3.4　应用软件

为了实现闹钟的功能，我们有必要写一段程序使其在设计的硬件上运行。我们将给出两个程序：一个用 C 语言编写，另一个用 Nios II 汇编语言编写。通过这两个程序来说明如何编写这样的程序。编写好的程序必须被编译成 Nios II 机器代码。通电时，要将该代码从配置设备装入片上存储器中。

我们使用下面的方法来编写程序。实际时间和闹铃时间保存为 32 位的二进制数，以分钟的形式来表示时间。每当分钟定时器达到 0 的时候，实际时间就会增加 1，并通过将状态寄存器中的 TO 位置为 1 对其进行显示。当实际时间增加的时候，需要检查它是否已达到了 1440，该值是一天的分钟数。如果已达到 1440，就必须将时间清为 0。这就相当于时间从 23:59 变成了 00:00。在按下按键设置时间的时候也需要进行类似的检测。

为了显示时间，在计算表示小时和分钟的四个十进制数字的时候，将时间除以 600，这样可以得到高位的小时数字，然后再将余数除以 60 得到低位的小时数字，等等。我们还会使用一张表来查找被发送到七段显示器的相应的段模式。

我们使用音调定时器来生成一个 500Hz 的方波信号，当将其连接到一个扬声器时，便会产生嗡嗡声。音调定时器在连续模式下运行。由于它的预定义超时周期为 1ms，所以我们可以通过将每个定时器周期结束时的信号逻辑值反转过来直接产生 500Hz 的信号。然而，为了说明如何在一个应用程序中定义不同的周期，我们将使用初始计数寄存器来指定所需的周期。在这种情况下，如果计数器是由一个 100MHz 的时钟驱动的，那么产生 500Hz 的信号就需要值 0x30D40。

在对定时器的超时进行轮询时，需要检查其状态寄存器的 TO 位。由于计数器在运行的时候 RUN 位始终等于 1，所以可以通过检查状态寄存器的内容是否等于 3 来执行轮询任务。由于 RUN 位的状态并不会受到写操作的影响，所以我们只需通过对状态寄存器写 0 就可以将 TO 位清为 0。

1. C 程序

图 11-6 给出了一个可能的 C 语言程序。由于从性能的角度来看，我们的应用并没有什么

430
↓
431

要求，因此我们使用轮询法来访问每一个 PIO 和定时器。程序中的注释用于解释各个语句的含义。我们观察到，宏 ADJUST 定义了一个表达式，这使得时间在不同情况下都可以被正确地增加。

```
#define   minute_timer (volatile int *) 0x5000
#define   tone_timer (volatile int *) 0x5020
#define   sliders (volatile int *) 0x5040
#define   pushbuttons (volatile int *) 0x5050
#define   display (int *) 0x5060
#define   LEDs (int *) 0x5070
#define   speaker (int *) 0x5080
#define   ADJUST(t, x) ((t + x) >= 1440) ? ((t + x - 1440) : (t + x)
int       actual_time, alarm_time, alarm_active, time;

  /* 一六进制到 7 段的转换表 */
  unsigned char table[16] = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12, 0x02, 0x78,
      0x00, 0x18, 0x3F, 0x3F, 0x3F, 0x3F, 0x3F, 0x3F};

  void initializeToneTimer()
  {
      *(tone_timer + 2) = 0x0D40;    /* 为连续写操作 */
      *(tone_timer + 3) = 0x03;      /* 设置超时周期 */
      *(tone_timer + 1) = 0x6;       /* 在连续模式下开始 */
  }
  void DISP(time)                    /* 获取显示器的 7 段模式 */
  {
      *display = table[time / 600] << 24
            table[(time % 600) / 60] << 16
            table[(time % 60) / 10] << 8
            table[(time % 10)];
  }

  main()
  {
      actual_time = alarm_time = alarm_active = 0;
      initializeToneTimer();
      *(minute_timer + 1) = 0x6;     /* 在连续模式下运行 */
      while (1)
      {
        if (*minute_timer == 3)      /* 过去 1 分钟 */
        {
            *minute_timer = 0;       /* 将 TO 位清 0*/
            actual_time = ADJUST(actual_time, 1);
        }
        if ((*sliders & 1) != 0)     /* 检测 "闹钟 开"（alarm on）开关 */
        {
            *LEDs = 7;               /* 打开闹铃 LED 灯 */
            if (actual_time == alarm_time)
                alarm_active = 1;    /* 启动闹铃警报 */
            else
                alarm_active = alarm_active & (*sliders & 1);
            if (*tone_timer == 3)    /* 生成方波 */
            {
                *speaker = (*speaker ^ 1) & alarm_active;
                *tone_timer = 0;     /* 将 TO 位清 0*/
```

图 11-6 闹钟的 C 程序

```
            }
        }
        else
        {
            *LEDs = 0;                  /* 关闭闹铃 LED 灯 */
            alarm_active = 0;
        }
        if ((*sliders & 4) != 0)        /* 检测 "设置当天时间（set-the-time-of-day）" 开关 */
        {
            DISP(actual_time);          /* 显示当天的时间 */
            if ((*(pushbuttons + 3) & 1) != 0)   /* 设置分钟? */
                actual_time = ADJUST(actual_time, 1);
            else if ((*(pushbuttons + 3) & 2) != 0)   /* 设置小时? */
                actual_time = ADJUST(actual_time, 60);
            *(pushbuttons + 3) = 0;     /* 将边沿捕获寄存器清 0*/
        }
        else if ((*sliders & 2) != 0)   /*检测 "设置闹铃时间"（set-the-alarm-time）开关 */
        {
            DISP(alarm_time);           /* 显示闹铃时间 */
            if ((*(pushbuttons + 3) & 1) != 0)   /* 设置分钟? */
                alarm_time = ADJUST(alarm_time, 1);
            else if ((*(pushbuttons + 3) & 2) != 0)   /* 设置小时? */
                alarm_time = ADJUST(alarm_time, 60);
            *(pushbuttons + 3) = 0;     /* 将边沿捕获寄存器清 0*/
        }
        else
            DISP(actual_time);          /* 显示当天的时间 */
    }
}
```

图 11-6　（续）

### 2. 汇编语言程序

图 11-7 给出了一个 Nios II 程序。为了说明中断的使用，该程序使用中断来处理分钟定时器，音调定时器则使用轮询法。

在设计阶段，当处理器接受一个中断请求时，SOPC Builder 分配 0x20 作为中断处理程序的起始地址。中断处理程序证实了由 Nios II 处理器的外部设备引起的中断已经发生，并根据情况调整返回地址（Nios II 中断机制的详细说明参见附录 B）。然后，它将分钟定时器的 TO 位清 0 并调用中断服务程序。注意，中断处理程序会保存寄存器 r2 和 ra 的内容，并在之后恢复。因为 ra 是在调用了程序时保存返回地址的链接寄存器，当程序中的某个子程序在执行时可能会发生定时器中断，所以中断处理程序会保存 ra 的内容。中断处理程序还会使用寄存器 ra 来调用中断服务程序 UPDATE_TIME，该程序负责增加实际时间并在午夜的时候将所存储的时间清为 0。

432
↓
433

主程序开始时先创建处理器堆栈并清除存储时间的存储单元。在初始化过程中，它也会设置音调定时器的超时周期。接着它会启动两个定时器并允许来自分钟定时器的中断。

主循环 LOOP 负责检查滑动开关的状态并采取必要的动作。子程序 DISP 则负责在七段显示器上显示实际时间或闹铃时间，这两个时间分别保存在存储单元 0x1000 和 0x1010 中。当相应的按键被按下时，子程序 SETSUB 用来对分钟和小时的值进行设置。每次按下按键时，相应的值被增加 1。

程序最后的那张表用来将十进制数字转换成相应的七位模式以便显示。

程序中的注释解释了各个语句的含义。控制寄存器 ctl3 和 ctl4 也可以分别用名称 ienable 和 ipending 来对其进行引用。我们会在程序中使用这些名称。

```
                .equ    minute_timer, 0x05000
                .equ    tone_timer, 0x5020
                .equ    sliders, 0x5040
                .equ    pushbuttons, 0x5050
                .equ    display, 0x5060
                .equ    LEDs, 0x5070
                .equ    speaker, 0x5080
                .equ    ACTUAL_TIME, 0x1000
                .equ    ALARM_TIME, 0x1010
                .equ    STACK, 0x2000
_start:         br      MAIN

/* 中断处理程序 */
                org     0x20
                subi    sp, sp, 8               /* 保存寄存器 */
                stw     r2, 0(sp)
                stw     r3, 4(sp)
                rdctl   et, ipending
                beq     et, r0, MAIN            /* 如果不是外部中断，则退出。 */
                                                /* 将其后作为复位 */
                subi    ea, ea, 4               /* 递减 ea，作返回主程序时执行 */
                                                /* 被中断的指令 */

                movia   r2, minute_timer        /* 将分钟定时器中的 TO 位清 0*/
                sthio   r0, (r2)
                call    UPDATE_TIME             /* 调用中断服务程序恢复寄存器 */
                ldw     r2, 0(sp)
                ldw     ra, 4(sp)
                addi    sp, sp, 8
                eret

/* 主程序 */
MAIN:           movia   sp, STACK               /* 设置堆栈指针 */
                movia   r2, ALARM_TIME          /* 将闹铃时间缓冲区清 0*/
                stw     r0, (r2)
                movia   r2, ACTUAL_TIME         /* 将实际时间缓冲区清 0*/
                stw     r0, (r2)
                movia   r2, sliders             /* 滑动开关的地址 */
                movia   r3, LEDs                /* LED 的地址 */
                movia   r4, display             /* 7 段显示器的地址 */
                movia   r5, pushbuttons         /* 按键的地址 */
                movi    r6, 6                   /* 打开两个垂直的 LED 灯 */
                stbio   r6, (r3)
                movia   r6, tone_timer
                ori     r7, r0, 0x0D40          /* 设置音调定时器周期 */
                sthio   r7, 8(r6)
                ori     r7, r0, 0x03
                sthio   r7, 12(r6)
                movi    r7, 6                   /* 启动音调定时器 */
                sthio   r7, 4(r6)
                movia   r6, minute_timer        /* 分钟定时器的地址 */
                addi    r7, r0, 7               /* 启动定时器 */
                sthio   r7, 4(r6)
                movi    r7, 1
                wrctl   ienable, r7             /* 允许定时器中断 */
                wrctl   status, r7              /* 允许外部中断 */
LOOP:           movia   r10, ACTUAL_TIME        /* 显示当天当时时间 */
                call    DISP
```

列 11-7 闹钟的 Nios II 程序

```
            ldbio   r7, (r2)
            andi    r11, r7, 1              /* 检查闹铃开关是否打开 */
            beq     r11, r0, NEXT
            movi    r11, 7                  /* 是，则打开闹铃 LED 灯 */
            stbio   r11, (r3)
            movia   r9, ALARM_TIME
            ldw     r11, (r9)               /* 将闹铃时间与实际时间进行比较 */
            ldw     r12, (r10)
            bne     r11, r12, NEXT          /* 时钟需要响铃吗? */
            movia   r8, tone_timer
            movi    r12, 1
RING_LOOP:
            call    DISP
            ldbio   r7, (r2)
            andi    r13, r7, 1              /* 检查闹铃开关是否打开 */
            beq     r13, r0, NEXT
            ldbio   r9, (r8)                /* 读取音调定时器的状态 */
            stbio   r0, (r8)                /* 将 TO 位清 0*/
            andi    r9, r9, 1               /* 检查计数器是否到达 0*/
            xor     r12, r9, r12            /* 生成下一个方波半周期 */
            movia   r11, speaker
            stbio   r12, (r11)              /* 将信号发送给扬声器 */
            br      RING_LOOP
NEXT:       movi    r11, 6                  /* 关闭"闹铃 开"LED 指示灯 */
            stbio   r11, (r3)
TEST_SLIDERS:
            ldbio   r7, (r2)
            andi    r11, r7, 2              /* "设置闹铃"开关是否是打开的 */
            beq     r11, r0, SETACT         /* 是，则设置闹铃时间 */
            movia   r10, ALARM_TIME         /* 设置闹铃时间 */
            br      SET_TIME
SETACT:
            andi    r11, r7, 4              /* "设置时间"开关是否是打开的 */
            beq     r11, r0, LOOP           /* 所有的滑动开关都是关闭的 */
            movia   r10, ACTUAL_TIME
SET_TIME:
            call    DISP
            call    SETSUB
            br      TEST_SLIDERS

/* 在 7 段显示器上显示时间 */
DISP:       subi    sp, sp, 24              /* 保存寄存器 */
            stw     r11, 0(sp)
            stw     r12, 4(sp)
            stw     r13, 8(sp)
            stw     r14, 12(sp)
            stw     r15, 16(sp)
            stw     r16, 20(sp)
            ldw     r11, (r10)              /* 加载将要显示的时间 */
            movi    r12, 600                /*除以 600，确定小时的第一位数字 */
            divu    r13, r11, r12
            ldb     r15, TABLE(r13)         /* 获取 7 段模式 */
            slli    r15, r15, 8             /* 为下一个数字腾出空间 */
            mul     r14, r13, r12           /* 计算除法操作的余数 */
            sub     r11, r11, r14
            movi    r12, 60                 /* 将余数除以 60，得到小时的第二位数字 */
            divu    r13, r11, r12
            ldb     r16, TABLE(r13)         /* 获取 7 段模式，将其与第一位数字 */
```

图 11-7　（续）

```
            or      r15, r15, r16       /* 连接起来，并移位 */
            slli    r15, r15, 8
            mul     r14, r13, r12       /* 确定要显示的分钟 */
            sub     r11, r11, r14
            movi    r12, 10             /* 除以 10，确定分钟的第一位数字 */
            divu    r13, r11, r12
            ldb     r16, TABLE(r13)     /* 查取 7 段模式，将其与前两位数字 */
            or      r15, r15, r16       /* 拼接起来，并移位 */
            slli    r15, r15, 8
            mul     r14, r13, r12       /* 计算余数，它是最后一位数字 */
            sub     r11, r11, r14
            ldb     r16, TABLE(r11)     /* 将最后一位数字与前 3 位数字拼接起来 */
            or      r15, r15, r16
            movia   r11, display
            stw     r15, (r11)          /* 显示所需到点模式 */
            ldw     r11, 0(sp)          /* 恢复寄存器 */
            ldw     r12, 4(sp)
            ldw     r13, 8(sp)
            ldw     r14, 12(sp)
            ldw     r15, 16(sp)
            ldw     r16, 20(sp)
            addi    sp, sp, 24
            ret

        /* 设置所需的时间 */
        SETSUB:
            subi    sp, sp, 16          /* 保存寄存器 */
            stw     r11, 0(sp)
            stw     r12, 4(sp)
            stw     r13, 8(sp)
            stw     r14, 12(sp)
            ldwio   r12, 12(r5)         /* 检查按键 */
            sthio   r0, 12(r5)          /* 将边沿捕获寄存器清 0*/
            andi    r13, r12, 1         /* 分钟按键是否被按下 */
            beq     r13, r0, HOURS      /* 如果没有，检查小时 */
            ldw     r11, (r10)          /* 加载当前时间 */
            movi    r12, 60             /* 除以 60，确定小时数 */
            divu    r13, r11, r12
            mul     r14, r13, r12       /* 整除得出的余数是分钟数 */
            sub     r11, r11, r14
            addi    r11, r11, 1         /* 增加分钟 */
            blt     r11, r12, SAVEM     /* 如果小于 60 就保存，否则将分钟设置为 00*/
            mov     r11, r0
SAVEM:      add     r11, r14, r11       /*( 小时 ×60 ) + 更新后的分钟 */
            stw     r11, (r10)          /* 保存新的时间 */
            br      DONE
HOURS:      andi    r13, r12, 2         /* 小时按键是否被按下？ */
            beq     r13, r0, DONE       /* 如果没有，则返回 */
            ldw     r11, (r10)          /* 加载当前时间（以分钟为单位）*/
            addi    r12, r11, 60        /* 加上 60 分钟 */
            movi    r13, 1440           /* 检查更新后的时间是否小于 24:00*/
            blt     r12, r13, SAVEH
            sub     r12, r12, r13       /* 将小时滚动到 00*/
SAVEH:      stw     r12, (r10)          /* 保存新的时间 */
DONE:
            ldw     r11, 0(sp)          /* 恢复寄存器 */
            ldw     r12, 4(sp)
            ldw     r13, 8(sp)
```

图 11-7 （续）

```
            ldw     r14, 12(sp)
            addi    sp, sp, 16
            ret

/* 更新实际时间的中断服务程序 */
UPDATE_TIME:
            subi    sp, sp, 12          /* 保存寄存器 */
            stw     r2, 0(sp)
            stw     r3, 4(sp)
            stw     r4, 8(sp)
            movia   r2, ACTUAL_TIME
            ldw     r3, (r2)            /* 加载一天的当前时间 */
            addi    r3, r3, 1           /* 增加 1 分钟 */
            movi    r4, 1440            /* 如果更新后的时间小于 24:00，则完成 */
            blt     r3, r4, SAVET
            mov     r3, r0              /* 否则，设置为 00:00*/
SAVET:      stw     r3, (r2)            /* 保存更新后的时间 */
            ldw     r2, 0(sp)           /* 恢复寄存器 */
            ldw     r3, 4(sp)
            ldw     r4, 8(sp)
            addi    sp, sp, 12
            ret

/* 十六进制数到 7 段的转换表 */
            .org    0x1050
TABLE:      .byte   0x40, 0x79, 0x24, 0x30, 0x19, 0x12, 0x02, 0x78
            .byte   0x00, 0x18, 0x3F, 0x3F, 0x3F, 0x3F, 0x3F, 0x3F
            end
```

图 11-7　（续）

## 11.4　结束语

嵌入式系统的设计者不可避免地会去寻求最简单和性价比最高的方法。当一个微控制器芯片拥有实现整个系统的所有资源时，它可能是最好的选择。但是如果需要额外的芯片来实现系统，那么情况就不同了。此时，FPGA 的解决方案是非常有吸引力的，因为它们倾向于使用更少的芯片来实现系统。

另一个考虑是可使用预先设计的模块。一个微控制器芯片包括了许多不同的模块。如果有任何的功能无法用这些模块实现，那么就必须使用额外的芯片来实现。FPGA 器件可使设计者设计任何类型的数字电路。使用现代 FPGA 器件可以实现非常庞大和复杂的电路。

实际的设计往往涉及执行常用任务的电路。这些电路通常可作为库模块使用，正如我们在上一节中使用的 I/O 接口和定时器电路所示的那样。对于信号处理应用来说，库中包含了典型的滤波电路。当一个系统要通过一个标准的互连方案（如 PCI Express）连接到另一台计算机上时，如果 PCI Express 接口可作为预先设计的模块使用，那么设计者的任务就会变得简单得多。

## 习题

[E] 11.1 在 11.3.4 节中，我们提到可以通过简单地将音调定时器每个超时周期结束时的信号逻辑值反转过来产生 500Hz 的方波信号。其中该定时器的超时周期被设计为 1ms。根据这一情况去修改图 11-6 中的程序。

[E] 11.2 针对图 11-7 中的程序，重复习题 11.1 中的问题。

434 ↓ 440

[M] 11.3 考虑 11.3 节中闹钟的显示格式。我们希望将时间显示为 12:00 到 11:59AM 或 PM 的形式，而不是从 00:00 到 23:59。假设现在要使第四个 LED，并将其连接到一个 4 位宽 PIO 的 $b_3$ 位上，而不是连接到 11.3 节中所使用的 3 位宽的 PIO 上。该 LED 亮的时候表示下午（PM）。修改图 11-6 中的程序以使用这种方式显示时间。

[M] 11.4 针对图 11-7 中的程序，重复习题 11.3 中的问题。

[E] 11.5 假设我们在 11.3 节的闹钟中只使用了一个默认超时周期为 1ms 的定时器。请对图 11-6 中的程序进行适当的修改来提供所需的功能。

[E] 11.6 针对图 11-7 中的程序，重复习题 11.5 中的问题。

[E] 11.7 在图 11-7 中，我们使用中断来处理分钟定时器，而对音调定时器采用轮询法。修改该程序，使这两个定时器都使用中断法。

[M] 11.8 在 11.3 节的闹钟中，其时间的设置是通过每按下按键一次就将其当前值增加 1 来实现的。如果一个按键要被按下多次，这将是相当乏味的。一个更好的方法是在一个按键被按下时，每 0.5 秒就将时间自动增加 1。为了提供这一功能，需要在硬件层面作出什么样的改变（如果有的话）？修改图 11-6 中的程序来实现这一动能。

[M] 11.9 针对图 11-7 中的程序，重复习题 11.8 中的问题。

[M] 11.10 我们希望在首次开启电源时，闹钟中两个垂直排列的 LED 可以每隔一秒闪烁一次。而一旦用户开始设置时间，它们就应该停止闪烁。修改图 11-6 中的程序来完成这一要求。

[M] 11.11 针对图 11-7 中的程序，重复习题 11.10 中的问题。

441 - 442

## 参考文献

1. Altera's web site at *www.altera.com*.

# 并行处理及性能

## 本章目标

在本章中你将学习以下内容：

- 多线程
- 通用处理器和图形处理器中的向量处理
- 共享存储器的多处理器
- 共享数据的高速缓存一致性
- 分布式存储系统中的消息传递
- 并行编程
- 性能的数学建模



在第 6 章中描述了作为处理器设计技术的指令流水线和超标量操作，其目的是为了提高指令执行速率，进而减少执行时间。在第 8 章中，通过使用高速缓存来减少访问指令和数据的平均延迟。

本章介绍了另外三种用于提高性能的技术，分别为多线程、向量处理和多重处理。这些技术是在通用计算机的多核处理器芯片中实现的，它们通过提高处理资源的利用率和并行执行更多的操作来提高性能。

## 12.1 硬件多线程

操作系统（OS）软件通过在不同的程序间进行上下文切换，使得在同一个处理器上可执行多个不同程序的任务。正如 4.9 节所述，操作系统将一个程序，连同描述其当前执行状态的所有信息视为一个实体，称为进程（process）。每个进程保持着由操作系统分配的关于存储器和其他资源的信息。进程可能与用户在计算机上打开的应用程序（如网页浏览、文字处理、音乐播放等）相关。

每个进程都有一个相应的线程，线程是程序中一条独立的执行路径。更确切地说，线程（thread）是指一个控制线程，其状态包括程序计数器和其他处理器寄存器的内容。我们将在 12.6 节中讨论，多个线程可以执行一个程序的不同部分，而且可以并行运行，就像它们分别对应一个单独的程序一样。两个或多个线程可以在不同的处理器上运行，它们可以对不同的数据执行同一个程序的相同部分，也可以执行同一个程序的不同部分。不同程序的线程也可以在不同的处理器上执行。一个程序中的所有线程都运行在同一个地址空间中并与同一个进程有关。

在这一节中，我们重点介绍在同一处理器上运行两个或多个程序的多任务处理，且每个程序只有一个线程。4.9 节描述了时间片技术，即操作系统从那些目前没有阻塞的进程中选择一个并允许该进程运行很短的一段时间。在时间片中，只有被选择进程所对应的线程才是处于活动状态的。该时间片结束时的上下文切换会使操作系统选择另一个不同的进程，其对应的线程将在下一个时间片中变为活动状态。定时器中断使操作系统调用一个中断服务程序来实现进程间的切换。

为了有效地处理多个线程，一个处理器往往包含多个相同的寄存器组，其中包括多个程序计数器，每组寄存器可专门用于一个不同的线程。因此，在上下文切换期间不会在保存和

恢复寄存器的内容上浪费时间。这种处理器被认为是使用了一种叫做硬件多线程（hardware multithreading）的技术。

通过使用多组寄存器，上下文切换是简单而快速的，所需要做的就是改变处理器中的硬
444 件指针，以使用不同的寄存器组来读取和执行后续的指令。切换到一个不同的线程可以在一个时钟周期内完成。先前的活动线程的状态被保存在它自己的那组寄存器中。

切换到一个不同的线程可以通过一个特定事件的出现而在任何时候触发，而不是在一个固定的时间间隔结束时才发生。例如，活动线程中执行 Load 或 Store 指令时可能会发生高速缓存失效，此时将会访问低速的主存而不是执行延迟操作。处理器可以快速切换到一个不同的线程，并继续读取和执行其他指令。这被称为粗粒度（coarse-grained）的多线程，因为在导致切换到另一个线程的事件（如高速缓存失效）发生之前，一个线程可能已经执行了很多条指令。

另一种由于特定事件而在线程间切换的方法是在每条指令被取出后再切换。这被称为细粒度（fine-grained）或交错（interleaved）的多线程，其目的是为了提高处理器的吞吐量。每一条新指令都是独立于其他线程中的前继指令的，这可以减少由于数据依赖而引起的延迟的发生。因此，通过将多个线程的指令交错起来可以增加吞吐量，但这需要较长的时间来完成给定线程的所有指令。附录 E 中描述的 Intel IA-32 体系结构的处理器中使用了只有两个线程的多线程交错形式。

## 12.2 向量（SIMD）处理

许多计算量大的应用程序使用循环来对数据向量执行操作，在这里向量（vector）是一个元素为整数或浮点数的数组。当处理器执行这种循环中的指令时，它对每个向量元素每次执行一个操作。因此，需要执行很多指令来处理所有的向量元素。

处理器的性能可通过使用多个 ALU 来提高。在这样的处理器中，可以使用一条单一的指令来对多个数据元素并行操作。这样的指令称为单指令流多数据流（single-instruction multiple-data，SIMD）指令，它们也被称为向量指令（vector instruction）。只有在并行执行的操作是独立的时候这些指令才能使用，这通常被称为数据并行性（data parallelism）。

向量指令的数据保存在向量寄存器（vector register）中，每个寄存器可容纳多个数据元素。每个向量寄存器中的元素数量 $L$ 被称为向量长度（vector length），它决定了可以在多个 ALU 中并行执行的操作数量。如果存在向量指令可以使不同大小的数据元素使用相同的向量寄存器，那么 $L$ 可有所不同。例如，Intel IA-32 体系结构拥有 128 位的向量寄存器，其向量长度的范围从 $L=2$ 到 $L=16$，对应的整型数据元素的大小为 64 位到 8 位。

下面通过一些典型的向量指令来说明如何使用向量寄存器。我们假设，在操作码助记符中增加一个后缀 S 来指定每个数据元素的大小，这就决定了一个向量中的元素数量 $L$。对于访
445 问存储器的指令来说，常规寄存器的内容通常用来计算有效地址。

向量指令

VectorAdd.S Vi, Vj, Vk

使用向量寄存器 Vj 和 Vk 中的元素来计算 $L$ 个和，并将结果放到向量寄存器 Vi 中。类似的指令还可用来执行其他的算术运算。

在向量寄存器和存储器之间传输多个数据元素需要使用特别的指令。指令

VectorLoad.S Vi, X(Rj)

使得从存储单元 X + [Rj] 开始的 $L$ 个连续元素被装入向量寄存器 Vi 中。同样，指令

VectorStore.S Vi, X(Rj)

将向量寄存器 Vi 的内容存放到存储器中 $L$ 个连续的单元中。

**向量化**

在用高级语言编写的源程序中，考虑对整数或浮点数数组进行操作的循环。如果每一次循环执行的操作都是与其他次循环相独立的，则称这个循环是可向量化的（vectorizable）。使用向量指令可以减少需要执行的指令数，并可以在多个 ALU 中并行执行多个操作。向量化编译器（vectorizing compiler）可以识别这样的不是太复杂的循环，并生成向量指令。

图 12-1a 给出了一个 C 语言编写的简单示例，来说明一个循环的向量化。假设 A、B 和 C 数组在存储器中的起始位置保存在寄存器 R2、R3 和 R4 中。使用传统的汇编语言指令，编译器可能会生成如图 12-1b 所示的循环，循环体中有 9 条指令，因此经过循环 $N$ 次共要执行 $9N$ 条指令。

为实现向量化循环，编译器必须识别：每次经过循环的计算是与其他次循环相独立的，并且可以同时对多个元素执行同样的操作。为简单起见，假设循环次数 $N$ 是可被 $L$ 整除的。循环开始处的 Load、Add 和 Store 指令由相应的向量指令替代，这些向量指令可一次操作 $L$ 个元素。因此，向量化的循环只需要 $N/L$ 次循环来处理数组中的所有数据。由于每次经过循环会处理 $L$ 个元素，所以寄存器 R2、R3 和 R4 中的地址指针将递增 $4L$，而寄存器 R5 中的计数值将递减 $L$。向量化的循环如图 12-1c 所示。我们假设汇编程序计算 Add 指令中以立即操作数形式给出的表达式 $4L$ 的值。在循环体内仍有 9 条指令，但是，因为现在的循环次数为 $N/L$，所以需要执行的指令总数只有 $9N/L$ 条。

```
for (i = 0; i < N; i++)
  A[i] = B[i] + C[i];
```

a）将向量元素相加的 C 语言循环

```
        Move            R5, #N          R5 为循环计数器
LOOP:   Load            R6, (R3)        R3 指向数组 B 中的一个元素
        Load            R7, (R4)        R4 指向数组 C 中的一个元素
        Add             R6, R6, R7      将数组中的一对元素相加
        Store           R6, (R2)        R2 指向数组 A 中的一个元素
        Add             R2, R2, #4      递增三个数组指针
        Add             R3, R3, #4
        Add             R4, R4, #4
        Subtract        R5, R5, #1      递减循环计数器的值
        Branch_if_[R5]>0 LOOP           如果没有结束则重复循环
```

b）该循环的汇编语言指令

```
        Move            R5, #N          R5 对元素中的元素数量进行计数
LOOP:   VectorLoad.S    V0, (R3)        从数组 B 中加载 L 个元素
        VectorLoad.S    V1, (R4)        从数组 C 中加载 L 个元素
        VectorAdd.S     V0, V0, V1      将数组中的 L 对元素相加
        VectorStore.S   V0, (R2)        将 L 个元素存储到数组 A 中
        Add             R2, R2, #4*L    将数组指针递增 L 个字
        Add             R3, R3, #4*L
        Add             R4, R4, #4*L
        Subtract        R5, R5, #L      将循环计数器的值递减 L
        Branch_if_[R5]>0 LOOP           如果没有结束则重复循环
```

c）该循环的向量化形式

图 12-1　循环的向量化示例

在计算机图形处理和数字信号处理等应用程序中存在着向量化循环。这样的循环中包含许多可以同时对多个数据元素执行的独立计算。如果应用程序的大部分执行时间花费在执行这种类型的循环中，那么这些循环的向量化可以显著减少总的执行时间。性能改善的程度是受向量长度 $L$ 的限制的。它决定了可以并行操作的 ALU 的数量。为获得更高的性能，可以用另一种方式实现对向量（SIMD）处理的支持，我们将在下一节中介绍。

446 / 447

### 图形处理单元

对计算机图形处理不断增长的要求促使了被称为图形处理单元（graphics processing unit，GPU）的专用芯片的发展。GPU 的主要目的是为了加快高分辨率三维图形（如视频游戏）所需要的大量浮点计算。因为在这些计算中所涉及的操作往往是独立的，所以一个大的 GPU 芯片往往包含数百个具有浮点 ALU 的简单内核来并行执行这些独立的操作。

显卡上包含有一个 GPU 芯片和它的专用存储器，这种显卡是插在主机的扩展槽中的，扩展槽使用互连标准，如第 7 章中所讨论的 PCIe 标准。有一个专门为 GPU 芯片的处理内核编写的小程序，很多内核会并行执行该程序，内核执行相同的指令，但却对不同的数据元素进行操作。一个单独的控制程序会在主机的通用处理器中运行，并在必要时调用 GPU 程序。在初始化 GPU 计算之前，主机上的程序必须首先将 GPU 程序所需要的数据从主存传送到 GPU 的专用存储器中。计算完成后，再将所产生的输出数据传回到主存中。

GPU 芯片的处理内核中有一个专门的指令集和硬件体系结构，这与通用处理器所使用的不同。一个例子就是 NVIDIA 公司在其 GPU 芯片的内核中使用的计算统一设备体系结构（Compute Unified Device Architecture，CUDA）。为了便于编写涉及通用处理器和 GPU 的程序，NVIDIA 公司 [1, 2] 已经开发出了一种扩展的 C 编程语言，被称为 CUDA C 语言。它用 C 语言编写程序，使用特殊的关键字来标记需要 GPU 芯片中处理内核执行的功能。编译器和相关的软件工具会自动将目标程序划分成不同的部分并将其分别翻译成主机和 GPU 芯片的机器指令。CUDA C 语言还提供了一些库例程，可以在基于 GPU 的显卡专用存储器中分配存储空间以及在主存和专用存储器之间传输数据。业界还提出了一个称为 OpenCL 的开放标准作为包括任何供应商 GPU 芯片的系统的编程框架 [3]。

## 12.3 共享存储器的多处理器

一个多处理器系统包含多个可同时执行独立任务的处理器。这些任务的粒度可以有很大的不同。一个任务可能只包含单次循环中的几条指令，也可能包含在子程序中执行的数千条指令。

在一个共享存储器的多处理器中，所有处理器都访问同一个存储器。在不同处理器中运行的任务可以使用相同的地址来访问存储器中的共享变量。共享存储器的大小可能是很大的。当多个处理器请求同时访问存储器时，在一个单一的模块内实现一个大的存储器将产生瓶颈。可通过将存储器分布到多个模块中来缓解这个问题，这样从不同处理器同时发出的请求很可能会根据其地址来访问不同的存储器模块。

448

互连网络使得任何处理器都可以访问共享存储器中的任何模块。当存储模块在物理上独立于处理器时，所有访问存储器的请求都必须通过网络来实现。这将引入延迟。图 12-2 显示了这样一种布局。所有从处理器到存储模块的访问都有相同网络延迟的系统被称为统一存储器访问（Uniform Memory Access，UMA）的多处理器。虽然延迟是相同的，但它对连接多个处理器和存储模块的网络而言可能是很大的。

为了获得更好的性能，需要将存储模块放置在接近每个处理器的位置。这样就形成了一

个节点（node）集合，每个节点包含了一个处理器和一个存储模块。然后，这些节点连接到网络中，如图 12-3 所示。当处理器发出访问本地存储器的请求时，可以避免网络延迟。然而，一个访问远程存储模块的请求必须通过网络。由于访问本地存储器和远程共享存储器的延迟是有差异的，所以这种类型的系统被称为非统一存储器访问（Non-Uniform Memory Access，NUMA）的多处理器。

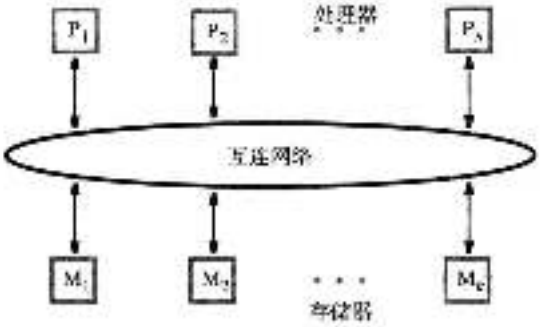



图 12-2　UMA 多处理器

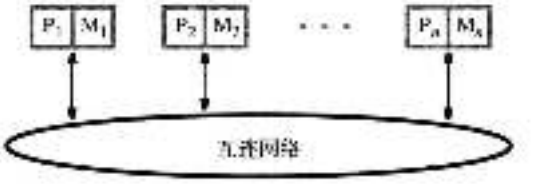



图 12-3　NUMA 多处理器

[449]

## 互连网络

互连网络必须能够允许信息在系统中的任意两个节点之间进行传输，网络还可以用来从一个节点向许多其他的节点广播信息。其中网络流量是由请求（例如读和写）和数据传输组成的。

某个特定网络是否适用往往是通过它的成本、带宽、有效吞吐量和实现的难易程度等方面来判断的。带宽（bandwidth）是指用来传输数据的传输链路的能力，它使用每秒传送的位数或字节数来表示。有效吞吐量（effective throughput）是数据传输的实际速率。这个速率是小于可用带宽的，因为一个给定的链路还必须承载用于协调数据传输的控制信息。

信息通过网络传输通常是以固定长度和指定格式的数据包（packet）形式进行的。例如，一个读请求可能是一个从处理器发送到存储模块的单一数据包。该数据包包含源和目的节点的标识符、将要被读取的单元的地址，以及一个用来指示读操作类型的命令字段。一个向存储模块中写入一个字的写请求也可能是一个包含将要被写入的数据的单一数据包。而另一方面，一个读响应可能会涉及整个高速缓存块，这需要使用多个数据包来进行数据传输。

理想情况下，一个完整的数据包可以在一个时钟周期内在网络中的任何节点或交换机上并行处理，这就意味着要有许多电线构成较宽的链路。然而，为了降低成本和复杂性，通常需要尽可能地减少链路的宽度。在这种情况下，一个数据包必须被划分成较小的块，以使于每一个小块都可以在一个时钟周期内被传输完成。

以下各小节将描述一些在多处理器中普遍使用的互连网络。

### 1. 总线

总线（bus）是指为信息传输提供共享路径的一组线路（电线）。如第 7 章所述，UMA 多处理器中通常使用总线来连接多个处理器和多个共享的存储模块。为了确保在任何时候只授权一个请求使用总线，需要使用仲裁。总线适用于处理器数量相对较少的情况，因为当连接了多个处理器时，对总线的访问会产生竞争，而且电力负荷会增加传播延迟。

一个简单的总线在当前请求得到响应之前不允许新的请求出现在总线上。然而，如果响应延迟很高，就有可能造成总线上出现相当大的空闲时间。

通过使用分裂事务（split-transaction）总线可以实现更高的性能。在这种总线中，请求及其相应的响应会被视为两个单独的事件，而其他的传输有可能在它们之间发生。考虑这样一种情况，其中多个处理器需要向存储器发出读请求。我们使用仲裁选择第一个处理器来使用总线发送请求。当其请求发送后，再选择第二个处理器来使用总线发送请求，而不让总线空闲。假定这个请求是对另一个不同的存储模块进行访问，那么这两个读访问可以并行进行。如果两个模块都没有完成其访问，那么将选择第三个处理器来发送其请求，并以此类推。最终，一个存
[450] 储模块会完成其读访问，它会被授予使用总线将数据传输给发送请求的处理器。当其他模块完成它们的访问后，总线就用来传输它们的响应。每个请求与相应的响应之间的实际时间长度可能会有所不同，因为存储器中不同事务的请求和响应是在总线上交错的，以便有效地利用可用带宽。

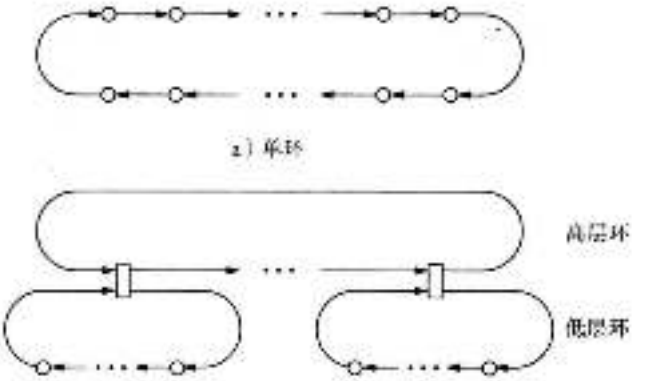
分割事务总线需要一个更复杂的总线协议。因为它需要将每个响应与其对应的请求相匹配。这通常是通过将一个独特的标签与总线上出现的每个请求相关联来处理的。然后每个响应以适当的标签出现，从而可以将其与原来的请求进行匹配。

### 2. 环状网

环状网是由节点之间的点对点连接形成的，如图 12-4 所示。图 12-4a 显示了一个单环。一个较长的单环会导致任意两个节点间通信的平均延迟较高。有两种不同的方法可以降低延迟。

一种方法是增加第二个环来以相反的方向连接节点。由此产生的双向环（bidirectional ring）可将平均延迟减半，并使带宽加倍。然而，通信的处理会更复杂。

另一种方法是使用层次结构（hierarchy）的环。图 12-4b 所示的是一个两层结构的环。高层的环连接低层的环。这种安排可以减少低层环上任意两个节点间通信的平均延迟。同一低层环上节点之间的传输不需要经过高层环。不同的低层环上节点之间的传输需要经过高层环的一部分。这种层次方案的缺点是：当不同的低层环上有许多节点经常相互通信时，高层环可能会成为瓶颈。




a）单环

b）环的层次结构

[451] 图 12-4 基于环的互连网络

### 3. 交叉开关网

交叉开关网（crossbar）是一种可以在连接到网络上的任意一对单元之间提供直接链接的网络，它通常用于 UMA 多处理器系统中，用以连接处理器和存储模块。如果多个请求的目的地不同，那么交叉开关网络可以使得多个传输同步进行。例如，我们可以使用图 12-5 所示的交叉开关集合来实现图 12-2 中的结构。对于 $n$ 个处理器和 $k$ 个存储器，需要 $n \times k$ 个开关。

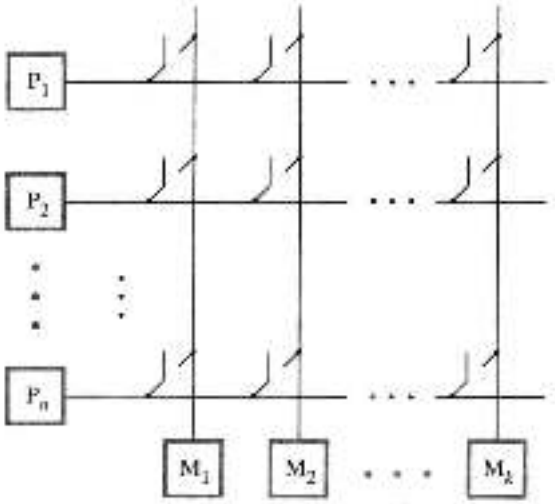



图 12-5　交叉开关互连网络

#### 4. 网状网

将大量节点连接起来的一种较自然的方式是二维网状网（mesh），如图 12-6 所示。网状网内部的每个节点有 4 个连接，其中每一个连接与其水平和垂直方向的相邻节点相连。而网状网边界和角落上的节点由于有较少的相邻节点，因此只有较少的连接。为了减少网状网中相隔很远的节点之间的通信延迟，可以在网状网相对边界的节点间引入环绕连接。具有这种连接的网络被称为圆环（torus）。一个圆环中的所有节点都有四个连接。平均延迟被降低了，但相对简单的网状网而言，通过圆环的路由请求和响应实现起来较为复杂。

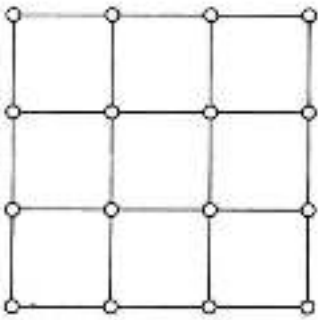


图 12-6　一个二维的网状网络

## 12.4　高速缓存一致性

共享存储器的多处理器比较易于编程。程序中的每一个变量都在存储器中有一个唯一的地址单元，任何处理器都可以访问该地址单元。然而，每个处理器都有它自己的高速缓存，因此，在几个高速缓存中可能保存着共享数据的副本。当任何一个处理器在它自己的高速缓存中对共享变量进行写操作时，那么所有包含该变量副本的其他高速缓存中将会有旧的、不正确的值。所以必须将这一变化告知相关的高速缓存，以使于它们可以将其副本更新为新值或者变为无效。这就是保持高速缓存一致性（cache coherence）的问题，它要求多个高速缓存中的共享数据要有一致的值。

在第 8 章中我们讨论了两种对高速缓存中的数据进行写操作的基本方法。直接写（write-through）方法是对高速缓存和主存中的数据同时修改，而写回（write-back）方法只改变高速缓存中的数据，而主存中的副本只在高速缓存中被修改的数据块必须要被替换时才会被更改。多处理器系统中的高速缓存一致性也可以采用类似的方法来解决。

### 12.4.1　直接写协议

直接写协议可以通过两种方式实现。一种方式是更新其他高速缓存中的值，第二种方式

是使其他高速缓存中的副本无效。

我们先来看一下更新（update）协议。当处理器对其高速缓存中的数据块写入一个新值时，这个新值也将被写入包含该修改块的存储模块中。由于这个块的副本在其他高速缓存中也可能存在，所以这些副本也必须被同时修改以反映出由该写操作所引起的变化。最简单的实现方法是向系统中所有处理器的高速缓存广播要写入的数据。当每个处理器接收到这个广播数据时，如果在它的高速缓存中存在这个受影响的高速缓存块，那么处理器就修改该块的内容。

实现直接写协议的第二种方式是使副本无效（invalidation）。当一个处理器向它的高速缓存中写入一个新值时，这个新值也被发送到存储器中的适当位置，并且使其他高速缓存中的所
453 有副本都无效。同样可以采用广播的方式在系统中发送使副本无效的请求。

### 12.4.2 写回协议

使用写回协议保持一致性是基于存储器中数据块的所有权概念的。最初，存储器是所有块的所有者，然后存储器保留由处理器读取的任何数据块的所有权，并将一个副本放到处理器的高速缓存中。

如果某个处理器想要向其高速缓存中写入一个数据块，那么它必须首先成为此数据块的独家所有者。要实现此目的，必须先通过一个广播请求使其他高速缓存中的所有副本都无效。然后该块的新所有者就可以随意修改相关内容，而无需采取任何其他行动。

当另一个处理器想要读取一个已被修改的块，其请求必须被转发到该块的当前所有者。然后再由当前所有者将数据发送给提出请求的处理器。该数据也会被发送给相应的存储模块，该模块会重新获取所有权并更新存储器中该块的内容。作为该块之前所有者的处理器，其高速缓存中还保留有该块的副本。此时，该块通过在两个高速缓存和存储器中的副本来实现共享。来自其他处理器的读取该数据块的后续请求由包含该数据块的存储模块响应。

当另一个处理器希望对一个已被修改的块进行写操作时，该块的当前所有者将数据发送给提出请求的处理器，同时将该块的所有权传送给提出请求的处理器并使其自身所缓存的副本无效。由于该块正在被新的所有者修改，所以存储器中该块的内容还没有被更新。因此，对该数据块的下一个请求将由新的所有者响应。

写回协议与直接写协议相比具有通信量少的优势。这是因为一个高速缓存块在被其他的处理器需要之前，一个处理器可能对这个块执行了若干次写操作。若使用写回协议，一旦获得所有权并发送一个无效请求后，这些写操作就只在该处理器自己的高速缓存中进行。而若使用直接写协议，每个写操作还必须对适当的存储模块进行操作，并向其他高速缓存进行广播。

到目前为止，我们一直假定这些协议中的更新和无效请求操作都是通过互连网络来广播的。实现这种广播的难易在很大程度上取决于互连网络的结构。支持广播的最自然的网络是单总线。在使用单总线将适量的处理器连接到存储模块上的多处理器系统中，高速缓存的一致性是利用监听方法来实现的。

### 12.4.3 监听高速缓存

在单总线系统中，处理器和存储模块之间的全部交互都是通过总线请求和响应完成的。实际上，它们是向连接到总线上的所有部件进行广播。假设每个处理器的高速缓存都有一个控制电路来观察或监听（snoop）总线上的所有交互情况。现在我们将描述写回协议的某些情况，
454 以及如何保持高速缓存一致性。

考虑一个处理器，它之前已经从存储器中读取了一个块的副本到它的高速缓存中。在

第一次对这个块进行写操作之前，处理器必须向其他所有的高速缓存广播无效请求（invalidation request），而其他高速缓存的控制器接受请求并使其保存的该块的副本无效。这个操作会导致提出请求的处理器成为该块的新所有者。然后这个处理器可以对该块进行写操作并将其标记为被修改的。该处理器再对其高速缓存中已被修改的这个块进行写操作时就不需要进行广播了。

现在，如果另一个处理器在总线上广播对该块的读请求（read request），存储器不能响应，因为它不是该块的当前所有者。拥有该请求块的处理器监听总线上的读请求。因为它在其高速缓存中保存了该请求块修改后的副本，所以它在总线上发出一个特殊的信号以防止存储器进行响应。然后所有者在总线上广播该块的副本，并将其标记为干净的（未被修改的）。总线上的数据响应被发出读请求的处理器的高速缓存接受。该数据响应同样也被存储器接受以更新其保存的该块的副本。这时，存储器重新获得该块的所有权，该块处于共享状态，因为其副本存在于两个处理器的高速缓存中。这两个被缓存的副本和存储器中的副本都包含相同的数据，所以就保持了一致性。此后来自任何处理器的后续请求都是由存储器来响应的。

现在考虑这样一种情况，两个处理器在其各自的高速缓存中存有相同块的副本，它们试图在同一时间对相同的高速缓存块进行写操作。由于该块处于共享状态，所以存储器是该块的所有者。因此，这两个处理器都请求使用总线来广播一条无效消息。其中一个处理器首先被授权使用总线，它将其无效请求进行广播并成为该块的新所有者。通过监听，另一个处理器高速缓存中该块的副本被置为无效。当该处理器后来被授权使用总线时，它将广播一个独占性的读请求（read-exclusive request）。这一请求结合了对同一块的读请求和无效请求。第一个处理器的控制器监听该独占性的读请求，在总线上提供一个数据响应，并将其高速缓存中的副本置为无效。因此该块的所有权被转移到第二个提出请求的处理器。因为该块马上要被再次修改，所以存储器中的内容不会更新。由于这两个处理器的请求被按顺序处理，所以在任何时候都保持了高速缓存的一致性。

刚才所描述的方案是基于高速缓存控制器能够观察总线上的活动并采取适当措施的能力的，这样的方案被称为监听高速缓存（snoopy-cache）技术。

出于性能方面的原因，监听功能是不能干扰处理器和其高速缓存的正常操作的。如果对于总线上的每个请求，高速缓存控制器都要访问高速缓存的标志位，便会产生这种干扰。在大多数情况下，高速缓存中不包含相关请求块的有效副本。为了消除不必要的干扰，每个高速缓存都提供了一套与高速缓存中块的状态信息相同的重复标志，但这些标志信息可以被监听电路单独访问。

455

### 12.4.4　基于目录的高速缓存一致性

监听高速缓存的概念在单总线系统中是很容易实现的。大型的共享存储器多处理器系统使用诸如环状网和网状网的互连网络。在这种系统中，将每个请求广播到所有处理器的高速缓存是无效率的。一个可扩展的，但更复杂的方案是在每个存储模块中使用目录（directory）来指出哪些节点中可能包含一个处于共享状态的给定块的副本。如果一个块被修改，目录将标识作为当前所有者的那个节点。每个来自处理器的请求都必须首先发送到包含有关块的存储模块中，用该块的目录信息来确定需要采取何种操作。如果该块被修改，那么读请求将被转发给该块的当前所有者。对于向一个共享块发出写请求的情况，无效请求只单独发送给包含有该块副本的节点。这种基于目录的实现高速缓存一致性的方案，由于其成本和复杂性，限制了它只能在大型系统中使用。小型的多处理器系统，包括目前的多核芯片，通常使用监听技术。

## 12.5 消息传递多计算机

使用多个处理器的另一种不同方式是将系统中的每个节点作为拥有它自己存储器的一台完整计算机来实现。系统中的其他计算机不能直接访问该计算机的存储器。需要共享的数据是通过在计算机之间发送消息来进行交换的。这种系统被称为消息传递多计算机（message-passing multicomputer）系统。

消息传递多计算机系统中并行程序的编写是不同于共享存储器的多处理器系统的。为了在节点间共享数据，在数据源计算机上运行的程序必须发送包含相关数据的消息到目的计算机上。在目的计算机上运行的程序收到消息后，再将数据复制到那个节点的存储器中。

为了方便消息传递，每个节点上的一个特殊通信部件通常负责格式化和解释所发送和接收消息的低层细节，并负责向或从节点的存储器复制消息数据。每个节点中的计算机发送命令到通信部件，然后计算机继续执行其他计算，而通信部件将会处理所发送和接收消息的细节。

## 12.6 多处理器并行编程

前几节讨论了共享存储器多处理器系统的硬件组成，它可以利用应用程序中的并行性。而并行性可能存在于循环中（其每次循环都是独立的），也可能存在于独立的高级任务中。

456 用高级语言编写的源程序允许程序员以易于理解的方式来表示所需的计算，它必须被编译器和汇编程序翻译成机器语言表示形式。处理器的硬件是被设计为按顺序执行机器语言指令的，以便能执行程序员所需的计算。它不能自动识别独立的、可以并行执行的高级任务。编译器也存在检测和利用并行性的局限性。因此，程序员需要将源程序中的整体计算划分为相关任务，并指定它们是如何在多个处理器上执行的。

共享存储器的多处理器系统中的编程是单一处理器中传统编程的自然延伸。高级语言源程序是用处理器可执行的任务编写的。但某些任务也可以在不同的处理器中同时执行。通过定义不同的处理器在执行分配给它们的任务时所读写的全局变量，可实现数据共享。当前通用计算机中使用的多核芯片，如实现 Intel IA-32 体系结构的多核芯片，就是以这种方式来编程的。

为了说明并行编程，我们考虑一个计算两个向量的点积的示例，其中每个向量包含 $N$ 个数，图 12-7 给出了一个实现该任务的 C 语言程序。此例中，初始化两个向量内容的细节被省略，而把重点放在并行编程的有关方面。

```
#include   <stdio.h>      /* 输入/输出例程 */
#define    N    100       /* 每个向量中的元素数 */
double     a[N], b[N];    /* 计算点积的向量 */
void       main (void)
{
           int i;
           double dot_product;

           < Initialize vectors a[], b[] – details omitted >
           dot_product = 0.0;
           for (i = 0; i < N; i++)
              dot_product = dot_product + a[i] * b[i];
           printf ("The dot product is %g\n", dot_product);
}
```

图 12-7 计算点积的 C 程序

循环中将 $N$ 个乘积的总和累加起来，每次循环都依赖于前一次循环所计算的部分和。在

最后一次循环中计算出的结果就是点积。尽管每次循环是相互依赖的，但还是可以通过利用加法的关联属性，将程序划分为独立的任务，以便同时执行。每个任务计算一个部分和，最终的结果可通过将所有的部分和相加而得到。

457

要实现上述计算点积的并行程序，我们需要回答以下两个问题：

- 如何使多个处理器参与到并行计算部分和的操作中？
- 如何确保在计算点积的最终结果前，每个处理器都已完成其部分和的计算？

1. 线程的创建

为回答第一个问题，我们将定义可分配给不同处理器执行的任务，然后描述这些任务是如何在多个处理器中执行的。我们可以编写一个并行版本的计算点积的程序，用参数 $P$ 表示处理器的数量，参数 $N$ 表示每个向量中的元素数。为简单起见，我们假定 $N$ 可被 $P$ 整除。整个计算涉及 $N$ 个乘积的和，对于 $P$ 个处理器，我们定义 $P$ 个独立任务，其中每个任务用于计算 $N/P$ 个乘积的部分和。

当一个程序在一个单处理器中执行时，有一个活动的执行控制线程。该线程是由操作系统（OS）在开始执行程序时隐式创建的。对于并行程序，我们需要使用多个执行控制线程（每个处理器一个）来单独处理独立的任务。这些线程必须被明确地创建。一个典型的方法是使用程序库中名为 create_thread 的例程来支持并行编程。该库例程接受一个输入参数，该参数是指向新创建线程所要执行的子程序的指针。该库例程调用操作系统的服务来创建一个具有独立堆栈的新线程，以便于它可以调用其他的子程序以及拥有其自己的局部变量。而所有的全局变量是在所有的线程之间共享的。

将线程彼此区分开来是很有必要的。一种方法是提供一个称为 get_my_thread_id 的库例程，它为每个线程返回一个介于 0 和 $P-1$ 之间的唯一整数。有了这些信息，线程便可以确定整个计算中它所负责的适当的子集。

2. 线程同步

第二个问题需要确定线程何时完成它们的任务，从而可以正确地计算最终的结果。这就需要多线程的同步（synchronization）技术了。有几种线程同步的方法，它们往往是通过为并行编程增加额外的库例程来实现的。在这里，我们讨论一种称为屏障（barrier）的方法。

屏障的目的是迫使线程等待，直到它们到达程序中某个指定的位置为止，在该位置处线程会调用屏障库例程。每一个线程在调用屏障例程后会进入一个忙等循环，直到最后一个线程调用屏障例程后，所有的线程继续执行。这就确保了线程都已完成了屏障调用指令前各自的计算。

3. 并行程序示例

前面已描述过关于线程的创建和同步的问题，以及用于线程管理的典型库例程，现在我们介绍一个并行的点积程序示例。图 12-8 显示了主程序和另一个称为 ParallelFunction 的例程，ParallelFunction 例程定义了并行执行的独立任务。当程序开始执行时，只有一个线程执行主程序。这个线程初始化向量，然后初始化屏障同步所需的共享变量。为了启动并行执行，需要在主程序中调用 $P-1$ 次 create_thread 例程来创建额外的线程，其中每一个线程都执行 ParallelFunction。然后，执行主程序的线程直接调用 ParallelFunction，使得 $P$ 个线程都参与到整体计算中。操作系统软件负责将线程分布到不同的处理器上并行执行。

458

每个线程都需要从 ParallelFunction 中调用 get_my_thread_id 来获得一个介于 0 至 $P-1$ 之间的唯一标识。利用此信息，线程可以为生成其部分和的循环计算开始和结束标志。执行完循环后，线程会使用它唯一的标识符作为数组的索引，将结果写入到共享数组 partial_sums 中一个单独的元素中。然后，线程调用屏障同步库例程来等待其他的线程完成计算。

最后一个线程完成其计算调用屏障例程后，所有的线程将返回到 ParallelFunction，而在 ParallelFunction 中没有进一步的计算了，因此这 $P-1$ 个在主程序中由库调用创建的线程都终止运行。从主程序中直接调用 ParallelFunction 的线程会返回，并使用 partial_sums 数组中的值来计算最终结果。

图 12-8 中的程序使用通用的库例程来说明线程的创建和同步。IEEE 1003.1 标准 [4] 中定义了 C 语言中并行编程的一个例程集合，这个集合也被称为 POSIX 线程或 Pthreads 库，它提供了多种线程管理和同步的机制，这个库的实现可用于各种广泛使用的操作系统中，以方便多处理器编程。

```
#include   < stdio.h>          /* 输入 / 输出例程 */
#include   "threads.h"         /* 线程创建 / 同步例程 */

#define    N    100            /* 每个向量中的元素数 */
#define    P    4              /* 并行执行的处理器数 */

double     a[N], b[N];         /* 用于计算点积的向量 */
double     partial_sums[P];    /* 由线程计算的结果数组 */
Barrier    bar;                /* 支持屏障同步的共享变量 */

void       ParallelFunction (void)
{
           int my_id, i, start, end;
           double s;

           my_id = get_my_thread_id ();    /* 获取线程的唯一标识符 */
           start = (N/P) * my_id;          /* 使用线程标识符确定其开始 / 结束 */
           end = (N/P) * (my_id + 1) - 1;  /* 假设 N 可被 P 整除 */
           s = 0.0;
           for (i = start; i <= end; i++)
             s = s + a[i] * b[i];
           partial_sums[my_id] = s;        /* 将结果保存到数组中 */
           barrier (&bar, P);              /* 与其他线程同步 */
}

void       main (void)
{
           int i;
           double dot_product;

           < Initialize vectors a[], b[] - details omitted.>
           init_barrier (&bar);
           for (i = 1; i < P; i++)         /* 创建 P-1 个额外的线程 */
             create_thread (ParallelFunction);
           ParallelFunction();             /* 主线程也加入并行执行 */
           dot_product = 0.0;              /* 屏障同步后，计算最终结果 */
           for (i = 0; i < P; i++)
             dot_product = dot_product + partial_sums[i];
           printf ("The dot product is %g\n", dot_product);
}
```

图 12-8 计算点积的并行 C 程序

## 12.7 性能建模

衡量一台计算机性能最重要的方式是看它执行程序的速度有多快。在考虑一个处理器时，其指令的读取和执行速度是受指令集体系结构和硬件设计的影响的，执行的指令总数是受编译

器以及指令集体系结构的影响的。第 6 章介绍了基本的性能等式，这是一个低级的数学模型，它反映了一个处理器的上述考虑因素。在该模型中的术语包括执行的指令数、每条指令的平均周期数以及时钟频率。该模型给出了足够详细的信息使得执行时间可被预测。

有一个更高级的模型，它可根据较少的详细信息来评估潜在的性能提升情况。考虑一个程序，它在某台计算机上的执行时间为 $T_{orig}$。我们的目标是评估当引入诸如并行处理这样的性能增强方式时执行时间可减少的程度。假定执行时间的一小部分 $f_{enh}$ 会受性能增强的影响，而其余的部分 $f_{unenh}=1-f_{enh}$ 是不变的。假设 $p$ 表示部分时间 $f_{enh}\times T_{orig}$ 由于性能增强而减少的因子，则新的执行时间为

459
460

$$T_{new}=T_{orig}\,(f_{unenh}+f_{enh}/p)$$

加速比（speedup）为 $T_{orig}/T_{new}$ 或者

$$1/(f_{unenh}+f_{enh}/p)$$

上面加速比的表达式被称为 Amdahl 定律（Amdahl's Law），它是以第一个正式提出上述推理的 Gene Amdahl 来命名的。如果某种性能增强可以影响大部分的执行时间，那么该定律可以直观地判断其收益的增加情况。

当确定了原始执行时间中可增强部分的比例后，就可用该定律确定可能的加速比上限。我们使用 $p\to\infty$ 来反映执行时间的 $f_{enh}$ 部分减少到零的理想情况，但这是不现实的。由此得到的加速比是 $1/f_{unenh}$，这意味着没有增强的执行时间部分是性能的限制性因素。一个较小的 $f_{unenh}$ 值可提供一个较大的加速比上限。例如，$f_{unenh}=0.1$ 给出的上限为 10，但 $f_{unenh}=0.05$ 给出的上限则为 20。然而，使用一个实际的 $p$ 值时预期的加速比往往远低于上限。例如，$p=16$ 而 $f_{unenh}=0.05$ 时加速比只有 1/（0.05+0.95/16）= 9.1，远低于上限 20。

本次讨论的重要结论是：原有执行时间的未增强部分会大大限制可实现的加速比，即使可增强部分能够提高任意倍数。如果在采用特定的增强措施之前，一个程序员可以，即便是只能大致地确定执行时间的 $f_{unenh}$ 和 $f_{enh}$ 部分，那么 Amdahl 定律就可以为预期的性能改善提供有益的见解。利用这些信息可以确定预期的性能收益是否值得为实现性能增强所付出的努力和代价。

## 12.8　结束语

流水线和高速缓存这样的基本技术是提高性能的重要途径，它们被广泛应用于计算机编程中。多线程、向量（SIMD）处理和多重处理等技术通过更有效地利用处理资源和并行执行多个操作可进一步改善性能。这些技术已被应用到通用多核处理器芯片中。

461

## 习题

[M] 12.1 假设总线传输需要 $T$ 秒，存储器访问时间为 $4T$ 秒。一个经过传统总线的读请求需要 $6T$ 秒完成。在相同的时间延迟下，需要多少根传统的总线才能等于或超过分割事务总线的有效吞吐量？只考虑读请求，忽略存储器冲突，并假定所有的存储模块都连接到多总线中的所有总线上。如果存储器访问时间增加的话，你的答案是增加还是减少？

[M] 12.2 假设一个计算包括 $k+1$ 个不同的任务，为了编写一个程序来实现所需的计算，用 C 语言将每一个任务都编写成一个函数，$k+1$ 个函数分别是 T0()、T1()、…、Tk()。执行每个函数需要 $t$ 个时间单位。由于数据依赖关系，函数 T1() 到 Tk() 必须在 T0() 之后执行，而函数 T1() 到 Tk() 之间没有数据依赖关系。

（a）使用给定的函数，写一个在单处理器上运行的 C 程序。

（b）写一个等价的 C 程序，但可以在 $k$ 个处理器上运行。

（c）推导出（b）部分中的程序相对于（a）部分中的程序的理想加速比表达式。

[D] 12.3 在保持高速缓存一致性中支持和反对无效策略与修改策略的观点分别是什么?

[D] 12.4 共享存储器和消息传递方法都支持相互交互的任务同时执行。这两种方法中哪一种方法更容易仿真另外一种方法?简要说明你的理由。

[E] 12.5 参考图 12-1，假设一个数组的大小 $N=32$。在处理器中执行每一条指令都需要一个时间单位，请计算向量长度为 4、8、16 和 32 时的向量化加速比。

[M] 12.6 对于向量化，效率被定义为加速比和向量长度的比率。请计算习题 12.5 中每个结果的效率。根据向量长度评论效率的变化。

[M] 12.7 在图 12-8 中计算点积的并行程序中，所有线程都到达屏障后，由一个处理器来计算所有部分积的和。修改程序，使得每个线程都将其部分和递增地累加到点积的一个全局和中。为了防止两个或多个线程试图同时修改全局和，需要采用同步机制。一种方法是使用一个共享的计数器变量，其值是被授权独占访问全局和变量的线程标识符。在线程独占访问并更新了全局和之后，它可以简单地增加这个计数器的值，以将独占访问权授予另一个线程。

[E] 12.8 假设一个多处理器系统中有 8 个处理器。根据在一个处理器上运行的现有程序，编写一个并行
462 程序使其在该多处理器系统上运行。假设程序并行部分的工作量可以均匀地分布在这 8 个处理器上。使用 Amdahl 定律来计算加速比为 5 时 $f_{par}$ 的值。

## 参考文献


1. NVIDIA Corporation, *NVIDIA C CUDA Programming Guide*, Version 3.1.1, July 2010, available at http://www.nvidia.com.
2. D. Kirk and W. M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, Burlington, Massachusetts, 2010.
3. Khronos Group, *OpenCL Introduction and Overview*, June 2010, available at http://www.khronos.org/opencl.
463 · 464
4. IEEE, *1003.1 Standard for Information Technology—Portable Operating System Interface (POSIX): System Interfaces, Issue 6*, 2001, available at http://www.ieee.org.

# 逻辑电路

## 附录目标

本附录简要介绍逻辑电路，你将学习以下内容：

- 逻辑函数的组合
- 触发器、寄存器和计数器
- 译码器和多路复用器
- 有限状态机
- 可编程逻辑器件
- 逻辑电路的实现



数字计算机中的信息是由称为逻辑电路（logic circuit）的电子网络表示和处理的。这些电路对二进制变量进行操作，我们可以为二进制变量（binary variable）赋两个不同的值，通常是0和1。本附录将对逻辑函数和实现电路做简要描述，并对集成电路技术做简单介绍。

## A.1 基本逻辑函数

下面用一个每个家庭都会遇到的实际问题来介绍二进制逻辑。来看一个由两个开关 $x_1$、$x_2$ 控制的灯泡。每个开关都只有两个位置：0或1，如图A-1a所示，这样就可以用一个二进制变量表示开关，我们将开关的名称作为其对应二进制变量的名称。图中还有一个电源和一个灯泡。开关端口的连接方式决定了开关控制灯的方式。只有存在一个从电源经开关网络到灯泡的闭合回路时，灯才会亮。我们用二进制变量 $f$ 表示灯的状态。如果灯亮，$f=1$；如果灯不亮，$f=0$。这样，$f=1$ 说明电路中至少有一个闭合回路；而 $f=0$ 说明没有闭合回路。显然，$f$ 是变量 $x_1$ 和 $x_2$ 的函数。

下面看一些控制灯状态的可能情况。首先，假设任一开关在1的位置时灯都会亮，即当

$$x_1 = 1 \text{ 且 } x_2 = 0$$

或

$$x_1 = 0 \text{ 且 } x_2 = 1$$

或

$$x_1 = 1 \text{ 且 } x_2 = 1$$

时，$f=1$。

实现这种控制的电路连接方式如图A-1b所示。电路图旁的逻辑真值表（truth table）表示了这一情况。表中列出了所有可能的开关状态及在每个状态下的 $f$ 值。在逻辑术语中，这个表格表示了两个变量 $x_1$ 和 $x_2$ 的“或”（OR）函数。这个操作在代数中用符号“+”或符号“∨”表示，即

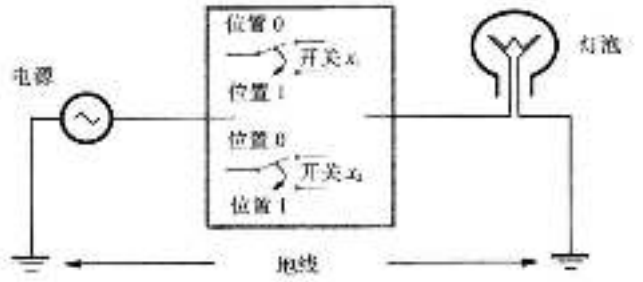
$$f = x_1 + x_2 = x_1 \vee x_2$$

我们称 $x_1$、$x_2$ 为输入（input）变量，$f$ 为输出（output）函数。

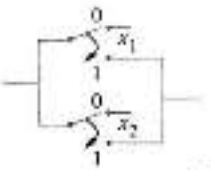
下面是OR操作的一些基本性质。它是可交换的，即

$$x_1 + x_2 = x_2 + x_1$$

这一性质可扩展到 $n$ 个变量中，即

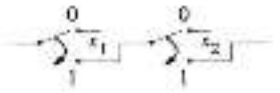



a) 由两个开关控制的灯泡



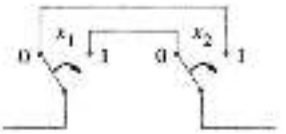

| $x_1$ | $x_2$ | $f(x_1, x_2) = x_1 + x_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

b) 并联（“或”控制）




| $x_1$ | $x_2$ | $f(x_1, x_2) = x_1 \cdot x_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

c) 串联（“与”控制）




| $x_1$ | $x_2$ | $f(x_1, x_2) = x_1 \oplus x_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

d) 异或联接（“异或”控制）

图 A-1 电灯开关示例

$$f = x_1 + x_2 + \cdots + x_n$$

如果任意一个 $x_i$ 的值为 1，$f$ 的值就为 1。这反映了在图 A-1b 中的两个开关上并联更多开关时的效果。而且，通过观察真值表可以看出

467

$$1 + x = 1$$

$$0 + x = x$$

现在，假设只有当两个开关都在 1 的位置时灯才会亮。这个状态的电路连接和相应的真值表表示如图 A-1c。这种情况为“与”（AND）操作，使用符号“·”或“∧”表示。即

$$f = x_1 \cdot x_2 = x_1 \wedge x_2$$

“与”操作的一些基本性质有

$$x_1 \cdot x_2 = x_2 \cdot x_1$$

$$1 \cdot x = x$$

$$0 \cdot x = 0$$

"与"函数也可扩展到 $n$ 个变量，即

$$f = x_1 \cdot x_2 \cdots x_n$$

上式只有当所有的 $x_i$ 变量都为 1 时才得 1。这反映了在图 A-1c 中两个开关上串联更多的开关时的情况。

最后要讨论的这种控制灯状态的开关方式是另一种常见的情况。假设走廊的两端都有开关，每个开关都应该能控制灯的开或关。即：如果灯是亮的，改变任一个开关的位置都应将其关闭；而如果灯是灭的，改变任一个开关的位置都应将其打开。假设两个开关都在 0 位置时，灯是灭的。那么将任一个开关置为 1 都能将灯打开。现在假定灯是亮的，且 $x_1 = 1$，$x_2 = 0$。显然将开关 $x_1$ 变为 0 可以把灯关闭。此外，将 $x_2$ 置为 1 也应该能将灯关闭，即当 $x_1 = x_2 = 1$ 时，$f = 0$。实现这种控制的电路如图 A-1d 所示。相应的逻辑操作称为"异或"（XOR，EXCLUSIVE-OR）操作，使用符号"⊕"表示。它的性质有

$$x_1 \oplus x_2 = x_2 \oplus x_1$$
$$1 \oplus x = \bar{x}$$
$$0 \oplus x = x$$

其中 $\bar{x}$ 代表 $x$ 的"非"（NOT）操作。单变量的函数 $f = \bar{x}$，当 $x = 0$ 时取 1，当 $x = 1$ 时取 0。我们称输入 $x$ 被取反（inverted）或求补（complemented）了。 468

### 电子逻辑门

使用闭合或断开电路的开关以及灯泡来说明逻辑变量和函数的思想是非常形象和方便的。前面介绍的逻辑概念同样适用于数字计算机中处理信息的电子电路。但物理变量是电路的电平和电流，而不是开关的位置以及闭合或断开的电路。例如，考虑一个电路，其输入电平为 +5 伏或 0 伏。电路输出电平同样是 +5 伏或 0 伏。现在，如果用 +5 伏代表逻辑 1，0 伏代表逻辑 0，那么就可以指定该电路逻辑操作的真值表，从而描述电路的作用。

借助于晶体管，我们可以设计出简单的电路完成与、或、异或以及非操作。习惯上将这些基本逻辑电路称为门（gate）。图 A-2 给出了这些门的标准符号。在逻辑门的输入或输出反相时，我们使用一种更为紧凑的图形记号来表示非操作，即用一个小圆圈表示。

逻辑门的电路实现将在 A.5 节中进行讨论。接下来继续讨论怎样使用逻辑门构造一些具有更复杂逻辑功能的逻辑网络。

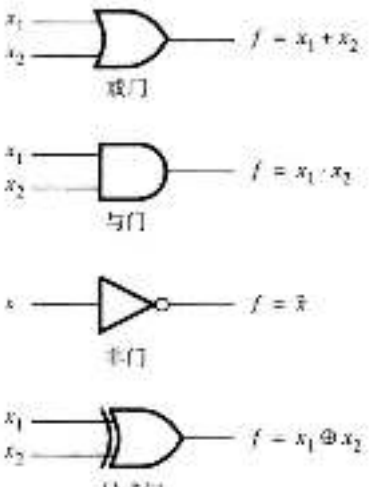



图 A-2 标准逻辑门符号 469

## A.2 逻辑函数的组合

考虑图 A-3a 中由两个与门和一个或门组成的网络。它可以用下面的式子表示

$$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

图 A-3b 是这个表达式的真值表。首先，与门的值由每个输入值决定。而函数 $f$ 的值由或操作决定。$f$ 的真值表与异或函数的真值表一样，所以图 A-3a 使用与门、或门和非门实现了异或函数。逻辑表示式 $x_1 \cdot \bar{x}_2 + \bar{x}_1 \cdot x_2$ 称为积之和（sum-of-product）的形式，因为 OR 操作有时被称作

"和"函数，而AND操作被称作"积"函数。

需要注意的是，为了说明执行操作的顺序，将表达式写为 $f=((\bar{x}_1)\cdot x_2)-(x_1\cdot(\bar{x}_2))$ 更为合适。为了简化表达式，我们需要对与、或、非三种操作的优先级进行定义。在没有圆括号时，逻辑表达式的操作应当按照以下顺序执行：非、与、然后是或。此外，在不产生歧义的情况下习惯上省略"·"运算符。

回到积之和的形式，我们现在解释如何直接从真值表推导出任意逻辑函数的积之和形式。考虑表A-1中的真值表，并假设函数 $f_1$ 用与门、或门、非门组成。对于表中 $f_1=1$ 的每一行，都引入积之和形式中的一个乘积（与）项。这个乘积项包括所有三个输入变量。非操作作用在单个变量上，使得只有当这些变量取与真值表的那一行对应的某个确定值时，乘积项才为1。这表明如果 $x_i=0$，则 $\bar{x}_i$ 就在乘积项中；如果 $x_i=1$，则 $x_i$ 在乘积项中。例如，表中第4行的函数值为1，而输入变量为

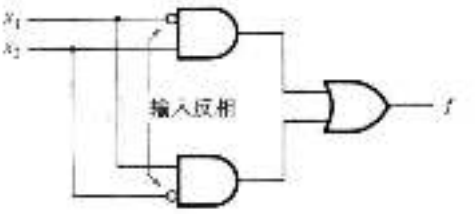
$$(x_1, x_2, x_3)=(0, 1, 1)$$

则相应的乘积项是 $\bar{x}_1x_2x_3$。将所有 $f$ 为1的行都进行同样的操作，得到

$$f_1=\bar{x}_1x_2\bar{x}_3+x_1\bar{x}_2x_3+x_1x_2\bar{x}_3+x_1x_2x_3$$

与这一表达式相对应的逻辑网络在图A-4的左方。异或函数的积之和表达式也可以使用这个方法由其真值表导出。这个方法可以从任意大小的真值表导出积之和表达式以及相应的逻辑网络。

470
|
471




$f=x_1\cdot\bar{x}_2+\bar{x}_1\cdot x_2$

a) 异或函数网络

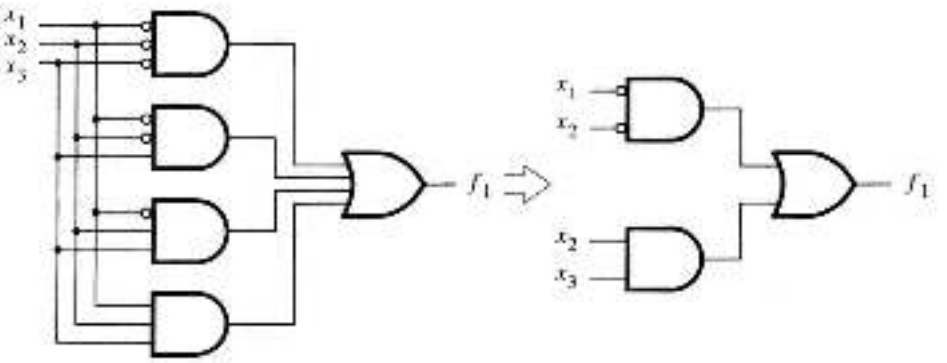
| $x_1$ | $x_2$ | $\bar{x}_1\cdot x_2$ | $x_1\cdot\bar{x}_2$ | $f=\bar{x}_1\cdot x_2+x_1\cdot\bar{x}_2$ $=x_1\oplus x_2$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |

b) $\bar{x}_1\cdot x_2+x_1\cdot\bar{x}_2$ 的真值表

图A-3 使用与门、或门和非门实现异或函数

表A-1 两个3变量函数

| $x_1$ | $x_2$ | $x_3$ | $f_1$ | $f_2$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |




$f_1=\bar{x}_1\bar{x}_2\bar{x}_3+\bar{x}_1\bar{x}_2x_3+\bar{x}_1x_2x_3+x_1x_2x_3$

$f_1=\bar{x}_1\bar{x}_2+x_2x_3$

图A-4 表A-1中 $f_1$ 的逻辑网络及与其等价的最小网络

## A.3 逻辑表达式的化简

前面介绍了如何从真值表导出一个积之和表达式。实际上，对于任意一个特定的真值表都有许多等价的表达式和逻辑网络。两个逻辑表达式或逻辑门电路如果具有相同的真值表，那么它们就是等价的。与上一节导出的$f_1$的积之和表达式等价的一个表达式是

$$\bar{x}_1\bar{x}_2 + x_2x_3$$

要证明这一点，我们需要列出这个简单表达式的真值表（表A-2），并指出它与表A-1中$f_1$函数的真值表是相同的。创建$\bar{x}_1\bar{x}_2 + x_2x_3$真值表的过程分为三步。首先，计算每种输入值下乘积项$\bar{x}_1\bar{x}_2$的值。然后，计算乘积项$x_2x_3$。最后，将这两列相“或”得到表达式的真值表。这个真值表与表A-1中的$f_1$函数真值表完全一致。

表A-2 表达式$\bar{x}_1\bar{x}_2 + x_2x_3$的真值表

| $x_1$ | $x_2$ | $x_3$ | $\bar{x}_1\bar{x}_2$ | $x_2x_3$ | $\bar{x}_1\bar{x}_2 + x_2x_3 = f_1$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 |

为了简化逻辑表达式，我们进行一系列的代数操作。在这些操作中，我们使用了新的逻辑规则：分配律

$$w(y + z) = wy + wz$$

和恒等式

$$w + \bar{w} = 1$$

表A-3给出了分配律的真值表证明。现在应该清楚，像这样的规则总是可以通过列出其左边和右边的真值表，然后指出它们相同的方式来证明。逻辑规则如分配律有时称为恒等式。尽管在 472
这里用不到，但为了完整性，再给出分配律的另一种形式：

$$w + yz = (w + y)(w + z)$$

表A-3 真值表方法证明表达式等价

| $w$ | $y$ | $z$ | $y + z$ | 等号左边 $w(y + z)$ | $wy$ | $wz$ | 等号右边 $wy + wz$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

逻辑化简的目的是根据一些标准减少实现特定逻辑函数的成本。特别地，我们希望从由真值表导出的积之和表达式开始，将它化简为最小积之和（minimal sum-of-products）表达式。为了定义化简的标准，有必要引入一个衡量积之和表达式的大小或成本的量度。通常成本量度是以图A-4的形式实现表达式所需的门和门输入的总数。例如，图中较大的表达式成本为21，由5个门和16个门输入组成。在计数过程中忽略了输入的反相。较小的表达式成本为9，由3个门和6个门输入组成。现在可以明确，如果没有其他等价的积之和表达式具有更小的成本， 473
那么这个积之和表达式就是最小的。下面所举的简单例子可以清楚地看出我们得到了最小的表

达式。因此不再给出最小化的严格证明。

简化一个给定表达式的一般算术操作如下。首先，将在一个乘积项中为反（$x$），而在另一个乘积项中为真（$x$），并且其他变量相同的项两两分组。根据分配率，提出由其他变量组成的公共乘积子项后，则剩下 $x + \bar{x}$ 项，其值为 1。对第一个 $f_1$ 的表达式应用这一操作，我们得到：

$$\begin{aligned} f_1 &= x_1x_2x_3 + x_1x_2\bar{x}_3 + x_1\bar{x}_2x_3 + \bar{x}_1x_2x_3 \\ &= x_1\bar{x}_2(\bar{x}_3 + x_3) - (\bar{x}_1 + x_1)x_2x_3 \\ &= x_1\bar{x}_2 \cdot 1 + 1 \cdot x_2x_3 \\ &= x_1\bar{x}_2 + x_2x_3 \end{aligned}$$

这个表达式是最小的。相应的网络连接如图 A-4 所示。

将乘积项成对分组得到最简表达式的最小化过程并不总是像前面例子那样明显。一个很有用的规则是

$$w + w = w$$

这个规则允许我们重复使用乘积项，从而在提取公因式的过程中，某个乘积项可与其他多个项组合。举个例子，看一下表 A-1 中的 $f_2$ 函数。其积之和表达式可直接由真值表导出为

$$f_2 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3$$

通过重写第一个乘积项 $x_1x_2x_3$ 并交换项的位置（根据交换律），我们得到

$$f_2 = x_1x_2x_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3 + x_1\bar{x}_2x_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2x_3$$

将乘积项结对并提取公因子，得到

$$\begin{aligned} f_2 &= x_1x_2(x_3 + \bar{x}_3) + x_1\bar{x}_2(x_3 + x_3) + x_3(\bar{x}_2 + x_2)x_3 \\ &= x_1x_2 + x_1\bar{x}_2 + \bar{x}_1\bar{x}_2 \end{aligned}$$

现在，通过因式分解化去前两项，得到最小表达式

$$f_2 = x_2 + \bar{x}_1x_3$$

这样就完成了我们对逻辑表达式的算术化简的讨论。这项数学练习的实际意义是显而易见的。由较少的门和输入构成的网络更加廉价并且易于实现。因此，确定与所给表达式等价的最简表达式符合经济利益的需要。表 A-4 中总结了操作逻辑表达式时应用的规则。它们成对出现，显示出“与”和“或”函数的对称性。到目前为止，我们还没有机会使用对合律或德摩根律，但是在下一节中会发现它们很有用。

表 A-4 二进制逻辑规则

| 名 称 | 算术等式 | |
|---|---|---|
| 交换律 | $w \cdot y = y \cdot w$ | $wy = yw$ |
| 结合律 | $(w + y) + z = w + (y + z)$ | $(wy)z = w(yz)$ |
| 分配律 | $w + yz = (w + y)(w + z)$ | $w(y + z) = wy + wz$ |
| 等幂律 | $w \cdot w = w$ | $ww = w$ |
| 对合律 | $\bar{\bar{w}} = w$ | |
| 互补律 | $w + \bar{w} = 1$ | $w\bar{w} = 0$ |
| 德摩根律 | $\overline{w + y} = \bar{w}\,\bar{y}$ | $\overline{wy} = \bar{w} + \bar{y}$ |
| | $1 + w = 1$ | $0 \cdot w = 0$ |
| | $0 + w = w$ | $1 \cdot w = w$ |

474

## A.3.1 使用卡诺图化简

在对表 A-1 中的函数 $f_1$ 和 $f_2$ 进行代数化简的过程中，有些时候必须猜测最佳的处理方法。例如，在 $f_2$ 的化简中判断第一步重写项 $x_1\bar{x}_2x_3$ 并不明显。有一种几何方法可以迅速地导出含有几个变量的逻辑函数的最小表达式，这依赖于真值表的另一种表示形式，称为卡诺图（Karnaugh map）。对于一个 3 变量函数，它是一个由 2 行 4 列共 8 个方块组成的长方形，如图 A-5a 所示。每个方块对应于输入变量的一组特定的值。例如，第一行的第三个方块表示值 $(x_1, x_2, x_3) = (1, 1, 0)$。因为 3 变量的真值表为 8 行，所以卡诺图显然需要 8 个方块，方块中的内容是与输入变量值相对应的函数值。

构建卡诺图的关键思想是水平和垂直相邻的方块对应着只有一个变量不同的输入变量值。当两个相邻的方块值都为1时，表明可以进行代数化简。在图A-5a函数 $f_2$ 的卡诺图中，第一行最左边的两个值为1的方块与乘积项 $x_1x_2x_3$、$x_1x_2x_3$ 对应。在最小化 $f_2$ 代数表达式过程中先执行化简

$$\bar{x}_1\bar{x}_2\bar{x}_3+\bar{x}_1x_2\bar{x}_3=\bar{x}_1\bar{x}_3$$

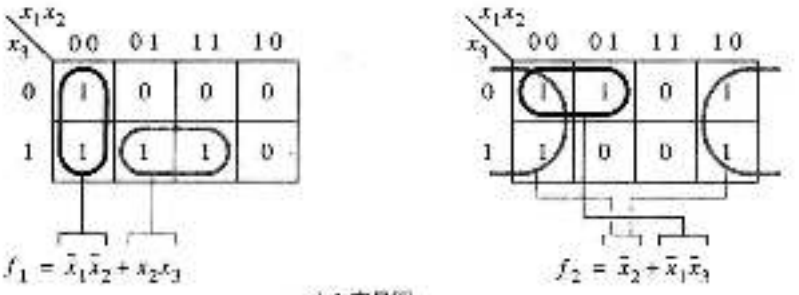
将图中的两个1格圈为一组就可以直接得到化简结果。与一组方块相对应的乘积项是在这些方块中值不变的那些输入变量的乘积。如果 $x_i$ 在一组1格中的值为0，则将 $\bar{x}_i$ 加入乘积项，如果 $x_i$ 的值是1，则将 $x_i$ 加入乘积项。两方块相邻的情况还包括最左端的方块与最右端的方块相邻。继续我们对 $f_2$ 的讨论，由最左列和最右列组成的4个1格的组化简得到单一变量项 $\bar{x}_2$，因为 $x_2$ 是该组中唯一一个值不变的变量。其他两个变量的所有4种可能的组合都在该组中出现过了。 475

卡诺图可以用于多于3个变量的情况。4变量的卡诺图可以由两个3变量的卡诺图得到。图A-5b是4变量卡诺图的例子，同时还给出了图中表示的函数的最小表达式。除了2个方块和4个方块的组外，现在还可以组建8个方块的组。这样的分组在 $g_2$ 的图中有所显示。注意 $g_2$ 中角上的4个方块也构成了一个合法的4格组，代表乘积项 $x_2x_4$。与3变量的图一样，与一组方块所对应的项是组中没有改变值的变量乘积。例如 $g_2$ 图中右上角的4格组表示为乘积项 $x_1x_3$，因为组中 $x_1=1$，$x_3=0$。该组中包含了变量 $x_2$ 与 $x_4$ 所有可能的取值组合。对于5变量函数也可以使用卡诺图。在这种情况下，将使用两个4变量卡诺图，其中一个对应于第5个变量取0的情况，另一个对应取1的情况。

在卡诺图中划分2格组、4格组、8格组等的一般方法可以很容易地得出。两个相邻的值为1的对可以组合为一个4格组。类似地，两个相邻的4格组可以组成一个8格组。通常，任意有效组中方格的个数一定为 $2^k$，其中 $k$ 为整数。

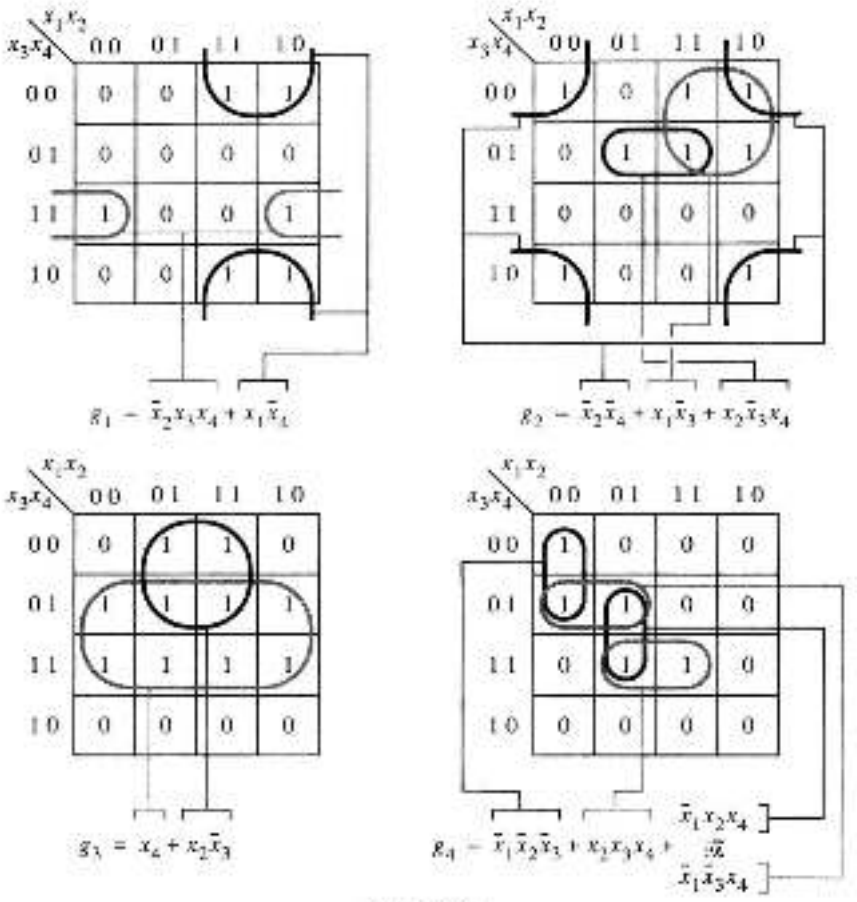
我们现在考虑使用卡诺图得到最小积之和表达式的过程。从图A-5中可以看出，大的组对应于小的乘积项。因此，要得到简单门电路，应该尽量用最少的组覆盖图中所有的1。一般说来，我们应当使用最少的组，并且每个组都尽可能的大，来覆盖所有的1。例如，考虑图A-5b中的函数 $g_2$。如图所示，四个角上值为1的方块组成了表示乘积项 $x_2x_4$ 的组。另一个4格组在右上角，表示乘积项 $x_1\bar{x}_3$。这两个组包括了除位置为 $(x_1, x_2, x_3, x_4)=(0, 1, 0, 1)$ 外所有值为1的方块。包含这个方块的最大的组是一个2格组，表示乘积项 $x_2x_3x_4$。因此，$g_2$ 的最小表达式是

$$g_2=\bar{x}_2\bar{x}_4+x_1\bar{x}_3+x_2\bar{x}_3x_4$$




a) 3变量图

图A-5 使用卡诺图进行化简




b）4变量图

图A-5 （续）

图中其他函数的最小表达式可使用类似方法得到。注意 $g_4$ 有两种可能的最小表达式，一个包括 $x_1x_3x_4$ 项，另一个包括 $x_1\bar{x}_3x_4$ 项。这种给定的函数有不止一个最小表达式的情况经常出现。

在我们给出的所有例子中，得到最小表达式都比较容易。通常这一过程有正式的算法，但这里不再讨论。

### A.3.2 无关项条件

476
|
477

在许多情况下，数字电路的一些输入值永远都不会出现。例如，考虑二进制编码的十进制数（BCD）表示法。四个二进制变量 $b_3$、$b_2$、$b_1$ 和 $b_0$ 表示十进制数0到9，如图A-6所示。这4个变量总共有16个不同的值，其中只有10个用于表示十进制数，剩下的值没有用到。因此，任何处理BCD码的逻辑电路都永远不会在其输入中碰到这6个值中的任何一个。

图A-6给出了一个操作BCD数的特定函数的真值表。我们并不关心那些从不使用的输入值的函数值是多少；因此，它们被称为无关项（don't-care），在真值表中用字母"d"表示。在实现电路时，对应于无关项的函数值可被任意赋值为0或1。最佳的赋值方法是使其能够用最小的逻辑门实现的方法。当无关项能够扩大一个1值的组时，我们就把它赋值为1。因为较大的组对应着较小的乘积项，适当地使用无关项能够更大限度地将电路最小化。

图 A-6 中的函数表示了如下对输入十进制数的处理过程：当输入是一个可被 3 整除的非零数时，输出 $f$ 为 1。我们需要 3 个组来包含图中的 3 个值为 1 的方块，并且尽可能使用无关项来扩大组。

| 十进制数表示 | 二进制码 | $f$ |
|---|---|---|
| 0 | 0 0 0 0 | 0 |
| 1 | 0 0 0 1 | 0 |
| 2 | 0 0 1 0 | 0 |
| 3 | 0 0 1 1 | 1 |
| 4 | 0 1 0 0 | 0 |
| 5 | 0 1 0 1 | 0 |
| 6 | 0 1 1 0 | 1 |
| 7 | 0 1 1 1 | 0 |
| 8 | 1 0 0 0 | 0 |
| 9 | 1 0 0 1 | 1 |
| 未使用 | 1 0 1 0 | d |
| | 1 0 1 1 | d |
| | 1 1 0 0 | d |
| | 1 1 0 1 | d |
| | 1 1 1 0 | d |
| | 1 1 1 1 | d |

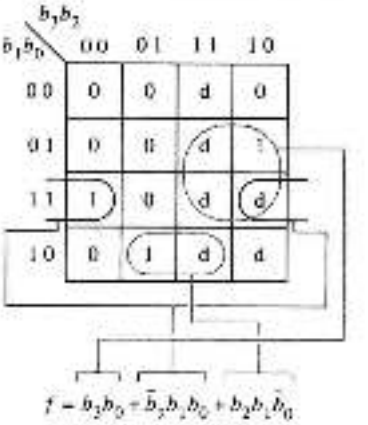



图 A-6 带有无关项的 4 变量卡诺图

## A.4 与非门、或非门的组合

现在来看另外两个被称为与非门（NAND）和或非门（NOR）的基本逻辑门，由于它们的电路实现很简单，因此在实践中被广泛使用。这些门的真值表如图 A-7 所示。它们等价地实现了“与”和“或”函数后接一个“非”函数，这便是它们的名称以及标准逻辑符号的由来。我们用箭头“↑”和“↓”代表与非及或非运算符，使用表 A-4 中的德摩根律，可以得到

$$x_1 \uparrow x_2 = x_1x_2 = \bar{x}_1 + \bar{x}_2$$

和

$$x_1 \downarrow x_2 = \overline{x_1 + x_2} = \bar{x}_1\bar{x}_2$$

存在多于两个输入的与非门和或非门，根据德摩根律的简单扩展可以得到

$$x_1 \uparrow x_2 \uparrow \cdots \uparrow x_n = x_1x_2\cdots x_n = \bar{x}_1 + \bar{x}_2 + \cdots + \bar{x}_n$$

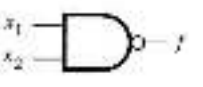
和

$$x_1 \downarrow x_2 \downarrow \cdots \downarrow x_n = \overline{x_1 + x_2 + \cdots + x_n} = \bar{x}_1\bar{x}_2\cdots\bar{x}_n$$

使用与非门和或非门进行逻辑设计不像使用与、或、非门那样直截了当。设计过程的主要难点之一是结合律对与非和或非操作是无效的。我们稍后再对这一问题展开讨论。先来讨论只使用与非门实现任意逻辑函数的简单过程。将一个积之和形式的逻辑网络变换成只由与非门组成的网络有一个直接的方法。借助下面的例子我们可以很容易地了解这个过程。考虑下面由三个 2 输入与非门组成的 4 输入网络的逻辑表达式的代数运算过程。

| $x_1$ | $x_2$ | $f$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$f = x_1 \uparrow x_2 = \overline{x_1x_2} = \bar{x}_1 + \bar{x}_2$




a) 与非门

| $x_1$ | $x_2$ | $f$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

$f = x_1 \downarrow x_2 = \overline{x_1 + x_2} = \bar{x}_1\bar{x}_2$




b) 或非门

图 A-7 与非门和或非门

$$
\begin{aligned}
(x_1 \uparrow x_2) \uparrow (x_3 \uparrow x_4) &= \overline{(\overline{x_1 x_2})(\overline{x_3 x_4})} \\
&= \overline{x_1 x_2} + \overline{x_3 x_4} \\
&= x_1 x_2 + x_3 x_4
\end{aligned}
$$

在推导过程中我们使用了德摩根律和对合律。图 A-8 给出了与这个推导对应的逻辑网络。因为任何逻辑函数都可以表示为“积之和”（与–或）形式，而且前述的导出过程完全可逆。我们得出结论，任何逻辑函数都可以表示为“与非–与非”（NAND-NAND）形式。可以看到这一结论对含有任意数量变量的函数都是正确的。与非门的输入数显然与相应的与门和或门的输入数相同。

478 / 480

我们回到关于结合律对与非操作符不适用这一话题上来。按照图 A-8 中的步骤使用与非门对逻辑网络进行设计时，可能需要具有更多输入数的与非门（比实际中能够实现的与非门输入数还要多）。这是因为与和或操作符是可结合的，可以将与门和或门直接级联起来。使用 2 输入门实现 3 输入与函数和或函数的情况如图 A-9a 所示。使用与非门的解决方案就不这么简单了。例如，一个 3 输入与非函数不能用两个 2 输入与非门级联实现，而需要三个门，如图 A-9b 所示。

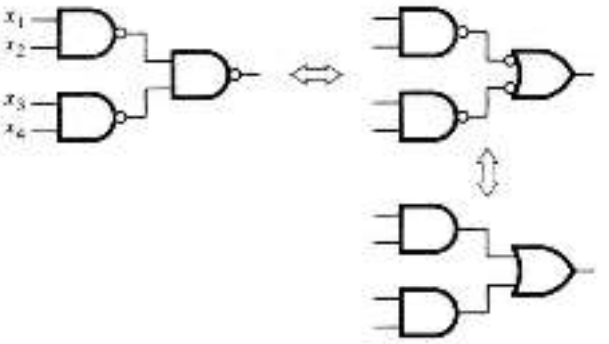
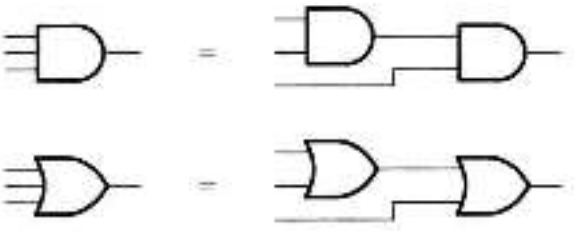
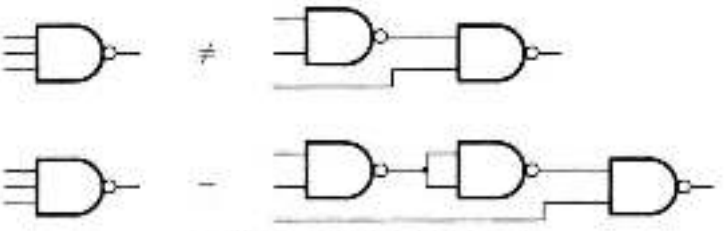



图 A-8 等价的与非–与非和与–或网络



a）用 2 输入门实现 3 输入与函数和或函数



b）用 2 输入门实现 3 输入与非函数

图 A-9 门的级联

对只使用或非门实现逻辑函数的讨论与前面类似。任意逻辑函数都可以表示为“和之积”（或–与）的形式。这样的网络可以由等价的或非–或非网络实现。

前面的讨论介绍了一些逻辑设计的基本概念。这一主题的详细讨论可在许多教科书中找到（参见参考文献 [1 ~ 7]）。

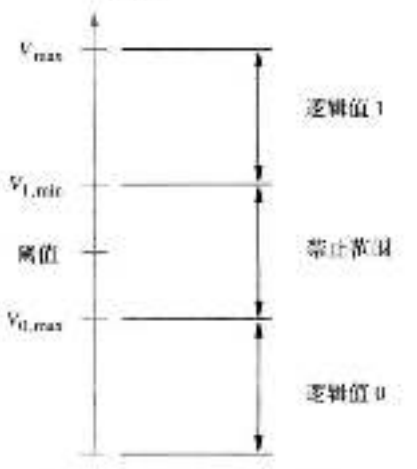
需要明确指出的是，一个给定的逻辑函数可能有许多不同的实现。由于实际应用的需要，我们要找到成本最低的实现方法，还经常需要考虑减少逻辑网络中的传输延迟。为了描述逻辑组合的性质和尽可能地降低成本，前几节中介绍了最小化的概念。例如，卡诺图图形化地表明了得到最佳结果的操作。尽管理解逻辑网络的最优化法则很重要，但并不需要人工实现最优化。复杂的计算机辅助设计（Computer-Aided Design，CAD）程序可以完成逻辑集成和最优化。设计者只需要指定所需的功能行为，CAD软件就会给出一个实现这一功能的经济且高效的网络。 [481]

## A.5 逻辑门的实现

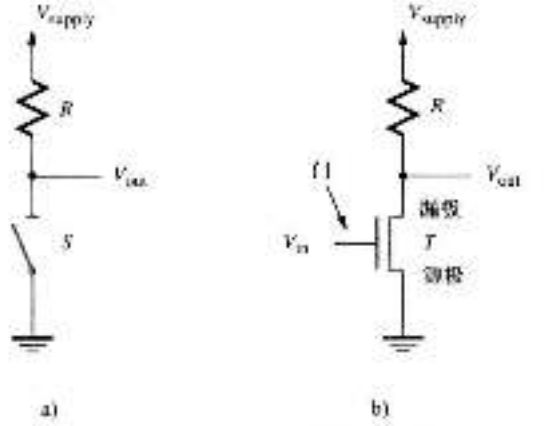
现在让我们关注在实践中表示逻辑变量和实现逻辑函数的方法。表示逻辑变量的物理参数的选择明显依赖于具体的技术。在电路里，电平或电流值都可以用于此目的。

为了建立电平和逻辑值或状态的对应关系，我们引入了阈值（threshold）这一概念。高于给定阈值的电平表示一个逻辑值，而低于阈值的电平表示另一个逻辑值。在实际情况下，由于各种各样的原因，电路中任何一点的电平都有小的随机变化。因为存在“噪声”，我们不能确定阈值附近电平的逻辑状态。如图A-10所示，为了避免出现这样的不确定性，应该确定一个“禁止范围”。这样，低于 $V_{0,max}$ 的电平表示0值，高于 $V_{1,min}$ 的电平表示1值。在下面的讨论中，将经常用到“低”和“高”这两个术语来分别表示与逻辑值0和1对应的电平。

我们对实现基本逻辑函数的电子线路的讨论将从简单的由电阻和晶体管开关组成的电路开始。考虑图A-11中的电路。当图A-11a中的开关 $S$ 闭合时，输出电平 $V_{out}$ 等于0（接地）。当 $S$ 打开时，输出电平 $V_{out}$ 等于提供的电平 $V_{supply}$。图A-11b用晶体管 $T$ 代替了开关 $S$，也可以得到同样的效果。当提供给晶体管的门输入电平为0时（即 $V_{in}=0$），晶体管相当于一个打开的开关，且 $V_{out}=V_{supply}$。当 $V_{in}$ 变化为 $V_{supply}$ 后，晶体管相当于一个闭合的开关，输出电平 $V_{out}$ 非常接近0。因此，该电路实现了一个逻辑非门的功能。




图A-10 用电平表示逻辑值




图A-11 一个反相器电路

现在可以讨论更为复杂的逻辑函数的实现。图A-12显示了或非门的实现电路。在这种情况下，只有当开关 $S_a$ 和 $S_b$ 都打开时，图A-12中的 $V_{out}$ 才是高电平。与之类似，图A-12b中只有输入电平 $V_a$ 和 $V_b$ 都是低电平时，$V_{out}$ 才为高电平。因此，该电路对应一个或非门，其中 $V_a$

和 $V_b$ 对应于两个输入变量 $x_1$、$x_2$。容易验证，将晶体管如图 A-13 那样串联，就可以得到一个与非门电路。逻辑函数“与”和“或”可以分别使用与非门和或非门，后接图 A-11 中的反相器实现。

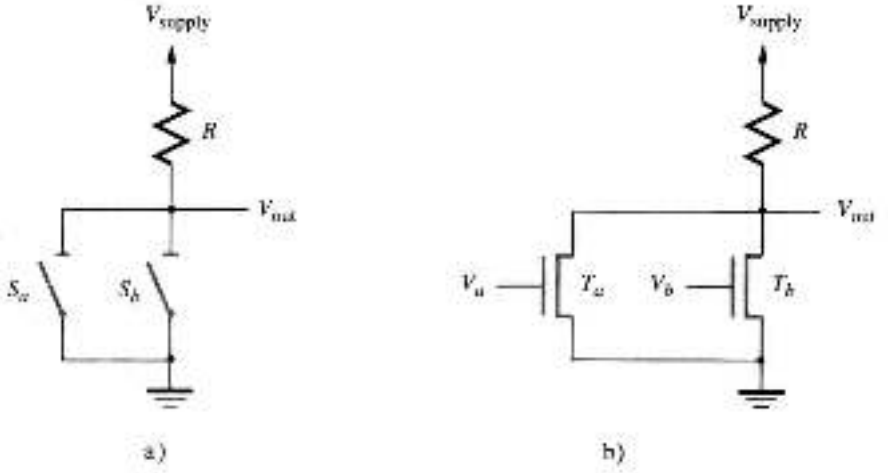



图 A-12 或非门的晶体管电路实现

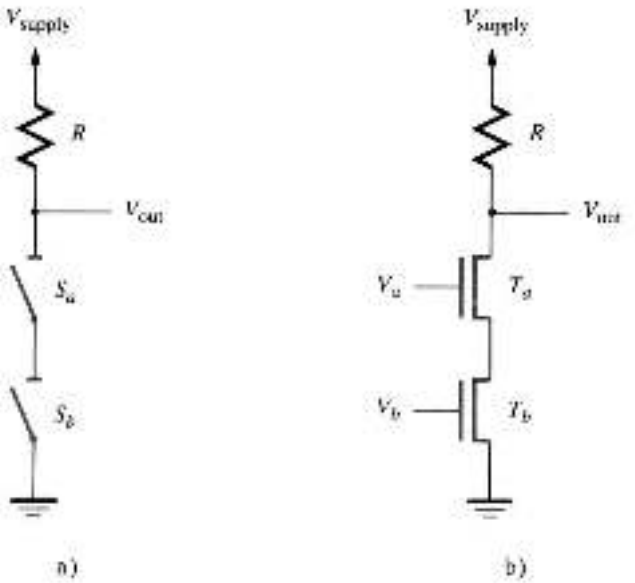



图 A-13 与非门的晶体管电路实现

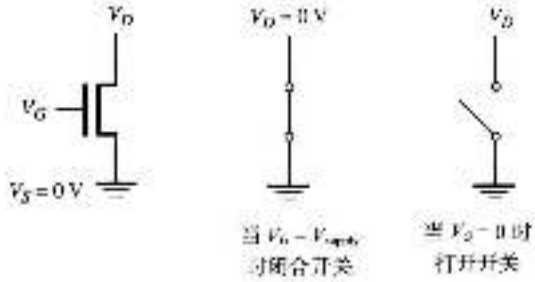
注意，与非门和或非门的电路实现比与门和或门要简单。因此，在实践中经常大量使用与非门和或非门实现逻辑函数。本书中所举的许多由与、或、非门组成电路的例子是为了便于理解。在实际应用中，逻辑电路包含所有这 5 种门电路。

### A.5.1 CMOS 电路

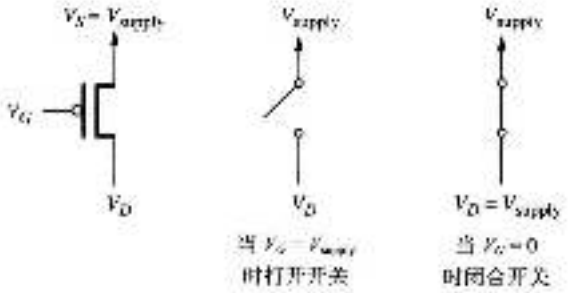
图 A-11 到图 A-13 示例了使用 NMOS 技术（NMOS technology）实现电路的一般结构。这个名称起源于实现逻辑函数的晶体管是 NMOS 型。两种类型的金属氧化物半导体（Metal-

Oxide Semiconductor，MOS）可以作为开关使用。我们称n沟道晶体管为NMOS类型，当它的门输入上升到正电源电平 $V_{supply}$ 时，表现为一个闭合的开关，如图A-14a所示。与其相反的是p沟道晶体管，即PMOS类型。当门电平 $V_G$ 等于 $V_{supply}$ 时，它相当于一个打开的开关，而当门电平 $V_G = 0$ 时，它相当于一个闭合的开关，如图A-14b所示。注意PMOS晶体管的图形表示在其门输入处有一个圆圈，表示它的行为与NMOS晶体管相反。还要注意PMOS晶体管的源极和漏极名称也与NMOS晶体管所接关系相反。NMOS晶体管的源极是接地的，而PMOS晶体管的源极与 $V_{supply}$ 相连。这些命名习惯是根据晶体管中的电流性质定义的。

482
/
484




a）NMOS晶体管



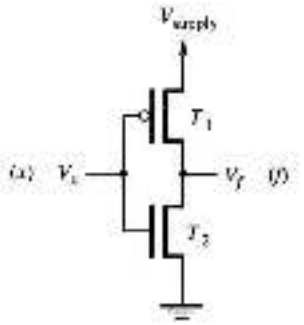

b）PMOS晶体管

图A-14 逻辑电路中的NMOS和PMOS晶体管

图A-11到图A-13电路的缺点在于它们的能量损耗。当开关闭合时提供了地与上拉电阻 $R$ 之间的通道，电流从 $V_{supply}$ 流向地。相反，当开关打开时，没有到地的通路也没有电流流过（MOS晶体管的门端没有电流流过）。因此，根据门的状态，在逻辑电路中可能会出现明显的能量损耗。

解决能量损耗问题的有效方法是同时使用NMOS和PMOS晶体管来实现电路，使之在稳定状态下不消耗能量。这种方法引出了CMOS（互补金属氧化物半导体）技术。图A-15的反相器电路示例了CMOS电路的基本思想。当 $V_x = V_{supply}$，相当于输入 $x$ 的逻辑值为1，晶体管 $T_1$ 关闭而 $T_2$ 打开。因此 $T_2$ 将输出电平 $V_f$ 下拉到0。当 $V_x$ 变为0时，晶体管 $T_1$ 打开而 $T_2$ 关闭。这样，$T_1$ 将输出电平 $V_f$ 上拉至 $V_{supply}$。因此，$x$ 和 $f$ 的逻辑值互补，电路实现了一个非门。 485

这个电路的关键特征在于晶体管 $T_1$ 和 $T_2$ 操作是完全相反的：当一个是打开时，另一个就是闭合的。因此，从输出点 $f$ 到 $V_{supply}$ 或地总有一条闭合的通路。但是，除了晶体管转换状态时非常短的过渡期外，任何时候在 $V_{supply}$ 和地之间都没有闭合通路。这表明当电路处于稳定状态时并不消耗多少能量，仅当它从一个逻辑状态转变到另一个逻辑状态时消耗能量。因此，这个电路的能量损耗是由状态转换的频率决定的。




a) 电路

| $x$ | $V_x$ | $T_1$ $T_2$ | $V_f$ | $f$ |
|---|---|---|---|---|
| 0 | 低 | 开 关 | 高 | 1 |
| 1 | 高 | 关 开 | 低 | 0 |

b) 真值表与晶体管状态

图 A-15 非门的 CMOS 实现

现在我们可以将 CMOS 的概念扩展到 $n$ 输入电路，如图 A-16 所示。用 NMOS 晶体管实现下拉网络，在函数 $F(x_1, \cdots, x_n)$ 等于 0 时建立输出点 $f$ 和地之间的闭合通路。上拉网络由 PMOS 晶体管构成，在函数 $F(x_1, \cdots, x_n)$ 等于 1 时建立输出点 $f$ 和 $V_{supply}$ 之间的闭合通路。上拉和下拉网络的功能是相反的，因此在稳定状态，输出点 $f$ 和 $V_{supply}$ 或地间只存在一条闭合通路，而不会两者同时存在。

下拉网络的实现方式与图 A-11 至图 A-13 相同。图 A-17 给出了与非门的实现，图 A-18 给出了或非门的实现。图 A-19 通过将与非门的输出反向实现了与门。

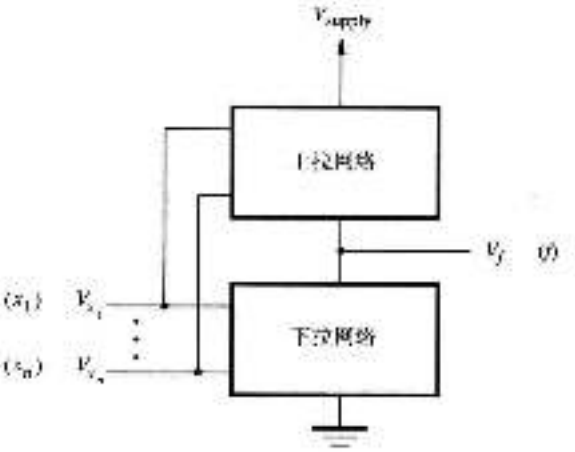
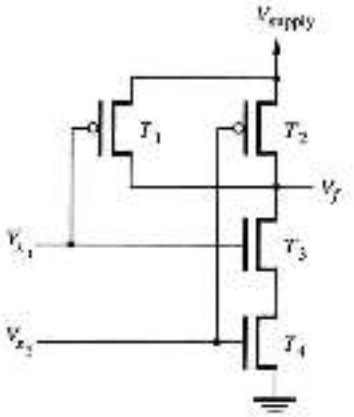



图 A-16 CMOS 电路结构




a) 电路

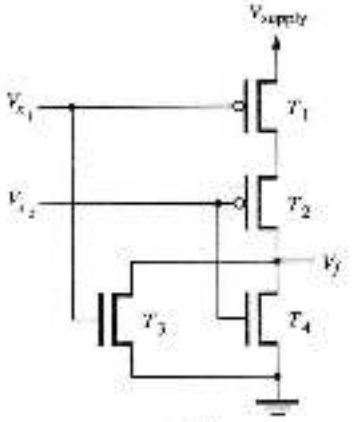
| $x_1$ $x_2$ | $T_1$ $T_2$ $T_3$ $T_4$ | $f$ |
|---|---|---|
| 0 0 | 开 开 关 关 | 1 |
| 0 1 | 开 关 关 开 | 1 |
| 1 0 | 关 开 开 关 | 1 |
| 1 1 | 关 关 开 开 | 0 |

b) 真值表和晶体管状态

图 A-17 与非门的 CMOS 实现

除了低能量损耗，CMOS 电路的优点还有 MOS 晶体管体积很小，因此在集成电路芯片上只占用很小的地方。这个特性有两个显著的优势。首先，它使得在芯片上集成亿万个晶体管成

为可能，从而能够实现现代的处理器和大型的存储芯片。其次，晶体管的体积越小，它从一个状态到另一个状态的转换就越快。因此，CMOS电路运行速度可达GHz级。



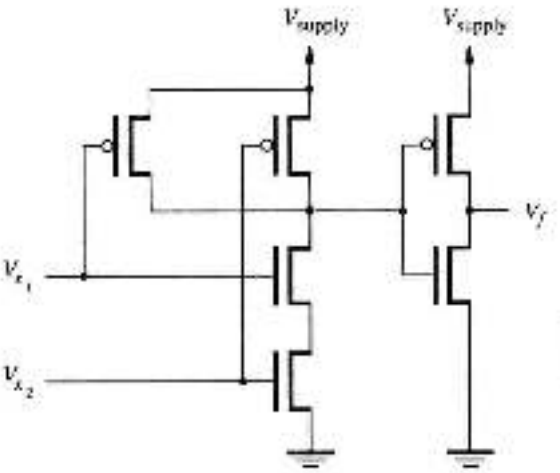

a）电路

| $x_1$ | $x_2$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $f$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 开 | 开 | 关 | 关 | 1 |
| 0 | 1 | 开 | 关 | 关 | 开 | 0 |
| 1 | 0 | 关 | 开 | 开 | 关 | 0 |
| 1 | 1 | 关 | 关 | 开 | 开 | 0 |

b）真值表和晶体管状态

图A-18 或非门的CMOS实现

不同CMOS电路的运行电平不同，最高可达到15V。最常用的电平值范围是1～5V。使用低电平的电路消耗的能量更少（能量消耗约与$V^2_{supply}$成正比），这意味着可以在一块芯片上放置更多的晶体管而不会过热。低电平的缺点是减小了噪声屏蔽。




图A-19 与门的CMOS实现

CMOS反相器中高低电平信号间传输的细节如图A-20所示。粗的曲线为传输特性曲线（transfer characteristic），显示了作为输入电平函数的输出电平的变化。它表明当输入电平经过$V_{supply}/2$附近时，输出电平会有一个很陡峭的变化。在此引入一个阈值电平$V_t$和一个小的值$\delta$，如果$V_{in} < V_t - \delta$，则$V_{out} \approx V_{supply}$，如果$V_{in} > V_t - \delta$，则$V_{out} \approx 0$。这说明输出正确的信号时，输入信号不一定严格等于限定电平值0或$V_{supply}$。输入信号中可能会有一些可以容忍的错误，即噪声（noise），它们不会引起不利的效果。可以容忍的噪声量称为噪声容限（noise margin）。当输入的逻辑值为1时，这一容限是$V_{supply} - (V_t + \delta)$，而当输入逻辑值是0时，这一容限是$V_t - \delta$。CMOS电路具有出色的噪声容限。

在本节中我们介绍了CMOS电路的基本特征，读者如想了解此技术的更多细节，可以查阅参考文献[1]和[8]。

## A.5.2 传播延迟

逻辑电路不能够立即从一个状态转到另一个状态。速度由状态变化的频率来衡量。一个相关的参数是传播延迟（propagation delay），由图A-21定义。当输入的状态变化后，在输出做出相应变化之前有一个延迟。如图所示，通常传播延迟由输入和输出的状态转换发生50%

486
↓
488

时刻之间的时间长度衡量。另一个重要的参数是转换时间（transition time），通常由信号幅度10%变化至90%之间的时间长度衡量，如图所示。逻辑电路运行的最大速度随着电路通过不同路径传播延迟的增加而减少。逻辑电路中任一通路的延迟是此通路中单个门延迟的总和。

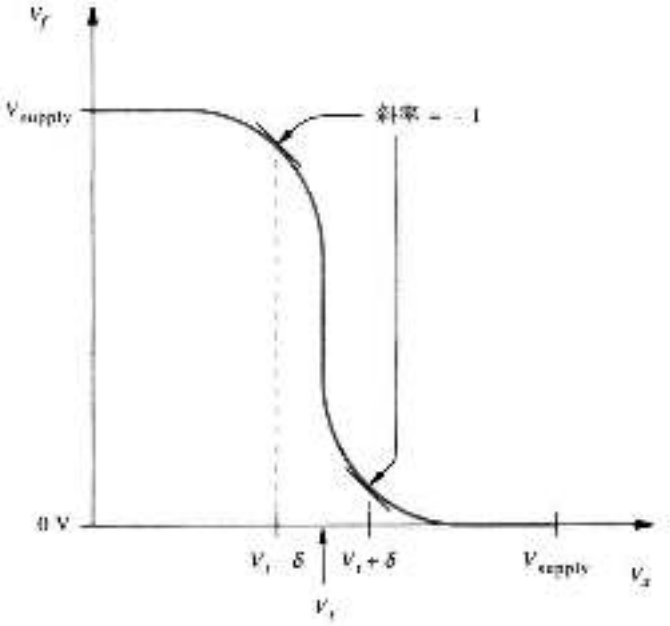



图 A-20 CMOS 反相器的电平传输特性

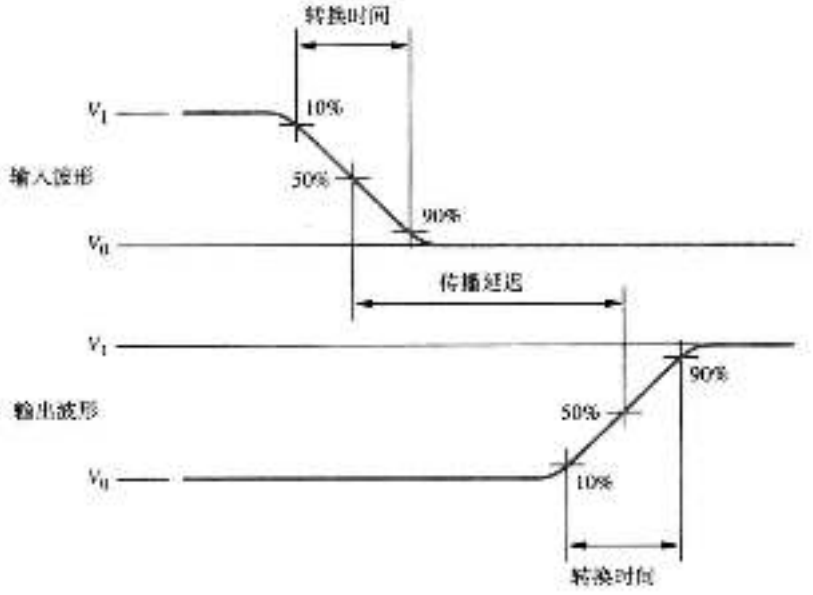



图 A-21 传播延迟和转换时间的定义

## A.5.3 扇入扇出限制

逻辑门的输入数称为扇入（fan-in），逻辑门输出驱动的门输入数称为扇出（fan-out）。实

际电路不允许过大的扇入和扇出，因为这会对传播延迟和电路速度产生不利的影响。

CMOS门中的每个晶体管都有一定的电容。当电容增加时，电路速度会变慢，而且信号电平和噪声容限也会变差。因此，需要对扇入和扇出进行限制，通常小于10。如果需要的输入数超过了最大的扇入值，必须再使用一个同类型的门。图A-9a显示了两个同类型的门如何级联。如果必须要某个门驱动的输出端数超过扇出数，可以使用两个同类型的门。

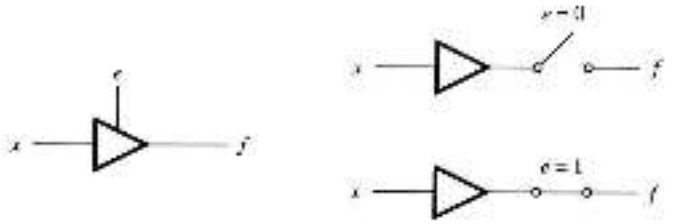
489 ; 490

### A.5.4 三态缓冲器

在目前讨论的逻辑门中，不能将两个门的输出连接在一起。因为当一个门的输出值是1而另一个是0时，我们不能确定组合的输出信号是什么值。这从逻辑角度讲是没有意义的。更重要的是，在CMOS电路中，输出为1的门建立了一条从输出端到$V_{supply}$的直接通路，而输出为0的门建立了到地的通路。因此，这两个门会形成对电源的短路，从而对门造成损坏。

但是，在计算机系统设计时，许多时候会出现电路的输入信号从许多不同的源中获得的情况。这时可以使用多路复用逻辑电路，这个内容将在A.10节中讨论。也可以使用称为三态缓冲器（tri-state buffer）的特殊门来实现。三态缓冲器具有三个状态。其中的两个状态产生普通的0和1信号。第三个状态将缓冲器的输出端置于高阻抗状态，使输出在电气上与其所驱动的输入断开。

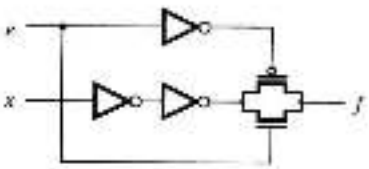
图A-22描述了一个三态缓冲器。这个缓冲器有两个输入和一个输出。使能输入$e$控制缓冲器的操作。当$e=1$时，输出$f$与输入$x$具有相同的逻辑值。当$e=0$时，输出被置于高阻抗状态Z。与其等效的电路如图A-22b所示。图中的三角形符号代表一个非反相驱动器。这个电路没有实现任何逻辑操作，因为它的输出仅仅复制了输入信号，目的是提供额外的电气驱动能力。在与图中的输出开关组合后，它的行为根据图A-22c的真值表动作。这个表描述了所需的三态行为。图A-22d给出了三态缓冲器的电路实现。将一个NMOS晶体管和一个PMOS晶体管并联起来实现一个开关，与驱动的输出连接。因为这两种晶体管的门输入端需要相反的控制信号，所以要使用一个反相器。当$e=0$时，两个晶体管都关闭，相当于一个打开的开关；当$e=1$时，两个晶体管都打开，相当于一个闭合的开关。




a）符号　　b）等效电路

| e | x | f |
|---|---|---|
| 0 | 0 | Z |
| 0 | 1 | Z |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

c）真值表




d）实现

图A-22　三态缓冲器

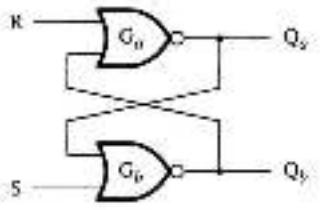
驱动电路可能需要驱动大量其他的门输入，这些门的总容量超过了普通的逻辑门电路的驱动能力。为了提供足够的驱动能力，驱动电路需要更大的晶体管。因此，实现驱动器的两个级联非门所使用的晶体管要比常规逻辑门中的大。

读者可能会疑惑为什么在输出开关中必须使用PMOS晶体管，因为从逻辑函数的角度来看，只使用NMOS晶体管也可以取得相同的效果。这样做的原因是这些晶体管必须将驱动电路产生的逻辑值"传送"到输出$f$，而结果表明NMOS晶体管能很好地传送0值，但传送1值的效果很差，而PMOS晶体管正好相反。因此NMOS和PMOS的并联能够很好地传递0和1值。关于这一问题和三态缓冲器的更详细讨论，读者可以参阅参考文献[1]。

## A.6 触发器

大部分数字逻辑的应用都需要信息的存储。例如，控制密码锁的电路必须记住所拨数字的顺序，以确定是否要打开锁。另一个重要的例子是数字计算机中的内存保存程序和数据。

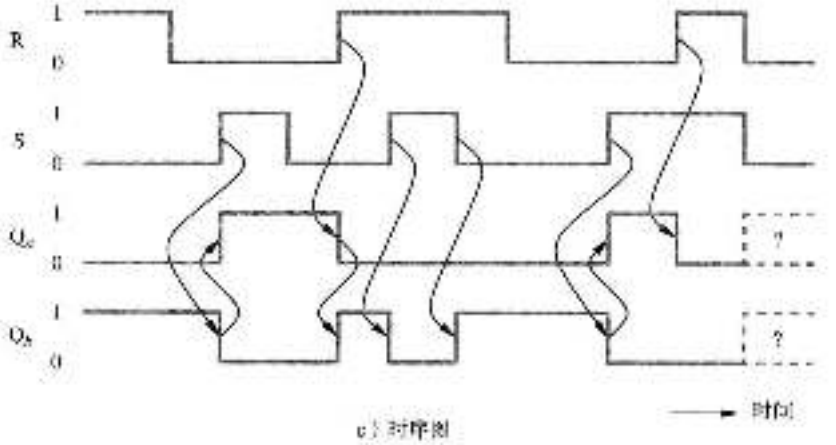
存储二进制信息的基本电子元件称为锁存器（latch）。考虑图A-23a中两个交叉耦合的或非门组成的电路。假设起始状态为R = 1，S = 0。简单的分析表明$Q_a = 0$，$Q_b = 1$。在这种情况下，门$G_b$的两个输入均为1，这样，如果R变为0，$Q_a$和$Q_b$的输出不会有任何变化。如果S设置为1而R等于0，$Q_a$和$Q_b$将分别是1和0，而且当S变回0时仍会保持这一状态。因此，这个逻辑电路构成了一个存储单元或锁存器，它记下了两个输入S和R中哪一个最近等于1。图A-23b给出了这个锁存器的真值表。图A-23c显示了锁存器的特征波形。其中的箭头表明了信号间的因果关系。注意当输入R和S同时从1变到0时，结果状态是不确定的。实际上，这时锁存器将随机假设为两个稳态中的一个。输入值R = S = 1在大多数锁存器中不使用。




a）网络

| S | R | $Q_a$ | $Q_b$ | |
|---|---|---|---|---|
| 0 | 0 | 0/1 | 1/0 | （不变化） |
| 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 0 | |

b）真值表




c）时序图

图A-23 使用或非门实现的基本锁存器

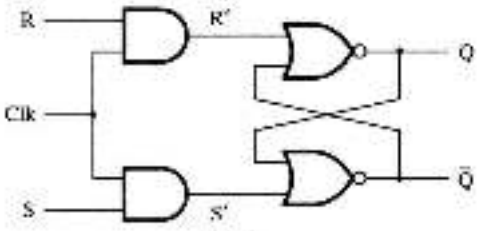
根据前面电路操作的性质，将S线和R线称作置位（set）和复位（reset）输入。由于通常不使用R = S = 1，故将$Q_0$和$Q_0$输出分别标记为Q和$\overline{Q}$。但是，$\overline{Q}$只是代表锁存器的第二个输出，而并不是Q的反，因为输入值R = S = 1的结果是Q = $\overline{Q}$ = 0。

## A.6.1 门控锁存器

许多应用需要有R和S以外的输入控制锁存器置位或复位的时间，这个输入称为时钟（clock）。组合的结果称作门控SR锁存器（gated SR latch）。该锁存器的逻辑电路、真值表、时序图和图形符号如图A-24所示。当时钟Clk等于1时，信号S'和R'的值分别等于S和R。另一方面，当Clk = 0时，信号S'和R'均等于0，并且锁存器不会发生任何状态变化。

至此我们一直使用真值表描述逻辑电路的行为。真值表给出了一个电路的各种输入值及相对应的输出。每个输入值唯一地定义输出的逻辑电路为组合电路（combinational circuit）。这是在A.1到A.4节中讨论的一类电路。当出现存储元件后，我们得到了一种不同种类的电路。这种电路的输出函数不只取决于输入变量的当前值，还反映了输入变量以前的行为。图A-24给出的就是这样一个例子。这种类型的电路称为时序电路（sequential circuit）。

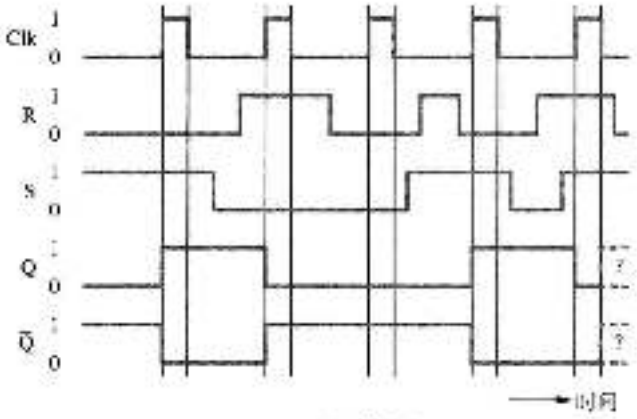
491 ↓ 493



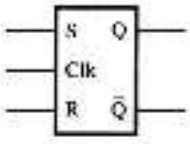

a）电路

| Clk | S | R | Q(t+1) |
|---|---|---|---|
| 0 | x | x | Q(t)（不变化） |
| 1 | 0 | 0 | Q(t)（不变化） |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | x |

b）真值表




c）时序图




d）图形符号

图A-24 门控SR锁存器

由于存储的特性，锁存器的真值表需要修改，以显示当前状态所产生的影响。图 A-24b 描述了门控 SR 触发器的行为，其中 $Q(t)$ 代表它的当前状态。到下一个状态 $Q(t+1)$ 的转换发生在一个时钟脉冲之后。注意当输入值为 S = R = 1 时，由于前面讨论的原因，$Q(t+1)$ 的值是不确定的。

如图 A-25 所示，门控 SR 锁存器可以用与非门实现。这是一个有用的例子，可以证明这个电路在功能上等效于图 A-24a 中的电路（参见习题 A.20）。

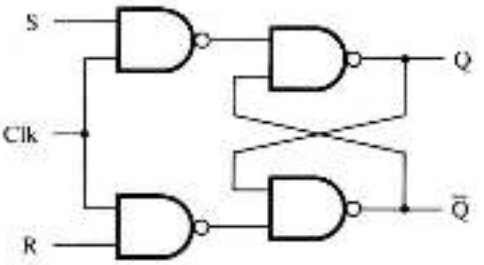
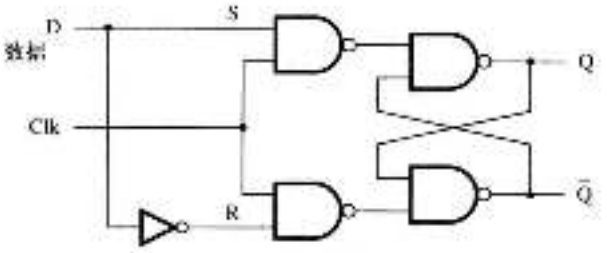



图 A-25 使用与非门实现门控 SR 锁存器

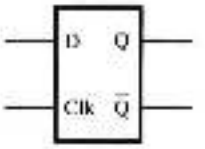
第二种类型的门控锁存器称为门控 D 锁存器（gated D latch），如图 A-26 所示。在这种情况下，S 和 R 两个信号是从单一输入 D 导出的。在一个时钟脉冲中，如果 D = 1，输出 Q 被置位为 1，或当 D = 0 时输出被复位为 0。D 触发器在时钟为高电平时采样输入 D 的值并存储该值，直到下一个时钟脉冲到来。



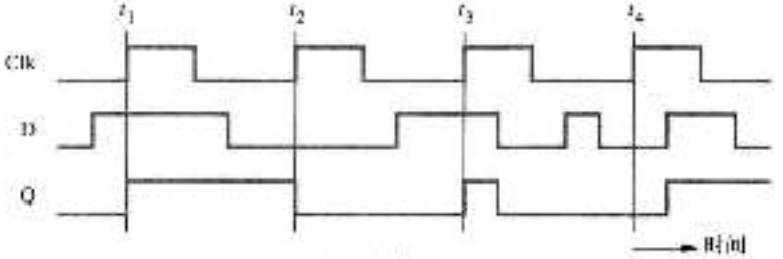

a）电路

| Clk | D | $Q(t+1)$ |
|---|---|---|
| 0 | x | $Q(t)$ |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

b）真值表




c）图形符号




d）时序图

图 A-26 门控 D 锁存器

## A.6.2 主从触发器

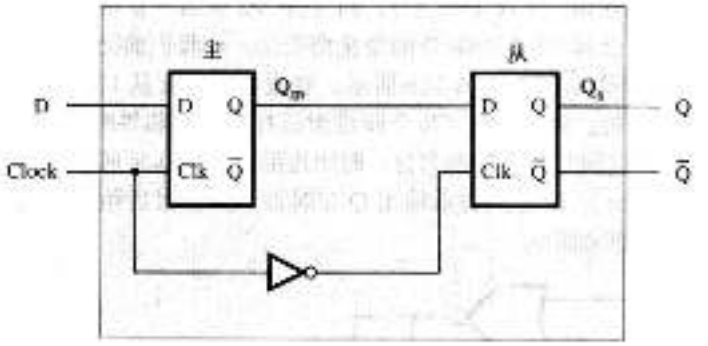
在图 A-24 的电路中，我们假设了当 Clk = 1 时，输入 S 和 R 不发生改变。考察电路会发现，在这个时间内，不管输入 S 和 R 发生什么变化，输出都会立即做出响应。类似地，在图 A-26 的电路中，当 Clk = 1 时，Q = D。在许多情况下，这种情况不是我们想要的，尤其是对于包含计数器和移位寄存器的电路，关于计数器和移位寄存器我们之后会讨论。在这样的电

路中，将逻辑条件立即从数据输入端（R、S和D）传递到锁存器的输出往往会导致错误的操作。主从（master-slave）结构解决了这个问题。如图A-27a所示，两个门控D锁存器可以连接成一个主从D触发器（master-slave D flip-flop）。首先，当Clock = 1时主触发器与输入D相连。时钟从1到0的跳变使主触发器与输入断开，并将主触发器中存储的内容传送到从触发器。我们可以看到，在输入D和输出Q之间任何时候都不存在直接的通路。

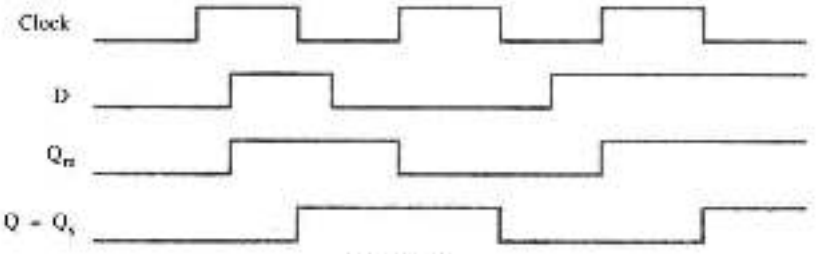
需要注意的是，当Clock = 1时，主触发器的状态直接受到输入D变化的影响。从触发器的功能是在将主触发器设置为由输入D决定的下一个状态值的期间，保持触发器的输出值。时钟从1跳变到0后，新状态就从主触发器传送到从触发器。这时，主触发器已经与输入断开，因此输入D的任何变化都不会影响传送过程。状态转换的例子如图A-27b的时序图所示。

触发器（flip-flop）是指一种在控制时钟信号的边缘改变状态的存储元件。在前面讨论的主从D触发器中，在时钟的下降沿（1到0）发生明显的变化。当变化达到从触发器的Q端时，就可以观察到该变化。注意在图A-27的电路中，也可以使用反向的时钟控制主触发器，而用原时钟控制从触发器。这时，触发器输出Q的变化就会在时钟的上升沿发生。

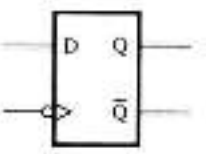
图A-27c给出了触发器的图形符号。我们使用一个箭头来代替标签Clk定义此触发器的时钟输入。这是定义触发器上升沿触发状态变化的标准表示方法。在我们的图中是下降沿触发状态变化的，因此在时钟输入端（箭头之外）再画一个小圆圈。




a）电路




b）时序图




c）图形符号

图A-27 主从D触发器

### A.6.3 边沿触发

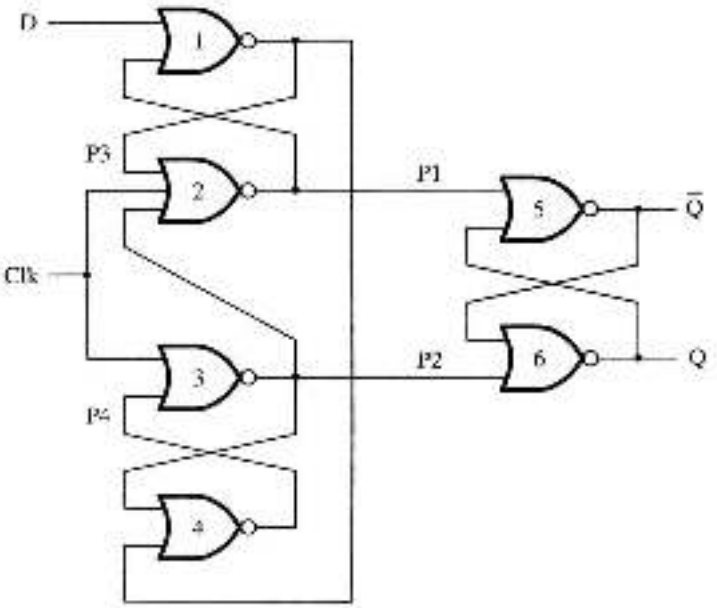
如果输入端的数据只在时钟信号跳变时传送给输出端，我们就说触发器是边沿触发（edge triggered）的。在其他所有时间输入与输出都是断开的。上升沿（前沿）触发（positive (leading) edge triggered）和下降沿（后沿）触发（negative (trailing) edge triggered）分别描述了数据传送发生在0到1和1到0时钟跳变的触发器。为了进行正确的操作，边沿触发的触发器要求时钟脉冲的触发沿定义明确并且跳变时间很短。图A-27中的主从触发器是一个下降沿触发器。

下降沿触发D触发器的另一种实现如图A-28a。让我们看一下这个触发器的操作过程。如果Clk = 1，门2和3的输出均等于0。因此，触发器输出Q和$\overline{Q}$维持触发器的当前状态。容易验证在这段时间内，P3点和P4点会立即响应D的变化。P3点保持与$\overline{D}$相等，而P4点保持与D相等。当Clk下降为0时，这些值通过门2和3分别传送到P1和P2。因而，由门5和6组成的输出锁存器得到了需要存储的新状态。

我们现在验证一下当Clk = 0时，D的变化不会改变P1点和P2点。考虑两种情况。首先，假设在Clk的下降沿D = 0。P2处的1使得门2和4都分别有一个输入保持为1，这使得P1和P2为0和1，与D的任何变化无关。接着，假设在Clk下降沿D = 1。P1处的1使得D的任何变化都不会影响到门1，门1保持为0。

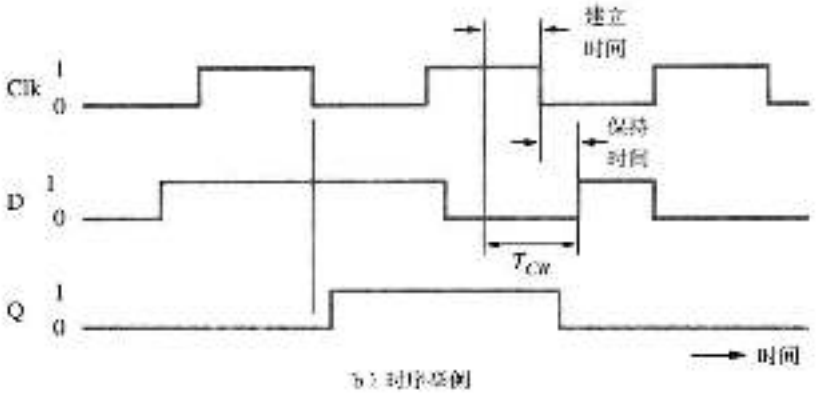
当下一个时钟脉冲开始时Clk上升为1，P1点和P2点再一次被强制置为0，使输出与电路的其他部分断开。P3点和P4点跟随D的变化而变化，如我们前面所描述的一样。

这种D触发器的操作示例如图A-28b所示。触发器在Clk从1跳变到0时的状态等于在这一跳变之前输入D的值。但是，在*Clk*下降沿附近存在一个临界时间段$T_{cs}$，期间D上的值不应改变，如图所示，这段时间分为两部分：时钟边沿前的建立时间（setup time）和时钟边沿后的保持时间（hold time）。时序图显示输出Q在时钟下降沿过后稍晚一些才有变化。这是由于受到了与非门传输延迟的影响。




a）网络

图A-28 下降沿触发的D触发器



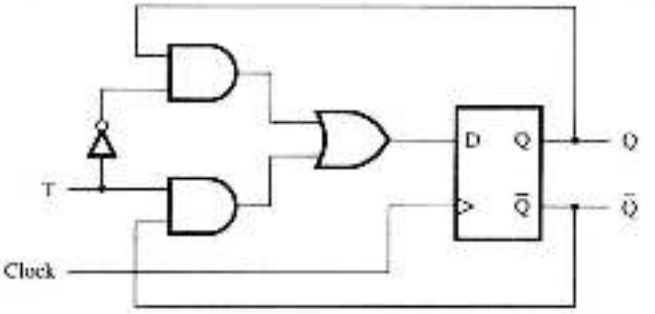

b）时序举例

图 A-28（续）

## A.6.4 T触发器

使用最为广泛的触发器是D触发器，因为它们对数据的临时存储很有用。但是，在一些应用中，使用其他类型的触发器会很方便。我们将在A.8节中讨论的计数器电路就是由T触发器构成的。当其输入T等于1时，T触发器（T flip-flop）在每个时钟周期都改变状态。我们称它“翻转”（toggle）自己的状态。

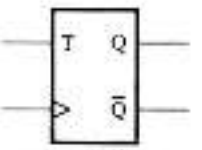
图A-29显示了T触发器。如图A-29a所示，它的电路从D触发器派生得来。图中还给出了它的真值表、图形符号以及时序图举例。注意现在假设的是一个上升沿触发的触发器。



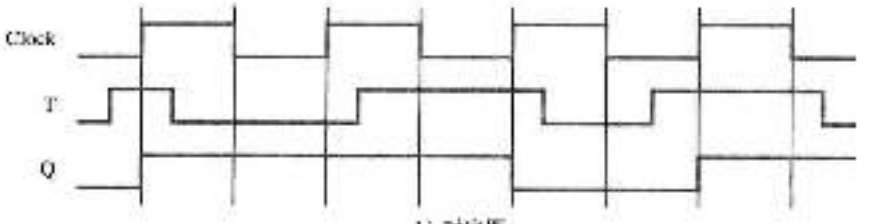

a）电路

| T | Q(t + 1) |
|---|---|
| 0 | Q(t) |
| 1 | $\bar{Q}(t)$ |

b）真值表




c）图形符号




d）时序图

图 A-29 T触发器

### A.6.5 JK触发器

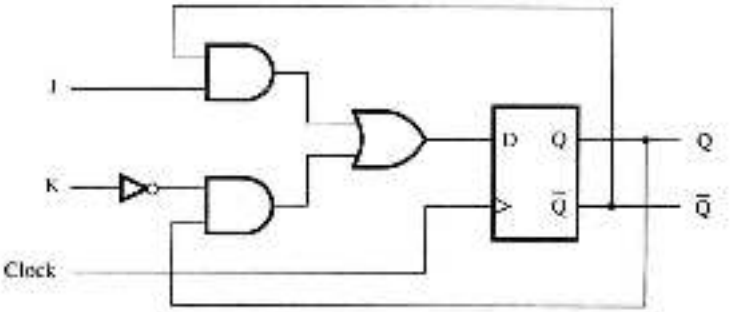
在实际中会遇到的另一种触发器是JK触发器（JK flip-flop），它组合了SR触发器和T触发器的功能，如图A-30所示。它的操作由图A-30b中给出的真值表定义。表中的前三行定义了与图A-24b（当Clk = 1时）中一样的表项，所以J和K分别对应S和R。当输入值J = K = 1时，下一个状态定义为与当前触发器相反的状态。即当J = K = 1，JK触发器的功能就相当于一个T触发器，翻转当前的状态。

JK触发器可以使用D触发器联结成

498 - 500

$$D = J\overline{Q} + \overline{K}Q$$

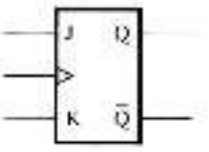
实现。相应的电路如图A-30a所示。




a）电路

| J | K | $Q(t+1)$ |
|---|---|---|
| 0 | 0 | $Q(t)$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q}(t)$ |

b）真值表




c）图形符号

图A-30 JK触发器

JK触发器可以有多种用途。它可以像D触发器一样用来存储数据。它也可以用来构造计数器，因为当J和K输入连在一起时，就表现为一个T触发器。

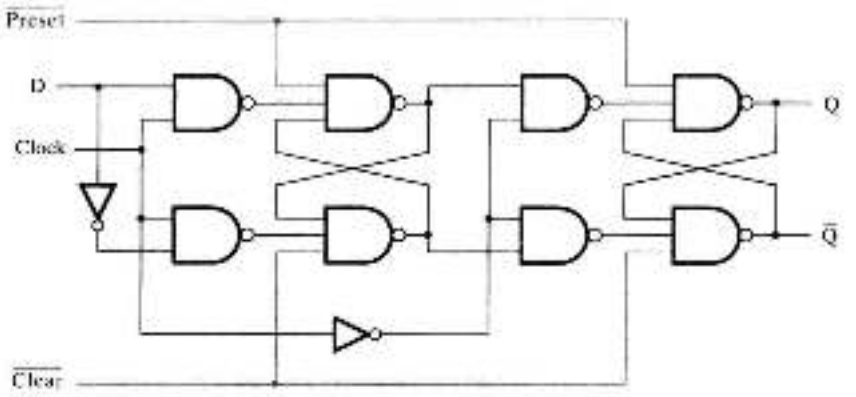
### A.6.6 带预置和清除的触发器

触发器的状态由它的当前状态和输入端的逻辑值决定。有时需要强制将触发器置于某个特定状态0或1，而不管它的当前状态和正常输入值是什么。例如，当打开计算机时，需要将所有的触发器都置于一个已知状态。通常是将它们的输出复位为0状态。有些时候也需要将一些触发器置为1状态。

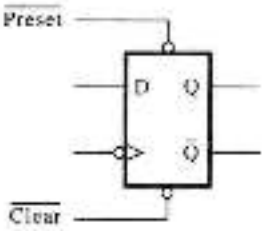
图A-31显示了如何将预置和清除控制输入加到主从D触发器上，不管D输入和时钟是什么，都将触发器强制置为1或0。图中的上划线和圆圈显示这些输入是低电平有效的。当预置（$\overline{\text{Preset}}$）和清除（$\overline{\text{Clear}}$）输入都等于1时，触发器就在正常情况下由时钟和D输入控制。

501

当Preset等于0时，触发器强置于1状态。而当$\overline{\text{Clear}} = 0$时，触发器强置于0状态。在其他类型的触发器中也常常加入预置和清除控制。




a) 电路




b) 图形符号

图 A-31 带预置和清除的主从 D 触发器

## A.7 寄存器与移位寄存器

每个触发器可以用来存储一位数据。但是，在以字为单位处理数据的机器中，字由许多位组成（可能是 64 位）。为了处理数据的方便，我们将许多触发器组合为一种常用的结构，称为寄存器（register）。寄存器中所有触发器的操作都由共同的时钟控制。因此，数据被写入（载入）触发器或从触发器中读出都同时进行。

处理数字数据经常要求有移位和循环移位数据的能力，因此需要提供有此能力的硬件。能实现这两种操作的简单器件是寄存器，其内容可以每次向左或向右移动一位。例如图 A-32 [502]
中的 4 位移位寄存器。它是由 D 触发器连接成的，故每个时钟脉冲会引起从 $F_i$ 到 $F_{i-1}$ 的内容（状态）传递，表现为“右移”。数据被顺序送入或送出寄存器，将输出接至输入可以实现数据的循环移位。

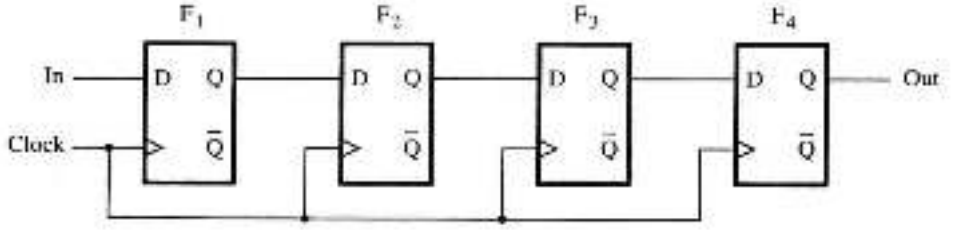



图 A-32 简单的移位寄存器

移位寄存器的正确操作要求其内容在每个时钟脉冲恰好移动一位。这就限制了可以使用

的存储元件类型。图 A-26 描述的门控锁存器不适合实现此操作，因为当时钟为高电平时，D 输入的值很快就传送到输出。接着，数据又以相同的方式经过下一个门控锁存器。因此，在一个时钟脉冲内不能控制移位发生的次数。这个次数由门控锁存器的传输延迟和时钟脉冲的持续时间决定。解决这个问题的办法是使用主从或边沿触发器。

能够并行载入和读取的移位寄存器是很实用的。这可以通过使用一些附加的门来实现。如图 A-33，它显示了用 D 触发器构成的 4 位寄存器。这个寄存器既可以串行也可以并行地载入。当计时开始时，如果 $\overline{\text{Shift}}/\text{Load}=0$ 就发生移位；否则，进行并行载入。

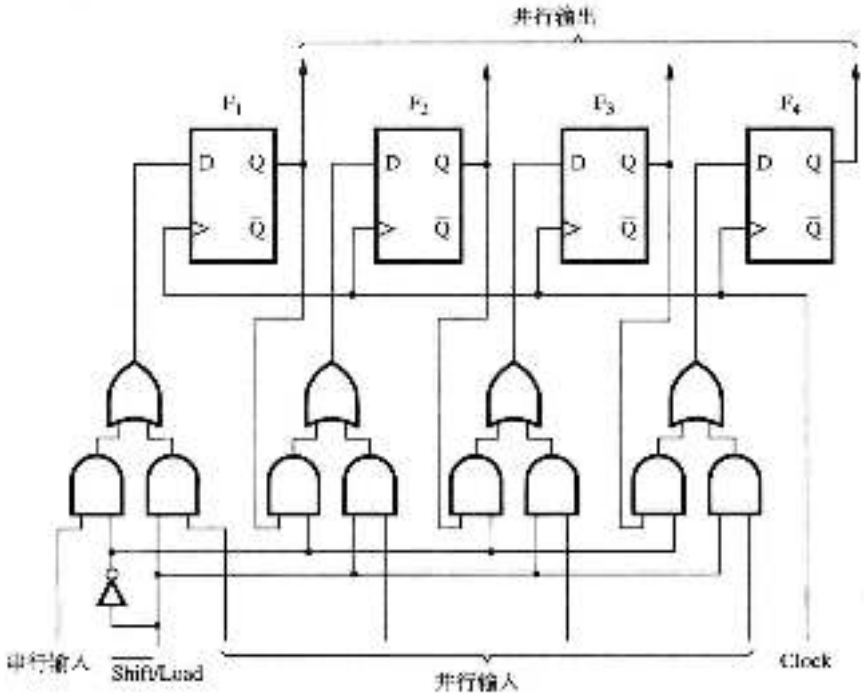



图 A-33 并行访问的移位寄存器

## A.8 计数器

在前面的一节中，我们讨论了触发器在构造移位寄存器时的应用。它们在实现计数器（counter）电路时也同样有用。计数器在数字机器中的重要性无需多言。除了具有一般的计数功能外，计数器还可以用来产生控制和时序信号。由高频时钟驱动的计数器可以产生频率为原始时钟分频的信号。在这样的应用中计数器作为定标器（scaler）使用。

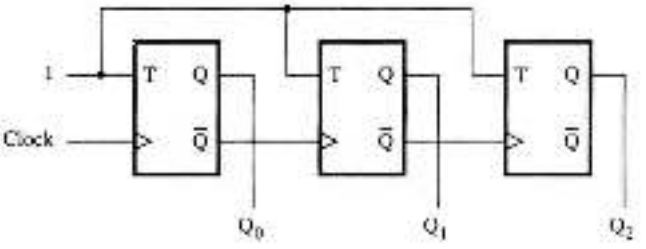
图 A-34 显示了由 T 触发器构成的一个简单的 3 段（或 3 位）计数器。回忆一下当 T 输入等于 1 时，触发器表现为一个双态触发器，即每个连续的时钟脉冲都会引起状态的变化。因此，两个时钟脉冲将会使 $Q_0$ 从 1 状态变为 0 状态再回到 1 状态，或从 0 到 1 再到 0。这说明 $Q_0$ 输出波形的频率是时钟频率的一半。类似地，因为第二个触发器是由 $Q_0$ 驱动的，所以 $Q_1$ 的波形是 $Q_0$ 频率的一半，即时钟频率的 1/4。注意我们假设每个触发器的状态都在时钟输入的上升沿发生改变。

这样的计数器常被称作行波计数器（ripple counter），因为输入时钟脉冲的影响像行波一样通过计数器。例如，脉冲 4 的上升沿将 $Q_0$ 的状态从 1 变到 0。$Q_0$ 的这个变化会迫使 $Q_1$ 从

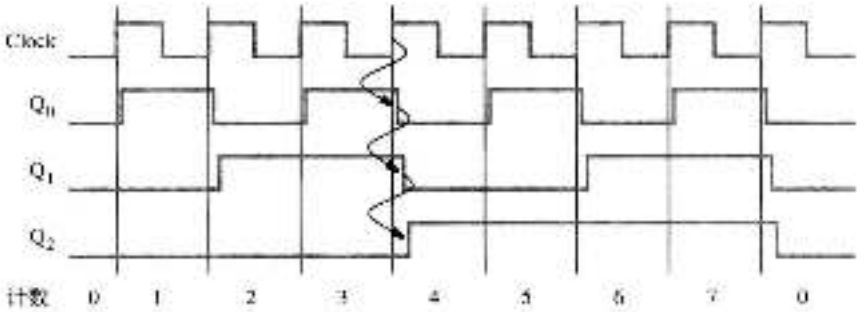
1 变到 0，而后按顺序迫使 $Q_2$ 从 0 到 1。如果每个触发器产生的延迟为 Δ，则 $Q_2$ 的延迟就是 3 Δ。当要求计数器高速运行时，这样的延迟就会成为一个问题。但在许多应用中，这些延迟与时钟周期相比非常小，因而可以被忽略。

增加一些额外的逻辑门，就可以构建一个“同步”计数器，其每个阶段都在公用时钟控制之下，因此所有触发器可以同时改变状态。因为总的延迟显著减少，所以这样的计数器可以高速运行。与之相对，图 A-34 中的计数器称为“异步”计数器。

503 ~ 504




a）电路

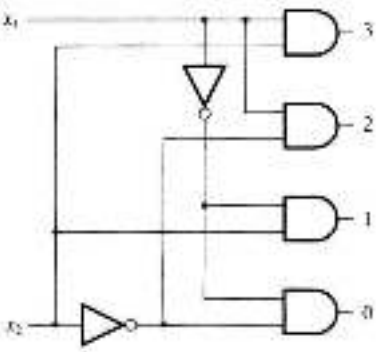



b）时序图

图 A-34 3 位升值计数器

## A.9 译码器

计算机中的许多信息都是高度编码的。在指令中，一个 $n$ 位字段可以用来表示从 $2^n$ 种可能的动作中选出一个去执行。为了执行所需的操作，编码的指令必须先被译码。能够接受 $n$ 变量输入并在 $2^n$ 个输出线路中产生一个相应输出信号的电路称作译码器（decoder）。图 A-35 给出了一个 2 输入 4 输出的简单译码器的例子。如图所示，由输入 $x_1$ 和 $x_2$ 从 4 条输出线路中选出一个。被选出的输出具有逻辑值 1，剩下的输出值都是 0。



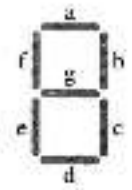

| $x_1$ | $x_2$ | 有效输出 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 2 |
| 1 | 1 | 3 |

图 A-35 2 输入 4 输出译码器

译码器还存在其他形式。例如，使用 BCD 码的信息经常需要一个具有 4 变量 BCD 输入的译码电路，它能从 10 个可能的输出中选择 1 个有效输出。考虑另一个具体的例子，一个能够驱动七段显示器的译码器。图 A-36 给出了用于显示的七段元件的结构。容易看出，

从0到9的任意十进制数都可以通过打开一些段（亮）而关闭另一些段（暗）的方式显示出来。表中给出了一些必要的函数。它们可以使用图中的译码器电路实现。注意这个电路是由与非门组成的。希望读者自己验证该电路实现了所需的功能。

| No. | $x_1$ | $x_2$ | $x_3$ | $x_4$ | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |



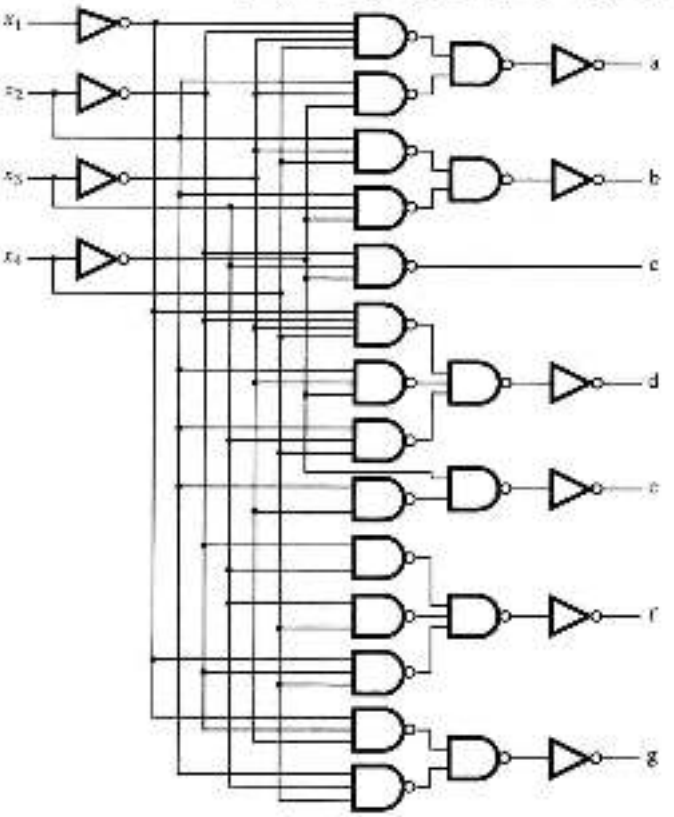




图 A-36 BCD 码七段显示器译码器

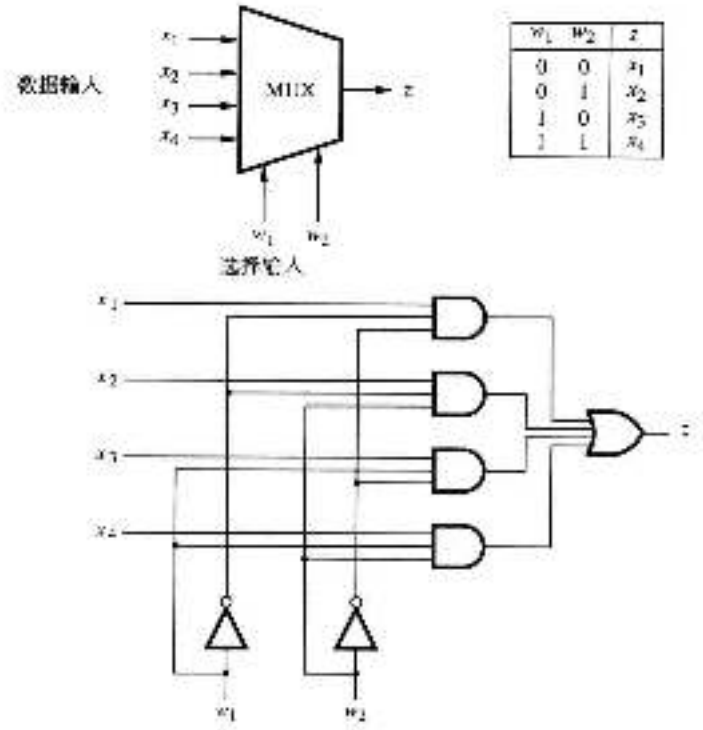
## A.10 多路复用器

在前面一节中，我们看到译码器根据输入信号选择一条输出线。被选中的输出线逻辑值为1，而其他输出均为0。另一种非常有用的选择电路能够从 $n$ 个数据输入中选择一个作为输出。选择操作是由一组“选择”输入控制的。这样的电路称作多路复用器（multiplexer）。图 A-37 给出了一个多路复用器电路的例子。它有两个选择输入端，$w_1$ 和 $w_2$。它们的4个可

能值用来从4个输入 $x_1$、$x_2$、$x_3$ 或 $x_4$ 中选择一个作为输出 $z$。图中也给出了能够实现所需操作的简单逻辑电路。显然，更大的多路复用器也可以用相同的结构实现。用 $k$ 个选择输入端将 $2^k$ 个数据输入端中的一个连接到输出。

多路复用器的一个常见应用是筛选可能来自许多不同源的数据。例如，从4个数据源加载一个16位数据寄存器可以用16个4输入多路复用器实现。

505 ~ 507




| $w_1$ | $w_2$ | $z$ |
|---|---|---|
| 0 | 0 | $x_1$ |
| 0 | 1 | $x_2$ |
| 1 | 0 | $x_3$ |
| 1 | 1 | $x_4$ |

图 A-37 4输入多路复用器

多路复用器也可作为实用的基本元件用来实现逻辑函数。考虑由图 A-38 中的真值表定义的函数 $f$。它可以如图中那样提取出变量 $x_1$ 和 $x_2$ 来表示。可以发现对 $x_1$ 和 $x_2$ 的每个值，函数 $f$ 对应于4项中的一个：0、1、$x_3$ 或 $\bar{x}_3$。这意味着可以使用一个4输入多路复用器电路，其中 $x_1$、$x_2$ 是选择4个数据输入之一的选择输入端。此时，如果将0、1、$x_3$ 和 $\bar{x}_3$ 按照真值表的要求连接到数据输入，则多路复用器的输出就是函数 $f$。这种方法完全是通用的。任何3变量函数都可以由一个4输入多路复用器实现。类似地，任何4变量的函数都可以由一个8输入多路复用器实现，依次类推。

| $x_1$ | $x_2$ | $x_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| $x_1$ | $x_2$ | $f$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | $x_3$ |
| 1 | 0 | 1 |
| 1 | 1 | $\bar{x}_3$ |

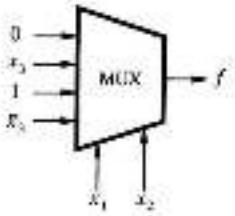



图 A-38 用多路复用器实现逻辑函数

508

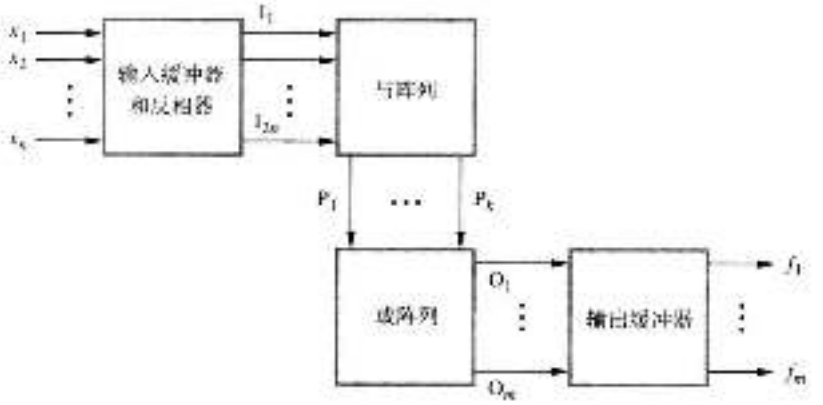
## A.11 可编程逻辑器件

在前面几节中我们介绍了如何使用门和触发

器来实现逻辑电路。在这一节中，我们将要考虑只需要对其编程就可以实现能执行所需功能的电路的器件。它们称作可编程逻辑器件（Programmable Logic Device，PLD）。

### A.11.1 可编程逻辑阵列

在A.2节和A.3节中解释过，任何组合逻辑函数都可以用积之和的形式实现。能实现多种组合函数的通用电路可以组织为图A-39所示的形式。它有 $n$ 个输入变量（$x_1, \cdots, x_n$）和 $m$ 个输出函数（$f_1, \cdots, f_m$）。每个函数 $f_i$ 都由包含输入变量的乘积项之和组成。变量 $x_1, \cdots, x_n$ 以真值或反值的形式被送入与阵列，形成 $k$ 个乘积项。然后这些乘积项又被送入或阵列，形成输出函数。为使该电路可被用户定制，可以使用与阵列和或阵列的可编程连接。




图A-39 PLD的框图

如果电路同与阵列、或阵列的连接是可编程的，则该电路称为可编程逻辑阵列（Programmable Logic Array，PLA）。图A-40用简单的例子说明了PLA的功能结构。当可编程连接没有同与门给出的输入相连时，该输入的表现就好像是逻辑1在驱动它一样，也就是说，该输入对这个门实现的乘积项不起作用。类似地，如果没有同或门给出的输入相连时，该输入对这个门的输出没有任何影响，就好像是逻辑0在驱动它一样。

可编程连接可以用不同的方法实现。一种方法是，熔断不需连接位置的金属丝。这需要使用高于常值的电流。另一种可行方法是使用可擦除存储元件（参见8.3.3节的EPROM存储电路）控制的晶体管开关来提供所需的连接。这样的PLA可以重新编程。

图A-40中简单的PLA可以从3个输入变量生成4个乘积项。使用这些乘积项可以实现两个输出函数。有些乘积项可以被不止一个输出函数使用。PLA实现了下面两个函数：

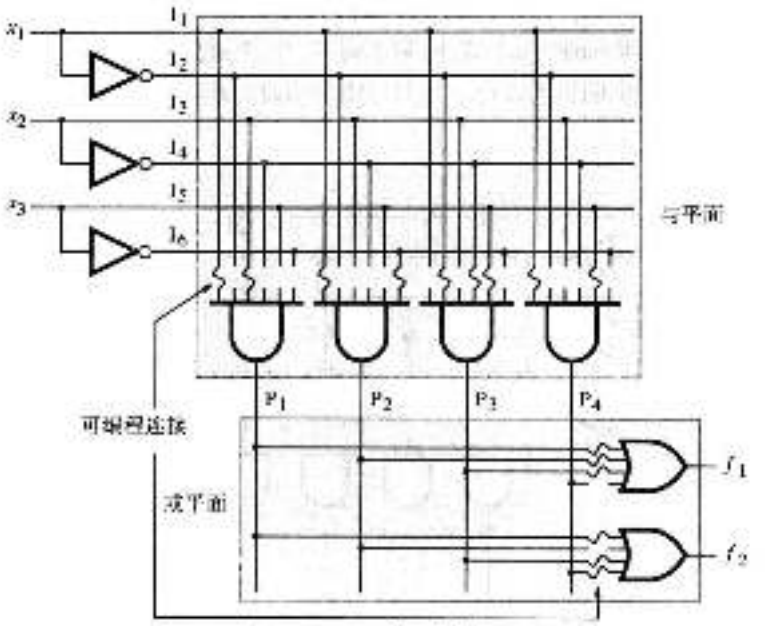
$$f_1 = x_1x_2 + x_1x_3 + x_1x_2x_3$$

$$f_2 = x_1x_2 + x_1x_3 + \bar{x}_1x_2x_3$$

在这两个函数中有两个相同项，因此只需要4个乘积项。

尽管图A-40清楚地描述了PLA的基本功能，但这种表示形式对于更大的PLA就显得有些笨拙了。在科技文献中的一个惯例是使用只有一条输入线的相应门符号来表示乘积项和求和项。对于每个已编程连接在该线上打一个叉号“×”。图A-41使用这种图示方法表示了图A-40中PLA的例子。为了实现输入变量的不同函数，图中任何垂直线和水平线的交点都可以是一个可编程的连接。

509
510




$$f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$$

$$f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 = x_1x_3$$

图 A-40 PLA 的功能结构

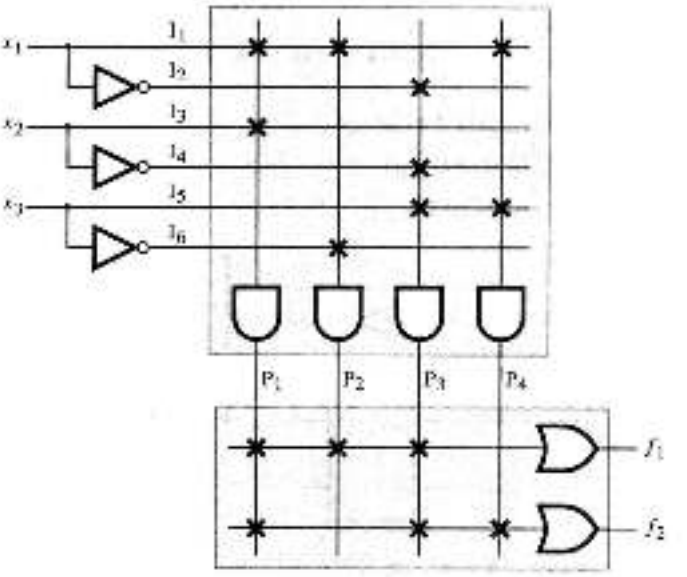


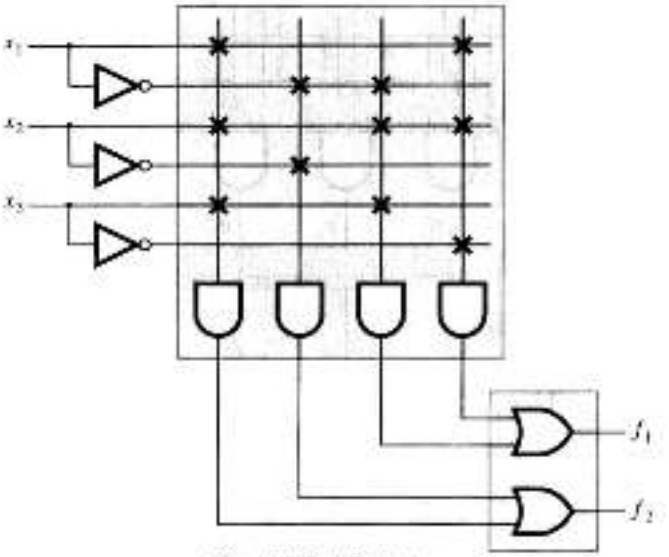

图 A-41 图 A-40 中 PLA 的简化图

## A.11.2 可编程阵列逻辑

在 PLA 中，与阵列、或阵列的输入都是可编程的。有一种相似电路，其与阵列的输入是可编程的但同或门的连接是固定的，在实际应用中它提供的灵活性就足够了。这样的电路称作

可编程阵列逻辑（Programmable Array Logic，PAL）电路。

图 A-42 给出了实现两个函数的 PAL 的简单例子。每个或门所连接的与门数量决定了给定乘数积之和表达式中乘积项的最大数目。与门与特定的或门永久地保持连接，这意味着在输出函数中不能共享乘积项。




$$f_1 = x_1x_2\bar{x}_3 + \bar{x}_1x_2x_3$$

$$f_2 = \bar{x}_1\bar{x}_3 + x_1x_2x_3$$

图 A-42 PAL 举例

或门的输出若连接触发器，会使 PAL 电路具有更多功能。图 A-43 显示了这种电路的灵活性。多路复用器用来选择在或门输出端表示的 $f$ 是真值、反值还是存储值（从前一个时钟周期得出）。多路复用器的选择输入以可编程连接的形式提供。

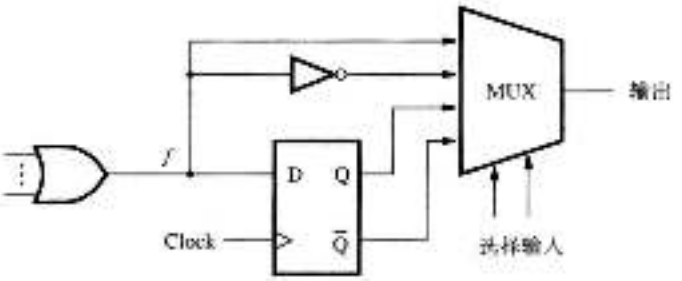



图 A-43 包含触发器的 PAL 元件

## A.11.3 复杂可编程逻辑器件（CPLD）

PAL 结构用于称为复杂可编程逻辑器件（Complex Programmable Logic Device，CPLD）的较大器件中。这些器件包括许多类 PAL 块和可编程互连线。图 A-44 给出了 CPLD 芯片的组织。每个类 PAL 块都与许多输入/输出管脚相连。类 PAL 块之间的连接通过对与互连线相关

联的开关编程来建立。

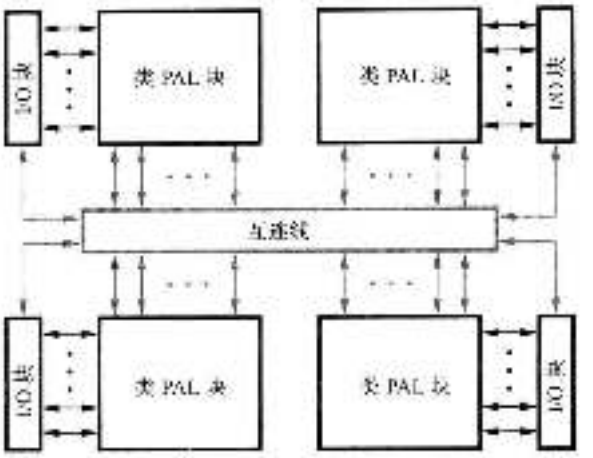



图 A.44 复杂可编程逻辑器件（CPLD）组织

互连线由水平线和垂直线构成。每条水平线都可通过对相应开关的编程与几条垂直线连接。要将每条水平线与任意的垂直线连接组成完全链接是不切实际的，因为所需的开关数会很大。少量的开关即可实现满足需求的连通度。

市场上的 CPLD 尺寸不同，从仅含 2 个到 100 个以上类 PAL 块的都有。通过向 JTAG 端口（JTAG port）加载串行位流的编程信息可以对 CPLD 芯片编程。JTAG 端口是一个 4 管脚端口，源于由联合测试工作组（Joint Test Action Group）制定的 IEEE 标准。

## A.12 现场可编程门阵列

最通用的可编程逻辑器件称为现场可编程门阵列（Field-Programmable Gate Array，FPGA）。图 A-45 给出了 FPGA 的概念框图。它由一个逻辑块（表示为较大的黑框）阵列组成，这些逻辑块可以由通用互连资源连接。互连开关（interconnect）用较小的方块表示，由导线和可编程开关组成。这些开关用于连接逻辑块和导线，以及在不同导线之间建立所需的连接。这给予了芯片很大的路由灵活性。I/O 块为芯片管脚的访问提供了输入和输出缓冲器。

逻辑块和互连结构的设计是多种多样的。逻辑块可能只是 A.10 节中简单的基于多路复用器的电路，可以实现逻辑函数。另一种流行的设计是将简单的查找表（Lookup Table，LUT）作为逻辑块。例如，一个 4 输入的 LUT 可以用 16 位存储电路实现，该电路存储逻辑函数的真值表。每个存储位对应于输入变量的真值或反值的一个组合。对这样的查找表编程可以实现 4 变量的任何函数。逻辑块可能会包含触发器，以提供类似图 A-43 中额外的灵活性。

除了逻辑块，许多 FPGA 芯片还包含相当数量的存储单元（图 A-45 中没有显示），可以实现诸如先进先出（FIFO）队列或片上系统的 RAM 和 ROM 组件等结构。这些在第 11 章进行了讨论。

FPGA 可用于多种规格中。最大的器件包含数十亿的晶体管，可以用来实现很大的逻辑电路。FPGA 的不断普及是因为它允许设计者在一块芯片上实现非常复杂的逻辑电路，而无需设计制造定制的 VLSI 芯片，这些芯片非常昂贵而且费时。使用 CAD 工具可以在几天内完成 FPGA 的设计，而制造定制 VLSI 芯片需要花费几个月的时间。关于使用 FPGA 器件和 CAD

工具进行电路设计的引导性讨论，读者可以查阅参考文献 [1]。

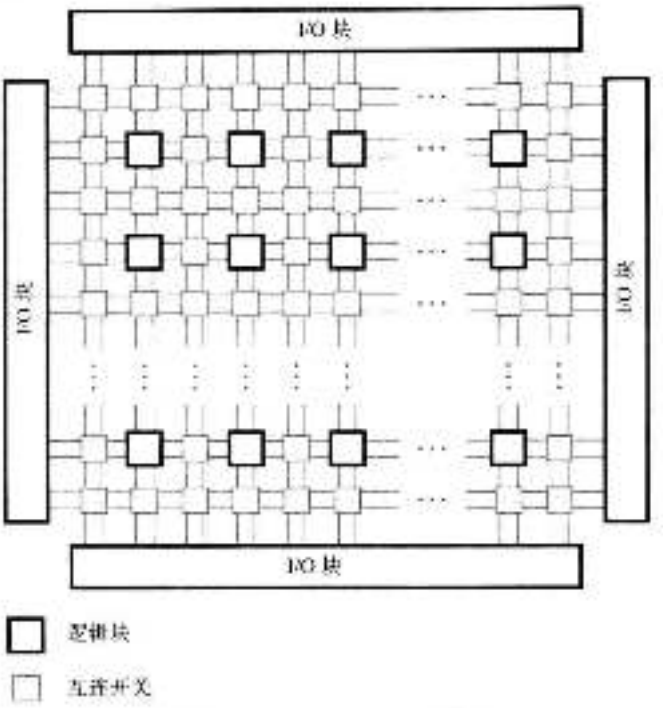



图 A-45 FPGA 的概念框图

## A.13 时序电路

组合电路的输出完全由当前的输入决定。A.9 节和 A.10 节给出的译码器和多路复用器就是组合电路的例子。另一类电路的输出是由当前的输入和以前的输入序列共同决定。它们称为时序电路（sequential circuit）。这样的电路可以处于不同的状态（state），这些状态由给定时间内的输入序列决定。电路的状态决定了电路在不同输入模式下的行为。在 A.7 节和 A.8 节中，我们曾经遇到这种电路的两种具体形式：移位寄存器和计数器。本节将介绍时序电路的一般形式，并简单介绍这类电路的设计方法。

### A.13.1 升值 / 降值计数器的时序电路设计

图 A-34 给出了由三个 T 触发器实现的升值计数器的结构，按照 0, 1, 2, …, 7, 0, …的顺序计数。也可以用类似的电路实现降值计数，即按 0, 7, 6, …, 1, 0, …计数（见习题 A.26）。这些简单的电路都利用了 T 触发器的翻转性质。

下面考虑一下使用 D 触发器实现计数器的可能性。我们将设计一个既能升值计数也能降值计数的计数器作为具体的例子，升值或降值由一个外部控制输入的值决定。为了使这个例子简单一点，我们将计数器限制为模 4 计数器，它只需要两个状态位来表示四个可能的计数值。下面将说明如何使用构造时序电路的基本方法来设计这个计数器。所需电路在输入信号 $x$ 为 0 时进行升值计数，$x$ 为 1 时进行降值计数。使用图 A-27 和图 A-28 中所介绍的 D 触发器，计数发生在时钟信号的下降沿。假设我们对计数为 2 时的状态感兴趣，则输出信号 $z$ 在计数为 2 时为 1，而在其他时刻都为 0。

511 / 515

这个计数器可以作为时序电路实现。当一个时钟脉冲到来时，为了确定新的计数，知道 $x$ 的值和当前的计数值就够了。我们无需知道先前输入值的实际顺序，而只需知道当前的计数值。这个计数值决定了电路的当前状态（present state），这是电路保存的关于先前输入的唯一信息。如果现在的计数是 2 并且 $x = 0$，下一个计数就是 3。从 3 降值计数或从 1 升值计数到 2 没有任何区别。

在给出电路实现之前，我们先用状态图描述计数器所需的行为。计数器具有 4 个不同的状态：S0，S1，S2 和 S3。状态图（state diagram）是用圆圈（有时称为节点）表示状态的图。状态之间的转换用带标记的箭头表示。与箭头对应的标记指示了会使特定转换发生的输入 $x$ 的值。图 A-46 显示了这个升值/降值计数器的状态图。例如，从状态 S1（计数值 = 1）发出的箭头在输入 $x = 0$ 时指向状态 S2，这表示向状态 S2 的转变。从 S2 到 S3 的箭头表明当 $x = 0$ 时，下一个时钟脉冲会发生从状态 S2 到状态 S3 的转变，并且当电路在状态 S2 时，输出 $z$ 必是 1，而在状态 S0、S1 和 S3 时，输出 $z$ 必为 0，图中的每个圆圈表明了这种情况。

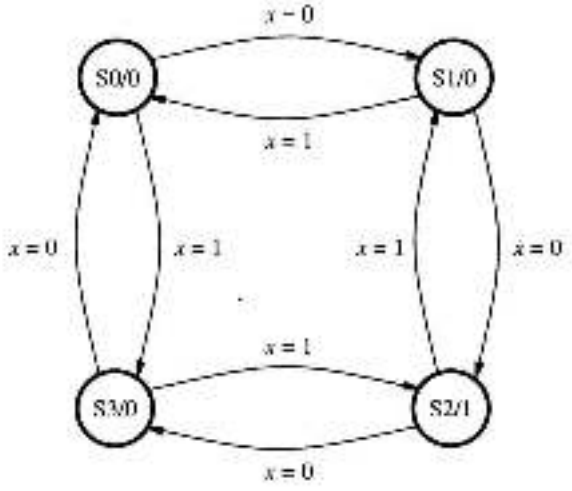



图 A-46 检测计数 2 的模 4 升值/降值计数器状态图

| 当前状态 | 下一状态 | | 输出 $z$ |
|---|---|---|---|
| | $x = 0$ | $x = 1$ | |
| S0 | S1 | S3 | 0 |
| S1 | S2 | S0 | 0 |
| S2 | S3 | S1 | 1 |
| S3 | S0 | S2 | 0 |

图 A-47 升值/降值计数器例子的状态表

注意，状态图描述了计数器的功能行为，而并没有提及它如何实现。图 A-46 可以用来描述按这种方式表现的电子数字电路，机械计数器或者计算机程序。状态图是描述具有时序行为的任何系统的有力工具。

表示状态图信息的另一种方法是使用状态表（state table）。图 A-47 给出了图 A-46 例子的状态表。表中显示了在输入 $x$ 下，从所有当前状态到下一状态（next state）的转换。表中还显示了每个状态中输出信号 $z$ 的值。

516 / 517

我们已经描述了一般条件下的升值/降值计数器，现在考虑它的实现。对表示计数值的 4 个状态进行编码需要两个位。设这两个位为 $y_2$（高位）和 $y_1$（低位）。计数器的状态由 $y_2$ 和 $y_1$ 的值决定。我们将它写成 $y_2\,y_1$ 的形式并给 $y_2\,y_1$ 赋值为：S0 = 00，S1 = 01，S2 = 10 和 S3 = 11。这样安排使得二进制数 $y_2\,y_1$ 可以明显地表示计数值。变量 $y_2\,y_1$ 称为时序电路的状态变量（state variable）。使用这一状态分配（state assignment），图 A-48 给出了我们所举例子的状态表。注意使用变量 $Y_2$ 和 $Y_1$ 来表示下一状态，它们的用法与表示当前状态的 $y_2$ 和 $y_1$ 一样。

需要注意的很重要的一点是，我们可以选择另一种不同的状态对 $y_2\,y_1$ 进行赋值。例如，可能赋值为 S0 = 10，S1 = 11，S2 = 01，S3 = 00。对于一个计数器电路来说，这种赋值方式没

有图 A-48 直观，但是结果电路仍然会运行正常。通常，使用不同状态赋值方式实现电路的成本也不相同（见习题 A.30）。

我们举这个例子的目的是使用 D 触发器存储连续时钟脉冲间的两个状态变量的值。触发器的输出 Q 是当前状态变量 $y_i$，输入 D 是下一状态变量 $Y_i$。注意 $Y_i$ 是 $y_2$、$y_1$ 和 $x$ 的函数，如图 A-48 所示。从图中我们可以看到

$$\begin{aligned} Y_2 &= \bar{y}_2 y_1 \bar{x} + y_2 \bar{y}_1 \bar{x} + \bar{y}_2 \bar{y}_1 x + y_2 y_1 x \\ &= y_2 \oplus y_1 \oplus x \\ Y_1 &= y_2 y_1 x + y_2 \bar{y}_1 \bar{x} + \bar{y}_2 \bar{y}_1 x + y_2 \bar{y}_1 x \\ &= \bar{y}_1 \end{aligned}$$

[518]

| 当前状态 | 下一状态 | | 输出 |
|---|---|---|---|
| | $x=0$ | $x=1$ | $z$ |
| $y_2 y_1$ | $Y_2 Y_1$ | $Y_2 Y_1$ | |
| 0 0 | 0 1 | 1 1 | 0 |
| 0 1 | 1 0 | 0 0 | 0 |
| 1 0 | 1 1 | 0 1 | 1 |
| 1 1 | 0 0 | 1 0 | 0 |

图 A-48 图 A-47 中例子的状态分配

输出 $z$ 由式子

$$z = y_2 y_1$$

决定。这些表达式形成了图 A-49 所显示的电路。

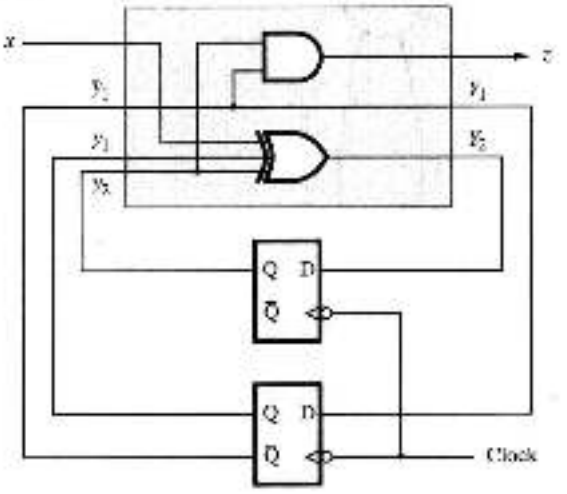



图 A-49 升值/降值计数器的实现

## A.13.2 时序图

了解计数器的时序图有助于完全掌握计数器电路的操作。图 A-50 给出了一个可能事件序列的例子。假设状态转换（触发器的值改变）发生在下降沿，并且计数器从状态 S0 开始。因为 $x=0$，计数器在 $t_0$ 时刻变为状态 S1，$t_1$ 时刻变为 S2，$t_2$ 时刻变为 S3。当计数器进入状态 S2 时，输出从 0 变到 1。当达到状态 S3 时又变到 0。在 $t_3$ 时刻，计数器返回 S0 状态。假设在这一时刻，输入 $x$ 变为 1，导致计数器降值计数。当计数再一次达到 S2 时，即在 $t_5$ 时刻，输出 $z$ 变为 1。

注意所有信号改变都在时钟下降沿之后立刻发生，并在下一个下降沿到来之前不会发生改变。从时钟边沿到变量 $y_i$ 改变之间的延迟是实现计数器电路触发器的传输延迟。还要注意
[519] 我们假设输入 $x$ 也由同一个时钟控制，并且它的转变只发生在临近时钟周期开始的时刻。这是所有改变均由一个时钟控制的电路的基本性质。这样的电路称作同步时序电路（synchronous

sequential circuit）。

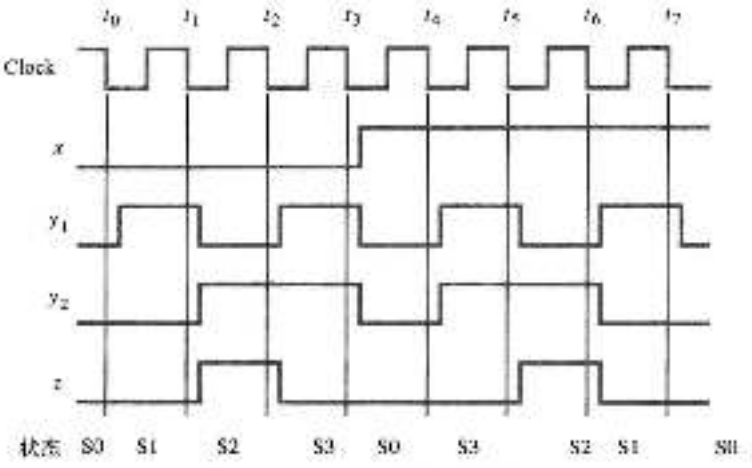



图 A-50 图 A-49 所示电路的时序图

另一个关注的重点是图 A-46 中状态图使用的标记与时序图的关系。例如，考虑 $t_1$ 与 $t_2$ 之间的时钟周期。在这个周期内，机器处于 S2 状态且输入值 $x=0$。这一情况在状态图中用从 S2 状态发出标号为 $x=0$ 的箭头表示。因为这个箭头指向 S3 状态，所以时序图显示了在下一时钟边沿 $t_2$ 时，$y_2$ 和 $y_1$ 变为 S3 状态相应的值。

## A.13.3 有限状态机模型

图 A-49 中使用触发器和组合逻辑门的同步时序电路实现升值/降值计数器的具体例子很容易概括为图 A-51 中的标准有限状态机（finite state machine）模型。在这个模型中，延迟元件的时间延迟等于时钟周期的长度。这就是 $Y_i$ 发生变化到相应的 $y_i$ 发生变化的时间。模型假设组合逻辑块没有延迟；因此，输出 $z$、$Y_1$ 和 $Y_2$ 是输入 $x$、$y_1$ 和 $y_2$ 的瞬态函数。实际电路中的触发器会产生一些延迟，如图 A-50 所示。如果组合逻辑块的延迟与时钟周期相比很小的话，电路就能正常工作。下一状态的输出 $Y_i$ 必须及时到位以使触发器能在时钟周期末尾改变到下一所需状态。

520

组合逻辑块的输入由表示当前状态的触发器输出 $y_i$ 和外部输入 $x$ 组成，其输出是触发器输入 $Y_i$ 和外部输出 $z$。当有效时钟边沿到来结束当前时钟周期时，$Y_i$ 线的值被写入触发器，它成为状态变量 $y_i$ 的下一组值。因为这些信号连接在组合块的输入端上，所以它们和外部输入 $x$ 将产生新的 $z$ 和 $Y_i$ 值。一个时钟周期过去之后，新的 $Y_i$ 值被传送给 $y_i$，如此反复。换句话说，触发器形成了组合块从输出到输入的反馈回路，并引入了一个时钟周期的延迟。

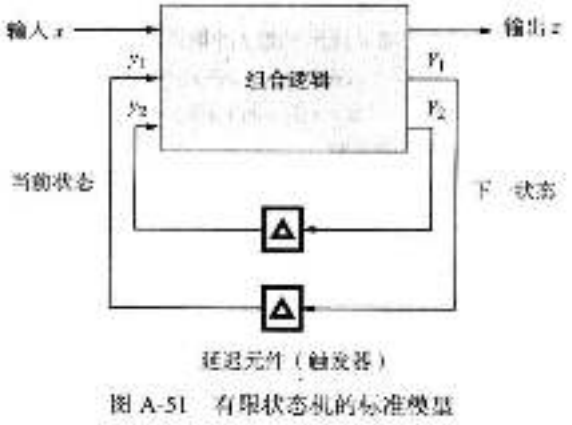



图 A-51 有限状态机的标准模型

尽管图 A-51 中只显示了一个外部输入、一个外部输出和两个状态变量，但是显而易见，这可以扩展成有多个输入、输出和状态变量的情况。

### A.13.4 有限状态机的组合

让我们总结一下设计具有图 A-51 基本结构的同步时序电路的步骤：

1）列出适当的状态图和状态表。

2）确定所需触发器的数目，选择合适类型的触发器。

521 3）确定状态图中的每个状态需要在触发器中存储的值。这称为状态分配。

4）列出状态赋值表。

5）推导出控制触发器输入的下一状态逻辑表达式。另外，也推导出电路输出的表达式。

6）使用得到的表达式实现电路。

时序电路可以很容易地用 CPLD 和 FPGA 实现，因为这些器件既包括触发器又包括组合逻辑门。现代的计算机辅助设计工具可以直接根据状态图的描述合成时序电路。

我们讨论的时序电路都是在同一时钟的控制下工作。不使用时钟也可以实现时序电路。这样的电路称为异步时序电路（asynchronous sequential circuit）。它们的设计不像同步时序电路这样直接。为了对两种时序电路有完整的了解，可以参考专门介绍逻辑设计的书籍 [1 – 7]。

## A.14 结束语

本附录的主要目的是使读者了解逻辑设计的基本概念和计算机体系结构中普遍使用的电路结构。熟悉了这些知识就可以更好地掌握在本书主要章节中介绍的体系结构概念。正如我们多次提到的，逻辑电路的详细设计需要借助 CAD 工具。这些工具考虑了许多细节问题，可以被经验丰富的设计人员有效地利用。

IC 技术和 CAD 工具的使用彻底改革了逻辑设计。市场上各种 IC 组件的成本不断降低，而且新的发现和技术的发展一直在进行。这个附录介绍了在数字系统设计中一些有用的基本组件。

## 习题

[E] A.1 使用积之和形式实现 COINCIDENCE 函数，其中 COINCIDENCE = $\overline{\text{XOR}}$。

[M] A.2 使用数学公式和真值表证明下列等式：

（a）$\overline{a \oplus b} \oplus c = \bar{a}\bar{b}\bar{c} + a\bar{b}c + \bar{a}bc + abc$

（b）$x + w\bar{x} = x + w$

522 （c）$x_1\bar{x}_2 + x_2\bar{x}_3 + x_3\bar{x}_1 = x_1\bar{x}_3 + x_2\bar{x}_1 + x_3\bar{x}_2$

[E] A.3 图 PA-1 给出了 4 个 3 变量函数 $f_1, f_2, f_3, f_4$ 的最小积之和形式。这些函数还有其他形式的最小表达式吗？如果有，将它们都写出来。

| $x_1$ | $x_2$ | $x_3$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | d | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | d |
| 1 | 0 | 0 | 1 | 0 | d | d |
| 1 | 0 | 1 | 0 | 0 | 0 | d |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |

图 PA-1 习题 A.3 的逻辑函数

[E] A.4 利用无关项 $d$ 找出函数 $f$ 的最简积之和形式，其中

$$f = x_1(x_2\bar{x}_3 + x_2x_4 - \bar{x}_2\bar{x}_3x_4) + x_2\bar{x}_4(\bar{x}_2 + x_1)$$

$$d = x_1\bar{x}_2(x_3x_4 + \bar{x}_3\bar{x}_4) + \bar{x}_1\bar{x}_3x_4$$

[M] A.5 考虑下面的函数

$$f(x_1, \cdots, x_4) = (x_1 \oplus x_3) \cdot (x_1x_2 + x_1\bar{x}_3)x_4 + x_2\bar{x}_3$$

（a）使用卡诺图找到 $f$ 的最小成本积之和（SOP）表达式。

（b）找到 $f$ 的互补项 $\bar{f}$ 的 SOP 表达式，然后（使用德摩根律）对这个 SOP 表达式取反，找到 $f$ 的表达式。得到的结果表达式应该是和之积形式（POS）。将它与（a）中所得的 SOP 表达式的成本进行比较。你可以从中得到什么结论？

[E] A.6 写出函数 $f(x_1, x_2, x_3, x_4)$ 的最小成本实现，其中如果一个或两个输入逻辑变量是 1 时，$f = 1$，否则 $f = 0$。

[M] A.7 对A-6定义了一个4位BCD码数字。设计一个电路，它有4个输入，标记为$b_3$，…，$b_0$和一个输出$f$。其中4位输入模式是有效BCD数时，$f=1$；否则$f=0$。给出这个电路的最小成本实现。 523

[M] A.8 两个2位数$A=a_1a_0$和$B=b_1b_0$由4变量函数$f(a_1,a_0,b_1,b_0)$进行比较。当满足

$$v(A)\leqslant v(B)$$

时，函数的值为1。其中对任意2位数，$v(X)=x_1\times 2^1+x_0\times 2^0$。假设变量$A$和$B$满足$|v(A)-v(B)|\leqslant 2$。使用最少的门实现$f$。

[M] A.9 重复习题A.8，要求当$v(A)=v(B)$时，函数$f=1$；其输入满足$v(A)+v(B)\leqslant 4$。

[E] A.10 证明结合率不适用于与非（NAND）操作符。

[M] A.11 使用与非门（不超过6个）实现下面的函数，每个门有二个输入。假设这二个输入的真值和反值都可用。

$$f=x_1x_2+x_1x_2x_3+x_1x_2x_3x_4+x_2x_3x_4$$

[M] A.12 使用6个或更少的2输入与非门实现下面函数。不能使用输入变量的反值。

$$f=x_1x_2+x_1+x_2x_3$$

[E] A.13 只使用与非门，尽可能经济地实现下面的函数。不允许使用输入变量的反值。

$$f=(x_1+x_2)(\bar{x}_2+\bar{x}_4)$$

[M] A.14 使用二进制表示的连续数字间只有1位不同的数字码称为Gray码。3位Gray码和二进制码变换的真值表如图PA-2a所示。

（a）只使用与非门实现函数$f_1,f_2,f_3$。

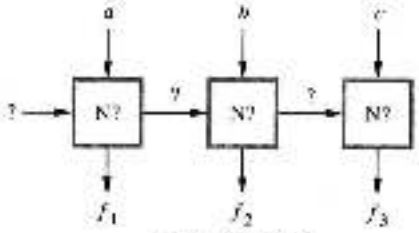
（b）注意到下列输入和输出变量的关系，可以使这一码制转换的网络成本更低。

$$f_1=a$$
$$f_2=f_1\oplus b$$
$$f_3=f_2\oplus c$$

使用这些关系，指明可重复的组合网络N的内容，如图PA-2b所示，从而实现转换。比较这种形式下实现转换所需的与非门数量与（a）部分中的与非门总数。

| 3位Gray码输入 | | | 二进制码输出 | | |
|---|---|---|---|---|---|
| $a$ | $b$ | $c$ | $f_1$ | $f_2$ | $f_3$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |

a）3位Gray码到二进制码的转换



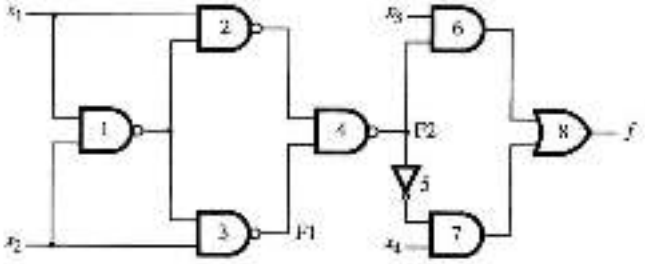

b）码制转换网络

图PA-2 习题A.14 Gray码转换的例子 524

[M] A.15 使用4个2输入与非门实现异或函数。

[M] A.16 图A-36定义了一个BCD码七段显示器译码器。使用与、或、非门给出该真值表的实现。证明其与图中的与非门电路实现了同样的功能。

[M] A.17 在图PA-3的逻辑网络中，门3出现故障，不管输入为何值，其输出F1都产生逻辑值1。重新画出网络图，尽可能化简，用最少的门实现一个与原始有故障网络等价的新网络。假设错误出在F2，它被锁定在逻辑值0，重复此问题。




图PA-3 一个有故障的网络

[M] A.18 图 A-16 显示了普通 CMOS 电路的结构。设计一个 CMOS 电路实现下列函数。

$$f(x_1, \cdots, x_4) = x_1\bar{x}_2 + \bar{x}_3\bar{x}_4$$

尽可能少用晶体管。(提示：考虑晶体管的串联/并联网络。注意图 A-17 和图 A-18 中上拉和下拉网络相反的串联和并联结构。)

525

[E] A.19 画出图 A-30 中 JK 电路输出 Q 的波形，利用图 PA-4 的输入波形并假设触发器开始处于 0 状态。

[E] A.20 写出图 PA-5 中与非门电路的真值表。将其与图 A-23b 中的真值表比较，然后证明图 A-25 中的电路与图 A-24a 中的电路等价。

[M] A.21 以或非门的延迟为单位，计算图 A-28 中下降沿触发的 D 触发器的建立时间和保持时间。

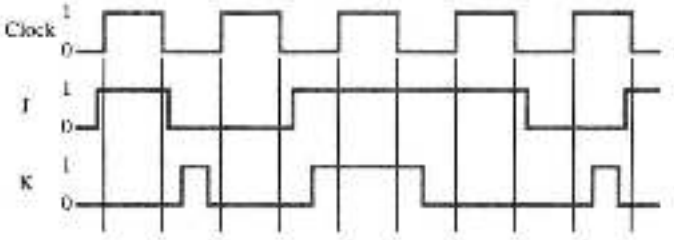



图 PA-4 JK 触发器的输入波形

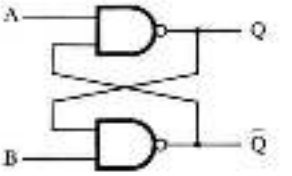



图 PA-5 与非锁存器

[M] A.22 在图 A-26a 的电路中，用或非门代替所有的与非门。写出结果电路的真值表。这个电路与图 A-26a 中的电路有何不同？

[M] A.23 图 A-32 显示了一个在时钟信号控制下每次将数据右移一位的移位寄存器网络。改进这个移位寄存器，使它能够在时钟和附加控制输入 ONE/TWO 的共同作用下每次将数据移动一位或两位。

[D] A.24 我们需要一个 4 位移位寄存器，它有两个控制输入端——INITIALIZE 和 RIGHT/LEFT。当 INITIALIZE 置为 1 时，二进制数 1000 被写入寄存器，而与时钟输入无关。当 INITIALIZE = 0 时，时钟输入的脉冲将这个模式循环移位。当 RIGHT/LEFT 的输入等于 1 或 0 时，这个模式分别循环右移或循环左移。使用图 A-31 中具有预置和清除输入的 D 触发器，给出这个寄存器的合理设计。

[M] A.25 构造一个 3 输入 8 输出的译码器网络，所使用的门输入不得超过 2 个。

[D] A.26 图 A-34 显示了一个 3 位升值计数器。按相反顺序（即 7, 6, …, 1, 0, 7, …）计数的计数器称为降值计数器。能在 UP/DOWN 信号的控制下按两种顺序计数的计数器叫做升值/降值计数器。画出一个 3 位升值/降值计数器的逻辑图，它能从外部逐步行加载触发器，从而可以被预置为任何状态。LOAD/COUNT 控制用来决定计数器是被加载还是做计数操作。

[D] A.27 图 A-34 显示了一个异步的 3 位升值计数器。设计一个 4 位同步升值计数器，按 0, 1, 2, …, 15, 0, …的顺序计数。在电路中使用 T 触发器。在同步计数器中，所有的触发器都能同时改变它们的状态，因此，主时钟输入应直接与所有触发器的时钟输入相连。

[M] A.28 实现如下表达式描述的逻辑函数

$$f(x_1, x_2, x_3, x_4) = x_1x_2\bar{x}_4 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_2\bar{x}_3\bar{x}_4$$

(a) 使用 8 输入多路复用器电路实现 $f$。

(b) 可以用 4 输入多路复用器实现 $f$ 吗？如果可以，请写出方法。

[M] A.29 当

$$f(x_1, x_2, x_3, x_4) = x_1\bar{x}_2x_3 + x_2x_3x_4 + \bar{x}_1\bar{x}_4$$

时重复习题 A.28。

526
|
527

[E] A.30 使用状态分配 S0 = 10，S1 = 11，S2 = 01，S3 = 00 完成对图 A-46 中升值/降值计数器的设计。这个设计与 A.13.1 节的有什么不同？

[M] A.31 使用 D 触发器设计一个如图 A-49 所示形式的 2 位同步计数器。按照…0, 3, 1, 2, 0, …的顺序计数。这个电路没有外部输入，输出是触发器自己的值。

[M] A.32 重复习题 A.31，设计一个按照…0, 1, 2, 3, 4, 5, 0, …计数的 3 位计数器。设计组合逻辑时，将未使用的计数值 6 和 7 作为无关项。

[M] A.33 有限状态机可以用来检测输入到机器的二进制序列中某些子序列的发生情况。这样的机器称为有限状态识别器（finite state recognizer）。假设每当出现输入模式 011，机器就产生输出 1。

（a）画出这台机器的状态图。

（b）假设使用 D 触发器，为所需数目的触发器进行状态分配并构造状态表。

（c）写出输出变量和下一状态变量的逻辑表达式。

[M] A.34 机器在输入序列中识别子序列 011 和 010，包括重叠出现的情况。重复习题 A.33 的（a）部分。例如，当输入序列是 1101010110…时产生的输出序列是 0000010101 0…。

## 参考文献


1. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3rd ed., McGraw-Hill, Burr Ridge, IL, 2009.
2. M.M. Mano and M.D. Ciletti, *Digital Design*, 4th ed., Prentice-Hall, Upper Saddle River, NJ, 2007.
3. J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed., Prentice-Hall, Englewood Cliffs, NJ, 2005.
4. R.H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed., Pearson Prentice-Hall, Upper Saddle River, NJ, 2005.
5. C.H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., Thomson/Brooks/Cole, Belmont, Ca., 2004.
6. D.D. Gajski, *Principles of Digital Design*, Prentice-Hall, Upper Saddle River, NJ, 1997.
7. J.P. Hayes, *Digital Logic Design*, Addison-Wesley, Reading, Mass., 1993.
8. A.S. Sedra and K.C. Smith, *Microelectronic Circuits*, 6th ed., Oxford, New York, 2009.

附录 B

Computer Organization and Embedded Systems, Sixth Edition

# Altera Nios II 处理器

## 附录目标

在本附录中，你将学习 Altera Nios II 处理器，它具有 RISC 风格的指令集。本附录主要讨论：

- 指令集体系结构
- 输入 / 输出能力
- 支持嵌入式应用

[529]

在第 2 和第 3 章中，我们主要从程序员的角度介绍了指令集设计中所使用的基本概念。在本附录中，我们将研究 Altera 公司的 Nios II 处理器，它是能体现前面所讨论概念的 RISC 风格商业产品的一个例子。本附录的讨论与第 2 和第 3 章所介绍的主题非常相似。

Nios II 处理器是在现场可编程门阵列（FPGA）器件（参见附录 A）中实现的。它以软件的形式提供，通过使用 Altera 公司提供的 Quartus II CAD（计算机辅助设计）工具可以很容易地将其合并到计算机系统中。然后，将所设计的系统下载到 FPGA 器件中，这样就可以实现一个具有典型功能的计算机系统。

Quartus II 软件包括一个称为 SOPC Builder 的工具，该工具可用来设计 SOPC 系统。SOPC Builder 提供了多种预先设计的模块，称为 IP 内核（IP core），它们可以很容易地添加到一个系统中。这些模块包括 Nios II 处理器、各种 I/O 接口和存储器控制器。这些模块的特点是有各种参数，以允许设计自定义系统的用户指定所需系统的确切性质。Nios II 处理器可以配置为具有多种不同的特性，每种特性需要不同数量的逻辑电路来实现，从而会影响最终系统的性能和成本。由于用户可以自定义微控电路的设计，所以我们说 Nios II 是个软处理器（soft processor）。

设计一个自定义的、可在单个 FPGA 芯片上实现的计算机系统的能力，在嵌入式应用中是很有吸引力的。我们在第 11 章中讨论了此类应用。

## B.1 Nios II 的特征

Nios II 处理器具有 RISC 风格的体系结构，其特点大体上与第 2 章中所描述的非常相似。

### 1. 数据大小

字长为 32 位。数据是以 32 位的字（word）、16 位的半字（halfword），或者 8 位的字节（byte）为单位进行处理的。一个字中的字节地址是按小端方式进行分配的，即低字节地址用于最低有效位的字节。

### 2. 存储器访问

存储器中的数据只能通过 Load 和 Store 指令进行访问。它们将数据装入通用寄存器中或者将寄存器中的数据进行存储。Load 和 Store 指令可以以字、半字或字节为单位进行数据传输。

### 3. 寄存器

Nios II 有 32 个通用寄存器和一些控制寄存器。所有寄存器的长度都是 32 位的。

### 4. 指令

[530] 所有指令的长度都是 32 位的。它们具有第 2 章介绍的所有 RISC 风格功能。

## B.2 通用寄存器

表 B-1 显示了处理器的 32 个通用寄存器，它们被称为 r0 到 r31。这是在汇编语言指令中使用的名称。一些寄存器用于特定的用途，因此给它们赋予更能指示其功能的名称，如表 B-1 所示。这些名称也可由汇编程序识别。用于特定用途的寄存器有：

- r0 始终包含常数 0。读取该寄存器将返回值 0；写入无效。
- r1 被汇编程序用作临时寄存器。不应该在用户程序中使用它。
- r24 和 r29 用于异常处理。
- r25 和 r30 被一个称为 JTAG 调试模块的调试工具专门使用。
- r26 是全局指针，指向用户程序中的数据。
- r27 是处理器堆栈指针。
- r28 是结构指针。
- r31 保存子程序调用时的返回地址。

表 B-1 Nios II 通用寄存器

| 寄存器 | 名 称 | 功 能 |
|---|---|---|
| r0 | zero | 0x00000000 |
| r1 | at | 汇编程序暂时寄存器 |
| r2 | | |
| r3 | | |
| ⋮ | ⋮ | ⋮ |
| r23 | | |
| r24 | et | 异常临时寄存器 |
| r25 | bt | 断点临时寄存器 |
| r26 | gp | 全局指针 |
| r27 | sp | 堆栈指针 |
| r28 | fp | 结构指针 |
| r29 | ea | 异常返回地址 |
| r30 | ba | 断点返回地址 |
| r31 | ra | 返回地址 |

其他寄存器用于一般用途。

531

因为寄存器 r0 始终包含值 0，所以，不管是否需要值 0，它都可以作为一个操作数包含在指令中。利用这个特点可以对其进行活用。例如，指令

```
add    r5, r0, r0
```

可以用来将寄存器 r5 清 0。同样，r0 可以是 Store 指令中的源操作数，用来将一个存储单元清 0。在 Compare（比较）与 Branch（转移）指令中，当要将另一个寄存器中的值与 0 进行比较时可以使用 r0。它也可以用于变址寻址方式，以提供绝对寻址方式的限制版本。

Nios II 处理器还有一些控制寄存器。我们将在 B.9 节中讨论这些寄存器，因为它们主要用于输入/输出传输。

## B.3 寻址方式

Nios II 处理器支持以下五种寻址方式：

- *立即方式*（immediate mode）——在指令中明确地给出一个 16 位的操作数，该操作数的值被符号扩展为 32 位，用在执行算术运算的指令中。
- *寄存器方式*（register mod）——操作数是一个通用寄存器的内容。
- *寄存器间接方式*（register indirect mode）——操作数的有效地址是一个寄存器的内容。
- *位移方式*（displacement mode）——操作数的有效地址是通过将一个寄存器的内容和指令中给出的一个 16 位有符号位移值相加得到的。这就是在第 2 章中讨论的变址方式。
- *绝对方式*（absolute mode）——通过使用位移方式以及寄存器 r0 来指定一个 16 位的操作数绝对地址。

表 B-2 给出了寻址方式及其汇编语法。注意，这里立即方式的语法与第 2 章中给出的立即方式不同，因为这里不使用数字符号（#）。相反的是，它会将立即数就地包含于操作码助记符中。例如，指令

```
addi   r3, r2, 24
```

将 r2 的内容与十进制立即值 24 相加，并将结果放到 r3 中。

我们还观察到，立即方式与绝对方式只有在立即数或者地址能用 16 位来表示时才可以使用。我们将在 B.4.4 节和 B.4.5 节中分别讨论 32 位立即数和地址的问题。

532

表 B-2 Nios II 寻址方式

| 名 称 | 汇编语法 | 寻址功能 |
|---|---|---|
| 立即方式 | 值 | 操作数 = 值 |
| 寄存器方式 | ri | EA = ri |
| 寄存器间接方式 | (ri) | EA = [ri] |
| 位移方式 | X(ri) | EA = [ri] + X |
| 绝对方式 | LOC(r0) | EA = LOC |

EA= 有效地址

值 = 一个 16 位有符号数

X= 一个 16 位的有符号位移值

## B.4 指令

Nios II 指令集举例说明了第 2 章讨论的 RISC 风格处理器的所有特点。所有指令的长度都是 32 位的。算术和逻辑运算只能对通用寄存器中的操作数进行操作。Load 和 Store 指令用来在存储器和寄存器之间传输数据。

指令有三种基本形式。那些指定三个寄存器操作数的指令具有以下形式：

```
Operation   dest_register, source1_register, source2_register
```

带有立即类操作数的指令形式为：

```
Operation   dest_register, source_register, immediate_operand
```

其中立即操作数的长度为 16 位，可以对其进行符号扩展以提供一个 32 位的操作数。第三种形式包括一个 26 位的无符号立即值，它只用于子程序调用指令中，如：

```
call   LABEL
```

### B.4.1 标记符号

在汇编语言程序中使用的标记符号（notation）受一个特殊的汇编程序的限制，该汇编程序用来将源程序汇编成处理器能够执行的机器代码。在许多汇编程序中，都假设源程序中的语句不区分大小写。这意味着语句

533

```
ADD   R2, R3, R4
```

和

```
add   r2, r3, r4
```

是等价的。但是 Altera 公司提供的 Nios II 汇编程序并不是这样。该汇编程序允许操作码助记符不区分大小写，但要求寄存器的名称为小写。因此，寄存器必须由表 B-1 中给出的名称来识别。例如，我们可以使用 r27 或 sp 来表示堆栈指针，但不能使用 R2 或 SP。

在 Altera 公司的文档资料中，小写字母可以用来指定操作码助记符。为了使用户更容易地查阅 Altera 公司的文献，我们将在本附录中使用相同的约定。

Nios II 指令集是相当广泛的。本附录中，我们将只介绍一个子集，它足以开发一个可理解的 Nios II 处理器。为了使后面的介绍更易于理解，我们将根据其功能分组讨论。我们还将展示如何使用这些指令来实现第 2 和第 3 章中的程序示例。对于指令集的完整描述，读者可以查阅 Nios II 处理器参考手册，这可以在 Altera 公司网站（altera.com）的文献部分中找到。

### B.4.2 Load 和 Store 指令

Load 和 Store 指令在存储器或 I/O 接口和通用寄存器之间传送数据。Load Word 指令的一般形式为

```
ldw    ri, source_operand
```

例如，指令

```
ldw    r2, 40(r3)
```

使用位移寻址方式，通过将十进制数 40 和寄存器 r3 的内容相加来确定存储单元的有效地址，然后它从存储器中读出这个 32 位的操作数将其装入 r2 中。有效地址必须是按字对齐的，这意味着它必须是 4 的倍数。

Store Word 指令的形式为

```
stw    ri, destination_operand
```

例如，指令

```
stw    r2, 40(r3)
```

将 r2 的内容保存到上述的同一存储单元中。

汇编语法要求所有 Load 和 Store 指令中的存储器操作数使用位移寻址方式 X(ri) 来指定。这就允许使用寄存器间接方式 0(ri)，简写为 (ri)，以及绝对方式 X(r0)。但是，即使标签的值 [534]
可在一条汇编指示（assembler directive）中定义，绝对方式也不能只使用一个标签来指定。因此，语句

```
ldw    r2, LOCATION
```

将会导致一个语法错误。

除了字大小的操作数，Load 和 Store 指令也可以处理字节大小和半字大小的操作数。操作数的大小在操作码助记符中指示。这样的 Load 指令有：

- ldb（Load Byte，加载字节）
- ldbu（Load Byte Unsigned，加载无符号字节）
- ldh（Load Halfword，加载半字）
- ldhu（Load Halfword Unsigned，加载无符号半字）

当一个较短的操作数被装入一个 32 位的寄存器中时，必须将它的值进行调整以适应寄存器的大小。这是通过 ldb 和 ldh 指令将 8 位或 16 位的值符号扩展为 32 位来完成的。在 ldbu 和 ldhu 指令中，操作数是用零扩展的，因为其值表示一个正整数。

相对应的 Store 指令有：

- stb（Store Byte，存储字节），将源寄存器的低位字节存储到有效地址所指定的存储器字节中。
- sth（Store Half Word，存储半字），将源寄存器的低位半字存储到有效地址（必须是半字对齐的）所指定的存储器半字中。

每一条 Load 或 Store 指令还有可以访问 I/O 接口中的单元的版本。这些指令是：

- ldwio（Load Word I/O，加载 I/O 接口中的字）
- ldbio（Load Byte I/O，加载 I/O 接口中的字节）
- ldbuio（Load Byte Unsigned I/O，加载 I/O 接口中的无符号字节）
- ldhio（Load Halfword I/O，加载 I/O 接口中的半字）
- ldhuio（Load Halfword Unsigned I/O，加载 I/O 接口中的无符号半字）
- stwio（StoreWord I/O，向 I/O 接口存储字）
- sthio（Store Byte I/O，向 I/O 接口存储字节）
- sthio（Store Halfword I/O，向 I/O 接口存储半字）

当 Nios II 处理器和一个高速缓存（第 8 章中讨论的概念）一起使用时，便需要用到 Load 和 Store 指令的 I/O 版本。高速缓存是一个相对较小但访问速度比主存储器快的存储器。通常

将主存储器中最近使用的指令和数据装入高速缓存中，这样，当再次需要它们的时候便可以更快速地访问它们。这对于通常可以在主存储器中找到的指令和数据来说是非常有利的。但是，如果一个用来访问特定数据的地址指向的是一个存储器映射 I/O 接口时，这种方法是不合
[535] 适的，因为 I/O 接口中的输入数据可随时变化，并且通常必须将输出数据直接发送给 I/O 设备。Load 和 Store 指令的 I/O 版本绕过高速缓存（如果有高速缓存的话），始终访问 I/O 单元。

### B.4.3 算术指令

算术指令是对通用寄存器中的数据或者指令中的立即值数据进行操作的，包括以下指令：

- add（Add Registers，寄存器加）
- addi（Add Immediate，立即数加）
- sub（Subtract Registers，寄存器减）
- subi（Subtract Immediate，立即数减）
- mul（Multiply，乘）
- muli（Multiply Immediate，立即数乘）
- div（Divide，除）
- divu（Divide Unsigned，无符号除）

Add 指令

add ri, rj, rk

将寄存器 rj 和 rk 中的内容相加，并将和放到寄存器 ri 中。

Add Immediate 指令

addi ri, rj, 85

将寄存器 rj 中的内容与立即值 85 相加，并将其结果放到寄存器 ri 中。指令中的立即操作数用 16 位表示，在加法运算之前将其符号扩展为 32 位。

Subtract 指令

sub ri, rj, rk

将寄存器 rj 中的内容减去寄存器 rk 中的内容，并将结果放到寄存器 ri 中。

Multiply 指令

mul ri, rj, rk

将寄存器 rj 和 rk 中的内容相乘，并将乘积的低 32 位放到寄存器 ri 中。乘法运算将操作数作为无符号数处理。不管操作数是无符号还是有符号的，如果所生成的乘积可以用 32 位表示，那么寄存器 ri 中的结果就是正确的。Multiply 指令的立即数版本为

muli ri, rj, Value16

[536] 其中的 16 位立即操作数 Value16 被符号扩展为 32 位。

Divide 指令

div ri, rj, rk

将寄存器 rj 中的内容除以寄存器 rk 中的内容，并把商的整数部分放到寄存器 ri 中。将操作数视为有符号整数。divu 指令以同样的方式执行，但其操作数视为无符号整数。

### B.4.4 逻辑指令

逻辑指令提供 AND（与）、OR（或）、XOR（异或）和 NOR（或非）运算。它们对通用寄存器中的数据或者指令中的立即值数据进行运算。

AND 指令

and　ri, rj, rk

对寄存器 rj 和 rk 中的内容进行按位逻辑与运算，并将结果保存在寄存器 ri 中。同样，指令 or、xor 和 nor 分别执行或、异或和或非运算。

AND Immediate 指令

andi　ri, rj, Value16

对寄存器 rj 中的内容和 16 位立即操作数 Value16（用零扩展到 32 位）执行按位逻辑与运算，并将结果保存到寄存器 ri 中。同样，指令 ori、xori 和 nori 使用立即操作数来分别执行或、异或和或非运算。

还可以将 16 位的立即操作数用作逻辑运算中的高 16 位，在这种情况下操作数的低 16 位是零，可通过如下指令来实现：

- andhi（AND High Immediate，立即数高位与）
- orhi（OR High Immediate，立即数高位或）
- xorhi（XOR High Immediate，立即数高位异或）

这提供了一种将 32 位的立即值装入寄存器中的机制：首先使用 orhi 指令将高 16 位放到寄存器中，然后使用 ori 指令在低 16 位进行或操作。

### B.4.5　Move 指令

Move（传送）指令将一个寄存器的内容复制到另一个寄存器中，或者将一个立即值放到一个寄存器中。这类指令是伪指令（pseudoinstruction），为程序员提供方便，而汇编程序使用其他指令来实现伪指令。指令 537

mov　ri, rj

将寄存器 rj 中的内容复制到寄存器 ri 中。该指令可以实现为

add　ri, rj, r0

Move Immediate（传送立即数）指令

movi　ri, Value16

将 16 位立即数 Value16 符号扩展为 32 位，并将它装入寄存器 ri 中。该指令可以实现为

addi　ri, r0, Value16

Move Unsigned Immediate（传送无符号立即数）指令

movui　ri, Value16

将 16 位立即数 Value16 用零扩展为 32 位，并将它装入寄存器 ri 中。该指令可以实现为

ori　ri, r0, Value16

Move Immediate Address（传送立即数地址）指令

movia　ri, LABEL

将地址 LABEL 所对应的 32 位值装入寄存器 ri 中。汇编程序可通过下面两条指令来实现该指令：

orhi　ri, r0, LABEL_HIGH
ori　ri, ri, LABEL_LOW

其中 LABEL_HIGH 和 LABEL_LOW 是 LABEL 的高 16 位和低 16 位。

### B.4.6 Branch 与 Jump 指令

使用 Branch（转移）或 Jump（跳转）指令可以改变程序的执行流程。Unconditional Branch（无条件转移）指令

br LABEL

无条件地转去执行地址 LABEL 处的指令。转移目标以在指令中包含一个 16 位的有符号偏移量的形式来指定。偏移量是紧跟在 br 之后的那条指令到地址 LABEL 之间的距离（以字节为
538 单位）。

使用 Conditional Branch（条件转移）指令可实现程序执行的条件转移，这些指令比较两个通用寄存器中的内容，如果结果满足转移条件，则产生一个转移。例如，Branch if Less Than Signed（如果小于有符号数则转移）指令

blt *ri*, *rj*, LABEL

执行比较操作 [*ri*] < [*rj*]，它将寄存器中的内容视为有符号数。

Branch if Less Than Unsigned（如果小于无符号数则转移）指令

bltu *ri*, *rj*, LABEL

执行比较操作 [*ri*] < [*rj*]，它将寄存器中的内容视为无符号数。

同样形式的其他条件转移指令还有：

- beq（Comparison [*ri*] = [*rj*]，比较 [*ri*] = [*rj*]）
- bne（Comparison [*ri*] ≠ [*rj*]，比较 [*ri*] ≠ [*rj*]）
- bge（Signed comparison [*ri*] ≥ [*rj*]，有符号比较 [*ri*] ≥ [*rj*]）
- bgeu（Unsigned comparison [*ri*] ≥ [*rj*]，无符号比较 [*ri*] ≥ [*rj*]）
- bgt（Signed comparison [*ri*] > [*rj*]，有符号比较 [*ri*] > [*rj*]）
- bgtu（Unsigned comparison [*ri*] > [*rj*]，无符号比较 [*ri*] > [*rj*]）
- ble（Signed comparison [*ri*] ≤ [*rj*]，有符号比较 [*ri*] ≤ [*rj*]）
- bleu（Unsigned comparison [*ri*] ≤ [*rj*]，无符号比较 [*ri*] ≤ [*rj*]）

任何转移指令的目标必须是在一个 16 位的偏移量可以指定的范围之内。对于此范围之外的目标，需要使用 Jump 指令

jmp *ri*

它无条件地转移到指定的寄存器 *ri* 中所包含的地址处执行。

**例 B.1**

为了说明 Nios II 指令的使用，我们考虑一下图 2-8 中将一个列表中的数字相加的程序。图 B-1 给出了这个程序的 Nios II 版本。这两个程序使用了相同的寄存器。使用绝对寻址方式将列表的大小从存储单元 N 装入寄存器 r2 中。假设地址 N 可以用 16 位表示，这是因为绝对寻址方式实际上是由位移方式实现的，而位移方式是使用一个 16 位的偏移量加上 r0 的内容来确定操作数的有效地址的。使用 Add 指令将寄存器 r0 中的 0 与寄存器 r3 中的内容相加来将寄存器 r3 清零。将地址 NUM1 指定为 Add Immediate 指令中的一个立即操作数来将其装入 r4 中。我们观察到，当需要指定一个立即操作数时，我们可以通过简单地给出它的名称（将由汇
539 编程序识别）或实际值来实现。操作码 addi 指明它是一个立即操作数。如果寄存器 r2 的内容大于零，那么条件转移指令将使得程序在 LOOP 处继续执行。最后，请注意标签 LOOP 后面必须跟有一个冒号，注释是由 "/*" 和 "*/" 字符来界定的。

```
        ldw   r2, N(r0)       /* 加载列表的大小 */
        add   r3, r0, r0      /* 将和初始化为 0*/
        addi  r4, r0, NUM1    /* 加载第一个数的地址 */
LOOP:   ldw   r5, (r4)        /* 获取下一个数 */
        add   r3, r3, r5      /* 把这个数加到和中 */
        addi  r4, r4, 4       /* 递增指向列表的指针 */
        subi  r2, r2, 1       /* 递减计数器 */
        bgt   r2, r0, LOOP    /* 如果没完成就循环回到前面 */
        stw   r3, SUM(r0)     /* 保存最终的和 */
```

图 B-1 图 2-8 中程序的 Nios II 实现

**例 B.2**

在图 B-1 的程序中，标签 N、NUM1 和 SUM 所对应的地址必须足够小以使其可以用 16 位来表示。如果不是这样的话，那么程序可以像图 B-2 所示的那样进行改进。这里，movia 指令用来将 32 位的地址装入寄存器中，mov 指令用来将 r3 清零。

```
        movia r2, N           /* 获取地址 N*/
        ldw   r2, (r2)        /* 加载列表的大小 */
        mov   r3, r0          /* 将和初始化为 0*/
        movia r4, NUM1        /* 加载第一个数的地址 */
LOOP:   ldw   r5, (r4)        /* 获取下一个数 */
        add   r3, r3, r5      /* 把这个数加到和中 */
        addi  r4, r4, 4       /* 递增指向列表的指针 */
        subi  r2, r2, 1       /* 递减计数器 */
        bgt   r2, r0, LOOP    /* 如果没完成就循环回到前面 */
        movia r6, SUM         /* 获取地址 SUM*/
        stw   r3, (r6)        /* 保存最终的和 */
```

图 B-2 图 2-8 中程序的一种更普遍的 Nios II 实现

**例 B.3**

图 2-11 中的程序将学生在不同测验中所取得的分数相加，图 B-3 给出了该程序的一种实现。 [540]

```
        movia r2, LIST        /* 获取地址 LIST*/
        mov   r3, r0          /* 将 r3 清零 */
        mov   r4, r0          /* 将 r4 清零 */
        mov   r5, r0          /* 将 r5 清零 */
        movia r6, N           /* 获取地址 N*/
        ldw   r6, (r6)        /* 加载 n 值 */
LOOP:   ldw   r7, 4(r2)       /* 将下一个学生的测验 1*/
        add   r3, r3, r7      /* 成绩加到部分和中 */
        ldw   r7, 8(r2)       /* 将这个学生的测验 2*/
        add   r4, r4, r7      /* 成绩加到部分和中 */
        ldw   r7, 12(r2)      /* 将这个学生的测验 3*/
        add   r5, r5, r7      /* 成绩加到部分和中 */
        addi  r2, r2, 16      /* 递增指针 */
        subi  r6, r6, 1       /* 递减计数器 */
        bgt   r6, r0, LOOP    /* 如果没完成就循环回到前面 */
        movia r7, SUM1        /* 将测验 1 的总和 */
        stw   r3, (r7)        /* 保存到单元 SUM1 中 */
        movia r7, SUM2        /* 将测验 2 的总和 */
        stw   r4, (r7)        /* 保存到单元 SUM2 中 */
        movia r7, SUM3        /* 将测验 3 的总和 */
        stw   r5, (r7)        /* 保存到单元 SUM3 中 */
```

图 B-3 图 2-11 中程序的实现

### B.4.7 子程序链接指令

Nios II 有两条调用子程序的指令。Call Subroutine（子程序调用）指令

```
call    LABEL
```

包括一个 26 位的无符号立即数。该指令将返回地址（下一条指令的地址）保存在寄存器 r31 中。然后，它把控制权转交给地址 LABEL 处的指令。该跳转地址是通过将程序计数器的高 4 位与立即值 Value26 和两个低位的 0 连接起来确定的，如下：

$$\text{Jump address} = PC_{31\text{-}28} : \text{Value26} : 00$$

因为 Nios II 指令必须是字对齐的，所以两位最低有效位是 0。

Call Subroutine in Register（通过寄存器调用子程序）指令

541

```
callr   ri
```

将返回地址保存在寄存器 r31 中，然后将控制权转交给其地址包含在寄存器 ri 中的指令。

从子程序返回是通过如下指令

```
ret
```

完成的。该指令将使得程序转移到寄存器 r31 所包含的地址处执行。

**例 B.4**

图 B-4 说明了如何将图 B-2 中的程序写成一个子程序的形式，其中的参数是通过处理器寄存器传递的。

```
调用程序
        movia   r2, N           /* 获取地址 N*/
        ldw     r2, (r2)        /* 加载列表的大小 */
        movia   r4, NUM1        /* 加载第一个数的地址 */
        call    LISTADD         /* 调用子程序 */
        movia   r6, SUM         /* 获取地址 SUM*/
        stw     r3, (r6)        /* 保存最终的和 */
        :

子程序
LISTADD: mov    r3, r0          /* 将和初始化为 0*/
LOOP:   ldw     r5, (r4)        /* 获取下一个数 */
        add     r3, r3, r5      /* 把这个数加到和中 */
        addi    r4, r4, 4       /* 递增指向列表的指针 */
        subi    r2, r2, 1       /* 递减计数器 */
        bgt     r2, r0, LOOP    /* 如果没完成就循环回到前面 */
        ret                     /* 返回到调用程序 */
```

图 B-4 将图 B-2 中的程序写成一个子程序；参数通过寄存器传递

**例 B.5**

图 B-5 显示了如何将图 B-2 中的程序写成一个子程序，其中参数是通过处理器堆栈来传

542 递的。

**例 B.6**

当 Subroutine Call（子程序调用）指令执行时，Nios II 处理器将返回地址保存在寄存器 r31（ra）中。在嵌套子程序中，该返回地址必须在调用第二个子程序之前保存在处理器堆栈中。图 B-6 显示了如何实现嵌套子程序，它对应于图 2-21 中的程序。

```
            movia   r2, NUM1      /* 将参数压入栈中 */
            subi    sp, sp, 4
            stw     r2, (sp)
            movia   r2, N
            ldw     r2, (r2)
            subi    sp, sp, 4
            stw     r2, (sp)
            call    LISTADD       /* 调用子程序 */
            ldw     r2, 4(sp)     /* 从栈中获取结果，*/
            movia   r3, SUM       /* 并将其保存到单元 SUM 中 */
            stw     r2, (r3)
            addi    sp, sp, 8     /* 恢复栈顶 */
            .
            .
LISTADD:    subi    sp, sp, 16    /* 保存寄存器 */
            stw     r2, 12(sp)
            stw     r3, 8(sp)
            stw     r4, 4(sp)
            stw     r5, (sp)
            ldw     r2, 16(sp)    /* 将计数器初始化为 n*/
            ldw     r4, 20(sp)    /* 初始化指向列表的指针 */
            mov     r3, r0        /* 将和初始化为 0*/
LOOP:       ldw     r5, (r4)      /* 获取下一个数 */
            add     r3, r3, r5    /* 把这个数加到和中 */
            addi    r4, r4, 4     /* 将指针增加 4*/
            subi    r2, r2, 1     /* 递减计数器 */
            bgt     r2, r0, LOOP  /* 如果没完成就循环回到前面 */
            stw     r3, 20(sp)    /* 将结果压入栈中 */
            ldw     r5, (sp)      /* 恢复寄存器 */
            ldw     r4, 4(sp)
            ldw     r3, 8(sp)
            ldw     r2, 12(sp)
            addi    sp, sp, 16
            ret                   /* 返回到调用程序 */
```

图 B-5 将图 B-2 中的程序写成一个子程序；参数通过堆栈传递 543

```
            movia   r2, PARAM2    /* 将参数放置到栈中 */
            ldw     r2, (r2)
            subi    sp, sp, 4
            stw     r2, (sp)
            movia   r2, PARAM1
            ldw     r2, (r2)
            subi    sp, sp, 4
            stw     r2, (sp)
            call    SUB1          /* 调用子程序 */
            ldw     r2, (sp)      /* 从栈中获取结果，*/
            movia   r3, RESULT    /* 并将其保存到单元 RESULT 中 */
            stw     r2, (r3)
            addi    sp, sp, 8     /* 恢复栈顶 */
            .
            .
SUB1:       subi    sp, sp, 24    /* 保存寄存器 */
            stw     ra, 20(sp)
            stw     fp, 16(sp)
            stw     r2, 12(sp)
            stw     r3, 8(sp)
            stw     r4, 4(sp)
            stw     r5, (sp)
```

图 B-6 嵌套子程序：图 2-12 中程序的实现

```
        addi    fp, sp, 16      /* 初始化结构指针 */
        ldw     r2, 8(fp)       /* 获取第一个参数 */
        ldw     r3, 12(fp)      /* 获取第二个参数 */
        .
        movia   r5, PARAM3      /* 获取必须传递给 */
        ldw     r4, (r5)        /* SUB2 的参数，*/
        subi    sp, sp, 4       /* 并将其压入栈中 */
        stw     r4, (sp)
        call    SUB2
        ldw     r4, (sp)        /* 从 SUB2 获取结果 */
        addi    sp, sp, 4
        .
        stw     r5, 8(fp)       /* 将答案放入栈中 */
        ldw     r5, (sp)        /* 恢复寄存器 */
        ldw     r4, 4(sp)
        ldw     r3, 8(sp)
        ldw     r2, 12(sp)
        ldw     fp, 16(sp)
        ldw     ra, 20(sp)
        addi    sp, sp, 24
        ret                     /* 返回到主程序 */

SUB2:   subi    sp, sp, 12      /* 保存寄存器 */
        stw     fp, 8(sp)
        stw     r2, 4(sp)
        stw     r3, (sp)
        addi    fp, sp, 8       /* 初始化结构指针 */
        ldw     r2, 4(fp)       /* 获取参数 */
        .
        stw     r3, 4(fp)       /* 将 SUB2 的结果放入栈中 */
        ldw     r3, (sp)        /* 恢复寄存器 */
        ldw     r2, 4(sp)
        ldw     fp, 8(sp)
        addi    sp, sp, 12
        ret                     /* 返回到 SUB1*/
```

图 B-6 （续）

## B.4.8 Comparison 指令

Comparison（比较）指令比较两个寄存器的内容或者将一个寄存器的内容和一个立即值进行比较，并将 1（如果为真）或 0（如果为假）写入结果寄存器。

Compare Less Than Signed（有符号数的小于比较）指令

```
cmplt   ri, rj, rk
```

将寄存器 $rj$ 和 $rk$ 中的有符号数进行 [$rj$] < [$rk$] 的比较操作，如果结果为真，就将 1 写入寄存器 $ri$ 中，否则，就写入 0。

Compare Less Than Unsigned（无符号数的小于比较）指令

```
cmpltu  ri, rj, rk
```

与 cmplt 指令执行相同的功能，但它将操作数视为无符号数。

这种类型的其他指令还有：

- cmpeq（Comparison [$rj$] – [$rk$]，比较 [$rj$] – [$rk$]）
- cmpne（Comparison [$rj$] ≠ [$rk$]，比较 [$rj$] ≠ [$rk$]）
- cmpge（Signed comparison [$rj$] ≥ [$rk$]，有符号比较 [$rj$] ≥ [$rk$]）

- cmpgeu（Unsigned comparison [*ri*] ≥ [*rk*]，无符号比较 [*ri*] ≥ [*rk*]）
- cmpgt（Signed comparison [*ri*] > [*rk*]，有符号比较 [*ri*] > [*rk*]）
- cmpgtu（Unsigned comparison [*ri*] > [*rk*]，无符号比较 [*ri*] > [*rk*]）
- cmple（Signed comparison [*ri*] ≤ [*rk*]，有符号比较 [*ri*] ≤ [*rk*]）
- cmpleu（Unsigned comparison [*ri*] ≤ [*rk*]，无符号比较 [*ri*] ≤ [*rk*]）

Comparison 指令的立即数版本包括一个 16 位的立即操作数。例如，Compare Less Than Signed Immediate（有符号立即数的小于比较）指令

cmplti *ri*, *rj*, Value16

将寄存器 *rj* 中的有符号数与符号扩展后的立即操作数进行比较。如果 [*rj*] < Value16，就将 1 写入寄存器 *ri* 中，否则写入 0。

Compare Less Than Unsigned Immediate（无符号立即数的小于比较）指令

cmpltui *ri*, *rj*, Value16

将寄存器 *rj* 中的无符号数与零扩展后的立即操作数进行比较。如果 [*rj*] < Value16，就将 1 写入寄存器 *ri* 中，否则写入 0。

这种类型的其他指令还有：

- cmpeqi（Comparison [*ri*] = Value16，比较 [*ri*] = Value16）
- cmpnei（Comparison [*ri*] ≠ Value16，比较 [*ri*] ≠ Value16）
- cmpgei（Signed comparison [*ri*] ≥ Value16，有符号比较 [*ri*] ≥ Value16）
- cmpgeui（Unsigned comparison [*ri*] ≥ Value16，无符号比较 [*ri*] ≥ Value16）
- cmpgti（Signed comparison [*ri*] > Value16，有符号比较 [*ri*] > Value16）
- cmpgtui（Unsigned comparison [*ri*] > Value16，无符号比较 [*ri*] > Value16）
- cmplei（Signed comparison [*ri*] ≤ Value16，有符号比较 [*ri*] ≤ Value16）
- cmpleui（Unsigned comparison [*ri*] ≤ Value16，无符号比较 [*ri*] ≤ Value16）

### B.4.9 Shift 指令

Shift（移位）指令将一个指定寄存器中的内容向右或向左移位。

Shift Right Logical（逻辑右移）指令：

srl *ri*, *rj*, *rk*

根据寄存器 *rk* 中的 5 位最低有效位（表示 0 到 31 范围内的数）所指定的位数，将寄存器 *rj* 中的内容向右移动，并将结果保存在寄存器 *ri* 中。被移位的操作数左边空出的位用零填充。 [546]

Shift Right Logical Immediate（逻辑右移立即数）指令

srli *ri*, *rj*, Value5

根据指令中给出的 5 位无符号值 Value5 所指定的位数，将寄存器 *rj* 的内容向右移位，并将结果保存在寄存器 *ri* 中。被移位的操作数左边空出的位用零填充。

其他的 Shift 指令还有：

- sra（Shift Right Arithmetic，算术右移）
- srai（Shift Right Arithmetic Immediate，算术右移立即数）
- sll（Shift Left Logical，逻辑左移）
- slli（Shift Left Logical Immediate，逻辑左移立即数）

sra 和 srai 指令执行的操作与 srl 和 srli 指令相同，除了将符号位 $r_{31}$ 复制到被移位的操作数左边空出的位上。

sll 和 slli 指令与 srl 和 srli 指令类似，但它们是将寄存器 *rj* 中的操作数左移并将右边空出的位用零填充。

### B.4.10 Rotate 指令

有三条 Rotate（循环移位）指令。Rotate Right（循环右移）指令：

ror *ri, rj, rk*

根据寄存器 *rk* 中的 5 位最低有效位（表示 0 到 31 范围内的数）所指定的位数，将寄存器 *rj* 中的位按照从左到右的方向循环移位，并将结果保存在寄存器 *ri* 中。

Rotate Right（循环左移）指令

rol *ri, rj, rk*

与 ror 指令类似，但它将操作数按照从右到左的方向进行循环移位。

Rotate Left Immediate（循环左移立即数）指令

roli *ri, rj*, Value5

根据指令中给出的 5 位无符号值 Value5 所指定的位数，将寄存器 *rj* 中的位按照从右到左的方向循环移位，并将结果保存在寄存器 *ri* 中。

**例 B.7**

在图 2-24 中，我们给出了一个程序，它将两个 BCD 数字打包成一个字节。该程序的一个 [347] Nios II 版本如图 B-7 所示。

```
movia   r2, LOC     /* r2 指向数据 */
ldb     r3, (r2)    /* 将第一个字节装入 r3 中 */
slli    r3, r3, 4   /* 左移 4 位 */
addi    r2, r2, 1   /* 递增指针 */
ldb     r4, (r2)    /* 将第二个字节装入 r4 中 */
andi    r4, r4, 0xF /* 将高位清为 0*/
or      r3, r3, r4  /* 连接 BCD 数字 */
movia   r2, PACKED  /* 将结果保存到单元 */
stb     r3, (r2)    /* PACKED 中 */
```

图 B-7 一个将两个 BCD 数字打包到一个字节中的程序，对应于图 2-24

### B.4.11 Control 指令

对于 B.9 节中将要讨论的控制寄存器，有两条对其进行读和写的特殊指令。Read Control Register（读控制寄存器）指令

rdctl *ri*, ctl*j*

将控制寄存器 ctl*j* 中的内容复制到通用寄存器 *ri* 中。

Write Control Register（写控制寄存器）指令：

wrctl ctl*j*, *ri*

将通用寄存器 *ri* 中的内容复制到控制寄存器 ctl*j* 中。

有两条处理异常的指令：trap 和 eret。它们与 call 和 ret 指令类似，但它们用于异常。我们将在 B.10.2 节中讨论它们。

还有管理高速缓存的指令：flushd（刷新数据高速缓存行）、flushi（刷新指令高速缓存行）、initd（初始化数据高速缓存行）和 initi（初始化指令高速缓存行）。

## B.5 伪指令

为了编程方便，具有各种不同的指令是很有用的。从硬件的角度来看，很多指令都需要大量的电路来实现。通常情况下，某些指令的操作可以通过其他指令来有效地实现。如果这些指令不通过硬件实现，则它们被称为伪指令（pseudoinstruction）。汇编程序会将伪指令替换为硬件实现的实际指令。 548

B.4.5 节中，我们看到，Move 指令为伪指令。本节将介绍一些其他的伪指令。

Subtract Immediate（立即数减法）指令

subi *ri, rj*, Value16

被实现为

addi *ri, rj*, –Value16

Branch Greater Than Signed（大于有符号数时转移）指令

bgt *ri, rj*, LABEL

被实现为 blt 指令，并交换寄存器操作数的顺序。

当编写一个程序时，程序员不需要知道某指令是否是伪指令。但是，如果程序员试图在调试过程中检查汇编代码，就必须知道是否是伪指令。

## B.6 汇编指示

Nios II 的汇编指示（assembler directive）符合广泛使用的 GNU 汇编程序所定义的规范，GNU 汇编程序是在公共领域中使用的软件。汇编指示符以一个句号开始。下面介绍一些常用的指示符。

.org *Value*

这是在第二章中讨论的 ORIGIN 指示符。

.equ *LABEL, Value*

名称 LABEL 与 Value 等价。例如

.equ LIST, 0x1000

将十六进制数 1000 赋值给 LIST。

.byte *expressions*

将字节大小的数据项放到存储器中。数据项通过用逗号分隔的表达式来指定。例如：23，6 + LABEL 和 Z – 4。每个表达式都被汇编到下一个字节中。

.hword *expressions* 549

这与 .byte 一样，除了表达式被汇编成连续的 16 位半字。

.word *expressions*

这与 .byte 一样，除了表达式被汇编成连续的 32 位字。

.skip *Size*

这是在第 2 章中讨论的 RESERVE 指令。它在存储器中保留 Size 所指定的字节数。

.end

指示源代码文件的结尾。该指示符之后的一切代码都将被汇编程序忽略。

**例 B.8**

图 B-8 说明了一些汇编指示符的使用。它对应于图 2-13。

```
        .org    100             /* 将该代码放到单元 100 中 */
        movia   r2, N           /* 获取地址 N*/
        ldw     r2, (r2)        /* 加载列表的大小 */
        mov     r3, r0          /* 将和初始化为 0*/
        movia   r4, NUM1        /* 加载第一个数的地址 */
LOOP:   ldw     r5, (r4)        /* 获取下一个数 */
        add     r3, r3, r5      /* 将这个数加到和中 */
        addi    r4, r4, 4       /* 递增指向列表的指针 */
        subi    r2, r2, 1       /* 递减计数器 */
        bgt     r2, r0, LOOP    /* 如果没完成就循环回到前面 */
        movia   r6, SUM         /* 获取地址 SUM*/
        stw     r3, (r6)        /* 保存最终的和 */
        /* 一条指令

        .org    200             /* 将数据放到单元 200 中 */
SUM:    .skip   4
N:      .word   150
NUM1:   .skip   600
        end
```

图 B-8 对应于图 2-13 的程序

[550]

## B.7 进位和溢出检测

当执行诸如 Add 或 Subtract 这样的算术运算时，往往最重要的是要了解最高有效位是否会产生进位或者是否会产生算术溢出。如 2.10.2 节中所讨论的，一个使用条件码的处理器会自动地设置 C 和 V 标志位来指示是否有发生进位或溢出。但是，Nios II 处理器不包含条件码标志。对于有符号和无符号操作数，Add 和 Subtract 指令都以同样的方式执行相应的操作。必须使用附加的指令来检测进位和溢出的产生。当无符号数相加或相减时，第 31 位的进位输出是非常有趣的。当运算中包含有符号操作数时溢出也是非常有趣的。

### 1. 加法中的进位与溢出

当执行如下指令时：

```
add    r4, r2, r3
```

可以通过检查寄存器 r4 中的无符号总和是否小于某一个无符号操作数来检测是否发生进位。如果该指令后面有如下指令：

```
cmpltu   r5, r4, r2
```

则进位位将会被写入到寄存器 r5 中。

当检测到一个进位 1 时，如果想转到位置 CARRY 处继续执行，则可以通过使用如下指令来实现：

```
add    r4, r2, r3
bltu   r4, r2, CARRY
```

算术溢出可以通过检查源操作数和结果的符号来进行检测。如果两个正数相加产生一个负数和或者两个负数相加产生一个正数和，就会发生溢出。利用这个事实，如果加法运算产生算术溢出，那么指令序列

```
add   r4, r2, r3
xor   r5, r4, r2
xor   r6, r4, r3
and   r5, r5, r6
```

```
blt   r5, r0, OVERFLOW
```

将会导致一个到 OVERFLOW 的跳转。两条 xor 指令用来比较总和与每一个被加数的符号。虽然这两条指令对所有 32 位数都执行 XOR 操作，但在随后的转移指令中只考虑符号位 $b_{31}$。只有当和与加数的符号不同时，这个位才被置为 1。and 指令使得只有当两个操作数的符号相同但总和的符号不同时，才会将位 $r5_{31}$ 置为 1。如果 r5 中的有符号数为负数（$r5_{31}$ 等于 1），blt 指令将导致一个转移。 551

**2. 减法中的进位和溢出**

减法运算中的进位和溢出条件可以使用类似的方法来检测。所生成的差的最高有效位的进位可以通过检查被减数是否小于减数来进行检测。例如，如果在减法运算中产生一个进位，那么指令

```
sub   r4, r2, r3
bltu  r2, r3, CARRY
```

将使得程序转移到位置 CARRY 处执行。

如果被减数和减数的符号不同并且所生成的差的符号与被减数的符号不同，那么就会发生算术溢出。这种情况可通过下列指令序列进行检测：

```
sub   r4, r2, r3
xor   r5, r2, r3
xor   r6, r2, r4
and   r5, r5, r6
blt   r5, r0, OVERFLOW
```

其中，两条 xor 指令分别将被减数的符号与减数和所生成的差的符号进行比较。只有当上述的溢出条件为真时，and 指令才会将位 $r5_{31}$ 置为 1。

**例 B.9**

考虑一个将两个整数相加的任务，这两个整数太大，不能放在 32 位的寄存器中。这可以通过将每个数装入两个不同的寄存器中然后执行上面所描述的进位检测加法来完成。我们将使用十六进制数，以便于查看一个数如何在两个寄存器中表示。假设 A = 10A72C10F8，B = 4A5C00FE04，则 C = A + B 可以如图 B-9 所示的那样进行计算。寄存器 r2 和 r3 中分别装入 A 的低 32 位和高 32 位，寄存器 r4 和 r5 以相同的方式保存 B。注意，寄存器 r2 和 r4 中的 32 位值是通过使用两个 16 位的立即操作数来进行装入的，如 B.4.4 节中所解释的那样。A 和 B 的低 32 位相加后，进位输出将包含在高 32 位的加法中。所生成的和 C = 5B032D0EFC，则放到寄存器 r6 和 r7 中。 552

```
orhi    r2, r0, 0xA72C    /* 现在 r2 中的值为 A72C0000 */
ori     r2, r2, 0x10F8    /* 现在 r2 中的值为 A72C10F8 */
ori     r3, r0, 0x10      /* 现在 r3 中的值为 10 */
orhi    r4, r0, 0x5C00    /* 现在 r4 中的值为 5C000000 */
ori     r4, r4, 0xFE04    /* 现在 r4 中的值为 5C00FE04 */
ori     r5, r0, 0x4A      /* 现在 r5 中的值为 4A */
add     r6, r2, r4        /* 将低 32 位相加 */
cmpltu  r7, r6, r2        /* 检查是否产生进位 */
add     r7, r7, r3        /* 将进位与高位相加 */
add     r7, r7, r5        /* 将进位与高位相加 */
```

图 B-9 例 B.9 的程序

## B.8 示例程序

2.12 节中，我们给出了两个示例程序。计算两个向量点积程序的 Nios II 版本由图 B-10 给出，它对应于图 2-27。搜索匹配字符串的程序如图 B-11 所示，它对应图 2-30。

```
        movia   r2, AVEC        /* r2 指向向量 A */
        movia   r3, BVEC        /* r3 指向向量 B */
        movia   r4, N           /* 获取地址 N */
        ldw     r4, (r4)        /* r4 作为计数器 */
        mov     r5, r0          /* r5 累加点积 */
LOOP:   ldw     r6, (r2)        /* 获取向量 A 的下一个元素 */
        ldw     r7, (r3)        /* 获取向量 B 的下一个元素 */
        mul     r8, r6, r7      /* 计算下一对元素的积 */
        add     r5, r5, r8      /* 累加到前面的和中 */
        addi    r2, r2, 4       /* 增加指向向量 A 的指针 */
        addi    r3, r3, 4       /* 增加指向向量 B 的指针 */
        subi    r4, r4, 1       /* 递减计数器 */
        bgt     r4, r0, LOOP    /* 如果没有完成再次循环 */
        movia   r2, DOTPROD     /* 将点积保存到 */
        stw     r5, (r2)        /* 存储器中 */
```

图 B-10 计算两个向量点积的程序，对应于图 2-27

```
        movia   r2, T           /* 获取 T(0) 的地址 */
        movia   r3, P           /* 获取 P(0) 的地址 */
        movia   r4, N           /* 获取地址 N */
        ldw     r4, (r4)        /* 读取值 n */
        movia   r5, M           /* 获取地址 M */
        ldw     r5, (r5)        /* 读取值 m */
        sub     r4, r4, r5      /* 计算 n - m */
        add     r4, r2, r4      /* T(n - m) 的地址 */
        add     r5, r3, r5      /* P(m) 的地址 */
LOOP1:  mov     r6, r2          /* 遍历字符串 T */
        mov     r7, r3          /* 遍历字符串 P */
LOOP2:  ldb     r8, (r6)        /* 比较字符串 T 和 */
        ldb     r9, (r7)        /* P 中的一对字符 */
        bne     r8, r9, NOMATCH /* P 中的下一个字符 */
        addi    r6, r6, 1       /* 指向 T 中的下一个字符 */
        addi    r7, r7, 1       /* 指向 P 中的下一个字符 */
        bgt     r5, r7, LOOP2   /* 如果没有完成再次循环 */
        movia   r9, RESULT      /* 将 T(i) 的地址保存 */
        stw     r2, (r9)        /* 到单元 RESULT 中 */
        br      DONE
NOMATCH: addi   r2, r2, 1       /* 指向 T 中的下一个字符 */
        bge     r4, r2, LOOP1   /* 如果没有完成循环回到前面 */
        movi    r8, -1          /* 在单元 RESULT 中写入 */
        movia   r9, RESULT      /* -1 表明没有发现匹配 */
        stw     r8, (r9)
DONE:   next instruction
```

图 B-11 对应于图 2-30 的字符串搜索程序

## B.9 控制寄存器

到目前为止，我们只考虑了通用寄存器的使用。在讨论 Nios II 的输入/输出方法之前，我们需要先介绍一下控制寄存器。在第 3 章中，我们解释了控制寄存器在中断处理中的使用。图 3-7 描述了四个能表示该任务所需功能的寄存器。Nios II 控制寄存器具有相同的功能。在

Nios II 处理器的基本配置中，有六个控制寄存器。当要实现诸如存储器管理单元或外部中断控制器等高级的硬件模块时，还需要提供额外的控制寄存器。

基本的控制寄存器如表 B-3 所示。它们被称为 ctl0 到 ctl5，在表中还有指示它们功能的又名（alternate name）。这两套名称都可以被汇编程序识别。控制寄存器可通过特殊的指令 rdctl 和 wrctl 进行读写。它们的用法如下：

表 B-3　Nios II 基本控制寄存器

| 寄存器 | 名　称 | $b_{31}\cdots b_2$ | $b_1$ | $b_0$ |
|---|---|---|---|---|
| ctl0 | status | 保留 | U | PIE |
| ctl1 | estatus | 保留 | EU | EPIE |
| ctl2 | bstatus | 保留 | BU | BPIE |
| ctl3 | ienable | 中断允许位 | | |
| ctl4 | ipending | 待处理中断位 | | |
| ctl5 | cpuid | 处理器标识符 | | |

- 寄存器 ctl0 是状态寄存器，它表明处理器当前的状态。在基本配置中，只使用其中的两位：
  - PIE 是处理器中断允许位。当 PIE= 1 时，处理器将接受来自 I/O 设备的中断请求，而当 PIE= 0 时，处理器将忽略中断请求。
  - U 是用户 / 管态模式位。0 表示管态模式，1 表示用户模式。
- 当正在执行一个中断或异常服务程序时，寄存器 ctl1 用于自动保存状态寄存器的内容。EU 位和 EPIE 位分别保存 U 位和 PIE 位的状态。
- 寄存器 ctl2 用来保存调试中断（debug break）过程中状态寄存器的内容。BU 和 BPIE 位保存 U 和 PIE 位的状态。
- 寄存器 ctl3 用来允许来自 I/O 设备的个别中断。其中每一位对应于 irq0 到 irq31 中的一个中断。值 1 和 0 分别用来允许和禁止每个中断。
- 寄存器 ctl4 表示哪些中断请求正在等待处理。如果中断 irq$k$ 处于激活状态，并且中断允许位 ctl3$k$（等于 1）允许该中断，则给定位 ctl4$_k$ 的值将会被置为 1。
- 寄存器 ctl5 用来保存一个值，该值能唯一标识多处理器系统中的一个处理器。

553 ! 554

**操作模式**

Nios II 处理器有两种不同的操作模式：

- 管态模式（supervisor mode），在这种模式下，处理器可以执行所有的指令，并能执行所有可用的功能。当处理器复位时，进入此模式。
- 用户模式（user mode），在这种模式下，一些控制指令不能执行。

在 Nios II 处理器的基本配置中，所有的程序都在管态模式下运行。当处理器被配置为包含存储器管理单元时，便可使用用户模式了。用户模式的唯一目的是支持操作系统，使得操作系统软件在管态模式下运行而应用程序在用户模式下运行。

## B.10　输入 / 输出

第 3 章所讨论的处理输入 / 输出传输的一般概念完全适用于 Nios II 处理器。I/O 设备采用存储器映射 I/O 的方式，I/O 传输在程序的控制下执行或者通过使用中断机制来执行。 555

### B.10.1　程序控制 I/O

3.1.2 节介绍了程序控制 I/O 的概念。图 3-4 给出了一个从键盘读入一行字符并将其发送给显示设备的 RISC 风格的程序。图 B-12 给出了该程序的 Nios II 实现。假定键盘和显示器接口具有图 3-3 所示的寄存器。这些寄存器的名称与图 3-3 所示的地址相关联。注意，可通过 ldbio 和 stbio 指令来访问 I/O 寄存器。正如 B.4.2 节所介绍的，这些指令可以绕过一个给定的 Nios II 系统中的高速缓存。

```
        .equ    KBD_DATA, 0x4000  /* 为键盘和显示器的数据寄存器 */
        .equ    DISP_DATA, 0x4010 /* 指定地址 */
        .text                     /* 存储该行字符的单元 */
        movia   r2, LOC           /* r2 指向键盘数据寄存器 */
        movia   r3, KBD_DATA      /* r3 指向显示器数据寄存器 */
        movia   r4, DISP_DATA     /* 载入回车符的 ASCII 码 */
        addi    r5, r0, 0x0D      /* 读取键盘状态寄存器 */
READ:   ldbio   r6, 4(r3)         /* 检查 RIN 标志 */
        andi    r6, r6, 2
        beq     r6, r0, READ
        ldbio   r7, (r2)          /* 从键盘读取字符 */
        stb     r7, (r2)          /* 将字符写入主存，*/
        addi    r2, r2, 1         /* 并递增指针 */
ECHO:   ldbio   r6, 4(r4)         /* 读取显示器状态寄存器 */
        andi    r6, r6, 4         /* 检查 DOUT 标志 */
        beq     r6, r0, ECHO
        stbio   r7, (r4)          /* 将字符发送给显示器 */
        bne     r5, r7, READ      /* 如果字符不是 CR（回车），则循环回到前面 */
```

图 B-12 一个读取并显示一行字符的程序，对应于图 3-4

## B.10.2 中断和异常

使用中断是执行 I/O 传输的一种有效方式。3.2 节概括描述了这种方法。Nios II 实现的中断符合这一描述。

Nios II 系统可以处理两种类型的中断。来自 I/O 设备的服务请求被认为是硬件中断（hardware interrupt）。其他任何中断都不称为中断，而称为异常（exception）。事实上，通常使用术语“异常”来描述任何硬件启动或软件启动的偏离正常运行的情况。

556

程序正常执行流程中的异常可由以下几个方面引起：

- 硬件中断
- 软件陷阱
- 未实现的指令

为响应异常，Nios II 处理器执行下列操作：

1）将 status 寄存器（ctl0）中的内容复制到 estatus 寄存器（ctl1）来保存现有的处理器状态信息。

2）清除 status 寄存器中的 U 位，以确保处理器处于管态模式。

3）清除 status 寄存器中的 PIE 位，以禁止更多的外部中断。

4）将返回地址（异常之后那条指令的地址）写入到 ea 寄存器（r29）中。

5）将程序的执行转移到异常处理程序（exception handler）。它确定了异常的原因，并调用所需的异常服务程序（exception-service routine）来响应异常。

异常处理程序的地址是在 Nios II 系统设计的时候指定的，不能在运行时通过软件来改变。这个地址可以由设计者提供；否则，默认地址就是从主存起始地址偏移 0x20 的位置。例如，如果主存地址从 0 开始，那么异常处理程序的默认地址就是 0x20。

1. 硬件中断

一个 I/O 设备通过发出处理器 32 个中断请求输入 irq0 到 irq31 中的一个来请求中断。仅当以下三个条件都为真时才会产生中断：

- status 寄存器中的 PIE 位被置为 1
- 中断请求输入 irq*i* 是有效的
- 相应的中断允许位 ctl3.*i* 置为 1

ipending 寄存器（ctl4）中的内容表明哪些中断请求正在等待处理。而异常处理程序决定哪个待处理的中断具有最高优先级，并调用相应的中断服务程序。

中断服务程序完成后，使用 eret（异常返回）指令将执行控制权返回给被中断的程序。当中断请求发生时，Nios II 处理器不会先完成正在执行的指令，而是马上开始处理该硬件中断（这与第 3 章中我们所讨论的不同，那一章中我们假定当前指令的执行被完成后才能响应中断。）因此，从中断服务程序返回后，被中断的指令必须被重新执行。为了实现这一目标，异常处理程序必须调整 ea 寄存器的内容，此时它指向被中断程序的下一条指令。因此，ea 寄存器中的地址必须在执行 eret 指令之前减去 4。 557

图 B-13 显示了如何使用中断来从键盘读取一行字符并使用轮询法进行显示。该程序对应于图 3-8 中的程序。为了使例子简单，我们不使用异常处理程序，而只使用必要的中断服务程序。假设键盘是异常的唯一来源，因此当中断请求到达时，程序自动将其视为来自键盘。

从程序中可以观察到，我们将地址 KBD 和 DISPAY 分别定义为 0x4000 和 0x4010，这是图 3-3 中寄存器 KBD_DATA 和 DISP_DATA 的实际地址。程序通过位移寻址方式访问键盘和显示器接口中的其他寄存器。通过使用伪指令 movia 来将 32 位的地址 KBD 和 DISPLAY，以及存储单元 PNTR、EOL 和 LINE 的地址装入处理器寄存器中。

还需注意的是，在返回到被中断的程序之前对寄存器 ea 中的异常返回地址进行了调整。

```
            .equ    KBD, 0x4000         /* 键盘地址 */
            .equ    DISPLAY, 0x4010     /* 显示器地址 */
            .equ    PNTR, 0x2000        /* 存储器中的缓冲区指针 */
            .equ    EOL, 0x2004         /* 行结束指示变量 */
            .equ    LINE, 0x2008        /* 缓冲区起始地址 */
        中断服务程序
            .org    0x0120
ILOC:       subi    sp, sp, 16          /* 保存寄存器 */
            stw     r2, 12(sp)
            stw     r3, 8(sp)
            stw     r4, 4(sp)
            stw     r5, (sp)
            movia   r2, PNTR
            ldw     r3, (r2)            /* 载入地址指针 */
            movia   r4, KBD
            ldbio   r5, (r4)            /* 从键盘读入字符 */
            stb     r5, (r3)            /* 将字符写入存储器 */
            addi    r3, r3, 1           /* 并递增指针 */
            stw     r3, (r2)            /* 更新存储器中的指针 */
            movia   r2, DISPLAY
ECHO:       ldbio   r3, 4(r2)           /* 查看显示器是否准备就绪 */
            andi    r3, r3, 4           /* 检查 DOUT 标志 */
            beq     r3, r0, ECHO
            stbio   r5, (r2)            /* 显示刚读入的字符 */
            addi    r3, r0, 0x0D        /* 回车符为 ASCII 码 */
            bne     r5, r3, RTRN        /* 如果字符不是 CR，则返回 */
            movi    r3, 1
            movia   r5, EOL
            stw     r3, (r5)            /* 指示行的结束 */
            stbio   r0, 8(r4)           /* 在 KBD 接口中禁止中断 */
RTRN:       ldw     r5, (sp)            /* 恢复寄存器 */
            ldw     r4, 4(sp)
            ldw     r3, 8(sp)
            ldw     r2, 12(sp)
            addi    sp, sp, 16
            subi    ea, ea, 4           /* 调整返回地址 */
            eret                        /* 从异常返回 */
```

图 B-13 使用中断方式读取一行字符，并使用轮询方式显示该行字符的程序

```
主程序
START:  movia   r2, LINE
        movia   r3, PNTR
        stw     r2, (r3)        /* 初始化缓冲区指针 */
        movia   r2, EOL
        stw     r0, (r2)        /* 清除行结束指示字符 */
        movia   r2, KBD
        movi    r3, 2           /* 允许键盘中断 */
        stbio   r3, 8(r2)       /*
        rdctl   r2, ienable
        ori     r2, r2, 2       /* 在处理器控制寄存器中 */
        wrctl   ienable, r2     /* 允许键盘中断 */
        rdctl   r2, status
        ori     r2, r2, 1
        wrctl   status, r2      /* 设置状态寄存器中的 PIE 位 */
        下一条指令
```

图 B-13 （续）

2. 软件陷阱

当在程序中执行 trap 指令时，就会发生软件异常。这将使得下一条指令的地址被保存在 ea 寄存器（r29）中。然后，中断被禁止并且程序的执行被转移到异常处理程序。

异常服务程序的最后一条指令是 eret，它将执行控制权返回到导致异常的 trap 指令之后的那条指令。返回地址是寄存器 ea 的内容。eret 指令通过将 estatus 寄存器的内容复制到 status 寄存器来恢复处理器的先前状态。

软件陷阱的一个常见用途是将控制权转移给一个不同的程序，比如操作系统，正如第 4 章所解释的那样。

3. 未实现的指令

当处理器遇到一条硬件没有实现的有效指令时，就会发生异常。例如，Nios II 处理器可能被配置为不包含执行乘法和除法运算的硬件电路。在这种情况下，如果遇到 mul 或 div 指令，将会发生异常。异常处理程序可能调用一个用软件实现所需的操作。

4. 异常处理程序

异常处理程序是一个处理异常情况的程序。当异常发生时，异常处理程序被装入存储器中的一个预定单元，并将执行控制权转移到该单元。如上所述，在 Nios II 系统中，异常处理程序的默认单元是 0x20。

图 B-14 给出了异常处理程序的框架。该程序从保存它使用的所有寄存器以及子程序链接寄存器 ra 开始，正如第 3 章例 3.3 中所解释的那样。然后，它必须确定异常请求的来源。首先，它检查该请求是否是一个硬件中断。它读取 ipending 控制寄存器，逐个测试这个字的每一位，以找出被置为 1 的位。这些位被检查的顺序决定了各种中断源被赋予的优先级。一旦找到被置为 1 的位，相应的中断服务程序就被执行。虽然可能有多达 32 种不同的中断源，但在一个典型的系统中，I/O 设备的数量还是要小得多。如果该请求不是一个硬件中断，那么就检查其他的异常并根据需要进行处理。在返回到被中断的程序之前，所保存的寄存器被恢复。

558 / 559

主程序必须初始化所需要的设置，以达到 I/O 设备所期望的中断行为。这类似于图 B-13 所示的初始化操作。

5. 复位

Nios II 系统必须包含复位（reset）功能，使其有可能从一个不能作为异常进行处理的错

误状态中恢复过来。这可以通过提供一个复位键来完成。按下该键时，处理器复位并执行一个适当的程序。如果存储器的地址从 0 开始，那么很自然地将会使用这个地址作为复位位置，在实现 Nios II 系统时可以这样设置。当处理器复位时，程序计数器和控制寄存器被清为零。因此，从地址 0 处的指令开始执行。为了能在复位后执行主程序，只需要在地址 0 处放置一条 Branch 指令，并将主程序的第一条指令作为转移目标。

```
            .org    0
RESET:      br      START           /* 转移到主程序 */
异常处理程序
            .org    0x020
ELOC:       subi    sp, sp, 12      /* 保存寄存器 */
            stw     ra, 8(sp)
            stw     et, 4(sp)
            stw     r2, (sp)
            ...
            rdctl   et, ipending    /* 获取待处理的中断请求 */
            beq     et, r0, OTHER   /* 不是外部中断 */
            subi    ea, ea, 4       /* 调整返回地址 */
IRQ0:       andi    r2, et, 1       /* 检查 irq0 是否处于激活状态 */
            beq     r2, r0, IRQ1    /* 如果不是，则检查 irq1*/
            call    ISR0            /* 处理 irq0 请求 */
IRQ1:       andi    r2, et, 2       /* 检查 irq1 是否处于激活状态 */
            beq     r2, r0, IRQ2    /* 如果不是，则检查 irq2*/
            call    ISR1            /* 处理 irq1 请求 */
            .
IRQ31:      orhi    r2, r0, 0x8000  /* 检查第 31 位的模式 */
            and     r2, et, r2      /* 检查 irq31 是否处于激活状态 */
            beq     DONE            /* 如果不是，结束外部中断的检查 */
            call    ISR31           /* 处理 irq31 请求 */
            br      DONE            /* 结束外部中断的检查 */
OTHER:      ...
            检查其他异常并调用适当的
            异常服务程序的指令
            ...
DONE:       ldw     r2, (sp)        /* 恢复寄存器 */
            ldw     et, 4(sp)
            ldw     ra, 8(sp)
            addi    sp, sp, 12
            eret                    /* 返回到被中断的程序 */
irq0 的中断服务程序
ISR0:       ...
            .
            ret
irq31 的中断服务程序
ISR31:      ...
            .
            ret
主程序
START:      ...
```

图 B-14　异常处理程序的框架

## B.11　NIOS II 处理器的高级配置

在前面的章节中，我们讨论了 Nios II 处理器基本配置中的一些特点。我们还可以实现更大的 Nios II 处理器，包括可以提供增强功能的额外硬件模块。在本节中，我们将讨论三个可能的增强功能。

### B.11.1 外部中断控制器

B.10.2 节描述了内部中断控制器所使用的机制，其中软件是用来确定中断请求的优先级的。这种方法比较简单，易于实现，但在一些应用中对中断进行处理时，它可能会导致不可接受的较长延迟。为了减少延迟，处理器可以包含一个使用向量中断的外部中断控制器电路。在这种情况下，控制器为每个中断请求提供其中断服务程序的地址。

为了进一步减少中断延迟，处理器还可以包含 32 个通用寄存器的影子寄存器。可以实现几组影子寄存器并将其与不同的中断关联起来。对于所接收到的中断请求，外部中断控制器确定将会用到的影子寄存器组。然后，使用所确定的影子寄存器组，而不使用一般的寄存器组。这样就不需要保存中断服务程序所使用的寄存器内容。

优先级与不同的中断相关联。当处理器正在处理一个给定优先级的中断时，它只能被另一个具有更高优先级的中断所打断。处理器的当前优先级和指示活跃影子寄存器组的标识是处理器状态的一部分。处理器的状态保存在状态控制寄存器 ctl0 中，而上述这些信息保存在表 B-3 中该寄存器的“保留”位中。

在定义 Nios II 处理器时，它可以被配置为使用内部中断控制器或者使用外部中断控制器。

### B.11.2 存储器管理单元

Nios II 系统可以包含一个存储器管理单元（MMU），它提供了 8.8 节所讨论的功能。包含 MMU 的目的是为了支持使用存储器管理能力的操作系统，包含 MMU 的系统可以使用管态模式和用户模式。MMU 是一个可选的单元，在设计系统的时候可以指定是否包含 MMU。

### B.11.3 浮点硬件

Nios II 体系结构提供了自定义指令（custom instruction）。这些指令可以用来定义各种操
562 作，但这些操作可能需要使用额外的电路。有一组预定的自定义指令可以用来实现浮点算术运算。如果需要进行浮点运算，就需要在设计 Nios II 系统的时候包含必要的硬件。

## B.12 结束语

本附录中，我们描述了 Nios II 处理器的基本实现中的主要特征。基本配置可提供一个应用广泛、功能强大的处理器。在 B.11 节中，我们介绍了可包含在 Nios II 系统中的用来提供额外功能的硬件。由于 Nios II 是一个软处理器，所以系统设计者可以定制系统的功能，以适应所需的应用。

Nios II 处理器主要用于商业和工业应用中，但它在教学环境中也非常具有吸引力。实现 Nios II 处理器的 FPGA 技术价格合理，易于使用。Altera 公司目前已经开发出了一套开发教育板，为向学生介绍数字技术提供了一个很好的平台。这些开发教育板中包括计算机系统的典型组件，可使学生很容易地开展关于计算机组织的硬件和软件方面的研究。

在 Altera 公司的网站（*http://www.altera.com*）上可以找到大量关于 Nios II 处理器和系统的文献。

## B.13 问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。

**例 B.10**

**问题**：假设有一个 ASCII 编码的字符串保存在存储器中，起始地址为 STRING。该字符串以回车

（CR）符结束。写一个 Nios II 程序来确定该字符串的长度。

**解答：** 图 B-15 给出了一个可能的程序。字符串中的每个字符与 CR（ASCII 码为 0x0D）进行比较，计数器递增，直到到达字符串的末尾。结果存储在单元 LENGTH 中。

```
        movia   r2, STRING      /* r2 指向字符串的开始 */
        add     r3, r0, r0      /* r3 是一个计数器，被清为 0 */
        addi    r4, r0, 0x0D    /* 加载回车符的 ASCII 码 */
LOOP:   ldb     r5, (r2)        /* 获取下一个字符 */
        beq     r5, r4, DONE    /* 如果是回车符，则结束 */
        addi    r2, r2, 1       /* 递增字符串指针 */
        addi    r3, r3, 1       /* 递增计数器 */
        br      LOOP            /* 如果没有结束，则循环回到前面 */
DONE:   movia   r2, LENGTH      /* 将计数值存放到 */
        stw     r3, (r2)        /* 存储单元 LENGTH 中 */
```

图 B-15　例 B.10 的程序

**例 B.11**

**问题：** 我们希望在一个 32 位的 n 个整数列表中找出最小的数。数据的存储地址从 $(1000)_{10}$ 开始。该地址处的字必须保存所找到的最小数，下一个字包含列表中的项数 $n$，之后的 $n$ 个字包含列表中的数。写一个 Nios II 程序，找出最小的数，并包含按照规定组织数据所需的汇编指示（assembler directive）。 563

**解答：** 图 B-16 中的程序实现了所需的任务。程序中的注释解释了这个任务如何执行。列表中包含了一些样本数字。

```
        .equ    LIST, 0x1000    /* 列表起始地址 */
        movia   r2, LIST        /* r2 指向列表的开始 */
        ldw     r3, 4(r2)       /* r3 是计数器，初始化为 n */
        addi    r4, r2, 8       /* r4 指向第一个数 */
        ldw     r5, (r4)        /* r5 保存目前为止所找到的最小的数 */
LOOP:   subi    r3, r3, 1       /* 递减计数器 */
        beq     r3, r0, DONE    /* 如果 r3 等于 0，则结束 */
        addi    r4, r4, 4       /* 递增列表指针 */
        ldw     r6, (r4)        /* 获取下一个数 */
        ble     r5, r6, LOOP    /* 检查是否可以找到更小的数 */
        add     r5, r6, r0      /* 修改所找到的最小的数 */
        br      LOOP
DONE:   stw     r5, (r2)        /* 将最小的数存储到 SMALL 中 */

        .org    0x1000
SMALL:  .skip   4               /* 所找到的最小数的存储空间 */
N:      .word   7               /* 列表中的项数 */
ENTRIES: .word  45,36,61,18,2   /* 列表中的项 */
        .end
```

图 B-16　例 B.11 的程序 564

**例 B.12**

**问题：** 写一个 Nios II 程序，将一个 $n$ 位十进制整数转换成二进制数。如果数字是通过键盘输入的，那么十进制数是以 $n$ 个 ASCII 编码字符的形式给出的。

**解答：** 考虑一个 4 位十进制数 $D = d_3d_2d_1d_0$。该数的值为 $((d_3 \times 10 + d_2) \times 10 + d_1) \times 10 + d_0$。该数的这种表示是图 B-17 中程序所使用转换技术的基础。注意，每个 ASCII 编码字符先被转换成一个二进制编码的十进制（BCD）数字，然后再用于计算。

**例 B.13**

**问题：** 考虑一个数字阵列 A($i$, $j$)，其中 $i = 0$ 到 $n-1$，是行索引，$j = 0$ 到 $m-1$，是列索引。该阵列按行存储在计算机的存储器中，每行元素占用 $m$ 个连续的字单元。写一个 Nios II 子程序，将第 $x$ 列的元素

逐一加到第 $y$ 列的元素上，和元素（sum element）放在第 $y$ 列上。索引值 $x$ 和 $y$ 通过寄存器 r2 和 r3 传递给子程序。参数 $n$ 和 $m$ 通过寄存器 r4 和 r5 传递给子程序。元素 A(0,0) 的地址通过寄存器 r6 进行传递。

```
        movia   r2, N           /* r2 是计数器。*/
        ldw     r2, (r2)        /* 初始化为 n*/
        movia   r3, DECIMAL     /* r3 指向 ASCII 数字 */
        add     r4, r0, r0      /* r4 用来保存二进制数 */
LOOP:   ldb     r5, (r3)        /* 获取下一个 ASCII 数字 */
        andi    r5, r5, 0x0F    /* 产生 BCD 数字 */
        add     r4, r4, r5      /* 加到中间结果中 */
        addi    r3, r3, 1       /* 递增数字指针 */
        subi    r2, r2, 1       /* 递减计数器 */
        beq     r2, r0, DONE
        muli    r4, r4, 10      /* 乘以 10 */
        br      LOOP            /* 如果未完成，则循环回到前面 */
DONE:   movia   r5, BINARY      /* 将结果存放到 */
        stw     r4, (r5)        /* 存储单元 BINARY 中 */
```

图 B-17 例 B.12 的程序

**解答**：图 B-18 给出了一个可能的程序。我们假定值 $x$、$y$、$n$ 和 $m$ 分别存放在存储单元 X、Y、N 和 M 中。此外，阵列中的元素存储在从单元 ARRAY 开始的连续的字中，ARRAY 是元素 A(0,0) 的地址。程序中的注释说明了每条指令的作用。

```
        movia   r2, X
        ldw     r2, (r2)        /*  加载值 x */
        movia   r3, Y
        ldw     r3, (r3)        /*  加载值 y */
        movia   r4, N
        ldw     r4, (r4)        /*  加载值 n */
        movia   r5, M
        ldw     r5, (r5)        /*  加载值 m */
        movia   r6, ARRAY       /*  主线 A(0,0) 的地址 */
        call    SUB
        下一条指令
        .
        .
SUB:    subi    sp, sp, 4
        stw     r7, (sp)        /*  保存寄存器 r7 的内容 */
        slli    r5, r5, 2       /*  确定一列中连续元素 */
                                /*  之间的距离（以字节为单位）*/

        sub     r3, r3, r2      /*  产生 y – x 的值 */
        slli    r3, r3, 2       /*  产生 4(y – x) 的值 */
        slli    r2, r2, 2       /*  产生 4x 的值 */
        add     r6, r6, r2      /*  r6 指向 A(0,x) */
        add     r7, r6, r3      /*  r7 指向 A(0,y) */
LOOP:   ldw     r2, (r6)        /*  获取第 x 列中的下一个数 */
        ldw     r3, (r7)        /*  获取第 y 列中的下一个数 */
        add     r2, r2, r3      /*  将两数相加，*/
        stw     r2, (r7)        /*  并保存总和 */
        add     r6, r6, r5      /*  递增第 x 列的指针 */
        add     r7, r7, r5      /*  递增第 y 列的指针 */
        subi    r4, r4, 1       /*  递减行计数器 */
        bgt     r4, r0, LOOP    /*  如果未完成，则循环回上 */
        ldw     r7, (sp)        /*  恢复寄存器 r7 的内容 */
        addi    sp, sp, 4
        ret                     /*  返回到调用程序 */
```

图 B-18 例 B.13 的程序

**例 B.14**

**问题**：假设存储单元 BINARY 中包含一个 32 位的模式。我们希望在一个具有图 3-3 所示接口的显示设备上将这些位显示为 8 个十六进制数字。写一个 Nios II 程序完成这个任务。

**解答**：首先，我们需要将这个 32 位的模式转化为用 ASCII 编码字符表示的十六进制数字。转换过程可以通过使用查表法来完成。必须构造一个包含 16 项的表，以便于为每一个十六进制数字提供其 ASCII 码。然后，对于 BINARY 中该模式的每一个 4 位的段，都可以在表中查找到其相应的字符，并将这些字符存储在从单元 HEX 开始的 8 个连续的字节单元中。最后，将这 8 个字符发送给显示器。图 B-19 给出了一个可能的程序。

```
            movia   r2, BINARY        /*  获取二进制数的地址 */
            ldw     r2, (r2)          /*  加载该二进制数 */
            movi    r3, 8             /*  r3 是数字计数器，被设置为 8 */
            movia   r4, HEX           /*  r4 指向十六进制数字 */
LOOP:       roli    r2, r2, 4         /*  将高位数字循环 */
                                      /*  移位到低位位置上 */
            andi    r5, r2, 0xF       /*  提取下一个数字 */
            ldb     r6, TABLE(r5)     /*  获取该数字的 ASCII 码，*/
            stb     r6, (r4)          /*  并将其存储在 HEX 缓冲区中 */
            subi    r3, r3, 1         /*  递减数字计数器 */
            addi    r4, r4, 1         /*  递增指向十六进制数字的指针 */
            bgt     r3, r0, LOOP      /*  如果不是最后一个数字，则循环回到前面 */
DISPLAY:    movi    r3, 8
            movia   r4, HEX
            movia   r2, DISP_DATA
DLOOP:      ldbio   r5, 4(r2)         /*  通过测试 DOUT 标志来检查 */
            andi    r5, r5, 4         /*  显示器是否准备就绪 */
            beq     r5, r0, DLOOP
            ldb     r6, (r4)          /*  获取下一个 ASCII 字符 */
            stbio   r6, (r2)          /*  并将其发送给显示器 */
            subi    r3, r3, 1         /*  递减计数器 */
            addi    r4, r4, 1         /*  递增字符指针 */
            bgt     r3, r0, DLOOP     /*  循环，直到所有的字符都显示完毕 */
            下一条指令

            .org    1000
HEX:        .skip   8                 /*  ASCII 编码数字的存储空间 */
TABLE:      .byte   0x30,0x31,0x32,0x33 /*
            .byte   0x34,0x35,0x36,0x37 /*  ASCII 码转换表 */
            .byte   0x38,0x39,0x41,0x42
            .byte   0x43,0x44,0x45,0x46
```

图 B-19 列 B.14 的程序

## 习题

[E] B.1 写一个程序，计算表达式 SUM = 580 + 68 400 × 80 000。

[E] B.2 写一个程序，计算表达式 ANSWER = A × B + C × D。

[M] B.3 写一个程序，在一个含有 $n$ 个 32 位整数的列表中找出所包含的负整数的个数，并将该计数保存在单元 NEGNUM 中。$n$ 保存在存储单元 N 中，列表中的第一个整数保存在单元 NUMBERS 中。在程序中包含必要的汇编指示（assembler directive）和一个样本列表，列表中含有 6 个数字，其中一些是负数。

[E] B.4 为图 B-3 中的程序写一个如图 B-8 所示风格的汇编语言程序。假设采用图 2-10 所示的数据布局。

[M] B.5 写一个 Nios II 程序来解决第 2 章的习题 2.10 中的问题。

[E] B.6 写一个 Nios II 程序来解决第 2 章中例 2.5 所描述的问题。

[M] B.7 写一个 Nios II 程序来解决第 3 章中例 3.5 所描述的问题。

[E] B.8 写一个 Nios II 程序来解决第 3 章中例 3.6 所描述的问题。

[E] B.9 假设 TABLE 的地址是 0x10100，写一个 Nios II 程序来解决第 3 章中例 3.6 所描述的问题。

[E] B.10 写一个程序，在视频显示器的一行上以十六进制形式显示主存中 10 个字节的内容。该字节串在存储器中的起始位置是 LOC。每个字节将显示为两个十六进制字符。连续字节应以空格分隔。

[M] B.11 假设存储单元 BINARY 中包含一个 16 位的模式。我们希望在一个具有图 3-3 所示接口的显示设备上将这些位显示为 0 和 1 组成的字符串。写一个程序完成这个任务。

[M] B.12 使用图 3-17 中的七段显示器和图 3-14 中的定时器电路，写一个程序，显示重复序列 0、1、2、…、9、0、…中的十进制数字，每个数字显示一秒钟。假设定时器电路中的计数器是由 100 MHz 的时钟驱动的。

[D] B.13 使用两个图 3-17 所示的七段显示器和图 3-14 所示的定时器电路，写一个程序，显示重复序列 0、1、2、…、98、99、0、…中的数，每个数显示一秒钟。假设定时器电路中的计数器是由 100 MHz 的时钟驱动的。

[D] B.14 写一个程序，计算实时时钟时间并以小时（0 ~ 23）和分钟（0 ~ 59）的形式显示时间。显示器包括 4 个图 3-17 所示的七段显示设备。还有一个具有图 3-14 所示接口的定时器电路，其计数器是由 100 MHz 的时钟驱动的。

[M] B.15 写一个 Nios II 程序来解决第 2 章的习题 2.22 中的问题。

[568] [D] B.16 写一个 Nios II 程序来解决第 2 章的习题 2.24 中的问题。

[M] B.17 写一个 Nios II 程序来解决第 2 章的习题 2.25 中的问题。

[M] B.18 写一个 Nios II 程序来解决第 2 章的习题 2.26 中的问题。

[M] B.19 写一个 Nios II 程序来解决第 2 章的习题 2.27 中的问题。

[M] B.20 写一个 Nios II 程序来解决第 2 章的习题 2.28 中的问题。

[M] B.21 写一个 Nios II 程序来解决第 2 章的习题 2.29 中的问题。

[M] B.22 写一个 Nios II 程序来解决第 2 章的习题 2.30 中的问题。

[M] B.23 写一个 Nios II 程序来解决第 2 章的习题 2.31 中的问题。

[D] B.24 写一个 Nios II 程序来解决第 2 章的习题 2.32 中的问题。

[D] B.25 写一个 Nios II 程序来解决第 2 章的习题 2.33 中的问题。

[M] B.26 写一个 Nios II 程序来解决第 3 章的习题 3.19 中的问题。

[M] B.27 写一个 Nios II 程序来解决第 3 章的习题 3.21 中的问题。

569 ~ 570 [D] B.28 写一个 Nios II 程序来解决第 3 章的习题 3.23 中的问题。

[D] B.29 写一个 Nios II 程序来解决第 3 章的习题 3.25 中的问题。

# ColdFire 处理器

## 附录目标

在本附录中，你将学习 ColdFire 处理器，它是 CISC 风格体系结构的代表。本附录主要讨论：

- 存储器组织和寄存器结构
- 寻址方式和指令类型
- 计算类任务和 I/O 的示例程序
- 浮点运算的扩展



ColdFire 处理器是由飞思卡尔半导体（Freescale Semiconductor）公司生产的，该公司以前曾是摩托罗拉公司的一部分。ColdFire 是在 20 世纪 90 年代中期推出的，是从摩托罗拉公司的 68000 处理器衍生而来的。自推出以来，ColdFire 已经被多次扩展、增加了新的功能。实现 ColdFire 指令集的处理器可以作为预制芯片，也可以作为可在现场可编程门阵列（FPGA）芯片中实现的软件设计，这两种类型的实现常用于嵌入式应用中。

在 2.9 节和 2.10 节的讨论中，我们已经选择 ColdFire 作为 CISC 风格处理器设计的一个例子。ColdFire 包括许多可将存储器访问和算术逻辑运算相结合的指令。这些指令的新增功能使得我们可以使用更少的指令来执行计算任务，从而减少了程序占用的存储器空间。各种各样的寻址方式使得指令可以使用寄存器和存储器操作数。这增加了开发程序的灵活性，但也增加了用硬件实现指令集的复杂性。

本附录介绍了基本的 ColdFire 指令集（由飞思卡尔半导体在“ColdFire 系列程序员参考手册”中定义的修订版 A）。我们将描述存储器组织、寄存器结构，操作数的寻址方式以及多种指令类型。我们通过实现第 2 和第 3 章中所介绍的计算任务来说明 ColdFire 指令的使用。同时我们还将简要介绍对基本指令集的浮点扩展。

## C.1 存储器组织

ColdFire 的字长为 16 位。数据是以 8 位的字节（byte）、16 位的字（word）或 32 位的长字（longword）为单位进行处理的。地址是由 32 位组成的，且存储器是按字节编址的。从指令集的角度来看，存储器是按图 C-1 所示的方式进行组织的。对字或长字中的字节采用大端地址分配方式。

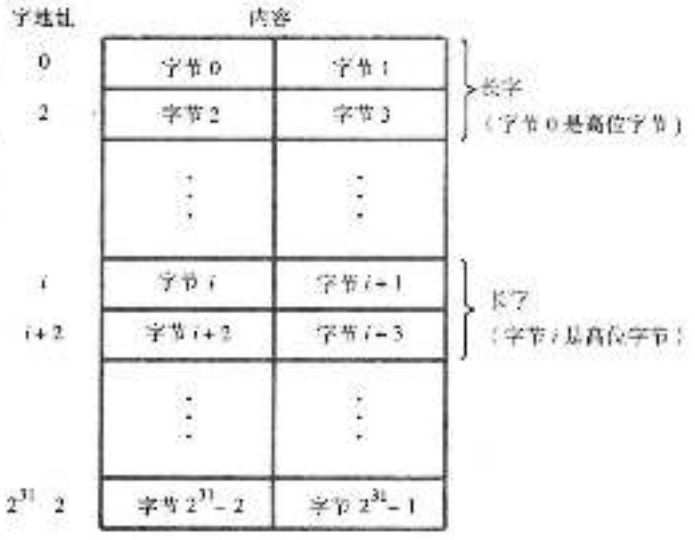



图 C-1 ColdFire 处理器中字节的大端存储器布局

## C.2 寄存器

图 C-2 显示了 ColdFire 的寄存器，其中有 8 个数据寄存器（data register）和 8 个地址寄存器（address

register），每个寄存器的长度都是 32 位。数据寄存器 D0 到 D7 可用作算术逻辑运算的通用寄存器或者其他用途。地址寄存器 A0 到 A7 主要用来保存确定操作数的存储器地址所需要的信息，其中一个地址寄存器 A7 专门用作处理器的堆栈指针（Stack Pointer，SP）。

还有一个状态寄存器（Status Register，SR），其中包含四个条件码标志：N、V、Z 和 C，这在 2.3.7 节中讨论过。这些标志可根据算术运算、逻辑运算或数据传输操作的结果进行设置或清除。其中还有一个称为 X（扩展）的额外标志，它的设置或清除方式同 C 标志一样，但它不像 C 标志那样受许多指令的影响。在对那些比数据寄存器的大小（32 位）大的数进行加法或减法运算时，它可用作一个扩展的进位输入/进位输出位，这将在 C.3.3 节中介绍。状态寄存器中的其余位用来控制处理器的行为，这将在 C.6 节中进行讨论。

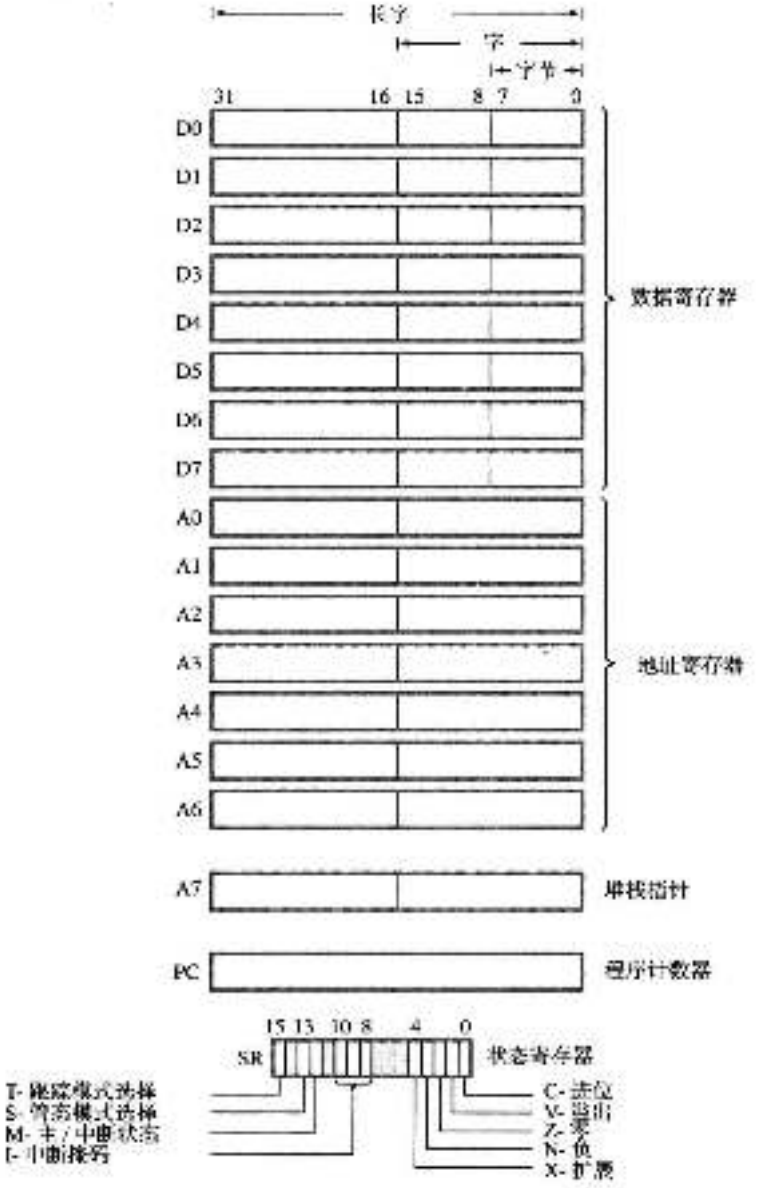



图 C-2 ColdFire 寄存器结构

## C.3　指令

ColdFire 指令由 16、32 或 48 位组成，在存储器中存储为一个、两个或三个连续的字。第一个字是指定所要执行操作的操作码（OP-code），该字也提供了一些寻址信息。如果某一给定类型的指令需要更多的寻址信息，便可以在一个或者两个扩展字（extension word）中提供。

大多数算术、逻辑和数据移动指令都有两个操作数，可用汇编语言写成如下形式：

OP　source, destination

该指令使用操作数执行 OP 操作，并将结果放到目的单元中，覆盖其原有的值。请注意，目的单元是第二个操作数，这个顺序与第 2 章中讨论的 CISC 风格的指令不同。

在汇编语言中，操作码是以表示所要执行操作的助记符以及表示数据操作数大小的长度说明符的形式给出的。长度说明符可以是 L、W 或 B，分别表示长字、字或字节。例如，指令

572 ↓ 574

ADD.L　LOC, D1

将存储单元 LOC 和处理器寄存器 D1 中的 32 位操作数相加，并将和放到 D1 中。并非所有长度的操作数都能用于所有的指令；加法指令只支持长字大小的操作数进行运算。

指令的汇编语言规范必须符合汇编程序所施加的约束（汇编程序用来生成可执行的机器语言代码）。飞思卡尔半导体公司提供的汇编程序对指令助记符和寄存器名称是大小写敏感的。ColdFire 的技术文档 [1] 一贯使用大写字符。为了符合这种表示风格，并使示例程序更容易阅读，我们使用大写字符来表示所有的指令助记符和寄存器名称。

### C.3.1　寻址方式

指令的操作数可能在处理器寄存器中，也可能在存储器中，或者作为立即值包含在指令中。以下为可用的寻址方式。

*立即方式*（immediate mode）——操作数是一个常量值，直接包含在指令中。可以指定四种大小的立即数。某些指令的操作码字中可以包含小的 3 位数。操作码字之后可以跟一个或两个扩展字，其中包含字节、字和长字操作数。

*绝对方式*（absolute mode）——操作数的存储器地址紧跟在指令的操作码字之后。这种方式有长模式和短模式两个版本。在长模式中，需要在两个扩展字中指定一个完整的 32 位地址。在短模式中，只需在一个扩展字中给出一个 16 位的值。这个值作为完整的 32 位地址的低 16 位。为确定高 16 位，需要将短值的符号位进行扩展。因此，短模式只能访问两个 32 KB 的存储区域：0 到 7FFF 或 FFFF8000 到 FFFFFFFF。

*寄存器方式*（register mode）——操作数在处理器寄存器 An 或 Dn 中，该寄存器在指令中指定。

*寄存器间接方式*（register indirect mode）——操作数的有效地址在地址寄存器 An 中，该寄存器在指令中指定。

*自增方式*（autoincrement mode）——操作数的有效地址在地址寄存器 An 中，该寄存器在指令中指定。操作数被访问后，An 的值递增 1、2 或 4，这取决于操作数是一个字节、一个字还是一个长字。

575

*自减方式*（autodecrement mode）——指令中指定的地址寄存器 An 的值首先递减 1、2 或 4，这取决于操作数是一个字节、一个字还是一个长字。然后，操作数的有效地址就由递减后的 An 值给出。

*基本变址方式*（basic index mode）——在指令中指定一个 16 位的有符号偏移量和一个地址寄存器 An。偏移量被符号扩展为 32 位，符号扩展后的偏移量与 32 位的 An 值之和为操作

数的有效地址。

完全变址方式（full index mode）——在指令中给出一个 8 位的有符号偏移量、一个地址寄存器 An 和一个变址寄存器 Rk（地址或数据寄存器）。操作数的有效地址是符号扩展后的偏移量与寄存器 An 中的值以及寄存器 Rk 中的有符号值相加的和。

基本相对方式（basic relative mode）——这种方式与基本变址方式相同，但是用程序计数器（PC）代替地址寄存器 An。

完全相对方式（full relative mode）——这种方式与完全变址方式相同，但是用程序计数器（PC）代替地址寄存器 An。

寻址方式及其汇编语法在表 C-1 中进行了总结。基本变址方式和完全变址方式对应于 2.4.3 节中讨论的变址寻址方式。两种相对方式是变址方式使用程序计数器而不使用地址寄存器的版本。在这些寻址方式中，偏移量表示所需操作数的存储单元与访问该操作数的指令之后的那个单元之间的距离。

表 C-1 ColdFire 寻址方式

| 寻址方式名称 | 汇编语法 | 寻址功能 |
|---|---|---|
| 立即方式 | #值 | 操作数 = 值 |
| 短绝对方式 | 值 | EA= 符号扩展后的 WValue |
| 长绝对方式 | 值 | EA= 值 |
| 寄存器方式 | Rn | EA=$R_n$，操作数 =[$R_n$] |
| 寄存器间接方式 | (An) | EA=[$A_n$] |
| 自增方式 | (An)+ | EA=[$A_n$]; 递增 $A_n$ |
| 自减方式 | -(An) | 递减 $A_n$，EA=[$A_n$] |
| 基本变址方式 | WValue(An) | EA = WValue + [$A_n$] |
| 完全变址方式 | BValue(An, Rk) | EA = BValue + [$A_n$] +[$R_k$] |
| 基本相对方式 | WValue(PC) | EA = WValue + [PC] |
| 完全相对方式 | BValue(PC, Rk) | EA = BValue + [PC] + [$R_k$] |

EA= 有效地址

Value= 明确给出的或用标签表示的数

BValue=8 位的值

WValue=16 位的值

$A_n$= 地址寄存器

$R_n$= 地址寄存器或数据寄存器

[576]

最后，重要的是要注意，并非所有指令都支持所有的寻址方式或所有的操作数大小。不同类别的指令在寻址方式和操作数大小上的一些限制将在本附录后面部分进行说明。ColdFire 处理器的技术文档对一些指令的有效组合提供了完整的细节信息 [1]。

## C.3.2 Move 指令

Move（传送）指令在存储器或 I/O 接口与处理器寄存器之间传输数据，传输的方向是从源端到目的端。如果 Move 指令所传输的值是零或者负数，那么该值会使得状态寄存器中的条件标志位 Z 或 N 被置为 1。该指令可使用所有的三种大小的操作数，源操作数可使用所有可用的寻址方式。至于目的操作数，则允许使用除了立即方式和相对方式之外的其他所有寻址方式。为确保指令的长度不超过三个字，不允许使用源操作数和目的操作数的寻址方式的某些组

合。例如，源操作数和目的操作数不能都使用绝对方式（短模式或长模式）。

为了说明如何指定不同的寻址方式和不同的操作数大小，考虑以下指令：

```
MOVE.L   D0, (A2)
```

它将一个 32 位的值从寄存器 D0 写到其地址由寄存器 A2 的内容给出的存储单元中。同样，指令

```
MOVE.B   CHARACTER, D3
```

将一个 8 位的值从存储单元 CHARACTER 传输到寄存器 D3 中。该传输操作只改变寄存器 D3 的低位字节；其余的位不会受到影响。

寄存器之间也可以进行数据传输，如

```
MOVE.W   D5, D7
```

它将寄存器 D5 中的低 16 位值传输到寄存器 D7 的低位中，D7 的高位不受影响。

也可以在存储单元之间进行直接的数据传输，如

```
MOVE.L   (A2), 16(A4)
```

它将一个 32 位的值从地址由寄存器 A2 的内容给出的存储单元传输到有效地址为寄存器 A4 的值加上 16 的存储单元中。

可通过以下指令将一个立即值装入一个寄存器或存储单元中：

```
MOVE.L   #$2A4C80, D7
```

[577]

该指令将指定的十六进制值装入寄存器 D7 中。需要注意的是，'S' 字符用来表示一个十六进制值。因为有 L 这个大小说明符，所以目的操作数的所有 32 位都会受到影响，因此 D7 中的结果值将是 002A4C80。图 C-3 显示了该指令是如何存储在存储器中的。操作码字表明这是一条 MOVE 指令，并指定了操作数的大小。它还指定了源操作数和目的操作数的寻址方式，其中包括指定寄存器 D7 是目的单元。操作码字之后的两个扩展字包含源操作数的 32 位立即值。




图 C-3　存储器中的指令 MOVE.L #$2A4C80, D7

MOVE 指令有两个专门的版本。MOVEQ（Move Quick，快速传送）指令在源操作数是一个足够小的、可以在 8 位中容纳的立即值并且目的操作数是一个数据寄存器的时候使用。这条指令的长度只有一个字的大小。MOVEA 指令在目的单元是一个地址寄存器的时候使用。MOVEA 指令只允许使用字和长字操作数。状态寄存器中的条件码不受这条指令的影响。ColdFire 汇编程序会在适当的地方用 MOVEQ 或 MOVEA 指令取代正常的 MOVE 指令。

还有一条指令 MOVEM（Move Multiple Registers，多个寄存器传送），它执行包含数个寄存器的多次传送，可用于 C.3.7 节所讨论的子程序链接中。

### C.3.3　算术指令

这一类指令包括算术运算以及比较、符号扩展、取反和清空操作。操作数可以在存储器中、数据寄存器中或者包含在指令中作为立即值。本节讨论的所有算术指令，除了其中的四条之外，其他的都只允许长字大小的操作数。除一条指令外，其他指令都需要至少一个寄存器操作数。

#### 1. 加法、减法、比较和取反

这种类型的指令有：

- ADD.L（加）
- ADDI.L（加立即数）
- ADDA.L（加地址）
- SUB.L（减）
- SUBI.L（减立即数）
- SUBA.L（减地址）
- CMP.L（比较）
- CMPI.L（比较立即数）
- CMPA.L（比较地址）
- NEG.L（取反；一个数据寄存器操作数）
- ADDX.L（扩展加；两个数据寄存器操作数）
- SUBX.L（扩展减；两个数据寄存器操作数）
- NEGX.L（扩展取反；一个数据寄存器操作数）

[578]

ADD 和 SUB 指令对两个长字操作数执行指定的算术运算并将结果放在目的单元中。SUB 指令将目的操作数减去源操作数。根据所得的结果，所有的条件码标志都会受到影响。

CMP 指令用于比较长字的值。目的操作数必须在寄存器中。该指令与 SUB 指令执行相同的操作，但它不会改变目的操作数的值。除了 X 标志外，该指令的结果会影响其他所有的条件码标志。

还有专用版本的 ADD、SUB 和 CMP 指令，用于两种情况：当源操作数是一个立即值（ADDI、SUBI 和 CMPI）时和当目的操作数是一个地址寄存器（ADDA、SUBA 和 CMPA）时。ColdFire 汇编程序会在适当的地方用这些指令的专用版本替换其正常版本。

考虑下面的例子，指令

```
ADD.L   D4, (A1)+
```

将 D4 寄存器中的长字加上由寄存器 A1 给定的存储单元中的长字，并将和放到同一存储单元中。寄存器 A1 中的值递增 4。指令

```
SUBI.L   #256, D7
```

将寄存器 D7 的内容减去 256，并将其结果放到 D7 中。注意，指令

```
CMPI.L   #256, D7
```

执行相同的减法运算，但不改变寄存器 D7 的内容。除了 X 标志外，其他所有的条件码标志都以与 SUB 指令同样的方式受到影响。

NEG.L 指令用来对数据寄存器中的长字操作数取反。取反操作是通过用 0 减去数据寄存器的值来实现的。根据所得的结果，所有的条件码标志都会受到影响。指令

[579]

```
NEG.L   D3
```

将寄存器 D3 中的长字取反并用取反后的值覆盖原来的值。

为了方便对大于 32 位的值进行算术运算，ADDX.L、SUBX.L 和 NEGX.L 指令将 X 标志用作一个进位输入位。所有的操作数都必须在数据寄存器中。所有条件码标志都会受到影响。例如，图 C-4 显示了 ADDX 指令是如何用来将不能在 32 位寄存器中容纳的大数相加的。将两个十六进制值 10A72C10F8 和 4A5C00FE04 相加。10A72C10F8 的低位和高位被分别装入寄存器 D2 和 D3 中。同样，4A5C00FE04 的低位和高位被分别装入寄存器 D4 和 D5 中。ADD.L 指令用来将低 32 位相加，产生进位输出 1，使得 X 标志和 C 标志被置为 1。然后，当高位部分相加时，ADDX.L 指令将 X 标志的新值作为进位输入位。相加和的低位和高位分别保存在寄

存器 D4 和 D5 中。C 标志和 X 标志都会受到 ADDX.L 指令结果的影响，但在这个例子中，它们将不会被使用，因为所需的 64 位加法已经完成了。

```
MOVE.L   #$A72C10F8, D2   D2 中的值为 A72C10F8
MOVE.L   #$10, D3         D3 中的值为 10
MOVE.L   #$5C00FE04, D4   D4 中的值为 5C00FE04
MOVE.L   #$4A, D5         D5 中的值为 4A
ADD.L    D2, D4           将低 32 位相加：根据进位输出位设置 X 和 C 标志
ADDX.L   D3, D5           以 X 标志为进位输入位，将高位部分相加
```

图 C-4　使用 ADDX 指令将大于 32 位的数相加的程序

2. 乘法

使用 MULS 和 MULU 指令可分别对有符号和无符号操作数执行乘法运算。操作数的大小可以是字或长字。目的操作数必须在数据寄存器中。

指令

MULS.W　#1340, D5

将 1340 与寄存器 D5 中的低 16 位有符号值相乘，并将 32 位的乘积放到 D5 中。

指令

MULS.L　D2, D5

将寄存器 D2 和 D5 中的长字相乘，并将乘积截取为 32 位，放到 D5 中。

MULU.W 和 MULU.L 指令对无符号操作数执行相同的操作。作为乘法运算的结果，N 和 Z 条件码标志会根据乘积的值来置为 1 或清为零，而 V 和 C 标志被清为零。 580

图 C-5 显示了对同一对字操作数 SFFFF 和 S0001 进行有符号和无符号乘法运算时获得的不同结果。图 C-5a 中的 MULS.W 指令将寄存器 D2 中 SFFFF 的低位字的值视为 −1。有符号乘法运算的长字结果为 SFFFFFFFF，表示 −1，并将 N 标志置为 1。而图 C-5b 中的 MULU.W 指令将 $FFFF 看做无符号数 65535，且无符号乘法运算的长字结果为 $0000FFFF，表示 65 535，并将 N 标志清零。

```
MOVE.W   #$FFFF, D2   D2 的低位字被视为 −1
MOVE.W   #$0001, D3   D3 的低位字包含 1
MULS.W   D2, D3       D3 中的有符号长字结果为 −1 或
                        $FFFFFFFF，因此 N 标志被置为 1
```

a) −1 × 1 = −1 的有符号计算

```
MOVE.W   #$FFFF, D2   D2 的低位字被视为 65 535
MOVE.W   #$0001, D3   D3 的低位字包含 1
MULU.W   D2, D3       D3 中的无符号长字结果为 65 535 或
                        $0000FFFF，因此 N 标志被清零
```

b) 65 535 × 1 = 65 535 的无符号计算

图 C-5　有符号乘法与无符号乘法的对比

3. 除法

使用 DIVS 和 DIVU 指令可分别对有符号和无符号操作数执行除法运算。操作数的大小可以是字或长字。目的操作数必须在数据寄存器中。

指令

DIVS.W　#2500, D1

将寄存器 D1 中的 32 位值除以 16 位的立即操作数 2500。16 位的商被放到 D1 的低位中，16

位的余数被放到D1的高位中。

指令

DIVS.L D2, D1

将D1中的值除以D2中的值，并将商放到D1中，余数被丢弃。

DIVU.W和DIVU.L指令对无符号操作数执行相同的操作。作为除法运算的结果，N和Z条件码标志会根据商的值来置为1或清为零，而V和C标志被清为零。

由于DIVS.L和DIVU.L操作中的余数被丢弃，所以还有两条其他的指令，用来在需要的时候获得余数。指令

REMS.L D2, D1:D4

将D1中的值除以D2中的值，将32位的余数放到寄存器D4中，并使D1的值保持不变。因此，如果该指令后跟有以下指令：

DIVS.L D2, D1

则余数和商都可以获得。需要注意的是，由冒号界定的第三个操作数必须在REMS.L指令中
581 指定。

REMU.L指令与REMS.L指令执行相同的操作，但它是对无符号操作数进行操作，同样也需要三个操作数。

4. 其他算术指令

当增加表示一个数的位数时，EXT（符号扩展）指令可用来对符号位进行扩展。它只有一个操作数且必须在数据寄存器中。所指定的大小决定该操作如何执行。EXT.L指令将低位字符号扩展为一个长字，EXT.W指令将低位字节符号扩展为一个字，而EXT.B指令将低位字节符号扩展为一个长字。对于所有这三条指令来说，N和Z条件码标志都会受到指令结果的影响，且V和C标志被清为零。

CLR（清除）指令可用来清除指定操作数中的位。大小说明符说明将要清除的是一个长字、一个字还是一个字节。该操作数可能在存储器中，也可能在数据寄存器中。该指令会将Z标志置为1，并将N、V和C标志清为零。

## C.3.4 Branch与Jump指令

条件转移指令具有如下的格式：

Bcc LABEL

其中cc指定条件码。表C-2总结了条件码和一些要测试的条件码标志组合。例如，如果Z标志被设置为1，则BEQ（Branch-if-equal，如果等于则转移）指令会导致一个转移，而BGE（Branch-if-greater-than-or-equal，如果大于等于则转移）指令则依赖于N和V标志的状态。还有一条无条件转移指令BRA，始终会产生转移。

转移指令指定一个有符号的偏移量，该偏移量与程序计数器中的值相加，以确定目标地址。根据转移指令之后的那个单元与转移目标之间的距离，可提供三种类型的偏移量。第一种类型中，当转移距离在±127字节之内时，在操作码字中

表C-2 Bcc指令的条件码

| 条件后缀cc | 名 称 | 测试条件 |
|---|---|---|
| HI | 高 | C∨Z=0 |
| LS | 低或相同 | C∨Z=1 |
| CC | 清除进位 | C=0 |
| CS | 设置进位 | C=1 |
| NE | 不等于 | Z=0 |
| EQ | 等于 | Z=1 |
| VC | 清除溢出 | V=0 |
| VS | 设置溢出 | V=1 |
| PL | 正 | N=0 |
| MI | 负 | N=1 |
| GE | 大于等于 | N⊙V=0 |
| LT | 小于 | N⊕V=1 |
| GT | 大于 | Z∨(N⊙V)=0 |
| LE | 小于等于 | Z∨(N⊕V)=1 |

包含一个较小的 8 位偏移量。第二种类型中，当转移距离高达 ±32K 字节时，在操作码字之后的扩展字中指定一个较大的 16 位偏移量。第三种类型中，当到转移目标的距离超过 16 位偏移量所支持的范围时，可在两个扩展字中指定一个 32 位的偏移量。

JMP（跳转）指令执行一个无条件跳转操作，跳转到下一条要执行指令的位置。用一个单一的操作数来指定目标地址。JMP 指令中可使用的寻址方式有绝对寻址、间接寻址、基本和完全变址寻址方式，以及基本和完全相关寻址方式。例如，指令

JMP　(A3)

跳转到地址寄存器 A3 的内容所指定的位置。

为了说明转移指令在一个循环中的使用，图 C-6 给出了图 2-26 中将一个列表中数字相加的循环程序的 ColdFile 版本。ADD.L 指令中使用自增寻址方式来自动增加指向列表项的指针。BGT（Branch-if-greater-than，如果大于则转移）指令用来检查根据 SUBQ.L 指令的执行结果而设置或清除的条件码标志。SUBQ.L 指令用来递减列表中剩余的待处理元素个数的计数。该指令是立即值可用 3 位表示时所使用的 SUBI 指令的另一个版本。

```
        MOVEA.L   #NUM1, A2     将地址 NUM1 放到 A2 中
        MOVE.L    N, D1         将列表的项数 n 放到 D1 中
        CLR.L     D0
LOOP:   ADD.L     (A2)+, D0     将和累加到 D0 中
        SUBQ.L    #1, D1
        BGT       LOOP
        MOVE.L    D0, SUM       结束时保存结果
```

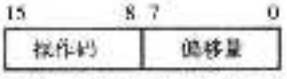
图 C-6　图 2-26 中将列表中数字相加程序的 ColdFire 版本

图 C-7 显示了具有小偏移量的条件转移指令的格式以及图 C-6 的程序其循环中的三条指令是如何存储在存储器中的。BGT 指令需要有一个负的偏移量，可由以下算式计算得到：

582
↓
583

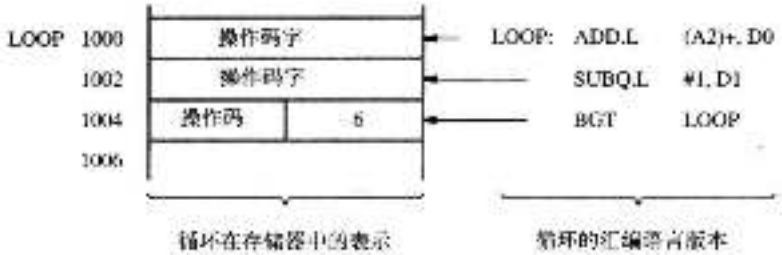
偏移量 = 目标地址 − [PC]

目标地址为 1000。在 BGT 指令执行时，PC 的值将是 1006，这是因为提取了 BGT 指令的操作码字之后 PC 的值会增加。因此，偏移量为 1000 − 1006 = −6。8 位的偏移量足以存储这个较小的值，这意味着该 BGT 指令只需要使用一个字来进行编码。




转移地址 = [更新后的 PC 值] + 偏移量

a）加偏移量转移指令的格式




在计算转移地址时，[PC] = 1006

转移地址 = 1006 − 6 = 1000

b）在图 C-6 的循环中使用转移指令的例子

图 C-7　转移指令的格式及其在存储器中的表示

584

图 C-8 给出了一个更复杂的例子，该例子说明了指令的一些其他方面的信息。它是图 2-11 中计算一组学生参加三项测验得到的所有分数之和程序的 ColdFire 版本。ADD 指令中可使用基本变址寻址方式访问存储器操作数，这样就省却了图 2-11 中循环体内的 Load 指令。

```
        MOVEA.L   #LIST, A2     获取地址 LIST
        CLR.L     D3
        CLR.L     D4
        CLR.L     D5
        MOVE.L    N, D6         载入 n 值
LOOP:   ADD.L     4(A2), D3     加上当前学生的测验 1 分数
        ADD.L     8(A2), D4     加上当前学生的测验 2 分数
        ADD.L     12(A2), D5    加上当前学生的测验 3 分数
        ADDA.L    #16, A2       递增指针
        SUBQ.L    #1, D6        递减计数器
        BGT       LOOP          如果没有结束，循环回到前面
        MOVE.L    D3, SUM1      保存测验 1 的总分数
        MOVE.L    D4, SUM2      保存测验 2 的总分数
        MOVE.L    D5, SUM3      保存测验 3 的总分数
```

图 C-8 图 2-11 中求测验分数总和程序的 ColdFire 版本

### C.3.5 逻辑指令

逻辑指令需要长字操作数，并且至少有一个操作数必须在数据寄存器中。有如下的逻辑指令：

- AND.L（按位逻辑与）
- ANDI.L（按位逻辑与；源操作数是一个立即值）
- OR.L（按位逻辑或）
- ORI.L（按位逻辑或；源操作数是一个立即值）
- EOR.L（按位逻辑异或；源操作数必须在数据寄存器中）
- EORI.L（按位逻辑异或；源操作数是一个立即值）
- NOT.L（按位求补；单一一个数据寄存器操作数）

根据指令的结果，所有逻辑指令都会影响 N 和 Z 条件码标志；V 和 C 标志被清为零。

例如，指令

585

```
ANDI.L   #$FF, D5
```

对十六进制值 000000FF 和数据寄存器 D5 中的长字执行按位逻辑“与”操作，并将结果放到 D5 中。指令

```
EOR.L   D3, (A6)+
```

对寄存器 D3 中的长字和由寄存器 A6 的内容给出的存储器地址中的长字执行按位逻辑“异或”操作，并将结果放到该存储器地址中。然后，寄存器 A6 的内容递增 4。

### C.3.6 Shift 指令

Shift（移位）指令有两个操作数：目的操作数必须在数据寄存器中，该寄存器保存将被移位的长字；源操作数指定要移动的位数，它要么是一个立即数，要么是一个数据寄存器中的内容。立即值必须是 1 ~ 8 的数，被编码在指令中。如果要移动的位数在数据寄存器中，那么它的值会被解释为对 64 取模，尽管数据寄存器的大小只是 32 位。

对于所有的移位指令来说，从目的数据寄存器移出的最后一位被复制到 C 和 X 条件码标志中。每个被移动到目的数据寄存器中的位都是 0，除了算术右移时 $b_{31}$ 位上的符号值会保留。

图 2-23 给出了每一种移位的情况。根据移位后的最终结果，N 和 Z 标志会受到影响，V 标志被清为零。

可用的移位指令有：

- LSL.L（逻辑左移）
- LSR.L（逻辑右移）
- ASL.L（算术左移）
- ASR.L（算术右移；符号位被保留）

下面的例子说明移位操作之间的差异。假设数据寄存器 D4 中的初值为十六进制数 80000000（$b_{31}$ 位是 1，其他位都是 0）。指令

```
LSR.L   #6, D4
```

将 D4 中的值右移 6 位，并在左端插入 0。D4 中的结果为十六进制数 02000000。因为从 D4 移出的最后一位是 0，所以 C 和 X 标志被清为零。

如果 D4 中的初值相同，指令

```
ASR.L   #6, D4
```

也将 D4 中的值右移 6 位，但保留 $b_{31}$ 位上的符号位。这样，移到数据寄存器最左端的位都是 1。D4 中的结果为十六进制数 FE000000。因为被移出的最后一位是 0，所以 C 和 X 标志被清为零。

586

**例 C.1**

考虑图 2-24 中的 BCD 数字打包程序。该程序的 ColdFire 版本如图 C-9 所示。在连续的存储器字节单元中的两个 ASCII 编码字符被放到寄存器 D0 和 D1 中。LSL 指令将 D0 中的第一个字节左移 4 位，用零填充低四位。ANDI 指令将寄存器 D1 中的所有高位都清为零。随后，所需 BCD 码的 4 位模式通过 OR 指令合并到 D1 中。最后，我们感兴趣的字节，寄存器 D1 最右边的字节，被放到存储单元 PACKED 中。需要注意的是，LSL、AND 和 OR 指令会影响寄存器操作数的所有 32 位，但我们所需的打包字节会被正确地生成，并存放在 D1 的最低 8 位中。

```
MOVEA.L   #LOC, A0      A0 指向两个连续的字节
MOVE.B    (A0)+, D0     将第一个字节装入 D0 中
LSL.L     #4, D0        左移 4 位
MOVE.B    (A0), D1      将第二个字节装入 D1 中
ANDI.L    #$F, D1       将 D1 中所有的高位清零
OR.L      D0, D1        连接数字
MOVE.B    D1, PACKED    保存结果
```

图 C-9 逻辑和移位指令在 BCD 数字打包中的使用

## C.3.7 子程序链接指令

ColdFire 提供了指令和处理器堆栈以支持子程序和 2.7 节所述方式的参数传递。地址寄存器 A7 作为堆栈指针，它必须始终有一个值，且是长字对齐的，即为 4 的倍数。这个寄存器不用于任何其他的用途。堆栈的增长方向为存储器地址降低的方向。寄存器 A7 中的堆栈指针值减去 4 可将新的信息压入栈中，增加 4 便可从栈中弹出信息。

有两个子程序调用指令 BSR（branch-to-subroutine，转移到子程序）和 JSR（jump-to-subroutine，跳转到子程序）。BSR 指令与其他的转移指令一样，可用 8 位、16 位或 32 位的偏移量来指定子程序的地址。JSR 指令可以使用绝对寻址、间接寻址、基本和完全变址寻址，或者基本和完全相对寻址方式来产生目标地址。BSR 和 JSR 指令都会自动地将返回地址压入处理器堆栈中，而不是像 2.7 节所描述的那样保存在一个链接寄存器中。

在子程序的末尾，RTS（return-from-subroutine，从子程序返回）指令用来返回到调用程序。RTS 指令从栈顶弹出返回地址并将其装入程序计数器中。

587

### 1. 参数传递

2.7.2 节讨论了两种不同的参数传递方法，这两种方法通过使用图 2-26 中将列表中数字相加的示例程序进行了说明。图 C-6 给出了该程序的 ColdFire 版本，它将是下面所讨论内容的基础。

图 C-10 是图 2-17 中程序（通过寄存器传递参数）的 ColdFire 版本。通过将数值列表的起始地址放到寄存器 A2 中来将其传递给子程序。列表中的元素个数通过寄存器 D1 传递给子程序。在将列表中的所有元素相加之后，子程序将相加和返回到寄存器 D0 中。

```
调用程序
        MOVEA.L   #NUM1, A2     将地址 NUM1 放到 A2 中
        MOVE.L    N, D1         将列表中的元素个数 n 放到 D1 中
        BSR       LISTADD       调用子程序 LISTADD
        MOVE.L    D0, SUM       将和保存在 SUM 中
        下一条指令
        :

子程序
LISTADD: CLR.L    D0
LOOP:    ADD.L    (A2)+, D0     将和累加到 D0 中
         SUBQ.L   #1, D1
         BGT      LOOP
         RTS
```

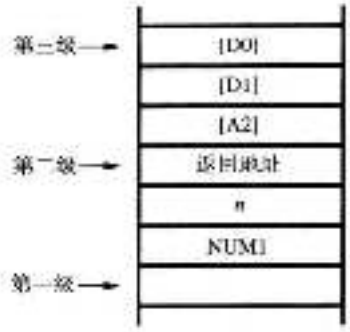
图 C-10 将图 C-6 中的程序写成一个子程序，通过寄存器传递参数

图 C-11 显示了图 2-18 中程序（通过处理器堆栈传递参数）的 ColdFire 版本。在调用子程序之前，列表的起始地址和元素个数被压入栈中。子程序会从栈中取出这些值。子程序完成后，将返回的结果放到堆栈中，以供调用程序取回。

```
假设栈顶初始时处于下面图表中的第一组
调用程序
        MOVE.L    #NUM1, -(A7)    将参数压入栈中
        MOVE.L    N, -(A7)
        BSR       LISTADD
        MOVE.L    4(A7), D0       从栈中获取结果
        MOVE.L    D0, SUM         保存结果
        ADDA.L    #8, A7          恢复栈顶
        :
子程序
LISTADD: ADDA.L   #-12, A7        调整堆栈指针以分配空间
         MOVEM.L  D0-D1/A2, (A7)  保存寄存器 D0、D1 和 A2
         MOVE.L   16(A7), D1      将计数器初始化为 n
         MOVEA.L  20(A7), A2      初始化指向列表的指针
         CLR.L    D0              将和初始化为 0
LOOP:    ADD.L    (A2)+, D0       加上列表中的项
         SUBQ.L   #1, D1
         BGT      LOOP
         MOVE.L   D0, 20(A7)      将结果放到栈中
         MOVEM.L  (A7), D0-D1/A2  恢复寄存器
         ADDA.L   #12, A7         调整堆栈指针以释放空间
         RTS
```

a）调用程序和子程序

图 C-11 将图 C-6 中的程序写成一个子程序，通过堆栈传递参数




b）堆栈在不同时刻的内容

图 C-11 （续）

图 C-11 中的程序也说明了如何使用 MOVEM（Move Multiple Registers，移动多个寄存器）指令向或从存储器中的连续单元保存或恢复寄存器的值。MOVEM 指令有两个操作数，并且操作数的顺序决定了寄存器的值是写入存储器还是从存储器中读出。其中一个操作数是单个寄存器的列表或者是寄存器范围的列表，如 D0-D1/A2。另一个操作数是存储器中值的起始地址，它必须通过间接寻址方式或基本变址寻址方式来指定。MOVEM 指令从给定的起始单元开始沿着地址增加的方向来读写存储器。

在子程序的开头，ADDA.L 指令将图 C-11b 中的堆栈指针从第一级调整到第三级。这为保存三个寄存器的内容分配了空间。然后 MOVEM 指令将寄存器 D0、D1 和 A2 的内容写入所分配的空间中。在子程序的末尾，用一条类似的 MOVEM 指令从栈中读出这些值并装回到寄存器中，但该指令的源/目的操作数的顺序是相反的。从子程序返回之前的最后一步是将堆栈指针从第三级再调整到第二级，以释放之前为保存寄存器的值而分配的空间。

588
↓
589

**2. 用于子程序嵌套的栈结构**

对于使用 BSR 或 JSR 指令的嵌套子程序调用，每次调用的返回地址都被自动压入到处理器栈中。随后，当嵌套序列中的子程序完成时，就会执行 RTS 指令。每条 RTS 指令执行时，会从栈中弹出相应的返回地址。

正如 2.7.3 节所讨论的那样，栈结构（stack frame）为嵌套序列中的每一个子程序在存储器中提供了工作空间。除了堆栈指针 A7 外，可将另一个地址寄存器用作了程序内的结构指针（frame pointer）来确定当前的栈结构。

ColdFire 提供了两条特殊的指令来管理栈结构。指令

LINK　Ai, #disp

用来在子程序的开头分配一个栈结构。它执行以下操作：

1）将寄存器 Ai 的内容（结构指针）压入到处理器栈中

2）将堆栈指针 A7 的内容复制到结构指针 Ai 中

3）将指定的位移值加到堆栈指针 A7 中

位移值是一个负数以使堆栈增长，来为子程序中的局部变量分配额外的空间。这些变量可通过使用寄存器 Ai（结构指针）的基本变址或完全变址寻址方式来访问。位移量还可以用来为保存子程序所使用的寄存器值分配空间。

第二条特殊指令

UNLK　Ai

用来在子程序的末尾释放栈结构。它与 LINK 指令的操作相反。它将 Ai 的内容装入 A7，这样

就将栈顶降低到其原始位置，即其加上位移值之前的位置。然后，该指令从栈中弹出所保存的寄存器 Ai 的内容，并将其放回到 Ai 中。

为了说明 LINK 和 UNLK 指令的使用，图 C-12 给出了列 2-21 中具有嵌套子程序调用的程序的 ColdFire 代码。子程序 SUB1 和 SUB2 的栈结构如图 C-13 所示。执行流程如下：

- 调用程序将参数 param2 和 param1 压入栈中，以供子程序 SUB1 使用。当通过 BSR 指令调用 SUB1 时，返回地址 2014 被压入栈中。SUB1 的地址 2100 与 BSR 指令相隔的距离在 128 个字节之内，因此，可使用一个 8 位的偏移量，此时 BSR 指令只需要操作码字。

590

- 子程序 SUB1 从一条分配栈结构的指令

  LINK A6, # −16

  开始。该指令将 A6 的当前值保存到栈中，并将 A7 的值写入 A6 以定义新的结构指针。然后调整 A7 的值以在栈中分配足够数量的空间来保存 SUB1 所使用的 4 个寄存器的值。

| 存储单元 | | 指令 | | 注释 |
|---|---|---|---|---|
| **调用程序** | | | | |
| | | ⋮ | | |
| 2000 | | MOVE.L | PARAM2, −(A7) | 将参数放置到栈中 |
| 2006 | | MOVE.L | PARAM1, −(A7) | |
| 2012 | | BSR | SUB1 | |
| 2014 | | MOVE.L | (A7), RESULT | 保存结果 |
| 2020 | | ADDA.L | #8, A7 | 恢复栈的级数 |
| 2024 | | 下一条指令 | | |
| | | ⋮ | | |
| **第一个子程序** | | | | |
| 2100 | SUB1: | LINK | A6, #−16 | 设置结构指针，分配栈空间 |
| 2104 | | MOVEM.L | D0−D2/A0,(A7) | 保存寄存器 |
| | | MOVEA.L | 8(A6), A0 | 载入参数 |
| | | MOVE.L | 12(A6), D0 | |
| | | ⋮ | | |
| | | MOVE.L | PARAM3, −(A7) | 将一个参数放置到栈中 |
| 2160 | | BSR | SUB2 | |
| 2164 | | MOVE.L | (A7)+, D1 | 将 SUB2 的结果弹出到 D1 中 |
| | | ⋮ | | |
| | | MOVE.L | D2, 8(A6) | 将结果放置到栈中 |
| | | MOVEM.L | (A7), D0−D2/A0 | 恢复寄存器 |
| | | UNLK | A6 | 恢复结构指针，释放栈空间 |
| | | RTS | | 返回 |
| **第二个子程序** | | | | |
| 3000 | SUB2: | LINK | A6, #−8 | 设置结构指针，分配栈空间 |
| | | MOVEM.L | D0−D1,(A7) | 保存寄存器 |
| | | MOVE.L | 8(A6), D0 | 载入参数 |
| | | ⋮ | | |
| | | MOVE.L | D1, 8(A6) | 将结果放置到栈中 |
| | | MOVEM.L | (A7), D0−D1 | 恢复寄存器 |
| | | UNLK | A6 | 恢复结构指针，释放栈空间 |
| | | RTS | | 返回 |

图 C-12 嵌套子程序

- 使用 MOVEM 指令，将 SUB1 所使用的 4 个寄存器的当前值保存在栈中。
- 然后 SUB1 使用结构指针及基本变址寻址方式来从栈中取出 param1 和 param2 的值。执行一些计算后，SUB1 将 param3 压入栈中并调用子程序 SUB2，返回地址 2164 被压入栈中。子程序 SUB2 的地址 3000 与 BSR 指令相隔的距离超过 128 字节，因此需要在指令的一个扩展字中包含一个 16 位的偏移量。
- SUB2 以它自己的 LINK 指令开始，为一个新的栈结构分配空间，用来保存寄存器值。LINK 指令之后是用来将两个寄存器的值写入堆栈的 MOVEM 指令。然后从栈中取出 param3 的值。
- 执行一些计算后，SUB2 将结果放入栈中，覆盖 param3 的值。两个已保存的寄存器 D0 和 D1 从栈中恢复，UNLK 指令释放当前的栈结构并在返回到 SUB1 之前恢复以前的结构指针。

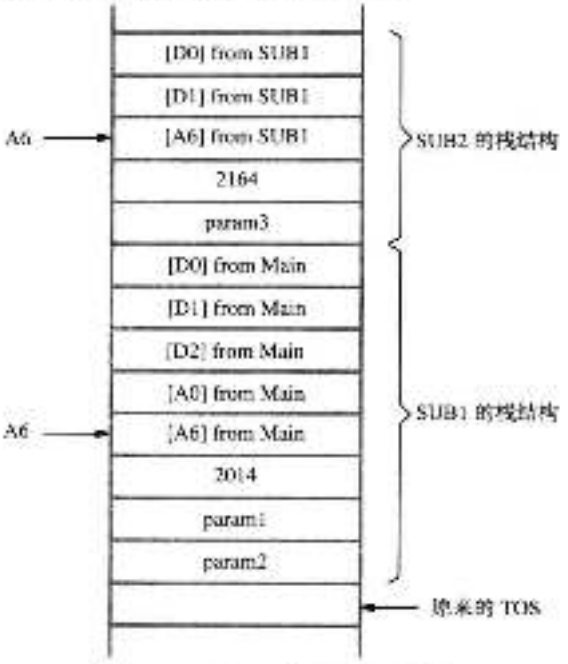



图 C-13　图 C-12 中程序的栈结构

- SUB1 在返回地址 2164 处恢复执行，从栈中弹出 SUB2 的结果。SUB1 完成其计算，并将结果放入栈中，覆盖 param1 的值。4 个已保存的寄存器从栈中恢复。UNLK 指令为返回到调用程序做准备。
- 主程序在返回地址 2014 处恢复执行，从栈中取出 SUB1 的结果。最后，ADDA 指令调整寄存器 A7 的值，使其指向最初的栈顶元素。最初的栈顶元素在图 C-13 中被标记为“原来的 TOS”。

## C.4　汇编指示

2.5 节讨论的汇编指示（assembler directive）可以用于 ColdFire 程序，只是在标记符号上有很小的差别。飞思卡尔半导体公司提供的汇编程序要求每个指示符都以一个句号开始，以区别于指令助记符。

- ORG 指示符指定了指令或数据块的起始地址。
- EQU 指示符将名字与数值关联起来。
- DC（定义常量）指示符可用来将数据常量插入到目标程序中。在一个 DC 指示符中可定义多个数据项。数据项的大小通过后缀 L、W 或 B 指明。例如，语句

```
.ORG    100
ITEMS:  .DC.B   23, $4F, %10110101
```

可将字节大小的十六进制值 17（$23_{10}$）、4F 和 B5 分别放置到存储单元 100、101 和 102 中。标签 ITEMS 被赋予值 100。注意，“$”字符表示十六进制值，“%”字符表示二进制值。

- DS（定义存储器）指示符可用来为数据保留一个未初始化的存储器块，用后缀来表示数据的大小。例如，语句

591 ' 592

```
ARRAY:   .DS.L   200
```

保留 200 个长字，并将标签 ARRAY 与第一个长字的地址关联起来。

[593] 汇编指示符的使用如图 C-14 所示，它对应于图 C-6 中将列表数值相加的程序。

```
        .ORG      1000          指令从地址 1000 处开始
        MOVEA.L   #NUM1, A2     将地址 NUM1 放到 A2 中
        MOVE.L    N, D1         将列表的元素 n 放到 D1 中
        CLR.L     D0
LOOP:   ADD.L     (A2)+, D0     将和累加到 D0 中
        SUBQ.L    #1, D1
        BGT       LOOP
        MOVE.L    D0, SUM       结束时保存结果

        .ORG      200           数据从地址 200 处开始
SUM:    .DS.L     1             为相加和保留一个长字
N:      .DC.L     150           列表中有 N=150 个长字
NUM1:   .DS.L     150           为 150 个长字保留存储器空间
```

图 C-14 对应于图 2-13 的 ColdFire 程序

## C.5 示例程序

在本节中，我们将给出第 2 章所描述的点积程序和字符串搜索程序的 ColdFire 版本。

### C.5.1 向量点积程序

图 2-28 中的程序计算两个向量 AVEC 和 BVEC 的点积。该程序的 ColdFire 版本如图 C-15 所示。我们假设向量元素用 16 位的有符号字来表示。MULS.W 指令将两个 16 位的有符号数相乘并产生一个 32 位的积。然后每次乘法的结果会被累加到一个 32 位的和中。

```
        MOVEA.L   #AVEC, A1     第一个向量的地址
        MOVEA.L   #BVEC, A2     第二个向量的地址
        MOVE.L    N, D0         将计数器的初始值为元素的数量
        CLR.L     D1            D1 作为计数器
LOOP:   MOVE.W    (A1)+, D2     从向量 A 中获取元素
        MULS.W    (A2)+, D2     乘以向量 B 的元素
        ADD.L     D2, D1        累加乘积
        SUBQ.L    #1, D0        递减计数器
        BGT       LOOP          如果计数器大于 0，则重复此过程
        MOVE.L    D1, DOTPROD   结束时保存结果
```

[594] 图 C-15 计算两个向量点积的程序

### C.5.2 字符串搜索程序

图 2-31 中的程序在一个给定的目标字符串 $T$ 中确定第一个与模式字符串 $P$ 相匹配的实例。该程序的 ColdFire 版本如图 C-16 所示。

回想一下，CMP 指令被限制为只能使用长字操作数。因此，它不可能以图 2-31 所示的方式用一条指令来对一个寄存器值和一个存储器操作数执行字节大小的比较操作。而必须先通过使用单独的 MOVE.B 指令将两个要比较的字符放到寄存器 D0 和 D1 中，如图 C-16 所示，然后再使用 CMP.L 指令将它们进行比较。因为 MOVE.B 指令只影响目的寄存器的低位字节，所以在进入主循环之前需要将寄存器 D0 和 D1 中的所有 32 位清零，使得每次循环中的比较结果都是正确的。

```
        MOVEA.L   #T, A2      A2 指向字符串 T
        MOVEA.L   #P, A3      A3 指向字符串 P
        MOVEA.L   N, A4       获取值 n
        MOVEA.L   M, A5       获取值 m
        SUBA.L    A5, A4      计算 n - m
        ADDA.L    A2, A4      A4 是 T(n - m) 的地址
        ADDA.L    A3, A5      A5 是 P(m) 的地址
        CLR.L     D0          将用于比较的数据
        CLR.L     D1          寄存器清零
LOOP1:  MOVEA.L   A2, A0      用 A0 遍历字符串 T
        MOVEA.L   A3, A1      用 A1 遍历字符串 P
LOOP2:  MOVE.B    (A0)+, D0   比较字符串 T
        MOVE.B    (A1)+, D1   和 P 中的
        CMP.L     D0, D1        一对字符
        BNE       NOMATCH
        CMPA.L    A1, A5      检测是否在 P(m) 中
        BGT       LOOP2       如果没有完成再次循环
        MOVE.L    A2, RESULT  保存 T(i) 的地址
        BRA       DONE
NOMATCH: ADDA.L   #1, A2      指向 T 中的下一个字符
        CMPA.L    A2, A4      检查是否在 T(n - m) 中
        BGE       LOOP1       如果没有完成再次循环
        MOVE.L    #-1, RESULT 没有找到匹配
DONE:   下一条语句
```

图 C-16 字符串搜索程序

595

## C.6 操作模式和其他控制功能

到目前为止，我们已经描述了指令使用地址/数据寄存器和存储器中的操作数的正常执行过程。我们还介绍了一些实现简单计算任务的程序。输入/输出操作和其他任务可能需要额外的功能（如中断）来改变正常的执行行为。这样的行为可以通过使用图 C-2 所示状态寄存器中的某些位来控制。本节将介绍这些位。

- S 位可从两种可能的操作模式中选择一种。在*管态模式*（supervisor mode）下，S 位被置为 1。处理器可以执行所有的指令并访问所有可用的功能，包括某些特权（privileged）功能。例如，将状态寄存器（SR）作为源或目的操作数的 MOVE 指令就是一条特权指令，因为它能访问 S 位和其他控制处理器行为的位。处理器复位时进入管态模式。系统软件（包括中断服务程序）在这种模式下执行。另一方面，正常的应用程序运行在*用户模式*（user mode）下。在这种模式下，S 位等于 0，这将阻止应用程序使用任何特权功能。从用户模式到管态模式的切换只能在进入一个中断服务程序的时候发生。而从管态模式到用户模式的切换可以直接通过执行一条修改状态寄存器的特权指令或者从中断服务程序返回来完成。
- $b_{10:8}$ 这三个位表示处理器当前的中断优先级（interrupt priority level）。这些位形成了中断屏蔽码（interrupt mask），它的值在 0 和 7 之间。给各种中断源分配不同的优先级（1 和 7 之间）。中断屏蔽码的设置可防止处理器响应优先级等于或低于当前屏蔽码的任何中断源的中断请求。将屏蔽位清零可允许所有的中断，而将这些位置为 7 则可禁止所有的中断，除了处于第 7 级的不可屏蔽的（nonmaskable）中断请求。这种基于中断屏蔽码的方法与 3.2.1 节所讨论的方法有很大的不同，那一节中图 3-7 所示的状态寄存器用单个 IE 位允许所有的中断。
- 当进入一个中断服务程序时，M 位被处理器自动清零。这个功能由系统软件使用，而

与一般的应用程序无关。

- T 位用于调试。当它被置为 1 时，每条指令执行后就会触发一个特殊的中断，以允许系统软件跟踪一个应用程序的执行情况。

上述所有位只能够在管态模式下进行修改。MOVE 指令的特权版本允许读取或修改状态寄存器中的所有位。而在用户模式下，MOVE 指令的非特权版本只允许读取或修改状态寄存器中的条件码标志。

596

## C.7 输入 / 输出

第 3 章通过一个将键盘上读取的字符进行显示的例子介绍了基于轮询和中断方式的 I/O 操作。现在我们使用该例子以及图 3-3 所示的 I/O 接口寄存器，来介绍 ColdFire 指令如何用于 I/O 操作。

图 C-17 给出了图 3-5 中对字符的输入和输出都使用轮询法的程序的 ColdFire 版本。BTST.B 指令对应于图 3-5 中的 TestBit 指令。使用立即操作数来指定 BTST.B 指令将要检测的位限制了目的操作数所能使用的寻址方式。不可以使用绝对寻址方式，因此图 C-17 中的程序将地址寄存器 A3 和 A4 初始化为两个 I/O 接口的状态寄存器的地址，使得 BTST.B 指令可以使用间接寻址方式。此外，CMP 指令限制为只能使用长字操作数，因此在执行图 C-17 中的循环之前，寄存器 D0 被清为零。每个字符从键盘接口读到寄存器 D0 中，以便于之后与回车字符 CR 进行比较。随着每个字符从寄存器 D0 存储到存储器中，寄存器 A2 中的指针递增。

```
        MOVEA.L   #LOC, A2          初始化寄存器 A2，使其指向主存
                                    中存储字符的第一个单元的地址

        MOVEA.L   #KBD_STATUS, A3   初始化寄存器 A3，使其指向键盘
                                    状态寄存器的地址
        MOVEA.L   #DISP_STATUS, A4  初始化寄存器 A4，使其指向显示
                                    器状态寄存器的地址
        CLR.L     D0                将用来保存字符的数据寄存器清零

READ:   BTST.B    #1, (A3)          等待一个字符被输入到键盘缓冲区
        BEQ       READ              中
        MOVE.B    KBD_DATA, D0      将字符从 KBD_DATA 中读取到寄
                                    存器 D0 中（这将 KIN 清为 0）
        MOVE.B    D0, (A2)+         将字符传送到主存并递增指针以存
                                    储下一个字符

ECHO:   BTST.B    #2, (A4)          等待显示器准备就绪
        BEQ       ECHO              将刚读入的字符移到显示器缓冲寄
        MOVE.B    D0, DISP_DATA     存器中（这将 DOUT 清为 0）

        CMPI.L    #CR, D0           检查刚读入的字符是否为 CR（回
                                    车符），如果不是，则转移回去读取
        BNE       READ              另一个字符
```

图 C-17 图 3-5 中基于轮询方式程序的 ColdFire 版本

为了说明 I/O 操作中如何使用中断，图 C-18 使用 ColdFire 指令实现了图 3-10 中的例子。假设图 C-18 所示主程序中的初始化代码在处理器处于管态模式的时候执行。两条特权 MOVE 指令可访问状态寄存器（SR）。使用这两条指令以及 ANDI 指令来将中断屏蔽位清 0，从而允许所有的中断。由于我们希望其他位保持不变，所以状态寄存器的当前内容首先被读入到数据寄存器中，然后只有与中断优先级屏蔽码有关的位才会通过 ANDI 指令清为 0。最后，将修改

后的内容写回到状态寄存器中。

```
中断服务程序
ILOC:   MOVE.L    A2, -(A7)              保存寄存器
        MOVE.L    D0, -(A7)
        MOVEA.L   PNTR, A2               从存储器中载入地址指针
        CLR.L     D0                     将寄存器清零，以保存字符
        MOVE.B    KBD_DATA, D0           从键盘读入字符
        MOVE.B    D0, (A2)+              写入存储器并递增指针
        MOVE.L    A2, PNTR               修改存储器中的指针
        MOVEA.L   #DISP_STATUS, A2       将 A2 设置为状态寄存器的地址
ECHO:   BTST.B    #2, (A2)               等待显示操作准备就绪
        BEQ       ECHO
        MOVE.B    D0, DISP_DATA          显示刚读入的字符
        CMP.L     #CR, D0                检查刚读入的字符是否为回车符（CR）
        BNE       RTRN                   如果不是 CR 则返回
        ADDQ.L    #1, EOL                指示行的结束
        MOVEA.L   #KBD_CONT, A2          将 A2 设置为控制寄存器的地址
        BCLR.B    #1, (A2)               禁止键盘中断
RTRN:   MOVE.L    (A7)+, D0              恢复寄存器
        MOVEA.L   (A7)+, A2
        RTE                              从中断返回
主程序
START:  MOVEA.L   #LINE, A0              初始化缓冲区指针
        MOVE.L    A0, PNTR
        CLR.L     EOL                    清除行结束指示变量
        MOVEA.L   #KBD_CONT, A0          将 A0 设置为控制寄存器的地址
        BSET.B    #1, (A0)               允许键盘中断
        MOVE.W    SR, D0                 将状态寄存器的内容读入到 D0 中
        ANDI.L    #$F8FF, D0             清除优先级屏蔽码，以允许中断
        MOVE.W    D0, SR                 将 D0 的值写入到状态寄存器中
```

图 C-18　图 3-10 中基于中断方式程序的 ColdFire 版本

597
2
598

## C.8　浮点运算

浮点运算指令包含在基本 ColdFire 指令集的扩展集中 [1]。具有该扩展集的处理器的硬件实现包含一个单独的浮点部件（floating-point unit，FPU）和额外的寄存器。FPU 允许浮点运算与其他指令并发执行。本节对 ColdFire 的浮点功能进行简要概述。浮点数的表示在第 1 章中进行了介绍，浮点算术运算在第 9 章中进行了讨论。

所有的浮点运算都是对 64 位的双精度数进行的。为实现这个目的，在 FPU 中提供了 8 个 64 位的浮点数据寄存器 FP0 到 FP7。FPU 还有额外的控制和状态寄存器（详细信息请查阅 ColdFire 的技术文档 [1]）。浮点状态寄存器（FPSR）具有诸如 N 和 Z 这样的条件码标志以及浮点运算所特有的额外标志，这些额外的标志受到涉及浮点数的数据移动、算术和比较操作的影响。

浮点寄存器总是保存 64 位的双精度数。存储器可能包含 32 位单精度或 64 位双精度表示的数。当涉及大量的浮点数据但并不需要很高的精度时，单精度表示会降低存储需求。在执行算术运算之前，FPU 会自动将存储器中的任何单精度操作数转换为双精度数。如果需要，在将双精度数传送到存储器时，也可以将其转换为单精度数。

FPU 也可以在整数和双精度浮点表示之间进行转换，后面将会介绍。在硬件中实现该功能不需要软件来进行转换，所以可减少执行时间和代码大小。

基本指令集的浮点扩展集需要在汇编语言中进行额外的大小说明。浮点指令使用后缀 D

表示双精度操作数，后缀 S 表示单精度操作数。

### C.8.1 FMOVE 指令

FMOVE 指令在存储单元和浮点寄存器或者两个寄存器之间传输数据。后缀指定了操作数的大小，可根据需要将操作数转换为双精度表示或者从双精度表示转换为其他的表示。源操作数可能在一个浮点寄存器（FP0 ~ FP7）或者数据寄存器（D0 ~ D7）中，也可能在一个由间接、自增、自减、基本变址或基本相对寻址方式指定的存储单元中。除了基本相对方式不可使用之外，目的操作数所允许使用的寻址方式跟源操作数是相同的。源操作数或目的操作数必须有一个在浮点寄存器中。当目的操作数在一个浮点寄存器中时，浮点状态寄存器（FPSR）中
[599] 的标志，如 N 和 Z，会受到最终的 64 位双精度结果的影响。当目的操作数在一个数据寄存器（D0 ~ D7）或者一个存储单元中时，FPSR 是不会受到影响的。

当操作数的大小说明符为 D 时，64 位双精度浮点数将不加修改地从浮点寄存器传送到存储器或从存储器传送到浮点寄存器，或者在两个浮点寄存器之间进行传送。例如，指令

FMOVE.D (A3), FP5

将双精度数从地址寄存器 A3 所指定的存储单元装入浮点寄存器 FP5 中。同样，指令

FMOVE.D FP2, 16(A5)

将双精度数从寄存器 FP2 存储到由寄存器 A5 的值加上 16 所给出的存储单元中。指令

FMOVE.D FP3, FP4

将双精度数从 FP3 传送到 FP4 中。

当为 FMOVE 指令指定任何其他的操作数大小后缀时，都会将要传输的信息自动转换为双精度表示或者从双精度表示转换为其他的表示。例如，指令

FMOVE.S FP1, (A4)

将寄存器 FP1 中的 64 位双精度浮点数转换成 32 位的单精度浮点数，然后将转换后的数写入由寄存器 A4 的值给出的存储单元中。指令

FMOVE.L 16(A2), FP3

使用变址寻址方式从存储器中读取一个长字，并将这个 32 位的整数转换为 64 位的双精度浮点数，再将转换后的数放到寄存器 FP3 中。指令

FMOVE.W FP7, D2

将寄存器 FP7 中的 64 位双精度浮点数转换为 16 位的整数，然后再将转换后的数放到寄存器 D2 的低位中。显然，将浮点表示转换为整数表示时，会有潜在的精度损失。

### C.8.2 浮点算术指令

对浮点数进行算术运算的基本指令有 FADD、FSUB、FMUL 和 FDIV。在所有的这些指令中，目的操作数都必须在浮点数寄存器中。FPSR 中的条件码标志，如 N 和 Z，会受到最终的 64 位双精度结果的影响。源操作数可能在一个浮点寄存器或数据寄存器中，也可能在一个由间接、自增、自减、基本变址或基本相对寻址方式所指定的存储单元中。需要一个后缀来说明
[600] 源操作数的格式。对于除了 D 以外的其他任何后缀，在执行指定的算术运算之前，源操作数将被自动转换为双精度表示。例如，指令

FADD.W (A2)+, FP6

从地址寄存器 A2 所给出的存储单元中读取一个 16 位的整数。A2 自动递增 2，并将这个 16 位的整数转换为 64 位的双精度浮点数，然后再将转换后的数与寄存器 FP6 中的数相加。

### C.8.3 比较和转移指令

使用 FCMP 指令可对两个浮点数进行比较。比较的结果会影响 FPSR 中的标志，如 N 和 Z，之后这些标志会被转移指令使用。FCMP 指令的目的操作数必须在浮点寄存器中。源操作数可能在一个浮点寄存器或数据寄存器中，也可能在一个由间接、自增、自减、基本变址或基本相对寻址方式所指定的存储单元中。必须用一个后缀来指定源操作数的大小并根据需要将源操作数转换为双精度表示。例如，指令

```
FCMP.S   (A1), FP3
```

读取由寄存器 A1 的值所给出的存储单元中的 32 位单精度浮点数，并将这个数转换为 64 位双精度表示，然后再将寄存器 FP3 中的值减去转换后的数，并根据减法运算的结果设置 FPSR 中的标志。寄存器 FP3 中的值没有被修改。

浮点条件转移指令的格式为

```
FBcc   LABEL
```

其中 *cc* 指定浮点条件码，可为 *cc* 指定诸如 EQ、NE、LT 和 GT 之类的测试，与 FPSR 中条件码标志的不同组合相对应。对于条件为真时的转移目标，FBcc 指令会在执行时指定一个相对于 PC 值的偏移量。根据 FBcc 指令到转移目标的距离，偏移量可由 16 位或 32 位表示。

由于 FADD 和 FSUB 之类的浮点算术指令会影响 FPSR 中的标志，所以在某些情况下可能不需要在 FBcc 指令之前加一条 FCMP 指令。C.8.5 节中的示例程序显示了一条浮点算术指令之后如何紧跟一条浮点转移指令。

### C.8.4 其他浮点指令

FPU 还支持一些其他的指令，如平方根（FSQRT）、求反（FNEG）和绝对值（FABS）。这些指令可以指定一个或两个操作数。对于单操作数来说，它必须在浮点寄存器中。对于双操作数来说，目的位置必须是浮点寄存器，且源操作数的有效寻址方式与基本浮点算术指令中的 601
源操作数寻址方式相同。FPSR 中的条件码标志会受到这些指令中每一条指令结果的影响。根据指令助记符后面的格式说明符，可按需要对源操作数进行数字转换。

### C.8.5 浮点程序示例

图 C-19 给出了一个示例程序。假设有两个点 $(x_0, y_0)$ 和 $(x_1, y_1)$，该程序确定经过这两个点的直线的斜率 $m$ 及其在 $y$ 轴上的截距 $b$，当这两个点位于一条垂直线上时除外。

假定两点的坐标 $x_0$、$y_0$、$x_1$ 和 $y_1$ 为 64 位的双精度浮点数，存储在从 COORDS 单元开始的连续存储单元中。该程序把计算得到的斜率和截距作为双精度数放在存储单元 SLOPE 和 INTERCEPT 中。直线的斜率公式为 $m = (y_1 - y_0) / (x_1 - x_0)$，因此，该程序在执行除法运算之前必须先检查分母是否为零。当分母为零时，两点在一条垂直线上。该程序将值 1 写入存储器中一个称为 VERT_LINE 的单独变量中以反映这种情况，并指出存储单元 SLOPE 和 INTERCEPT 中的值是无效的，不做进一步的计算。否则，该程序计算出斜率值，并将其保存到存储器中。对于有效的斜率，程序计算出截距 $b = y_1 - m \cdot x_0$，也将其保存到存储器中。最后，程序将 0 写

到存储单元 VERT_LINE 中来表明斜率和截距是有效的。

```
                MOVEA.L   #COORDS, A2      A2 指向坐标列表
                FMOVE.D   (A2), FP0        FP0 中的值是 x1
                FMOVE.D   8(A2), FP1       FP1 中的值是 y1
                FMOVE.D   16(A2), FP2      FP2 中的值是 x2
                FMOVE.D   24(A2), FP3      FP3 中的值是 y2
                FSUB.D    FP0, FP2         计算 x1 - x2，可能会设置
                                           FPSR 中的 Z 标志
                FBEQ      NO_SLOPE         如果分母为 0，则 m 是不确定的
                FSUB.D    FP1, FP3         计算 y1 - y2
                FDIV.D    FP2, FP3         计算 m = (y1 - y2) / (x1 - x2)
                MOVEA.L   #SLOPE, A2       A2 指向存储单元 SLOPE
                FMOVE.D   FP3, (A2)        将斜率保存到存储器中
                FMUL.D    FP3, FP0         计算 m · x1
                FSUB.D    FP1, FP0         计算 b = y1 - m · x1
                MOVEA.L   #INTERCEPT, A2   A2 指向存储单元 INTERCEPT

                FMOVE.D   FP1, (A2)        将截距保存到存储器中
                MOVEQ.L   #0, D0           表明直线不是垂直的
                BRA       DONE
NO_SLOPE:       MOVEQ.L   #1, D0           表明直线是垂直的
DONE:           MOVE.L    D0, VERT_LINE
```

602

图 C-19 计算直线斜率与截距的浮点程序

## C.9 结束语

ColdFire 实现了一个 CISC 风格的指令集，该指令集在其许多指令中合并了算术 / 逻辑运算和存储器访问操作。这种方法减少了完成一个给定的计算任务所必须执行的指令数目。根据所要执行的操作的复杂性和产生操作数有效地址所需的信息量，指令被编码为一个、两个或三个 16 位的存储字。支持多种寻址方式。除了整数指令，ColdFire 已定义了一个浮点扩展集。整数和浮点数表示之间能自动进行转换，这是浮点扩展集的一个功能。

## C.10 问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。

**例 C.2**

**问题**：假设有一个 ASCII 编码的字符串保存在存储器中，起始地址为 STRING，该字符串以回车符（CR）结束。写一个 ColdFire 程序来确定该字符串的长度。

**解答**：图 C-20 给出了一个可能的程序。字符串中的每个字符与 CR（ASCII 码为 0D）进行比较，计数器递增，直至到达字符串的末尾。结果存储在单元 LENGTH 中。

```
                MOVEA.L   #STRING, A2      A2 指向字符串的开始
                CLR.L     D3               D3 是计数器，被清为 0
                CLR.L     D5               D5 用于长字比较，被清为 0
LOOP:           MOVE.B    (A2)+, D5        获取下一个字符并递增指针
                CMPI.L    #$0D, D5         将字符与 CR 进行比较
                BEQ       DONE             如果匹配，则结束
                ADDQ.L    #1, D3           递增计数器
                BRA       LOOP             如果没有结束，则循环回到前面
DONE:           MOVE.L    D3, LENGTH       将计数值存放到存储单元 LENGTH 中
```

603

图 C-20 例 C.2 的程序

**例 C.3**

**问题**：我们希望在一个 32 位的非负整数列表中找出最小的数。假设问题有关的所有数据的存储地址从 $(1000)_{16}$ 开始。该地址处的长字必须保存所找到的最小数。地址 $(1004)_{16}$ 处的下一个长字包含列表中的项数 $n$。之后从地址 $(1008)_{16}$ 开始的 $n$ 个长字包含列表中的数。写一个程序，找出最小的数，并包含按照规定组织数据所需的汇编指示（assembler directive）。

**解答**：图 C-21 中的程序实现了所需的任务。程序中的注释解释了这个任务如何执行。该程序假设 $n \geq 1$。列表中包含了一些样本数字。

```
LIST:     .EQU      $1000              列表起始地址
          MOVEA.L   #LIST, A2          A2 指向列表的开始
          MOVE.L    4(A2), D3          D3 是计数器，初始化为 n
          MOVEA.L   A2, A4             调整 A4 的值后，A4 指向第一个数
          ADDA.L    #8, A4
          MOVE.L    (A4), D5           D5 保存目前为止所找到的最小的数
LOOP:     SUBQ.L    #1, D3             递减计数器
          BEQ       DONE               如果 D3 为 0，则结束
          MOVE.L    (A4)+, D6          获取下一个数并递增指针

          CMP.L     D6, D5             将下一个数与目前的最小数进行比较
          BLE       LOOP               如果下一个数不小于目前的最小数，则再次循环
          MOVE.L    D6, D5             否则，更新目前的最小数
          BRA       LOOP               再次循环
DONE:     MOVE.L    D5, (A2)           将最小的数保存到 SMALL 中

          .ORG      $1000
SMALL:    .DS.L     1                  所找到的最小数的存储空间
N:        .DC.L     7                  列表中的项数
ENTRIES:  .DC.L     4, 5, 3, 6, 1, 8, 2   列表中的项
```

图 C-21　例 C.3 的程序

**例 C.4**

**问题**：写一个 ColdFire 程序，将一个 $n$ 位十进制整数转换成二进制数。如果数字是通过键盘输入的，那么十进制数是以 $n$ 个 ASCII 编码字符的形式给出的。 604

**解答**：考虑一个 4 位十进制数 $D$，由数字 $d_3d_2d_1d_0$ 表示。该数的值为 $((d_3 \times 10 + d_2) \times 10 + d_1) \times 10 + d_0$。该数的这种表示是图 C-22 中程序所使用转换技术的基础。注意，每个 ASCII 编码字符先通过 ANDI 指令转换成一个二进制编码的十进制（BCD）数字，然后再用于计算。

```
          MOVE.L    N, D2              D2 是计数器，初始化为 n
          MOVEA.L   #DECIMAL, A3       A3 指向 ASCII 数字
          CLR.L     D4                 D4 用来保存二进制数
          MOVEQ.L   #10, D6            D6 用来乘以 10
LOOP:     MOVE.B    (A3)+, D5          获取下一个 ASCII 数字并递增指针

          ANDI.L    #$0F, D5           产生 BCD 数字
          ADD.L     D5, D4             加到中间结果中
          SUBQ.L    #1, D2             递减计数器
          BEQ       DONE               如果结束，则退出循环
          MULU.L    D6, D4             乘以 10
          BRA       LOOP               如果未完成，则循环回到前面
DONE:     MOVE.L    D5, BINARY         将结果存放到存储单元 BINARY 中
```

图 C-22　例 C.4 的程序

**例 C.5**

**问题**：考虑一个二维数字阵列 A(*i*, *j*)，其中 *i* = 0 到 *n*–1，是行索引，*j* = 0 到 *m*–1，是列索引。该阵列按行存储在计算机的存储器中，每行元素占用 *m* 个连续的字单元。写一个子程序，将第 *x* 列的元素逐一加到第 *y* 列的元素上，和元素（sum element）放在第 *y* 列上。索引值 *x* 和 *y* 通过寄存器 D2 和 D3 传递给子程序。参数 *n* 和 *m* 通过寄存器 D4 和 D5 传递给子程序，元素 A(0,0) 的地址通过寄存器 A0 进行传递。

**解答**：图 C-23 给出了一个可能的程序。我们假定值 *x*、*y*、*n* 和 *m* 分别存放在存储单元 X、Y、N 和 M 中。此外，阵列中的元素存储在从单元 ARRAY 开始的连续的字中，ARRAY 是元素 A(0,0) 的地址。程序中的注释说明了每条指令的作用。

```
        MOVE.L    X, D2          载入值 x
        MOVE.L    Y, D3          载入值 y
        MOVE.L    N, D4          载入值 n
        MOVE.L    M, D5          载入值 m
        MOVEA.L   #ARRAY, A0     载入 A(0,0) 的地址
        ISR       SUB
        下一条指令
        :

SUB:    MOVE.L    A1, –(A7)      保存寄存器 A1 的内容
        LSL.L     #2, D5         确定一列中连续元素之间的距离（以字
                                 节为单位）
        SUB.L     D2, D3         产生 y – x 的值
        LSL.L     #2, D3         产生 4(y – x) 的值
        LSL.L     #2, D2         产生 4x 的值
        ADDA.L    D2, A0         A0 指向 A(0,x)
        MOVEA.L   A0, A1
        ADDA.L    D3, A1         A1 指向 A(0,y)
LOOP:   MOVE.L    (A0), D2       获取第 x 列中的下一个数
        MOVE.L    (A1), D3       获取第 y 列中的下一个数
        ADD.L     D3, D2         将两数相加，并保存和
        MOVE.L    D2, (A1)
        ADDA.L    D5, A0         递增第 x 列的指针
        ADDA.L    D5, A1         递增第 y 列的指针
        SUBQ.L    #1, D4         递减行计数器
        BGT       LOOP           如果未完成，则循环回去
        MOVE.L    (A7)+, A1      恢复寄存器 A1 的内容
        RTS                      返回到调用程序
```

图 C-23 例 C.5 的程序

**例 C.6**

**问题**：假设存储单元 BINARY 中包含一个 32 位的模式。我们希望在一个具有图 3-3 所示接口的显示设备上将这些位显示为 8 个十六进制数字。写一个程序完成这个任务。

**解答**：首先，我们需要将这个 32 位的模式转化为用 ASCII 编码字符表示的十六进制数字。转换过程可以通过查表法来完成。必须构造一个包含 16 项的表，以便于为每一个十六进制数字提供其 ASCII 码。然后，对于 BINARY 中该模式的每一个 4 位的段，都可以在表中查找到其相应的字符，并将这些字符存储在从地址 HEX 开始的连续字节单元中。最后，将从地址 HEX 开始的这 8 个字符发送给显示器。图 C-24 给出了一个可能的程序。因为 ColdFire 不包含循环移位指令，所以我们使用 4 对 LSL 和 ADDX 指令来达到将一个寄存器中的值循环移动 4 位的效果。

```
        MOVE.L   BINARY, D2      载入该二进制数
        MOVE.L   #8,D3           D3 是数字计数器，被设置为 8
        MOVEA.L  #HEX, A4        A4 指向十六进制数字
        MOVEA.L  #TABLE, A6      A6 指向 ASCII 码转换表
        CLR.L    D0              D0 为 0，用于下面的循环移位
LOOP:   LSL.L    #1, D2          通过使用 X 标志和加 0 操作（4 次）将高
        ADDX.L   D0, D2          位数字循环移位到低位位置上
        LSL.L    #1, D2
        ADDX.L   D0, D2
        LSL.L    #1, D2
        ADDX.L   D0, D2
        LSL.L    #1, D2
        ADDX.L   D0, D2
        MOVE.L   D2, D5          将当前值复制到另一个寄存器中
        ANDI.L   #$0F, D5        提取下一个数字
        MOVE.B   (A6,D5), D6     获取该数字的 ASCII 码
        MOVE.B   D6, (A4)+       将其存储在 HEX 缓冲区中，并递
                                 增数字指针
        SUBQ.L   #1, D3          递减数字计数器
        BGT      LOOP            如果不是最后一个数字，则循环回到前面
DISPLAY: MOVEQ.L #8, D3
        MOVEA.L  #HEX, A4
        MOVEA.L  #DISP_DATA, A2
DLOOP:  MOVE.L   4(A2), D5       通过测试 DOUT 标志来检查显示器
        ANDI.L   #4, D5          是否准备就绪
        BEQ      DLOOP
        MOVE.B   (A4)+, (A2)     获取下一个 ASCII 字符，递
                                 增字符指针，并将其发送给
                                 显示器
        SUBQ.L   #1, D3          递减计数器
        BGT      DLOOP           循环，直到所有的字符都显示完毕
        下一条指令

        .ORG     1000
HEX:    .DS.B    8               ASCII 码数字的存储空间
TABLE:  .DC.B    $30, $31, $32, $33   ASCII 码转换表
        .DC.B    $34, $35, $36, $37
        .DC.B    $38, $39, $41, $42
        .DC.B    $43, $44, $45, $46
```

图 C-24　例 C.6 的程序　607

## 习题

[E] C.1 写一个程序，计算表达式 SUM = 580 + 68 400 + 80 000。

[E] C.2 写一个程序，计算表达式 ANSWER = A × B + C × D。

[M] C.3 写一个程序，在一个含有 $n$ 个 32 位整数的列表中找出所包含的负整数的个数，并将该计数保存在单元 NEGNUM 中，$n$ 保存在存储单元 N 中，列表中的第一个整数保存在单元 NUMBERS 中。在程序中包含必要的汇编指示（assembler directive）和一个样本列表，列表中含有 6 个数字，其中一些是负数。

[E] C.4 为图 C-8 中的程序写一个如图 C-14 所示风格的汇编语言程序。假设采用图 2-10 所示的数据布局。

[M] C.5 写一个 ColdFire 程序来解决第 2 章的习题 2.10 中的问题。

[M] C.6 写一个 ColdFire 程序来解决第 2 章中例 2.5 所描述的问题。

[M] C.7 写一个 ColdFire 程序来解决第 3 章中例 3.5 所描述的问题。

[M] C.8 写一个 ColdFire 程序来解决第 3 章中例 3.6 所描述的问题。

[M] C.9 假设 TABLE 的地址是 0x10100，写一个 ColdFire 程序来解决第 3 章中例 3.6 所描述的问题。

[M] C.10 写一个程序，在视频显示器的一行上以十六进制形式显示主存中 10 个字节的内容。该字节串在存储器中的起始位置是 LOC。每个字节将显示为两个十六进制字符。连续字节应以空格分隔。

[M] C.11 假设存储单元 BINARY 中包含一个 16 位的模式。我们希望在一个具有图 3-3 所示接口的显示设备上将这些位显示为 0 和 1 组成的字符串。写一个 ColdFire 程序完成这个任务。

[M] C.12 使用图 3-17 中的七段显示器和图 3-14 中的定时器电路，写一个程序，显示序列 0、1、2、…、9、0、…中的十进制数字，每个数字显示一秒钟。假设定时器电路中的计数器是由 100 MHz 的时钟驱动的。

[M] C.13 使用两个图 3-17 所示的七段显示器和图 3-14 所示的定时器电路，写一个程序，显示序列 0、1、2、…、98、99、0、…中的数，每个数显示一秒钟。假设定时器电路中的计数器是由 100 MHz 的时钟驱动的。

[M] C.14 写一个程序，计算实时时钟时间并以小时（0 ~ 23）和分钟（0 ~ 59）的形式显示时间。显示器包括 4 个图 3-17 所示的七段显示设备。还有一个具有图 3-14 所示接口的定时器电路，其计数器是由 100 MHz 的时钟驱动的。

[M] C.15 写一个 ColdFire 程序来解决第 2 章的习题 2.23 中的问题。

608 [D] C.16 写一个 ColdFire 程序来解决第 2 章的习题 2.24 中的问题。

[M] C.17 写一个 ColdFire 程序来解决第 2 章的习题 2.25 中的问题。

[M] C.18 写一个 ColdFire 程序来解决第 2 章的习题 2.26 中的问题。

[D] C.19 写一个 ColdFire 程序来解决第 2 章的习题 2.27 中的问题。

[M] C.20 写一个 ColdFire 程序来解决第 2 章的习题 2.28 中的问题。

[M] C.21 写一个 ColdFire 程序来解决第 2 章的习题 2.29 中的问题。

[M] C.22 写一个 ColdFire 程序来解决第 2 章的习题 2.30 中的问题。

[M] C.23 写一个 ColdFire 程序来解决第 2 章的习题 2.31 中的问题。

[D] C.24 写一个 ColdFire 程序来解决第 2 章的习题 2.32 中的问题。

[D] C.25 写一个 ColdFire 程序来解决第 2 章的习题 2.33 中的问题。

[D] C.26 写一个 ColdFire 程序来解决第 3 章的习题 3.20 中的问题。

[D] C.27 写一个 ColdFire 程序来解决第 3 章的习题 3.22 中的问题。

[D] C.28 写一个 ColdFire 程序来解决第 3 章的习题 3.24 中的问题。

[D] C.29 写一个 ColdFire 程序来解决第 3 章的习题 3.26 中的问题。

[D] C.30 当 $0 \le x \le \pi/2$ 时，函数 sin(x) 能以合理的精度近似为 $x - x^3/6 + x^5/120 = x(1 - x^2(1/6 - x^2(1/120)))$。写一个子程序 SIN，接受一个输入参数，表示浮点寄存器中的 $x$，并使用上述只涉及 $x$ 和 $x^2$ 的第二个表达式来计算 sin(x) 的近似值。计算所得的值应返回到浮点寄存器中。子程序使用的任何寄存器，包括浮点寄存器，都应根据需要进行保存和恢复。

## 参考文献

609 ~ 610

1.Freescale Semiconductor, Inc., *ColdFire Family Programmer's Reference Manual*, Document Number CFPRM Rev. 3, March 2005. Available at http://www.freescale.com.

# ARM 处理器

## 附录目标

在本附录中，你将学习 ARM 处理器。本附录主要讨论：

- 指令集体系结构
- 输入 / 输出能力
- 对嵌入式应用的支持



第 2 章中，我们介绍了指令集和寻址方式设计中使用的基本概念。第 3 章讨论了 I/O 操作。在本附录中，我们将介绍这些概念如何在 ARM 处理器中实现，并用 ARM 汇编语言展示第 2 章和第 3 章中给出的一般程序。

ARM（Advanced RISC Machine）有限公司设计出了一个 RISC 风格的处理器系列。ARM 公司将这些设计和一些软件工具授权给其他公司，用于芯片制造以及系统的开发和仿真。ARM 处理器主要用于低功耗和低成本的嵌入式应用，如移动电话、通信调制解调器和汽车引擎管理系统。

所有的 ARM 处理器共享一个基本的机器指令集。这里使用的 ISA 版本被 ARM 公司称为第 4 版 [1]。后来的版本中增加了一些不需要在本附录中讨论的扩展信息。不过，在后面的章节中我们将简要总结其中的一些扩展信息。Furber 的专著 [2] 是 ARM 处理器及其设计原理的信息来源。Hohl 的专著 [3] 介绍了 ARM 汇编语言。

## D.1 ARM 的特点

ARM 的字长是 32 位，存储器是按字节编址的，每个字节的地址是 32 位，处理器中寄存器的长度也是 32 位。在存储器和处理器寄存器之间传送数据时采用三种操作数长度：字节（8 位）、半字（16 位）和字（32 位）。字和半字的地址必须是对齐的，也就是说，它们必须分别是 4 和 2 的倍数。支持小端和大端两种编址方案，由一条与处理器相连的外部输入控制线做出选择。

在大多数情况下，ARM ISA 反映了 RISC 风格的体系结构，但它也有一些 CISC 风格的特征。

RISC 风格方面

- 所有指令的长度是固定的，为 32 位。
- 只有 Load 和 Store 指令可以访问存储器。
- 所有的算术和逻辑指令都只能对处理器寄存器中的操作数进行操作。

CISC 风格方面

- 提供自增、自减和基于 PC 的相对寻址方式。
- 条件码（N、Z、V 和 C）用于转移和指令的条件执行，它们的含义已在 2.10.2 节中作过解释。
- 使用单一一条指令就可以将一个连续的存储器字块装入多个寄存器中或者将多个寄存器的内容存储在一个块中。

### ARM 体系结构的特色

ARM 体系结构具有一些现代处理器所没有的特征。

**1. 指令的条件执行**

ARM 处理器的一个特色是所有的指令都是有条件地执行。只有条件码标志的当前值满足
612 了指令的一个 4 位字段所指定的条件时指令才会被执行。否则，处理器将处理下一条指令。其中有一个条件用来指定该指令总是被执行的。条件执行的优点将在 D.9 节中通过例子来说明。现在，我们暂时忽略这个特点并假定该指令的条件字段指定为“总是被执行”。

**2. 无移位或除法指令**

指令集中没有明确地提供 Shift（移位）指令。但是算术、逻辑或 Move（移动）指令中的一个立即值或一个寄存器操作数可以在其参与操作之前移位一个预定的量，这将在 D.4.2 节中进行解释。此功能可以用来隐含地实现移位指令。

指令集中有许多不同的乘法指令，并可以进行多种变化以用于信号处理应用中。但是没有硬件除法指令，除法运算必须用软件实现。

## D.2 寄存器结构

ARM 处理器有 16 个 32 位的处理器寄存器可用于用户应用程序，标记为 R0 到 R15，如图 D-1 所示。其中包含 15 个通用寄存器（R0 到 R14）和一个程序计数器（PC），也就是寄存器 R15。通用寄存器中可以保存存储器地址或者数据操作数。寄存器 R13 和 R14 专门用于处理器堆栈和子程序的管理中。这将在 D.4.8 节中进行讨论。

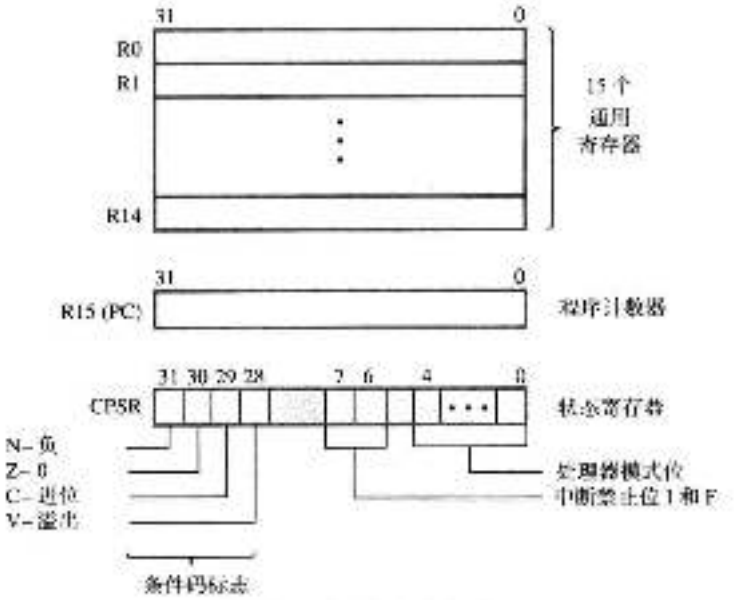



图 D-1 ARM 的寄存器结构

当前程序状态寄存器（CPSR），或简称为状态寄存器，如图 D-1 所示，保存条件码标志（N、Z、C、V）、中断禁止位和处理器模式位。处理器有 7 种操作模式。应用程序运行在用户模式下。其他 6 种模式用于处理 I/O 设备中断、处理器上电/复位、软件中断和存储器访问冲突。处理器模式和中断禁止位的用法将在 D.7 节和 D.8 节中进行描述。首先，我们假设处理器

在用户模式下执行一个应用程序。

此外，还有一些附加的通用寄存器，称为后备寄存器（banked register）。它们保存的是 R0 到 R14 中一些寄存器内容的副本。当处理器从用户模式切换到其他的操作模式时，就要使用不同的后备寄存器。使用后备寄存器可避免模式转换时对一些用户模式寄存器的内容进行保存和恢复的需要。各非用户模式下状态寄存器的已保存副本也是可用的。后备寄存器以及状态寄存器的副本内容都将在 D.7 节中进行讨论。

## D.3 寻址方式

2.4 节中讨论的立即数、寄存器、绝对、间接以及变址寻址方式都能以某种形式用于 ARM 体系结构中。除了这些基本方式（通常用于 RISC 处理器中），2.10.1 节中所介绍的相对方式以及自增和自减方式的变体形式也有提供。在 ARM 体系结构中，很多寻址方式都来自于不同形式的变址寻址方式。

### D.3.1 基本变址寻址方式

寻址存储器操作数的基本方法是变址寻址方式，定义为：

预变址方式（Pre-indexed mode）—— 操作数的有效地址是基址寄存器 Rn 的内容和一个有符号的偏移量之和。

我们将使用 Load 和 Store 指令（分别用汇编语言助记符 LDR 和 STR 表示，可将一个字从存储器加载到寄存器中或者将一个字从寄存器存储到存储器中）来说明变址寻址方式如何运作。指令的格式如图 D-2 所示。在所有的 ARM 指令中，高 4 位指定了一个条件，这个条件决定该指令是否执行，如 D.1.1 节所描述的那样。偏移量的大小以一个立即值的形式给出，包含在指令的低 12 位中，或者以另一个寄存器 Rm（在指令的低 4 位中指定）的内容给出。可以用操作码字段中的一个位来区分这两种情况。操作码字段中的另一个位可指定偏移量的符号（或方向）。在汇编语言中，符号是通过偏移量指出的。

613 ↓ 614

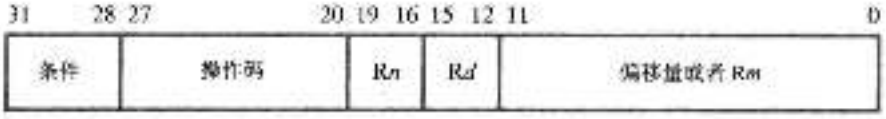



图 D-2 Load 和 Store 指令的格式

Load 指令

$$\text{LDR} \quad Rd, [Rn, \#\text{offset}]$$

用立即数方式指定偏移量（表示为一个有符号数），并执行操作

$$Rd \leftarrow [[Rn] - \text{offset}]$$

指令

$$\text{LDR} \quad Rd, [Rn, Rm]$$

执行操作

$$Rd \leftarrow [[Rn] + [Rm]]$$

由于 Rm 的内容是偏移量的大小，所以，如果需要负的偏移量，就要在 Rm 的前面加上一个减号。请注意，在 ARM 汇编语言中使用方括号而不是括号来表示间接寻址。预变址寻址方式的这两个版本分别与 2.4.3 节中定义的变址寻址和带基址的变址寻址方式相同。

如果偏移量为零，则不必在汇编语言中明确指定。因此，指令

LDR Rd, [Rn]

执行操作

Rd ← [[Rn]]

这被定义为 2.4.2 节中的间接寻址方式。

### D.3.2 相对寻址方式

在预变址寻址方式中，可将程序计数器 PC 用作基址寄存器 Rn。这实际上就是 2.10.1 节
615 所描述的相对寻址方式。程序员只需简单地将所需的地址标签放到操作数字段中来表明这种方式。因此，指令

LDR R1, ITEM

将存储单元 ITEM 中的内容装入寄存器 R1 中。汇编程序将立即数偏移量确定为操作数地址和更新后的 PC 内容之间的差。在指令执行期间计算有效地址的时候，PC 的内容已经被更新为从包含该相对寻址方式的指令向前两个字（8 个字节）的地址上。这是因为在指令的流水线执行中 ARM 处理器已经提取了下一条指令，我们在第 6 章中描述了指令的流水线执行。

### D.3.3 带写回的变址方式

在预变址寻址方式中，寄存器 Rn 的原始内容在产生操作数有效地址的过程中没有被改变。这种方式有一个变体，称为带写回（writeback）的预变址方式，在这种方式中 Rn 的内容会被改变。还有一种称为后变址的方式，也会改变寄存器 Rn 的内容。这些方式分别是 2.10.1 节中所介绍的自减和自增寻址方式的一般化形式。它们被定义为：

带写回的预变址方式（Pre-indexed with writeback mode）— 操作数有效地址的产生方法与预变址方式相同，然后有效地址要被写回到寄存器 Rn 中。

后变址方式（post-indexed mode）— 操作数的有效地址是寄存器 Rn 的内容，然后将偏移量加到这个地址上，并将结果写回到寄存器 Rn 中。

表 D-1 说明了所有寻址方式的汇编语言语法，并给出了计算有效地址 EA 的表达式。表中还显示了如何指定写回操作。在预变址方式中，惊叹号字符 "!" 表示要进行写回操作。后变址方式始终包含写回操作，所以不需要惊叹号字符。

从表 D-1 中可以看出，预变址和后变址方式的区别在于使用方括号的方式。当方括号中只有基址寄存器时，该寄存器的内容当作有效地址使用。在访问完操作数之后，将偏移量与寄存器的内容相加。换句话说，这是后变址方式。当基址寄存器和偏移量都放在方括号内时，两者之和就作为操作数的有效地址，也就是说这使用的是预变址方式。如果要执行写回操作，就必须通过惊叹号字符来指示。

表 D-1 ARM 变址寻址方式

| 名 称 | 汇编语法 | 寻址功能 |
|---|---|---|
| **带有立即数偏移量** | | |
| 预变址 | [Rn, #offset] | EA = [Rn] + offset |
| | | EA = [Rn] + offset; |
| 带写回的预变址 | [Rn, #offset]! | Rn ← [Rn] + offset |
| 后变址 | [Rn], #offset | EA = [Rn]; |
| | | Rn ← [Rn] + offset |

（续）

| 名 称 | 汇编语法 | 寻址功能 |
| --- | --- | --- |
| $R_n$ 中带有偏移量值 | | |
| 预变址 | [Rn, ± Rm, shift] | EA = [Rn] ± [Rm] shifted |
| 带写回的预变址 | [Rn, ± Rm, shift]! | EA = [Rn] ± [Rm] shifted;<br>Rn ← [Rn] + [Rm] shifted |
| 后变址 | [Rn], ± Rm, shift | EA = [Rn];<br>Rn ← [Rn] ± [Rm] shifted |
| 相对寻址<br>带有立即数偏移量的预变址 | Location | EA = Location<br>= [PC] + offset |

EA= 有效地址

offset= 包含在指令中的一个有符号数

shift= 方向 # 移数

这里方向为 LSL 时表示左移，为 LSR 时表示右移；移数是一个 5 位的无符号数，用来说明移位的量

±Rm= 寄存器 Rm 中的偏移量值，可以与基址寄存器 Rn 的内容相加或者从基址寄存器 Rn 的内容中减去

### D.3.4 偏移量的确定

在所有这三种变址寻址方式中，可以将偏移量指定为 -4095 范围内的一个立即数。还可以通过寄存器 Rm 的内容来指定偏移量的值，偏移量的符号（方向）由该寄存器名称前面的前缀“±”指定。例如，指令

LDR R0, [R1, –R2]!

执行操作

R0 ← [[R1] – [R2]]

由于指定了写回，所以操作数的有效地址 [R1] – [R2] 之后会被装入 R1 中。

当偏移量在寄存器中给出时，它在使用前可能会通过右移或左移缩放 2 的幂次倍。在汇编语言中，这可以通过将移位方向（LSL 表示左移，LSR 表示右移）和移位量放在寄存器的名称 Rm 之后来表示，如表 D-1 所示。移动的位数可用 0 到 31 范围内的一个立即数来指定。移位方向和移动的位数被编码在指令中指定 Rm 的同一字段中，如图 D-2 所示。例如，上例中 R2 的内容在作为偏移量使用之前可能要乘以 16，可将指令修改如下：

616 ~ 617

LDR R0, [R1, –R2, LSL #4]!

该指令执行操作

R0 ← [[R1] –16 × [R2]]

由于指定了写回操作，所以有效地址之后会被装入 R1 中。

### D.3.5 寄存器、立即数和绝对寻址方式

寄存器寻址方式是在算术和逻辑指令中访问操作数的主要方式。这将在 D.4 节中进行讨论。这些指令中也可以使用常量，它们以 8 位立即数的形式提供。

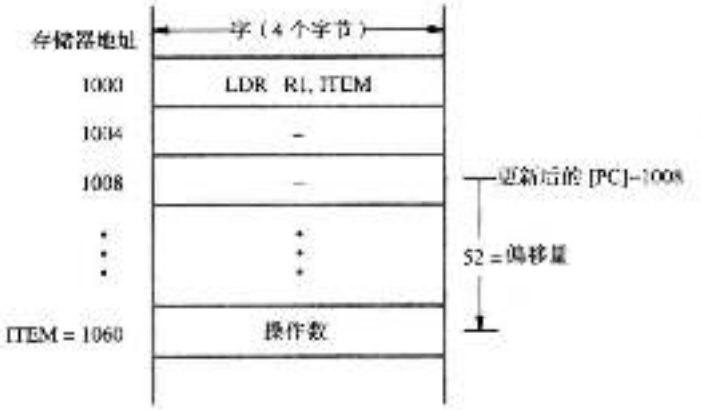
如果预变址方式中的基址寄存器包含值 0，则可得到访问存储器操作数的绝对寻址方式的限制形式。在这种情况下，12 位的偏移值是有效地址。

这里所描述的立即数寻址和绝对寻址方式分别只包含 8 位和 12 位的值。32 位值的生成及其作为立即操作数或存储器地址的用法将在 D.5.1 节中描述。

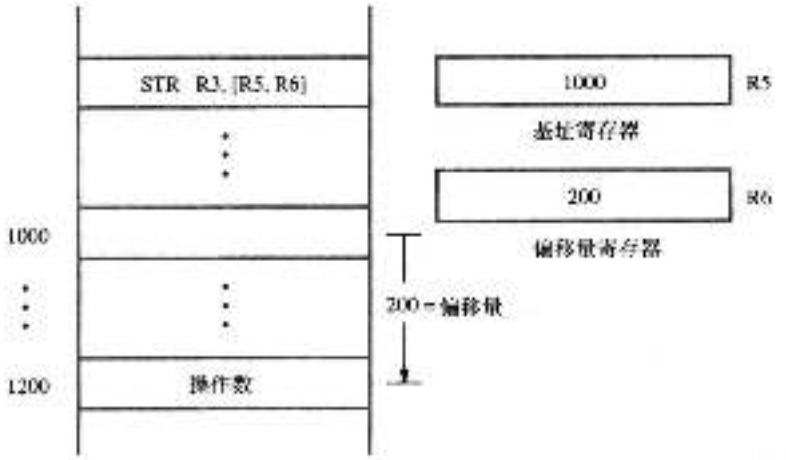
### D.3.6 寻址方式示例

图 D-3a 中给出了相对寻址方式的一个例子。操作数的地址（在指令中以符号 ITEM 的形式给出）为 1060。在 ARM 体系结构中没有绝对寻址方式，除了上一节所描述的绝对方式的限制形式。因此，当一个存储单元的地址是通过将一个地址标签放置在操作数字段中来指定的时候，汇编程序采用相对寻址方式。这通过带有一个立即数偏移量、使用 PC 作为基址寄存器的预变址方式实现。如图所示，汇编程序计算得到的偏移量是 52，因为程序执行过程中，当偏移量加到更新后的 PC 中时，PC 中包含 1008。因此该指令所产生的有效地址是 1060 = 1008 + 52。操作数必须处于更新后的 PC 之前或之后 4095 字节的范围内。如果操作数的地址超出了这个范围，那么汇编程序将显示错误，并使用其他的寻址方式来访问该操作数。

图 D-3b 给出的是一个预变址方式的例子，其偏移量放在寄存器 R6 中，基址值放在 R5 中。Store 指令（STR）将 R3 中的内容保存到存储单元 1200 的字中。




a）相对寻址方式




b）预变址寻址方式

图 D-3 存储器寻址方式示例

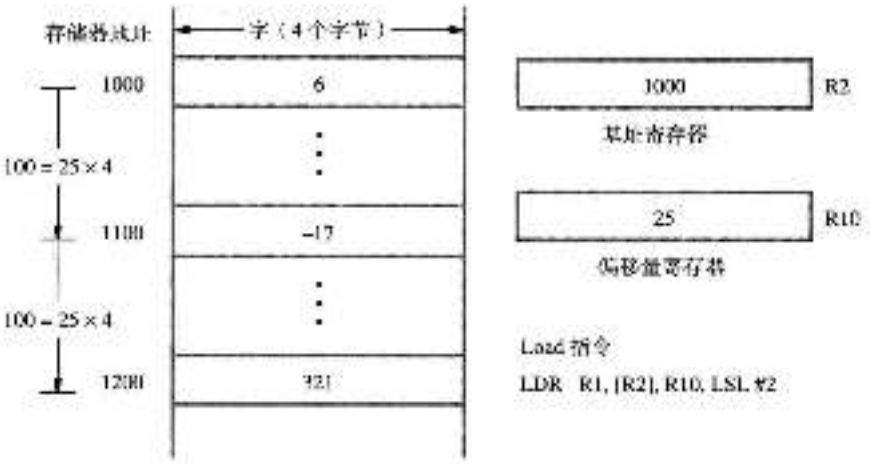
图 D-4 所示的例子说明在了后变址和预变址寻址方式中写回功能的用处。图 D-4a 显示了一个包含 25 个数的列表中的前三个数，该列表在存储器中的起始地址是 1000，两数之间间

隔 25 个字。它们组成了一个按列存储的 25 × 25 数字矩阵的第一行。存储单元 1000、1004、1008、…、1096 包含矩阵的第一列。矩阵第一行的第一个数存储在字单元 1000 中，第一行中的后续数分别保存在地址 1100、1200、…、3400 处。通过使用后变址寻址方式以及一个寄存器中的偏移量，可以方便地在一个程序循环中访问矩阵第一行中的数。假设 R2 用作基址寄存器并且包含初始地址值 1000。同时假设寄存器 R10 用来保存偏移量，并且它被装入值 25。指令

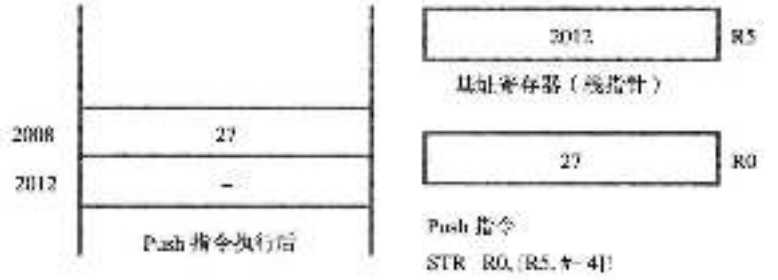
618 / 619

```
LDR   R1, [R2], R10, LSL #2
```

可以用在一个程序循环体中，在连续通过循环时，将矩阵第一行的连续元素装入寄存器 R1 中。




a）后变址寻址方式




b）预写回的变址寻址方式

图 D-4　包含写回的存储器寻址方式

下面让我们一步一步地分析它是如何工作的。第一次执行 Load 指令时，有效地址是 [R2] = 1000。因此，该地址中的数 6 被装入 R1 中。然后，写回操作将 R2 的内容从 1000 改为 1100，这样它指向第二个数 -17。这是通过将偏移寄存器 R10 的内容 25 左移 2 位之后再与 R2 的内容相加得到的。在此过程中 R10 的内容不会改变。左移两位相当于将 25 乘以 4，产生所需的偏移量 100。在第二遍循环中执行 Load 指令时，第二个数 -17 会被装入 R1 中。第三个数 321 在第三遍循环中被装入 R1 中，依次类推。

620

本例中包括将偏移量寄存器中的内容进行移位并与基址寄存器中的内容相加的操作。如

表 D-1 所示，移位后的偏移量还可以与基址寄存器的内容相减。可以选择的移动位数的范围是 0 到 31，并可以指定左移或右移。

图 D-4b 给出的例子将寄存器 R0 中的内容 27 压入程序员定义的一个栈中。寄存器 R5 用作堆栈指针。最初，它包含了当前 TOS（栈顶）元素的地址 2012。通过在下列指令中使用带写回的预变址寻址方式来执行 Push 操作：

```
STR   R0, [R5, #-4]!
```

立即数偏移量 4 与 R5 的内容相加，并且将新的值写回到 R5 中。然后，这个新栈顶的地址值 2008 被用作 Store 操作的有效地址。接着，寄存器 R0 的内容被存储到这个单元中。

## D.4 指令

ARM 体系结构中的每条指令都被编码成一个 32 位的字。存储器的访问只能通过 Load 和 Store 指令来进行。所有的算术和逻辑指令都只对处理器寄存器进行操作。

### D.4.1 Load 和 Store 指令

在上一节关于寻址方式的介绍中，我们使用 Load 和 Store 指令在存储器和寄存器之间移动单个的字操作数。操作码助记符 LDR 和 STR 用于这些指令。

存储器和寄存器之间也可以传送字节和半字的值。如果操作数是一个字节，它位于寄存器的低字节位置中。如果操作数是一个半字，它位于寄存器的低半部分中。对于 Load 指令来说，字节和半字的值可以通过使用指令助记符 LDRB 和 LDRH 进行零扩展或者通过使用 LDRSB 和 LDRSH 进行符号扩展得到 32 位的寄存器长度。字节和半字 Store 指令的助记符是 STRB 和 STRH。

**加载和存储多个操作数**

有两条指令可以加载和存储多个操作数，它们被称为块传输指令。这类指令对通用寄存器的任意子集都可以进行加载或存储。但是只允许字操作数。所使用的操作码为 LDM（Load
621 Multiple，加载多个）和 STM（Store Multiple，存储多个）。存储器操作数必须在连续的字单元中。带或不带写回的预变址和后变址的所有形式都是可用的。它们对指令中在图 D-2 所示位置上指定的基地址寄存器 Rn 进行操作。这些指令中偏移量的值始终是 4，所以不需要在指令中明确地指定。寄存器列表在指令的汇编语言表示中必须按升序出现，但并不需要是连续的。它们可在编码机器指令的位 $b_{15..0}$ 中指定，如果寄存器 Rj 在列表中，则位 $b_j = 1$。

作为一个例子，假设寄存器 R10 是基址寄存器，它的初始值为 1000。指令

```
LDMIA   R10!, {R0, R1, R6, R7}
```

将字从单元 1000、1004、1008 和 1012 传送到寄存器 R0、R1、R6 和 R7 中，因为惊叹号字符指示了写回操作，所以在最后一次传输之后，将地址值 1016 放在 R10 中。操作码中的后缀 IA 指示“Increment After”，对应于后变址。我们将在 D.4.8 节的子程序中讨论利用 Load/Store 多操作数指令来保存 / 恢复寄存器的值。

### D.4.2 算术指令

ARM 指令集中有很多可对操作数进行算术运算的指令，这些操作数可以在通用寄存器中，也可以是在指令中给出的立即操作数。这些指令的格式与图 D-2 所示的 Load 和 Store 指令是相同的，但第二个源操作数的字段标签“偏移量或 Rm”要替换为标签“立即数或 Rm”。

算术指令的基本汇编语言格式是

OP　Rd, Rn, Rm

其中 OP 码（操作码）所指定的操作是对通用寄存器 Rn 和 Rm 中的源操作数进行的。结果保存在目的寄存器 Rd 中。

1. 加法和减法

指令

ADD　R0, R2, R4

执行操作

R0 ← [R2] + [R4]

指令

SUB　R0, R6, R5

执行操作

R0 ← [R6] − [R5]　622

第二个源操作数可以用立即数方式指定。例如，指令

ADD　R0, R3, #17

执行操作

R0 ← [R3] + 17

立即操作数是一个 8 位的值，保存在编码机器指令的位 $b_{7:0}$ 中。它是 0 到 255 范围内的一个无符号数。汇编语言允许立即操作数为负值。如果一个程序中使用了指令

ADD　R0, R3, #−17

则汇编程序会将其替换为指令

SUB　R0, R3, #17

2. 第二个源操作数的移位或循环移位

当第二个源操作数被指定为一个寄存器的内容时，它们在用于运算之前可以被移位或循环移位。第 2 章 2.8.2 节中描述的逻辑左移（LSL）、逻辑右移（LSR）、算术右移（ASR）和循环右移（ROR）都是可用的。进位位 C 不参与这些运算。移位或循环移位是在第二个源操作数的寄存器名之后指定的。例如，指令

ADD　R0, R1, R5, LSL #4

的执行如下：保存在寄存器 R5 中的第二个源操作数被左移 4 位（等价于 [R5] × 16）之后再与寄存器 R1 的内容相加。将和保存在寄存器 R0 中。移位或循环移位的位数也可以在第四个寄存器中指定。

如果第二个源操作数在汇编语言指令中被指定为 0 到 255 范围内的一个立即值，那么如上所述，可从该编码的机器指令字的低字节中直接获得该操作数。对程序员来说，还可以指定一部分 32 位值。它们可在指令执行时通过对机器指令低位字节上的一个 8 位立即值按以下方式进行操作来产生：该 8 位的值首先被零扩展为 32 位，然后循环右移偶数位来产生所需的值。8 位的值和循环移位的位数是由汇编程序根据程序员所指定的立即值来决定的。这两个量被编码到指令的低 12 位中。如果用这种方式不能产生所需的值，将会报告一个错误，且程序员必须使用其他某种方式来产生所需的值，如 D.5.1 节所描述的那样。

3. 多字操作数

进位标志 C 可用于包含多字数字的加法和减法运算。有单独的指令用于此目的，其汇编语言助记符是 ADC（add with carry，带进位的加法）和 SBC（subtract with carry，带进位的减法）。例如，假设要将两个 64 位的操作数相加。假设第一个操作数包含在寄存器对 R3 和 R2　623

中，第二个操作数包含在寄存器对 R5 和 R4 中。每个操作数的高位字包含在较高编号的寄存器中。可通过使用指令

ADDS R6, R2, R4

以及后面紧跟的指令

ADC R7, R3, R5

来将这两个 64 位的操作数相加，所产生的 64 位和包含在寄存器对 R7 和 R6 中。ADDS 运算的进位输出用作 ADC 运算中的进位输入，从而来执行 64 位的加法。需要使用 ADD 指令的后缀 S 来设置 C 标志。

**4. 乘法**

乘法指令有两个基本版本。第一个版本将两个寄存器中的内容相乘，并将乘积的低 32 位保存到第三个寄存器中，乘积的高位将被丢弃。如果操作数是补码表示的数，并且它们的乘积可以用 32 位表示，那么所保留的低 32 位乘积就表示正确的结果。

例如，指令

MUL R0, R1, R2

执行操作

R0 ← [R1] × [R2]

基本乘法指令的第二个版本指定了第四个寄存器，在将结果保存到目的寄存器之前，它的内容与乘积相加。因此，指令

MLA R0, R1, R2, R3

执行操作

R0 ← ([R1] × [R2]) + [R3]

这被称为“乘法累加”操作，它经常用于信号处理应用中。

ARM 指令集中还提供了可以产生双倍字长（64 位）乘积的乘法和乘法累加指令。对于有符号和无符号操作数，这些指令还有不同的版本。

624 在操作数用于乘法运算之前，并没有提供对其进行移位或循环移位的指令。

### D.4.3 Move 指令

我们经常需要将一个寄存器的内容复制到另一个寄存器中，或者是将一个立即值装入一个寄存器中。Move（传送）指令

MOV Rd, Rm

将寄存器 Rm 中的内容复制到寄存器 Rd 中。指令

MOV Rd, #value

将一个 8 位的立即值装入目的寄存器中。

Move 指令的第二个版本，被称为 “Move Negative”，操作码助记符为 MVN，对源操作数按位求补，然后再放入目的寄存器中。回想一下，一个 8 位的立即值是 0 到 255 范围内的无符号数。MVN 指令可以按如下方式来加载补码表示的负值。假设我们希望将 −5 装入寄存器 R0 中。指令

MVN R0, #4

可获得所需的结果，因为 4 的位补是 −5 的补码表示。通常情况下，为了将 $-c$ 装入一个寄存器中，我们可以使用 MVN 指令，以及一个立即源操作数 $c-1$。为方便程序员，汇编程序接受一条指令，如

MOV R0, #-5

并将其替换为指令

MVN R0, #4

一条带有负的立即源操作数的 MOV 指令是伪指令的一个例子。汇编程序将其替换为一条实际的机器指令来获得所需的结果。

像 D.4.2 节所描述的那样，Move 指令中的源操作数可以被移位，然后再写入目的寄存器中。

**实现移位和循环移位的指令**

ARM 处理器没有显式的指令来对寄存器的内容进行移位或循环移位，如第 2 章中 2.8.2 节所述。但是，Move 指令中对源寄存器操作数进行移位或循环移位的能力提供了相同的功能。例如，指令

MOV Ri, Rj, LSL #4

可达到与通用指令

LShiftL Ri, Rj, #4

相同的结果，如 2.8.2 节所述。 625

## D.4.4 逻辑和测试指令

逻辑运算 AND（与）、OR（或）、XOR（异或）和 Bit-Clear（位清除）可分别由操作码为 AND、ORR、EOR 和 BIC 的指令实现。这些指令与算术指令具有相同的格式。指令

AND Rd, Rn, Rm

对寄存器 Rn 和 Rm 中的操作数按位进行逻辑与操作，并将结果放到寄存器 Rd 中。例如，如果寄存器 R0 包含十六进制数 02FA62CA，寄存器 R1 包含 0000FFFF，那么指令

AND R0, R0, R1

将产生结果 000062CA，并将其放到寄存器 R0 中。

Bit Clear（位清除）指令 BIC 与 AND 指令密切相关。该指令首先将操作数 Rm 按位求反之后再将其与寄存器 Rn 中的位进行与操作。使用与上例中相同的 R0 和 R1 位模式，指令

BIC R0, R0, R1

将结果 02FA0000 放到 R0 中。

**1. 数字包程序**

图 D-5 在一个 ARM 程序中说明了逻辑指令的使用。该程序将两个 4 位的十进制数字打包存放在一个存储器字节单元中。这个程序的一般版本如图 2-24 所示，并在 2.8.2 节中进行了描述。用 ASCII 码表示的十进制数字存储在字节单元 LOC 和 LOC+1 中。该程序将相应的 4 位 BCD 码打包成一个字节并放到 PACKED 中。

在为这个任务编写程序时，我们需要将一个 32 位的地址装入一个寄存器中。ARM 指令由一个 32 位的字组成，所以不能在 Move 指令中使用立即值来表示地址。

汇编程序接受如下形式的指令

LDR Ri, =ADDRESS

它将地址值 ADDRESS 装入寄存器 Ri 中。这不表示一条实际的机器指令，这是伪指令的另一个例子。汇编程序实现上述指令的方法稍后将在 D.5 节中进行讨论。当需要将一个地址值装入寄存器中时，我们通常会在程序示例中使用这条伪指令。

图 D-5 程序中的第一条指令将地址 LOC 装入寄存器 R0 中。接下来的两条 Load 指令分

别将两个 ASCII 字符（在其低 4 位中包含 BCD 数字）装入寄存器 R1 和 R2 的低位字节中。AND 指令将 R2 的高 28 位清零，将第二个 BCD 数字放在低 4 位上。该指令中的“&”字符表示立即值的十六进制形式。然后 ORR 指令将 R1 中的第一个 BCD 数字左移 4 位，并将其放到 R2 中第二个 BCD 数字的左边。然后，打包到 R2 低位字节中的两个数字被存储到单元 PACKED 中。

```
LDR     R0, =LOC            将地址 LOC 装入 R0
LDRB    R1, [R0]            将 ASCII 字符装入 R1 和 R2
LDRB    R2, [R0, #1]
AND     R2, R2, #&F         将 R2 的高 28 位清零
ORR     R2, R2, R1, LSL #4  将 R1 的内容左移之后，与
                            R2 的内容进行逻辑或运算，
                            并将结果保存到 R2 中
STRB    R2, PACKED          将打包的 BCD 数字存到
                            PACKED 中
```

图 D-5 将两个 4 位的十进制数字打包到一个字节中的程序

2. Test（测试）指令

Test（TST）和 Test Equivalence（TEQ）指令分别对字操作数执行逻辑 AND 和 XOR 操作，然后根据结果对条件码标志进行设置。这些指令不在寄存器中保存结果。它们可以用来测试一个未知的位模式如何与一个已知的位模式相匹配，后面可跟一条 Branch 指令，其转移条件基于这些测试指令的结果。

例如，Test（测试）指令

TST Rn, #1

执行一个 AND 操作，以测试寄存器 Rn 的最低位是否等于 1。如果测试结果是正的，也就是说，如果寄存器 Rn 中内容的最低位等于 1，那么 AND 操作的结果就为 1，且 Z 位被清零。使用这种类型的指令可以检查 I/O 设备寄存器中的状态位。

Test Equivalence（测试等价）指令

TEQ Rn, #5

执行 XOR 操作来测试寄存器 Rn 中是否包含值 5。如果 Rn 中包含 5，那么按位异或操作的结果将是零，且 Z 位将被置为 1。

### D.4.5 比较指令

Compare（比较）指令

CMP Rn, Rm

626 ~ 627

执行操作

[Rn] – [Rm]

并根据减法运算的结果对条件码标志进行设置，结果本身将被丢弃。

Compare Negative 指令

CMN Rn, Rm

执行操作

[Rn] + [Rm]

并根据运算的结果来设置条件码标志，运算的结果将被丢弃。

在这两条指令中，第二个操作数还可以是一个立即值。第二个操作数的任一版本都可以如 D.4.2 节所描述的那样进行移位。

### D.4.6　设置条件码标志

Compare 和 Test 指令总是会修改条件码标志。它们后面通常跟着条件转移指令，条件转移指令将在下一节描述。算术、逻辑和 Move 指令只有在操作码字段的一个位中明确说明要修改条件码标志时，才会影响条件码标志。这种方式通过在汇编语言操作码助记符后加上后缀 S 来表示。例如，指令

```
ADDS   R0, R1, R2
```

是要设置条件代码标志的，而

```
ADD    R0, R1, R2
```

就不设置条件码标志。

### D.4.7　转移指令

条件转移指令包含一个 24 位的补码值，该值按如下方式生成转移偏移量。当执行该指令时，这个值被左移 2 位（因为所有的转移目标地址都是字对齐的），然后被符号扩展为 32 位，以生成偏移量。该偏移量与程序计数器的更新内容相加以生成转移目标地址。在图 D-6 中给出了一个例子。BEQ 指令（如果等于 0 就转移）在 Z 标志被置为 1 时便会产生转移。汇编程序会计算出指令中适当的 24 位值。在本例中，它是 92 / 4 − 23。

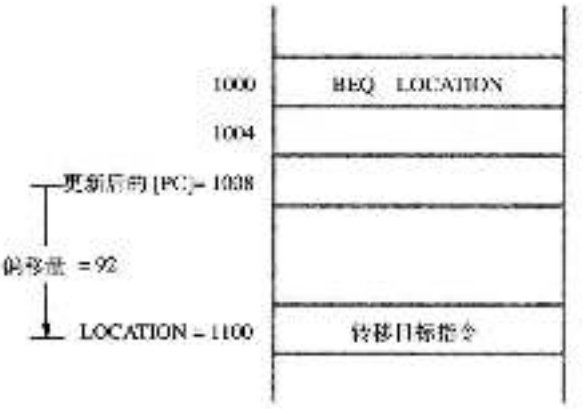



图 D-6　转移指令目标地址的确定

对其进行测试以确定是否产生转移的条件是在指令字的高 4 位 $b_{31-28}$ 中指定的。Branch（转移）指令与其他的 ARM 指令一样按照其他相同的方式执行；也就是说，只有条件码标志的当前状态与相应指令的条件字段所指定的条件相一致时，才会产生转移。表 D-2 给出了所有的条件。汇编程序接受操作码 B 作为一个无条件转移。在这里没有必要使用后缀 AL。

表 D-2　ARM 指令的条件字段编码

| 条件字段 $b_{31} \cdots b_{28}$ | 条件后缀 | 名　称 | 条件码测试 |
|---|---|---|---|
| 0000 | EQ | 等于（0） | Z = 1 |
| 0001 | NE | 不等于（非 0） | Z = 0 |
| 0010 | CS/HS | 进位设置 / 无符号大于或等于 | C = 1 |
| 0011 | CC/LO | 进位清零 / 无符号小于 | C = 0 |
| 0100 | MI | 负号（负的） | N = 1 |
| 0101 | PL | 加号（正或零） | N = 0 |
| 0110 | VS | 溢出 | V = 1 |
| 0111 | VC | 未溢出 | V = 0 |

（续）

| 条件字段 $b_{31}\cdots b_{28}$ | 条件后缀 | 名　称 | 条件码测试 |
|---|---|---|---|
| 1000 | HI | 无符号大于 | $\overline{C} \vee Z = 0$ |
| 1001 | LS | 无符号小于或等于 | $C \vee Z = 1$ |
| 1010 | GE | 有符号大于或等于 | $N \oplus V = 0$ |
| 1011 | LT | 有符号小于 | $N \oplus V = 1$ |
| 1100 | GT | 有符号大于 | $Z \vee (N \oplus V) = 0$ |
| 1101 | LE | 有符号小于或等于 | $Z \vee (N \oplus V) = 1$ |
| 1110 | AL | 总是 | |
| 1111 | | 未使用 | |

628 ~ 629

在计算转移目标地址的同时，由于指令的流水线执行，PC 的内容已经被更新为 Branch 指令之后两个字的那条指令的地址。如 D.3.2 节所述，如果 Branch 指令的地址是 1000，且转移目标地址为 1100，如图 D-6 所示，那么由于在计算转移目标地址 1100 时，更新后的 PC 内容将是 1000 + 8 = 1008，所以偏移量为 92。

**1. 用于数值相加的程序**

我们现在已经描述了足够多的 ARM 指令，使我们能够介绍第 2 章给出的一些一般形式的程序。图 D-7 给出了一个将列表中的数值相加的程序，它模仿图 2-26 中的程序编写。单元 N 中包含列表中数据的个数，单元 SUM 用来保存相加的和。第一条和最后一条指令使用相对寻址方式来执行 Load 和 Store 操作。假设存储单元 N 和 SUM 都在相对于 PC 的偏移量所能达到的范围内。第一个相加数的地址 NUM1 被第二条指令装入寄存器 R2 中。循环中的第一条指令使用了带写回的后变址寻址方式。这种方式可达到与图 2-26 中的自增寻址方式相同的效果。

```
        LDR   R1, N          将计数值装入 R1
        LDR   R2, =NUM1      将地址 NUM1 装入 R2
        MOV   R0, #0         清空累加器 R0
LOOP    LDR   R3, [R2], #4   将下一个数装入 R3
        ADD   R0, R0, R3     将数加到 R0 中
        SUBS  R1, R1, #1     递减循环计数器 R1
        BGT   LOOP           如果没有完成，则转移回到前面
        STR   R0, SUM        保存累加和
```

图 D-7　用于数值相加的程序

**2. 用于将测验成绩相加的程序**

利用 ARM 变址寻址方式的灵活性可以为图 2-11 所示的将学生测验成绩相加的程序编写一个高效的版本。假设存储器中的数据布局与图 2-10 所示的相同。

图 D-8 给出了该程序。在程序的开头地址 N 被装入寄存器 R2 中。寄存器 R2 用作后变址寻址方式中的变址寄存器，用来访问连续学生记录中的测验成绩。注意偏移量 8、4 以及 4 连同写回操作是如何使得寄存器 R2 的内容在每遍循环（包括第一遍循环）时都能正确地增加，以跳过学生 ID 单元的。偏移值的灵活性和写回功能的结合意味着在每遍循环结束时不需要图 2-11 程序中的最后一条 Add 指令来增加指针寄存器 R2 的值。

630

## D.4.8　子程序链接指令

转移与链接（BL）指令通常用来调用一个子程序。它的操作方式与其他转移指令的操作方式相同，只是增加了一步，将返回地址（BL 指令的下一条指令地址）装入寄存器 R14（它

实际上作为链接寄存器使用）中。由于子程序可能是嵌套的，所以链接寄存器的内容必须在嵌套调用另一个子程序之前保存在处理器堆栈中。寄存器 R13 通常用作处理器堆栈的指针。

```
        LDR   R2, =N        将地址 N 送入 R2
        MOV   R3, #0
        MOV   R4, #0
        MOV   R5, #0
        LDR   R6, N         加载 n 值
LOOP    LDR   R7, [R2, #8]! 将当前学生的测验 1 成绩加到部分和上
        ADD   R3, R3, R7
        LDR   R7, [R2, #4]! 将当前学生的测验 2 成绩加到部分和上
        ADD   R4, R4, R7
        LDR   R7, [R2, #4]! 将当前学生的测验 3 成绩加到部分和上
        ADD   R5, R5, R7
        SUBS  R6, R6, #1    减计数器
        BGT   LOOP          如果没完成就循环回到前面
        STR   R3, SUM1      保存测验 1 的总和
        STR   R4, SUM2      保存测验 2 的总和
        STR   R5, SUM3      保存测验 3 的总和
```

图 D-8 对 2-11 中将测验成绩相加程序的 ARM 版本

图 D-9 给出的是使用寄存器传递参数来将图 D-7 中的程序重写为一个子程序的程序。调用程序利用寄存器 R1 和 R2 将数值列表的大小和第一个数的地址传送给子程序。子程序利用寄存器 R0 向调用程序返回计算和。该子程序还用到了寄存器 R3。因此，它的内容连同链接寄存器 R14 的内容一起通过使用 STMFD 指令保存到堆栈中。该指令中的后缀 FD 说明堆栈是向低地址方向增长的，在将字压入堆栈之前，堆栈指针 R13 要预先递减。LDMFD 指令恢复寄存器 R3 中的内容并将所保存的返回地址弹出到 PC（R15）中，自动执行返回操作。 631

```
调用程序
        LDR     R1, N
        LDR     R2, =NUM1
        BL      LISTADD
        STR     R0, SUM
        :

子程序
LISTADD STMFD   R13!, {R3, R14}   将 R3 和 R14 中的返回地址保存到堆栈中，
                                  R13 用作堆栈指针
        MOV     R0, #0
LOOP    LDR     R3, [R2], #4
        ADD     R0, R0, R3
        SUBS    R1, R1, #1
        BGT     LOOP
        LDMFD   R13!, {R3, R15}   恢复 R3 并将返回地址装入 PC（R15）中
```

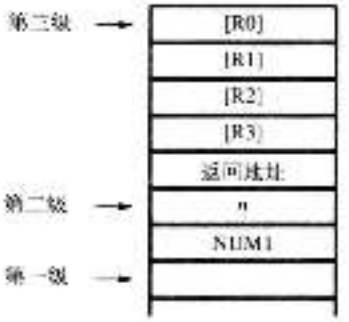
图 D-9 将图 D-7 的程序编写为一个子程序，通过寄存器传递参数

图 D-10a 给出的是采用处理器堆栈传递参数来将图 D-7 中的程序重写为一个子程序的程序，调用程序的前四条指令将参数 NUM1 和 $n$ 压入栈中。子程序中的寄存器 R0 和 R3 与图 D-7 中的用途相同。它们的内容连同 R14 中的返回地址一起由子程序中的第一条指令保存到堆栈中。图 D-10b 给出了不同时刻堆栈中的内容。在参数已经压入堆栈并且执行完调用指令

（BL）之后，栈顶处于第二级。当子程序中的第一条指令将全部的寄存器保存完之后，栈顶处于第三级。接下来的两条指令利用偏移量 20 和 24 来访问堆栈中的参数 *n* 和 NUM1 并将其分别装入寄存器 R1 和 R2 中。当和已经在 R0 中累加完后，由 Store 指令（STR）将其写入到栈中，覆盖 NUM1 的内容。

```
（假设栈顶在第一级以下）
调用程序
            LDR     R0, =NUM1       将 NUM1 压入堆栈中
            STR     R0, [R13, #-4]!
            LDR     R0, N           将 n 压入堆栈中
            STR     R0, [R13, #-4]!
            BL      LISTADD
            LDR     R0, [R13, #4]   将和移动到存储单元 SUM 中
            STR     R0, SUM
            ADD     R13, R13, #8    从堆栈中移除参数
            :
子程序
LISTADD     STMFD   R13!, {R0-R3, R14}  保存寄存器
            LDR     R1, [R13, #20]  从堆栈中载入参数
            LDR     R2, [R13, #24]
            MOV     R0, #0
LOOP        LDR     R3, [R2], #4
            ADD     R0, R0, R3
            SUBS    R1, R1, #1
            BGT     LOOP
            STR     R0, [R13, #24]  将和放入堆栈
            LDMFD   R13!, {R0-R3, R15}  恢复寄存器并返回
```

a）调用程序和子程序




b）不同时刻的栈顶

图 D-10 将图 D-7 的程序编写为一个子程序，通过堆栈传递参数

子程序的最后一个例子是对嵌套调用情况的处理。图 D-11 给出了图 2-21 中程序的 ARM 代码。第一个和第二个子程序对应的堆栈结构如图 D-12 所示。寄存器 R12 用作结构指针。为了程序的可读性，在本例中一些寄存器使用了符号名称。寄存器 R12（结构指针）、R13（堆栈指针）、R14（链接寄存器）和 R15（程序计数器）被分别标记为 FP、SP、LR 和 PC。汇编程序可预定义 LR 和 PC，而汇编指示符 RN 可以用来定义名称 FP 和 SP，这将在 D.5 节中进行介绍。

632
|
633

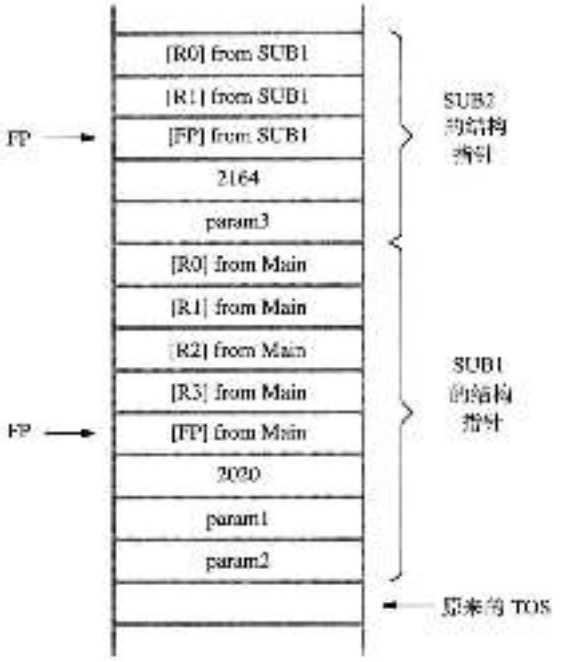
| 存储单元 | | 指令 | | 注释 |
|---|---|---|---|---|
| **主程序** | | | | |
| | | ⋮ | | |
| 2000 | | LDR | R10, PARAM2 | 将参数放入堆栈 |
| 2004 | | STR | R10, [SP, #-4]! | |
| 2008 | | LDR | R10, PARAM1 | |
| 2012 | | STR | R10, [SP, #-4]! | |
| 2016 | | BL | SUB1 | |
| 2020 | | LDR | R10, [SP] | 保存 SUB1 的结果 |
| 2024 | | STR | R10, RESULT | |
| 2028 | | ADD | SP, SP, #8 | 从堆栈中移除参数 |
| 2032 | | 下一条指令 | | |
| | | ⋮ | | |
| **第一个子程序** | | | | |
| 2100 | SUB1 | STMFD | SP!, {R0–R5, FP, LR} | 保存寄存器 |
| 2104 | | ADD | FP, SP, #16 | 载入结构指针 |
| 2108 | | LDR | R0, [FP, #8] | 载入参数 |
| 2112 | | LDR | R1, [FP, #12] | |
| | | ⋮ | | |
| | | LDR | R2, PARAM3 | 将参数放入堆栈 |
| | | STR | R2, [SP, #-4]! | |
| 2160 | | BL | SUB2 | |
| 2164 | | LDR | R2, [SP], #4 | 将 SUB2 的结果弹出放入 R2 |
| | | ⋮ | | |
| | | STR | R3, [FP, #8] | 将结果放入堆栈 |
| | | LDMFD | SP!, {R0–R3, FP, PC} | 恢复寄存器并返回 |
| **第二个子程序** | | | | |
| 3000 | SUB2 | STMFD | SP!, {R0, R1, FP, LR} | 保存寄存器 |
| | | ADD | FP, SP, #8 | 载入结构指针 |
| | | LDR | R0, [FP, #8] | 载入参数 |
| | | ⋮ | | |
| | | STR | R1, [FP, #8] | 将结果放入堆栈 |
| | | LDMFD | SP!, {R0, R1, FP, PC} | 恢复寄存器并返回 |

图 D-11 嵌套子程序

调用程序和子程序的结构与图 2-21 相同。ARM 的特征如下：返回地址和结构指针中原来的内容由每个子程序中的第一条指令保存到栈中。第二条指令将结构指针设置成指向它所保存的值，如图 D-12 所示。这是与图 2-20 和图 2-22 中的结构指针位置相一致的。于是，参数就可以在偏移量为 8、12、…处进行引用了。每个子程序中的最后一条指令都用来恢复结构指针以及其他寄存器的已保存值，并将返回地址从堆栈弹出到 PC 中。

## D.5 汇编语言

ARM 汇编语言利用汇编指示（assembler directive）来保留存储空间、给地址标号和常量符号分配数值、定义程序和数据块在存储器中的放置位置，以及指明源程序文本的结束位置。这些汇编指示的一般形式在 2.5.1 节中已经描述过了。




634
:
635

图 D-12 图 D-11 的栈结构

图 D-13 通过给出图 D-7 中程序的一个完整源程序来列举一些 ARM 的汇编指示。AREA 指示符使用参数 CODE 或 DATA 来表示包含程序指令或数据的存储器块的开始。还需要其他的参数来说明代码和数据块在特定存储区域中的放置位置。ENTRY 指示符说明程序的执行是从后面的 LDR 指令开始的。

| | 存储器地址标号 | 操作 | 寻址或数据信息 |
|---|---|---|---|
| 汇编指示 | | AREA | CODE |
| | | ENTRY | |
| 产生机器指令的语句 | | LDR | R1, N |
| | | LDR | R2, POINTER |
| | | MOV | R0, #0 |
| | LOOP | LDR | R3, [R2], #4 |
| | | ADD | R0, R0, R3 |
| | | SUBS | R1, R1, #1 |
| | | BGT | LOOP |
| | | STR | R0, SUM |
| 汇编指示 | | AREA | DATA |
| | SUM | DCD | 0 |
| | N | DCD | 5 |
| | POINTER | DCD | NUM1 |
| | NUM1 | DCD | 3, -17, 27, -12, 322 |
| | | END | |

图 D-13 图 D-7 中程序的汇编语言源程序

在紧跟代码区之后的数据区中，DCD 指示符用来标记和初始化数据操作数。字单元 SUM 和 N 被前两个 DCD 指示符分别初始化为 0 和 5。地址 NUM1 被下一个 DCD 指示符放到单元

POINTER 中。指令

LDR R2, POINTER

和数据声明

POINTER DCD NUM1

的组合是实现图 D-7 中伪指令

636

LDR R2, =NUM1

的一种方式，如 D.5.1 节所述。最后的 DCD 指示符指明要相加的五个数将被放到从 NUM1 开始的连续存储字单元中。

以十六进制记数法表示的常量由前缀“&”指定，以基数 $n$（$n$ 在 2 和 9 之间）表示的常量由一个表明基数的前缀指定。例如，2_101100 表示一个二进制常量，8_70375 表示一个八进制常量。基数是 10 的常量不需要前缀。

EQU 指示符可用来声明常量的符号名称。例如，语句

TEN EQU 10

允许 TEN 代替十进制常量 10 在程序中使用。

使用与寄存器的用途相关的符号名称是非常方便的。RN 指示符就是为此目的而设置的。例如

COUNTER RN 3

为寄存器 R3 建立了名称 COUNTER。寄存器名称 R0 到 R15、PC（R15）以及 LR（R14）都是由汇编程序预定义的。

### 伪指令

伪指令（pseudoinstruction）是汇编语言指令，它执行一些所需的操作，但并不直接对应于真正的机器指令。汇编程序接受这样的指令，并将其替换为执行所需操作的真正的机器指令。在某些情况下，可能需要一个真正机器指令的短序列。但伪指令为程序员提供了方便。

我们已经在 D.4.3 节和 D.4.4 节中看到了伪指令的例子。在这里，我们将对伪指令进行更完整的讨论，可使用伪指令将一个 32 位的数值或地址值装入一个寄存器中。

**1. 加载 32 位值**

伪指令

LDR Rd, =value

可用来将任何一个 32 位值装入寄存器中。值前面的等号可将该指令与真正的 Load 指令区别开来。如果该值能由一条 MOV 或者 MVN 指令形成并装入 Rd 中，那么汇编程序就会选择这么做。如果这是不可能的，那么汇编程序将在真正的 LDR 指令中使用相对寻址方式来从一个存储单元中加载该值，该存储单元在汇编程序所分配的数据区中。

例如，指令

637

LDR R3, =127

将被替换为指令

MOV R3, #127

但指令

LDR R3, =&A123B456

将被替换为

LDR R3, MEMLOC

其中十六进制值 A123B456 是存储单元 MEMLOC 的内容，通过相对寻址方式来访问。

地址标号的值也可以用这种方式加载到寄存器中，就像我们在本附录的大部分程序示例中所做的那样。

**2. 加载地址值**

除了刚刚所描述的方法，当一个地址接近程序计数器 PC（R15）中的当前值时，还有一种更有效的方法可用来将该地址装入一个寄存器中。这种方法避免了将所需的地址值放到数据区中的需要。

伪指令

ADR Rd, LOCATION

将 LOCATION 所表示的 32 位地址装入 Rd 中。ADR 指令的执行如下：汇编程序计算从 PC 的当前值到 LOCATION 的偏移量。如果 LOCATION 是在前方，那么 ADR 将由下列的指令实现：

ADD Rd, R15, #offset

如果 LOCATION 是在后方，那么将使用指令

SUB Rd, R15, #offset

来实现 ADR 伪指令。

在这两种情况下，偏移量都是 0 到 255 范围内的一个 8 位无符号数，就像前面介绍算术指令时所描述的那样。通过对一个 8 位的值进行循环移位可产生某些较大的偏移值，就像 D.4.2 节所描述的那样。

## D.6 示例程序

在本节中，我们将给出 2.12 节介绍过的向量点积程序和字符串搜索程序的 ARM 版本。
638 我们将只描述 ARM 代码与通用程序不同的部分。

### D.6.1 向量点积程序

图 D-14 给出了一个计算 A 和 B 这两个向量的点积的程序。前两条指令将这两个向量的起始地址 AVEC 和 BVEC 装入寄存器 R1 和 R2 中。这里使用相对寻址方式来访问 N 和 DOTPROD 的内容。在循环的前两条指令中使用后变址寻址方式（总是包含写回）。乘法累加指令（MLA）执行必要的算术运算。它将 R4 和 R5 中的向量元素相乘，并将得到的乘积累加到 R0 中。

```
        LDR   R1, =AVEC     R1 指向向量 A
        LDR   R2, =BVEC     R2 指向向量 B
        LDR   R3, N         R3 为循环计数器
        MOV   R0, #0        R0 累加点积
LOOP    LDR   R4, [R1], #4  装入 A 元素
        LDR   R5, [R2], #4  装入 B 元素
        MLA   R0, R4, R5, R0  A、B 元素相乘，结果累加到 R0 中

        SUBS  R3, R3, #1    递减计数器
        BGT   LOOP          如果没完成，则转移回到前面
        STR   R0, DOTPROD   保存点积
```

图 D-14 点积程序

### D.6.2 字符串搜索程序

图 D-15 中的 ARM 程序与图 2-30 中的一般程序非常相似。有两个需要注意的差别。ARM 程序中 LOOP2 的前两条指令使用了后变址寻址方式，这就不需要使用一般程序 LOOP2 中的两条 ADD 指令；这里需要使用三对 ARM 指令 CMP/BNE、CMP/BGT 以及 CMP/BGE 来实现三条一般的 Branch_if 指令。

```
        LDR     R2, =T          将地址 T 装入 R2
        LDR     R3, =P          将地址 P 装入 R3
        LDR     R4, N           获取值 n
        LDR     R5, M           获取值 m
        SUB     R4, R4, R5      计算 n – m
        ADD     R4, R2, R4      R4 是 T(n – m) 的地址
        ADD     R5, R3, R5      R5 是 P(m) 的地址
LOOP1   MOV     R6, R2          用 R6 遍历字符串 T
        MOV     R7, R3          用 R7 遍历字符串 P
LOOP2   LDRB    R8, [R6], #1    比较字符串 T 和 P 中
        LDRB    R9, [R7], #1    的一对字符
        CMP     R8, R9
        BNE     NOMATCH
        CMP     R5, R7          检查是否在 P(m) 中
        BGT     LOOP2           如果没有完成再次循环
        STR     R2, RESULT      保存 T(i) 的地址
        B       DONE
NOMATCH ADD     R2, R2, #1      指向 T 中的下一个字符
        CMP     R4, R2          检查是否在 T(n – m) 中
        BGE     LOOP1           如果没有完成再次循环
        MOV     R8, #-1         没有发现匹配
        STR     R8, RESULT
DONE    下一条指令
```

图 D-15 字符串搜索程序

## D.7 操作模式和异常

ARM 处理器有七种操作模式。应用程序是在用户模式（User mode）下运行的。有五种异常模式。当发生一个异常时进入其中的一种异常模式。第七种操作模式是系统模式（System mode），只能从其中一种异常模式进入系统模式，这将在 D.7.3 节中进行讨论。

五种异常模式以及导致进入异常模式的异常总结如下：

- 当外部设备为获得紧急服务而发出一个快速中断请求时，进入快速中断（FIQ）模式。
- 当外部设备发出一个正常的中断请求时，进入普通中断（IRQ）模式。
- 在系统上电或复位时，或者当用户程序执行一条软件中断指令（SWI）来调用一个操作系统程序执行时，进入管态（Supervisor，SVC）模式。
- 在当前程序试图读取一条（个）可导致存储器访问冲突的指令或数据操作数时，进入存储器访问冲突（终止）模式。
- 在当前程序试图执行一条未实现的指令时，进入未实现的指令（未定义）模式。

639 | 640

状态寄存器中的中断禁止位 I 和 F 可用来确定当在相应的中断线（IRQ 和 FIQ）上发出中断请求时，处理器是否被中断。如果禁止位为 1，则处理器不会被中断；如果禁止位为 0，则处理器就会被中断。

五种异常模式和系统模式都是特权模式（privileged mode）。当处理器处于特权模式下时，允许访问状态寄存器（图 D-1 中的 CPSR），所以可以对模式位和中断禁止位进行操作，这可以通过一些不可在用户模式（非特权模式）下使用的指令来完成。

### D.7.1 后备寄存器

当处理器运行在用户模式或系统模式下时，都会使用图 D-1 所示的 16 个普通的处理器寄存器。当发生一个异常且从用户模式切换到其中一种异常模式时，16 个寄存器中的某一些会被同等数量的后备寄存器替换，如 D.2 节所述。被替换的寄存器其内容保持不变。每一种异常

模式都有一组不同的后备寄存器，在图 D-16 中用灰色显示。

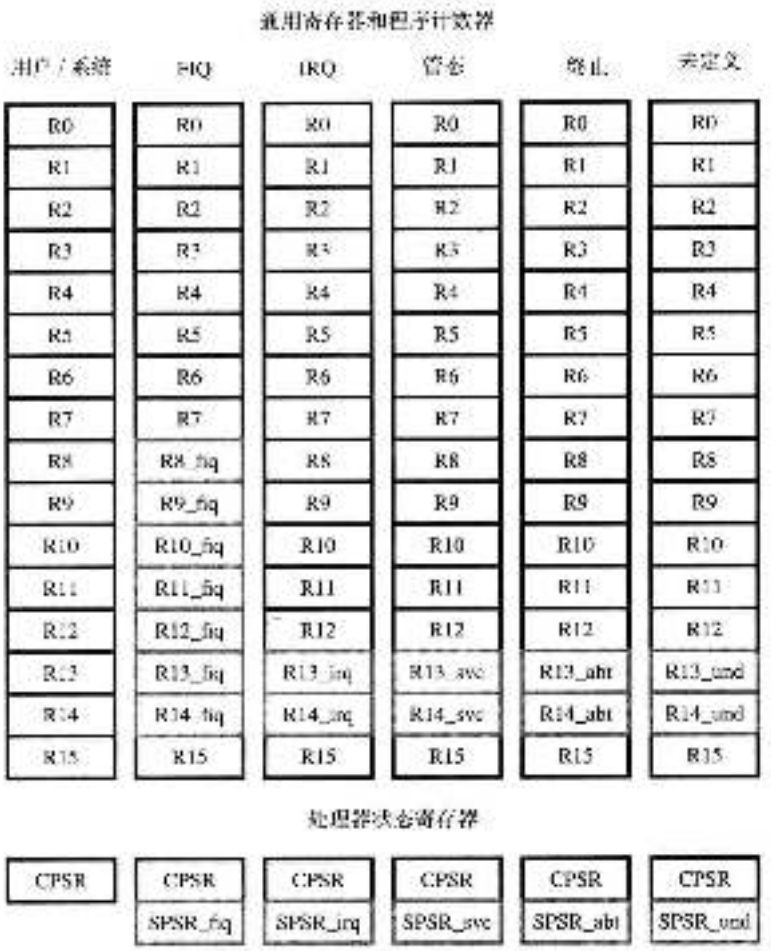



图 D-16 ARM 处理器不同模式下的可访问寄存器

当异常发生时，从用户模式切换到相应的异常模式时将执行如下操作：

1）程序计数器（R15）的内容被装入异常模式的后备链接寄存器（R14_mode）中。

2）状态寄存器（CPSR）的内容被装入后备的备份状态寄存器（SPSR_mode）中。

3）改变 CPSR 的模式位来表示相应的异常模式，并将中断禁止位 I 和 F 进行适当的设置。

4）将相应的异常向量地址装入程序计数器（R15）中，该地址处的指令被提取并执行以开始异常服务程序。

活跃的堆栈指针寄存器（R13_mode）始终指向处理器堆栈的栈顶元素，该堆栈位于为相关的异常模式分配的存储区域中。R13_mode 的内容由操作系统初始化。

当异常服务程序执行完成时，需要返回到用户模式，继续执行被中断的程序。这是通过将模式链接寄存器（R14_mode）中的内容传送给程序计数器以及将备份状态寄存器（SPSR_mode）中的内容传送给状态寄存器（CPSR）来完成的。

我们已经对刚刚描述的从用户模式切换到异常模式，然后又切换回用户模式的操作进行了笼统地介绍。实际的异常和所进入的模式的不同也使操作细节有所不同。在下面的章节中我们将进一步描述这些细节。

## D.7.2 异常的类型

异常有七种。表 D-3 列出了这些异常以及异常发生时所进入的处理器模式，表中还列出了异常的向量地址。在地址空间低端的这些字单元必须包含可跳转到异常服务程序开始的转移指令。快速中断程序可以立即开始，而不需要转移指令，因为它的向量地址（28）在列表的最后。当多个异常同时发生时，它们被服务的优先顺序如表 D-3 的最后一列所示。

641 / 642

表 D-3 异常和处理器模式

| 异常 | 所进入的处理器模式 | 向量地址 | 优先级（最高 = 1） |
|---|---|---|---|
| 快速中断 | FIQ | 28 | 3 |
| 普通中断 | IRQ | 24 | 4 |
| 软件中断 | 管态 (SVC) | 8 | — |
| 上电 / 复位 | 管态 (SVC) | 0 | 1 |
| 数据访问冲突 | 终止 | 16 | 2 |
| 指令访问冲突 | 终止 | 12 | 5 |
| 未实现的指令 | 未定义 | 4 | 6 |

对异常更为详细的描述如下：

- 快速中断（FIQ）和普通中断（IRQ）——输入 / 输出设备使用两根中断请求线中的一根来请求服务。FIQ 中断用于一台或少数几台设备需要快速响应的情况。除了堆栈指针寄存器 R13_fiq 和链接寄存器 R14_fiq 之外，图 D-16 所示的 FIQ 处理器模式的后备寄存器还包括五个通用寄存器 R8_fiq 到 R12_fiq。如果这五个通用寄存器能为 FIQ 中断服务程序提供足够的工作空间，那么将不再需要保存和恢复其他的用户模式寄存器。其他所有的 I/O 设备都使用 IRQ 中断线来请求服务。
- 软件中断——用户程序通过执行 SWI 指令来请求操作系统的服务。这是一种可导致处理器进入管态模式的异常。指令中的参数字段指明了所请求的服务，并且管态程序可以访问该参数字段。
- 上电 / 复位——这是优先级最高的一种异常。它使处理器处于一个已知的初始状态，使得操作系统软件可以正常开始或重新启动。该异常发生时，任何程序的执行都会被中止。
- 数据和指令访问冲突——处理器的实现中可能包括一个存储器管理单元，该单元将程序限制到其指令和数据地址空间的有效区域中。我们需要这样一个单元来实现第 8 章所描述的虚拟存储器。如果处理器发出的指令提取地址或数据操作数访问地址在其有效区域之外，那么就会发生一个异常并进入终止（Abort）模式。该模式也能处理这样一种情况：地址是有效的，但目前并未映射到主存中，需要从辅助存储设备进行传输。
- 未实现的指令——如果处理器试图执行一条指令，但该指令并未在硬件中实现，就会发生一个异常并进入未定义（Undefined）模式。例如，浮点算术运算可以被特殊的硬件支持，但在当前的处理器中可能还没有实现。在这种情况下，异常的发生可能会使得处理器去执行实现该浮点运算的软件。

643

## D.7.3 系统模式

系统模式是同用户模式使用同样寄存器的一种特权模式。系统模式只能从另外一种异常模式进入。该模式的目的是为了在异常处理期间能很容易地链接到子程序中，而不破坏链接寄存器 R14_mode 的内容。当处于系统模式下时，子程序调用指令使用正常的链接寄存器 R14。从所有的子程序调用返回后，重新进入原来的异常模式，并可重新访问链接寄存器 R14_mode。

### D.7.4 异常处理

当出现一个异常后，从用户模式切换到相应的异常模式，然后再返回到用户模式所需的一般操作已经在 D.7.1 节中进行了简要介绍。这些操作的细节有所不同，主要取决于异常和所进入的异常模式。在这里，我们考虑一些这样的细节。

1. 流水执行、程序计数器和状态寄存器

ARM 处理器将连续指令的提取和执行重叠起来，以提高指令的吞吐量。这种技术被称为指令的流水线执行。我们已经在第 6 章中进行了描述。在指令的流水线执行过程中，程序计数器的更新按如下步骤进行。假设处理器从地址 A 提取指令 $I_1$。PC 的内容递增为 A + 4，然后开始执行 $I_1$。$I_1$ 的执行完成之前，处理器从地址 A + 4 提取指令 $I_2$，然后将 PC 递增为 A + 8。

现在假设在 $I_1$ 执行结束时，处理器发现已经收到了一个普通的中断请求（IRQ）。处理器执行 D.7.1 节所描述的操作，进入 IRQ 异常模式去处理该中断。它将 CPSR 的内容复制到 SPSR_irq 中，将 PC 的内容（现在是 A + 8）复制到链接寄存器 R14_irq 中。已经提取但尚未完全执行的指令 $I_2$ 将被丢弃。而中断服务程序必须返回到该指令（$I_2$）。所以中断服务程序必须从 R14_irq 中减去 4，然后再将其内容作为返回地址。还必须恢复已保存的状态寄存器的副本。所需的操作可由单一条指令

SUBS PC, R14_irq, #4

来实现。该指令从 R14_irq 中减去 4 并将结果保存到 PC 中。操作码中的后缀 S 通常是指“设置条件码”。但是，当指令的目的寄存器是 PC 时，后缀 S 会使得处理器将 SPSR_irq 的内容复制到 CPSR 中，从而完成返回到被中断的程序所需的操作。 644

表 D-4 从异常返回时的地址修正

| 异 常 | 已保存的地址 | 所需的返回地址 | 返回指令 |
|---|---|---|---|
| 未定义的指令 | PC+4 | PC+4 | MOVS PC, R14_und |
| 软件中断 | PC+4 | PC+4 | MOVS PC, R14_svc |
| 指令终止 | PC+4 | PC | SUBS PC, R14_abt, #4 |
| 数据终止 | PC+8 | PC | SUBS PC, R14_abt, #8 |
| IRQ | PC+4 | PC | SUBS PC, R14_irq, #4 |
| FIQ | PC+4 | PC | SUBS PC, R14_fiq, #4 |

PC 是导致异常的指令地址。对于 IRQ 和 FIQ 来说，它是第一条由于中断而未被执行的指令地址

在执行 SWI 指令而触发一个软件中断的情况下，保存在 R14_svc 中的值是正确的返回地址。从一个软件中断返回的操作可以通过使用指令

MOVS PC, R14_svc

来完成，该指令也将 SPSR_svc 的内容复制到 CPSR 中。

表 D-4 给出了正确的返回地址值以及表 D-3 中除了上电/复位（该异常会中止任何当前正在执行的程序）之外的每个异常用来返回到被中断程序的指令。注意，对于数据访问或指令访问冲突，返回地址是导致异常的指令地址。因为在解决了冲突之后，该指令必须被取新执行。

2. 对状态寄存器中的位进行操作

当处理器在特权模式下运行时，可以使用特殊的 Move 指令 MRS 和 MSR 来向或从通用寄存器传输当前或已保存的处理器状态寄存器的内容。例如，指令

MRS Rd, CPSR

将 CPSR 的内容复制到寄存器 Rd 中。类似地，指令

MSR SPSR, Rm

将寄存器 Rm 的内容复制到 SPSR_mode 中。

状态寄存器的内容被装入寄存器之后，可以使用逻辑指令来对单个的位进行操作。然后，该寄存器的内容可被复制回到状态寄存器中以完成所需的更改。例如，这些步骤可用于设置或清除异常服务程序中的中断禁止位。我们将在 D.8 节中处理 I/O 设备中断时看到如何完成这样的操作。 [645]

**3. 异常服务程序的嵌套**

回顾一下，子程序的嵌套是比较容易的，可通过将链接寄存器的内容保存到与子程序（该子程序调用了另一个子程序）相关的栈结构中来实现。但是，当一个异常服务程序被一个具有更高优先级且其服务程序运行在不同处理器模式下的异常打断时，则不需要该操作。这是因为每种模式都有其自己的后备链接寄存器。

例如，假设一个 IRQ 模式的例程正在处理一个普通中断，这时接收到一个需要获得快速服务的中断。该例程被中断并进入 FIQ 模式来处理第二个中断。被中断而转去处理 IRQ 中断的程序其返回地址在链接寄存器 R14_irq 中保持不变。而 IRQ 例程的返回地址保存在 R14_fiq 中。因此，使用后备寄存器可以避免覆盖已保存的返回地址，并且当出现异常程序嵌套时，不需要将这些地址放到栈中。但是，如果在同一处理器模式下处理不同的异常的话，就需要将它们的返回地址进行保存（如果允许嵌套的话）。

## D.8 输入 / 输出

ARM 体系结构采用的是 3.1 节中描述过的存储器映射 I/O 方法。正如在 3.1.2 节中介绍的，从键盘读取一个字符或向显示器发送一个字符都可以通过程序控制 I/O 来完成，也可以通过 3.2 节中描述的中断驱动 I/O 来完成。本节将通过介绍一些程序示例来说明这两种方法，这些程序示例显示了第 3 章中的一般程序（涉及键盘和显示设备）是如何用 ARM 汇编语言实现的。

### D.8.1 程序控制 I/O

我们首先给出一个从键盘读取字符和将字符写到显示器的短指令序列。

**1. 键盘字符输入**

假设键盘接口中的数据、状态和控制寄存器的布局如图 3-3a 所示。此外，假设地址 KBD_DATA（0x4000）已经装入寄存器 R1 中。指令序列

```
READWAIT    LDRB    R3, [R1, #4]
            TST     R3, #2
            BEQ     READWAIT
            LDRB    R3, [R1]
```

当键盘上有一个键被按下时将该字符读入到寄存器 R3 中。测试（TST）指令对它的两个操作数进行按位逻辑与操作，并根据结果来设置条件码标志。立即数 2 在 $b_1$ 位上是 1。因此，TST 操作的结果直到 KIN = 1 之前都将是 0，KIN=1 意味着在 KBD_DATA 中有一个字符存在。如果 KIN = 0，则 BEQ 指令转移返回到 READWAIT。这将一直循环直到有按键被按下将 KIN 设置为 1 时为止。然后，不再发生转移，而是将该字符装入寄存器 R3 中。 [646]

**2. 显示器字符输出**

假设地址 DISP_DATA 已经被装入寄存器 R2 中，指令序列

```
WRITEWAIT   LDRB    R4, [R2, #4]
            TST     R4, #4
            BEQ     WRITEWAIT
            STRB    R3, [R2]
```

当显示器准备接受字符时将寄存器 R3 中的字符发送到 DISP_DATA 寄存器中。

3. 完整的输入 / 输出程序

刚刚描述的两个程序可以从键盘读取一行字符，将它们存储到存储器中，并将它们显示在显示器上。如图 D-17 中的程序所示。该程序模仿了图 3-4 中的一般程序。假设寄存器 R0 中保存了将要存储该行字符的存储区域的第一个字节的地址，寄存器 R1 到 R4 与刚刚描述的 READWAIT 和 WRITEWAIT 循环中的用法相同。第一条 Store 指令（STRB）将从键盘读入的字符存储到存储器中。该指令中采用带写回的后变址寻址方式来遍历存储区。测试相等的指令（TEQ）用来检测两个操作数是否相等，并相应地设置 Z 条件码标志。

```
READ    LDRB    R3, [R1, #4]    载入 KBD_STATUS 字节
        TST     R3, #2          并等待字符
        BEQ     READ
        LDRB    R3, [R1]        读取字符并将其保存在存
        STRB    R3, [R0], #1    储器中
ECHO    LDRB    R4, [R2, #4]    载入 DISP_STATUS 字节
        TST     R4, #4          并等待显示器
        BEQ     ECHO              准备就绪
        STRB    R3, [R2]        向显示器发送字符
        TEQ     R3, #CR           如果不是回车符，则继
        BNE     READ            续读取字符
```

图 D-17 读取一行字符并进行显示的程序

[647]

## D.8.2 中断驱动 I/O

D.7 节中描述的 ARM 中断设施可以在中断驱动的控制下读取键盘上输入的一行字符。我们假设键盘有其自己的中断请求线连接到处理器的 IRQ 中断输入端。

可能会有很多设备可在 IRQ 线上发出中断。如果是这样的话，可在 IRQ 中断服务程序中按某种优先级顺序对这些设备进行软件轮询，以识别出第一个发出中断的设备。为简单起见，我们将假设键盘是唯一一个可在 IRQ 线上发出中断请求的设备。

图 3-8 给出了一个一般程序，该程序使用中断方式来读取一行字符直到遇到一个回车符（CR）。在从键盘读取这些字符的时候，中断服务程序使用程序控制 I/O 方式将它们发送给显示器。我们将在 ARM 处理器上实现该任务。在 IRQ 处理器模式 SPSR_irq 下，状态寄存器 CPSR 和备份状态寄存器是与中断处理相关的处理器控制寄存器。

该 I/O 程序示例中，需要使用以下这些存储单元：

- PNTR 是一个指针单元，其中包含从键盘读取的下一个字符将要装入的存储器地址。
- LINE 是该行第一个字符将要放置的存储器字节单元。
- EOL 是一个包含一个二进制变量的存储单元，该变量向主程序说明已经读取了一个完整的行。

图 D-18 给出了一个 ARM 的 IRQ 中断服务程序和一个主程序，它们分别对应于图 3-8 中的中断服务程序和主程序。假设主程序在管态模式下运行。前六条指令将 PNTR 初始化为地址 LINE，清除 EOL 标志，并在键盘控制寄存器 KBD_CONT 中允许中断。最后一条指令通过使用 MSR 指令将十六进制值 50 装入 CPSR 中，从而清除 IRQ 禁止位（I）并使处理器切换到用户模式。

该 IRQ 中断服务程序与图 3-8 中的一般程序非常相似。为简单起见，许多 Load 和 Store 指令使用相对寻址方式，假设所列出的存储单元和设备寄存器都在相对于程序计数器的偏移量

可达到的范围之内。

```
中断服务程序
IRQLOC  STMFD   R13!, {R2, R3}      将 R2 和 R3 的内容保存到栈中
        LDR     R2, PNTR            载入地址指针
        LDRB    R3, KBD_DATA        从键盘读取字符
        STRB    R3, [R2], #1        将字符写入存储器并递增指针

        STR     R2, PNTR            更新存储器中的指针
ECHO    LDRB    R2, DISP_STATUS     等待显示器准备就绪
        TST     R2, #4
        BEQ     ECHO
        STRB    R3, DISP_DATA       向显示器发送字符
        CMP     R3, #CR             检查字符是否为回车符
        BNE     RTRN                如果不是回车符（CR），则返回
        MOV     R2, #1              如果是回车符（CR），则指示该行结束
        STR     R2, EOL
        MOV     R2, #0              禁止键
        STRB    R2, KBD_CONT        盘中断
RTRN    LDMFD   R13!, {R2, R3}      恢复寄存器，并从中断返回
        SUBS    R15, R14, #4

主程序

        LDR     R2, =LINE           初始化缓冲区指针
        STR     R2, PNTR
        MOV     R2, #0              清除行结束指示变量
        STR     R2, EOL
        MOV     R2, #2              允许键盘中断
        STRB    R2, KBD_CONT
        MOV     R2, #&50            允许 IRQ 中断，并切换到用户模式
        MSR     CPSR, R2
        下一条指令
```

图 D-18　一个使用中断方式从键盘读取一行字符并使用轮询方式显示该行字符的程序

## D.9　指令的条件执行

所有的 ARM 指令都可有条件地执行，这允许编写更短的程序以替换为传统的 RISC 机器编写的程序（其中包含很多转移指令）。

考虑下面的例子。图 D-19a 给出了一个程序的循环部分，该循环使用 RISC 风格的指令来找出两个非零正整数的最大公约数（GCD）[4]。当进入该程序时，这两个数分别包含在寄存器 R2 和 R3 中。在每遍循环的开始处，如果这两个数不相等，那么该程序将从较大的数中减去较小的数，并返回到循环的开始。如果这两个数相等，则从循环中退出，跳到位置 NEXT 处，GCD 就包含在这两个寄存器中。

```
LOOP    Branch_if_[R2]=[R3]     NEXT
        Branch_if_[R2]>[R3]     REDUCE
        Subtract                R3, R3, R2
        Branch                  LOOP
REDUCE  Subtract                R2, R2, R3
        Branch                  LOOP
NEXT    下一条指令
```

a）使用 RISC 风格指令的 GCD 算法

```
LOOP    CMP     R2, R3
        SUBGT   R2, R2, R3
        SUBLT   R3, R3, R2
        BNE     LOOP
NEXT    下一条指令
```

b）使用 ARM 指令的 GCD 算法

图 D-19　指令的条件执行

该任务的 ARM 程序如图 D-19b 所示。Compare 指令用来设置条件码。每条 Subtract 指令中的操作码后缀都指定了执行 Subtract 指令的条件。仅当寄存器 R2 的内容大于寄存器 R3 的内容时，才执行第一条 Subtract 指令；只有寄存器 R3 的内容大于寄存

器 R2 的内容时，才执行第二条 Subtract 指令。在每次循环中，当这两个寄存器的内容不相等时，只执行其中一条 Subtract 指令。当这两个寄存器的内容相等时（这可能是初始情况），这两条 Subtract 指令都不会被执行，且不会发生转移跳回到 LOOP 处。

当传统代码中 Branch 指令的密度相对较高时，上述情况所产生的较短的 ARM 代码序列是最有效的。节省代码空间在一些小型的嵌入式系统应用中是非常重要的。

## D.10 协处理器

被称为协处理器（coprocessor）的硬件单元可以连接到 ARM 处理器上。它们可用来执行基本 ARM 指令集中所没有包含的操作。其中一个例子是可以对浮点数进行算术运算的硬件单元。其他的例子包括可对数字信号或视频数据进行特定处理的硬件单元。使用包含如下三种类型的 ARM 指令扩展集的协处理器编写程序将会更加方便：

648 ↑ 650

- 协处理器中的数据操作
- ARM 和协处理器寄存器之间的传输
- 存储器和协处理器寄存器之间的 Load 和 Store 传输

一种用来综合硬件实现的定义协处理器单元的软件可与定义基本 ARM 处理器的软件结合起来，以便将协处理器单元和 ARM 处理器集成为一个芯片。

## D.11 嵌入式应用和 Thumb ISA

低成本、低功耗的嵌入式系统，如移动电话，是 ARM 处理器的主要应用领域。这种系统的设计者努力减少存储需要的程序所需的片上存储空间的大小。在 D.9 节中我们可以看到，指令的条件执行可减少代码空间。用来在多个寄存器和一个存储器字块之间传输字的块传输指令，也可以减少代码空间。

使用最常用的 ARM 指令的子集可进一步减少代码空间，该子集中的每一条指令只使用 16 位来表示。这个子集被称为 Thumb 指令集。Thumb 指令的执行如下：首先，从存储器中提取 Thumb 指令。然后，将这些 Thumb 指令从 16 位的编码格式解压缩（扩展）为相应的 32 位 ARM 指令，并以通常的方式执行它们。

状态寄存器（CPSR）中的 $b_5$ 位，标记为 T，用来确定到达的指令流中是否包含 Thumb 指令（T = 1）或者标准的 32 位 ARM 指令（T = 0）。一个程序中可以既包含 Thumb 指令例程，也包含标准指令例程。当在这两种指令集之间进行切换时，需要特殊的指令来管理 T 位。

Thumb 指令和标准的指令之间有两个主要的区别。首先，许多 Thumb 指令使用双操作数格式，在这种格式中目的寄存器也是其中一个源操作数寄存器。第二，条件执行（适用于所有标准的 ARM 指令）主要用于 Thumb 指令集中的转移指令。这些差异可节省指令编码的位空间。

## D.12 结束语

ARM 处理器在嵌入式系统市场中已经取得了显著的商业成功。其设计被授权给一些制造手持通信设备的公司。ARM 指令集的 16 位 Thumb 版本特别适合于低成本、低功耗的应用，因为它允许编写简洁型的程序。灵活的变址寻址方式和 32 位 ISA 中的块传输指令对于许多应用都是非常有用的。虽然这两个特点反映了 CISC 风格的特性，但 ARM 通常被认为是具有 RISC 风格的 Load/Store 体系结构。

651

## D.13 问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。

**例 D.1**

**问题**：假设有一个 ASCII 编码字符的字节串保存在存储器中，起始地址为 STRING。该字符串以回车符（CR）结束。写一个 ARM 程序来确定该字符串的长度，并将所得到的长度存储在单元 LENGTH 中。

**解答**：图 D-20 给出了一个可能的程序。将字符串中的字符与 CR（ASCII 码为 &0D）进行比较，计数器递增，直至到达字符串的末尾。

```
        LDR   R2, =STRING    加载字符串的起始地址
        MOV   R3, #0         将字符串的长度加载为 0
        MOV   R4, #&0D       加载回车符的 ASCII 码
LOOP    LDRB  R5, [R2], #1   加载下一个字符
        CMP   R4, R5         检查是否为回车符尾结束。
        BEQ   DONE           或者递增长度计数并返回
        ADD   R3, R3, #1
        B     LOOP
DONE    STR   R3, LENGTH     保存字符串的长度
```

图 D-20 例 D.1 的程序

**例 D.2**

**问题**：写一个 ARM 程序，在一个 32 位的非负整数列表中找出最小的数。程序数据区中连续的存储单元 SMALL 和 N 分别用来存储最小的数和列表的大小。跟在这两个单元后面的是该列表，其第一个数存储在单元 ENTRIES 中。在程序中包含按照规定组织程序和数据区所需要的汇编指示（assembler directive）。用一个包含 7 个整数的小列表作为例子。

**解答**：程序指令和数据如图 D-21 所示。程序中包含的注释解释了该程序是如何完成所需任务的。注意，将地址 ENTRIES 装入寄存器 R2 的方法就是汇编程序在之前的程序示例中所使用的替换伪指令的方法，在 D.5.1 节中进行了介绍。

652

```
        AREA    CODE
        ENTRY
        LDR     R2, POINTER     R2 指向 ENTRIES 处的列表
        LDR     R3, [R2, #-4]   将计数器 R3 初始化为 n
        LDR     R5, [R2]        R5 保存目前为止的最小数
LOOP    SUBS    R3, R3, #1      递减计数器
        BEQ     DONE            如果 R3 包含了 0，则完成
        LDR     R6, [R2, #4]!   递增列表指针并获取下一个数

        CMP     R5, R6          检查是否可以找到更小的数
        BLE     LOOP            如果没有找到更小的数，则转移回到前面
        MOV     R5, R6          否则，将找到的更小的数移到 R5 中
        B       LOOP            然后转移回到前面
DONE    STR     R5, SMALL       将最小的数存储到 SMALL 中

        AREA    DATA
POINTER DCD     ENTRIES         指向列表开始的指针
SMALL   DCD     0               存储最小数的单元
N       DCD     7               列表中的项数
ENTRIES DCD     4, 5, 3, 6, 1, 8, 2   数字列表
        END
```

图 D-21 例 D.2 的程序

**例 D.3**

**问题**：写一个 ARM 程序，将一个 $n$ 位十进制整数转换成二进制数。十进制数以 $n$ 位 ASCII 编码字符的形式给出，并将其存放在存储器的连续字节单元中，起始地址为 DECIMAL。转换后的数存储在单元 BINARY 中。单元 N 中包含值 $n$。

**解答**：考虑一个 4 位的十进制数 $D = d_3d_2d_1d_0$。其值可由表达式 $((d_3 \times 10 + d_2) \times 10 + d_1) \times 10 + d_0$ 给出。该表达式是图 D-22 中程序所使用的转换技术的基础。每个 ASCII 编码字符先被转换成一个二进

制编码的十进制（BCD）数字，然后再对于计算。这里假设转换后的二进制值可以用不超过 32 位来表示。

```
        LDR     R2, N           将计数器 R2 初始化为 n
        LDR     R3, =DECIMAL    R3 指向 ASCII 数字
        MOV     R4, #0          R4 用来保存二进制数
        MOV     R6, #10         R6 用来保存常量 10
LOOP    LDRB    R5, [R3], #1    获取下一个 ASCII 字符并递增指针

        AND     R5, R5, #&0F    产生 BCD 数字
        ADD     R4, R4, R5      加到中间结果中
        SUBS    R2, R2, #1      递减计数器
        BEQ     DONE            如果完成，则保存结果
        MUL     R4, R6, R4      将中间结果乘以 10
        B       LOOP            循环回到前面
DONE    STR     R4, BINARY      将结果存储到 BINARY 中
```

图 D-22 例 D.3 的程序

**例 D.4**

**问题**：考虑一个数字阵列 A(*i*, *j*)，其中 *i* = 0 到 *n*–1，是行索引，*j* = 0 到 *m*–1，是列索引。该阵列按行存储在存储器中，每行占用 *m* 个连续的字单元。写一个 ARM 子程序，将第 *x* 列的元素逐一加到第 *y* 列的元素上，并将和元素（sum element）存放在第 *y* 列上。索引值 *x* 和 *y* 通过寄存器 R2 和 R3 传递给子程序。参数 *n* 和 *m* 通过寄存器 R4 和 R5 传递给子程序。元素 A(0,0) 的地址通过寄存器 R6 传递给子程序。

**解答**：图 D-23 给出了实现该任务的一个可能的主程序和子程序。我们假定值 *x*、*y*、*n* 和 *m* 分别存储在存储单元 X、Y、N 和 M 中。元素 A(0,0) 的地址是 ARRAY。程序中的注释解释了该任务是如何完成的。有趣的是，将 ARM 子程序中的指令数与图 2-36 中 RISC 风格子程序的指令数进行比较，可以看出 ARM 子程序更短。这是由于 ARM 子程序使用了块传输指令以及灵活的变址寻址方式。

```
主程序
        LDR     R2, X           载入值 x
        LDR     R3, Y           载入值 y
        LDR     R4, N           载入值 n
        LDR     R5, M           载入值 m
        LDR     R6, =ARRAY      载入元素 A(0,0) 的地址 ARRAY

        BL      SUB             调用子程序

子程序
SUB     STMFD   R13!, {R10, R11, R14}   将寄存器 R10、R11 和链接寄存器（R14）
                                        的值保存到堆栈中
        ADD     R2, R6, R2, LSL #2      将 A(0, x) 的地址加载到 R2 中
        ADD     R3, R6, R3, LSL #2      将 A(0, y) 的地址加载到 R3 中
LOOP    LDR     R10, [R2], R5, LSL #2   将第 x 列的值加载到 R10 中，并递增到地址

        LDR     R11, [R3]               将第 y 列的值加载到 R11 中
        ADD     R11, R11, R10           将两列的值相加
        STR     R11, [R3], R5, LSL #2   将和存储到第 y 列并递增到地址

        SUBS    R4, R4, #1              递减行计数器，如果未完成则循环回到前面
        BGT     LOOP
        LDMFD   R13!, {R10, R11, R15}   恢复寄存器 R10、R11 和程序计数器（R15）
                                        的值，并从子程序返回
```

图 D-23 例 D.4 的程序

**例 D.5**

**问题：** 假设存储单元 BINARY 中包含一个 32 位的模式。我们希望在一个具有图 3-3 所示接口的显示设备上将这些位显示为 8 个十六进制数字字符。写一个 ARM 程序完成这个任务，使用程序控制 I/O 来显示字符。

**解答：** 图 D-24 给出了一个可能的程序。首先，通过查表法在一个 16 项的表中进行查找，将十六进制数字转换为 ASCII 字符。这 8 个将要显示的 ASCII 字符存储在一个存储器字节块中，起始地址为 HEX。然后，将字符发送给显示器。注释描述了程序中所采取的具体操作。注意 LOOP 处的 ORR 指令，它对寄存器 R2 的内容进行循环右移操作。每次循环移位会将下一次要转换的 4 位十六进制数字移到 R2 的低 4 位上。还要注意的是，在将地址值装入寄存器中时，使用了 ADR 伪指令。ADR 指令在 D.5.1 节中进行了介绍。

653
|
655

```
                AREA    CODE
                ENTRY
                MOV     R0, #0          ORR 指令所需的操作
                LDR     R2, BINARY      加载二进制模式
                ADR     R3, TABLE       R3 指向 ASCII 表
                ADR     R4, HEX         R4 指向十六进制字符
                MOV     R5, #8          加载数字计数
LOOP            ORR     R2, R0, R2, ROR #28   将下一个数字循环移位到低 4 位上
                AND     R6, R2, #&F     提取数字，并将其装入 R6 中
                LDRB    R7, [R3, +R6]   加载数字的 ASCII 码
                STRB    R7, [R4], #1    将数字保存到字符串中，并递增指针

                SUBS    R5, R5, #1      递减数字计数器
                BGT     LOOP            如果未完成，则循环回到前面
DISPLAY         MOV     R5, #8          为显示例程加载数字计数
                ADR     R4, HEX         R4 指向十六进制字符
                ADR     R2, DISP_DATA   R2 指向设备寄存器
SENDCHAR        LDRB    R3, [R2, #4]    通过测试 DOUT 标志来检查显示器
                TST     R3, #4          是否准备就绪
                BEQ     SENDCHAR
                LDRB    R6, [R4], #1    获取下一个 ASCII 字符，并递增指针

                STRB    R6, [R2]        将字符发送给显示器
                SUBS    R5, R5, #1      递减数字计数器
                BGT     SENDCHAR        循环，直到所有的字符都显示完毕
                下一条指令

                AREA    DATA
BINARY          DCD     &A123B456       二进制模式
HEX             SPACE   8               ASCII 编码数字的存储空间
TABLE           DCB     &30,&31,&32,&33 ASCII 码转换表
                DCB     &34,&35,&36,&37
                DCB     &38,&39,&41,&42
                DCB     &43,&44,&45,&46
```

图 D-24 例 D.5 的程序

656

## 习题

[E] D.1 假设在一台 ARM 计算机中，寄存器 R0、R1、R2、R6 和 R7 中的值分别为 1000、2000、1016、20 和 30。数字 1、2、3、4、5 和 6 存放在存储器中起始地址为 1000 的连续字单元中。每次都以给定的初始值开始，请问执行以下两个指令块的效果各是什么？

(a) LDR R8, [R0]
LDR R9, [R0, #4]
ADD R10, R8, R9

(b) STR R6, [R1, #-4]!
STR R7, [R1, #-4]!
LDR R8, [R1], #4
LDR R9, [R1], #4
SUB R10, R8, R9

[M] D.2 下面哪些 ARM 指令会使得汇编程序发出语法错误消息？为什么？

(a) ADD R2, R2, R2
(b) SUB R0, R1, [R2, #4]
(c) MOV R0, #2_1010101
(d) MOV R0, #257
(e) ADD R0, R1, R11, LSL #8

[M] D.3 写一个 ARM 程序，将寄存器 R2 中各个位的顺序颠倒过来。例如，如果 R2 中的初始值是 1110…0100，则 R2 中的最后结果应该是 0010…0111。（提示：使用移位和循环移位操作。）

[M] D.4 考虑图 D-7 中的程序，列出 BGT 指令的第三次执行中每一次执行之后寄存器 R0、R1 和 R2 中的内容。将结果在一个表中进行显示，该表以三个寄存器作为列标题。用三行列出每次执行 BGT 指令之后这三个寄存器中的内容。图 D-13 给出了该程序的数据。

[M] D.5 写一个 ARM 程序，比较两个字节列表中的相应字节，并把较大的字节存放在第三个列表中。这两个列表分别从字节单元 X 和 Y 开始，而较大字节的列表从 LARGER 开始。列表的长度存放在存储单元 N 中。

[M] D.6 写一个 ARM 程序，产生斐波纳契数列的前 $n$ 个数。在此数列中，前两个数是 0 和 1，随后的每一个数都是通过将前面的两个数相加产生的。例如，当 $n = 8$ 时，该数列为 0，1，1，2，3，5，8，13。你的程序需要将数列中的数存放在存储器中起始地址为 MEMLOC 的连续字单元中。假设值 $n$ 存储在单元 N 中。

[M] D.7 写一个 ARM 程序，将一个文本字（a word of text）从小写转化为大写。这个字由一些 ASCII 字符组成，这些字符存放在存储器中的连续字节单元中，起始地址为 WORD，并以一个空格字符结束。（ASCII 码参见第 1 章中的表 1-1。）

657

[M] D.8 将图 2-10 所示的学生成绩列表修改为每个学生包含有 $j$ 项测验分数。假设有 $n$ 个学生。写一个 ARM 程序来计算每项测验的分数和，并将这些和存储在存储器中地址为 SUM，SUM + 4，SUM + 8，… 的字单元中。由于测验的项数 $j$ 比处理器中的寄存器个数要大，所以图 D-8 所示的用于 3 项测验的程序已不能使用。使用两个嵌套循环，内部循环累加每项特定测验的和，而外部循环遍历测验的数目 $j$。假设 $j$ 存放在存储单元 J 中，该单元在图 2-10 中的单元 N 之前。

[E] D.9 写一个 ARM 程序，计算表达式 SUM = 580 + 68400 + 80000。

[E] D.10 写一个 ARM 程序，计算表达式 ANSWER = A × B + C × D。

[M] D.11 写一个 ARM 程序，在一个含有 $n$ 个 32 位整数的列表中找出所包含的负整数的个数，并将该计数保存在单元 NEGNUM 中，$n$ 保存在存储单元 N 中，列表中的第一个整数保存在单元 NUMBERS 中。在程序中包含必要的汇编指示（assembler directive）和一个样本列表，列表中含有 6 个数字，其中一些是负数。

[M] D.12 为第 2 章中例 2.5 所描述的字节排序程序写一个 ARM 程序。

[M] D.13 写一个 ARM 程序来解决第 2 章的习题 2.22 中的问题。

[M] D.14 写一个 ARM 程序来解决第 2 章的习题 2.24 中的问题。

[M] D.15 写一个 ARM 程序来解决第 2 章的习题 2.25 中的问题。

[M] D.16 写一个 ARM 程序来解决第 2 章的习题 2.26 中的问题。

[M] D.17 写一个 ARM 程序来解决第 2 章的习题 2.27 中的问题。

[M] D.18 写一个 ARM 程序来解决第 2 章的习题 2.28 中的问题。

[M] D.19 写一个 ARM 程序来解决第 2 章的习题 2.29 中的问题。

[M] D.20 写一个 ARM 程序来解决第 2 章的习题 2.30 中的问题。

[M] D.21 写一个 ARM 程序来解决第 2 章的习题 2.31 中的问题。

[D] D.22 写一个 ARM 程序来解决第 2 章的习题 2.32 中的问题。

[D] D.23 写一个 ARM 程序来解决第 2 章的习题 2.33 中的问题。

[M] D.24 写一个 ARM 程序，从键盘读取 $n$ 个字符，读取这些字符的同时将其压入用户堆栈中，然后再将它们回显到显示器上。寄存器 R6 用作堆栈指针。计数值 $n$ 保存到存储器的字单元 N 中。

[M] D.25 假设图 D-17 所示的程序中，获取和执行一条指令所需的平均时间是 5 纳秒。如果通过键盘输入字符的速度为每秒 10 个字符，那么每输入一个字符大约要执行多少次 BEQ READ 指令？假设显示每个字符所需的时间远小于在键盘上连续输入两个字符之间的时间。 658

[M] D.26 将图 D-17 中的程序改写为一个主程序调用一个名为 GETCHAR 的子程序的形式，该子程序读取一个字符并调用另一个名为 PUTCHAR 的子程序来显示这个字符。地址 KBD_STATUS 通过寄存器 R1 传递给 GETCHAR，而主程序则希望通过寄存器 R3 来获取传递回的字符。地址 DISP_STATUS 和将要显示的字符分别通过寄存器 R2 和 R3 传递给 PUTCHAR。任一子程序所使用的任何其他寄存器都必须由子程序使用堆栈（其指针在寄存器 R13 中）来保存和恢复。将字符保存到存储器中和检查行结束字符 CR 的工作都在主程序中完成。

[M] D.27 使用堆栈来传递参数，重复习题 D.26。

[M] D.28 写一个 ARM 程序，从键盘接受三个十进制数字。每个数字用 ASCII 码（参见第 1 章中的表 1-1）表示。假设这三个数字表示 0 到 999 范围内的一个十进制整数。将该整数转换为二进制数表示形式。先接收高位数字。为了方便该转换，在存储器中保存了两个表，每个表有 10 项。第一个表从字单元 TENS 开始，其中包含十进制值 0、10、20、…、90 的二进制表示。第二个表从字单元 HUNDREDS 开始，其中包含用二进制表示的十进制值 0、100、200、…、900。

[M] D.29 使用两个独立子程序实现习题 D.28 中十进制到二进制的转换程序。调用第一个子程序的主程序通过将两个参数压栈（堆栈指针寄存器是 R13）来传递参数。第一个参数是一个 3 字节存储器缓冲区的地址，该缓冲区用来保存所输入的十进制数字字符。第二个参数是用来保存转换后二进制值的单元地址。第一个子程序从键盘读取三个字符，然后调用第二个子程序来执行转换。通过处理器寄存器将所需要的参数传递给第二个子程序。这两个子程序都必须保存它们在堆栈中使用的任一寄存器的内容。

（a）为 ARM 处理器编写这两个子程序。

（b）给出调用第二个子程序的指令执行后堆栈中的内容。

[M] D.30 写一个 ARM 程序，在视频显示器的一行上以十六进制形式显示主存中 10 个字节的内容。该字节串在存储器中的起始位置是 LOC。每个字节将显示为两个十六进制字符。连续字节应以空格分隔。

[M] D.31 假设存储单元 BINARY 中包含一个 16 位的模式。我们希望在一个具有图 3-3 所示接口的显示设备上将这些位显示为 0 和 1 组成的字符串。写一个 ARM 程序完成这个任务。

[M] D.32 使用图 3-17 中的七段显示器和图 3-14 中的定时器电路，写一个 ARM 程序，显示重复序列 0、1、2、…、9、0、…中的十进制数字，每个数字显示一秒钟。假设定时器电路中的计数器是由 100 MHz 的时钟驱动的。 659

[D] D.33 使用两个图 3-17 所示的七段显示器和图 3-14 所示的定时器电路。写一个 ARM 程序，显示重复序列 0、1、2、…、98、99、0、…中的数。每个数显示一秒钟。假设定时器电路中的计数器是由 100 MHz 的时钟驱动的。

[D] D.34 写一个 ARM 程序，计算实时时钟时间并以小时（0～23）和分钟（0～59）的形式显示时间。显示器包括 4 个图 3-17 所示的七段显示设备。还有一个具有图 3-14 所示接口的定时器电路，其计数器是由 100 MHz 的时钟驱动的。

[M] D.35 写一个 ARM 程序来解决第 3 章中例 3.5 所描述的问题。

[M] D.36 写一个 ARM 程序来解决第 3 章中例 3.6 所描述的问题。
[M] D.37 写一个 ARM 程序来解决第 3 章的习题 3.19 中的问题。
[M] D.38 写一个 ARM 程序来解决第 3 章的习题 3.21 中的问题。
[M] D.39 写一个 ARM 程序来解决第 3 章的习题 3.23 中的问题。
[M] D.40 写一个 ARM 程序来解决第 3 章的习题 3.25 中的问题。

## 参考文献

1. ARM Limited, *ARM7TDMI Technical Reference Manual—revision r4p1*, Document number ARM DDI 0210C, November 2004. Available at http://www.arm.com.
2. Steve Furber, *ARM System-on-chip Architecture*, 2nd Ed., AddisonWesley, Harlow, England, 2000.
3. William Hohl, *ARM Assembly Language: Fundamentals and Techniques*, CRC Press, 2009.
660 4. ARM Limited, *The ARM Instruction Set—V1.0*, ARM University Program.

# Intel IA-32 体系结构

## 附录目标

在本附录中，你将学习 Intel IA-32 体系结构的特点：

- 存储器组织与寄存器结构
- 寻址方式与指令类型
- 输入/输出能力
- 标量浮点运算
- 多媒体操作
- 向量浮点运算



Intel 公司使用通用名称 Intel 体系结构（IA）来命名其处理器产品系列的指令集。我们将描述处理器的 IA-32 指令集，这种处理器使用 32 位的存储器地址，能处理 32 位的操作数。IA-32 指令集是非常庞大的，除了提供典型的整数和浮点数指令，它还包括多媒体应用和向量数据处理的专用指令。我们将只关注基本的指令和寻址方式。参考文献 [1] 全面概述了 IA-32 体系结构，Intel 的网站（http://www.intel.com）则提供了一些更详细的技术文档。

## E.1 存储器组织

在 IA-32 体系结构中，存储器采用字节可寻址的 32 位地址。指令的操作数通常是 8 位或 32 位的。在 Intel 的术语中，这些操作数的大小被称为字节（byte）或双字（doubleword）。16 位的操作数在早期的 16 位 Intel 处理器中被称为字（word）。对于双精度浮点数和打包的整数数据，还有更大的 64 位的操作数大小，被称为四字（quadword）。这里使用的是 2.1.2 节中描述的小端编址方式。多字节的数据操作数可以在任意的字节地址处开始，不需要在存储器中特定的地址边界上对齐。

## E.2 寄存器结构

图 E-1 所示的是处理器寄存器。有 8 个 32 位的通用寄存器，可以用来保存整数数据或寻址信息。我们并不对这些寄存器进行连续编号，而是通过唯一的名称来标识它们，这将在本节后面进行介绍。另外，有 8 个额外的寄存器可用于浮点指令，这将在 E.9 节中进行讨论。这些寄存器还可以用于 E.10 节描述的多媒体指令。还有另外一组寄存器，在图 E-1 中尚未列出，这些寄存器可被 E.11 节中讨论的向量处理指令使用。

IA-32 体系结构具有不同的存储访问模型。段式存储模型（segmented memory model）将称为段（segment）的不同存储区域与不同的用途联系在一起。代码段（code segment）保存程序的指令，堆栈段（stack segment）包含着处理器堆栈，两个数据段（data segment）用来保存数据操作数。图 E-1 所示的六个段寄存器保存着用于在存储器地址空间中确定这些段起始位置的选择器值。这些寄存器的具体功能在本附录中不作详细讨论。本附录将采用 IA-32 体系结构的平展存储模型（flat memory model），在该模型中，一个 32 位的地址可以访问代码、处理器堆栈或数据区域中的任何一个存储单元。在这种情况下，段寄存器被初始化为指向存储器中地址 0 的选择器值。

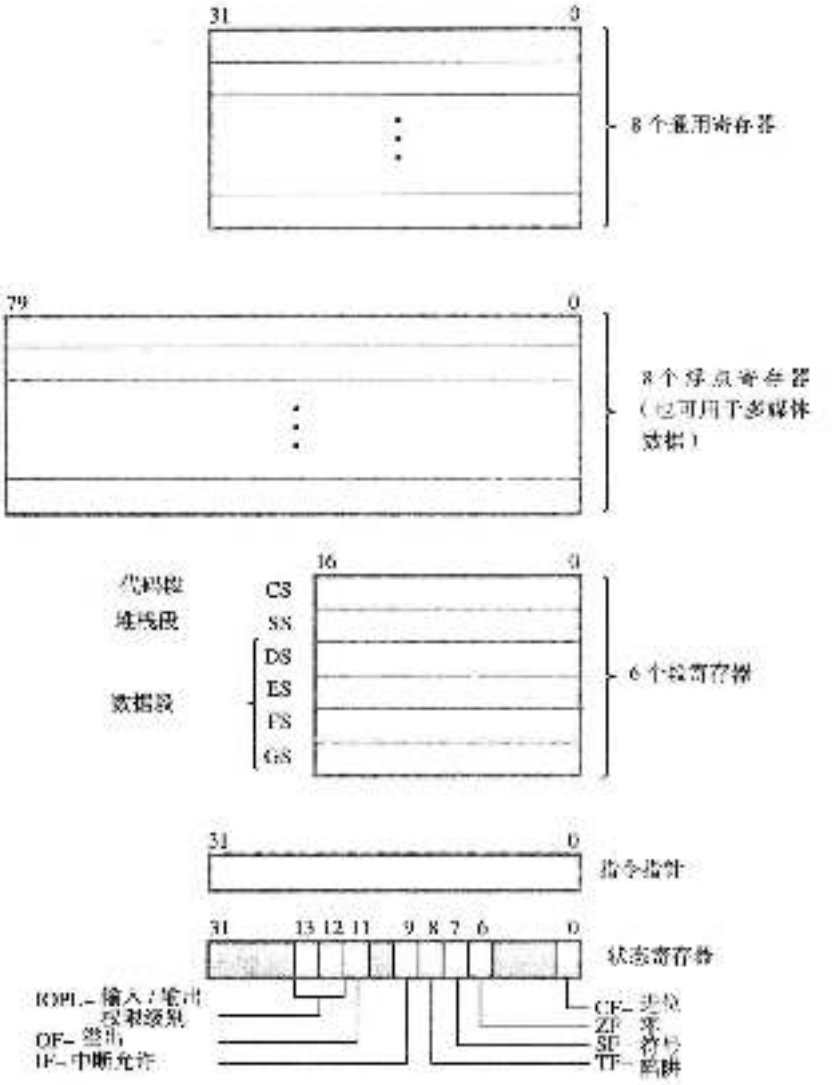



图 E-1 IA-32 寄存器结构

图 E-1 底部所示的两个寄存器是指令指针和状态寄存器，其中指令指针作为程序计数器，保存着下一条将要执行的指令地址，而状态寄存器保存着条件码标志（CF、ZF、SF、OF），这些标志中包含有关算术运算的结果信息。程序执行模式位（IOPL、IF、TF）与输入/输出操作和中断相关。

662
~
663

IA-32 的通用寄存器与早期的 8 位和 16 位 Intel 处理器寄存器保持了兼容性。在那些处理器中，某些寄存器的使用有一些限制。图 E-2 给出了 IA-32 寄存器与早期处理器中寄存器之间的关系。8 个通用寄存器被分为三种不同的类型：数据寄存器用来保存操作数，指针寄存器用来保存地址，变址寄存器用来保存变址地址。指针和变址寄存器用来确定存储器操作数的有效地址。

在 Intel 最初的 8 位处理器中，数据寄存器被称为 A、B、C 和 D。在后来的 16 位处理器中它们被标志为 AX、BX、CX 和 DX。每个寄存器中的高位字节和低位字节分别用后缀 H 和 L 来标识。例如，寄存器 AX 中的两个字节是 AH 和 AL。在 IA-32 处理器中，前缀 E 用来标

识相应的可扩展 32 位寄存器：EAX、EBX、ECX 和 EDX。前缀标志 E 还可以用于图 E-2 所示的其他 32 位寄存器。它们是早期处理器中相应的 16 位寄存器的扩展版本。

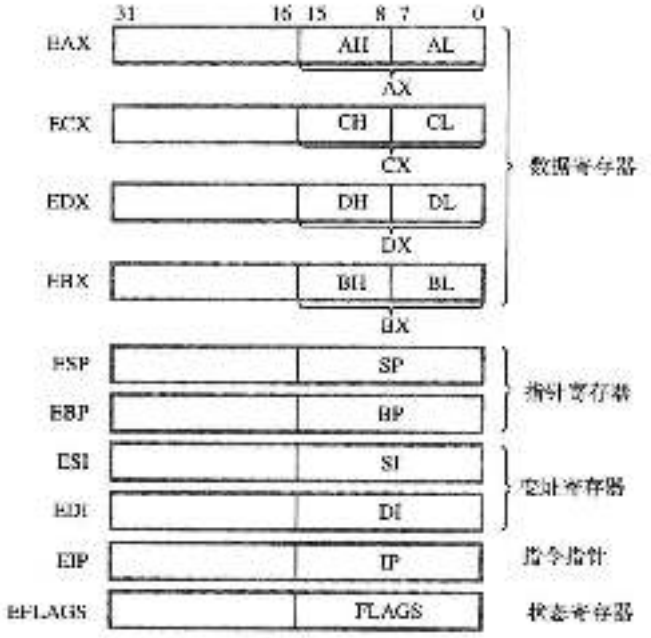



图 E-2 IA-32 寄存器结构与早期 Intel 处理器寄存器结构的兼容性

这种寄存器标志在 Intel 的技术文档 [1] 以及 Intel 处理器的其他描述中使用。由于保持历史标志的原因，Intel 在它的处理器系列中保持了向上兼容的特点。也就是说，为早期的 16 位处理器编写的机器语言程序，只要 IA-32 处理器的状态设置正确，就可以不加修改地在目前的 IA-32 处理器上正确运行。我们将在给定的汇编语言程序示例中，使用带有 E 前缀的寄存器标志，因为 IA-32 处理器汇编语言当前版本中使用的就是这些助记符。当字节操作数保存在相应的 32 位寄存器中的低八位时，将使用 AL、BL 等标志。 664

## E.3 寻址方式

IA-32 体系结构有一套丰富且灵活的寻址方式。它们既可以访问单个数据项，也可以访问从一个指定的存储器地址开始的有序列表数据项。基本的寻址方式与大多数处理器的寻址方式相同，如 2.4 节所述。它们是：立即方式、绝对方式、寄存器方式和寄存器间接方式。Intel 使用直接（direct）方式来表示绝对方式，所以在这里我们也采用同样的方式。还有几种用来访问存储器中数据操作数的更加灵活的寻址方式。在 2.4 节中描述的最灵活的方式就是变址方式，这种方式的通用记号为 X(*Ri*, *Rj*)。操作数的有效地址 EA 计算如下：

$$EA = [Ri] + [Rj] + X$$

其中 *Ri* 和 *Rj* 是通用寄存器，X 是一个常量。寄存器 *Ri* 和 *Rj* 分别被称为基址（base）和变址（index）寄存器，常量 X 被称为*位移量*（displacement）。IA-32 寻址方式中包括这种方式和该种方式的更简单变种形式。

IA-32 的寻址方式定义如下：

*立即方式*（immediate mode）—— 操作数包含在指令中。它是一个 8 位或 32 位的有符号

数，其长度用指令操作码中的一位来指定。该位是 0 表示短版本，是 1 表示长版本。

直接方式（direct mode）——操作数的存储器地址是由指令中的一个 32 位值给出的。

寄存器方式（register mode）——操作数包含在指令中指定的一个通用寄存器中。

寄存器间接方式（register indirect mode）——操作数的存储器地址包含在指令中指定的一个通用寄存器中。

带有位移量的基址方式（base with displacement mode）——指令中给出了一个 8 位或 32 位的有符号位移量以及一个用作基址寄存器的通用寄存器。操作数的有效地址是基址寄存器的内容和位移量的和。

带有位移量的变址方式（index with displacement mode)——指令中给出了一个 32 位的有符号位移量，一个用作变址寄存器的通用寄存器和一个比例因子（1、2、4 或 8）。操作数的有效地址是变址寄存器的内容乘以比例因子之后再与位移量相加的和。

带有变址的基址方式（base with index mode）——指令中给出了两个通用寄存器和一个比例因子（1、2、4 或 8）。这两个寄存器分别用作基址寄存器和变址寄存器。操作数的有效地址是变址寄存器的内容乘以比例因子之后再与基址寄存器的内容相加的和。

665

带有变址和位移量的基址方式（base with index and displacement mode)——指令中给出了一个 8 位或 32 位的有符号位移量，两个通用寄存器和一个比例因子（1、2、4 或 8）。这两个寄存器分别用作基址寄存器和变址寄存器。操作数的有效地址是变址寄存器的内容乘以比例因子之后再与基址寄存器的内容和位移量相加的和。

IA-32 的寻址方式及其汇编语言的表示方式在表 E-1 中给出。表中还给出了操作数有效地址的计算。正如表中的脚注所说明的那样，寄存器 ESP 不能用作变址寄存器，因为它是作为处理器堆栈指针使用的。

表 E-1 IA-32 寻址方式

| 名 称 | 汇编语法 | 寻址功能 |
|---|---|---|
| 立即方式 | Value | Operand = Value |
| 直接方式 | Location | EA = Location |
| 寄存器方式 | Reg | EA= Reg，即操作数 = [Reg] |
| 寄存器间接方式 | [Reg] | EA = [Reg] |
| 带有位移量的基址方式 | [Reg + Disp] | EA = [Reg] + Disp |
| 带有位移量的变址方式 | [Reg * S + Disp] | EA = [Reg] × S + Disp |
| 带有变址的基址方式 | [Reg1 + Reg2 * S] | EA = [Reg1] + [Reg2] × S |
| 带有变址和位移量的基址方式 | [Reg1 + Reg2 * S + Disp] | EA = [Reg1] + [Reg2] × S + Disp |

Value=8 位或 32 位的有符号数

Location=32 位的地址

Reg, Reg1, Reg2= 通用寄存器 EAX、EBX、ECX、EDX、ESP、EBP、ESI、EDI 中的一个，其中的 ESP 不能用作变址寄存器

Disp=8 位或 32 位的有符号数，在带有位移量的变址方式中只能使用 32 位的有符号数

666 S= 比例因子 1、2、4 或 8

指令可以有 0 个、1 个或 2 个操作数。在双操作数指令中，源操作数（src）和目的操作数（dst）使用汇编语言指定的顺序为

OP dst, src

这个顺序与第 2 章中的相同。

使用 Move 指令来说明 IA-32 的寻址方式及其在汇编语言中的标记符号（notation）是非常方便的，指令

MOV　EAX, 25

将十进制数值 25 传送到目的寄存器 EAX 中，其中源操作数使用立即寻址方式指定，目的操作数使用寄存器寻址方式指定。

当一个数字常量作为一个操作数单独出现时，它就表示一个立即数值。数字常量可以用数字 0 到 9 这种十进制形式来表示。根据所使用的汇编程序，还可以用前缀 0x 或后缀 H 来表示十六进制数。在后一种情况中，以 A 到 F 开头的数还需要一个前缀 0，这样汇编程序就可以将一个十六进制数和一个标志区分开来。一些汇编程序还允许使用后缀 B 来表示二进制数。

符号名也可以当作操作数来使用。如果符号名 LOCATION 被定义为一个地址标志，则指令

MOV　EAX, LOCATION

隐式地使用直接寻址方式将存储地址 LOCATION 处的双字传送到寄存器 EAX 中。还可以显式地使用直接寻址方式。指令

MOV　EAX, DWORD PTR LOCATION

使用关键字 DWORD PTR 来表明标志 LOCATION 应该被解释为 32 位操作数的地址。

当需要将一个地址标志当作一个立即操作数来处理时，使用关键字 OFFSET。例如，指令

MOV　EBX, OFFSET LOCATION

使用立即寻址方式将地址标志 LOCATION 的值传送到寄存器 EBX 中。

一旦一个地址被装入一个寄存器中，就可以使用寄存器间接寻址方式来访问存储器中的操作数。指令

MOV　EAX, [EBX]

将地址包含在寄存器 EBX 中的存储单元的内容传送到寄存器 EAX 中。

以上例子说明了基本的寻址方式：立即、直接、寄存器和寄存器间接方式。其余的四种寻址方式在访问存储器中的数据操作数时提供了更加灵活的方式。 667

带有位移量的基址方式在图 E-3a 中作了说明。寄存器 EBP 被用作基址寄存器。位于地址 1060（与基地址 1000 相距 60 字节）处的双字操作数可由以下指令传送到寄存器 EAX 中：

MOV　EAX, [EBP+60]

指令可以对字节操作数和双字操作数进行处理。例如，还是假设基址寄存器 EBP 中的地址是 1000，可以使用指令

MOV　AL, [EBP+10]

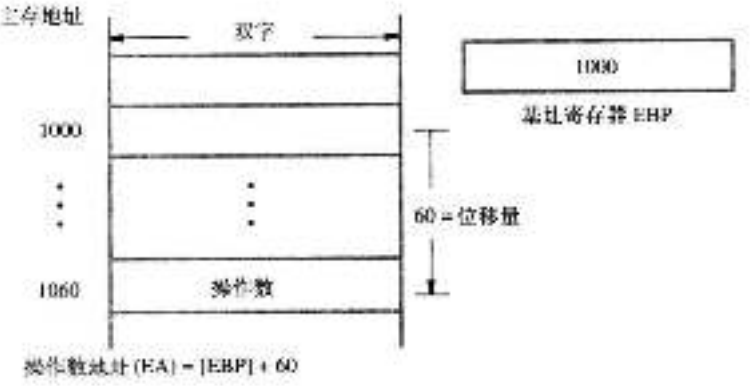
将地址 1010 处的字节操作数装入 EAX 寄存器的低字节部分。由于目的操作数 AL 是 EAX 寄存器的低字节部分，所以汇编程序选择用于字节数据的 Move 操作码版本。

最灵活的寻址方式就是带有变址和位移量的基址方式，图 E-3b 给出了一个例子，其中使用 EBP 和 ESI 作为基址和变址寄存器。这个例子展示了如何使用这种方式去访问双字操作数列表中的一个双字操作数。该列表在与基地址 1000 相距 200 的地方开始。对变址寄存器中的内容使用比例因子 4，就可以使用变址寄存器 ESI 中的变址序列 0，1，2，…来访问地址 1200，1204，1208，…处的连续双字操作数。在该图所示的例子中，当变址寄存器的值为 40 时，就访问地址 1360（也就是 1000 + 200 + 4 × 40）处的双字。这个操作数可通过指令

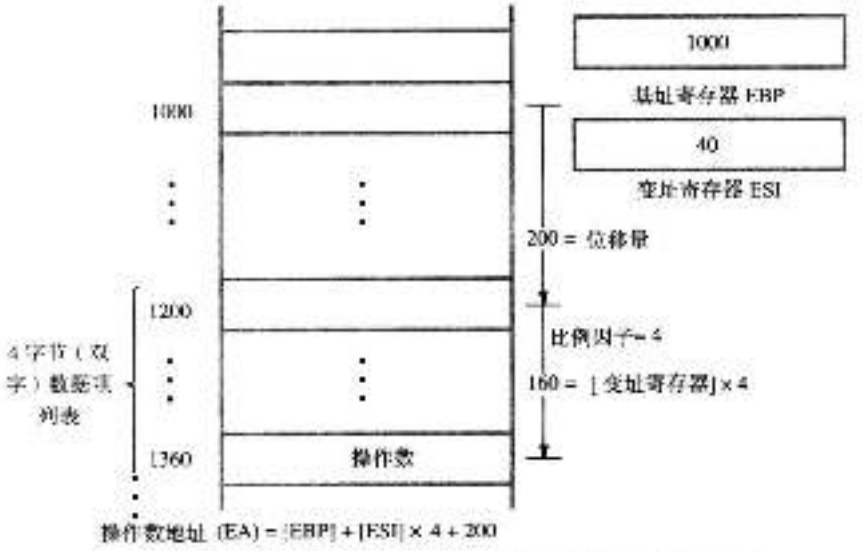
MOV　EAX, [EBP+ESI*4+200]

装入寄存器 EAX 中。在这种寻址方式中使用比例因子 4，可使得在一个程序循环中访问列表

中的连续双字操作数更加容易，只需在每遍循环中简单地将变址寄存器的值递增 1 即可。在对这两种方式作了比较详细的讨论之后，与其密切相关的带位移量的变址方式和带变址的基址方式将会很容易理解了。




a）带有位移量的基址方式，表示为 [EBP + 60]




b）带有位移量和变址的基址方式，表示为 [EBP + ESI * 4 + 200]

图 E-3 IA-32 体系结构中寻址方式举例

在结束寻址方式的讨论之前，对表 E-1 中所描述的两种方式进行注释是很有用的。带有位移量的基址方式看起来是多余的，因为通过带有位移量的变址方式（比例因子为 1）可以实现同样的效果。但是前一种方式可以编码在一个字节之中，所以是很有用的。此外，带有位移量的变址方式中的位移量只能是 32 位的，而带有位移量的基址方式中的位移量还可以是 8 位的。

## E.4 指令

IA-32 的指令集是很庞大的。它可以按照可变长度的指令格式进行编码，而无需使用完全规则的编码方案。大多数指令都有一个或两个操作数。在双操作数的情况下，只能有一个操作数在存储器中。另外一个操作数必须在处理器寄存器中或者是指令中指定的一个立即数。指令

集中提供了在存储器与处理器寄存器之间传送数据的指令，执行算术运算和逻辑运算的指令，以及移位/循环移位操作的指令。此外还包括 Jump（跳转）指令和子程序调用/返回指令，并直接支持对处理器堆栈进行的 Push（压栈）和 Pop（弹出）操作。

## E.4.1 机器指令格式

机器指令的一般格式如图 E-4 所示。指令是可变长的，变化的范围从 1 到 12 字节，由四个字段构成。操作码字段由一个或两个字节构成，大多数指令只需要一个字节。紧跟在操作码字段之后需要用一到两个字节保存寻址方式信息。

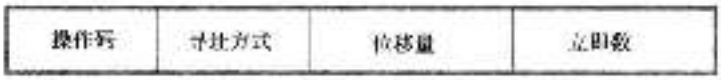



图 E-4 IA-32 指令格式

对于只使用一个寄存器来产生存储器操作数有效地址的指令，寻址方式字段只需要一个字节。表 E-1 中最后两种寻址方式的编码则需要两个字节，这些方式使用两个寄存器来产生存储器操作数的有效地址。

如果在计算存储器操作数有效地址的过程中需要使用一个位移量，那么这个位移量就用一个或四个字节进行编码，并放在紧跟着寻址方式字段之后的字段中。如果其中一个操作数是立即数，那么它被放在指令的最后一个字段中，占用一个或四个字节。

一些简单的指令，如将寄存器自增或自减的指令，只占用一个字节。例如，指令

```
INC   EDI
```

将寄存器 EDI 的值增加 1。在这种情况下，寄存器操作数通过操作码字节中的 3 位编码来指定。然而，对于大多数的指令和寻址方式来说，所使用的寄存器都是在寻址方式字段中指定的。

## E.4.2 汇编语言符号

汇编语言符号的部分内容已经在 E.3 节的寻址方式中作过介绍。本节将总结所使用的符号，以及汇编语言中的地址、立即值、操作数大小以及大写字符的使用等细节。

操作数名称之前的关键字 DWORD PTR 表示该名称是一个 32 位操作数的地址。同样，操作数名称之前的关键字 BYTE PTR 表示该名称是一个 8 位操作数的地址。另一方面，名称之前的关键字 OFFSET 表示该名称是一个立即值。

每一条汇编语言指令都必须包含足够的信息，以便于汇编程序确定操作数的大小。在寄存器寻址方式中，寄存器的名称提供了必要的信息。例如，作为指令操作数的寄存器 EAX 表示操作数的大小为 32 位，而寄存器 AL 则表示操作数的大小是 8 位的。汇编程序能够生成与操作数大小相对应的操作码。

在不包含寄存器寻址方式的情况下，汇编程序需要额外的信息。例如，单操作数指令可通过使用间接寻址方式或位移量寻址方式来指定存储器操作数。为了指明操作数的大小，需要包含关键字 DWORD PTR 或 BYTE PTR。

许多 IA-32 汇编程序对指令助记符和寄存器名称是不区分大小写的。Intel 的技术文档则一贯使用大写字符 [1]。为了符合这种描述风格，我们对所有的指令助记符和寄存器名称都使用大写字符。

### E.4.3 Move 指令

MOV 指令在存储器或 I/O 接口和处理器寄存器之间传送数据。数据传送的方向是从源端到目的端。状态寄存器中的条件码标志不受 MOV 指令执行的影响。

E.3 节中的例子显示了 MOV 指令是如何将数据从存储器传送到寄存器的。寄存器中的内容同样也可以传送到存储器或者其他的寄存器中。指令

MOV LOCATION, ECX

将寄存器 ECX 中的双字传送到地址为 LOCATION 的存储单元中。指令

MOV EBP, EDI

将寄存器 EDI 中的双字传送到寄存器 EBP 中，而寄存器 EDI 中的值保持不变。

MOV 指令不能对两个存储器操作数进行操作，但它可以将一个立即数传送到一个存储单元中，如

MOV DWORD PTR [EAX+16], 100

需要注意的是，汇编程序要求使用关键字 DWORD PTR（或 BYTE PTR）来指明指令中的操作数大小。

### E.4.4 加载有效地址（Load-Effective-Address）指令

E.3 节描述了 MOV 指令是如何使用关键字 OFFSET 来将一个地址装入寄存器中的。指令 LEA（Load-effective-address）同样可以实现这个功能。例如，如果 LOCATION 被定义为一个地址标志，则指令

[671]

LEA EAX, LOCATION

与

MOV EAX, OFFSET LOCATION

具有完全相同的效果。

LEA 指令可用来加载执行过程中所计算出的有效地址。例如，假设需要使用寄存器 EBX 作为指向存储器中数据操作数的指针，还假设所需的操作数是数组中的一个元素，该元素位于从数组起始位置偏移 12 个字节的地方。如果寄存器 EBP 中保存了数组的起始地址，那么指令

LEA EBX, [EBP-12]

计算所需的有效地址，并将其保存在寄存器 EBX 中。然后，该操作数便能被 Move 指令或其他指令通过使用 EBX 寄存器的寄存器间接寻址方式进行访问。

### E.4.5 算术指令

这一类的指令包括算术运算以及比较运算和求反运算。操作数可以放在存储器中，也可以放在寄存器中，或者被指定为立即值（双操作数指令）。操作数的大小可以是双字或者字节。

**1. 加法、减法、比较和求反运算**

双操作数的算术指令有：

- ADD（加法）
- ADC（带进位的加法，适用于多精度的算术运算）
- SUB（减法）
- SBB（带借位的减法，适用于多精度的算术运算）
- CMP（比较，目的操作数的值保持不变）

根据所执行运算的结果，这些指令会影响所有的条件码标志。指令

ADD EAX, EBX

执行 32 位运算

EAX ← [EAX] + [EBX]

指令

CMP [EBX+10], AL

执行 8 位运算

[[EBX] + 10] – [AL]

使用寄存器 AL 意味着操作数的大小是 1 字节。根据减法运算是否有溢出或进位、运算结果是否为负数或者零来对条件码标志进行设置。减法运算的结果将被丢弃。 [672]

单操作数的算术指令有：

- INC（自增）
- DEC（自减）
- NEG（求反）

NEG 指令会影响所有的条件码标志，但 INC 和 DEC 指令则不会影响 CF 标志。这些指令都必须包含关键字来指明操作数的大小，除非操作数使用的是寄存器寻址方式。指令

INC DWORD PTR [EDX]

使得地址在寄存器 EDX 中的存储单元内的双字自增 1。

2. 乘法

有符号整数的乘法指令 IMUL 执行 32 位乘法运算。根据所使用的指令形式，目的操作数可以是隐含的，64 位的乘积可以被截取为 32 位。

该指令的一种形式是

IMUL src

它隐含地使用 EAX 寄存器作为被乘数。由 src 指定的乘数可以放在寄存器中也可以放在存储器中。完整的 64 位乘积可以保存在寄存器 EDX（高 32 位）和 EAX（低 32 位）中。

该指令的第二种形式是

IMUL REG, src

目的操作数 REG 必须是一个通用寄存器。源操作数可以放在寄存器中也可以放在存储器中。在将乘积放到目的寄存器之前要将其截取为 32 位。

对于这两种形式，如果 64 位乘积的高 32 位中有一个 1（包括符号位），那么就会对 CF 和 OF 标志进行设置，否则 CF 和 OF 将被清零。其他的标志位是不确定的。

3. 除法

整数除法指令 IDIV 对 64 位的被除数和 32 位的除数进行操作，生成 32 位的商和 32 位的余数。该指令的格式是

IDIV src

其中源操作数是除数。64 位的被除数是由寄存器 EDX（高 32 位）和 EAX（低 32 位）中的内容组成的。执行完除法运算后，商被放入 EAX 中而余数则被放入 EDX 中。所有的条件码标志都是不确定的。除零操作会导致异常。

如果被除数是用 32 位来表示的，那么它必须先被放入 EAX 寄存器中，然后再被符号扩展为所需的 64 位操作数大小，放在寄存器 EAX 和 EDX 中。这是通过指令 CDQ（将双字转换为四字）来完成的，该指令没有操作数，因为其源操作数和目的操作数被隐含地指定为寄存器 EAX 和 EDX。 [673]

### E.4.6 Jump 指令和 Loop 指令

在 IA-32 的术语中，所有的转移指令都被称作 Jump（跳转）。这包括条件跳转和无条件跳转指令。这些指令可用来实现循环。通常情况下，计数器变量在每次通过循环的时候递减，同时条件跳转指令检查计数是否仍大于零，以便决定是否执行更多次的循环。因为这种方法是很常见的，所以有一条特殊的 Loop 指令可以将递减和条件跳转操作结合起来。

**1. 条件跳转指令与条件码标志**

条件跳转指令检测状态寄存器中的四个条件码标志。指令

```
JG  LABEL
```

是条件跳转指令的一个例子。跳转的条件是大于（greater-than），在操作码中用后缀 G 表示。表 E-2 总结了条件跳转指令以及相应的要检测的条件码标志组合。当前面一条算术或比较指令的操作数是有符号数时，将使用可以检测符号标志（SF）的 Jump 指令。例如，当涉及有符号数时，JG 指令就进行大于条件的测试，并考虑 SF 标志。对于无符号数，JA（jump-above）指令检测大于条件，但并不考虑 SF 标志。

表 E-2 IA-32 条件跳转指令

| 助记符 | 条件名称 | 条件检测 |
|---|---|---|
| JS | 符号（负数） | SF = 1 |
| JNS | 无符号（正数或 0） | SF = 0 |
| JE/JZ | 相等 / 为 0 | ZF = 1 |
| JNE/JNZ | 不等 / 不为 0 | ZF = 0 |
| JO | 溢出 | OF = 1 |
| JNO | 不溢出 | OF = 0 |
| JC/JB | 进位 / 无符号小于 | CF = 1 |
| JNC/JAE | 无进位 / 无符号大于或等于 | CF = 0 |
| JA | 无符号大于 | CF ∨ ZF = 0 |
| JBE | 无符号小于或等于 | CF ∨ ZF = 1 |
| JGE | 有符号大于或等于 | SF ⊕ OF = 0 |
| JL | 有符号小于 | SF ⊕ OF = 1 |
| JG | 有符号大于 | ZF ∨ (SF ⊕ OF) = 0 |
| JLE | 有符号小于或等于 | ZF ∨ (SF ⊕ OF) = 1 |

674

当汇编程序生成机器码时，条件跳转指令使用与紧跟 Jump 指令之后那条指令的地址相关的偏移量进行编码。该地址反映了提取 Jump 指令后指令指针（Instruction Pointer）被更新后的内容。如果偏移量在 −128 到 +127 的范围内，那么单个字节就足够了，这时，对条件跳转指令进行编码的字节总数就是两个字节，包括操作码字节。当跳转目标的距离超出这个范围时，就要使用四个字节的偏移量。

**2. 无条件跳转指令**

无条件跳转指令 JMP 会产生一个到目标地址处指令的转移。除了使用短（一个字节）或长（四个字节）的相对有符号偏移量来确定目标地址以外，像在条件跳转指令中一样，JMP 指令也可以使用其他的寻址方式。这种灵活性在产生目标地址时是非常有用的。考虑一下很多高级语言中的 Case 语句。在程序中的某处，可以使用 Case 语句来从很多可供选择的计算中选择一个来执行，其中每一个选择都被认为是一种情况。假设对于每一种情况，都定义一个程序去执行相应的计算。同样也假设这些程序的 4 字节起始地址保存在存储器内的一张表中，该表的

起始位置是 JUMPTABLE。用 0、1、2、…来对这些情况进行编号。在程序执行过程中，所选情况的编号被装入变址寄存器 ESI 中。通过执行指令

```
JMP   [JUMPTABLE + ESI * 4]
```

就可跳转到所选择情况对应的程序中，该指令使用的是带位移量的变址寻址方式。

3. 循环指令

循环通常依赖于一个计数器变量，该计数器变量在每一轮循环中减 1。将该变量保存在寄存器中可减少执行时间。当一条将寄存器中计数器减 1 的指令影响条件码标志时，在条件转移指令之前不需要进行显式地比较。一个循环可以实现如下：

```
        MOV   ECX, NUM_PASSES
START:
        ...
        DEC   ECX
        JG    START
```

这种形式的循环可以通过 LOOP 指令来用更简洁的方式表示。它结合了 DEC 和 JG 指令的功能，并隐含地使用寄存器 ECX 作为计数器变址。使用这条指令，循环可以实现如下：

```
        MOV ECX, NUM_PASSES
START:
        ...
        LOOP  START
```

LOOP 指令不会影响条件码标志。

675

**例 E.1**

使用目前为止所介绍过的指令，现在我们可以给出一个使用循环将数字相加的程序了，该程序与图 2-26 中的程序类似。假设存储单元 N 中保存了一个列表中 32 位整数的个数，而该列表在存储器中的起始地址是 NUM1。图 E-5a 所示的汇编语言程序将这些数相加，并将它们相加的和放到存储单元 SUM 中。

寄存器 EBX 中保存着地址值 NUM1。在 STARTADD 处的指令（该循环的第一条指令）中，寄存器 EBX 被用作带有变址的基址寻址方式中的基址寄存器，寄存器 EDI 被用作变址寄存器，在进入循环之前它被清为零。在第一遍循环中，地址 NUM1 处的第一个数被加到初值为 0 的 EAX 寄存器中，之后变址寄存器增加 1。在第二遍循环中，ADD 指令中的比例因子 4 将使得地址 NUM1 + 4 处的第二个 32 位数被加到 EAX 中。在后续的循环中，地址 NUM1 + 8、NUM1 + 12、…处的数也被加到 EAX 中。寄存器 ECX 被用作计数寄存器。该寄存器最初由程序的第二条指令将存储单元 N 中的内容装载到其中，并在每遍循环中减 1。当 [ECX] > 0 时，条件转移指令 JG 将使得程序转移回到 STARTADD 处。当 ECX 的内容达到 0 时，全部的数都已经相加完毕。不再执行转移，而是用 MOV 指令将寄存器 EAX 中的和写到存储单元 SUM 中。

观察图 E-5a 中的程序，我们可以为这个任务编写一个更简洁的程序。观察的第一点是两条指令的序列

```
DEC   ECX
JG    STARADD
```

可由如下这一条指令替代：

```
LOOP   STARADD
```

```
            LEA     EBX, NUM1          将 EBX 用作基址寄存器
            MOV     ECX, N             将 ECX 用作计数寄存器
            MOV     EAX, 0             将 EAX 用作累加器
            MOV     EDI, 0             将 EDI 用作变址寄存器
STARTADD:   ADD     EAX, [EBX + EDI * 4]  将下一个数加到 EAX 中
            INC     EDI                变址寄存器增 1
            DEC     ECX                计数寄存器减 1
            JG      STARTADD           如果 [ECX] > 0，则转移返回
            MOV     SUM, EAX           将和保存到存储器中
```

a) 直接方法

```
            LEA     EBX, NUM1          装入基址寄存器 EBX 并进行
            SUB     EBX, 4             调整使其保存 NUM1 – 4
            MOV     ECX, N             初始化计数/变址寄存器 ECX
            MOV     EAX, 0             将 EAX 用作累加器
STARTADD:   ADD     EAX, [EBX + ECX * 4]  将下一个数加到 EAX 中
            LOOP    STARTADD           ECX 减 1，如果 [ECX] > 0，
                                       则转移返回
            MOV     SUM, EAX           将和保存到存储器中
```

b) 更简洁的程序

图 E-5 图 2-26 中程序的实现

该指令将 ECX 寄存器减 1，之后如果 ECX 的内容还没有达到 0，就转移到目标地址 STARTADD 处。观察的第二点就是使用了两个寄存器 EDI 和 ECX 作为计数器。如果我们反向扫描所要相加的数字列表，即从列表的最后一个数开始，则只需使用一个计数寄存器。由于寄存器 ECX 是 LOOP 指令隐含引用的寄存器，所以我们使用该寄存器。假设 [N] = $n$，当 EDI 中包含的值序列是 0、1、2、…、($n$ 1) 时，第一个程序就使用地址序列 NUM1，NUM1+4，NUM1+8，…、NUM1+4($n$–1) 来访问这些数。新程序如图 E-5b 所示，当 ECX 中包含的值序列是 $n$、$n$ 1、…、1 时，该程序使用的地址序列是 (NUM1 4)+4$n$、(NUM1 4)+4($n$ 1)、…、(NUM1–4)+4(1)。因此，为了解决 EDI 序列与 ECX 序列之间的不同，要将新程序的基址寄存器 EBX 中的值从 NUM1 改成 NUM1 4。在新程序的最后一遍循环中，执行 LOOP 指令之前，[ECX] = 1，最后一个要相加的数在存储单元 NUM1 中。

**例 E.2**

图 2-11 中的程序计算了一组学生所参与的三项测验的所有分数之和。在程序中使用 Load 指令来从存储器中提取操作数。图 E-6 给出了该程序的 IA-32 版本。ADD 指令中可以使用带有位移量的基址寻址方式，这样，我们就不必使用单独的指令去访问存储器操作数。

```
            MOV     EAX, OFFSET LIST   获取地址 LIST
            MOV     EBX, 0
            MOV     ECX, 0
            MOV     EDX, 0
            MOV     EDI, N             加载 n 值
LOOP:       ADD     EBX, [EAX + 4]     加上当前学生的测验 1 成绩
            ADD     ECX, [EAX + 8]     加上当前学生的测验 2 成绩
            ADD     EDX, [EAX + 12]    加上当前学生的测验 3 成绩
            ADD     EAX, 16            递增指针
            DEC     EDI                递减计数器
            JG      LOOP               如果没完成就循环回到前面
            MOV     SUM1, EBX          保存测验 1 的总和
            MOV     SUM2, ECX          保存测验 2 的总和
            MOV     SUM3, EDX          保存测验 3 的总和
```

图 E-6 图 2-11 中程序的实现

### E.4.7 逻辑指令

IA-32 体系结构中有执行逻辑与、或和异或操作的指令。这种指令对两个操作数进行位操作，并将结果保存在目的单元中。例如，假设寄存器 EAX 中的值是十六进制数 0000FFFF，寄存器 EBX 的值是 02FA62CA。指令

```
AND   EBX, EAX
```

将 EBX 的左半部分全部清为 0，右半部分保持不变。结果放在 EBX 中，是 000062CA。

指令集中还有一条 NOT 指令，它对操作数的所有位进行逻辑求反，也就是将全部的 1 变成 0，全部的 0 变成 1。

### E.4.8 移位和循环移位指令

利用逻辑移位或算术移位可以将一个操作数左移或右移，移位的位数由一个给定的数值确定。移位指令的格式是

```
OP   dst, count
```

其中要被移位的目的操作数可使用任意一种寻址方式指定，count 可以用一个 8 位的立即数给出，也可以放在一个 8 位的寄存器 CL 中。共有四条移位指令：

- SHL（逻辑左移）
- SHR（逻辑右移）
- SAL（算术左移；其操作等同于 SHL）
- SAR（算术右移）

移位操作已在 2.8.2 节中作过讨论，并在图 2-23 中进行了展示。

除了移位指令之外，还有四条循环移位指令：

- ROL（不带进位标志 CF 的循环左移）
- ROR（不带进位标志 CF 的循环右移）
- RCL（带进位标志 CF 的循环左移）
- RCR（带进位标志 CF 的循环右移）

所有这四种操作已在图 2-25 中做了说明。循环移位指令要求 count 参数是一个 8 位的立即数或者是寄存器 CL 中的一个 8 位值。

**例 E.3**

考虑图 2-24 所示的 BCD 数字打包程序，程序中使用了移位指令和逻辑指令。该程序的 IA-32 代码如图 E-7 所示。将两个 ASCII 码字节装入寄存器 AL 和 BL 中。SHL 指令将 AL 中的字节左移四位，低四位用 0 填充。AND 指令将第二个字节的高四位置 0。最后，使用 OR 指令将所期望的 4 位形式的 BCD 码合并到 AL 中，之后再将合并的结果保存到存储器的字节单元 PACKED 中。

```
LEA   EBP, UNPK      EBP 指向第一个字节
MOV   AL, [EBP]      将第一个字节装入 AL 中
SHL   AL, 4          左移 4 位
MOV   BL, [EBP+1]    将第二个字节装入 BL 中
AND   BL, 0FH        将高 4 位清零
OR    AL, BL         连接 BCD 数字
MOV   PACKED, AL     保存结果
```

图 E-7 将两个 BCD 数字打包成一个字节的程序，对应于图 2-24

676 ? 678

### E.4.9 子程序链接指令

2.7 节描述了处理器堆栈在子程序链接中的使用。在 IA-32 体系结构中，寄存器 ESP 被用

作堆栈指针，指向处理器堆栈的当前栈顶元素（TOS）。堆栈的增长方向是存储器地址的低地址方向。堆栈的宽度是 32 位，也就是说，所有的堆栈项都是双字的。

有两条指令用来将单个元素压入堆栈和从堆栈中弹出。指令

PUSH src

将 ESP 中的值减 4，然后将 src 处的双字保存到 ESP 所指向的存储单元中。指令

POP dst

是一个相反的过程，它释放 ESP 所指向的 TOS 处的双字，将其保存在 dst 处，然后将 ESP 的值加 4。这些指令隐含地将 ESP 作为堆栈指针。源操作数和目的操作数采用 IA-32 寻址方式来指定。

此外，还有两条指令可以用来将多个寄存器的内容压栈和弹出。指令

679 PUSHAD

可以将所有 8 个通用寄存器 EAX 到 EDI 中的内容压入堆栈，而指令

POPAD

则按照相反的顺序将它们弹出。当 POPAD 到达原来保存在 ESP 中的栈顶时，它将丢弃那四个字节而不将它们装入 ESP 中，并继续将剩余的值弹出到各自的寄存器中。这两条指令作为实现了程序的一部分，可用来有效地保存和恢复全部寄存器的内容。

图 E-5a 中的列表加法程序可以写成一个如图 E-8a 所示的子程序，参数通过寄存器进行传递。调用程序将列表中第一个数的存储器地址 NUM1 装入寄存器 EBX 中，保存在存储单元 N 中的列表项个数被装入寄存器 ECX 中。调用程序希望获得通过寄存器 EAX 返回的最终的和。因此，寄存器 EBX、ECX 和 EAX 就用来传递参数。寄存器 EDI 在子程序执行加法运算时作为变址寄存器，所以它的内容必须在子程序中通过 PUSH 和 POP 指令进行保存和恢复。

子程序可通过下面的指令来调用：

CALL LISTADD

该指令首先将返回地址压入堆栈，然后跳转到 LISTADD 处。返回地址是紧跟在 CALL 指令之后的 MOV 指令的地址。子程序将寄存器 EDI 的内容保存在堆栈中。图 E-8b 给出了此时堆栈中的内容。在执行完循环之后，寄存器 EDI 中原来被保存的内容即可被恢复。指令 RET 通过将栈顶元素弹出到指令指针（寄存器 EIP）中来将执行控制返回给调用程序。

图 E-9a 给出的是对图 E-5a 中的程序使用堆栈来传递参数而重新编写的一个子程序。参数 NUM1 和 $n$ 由调用程序中的两条 PUSH 指令压入堆栈。注意，需要使用关键字 OFFSET 来将 NUM1 所表示的地址压入堆栈。在执行完 CALL 指令之后，栈顶处于第二级，如图 E-9b 所示。寄存器 EDI、EAX、EBX 和 ECX 在这个子程序中的作用与在图 E-8 中的相同。在它们的值被保存之后，子程序中的前八条指令将初始值和参数装入其中。这时，栈顶处于第三级。当这些数值由四条指令的循环相加之后，相加的和被放到堆栈中，覆盖参数 NUM1。在执行 RET 指令之后，调用程序中的 ADD 指令和 POP 指令将参数 $n$ 从堆栈中移除并将所返回的相加和弹出到存储单元 SUM 中。此时栈顶被恢复到第一级。

我们还要考虑子程序嵌套的情况。图 E-10 给出了图 2-21 中程序的 IA-32 码。在图 E-11 中给出了第一个和第二个子程序相对应的堆栈结构。寄存器 EBP 被用作结构指针。在图 E-10 中我们不使用 PUSHAD 和 POPAD 指令来将所有八个通用寄存器的内容进行压栈和弹出，而
680 是选择使用单个的 PUSH 和 POP 指令，因为这个子程序只使用了一半数量的寄存器。

```
调用程序
                .
                .
        LEA     EBX, NUM1           将参数装入 EBX 和 ECX 中
        MOV     ECX, N
        CALL    LISTADD             转移到子程序
        MOV     SUM, EAX            将和保存到存储器中
                .
                .
子程序
LISTADD:PUSH    EDI                 保存 EDI
        MOV     EDI, 0              将 EDI 用作变址寄存器
        MOV     EAX, 0              将 EAX 用作累加器
STARTADD:ADD    EAX, [EBX + EDI * 4] 加上下一个数
        INC     EDI                 递增变址
        DEC     ECX                 递减计数器
        JG      STARTADD            如果 [ECX] > 0，则转移返回
        POP     EDI                 恢复 EDI
        RET                         返回到调用程序
```

a) 调用程序和子程序



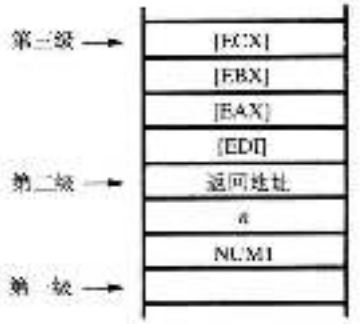

b) 子程序中保存 EDI 之后的堆栈内容

图 E-8 将图 E-5a 中的程序写成一个子程序：参数通过寄存器传递

```
（假设栈顶位于第一级之下）
调用程序
        PUSH    OFFSET NUM1         将参数压入堆栈
        PUSH    N
        CALL    LISTADD             转移到子程序
        ADD     ESP, 4              将 n 从栈中移除
        POP     SUM                 将相加和弹出到 SUM 中
                .
                .
子程序
LISTADD:PUSH    EDI                 保存寄存器
        PUSH    EAX
        PUSH    EBX
        PUSH    ECX
        MOV     EDI, 0              将 EDI 用作变址寄存器
        MOV     EAX, 0              使用 EAX 来累加和
        MOV     EBX, [ESP + 24]     载入地址 NUM1
        MOV     ECX, [ESP + 20]     载入计数值 n
STARTADD:ADD    EAX, [EBX + EDI * 4] 加上下一个数
        INC     EDI                 递增变址
        DEC     ECX                 递减计数器
        JG      STARTADD            如果没完成，则转移返回
        MOV     [ESP + 24], EAX     用相加和将堆栈中的 NUM1 覆盖
        POP     ECX                 恢复寄存器
        POP     EBX
        POP     EAX
        POP     EDI
        RET                         返回
```

a) 调用程序和子程序

图 E-9 将图 E-5a 中的程序写成一个子程序：参数通过堆栈传递




b）不同时刻的堆栈内容

图 E-9 （续）

| 地址 | | 指令 | | 注释 |
|---|---|---|---|---|
| **调用程序** | | | | |
| | | ⋮ | | |
| 2000 | | PUSH | PARAM2 | 将参数放入堆栈 |
| 2006 | | PUSH | PARAM1 | |
| 2012 | | CALL | SUB1 | |
| 2017 | | POP | RESULT | 保存结果 |
| | | ADD | ESP, 4 | 恢复堆栈级别 |
| | | ⋮ | | |
| **第一个子程序** | | | | |
| 2100 | SUB1: | PUSH | EBP | 保存结构指针寄存器 |
| | | MOV | EBP, ESP | 载入结构指针 |
| | | PUSH | EAX | 保存寄存器 |
| | | PUSH | EBX | |
| | | PUSH | ECX | |
| | | PUSH | EDX | |
| | | MOV | EAX, [EBP + 8] | 获取第一个参数 |
| | | MOV | EBX, [EBP + 12] | 获取第二个参数 |
| | | ⋮ | | |
| | | PUSH | PARAM3 | 将参数放入堆栈 |
| 2160 | | CALL | SUB2 | |
| 2165 | | POP | ECX | 将 SUB2 的结果弹出放入 ECX |
| | | ⋮ | | |
| | | MOV | [EBP + 8], EDX | 将答案放入堆栈 |
| | | POP | EDX | 恢复寄存器 |
| | | POP | ECX | |
| | | POP | EBX | |
| | | POP | EAX | |
| | | POP | EBP | 恢复结构指针寄存器 |
| | | RET | | 返回到主程序 |
| **第二个子程序** | | | | |
| 3000 | SUB2: | PUSH | EBP | 保存结构指针寄存器 |
| | | MOV | EBP, ESP | 载入结构指针 |
| | | PUSH | EAX | 保存寄存器 |
| | | PUSH | EBX | |
| | | MOV | EAX, [EBP + 8] | 获取参数 |
| | | ⋮ | | |
| | | MOV | [EBP + 8], EBX | 将 SUB2 的结果放入堆栈 |
| | | POP | EBX | 恢复寄存器 |
| | | POP | EAX | |
| | | POP | EBP | 恢复结构指针寄存器 |
| | | RET | | 返回到第一个子程序 |

图 E-10 嵌套子程序：图 2-21 中程序的实现

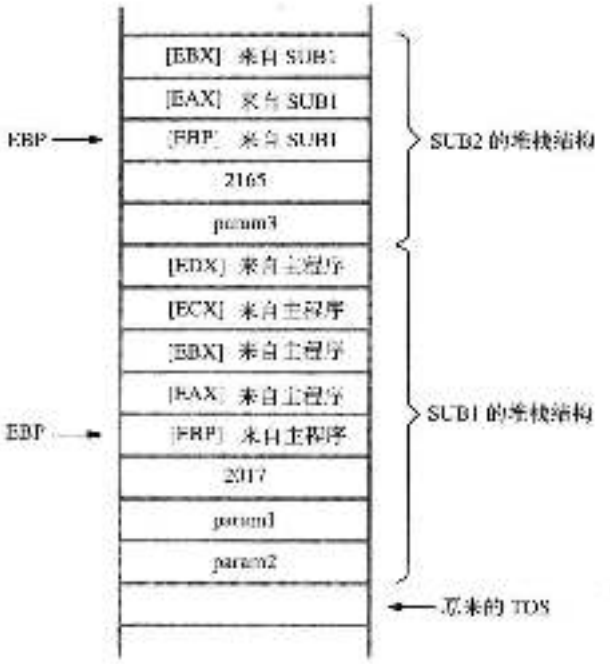



图 E-11　图 E-10 的堆栈结构

## E.4.10　大数操作

E.4.5 节描述了多种算术指令，包括那些可以对长度超过单个通用寄存器的 32 位宽度的数据进行运算的指令。ADC 和 SBB 指令将状态寄存器中的 CF 标志位作为进位位。这些指令对于多精度算术运算是非常有用的。

**例 E.4**

使用 ADC 指令将超出 32 位寄存器范围的比较大的数相加，如图 E-12 所示。要相加的两个十六进制数是 10A72C10F8 和 4A5C00FE04。寄存器 EAX 和 EBX 分别保存 10A72C10F8 的低位和高位部分。同样，寄存器 ECX 和 EDX 分别保存 4A5C00FE04 的低位和高位部分。ADD 指令用来将低 32 位相加，产生了进位输出 1，使得 CF 标志被置为 1。然后，在将高位部分相加时，ADC 指令将这个标志作为进位输入位。相加和的低位和高位部分分别保存在寄存器 EAX 和 EBX 中。

```
MOV   EAX, 0A72C10F8H   ; EAX 中保存着 A72C10F8
MOV   EBX, 10H          ; EBX 中保存着 10
MOV   ECX, 5C00FE04H    ; ECX 中保存着 5C00FE04
MOV   EDX, 4AH          ; EDX 中保存着 4A
ADD   EAX, ECX          ; 将低 32 位相加，根据进位输出设置 CF 标志
ADC   EBX, EDX          ; CF 标志作为进位输入位，将高 32 位相加
```

图 E-12　使用 ADC 指令将大于 32 位的数相加

## E.5　汇编指示

如 2.5.1 节所述，需要使用汇编指示（Assembler Directive）来定义程序的数据区，以及数据单元的符号名与实际的物理地址值之间的对应关系。

图 E-5b 中程序的完整汇编语言程序如图 E-13 所示，它对应于图 2-13 中的程序。图 E-13 中的汇编指示符与广泛使用的微软 MASM 汇编程序所定义的汇编指示符一致。.CODE 和 .DATA 指示符定义了程序代码段以及数据段的开头。在数据段中，DD 指示符为双字大小的数据分配存储空间。标签 SUM 是保存计算得到的总和这个双字的单元地址，它被初始化为 0。标签 N 保存数字 150 的单元地址。最后为数字表列分配存储空间。DUP 关键字用来将存储器中指定数量的连续单元初始化为指定的值。这样，150 个连续单元被初始化为 0。此外，还有其他的汇编指示符，例如，DB 可为字节大小的数据分配存储空间，EQU 可以给一个标签分配一个常量值。

```
                .CODE
                LEA     EBX, NUM1
                SUB     EBX, 4
                MOV     ECX, N
                MOV     EAX, 0
STARTADD:       ADD     EAX, [EBX + ECX * 4]
                LOOP    STARTADD
                MOV     SUM, EAX

                .DATA
SUM             DD      0               为相加和预留一个双字大小的空间
N               DD      150             列表中有 N=150 个双字
NUM1            DD      150 DUP(0)      为 150 个双字预留存储空间
                END
```

图 E-13 对应于图 2-13 的程序

681 - 685

## E.6 示例程序

本节将介绍 2.12 节中所描述示例程序的 IA-32 代码。

### E.6.1 向量点积程序

图 E-14 所示的是对保存在存储器中、起始地址是 AVEC 和 BVEC 的两个数值向量计算其点积的程序，它对应于图 2-28 中的程序。用带变址的基址寻址方式来访问每个向量中的连续元素。寄存器 EDI 被用作变址寄存器。由于假设向量元素是双字长（4 字节）的数，所以使用了比例因子 4。寄存器 ECX 被用作循环计数器，初始值为 $n$。这里允许使用 LOOP 指令，首先将 ECX 减 1，然后如果 ECX 的内容还没有到达 0，就条件转移到目的地址 LOOPSTART 处。假设两个向量元素的乘积可以放在一个双字中，所以乘法指令 IMUL 明确指定了所需的目的寄存器 EDX，正如 E.4.5 节中所描述的那样。

```
                LEA     EBP, AVEC       EBP 指向向量 A
                LEA     EBX, BVEC       EBX 指向向量 B
                MOV     ECX, N          ECX 是循环计数器
                MOV     EAX, 0          EAX 累加点积
                MOV     EDI, 0          EDI 是变址寄存器
LOOPSTART:      MOV     EDX, [EBP + EDI * 4]    计算下一对元素的点积
                IMUL    EDX, [EBX + EDI * 4]
                INC     EDI             递增变址
                ADD     EAX, EDX        累加到先前的和中
                LOOP    LOOPSTART       如果没有完成，则转移返回
                MOV     DOTPROD, EAX    将点积保存到存储器中
```

图 E-14 计算两个向量点积的程序

## E.6.2 字符串搜索程序

图 E-15 给出了图 2-31 中程序的 IA-32 版本。它在一个给定的目标字符串 T 中确定第一个与模式字符串 P 相匹配的实例。由于只有 8 个通用寄存器，所以 EAX 中的双字值被保存到存储单元 TMP 中，这样，可以在循环中使用字节大小的寄存器 AL。当再次需要所保存的双字值时，可以将该值恢复到 EAX 中。 686

```
        MOV   EAX, OFFSET T          EAX 指向字符串 T
        MOV   EBX, OFFSET P          EBX 指向字符串 P
        MOV   ECX, N                 获取值 n
        MOV   EDX, M                 获取值 m
        SUB   ECX, EDX               计算 n - m
        ADD   ECX, EAX               ECX 是 T(n - m) 的地址
        ADD   EDX, EBX               EBX 是 P(m) 的地址
LOOP1:  MOV   ESI, EAX               用 ESI 遍历字符串 T
        MOV   EDI, EBX               用 EDI 遍历字符串 P
        MOV   DWORD PTR TMP, EAX     将 EAX 保存到存储器中，允许使用 AL
LOOP2:  MOV   AL, [ESI]              从字符串 T 中获取字符
        CMP   AL, [EDI]              将其与字符串 P 中的字符进行比较
        JNE   NOMATCH
        INC   ESI                    字符串 T 的指针增加 1
        INC   EDI                    字符串 P 的指针增加 1
        CMP   EDX, EDI               检查是否在 P(m) 中
        JG    LOOP2                  如果没有完成，再次循环
        MOV   EAX, DWORD PTR TMP     临时使用后恢复 EAX
        MOV   DWORD PTR RESULT, EAX  保存 T(i) 的地址
        JMP   DONE
NOMATCH: MOV  EAX, DWORD PTR TMP     临时使用后恢复 EAX
        ADD   EAX, 1                 指向 T 中的下一个字符
        CMP   ECX, EAX               检查是否在 T(n - m) 中
        JGE   LOOP1                  如果没有完成，再次循环
        MOV   DWORD PTR RESULT, -1   没有找到匹配
DONE:   下一条指令
```

图 E-15 字符串搜索程序

## E.7 中断与异常

实现 IA-32 体系结构的处理器使用两根中断请求线，一根是不可屏蔽中断请求线 NMI，另一根是可屏蔽中断请求线，也被称为用户中断请求线 INTR。NMI 上的中断请求总是会被处理器接收，而 INTR 上的请求仅当它们的优先级比当前正在运行程序的优先级高的情况下才会被接收。可通过设置状态寄存器中的中断允许位 IF 来允许或禁止 INTR 中断。 687

除了外部中断，还会发生其他可使得程序在执行过程中产生异常的事件，这些事件包括无效的操作码、除零操作和溢出，还包括跟踪和断点中断。

这些事件中任一事件的发生都会使处理器转移到一个中断服务程序。我们为每个中断或异常分配一个向量号。在 INTR 情况下，当中断请求得到确认后，I/O 设备通过总线发送其向量号。对于其他的异常，向量号是预先分配好的。根据向量号，处理器就可以通过一个中断描述符表（Interrupt Descriptor Table）来确定中断服务程序的起始地址了。

IA-32 处理器依赖于一个配套的高级可编程中断控制器（Advanced Programmable Interrupt Controller，APIC），各种 I/O 设备都是通过这个控制器来与处理器连接的。中断控制器实现了不同设备间的优先级结构，并为每一个设备发送一个适当的向量号给处理器。

图 E-1 所示的状态寄存器中包含了中断允许标志（IF）、陷阱标志（TF）以及 I/O 权限

级别（IOPL）。当 IF=1 时，可以接收 INTR 中断。陷井标志允许每条指令执行后都产生跟踪中断。

中断在操作系统的上下文中尤其重要。IA-32 体系结构定义了一个复杂的特权结构，从而可以使操作系统的不同部分在四个不同的特权级别上执行。每一个特权级别使用处理器地址空间中的不同的段。从一个级别切换到另一个级别包含了大量的检查操作，这些操作由一种称为门（gate）的机制来实现。这样就可以构建一个高度安全的操作系统。处理器也可以运行在一种简单的模式下，在这种模式中没有实现特权，所有的程序都运行在相同的段中。在此我们只讨论这种简单模式。

当收到一个中断请求或者发生一个异常时，处理器执行以下操作：

1）它将状态寄存器、代码段寄存器（CS）和指令指针（EIP）压入处理器堆栈中。

2）对于不正常的执行条件所引发的异常，处理器会将一个代码压入堆栈来描述产生异常的原因。

3）处理器清除相应的中断允许标志，这样可以禁止同一中断源再次发送的中断。

4）处理器根据中断的向量号从中断描述符表中读取中断服务程序的起始地址，并将其装入 EIP 中去，然后继续执行。

响应中断请求后，中断服务程序使用中断返回指令 IRET 返回到被中断的程序。IRET 指令将 EIP、CS 和状态寄存器的内容从堆栈弹出到相应的寄存器中，从而恢复处理器的状态。

对于子程序，中断服务程序可能会通过保存寄存器或者使用堆栈结构保存局部变量来创建一个临时的工作空间。在执行 IRET 指令之前，它必须恢复先前保存的寄存器并确保堆栈指针 ESP 指向返回地址。 688

## E.8 输入/输出示例

本节利用第二章描述的 I/O 示例和接口寄存器来展示 IA-32 指令是如何用于轮询以及基于中断的 I/O 操作的。

图 E-16 给出了图 3-5 中程序的 IA-32 版本，这里使用轮询的方式实现输入和输出操作。BT 指令对应于图 3-5 中的 TestBit 指令，它将指定位的值复制到状态寄存器的 CF 标志中。该程序将每个字符从键盘接口读入到寄存器 AL 中，以便于之后与回车符 CR 作比较。随着每个字符从寄存器 AL 保存到存储器中，寄存器 EBX 通过递增来增加指针。

```
        LEA   EBX, LOC          初始化寄存器 EBX，使其指向主存中存储字符的
                                第一个单元的地址
READ:   BT    KBD_STATUS, 1     等待一个字符被输入到键盘缓冲区 KBD_DATA 中
        JNC   READ
        MOV   AL, KBD_DATA      将字符传送到 AL 中（这将 KIN 清为 0）
        MOV   [EBX], AL         将字符存储到主存中，并递增指针
        INC   EBX
ECHO:   BT    DISP_STATUS, 2    等待显示器准备就绪
        JNC   ECHO
        MOV   DISP_DATA, AL     把刚读入的字符移到显示器缓冲寄存器中
                                （这将 DOUT 清为 0）
        CMP   AL, CR            如果不是 CR，则转移回去读取另一个字符
        JNE   READ
```

图 E-16 读取并显示一行字符的程序

为了说明中断是如何用于 I/O 操作的，图 E-17 实现了图 3-10 中的例子。初始化代码中的

BTS 指令用来对键盘接口中的中断允许位进行设置，STI 指令将状态寄存器中的 IF 标志置为 1，使得处理器可以响应中断请求。中断服务程序中的 BTR 指令用于清除键盘接口中的中断允许位。指针被保存在一个存储单元中，中断服务程序每处理一个字符指针便增加 1。我们假设键盘发送一个向量号为 $i$ 的中断请求，并且将中断服务程序的起始地址 READ 装入中断描述符表 689
中相应的表项 $i$ 中。

```
中断服务程序

READ:   PUSH    EAX                         将寄存器 EAX 保存到堆栈中
        PUSH    EBX                         将寄存器 EBX 保存到堆栈中
        MOV     EBX, PNTR                   载入地址指针
        MOV     AL, KBD_DATA                将字符传送到 AL 中
        MOV     [EBX], AL                   将字符写入存储器并递增指针
        INC     DWORD PTR PNTR
ECHO:   BT      DISP_STATUS, 2              等待显示器准备就绪
        JNC     ECHO
        MOV     DISP_DATA, AL               显示刚读入的字符
        CMP     AL, CR                      检查刚读入的字符是否为回车符（CR）
        JNE     RTRN                        如果不是 CR 则返回
        MOV     DWORD PTR EOL, 1            指示行的结束
        BTR     KBD_CONT, 1                 禁止键盘中断
RTRN:   POP     EBX                         恢复寄存器
        POP     EAX
        IRET                                从中断返回

主程序

        MOV     DWORD PTR PNTR, OFFSET LINE 初始化缓冲区指针
        MOV     DWORD PTR EOL, 0            清除行结束指示变量
        BTS     KBD_CONT, 1                 允许键盘中断
        STI                                 设置处理器寄存器中的中断标志
        下一条指令
```

图 E-17 使用中断方式读取一行字符并使用轮询方式显示该行字符的程序

## E.9 标量浮点运算

IA-32 体系结构定义了许多浮点运算指令。它们由一个单独的浮点运算单元（floating-point unit，FPU）执行，该浮点运算单元中包含了额外的寄存器。FPU 允许浮点运算与其他指令并发执行。本节将概括介绍 IA-32 体系结构的浮点功能以及一些浮点运算指令。第 1 章中已 690
经介绍了浮点数的表示方法，浮点数的算术运算也在第 9 章中进行了讨论。

所有的浮点运算都是对 80 位的扩展双精度数进行的。图 E-1 所示的 8 个 80 位浮点数据寄存器就是为此目的而提供的。对这些寄存器中保存的扩展双精度数进行运算可减小计算过程中累加的舍入误差，如 9.7 节所介绍的那样。FPU 中还有额外的控制和状态寄存器，其细节可在 Intel 的技术文档 [1] 中查阅。

FPU 的一个独特功能是可将 8 个浮点数据寄存器看作堆栈。一些使用这些寄存器进行算术运算或数据传输操作的指令也可以执行压栈或弹出操作。指向寄存器堆栈栈顶的指针可由 FPU 维护，不需要对该指针进行特殊的初始化操作。压栈和弹出操作将指针调整为对 8 取模的形式，使其在必要的时候可以循环回去。

在汇编语言中，浮点寄存器操作数可由助记符 ST($i$) 确定，这里 $i$ 是与寄存器堆栈栈顶相关的变址（$0 \leqslant i \leqslant 7$）。例如，ST(0) 表示寄存器堆栈当前栈顶的那个寄存器，ST(1) 表示堆栈中的下一个寄存器，以此类推，ST(7) 指的是最后一个寄存器。在用汇编语言编写程序时，

程序员需要跟踪浮点指令序列所执行的压栈与弹出操作，从而可以正确地识别每条指令的操作数。

浮点加载指令都有一个显式的操作数用来指明存储器中的源单元。这些指令将从存储器中读取的值压入寄存器堆栈中。目的单元是在执行加载指令时被隐含地确定为 ST(7) 的寄存器。执行完加载指令后，目的寄存器变成 ST(0)，作为寄存器堆栈新的栈顶。这是因为指向寄存器堆栈栈顶的指针被调整为对 8 取模的形式。先前的寄存器 ST(0) 变成 ST(1)，ST(1) 变成 ST(2)，依此类推。

对于浮点存储指令，源操作数隐含地保存在寄存器 ST(0) 中。另一个明确的操作数指明了存储器中的目的单元，可将寄存器 ST(0) 中的值写入到该单元中去。一些存储指令使寄存器堆栈保持不变，其他的会弹出 ST(0)。指向寄存器堆栈栈顶的指针被调整为对 8 取模的形式，但跟加载指令不同的是，它们的方向是相反的。完成弹出操作后，先前的寄存器 ST(0) 变成 ST(7)，ST(1) 变成 ST(0)，依此类推。

浮点算术指令的源操作数或目的操作数必须在寄存器 ST(0) 中。对于某些指令，寄存器 ST(0) 被隐含地指定为目的单元，只有源操作数需要明确地指定，且必须在存储器中。其他指令则显式地指明两个操作数，一个必须在寄存器 ST(0) 中，它既可以是源操作数也可以是目的操作数，而另外一个可以在任何一个寄存器 ST(i) 中。一些指令不需要显式的操作数，因为对于源单元和目的单元它们都隐含地使用了寄存器 ST(0)。

浮点寄存器总是保存 80 位的扩展双精度数，而存储器可以保存 32 位单精度或 64 位双精
691 度表示的数。当涉及大量的浮点数但不需要太高的精度时，单精度表示就降低了对存储空间的需求。在进行算术运算之前，FPU 会自动地将存储器中的单精度或双精度操作数转换成扩展的双精度数。当需要将它们传送到存储器中时，扩展的双精度数又被转换成单精度或双精度的表示形式。在与存储器传输数据的时候，FPU 还会将整数和扩展的双精度浮点表示互相转换。不用软件而用硬件来实现这样的转换可减少执行时间和代码大小。

### E.9.1 Load 指令和 Store 指令

FLD 指令从一个存储单元中读取一个浮点数并将其压入到浮点寄存器堆栈中。关键字 DWORD PTR 用来指示单精度操作数，而关键字 QWORD PTR 则用来指示双精度操作数。在这两种情况下，从存储器中读取的值都会被转换成 80 位的扩展双精度格式。例如，指令

```
FLD   DWORD PTR [EAX]
```

从存储单元 [EAX] 处读取一个单精度浮点数，将其转换成 80 位的格式，并压入到寄存器堆栈中。

FST 指令把 ST(0) 中的浮点数写入到存储器中，在这种情况下，指向寄存器堆栈栈顶的指针不受影响。需要使用适当的关键字 DWORD PTR 或 QWORD PTR 来指明要写入存储器的值的大小，这决定了 ST(0) 中的 80 位表示形式是转换成单精度格式还是双精度格式。例如，指令

```
FST   QWORD PTR [EDX-8]
```

将 ST(0) 中的 80 位浮点数转换成 64 位的双精度格式，并将转换后的值写入到存储单元 [EDX]-8 中。

IA-32 体系结构还提供了加载和存储 32 位整数操作数的指令。这些指令可将整数自动转换成 80 位的浮点格式或将 80 位的浮点格式转换成整数。FILD 指令从存储器中读取一个 32 位的整数，将其转换成一个 80 位的浮点数之后再压入到寄存器堆栈中。FIST 指令将寄存器

ST(0) 中的 80 位浮点数转换成一个 32 位的整数，然后再将其写入到存储器中。这不会改变指向寄存器堆栈栈顶的指针。

最后，IA-32 体系结构中还提供了存储并弹出指令，这些指令在将寄存器 ST(0) 中的内容（经过适当的转换）写入到存储器中后会将其从栈中弹出。这意味着先前的寄存器 ST(0) 变成 ST(7)，ST(1) 变成 ST(0)，依此类推。FSTP 指令执行的操作与 FST 指令一样，但是它还将 ST(0) 从栈中弹出。同样，FISTP 指令也结合了 FIST 指令的操作和弹出操作。

## E.9.2 算术指令

对浮点数执行算术运算的基本指令有 FADD、FSUB、FMUL 以及 FDIV。对于需要显式指明一个操作数的指令，这个操作数必须是源操作数，并且必须在存储器中。目的操作数被隐含地保存在寄存器 ST(0) 中。来自存储器的操作数在执行算术运算之前被转换成 80 位的扩展双精度格式。例如，指令

```
FADD    QWORD PTR [EAX]
```

从存储单元 [EAX] 中读取 64 位的双精度数，将其转换成 80 位的扩展双精度表示形式，然后将转换后的值与寄存器 ST(0) 中的当前值相加，并将相加和放到 ST(0) 中。

对于需要显式指明两个操作数的指令，这两个操作数都必须在寄存器中，其中一个寄存器还必须是 ST(0)。例如，指令

```
FMUL    ST(0), ST(3)
```

将 ST(0) 和 ST(3) 中的值相乘，并将乘积放到 ST(0) 中。

指令 FADDP、FSUBP、FMULP 以及 FDIVP 在执行完算术运算后会将寄存器堆栈的栈顶元素弹出。这些指令都必须指明保存在寄存器中的两个操作数。使用 ST(1) 作为目的单元是比较适当的。弹出操作完成后，结果就在寄存器堆栈新的栈顶寄存器中。例如，指令

```
FSUBP   ST(1), ST(0)
```

从 ST(1) 中减去 ST(0) 的值，并将结果放到 ST(1) 中，然后弹出 ST(0)。因此，现在结果位于堆栈的栈顶 ST(0) 中。

指令 FIADD、FISUB、FIMUL 以及 FIDIV 显式指明了存储器中的一个 32 位操作数。目的操作数则隐含地保存在寄存器 ST(0) 中。在执行算术运算之前，这个 32 位的整数被自动转换成 80 位的扩展双精度表示形式。例如，指令

```
FIDIV   DWORD PTR [ECX + 4]
```

从存储单元 [ECX]+4 中读取一个 32 位的整数，将其转换成 80 位的浮点表示形式，然后再将转换后的值除以 ST(0) 中的值，并把运算结果放入 ST(0) 中。

对于减法和除法运算，操作数的顺序是非常重要的。因为浮点寄存器被组织成一个堆栈，它有时可能需要将这些算术运算的操作数的顺序颠倒过来，以便于随后可从堆栈中弹出一个特殊的结果或操作数。指令 FSUBR 和 FDIVR 就是可实现此目的的指令。例如，指令

```
FSUBR   ST(3), ST(0)
```

执行操作 ST(3) ← [ST(0)] – [ST(3)]。而指令

```
FSUB    ST(3), ST(0)
```

则执行操作 ST(3) ← [ST(3)] – [ST(0)]。

## E.9.3 比较指令

浮点比较指令可以用来设置状态寄存器中的条件码标志。这样表 E-2 中的条件跳转指令就

可以用来检测不同的条件了。FUCOMI 和 FUCOMIP 指令可对寄存器中的两个浮点操作数进行比较。目的操作数必须在 ST(0) 中。执行完比较操作后，FUCOMIP 指令同样也会弹出 ST(0)。例如，指令

FUCOMI ST(0), ST(4)

执行减法 [ST(0)] − [ST(4)]，并根据结果来设置状态寄存器中的 ZF 和 CF 标志。然后将结果丢弃。这样，随后的 Jump 指令就可以检测适当的条件了：ST(0) = ST(4) 时为 JE，ST(0) >ST(4) 时为 JA，ST(0) < ST(4) 时为 JB。

### E.9.4 其他指令

IA-32 还提供了一些其他的指令，如求平方根（FSQRT）、更改符号（FCHS）、求绝对值（FABS）、求正弦（FSIN）和余弦（FCOS）的指令。这些指令都不需要显式的操作数，因为对于源操作数和目的操作数，它们都隐含地使用了 ST(0)。换句话说，寄存器堆栈栈顶寄存器中的值被运算结果替代，且指向栈顶的指针没有改变。

IA-32 还提供了一些指令，可用来将常用的浮点常量压栈。FLDZ 指令可将 0.0 压入寄存器堆栈中，FLD1 指令可将 1.0 压入寄存器堆栈中。FLDPI 指令可将 $\pi$ 的扩展双精度浮点表示（精确到小数点后 19 位）压入寄存器堆栈中。这些指令都不需要显式地指明操作数。

### E.9.5 浮点程序示例

图 E-18 给出了一个示例程序。假设有两个点 $(x_0, y_0)$ 和 $(x_1, y_1)$。该程序确定经过这两个点的直线的斜率 $m$ 和截距 $b$，当这两个点位于一条垂直线上时除外。

```
        MOV     EAX, OFFSET COORDS      EAX 指向坐标列表
        FLD     QWORD PTR [EAX + 24]    将 y1 压入寄存器堆栈
        FLD     QWORD PTR [EAX + 16]    将 x1 压入寄存器堆栈
        FLD     QWORD PTR [EAX + 8]     将 y0 压入寄存器堆栈
        FLD     QWORD PTR [EAX]         将 x0 压入寄存器堆栈
        FSUBP   ST(2), ST(0)            计算 x1 − x0；弹出 x0
        FLDZ                            将 0.0 压栈
        FUCOMIP ST(0), ST(2)            确定分母是否为 0
        JE      NO_SLOPE                如果分母为 0，则斜率 m 是不确定的
        FSUBP   ST(2), ST(0)            计算 y1 − y0；弹出 y1
        FDIVP   ST(1), ST(0)            计算 m = (y1 − y0) / (x1 − x0)
        MOV     EBX, OFFSET SLOPE       EBX 指向存储单元 SLOPE
        FST     QWORD PTR [EBX]         将斜率保存到存储器中
        FLD     QWORD PTR [EAX + 8]     将 y0 压入寄存器堆栈
        FLD     QWORD PTR [EAX]         将 x0 压入寄存器堆栈
        FMULP   ST(2), ST(0)            计算 m · x0；弹出 x0
        FSUBRP  ST(1), ST(0)            计算 b = y0 − m · x0；弹出 y0
        MOV     EBX, OFFSET INTERCEPT   EBX 指向存储单元 INTERCEPT

        FSTP    QWORD PTR [EBX]         将截距保存到存储器中；弹出栈顶元素

        MOV     EBX, 0                  表明直线不是垂直的
        JMP     DONE
NO_SLOPE: MOV   EBX, 1                  表明直线是垂直的
DONE:   MOV     DWORD PTR VERT_LINE, EBX
```

图 E-18 计算直线斜率与截距的浮点程序

假定两点的坐标 $x_0$、$y_0$、$x_1$ 和 $y_1$ 为 64 位的双精度浮点数，存储在从 COORDS 单元开始的连续存储单元中。该程序把计算得到的斜率和截距作为双精度数放在存储单元 SLOPE 和

INTERCEPT 中。直线的斜率由式子 $m = (y_1 - y_0) / (x_1 - x_0)$ 确定，因此，该程序在执行除法运算之前必须先检查分母是否为零。当分母为零时，两点在一条垂直线上。该程序就将值 1 写到存储单元 VERT_LINE 中以反映这种情况，并指出存储单元 SLOPE 和 INTERCEPT 中的值是无效的，不做进一步的计算。否则，该程序计算出斜率值，并将其保存到存储器中。对于有效的斜率，程序计算出截距 $b = y_0 - m \cdot x_0$，然后将其写到存储器中。在此例中，减法运算使用了 FSUBR 指令，该指令将操作数的顺序颠倒过来。最后，程序将零写到存储单元 VERT_LINE 中来表明斜率和截距是有效的。

## E.10 多媒体扩展（MMX）操作

一个二维图形或视频图像可以用一个包含采样图像点的大矩阵来表示，这些采样图像点被称为像素（pixel）。每个点的颜色和亮度可以被编码成一个 8 位的数据项。这样的数据处理有两个主要特点。第一个特点是单个像素的操作通常只包括非常简单的算术或逻辑运算。另一个特点是一些实时的显示应用需要很高的计算性能。而对于采样的音频信号或语音处理（在一定的时间间隔内对连续的模拟信号进行采样并用一系列有符号数来表示）来说，也具有这些特点。

在这样的应用中，如果单个数据项是字节或 16 位的字，并能打包成其元素可以并行处理的小组，那么就可以获得很高的处理效率。这种能并行处理的向量或单指令多数据（SIMD）指令在第 12 章中进行了描述。IA-32 指令集中包含了很多 SIMD 指令，这些指令被称为多媒体扩展（MultiMedia eXtension，MMX）指令。它们能同时对多个数据元素（被打包成 64 位的四字）执行相同的操作。MMX 指令的操作数可以在存储器或八个浮点寄存器中，这样，这些寄存器具有双重作用。它们可以保存浮点数或 MMX 操作数。当被 MMX 指令使用时，这些寄存器作为 MM0 到 MM7 来引用，并且每个 80 位寄存器中只有最低的 64 位是与 MMX 操作相关的。与 E.9 节中的浮点指令不同的是，MMX 指令不将这个共享的寄存器组作为堆栈来管理。

694 ~ 695

IA-32 体系结构提供了 MOVQ 指令，用于在存储器和 MMX 寄存器之间传送 64 位的四字操作数。例如，指令

```
MOVQ   MM0, [EAX]
```

将寄存器 EAX 所指向的存储单元中的四字装入寄存器 MM0 中。MOVQ 指令还可以用来在 MMX 寄存器之间传送数据。例如，指令

```
MOVQ   MM3, MM4
```

将寄存器 MM4 中的内容传送到寄存器 MM3 中。

IA-32 体系结构中还提供了一些指令，可以对打包的四字操作数（packed quadword operand）中的多个元素并行执行算术和逻辑运算。源操作数可以在存储器或一个 MMX 寄存器中，但是目的地必须是一个 MMX 寄存器。大多数的 MMX 指令使用一个后缀来指示打包的四字操作数中数据元素的大小（和个数）：B 表示字节（8 个元素），W 表示字（4 个元素），D 表示双字（2 的元素），Q 表示四字（1 个元素）。例如，指令

```
PADDB   MM2, [EBX]
```

将寄存器 MM2 中和寄存器 EBX 所指向的存储单元中的八个相应字节相加。八个相加和并行计算，结果放在寄存器 MM2 中。

IA-32 体系结构中还提供了减法指令（PSUB）、乘法指令（PMUL）、将乘法和加法合并的指令（PMADD）、逻辑运算指令（PAND、POR 和 PXOR），以及可以对打包的四字操作数进行很多其他操作的指令。

## E.11 向量（SIMD）浮点运算

E.9 节描述了对单个浮点数进行运算的指令。IA-32 体系结构中还提供了向量（SIMD）指令来同时对多个浮点数进行运算。在 Intel 术语中，这些指令被称为流式 SIMD 扩展（Streaming SIMD Extension，SSE）指令。它们处理的是打包的 128 位双四字操作数，每一个操作数由四个 32 位的浮点数组成。可使用八个额外的 128 位寄存器 XMM0 到 XMM7 来保存这些操作数。

MOVAPS 和 MOVUPS 指令可在存储器和 XMM 寄存器之间或者在 XMM 寄存器之间传送
696 打包的双四字。PS 后缀表示双四字中的打包单精度浮点值。A 或 U 标号用来确定存储器地址是否必须与 16 位字的边界对齐或者可以不对齐。指令

MOVUPS XMM3, [EAX]

将 128 位的双四字从寄存器 EAX 所指向的存储单元装入寄存器 XMM3 中。指令

MOVUPS XMM4, XMM5

将寄存器 XMM5 中的双四字传输到寄存器 XMM4 中。

还可对两个双四字操作数中的四对 32 位浮点数同时执行基本的算术运算。源操作数可以在存储器或一个 XMM 寄存器中，但是目的地必须是一个 XMM 寄存器。这样的指令有 ADDPS、SUBPS、MULPS 和 DIVPS。例如，指令

ADDPS XMM0, XMM1

将寄存器 XMM0 和 XMM1 中四对相应的浮点数相加，并将四个相加和放到寄存器 XMM0 中。

## E.12 问题解析

本节将介绍一些可能要求学生解决的典型问题，并分析说明如何解决这样的问题。

**例 E.5**

**问题**：假设有一个 ASCII 编码的字符串保存在存储器中，起始地址为 STRING。该字符串以回车符（CR）结束。写一个 IA-32 程序来确定该字符串的长度。

**解答**：图 E-19 给出了一个可能的程序。字符串中的每个字符与 CR（ASCII 码为 0D）进行比较。计数器递增，直至到达字符串的末尾。结果存储在单元 LENGTH 中。

```
        MOV   EAX, OFFSET STRING        EAX 指向字符串的开始
        MOV   EDI, 0                    EDI 是计数器，被清为 0
LOOP:   MOV   BL, BYTE PTR [EAX + EDI]  将下一个字符装入 EBX 的最低字节中
        CMP   BL, 0DH                   将字符与 CR 进行比较
        JE    DONE                      如果匹配，则结束
        INC   EDI                       递增计数器
        JMP   LOOP                      如果没有结束，则循环回到前面
DONE:   MOV   DWORD PTR LENGTH, EDI     将计数值存放到存储单元 LENGTH 中
```

697 图 E-19 例 E.5 的程序

**例 E.6**

**问题**：我们希望在一个 32 位的非负整数列表中找出最小的数。数据的存储地址从 1000 开始。该地址处的双字必须保存所找到的最小数。下一个双字包含列表中的项数 $n$，之后的 $n$ 个双字包含列表中的数。写一个程序，找出最小的数，并包含按照规定组织数据所需的汇编指示（assembler directive）。

**解答**：图 E-20 中的程序实现了所需的任务。该程序假设 $n \geq 1$，且列表中包含了一些样本数字。

```
LIST    EQU   3000              列表起始地址
        .CODE
        MOV   EAX, OFFSET LIST  EAX 指向列表的开始
        MOV   EDI, [EAX + 4]    EDI 是计数器，初始化为 n
        MOV   EBX, EAX          调整 EBX 的值后，EBX 指向第一个数
        ADD   EBX, 8
        MOV   ECX, [EBX]        ECX 保存目前为止所找到的最小的数
LOOP:   DEC   EDI               递减计数器
        JZ    DONE              如果 EDI 为 0，则结束
        MOV   EDX, [EBX]        获取下一个数
        ADD   EBX, 4            递增指针
        CMP   ECX, EDX          将下一个数与目前的最小数进行比较
        JLE   LOOP              如果下一个数不小于目前的最小数，则再次循环
        MOV   ECX, EDX          否则，更新目前为最小数
        JMP   LOOP              再次循环
DONE:   MOV   [EAX], ECX        将最小的数保存到 SMALL 中

        .DATA
        ORG   1000
SMALL   DD    0                 所找到的最小数的存储空间
N       DD    7                 列表中的项数
ENTRIES DD    4, 5, 3, 6, 1, 8, 2   列表中的项
```

图 E.20 例 E.6 的程序

**例 E.7**

**问题**：写一个程序，将一个 $n$ 位十进制整数转换成二进制数。如果数字是通过键盘输入的，那么十进制数是以 $n$ 个 ASCII 编码字符的形式给出的。 698

**解答**：考虑一个 4 位的十进制数 $D$，由数字 $d_3d_2d_1d_0$ 表示。该数的值为 $((d_3 \times 10 + d_2) \times 10 + d_1) \times 10 + d_0$。该数的这种表示是图 E-21 中程序所使用转换技术的基础。注意，每个 ASCII 编码字符先通过 AND 指令转换成一个二进制编码的十进制（BCD）数字，然后再用于计算。

```
        MOV   ECX, DWORD PTR N      ECX 是计数器，初始化为 n
        MOV   ESI, OFFSET DECIMAL   ESI 指向 ASCII 数字
        MOV   EBX, 0                EBX 用来保存二进制数
        MOV   EDI, 10               EDI 用来乘以 10
LOOP:   MOV   DL, [ESI]             获取下一个 ASCII 数字
        AND   EDX, 0FH              产生 BCD 数字
        ADD   EBX, EDX              加到中间结果中
        DEC   ECX                   递减计数器
        JZ    DONE                  如果结束，则退出循环
        IMUL  EBX, EDI              乘以 10
        INC   ESI                   递增指针
        JMP   LOOP                  如果未完成，则循环回到前面
DONE:   MOV   DWORD PTR BINARY, EBX 将结果存放到存储单元 BINARY 中
```

图 E-21 例 E.7 的程序

**例 E.8**

**问题**：考虑一个数字阵列 $A(i, j)$，其中 $i = 0$ 到 $n-1$，是行索引，$j = 0$ 到 $m-1$，是列索引。该阵列按行存储在计算机的存储器中，每行元素占用 $m$ 个连续的字单元。写一个子程序，将第 $x$ 列的元素逐一加到第 $y$ 列的元素上，和元素（sum element）放在第 $y$ 列上。索引值 $x$ 和 $y$ 通过寄存器 EAX 和 EBX 传递给子程序。参数 $n$ 和 $m$ 通过寄存器 ECX 和 EDX 传递给子程序，元素 A(0,0) 的地址通过寄存器 EDI 进行传递。

**解答**：图 E-22 给出了一个可能的程序。我们假定值 $x$、$y$、$n$ 和 $m$ 分别存放在存储单元 X、Y、N 和 M 中。此外，阵列中的元素存储在从单元 ARRAY 开始的连续的字中，ARRAY 是元素 A(0,0) 的地址。

程序中的注释说明了每条指令的作用

```
        MOV   EAX, DWORD PTR X      载入值 x
        MOV   EBX, DWORD PTR Y      载入值 y
        MOV   ECX, DWORD PTR N      载入值 n
        MOV   EDX, DWORD PTR M      载入值 m
        MOV   EDI, OFFSET ARRAY     载入 A(0,0) 的地址
        CALL  SUB
        下一条指令
        :

SUB:    PUSH  ESI                   保存寄存器 ESI 的内容
        SHL   EDX, 2                确定一列中连续元素之间的距
                                    离（以字节为单位）
        SUB   EBX, EAX              产生 y - x 的值
        SHL   EBX, 2                产生 4(y - x) 的值
        SHL   EAX, 2                产生 4x 的值
        ADD   EDI, EAX              EDI 指向 A(0,x)
        MOV   ESI, EDI
        ADD   ESI, EBX              ESI 指向 A(0,y)
LOOP:   MOV   EAX, [EDI]            获取第 x 列中的下一个数
        MOV   EBX, [ESI]            获取第 y 列中的下一个数
        ADD   EAX, EBX              将两数相加，并保存和
        MOV   [ESI], EAX
        ADD   EDI, EDX              递增第 x 列的指针
        ADD   ESI, EDX              递增第 y 列的指针
        DEC   ECX                   递减行计数器
        JG    LOOP                  如果未完成，则循环回去
        POP   ESI                   恢复寄存器 ESI 的内容
        RET                         返回到调用程序
```

图 E-22 例 E.8 的程序

**例 E.9**

**问题**：假设存储单元 BINARY 中包含一个 32 位的模式。我们希望在一个具有图 3-3 所示接口的显示设备上将这些位显示为 8 个十六进制数字。写一个程序完成这个任务。

**解答**：首先，我们需要将这个 32 位的模式转化为用 ASCII 编码字符表示的十六进制数字。转换过程可以使用查表法来完成，必须构造一个包含 16 项的表，以便于为每一个十六进制数字提供其 ASCII 码。然后，对于 BINARY 中该模式的每一个 4 位的段，都可以在表中查找到其相应的字符，并将这些字符存储在从地址 HEX 开始的连续字节单元中。最后，将从地址 HEX 开始的这 8 个字符发送给显示器。图 E-23 给出了一个可能的程序。

699
/
700

```
        .CODE
        MOV   EAX, DWORD PTR BINARY   载入该二进制数
        MOV   ECX, 8                  ECX 是计数器，被置为 8
        MOV   EDI, OFFSET HEX         EDI 指向十六进制数字
        MOV   ESI, OFFSET TABLE       ESI 指向 ASCII 码转换表
LOOP:   ROL   EAX, 4                  将高位数字循环移位到低位位置上
        MOV   EBX, EAX                将当前值复制到另一个寄存器中
        AND   EBX, 0FH                提取下一个数字
        MOV   DL, [ESI + EBX]         获取该数字的 ASCII 码
        MOV   [EDI], DL               将其存储在 HEX 缓冲区中
        INC   EDI                     递增数字指针
        DEC   ECX                     递减数字计数器
        JG    LOOP                    如果不是最后一个数字，则循环回到前面
```

图 E-23 例 E.9 的程序

```
DISPLAY: MOV   ECX, 8
         MOV   EDI, OFFSET HEX
         MOV   ESI, OFFSET DISP_DATA
DLOOP:   MOV   EDX, [ESI + 4]        通过测试 DOUT 标志来检查显示器是否准备就绪
         AND   EDX, 4
         JZ    DLOOP
         MOV   DL, [EDI]            获取下一个 ASCII 字符
         INC   EDI                  递增字符指针
         MOV   [ESI], DL            将其发送给显示器
         DEC   ECX                  递减计数器
         JG    DLOOP                循环，直到所有的字符都显示完毕
         下一条指令

         .DATA
         ORG   1000
HEX      DB    8 DUP (0)            ASCII 编码数字的存储空间
TABLE    DB    30H, 31H, 32H, 33H   ASCII 转换表
         DB    34H, 35H, 36H, 37H
         DB    38H, 39H, 41H, 42H
         DB    43H, 44H, 45H, 46H
```

图 E.23 （续）

701

## E.13 结束语

IA-32 指令集是应用非常广泛的 CISC 设计的一个例子。对不同类型的数据，如单个整数和浮点数，以及打包的整数和浮点数向量，IA-32 指令集都可以对其进行广泛的操作。尽管如此大的指令集有一定的挑战性，但 IA-32 指令集还是在一些高性能处理器中实现了。

## 习题

[E] E.1 写一个程序，计算表达式 SUM = 580 − 68 400 + 80 000。

[E] E.2 写一个程序，计算表达式 ANSWER = A × B − C × D。

[M] E.3 写一个程序，在一个含有 $n$ 个 32 位整数的列表中找出所包含的负整数的个数，并将该计数保存在单元 NEGNUM 中。$n$ 保存在存储单元 N 中，列表中的第一个整数保存在单元 NUMBERS 中。在程序中包含必要的汇编指示（assembler directive）和一个样本列表，列表中含有 6 个数字，其中一些是负数。

[E] E.4 为图 E-6 中的程序写一个如图 E-13 所示风格的汇编语言程序。假设采用图 2-10 所示的数据布局。

[M] E.5 写一个 IA-32 程序来解决第 2 章的习题 2.10 中的问题。

[E] E.6 写一个 IA-32 程序来解决第 2 章中例 2.5 所描述的问题。

[M] E.7 写一个 IA-32 程序来解决第 3 章中例 3.5 所描述的问题。

[E] E.8 写一个 IA-32 程序来解决第 3 章中例 3.6 所描述的问题。

[E] E.9 假设 TABLE 的地址是 0x10100，写一个 IA-32 程序来解决第 3 章中例 3.6 所描述的问题。

[E] E.10 写一个程序，在视频显示器的一行上以十六进制形式显示主存中 10 个字节的内容。该字节串在存储器中的起始位置是 LOC。每个字节将显示为两个十六进制字符。连续字节应以空格分隔。

[M] E.11 假设存储单元 BINARY 中包含一个 16 位的模式。我们希望在一个具有图 3-3 所示接口的显示设备上将这些位显示为 0 和 1 组成的字符串。写一个程序完成这个任务。

[M] E.12 使用图 3-17 中的七段显示器和图 3-14 中的定时器电路，写一个程序，显示序列 0、1、2、…、9、0、…中的十进制数字。每个数字显示一秒钟。假设定时器电路中的计数器是由 100 MHz 的

时钟驱动的。

[D] E.13 使用两个图 3-17 所示的七段显示器和图 3-14 所示的定时器电路，写一个程序，显示序列 0、1、2、…、98、99、0、…中的数，每个数显示一秒钟。假设定时器电路中的计数器是由 100 MHz 的时钟驱动的。

[D] E.14 写一个程序，计算实时时钟时间并以小时（0 ~ 23）和分钟（0 ~ 59）的形式显示时间。显示器包括 4 个图 3-17 所示的七段显示设备。还有一个具有图 3-14 所示接口的定时器电路，其计数器是由 100 MHz 的时钟驱动的。

[M] E.15 写一个 IA-32 程序来解决第 2 章的习题 2.22 中的问题。假设将要被压入/弹出的元素位于寄存器 EAX 中，并假设寄存器 EBX 用作用户栈的栈指针。

[M] E.16 写一个 IA-32 程序来解决第 2 章的习题 2.24 中的问题。

[M] E.17 写一个 IA-32 程序来解决第 2 章的习题 2.25 中的问题。

[M] E.18 写一个 IA-32 程序来解决第 2 章的习题 2.26 中的问题。

[M] E.19 写一个 IA-32 程序来解决第 2 章的习题 2.27 中的问题。

[M] E.20 写一个 IA-32 程序来解决第 2 章的习题 2.28 中的问题。

[M] E.21 写一个 IA-32 程序来解决第 2 章的习题 2.29 中的问题。

[M] E.22 写一个 IA-32 程序来解决第 2 章的习题 2.30 中的问题。

[M] E.23 写一个 IA-32 程序来解决第 2 章的习题 2.31 中的问题。

[D] E.24 写一个 IA-32 程序来解决第 2 章的习题 2.32 中的问题。

[D] E.25 写一个 IA-32 程序来解决第 2 章的习题 2.33 中的问题。

[M] E.26 写一个 IA-32 程序来解决第 3 章的习题 3.20 中的问题。

[M] E.27 写一个 IA-32 程序来解决第 3 章的习题 3.22 中的问题。

[D] E.28 写一个 IA-32 程序来解决第 3 章的习题 3.24 中的问题。

[D] E.29 写一个 IA-32 程序来解决第 3 章的习题 3.26 中的问题。

[D] E.30 当 $0 \le x \le \pi/2$ 时，函数 $\sin(x)$ 能以合理的精度近似为 $x - x^3/6 - x^5/120 = x(1 - x^2(1/6 - x^2(1/120)))$。写一个子程序 SIN，接受一个输入参数（该参数是指向存储器中一个浮点值 $x$ 的指针），并使用上述只涉及 $x$ 和 $x^2$ 的第二个表达式来计算 $\sin(x)$ 的近似值。计算所得的值应返回到存放输入参数 $x$ 的存储单元中。子程序使用的任何寄存器都应根据需要进行保存和恢复。通过比较该子程序的运行结果和 IA-32 指令集中 FSIN 指令的执行结果，对该子程序中所采用的近似精度进行研究。

## 参考文献

1. Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*, Document number 253665-033US, December 2009. Available at http://www.intel.com.

# 索　引

索引中的页码为英文原书页码，与书中边栏页码一致。


A

A/D（模/数），参见 Analog-to-Digital conversion
Access time（访问时间）
  magnetic disk（磁盘），315
  main memory（主存储器），5，269
  effect of cache（高速缓存的影响），301
Actuators（执行器），410
Adder（加法器）
  BCD，378
  carry-lookahead（超前进位），340
  circuit（电路），337
  full adder（全加器），336
  half adder（半加器），377
  least-significant bit（最低有效位），336
  most-significant bit（最高有效位），339
  propagation delay（传播延迟），339
  ripple-carry（行波进位），336
Addition（加法），11，13，336
  carry（进位）11，336
  carry-save（进位保留），353
  end-around carry（循环进位），382
  floating-point（浮点），267
  generate function（生成函数），340
  modular（模），12
  overflow（溢出），14，15，337
  propagate function（传播函数），340
  sum（和），11，336
Address（地址），5，29
  aligned/unaligned（对齐的/不对齐的），31
  big-endian（大端），30
  little-endian（小端），30
Address pointer（地址指针），42
Address space（地址空间），29，268
Address translation（地址转换），306
Addressing mode（寻址方式），35，40，75
  absolute（绝对），41
  autodecrement（自动减量），76
  autoincrement（自动增量），75
  immediate（立即），42
  index（变址），45，157
  indirect（间接），42
  register（寄存器），41
  relative（相对），76
  Nios II，532
  ColdFire，575
  ARM，614
  IA-32，665
Alarm clock（闹钟），428
Alphanumeric characters（字母数字字符），17
  ASCII，18
Amdahl's law（Amdahl 定律），461
Analog to digital（A/D）conversion（模/数转换），248，388
Arbiter（仲裁者），237
Arbitration（仲裁），237
Architecture（体系结构），2
Arithmetic and logic unit（ALU，算术逻辑部件），5，153，160
ARM processor（ARM 处理器），611
ASCII（American Standards Committee on Information Exchange）code（美国信息交换标准码，ASCII 码），17
Assembler（汇编程序），49，130
Assembly language（汇编语言），49
  directives（commands）（指示符（命令）），50
  generic（通用），28
  mnemonics（助记符），34，49
  notation（标记），33
  syntax（语法），49

Nios II，532，549
ColdFire，573，593
ARM，614，635
IA-32，670，685
Associative search（相联检索），294
Asynchronous DRAM（异步 DRAM），276
Asynchronous bus（异步总线），参见 Bus
Asynchronous transmission（异步传输），245

B

Bandwidth（带宽）：
interconnection network（互连网络），450
memory（存储器），279
Barrier synchronization（屏障同步），458
Base register（基址寄存器），48
BCD，参见 Binary-coded decimal
Big-endian（大端），30
Binary-coded decimal（BCD，二进制编码的十进制），17，54
addition（加法），378
Binary variable（二进制变量），466
Bit（位），4，28
Booth algorithm（Booth 算法），348
bit-pair recoding（位偶重编码），352
skipping over 1s（跳过 1 位），350
Boot-strapping（引导），144
Bounce（反弹），参见 Contact bounce
Branch（转移）
delay slot（延迟槽），204
delayed（延迟转移），205
instruction（转移指令），38，77，168
offset（偏移量），54
penalty（转移代价），202
prediction（转移预测），205
target（转移目标），38
target buffer（转移目标缓冲区），208
Branching（转移），37，77
Breakpoint（断点），136
Bridge（桥），252
Broadcast（广播），453
Bus（总线），179，450
arbitration（仲裁），237
asynchronous（异步），233
driver（驱动器），179，236
master（主控），230，237
propagation delay（传播延迟），231
protocol（协议），229
skew（相位偏移），234
slave（从动设备），230
synchronous（同步），230
timing（时序），231
Bus standards（总线标准）
ISA，258
FireWire（火线），251
PATA，258
PCI，252
PCI Express，258
SCSI，256
SAS，258
SATA，258
USB，247
Bus structure（总线结构），228
Byte（字节），17，29
Byte addressable memory（按字节寻址存储器），30

C

Cache memory（高速缓冲存储器），5，269，289
associative mapping（相联映射），293，299
block（块），290
coherence（一致性），296，453
directory-based（基于目录），456
snooping（监听），454
direct mapping（直接映射），298
dirty bit（脏位），290
hit（命中），290
hit rate（命中率），301
levels（层次），289
line（行），290
load through（直接装入），291
lockup-free（无锁定），305
mapping function（映射功能），290，291
miss（失效），291
miss penalty（失效开销），301
miss rate（失效率），301
prefetching（预取），304
replacement algorithm（替换算法），290，296
set-associative mapping（组相联映射），294，299

stale data（过时数据），294
tag（标志），293
valid bit（有效位），294
write-back（写回），290，294，454
write buffer（写缓冲区），303
write-through（直接写），290，453
Carry（进位），11，336
detection（检测），551
Cartridge tape（盒式磁带），323
CD-ROM（只读光盘），318
Character codes（字符码），17
ASCII，18
Charge-coupled device（CCD，电荷耦合器件），387
Chip（芯片），17
CISC（Complex Instruction Set Computer，复杂指令集计算机），34，74，178，572，661
Circular buffer（queue）（环形缓冲区（队列）），92
Clock（时钟），230，493
rate（速率），209
period（周期），154
duty cycle（占空比），260
self-clocking（自同步时钟），313
Clock recovery（时钟复位），246，313
Cloud computing（云计算），5
Coherence（一致性），参见 Cache memory
ColdFire processor（ColdFire 处理器），571
Combinational circuits（组合电路），154，494
Comparator（比较器），170，186
Compiler（编译程序，编译器），133
optimizing（优化），134
vectorizing（向量化），446
Complement（补码），468
Complementary metal-oxide semiconductor（CMOS，互补金属氧化物半导体），484
Complex Instruction Set Computer（复杂指令集计算机），参见 CISC
Complex programmable logic device（CPLD，复杂可编程逻辑器件），513
Compute Unified Device Architecture（CUDA，统一计算设备体系结构），448
Computer types（计算机类型），2
Computer-aided design（CAD，计算机辅助设计），424
Condition codes（条件码），77
flag（标志），77
in pipelined processor（在流水线处理器中），218
side effect（副作用），218
ColdFire，572，599
ARM，629
IA-32，663
Condition code register（条件码寄存器），77
Conditional branch（条件转移），38，77，169
Configuration device（配置器件），423
Contact bounce（接触反弹），239
Context switch（上下文切换），148
Control signals（控制信号），172
Control store（控制存储器），183
Control unit（控制单元），6
Control word（控制字），183
Counter（计数器）
ripple（行波），504
synchronous（同步），504，516
Crossbar（交叉开关），452

D

D/A（数/模），参见 Digital-to-analog conversion
Data（数据），4
Data dependency（数据依赖性），197，457
Data striping（数据条带化），317
Data types（数据类型）
bit（位），4
byte（字节），17
character（字符），17
floating-point（浮点），16，363
integer（整型，整数），10
fraction（小数），365，372
string（串），81
word（字），5，29
Datapath（数据通路），161
Deadlock（死锁），217
Debouncing（消除反弹），239
Debugger（调试器），134
Debugging（调试），54，117
Decoder（译码器），505
instruction decoder（指令译码器），176
De Morgan's rule（德摩根定律），475
Desktop computer（台式机），2
Device driver（设备驱动程序），148

Device interface（设备接口），97
Differential signaling（差分信号），251，256，258，279
Digital camera（数码照相机），387
Digital-to-analog conversion（数/模转换），248
Direct memory access（直接存储器访问），参见 DMA
Directory-based cache coherence（基于目录的高速缓存一致性），456
Dirty bit（脏位），参见 Cache memory
Disk（盘），参见 Magnetic disk
Disk arrays（磁盘阵列），317
Dispatch unit（调度部件），212
Display（显示器），6，97
Division（除法），360
  floating-point（浮点），368
  non-restoring（不恢复），361
  restoring（恢复），360
DMA（Direct Memory Access，直接存储器访问），285，306
  controller（控制器），286
Don't care condition（无关项条件），477
DVD（数字多功能光盘），5，321
Dynamic memory（DRAM，动态存储器），274

E

Echoback（回显），101
Edge-triggered flip-flop（边沿触发触发器），498
Effective address（有效地址），42
Effective throughput（有效吞吐量），450
Embedded computer（嵌入式计算机），2
Embedded system（嵌入式系统），2，386，422，530，651
Enterprise system（企业系统），2
Error correcting code（ECC，纠错码），316
Ethernet（以太网），252
Exception（异常），116，556，639，687，参见 Interrupt
  floating-point（浮点），367
  handler（处理程序），558
  imprecise（不精确），216
  precise（精确的），216
Execution phase（执行阶段），37，153
Execution steps（执行步骤），155，185
Execution step counter（执行步计数器），175
Execution step timing（执行步时序），171，178
External name（外部名），132

F

Fan-in（扇入），490
Fan-out（扇出），490
Fetch phase（取指令阶段），37，153
Field-programmable gate array（FPGA，现场可编程门阵列），21，423，514
FIFO（first-in，first-out）queue（先进先出队列），92
File（文件），130，322
Finite state machine（有限状态机），520
FireWire（火线），251
Fixed point（定点），16
Flash memory（闪存），284
Flash cards（闪存卡），284
Flash drives（闪存驱动器），284
Flip-flops（触发器），496
  D（D 触发器），495
  edge-triggered（边沿触发），498
  gated latch（门控锁存器），493
  JK（JK 触发器），499
  latch（锁存器），492
  master-slave（主从），495
  SR latch（SR 锁存器），494
  T（T 触发器），498
Floating point（浮点），16，363
  addition-subtraction unit（加法/减法部件），369
  arithmetic operation（算术运算），367
  double precision（双精度），365
  exception（异常），367
    inexact（不精确），367
    invalid（非法），367
  exponent（指数），16，364
    excess-$x$ representation（余 $x$ 表示），365
  format（格式），364
  guard bits（保护位），368
    sticky bit（粘着位），369
  IEEE standard（IEEE 标准），16，364
  mantissa（尾数），365
  normalization（规格化），365
  overflow（溢出），366

representation（表示），364
scale factor（比例因子），16，364
base（基数），16，364
exponent（指数），16，364
significant digits（有效数字），364
single precision（单精度），364
special values（特殊值），366
denormal（非规格化），366
gradual underflow（逐级下溢），366
infinity（无穷大），366
Not a Number（NaN，非数，无定义数），367
truncation（截取），368
biased/unbiased（有偏/无偏），368
chopping（截断），368
rounding（舍入），368
von Neumann（冯·诺依曼），368
underflow（下溢），366
Nios II，562
ColdFire，599
ARM，650
IA-32，690，696
Floppy disk（软盘），316

## G

Gated latch（门控锁存器），493
General-purpose register（通用寄存器），8
GPU（Graphics Processing Unit）（图形处理单元），448
Gray code（Gray 码），524
Grid computer（网格计算机），2

## H

Handshake（握手），233
full（完全握手），235
interlocked（互锁），235
Hardware（硬件），2
Hardware interrupt（硬件中断），103
Hardwired control（硬件控制），175
Hazard（冲突），197
branch delay（转移延迟），202
data dependency（数据依赖性），197
memory delay（存储器延迟），201
resource limitation（资源限制），209
Hexadecimal（十六进制），参见 Number representation
High-level language（高级语言），17，20，133，137，399，432，456
History of computers（计算机历史），19
Hit（命中），参见 Cache memory
Hold time（保持时间），231，498
Hot-pluggable（可热插拔的），249

## I

IA-32 processor（IA-32 处理器），661
IEEE standards（IEEE 标准），参见 Standards
Immediate operand（立即操作数），164
Index register（变址寄存器），45
Ink jet printer（喷墨打印机），6
Input/output（I/O，输入/输出）
address space（地址空间），97
device interface（设备接口），97，229，238
in operation system（在操作系统中），146
interrupt-driven（中断驱动），103，556，598，648，689
memory-mapped（存储器映射），97，229
port（端口），239
privilege level（优先级），688
program-controlled（程序控制），97，556，597，646，689
status flag（状态标志），98
register（寄存器），97
unit（设备，部件），4，6
Nios II，555
ColdFire，597
ARM，646
IA-32，689
Input unit（输入设备），4
Instruction（指令），4，7，32
commitment（提交），216
completion queue（完成队列），216
execution phase（执行阶段），37，153
dispatch（调度），217
fetch phase（取指令阶段），37，153
queue（队列），212
reordering（重新排序），204
retired（释放），217
side effects（副作用），218

Instruction encoding（指令编码），82，168
Instruction execution（指令执行），155，165
Instruction fetch（指令提取），164
Instruction format（指令格式）：
generic（通用的），84
one-address（单地址），670
three-address（三地址），35
two-address（两地址），74
Nios II，533
ColdFire，573
ARM，615
IA-32，670
Instruction register（IR，指令寄存器），7，37，152
Instruction set architecture（ISA，指令集体系结构），28
Instructions（指令）：
arithmetic（算术），7，536，578，622，672
branch（转移），38，538，582，628，674
control（控制），111，548，596
data transfer（数据传送），7，534，577，621，671
input/output（输入/输出），535
logic（逻辑），67，537，585，626，677
move（移动），43，537，577，625，671
multimedia（多媒体），695
shift and rotate（移位和循环移位），68，546，586，625，678
subroutine（子程序），56，541，587，631，679
vector（SIMD，向量），445，696
Integrated circuit（IC，集成电路），21
Intellectual property（IP，知识产权），422
Interconnection network（互连网络），3，450
bandwidth（带宽），450
bus（总线），179，228，450
split-transaction（分割事务），450
computer system（计算机系统），252，258
crossbar（交叉开关），452
effective throughput（有效吞吐量），450
mesh（网状），452
packet（数据包），450
ring（环状），451
torus（圆环形），453
tree（树状），249
Interface（接口）：
input（输入），239
output（输出），242
parallel（并行），239，392
serial（串行），242，395
Internet（国际互联网），3，4
Interrupt（中断），9，103，参见 Exception
acknowledge（确认），105
disabling（禁止），106
enabling（允许），106，109
execution steps（执行步），185
handler（处理程序），116
hardware（硬件），103
in operating systems（在操作系统中），146
latency（等待），105
nesting（嵌套），108
nonmaskable（不可屏蔽），596，687
priority（优先级），109
service routine（服务程序），9，104
software（软件），146
vectored（向量），108，139
Nios II，556
ColdFire，596
ARM，639
IA-32，687
Invalidation protocol（无效协议），453
I/O，参见 Input/Output
IP core（IP 内核），530
IR（instruction register，指令寄存器），7
Isochronous（等时），248，249，251

J

Joystick（操作杆），4
JTAG port（JTAG 端口），514

K

Karnaugh map（卡诺图），475
Keyboard（键盘），4，97
interface（接口），99，239

L

Laser printer（激光打印机），6
Latch（锁存器），492
Library（库），133

LIFO（last-in，first-out）queue（后进先出队列），55
Link register（链接寄存器），57
Linker（连接程序），132
Little-endian（小端），30
Load through（直接装入）参见 Cache memory
Load-store multiple operands（加载－存储多操作数），62
ColdFire，588
ARM，621
IA-32，679
Loader（装载程序），54，131
Locality of reference（引用局部性），289，296
Logic circuits（逻辑电路），466
Logic function（逻辑函数），466
AND（与），468
Exclusive-OR（XOR，异或），468
minimization（最小化），472
NAND（与非），479
NOR（或非），479
NOT（非），468
OR（或），466
synthesis（组合），470，508
Logic gates（逻辑门），469
fan-in（扇入），490
fan-out（扇出），490
noise margin（噪声容限），489
propagation delay（传播延迟），489
threshold（阈值），482
transfer characteristic（传输特性），488
transition time（转换时间），490
Logical memory address（逻辑存储器地址），305
LRU（least-recently used）replacement（最近最少使用替换），296

## M

Machine instruction（机器指令），4
Machine language（机器语言），130
Magnetic disk（磁盘），5，311
access time（访问时间，存取时间），315
communication with（与……通信），257
controller（控制器），313，315
cylinder（柱面），314
data buffer/cache（数据缓冲区/高速缓存），315
data encoding（数据编码），313
drive（驱动器），313
floppy disk（软盘），316
logical partition（逻辑分区），314
rotational delay（旋转延迟），315
sector（扇区），314
seek time（寻道时间），315
track（磁道），314
Winchester（温切斯特），313
Magnetic tape（磁带），322
cartridge（盒式磁带），323
Manchester encoding（曼彻斯特编码），313
Master-ready（主控就绪），234
Master-slave（主从），参见 Flip-flop
Mechanical computing devices（机械计算设备），20
Memory（存储器），4
access time（访问时间），5，269
address（地址），5
address space（地址空间），29
asynchronous DRAM（异步 DRAM），276
bandwidth（带宽），279
bit line（位线），270
byte-addressable（按字节寻址），30
cache（高速缓存），参见 Cache memory
cell（单元），271，273，274，283
controller（控制器），281
cycle time（周期时间），269
DDR SDRAM，279
delay（延迟），171
DIMM，参见 Memory module
dual-ported（双端口），158
dynamic（DRAM，动态 RAM），274
fast page mode（快速页模式），276
flash（闪存），5
hierarchy（层次结构），288
latency（延迟），278
main（主要的），4
module（模块），281
multiple module（多模块），449
primary（主要的），4
Rambus，279

random-access memory（RAM，随机访问存储器），5，269，270
read cycle（读周期），273，274
read-only memory（ROM，只读存储器），282
refreshing（刷新），274，282
secondary（辅助的），5
SIMM，参见 Memory module
static（SRAM，静态存储器），271
synchronous DRAM（SDRAM，同步动态随机存储器），276
unit（部件），4
virtual（虚拟），参见 Virtual memory
word（字），5
word length（字长），5
word line（字线），270
write cycle（写周期），273，274
Memory Function Completed signal（存储器功能完成信号），171
Memory management unit（MMU，存储器管理部件），306
Memory pages（存储器页），306
Memory-mapped I/O（存储器映射 I/O），97，229
Memory segmentation（存储器分段），662
Mesh network（网状网络），452
Message passing（消息传递），19，456
Microcontroller（微控制器），386，390
ARM，391，414
Freescale，391，413
Intel，391，413
Microinstruction（微指令），183
Microprocessor（微处理器），21
Microprogram（微程序），183
Microprogram counter（微程序计数器），184
Microprogram memory（微程序存储器），183
Microprogrammed control（微程序控制），183
Microroutine（微例程），183
Microwave oven（微波炉），386
Miss（失效），参见 Cache memory
MMU，参见 Memory management unit
Mnemonic（助记符），34，49
Mouse（鼠标），4
Multicomputer（多计算机），19，456
Multicore processor（多核处理器），19，457
Multiple issue（多发操作），212
Multiple-precision arithmetic（多精度运算），77，336，552，572，623，681
Multiplexer（多路复用器），507
Multiplication（乘法），344
array implementation（阵列实现），344
Booth algorithm（Booth 算法），348
carry-save addition（进位保留加法），353
CSA tree（CSA 树），355
3-2 reducer（3-2 简化器），355
4-2 reducer（4-2 简化器），357
7-3 reducer（7-3 简化器），380
floating-point（浮点），367
sequential implementation（顺序实现），346
signed-operand（有符号操作数），346
Multiprocessor（多处理器），19，448
cache coherence（高速缓存一致性），453
interconnection network（互连网络），449
local memory（局部存储器），449
non-uniform memory access（NUMA，非统一存储器访问），449
program parallelism（程序并行性），456
shared memory（共享存储器），19，448
shared variables（共享变量），448，457
speedup（加速），461
uniform memory access（UMA，统一存储器访问），449
Multiprogramming（多道程序），145
Multistage hardware（多段硬件），155，162
Multitasking（多任务），145
Multithreading（多线程），444


## N


Nios II processor（Nios II 处理器），529
Noise（噪声），251，482，489
Noise margin（噪声容限），489
Notebook computer（笔记本电脑），2
Number conversion（数字转换），23，372
Number representation（数的表示），9
binary positional notation（二进制按位计数法），9
fixed-point（定点），16
floating-point（浮点），364
hexadecimal（十六进制），54

1's-complement（反码），10
sign-and-magnitude（原码），10
signed integer（有符号整数），10
ternary（三进制），378
2's-complement（补码），10
unsigned integer（无符号整数），10

O

Object program（目标程序），49，130
On-chip memory（片上存储器），425
1's-complement representation（反码表示），10
OP code（操作码），49
Operands（操作数），4
Operand forwarding（操作数转发），198
Operating system（操作系统），143
interrupts（中断），146
multitasking（多任务），146
process（进程），148，444
scheduling（调度），148
Optical disk（光盘），5，317
CD-Recordable（CD-R）（可刻录 CD），320
CD-Rewritable（CD-RW）（可擦写 CD），321
CD-ROM（只读光盘），318
DVD（数字多功能光盘），5，321
Out-of-order execution（无序执行），215
Output unit（输出设备），6
Overflow（溢出），14，15
detection（检测），551

P

Page（页），参见 Virtual memory
Page fault（页故障），309
Parallel I/O interface（并行 I/O 接口），425
Parallel processing（并行处理），444
Parallel programming（并行编程），456
Parallelism（并行化），17
instruction-level（指令级），17
processor-level（处理器级），17
Parameter passing（参数传递），59
by value（按值），62
by reference（按地址），62
PC，参见 Program Counter
PCI，参见 Bus standards
PCI Express，参见 Bus standards
Peer-to peer（点对点），251
Performance（性能），17
basic equation（基本公式），209
memory（存储器），300
modeling（建模），460
pipeline（流水线），209
Personal computer（个人计算机），2
Phase encoding（相位编码），313
Physical memory address（物理存储器地址），305
Pin assignment（引脚分配），431
Pipelining（流水线），19，194
bubbles（气泡），198
hazards（冲突），197
performance（性能），209
stalling（暂停），197
ColdFire，219
IA-32，219
Pixel（像素），388
Plug-and-play（即插即用），249，253
Pointer register（指针寄存器），42
Polling（轮询），98
Pop operation（出栈操作），参见 Stack
Portable computer（便携式计算机），2
POSIX threads（Pthreads）library（POSIX 线程库），460
Prefetching（预取），304
Primary storage（主存储器），4
Printer（打印机），6
Priority（优先级），237，参见 Arbitration
Privileged instruction（特权指令），311
Process（进程），参见 Operation system
Processor（处理器），4，152
core（核，核心），422
multicore（多核），19
Processor stack（处理器堆栈），55
Processor status register（处理器状态寄存器），106
Nios II，554
ColdFire，572
ARM，613
IA-32，663
Processor register（处理器寄存器），8

Program（程序），4
Program-controlled I/O（程序控制 I/O），参见 Input/Output
Program counter（PC，程序计数器），7，37，152，178
Program state（程序状态），148
Programmable array logic（PAL，可编程阵列逻辑），511
Programmable logic array（PLA，可编程逻辑阵列），509
Propagation delay（传播延迟）：
  logic circuit（逻辑电路），489
  bus（总线），231
Protection（保护），311
Pseudoinstruction（伪指令），44，537，548，637
Push operation，进栈操作，参见 Stack

## Q

Quartus II software（Quartus II 软件），425
Queue（队列），55，92，参见 Instruction

## R

RAID disk systems（RAID 磁盘系统），317
Rambus memory（Rambus 存储器），279
Random-access memory（RAM，随机访问存储器），5，269
Reaction timer（反应定时器），401
Reactive system（反应系统），386，401
Read operation（读操作），8
Read-only memory（ROM，只读存储器），282
  electrically erasable（EEPROM，电可擦除可编程只读存储器），284
  erasable（EPROM，可擦除可编程只读存储器），284
  flash（闪存），284
  programmable（PROM，可编程只读存储器），283
Real-time processing（实时处理），106
Reduced Instruction Set Computer（精简指令集计算机），参见 RISC
Refreshing memories（刷新存储器），274，282
Register（寄存器），6，502
  access time（访问时间），6
  base（基址），48
  control（控制），110，553
  general-purpose（通用），8
  index（变址），45
  port（端口），158
  renaming（重命名），216
Register file（寄存器文件），158
Register transfer notation（RTN，寄存器传送标记），33
Reorder buffer（重排序缓冲器），216
Replacement algorithm（置换算法），296
Reservation station（保留站），215
Ring network（环状网络），451
RISC（Reduced Instruction Set Computer，精简指令集计算机），34，154，530，612
ROM，参见 Read-Only Memory
Rounding（舍入），参见 Floating point

## S

SAS，参见 Bus standards
SATA，参见 Bus standards
Scaler（定标器），503
Scheduling（调度），参见 Arbitration，Operating system
SCSI bus（SCSI 总线），参见 Bus standards
Secondary storage（辅助存储器），5
Sensors（传感器），407
Sequential circuits（时序电路），494，516
  finite state machine（有限状态机），520
  state assignment（状态分配），516
  state diagram（状态图），516
  state table（状态表），517
  synchronous（同步），520
Serial transmission（串行传输），243
Server（服务器），2
Setup time（建立时间），231，498
Seven-segment display（七段显示器），124，507
Shadow registers（影子寄存器），105
Shared memory（共享存储器），448
Shift register（移位寄存器），503
Side effects（副作用），参见 Instruction
Sign bit（符号位），10
Sign extension（符号扩展），15
Single-instruction multiple-data（SIMD）processing

（单指令流多数据流处理），445
Skew（相位偏移），参见 Bus
Slave-ready（从动就绪），233
Snoopy cache（监听高速缓存），455
Soft processor core（软处理器核），530
Software interrupts（软件中断），146
SOPC Builder，425
Source program（源程序），49
Speculative execution（推测执行），214
Speedup（加速），22，461
Split-transaction protocol（分割事务协议），450
SR latch（SR 锁存器），494
Stack（栈，堆栈），55
  frame（结构），63
  frame pointer（FP）（结构指针 FP），63
  in subroutines（在子程序中），60
  pointer（SP）（指针 SP），55
  pushdown（下推），55
  push and pop operations（进栈和出栈操作），55
Standards（标准），参见 Bus standards
  IEEE floating-point（IEEE 浮点），16，364
  IEEE-1149.1，416
Start-stop format（起止方式），245
State diagram（状态图），516
State table（状态表），517
Static memory（SRAM）（静态存储器），271
Status flag（状态标志），参见 Input/Output
Status register（状态寄存器），106
Stored program（存储程序），4
Subroutine（子程序），56，170，186
  linkage（链接），57
  nesting（嵌套），58，64
  parameter passing（参数传递），59
  Nios II，541
  ColdFire，587
  ARM，631
  IA-32，679
Subtraction（减法），13
  floating-point（浮点），367
Sum-of-products form（积之和形式），470
Supercomputer（超级计算机），2
Superscalar processor（超标量处理器），212
Supervisor mode（管态模式），311，555，596，640
Symbol table（符号表），131
Synchronization（同步），458
Synchronous DRAM（SDRAM，同步 DRAM），276
Synchronous sequential circuit（同步时序电路），520
Synchronous transmission（同步传输），246
Syntax（语法），49
System-on-a-chip（片上系统），421
System space（系统空间），310

T

Technology（技术），17
Telemetry（遥测技术），390
Testability（可测试性），416
Text editor（文本编辑器），130
Thread（线程），444
  creation（创建），458
  synchronization（同步），458
Three-state（三态），参见 Tri-state
Thrashing（颠簸），327
Threshold（阈值），482
Time slicing（时间片），146
Timers（定时器），120，397，427
Timing signals（时序信号），6，171
Torus network（圆环形网络），453
Touchpad（触摸板），4
Trace mode（跟踪模式），136
Trackball（跟踪球），4
Transducers（传感器），407
Transistor（晶体管），17，20
Transition time（转换时间），490
Translation lookaside buffer（TLB，转换监视缓冲区），308
Transmission（传输）
  asynchronous（异步），245
  differential（差分），251，256，258，279
  single-ended（单端），250，256
  start-stop（起止），245
  synchronous（同步），246
Tri-state gate（三态门），179，236，240，281，491
Truth table（真值表），466

2's-complement representation（补码表示），10

U

UART（Universal Asynchronous Receiver Transmitter，通用异步收发器），395
Universal Serial Bus（USB，通用串行总线），247
Update protocol（更新协议），453
User mode（用户模式），311，555，596，639
User space（用户空间），310

V

Vector processing（向量处理），445，695，696
Vectorization（向量化），446
Very large-scale integration（VLSI，超大规模集成），17
Virtual address（虚拟地址），305
Virtual memory（虚拟存储器），269，305
  address translation（地址转换），306
  page（页），306
  page fault（页故障），309
  page frame（页帧），308
  page table（页表），308
  translation lookaside buffer（TLB，转换监视缓冲区），308
  virtual address（虚拟地址），305
Volatile（易失性）
  variable（变量），138
  memory（存储器），274

W

Wait loop（等待循环），100
Waiting for memory（等待存储器），171
Winchester disks（温切斯特磁盘），313
Word（字），5，29
Word alignment（字对齐），31
Word length（字长），5，29
Workstation（工作站），2
Write-back protocol（写回协议），参见 Cache memory
Write buffer（写缓冲区），303
Write operation（写操作），8，32
Write-through protocol（直接写协议），参见 Cache memory