计 算 机 科 学 丛 书

原书第8版

Intel微处理器

(美) Barry B. Brey 著 金惠华 艾明晶 尚利宏 等译
佐治亚州立大学 北京航空航天大学



The Intel Microprocessors

8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, and Core2 with 64-bit Extensions

Architecture, Programming, and Interfacing

Eighth Edition

机械工业出版社
China Machine Press

Intel微处理器（原书第8版）

这是一本将微型计算机原理、汇编语言程序设计和PC机接口通信技术有机整合在一起的著作，可作为高等院校计算机、通信和自动控制专业的教材，也可供工程技术人员参考。

本书主要内容：

- 采用200多个相关编程实例（许多用Visual C++及嵌入式汇编语言编写）来阐述基本概念。
- 开发软件，控制应用系统与微处理器接口。
- 用嵌入汇编语言的Microsoft Visual C++程序设计环境编写微处理器程序，控制PC机。
- 开发软件，控制键盘、显示器及其他各种计算机部件。
- 编写算术协处理器程序、MMX程序、SSE部件程序，求解复杂方程式。
- 解释Intel系列各种处理器的区别，明确每一型号的特性。
- 描述微处理器实模式（DOS）和保护模式（Windows）的用途。
- 说明存储器管理操作，控制保护模式和分页机制，分配存储器。
- 设计存储器、I/O系统到微处理器的接口。
- 开发驱动硬件接口和应用系统的软件。
- 解释嵌入式环境中实时操作系统（RTOS）的工作。
- 解释磁盘及视频系统的操作。
- 建立小型系统与PC机的ISA总线、PCI总线、并口或串口、USB总线之间的接口。
- 详述Pentium 4微处理器新的64位扩展（EMT-64）。

本版更新内容：

- 覆盖最新的Pentium 4和Core2处理器的内容，包括如何基于Pentium Core2及其新的64位体系化结构编程。
- 在DOS或者Windows环境下，如何使用Visual C++ Express编写C/C++与汇编程序接口的内容。
- 针对微处理器领域的最新进展进行了更新。

作者简介

Barry B. Brey

德福瑞大学（DeVry University）荣誉退休教授，他是美国关于微处理器和汇编语言著作的主要作者，至今著有33部教材。其个人主页为http://members.ee.net/brey/index.html。



PEARSON Education

PEARSON

www.pearsonhighered.com

ISBN 978-7-111-30485-2

9 787111 304852

定价：89.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjs@hzbook.com

华章网站 http://www.hzbook.com

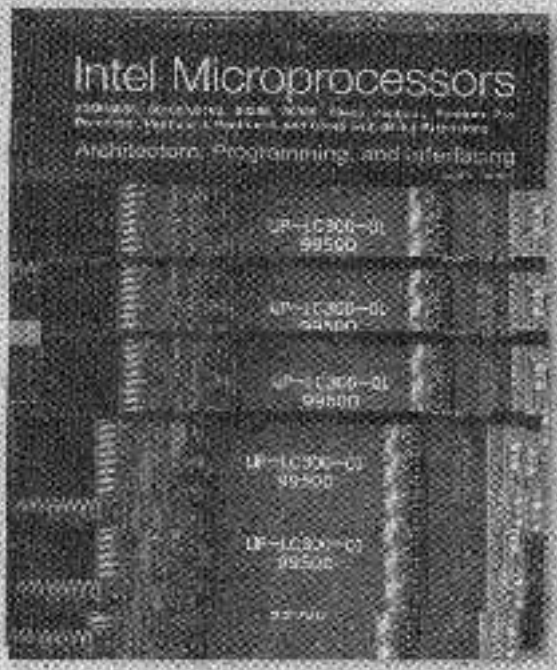
网上购书：www.china-pub.com

计 算 机 科 学 丛 书

原书第8版

Intel微处理器

(美) Barry B. Brey 著
德州消元学
金惠华 艾明晶 尚利宏 等译
北京[illegible]大学



The Intel Microprocessors

8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, and Core2 with 64-bit Extensions

Architecture, Programming, and Interfacing

Eighth Edition

机械工业出版社
China Machine Press

本书重点讲解 Intel 系列微处理器（8086/8088、80186/80188、80286、80386、80486、Pentium、Pentium Pro Processor、Pentium Ⅱ、Pentium Ⅲ、Pentium 4 和 Core2）的体系结构、程序设计和接口通信技术，并通过微型计算机原理把三者有机地整合在一起。本书以 Intel 系列微处理器为背景，以 DOS、Windows 和 Visual C/C++ 为编程环境，通过示例为读者深入揭示了微型计算机工作原理和最新的技术进步。许多示例都可以作为开发类似应用的样板或原型，用以指引开发新的应用。

本书适合作为高等院校计算机、电子通信和自动控制等专业教材，也可供工程技术人员参考。

Simplified Chinese edition copyright © 2010 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium Ⅱ, Pentium Ⅲ, Pentium 4, and Core2 with 64-bit Extensions: Architecture, Programming, and Interfacing, Eighth Edition* (ISBN 978-0-13-502645-8) by Barry B. Brey, Copyright © 2009, 2006, 2003, 2000, 1997, 1994, 1991, 1987.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice-Hall.

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2009-6760

图书在版编目（CIP）数据

Intel 微处理器（原书第 8 版）/（美）布雷（Brey, B. B.）著；金惠华等译. —北京：机械工业出版社，2010.6

（计算机科学丛书）

书名原文：The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium Ⅱ, Pentium Ⅲ, Pentium 4, and Core2 with 64-bit Extensions: Architecture, Programming, and Interfacing, Eighth Edition

ISBN 978-7-111-30485-2

Ⅰ. I…　Ⅱ. ①布…　②金…　Ⅲ. 微处理器，Intel 系列　Ⅳ. TP332

中国版本图书馆 CIP 数据核字（2010）第 074760 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码 100037）

责任编辑：秦　健

北京瑞德印刷有限公司印刷

2010 年 6 月第 2 版第 1 次印刷

184mm × 260mm · 44.25 印张

标准书号：ISBN 978-7-111-30485-2

定价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@ hzbook. com

出版者的话

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

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

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

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

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

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



华章教育
华章科技图书出版中心

译 者 序

这本讲述 Intel 系列微处理器程序设计和接口技术的参考书已经是第 8 版了。随着技术的发展和进步，作者多次改编增补删减，与时俱进，吐故纳新，在选材、知识点配置和编程环境选择方面充分体现了先进性和实用性。与前几版相比，第 8 版主要增加了讲解在 DOS 和 Windows 环境下如何使用可自由下载的 Visual C++ Express 编写 C/C++ 与汇编语言的接口，说明了在微处理器和微处理器接口方面出现的新成果。

本书的特点是：

- 取材涵盖微机原理、汇编语言和接口通信技术的教学要求和知识点，各部分有机整合，适合国内教学要求。
- 以 Intel 系列微处理器为背景，以 DOS、Windows 和 Visual C/C++ 为编程环境，通过示例为读者深入揭示了微机工作原理和最新技术进步。许多示例都可以作为读者开发类似应用的样板或原型，指引读者开发新的应用。
- 每章开头提示本章学习目的，末尾概要总结知识要点，最后有大量习题检验学习成果。编排符合学习规律，适合读者自学。

这是一本非常实用的教材，有助于读者彻底掌握 Intel 系列微处理器程序设计和接口技术，灵活自如地使用微机的各种资源，解决学习和开发工作中的实际问题。

本书由金惠华译第 1 ～6 章，艾明晶译第 9 ～15 章，尚利宏译第 16 ～19 章，高洁译第 7 章，郝广奇译第 8 章，李雅倩译附录，崔代锐、尚利荣、邓媛、刘云峰、徐其志参与了部分章节初译、示例习题核对及文稿录入。全书由金惠华统稿审校。译稿对原书中的笔误和疏漏进行了更正。由于译审者水平有限，译文中难免有不妥之处，敬请读者批评指正。

译者

2010 年 4 月

前　言

这本非常实用的参考书写给那些需要彻底掌握 Intel 系列微处理器程序设计和接口技术的大学生们。如今，任何在计算机应用领域里学习或工作的人都必须懂得汇编语言程序设计、一种 C 语言和接口技术，因为 Intel 系列微处理器已经在电子、通信、控制系统，特别是台式计算机系统等许多方面都得到了广泛而且有时独一无二的应用。第 8 版主要增加了讲解在 DOS 和 Windows 环境下如何使用可以从 Microsoft 自由下载的 Visual C++ Express 编写 C/C++ 与汇编语言的接口。很多应用程序包含 Visual C++ 作为用内嵌汇编程序学习汇编语言的基础。更新部分详细说明了在微处理器和微处理器接口方面出现的新成果。

组织结构和取材范围

为了培养综合的学习方法，每章开头都简明叙述了本章的目标。各章都包含了大量程序设计应用和实例，以阐明主题。每章末尾的数条小结对于指导学习事半功倍，并总结了前面讲解过的内容。习题部分则是对所学知识的进一步强化，并提供了实践机会。

本书大量地使用微软宏汇编（Microsoft Macro Assembler）程序和在 Visual C++ 环境中的内嵌汇编程序作实例，为学习编写 Intel 系列微处理器的程序提供了机会。有关程序设计环境的操作，包括链接器、库、宏、DOS 功能调用、BIOS 功能调用和 Visual C/C++ 程序开发等。对于各种版本 Visual C++ 在 16 位和 32 位两种编程环境下的内嵌汇编器（C/C++）都做了详细说明。本书是用 Visual C++ Express 2005 或 2008 作为开发环境写的，但也可以几乎不作更改地使用任何版本的 Visual Studio。

本书还详尽说明了系列中每种微处理器、存储系统和各种 I/O 系统（包括磁盘存储器、ADC 和 DAC、16550 UART、PIA、定时器、键盘/显示控制器、算术协处理器和视频显示系统），并讨论了 PC 机的各种总线（AGP、ISA、PCI、PCI Express、USB、串口和并口）。通过这些系统，可以学习到实用的微处理器接口技术。

学习方法

由于 Intel 系列微处理器各不相同，本书开头集中讨论实模式下的程序设计，它与 Intel 系列所有型号微处理器兼容。针对这些系列成员的指令，比较 8086/8088 微处理器和 80386、80486、Pentium、Pentium Pro、Pentium Ⅱ、Pentium Ⅲ及 Pentium 4 的异同，会发现所有这些微处理器非常相似，因此一旦学懂了基本类型的 8086/8088，就可以较容易地学习更高级的版本及其指令。注意，8086/8088 及随后的升级产品 80186/80188 和 80386EX 嵌入式微处理器仍然用于嵌入式系统中。

本书还讲解了算术协处理器、MMX 扩展和 SIMD 扩展的程序设计和操作，它们在系统中提供浮点计算的能力，这在控制系统、视频图像和计算机辅助设计（CAD）等应用领域是很重要的。算术协处理器允许程序完成复杂的算术运算，而用普通微处理器编程方法是难以胜任的。MMX 和 SIMD 指令可以使整数或者浮点数并行高速操作。

本书描述了 8086～80486 和所有 Pentium 微处理器的引脚及功能。在接口技术部分，首先讨论用于 8086/8088 的一些通用外围接口部件。说明基本部件后，再重点研究更先进的 80186/80188、80386、80486、Pentium 到 Pentium 4 微处理器。对 80286 的叙述很少，因为它与 8086 和 80386 很相似。我们将重点放在尽可能详细地讲述 80386、80486 和各种 Pentium 版本的微处理器上。

通过首先考虑各种先进微处理器的操作和程序设计，进而学习所有系列成员的接口技术，能够提供 Intel 系列微处理器的工作和实用背景。读者完成本书的学习后将能够：

1）开发软件，控制微处理器应用接口。通常，开发出的软件应能用于所有型号的微处理器，包括基于 DOS 的应用和基于 Windows 的应用。主要强调在 Windows 环境下开发内嵌汇编和 C++ 混合语言程序。

2）使用 MFC 控件处理程序和函数调用编写汇编语言和 C++ 程序，控制键盘、视频显示系统及磁盘存储器。

3）使用宏序列、过程、条件汇编、流程控制汇编指令开发软件，并链接到一个 Visual C++ 程序中。

4）使用查找表和算法开发代码变换软件。

5）对算术协处理器编程，求解复杂的方程式。

6）开发 MMX 和 SIMD 扩展软件。

7）解释 Intel 系列的各种处理器的区别，明确每一型号的特性。

8）描述并使用微处理器的实模式和保护模式操作。

9）设计存储器、I/O 系统到微处理器的接口。

10）对 Intel 系列中各微处理器及其软件和硬件接口进行详细且全面的比较。

11）解释嵌入式应用中实时操作系统的功能。

12）解释磁盘及视频系统的操作。

13）建立小型系统与 PC 系统的 ISA、PCI、串口、并口和 USB 总线之间的接口。

内容概述

第 1 章以基于微处理器的计算机系统为重点，介绍了 Intel 微处理器系列，包括微处理器的历史、操作和基于微处理器系统中存储数据的方法，还包括数制及其变换。第 2 章介绍了微处理器程序设计模型和系统结构，解释了实模式和保护模式的工作原理。

当我们了解了基本的计算机后，第 3 章到第 6 章讲解了 Intel 微处理器系列每条指令的功能，还提供了简单的应用程序来说明这些指令的操作，使读者建立程序设计的基本概念。

第 7 章介绍 Visual C/C++ Express 如何与内嵌汇编程序及单独的汇编语言程序设计模块一起使用，并说明如何配置一个简单的带汇编应用程序的 Visual C/C++ Express 程序。

有了程序设计基础之后，第 8 章提供了一些使用带内嵌汇编程序的 Visual C++ Express 编写的应用程序，这些应用程序包括通过消息处理函数在 Windows 环境下使用键盘和鼠标进行程序设计。把磁盘文件解释成 File 类，就像键盘和视频显示器一样通过 Windows 在 PC 机上操作。这一章提供了在 Windows 环境下几乎可在 PC 机系统上开发任何程序的工具。

第 9 章介绍了 8086/8088 系列，作为学习后面章节中基本存储器和 I/O 接口的基础，本章还解释了系统缓冲和系统定时。

第 10 章解释存储器接口，包括使用集成译码器的接口和用 VHDL 的可编程逻辑器件的接口，提供了 8 位、16 位、32 位和 64 位存储系统，因而 8086 ～ 80486 和 Pentium ～ Pentium 4 微处理器与存储器之间可以有接口。

第 11 章详细讨论了 I/O 接口技术，包括 PIA、定时器、16550 UART 和 ADC/DAC。本章还说明了直流电机和步进电机的接口。

在理解了这些基本 I/O 部件及它们与微处理器的接口后，第 12 章和第 13 章提供了一些高级 I/O 技术，包括中断和直接存储器存取（DMA）及其应用（打印机接口、实时时钟、磁盘存储器和视频显示系统）。

第 14 章详细叙述了 8087 ～ Pentium 4 系列算术协处理器的操作和程序设计技术，以及 MMX 和 SIMD 指令。今天，几乎没有不利用协处理器就能高效运行的应用程序。记住，自从 80486 以后，所有 Intel 微处理器都有了协处理器；自 Pentium 后都有一个 MMX 部件；自 Pentium Ⅱ后都有一个 SIMD 部件。

第 15 章阐明了如何通过并口、串口、ISA 和 PCI 总线使小型系统与 PC 机接口。

第 16 章和第 17 章涵盖 80186/80188 ～80486 这些先进的微处理器，探讨了它们与 8086/8088 微处理器的区别，以及它们的增强功能和特性，讲述了用于 80386 和 80486 微处理器的高速缓冲存储器、交叉存储和猝发存储。第 16 章还包括实时操作系统（RTOS），第 17 章还讨论了内存管理和内存分页技术。

第 18 章详述了 Pentium 和 Pentium Pro 微处理器，这些微处理器也基于最初的 8086/8088 微处理器。

第 19 章介绍了 Pentium Ⅱ、Pentium Ⅲ、Pentium 4 和 Core2 微处理器，包括一些新特性、封装类型和加到原指令系统中的指令集。

附录使本书更加充实。附录 A 列出了全部 DOS INT 21H 功能调用，还详细说明了汇编程序和 Windows Visual C++ 接口的使用。附录 B 给出所有 8086 ～Pentium 4 和 Core2 指令的完整列表，包括许多指令示例和十六进制机器编码，以及时钟定时信息。附录 C 简要列出了改变标志位的所有指令。附录 D 提供了本书编号为偶数的习题的答案。

致谢

非常感谢下列专家的反馈意见：Brigham Young 大学的 James K. Archibald 和 Broome 社区学院的 William H. Murray Ⅲ。

联络方式

我们可以通过 Internet 保持联络。我的网站包含本人全部教科书的信息和许多到 PC 机、微处理器、硬件和软件的重要链接。此外，还可以从中获得每周一次详述 PC 机许多方面的讲座，许多话题给出了本书未涉及的特别有趣的“技术环节”。如果你需要任何帮助，请在 bbrey@ ee .net 上与我联系，我会在 24 个小时以内回答所有我的电子邮件。

我的网站：http：//members. ee .net/brey

目　录

出版者的话
译者序
前言

第 1 章　微处理器和计算机导论 …… 1
1.1　历史背景 …… 1
1.1.1　机械时代 …… 1
1.1.2　电子时代 …… 2
1.1.3　程序设计的进步 …… 3
1.1.4　微处理器时代 …… 4
1.1.5　现代微处理器 …… 5
1.2　基于微处理器的 PC 系统 …… 12
1.2.1　存储器和 I/O 系统 …… 12
1.2.2　微处理器 …… 17
1.3　数制 …… 20
1.3.1　数字 …… 20
1.3.2　按位计数法 …… 20
1.3.3　其他数制转换到十进制 …… 21
1.3.4　十进制转换成其他进制 …… 22
1.3.5　二进制编码的十六进制 …… 23
1.3.6　补码 …… 24
1.4　计算机数据格式 …… 24
1.4.1　ASCII 和 Unicode 数据 …… 25
1.4.2　BCD 数据 …… 26
1.4.3　字节数据 …… 27
1.4.4　字数据 …… 28
1.4.5　双字数据 …… 30
1.4.6　实数 …… 31
1.5　小结 …… 32
1.6　习题 …… 33
第 2 章　微处理器及其体系结构 …… 36
2.1　微处理器的内部体系结构 …… 36
2.1.1　程序设计模型 …… 36
2.1.2　多功能寄存器 …… 38
2.2　实模式存储器寻址 …… 41
2.2.1　段和偏移 …… 41
2.2.2　默认段和偏移寄存器 …… 42
2.2.3　段和偏移寻址机制允许重定位 …… 43
2.3　保护模式存储器寻址简介 …… 43
2.3.1　选择子和描述符 …… 44
2.3.2　程序不可见寄存器 …… 47
2.4　内存分页 …… 48
2.4.1　分页寄存器 …… 48
2.4.2　页目录和页表 …… 50
2.5　平展模式内存 …… 51
2.6　小结 …… 52
2.7　习题 …… 52
第 3 章　寻址方式 …… 54
3.1　数据寻址方式 …… 54
3.1.1　寄存器寻址 …… 57
3.1.2　立即寻址 …… 58
3.1.3　直接数据寻址 …… 60
3.1.4　寄存器间接寻址 …… 63
3.1.5　基址加变址寻址 …… 65
3.1.6　寄存器相对寻址 …… 67
3.1.7　相对基址加变址寻址 …… 68
3.1.8　比例变址寻址 …… 70
3.1.9　RIP 相对寻址 …… 71
3.1.10　数据结构 …… 71
3.2　程序存储器寻址 …… 72
3.2.1　直接程序存储器寻址 …… 72
3.2.2　相对程序存储器寻址 …… 72
3.2.3　间接程序存储器寻址 …… 73
3.3　堆栈存储器寻址 …… 74
3.4　小结 …… 76
3.5　习题 …… 78
第 4 章　数据传送指令 …… 80
4.1　MOV 回顾 …… 80
4.1.1　机器语言 …… 80
4.1.2　Pentium 4 和 Core2 的 64 位模式 … 86
4.2　PUSH/POP 指令 …… 87
4.2.1　PUSH 指令 …… 87
4.2.2　POP 指令 …… 89
4.2.3　初始化堆栈 …… 90
4.3　装入有效地址 …… 91
4.3.1　LEA 指令 …… 91
4.3.2　LDS、LES、LFS、LGS 和 LSS

指令 …… 92
4.4 数据串传送 …… 94
4.4.1 方向标志 …… 94
4.4.2 DI 和 SI …… 94
4.4.3 LODS 指令 …… 94
4.4.4 STOS 指令 …… 95
4.4.5 MOVS 指令 …… 96
4.4.6 INS 指令 …… 98
4.4.7 OUTS 指令 …… 99
4.5 其他数据传送指令 …… 99
4.5.1 XCHG 指令 …… 99
4.5.2 LAHF 和 SAHF 指令 …… 100
4.5.3 XLAT 指令 …… 100
4.5.4 IN 和 OUT 指令 …… 101
4.5.5 MOVSX 和 MOVZX 指令 …… 102
4.5.6 BSWAP 指令 …… 103
4.5.7 CMOV 指令 …… 103
4.6 段超越前缀 …… 103
4.7 汇编程序详述 …… 104
4.7.1 伪指令 …… 104
4.7.2 存储器组织 …… 108
4.7.3 程序举例 …… 110
4.8 小结 …… 112
4.9 习题 …… 113
第 5 章 算术和逻辑运算指令 …… 115
5.1 加法、减法和比较指令 …… 115
5.1.1 加法指令 …… 115
5.1.2 减法指令 …… 119
5.1.3 比较指令 …… 121
5.2 乘法和除法指令 …… 122
5.2.1 乘法指令 …… 122
5.2.2 除法指令 …… 124
5.3 BCD 码和 ASCII 码算术运算指令 …… 127
5.3.1 BCD 算术运算指令 …… 127
5.3.2 ASCII 算术运算指令 …… 128
5.4 基本逻辑运算指令 …… 130
5.4.1 AND 指令 …… 130
5.4.2 OR 指令 …… 131
5.4.3 XOR 指令 …… 132
5.4.4 测试和位测试指令 …… 133
5.4.5 NOT 指令和 NEG 指令 …… 134
5.5 移位指令和循环移位指令 …… 134
5.5.1 移位指令 …… 134
5.5.2 双精度移位指令 …… 135
5.5.3 循环移位指令 …… 136
5.5.4 位扫描指令 …… 136
5.6 串比较指令 …… 137
5.6.1 SCAS 指令 …… 137
5.6.2 CMPS 指令 …… 137
5.7 小结 …… 138
5.8 习题 …… 139
第 6 章 程序控制指令 …… 141
6.1 转移指令 …… 141
6.1.1 无条件转移指令 …… 141
6.1.2 条件转移和条件设置 …… 145
6.1.3 LOOP 指令 …… 148
6.2 控制汇编语言程序的流程 …… 149
6.2.1 WHILE 循环 …… 151
6.2.2 REPEAT-UNTIL 循环 …… 152
6.3 过程 …… 153
6.3.1 CALL 指令 …… 154
6.3.2 RET 指令 …… 156
6.4 中断概述 …… 157
6.4.1 中断向量 …… 157
6.4.2 中断指令 …… 158
6.4.3 中断控制 …… 159
6.4.4 PC 机的中断 …… 159
6.4.5 64 位模式中断 …… 160
6.5 机器控制及其他指令 …… 160
6.5.1 控制进位标志位 …… 160
6.5.2 WAIT 指令 …… 160
6.5.3 HLT 指令 …… 161
6.5.4 NOP 指令 …… 161
6.5.5 LOCK 前缀 …… 161
6.5.6 ESC 指令 …… 161
6.5.7 BOUND 指令 …… 161
6.5.8 ENTER 和 LEAVE 指令 …… 161
6.6 小结 …… 162
6.7 习题 …… 163
第 7 章 在 C/C++ 中使用汇编语言 …… 165
7.1 在 16 位 DOS 应用程序中使用汇编语言与 C/C++ 语言 …… 165
7.1.1 基本规则和简单程序 …… 166
7.1.2 _asm 块中不能使用的 MASM 功能 …… 167
7.1.3 使用字符串 …… 167
7.1.4 使用数据结构 …… 169
7.1.5 混合语言编程的例子 …… 171
7.2 在 32 位应用程序中使用汇编语言与 Visual C/C++ 语言 …… 172

7.2.1 使用控制台 I/O 访问键盘和显示器的例子 …… 173
7.2.2 直接访问 I/O 端口 …… 174
7.2.3 开发 Windows 的 Visual C++ 应用程序 …… 174
7.3 汇编和 C++ 混合目标码 …… 180
7.3.1 用 Visual C++ 链接汇编语言 …… 181
7.3.2 在 C/C++ 程序中添加新的汇编语言指令 …… 184
7.4 小结 …… 185
7.5 习题 …… 185
第 8 章 微处理器程序设计 …… 187
8.1 模块化程序设计 …… 187
8.1.1 汇编程序和链接程序 …… 187
8.1.2 PUBLIC 和 EXTRN …… 189
8.1.3 库 …… 190
8.1.4 宏 …… 193
8.2 使用键盘和视频显示器 …… 195
8.2.1 读取键盘 …… 195
8.2.2 使用视频显示器 …… 198
8.2.3 在程序中使用定时器 …… 201
8.2.4 鼠标 …… 202
8.3 数据转换 …… 204
8.3.1 二进制转换为 ASCII 码 …… 204
8.3.2 ASCII 码转换为二进制 …… 205
8.3.3 显示和读入十六进制数 …… 206
8.3.4 使用查找表实现数据转换 …… 208
8.3.5 使用查找表的示例程序 …… 209
8.4 磁盘文件 …… 210
8.4.1 磁盘的组织 …… 211
8.4.2 文件名 …… 212
8.4.3 顺序存取文件 …… 212
8.4.4 随机存取文件 …… 220
8.5 程序举例 …… 222
8.5.1 时间/日期显示程序 …… 222
8.5.2 数字排序程序 …… 223
8.5.3 数据加密 …… 225
8.6 小结 …… 226
8.7 习题 …… 227
第 9 章 8086/8088 硬件特性 …… 228
9.1 引脚和引脚功能 …… 228
9.1.1 引脚 …… 228
9.1.2 电源要求 …… 229
9.1.3 直流特性 …… 229
9.1.4 引脚定义 …… 229
9.2 时钟产生器 8284A …… 232
9.2.1 8284A 时钟产生器 …… 232
9.2.2 8284A 的操作 …… 233
9.3 总线缓冲及锁存 …… 233
9.3.1 多路分离总线 …… 234
9.3.2 缓冲系统 …… 236
9.4 总线时序 …… 237
9.4.1 基本的总线操作 …… 237
9.4.2 一般的时序 …… 238
9.4.3 读时序 …… 238
9.4.4 写时序 …… 241
9.5 就绪和等待状态 …… 241
9.5.1 READY 输入 …… 241
9.5.2 RDY 和 8284A …… 242
9.6 最小模式与最大模式 …… 244
9.6.1 最小模式操作 …… 244
9.6.2 最大模式操作 …… 244
9.6.3 8288 总线控制器 …… 244
9.7 小结 …… 246
9.8 习题 …… 246
第 10 章 存储器接口 …… 248
10.1 存储器器件 …… 248
10.1.1 存储器引脚 …… 248
10.1.2 ROM 存储器 …… 249
10.1.3 静态 RAM（SRAM）器件 …… 251
10.1.4 动态 RAM（DRAM）存储器 …… 254
10.2 地址译码 …… 258
10.2.1 为什么要进行存储器译码 …… 258
10.2.2 简单的与非门译码器 …… 258
10.2.3 3-8 线译码器（74LS138） …… 259
10.2.4 双 2-4 线译码器（74LS139） …… 261
10.2.5 PLD 可编程译码器 …… 262
10.3 8088 和 80188（8 位）存储器接口 …… 265
10.3.1 基本的 8088/80188 存储器接口 …… 265
10.3.2 与快闪存储器接口 …… 268
10.3.3 错误校正 …… 270
10.4 8086、80186、80286 和 80386SX（16 位）存储器接口 …… 271
10.5 80386DX 和 80486（32 位）存储器接口 …… 278
10.5.1 存储体 …… 278
10.5.2 32 位存储器接口 …… 279
10.6 Pentium ~ Core2（64 位）存储器接口 …… 281

10.7 DRAM …… 284
10.7.1 DRAM 回顾 …… 284
10.7.2 EDO 存储器 …… 286
10.7.3 SDRAM …… 286
10.7.4 DDR …… 286
10.7.5 DRAM 控制器 …… 287
10.8 小结 …… 287
10.9 习题 …… 288
第 11 章 基本 I/O 接口 …… 289
11.1 I/O 接口概述 …… 289
11.1.1 I/O 指令 …… 289
11.1.2 独立编址 I/O 与存储器映像 I/O …… 290
11.1.3 PC 机 I/O 映像 …… 291
11.1.4 基本输入输出接口 …… 291
11.1.5 握手 …… 293
11.1.6 关于接口电路的注释 …… 294
11.2 I/O 端口地址译码 …… 296
11.2.1 译码 8 位 I/O 地址 …… 296
11.2.2 译码 16 位 I/O 地址 …… 297
11.2.3 8 位与 16 位 I/O 端口 …… 298
11.2.4 32 位 I/O 端口 …… 300
11.3 可编程外围设备接口 …… 303
11.3.1 82C55 基本描述 …… 303
11.3.2 82C55 编程 …… 304
11.3.3 方式 0 操作 …… 305
11.3.4 与 82C55 接口的 LCD 显示器 … 309
11.3.5 方式 1 选通输入 …… 319
11.3.6 方式 1 选通输出 …… 321
11.3.7 方式 2 双向操作 …… 322
11.3.8 82C55 方式小结 …… 324
11.3.9 串行 EEPROM 接口 …… 325
11.4 8254 可编程间隔定时器 …… 326
11.4.1 8254 功能描述 …… 326
11.4.2 8254 编程 …… 327
11.4.3 直流电机速度与方向控制 …… 331
11.5 16550 可编程通信接口 …… 334
11.5.1 异步串行数据 …… 335
11.5.2 16550 功能描述 …… 335
11.5.3 16550 编程 …… 336
11.6 模/数转换器（ADC）与数/模转换器（DAC） …… 340
11.6.1 DAC0830 数/模转换器 …… 341
11.6.2 ADC080X 模/数转换器 …… 342
11.6.3 使用 ADC0804 和 DAC0830 的实例 …… 344
11.7 小结 …… 345
11.8 习题 …… 346
第 12 章 中断 …… 348
12.1 基本中断处理 …… 348
12.1.1 中断的目的 …… 348
12.1.2 中断 …… 349
12.1.3 中断指令：BOUND、INTO、INT、INT 3 和 IRET …… 351
12.1.4 实模式中断操作 …… 351
12.1.5 保护模式中断操作 …… 352
12.1.6 中断标志位 …… 352
12.1.7 将一个中断向量存入向量表 …… 353
12.2 硬件中断 …… 354
12.2.1 INTR 和$\overline{\text{INTA}}$ …… 355
12.2.2 82C55 键盘中断 …… 358
12.3 扩展中断结构 …… 360
12.3.1 使用 74ALS244 扩展 …… 360
12.3.2 菊花链中断 …… 361
12.4 8259A 可编程中断控制器 …… 362
12.4.1 8259A 概述 …… 362
12.4.2 连接单个 8259A …… 363
12.4.3 级联多个 8259A …… 364
12.4.4 8259A 编程 …… 364
12.4.5 8259A 编程实例 …… 368
12.5 中断实例 …… 375
12.5.1 实时时钟 …… 375
12.5.2 中断处理键盘 …… 377
12.6 小结 …… 379
12.7 习题 …… 379
第 13 章 直接存储器存取及 DMA 控制 I/O …… 381
13.1 基本 DMA 操作 …… 381
13.2 8237 DMA 控制器 …… 382
13.2.1 软件命令 …… 386
13.2.2 地址寄存器和计数寄存器编程 …… 386
13.2.3 8237 与 80X86 微处理器相连 … 387
13.2.4 用 8237 进行存储器到存储器传输 …… 387
13.2.5 DMA 处理的打印机接口 …… 392
13.3 共享总线操作 …… 394
13.3.1 定义的总线类型 …… 395
13.3.2 总线仲裁器 …… 395
13.4 磁盘存储系统 …… 400

13.4.1 软盘存储器 …… 400
13.4.2 笔式驱动器 …… 403
13.4.3 硬盘存储器 …… 403
13.4.4 光盘存储器 …… 405
13.5 视频显示器 …… 406
13.5.1 视频信号 …… 407
13.5.2 TTL RGB 显示器 …… 407
13.5.3 模拟 RGB 显示器 …… 408
13.6 小结 …… 411
13.7 习题 …… 412
第 14 章 算术协处理器、MMX 和 SIMD 技术 …… 413
14.1 算术协处理器的数据格式 …… 413
14.1.1 带符号的整数 …… 413
14.1.2 二进制编码的十进制（BCD） … 414
14.1.3 浮点数 …… 414
14.2 80X87 的结构 …… 416
14.3 指令系统 …… 421
14.3.1 数据传送指令 …… 421
14.3.2 算术运算指令 …… 422
14.3.3 比较指令 …… 423
14.3.4 超越运算指令 …… 424
14.3.5 常数操作指令 …… 424
14.3.6 协处理器控制指令 …… 424
14.3.7 协处理器指令 …… 426
14.4 算术协处理器编程 …… 438
14.4.1 计算圆的面积 …… 438
14.4.2 求谐振频率 …… 439
14.4.3 使用一元二次方程求根 …… 440
14.4.4 使用内存数组存储结果 …… 441
14.4.5 将单精度浮点数转换为字符串 … 442
14.5 MMX 技术简介 …… 443
14.5.1 数据类型 …… 443
14.5.2 指令系统 …… 444
14.6 SSE 技术概述 …… 452
14.6.1 浮点数 …… 453
14.6.2 指令集 …… 454
14.6.3 控制/状态寄存器 …… 454
14.6.4 编程实例 …… 455
14.6.5 优化 …… 458
14.7 小结 …… 458
14.8 习题 …… 459
第 15 章 总线接口 …… 461
15.1 ISA 总线 …… 461
15.1.1 ISA 总线的发展 …… 461
15.1.2 8 位 ISA 总线输出接口 …… 461
15.1.3 8 位 ISA 总线输入接口 …… 466
15.1.4 16 位 ISA 总线 …… 468
15.2 外围部件互连（PCI）总线 …… 468
15.2.1 PCI 总线的引脚图 …… 469
15.2.2 PCI 总线的地址/数据线 …… 469
15.2.3 配置空间 …… 470
15.2.4 PCI 总线的 BIOS …… 472
15.2.5 PCI 接口 …… 474
15.2.6 PCI Express 总线 …… 474
15.3 并行打印机接口（LPT） …… 475
15.3.1 端口介绍 …… 475
15.3.2 使用并行端口而不需要 ECP 支持 …… 477
15.4 串行 COM 端口 …… 477
15.5 通用串行总线（USB） …… 480
15.5.1 连接器 …… 480
15.5.2 USB 数据 …… 480
15.5.3 USB 命令 …… 481
15.5.4 USB 总线节点 …… 482
15.5.5 USBN9604/3 编程 …… 482
15.6 加速图形端口（AGP） …… 485
15.7 小结 …… 485
15.8 习题 …… 485
第 16 章 80186、80188 及 80286 微处理器 …… 487
16.1 80186/80188 的结构 …… 487
16.1.1 80186/80188 的型号 …… 487
16.1.2 80186 基本结构框图 …… 488
16.1.3 80186/80188 基本特征 …… 488
16.1.4 引脚 …… 490
16.1.5 直流工作特性 …… 492
16.1.6 80186/80188 时序 …… 492
16.2 80186/80188 增强功能编程 …… 495
16.2.1 外设控制块（PCB） …… 495
16.2.2 80186/80188 的中断 …… 495
16.2.3 中断控制器 …… 496
16.2.4 定时器 …… 500
16.2.5 DMA 控制器 …… 505
16.2.6 片选单元 …… 507
16.3 80C188EB 接口举例 …… 510
16.4 实时操作系统（RTOS） …… 516
16.4.1 实时操作系统（RTOS）概述 … 516
16.4.2 实例系统 …… 517
16.4.3 线程系统 …… 519

16.5 80286 简介 …… 523
16.5.1 硬件特性 …… 523
16.5.2 新增指令 …… 524
16.5.3 虚拟存储机 …… 525
16.6 小结 …… 526
16.7 习题 …… 526
第 17 章 80386 和 80486 微处理器 …… 528
17.1 80386 微处理器简介 …… 528
17.1.1 存储系统 …… 530
17.1.2 输入/输出系统 …… 536
17.1.3 存储器和 I/O 控制信号 …… 537
17.1.4 时序 …… 537
17.1.5 等待状态 …… 538
17.2 特定的 80386 寄存器 …… 538
17.2.1 控制寄存器 …… 538
17.2.2 调试和测试寄存器 …… 540
17.3 80386 存储管理 …… 541
17.3.1 描述符和选择子 …… 541
17.3.2 描述符表 …… 544
17.3.3 任务状态段（TSS） …… 545
17.4 向保护模式转换 …… 547
17.5 虚拟 8086 模式 …… 556
17.6 内存分页机制 …… 557
17.6.1 页目录 …… 557
17.6.2 页表 …… 557
17.7 80486 微处理器简介 …… 559
17.7.1 80486DX 和 80486SX 微处理器的引脚 …… 560
17.7.2 80486 的基本结构 …… 563
17.7.3 80486 的存储系统 …… 564
17.8 小结 …… 566
17.9 习题 …… 566
第 18 章 Pentium 和 Pentium Pro 微处理器 …… 568
18.1 Pentium 微处理器简介 …… 568
18.1.1 存储系统 …… 571
18.1.2 输入/输出系统 …… 572
18.1.3 系统时序 …… 572
18.1.4 分支预测逻辑 …… 574
18.1.5 高速缓存结构 …… 574
18.1.6 超标量体系结构 …… 574
18.2 Pentium 的特定寄存器 …… 574
18.2.1 控制寄存器 …… 574
18.2.2 EFLAG 寄存器 …… 575
18.2.3 内置自检（BIST） …… 575
18.3 Pentium 的存储管理 …… 576
18.3.1 分页单元 …… 576
18.3.2 存储管理模式 …… 576
18.4 Pentium 的新指令 …… 577
18.5 Pentium Pro 微处理器简介 …… 581
18.5.1 Pentium Pro 的内部结构 …… 582
18.5.2 引脚连接 …… 583
18.5.3 存储系统 …… 586
18.5.4 输入/输出系统 …… 587
18.5.5 系统时序 …… 587
18.6 Pentium Pro 的特性 …… 587
18.7 小结 …… 588
18.8 习题 …… 589
第 19 章 Pentium Ⅱ、Pentium Ⅲ、Pentium 4 和 Core2 微处理器 …… 590
19.1 Pentium Ⅱ微处理器简介 …… 590
19.1.1 存储系统 …… 595
19.1.2 输入/输出系统 …… 596
19.1.3 系统时序 …… 596
19.2 Pentium Ⅱ软件变化 …… 597
19.2.1 CPUID 指令 …… 597
19.2.2 SYSENTER 和 SYSEXIT 指令 …… 597
19.2.3 FXSAVE 和 FXRSTOR 指令 …… 598
19.3 Pentium Ⅲ …… 598
19.3.1 芯片组 …… 598
19.3.2 总线 …… 598
19.3.3 引脚 …… 599
19.4 Pentium 4 和 Core2 …… 600
19.4.1 存储器接口 …… 600
19.4.2 寄存器组 …… 601
19.4.3 超线程技术 …… 602
19.4.4 多核技术 …… 602
19.4.5 CPUID …… 602
19.4.6 特定模型寄存器 …… 605
19.4.7 性能监视寄存器 …… 605
19.4.8 64 位扩展技术 …… 606
19.5 小结 …… 607
19.6 习题 …… 607
附录 A 汇编程序、Visual C++ 和 DOS …… 608
附录 B 指令系统一览 …… 614
附录 C 标志位的变化 …… 674
附录 D 偶数号习题的答案 …… 676

第 1 章　微处理器和计算机导论

引言

本章介绍 Intel 系列微处理器的概况，讨论计算机的发展历史和基于微处理器的计算机系统中微处理器的功能。并且介绍计算机领域中使用的术语，这样，当我们讨论微处理器和计算机时就可以理解计算机行话了。

方框图及其功能说明详述计算机系统的操作，框图中的块表示 PC 机的存储器和输入/输出的相互联系。本章详细说明数据如何在存储器中存储，以便开发软件时使用各类数据。数值型数据以整数、浮点数和二进制编码的十进制（BCD）形式存储；而字母型数据以 ASCII 码（American Standard Code for Information Interchange，美国标准信息交换码）和 Unicode 码的形式存储。

目的

读者学习完本章后将能够：

1）使用适当的计算机术语交谈，例如位、字节、数据、实存储系统、保护模式存储系统、Windows、DOS、I/O 等。
2）简洁地叙述计算机的历史，并且列出计算机系统能执行的应用程序。
3）说明 80X86 和 Pentium ～ Pentium 4 系列各个成员的概况。
4）画出计算机系统的方框图，并且说明每块的功能。
5）叙述微处理器的功能，并且详述它的基本操作。
6）定义 PC 中存储系统的内容。
7）进行二进制、十进制和十六进制数据之间的转换。
8）区分和表示数字及字母信息，如整数、浮点数、BCD 和 ASCII 数据。

1.1　历史背景

本节概述导致微处理器发展的历史事件，并且具体讲解功能强大且十分流行的 80X86[⊖]、Pentium、Pentium Pro、Pentium Ⅲ、Pentium 4[⊖] 和 Core2 微处理器。尽管研究历史不是理解微处理器所必需的，但是它从历史的角度展示了计算机的快速发展。

1.1.1　机械时代

计算系统的思想并不是新的，远在现代电气科学和电子器件出现以前它就已经存在了。用机器计算的概念在公元前 500 年就有记载，那时的巴比伦人发明了**算盘**（**abacus**），这是第一个机械式计算器。算盘用串珠实现计算，古代的巴比伦神父用它管理他们的巨大粮仓。算盘直到今天还在使用，始终没有改进，到 1642 年，当时的数学家 Blaise Pascal 发明了由齿轮和转轮构成的计算器。每个齿轮有十个齿牙，当其中一个齿轮转动一圈时，第二个齿轮推进一个齿牙。这和汽车里程表的原理一样，是所有机械计算器的基础。顺便说一下，PASCAL 程序设计语言就是为了纪念 Blaise Pascal 在数学和机械计算器方面的开拓性工作而命名的。

第一个实际用于自动计算信息的轮式机械计算器可追溯到 19 世纪初，这是在人类发明灯泡和深入了解电之前。在这个计算机的萌芽时代，人们梦想有会用程序计算数据的机器——而不仅仅用计算器

⊖ 80X86 是 8086、8088、80186、80188、80286、80386、80486 以及 Pentium 系列的简写形式。

⊖ Pentium、Pentium Pro、Pentium Ⅱ、Pentium Ⅲ、Pentium 4 和 Core2 是 Intel 公司的注册商标。

计算几个数据。

1937 年人们通过一些设计图纸和日记发现，机械式计算机器的一位早期的开拓者是 Charles Babbage，在 Lovelace 伯爵夫人 Augusta Ada Byron 的帮助下，受人不到颇皇家天文协会委托，Babbage 于 1823 年研制可编程序的计算机器，这个机器要为皇家海军绘制导航表。他接受了挑战，并开始建造他称为分析机（**Analytical Engine**）的机器。这个机器就是由蒸汽驱动的机械式计算机，它存储 1000 个 20 位长的十进制数字和一个可变的程序，程序能够改机器功能以便执行各种计算任务。这个机器通过穿孔卡片输入，酷似 20 世纪五六十年代计算机使用的穿孔卡片。他可能借鉴了法国人 Joseph Jacquard 提出的用穿孔卡片的思想，后者在 1801 年就在他发明的现今称为 "Jacquard 织布机" 的编织机器中使用了穿孔卡片作为输入。Jacquard 织布机用穿孔卡片为其生产的布匹选择复杂的编织图案，人们称其为穿孔卡片编程织布机。

努力多年以后，Babbage 对他的梦想逐渐失去信心，因为他认识到那个时代的机械师不可能制造出完成工作所需要的机械零件。分析机需要 50 000 多个机械零件，无法以足够的精密度做出来，因此无法使分析机正常地工作。

1.1.2 电子时代

19 世纪出现了电动机（由 Michael Faraday 构想），并且出现了许多电动机驱动的加法机，这些都建立在 Blaise Pascal 的机械计算器基础上。这些电动的机械计算器一直作为通用办公设备使用，直到 20 世纪 70 年代初出现了由 Bomar 公司首先推出，叫 **Bomar Brain** 的小型手持电子计算器。Monroe 也是电子计算器的先驱者，但他的机器是台式的，相当于一台 4 抽屉书桌的大小。

1889 年 Herman Hollerith 研制了存储数据的穿孔卡片。如同 Babbage 一样，他也显然借鉴了 Jacquard 穿孔卡片的思想。他还开发了由一种新式电机驱动的机械式计算器，这个计算器可计算、分类和比较存储在穿孔卡片上的信息。用机器进行计算的想法引起了美国政府的兴趣，因此委托 Hollerith 用穿孔卡片系统存储 1890 年人口普查的资料并制成表格。

1896 年，Hollerith 组建了 Tabulating Machine Company 公司（制表机械公司），这个公司开发了用穿孔卡片制表的行式机器。经过数次兼并后，Tabulating Machine Company 成为 International Business Machines Corporation（国际商用机器公司），现在称为 IBM。为了纪念 Herman Hollerith，我们通常将计算机系统中使用的穿孔卡片称为 **Hollerith 卡**，穿孔卡片使用的 12 位代码称为 **Hollerith 码**。

用电机驱动的机械式机器，一直主导着信息处理世界，直到 1941 年出现第一台电子计算机。作为工程师为柏林 Henschel 飞机公司工作的德国发明家 Konrad Zuse 发明了第一台现代计算机。在 1936 年 Zuse 构造了他的一个机械版的系统，并且随后在 1939 年构造了他的第一个电－机计算机系统，叫做 Z2。他的 Z3 计算机，如图 1-1 所示，第二次世界大战期间德国人用它设计飞机和导弹。Z3 是工作时钟为 5.33 Hz 的一台继电器逻辑机器（比最新的几个 GHz 级的微处理器慢得太多了）。如果当时德国政府给予 Zuse 足够的资金，他很可能研制出功能更强的计算机系统。今天 Zuse 虽然得到了一些迟到的承认，对他在数字电子领域和 Z3 计算机系统的开创性工作表示敬意。

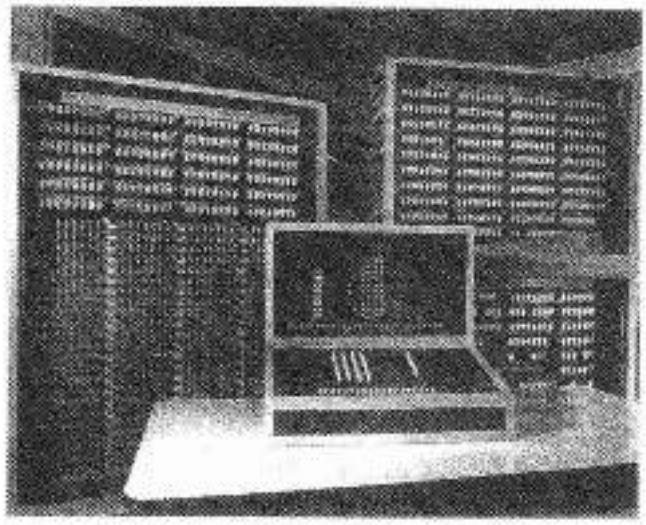


图 1-1 Konrad Zuse 研制的 Z3 计算机，时钟频率为 5.33 Hz
（承蒙 Konrad 的儿子 Horst Zuse 提供）

最近发现（通过解密的英国军事文件）第一台真正的电子计算机于1943年安装运转，用于破译德国军事密码。这第一台使用了真空管的电子计算机系统是由Alan Turing发明的。Turing称他的机器为**巨人（Colossus）**，或许是因为机器的尺寸庞大。巨人的问题是，虽然它可以破译由**英格玛机（Enigma machine）**生成的德国军事密码，但是却不能解决其他问题。巨人不是可编程的，它是固定程序的计算机系统，今天通常称为**专用计算机**。

第一台通用可编程电子计算机系统于1946年由宾夕法尼亚大学研制成功。这是第一台现代计算机，称为**ENIAC（Electronic Numerical Integrator and Calculator，电子数字积分器和计算器）**。ENIAC是个庞大的机器，它使用了17 000多个真空管和超过500英里长的导线。这个庞然大物的重量超过30吨，而每秒只能执行约10万次运算。ENIAC推动世界进入了电子计算机时代。ENIAC采用重新连接线路的方法实现编程，这个过程需要许多工人花几天时间才能完成。工人们改变插接板上的电路连接，此操作方式很像早期的电话接线总机。ENIAC的另一个问题是真空管器件的寿命低，需要经常维护。

随后的突破性进展是1947年12月23日由贝尔实验室的John Bardeen，William Shockley和Walter Brattain研制出了晶体管。其后，在1958年得克萨斯仪器公司的Jack Kilby发明了集成电路。集成电路导致20世纪60年代数字集成电路（RTL，即电阻-晶体管逻辑）的发展，以及1971年Intel公司第一种微处理器的诞生。当时Intel工程师Federico Faggin，Ted Hoff和Stan Mazor研制出了4004微处理器（美国专利号：No.3,821,715），该微处理器启动了今天还在继续加速进行着的微处理器革命。

1.1.3 程序设计的进步

既然开发出了可编程序的机器，因此程序和程序设计语言也开始相继出现。如上所述，第一个可以编写程序的电子计算机系统是通过重新连接线路实现编程的。由于这在实际应用中很麻烦，因此在计算机系统发展的早期，产生了用于控制计算机的计算机语言。第一种这样的语言叫作**机器语言**，是由多个1和0组成的二进制代码，以指令组的形式存储在计算机系统中，被称为程序。这种方法比通过重新连接机器线路进行编程的方法有效，但是开发程序仍然非常耗费时间，因为完全要用数码来编程。数学家冯·诺依曼（John Von Neumann）首先开发了接受指令并且可将指令存储到存储器中的系统。为了纪念他，计算机常常称为**冯·诺依曼机器（Von Neumann machine）**。（回想一下，Babbage比冯·诺依曼更早地提出了这一概念。）

20世纪50年代初，随着UNIVAC之类的计算机系统投入使用，**汇编语言**简化了以二进制代码为计算机输入指令的繁琐工作。汇编语言允许程序员用助记符代替二进制码，例如用ADD表示加法，代替二进制码0100 0111。虽然汇编语言可以帮助进行程序设计，但是编程仍然很不容易，直到1957年Grace Hopper开发了称为**FLOWMATIC**的第一个高级程序设计语言。同年，IBM为它的计算机系统开发了**FORTRAN（FORmula TRANslator，公式翻译器）**。FORTRAN语言允许程序员开发使用公式解决数学问题的程序，至今，一些科学家仍然使用FORTRAN。比FORTRAN约晚一年出现了另一种类似的语言**ALGOL（ALGOrithmic Language，算法语言）**。

第一个真正成功并广泛用于商业的程序设计语言是**COBOL（COmputer Business Oriented Language，面向商业计算机的语言）**。尽管近几年COBAL的使用减少了，但是在许多大的商业系统中它仍然发挥着主要作用。另外一种一度很流行的商业语言是**RPG（Report Program Generator，报告程序生成器）**，它允许通过规范输入、输出和运算的格式进行程序设计。

在这些早期的程序设计语言之后，更多的语言相继出现了，比较普及的是BASIC、Java、C#、C/C++、PASCAL和ADA。BASIC和PASCAL被设计成为教学用语言，但其使用范围早已扩展到了许多计算机系统中。BASIC语言可能是所有语言中最容易学习的，据估计80%的PC用户程序是用BASIC语言编写的。在十年前，BASIC新版本Visual BASIC的出现使Windows环境中的程序设计更容易了。Visual BASIC语言可能最终取代C/C++和PASCAL作为一种科学语言，但这值得怀疑。比起C#更贴近硬件，它更加表面；现实点可能替代C/C++和包括Java的大多数其他语言，可能最终替换BASIC。这当然仅是推测，只有将来才会看到哪种语言最终能成为霸主。

在科学界，C/C++偶尔还有PASCAL和FORTRAN通常用于控制程序。最近一个对嵌入式系统的

调查表明：60%用C语言开发，30%用汇编语言开发，剩余的用BASIC和Java语言开发。这些语言，特别是C/C++允许程序员几乎完全控制编程环境和计算机系统。许多情况下，C/C++正在替代某些低级机器控制软件或驱动程序，通常它们都是留给汇编语言的。但即使如此，汇编语言在程序设计中仍然起着重要的角色，为PC写的视频游戏程序几乎只用汇编语言。为了更有效地实现机器控制功能，汇编语言也经常与C/C++混合使用。在最新Pentium和Core2微处理器上出现的一些很新的并行指令只能在汇编语言里编程。

ADA语言广泛用于国防部门中。称为ADA语言是为了纪念Augusta Ada Byron，即Lovelace伯爵夫人，19世纪初她与Charles Babbage一起开发了分析机软件。

1.1.4 微处理器时代

世界上的第一个微处理器，Intel 4004，是一个4位微处理器，是可编程单片控制器。它只寻址4096个4位宽存储单元**位**（**bit**），是取值为1或者0的二进制数，4位宽的存储单元通常称为**半字节**（**nibble**）。4004指令系统只有45条指令，用P沟道MOSFET技术制造，允许以50 KIPS（**kilo-instructions per second，每秒千条指令**）的速度执行指令。这比1946年的重30吨的ENIAC计算机所能达到的100KIPS的速度要慢，但4004的重量远小于1盎司。

最初，这个器件用量很大。4位微处理器首先用于早期的视频游戏和基于微处理器的小型控制系统中。这种早期的视频游戏之一，推移板游戏，是由Bailey设计的。这种早期微处理器的主要问题是它的速度、字宽度和存储器容量不足。当Intel推出对早期4004的改进型号4040时，4位微处理器的改革就此结束了。尽管4040对字宽度和存储器容量方面的改进不够，但是4040的运行速度有了提高。其他公司，特别是得克萨斯仪器仪表公司也生产了4位微处理器（TMS-1000）。4位微处理器在低档应用领域中依然存在，如用于微波炉和小型控制器系统中，并且仍然可以从某些微处理器厂商那儿得到它们。大部分计算器也仍然是基于4位微处理器的，处理4位BCD（binary-coded decimal，二进制编码的十进制）码。

1971年年末，Intel公司认识到微处理器是个可赢利的产品，因此又推出了8008，这是4004的8位扩展型微处理器。8008可寻址的存储器空间扩大了（16KB），并且增加了指令（总计48条），这些为它在许多高级系统中的应用提供了机会。**字节**通常是8位宽的二进制数，**K**代表1024。通常，存储器容量按KB计算。

工程师们研究了针对8008微处理器的许多应用需求，他们发现它的存储器容量小，速度慢并且指令系统也有限，因此限制了它的应用。Intel认识到这些局限，于1973年推出了8080微处理器，这是第一个现代的8位微处理器。大约在Intel发布8080微处理器6个月后，Motorola公司推出了它的MC6800微处理器，从此打开了微处理器的闸门。8080和MC6800在某种程度上开创了微处理器的时代。不久，其他公司也相继推出了它们自己的8位微处理器。表1-1列出了这些早期的微处理器以及它们的生产厂家。这些早期的微处理器厂家中，只有Intel和Motorola继续成功地生产不断更新换代的微处理器，IBM也生产Motorola类型的微处理器。Motorola已经出售它的半导体部门，即现在的Freescale Semiconductors公司。Zilog仍然制造微处理器，但它坚持在自己特定的领域里，集中研制微控制器和嵌入式控制器，而不是通用微处理器。Rockwell几乎放弃了开发微处理器，而转向开发调制解调电路。Motorola已从微处理器市场占有份额的50%降到更少。Intel目前在台式机和笔记本电脑市场占有率接近100%。

表1-1 早期的8位微处理器

制造商	型号
Fairchild	F-8
Intel	8080
MOS Technology	6502
Motorola	MC6800
National Semiconductor	IMP-8
Rockwell International	PPS-8
Zilog	Z-8

8080的特点

8080不仅扩充了可寻址的存储器容量和指令系统，而且指令执行速度是8008的10倍。8008系统的加法需要20μs（每秒5万条指令），而8080系统只需要2μs（每秒50万条指令）。另一方面8080可直接与TTL（晶体管-晶体管逻辑）兼容，而8008则不能。这样就使得接口设计更容易，而且价格更

便宜。8080 可寻址的范围（64KB）是 8008（16KB）的 4 倍，这些改进导致进入了 8080 时代，并且使微处理器继续繁荣昌盛。随后，1974 年第一台 PC 机 MITS Altair 8800 问世了（注意，选择 8800 这个名字，可能是为了避免侵犯 Intel 的版权）。为 Altair 8800 计算机写的 BASIC 语言解释程序是由 Bill Gates（比尔·盖茨）和 Paul Allen 于 1975 年开发的，他们是 Microsoft 公司的创始人。Altair 8800 的汇编程序是由 Digital Research 公司编写的，它曾为 PC 机开发了 DR-DOS。

8085 微处理器

1977 年 Intel 公司推出了 8080 的更新版本——8085。这是 Intel 公司开发的最后一个 8 位通用微处理器。尽管它只比 8080 稍微先进了一些，但是它执行软件的速度更高。8080 的加法操作花费 2.0μs（每秒 50 万条指令），而 8085 只花费 1.3μs（每秒执行 769 230 条指令）。8085 的主要优点是有内部时钟发生器、内部系统控制器和更高的时钟频率。这种高的组件集成度降低了成本，增加了 8085 微处理器的实际应用范围。Intel 已经销售了 1 亿片 8085 微处理器，这是它最成功的 8 位通用微处理器。因为许多其他公司也生产（第二货源）8085，所以这种微处理器已超过 2 亿片。含有 8085 的电器至今仍然被使用着，并且将来很可能继续用它。另外，Zilog 公司也销售了 5 亿片 8 位微处理器，即 Z-80 微处理器。Z-80 使用与 8085 兼容的机器语言代码，这意味着执行 8085/Z-80 兼容代码的微处理器已经超过 7 亿片。

1.1.5 现代微处理器

1978 年，Intel 推出了 8086 微处理器，并在一年多以后推出了 8088。这两种都是 16 位微处理器，执行一条指令只需要 400ns（2.5**MIPS**，即每秒执行 250 万条指令），执行速度大大超过 8085，这表示 8086 在运行速度上已取得了很大的进步。另外，8086 和 8088 可寻址 1MB 存储器，比 8085 多 16 倍（1**MB** 存储器容纳 1024KB 存储单元，即 1 048 576 字节）。更高的执行速度和更大的存储器容量使得 8086 和 8088 在许多应用中能替代类似的小型计算机。8086/8088 的另一个显著特点是使用了小型的 4 字节或 6 字节的指令高速缓冲存储器或者说指令队列，在指令执行前就可预先取出几条指令排队。队列使多指令序列的操作加快了许多，并且为现代微处理器中更大的指令高速缓冲存储器奠定了基础。

8086 和 8088 存储器容量的扩大和指令的扩充，使得微处理器的应用范围更加广泛。扩充的指令系统包括早期微处理器所没有的乘法和除法指令。指令的数量从 4004 的 45 条增加到 8085 的 246 条，直到 8086 和 8088 微处理器的 20 000 多条。注意，这些微处理器因为指令的数量多和复杂程度高，而称为 **CISC**（**complex instruction set computer**，**复杂指令系统计算机**）。虽然指令的数量多，学习时费时间，但是增加指令使得开发高效和复杂的应用任务变得更容易。16 位微处理器比 8 位微处理器还增加了更多的内部寄存器，这就使编写的软件效率更高。

因为需要更大的存储容量，16 位微处理器成为发展主流。1981 年，IBM 决定在它的 PC 机中使用 8088 微处理器，使得 Intel 系列更加普及。像电子表格、文字处理、拼写检查、计算机辞典等应用都需要大的存储器容量，要求比 8 位微处理器内的 64KB 更大的存储空间。16 位的 8086 和 8088 微处理器为这些应用提供 1MB 存储器。不久，甚至 1MB 存储器也限制了大的数据库以及其他方面的应用。这就导致 Intel 于 1983 年推出 80286 微处理器，它是 8086 的更新换代产品。

80286 微处理器

80286 微处理器（也是 16 位微处理器）除了寻址 16MB 存储系统而不是 1MB 存储器以外，几乎和 8086/8088 完全相同。80286 的指令系统也几乎和 8086/8088 完全相同，只是为了管理额外的 15MB 存储器而增加了几条指令。80286 的时钟速度增加了，8.0MHz 时钟的 80286 执行某些指令的时间还不到 250ns（4.0MIPS）。指令内部执行部分也有些改进，因此，与 8086/8088 相比，许多指令速度提高了 8 倍。

32 位微处理器

各种计算机应用开始要求微处理器的速度更高，存储器容量更大，并且数据通路更宽。因此，导致 Intel 公司于 1986 年推出了 80386，80386 对 16 位 8086 ～ 80286 微处理的结构做了很大的改进。80386 是 Intel 的第一个包含 32 位数据总线和 32 位地址线的微处理器（注意，此前 Intel 曾生产过 32 位微处理器，称为 iapx-432，但成绩不佳）。80386 通过 32 位总线可寻址高达 4GB 的存储器（1GB 存储器

包含1024MB，即1 073 741 824个单元）。4GB存储器可以存储100万张双面打印纸的ASCII文本数据。80386还有几个变体版本，例如80386SX，通过16位数据总线和24位地址总线，可以寻址16MB存储器空间，而80386SL/80386SLC则可以通过16位数据总线和25位地址总线寻址32MB存储器空间。80386SLC型含有内部高速缓冲存储器，允许以更高的速度处理数据。1995年Intel推出了80386EX微处理器。80386EX称为**嵌入式（embedded）**PC，因为在单片集成电路上包含了AT级PC机的全部组件，80386EX还包含24根输入/输出数据线、26位地址总线、16位数据总线、DRAM刷新控制器和可编程的片选逻辑。

使用GUI（graphical user interface，图形用户接口）的软件系统要求微处理器具有更高的速度和更大的存储器容量。现代图形显示通常包含256 000或者更多的图形元素（**pixel**或**pel**，**像素**或**像元**）。最小复杂度的**VGA（variable graphics array，可变图形阵列）**视频显示器的分辨率为每条扫描线640个像素，共有480条扫描线。为了显示一屏信息，每个像素必须可变，这就需要高速微处理器。几乎所有新软件包都使用这种视频接口，这些基于GUI的软件包要求高速的微处理器和加速的图形适配器，以便快速而有效地处理视频文本和图形数据。要求为其图形显示接口进行高速计算的著名系统是微软公司的Windows[⊖]。我们通常称GUI为**WYSIWYG（What you see is what you get，所见即所得）**显示器。

需要32位微处理器，是因为其数据总线的宽度适于传送32位宽的实数（单精度浮点数）。为了有效地处理32位实数，微处理器必须在它自己和存储器之间有效传递数据。通过8位数据总线传送32位数据需要花费4个读写周期，而通过32位数据总线传送只需要1个读写周期，这就有效地提高了处理实数程序的速度。大多数高级语言、电子表格和数据库管理系统都使用实数存储数据。实数也用于图形设计软件包中，它用矢量绘制视频屏幕上的图形，如AVTOCAD、ORCAD等**CAD（computer-aided drafting/design，计算机辅助画图/设计）**系统。

除了提供更高的时钟速度以外，80386还包含存储管理部件，允许操作系统分配和管理存储器资源。早期的微处理器将存储管理留给软件完成。80386包括用于存储管理和存储器分配的硬件电路，因此提高了效率，减少了软件开销。

80386的指令系统与早期的8086、8088和80286微处理器是兼容的。增加的指令引用32位寄存器，并且管理存储系统。注意，用于80286的存储管理指令和技术也与80386微处理器兼容。这些特性允许早期的16位软件可在80386微处理器上运行。

80486微处理器

1989年Intel公司推出了80486微处理器，它将类似80386的微处理器、类似80387的算术协处理器和8KB的高速缓冲存储器合并到一片集成块中。虽然80486与80386没有根本的差别，但还是有一处显著的改进。80486修改了80386的内部结构，大约一半的指令只在一个时钟周期内完成，而不是两个时钟周期。50MHz的80486大约一半指令的执行只花费25ns（50MIPS）。在同样的时钟速度下，执行典型的混合指令的平均速度约比80386提高了50%。更新型的80486执行指令的速度更高，采用了66MHz的倍频（80486DX2）。66MHz倍频型按66MHz速率执行指令，按33MHz速率进行存储器传送。Intel三倍频型80486DX4的内部执行速度提高到100MHz，存储器传送速度按33MHz。注意，80486DX4执行指令的速度几乎与60MHz的Pentium一样。它也包括扩充的16KB高速缓冲存储器，代替早期80486微处理器的标准8KB高速缓冲存储器。Advanced Micro Device（AMD）公司制造了三倍频型微处理器，其总线运行速度为40MHz，而时钟速度为120MHz。将来一定能够出现指令内部执行速度高达10GHz甚至更高的微处理器。

另一种型号的80486称为OverDrive[⊖]处理器，它实际上是倍频型的80486DX，用来取代80486SX或低速的80486DX。当把它插入到插座上时，可禁用或取代80486SX或80486DX，并且如同倍频型微处理器一样工作。例如，如果用OverDrive处理器替换一个工作于25MHz的80486SX，它就相当于一个

⊖ Windows是微软公司的注册商标，现有Windows 98、Windows 2000、Windows ME和Windows XP。

⊖ OverDrive是Intel公司的注册商标。

存储器传输率为25MHz而主频为50MHz的80486DX2微处理器。

表1-2列出了许多Intel和Motorola制造的微处理器，并且给出了它们的字长和存储器容量。其他公司也制造微处理器，但没有一个达到Intel或Motorola那样的成就。

表1-2 多种现代Intel和Motorola微处理器

制造商	型号	数据总线宽度	存储器容量（B）
Intel	8048	8	2K内部存储器
	8051	8	8K内部存储器
	8085A	8	64K
	8086	16	1M
	8088	8	1M
	8096	16	8K内部存储器
	80186	16	1M
	80188	8	1M
	80251	8	16K内部存储器
	80286	16	16M
	80386EX	16	64M
	80386DX	32	4G
	80386SL	16	32M
	80386SLC	16	32M+8K高速缓存
	80386SX	16	16M
	80486DX/DX2	32	4G+8K高速缓存
	80486SX	32	4G+8K高速缓存
	80486DX4	32	4G+16K高速缓存
	Pentium	64	4G+16K高速缓存
	Pentium OverDrive	32	4G+16K高速缓存
	Pentium Pro	64	64G+16K L1高速缓存 +256K L2高速缓存
	Pentium Ⅱ	64	64G+32K L1高速缓存 +256K L2高速缓存
	Pentium Ⅲ	64	64G+32K L1高速缓存 +256K L2高速缓存
	Pentium 4	64	64G+8K L1高速缓存 +512K L2高速缓存或更大 (64位可扩展到IT)
	Pentium 4 D（Dual Core）	64	IT+32K L1高速缓存 +2M或4M L2高速缓存
	Core2	64	IT+32K L1高速缓存 +共享2M或4M L2高速缓存
	Itanium（Dual Core）	128	IT+2.5M L1和L2高速缓存 +24M L3高速缓存
Motorola	6800	8	64K
	6805	8	2K
	6809	8	64K
	68000	16	16M
	68008D	8	4M
	68008Q	8	1M
	68010	16	16M
	68020	32	4G
	68030	32	4G+256高速缓存
	68040	32	4G+8K高速缓存
	68050	32	提出过，但是并没有提供
	68060	64	4G+16K高速缓存
	PowerPC	64	4G+32K高速缓存

Pentium 微处理器

1993 年推出的 Pentium 微处理器类似于 80386 和 80486 微处理器。这个微处理器原来称为 P5 或 80586。但是 Intel 决定不使用数字，因为它不可能取得一个数字的版权。Pentium 有两个先导型，一个时钟频率为 60MHz 或 66MHz，指令执行速度 110MIPS；而另一版本的时钟频率高达 100MHz，一倍半频，其指令执行速度为 150MIPS。也可以得到较高速的型号，一种二倍频的 Pentium，运行速度为 120MHz 和 133MHz（Intel 制造的最快版本的 Pentium 时钟频率为 233MHz，是三倍半频类型）。Pentium 的另一个区别是其高速缓冲存储器的容量从 80486 基本型的 8KB 增加到 16KB。Pentium 包含 8KB 指令高速缓冲存储器和 8KB 的数据高速缓冲存储器，由于得益于高速缓冲存储器，所以允许程序传送大量存储数据。存储器容量高达 4GB，其数据总线宽度从 80386 和 80486 中的 32 位拓宽到 64 位。根据 Pentium 型号的不同，数据总线传输速度是 60MHz 或者是 66MHz（前面提到 80486 的总线速度是 33MHz）。这么宽的数据总线允许使用双精度浮点数来实现由高速向量生成图形显示。这种高速度的总线允许虚拟现实软件和视频显示以更逼真的速度在当前和将来的 Pentium 平台上操作。Pentium 拓宽的数据总线和如此高的执行速度允许以 30Hz 或者比商业电视机更高的扫描频率实现全屏视频显示。最新型号的 Pentium 还包含称为多媒体扩展的附加指令，或称为 MMX 指令。虽然 Intel 希望 MMX 指令被广泛使用，但是目前只有少数公司使用它，主要原因是这些指令没有高级语言的支持。

Intel 还推出了人们期待已久的 Pentium OverDrive（P24T），它以 63MHz 或 83MHz 的速度运行。63MHz 的型号是 80486DX2 50MHz 系统的改进型，83MHz 的型号是 80486DX2 66MHz 系统的改进型。改进后的 83MHz 系统的运行速度介于 66MHz Pentium 和 75MHz Pentium 之间。如果说早期的 VESA 局部总线视频显示和磁盘缓冲控制器好像造价太贵，Pentium OverDrive 则代表了一种从 80486 到 Pentium 的理想升级途径。

或许 Pentium 最有创意的特性是它的双整数处理技术。Pentium 同时执行两条彼此独立的指令，因为其内部有两个独立的整数处理器，这称为超标量技术，这种技术允许 Pentium 每个时钟周期执行两条指令。另一个增强性能的特性是转移预测技术，加快了包含循环的程序的执行。如同 80486 一样，Pentium 也有内部浮点协处理器，它能够以高于 80486 五倍的速度处理浮点数据。这些特性预示 Intel 系列微处理器将继续取得成功，也可使 Pentium 取代某些在每个时钟执行一条指令的 **RISC**（**reduced instruction set computer**，**精简指令系统计算机**）机器。注意，一些新型的 RISC 处理器通过引入超标量技术每个时钟周期也可执行一条以上的指令。Motorola、Apple 和 IBM 最近推出了 PowerPC，它是有两个整数部件和一个浮点部件的 RISC 微处理器。PowerPC 确实提高了 Apple Macintosh[⊖] 的性能，但是仍然竞争不过 Intel 系列微处理器。测试表明，PowerPC 执行 DOS 和 Windows 应用程序比 80486DX 25MHz 微处理器还慢。因此，Intel 系列微处理器在 PC 机领域仍然处于遥遥领先地位。注意，现今约有 6 百万台 Apple Macintosh 系统，而基于 Intel 微处理器的 PC 机超过 2.6 亿台。1998 年的报告说明有 96% 的 PC 机安装了 Windows 操作系统。

最近，苹果电脑采用 Intel 的 Pentium 替代了原来在其大多数系统中安装的 PowerPC。这一举动表明 PowerPC 不能够与 Intel 的 Pentium 系列并驾齐驱了。

为了比较各种不同微处理器的速度，Intel 策划使用 iCOMP 评测指数方案，这个指数是 SPEC92、ZD Bench 和 Power Meter 的组合。iCOMP1 用于直到 Pentium 的全部 Intel 系列微处理器的速度评测。图 1-2 显示了微处理器的相对速度，其中 80386SX 16MHz 在图的下端，而 Pentium 200MHz 在图的最上端。

自从推出 Pentium Pro 和 Pentium Ⅱ，Intel 改为用 iCOMP2 指数进行评测，它相当于 10 倍的 iCOMP1 速率指数。如果某微处理器用 iCOMP1 测速得到的指数为 1000，则相当于用 iCOMP2 测速时的 100。另一个区别是用基准程序计分。图 1-3 给出了直到 Pentium Ⅲ 1000MHz 的 iCOMP2 参数表。图 1-4 为 Pentium Ⅲ和 Pentium 4 的 SYSmark 2002 评价指标。遗憾的是，Intel 自从 SYSmark 2002 评价指标以外没有发布任何用于比较微处理各个版本的基准程序。更新的基准程序不能用于与其他版本比较。

⊖ Macintosh 是 Apple 计算机公司的注册商标。

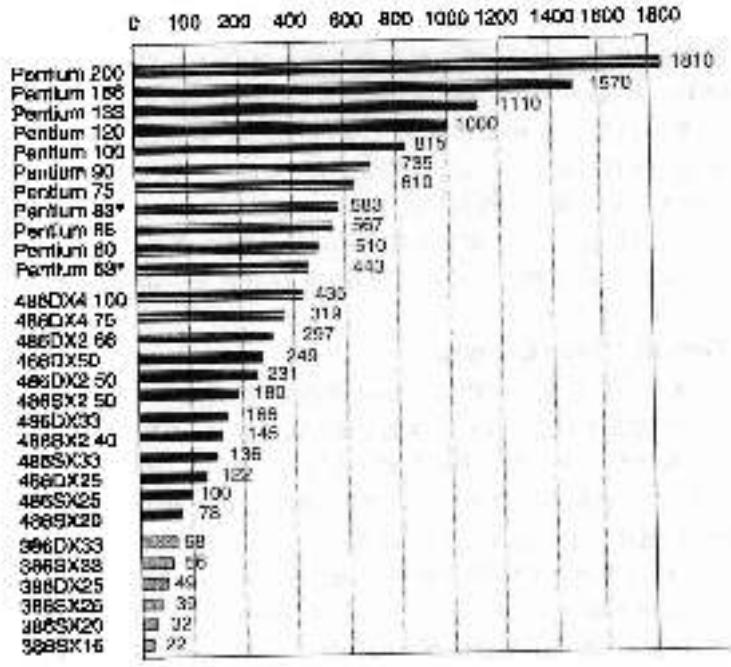


图 1-2 Intel 的 iCOMP1 指数

注：* 表示 Pentium OverDrive，第一部分非线性增长，166MHz 和 200MHz 的 Pentium 采用了 MMX 技术。

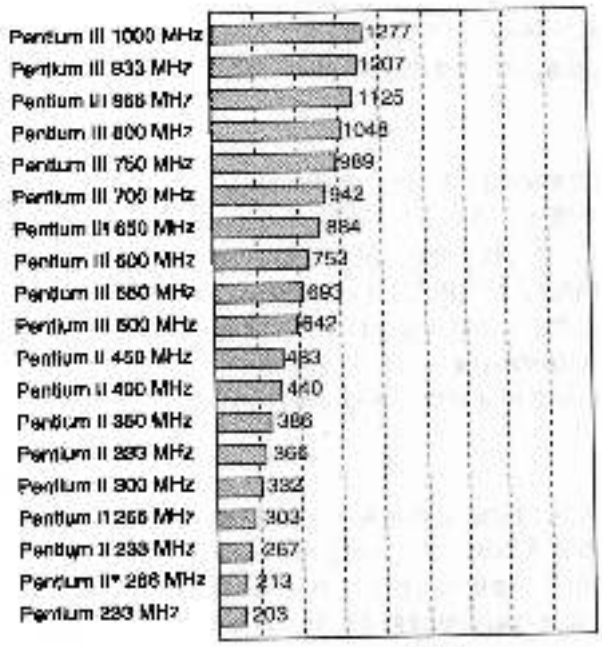


图 1-3 Intel 的 iCOMP2 指数

注：* Pentium Ⅱ Celeron 没有高缓冲器，上面给出的 iCOMP2 指数乘以 2.568 即可转换为 iCOMP3 指数。

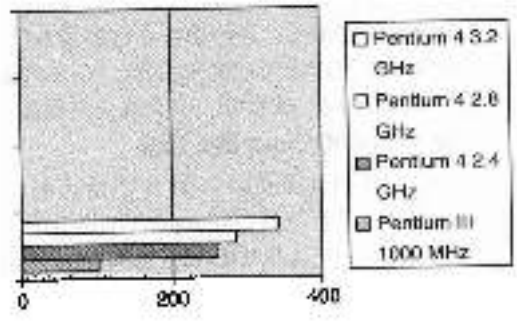


图 1-4 用 SYSmark 2002 评价指标的 Intel 微处理器性能

Pentium Pro 处理器

不久前 Intel 推出了 Pentium Pro 微处理器，原来称作 P6 微处理器。Pentium Pro 处理器有 2100 万个晶体管，3 个整数部件和一个用于提高多数软件执行速度的浮点单元。1995 年年底的产品基本时钟频率为 150MHz 和 166MHz。Pentium Pro 处理器除了内部有 16KB 的一级（L1）高速缓冲存储器（8KB 用于数据，8KB 用于指令）以外，还有 256KB 的二级（L2）高速缓冲存储器。另一个显著变化是 Pentium Pro 处理器使用三个执行部件，因此可同时执行三条指令，即使它们有冲突仍然可并行执行。这与只有不发生冲突时才能并行执行两条指令的 Pentium 是不同的。Pentium Pro 微处理器通过优化可高效率地处理 32 位代码，因此，通常它捆绑安装了 Windows NT，而不是普通版本的 Windows 95。还有一个区别是，它既可寻址 4GB 存储系统也可寻址 64GB 存储系统。如果配置了 64GB 存储系统，则 Pentium Pro 具有 36 位地址总线。

Pentium Ⅱ和 Pentium Ⅱ Xeon 微处理器

Pentium Ⅱ微处理器（1997 年推出）代表了 Intel 的新方向。它被安装在一块小型电路板上，而不是如同以前微处理器那样的集成电路。如此改变的主要原因是，将 L2 高速缓冲存储器放在 Pentium 的主电路板上满足不了这种新型微处理器的快速要求。在这种 Pentium 系统中，二级（L2）高速缓冲存储器以 60MHz 或 66MHz 系统总线速度操作。二级高速缓冲存储器和微处理器都放在称为 **P Ⅱ 模块**（**Pentium Ⅱ module**）的电路板上，这种板上的 L2 高速缓冲存储器以 133MHz 的速度工作，可以存储 512KB 信息。P Ⅱ模块上的微处理器实际是带 MMX 扩展的 Pentium Pro 微处理器。

1998 年，Intel 改变了 Pentium Ⅱ的总线速度。由于从 266MHz 到 333MHz 的 Pentium Ⅱ微处理器使用速度为 66MHz 的外部总线，出现了瓶颈问题，所以新型 Pentium Ⅱ微处理器使用速度为 100MHz 的总线。速度为 350MHz、400MHz 和 450MHz 的 Pentium Ⅱ微处理器全部都使用速度高于 100MHz 的存储器总线。高速度的存储器总线要求采用 8ns 的 SDRAM 取代用 66MHz 总线时的 10ns SDRAM。

1998 年中期，Intel 公布了称为 Xeon[⊖] 的新型 Pentium Ⅱ，它是专为高端工作站和服务器应用而设计的。Pentium Ⅱ与 Pentium Ⅱ Xeon 之间的主要区别是，Xeon 可使用 32KB 的 L1 高速缓冲存储器和 512KB、1MB 或 2MB 的 L2 高速缓冲存储器。Xeon 与 440GX 芯片组一起工作，设计成 4 个 Xeon 在同一系统中运行，类似于 Pentium Pro。这种新产品标志着 Intel 的战略变革：Intel 现在生产专业型的和家用/商用型的 Pentium Ⅱ微处理器。

Pentium Ⅲ微处理器

Pentium Ⅲ微处理器采用比 Pentium Ⅱ微处理器更快的内核，但它仍属于 P6 或 Pentium Pro 微处理器范畴。它有内嵌在塑料盒里、插入 slot1 插槽中的版本，也有一种看起来很像老式 Pentium 封装、被称为倒装式芯片的 370 插座版本。Intel 声称倒装式芯片的版本价格较低。另外的差别是 Pentium Ⅲ的时钟频率可达 1GHz。slot1 版含有 512KB 高速缓冲存储器，而倒装式芯片版含有 256KB 高速缓冲存储器。因为 slot1 版高速缓冲存储器工作在 1.5 倍的时钟频率上，所以速度比较快；而倒装式芯片式版本工作在 1 倍的时钟频率上。两种版本的存储器总线都用 100MHz；但 Celeron[⊖] 用 66MHz 的存储器总线。

从微处理器连接到存储控制器、PCI 控制器和 AGP 控制器的处理器总线是 100MHz 或者 133MHz。纵然存储器工作在 100MHz，也会改善其性能。

Pentium 4 和 Core2 微处理器

Pentium 4 微处理器是 2000 年末推出的，Pentium 的最新版本被称为 Core2。像 Pentium Pro 直到 Pentium Ⅲ（PⅢ）一样，Pentium 4 采用 Intel 的 P-6 体系结构。主要区别是 Pentium 4 可以有 3.2GHz 或更高速的版本，并且芯片组支持 Pentium 4 用 RAMBUS 存储器总线技术或 DDR 存储器代替曾经是标准的 SDRAM 技术。Core2 可达到 3GHz 的处理速度。如此高的微处理器速度是通过改进内部集成电路的

⊖ Xeon 是 Intel 公司的注册商标。

⊖ Celeron 是 Intel 公司的商标。

尺寸达到的，目前是0.045微米技术或者45纳米技术。相当有趣的是：Intel将一级高速缓冲存储器的大小由32KB变到了8KB和大部分最新的达到64KB。研究表明：对于最初推出的微处理器，这种尺寸足够大了，对于将来的微处理器可能还是含有64KB L1高速缓冲存储器。正如Pentium敷铜版微处理器那样，L2高速缓冲存储器仍然保持为256KB，新款可包含512KB。Pentium 4 Extreme Edition含有2MB的L2，Pentium 4e的L2是1MB，而Core2的L2是2MB或4MB。

另外可能发生的变化是：内部连接从铝连线变成铜连线。铜是良导体，将来它可以提高微处理器的时钟频率。使用铜连接的方法已经在IBM公司实现了，这是千真万确的事实。我们还可看到另外的事实是：处理器总线速度提高了，很可能会超过当前的最大值1033 MHz。

表1-3显示了Intel各种P号与微处理器的归属关系，P号表示在各种Intel处理器中是什么样的微处理器核。注意，自从Pentium Pro以后，所有处理器都用一样的基本微处理器核。

表1-3 Intel微处理器内核（P）对应的型号

内核型号（P）	微处理器
P1	8086、8088（80186和80188）
P2	80286
P3	80386
P4	80486
P5	Pentium
P6	Pentium Pro、Pentium Ⅱ、Pentium Ⅲ、Pentium 4和Core2
P7	Itanium

Pentium 4和Core2，64位和多核微处理器

最近Intel已把一些新修改版包含在Pentium 4和Core2中，包括一个64位核和多核。64位修改版允许微处理器通过64位宽地址寻址比4GB更多的内存。通常这些新版本的40个地址管脚允许访问到1TB（万亿字节）内存。64位机器还允许64位整数运算，但这比起能寻址更多的内存来并不是很重要。

最大的技术进步不在于64位操作，而是有了多核。每个核执行程序中一个单独的任务，如果程序利用多核设计，就可以提高执行速度。这样设计的程序称为多线程应用。现在，Intel生产双核和四核处理器，但将来核的数目很可能增加为八核，甚至十六核。Intel面对的问题是时钟速度不能提高到很高的速率，因此多核就是目前提供高速微处理器的解决办法。按照这个意思，高速时钟是不是就没有用了呢？只有将来才会预示它有用还是没用。

Intel新近演示了一个包含80个核的用45nm制造技术的Core2。Intel预期下个5年的某个时间发行80核版，制造技术将变成更精细的35nm或者25nm技术。

微处理器的未来

没有人能真正准确地预见未来，但是Intel系列的成功还将继续一些年。有可能朝着RISC技术方向发展，但是更可能朝着Intel和Hewlett-Packard联合开发被称作超线程的新技术方向发展。即使这种新技术也会内嵌80X86系列微处理器的CISC指令系统，以便用于这种系统的软件能够继续服务。这种技术的基本思想是许多微处理器都能与其他微处理器直接通信，允许不修改指令系统和程序就能进行并行处理。当前，超标量技术使用了许多微处理器，但是它们都共享相同的寄存器组。这种新技术，将包含多个微处理器，每个微处理器都有自己的寄存器组，而且与其他微处理器的寄存器组连接。这种技术将不需要任何专用程序就能真正实现并行处理。

超线程技术会在未来继续发展，导致更多并行处理器（目前是两个处理器）。有迹象显示Intel也会将芯片组合并到处理器封装中。

2002年末Intel推出了64位宽的新型微处理器体系结构，有128位宽的数据总线。这种新型体系结构命名为Itanium[⊖]，是Intel与Hewlett-Packard合作的尝试，称作EPIC（Explicitly Parallel Instruction Computing，显式并行指令计算）。Itanium结构与Pentium Ⅲ或Pentium 4之类的传统结构比较，允许更大的并行度。这些变化包括128个通用整数寄存器，128个浮点寄存器，64个判定寄存器和多个执行

⊖ Itanium是Intel公司的商标。

部件，确保为软件提供充足的硬件资源。Itanium 是为服务器市场设计的，未来很可能会向下渗透到家用及商用市场。

图 1-5 是比较 80486 ～ Pentium 4 微处理器的概念视图。每个视图都展示了这些微处理器的内部结构：CPU、算术协处理器和高速缓冲存储器（cache），说明了每种微处理器的复杂性和集成度。

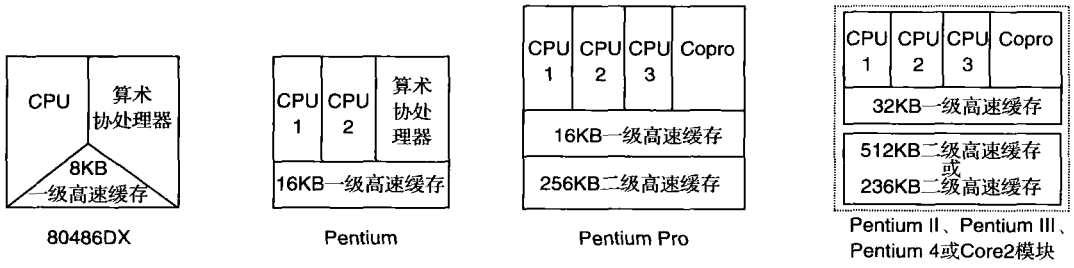


图 1-5 80486、Pentium Pro、Pentium Ⅱ、Pentium Ⅲ、Pentium 4 和 Core2 微处理器的概念视图

1.2 基于微处理器的 PC 系统

近期计算机系统经历了许多变化。以前占用很大场地的机器，由于使用了微处理器，缩小成为台式的计算机系统。虽然这些台式计算机都是小型的，但是它们的处理和计算能力在以前还是个幻想。20 世纪 80 年代初耗资百万美元的大型计算机还不如当今基于 Core2 的计算机功能强。实际上，许多小公司正在用基于微处理器的计算机系统取代它们的大型计算机。一些如 DEC（现在属于 Hewlett-Packard 公司）之类的公司，为了专注基于微处理器的计算机系统的生产，已经停产了大型计算机。

本节讨论基于微处理器的 PC 系统结构，包括许多基于微处理器的计算机系统中使用的存储器和操作系统的知识。

参见图 1-6 中 PC 的框图，该图适用于任何计算机系统，从早期的大型机到近期的微型机。框图由三个框组成，通过**总线**互连（总线是一组公用连线，传递同类信息。例如，地址总线包含 20 条或更多条连接线，把存储地址传送到存储器）。这些方框以及它们在 PC 中的功能将在本节讲述。

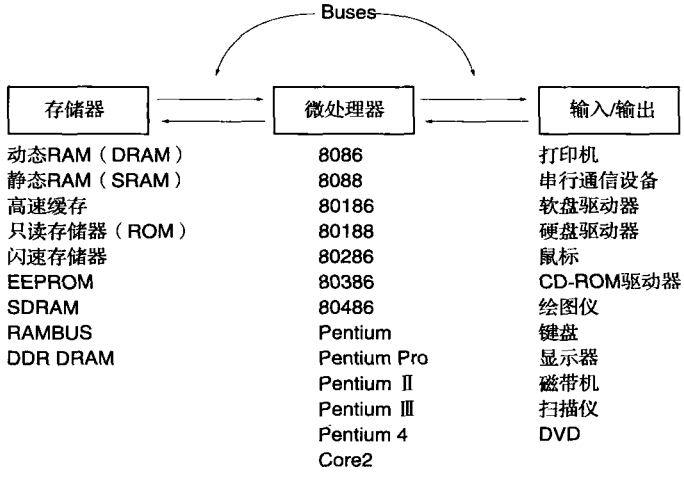


图 1-6 基于微处理器的计算机系统框图

1.2.1 存储器和 I/O 系统

所有基于 Intel 80X86 ～ Pentium 4 的 PC 系统的存储器结构都是类似的，包括 1981 年 IBM 推出的基于 8088 的第一台 PC，直到今天基于速度高、功能强的 Pentium 4 或 Core2 的 PC。图 1-7 表示 PC 系统的

存储器映像图，这个映像图适用于任何 IBM 的 PC 以及任何现有的与 IBM 兼容的 PC。

存储系统划分为三个主要部分：**TPA**（**transient program area，临时程序区**），**系统区**（**system area**）和 **XMS**（**extended memory system，扩展内存系统**）。由计算机中微处理器的类型决定是否存在扩展内存。如果计算机是基于早期的 8086 或 8088（PC 或 XT），则有 TPA 区和系统区，而没有扩展内存。PC 或 XT 包含 640KB 的 TPA 和 384KB 的系统存储器，存储器容量总计 1MB。我们通常把第一个 1MB 存储器称为实存储器或常规存储器，因为每个 Intel 微处理器设计成在实模式下操作时，都在这个区域内运行。

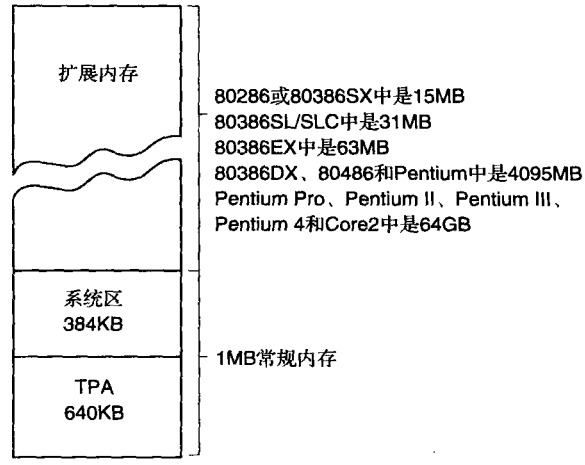


图 1-7 PC 系统的存储器映像

基于 80286 ～ Pentium 4 的计算机系统，不仅包括 TPA（640KB）和**系统区**（384KB），还可能包含扩展内存系统，这些机器通常称为 AT 型的机器。IBM 生产的 PS/1 和 PS/2 是具有同样存储器结构的另两个型号。有时这些机器也称为 **ISA**（**industry standard architecture，工业标准体系结构**）或者 **EISA**（**extended ISA，扩展的 ISA**）机器。PS/2 称为微通道（Micro-channel）体系结构系统还是称为 ISA 系统取决于型号。

Pentium 微处理器和 ATX 类型机器的推出带来了一个变化，就是增加了称为 **PCI**（**peripheral component interconnect，外设部件互连**）的新总线，现在已用于几乎所有的 Pentium 到 Pentium 4 系统中。在基于 80286 和 80386SX 的计算机中扩展内存的容量最高达 15MB，而在 80386DX、80486 和 Pentium 微处理器中扩展内存的容量则高达 4095MB，此外它们还有第一个 1MB 的实存储器。在 Pentium Pro 到 Core2 计算机系统中有高达 1MB 到 4GB 或到 64GB 的扩展内存。服务器趋向使用 64GB 大存储空间，家用及商用则使用 4GB 存储空间。ISA 机有一个 8 位外围总线用于将 8 位设备接口到基于 8086/8088 的 PC 或 XT 计算机系统。AT 档机器也称为 ISA 机器，使用 16 位外围接口总线，并可以包含 80286 或更高型号的微处理器。EISA 总线是 32 位外围接口总线，存在于较早的 80386DX 和 80486 系统中。注意，这些总线中的每一种都与较早的类型兼容。也就是，8 位接口卡可工作在 8 位 ISA、16 位 ISA 或 32 位 EISA 标准总线上；同样，16 位接口卡可工作在 16 位 ISA 或 32 位 EISA 标准总线上。

另一种在许多基于 80486 的 PC 中出现的总线称为 **VESA** 局部总线或者 VL 总线。**局部总线**在局部总线一级将磁盘和视频显示器接口到微处理器，因此允许 32 位接口在与微处理器时钟相同的速度下工作。最近修改后的 VESA 局部总线支持 Pentium 微处理器的 64 位数据总线，并可直接与 PCI 总线竞争，虽然这意义不大。ISA 和 EISA 标准只能工作在 8MHz，使用这些标准降低了磁盘和视频接口的性能。PCI 总线是 32 位或 64 位，是专门为 Pentium 到 Pentium 4 微处理器设计的，其总线速度为 33MHz。

三种更新的总线出现在 ATX 类系统中。首先出现的是 **USB**（**universal serial bus，通用串行总线**）。通用串行总线通过串行数据通路和双绞线将键盘、鼠标、调制解调器和声卡之类的外围设备连接到微处理器。这种方案主要是通过减少导线数目来降低系统成本。另一个优点是音响系统可以由来自 PC 的独立电源供电，因此大大减少了噪声干扰。目前通过 USB 的数据传输速度对于 USB-1 为 10Mb/s，对于 USB-2 增加到 480Mb/s。

第二个新总线是用于视频显示卡的 **AGP**（**advanced graphics port，高级图形端口**）。高级图形端口在视频显示卡与微处理器之间高速传输数据（64 位数据通路的速度为 66MHz，或每秒 533MB），比任何其他总线或连接的速度都快，最近的 AGP 速度是 8X，即 2GB/s。这种视频显示子系统使得 PC 机能够接纳新型 DVD 游戏机。

最新出现的总线是 **SATA**（**SerialATA interface，串行 ATA 接口**）和用于视频显示卡的 PCI Ex-

press 总线。SATA 以 150MB/s 的速率在 PC 与硬盘之间传输数据，而 SATA-2 的速率是 300MB/s。最终 SATA 标准将提高到 450MB/s。目前 PCI Express 总线的视频显示卡可达到 16 倍的速度。

TPA

临时程序区（TPA）驻留 **DOS（disk operating system）**操作系统和其他控制计算机系统的程序。TPA 是一个 DOS 概念，在 Windows 中是不适用的。TPA 也存放任何当前激活的或者非激活的 DOS 应用程序，TPA 的容量为 640KB。正如前面提到的，该存储区驻留 DOS 操作系统，它要求分配一部分 TPA，以便工作。实际上，如果使用的操作系统为 MSDOS[⊖] 的 7. x 版，则为应用软件剩余的存储区约为 628KB。早期的 DOS 版本占用更多的 TPA 区，往往只留给应用程序 530KB 或更少。图 1-8 展示了运行 DOS 的计算机系统中 TPA 的组织方式。

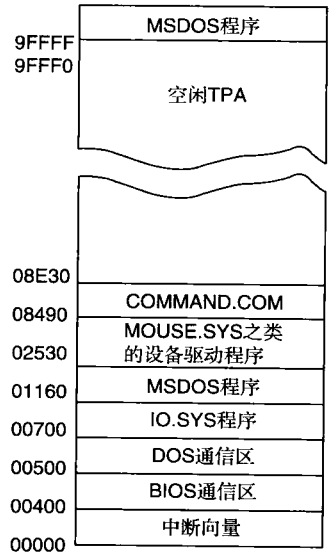


图 1-8 PC 中的 TPA 存储器映像

注：这个映像随系统的不同而不同。

DOS 存储器映像图给出了 TPA 的哪些区域用于系统程序、数据和驱动程序，也表明还有很大的存储区可用于应用程序。每个区域左边的十六进制数表示该存储区的起始和结束的地址。十六进制的**存储器地址**或**存储单元**用于表示存储系统每个字节的号数（**十六进制数**是一种数的表示法，它以 16 为基，即以 16 为底，它的每一位数为从 0 ～ 9 和 A ～ F 中的一个值。我们通常将 H 写在十六进制数的末尾，表示它是十六进制的值。例如，1234H 表示十六进制的 1234。还可用 0x1234 表示）。

中断向量访问 DOS、**BIOS（basic I/O system，基本 I/O 系统）**和应用程序的各种特性。BIOS 是存储在只读存储器（ROM）或快闪存储器中的程序集，用于操作连到计算机系统上的许多 I/O 设备。系统 BIOS 和 DOS 通信区包含程序访问 I/O 设备的临时数据和计算机系统的内部特征，这些数据存储在 TPA 区，因此它们能随系统的操作而变化。

每当 MSDOS 系统启动时，都要将 IO. SYS 程序从磁盘装入 TPA 中。IO .SYS 包含一些程序，允许 DOS 使用键盘、视频显示器、打印机和其他计算机中常见的 I/O 设备。IO .SYS 程序将 DOS 与 BIOS ROM 中存储的程序链接到一起。

设备驱动程序区的大小和设备驱动程序的数目随计算机的不同而不同。**设备驱动程序**是控制可安装 I/O 设备的程序，如鼠标、磁盘高速缓冲存储器、手持扫描器、**CD-ROM 存储器（Compact Disk Read-Only Memory，压缩光盘只读存储器）**、DVD（**Digital Versatile Disk，数字多用途盘**）或其他可安装设备及程序。DOS 设备驱动程序是带有扩展符 .SYS 的标准文件，如 MOUSE .SYS；而对于 DOS 3. 2 及其以后的版本，扩展符为 .EXE，如 EMM386 .EXE。注意，尽管 Windows 不使用这些文件，但是它们仍然可用于 Windows 下运行 DOS 的应用程序，即使在 Windows XP 下也是如此。Windows 使用称为 SYSTEM .INI 的文件，装载由 Windows 使用的驱动程序。近期的 Windows 版本，例如 Windows XP，增加了注册表的内容，包括系统和系统使用的驱动程序方面的信息。用 REGEDIT 程序可以查看注册表内容。

当在 DOS 模式下操作时，COMMAND .COM 程序，即**命令处理程序**，控制键盘命令的操作。COMMAND .COM 程序处理从键盘输入的 DOS 命令。例如键入 DIR，则 COMMAND .COM 程序显示当前磁盘目录下的磁盘文件目录。如果删除了 COMMAND .COM 程序，则 DOS 模式下就不能通过键盘使用计算机了。千万不要为了给其他软件腾出空间而删除 COMAND .COM、IO .SYS 或 MSDOS .SYS，否则计算机将不能工作。

⊖ MSDOS（Microsoft Disk Operating System）是微软公司的注册商标，7. x 版安装在 Windows XP 中。

系统区

系统区虽然比 TPA 区小，但它的确是同等重要的。系统区包括只读存储器（ROM）或快闪存储器中的程序，以及读/写存储器（RAM）的数据区。图 1-9 给出了典型个人计算机系统的系统区。如同 TPA 映像图那样，这个映像图也包括各个区域的十六进制存储器地址。

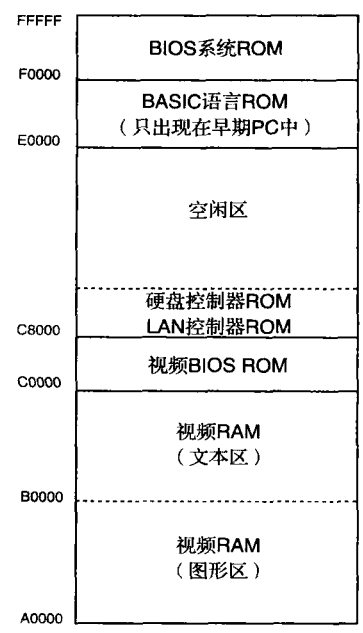


图 1-9 典型 PC 的系统区

系统空间的第一个区域包括视频显示 RAM 和在 ROM 或快闪存储器中的视频显示控制程序。这个区域通常起始于 A0000H 地址并延伸到 C7FFFH 地址，其存储器容量取决于系统配置的视频显示适配器的类型。通常，位于 A0000H ～ AFFFFH 的视频显示 RAM 区域存放图形或位映像数据，而位于 B0000H ～ BFFFFH 的存储区域存放文本数据。装入 ROM 或快闪存储器中的视频显示 BIOS 位于 C0000H ～ C7FFFH 地址的区域，包括控制 DOS 视频显示的控制程序。

位于 C8000H ～ DFFFFH 的区域通常是开放的，即空闲的。该区用于 PC 或 XT 系统中的扩展内存系统（EMS），或者 AT 系统的上位内存系统。它的使用取决于系统及其配置。扩展内存系统允许应用程序使用 64KB 存储器页帧。这个 64KB 页帧（通常定位于 D0000H ～ DFFFFH）用于扩展内存系统，通过将存储器页从 EMS 转换到该存储器地址范围进行扩展。

IBM 早期的 PC 系统中，位于 E0000H ～ EFFFFH 的存储器区域包含存储在 ROM 中的 BASIC 语言。在新型计算机系统中，这个区域通常是开放的或空闲的。

最后，系统 BIOS ROM 定位于系统区顶端的 64KB（F0000H ～ FFFFFH）区域。该 ROM 控制连接到计算机系统的基本 I/O 设备操作，但不控制视频显示系统的工作，视频显示系统有它自己的 ROM，位于 C0000H。系统 BIOS 的第一部分（F0000H ～ F7FFFH）包含启动计算机的程序，而第二部分包含控制基本 I/O 系统的过程。

Windows 系统

现代计算机使用与图 1-8 和图 1-9 所示的 DOS 存储器映像图不同的 Windows 存储器映像图。图1-10 为 Windows 存储器映像图，它有两个主要的区域：一个 TPA 区和一个系统区。它和 DOS 存储器映像图之间的差别是这些区域的大小和位置。

Windows TPA 位于存储系统从 00000000H 单元到 7FFFFFFFH 单元的第一个 2GB。Windows 系统区域位于存储器从 80000000H 单元到 FFFFFFFFH 单元的最后 2GB。似乎过去构造 DOS 存储器映像图的想法，也用在现代基于 Windows 的系统中。系统 BIOS 和视频存储器被定位在系统区里。在系统区里还定位了实际的 Windows 程序和驱动程序。为 Windows 写的每个程序都能使用位于线性地址 00000000H ～ 7FFFFFFFH 的 2GB 存储器。在 64 位系统中也是这样，允许访问更多存储器，但不能直接作为 Windows 的一部分。超过 2GB 的信息必须从存储器其他区交换到 Windows TPA 区。这些在将来的 Windows 和 Pentium 版本中大概会改变。目前的 Windows 64（Windows Vista 是其一部分）支持到 8GB 的 Windows 存储器。

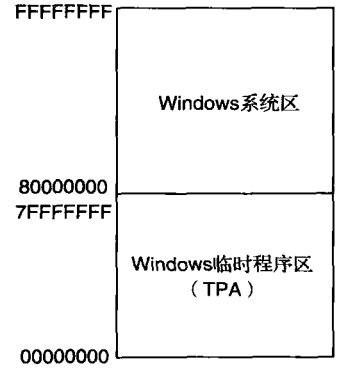


图 1-10 Windows XP 所用的存储器映像图

这意味着任何为 Windows 编写的程序将从物理地址 00000000H 开始吗？不是，存储系统的物理地址映像图和图 1-10 所示的线性编程模型是不同的。在 Windows Vista、Windows XP 或 2000 系统中的每个进程有它自己的页表，它定义进程

的每4KB页在物理存储器的具体位置。这种方法可使页位于存储器的任何地方，甚至在不连续的页。微处理器的页表和分页机制随后时候在本章讨论，此时超出了讨论范围。就某个应用程序来说，即使计算机存储器少一些，你也总是有2GB的存储器可用。操作系统（Windows）负责把物理存储器分配给应用程序，并且如果物理存储器不够，它就使用硬盘来改善虚拟存储器。

I/O空间

计算机系统中的I/O（输入/输出）空间从I/O端口0000H延伸到端口FFFFH（一个I/O端口地址类似于一个存储器地址，只是它不寻址存储器而寻址I/O设备）。I/O设备允许微处理器与外部设备通信。I/O空间允许计算机访问多达64K个不同的8位I/O设备、32K个不同的16位I/O设备，或16K个不同的32位I/O设备。64位可扩展支持32位版本的相同I/O空间和I/O大小，但是系统中没有添加64位I/O设备。这些地址中很多用于大部分计算机系统的扩展。图1-11显示了许多PC系统中



图1-11 典型PC中的I/O位置

的 I/O 映像表。在 Windows 下，要查看计算机的 I/O 映像表，可逐步点击："控制面板"、"性能和维护"、"系统"、"硬件"栏、"设备管理器"、"查看"栏，然后按类型选择"依类型排序资源"，再点击"输入/输出（I/O）"旁边的"+"号。

I/O 区主要包括两个部分，低于 0400H 地址的 I/O 区域是为系统设备保留的，多数已在图 1-10 中描述了。剩余的区域是可用于扩展的 I/O 空间，从 I/O 端口 0400H 到 FFFFH。通常，0000H～00FFH 地址区域用于寻址计算机主板上的器件，而 0100H～03FFH 地址区域寻址位于插卡上或主板上的器件。注意，IBM 规定，原 PC 标准限定 I/O 地址区域为 0000H～03FFH。使用 ISA 总线时，只能使用 0000H～03FFH 之间的地址，PCI 总线使用 0400H～FFFFH 之间的地址。

通常并不直接访问控制系统操作的各种 I/O 设备，而由系统 BIOS ROM 寻址这些基本设备，这些设备的地址和功能在不同的计算机中可能有些区别。大多数 I/O 设备的访问总是通过 Windows、DOS 或 BIOS 功能调用实现的，以便保持不同计算机之间的兼容性。图 1-11 作为一个向导说明了系统中的 I/O 空间。

1.2.2 微处理器

基于微处理器的计算机系统的核心是微处理器集成电路。微处理器是计算机系统的控制单元，有时也称为 **CPU**（**central processing unit，中央处理器**）。微处理器通过称为总线的一组连线控制存储器和输入/输出操作。总线选择 I/O 或存储器设备，在 I/O 设备或存储器与微处理器之间传送数据，并且控制 I/O 和存储系统。通过微处理器执行存储在存储器中的指令，即可实现对存储器和 I/O 的控制。

微处理器为计算机系统完成三项主要任务：1）在处理器与存储器或者 I/O 之间传送数据；2）简单的算术和逻辑运算；3）通过简单的判定控制程序的流向。虽然这是一些简单的工作，但实际上正是通过它们微处理器才能够完成任何操作或任务。

微处理器的强大威力在于它能够每秒执行上亿条指令，这些指令组成的程序或软件（**指令组**）存储在存储系统中。这种存储程序的概念使微处理器和计算机系统成为功能强大的设备（前面讲过，Babbage 也曾想在他的分析机中使用存储程序的概念）。

表 1-4 给出了 Intel 系列微处理器执行的算术和逻辑运算。这些运算都是很基本的，然而通过它们可以解决复杂的问题。数据从存储系统或内部寄存器中取出来进行运算。数据宽度是可变的，包括**字节**（8 位）、**字**（16 位）和**双字**（32 位）。注意，只有 80386～Pentium 4 直接处理 8 位、16 位和 32 位的数据。早期的 8086～80286 处理 8 位和 16 位数，而没有 32 位数。从 80486 开始，微处理器内包含了一个数字协处理器，允许用浮点数完成复杂的计算。在基于 8086～80386 的 PC 中，数字协处理器是附加的部件，类似于计算器芯片。数字协处理器也能完成**四字**（64 位）整数运算。在 Pentium～Pentium 4 里的 MMX 和 SIMD 部件并行地完成整数和浮点数功能，SIMD 部件要求数据按**八字**（128 位）长存储。

表 1-4 简单的算术和逻辑操作

操　作	说　明
加	
减	
乘	
除	
AND	逻辑乘
OR	逻辑加
NOT	逻辑反
NEG	算术取反
移位	
循环	

另一个使得微处理器功能强大的特征是，它具有以实际数值为基础进行简单判定的能力。例如，微处理器可以判定一个数是否为零、是否为正以及其他等等。这些简单判定使得微处理器可以改变程序的流向，好像程序可根据这些判定来思考一样。表 1-5 列出了 Intel 系列微处理器可以做出的判定。

表 1-5 8086～Core2 微处理器的判定

判　断	说　明
零	测试数是零或者不是零
符号	测试数是正还是负
进位	测试加法的进位或者减法的借位
奇偶	测试数中 1 的个数是奇数还是偶数
溢出	测试溢出，指示加法或减法后有符号数的结果无效

总线

总线是在计算机系统中互连各部件的一组公用导线，连接计算机系统各部分的总线负责在微处理器与它的存储器和 I/O 之间传送地址、数据和控制信息。在基于微处理器的计算机系统中，

有三种传送信息的总线：地址、数据和控制总线。图 1-12 表示这些总线是如何连接各个系统部件的，如微处理器、读/写存储器（RAM）、只读存储器（ROM 或快闪）和一些 I/O 设备。

地址总线请求存储器的一个存储单元或者 I/O 设备的一个 I/O 单元。如果寻址 I/O，则地址总线包含 0000H ～ FFFFH 的 16 位 I/O 地址。16 位 I/O 地址（或者端口号）选择 64K 个不同 I/O 设备中的一个。如果寻址存储器，则地址总线包含存储器地址，地址总线宽度随着微处理器的不同而变化。8086 和 8088 寻址 1MB 存储器，使用 20 位地址选择 00000H ～ FFFFFH 之间的单元。80286 和 80386SX 寻址 16MB 存储器，用 24 位地址选择 000000H ～ FFFFFFH 之间的单元。80386SL、80386SLC 和 80386EX 寻址 32MB 存储器，用 25 位地址选择 0000000H ～ 1FFFFFFH 之间的单元。80386DX、80486SX 和 80486DX 寻址 4GB 存储器，用 32 位地址选择 00000000H ～ FFFFFFFFH 之间的单元。Pentium 也寻址 4GB 存储器，但它使用 64 位数据总线，一次可访问多达 8 个字节的存储器。Pentium Pro ～ Core2 有一条 64 位数据总线和一条 32 位地址总线，寻址位于 00000000H ～ FFFFFFFFH 地址之间的 4GB 存储器，或者用 36 位地址总线寻址位于 000000000H ～ FFFFFFFFH 地址之间的 64GB 存储器，这取决于它们的配置。表 1-6 列出了全部 Intel 系列微处理器的总线宽度和存储器容量。

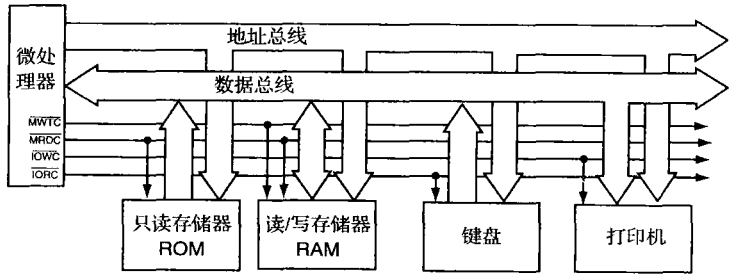


图 1-12 计算机系统中地址、数据和控制总线结构的框图

表 1-6 Intel 系列微处理器的总线和存储器容量

微处理器	数据总线宽度	地址总线宽度	存储器容量	微处理器	数据总线宽度	地址总线宽度	存储器容量
8086	16	20	1M	80386DX	32	32	4G
8088	8	20	1M	80386EX	16	26	64M
80186	16	20	1M	80486	32	32	4G
80188	8	20	1M	Pentium	64	32	4G
80286	16	24	16M	Pentium Pro ～ Core2	64	32	4G
80386SX	16	24	16M	Pentium Pro ～ Core2（如允许扩展寻址）	64	36	64G
				Pentium 和 Core2	64	40	1T
				64 位可扩展的 Itanium	128	40	1T

64 位扩展的 Pentium 系列在其现在的版本中提供 40 个地址管脚，允许通过其 10 位十六进制地址访问多达 1TB 的存储器。注意，2^{40} 是 1T（Tera）。在将来 64 位微处理器修订版中，Intel 计划扩大地址位数到 52 位，并且最终到 64 位。52 位地址总线可访问 4PB（Peta）的存储器，而 64 位地址总线可访问 16EB（百亿亿）的存储器。

数据总线在微处理器与它的存储器和 I/O 地址空间之间传送信息。Intel 微处理器系列各个成员传送数据的宽度各不相同，从 8 位到 64 位宽。例如，8088 有 8 位数据总线，一次传送 8 位数据；8086、80286、80386SL、80386SX 和 80386EX 通过其数据总线传送 16 位数据；80386DX、80386SX 和 80486DX 传送 32 位数据；Pentium ～ Core2 传送 64 位数据。较宽数据总线的优势在于能够提高数据传送的速度。例如，如果一个 32 位数存储在存储器中，要获取它，8088 微处理器需要 4 次传送操作才能完成，因为它的数据总线只有 8 位宽。80486 完成同样的工作只需一次传送，因为它的数据总线是 32 位宽。图 1-13 给出了 8086 ～ 80486 和 Pentium ～ Core2 微处理器的存储器宽度和容量。注意 Intel 微处理器各个成员之间在存储器容量和组织结构方面有什么差别。所有微处理器的存储器容量都按字节计算。注意 Pentium ～ Core2 具有 64 位宽的数据总线。

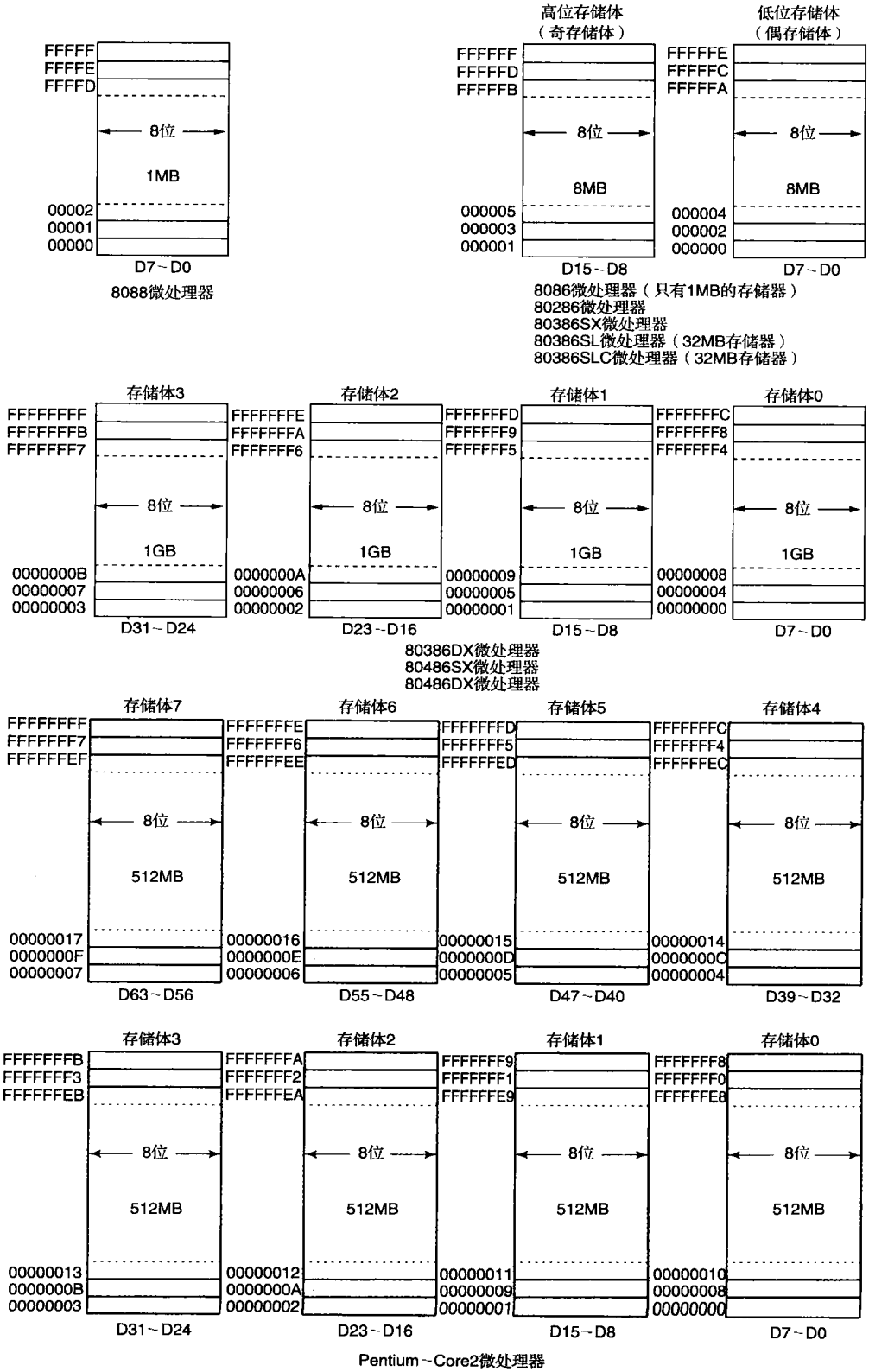


图 1-13　8086 ~ Core2 微处理器系列的物理存储系统

控制总线用于选择存储器或 I/O 并使它们完成读或写操作。大多数计算机系统都有 4 条控制连线：$\overline{\text{MRDC}}$（**存储器读控制**）、$\overline{\text{MWTC}}$（**存储器写控制**）、$\overline{\text{IORC}}$（**I/O 读控制**）和$\overline{\text{IOWC}}$（**I/O 写控制**）。注意，上面的横线表示控制信号是低电平有效，也就是说，当逻辑 0 出现在控制线上时，它才起作用。例如，如果$\overline{\text{IOWC}}=0$，则微处理器将通过数据总线向一个 I/O 设备写数据，该设备的地址出现在地址总线上。需要注意的是，这些控制信号的名字在微处理器的不同版本有稍微的不同。

微处理器读取一个存储单元的内容时，通过地址总线向存储器发出一个地址，然后发出存储器读控制信号（MRDC），以便从目的存储器读取数据，最后将从存储器读出的数据通过数据总线送到微处理器。当存储器写、I/O 写或 I/O 读出现时，也按同样的顺序依次进行，区别是：发出的是写操作控制信号，并且数据通过数据总线从微处理器流出。

1.3 数制

使用微处理器需要掌握二进制、十进制和十六进制数制系统的基本知识。本节为那些不熟悉数制系统的读者提供这方面的背景知识，说明了十进制与二进制之间、十进制与十六进制之间，及二进制与十六进制之间的转换。

1.3.1 数字

将数从一种数制向另一种数制转换之前，必须了解数制系统中的数字。在我们的早期教育中，已学习了十进制数（以 10 为基的数），它由 10 个数字组成：0 到 9。任何数制的第一个数字总是零。例如，以 8 为基的数（**八进制**）包含 8 个数字：0 到 7；而以 2 为基的数（**二进制**）包含 2 个数字：0 和 1。如果基数大于 10，则其余的数字用从 A 开始的字母表示，例如，以 12 为基的数包含 12 个数字：0 到 9，之后用 A 代表 10，B 代表 11。注意，以 10 为基的数不包含数字 10，如同以 8 为基的数不包括数字 8 一样。计算机中最通用的数制系统是十进制、二进制、八进制和十六进制（基为 16）。每种数制都将在本节中进行说明和使用。

1.3.2 按位计数法

一旦我们理解了数制系统中的数字后，就可用按位计数法构造更大的数值。在小学时我们都学过个位的左边一位是十位，十位左边一位是百位，以此类推（例如十进制数 132，这个数有 1 个百位，3 个十位和 2 个个位）。或许我们没有学过每个位的指数值：个位的权为 10^0，即 1，十位的权为 10^1 或 10，而百位的权为 10^2 或 100。在理解其他数制中的数时，以位的指数幂表示是个关键。**基数**（**number base**）小数点（在十进制中称为十进制小数点）左边的位在任何数制中都是个位。例如，二进制小数点左边的位是 2^0 或 1，而八进制小数点左边的位是 8^0 或 1。在任何情况下，任何数的零次幂总是 1 或个位。

个位左边的位总是基数的 1 次幂。在十进制系统中是 10^1，或 10；在二进制中是 2^1，或 2；而在八进制中是 8^1，或 8。因此，十进制的 11 与二进制的 11 相比有不同的值。十进制 11 表示一个 10 加上一个 1，值为 11；二进制 11 表示一个 2 加上一个 1，值为 3；八进制表示的 11，其值为 9。

在十进制系统中，对于十进制小数点右边的位，它的幂为负数。十进制小数点右边第一位数的值为 10^{-1}，或 0.1。在二进制中，二进制小数点右边第一位数的值为 2^{-1} 或 0.5。一般来说，十进制使用的计数法可以用于任何其他数制。

例 1-1 给出了一个二进制数 110.101（通常写成 110.101_2），也给出了这个数每个位的幂、权和值。为了把二进制数转换为十进制，将每位数字的权相加，就得到了它的等效十进制值。二进制 110.101 等于十进制的 6.625（4 + 2 + 0.5 + 0.125）。注意，这个和的整数部分是由 2^2（4）加 2^1（2）构成，没有用 2^0（1）是因为这个位的数为零。小数部分由 2^{-1}（0.5）加 2^{-3}（0.125）构成，但是没有用 2^{-2}（0.25）。

例 1-1

| | | |

例 1-5

```
幂      16^1   16^0   16^-1
权      16     1      .0625
数      6      A      .C
数值    96  +  10  +  .75  = 106.75
```

1.3.4 十进制转换成其他进制

由十进制转换成其他进制比其他进制转换成十进制困难。转换十进制整数部分时，要除以基数；转换分数部分时，要乘以基数。

转换十进制整数部分

将十进制整数转换成其他数制时，要除以基数并保存余数，作为结果的有效数字。这种转换的算法如下：

1）十进制数除以基数。

2）保存余数（最先得到的余数是最低有效位数字）。

3）重复步骤 1 和 2，直到商为零。

例如，将十进制的 10 转换成二进制，要除以 2，结果为 5，余数为 0。第一个余数是结果的个位（此例中是 0）。接下来用 5 除以 2，结果为 2，余数为 1，则 1 是第二位（2^1）的值。继续做除法，直到商为零。例 1-6 给出了这个转换过程。从下向上读，这个结果为 1010_2。

例 1-6

```
2)  10      余数 = 0
 2)  5      余数 = 1
  2)  2     余数 = 0
   2)  1    余数 = 1        结果 = 1010
       0
```

将十进制的 10 转换成八进制，要除以 8，如例 1-7 所示。十进制的 10 是八进制的 12。

例 1-7

```
8)  10      余数 = 2
 8)  1      余数 = 1        结果 = 12
     0
```

从十进制转换到十六进制要除以 16 来完成。余数在 0～15 之间，而 10～15 的余数要转换为十六进制的字母 A～F。例 1-8 给出了十进制数 109 到十六进制数 6DH 的转换。

例 1-8

```
16)  109    余数 = 13 (D)
 16)  6     余数 = 6        结果 = 6D
      0
```

转换十进制小数部分

转换十进制小数部分是通过乘以基数来完成的。例如，要将十进制小数转换成二进制，要乘以 2。乘法之后，乘积的整数部分保存起来作为结果的一个有效位，剩余的小数部分再乘以基数 2。当剩余的小数部分为 0 时，乘法结束。有些数可能永远不会结束，即余数总不为 0。转换十进制小数部分的算法如下：

1）用基数乘以十进制小数。

2）保存结果的整数部分（即使是 0）作为一位数字。注意，第一个得到的结果写在紧挨着小数点的右边。

3）用步骤 2 的小数部分重复步骤 1 和 2，直到步骤 2 的小数部分是 0。

假如要将十进制的0.125转换成二进制。完成这个转换要乘以2，如例1-9所示。注意，乘法直到小数部分为0时才停止。本例的结果为二进制小数0.001。

例1-9

```
  .125
×   2
 0.25        数为0
  .25
×   2
  0.5        数为0
  .5
×   2
  1.0        数为1，结果 = 0.001₂
```

同样的方法也适用于将十进制数转换成其他任何进制的数。在例1-10中，将例1-9中的十进制小数0.125转换成八进制时，要乘以8。

例1-10

```
  .125
×    8
   1.0       数为1，结果 = 0.1₈
```

例1-11给出了转换成十六进制小数的例子。这里用16乘以十进制数0.046875，以便转换为十六进制。注意，0.046875转换成十六进制是0.0CH。

例1-11

```
  .046875
×      16
    0.75     数为0
  .75
×   16
  12.0       数为12（C），结果为十六进制的0.0C₁₆
```

1.3.5 二进制编码的十六进制

二进制编码的十六进制（Binary-Coded Hexadecimal，BCH）数是用二进制编码表示的十六进制数据。二进制编码的十六进制数是将十六进制数的每一位都用4位二进制数表示。表1-7给出了BCH数的值。

表1-7 二进制编码的十六进制（BCH）码

十六进制数	BCH码	十六进制数	BCH码	十六进制数	BCH码	十六进制数	BCH码
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

用BCH表示十六进制数时，将每个十六进制数字都转换成BCH数，并且每个数位之间用空格分开。例1-12显示了如何将2AC转换成BCH数，注意每个BCH数之间用空格分开。

例1-12

```
2AC = 0010  1010  1100
```

*BCH码的目的在于能将十六进制数以二进制的形式写出，使BCH数与十六进制数之间的转换很容易。*例1-13显示了如何将BCH数转换为十六进制数。

例 1-13

```
1000 0011 1101 . 1110 = 83D.E
```

1.3.6 补码

有时，数据以补码的形式存储，以便表示负数。有两种表示负数的方式：**补码（基数的补）**和**反码（基数减 1 的补）**。最早的方式是反码，为了得到负数的反码表示，用基数 - 1 减去该数的每一个数位上的数字。

例 1-14 显示了如何将 8 位二进制数 01001100 对 1 取补（基数减 1 的补），以便表示成一个负数。注意，用 1 减去该数的每一位数字，以便生成反码。在此例中，01001100 的负数是 10110011。同样的技术可适用于任何数制，如例 1-15 所示，十六进制数 5CD 的反码是从 15（基数 - 1）中减去它的每一位数字得到的。

例 1-14

```
  1 1 1 1  1 1 1 1
- 0 1 0 0  1 1 0 0
  1 0 1 1  0 0 1 1
```

例 1-15

```
  15  15  15
-  5   C   D
   A   3   2
```

如今，反码不单独使用，而作为求补码的一个步骤使用，补码是当代计算机系统表示负数的方法（反码用于早期的计算技术中）。反码的主要问题是它存在负零或者正零，而补码系统中只能存在正零。

为得到补码，先求反码，然后将 1 加到结果上。例 1-16 显示了如何通过对 2（基为 2）取补的方式，将数 0100 1000 转换成负数。

例 1-16

```
  1 1 1 1  1 1 1 1
- 0 1 0 0  1 0 0 0
  1 0 1 1  0 1 1 1    (1 的补码)
+ _______        1
  1 0 1 1  1 0 0 0    (2 的补码)
```

为验证 0100 1000 是 1011 1000 的反（负数），将两者相加得到一个 8 位结果。丢掉第 9 位数字，结果是零，因为 0100 1000 是正数 72，而 1011 0111 是负数 72。同样的技术可用于任何数制。例 1-17 表示如何求十六进制数 345 的负数，首先求该数 15 的补，然后将 1 加到结果上，得到 16 的补。与前面类似，如果把原来的 3 位数 345 加上其负数 CBB，则结果是 3 位 000。丢掉第 4 位（进位）。这证明了 345 是 CBB 的反。关于 1 的补和 2 的补，更进一步的资料将在下一节介绍有符号数时给出。

例 1-17

```
  15  15  15
-  3   4   5
   C   B   A      (15 的补码)
+ _______   1
   C   B   B      (16 的补码)
```

1.4 计算机数据格式

成功的程序设计者需要清晰地理解数据格式。本节将详细说明许多通用计算机使用的数据格式，它们与 Intel 系列微处理器使用的数据格式一样。通常，数据以 ASCII、Unicode、BCD、有符号和无符号整数，或者浮点数（实数）的形式出现。也可以使用其他格式，但是这里不予说明，因为它们不通用。

1.4.1 ASCII 和 Unicode 数据

ASCII（American Standard Code for Information Interchange，美国标准信息交换码）数据表示计算机系统存储器中的字母数字符号，参见表 1-8。标准 ASCII 码是 7 位代码，它的第 8 位，即最高有效位，在某些已过时的系统中用于保存其奇偶性。如果 ASCII 数据用于打印机，则其最高有效位为 0 时，进行字符打印；最高有效位为 1 时，进行图形打印。在 PC 中，将逻辑 1 放在最左边的位上用于选择扩展的 ASCII 字符集。表 1-9 列出了使用代码 80H ～ FFH 的扩展 ASCII 字符集。扩展的 ASCII 字符保存一些非英文字母和标点、希腊字符、算术字符、图框符及其他特殊字符。注意，扩展字符可能随打印机不同而不同。这个表提供的字符是为使用 IBM ProPrinter 打印机设计的，它也与某些字处理程序的专用字符集相匹配。

表 1-8 ASCII 码

	第二位															
	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
第一位																
0X	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1X	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EMS	SUB	ESC	FS	GS	RS	US
2X	SP	!	"	#	$	%	&	'	(	)	*	+	,	−	.	/
3X	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4X	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5X	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6X	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7X	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂

表 1-9 由 IBM ProPrinter 打印的扩展的 ASCII 码

第一位	第二位															
	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X		☺	☻	♥	♦	♣	♠	•	◘	○	■	♂	♀	♪	♫	☼
1X	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
8X	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9X	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
AX	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
BX	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
CX	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
DX	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
EX	α	ß	Γ	π	Σ	σ	µ	γ	Φ	Θ	Ω	δ	∞	φ	∈	∩
FX	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	n	2	■	

ASCII 控制符也列入了表 1-8 中，它们在计算机系统中实现控制功能，包括清除屏幕、退格、换行，等等。为了从计算机键盘输入控制码，键入字母时要按着 Ctrl 键。为得到控制码 01H，键入 Ctrl-A，为了得到 02H，键入 Ctrl-B，等等。注意，在 DOS 提示符下，控制代码出现在屏幕上，Ctrl-A 显示为 ^A，Ctrl-B 显示为 ^B，以此类推。还要注意，在许多现代键盘上回车码（CR）就是“回车”键。CR 的作用是使光标或打印头返回到最左边。另一个在许多程序中都出现的代码是换行码（LF），它将光标下移一行。

为了使用表 1-8 或 1-9 将字符或控制字符转换成 ASCII 字符，首先定位那个要转换的字符，然后寻找十六进制 ASCII 码的第一位数字，再找第二位数字。例如，大写字母 A 的 ASCII 码是 41H，而小写字母 a 的 ASCII 码是 61H。从 Windows 95 开始，许多基于 Windows 的应用使用单一码制（Unicode）存

储字母数据。这里把每个字符存放成16位数据，代码0000H～00FFH和标准ASCII码相同，其余的代码0100H～FFFFH用于存放许多世界范围内采用的字符集构成的所有专用字符。这样，为Windows环境写的软件就可以在世界上许多国家内使用。

ASCII数据在存储器中存储时，通常用汇编程序的专用伪指令说明，这个伪指令称为“定义字节”(define byte)或DB（汇编程序是使用计算机本身的二进制机器语言编程的程序）。DB可以用单词BYTE取代。在例1-18中给出了DB和BYTE伪指令，还给出了与ASCII码字符串一起使用的几个例子。注意，每个字符串如何用撇符（'）括起来，千万不能用引号（"）。还要注意汇编程序将每个字符的ASCII码值列在了该字符串的左边。在最左边是十六进制存储器地址，这是存储在存储系统内的字符串的首地址。例如，字符串WHAT存在以001DH为起始地址的存储器内，并且先存储了第一个字母57（W），接下来是68（H），以此类推。例1-19表示使用Visual C++ Express 2005和2008用String^字符串来定义同样的三个字符串，注意，Visual C++使用引号把字符串括起来。如果用C++较早期的版本，Microsoft Visual C++用CString定义串，而不用String^。符号^表示该串是存储管理无用单元搜集堆的成员。当目标可见性消失或从C++程序作用域中消失时，无用单元搜集清除存储系统（释放无用存储器），并且它还可以预防存储器泄漏。

例1-18

```
0000  42 61 72 72 79    NAMES  DB   'Barry B. Brey'
      20 42 2E 20 42
      72 65 79
000D  57 68 65 20 63    MESS   DB   'Where can it be?'
      20 63 61 6E 20
      69 74 20 62 65
      3F
001D  57 69 20 74 20    WHAT   DB   'What is on first.'
      69 73 20 6F 6E
      20 66 69 72 73
      74 2E
```

例1-19

```
String^ NAMES = "Barry B. Brey"  // C++ Express version

String^ MESS = "Where can it be?"

String^ WHAT = "What is on first."
```

1.4.2 BCD数据

二进制编码的十进制（Binary-coded decimal，BCD）信息以压缩或者非压缩格式存储。**压缩BCD（packed BCD）**数据以每字节2位数字的形式存储，而**非压缩BCD（unpacked BCD）**数据以每字节1位数字的形式存储。BCD数的范围是0000_2～1001_2，或十进制数0～9。非压缩BCD数常常从键盘或数字小键盘返回，而压缩BCD数用于微处理器指令系统内的某些指令，包括BCD加法和减法。

表1-10列出了一些十进制数转换成的压缩和非压缩两种格式的BCD数。需要应用BCD数的有销售终端或其他任何实现少量简单运算的设备。如果系统要求复杂的算术运算，则很少用BCD数，因为没有简单有效的方法完成复杂的BCD运算。

表1-10 压缩和非压缩BCD数据

十进制数	压缩格式	非压缩格式
12	0001 0010	0000 0001 0000 0010
623	0000 0110 0010 0011	0000 0110 0000 0010 0000 0011
910	0000 1001 0001 0000	0000 1001 0000 0001 0000 0000

例 1-20 给出了如何用汇编语言定义压缩和非压缩 BCD 数。例 1-21 显示如何用 Visual C++ 和“char”或“bytes”定义同样的数。在所有情况下，都遵循首先存储最低有效数据的规则。这意味着将 83 存储到存储器中时，先存储 3，然后存储 8。对于压缩数据，字母 H（十六进制）跟在数字后面，以保证汇编程序存储的是 BCD 值而不是压缩 BCD 数的十进制值。注意怎样在存储器内存储非压缩数据，每字节一位；或者压缩数据，每字节 2 位。

例 1-20

```
                    ;非压缩 BCD 数（最低有效位在先）
                    ;
0000  03 04 05  NUMB1  DB    3,4,5        ;定义数 543
0003  07 08     NUMB2  DB    7,8          ;定义数 87
                    ;
                    ;压缩 BCD 数（最低有效位在先）
                    ;
0005  37 34     NUMB3  DB    37H,34H      ;定义数 3437
0007  03 45     NUMB4  DB    3,45H        ;定义数 4503
```

例 1-21

```
                    //非压缩 BCD 数（最低有效位在先）
                    //
                    char Numb1 = 3,4,5       ;定义数 543
                    char Numb2 = 7,8         ;定义数 87
                    //
                    //压缩 BCD 数（最低有效位在先）
                    //
                    char Numb3 = 0x37,0x34   ;定义数 3437
                    char Numb4 = 3,0x45      ;定义数 4503
```

1.4.3 字节数据

字节数据以无符号和有符号的整数形式存储。图 1-14 形象地说明了字节整数的无符号和有符号两种形式。它们之间的区别是最左边位的权，对于无符号整数，其值是 128；而对于有符号整数，其值是 -128。在有符号整数格式中，最左位表示数的符号，以及 -128 的权。例如 80H，作为无符号数，表示 128；而作为有符号数，它表示 -128。无符号整数值的范围是从 00H 到 FFH（0～255），而有符号整数值的范围是从 -128～0～+127。



图 1-14 无符号和有符号字节，给出了每个二进制位的权

尽管有符号负数用这种方法表示，但是它以 2 的补码形式存储。用每位的权求有符号数数值的方法，比用对该数求 2 的补来求值要更加容易。在为程序员设计的运算器领域中尤其如此。

每次对一个数求 2 的补时，它的符号从 - 变为 +，或者从 + 变为 -。例如，数 00001000 是 +8，通过对 +8 求 2 的补得到它的负数（-8）。为求 2 的补，先求该数 1 的补（反码），然后再加 1。为求一个数 1 的补，只要将该数的每一位取反，即从 0 变 1 或者从 1 变 0。一旦得到 1 的补，只要将 1 的补加上 1 就可得到 2 的补。例 1-22 给出了如何使用这种方法对一个数求 2 的补。

例 1-22

```
+8 = 00001000
     11110111 (1 的补)
+         1
-8 = 11111000 (2 的补)
```

另一种可能更简单的求 2 的补的方法是从最右位数字开始。从右向左写下该数，抄写原数字直到出现第一个 1，第一个 1 照抄，然后将剩余的所有位都取反。例 1-23 采用这种技术对与例 1-22 同样的数求 2 的补。

例 1-23

```
+8 = 00001000
         1000 (抄写该数至第一个 1)
     1111     (剩下的位取反)
-8 = 11111000
```

为了使用汇编程序将 8 位数据存到存储器中，可以如同前面例子一样使用 DB 伪指令，或像在 Visual C++ 的例子中那样，使用“char”语句。例 1-24 列出了许多用汇编程序在存储器内存储 8 位数的格式。注意，在这个例子中，十六进制的数用数字后面跟随字母 H 表示，而十进制数后不跟任何字母。例 1-25 给出了同样的字节数据用于 Visual C++ 程序时的定义。在 C/C++ 中十六进制以 0x 的形式表示一个十六进制的数。

例 1-24

```
          ;无符号字节数据
          ;
0000 FE   DATA1 DB   254     ;定义十进制数 254
0001 87   DATA2 DB   87H     ;定义十六进制数 87
0002 47   DATA3 DB   71      ;定义十进制数 71
          ;
          ;有符号字节数据
          ;
0003 9C   DATA4 DB   -100    ;定义十进制数 -100
0004 64   DATA5 DB   +100    ;定义十进制数 +100
0005 FF   DATA6 DB   -1      ;定义十进制数 -1
0006 38   DATA7 DB   56      ;定义十进制数 56
```

例 1-25

```
//无符号字节数据
//
unsigned char Data1 = 254;    //定义十进制数 254
unsigned char Data2 = 0x87;   //定义十六进制数 87
unsigned char Data3 = 71      //定义十进制数 71
//
//有符号字节数据
//
char Data4 = -100;            //定义十进制数 -100
char Data5 = +100;            //定义十进制数 +100
char Data6 = -1;              //定义十进制数 -1
char Data7 = 56;              //定义十进制数 56
```

1.4.4 字数据

一个字（16 位）由两个字节数据组成。其最低有效字节总是存储在最低地址存储单元中，而最高有效字节存储在最高地址存储单元中。这种存放数据的方法称为**小端（little endian）**格式。另一种方

法 Intel 系列微处理器不用，称作**大端**（**big endian**）格式。从大到小格式，依照最低地址单元存储最高位有效数字的方式存储数据。从大到小格式用于 Motorola 系列微处理器。图 1-15a 给出了一个字数据中每一位数的权，图 1-15b 给出了怎样在存储单元 3000H 和 3001H 中存储数字 1234H。有符号和无符号字之间惟一的区别是最左边的位。对于无符号格式，最左边的位是无符号的，其权是32768；对于有符号格式，这一位的权是 -32768。如同有符号字节数据一样，有符号字也以 2 的补码形式表示负数。还要注意，低位数据字节存储在最低地址存储单元（3000H），而高位数据字节存储在最高地址存储单元（3001H）。

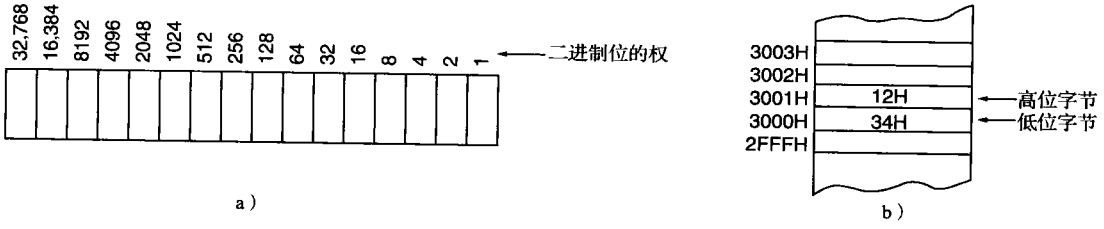


图 1-15　16 位字存储格式

a）在寄存器中的无符号字　b）在存储器的两个字节 3000H 和 3001H 存储单元的内容是字 1234H

例 1-26 给出了几个用汇编程序存储在存储器中的有符号和无符号字数据。例 1-27 表示在 Visual C++ 程序（假定为 5.0 或更新的版本）中如何存储同样的数据，这里用 **short** 伪指令存储 16 位整数。注意，**定义字**（**define word**）伪指令，即 **DW**，导致汇编程序在存储器中存储字数据，而不是前面例子中的字节。WORD 伪指令也用来定义字。注意，字数据通过汇编程序以输入时同样的格式显示。例如，1000H 由汇编程序显示为 1000。这是为了方便我们观看，因为实际上数据是以 00 10 的形式存储在存储器的两个连续字节中的。

例 1-26

```
                ;无符号字数据
                ;
0000  09F0    DATA1 DW   2544      ;定义十进制数2544
0002  87AC    DATA2 DW   87ACH     ;定义十六进制数87AC
0004  02C6    DATA3 DW   710       ;定义十进制数710
                ;
                ;有符号字数据
                ;
0006  CBA8    DATA4 DW   -13400    ;定义十进制数-13400
0008  00C6    DATA5 DW   +198      ;定义十进制数+198
000A  FFFF    DATA6 DW   -1        ;定义十进制数-1
```

例 1-27

```
//无符号字数据
//
unsigned short Data1 = 2544;     //定义十进制数2544
unsigned short Data2 = 0x87AC;   //定义十六进制数87AC
unsigned short Data3 = 710;      //定义十进制数710
//
//有符号字数据
//
short Data4 = -13400;            //定义十进制数-13400
short Data5 = +198;              //定义十进制数+198
short Data6 = -1;                //定义十进制数-1
```

1.4.5 双字数据

双字数据需要4个字节存储器，因为它是32位数。像乘法后的乘积或除法前的被除数都以双字数据出现。在80386～Core2中，存储器和寄存器都是32位宽。图1-16给出了双字在存储器中存储的格式以及每位二进制的权值。

当双字存储在存储器中时，它的最低有效字节存储在最低地址存储单元中，最高有效字节存储在最高地址存储单元中，采用从小到大格式。前面讲述的字数据也是这样存储的。例如，12345678H存储在存储单元00100H～00103H中，其中78H放在00100H单元中，56H存放在00101H单元，34H存放在00102H单元，12H存放在00103H单元。

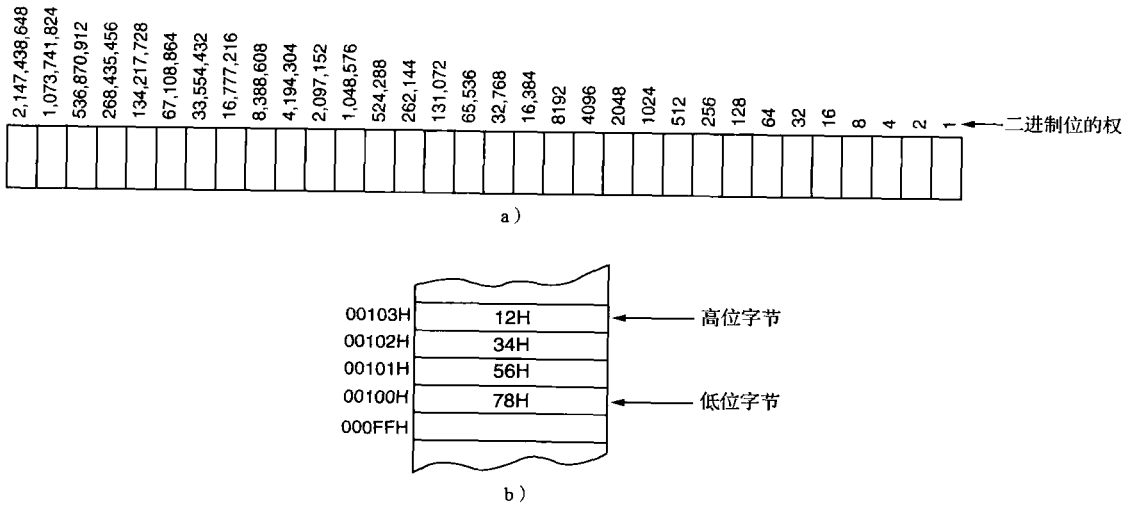


图1-16 32位双字存储格式

a）在寄存器中的无符号双字 b）在存储器中存储单元00100H～00103H中的内容是双字12345678H

为了定义双字数据，可用汇编程序的“**定义双字**”（**define doubleword**）伪指令，即**DD**（也可以用DWORD伪指令代替DD）定义双字。例1-28给出了使用DD伪指令将有符号数和无符号数据存入存储器中的例子。例1-29给出如何用**int**伪指令在Visual C++程序中定义同样的双字。

例1-28

```
                    ;无符号双字数据
                    ;
0000  0003E1C0  DATA1 DD  254400      ;定义十进制数254400
0004  87AC1234  DATA2 DD  87AC1234H   ;定义十六进制数87AC1234
0008  00000046  DATA3 DD  70          ;定义十进制数70
                    ;
                    ;有符号双字数据
                    ;
000C  FFEB8058  DATA4 DD  -1343400    ;定义十进制数-1343400
0010  000000C6  DATA5 DD  +198        ;定义十进制数+198
0014  FFFFFFFF  DATA6 DD  -1          ;定义十进制数-1
```

例1-29

```
//无符号双字数据
//
unsigned int Data1 = 254400;       //定义十进制数254400
unsigned int Data2 = 0x87AC1234;   //定义十六进制数87AC1234
unsigned int Data3 = 70;           //定义十进制数70
```

```
//
//有符号双字数据
//
int Data4 = -1343400;              //定义十进制数-1343400
int Data5 = +198;                  //定义十进制数+198
int Data6 = -1;                    //定义十进制数-1
```

任意宽的整数都可存入存储器中。这里列出的是标准的格式，但这不意味着一个256字节宽的整数就不能存入到存储器中。微处理器非常灵活，允许任意长的数据。当非标准宽度的数据存到存储器中时，通常是使用DB伪指令。例如，24位数123456H就可用DB 56H,34H,12H指令存储。注意，这也要服从于从小到大格式。在Visual C++程序中用char伪指令也可以这样做。

1.4.6 实数

因为许多高级语言使用Intel系列微处理器，所以常常会遇到实数。实数通常称为**浮点数**，它包括两个部分：尾数（有效小数）和指数（阶）。图1-17描述了存储在Intel系统中4字节的和8字节的实数。注意，4字节实数称为**单精度**实数，8字节实数称为**双精度**实数。这里给出的格式与IEEE[⊖]标准，即10.0版IEEE-754规定的格式相同。这个标准已作为实数的标准格式用于几乎所有高级程序设计语言和许多应用程序包中。这个标准也适用于PC中数字协处理器使用的数据。图1-17a给出了单精度的格式，包括一个符号位、8位阶（指数）和24位小数（尾数）。注意，因为实际应用中常常要求双精度浮点数，如图1-17b所示，所以具有64位数据总线的Pentium～Core2以两倍于80386/80486微处理器的速度进行存储器传送。

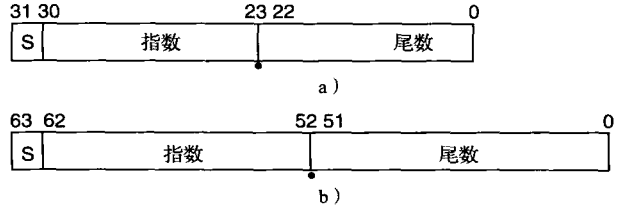


图1-17 浮点数
a）使用偏移7FH的单精度浮点数
b）使用偏移3FFH的双精度浮点数

简单的算术运算表明存储数据的三个部分要占用33位。事实并非如此，24位尾数包括1个默认**隐藏位**，因此允许只存储23位就可以表示24位尾数。默认的1位是规格化实数的第一位。当规格化一个数时，调整它使其值大于等于1而小于2。例如，如果将12转换为二进制数（1100_2），将它规格化并得结果为1.1×2^3。1不存储在23位尾数部分内，这个1是默认位。表1-11给出了这个数和其他一些数的单精度格式。

表1-11 单精度实数

十进制数	二进制数	非规格化数	符 号	移码阶	尾 数
+12	1100	1.1×2^3	0	10000010	10000000 00000000 00000000
-12	1100	1.1×2^3	1	10000010	10000000 00000000 00000000
+100	1100100	1.1001×2^6	0	10000101	10010000 00000000 00000000
-1.75	1.11	1.11×2^0	1	01111111	11000000 00000000 00000000
+0.25	0.01	1.0×2^{-2}	0	01111101	00000000 00000000 00000000
+0.0	0	0	0	00000000	00000000 00000000 00000000

阶是以**移码**（**biased exponent**）形式存储的。对于单精度格式的实数，偏移量为127（7FH）；而双精度格式下，偏移量为1023（3FFH）。存储浮点数阶码部分之前，偏移量要先加到阶码上。前两个例子中，阶为2^3，在单精度格式中，移码后的阶表示为127+3即130（82H）；在双精度格式中，它为1026（402H）。

浮点数规则有两个例外。数0.0存储为全零。无限大数的阶码部分存储为全1，尾数部分存储为全零。符号位指示正无限大或是负无限大。

⊖ IEEE代表电气电子工程师学会。

与其他数据类型一样，汇编程序可以将实数定义为单精度或双精度格式。因为单精度数是32位数，用DD伪指令定义；用“**定义四字**”（define quadword）或DQ伪指令定义64位的双精度数。可选的用于定义实数的伪指令是REAL4、REAL8和REAL10，分别定义单精度、双精度和扩展精度实数。例1-30给出了一些定义实数格式数据的例子。如果在VC++中使用内嵌汇编程序，则单精度数被定义成float型，双精度数被定义成**double**型，如例1-31所示。无法在VC++中定义扩展精度浮点数。

例1-30

```
                        ;单精度实数
                        ;
0000 3F9DF3B6           NUMB1   DD      1.234    ;定义1.234
0004 C1BB3333           NUMB2   DD      -23.4    ;定义-23.4
0008 43D20000           NUMB3   REAL4   4.2E2    ;定义420
                        ;
                        ;双精度实数
                        ;
000C 405ED9999999999A   NUMB4   DQ      123.4    ;定义123.4
0014 C1BB333333333333   NUMB5   REAL8   -23.4    ;定义-23.4
                        ;
                        ;扩展精度实数
                        ;
001C 4005F6CCCCCCCCCCCD NUMB6   REAL10  123.4    ;定义10字节实数123.4
```

例1-31

```
//单精度实数
//
float Numb1 = 1.234;
float Numb2 = -23.4;
float Numb3 = 4.3e2;
//
//双精度实数
double Numb4 = 123.4;
double Numb5 = -23.4;
```

1.5 小结

1）机械计算机时代开始于公元前500年出现的算盘。第一代机械计算器一直保持到1642年才有所改变，是Blaise Pascal对它进行了改进。一种早期的机械计算机系统是Charles Babbage于1823年开发的分析机。遗憾的是，这个机器从来没有实现过，因为当时不可能制造出必要的机械零件。

2）第一个电子计算机器是第二次世界大战期间由Konrad Zuse发明的，他是早期数字电子的开拓者。他的计算机，Z3，用于飞机和火箭设计。

3）第一台电子计算机采用真空管，于1943年投入运行，用于破译德国军事密码。这个第一台计算机系统，巨人，是由Alan Turing发明的。它的问题在于程序固定不能更改。

4）第一台通用可编程电子计算机于1946年在宾夕法尼亚大学开发成功。这个现代第一台计算机称为ENIAC。

5）第一种高级程序设计语言称为FLOWMATIC，是20世纪50年代初由Grace Hopper为UNIVAC I计算机开发的。它导致了FORTRAN以及COBOL之类的其他早期程序设计语言的产生。

6）世界上第一个微处理器，Intel 4004，是4位微处理器，是一种单片可编程控制器。按今天的标准它的功能非常不足，只能寻址4096个4位存储单元，它的指令系统只包含45条指令。

7）今天通用的微处理器包括8086/8088，是最早的16位微处理器。在这些早期16位机器之后又出现了80286、80386、80486、Pentium、Pentium Pro、Pentium Ⅱ、Pentium Ⅲ、Pentium 4和Core2处理器。其体系结构从16位改变成32位，很快又出现了64位的Itanium。每一种更新换代版本的出现，都使处理器的速度和性能有很大的改进。各种迹象表明，这种速度和性能的改进过程还将继续下去，尽管性能改进并不总是通过提高时钟频率而得到。

8）基于DOS的PC机存储系统包含三个主要区域：TPA（临时程序区）、系统区和扩展内存。TPA包含应用程序、

操作系统和驱动程序；系统区包含用于视频显示卡、磁盘驱动器和 BIOS ROM 的存储器。扩展内存只用于 AT 或 ATX 型的 PC 系统的 80286 ~ Core2 微处理器中。基于 Windows 的 PC 机存储系统包括两个主要区域：TPA 和系统区。

9）8086/8088 寻址从 00000H ~ FFFFFH 的 1MB 存储器。80286 和 80386SX 寻址 000000H ~ FFFFFFH 的 16MB 存储器。80386SL 寻址 0000000H ~ 1FFFFFFH 的 32MB 存储器。80386DX ~ Core2 处理器寻址 00000000H ~ FFFFFFFFH 的 4GB 存储器。另外，Pentium Pro ~ Core2 能够以 36 位地址运行，可寻址从 000000000H ~ FFFFFFFFFH 地址的高达 64GB 的存储器。Pentium 4 或 Core2 64 位可扩展地址内存，可寻址从 0000000000H ~ FFFFFFFFFH 的 ITB 存储器。

10）8086 ~ Core2 所有型号的微处理器都可以寻址 64KB 的 I/O 地址空间。这些 I/O 端口编号范围为 0000H ~ FFFFH，其中 I/O 端口 0000H ~ 03FFH 保留给 PC 系统使用。PCI 总线使用端口 0400H ~ FFFFH。

11）早期的 PC 中的操作系统是 MSDOS（Microsoft 磁盘操作系统）或 PCDOS（IBM PC 磁盘操作系统）之一。操作系统实现操作和控制计算机系统及 I/O 设备的任务。现代计算机用微软 Windows 作为操作系统，代替了 DOS。

12）微处理器是计算机系统中的控制部件。微处理器完成数据传送、简单的算术和逻辑运算，并进行简单的判定。微处理器执行存储在存储系统中的程序，以便在短时间周期内完成复杂的操作。

13）所有计算机系统都包含三种总线，用于控制存储器和 I/O。地址总线用于请求存储单元或 I/O 设备。数据总线在微处理器与它的存储器及 I/O 空间之间传送数据。控制总线控制存储器和 I/O，并请求读或写数据。控制是通过使用 $\overline{\text{IORC}}$（I/O 读控制）、$\overline{\text{IOWC}}$（I/O 写控制）、$\overline{\text{MRDC}}$（存储器读控制）和 $\overline{\text{MWTC}}$（存储器写控制）完成的。

14）记住数制每一位的权就可以将任一数制的数转换为十进制。任何数制系统中，小数点左边位的权总是个位，个位左边位的权总是基数乘以 1，后续位的权是由前一位的权乘以基数确定的。小数点右边位的权总是可以用除以基数的方法得到。

15）由十进制整数转换为其他进制时，可用除以基数的方法来实现。要转换十进制小数，则用基数乘以它来完成。

16）十六进制数据表示为十六进制格式，或表示为二进编码的十六进制（BCH）格式。对于一个二进制编码的十六进制数，可用 4 位二进制数表示每一位十六进制数字。

17）ASCII 码用于存储字母或数字数据。ASCII 码是 7 位代码，也可用它的第 8 位将字符集从 128 个代码扩展到 256 个代码。回车（Enter）码使打印头或光标返回到左边，换行代码使光标或打印头下移一行。现代应用程序使用 Unicode，它包含 0000H ~ 00FFH 之间的 ASCII 码。

18）有时计算机系统用二进制编码的十进制（BCD）存储十进制数据。这些数据以压缩格式（每字节 2 位数字）或者非压缩格式（每字节 1 位数字）存储。

19）在计算机系统中，二进制数据以字节（8 位）、字（16 位）或双字（32 位）存储。这些数可以是无符号的或有符号的。有符号负数总是以 2 的补码格式存储。比 8 位宽的数据总是以从小到大格式存储。在 32 位 Visual C++ 中，这些数据用“char”（8 位）、“short”（16 位）和“int”（32 位）来定义。

20）浮点数用于在计算机系统中存储整数、混合数及小数。浮点数由符号、尾数和阶组成。

21）汇编程序伪指令 DB 或 BYTE 定义字节，DW 或 WORD 定义字，DD 或 DWORD 定义双字，DQ 或 QWORD 定义四字。

1.6 习题

1. 谁开发了分析机？
2. 1890 年人口普查用了一种称为穿孔卡片机的新设备，这种设备是谁发明的？
3. 谁是 IBM 公司的创始人？
4. 谁发明了第一个电子计算器？
5. 第一台真正的电子计算机系统是为什么目的开发的？
6. 第一台通用可编程序计算机称为_____。
7. 世界上第一个微处理器是在 1971 年由_____开发的。
8. 谁是 Lovelace 伯爵夫人？
9. 谁开发了第一个高级程序设计语言 FLOWMATIC？
10. 什么是冯·诺依曼机器？
11. 哪一种 8 位微处理器导致进入了微处理器时代？
12. 1997 年推出的 8085 微处理器已销售了_____份？
13. 哪一种 Intel 微处理器第一个寻址 1MB 存储器？
14. 80286 可寻址_____字节存储器。
15. 80486 微处理器可使用多少存储器？
16. 什么时候 Intel 推出了 Pentium 微处理器？
17. 什么时候 Intel 推出了 Pentium Pro 微处理器？
18. 什么时候 Intel 推出了 Pentium 4 微处理器？
19. 哪些 Intel 微处理器寻址 1TB 存储器？
20. 缩写的 MIPS 是什么意思？
21. 缩写的 CISC 是什么意思？
22. 一个二进制位存储为一个_____或一个_____。
23. 计算机的 K 相当于_____字节。
24. 计算机的 M 相当于_____KB。
25. 计算机的 G 相当于_____MB。
26. 计算机的 P 相当于_____TB。
27. 在 4GB 存储系统能存储多少打字页的信息？
28. 在基于 DOS 的计算机系统中的第一个 1MB 存储器中包含_____区和_____区。

29. Windows 应用程序编程区有多大?
30. 在 DOS 临时程序区（TPA）中可找到多大容量的存储器?
31. 在 Windows 系统区中可找到多大容量的存储器?
32. 8086 微处理器可寻址_____字节存储器。
33. Core2 微处理器可寻址_____字节存储器。
34. 哪些微处理器可寻址 4GB 存储器?
35. 第一个 1MB 以上的存储器称为_____存储器。
36. 什么是系统 BIOS?
37. 什么是 DOS?
38. XT 和 AT 计算机系统之间的差别是什么?
39. 什么是 VESA 局部总线?
40. ISA 总线包括_____位接口卡。
41. 什么是 USB?
42. 什么是 AGP?
43. 什么是 XMS?
44. 什么是 SATA 接口?
45. 驱动程序存储在_____区。
46. PC 系统可寻址_____字节 I/O 空间。
47. BIOS 是做什么用的?
48. 画一个计算机系统的方框图。
49. 在基于微处理器的计算机系统中微处理器的作用是什么?
50. 列出计算机系统中的三种总线。
51. 哪种总线将存储器地址传送到 I/O 设备或存储器?
52. 什么控制信号导致存储器执行读操作?
53. $\overline{\text{IORC}}$信号的作用是什么?
54. 如果$\overline{\text{MRDC}}$信号是逻辑 0，那么微处理器执行什么操作?
55. 说明下列一些汇编程序伪指令的作用:
    (a) DB (b) DQ (c) DW (d) DD
56. 说明下列 32 位 Visual C++ 伪指令的作用:
    (a) char (b) short (c) int (d) float
    (e) double
57. 将下列二进制数转换为十进制:
    (a) 1101.01 (b) 111001.0011
    (c) 101011.0101 (d) 111.0001
58. 将下列八进制数转换为十进制:
    (a) 234.5 (b) 12.3
    (c) 7767.07 (d) 123.45
    (e) 72.72
59. 将下列十六进制数转换为十进制:
    (a) A3.3 (b) 129.C (c) AC.DC
    (d) FAB.3 (e) BB8.0D
60. 将下列十进制整数转换为二进制、八进制和十六进制:
    (a) 23 (b) 107 (c) 1238
    (d) 92 (e) 173
61. 将下列十进制数转换为二进制、八进制和十六进制:
    (a) 0.625 (b) 0.00390625 (c) 0.62890625
    (d) 0.75 (e) 0.9375
62. 将下列十六进制数转换为二进制编码的十六进制数(BCH):
    (a) 23 (b) AD4 (c) 34.AD
    (d) BD32 (e) 234.3
63. 将下列二进制编码的十六进制数转换为十六进制:
    (a) 1100 0010 (b) 0001 0000 1111 1101
    (c) 1011 1100 (d) 0001 0000
    (e) 1000 1011 1010
64. 将下列二进制数转换为 1 的补码:
    (a) 1000 1000 (b) 0101 1010
    (c) 0111 0111 (d) 1000 0000
65. 将下列二进制数转换为 2 的补码:
    (a) 1000 0001 (b) 1010 1100
    (c) 1010 1111 (d) 1000 0000
66. 定义字节、字和双字。
67. 将下列单词转换为 ASCII 码字符串:
    (a) FROG (b) Arc
    (c) Water (d) Well
68. 回车键的 ASCII 码是什么? 它的作用是什么?
69. 什么是单一码（Unicode)?
70. 用汇编程序伪指令在存储器中存储 ASCII 码字符串 'What time is it?'。
71. 将下列十进制数转换为 8 位有符号二进制数:
    (a) +32 (b) -12
    (c) +100 (d) -92
72. 将下列十进制数转换为有符号二进制字:
    (a) +1000 (b) -120
    (c) +800 (d) -3212
73. 用汇编伪指令在存储器中存储字节数 -34。
74. 在 Visual C++ 中建立一个字节变量 Fred1，并在里面存入 -34。
75. 说明下列 16 位十六进制数如何存储在存储系统中(使用标准 Intel 从小到大格式):
    (a) 1234H (b) A122H (c) B100H
76. 为了存储宽度大于 8 位的数据，从大到小与从小到大的格式有什么区别?
77. 用汇编程序伪指令将十六进制数 123A 存储到存储器中。
78. 将下列十进制数转换为压缩和非压缩格式的 BCD 码:
    (a) 102 (b) 44 (c) 301 (d) 1000
79. 将下列二进制数转换为有符号十进制数:
    (a) 10000000 (b) 00110011
    (c) 10010010 (d) 10001001
80. 将下列 BCD 数（假定它们是压缩格式）转换为十进制数:
    (a) 10001001 (b) 00001001
    (c) 00110010 (d) 00000001
81. 将下列十进制数转换为单精度浮点数:

(a) +1.5 (b) -10.625

(c) +100.25 (d) -1200

82. 将下列单精度浮点数转换为十进制数：

(a) 0 10000000 11000000000000000000000

(b) 1 01111111 00000000000000000000000

(c) 0 10000010 10010000000000000000000

83. 利用Internet写一篇短报告，介绍下列计算机先驱中的一位：

(a) Charles Babbage (b) Konrad Zuse

(c) Joseph Jacquard (d) Herman Hollerith

84. 利用Internet写一篇关于下列计算机语言之一的短报告：

(a) COBOL

(b) ALGOL

(c) FORTRAN

(d) PASCAL

85. 利用Internet写一篇短报告，详述Itanium 2微处理器的特性。

86. 利用Internet详述Intel的45nm（纳米）制造技术。

第2章　微处理器及其体系结构

引言

本章将微处理器作为可编程器件讨论，首先介绍其内部程序设计模型，然后介绍如何寻址存储器空间。在介绍 Intel 系列微处理器各成员存储器寻址方式的同时，介绍该系列微处理器的体系结构。

讨论这些高效系列微处理器的寻址方式时，将描述实模式操作、保护模式操作和平展模式操作。实模式存储器（DOS 存储器）位于00000H～FFFFFH，即存储系统的起始1MB 空间（适用于全部微处理器型号）。保护模式存储器（Windows 存储器）可位于整个保护存储系统的任何位置，但它只能用于80286～Core2 微处理器，不能用于早期的 8086 和 8088 微处理器。80286 包含 16MB 的保护模式存储器，80386～Pentium 包含 4GB，Pentium Pro～Core2 微处理器包含 4GB 或者 64GB。通过使其能 64 位扩展，Pentium 4 和 Core2 可以在平展内存模型（flat memory model）下寻址 1TB 的内存。为了能在 64 位模式下操作 Pentium 4 或 Core2 和使用平展模式内存寻址全部 1TB 的内存空间，需要使用 Windows Vista 或 Windows 64。

目的

读者学习完本章后将能够：

1）描述 8086～Core2（包括 64 位扩展）微处理器中每个程序可见寄存器的功能和用途。
2）详细描述标志寄存器和每个标志位的用途。
3）描述如何用实模式存储器寻址技术访问存储器。
4）描述如何用保护模式存储器寻址技术访问存储器。
5）描述如何用 64 位平展存储器模式访问存储器。
6）描述 80286～Core2 微处理器中的程序不可见寄存器。
7）详细描述内存分页机制的运作方式。

2.1　微处理器的内部体系结构

在编写程序和研究任何指令前，首先必须了解微处理器的内部结构。本章在这一节详述 8086～Core2 微处理器的程序可见的内部结构，同时详细说明每个内部寄存器的功能和用途。注意，在多核微处理中每个核心都包含相同的可编程模式。惟一的不同是各个核心都运行同步的独立线程或任务。

2.1.1　程序设计模型

8086～Core2 的程序设计模型是**程序可见（program visible）**的，因为在程序设计期间要使用由指令指定的寄存器。本章后面叙述的其他寄存器是**程序不可见（program invisible）**的，因为在应用程序设计期间不能直接寻址它们，但在系统程序设计期间可以被间接引用。只有 80286 及更高档型号的微处理器包含程序不可见寄存器，它们用于控制和操作保护模式存储系统和其他特征。

图 2-1 说明了 8086～Core2 微处理器的程序设计模型。早期的 8086、8088 和 80286 包含 **16 位**内部结构，是图 2-1 所示寄存器组的子集。80386～Core2 微处理器包括全部的 32 位内部结构。早期的 8086～80286 的结构与 80386～Core2 的结构完全向上兼容。这个图中阴影区域的寄存器在 8086、8088 和 80286 微处理器中不存在，它们是 80386～Core2 微处理器中新增的。

程序设计模型包括 8 位、16 位和 32 位寄存器。Pentium 4 和 Core2 在程序设计模型中用 64 位模式操作时也包括 64 位寄存器。8 位寄存器有 AH、AL、BH、BL、CH、CL、DH 和 DL，在指令中用这些双字母的名字引用它们。例如 ADD AL，AH 指令，将 8 位寄存器 AH 的内容加到 AL 中（这条指令只

改变 AL 的内容)。16 位寄存器有 AX、BX、CX、DX、SP、BP、DI、SI、IP、FLAGS、CS、DS、ES、SS、FS 和 GS。注意，开始的 4 个 16 位寄存器包含两个 8 位寄存器，例如 AX 包括 AH 和 AL。这些寄存器也用双字母名字引用如 AX。例如 ADD DX, CX 指令，将 16 位寄存器 CX 的内容加到 DX 中（这条指令只改变 DX 的内容）。32 位扩展寄存器是 EAX、EBX、ECX、EDX、ESP、EBP、EDI、ESI、EIP 和 EFLAGS。这些 32 位扩展寄存器和 16 位寄存器 FS、GS 只用于 80386 及更高型号的微处理器中。这些寄存器中，前个面的 16 位寄存器采双字母名字 FS 或 GS 引用，而 32 位寄存器则用三字母名字引用，例如 ADD ECX, EBX 指令，将 EBX 中的 32 位内容加到 ECX 中（这条指令只改变 ECX 的内容）。

一些寄存器是通用寄存器或多功能寄存器，而另一些是专用寄存器。多功能寄存器包括 EAX、EBX、ECX、EDX、EBP、EDI 和 ESI。这些寄存器存储各种长度的数据（字节、字或双字），并用于程序指定的各种用途。

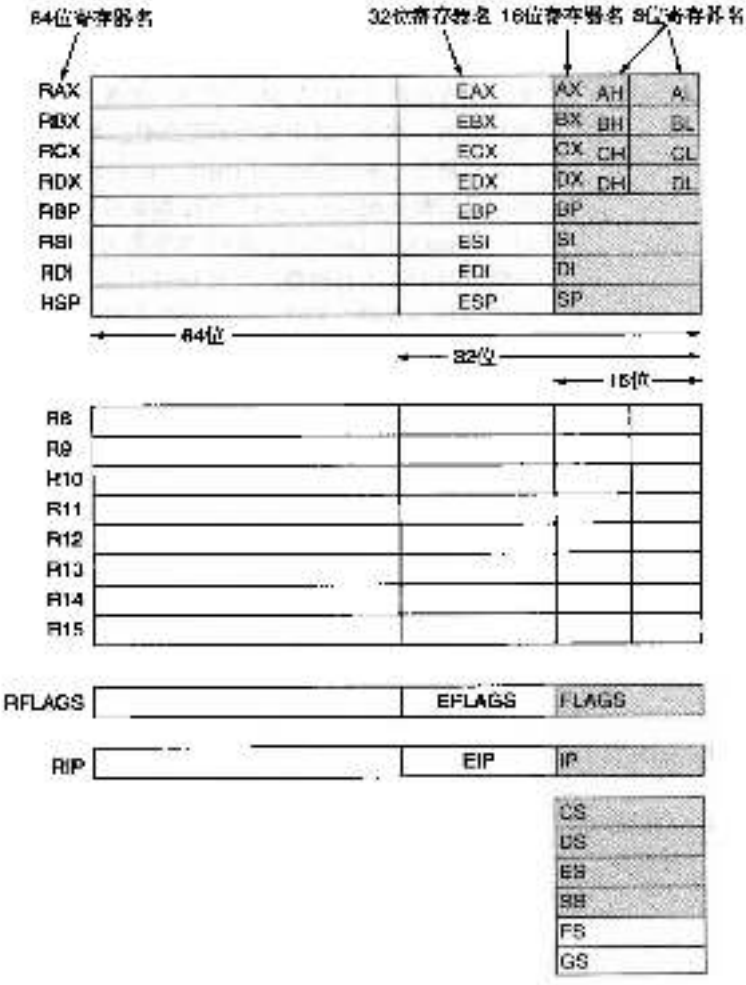


图 2-1 Intel 8086 ~ Core2（包括 64 位扩展）的程序设计模型

64 位寄存器被指定为 RAX、RBX 等。除了把寄存器重命名为 64 位宽之外，还有额外的名为 R8 ~ R15 的 64 位寄存器。这种 64 位的扩展使得 Pentium 4 和 Core2 与图 2-1 中的影部分所示的原始微处理器结构相比，其可用寄存器空间增加了 8 倍多。一个 64 位指令的例子是 ADD RCX, RBX 指令，它把 RBX 的 64 位内容加到 RCX。（这条指令只会改变 RCX。）一个不同之处在于：这些额外的 64 位寄存器（R8 ~ R15）是按照字节、字、双字或者四字的方式寻址的，即是只有最右边的 8 位是一个字节。R8 ~ R15 不支持把其第 8 位~第 15 位作为一个字节来直接寻址。在 64 位模式中，一个合法的高字节寄存器（AH、BH、CH 或者 DH）不能够与一个由 R8 ~ R15 的寄存器所表示的字节在同一个指令中寻址。由于合法软件不会访问 R8 ~ R15，因此不会为现有的 32 位程序带来任何问题，现有程序无需修改即可工作。

表 2-1 列出了用于访问 64 位寄存器的部分内容的控制字。为了访问 R8 寄存器的低位字节，可以使用 R8B（其中 B 表示低位字节）。同样地，为了访问一个编号寄存器的低位字，例如 R10，可以在指令中使用 R10W。字母 D 用于访问双字。例如，把 R8 的低位双字内容拷贝到 R11 的低位双字的指令为 MOV R11D, R8D。对完整的 64 位寄存器的访问不需要任何特殊的控制字。

表 2-1 对编号寄存器的平展模式 64 位访问

寄存器大小	控 制 字	访 问 位 数	示 例
8 字节	B	7～0	MOV R9B, R10B
16 字节	W	15～0	MOV R10W, AX
32 字节	D	31～0	MOV R14D, R15D
64 字节	–	63～0	MOV R13, R12

2.1.2 多功能寄存器

RAX（累加器） RAX 可作为 64 位寄存器（RAX）、32 位寄存器（EAX）、16 位寄存器（AX）或两个 8 位寄存器（AH 或 AL）引用。注意，如果是 8 位或 16 位寻址，则只改变 32 位寄存器的一部分，其余部分不受影响。累加器用于乘法、除法及一些调整指令。对于这些指令，累加器有专门用途，但它通常被认为是多功能寄存器。在 80386 及更高型号的微处理器中，EAX 寄存器也可以保存访问存储单元的偏移地址。在 64 位的 Pentium 4 和 Core2 中，RAX 寄存器可保持 64 位的偏移地址，可以通过 40 位地址总线访问 1TB 的存储器。未来 Intel 计划扩展到 52 位地址总线访问 4PB 的存储器。

RBX（基址） RBX 可作为 RBX、EBX、BX、BH 或 BL 寻址。在所有型号的微处理器中，RBX 有时用于保存访问存储单元的偏移地址。在 80386 及更高型号的微处理器中，EBX 也能寻址存储器数据。在 64 位的 Pentium 4 和 Core2 中，RBX 也能寻址存储器数据。

RCX（计数） RCX 可作为 RCX、ECX、CX、CH 或 CL 寻址，它是个通用寄存器，也可保存许多指令的计数值。在 80386 及更高型号的微处理器中，ECX 寄存器也可保存访问存储器数据的偏移地址。64 位的 Pentium 4 中，RCX 也可以寻址存储器数据。用于计数的指令是重复的串指令（REP/REPE/REPNE）以及移位、循环和 LOOP/LOOPD 指令。移位和循环指令用 CL 计数，重复的串指令用 CX 计数，LOOP/LOOPD 指令用 CX 或 ECX 计数。如果以 64 位模式操作，LOOP 指令使用 64 位 RCX 寄存器进行循环计数。

RDX（数据） RDX 可作为 RDX、EDX、DX、DH 或 DL 寻址，它是通用寄存器，用于保存乘法形成的部分结果，或者除法之前的部分被除数。对于 80386 及更高型号的微处理器，这个寄存器也可寻址存储器数据。

RBP（基指针） RBP 可作为 RBP、EBP 或 BP 寻址，在所有型号的微处理器中，为了传送存储器数据，RBP 指向存储单元。

RDI（目的变址） RDI 可作为 RDI、EDI 或 DI 寻址，它常用于寻址串指令的目的数据串。

RSI（源变址） RSI 可作为 RSI、ESI 或 SI 使用。源变址寄存器通常为串指令寻址源数据串。如同 RDI 一样，RSI 也作为通用寄存器使用。如果它作为 16 位寄存器，就由 SI 寻址；如果作为 32 位寄存器，就由 ESI 寻址；如果作为 64 位寄存器，就由 RSI 寻址。

R8～R15 这些寄存器只存在于 Pentium 4 和 Core2 中 64 位扩展允许的情况下。如前所述，这些寄存器中的数据是用于通用目的的，按照 64、32、16 或 8 位大小寻址。直到 64 位处理器广泛使用，大部分应用程序才会使用这些寄存器。请注意 8 位部分只是寄存器中最右边的 8 位，第 8 位～第 15 位不按照一个字节直接寻址。

专用寄存器

专用寄存器包括：RIP、RSP 和 RFLAGS 以及段寄存器 CS、DS、ES、SS、FS 和 GS。

RIP（指令指针） RIP 寻址代码段存储区内的下一条指令。当微处理器工作在实模式下时，这个寄存器是 IP（16 位）；当 80386 及更高型号的微处理器工作于保护模式下时，则是 EIP（32 位）。注意，8086、8088 和 80286 不包含 EIP 寄存器，而且只有 80286 及更高型号的微处理器可以工作于保护模式。指令指针指向程序的下一条指令，用于微处理器在程序中顺序地寻址代码段内的下一条指令。指令指针也可由转移指令或调用指

令修改。在64位模式中，RIP包含40位地址总线，可用于寻址1TB平展模式地址空间。

RSP（堆栈指针） RSP寻址一个称为堆栈的存储区。通过这个指针存取堆栈存储器数据，具体操作将在本书后面讲解访问堆栈存储器数据的指令时再进行说明。这个寄存器作为16位寄存器被引用时，为SP；如果作为32位寄存器，则是ESP。

RFLAGS RFLAGS用于指示微处理器的状态并控制它的操作。图2-2展示了所有型号微处理器的标志寄存器。注意，从8086/8088直到Core2微处理器是向上兼容的。8086～80286包含FLAG寄存器（16位），80386及更高型号的微处理器包含EFLAG寄存器（32位扩展的标志寄存器）。64位RFLAGS包含EFLAG寄存器，这在64位版本中也是不变的。

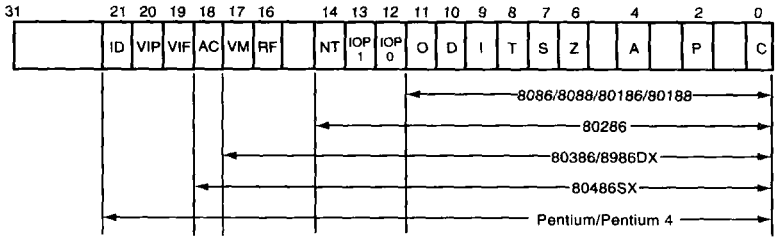


图2-2 80X86～Pentium全系列微处理器的EFLAG和FLAG寄存器

最右边的5位标志和溢出标志在执行算术和逻辑指令后会改变，而对于任何数据传送或程序控制操作这些标志都不改变。某些标志也用于控制微处理器中的一些特性功能。下面列出每个标志位及其功能的简要说明。后续章节里介绍指令时，再给出关于标志位的更详细的叙述。

C（进位） 进位标志保存加法以后的进位或减法以后的借位。也可用进位标志指示由某些程序或过程引发的错误条件，这对DOS功能调用尤其有用。

P（奇偶校验） 奇偶校验标志表示结果数中1的个数是奇数还是偶数，是奇数则标志是逻辑0，是偶数则该标志是逻辑1。如果某个二进制数含有3个为1的位，它的奇偶性为奇数。如果某个二进制数包含0个为1的位，它的奇偶性为偶数。奇偶校验标志在现代程序设计中很少使用，它是早期Intel微处理器在数据通信环境中校验数据的一种手段。今天，奇偶校验常常由数据通信设备完成，而不是由微处理器完成。

A（辅助进位） 辅助进位标志保存加法后结果中的第3位与第4位之间的进位（半进位），或者减法后结果中的第3与第4位之间的借位。DAA和DAS指令测试这个特殊标志位，以便在BCD加法或减法后对AL中的值进行十进制调整。除此以外，微处理器或者任何其他指令都不使用A标志位。

Z（零） 零标志表示一个算术或逻辑操作的结果是否为零。如果Z=1，表示结果为0；如果Z=0，说明结果不为0。这可能令人迷惑，但Intel就是这样命名这个标志的。

S（符号） 符号标志保持执行算术或逻辑运算指令后所得结果的算术符号。如果S=1，则符号位（数的最左一位）为1或为负；如果S=0，则符号位为0或为正。

T（陷阱） 陷阱标志使能微处理器芯片上的调试功能（对程序进行调试，以便找到错误或故障）。如果T标志为使能（为1），则微处理器根据调试寄存器和控制寄存器的指示中断程序流；如果T标志为逻辑0，则禁止陷阱（调试）性能。Visual C++调试工具可以利用陷阱特性和调试寄存器调试有缺陷的软件。

I（中断） 中断标志控制INTR（中断请求）输入引脚的操作。如果I=1，则使能INTR引脚；如果I=0，则禁止INTR引脚。I标志的状态由STI（**置位I标志**）和CLI（**清除I标志**）指令控制。

D（方向） 在串指令操作期间，方向标志为 DI 和/或 SI 寄存器选择递增方式或递减方式。如果 D = 1，则寄存器内容自动地递减；如果 D = 0，则寄存器内容自动地递增。D 标志用 STD（**置位方向**）指令置位，用 CLD（**清除方向**）指令清除。

O（溢出） 溢出标志在有符号数进行加或减时可能出现。溢出指示运算结果已超出机器能够表示的范围。例如，用 8 位加法将 7FH（+127）加上 01H（-1），结果为 80H（-128）。这个有符号数加法结果的溢出状况由溢出位指示。对于无符号数的操作，不考虑溢出标志。

IOPL（I/O 优先级） 输入/输出优先级标志用于在保护模式下操作时为 I/O 设备选择优先级。如果当前任务的优先级高于或等于 IOPL，则 I/O 指令能顺利执行。如果当前优先级比 IOPL 低，则产生中断，导致执行程序被挂起。注意，00 级是最高（最大）优先级，11 是最低（最小）优先级。

NT（任务嵌套） 任务嵌套标志指示在保护模式下当前执行的任务嵌套于另一任务中。当任务被软件嵌套时，这个标志置位。

RF（恢复） 恢复标志在调试时使用，控制在下条指令后恢复程序的执行。

VM（虚拟模式） 虚拟模式标志位用于在保护模式系统中选择虚拟操作模式。虚拟模式系统允许多个 1MB 长的 DOS 存储器分区共存于存储系统中。这样可以允许系统执行多个 DOS 程序。VM 用于在现代 Windows 环境下仿真 DOS。

AC（对齐检查） 当寻址一个字或双字时，如果地址不是在字或双字的边界上，对齐检查标志位就被激活为 1。只有 80486SX 微处理器包含对齐检查位，这个位用来与其配套的协处理器 80487SX 同步。

VIF（虚拟中断） 虚拟中断标志是中断标志位的副本，只有 Pentium ～ Pentium 4 微处理器才有。

VIP（虚拟中断挂起） 虚拟中断挂起标志为 Pentium ～ Pentium 4 微处理器提供有关虚拟模式中断的信息。它用于多任务环境下，为操作系统提供虚拟中断标志和中断挂起信息。

ID（标识） 标识标志指示 Pentium ～ Pentium 4 微处理器支持 CPUID 指令。CPUID 指令给系统提供有关 Pentium 微处理器的信息，如版本号和制造商。

段寄存器

另外的一些寄存器叫做段寄存器，用来和微处理器中的其他寄存器结合生成存储器地址。不同型号的微处理器中有 4 个或者 6 个段寄存器。段寄存器的功能在实模式下和保护模式下是不同的。本章后面对段寄存器在实模式下和在保护模式下的功能进行详细说明。在 64 位平展模式操作中，段寄存器除了代码段寄存器外很少在程序中使用。下面先列出各个段寄存器及其在系统中的功能：

CS（代码段） 代码段是一个存储器区域，在这里保存微处理器使用的代码（程序和过程）。代码段寄存器定义了存放代码的存储器段的起始地址。在实模式下工作时，它定义一个 64KB 存储器段的起始地址；在保护模式下工作时，它选择一个描述代码存储器起始地址和长度的描述符。对于 8086 ～ 80286，代码段限制为 64KB；80386 及更高型号的微处理器工作在保护模式下时，代码段限制为 4GB。在 64 位模式中，代码段寄存器仍然应用于平展模式，但是它的用法与 2.5 小节介绍的其他可编程模式不同。

DS（数据段） 数据段也是一段存储区域，含有程序使用的大部分数据。可以通过偏移地址或者其他含有偏移地址的寄存器的内容访问数据段里的数据。和代码段及其他段一样；对于 8086 ～ 80286，数据段的长度限制为 64KB；对于 80386 及更高型号的微处理器，数据段的长度限制为 4GB。

ES（附加段） 附加段是一个附加的数据段，为某些串指令存放目的数据。

SS（堆栈段） 堆栈段为堆栈定义一个存储区域。由堆栈段和堆栈指针寄存器确定堆栈段内当前的入口地址。BP 寄存器也可以寻址堆栈段内的数据。

FS 和 GS FS 和 GS 段是在 80386 ～ Core2 微处理器中增加的段寄存器，以便允许程序访问这两个附加的存储器段。Windows 将这些段寄存器用于内部操作，但没有说明其使用方法。

2.2 实模式存储器寻址

80286 及更高型号的微处理器可以工作于实模式或者保护模式，而 8086 和 8088 只能工作于实模式。在 Pentium 4 和 Core2 的 64 位操作模式中，不存在实模式操作。本节详细叙述实模式下微处理器的操作方式。**实模式操作方式**（**real mode operation**）只允许微处理器寻址起始的 1MB 存储器空间，即使 Pentium 4 和 Core2 微处理器也是如此。注意，起始的 1MB 存储器称为**实模式存储器**（**real memory**）、**常规内存**（**conventional memory**）或 **DOS 存储器**系统。DOS 操作系统要求微处理器工作于实模式，Windows 不能用实模式。实模式操作时允许为 8086/8088（只包含 1MB 存储器）设计的应用软件不作修改就可以在 80286 及更高型号的微处理器中运行。软件的向上兼容性是 Intel 系列微处理器不断成功的重要原因之一。在任何情况下，这些微处理器每次加电或复位后都默认以实模式开始工作。注意，如果 Pentium 4 或 Core2 处于 64 位模式，那么将不存在实模式操作方式；从而 DOS 应用程序不存在于 64 位模式，除非程序为 64 位模式虚拟编写 DOS。

2.2.1 段和偏移

实模式下，用段地址和偏移地址的组合访问存储单元，所有实模式存储单元的地址都由段地址加偏移地址组成。装在段寄存器内的**段地址**（**segment address**）确定任何 64KB 存储器段的起始地址。**偏移地址**（**offset address**）用于在 64KB 存储器段内选择任一单元。实模式段的长度总是 64KB。图 2-3 说明了**段加偏移**（**segment plus offset**）的寻址机制如何选择存储单元。图中显示了一个 64KB 长的存储器段，这个段起始于 10000H，结束于 1FFFFH。它也显示了偏移地址 F000H 怎样选择存储系统中的 1F000H 单元，偏移地址有时也称为**位移**（**displacement**）。注意，如图 2-3 所示，偏移或者位移是从段的起始位置向上到所选单元的距离。

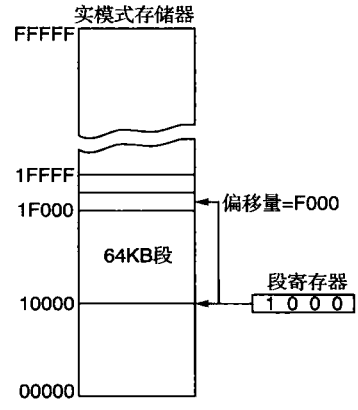


图 2-3 实模式存储器寻址机制，使用段地址加偏移地址

图 2-3 中段寄存器内容为 1000H，然而它寻址的段起始于 10000H。在实模式中，每个段寄存器内容的最右边增加一个 **0H**，如此形成 20 位存储器地址，它可以访问一个存储器段的起始点。微处理器必须生成 20 位的存储器地址，以便访问起始 1MB 存储器内的一个单元。例如，如果段寄存器内容为 1200H，则它寻址起始于 12000H 单元的 64KB 存储器段。类似地，如果段寄存器内容是 1201H，则它寻址起始于 12010H 单元的存储器段。因为内部添加了 0H，实模式下，段只能起始于存储系统内 16 字节整数倍的边界。这个 16 字节边界通常称为**小段**（**paragraph**）。

由于实模式存储器段长为 64KB，一旦知道段的起始地址，再加上 FFFFH 就可得到段的**结束地址**（**ending address**）。例如，如果段寄存器内容为 3000H，则段的起始地址是 30000H，其结束地址是 30000H + FFFFH，即 3FFFFH。表 2-2 给出了几个段寄存器的内容及由每个段地址选择的存储器段的起始地址和结束地址。

作为地址一部分的偏移地址与段的起始地址相加，用于寻址存储系统内的存储单元。例如，如果段地址为 1000H，偏移地址为 2000H，则微处理器寻址存储单元 12000H。为了定位数据，偏移地址总是加到段起始地址上。段和偏移地址有时也写成 1000：2000 形式，表示段地址为 1000H，偏移地址为 2000H。

表 2-2 实模式段地址的例子

段寄存器	起始地址	结束地址
2000H	20000H	2FFFFH
2001H	20010H	3000FH
2100H	21000H	30FFFH
AB00H	AB000H	BAFFFH
1234H	12340H	2233FH

在 80286（有专门外部电路）及 80386 ～ Pentium 4 中，

当段地址是 FFFFH，而且系统中安装了用于 DOS 的驱动程序 HIMEM. SYS 时，可以寻址 64KB 减 16 字节的附加存储器区域。这个可寻址的存储器区域（0FFFF0H ～ 10FFEFH）作为**高端存储器**（**high memory**）访问。当用段地址 FFFFH 生成地址时，地址 A20 引脚被置位（如果支持）。例如，如段地址为 FFFFH，偏移地址为 4000H，则机器寻址位于 FFFF0H + 4000H 的存储单元，即 103FF0H。注意，此时 A20 保持逻辑 1。如果不支持 A20，则生成地址是 03FF0H，因为 A20 总保持逻辑 0。

有些寻址方式将多个寄存器内容与一个偏移量结合，形成偏移地址。这种情况下，这些值之和可能超过 FFFFH。例如，如果段地址为 4000H，指定偏移地址为 F000H 与 3000H 相加之和，则将寻址 42000H 单元，而不是 52000H 单元。因为，当 F000H 与 3000H 相加时，形成的偏移地址为 16 位（模 2^{16}）的和 2000H，而不是真正的和 12000H。注意，这个加法的进位 1（F000H + 3000H = 12000H）被丢掉了，因此生成的偏移地址为 2000H。这样，生成的地址就是 4000：2000 或 42000H。

2.2.2 默认段和偏移寄存器

微处理器有一套规则，用于每次访问存储器段。这套规则既适于实模式也适于保护模式，规则定义了各种寻址方式中段地址寄存器和偏移地址寄存器的组合方式。例如，代码段寄存器总是和指令指针组合用于寻址程序的下一条指令。根据微处理器的操作模式，这种组合是 **CS：IP** 或者 **CS：EIP**。**代码段**（**code segment**）寄存器定义代码段的起点，**指令指针**（**instruction pointer**）指示代码段内的下一条指令的位置。这样的组合（**CS：IP** 或 **CS：EIP**）定位微处理器执行的下一条指令。例如，如果 CS = 1400H 且 IP/EIP = 1200H，则微处理器从存储器的 14000H + 1200H 单元（即 15200H 单元）取下一条指令。

另外一种默认组合用于**堆栈**（**stack**）。通过栈指针（SP/ESP）或者基指针（BP/EBP）寻址堆栈段中某存储单元的堆栈数据。这些组合用 SS：SP（SS：ESP）或者 SS：BP（SS：EBP）表示。例如，如果 SS = 2000H 且 BP = 3000H，则微处理器寻址堆栈段的 23000H 存储单元。注意，在实模式下，只用扩展寄存器的最右边 16 位寻址存储器段内的某个单元。在 80386 ～ Pentium 4 中，如果处理器是工作于实模式，那么决不能将大于 FFFFH 的数据放入到偏移寄存器中，否则将引起系统停机并指示寻址错误。

用 Intel 微处理器 16 位寄存器寻址存储器的其余默认组合如表 2-3 所示。表 2-4 表示 80386 和更高型号的处理器使用 32 位寄存器组合寻址的默认情况。注意，80386 及更高型号的微处理器与 8086 ～ 80286 微处理器相比，段-偏移地址寻址组合的选择范围更大。

表 2-3 默认的 16 位段 + 偏移寻址组合

段	偏移量	特殊用途
CS	IP	指令地址
SS	SP 或 BP	堆栈地址
DS	BX、DI、SI、8 位数或 16 位数	数据地址
ES	串指令的 DI	串目标地址

表 2-4 默认的 32 位段 + 偏移寻址组合

段	偏移量	特殊用途
CS	EIP	指令地址
SS	ESP 或 EBP	堆栈地址
DS	EAX、EBX、ECX、EDX、ESI、EDI，一个 8 位或 32 位数	数据地址
ES	串指令的 EDI	串目标地址
FS	无默认值	一般地址
GS	无默认值	一般地址

8086 ～ 80286 微处理器允许访问 4 个存储器段，80386 ～ Pentium 4 微处理器可以访问 6 个存储器段。图 2-4 表示一个包含 4 个存储器段的系统。注意存储器段可以邻接甚至重叠。如果一段不需要 64KB 存储器，则它可与其他段重叠。我们可以把段想象成一个窗口，它可以移动，覆盖任何存储区，以便访问数据或代码。一个程序可以有多于 4 个或 6 个存储器段，但每次只能访问 4 个或 6 个段。

假定某个应用程序的代码需要 1000H 个字节的存储器空间，数据需要 190H 个字节的存储器空间，堆栈需要 200H 个字节的存储器空间，这个应用程序不需要附加段。当 DOS 将这个程序装入存储器时，它被存入第一个有效存储区的 TPA 区，位于设备驱动程序和其他 TPA 程序之上。该区由一个 DOS 管理的空闲指针（free-pointer）指示。程序的装入由 DOS **程序装入程序**（**program loader**）自动管理。图 2-5 表示了这个应用程序在存储器内是怎样存储的。各个段相互重叠，因为其中的数据不需要 64KB 存储器空间。段的侧视图清楚地显示了各段是重叠的，以及通过改变段的起始地址，可以把段移到任何存储区。幸运的是，段的起始地址是由 DOS

程序装入程序计算并分配的。

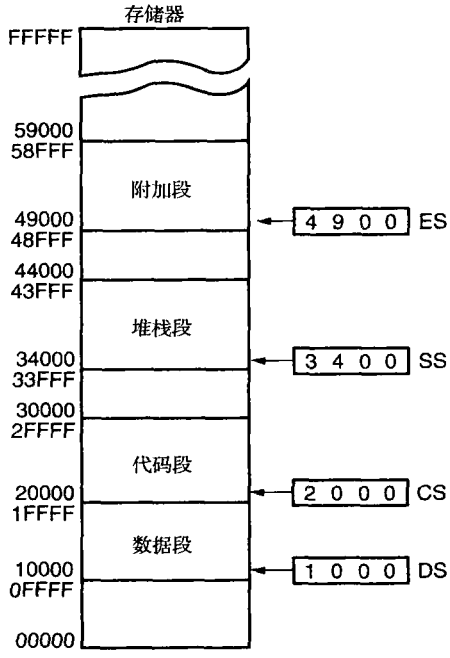


图 2-4 安排有 4 个存储段的存储系统

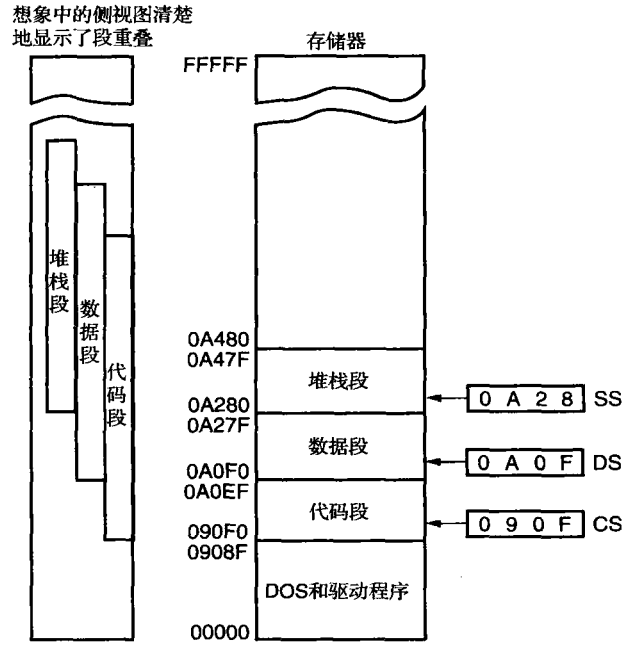


图 2-5 含有代码段、数据段和堆栈段的应用程序装入 DOS 系统存储器中

2.2.3 段和偏移寻址机制允许重定位

段和偏移寻址机制似乎非常复杂。它的确复杂，但它给系统带来许多优点。这种复杂的段加偏移的寻址机制允许 DOS 程序在存储器内重定位，允许为在实模式下运行而编写的程序在保护模式下也可运行。**可重定位程序**（**relocatable program**）是一个可以放入存储器的任何区域，且不需修改而仍能执行的程序。**可重定位数据**（**relocatable data**）是可以放在存储器的任何区域，且不需要修改就可以被程序引用的数据。段和偏移寻址机制允许程序和数据不需要任何修改即可进行重定位。这对于包含不同存储器区域的通用计算器系统是非常理想的，因为各种 PC 的存储器结构并不相同，要求软件和数据能够重新定位。

因为存储器是用偏移地址在段内寻址的，所以可将整个存储器段移动到存储系统内的任何地方而无需改变任何偏移地址。通过把整个程序像块一样移到新的区域，然后只是改变段寄存器的内容，就实现了重定位。如果一条指令位于距段首 4 个字节的位置，那么它的偏移地址是 4。如果整个程序移到新的存储区，这个偏移地址 4 仍然指向距段首 4 个字节的位置。只是段寄存器内容必须变为程序所在的新存储区的地址。没有这种特性，一个程序在移动之前就必须大范围地重写或更改，因而需要大量的时间，或为许多不同配置的计算机系统开发不同的程序版本。这个概念也用于编写保护模式下执行的 Windows 程序。在 Windows 环境下编写任何程序时都假定：代码和数据都可以得到起始的 2GB 存储器，当加载程序到实际存储器时，可以把它放在任何地方，甚至可以把一部分程序以交换文件的形式放在磁盘上。

2.3 保护模式存储器寻址简介

保护模式存储器寻址（80286 及更高型号的微处理器）允许访问位于起始 1MB 及起始 1MB 以上的存储器内的数据和程序。Windows 运行在**保护模式**（**protected mode**）下。寻址这个扩展的存储区，需要更改用于实模式存储器寻址的段加偏移寻址机制。在保护模式下，当寻址扩展内存里的数据和程序时，仍然使用偏移地址访问位于存储器段内的信息。两者的区别是，保护模式下不再像实模式那样提

供段地址。在原来放段地址的段寄存器里含有一个**选择子**（**selector**），用于选择描述表内的一个描述符。**描述符**（**descriptor**）描述存储器段的位置、长度和访问权限。由于段寄存器和偏移地址仍然用于访问存储器，所以保护模式指令和实模式指令是完全相同的。事实上，很多为在实模式下运行编写的程序，不用更改就可在保护模式下运行。两种模式之间的区别是微处理器访问存储段时对段寄存器的解释不同。在80386及更高型号的微处理器中，两种模式的另一个差别是：在保护模式下偏移地址可以是32位数而非16位数。32位偏移地址允许微处理器访问长达4GB的段内数据。为32位保护模式编写的程序可以在Pentium 4的64位模式下运行。

2.3.1 选择子和描述符

装在段寄存器里的选择子从两个描述符表之一选择8192个描述符中的一个。描述符说明存储段的位置、长度和访问权限。段寄存器仍然选择一个存储器段，但不再像实模式那样直接选择而是间接选择。例如，在实模式中，如果CS=0008H，则代码段起始于00080H单元。而在保护模式中，这个段号可以寻址整个系统内作为代码段的任何存储器区。

段寄存器可以访问两个描述符表：一个包括全局描述符，另一个包括局部描述符。**全局描述符**（**global descriptor**）包含适用于所有程序的段定义，而**局部描述符**（**local descriptor**）通常用于惟一的应用程序。可以把全局描述符称为**系统描述符**（**system descriptor**），把局部描述符称为**应用描述符**（**application descriptor**）。每个描述符表包含8192个描述符，所以在任何时刻应用程序最多可有16 384个描述符。因为一个描述符说明一个存储段，这就允许每个应用程序可描述多达16 384个存储段。既然一个存储器段能达到4GB，就意味着一个应用程序能够访问4G×16 384B，即64TB存储器。

图2-6表示80286～Core2的描述符格式。注意每个描述符长8个字节，所以全局和局部描述符表每个最长为64KB。80286的描述符和80386～Core2的描述符稍有区别，但80286描述符是向上兼容的。

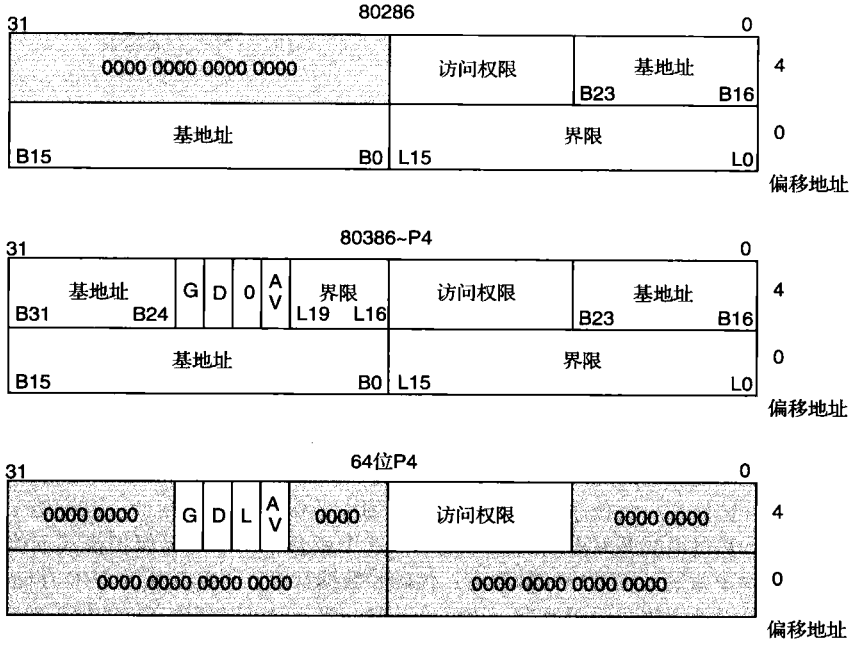


图2-6 80286～Core2微处理器的描述符

描述符的**基地址**（**base address**）部分指示存储器段的起始位置。对于80286微处理器，基地址长24位，所以段可起始于16MB存储器内的任何地方。注意，当这些微处理器工作在保护模式时，取消了小段边界限制，所以段可以在任何地址处开始。80386及更高型号的微处理器用32位基地址，允许段起始于4GB存储器的任何地方。注意80286描述符中的基地址是如何与80386～Pentium 4向上兼容

的，因为其基地址最高16个有效位是0000H。详细内容参见第18章和第19章有关Pentium Pro～Core2提供的64 GB存储器空间部分。

段界限（segment limit）包含该段中最大的偏移地址。例如，如果某个段起始于存储器F00000H地址，结束于F000FFH地址，则其基地址是F00000H，界限是FFH。对于80286微处理器，基地址是F00000H，界限是00FFH。对于80386及更高型号的微处理器，基地址是00F00000H，界限是000FFH。注意，80286的界限是16位，80386～Pentium 4的界限是20位。80286访问长度为1B至64KB之间的存储器段，80386及更高型号的微处理器访问长度为1B～1MB之间或者4KB～4GB之间的存储器段。

80386～Pentium 4描述符中还用了一个80286描述符中没有的特征位：G位，即**粒度位**（**granularity bit**）。如果G=0，说明段的界限为00000H～FFFFFH（0～1MB）。如果G=1，则界限值要乘以4KB（在尾部添上FFFH），界限为00000FFFH～FFFFFFFFH。如果G=1，则允许段的长度为4KB～4GB，以4KB为单位。因为80286的16位内部体系结构使其偏移地址总是16位的，所以80286的段长为64KB。80386及更高型号的微处理器采用32位结构，在保护模式下允许32位的偏移地址。32位偏移地址允许段长为4GB，16位偏移地址允许段长为64KB。操作系统运行在16位或32位环境下。例如，DOS运行在16位环境下，而多数Windows应用程序运行在32位环境下，称为**WIN32**。

在64位描述符中，L位可能是大的意思，但是Intel称之为64位，在L=1时选择Pentium 4或者Core2中的带有64位扩展的64位地址，在L=0时选择32位兼容模式。在64位保护模式下，代码段寄存器仍用于从内存中选择一个段代码。注意64位描述符没有界限或基地址。它只包括一个访问权限字节和若干控制位。在64位模式下，描述符中没有段或者界限。段的基地址为00 0000 0000H，尽管它没有在描述符中出现。这意味着为了实现64位操作，所有的代码段都从地址0开始。64位代码段没有边界检查。

例2-1展示了如果段的基地址为10000000H、界限为001FFH且G位为0时，段的起始地址和段的结束地址。

例2-1

```
Base = Start = 10000000H
G = 0
End = Base + Limit = 10000000H + 001FFH = 100001FFH
```

例2-2使用了与例2-1相同的数据，只是此例的G位为1。注意，界限后面附加了FFFH，以便确定段结束地址。

例2-2

```
Base = Start = 10000000H
G = 1
End = Base + Limit = 10000000H + 001FFFFFH = 101FFFFFH
```

80386及更高型号的微处理器描述符中的AV位指示段有效（AV=1）或者段无效（AV=0）。D位指示在保护模式或实模式下80386～Core2指令如何访问寄存器和存储器数据。如果D=0，则指令与8086～80286微处理器兼容，是16位指令。这意味着指令在默认情况下用16位偏移地址和16位寄存器。这种模式通常称为16位指令模式或DOS模式。如果D=1，则指令是32位指令，在默认情况下，32位指令模式假定所有偏移地址和所有寄存器都为32位。注意，在16位和32位两种指令模式中，默认的寄存器长度和偏移地址长度都可以被超越。MSDOS和PCDOS两个操作系统都要求工作于16位指令模式。Windows 3.1和任何为它所编写的应用程序也要求选择16位指令模式。注意，32位指令模式只能用于保护模式系统中，如Windows Vista。关于这些模式及其在指令系统中的应用的细节将在第3章和第4章详细叙述。

访问权限字节（**access rights byte**）（如图2-7所示）控制着对保护模式中存储器段的访问。这个字节描述了段在系统中怎样起作用。访问权限字节始终全面地控制着段。如果是数据段，则指定其增长方向。如果段的增长超出了它的界限，则中断微处理器的操作系统程序，并指示一般性保护错误。

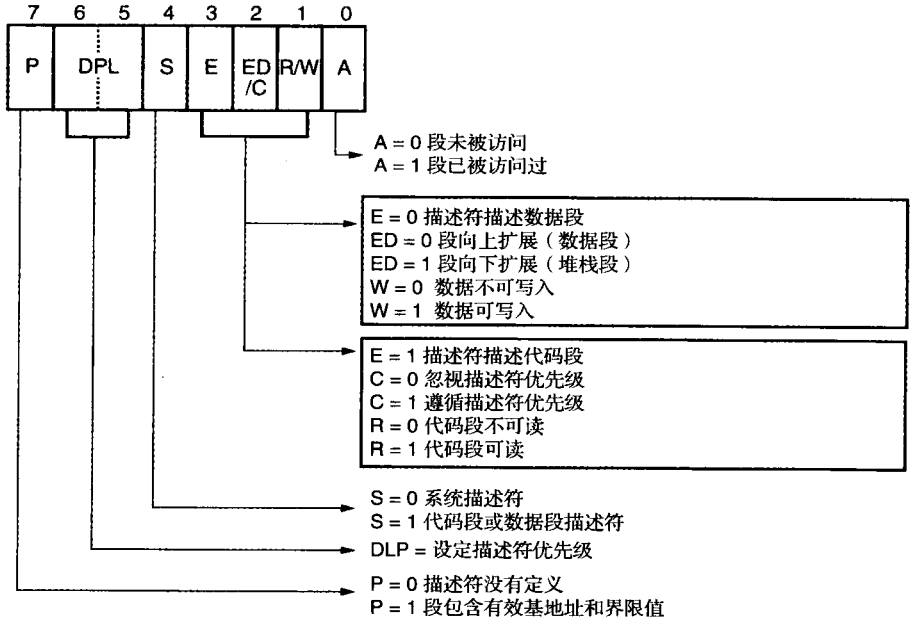


图 2-7 80286 ~ Core2 描述符中的访问权限字节

注：用于描述访问权限位的一些字母与 Intel 文档中标注得不一样。

用户可以指定一个数据段是否为可写或写保护。也可用类似的方式控制代码段，为了保护软件还可以禁止读。注意，在 64 位模式中只有代码段没有其他段描述符类型。一个 64 位平展模式的程序在代码段包含数据和堆栈。

描述符是通过段寄存器从描述符表中选择的。图 2-8 展示了段寄存器在保护模式系统中的功能。段寄存器包括 13 位的选择子字段、表选择子（**TI**）位和请求优先级字段。13 位**选择子**可从描述符表的 8192 个描述符中选择一个。TI 位选择全局描述符表（TI = 0）或局部描述符表（TI = 1）。**请求优先级**（**requested privilege level，RPL**）请求存储器段的访问优先级。最高优先级是 00，最低是 11。如果请求优先级与访问权限字节设定的优先级相匹配或者高于它，则访问得到允许。例如，如果请求优先级是 10，而访问权限字节设定段优先级是 11，则允许访问，因为优先级 10 比优先级 11 高。优先级用于多用户环境中。Windows 将优先级 00（**环 0**）赋予内核和驱动程序，而分派给应用程序为优先级 11（**环 3**）。Windows 不用优先级 01 或 10。如果违背了优先级，则系统通常指示一个应用程序或优先级违例。

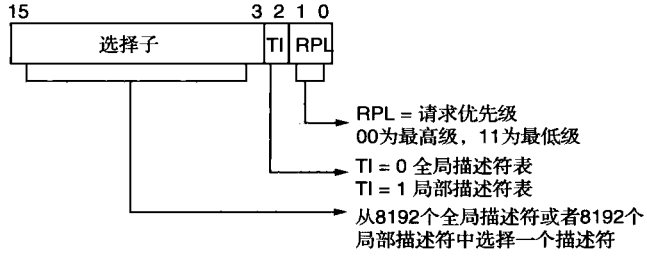


图 2-8 80286 ~ Core2 微处理器在保护模式操作期间段寄存器的内容

图 2-9 展示了含有选择子的段寄存器如何从全局描述符表中选择一个描述符。在全局描述符表内

的一项选择存储器中的一个段。图中 DS 包含 0008H，用请求优先级 00 从全局描述符表中寻址描述符 1。描述符 1 定义基地址为 00100000H，段界限为 000FFH。这意味着，0008H 装入 DS 后使得微处理器使用位于存储器 00100000H ～ 001000FFH 的区域，将这个区域作为数据段。注意描述符 0 称为空描述符，它包括全 0，并且不能用于寻址存储器。

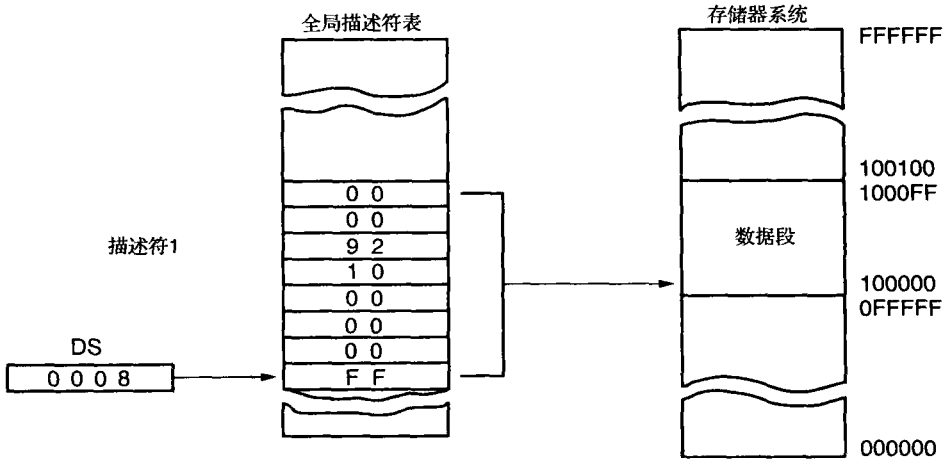


图 2-9 用 DS 寄存器从全局描述符表中选择一个描述符。DS 寄存器寻址位于存储器 100000H ～ 1000FFH 的区域，这个区域作为数据段

2.3.2 程序不可见寄存器

存储系统中有全局描述符表和局部描述符表。为了访问和指定这些表的地址，80286 ～ Core2 微处理器中包含一些程序不可见寄存器。程序不可见寄存器不直接被软件访问，故得此名（虽然其中有些寄存器可以被系统软件访问）。图 2-10 给出了在 80286 ～ Core2 中出现的程序不可见寄存器。在保护模式

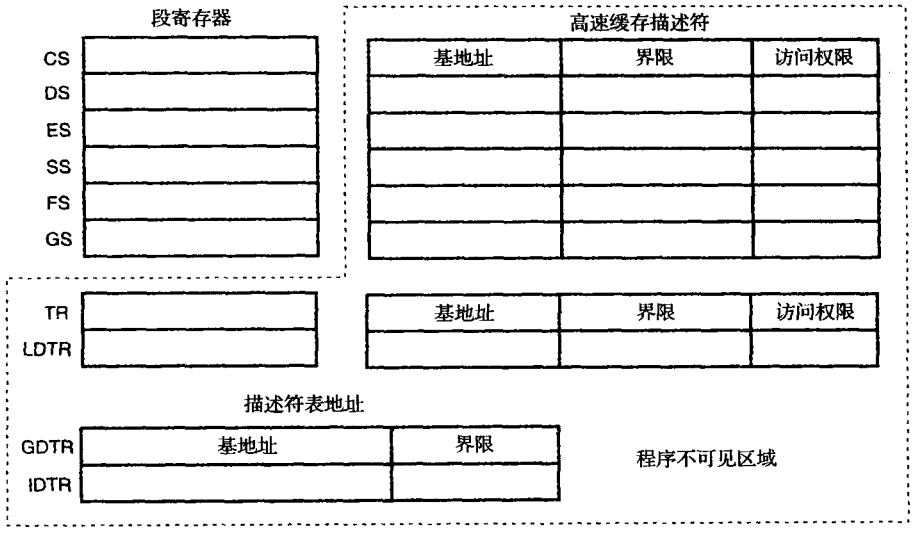


图 2-10 80286 ～ Core2 微处理器中的程序不可见寄存器

注：1. 80286 既没有 FS 和 GS 也没有这些程序不可见寄存器。
2. 80286 有一个 24 位的基地址和一个 16 位的界限。
3. 80386/80486/Pentium/Pentium Pro 有一个 32 位的基地址和一个 20 位的界限。
4. 80286 中访问权限寄存器为 8 位，80386/80486/Pentium ～ Core2 中访问权限寄存器为 12 位。

下操作时，由这些寄存器控制微处理器。

在保护模式中，每个段寄存器都含有一个程序不可见区域。这些寄存器的程序不可见区域通常叫做高速缓冲存储器（cache），因为它也是存储信息的存储器。这些高速缓冲存储器与微处理器中的一级或二级高速缓冲存储器不能混淆。每当段寄存器中的数发生改变时，基地址、界限和访问权限就装入段寄存器的程序不可见区域。当一个新的段号被放入段寄存器里时，微处理器就访问一个描述符表，并把描述符装入该段寄存器的程序不可见高速缓冲存储器区域内。这个描述符一直保存在此处，并在访问内存段时使用，直到段号再次发生变化。这就允许微处理器重复访问一个内存段时，不必每次都去查询描述符表（因此称为高速缓冲存储器）。

GDTR（global descriptor table register，全局描述符表寄存器）和**IDTR（interrupt descriptor table register，中断描述符表寄存器）**包含描述符表的基地址和它的界限。因为描述符表的最大长度为64KB，所以每个表的界限为16位。当工作于保护模式时，全局描述符表地址和它的界限被装入GDTR。

在使用保护模式前，必须初始化中断描述符表和IDTR。后续章节中将提供保护模式操作的详细说明。这里不可能进行程序设计和附加说明这些寄存器。

局部描述符表的位置是从全局描述符表中选择的。为寻址局部描述符表，建立了一个全局描述符。为访问局部描述符表，将选择子装入**LDTR（local descriptor table register，局部描述符表寄存器）**，如同在段寄存器装入选择子一样。这个选择子访问全局描述符表，并且将局部描述符表的基地址、界限和访问权限装入LDTR的高速缓冲存储区。

TR（task register，任务寄存器）包含一个选择子，该选择子用于访问一个确定任务的描述符。任务通常就是过程或应用程序。过程或应用程序的描述符存储在全局描述符表中，因此可通过优先级控制对它的访问。任务寄存器允许在约17μs内完成上下文或任务的切换。任务切换机制使微处理器在足够短的时间内实现任务之间的切换，也使多任务系统以简单而有序的方式从一个任务切换到另一个。

2.4 内存分页

80386及更高型号微处理器的**内存分页机制（memory paging mechanism）**允许为任何线性地址分配任何物理存储器地址。**线性地址（linear address）**定义为由程序产生的地址，而**物理地址（physical address）**是程序访问的实际存储器地址。通过内存分页机制，线性地址透明地转换为任何物理地址，这样就能使需要在特定地址上运行的应用程序通过分页机制重定位，还可以将存储器放在“根本不存在的”存储区域。例如在DOS系统中由EMM386.EXE提供的高端内存块。

EMM386.EXE程序以4KB块为单位，把扩展内存重新分配到视频BIOS和系统BIOS ROM之间的系统存储区，作为高端内存块。没有分页机制，就不可能使用这个存储区。

Windows允许每个应用程序有2G线性地址空间（0000 0000H～7FFFFFFH），而不管是否有足够的存储器与之对应。Windows应用程序能够在硬盘分页和存储器分页上运行，这些分页通过存储器分页部件来实现。

2.4.1 分页寄存器

微处理器中控制寄存器的内容控制着分页部件。控制寄存器CR0到CR3的内容见图2-11。注意，这些寄存器只存在于80386～Core2微处理器中。从Pentium微处理器开始，又增加了一个名叫CR4的控制寄存器，它控制由Pentium成更新的微处理器提供的对基本体系结构的扩展。其特性之一就是置位CR4能使用一个2MB或4MB的页。

对分页部件至关重要的寄存器是CR0和CR3。CR0的最左一位（PG）置位（1）时，就选择分页。如果PG位被清0，则程序产生的线性地址就是用于访问存储器的物理地址。如果PG位置位，则线性地址通过分页机制转换为物理地址。在实模式和保护模式下分页机制都可工作。

CR3的内容包括页目录基地址（或根地址）及PCD位和PWT位。PCD位和PWT位控制微处理器PCD和PWT引脚的操作。如果PCD置位（1），则PCD引脚在非分页总线周期变为逻辑1，这就允许外部硬件控制二级高速缓冲存储器（注意，对于现代版Pentium，二级高速缓冲存储器是内部高速存储

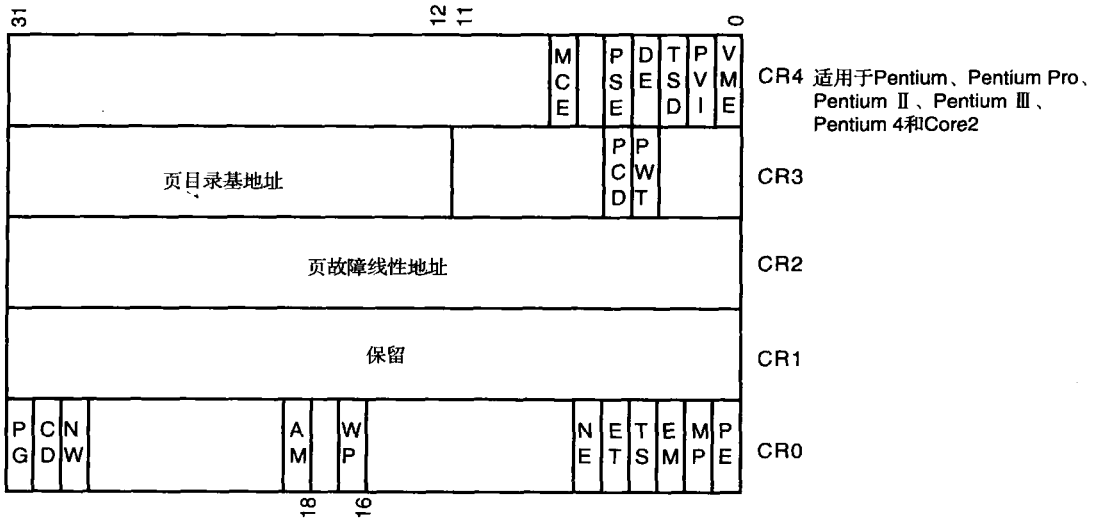


图 2-11 微处理器的控制寄存器结构

器，它是微处理器和主 DRAM 存储系统之间的缓冲器）。PWT 位也在非分页总线周期出现在 PWT 引脚上，用于控制系统中的通写（write-through）高速缓冲存储器。页目录基地址用于为页转换部件寻址页目录。注意，因为是在内部增添了 000H，这个地址将寻址存储系统中以 4KB 为边界的页目录。页目录包含 1024 个目录项，每项长 4 字节。每个页目录项寻址一个包含 1024 项的页表。

由软件生成的线性地址分为三部分，分别用于访问**页目录项**（**page directory entry**）、**页表项**（**page table entry**）和**存储器页偏移地址**（**page offset address**）。图 2-12 表示了线性地址和它的分页结构。注意，最左边 10 位怎样寻址页目录中的一项。对应线性地址 00000000H ～ 003FFFFFH，页目录的第一项被访问。每一个页目录项代表新或重分页内存系统中的一个 4MB 区域。页目录的内容选择由随后的 10 位线性地址（位 12 ～ 21）所指示的页表。这意味着地址 00000000H ～ 00000FFFH 将选择页目

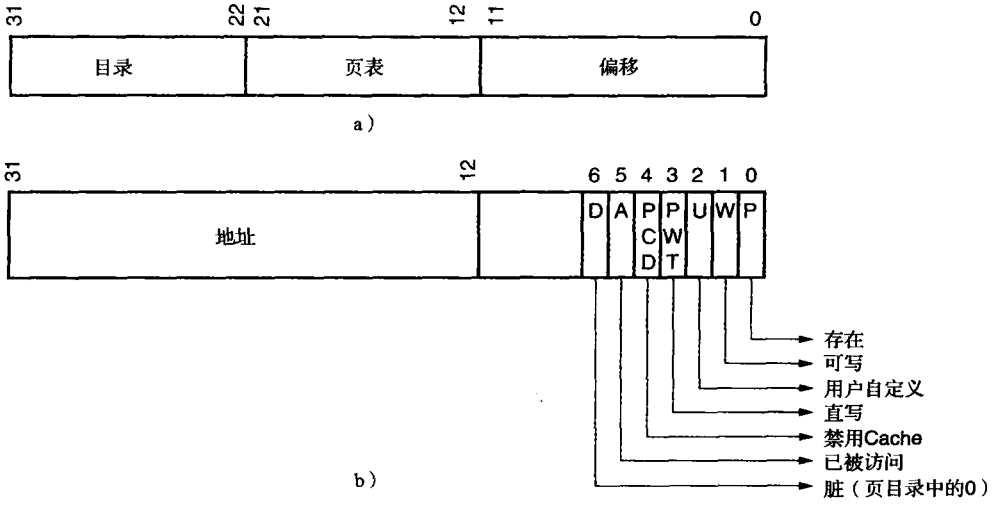


图 2-12 线性地址和宅的颁结构

a）线性地址的格式 b）页目录或者页表项

录项0和页表项0。注意，这是个4KB的区域。线性地址的偏移部分（位0～11）随后选择4KB内存页内的一个字节。在图2-12中，如果页表项0包含地址00100000H，则与线性地址00000000H～00000FFFH对应的物理地址为00100000H～00100FFFH。也就是说，当程序寻址00000000H～00000FFFH之间的地址时，微处理器实际上是寻址00100000H～00100FFFH之间的物理地址。

因为进行4KB存储器区重新分页的操作需要访问存储器内的页目录和页表，所以Intel构造了一个称为**TLB（translation look-aside buffer，转换后备缓冲区）**的高速缓冲存储器。在80486微处理器中，TLB保存了32个最近使用的页转换地址，即最后32个页表转换被存入了TLB中，因此如果访问某个存储区，其地址已经在TLB中，就不需要再访问页目录和页表，这样加速了程序的执行。如果一个页表转换不在TLB中，则必须访问页目录和页表，这就需要额外的执行时间。Pentium～Pentium 4微处理器的每个指令和数据高速缓冲存储器各有一个TLB。

2.4.2 页目录和页表

图2-13展示了页目录、几个页表和一些内存页。在系统中只有一个页目录，页目录包含1024个双字地址，最多可以寻址1024个页表。页目录和每个页表的长度均为4KB。如果将总计4GB的内存分页，那么系统必须为页目录分配4KB存储器空间，为1024个页表分配4KB×1024，即4MB空间。这表示将占用相当大的存储器资源。

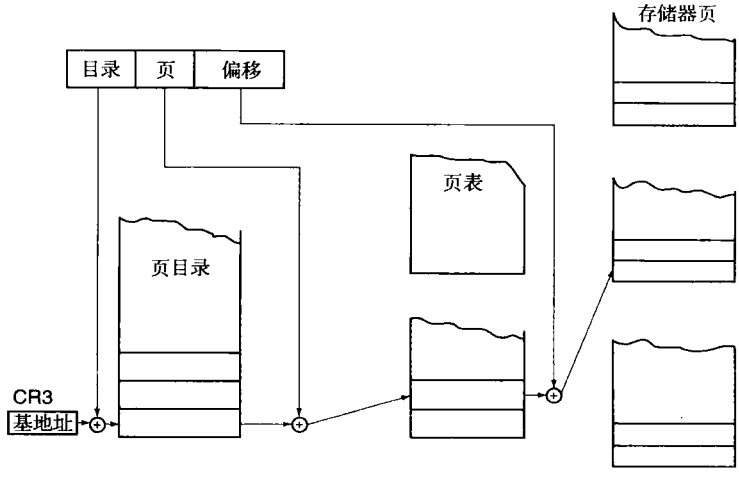


图2-13 80386～Core2微处理器中的分页机制

DOS系统和EMM 386. EXE程序使用页表重定义C8000H～EFFFFH之间的存储区为高端内存块。它将扩展内存重新分页，回填到常规内存的这个部分，以便允许DOS访问额外的存储区。假定EMM386. EXE程序允许通过分页访问常规内存和16MB扩展内存，并且地址C8000H～EFFFFH必须重新分页到地址110000H～138000H，而所有其他存储区分页到正常地址。图2-14描绘了这样一个方案。

这里页目录包含4项，回忆一下，页目录中每一项对应4MB的物理存储器。系统还包括4个页表，每个页表里有1024项，页表内每一项对应重新分页的4KB物理存储器。这个方案要求16KB存储器用于4个页表，还要求16B存储器用于页目录。

与DOS一样，Windows程序也对存储系统重新分页。当前，Windows 3. 11版只支持对16MB内存分页，因为需要大量存储器空间存放页表。新版Windows对全部存储系统重新分页。对于Pentium～Core2微处理器，可以按4KB、2MB或者4MB长度分页。在2MB和4MB变量中，只有1个页面目录和1个内存分页，而没有页表。

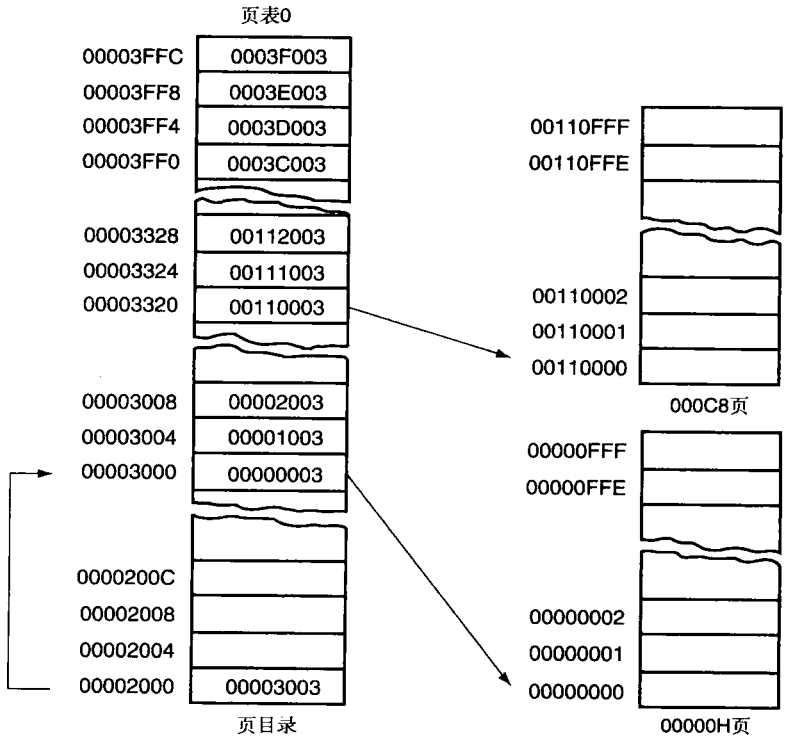


图 2-14 页目录、页表 0 和两个内存页。注意页地址 000C8000H ~ 000C9000H 如何移到 00110000H ~ 00110FFFH

2.5 平展模式内存

采用 64 位扩展的基于 Pentium 处理器的计算机（Pentium 4 或者 Core2）的内存系统为平展模式内存系统。**平展模式内存**系统是不存在分段的系统。内存中第一个字节的地址为 00 0000 0000H，最后的位置为 FF FFFF FFFFH（40 位地址）。平展模式内存不使用段寄存器进行寻址。CS 段寄存器用来从只定义代码段访问权限的描述符表中选择描述符。段寄存器仍然负责选择软件的优先级级别。平展模式不使用描述符中的基址和界限来选择段的内存地址（见图 2-6）。在 64 位模式下，描述符不会像在 32 位模式下那样修改实际内存地址。64 位模式下的偏移地址即实际物理地址。关于平展模式内存模型请参考图 2-15。

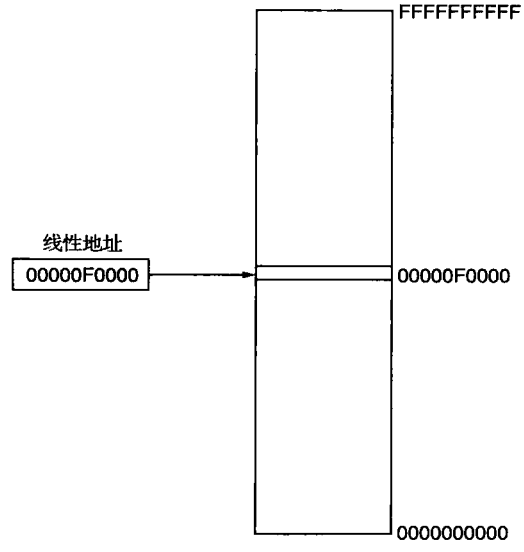


图 2-15 64 位平展模式内存模型

这种寻址方式更加容易理解，但是没有通过硬件为系统提供保护，2.3 节讨论的保护模式系统也是如此。当处理器工作在 64 位模式时，实模式系统是不可用的。64 位模式允许保护机制和页面调度。CS 寄存器仍用于 64 位模式的保护模式操作。

在 64 位模式下如果把地址设置为 IA32 兼容的（当描述符中的 L 位被设置为 0 时），那么地址是 64

位的，但是由于地址中只有40位被引出到地址线，任何超过40位的地址都会被截断。使用偏移地址的指令只能使用32位偏移，即允许从当前指令开始的±2GB的地址范围。这种寻址方式被称为RIP相对寻址，这将在第3章中对其进行解释。立即传送指令允许完全64位寻址和对任意平展模式内存地址的访问。其他指令不允许对4GB以上的地址空间进行访问，因为其偏移量地址仍为32位。

如果Pentium工作在完全64位模式下（当描述符中的L=1时），其地址可为64位或者32位。这将在下一章关于寻址模式的例子中表明，并在第4章得到更详细的阐述。目前多数程序都是工作在IA32兼容模式下的，因此当前各种版本的Windows软件都工作良好，但是随着内存的增大和大多数人对64位计算机的使用，这一切将在几年之内改变。这是另一个揭示工业是如何使得软件被硬件变化淘汰的例子。

2.6 小结

1）8086～80286的程序设计模型包含8位和16位的寄存器。80386及更高型号微处理器的程序设计模型包含8位、16位和扩展的32位寄存器，以及两个附加的16位段寄存器：FS和GS。

2）8位寄存器有AH、AL、BH、BL、CH、CL、DH和DL。16位寄存器有AX、BX、CX、DX、SP、BP、DI和SI。段寄存器有CS、DS、ES、SS、FS和GS。32位扩展寄存器有EAX、EBX、ECX、EDX、ESP、EBP、EDI和ESI。在一个具有64位扩展的Pentium 4的64位寄存器有RAX、RBX、RCX、RDX、RSP、RBP、RDI、RSI和R8～R15。另外，微处理器包含指令指针（IP/EIP/RIP）和标志寄存器（FLAGS、EFLAGS或RFLAGS）。

3）实模式下所有存储器地址都是段地址加偏移地址的组合。段的起始地址由段寄存器内的16位数后添加十六进制0H来确定。偏移地址为16位数，它与20位段地址相加构成实模式的存储器地址。

4）所有指令（代码）由CS（段地址）加IP或EIP（偏移地址）组合寻址。

5）数据通常通过DS（数据段）与偏移地址或者含有偏移地址的寄存器的内容组合来引用。如果8086～Core2微处理器选择16位寄存器，则使用BX、DI和SI作为默认的数据偏移地址寄存器。80386和更高型号的微处理器可以用32位寄存器EAX、EBX、ECX、EDX、EDI和ESI作为默认的数据偏移地址寄存器。

6）在保护模式下，80286～Core2微处理器允许访问起始1MB以上的存储器空间。如同实模式一样，这个扩展内存系统（XMS）也通过段地址加偏移地址访问。不同的只是段地址不存放在段寄存器中。在保护模式下，段起始地址存放在由段寄存器选择的描述符中。

7）在保护模式下，描述符包含基地址、界限和访问权限字节。基地址定位存储器段的起始地址，界限定义段的最大偏移地址，访问权限字节定义程序怎样访问存储器段。80286微处理器使用24位基地址，允许存储器段起始于其16MB存储区域的任何地方。80386及更高型号的微处理器使用32位基地址，允许存储器段起始于其4GB存储区域的任何地方。在80286中，界限为16位数；在80386及更高型号的微处理器中，界限为20位数。因此允许80286存储器段限定为64KB，而80386及更高型号的微处理器的存储器段限定为1MB（G=0）或者4GB（G=1）。在代码描述符中L位选择64位地址操作。

8）在保护模式下，段寄存器包含三个信息字段。段寄存器最左边13位用于从描述符表内的8192个描述符中寻址一个描述符。TI位用于确定访问全局描述符表（TI=0）或局部描述符表（TI=1）。段寄存器最右边2位用于选择对存储器段访问的请求优先级。

9）程序不可见寄存器被80286及更高型号的微处理器用于访问描述符表。每个段寄存器包含一个高速缓冲存储器区，用于在保护模式下保存从描述符中获得的基地址、界限和访问权限。访问存储器段时高速缓冲存储器允许微处理器不必重复访问描述符表，直到段寄存器内容改变。

10）一个内存页为4KB长。由程序产生的线性地址通过80386～Pentium 4微处理器的分页机制可以被映像为任何物理地址。

11）通过控制寄存器CR0和CR3完成内存分页。CR0的PG位使能分页，CR3的内容寻址页目录。页目录包含多达1024个页表地址，用来寻址页表。页表含有1024个页表项，每个页表项用于定位4KB内存页的物理地址。

12）TLB高速缓存最近使用的32个页表转换。如果页表转换保存在TLB中，则不必进行页表转换，从而加速了软件的执行速度。

13）平坦模式存储器使用40位地址总线访问1TB的存储器。未来Intel计划扩展到52位地址总线访问4PB的存储器。只有Pentium 4和Core2的64位可扩展中平展模式才可用。

2.7 习题

1. 什么是程序可见寄存器？

2. 80286可寻址的寄存器为8位和_____位宽。

3. 哪些微处理器能够寻址扩展寄存器?
4. 寻址扩展的 BX 寄存器时，写为_____。
5. 对于某些指令，哪个寄存器用于保存计数值?
6. IP/EIP 寄存器的用途是什么?
7. 哪些算术运算不能修改进位标志位?
8. 如果带符号数 FFH 与 01H 相加，会出现溢出吗?
9. 一个数包含 3 个为 1 的位，它具有_____奇偶性。
10. 哪个标志位控制微处理器的 INTR 引脚。
11. 哪种微处理器包含 FS 段寄存器?
12. 微处理器在实模式下工作时，段寄存器的用途是什么?
13. 在实模式下，段寄存器中装入如下数值，写出每个段的起始地址和结束地址。
    (a) 1000H
    (b) 1234H
    (c) 2300H
    (d) E000H
    (e) AB00H
14. 微处理器工作在实模式下，对于下列 CS：IP 组合，找出要执行的下一条指令的存储器地址。
    (a) CS = 1000H 和 IP = 2000H
    (b) CS = 2000H 和 IP = 1000H
    (c) CS = 2300H 和 IP = 1A00H
    (d) CS = 1A00H 和 IP = B000H
    (e) CS = 3456H 和 IP = ABCDH
15. 实模式存储器地址允许访问低于哪个地址的存储区?
16. 在微处理器中，哪个或哪些寄存器被用作串指令目的的偏移地址?
17. 在 Pentium 4 微处理器中，哪个或哪些 32 位寄存器被用来存放数据段数据的偏移地址?
18. 堆栈存储器由_____段寄存器加_____偏移的组合寻址。
19. 如果用基指针（BP）寻址存储器，则数据包含在_____段内。
20. 80286 工作在实模式下，给出下列寄存器组合所寻址的存储单元地址。
    (a) DS = 1000H 和 DI = 2000H
    (b) DS = 2000H 和 SI = 1002H
    (c) SS = 2300H 和 BP = 3200H
    (d) DS = A000H 和 BX = 1000H
    (e) SS = 2900H 和 SP = 3A00H
21. Core2 在实模式下操作，给出下列寄存器组合所寻址的存储单元地址。
    (a) DS = 2000H 和 EAX = 00003000H
    (b) DS = 1A00H 和 ECX = 00002000H
    (c) DS = C000H 和 ESI = 0000A000H
    (d) SS = 8000H 和 ESP = 00009000H
    (e) DS = 1239H 和 EDX = 0000A900H
22. 保护模式存储器寻址允许访问 80286 微处理器的哪些存储器区域?
23. 保护模式存储器寻址允许访问 Pentium 4 微处理器的哪些存储器区域?
24. 保护模式存储器寻址中，段寄存器的作用是什么?
25. 保护模式的全局描述符表中有多少个描述符是可访问的?
26. 一个 80286 描述符中包含基地址 A00000H 和界限 1000H，由这个描述符寻址的起始地址和结束地址是什么?
27. 一个 Pentium 4 描述符中包含基地址 01000000H 和界限 0FFFFH，并且 G = 0。由这个描述符寻址的起始地址和结束地址是什么?
28. 一个 Pentium 4 微处理器的描述符中含有基地址 00280000H，界限 00010H，并且 G = 1，由这个描述符寻址的起始地址和结束地址是什么?
29. 如果保护模式下 DS 寄存器的内容是 0020H，则哪个全局描述符表项被访问?
30. 如果保护模式下的 DS = 0103H，则请求优先级是_____。
31. 如果保护模式下的 DS = 0105H，则选择了哪个表、表项和请求优先级?
32. Pentium 4 微处理器中，全局描述符表的最大长度是多少?
33. 编码一个描述符，用于描述从 210000H 单元开始至 21001FH 结束的存储器段，该段为可读的代码段。该描述符用于 80286 微处理器。
34. 编码一个描述符，用于描述从 03000000H 单元开始至 05FFFFFFH 单元结束的存储器段。该段是向上增长并且可写的数据段。这个描述符用于 Pentium 4 微处理器。
35. 哪个寄存器寻址全局描述符表?
36. 如何访问在存储系统中的局部描述符表?
37. 微处理器工作于保护模式时，将一个新数装入段寄存器时会发生什么事情?
38. 什么是程序不可见寄存器?
39. GDTR 的用途是什么?
40. 一个内存页内包含多少字节?
41. 在 80386、80486、Pentium、Pentium Pro、Pentium 4 和 Core2 微处理器中，哪种寄存器能使用分页机制?
42. 页目录中存放多少个 32 位地址?
43. 页目录中每一项可把一个多大的线性存储器空间转换为物理存储器空间?
44. 如果微处理器将线性地址 00200000H 送到具有分页机制的系统，哪个页目录项被访问?哪个页表项被访问?
45. 为了将线性地址 20000000H 重定位到物理地址 30000000H，放入页表内的值是什么?
46. 在 Pentium 一类微处理器中设置 TLB 的目的是什么?
47. 利用 Internet 网，写一份详细叙述 TLB 的报告。提示：要到 Intel 网页上查找信息。
48. 在 Intel 网上查找有关分页的论文，并写报告详述系统在各种情况下如何使用分页。
49. 什么是平展模式存储系统?
50. 平展模式存储系统在当前的 Pentium 4 和 Core2 64 位版本中允许这些微处理器访问_____ bytes 的存储器。

第3章 寻址方式

引言

高效率地开发微处理器软件，需要通晓每条指令采用的寻址方式。本章将用 **MOV**（**move data**，**数据传送**）指令说明数据的寻址方式。在 8086～80286 中，MOV 指令在寄存器之间或寄存器与存储器之间传送字节数据或者字数据。在 80386 及更高型号的微处理器中，MOV 指令用来传送字节、字或双字。在描述程序存储器寻址方式时，将用 CALL 和 JMP 指令说明怎样修改程序流程。

8086 到 80286 微处理器的数据寻址方式包括寄存器寻址、立即寻址、直接寻址、寄存器间接寻址、基址加变址寻址、寄存器相对寻址和相对基址加变址寻址。80386 及更高型号的微处理器还包含比例变址方式的存储器数据寻址。程序存储器寻址方式包括程序相对寻址、直接寻址和间接寻址。本章还说明了堆栈存储器的操作，以便充分理解 PUSH 和 POP 指令及其他堆栈操作。

目的

读者学习完本章后将能够：

1）说明每种数据寻址方式的操作。
2）使用数据寻址方式构造汇编语言语句。
3）说明每种程序存储器寻址方式的操作。
4）使用程序存储器寻址方式构造汇编语言和机器语言的语句。
5）为完成给定的任务选择合适的寻址方式。
6）详细说明寻址存储器数据时实模式操作与保护模式操作之间的区别。
7）说明将数据放入堆栈，或从堆栈中取出的顺序。
8）说明怎样将数据结构放入存储器和用于软件。

3.1 数据寻址方式

由于 MOV 指令是个通用且使用灵活的指令，所以我们以它为基础解释数据寻址方式。图 3-1 说明了 MOV 指令，并且定义了数据流向。右边是**源**（**source**），左边是**目的**（**destination**），再左边是操作码 MOV。**操作码**（**opcode**）通知微处理器执行什么操作。大家最初可能不习惯这种适用于所有指令的数据流动方向，因为我们习惯于从左移向右，而这里数据是从右移向左的。注意，指令中目的与源总是用逗号分隔。还要注意，除了 MOVS 指令以外，任何其他指令都不允许存储器到存储器的传送。

图 3-1 中，指令 MOV AX，BX 将源寄存器（BX）的字内容传送到目的寄存器（AX）中。源的内容保持不变，但是目的的内容一般情况下总是要改变的[⊖]。必须记住：MOV 指令总是把源复制到目的中，实际上它并不取走源数据。还要注意，多数的数据传送指令不影响标志寄存器。源和目的通常称为**操作数**（**operand**）。

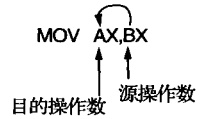


图 3-1 MOV 指令给出了源操作数、目的操作数及数据流向

图 3-2 给出了用于 MOV 指令的所有可能的数据寻址方式。这个图示说明有助于理解 MOV 指令使用的各种数据寻址方式，也可作为应用它的参考。注意，Intel 所有型号的微处理器有相同的寻址方式，比例变址寻址方式除外，它只能用于 80386～Core2。RIP 的相对寻址模式并没在图表中说明，只有 Pentium 4 和 Core2 在 64 位模式下操作

⊖ CMP 和 TEST 指令例外，它们永远不改变目的数据。这些指令将在后续章节中说明。

时才可用。数据寻址方式有：

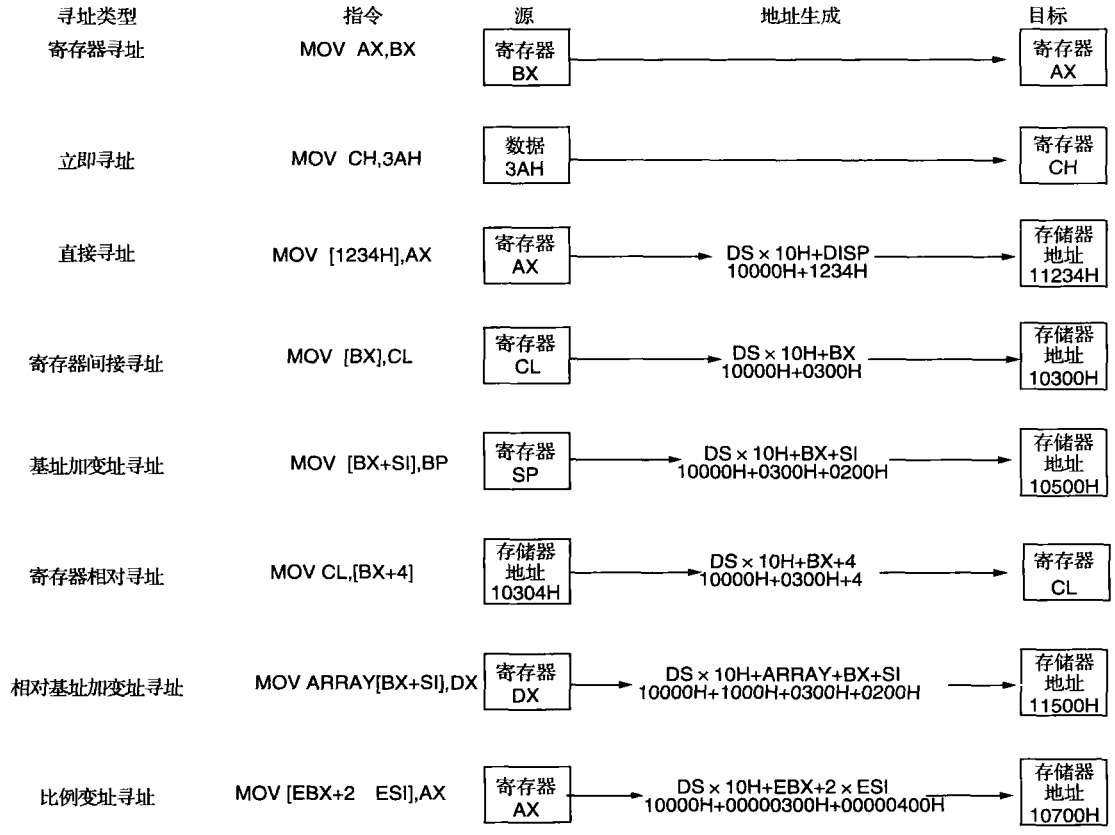


图 3-2 8086 ~ Core2 的数据寻址方式

注：EBX = 00000300H，ESI = 00000200H，ARRAY = 1000H，和 DS = 1000H。

寄存器寻址 把源操作数寄存器或存储单元内的字节或者字的复本传送到目的操作数寄存器或者存储单元（如 MOV CX，DX 指令，将 DX 寄存器的字内容复制到 CX 中）。在 80386 及更高型号的微处理器中，也可从源操作数寄存器或存储单元，把双字传送到目的操作数寄存器或存储单元（如 MOV ECX，EDX 指令，将寄存器 EDX 的双字内容复制到 ECX 中）。Pentium 4 在 64 位模式下操作时，任何 64 位寄存器都是可用的。例如 MOV ROX，RCX 指令，将寄存器 RCX 的四字内容复制到寄存器 RDX。

立即寻址 将源立即数字节、字、双字、四字传送到目的寄存器或存储单元（如 MOV AL，22H 指令，将字节 22H 复制到寄存器 AL 中）。80386 及更高型号的微处理器中，可以将双字的源立即数传送到目的寄存器或存储单元（如 MOV EBX，12345678H 指令，将双字 12345678H 复制到 32 位宽的寄存器 EBX 中）。Pentium 4 或 Core2 的 64 位操作中，只有 MOV 立即寻址指令允许使用 64 位线性地址访问任何存储单元。

直接寻址 在存储单元和寄存器之间直接传送字节或者字。除了 MOVS 指令以外，指令系统不支持存储器到存储器的传送（如 MOV CX，LIST 指令，将存储单元 LIST 的字内容复制到寄存器 CX 中）。80386 及更高型号的微处理器中，也能寻址双字的存储单元（如 MOV ESI，LIST 指令，将存储在地址 LIST 处 4 个连续字节中的 32 位

数复制到寄存器 ESI 中）。在 64 位模式中直接寻址指令使用 64 位线性地址。

寄存器间接寻址 在寄存器和存储单元之间传送字节或者字，而存储单元由变址或基址寄存器寻址。变址和基址寄存器是 BP、BX、DI 和 SI（如 MOV AX，［BX］指令，将数据段中 BX 作为偏移地址的存储单元的字数据复制到寄存器 AX 中）。80386 及更高型号的微处理器中，可在寄存器与存储单元之间传送一个字节、字或双字数据，存储单元用 EAX、EBX、ECX、EDX、EBP、EDI 或 ESI 寄存器寻址（如 MOV AL，［ECX］指令，将数据段中的一个字节数据装入 AL 中，该数据所在存储单元的偏移地址由 ECX 的内容确定）。在 64 位模式下，间接地址仍保持为 32 位大小，这意味着如果程序工作在 32 位兼容模式下，那么目前这种寻址形式只允许访问4GB 的地址空间。在完全64 位模式下，对任何地址的访问都采用 64 位地址或者包含在寄存器内的地址。

基址加变址寻址 在寄存器和存储单元之间传送字节或者字，该存储单元由基址寄存器（BP 或 BX）加变址寄存器（DI 或 SI）寻址（如 MOV［BX + DI］，CL 指令，将寄存器 CL 的字节内容复制到数据段中 BX 加 DI 寻址的存储单元中）。在 80386 及更高型号的微处理器中，寄存器 EAX、EBX、ECX、EDX、EBP、EDI 或 ESI 可以组合生成存储单元地址（如 MOV［EAX + EBX］，CL 指令，将寄存器 CL 的字节内容复制到数据段中由 EAX 加 EBX 寻址的存储单元中）。

寄存器相对寻址 在寄存器和变址寻址的存储单元或基址寄存器加位移量寻址的存储单元之间传送字节或字数据（如 MOV AX，［BX +4］或 MOV AX，ARRAY［BX］。第一条指令将数据段中由 BX 加 4 寻址的单元的内容装入 AX。第二条指令将数据段中由 ARRAY 加 BX 内容寻址的存储单元中的数据装入 AX）。80386 及更高型号的微处理器用任何 32 位寄存器（除 ESP 以外）寻址存储器（如 MOX AX，［ECX +4］或 MOV AX，ARRAY［EBX］。第一条指令将数据段中由 ECX 加 4 寻址的存储单元的数据装入 AX。第二条指令将数据段中由 ARRAY 加 EBX 内容寻址的存储单元的数据装入 AX）。

相对基址加变址寻址 在寄存器和存储单元之间传送字节或字数据，该存储单元是由基址寄存器加变址寄存器再加位移量寻址的（如 MOV AX，ARRAY［BX + DI］或 MOV AX，［BX + DI +4］。这两条指令都将数据从数据段的存储单元装入 AX。第一条指令用 ARRAY、BX 和 DI 相加形成存储单元地址；第二条指令用 BX、DI 和 4 相加形成存储单元地址）。对于 80386 及更高型号的微处理器，MOV EAX，ARRAY［EBX + ECX］指令将数据段中由 ARRAY、EBX 及 ECX 之和寻址的存储单元的数据装入 EAX。

比例变址寻址 这种寻址方式只能用于 80386 ～ Pentium 4 微处理器。一对寄存器中的第二个寄存器内容用 2 倍、4 倍或 8 倍比例因子修改，产生操作数的存储器地址（如 MOV EDX，［EAX +4 * EBX］指令，将数据段中地址为 EAX 加上 4 倍 EBX 的存储单元的内容装入 EDX）。比例因子允许存取存储器中数组的字（2 ×）、双字（4 ×）或者四字（8 ×）数据。注意，也存在 1 倍的比例因子，但它一般不在指令中明确写出。MOV AL，［EBX + ECX］就是比例因子为 1 的例子。换句话说，这条指令也可以写成 MOV AL，［EBX + 1 * ECX］。另一个例子是指令 MOV AL，［2 * EBX］，它只用一个比例寄存器寻址存储器。

RIP 相对寻址 这种寻址模式只能用于 Pentium 4 或者 Core2 上的 64 位扩展。通过向 64 位指令指针的 64 位内容增加一个 32 位的偏移，这一模式允许对内存系统中的任意位置进行访问。例如，如果 RIP = 1000000000H，一个 32 位偏移为 300H，那么被访问的位置为 1000000300H。偏移是有符号的，因此位于指令 ±2GB 范围内的

数据都可以通过这一寻址模式访问。

3.1.1 寄存器寻址

寄存器寻址是最通用的数据寻址方式，只要记住寄存器名，就很容易使用。微处理器包含下列用于寄存器寻址的8位寄存器：AH、AL、BH、BL、CH、CL、DH和DL，也可以用以下16位寄存器：AX、BX、CX、DX、SP、BP、SI和DI。在80386及更高型号的微处理器中，扩展的32位寄存器是：EAX、EBX、ECX、EDX、ESP、EBP、EDI和ESI。Pentium 4的64位模式下的寄存器有：RAX、RBX、RCX、RDX、RSP、RBP、RDI、RSI和R8～R15。有些MOV指令及PUSH和POP指令中，寄存器寻址方式可用16位的段寄存器（CS、ES、DS、SS、FS和GS）。指令中使用相同长度的寄存器是很重要的。8位寄存器与16位寄存器，8位寄存器与32位寄存器，或16位寄存器与32位寄存器，都绝不能混用，这是微处理器所不允许的，否则汇编时会发生错误。同样不能混淆64位寄存器和其他任何大小的寄存器。像MOV AX，AL或MOV EAX，AL指令看起来似乎是讲得通，但也不允许出现，因为这些寄存器的长度不同。少数指令，例如SHL DX，CL指令，是这个规则的例外，我们将在后续章节中进行说明。注意，MOV指令都不影响标志位，这一点也很重要。标志位通常由算术或逻辑指令修改。

表3-1给出了多种寄存器传送指令的例子。要给出全部可能的组合是不可能的，因为实在太多了，例如，8位MOV指令的子集就有64种类型。只就寄存器类MOV指令而言，不允许段寄存器到段寄存器的MOV指令。注意，代码段寄存器不能用MOV指令改变，因为下一条指令的地址需要由IP/EIP和CS两者共同确定。如果只改变了CS寄存器的值，下一条指令的地址将是不可知的，因此不允许用MOV指令改变CS寄存器。

表3-1 寄存器寻址指令的例子

汇编语句	长度	操作
MOV AL, BL	8位	把BL复制到AL中
MOV CH, CL	8位	把CL复制到CH中
MOV R8B, CL	8位	把CL复制到为R8的字节部分（64位模式下）
MOV R8B, CH	8位	不允许
MOV AX, CX	16位	把CX复制到AX中
MOV SP, BP	16位	把BP复制到SP中
MOV DS, AX	16位	把AX复制到DS中
MOV BP, R10W	16位	把R10复制到BP（64位模式下）
MOV SI, DI	16位	把DI复制到SI中
MOV BX, ES	16位	把ES复制到BX中
MOV ECX, EBX	32位	把EBX复制到ECX中
MOV ESP, EDX	32位	把EDX复制到ESP中
MOV EDX, R9D	32位	把R9复制到EDX（64模式下）
MOV RAX, RDX	64位	把RDX复制到RAX
MOV DS, CX	16位	把CX复制到DS中
MOV ES, DS	—	不允许（段到段）
MOV BL, DX	—	不允许（长度不同）
MOV CS, AX	—	不允许（代码段寄存器不能作为目的寄存器）

图3-3给出了MOV BX，CX指令的操作。注意，源寄存器的内容不变，但是目的寄存器改变了。这条指令将CX寄存器中的1234H传送（复制）到BX寄存器中。它抹除了BX寄存器原来的内容(76AFH)，但是CX的内容保持不变。除了CMP和TEST指令以外，所有指令中目的寄存器或目的存储单元的内容都会改变。注意MOV BX，CX指令不影响EBX寄存器最左边的16位。

例3-1给出了在8位、16位和32位寄存器之间复制各种数据的汇编语言指令序列。如上所述，从一个寄存器到另一个寄存器传送数据的操作只改变目的寄存器的内容，从来不改变源寄存器。这个例

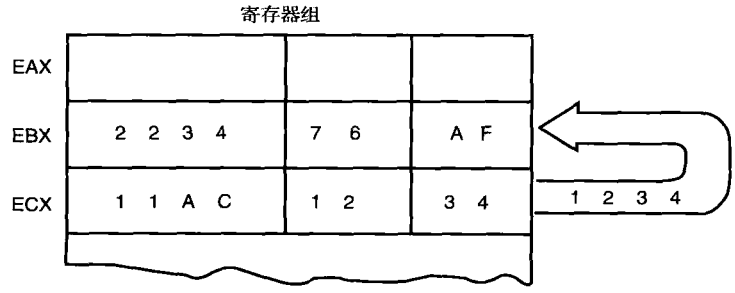


图 3-3 MOV BX，CX 指令执行的结果，此时为 BX 寄存器改变前的一瞬间

注：只是 EBX 寄存器右边的 16 位发生了变化。

子的最后一条指令（MOV CS，AX）汇编时没有错误，但是如果执行就会出现问题。因为，如果只改变 CS 寄存器的内容而不改变 IP，程序的下一步将是不可知的，将会导致程序出错。

例 3-1

```
0000 8B  C3      MOV AX,BX        ;把 BX 的内容复制到 AX
0002 8A  CE      MOV CL,DH        ;把 DH 的内容复制到 CL
0004 8A  CD      MOV CL,CH        ;把 CH 的内容复制到 CL
0006 66 8B C3    MOV EAX,EBX      ;把 EBX 的内容复制到 EAX
0009 66 8B D8    MOV EBX,EAX      ;把 EAX 的内容复制到 EBX
000C 66 8B C8    MOV ECX,EAX      ;把 EAX 的内容复制到 ECX
000F 66 8B D0    MOV EDX,EAX      ;把 EAX 的内容复制到 EDX
0012 8C  C8      MOV AX,CS        ;把 CS 的内容复制到 DS（两步）
0014 8E  D8      MOV DS,AX
0016 8E  C8      MOV CS,AX        ;复制 AX 到 CS（汇编了，但是将会出现问题）
```

3.1.2 立即寻址

另一种数据寻址方式是立即寻址。术语**立即数（immediate）**意味着在存储器中数据紧接着放在十六进制操作码后面。注意，立即数是**常数（constant data）**，而由寄存器或存储单元传送的数据是**变数（variable data）**。立即寻址可操作字节或者字数据。在 80386 ～ Core2 微处理器中，立即寻址也操作双字数据。立即 MOV 指令将立即数的副本传送到寄存器或存储单元。图 3-4 给出了 MOV EAX，13456H 指令的操作。这条指令将指令中的 13456H 复制到寄存器 EAX 中，在存储器中该立即数紧跟在十六进制操作码后面。与图 3-3 所示的 MOV 指令一样，源数据覆盖了目的数据。

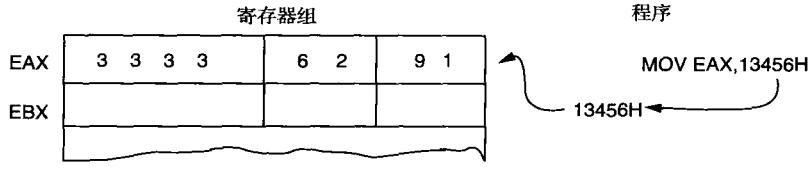


图 3-4 MOV EAX，13456H 指令的操作，这条指令将立即数 13456H 复制到 EAX 中

在符号汇编语言中，符号#在某些汇编程序中放在立即数的前面。如 MOV AX，#3456H 指令。但多数汇编程序不使用#符号，而是像指令 MOV AX，3456H 这样表示立即数。本书中立即数之前不使用#符号。最常用的汇编程序，如 Intel ASM、Microsoft MASM[⊖] 及 Borland TASM[⊖]，都不使用#符号表示立

⊖ MASM（MACRO 汇编程序）是 Microsoft 公司的注册商标。

⊖ TASM（Turbo 汇编程序）是 Borland 公司的注册商标。

即数，但是一些老的汇编程序，如用于 Hewlett-Packard 的逻辑开发系统的汇编程序就使用它，可能还有其他系统也用它。

如上所述，MOV 立即指令在 64 位操作下可以包括 64 位立即数。在 64 位模式如 MOV RAX, 123456780A311200H 是允许的。

符号汇编程序用许多方式表示立即数。用字母 H 表示十六进制数。如果十六进制数以字母开头，则汇编程序要求在它前面加一个 0。例如，汇编语言用 0F2H 表示十六进制数 F2。在某些汇编程序中，用 'h 表示十六进制数，例如 MOV AX, #'h1234 中的十六进制数（在 MASM、TASM 和本书中不用）。十进制数不要求特殊的代码和符号，例如 MOV AL, 100 指令中的十进制数 100。如果用撇号将 ASCII 码括起来，一个 ASCII 码字符或几个字符可表示为立即数。例如，MOV BH, 'A' 指令，把 A 的 ASCII 码（41H）传送到寄存器 BH。注意，对 ASCII 码数据要使用撇号（'）标识，而不是单引号（`）。二进制数后面跟着字母 B 时，表示该数据是二进制数据，有些汇编程序中用字母 Y 表示。表 3-2 给出了多种使用立即数的 MOV 指令。

表 3-2 使用立即寻址的 MOV 指令示例

汇编语句	长度	操作
MOV BL, 44	8 位	把十进制数 44（2CH）传送到 BL 中
MOV AX, 44H	16 位	把十六进制数 44 传送到 AX 中
MOV SI, 0	16 位	把 0000H 传送到 SI 中
MOV CH, 100	8 位	把十进制数 100（64H）传送到 CH 中
MOV AL, 'A'	8 位	把 ASCII A（41H）传送到 AL 中
MOV AH, 1	8 位	64 位模式下不允许，在 32 位和 16 位模式下允许
MOV AX, 'AB'	16 位	把 ASCII 码 BA① (4241H) 传送到 AX 中
MOV CL, 1100 1110B	8 位	把二进制数 1100 1110 传送到 CL 中
MOV EBX, 1234 0000H	32 位	把 12340000H 传送到 EBX 中
MOV ESI, 12	32 位	把十进制数 12 传送到 ESI 中
MOV EAX, 100B	32 位	把二进制数 100 传送到 EAX 中
MOV RCX, 100H	64 位	把 100H 复制到 RCX

① 这不是错误，因为当用一个字存储两个 ASCII 字符时，ASCII 字符存储为 BA。

例 3-2 给出了包含几种立即指令的小程序，程序将 0000H 放入 16 位寄存器 AX、BX 和 CX 中，然后用寄存器寻址指令把 AX 的内容复制到寄存器 SI、DI 和 BP 中。这是一个使用程序设计模型的完整程序，可以用 MASM 汇编并执行。.MODEL TINY 语句指示汇编程序把这个程序汇编成一个代码段。.CODE 语句或伪指令，指明代码段的开始。.STARUP 语句指明程序中指令的开始，而 .EXIT 语句使程序返回到 DOS。END 语句指明程序文件结束。这个程序可以用 MASM 汇编，可用 CodeView⊖（CV）观察它的执行。注意：最新版本 TASM 在不加任何修改的情况下也接受 MASM 代码。用 DOS EDIT 程序、Windows NotePad⊜或者 Programmer's WorkBench⊜（PWB），可以把程序存储到系统中。注意，TINY 程序总是汇编成命令（.COM）程序。

例 3-2

```
                    .MODEL TINY          ;选择 TINY 模型
0000                .CODE                ;指示代码段的开始
                    .STARTUP             ;指示程序的开始
0100  B8  0000      MOV AX,0             ;把 0000H 放入 AX
```

⊖ CodeView 是 Microsoft 公司的注册商标。

⊜ Windows NotePad 是 Microsoft 公司的注册商标。

⊜ Programmer's WorkBench 是 Microsoft 公司的注册商标。

```
0103  BB 0000      MOV BX,0              ;把0000H放入BX
0106  B9 0000      MOV CX,0              ;把0000H放入CX

0109  8B F0        MOV SI,AX             ;复制AX到SI
010B  8B F8        MOV DI,AX             ;复制AX到DI
010D  8B E8        MOV BP,AX             ;复制AX到BP

                   .EXIT                 ;返回到DOS
                   END                   ;程序结束
```

汇编语言程序中的每条语句由4个字段组成，如例3-3所示。最左边的字段称为标号label字段，用来存放它所代表的存储单元的符号名。所有的标号必须以字母或者下列特殊符号之一开始：@、$、-或?。标号的长度只能是1～35个字符。程序中的标号用来标识存放数据的存储单元或者用于其他目的，本书后面将会讲解有关内容。下一个字段称为操作码字段，用于存放指令或操作码。数据传送指令中的MOV就是它的操作码。操作码右边的字段是操作数字段，容纳操作码使用的信息。例如，MOV AL，BL指令有操作码MOV和操作数AL及BL。注意，一些指令包含有0～3个操作数。最后的字段是注释（comment）字段，存放有关指令或指令组的注释。注释总是以分号（;）开始。

例3-3

```
LABEL    OPCODE    OPERAND    COMMENT

DATA1    DB        23H        ;定义DATA1为字节23H
DATA2    DW        1000H      ;定义DATA2为字1000H

START:   MOV       AL,BL      ;把BL的内容复制到AL
         MOV       BH,AL      ;把AL的内容复制到BH
         MOV       CX,200     ;把十进制数200装入CX
```

当程序被汇编后，生成的清单（.LST）如例3-2所示。其中最左边的十六进制数字是指令或数据的偏移地址，这些数字是由汇编程序生成的。偏移地址右边的数字是指令的机器码或者是数据，也由汇编程序生成。例3-2中，文件中有一条MOV AX，0指令，文件被汇编后，该指令出现在0100存储单元中，它的十六进制机器语言形式是B8 0000。B8是机器语言操作码，0000是数值0的16位数据。写入程序时，只需在编辑程序中键入MOV AX，0，由汇编程序生成它的机器码和地址，并且把程序存储到带有扩展名.LST的文件中。本书中所有的程序都是以汇编程序生成的形式给出的。

程序也可以用内嵌汇编程序写在Visual C++程序中，例3-4给出一个Visual C++程序函数，其中包括一些用内嵌汇编程序写的代码。该函数把一个数加上20H后再返回。注意，汇编代码访问C++变量temp，所有汇编代码放在一个_asm代码块中。本书中许多例子是用内嵌汇编程序与C++程序写的。

例3-4

```
int MyFunction (int temp)
{
       _asm
       {
              mov eax,temp
              add eax,20h
              mov temp,eax
       }
       return temp;          //返回一个32位的整数
}
```

3.1.3 直接数据寻址

多数指令可以使用直接数据寻址方式。事实上，典型程序中的许多指令都采用直接数据寻址。直

接数据寻址有两种基本形式：1）**直接寻址（direct addressing）**，用于存储单元与 AL、AX 或 EAX 之间的 MOV 指令。2）**位移量寻址（displacement addressing）**，用于指令系统中几乎所有的指令。无论哪种情况，都是把位移量加到默认的数据段地址或其他段地址上形成地址。在 64 位操作中，直接寻址的指令也可以采用 64 位线性地址，这使得可以对任何内存位置进行访问。

直接寻址

MOV 指令所用的直接寻址在数据段存储单元与 AL（8 位）、AX（16 位）和 EAX（32 位）寄存器之间传送数据。这种指令通常是 3 字节长的指令（在 80386 及更高型号的微处理器中，指令前面可能出现一个寄存器长度的前缀，使它超过 3 字节长）。

正如多数汇编程序表示的那样，MOV AL，DATA 指令将数据段存储单元 DATA（1234H）中的数据装入 AL，DATA 是存储单元的符号地址，1234H 是实际的十六进制地址。对某些汇编程序，这条指令可以表示为 MOV AL，[1234H]⊖。[1234H] 是绝对存储单元地址，不是所有汇编程序都允许的。注意，某些汇编程序可能要求 MOV AL，DS:[1234H] 这样的指令形式，指出是数据段内的地址。图 3-5 指出如何将存储单元 11234H 中的字节内容复制到 AL 中。在以实模式操作的系统中，它的有效地址由 1234H（偏移地址）加 10000H（数据段地址 1000H 乘以 10H）形成。

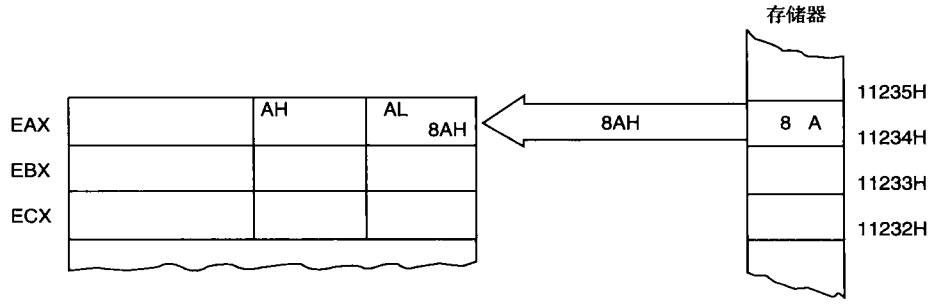


图 3-5 当 DS = 1000H 时，MOV AL，[1234H] 指令的操作

表 3-3 列出了 3 种直接寻址的指令。这些指令经常出现在程序中，因此 Intel 为了减少程序的长度，规定其为 3 字节长的指令。所有其他将数据从存储单元移动到寄存器的指令，称为**位移量寻址指令（displacement-addressed instruction）**，需要 4 个或更多字节。

表 3-3 使用 EAX、AX、AL 和 64 位模式下的 RAX 的直接寻址指令

汇编语句	长 度	操 作
MOV AL，NUMBER	8 位	将数据段存储单元 NUMBER 中的字节内容复制到 AL 中
MOV AX，COW	16 位	将数据段存储单元 COW 中的字内容复制到 AX 中
MOV EAX，WATER①	32 位	将数据段存储单元 WATER 中的双字内容复制到 EAX 中
MOV NEWS，AL	8 位	将 AL 的内容复制到字节存储单元 NEWS 中
MOV THERE，AX	16 位	将 AX 的内容复制到字存储单元 THERE 中
MOV HOME，EAX①	32 位	将 EAX 的内容复制到双字存储单元 HOME 中
MOV ES:[2000H]，AL	8 位	将 AL 的内容复制到附加数据段存储单元 2000H 中
MOV AL，MOUSE	8 位	将 MOUSE 单元的内容复制到 AL；在 64 位模式中 MOUSE 可以是任何地址
MOV RAX，WHISKEY	64 位	将存储单元 WHISKEY 的 8 个字节复制到 RAX 中

① 80386 ~ Pentium 4 微处理器为了在 EAX 与存储器之间移动 32 位数，有时需要多于 3 字节的存储器。

位移量寻址

除了指令是 4 字节而不是 3 字节以外，位移量寻址几乎等同于直接寻址。在 80386 ~ Pentium 4 中，

⊖ 这种格式可在 MASM 中使用，但它更常出现在使用调试工具的时候。

规定用32位寄存器和32位位移量，这些指令可以长达7字节。因为许多指令使用这种类型的直接数据寻址，所以它是最灵活的。

如果将MOV CL，DS:[1234H]指令与MOV AL，DS:[1234H]指令对比，两者基本上执行相同的操作，只是目的寄存器不同（CL变为AL）。根据汇编程序对这两条指令的解释，另一个区别很直观，MOV AL，DS:[1234H]指令是3字节长，而MOV CL，DS:[1234H]指令是4字节长，如例3-5所示。这个例子说明了汇编程序怎样将这两条指令汇编为十六进制机器语言。在例子中，在指令的:[偏移量]前面，必须有段寄存器DS:，可以用任何段寄存器，但多数情况下数据是存在数据段中，所以该例用DS:[1234H]。

例3-5

```
0000 A0 1234 R        MOV AL,DS:[1234H]
0003 BA 0E 1234 R     MOV CL,DS:[1234H]
```

表3-4列出了某些使用位移直接寻址的MOV指令。因为这类指令太多，没有列出所有的类型。注意，可以由段寄存器存入存储器，也可以由存储器装入段寄存器。

表3-4 使用位移量的直接数据寻址的示例

汇编语句	长 度	操 作
MOV CH，DOG	8位	把数据段存储单元DOG的字节内容装入CH中（DOG的偏移地址由汇编程序计算）
MOV CH，DS:[1000H]①	8位	把数据段存储单元1000H的字节内容装入CH中
MOV ES，DATA 6	16位	把数据段存储单元DATA6的字内容装入ES中
MOV DATA7，BP	16位	把寄存器BP的内容复制到数据段存储单元DATA7中
MOV NUMBER，SP	16位	把SP的内容复制到数据段存储单元NUMBER中
MOV DATA1，EAX	32位	把EAX的内容复制到数据段存储单元DATA1中
MOV EDI，SUM1	32位	把数据段存储单元SUM1的双字内容装入EDI中

① 多数汇编程序很少使用这种寻址模式，因为在程序中很少访问实际数字的偏移地址。

例3-6给出了使用模型的短程序，寻址数据段中的信息。注意数据段（data segment）以.DATA语句开始，通知汇编程序数据段从哪里开始。模型的规模由例3-3的TINY调整为SMALL，因而包含了一个数据段。SMALL模型允许一个数据段和一个代码段，当程序需要存储器数据时，通常使用SMALL模型。SMALL模型程序汇编成执行（.EXE）程序文件。注意这个例子怎样使用DB和DW伪指令在数据段中分配存储单元。.STARTUP语句不仅指示代码段的开始，也将数据段的段地址装入数据段寄存器。如果这个程序用CodeView汇编并且执行，则程序执行时能观察到指令、寄存器及存储单元的变化。

例3-6

```
                        .MODEL SMALL        ;选择SMALL模型
0000                    .DATA               ;指示数据段的开始

0000 10             DATA1 DB  10H           ;把10H存入DATA1
0001 00             DATA2 DB  0             ;把00H存入DATA2
0002 0000           DATA3 DW  0             ;把00H存入DATA3
0004 AAAA           DATA4 DW  0AAAAH        ;把0AAAAH存入DATA4

0000                    .CODE               ;指示代码段的开始
                        .STARTUP            ;指示程序的开始

0017 A0 0000 R          MOV AL,DATA1        ;把DATA1的内容复制到AL
001A 8A 26 0001 R       MOV AH,DATA2        ;把DATA2的内容复制到AH
```

```
001E  A3  0002  R          MOV DATA3,AX      ;把 AX 存入 DATA3
0021  8B  1E  0004  R      MOV BX,DATA4      ;把 DATA4 存入 BX

                           .EXIT             ;返回到 DOS
                           END               ;程序结束
```

3.1.4 寄存器间接寻址

寄存器间接寻址允许寻址任何存储单元的数据，通过下面这些寄存器保存偏移地址：BP、BX、DI和SI。例如，如果寄存器BX的内容是1000H，那么执行MOV AX，[BX]指令后，数据段偏移地址1000H处的字内容被复制到AX寄存器中。如果微处理器按实模式操作，而且DS=0100H，那么这条指令寻址存放在存储器2000H和2001H中的字，并且将它们传送到寄存器AX（见图3-6）。注意2000H的内容送到AL，而2001H的内容送到AH。符号[ ]在汇编语言中指示间接寻址。除了用BP、BX、DI和SI寄存器作间接寻址存储器以外，80386和更高型号微处理器的寄存器间接寻址允许使用ESP以外的任何扩展寄存器。采用间接寻址的一些典型指令列在表3-5中。如果有工作在64位模式的Pentium 4和Core2，那么可以使用任意64位寄存器来保存一个64位线性地址。在64位模式下，段寄存器不用于平展模型的寻址。

表3-5 寄存器间接寻址的示例

汇编语句	长度	操作
MOV CX，[BX]	16位	把数据段中由BX寻址的存储单元的字内容复制到CX中
MOV [BP]，DL①	8位	把寄存器DL的内容复制到堆栈段由BP寻址的存储单元中
MOV [DI]，BH	8位	把寄存器BH的内容复制到数据段由DI寻址的存储单元中
MOV [DI]，[BX]	—	除了串指令以外，不允许存储器到存储器的传送
MOV AL，[EDX]	8位	把数据段由EDX寻址的存储单元的字节内容复制到AL
MOV ECX，[EBX]	32位	把数据段由EBX寻址的存储单元的双字内容复制到ECX
MOV RAX，[RDX]	64位	把RDX中由线性地址确定的存储单元的四字内容复制到RAX（64位模式下）

① 由BP或EBP寻址的数据默认为在堆栈段中，而所有其他间接寻址指令默认使用数据段。

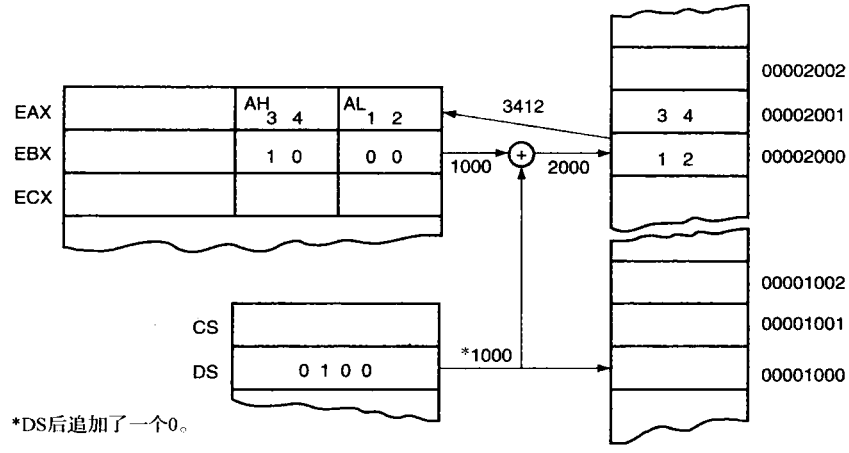


图3-6 BX=1000H和DS=0100H时，MOV AX，[BX]指令的操作

注：图示为存储器的内容传送给AX之后。

当用BX、DI和SI寻址存储器时，寄存器间接寻址或任何其他寻址模式默认使用**数据段**（**data segment**）。如果用寄存器BP寻址存储器，则默认使用**堆栈段**（**stack segment**）。这些被认为是对这4个变址和基址寄存器的默认设置。对于80386及更高型号的微处理器，EBP默认寻址堆栈段中的存储器，而EAX、EBX、ECX、EDX、EDI和ESI默认寻址数据段中的存储器。在实模式下用32位寄存器

寻址存储器时，32 位寄存器的内容不允许超过 0000FFFFH。在保护模式下，只要不访问由访问权限字节规定的段之外的存储单元，任何值都可以在用于间接寻址寄存器的 32 位寄存器中使用。例如，在 80386～Pentium 4 中，MOV EAX，[EBX] 指令将位于数据段并由 EBX 提供偏移地址的存储器中的双字数据装入 EAX。在 64 位模式下，段寄存器并不用于地址计算，这是因为寄存器包含真实的线性存储地址。

有些情况下，间接寻址要求用**特殊汇编伪指令**（**special assembler directive**）BYTE PTR、WORD PTR、DWORD PTR 规定传送数据的长度。这些伪指令指明了由存储器指针（PTR）寻址的存储器数据的长度。例如，MOV AL，[DI] 指令清楚地表明是字节传送指令，而 MOV [DI]，10H 指令是含糊的。MOV [DI]，10H 指令是寻址字节、字、双字还是四字长度的存储单元？汇编程序不能确定 10H 的长度。指令 MOV BYTE PTR [DI]，10H 清楚地指出 DI 寻址的存储单元是字节存储单元。同样的，MOV DWORD PTR [DI]，10H 清楚地定义存储单元是双字长度型。BYTE PTR、WORD PTR 和 DWORD PTR 伪指令只用于带有立即数的通过指针或变址寄存器寻址的存储单元，以及后续章节中将会描述的少量其他指令中。另一个偶尔用到的伪指令是 QWORD PTR，这里的 QWORD 是指四字（64 位）。如果程序使用 SIMD 指令，则 QWORD PTR 也用来表示 128 位宽的数据。

间接寻址通常允许程序引用存储系统中的数据表。例如，假定要建立一个来自 0000:046C 存储单元的包含 50 个采样值的信息表。单元 0000:046C 含有一个由 PC 实时时钟维持的计数器。图 3-7 给出了这个表和用于顺序寻址表中各个单元的寄存器 BX。为了实现这个任务，首先用立即寻址的 MOV 指令将表的起始地址装入 BX 中，初始化表的起始地址以后，用寄存器间接寻址方式顺序存储这 50 个采样值。

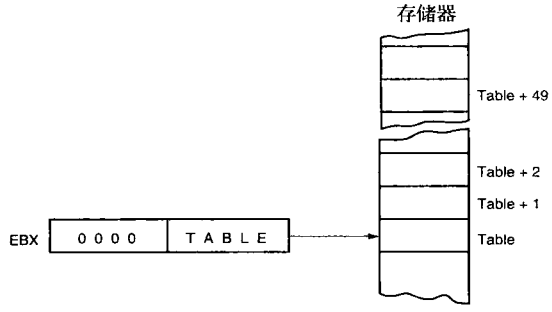


图 3-7 一个由 BX 间接寻址的包含 50 个字节的数组（TABLE）

例 3-7 给出的程序先将表的起始地址装入 BX 寄存器，再将 50 装入 CX 寄存器，以便初始化计数器。**OFFSET** 伪指令通知汇编程序把存储单元 TABLE 的偏移地址装入 BX，而不是装入 TABLE 的内容。例如 MOV BX，DATAS 指令，将存储单元 DATAS 的内容复制到 BX 中，而 MOV BX，OFFSET DATAS 指令将 DATAS 的偏移地址复制到 BX 中。当 OFFSET 伪指令用于 MOV 指令时，汇编程序先计算偏移地址，再用立即寻址的 MOV 指令将这个地址装入指定的 16 位寄存器中。

例 3-7

```
                        .MODEL SMALL            ;选择 SMALL 模型
0000                    .DATA                   ;指示数据段的开始

0000 0032 [         DATAS   DW 50 DUP(?)        ;建立 50 个字的数组
          0000
             ]
0000                    .CODE                   ;指示代码段的开始
                        .STARTUP                ;指示程序的开始
0017 B8 0000            MOV AX,0
001A 8E C0              MOV ES,AX               ;把段地址 0000 放入 ES
001C B8 0000 R          MOV BX,OFFSET DATAS     ;把 DATAS 的偏移地址复制到 BX 中
```

```
001F B9 0032              MOV CX,50            ;把 50 装入计数器 CX
0022                AGAIN:
0022 26:A1 046C           MOV AX,ES:[046CH]    ;得到时钟值
0026 89 07                MOV [BX],AX          ;把时钟值保存到 DATAS 中
0028 43                   INC BX               ;BX 加 1 指向表的下一个单元
0029 43                   INC BX
002A E2 F6                LOOP AGAIN           ;重复循环 50 次

                          .EXIT                ;返回到 DOS
                          END                  ;程序结束
```

当计数器和指针被初始化后，就重复执行循环，直到 CX = 0。程序用 MOV AX，ES:［046CH］指令从附加段存储单元 46CH 读出数据，并利用寄存器 BX 中的偏移地址，使用间接寻址方式将数据存入存储单元中。然后 BX 递增（BX 加 1）两次指向表的下一个字。最后 LOOP 指令重复循环 50 次。LOOP 指令使计数器 CX 递减（CX 减 1），如果 CX 不是 0，LOOP 指令转移到存储器 AGAIN 处。如果 CX 为 0，不执行转移，这个指令序列结束。本例将最近的 50 个时钟值复制到存储器数组 DATAS 中。这个程序通常在各个单元观察到同样的值，因为时钟计数器每秒只变化 18.2 次。为了观察程序和它的执行过程，可以使用 CodeView 程序。要使用 CodeView，键入 CV XXXX .EXE，XXXX .EXE 是要调试的程序名，或者在 Programmer's WorkBench 程序的 RUN 菜单下用 DEBUG 调用它。CodeView 只用于 .EXE 或 .COM 文件。几个实用的 CodeView 开关是：/50 用于每次显示 50 行，/S 用于高分辨率的视频显示。为了用 50 行的形式调试 TEST .COM 文件，应该在 DOS 提示符下键入 CV /50/S　TEST .COM。

3.1.5　基址加变址寻址

基址加变址寻址类似于间接寻址，因为它间接地访问存储器数据。在 8086 ～ 80286 中，这种寻址用一个基址寄存器（BP 或 BX）和一个变址寄存器（DI 或 SI）间接寻址存储器。通常基址寄存器保存存储器数组的起始位置地址，而变址寄存器保存数组元素的相对位置。每次 BP 寄存器寻址存储器数组时，由 BP 寄存器和堆栈段寄存器两者生成有效地址。

在 80386 及更高型号的微处理器中，这种寻址方式允许除了 ESP 以外的任意两个 32 位扩展寄存器组合使用。例如，MOV DL，［EAX + EBX］指令是用 EAX 作为基址加上 EBX 作为变址的例子。如果用 EBP 寄存器，则数据在堆栈段中而不在数据段中。

用基址加变址寻址定位数据

图 3-8 指出了微处理器在实模式下操作时，MOV DX，［BX + DI］指令怎样寻址数据。在这个例子中，BX = 1000H，DI = 0010H，DS = 0100H，因此存储器地址是 02010H。这条指令传送位于 02010 单元的字数据到 DX 寄存器。表 3-6 列出了一些使用基址加变址寻址的指令。注意 Intel 汇编程序要求这种寻址方式以［BX］［DI］形式出现，而不是［BX + DI］形式。因此 MOV DX，［BX + DI］指令就是 Intel ASM 汇编程序中的 MOV DX，［BX］［DI］。本书的所有例子使用第一种形式［BX + DI］，但是第二种形式［BX］［DI］也可以用于多种汇编程序中，包括 Microsoft 的 MASM。MOV DI，［BX + DI］指令也能汇编，但不会正确执行。

表 3-6　基址加变址寻址的示例

汇编语句	长　度	操　作
MOV CX，［BX + DI］	16 位	把由 BX + DI 寻址的数据段存储单元内的字内容装入 CX
MOV CH，［BP + SI］	8 位	把由 BP + SI 寻址的堆栈段存储单元内的字节内容装入 CH
MOV［BX + SI］，SP	16 位	把 SP 的内容存入由 BX + SI 寻址的数据段存储单元
MOV［BP + DI］，AH	8 位	把 AH 的内容存入由 BP + DI 寻址的堆栈段存储单元
MOV CL，［EDX + EDI］	8 位	把由 EDX + EDI 寻址的数据段存储单元内的字节内容装入 CL
MOV［EAX + EBX］，ECX	32 位	把 ECX 中的双字存入由 EAX + EBX 寻址的数据段存储单元
MOV［RSI + RBX］，RAX	64 位	把由 RSI + RBX 寻址的线性存储单元装入 RAX

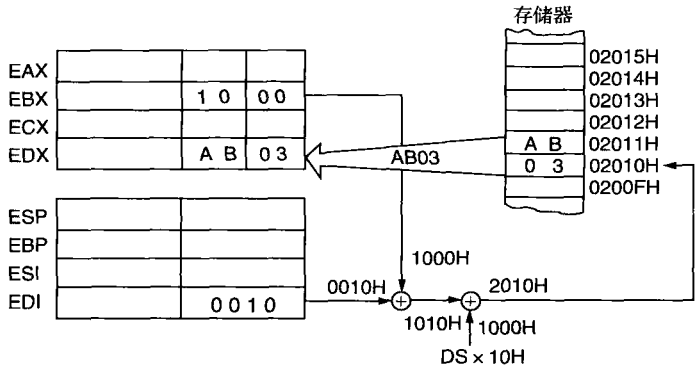


图 3-8 本例指出基址加变址寻址怎样用于 MOV DX,［BX + DI］指令

注：DS = 0100H，BX = 1000H，DI = 0010H，所以被访问的存储器地址是 02010H。

用基址加变址寻址定位数组数据

基址加变址寻址方式的主要用途是寻址存储器数组中的元素。假设在数据段存储器地址 ARRAY 处为一数组，现要存取数组中的元素。为此，将数组的起始地址装入寄存器 BX（基址），而把要存取的元素在数组中的序号数存入寄存器 DI（变址）。图 3-9 展示了用 BX 和 DI 存取数组元素的例子。例 3-8 中列出的短程序将数组中 10H 号元素装入数组单元 20H。注意，它是用装入 DI 寄存器中的数组元素序号寻址数组元素的。还要注意如何初始化 ARRAY 的内容，使 10H 号元素包含 29H。

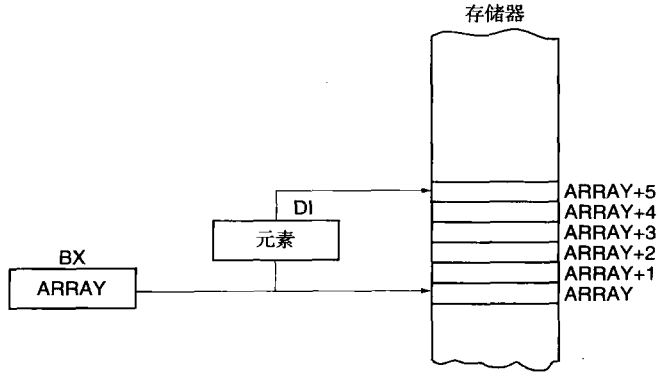


图 3-9 基址加变址寻址的例子，寻址 ARRAY（BX）中的元素（DI）

例 3-8

```
                        .MODEL SMALL          ;选择 SMALL 模型
0000                    .DATA                 ;指示数据段的开始
0000 0010 [     ARRAY DB 16 DUP(?)            ;建立 16 字节的数组 ARRAY
          00
             ]
0010 29                 DB 29H                ;将采样数据置入 10H 数据元素中
0011 001E [             DB 20 dup (?)
          00
             ]
0000                    .CODE                 ;指示代码段的开始
                        .STARTUP              ;指示程序的开始
0017 B8 0000 R          MOV BX,OFFSET ARRAY ;寻址 ARRAY
```

```
001A BF 0010          MOV DI,10H          ;寻址单元10H
001D 8A 01            MOV AL, [BX+DI]     ;得到元素10H
001F BF 0020          MOV DI,20H          ;寻址单元20H
0022 88 01            MOV [BX+DI],AL      ;保存到单元20H中

                      .EXIT               ;返回到DOS
                      END                 ;程序结束
```

3.1.6 寄存器相对寻址

寄存器相对寻址类似于基址加变址寻址和位移量寻址。在寄存器相对寻址中，用位移量加基址或变址寄存器（BP、BX、DI或SI）的内容寻址存储器段中的数据。图3-10指出了MOV AX，[BX+1000H]指令的操作。在此例中BX=0100H，DS=0200H，因此地址是DS×10H、BX及位移量1000H的和，即03100H。要记住的是，BX、DI或SI寻址数据段，而BP寻址堆栈段。在80386及更高型号的微处理器中，位移量可以是32位的数字，寄存器可以是除了ESP以外的任何32位寄存器。注意实模式中的段长为64KB。表3-7列出了几条用寄存器相对寻址的指令。

位移量可以是在[ ]符号内加到寄存器上的一个数，例如指令MOV AL，[DI+2]；也可以是从寄存器中减去的数，例如MOV AL，[SI-1]。位移量还可以是加在[ ]号前面的偏移地址，例如MOV AL，DATA[DI]。两种形式的位移量可能同时出现，例如MOV AL，DATA[DI+3]指令。所有情况下，两种形式的位移量都加到基址中，即加到方括号[ ]内的基址和变址寄存器中。在8086～80286微处理器中，位移量的值限制为16位的有符号数，其值的范围是+32767（7FFFH）～-32768（8000H）。在80386及更高型号的微处理器中，位移量可以是32位数，其值的范围是+2 147 483 647（7FFFFFFFH）～-2 147 483 648（80000000H）。

表3-7 寄存器相对寻址的示例

汇编语句	长度	操作
MOV AX，[DI+100H]	16位	把由DI+100H寻址的数据段存储单元中的字内容装入AX
MOV ARRAY[SI]，BL	8位	把BL中的字节存入由ARRAY+SI寻址的数据段存储单元
MOV LIST[SI+2]，CL	8位	把CL中的字节存入由LIST+SI+2之和寻址的数据段存储单元
MOV DI，SET_IT[BX]	16位	把由SET_IT+BX寻址的数据段存储单元的字内容装入DI
MOV DI，[EAX+10H]	16位	把由EAX+10H寻址的数据段存储单元的字内容装入DI
MOV ARRAY[EBX]，EAX	32位	把EAX的内容存入由ARRAY+EBX寻址的数据段存储单元中
MOV ARRAY[RBX]，AL	8位	把AL的内容存入由ARRAY+RBX寻址的存储单元中（64位）
MOV ARRAY[RCX]，EAX	32位	把EAX的内容存入由ARRAY+RCX寻址的存储单元中（64位）

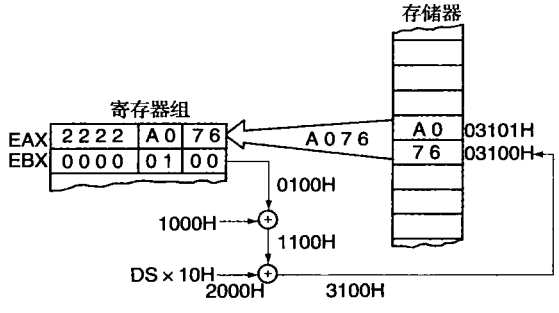


图3-10 当BX=0100H且DS=0200H时，MOV AX，[BX+1000H]指令的操作

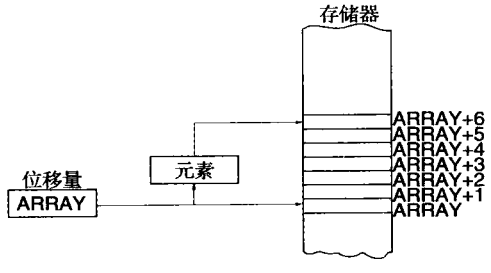


图3-11 用寄存器相对寻址访问ARRAY中的元素。位移量用于寻址ARRAY的起点，而DI用于存取一个元素

用寄存器相对寻址方式寻址数组数据

如同基址加变址寻址一样，可以用寄存器相对寻址方式寻址数组数据。在图3-11中，用与基址加

变址寻址相同的例子说明了寄存器相对寻址方式。这里指出了如何用位移量 ARRAY 加变址寄存器 DI 生成数组单元的指针。

例 3-9 指出了这种新的寻址方式怎样传送数组 10H 单元的内容到数组 20H 单元。注意这个例子与例 3-8 类似，它们的主要区别是，例 3-9 不用 BX 寄存器寻址 ARRAY 存储区域，而是将 ARRAY 作为位移量完成了相同的任务。

例 3-9

```
                        .MODEL SMALL            ;选择 SMALL 模型
0000                    .DATA                   ;指示数据段的开始
0000 0010 [      ARRAY  DB 16 dup(?)            ;建立数组 ARRAY
           00
             ]
0010 29                 DB 29                   ;在单元 10H 取样数据
0011 001E [             DB 30 dup (?)
           00
             ]
0000                    .CODE                   ;指示代码段的开始
                        .STARTUP                ;指示程序的开始
0017 BF 0010            MOV DI,10H              ;寻址单元 10H
001A 8A 85 0000 R       MOV AL,ARRAY[DI]        ;得到元素 10H
001E BF 0020            MOV DI,20H              ;寻址单元 20H
0021 88 85 0000 R       MOV ARRAY [DI],AL       ;保存到单元 20H 中
                        .EXIT                   ;返回到 DOS
                        END                     ;程序结束
```

3.1.7 相对基址加变址寻址

相对基址加变址寻址方式类似于基址加变址寻址方式，但是它用基址寄存器和变址寄存器加位移量形成存储器地址。这种寻址方式通常用来寻址存储器的二维数组中的数据。

用相对基址加变址寻址数据

相对基址加变址是很少使用的寻址方式，图 3-12 指出的是，如果微处理器执行 MOV AX，[BX + SI + 100H] 指令，怎样访问数据。位移量 100H 加上 BX 和 SI 形成数据段中的偏移地址。寄存器 BX = 0020H，SI = 0010H，而 DS = 1000H，因此这条指令的有效地址是这些寄存器加上位移量 100H 的和，即 10130H。这种寻址方式太复杂，在程序中很少使用。表 3-8 中显示了一些典型的用相对基址加变址寻址方式的指令。注意，在 80386 及更高型号的微处理器中，有效地址由两个 32 位寄存器和 32 位的位移量之和产生。

表 3-8 相对基址加变址寻址的示例

汇编语句	长 度	操 作
MOV DH, [BX + DI + 20H]	8 位	把由 BX、DI 及 20H 之和寻址的数据段存储单元的字节内容装入 DH
MOV AX, FILE [BX + DI]	16 位	把由 FILE、BX 及 DI 之和寻址的数据段存储单元的字节内容装入 AX
MOV LIST [BP + DI], CL	8 位	把 CL 存储到由 LIST、BP 及 DI 之和寻址的堆栈段存储单元中
MOV LIST [BP + SI + 4], DH	8 位	把 DH 存储到由 LIST、BP、SI 及 4 之和寻址的堆栈段存储单元中
MOV EAX, FILE [EBX + ECX + 2]	32 位	把由 FILE、EBX、ECX 及 2 之和寻址的数据段存储单元的双字内容装入 EAX

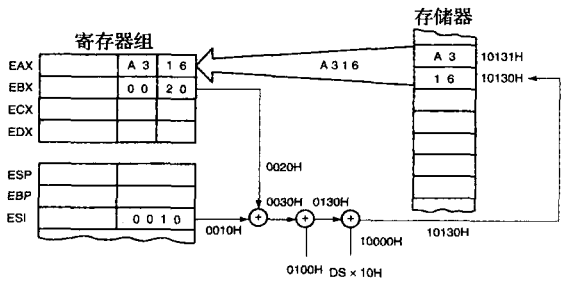


图 3-12 使用相对基址加变址寻址的例子 MOV AX，[BX + SI + 100H] 指令

注：DS = 1000H。

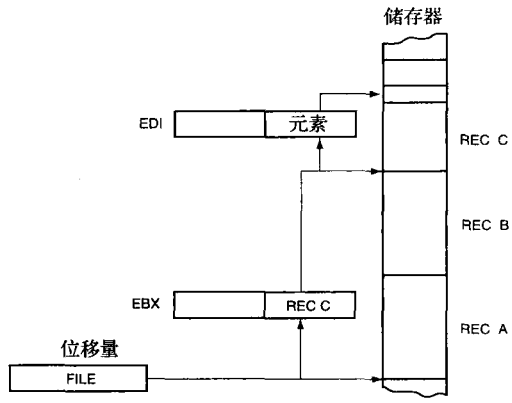


图 3-13 用相对基址加变址寻址方式存取包含有多个记录（REG）的文件

用相对基址加变址寻址访问数组

假定存储器中的文件包含多个记录，每个记录包含多个元素。这时可以用位移量寻址文件，基址寄存器寻址记录，而变址寄存器寻址记录中的元素。图 3-13 说明了这种非常复杂的寻址方式。

例 3-10 中的程序使用相对基址加变址寻址方式将记录 A 中的元素 0 复制到记录 C 中的元素 2。此例中的 FILE 包含 4 个记录，每个记录包含 10 个元素。注意程序中怎样用 THIS BYTE 伪指令定义标号 FILE 和 RECA 为同一存储单元。

例 3-10

```
                        .MODEL SMALL            ;选择 SMALL 模型
0000                    .DATA                   ;指示数据段的开始
0000  = 0000      FILE  EQU THIS BYTE           ;将 FILE 赋予 THIS BYTE
0000 000A [       RECA  DB 10 dup(?)            ;为记录 A 保留 10 个字节
           00
             ]
000A 000A [       RECB  DB 10 dup (?)           ;为记录 B 保留 10 个字节
           00
             ]
0014 000A [       RECC  DB 10 dup(?)            ;为记录 C 保留 10 个字节
           00
             ]
001E 000A [       RECD  DB 10 dup (?)           ;为记录 D 保留 10 个字节
           00
             ]
0000                    .CODE                   ;指示代码段的开始
                        .STARTUP                ;指示程序的开始
0017 BB 0000 R          MOV BX,OFFSET RECA      ;寻址记录 A
001A BF 0000            MOV DI,0                ;寻址元素 0
001D 8A 81 0000 R       MOV AL,FILE [BX + DI]   ;得到数据
0021 BB 0014 R          MOV BX,OFFSET RECC      ;寻址记录 C
0024 BF 0002            MOV DI,2                ;寻址元素 2
0027 88 81 0000 R       MOV FILE [BX + DI],AL   ;保存数据
                        .exit                   ;返回到 DOS
                        end                     ;程序结束
```

3.1.8 比例变址寻址

比例变址寻址是本章讨论的最后一种数据寻址方式，这种寻址方式是80386～Core2微处理器所特有的。比例变址寻址使用两个32位寄存器（基址寄存器和变址寄存器）访问存储器。第2个寄存器（变址寄存器）与比例因子相乘。比例因子可以是1×、2×、4×或8×。默认比例因子是1×，不要求包含在汇编语言指令中（例如，MOV AL，[EBX+ECX]）。比例因子2×用来寻址字存储器数组，比例因子4×用来寻址双字存储器数组，而比例因子8×用来寻址四字存储器数组。

例如，指令MOV AX，[EDI+2*ECX]，这条指令使用比例因子2×，ECX的内容先乘2，再与EDI寄存器相加，形成存储器地址。如果ECX的内容是0000 0000H，则寻址0号字存储器元素（Element 0）。如果ECX的内容是0000 0001H，则寻址1号字存储器元素（Element 1），以此类推。比例变址（ECX）乘以2是为了访问字存储器数组。表3-9列举了一些关于比例变址寻址的例子。事实上存在多种比例变址寄存器的组合。比例也可以用于使用单个间接寄存器寻址存储器的指令中。例如，MOV EAX，[4*EDI]指令，就是用一个寄存器间接寻址存储器的比例变址指令。在64位模式中，程序中可能出现如MOV RAX，[8*RDI]的指令。

表3-9 比例变址寻址的示例

汇编语句	长度	操作
MOV EAX，[EBX+4*ECX]	32位	把由EBX加4倍ECX之和寻址的数据段存储单元的双字内容装入EAX
MOV [EAX+2*EDI+100H]，CX	16位	把CX的内容存储到由EAX加2倍EDI再加100H寻址的数据段存储单元中
MOV AL，[EBP+2*EDI+2]	8位	把由EBP加2再加2倍EDI寻址的堆栈段存储单元的字节内容装入AL
MOV EAX，ARRAY[4*ECX]	32位	把由ARRAY加4倍ECX寻址的数据段存储单元的双字内容装入EAX

例3-11给出的指令序列用比例变址寻址方式访问LIST字数组数据。注意，此例用MOV EBX，OFFSET LIST指令将偏移地址LIST装入寄存器EBX。为了访问数组元素，在EBX寻址数组LIST之前，用比例因子2乘以那些字数组的元素号2、4、7（放在ECX中）。这个程序将单元2中的内容2存储到单元4和单元7。注意，.386伪指令选择80386微处理器，这个伪指令必须放在.MODEL语句后面，以便汇编程序在DOS下处理80386指令。如果用80486，则在.MODEL语句后放.486伪指令；如果用Pentium、Pentium Pro、Pentium Ⅱ、Pentium Ⅲ、Pentium 4或Core2，则在.MODEL语句后放.586伪指令。如果微处理器选择伪指令出现在.MODEL语句前，则微处理器按32位保护模式执行指令，必须在Windows中执行。

例3-11

```
                                .MODEL SMALL            ;选择SMALL模型
                                .386                    ;用80386微处理器
0000                            .DATA                   ;指示数据段的开始
0000 0000 0001 0002 LIST        DW 0,1,2,3,4            ;定义数组LIST
     0003 0004
000A 0005 0006 0007             DW 5,6,7,8,9
     0008 0009
0000                            .CODE                   ;指示代码段的开始
                                .STARTUP                ;指示程序的开始
0010 66! BB 00000000 R          MOV EBX,OFFSET LIST     ;寻址数组LIST
0016 66! B9 00000002            MOV ECX,2               ;寻址单元2
001C 67& 8B 04 4B               MOV AX,[EBX+2*ECX]      ;得到单元2
0020 66! B9 00000004            MOV ECX,4               ;寻址单元4
```

```
0026  67& 89  04  4B          MOV [EBX+2*ECX],AX  ;存入单元4
002A  66| B9  00000007        MOV ECX,7           ;寻址单元7
0030  67& 89  04  4B          MOV [EBX+2*ECX],AX  ;存入单元7
                              .exit               ;返回到 DOS
                              end                 ;程序结束
```

3.1.9 RIP 相对寻址

这种寻址方式在64位模式下采用64位指令指针寄存器来寻址平展内存模式下的线性位置。Visual C++可用的内嵌汇编程序无法使用这种寻址模式或其他任何64位寻址模式。Microsoft Visual C++目前也不支持64位汇编代码的开发。指令指针通常采用符号*来编址，如*34意味着在程序中有34字节的提前量。当Microsoft最终为了支持64位模式在Visual C++中放置内嵌汇编程序的时候，这将是RIP相对寻址最可能呈现的方式。

一个来源是Intel，它为64位代码生产具有内嵌汇编程序的编译器（http：//www. intel. com/cd/software/products/asmo-na/eng/compilers/cwin/279582. htm）

3.1.10 数据结构

数据结构用来规定信息怎样存储到存储器数组中，对于使用数组的应用程序是非常有用的。最好把数据结构想象为一个数据模板。汇编语言伪指令STRUC定义结构的开始，ENDS语句定义结构的结束。在例3-12中定义了典型的数据结构，并且使用了3次。注意结构的名字是与STRUC和ENDS语句一起出现的。除了未进行汇编外，该例表示的数据结构是很典型的。

例3-12

```
;定义 INFO 数据结构
;
INFO        STRUC
NAMES       DB  32  dup(?)    ;名字为32字节长
STREET      DB  32  dup(?)    ;街道地址为32字节长
CITY        DB  16  dup(?)    ;城市名为16字节长
STATE       DB   2  dup(?)    ;州名为2字节长
ZIP         DB   5  dup(?)    ;邮政编码为5字节长

INFO        ENDS

NAME1       INFO <'Bob Smith', '123  Main Street', 'Wanda ', 'OH', '44444'>
NAME2       INFO <'Steve Doe', '222  Mouse Lane', 'Miller', 'PA', '18100'>
NAME3       INFO <'Jim Dover', '303  Main Street', 'Orender', 'CA', '90000'>
```

例3-12中的数据结构定义了5个字段。第一个字段长32字节，存放姓名；第二个长32字节，存放街道地址；第三个长16字节，存放城市名；第四个长2字节，存放州名；第五个长5字节，存放邮政编码。定义了结构（INFO）以后，就可以按照说明填入姓名和地址，例中给出了三个使用INFO的实例。注意，使用数据结构定义数据时，所有的文字用撇号（'）括起来，并且全部的字段用符号< >括起来。

当寻址结构中的数据时，用结构名和字段名选择结构中的一个字段。例如，为了寻址NAME2中的STREET，用操作数NAME2. STREET，即结构名后面跟随一个分隔符（.），然后是字段名。同样地，可以用NAME3. CITY访问结构NAME3中的CITY。

例3-13给出的程序用来清除结构NAME1中的姓名、结构NAME2中的街道地址和结构NAME3中的邮政编码。这个程序中一些指令的功能和操作将在后续章节中定义，这些指令学完以后，可以再返回来参看这个例子。

例3-13

```
                ;清除数组 NAME1 中的姓名
0000  B9  0020      MOV CX,32
```

```
0003 B0 00          MOV AL,0
0005 BE 0000 R      MOV DI,OFFSET NAME1.NAMES
0008 F3/AA          REP STOSB
                    ;
                    ;清除数组 NAME2 中的街道
                    ;
000A B9 0020        MOV CX,32
000D B0 00          MOV AL,0
000F BE 0077 R      MOV DI,OFFSET NAME2.STREET
0012  F3/AA         REP STOSB
                    ;
                    ;清除数组 NAME3 中的邮政编码
                    ;
0014 B9 0005        MOV CX,5
0017 B0 00          MOV AL,0
0019 BE 0100 R      MOV DI,OFFSET NAME3.ZIP
001C F3/AA          REP STOSB
```

3.2 程序存储器寻址

用于 JMP（转移）和 CALL（调用）指令的程序存储器寻址方式有三种形式：直接寻址、相对寻址和间接寻址。本节介绍这三种寻址方式，用 JMP 指令解释其操作。

3.2.1 直接程序存储器寻址

许多早期微处理器中，所有转移和调用指令都使用直接程序存储器寻址。高级语言中也用直接程序存储器寻址，例如 BASIC 语言中的 GOTO 和 GOSUB 指令。虽然目前微处理器还在使用这种寻址方式，但已经远不如使用相对和间接程序存储器寻址那么频繁了。

直接程序存储器寻址指令的地址与操作码一同存储。例如，如果程序要转移到存储单元 10000H 处执行下一条指令，则地址 10000H 在存储器中被放在操作码的后面。图 3-14 表示了直接段间 JMP 指令，需要 4 个字节存放地址 10000H。这条 JMP 指令将 1000H 装入 CS，0000H 装入 IP，因此转移到存储单元 10000H 处执行下一条指令。**段间转移**（**intersegment jump**）可以转移到整个存储系统内的任何位置。直接转移通常称为远转移（far jump），因为它可以转移到任何存储单元执行下一条指令。在实模式中，远转移通过改变 CS 和 IP 两者的内容，访问存储器第一个 1MB 内的任何单元。在保护模式操作中，远转移访问描述符表里的新的代码段描述符，允许转移到 80386～Core2 微处理器的整个 4GB 地址范围内的任何存储单元。

操作码	偏移量(低)	偏移量(高)	段(低)	段(高)
EA	00	00	00	10

图 3-14 JMP［10000H］指令的 5 字节机器语言形式

在 Pentium 4 和 Core2 的 64 位模式下，jump 或者 call 指令可以跳转到系统的任意内存位置。CS 段仍然使用，但是不用于 jump 或者 call 指令的寻址。CS 寄存器包含一个指向描述符的指针，描述符描述了代码段的访问权限和优先级，但是不包含 jump 或 call 指令的地址。

另一种使用直接程序存储器寻址的指令是段间调用指令，即远 CALL 指令。通常，调用或跳转的位置引用称为标号的内存地址名，而不用实际的数字地址。CALL 或 JMP 指令中使用标号时，多数汇编程序将自动选择最合适的程序寻址方式。

3.2.2 相对程序存储器寻址

相对程序存储器寻址不能用于所有早期的微处理器中，但是可用于 Intel 系列中。术语**相对**（**relative**）意味着相对于指令指针（IP）。例如，如果 JMP 指令跳过后面两个存储器字节，则相对于指令指针的地址是 2，将 2 与指令指针相加，就得到程序下一条指令的地址。图 3-15 给出了相对 JMP 指令的

例子。注意，JMP 指令的格式是 1 字节操作码加 1 个或 2 个字节的位移量，位移量将与指令指针相加。1 个字节位移量用于**短**（**short**）转移；2 个字节位移量用于**近**（**near**）转移和调用。这两种类型都为**段内转移**（**intrasegment jump**），段内转移是转移到当前代码段内的任何位置。在 80386 及更高型号的微处理器中，位移量还可以是 32 位数，允许用相对寻址转移到 4GB 代码段内的任何位置。

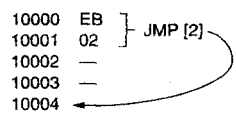


图 3-15 JMP［2］指令，该指令跳过其后的两个字节继续执行

相对 JMP 和 CALL 指令包含 8 位或 16 位带符号的位移量，允许向前或者向后访问存储器（在 80386 及更高型号的微处理器中，可以有 8 位或 32 位的位移量）。所有的汇编程序都能自动地用位移量计算距离，并且选择合适的 1、2 或 4 字节形式。8086 ～ 80286 微处理器中，如果距离太远，超出两字节的位移量，有些汇编程序就使用直接转移。8 位的位移量（短转移）具有相对于下条指令 +127 到 -128 字节的转移范围，而 16 位的位移量（近转移）具有 ±32KB 的范围。80386 及更高型号的微处理器中，32 位的位移量允许转移范围为 ±2GB，32 位的位移量只能用于保护模式。

3.2.3 间接程序存储器寻址

对于 CALL 和 JMP 指令，微处理器提供了几种间接程序存储器寻址形式。表 3-10 列出了一些可接受的间接程序转移指令，它们可以用任何 16 位寄存器（AX、BX、CX、DX、SP、BP、DI 或 SI），任何相对寄存器（［BP］、［BX］、［DI］或［SI］），或任何带有位移量的相对寄存器。80386 和更高型号的微处理器中，可以用扩展寄存器存放相对 JMP 或 CALL 的地址或间接地址。例如，JMP EAX 指令转移到由 EAX 寻址的位置。

表 3-10 间接程序存储器寻址的例子

汇编语句	操作
JMP AX	转移到当前代码段内由 AX 的内容寻址的位置
JMP CX	转移到当前代码段内由 CX 的内容寻址的位置
JMP NEAR PTR［BX］	转移到当前代码段位置上，地址存放在数据段首加 BX 寻址的单元中
JMP NEAR PTR［DI+2］	转移到当前代码段位置上，地址存放在数据段首加 DI+2 寻址的单元
JMP TABLE［BX］	转移到当前代码段内由 TABLE 加 BX 的内容寻址的那个位置
JMP ECX	转移到当前代码段内由 ECX 的内容寻址的单元
JMP RDI	转移到 RDI 寄存器包含的线性地址单元（64 位）

如果用 16 位寄存器存放 JMP 指令的地址，则是近转移。例如，如果 BX 寄存器含有 1000H，执行 JMP BX 指令以后，微处理器转到当前代码段内偏移地址 1000H 处。

如果用相对寄存器保存地址，则这种转移是间接转移。例如，JMP［BX］指令要转移到的地址在数据段内某个存储单元内，该存储单元的偏移地址在 BX 中。在该偏移地址单元内是 16 位数，作为段内转移的偏移地址。这种类型的转移称为间接-间接或双重间接转移。

图 3-16 给出了一个跳转表，它存储在从存储单元 TABEL 开始的区域内。这个跳转表由例 3-14 中的程序访问。在例 3-14 中，将 4 装入 BX 寄存器，因此当 4 与 JMP TABEL［BX］指令中的 TABLE 结合后，生成的有效地址指向 16 位宽的转移表中的第 2 项。

```
TABLE  DW  LOC0
       DW  LOC1
       DW  LOC2
       DW  LOC3
```

图 3-16 存储各种程序地址的跳转表。从跳转表中选择的确切地址取决于和转移指令存放在一起的变址值

例 3-14

```
;使用间接寻址形式的转移
;
```

```
0000  BB 0004          MOV BX,4            ;寻址 LOC2
0003  FF A7 23A1 R     JMP TABLE [BX]      ;转移到 LOC2
```

3.3 堆栈存储器寻址

堆栈在所有微处理器中都起着重要的作用，它用来暂时存放数据，为程序保存返回地址。堆栈存储器是 **LIFO（Last-in，first-out，后进先出）**存储区，这形象地说明了数据存入和从堆栈取出的方式。数据用 **PUSH 指令（PUSH instruction）**压入堆栈，用 **POP 指令（POP instruction）**弹出堆栈。CALL 指令用堆栈保存程序返回地址，而 RET（返回）指令从堆栈取出返回地址。

堆栈存储器用两个寄存器维护：**堆栈指针（Stack Pointer，SP 或 ESP）**和**堆栈段寄存器（Stack Segment，SS）**。如图 3-17a 所示，当字数据被压入堆栈时，高 8 位放入由 SP－1寻址的单元，低 8 位放入由 SP－2寻址的单元，然后 SP 中的值减 2，因此使下一个数据字存入下一个可用的堆栈存储器单元。SP/ESP 寄存器总是指向位于堆栈段内的存储区域。在实模式中，SP/ESP 寄存器加上 SS × 10H 形成堆栈存储器地址。以保护模式操作时，SS 寄存器保存一个用于访问一个描述符的选择子，通过描述符得到堆栈段的基地址。

如图 3-17b 所示，当数据从堆栈弹出时，低 8 位从由 SP 寻址的单元移出。高 8 位从由 SP＋1 寻址的单元移出，然后 SP 寄存器加 2。表 3-11 列出了一些可用于微处理器的 PUSH 和 POP 指令。注意，在 8086～80286 微处理器中，PUSH 和 POP 指令总是按字（而不是字节）数据存储和恢复的。80386 及更高型号的微处理器允许字或双字传送到堆栈或从堆栈移出。任何 16 位寄存器或段寄存器的数据都可以压入堆栈，而在 80386 及更高型号的微处理器中，任何 32 位扩展寄存器都可以压入堆栈。数据可以从堆栈弹出到任何 16 位寄存器，或除了 CS 以外的任何段寄存器。数据之所以不能从堆栈弹出到 CS，原因是这样只改变了下一条指令的部分地址。在 Pentium 4 或 Core2 的 64 位操作中，64 位寄存器可以弹出或压入堆栈，但是它们都是 8 个字节长度。

表 3-11 PUSH 和 POP 指令的例子

汇编语句	操作
POPF	从堆栈弹出一个字，放入标志寄存器
POPFD	从堆栈弹出双字，放入标志寄存器 EFLAG
PUSHF	将标志寄存器的内容复制到堆栈中
PUSHFD	将 EFLAG 的内容复制到堆栈中
PUSH AX	将 AX 的内容复制到堆栈中
POP BX	从堆栈弹出一个字，放入 BX 中
PUSH DS	将 DS 的内容复制到堆栈中
PUSH 1234H	将 1234H 压入到堆栈中
POP CS	非法指令
PUSH WORD PTR [BX]	将由 BX 寻址的数据段存储单元内的字复制到堆栈中
PUSHA	把寄存器 AX、CX、DX、BX、SP、BP、DI 和 SI 的内容复制到堆栈中
POPA	从堆栈弹出字数据，顺序放入 SI、DI、BP、SP、BX、DX、CX 和 AX 中
PUSHAD	把寄存器 EAX、ECX、EDX、EBX、ESP、EBP、EDI 和 ESI 的内容复制到堆栈中
POPAD	从堆栈弹出双字数据，顺序放入 ESI、EDI、EBP、ESP、EBX、EDX、ECX 和 EAX 中
POP EAX	从堆栈弹出双字数据，放入 EAX 中
POP RAX	从堆栈弹出四字数据，放入 RAC 中（64 位）
PUSH EDI	将 EDI 的内容复制到堆栈中
PUSH RSI	将 RSI 的内容复制到堆栈中（64 位）
PUSH QWORD PTR [RDX]	将 RDX 寻址的数据段存储单元内的四字复制到堆栈中

PUSHA 和 POPA 指令用于压入到堆栈或者从堆栈弹出除了段寄存器以外的所有寄存器。这些指令

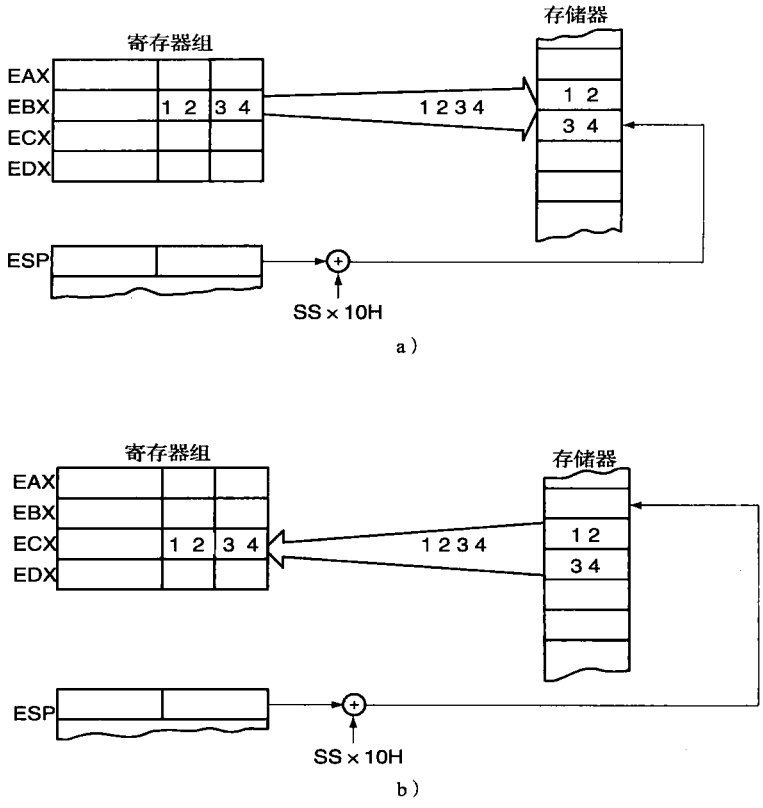


图 3-17 PUSH 和 POP 指令，本图给出了这两条指令执行后的状况

a) PUSH BX 将 BX 的内容放入堆栈 b) POP CX 从堆栈返回数据并且放入 CX 中

不适合于 8086/8088 微处理器。压入立即数到堆栈的指令也是 80286 ～ Core2 微处理器的新指令。注意，表 3-11 指出了 PUSHA 和 POPA 指令将寄存器入栈和出栈的顺序。80386 及更高型号的微处理器允许将扩展寄存器的内容压入或弹出堆栈。Pentium 4 和 Core2 的 64 位模式中不包含有 PUSHA 或 POPA 指令。

例 3-15 给出了将 AX、BX 和 CX 的内容压入到堆栈的短程序。第一个 POP 指令将原来从 CX 压入到堆栈的值取出来并且放入 AX，第二个 POP 指令弹出 BX 原来的值并且放入 CX，最后的 POP 指令弹出 AX 原来的值放入 BX。

例 3-15

```
                    .MODEL TINY         ;选择 TINY 模型
0000                .CODE               ;指示代码段的开始
                    .STARTUP            ;指示程序的开始
0100  B8  1000      MOV AX,1000H        ;装入测试数据
0103  BB  2000      MOV BX,2000H
0106  B9  3000      MOV CX,3000H

0109  50            PUSH AX             ;将 1000H 压入到堆栈
010A  53            PUSH BX             ;将 2000H 压入到堆栈
010B  51            PUSH CX             ;将 3000H 压入到堆栈

010C  58            POP AX              ;将 3000H 弹出到 AX
010D  59            POP CX              ;将 2000H 弹出到 CX
```

```
010E 5B          POP BX          ;将1000H弹出到BX
                 .exit           ;返回到DOS
                 end             ;程序结束
```

3.4 小结

1）数据寻址方式包括：寄存器寻址、立即寻址、直接寻址、寄存器间接寻址、基址加变址寻址、寄存器相对寻址和相对基址加变址寻址。在80386～Pentium 4微处理器中，增加了一种新的寻址方式，称为比例变址寻址。

2）程序存储器寻址方式包括：直接寻址、相对寻址和间接寻址。

3）表3-12列出了用于8086～80286的所有实模式的数据寻址方式。注意80386及更高型号的微处理器除了使用这些模式外，又增加了本章定义的许多其他方式。在保护模式中，段寄存器的功能是寻址包含存储器段基地址的一个描述符。

4）80386～Core2微处理器又增加了一些寻址方式，允许扩展寄存器EAX、EBX、ECX、EDX、EBP、EDI和ESI寻址存储器。这些寻址方式以表的形式列出来要占很多篇幅，一般而言，这些扩展寄存器的作用与表3-12中列出的那些寄存器是相同的。例如，MOV AL，TABLE［EBX＋2＊ECX＋10H］是用于80386～Core2微处理器的有效寻址方式。

5）除了寄存器为64位宽且包含线性地址之外，Pentium 4和Core2微处理器的64位模式采用与32位的Pentium 4或Core2相同的寻址模式。在64位模式中存在一个额外的名为RIP相对寻址的寻址模式，为与指令寄存器中地址相关的数据寻址。

表3-12 实模式数据寻址方式示例

汇编语句	地址的产生
MOV AL，BL	8位寄存器寻址
MOV AX，BX	16位寄存器寻址
MOV EAX，ECX	32位寄存器寻址
MOV DS，DX	段寄存器寻址
MOV AL，LIST	（DS×10H）＋LIST
MOV CH，DATA1	（DS×10H）＋DATA1
MOV ES，DATA2	（DS×10H）＋DATA2
MOV AL，12	十进制立即数12
MOV AL，［BP］	（SS×10H）＋BP
MOV AL，［BX］	（DS×10H）＋BX
MOV AL，［DI］	（DS×10H）＋DI
MOV AL，［SI］	（DS×10H）＋SI
MOV AL，［BP＋2］	（SS×10H）＋BP＋2
MOV AL，［BX－4］	（DS×10H）＋BX－4
MOV AL，［DI＋1000H］	（DS×10H）＋DI＋1000H
MOV AL，［SI＋300H］	（DS×10H）＋SI＋300H
MOV AL，LIST［BP］	（SS×10H）＋LIST＋BP
MOV AL，LIST［BX］	（DS×10H）＋LIST＋BX
MOV AL，LIST［DI］	（DS×10H）＋LIST＋DI
MOV AL，LIST［SI］	（DS×10H）＋LIST＋SI
MOV AL，LIST［BP＋2］	（SS×10H）＋LIST＋BP＋2
MOV AL，LIST［BX－6］	（DS×10H）＋LIST＋BX－6
MOV AL，LIST［DI＋100H］	（DS×10H）＋LIST＋DI＋100H
MOV AL，LIST［SI＋200H］	（DS×10H）＋LIST＋SI＋200H
MOV AL，［BP＋DI］	（SS×10H）＋BP＋DI
MOV AL，［BP＋SI］	（SS×10H）＋BP＋SI
MOV AL，［BX＋DI］	（DS×10H）＋BX＋DI
MOV AL，［BX＋SI］	（DS×10H）＋BX＋SI
MOV AL，［BP＋DI＋8］	（SS×10H）＋BP＋DI＋8

（续）

汇编语句	地址的产生
MOV AL, [BP + SI - 8]	(SS × 10H) + BP + SI - 8
MOV AL, [BX + DI + 10H]	(DS × 10H) + BX + DI + 10H
MOV AL, [BX + SI - 10H]	(DS × 10H) + BX + SI - 10H
MOV AL, LIST [BP + DI]	(SS × 10H) + LIST + BP + DI
MOV AL, LIST [BP + SI]	(SS × 10H) + LIST + BP + SI
MOV AL, LIST [BX + DI]	(DS × 10H) + LIST + BX + DI
MOV AL, LIST [BX + SI]	(DS × 10H) + LIST + BX + SI
MOV AL, LIST [BP + DI + 2]	(SS × 10H) + LIST + BP + DI + 2
MOV AL, LIST [BP + SI - 7]	(SS × 10H) + LIST + BP + SI - 7
MOV AL, LIST [BX + DI + 3]	(DS × 10H) + LIST + BX + DI + 3
MOV AL, LIST [BX + SI - 2]	(DS × 10H) + LIST + BX + SI - 2

6）MOV 指令将源操作数内容复制到目的操作数中。任何此类指令都不改变源操作数。

7）寄存器寻址可以指定任何一个 8 位寄存器（AH、AL、BH、BL、CH、CL、DH 或 DL）或任何一个 16 位寄存器（AX、BX、CX、DX、SP、BP、SI 或 DI）。在段寄存器与 16 位寄存器/存储单元之间传送数据时，或者在 PUSH 及 POP 指令中，也可以用段寄存器（CS、DS、ES 或 SS）寻址。在 80386 ～ Core2 微处理器中，扩展寄存器也可以用于寄存器寻址，分别是：EAX、EBX、ECX、EDX、ESP、EBP、EDI 和 ESI。80386 及更高型号的微处理器还可以用 FS 和 GS 段寄存器。64 位中，寄存器有 RAX、RBX、RCX、RDX、RSP、RBP、RDI、RSI 和 R8 ～ R15。

8）MOV 立即数指令将直接跟在操作码后面的字节或字送到寄存器或存储单元。立即寻址方式操作程序中的常数。在 80386 及更高型号的微处理器中，可以将双字立即数装入 32 位寄存器或存储单元。

9）汇编语言使用 . MODEL 语句定义文件的开始和文件使用的存储模型。如果是 TINY 模型，程序只有一个段（代码段）并且汇编为命令（. COM）程序。如果用 SMALL 模型，程序使用代码段和数据段并且汇编为执行（. EXE）程序。其他模型的规模和属性列于附录 A 中。

10）直接寻址以两种形式出现在微处理器中：直接寻址和位移量寻址。两种寻址方式是等同的，只是直接寻址用于在 EAX、AX 或 AL 与存储器之间传送数据，而位移量寻址用于在任何寄存器与存储器之间传送数据。直接寻址需要 3 个字节存储空间，而位移量寻址需要 4 个字节。在 80386 及更高型号的微处理器中，由于需要加上寄存器前缀，还有操作数的长度，因此这些指令可能还需要增加字节。

11）寄存器间接寻址允许通过基地址（BP 和 BX）或变址寄存器（DI 和 SI）指向的存储单元中的数据地址来访问数据。在 80386 及更高型号的微处理器中可以用扩展寄存器 EAX、EBX、ECX、EDX、EBP、EDI 和 ESI 寻址存储器数据。

12）基址加变址寻址通常寻址数组中的数据。这种方式的存储器地址由基址寄存器、变址寄存器和 10H 倍的段寄存器的内容相加构成。在 80386 及更高型号中，基址寄存器和变址寄存器可以是除了 EIP 和 ESP 以外的任何 32 位寄存器。

13）寄存器相对寻址用基址寄存器或者变址寄存器加位移量去访问存储器中的数据。

14）相对基址加变址寻址对于寻址二维存储器数组很有用。地址由基址寄存器、变址寄存器、相对位移量和 10H 倍的段寄存器的内容相加构成。

15）比例变址寻址只适用于 80386 ～ Core2。两个寄存器中的第 2 个寄存器（变址寄存器）乘以比例因子 2 ×、4 × 或 8 ×，以便寻址存储器数组中的字、双字或四字。MOV AX, [EBX + 2 * ECX] 和 MOV [4 * ECX], EDX 就是比例变址寻址指令的例子。

16）数据结构是存储一组数据的模板，其中数据用数组名和字段名来寻址。例如，数组 NUMBER 中的字段 TEN 用 NUMBER. TEN 寻址。

17）直接程序存储器寻址允许 JMP 和 CALL 指令调用存储系统中的任何单元。在这种寻址方式中，偏移地址和段地址存放在指令中。

18）相对程序存储器寻址允许 JMP 和 CALL 指令向前或向后转移到当前代码段内 ±32KB 范围的位置。在 80386 及更高型号的微处理器中，32 位的位移量允许转移到代码段内 ±2GB 位移量的任何位置。32 位的位移量只能用于保护模式。

19）间接程序存储器寻址允许 JMP 和 CALL 指令通过寄存器或存储单元间接寻址其他区域的程序或子程序。

20）PUSH 和 POP 指令在堆栈与寄存器或堆栈与存储单元之间传送字数据。为了把立即数放入堆栈，用 PUSH 立即指令。PUSHA 和 POPA 指令在堆栈与寄存器 AX、CX、DX、BX、BP、SP、SI 和 DI 之间传送数据。在 80386 及更高型号的

微处理器中，扩展寄存器及扩展的标志也可以与堆栈之间进行传送。如 PUSHFD 指令存储 EFLAGS，而 PUSHF 指令存储 FLAGS。64 位中不包含有 PUSHA 和 POPA 指令。

3.5 习题

1. 下面的 MOV 指令完成什么操作？
   (a) MOV AX, BX
   (b) MOV BX, AX
   (c) MOV BL, CH
   (d) MOV ESP, EBP
   (e) MOV AX, CS
2. 列出寄存器寻址使用的 8 位寄存器。
3. 列出寄存器寻址使用的 16 位寄存器。
4. 列出 80386 ~ Core2 微处理器寄存器寻址使用的 32 位寄存器。
5. 列出 64 位的 Pentium 4 和 Core2 微处理器中使用的 64 位寄存器。
6. 列出由 MOV、PUSH、POP 寄存器寻址使用的 16 位段寄存器。
7. 指令 MOV BL, CX 存在什么错误？
8. 指令 MOV DS, SS 存在什么错误？
9. 为下面的每个任务选择指令。
   (a) 把 EBX 复制到 EDX
   (b) 把 BL 复制到 CL
   (c) 把 SI 复制到 BX
   (d) 把 DS 复制到 AX
   (e) 把 AL 复制到 AH
10. 为下面的每个任务选择指令。
    (a) 将 12H 传送到 AL 中
    (b) 将 123AH 传送到 AX 中
    (c) 将 0CDH 传送到 CL 中
    (d) 将 1000H 传送到 SI 中
    (e) 将 1200A2H 传送到 EBX 中
11. 有时用哪些特殊符号表示立即数？
12. .MODEL TINY 语句的作用是什么？
13. 什么样的汇编语言伪指令指明 CODE 段的开始？
14. 什么是标号？
15. MOV 指令放在语句的什么字段？
16. 标号可以由哪些字符开始？
17. .EXIT 伪指令的作用是什么？
18. .MODEL TINY 语句可以使程序被汇编成执行程序(.EXE)吗？
19. 在 SMALL 存储模型中，.STARTUP 伪指令完成什么任务？
20. 什么是位移量？怎样确定 MOV DS:[2000H], AL 指令中的存储器地址？
21. 符号 [ ] 指示什么？
22. 假定按实模式操作，DS = 0200H，BX = 0300H 和 DI = 400H. 确定下面每条指令寻址的存储器地址。
    (a) MOV AL, [1234H]
    (b) MOV EAX, [BX]
    (c) MOV [DI], AL
23. 指令 MOV [BX], [DI] 的错误是什么？
24. 选择一条需要 BYTE PTR 的指令。
25. 选择一条需要 WORD PTR 的指令。
26. 选择一条需要 DWORD PTR 的指令。
27. 选择一条需要 QWORD PTR 的指令。
28. 说明 MOV BX, DATA 和 MOV BX, OFFSET DATA 指令之间的区别。
29. 给定 DS = 1000H，SS = 2000H，BP = 1000H，DI = 0100H。假定按实模式操作，确定下面每条指令寻址的存储器地址。
    (a) MOV AL, [BP + DI]
    (b) MOV CX, [DI]
    (c) MOV EDX, [BP]
30. MOV AL, [BX] [SI] 指令中有错误吗？如果有，请说明它的错误是什么？
31. 给定 DS = 1200H，BX = 0100H 和 SI = 0250H。假定按实模式操作，确定下面每条指令寻址的存储器地址。
    (a) MOV [100H], DL
    (b) MOV [SI + 100H], EAX
    (c) MOV DL, [BX + 100H]
32. 给定 DS = 1100H，BX = 0200H，LIST = 0250H 和 SI = 0500H。假定按实模式操作，确定下面每条指令寻址的存储器地址。
    (a) MOV LIST [SI], EDX
    (b) MOV CL, LIST [BX + SI]
    (c) MOV CH, [BX + SI]
33. 给定 DS = 1300H，SS = 1400H，BP = 1500H 和 SI = 0100H。假定按实模式操作，确定下面每条指令寻址的存储器地址。
    (a) MOV EAX, [BP + 200H]
    (b) MOV AL, [BP + SI - 200H]
    (c) MOV AL, [SI - 0100H]
34. 哪些基址寄存器可以寻址堆栈段的数据？
35. 给定 EAX = 00001000H，EBX = 00002000H 和 DS = 0010H。假定工作在实模式，确定下面每条指令所访问的地址。
    (a) MOV ECX, [EAX + EBX]
    (b) MOV [EAX + 2 * EBX], CL
    (c) MOV DH, [EBX + 4 * EAX + 1000H]
36. 给出有 5 个字段的字数据结构，字段名是 F1、F2、F3、F4 和 F5，结构名是 FIELDS。
37. 在程序中怎样寻址习题 34 中数据结构的 F3 字段？
38. 三种程序存储器寻址方式是什么？

39. 存放远直接转移指令要用多少存储器字节？每个字节存储什么？
40. 段间转移和段内转移之间的区别是什么？
41. 如果近转移指令使用16位有符号的位移量，说明怎样转移到当前代码段内的某一存储单元？
42. 80386及更高型号的微处理器用_____位的位移量转移到4GB代码段内的任何位置。
43. 什么是远转移？
44. 如果JMP指令存储在当前代码段内的100H地址，它要转移到当前代码段内的200H地址，不能用_____转移。
45. 如果JMP THERE指令存储在存储器地址10000H处，并且THERE地址是下面的一些值，指出JMP指令汇编为哪类（短、近或远）转移？
    (a) 10020H
    (b) 11000H
    (c) 0FFFEH
    (d) 30000H
46. 构造一条JMP指令，转移到由BX寄存器指示地址。
47. 选择JMP指令转移到某个地址，该地址在存储器TABLE单元中。假定是近JMP指令。
48. 用PUSH AX指令可以把多少个字节存放到堆栈中？
49. 解释PUSH [DI] 指令是怎样工作的？
50. PUSHA指令将哪些寄存器放入堆栈中？按什么样的顺序存放？
51. PUSHAD指令完成什么工作？
52. 在Pentium 4微处理器中，哪种指令把EFLAGS存放到堆栈中？
53. PUSHA指令在64位的Pentium 4或Core2中可用吗？

第4章　数据传送指令

引言

本章集中介绍数据传送指令。数据传送指令包括：MOV、MOVSX、MOVZX、PUSH、POP、BSWAP、XCHG、XLAT、IN、OUT、LEA、LDS、LES、LFS、LGS、LSS、LAHF、SAHF，另外还包括串指令 MOVS、LODS、STOS、INS 和 OUTS。最后是在 Pentium Pro ～ Pentium 4 上执行的数据传送指令 CMOV（条件传送）指令。因为数据传送指令在程序中用得最普遍，并且最容易理解，所以首先介绍数据传送指令。

因为机器语言指令太复杂，手工编写太麻烦，所以微处理器需要由汇编程序生成机器语言。本章叙述汇编语言语法和它的一些伪指令。本书假定用户在 IBM PC 或兼容机上开发软件。建议使用 Microsoft 公司的 MACRO 汇编程序（MASAM）作为开发工具，但是 Intel 汇编程序（ASM）、Borland Turbo 汇编程序（TASM）或类似的汇编程序软件也可以作为开发工具。最新版本的 TASM 完全模仿 MASM 程序。本书给出的程序都是使用 Microsoft MASM 开发的。然而大多数程序用其他汇编程序汇编时也可以不必修改。附录 A 解释了 Microsoft 公司的汇编程序，并且提供了连接程序的详细说明。更新的选择是 Visual C++ 编译器及其内嵌汇编程序也可以用作开放系统，两者都会在本书详细说明。

目的

读者学习完本章后将能够：

1）解释适当寻址方式下每条数据传送指令的操作。
2）说明汇编语言伪操作和关键字 ALIGN、ASSUME、DB、DD、DW、END、ENDS、ENDP、EQU、.MODEL、OFFSET、ORG、PROC、PTR、SEGMENT、USE16、USE32 和 USES 的功能和用法。
3）选择合适的汇编语言指令，完成指定的数据传送任务。
4）确定十六进制机器语言指令中的操作码、源、目的和寻址方式。
5）用汇编语言程序设置数据段、堆栈段和代码段。
6）指出怎样用 PROC 和 ENDP 建立过程。
7）解释 MASM 汇编程序存储器模型与完整段定义之间的区别。
8）用 Visual C++ 内嵌汇编程序完成数据传送任务。

4.1　MOV 回顾

在第 3 章中介绍了 MOV 指令，说明了 8086 ～ Core2 的各种寻址方式。本章用 MOV 指令介绍了在各种寻址方式下的机器语言指令。介绍机器码是因为有时必须解释由汇编程序或 Visual C++ 内嵌汇编程序产生的机器语言程序。读懂机器自身的语言（**机器语言**），就可以在机器语言这一级上进行调试和修改。有时要用 DOS 的 DEBUG 程序，也可能在 Windows 下的 Visual C++ 用机器语言修补程序，因此需要具备一些机器语言知识。附录 B 说明了机器语言和汇编语言指令之间的转换。

4.1.1　机器语言

机器语言是一种作为指令由微处理器理解和使用的二进制代码，用它来控制微处理器自身的运行。8086 ～ Core2 的机器语言指令长度可以从 1 个字节到 13 个字节。尽管机器语言好像很复杂，但这些微处理器的机器语言却也很规则。共有 100 000 多种变化形式的机器语言指令。这意味着不能列一个完整的指令表来包涵这么多变形。因此，在机器语言指令中，某些二进制位是已给定的，其余的二进制位则由每条指令的变化形式确定。

8086 ～80286 的指令是 16 位指令模式，图 4-1a 说明了它们的组成格式。16 位指令模式与 80386 及

更高型号微处理器工作在16位指令模式时是兼容的，但这些微处理器必须如图4-1b指出的那样带有指令前缀。80386及更高型号微处理器当按实模式操作时，假定所有指令都是16位指令模式。在保护模式中，描述符的高端字节包含选择16位模式或32位模式指令的D位。目前只有Windows 95～Windows XP和Linux按32位指令模式操作。图4-1b给出了32位模式指令的格式，它们通过在16位指令模式前加前缀而构成，这些前缀将在本章后面解释。

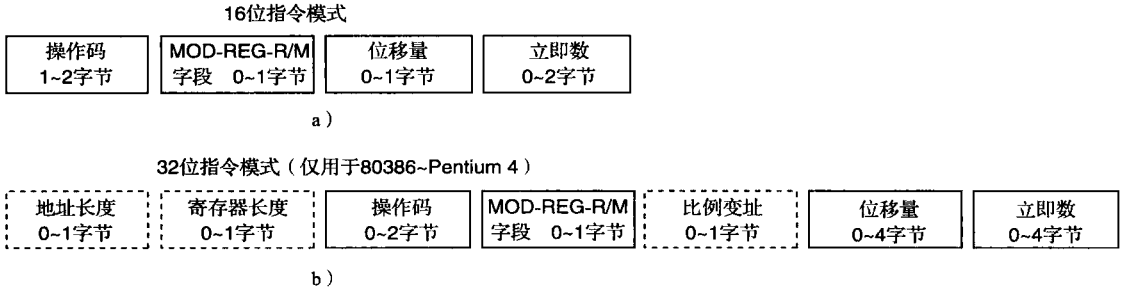


图4-1 8086～Core2指令的格式

a）16位格式 b）32位格式

32位指令格式的头两个字节，因为不常出现，所以称为**超越前缀**（**override prefix**）。第一个字节用来修改指令操作数地址的长度，第2个字节修改寄存器的长度。如果80386～Pentium 4按16位指令模式的机制操作（实模式或保护模式），而使用32位寄存器，则指令的前面出现**寄存器长度前缀**（**register-size prefix**）66H。如果微处理器按32位指令模式操作（只在保护模式），而且使用32位寄存器，则不存在寄存器长度前缀。如果在32位指令模式中出现16位寄存器，则要用寄存器长度前缀选择16位寄存器。**地址长度前缀**（**address size prefix**）67H的用法类似，将在本章后面解释。在带有前缀的指令中，前缀把寄存器及操作数地址的长度从16位转换到32位，或是从32位转换到16位。注意，16位指令模式用8位及16位寄存器和寻址方式；而32位指令模式使用8位及32位寄存器和寻址方式，这是默认的用法。前缀可超越这些默认值，因此32位寄存器可以用于16位模式，而16位寄存器可以用于32位模式。**操作模式**（**mode of operation**）的选择（16位或32位）要符合现有的应用程序。如果应用程序中多是8位和32位数据，则要选择32位模式；同样，如果大多用8位或16位数据，则要选择16位模式。通常选择模式是操作系统的工作（记住，DOS只能按照16位模式操作，而Windows可以工作于两类模式）。

操作码

操作码（**opcode**）选择微处理器执行的操作（加、减、传送等）。多数机器语言指令的操作码长为1或2个字节。图4-2说明了多数（但并不是所有）机器语言指令第一个操作码字节的一般格式。第一个字节的前6位是操作码，其余的两位指示数据流的**方向**（**D**）和数据是字节还是字（W）。请不要把数据流的方向（D）与指令模式位（16/32）或用于串操作指令的方向标志位混淆。在80386及更高号微处理器中，当W=1时指示字或者双字。指令模式和寄存器长度前缀（66H）确定W表示字还是双字。

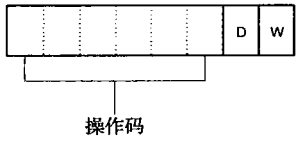


图4-2 多数机器语言指令的字节1，显示D位和W位的位置。

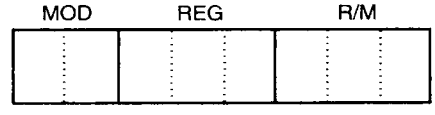


图4-3 多数机器语言指令的字节2，显示MOD、REG和R/M字段的位置

如果方向位D=1，数据从位于指令第二字节的R/M字段流向寄存器（REG）字段。如果操作码中D=0，数据从REG字段流向R/M字段。如果W=1，数据的长度是字或是双字，如果W=0，数据

的长度是字节。W 位出现在大多数指令中，而 D 位只出现在 MOV 或其他一些指令中。参考图 4-3 关于多数指令第二操作码字节的二进制位的划分。图中给出了 MOD（模式）、REG（寄存器）和 R/M（寄存器/存储器）字段的位置。

MOD 字段

MOD 字段规定指令的寻址方式（MOD）。MOD 字段选择寻址类型及所选的类型是否有位移量。表 4-1 列出了在没有操作数地址长度超越前缀（67H）时，16 位指令模式下 MOD 字段各种可能的取值。如果 MOD 字段的内容是 11，它选择寄存器寻址方式。寄存器寻址用 R/M 字段指定一个寄存器而不是存储单元。如果 MOD 字段的内容是 00、01 或 10，R/M 字段选择数据存储器寻址方式之一。当 MOD 字段选择了数据存储器寻址方式时，00 表示寻址方式没有位移量，01 表示包含 8 位有符号扩展的位移量，10 表示包含 16 位的位移量。MOV AL，[DI] 指令是没有位移量的例子。MOV AL，[DI + 2] 指令用 8 位的位移量（+2）。MOV AL，[DI + 1000H] 指令用 16 位的位移量（+1000H）。

表 4-1 16 位指令模式中的 MOD 字段

MOD	功 能
00	没有位移量
01	8 位符号扩展的位移量
10	16 位有符号的位移量
11	R/M 是寄存器

当微处理器执行指令时，将所有 8 位的位移量**符号扩展**（**sign-extended**）成 16 位的位移量。如果 8 位的位移量是 00H ～ 7FH（正的），在加到偏移地址之前扩展成 0000H ～ 007FH。如果 8 位的位移量是 80H ～ FFH（负的），扩展成 FF80H ～ FFFFH。符号扩展的数字是把它的符号位复制到后序的高字节中，使得高字节成为 00H 或 FFH。注意，有些汇编程序不使用 8 位位移量，并且都换成 16 位位移量。

在 80386 ～ Core2 微处理器中，对于 16 位指令模式，MOD 字段与表 4-1 给出的一样；如果指令模式是 32 位的，则 MOD 字段如表 4-2 所示。MOD 字段的含义由地址长度超越前缀选择，或由微处理器的操作模式确定。在 80386 ～ Core2 微处理器中，这种 MOD 字段含义的变化，允许指令支持许多附加的寻址方式。主要的区别是，当 MOD 字段是 10 时，16 位的位移量变成 32 位的位移量，允许访问保护模式的存储器单元（4G 字节）。80386 及更高型号微处理器工作在 32 位指令模式下，当不用地址长度超越前缀时，只允许用 8 位或 32 位的位移量。注意，如果选择了 8 位的位移量，微处理器符号扩展为 32 位的位移量。

表 4-2 32 位指令模式 MOD 字段（只限于 80386 ～ Core2）

MOD	功 能
00	没有位移量
01	8 位符号扩展的位移量
10	32 位有符号的位移量
11	R/M 是寄存器

寄存器分配

表 4-3 列出了 REG 字段和 R/M 字段（当 MOD = 11 时）寄存器的分配。这个表包含 3 种寄存器分配表：一种用于 W = 0（字节）时，其他两种用于 W = 1（字或双字）时。注意双字寄存器只能用于 80386 ～ Core2。

表 4-3 REG 和 R/M 的分配（当 MOD = 11 时）

代 码	W = 0（字节）	W = 1（字）	W = 1（双字）
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

假定在机器语言程序中有一个 2 字节指令 8BECH。由于第一个字节既不是 67H（操作数地址长度超越前缀）也不是 66H（寄存器长度超越前缀），因此第一个字节是操作码。假定微处理器是按

16 位指令模式操作，转换成二进制的这条指令按字节 1 和字节 2 的指令格式存放，见图 4-4，操作码是 1000 1011。如果参考附录 B 列出的机器语言指令，会发现这是 MOV 指令的操作码。还要注意 D 位和 W 位两者都是逻辑 1，这意味着要将一个字传送到 REG 字段指定的目的寄存器。REG 字段是 101，代表寄存器 BP，因此 MOV 指令传送数据到寄存器 BP。由于 MOD 字段是 11，R/M 字段也指示寄存器。这里 R/M = 100（SP），因此，这条指令是将数据从 SP 传送到 BP，写成符号形式是 MOV BP，SP 指令。



操作码 = MOV
D = 传送到寄存器（REG）
W = 字
MOD = R/M 是个寄存器
REG = BP　R/M = SP

图 4-4　按照图 4-2 和图 4-3 的格式将 8BEC 指令放入字节 1 和字节 2。指令的符号形式是 MOV BP，SP

假定 80386 及更高型号微处理器以 16 位指令模式操作时，出现 668BE8H 指令。第一字节的 66H 是寄存器长度超越前缀，为了在 16 位指令模式下选择 32 位寄存器。指令的剩余部分指示指令的操作码是 MOV，其源操作数是 EAX，而目的操作数是 EBP。这条指令是 MOV EBP，EAX。在 80386 及更高型号微处理器中如果按 32 位指令模式操作，由于寄存器长度超越前缀选择 16 位寄存器，同样这条指令就变成了 MOV BP，AX。幸亏汇编程序能保持对寄存器及地址长度前缀和操作方式的跟踪。回想一下，如果将 .386 开关放在 .MODE 语句的前面，则选择 32 位模式；如果放在 .MODE 语句后面，则选择 16 位模式。所有 Visual C++ 中内嵌汇编写的程序总是 32 位模式。

R/M 存储器寻址

如果 MOD 字段的内容是 00、01 或 10，则 R/M 按新的意义理解。表 4-4 列出了当 MOD 是 00，01 或 10 时，16 位指令模式的存储器寻址方式。

表 4-4　16 位的 R/M 存储器寻址方式

R/M 代码	寻 址 方 式
000	DS:［BX + SI］
001	DS:［BX + DI］
010	SS:［BP + SI］
011	SS:［BP + DI］
100	DS:［SI］
101	DS:［DI］
110	SS:［BP］①
111	DS:［BX］

① 见下一节：特殊寻址方式。

表 4-4 中给出了在第 3 章中出现的所有 16 位寻址方式。第 3 章中讨论的位移量是用 MOD 字段定义的。如果 MOD = 00 并且 R/M = 101，寻址方式是［DI］。如果 MOD = 01 或 10，对于 16 位指令模式寻址方式是［DI + 33H］或 LIST［DI + 22H］。这个例子使用了 LIST，33H 和 22H 作为位移量。

图 4-5 说明了 16 位模式指令 MOV DL，［DI］的机器语言形式，即指令 8A15H。这条指令长度为两个字节，操作码是 100010，D = 1（从 R/M 传送到 REG），W = 0（字节），MOD = 00（没有位移量），REG = 010（DL）和 R/M = 101（［DI］）。如果指令变成 MOV DL，［DI + 1］，MOD 字段变成 01，构成 8 位的位移量，但是指令的前两个字节保持相同。现在指令变成了 8A5501H，



操作码 = MOV
D = 传送到寄存器（REG）
W = 字节
MOD = 没有位移量
REG = DL
R/M = DS:［DI］

图 4-5　指令 MOV DL，［DI］的机器语言指令格式

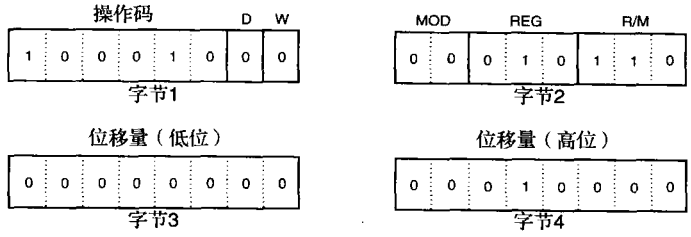
而不是8A15H。注意，8位的位移量加到前两个字节后面，因此构成了3字节指令而不是2字节指令。如果指令再变成MOV DL，［DI＋1000H］，则机器语言形式变成8A750010H。这里16位的位移量1000H（编码是0010H）加到操作码后面。

特殊寻址方式

一种特殊寻址方式没有在表4-2、表4-3或表4-4中出现。这种寻址方式，在16位指令模式下，只用位移量寻址存储器的数据。例如MOV［1000H］，DL和MOV NUMB，DL指令。第一条指令传送寄存器DL的内容到数据段中的存储单元1000H。第二条指令将寄存器DL的内容传送到数据段标号为NUMB的存储单元。

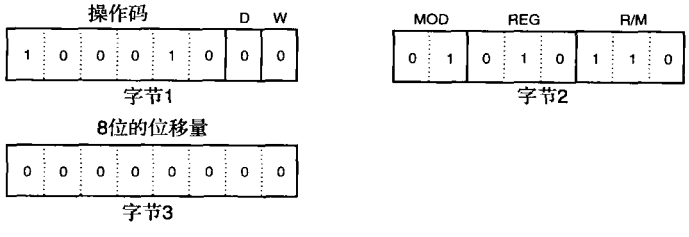
当指令只有一个位移量时，MOD字段总是00，而R/M字段总是110。如前面几个表格所指出的，这种组合表示指令用［BP］寻址方式，没有位移量。实际上机器语言中不可以用没有位移量的［BP］寻址方式。每当指令中出现BP寻址方式时，汇编程序就使用一个8位位移量（MOD＝01）00H。这意味着即使指令用了［BP］，也将［BP］寻址方式汇编成［BP＋0］。在32位模式中也可以用同样的特殊寻址方式。

图4-6给出了MOV［1000H］，DL指令译成机器语言的二进制代码格式。如果不了解特殊寻址方式，孤立地翻译这条指令为机器码，就可能错误地翻译成MOV［BP］，DL指令。图4-7指出了MOV［BP］，DL指令的实际形式。注意，是带有位移量00H的3字节指令。



操作码＝MOV
D＝从寄存器（REG）传送出
W＝字节
MOD＝因为R/M是［BP］（特殊寻址方式）
REG＝DL
R/M＝DS：［BP］
位移量＝1000H

图4-6 使用特殊寻址方式的指令MOV［1000H］，DL



操作码＝MOV
D＝从寄存器（REG）传送出
W＝字节
MOD＝因R/M是［BP］（特殊寻址方式）
REG＝DL
R/M＝DS：［BP］
位移量＝OOH

图4-7 转换成二进制机器语言的MOV［BP］，DL指令

32 位寻址方式

80386 及更高型号微处理器的 32 位寻址方式可通过 32 位指令模式或者用带地址长度前缀 67H 的 16 位指令模式运行机制获得。表 4-5 指出了用于指定 32 位寻址方式的 R/M 编码。注意，当 R/M = 100 时，指令中出现称为**比例变址字节**（**scaled-index byte**）的附加字节。该字节指示表 4-5 中未出现的附加的比例变址寻址方式。当指令中的两个寄存器的内容相加形成指定的存储器地址时，要使用比例变址字节。因为比例变址字节是加到指令中的，操作码占用 7 位，比例变址字节占用 8 位。这意味着比例变址有 2^{15}（32K）种可能的组合。仅在 80386 ～ Core2 微处理器中 MOV 指令就有 32000 多种不同的变化形式。

表 4-5　由 R/M 选择的 32 位寻址方式

R/M 代码	功　能
000	DS:[EAX]
001	DS:[ECX]
010	DS:[EDX]
011	DS:[EBX]
100	使用比例变址字节
101	SS:[EBP]①
110	DS:[ESI]
111	DS:[ESI]

① 参见本书中“特殊寻址方式”一节。

图 4-8 给出的是，80386 和更高型号微处理器使用 32 位地址并且指令的 R/M 字段值为 100 时，选择的比例变址字节的格式。最左面两位选择比例因子（乘数）是 1X、2X、4X 或 8X。注意，比例因子 1X 意味着指令中包含了两个 32 位的间接寻址寄存器。变址和基址字段均包含一个寄存器号，如表 4-3 中定义的 32 位寄存器。

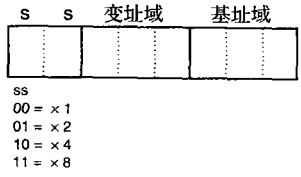
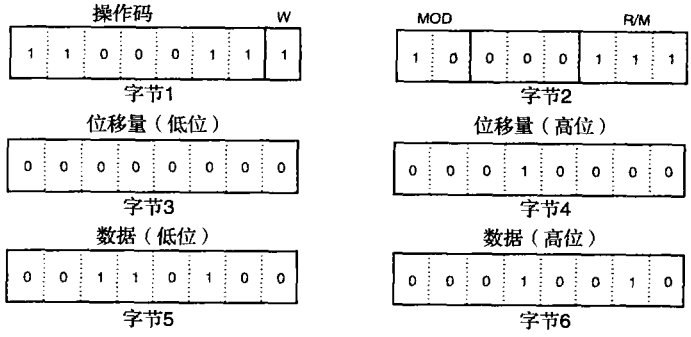


图 4-8　比例变址字节

指令 MOV EAX,[EBX +4 * ECX] 的编码是 67668B048BH。这条指令出现了地址长度（67H）和寄存器长度（66H）两个超越前缀。这意味着 80386 和更高型号微处理器按 16 位指令模式操作时，指令编码是 6766 8B048BH。如果处理器按 32 位指令模式操作，则不使用这两个前缀，指令编码就变成了 8B048BH。前缀的使用取决于微处理器的操作模式。比例变址也可以用单个寄存器乘以比例因子。例如，MOV AL,[2 * ECX] 指令将数据段的一存储单元的内容复制到 AL 中，该存储单元的位移量是 2 乘 ECX 的内容。

立即指令

我们以 MOV WORD PTR [BX + 1000H]，1234H 指令作为例，研究使用 16 位立即数寻址的指令。这个例子将 1234H 传送到用 1000H，BX 及 DS × 10H 之和寻址的存储单元中。6 字节的指令用两个字节作为操作码、W、MOD 和 R/M 字段。6 个字节中的另外两个字节是有效数据 1234H，还有两个字节是位移量 1000H。图 4-9 给出了这条指令每个字节的二进制位模式。



操作码=MOV（立即数）
W=字
MOD=16位的位移量
REG=000（不用于立即寻址方式）
R/M=DS:[BX]
位移量=1000H
数据=1234H

图 4-9　转换成二进制机器语言的 MOV WORD PTR [BX + 1000H]，1234H 指令

这条指令的助记符形式包含 WORD PTR。WORD PTR 告诉汇编程序该指令用了长度为字的存储器指针。如果指令传送字节型的立即数，则用 BYTE PTR 代替指令中的 WORD PTR。类似地，如果指令用双字型立即数，则用 DWORD PTR 代替 BYTE PTR。大多数指令通过指针访问存储器时，不需要 BYTE PTR、WORD PTR 或 DWORD PTR 伪指令。只有不清楚操作数是字节、字还是双字时，才必须用它们。MOV [BX]，AL 指令，很清楚是字节传送指令，而 MOV [BX]，9 指令是不确定的，因为字节、字或双字长度的传送都有可能。这条指令必须被写成 MOV BYTE PTR [BX]，9 或 MOV WORD PTR [BX]，9 或 MOV DWORD PTR [BX]，9。如果不这样，因为汇编程序无法确定指令的意图（操作数的长度），将标识它存在错误。

段寄存器 MOV 指令

如果段寄存器的内容通过 MOV、PUSH 或 POP 指令传送，则用一组专门的寄存器位（REG 字段）选择段寄存器（见表 4-6）。

表 4-6 段寄存器选择位

代 码	段 寄 存 器
000	ES
001	CS①
010	SS
011	DS
100	FS
101	GS

① 微处理器不允许 MOV CS，R/M（16）及 POP CS。

图 4-10 给出了转换成二进制的 MOV BX，CS 指令。这种 MOV 指令的操作码不同于以前的 MOV 指令。段寄存器可以与任意 16 位寄存器或 16 位存储单元之间传送数据。例如，MOV [DI]，DS 指令将 DS 的内容存储到数据段由 DI 寻址的存储单元中。指令系统中不存在用立即寻址方式的段寄存器 MOV 指令。为了把立即数装入段寄存器，首先要将数据装入另外的寄存器，然后再传送到段寄存器。

虽然这里没有讲述全部机器语言编码，但是已为用机器语言编程提供了足够的信息。注意，用助记符汇编语言（即汇编语言）写出的程序很少用手工转换成二进制的机器语言，而是用汇编程序将助记符汇编语言转换为二进制的机器语言。因为微处理器有多达 100000 种以上的指令形式，虽然手工汇编不是不可能，但是非常耗费时间，因此应该使用汇编程序。

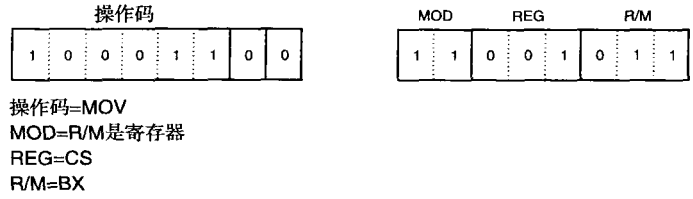


图 4-10 转换成二进制机器语言的 MOV BX，CS 指令

4.1.2 Pentium 4 和 Core2 的 64 位模式

目前涉及的信息还没有讨论 Pentium 4 或 Core2 的 64 位操作问题。64 位模式增加了一个额外的名为 REX（寄存器扩展）的前缀。被编码为 40H-4FH 的 REX 前缀跟在其他前缀之后，紧紧位于操作码之前，可以把操作码修改为 64 位操作模式。REX 前缀的目的是修改指令的第二个字节中的 reg 和 r/m 字段。REX 用于寻址 R8 ～ R15 的寄存器。图 4-11 阐明了 REX 的结构和它在操作码第二个字节上的应用。

为了实现 64 位操作，寄存器和内存对 rrrr 和 mmmm 字段的地址分配如表 4-7 所示。与在其他操作模式下一样，reg 字段只能包含寄存器的地址分配。r/m 字段则包含一个寄存器或内存地址分配。

表 4-7 64 位寄存器和内存的 rrrr 和 mmmm 字段的标志符

代 码	寄 存 器	内 存
0000	RAX	[RAX]
0001	RCX	[RCX]
0010	RDX	[RDX]
0011	RBX	[RBX]
0100	RSP①	—
0101	RBP	[RBP]
0110	RSI	[RSI]
0111	RDI	[RDI]
1000	R8	[R8]
1001	R9	[R9]
1010	R10	[R10]
1011	R11	[R11]
1100	R12	[R12]
1101	R13	[R13]
1110	R14	[R14]
1111	R15	[R15]

① 这个寻址模式指比例变址字节所包含的情况。

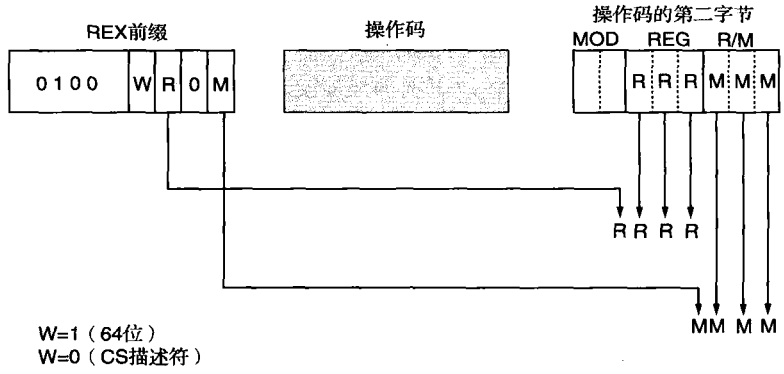


图 4-11 REX 无比例变址的应用

图 4-12 显示了采用比例变址字节和 REX 前缀来支持更加复杂的寻址模式和对 64 位操作模式下的比例因子的使用。与 32 位指令一样，比例变址字节允许的模式也同样允许寄存器对来寻址内存，比例因子也同样为 2×、4×或 8×。例如指令 MOV RAXW，[RDX + RCX-12]，它要求比例变址字节具有一个值为 1 的变址域，这是可以理解但从不会在指令中采用的。

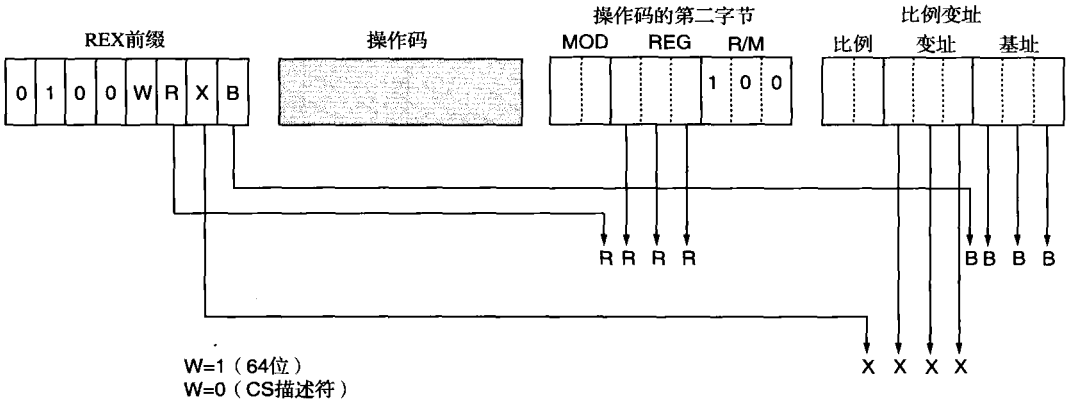


图 4-12 用于 64 位操作的比例变址字节和 REX 前缀

4.2 PUSH/POP 指令

PUSH 和 POP 指令很重要，它们用来将数据存入 LIFO（后进先出）堆栈存储器，或从堆栈存储器取出数据。微处理器有 6 种形式的 PUSH 和 POP 指令：寄存器、存储器、立即数、段寄存器、标志寄存器及全部寄存器。早期的 8086/8088 微处理器没有 PUSH 和 POP 立即数，以及 PUSHA 和 POPA（全部寄存器）指令，但是 80286 ～ Core2 有了。

用寄存器寻址，可将任何 16 位寄存器的内容传送到堆栈或者从堆栈传送到寄存器。在 80386 及更高型号中，32 位扩展寄存器和标志寄存器（EFLAGS）也可以压入到堆栈或者从堆栈弹出。采用存储器寻址的 PUSH 和 POP 指令，可将 16 位存储单元的内容（80386 或更高型号中是 32 位存储单元）压入到堆栈，或者从堆栈弹出到存储单元。立即寻址允许将立即数压入堆栈，但是不能从堆栈弹出。段寄存器寻址允许将任一段寄存器的内容压入堆栈或者从堆栈弹出（CS 的内容可以被压入堆栈，但是从堆栈弹出的数据绝对不能进入 CS）。标志寄存器可以压入或者从堆栈弹出，全部寄存器的内容可以压入或者从堆栈弹出。

4.2.1 PUSH 指令

8086 ～ 80286 PUSH 指令总是传送两个字节的数据到堆栈，而 80386 及更高型号可传送两个或四个

字节的数据，这取决于寄存器或存储单元的长度。数据源可以是任何内部的16/32位寄存器、立即数、任何段寄存器或者任何两字节的存储器数据。PUSHA指令把全部内部寄存器的内容复制到堆栈，但是段寄存器除外。**PUSHA**（**push all**）指令按照下面的顺序复制寄存器的内容到堆栈：AX、CX、DX、BX、SP、BP、SI和DI。压入堆栈的SP内容是其在PUSHA指令执行前的值。**PUSHF**（**push flags**）指令把标志寄存器的内容复制到堆栈。PUSHAD和POPAD指令压入或者弹出80386～Pentium 4中全部32位寄存器的内容。PUSHA和POPA指令在64位的Pentium 4中不起作用。

每当数据被压入到堆栈时，第一（最高有效）数据字节传送到由SP－1寻址的堆栈段存储单元。第二（最低有效）数据字节传送到由SP－2寻址的存储单元。数据用PUSH指令存储以后，SP寄存器的内容减2。双字压栈同样是这样，不同的是传送4个字节到堆栈存储器（最高有效字节首先压入），然后堆栈指针减4。图4-13给出了PUSH AX指令的操作。这条指令复制AX的内容到堆栈，SS：[SP－1] =AH，SS：[SP－2] =AL，然后SP=SP－2。在64中将使用8个字节压入堆栈。

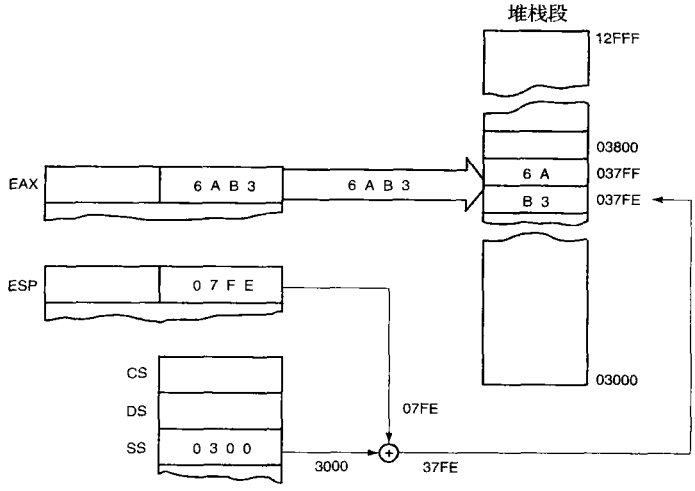


图4-13 执行PUSH AX指令对ESP及堆栈存储单元37FFH和37FEH的影响，并给出了这条指令执行以后的情况。

PUSHA指令将全部16位内部寄存器压入堆栈，如图4-14所示。这条指令存储8个16位寄存器的内容，需要16个字节的堆栈存储器空间。全部寄存器都压入堆栈以后，SP的内容减16。在80286及更高型号微处理器中，任务执行期间需要保存全部的寄存器时（微处理器环境），PUSHA指令是非常有用的。PUSHAD指令将80386～Core2中的全部32位寄存器压入堆栈。PUSHAD需要32字节的堆栈存储器空间。

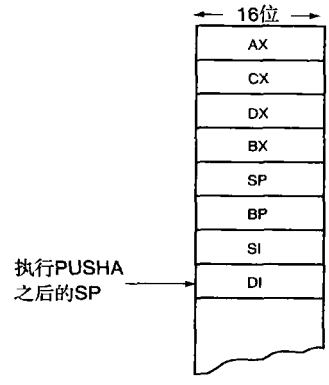


图4-14 PUSHA指令的操作，并列出了堆栈数据的位置和顺序

PUSH立即数指令有两个不同的操作码，但是两种操作码都是将一个16位立即数传送到堆栈中；如果用PUSHD，可将32位立即数压入堆栈中。如果立即数数值是00H～FFH，则操作码是6A；如果立即数数值是0100H～FFFFH，则操作码是68H。PUSH 8指令将0008H压入堆栈，汇编成6A08H，而PUSH 1000H指令汇编成680010H。另一个PUSH立即指令的例子是PUSH 'A'指令，是将0041H压入堆栈，这里41H是字母A的ASCII码。

表4-8给出了PUSH指令的格式，包括PUSHA和PUSHF。注意指令集怎样为汇编程序指定数据的长度。

表 4-8 PUSH 指令

符　号	例　子	注　释
PUSH reg16	PUSH BX	16 位寄存器
PUSH reg32	PUSH EDX	32 位寄存器
PUSH mem16	PUSH WORD PTR [BX]	16 位指针
PUSH mem32	PUSH DWORD PTR [EBX]	32 位指针
PUSH mem64	PUSH QWORD RTR [RBX]	64 位指针（64 位模式下）
PUSH seg	PUSH DS	段寄存器
PUSH imm8	PUSH 'R'	8 位立即数
PUSH imm16	PUSH 1000H	16 位立即数
PUSHD imm32	PUSHD 20	32 位立即数
PUSHA	PUSHA	保存所有 16 位寄存器
PUSHAD	PUSHAD	保存所有 32 位寄存器
PUSHF	PUSHF	保存标志寄存器
PUSHFD	PUSHFD	保存 EFLAGS

4.2.2 POP 指令

POP 指令实现与 PUSH 指令相反的操作。POP 指令从堆栈弹出数据，并且放入指定的 16 位寄存器、段寄存器或者 16 位存储单元。在 80386 及更高型号中，POP 指令可以从堆栈弹出 32 位数据，并且用 32 位的地址。POP 指令不能使用立即寻址方式。**POPF（POP flag）**指令从堆栈弹出 16 位数字放入标志寄存器。POPFD 从堆栈弹出 32 位数字放入扩展标志寄存器。**POPA（POP all）**指令从堆栈弹出 16 字节数据并且按顺序放入以下的寄存器中：DI、SI、BP、SP、BX、DX、CX 和 AX。顺序和用 PUSH A 指令把它们放入堆栈时的顺序相反，如此实现了把原来的数据返回到原来的寄存器中。在 80386 及更高型号微处理器中，POPAD 指令可从堆栈重新装载所有 32 位寄存器。

假定执行 POP BX 指令。从堆栈（堆栈段中用 SP 寻址的存储单元）弹出的第 1 个字节，放入寄存器 BL，从堆栈段存储器单元 SP + 1 处弹出第 2 字节并且放入寄存器 BH。从堆栈弹出两个字节以后，SP 寄存器的内容增 2。图 4-15 给出了 POP BX 指令怎样从堆栈弹出数据并且把它们放入寄存器 BX。

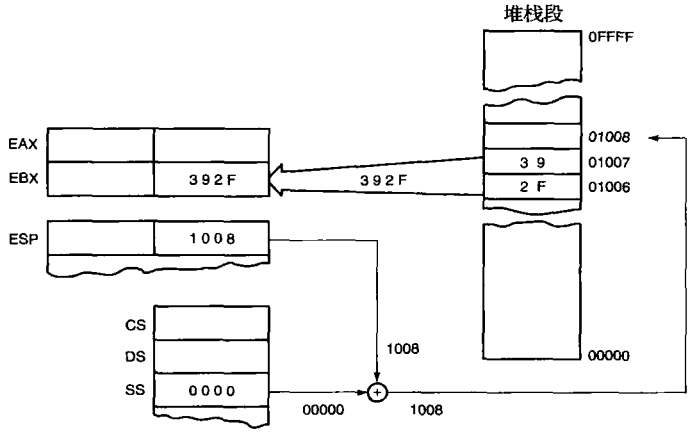


图 4-15 POP BX 指令，指出数据怎样从堆栈中弹出。给出了这条指令执行后的情况

POP 指令所用的操作码和它的全部类型在表 4-9 中给出。注意 POP CS 是无效的指令。如果执行 POP CS 指令，只改变了下一条指令的部分地址（CS），这使得 POP CS 指令造成了不可预知的结果，因此是不允许的。

表 4-9 POP 指令

符　号	例　子	注　释
POP reg16	POP CX	16 位寄存器
POP reg32	POP EBP	32 位寄存器
POP mem16	POP WORD PTR [BX + 1]	16 位指针
POP mem32	POP DATA 3	32 位存储地址
POP mem64	POP FROG	64 位存储地址（64 位）
POP seg	POP FS	段寄存器
POPA	POPA	恢复所有 16 位寄存器
POPAD	POPAD	恢复所有 32 位寄存器
POPF	POPF	恢复标志寄存器
POPFD	POPFD	恢复 EFLAGS

4.2.3 初始化堆栈

初始化堆栈时，应当加载堆栈段寄存器和堆栈指针寄存器。通常把堆栈段的栈底地址压入 SS，以便为堆栈段分配存储区域。

例如，如果堆栈段驻留在存储器地址 10000H～1FFFFH 处，则将 1000H 压入 SS（回忆一下，在实模式下，堆栈段寄存器最右端要添加一个 0）。将 0000H 压入栈指针（SP），以便指向这个 64K 字节堆栈段的顶部作为起始栈顶。同样的，为了指定 10FFFH 地址为起始栈顶，SP 中的值用 1000H。图 4-16 给出了 PUSH CX 指令怎样根据这个值将数据压入堆栈的顶部。记住，所有的段都是自然循环的，也就是说段顶部单元和段底部单元是邻接的。

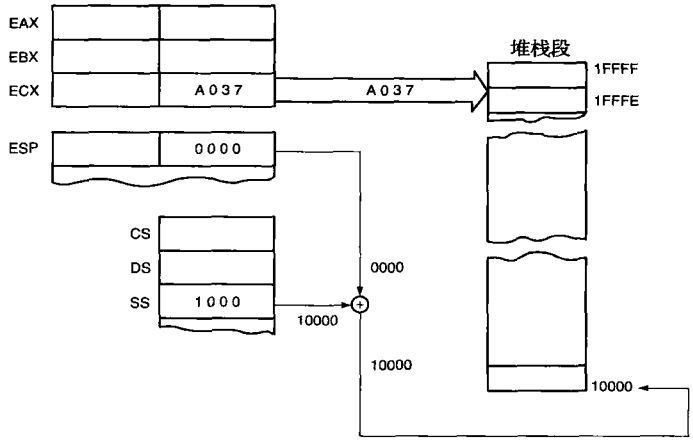


图 4-16 PUSH 指令。指示堆栈段的周期性质，说明这条指令执行前栈底与栈顶是邻接的

汇编语言中堆栈段的设置如例 4-1 所示。第一条语句定义堆栈段的开始，最后一条语句定义堆栈段的结束。汇编和连接程序将正确的堆栈段的地址压入 SS，把堆栈段的长度（栈顶地址）压入 SP。除非由于某些原因要改变这些初始化值，否则程序没有必要加载这些寄存器。

例 4-1

```
0000                    STACK_SEG       SEGMENT STACK
0000  0100 [                            DW 100H DUP (?)
           ????
               ]
0200                    STACK_SEG ENDS
```

另一种定义堆栈段的方法是使用存储器模型，这只适用于 MASM 汇编程序（参考附录 A），其他汇编程序不使用，即使使用，它们也与 MASM 中的不完全相同。例 4-2 中，.STACK 语句后是分配给堆栈的字节数，用于定义堆栈的范围。这个例子的功能与例 4-1 相同。.STACK 语句初始化 SS 和 SP。注意，书中使用了为 Microsoft 的宏汇编程序 MASM 设计的存储器模型。

例 4-2

```
.MODEL SMALL
.STACK 200H ; 设置堆栈
```

如果不使用以上任何方法确定堆栈的范围，程序连接时将出现警告。如果堆栈的深度是 128 字节或更少些，可以忽略这个警告。系统自动地（通过 DOS）分配至少 128 字节给堆栈存储器，这一存储区位于**程序段前缀（program segment prefix，PSP）**中，PSP 附加在每个程序文件的开头。如果堆栈需要用更多的存储区域，将覆盖对于程序及计算机操作都至关重要的 PSP 中的信息。这种错误常常会使计算机程序崩溃。如果使用 TINY 存储器模型，将自动把堆栈定位在每个段的末端，这样就可以得到较长的堆栈区域。

4.3 装入有效地址

在微处理器指令系统中，有几种装入有效地址的指令。LEA 指令把一偏移地址装入 16 位寄存器，这个地址由该指令选定的寻址方式确定。LDS 或 LES 指令把从存储单元取出的偏移地址装入任何 16 位寄存器，然后把从另一存储单元取出的段地址装入 DS 或 ES。80386 和更高型号微处理器的指令系统中增加了 LFS、LGS 和 LSS，从而可以选择 32 位寄存器接收存储器的 32 位位移量。在 Pentium 4 的 64 位模式下，LDS 和 LES 指令是无效的和不可用，这是因为段在平坦存储模式没有功能。表 4-10 列出了这些装入有效地址的指令。

表 4-10 装入有效地址指令

汇编语言指令	操 作
LEA AX，NUMB	将 NUMB 的偏移地址装入 AX
LEA EAX，NUMB	将 NUMB 的偏移地址装入 EAX
LDS DI，LIST	将数据段 LIST 存储单元的 32 位内容装入 DI 和 DS
LDS EDI，LIST1	将数据段 LIST 存储单元的 48 位内容装入 EDI 和 DS
LES BX，CAT	将数据段 CAT 存储单元的 32 位内容装入 BX 和 ES
LFS DI，DATA1	将数据段 DATA1 存储单元的 32 位内容装入 DI 和 FS
LGS SI，DATA5	将数据段 DATA5 存储单元的 32 位内容装入 SI 和 GS
LSS SP，MEM	将数据段 MEM 存储单元的 32 位内容装入 SP 和 SS

4.3.1 LEA 指令

LEA 指令把由操作数字段指定的数据的偏移地址装入 16 位或 32 位寄存器。以表 4-9 给出的第一行指令为例，装入寄存器 AX 的是地址 NUMB，而不是 NUMB 地址的内容。

比较 LEA 和 MOV 指令，很明显 LEA BX，[DI] 指令是将 DI 指示的偏移地址（DI 的内容）装入 BX；而 MOV BX，[DI] 则是将由 DI 寻址的存储单元内的数据装入寄存器 BX。

本书前面给出了几个用 OFFSET 伪指令的例子。如果操作数是位移量，则 OFFSET 伪指令实现的功能与 LEA 指令相同。例如，MOX BX，OFFSET LIST 实现的功能与 LEA BX，LIST 相同。这两条指令都是将存储单元 LIST 处的偏移地址装入 BX 寄存器。参考例 4-3 给出的短程序，将 DATA1 的地址装入 SI，DATA2 的地址装入 DI，然后交换这些存储单元的内容。注意 LEA 与带有 OFFSET 的 MOV 指令的长度相同（3 字节长）。

例 4-3

```
                    .MODEL SMALL          ;选择 SMALL 模型
0000                .DATA                 ;指示数据段开始
0000 2000   DATA1   DW     2000H          ;定义 DATA1
0002 3000   DATA2   DW     3000H          ;定义 DATA2
0000                .CODE                 ;指示代码段开始
                    .STARTUP              ;指示程序开始
0017 BE 0000 R      LES SI,DATA1          ;用 SI 寻址 DATA1
001A BF 0002 R      MOV DI,OFFSET DATA2 ;用 DI 寻址 DATA2
001D 8B 1C          MOV BX, [SI]          ;DATA1 与 DATA2 交换
001F 8B 0D          MOV CX, [DI]
0021 89 0C          MOV [SI],CX
0023 89 1D          MOV [DI],BX
                    .EXIT
                    END
```

既然 OFFSET 伪指令能完成相同的任务，为什么还用 LEA 指令呢？因为 OFFSET 只能用于如 LIST 那样的简单操作，它不能用于如［DI］、LIST［SI］这样的操作数。对于简单的操作数，OFFSET 伪指令比 LEA 指令更有效。微处理器执行 LEA BX，LIST 指令比执行 MOV BX，OFFSET LIST 花费的时间更长。例如，80486 微处理器执行 LEA BX，LIST 指令需要两个时钟周期，而执行 MOV BX，OFFSET LIST 只需要一个时钟周期。MOV BX，OFFSET LIST 指令执行快的原因是由汇编程序计算出了 LIST 的偏移地址，而 LEA 指令是微处理器执行时才计算的，因此 MOV BX，OFFSET LIST 指令效率更高。

假定微处理器执行 LEA BX，［DI］指令，DI 的内容是 1000H。由于 DI 包含了偏移地址，微处理器 DI 的内容复制到 BX 中。MOV BX，DI 指令以较少的时间完成这个任务，通常它优于 LEA BX，［DI］指令。

另一个例子是 LEA SI，［BX + DI］，这条指令将 BX 的内容加到 DI，它们的和存入 SI 寄存器。这些寄存器产生的和是以 64K 为模的和。以 64K 为模的和丢弃 16 位结果的进位。如果 BX = 1000H，DI = 2000H，送入 SI 的偏移地址是 3000H。如果 BX = 1000H，DI = FF00H，则偏移地址是 0F00H，而不是 10F00H。注意第二个地址 0F00H 是以 64K 为模的和。

4.3.2 LDS、LES、LFS、LGS 和 LSS 指令

LDS、LES、LFS、LGS 和 LSS 指令把偏移地址装入任何 16 位或 32 位寄存器，并且把段地址装入 DS、ES、FS、GS 或 SS 段寄存器。这些指令可以用任何寻址方式访问 32 位或 48 位存储区，该区包含段地址和偏移地址。32 位的存储器区域包含 16 位的偏移地址和 16 位段地址，而 48 位的存储区包含 32 位的偏移地址和 16 位段地址。这些指令不能用寄存器寻址方式（MOD = 11）。注意 LFS、LGS 和 LSS 指令同 32 位寄存器一样，只用于 80386 及更高型号的微处理器。

图 4-17 图解说明了 LDS BX，［DI］指令。这条指令将数据段中由 DI 寻址的 32 位数传送到 BX 和 DS 寄存器。LDS、LES、LFS、LGS 和 LSS 指令从存储器获得新的远地址。注意，偏移地址在先，段地址在后。这种形式常用来存储所有 32 位的存储器地址。

远地址可以由汇编程序存入存储器。例如：ADDR DD FAR PTR FROG 指令将 FROG 的偏移地址和段地址（远地址）存入从 ADDR 处开始的 32 位存储区中。DD 伪指令通知汇编程序在存储器地址 ADDR 处存入 32 位的双字。

在 80386 和更高型号的微处理器中，LDS EBX，［DI］指令将数据段中由 DI 寻址的存储区的 4 个字节装入 EBX，然后将这 4 个字节后面的字装入 DS 寄存器。注意，在 80386 及更高型号的微处理器中，当 32 位偏移地址装入 32 位寄存器时，可寻址 48 位存储器区而不是寻址 32 位存储器区。开头 4 个字节包含装入 32 位寄存器的偏移值，后两个字节包含段地址。

最实用的装入指令是 LSS 指令。例 4-4 给出了保存旧的堆栈区地址以后，建立新堆栈区的短程序。在执行一些指令以后，通过用 LSS 指令装入 SS 和 SP 再重新激活旧的堆栈区。注意，为了禁止中断，

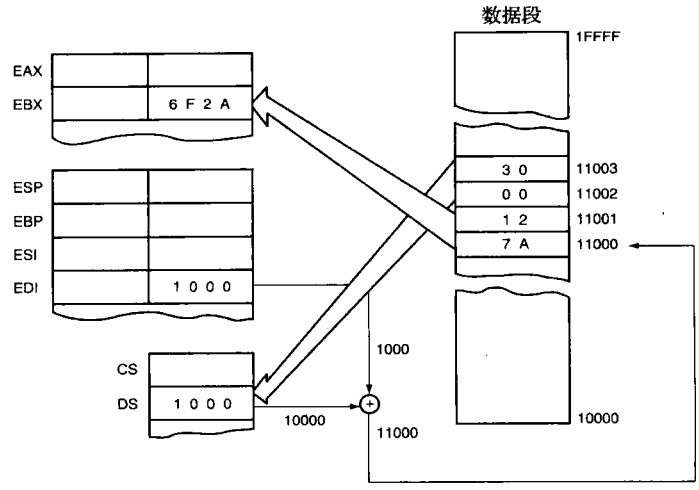


图 4-17 LDS BX,[DI] 指令将 11000H 和 11001H 单元的内容装入寄存器 BX，将 11002H 和 11003H 单元的内容装入寄存器 DS，并指出了在 DS 变为 3000H 和 BX 变为 127A 以前的情况

必须包含 **CLI**（**disable interrupt，禁止中断**）和 **STI**（**enable interrupt，使能中断**）指令，这是本章最后要讨论的主题。由于 LSS 指令作用于 80386 及更高型号的微处理器，因此，.386 语句出现在 .MODE 语句后面选择 80386 微处理器。还要注意：为寻址旧堆栈存储区，可使用 WORD PTR 字指针超越双字定义符（DD）。如果使用 80386 或更新的微处理器，建议开发 80386 微处理器软件时使用 .386 开关。即使微处理器是 Pentium、Pentium Pro、Pentium Ⅱ、Pentium Ⅲ、Pentium 4 或 Core2 也应如此。理由是：与 80386 相比，80486 ～ Core2 只提供了很少的几条附加指令，并且在软件开发中很少使用。如果要求使用 CMPXCHG、CMPXCHG8（Pentium 中新的指令）、XADD 或 BSWAP 指令，则用 .486 开关选择 80486 微处理器，而用 .586 开关选择 Pentium ～ Pentium 4。甚至可用 .686 开关指定 Pentium Ⅱ～ Core2。

例 4-4

```
                        .MODEL SMALL            ;选择 SMALL 模型
                        .386                    ;选择 80386 微处理器
0000                    .DATA                   ;指示数据段开始
0000 0000000        SADDR DD    ?               ;旧的堆栈地址
0004 1000 [         SAREA DW    1000H DUP (?)   ;新的堆栈区域
          ????
             ]
2004 =2004          STOP  EQU THIS WORD         ;定义新的堆栈
0000                      .CODE                 ;指示代码段开始
                          .STARTUP              ;指示程序开始
0010 FA                   CLI                   ;禁止中断
0011 8B C4                MOV AX,SP             ;保存旧的 SP
0013 A3 0000 R            MOV WORD PTR SADDR,AX
0016 8C D0                MOV AX,SS             ;保存旧的 SS
0018 A3 0002 R            MOV WORD PTR SADDR+2,AX


001B 8C D8                MOV AX,DS             ;装入新的 SS
001D 8E D0                MOV SS,AX
001F B8 2004 R            MOV AX,OFFSET STOP    ;装入新的 SP
0022 8B E0                MOV SP,AX
0024 FB                   STI                   ;允许中断
```

```
0025  8B C0               MOV AX,AX          ;执行哑指令
0027  8B C0               MOV AX,AX
0029  9F B2 26 0000 R     LSS SP,SADDR       ;装入旧的 SS 和 SP
                          .EXIT              ;返回 DOS
                          END                ;指示文件结束
```

4.4 数据串传送

有5条数据串传送指令：LODS、STOS、MOVS、INS和OUTS。每种数据串传送指令都允许传送一个字节、字或双字数据（如果是重复传送，则是字节块、字块或双字块）。在使用串指令前必须了解D标志位（方向位）和DI及SI寄存器对串指令的作用。在64位的Pentium 4和Core2中四字块也可以使用串指令如LODSQ。

4.4.1 方向标志

方向标志（D）位于标志寄存器中，用于在串操作期间选择DI和SI寄存器自动递增（D=0）或是自动递减（D=1）操作。方向标志只用于串操作指令。CLD指令清除方向标志D（D=0），而STD指令置位它（D=1）。因此CLD指令选择自动递增方式（D=0），而STD指令选择自动递减方式（D=1）。

每次当串操作指令传送字节时，DI和/或SI的内容加1或者减1；如果传送字，DI和/或SI增2或者减2；传送双字使得DI和/或SI增4或者减4。只有被串操作指令实际使用的寄存器才增量或者减量。例如STOSB指令只用DI寄存器寻址存储器单元，执行STOSB指令时，只有DI寄存器被增量或者减量，并不作用于SI。同样的，LODSB指令使用SI寄存器寻址存储器数据。LODSB指令只使SI增量或减量而不影响DI。

4.4.2 DI和SI

串操作指令执行期间，对存储器的访问是通过DI和SI两个寄存器或其中之一实现的。对于所有用DI的串操作指令而言，DI偏移地址是用于访问附加段中的数据；SI偏移地址默认用于访问数据段中的数据。SI的段分配可以通过本章后面描述的段超越前缀来改变。当执行串指令时，分配给DI的段总是附加段，这种分配不能改变。在MOVS指令中，一个指针寻址附加段中的数据，另一个指针寻址数据段中的数据，这样它可以将64K字节的数据从存储器的一个段传送到另一个段。

当80386及更高型号的微处理器操作在32位模式时，用EDI和ESI代替DI和SI，允许串指令使用微处理器全部4G字节保护模式寻址空间的任何存储单元。

4.4.3 LODS指令

LODS指令将存储在数据段用SI寄存器寻址的数据装入AL，AX或EAX（注意，只有80386和更高型号的微处理器可以用EAX）。将字节装入AL，字装入AX，双字装入EAX以后，如果D=0，SI寄存器的内容增量；如果D=1，SI寄存器的内容减量。如果是字节型LODS，则SI加1或减1；如果是字型LODS，则SI加2或减2；如果是双字型LODS，则SI加4或减4。

表4-11列出了LODS指令允许的格式。**LODSB**（**load a byte，装入字节**）指令将字节装入AL，

表4-11 LODS指令的格式

汇编语言指令	操　作
LODSB	AL=DS:[SI]；SI=SI±1
LODSW	AX=DS:[SI]；SI=SI±2
LODSD	EAX=DS:[SI]；SI=SI±4
LODSQ	RAX=[RSI]；[RSI] =RSI±8（64位）
LODS LIST	AL=DS:[SI]；SI=SI±1（如果LIST是字节）
LODS DATA1	AX=DS:[SI]；SI=SI±2（如果DATA1是字）
LODS FROG	EAX=DS:[SI]；SI=SI±4（如果FROG是双字）

注：段寄存器可以用段超越前缀替换，如同LODS ES:DATA4。

LODSW（load a word，装入字）将一个字装入 AX，而 **LODSD（load a doubleword，装入双字）**将双字装入 EAX。也可以用 LODS 指令后面跟着字节、字、或双字操作数的方式代替 LODSB、LODSW 或 LODSD，但不常用。操作数常用 DB 定义为字节，用 DW 定义为字，用 DD 定义为双字。伪指令 DB 定义字节，伪指令 DW 定义字，伪指令 DD 定义双字。

图 4-18 给出了执行 LODSW 指令的结果，假定标志 D = 0，SI = 1000H 和 DS = 1000H。这里将存储在存储器地址 11000H 和 11001H 的 16 位数传送到 AX。因为 D = 0，而且是字传送，因此存储器数据装入 AX 以后，SI 寄存器的内容增 2。

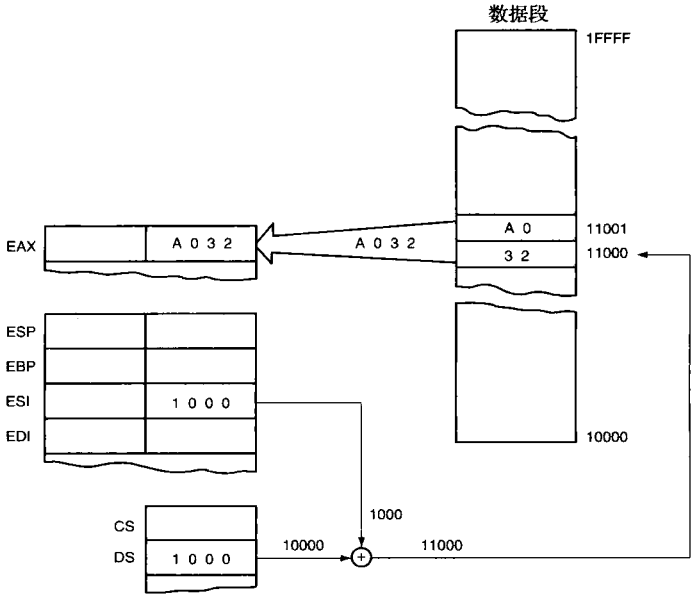


图 4-18　LODSW 指令的操作。假定 DS = 1000H，SI = 1000H，D = 0，11000H = 32，11001H = A0。给出了存储器内容装入 AX 以后和 SI 增 2 之前的情况

4.4.4　STOS 指令

STOS 指令将 AL，AX 或 EAX 存储到附加段内由 DI 寄存器寻址的存储单元（注意只有 80386 ～ Core2 中可以使用 EAX 和双字）。表 4-12 列出了 STOS 指令的全部格式，类似于 LODS 指令，为了传送字节、字或双字，STOS 指令可以附加 B、W 或 D。**STOSB（store a byte，存储字节）**指令将 AL 中的字节存入附加段由 DI 寻址的存储单元；**STOSW（store a word，存储字）**指令将 AX 中的字存入附加段由 DI 寻址的存储单元；**STOSD（store a doubleword，存储双字）**指令将 EAX 中的双字存入附加段由 DI 寻址的存储单元。字节（AL）、字（AX）或双字（EAX）存储以后，DI 的内容增量或减量。

表 4-12　STOS 指令的格式

汇编语言指令	操　作
STOSB	ES:[DI] = AL; DI = DI ± 1
STOSW	ES:[DI] = AX; DI = DI ± 2
STOSD	ES:[DI] = EAX; DI = DI ± 4;
STOSQ	[RDI] = RAX; RDI = RDI ± 8（64 位）
STOS LIST	ES:[DI] = AL; DI = DI ± 1（如果 LIST 是字节）
STOS DATA3	ES:[DI] = AX; DI = DI ± 2（如果 DATA3 是字）
STOS DATA4	ES:[DI] = EAX ; DI = DI ± 4（如果 DATA4 是双字）

带有 REP 的 STOS

重复前缀（repeat prefix，REP）可以加到除了 LODS 指令以外的任何串数据传送指令上。执行重复的 LODS 操作没有任何意义。REP 前缀使得每次执行串指令后 CX 减 1。CX 减 1 以后，重复执行串指令，直到 CX 值为 0 时，指令终止，程序继续执行指令序列的下一条。因此，如果 CX 装入了 100，执行 REP STOSB 指令，则微处理器自动重复执行 STOSB 指令 100 次。因为每个数据存储以后，DI 寄存器自动增量或减量，所以这条指令是将 AL 的内容存入存储块，而不是单个的字节存储单元。在 64 位的 Pentium 4 中 RCX 寄存器使用 REP 前缀。

假定 STOSW 指令用于在 C++ 环境下用内嵌汇编清除一个称为 Buffer 的存储区，用一个称为 Count 的计数器，该程序是一个称为 ClearBuffer 的功能函数（见例 4-5）。注意，参数 Count 和 Buffer 地址要传送给功能函数。REP STOSW 指令清除称为 Buffer 的存储缓冲区。注意，Buffer 是被该功能函数清除的实际缓冲区的指针。

例 4-5

```
void ClearBuffer (int Count,short* Buffer)
{
    _asm {
          push edi          ;保存寄存器中的内容
          push es
          push ds
          mov ax,0
          mov ecx,Count
          mov edi,Buffer
          pop es            ;把 DS 装入 ES
          rep stosw         ;清除缓冲区
          pop es            ;恢复寄存器
          pop edi
  }
}
```

程序中的操作数可以用乘号（*）之类的算术或逻辑运算符修改。其他运算符列在表 4-13 中。

表 4-13 常用操作数的运算符

运 算	例 子	注 释
+	MOV AL, 6+3	把 9 复制到 AL
-	MOV AL, 6-3	把 3 复制到 AL
*	MOV AL, 4*3	把 12 复制到 CX
/	MOV AX, 12/5	把 2 复制到 AX（余数丢失）
MOD	MOV AX, 12 MOD 7	把 5 复制到 AX（商丢失）
AND	MOV AX, 12 AND 4	把 4 复制到 AX（1100 AND 0100 = 0100）
OR	MOV EAX, 12 OR 1	把 13 复制到 EAX（1100 OR 0001 = 1101）
NOT	MOV AL, NOT 1	把 254 复制到 AL（0000 0001 的非等于 1111 1110，即 254）

4.4.5 MOVS 指令

MOVS 是更实用的数据串传送指令之一，因为它将数据从一个存储单元传送到另一个存储单元。这是 8086 ~ Pentium 4 惟一允许的存储器到存储器的传送指令。MOVS 指令从数据段内由 SI 寻址的存储单元把字节、字或双字传送到附加段内由 DI 寻址的存储单元，和其他串指令一样，然后根据方向标志的指示指针增量或减量。表 4-14 列出了许可的所有 MOVS 指令的格式。注意，只能对源操作数（SI）（通常位于数据段）使用段超越前缀，因此它可以放在其他段，而目的操作数（DI）必须放在附加段。

例 4-6

```
//函数功能是使用内联汇编程序把BlockA内容复制到BlockB里
//
void TransferBlocks (int BlockSize, int* BlockA, int* BlockB)
{
      _asm{
            push es                    ;保存寄存器中的内容
            push edi
            push esi
            push ds                    ;把DS复制到ES
            pop  es
            mov  esi, BlockA           ;取得块A的地址BlockA
            mov  edi, BlockB           ;取得块B的地址BlockB
            mov  ecx, BlockSize        ;把块的大小装入ecx
            rep  movsd                 ;传送数据
            pop  es                    ;恢复寄存器
            pop  esi
            pop  edi
      }
}
```

例 4-7

```
//C++版的例4-6
//
void TransferBlocks (int BlockSize, int* BlockA, int*BlockB)
{
      for (int a = 0; a < BlockSize; a++)
      {
            BlockA = BlockB++;
            BlockA++;
      }
}
```

例 4-8

```
void TransferBlocks(int BlockSize, int* BlockA, int* BlockB)
{
004136A0  push        ebp
004136A1  mov         ebp,esp
004136A3  sub         esp,0D8h
004136A9  push        ebx
004136AA  push        esi
004136AB  push        edi
004136AC  push        ecx
004136AD  lea         edi,[ebp-0D8h]
004136B3  mov         ecx,36h
004136B8  mov         eax,0CCCCCCCCh
004136BD  rep stos    dword ptr [edi]
004136BF  pop         ecx
004136C0  mov         dword ptr [ebp-8],ecx
      for( int a = 0; a < BlockSize; a++ )
004136C3  mov         dword ptr [a],0
004136CA  jmp         TransferBlocks+35h (4136D5h)
004136CC  mov         eax,dword ptr [a]
004136CF  add         eax,1
004136D2  mov         dword ptr [a],eax
004136D5  mov         eax,dword ptr [a]
004136D8  cmp         eax,dword ptr [BlockSize]
004136DB  jge         TransferBlocks+57h (4136F7h)
      {
          BlockA = BlockB++;
004136DD  mov         eax,dword ptr [BlockB]
004136E0  mov         dword ptr [BlockA],eax
004136E3  mov         ecx,dword ptr [BlockB]
004136E6  add         ecx,4
004136E9  mov         dword ptr [BlockB],ecx
```

```
            BlockA++;
004136EC  mov         eax,dword ptr [BlockA]
004136EF  add         eax,4
004136F2  mov         dword ptr [BlockA],eax
        }
004136F5  jmp         TransferBlocks+2Ch (4136CCh)
}
004136F7  pop         edi
004136F8  pop         esi
004136F9  pop         ebx
004136FA  mov         esp,ebp
004136FC  pop         ebp
004136FD  ret         0Ch
```

经常需要把存储器一个区域的内容转移到另一个区域。假定有两块双字存储器块，BlockA 和 BlockB，需要把 BlockA 复制到 BlockB 里。例4-6表明，这可以用 MOVSD 指令来完成，它被用在由内嵌汇编写成的 C++ 功能函数里。该功能函数从调用者那里收到三个信息：块的大小、块 A 的地址 BlockA 和块 B 的地址 BlockB。注意，所有数据在一个 Visual C++ 程序的数据段里，因此需要用 PUSH DS 跟着一个 POP ES 把 DS 复制到 ES。还要保存除 EAX、EBX、ECX 和 EDX 之外所有被改变的寄存器。

例4-7表示单独用 C++ 写的同样的功能函数，两种方法可对照比较。为了和例4-6比较，例4-8给出例4-7的汇编语言版本，注意到与例4-8 C++ 生成的汇编语言版本比较，例4-6要短得多。虽然 C++ 版本很容易键入，但如果执行速度是重要的，例4-6要比例4-7运行快得多。

表4-14 MOVS 指令的格式

汇编语言指令	操 作
MOVSB	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1（传送字节）
MOVSW	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2（传送字）
MOVSD	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4（传送双字）
MOVSQ	[RDI] = [RSI]; RDI = RDI ± 8; RSI = RSI ± 8（64位）
MOVS BYTE1, BYTE2	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1（如果 BYTE1 和 BYTE2 是字节型的）
MOVS WORD1, WORD2	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2（如果 WORD1 和 WORD2 是字型的）
MOVS TED, FRED	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4（如果 TED 和 FRED 是双字型的）

4.4.6 INS 指令

INS（input string，串输入）指令（不能用于 8086/8088 微处理器）从 I/O 设备把字节、字或双字数据传送到附加段内由 DI 寻址的存储单元。I/O 地址存放在 DX 寄存器中。这条指令对于将外部 I/O 设备的数据块直接输入到存储器非常有用。应用程序可以把数据从磁盘驱动器传送到存储器，磁盘驱动器很常见，并作为 I/O 设备与计算机系统接口。

类似于上述串操作指令，INS 指令有三种基本的格式，INSB 从 8 位 I/O 设备输入数据并且存入 DI 指向的字节存储单元，INSW 指令从 16 位 I/O 设备输入数据并且存入字存储单元，INSD 指令输入一个双字。这些指令可以使用 REP 前缀重复操作，这就允许从 I/O 设备输入完整的数据块并存入存储器。表4-15列出了各种 INS 指令的格式。注意，在 64 位模式中没有 64 位输入，但是存储地址是 64 位并由 INS 指令在 RDI 中定位。

表4-15 INS 指令的格式

汇编语言指令	操 作
INSB	ES:[DI] = [DX]; DI = DI ± 1（传送字节）
INSW	ES:[DI] = [DX]; DI = DI ± 2（传送字）
INSD	ES:[DI] = [DX]; DI = DI ± 4（传送双字）
INS LIST	ES:[DI] = [DX]; DI = DI ± 1（如果 LIST 是字节）
INS DATA4	ES:[DI] = [DX]; DI = DI ± 2（如果 DATA4 是字）
INS DATA5	ES:[DI] = [DX]; DI = DI ± 4（如果 DATA5 是双字）

注：[DX] 指明 DX 含有 I/O 设备地址。这些指令不能用于 8086/8088 微处理器。

例 4-9 给出一个指令序列，从地址为 03ACH 的 I/O 设备输入 50 个字节数据存入附加段的 LISTS 存储器数组中。该软件假定随时都可以使用来自 I/O 设备的数据，否则软件必须检验 I/O 设备是否准备好传送数据，不能使用 REP 前缀。

例 4-9

```
                    ;用 REP INSB 输入数据到内存数组
                    ;
0000 BF 0000 R      MOV DI,OFFSET LISTS     ;用 DI 寻址数组
0003 BA 03AC        MOV DX,3ACH             ;寻址 I/O
0006 FC             CLD                     ;自动加 1
0007 B9 0032        MOV CX,50               ;装入计数值
000A F3/6C          REP INSB                ;输入数据
```

4.4.7 OUTS 指令

OUTS（output string，串输出）指令从数据段把由 SI 寻址的存储单元的字节、字或双字传送到 I/O 设备（不能用于 8086/8088 微处理器）。类似于 INS 指令，I/O 设备由 DX 寄存器内容寻址。表 4-16 给出了各种可用的 OUTS 指令格式。在 64 位的 Pentium 4 和 Core2 中，没有 64 位的输出，但是 RSI 的地址是 64 位宽的。

表 4-16 OUTS 指令的格式

汇编语言指令	操　作
OUTSB	[DX] =DS:[SI]; SI=SI±1（传送字节）
OUTSW	[DX] =DS:[SI]; SI=SI±2（传送字）
OUTSD	[DX] =DS:[SI]; SI=SI±4（传送双字）
OUTS DATA7	[DX] =DS:[SI]; SI=SI±1（如果 DATA7 是字节）
OUTS DATA8	[DX] =DS:[SI]; SI=SI±2（如果 DATA8 是字）
OUTS DATA9	[DX] =DS:[SI]; SI=SI±4（如果 DATA9 是双字）

注：[DX] 指明 DX 包含 I/O 设备的地址。这些指令不能用于 8086/8088 微处理器。

例 4-10 给出了一个短指令序列，从数据段的存储器数组（ARRAY）把数据传送到地址为 3ACH 的 I/O 设备。该软件假定 I/O 设备总是把数据准备好了。

例 4-10

```
                    ;用 REP INSB 从内存数组输出数据
                    ;
0000 BE 0064 R      MOV SI,OFFSET ARRAY     ;用 SI 寻址数组
0003 BA 03AC        MOV DX,3ACH             ;寻址 I/O
0006 FC             CLD                     ;自动加 1
0007 B9 0064        MOV CX,100              ;装入计数值
000A F3/6E          REP OUTSB               ;输出数据
```

4.5 其他数据传送指令

在程序中确实使用了其他一些数据传送指令。这一节讨论的数据传送指令是：XCHG、LAHF、SAHF、XLAT、IN、OUT、BSWAP、MOVSX、MOVZX 和 CMOV。由于其他各种指令不如 MOV 指令那样经常使用，所以这一节集中说明它们。

4.5.1 XCHG 指令

交换指令（exchange，XCHG）将寄存器的内容与任何其他寄存器或存储单元的内容交换。XCHG 指令不能实现段寄存器之间的交换，或存储器和存储器之间的数据交换。可以交换字节、字或双字长度（80386 及更高型号的微处理器），并且可以使用第 3 章中讨论的除了立即寻址以外的任何寻址方

式。表4-17给出了一些XCHG指令的例子。

表4-17 XCHG指令的格式

汇编语言语句	操 作
XCHG AL, CL	AL与CL的内容交换
XCHG CX, BP	CX与BP的内容交换
XCHG EDX, ESI	EDX与ESI的内容交换
XCHG AL, DATA2	AL与数据段存储单元DATA2的内容交换
XCHG RBX, RCX	RBX与RCX的内容交换

使用16位的AX寄存器与另外一个16位寄存器的XCHG指令是最有效的交换指令。这种指令占用一个字节存储器。其他的XCHG指令需要两个或更多字节的存储器，取决于所选的寻址方式。

当使用汇编程序并使用存储器寻址方式时，用哪个操作数去寻址存储器并不重要。同一汇编程序里XCHG AL, [DI]指令就是XCHG [DI], AL指令。

如果使用80386～Core2微处理器，则XCHG指令可以交换双字数据。例如，XCHG EAX, EBX指令即可交换EAX寄存器与EBX寄存器的内容。

4.5.2 LAHF和SAHF指令

因为LAHF和SAHF指令被设计成一种桥接指令，现已很少使用。这些指令允许通过翻译程序把8085（一种早期的8位微处理器）软件转化为8086软件。由于需要转化的软件很可能是许多年以前的，这些指令今天已很少用到。LAHF指令是把标志寄存器的最右面的8位传送到AH寄存器。SAHF指令把AH寄存器传送到标志寄存器的最右面8位。

偶尔在使用数字协处理器的应用软件中可能会发现SAHF指令。数字协处理器里的状态寄存器内容要用FSTSW指令复制到AX中。再用SAHF指令把AH复制到标志寄存器，然后测试这些标志，以便了解数字协处理器的某些状态。第14章将详细地叙述这些内容，说明数字协处理器的编程和操作。由于LAHF和LAFH是旧系统的指令，它们在64位模式下是无效的和不起作用的。

4.5.3 XLAT指令

XLAT（translate，换码）指令把AL寄存器中的内容转换成存储在存储器表中的一个数字。这条指令通常使用于查找表技术，实现将一个代码转换为另一个代码。XLAT指令首先将AL与BX的内容相加，形成数据段内的存储器地址，然后将这个地址中的内容复制到AL中。这是惟一一条把8位数字加到16位数字上的指令。

假定7段LED显示器编码查找表存放在存储器地址TABLE处。用XLAT指令把AL中BCD码数字转换成AL中的7段码。例4-11提供了从BCD码转换成7段码的短程序。图4-19指出了这个例子程序的操作。如果TABLE=1000H，DS=1000H，初始化AL=05H（BCD码5）。转换以后AL=6DH。

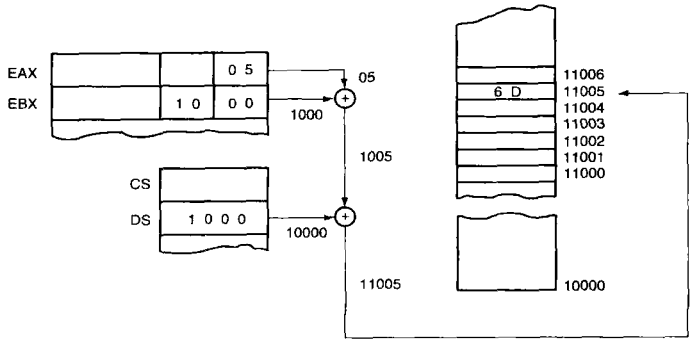


图4-19 6DH装入AL前XLAT指令的操作

例 4-11

```
                    TABLE  DB  3FH, 06H, 5BH, 4FH    ; 查找表
                           DB  66H, 6DH, 7DH, 27H
                           DB  7FH, 6FH
0017 B0 05          LOOK:  MOV AL,5                  ; 把 5（测试数据）装入 AL
0019 BB 1000 R             MOV BX,OFFSET TABLE       ; 寻址查找表
001C D7                    XLAT                      ; 转换
```

4.5.4 IN 和 OUT 指令

表 4-18 列出了执行 I/O 操作的 IN 和 OUT 指令的格式。注意，在 I/O 设备与微处理器之间只能传送 AL、AX 或 EAX 的内容。IN 指令将外部 I/O 设备的数据传送到 AL、AX 或 EAX，而 OUT 指令传送 AL、AX 或 EAX 的数据到外部的 I/O 设备（只有 80386 及更高型号的微处理器有 EAX）。

表 4-18 IN 和 OUT 指令

汇编语言指令	操　作
IN AL, p8	从 P8 端口输入 8 位数据到 AL
IN AX, p8	从 P8 端口输入 16 位数据到 AX
IN EAX, p8	从 P8 端口输入 32 位数据到 EAX
IN AL DX	从 DX 端口输入 8 位数据到 AL
IN AX, DX	从 DX 端口输入 16 位数据到 AX
IN EAX, DX	从 DX 端口输入 32 位数据到 EAX
OUT p8, AL	把 AL 中的 8 位数据发送到 P8 端口
OUT p8, AX	把 AX 中的 16 位数据发送到 P8 端口
OUT p8, EAX	把 EAX 中的 32 位数据发送到 P8 端口
OUT DX, AL	把 AL 中的 8 位数据发送到端口 DX
OUT DX, AX	把 AX 中的 16 位数据发送到端口 DX
OUT DX, EAX	把 EAX 中的 32 位数据发送到端口 DX

注：P8 代表 8 位 I/O 端口号数，DX 中存放 16 位端口地址。

对于 IN 和 OUT 指令，I/O 设备地址端口（Port）以两种形式存在：固定端口和可变端口。固定端口寻址允许在 AL、AX 或 EAX 与使用 8 位 I/O 端口地址的设备之间传送数据。因为端口号跟在指令操作码后面，所以称为固定端口寻址。通常，指令存储在 ROM 中，因为 ROM 是只读的，存储在 ROM 中的固定端口指令有永久的固定端口号。如果固定端口地址存储在 RAM 中，它有可能被修改，而这样的修改不是好的程序设计风格。

I/O 操作期间，端口地址出现在地址总线上。对于 8 位固定端口的 I/O 指令，8 位端口地址用零扩展成 16 位地址。例如，执行 IN AL，6AH 指令时，将来自 I/O 地址 6AH 的数据输入 AL。地址以 16 位的 006AH 的形式出现在地址总线 A0 ～ A15 上。对于 IN 或 OUT 指令，地址总线位 A16 ～ A19（8086/8088）、A16 ～ A23（80286/80386SX）、A16 ～ A24（80386SL/80386 SLC/80386EX）或 A16 ～ A32（80386 ～ Core2）是没有定义的。注意，Intel 为它的某些外围部件保留最后的 16 个 I/O 端口。

可变端口寻址允许数据在 AL、AX 或 EAX 与 16 位端口地址之间传送。称为可变端口寻址是因为在程序执行期间寄存器 DX 中存放的 I/O 端口号可以改变。16 位 I/O 端口地址出现在地址总线 A0 ～ A15 上。IBM PC 用 16 位端口地址访问它的 I/O 空间，PC 的 ISA 总线的 I/O 空间位于 I/O 端口 0000H ～ 03FFH。注意，PCI 总线使用的 I/O 地址可能超过 03FFH。

图 4-20 说明了 OUT 19H，AX 指令的执行，将 AX 的内容传送到 I/O 端口 19H。注意，I/O 端口号以 0019H 的形式出现在 16 位地址总线上，而来自 AX 的数据出现在微处理器数据总线上。系统控制信号$\overline{\text{IOWC}}$（I/O 写控制）为逻辑 0 时，允许向 I/O 设备传送数据。

例 4-12 中的短程序使 PC 中的扬声器发出“咔哒”声。通过访问 I/O 端口 61H 控制扬声器发声（只能在 DOS 中），如果这个端口最右边两位被置位（11），然后又被清除（00），就听到了扬声器

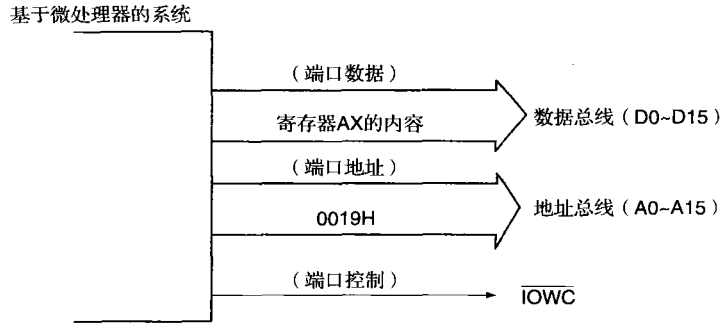


图 4-20 基于微处理器系统的信号对 OUT 19H，AX 指令的作用

“咔哒”声。注意这个程序用 OR 指令设置这两位，用 AND 指令清除它们。这些逻辑操作指令在第 5 章中说明。MOV CX，8000H 指令后面跟着 LOOP L1 指令用来延时。如果增加计数值，咔哒声将变长；如果减少计数值，咔哒声将变短；想要听到一串咔哒声，必须修改程序，重复许多次。

例 4-12

```
                .MODEL TINY             ;选择 TINY 模型
0000            .CODE                   ;指示代码段开始
                .STARTUP                ;指示程序开始
0100  E4  61          IN AL,61H         ;读端口 61H
0102  0C  03          OR AL,3           ;置位最右两位
0104  E6  61          OUT 61H,AL        ;使扬声器发声
0106  B9  8000        MOV CX,8000H      ;装入用来延时的计数值
0109            L1:
0109  E2  FE          LOOP L1           ;延时
010B  E4  61          IN AL,61H         ;读端口 61H
010D  24  FC          AND AL,0FCH       ;清除最右两位
010F  E6  61          OUT 61H,AL        ;使扬声器关闭
                .EXIT                   ;返回 DOS
                END                     ;指示文件结束
```

4.5.5 MOVSX 和 MOVZX 指令

MOVSX（move and sign-extend，传送及符号扩展）和 **MOVZX（move and zero-extend，传送及零扩展）**指令只出现在 80386 ～ Pentium 4 指令系统中，这些指令传送数据时，对数据进行符号扩展或者零扩展。表 4-19 用几个例子说明了这些指令如何使用。

当数字是零扩展的时，最高有效部分用零填充。例如，如果 8 位的 34H 零扩展成 16 位数字，是 0034H。零扩展指令 MOVZX 通常用于将 8 位数字转换为 16 位数字，或将 16 位数字转换为 32 位数字。

表 4-19 MOVSX 和 MOVZX 指令

汇编语言指令	操　作
MOVSX CX，BL	将 BL 中的内容符号扩展送入 CX 中
MOVSX ECX，AX	将 AX 中的内容符号扩展送入 ECX 中
MOVSX BX，DATA1	将 DATA1 单元中的字节内容符号扩展送入 BX 中
MOVSX EAX，[EDI]	将数据段由 EDI 寻址的存储单元中的字内容符号扩展送入 EAX 中
MOVSX RAX，[RDI]	将由 RDI 地址的双字内容符号扩展送入 RAX 中（64 位）
MOVZX DX，AL	将 AL 中的内容零扩展送入 DX 中
MOVZX EBP，DI	将 DI 中的内容零扩展送入 EBP 中
MOVZX DX，DATA2	将 DATA2 中的字节内容零扩展送入 DX 中
MOVZX EAX，DATA3	将 DATA3 中的字内容零扩展送入 EAX 中
MOVZX RBX，ECX	将 RBX 中的内容零扩展送入 RBX 中

当把数字的符号位复制到它的高位部分时，数字就被符号扩展了。例如，8 位的 84H 被符号扩展为 16 位数字，结果是 FF84H。84H 的符号位是 1，将 1 复制到最高位部分的结果就是 FF84H。我们常常用符号扩展指令 MOVSX 将 8 位有符号数转换为 16 位有符号数，或将 16 位有符号数转换成 32 位有符号数。

4.5.6 BSWAP 指令

字节交换指令（byte swap，BSWAP）只能用于 80486 ～ Pentium 4 微处理器，这条指令将 32 位寄存器内的第 1 字节与第 4 字节交换，第 2 字节与第 3 字节交换。例如，设 EAX = 00112233H，执行 BSWAP EAX 指令将 EAX 字节交换后，结果 EAX = 33221100H。注意，4 个字节的顺序被这条指令全部颠倒了。这条指令可以把由大到小格式的数据转换为由小到大的格式，反之亦然。

4.5.7 CMOV 指令

CMOV（conditional move，条件传送）指令是 Pentium Pro ～ Core2 指令系统的新指令。CMOV 指令有许多种，表 4-20 列出了这些 CMOV 指令。只有条件为真时，这些指令才传送数据。例如，CMOVZ 指令只有在前面的指令执行结果为零时才传送数据。这类指令的目的操作数只能是 16 位或 32 位寄存器，而源操作数可以是 16 位、32 位寄存器或者存储单元。

由于这是新指令，只有提供了 .686 开关时，汇编程序才可以使用它。

表 4-20 条件传送指令

汇编语言指令	测试的标志位	操　作
CMOVB	C = 1	低于则传送
CMOVAE	C = 0	高于或等于则传送
CMOVBE	Z = 1 或 C = 1	低于或等于则传送
CMOVA	Z = 0 和 C = 0	高于则传送
CMOVE 或 CMOVZ	Z = 1	等于或 Z = 1 则传送
CMOVNE 或 CMOVNZ	Z = 0	不等于或 Z = 0 则传送
CMOVL	S! = 0	小于则传送
CMOVLE	Z = 1 或 S! = 0	小于或等于则传送
CMOVG	Z = 0 和 S = 0	大于则传送
CMOVGE	S = 0	大于或等于则传送
CMOVS	S = 1	结果为负则传送
CMOVNS	S = 0	结果为正则传送
CMOVC	C = 1	有进位则传送
CMOVNC	C = 0	无进位则传送
CMOVO	O = 1	溢出则传送
CMOVNO	O = 0	无溢出则传送
CMOVP 或 CMOVPE	P = 1	有奇偶测试则传送或奇偶测试为偶则传送
CMOVNP 或 CMOVPO	P = 0	无奇偶测试则传送或奇偶测试为奇则传送

4.6 段超越前缀

段超越前缀（segment override prefix）可以附加到几乎任何指令的存储器寻址方式前，它允许程序设计者偏离默认的段。段超越前缀出现在指令前的附加字节上，以便选择代替的段寄存器。不能加前缀的指令只有转移和调用指令，它们必须用代码段寄存器生成的地址。在 80386 ～ Core2 微处理器中，段超越前缀也用来选择 FS 和 GS 段。

例如 MOV AX，[DI] 指令，默认的情况是访问数据段中的数据，如果程序要求，可以变成带段超越前缀的指令。假定数据是在附加段，而不是在数据段，将指令改成 MOV AX，ES:[DI]，则这条指令就可以寻址附加段。

表 4-21 给出了一些不寻址默认段，而寻址其他存储器段的替换指令。当指令附加了段超越前缀

时，指令就长了一个字节。虽然指令的长度没有明显的改变，但是增加了执行时间。为了使软件短小高效，通常尽量少使用段超越前缀。

表 4-21 包含有段超越前缀的指令

汇编语言指令	访问的段	默认段
MOV AX, DS: [BP]	数据段	堆栈段
MOV AX, ES: [BP]	附加段	堆栈段
MOV AX, SS: [DI]	堆栈段	数据段
MOV AX, CS: LIST	代码段	数据段
MOV ES: [SI], AX	附加段	数据段
LODS ES: DATA1	附加段	数据段
MOV EAX, FS: DATA2	FS 段	数据段
MOV GS: [ECX], BL	GS 段	数据段

4.7 汇编程序详述

微处理器的汇编程序[⊖]能够以两种方式使用：1）针对一种特定的汇编程序的模型；2）完整的段定义方式，可完全控制汇编的全过程，并且可用于所有的汇编程序。本节给出了这两种方法，说明了怎样使用汇编程序组织程序的存储器空间。也说明了用于汇编程序的一些更重要的伪指令的用法和作用。附录 A 提供了关于汇编程序详尽的叙述。

在很多情况下，使用 Visual C++ 中的内嵌汇编开发 C++ 程序中用的汇编代码，但有的场合要求使用汇编程序写成分开的汇编模块。在这一节将对比内嵌汇编和汇编的差别在哪里。

4.7.1 伪指令

讨论汇编语言程序的格式之前，先详细说明控制汇编处理的伪指令（**pseudo-operation，伪操作码**）。表 4-22 中列出了一些通用的汇编语言伪指令。**伪指令**（**directive**）指示汇编程序应怎样去处理操作数或一段程序，有些伪指令生成信息并将其存储到存储器中，而另一些则不。**DB**（**define byte，定义字节**）伪指令指示将数据字节存储到存储器中，而 BYTE PTR 伪指令则不存储数据。**BYTE PTR** 伪指令指明由指针或变址寄存器访问的数据的长度。注意，在 Visual C++ 的内嵌汇编程序部分没有伪指令功能，如果只使用内嵌汇编，可以跳过本书这一部分。要了解汇编的复杂细节，仍然要用 MASM 编写。

注意，在默认情况下，汇编程序只接受 8086/8088 指令，除非 .686 或 .686P 伪指令或者微处理器选择的其他的开关之一放在程序前面。.686 伪指令通知汇编程序按实模式使用 Pentium Pro 指令系统，而 .686P 通知汇编程序使用 Pentium Pro 保护模式指令系统。多数现代软件都是假定微处理器是 Pentium Pro 或者更新的微处理器，因此常常使用 .686 开关。Windows 95 是第一个依照 80386 体系结构的主要 32 位操作系统。Windows XP 要求一台 Pentium 类型的机器（.586 开关），至少使用 233MHz 微处理器。

表 4-22 MASM 常用的伪指令

伪指令	功能
.286	选择 80286 指令系统
.286P	选择 80286 保护模式指令系统
.386	选择 80386 指令系统
.386P	选择保护模式的 80386 指令系统
.486	选择 80486 指令系统
.486P	选择 80486 保护模式的指令系统
.586	选择 Pentium 指令系统
.586P	选择保护模式的 Pentium 指令系统

⊖ 本书使用的汇编程序为 Microsoft 公司的 MACRO 汇编程序 MASM，6.1X 版。

（续）

伪 指 令	功 能
.686	选择 Pentium Pro-Pentium 4 指令系统
.686P	选择 Pentium 保护模式的 Pro-Pentium 4 指令系统
.287	选择 80287 数字协处理器
.387	选择 80387 数字协处理器
.CODE	指示代码段的开始
.DATA	指示数据段的开始
.EXIT	返回到 DOS
.MODEL	选择编程模型
.STACK	选择堆栈段的开始（仅用于模型）
.STARTUP	在用编程模型时，指示程序的开始
ALIGN n	按字边界起始的数据（ALIGN4 选择以双字边界起始的数据）
ASSUME	在完整的段定义方式下，通知汇编程序每个段的名字
BYTE	指示数据的长度为字节，如在 BYTE PTR 中
DB	定义字节（8 位）数据
DD	定义双字（32 位）数据
DQ	定义 4 字（64 位）数据
DT	定义 10 个字节（80 位）数据
DUP	产生重复的字符或数字
DW	定义字（16 位）数据
DWORD	定义数据长度为双字，如 THIS DWORD 中
END	指示程序结束
ENDM	指示宏序列结束
ENDP	指示过程结束
ENDS	指示段或者数据结构结束
EQU	标号等于数据
FAR	定义一个远指针，如在 FAR PTR
MACRO	表明宏定义的开始
NEAR	定义近指针，如在 NEAR PTR
OFFSET	规定偏移地址
ORG	设置段的起始地址
OWORD	指示八进制字，如在 OWORD PTR
PROC	定义过程的开始
PTR	指示一个指针
QWORD	指示数据长度为四字，如在 QWORD PTR 中
SEGMENT	定义存储器段的起点
STACK	堆栈段开始
STRUC	指定数据结构的开始
USES	在过程中，自动将寄存器压栈和出栈
USE16	使用 16 位的指令模式
USE32	使用 32 位的指令模式
WORD	指示数据长度为字，如在 WORD PTR 中

在存储器段中存储数据

在第 1 章中出现的 **DB（define byte，定义字节）**、**DW（define word，定义字）** 和 **DD（define doubleword，定义双字）** 伪指令，经常用于 MASM 在存储器里定义和存储数据。如果系统中的数字协处理器执行软件，则也经常使用 **DQ（define quadword，定义四字）** 和 DT（**define ten byte，定义十个**

字节）伪指令。这些伪指令用符号名定义存储单元，并且指定其长度。

例4-13给出了一个存储器段，包括定义各种数据格式的伪指令。⊿给出了完整的段定义，第一个SEGMENT语句指示段的开始和它的符号名。当然，也可以像前面的例子一样，使用带.DATA语句的SMALL模型定义该段。这个例子最后的语句是ENDS伪指令，指示段结束。段名（LIST_SEG）可以是程序员所希望的任何名字。这就允许程序根据需要包含多个段。

例4-13

```
                        ;使用DB、DW和DD伪指令
                        ;
0000                    LIST_SEG SEGMENT

0000 01 02 03           DATA1   DB  1,2,3          ;定义字节
0003 45                         DB  45H            ;十六进制数
0004 41                         DB  'A'            ;ASCII码
0005 F0                         DB  11110000B      ;二进制数
0006 000C 000D          DATA2   DW  12,13          ;定义字
000A 0200                       DW  LIST1          ;符号
000C 2345                       DW  2345H          ;十六进制数
000E 00000300           DATA3   DD  300H           ;定义双字
0012 4007DF3B                   DD  2.123          ;实数
0016 544269E1                   DD  3.34E+12       ;实数
001A 00                 LISTA   DB  ?              ;保留一个字节
001B 00A[               LISTB   DB  10 DUP (?)     ;保留十个字节
         ??
           ]
0025 00                         ALIGN  2           ;设置字边界
0026 0100[              LISTC   DW  100H DUP (0)   ;保留100H个字
         0000
            ]
0226 0016[              LISTD   DD  22 DUP (?)     ;保留22个双字
         ????????
                 ]
027E 0064[              SIXES   DB  100 DUP (6)    ;保留100个字节
         06
           ]
02E2                    LIST_SEG      ENDS
```

例4-13在DATA1处定义了各种形式的字节数据。每行里可以按二进制、十六进制、十进制或ASCII码定义多个字节的数据。DATA2标号处指出了怎样存储各种格式的字数据。双字存储在DATA3处，包括了浮点单精度实数。

用“?”号作为DB、DW或DD伪指令的操作数，可以**保留**（**reserve**）一些存储单元，以便以后使用。当用“?”代替数字或ASCII值时，汇编程序保留这个单元，而不把它初始化为任何指定的值（实际上汇编程序是在指定为“?”的地方存入0）。**DUP**（**duplicate**，**重复**）伪指令用于建立数组。例4-13中给出了几种建立数组的方法，10 DUP（?）保留10个存储单元，但是不规定这10单元中存储的值。如果在DUP语句的（）内出现数字，汇编程序用指定的数字初始化保留的存储区。例如，DATA1 DB 10 DUP（2）伪指令，为数组DATA1保留10个字节，并且将每个存储单元初始化为02H。

这个例子中的ALIGN伪指令确认存储器数组按字边界存储。ALIGN 2按**字边界**存储数组，而ALIGN 4按**双字边界**存储数组。在Pentium～Pentium 4中，双精度浮点数的4字数据要采用ALIGN 8伪指令。*将字数据按字边界存储，而双字数据按双字边界存储，这是非常重要的。如果不这样做，微处理器要花费额外的时间访问这些数据。*访问存储在奇数存储器地址的字所花费的时间，相当于访问存储在偶数存储器地址的字所花费时间的两倍。注意，ALIGN伪指令不能用于存储器模型，因为模型的长

度确定了数据的对齐方式。如果先定义所有的双字，再定义字，最后定义字节数据，则不必再用ALIGN语句来对齐数据。

ASSUME、EQU 和 ORG

等于（EQU）伪指令把一个数值、ASCII 字符或者标号赋给另一个标号。EQU 使得程序更清晰并且简化了调试。例 4-14 给出了几个 EQU 语句和几条指令，说明了它们在程序中如何起作用。

例 4-14

```
                ;使用 EQU 伪指令
                ;
=000A           TEN   EQU 10
=0009           NINE  EQU 9

0000 B0 0A          MOV AL,TEN
0002 04 09          ADD AL,NINE
```

THIS 伪指令出现的形式总是 THIS BYTE、THIS WORD、THIS DWORD、THIS QWORD。在某些场合，数据必须既可按字访问又可按字节访问。汇编程序只能为标号分配字节地址、字地址或者双字地址。为了将字节标号分配给字，可以参见例 4-15。

例 4-15

```
                        ;使用THIS及ORG伪指令
                        ;
0000                    DATA_SEG      SEGMENT

0300                          ORG     300H

= 0300                  DATA1   EQU   THIS BYTE
0300                    DATA2   DW    ?
0302                    DATA_SEG      ENDS

0000                    CODE_SEG       SEGMENT 'CODE'
                              ASSUME CS:CODE_SEG, DS:DATA_SEG
0000 8A 1E 0300 R             MOV   BL,DATA1
0004 A1 0300 R                MOV   AX,DATA2
0007 8A 3E 0301 R             MOV   BH,DATA1+1

000B                    CODE_SEG      ENDS
```

这个例子也说明了怎样用 **ORG**（**origin**）语句将数据段中数据的起始偏移地址改变为 300H，有时必须用 ORG 语句给数据或代码的起始点分配一个绝对的偏移地址。**ASSUME** 语句通知汇编程序为代码段、数据段、附加段及堆栈段选择了什么名字。没有 ASSUME 语句时，汇编程序假定不分段，并且自动把段超越前缀用于所有寻址存储器数据的伪指令。ASSUME 语句只能用于完整的段定义，参见本节后面的说明。

PROC 和 ENDP

PROC 和 ENDP 伪指令指明过程（**子程序**）的开始和结束，这两条伪指令将过程强制结构化，因此清晰地定义了过程。如果不必结构化，可使用 CALLF、CALLN、RETF 和 RETN 伪指令。PROC 和 ENDP 两个伪指令都要求用标号指明过程的名字。指示过程开始的 PROC 伪指令的后面必须跟随 NEAR 和 FAR。NEAR 过程驻留在与程序相同的代码段中。FAR 过程可以驻留在存储系统的任何位置。我们认为调用 NEAR 过程是局部的调用，调用 FAR 过程是全局的调用。术语**全局**表示这个过程可以被任何程序使用，而**局部**过程只能被当前程序使用。在过程块内定义的全部标号也应被定义为局部的（NEAR）或者全局的（FAR）。

例 4-16 给出了将 BX、CX 和 DX 相加，结果存入 AX 寄存器的过程。尽管这个过程很短，并且可能不实用，但是它说明了怎样使用 PROC 和 ENDP 伪指令定义过程。注意，有关过程的功能信息应在注释中注明，指出被该过程和过程的结果改变的寄存器。

例 4-16

```
                ;BX、CX 和 DX 相加,结果存入寄存器 AX
                ;
0000            ADDEMPROC FAR       ;指示过程开始
0000 03 D9          ADD BX,CX
0002 03 DA          ADD BX,DX
0004 8B C3          MOV AX,BX
0006 CB             RET

0007            ADDEMENDP           ;指示过程结束
```

如果使用 Microsoft 公司的 MASM 6. X 版汇编程序，PROC 伪指令可自动保存过程中使用的任何寄存器。USES 语句指示过程使用的寄存器，使汇编程序在过程开始前就能自动的保存它们，而在过程用 RET 伪指令结束前恢复这些寄存器。例如 ADDS PROC USES AX BX CX 语句，在过程开始前自动将 AX、BX、CX 压入堆栈，而在过程末尾的 RET 指令执行前从堆栈中弹出它们。例 4-17 说明用 MASM 6. X 编写的使用 USES 语句的过程。注意，其中列出的寄存器不是用逗号而是用空格分开；因为已经用 . LISTALL 伪指令汇编过了，所以在过程表中显示了 PUSH 和 POP 指令。以 * 开头的指令是由汇编程序插入的，而不是由源文件输入的。本书另外的地方也出现有 USES 语句，如果使用 MASM 5. 10，则代码需要修改。

例 4-17

```
                                ;使用USES语句的过程。在过程开始前自动将过程中使用的AX、BX、CX压入堆栈,
                                ;而在过程末尾的RET指令执行前恢复它们
                                ;
0000                ADDS    PROC    NEAR    USES BX CX DX

0000 53         *           push    bx
0001 51         *           push    cx
0002 52         *           push    dx
0003 03 D8                  ADD     BX,AX
0005 03 CB                  ADD     CX,BX
0007 03 D1                  ADD     DX,CX
0009 8B C2                  MOV     AX,DX
                            RET
000B 5A         *           pop     dx
000C 59         *           pop     cx
000D 5B         *           pop     bx
000E C3         *           ret     0000h

000F                ADDS    ENDP
```

4. 7. 2 存储器组织

汇编程序使用两种基本格式开发软件，一种是使用模型，另一种是使用完整的段定义。这一节以及第 2 和第 3 章中简单描述的存储器模型只适用于 MASM 汇编程序。TASM 汇编程序也使用存储器模型，但是与 MASM 的模型有些区别。完整的段定义是多数汇编程序通用的，包括 Intel 的汇编程序，并且是软件开发时经常使用的。模型适用于简单的任务，而完整的段定义方式通常能较好地控制汇编语言任务，因此被推荐用于复杂的程序。由于模型很容易理解，它已经用在前面的章节中，模型也可以用于 C/C++ 高级语言使用的汇编子程序中。虽然作为程序设计例子本书充分研究并使用了存储器模型定义，但应承认完整段定义在某些方面更优于存储器模型，正如这一节后面讨论的那样。

模型

MASM 汇编程序可以使用从小到大多种模型。附录 A 中的表格列出了汇编语言可利用的所有模型。为了标记模型，使用 . MODEL 语句，后面跟随存储系统的长度。**TINY 模型**对于许多小的程序很适用，它要求将全部的软件和数据都安排在 64K 字节存储器段内。**SMALL 模型**要求只用一个数据段和一个代码段，总计占 128K 字节的存储器。还有其他一些模型，最大到 HUGE 型。

例4-18说明了.MODEL语句怎样定义短程序的参数。该程序将一个存储器块（LISTA）100个字节的内容复制到第二个存储器块（LISTB）。这个例子也指出怎样定义堆栈段、数据段和代码段。.EXIT 0伪指令返回DOS并且带有错误代码0（没有错误）。如果没有参数加到.EXIT上，仍然返回DOS，但是不定义错误代码。还要注意，如@DATA（见附录A）之类的特殊伪指令，用来确认各种段。如果用.STARTUP伪指令（MASM 6.X版），则MOV DS，AX语句前面的MOV AX，@DATA语句可以取消。.STARTUP伪指令也可以删除标号END后面的起始地址。如果汇编语言程序包含在C/C++程序中，模型对于Microsoft C/C++和Borland C/C++的开发系统都很重要。两种开发系统都采用内嵌汇编编程，以便加上汇编语言指令，并且要求理解程序设计模型。

例4-18

```
                              .MODEL SMALL          ;选择SMALL模型
                              .STACK 100H           ;定义堆栈段
                              .DATA                 ;定义数据段

0000 0064[          LISTA  DB    100 DUP(?)
          ??
            ]
0064 0064[          LISTB  DB    100 DUP(?)
          ??
            ]

                              .CODE                 ;定义代码段

0000 B9 ---- ?      HERE:  MOV   AX,@DATA         ;装入ES和DS
0003 8E C0                 MOV   ES,AX
0005 8E D8                 MOV   DS,AX
0007 FC                    CLD                   ;传送数据
0008 BE 0000 R             MOV   SI,OFFSET LISTA
000B BF 0064 R             MOV   DI,OFFSET LISTB
000E B9 0064               MOV   CX,100
0011 F3/A4                 REP   MOVSB

0013                       .EXIT 0               ;返回DOS
                           END HERE
```

完整段定义

例4-19说明用完整段定义的同一程序。完整段定义也用于Borland和Microsoft C/C++环境中用汇编语言设计的过程。例4-19中的程序比例4-18中的更长，但比模型方法建立的程序结构性更强。定义的第一个段是STACK_SEG，用SEGMENT和ENDS伪指令清晰的指出在这两个伪指令之间，DW 100H DUP (?)为堆栈段安排了100H个字。由于STACK出现在SEGMENT后面，汇编程序和连接程序自动加载堆栈段寄存器（SS）和栈指针（SP）。

例4-19

```
0000                STACK_SEG      SEGMENT       'STACK'
0000 0064[                 DW      100H DUP(?)
          ????
              ]
0200                STACK_SEG      ENDS

0000                DATA_SEG       SEGMENT       'DATA'
0000 0064[          LISTA  DB      100 DUP(?)
          ??
            ]
0064 0064[          LISTB  DB      100 DUP(?)
          ??
            ]
00CB                DATA_SEG       ENDS

0000                CODE_SEG       SEGMENT       'CODE'
                           ASSUME CS:CODE_SEG,DS:DATA_SEG
```

```
                        ASSUME SS:STACK_SEG
0000                MAIN    PROC    FAR
0000 B8 ---- R              MOV     AX,DATA_SEG             ;装入ES,和DS
0003 8E C0                  MOV     ES,AX
0005 8E D8                  MOV     DS,AX
0007 FC                     CLD                             ;移动数据
0008 BE 0000 R              MOV     SI,OFFSET LISTA
000B BF 0064 R              MOV     DI,OFFSET LISTB
000E B9 0064                MOV     CX,100
0011 F3/A4                  REP     MOVSB
0013 B4 4C                  MOV     AH,4CH                  ;返回DOS
0015 CD 21                  INT     21H
0017                MAIN    ENDP
0017                CODE_SEG        ENDS
                            END     MAIN
```

然后，在 DATA_SEG 中定义数据。程序中出现了两个数组 LISTA 和 LISTB，每个数组包含 100 个字节的存储空间。这个程序中段的名字可以改为任何名字，但是一定要包含组名‘DATA’，以便使 Microsoft 程序 CodeView 可以对这个软件进行符号化调试。CodeView 是 MASM 软件包的一部分。为了访问 CodeView 在 DOS 命令行下键入“CV”后面跟随文件名。如果由 Programmer's WorkBench 运行，选择 RUN 菜单中的 Debug。如果组名没有放在程序中，仍然可以使用 CodeView 调试程序，但是程序将不能以符号形式调试。附录 A 中列出了‘STACK’，‘CODE’等其他的组名。如果希望用 CodeView 观察符号化形式的程序，则必须把‘CODE’放在代码段 SEGMENT 语句的后面。

因为多数软件都是面向过程的，CODE_SEG 构造成了远过程。程序开始前，代码段安排了 ASSUME 语句。ASSUME 语句通知汇编程序和连接程序代码段（CS）使用的名字是 CODE_SEG。它也通知汇编程序和连接程序数据段是 DATA_SEG，而堆栈段是 STACK_SEG。注意，组名‘CODE’用于代码段是为了使用 CodeView。其他组名在附录 A 中与模型同时给出。

程序将数据段的地址装入附加段寄存器和数据段寄存器以后，从 LISTA 传送 100 字节到 LISTB。后面的两条指令控制返回 DOS（磁盘操作系统）。注意，加载程序不能自动初始化 DS 和 ES。必须将程序中说明的段地址装入这些寄存器。

程序中最后的语句是 END MAIN，END 语句指示程序结束和第一条可执行的指令的位置。这里我们希望机器执行 MAIN 过程，因此在 END 语句后面有一个标号。

在 80386 ～ Core2 微处理器中，代码段中要附加一个伪指令 USE16 或 USE32，该伪指令通知汇编程序，令微处理器使用 16 位或 32 位指令模式。为 DOS 环境开发的软件必须使用 USE16 伪指令，以便使程序在 80386 ～ Core2 中正确运行。因为这时 MASM 假定默认的所有的段是 32 位的，全部的指令模式是 32 位的。

4.7.3 程序举例

例 4-20 提供了使用完整段定义的例子程序，该程序从键盘读字符并将它显示在 CRT 屏幕上。虽然这个程序很普通，但它是一个完全可以工作的程序，它可以在任何使用 DOS 的 PC 上运行，从早期的 8088 系统到基于 Core2 的系统。这个程序也说明了几个 DOS 功能调用的使用（附录 A 列出了 DOS 功能调用和它们的参数）。BIOS 功能调用能够用于键盘、打印机、盘驱动器和计算机系统中使用的任何设备。

例 4-20

```
                ;DOS 完整段定义的例子，该程序从键盘读字符并且在屏幕上显示它。
                ;注意，以符号@结束程序。
                ;
0000            CODE_SEG        SEGMENT 'CODE'
                      ASSUME    CS: CODE_SEG

0000            MAIN  PROC      FAR
```

```
0000  B4  06          MOV   AH,06H          ;读键盘输入
0002  B2  FF          MOV   DL,0FFH
0004  CD  21          INT   21H
0006  74  F8          JE    MAIN            ;如果没有键按下
0008  3C  40          CMP   AL,'@'          ; 测试是否是@
000A  74  08          JE    MAIN1           ;如果是@
000C  B4  06          MOV   AH,06H          ;显示键盘输入的字符
000E  8A  00          MOV   DL,AL
0010  CD  21          INT   21H
0012  EB  EC          JMP   MAIN            ;重复读键
0014            MAIN1:
0014  B4  4C          MOV   AH,4CH          ;返回 DOS
0016  CD  21          INT   21H

0018            MAIN  ENDP
0018                  END   MAIN
```

这个例子只用了代码段，因为它没有数据。应该出现堆栈段，但是省去了，因为 DOS 自动为所有的程序分配了 128 字节的堆栈。这个例子只为调用 DOS 过程的 **INT 21H** 指令使用了一次堆栈。连接这个程序时，连接程序给出没有堆栈出现的警告信号。这个警告在本例中可以忽略，因为它只需小于 128 字节的堆栈。

注意，整个程序放在称为 MAIN 的远过程中。这是一个好的程序设计的实例，全部软件按照过程的格式编写。如果将来需要时，允许把这个程序作为过程来调用。程序头部的信息也非常重要，它记录程序使用的寄存器和所需要的参数，它是在程序开头出现的注释。

程序使用了 DOS 功能调用 **06H** 和 **4CH**。在 **INT 21H** 指令执行前，功能号先放入 AH 中。如果 **DL =0FFH**，**06H** 功能调用读键盘。如果 DL 不等于 0FFH，则 06H 功能调用显示 DL 中的 ASCII 内容。测试点之前，程序的第一部分将 06H 放入 AH，0FFH 送入 DL，因此是从键盘读键。INT 21H 测试键盘，如果没有键输入，就按等于条件返回。JE 指令测试等于条件，如果没有键输入时，则跳转到 MAIN。

如果有键输入时，程序继续下一步。这一步是比较 AL 的内容与符号@。因为从 INT 21H 返回时，输入的 ASCII 码字符放在 AL 中。在这个例子中，如果键入符号@，程序结束。如果键入的不是符号@，则程序继续执行下一条 INT 21H 指令显示键盘输入的字符。

第 2 条 INT 21H 指令将 ASCII 字符移入 DL，这样就可以在 CRT 屏幕上显示。显示字符以后执行 JMP 指令。使程序在 MAIN 处继续，重复读键盘。

如果输入字符号@，程序在 MAIN1 处继续，执行 4CH 号 DOS 功能调用。程序返回 DOS 提示符，使得计算机可以处理其他任务。

关于汇编程序及其应用的更多的信息参考附录 A 和后面的几章。附录 A 提供了汇编程序、连接程序和 DOS 功能调用的完整综述，也提供了 BIOS（基本输入/输出系统）功能调用表。后面章节提供的信息阐明了怎样用汇编程序完成一个指定的任务。

例 4-21

```
                ;用 DOS 模型编程的例子，该程序从键盘读入字符并且在屏幕上显示它。
                ;注意，以符号@结束程序。
                ;
                .MODEL TINY
0000            .CODE
                .STARTUP

0100            MAIN:
0100  B4  06          MOV   AH,6            ; 读键盘输入
0102  B2  FF          MOV   DL,0FFH
0104  CD  21          INT   21H
```

```
0106 74 F8          JE    MAIN          ;如果没有键按下
0108 3C 40          CMP   AL,'@'        ;测试是否是@
010A 74 08          JE    MAIN1         ;如果是@
010C B4 06          MOV   AH,06H        ;显示键盘输入的字符
010E 8A D0          MOV   DL,AL
0110 CD 21          INT   21H
0112 EB EC          JMP   MAIN          ;重复读键
0114          MAIN1:
              .EXIT                     ;返回 DOS
              END
```

例4-21给出在例4-20列出过的程序，只是用模型而不用完整段描述。请比较两个程序的差别。注意看按模型写程序多么简短而且清晰。

4.8 小结

1）数据传送指令在两寄存器之间，寄存器与存储器之间，寄存器与堆栈之间，存储器与堆栈之间，累加器与I/O端口之间，标志寄存器与堆栈之间传送数据。存储器到存储器的传送只允许用MOVS指令。

2）数据传送指令包括MOV、PUSH、POP、XCHG、XLAT、IN、OUT、LEA、LDS、LES、LSS、LGS、LFS、LAHF、SAHF和串操作指令：LODS、STOS、MOVS、INS和OUTS。

3）指令的第一字节存放操作码。操作码规定微处理器执行的操作。有些指令的操作码前可以有一个或多个超越前缀。

4）许多指令中的D位用于选择数据流的方向。如果D=0，数据从REG字段流向指令的R/M字段；如果D=1，则数据从R/M字段流向REG字段。

5）大多数指令中的W位用于选择数据的长度。如果W=0，数据是字节长度的；如果W=1，数据是字长度的。在80386及更高型号的微处理器中，W=1，指定字或者双字的寄存器。

6）MOD为机器语言指令的R/M字段选择寻址方式。如果MOD=00，表示没有位移量；如果MOD=01，有8位符号扩展的位移量；如果MOD=10，则有16位的位移量；如果MOD=11，则操作数为寄存器而不是存储单元。在80386及更高型号的微处理器中，MOD也可规定用32位的位移量。

7）当MOD=11时，用3位二进制编码指定REG和R/M字段。8位寄存器是AH、AL、BH、BL、CH、CL、DH和DL，16位寄存器是AX、BX、CX、DX、SP、BP、DI和SI，32位寄存器是EAX、EBX、ECX、EDX、ESP、EBP、EDI和ESI。为了访问64位寄存器，需要添加一个称为REX前缀的新前缀，其中的第4位用于访问寄存器R8～R15。

8）当R/M字段定义为存储器寻址方式时，在16位指令中3位二进制码可选择下面的方式之一：[BX+DI]、[BX+SI]、[BP+DI]、[BP+SI]、[BX]、[BP]、[DI]或[SI]。在80386及更高型号的微处理器中，R/M字段指定EAX、EBX、ECX、EDX、EBP、EDI、ESI或寻址存储器数据的比例变址方式之一。如果选择了比例变址方式（R/M=100），则加到指令上的附加字节（比例因子变址字节）指定基址寄存器、变址寄存器及比例因子。

9）除了BP或EBP寻址存储器以外，所有存储器寻址方式都默认寻址数据段中的数据。BP或EBP寄存器寻址堆栈段中的数据。

10）段寄存器只能通过MOV、PUSH或POP指令访问。MOV指令可以将段寄存器的内容传送到16位寄存器，反之亦然。MOV CS，reg或POP CS指令是不允许的，因为这些指令只改变了指令地址的一部分。在80386～Pentium 4中有两个附加的段寄存器：FS和GS。

11）在寄存器与堆栈之间或存储单元与堆栈之间通过PUSH和POP指令传送数据。这些指令允许立即数压入堆栈，允许在标志寄存器与堆栈之间传送，以及允许在堆栈与寄存器之间传送全部16位通用寄存器。当数据传送到堆栈时，一次总是传送两个字节（8086～80286），高字节放入SP-1地址单元，而低字节放入SP-2地址单元。数据存入堆栈以后，SP内容减2。在80386～Core2中，来自存储单元或寄存器的4字节数据也可以传送到堆栈。

12）在堆栈和标志寄存器之间传送数据的操作码是PUSHF和POPF。在堆栈和寄存器之间传送全部16位寄存器的操作码是PUSHA和POPA。在80386和更高档型号的微处理器中，PUSHFD和POPFD在微处理器和堆栈之间传送EFLAGS的内容，PUSHAD和POPAD传送全部32位寄存器。在64位模式中PUSHA和POPA指令是无效的。

13）LEA、LDS和LES指令将有效地址装入一个寄存器或两个寄存器。LEA指令将有效地址装入任一16位寄存器。而LDS和LES将有效地址装入任一16位寄存器及DS或ES。在80386及更高型号的微处理器中，附加指令LFS、LGS和LSS用于加载一个16位寄存器和FS、GS或SS。

14）串数据传送指令使用 DI 及/或 SI 寻址存储器。DI 偏移地址定位在附加段，而 SI 偏移地址位于数据段。如果 80386 ~ Core2 工作于保护模式，ESI 和 EDI 也用于串操作指令。

15）方向标志（D）为用于串操作指令的 DI 或 SI 选择自动增量或自动减量操作方式。为了清除 D 为 0，使用 CLD 指令，以便选择自动增量方式；为了设置 D 为 1，使用 STD 指令，以便选择减量方式；对于字节操作 DI 和/或 SI 增 1 或减 1，字操作为增 2 或减 2，而双字操作为增 4 或减 4。

16）LODS 指令将由 SI 寻址的存储单元的数据装入 AL、AX 或 EAX。STOS 将 AL、AX 或 EAX 的内容存入由 DI 寻址的存储单元。MOVS 指令将 SI 寻址的存储单元的字节或者字传送到由 DI 寻址的存储单元。

17）INS 指令输入由 DX 寻址的 I/O 设备的数据，将它存入由 DI 寻址的存储单元。OUTS 指令输出由 SI 寻址的存储单元的内容，将它输出到由 DX 寻址的 I/O 设备。

18）REP 前缀可以附加到任何串指令上，以便重复执行该指令。REP 前缀重复串指令的次数放在寄存器 CX 中。

19）在汇编语言中可以使用算术和逻辑运算。例如 MOV AX，34 * 3，功能是将 102 装入 AX。

20）换码（XLAT）指令将 AL 中的数据转换为存储在由 BX 加 AL 寻址的存储单元中的数字。

21）IN 和 OUT 在 AL、AX 或 EAX 与外部 I/O 设备之间传送数据。I/O 设备的地址存储在指令中（固定端口）或寄存器 DX 中（可变端口）。

22）条件传送指令 CMOV，是 Pentium Pro ~ Core2 中包含的新指令。只有条件为真时，这条指令才执行传送。

23）段超越前缀为存储单元选择一个有别于默认段的段寄存器。例如，MOV AX，［BX］指令使用数据段，但是 MOV AX，ES:［BX］指令因为有前缀 ES:而使用附加段。只有 80386 ~ Pentium 4 中有寻址 FS 和 GS 段的段超越前缀。

24）80386 及更高型号的微处理器中的 MOVZX（传送和零扩展）及 MOVSX（传送和符号扩展）指令，将字节长度增加到字或者字长度增加到双字。零扩展指令通过将零填充到高位来增加数据的长度，符号扩展指令通过将符号位复制到数据的高有效位来增加数据的长度。

25）汇编伪指令 DB（定义字节）、DW（定义字）、DD（定义双字）和 DUP（重复的）用于在存储系统中存储数据。

26）EQU（等于）伪指令允许标号等于一个数据或者另一个标号。

27）使用完整的段定义时，SEGMENT 伪指令指示存储器段的开始，而 ENDS 伪指令指示段的结束。

28）当完整的段定义成为事实时，ASSUME 伪指令通知汇编程序已经给 CS、DS、ES 和 SS 选定了段名。在 80386 和更高档型号的微处理器中，ASSUME 也为 FS 和 GS 指定段名。

29）PROC 和 ENDP 伪指令指示过程的开始和结束。如果 USES 伪指令（MASM 6. X 版）与 PROC 伪指令一起出现时，能够将任何数量的寄存器自动保存和恢复。

30）汇编程序假定软件是为 8086/8088 微处理器开发的，但是可以用 . 286、. 386、. 486、. 586 或 . 686 伪指令选择某个其他类型的微处理器。这些伪指令跟随在 . MODEL 语句后面为 16 位指令模式，而放在它前面为 32 位指令模式。

31）存储器模型可以用于短程序，但是对于非常大的程序可能会引起问题。还要注意各种汇编程序中的存储器模型互不兼容。

4.9 习题

1. 若指令不包含超越前缀，指令的第一字节是_____。
2. 说明某些机器语言指令中的 D 位和 W 位的作用。
3. 机器语言指令中，MOD 字段的含义是什么？
4. 假定指令是 16 位模式指令，如果指令寄存器字段（REG）的内容是 010 而且 W = 0，选择哪个寄存器？
5. 怎样为 Pentium 4 微处理器选择 32 位寄存器？
6. 若 R/M = 001，MOD = 00，为 16 位指令指定了哪种存储器寻址方式？
7. 说明分配给下列寄存器的默认段寄存器。
   （a）SP
   （b）EBX
   （c）DI
   （d）EBP
   （e）SI
8. 将机器语言 8B07H 翻译为汇编语言。
9. 将机器语言 8B9E004CH 转换为汇编语言。
10. 如果 MOV SI，［BX + 2］指令出现在程序中，与它等效的机器语言是什么？
11. 如果一个 MOV ESI，［EAX］指令出现在工作于 16 位指令模式的 Core2 微处理器的程序中，它对应的机器语言是什么？
12. REX 的目的是什么？
13. MOV CS，AX 指令会带来什么错误？
14. 设计一个短指令序列，将 1000H 装入数据段寄存器。
15. 80386 ~ Core2 微处理器中 PUSH 和 POP 指令在堆栈与寄存器或存储单元之间总是传送_____位数字。
16. 创建一条指令用于 64 位的 Pentium 4 中确定 RAX 在堆栈中的位置。
17. 不能从堆栈向哪个段寄存器弹出数据？
18. PUSHA 指令将哪些寄存器压入堆栈？
19. PUSHAD 指令将哪些寄存器压入堆栈？
20. 说明下面每条指令的操作：

(a) PUSH AX
(b) POP ESI
(c) PUSH [BX]
(d) PUSHFD
(e) POP DS
(f) PUSHD 4

21. 说明 PUSH BX 指令执行时会发生什么操作？假设 SP = 0100H，SS = 0200H，确切指出 BH 和 BL 分别存储在哪个存储单元中？
22. 对于 PUSH EAX 指令重复回答习题 19。
23. 16 位 POP 指令（POPA 除外）将 SP 加_____。
24. 如果堆栈中的存储器单元 02200H 被访问，SP 和 SS 中装入什么值？
25. 比较 MOV DI，NUMB 指令和 LEA DI，NUMB 指令的操作。
26. LEA SI，NUMB 指令和 MOV SI，OFFSET NUMB 指令之间的区别是什么？
27. 带有 OFFSET 的 MOV 指令与 LEA 指令比较，哪条指令效率更高？
28. 描述 LDS BX，NUMB 指令怎样操作？
29. LDS 指令与 LSS 指令有什么区别？
30. 设计指令序列，传送数据段存储单元 NUMB 和 NUMB + 1 的内容到 BX、DX 和 SI 中。
31. 方向标志的作用是什么？
32. 哪些指令设置和清除方向标志？
33. 串指令用 DI 和 SI 寻址哪个存储器段中的数据？
34. 说明 LODSB 指令的操作。
35. 说明 64 位的 Pentium 4 或 Core2 的 LODSQ 指令的操作。
36. 说明 OUTSB 指令的操作。
37. 说明 STOSW 指令的操作。
38. 设计指令序列，将 12 个字节的数据由 SOURCE 寻址的存储区域复制到由 DEST 寻址的存储区域内。
39. REP 前缀的作用是什么，什么类型的指令与它一起使用？
40. 选择一条交换 EBX 寄存器与 ESI 寄存器内容的汇编语言指令。
41. 对于 INSB 指令，I/O 地址（端口号）存储在哪里？
42. 在软件中常使用 LAHF 指令和 SAHF 指令吗？
43. 写一个短程序，用 XLAT 指令将 BCD 码数字 0 ~ 9 转换为 ASCII 数字 30H ~ 39H。ASCII 代码存入数据段中的 TABLE 表中。
44. 说明 XLAT 指令怎样转换 AL 寄存器中的内容。
45. 说明 IN AL，12H 指令实现什么功能。
46. 说明 OUT DX，AX 指令是怎样操作的。
47. 什么是段超越前缀？
48. 选择一条指令，将附加段中用 BX 寻址的存储单元中的字节数据传送到 AH 寄存器中。
49. 设计指令序列，交换 AX 与 BX，ECX 与 EDX 和 SI 与 DI 之间的数据。
50. 什么是汇编语言伪指令？
51. Pentium 4 微处理器中 CMOVNE CX，DX 指令实现什么操作？
52. 说明下列汇编语言伪指令的作用：DB、DW 和 DD。
53. 选择汇编语言伪指令，为 LIST1 数组保留 30 个字节存储单元。
54. 说明 EQU 伪指令的作用。
55. 说明 .686 伪指令的作用。
56. 说明 .MODEL 伪指令的作用。
57. 如果用 .DATA 定义段的开始，那么存储器组织是什么类型的？
58. 如果 SEGMENT 伪指令定义段的开始，事实上是什么类型的存储器组织？
59. 如果 AH 内容是 4CH，则 INT 21H 实现什么功能？
60. 什么伪指令指示过程的开始和结束？
61. USE 语句用于 MASM 6.X 版的过程时，说明它的作用。
62. 设计近过程，将 AL 的内容存入数据段中用 DI 寄存器寻址的 4 个连续的存储单元中。
63. 应当怎样指示 Pentium 4 微处理器使用 16 位指令模式？
64. 设计远过程将存储单元 CS：DATA4 中的字内容复制到 AX、BX、CX、DX 和 SI 寄存器。

第5章　算术和逻辑运算指令

引言

这一章讨论算术和逻辑运算指令。算术运算指令包括加、减、乘、除、比较、求补、加1和减1指令。逻辑运算指令包括AND、OR、XOR、NOT、移位、循环和逻辑比较（TEST）指令。还给出了80386～Core2的XADD、SHRD、SHLD、位测试和位搜索指令。本章还介绍了串比较指令，这些指令用来搜索表中的数据和比较两个数据存储区域的数据。使用串搜索（SCAS）和串比较（CMPS）指令有效地实现了这两个任务。

如果熟悉8位微处理器，就会发现8086～Core2的指令系统比8位微处理器更好，因为大多数指令有两个操作数而不是一个。即使这是你第一个接触的微处理器，也会很快认识到这类微处理器的算术和逻辑运算指令系统功能强，并且容易使用。

目的

读者学习完本章后将能够做到：

1）用算术指令和逻辑指令完成简单的二进制、BCD和ASCII算术运算。

2）用AND、OR和XOR实现二进制位操作。

3）使用移位指令和循环指令。

4）解释80386～Core2的交换加法、比较交换、双精度移位、位测试和位搜索指令的操作。

5）用串指令查找表中的匹配项。

5.1　加法、减法和比较指令

任何微处理器的算术运算指令都包括加法、减法和比较指令。在本节将说明加法、减法和比较指令，并给出它们在处理寄存器和存储数据方面的应用。

5.1.1　加法指令

在微处理器中加法（ADD）指令以多种形式出现。本节详细叙述了用于8位、16位和32位二进制加法的ADD指令，并且还讲解了另一种形式的加法指令，即**带进位的加法指令ADC（add with-carry）**。最后，本节介绍加1指令（INC），这是特殊类型的加法，使某个数加1。在5.3节中说明其他形式的加法，诸如BCD和ASCII加法，以及80486～Pentium 4中的XADD指令。

表5-1说明了ADD指令提供的寻址方式（这些寻址方式几乎全是在第3章中叙述过的）。然而，因为ADD指令有超过32 000多种变形，这个表不可能全部列出它们。不允许的是存储器与存储器的加法，以及与段寄存器相关的加法。段寄存器只能被传送、压栈或出栈。注意，和其他指令一样，32位寄存器只能用于80386～Core2微处理器。在Pentium 4和Core2的64位模式中同样可以使用64位寄存器。

表5-1　加法指令部分寻址方式举例

汇编语言指令	操　作
ADD AL, BL	AL = AL + BL
ADD CX, DI	CX = CX + DI
ADD EBP, EAX	EBP = EBP + EAX
ADD CL, 44H	CL = CL + 44H
ADD BX, 245FH	BX = BX + 245FH
ADD EDX, 12345H	EDX = EDX + 12345H
ADD [BX], AL	AL加数据段由BX寻址的存储单元的内容，结果存入这个存储单元

（续）

汇编语言指令	操　作
ADD CL, [BP]	用 BP 作为偏移地址寻址的堆栈段存储单元的内容加 CL，结果存入 CL
ADD AL, [EBX]	用 EBX 作为偏移地址数据段寻址的存储单元的内容加 AL，结果存入 AL
ADD BX, [SI+2]	用 SI+2 寻址的数据段存储单元的单字长度的内容加 BX，结果存入 BX
ADD CL, TEMP	数据段 TEMP 存储单元的字节内容加 CL，结果存入 CL
ADD BX, TEMP [DI]	由 TEMP+DI 寻址的数据段存储单元的字内容加 BX，结果存入 BX
ADD [BX+DI], DL	将 DL 的内容与由 BX+DI 寻址的存储单元的字节内容相加，结果存入同一存储单元
ADD BYTE PTR [DI], 3	把 3 加到数据段中的 DI 寻址的存储单元的字节内
ADD BX, [EAX+2*ECX]	用 2 倍 ECX 加 EAX 之和寻址的数据段存储单元的字加 BX，结果存入 BX
ADD RAX, RBX	将 RBX 和 RAX 的内容相加，结果存入 RAX 中
ADD EDX, [RAX+RCX]	将 RAX+RCX 寻址的存储单元的双字内容相加，结果存入 EDX

寄存器加法

例 5-1 给出一个寄存器加法的简单程序，把几个寄存器内容加起来。在这个例子中，AX、BX、CX 和 DX 的内容累加，形成 16 位的结果并将其存入 AX 寄存器。

例 5-1

```
0000  03  C3        ADD  AX,BX
0002  03  C1        ADD  AX,CX
0004  03  C2        ADD  AX,DX
```

每次执行算术和逻辑运算指令总要改变标志寄存器的内容。注意，在算术和逻辑运算期间，中断、陷阱和其他一些标志不改变，只改变标志寄存器的最右边 8 位和溢出位。最右边这些标志指示算术和逻辑运算的结果。任何 ADD 指令都修改符号、零、进位、辅助进位、奇偶和溢出标志，在第 4 章中讲的数据传送指令不会改变标志位。

立即数加法

当常数或已知数相加时总是使用立即数加法。例 5-2 中给出了 8 位立即数加法。通过立即数传送指令将 12H 装入 DL，然后使用立即数加法指令将 33H 加到 DL 中的 12H 上。加完以后，和数（45H）放在 DL 中，标志位改变如下：

Z=0（结果非零）
C=0（没有进位）
A=0（没有半进位）
S=0（结果为正）
P=0（奇偶性为奇）
O=0（没有溢出）

例 5-2

```
0000  B2  12        MOV  DL,12H
0002  80  C2  33    ADD  DL,33H
```

存储器与寄存器的加法

假定应用程序要求存储器数据加到 AL 寄存器中。例 5-3 给出了一个程序，将存储在数据段偏移地址 NUMB 和 NUMB+1 处两个连续单元的字节数据累加到 AL 寄存器。

例 5-3

```
0000  BF  0000  R    MOV  DI,OFFSET NUMB    ;地址 NUMB
0003  B0  00         MOV  AL,0               ;清除和数
```

```
0005 02 05          ADD  AL,[DI]          ;加 NUMB
0007 02 45 01       ADD  AL,[DI+1]        ;加 NUMB+1
```

首先将 NUMB 的偏移地址装入目标变址寄存器（DI），这个例子用 DI 寄存器寻址数据段中从存储器地址 NUMB 开始的数据。在把和清除为 0 之后，ADD AL，[DI] 指令把存储单元 NUMB 的内容加到 AL 中。最后，ADD AL，[DI+1] 指令，把 NUMB+1 存储单元的内容加到 AL 寄存器。执行两条 ADD 指令以后，NUMB 内容加 NUMB+1 内容的结果出现在 AL 寄存器中。

数组加法

存储器数组是顺序排列的数据表。假定数据数组（ARRAY）包括从元素 0 到元素 9 共 10 个字节数。例 5-4 给出如何将元素 3、元素 5 和元素 7 内容累加。

例 5-4

```
0000 B0 00           MOV  AL,0              ;和清 0
0002 BE 0003         MOV  SI,3              ;指向元素 3
0005 02 84 0000 R    ADD  AL,ARRAY[SI]      ;加元素 3
0009 02 84 0002 R    ADD  AL,ARRAY[SI+2]    ;加元素 5
000D 02 84 0004 R    ADD  AL,ARRAY[SI+4]    ;加元素 7
```

这个例子首先将 AL 清 0，这样它才可以用来累加求和。然后，把 3 装入寄存器 SI，初始化为寻址数组元素 3。ADD AL，ARRAY［SI］指令，累加数组元素 3 到 AL 中。随后的指令累加数组元素 5 和元素 7 到 AL 中，用 SI 中的 3 加位移量 2 寻址元素 5，加位移量 4 寻址元素 7。

假定数组元素为 16 位数，要用数组元素来形成 16 位的和并存于寄存器 AX 中。例 5-5 是为 80386 及更高档微处理器写的指令序列。用比例变址的寻址方式，求 ARRAY 存储区的元素 3、元素 5 和元素 7 的累加和。这个例子把地址 ARRAY 装入 EBX 中，在 ECX 中保存数组元素的序号。注意如何使用比例因子将 ECX 寄存器的内容乘以 2 来寻址字数据（一个字是 2 个字节长）。

例 5-5

```
0000 66 | BB 00000000 R    MOV  EBX,OFFSET ARRAY    ;地址 ARRAY
0006 66 | B9 00000003      MOV  ECX,3               ;元素 3 的地址
000C 67 &8B 04 4B          MOV  AX,[EBX+2*ECX]      ;得到元素 3
0010 66 | B9 00000005      MOV  ECX,5               ;元素 5 的地址
0016 67 &03 04 4B          ADD  AX,[EBX+2*ECX]      ;加元素 5
001A 66 | B0 00000007      MOV  ECX,7               ;元素 7 的地址
0020 67 &03 04 4B          ADD  AX,[EBX+2*ECX]      ;加元素 7
```

加 1 指令

加 1 指令（INC）使寄存器或存储单元内容加 1。除了段寄存器以外，INC 指令可使任何寄存器或存储单元加 1。表 5-2 说明了可用于 8086 ～ Core2 微处理器的一些加 1 指令的可能格式。但这里不可能给出全部样式的 INC 指令，因为数目太多。

表 5-2 加 1 指令

汇编语言指令	操　作
INC BL	BL = BL + 1
INC SP	SP = SP + 1
INC EAX	EAX = EAX + 1
INC BYTE PTR［BX］	数据段中由 BX 寻址的存储单元的字节内容加 1
INC WORD PTR［SI］	数据段中由 SI 寻址的存储单元的字内容加 1
INC DWORD PTR［ECX］	数据段中由 ECX 寻址的存储单元的双字的内容加 1
INC DATA1	数据段 DATA1 单元的内容加 1
INC RCX	RCX = RCX + 1（64 位）

对于用间接寻址存储器的加1指令，数据的长度必须用BYTE PTR、WORD PTR或DWORD PTR伪指令说明。因为汇编程序不能确定是对字节、字还是双字加1，如例子中的INC［DI］指令。INC BYTE PTR［DI］指令清楚地指明是字节型数据加1，INC WORD PTR［DI］指令指明是字型数据加1，而INC DWORD PTR［DI］指令指明是双字型数据加1。在Pentium 4和Core2的64位模式操作中，INC QWORD PTR［RSI］指令指明是四字型数据加1。

例5-6说明了如何修改例5-3，用加1指令寻址NUMB和NUMB+1。这里，INC DI指令使DI寄存器的内容由偏移地址NUMB变为NUMB+1。例5-3和5-6两者都是NUMB和NUMB+1的内容相加，两个程序之间的区别是DI寄存器内容形成数据地址的方式的不同，后者使用了加1指令。

例5-6

```
0000  BF 0000 R       MOV  DI,OFFSET NUMB    ;寻址 NUMB
0003  B0 00           MOV  AL,0              ;将 AL 清 0
0005  02 05           ADD  AL,[DI]           ;加 NUMB
0007  47              INC  DI                ;DI 加 1
0008  02 05           ADD  AL,[DI]           ;加 NUMB +1
```

加1指令像多数的其他算术和逻辑运算那样影响标志位。不同的是加1指令不影响进位位。不改变进位位是因为我们常常根据程序中进位位的内容使用加1指令。注意，加1指令只用在字节型数据数组中用来指向下一个存储单元。如果寻址字型数据，最好用ADD DI，2指令修改指针DI，而不要用两次INC DI指令。对于双字数组，用ADD DI，4指令修改指针DI。但某些情况下必须保护进位位，这意味着为修改指针在程序中只能连续用2个或4个INC指令。

带进位的加法指令

带进位加法指令（ADC）把进位标志（C）加到操作数中。这条指令主要用在8086～80286中对宽于16位的数字相加，或80386～Core2微处理器中对宽于32位数字相加的软件中。

表5-3列出了几个带进位位的加法指令，说明了它们的操作。像ADD指令一样，ADC加法完成后也影响标志。

表5-3 带进位加法指令

汇编语言指令	操　作
ADC AL，AH	AL = AL + AH + C（进位位）
ADC CX，BX	CX = CX + BX + C
ADC EBX，EDX	EBX = EBX + EDX + C
ADC RBX，0	RBX = RBX + 0 + C（64位）
ADC DH，［BX］	由BX寻址的数据段存储单元的字节内容加DH和进位位，结果存入DH
ADC BX，［BP+2］	由BP加2寻址的堆栈段存储单元的字内容加BX和进位位，结果存入BX
ADC ECX，［EBX］	由EBX寻址的数据段存储单元的内容加ECX和进位位，结果存入ECX

假定一个为8086～80286写的程序实现BX和AX中的32位数字加DX和CX中的32位数字。图5-1说明了这个加法，清楚地说明了进位标志位的位置和功能。这类加法不能简单地用没有进位的加法指令实现，因为8086～80286只能对8位或16位数字相加。例5-7说明寄存器AX和CX的内容相加形成和的低16位，这个加法可能产生也可能不会产生进位。如果和大于FFFFH，进位标志中将出现进位。因为事先不可能断定有无进位，所以高16位加法要采用带进位的加法指令ADC。ADC指令把进位标志1或0加到高16位的和上。这个程序将BX-AX和

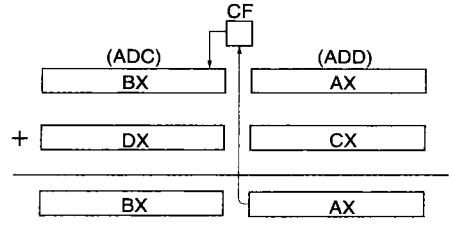


图5-1 带进位位的加法，指出了进位标志（C）怎样将两个16位加法连接为32位加法

DX-CX 相加，而和出现在 BX-AX 中。

例 5-7

```
0000 03 C1      ADD  AX,CX
0002 13 DA      ADC  BX,DX
```

假设为 80386 ～ Core2 微处理器重写这个软件，在 32 位中要修改为两个 64 位数字相加。这就要求修改指令，要用扩展寄存器保存数据。这种改变如例 5-8 所示，它实现了两个 64 位数相加。在 Pentium 4 和 Core2 的 64 位模式中，如果操作数的存储单元换成 RAX 和 RBX，那么加法用单一 ADD 指令处理，如指令 ADD RAX，RBX，将 RBX 加到 RAX 中。

例 5-8

```
0000 66 | 03 C1      ADD  EAX,ECX
0003 66 | 13 DA      ADC  EBX,EDX
```

用于 80486 ～ Core2 微处理器的交换并相加

称为**交换并相加**（**exchange and add**，**XADD**）的新型加法出现在 80486 ～ Core2 指令系统中。与所有加法类似，XADD 指令把源操作数加到目的操作数上，而和存入目的操作数中。不同的是加法以后操作数占用的位置，目的操作数原始值被复制到源操作数中。这是几个改变源操作数的指令之一。

例如，如果 BL = 12H 及 DL = 02H，执行了 XADD BL，DL 指令以后，BL 寄存器有和数 14H，而 DL 变成了 12H。14H 是产生的和，而原来目标里的 12H 代替了源操作数。这条指令如同 ADD 指令一样，它可以使用各种长度的寄存器和存储器操作数。

5.1.2 减法指令

指令系统有多种形式的减法指令（SUB），这些指令可以使用 8 位、16 位或 32 位数据的任何寻址方式。特殊形式的减法指令是减 1 或 DEC 指令，实现从任何寄存器或存储单元的内容中减去 1。5.3 节给出怎样实现 BCD 和 ASCII 数据减法。和加法一样，在必须进行比 16 位或 32 位宽的数字的减法时，可用**带借位的减法指令**（**subtract-with-borrow**，**SBB**）实现。在 80486 ～ Core2 指令系统中还包括一条比较交换指令。在 Pentium 4 和 Core2 的 64 位模式中，64 位的减法指令仍然有效。

表 5-4 列出了部分可用于减法指令（SUB）的寻址方式。实际上存在着超过上千种减法指令，远远多于所列出的这些。惟独不允许的是存储器对存储器和段寄存器的减法。类似于其他的算术指令，减法指令会影响标志位。

表 5-4 减法指令

汇编语句	操作
SUB CL，BL	CL = CL - BL
SUB AX，SP	AX = AX - SP
SUB ECX，EBP	ECX = ECX - EBP
SUB DH，6FH	DH = DH - 6FH
SUB AX，0CCCCH	AX = AX - 0CCCCH
SUB ESI，2000300H	ESI = ESI - 2000300H
SUB [DI]，CH	从由 DI 寻址的数据段存储单元的字节内容中减去 CH
SUB CH，[BP]	从 CH 中减去由 BP 寻址的堆栈段存储单元字节内容
SUB AH，TEMP	从 AH 中减去数据段存储器地址 TEMP 的字节内容
SUB DI，TEMP [ESI]	从 DI 中减去由 TEMP + ESI 寻址的数据段存储单元的字内容
SUB ECX，DATA1	从 ECX 中减去按 DATA1 寻址的数据段存储单元中的双字内容

寄存器减法指令

例 5-9 给出的指令序列实现寄存器减法，这个例子从 BX 寄存器内容中减去 16 位 CX 和 DX 的内

容，每次减法以后微处理器修改标志寄存器的内容。多数的算术和逻辑运算指令会改变标志位。

例 5-9

```
0000 2B D9      SUB  BX,CX
0002 2B DA      SUB  BX,DX
```

立即数减法

和加法一样，对于常数的减法，微处理器允许使用立即数作操作数。例 5-10 给出一个短指令序列，从 22H 中减去 44H。首先用立即数传送指令将 22H 装入 CH。然后用带立即数 44H 的 SUB 指令从 22H 中减去 44H。减法以后的差值（DEH）放在 CH 寄存器，对于这个减法，标志位的变化如下：

Z = 0（结果不是 0）
C = 1（借位）
A = 1（半借位）
S = 1（结果为负）
P = 1（奇偶性为偶）
O = 0（没有溢出）

减法执行后两个标志位（C 和 A）存放借位，而不是加法之后的进位。注意，例 5-10 里没有溢出。例 5-10 从 22H（+34）中减去 44H（+68），结果是 0DEH（-34）。结果 -34 是正确的 8 位有符号数，没有溢出。只有 8 位带符号的结果出现大于 +127 或小于 -128 的情况才会溢出。

例 5-10

```
0000 B5 22      MOV  CH,22H
0002 80 ED 44   SUB  CH,44H
```

减 1 指令

减 1 指令（DEC）从寄存器或存储单元的内容中减去 1，表 5-5 列出了一些寄存器和存储器寻址的减 1 指令。

表 5-5 减 1 指令

汇编语句	操作
DEC BH	BH = BH - 1
DEC CX	CX = CX - 1
DEC EDX	EDX = EDX - 1
DEC R14	R14 = R14 - 1（64 位模式）
DEC BYTE PTR [DI]	由 DI 寻址的数据段存储单元字节的内容减 1
DEC WORD PTR [BP]	由 BP 寻址的堆栈段存储单元字的内容减 1
DEC DWORD PTR [EBX]	由 EBX 寻址的数据段存储单元双字的内容减 1
DEC QWORD PTR [RSI]	由 RSI（64 位模式）寻址的存储单元四字的内容减 1
DEC NUMB	数据段存储单元 NUMB 的内容减 1

间接寻址的存储器数据减 1 指令要求用 BYTE PTR、WORD PTR、DWORD PTR、QWORD PTR 标识字长，因为当变址寄存器寻址存储器时，汇编程序无法辨认字节和字。例如，DEC [SI] 是含糊的，因为汇编程序不能确定由 SI 寻址的存储单元是字节、字还是双字。用 DEC BYTE PTR [SI]、DEC WORD PTR [DI] 或 DEC DWORD PTR [SI] 能指示数据的长度。在 64 位模式中，DEC QWORD PTR [SI] 表示由 RSI 寄存器指向地址四字的内容减 1。

带借位的减法指令

SBB 指令（带借位的减法）与正规的减法运算指令基本一样，不同的是还要将存于进位标志（C）中的借位从差中减去。这条指令通常多用于 8086 ~ 80286 中比 16 位数宽的减法，在 80386 ~ Core2 中用于宽于 32 位的减法。像多字节加法要传递进位一样，多字节减法需要传递借位。

表 5-6 列出了一些 SBB 指令，解释了它们的操作。像 SUB 指令一样，SBB 影响标志位。注意，这个表中从存储器中减去立即数的指令要求用 BYTE PTR、WORD PTR、DWORD PTR QWORD PTR 伪指令。

表 5-6 带借位的减法指令

汇编语句	操作
SBB AH, AL	AH = AH - AL - 借位
SBB AX, BX	AX = AX - BX - 借位
SBB EAX, ECX	EAX = EAX - ECX - 借位
SBB CL, 2	CL = CL - 2 - 借位
SBB BYTE PTR [DI], 3	从由 DI 寻址的数据段字节存储单元内容中减去 3 和借位
SBB [DI], AL	从由 DI 寻址的数据段字节存储单元的内容中减去 AL 内容和借位
SBB DI, [BP+2]	从 DI 中减去由 BP+2 寻址的堆栈段字存储单元的内容及借位
SBB AL, [EBX+ECX]	从 AL 中减去由 EBX 和 ECX 之和寻址的数据段字节存储单元内容及借位

当存于 BX 和 AX 中的 32 位数减去存于 SI 和 DI 的 32 位数时，进位标志传递两个 16 位减法之间的借位，进位标志保存减法的借位。图 5-2 指出在这个任务中是怎样通过进位标志传递借位的。例 5-11 给出了程序如何实现这种减法。对于多字节的减法，最低有效 16 位或 32 位数据相减用 SUB 指令，后续的所有高位有效数字相减用 SBB 指令。这个例子用 SUB 指令从 AX 中减去 DI，然后用带借位的减法指令 SBB，从 BX 中减去 SI。

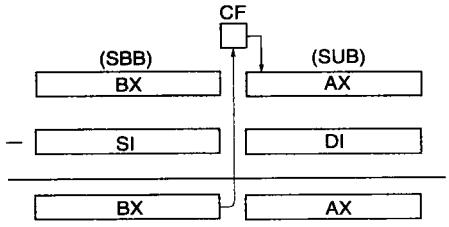


图 5-2 带借位的减法，指出了进位标志（C）怎样传递借位

例 5-11

```
0000 2B C7      SUB  AX,DI
0002 1B DE      SUB  BX,SI
```

5.1.3 比较指令

比较指令（CMP）是只改变标志位而目的操作数不变的减法指令。用比较指令常常是为了对照某一个数值检查寄存器或存储单元内容的大小。CMP 指令后面常常跟随测试标志位状态的条件转移指令。

表 5-7 列出了条件比较指令，它们使用的寻址方式与前面已经出现过的加法和减法指令相同。惟有存储器和存储器比较以及与段寄存器比较是非法的。

表 5-7 比较指令

汇编语句	操作
CMP CL, BL	CL - BL
CMP AX, SP	AX - SP
CMP EBP, ESI	EBP - ESI
CMP RDI, RSI	RDI - RSI（64 位）
CMP AX, 2000H	AX - 2000H
CMP R10W, 12H	R10（字部分）- 12H（64 位）
CMP [DI], CH	用 DI 寻址的数据段存储单元的字节内容减 CH
CMP CL, [BP]	用 CL 减由 BP 寻址的堆栈段存储单元的字节内容
CMP AH, TEMP	用 AH 减由 TEMP 寻址的数据段存储单元的字节内容
CMP DI, TEMP [BX]	用 DI 减由 TEMP + BX 寻址的数据段存储单元的字内容
CMP AL, [EDI+ESI]	用 AL 减由 EDI 与 ESI 之和寻址的数据段存储单元的字节内容

例 5-12 给出后面跟随一条条件转移指令的比较指令。在这个例子中，AL 的内容与 10H 比较，比较指令后面的转移指令的条件通常是 **JA（jump above，高转移）**或 **JB（jump below，低转移）**。如果 JA 跟在比较指令之后，则当 AL 中的值高于 10H 时出现转移。如果 JB 跟在比较指令之后，则当 AL 中的值低于 10H 时出现转移。这个例子中 JAE 跟在比较之后。如果 AL 中的值等于或者高于 10H，这条指令使得程序转向存储器地址 SUBER 处继续执行。如果用 **JBE（jump below or equal，低于或等于时转移）**指令跟在比较指令的后面，则是低于或等于 10H 时转移。第 6 章将更详细地说明比较和条件转移指令。

例 5-12

```
0000 3C 10    CMP  AL,10H    ;与10H比较
0002 73 1C    JAE  SUBER     ;如果等于或高于10H
```

比较交换指令（只用于 80486 ～ Core2）

在 80486 ～ Core2 指令系统中安排的比较交换指令（CMPXCHG）使目的操作数与累加器内容比较。如果相等，源操作数就复制到目的操作数中；如果它们不相等，目的操作数复制到累加器中。这条指令用 8 位、16 位或 32 位数据操作。

例如，比较交换指令 CMPXCHG CX，DX，首先比较 CX 与 AX 的内容，如果 CX 等于 AX，则 DX 复制到 CX 中；如果 CX 不等于 AX，则 CX 复制到 AX 中。如果操作数是 8 位的，这条指令用 8 位数据与 AL 比较；如果操作数是 32 位的，就用 32 位数与 EAX 比较。

只有在 Pentium ～ Core2 中可以使用 CMPXCHG8B 指令比较两个四字数据，与先前的微处理器相比，这是 Pentium ～ Core2 新增的数据管理指令。该指令比较并交换 8 个字节，使装在 EDX：EAX 中的 64 位数与存储器中的 64 位数字相比较。例如，CMPXCHG8B TEMP 指令，比较 EDX：EAX 和 TEMP 的内容，如果 TEMP 内容等于 EDX：EAX 内容，用 ECX：EBX 中的值替代 TEMP 中的内容；如果 TEMP 内容不等于 EDX：EAX 内容，则 TEMP 中的数装入 EDX：EAX 中。

零标志位指示比较以后是否相等。这条指令有个毛病，它可能引起操作系统崩溃，有关这个缺点更详细的信息可在 www. intel. com 上找到。在 64 位的 Pentium 4 微处理器中 CMPXCHG16B 指是可用的。

5.2 乘法和除法指令

只有现代微处理器才包含乘法和除法指令。早期 8 位的微处理器不能做乘法和除法，除非使用一系列移位和加法或减法程序实现。微处理器的制造商意识到这是不恰当的，因此在较新的微处理器的指令系统中加入了乘法和除法指令。事实上，Pentium ～ Core2 中包含的特殊的电路，只用一个时钟周期时间就能完成乘法操作。早期的 Intel 微处理器实现同样的乘法要花费 40 多个时钟周期。

5.2.1 乘法指令

乘法指令可对字节、字或双字操作，而且可以对有符号（IMUL）或无符号（MUL）整数操作，只有 80386 ～ Core2 能对两个 32 位双字做乘法。乘法以后的积总是双倍宽的积。如果两个 8 位数相乘，则产生 16 位的积；如果是 16 位数相乘，则产生 32 位的积；如果是两个 32 位的数相乘，则产生 64 位的积。在 64 位的 Pentium 4 中，两个 64 位的数相乘，则产生 128 位的积。

执行乘法指令时改变某些标志位，溢出标志位 O 和进位标志位 C 产生可以预知的结果。其他标志也改变，但是结果是不可预知的，它们没有被使用。8 位乘法中，如果结果的最高有效 8 位全是 0，C 和 O 两个标志位就是 0。这些标志位指示结果是 8 位宽（C = 0）还是 16 位宽（C = 1）；在 16 位乘法中，如果积的最高有效 16 位全是 0，C 和 O 两者清除为 0；在 32 位乘法中，C 和 O 两者指示积的最高有效 32 位全是 0。

8 位乘法指令

对于有符号或无符号数的 8 位乘法，被乘数总是在 AL 中。乘数可以在任何寄存器或任何存储单元中。不允许使用立即数乘法，除非是带符号的立即数乘法，这种乘法指令将在本节后面的程序中出现

时讨论。乘法指令中包含一个操作数，因为总是将它与 AL 寄存器的内容相乘。例如 MUL BL 指令，将 BL 中的无符号数与 AL 中的无符号数相乘，然后双倍宽的无符号乘积放在 AX 中。表 5-8 列出了一些 8 位乘法指令。

表 5-8 8 位乘法指令

汇编语句	操作
MUL CL	AL 内容乘 CL 内容，无符号乘积放在 AX 中
IMUL DH	AL 内容乘 DH 内容，有符号乘积放在 AX 中
IMUL BYTE PTR [BX]	AL 内容乘数据段中由 BX 寻址的存储单元的字节内容，有符号乘积放在 AX 中
MUL TEMP	AL 内容乘数据段中由 TEMP 寻址的字节存储单元的内容，无符号乘积放在 AX 中

假定 BL 和 CL 中各自包含一个 8 位无符号数，要把两数相乘，产生的 16 位乘积存储到 DX 中。这个乘积不能用一条指令产生，因为对于 8 位乘法我们只能乘 AL 寄存器中的数字。例 5-13 给出了这个实现 DX = BL × CL 的短程序。这个例子把数据 5 和 10 装入寄存器 BL 和 CL。乘法以后使用 MOV DX, AX 指令，把乘积 50 从 AX 传送到 DX。

例 5-13

```
0000 B3 05    MOV  BL,5     ;装入数据
0002 B1 0A    MOV  CL,10
0004 8A C1    MOV  AL,CL    ;装入数据
0006 F6 E3    MUL  BL       ;相乘
0008 8B D0    MOV  DX,AX    ;传送乘积
```

对于带符号数的乘法，如果乘积为正，乘积是真正的二进制格式；如果乘积为负，乘积是 2 - 补码格式，这和微处理器存储所有带符号正、负数字用的格式相同。如果要用例 5-13 的程序完成两个有符号数相乘，只需将 MUL 指令变为 IMUL 即可。

16 位乘法指令

字乘法指令类似于字节乘法指令，区别是用 AX 存放被乘数，而不是 AL，而且积出现在 DX-AX 中，而不是 AX 中。DX 寄存器总是包含积的最高有效 16 位，而 AX 总是包含最低有效 16 位。类似于 8 位乘法，乘数寻址方式的选择取决于程序员。表 5-9 给出了几个不同的 16 位乘法指令。

表 5-9 16 位乘法指令

汇编语句	操作
MUL CX	AX 内容乘 CX 内容；无符号积放入 DX-AX 中
IMUL DI	AX 内容乘 DI 内容；有符号积放入 DX-AX 中
MUL WORD PTR [SI]	AX 内容乘数据段内由 SI 寻址的字存储单元的内容；无符号积放入 DX-AX 中

特殊的立即数 16 位乘法指令

8086/8088 不能执行立即数乘法，但是 80186 ～ Core2 可以使用这种特殊类型的乘法指令。立即数乘法指令必须是有符号的乘法，而且指令格式与其他指令不同，因为它包含三个操作数：第 1 个操作数是 16 位目的寄存器，第 2 个操作数是容纳 16 位被乘数的寄存器或存储单元，而第 3 个操作数是作为乘数的 8 位或 16 位立即数。

IMUL CX，DX，12H 指令使 12H 乘以 DX 内容，而将带符号的结果放入 CX 中。如果立即数是 8 位的，在乘法进行前将其符号扩展成 16 位数字。另一个例子是 IMUL BX，NUMBER，1000H 指令。它把 NUMBER 乘 1000H 后，乘积放入 BX 中，目的操作数及被乘数必须都是 16 位数字。立即数乘法指令的局限性限制了它的用途，特别是它只用于有符号数乘法，而乘积又只能是 16 位宽。

32 位乘法指令

在 80386 及更高型号的微处理器中允许执行 32 位乘法，因为这些微处理器包含 32 位的寄存器。

与8位、16位乘法指令一样，通过使用IMUL或MUL指令，32位乘法可以是有符号数的或无符号数的。对于32位乘法，EAX的内容乘以指令规定的操作数，乘积（64位宽）放在EDX-EAX中，EAX包含积的低32位。表5-10列出了在80386及更高型号的微处理器指令系统中用到的一些32位乘法指令。

表5-10 32位乘法指令

汇编语句	操作
MUL ECX	EAX内容乘以ECX内容，无符号的积放入EDX-EAX中
IMUL EDI	EAX内容乘以EDI内容，有符号的积放入EDX-EAX中
MUL DWORD PTR［ESI］	EAX内容乘以数据段内由ESI寻址的存储单元的双字内容，无符号的积放入EDX-EAX

64位乘法指令

Pentium 4中64位乘法的结果出现在RDX:RAX寄存器对中，为128位乘积。虽然这么长的乘法比较罕见，但Pentium 4和Core2都能以有符号数或无符号数执行它。表5-11表示了几个这种高精度乘法的例子。

表5-11 64位乘法指令

汇编语句	操作
MUL RCX	RAX内容乘以RCX内容，无符号的积放入RDX-RAX中
IMUL RDI	RAX内容乘以RDI内容，有符号的积放入RDX-RAX中
MUL QWORD PTR［RSI］	RAX内容乘以数据段内由RSI寻址的存储单元的四字内容，无符号的积放入RDX-RAX

5.2.2 除法指令

类似于乘法指令，8086～80286微处理器有8位数的或16位数的除法指令，80386～Pentium 4也有32位数的除法指令。这些操作数是有符号的（IDIV）或无符号的（DIV）整数。被除数的长度总是操作数的两倍，这意味着8位除法用16位数除以8位数，16位除法用32位数除以16位数，32位除法用64位数除以32位数。在任何微处理器中都不存在立即数除法指令。在Pentium 4和Core2的64位模式中，64位除法用128位数除以64位数。

对于除法，标志位是不可预知的。除法可以发生两种不同类型的错误。其中一个是试图除以0；另一个是除法溢出。在很大的数除以较小的数时可能会产生除法溢出。例如AX=3000被2除。因为对8位除法而言，商在AL中，结果1500使得除法溢出。这两种类型的错误使微处理器产生中断。在多数系统中，除法错误中断在视频显示屏幕上显示一个出错信息。微处理器的除法错误中断和其他所有的中断在第6章中说明。

8位除法指令

8位除法指令用AX寄存器存放被除数，可除以任何8位寄存器或存储单元的内容。除法完成后商放在AL中，用AH保存全部的余数。对于有符号除法，商是正的或负的，而余数总是有被除数的符号并且是整数。例如，如果AX=0010H（+16）和BL=0FDH（-3），执行IDIV BL指令，结果AX=01FBH，表示商是-5（AL），余数是1（AH）。如果是+3除以-16，结果将是商为-5（AL），余数为-1（AH）。表5-12列出了一些8位除法指令。

表5-12 8位除法指令

汇编语句	操作
DIV CL	AX内容除以CL内容，AL中存放得到的无符号的商，而余数在AH中
IDIV BL	AX内容除以BL内容，AL中存放得到的有符号的商，而余数在AH中
DIV BYTE PTR［BP］	AX除以堆栈段中用BP寻址的字节存储单元的内容，无符号的商在AL中，而余数在AH中

用8位除法指令时，操作数通常是8位宽。这就意味着另一个数，也就是被除数必须转换为16位宽的数字并放在AX中。对于有符号的与无符号的数字，实现这种转换是有区别的。对于无符号的数字，最高有效8位必须清除为**零**（**zero-extended，零扩展**），第4章中叙述过的MOVZX指令能够用于80386～Core2微处理器中数的零扩展；对于有符号数，低8位的符号要扩展到高8位。在微处理器中有一条专门的指令将AL中的符号扩展到AH，即把AL中的8位有符号数转换为AX中的16位有符号数。**CBW**（**convert byte to word，扩展字节为字**）指令执行这种转换。在80386～Core2微处理器中，MOVSX（见第4章）指令可以实现数的符号扩展。

例5-14列出了一个无符号除法的短程序。字节存储单元NUMB中的无符号数除以存储单元NUMB1中的无符号数，商存入单元ANSQ，而余数存入单元ANSR中。注意NUMB单元的内容是怎样从存储器取出，然后零扩展成16位无符号格式的数来作为被除数的。

例5-14

```
0000 A0 0000 R      MOV  AL,NUMB    ;取NUMB
0003 B4 00          MOV  AH,0       ;零扩展
0005 F6 36 0002 R   DIV  NUMB1      ;被NUMB1除
0009 A2 0003 R      MOV  ANSQ,AL    ;保存商
000C 88 26 0004 R   MOV  ANSR,AH    ;保存余数
```

16位除法指令

16位除法指令类似于8位除法指令，只是除数是16位的数字，DX-AX中的32位数是被除数，商在AX中，余数在DX中。表5-13列出了一些16位除法指令。

表5-13 16位除法指令

汇编语句	操作
DIV CX	DX-AX内容除以CX内容；无符号的商在AX中而余数在DX中
IDIV SI	DX-AX的内容除以SI内容，有符号的商在AX中而余数在DX中
DIV NUMB	DX-AX的内容除以数据段存储器地址NUMB内的字内容；无符号的商在AX中而余数在DX中

与8位除法一样，16位除法中被除数的数字必须被转换成适当的格式。如果16位无符号数放在AX中，DX必须被清除为零。在80386及更高型号的微处理器中，数的零扩展用MOVZX指令。如果AX是16位的有符号数，**CWD**（**convert word to doubleword，转换字为双字**）指令把它扩展成为32位的有符号数。如果可以使用80386及更高型号的微处理器，也能用MOVSX指令对数进行符号扩展。

例5-15给出了两个16位有符号数的除法，即AX中的－100被CX中的＋9除。执行除法以前，CWD指令将AX中的－100转换为DX-AX中的－100。除法以后结果出现在DX-AX中，其中商－11在AX中，而余数－1在DX中。

例5-15

```
0000 B8 FF9C    MOV  AX,-100    ;装入-100
0003 B9 0009    MOV  CX,9       ;装入+9
0006 99         CWD             ;符号扩展
0007 F7 F9      IDIV CX
```

32位除法指令

80386～Pentium 4微处理器可以执行32位有符号或无符号数的除法。指令使EDX-EAX的64位内容除以指令指定的操作数，32位的商留在EAX中，32位的余数放在EDX中。除了寄存器长度不同以外，这条指令按照8位或16位除法相同的方式工作。表5-14给出了一些32位除法指令。在有符号的除法前用**CDQ**（**convert doubleword to quadword，转换双字为四字**）指令将32位EAX的内容转换为EDX-EAX中的64位有符号数。

表 5-14 32 位除法指令

汇编语句	操作
DIV ECX	EDX-EAX 的内容除以 ECX 内容；无符号的商在 EAX 中而无符号的余数在 EDX 中
IDIV DATA4	EDX-EAX 的内容除以数据段中用 DATA4 寻址的双字存储单元内容；有符号的商在 EAX 中而有符号的余数在 EDX 中
DIV DWORD PTR [EDI]	EDX-EAX 的内容除以数据段用 EDI 寻址的双字存储单元的内容；无符号的商在 EAX 中而无符号的余数在 EDX 中

余数

如何处理除法以后的余数呢？有几种可能的选择。余数可用来对结果进行四舍五入，也可截断为近似的结果。如果是无符号的除法，四舍五入要求将余数与除数的一半比较，以决定余数是入到商中还是舍去。余数还可以转换成小数形式。

例 5-16 给出一个指令序列：AX 内容除以 BL 内容，无符号的结果四舍五入。这个程序在余数与 BL 比较以前把余数加倍，以便决定是否向商数四舍五入。这里，比较以后用 INC 指令对 AL 中的商入 1。

例 5-16

```
0000  F6  F3        DIV  BL        ;除
0002  02  E4        ADD  AH,AH     ;余数加倍
0004  3A E3         CMP  AH,BL     ;测试舍入吗?
0006  72  02        JB   NEXT      ;如果是
0008  FE C0         INC  AL        ;舍入
000A          NEXT:
```

假定要求用小数表示余数，而不是整数余数，可通过以下方法求得：先把商保存起来，将 AL 寄存器清除为零，然后用 AX 中剩余的数除以原来的操作数，产生小数形式的余数。

例 5-17 说明怎样实现 13 被 2 除。8 位的商保存在存储单元 ANSQ 中，AL 被清 0，然后 AX 的内容再除以 2 产生小数余数。除法以后，AL 寄存器等于 80H 也就是 10000000_2。如果二进制数小数点放在 AL 的最左面，AL 中小数形式的余数是 0.1000000_2 或十进制数的 0.5。余数保留在存储单元 ANSR 中。

例 5-17

```
0000  B8  000D      MOV  AX,13      ;装入 13
0003  B3  02        MOV  BL,2       ;装入 2
0005  F6  F3        DIV  BL         ;13/2
0007  A2  0003  R   MOV  ANSQ,AL    ;保存商
000A B0  00         MOV  AL,0       ;AL 清 0
000C F6  F3         DIV  BL         ;产生余数
000E A2  0004  R    MOV  ANSR,AL    ;保存余数
```

64 位除法指令

在 64 位模式操作的 Pentium 4 微处理器执行 64 位有符号或无符号数的除法。64 位除法用 RDX：RAX 寄存器对放被除数，除之后，商留在 RAX 中，余数放入 RDX 中。表 5-15 给出了几个 64 位除法指令。

表 5-15 64 位除法指令举例

汇编语句	操作
DIV RCX	RDX-RAX 的内容除以 RCX 内容；无符号的商在 RAX 中而无符号的余数在 RDX 中
IDIV DATA4	RDX-RAX 的内容除以存储单元 DATA4 中的四字内容；有符号的商在 RAX 中而有符号的余数在 RDX 中
DIV QWORD PTR [RDI]	RDX-RAX 的内容被用 RDI 寻址的四字存储单元的内容除；无符号的商在 RAX 中而无符号的余数在 RDX 中

5.3 BCD 码和 ASCII 码算术运算指令

微处理器可以进行**二进制编码的十进制（binary-coded decimal，BCD）**数和**美国标准信息交换码（American Standard Code for Information Interchange，ASCII）**数的算术运算。这是通过调整 BCD 和 ASCII 算术运算结果的指令实现的。

BCD 操作会出现在诸如销售点终端（例如现金出纳机）和其他很少需要复杂算术运算的系统中。许多程序处理 ASCII 码数据时需要执行 ASCII 码运算。BCD 或 ASCII 算术运算目前已很少使用了，但某些操作可能用于其他用途。

在本章这一节中没有详细解释任意一条 Pentium 4 或 Core2 在 64 位模式的指令功能，将来 BCD 和 ASCII 指令很可能会变成没用的指令。

5.3.1 BCD 算术运算指令

BCD 数据的算术运算有两种：加法和减法。指令系统提供了两条指令，修正 BCD 加法和减法的结果。**DAA（decimal adjust after addition，加法后十进制调整）**指令跟在 BCD 加法后面，而 **DAS（decimal adjust after subtraction，减法后十进制调整）**指令跟在 BCD 减法后面。两条指令把加法和减法的结果调整为 BCD 数字。

BCD 码数据总是以压缩格式出现，每个字节存放两个 BCD 数字位。调整指令只在 BCD 加法和减法以后对 AL 寄存器进行调整。

DAA 指令

DAA 指令跟随在 ADD 或 ADC 指令之后，把运算结果调整为 BCD 结果。假定 DX 和 BX 每个都包含有 4 位压缩 BCD 数。例 5-18 提供一个短程序，将 DX 和 BX 中的 BCD 数相加，并且将结果存入 CX 中。

例 5-18

```
0000 BA 1234    MOV  DX,1234H    ;装入 1234 (BCD)
0003 BB 3099    MOV  BX,3099H    ;装入 3099 (BCD)
0006 8A C3      MOV  AL,BL       ;BL 和 DL 之和
0008 02 C2      ADD  AL,DL
000A 27         DAA
000B 8A C8      MOV  CL,AL       ;结果送 CL
000D 9A C7      MOV  AL,BH       ;BH、DH 及进位位之和
000F 12 C6      ADC  AL,DH
0011 27         DAA
0012 8A E8      MOV  CH,AL       ;结果送 CH
```

由于 DAA 指令只对 AL 寄存器的结果进行调整，这里加法必须每次只做 8 位。BL 和 DL 寄存器内容相加以后，用 DAA 指令调整结果，将结果存入 CL。然后 BH 与 DH 带进位位相加，再用 DAA 指令调整结果，将结果存入 CH。在这个例子中，1234 加 3099 产生的和为 4333，加法后放在 CX 中。注意 BCD 数 1234 与 1234H 相同。

DAS 指令

DAS 指令的作用类似于 DAA 指令，只是它跟随在减法（而不是加法）之后。例 5-19 与例 5-18 基本相同，只是用 DX 和 BX 的减法替代了加法。这两个程序的主要区别是 DAA 指令变成了 DAS 指令，ADD 和 ADC 指令变成了 SUB 和 SBB 指令。

例 5-19

```
0000 BA 1234    MOV  DX,1234H    ;装入 1234
0003 BB 3099    MOV  BX,3099H    ;装入 3099
0006 8A C3      MOV  AL,BL       ;BL 减去 DL
```

```
0008 2A C2      SUB  AL,DL
000A 2F         DAS
000B 8A C8      MOV  CL,AL      ;结果送 CL
000D 9A C7      MOV  AL,BH      ;减 DH
000F 1A C6      SBB  AL,DH
0011 2F         DAS
0012 8A E8      MOV  CH,AL      ;结果在 CH
```

5.3.2 ASCII 算术运算指令

ASCII 算术运算指令对 ASCII 码数据进行操作，30H ～ 39H 之间的这些 ASCII 码对应于 0 ～ 9 十个数码。ASCII 算术运算有 4 条指令：**AAA（ASCII adjust after addition，加法后 ASCII 调整）**、**AAD（ASCII adjust before division，除法前 ASCII 调整）**、**AAM（ASCII adjust after multiplication，乘法后 ASCII 调整）**和 **AAS（ASCII adjust after subtraction，减法后 ASCII 调整）**。这些指令都以寄存器 AX 作为源操作数或目的操作数。

AAA 指令

两个 1 位的 ASCII 码数据相加不产生任何实用的数据。例如，如果 31H 和 39H 相加，结果是 6AH。这一 ASCII 加法（1 +9）应当产生等于十进制数 10 的两位 ASCII 结果（ASCII 码 31H 和 30H）。如果加法以后执行 AAA 指令，AX 的内容将是 0100H，虽然这不是 ASCII 码，但是通过加 3030H 可以转换成 ASCII 码，得到 3130H。当结果小于 10 时，AAA 指令将 AH 清 0；当结果大于 10 时，则使 AH 加 1。

例 5-20 给出了微处理器中的 ASCII 加法。请注意，加法前用 MOV AX，31H 指令将 AH 清 0，即把操作数 0031H 的 00H 放入 AH，而 31H 放入 AL。

例 5-20

```
0000 B8 0031    MOV  AX,31H     ;装入 ASCII 1
0003 04 39      ADD  AL,39H     ;装入 ASCII 9
0005 37         AAA             ;调整结果
0006 05 3030    ADD  AX,3030H   ;结果变成 ASCII 码
```

AAD 指令

与其他所有的调整指令都不一样，AAD 指令只用于除法之前。AAD 指令在执行除法之前要求 AX 寄存器含有两个非压缩 BCD 数字（不是 ASCII）。用 AAD 指令调整 AX 寄存器以后，除以一个非压缩的 BCD 数，在 AL 中产生 1 位 BCD 数结果，余数放在 AH 中。

例 5-21 说明了非压缩的 BCD 数 72 怎样除以 9 而产生商数 8。0702H 装入 AX 寄存器以后被 ADD 指令调整为 0048H。注意，这一转换是将一个两位的非压缩 BCD 数转换为二进制数，这样就可以用二进制除法指令了。AAD 指令把从 00 到 99 之间的非压缩的 BCD 数转换为二进制数。

例 5-21

```
0000 B3 09      MOV  BL,9       ;装入除数
0002 B8 0702    MOV  AX,702H    ;装入被除数
0005 D5 0A      AAD             ;调整
0007 F6 F3      DIV  BL         ;除
```

AAM 指令

AAM 指令跟在两个 1 位非压缩 BCD 数相乘的乘法指令后面。例 5-22 给出了 5 ×5 的短程序。执行乘法以后 AX 中的结果是 0019H，用 AAM 指令调整结果后，AX 的内容是 0205H，这正是非压缩 BCD 码的 25。如果用 3030H 和 0205H 相加就变成了 ASCII 码的结果 3235H。

例 5-22

```
0000 B0 05    MOV  AL,5     ;装入被乘数
0002 B1 05    MOV  CL,5     ;装入乘数
```

```
0004  F6  E1    MUL  CL
0006  D4  0A    AAM          ;调整
```

AAM 指令通过用 AX 的内容除以 10 来实现这样的转换，余数在 AL 中而商在 AH 中。注意，AAM 指令的第二字节是 0AH，如果将 0AH 变成另外的值，AAM 指令就除以这个值。例如，如果指令第二字节的内容变成 0BH，AAM 指令除以 11。

AAM 指令的另一个用处是它把二进制数转换为非压缩的 BCD。如果 0000H 到 0063H 之间的一个二进制数放在 AX 寄存器内，AAM 指令可将它转换为 BCD 码。例如，如果执行 AAM 前 AX 的内容是 0060H，AAM 指令执行以后它的内容将是 0906H，这正是十进制数 96 的非压缩 BCD 码。如果将 3030H 和 0906H 相加，结果就变成了 ASCII 码。

例 5-23 给出如何用除法和 AAM 指令将 AX 中的 16 位二进制转换成 4 位 ASCII 码的字符串。注意，这个程序只能对 0 到 9999 之间的数转换。首先 DX 清 0，然后 DX-AX 内容被 100 除。例如，若 AX = 245_{10}，执行除法以后 AX = 2，DX = 45，用 AAM 指令把这两项分别转换成 BCD 数据，然后加 3030H 转换成 ASCII 码。

例 5-23

```
0000  33  D2      XOR  DX,DX      ;清 DX 寄存器
0002  B9  0064    MOV  CX,100     ;DX-AX 被 100 除
0005  F7  F1      DIV  CX
0007  D4  0A      AAM             ;把商变为 BCD 码
0009  05  3030    ADD  AX,3030H   ;变换为 ASCII 码
000C  92          XCHG AX,DX      ;重复变换余数一遍
000D  D4  0A      AAM
000F  05  3030    ADD  AX,3030H
```

例 5-24 使用了 AAM 指令，用 AH = 02H 的 DOS 21H 号功能调用，在视频显示器上以十进制形式显示一个数字。注意怎样用 AAM 把 AL 内容转换为 BCD 码，然后用 ADD AX，3030H 指令把 AX 中的 BCD 码转换为 ASCII 码，以便用 DOS INT 21H 调用显示。数据被转换成 ASCII 码后，为显示它们，高位数由 AH 装入 DL 中，显示高位数字；然后低位数再由 AL 装入 DL 中，显示低位数字。注意，DOS INT 21H 功能调用会改变 AL 中的内容。

例 5-24

```
                ;显示第一条指令装入 AL 的十进制数（48H）的程序
                ;
                .MODEL TINY          ;选 TINY 模型
0000            .CODE                ;代码段开始
                .STARTUP             ;程序开始
0100  B0  48        MOV  AL,48H      ;测试数据装入 AL
0102  B4  00        MOV  AH,0        ;清 AH
0104  D4  0A        AAM              ;变换为 BCD
0106  05  3030      ADD  AX,3030H    ;变换为 ASCII
0109  8A  D4        MOV  DL,AH       ;显示最高有效位
010B  B4  02        MOV  AH,2
010D  50            PUSH AX
010E  CD  21        INT  21H
0110  58            POP  AX
0111  8A  D0        MOV  DL,AL       ;显示最低有效位
0113  CD  21        INT  21H
                .EXIT                ;返回到 DOS
                END
```

AAS 指令

与其他 ASCII 调整指令类似，AAS 指令调整 ASCII 减法以后的 AX 寄存器内容。例如，假定从 39H 中减去 35H，结果是 04H，不需要修正，此时 AAS 指令既不修正 AH，也不修正 AL。另一种情况，如果从37H 中减去38H，AL 将等于09H，而且从 AH 里的数中减 1。这种减 1 功能使得多位 ASCII 数据减法成为可能。

5.4 基本逻辑运算指令

基本的逻辑运算指令包括 AND、OR、XOR 和 NOT。这一节还讲解了另一个逻辑运算指令 TEST，因为它是逻辑运算指令 AND 的一种特殊形式，同时介绍了类似于 NOT 指令的 NEG 指令。

逻辑运算操作在底层软件中提供了对二进制位的控制。逻辑运算指令可对位进行置位、清 0 或取补。底层软件以汇编语言或机器语言的形式出现，常用于控制系统中的 I/O 设备。全部的逻辑运算指令都影响标志位。它们总是将进位位和溢出位清 0，其他标志位的变化取决于结果的条件。

当管理寄存器或存储单元中的二进制数据时，将最低位记为第 0 位。字节中位的位置序号是从第 0 位向左递增到第 7 位，而字是到第 15 位，32 位双字最高位是第 31 位，64 位四字（64 位模式下）最高位是第 63 位。

5.4.1 AND 指令

AND 操作执行图 5-3 中真值表所示的逻辑乘操作，A 和 B 两位相“与”，产生结果 T。正像真值表指示的那样，只有当 A 和 B 都是逻辑 1 时，T 才是逻辑 1。对于 A 和 B 的所有其他输入条件，T 是逻辑 0。任何数和 0 相“与”总是逻辑 0；而 1 AND 1 总是逻辑 1，这是非常重要的。

如果速度要求不是太高，AND 指令可以替代“与”门，然而这通常只用在嵌入式控制应用中（Intel 公司已经推出了嵌入式控制器 80386 EX，它包含了 PC 系统的基本结构）。对于 8086 微处理器，AND 指令执行时间大约是 1 微秒。较新型的微处理器执行速度大大提高。就 3.0GHz 的 Pentium 来说，其时钟是 1/3 ns，每个时钟可以执行三条 AND 指令（每个 AND 操作 1/9 ns）。如果 AND 指令所代替的电路速度比微处理器工作速度慢很多，就可以用 AND 指令代替逻辑电路。这样代替可以节省相当数量的资金。单个的 AND 门集成电路（74HC08）价值约 40 美分，而在只读存储器中存储 AND 指令花费少于 1/100 美分。注意，逻辑电路的替代只出现在基于微处理器的控制系统中，在 PC 中通常很少应用。

A	B	T
0	0	0
0	1	0
1	0	0
1	1	1

a)



b)

图 5-3 a）AND 操作的真值表和 b）AND 门的逻辑运算符号

```
  xxxx xxxx   未知数
• 0000 1111   屏蔽码
-------------
  0000 xxxx   结果
```

图 5-4 AND 操作怎样使二进制数的某些位清 0

AND 操作也可用于二进制数的某些位清 0。这种将二进制数某位清 0 的工作称为**屏蔽**（**masking**）。图 5-4 说明了屏蔽处理。注意，最左 4 位清除为零，因为 0 与任何数“与”都是 0；和 1“与”的位不变，这是因为 1 AND 1，结果为 1；而 1 AND 0，结果为 0。

除了存储器-存储器和段寄存器寻址方式以外，AND 指令可以使用任何寻址方式。表 5-16 列出了一些 AND 指令和它们的操作。

表 5-16 AND 指令

汇编语句	操作
AND AL, BL	AL = AL and BL
AND CX, DX	CX = CX and DX
AND ECX, EDI	ECX = ECX and EDI
AND RDX, RBP	RDX = RDX and RBP（64 位）

（续）

汇编语句	操　作
AND CL, 33H	CL = CL and 33H
AND DI, 4FFFH	DI = DI and 4FFFH
AND ESI, 34H	ESI = ESI and 34H
AND RAX, 1	RAX = RAX and 1（64 位）
AND AX, [DI]	AX 内容和数据段中由 DI 寻址的字存储单元的内容相与
AND ARRAY [SI], AL	数据段中由 ARRAY 加 SI 寻址的字节存储单元的内容和 AL 内容相与
AND [EAX], CL	数据段中由寄存器 EAX 寻址的字节存储单元的内容和 CL 的内容相与

通过用 AND 指令把 ASCII 码数的最左边 4 位二进制位屏蔽掉，就可以转换为 BCD 码。这样可将 ASCII 码 30H～39H 转换为 0～9。例 5-25 给出了将 BX 中的 ASCII 内容转换为 BCD 码的短程序。这里 AND 指令将两位 ASCII 码同时转换为 BCD 码。

例 5-25

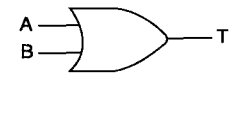
```
0000 BB 3135      MOV  BX,3135H    ;装入 ASCII
0003 81 E3 0F0F   AND  BX,0F0FH    ;屏蔽 BX
```

5.4.2 OR 指令

OR 操作实现的逻辑加，通常称为“或”运算。如果它的输入中有任意一个是 1，则 OR 产生逻辑运算输出为 1。只有当全部输入都是 0 时，输出才为 0。图 5-5 给出了“或”操作的真值表，其中输入 A 和 B “或”，结果产生输出 T。重要的是要记着 1 和任何位“或”仍然是 1。

A	B	T
0	0	0
0	1	1
1	0	1
1	1	1

a)



b)

图 5-5　a）OR 操作真值表和
b）OR 门的逻辑运算符号

在嵌入式控制器应用中，OR 指令也可以替代离散的“或”门，以节约开支。因为一个 4-2 输入端“或”门（74HC32）花费大约 40 美分，而存储在只读存储器中的 OR 指令的花费少于 1/100 美分。

图 5-6 给出了“或”门怎样将二进制的任何一位置 1。其中未知数（XXXX XXXX）与 0000 1111 “或”，产生结果 XXXX 1111。右边 4 位被置 1，而左边 4 位保持不变。OR 操作使某位置位；而 AND 操作使某位清 0。

```
  xxxx xxxx   未知数
+ 0000 1111   屏蔽码
  xxxx 1111   结果
```

图 5-6　指出“或”功能操作怎样使一个数的某些位置 1

除了段寄存器寻址以外，OR 指令还可以使用其他指令允许的任何寻址方式。表 5-17 说明了几个 OR 指令和它们的操作。

表 5-17　OR 指令

汇编语句	操　作
OR AH, BL	AH = AH or BL
OR SI, DX	SI = SI or DX
OR EAX, EBX	EAX = EAX or EBX
OR R9, R10	R9 = R9 or R10（64 位）
OR DH, 0A3H	DH = DH or 0A3H
OR SP, 990DH	SP = SP or 990DH
OR EBP, 10	EBP = EBP or 10
OR RBP, 1000H	RBP = RBP or 1000H（64 位）
OR DX, [BX]	*DX 的内容和数据段由 BX 寻址的字存储单元的内容相“或”*
OR DATES [DI+2], AL	数据段中用 DATES 加 DI 加 2 寻址的存储单元的内容和 AL 的内容相“或”

假定两个 BCD 数字相乘并且用 AAM 指令将结果调整为两个非压缩的 BCD 数字存放在 AX 中。例 5-26 说明了这个乘法操作以及怎样用 OR 指令将两个十进制的结果转换为 ASCII 码。其中，指令 OR AX，3030H 将 AX 中的 0305H 转换为 3335H。OR 操作也可以用 ADD AX，3030H 指令代替，得到同样的结果。

例 5-26

```
0000 B0 05      MOV  AL,5        ;加载数据
0002 B3 07      MOV  BL,7
0004 F6 E3      MUL  BL
0006 D4 0A      AAM              ;调整
0008 0D 3030    OR   AX,3030H    ;变换为 ASCII 码
```

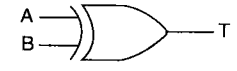
5.4.3 XOR 指令

异或指令（XOR）与“或”（OR）不同，在输入 1 和 1 的条件下 OR 操作的结果为 1；而在输入 1 和 1 的条件下 XOR 操作的结果为 0。“异或”操作是条件互斥；而“或”操作是条件包含。

图 5-7 给出了“异或”功能的真值表（与图 5-5 比较鉴别这两者之间的区别）。如果“异或”输入的是两个 0 或两个 1，输出是 0；如果输入不同，输出是 1。由于这个原因，“异或”有时也称为比较器。

A	B	T
0	0	0
0	1	1
1	0	1
1	1	0

a)



b)

图 5-7 a）“异或”操作真值表和 b）“异或”门的逻辑运算符号

除了段寄存器以外，“异或”指令可使用任何寻址方式。表 5-18 列出了“异或”指令的格式以及有关它们的操作。

类似于 AND 和 OR 指令，“异或”可以替代嵌入式控制应用中的离散电路。74HC86 中 4-2 输入端的“异或”门可用一条 XOR 指令替代。74HC86 花费约 40 美分，而存储在存储器中的指令花费少于 1/100 美分。替换 74HC86 能节省相当数量的资金，特别是在要大量制造的时候。

表 5-18 XOR 指令

汇编语句	操作
XOR CH, DL	CH = CH xor DL
XOR SI, BX	SI = SI xor BX
XOR EBX, EDI	EBX = EBX xor EDI
XOR RAX, RBX	RAX = RAX xor RBX（64 位）
XOR AH, 0EEH	AH = AH xor 0EEH
XOR DI, 00DDH	DI = DI xor 00DDH
XOR ESI, 100	ESI = ESI xor 100
XOR R12, 20	R12 = R12 xor 20（64 位）
XOR DX, [SI]	DX “异或”数据段内 SI 寻址的字存储单元的内容
XOR DEAL [BP+2], AH	堆栈段中由 BP 加 2 寻址的字节存储单元的内容和 AH “异或”

如果寄存器或存储单元中的一些位必须取反，“异或”指令就很实用。这条指令允许数的一部分取反或取补。图 5-8 给出了未知数的一部分怎样用 XOR 取反。注意，当 1 “异或” X 时结果是 $\overline{X}$；而如果是 0 “异或” X，则结果是 X。

```
xxxx xxxx   未知数
⊕0000 1111  屏蔽码
xxxx x̄x̄x̄x̄  结果
```

图 5-8 指出“异或”操作功能怎样使数据位取反

假定要将 BX 寄存器中的左边 10 位取反而不改变右边的 6 位，XOR BX，0FFC0H 指令就可完成这个任务。AND 指令使某些位清 0，OR 指令使之置位，而 XOR 指令对某些位取反。这三条指令允许程序直接控制存放在寄存器或存储单元中的位。这样适合于设备必须打开（1）；必须关闭（0）；从开转换到关或从关转换到开等控制系

统的应用程序。

“异或”指令一个十分普通的应用是使寄存器清0。例如，XOR CH，CH 指令清除寄存器 CH 为00H，指令要占两个字节内存。MOV CH，00H 指令也使 CH 清除为00H，但是需要三个字节的内存。为节省内存空间，常用 XOR 指令代替立即数传送指令清除寄存器。

例 5-27 给出一个短指令序列，清除 CX 的第0位和第1位，置位 CX 的第9位和第10位，并且使 CX 的第12位取反。

例 5-27

```
0000  81  C9  0600    OR   CX,0600H    ;置位位 9 和位 10
0004  83  E1  FC      AND  CX,0FFFCH   ;清除位 0 和位 1
0007  81  F1  1000    XOR  CX,1000H    ;第 12 位取反
```

5.4.4 测试和位测试指令

TEST 指令执行 AND 操作，区别是 AND 指令改变目的操作数，而 TEST 指令不改变目的操作数。TEST 只影响标志寄存器的状态，指示测试的结果。TEST 指令使用与 AND 指令相同的寻址方式。表 5-19 列出了一些 TEST 指令的格式和它们的操作。

表 5-19 TEST 指令

汇编语句	操作
TEST DL，DH	DL 内容和 DH 内容相“与”
TEST CX，BX	CX 内容和 BX 内容相“与”
TEST EDX，ECX	EDX 内容和 ECX 内容相“与”
TEST RDX，R15	RDX 内容和 R15 内容相“与”
TEST AH，4	AH 内容和 4 相“与”
TEST EAX，256	EAX 内容和 256 相“与”

TEST 指令的作用和 CMP 指令相似。不同的是 TEST 指令通常测试单个位（偶尔为多位），而 CMP 指令测试整个字节、字或双字。如果被测试的位是0，则零标志（Z）是逻辑1（指示结果为0）；如果测试的位不为0，零标志（Z）是逻辑0（指示非零结果）。

通常测试指令后面跟着 **JZ（jump if zero，零转移）**或 **JNZ（jump if not zero，非零转移）**指令。目的操作数通常对照着一个立即数来测试，立即数的值为1是测试最右边一位，为2是测试下一位，为4是测试再下一位，以此类推。

例 5-28 给出的短程序测试 AL 寄存器的最右一位和最左一位。用1选择最右一位，128选择最左一位（注意，128是80H）。每个 JNZ 指令跟在测试指令的后面，根据测试的结果跳转到不同的存储器位置。当被测试的位不是零时 JNZ 指令跳转到操作数指示的地址（例中的 RIGHT 或者 LEFT）。

例 5-28

```
0000  A8  01    TEST  AL,1     ;测试最右一位
0002  75  1C    JNZ   RIGHT    ;如果置位
0004  A8  80    TEST  AL,128   ;测试最左一位
0006  75  38    JNZ   LEFT     ;如果置位
```

80386 ～ Pentium 4 微处理器还包括新增测试单一位的位测试指令。表 5-20 列出了这些微处理器增加的4个位测试指令。

四种格式的位测试指令都是测试目的操作数中由原操作数指定的位。例如，BT AX，4 指令测试 AX 中的第4位。测试的结果放入进位标志位。如果第4位是1，进位位置位；如果第4位是0，进位位清0。

表 5-20 位测试指令

汇编语言指令	操作
BT	测试由原操作数规定的目的操作数的某一位
BTC	测试和取反由原操作数规定的目的操作数的一位
BTR	测试和复位由原操作数规定的目的操作数的一位
BTS	测试和置位由原操作数规定的目的操作数的一位

其余的三条位测试指令也是把被测试的位放入进位标志位，但是以后要改变被测试的位。BTC AX，4 指令测试第4位以后把它取反，BTR AX，4 指令测试以后把它清0，BTS AX，4 指令测试以后把它置位。

例 5-29 重复给出了例 5-27 中列出的指令序列。其中 BTR 指令使 CX 中的某位清0，BTS 指令将 CX

中的某位置位，而 BTC 指令对 CX 中的某位取反。

例 5-29

```
0000 0F BA E9 09    BTS  CX,9    ;第 9 位置位
0004 0F BA E9 0A    BTS  CX,10   ;第 10 位置位
0008 0F BA F1 00    BTR  CX,0    ;第 0 位清 0
000C 0F BA F1 01    BTR  CX,1    ;第 1 位清 0
0010 0F BA F9 0C    BTC  CX,12   ;第 12 取反
```

5.4.5 NOT 指令和 NEG 指令

除了下一节的移位和循环指令以外，本节介绍的最后两个逻辑操作指令是逻辑取反（1 的补或 NOT）和算术符号取反（2 的补或 NEG）。这两条指令只有 1 个操作数。表 5-21 列出了一些 NOT 指令和 NEG 指令的形式。像其他多数指令一样，NOT 指令和 NEG 可以用除了段寄存器寻址以外的其他任何寻址方式。

NOT 指令使字节、字或双字的所有位取反。NEG 指令对一个数求 2 的补码，这意味着将有符号数的算术符号从正变为负，或者由负变为正。NOT 操作是逻辑操作，而 NEG 操作是算术操作。

表 5-21 NOT 和 NEG 指令

汇编语句	操作
NOT CH	对 CH 求 1 的补码
NEG CH	对 CH 求 2 的补码
NEG AX	对 AX 求 2 的补码
NOT EBX	对 EBX 求 1 的补码
NEG ECX	对 ECX 求 2 的补码
NOT TEMP	对数据段中由 TEMP 寻址的存储单元的内容求 1 的补码
NOT BYTE PTR [BX]	对数据段里 BX 寻址的字节存储单元的内容求 1 的补码
NOT RAX	对 RAX 求 1 的补码（64 位）

5.5 移位指令和循环移位指令

和 AND、OR、XOR 和 NOT 一样，移位和循环移位指令在二进制位一级上控制二进制数。移位和循环移位通常多应用于底层软件中控制 I/O 设备。微处理器有一套完整的移位和循环移位指令，可对任一存储单元或寄存器中的数据进行移位或循环移位。

5.5.1 移位指令

移位指令把寄存器或存储单元中的数向左或向右放置，即移动。它们也实现简单的算术运算，如乘以 2^{+n}（左移）和乘以 2^{-n}（右移）。微处理器指令系统包括 4 种不同类型的移位指令：两种逻辑移位和两种算术移位。4 种移位操作全部呈现在图 5-9 中。

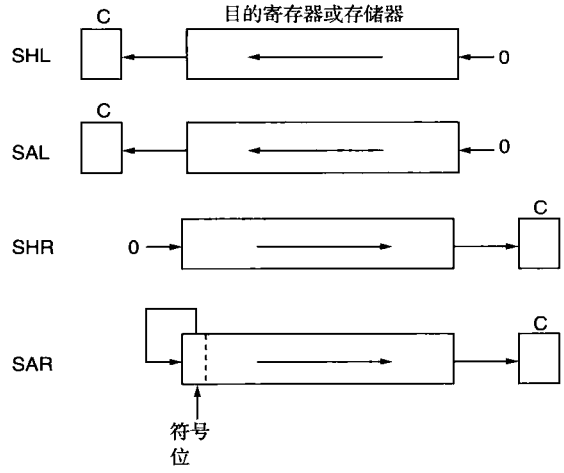


图 5-9 移位指令的操作和移动方向

注意，图 5-9 中有两种右移和两种左移。对于逻辑移位，逻辑左移是把 0 移入最低位，而逻辑右移是把 0 移入最高位。算术移位也有两种，算术左移和逻辑左移相同，而算术右移和逻辑右移不同，因为算术右移是把符号位复制到数字中，而逻辑右移是把 0 复制到数字中。

逻辑移位操作用于无符号数，而算术移位操作用于有符号数。逻辑移位是乘或者除一个无符号数据，而算术移位是乘或者除一个有符号数据。每左移一位就相当乘以 2，而每次右移一位就相当除以 2。如果数字被左移或右移两个位置则相当乘 4 或者除 4。

表 5-22 说明了各种移位指令允许的一些寻址方式。两种不同形式的移位指令，均允许移动任何寄存器（除了段寄存器以外）或存储单元。一种形式用立即数计算移位次数，另一种形式是将移位次数装入寄存器 CL。注意只能用 CL 寄存器保存移位次数。当 CL 作为移位计数器时，移位指令执行时它不改变。注意移位计数是模 32 的计数，这意味着如果移位计数是 33，只把数据移动一个位置（33 除以 32 的余数为 1）。同样可以应用于 64 位的数字，只是移位计数是模 64 的计数。

例 5-30 指出了怎样以两种不同的方式将 DX 寄存器向左移 14 次。第一种方法用立即数计数移位 14 次。第二种方法将 14 装入 CL，然后用 CL 计数移位次数。两条指令都使 DX 的内容向左逻辑移位 14 位。

表 5-22 移位指令

汇编语句	操作
SHL AX, 1	AX 内容逻辑左移一位
SHR BX, 12	BX 内容逻辑右移 12 位
SHR ECX, 10	ECX 内容逻辑右移 10 位
SHL RAX, 50	RAX 内容逻辑左移 50 位（64 位）
SAL DATA1, CL	数据段中的 DATA1 的内容按 CL 规定的位数算术左移
SHR RAX, CL	RAX 的内容按 CL 规定的位数算术右移
SAR SI, 2	SI 内容算术右移两位
SAR EDX, 14	EDX 内容算术右移 14 位

例 5-30

```
0000 C1 E2 0E      SHL  DX,14
                或
0003 B1 0E         MOV  CL,14
0005 D3 E2         SHL  DX,CL
```

像例 5-31 那样，假定要将 AX 的内容乘 10。这可用乘法指令或移位和加法指令两种方法来实现。将数字向左移动一位，就是该数的 2 倍。数字 2 倍以后，再加上原数字的 8 倍，结果就是该数的 10 倍。十进制数字 10 的二进制形式是 1010，即权为 2 和权为 8 的位是逻辑 1，某个数的 2 倍加上 8 倍这个数，结果就是该数的 10 倍。用这种技术可以写出任何常数的乘法。早期的 Intel 微处理器中，用这种技术执行乘法通常比用乘法指令快。

例 5-31

```
                ;AX 的内容乘 10 (1010)
                ;
0000 D1 E0          SHL  AX,1          ;2 倍 AX 的数字
0002 8B D8          MOV  BX,AX
0004 C1 E0 02       SHL  AX,2          ;8 倍原 AX 的数字
0007 03 C3          ADD  AX,BX         ;10 倍原 AX 的数字
                ;
                ;AX 的内容乘 18 (10010)
                ;
0009 D1 E0          SHL  AX,1          ;2 倍原 AX 的数字
000B 8B D8          MOV  BX,AX
000D C1 E0 03       SHL  AX,3          ;16 倍原 AX 的数字
0010 03 C3          ADD  AX,BX         ;18 倍原 AX 的数字
                ;
                ;AX 的内容乘 5 (101)
                ;
0012 8B D8          MOV  BX,AX
0014 C1 E0 02       SHL  AX,2          ;4 倍原 AX 的数字
0017 03 C3          ADD  AX,BX         ;5 倍原 AX 的数字
```

5.5.2 双精度移位指令

80386 及更高型号的微处理器包含两条双精度移位指令（只是 80386 ～ Core2 才有）：SHLD（左

移）和 SHRD（右移）。每条指令有三个操作数而不像其他移位指令只有两个操作数。两条指令都可对两个 16 位或 32 位寄存器进行操作，或者是一个 16 位或 32 位存储单元和一个寄存器进行操作。

SHRD AX，BX，12 指令是一个双精度右移指令的例子。这条指令将 AX 寄存器逻辑右移 12 位，BX 的右边 12 位移入 AX 的左边 12 位中，而 BX 的内容保持不变。移位计数可以用立即数计数，如这个例子给出的那样，或者类似于其他的移位指令放在寄存器 CL 中。

SHLD EBX，ECX，16 指令向左移位 EBX。移位以后，ECX 的最左 16 位移入 EBX 的最右边 16 位，与前面一条指令一样，第二操作数 ECX 的内容保持不变。这条指令以及 SHRD 指令都影响标志位。

5.5.3 循环移位指令

循环移位指令可将寄存器或存储器中的二进制数据从一端循环移位到另一端，或者通过进位标志位从一端循环移动到另一端，它们通常用于 8086～80286 微处理器中比 16 位宽的数据的移位，或者用于 80386～Core2 中比 32 位宽的数据的移位。图 5-10 中给出了 4 种可用的循环指令。

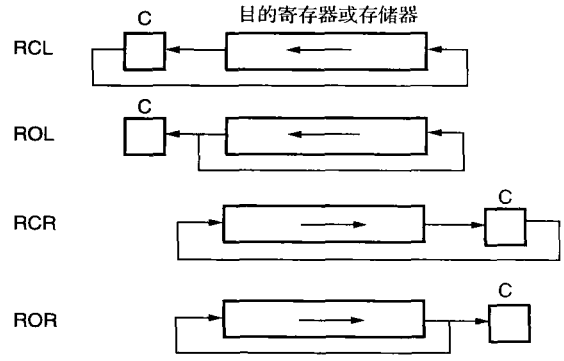


图 5-10 循环移位指令的方向和每种循环移位操作

对于通过寄存器/存储器和 C 标志位（进位位），和只通过寄存器/存储器两种循环移位指令，程序员都可以选择向左循环或向右循环。循环移位指令使用的寻址方式与移位指令使用的相同。循环计数可以是立即数或者装入 CL 寄存器中。表 5-23 列出了一些循环移位指令。如果用 CL 为循环计数，它保持不变。与移位指令一样，CL 中的计数是模为 32 的计数用于 32 位操作和模为 64 的计数用于 64 位操作。

表 5-23 循环移位指令

汇编语句	操作
ROL SI，14	SI 内容循环左移 14 位
RCL BL，6	BL 内容通过进位位循环左移 6 位
ROL ECX，18	ECX 内容循环左移 18 位
ROL RDX，40	RDX 内容循环左移 40 位
RCR AH，CL	AH 内容通过进位位循环右移 CL 内容确定的位数
ROR WORD PTR [BP]，2	堆栈段内由 BP 寻址的字存储单元的内容循环右移 2 位

循环指令通常用于对一个较宽的数据向左或向右移位。例 5-32 中列出了将寄存器 DX、BX 和 AX 中的 48 位数据向左移一位的程序。注意，最低有效 16 位（AX）首先向左移位，这样就使 AX 的最左一位移入进位标志。然后循环移位 BX 的指令将进位位移入 BX，而 BX 的最左一位移入进位位。最后的指令将进位位循环移位到 DX，从而完成移位。

例 5-32

```
0000  D1  E0      SHL  AX,1
0002  D1  D3      RCL  BX,1
0004  D1  D2      RCL  DX,1
```

5.5.4 位扫描指令

虽然位扫描指令不对数据移位或循环移位，但它们对整个数据扫描以便搜索数据中为 1 的位，因为在微处理器内这是通过移位数据实现的，所以位扫描指令也包括在本节中。

位扫描指令 **BSF（bit scan forward，向前位扫描）** 和 **BSR（bit scan reverse，向后位扫描）** 只可以用于 80386～Pentium 4 微处理器中，两种格式的扫描指令均扫描整个数据，搜索首先遇到的值为 1 的位。BSF 指令从最低位向高位扫描数据，而 BSR 从最高位向低位扫描数据。如果遇到值为 1 的位，将零标志位置 1，并且把该位的位置放入目的操作数。如果没有遇到为 1 的位（也就是数据为全零），则零标志位被清 0。这样，如果遇到值为 1 的位，结果就不是零。

例如，如果 EAX = 60000000H 并且执行 BSF EBX，EAX 指令，从最低位向高位扫描数字，首先遇

到的值为 1 的位置是位 29，把 29 放入 EBX 内，并且将零标志位置 1。如果对于 EAX 中同样的值，用 BSR 指令扫描，则 EBX 寄存器装入 30，并且置位零标志位。

5.6 串比较指令

如第 4 章所述，串指令有非常强的功能，因为它们使程序员相对容易地操作大数据块。数据块的管理可用串指令 MOVS、LODS、STOS、INS 和 OUTS 实现。

这一节讨论新增加的串指令，它们可以在存储区中查找一个常数，或者比较两个存储区的内容。为实现这些任务，要使用 **SCAS**（**string scan，串扫描**）或 CMPS（**string compare，串比较**）指令。

5.6.1 SCAS 指令

串扫描指令（SCAS）可以比较 AL 寄存器与字节存储区的内容，比较 AX 寄存器与字存储区的内容或比较 EAX 寄存器（只用于 80386 ～ Core2）与双字存储区的内容。SCAS 指令用 AL、AX 或 EAX 中的内容减存储单元中的数，但既不改变寄存器的内容，也不改变存储单元的内容。字节比较使用的操作码是 SCASB，字比较使用的操作码是 SCASW，而双字比较使用的操作码是 SCASD。在所有的情况中，都是附加段中由 DI 寻址的存储单元的内容与 AL、AX 或 EAX 相比较。前面讲过，默认段（ES）是不能用段超越前缀来改变的。

像其他串指令一样，SCAS 指令用方向标志（D）选择对 DI 是自动加 1 还是自动减 1。也可在指令前用条件重复前缀，使 SCAS 指令重复执行。

假定在 BLOCK 处开始的存储区域长为 100 字节，要求测试这个存储区域，查看哪个单元包含有 00H。例 5-33 中的程序给出怎样用 SCASB 指令在这部分存储区搜索 00H。在这个例子中，SCASB 指令带有 **REPNE**（**repeat while not equal，不等于则重复**）前缀。REPNE 前缀使得 SCASB 指令重复直到或者 CX 寄存器达到 0，或者按照 SCASB 指令比较的结果满足相等条件。另一个条件重复前缀是 **REPE**（**repeat while equal，等于则重复**）。无论用哪个重复前缀，CX 的内容均递减 1 且都不影响标志位，而 SCASB 指令中的比较操作使标志位改变。

例 5-33

```
0000  BF 0011 R      MOV  DI,OFFSET BLOCK    ;数据地址
0003  FC             CLD                     ;自动增量
0004  B9 0064        MOV  CX,100             ;加载计数器
0007  32 C0          XOR  AL,AL              ;AL 清 0
0009  F2/AE          REPNE SCASB             ;搜索
```

假设要设计一个程序，跳过存储器数组中的 ASCII 码空格符（在过程里的这个任务如例 5-34 所示）。这个过程假定 DI 寄存器已经寻址到 ASCII 码字符串，而且串长度是 256 字节或更短些。由于这个程序是要跳过空格（20H），SCASB 指令必须用 REPE 前缀（等于则重复）。只要等于条件存在，SCASB 指令就重复比较，搜索 20H。

例 5-34

```
0000  FC             CLD                     ;自动增量
0001  B9 0100        MOV  CX,256             ;计数器
0004  B0 20          MOV  AL,20H             ;取空格
0006  F3/AE          REPE SCASB
```

5.6.2 CMPS 指令

串比较指令（CMPS）总是按字节（CMPSB）、字（CMPSW）或双字（CMPSD）比较两个存储区域的内容。注意，只有 80386 ～ Core2 可以用双字。在 Pentium 4 或 Core2 的 64 操作模式下，CMPSQ 指令使用四字。比较是在数据段内由 SI 寻址的存储单元的内容和附加段内由 DI 寻址的存储单元的内容之间进行。CMPS 指令使 SI 和 DI 都自动加 1 或减 1，它常常和 REPE 或 REPNE 前缀配合使用。可以替

换这些前缀的是**REPZ**（**repeat while zero，等于零则重复**）和**PEPNZ**（**repeat while not zero，不等于零则重复**）。但是程序设计中通常是用 REPE 或 REPNE。

例 5-35 展示一个比较两个存储区，检查内容是否匹配的短程序。CMPSB 指令带有前缀 REPE，使得只要等于条件存在就继续检索。当 CX 寄存器变成 0 或出现不等条件时，CMPSB 指令就停止执行。CMPSB 指令结束以后，CX 寄存器是零或标志位指示相等，则两个串匹配；如果 CX 不是零或标志位指示不相等，则两个串不匹配。

例 5-35

```
0000  BE 0075 R       MOV  SI,OFFSET LINE      ;LINE 地址
0003  BF 007F R       MOV  DI,OFFSET TABLE     ;TABLE 地址
0006  FC              CLD                      ;自动增量
0007  B9 000A         MOV  CX,10               ;计数器
000A  F3/A6           REPE CMPSB               ;搜索
```

5.7 小结

1）加法（ADD）可以是 8 位、16 位或 32 位的。ADD 指令允许除了段寄存器以外的任何寻址方式。当 ADD 指令执行后，大多数标志位（C、A、S、Z、P 和 O）会改变。另一种类型的加法：带进位位加法（ADC），将两个操作数及进位标志位（C）的内容相加。80486 ~ Core2 微处理器还增加了加法和交换组合的指令（XADD）。

2）加 1 指令（INC）将字节、字、双字寄存器或存储单元的内容加 1。加 1 指令对标志位的影响，除了进位标志外与加法指令相同。当存储单元的内容由一指针寻址时，要在 INC 指令中使用 BYTE PTR、WORD PTR、DWORD PTR、QWORD PTR 伪指令。

3）减法（SUB）有字节的、字的、双字的或四字的，并且针对寄存器或存储单元执行。惟一不允许 SUB 指令使用段寄存器寻址方式。减法指令对标志位的影响与 ADD 相同，而且如果是 SBB 形式则要减进位位。

4）减 1 指令（DEC）从寄存器或存储单元的内容减 1。DEC 指令不允许用立即数或段寄存器寻址方式。DEC 指令不影响进位标志，并且经常与 BYTE PTR、WORD PTR、DWORD PTR、QWORD PTR 一起使用。

5）比较指令（CMP）是特殊形式的减法指令。它不保存差值，而是用标志的变化来反映差的特征。比较指令用于比较任何寄存器（除段寄存器）或存储单元中的整个字节或整个字。新增的比较指令（CMPXCHG）是比较和交换指令的组合，用于 80486 ~ Core2 微处理器中。在 Pentium ~ Core2 微处理器中，CMPXCHG8B 指令比较和交换四字数据。在 Pentium 4 和 Core2 的 64 位模式中，COMPXCHG16B 指令是可用的。

6）乘法有字节、字或双字的，而且可以是有符号的（IMUL）或无符号的（MUL）。8 位乘法总是用操作数乘以 AL 寄存器内容，而积存于 AX 中。16 位乘法总是用操作数乘以 AX 寄存器内容，而积存于 DX-AX 中。32 位乘法总是用操作数乘以 EAX 寄存器内容，而积存于 EDX-EAX 中。80186 ~ Core2 中，有一条特殊的立即数 IMUL 指令，它包含 3 个操作数。如 IMUL BX，CX，3，该指令用 3 乘以 CX 的内容，而积放入 BX 中。在 Pentium 4 和 Core2 的 64 位模式中，乘法也是 64 位的。

7）除法有字节、字或双字的，而且可以是有符号的（IDIV）或无符号的（DIV）。对于 8 位除法，AX 寄存器内容除以操作数，然后商放在 AL 中，而余数放在 AH 中；16 位除法中，DX-AX 寄存器内容除以操作数，然后 AX 寄存器存放商而 DX 寄存器存放余数；32 位除法中，EDX-EAX 寄存器内容除操作数，然后 EAX 寄存器存放商而 EDX 寄存器存放余数。注意，有符号除法完成后，余数总是与被除数的符号相同。

8）压缩格式 BCD 数据进行加法和减法，由 DAA 调整加法的结果，由 DAS 调整减法的结果。通过 AAA、AAS、AAM 和 AAD 的调整操作，ASCII 数据可以进行加、减、乘或除。这些指令在 64 位模式下不可用。

9）AAM 指令具有另外一个很有意思的特性，可将二进制数转换为非压缩的 BCD 码。这条指令把 00H ~ 63H 之间的二进制数转换为 AX 中的非压缩的 BCD 码。AAM 指令将 AX 除以 10，并且把余数留在 AL 而商在 AH 中。

10）AND、OR 和 XOR 指令可以对存储在寄存器或存储单元中的字节、字或双字进行逻辑操作。这些指令改变全部的标志，而进位标志（C）和溢出（O）被清 0。

11）TEST 指令实现 AND 的操作，但是丢掉逻辑运算结果。这种指令通过改变标志位来指示测试的结果。

12）NOT 和 NEG 指令实现逻辑取反和算术取反。NOT 指令对操作数取 1 的补，而 NEG 指令对操作数取 2 的补。

13）有 8 种不同的移位和循环指令。这些指令中的每一个可对字节、字或双字寄存器或存储单元中的数据进行移位和循环。这些指令有两个操作数：第一个是被移位或循环的数据的地址，而第二个是计数移位或循环次数的立即数或

CL。如果第二个操作数是 CL，则 CL 保存循环或移位的次数。在 80386 ~ Core2 微处理器中，有两个增加的双精度移位指令（SHRD 和 SHLD）。

14）串扫描（SCAS）指令比较 AL、AX 或 EAX 的内容与由 DI 寻址的附加段存储单元的内容。

15）串比较（CMPS）指令比较两个存储区域字节、字或双字的内容。一个区域在由 DI 寻址的附加段中，而另一个区域在由 SI 寻址的数据段中。

16）SCAS 和 CMPS 指令，用 REPE 或 REPNE 前缀重复。REPE 前缀当等于条件满足时重复串指令，而 REPNE 前缀当不等条件满足时重复串指令。

5.8 习题

1. 为完成以下操作选择 ADD 指令：
   （a）BX 加到 AX
   （b）12H 加到 AL
   （c）EDI 加 EBP
   （d）22H 加到 CX
   （e）由 SI 寻址的数据加到 AL
   （f）CX 加到存储器地址 FROG 中存储的数据
   （g）234H 加到 RCX
2. ADD ECX，AX 指令的错误是什么？
3. 能够用 ADD 指令将 CX 内容加到 DS 中吗？
4. 如果 AX = 1001H，DX = 20FFH，执行 ADD AX，DX 指令以后，列出和及标志寄存器中每个位的内容（C、A、S、Z 和 O）。
5. 设计短指令序列，累加 AL、BL、CL、DL 和 AH 内容，并将和存入寄存器 DH。
6. 设计短指令序列，累加 AX、BX、CX、DX 和 SP 内容，并将和存入寄存器 DI。
7. 设计短指令序列，累加 ECX、EDX 和 ESI，并将和存入寄存器 EDI。
8. 设计短指令序列，累加 RCX、RDX 和 RSI，并将和存入寄存器 R12。
9. 选择指令，把 BX 内容到加到 DX 中，还要加上进位标志。
10. 挑选指令，使 SP 寄存器的内容加 1。
11. INC［BX］指令的错误是什么？
12. 为以下各减法选择 SUB 指令：
    （a）从 CX 中减去 BX 内容
    （b）从 DH 中减去 0EEH
    （c）从 SI 中减去 DI 内容
    （d）从 EBP 中减去 3322H
    （e）从 CH 中减去由 SI 寻址的数据
    （f）把由 SI 寻址的某单元的后面存放的 10 个字数据从 DX 中减去
    （g）从存储单元 FROG 中减去 AL 内容
    （h）从 R10 中减去 R9
13. 如果 DL = 0F3H，BH = 72H，列出从 DL 内容减去 BH 内容以后的差，并且给出标志寄存器各位的内容。
14. 写出短指令序列，从 AX 寄存器中减去 DI、SI 和 BP 中的数据，差存入寄存器 BX。
15. 选择一个指令，从 EBX 寄存器内容中减去 1。
16. 说明 SBB［DI - 4］，DX 指令实现什么功能？
17. 说明 SUB 和 CMP 指令之间的区别。
18. 当两个 8 位数相乘时，积放在哪里？
19. 当两个 16 位数相乘时，积放在哪两个寄存器中？指出哪个寄存器存放积的高有效位部分，哪个放积的低有效位部分。
20. 当两数相乘时，标志位 O 和 C 是什么样的？
21. MUL EDI 指令将积存放在哪里？
22. 写一短指令序列，求 DL 寄存器中 8 位数的三次方，起初将 5 装入 DL，要确保结果是 16 位的数字。
23. IMUL 和 MUL 指令的区别是什么？
24. 说明 IMUL BX，DX，100H 指令的操作。
25. 当执行 8 位数除法指令时，被除数放在哪个寄存器中？
26. 当执行 16 位数除法指令时，商放在哪个寄存器中？
27. 当执行 64 位数除法指令时，商放在哪个寄存器中？
28. 除法期间，能检测出哪种类型错误？
29. 说明 IDIV 和 DIV 指令之间的区别。
30. 执行 8 位数除法指令后，余数放在哪里？
31. 执行 64 位数除法指令后，商放在哪里？
32. 写出一个短指令序列。用 BL 中的数据除以 CL 中的数据，然后将结果乘以 2。最后的结果是存入 DX 寄存器中的 16 位数。
33. BCD 码算术运算使用哪些指令？
34. 解释 AAM 指令怎样将二进制数转换为 BCD 码。
35. ASCII 码算术运算使用哪些指令？
36. 设计短指令序列，使 AX 中的无符号数字（值为 0 ~ 65535）转换为 5 位 BCD 数据，并且存入起始位置由 BX 寄存器寻址的数据段存储器中。注意先存储高位字符，不要删除前面的 0。
37. 设计一个短指令序列。AX 和 BX 中的 8 位 BCD 数加 CX 和 DX 中的 8 位 BCD 数（AX 和 CX 是最高有效寄存器）。加法以后结果必须存入 CX 和 DX 中。
38. AAM 指令在 64 位模式中起作用吗？
39. 为下列各操作选择 AND 指令：
    （a）BX 与 DX，结果存入 BX
    （b）0EAH 与 DH
    （c）DI 与 BP，结果存入 DI 中
    （d）1122H 与 EAX
    （e）由 BP 寻址的存储单元的数据和 CX 相与，而结果存入存储单元中

(f) 把由 SI 寻址的某存储单元的前面存放的四个字数据和 DX 相与，结果存入 DX 中

(g) AL 和 WHAT 存储单元中的内容相与，结果存入 WHAT 单元

40. 设计短指令序列，将 DH 中的最左3 位清0，而不改变 DH 中的其他位，结果存入 BH 中。

41. 为下列各操作选择 OR 指令：

(a) BL 或 AH，结果存入 AH 中

(b) 88H 或 ECX

(c) DX 或 SI，结果存入 SI 中

(d) 1122H 或 BP

(e) BX 寻址的数据或 CX，结果放入存储单元中

(f) 由 BP 寻址的某存储单元的后面存放的40 个字节与 AL 相或，结果存入 AL 中

(g) AH 与存储单元 WHEN 相或，结束存入 WHEN 中

42. 设计短指令序列，将 DI 中的最右 5 位置 1，而不改变 DI 中的其他位，结果存入 SI 中。

43. 为下列各操作选择 XOR 指令：

(a) BH 内容与 AH 内容“异或”，结果存入 AH 中

(b) 99H 与 CL 内容“异或”

(c) DX 内容与 DI 内容“异或”，结果存入 DX 中

(d) 1A23H 与 RSP 内容“异或”

(e) 由 EBX 寻址的数据与 DX 内容“异或”，结果存入存储单元中

(f) 由 BP 寻址的某存储单元之后存放的 30 个字数据和 DI “异或”，结果存入 DI 中

(g) DI 与存储单元 WELL “异或”，结果存入 DI 中

44. 设计短指令序列，把 AX 中的最右 4 位置位（1），将 AX 中的最左 3 位清 0，并且把 AX 中的 7、8、9 位取反。

45. 说明 AND 与 TEST 指令之间的区别。

46. 为测试寄存器 CH 中的第 2 位，选择指令。

47. NOT 和 NEG 指令的区别是什么？

48. 选择正确的指令实现以下的任务：

(a) DI 右移 3 位，并把零移入最高位

(b) AL 中所有位左移 1 位，使 0 移入最低位

(c) AL 循环左移 3 位

(d) EDX 带进位位循环右移 1 位

(e) DH 寄存器右移 1 位，并且使结果的符号位与原数符号相同

49. SCASB 指令完成的操作是什么？

50. 对于串指令，DI 总是用来寻址_____段中的数据。

51. 标志位 D 的作用是什么？

52. 解释 REPE 前缀与 SCASB 指令结合可实现什么功能？

53. 什么条件将终止 REPE SCASB 串指令的重复？

54. 说明 CMPSB 指令可实现什么功能。

55. 设计指令序列，为了检索 66H，扫描位于数据段内的 300 个字节长的存储区 LIST。

56. 如果 AH = 02H，DL = 43H，执行 INT 21H 指令时将发生什么？

第6章　程序控制指令

引言

程序控制指令用于引导和改变程序的流程。程序流向的改变通常发生在判定以后，也就是由 CMP 或 TEST 指令后面的条件转移指令来实现。本章介绍程序控制指令，包括：转移指令、调用指令、返回指令、中断指令和机器控制指令。

另外，本章还介绍了条件汇编语句（.IF、.ELSE、.ELSEIF、.ENDIF、.WHILE、.ENDW、.REPEAT、和.UNTIL），这些语句在 MASM 6.X 及更高版本中或者在与 MASM 兼容的 TASM 5.X 版中都是有效的。这些条件汇编命令使程序员能像使用 C/C++ 语言那样去写程序控制流程。

目的

读者学习完本章后将能够：

1）使用条件和无条件转移指令控制程序的流程。
2）在程序中使用条件汇编语句 .IF、.REPEAT、.WHILE 等。
3）使用调用和返回指令在程序结构中嵌入过程。
4）解释中断和中断控制指令的操作。
5）使用机器控制指令修改标志位。
6）使用 ENTER 和 LEAVE 进入和退出编程结构。

6.1　转移指令

转移指令（**jump**，**JMP**）是重要的程序控制指令，它允许程序员跳过一段程序，跳转到存储器的任何位置执行下一条指令。条件转移则允许程序员根据对数值的测试做出决定。这些数值测试的结果保存在标志位中，再由条件转移指令检测它们。类似于条件转移指令的另外一条指令，即条件设置指令，也在本节中说明。

本节中的所有转移指令都通过举例程序来说明，也再次涉及了在第 3 章中首先出现的 LOOP 和条件 LOOP 指令。因为它们也是跳转指令的一种形式。

6.1.1　无条件转移指令

微处理器可以使用三种类型的无条件转移指令（见图 6-1）：短转移、近转移和远转移。**短转移**（**short jump**）是两字节指令，允许分支或转移到相对于当前指令地址 + 127 和 - 128 字节范围以内的某个存储单元。3 字节的**近转移**（**near jump**）指令允许分支或转移到代码段内当前指令 ±32KB 范围以内（即当前代码段内的任何位置）。记住，段实际上是周期性的，这意味着偏移地址 FFFFH 的上一个位置是偏移地址 0000H。由于这个原因，如果指令指针指向偏移地址 FFFFH，而要转移到存储器中的后两个字节，则程序流在偏移地址 0001H 处继续。因此 ±32KB 的位移量允许转移到当前代码段内的任何位置。5 字节的**远转移**（**far jump**）允许转移到整个实存储系统内的任何内存单元。短的和近的转移通常称

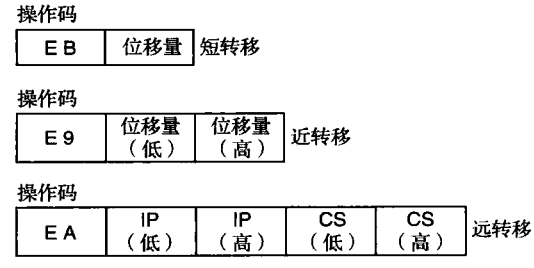


图 6-1　转移指令的三种主要格式。注意偏移量是 8 位或 16 位带符号的位移量或距离

为**段内转移**（**intrasegment jump**），而远转移通常称为**段间转移**（**intersegment jump**）。

在80386～Core2微处理器中，如果机器按保护模式运行，有4GB的代码段，则近转移是在±2GB范围内；如果是按实模式运行，则近转移是在±32KB范围内。在保护模式中，80386及更高型号的微处理器使用32位的位移量，图6-1中没有表示出。如果Pentium 4在64位模式下操作，那么允许转移到1TB存储空间的任何地址。

短转移

短转移也称为**相对转移**（**relative jump**），因为它们可以与相关的软件一起移动到当前代码段内的任何位置而无需更改。这是因为转移地址不与操作码一起存储。替代转移地址的是操作码后面的**距离**（**distance**），即位移量。短转移指令的位移量是用一个字节的有符号数表示的距离，这个值的范围是+127到-128。短转移指令表示在图6-2中。当微处理器执行短转移时，位移量先被符号扩展，然后加到指令指针上（IP/EIP），从而得到当前代码段内的转移地址。短转移指令分支到这个新的地址，即程序下条指令的地址。

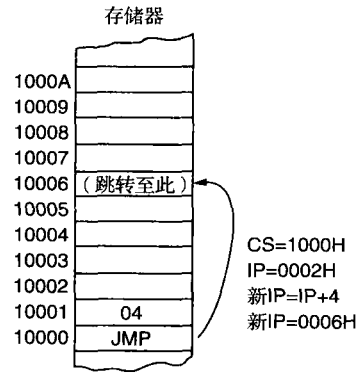


图6-2 短转移到越过下条指令地址4个存储单元的位置

例6-1指出短转移指令怎样控制从程序的一个部分转到另一个部分。也说明了和转移指令一起的**标号**（存储器地址的符号名）的用法。注意，第一条转移指令（JMP SHORT NEXT）用了SHORT伪指令强制进行短转移，而其他转移指令就没用。例6-1中的第二条转移指令（JMP START）是多数汇编程序常采用的转移指令格式，也按短转移汇编。如果下一条指令的地址（0009H）加上第一条转移指令的符号扩展位移量（0017H），则得NEXT的地址为0017H+0009H，即0020H处。

例6-1

```
0000 33 DB            XOR  BX,BX
0002 B8 0001   START: MOV  AX,1
0005 03 C3            ADD  AX,BX
0007 EB 17            JMP  SHORT NEXT

                <跳过的存储单元>

0020 8B D8     NEXT:  MOV  BX,AX
0022 EB DE            JMP  START
```

每当转移指令引用地址时，常用标号代表这个地址，JMP NEXT就是个例子，它转移到标号NEXT确认的下条指令。转移指令极少使用实际的十六进制地址，但是汇编程序支持使用$+a位移量，即相对于指令指针的寻址。例如，JMP $+2就是相对于JMP指令向后越过两个存储单元。标号NEXT后面必须有冒号（NEXT:），以便转移指令引用它。如果标号后面没有冒号，就不能转移到该标号那去。注意，只有当标号要被转移指令或调用指令引用时，标号后面才需用冒号。在Visual C++中也是如此。

近转移

除了距离较大以外，近转移类似于短转移。近转移指令控制转移到当前代码段内距离该转移指令±32KB范围内，而在保护模式下的80386及更高型号的微处理器中是±2GB范围内。近转移是3字节指令，包括操作码和它后面的带符号的16位的位移量。在80386～Pentium 4微处理器中，位移量是32位，因此近转移指令是5字节长。带符号的位移量加到指令指针（IP）上产生转移地址。由于带符号的位移量是在±32KB范围内，因此近转移可以转移到实模式当前代码段内的任何位置。在80386及更高型号的微处理器中，保护模式下的代码段长度达4GB，因此32位的位移量允许近转移到±2GB范围内的任何位置。图6-3说明了实模式近转移指令的操作。

近转移类似于短转移，也是可重定位的，因为它也是相对转移。如果代码段移到存储器新的位置，转移指令与操作数之间的距离保持不变，就允许通过简单地移动代码段实现重定位。这个特性与可重定位数据段一起使得 Intel 系列微处理器完美地用于通用计算机系统。由于相对转移及可重定位数据段的原因，可把软件写成定位到存储器的任何位置，而不改变其功能。

例 6-2 给出了与例 6-1 相同的基本程序，只是转移的距离大些。第一个转移（JMP NEXT）把控制传递到代码段内存偏移地址 0200H 处的指令。注意，指令汇编成 E9 0200 R。字母 R 指示 0200H 为**可重定位转移**（**relocatable jump**）地址。可重定位的 0200H 地址只由汇编程序内部使用。实际汇编成的机器语言指令为 E9 F6 01，没有出现在汇编程序列表中。对应这个转移的实际的位移量是 01F6H。汇编程序列表转移地址为 0200 R，使得这个地址在软件开发中更容易理解。如果连接完成后的执行文件（.EXE）或命令文件（.COM）按十六进制显示，转移指令则以 E9 F6 01 形式出现。

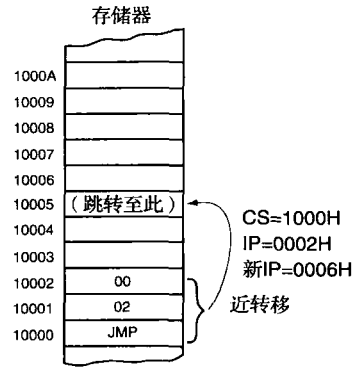


图 6-3 近转移到位移量（0002H）加 IP 的内容处

例 6-2

```
0000  33  DB              XOR  BX,BX
0002  B8  0001      START: MOV  AX,1
0005  03  C3              ADD  AX,BX
0007  E9  0200  R          JMP  NEXT

                  <跳过的存储单元>

0200  8B D8         NEXT:  MOV  BX,AX
0202  E9  0002  R          JMP  START
```

远转移

远转移（见图 6-4）从指令中得到新的段地址和偏移地址，以实现转移。这个 5 字节指令的第 2 和第 3 字节存放新的偏移地址，而第 4 和第 5 字节存放新的段地址。如果微处理器（80286 ～ Core2）按保护模式操作，段地址寻址包含远转移段基地址的描述符。16 位或 32 位的偏移地址是新代码段内的偏移地址。

例 6-3 给出了使用远转移指令的短程序。远转移指令有时用 FAR PTR 伪指令作为说明。获得远转移的另一种方法是定义标号为**远标号**（**far label**）。如果标号是当前代码段或过程外面的，它就是远标号。这个例子中的 JMP UP 指令引用了远标号。标号 UP 用 EXTERN UP: FAR 定义为远标号。**外部标号**（**external label**）出现在几个程序中，这些程序包含在多个程序文件中。定义全局标号的另一种方式是用双冒号（LABEL::）替换标号后面的单冒号。如果从外部过程块访问内部定义为近过程的标号，就需要这样定义。

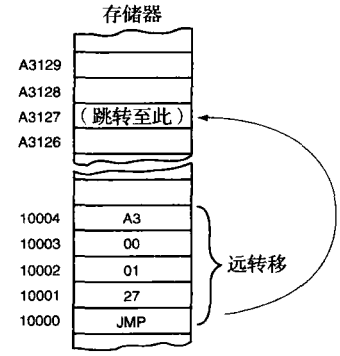


图 6-4 远转移指令，用操作码后面的 4 个字节替换 CS 和 IP 的内容

例 6-3

```
                        EXTRN  UP:FAR

0000  33  DB              XOR  BX,BX
0002  B8  0001      START: ADD  AX,1
```

```
0005  E9  0200 R              JMP   NEXT
                    跳过的存储单元

0200  8B  D8          NEXT:   MOV   BX,AX
0202  EA  0002----R           JMP   FAR PTR START

0207  EA  0000----R           JMP   UP
```

当对程序文件进行连接时，连接程序把标号 UP 的地址插入 JMP UP 指令，也将段地址插入 JMP START 指令。JMP FAR PTR START 中的段地址，按照----R 的形式列出，表示可重定位；JMP UP 中的段地址，按----E 列出，表示是外部地址。当连接或拼接程序文件时，两种情况中的----由连接程序填入。

使用寄存器操作数的转移

转移指令也可以用 16 位或 32 位寄存器作为操作数，这就自动将指令设置为**间接转移**（**indirect jump**），转移地址在转移指令指定的寄存器内。这与近转移的位移量不同，寄存器的内容直接传送到指令指针中，而不像短转移和近转移那样把位移量加到指令指针中。例如 JMP AX 指令，当出现转移时把 AX 的内容复制到 IP 中。这就允许转移到当前代码段内的任何位置。对于 80386 及更高型号的微处理器，JMP EAX 指令转移到当前代码段内的任何位置。区别是，在保护模式中代码段的长度可以达到 4GB 长，所以要求用 32 位偏移地址。

例 6-4 给出了 JMP AX 指令怎样访问代码段中的转移表。这个 DOS 程序从键盘读入键值，然后修正这个 ASCII 码。如果键入 1、2 或 3，是 AL 中的 00H、01H 或 02H 分别对应的 1、2 或 3，AH 清除为 00H。因为转移表中包含 16 位的偏移地址，为了存取转移表中 16 位的地址，AX 的内容被加倍成 0、2 或 4。然后，转移表的起始地址装入 SI，与 AX 相加形成指向转移目标的地址。MOV AX，[SI] 指令从转移表中获取地址，这样 JMP AX 指令能转移到存储在转移表中（1、2 或 3）的地址。

例 6-4

```
                    ;程序从键盘读取输入命令 1、2 或 3。
                    ;用转移表显示数字 1、2 或 3。
                    ;
                    .MODEL SMALL                          ;选择 SMALL 模型
0000                .DATA                                 ;数据段开始
0000  0030  R       TABLE: DW        ONE                  ;定义转移表
0002  0034  R              DW        TWO
0004  0038  R              DW        THREE
0000                .CODE                                 ;代码段开始
                    .STARTUP                              ;程序开始
0017  B4  01        TOP:   MOV       AH,1                 ;读键盘，放入 AL
0019  CD  21               INT       21H
001B  2C  31               SUB       AL,31                ;变换为 BCD 码
001D  72  F9               JB        TOP                  ;如果键值 <1
001F  32  02               CMP       AL,2
0021  77  F4               JA        TOP                  ;如果键值 >3
0023  B4  00               MOV       AH,0                 ;键码值加倍
0025  03  C0               ADD       AX,AX
0027  BE  0000  R          MOV       SI,OFFSET TABLE      ;地址 TABLE
002A  03  F0               ADD       SI,AX                ;从转移表地址
002C  8B  04               MOV       AX,[SI]              ;获取 ONE、TWO 或;THREE 的地址
002E  FF  E0               JMP       AX                   ;转移到 ONE、TWO 或 THREE
0030  B2  31        ONE:   MOV       DL,'1'               ;取 ASCII 码 1
0032  EB  06               JMP       BOT
0034  B2  32        TWO:   MOV       DL,'2'               ;取 ASCII 码 2
0036  EB  02               JMP       BOT
```

```
0038 B2 33        THREE: MOV       DL,'3'          ;取 ASCII 码 3
003A B4 02        BOT:   MOV       AH,2            ;显示数据
003C CD 21               INT       21H
                  .EXIT
                  END
```

使用变址寻址方式的间接转移

转移指令也可以使用[ ]寻址方式直接访问转移表。转移表中可包含近间接转移的偏移地址或者远间接转移的段和偏移地址（如果寄存器转移称为间接转移，这种转移类型也可以理解为是**双间接转移**）。除非用 FAR PTR 伪指令指明是远转移指令，否则汇编程序假定是近转移。例 6-5 用 JMP TABLE [SI]指令替换例 6-4 的 JMP AX 指令，这样就减少了程序的长度。

例 6-5

```
                  ;程序读从键盘输入命令 1、2 或 3。
                  ;用转移表显示数字 1、2 或 3。
                  ;
                  .MODEL SMALL                ;选择 SMALL 模型
0000              .DATA                       ;数据段开始
0000 002D R       TABLE: DW   ONE             ;转移表
0002 0031 R              DW   TWO
0004 0035 R              DW   THREE
0000              .CODE                       ;代码段开始
                  .STARTUP                    ;程序开始
0017 B4 01        TOP:   MOV  AH,1            ;读键盘输入，并且放入 AL
0019 CD 21               INT  21H
001B 2C 31               SUB  AL,31          ;变换为 BCD 码
001D 72 F9               JB   TOP            ;如果键值 <1
001F 32 02               CMP  AL,2
0021 77 F4               JA   TOP            ;如果键值 >3
0023 B4 00               MOV  AH,0           ;键码值加倍
0025 03 C0               ADD  AX,AX
0027 B5 F0               MOV  SI,AX          ;从转移表地址
0029 FF A4 0000 R        JMP  TABLE [SI]     ;转移到 ONE、TWO、THREE
002D B2 31        ONE:   MOV  DL,'1'         ;取 ASCII 码 1
002F EB 06               JMP  BOT
0031 B2 32        TWO:   MOV  DL,'2'         ;取 ASCII 码 2
0033 EB 02               JMP  BOT
0035 B2 33        THREE: MOV  DL,'3'         ;取 ASCII 码 3
0037 B4 02        BOT:   MOV  AH,2           ;显示数据
0039 CD 21               INT  21H
                  .EXIT
                  END
```

访问存储器表使用的机制与访问正常的存储器完全相同。JMP TABLE[SI]指令指向一个转移地址，它存放在代码段内并用 SI 作为偏移地址来访问，然后转移到这个存储单元存放的地址。不论寄存器转移指令还是间接变址转移指令，通常寻址 16 位位移量。这就意味着两种类型的转移都是近转移。如果程序中出现 JMP FAR PTR[SI]指令，或者用 DD 伪指令定义 TABLE 数据的 JMP TABLE[SI]指令，则微处理器认为转移表中包含 32 位的双字地址（IP 和 CS）。

6.1.2 条件转移和条件设置

8086～80286 微处理器的条件转移指令都是短转移。这就把条件转移的范围限制在相对条件转移指令位置的 +127 到 -128 字节以内。80386 以上的微处理器，条件转移是短转移或是近转移（在 ±32KB范围内），在 Pentium 4 的 64 位模式下，条件转移是近转移（范围是 ±2GB），因此允许这些微

处理器有条件地转移到当前代码段内的任何位置。表 6-1 列出了带有测试条件的全部条件转移指令。注意，如果距离太大，Microsoft 公司的 MASM 6. X 汇编程序自动调整条件转移。

表 6-1 条件转移指令

汇 编 语 句	测试的条件	操 作
JA	Z = 0 和 C = 0	高于转移
JAE	C = 0	高于或等于转移
JB	C = 1	低于转移
JBE	Z = 1 或 C = 1	低于或等于转移
JC	C = 1	进位置位转移
JE 或 JZ	Z = 1	等于转移或零转移
JG	Z = 0 和 S = 0	大于转移
JGE	S = 0	大于或等于转移
JL	S! = 0	小于转移
JLE	Z = 1 或 S! = 0	小于或等于转移
JNC	C = 0	进位位清除转移
JNE 或 JNZ	Z = 0	不等于转移或非零转移
JNO	O = 0	无溢出转移
JNS	S = 0	符号位为零转移
JNP 或 JPO	P = 0	无奇偶或奇偶位为奇转移
JO	O = 1	溢出位置位转移
JP 或 JPE	P = 1	奇偶位置位或奇偶位为偶转移
JS	S = 1	符号位置位转移
JCXZ	CX = 0	CX = 0 转移
JECXZ	ECX = 0	ECX = 0 转移
JRCXZ	RCX = 0	RCX = 0 转移（64 位）

条件转移指令测试以下标志位：符号（S）、零（Z）、进位（C）、奇偶（P）和溢出（O）。如果条件是真，分支到与转移指令相关联的标号；如果条件是假，顺序执行程序中的下一条指令。例如 JC，表示如果有进位则转移。

多数条件转移指令是简单明了的，因为通常它们只测试一个标志位，但也有一些条件转移指令测试多个标志位。比较两个数数值相对大小的条件转移指令，就要复杂些，要测试多于一个标志位。

由于程序设计中使用了有符号数和无符号数，还由于两类数字的排序不同，有两套用于比较大小的条件转移指令。图 6-5 给出有符号的和无符号的两种 8 位数字的排序。以此类推，16 位及 32 位数字与 8 位数字同样的方式排序，只是它们更长些。注意，在无符号数字集合中，FFH（255）大于 00H，而有符号数字 FFH（-1）小于 00H。因此，无符号数中 FFH 大于 00H，而有符号数中 FFH 小于 00H。

比较有符号数字时用 JG、JL、JGE、JLE、JE 和 JNE 指令。有符号数使用术语“大于”或“小于”。比较无符号数字时用 JA、JB、JAE、JBE、JE 和 JNE 指令。无符号数字使用术语“高于”或“低于”。

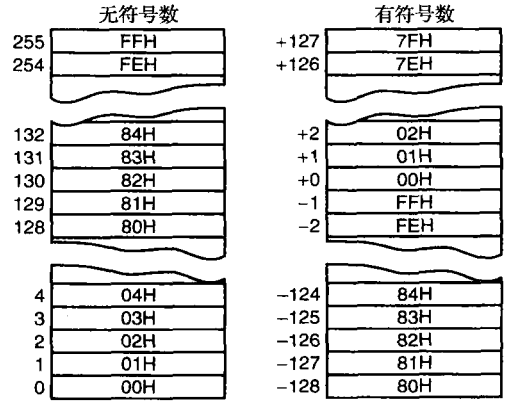


图 6-5 有符号数字和无符号数字排序的区别

剩下的条件转移测试单个标志位，诸如溢出、奇偶等。注意 JE 有一个替换的操作码 JZ。所有指令都有可替代的指令，但是因为它们不适合测试，多数在程序设计中不用（在附录 B 的指令系统表中给

出了这些替代指令)。例如，JA 指令（高于转移）有时替代为 JNBE（不低于或等于转移）。JA 的功能与 JNBE 完全相同，但是比较时 JNBE 比 JA 更麻烦。

除了 JCXZ（CX = 0 转移）和 JECXZ（ECX = 0 转移）以外，全部条件转移指令都测试标志位。而 JCXZ 直接测试 CX 寄存器的内容，JECXZ 测试 ECX 寄存器的内容，不测试标志位也不影响标志位。对于 JCXZ 指令，如果 CX = 0，则出现转移，如果 CX! = 0，则不出现转移。对于 JECXZ 指令也是类似的：如果 ECX = 0，出现转移；如果 ECX! = 0，则不转移。Pentium 4 或 Core2 在 64 位操作下，JRCXZ 指令转移的条件是 RCX = 0。

例 6-6 给出了使用 JCXZ 的程序。程序中 SCASB 指令搜索表内的 0AH，进行搜索后，JCXZ 指令测试 CX，检查它的内容是否为零。如果内容是零，则在表内没有找到 0AH。在这个例子中使用进位标志向调用程序传递没有找到的条件。测试是否找到数据的另一种方法是使用 JNE 指令。如果使用 JNE 替换 JCXZ 指令，可实现相同的功能。执行 SCASB 指令以后，如果在表内没有找到数据，标志指示不等条件。

例 6-6

```
                    ;指令在 100 个字节长的表内搜索 0AH
                    ;假定 TABLE 的偏移地址经在 SI 中
                    ;
0017  B9  0064          MOV  CX,100        ;装入计数值 100
001A  B0  0A            MOV  AL,0AH        ;0AH 装入 AL
001C  FC                CLD                ;自动递增
001D  F2/AE             REPNE SCASB        ;为搜索 0AH
001F  F9                STC                ;找到,进位位置位
0020  E3  01            JCXZ NOT_FOUND     ;如果没有找到，转移
0022                NOT_FOUND
```

条件设置指令

除了条件转移指令以外，80386 ～ Core2 还包括条件设置指令。条件转移指令要测试的条件可以由条件设置指令来建立。条件设置指令或者把一个字节设置为 01H，或者把该字节清除为 00H，这取决于对条件测试的结果。表 6-2 列出了各种条件设置指令的格式。

表 6-2 条件设置指令

汇 编 语 句	测试的条件	操 作
SETA	C = 0 或 Z = 0	高于则设置字节
SETAE	C = 0	高于或等于则设置字节
SETB	C = 1	如果低于则设置字节
SETBE	C = 1 或 Z = 1	低于或等于则设置字节
SETC	C = 1	有进位则设置字节
SETE 或 SETZ	Z = 1	等于设置字节/是零设置字节
SETG	Z = 0 和 S = 0	大于则设置字节
SETGE	S = 0	大于或等于则设置字节
SETL	S! = 0	小于设置字节
SETLE	Z = 1 或 S! = 0	小于或等于设置字节
SETNC	C = 0	无进位则设置字节
SETNE 或 SETNZ	Z = 0	不等于设置字节/不是零设置字节
SETNO	O = 0	无溢出则设置字节
SETNS	S = 0	无符号（为正）则设置字节
SETNP 或 SETPO	P = 0	奇偶为奇则设置字节
SETO	O = 1	有溢出则设置字节
SETP 或 SETPE	P = 1	奇偶为偶则设置字节
SETS	S = 1	有符号（为负）则设置字节

当必须在后面程序的某一点测试一个条件时，这些指令很有用。例如，在程序的某一点使用 SETNC MEM 指令使一个字节置位，以指示进位位清 0。如果进位位清 0，这条指令将 01H 放入存储单元 MEM；如果进位位置位，则放入 00H。程序执行 SETNC MEM 指令后，在后面的某个地方就可以测试 MEM 的内容，以便判定在前面 SETNC MEM 指令执行的那一点进位位是否清除 0。

6.1.3 LOOP 指令

LOOP 指令是 CX 减 1 和 JNZ 组合而成的条件转移指令。在 8086 ～ 80286 中，LOOP 使 CX 内容减 1，如果 CX 内容不等于零，它转移到标号指示的地址；如果 CX 为零则执行下一条指令。在 80386 及更高型号的微处理器中，LOOP 指令使 CX 或使 ECX 减 1，这取决于指令的模式。如果 80386 ～ Core2 按 16 位指令模式操作，LOOP 指令使用 CX。如果它们按 32 位指令模式操作，LOOP 指令用 ECX。在 80386 ～ Core2 中通过 LOOPW（使用 CX）和 LOOPD（使用 ECX）指令改变这种默认的情况。在 64 位模式下，循环计数使用 64 位宽的 RCX。

例 6-7 指出怎样使用 LOOP 控制多个数字相加，实现一个存储器块（BLOCK1）的数据与第二个存储器块（BLOCK2）的数据相加。LODSW 和 STOSW 指令访问数据块 1 和数据块 2 中的数据。ADD AX，ES:[DI] 指令访问附加段中位于 BLOCK2 中的数据。用 DI 为 STOSW 指令寻址附加段的数据，是因为 BLOCK2 在附加段中。.STARTUP 伪指令只是将数据段的地址装入 DS。在这个例子中，附加段也寻址数据段中的数据，为此将 DS 的内容通过累加器复制到 ES 中。可惜的是，没有直接从段寄存器到段寄存器传送的指令。

例 6-7

```
                    ;实现一个存储器块（BLOCK1）的数据与第二个存储器块（BLOCK2）的数
                    ;据相加，结果存入第二个存储器块（BLOCK2）。
                    ;
                    .MODEL SMALL                    ;选择 SMALL 模型
0000                .DATA                           ;指示数据段开始
0000 0064 [         BLOCK1 DW  100 DUP (?)          ;为 BLOCK1 保留 100 个字
          0000
             ]
00C8 0064 [         BLOCK2 DW  100 DUP (?)          ;为 BLOCK2 保留 100 个字
          0000
             ]
0000                .CODE                           ;指示代码段开始
                    .STARTUP                        ;指示程序开始
0017 8C D8                MOV AX,DS                 ;初始化 DS 和 ES
0019 8E C0                MOV ES,AX
001B FC                   CLD                       ;选择加 1
001C B9 0064              MOV CX,100                ;装入计数值 100
001F BE 0000 R            MOV SI,OFFSET BLOCK1      ;BLOCK1 的地址
0022 BF 00C8 R            MOV DI,OFFSET BLOCK2      ;BLOCK2 的地址
0025 AD             L1:   LODSW                     ;将 BLOCK1 的数据装入 AX
0026 26: 03 05            ADD AX,ES:[DI]            ;将 BLOCK2 的数据加到 AX
0029 AB                   STOSW                     ;和存入 BLOCK2
002A E2 F9                LOOP L1                   ;重复 100 遍
                    .EXIT
                    END
```

条件 LOOP 指令

类似于 REP，条件 LOOP 指令的格式也有：LOOPE 和 LOOPNE。如果 CX 不等于零而且等于条件成立，则 LOOPE（等于则循环）指令转移；如果不等条件成立或者 CX 寄存器减 1 后为零，则跳出循环。如果 CX 不等于零而且不等于条件存在，LOOPNE（不等于则循环）指令转移；如果等于条件成立或者 CX 寄存器减到了零，则跳出循环。在 80386 ～ Core2 中，条件 LOOP 指令可以用 CX 或 ECX 作为计数

器。如果需要，也可以用 LOOPEW、LOOPED、LOOPNEW 或 LOOPNED 指令替换 LOOP 指令。在 64 位模式下，循环计数使用 64 位宽的 RCX。

LOOPE 和 LOOPNE 存在着替换指令：LOOPE 与 LOOPZ 一样，而 LOOPNE 与 LOOPNZ 相同。但大多数程序中只使用 LOOPE 和 LOOPNE。

6.2 控制汇编语言程序的流程

使用汇编语言语句 .IF、.ELSE、.ELSEIF 和 .ENDIF 比使用条件转移语句更容易控制程序流程，这些语句用于为 MASM 指示特殊的汇编语言命令。注意，这种以句点开始的控制程序流程的语句，仅适合于 MASM 6.X 版本，不能用于诸如 5.10 等早期版本。这一节介绍的语句还有 .REPEAT-.UNTIL 和 .WHILE-.ENDW 语句。当用 Visual C++ 内嵌汇编时，这些语句（**带句点的命令**）不起作用。

例 6-8a 表示如何用这些语句通过测试 AL 内容是否在 ASCII 字符 'A' ～ 'F' 之间，从而控制程序流程。

例 6-8 a

```
.IF AL >= 'A'  &&  AL <= 'F'
      SUB  AL,7
.ENDIF
SUB AL,30H
```

例 6-8 b

```
char temp;
_asm{
      mov  al,temp
      cmp  al,41h
      jb   Later
      cmp  al,46h
      ja   Later
      sub  al,7
Later:
      sub  al,30h
      mov  temp,al
}
```

通常会碰到用 Visual C++ 内嵌汇编完成某些任务，宁可在 Visual C++ 中完成也不要在汇编语言中完成。例 6-8b 表示在 Visual C++ 中用内嵌汇编并以汇编语言写条件转移来完成同样的任务，也显示了在 Visual C++ 中的汇编块里如何使用标号。这个例子说明，除了不能用句点命令外，它完成同样任务更加困难。在汇编语言中绝对不要使用大写字母写汇编指令，因为某些指令是为 C++ 保留的，可能会引起问题。

例 6-8a 中，注意在 .IF 语句里符号 && 代表 AND 功能。例 6-8b 中没有 .IF 语句，因为用几个同样操作的比较（CMP）指令完成了这一任务。与使用 .IF 语句相关的关系操作符完整列表参见表 6-3。注意，许多这样的条件（例如 &&）也用于众多的高级语言，如 C 和 C++ 中。

表 6-3 用于 .IF 语句的关系运算符

运 算 符	功 能	运 算 符	功 能
==	等于或相同	&	位测试
!=	不等于	!	逻辑“非”
>	大于	&&	逻辑“与”
>=	大于或等于	||	逻辑“或”
<	小于	|	“或”
<=	小于或等于		

例6-9则举出了使用条件.IF语句的另一个例子，把所有ASCII码字母转换为大写。首先利用DOS中断21H中06H号功能调用无回显地读键盘输入字符。然后，如果必要，用.IF语句转换字符为大写。在该例中，逻辑AND（&&）用来判定一个字符是否是小写。如果是小写则减去20H变为大写。这个程序从键盘读入一个键，并在显示之前将其转换成大写。注意，当接收到Ctrl + C（ASCII码 = 03H）键时，程序怎样终止。使用.LISTALL伪指令将使汇编时的各种情况均被列表显示，包括由.STARTUP伪指令产生的标号@ startup。.EXIT伪指令也用.LISTALL展开，以给出DOS INT 21H中4CH号功能调用，返回DOS。

例6-9

```
                ;本DOS程序用以从键盘上输入字符,并在显示之前转换全部小写字母为大写字母。
                ;
                ;本程序用Ctrl-C键终止运行。
                ;
                .MODEL TINY                ;选择TINY模型
                .LISTALL                   ;列出全部汇编后产生的语句
0000            .CODE                      ;代码段开始
                .STARTUP                   ;程序开始
0100            * @ Startup
0100 B4 06      MAIN1:MOV  AH,6            ;读键码，但不显示
0102 B2 FF            MOV  DL,0FFH
0104 CD 21            INT  21H
0106 74 F8            JE   MAIN1           ;若没有键按下
0108 3C 03            CMP  AL,3            ;是Ctrl-C键吗?
010A 74 10            JE   MAIN2           ;是Ctrl-C键则转向MAIN2

                      .IF  AL >= 'a' && AL <= 'z'

010C 3C 61    *               cmp  al,'a'
010E 72 06    *               jb   @C0001
0110 3C 7A    *               cmp  al,'z'
0112 77 02    *               ja   @C0001
0114 2C 20                    SUB  AL,20H

                      .ENDIF

0116            * @C0001:
0116 8A D0            MOV  DL,AL           ;显示字符
0118 CD 21            INT  21H
011A EB E4            JMP  MAIN1           ;重复
011C              MAIN2:
                  .EXIT
011C B4 4C    *       MOV  AH,4CH
011E CD 21    *       INT  21H
                  END
```

在这段程序中，根据.IF AL >= 'a'&& AL <= 'z'语句将一个小写字母转换成大写，若AL中的值大于等于a并小于等于z（即为a～z之间的值），则执行.IF和.ENDIF之间的指令。该语句（SUB AL, 20H）即将AL中的数减去20H，从而把小写字母转换成大写字母。注意汇编程序是如何处理.IF语句的（见前面带有*号标注的行），其中标号@C0001是汇编程序产生的，由程序里.IF语句处的条件跳转指令使用。

例6-10是另外一个使用.IF条件语句的实例，该程序从键盘上读入一个字符并且转换成十六进制数，这个程序未以展开的形式列出。

例 6-10 a

```
                    ;本程序从键盘上输入数字并将其转换成
                    ;十六进制数存到内存地址 TEMP 处。
                    ;
                    .MODEL SMALL                          ;选择 SMALL 模型
0000                .DATA                                 ;数据段开始
0000 00             TEMP DB     ?                         ;定义 TEMP
0000                .CODE                                 ;代码段开始
                    .STARTUP                              ;程序开始
0017 B4 01              MOV  AH,1                         ;读入键的代码
0019 CD 21              INT  21H
                        .IF AL >= 'a' && AL <= 'f'        ;如果是小写字母 a ~ f,则减去 57H
0023 2C 57                  SUB  AL,57H
                        .ELSEIF .IF AL >= 'A' && AL <= 'F';如果是大写字母 A ~ F,则减去 37H
002F 2C 37                  SUB  AL,37H
                        .ELSE
0033 2C 30                  SUB  AL,30H                   ;否则,则减去 30H
                        .ENDIF
0035 A2 0000 R          MOV  TEMP,AL                      ;在内存地址 TEMP 处保存
                    .EXIT                                 ;返回 DOS
                    END                                   ;文件结束
```

例 6-10 b

```
char Convert (char temp)
{
      if (temp >= 'a' && temp <= 'f')
            temp -= 0x57;
      else if (temp >= 'A' && temp <= 'F')
            temp -= 0x37;
      else
            temp -= 0x30;
      return temp;
}
```

在该例中，若 AL 中包含 a ~ f 之间的小写字母时，用 .IF AL >= 'a' && AL <= 'f' 语句使其下一条指令（SUB AL, 57H）执行，把它们转换成十六进制数。若 AL 中的数不是 a ~ f 之间的字母，则用 .ELSEIF 语句判定它是否是 A ~ F 之间的大写字母，若是，则减去 37H，否则减去 30H，转换后的结果存入数据段内 TEMP 处。例 6-10b 的程序段表示，同样的转换可以用一个 C++ 功能函数实现。

6.2.1 WHILE 循环

类似于许多高级语言，宏汇编程序 MASM 6.X 版本也提供了 WHILE 循环语句。.WHILE 语句以相应的条件开始循环，以 .ENDW 语句结束循环。

例 6-11 给出如何用 .WHILE 循环语句从键盘上读入数据并存入 BUF 数组中，直到按下回车键（0DH）。因为是用指令 STOSB 将键盘数据值存入内存的，所以假定数组 BUF 存在附加段中。注意，程序中 .WHILE 循环部分表示为展开的格式，语句前面标有一个 * 号，说明该语句是由汇编程序加入的。在接收到回车键（ODH）后，字符串之后加上 $，以便于使用 DOS 中断 21H 的 09H 号功能调用在屏幕上显示。

例 6-11

```
                    ;本 DOS 程序从键盘读入 1 个字符串，当用回车键结束时，显示该字符串。
                    ;
                    .MODEL SMALL                          ;选择 SMALL 模型
0000                .DATA                                 ;数据段开始
```

```
0000 0D 0A          MES         DB  13,10              ;回车和换行符
0002 0100 [         BUF         DB 256 DUP (?)         ;字符串缓冲区
          00
            ]
0000                .CODE                              ;代码段开始
                    .STARTUP                           ;程序开始
0017 8C D8                      MOV AX,DX              ;ES = DS
0019 8C C0                      MOV ES,AX
001B FC                         CLD                    ;选择递增方式
001C BF 0002 R                  MOV DI,OFFSET BUF      ;取缓冲区地址

                                .WHILE AL ! = 0DH      ;循环到 AL 中是回车时结束
001F EB 05          *           jmp @ C0001
0021                * @ C0002:
0021 B4 01                      MOV AH,1               ;读入键码并显示
0023 CD 21                      INT 21H
0025 AA                         STOSB                  ;保存键码
                                .ENDW
0026                * @ C0001:
0026 3C 0D          *           cmp al,odh
0028 75 F7          *           jne @ C0002
002A C6 45 FF 24                MOV BYTE PTR[DI-1] '&'
002E BA 0000 R                  MOV DX,OFFSET MES
0031 B4 09                      MOV AH,9
0033 CD 21                      INT 21H                ;显示 MES
                    .EXIT
                    END
```

例 6-11 中，只要执行 . WHILE 语句时 AL 中装的数不是 0DH，循环就将继续。我们可以在 . WHILE 循环语句前加入指令 MOV AL，0DH 终止循环。虽然在例子中没有表示，. BREAK 语句可以和 . WHILE 循环语句一起使用。. BREAK 语句经常跟着 . IF 语句来选择断点条件，如 . BREAK . IF AL = = 0DH。CONTINUE 语句可用于当一个确定条件满足时允许 DO- . WHILE 循环继续执行，也可和 . BREAK 一起使用。举例来说，语句 . CONTINUE . IF AL = = 15 测试到 AL 等于 15 时循环继续运行。在 C++ 程序里，可以说 . BREAK 和 . CONTINUE 语句的功能是一样的。

6.2.2 REPEAT-UNTIL 循环

REPEAT-UNTIL 是汇编程序可用的另一种结构，它重复执行一组指令直到某些条件满足。. REPEAT 语句定义循环开始，带结束条件的 . UNTIL 语句定义循环结束。注意，只有 MASM 6. X 版本才有 . REPEAT 和 . UNTIL 语句。

如果用 REPEAT-UNTIL 结构重新编写例 6-11 的程序，可得到更好的效果。例 6-12 中的程序从键盘读取键值，并把键盘数据存到附加段数组 BUF 中，直到回车键被按下为止。这段程序也是将键盘数据填入缓冲区中，直到按下回车键（0DH）。按下回车键后，将整个字符串之后加上 $ 符，然后使用 DOS 中断 21H 的 09H 号功能调用在屏幕上显示字符串。在例子中会看到，. UNTIL AL = = 0DH 语句生成的代码（前面带 * 标记的语句）是怎样测试是否是回车键的。

例 6-12

```
                    ;本 DOS 程序从键盘读入字符串,然后显示该字符串。
                    ;
                    .MODEL SMALL                       ;选择 SMALL 模型
0000                .DATA                              ;数据段开始
0000 0D 0A          MES         DB  13,10              ;回车和换行符
0002 0100 [         BUF         DB 256 DUP (?)         ;字符串缓冲区
```

```
          00
            ]
0000                .CODE                           ;代码段开始
                    .STARTUP                        ;程序开始
0017 8C D8                      MOV AX,DX           ;ES = DS
0019 8C C0                      MOV ES,AX
001B FC                         CLD                 ;选择递增方式
001C BF 0002 R                  MOV DI,OFFSET BUF   ;给出 BUF 地址

                                .REPEAT             ;重复直到按下回车

001F                * @C0001:
001F B4 01                      MOV AH,1            ;带回显读入键值
0021 CD 21                      INT 21H
0023 AA                         STOSB               ;将键值存入 BUF

                                .UNTIL AL == 0DH

0024 3C 0D          *           cmp al,0dh
0026 75 F7          *           jne @c0001
0028 C6 45 FF 24                MOV BYTE PTR[DI-1] '&'
002C BA 0000 R                  MOV DX,OFFSET MES
002E B4 09                      MOV AH,9
0031 CD 21                      INT 21H             ;显示 MES
                    .EXIT
                    END
```

另外，还有一个.UNTILCXZ 语句也用 LOOP 指令测试循环条件 CX，.UNTILCXZ 语句用 CX 寄存器做计数器，使循环重复指定的次数。例 6-13 表示使用.UNTILCXZ 语句的指令序列，将字节数组 ONE 中的数与字节数组 TWO 中的数相加，和存入数组 THREE 中，由于每个数组均包含 100 个字节，所以需要循环 100 次。本例假定数组 THREE 在附加段，而数组 ONE 和 TWO 均在数据段。

例 6-13

```
012C B9 0064                MOV  CX,100             ;设定计数器
012F BF 00C8 R              MOV  DI,OFFSET THREE    ;寻址数组
0132 BE 0000 R              MOV  SI,OFFSET ONE
0135 BB 0064 R              MOV  BX,OFFSET TWO

                            .REPEAT

0138              * @C0001:
0138 AC                     LODSB
0139 02 07                  ADD  AL,[BX]
013B AA                     STOSB
013C 43                     INC  BX

                            .UNTILCXZ

013D E2 F9          *       LOOP @C0001
```

6.3 过程

过程、子程序或函数是所有计算机系统结构的重要组成部分。过程通常是执行某个任务的指令组。过程是存储在存储器中可重复使用的一段软件，而且是经常要用的，这样就节省了存储空间，并且能较容易地开发软件。过程的缺点仅仅是连接过程和从它返回时要花费计算机少量的时间。CALL 指令连接到过程，而 RET（返回）指令从过程返回。

当程序执行期间调用过程时，在堆栈中存储返回地址。CALL 指令将其后指令的地址（**返回地址**）压入堆栈。RET 指令从堆栈弹出地址，因此能够返回到 CALL 之后的指令。

汇编程序对于过程的存放有特殊的规则。过程要以 PROC 伪指令开始并且以 ENDP 伪指令结束。两个伪指令与过程的名字一起出现。使用这种程序结构能很容易在程序列表中找到过程。PROC 伪指令后面是过程的类型：NEAR 或 FAR。例 6-14 给出了汇编程序怎样使用 NEAR（段内的）和 FAR（段间的）两种过程定义。在 MASM 6. X 中 NEAR 或 FAR 过程后面可以用 USES 语句。USES 语句使进入过程后将一些寄存器内容自动地压入堆栈，退出过程前自动地弹出堆栈。USES 语句的应用见例 6-14。

例 6-14

```
0000             SUMS  PROC NEAR
0000 03 C3             ADD  AX,BX
0002 03 C1             ADD  AX,CX
0004 03 C2             ADD  AX,DX
0006 C3                RET
0007             SUMS  ENDP

0007             SUMS1 PROC FAR
0007 03 C3             ADD  AX,BX
0009 03 C1             ADD  AX,CX
000B 03 C2             ADD  AX,DX
000D CB                RET
000E             SUMS1 ENDP

000E             SUMS2 PROC NEAR  USE BX CX DX
0011 03 C3             ADD  AX,BX
0013 03 C1             ADD  AX,CX
0015 03 C2             ADD  AX,DX
0017 C3                RET
001B             SUMS2 ENDP
```

比较前两个过程，只是返回指令的操作码不同。近返回指令用操作码 C3H，而远返回用操作码 CBH。近返回从堆栈弹出 16 位数字，把它放入指令计数器，实现从当前代码段的过程返回。而远返回是从堆栈返回 32 位数字，并且放入 IP 和 CS，实现从过程返回到任何存储单元。

由整个软件使用的过程（**global，全局**），要写成远过程。由指定任务使用的过程（**local，局部**）常常定义为近过程。

6.3.1 CALL 指令

CALL 指令把程序流程传递到被调用的过程。CALL 指令不同于转移指令，因为 CALL 在堆栈内保存返回地址。执行 RET 指令时，控制返回到紧跟在 CALL 指令之后的那条指令。

近 CALL 指令

近 CALL 指令是 3 字节长，第 1 个字节包含操作码。对于 8086～80286 微处理器，第 2 和第 3 字节包含 ±32KB 的位移量或距离，这与近转移指令的格式相同。80386 及更高型号的微处理器按保护模式操作时用 32 位的位移量，允许 ±2GB 的距离。执行近 CALL 指令时，首先将下一条指令的偏移地址压入堆栈。下一条指令的偏移地址在指令指针（IP 或 EIP）中。存储这个返回地址以后，再将第 2 和第 3 字节的位移量加到 IP 上，从而把控制权传送给被调用过程。不存在短 CALL 指令。另一种操作码形式是 CALLN，但是只要用 PROC 语句把 CALL 指令定义为近的，就可以避免用它。

*为什么在堆栈上保存 IP 和 EIP 呢？*因为指令指针总是指示程序的下一条指令，CALL 指令把 IP/EIP 的内容压入堆栈，这样当被调用过程结束后，程序控制权就能送回到 CALL 指令的下一条指令。图 6-6 给出了堆栈中存储的返回地址（IP）和对过程的调用。

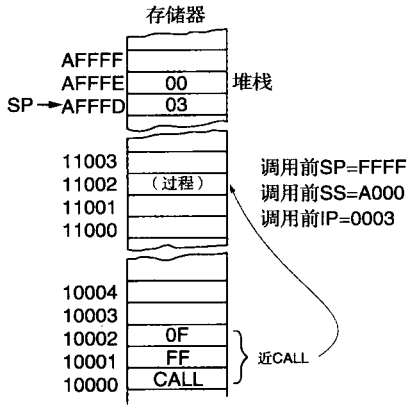


图 6-6 近 CALL 指令对堆栈和指令指针的影响

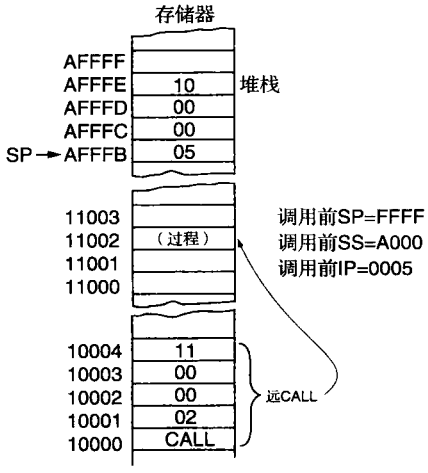


图 6-7 远 CALL 指令的作用

远 CALL 指令

远 CALL 指令类似于远转移指令，因为它可以调用存放在系统存储器任何位置的过程。远 CALL 指令是 5 字节的指令，操作码后面跟随 IP 和 CS 寄存器的值，字节 2 和 3 包含新的 IP 内容，而字节 4 和 5 包含新的 CS 内容。

在远 CALL 指令转移到由第 2 ～ 5 字节指示的指令地址之前，先将 IP 和 CS 的内容压入堆栈。这就允许远 CALL 指令调用存储器中任何位置的过程，然后从这个过程返回。

图 6-7 指出了远 CALL 指令是怎样调用远过程的，它先将 IP 和 CS 的内容压入堆栈，然后程序转移到被调用过程。远调用有另一种形式 CALLF，但应该避免使用它，而用 PROC 语句定义调用指令的类型。

在 64 位模式中，远调用指令可转到任何存储单元，并且存放在堆栈上的信息是 8 字节数。同样，远返回指令也从堆栈恢复一个 8 字节返回地址，并且放入 RIP。

用寄存器操作数的 CALL 指令

类似于转移指令，CALL 指令也可以有寄存器操作数，CALL BX 指令就是这样的例子。这条指令将 IP 的内容压入堆栈，然后转移到当前代码段位于 BX 寄存器中的偏移地址处。这类 CALL 指令可以使用一个 16 位偏移地址，该地址存放在除段寄存器以外的任何 16 位寄存器中。

例 6-15 说明用寄存器 CALL 指令完成调用的过程，该过程在偏移地址 DISP 开始（这种调用过程也可以用 CALL DISP 指令直接调用）。首先将 DISP 的偏移地址（OFFSET）放在 BX 寄存器中，然后 CALL BX 指令调用在 DISP 地址开始的过程。这个程序在显示器屏幕上显示 “OK”。

例 6-15

```
                    ;DOS 程序调用过程 DISP，在显示器屏幕上显示 "OK"
                    ;
                    .MODEL TINY                 ;选择 TINY 模型
0000                .CODE                       ;指示代码段开始
                    .STARTUP                    ;指示程序开始
0100 BB 0110 R          MOV  BX,OFFSET DISP     ;用 BX 寻址 DISP
0103 B2 4F              MOV  DL,'O'             ;显示'O'
0105 FF D3              CALL BX
0107 B2 4B              MOV  DL,'K'             ;显示'K'
0109 FF D3              CALL BX
                    .EXIT
                    ;
                    ;在显示器屏幕上显示 DL 内
```

```
                ;ASCII 码的程序。
                ;
0110            DISP PROC  NEAR
0110 B4 02           MOV  AH,2              ;选择功能号 02H
0112 CD 21           INT  21H               ;执行 DOS 功能调用
0114 C3              RET                    ;从过程返回
0115            DISP ENDP
                END
```

用间接存储器寻址的 CALL 指令

当程序需要从不同的子程序中选择一个时，用间接存储器寻址的 CALL 是有实际意义的。这种选择处理通常是先键入数字，然后寻找查找表中的 CALL 地址。这和本章前面用查找表查找转移地址来间接转移在本质上是一样的。

例 6-16 表示如何用间接 CALL 指令访问一个地址表，例中表明该表包含三个由数字 0、1 和 2 引用的独立的子程序。这个例子用比例变址寻址方式，使 EBX 乘以 2，准确地访问查找表内的正确入口。

CALL 指令也可以引用远指针，如果表中的数据用 DD 伪指令定义为双字，使用 CALL FAR PTR [4 * EBX]或 CALL TABLE [4 * EBX] 指令。这些指令就从数据段由 EBX 寻址的存储单元得到 32 位（4 字节）地址，并用它作为远过程的地址。

例 6-16

```
                ;根据 EBX 中的值调用过程：ZERO、ONE 或 TWO
                ;
TABLE   DW      ZERO            ;过程 ZERO 的地址
        DW      ONE             ;过程 ONE 的地址
        DW      TWO             ;过程 TWO 的地址

        CALL    TABLE [2* EBX]
```

6.3.2 RET 指令

返回指令（RET）从堆栈中取出 16 位数字（**近返回**）放入 IP，或者取出 32 位数字（**远返回**）放入 IP 和 CS 中。近返回指令和远返回指令都在过程的 PROC 伪指令中定义，因此能自动选择合适的返回指令。80386 ～ Pentium 4 微处理器按保护模式操作时，远返回从堆栈中取回 6 个字节：前 4 个字节包含 EIP 的新值，后两个字节包含 CS 的新值。80386 及更高型号的微处理器保护模式的近返回从堆栈取回 4 个字节，并且把它放入 EIP 中。

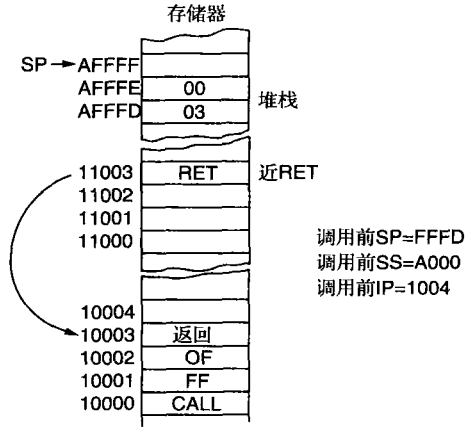


图 6-8 近返回指令对堆栈和指令指针的影响

改变了 IP/EIP 或 IP/EIP 及 CS 后，下条指令地址指向存储器新的位置。这个新位置就是直接跟在最近调用过程的 CALL 指令后面的地址。图 6-8 指出 8086 ～ Core2 在实模式下，CALL 指令怎样连接到过程及 RET 指令怎样从过程返回。

还有另一种形式的返回指令，它从堆栈弹出返回地址后，再给堆栈指针（SP）的内容加上一个数值。这种带立即操作数的返回指令非常适用于那些用 C/C++ 或 PASCAL 调用规则的系统（的确如此，C/C++ 和 PASCAL 调用规则要求调用者为许多功能调用删除堆栈数据）。这些规则在调用过程前先把参数压入堆栈，如果返回时要丢弃这些参数，返回指令要求包含一个数字，表示压入到堆栈中参数的字节数。

例 6-17 指出了这种类型的返回指令是如何抹除先前压入堆栈中的数据的。RET 4 指令从堆栈弹出

返回地址后，再将 SP 内容加 4。以前 PUSH AX 和 PUSH BX 一共把 4 个字节数据放入堆栈，这种返回方式有效地从堆栈中删除了 AX 和 BX。这种类型的返回指令在汇编语言程序中很少出现，但是在高级语言程序中，用它清除调用过程以后的堆栈数据。注意怎样用 BP 寄存器寻址堆栈上的参数，默认情况下 BP 寻址堆栈段。用堆栈传递参数对于用 C++ 或 PASCAL 调用规则为 C++ 或 PASCAL 写的过程是通用的。

例 6-17

```
0000 B8 001E          MOV   AX,30
0003 BB 0028          MOV   BX,40
0006 50               PUSH  AX            ;堆栈参数 1
0007 53               PUSH  BX            ;堆栈参数 2
0008 E8 0066          CALL  ADDM          ;加来自堆栈上的参数
0071             ADDM PROC  NEAR
0071 55               PUSH  BP            ;保存 BP
0072 8B EC            MOV   BP,SP         ;用 BP 寻址堆栈
0074 8B 46 04         MOV   AX,[BP+4]     ;得到堆栈数据 1
0077 03 46 06         ADD   AX,[BP+6]     ;加堆栈数据 2
007A 5D               POP   BP            ;恢复 BP
007B C2 0004          RET   4             ;返回并且抹除放到堆栈上的数据
007E             ADDM ENDP
```

与 CALLN 和 CALLF 指令类似，返回指令也有另一种形式：RETN 和 RETF，同样应避免使用这类形式，最好用 PROC 语句定义调用和返回指令的类型。

6.4 中断概述

中断或者是**硬件产生**（**hardware-generated**）的 CALL（由硬件信号从外部驱动），或者是**软件产生**（**software-generated**）的 CALL（由执行指令内部驱动或者是由于某些内部事件引发）。有时内部中断称为**异常**（**exception**）。任何一种类型都是通过调用**中断服务程序**（**interrupt service procedure**，**ISP**）或者中断处理程序使程序中断。

这一节解释软中断，它们是特殊类型的调用指令。这一节中叙述了三种类型的软中断指令（INT、INTO 和 INT3），提供了中断向量的映像，说明了专用的中断返回指令（IRET）的作用。

6.4.1 中断向量

当微处理器按实模式操作时，**中断向量**（**interrupt vector**）是 4 个字节的数字，它们存储在存储器的第一个 1024 单元（000000H ～ 0003FFH）。在保护模式中，用中断描述符表替代向量表，每个中断用 8 个字节的中断描述符说明。共有 256 个不同的中断向量，每个中断向量包含了一个中断服务程序的地址。表 6-4 列出了中断向量并且有简单的说明和实模式中每个向量在存储器中的位置。每个向量包含形成中断服务程序地址的 IP 和 CS 的值。前两个字节包含 IP 的值，而后两个字节包含 CS 的值。

Intel 为现在和将来的微处理器产品保留前 32 个中断向量。其余的中断向量（32 ～ 255）是用户可以使用的。在保留向量中，有些是为软件执行期间出现错误而保留的向量，例如除法错误中断；有些向量是为协处理器保留的；其余的由系统中的正常事件占用。正如这一节后面说明的那样，PC 中保留的向量用于系统功能。向量 1 ～ 6、7、9、16 和 17 用于实模式和保护模式，其余向量只用于保护模式。

表 6-4 中断

号 数	地 址	微处理器	功 能
0	0H ～ 3H	所有处理器	除法错
1	4H ～ 7H	所有处理器	单步
2	8H ～ BH	所有处理器	NMI 引脚（硬件中断）
3	CH ～ FH	所有处理器	断点
4	10H ～ 13H	所有处理器	溢出中断
5	14H ～ 17H	80186 ～ Core2	边界指令中断（越界）

（续）

号　数	地　址	微处理器	功　能
6	18H～1BH	80186～Core2	无效操作码
7	1CH～1FH	80186～Core2	协处理器仿真中断
8	20H～23H	80386～Core2	双重故障
9	24H～27H	80386	协处理器段越界
A	28H～2BH	80386～Core2	无效任务状态段
B	2CH～2FH	80386～Core2	段没有出现
C	30H～33H	80386～Core2	堆栈故障
D	34H～37H	80386～Core2	通用保护故障（GPF）
E	38H～3BH	80386～Core2	页故障
F	3CH～3FH	—	保留
10	40H～43H	80286～Core2	浮点错误
11	44H～47H	80486SX	对齐检测中断
12	48H～4BH	Pentium～Core2	机器检测异常
13～1F	4CH～7FH	—	保留
20～FF	80H～3FFH	—	用户中断

6.4.2 中断指令

微处理器有三种不同的中断指令：INT、INTO 和 INT3。在实模式中，这些指令中的每一条从向量表获取向量，然后调用过程，该过程存放在向量指向的那个位置；在保护模式中，这些指令中的每一条从中断描述符表中获取中断描述符。这些描述符指定了中断服务程序的地址。中断调用类似于远 CALL 指令，因为它也把返回地址（IP/EIP 和 CS）存放在堆栈中。

INT 指令

程序员可以使用256种不同的软中断指令（INT），每个软中断指令带有一个操作数，范围是0～255（00H～FFH）。例如，INT 100 使用中断向量100，它出现在存储器地址190H～193H处。中断向量地址由中断类型号乘4确定。例如，在实模式中 INT 10H 指令调用的中断服务程序的地址存储在40H（10H×4）开始的存储单元中。在保护模式中，中断描述符位于中断类型号乘8的单元中，因为每个描述符占8个字节。

每个 INT 指令为两个字节，第一个字节包含操作码，第二个字节包含向量类型号。只有 INT3 例外，规定一个字节用于断点的软件中断。

每次执行软中断指令时，操作顺序为：（1）将标志寄存器压入堆栈；（2）清除 T 和 I 标志位；（3）将 CS 压入堆栈；（4）从中断向量获取新的 CS 值；（5）将 IP/EIP 压入堆栈；（6）从中断向量中获取新的 IP/EIP 值；（7）转移到由 CS 及 IP/EIP 寻址的新位置。

INT 指令的执行类似于远 CALL，只是它不仅要把标志寄存器压入堆栈，而且还要把 CS 和 IP 压入堆栈。INT 指令先执行 PUSHF 操作，紧跟着完成远 CALL 指令。

注意，中断指令执行时，清除中断标志（I），因为它控制外部硬件中断输入 INTR 引脚（中断请求）。当 I=0 时，微处理器禁止 INTR 引脚；当 I=1 时，微处理器使能 INTR 引脚。

软中断通常用来调用系统过程，因为不需要知道系统函数地址。系统过程是全系统和应用软件公共的。这些中断通常控制打印机、视频显示器及磁盘驱动器。INT 指令可替代远 CALL，使程序不必再记着系统调用的地址，而是用另外一种方式调用系统功能。INT 指令是2字节长，而远 CALL 是5字节长。每次 INT 指令替代远 CALL 指令时，节省了3个字节的存储单元。如果作为系统调用的 INT 指令经常出现在程序中，可以节省相当数量的存储空间。

IRET/IRETD 指令

中断返回指令（IRET）只用于软件或硬件中断服务程序中。与简单的返回指令不同，IRET 指令

能够：1）弹出堆栈数据返还到IP；2）弹出堆栈数据返还到CS；3）弹出堆栈数据返还到标志寄存器。IRET指令与后面跟随POPF的远RET指令实现相同的功能。

每次执行IRET指令时，从堆栈恢复I和T的内容。保护这些标志位的状态很重要。如果在中断服务过程之前是允许中断的，则IRET指令可以自动再允许中断，因为它恢复了标志寄存器。

在80386～Core2微处理器中，IRETD指令用于从保护模式调用的中断服务程序中返回。它与IRET指令的区别是它从堆栈弹出32位的指令指针（EIP）。IRET指令用于实模式，而IRETD指令用于保护模式。

INT3指令

INT3指令是指定用于设置软件**断点**（**break point**）的特殊软中断指令。与其他软中断的区别是，INT3是单字节指令，而其他的是两字节指令。

通常在软件中插入一条INT3指令是为了中断或中止程序的执行，这种功能称为断点中断。虽然任何软中断都能用来设置断点，但因为INT3是单字节的，更适用于完成这个功能。断点中断有助于调试有错误的软件。

INTO指令

溢出中断（INTO）是测试溢出标志的条件软中断。如果溢出标志O=0，则INTO指令不操作；而如果O=1，则执行INTO指令，产生中断向量类型号4的中断。

INTO指令出现在有符号二进制数的加法或减法软件中。因为这些操作可能有溢出，要用JO指令或INTO指令检测溢出条件。

中断服务程序

假定要求程序累加DI、SI、BP和BX的内容，并且把和存入AX。因为这是该系统中的一个常用任务，把这个任务设计成软中断是值得的。尽管中断通常留作处理系统事件，然而这个例子说明它如何导致一个中断服务过程。例6-18给出了这个软中断。这个过程与正常的远过程之间的区别是：它的末尾用IRET指令而不是RET指令，而且在它执行期间把标志寄存器的内容保存到堆栈中。用USES保存所有被过程改变的寄存器也相当重要。

例6-18

```
0000            INTS  PROC  FAR USES AX
0000 03 C3            ADD   AX,BX
0002 03 C5            ADD   AX,BP
0004 03 C7            ADD   AX,DI
0006 03 C6            ADD   AX,SI
0008 CF               IRET
0009            INTS  ENDP
```

6.4.3 中断控制

虽然这一节不介绍硬中断，但要介绍两个控制INTR引脚的指令。**设置中断标志**（**set interrupt flag，STI**）指令把1放入I标志位，使能INTR引脚输入。**清除中断标志**（**clear interrupt flag，CLI**）指令把0放入I标志位，禁止INTR引脚输入。STI指令使能INTR；CLI指令禁止INTR。在软件中断服务程序中，把允许硬件中断作为第一步，这就是用STI指令实现的。在中断服务程序开始就允许中断的理由是，可能恰好有与PC机相关的各种I/O设备中断需要及时处理。如果中断被禁止得太久，会导致某些严重的系统错误。

6.4.4 PC机的中断

PC内的中断与表6-4中给出的中断有些区别，原因是最初的PC机是基于8086/8088的系统，只包括Intel指定的中断0～4。这样的设计是为了适应系统的发展，使较新的系统可以与早期的PC兼容。

采用由Windows完成的访问保护模式中断结构，要通过Microsoft公司提供的内核功能调用而不能

直接寻址。保护模式中断使用一个中断描述符表，它超出了本章的范围，将在后面章节中详细讨论保护模式中断。

图 6-9 表示在作者的计算机中可以看到的中断。中断配置可以在 Windows 执行和维护控制面板中看到，点击“系统”、“硬件”和“设备管理器”，然后选择“查看”栏中的“依类型排序资源”，最后选择“中断请求（IRQ）”。

6.4.5 64 位模式中断

64 位系统用 IRETQ 指令从中断服务过程返回。IRETQ 和 IRET/IRETD 指令的主要不同是 IRETQ 从堆栈恢复一个 8 字节返回地址。IRETQ 指令也从堆栈恢复 32 位 EFLAG 寄存器并放进 RFLAG 寄存器中。Intel 大概不打算使用 RFLAG 最左边 32 位。其他方面，64 位模式中断与 32 位模式中断相同。

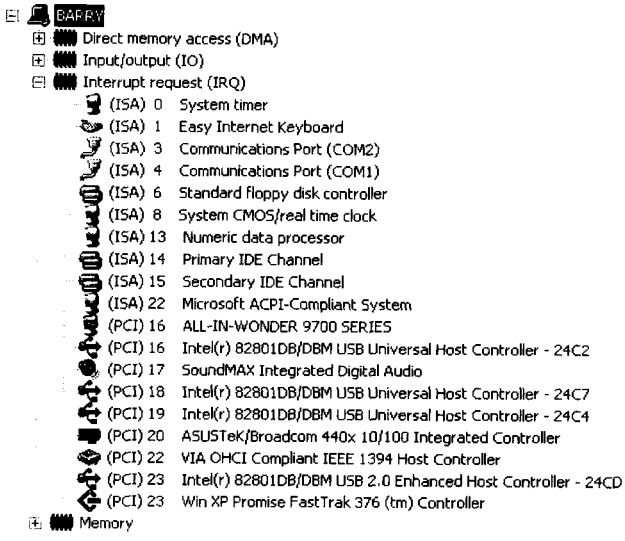


图 6-9 典型 PC 的中断

6.5 机器控制及其他指令

最后一类微处理器实模式指令是机器控制及其他指令。这些指令可控制进位，测试 BUSY/TEST 引脚，并可执行其他各种功能。由于这些指令中多数是控制硬件用的，因此只需要针对这一点进行简要说明。

6.5.1 控制进位标志位

进位标志（C）在多字或双字加减时用于传递进位或借位，也用于指示程序中的错误。有三种控制进位标志内容的指令：STC（将进位位置 1）、CLC（将进位位清 0）和 CMC（将进位位取反）。

除了多字加减以外很少使用进位标志，故该标志可以用于其他用途。进位标志最通常的作用是指示从过程返回时是否有错。假定过程从磁盘文件读数据，这个操作可能成功，也可能出现错误，如没有找到文件。从过程返回时，如果 C = 1，表示有错误出现；如果 C = 0，表示没有错误。多数的 DOS 过程和 BIOS 过程用进位标志指示错误条件。在 Visual C/C++ 与 C++ 一起用时，不能用这个标志。

6.5.2 WAIT 指令

WAIT 指令监控 80286 和 80386 上的硬件$\overline{\text{BUSY}}$引脚，在 8086/8088 上监控 TEST 引脚。80286 这个引脚的名字由$\overline{\text{TEST}}$变为$\overline{\text{BUSY}}$。如果执行 WAIT 指令时$\overline{\text{BUSY}}$ = 1，则没有什么情况发生，继续执行下一条指令。如果执行 WAIT 指令时引脚$\overline{\text{BUSY}}$ = 0，则微处理器要等待$\overline{\text{BUSY}}$引脚变回 1。这个引脚是逻辑 0 时指示忙状态。

微处理器的$\overline{BUSY}$/$\overline{TEST}$引脚通常与 8087 到 80387 数字协处理器的$\overline{BUSY}$引脚连接。这样连接使得微处理器等待，直到协处理器完成任务。由于协处理器位于 80486 ～ Core2 的内部，所以这些微处理器中没有$\overline{BUSY}$引脚。

6.5.3 HLT 指令

暂停指令（HLT）停止软件的执行。有三种方式退出暂停：通过中断，通过硬件复位或由于 DMA 操作。程序中出现这条指令通常是为了等待中断。它常常使外部硬件中断与软件系统同步。注意，DOS 和 Windows 都大量使用中断，所以当在这些操作系统上工作时 HLT 并不停止计算机。

6.5.4 NOP 指令

当微处理器遇到无操作指令（NOP）时，它用一小段时间执行该指令。早些年在软件开发工具出现之前，NOP 实现绝对的无操作，通常是用来给软件留空，作为将来填补机器语言指令的空间。如果你正在开发机器语言程序（当然这是极少见的），建议你最好在每隔 50 个字节左右放上 10 个 NOP，以防万一将来要在某些点加一些指令。NOP 指令也可以用于使程序延时一个短时间周期。由于现代微处理器中采用高速缓冲存储器和流水线，用 NOP 定时是不太准确的。

6.5.5 LOCK 前缀

LOCK 前缀附加在指令前，使$\overline{LOCK}$引脚变成逻辑 0。$\overline{LOCK}$引脚通常用来禁止外部总线上的主控制器或其他系统部件。LOCK 前缀使得在锁定指令期间激活$\overline{LOCK}$引脚。如果封锁多个指令的序列，在锁定的指令序列期间$\overline{LOCK}$引脚一直保持逻辑 0。如 LOCK：MOV AL，［SI］就是被锁定指令的例子。

6.5.6 ESC 指令

转义（ESC）指令从微处理器向浮点协处理器传递指令。如果需要，每次执行 ESC 指令时，微处理器会提供一个存储器地址，否则执行一次空操作 NOP。协处理器从 ESC 指令中的 6 位得到其操作码，并且开始执行协处理器指令。

ESC 操作码从来不在程序中作为 ESC 出现。它的位置是协处理器指令（FLD、FST 和 FMUL 等），汇编程序把它们看做是协处理器的 ESC。第 13 章详述了 8087 ～ Core2 数字协处理器。

6.5.7 BOUND 指令

第一次在 80186 微处理器中使用的 BOUND 指令是可以引起中断的比较指令（中断类型号 5）。这条指令将任何 16 位或 32 位寄存器的内容与两个字或双字的存储器的内容：上界和下界，对照比较。如果与存储器比较的寄存器值不在上界和下界以内，产生类型 5 中断。如果在界限以内，则顺序执行下条指令。

例如，如果执行 BOUND SI，DATA 指令，则把字单元 DATA 的内容作为下界，字单元 DATA +2 的内容作为上界。如果 SI 内的数据小于存储器单元 DATA 或大于存储器单元 DATA +2 的内容，出现 5 号中断。当出现这种中断时，返回地址指向 BOUND 指令，而不是 BOUND 后面的指令。这与正常中断返回地址指向程序中的下条指令不一样。

6.5.8 ENTER 和 LEAVE 指令

第一次在 80186 微处理器中出现的 ENTER 和 LEAVE 指令与堆栈帧一起使用。栈帧是通过堆栈向过程传递参数的机制。栈帧也为过程保存局部存储器变量。栈帧为多用户环境中的过程提供动态存储器区域。

ENTER 指令通过将 BP 压入堆栈，然后再将栈帧的最高地址装入 BP 寄存器中，从而建立一个栈帧。

这就允许通过 BP 寄存器访问栈帧变量。ENTER 指令有两个操作数：第一个操作数规定在栈帧中为变量保留的字节数，第二个操作数规定该过程的级别。

假定执行 ENTER 8，0 指令。这条指令为栈帧保留 8 个字节存储器，规定其级别为 0 级。图 6-10

给出了用这条指令建立的栈帧。这条指令将 BP 内容存入栈顶，然后栈指针减 8，留下 8 个字节的存储空间以便存储暂时数据。这 8 个字节的暂时存储区域的最高地址在 BP 寄存器中。LEAVE 指令把原来的值分别再装入 SP 和 BP 寄存器中，而使这个处理过程倒过来。在 Windows3.1 中 ENTER 和 LEAVE 指令用于调用 C++ 功能函数，但以后在现代版 Windows 中使用 CALL 调用 C++ 函数。

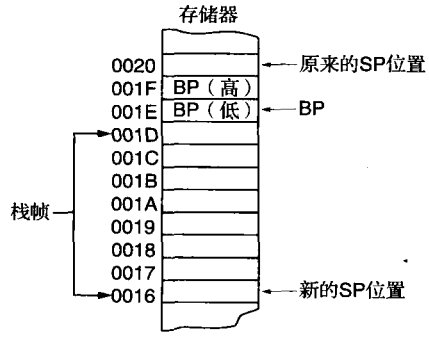


图 6-10 用 ENTER 8，0 建立的栈帧。注意，开始先将 BP 存储在栈的顶部。后面是称为栈帧的 8 个字节的存储区域

6.6 小结

1）有三种无条件转移指令：短转移、近转移和远转移。短转移允许转移到 +127 和 -128 字节以内。近转移（使用 ±32KB 的位移量）允许转移到当前代码段内的任何位置（段内转移）。远转移允许转移到系统存储器内的任何位置（段间转移）。80386 ~ Core2 中的近转移是在 ±2GB 以内，因为这些微处理器可以使用 32 位有符号的位移量。

2）如标号出现在 JMP 指令或条件转移指令中，该标号位于标号段定义时，它的后面必须跟一个冒号（LABEL:）。如，JMP DOGGY 指令转移到存储单元 DOGGY：处。

3）短转移或近转移后面的位移量是从该指令的下条指令地址到转移目标地址的距离。

4）间接转移可用两种形式：（1）转移到寄存器中存放的地址；（2）转移到字存储单元（近间接转移）中的地址或双字存储单元（远间接转移）中的地址。

5）条件转移全部是短转移，它测试一个或多个标志位：C、Z、O、P 或 S。如果条件是真，出现转移；如果条件是假，顺序执行下一条指令。注意 80386 及更高型号的微处理器允许 16 位有符号的位移量用于条件转移指令。64 位模式的微处理器允许 32 位的位移量在 ±2GB 范围内转移。

6）专用的条件转移指令（LOOP）使 CX 内容减 1，当 CX 内容不是零时转移到指令中的标号处。其他循环指令形式包括：LOOPE、LOOPNE、LOOPZ 和 LOOPNZ。LOOPE 指令，当 CX 不是零且存在相等条件，则转移。在 80386 ~ Core2 中，LOOPD、LOOPED 和 LOOPNED 指令用 ECX 寄存器作为计数器。64 位模式的微处理器使用 RCX 寄存器作为计数器。

7）在 80386 ~ Core2 中，有条件设置指令，或者设置字节为 01H，或者清除它为 00H。如果被测试的条件为真，字节操作数被设置为 01H；如果被测试的条件为假，字节操作数被清除为 00H。

8）在汇编语言程序要进行判定时，.IF 和 .ENDIF 语句非常有用，这些语句使汇编程序生成条件转移指令，以改变程序流程。

9）.WHILE 和 .ENDW 语句使汇编语言程序可用 WHILE 结构，.REPEAT 和 .UNTIL 语句使汇编语言程序可用 REPEAT-UNTIL 结构。

10）过程是实现一个任务的指令组，并可在程序中的任何地方被引用。CALL 指令调用过程，而 RET 指令从过程返回。在汇编语言中，PROC 伪指令定义过程的名字和类型，而 ENDP 伪指令声明过程结束。

11）CALL 指令是 PUSH 指令和 JMP 指令的组合。当执行 CALL 指令时，它把返回地址压入堆栈，然后转移到被调过程。近 CALL 指令将 IP 的内容放入堆栈，而远 CALL 指令将 IP 和 CS 的内容都放入堆栈。

12）RET 指令通过从堆栈弹出返回地址并将它放入 IP（近返回），或放入 IP 和 CS（远返回），实现从过程返回。

13）中断既可以是类似于 CALL 指令的软件指令，也可以是用来调用过程的硬件信号。这种操作将中断当前的程序并调用一个过程。过程结束后，用专用的 IRET 指令控制返回到被中断的软件。

14）实模式中断向量为 4 字节长，包含了中断服务程序的地址（IP 和 CS）。微处理器有 256 个中断向量，存放在存储器的第一个 1KB 内。Intel 定义了前面的 32 个，其余的 244 个是用户中断。在保护模式中，中断向量为 8 个字节长，而

且中断向量表可以在存储器的任何区域重定位。

15）微处理器响应中断时，把标志、IP 和 CS 压入堆栈。除了标志压入堆栈外，还清除 T 和 I 标志位，禁止跟踪功能和响应 INTR 引脚输入。最后是从中断向量表获得中断向量，然后转到相应的中断服务程序。

16）软中断指令 INT 经常替代系统调用，每次用软中断代替 CALL 指令，可节省 3 个字节存储空间。

17）从中断服务程序返回必须使用专用的返回指令 IRET。IRET 指令不仅从堆栈弹出 IP 和 CS，还从堆栈弹出标志。

18）溢出中断 INTO 是条件中断，当溢出标志 O = 1 时调用中断服务过程。

19）中断允许标志 I 控制微处理器的 INTR 引脚。如果执行 STI 指令，置 I 位为 1，允许 INTR 引脚输入。如果执行 CLI 指令，清除 I，则禁止 INTR 引脚输入。

20）通过 CLC、STC 和 CMC 指令对进位标志位清除、置 1 或取反。

21）WAIT 指令测试微处理器上$\overline{\text{BUSY}}$或$\overline{\text{TEST}}$引脚的状态。如果$\overline{\text{BUSY}}$或$\overline{\text{TEST}}$ = 1，WAIT 不等待；但是如果$\overline{\text{BUSY}}$或$\overline{\text{TEST}}$ = 0，WAIT 继续测试$\overline{\text{BUSY}}$或$\overline{\text{TEST}}$引脚，直到它变成逻辑 1。注意，8086/8088 有$\overline{\text{TEST}}$引脚，而 80286 和 80386 有$\overline{\text{BUSY}}$引脚，80486 ~ Core2 没有$\overline{\text{BUSY}}$或$\overline{\text{TEST}}$引脚。

22）LOCK 前缀在被封锁指令执行期间，使得$\overline{\text{LOCK}}$引脚变成逻辑 0。ESC 指令为数字协处理器传递指令。

23）BOUND 指令将任一 16 位寄存器的内容，与两个字存储单元的内容——上界和下界做比较。如果寄存器的值不在上界和下界范围内，则产生类型 5 中断。

24）ENTER 和 LEAVE 指令与栈帧一起使用，栈帧是通过堆栈存储器将参数传递给过程的机制。栈帧可存放过程的局部存储器变量。ENTER 指令建立栈帧，而 LEAVE 指令从堆栈清除栈帧。BP 寄存器访问栈帧中的数据。

6.7 习题

1. 什么是短转移?
2. 要转移到当前代码段内的任何位置时，应使用哪种类型的 JMP 指令?
3. 哪种 JMP 指令允许程序转移到系统内存储器的任何地址继续执行?
4. 哪种 JMP 指令是 5 字节长?
5. 在 80386 ~ Core2 微处理器中，近转移的范围是多少?
6. 以下各种情况汇编成哪种类型的 JMP 指令：（短、近、远）
   （a）如果距离是 0210H 字节
   （b）如果距离是 0020H 字节
   （c）如果距离是 10000H 字节
7. 标号后跟着冒号意味着什么?
8. 近转移通过改变哪些寄存器的值来修改程序地址?
9. 远转移通过改变哪些寄存器的值来修改程序地址?
10. 解释 JMP AX 指令实现什么功能？辨别它是近转移还是远转移指令?
11. 比较 JMP DI 与 JMP［DI］指令的操作。
12. 比较 JMP［DI］与 JMP FAR PTR［DI］指令的操作。
13. 列出条件转移指令测试的 5 个标志位。
14. 说明 JA 指令怎样操作。
15. JO 指令什么时候转移?
16. 哪些条件转移跟在有符号数比较以后?
17. 哪些条件转移跟在无符号数比较以后?
18. 哪些条件转移指令同时测试 Z 和 C 标志位?
19. JCXZ 转移指令什么时候执行转移?
20. 如果标志位指示零条件，哪个 SET 指令用来设置 AL?
21. Pentium 4 的 LOOPD 指令使_____寄存器减 1，并且为了决定是否发生转移测试它是否为 0。
22. 在 8086 微处理器中 LOOP 指令使_____寄存器减 1，并且为了决定是否发生转移测试它是否为 0。
23. 在 Core2 的 64 位模式中 LOOP 指令减少寄存器_____并且测试其是否为 0 用于决定是否转移。
24. 设计指令序列，把 00H 存储到附加段起始地址为 DATAZ 的 150H 个字节的存储区中。必须用 LOOP 指令帮助实现这个任务。
25. 解释 LOOPE 指令如何操作?
26. 指出下列语句生成的汇编语言指令序列：

```
. IF AL = =3
   ADD AL,2
. ENDIF
```

27. 设计指令序列，在 100H 字节的存储块内检索。这个程序必须统计所有高于 42H 的无符号数的数目和低于 42H 的无符号数的数目，高于 42H 的计数值放在数据段存储单元 UP 中，而低于 42H 数字的计数值放在数据段单元 DOWN 中。
28. 编写指令序列，用 REPEAT-UNTIL 结构，将字节存储区 BLOCKA 的内容复制到字节存储区 BLOCKB，直到一个 00H 被传送为止。
29. 如果 . WHILE 1 指令放在程序中会产生什么结果?
30. 用 WHILE 结构设计指令序列，当和数不是 12H 时将 BLOCKA 中的字节内容加到 BLOCKB 中。
31. . BREAK 伪指令的作用是什么?
32. 什么是过程?
33. 解释近调用和远调用指令是怎么执行的。
34. 过程中最后一条可执行的指令必须是_____。
35. 怎样完成近返回指令的功能?
36. 怎样定义过程为近过程或远过程?
37. 哪一个伪指令定义过程的开始?
38. 写出一个近过程，求 CX 寄存器内容的三次方，这个

过程除了 CX 寄存器以外不影响任何其他的寄存器。

39. 解释 RET 6 指令实现的操作。
40. 写一过程，DI 内容乘以 SI 内容后，结果被 100H 除，从过程返回时使结果留在 AX 中。这个过程不影响除了 AX 以外的任何其他寄存器。
41. 写出求 EAX、EBX、ECX 和 EDX 之和的过程。如果出现进位，将逻辑 1 放入 EDI，如果不出现进位，将 0 放入 EDI。程序执行以后，和要放在 EAX 中。
42. 什么是中断?
43. 哪些软件指令调用中断服务程序?
44. 微处理器中可以使用多少种不同的中断类型号?
45. 举例说明中断向量的内容，并且解释每个部分的作用。
46. 0 号向量中断的用途是什么?
47. IRET 指令与 RET 指令有什么区别?
48. IRETD 指令的用途是什么?
49. IRETQ 指令的用途是什么?
50. 什么条件下 INTO 指令才中断程序执行?
51. INT 40H 指令的中断向量存储在哪些存储单元?
52. 什么指令控制 INTR 引脚的功能?
53. 哪条指令测试$\overline{\text{BUSY}}$引脚?
54. 什么时候 BOUND 指令使程序中断?
55. ENTER 16，0 指令建立的栈帧包含_____字节。
56. 执行 ENTER 指令时，哪个寄存器内容压入堆栈?
57. 哪种指令将操作码传递到数字协处理器?

第7章　在C/C++中使用汇编语言

引言

如今，用汇编语言来开发整个系统已经很少见了，通常我们用汇编语言和C/C++一起进行开发。汇编语言部分通常用来完成C/C++语言难以实现或实现效率不能满足要求的任务，这些任务常常包含了外设接口的控制软件和中断驱动程序。汇编语言在C/C++程序中的另外一个应用就是为了使用MMX和SEC指令，这些指令是Pentium类处理器指令的一部分，在C/C++中还不支持。尽管C++中有了这些命令的宏，但是用起来要比使用汇编语言复杂。本章详述了汇编语言和C/C++语言混合编程的思想。后面章节中的一些应用程序也进一步说明了如何使用汇编语言和C/C++语言完成微处理器任务。

本章使用的是Microsoft Visual C/C++，其他版本的C/C++只要是标准ANSI格式的也可以使用。如果愿意，可以用C/C++环境输入并运行文中所有的应用程序。16位的应用程序用Microsoft Visual C/C++1.52或更新版本编写（可用免费的**CL.EXE**作为Microsoft公司**Windows驱动程序开发包**（**Driver Development Kit**，**DDK**）中的应用程序）；32位的应用程序用Microsoft Visual C/C++6或更新版本编写，最好是Microsoft Visual C/C++.NET 2003或Visual C++ Express版本。文中所写的例子是假设你已经有最新版的**Visual C/C++ Express**，这是一个Visual C++的免费下载版本。请访问http://msdn.com获得Visual C++ Express程序。

目的

读者学习完本章后将能够：

1）在C/C++的_asm块中使用汇编语言。

2）了解运用混合语言开发软件的规则。

3）在汇编语言中使用公共的C/C++数据和结构。

4）在汇编语言代码中使用16位（DOS）和32位（Microsoft Windows）两种界面。

5）在C/C++程序中使用汇编语言的目标代码。

7.1　在16位DOS应用程序中使用汇编语言与C/C++语言

本节讲解怎样在C/C++程序中插入汇编语言命令。这一点很重要，因为程序的性能通常依赖于为加速其运行而插入的汇编语言序列。正如在本章引言中提到的，汇编语言在嵌入式系统中用来进行I/O操作。本章假定你使用的是某个版本的Microsoft C/C++程序，如果其他的C/C++支持内嵌汇编命令，则也可以使用。惟一的变化可能是要设置C/C++包，使其与汇编语言一起使用。本节我们假定正在构造16位的DOS应用程序。在尝试本节的程序之前首先确定你的软件可以构造16位的应用程序。如果构造32位应用程序并试图用DOS INT 21H功能，因为不允许直接的DOS调用，程序可能会崩溃。事实上，在32位的应用中DOS调用已经无效了。

为了构造16位DOS应用程序，需要老的16位编译器，通常可以在Windows DDK目录C:\WINDDK\2600.1106\bin\win_me\bin16下找到（Windows驱动程序开发包只需要很少的费用就可以从Microsoft公司得到）。编译器是CL.EXE，16位程序的链接器是**LINK.EXE**。两个文件都在所列的目录或文件夹下。由于正在使用的计算机路径可能是指向32位链接器程序的，因此明智的做法就是在该目录下工作，以便使用正确的链接器链接编译器生成的目标文件。编译和链接必须在命令行下完成，因为编译器和链接器没有提供可视化界面或编辑器。程序可以用NotePad或者DOS的Edit来编辑。

7.1.1 基本规则和简单程序

在汇编语言代码放到C/C++程序中之前，必须了解一些规则。例7-1展示了如何在短的C/C++程序中将汇编代码放到汇编语言块中。注意该例子中的所有汇编代码都放到了**_asm**块中。该例中使用了标号，如程序中所示的**big:**。在内嵌汇编代码中使用小写字母，这一点非常重要。如果使用大写，就会发现一些汇编语言命令和寄存器是C/C++语言的保留字或已定义字。

例7-1中除了main过程，没有使用其他C/C++命令。用NotePad或Edit输入这个程序。该程序从控制台键盘读取一个字符，接着用汇编语言对它进行过滤，使只有0～9的数字回显到显示器。虽然这个编程例子没有完成更多功能，但它说明了如何在C/C++环境中设置和使用简单的编程结构以及如何使用内嵌汇编程序。

例7-1

```
//接收并显示0～9中的一个字符
//忽略其他所有的字符

void main (void)
{
      _asm
      {
            mov ah,8       ;读键盘不回显
            int 21h
            cmp al,'0'     ;过滤键码
            jb  big
            cmp al,'9'
            ja  big
            mov dl,al      ;回显0～9
            mov ah,2
            int 21h
      big:
      }
}
```

在例7-1中没有保存寄存器AX，但程序中用到了它。注意，Microsoft C/C++没有使用AX、BX、CX、DX和ES寄存器，这一点很重要（AX在过程中返回的功能将在本章后面进行介绍）。这些寄存器通常用作**中间结果**（**scratch pad**）寄存器，汇编语言可以使用。如果想要使用其他寄存器，一定要确保在使用它们之前用PUSH指令将其保存，然后用POP指令恢复。如果对程序中用到的寄存器未进行保存，程序就有可能工作不正常，并且有可能使计算机崩溃。如果使用80386或更高处理器作为程序的基础，就不需要保存EAX、EBX、ECX、EDX和ES。如果使用其他寄存器则必须保存，否则程序可能崩溃。

要编译程序，首先要启动命令行提示符程序，该程序位于Windows开始菜单下的附件中。改变路径到C:\WINDDK\2600.1106\bin\win_me\bin16目录下，当然前提是Windows DDK安装在该目录下。还需要到C:\WINDDK\2600.1106\lib\win_me目录下，复制slibce.lib到C:\WINDDK\2600.1106\bin\win_me\bin16目录下。确定已经在相同目录下用.c扩展名保存了文件。如果使用NotePad，确定在保存时文件类型选择为“所有文件”。要编译程序，输入CL/G3 filename.c <CR>，为程序生成.exe文件（/G3是指80386）（编译器/G开关的说明参见表7-1）。如果出现任何错误，按回车键忽略，这些错误产生的警告在程序执行时不会产生问题。

表7-1 编译器（16位）G选项

编译开关	功能
/G1	选择8088/8086
/G2	选择80188/80186/80286
/G3	选择80386
/G4	选择80486
/G5	选择Pentium
/G6	选择Pentium Pro～Pentium 4

注：32位C++编译器不认/G1或/G2。

当程序执行时，你将看到只有数字才能回显到 DOS 屏幕上。

例 7-2 说明了如何用短的汇编语言程序访问 C 的变量。在这个例子中，8 位字节数据用字符变量类型（在 C 中为一个字节）表示以节省空间。程序本身实现 X + Y = Z 的运算，这里 X 和 Y 是两个 1 位数的数字，Z 是结果。正如读者可能想到的，可以在 C 中用内嵌汇编来学习汇编语言并编写本教材中给出的许多程序。和一般的汇编程序一样，可以用分号在_asm 块中的汇编程序清单上添加注释。

例 7-2

```
void main (void)
{
      char a, b;
      _asm
      {
           mov  ah,1       ;读第一个数
           int  21h
           mov  a,al
           mov  ah,1       ;读加号
           int  21h
           cmp  al,'+'
           jne  endl       ;如果不是加
           mov  ah,1
           int  21h        ;读第二个数
           mov  b,al
           mov  ah,2       ;显示 =
           mov  dl,'='
           int  21h
           mov  ah,0
           mov  al,a       ;产生和
           add  al,b
           aaa             ;调整和为 ASCII
           add  ax,3030h
           cmp  ah,'0'
           je   down
           push ax         ;显示十位
           mov  dl,ah
           mov  ah,2
           int  21h
           pop  ax
      down:
           mov  dl,al      ;显示个位
           mov  ah,2
           int  21h
      endl:
      }
}
```

7.1.2 _asm 块中不能使用的 MASM 功能

虽然 MASM 含有许多好的特性，例如条件命令（.IF、.WHILE 和 .REPEAT 等），但内嵌汇编程序不包括 MASM 中的条件命令，也不包括汇编程序中的宏功能。内嵌汇编程序中的数据分配由 C 来处理，而不是用 DB、DW、DD 等定义。其他的功能内嵌汇编程序都支持。内嵌汇编程序中的这些删节会引起一些小问题，我们将在本章的后面进行讨论。

7.1.3 使用字符串

例 7-3 给出了一个简单的程序，该程序使用由 C 定义的字符串，并显示该串，每个单词列在独立

的行中。注意C语句和汇编语言语句的混合。WHILE语句重复汇编语言命令，直到在字符串末尾发现空（NULL，00H）字符为止。如果没有发现空字符，汇编语言指令将显示串中的字符，空格字符除外。对于每个空格字符程序将显示一个回车/换行组合。这使得每个单词被显示到独立的一行。

例7-3

```
//每行显示一个单词的例子
void main (void)
{
     char string1[] = "This is my first test application using _asm. \n";
     int sc = -1;
     while (string1[sc++]! =0)
     {
          _asm
          {
              push si
              mov  si,sc              ;取指针
              mov  dl,string1 [si]    ;取字符
              cmp  dl,' '             ;是否为空格
              jne  next
              mov  ah,2               ;显示新行
              mov  dl,10
              int  21h
              mov  dl,13
        next: mov  ah,2               ;显示字符
              int  21h
              pop  si
          }
    }
}
```

假如想在程序中显示不止一个串，而且又想用汇编语言开发显示串的软件。例7-4所示的程序中创建了一个显示字符串的过程。这个过程每调用一次，程序就会显示一个字符串。注意，这个程序每行显示一个字符串，与例7-3不同。

例7-4

```
//显示C语言字符串的汇编语言过程举例

char string1[] =" This is my first test program using _asm.";
char string2[] =" This is the second line in this program.";
char string3[] =" This is the third.";

void main (void)
{
     Str (string1);
     Str (string2);
     Str (string3);
}

Str (char * string_adr)
{
     _asm
     {
          mov  bx,string_adr    ;取字符串地址
          mov  ah,2
     top:
```

```
        mov  dl,[bx]
        inc  bx
        cmp  dl,0              ;如为空
        je   bot
        int  21h               ;显示字符
        jmp  top
    bot:
        mov  dl,13             ;显示回车换行
        int  21h
        mov  dl,10
        int  21h
    }
}
```

7.1.4 使用数据结构

数据结构是许多程序中的重要部分。本节将说明 C 创建的数据结构与对该结构中的数据进行操作的汇编语言部分如何接口。例 7-5 给出了一小段程序，该程序用数据结构存储姓名、年龄和收入，接着用一些汇编语言过程显示了每一条记录。尽管有例 7-4 中所示的 string 过程可以显示字符串，但没有显示回车/换行组合，而只是显示了空格。Crlf 过程显示回车/换行组合，Numb 过程显示一个整数。

例 7-5

```
//显示 C 的数据结构内容的汇编语言过程举例

//一个简单的数据结构

typedef struct records
{
    char first_name[16];
    char last_name[16];
    int age;
    int salary;
} RECORD;

//初始化结构数组

RECORD record[4] =
{ {" Bill"," Boyd",56,23000},
  {" Page"," Turner",32,34000},
  {" Bull"," Dozer",39,22000},
  {" Hy"," Society",48,62000}
};

//程序

void main (void)
{
    int pnt = -1;
    while (pnt++ <3)
    {
        Str (record[pnt].last_name);
        Str (record[pnt].first_name);
        Numb (record[pnt].age);
        Numb (record[pnt].salary);
        Crlf ();
    }
}
```

```
Str (char * string_adr[])
{
     _asm
     {
          mov  bx,string_adr
          mov  ah,2
     top:
          mov  dl,[bx]
          inc  bx
          cmp  dl,0
          je   bot
          int  21h
          jmp  top
     bot:
          mov  dl,20h
          int  21h
     }
}
Crlf ()
{
     _asm
     {
          mov  ah,2
          mov  dl,13
          int  21h
          mov  dl,10
          int  21h
     }
}

Numb (int temp)
{
     _asm
     {
          mov  ax,temp
          mov  bx,10
          push bx
     L1:
          mov  dx,0
          div  bx
          push dx
          cmp  ax,0
          jne  L1
     L2:
          pop  dx
          cmp  dl,bl
          je   L3
          mov  ah,2
          add  dl,30h
          int  21h
          jmp  L2
     L3:
          mov  dl,20h
          int  21h
     }
}
```

7.1.5 混合语言编程的例子

为了说明怎样将这项技术应用到程序中，例7-6展示了程序是如何部分用C部分用汇编语言完成一些操作的。这里，程序中仅有的汇编部分是Dispn过程和ReadNum过程，Dispn过程显示了一个整数，Readnum过程读一个整数。例7-6中的程序没有尝试检测或纠正错误。此外，该程序只有结果为正且小于64K时才能正确运行。注意，这个例子用汇编语言执行I/O和数据处理，C代码部分完成其他所有操作，实现程序的外壳。

例7-6

```
/*
//完成加、减、乘、除运算的简单计算器程序。格式为X <操作> Y =
*/

int temp;

void main(void)
{
      int temp1, oper;
      while (1)
      {
            oper = Readnum();                  //取得第一个数
            temp1 = temp;
            if ( Readnum() == '=' )            //取得第二个数
            {
                  switch (oper)
                  {
                        case '+':
                              temp += temp1;
                              break;
                        case '-':
                              temp = temp1 - temp;
                              break;
                        case '/':
                              temp = temp1 / temp;
                              break;
                        case '*':
                              temp *= temp1;
                              break;
                  }
                  Dispn(temp);                 //显示结果
            }
            else
                  Break;
            }
      }
}

int Readnum()
{
      int a;
      temp = 0;
      _asm
      {
      Readnum1:
            mov   ah,1
            int   21h
            cmp   al,30h
            jb    Readnum2
            cmp   al,39h
            ja    Readnum2
            sub   al,30h
            shl   temp,1
            mov   bx,temp
            shl   temp,2
```

```
            add  temp,bx
            add  byte ptr temp,al
            adc  byte ptr temp+1,0
            jmp  Readnum1
      Readnum2:
            Mov  ah,0
            mov  a,ax
      }
      return a;
}

Dispn (int DispnTemp)
{
      _asm
      {
            mov  ax,DispnTemp
            mov  bx,10
            push bx
      Dispn1:
            mov  dx,0
            div  bx
            push dx
            cmp  ax,0
            jne  Dispn1
      Dispn2:
            pop  dx
            cmp  dl,bl
            je   Dispn3
            add  dl,30h
            mov  ah,2
            int  21h
            jmp  Dispn2
      Dispn3:
            mov  dl,13
            int  21h
            mov  dl,10
            int  21h
      }
}
```

7.2 在32位应用程序中使用汇编语言与Visual C/C++语言

16位应用程序与32位应用程序之间存在着较大差异。32位应用程序用Windows下的Microsoft Visual C/C++ 编写，而16位应用程序用Microsoft Visual C/C++ for DOS编写。主要区别就是现在Microsoft Visual C/C++应用得更普遍一些，但是它不容易使用类似于INT 21H的DOS功能调用。建议不需要可视化界面的嵌入式应用程序用16位C或C++编写，而与Windows或Windows CE（用于使用ROM或Flash[⊖]设备的嵌入式应用）一起使用的应用程序则用32位Windows下的Visual C/C++。

32位应用程序可以使用32位寄存器，对于Windows存储空间基本限定在2GB。免费版本的Visual C++ Express不支持用汇编语言编写的64位应用程序。惟一不同之处就是不能使用DOS功能调用，改为用控制台的getch()或getche()和putch()等C/C++语言的函数，这些函数可用于DOS控制台应用程序。嵌入式应用程序可以直接使用汇编语言指令访问嵌入式系统中的I/O设备。在可视化界面下，所有的I/O都由Windows操作系统框架来处理。

Win32下的控制台应用运行在本地（native）模式下，允许将汇编语言包含在程序中，除了用_asm关键字外，不需要其他什么。Windows窗体（Windows Forms）应用更具有挑战性，因为它们运行在托管模式（manage mode）下，而不是运行在处理器的本地模式下。托管应用运行在虚拟模式

⊖ Flash是Intel公司的注册商标。

(pseudo mode) 下，不产生本地代码。

7.2.1 使用控制台 I/O 访问键盘和显示器的例子

例 7-7 所示的是利用控制台 I/O 命令从控制台读写数据的一个简单例子。进入这个应用程序（假定 Visual Studio .NET 2003 或 Visual C++ Express 可用），在新建项目选项中（参见图 7-1）选择 WIN32 控制台应用。注意，我们用 conio.h 库取代了惯用的 stdio.h 库。该例程可以从二进制到十六进制之间的任意进制显示 1～1000 之间的数字。注意主程序不再像早期的 C/C++ 程序那样叫 main，在当前的 Visual C/C++ Express 中用作控制台应用时叫_tmain。argc 是从命令行传给_tmain 过程的参数个数，argv[ ]是包含命令行参数串的数组。

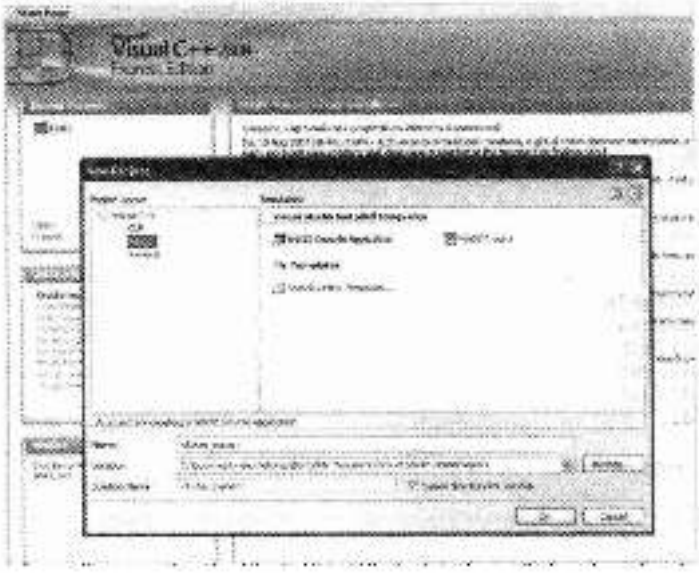


图 7-1 WIN32 控制台应用程序的新建项目窗口选择

例 7-7

```
//该程序以二进制到十六进制之间的任意进制显示任意数字。

#include "stdafx.h"
#include <conio.h>

char *buffer = "Enter a number between 0 and 1000: ";
char *buffer1 = "Base : ";
int a, b = 0;

void disp(int base, int data);

int _tmain(int argc, _TCHAR* argv[])
{
      int i;
       cputs(buffer);
      a = _getche();
      while ( a >= '0' && a <= '9' )
      {
            _asm sub a, 30h;
            b = b * 10 + a;
            a =  getche();
      }
      _putch(10);
       putch(13);
      for (i = 2; i < 17; i++ )
      {
```

```
        _cputs(buffer1);
        disps(10,i );
        _putch(' ');
        disps(i, b);
        _putch(10);
        _putch(13);
}
getche();                          //等待任意键
        return 0;
}
void disps(int base, int data)
{
        int temp;
        _asm
        {
                mov  eax, data
                mov  ebx, base
                push ebx
        disps1:
                mov  edx,0
                div  ebx
                push edx
                cmp  eax,0
                jne  disps1
        disps2:
                pop  edx
                cmp  ebx,edx
                je   disps4
                add  dl,30h
                cmp  dl,39h
                jbe  disps3
                add  dl,7
        disps3:
                mov  temp,edx
        }
        _putch(temp);
        _asm jmp  disps2;
disps4:;
}
```

这个例子显示汇编语言和C/C++语言命令很好地实现了混合。过程disps（base，data）做了这个程序的大部分工作，它可以以二进制到三十六进制之间的任意进制显示整数。由于在字母Z之后我们用完了表示基数的字母，所以出现了上界。如果需要转换成更大的基数，就需要为超过36的基数开发新方案。或许字母a到z可以用作37到52的基数。例7-7可以显示以二进制到十六进制方式输入的数。

7.2.2 直接访问I/O端口

如果编写的程序必须访问实际的端口号，我们可以使用控制台I/O命令，例如_inp（port）命令可输入字节数据，_outp（port，byte_data）命令可输出字节数据。编写PC机软件时，很少直接访问I/O端口，但为嵌入式系统编写软件时，我们经常直接访问I/O端口。_inp和_outp命令可以用汇编语言替换，在多数情况下汇编语言更有效。如果使用Windows NT，Windows 2000、Windows XP、Windows Vista，那么一定要知道在这样的Windows环境下I/O端口是不能直接访问的。这些系统下访问I/O端口的惟一办法就是开发内核驱动。本书不再给出开发这样的驱动的实践。如果使用Windows 98甚至Windows 95，那么就可以用inp和outp直接访问I/O端口。

7.2.3 开发Windows的Visual C++应用程序

本节将展示如何用Visual C++ Express为MFC（Microsoft Foundation Classes）库开发基于对话框的应用。MFC是一组类的集合，可以使我们轻松使用Windows的接口界面。在Visual C++ Express中MFC更名为公共语言运行库（common language runtime，CLR）。用于学习和开发的最容易的应用程序就是这里介绍的基于窗体应用程序。这种基础应用程序类型将用来编程和测试本书中所给出的在Visual

C++ Express 环境下编写的所有例程。

要创建一个基于窗体的 Visual C++ 应用程序，首先启动 Visual C++ Express，接着点击屏幕左上角的 Create Project。（如果还没有 Visual C++ Express，可以从微软公司的网站 http://msdn.com 上免费获取。）下载和安装最新版本的 Visual C++ Express，beta 版本也可以，图 7-2 给出了在 Visual C++ Express 项目下选择 CLR Windows 窗体应用程序（Windows Form Application）类型时所显示的内容。输入项目名称并为项目选择正确的路径，然后点击 OK（确认）按钮。

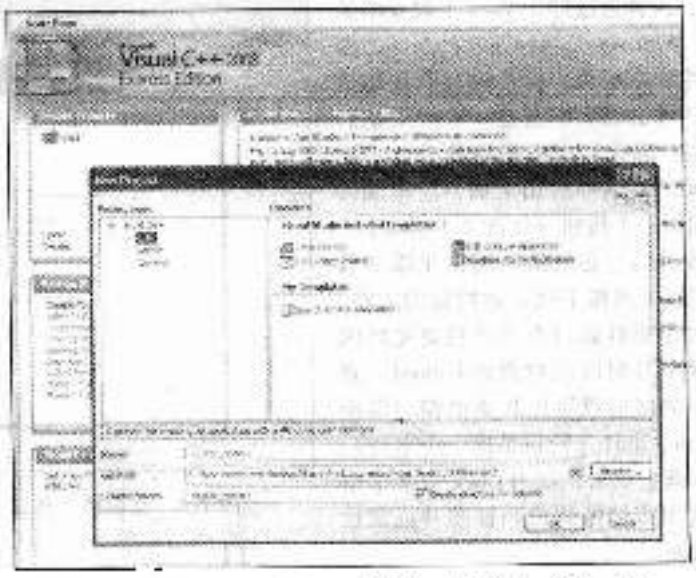


图 7-2 用 Visual C++ Express 新建一个 Windows C++ 程序

片刻后，设计窗口将显示如图 7-3 所示的画面。中间部分就是为这个应用创建的窗体（form）。要测试该应用程序，可以找到位于屏幕顶部 Windows 菜单条下方、窗体上方的绿色按钮并点击，则开始编译、链接和执行对话框应用。（在出现 "Would you like to build the application?" 时，回答是。）点击标题条中的 X，可以关闭应用程序。你已经创建并测试了第一个 Visual C++ Express 应用程序。

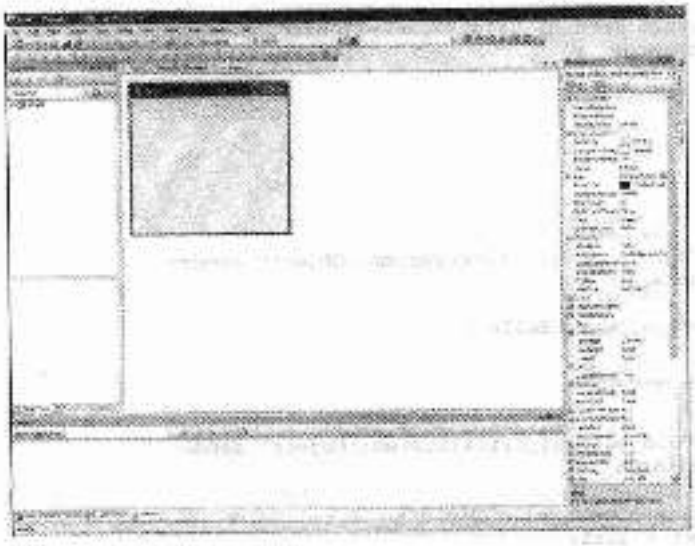


图 7-3 设计窗体的屏幕截图

图 7-1 展示了对程序创建和开发非常重要的一些内容。屏幕右边的区域是属性窗口，它包含了窗体的属性。屏幕左边区域是解决方案管理器（solution explorer）。解决方案管理器下面的标签页允许显示其他视图，如在这个区域可以查看类的视图等。属性窗口下面的标签页使类、属性、动态帮助或输出能够显示到这个窗口。你的屏幕可以如图 7-3 所示风格一样，也可以不一样，因为它可以修改成你喜欢的风格。

为了创建简单的应用程序，点击屏幕顶部的 Tools 并选择 toolbox，或者通过打开 View 下拉菜单从列表中选择 toolbox。Windows 是事件驱动的系统，窗体上需要对象或控件发起事件。控件可以是按钮或大部分从 toolbox 中选择的东西。点击 toolbox 上端的按钮控件，选择按钮，现在将鼠标指针移动（不要拖按钮）到屏幕中间的对话框应用上面，左击鼠标并调整大小，在中间画一个按钮（如图 7-4 所示）。

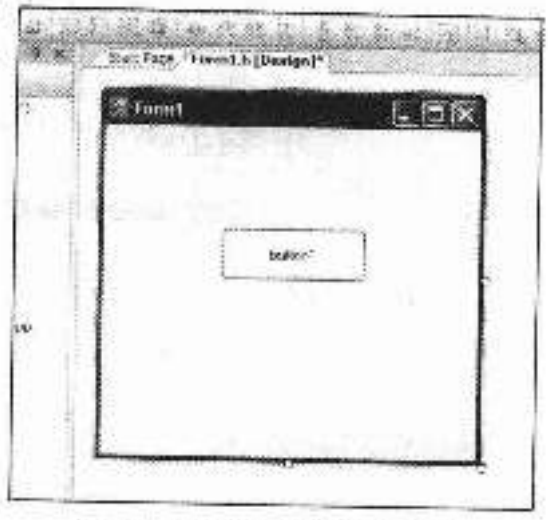


图 7-4 在窗体上放置一个控件按钮

按钮在屏幕上放好后，必须给应用程序添加事件处理函数，以便能够处理按下或点击按钮的动作。事件处理函数都通过在属性窗口点击黄色高亮的闪电图标来选择。确保事件对应的对象是 button1。点击高亮闪电图标左边的图标就可以从事件窗口切换到属性窗口，找到 Click 事件（应该是第一个事件），然后在其右边的文本框上双击，就为 Click 安装了事件处理函数。这时视图将切换到代码视图并且定位到点击按钮处理函数处。

视图中当前的软件是 button1_Click 函数，在用户点击按钮时将调用。这个程序如例 7-8 所示。要测试这个按钮，可将例 7-8 中的软件修改为例 7-9a 所示的内容。点击绿色箭头编译、链接、执行该对话框应用程序。程序运行后点击 button1，如果按钮足够宽，button1 的标签将改变为 "Wow, Hello"。这是第一个可工作的应用程序，但是它没有使用任何汇编代码。例 7-9a 使用了 button1 对象的 Text 成员属性改变 button1 上的文本显示。例 7-9b 出现了一个使用字符串对象（String^）的变量来显示 "Wow, Hello World"。

例 7-8

```
private: System::Void button1_Click(System::Object^ sender,
        System::EventArgs^ e)
{

}
```

例 7-9

```
//a 版本

private: System::Void button1_Click(System::Object^ sender,
        System::EventArgs^ e)
{
        button1->Text = "Wow, Hello";
}

//b 版本

private: System::Void button1_Click(System::Object^ sender,
        System::EventArgs^ e)
{
        String^ str1 = "Wow, Hello World";
        button1->Text = str1;
}
```

现在一个简单的应用程序已经写好了，我们可以将其进一步修改为如图7-5所示的更复杂的应用程序。按钮上的标题可以改为单词"Convert"。选择标题为form1.h [design]*的编辑窗口返回到设计屏幕。在设计窗口下，点击按钮并从属性窗口的button1的属性中找到Text属性，修改该属性就可以改变按钮的标题。将Text属性改为"Convert"。图7-5中Convert按钮左边有三个标签控件和三个文本框控件。这些控件位于toolbox中。在屏幕上如图7-5所示的大约位置上画这些控件。标签在每个标签控件的属性中修改。将每个标签的文本修改为所示内容。

在这个例子中我们的目标是将由于进制输入框输入的任意十进制数显示成任意基数的一个数，该数基由基数输入框输入。结果在点击Convert按钮后出现在结果框中。点击设计窗口上面的form1.h标签页可切换到程序视图。

要从编辑控件获得值，可用Text属性获得该数字的字符串形式。问题是这里需要的是整数而不是字符串，字符串必须转换成整数。C++中的Convert类可以完成多种数据类型的转换。在这种情况下，Convert类的成员函数ToInt32可将串转换为整数。这个例子的难点在于从基数10到任意基数的变换。例7-10给出了Convert类如何将来自文本框的字符串转换为整数。如果textbox1输入的数字是一个整数，就能正确地完成功能。如果输入字母或其他任何内容，程序将会崩溃并显示错误。



图7-5 为一个应用程序

例7-10

```
private: System::Void button1_Click(System::Object^ sender,
      System::EventArgs^ e)
{
      int number = Convert::ToInt32(textBox1->Text);
}
```

为了处理输入错误，可以使用例7-11中的try...catch代码。try部分测试代码，而catch状态捕获任何错误并用MessageBox类的Show成员函数显示一条信息。

例7-11

```
private: System::Void button1_Click(System::Object^ sender,
      System::EventArgs^ e)
{
      try
      {
            int number = Convert::ToInt32(textBox1->Text);
      }
      catch (...)                    //捕获任何错误
      {
            MessageBox::Show("The input must be an integer!");
      }
}
```

应用程序的其余部分在例7-12的button1_Click函数中。这个程序采用霍纳算法（Horner's algorithm）实现一个整数以2～36的任意数为基数的数制转换。这个变换算法用期望的基数除要变换的数直到结果为0。在每次除法运算后，余数作为特征数保留在结果中，而商则再次被基数除。注意，Windows没有用ASCII码，而是用到了Unicode，因此，需要用Char（16位Unicode）代替8位的char。需要注意余数放到结果串中的顺序，新的余数总是被加到串的左边。在这个例子中，每个数字被加上了0x30而变成了ASCII码。

霍纳算法：

1）用期望的基数除要转换的数。

2）保存余数，用商替换要转换的数。

3）重复第1步和第2步，直至商为0。

例7-12

```
private: System::Void button1_Click(System::Object^ sender,
       System::EventArgs^ e)
{
       String^ result = "";
       int number;
       int radix;
       try
       {
              number = Convert::ToInt32(textBox1->Text);
              radix = Convert::ToInt32(textBox2->Text);
       }
       catch (...)                    //捕获所有转换错误
       {
              MessageBox::Show("All inputs must be integers!");
       }
       if (radix < 2 || radix > 36)
       {
              MessageBox::Show("The radix must range between 2 and 36");
       }
       else
       {
              do    //转换算法
              {
                     char digit = number % radix;
                     number /= radix;
                     if (digit > 9)·                    //字符
                     {
                            digit += 7;                 //增加偏移
                     }
                     digit += 0x30;                     //转换为ASCII码
                     result = digit + result;
              }
              while (number != 0);
       }
       textBox3->Text = result;
}
```

由于本书是汇编语言教材，这是一个不用 Convert 类的很好理由，另外，Convert 类的功能太大。要想看有多大，你可以在软件中 Convert 函数的左边设置断点，点击代码行左侧灰色条就会出现一个褐色的圈，即断点。如果运行程序，将会在这一点上中断（停止）并且进入调试模式，该模式下可以以汇编语言形式查看代码。要看反汇编代码，可以运行程序直到中断，接着到 Debug 菜单下选择 Windows 子菜单，在 Windows 子菜单下，在接近底部的位置找到“Disassembly”。汇编程序单步执行的寄存器也能显示。

正如所看到的，按所述方式调试程序，如果用内嵌汇编，有大量的代码可以缩减。例7-13描述了汇编语言版的 Convert:: ToInt32 函数。这个函数非常的短小（如果在反汇编窗口下调试和查看），运行速度比例7-12中的 Convert 要快很多倍。这个例子指出了高级语言产生的代码的低效率性，这一点或许不总是很重要，但是在许多情况下，系统需要紧凑和高效率的代码时，就只能用汇编语言。我推测当处理器达到速度峰值时，更多的东西则需要用汇编语言来写。另外，像 MMX 和 SSE 这些新的指令在高级语言中也不能用，它们需要非常好的汇编语言的应用知识。

运用内嵌汇编代码的主要问题是汇编代码不能放在托管类的 Windows 托管的窗体应用程序中。为了使用汇编，函数必须在托管类之前放置，以便于编译。为此，在项目的属性中，Common Runtime Support 必须从缺省的设置/clr: pure 改为/clr，使得能够成功编译。（如何改变 Common Runtime Support 为/clr 参见图7-6。）托管程序运行在虚拟机上称为 .net。非托管应用工作在本地模式下。内嵌汇编给处理器产生了本地代码，因此它必须是非托管的并且在程序中的托管类之前驻留。

例7-13举例说明了如何用汇编语言代码替换 Adjust 函数中的霍纳算法。Adjust 函数中将数字与9

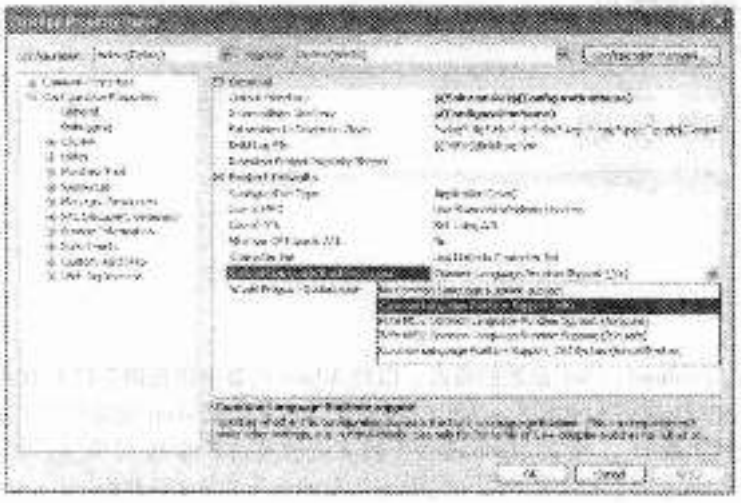


图7-6 为汇编语言改为/clr

比较，如果大于9，它将加上0x7，再加上0x30变为ASCII，最后由函数返回。注意，本例中汇编代码紧跟在程序开头using语句后。为了保证程序能正确执行，任何汇编函数必须放在这个位置。应用程序从本地模式启动，当遇到托管类时就会切换到托管模式。将汇编代码放在托管类前使得汇编代码对整个应用程序可用，汇编代码在非托管或本地模式下运行。

在例7-13结尾给出了另一个更高效的Ajust版本，这个版本没有返回指令，那么它是如何工作的呢？之所以没有出现返回值是因为任何汇编语言函数的返回值如果是字节则在AL中，如果是字或short类型则在AX中，如果是整数则在EAX中。注意返回值规定了要返回的值的长度。

例7-13

```
#pragma once

namespace FirstApp {

	using namespace System;
	using namespace System::ComponentModel;
	using namespace System::Collections;
	using namespace System::Windows::Forms;
	using namespace System::Data;
	using namespace System::Drawing;

		//short是一个16位变量类型

	short Adjust(short n)
	{
		_asm
		{
			mov  ax,n
			cmp  ax,9
			jle  later
			add  ax,7
		later:
			add  ax,30h
			mov  n,ax
		}
		return n;
	}

	 //作为一个机构序的版本
```

```
short Adjust(short n)
{
      _asm
      {
            mov  ax,n
            add  ax,30h
            cmp  ax,39h
            jle  later
            add  ax,7
      later:
      }
}

*/
//托管类紧跟其后
```

图7-14给出了对button1_click函数的修改，以便Adjust代替出现在例7-12的代码中。例7-13和例7-14出现的用于创建应用程序的代码没有给出。注意汇编函数中用short变量代替了字符变量。short是一个用在非托管模式下的16位数字，Char是一个用在托管模式下的16位数字，代表一个Unicode字符。这里用了一个Char强制转换，因为没有它将显示数值而不是ASCII代码。

例7-14

```
//在本应用程序中的唯一事件处理器

private: System::Void button1_Click(System::Object^ sender,
      System::EventArgs^ e)
{
      String^ result = "";
      try
      {
            unsigned int number = Convert::ToUInt32(textBox1->Text);
            unsigned int radix = Convert::ToUInt32(textBox2->Text);
            if (radix < 2 || radix > 36)
            {
                  MessageBox::Show("The radix must range between 2 and 36");
            }
            else
            {
                  do
                  {
                        result = (Char)Adjust(number % radix) + result;
                        number /= radix;
                  }
                  while (number != 0);
                  textBox3->Text = result;
            }
      }
      catch (...)
            //捕获任何错误
      {
            MessageBox::Show
                  ("The decimal input must between 0 and 4294967295!");
      }
}
```

7.3 汇编和C++混合目标码

正如前文所述，内嵌汇编程序是受限制的，因为它不能使用宏（MACRO）序列和第6章中介绍的条件程序流程伪指令。在有些情况下，开发独立的汇编语言模块然后再与C++连接，尤其在应用程序由一组程序员开发时，这种方法更灵活。本节详述了用汇编语言和C++两种不同目标程序链接形成一个程序的用法。本节所覆盖的内容适用于Microsoft Visual C/C++ for Windows。

7.3.1 用 Visual C++ 链接汇编语言

例 7-15 说明了可以链接到 C++ 程序中的平展模型（flat model）的子程序。我们通过在 model 语句的“flat”后用字母“C”表示该汇编模块是 C++ 语言模块。字母 C 指定的链接器对于 C 和 C++ 是一样的。平展模型可以允许汇编语言软件的最大长度为 2GB。注意，.586 开关出现在 .model 语句前，使汇编程序产生运行在 32 位保护模式下的代码。例 7-15 所示的 Reverse 过程从 C++ 程序中接受字符串，将其顺序颠倒后返回到 C++ 程序中。请注意该程序是如何使用条件程序流程指令的，这些指令在上一节描述的内嵌汇编程序中是不可用的。汇编语言模块可以任意命名，可以包含多个过程，只要每个过程中含有 PUBLIC 语句将该过程名定义为公共类型即可。C++ 程序和汇编语言程序之间传递的任何参数由跟在过程名后的反斜线符号指出。这样就可以命名汇编语言程序的参数（它可以与 C++ 中的名称不同）并指出参数的大小。在这个例子中，参数为字符串指针，返回结果替代原始串内容。

例 7-15

```
;
;外部函数反转字符串中字符的次序
;
.586                                    ;选择Pentium和32位模式
.model flat, C                          ;选择C/C++连接的flat模式
.stack 1024                             ;分配堆栈空间
.code                                   ;开始代码段

public Reverse                          ;定义Reverse过程为public类型

Reverse proc uses esi, \                ;定义过程
arraychar:ptr                           ;定义外部指针

      mov  esi,arraychar                ;寻址串
      mov  eax,0
      push eax                          ;指示串结束

      .repeat                           ;将串中字符压入堆栈
            mov  al,[esi]
            push eax
            inc  esi
      .until byte ptr [esi] == 0

      mov esi,arraychar                 ;寻址串的首地址

      .while eax != 0                   ;按倒序取出字符
            pop eax
            mov [esi],al
            inc esi
      .endw
      Ret

Reverse     endp
End
```

例 7-16 举例说明了使用 Reverse 汇编语言过程的 DOS 控制台应用的 C++ 语言程序。EXTERN 语句用来说明一个称为 Reverse 的外部过程将要在 C++ 语言程序中使用。过程的名称是大小写敏感的，因此一定要确保它在汇编语言模块和 C++ 语言模块中拼写一致。例 7-16 中的 EXTERN 语句说明外部的汇编语言过程传递了一个字符串到该过程，并且没有返回值。如果有数据从汇编语言过程返回，对于字节、字或双字返回值就是寄存器 EAX 中的值。如果返回浮点数，则它们必须返回到浮点协处理器的堆栈中。如果返回指针，那么它一定在 EAX 中。

例 7-16

```
/* 反转字符串中字符次序的程序 */

#include <stdio.h>
#include <conio.h>
```

```
extern "C" void Reverse(char *);

char chararray[1?] = "So what is this?";

int main(int argc, char* argv[])
{
  printf ("%s \n", chararray);
  Reverse (char array);
  printf ("%s\n", chararray);
  getche();                                    //等待看结果
  return 0;
}
```

一旦编写了 C++ 语言程序和汇编语言程序，就需要对 Visual C++ 开发系统进行设置，使其可以将两者链接在一起。为了链接和汇编，我们假设汇编语言模块称为 Reverse.txt（不需要给包含在项目中的文件列表添加 .asm 扩展名，使用 .txt 扩展名添加 .txt 文件即可），C++ 语言模块称为 Main.epp。两个模块都存储在 C:\PROJECT\MINE 目录下或者其他所选择的目录。将这两个模块放到相同的项目工作空间后，Programmer's WorkBench 程序就可以用来编辑汇编语言模块和 C 语言模块了。

将 Visual C++ 开发环境设置成可以编译、汇编和链接这些文件，需要完成以下步骤：

1）启动开发环境，从 File 菜单中选择 New。

a）选择 New Project。

b）当应用程序向导出现时，点击 Visual C++ 项目。

c）选择 C++ Console Application，将项目名称设定为 MINE。

d）接着点击 OK 按钮。

2）在左边靠中间位置的 Solution Explorer 窗口中可以看到项目。它有一个称为 Main.epp 的文件，这是一个 C++ 程序的文件。修改它如例 7-16 所示。

3）添加汇编语言模块。在 Source Files（源文件）行右击鼠标，选择 Add from Menu（从菜单添加），从列表中选择 Add New Item（添加新项），滚动文件类型列表直至找到 Text Files（文本文件）并选择该项，然后输入文件名称 Reverse，接着点击 Open 按钮。这样就创建了汇编模块 Reverse.txt。可以将例 7-15 的代码输入到该文件中。

4）在 Solution Explorer 窗口中列出的源文件下，在 Reverse.txt 上右击鼠标，选择 Properties（属性）。图 7-7 显示了点击 Custom Build Step（定制构造步骤）后需要在这个向导下输入的内容。确定在

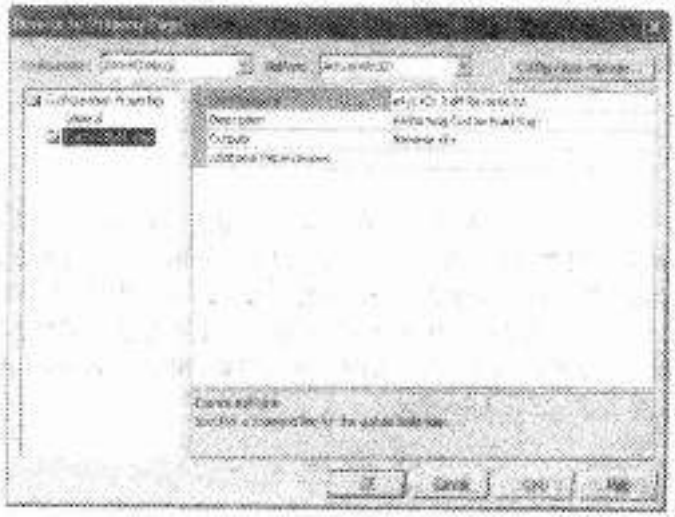


图 7-7 使用汇编程序汇编 Visual C++ 中的一个模块

输出框中输入目标文件名称（Reverse. obj），在命令行框中输入 ml /c /Cx /coff Reverse. txt。Reverse 汇编语言文件将被汇编并包含在项目中。

5）假定例 7-15 和例 7-16 都已经输入完毕并且所有步骤都已完成，那么程序就可以工作了。

最后，可以运行该程序。点击蓝色箭头，将看到显示的两行 ASCII 文本数据。第一行是正确的向前的顺序，第二行是相反的顺序。虽然这是一个微不足道的应用程序，但它说明了如何与 C++ 语言一起创建和链接汇编语言。

既然我们对汇编语言与 C++ 链接有了很好的理解，就需要一个在 C++ 语言程序中使用几个汇编过程的较长的例子。例 7-17 举例说明了一个汇编语言程序包，该程序包中有一个过程（Scan），按一个对照查找表检测字符输入，并返回标识该字符在表中相对位置的数。第 2 个过程（Look）用传给它的数返回代表摩尔斯码的字符串。（码本身并不重要，除非你感兴趣。表 7-2 列出了摩尔斯码。）

表 7-2 摩尔斯码

A	·–	J	–––	S	···
B	–···	K	–·–	T	–
C	–·–·	L	·–··	U	··–
D	–··	M	––	V	···–
E	·	N	–·	W	·––
F	··–·	O	–––	X	–··–
G	––·	P	·––·	Y	–·––
H	····	Q	––·–	Z	––··
I	··	R	·–·		

例 7-17

```
.586
.model flat, C
.data

table db  2,1,4,8,4,10,3,4      ;ABCD
      db  1,0,4,2,3,6,4,0       ;EFGH
      db  2,0,4,7,3,5,4,4       ;IJKL
      db  2,3,2,2,3,7,4,6       ;MNOP
      db  4,13,3,2,3,0,1,1      ;QRST
      db  3,1,4,1,3,3,4,9       ;UVWX
      db  4,11,4,12             ;YZ

.code
Public Scan
Public Look

Scan proc uses ebx,\
char:dword

      mov  ebx,char
      .if  bl >= 'a' && bl <= 'z'
           sub  bl,20h
      .endif
      sub  bl,41h
      add  bl,bl
      add  ebx,offset table
      mov  ax,word ptr[ebx]
      ret

Scan  endp

Look  proc uses ebx ecx,\
numb:dword,\
pntr:ptr
```

```
        mov  ebx,pntr
        mov  eax,numb
        mov  ecx,0
        mov  cl,al
        .repeat
              shr  ah,1
              .if carry?
                    mov byte ptr[ebx],'-'
              .else
                    mov byte ptr[ebx],'.'
              .endif
                    inc ebx
              .untilcxz
              mov byte ptr[ebx],0
              ret

Look          endp
end
```

A 和 Z 之间的每个字符在例 7-17 的查找表中都含有两个字节。例如，A 的代码为 2 后面跟 1。2 表示有多少个点或者长划，用来形成摩尔斯码字符，1（01）是字母 A（.—）的代码，这里 0 为点，1 为长划。Scan 过程访问查找表获取正确的摩尔斯码，它由 AX 寄存器作为参数返回给 C++ 语言调用。其余的汇编代码都很通俗易懂。

例 7-18 列出了调用例 7-17 中两个过程的 C++ 程序。这个软件很容易理解，因此不再赘述。

例 7-18

```
// Morse.cpp : 定义控制台应用程序的入口

#include <iostream>
using namespace std;

extern "C" int Scan(int);
extern "C" void Look(int, char *);

int main(int argc, char* argv[])
{
      int a = 0;
      char chararray[] = "This, is the trick!\n";
      char chararray1[10];
      while ( chararray[a] != '\n' )
      {
            if ( chararray[a] < 'A' || chararray[a] > 'z' )
                  cout << chararray[a] << '\n';
            else
            {
                  Look ( Scan ( chararray[a] ), chararray1 );
                  cout << chararray[a] << " = " << chararray1 << '\n';

            }
            a++;
      }
      cout << "Type enter to quit!";
      cin.get();
      return 0;
}
```

尽管这里给出的例子是控制台应用程序，但是 Visual Studio 汇编和链接汇编语言模块的方法同样也适用于 Windows 下的可视化应用程序。主要区别就是 Windows 应用程序不使用 printf 或 cout。第 8 章将说明库文件如何与 Visual C++ 一起使用，并给出许多编程例子。

7.3.2 在 C/C++ 程序中添加新的汇编语言指令

有时候，Intel 在推出新的微处理器的同时，也给出了新的汇编语言指令。除非为这些新指令开发在 C++ 中使用的宏，并将其包含在程序中，否则它们不能在 C++ 中使用。CPUID 指令就是一个例子。

它在 C++ 中的_asm 块中不起作用，因为内嵌汇编程序不识别它。另外一组较新的指令是 MMX 和 SEC 指令，这些指令也不被识别。为了说明如何添加汇编程序中没有的新指令，我们给出一种方法。要使用这些新指令，首先从附录 B 中或 Intel 的网站（www. intel. com）上查找机器语言代码。例如，CPUID 指令的机器码为 0F A2。这个两字节的指令可以定义为 C++ 的宏，如例 7-19 所示。在 C++ 程序中只需要输入 CPUID 就可以使用这个宏了。_emit 宏在程序中存储它后面跟着的字节。

例 7-19

```
#define CPUID _asm _emit 0x0f _asm _emit 0xa2
```

7.4 小结

1）内嵌汇编程序用于给 C++ 程序插入短的、有限的汇编语言序列。内嵌汇编程序的主要限制在于不能使用宏和带条件的程序流程指令。

2）可以获得两个版本的 C++ 语言：一个用于设计 16 位 DOS 控制台应用程序，另一个用于设计 32 位的 Windows 应用程序。应用程序类型的选择取决于环境，但是现在大多数情况下程序员都使用的是 Windows 和 32 位 Visual 版。

3）16 位汇编语言应用可以使用 DOS 的 INT 21H 功能调用访问系统中的设备。32 位汇编语言应用程序不能有效或轻易使用 DOS 的 INT 21H 功能调用，尽管许多调用还存在。

4）在 C++ 语言中与汇编语言链接最灵活、最常用的方法就是通过独立的汇编语言模块。惟一的区别就是这些独立的汇编模块必须用 C 语言的伪指令定义，在 . model 语句后定义该模块作为 C/C++ 兼容的模块链接。

5）汇编语言模块中用 PUBLIC 语句表明过程名是公用的，可以和其他模块一起使用。在汇编语言模块中通过使用 PROC 语句中的过程名称就可以定义外部参数。参数通过 EAX 寄存器从汇编语言过程返回到调用它的 C/C++ 程序中。

6）在 C++ 程序中汇编语言模块用 extern 伪指令声明为外部模块。如果 extern 伪指令后跟字母 C，那么这个伪指令用在 C/C++ 语言的程序中。

7）使用 Visual Studio 时，可以让它汇编汇编语言模块，点击模块属性，添加汇编语言程序（ml /c /Cx /coff Filename. txt），添加输出文件作为用户为该模块定制构造步骤时的目标文件（Filename. obj）。

8）汇编语言模块可以包含多个过程，但绝对不能包含使用 . startup 伪指令的程序。

7.5 习题

1. 内嵌汇编程序支持汇编语言的宏序列吗？
2. 在内嵌汇编程序中可以用伪指令 DB 定义字节吗？
3. 在内嵌汇编程序中怎样定义标号？
4. 在汇编语言（内嵌汇编程序或链接模块）中哪些寄存器不需要保存就可以使用？
5. 什么寄存器用来从汇编语言给 C++ 语言调用者返回整数数据？
6. 什么寄存器用来从汇编语言给 C++ 语言调用者返回浮点数据？
7. 在内嵌汇编程序中能够使用 . if 语句吗？
8. 解释例 7-3 中 mov dl, string1 [si] 指令是怎样访问 string1 的数据的？
9. 例 7-3 中为什么 SI 寄存器被压栈和出栈？
10. 注意在例 7-5 中没有用到 C++ 的库（#include），你认为这个程序的编译代码比用 C/C++ 语言完成相同任务的程序的编译代码小吗？为什么？
11. 在使用内嵌汇编程序时，16 位和 32 位版本 C/C++ 的主要区别是什么？
12. 用于访问 DOS 功能的 INT 21H 指令可以用在使用 32 位版本 C/C++ 编译器的程序中吗？为什么？
13. 程序中#include <conio. h> C/C++ 库用来做什么？
14. 编写一个小 C/C++ 程序，用_getche（）函数读一个键，用_putch（）函数显示一个键，输入‘@’时程序结束。
15. 不是为 PC 编写的嵌入式应用程序会使用 conio. h 库吗？
16. 在例 7-7 中，_putch（10）指令序列有何作用？接下来的_putch（13）呢？
17. 在例 7-7 中，说明一个数是如何以任意进制显示的。
18. 内嵌汇编程序或汇编语言模块连接到 C/C++ 时，哪一个应用更灵活？
19. 在汇编代码模块中 PUBLIC 语句有何作用？
20. 与 C++ 语言一起使用时，汇编代码模块应如何准备？
21. 在 C++ 语言程序中，“extern void GetIt（int）”；语句表明了关于 GetIt 函数的什么说明？
22. C++ 中 16 位字的数据如何定义？
23. 什么是 C++ Visual 程序中的控件？如何获得？
24. 什么是 C++ Visual 程序中的事件？什么是事件处理程序？
25. 在例 7-13 中，什么长度的参数是短的？
26. C++ Visual Studio 的编辑屏幕可以用来输入和编辑汇编语言程序模块吗？
27. 用汇编语言编写的外部程序怎样指示给 C++？

28. 给出RDTSC指令（操作码是0F过程31）怎样用_emit宏添加到C++程序中。
29. 解释例7-17中Scan过程使用了什么数据类型？
30. 编写一个短的可由C++使用的汇编语言模块，该程序将一个数字循环左移三位。程序名称为RotateLeft3，假定数字是一个8位char（汇编中的字节）。
31. 重复上题，但是不要用汇编语言而是用C++写相同的函数。
32. 编写一个短的汇编语言模块，该程序接收一个参数（字节），并给调用者返回一个字节大小的结果。要求该程序必须取到这个字节并将它变成大写字母。如果出现大写字母或者出现其他东西，则不修改这个字节。
33. 如何从Visual Studio执行CLR Visual C++ Express的应用程序。
34. 什么是Visual C++应用程序的属性？
35. 什么是ActiveX控件或对象？
36. 给出如何将单条汇编语言指令（如inc ptr）插入到Visual C++的程序中。

第8章　微处理器程序设计

引言

本章主要介绍如何使用 Visual C++ Express 的内嵌汇编程序进行程序设计。Visual C++ 的内嵌汇编程序虽然已经在前面章节解释和说明过，但在这一章中仍有许多内容值得学习。

本章中介绍的程序设计技术包括汇编语言模块的宏指令序列、键盘和显示器操作、程序模块、库文件、鼠标、计时器和其他重要的可编程技术的使用。本章虽然只对程序设计做了简单介绍，但却提供了非常有价值的程序设计技术，使用这些技术作为背景知识，以 Visual C++ Express 的内嵌汇编程序作为 Windows 可视化应用程序的起点，可以非常容易地为 PC 机开发软件。

目的

读者学习完本章后将能够：

1）用 MASM 汇编程序和链接程序来生成包含多个模块的程序。
2）解释用于模块化程序设计的 EXTRN 和 PUBLIC 的用法。
3）建立包含通用子程序的库文件，学会如何使用 DUMPBIN 程序。
4）使用 MACRO 和 ENDM 开发宏指令序列，这些宏指令序列用于链接到 C++ 代码的模块的线性程序设计中。
5）说明在系统中如何开发顺序存取文件和随机存取文件。
6）使用事件处理程序开发用于完成键盘操作和显示器任务的程序。
7）在程序中使用条件汇编语句。
8）在程序实例中使用鼠标。

8.1　模块化程序设计

许多程序过于庞大，无法由一个人进行开发，而需要多组程序员分块地开发。Visual Studio 提供的链接程序可将各程序模块链接起来形成一个完整的程序。链接也可由 Windows 提供的命令提示符实现。本节讲解了链接程序、任务的链接、库文件、EXTRN 和 PUBLIC，以及它们在模块化程序设计中的应用。

8.1.1　汇编程序和链接程序

汇编程序（assembler program）用来将符号化的**源模块/源文件（source module/source file）**转换成十六进制的**目标文件（object file）**，它是 Visual Studio 中的一部分，位于 C:\Program Files\Microsoft Visual Studio. NET 2003\Vc7\bin 文件夹中。在前面的章节中已经给出许多用汇编语言写的符号化源文件的例子。例 8-1 显示了汇编程序对 NEW. ASM 源模块进行汇编时的对话界面。注意，例 8-1 是在 6. 15 版 DOS 命令行下使用的对话界面。VC 版本不支持 16 位的 DOS 程序。如果需要 16 位汇编和链接，可从 Windows 的 DDK（Driver Development Kit）获得。当创建源文件时，应以 ASM 作为扩展名，但通过第 7 章的学习可知，这并不总是可行的。源文件可用记事本编辑程序来创建，也可用任何能生成 ASCII 码文件的字处理程序或编辑程序创建。

例 8-1

```
C:\masm611\BIN>ml new.asm

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.
```

```
 Assembling: new.asm

Microsoft (R) Segmented Executable Linker Version 5.60.220 Sep 9 1994
Copyright (C) Microsoft Corp 1984-1993.  All rights reserved.

Object Modules [.obj]: new.obj
Run File [new.exe]: "new.exe"
List File [nul.map]: NUL
Libraries [.lib]:
Definitions File [nul.def]:
```

汇编程序（ML）要求在ML后跟上源文件名，本例中/Fl开关用于建立一个名为NEW.LST的列表文件。虽然这一项是可选的，但建议使用它，这样就可以浏览汇编程序的输出以便调试排错。源列表文件（.LST）包含了源文件的已汇编版本和与之等效的十六进制机器代码。交叉引用文件（.CRF）列出了所有标号和交叉引用所需的相关信息，在本例中没有生成该文件。由ML产生的目标文件也作为链接程序的输入。在许多情况下，我们只需生成一个用/c开关完成的目标文件。

ML第二步执行**链接程序**（**linker program**），读取由汇编程序生成的目标文件，并且将它们链接成一个可执行文件。创建的**可执行文件**（**execution file**）其文件扩展名为EXE。通过在DOS提示符（C:\）下键入文件名来选择可执行文件。例如要执行FROG.EXE文件，则可在DOS提示符下键入FROG。

如果一个文件足够小（小于64KB），则可将它从.EXE文件转换成.COM **命令文件**（**command file**）。命令文件与可执行文件稍有不同，COM格式的程序在执行前必须起始于0100H单元处。这就意味着此程序的长度不能超过64KB－100H。如果使用微小（tiny）模型且起始地址为100H，则ML程序就生成一个COM文件。命令文件仅仅与DOS一起使用，或者在真正需要一个二进制版本（用于EPROM/FLASH编程器）时才使用命令文件。命令文件的主要优点是将它从磁盘装入计算机时比可执行文件快得多，与可执行文件等效的命令文件所占用的磁盘存储空间也少。

例8-2说明了用链接程序链接文件NEW、WHAT和DONUT时的协议。链接程序也能链接库文件（LIBS），因此LIBS中的过程也能被链接到可执行文件中。要调用链接程序，可按照例8-2说明的那样在DOS命令提示符下键入LINK。注意，在文件被链接前，它们必须先被汇编且必须没有汇编错误。ML不仅链接文件，而且还在链接之前对这些文件进行汇编。

例8-2

```
C:\masm611\BIN>ml new.asm what.asm donut.asm
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

 Assembling: new.asm
 Assembling: what.asm
 Assembling: donut.asm

Microsoft (R) Segmented Executable Linker Version 5.60.220 Sep 9 1994
Copyright (C) Microsoft Corp 1984-1993.  All rights reserved.

Object Modules [.obj]: new.obj+
Object Modules [.obj]: "what.obj"+
Object Modules [.obj]: "donut.obj"/t
Run File [new.com]: "new.com"
List File [nul.map]: NUL
Libraries [.lib]:
Definitions File [nul.def]:
```

在这个例子中，在键入ML后，链接程序要求键入由汇编程序生成的“目标模块”。在此例中有三个目标模块：NEW、WHAT和DONUT。如果存在不止一个目标文件，则首先键入主程序文件（本例中

为 NEW），然后键入其他需要链接的模块。

在文件名和开关/LINK 后可输入库文件名。在本例中并没有输入库文件名。要在汇编名为 NEW. ASM 的程序时使用名为 NUMB. LIB 的库文件，可键入 ML NEW. ASM /LINK NUMB. LIB。

在 Windows 环境下不能链接程序——只能汇编程序。在 build 过程中必须用 Visual Studio 链接程序。可以用 Visual C++ 汇编一个或多个文件并产生目标文件。例 8-3 解释了如何编译一个模块，但不是用 ML 链接。/c 开关（小写 c）告诉汇编程序编译并产生目标文件，/Cx 保存所有函数和变量，/coff 为目标文件产生一个用于 32 位环境下的目标文件的**通用目标文件格式（common object file format）**的输出。

例 8-3

```
C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\bin>ml /c /Cx /coff new.asm
Microsoft (R) Macro Assembler Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

 Assembling: new.asm
```

8.1.2 PUBLIC 和 EXTRN

PUBLIC 和 EXTRN 伪指令对模块化的程序设计非常重要，因为它们允许模块间通信。PUBLIC 用于将指令标号、数据标号或段标号声明为对其他程序模块可用。EXTRN（外部的）用于将模块中使用的一些标号声明为外部的。没有这些语句，各模块就不能链接在一起从而创建一个程序。或许它们会产生链接，但是模块之间是不能通信的。

PUBLIC 伪指令通常放在汇编语言语句的操作码域，用于将一个标号定义为公用的，以便供其他模块使用或看到。此标号可以是一个转移地址、数据地址或整个段的首地址。例 8-4 显示了如何用 PUBLIC 语句将一些标号定义为公用的，以供一个程序片段中的其他模块使用。当将一些段定义为公用的时，汇编程序将这些段与其他同名的公用段相连。

例 8-4

```
                    .model  flat,   c
                    .data

                            public Data1          ;声明Data1和Data2为public类型
                            public Data2

                    Data1   db      100 dup(?)
0000 0064[
          00
            ]
0064 0064[          Data2   db      100 dup(?)
          00
            ]
                    .code
                    .startup

                          public Read             ;声明 Read 为 public 类型

                  Read    proc    far
0006 B4 06                mov ah,6
```

EXTRN 语句可出现在数据段和代码段中，用于将标号定义为外部的。如果数据被定义为外部的，则其长度必须定义为 BYTE、WORD 或 DWORD 类型。如果一个转移或调用地址为外部的，则该地址必须定义为 NEAR 或 FAR 类型。例 8-5 说明了如何用 EXTRN 语句将一些标号定义为相对于所列程序是外部的。注意，在本例中，所有的外部地址或数据在十六进制汇编清单中均由字符 E 定义。假设例 8-4 和例 8-5 是链接在一起的。

例 8-5

```
                        .model flat, c
                        .data
                                extrn Data1:byte
                                extrn Data2:byte
                                extrn Data3:word
                                extrn Data4:dword
                         .code
                                extrn Read:far
                         .startup
0005 Bf 0000 E                  mov    dx,offset Data1
0008 B9 000A                    mov     cx,10
000B                    start:
000B 9A 000A    ---- E          call    Read
0010 AA                         stosb
0011 E2 F8                      loop    start
                        .exit
                        End
```

8.1.3 库

库文件是公共过程的集合，它们位于同一个地方，可供许多不同的应用程序使用。这些过程由 MASM 汇编程序中的 LIB 程序汇编和编译成一个库文件。在第 7 章中，当用 Visual C++ 建立汇编语言模块时，你可能会注意到许多库文件在 Visual C++ 使用的链接列表里。当使用链接程序链接一个程序时，库文件（FILENAME. LIB）被调用。

为什么要麻烦地使用库文件呢？因为库文件是存储相关过程最好的地方。当库文件与一程序相链接时，只有该程序需要的那些过程被从库文件中取出并加到该程序中去。要想高效地完成汇编语言的程序设计，一组好的库文件是必不可少的，因为可以节约重新编码公共函数的许多时间。

创建库文件

通过 LIB 命令可建立一个库文件，这个命令执行由 Visual Studio 提供的 LIB. EXE 程序。一个库文件是一组已汇编的 . OBJ 文件的集合，其中每个 . OBJ 文件包含以汇编语言或任何其他语言编写的过程或任务。例 8-6 给出了用来构成一个库文件的两个独立的函数（UpperCase 和 LowerCase），这两个文件包含在 Windows 的一个模块里。请注意，库文件中的过程名必须用 PUBLIC 声明，且不必与文件名相同，虽然在本例中是相同的。变量传递到每个文件中，因此 EXTRN 语句也出现在每一个过程中，以获得对外部变量的访问权。例 8-7 显示了在 C++ 程序中使用库文件里的函数所需的 C++ 协议。

例 8-6

```
.586
.model flat,c
.code
        public UpperCase
        public LowerCase
UpperCase proc ,\
        Data1:byte
        mov   al,Data1
        .if al >= 'a' && al <= 'z'
                sub al,20h
        .endif
        ret
UpperCase endp
Lowercase proc  ,\
        Data2:byte
        mov   al,Data2
        .if al >= 'A' && al <= 'Z'
                add al,20h
        .endif
        ret
LowerCase endp
End
```

例 8-7

```
extern "C" char UpperCase(char);
extern "C" char LowerCase(char);
```

LIB 程序首先显示一条 Microsoft 的版权信息，接着显示 Library name 提示输入库文件名，这里输入 CASE 表示要建立的库文件名为 CASE. LIB。由于这是一个新文件，故库程序被提示以目标文件名。必须首先用 ML 汇编 CASE. ASM。实际的 LIB 命令列于例 8-8 中。注意用命令行中目标文件名调用 LIB 程序。

例 8-8

```
C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\bin>lib case.obj
Microsoft (R) Library Manager Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.
```

提供名为 DUMPBIN. EXE 的实用程序以显示库或其他文件的内容。例 8-9 显示了用/all 开关显示库模块 CASE. LIB 和它的所有组件的二进制转储的结果。靠近这个清单顶部的是_ UpperCase 和_ LowerCase 的公共名字。Raw Data#1 段包括两个程序的实际的十六进制编码指令。

例 8-9

```
C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\bin>dumpbin /all case.lib

Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file case.lib

File Type: LIBRARY

Archive member name at 8: /
401D4A83 time/date Sun Feb 01 13:50:43 2004
         uid
         gid
       0 mode
      22 size
correct header end

    2 public symbols

        C8 _LowerCase
        C8 _UpperCase
Archive member name at 66: /
401D4A83 time/date Sun Feb 01 13:50:43 2004
         uid
         gid
       0 mode
      26 size
correct header end

    1 offsets

        1       C8

    2 public symbols

        1 _LowerCase
        1 _UpperCase
Archive member name at C8: case.obj/
401D43A6 time/date Sun Feb 01 13:21:26 2004
         uid
         gid
  100666 mode
     228 size
correct header end
FILE HEADER VALUES
             14C machine (x86)
```

```
              3 number of sections
       401D43A6 time date stamp Sun Feb 01 13:21:26 2004
            124 file pointer to symbol table
              D number of symbols
              0 size of optional header
              0 characteristics
SECTION HEADER #1
.text name
       0 physical address
       0 virtual address
      24 size of raw data
      8C file pointer to raw data (0000008C to 000000AF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60500020 flags
         Code
         16 byte align
         Execute Read

RAW DATA #1
  00000000: 55 8B EC 8A 45 08 3C 61 72 06 3C 7A 77 02 2C 20  U.ì.E.<ar.<zw.,
  00000010: C9 C3 55 8B EC 8A 45 08 3C 41 72 06 3C 5A 77 02  ÉAU.ì.E.<Ar.<Zw.
  00000020: 04 20 C9 C3                                      . ÉA

SECTION HEADER #2
   .data name
      24 physical address
       0 virtual address
       0 size of raw data
       0 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0500040 flags
         Initialized Data
         16 byte align
         Read Write

SECTION HEADER #3
.debug$S name
      24 physical address
       0 virtual address
      74 size of raw data
      B0 file pointer to raw data (000000B0 to 00000123)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42100040 flags
         Initialized Data
         Discardable
         1 byte align
         Read Only

RAW DATA #3
 00000000: 04 00 00 00 F1 00 00 00 00 00 00 00 30 00 01 11  ....ñ.......0...
 00000010: 00 00 00 00 43 3A 5C 50 52 4F 47 52 41 7E 31 5C  ....C:\PROGRA~1\
 00000020: 4D 49 43 52 4F 53 7E 31 2E 4E 45 54 5C 56 63 37  MICROS~1.NET\Vc7
 00000030: 5C 62 69 6E 5C 63 61 73 65 2E 6F 62 6A 00 34 00  \bin\case.obj.4.
 00000040: 16 11 03 02 00 00 05 00 00 00 00 00 00 07 00     ................
 00000050: 0A 00 05 0C 4D 69 63 72 6F 73 6F 66 74 20 28 52  ....Microsoft (R
 00000060: 29 20 4D 61 63 72 6F 20 41 73 73 65 6D 62 6C 65  ) Macro Assemble
 00000070: 72 00 00 00                                      r...

COFF SYMBOL TABLE
000 00000000 DEBUG  notype       Filename     | .file
```

```
    C:\PROGRA~1\MICROS~1.NET\Vc7\bin\case.asm
004 000F0C05 ABS    notype       Static       | @comp.id
005 00000000 SECT1  notype       Static       | .text
    Section length   24, #relocs    0, #linenums    0, checksum       0
    007 00000000 SECT2  notype       Static       | .data
        Section length    0, #relocs    0, #linenums    0, checksum       0
    009 00000000 SECT3  notype       Static       | .debug$S
        Section length   74, #relocs    0, #linenums    0, checksum       0
    00B 00000000 SECT1  notype ()    External     | _UpperCase
    00C 00000012 SECT1  notype ()    External     | _LowerCase

    String Table Size = 0x1A bytes

      Summary

               0 .data
              74 .debug$S
              24 .text
```

一旦库文件被连接到程序文件中，就只有那些被程序使用的库过程才被放入可执行文件中。当在C++程序中使用来自库文件的一个函数时，不要忘记使用外面的“C”语句。

在Visual C++ Express中，库的创建是通过选择Create菜单中的Class Libray选项来实现的。这个特征可以创建能够包含在其他任何C++应用程序中的DLL（dynamic link library，动态链接库）文件。DLL可以包含C++代码或汇编代码。在其他程序中要包含DLL，需在Project菜单下，选在“add reference”选项，浏览选择DLL文件。一旦添加DLL，就会在用到DLL的程序的类的开始处放置#include语句。

8.1.4 宏

如同过程一样，宏（macro）也是执行某一任务的指令组，它们的区别在于过程是通过“CALL”指令访问，而宏及所有在宏内定义的指令，是被插入到程序中使用它的地方。创建宏与创建新的操作码非常相似，它们实际上是一组可用于程序中的指令序列。使用宏时，键入宏的名字和与它关联的每个参数，则汇编以后，它们将被插入程序中。宏序列比过程执行得快，因为它们不执行CALL和RET指令。汇编程序将宏展开成指令序列并插入到程序中引用它们的地方。注意，宏不会用内嵌汇编程序运行，它们只运行在外部汇编模块上。

MACRO和ENDM伪指令定义了一个宏序列。宏的第一条语句是MACRO指令，包含宏的名字和与它关联的每个参数。例如MOVE MACRO A，B指令，定义了一个宏的名字为MOVE，这个新的伪操作码使用了两个参数：A和B。宏的最后一条语句是ENDM指令，它自己占一行。不要在ENDM语句前放置标号，否则此宏将不能够成功汇编。

例8-10显示了如何创建和在程序中使用宏。前六行代码定义宏。这个宏将字从存储单元B传送到存储单元A中。在此例中，宏被定义以后，被使用了两次。宏被汇编程序扩展，因此可以看到它是如何被汇编成传送程序的。任何紧跟数字（本例中为1）的十六进制机器语言语句都是宏扩展语句。这些扩展语句不键入源程序中，它们是由汇编程序生成（如果程序中包含.LISTALL）的，以显示出汇编程序已将它们插入到程序中。注意，按照惯例，在宏中的注释应以“;;”开头，而不是以习惯的“;”开头。在程序中，宏序列必须在使用之前定义，因此它们通常出现在代码段的顶部。

例8-10

```
MOVE    MACRO A,B
        PUSH  AX
        MOV   AX,B
        MOV   A,AX
        POP   AX
        ENDM

        MOVE  VAR1,VAR2      ;将VAR2移入VAR1中
```

```
0000 50            1          PUSH  AX
0001 A1 0002 R     1          MOV   AX,VAR2
0004 A3 0000 R     1          MOV   VAR1,AX
0007 58            1          POP   AX

                              MOVE  VAR3,VAR4      ;将 VAR4 移入 VAR3 中
0008 50            1          PUSH  AX
0009 A1 0006 R     1          MOV   AX,VAR4
000C A3 0004 R     1          MOV   VAR3,AX
000F 58            1          POP   AX
```

宏中的局部变量

有时宏包含局部变量。**局部变量**（**local variable**）是只能在宏的内部出现的变量，而不是宏外部的变量。为了定义局部变量，使用 LOCAL 伪指令。例 8-11 说明了用作转移地址的局部变量如何在宏内定义。如果此转移地址没有被定义为局部变量，则当第二次以及后续调用此宏时，汇编程序将指示出错。

例 8-11

```
                                FILL    MACRO WHERE, HOW_MANY     ;填充内容
                                        LOCAL FILL1
                                        PUSH  SI
                                        PUSH  CX
                                        MOV   SI,OFFSET WHERE
                                        MOV   CX,HOW_MANY
                                        MOV   AL,0
                                FILL1:  MOV   [SI],AL
                                        INC   SI
                                        LOOP  FILL1
                                        POP   CX
                                        POP   SI
                                        ENDM

                                        FILL MES1,5

                          1             LOCAL FILL1
0014  56                  1             PUSH  SI
0015  51                  1             PUSH  CX
0016  BE 0000 R           1             MOV   SI,OFFSET MES1
0019  B9 0005             1             MOV   CX,5
001C  B0 00               1             MOV   AL,0
0029  88 04               1     ??0000: MOV   [SI],AL
002B  46                  1             INC   SI
002C  E2 FB               1             LOOP  ??0000
002E  59                  1             POP   CX
002F  5E                  1             POP   SI

                                        FILL  MES2,10

                          1             LOCAL FILL1
0030  56                  1             PUSH  SI
0031  51                  1             PUSH  CX
0032  BE 0014 R           1             MOV   SI,OFFSET MES2
0035  B9 000A             1             MOV   CX,10
0038  B0 00               1             MOV   AL,0
003A  88 04               1     ??0001: MOV   [SI],AL
003C  46                  1             INC   SI
003D  E2 FB               1             LOOP  ??0001
003F  59                  1             POP   CX
0040  5E                  1             POP   SI
                                .EXIT
```

例 8-11 显示了一个 FILL 宏，它存储任意多个（参数 HOW_MANY）00H 到由参数 WHERE 标记的内存区域中。注意，当宏扩展的时候，地址 FILL1 是怎么处理的。汇编器使用以“??”开头的标号指定汇编程序生成的标号。

LOCAL 伪指令必须紧跟在 MACRO 伪指令之后，否则会发生错误。LOCAL 语句最多可以有 35 个标号，并分别用逗号隔开。

将宏定义放入它们自己的模块中

如前面所示，宏定义可以放入程序文件中，也可以放入它们自己的宏模块中。一个文件可被创建成只包括宏，而宏可供其他程序引用。我们使用 INCLUDE 伪指令指示程序文件将包含一个有一些外部宏定义的模块。虽然这不是库文件，但其实际作用却如同宏序列的一个库一样。

当把宏序列放入一个文件中时（通常以 INC 或 MAC 为扩展名），它们不像库文件那样包含 PUBLIC 语句。如果一个称为 MACRO. MAC 的文件包含宏序列，则把 INCLUDE 语句放入该程序文件中，如 INCLUDE C:\ASSM\MACRO. MAC。注意，本例中宏文件在驱动器 C 的子目录 ASSM 中。INCLUDE 语句包含这些宏，就如同将它们键入到此文件中一样。要访问被包含的宏语句，不必使用 EXTRN 语句。程序既可以包括宏包含文件，也可以包括库文件。

8.2 使用键盘和视频显示器

今天，很少有不使用键盘和视频显示的程序。本节解释如何使用连接到 IBM PC 或兼容计算机的键盘和视频显示器，这些计算机运行在 Windows 环境下。

8.2.1 读取键盘

PC 机的键盘可通过 Visual C++ 的不同对象进行读取。从键盘读入的数据为 ASCII 码形式或扩展的 ASCII 码形式。它们是以 8 位 ASCII 码或者 16 位 Unicode 码的形式存储的。如前面章节提到的那样，Unicode 包括 0000H～00FFH 代码段的 ASCII 码。其余的代码用于外语字符集。在 Visual C++ 中，我们不像 DOS C++ 控制台应用程序那样用 cin 或者 getch 来读取键盘，而是用对象来完成同样的任务。

ASCII 码数据在 1.4 节的表 1-8 中列出，表 1-9 中列出的扩展字符集是只用于打印和显示的数据，不是键盘数据。注意，表 1-8 中的 ASCII 码对应于键盘中的大多数键。通过键盘也可得到扩展的 ASCII 键码数据，表 8-1 列出了多数的扩展 ASCII 码，它们可由各种键或键组合得到。注意，键盘上的大多数键都有替换键码。每一功能键都有 4 个代码，分别通过单个功能键、Shift + 功能键、Alt + 功能键和 Ctrl + 功能键选择。

表 8-1 从键盘返回的扫描码和扩展的 ASCII 码扩展的 ASCII 码

键	扫描码	无	Shift	Control	替换	键	扫描码	无	Shift	Control	替换
Esc	01				01	T	14				14
1	02				78	Y	15				15
2	03			03	79	U	16				16
3	04				7A	I	17				17
4	05				7B	O	18				18
5	06				7C	P	19				19
6	07				7D	[	1A				1A
7	08				7E	]	1B				1B
8	09				7F	Enter	1C				1C
9	0A				80	Enter	1C				A6
0	0B				81	Lctrl	1D				
-	0C				82	Rctrl	1D				
+	0D				83	A	1E				1E
Bksp	0E				0E	S	1F				1F
Tab	0F		0F	94	A5	D	20				20
Q	10				10	F	21				21
W	11				11	G	22				22
E	12				12	H	23				23
R	13				13	J	24				24

（续）

键	扫描码	Alt	Shift	Control	分离
K	25				25
L	26				26
;	27				27
'	28				28
`	29				29
LShft	2A				
\	2B				
Z	2C				2C
X	2D				2D
C	2E				2E
V	2F				2F
B	30				30
N	31				31
M	32				32
,	33				33
.	34				34
/	35				35
Gray/	35			95	A4
RShft	36				
PrtSc	E0 2A E0 37				
LAlt	38				
RAlt	38				
Space	39				
Caps	3A				
F1	3B	3D	54	5E	68
F2	3C	3E	55	5F	69

键	扫描码	无	Shift	Control	分离
F3	3D	3F	56	60	6A
F4	3E	40	57	61	6B
F5	3F	41	58	62	6C
F6	40	42	59	63	6D
F7	41	43	5A	64	6E
F8	42	44	5B	65	6F
F9	43	45	5C	66	70
F10	44	46	5D	67	71
F11	57	85	87	89	8B
F12	58	86	88	8A	8C
Num	45				
Scroll	46				
Home	E0 47	47	47	77	97
Up	48	48	48	8D	98
Pgup	E0 49	49	49	84	99
Gray −	4A				
Left	4B	4B	4B	73	9B
Center	4C				
Right	4D	4D	4D	74	9D
Gray +	4E				
End	E0 4F	4F	4F	75	9F
Down	E0 50	50	50	91	A0
Pgdn	E0 51	51	51	76	A1
Ins	E0 52	52	52	92	A2
Del	E0 53	53	53	93	A3
Pause	E0 10 45				

创建一个包含简单文本框的 Visual C++ Express 应用程序，便于理解在 Windows 下如何读一个按键。图 8-1 所示了这样一个基于窗体的应用程序。记得需要创建一个基于窗体的应用：

1）启动 Visual C++ Express；

2）点击 Create：Project；

3）选择 CLR Windows Forms Application，然后给一个名字并点击 OK 按钮。

新的基于窗体的应用创建后，可以从工具箱中选择文本框控件，在对话框屏幕上画一个文本框，如图 8-1 所示。



图 8-1 使用增加能的 textbox

设置焦点

给应用程序添加的第一件事情就是将焦点设置到文本框控件上。焦点一旦设置，光标就会移动到对象上，在本例中焦点就是文本框。因为本例中文本框控件的名称为 textBox1，通过 textBox1 -> Focus（）给该控件设置焦点。这条语句放在 Form_Load 函数中，该函数是通过在窗体空白区域双击来安装的。Form_Load 函数也可以通过其他方法来添加，点上黄色高亮闪电图标并选择 Load。读者通过在其右边的空白文本框上双击完成添加。应用程序就会在启动时将当前的焦点设置到文本框上。这就意味着闪烁的光标将会出现在文本框控件里。

当应用程序运行并且文本框有输入时，程序就会读取键盘并显示每一个输入的键。在有些情况下，这并不是所期望的，或许还需要一些过滤。情况之一就是应用程序需要用户输入十六进制数据。为了查找输入的按键，文本框需要添加事件处理函数 KeyDown 和 KeyPress。KeyDown 事件处理函数在键按下时调用，它会伴随着 KeyPress 事件处理函数的调用。要给文本框控件插入这些函数到应用程序中，首先要点击文本框并选择属性窗口，接着找到黄色高亮闪电图标并点击，然后为文本框控件添加

KeyDown 和 Keypress 事件处理函数。

为了举例说明过滤，本应用程序用 KeyDown 函数在程序用按键之前查看每一个输入的字符。这使得字符可以修改。在这里，应用程序仅允许数字 0～9 和字母 A～F 可以从键盘输入。如果输入字母为小写，就将其变成大写，如果输入其他键则忽略。

完成过滤，用 KeyEventArgs^ 类参数 e，它被传递到 KeyDown 函数中，如例 8-12 所示。在这个例子中，用 C++ 完成从键盘输入到文本框控件的过滤任务。变量 keyHandled 用来指示键是否进行过滤处理。如果 keyHandled 为假，表明该键未被过滤将出现在文本框上。同样，如果 keyHandled 为真，表明该键被过滤处理将不出现在文本框上。keyHandled 条件被传给在例 8-12 中出现的 KeyPress 事件中。注意，Keys::D0～Keys::D9 是 QWERTY 键盘上的数字键，Keys::NumberPad0～Keys::NumberPad9 是数字键盘上的数字键。未按上档键（Shift）的 D8 是 8 键，按下上档键（Shift）的 D8 为星号（*）键。

例 8-12

```
private: System::Void Form1_Load(System::Object^ sender,
        System::EventArgs^ e)
{
        textBox1->Focus();          // 设置 textbox1 为焦点
}

bool keyHandled;

private: System::Void textBox1_KeyDown(System::Object^ sender,
        System::Windows::Forms::KeyEventArgs^ e)
{       // 第一次调用
        keyHandled = true;
        if (e->KeyCode >= Keys::NumPad0 && e->KeyCode <= Keys::NumPad9 ||
                e->KeyCode >= Keys::D0 && e->KeyCode <= Keys::D9 &&
                e->Shift == false ||
                e->KeyCode >= Keys::A && e->KeyCode <= Keys::F ||
                e->KeyCode == Keys::Back)
        {
                keyHandled = false;
        }
}

private: System::Void textBox1_KeyPress(System::Object^ sender,
        System::Windows::Forms::KeyPressEventArgs^ e)
{       // 第二次调用
        if (e->KeyChar >= 'a' && e->KeyChar <= 'f')
        {
                e->KeyChar -= 32;           // 变成大写
        }
        e->Handled = keyHandled;
}
```

如果 KeyPress 事件中的语句检测到 e -> KeyCode 的值为字母 a、b、c、d、e 和 f，它不是大写就是小写。KeyDown 事件检测键盘和数字键盘上的数字 0～9。后退键（backspace）也被检测。如果按下这些键，keyHandled 将被设置为假，表明这些键未被 KeyDown 函数处理。KeyPress 事件确定字母 a～f 是否输入，并通过减 32 变为大写。32 是 ASCII 编码中大写和小写字母之间的偏移值。接着，e -> Handled 的返回值被置为真或假。返回真，Windows 就会丢弃此按键。在这种情况下，如果输入为数字键，或者字母 a～f（或 A～F），KeyPress 函数结尾的正常返回值为假，按键就会被传到 Windows 的框架中，这样就只有按键 A～F 或 0～9 出现在编辑框中。

在例 8-13 中，用内嵌汇编重复完成例 8-12 中的相同任务。这里，称为 filter 的函数返回真或假，传给 KeyDown 函数中的 KeyHandled 变量。在这个例子中，C++ 代码好像比汇编语言代码少，但是这些 C++ 代码对于能够可视化两个窗体是非常重要的。不要忘记将项目的属性从公共语言运行库支持（Common Language Runtime Suport）改变为公共语言运行库（CLR），这样才能正常工作（见第 7

章)。注意在汇编版中用 Key 给 Filter 函数传递字符。还要注意，返回值为整数 0 则是假，为整数 1 则是真。

例 8-13

```
//在程序的顶部放置下列使用声明
int Filter(char key)
{
        int retval;          // 0 = false, 1 = true
        _asm
        {
                mov eax,1
                cmp key,8            ; backspace
                jb good
                cmp key,30h
                jb bad
                cmp key,39h
                jbe good
                cmp key,41h
                jb bad
                cmp key,46h
                jbe good
                cmp key,61h
                jb bad
                cmp key,66h
                jbe good
good:           dec eax
bad:            mov retval,eax
        }
        return retval;
}


private: System::Void Form1_Load(System::Object^ sender,
        System::EventArgs^ e)
{
        textBox1->Focus();
}

bool keyHandled;

//textbox1_Key Down 的新版本

private: System::Void textBox1_KeyDown(System::Object^ sender,
        System::Windows::Forms::KeyEventArgs^ e)
{
        keyHandled = Filter(e->KeyValue);
}
private: System::Void textBox1_KeyPress(System::Object^ sender,
        System::Windows::Forms::KeyPressEventArgs^ e)
{
        if (e->KeyChar >= 'a' && e->KeyChar <= 'f')
        {
                e->KeyChar -= 32;
        }
        e->Handled = keyHandled;
}
```

如果这个代码被添加到应用程序中并运行，文本框控件中只显示 0～9 和 A～F 键。在 Visual C++ Express 环境中其他许多过滤都可以用这样的方式来实现。在文本框控件的属性中包括了 Character Casing 属性，可以将键入的所有字符更改为大写字符，减少过滤任务，但在这里是用软件实现这个大写特性的。

8.2.2 使用视频显示器

正如键盘那样，Visual C++ 中用对象显示信息。和其他很多对象一样文本框控件能用来读取或显

示数据。修改图 8-1 中的应用，使其包含另外一个文本控件，如图 8-2 所示。注意给窗体中添加了一些标识文本框控件内容的标签控件。在这个新的应用程序中，键盘数据仍然从 textBox1 文本框控件中读入，但是当回车（Enter）键按下后，textBox1 中输入的十六进制数的十进制形式就会显示在 textBox2——第二个文本框控件上。为了保持与这里所给软件的需要保证第二个控件名称为 textBox2。



图 8-2 十六进制数转换成十进制数

为了使程序对回车键响应，该键的 ASCII 码为 13（0DH 或 0x0d），将例 8-13 中的 KeyPress 函数修改为例 8-14所示。注意：回车键用 else if 检测。当检测到回车键时，textBox1 中的内容就被转换为 textBox2 显示的十进制，如例 8-15 所示。

例 8-14

```
private: System::Void textBox1_KeyPress(System::Object^ sender,
        System::Windows::Forms::KeyPressEventArgs^ e)
{
        if (e->KeyChar >= 'a' && e->KeyChar <= 'f')
        {
                e->KeyChar -= 32;
        }
        else if (e->KeyChar == 13)
        {
                //列下在 textBox2 中显示十进制的软件

                keyHandled = true;
        }
        e->Handled = keyHandled;
}
```

文本框数据有一个小问题，输入到文本框控件的数据是以字符串访问的，而不是十六进制数。在这个例子中（见例 8-15），Convert 函数将十六进制字符串转换为数。这个程序中现在有两个含有汇编代码的函数。

例 8-15

```
// 放置在程序顶部的适用声明之后
int Filter(char key)
{
        int retval;
        _asm
        {
                mov eax,1
                cmp key,8           ; backspace
                je  good
                cmp key,30h
                jb  bad
                cmp key,39h
                jbe good
                cmp key,41h
                jb  bad
                cmp key,46h
                jbe good
                cmp key,61h
                jb  bad
                cmp key,66h
                jbe good
good:           dec eax
bad:            mov retval,eax
```

```
        }
        return retval;
}

int Converts(int number, short digit)
{
        _asm
        {
                mov eax,number
                shl eax,4
                mov dx,digit
                sub dx,30h
                cmp dx,9
                jbe later
                sub dx,7
later:          or  al,dl
                mov number,eax
        }
        return number;
}
private: System::Void Form1_Load(System::Object^ sender,
        System::EventArgs^ e)
{
        textBox1->Focus();
}

bool keyHandled;

private: System::Void textBox1_KeyDown(System::Object^ sender,
        System::Windows::Forms::KeyEventArgs^ e)
{
        keyHandled = Filter(e->KeyValue);
}
private: System::Void textBox1_KeyPress(System::Object^ sender,
        System::Windows::Forms::KeyPressEventArgs^ e)
{
        if (e->KeyChar >= 'a' && e->KeyChar <= 'f')
        {
                e->KeyChar -= 32;
        }
        else if (e->KeyChar == 13)
        {
                int number = 0;
                for (int a = 0; a < textBox1->Text->Length; a++)
                {
                        number = Converts(number, textBox1->Text[a]);
                }
                textBox2->Text = Convert::ToString(number);
                keyHandled = true;
        }
        e->Handled = keyHandled;
}
```

例8-15给出了一个完整的应用。当回车键按下时，程序从textBox1获得一个字符串，一个字符一个字符地调用Converts函数将该串转换为整数。整数产生后，用Convert类的成员函数ToString又将其转换为textBox2显示的字符串。

Converts函数中的汇编语言通过给每个数字减30h，从而使ASCII转换成二进制。这个动作将ASCII数字30H～39H转换成二进制0～9。这里没有转换字母41H～46H（A到F）到二进制，因为其结果是11H～16H，而不是0AH～0FH。为了调整从字母产生的值，用cmp（比较）指令检测11H～16H范围，然后再多减去7，将11H～16H转换为0AH～0FH。ASCII数字转换为二进制格式后，整数数字（暂存于EAX中）被左移二进制的四位后，与二进制格式的ASCII数字（暂存于DX中）进行或运算，运算结果保存到temp1（number）中。

当来自textBox1的十六进制数被转换成二进制数字，用Convert类中的ToString函数显示在textBox2

中。如前所述，返回值为真则通知 Windows 界面回车键被程序处理。如果在 KeyPress 中 Enter 键返回值为假，计算机将发出哪嘟的错误提示音。

8.2.3 在程序中使用定时器

定时器在编程中非常重要。定时器被编程在一定的时间后（以毫秒为单位）激发或触发。定时器允许的时间范围为 1～2G 毫秒。这使得定时器可被编程为任意所需的时间。如果被编程为 1000，定时器 1 秒激发，如果被编程为 60 000，定时器 1 分钟激发，等等。程序中定时器可以用作单次或多次，或者需要的任意多次（最高 20 亿次）。定时器可以在设计窗口的工具箱中找到。

为了举例说明定时器及其在程序中的用法，图 8-3 给出了一个设计，用户可演示二进制数的移位或循环移位。在这个程序中移位和循环移位用定时器激励。设计包含有两个标签控件，两个按钮控件和一个定时器控件。将这五个控件添加到该应用程序中。定时器控件不在窗体上出现，而出现在接近设计屏幕底部的区域上。注意：这个窗体的一些属性被做了修改，不显示图标，文本属性被设置为 Shift/Rotate，而不是 Form1。同样，标签和按钮的文本属性也做了相应修改。

当窗体看上去如图 8-3 所示时，就可以为两个命令按钮和定时器添加事件处理函数（点击黄色高亮闪电图标）。对于按钮添加点击事件处理函数，对于定时器添加嘀嗒事件处理函数。要添加事件处理函数，首先点击按钮或定时器，接着在屏幕右边的属性窗口点击黄色高亮闪电图标选择事件。该程序有三个事件处理函数，两个是按钮的点击，一个是定时器嘀嗒。

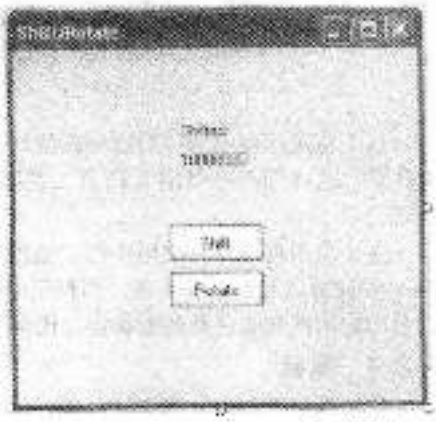


图 8-3 Shift/Rotate 应用设计截图

在设计屏幕找到定时器的属性，将间隔设为 500，即 1/2 秒。不要使能定时器。定时器在点击以 1/2 秒速度循环移位或移位一个数的按钮时由软件来使能。

例 8-16 举例说明了实现图 8-3 所示应用程序需要在三个事件处理函数里添加的软件。除了布尔（Boolean）变量 shift，两个按钮点击事件处理函数的软件几乎相同。每个函数中都用两条语句将文本显示到标签上。如果 shift（移位）按钮按下，label1 就会显示 "Shifted"，如果 rotate（循环移位）按下，label1 就会显示 "Rotated"。第二标签显示初始数据 "00011101"。布尔变量 shift 由 Shift 按钮设为真，由 Rotate 按钮设为假。在两个按钮事件处理函数中 count 设为 8，表示移位 8 位或循环移位 8 位。最后，每个按钮事件处理函数的最后一条语句就是启动定时器。一旦定时器启动或使能，在 1/2 秒激发并调用 timer1 的嘀嗒事件处理函数，程序中的所有工作将在该函数中实现。

定时器嘀嗒函数的第一条语句是将数字串清零。这个是要移到 label2 最右边的数字。对于移位，最右边移入的数将保持 0。对于循环移位，最右边移入的数将依赖于 label2 左边最高位的数字。如果移位是假（即循环移位），label2 左边最高位为 1，该位数就变为 1。If 语句后，数字就会被移位或循环移位并放到 label2 中。4 秒 8 次循环移位后，如果 count 到 0，定时器就会通过设置 Enabled 成员为假而停止激活。

例 8-16

```
bool shift;
int count;

private: System::Void button1_Click(System::Object^ sender,
         System::EventArgs^ e)
{
         label1->Text = "Shifted";
         label2->Text = "00011001";
         shift = true;
         count = 8;
```

```
        timer1->Start();
}
private: System::Void button2_Click(System::Object^ sender,
        System::EventArgs^ e)
{
        label1->Text = "Rotated";
        label2->Text = "00011001";
        shift = false;
        count = 8;
        timer1->Start();
}

private: System::Void timer1_Tick(System::Object^ sender,
        System::EventArgs^ e)
{
        String^ digit = "0";
        if (shift == false && label2->Text[0] == '1')
        {
                digit = "1";
        }
        label2->Text = label2->Text->Remove(0,1) + digit;
        if (--count == 0)
        {
                timer1->Enabled = false;
        }
}
```

这个应用程序添加一对选择移位（或旋转）方向为左边和右边的单选按钮。Label2 也可以用文本框代替，这样用户可以输入任意二进制数（通过适当的验证），可以进行左移位（旋转）或右移位（旋转）。

这个应用程序没有使用任何汇编语言，但是，如果在定时器函数中插入断点，在 Visual C++ Express 中就可以看到汇编代码。当程序中断时，到 Debug 菜单中选择窗口，接着，选择反汇编窗口就可以显示定时器响应函数的汇编语言代码了。

8.2.4 鼠标

鼠标指示设备以及滚轮可以通过 Visual C++ Express 的框架访问。与 Windows 控制的其他设备一样，鼠标也可以在应用程序中添加消息处理函数，以便程序中可以使用鼠标移动和其他功能。如前面的例子那样，应用程序中添加消息处理函数（事件处理函数）可通过在 Properties 部分中点击高亮条右边的图标完成。鼠标消息处理函数如表 8-2 所示。

为了说明如何使用鼠标，可参考图 8-4 所示的应用程序。这个例子显示了在对话框应用程序中随着鼠标移动鼠标指示点的坐标。尽管这个程序（例 8-17）没有用到汇编语言，但是它说明了如何获得并显示鼠标指示的位置。注意看如何使用 MouseEventArgs^ 获得使用 X 和 Y 坐标的鼠标指示的位置。

表 8-2 鼠标消息

处理函数	触发器
MouseDown	鼠标键按下
MouseEnter	鼠标指针指向控件
MouseHover	鼠标在一段时间内没有移动
MouseLeave	鼠标指针离开控件
MouseMove	鼠标移动
MouseUp	鼠标键释放



图 8-4 显示鼠标的坐标

例 8-17

```
private: System::Void Form1_MouseMove(System::Object^ sender,
       System::Windows::Forms::MouseEventArgs^ e)
{
       label1->Text = "X-coordinate = " + Convert::ToString(e->Location.X);
       label2->Text = "Y-coordinate = " + Convert::ToString(e->Location.Y);
}
```

例 8-17 给出了应用程序中用于显示鼠标坐标的惟一修改的部分。MouseMove 函数在程序的 MouseMove 事件处理函数安装时安装。这个应用程序用了两个标签控件显示鼠标的坐标。这两个对象在应用程序中命名为 label1 和 label2。MouseMove 函数在 Location 数据结构的 X 和 Y 成员中返回鼠标指针的位置。这个例子用 Covert 类将作为鼠标点位置 X 和 Y 返回的整数变换为放在标签上的 ASCII 字符串。

在这个应用程序中，鼠标指针没有跟踪两个标签的位置。为了在程序中能够显示窗体中的标签的坐标，需要为两个标签安装 MouseMove 处理函数。例 8-18 给出了另外的两个 MouseMove 函数以及标签跟踪的 X 和 Y 坐标需要加的偏移值。偏移值从何处获取呢？偏移值在每一个标签的 Location 属性中，在该属性中 X 和 Y 给出了标签的位置。在应用程序中，这个数字依赖于标签放置到窗体中的位置，通过 label1 -> Location. X 等获取。

例 8-18

```
private: System::Void Form1_MouseMove(System::Object^ sender,
       System::Windows::Forms::MouseEventArgs^ 'e)
{
       label1->Text = "X-coordinate = " + Convert::ToString(e->Location.X);
       label2->Text = "Y-coordinate = " + Convert::ToString(e->Location.Y);
}
private: System::Void label1_MouseMove(System::Object^ sender,
       System::Windows::Forms::MouseEventArgs^ e)
{
       label1->Text = "X-coordinate = " + Convert::ToString(e->Location.X+159);
       label2->Text = "Y-coordinate = " + Convert::ToString(e->Location.Y+232);
}
private: System::Void label2_MouseMove(System::Object^ sender,
       System::Windows::Forms::MouseEventArgs^ e)
{
       label1->Text = "X-coordinate = " + Convert::ToString(e->Location.X+159);
       label2->Text = "Y-coordinate = " + Convert::ToString(e->Location.Y+246);
}
```

为 MouseDown 事件安装鼠标处理函数。添加如例 8-19 所示的 MouseDown 事件处理函数来修改应用。该函数使得鼠标左键点击时颜色标号变为红色，鼠标右键按下时颜色标号变为蓝色。在 Visual C++ 的许多函数中用到的最常见的颜色编码可以在 Color 类中找到。这个应用使用 Button 的 MouseEventArgs 对象所示的成员测试鼠标左键和右键。（微软为 MouseButtons 枚举选择名称 mouses。）

例 8-19

```
private: System::Void Form1_MouseDown(System::Object^ sender,
       System::Windows::Forms::MouseEventArgs^ e)
{
       if (e->Button == ::mouses::MouseButtons::Left)
       {
              label1->ForeColor = Color::Red;
              label2->ForeColor = Color::Red;
       }
       else if (e->Button == ::mouses::MouseButtons::Right)
       {
                label1->ForeColor = Color::Blue;
                label2->ForeColor = Color::Blue;
       }
}
```

8.3 数据转换

在计算机系统中，数据很少以恰当的形式表示。系统的一个主要任务就是将数据从一种形式转换成另一种形式。本节将主要介绍二进制和 ASCII 码之间的转换。为了在视频显示器上显示，需要将寄存器或存储单元中的二进制数移出，并且转换成 ASCII 码。许多情况下，从键盘键入时，ASCII 码数据要转换为二进制。本节还将介绍 ASCII 码和十六进制数之间的转换。

8.3.1 二进制转换为 ASCII 码

有三种方法完成从二进制到 ASCII 码的转换：（1）如果数据小于 100，则用 AAM 指令实现（64 位扩展不用于转换）；（2）通过一系列的十进制除法（除以 10）实现；（3）用 C++ 的转换类型功能函数 ToString。前两种技术在本节中都有说明。

AAM 指令将 AX 中的数值转换为两位非压缩的 BCD 码，存入 AX 中。例如，如果执行 AAM 指令前，AX 中的值为 0062H（即十进制数的 98），则执行 AAM 指令后，AX 中的内容为 0908H。这虽不是 ASCII 码，但可通过加 3030H 转换为 ASCII 码。例 8-20 给出了一个使用该过程的程序，该过程处理 AL 中的二进制数（0～99），并以十进制的形式显示在屏幕上。这一过程在以 0 开始的数字 0～9 前面填入空格（即空格的 ASCII 码）。该程序在屏幕上显示数字 74（测试数据）。为了实现这个程序，需要在 Visual C++ 中创建一个基于对话框的程序，并在对话框上放置一个名为 Label1 的标签在程序的开头部分最后的 using 语句后，如果加入例 8-20 中的汇编语言函数，则数字 74 将出现，该项目将变为/CLR 程序。对汇编语言函数的调用放在窗体的 Load 事件的处理函数中。

例 8-20

```
// 置于程序的开始部分
// 在 64 位模式中不起作用
void ConvertAam(char number, char* data)
{
        _asm
        {
                mov ebx,data            ; 指向 ebx
                mov al,number           ; 获取测试数据
                mov ah,0                ; 清空 AH
                aam                     ; 转换 BCD 码
                add ah,20h
                cmp al,20h              ; 测试是 0 开头吗
                je D1                   ; 如是 0 开头则跳转
                add ah,10h              ; 转换为 ASCII 码
D1:
                mov [ebx], ah
                add al,30h              ; 转换为 ASCII 码
                mov [ebx+1], al
        }
}


private: System::Void Form1_Load(System::Object^ sender,
        System::EventArgs^ e)
{
        char temp[2];                           //放置结果
        ConvertAam(74, temp);
        Char a = temp[0];
        Char b = temp[1];
        label1->Text = Convert::ToString(a) + Convert::ToString(b);
}
```

AAM 指令将任何 0～99 之间的数转换为两位的非压缩 BCD 码，是因为它将 AX 中的数据除以 10，结果存入 AX 的左端，因此 AH 中为商，AL 中为余数。这种除以 10 的方式可以扩展，用于将任意进制数以二进制转换为 ASCII 码字符串，以便能够在视频显示器上显示。例如，如果将 AX 中的数据除以 8

而不是除以 10，则可以八进制形式显示该数。

二进制转换成 ASCII 码的 **Horner 算法**（**Horner's algorithm**）如下：

1）除以 10，然后将余数存入堆栈作为 BCD 数的有效位。

2）重复第 1 步，直到商为 0。

3）取出每一位余数，并加上 30H 将其转换成 ASCII 码，以便显示或打印。

例 8-21 给出了 EAX 中的 32 位无符号数如何转换成 ASCII 码，并且在视频显示屏上显示。程序中，EAX 内容除以 10，并将每次除法后的余数存入堆栈，以便最后转换为 ASCII 码。全部数字都转换完后，从堆栈移出余数，转换为 ASCII 码，在视频显示屏上显示。该程序不显示数字的前导 0。正如上面提到的那样，本例中可以通过改变基数的变量来使用任意数字基数。而且，为了实现这个例子，可以用/CLR 选项和一个名为 Label1 的标签创建一个窗口应用程序。如果基数大于 10，用字母表示超过 9 的字符。软件支持的基数为 2 ～ 36。

例 8-21

```
void Converts(int number, int radix, char* data)
{
      _asm
      {
            mov   ebx,data          ; 初始化指针
            push  radix
            mov   eax, number       ; 获得测试数据
L1:
            mov   edx,0             ;edx 清零
            div   radix             ; 除以基数
            push  edx               ; 保存余数
            cmp   eax,0
            jnz   L1                ; 重复直至为 0
L2:
            pop   edx               ; 取余数
            cmp   edx,radix
            je    L4                ; 如果结束
            add   dl,30h            ; 转换为 ASCII 码
            cmp   dl,39h
            jbe   L3
            add   dl,7
L3:
            mov   [ebx],dl          ; 取数
            inc   ebx               ; 指向下一个
            jmp   12                ; 重复直至完成
L4:
            mov   byte ptr[ebx],0   ; 在串中存 0
      }
}


private: System::Void Form1_Load(System::Object^ sender,
      System::EventArgs^ e)
{
      char temp[32];                    // 放置结果
      Converts(7423, 10, temp);
      String^ a = "";
      int count = 0;
      while (temp[count] != 0)          // 转换成字符串
      {
            Char b = temp[count++];
            a += b;
      }
      label1->Text = a;
}
```

8.3.2 ASCII 码转换为二进制

ASCII 码转换为二进制通常从键盘输入开始。如果键入单键，则从该数中减去 30H 实现转换；如

果键入的多于一个键，则转换仍然从键值中减去30H，但是增加了一个步骤。减去30H以后，前面的结果先乘10，再加上刚才所得的数字。

将ASCII码转换成二进制数的算法如下：

1）将二进制结果单元清0。

2）从键盘键入的字符中减去30H，把它转换为BCD码。

3）结果乘以10，再加上新的BCD数字。

4）重复第2步和第3步，直到键入的字符不是ASCII编码的数字为止。

例8-22给出的程序实现了此算法。这里，用Covert类型函数从所关联的变量temp中获得并显示二进制数字。程序每执行一次，就从字符变量numb中读一个数字，并把它转换为二进制数字，显示在标签上。

例8-22

```
int ConvertAscii(char* data)
{
       int number = 0;
       _asm
       {
              mov  ebx,data     ; 初始化指针
              mov  ecx,0
B1:
              mov  cl,[ebx]     ; 取数
              inc  ebx          ; 寻址下一个数
              cmp  cl,0         ; 如发现是空
              je   B2
              sub  cl,30h       ; 从ASCII码转换为BCD码
              mov  eax,10       ; 乘以10
              mul  number
              add  eax,ecx      ; 将数累加
              mov  number,eax   ; 存结果
              jmp  B1
B2:
       }
       return number;
}


private: System::Void Form1_Load(System::Object^ sender,
       System::EventArgs^ e)
{
       char temp[] = "2316";                    // 字符串
       int number = ConvertAscii(temp);
       label1->Text = Convert::ToString(number);
}
```

8.3.3 显示和读入十六进制数

与十进制数相比，十六进制数据更容易从键盘读入和显示。这种数据用于系统层而不用于应用层。系统层数据通常为十六进制，并且必须以十六进制的形式显示或从键盘读入。

读入十六进制数据

十六进制数据由0～9和A～F组成。从键盘键入的十六进制数的ASCII码，30H～39H表示数字0～9，而41H～46H（A～F）或61H～66H（a～f）表示字母。为了使程序更实用，读入十六进制数据时，必须既能接收小写字母，又能接收大写字母。

例8-23给出了两个函数：一个（Conv）将无符号字符的内容从ASCII码转换为一位十六进制数字；另一个（Readh）将8位十六进制数字的CString转换为数字，它是以32位无符号整数的形式返回的。这个例子用一个C++和汇编语言的混合程序来完成转换。

例 8-23

```
unsigned char Conv(unsigned char temp)
{
       _asm
       {
              cmp   temp,'9'
              jbe   Conv2            ;如果为0~9
              cmp   temp,'a'
              jb    Conv1            ;如果为A~F
              sub   temp,20h         ;转换为大写字母
Conv1:
              sub   temp,7
Conv2:
              sub   temp,30h
       }
       return temp;
}


private: System::UInt32 ReadH(String^ temp)
{
       unsigned int numb = 0;
       for ( int a = 0; a < temp->Length; a++ )
       {
              numb <= 4;
              numb += Conv(temp[a]);
       }
       return numb;
}


private: System::Void Form1_Load(System::Object^ sender,
       System::EventArgs^ e)
{
       unsigned int temp = ReadH("2AB4");
       label1->Text = Convert::ToString(temp);        //用十进制显示
}
```

显示十六进制数据

为显示十六进制数据，必须把数分解成2、4或8位的段，并将每段转换为十六进制数字。通过将30H加到数字0至9上，将37H加到字母A至F上完成这类转换。

Disph函数，返回一个string，其内容为传递给该函数的无符号整数。该函数把无符号整数转换为2、4或8位的字符串。该函数如例8-24所示。Disph（number，2）把一个无符号整数转换为两位十六进制的string对象，Disph(number，4)把一个无符号整数转换为四位十六进制的string对象，Disph(number，8)把一个无符号整数转换为八位十六进制的字符string。

例 8-24

```
void Disph(unsigned int number, unsigned int size, char* temp)
{
       int a;
       number <i<= ( 8 - size ) * 4;      //对准位置
       for (a = 0; a < size; a++)
       {
              char temp1;
              _asm
              {
                     rol number, 4;
                     mov al,byte ptr number
                     and al,0fh            ;产生0~f
                     add al,30h            ;转换为ASCII码
                     cmp al,39h
                     jbe Disph1
                     add al,7
```

```
Disph1:
                    mov temp1,al
                }
                temp[a] = temp1;                    //把数字加到数据串
        }
        temp[a] = 0;                                //以空字符串结束
}
private: System::Void Form1_Load(System::Object^ sender,
        System::EventArgs^ e)
{
        char temp[9];
        Disph(1000,4,temp);
        String^ a = "";
        int count = 0;
        while (temp[count] != 0)                    //转换为字符串
        {
            Char b = temp[count++];
            a += b;
        }
        label1->Text = a;
}
```

8.3.4 使用查找表实现数据转换

将数据从一种形式转换为另一种形式时，经常使用查找表。查找表是存储器中的数据表，由实现转换的过程引用。对于许多查找表，通常可用 XLAT 指令查找表中的数据，所提供的表包含 8 位宽的数据，长度小于或等于 256 字节。

将 BCD 码转换为 7 段码

一个使用查找表的简单应用是 BCD 码到 7 段码的转换。例 8-25 给出了含有数字 0～9 的 7 段码的一个查找表。这些代码用于如图 8-5 所示的 7 段显示器。这种 7 段显示器用高电平（逻辑 1）输入点亮某个段。查找表代码（数组 temp1）的排列是：第 0 位控制段 a，第 6 位控制段 g。本例中第 7 位为 0，但它可根据需要用来显示小数点。

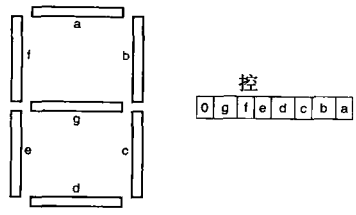


图 8-5 7 段显示器

例 8-25

```
unsigned char LookUp(unsigned char temp)
{
        unsigned char temp1[] = {0x3f, 6, 0x5b, 0x4f, 0x66,
                                 0x6d, 0x7d, 7, 0x7f, 0x6f};
        _asm
        {
                lea ebx,temp1
                mov al,temp
                xlat
                mov temp,al
        }
        return temp;
}
```

用以实现转换的 LookUp 函数仅包括几条指令，而且假定 temp 参数包括 BCD 码（0～9）以转换为 7 段码并以无符号字符的形式返回。第一条指令将查找表的地址装入到 EBX 中来寻址查找表，其他指令实现转换并以一个无符号字符的形式返回 7 段码。这里 temp1 数组被 BCD 索引传递到函数的 temp 中。

使用查找表访问 ASCII 码数据

有些程序需要将数字转换成 ASCII 码字符串。例如，某日历程序要显示星期几，由于星期日至星期六（Sunday～Saturday）包含 ASCII 码字符的数目不同，因此必须使用查找表实现数字与星期几的转换。

例 8-26 中的程序给出了一个查找表，该表引用了代码段中的 ASCII 码字符串，每个字符串是 ASCII 码表示的星期几。这个表包括了一个星期中每一天的引用。这个访问星期几的函数使用参数 day，用 0～

6之间的数据代表星期天到星期六，如果调用这个函数时，day 这个参数值为 2，那么 "Tuesday" 就会显示到屏幕上。注意这个函数并没有使用汇编代码，因为我们只是用一个表示星期几的索引访问这个数组中的元素。这个例子显示了数组的一个附加用途，它们可能还会用于嵌入式微处理器的应用程序中。

例 8-26

```
private: System::String^ GetDay(unsigned char Day)
{
        array<String^>^ temp =
        {
                "Sunday",
                "Monday",
                "Tuesday",
                "Wednesday",
                "Thursday",
                "Friday",
                "Saturday"
        };
        return temp[day];
}
```

8.3.5 使用查找表的示例程序

图 8-6 显示了一个基于对话框的名为 Display 的应用程序，每当按下键盘上的一个数字时，就会显示 7 段码样式的字符。如前面的例子提到的那样，在一个 Visual C++ 程序中，用 KeyDown 和 KeyPress 函数可以截获键盘消息，这样程序正好可以从键盘获得这个键值。接下来，键入码被过滤，只接受从 0～9 的键，然后使用一个查找表来访问 7 段码以便于显示。

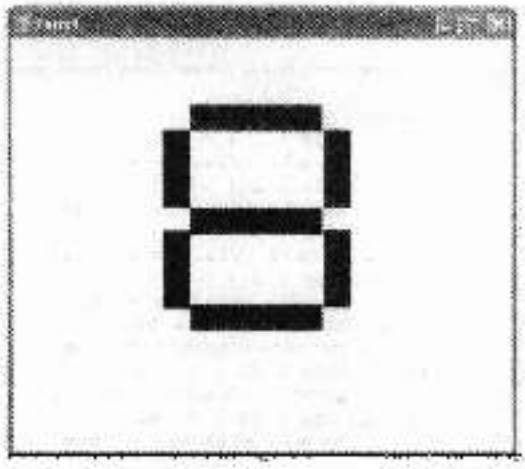


图 8-6 一个显示 7 段码形式数字的例子

显示的数字是用面板控件对象绘制的。水平条尺寸为 120×25，垂直条的尺寸为 25×75。对象的尺寸是显示在 Visual Studio 的资源屏幕的最右下角。要保证所添加的面板的顺序与显示的顺序一致，也就是说，在图 8-5 所示的 7 段显示器中，先添加标号 a 段，接着 b 段，等等。用 panel1 一 panel7 作为这个应用中的面板的变量名，记得将面板的背景色选为黑色。

给程序添加例 8-27 中列出的清除显示的名为 Clear 的函数。这个函数用来在程序首次执行或新的数字显示时清除屏幕上的数字。注意：面板的 Visible（可见）属性用于隐藏数字。另一种方法是改变面板的颜色。

例 8-27

```
private: System::Void Clear()
{
        panel1->Visible = false;
        panel2->Visible = false;
        panel3->Visible = false;
        panel4->Visible = false;
        panel5->Visible = false;
        panel6->Visible = false;
        panel7->Visible = false;
}
```

一旦键被按下，KeyDown 函数（参见例 8-28）就会过滤按键信息，并用查找表把它转换为 7 段码的形式。转换为 7 段码后，就会调用 ShowDigit 函数，并在屏幕上显示这个数字。ShowDigit 函数测试 7 段码的每一位，改变每一个面板的可见性来显示一个数字。这个程序的操作函数中没有用任何汇编语

言代码。

例 8-28

```
private: System::Void Clear()
{
       panel1->Visible = false;
       panel2->Visible = false;
       panel3->Visible = false;
       panel4->Visible = false;
       panel5->Visible = false;
       panel6->Visible = false;
       panel7->Visible = false;
}

private: System::Void Form1_KeyDown(System::Object^ sender,
       System::Windows::Forms::KeyEventArgs^ e)
{
       char lookup[] = {0x3f, 6, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 7, 0x7f, 0x6f};
       if (e->KeyCode >= Keys::D0 && e->KeyCode <= Keys::D9)
       {
              ShowDigit(lookup[e->KeyValue - 0x30]);  //显示这个数字
       }
}

private: System::Void ShowDigit(unsigned char code)
{
       Clear();
       if (( code & 1 ) == 1)
              panel1->Visible = true;     //测试a段
       if (( code & 2 ) == 2)             //测试b段
              panel4->Visible = true;
       if (( code & 4 ) == 4)             //测试c段
              panel5->Visible = true;
       if (( code & 8 ) == 8)             //测试d段
              panel3->Visible = true;
       if (( code & 16 ) == 16)           //测试e段
              panel6->Visible = true;
       if (( code & 32 ) == 32)           //测试f段
              panel7->Visible = true;
       if (( code & 64 ) == 64)           //测试g段
              panel2->Visible = true;
}

private: System::Void Form1_Load(System::Object^ sender,
       System::EventArgs^ e)

{
       Clear();
}
```

8.4 磁盘文件

数据通常以文件的形式存储在磁盘上。磁盘本身由 4 个主要部分组成：引导扇区、**文件分配表**（**file allocation table**，**FAT**）、根目录和数据存储区。Windows 的 **NTFS**（**New Technology File System**，**新技术文件系统**）包括一个引导扇区和一个**主文件表**（**master file table MFT**）。磁盘的第一个扇区为引导扇区，它被用于当计算机加电时将磁盘操作系统（DOS）从磁盘装入内存。

FAT（或 MFT）是操作系统存储文件/子目录名及其在磁盘中位置的表。对任一磁盘文件的访问均通过 FAT（或 MFT）表来管理。所有其他子目录和文件则通过 FAT 系统中的**根目录**（**root directory**）来访问。NTFS 系统没有根目录，尽管文件系统可能仍会有根目录。磁盘文件均被认为是顺序存取文件，即当访问这些文件时，是从头到尾一次一个字节地访问。NTFS 文件系统和 FAT 文件系统都在使用中，许多现代的 Windows 系统的硬盘驱动器使用 NTFS，而软盘、CD－ROM 和 DVD 使用 FAT 系统。

8.4.1 磁盘的组织

图 8-7 说明了磁盘表面上扇区和磁道的组织形式。这种组织形式适用于软盘和硬盘存储系统。对于软磁盘，最外面的磁道总是 0 磁道，最里面的磁道为 39（双密度）或 79（高密度）磁道。而硬磁盘的最里面磁道由磁盘容量决定，它可以为 10 000，对于超大容量的硬盘还可更高。

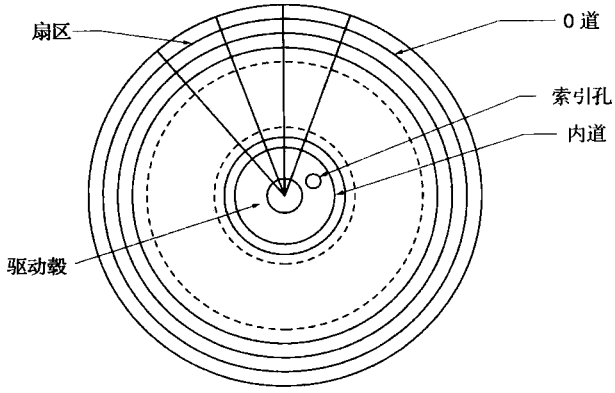


图 8-7 磁盘的结构

图 8-8 给出了磁盘上数据的组织形式。其中 FAT 表的长度由磁盘容量决定。在 NTFS 系统中，MFT 的长度由存储在磁盘上的文件数目决定。同样，根目录的长度由文件及其子目录的数目决定。引导扇区总是 512 字节长，它位于最外面磁道的 0 扇区，即第一个扇区。

引导扇区包含一个**引导装入程序**（**bootstrap loader**），当系统加电时，该程序被读入 RAM 中。然后引导装入程序开始执行，并将操作系统装入 RAM。接着，引导装入程序将控制权交给操作系统控制程序，使计算机处于 Windows 命令处理程序的控制之下。如果磁盘上有 Linux 操作系统，也是执行同样的事件序列。

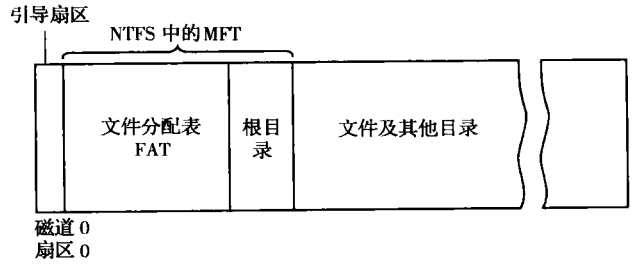


图 8-8 磁盘上的主要数据存储区

FAT 表指明了哪些扇区是空的，哪些是损坏的（不可用的）以及哪些扇区含有数据。操作系统每次向磁盘上写数据时，都要访问 FAT 表，以便找到空扇区。在 FAT 表中，每个空簇用 0000H 表示，每个已被占用的扇区用簇号表示。一个**簇**（**cluster**）可由一个扇区到任意多个扇区组成。许多硬盘存储系统中，每簇有 4 个扇区，这意味着最小的文件为 512 ×4，即 2048 字节长。在使用 NTFS 的系统中，簇的大小一般为 4K 字节（8 个扇区）。

图 8-9 给出了根目录下每一目录项的格式，或任何其他目录或子目录下每一目录项的格式。每一目录项均包含名字、扩展名、属性、时间、日期、存储位置和长度。文件的长度存储为一个 32 位数，这意味着一个文件的最大长度为 4GB。存储位置为起始簇号。

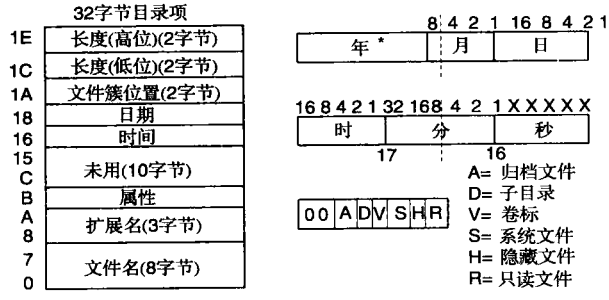


图 8-9 FAT 目录或子目录项的格式

* 注：年 8 = 1988，年 9 = 1989，年 10 = 1990 等。

Windows 的 NTFS 文件系统用一个比 FAT 文件系统（32 字节）更大的目录项或记录（1024 字节）。MFT 记录包括文件名、文件日期、属性和数据。数据可以是文件的整个内容，或者是指向数据在磁盘上位置的指针，叫做 **file run**。通常小于 1500 字节的文件适合 MFT 记录，更长的文件适合一个或多个 file

run。file run 是用来存储文件数据的一系列相邻簇。图 8-10阐明了 Windows NTFS 系统中的一个 MFT 记录。信息属性包括创建日期、最后修改日期、创建时间、最后修改时间，以及一些如只读、存档之类的文件属性。安全属性存储了 Windows 系统中文件访问权限的所有安全信息。Header 存储了关于记录类型、大小、名字（可选），以及是否常驻等信息。

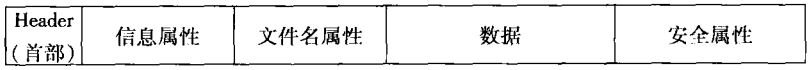


图 8-10 NTFS 系统中主文件表的一条记录

8.4.2 文件名

文件和程序均通过文件名和其后的扩展名在磁盘上存储和访问。在 DOS 操作系统下，文件名长度为1 ～8 个字符，这些字符为除空格和 " \ . / [ ] * , : < >¦ ;? = 之外的所有 ASCII 字符。除文件名之外，文件还可以有 1 ～3 个字符的扩展名。注意，文件名和扩展名之间总是用句点分隔。如果使用 Windows 95 ～Windows XP 操作系统，则文件名可以长达 255 个字符，甚至可以包含空格。这对于 DOS 系统 8 个字符文件名的限制是一大改进。

目录和子目录名

DOS 文件管理系统通过目录和子目录来组织磁盘上的数据和程序。在 Windows 中，目录和子目录被称为文件夹。应用于文件命名的规则也同样用于文件夹名字的命名。磁盘在首次格式化时，被构造成包含一个根目录。如硬盘驱动器 C 的根目录就是 C:\，任何其他目录均放置在根目录下。例如，C:\DATA 表示根目录下的目录 DATA。每个位于根目录下的目录还可包含子目录。例如子目录 C:\DATA\AREA1 和 C:\DATA\AREA2，这里，目录 DATA 包含两个子目录，即 AREA1 和 AREA2。子目录还可以包含其他子目录，如 C:\DATA\AREA2\LIST，它表示目录 DATA 下有子目录 AREA2，在子目录 AREA2 下还包含子目录 LIST。

8.4.3 顺序存取文件

所有 DOS 文件和 Windows 文件均为顺序存取文件。一个顺序文件是从头至尾地进行存储和访问，这就是说，要读文件的最后一个字节，则必须访问第一个字节及它与最后一个字节之间的所有字节。幸运的是，在 C++ 中，使用 File 类可以对文件进行读写，它简化了对文件的存取和操作。本节主要介绍如何完成一个顺序存取文件的创建、读、写、删除以及重命名。要获得对文件类的访问，在程序的开始处必须使用 using 语句添加新的导入配件元数据。如果需要文件访问，可以给程序中添加**using namespace System::IO;**语句。

创建文件

在使用一个文件之前，它必须已存在于磁盘中。创建文件可由 File 类用 Create 作为一个指示 File 去创建文件的属性来完成。如例 8-29 所示，通过调用 Create 来创建一个文件。这里，通过程序创建的文件的名称可以存储于一个名为 FileName 的 String 对象中。接下来，File 类用于测试和检查创建它之前是否已经存在。最后 if 条件语句创建并打开一个文件。

在本例中，如果由于磁盘空间满或者文件夹没有找到而导致失败，就会弹出一个消息框显示“不能打开［文件名］”消息，如果点击消息框中的 OK 按钮，则退出程序。为了试验这个例子，我们创建一个基于对话框的应用程序，并把代码放到 OnInitDialog 函数中的 TODO:语句后面。选择一个不存在的文件夹名并运行程序，这时就会看到错误消息。如果改变 FileName 使它不包括这个文件夹，则不会出现错误消息。

例 8-29

```
String^ fileName = "C:\\Test.txt";

if (File::Exists(fileName) == false)
{               // 不要忘记使用命名空间 System::IO;
        try
        {
```

```
            File::Create(fileName);
        }
        catch (...)
        {
            MessageBox::Show("Cannot create " + fileName);
            Application::Exit();
        }
}

//test.txt 存在于文件夹 test 中，长度为 0 字节
```

写文件

一旦文件存在，就可以对它写入数据了。实际上，如果创建一个文件而不向里面写入数据的话是很不正常的。一次可以向文件中写入一个字节的数据。用 File Stream 类可向文件中写入一个数据流。数据的写入是从文件中的第一个字节处开始的。例 8-30 列出了一个程序，它在根目录下创建了一个名为 Test1. txt 的文件，并在每 256 个字节中存储字母 A。如果运行这段代码，并用记事本打开的话，就会发现文件中有 256 个字母 A。注意，结束时应调用 Close（）函数关闭文件。在这个例子中还需要注意，数组的字节大小利用 C++ 中的垃圾回收类来创建。利用这个类来创建受管理的数据数组是非常重要的。

例 8-30

```
String^ fileName = "C:\\Test1.txt";
array<Byte>^ buffer = gcnew array<Byte>(256);

try
{
        FileStream^ fs = File::OpenWrite(fileName);
        for (int a = 0; a < 256; a++)
        {
        buffer[a] = 'A';
        }
        fs->Write(buffer, 0, buffer->Length);
        fs->Close();
}
catch (...)
{
        MessageBox::Show("Disk error");
        Application::Exit();
}
```

假设文件要写入一个 32 位的整数。由于只能写入字节，所以必须用一种方法把 4 个字节的整数转换为可以写入文件的形式。在 C++ 中，shifts 用于将字节正确存储在数组中。汇编语言也可以用更少的字节来实现同样的任务，如例 8-31 所示。如果比较每种方法的汇编代码，就会知道汇编语言代码更短而且速度更快。如果考虑的侧重点是速度和大小，那么就要首选汇编语言代码，在这种情况下生成的代码十分高效。

例 8-31

```
int number = 0x20000;
array<Byte>^ buf = gcnew array<Byte>(4);
//C++ 转换

buf[0] = number;
buf[1] = number >> 8;
buf[2] = number >> 16;
buf[3] = number >> 24;

//汇编语言转换
_asm
{
        mov     eax,number
        mov     buf[0],al
```

```
        mov     buf[1],ah
        bswap   eax             ;从小到大转换为从大到小
        mov     buf[2],ah
        mov     buf[3],al
}
```

读文件数据

调用 File 的 OpenRead 成员函数可以读取从文件的开头到结尾的数据。例 8-32 显示了一个例子，它读取例 8-30 中文件到称作 buffer1 的缓冲区里。OpenRead 函数返回实际从文件中读到的字节数，但是在这个例子中没有用到。如果在此知道文件的大小将会起到很好的作用，但是假设文件的长度不知道。FileInfo 类用于获得文件的长度就像例 8-33 一样。

例 8-32

```
String^ fileName = "C:\\Test1.txt";
array<Byte>^ buffer1 = gcnew array<Byte>(256);

try
{
        FileStream^ fs = File::OpenRead(fileName);
        fs->Read(buffer1, 0, 255);
        fs->Close();
}
catch (...)
{
        MessageBox::Show("Disk error");
        Application::Exit();
}
```

例 8-33

```
String^ fileName = "C:\\Test1.txt";
FileInfo^ fi = gcnew FileInfo(fileName);
int fileLength = fi->Length;
```

二进制转储程序的示例

Windows 不可用的一个工具就是以十六进制代码显示文件为容的程序。尽管大多数程序设计员都用不到这一点，但在开发软件的时候往往会用到，以便文件的实际内容可以以十六进制的格式显示。打开一个基于对话框的 Windows MFC 程序，并命名为 HexDump。在窗体上放置一个称为富文本框（Rich Textbox）的控件，如图 8-11 所示。在设置 Rich Textbox 控件属性时，确保把 Locked 属性改为 true 并把滚动条设置为垂直。如果显示一个非常大的文件，就可以往下滚动。非常大的文件需要花费一些时间加载到这个程序中。



图 8-11 HexDump 程序的屏幕截图

这个程序用 Disph 函数把地址显示为 8 位十六进制地址，并把该地址的内容显示为 2 位十六进制的数字。给程序添加 Disp 函数，使其返回一个串，该串位于第三个参数 char temp 所指向的空间。前两个参数包含了两个整数，一个是数字，一个是数字的位数，如例 8-34 所示。

例 8-34 显示了一个执行十六进制转储的程序。大部分程序都是由 Visual C++ 生成，只有顶层函数和底层的一些函数需要输入以创建应用程序。注意改变文件需要在程序中改变文件名称，可以用一个编辑框输入文件名来修改，但本例为了简明起见没有做这项工作。在这个程序中一次读取 16 字节的内容并加以格式化以便显示。这一过程一直持续到文件中没有数据并且 FileByteCount 返回零值。显示在十六进制列表末尾的 ASCII 数据被过滤以使低于 32（空格）的 ASCII 字符显示为句点。这

一点是很重要的，否则如换行、退格等控制字符可能会破坏 ASCII 文本的屏幕格式，而这是我们所不期望的。

例 8-34

```
#pragma once
namespace HexDump1 {

	using namespace System;
	using namespace System::ComponentModel;
	using namespace System::Collections;
	using namespace System::Windows::Forms;
	using namespace System::Data;
	using namespace System::Drawing;
	using namespace System::IO;

//此处的汇编代码使用/CLR开关编译
void Disph(unsigned int number, unsigned int size, char* temp)
{
	int a;
	number <= ( 8 - size ) * 4;		//调整位置
	for (a = 0; a < size; a++)
	{
		char temp1;
		_asm
		{
			rol number, 4;
			mov al,byte ptr number
			and al,0fh		;产生0~F
			add al,30h		;转换为ASCII码
			cmp al,39h
			jbe Disph1
			add al,7
Disph1:
			mov temp1,al
		}
		temp[a] = temp1;		//把数字加到数字串
	}
	temp[a] = 0;			//以空数字串结束
}

	/// <summary>
	/// Summary for Form1
	///
	/// WARNING: If you change the name of this class, you will need to
	change the
	///          'Resource File Name' property for the managed resource
	compiler tool
	///          associated with all .resx files this class depends on.
	Otherwise,
	///          the designers will not be able to interact properly with
	localized
	///          resources associated with this form.
	/// </summary>
	public ref class Form1 : public System::Windows::Forms::Form
	{
	public:
		Form1(void)
		{
			InitializeComponent();
			//
			//做法：在此添加构造代码
			//
		}
	protected:
		/// <summary>
		/// Clean up any resources being used.
```

```
        /// </summary>
        ~Form1()
        {
            if (components)
            {
                delete components;
            }
        }
    private: System::Windows::Forms::RichTextBox^ richTextBox1;
    private: System::Windows::Forms::OpenFileDialog^ openFileDialog1;
    private: System::Windows::Forms::Button^ button1;
    protected:

    private:
        /// <summary>
        /// Required designer variable.
        /// </summary>
        System::ComponentModel::Container ^components;
#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->richTextBox1 = (gcnew
System::Windows::Forms::RichTextBox());
            this->openFileDialog1 = (gcnew
System::Windows::Forms::OpenFileDialog());
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->SuspendLayout();
            //
            // richTextBox1
            //
            this->richTextBox1->Font = (gcnew
    System::Drawing::Font(L"Courier New", 9.75F,
    System::Drawing::FontStyle::Regular,
    System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->richTextBox1->Location = System::Drawing::Point(12,
    12);
            this->richTextBox1->Name = L"richTextBox1";
            this->richTextBox1->ScrollBars =
    System::Windows::Forms::RichTextBoxScrollBars::Vertical;
            this->richTextBox1->Size = System::Drawing::Size(657, 420);
            this->richTextBox1->TabIndex = 0;
            this->richTextBox1->Text = L"";
            //
            // openFileDialog1
            //
            this->openFileDialog1->FileName = L"openFileDialog1";
            //
            // button1
            //
            this->button1->Location = System::Drawing::Point(601, 438);
            this->button1->Name = L"button1";
            this->button1->Size = System::Drawing::Size(68, 25);
            this->button1->TabIndex = 1;
            this->button1->Text = L"Open";
            this->button1->UseVisualStyleBackColor = true;
            this->button1->Click += gcnew System::EventHandler(this,
    &Form1::button1_Click);
            //
            // Form1
            //
            this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
```

```
                this->AutoScaleMode =
        System::Windows::Forms::AutoScaleMode::Font;
                this->ClientSize = System::Drawing::Size(681, 468);
                this->Controls->Add(this->button1);
                this->Controls->Add(this->richTextBox1);
                this->Name = L"Form1";
                this->ShowIcon = false;
                this->StartPosition =
        System::Windows::Forms::FormStartPosition::CenterScreen;
                this->Text = L"HexDump";
                this->ResumeLayout(false);
        }
#pragma endregion

private: System::String^ Disp(int number, int size)
{
        char temp[9];
        Disph(number,size,temp);
        String^ a = "";
        int count = 0;
        while (temp[count] != 0)          // convert to string
        {
                Char b = temp[count++];
                a += b;
        }
        return a;
}

private: System::Void button1_Click(System::Object^ sender,
        System::EventArgs^ e)
{
        array<Byte>^ buffer = gcnew array<Byte>(1000000);
        int fileLength;
        String^ line = "";
        if (openFileDialog1->ShowDialog() ==
                System::Windows::Forms::DialogResult::OK)
        {
                try
                {
                        FileStream^ fs = File::OpenRead(openFileDialog1->FileName);
                        fs->Read(buffer, 0, 1000000);
                        fs->Close();
                        FileInfo^ fi = gcnew FileInfo(openFileDialog1->FileName);
                        fileLength = fi->Length;
                        this->Text = "HexDump -- " + openFileDialog1->FileName;
                }
                catch (...)
                {
                        MessageBox::Show("Disk error");
                        Application::Exit();
                }
                for (int a = 0; a < fileLength; a++)
                {
                        if (a % 16 == 0)
                        {
                                if (a != 0)
                                {
                                        richTextBox1->Text += " " + line;
                                        line = "";
                                        richTextBox1->Text += "\n";
                                }
                                richTextBox1->Text += Disp(a, 8);
                        }
                        richTextBox1->Text += " " + Disp(buffer[a], 2);
                        if (buffer[a] >= 32 && buffer[a] < 128)
                                line += Convert::ToChar(buffer[a]);
                        else
                                line += ".";
```

```
            }
            richTextBox1->Text += " " + line;
        }
        else
        {
            this->Text = "HexDump";
        }
}

};
}
```

文件指针和寻址

当打开、写或读一个文件时，文件指针寻址顺序存取文件的当前位置。当打开一个文件时，文件指针总是寻址文件的第一个字节。若一个文件为1024字节长，则读功能调用读取1023个字节，文件指针寻址文件的最后一个字节，而不是文件的末尾。

文件指针（file pointer）是一个32位数，它可以寻址文件中的任一字节。File的Append成员函数用于在文件结尾添加新的信息。文件指针可以从文件开头或者文件的结尾开始移动。Open函数将文件指针移到文件开头。在实际应用中，这两个成员函数用于访问文件的不同部分。FileStream的成员函数Seek可以使得文件指针移动到文件开头（SeekOrigin::Begin）、文件结尾（SeekOrigin::End）或文件中的当前位置（SeekOrigin::Current）。Seek函数的第一个参数是偏移量。如果要访问文件中的第三个字节，可以调用Seek（2，SeekOrigin::Begin）函数。（第三个字节的偏移量是2。）注意，Write函数中的第二个参数也是偏移量，与Seek的使用方式相同。

假设一文件已存在于磁盘上，要在该文件后附加256字节的新信息。当打开文件后，文件指针寻址此文件的第一个字节。这时若不先将文件指针移到文件末尾，而直接写入数据，则新数据将覆盖此文件的前256个字节的旧数据。例8-35给出了一个指令序列，它首先打开一个文件，然后将文件指针移到文件的末尾，并写入256个字节的数据，最后关闭文件。这样就为该文件添加了256个字节的新数据。

例8-35

```
String^ fileName = "C:\\Test1.txt";
array<Byte>^ buffer = gcnew array<Byte>(256);

try
{
    FileStream^ fs = File::OpenWrite(fileName);
    for (int a = 0; a < 256; a++)
    {
        buffer[a] = 'S';
    }
    fs->Seek(0, SeekOrigin::End);
    fs->Write(buffer, 0, buffer->Length);
    fs->Close();
}
catch (...)
{
    MessageBox::Show("Disk error");
    Application::Exit();
}

//或者在下面的写函数中使用偏移数执行相同操作

String^ fileName = "C:\\Test1.txt";
array<Byte>^ buffer = gcnew array<Byte>(256);
```

```
try
{
        FileStream^ fs = File::OpenWrite(fileName);
        for (int a = 0; a < 256; a++)
        {
                buffer[a] = 'S';
        }
        fs->Write(buffer, 256, buffer->Length);
        fs->Close();
}
catch (...)
{
        MessageBox::Show("Disk error");
        Application::Exit();
}
```

一种较难的文件操作是在文件中间插入新的数据。图 8-12 显示了如何通过创建第二个文件来实现这一操作。注意，首先将插入点之前的文件部分复制到新文件中，然后将新的信息插入到新文件中。最后再将文件剩余的部分复制到新文件中。一旦新文件完成，则旧文件被删除，并将新文件重命名为旧文件名。

例 8-36 给出了一个在旧文件中插入新数据的程序。该程序将 DATA. NEW 文件插入到 DATA. OLD 文件中，插入点为 DATA. OLD 文件的前 256 个字节之后。Buffer2 的新数据增加到文件中，接下来是旧文件的其余部分。新的 CFile 成员函数会被调用以删除旧的文件并将新文件重命名为旧文件名。

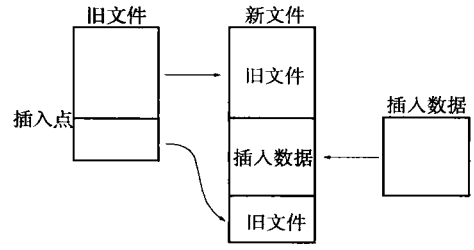


图 8-12 在一个旧文件中插入新数据

例 8-36

```
private: System::Void Form1_Load(System::Object^ sender,
        System::EventArgs^ e)
{
        String^ fileName1 = "C:\\Data.old";
        String^ fileName2 = "C:\\Data.new";
        int fileLength;
        array<Byte>^ buffer1 = gcnew array<Byte>(256);
        array<Byte>^ buffer2 = gcnew array<Byte>(6);

        try
        {
                FileStream^ fs1 = File::OpenWrite(fileName1);
                FileStream^ fs2 = File::OpenWrite(fileName2);
                FileInfo^ fi = gcnew FileInfo(fileName1);
                fileLength = fi->Length;
                fs1->Read(buffer1, 0, 256);
                fs2->Write(buffer1, 0, 256);
                fs2->Write(buffer2, 0, 6);
                fileLength -= 256;
        while (fileLength > 0)
        {
                fs1->Read(buffer1, 0, 256);
                fs2->Write(buffer1, 0, 256);
                fileLength -= 256;
        }
        fs1->Close();
        fs2->Close();
}
catch (...)
{
        MessageBox::Show("Disk error");
        Application::Exit();
}
}
```

8.4.4 随机存取文件

随机存取文件是用顺序存取文件通过软件方式实现的。一个随机存取的文件是通过一个记录号寻址，而不是通过文件搜寻数据的方式。创建随机存取文件时，Seek 函数是一个非常重要的函数。随机存取文件在有大量数据（通常称为数据库）时更易于使用。

创建随机存取文件

要创建一个随机存取文件系统，事先做好计划极为重要。假设需用随机存取文件存储客户姓名。每个客户记录需 32 个字节存放姓，32 个字节存放名，1 个字节存放中间名的第一个大写字母。每个客户记录还包含两行街道地址，每行 64 个字节；一行城市名，占 32 个字节；州代码 2 个字节；邮政编码 9 个字节。故客户基本信息需要 236 个字节，附加的信息使客户记录扩展为 512 个字节。由于业务发展的需要，现预留出 5000 个客户的记录空间，这意味着该随机存取文件总长度为 2 560 000 字节。

例 8-37 给出了一个短程序，它创建一个名为 CUST. FIL 的文件，并插入 5000 个空记录，每个记录占 512 个字节。一个空记录中每个字节均为 00H。这似乎是一个很大的文件，但它可装在最小的硬盘中。

例 8-37

```
private: System::Void Form1_Load(System::Object^ sender,
       System::EventArgs^ e)
{
       String^ fileName = "C:\\Cust.fil";
       array<Byte>^ buffer = gcnew array<Byte>(512);

       for ( int a = 0; a < 512; a++ )          //文件缓冲区
       {
             buffer[a] = 0;
       }

       try
       {
             FileStream^ fs = File::OpenWrite(fileName);
             for (int a = 0; a < 5000; a++)
             {
                   fs->Write(buffer, 0, 512);
             }
             fs->Close();
       }
       catch (...)
       {
             MessageBox::Show("Disk error");
             Application::Exit();
       }
}
```

读、写一条记录

一旦必须读一条记录时，则通过调用 Seek 函数找到记录号，例 8-38 列出了一个用于寻找一条记录的函数。此过程假定 CustomerFile 文件已经打开，而且 CUST. FIL 始终保持打开状态。

请注意，如何用记录号乘以 512 来得到移动指针功能调用的计数值。每次要将文件指针从文件开始移至要读的记录处。

例 8-38

```
void CCusDatabaseDlg::FindRecord(unsigned int RecordNumber)
{
       File.Seek( RecordNumber * 512, CFile::begin );
}
```

需要调用其他函数（见例 8-39）管理客户数据库，其中包括 WriteRecord、ReadRecord、FindLastNameRecord、FindBlankRecord 等。其中一些函数以及包括每一条记录信息的数据结构都列在例子中。

例 8-39

```
//将类置于form1类之前用于包含一个记录

public ref class Customer
{
        public: static array<Byte>^ FirstName = gcnew array<Byte>(32);
        public: static array<Byte>^ Mi = gcnew array<Byte>(1);
        public: static array<Byte>^ LastName = gcnew array<Byte>(32);
        public: static array<Byte>^ Street1 = gcnew array<Byte>(64);
        public: static array<Byte>^ Street2 = gcnew array<Byte>(64);
        public: static array<Byte>^ City = gcnew array<Byte>(32);
        public: static array<Byte>^ State = gcnew array<Byte>(2);
        public: static array<Byte>^ ZipCode = gcnew array<Byte>(9);
        public: static array<Byte>^ Other = gcnew array<Byte>(276);
};
//函数置于form1类的最后

static array<Byte>^ buffer = gcnew array<Byte>(512);
static String^ fileName = "C:\\Cust.fil";
static FileStream^ fs;
static Customer Record;


private: System::Void Form1_Load(System::Object^ sender,
        System::EventArgs^ e)
{       //当应用程序启动时打开文件
        Customer Record;
        try
        {
                fs = File::OpenWrite(fileName);
                for (int a = 0; a < 5000; a++)
                {
                     fs->Write(buffer, 0, 512);
                }
                fs->Close();
        }
        catch (...)
        {
                MessageBox::Show("Disk error");
                Application::Exit();
        }
}

private: System::Void FindRecord(unsigned int RecordNumber)
{
        fs->Seek(RecordNumber * 512, SeekOrigin::Begin);
}

private: System::Void WriteRecord(unsigned int RecordNumber)
{
        FindRecord(RecordNumber);
        fs->Write(Record.FirstName, 0, 32);
        fs->Write(Record.Mi, 0, 1);
        fs->Write(Record.LastName, 0, 32);
        fs->Write(Record.Street1, 0, 64);
        fs->Write(Record.Street2, 0, 64);
        fs->Write(Record.City, 0, 32);
        fs->Write(Record.State, 0, 2);
        fs->Write(Record.ZipCode, 0, 9);
}

private: System::Void ReadRecord(unsigned int RecordNumber)
{
        FindRecord(RecordNumber);
        fs->Read(Record.FirstName, 0, 32);
        fs->Read(Record.Mi, 0, 1);
        fs->Read(Record.LastName, 0, 32);
```

```
        fs->Read(Record.Street1, 0, 64);
        fs->Read(Record.Street2, 0, 64);
        fs->Read(Record.City, 0, 32);
        fs->Read(Record.State, 0, 2);
        fs->Read(Record.ZipCode, 0, 9);
}

private: System::UInt32 FindFirstName(array<Byte>^ FirstName)
{
        for ( int a = 0; a < 5000; a++ )
        {
                ReadRecord(a);
                if (Record.FirstName == FirstName)
                {
                        return a;       //如果找到则返回记录号
                }
        }
        return 5001;                    //如未找到则返回5001
}

private: System::UInt32 FindBlankRecord()
{
        for ( int a = 0; a < 5000; a++ )
        {
                ReadRecord(a);
                if (Record.LastName[0] == 0 )
                {
                        return a;
                }
        }
        return 0;
}
```

8.5 程序举例

前面已讨论了许多基本编程模块，这里介绍几个应用程序实例。尽管这些程序实例似乎都不足道，但它们却体现了某些新的编程技术和对微处理器的编程风格。

8.5.1 时间/日期显示程序

尽管这个程序没有用到汇编语言，但它演示了如何利用 Windows API 获取日期和时间，并格式化后显示。它也说明了如何在 Visual C++ 中使用定时器。例 8-40 列出了一个使用定时器的程序，每隔一秒就中断一次程序，以显示日期和时间。通过调用 DateTime 对象读取计算机的时间和日期，并把它存到名为 dt 的变量中。TimeDate 函数用于格式化 dt 变量。创建一个基于对话框的应用程序，命名为 DateTime，并放置两个标签，如图 8-13 所示。



图 8-13 时间和日期应用程序

例 8-40

```
private: System::Void Form1_Load(System::Object^ sender,
        System::EventArgs^ e)
{
        ShowDateTime();
}

private: System::Void ShowDateTime()
{
        DateTime dt = DateTime::Now;            //获得当前时间
        label1->Text = dt.Hour.ToString() + ":" + dt.Minute.ToString();
        label2->Text = dt.Date.ToLongDateString();
```

```
)
private: System::Void timer1_Tick(System::Object^ sender,
        System::EventArgs^ e)
{
        ShowDateTime();
}
```

8.5.2 数字排序程序

有时，必须对一组数据按大小排序，这常用冒泡排序法来完成。图 8-14 显示了用冒泡排序法对 5 个数进行排序的过程。注意，5 个数需通过 4 轮测试 4 次；在每一轮中，对相邻的 2 个数据进行比较，且根据比较的结果决定此两相邻数据是否要交换。还应注意在第 1 轮中共比较 4 次，在第 2 轮中共比较 3 次，依此类推。



图 8-14 用冒泡排序法对数据进行排序的过程

注：对 5 个数排序可能需要 4 轮比较。



图 8-15 冒泡排序应用程序

例 8-41 中的程序可从键盘接收 10 个数（32 位整数），当这些 32 位数被接收并存入内存区 Numbers 中之后，便使用冒泡排序技术对它们进行排序。此冒泡排序法使用了一个标志来确定在一轮中是否曾有 2 个数据交换，若没有数据交换，则数据已排好序，排序终止。这样及早的中止通常可以提高排序的效率，因为数字很少完全颠倒顺序。

一旦数据排好序，则将它们以升序的顺序显示在显示屏上。为了指定一个数组，到 Dlg 类的头部手动添加无符号整型数组 Numbers[10]。头部公共段的内容显示在例 8-41 中。图 8-15 显示了程序执行后的结果。

例 8-41

```
void Sort(int* data)
{
        char flag;
        __asm
        {
                mov   ecx,9             ;10个数需要9轮
L1:
                mov   flag,0            ;清交换标志
                mov   edx,0
L2:
                mov   ebx,data
                mov   eax,[ebx+edx*4]
                cmp   eax,[ebx+edx*4+4]
                jbe   L3
                push  eax               ;交换
                mov   eax,[ebx+edx*4+4]
                mov   [ebx+edx*4],eax
                pop   dword ptr [ebx+edx*4+4]
```

```
                mov   flag,1            ;标志置1
L3:
                inc   edx
                cmp   edx,ecx
                jne   L2
                cmp   flag,0
                jz    L4                ;如果没有交换
                loop  L1
L4:
        }
}
bool isHandled;

private: System::Void button1_Click(System::Object^ sender,
        System::EventArgs^ e)
{
        int numbers[10];
        int count = 0;
        int digit = 0;
        int a;
        for(a = 0; a < 10; a++)
        {
                numbers[a] = 0;
                while (digit < textBox1->Text->Length && textBox1->Text[digit]
                        != ',')
                {
                        numbers[a] = numbers[a] * 10 +
                                (int)(textBox1->Text[digit] - 0x30);
                        digit++;
                }
                digit++;
                if (digit >= textBox1->Text->Length)
                {
                        break;
                }
        }
        if (a == 9)
        {
                Sort(numbers);
                label2->Text = "";
                for (int a = 0; a < 9; a++)
                {
                        label2->Text += numbers[a].ToString() + ", ";
                }
                label2->Text += numbers[9].ToString();
        }
        else
        {
                MessageBox::Show(
                        "10 numbers must be entered separated by commas");
        }
}

private: System::Void textBox1_KeyDown(System::Object^ sender,
        System::Windows::Forms::KeyEventArgs^ e)
{
        isHandled = true;
        if (e->KeyCode >= Keys::D0 && e->KeyCode <= Keys::D9 ||
                e->KeyCode == Keys::Oemcomma || e->KeyCode == Keys::Back)
        {
                isHandled = false;
        }
}

private: System::Void textBox1_KeyPress(System::Object^ sender,
        System::Windows::Forms::KeyPressEventArgs^ e)
```

```
{
        e->Handled = isHandled;
}
```

8.5.3 数据加密

由于许多系统安全方面的因素，现在数据加密似乎已经是一种时尚。为了说明一个字符串的简单数据加密，假定字符串中的每一个字符与一个数字异或，我们把这个数字称为加密密钥。这肯定会改变字符的编码，但为了使其更具有随机性，假定加密密钥在每个字符加密后都会改变。这样就会更加难以探测被加密的信息，使其更难被破译。

为了说明这个简单的设计，图 8-16 列出了一个程序，它用一个文本框接受一个字符串，并用一个标签显示加密后的信息。这个例子用 0x45 这个初始加密密钥产生。如果初始值变化，那么这个加密后的信息就会改变。

例 8-42 列出了这个程序，它在文本框中产生加密后的信息。按钮事件处理函数读取了用于输入被加密字符串的文本框控件的内容，用一个对的汇编语言函数加密该串。注意程序怎样使用汇编语言来用加密密钥异或字符串的每一个字符，然后加密密钥如何为下一个字符作出改变。这里所用的技术增加了密钥，改变密钥大于 7FH。它可以一直更改以使其更难破译。比如，假定密钥在每一个其他字符上都加 1，并轮流使密钥反相，如例 8-43 所示。几乎可以使用操作的任意组合来修改密钥，使之难以解码。实际上，我们使用 128 位密钥，修改密钥的技术是较难的，虽然如此，但这里基本说明了加密是如何完成的。因为例 8-42 使用 8 位密钥，加密后的信息可以通过尝试所有 256（2^8）个可能的密钥来破解，但如果使用 128 位密钥，就需要尝试更多（2^{128}）的密钥来破解——一个几乎不可能的数字！



图 8-16 数据加密应用程序

例 8-42

```
char EncryptionKey = 0x45;
char Encrypt(char code)
{
        _asm
        {
                mov  al,code
                xor  al,EncryptionKey
                mov  code,al
                mov  al,EncryptionKey
                inc  al
                and  al,7fh
                mov  EncryptionKey,al
        }
        return code;
}

private: System::Void button1_Click(System::Object^ sender,
        System::EventArgs^ e)
{
        richTextBox1->Text = "";
        for (int a = 0; a < textBox1->Text->Length; a++)
        {
        richTextBox1->Text += Convert::ToChar(Encrypt(textBox1->Text[a]));
        }
}
```

例 8-43

```
//仅仅是程序的汇编语言部分
char EncryptionKey = 0x45;
char everyOther = 0;

char Encrypt(char code)
{
        _asm
        {
                mov  al,code
                xor  al,EncryptionKey
                mov  code,al
                mov  al,everyOther
                inc  al
                and  al,1
                mov  everyOther,al
                mov  bl,EncryptionKey
                cmp  al,0
                jz   L1
                inc  bl
                jmp  L2
L1:
                not  bl
L2:
                and  bl,7fh
                mov  EncryptionKey,bl
        }
        return code;
}
```

8.6 小结

1）汇编程序（ML. EXE）用于对程序模块进行汇编，这些模块包含 PUBLIC 变量和加有 EXTRN（外部）变量的段。连接程序(LINK. EXE)用于连接程序模块和库文件，以生成能在 DOS 命令行下执行的可执行程序。可执行程序的扩展名通常是 EXE，但也可能是 COM。

2）MACRO 和 ENDM 伪指令产生一个用于程序中的新的操作码。宏除了没有调用和返回外，与过程相似。代替调用和返回的是，每当宏被调用时，汇编程序将宏序列的代码插入到程序中。宏可以包含给宏序列传递信息和数据的变量。

3）大多数对象可以通过 SetFocus 成员函数设置对象的焦点。

4）C++ 中 Convert 类在很多情况下将一种类型转换到另一种类型，但是并不是在所有情况中都适用。

5）给不同的 Windows 事件如 Mouse Move、MouseDown 等可以从 Windows 访问鼠标驱动器。

6）二进制到 BCD 的数据转换，对于小于 100 的数可用 AAM 指令来实现，对于大于 100 的数可用不断除以 10 来实现。一旦转换成 BCD 数，则可在每一位上加 30H 将其转换成 ASCII 码，以便在 string 中放置。

7）当从 ASCII 数转换成 BCD 数时，可从每一位上减去 30H。为获得等值的二进制数，可不断乘以 10。

8）查找表可用于代码转换。若代码为 8 位，则可用 XLAT 指令完成转换；若代码大于 8 位，则可用一短过程访问查找表来完成转换。查找表也可用于存放地址，从而可以选择程序的不同部分或不同过程。

9）条件汇编语句允许当条件满足时，对部分程序进行汇编。这对于将软件裁剪成为应用程序很有用。在 Visual C++ Express 中，程序中包含汇编代码时必须使用/CLR 开关进行编译。

10）磁盘存储系统由磁道组成，磁道上包含存于扇区上的信息。许多磁盘系统每扇区存储 512 字节的信息。磁盘上的数据按引导扇区、文件分配表、根目录及数据存储区来组织。引导扇区将 DOS 系统从磁盘装入计算机的内存系统；FAT 或 MFT 表指明存在哪些扇区以及它们是否有数据；根目录中含有文件名和子目录，通过它可访问所有磁盘文件；数据存储区中含有所有子目录和数据文件。

11）可使用 Visual C++ 的 CFile 对象对文件进行操作。当读一个磁盘文件时，首先必须打开文件，读文件，然后关闭文件；当写一个磁盘文件时，也必须先打开文件，写文件，然后关闭文件。当打开文件时，文件指针寻址该文件的第一个字节。要访问其他地址中的数据，需在读/写数据前移动文件指针。

12）顺序存取文件是一种从头到尾顺序存取的文件。随机存取文件是一种可在任意位置存取的文件。尽管所有磁盘文件均为顺序文件，但通过使用软件可将它们作为随机存取文件处理。

8.7 习题

1. 汇编程序将源文件转换为_____文件。
2. 当源文件 TEST. ASM 被 ML. EXE 处理后，将生成哪些文件?
3. 连接程序连接目的文件和_____文件，生成一个可执行文件。
4. 当 PUBLIC 伪指令用在程序模块中时，其含义是什么?
5. 当 EXTRN 伪指令用在程序模块中时，其含义是什么?
6. 什么伪指令将与外部标号同时出现?
7. 当连接程序连接库文件和其他目的文件时，库文件是如何工作的?
8. 哪些汇编语言伪指令用于描述一个宏序列?
9. 什么是宏序列?
10. 如何将参数传送给宏序列?
11. 设计一个名为 ADD32 的宏，将 DX - CX 中的 32 位数与 BX - AX 中的 32 位数相加。
12. 在宏序列中，如何使用 LOCAL 伪指令?
13. 设计一个名为 ADDLIST PARA1，PARA2 的宏，将 PARA1 中的数与 PARA2 中的数相加。这里每个参数均代表一个存储区。在调用此宏前，相加的字节数用寄存器 CX 表示。
14. 设计一个名为 ADDM LIST，LENGTH 的宏，求一组字节数的累加和。其中标号 LIST 为数据块的起始地址，LENGTH 为相加数据的个数。在宏序列的末尾必须返回 16 位的和，且存放于 AX 中。
15. INCLUDE 伪指令的作用是什么?
16. 修改例 8-12 中的函数，使其仅过滤掉来自大键盘而不是小键盘的 0 ~ 9 的数字而忽略所有其他字符。
17. 修改例 8-12 中的函数，使其产生一个随机的 8 位数字并放到字符型变量 Random 中（提示：为了实现这一功能，每当 KeyDown 函数被调用时就使 Random 加 1，而不管哪种类型的 Windows 消息被处理）。
18. 修改问题 17 中的程序，使其产生一个从 9 到 62 的随机数。
19. 修改例 8-15 的函数，使十六进制数字用小写字母 a 到 f 替代大写字母。
20. 修改例 8-16 使其能够往左或右移位/旋转，可以通过增加两个单选按钮选择方向。
21. 在 Visual C++ 编程环境中，用什么事件处理程序可以访问鼠标? 什么事件可以调用每一个处理程序?
22. 如何检测鼠标右键是否被按下?
23. 如何检测鼠标双击事件?
24. 编写程序，用以检测鼠标的右键和左键同时被按下。
25. 在一个程序中如何用 Visual C++ 选择一种颜色?
26. ForeColor 属性的目的是什么?
27. 将一个小于 100 的数从二进制转换为 BCD 数时，用_____指令可完成此转换。
28. 如何将一个大于 100（十进制）的数从二进制转换为 BCD 数?
29. 如何将一个二进制数显示为八进制数?
30. 通过加上_____可将一个 BCD 数转换为 ASCII 编码的数。
31. 通过减去_____可将一个 ASCII 编码的数转换为 BCD 数。
32. 设计一个函数，从键盘输入到文本框控件中的字符（使用 KeyDown）读取一个 ASCII 数，并以一个无符号整数返回。文本框中的数字是一个八进制数字，它通过该函数转换为二进制数。
33. 解释如何将一个 3 位 ASCII 码表示的数转换为二进制数。
34. 设计一函数，将所有的小写 ASCII 码字母转换为大写 ASCII 码字母，此过程不能改变除字母 a ~ z 外的任何其他字符。
35. 设计一个查找表程序，将十六进制数 00H ~ 0FH 转换为表示这些十六进制数的 ASCII 码字符。要求给出查找表及转换所需的程序。建议创建函数进行转换。
36. 解释 FAT 系统中引导扇区、FAT 表和根目录的作用。
37. 解释在 NTFS 文件系统中 MFT 的作用。
38. 磁盘表面被划分成许多磁道，磁道又被划分为_____。
39. 什么是引导装入程序? 它在什么地方?
40. 什么是簇?
41. NTFS 文件系统经常使用_____字节长度的簇?
42. 一个文件的最大长度是多少?
43. 当使用长文件名时，使用什么编码来存储文件名?
44. DOS 文件名最多有_____个字符?
45. 一般情况下扩展名有多少个字符?
46. 在长文件名中，可以有多少个字符?
47. 设计一个程序，打开一个名为 TEST. LST 的文件，从中读取 512 个字节到数据段存储区 ARRAY，然后关闭此文件。
48. 设计一个程序，将文件 TEST. LST 重命名为 TEST. LIS。
49. File 的 Move 成员函数有什么作用?
50. 什么是 ActiveX 控件?
51. 编写一个程序，读入一个 0 至 2G 之间的任意十进制数，然后以 32 位二进制的形式在视频显示器上显示。
52. 编写一个程序，在显示屏上显示 2 的 0 到 7 次幂（用十进制表示），2 的每次幂显示为：2^n = 值。
53. 使用计时器生成随机数的技术设计一个程序，显示1 ~ 47（或任意数）之间的随机数，用于抽奖算法。
54. 修改例 8-28 中程序，使其能以十六进制七段码的形式显示 A、b、C、d、E、F。
55. 修改例 8-42 中程序，使用你自己设计的算法来加密信息。
56. 为 CString 设计一个加密函数，实现第 55 题的加密。

第 9 章　8086/8088 硬件特性

引言

本章描述了 8086 和 8088 微处理器的引脚功能，并详细介绍了以下硬件知识：时钟产生、总线缓冲、总线信号锁存、时序、等待状态以及最小模式操作与最大模式操作等。首先介绍简单的微处理器，由于其简单的结构，可以作为对 Intel 微处理器系列的入门。

在将任何器件与微处理器连接起来之前，必须了解微处理器的引脚功能及时序特性。这些微处理器与最新的 Pentium 4 或 Core2 微处理器包含有相同的引脚。因此，要完全理解本书后面章节中所讨论的存储器及 I/O 接口，本章内容是必须掌握的。

目的

读者学习完本章后将能够：

1）描述 8086/8088 每一引脚的功能。

2）了解微处理器的直流特性并指出其对于通用系列逻辑器件的扇出能力。

3）利用时钟产生器芯片（8284A）为微处理器提供时钟。

4）将缓冲器和锁存器与总线相连。

5）解释时序图。

6）描述等待状态，并设计产生不同数目等待状态所需的电路。

7）比较最小模式操作与最大模式操作之间的差别。

9.1　引脚和引脚功能

本节说明微处理器每一引脚的功能，有些引脚在特定情况下有多种功能。另外，还讨论其直流特性，为了解后面章节中有关缓冲和锁存的内容打下基础。

9.1.1　引脚

图 9-1 描述了 8086/8088 微处理器的引脚。仔细比较后发现，这两种微处理器实质上没有太大的区别，二者均为 40 引脚**双列直插封装（dual in-line package，DIP）**。

正如第 1 章所述，8086 是具有 16 位数据总线的 16 位微处理器，而 8088 是具有 8 位数据总线的 16 位微处理器（如引脚图所示，8086 有引脚 $AD_0 \sim AD_{15}$，而 8088 有引脚 $AD_0 \sim AD_7$），因此数据总线宽度是二者的主要区别。8086 的这个特点使得它能够更高效地传送 16 位数据。

然而，在控制信号中还有一个微小的区别，即 8086 有一个 M/$\overline{IO}$引脚，而 8088 有一个 IO/$\overline{M}$ 引脚。硬件上的另一区别表现在二者的引脚 34 定义不同：8088 为 SS0 引脚，而 8086 为$\overline{BHE}$/S7 引脚。

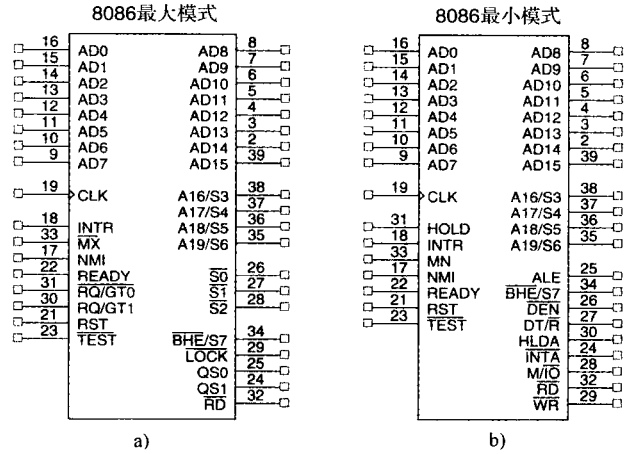


图 9-1　a）最大模式的 8086 微处理器的引脚图

b）最小模式的 8086 微处理器的引脚图

9.1.2 电源要求

8086 和 8088 微处理器都使用 +5.0V 电源电压，其允许偏差为 ±10%。8086 和 8088 需要的最大电源电流分别为 360mA 和 340mA。二者的工作环境温度为 32°F 到 180°F 之间。此温度范围不够宽，不允许微处理器工作在冬天或夏天的户外，但我们可得到具有扩展温度范围的 8086/8088 微处理器。另外还有一种 CMOS 型微处理器，它只需要很小的电源电流，并且温度范围很宽。80C88 和 80C86 均为 CMOS 型，它们只需 10mA 电源电流，而工作温度范围可达 -40°F 到 +225°F。

9.1.3 直流特性

如果不知道每一输入引脚的输入电流要求和每一输出引脚的输出电流驱动能力，是不可能将其他器件与微处理器的引脚相连的。具备这一知识使硬件设计者能够选择适当的接口器件与微处理器一起使用，而不必担心会损坏器件。

输入特性

这两种微处理器的输入特性与当今所有标准逻辑器件兼容。表 9-1 描述了其任一输入引脚的输入电压及输入电流要求。输入电流值非常小，这是因为输入均是场效应管门连接，所表现的仅仅是泄漏电流。

表 9-1 8086/8088 微处理器的输入特性

逻辑电平	电 压	电 流
0	最大 0.8V	最大 ±10μA
1	最小 2.0V	最大 ±10μA

表 9-2 8086/8088 微处理器的输出特性

逻辑电平	电 压	电 流
0	最大 0.45V	最大 2.0μA
1	最小 2.4V	最大 -400μA

输出特性

表 9-2 描述了这两种微处理器所有输出引脚的输出特性。8086/8088 的逻辑 1 电平与大多数标准逻辑器件系列兼容，但逻辑 0 电平则不然。标准逻辑电路的逻辑 0 电平最大为 0.4V，而 8086/8088 最大为 0.45V，二者相差 0.05V。

这一差别使得抗干扰能力从 400mV 的标准值降低到 350mV。**抗干扰能力**（**noise immunity**）是逻辑 0 输入电平和逻辑 0 输出电平之差。抗干扰能力的降低将导致连接长导线或过多负载的困难。因此建议，如果没有缓冲，则连接到输出引脚的任何类型的负载或负载组合不要超过 10 个。若超过此数，则噪声将引起时序问题。

表 9-3 列出了一些很通用的逻辑器件系列以及推荐的 8086/8088 扇出。与 8086/8088 输出引脚相连接的最佳器件类型是 LS、74ALS 或 74HC 逻辑器件。注意，尽管某些情况下扇出电流超过了 10 个负载所需的扇出电流，但还是建议：若所需的扇出超过 10 个负载时，信号必须经过缓冲驱动。

表 9-3 推荐的 8086/8088 引脚扇出

系 列	吸收电流	源电流	扇 出
TTL (74)	-1.6mA	40μA	1
TTL (74LS)	-0.4mA	20μA	5
TTL (74S)	-2.0mA	50μA	1
TTL (74ALS)	-0.1mA	20μA	10
TTL (74AS)	-0.5mA	25μA	10
TTL (74F)	-0.5mA	25μA	10
CMOS (74HC)	-10μA	10μA	10
CMOS (CD)	-10μA	10μA	10
NMOS	-10μA	10μA	10

9.1.4 引脚定义

$AD_7 \sim AD_0$ 8088 **地址/数据总线**，构成了 8088 的地址/数据多路复用总线。当 ALE 有效（逻辑 1）时，作为存储器的低 8 位地址或 I/O 端口地址；当 ALE 无效（逻辑 0）时，作为数据总线。在“保持响应”（HOLD）期间，这些引脚为高阻抗状态。

$A_{15} \sim A_8$ 8088 **地址总线**，在整个总线周期内提供存储器高 8 位地址。在“保持响应”期间，这

些引脚为高阻抗状态。

AD_{15}～AD_8 8086 **地址/数据总线**，构成了 8086 的高 8 位地址/数据多路复用总线。当 ALE 为逻辑 1 时，作为地址位 A_{15}～A_8；当 ALE 为逻辑 0 时，作为数据总线 D_{15}～D_8。在“保持响应”期间，这些引脚为高阻抗状态。

A_{19}/S_6～A_{16}/S_3 多路复用地址/状态总线，提供地址信号 A_{19}～A_{16}及状态位 S_6～S_3。在“保持响应”期间，这些引脚为高阻抗状态。

状态位 S_6 一直保持逻辑 0，S_5 表示中断允许标志位（IF）的状态，S_4 和 S_3 指示当前总线周期内被访问的段。表 9-4 为 S_4 和 S_3 的真值表。这两个状态位还可被译码为 A_{21} 和 A_{20}，用来寻址 4 个独立的 1MB 存储区。

表 9-4 状态位 S_4、S_3 的功能

S_4	S_3	功 能
0	0	附加段
0	1	堆栈段
1	0	代码段或不用
1	1	数据段

$\overline{RD}$ **读信号**。当它为逻辑 0 时，数据总线接收来自存储器或与系统相连的 I/O 设备的数据。在“保持响应”期间，该引脚为高阻抗状态。

READY **就绪输入信号**，用于在微处理器时序中插入等待状态。若该引脚被置为逻辑 0，则微处理器进入等待状态并保持空闲；若该引脚被置为逻辑 1，则它对微处理器的操作不产生影响。

INTR **中断请求信号**，用来申请一个硬件中断。当 IF = 1 时，若 INTR 保持高电平，则 8086/8088 在当前指令执行完毕后就进入中断响应周期（INTA 变为有效）。

$\overline{TEST}$ 这是一个**测试输入信号**，由 WAIT 指令来测试。若$\overline{TEST}$为逻辑 0，则 WAIT 指令的功能相当于 NOP 空操作指令；若$\overline{TEST}$为逻辑 1，则 WAIT 指令重复测试$\overline{TEST}$引脚，直到它变为逻辑 0。该引脚大多与 8087 算术协处理器相连。

NMI **非屏蔽中断输入信号**。与 INTR 信号类似，但 NMI 中断不必检查 IF 标志位是否为 1。若 NMI 被激活，则该中断输入使用中断向量 2。

RESET **复位输入信号**。若该引脚保持 4 个时钟周期以上的高电平，则导致微处理器复位。一旦 8086 或 8088 复位，则它从存储单元 FFFF0H 开始执行指令，并使 IF 标志位清零，禁止中断。

CLK **时钟引脚**，为微处理器提供基本的定时信号。时钟信号占空比必须为 33%（即时钟周期的 1/3 为高电平，而 2/3 为低电平），以便为 8086/8088 提供正确的内部定时基准。

VCC **电源输入**，为微处理器提供 +5.0V，±10% 电源输入。

GND **接地引脚**。注意，8086/8088 微处理器有两个引脚均标为 GND，为保证正常工作，二者必须都接地。

MN/$\overline{MX}$ **最小/最大模式引脚**，为微处理器选择最小模式或最大模式工作方式。若选择最小模式，则该引脚必须直接接 +5.0V。

$\overline{BHE}/S_7$ **高 8 位总线允许引脚**，用在 8086 中。在读操作或写操作期间允许高 8 位数据总线 D_{15}～D_8有效。状态位 S_7 始终为逻辑 1。

最小模式引脚：

当将 MN/$\overline{MX}$引脚直接连至 +5.0V 时，8086/8088 工作于最小模式。注意不要将该引脚通过一个上拉电阻与 +5.0V 相接，否则会导致工作异常。

IO/$\overline{M}$ 或 M/$\overline{IO}$ IO/$\overline{M}$（8088）或 M/$\overline{IO}$（8086）引脚选择存储器或 I/O 端口。该引脚指示，微处理器地址总线是存储器地址还是 I/O 端口地址。在“保持响应”期间，该引脚为高阻抗状态。

$\overline{WR}$ **写选通信号**，指示 8086/8088 正在输出数据给存储器或 I/O 设备。在$\overline{WR}$为逻辑 0 期间，数据总线包含给存储器或 I/O 设备的有效数据。在“保持响应”期间，该引脚为高阻抗状态。

$\overline{INTA}$ **中断响应信号**，响应 INTR 输入。该引脚常用来选通中断向量号以响应中断请求。

ALE **地址锁存允许**，表明 8086/8088 的地址/数据总线包含地址信息。该地址可以是存储器地址或 I/O 端口号。注意，在“保持响应”期间，ALE 不会被浮置。

DT/$\overline{R}$ **数据传送/接收信号**，表明微处理器数据总线正在传送（DT/$\overline{R}$ = 1）或接收(DT/$\overline{R}$ = 0)数据。该信号用来允许外部数据总线缓冲器。

DEN **数据总线允许**，用来激活外部数据总线缓冲器。

HOLD **保持输入信号**，用来请求直接存储器存取（DMA）。若 HOLD 信号为逻辑 1，微处理器停止执行软件，并将其地址、数据和控制总线置成高阻抗状态；若 HOLD 信号为逻辑 0，微处理器正常执行软件。

HLDA **保持响应信号**，指示 8086/8088 已进入保持状态。

$\overline{SS0}$ $\overline{SS0}$ 状态线相当于微处理器最大模式下的 S_0 引脚。该信号与 IO/$\overline{M}$ 及 DT/$\overline{R}$ 组合在一起，译码当前总线周期的不同功能（见表 9-5）。

表 9-5 8088 使用$\overline{SS0}$的总线周期状态

IO/$\overline{M}$	DT/$\overline{R}$	$\overline{SS0}$	功 能
0	0	0	中断响应
0	0	1	读存储器
0	1	0	写存储器
0	1	1	停止
1	0	0	取操作码
1	0	1	读 I/O
1	1	0	写 I/O
1	1	1	无效状态

表 9-6 总线控制器（8288）使用$\overline{S2}$、$\overline{S1}$和$\overline{S0}$产生的总线控制功能

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	功 能
0	0	0	中断响应
0	0	1	读 I/O
0	1	0	写 I/O
0	1	1	停止
1	0	0	取操作码
1	0	1	读存储器
1	1	0	写存储器
1	1	1	无效状态

最大模式引脚：

为使微处理器工作于最大模式，从而与外部协处理器一起工作，应将 MN/$\overline{MX}$引脚接地。

$\overline{S2}$、$\overline{S1}$和$\overline{S0}$ 这些**状态位**指示当前总线周期的功能。它们通常由 8288 总线控制器译码，后者将在本章后面部分介绍。表 9-6 给出了这 3 个状态位在最大模式下的功能。

$\overline{RQ}$/$\overline{GT1}$和$\overline{RQ}$/$\overline{GT0}$ **请求/同意引脚**，在最大模式下请求直接存储器存取（DMA）。这两个引脚都是双向的，既可用于请求 DMA 操作，又可用于同意 DMA 操作。

$\overline{LOCK}$ **锁定输出信号**，用来锁定外围设备对系统总线的控制权。该引脚通过在指令前加前缀 LOCK：激活。

QS_1 和 QS_0 **队列状态位**，表明内部指令队列的状态。这些引脚被算术协处理器（8087）访问，以监视微处理器内部指令队列的状态。见表 9-7 队列状态位的操作。

表 9-7 队列状态位

QS_1	QS_0	功 能
0	0	队列空闲
0	1	操作码的第一个字节
1	0	队列空
1	1	操作码的后续字节

9.2 时钟产生器8284A

本节描述了时钟产生器（8284A）和RESET信号，并介绍了8086/8088微处理器的READY信号（READY信号及其相关电路在9.5节详细讨论）。

9.2.1 8284A时钟产生器

8284A是8086/8088微处理器的一个辅助器件。如果没有时钟产生器，在基于8086/8088的系统中就需要许多附加电路来产生时钟（CLK）。8284A提供以下基本功能或信号：时钟产生、RESET同步、READY同步以及一个TTL电平的外围设备时钟信号。图9-2描述了8284A时钟产生器的引脚图。

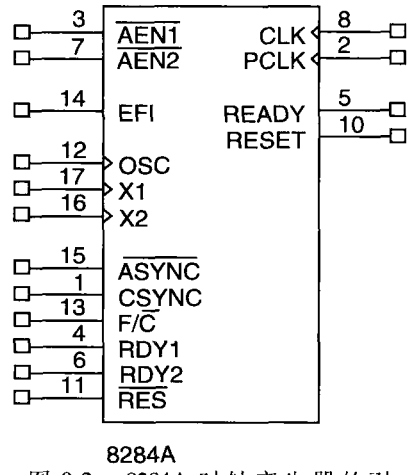


图9-2 8284A时钟产生器的引脚图

引脚功能

8284A是专为8086/8088微处理器设计的一个18脚集成电路，各引脚及其功能如下所示：

$\overline{AEN1}$和$\overline{AEN2}$ **地址允许引脚**，分别用来制约总线就绪信号RDY_1和RDY_2。在9.5节中描述了这两个引脚的用途，即与RDY_1和RDY_2输入一起产生等待状态。等待状态是由8086/8088微处理器的READY引脚产生的，而READY信号受$\overline{AEN1}$和$\overline{AEN2}$这两个输入信号控制。

RDY_1和RDY_2 **总线就绪输入信号**。在基于8086/8088的系统中，与$\overline{AEN1}$和$\overline{AEN2}$输入一起产生等待状态。

$\overline{ASYNC}$ **就绪同步**选择输入，为RDY_1和RDY_2输入选择一级同步方式或二级同步方式。

READY **就绪**是一个输出引脚，与8086/8088的READY输入引脚相连。此信号与RDY_1和RDY_2输入同步。

X_1和X_2 **晶体振荡器引脚**，与外部晶体相连，用作时钟产生器及其所有功能的定时源。

F/$\overline{C}$ **频率/晶体**选择输入，为8284A选择时钟源。若该引脚保持高电平，则一个外部时钟提供给EFI（外部频率输入）输入引脚；若该引脚保持低电平，则由内部晶振提供定时信号。当F/$\overline{C}$引脚为高电平时，使用外部频率输入。只要F/$\overline{C}$引脚为高电平，则由EFI提供定时信号。

CLK **时钟输出**引脚，为8086/8088微处理器及系统中其他器件提供时钟输入信号。CLK引脚的输出信号是晶体或EFI输入频率的1/3，其占空比为33%，这是8086/8088所要求的。

PCLK **外围设备时钟**信号，其频率为晶体或EFI输入频率的1/6，其占空比为50%。PCLK输出为系统中的外围设备提供时钟信号。

OSC **振荡器输出**，是一个TTL电平信号，其频率与晶体或EFI输入的频率相同。OSC输出在某些多处理器系统中为其他8284A时钟产生器提供EFI输入。

$\overline{RES}$ **复位输入**，对8284A是低电平输入有效。该引脚常与一个RC网络相连，以提供上电复位。

RESET **复位输出**，与8086/8088的RESET输入引脚相连。

CSYNC **时钟同步**引脚，在多处理器系统中当EFI输入提供同步信号时使用。如果已使用了内部晶振，则该引脚必须接地。

GND **接地引脚**。

VCC **电源输入**，为8284A提供+5.0V，±10%电源输入。

9.2.2 8284A 的操作

8284A 是比较容易理解的器件。图 9-3 示出了它的内部框图。

时钟部分的操作

逻辑框图的上半部分是时钟和同步复位部分。如图 9-3 所示，晶振有两个输入：X_1 和 X_2。若晶体接到 X_1 和 X_2 上，则振荡器产生一个与晶体同频率的方波信号，此信号送到一个与门，同时送到一个反相缓冲器，以提供 OSC 输出信号。OSC 信号有时用作系统中其他 8284A 电路的 EFI 输入。

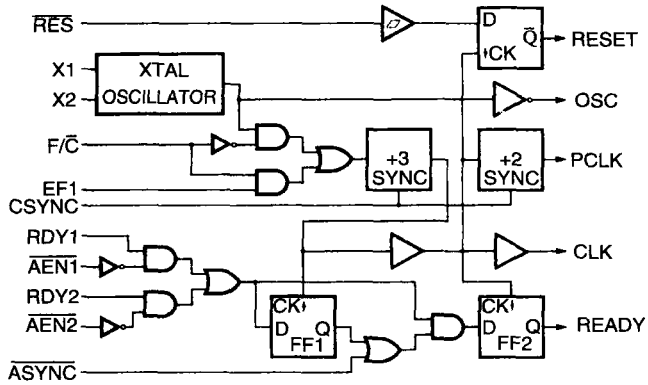


图 9-3 8284A 时钟产生器内部框图

从与门可以看出，当 F/$\overline{C}$ 为逻辑 0，振荡器输出送到一个 3 分频计数器；当 F/$\overline{C}$ 为逻辑 1，EFI 信号送到此计数器。

3 分频计数器的输出为就绪同步信号产生定时，同时产生另一个计数器（2 分频）的输入信号，还产生给 8086/8088 微处理器的 CLK 信号。CLK 信号在由时钟产生器输出前已经过缓冲。注意，第一个计数器的输出作为第二个计数器的输入，这两个级联计数器在 PCLK 端提供了一个 6 分频的输出，即外围设备时钟输出。

图 9-4 表明 8284A 是如何与 8086/8088 相连的。注意，F/$\overline{C}$ 和 CSYNC 均接地以选择晶振；15MHz 的晶体为 8086/8088 提供标准的 5MHz 的时钟信号及 2.5MHz 的外围设备时钟信号。

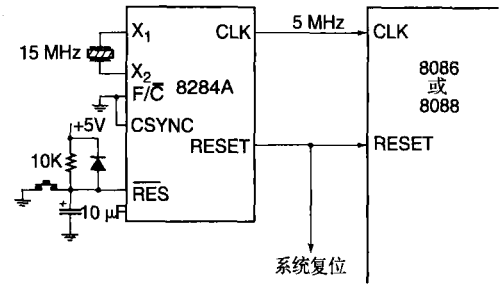


图 9-4 8284A 和 8086/8088 时钟及复位信号的连接，15MHz 的晶体为微处理器提供 5MHz 的时钟

复位部分的操作

8284A 的复位部分非常简单，它由一个施密特触发缓冲器和一个 D 触发器电路组成。D 触发器保证满足 8086/8088 RESET 输入的定时要求。此电路在每个时钟的下降沿（1 到 0 跳变）将 RESET 信号加到微处理器上，而 8086/8088 在时钟的上升沿（0 到 1 跳变）采样 RESET，因此，此电路满足 8086/8088 的定时要求。

参照图 9-4，当刚上电时，RC 电路为$\overline{\text{RES}}$输入引脚提供了一个逻辑 0 电平。经过一段短暂的时间，由于电容通过电阻充电趋于 +5.0V，故$\overline{\text{RES}}$输入变为逻辑 1。操作员可通过一个按钮开关对微处理器进行复位。正确的复位定时要求，在系统上电后不到 4 个时钟周期内，RESET 输入必须变为逻辑 1，并保持高电平至少 50μs 时间。触发器保证了 RESET 在 4 个时钟周期内变为高电平，RC 时间常数保证了它保持高电平至少 50μs 时间。

9.3 总线缓冲及锁存

在 8086/8088 微处理器能与存储器或 I/O 端口一起使用前，其多路复用总线必须分离。本节详细介绍如何对总线进行多路分离，以及在非常大的系统中如何对总线进行缓冲（由于最大扇出为 10，所以若系统超过 10 个器件，则必须经过缓冲）。

9.3.1 多路分离总线

为减少 8086/8088 微处理器集成电路的引脚数目，其上的地址/数据总线是多路复用（共享）的。遗憾的是，这加重了硬件设计者的负担，他们必须从这些多路复用的引脚提取或分离信息。

为什么不使总线一直多路复用呢？这是因为存储器和 I/O 要求在整个读周期或写周期期间地址保持有效和稳定。若总线是多路复用的，则存储器和 I/O 的地址改变，会使它们在错误的地址中读或写数据。

所有计算机系统有三种总线：1）地址总线，为存储器和 I/O 提供存储器地址或 I/O 端口号；2）数据总线，在系统中用于微处理器与存储器及 I/O 之间传输数据；3）控制总线，为存储器和I/O提供控制信号。系统必须有这三种总线，以便与存储器和 I/O 接口。

多路分离 8088

图 9-5 描述了 8088 微处理器和多路分离其总线所需的器件。在这种情况下，使用了两片 74LS373 透明锁存器来分离地址/数据总线 $AD_7 \sim AD_0$ 及地址/状态线 $A_{19}/S_6 \sim A_{16}/S_3$。

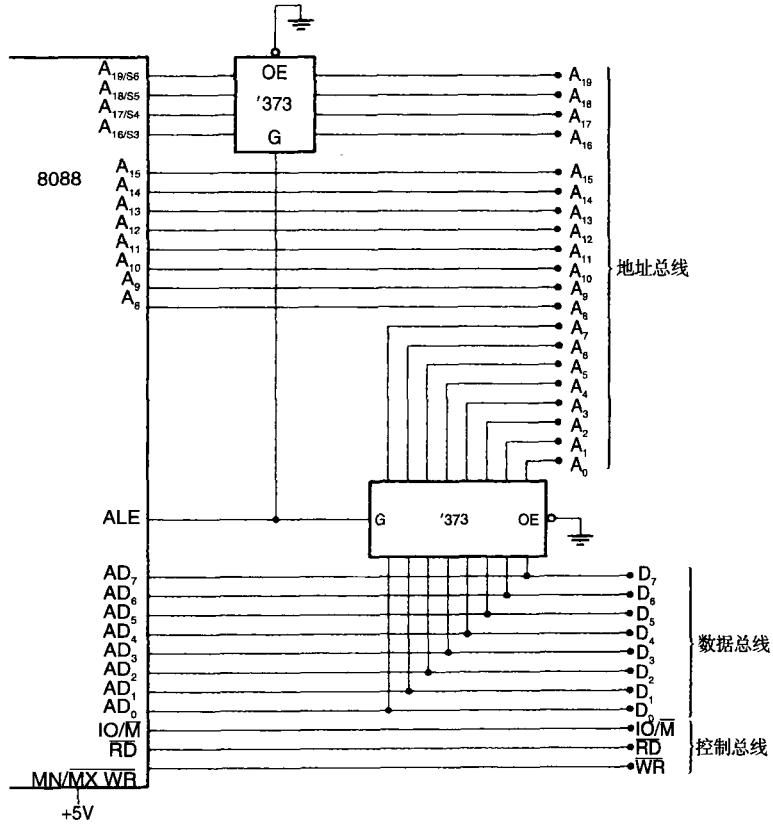


图 9-5 具有分离地址总线的 8088 微处理器，此模型用于许多基于 8088 的系统

在地址锁存允许引脚（ALE）变为逻辑 1 时，这些透明锁存器就像导线一样，将输入传送到输出。经过一段短暂的时间，ALE 回到逻辑 0 状态，使得锁存器记忆 ALE 变到逻辑 0 时的输入。在这种情况下，$A_7 \sim A_0$ 被存储在下面的锁存器中，$A_{19} \sim A_{16}$被存储在上面的锁存器中。这样就产生了一个独立的地址总线$A_{19} \sim A_0$。这些地址线允许 8088 寻址 1MB 的存储空间。由于数据总线是独立的，所以允许它被接到任何 8 位外围设备器件或存储器器件上。

多路分离 8086

正如 8088 一样，8086 系统也需要独立的地址、数据和控制总线。它们主要在多路复用的引脚数目上有所不同。在 8088 中，只有 $AD_7 \sim AD_0$、$A_{19}/S_6 \sim A_{16}/S_3$ 是多路复用的；而在 8086 中，多路复用的引脚包括 $AD_{15} \sim AD_0$、$A_{19}/S_6 \sim A_{16}/S_3$ 和 $\overline{BHE}/S_7$。所有这些信号都必须被分离。

图 9-6 描述了一个经过多路分离的 8086。包括三种总线：地址总线（$A_{19} \sim A_0$ 及 $\overline{BHE}$），数据总线（$D_{15} \sim D_0$）和控制总线（M/$\overline{IO}$、$\overline{RD}$及$\overline{WR}$）。

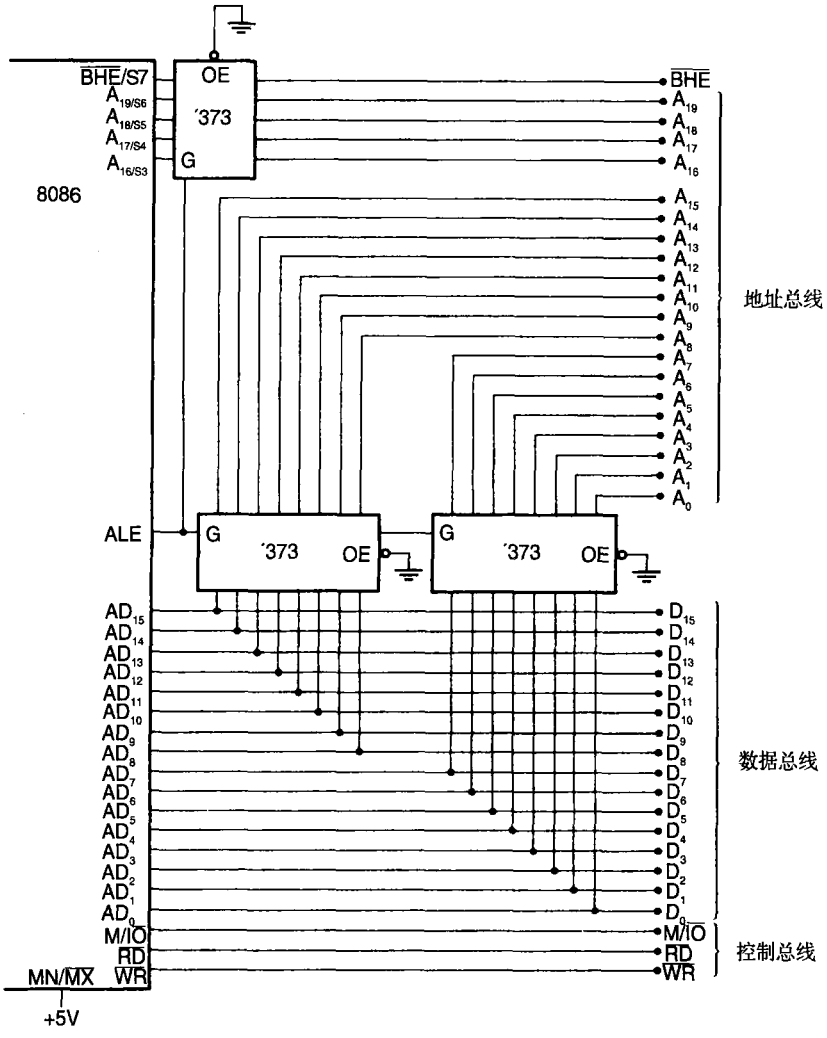


图 9-6　具有分离地址总线的 8086 微处理器，此模型用于许多基于 8086 的系统

图 9-6 所示的电路几乎与图 9-5 相同。不同的是，增加了一片 74LS373 来分离地址/数据总线 $AD_{15} \sim AD_8$，一个 $\overline{BHE}/S_7$ 输入加到上面的 74LS373，以选择 8086 的 16 位存储系统中的高位存储体。这里，存储器和 I/O 系统视 8086 为具有 20 位地址总线（$A_{19} \sim A_0$）、16 位数据总线（$D_{15} \sim D_0$）和 3 位控制总线（M/$\overline{IO}$、$\overline{RD}$及$\overline{WR}$）的器件。

9.3.2 缓冲系统

如果任一总线引脚上负载超过 10 个，则整个 8086 或 8088 系统必须经过缓冲。经过分离的引脚已由 74LS373 锁存器缓冲，这种锁存器用于驱动微处理器系统中高容量总线。缓冲器的输出电流增大后，可以驱动更多的 TTL 负载：逻辑 0 输出提供最大 32mA 吸收电流；逻辑 1 输出提供最大 5.2mA 源电流。

一个完全缓冲后的信号将给系统引入一个定时延迟。这将不会带来困难，除非使用了存储器或I/O设备，且工作在接近总线的最大速度下。9.4 节将讨论这个问题，并详细地讨论时间延迟。

完全缓冲的 8088

图 9-7 描述了一个经过完全缓冲的 8088 微处理器。注意，余下的 8 个地址引脚 A_{15}～A_8，使用的是 74LS244 八缓冲器；8 个数据总线引脚 D_7～D_0，使用的是 74LS245 八双向总线缓冲器；控制总线信号 IO/$\overline{M}$、$\overline{RD}$和$\overline{WR}$，使用的是 74LS244 缓冲器。一个经过完全缓冲的 8088 系统，需要两片 74LS244、一片 74LS245 和两片 74LS373。74LS245 的方向由 DT/$\overline{R}$ 信号控制，由$\overline{DEN}$信号允许和禁止。

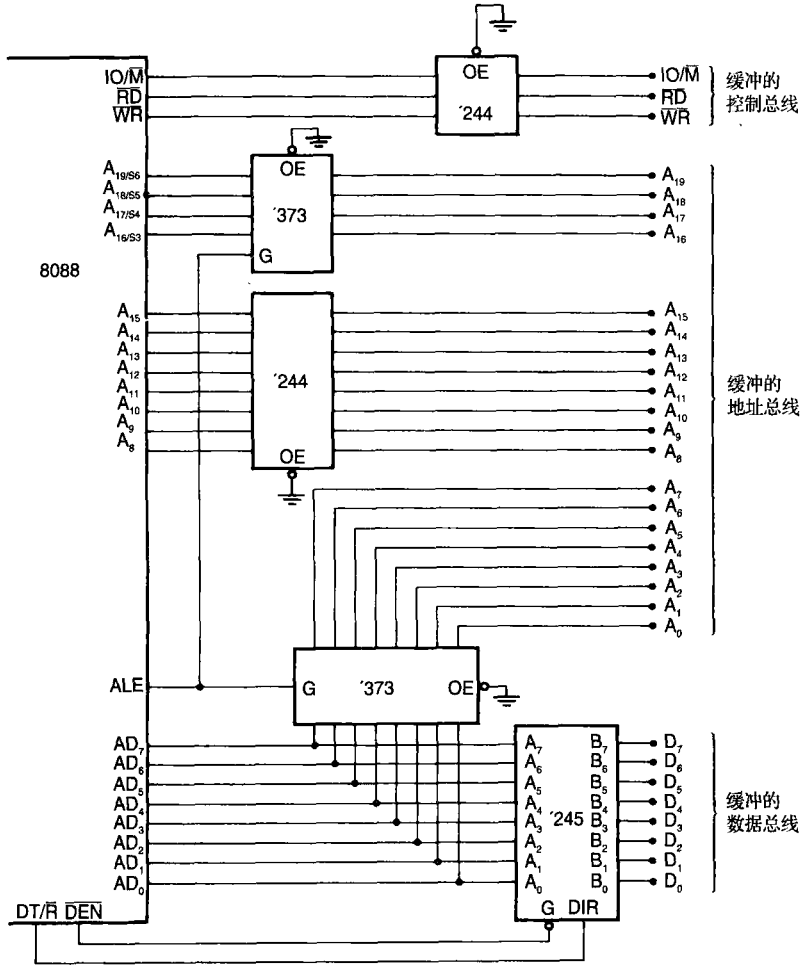


图 9-7 经过完全缓冲的 8088 微处理器

完全缓冲的 8086

图 9-8 描述了一个经过完全缓冲的 8086 微处理器。其地址引脚经过 74LS373 地址锁存器缓冲；其数据总线使用两片 74LS245 八双向总线缓冲器；控制总线信号 M/$\overline{IO}$、$\overline{RD}$和$\overline{WR}$，使用一片 74LS244 缓

冲器。一个经过完全缓冲的 8086 系统，需要一片 74LS244、两片 74LS245 和三片 74LS373。8086 比 8088 多需要一个缓冲器，这是因为 8086 有另外的 8 位数据总线 D_{15}～D_8。8086 还有一个经过缓冲的 $\overline{BHE}$信号，用于选择存储体。

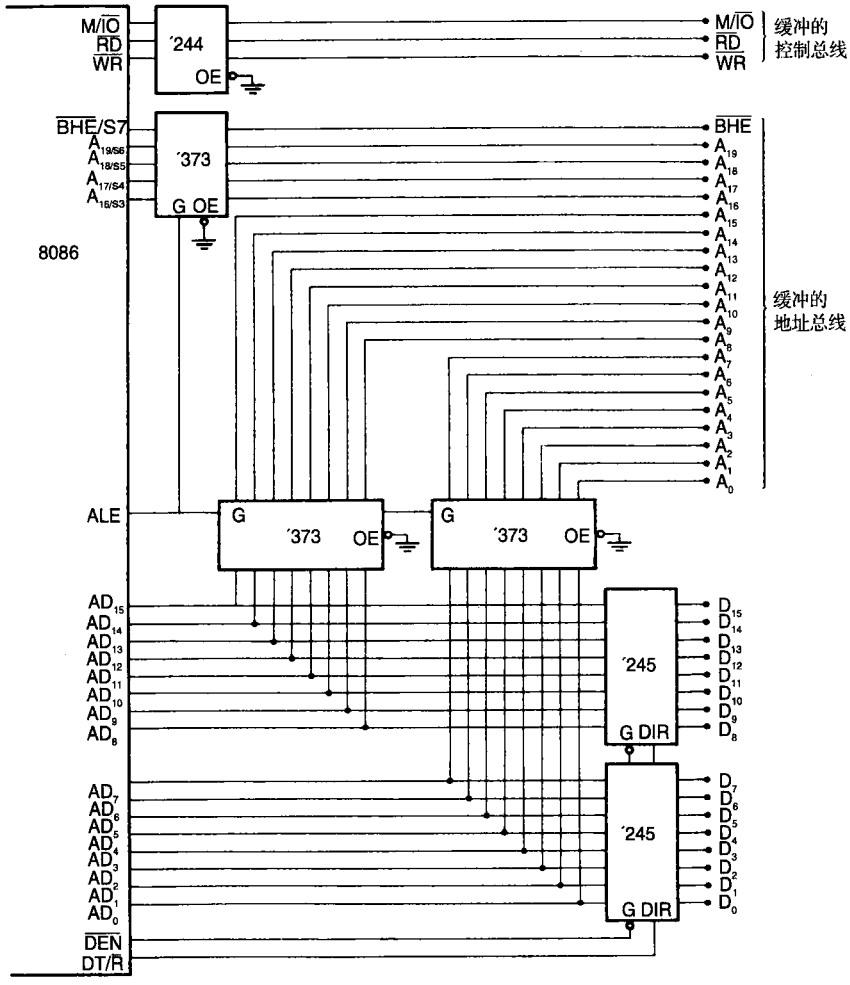


图 9-8 一个经过完全缓冲的 8086 微处理器

9.4 总线时序

在选择一个存储器或 I/O 设备与 8086/8088 微处理器接口之前，必须了解系统总线时序。本节分析总线信号的操作，以及 8086/8088 基本的读写时序。注意，本节仅仅讨论那些影响存储器和 I/O 接口的时序。

9.4.1 基本的总线操作

8086 和 8088 的三种总线——地址、数据和控制总线，与任何其他微处理器的工作方式相同。若数据要写入存储器（参见图 9-9 简化的写时序），则微处理器输出存储器地址到地址总线上，将要写入存储器的数据输出到数据总线上，并发出一个写命令（$\overline{WR}$）给存储器。对 8088 来说，$IO/\overline{M}=0$；对 8086 来说，$M/\overline{IO}=1$。若数据从存储器读出（见图 9-10 简化的读时序），则微处理器输出存储器地址

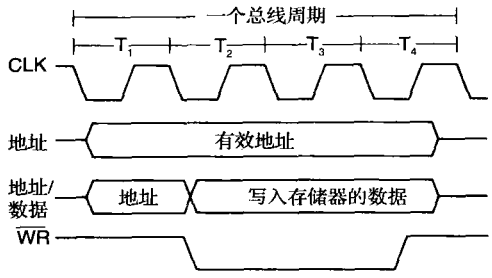


图 9-9 简化的 8086/8088 写总线周期

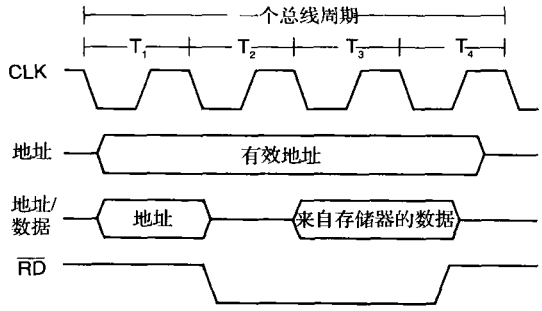


图 9-10 简化的 8086/8088 读总线周期

到地址总线上，发出一个读存储器信号（$\overline{RD}$），并通过数据总线接收数据。

9.4.2 一般的时序

8086/8088 按周期访问存储器和 I/O 端口，被称为**总线周期**（**bus cycle**）。每一总线周期等于 4 个系统时钟周期（T 状态）。某些新的微处理器把总线周期分为 2 个时钟周期。若时钟以 5MHz（这两种微处理器的基本工作频率）的频率工作，则完成一个 8086/8088 总线周期需要 800ns。这意味着微处理器在它自己和存储器或 I/O 之间，以最大每秒 1.25 百万次的速率读或写数据（由于 8086/8088 的内部指令队列，在突发状态下它每秒可执行 2.5 百万条指令［MIPS］）。其他一些微处理器由于其更高的时钟频率，能以更高的传输速率工作。

在总线周期的第一个时钟周期（即 T_1）内，发生了许多操作。存储器或 I/O 端口的地址通过地址总线和地址/数据总线被送出（地址/数据总线是多路复用的，有时是存储器地址信息，有时是数据）。在 T_1 期间，还输出控制信号 ALE、DT/$\overline{R}$ 和 IO/$\overline{M}$（8088）或 M/$\overline{IO}$（8086）。IO/$\overline{M}$ 或 M/$\overline{IO}$信号指示地址总线包含的是存储器地址，还是 I/O 设备（端口）号。

在 T_2 期间，8086/8088 微处理器发送$\overline{RD}$或$\overline{WR}$信号及$\overline{DEN}$信号，在写操作的情况下，要写入的数据出现在数据总线上。这些事件使得存储器或 I/O 设备开始执行一个读操作或写操作。若系统中存在数据总线缓冲器，则$\overline{DEN}$信号选通数据总线缓冲器，这样存储器或 I/O 就可接收要写入的数据，或是微处理器可在读操作时接收从存储器或 I/O 读入的数据。若正好是一个写总线周期，则数据通过数据总线被发送到存储器或 I/O。

在 T_2 结束时采样 READY 信号，如图 9-11 所示。若 READY 此时是低电平，则 T_3 之后将会出现一个等待状态（T_W），更多的细节见第 9.5 节。这一时钟周期允许存储器有时间存取数据。若总线周期正好是一个读总线周期，则在 T_3 结束时采样数据总线。

在 T_4 期间，所有总线信号变为无效，为下一个总线周期做准备。这个时间也是 8086/8088 采样数据总线读取存储器或 I/O 中数据的时间。另外，此时$\overline{WR}$信号的后沿传送数据给存储器或 I/O，当$\overline{WR}$信号回到逻辑 1 电平时，存储器或 I/O 被激活，写入数据。

9.4.3 读时序

图 9-11 描述了 8088 微处理器的读时序。8086 的读时序与之相同，除了 8086 是 16 位数据总线而不是 8 位以外。读者应该仔细观察此时序图，了解每一个时钟周期的主要事件。

在读时序图中最重要的一项是，允许存储器或 I/O 读取数据的时间长短。应根据存取时间来选择存储器。这是一个固定的时间值，在此时间内，微处理器在读操作中允许存储器存取数据。因此，所选的存储器应遵守系统的限制，这是极为重要的。

微处理器时序图没有直接提供存储器存取时间，因此有必要通过几个时间的组合来算出存取时间。为从图中求出存储器的存取时间，首先定位到 T_3 开始数据采样的点。仔细观察时序图，将注意到有一

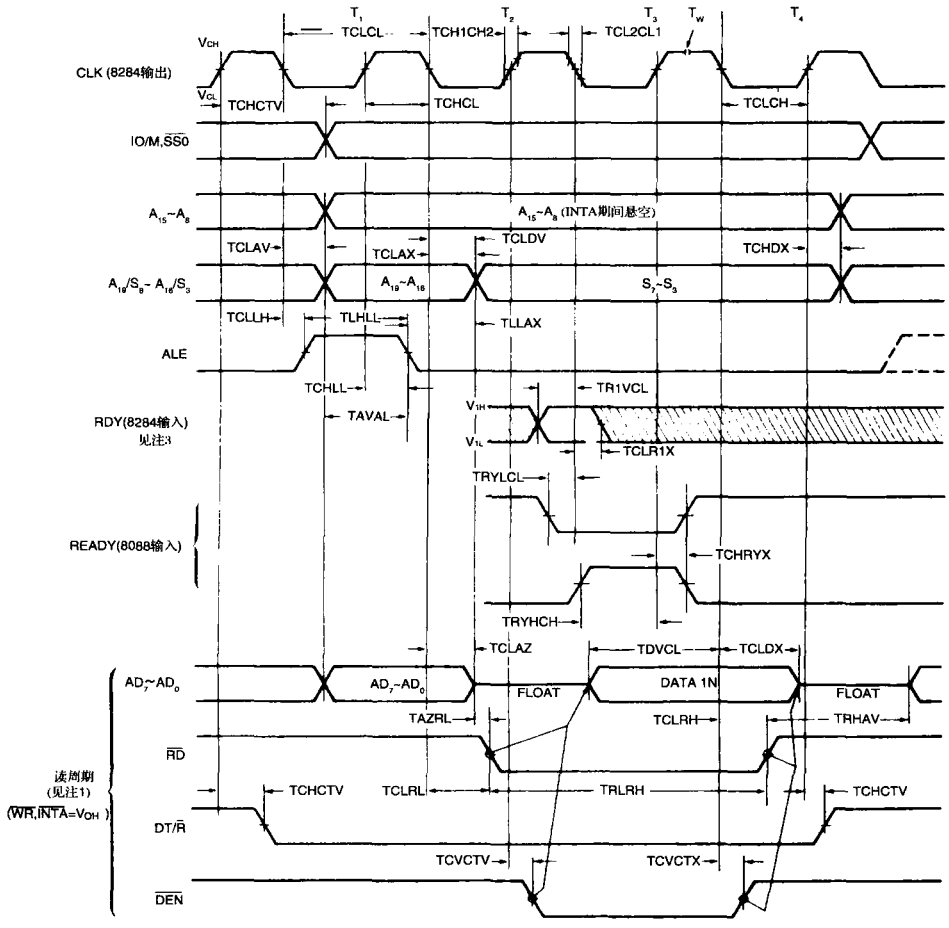


图 9-11　最小模式下 8088 读操作的总线时序

条线从 T_3 的结束处向下延伸到数据总线。在 T_3 结束时，微处理器采样数据总线。

存储器存取时间从地址出现在存储器地址总线上时开始，并持续到微处理器在 T_3 状态采样存储器数据时为止。在这段时间内大约经过 3 个时钟周期（见图 9-12）。地址在 T_1 状态开始经过 T_{CLAV}（若时钟为 5MHz，则 T_{CLAV} = 110ns）的时间才出现，这意味着必须从 3 个时钟状态（600ns）中减去 T_{CLAV}，它是与地址的出现（T_1）和数据的采样（T_3）分开的。还必须减去另一个时间：数据建立时间 T_{DVCL}，它在 T_3 前出现。因此，存储器存取时间等于 3 个时钟状态减去 T_{CLAV} 和 T_{DVCL} 之和。由于在 5MHz 时钟下 T_{DVCL} = 30ns，则允许的存储器存取时间只有 460ns（存取时间 = 600ns − 110ns − 30ns）。

当 8086/8088 工作于 5MHz 时钟下时，选择与之相连的存储器件必须在小于 460ns 的时间内存取数据，这是由于系统中的地址译码器和缓冲器引入了时间延迟。至少应留出 30 ～ 40ns 的充足时间供这些电路工作。因此，存储器速度应不低于 420ns，以便与 8086/8088 微处理器配合起来正常工作。

可影响存储器操作的惟一其他时序因素是 $\overline{RD}$ 选通信号的宽度。在时序图中，读选通的宽度是 T_{RLRH}。它的时间是 325ns（5MHz 时钟下），对于存取时间在 400ns 以内的几乎所有存储器件，这个时间宽度足够了。

交流特性 （8088：TA = 0℃～70℃，V_{CC} = 5V ± 10%）
（8088－2：TA = 0℃～70℃，V_{CC} = 5V ± 5%）

最小复杂度系统定时要求

		8088		8088－2			
符 号	参 数	最 小	最 大	最 小	最 大	单 位	测试条件
TCLCL	CLK 周期	200	500	125	500	ns	
TCLCH	CLK 为低的时间	118		68		ns	
TCHCL	CLK 为高的时间	69		44		ns	
TCH1CH2	CLK 上升时间		10		10	ns	1.0V～3.5V
TCL2CL1	CLK 下降时间		10		10	ns	3.5V～1.0V
TDVCL	数据建立时间	30		20		ns	
TCLDX	数据保持时间	10		10		ns	
TR1VCL	RDY 到 8284 的建立时间（见注1、2）	35		35		ns	
TCLR1X	RDY 到 8284 的保持时间（见注1、2）	0		0		ns	
TRYHCH	READY 到8088 的建立时间	118		68		ns	
TCHRYX	READY 到8088 的保持时间	30		20		ns	
TRYLCL	READY 对 CLK 无效时间（见注3）	－8		－8		ns	
THVCH	HOLD 建立时间	35		20		ns	
TINVCH	INTR、NMI 和TEST的建立时间（见注2）	30		15		ns	
TILIH	输入上升时间（除 CLK 外）		20		20	ns	0.8V～2.0V
TIHIL	输入下降时间（除 CLK 外）		12		12	ns	2.0V～0.8V

交流特性

时间响应

		8088		8088－2			
符 号	参 数	最 小	最 大	最 小	最 大	单 位	测试条件
TCLAV	地址有效延迟	10	110	10	60	ns	
TCLAX	地址保持时间	10		10		ns	
TCLAZ	地址浮置延迟	TCLAX	80	TCLAX	50	ns	
TLHLL	ALE 宽度	TCLCH－20		TCLCH－10		ns	
TCLLH	ALE 有效延迟		80		50	ns	
TCHLL	ALE 无效延迟		85		55	ns	
TLLAX	地址保持时间到 ALE 无效时	TCHCL－10		TCHCL－10		ns	
TCLDV	数据有效延迟	10	110	10	60	ns	
TCHDX	数据保持时间	10		10		ns	对于所有8088 输出及内部负载，CL =20pF～100pF
TWHDX	在 WR 之后的数据保持时间	TCLCH－30		TCLCH－30		ns	
TCVCTV	控制有效的延迟 1	10	110	10	70	ns	
TCHCTV	控制有效的延迟 2	10	110	10	60	ns	
TCVCTX	控制无效的延迟	10	110	10	70	ns	
TAZRL	地址浮置到 READ 有效时	0		0		ns	
TCLRL	RD有效的延迟	10	165	10	100	ns	
TCLRH	RD无效的延迟	10	150	10	80	ns	
TRHAV	RD无效到下一地址有效时	TCLCL－45		TCLCL－40		ns	
TCLHAV	HLDA 有效的延迟	10	160	10	100	ns	
TRLRH	RD宽度	2TCLCL－75		2TCLCL－50		ns	
TWLWH	WR宽度	2TCLCL－60		2TCLCL－40		ns	
TAVAL	地址有效到 ALE 低电平时	TCLCH－60		TCLCH－40		ns	
TOLOH	输出上升时间		20		20	ns	0.8V～2.0V
TOHOL	输出下降时间		12		12	ns	2.0V～0.8V

图 9-12　8088 交流特性

9.4.4 写时序

图 9-13 描述了 8088 微处理器的写时序图（8086 的写时序几乎与 8088 的相同，故这里不再画出 8086 的写时序图）。

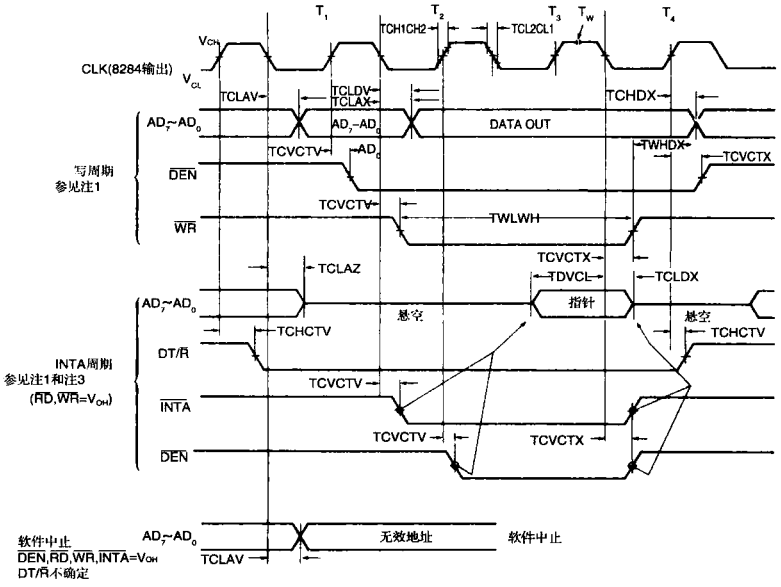


图 9-13 最小模式下 8088 写总线时序

注：1. 除非特别说明，所有信号均在 V_{OH} 和 V_{OL} 间切换。

2. RDY 在 T_2、T_3 和 T_W 快结束时被采样，确定是否插入 T_W 状态。

3. 两个 INTA 周期相继执行。在这两个周期中 8808 局部地址/数据总线是悬空状态（亦称第三态）的。这里给出了第 2 个 INTA 周期的控制信号。

4. 这里 8284A 的信号仅供参考。

5. 除非特别说明，所有定时测量均在 1.5V 下进行。

读和写时序之间的差别是很小的。$\overline{RD}$选通信号被$\overline{WR}$选通信号所代替，数据总线包含的是给存储器的信息而不是来自存储器的信息，$DT/\overline{R}$ 在整个总线周期保持逻辑 1 而不是逻辑 0。

当与一些存储器件接口时，在$\overline{WR}$变为逻辑 1 时和数据从数据总线上移走这段时间之间，时序要求特别苛刻。这是因为，存储器数据在$\overline{WR}$选通脉冲的后沿被写入。根据时序图，当 8088 以 5MHz 时钟工作时，这一严格的时间为 T_{WHDX} 或 88ns。保持时间常常比它小得多，事实上，对存储器件来说经常为 0ns。在 5MHz 时钟速率下，$\overline{WR}$选通脉冲的宽度为 T_{WLWH} 或 340ns。对于大多数存取时间在 400ns 以内的存储器件，这一宽度都是合适的。

9.5 就绪和等待状态

正如本章前面所提到的，READY 输入为较慢的存储器和 I/O 器件产生等待状态。一个等待状态（T_W）是一个额外的时钟周期，在 T_2 和 T_3 之间插入，以延长总线周期。若插入一个等待状态，则存储器存取时间由通常的 460ns（在 5MHz 时钟下），延长一个时钟周期（200ns）至 660ns。

本节讨论 8284A 时钟产生器内部的 READY 同步电路，描述如何在总线周期中有选择性地插入一个或多个等待状态，并检查 READY 输入及其所需的同步时间。

9.5.1 READY 输入

READY 输入在 T_2 结束时被采样，如果有等待状态，则在 T_W 中间被再次采样。若在 T_2 结束时 READY 是逻辑 0，则在 T_2 和 T_3 之间插入 T_W，T_3 被延迟。READY 在 T_W 中间被再次采样，以确定下一

状态是 T_W 还是 T_3。在 T_2 结束，当时钟由 1 跳变为 0 时测试 READY 是否为逻辑 0；在 T_W 中间，当时钟由 0 跳变为 1 时测试 READY 是否为逻辑 1。

输入到 8086/8088 的 READY 信号有一些严格的定时要求。图 9-14 的时序图表明 READY 引起了一个等待状态（T_W），以及要求它对系统时钟的建立和保持时间。这个操作的定时要求由 8284A 时钟产生器内部的 READY 同步电路来实现。当使用 8284A 产生 READY 时，RDY 输入（给 8284A 的就绪输入信号）出现在每个 T 状态快结束时。

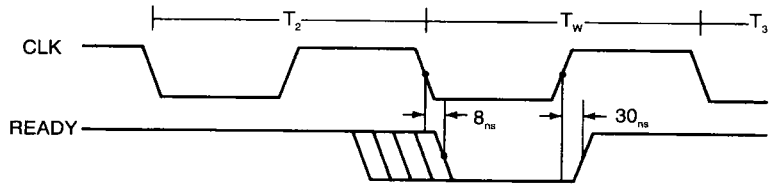


图 9-14 8086/8088 READY 输入时序

9.5.2 RDY 和 8284A

RDY 是给 8284A 时钟产生器的已同步的就绪输入信号。图 9-15 提供了此输入的时序图。尽管它不同于 8086/8088 的 READY 输入时序，但 8284A 内部电路保证了提供给 8086/8088 微处理器的 READY 同步的精度。

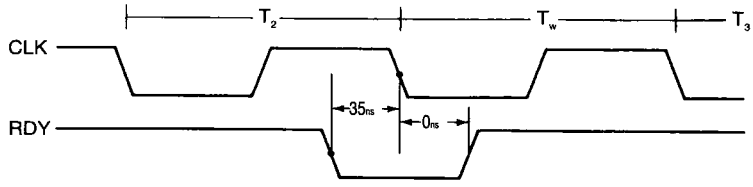


图 9-15 8284A 的 RDY 输入时序

图 9-16 再次描述了 8284A 的内部结构。该图的下半部分是 READY 同步电路。在最左边，RDY_1 和 $\overline{AEN1}$ 相“与”，RDY_2 和 $\overline{AEN2}$ 相“与”。两个“与”门的输出相“或”后，产生作为一级同步和二级同步的输入。为在触发器的输入端得到逻辑 1，RDY_1 和 $\overline{AEN1}$ 相“与”后必须为有效，或者 RDY_2 和 $\overline{AEN2}$ 相“与”后必须为有效。

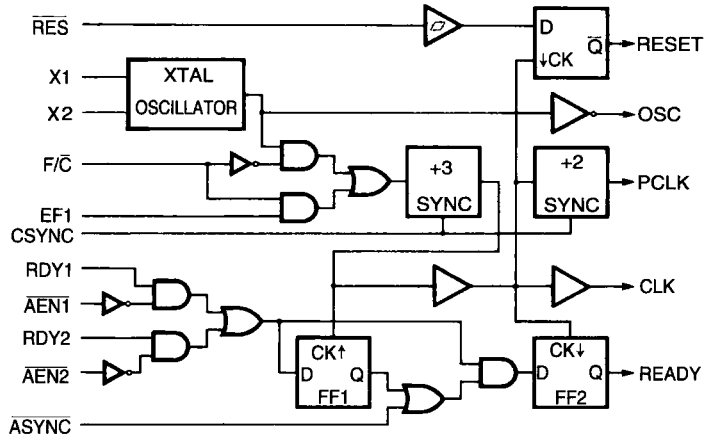


图 9-16 8284A 时钟产生器内部框图

当 ASYNC 输入为逻辑 1，选择一级同步；当它为逻辑 0，选择二级同步。若选择一级同步，则 RDY 信号从到达 8086/8088 的 READY 引脚开始一直保持，直到时钟的下一个下降沿为止。若选择二级同步，则在时钟的第一个上升沿就把 RDY 信号送到第一个触发器。该触发器的输出被送入第二个触发器，故在时钟的下一个下降沿，第二个触发器捕获到 RDY 信号。

图 9-17 描述了用于为 8086/8088 微处理器产生不同数目等待状态的电路。这里，一个 8 位串行移位寄存器（74LS164），将逻辑 0 移动一个或多个时钟周期，并从它的一个 Q 输出端送到 8284A 的 RDY_1 输入端。通过适当的跳线，此电路可提供不同数目的等待状态。还应注意移位寄存器是如何被清零回到起始点的。当$\overline{RD}$、$\overline{WR}$和$\overline{INTA}$引脚均为逻辑 1 时，寄存器输出被强制置为高电平。这 3 个信号一直保持高电平直到 T_2 状态，故移位寄存器在 T_2 的上升沿到达时第一次移位。如果需要一个等待状态，则将输出 Q_B接到或门；如果需要两个等待状态，则将输出 Q_C接到或门。依此类推。

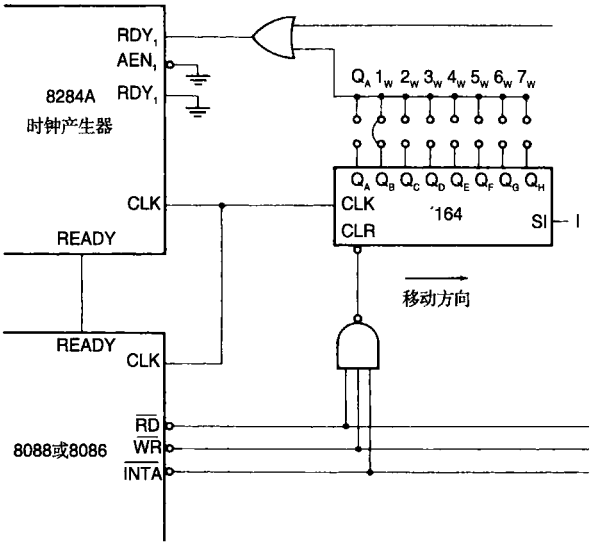


图 9-17　可产生 0 到 7 个等待状态的电路

注意，图 9-17 的电路并不总是产生等待状态，只有那些需要插入等待状态的存储器件才需要它。若来自一个存储器件的选择信号为逻辑 0，则该器件被选中，然后此电路将产生等待状态。

图 9-18 描述了这个移位寄存器作为等待状态产生器，被接成插入一个等待状态时的时序图。此图还描述了移位寄存器里触发器的内容，展示了其更详细的操作。在此例中，只产生一个等待状态。

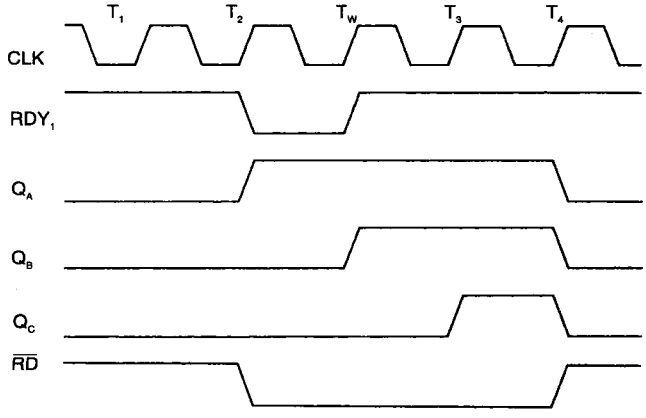


图 9-18　图 9-17 电路的等待状态产生时序

9.6 最小模式与最大模式

8086/8088 微处理器有两种工作模式：最小模式和最大模式。当模式选择引脚 MN/$\overline{MX}$连到 +5.0V 上时，选择最小模式；当将该引脚接地时，选择最大模式。这两种模式允许 8086/8088 微处理器有不同的控制结构。最小模式的操作与 8085A 的相似，8085A 是 Intel 最新一代 8 位微处理器。最大模式非常独特，用于系统中有协处理器的情况。注意，从 80286 微处理器开始，Intel 系列微处理器已不再采用最大模式。

9.6.1 最小模式操作

最小模式操作是 8086/8088 微处理器开销较小的方式（见图 9-19 最小模式的 8088 系统）。其成本较低，是因为给存储器和 I/O 的所有控制信号均由微处理器产生。这些控制信号与 Intel 8085A 的相同，8085A 是较早的 8 位微处理器。最小模式允许 8085A 的 8 位外围设备与 8086/8088 一起使用，而不必进行特殊的考虑。

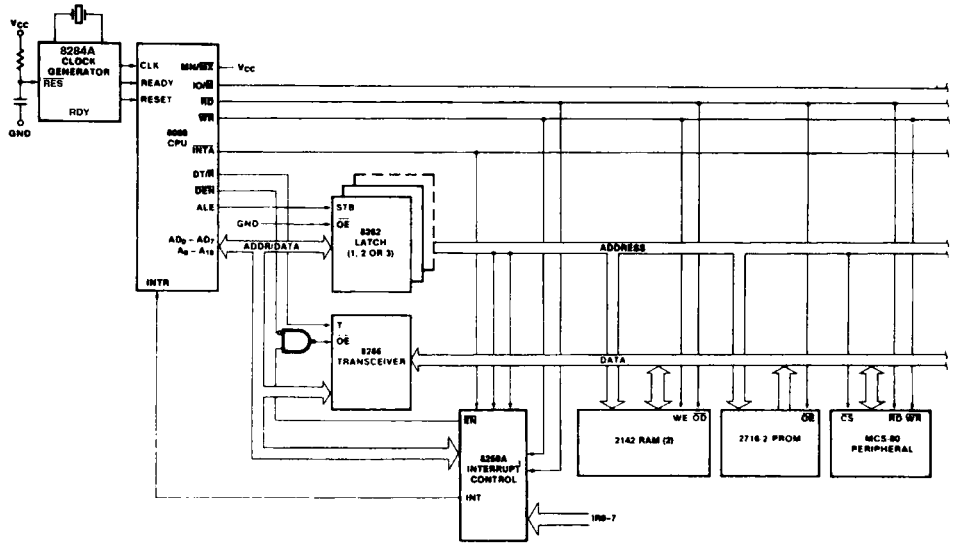


图 9-19 最小模式的 8088 系统

9.6.2 最大模式操作

最大模式操作不同于最小模式，它的某些控制信号必须由外部产生。这就需要增加一个外部总线控制器——8288 总线控制器（见图 9-20 最大模式的 8088 系统）。在最大模式下，8086/8088 没有足够的引脚用于总线控制，因为一些新的引脚和特性代替了原来的某些引脚。最大模式仅仅用在系统包含诸如 8087 算术协处理器之类的外部协处理器的情况下。

9.6.3 8288 总线控制器

工作于最大模式的 8086/8088 系统必须有一个 8288 总线控制器，以提供最大模式操作从 8086/8088 中去掉的信号。图 9-21 描述了 8288 总线控制器的框图及引脚图。

注意，8288 总线控制器产生的控制总线包含独立的 I/O 信号（$\overline{IORC}$和$\overline{IOWC}$）和存储器信号（$\overline{MRDC}$和$\overline{MWTC}$）；还包含前置存储器写选通信号（$\overline{AMWC}$）及 I/O 写选通信号（$\overline{AIOWC}$）；以及$\overline{INTA}$信号。这些信号取代了最小模式的 *ALE*、$\overline{WR}$、*IO*/$\overline{M}$、*DT*/$\overline{R}$、$\overline{DEN}$和$\overline{INTA}$信号，当 8086/8088 微处理器工作于最大模式时，$\overline{INTA}$等信号就没有了。

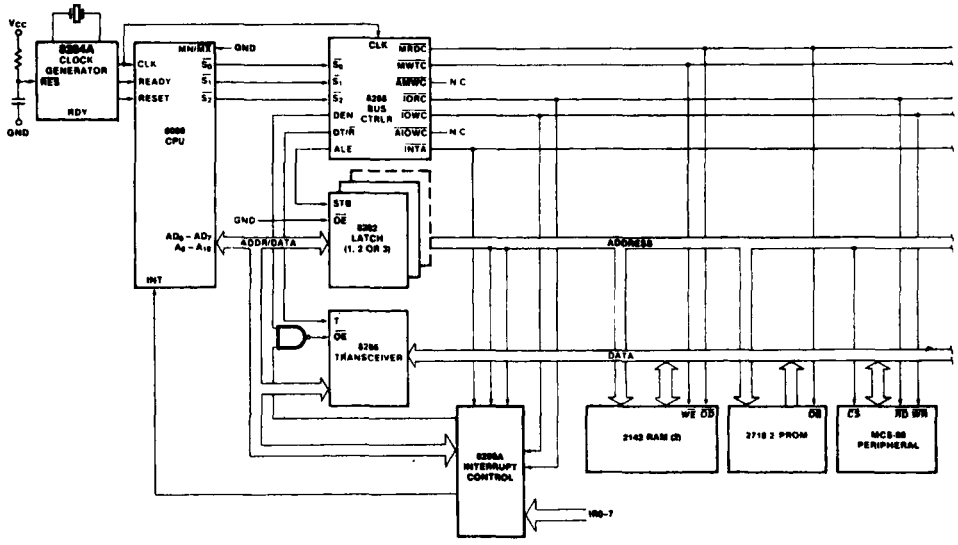


图 9-20 最大模式的 8088 系统

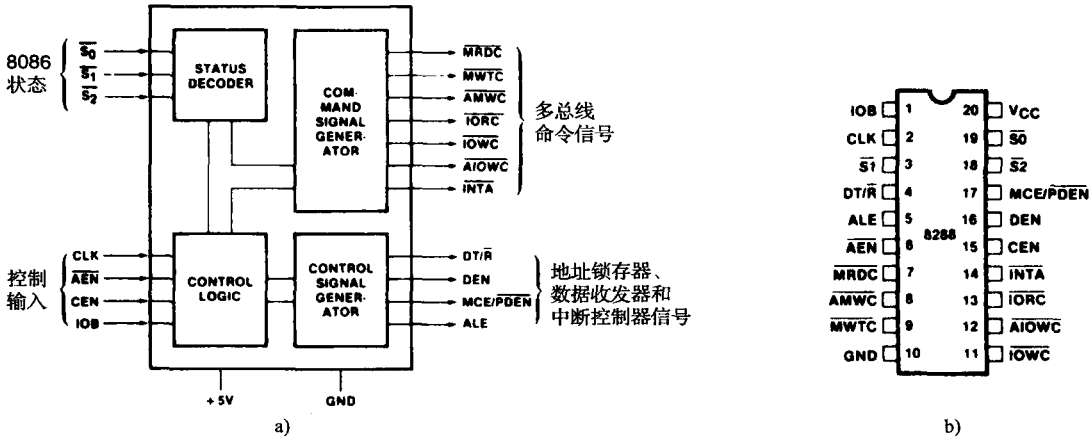


图 9-21 8288 总线控制器

a) 框图 b) 引脚图

引脚功能

下面列出了 8288 总线控制器每个引脚的说明：

S_2、S_1 和 S_0 **状态输入**，与 8086/8088 微处理器的状态输出引脚相连。这 3 个信号被译码，产生系统的定时信号。

CLK **时钟输入**，提供内部定时，必须与 8284A 时钟产生器的 CLK 输出引脚相连。

ALE **地址锁存允许输出**，用于多路分离地址/数据总线。

DEN **数据总线允许引脚**，控制系统中的双向数据总线缓冲器。注意这是一个高有效输出引脚，与工作在最小模式的微处理器上的 DEN 信号极性相反。

$DT/\overline{R}$ **数据传送/接收信号**，由 8288 输出，控制双向数据总线缓冲器的方向。

$\overline{AEN}$ **地址允许输入**，使 8288 允许存储器控制信号。

CEN **控制允许输入**，允许 8288 上的命令输出引脚。

IOB **I/O 总线模式**输入，选择 I/O 总线模式或系统总线模式操作。

$\overline{\text{AIOWC}}$ **前置 I/O 写**信号是一个命令输出，用于给 I/O 提供一个前置的 I/O 写控制信号。

$\overline{\text{IORC}}$ **I/O 读命令**输出，给 I/O 提供读控制信号。

$\overline{\text{IOWC}}$ **I/O 写命令**输出，给 I/O 提供主要的写信号。

$\overline{\text{AMWT}}$ **前置存储器写**控制引脚，给存储器提供一个早的或前置的写信号。

$\overline{\text{MWTC}}$ **存储器写**控制引脚，给存储器提供正常的写控制信号。

$\overline{\text{MRDC}}$ **存储器读**控制引脚，给存储器提供一个读控制信号。

$\overline{\text{INTA}}$ **中断响应**输出，响应由 INTR 引脚上输入的中断请求。

MCE/$\overline{\text{PDEN}}$ **主控级联/外围设备数据**输出，若 IOB 接地，则为一个中断控制器选择级联操作；若 IOB 接高电平，则允许 I/O 总线收发器。

9.7 小结

1）8086 和 8088 的主要区别是：（a）8088 是 8 位数据总线，而 8086 是 16 位数据总线；（b）8088 的$\overline{\text{SS0}}$引脚取代了 8086 的$\overline{\text{BHE}}$/S7 引脚；（c）8088 的 IO/$\overline{\text{M}}$ 引脚取代了 8086 的 M/$\overline{\text{IO}}$引脚。

2）8086 和 8088 都需要一个单 +5.0V 电源，其公差为 ±10%。

3）如果抗扰能力从通常的 400mV 减到 350mV，则 8086/8088 微处理器是与 TTL 兼容的。

4）8086/8088 微处理器可驱动 1 个 74XX、5 个 74LSXX、1 个 74SXX、10 个 74ALSXX 或 10 个 74HCXX 负载。

5）8284A 时钟产生器提供系统时钟（CLK）、READY 同步和 RESET 同步信号。

6）将一个 15MHz 晶体与 8284A 时钟产生器相连，可得到 8086/8088 标准的 5MHz 工作频率。PCLK 输出包含一个 TTL 兼容信号，频率为 CLK 的 1/2。

7）一旦 8086/8088 被复位，则它从存储器地址 FFFF0H（FFFF：0000）开始执行软件，同时中断请求引脚被禁止。

8）由于 8086/8088 总线是多路复用的，而大多数存储器和 I/O 设备却不是，所以系统在与存储器或 I/O 接口之前必须经过多路分离。多路分离由 8 位锁存器完成，其驱动脉冲由 ALE 信号得到。

9）在一个大系统中，总线必须经过缓冲，这是因为 8086/8088 微处理器只能驱动 10 个负载，而大系统常常有更多的负载。

10）总线时序在本书的后面章节中非常重要。基本的系统时序是一个总线周期，它包括 4 个时钟周期。每一总线周期能够在微处理器和存储器或 I/O 系统之间读写数据。

11）一个总线周期被分为 4 个状态或 T 时段：T_1 状态微处理器向存储器或 I/O 发送地址，向多路分离器发送 ALE 信号；T_2 状态给存储器发送写入的数据，并测试 READY 引脚，激活控制信号$\overline{\text{RD}}$或$\overline{\text{WR}}$；T_3 状态允许存储器有时间存取数据，并允许数据在微处理器和存储器或 I/O 之间传输；在 T_4 状态时写入数据。

12）当 8086/8088 微处理器以 5MHz 时钟工作时，允许存储器和 I/O 有 460ns 的时间存取数据。

13）等待状态（T_W）将总线周期延长了一个或多个时钟周期，允许存储器和 I/O 有额外的存取时间。通过控制 8086/8088 的 READY 输入来插入等待状态。在 T_2 结束时和 T_W 期间采样 READY。

14）最小模式操作与 Intel 8085A 微处理器的类似；而最大模式操作是新设计的，专门用于 8087 算术协处理器操作。

15）8288 总线控制器必须用于最大模式，给存储器和 I/O 提供控制总线信号。这是因为 8086/8088 的最大模式操作去掉了系统的一些控制信号线，以用作协处理器的控制信号。8288 可以重建这些被去掉的控制信号。

9.8 习题

1. 列出 8086 和 8088 微处理器的区别。
2. 8086/8088 是 TTL 兼容的吗？请回答并解释为什么。
3. 8086/8088 对以下器件的扇出是多少：
   （a）74XXX TTL
   （b）74ALSXXX TTL
   （c）74HCXXX CMOS
4. 当 ALE 有效时，8088 的地址/数据总线上将出现什么信息？
5. 状态位 S_3 和 S_4 的用途是什么？
6. 8086/8088 的$\overline{\text{RD}}$引脚上是逻辑 0 意味着什么？
7. 解释$\overline{\text{TEST}}$引脚和 WAIT 指令的操作。
8. 描述加在 8086/8088 微处理器的 CLK 输入引脚上的信号。
9. 当 MN/$\overline{\text{MX}}$接地时选择的是哪种工作模式？
10. 8086/8088 的$\overline{\text{WR}}$选通信号指示了有关 8086/8088 的什么操作？
11. ALE 何时被置为高阻抗状态？
12. 当 DT/$\overline{\text{R}}$ 为逻辑 1 时，它指示了有关 8086/8088 的什么

操作?

13. 当8086/8088的HOLD输入被置为逻辑1电平时，将发生什么情况?
14. 哪3个最小模式下的8086/8088引脚被译码用来发现处理器是否被中断?
15. 解释$\overline{LOCK}$引脚的操作。
16. QS_1和QS_0引脚指示了有关8086/8088的什么情形?
17. 8284A时钟产生器提供了哪3种内务事件?
18. 8284A时钟产生器是用什么方式来对晶振的输出频率进行分频的?
19. 若F/$\overline{C}$引脚被置为逻辑1电平，则晶振被禁止。在这种情况下连接到8284A的定时输入信号是什么?
20. 如果晶振工作于14MHz，则8284A的PCLK输出是_____ MHz。
21. 8284A的$\overline{RES}$输入被置为逻辑_____电平，以便复位8086/8088。
22. 8086微处理器的哪些总线是典型的经过多路分离的?
23. 8088微处理器的哪些总线是典型的经过多路分离的?
24. 哪一种TTL集成电路常用于多路分离8086/8088总线?
25. 8086微处理器上经过多路分离的$\overline{BHE}$信号有什么用途?
26. 为什么在基于8086/8088的系统中经常需要缓冲器?
27. 8086/8088的什么信号被用来选择通过74LS245双向总线缓冲器的数据流的方向?
28. 一个总线周期等于_____个时钟周期。
29. 如果8086/8088的CLK输入是4MHz，那么一个总线周期是多少?
30. 在一个总线周期中会出现哪两种8086/8088总线操作?
31. 当8086/8088以10MHz时钟工作时，它可达到多少MIPS?
32. 简短描述以下每个T状态的用途：
    (a) T_1
    (b) T_2
    (c) T_3
    (d) T_4
    (e) T_W
33. 当8086/8088以5MHz时钟工作时，允许存储器的存取时间是多少?
34. 若8088以5MHz时钟工作，则$\overline{DEN}$的宽度是多少?
35. 若READY引脚接地，则它将在8086/8088的总线周期中引入_____状态。
36. 8284A的$\overline{ASYNC}$输入完成什么功能?
37. 为在READY引脚上获得逻辑1，应在$\overline{AEN1}$和RDY_1上加上什么逻辑电平（假定$\overline{AEN2}$为逻辑1)?
38. 对比8086/8088工作的最小模式和最大模式。
39. 当8086/8088最大模式操作时，8288总线控制器主要提供什么功能?

第 10 章　存储器接口

引言

无论是简单还是复杂，每个基于微处理器的系统都有存储器。在这个方面，Intel 系列微处理器与其他任何微处理器没有区别。几乎所有的系统都包括两种主要类型的存储器：只读存储器（ROM）和随机存取存储器（RAM），即读/写存储器。只读存储器存放系统软件和永久性系统数据，而 RAM 存放临时数据和应用软件。本章介绍如何将这两种存储器与 Intel 系列微处理器接口，示例了如何使用不同的存储器地址范围，将存储器与 8 位、16 位和 32 位数据总线接口。实际上这就允许任何微处理器与任何存储系统接口。

目的

读者学习完本章后将能够：

1）译码存储器地址，并利用译码器的输出选择不同的存储器器件。
2）使用可编程逻辑器件（PLD）译码存储器地址。
3）解释如何将 RAM 和 ROM 与微处理器接口。
4）解释奇偶校验怎样检测存储器错误。
5）将存储器与 8 位、16 位、32 位和 64 位数据总线接口。
6）解释动态 RAM 控制器的操作。
7）将动态 RAM 与微处理器接口。

10.1　存储器器件

在试图将存储器与微处理器接口之前，必须完全了解存储器器件的操作。本节我们解释 4 种通用类型存储器的功能：只读存储器（ROM）、快闪存储器（EEPROM）、静态随机存取存储器（SRAM）和动态随机存取存储器（DRAM）。

10.1.1　存储器引脚

所有存储器件都通用的引脚是地址输入引脚、数据输出或数据输入/输出引脚、某些类型的选择输入引脚以及至少一个用于选择读或写操作的控制输入引脚。见图 10-1 中的 ROM 和 RAM 通用存储器器件。

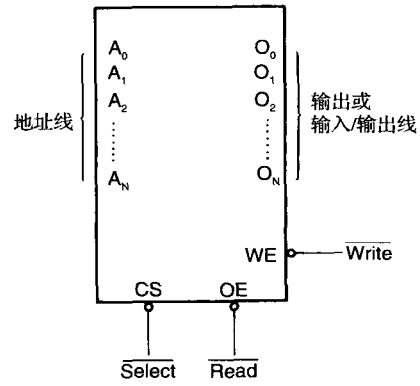


图 10-1　一个描述地址、数据和控制线的虚构的存储器器件

地址线

所有存储器件都有地址输入，用来选择存储器件中的一个存储单元。地址输入几乎总是被标为从 A_0（最低有效地址输入）到 A_n，这里的 n 可为任意值，但总是比地址引脚的总数小 1。例如，一个存储器件有 10 个地址引脚，则它的地址引脚被标为从 A_0 到 A_9。一个存储器件的地址引脚个数由其中的存储单元的数目决定。

目前，更通用的存储器件有 1K（1024）至 1G（1 073，741，824）个存储单元，可望达到 4G 甚至更多个存储单元。1K 个存储单元的存储器件有 10 个地址引脚（$A_0 \sim A_9$）；因此，需要 10 个地址输入来选择 1024 个存储单元中的任何一个，用一个 10 位二进制数（1024 种不同的组合）来选择有 1024 个单元的存储器件中

的一个单元。如果一个存储器件有 11 个地址引脚，则它有 2048（2K）个内部存储单元。因此存储单元的数目可由地址引脚的数目来推断。例如，一个 4K 存储器件有 12 个地址线，一个 8K 存储器件有 13 个地址引脚，依次类推。一个包含 1M 存储单元的存储器件需要 20 位地址（A_0 - A_{19}）。

400H 代表存储系统的 1KB 区域。如果一个存储器件被译码，其首地址为 10000H，并且它是一个 1KB 的存储器件，则它的最后一个单元地址为 103FFH（比 400H 小一个地址单元）。另一个需记住的重要的十六进制数是 1000H，因为 1000H 是 4K。一个存储器件的起始地址为 14000H，它有 4KB 长，则它在 14FFFH 单元结束（比 1000H 小一个地址单元）。第 3 个要记住的数是 64K 或 10000H。一个存储器从 30000H 单元开始，在 3FFFFH 单元结束，则它是一个 64KB 的存储器。最后要记住的一个数是很常用的 1MB 存储器，一个 1MB 存储器包含 100000H 个存储单元。

数据线

所有存储器件都有一组数据输出引脚或数据输入/输出引脚。图 10-1 中描述的器件有一组公共的输入/输出引脚。现今许多存储器件是双向公共 I/O 引脚。

通过数据线输入数据以便存储，或提取数据以便读出。对于一个 8 位宽的存储器件，其上的数据引脚被标为从 D_0 到 D_7。在此例中，有 8 个 I/O 线，这意味着存储器件在它的每个存储单元中存储 8 位数据。一个 8 位宽的存储器件常常被称为**字节宽**（byte-wide）存储器。尽管现在大多数存储器是 8 位宽，但仍有某些存储器为 16 位、4 位或只有 1 位宽。

存储器件的分类表通常给出存储单元数乘以每单元的位数。比如，一个存储器件有 1K 存储单元，每单元存储 8 位数据，则制造商经常把它写为 1K×8。一个 16K×1 的存储器件包含 16K 的 1 位存储单元。存储器件经常根据总的位容量来分类。比如，一个 1K×8 的存储器件有时写为 8K 存储器件，或者一个 64K×4 的存储器件被写为 256K 器件。各制造商的表示方法可能不同。

选择线

每个存储器件都有一个输入（有时不止一个）用来选择或允许存储器件。这种输入常称为**片选**（$\overline{CS}$）、**片使能**（$\overline{CE}$）或简称为**选择**（$\overline{S}$）输入。RAM 存储器一般有至少一个$\overline{CS}$或 $\overline{S}$ 输入，ROM 有至少一个$\overline{CE}$。如果$\overline{CE}$、$\overline{CS}$或 $\overline{S}$ 输入有效（在这种情况下因为有上划线，所以是逻辑 0），则存储器件执行一次读或写操作。如果它是无效的（在这种情况下是逻辑 1），则存储器件不能进行读或写操作，因为它是被关闭或禁止的。若存在不止一个$\overline{CS}$引脚，则所有这些选择线都必须被激活，才可以读或写数据。

控制线

所有存储器件都有一些控制输入。ROM 通常只有一个控制输入，而 RAM 通常有一个或两个控制输入。

ROM 上的控制输入通常是**输出使能**（$\overline{OE}$）或是**输出选通**（$\overline{G}$），它允许数据从 ROM 的输出数据线上流出。若$\overline{OE}$和选择输入$\overline{CE}$均有效，则输出被使能；若$\overline{OE}$无效，则输出被禁止在高阻抗状态。$\overline{OE}$线允许和禁止一组位于存储器件中的三态缓冲器，在读数据时$\overline{OE}$必须有效。

RAM 存储器件有一个或两个控制输入。若只有一个控制输入，则它常被称为 R/$\overline{W}$。只有当器件被选择输入（$\overline{CS}$）选中时，该控制线选择一次读操作或写操作。若 RAM 有两个控制输入，通常被标为$\overline{WE}$（或 $\overline{W}$）和$\overline{OE}$（或 $\overline{G}$）。这里，$\overline{WE}$（写允许）必须有效，才能执行一次存储器写操作；$\overline{OE}$必须有效，才能执行一次存储器读操作。当这两个控制信号（$\overline{WE}$和$\overline{OE}$）都存在时，它们绝不能同时有效；若两个控制输入均无效（逻辑 1），则数据既不写入也不读出，数据线处于高阻抗状态。

10.1.2 ROM 存储器

只读存储器（read-only memory，ROM）永久性地存储驻留在系统中的程序和数据，即使未接电源，其存储内容也不会改变。ROM 被永久性地编程以使数据永远保存，甚至是在不接电源时。因此这种存储器常被称为**非易失性存储器（nonvolatile memory）**。

现今 ROM 有多种型号，可从制造商那里批量购买，它们在工厂制作过程中已被编程，我们就称其

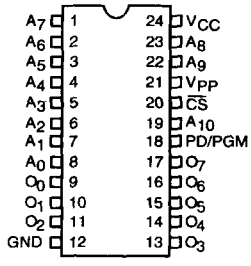
为 ROM。**EPROM**（**erasable programmable read-only memory，可擦除可编程只读存储器**）是另一种 ROM。当软件经常需要改变，或是只需要数量很少的 ROM 又想降低成本时，就常常使用 EPROM。使用 ROM 时，我们通常必须购买至少 10 000 片以补偿工厂的编程费用。EPROM 在称为 EPROM 编程器的装置上现场编程。如果将 EPROM 暴露在高强度紫外线下约 20 分钟左右，则 EPROM 也可被擦除，时间的长短依 EPROM 的类型而定。

还有 **PROM**（**programmable read-only memory 可编程只读存储器**）存储器件，尽管它们现在不是很常用。**PROM** 也可在现场被编程，是通过烧断微小的镍－铬丝或硅氧化物熔丝来实现的，但是一旦它被编程，就不能被擦除。

另外还有一种更新型的**以读为主的存储器**（**read-mostly memory，RMM**），被称为**快闪存储器**（**Flash memory**）⊖，也经常被称为 **EEPROM**（**electrically erasable programmable ROM，电可擦除可编程** ROM）、**EAROM**（**electrically alterable ROM，电可改写 ROM**）或 **NOVRAM**（**nonvolatile RAM，非易失性 RAM**）。这些存储器件在系统中是电可擦除的，但它们比普通的 RAM 需要更多的时间来擦除。快闪存储器件用于为系统存储设置信息，如计算机中的视频卡。它也可能很快取代许多计算机系统中用作 BIOS 存储器的 EPROM。一些系统将密码存储在快闪存储器件中。快闪存储器在数码相机的内存卡和 MP3 播放器的存储器中具有最大的影响力。

图 10-2 描述了 2716 EPROM，它在大多数通用 EPROM 中具有代表性。它包含 11 个地址输入和 8 个数据输出。2716 是一个 2K×8 的只读存储器件。27XXX 系列 EPROM 包括以下器件：2704（512×8）、2708（1K×8）、2716（2K×8）、2732（4K×8）、2764（8K×8）、27128（16K×8）、27256（32K×8）、27512（64K×8）以及 271024（128K×8）。每个器件均包含地址引脚、8 个数据引脚、一个或多个芯片选择输入（$\overline{CE}$）以及一个输出使能引脚（$\overline{OE}$）。

引脚配置



模式选择

模式 \ 引脚	PD/PGM (18)	$\overline{CS}$ (20)	V_{PP} (21)	V_{CC} (24)	输出 (9～11，13～17)
读	低电平	低电平	+5	+5	D_{OUT}
未选中	无关	高电平	+5	+5	高阻
功率下降	高电平	无关	+5	+5	高阻
编程	由低到高的脉冲	高电平	+25	+5	D_{IN}
程序检验	低电平	低电平	+25	+5	D_{OUT}
程序禁止	低电平	高电平	+25	+5	高阻

引脚名

$A_0 \sim A_{10}$	地址
PD/PGM	功率下降/编程
$\overline{CS}$	片选
$O_0 \sim O_7$	输出

简图

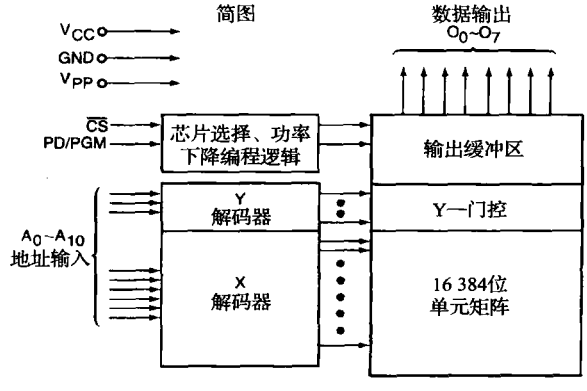


图 10-2 2716（2K×8 EPROM）的引脚图

⊖ 快闪存储器是 Intel 公司的注册商标。

图 10-3 描述了 2716 EPROM 的时序图。只有在$\overline{CE}$和$\overline{OE}$引脚都为逻辑 0 后，数据才出现在输出线上。若$\overline{CE}$和$\overline{OE}$不都是逻辑 0，则数据输出线保持在高阻抗状态或关闭状态。注意，要从 EPROM 读出数据时，V_{PP}引脚必须为逻辑 1。在某些情况下，V_{PP}引脚与 SRAM 上的$\overline{WE}$引脚在同一位置。这就允许同一个插座既可插 EPROM 又可插 SRAM。例如 27256 EPROM 和 62256 SRAM，二者都是 32K×8 器件，具有相同的引脚输出，不同的只是 EPROM 上的 V_{PP}引脚在 SRAM 上为$\overline{WE}$引脚。

A. C. 特性

T_A =0℃～70℃，V_{CC}[1] = +5V ±5%，V_{PP}[2] = V_{CC} ±0.6V[3]

符号	参数	范围			单位	测试条件
		最小值	典型值[4]	最大值		
t_{ACC1}	地址到输出延迟		250	450	ns	PD/PGM = $\overline{CS}$ = V_{IL}
t_{ACC2}	PD/PGM 到输出延迟		280	450	ns	$\overline{CS}$ = V_{IL}
t_{CO}	片选到输出延迟			120	ns	PD/PGM = V_{IL}
t_{PF}	PD/PGM 到输出延迟	0		100	ns	$\overline{CS}$ = V_{IL}
t_{DF}	未选中芯片到输出浮置	0		100	ns	PD/PGM = V_{IL}
t_{OH}	地址到输出保持	0			ns	PD/PGM = $\overline{CS}$ = V_{IL}

电容[5] T_A =25°C，f=1 MHz

符号	参数	典型值	最大值	最小值	条件
C_{IN}	输入电容	4	6	pF	V_{IN} =0V
C_{OUT}	输出电容	8	12	pF	V_{OUT} =0V

A. C. 测试条件：
输出负载：1TTL 门，C_L =100pF
输入上升和下降时间：≤20ns
输入脉冲电平：0.8V 到 2.2V
时间测量参考电平：
输入 1V 和 2V
输出 0.8V 和 2V

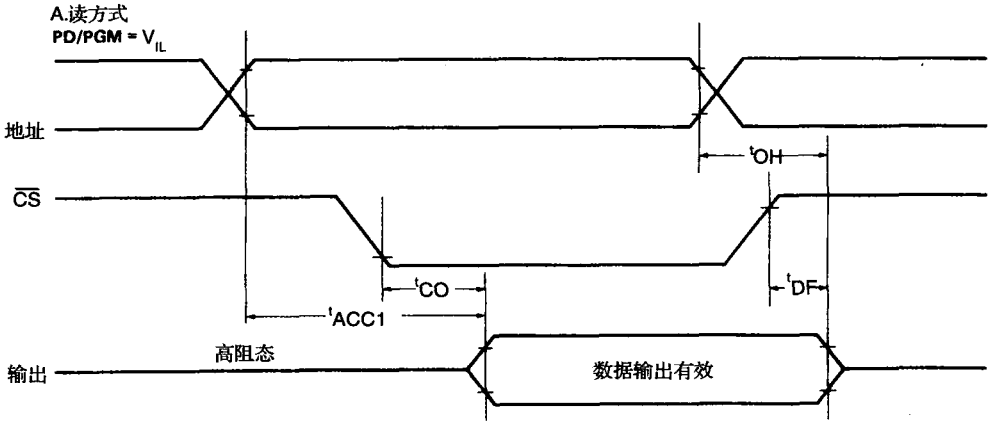


图 10-3 2716 EPROM 的交流特性时序图

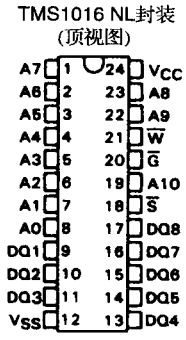
时序图和数据表提供的一条重要信息是存储器存取时间，即存储器用于读取信息的时间。正如图 10-3 所描述的，存储器存取时间（T_{ACC}）从地址出现在地址输入上开始计时，到数据出现在输出引脚上时截止。但这必须是在假定$\overline{CE}$变为低电平，同时地址输入变得稳定的基础上。同样，也必须为逻辑 0，以使输出引脚有效。这种 EPROM 的基本速度是 450ns（8086/8088 工作在 5MHz 时钟下时存储器存取数据的时间是 460ns）。这种存储器件需要等待状态才能与 8086/8088 微处理器一起正常工作，这是由于它有相当长的存取时间。如果不要等待状态，可利用更高速度的 EPROM，但需要增加成本。目前，可以得到的 ERROM 的最小存取时间是 100ns。显然，对现代微处理器中的每一个 EPROM 器件来说，等待状态是必需的。

10.1.3 静态 RAM（SRAM）器件

只要 DC 电源接通，静态 RAM 存储器件就可以保存数据。因为不需要特殊的动作（除电源外）来保

留所存储的数据，故这些器件被称为**静态存储器**（**static memory**），也被称为**易失性存储器**（**volatile memory**），因为在没有电源的情况下它们将不会保存数据。ROM 和 RAM 的主要区别在于 RAM 是在正常操作下被写入数据的，而 ROM 是在计算机外被编程，且一般只能读出数据。SRAM 存储临时性数据，用于读/写存储器容量相对比较小的情况。现今，一个小容量的存储器指容量小于 1MB 的存储器。

图 10-4 描述了 4016 SRAM，它是一个 2K×8 的读/写存储器。它有 11 个地址输入和 8 个数据输入/输出引脚。它在所有 SRAM 器件中具有代表性，只是地址和数据引脚数目各不相同。



引脚功能	
A0 ~ A10	地址
DQ1 ~ DQ8	数据输入/数据输出
$\overline{G}$	输出允许
$\overline{S}$	片选
V_{CC}	+5V 电源
V_{SS}	地
$\overline{W}$	写使能

图 10-4 TMS4016（2K×8）SRAM 的引脚图

这种 RAM 的控制输入与前面提到的那些 RAM 稍有不同。$\overline{OE}$引脚用 $\overline{G}$ 标识，$\overline{CS}$引脚用 $\overline{S}$ 标识，$\overline{WE}$引脚用 $\overline{W}$ 标识。尽管名称改变了，但功能却与以前的大致相同。其他制造商将这种流行的 SRAM 产品型号定为 2016 和 6116。

图 10-5 描述了 4016 SRAM 的时序图。如读周期时序所示，存取时间为 $t_{a(A)}$。4016 的最慢器件的存取时间为 250ns，它已足够快，可以直接与工作在 5MHz 下的 8086 或 8088 相连而不需要等待状态。再次指出，记住必须检查存取时间，以确定存储器件和微处理器的兼容性。

在推荐工作气温范围下的电特性（除非特别说明）

参数		测试条件	最小值	典型值*	最大值	单位
V_{OH}	高电平电压	I_{OH} = −1mA，V_{CC} =4.5V	2.4			V
V_{OL}	低电平电压	I_{OL} =2.1mA，V_{CC} =4.5V			0.4	V
I_I	输入电流	V_I = 0V ~ 5.5V			10	μA
I_{OZ}	关闭状态输出电流	$\overline{S}$ 或 $\overline{G}$ 为2V，或 $\overline{W}$ 为0.8V V_O =0V ~ 5.5V			10	μA
I_{CC}	V_{CC}电源电流	I_O =0mA，V_{CC} =5.5V T_A = 0℃（最坏情况）		40	70	mA
C_i	输入电容	V_I =0V，f=1MHz			8	pF
C_O	输出电容	V_O = 0V，f=1MHz			12	pF

* 所有典型值均在 V_{CC} =5V，T_A =25℃

在推荐电源电压范围及工作气温范围下的定时要求

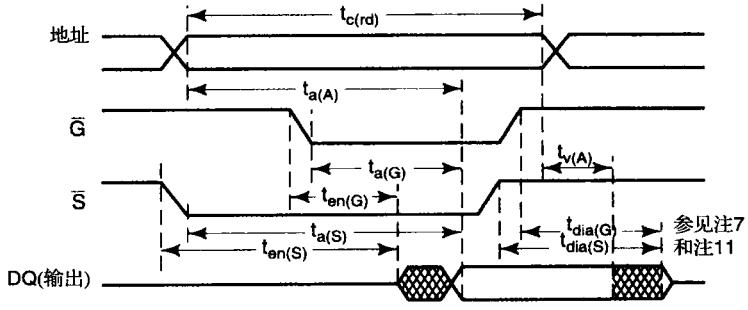
参数		TMS4016 -12		TMS4016 -15		TMS4016 -20		TMS4016 -25		单位
		最小值	最大值	最小值	最大值	最小值	最大值	最小值	最大值	
$t_{c(rd)}$	读周期时间	120		150		200		250		ns
$t_{c(wr)}$	写周期时间	120		150		200		250		ns
$t_{w(W)}$	写脉冲宽度	60		80		100		120		ns
$t_{su(A)}$	地址建立时间	20		20		20		20		ns
$t_{su(S)}$	片选建立时间	60		80		100		120		ns
$t_{su(D)}$	数据建立时间	50		60		80		100		ns
$t_{h(A)}$	地址保持时间	0		0		0		0		ns
$t_{h(D)}$	数据保持时间	5		10		10		10		ns

图 10-5 TMS4016 SRAM 的交流特性和时序图

在推荐的电压范围及 T_A = 0℃至70℃下的转换特性

参数		TMS4016-12	TMS4016-15	TMS4016-20	TMS4016-25	单位
		最小值 最大值	最小值 最大值	最小值 最大值	最小值 最大值	
$t_{a(A)}$	从地址开始的存取时间	120	150	200	250	ns
$t_{a(S)}$	从片选为低电平开始的存取时间	60	75	100	120	ns
$t_{a(G)}$	从输出允许为低电平开始的存取时间	50	60	80	100	ns
$t_{v(A)}$	地址改变后输出数据有效	10	15	15	15	ns
$t_{dis(S)}$	片选为高电平后输出禁止时间	40	50	60	80	ns
$t_{dis(G)}$	输出允许为高电平后输出禁止时间	40	50	60	80	ns
$t_{dis(W)}$	写允许为低电平后输出禁止时间	50	60	60	80	ns
$t_{en(S)}$	片选为低电平后输出允许时间	5	5	10	10	ns
$t_{en(G)}$	输出允许为低电平后输出允许时间	5	5	10	10	ns
$t_{en(W)}$	写允许为高电平后输出允许时间	5	5	10	10	ns

读周期时序波形（参见注3）



读周期时序波形（参见注4）

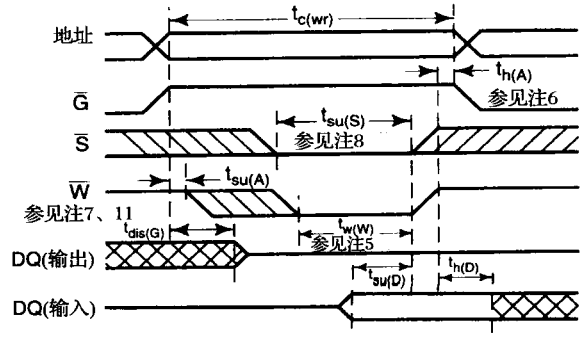


图 10-5 （续）

第2个写周期时序波形（参见注4和注9）

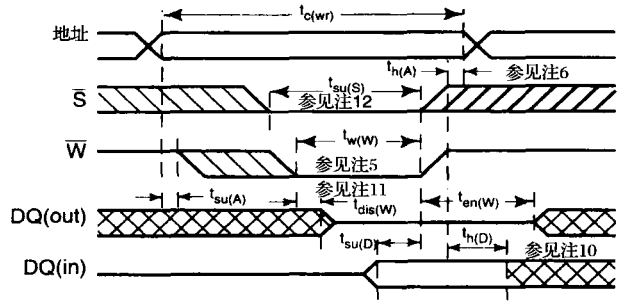


图 10-5 （续）

注：1. 除 $t_{dis(W)}$ 和 $t_{en(W)}$ 要求 $C_L = 5pF$ 外，所有测试均要求 $C_L = 100pF$。

2. t_{dis} 和 t_{en} 参数是采样得到的，并未经过100%测试。

3. $\overline{W}$ 为高电平时是读周期。

4. 在所有地址转换期间，$\overline{W}$ 必须为高电平。

5. 在 $\overline{S}$ 和 $\overline{W}$ 均为低电平时出现一次写操作。

6. $t_{h(A)}$ 从 $\overline{S}$ 或 $\overline{W}$ 变为高电平时开始测量，一直到写周期结束。

7. 在这个周期里I/O引脚是输出状态，以便反相的输入信号不会加到输出上。

8. 若在 $\overline{W}$ 向低跳变的同时或在 $\overline{W}$ 跳变后出现S向低跳变，则输出保持在高阻抗状态。

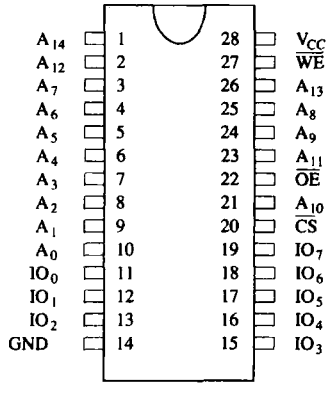
9. G继续为低电平（$G = V_{IL}$）。

10. 若 $\overline{S}$ 在这个周期里为低电平，则I/O引脚是输出状态，反相的数据输入信号不能加到输出上。

11. 跳变在偏离稳定状态电压±200mV测量。

12. 若 $\overline{S}$ 向低跳变出现在 $\overline{W}$ 向低跳变之后，则在 $\overline{W}$ 向低跳变后的 $t_{dis(W)}$ 这段时间，反相的数据输入信号不能加到输出上。

图10-6描述了62256 SRAM的引脚图，它是一个32K×8的静态RAM。28脚集成电路封装形式，存取时间为120ns或150ns。其他通用SRAM器件的容量有8K×8、128K×8、256K×8、512K×8和1M×8。SRAM的存取时间小到只有1ns，通常用在计算机的高速缓冲存储系统中。



引脚功能	
$A_0 \sim A_{14}$	地址
$IO_0 \sim IO_7$	数据引脚
$\overline{CS}$	片选
$\overline{OE}$	输出允许
$\overline{WE}$	写允许
V_{CC}	+5V电源
GND	地

图10-6 62256（32K×8）SRAM的引脚图

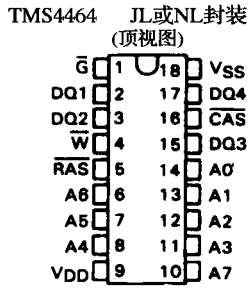
10.1.4 动态RAM（DRAM）存储器

现在可得到的最大静态RAM是1M×8。而另一方面，动态RAM有更大的容量，可达512M×1。在所有其他方面，DRAM基本上与SRAM相同，只是它在一个集成电容上仅将数据保留2ms或4ms的时间。2ms或4ms之后DRAM中的内容必须全部重写（刷新），因为存储逻辑1或逻辑0的电容上的电荷放掉了。

用一个程序来读每个存储单元的内容，然后再重写它们，这几乎是不可能的工作，因此制造商从

DRAM 内部构造上使其与 SRAM 不同。在 DRAM 中，通过在 2ms 或 4ms 的间隔时间读 256 次，存储器的整个内容就被刷新了。刷新也出现在一次写、一次读或是一个特殊的刷新周期内。有关 DRAM 刷新的更多信息见 10.6 节。

DRAM 存储器的另一个缺点是它需要的地址引脚太多，因此制造商决定多路复用地址输入。图10-7描述了一个 64K×4 DRAM，即 TMS4464，它存储 256K 位数据。注意，它只有 8 个地址引脚，而它应有 16 个——用来寻址 64K 存储单元。要将 16 位地址强制用 8 个地址引脚来表示，惟一的办法是用两个 8 位来扩展。这一操作需要两个特殊引脚：**列地址选通**（$\overline{CAS}$）和**行地址选通**（$\overline{RAS}$）。首先，$A_0 \sim A_7$ 被置于地址线上，由$\overline{RAS}$选通进入一个内部的行锁存器作为行地址。然后，$A_8 \sim A_{15}$被置于同样的 8 个地址线上，由$\overline{CAS}$选通进入一个内部的列锁存器作为列地址（见图 10-8 时序图）。保持在这些内部锁存器中的 16 位地址寻址 4 位存储单元中的内容。注意，$\overline{CAS}$还完成对 DRAM 的片选输入功能。



a)

引脚功能	
A0 ~ A7	地址
$\overline{CAS}$	列地址选通
DQ1 ~ DQ4	数据输入/数据输出
$\overline{G}$	输出允许
$\overline{RAS}$	行地址选通
V_{DD}	+5V 电源
V_{SS}	地
$\overline{W}$	写允许

b)

图 10-7 TMS4464（64K×4）DRAM 的引脚图

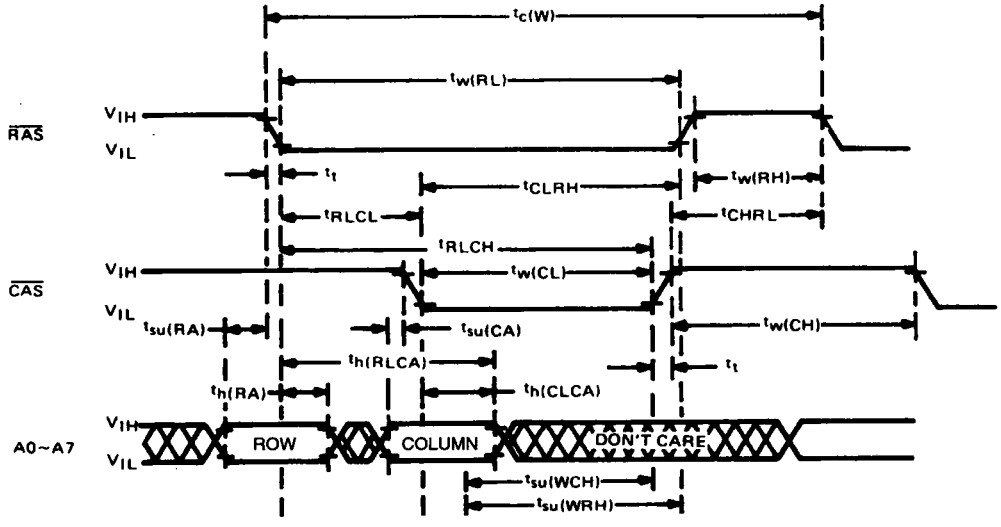


图 10-8 TMS4464 DRAM 的$\overline{RAS}$、$\overline{CAS}$和地址输入时序

图 10-9 图示了一组多路转换器，用于把列地址和行地址成对地选通到 TMS4464 DRAM 的 8 位地址线上。这里，$\overline{RAS}$信号不仅选通 DRAM 的行地址，还选择哪一部分地址加在地址输入上，这是由于多路转换器有长时间的传播延迟。当$\overline{RAS}$为逻辑 1 时，B 输入连接到多路转换器的 Y 输出；当$\overline{RAS}$变为逻辑 0 时，A 输入连接到 Y 输出。由于内部行地址锁存器是边沿触发，故它在输入端的地址变为列地址之前就捕获到了行地址。关于 DRAM 及其接口的更多细节参见 10.6 节。

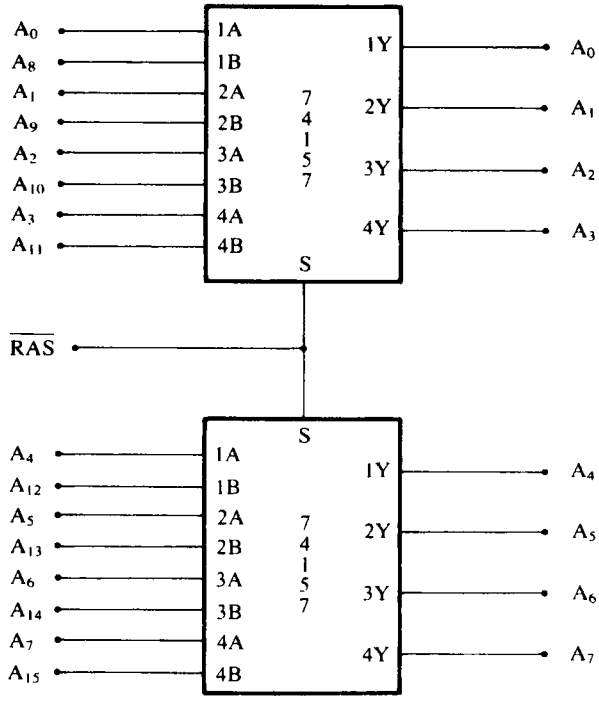
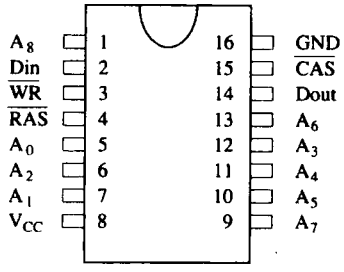


图 10-9 TMS4464 DRAM 的地址多路转换器

正如 SRAM 一样，R/$\overline{W}$ 线为逻辑 0 时对 DRAM 写数据，但 DRAM 没有 $\overline{G}$ 引脚或允许引脚，也没有 $\overline{S}$（选择）输入。正如前面提及过的，$\overline{CAS}$输入用来选择 DRAM。如果 DRAM 被选中，当 R/$\overline{W}$ = 0，将数据写入 DRAM；当 R/$\overline{W}$ = 1，从 DRAM 读出数据。

图 10-10 显示了 41256 动态 RAM 的引脚图。该器件被组织成一个 256K × 1 存储器，只需要 70ns 来存取数据。



$A_0 \sim A_8$	地址
Din	数据输入
Dout	数据输出
$\overline{CAS}$	列地址选通
$\overline{RAS}$	行地址选通
$\overline{WR}$	写使能
V_{CC}	+5V电源
GND	地

图 10-10 组织成 256K × 1 存储器件的 41256DRAM

最近，又出现了更大容量的 DRAM，如 16M × 1、256M × 1、512M × 1、1G × 1 和 2G × 1 存储器。现在又计划研制 4G × 1 的存储器。DRAM 常置在被称为 SIMM（Single In-line Memory Modules，单列直插式存储器模块）的小电路板上，图 10-11 给出了两个最通用的 SIMM 的引脚图。30 引脚的 SIMM 常被组织成 1M × 8 或 1M × 9，4M × 8 或 4M × 9（图 10-11 中为 4M × 9），第 9 位是奇偶校验位。图 10-11 中还给出了更新的 72 脚 SIMM，它常被组织成 1M × 32 或 1M × 36（带奇偶校验位）。其他容量的还有 2M × 32、4 M × 32、8 M × 32 和 16M × 32，它们也有奇偶校验位。图 10-11 所示的是一个4 M × 36 SIMM，有 16MB 容量的存储器。

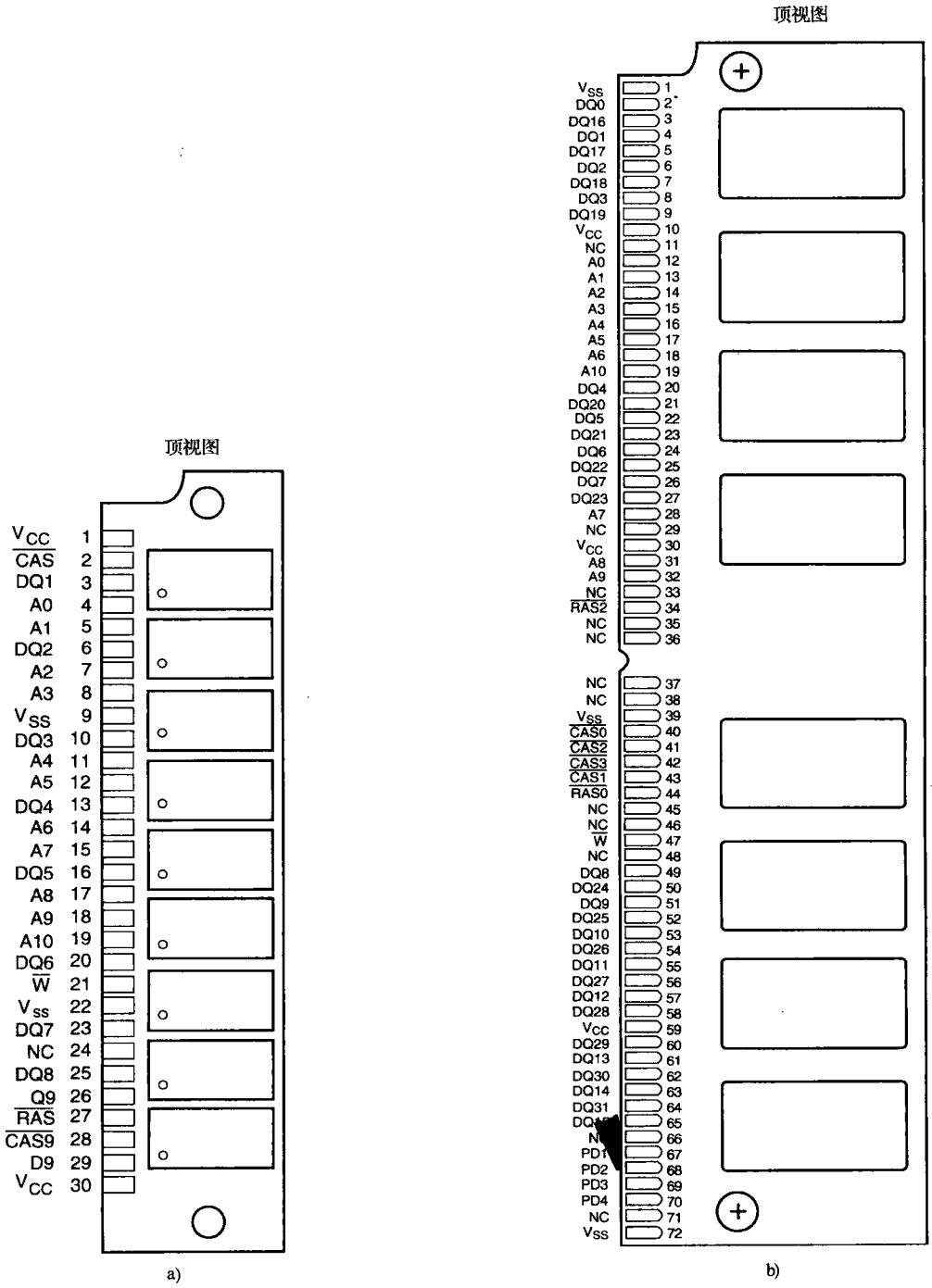


图 10-11　30 引脚和 72 引脚 SIMM 的引脚图

a）组织成 4M×9 的 30 引脚 SIMM　b）组织成 4M×36 的 72 引脚 SIMM

近来，许多系统正在使用 Pentium ～ Pentium 4 微处理器。这些微处理器有 64 位数据总线，它们不使用这里描述的 8 位 SIMM。72 引脚的 SIMM 用起来也很麻烦，因为它们必须成对使用，以获得 64 位宽的数据线。今天，64 位 DIMM（Dual In-line Memory Modules，双列直插式存储器模块）正变为大多

数系统的标准。这些模块中的存储器被组织成 64 位宽。通常容量是 16MB（2M×64）、32MB（4M×64）、64MB（8M×64）、128MB（16M×64）、256MB（32M×64）、512MB（64M×64）和 1GB（128×64）。DIMM 的引脚输出如图 10-12 所示。DIMM 模块可以有 DRAM、EDO、SDRAM 和 DDR（双数据速率）多种形式，既可以有 EPROM，也可以没有 EPROM。EPROM 为系统提供了存储器件即插即用时的容量和速度信息。

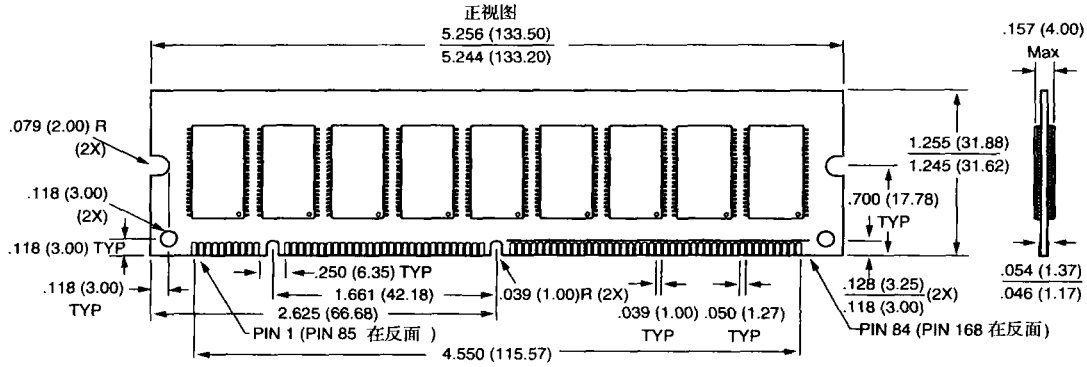


图 10-12 168 引脚 DIMM 的引脚图

存储器市场又增加了 RAMBUS 公司推出的 RIMM 存储器模块，但是这种存储器类型已经退出了市场。同 SDRAM 一样，RIMM 也有 168 个引脚，但每一引脚有两个平面，因此总的引脚数为 336。目前最快的 SDRAM 是 PC-4400 或 500 DDR，它以 4.4GB/s 的速率工作。相比之下，800MHz 的 RIMM 以 3.2GB/s 的速率工作。RIMM 模块被组织成 32 位宽的器件。这意味着如果要在板上组装 Pentium 4 存储器，RIMM 存储器必须成对使用。Intel 公司宣称，使用 RIMM 模块的 Pentium 4 系统比使用 PC－100 存储器的 PentiumⅢ要快 3 倍。据 RAMBUS 公司称，目前的 800MHz RIMM 在不久的将来其速度将提高到 1200MHz，但速度仍不足以占据足够的市场份额。

目前最新的 DRAM 是 **DDR**（**double-data rate，双数据速率**）和 DDR2 存储器。DDR 存储器在每一个时钟边缘都传输数据，这使得它的速度是 SDRAM 的 2 倍。这不影响存储器的存取时间，仍需要许多等待状态来操作这种类型的存储器，但它可以比一般 SDRAM 甚至包括 RDRAM 在内的存储器快很多。

10.2 地址译码

为了将一个存储器件与微处理器相连，有必要译码微处理器发送来的地址。译码使得存储器工作在存储器映射表中的惟一区域或分区里。如果没有地址译码器，那么只有一个存储器件可以与微处理器相连，但它实质上是没有用的。在本节中，我们介绍了几种最通用的地址译码技术，以及在许多系统中常见的译码器。

10.2.1 为什么要进行存储器译码

比较 8088 微处理器和 2716 EPROM，二者的地址引脚数目明显有差别——EPROM 有 11 个地址引脚，而微处理器有 20 个地址引脚。这意味着当微处理器读或写数据时，发送 20 位存储器地址。由于 EPROM 只有 11 个地址引脚，所以必须要修正不匹配的问题。如果只将 8088 的 11 个地址引脚与存储器相连，则 8088 只能寻址 2KB 的存储器，而不是期望的 1MB。译码器通过译码那些未与存储器器件相连的地址引脚来修正不匹配问题。

10.2.2 简单的与非门译码器

当使用 2K×8 EPROM 时，8088 的地址线 A_{10}～A_0 与 EPROM 的地址输入 A_{10}～A_0 相连。余下的 9 个地址线（A_{19}～A_{11}）被连到一个与非门译码器的输入端（见图 10-13）。译码器从 8088 微处理器的整

个 1MB 地址范围内的许多个 2KB 段中选择 EPROM。

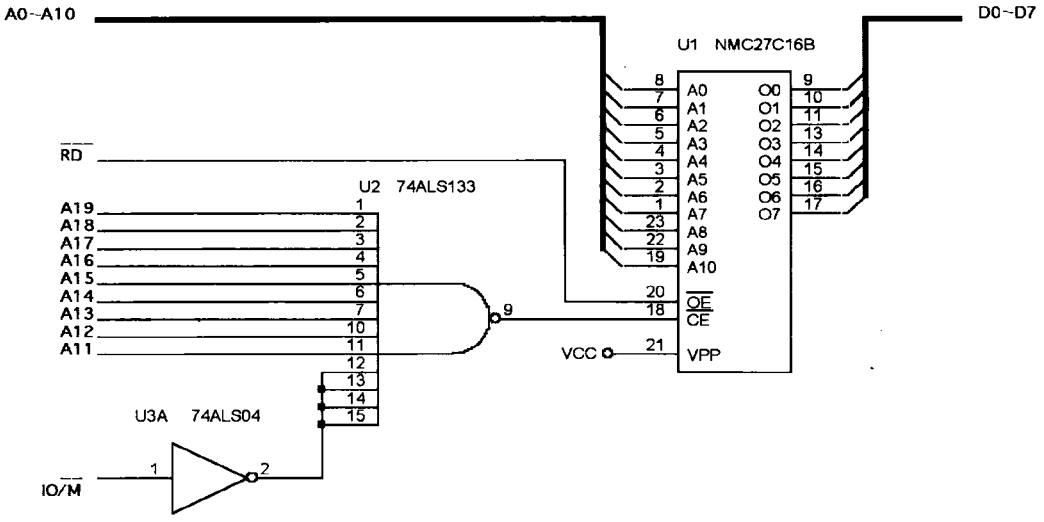


图 10-13 一个简单的与非门译码器，用于选择存储单元为 FF800H ～ FFFFFH 的 2716 EPROM 存储器件

在这个电路中，用一个简单的与非门译码存储器地址。如果连接到输入端（A_{19}～A_{11}）的 8088 地址线均为逻辑 1，则与非门的输出为逻辑 0。与非门译码器的低有效，即逻辑 0 输出与$\overline{CE}$输入线相连，以选中（允许）EPROM。前面讲过，只要$\overline{CE}$为逻辑 0，则仅当$\overline{OE}$也是逻辑 0 时，数据将从 EPROM 中读出。$\overline{OE}$引脚由 8088 的$\overline{RD}$信号或其他系列微处理器的$\overline{MRDC}$（**存储器读控制**）信号激活。

如果 20 位二进制地址由与非门译码，写成最左边 9 位是逻辑 1，最右边 11 位是无关项（X），则可以确定 EPROM 的实际地址范围（无关项为逻辑 1 或逻辑 0 都可以）。

例 10-1 描述了这个 EPROM 的地址范围是如何被确定的，即写下其外部译码地址位（A_{19}～A_{11}），而 EPROM 译码的地址位（A_{10}～A_0）为无关项。正如例中所描述的，无关项首先被写为逻辑 0 来定位最低地址，然后被写为逻辑 1 来定位最高地址。例 10-1 还将这些二进制分界线表示为十六进制地址。这里，2K EPROM 被译码为存储器地址单元 FF800H ～ FFFFFH。注意，这是存储器的一个 2KB 段，位于 8086/8088 的复位地址（FFFF0H），对一个 EPROM 来说是最合适的位置。

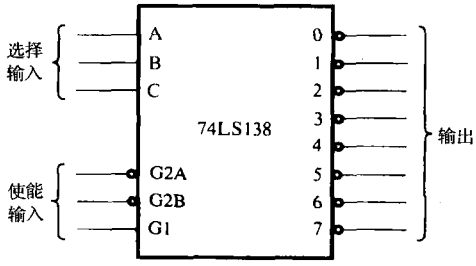
例 10-1

```
1111 1111 1XXX XXXX XXXX

          或

1111 1111 1000 0000 0000 = FF800H
          到
1111 1111 1111 1111 1111 = FFFFFH
```

尽管本例用来说明译码，但很少用与非门进行存储器译码，因为每个存储器件需要它自己的与非门译码器。由于与非门译码器和常常配合它使用的反相器增加了额外的成本，所以需要寻找另外的译码器。

10.2.3 3-8 线译码器（74LS138）

许多基于微处理器的系统中用到的，更通用但不是惟一的集成电路译码器是 74LS138 3-8 线译码器。图 10-14 描述了这种译码器及其真值表。



输入						输出							
使能			选择										
$\overline{G2A}$	$\overline{G2B}$	G1	C	B	A	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{6}$	$\overline{7}$
1	X	X	X	X	X	1	1	1	1	1	1	1	1
X	1	X	X	X	X	1	1	1	1	1	1	1	1
X	X	0	X	X	X	1	1	1	1	1	1	1	1
0	0	1	0	0	0	0	1	1	1	1	1	1	1
0	0	1	0	0	1	1	0	1	1	1	1	1	1
0	0	1	0	1	0	1	1	0	1	1	1	1	1
0	0	1	0	1	1	1	1	1	0	1	1	1	1
0	0	1	1	0	0	1	1	1	1	0	1	1	1
0	0	1	1	0	1	1	1	1	1	1	0	1	1
0	0	1	1	1	0	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	1	1	1	0

图 10-14 74LS138 3-8 线译码器及其功能表

其真值表表明，在任何时候 8 个输出只有一个会变成低电平。为使译码器的任一输出变为低电平，3 个允许输入（$\overline{G2A}$、$\overline{G2B}$和 G1）均必须有效。即$\overline{G2A}$和$\overline{G2B}$输入必须都为低电平（逻辑 0），G1 必须为高电平（逻辑 1）。一旦 74LS138 被允许，地址输入（C、B 和 A）就选择某一个输出引脚变低。试想象一下，8 个 EPROM 的$\overline{CE}$输入连接到译码器的 8 个输出上！这是一个非常有用的器件，因为它同时可选择 8 个不同的存储器件。直到目前这种器件还在广泛使用。

译码器电路示例

如图 10-15 所示，译码器输出连接到 8 个不同的 2764 EPROM 存储器件上。这里，译码器选择了 8 个 8KB 的存储体，总的存储器容量为 64KB。该图还描述了每一存储器件的地址范围，以及与存储器件相连的公共连线。注意，8088 的所有地址线都连接到这个电路上。还有，注意译码器的输出连到 EPROM 的$\overline{CE}$输入，来自 8088 的$\overline{RD}$信号连到 EPROM 的$\overline{OE}$输入。这使得只有选中的 EPROM 可以被允许，并在$\overline{RD}$变为逻辑 0 时通过数据总线向微处理器发送数据。

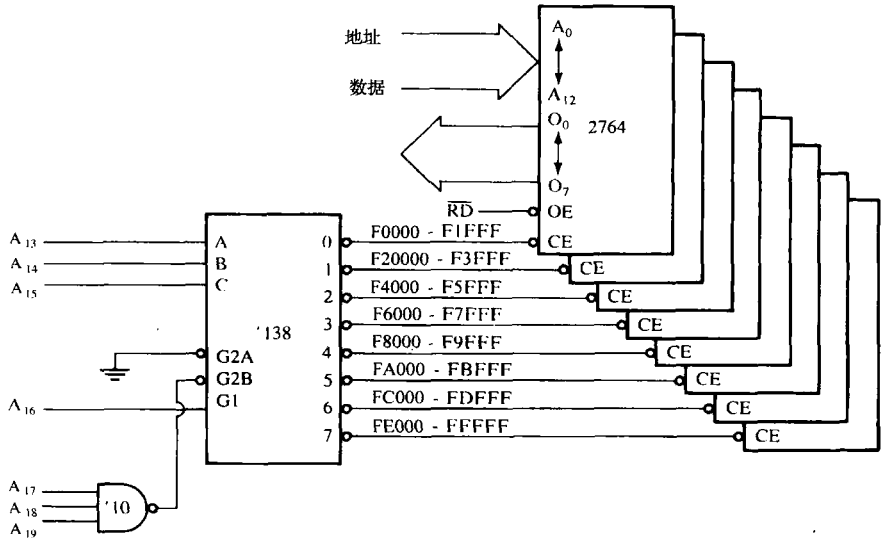


图 10-15 在一个基于 8088 微处理器的系统中，使用 8 个 2764 EPROM 来形成 64K×8 存储区的电路，所选地址为 F0000H～FFFFFH

在此电路中，一个 3 输入与非门连接到地址位 A_{19}～A_{17}。当所有 3 个地址输入为高电平时，与非门的输出变为低电平，从而使能 74LS138 的输入$\overline{G2B}$。输入 G1 直接与 A_{16}相连。换句话说，为了使能这个译码器，头 4 个地址线（A_{19}～A_{16}）必须均为高电平。

地址输入 C、B 和 A 连到微处理器的地址线 A_{15}～A_{13}。这 3 个地址输入确定哪一个输出线变低，以及

当 8088 输出一个在此范围内的存储器地址给存储系统时，选中哪一个 EPROM。

例 10-2 说明了如何确定整个译码器的地址范围。注意范围为 F0000H ～ FFFFFH，这是一个 64KB 范围的存储区。

例 10-2

```
1111 XXXX XXXX XXXX XXXX
          或

1111 0000 0000 0000 0000 = FOOOOH
          到
1111 1111 1111 1111 1111 = FFFFFH
```

它是如何确定与译码器输出相连的每个存储器件的地址范围呢？再次写下二进制位模式，这次 C、B 与 A 的输入不是无关项。例 10-3 体现了译码器的输出 0 如何变为低电平，以选中与之相连的 EPROM。这里，C、B 与 A 均为逻辑 0。

例 10-3

```
     CBA
1111 OOOX XXXX XXXX XXXX

          或

1111 0000 0000 0000 0000 = FOOOOH
          到
1111 0001 1111 1111 1111 = F1FFFH
```

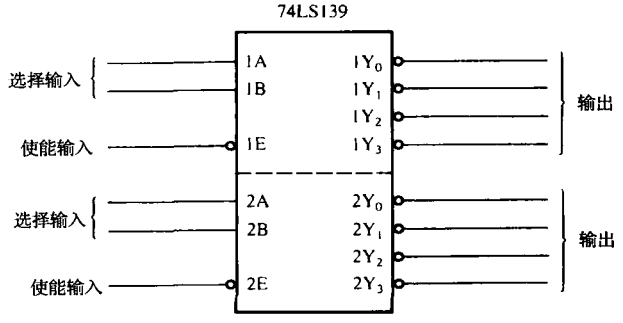
如果需要与译码器的输出 1 相连的 EPROM 的地址范围，则严格按照与输出 0 同样的方式来确定。惟一区别是现在 C、B 与 A 输入为 001，而不是 000（参见例 10-4）。余下的输出地址范围按同样的方式确定，只需将输出线的二进制地址代入 C、B 与 A 即可。

例 10-4

```
     CBA
1111 001X XXXX XXXX XXXX

          或

1111 0010 0000 0000 0000 = F2000H
          到
1111 0011 1111 1111 1111 = F3FFFH
```

10.2.4 双 2-4 线译码器（74LS139）

另一种有用的译码器是 74LS139 双 2-4 线译码器。图 10-16 描述了这种译码器的引脚图和真值表。74LS139 包含两个独立的 2-4 线译码器——每个译码器具有自己的地址、使能和输出线。



输入			输出			
$\overline{E}$	A	B	$\overline{Y_0}$	$\overline{Y_1}$	$\overline{Y_2}$	$\overline{Y_3}$
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	X	X	1	1	1	1

图 10-16 74LS139 双 2-4 线译码器的引脚图和真值表

使用74LS139的更复杂的一种译码器如图10-17所示。这个电路用到了一个128K×8的EPROM（271000）和一个128K×8的SRAM（621000）。EPROM被译码到存储器E0000H～FFFFFH地址范围内，SRAM被译码到00000H～1FFFFH地址范围内。这是一个典型的小型嵌入式系统，EPROM定位到存储器空间的顶部，SRAM定位到存储器空间的底部。

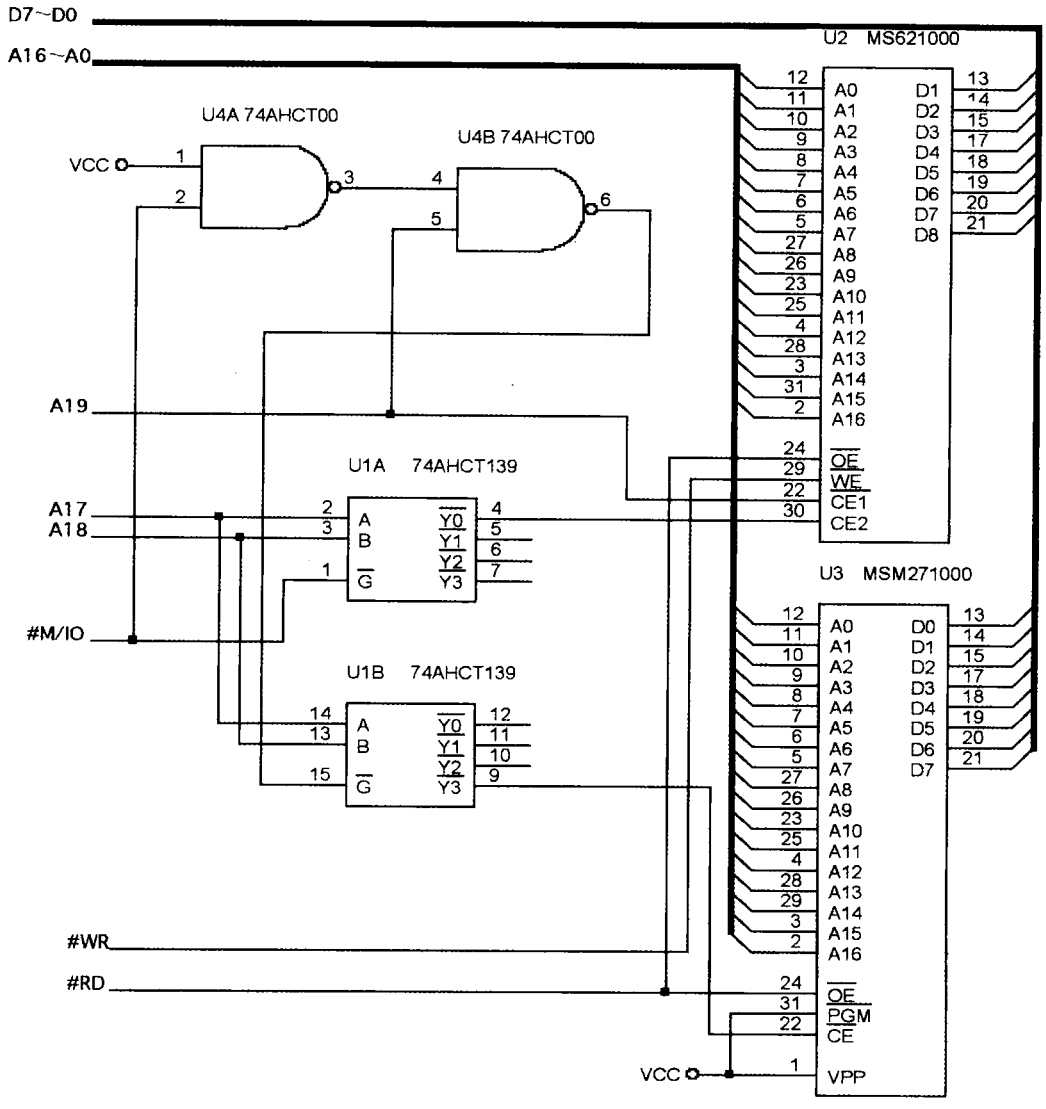


图10-17 用74HC139构成的一个采样存储系统

如果IO/$\overline{M}$信号和地址A_{19}均为逻辑0，每当地址A_{17}和A_{18}为逻辑0时，译码器U1A的输出$\overline{Y0}$就会激活SRAM。这时就可以选择SRAM从00000H～1FFFFH的任意地址。第二个译码器（U1B）就稍微复杂一些，因为在IO/$\overline{M}$是逻辑0，A_{19}为逻辑1时，与非门（U4B）会选择译码器，这时就会选择EPROM从E0000H到FFFFFH的地址。

10.2.5 PLD可编程译码器

本节介绍可编程逻辑器件PLD作为译码器的使用情况。在最新的存储器接口中PAL替代了

PROM 地址译码器。有 3 种 SPLD（simple PLD）器件以同样的方式工作，但名字不同：PLA（programmable logic array，可编程逻辑阵列），PAL（programmable array logic，可编程阵列逻辑）和 GAL（gated array logic，门阵列逻辑）。尽管这些器件从 70 年代中期起就已出现，但只是从 90 年代开始才广泛在存储系统和数字设计中。PAL 和 PLA 与 PROM 一样都是熔丝型可编程器件，还有一些 PLD 器件为可擦除器件（同 EPROM 一样）。本质上，这 3 种器件都是可编程逻辑元件阵列。

还有其他一些 PLD 器件，如 CPLD（complex programmable logic device，复杂可编程逻辑器件）器件，FPGA（field programmable gate array，现场可编程门阵列），FPIC（field programmable interconnect，现场可编程互连）。这些类型的 PLD 器件比那些普通用于设计一个完整系统的 SPLD 要复杂得多。如果仅仅用于地址译码，那么就使用 SPLD；如果要用于实现完整系统，那么就用 CPLD 或 FPIC 进行设计。这些器件也会涉及 ASIC（application-specific integrated circuit，专用集成电路）。

组合可编程逻辑阵列

两种基本类型 PAL 中有一种是组合可编程逻辑阵列。该器件内部被构造成组合逻辑电路的可编程阵列。图 10-18 描述了由与/或门逻辑构成的 PAL16L8 的内部结构。此器件非常通用，有 10 个固定输入，2 个固定输出。另外还有 6 个引脚可编程为输入或输出，每个输出信号由一个 7 输入或门产生，该

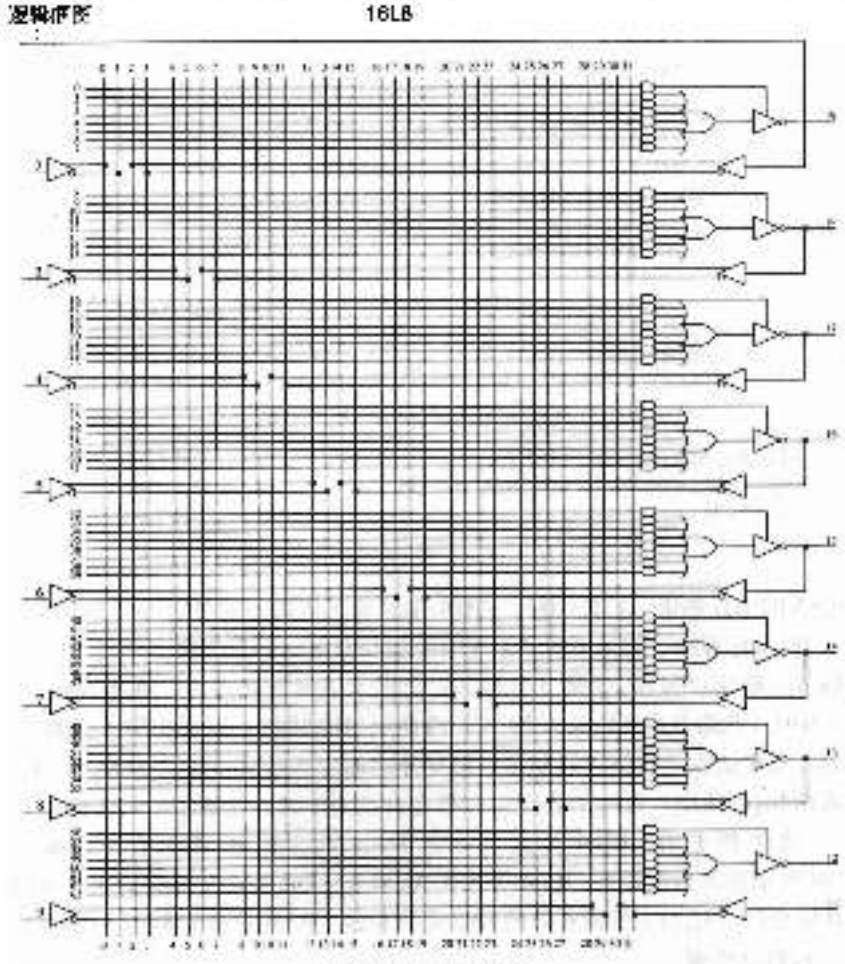


图 10-18 PAL16L8

或门的每个输入与一个与门相连。或门通过一个三态反相器输出，该反相器定义每一输出为与/或非（AND/NOR）功能。最初，所有的熔丝连接所有的垂直/水平线，如图 10-18 所示。编程是通过熔断熔丝而将不同的输入连接到或门阵列上实现的。线与功能在每条输入线上完成，它允许最多 16 个输入的乘积项。使用 PAL16L8 的逻辑表达式可有最多 7 个乘积项，每个乘积项最多有 16 个输入经或非门后产生输出表达式。该器件是存储器地址译码器的理想器件，一方面是由于它的结构，另一方面是由于其输出是低有效。

幸运的是，我们不必像这种器件刚出现时那样选熔丝号数来编程，现在，是通过使用一个软件包，如 PALASM——PAL 汇编程序来编程 PAL。最近，用 **HDL（hardware description Language，硬件描述语言）**或 **VHDL（Verilog HDL）**进行 PLD 设计。VHDL 语言和它的语法现在已成为 PLD 器件编程的工业标准。例 10-5 给出了一个程序，对图 10-17 中译码过的同一存储器区进行译码。注意此程序使用一个文本编辑器来开发，如微软 DOS 7.1 XP 版本中的 EDIT，或 Windows 中的 NotePad。此程序还可用 PALASM 软件包或任何其他 PAL 汇编程序所用的编辑器来开发。各种编辑器都用来简化引脚定义工作，但我们相信使用 EDIT 更容易一些，如下所示。

例 10-5

```
-- 图10-17中译码器的VHDL代码

library ieee;
use ieee.std_logic_1164.all;

entity DECODER_10_17 is

port (
       A19, A18, A17, MIO: in STD_LOGIC;
       ROM, RAM, AX19: out STD_LOGIC
);

end;

architecture V1 of DECODER_10_17 is

begin

       ROM  <= A19 or A18 or A17 or MIO;
       RAM  <= not (A18 and A17 and (not MIO));
       AX19 <= not A19;

end V1;
```

例 10-5 中的 VHDL 代码第一行是注释，VHDL 的注释是以两个减号（--）开始的。库以及它的使用声明指定了 IEEE 标准库。实体声明是给 VHDL 模块命名，在这个例子中为 DECODER_10_17。端口声明定义输入、输出以及输入/输出引脚，它们用于逻辑表达式中，并出现在开始的区域中。A_{19}、A_{18}、A_{17}、MIO（此信号不能定义为 IO/$\overline{M}$，因此用 MIO 代替）定义为输入引脚，ROM 和 RAM 定义为输出引脚，与存储器件的$\overline{CS}$引脚相连。结构体声明仅涉及这个设计的版本（V_1）。最后，一些设计的等式放在 begin 块中。每一个输出引脚都有自己的等式。关键字 not 用于逻辑非，and 用于逻辑与操作。在这个例子中，ROM 等式只有在 A_{19}、A_{18}、A_{17} 和 MIO 全为逻辑 0（00000H～1FFFFH）时，ROM 引脚才为逻辑 0。RAM 等式在 A_{18} 和 A_{17} 同时为 1 且 MIO 为逻辑 0 时，RAM 引脚才为 1。被 PLD 反相后，A_{19} 与上升沿有效的 CE2 引脚相连。RAM 选择60000H～7FFFFH的地址。图 10-19 是例 10-5 的 PLD 实现。

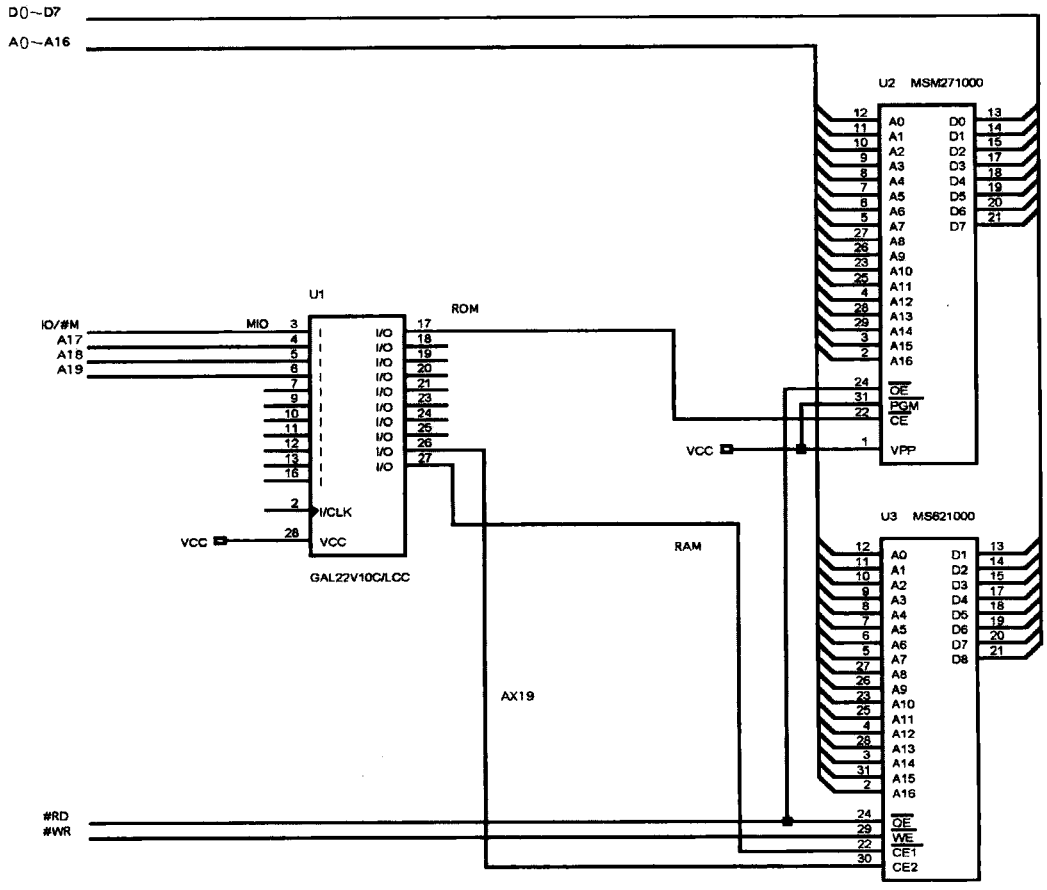


图 10-19 使用可编程逻辑器件的 RAM 和 ROM 接口

10.3 8088 和 80188（8 位）存储器接口

对于具有 8 位数据总线的 8088 和 80188，具有 16 位数据总线的 8086、80186、80286 和 80386SX，具有 32 位数据总线的 80386DX 和 80486，以及具有 64 位数据总线的 Pentium ～ Core2，本章中有单独的章节讨论它们的存储器接口。提供不同的章节是因为包含不同数据总线宽度的微处理器用于寻址存储器的方法稍有不同。希望在 16 位、32 位和 64 位存储器接口方面拓展技术的硬件工程师或技术人员必须学习所有章节。本节比有关 16 位和 32 位存储器接口的章节更完整，因为后者只介绍 8088/80188 中未涉及的内容。

本节中，我们讨论与 RAM 和 ROM 的存储器接口，并解释奇偶校验，它在许多基于微处理器的计算机系统中仍然很常见。我们还简短提及存储系统设计人员现在用到的错误校正方案。

10.3.1 基本的 8088/80188 存储器接口

8088 和 80188 微处理器有 8 位数据总线，它们是与现在普通的 8 位存储器件相连的理想的微处理器。8 位存储器容量使得 8088，尤其是 80188 成为一个理想的简单控制器。然而，为使 8088/80188 与存储器一起正确地运行，存储系统必须对地址译码，以选择存储器件。它还必须使用 8088/80188 提供的$\overline{RD}$、$\overline{WR}$和 IO/$\overline{M}$ 控制信号，来控制存储系统。

在本节中使用了最小模式配置，对存储器接口来说，在本质上它与最大模式系统相同。二者的主要区别在于，在最大模式中，IO/$\overline{M}$ 信号与$\overline{RD}$组合，产生$\overline{MRDC}$信号；IO/$\overline{M}$ 信号与$\overline{WR}$组合，产生

$\overline{MWTC}$信号，而最大模式中这些控制信号是在 8288 总线控制器内产生的。在最小模式中，存储器将 8088 或 80188 看作一个具有 20 条地址线（A_{19}～A_0）、8 条数据总线（AD_7～AD_0）及控制信号 IO/$\overline{M}$、$\overline{RD}$和$\overline{WR}$的器件。

将 EPROM 与 8088 接口

本节与 10.2 节在译码器方面非常相似。惟一区别在于，本节我们讨论等待状态和允许译码器的 IO/$\overline{M}$ 信号的使用。

图 10-20 描述了一个 8088 和 80188 微处理器是如何与 3 个 27256 EPROM（32K×8）存储器件相连的。27256 比 27128 多一个地址输入（A_{15}），存储容量是 27128 的两倍。图中的 74HCT138 译码器件译码 3 个 32K×8 存储体，即总共 96K×8 位的 8088/80188 物理地址空间。

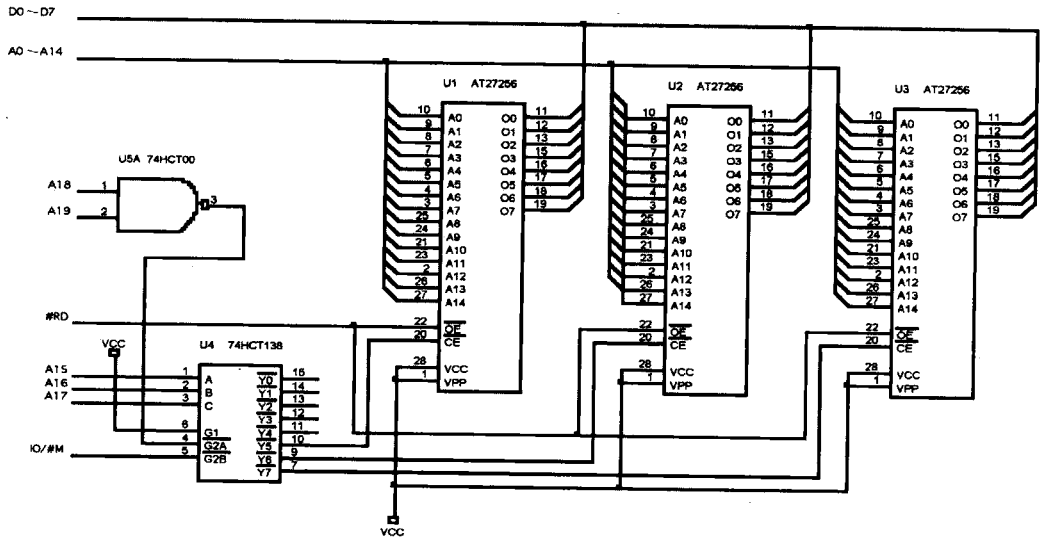


图 10-20 与 8088 微处理器接口的 3 个 27256 EPROM

译码器（74LS138）的连接与所期望的稍有差别，这是因为这种较低速度 EPROM 的存储器存取时间为 450ns。回忆第 9 章，当 8088 工作在 5MHz 时钟下时，它允许存储器在 460ns 的时间内存取数据。由于译码器增加了时间延迟（12ns），存储器在 460ns 内完成操作是不可能的。为解决这个问题，必须增加一个与非门来产生一个信号以使能译码器，并用该信号触发产生等待状态，这在第 9 章中讨论过（注意，80188 可以在内部插入 0～15 个等待状态而不需要任何额外的外部硬件，故不需要这个与非门）。每次插入一个等待状态就可访问该段的存储器，8088 使 EPROM 有 660ns 的时间去存取数据。回想一下，一个额外的等待状态给存取时间增加了 200ns（1 个时钟）。660ns 时间对于 450ns 的存储器件存取数据是足够了，甚至算上译码器和任何加在数据总线上的缓冲器所引入的延迟也足够了。

在这个系统中，为 C0000H～FFFFFH 的存储空间插入了等待状态。如果出现问题，则译码器的 3 个输出会加上一个 3 输入或门，为这个系统的实际地址（E8000H～FFFFFH）产生一个等待信号。

注意，为存储器地址范围选择了一个译码器，该范围从地址 E8000H 开始并延续到地址 FFFFFH（这是存储器的上半部 96KB）。这段存储器是一个 EPROM，因为 FFFF0H 是 8088 在硬件复位后开始执行指令的位置，所以常称地址 FFFF0H 为**冷启动地址（cold start location）**。存储在此段存储器的软件在单元 FFFF0H 中包含一个 JMP 指令，以跳到 E8000H 单元，这样余下的程序就可执行了。在这个电路中，U_1 被译码到 E8000H～EFFFFH，U_2 被译码到 F0000H～F7FFFH，U_3 被译码到 E8000H～EFFFFH。

将 RAM 与 8088 接口

RAM 接口比 EPROM 接口稍容易一些，因为大多数 RAM 存储器件不需要等待状态。RAM 理想的

存储器段在最底部，该段包含中断向量。中断向量（将在第 12 章更详细地讨论）经常被软件包所修改，因此这段存储器用 RAM 是相当重要的。

在图 10-21 中，16 个 62256 32K × 8 静态 RAM 与 8088 接口，从存储单元 00000H 开始。此电路板用了 2 个译码器来选择 16 个不同的 RAM 存储器件，第 3 个译码器用来选择其他译码器，使之选择适当的存储器段。16 个 32K 的 RAM 覆盖存储器从 00000H 单元到 7FFFFH 单元，总的容量为 512KB。

此电路中的第 1 个译码器（U_4）选择其他 2 个译码器，以 00 开始的地址选择译码器 U_3，以 01 开

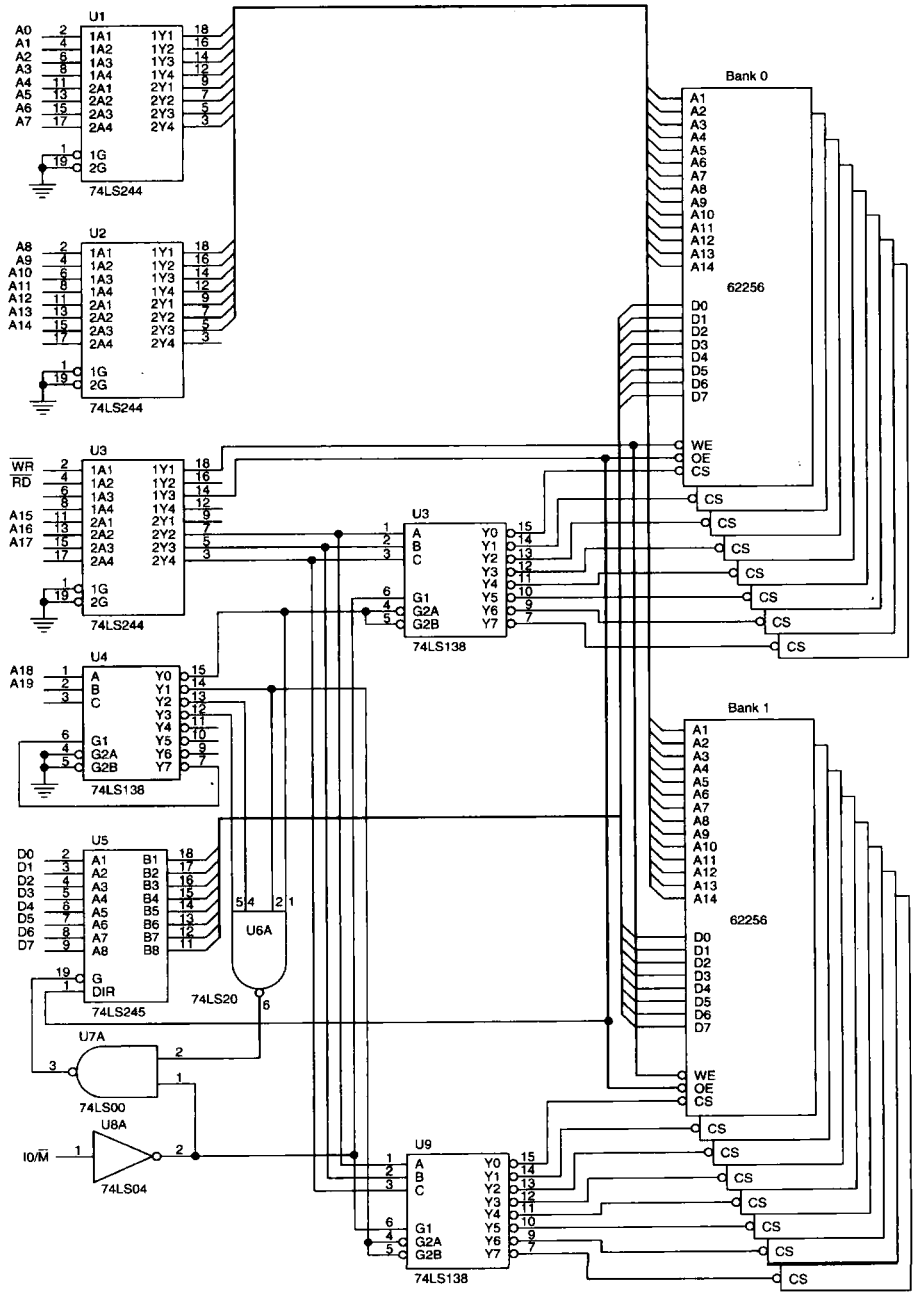


图 10-21　使用 16 个 62256 SRAM 的一个 512KB 静态存储系统

始的地址选择译码器 U_9。注意，译码器 U_4 的输出保留了其余的引脚以备将来扩展。这些引脚允许更多的 256K×8RAM 块，总共可扩展 1M×8 的 RAM，只需简单地增加 RAM 及另外的二级译码器即可。

从图 10-21 中的电路还应注意到，所有接到这个存储器段的地址输入都经过了缓冲，正如数据总线及控制信号$\overline{RD}$和$\overline{WR}$一样。当许多器件出现在一块板或一个系统上时，缓冲是非常重要的。假设有另外 3 块这样的板被插入一个系统中，每块板上没有缓冲，则在系统地址、数据和控制总线上的负载太大，系统将不能正常工作（过多的负载导致逻辑 0 输出上升到超过了系统所允许的最大值 0.8V）。缓冲器通常用于存储器在将来可能增加的情况，如果存储器容量再也不会增加，则缓冲器可以不用。

10.3.2 与快闪存储器接口

快闪存储器（EEPROM）越来越普遍地用于存储视频卡的初始化信息，也用于在 PC 中存储系统 BIOS。快闪存储器还应用于 MP3 播放器和 USB 笔驱动，除此之外还用于许多其他场合，如存储那些只会偶尔修改的信息。

快闪存储器和 SRAM 间的惟一区别是快闪存储器需要 12V 编程电压来擦除数据和写入新的数据。12V 电压可以通过电源得到，或从一个用于快闪存储器的 5V～12V 转换器得到。最新的快闪存储器可以在 5V 甚至 3.3V 信号下擦除，而不需要转换。

EEPROM 既可用作并行接口也可以用作串行接口的存储设备。然而串行设备太小，不适合存储容量的扩展，但可用作 I/O 设备，类似于快闪存驱动器一样来存储信息。本节详细介绍了这两种存储器类型。

图 10-22 描述了与 8088 微处理器接口的 28F400 Intel 快闪存储器。28F400 既可用作 512K×8 存储器件，又可用作 256K×16 存储器件。由于这里它与 8088 接口，所以它的配置是 512K×8。注意，此器件的控制线与 SRAM 的$\overline{CE}$、$\overline{OE}$和$\overline{WE}$一样 。惟一新的引脚是 V_{pp}，它与 12V 相连，用于擦除和编程；另一个是$\overline{PWD}$，逻辑 0 时选择功率下降模式，也可用于编程；还有$\overline{BYTE}$，它选择字节（为 0 时）或字（为 1 时）操作。注意，当工作于字节模式时，引脚 DQ_{15}的功用是作为最低有效地址输入位。与 SRAM 的另一区别是完成一次写操作所需的时间长短不同。SRAM 可以在少到 10ns 的时间内完成一次写操作，但快闪存储器需要大约 0.4s 来擦除一个字节。有关快闪存储器的编程将在第 11 章里与 I/O 设备

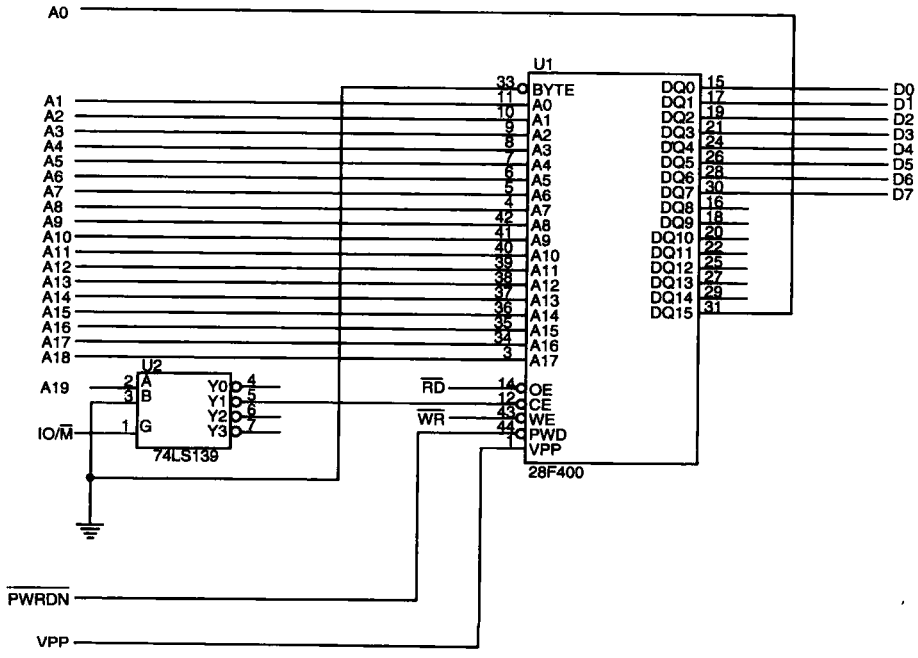


图 10-22 与 8088 微处理器接口的 28F400 快闪存储器

一起讨论。快闪存储器具有内部寄存器，这些内部寄存器通过使用 I/O 技术被编程，这在后面将讨论。本章重点放在与微处理器的接口上。

注意，图 10-22 中选择的译码器是 74LS139，因为像快闪存储器这么大容量的存储器只需要一个简单的译码器就够了。译码器使用地址线 A_{16} 和 IO/M 作为输入。A_{16} 信号选择快闪存储器作为从 8000DH 单元到 FFFFFH 单元的存储区，IO/M 用来使能译码器。

前面已经提到，多数新型的快闪存储器设备采用的是串行接口，因为采用串行方式可以减少集成电路的引脚数和面积，从而降低集成电路的成本。目前串行快闪存储器容量可达 4GB，而且在速度和可擦除次数上已与并行的闪存设备相当。实际上，大部分闪存正常工作只需 5V 或 3.3V 电压，无需更高的编程电压。擦除次数高达 1 000 000 次，存储使用时间可达 200 年。

图 10-23 给出了一个小型串行闪存设备（256K 器件，32K×8）的接口电路图。三个地址线引脚采用硬连接，允许不止一个设备连到串行总线上。图中 U_1 连到地址 001，U_2 连到地址 000。还有一个串行数据连接用得上拉电阻在图中未画出，上拉电阻可置于微处理器内或者需要外部连接，怎样使用这就要取决于微处理器以及与存储器相连的接口了。

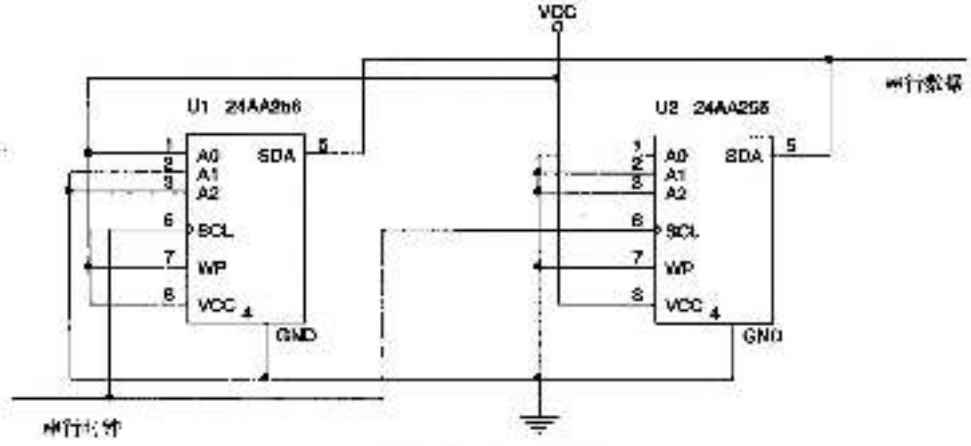


图 10-23 串行 EEPROM 接口

此存储器接口电路中还有两条单独的信号线，一个是串行时钟线（SCL），另一个是双向串行数据线（SDA）。由于时钟频率不超过 400kHz，所以这种存储器不能取代系统的主存，但对于音乐或其他的低速数据传输还是足够快的。关于串行接口将在第 11 章叙述。

图 10-24 给出了串行 EEPROM 的基本串行数据格式。串行数据的第一个字节包含地址（A_2、A_1、

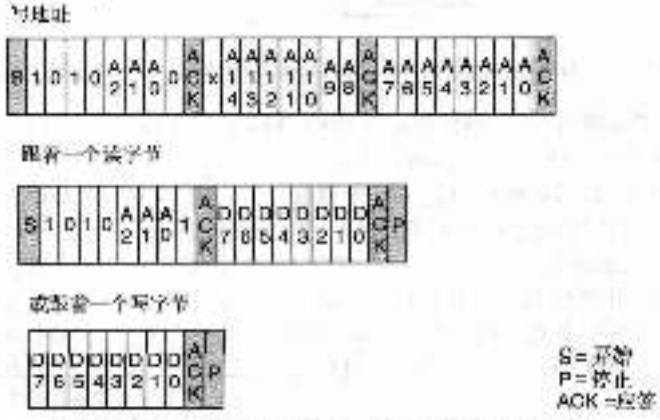


图 10-24 对串行 EEPROM 一次读或写的数据信号

A_2 引脚）和设备标识码 1010（1010 表示 EEPROM），其他串行设备则有不同的设备标识码。接下来的几个字节就是存储器地址和数据部分了。

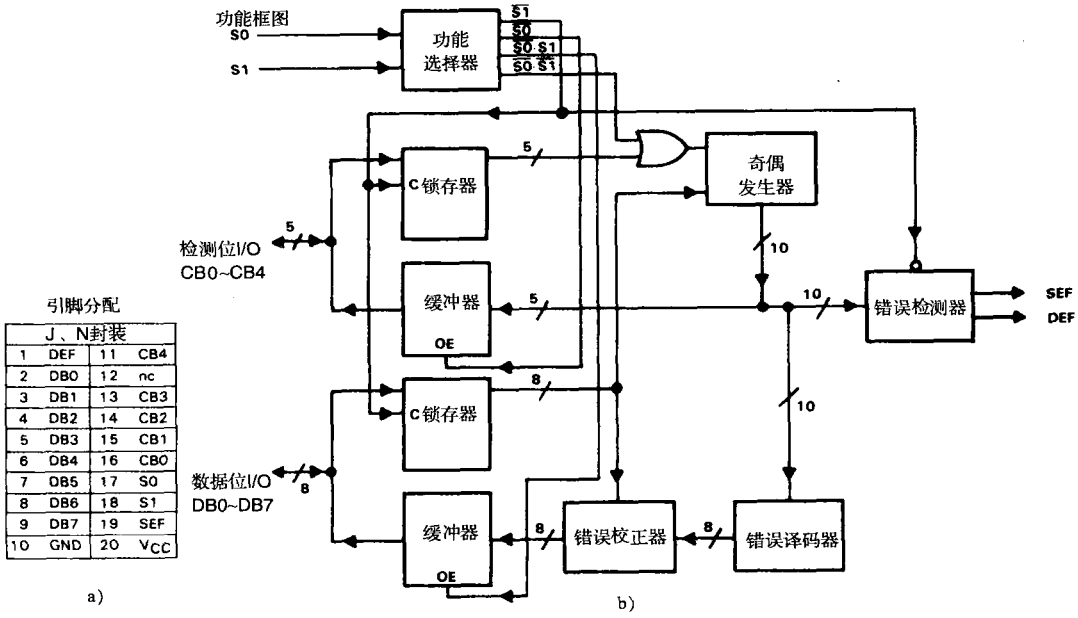
10.3.3 错误校正

错误校正方案已出现了很久，但集成电路制造商也只是在最近才开始生产错误校正电路。一个这样的电路是 74LS636，它是一个 8 位错误检测和校正电路，它可校正存储器的任意一个 1 位读错误并可检测到任意一个 2 位错误，叫做 SECDED（single error correction/double error detecting check，单错校正/双错检测）。此器件用于高端计算机系统中，因为实现一个具有错误校正功能的系统的成本较高。

最新的计算机系统现在正开始使用具有 ECC（error-correction code，错误校正码）的 DDR 存储器。对于可能出现在这些存储器件中的错误，其校正方案与本书中讨论的方案一样。

74LS636 通过给每个字节数据存储 5 个奇偶校验位来校正错误。这确实增加了所需存储器的数量，但也提供了 1 位错误的自动校正功能。如果错误超过 2 位，则此电路将检测不到。幸运的是这种情况很少，而且校正超过 1 位错误需做的额外努力代价非常高，不值得这样做。一旦一个存储器件工作完全不正常，其数据位或者是全 1，或者是全 0。在这种情况下，电路会显示处理器有多位错误的迹象。

图 10-25 描述了 74LS636 的引脚。注意它有 8 个数据 I/O 引脚，5 个检查位 I/O 引脚，2 个控制输



引脚分配

J、N封装			
1	DEF	11	CB4
2	DB0	12	nc
3	DB1	13	CB3
4	DB2	14	CB2
5	DB3	15	CB1
6	DB4	16	CB0
7	DB5	17	S0
8	DB6	18	S1
9	DB7	19	SEF
10	GND	20	V_{CC}

a) b)

图 10-25 a）74LS636 的引脚图；b）74LS636 的框图（德州仪器公司提供）

入（S_0 和 S_1）和 2 个错误输出：1 位错误标志（SEF）和 2 位错误标志（DEF）。控制输入用于选择所执行的操作类型，在表 10-1 中列出了它们的真值表。

当检测到一个 1 位错误，74LS636 就进入一个错误校正周期：它产生一个等待状态使 S_0 和 S_1 置为 01，然后在错误校正后进行一次读操作。

图 10-26 描述了用 74LS636 校正 1 位错误，并通过 NMI 引脚对 2 位错误进行中断处理的电路。为叙述简单，仅描述一个 2K × 8RAM 和一个存储 5 位检测码的另一个 2K × 8RAM。

表 10-1 74LS636 的控制位 S_0 和 S_1

S_1	S_0	功 能	SEF	DEF
0	0	写检测字	0	0
0	1	校正数据字	*	*
1	0	读数据	0	0
1	1	锁存数据	*	*

* 这些电平由错误类型确定。

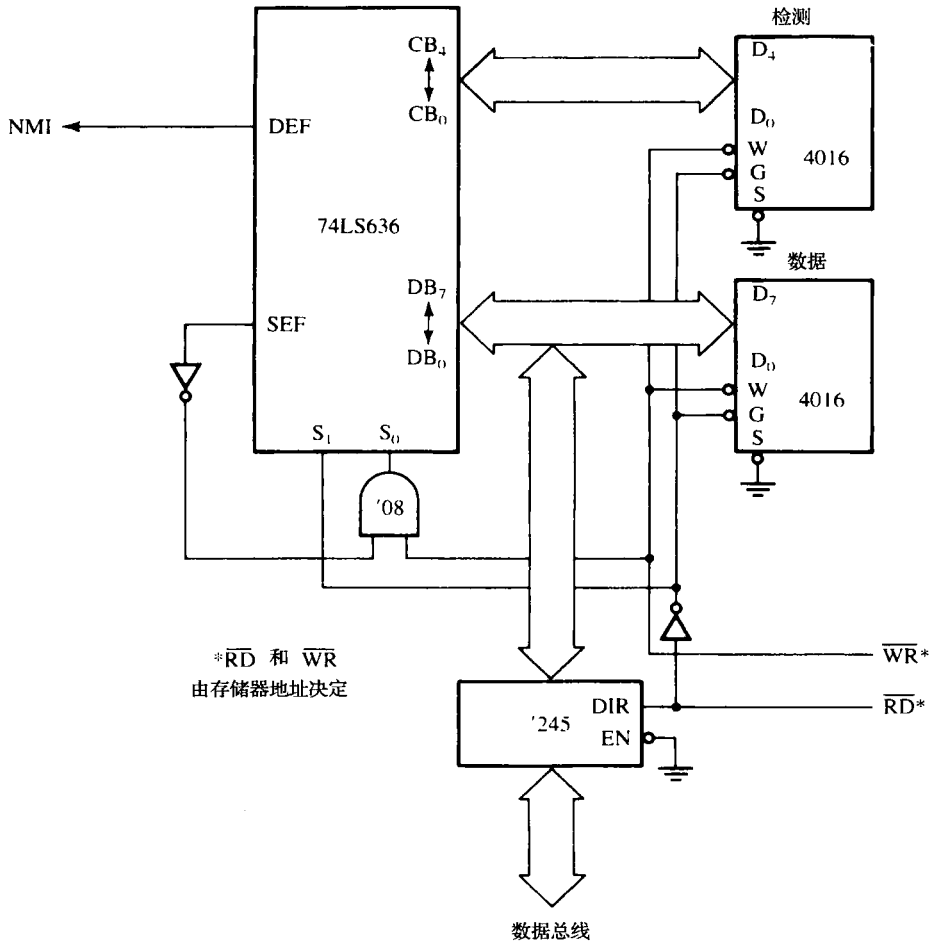


图 10-26 使用 74LS636 的错误检测和校正电路

这个存储器件的引脚与前面的例子不同。注意这里 S 或$\overline{CS}$引脚接地，数据总线缓冲器控制数据流向系统总线。要想在$\overline{RD}$选通变低之前从存储器中读取数据，这是必须的。

在$\overline{RD}$有效之后的下一个时钟的下降沿，74LS636 检查 1 位错误标志（SEF），以确定是否出现了一个错误。如果是，则错误校正周期使得检测到的 1 个错误被校正。如果出现 2 个错误，则 2 位错误标志（DEF）输出产生一个中断请求，该输出与微处理器的 NMI 引脚相连。

现代的 DDR 错误校正存储器（ECC）在底板上实际没有那些检错纠错的逻辑电路。因为 Pentium 微处理器合并了进行检错/纠错的逻辑电路，使内存可以存储那些存储 ECC 编码所需的额外的 8 位数据。ECC 存储器是 72 位宽，用 8 个附加位存储错误校正码。如果发生错误，则微处理器运行校正循环以校正错误。一些像三星存储器之类的存储器件也提供内部错误检测。三星错误校正码使用 3 个字节检测存储器的每 256 个字节，这就使其变得更高效了。三星 ECC 算法的附加信息可以从三星网站上获得。

10.4 8086、80186、80286 和 80386SX（16 位）存储器接口

8086、80186、80286 和 80386SX 微处理器与 8088/80188 比较，有以下 3 个不同之处：（1）数据总线为 16 位宽，而 8088 为 8 位宽；（2）8088 的 IO/$\overline{M}$ 引脚换成 M/$\overline{IO}$引脚；（3）有一个新的称为总线高使能

($\overline{BHE}$) 的控制信号。地址位 A_0 或$\overline{BLE}$的使用方式也不同（因为本节以 10.3 节的知识为基础，所以首先阅读前面的章节是极为重要的）。在 8086/80186 和 80286/80386SX 之间存在少许其他差别。80286/80386SX 包含 24 位地址总线（A_{23}～A_0），而 8086/80186 包含 20 位地址总线（A_{19}～A_0）。8086/80186 包含 M/$\overline{IO}$信号，而 80286 系统和 80386SX 微处理器包含控制信号$\overline{MRDC}$和$\overline{MWTC}$，而不是$\overline{RD}$和$\overline{WR}$。

16 位总线控制

8086、80186、80286 和 80386SX 的数据总线宽度是 8088/80188 的 2 倍。较宽的数据总线给我们带来一些以前不曾遇到过的独特的问题。8086、80186、80286 和 80386SX 必须能够将数据写入任何 16 位或 8 位存储单元。这意味着 16 位数据总线必须分为 2 个独立的 8 位宽的段（**存储体**），以便微处理器可以在半个区域（8 位）或整个区域（16 位）中写入数据。图 10-27 描述了存储器的 2 个存储体：一个存储体（**low bank，低位存储体**）包含所有地址为偶数的存储单元；另一个存储体（**high bank，高位存储体**）包含所有地址为奇数的存储单元。

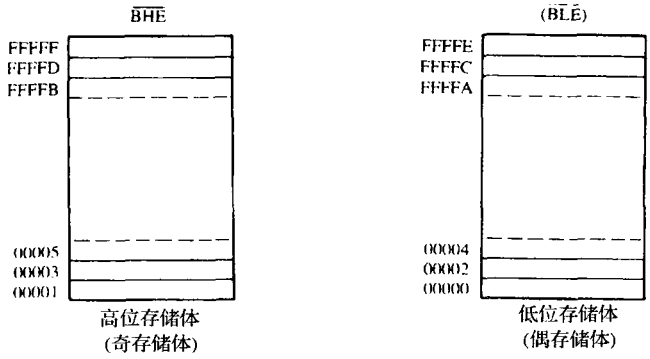


图 10-27 8086、80186、80286 和 80386SX 微处理器的高（奇）和低（偶）8 位存储体

注：在 80386SX 中 A_0 被标为$\overline{BLE}$（总线低使能）。

8086、80186、80286 和 80386SX 用$\overline{BHE}$信号（高位存储体）和 A_0 地址位或$\overline{BLE}$（总线低位使能）来选择 1 个或 2 个存储体进行数据传送。表 10-2 描述了这两个引脚上的电平和所选择的存储体。

表 10-2 使用 BHE 和 BLE（A_0）选择存储体

$\overline{BHE}$	$\overline{BLE}$	功 能
0	0	允许 2 个存储体进行 16 位数据传送
0	1	允许高位存储体进行 8 位数据传送
1	0	允许低位存储体进行 8 位数据传送
1	1	2 个存储体都未选中

存储体的选择以两种方式完成：（1）产生一个独立的写信号来选择对每个存储体的写操作；（2）每个存储体使用独立的译码器。仔细比较一下，第 1 项技术是迄今为止 8086、80186、80286 和 80386SX 微处理器与存储器接口的成本最低的方法。第二项技术仅用于那些必须达到电源最高效应用的系统中。

独立的存储体译码器

使用独立的存储体译码器为 8086、80186、80286 和 80386SX 微处理器译码存储器地址，是一种效率很低的方式。有时也采用这种方法，但大多数情况下难以理解为什么使用这种方法。其中一个原因也许是为了节约能耗，因为只有被选中的 1 个或几个存储体才允许进行数据传送。而后面将讨论的独立读写信号方式却不总是这样的。

图 10-28 描述了 2 个 74LS138 译码器，用于为 80386SX（24 位地址）选择 64K RAM 存储器件。这里，译码器 U_2 将$\overline{BHE}$引脚与其$\overline{G2A}$引脚相连，译码器 U_3 将$\overline{BLE}$（A_0）引脚与其$\overline{G2A}$输入相连。因为译码器只有在其所有允许输入有效时才会被激活，所以译码器 U_3 只用于 16 位操作或低位存储体的 8 位操作，译码器 U_2 用于 16 位操作或对高位存储体的 8 位操作。这 2 个译码器及其控制的 16 个 64KB RAM，代表

80386SX 存储系统的 1MB 范围。译码器 U_1 允许 U_2 和 U_3 寻址的存储器范围为 000000H ～ 0FFFFFH。

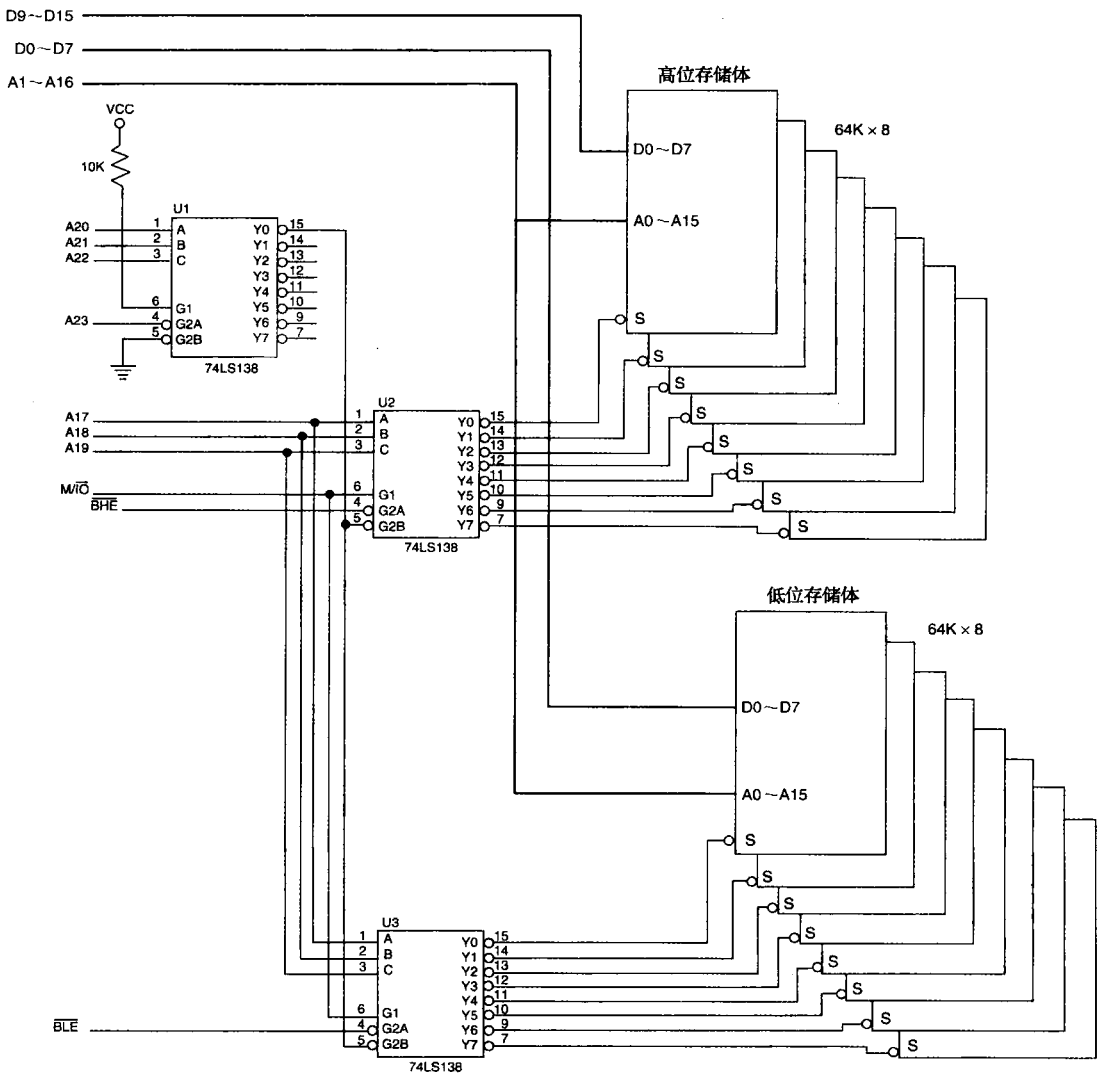


图 10-28　独立的存储体译码器

注意，在图 10-28 中地址线 A_0 未与存储器相连，因为它在 80386SX 中不存在。还应注意，地址总线位 A_1 与存储器地址输入 A_0 相连，地址总线位 A_2 与存储器地址输入 A_1 相连，其他连接依此类推。原因是 8086/80186 的 A_0（或 80286/80386SX 的$\overline{BLE}$）已与译码器 U_2 相连，不必再连到存储器上。若 A_0 或$\overline{BLE}$与存储器的地址线 A_0 相连，则每个存储体将每隔一个存储单元被使用。这意味着如果 A_0 或$\overline{BLE}$与存储器的 A_0 相连，那么将会浪费掉存储器的一半空间。

独立的存储体写选通

处理存储体选择的最有效方法是为每个存储体产生一个独立的写选通。这项技术只需 1 个译码器来选择一个 16 位的存储器，既节约成本，又减少了系统中器件的数量。

为什么不为每个存储体也产生独立的读选通呢？这通常是不必要的，因为 8086、80186、80286 和 80386SX 微处理器每次只读 1 字节数据，在任何给定的时间内只需要数据总线的一半。

若在一次读操作中16位数据一直存在于数据总线上，则微处理器忽略它不需要的8位数据，而不会有任何冲突或特别的问题。

图10-29描述了为存储器产生独立的8086写选通的方法。这里，74LS32或门组合A_0和$\overline{WR}$产生低位存储体选择信号（$\overline{LWR}$），组合$\overline{BHE}$和$\overline{WR}$产生高位存储体选择信号（$\overline{HWR}$）。80286/80386SX的写选通是使用$\overline{MWTC}$信号而不是$\overline{WR}$来产生的。

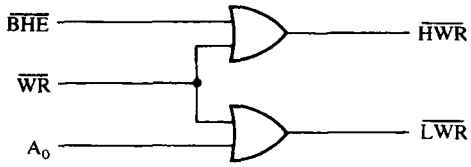


图10-29 存储体写选择输入信号：$\overline{HWR}$（高位存储体写）和$\overline{LWR}$（低位存储体写）

使用独立写选通的存储系统在结构上不同于8位系统（8088）或使用独立存储体的系统。在系统中使用独立写选通的存储器被译码为16位宽的存储器。例如，假定一个存储系统具有64KB的SRAM存储器，它需要2个32KB存储器件（62256），以便可以构造一个16位宽的存储器。由于存储器为16位宽，且由另一电路产生存储体写信号，所以地址位A_0变成无关项。事实上，A_0甚至不是80386SX微处理器上的引脚。

例10-6表明一个存储在存储单元060000H～06FFFFH中的16位宽存储器是如何被译码给80286或80386微处理器的。此例中的存储器被译码时，A_0对于译码器是无关项。地址位A_1～A_{15}与存储器件的地址线A_0～A_{14}相连。一旦地址06XXXXH出现在地址总线上，译码器（GAL22V10）就使用地址线A_{23}～A_{15}选择存储器，从而使能2个存储器件。

例10-6

```
0000 0110 0000 0000 0000 0000 = 060000H
          到
0000 0110 1111 1111 1111 1111 = 06FFFFH

0000 0110 XXXX XXXX xxxx XXXX = 06XXXXH
```

图10-30描述了这个简单电路，它使用GAL22V10译码存储器并产生独立的写选通信号。GAL22V10

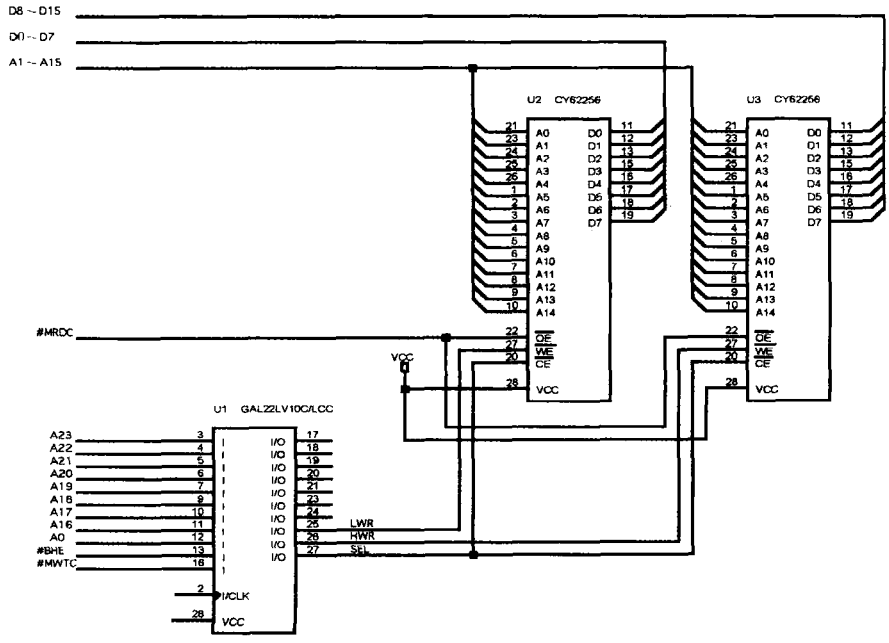


图10-30 使存储器位于地址060000H～06FFFFH的16位存储器译码器

译码器的程序如例 10-7 所示。注意，PLD 不仅选择存储器，而且还产生低位写选通和高位写选通信号。

例 10-7

```
-- 图10-30中译码器的VHDL代码

library ieee;
use ieee.std_logic_1164.all;

entity DECODER_10_30 is

port (
       A23, A22, A21, A20, A19, A18, A17, A16, A0, BHE, MWTC: in STD_LOGIC;
       SEL, LWR, HWR: out STD_LOGIC
);

end;

architecture V1 of DECODER_10_30 is

begin

       SEL <= A23 or A22 or A21 or A20 or A19 or (not A18) or (not A17) or A16;
       LWR <= A0 or MWTC;
       HWR <= BHE or MWTC;

end V1;
```

图 10-31 描述了 8086 微处理器的小型存储系统，它包含一个 EPROM 段和一个 RAM 段。这里，4 个 27128 EPROM（16K × 8）组成一个 32K × 16 位的存储器，地址范围为 F0000H ～ FFFFFH；4 个 62256（32K × 8）RAM 组成一个 64K × 16 位存储器，地址范围为 00000H ～ 1FFFFH（注意，尽管存储器是 16 位宽，它仍然按字节编号）。

此电路使用了一个 74LS139 双 2 – 4 线译码器，其中的一个译码器选择 EPROM，另一个选择 RAM。它对 16 位宽的存储器译码，而不是以前介绍的 8 位。注意，$\overline{RD}$选通与所有 EPROM 的$\overline{OE}$输入以及所有 RAM 的 $\overline{G}$ 输入引脚相连。这么做是因为即使 8086 正在读的仅仅是 8 位数据，但数据总线上其余 8 位数据的应用对 8086 的操作也不会产生影响。

$\overline{LWR}$和$\overline{HWR}$选通被接到 RAM 存储器的不同存储体上。这里，它与微处理器正在执行 16 位还是 8 位的写操作有关系。若 8086 写 16 位数据给存储器，$\overline{LWR}$和$\overline{HWR}$都变低并允许 2 个存储体的 $\overline{WE}$ 引脚。但是若 8086 执行 8 位的写操作，则只有 1 个写选通变低，只对 1 个存储体写入数据。存储体之间的差别只是表现在存储器写操作时。

注意，EPROM 译码器的信号发送给 8086 等待状态产生器，因为 EPROM 存储器通常需要一个等待状态。来自与非门的信号用于选择 EPROM 译码器部分，这样，如果 EPROM 被选中，则请求一个等待状态。

图 10-32 描述了一个与 80386SX 微处理器相连的存储系统，它使用一个 GAL22V10 作为译码器。此接口包含 256KB 的 EPROM，由 4 个 27512（64K × 8）EPROM 组成；128KB 的 SRAM，由 4 个 62256（32K × 8）SRAM 组成。

注意，在图 10-32 中，PAL 还产生存储体写信号$\overline{LWR}$和$\overline{HWR}$。从此电路可看出，在大多数情况下，需要与存储器接口的器件数目已减少到只有 1 个（PLD）。PLD 的程序清单见例 10-8。PLD 对 16 位存储器的地址进行译码，SRAM 的地址范围是 000000H ～ 01FFFFH，EPROM 的地址范围是 FC0000H ～ FFFFFFH。

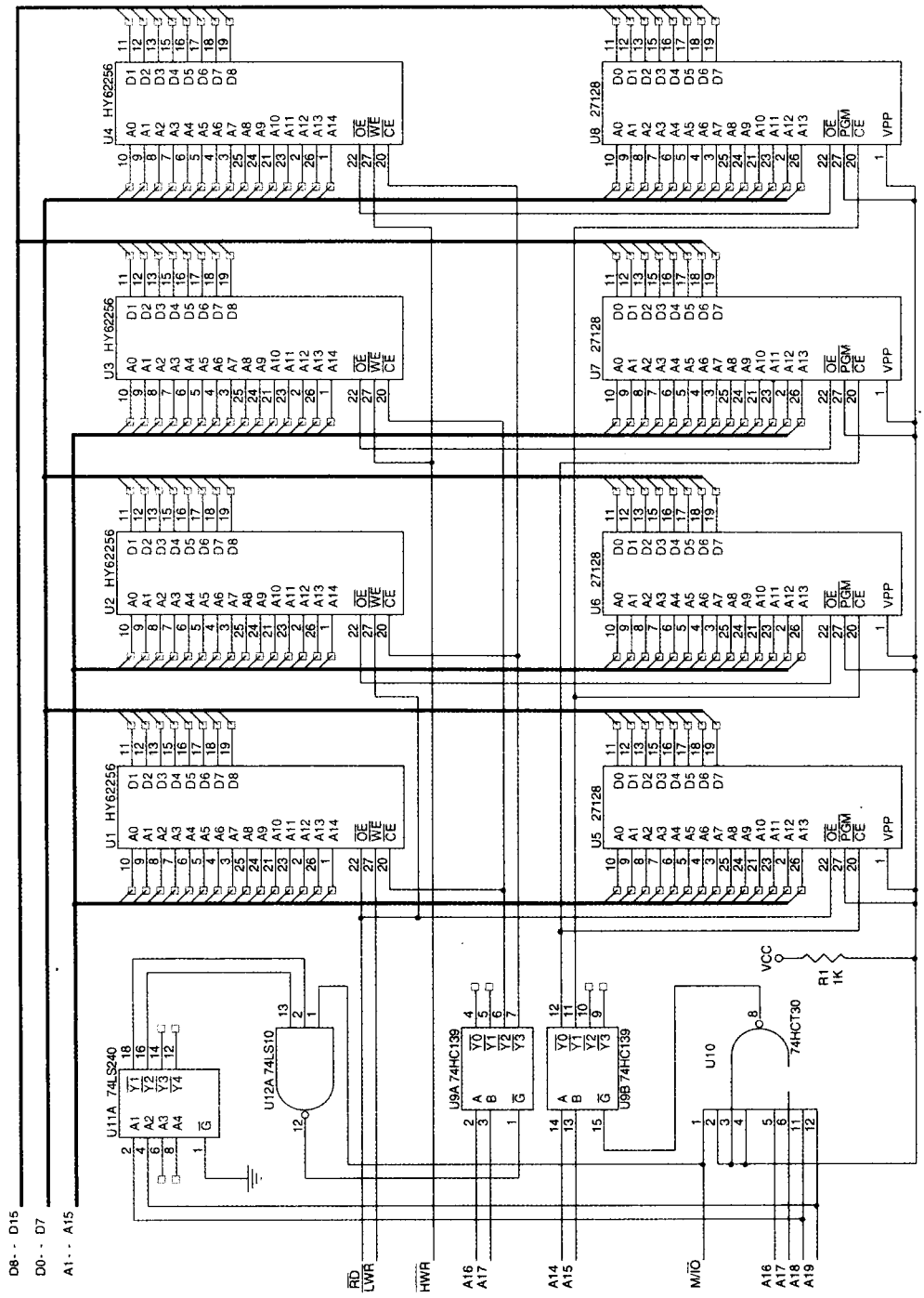


图 10-31 8086的存储器系统，包含一个 64KB EPROM 和一个 128KB SRAM

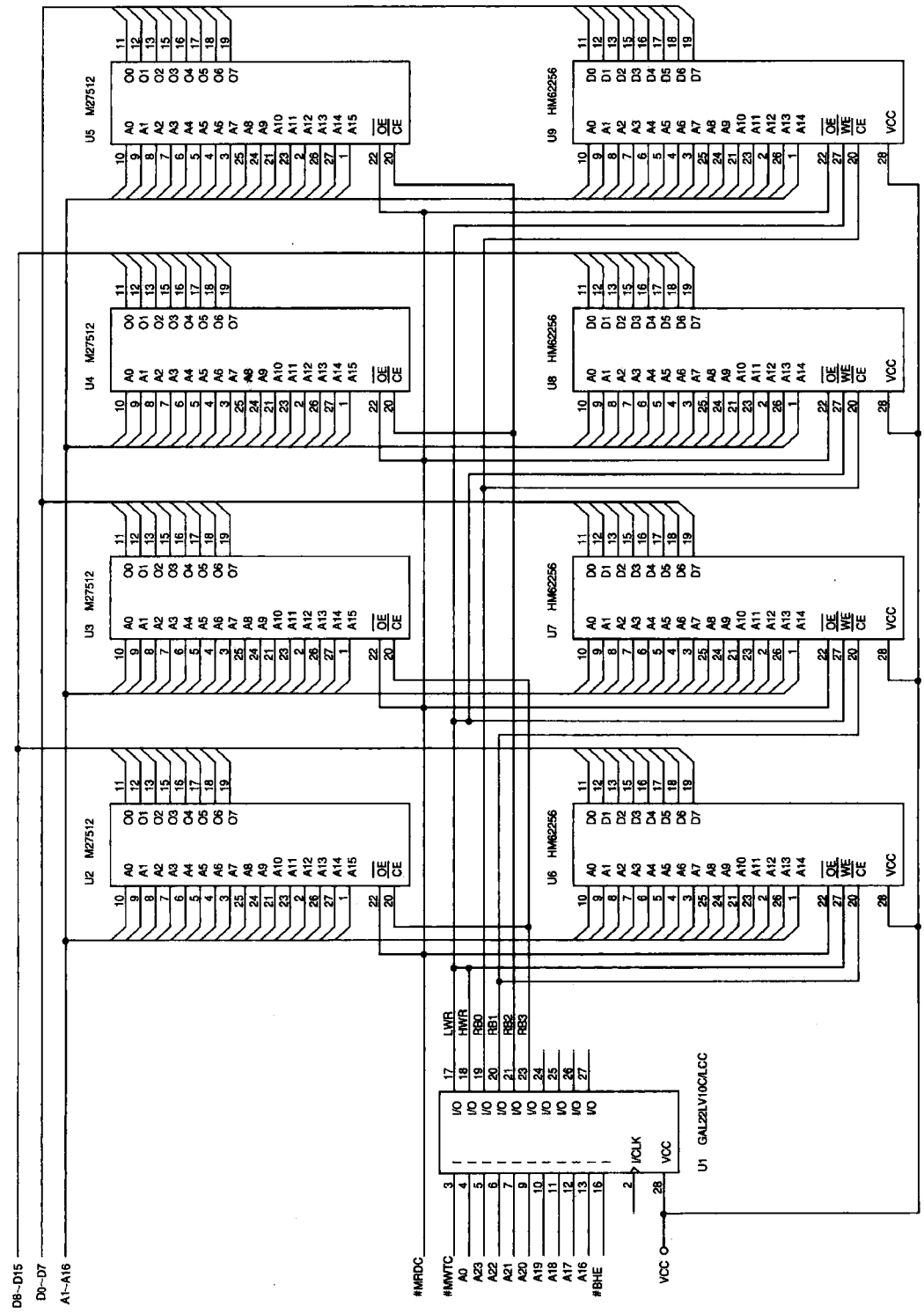


图 10-32 80386SX 的存储器系统，包含 256KB EPROM 和 128KB SRAM

例 10-8

```
-- 图10-32中译码器的VHDL代码

library ieee;
use ieee.std_logic_1164.all;

entity DECODER_10_32 is

port (
       A23, A22, A21, A20, A19, A18, A17, A16, A0, BHE, MWTC: in STD_LOGIC;
       LWR, HWR, RB0, RB1, RB2, RB3: out STD_LOGIC
);

end;

architecture V1 of DECODER_10_32 is

begin

       LWR <= A0 or MWTC;
       HWR <= BHE or MWTC;
       RB0 <= A23 or A22 or A21 or A20 or A19 or A18 or A17 or A16;
       RB1 <= A23 or A22 or A21 or A20 or A19 or A18 or A17 or not(A16));
       RB2 <= not(A23 and A22 and A21 and A20 and A19 and A18 and A17);
       RB3 <= not(A23 and A22 and A21 and A20 and A19 and A18 and not(A17));

end V1;
```

10.5 80386DX 和 80486（32 位）存储器接口

正如 8 位和 16 位存储系统一样，微处理器通过数据总线和选择独立存储体的控制信号与存储器接口。32 位存储系统与它们的区别在于微处理器有 32 位数据总线和 4 个存储体，而不是 1 个或 2 个。另一区别是 80386DX 和 80486（SX 和 DX）均包含 32 位地址总线，由于它们的地址位数目相当大，因此通常需要 PLD 而不是集成电路作为译码器。

10.5.1 存储体

图 10-33 描述了 80386DX 和 80486 微处理器的存储体。注意，这些大存储系统包含 4 个 8 位存储体，每个存储体包含最多 1GB 存储器。存储体选择由存储体选择信号$\overline{BE3}$、$\overline{BE2}$、$\overline{BE1}$和$\overline{BE0}$实现。如果传送一个 32 位数，则所有 4 个存储体都被选中；如果传送一个 16 位数，则 2 个存储体（通常是$\overline{BE3}$和$\overline{BE2}$，或$\overline{BE1}$和$\overline{BE0}$）被选中；如果传送一个 8 位数，则 1 个存储体被选中。

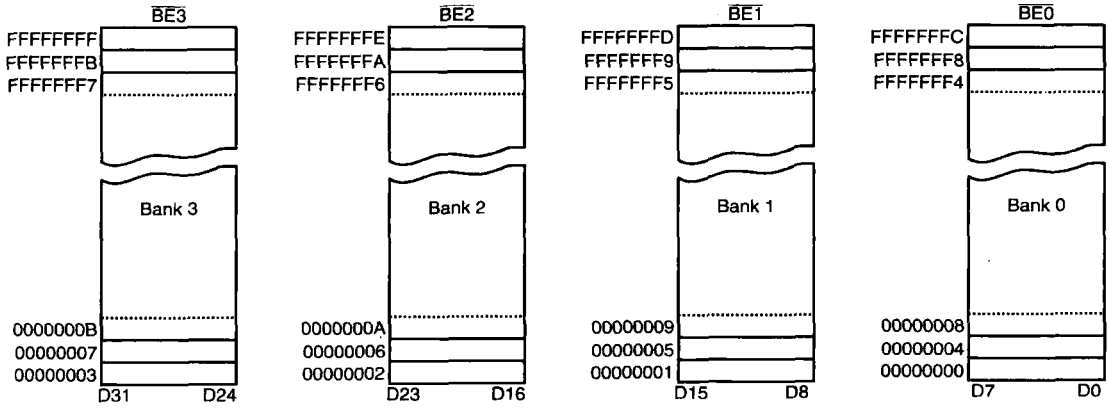


图 10-33 80386DX 和 80486 微处理器的存储器组织

与 8086/80286/80386SX 一样，80386DX 和 80486 对每个存储体需要独立的写选通信号。这些独立的写选通信号是通过使用一个简单的或门，或其他逻辑器件产生的，如图 10-34 所示。

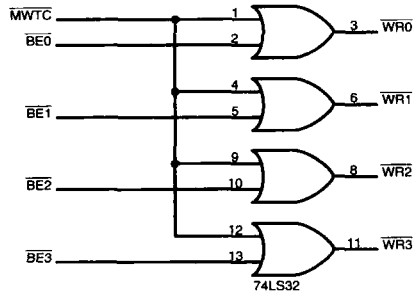


图 10-34　80386DX 和 80486 微处理器的存储体写信号

10.5.2　32 位存储器接口

从前面的讨论中可以看出，80386DX 和 80486 的存储器接口需要产生 4 个存储体写选通并译码 32 位地址。没有一个集成的译码器（如 74LS138）适合作为 80386DX 和 80486 微处理器的存储器接口。注意，当 32 位宽的存储器被译码时，地址位 A_0 和 A_1 为无关项，这 2 个地址位用在微处理器中产生存储体使能信号；而地址总线 A_2 与存储器地址线 A_0 相连，这是因为 80486 微处理器上没有 A_0 或 A_1 引脚。

图 10-35 描述了 80486 微处理器的一个 512K × 8 存储系统。该接口使用了 8 个 64K × 8SRAM 存储器件、1 个 PLD 和一个或门器件。需要或门器件是因为微处理器的地址线数目较多。此系统使 SRAM 存储器位于存储单元 02000000H ～ 0203FFFFH。PLD 器件的程序如例 10-9 所示。

例 10-9

```
library ieee;
use ieee.std_logic_1164.all;

entity DECODER_10_35 is

port (
       A30, A29, A28, A27, A26, A25, A24, A23, A22, A21, A19, BE0, BE1, BE2,
            BE3, MWTC: in STD_LOGIC;
       RB0, RB1, WR0, WR1, WR2, WR3: out STD_LOGIC
);

end;

architecture V1 of DECODER_10_35 is
begin

       WR0 <= BE0 or MWTC;
       WR1 <= BE1 or MWTC;
       WR2 <= BE2 or MWTC;
       WR3 <= BE3 or MWTC;
       RB0 <= A30 or A29 or A28 or A27 or A26 or A25 or A24 or A23 or A22
              or A 21 or A19;
       RB1 <= A30 or A29 or A28 or A27 or A26 or A25 or A24 or A23 or A22
              or A 21 or not(A19);

end V1;
```

尽管本节中没有提到，但实际上 80386DX 和 80486 微处理器以非常高的时钟频率工作，在存储器存取时通常需要等待状态。这些微处理器存取时间的计算将在第 17 章和第 18 章中讨论。接口提供了一个信号，与等待状态产生器一起使用，该信号在本节中没有描述。与这些较高速度的微处理器接口的其他器件是高速缓冲存储器（cache memory）和交叉存取存储器（interleaved memory）系统。在第 17 章讲解 80386DX 和 80486 微处理器时也会介绍这些器件。

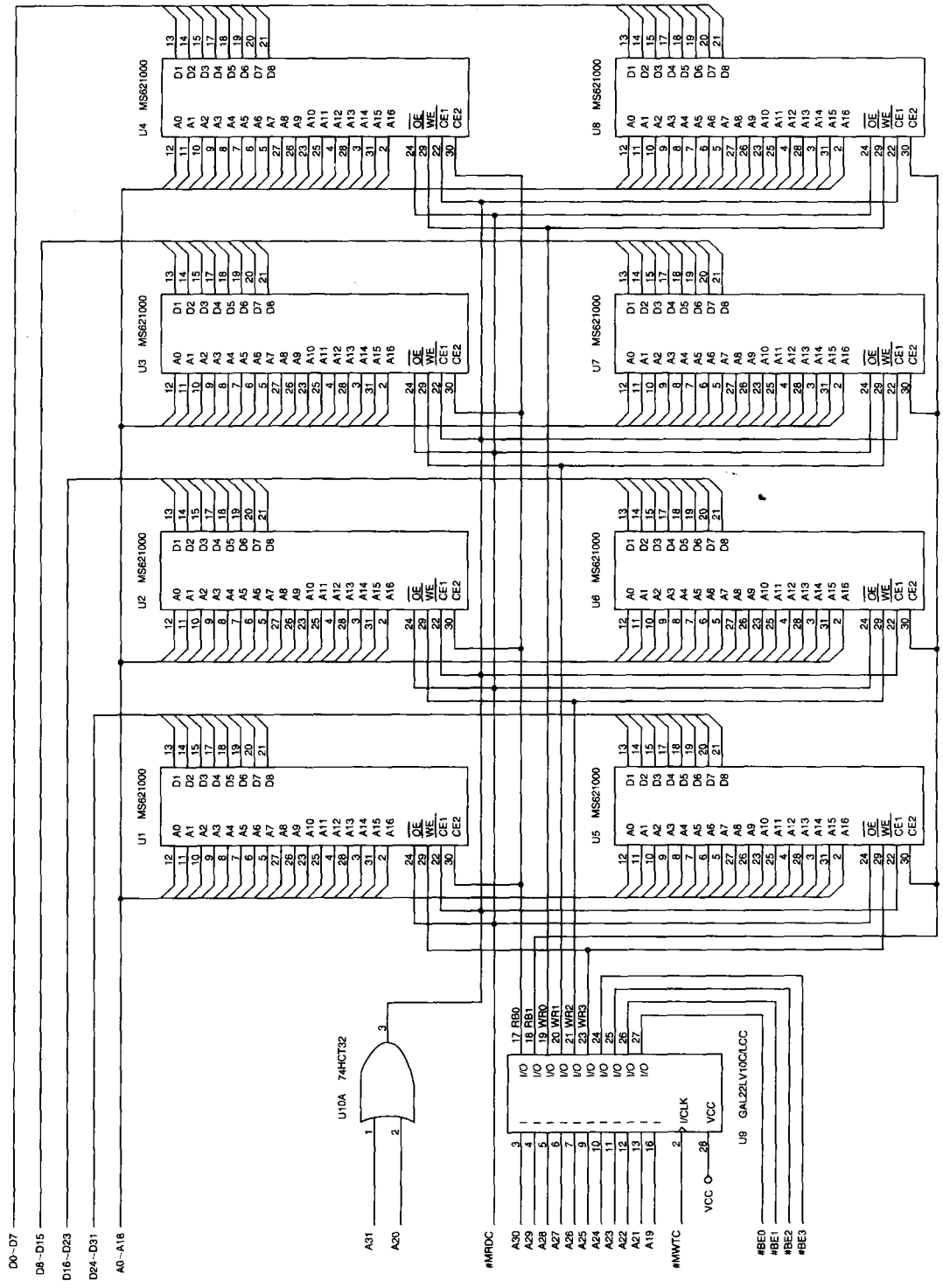


图 10-35 80486 微处理器的一个 512K×8SRAM 存储器系统

10.6 Pentium ~ Core2（64 位）存储器接口

Pentium ~ Core2 微处理器（除 Pentium 的 P24T 版本外）均具有 64 位数据总线，需要 8 个译码器（每个存储体 1 个）或 8 个独立的写信号。在大多数系统中，当微处理器与存储器接口时使用独立的写信号。图 10-36 描述了 Pentium 的存储器组织及其 8 个存储体。注意，它与 80486 几乎是相同的，只是它包含 8 个存储体而不是 4 个。

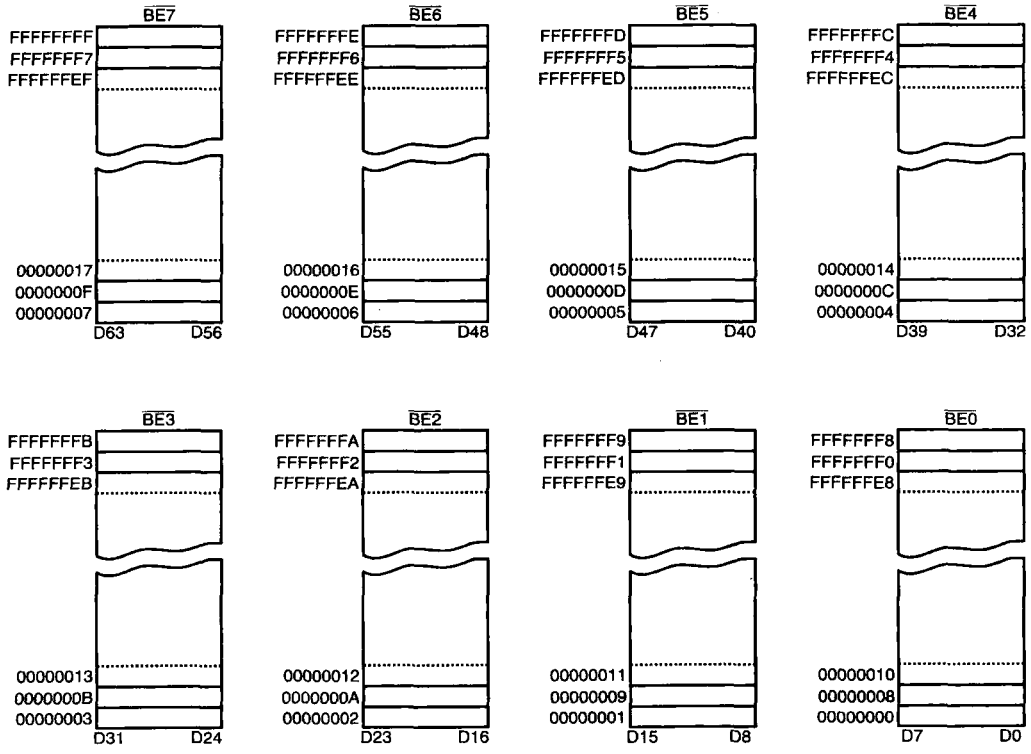


图 10-36 Pentium ~ Core2 微处理器的存储器组织

对于较早型号的 Intel 微处理器，为了达到向上的存储器兼容性，需要使用这种组织方式：将存储体允许信号和$\overline{\text{MWTC}}$信号组合后得到独立的写选通信号，这里$\overline{\text{MWTC}}$信号是由 M/$\overline{\text{IO}}$和 W/$\overline{\text{R}}$ 组合产生的。用于产生存储体写信号的电路如图 10-37 所示。我们常使用 PLD 来产生存储体的写信号。

64 位存储器接口

图 10-38 描述了一个小型的 Pentium ~ Core2 存储系统。该系统使用 1 片 PLD 译码存储器地址。该系统包含 8 个 27C4001 EPROM 存储器件（512K × 8），与 Pentium ~ Core2 接口，地址范围为 FFC00000H ~ FFFFFFFFH。存储器的总容量为 4MB，每个存储体包含 2 个存储器件。注意，Pentium Pro ~ Core2可以被配置为 36 条地址线，可允许最大 64GB 存储器。Pentium 4 和 Core2 也可以在平坦模式中被配置，最高可达 40 条地址线（Core2 只包含 36 条）。

正如例 10-10 所描述的，存储器译码与前面的例子类似，只是对 Pentium ~ Core2，最右边 3 个地址位（A_2 ~ A_0）不用。在这种情况下，译码器选择 64 位宽存储器的 2 个段，共包含 4MB 的 EPROM 存储器。

每个存储器件的 A_0 地址输入与 Pentium ~ Core2 的 A_3 地址输出相连，A_1 地址输入与 Pentium ~ Core2 的 A_4 地址输出相连。这种歪斜的地址连接一直持续到存储器的 A_{18} 地址输入与 Pentium 的 A_{22} 地址输出相连。地址位 A_{22} ~ A_{31} 由 PLD 进行译码。PLD 器件的程序如例 10-10 所示，存储单元为 FFC00000H ~ FFFFFFFFH。

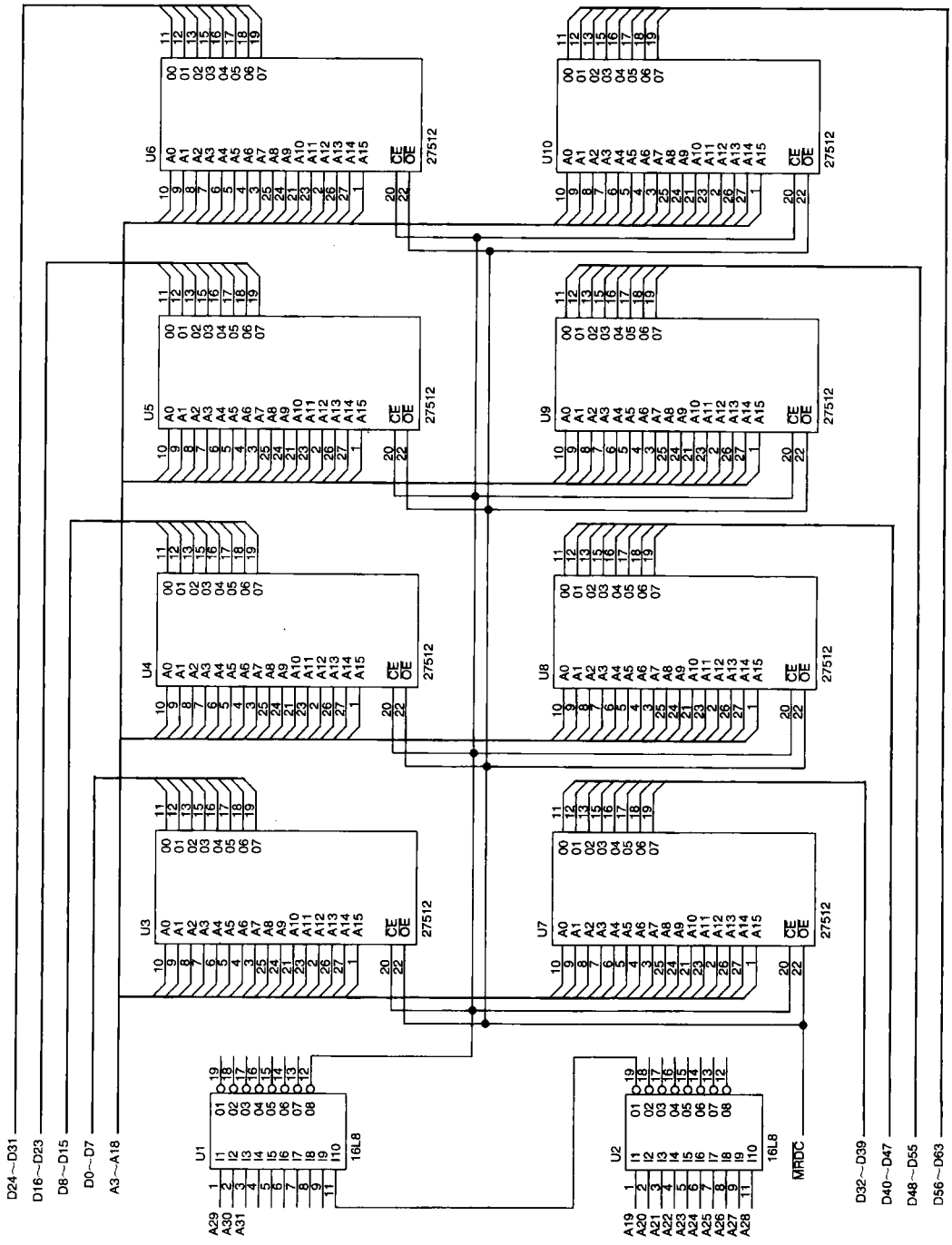


图 10-37 与 Pentium~Core2 微处理器相连的一个 512KB EPROM 存储器

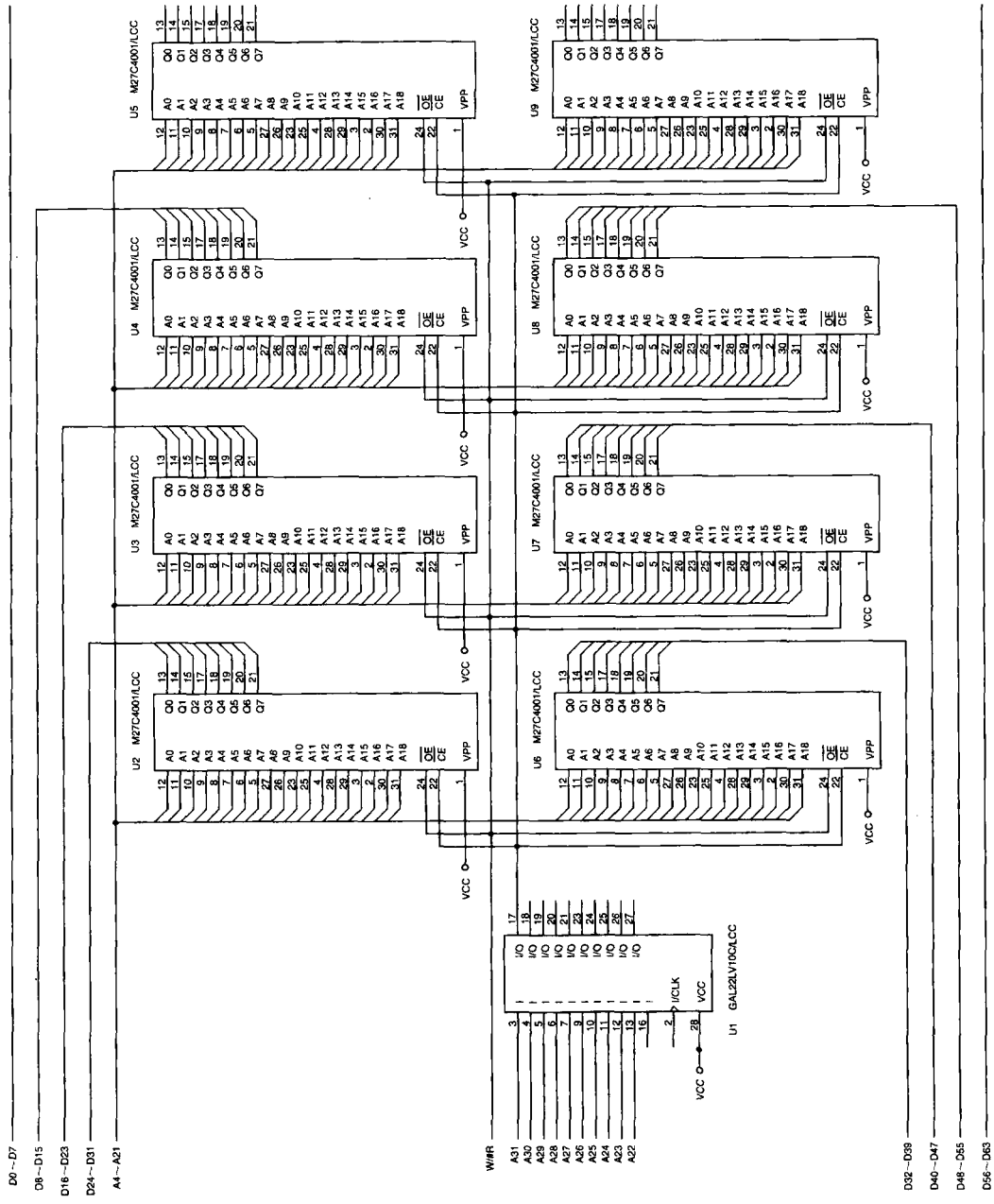


图 10-38 Pentium~Core2 微处理器的一个小型 4MB EPROM 存储器

例 10-10

```
library ieee;
use ieee.std_logic_1164.all;

entity DECODER_10_38 is

port (
       A31, A30, A29, A28, A27, A26, A25, A24, A23, A22: in STD_LOGIC;
       SEL: out STD_LOGIC
);

end;

architecture V1 of DECODER_10_38 is

begin

       SEL <= not(A31 and A30 and A29 and A28 and A27 and A26 and A25 and A24
              and A23 and A22);

end V1;
```

还有一点没有提到，那就是 Intel 的安腾和安腾Ⅱ的存储器接口，它的数据总线宽度为 128 位。从本章不难总结出，为安腾设计一个具有 16 个存储体的存储器是一件相当容易的事情。

10.7 DRAM

由于 RAM 存储量通常很大，它需要许多成本很高的 SRAM 器件，或是需要几个成本低得多的 **DRAM（dynamic RAM，动态 RAM）**。正如 10.1 节中讨论过的，DRAM 存储器相当复杂，因为它需要地址多路复用和刷新。幸运的是，集成电路制造商提供了一个动态 RAM 控制器，它包括地址多路转换器和刷新所需的所有定时电路。

本节比 10.1 节更详细地讨论 DRAM 存储器件，并介绍在存储系统中如何使用动态控制器。

10.7.1 DRAM 回顾

正如 10.1 节中提到的，DRAM 只能保留数据 2 ～ 4ms，并需要对地址输入多路复用。我们已在 10.1节中讨论了地址多路转换器，这里将详细分析 DRAM 在刷新期间的操作。

正如以前提到的，DRAM 必须周期性地刷新，因为它在电容上存储数据，而在一个短暂的时间后电容上的电荷会流失。为刷新 DRAM，存储器中的内容必须周期性地读出或写入。任何一次读或写都会自动刷新整个一段 DRAM。被刷新的位数取决于存储器件的容量及其内部组织。

刷新周期是由一次读操作、一次写操作或并不读写数据的特殊的刷新周期完成的。刷新周期对 DRAM 来说是内部操作，并在系统中的其他存储器件操作时完成。这种存储器刷新被称为隐藏刷新或透明刷新，有时也被称为周期挪用。

当其他存储器件正在操作时，为了完成一个隐藏刷新，$\overline{\text{RAS}}$周期性地选通一个行地址送入 DRAM，以选择要被刷新的一行。$\overline{\text{RAS}}$输入还使得选中的行在内部读出并回写，使存储数据的内部电容再次被充电。这种刷新被系统隐藏起来，因为它出现在微处理器正在读或写存储器的其他段时。

DRAM 的内部组织包含一系列行和列。一个 256K × 1 的 DRAM 有 256 列，每列 256 位；或按行分成 4 个段，每段 64K 位。当寻址一个存储单元时，列地址选中 1 列（或内部存储字），共 1024 位（每段 DRAM 256 位）。参见图 10-39 中 256K × 1 DRAM 的内部结构。注意，较大的存储器件结构与 256K × 1器件类似，区别在于每个段的容量不同，或并行段的数目不同。

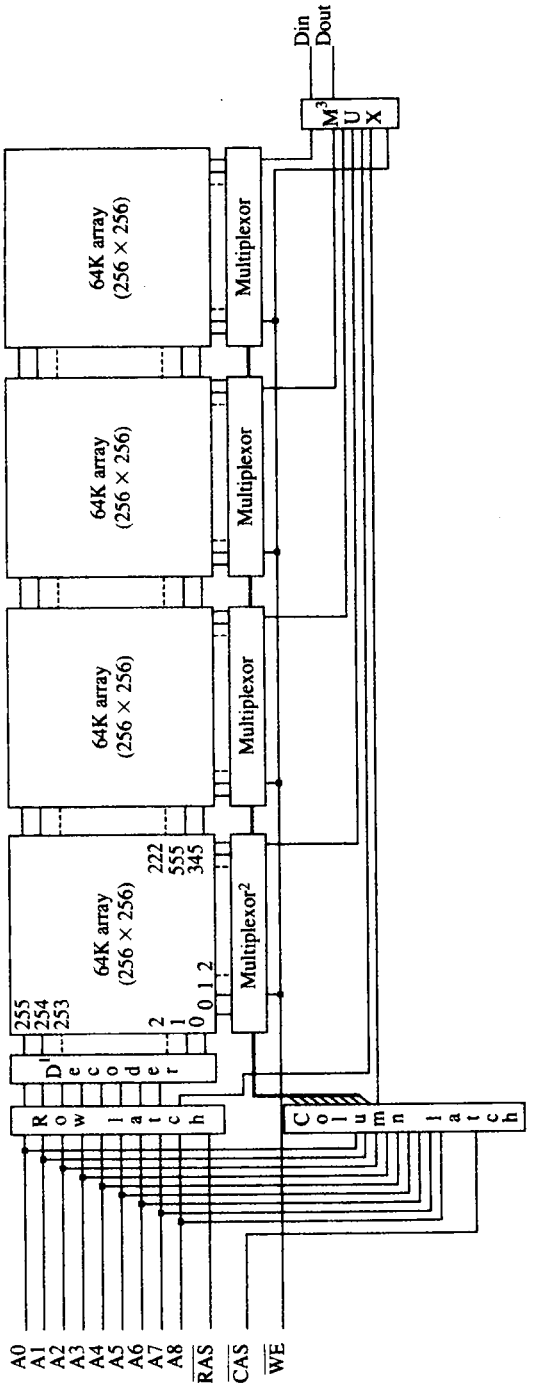


图 10-39 256K × 1 DRAM 的内部结构，内部 256 个字每个为 1024 位宽

注：1. 译码器为一个 8 线 256 线译码器。

2. 此多路器为 256 选 1。

3. 此多路器为 4 选 1。

图 10-40 描述了$\overline{RAS}$刷新周期的时序。$\overline{RAS}$与读写操作的区别在于它只需要刷新地址，通常由一个 7 位或 8 位二进制计数器提供。计数器的大小由被刷新的 DRAM 类型确定。刷新计数器在每个刷新周期结束时加 1，这样所有行在 2ms 或 4ms 内被刷新一遍，具体时间取决于 DRAM 的类型。

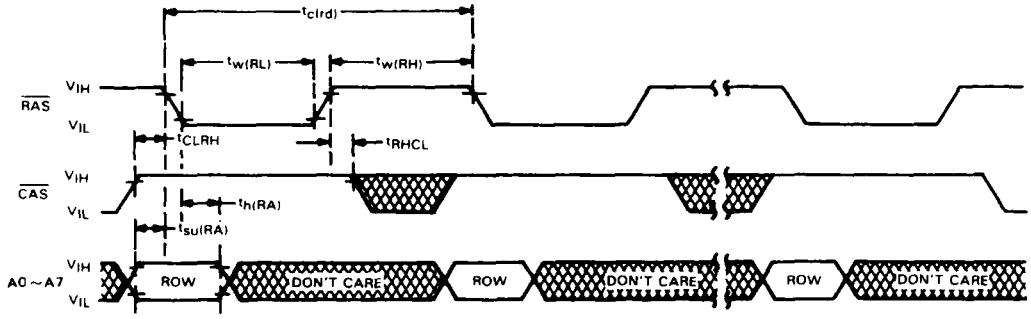


图 10-40 TMS4464 DRAM 的$\overline{RAS}$刷新周期时序图

若在 4ms 内有 256 行要刷新，例如 256K × 1 DRAM，则至少每 15.6μs 刷新周期必须被激活一次。如工作在 5MHz 时钟频率下的 8086/8088，执行一次读操作或写操作需要 800ns。因为 DRAM 每 15.6μs 必须有一个刷新周期，那么每对存储器执行 19 次读操作或写操作，存储系统必须执行一次刷新周期，否则存储器内数据将会丢失。这样将浪费计算机 5% 的时间，这正是因使用动态 RAM 节约成本所付出的小小代价。在一个基于如 Pentium 4 3.0GHz 的现代系统中，15.6μs 是一个相当长的时间。由于 Pentium 4 3.0GHz 可以在 1/3 ns 内执行一条指令（许多指令都在单时钟周期内执行），所以在刷新周期之间可以执行大概 46000 条指令。这就意味着，对这种机器来说，一个刷新周期占用远少于 1%（大概 0.002%）的时间。

10.7.2 EDO 存储器

对 DRAM 的结构稍做修改，则器件变成了 **EDO**（**extended data output，扩展数据输出**）DRAM 器件。在 EDO 存储器内，任何存储器存取，包括刷新，都将$\overline{RAS}$选中的 256 位数据存储在锁存器中。故在大多数顺序执行的程序中，不需要任何等待状态即可获得数据。对 DRAM 内部结构稍做修改就提高了 15% 到 25% 的系统性能。尽管 EDO 存储器现在已不再使用，但这项技术仍在现代 DRAM 中使用。

10.7.3 SDRAM

同步动态 RAM（**synchronous dynamic RAM，SDRAM**）由于其速度快而用于大多数较新的系统中。现在已可得到在 66MHz 系统总线频率下存取时间为 10ns，在 100MHz 下存取时间为 8ns，在 133MHz 下存取时间为 7ns 的版本。首先，如此短的存取时间可能会使人联想到这些器件的操作不需要等待状态，但事实上并非如此。毕竟，DRAM 的存取时间为 60ns，而 SDRAM 只有 10ns，因此 10ns 存取时间容易引起误解，因为它只适用于第 2 次、第 3 次和第 4 次从器件中读出 64 位数据，第 1 次读操作仍需要与标准 DRAM 相同的等待状态个数。

当微处理器突发地对 SDRAM 传送数据时，在第 1 个 64 位数读出前，需要 3 个或 4 个总线时钟等待。每个后续数读出则不需要等待状态，而只需要一个总线周期。由于 SDRAM 突发读 4 个 64 位数，第 2 个到第 4 个数不需要等待状态，每个数在一个总线周期内就可读出，所以 SDRAM 的性能优于标准 DRAM 甚至是 EDO 存储器。这意味着如果第 1 个数需要 3 个总线周期，后 3 个数又需要 3 个总线周期，则读 4 个 64 位数总共需要 7 个总线时钟；DRAM 每个数需要 3 个时钟，则 4 个数需要 12 个时钟。与之相比，可看出 SDRAM 速度提高了。估计 SDRAM 比 EDO 存储器提高了约 10% 的性能。

10.7.4 DDR

双数据速率（DDR）存储器是对 DRAM 的最新改进，它以两倍于 SDRAM 的速率传输数据，因为它在每个时钟的边沿传送数据。同时利用积极和消极的边沿传送数据。尽管看起来它好像是原来速率

的两倍，其实不然。主要原因就是存取时间问题仍然存在，就算最高级的存储器仍需要40ns的存取时间。如果微处理器以若干GHz的速率运行，那么对于存储器来说就要等待相当长的时间了。因此其速度并不像名字那样可以是双速率。

10.7.5 DRAM控制器

在大多数系统中，DRAM控制器集成电路完成多路复用地址和产生DRAM控制信号的功能。一些较新的嵌入式微处理器，如80186/80188，将刷新电路作为微处理器的一部分。许多现代计算机的芯片组中都包含DRAM控制器，所以单机DRAM控制器是不可用的。微处理器芯片组中的DRAM控制器记录刷新周期，并把刷新周期加入时序。存储器刷新对微处理器透明，因为它实际上并不控制刷新。

对于Pentium Ⅱ、Ⅲ及Pentium 4，DRAM控制器集成在Intel或AMD提供的芯片组中。许多年来，一个单独的DRAM控制器一直被用来构建一个计算机系统。相信不久的将来，微处理器将由芯片组构成。

10.8 小结

1）所有存储器件都有地址输入，都有数据输入和输出，或只有数据输出，还有一个选择引脚，有一个或多个控制存储器操作的引脚。

2）存储器件的地址引脚用于选择器件内的一个存储单元。10个地址引脚有1024种组合，因此能够寻址1024个不同的存储单元。

3）存储器的数据线用于把要存储的信息输入给存储单元，也可检索从存储单元读出的信息。制造商将他们的存储器标识为4K×4，表明此器件有4K存储单元（4096），每个存储单元存储4位数据。

4）存储器选择是由多数RAM上的片选引脚（$\overline{CS}$）或多数EPROM或ROM上的芯片允许引脚（$\overline{CE}$）实现的。

5）存储器功能由输出允许引脚（$\overline{OE}$）或写允许引脚（$\overline{WE}$）选择。读数据时，$\overline{OE}$通常与系统读信号（$\overline{RD}$或$\overline{MRDC}$）相连；写数据时，$\overline{WE}$通常与系统写信号（$\overline{WR}$或$\overline{MWTC}$）相连。

6）EPROM存储器由EPROM编程器编程；若暴露在紫外线下，则内容被擦除。现在EPROM的容量可从1K×8到512K×8甚至更大。

7）快闪存储器（EEPROM）通过使用12V或5V编程脉冲在系统内编程。

8）静态存储器（SRAM）只要系统电源接通即可保持数据。其容量最大可达128K×8。

9）动态存储器（DRAM）只能把数据保持短暂的时间，通常是2～4ms。因为DRAM必须周期性地刷新，所以给存储系统设计人员带来一些问题。DRAM还有多路复用的地址输入，因此需要额外的多路转换器件，以便在适当的时候分别提供一半地址。

10）存储器地址译码器在特定的存储器区域内选择EPROM或ROM。常用的地址译码器有74LS138 3－8译码器，74LS139 2－4译码器以及可编程逻辑PROM或PLD。

11）用于8088～Pentium 4微处理器的PROM和PLD地址译码器，减少了完成存储器功能所需集成电路的数目。

12）8088最小模式的存储器接口有20条地址线、8条数据线和3条控制线：$\overline{RD}$、$\overline{WR}$和IO/$\overline{M}$。只有当所有这些线都用于存储器接口时，8088存储器才能正常工作。

13）EPROM的存取速度必须和与之接口的微处理器兼容。现在很多EPROM的存取时间为450ns，这对5MHz的8088来说太慢。为解决此问题，必须插入一个等待状态，将存储器存取时间增加到660ns。

14）错误校正现在也是存储系统的特性，但这需要存储更多的位。如果一个8位数用一个错误校正电路存储，则实际上需要13位的存储器：5位用于错误校正码，8位用于数据。大多数错误校正电路只能校正1位错误。

15）8086/80286/80386SX存储器接口有一条16位的数据总线和一个M/$\overline{IO}$控制引脚，而8088有一条8位的数据总线和一个IO/$\overline{M}$控制引脚。除这些不同外，8086/80286/80386SX还有一个额外的控制信号，即总线高允许（$\overline{BHE}$）。

16）8086/80286/80386SX存储器组织成2个8位存储体：高位存储体和低位存储体。高位存储体由$\overline{BHE}$控制信号允许，而低位存储体由A_0地址信号或$\overline{BLE}$控制信号允许。

17）在基于8086/80286/80386SX的系统中，2个选择存储体的常见方案是：（1）每个存储体有一个独立的译码器；（2）每个存储体有一个独立的$\overline{WR}$控制信号和一个公用的译码器。

18）与80386DX和80486接口的存储器为32位宽，并由32位地址总线选择。由于这种存储器的宽度为32位，故它被组织成4个存储体，每个存储体8位宽。存储体选择信号由微处理器的$\overline{BE3}$、$\overline{BE2}$、$\overline{BE1}$和$\overline{BE0}$提供。

19）与Pentium～Core2接口的存储器为64位宽，并由32位地址总线选择。由于这种存储器的宽度是64位，故它被组织成8个存储体，每个存储体8位宽。存储体选择信号由微处理器的$\overline{BE7}$～$\overline{BE0}$提供。

20）动态RAM控制器用于控制DRAM存储器件。今天许多基于芯片组的DRAM控制器都有地址多路转换器、刷新计数器以及周期性刷新DRAM存储器所需的电路。

10.9 习题

1. 有哪些类型的引脚对全体存储器件是共有的?
2. 列出具有下列数目地址引脚的每个存储器件的存储字个数:
   (a) 8
   (b) 11
   (c) 12
   (d) 13
   (e) 20
3. 列出存储在下列每个存储器件中的数据项个数及每个数据的位数:
   (a) 2K×4
   (b) 1K×1
   (c) 4K×8
   (d) 16K×1
   (e) 64K×4
4. 存储器件上$\overline{CS}$或$\overline{CE}$引脚的用途是什么?
5. 存储器件上$\overline{OE}$引脚的用途是什么?
6. RAM 上$\overline{WE}$引脚的用途是什么?
7. 下列 EPROM 存储器件可容纳多少字节数据?
   (a) 2708
   (b) 2716
   (c) 2732
   (d) 2764
   (e) 27512
8. 为什么450ns 的 EPROM 不能直接与5MHz 的8088 一起工作?
9. 在快闪存储器中,用什么来表示擦除和写存储单元的时间?
10. SRAM 是哪种器件的缩写?
11. 4016 存储器有一个$\overline{G}$引脚、一个$\overline{S}$引脚和一个$\overline{W}$引脚。这些引脚在 RAM 中的用途是什么?
12. 速度最慢的4016 要求的存取时间是多少?
13. DRAM 是哪种器件的缩写?
14. 256M DIMM 有 28 个地址输入,然而它是一个 256M DRAM。解释一个28 位存储器地址是如何强制进入14 位地址输入的。
15. DRAM 的$\overline{CAS}$和$\overline{RAS}$输入的用途是什么?
16. 刷新典型的 DRAM 需要多长时间?
17. 为什么存储器地址译码器很重要?
18. 修改图 10-13 中的与非门译码器,使它选择的存储器地址范围为 DF800H ~ DFFFFH。
19. 修改图 10-13 中的与非门译码器,使它选择的存储器地址范围为 40000H ~ 407FFH。
20. 当 G1 输入为高电平,$\overline{G2A}$和$\overline{G2B}$均为低电平时,74LS138 3 - 8 译码器的输出是什么?
21. 修改图 10-15 中的电路,使其寻址存储器的地址范围为 70000H ~ 7FFFFH。
22. 修改图 10-15 中的电路,使其寻址存储器的地址范围为 40000H ~ 4FFFFH。
23. 说明 74LS139 译码器的原理。
24. 什么是 VHDL?
25. VHDL 中,对应五个主要逻辑功能(与,或,与非,异或,反相)的关键字是什么?
26. 等式出现在 VHDL 程序的什么模块中?
27. 通过重写 PLD 程序修改图 10-19 中的电路,使其寻址 ROM 存储器的地址范围为 A0000H ~ BFFFFH。
28. 8086 最小模式下的$\overline{RD}$和$\overline{WR}$控制信号在最大模式中由哪两个信号代替了?
29. 修改图 10-20 中的电路,使其选择的存储器地址范围为 68000H ~ 6FFFFH。
30. 修改图 10-20 中的电路,使其选择 8 个 27256 (32K × 8) EPROM,存储器地址范围为 40000H ~ 7FFFFH。
31. 给图 10-21 的电路增加 1 个译码器,使得另外可增加 8 个 62256 SRAM,其地址范围为 C0000H ~ FFFFFH。
32. 74LS636 错误校正与检测电路为每字节数据存储一个校验码。校验码是多少位?
33. 74LS636 上的 SEF 引脚有什么用途?
34. 74LS636 可校正_____位错误。
35. 概述 8086 和 8088 微处理器总线之间的主要区别。
36. 8086 微处理器上的$\overline{BHE}$和 A_0 引脚有什么用途?
37. $\overline{BLE}$是什么引脚?它可取代其他哪个引脚?
38. 在 8086 微处理器中使用哪两种方法选择存储器?
39. 若$\overline{BHE}$为逻辑 0,则_____存储体被选中。
40. 若 A_0 为逻辑 0,则_____存储体被选中。
41. 当存储器与 8086 接口时,为什么不必产生独立的存储体读选通($\overline{RD}$)?
42. 修改图 10-31 中的电路,使 EPROM 的地址范围为 C0000H ~ CFFFFH, RAM 的地址范围为 30000H ~ 4FFFFH。
43. 为 80386SX 设计一个 16 位宽的存储器接口,它包含 SRAM 存储器,地址范围为 200000H ~ 21FFFFH。
44. 设计一个 32 位宽的存储器接口,它包含 EPROM 存储器,地址范围为 FFFF0000H ~ FFFFFFFFH。
45. 为 Pentium ~ Core2 设计一个 64 位宽的存储器接口,它包含 EPROM 存储器,地址范围为 FFF00000H ~ FFFFFFFFH;还包含 SRAM 存储器,地址范围为 00000000H ~ 003FFFFFH。
46. 在 Internet 上搜索最大容量的 EPROM,列出其大小和厂商。
47. 什么是$\overline{RAS}$周期?
48. DRAM 被刷新时,其他存储器段可以进行操作吗?
49. 如果一个 1M × 1 DRAM 需要 4ms 刷新一次,且每次需要刷新 256 行,那么在下一行被刷新前,至多要经过_____的时间。
50. 在 Intel 的 Itanium 中数据总线是多宽?
51. 在 Internet 上搜索目前可用的最大的 DRAM。
52. 写一个关于 DDR 存储器的报告(提示:三星发明了 DDR)。
53. 写一个描述 RAMBUS RAM 的报告,试着弄清楚为什么此技术似乎将半途而废。

第11章　基本I/O接口

引言

微处理器解决问题是很有效的，但如果不能与外界通信，它就没有什么价值。本章概述了人类或其他机器与微处理器之间通信的一些基本方法，包括串行与并行通信。

本章首先介绍基本I/O接口，并讨论对I/O设备的译码，然后详细介绍应用很广的并行接口和串行接口。作为应用实例，我们将模/数转换器、数/模转换器以及直流电机和步进电机连接到微处理器上。

目的

读者学习完本章后将能够：

1）解释基本输入输出接口的操作。
2）译码8位、16位和32位I/O设备，使之能用于任一I/O端口地址。
3）定义握手并解释如何将它用于I/O设备。
4）连接并编程82C55可编程并行接口。
5）将LCD显示器、LED显示器、键盘、ADC、DAC以及其他各种器件连接到82C55上。
6）连接并编程16550串行通信接口适配器。
7）连接并编程8254可编程间隔定时器。
8）将一个模/数转换器和一个数/模转换器连接到微处理器上。
9）将直流电机与步进电机连接到微处理器上。

11.1　I/O接口概述

本节介绍I/O指令（IN、INS、OUT及OUTS）的操作，解释独立编址I/O（有时称为直接I/O或I/O映像I/O）与存储器映像I/O的概念，说明基本输入输出接口以及握手。这些知识使我们更容易理解可编程接口器件的连接与操作，以及本章后面和其他章节将讨论的I/O技术。

11.1.1　I/O指令

指令系统包含向I/O设备传送信息的指令（OUT）和从I/O设备读出信息的指令（IN）。除8086/8088外，Intel所有微处理器都有INS与OUTS指令，用于在存储器和I/O设备间传送数据串。表11-1列出了微处理器指令系统中每条指令的所有形式。

表11-1　输入/输出指令

指　令	数据宽度	功　能
IN AL, p8	8	从端口p8输入一个字节到AL
IN AX, p8	16	从端口p8输入一个字到AX
IN EAX, p8	32	从端口p8输入一个双字到EAX
IN AL, DX	8	从DX寻址的端口输入一个字节到AL
IN AX, DX	16	从DX寻址的端口输入一个字到AX
IN EAX, DX	32	从DX寻址的端口输入一个双字到EAX
INSB	8	从DX寻址的端口输入一个字节到由DI寻址的附加段存储单元，然后DI=DI±1
INSW	16	从DX寻址的端口输入一个字到由DI寻址的附加段存储单元，然后DI=DI±2
INSD	32	从DX寻址的端口输入一个双字到由DI寻址的附加段存储单元，然后DI=DI±4
OUT p8, AL	8	从AL输出一个字节到端口p8
OUT p8, AX	16	从AX输出一个字到端口p8

（续）

指　令	数据宽度	功　能
OUT p8, EAX	32	从 EAX 输出一个双字到端口 p8
OUT DX, AL	8	从 AL 输出一个字节到 DX 寻址的端口
OUT DX, AX	16	从 AX 输出一个字到 DX 寻址的端口
OUT DX, EAX	32	从 EAX 输出一个双字到 DX 寻址的端口
OUTSB	8	从由 SI 寻址的数据段存储单元输出一个字节到 DX 寻址的端口，然后 SI = SI ±1
OUTSW	16	从由 SI 寻址的数据段存储单元输出一个字到 DX 寻址的端口，然后 SI = SI ±2
OUTSD	32	从由 SI 寻址的数据段存储单元输出一个双字到 DX 寻址的端口，然后 SI = SI ±4

在 I/O 设备和微处理器的累加器（AL、AX 或 EAX）之间传送数据的指令叫做 **IN** 和 **OUT** 指令。I/O 地址存储在寄存器 DX 中作为一个 16 位 I/O 地址，或存储在紧跟操作码的字节（p8）里作为一个 8 位 I/O 地址。Intel 称 8 位形式（p8）为**固定地址**（**fixed address**），因为它通常与指令一起存储在 ROM 里。DX 中的 16 位 I/O 地址被称为**可变地址**（**variable address**），因为它存储在 DX 中，用于寻址 I/O 设备。其他使用 DX 寻址 I/O 的指令是 INS 和 OUTS 指令。I/O 端口的位宽是 8 位，所以任何时候访问一个 16 位的端口，就要访问两个 8 位的连续编址端口。实际上，一个 32 位的端口，就是四个 8 位的端口。例如，以字的方式访问端口 100H，实际上就是访问了 100H 与 101H 这两个端口。端口 100H 包含数据的低 8 位，而 101H 包含数据的高 8 位。

当使用 IN 或 OUT 指令传送数据时，常被称为**端口号**（或端口）的 I/O 地址出现在地址总线上，外部 I/O 接口就像译码存储器地址一样对端口号进行译码。8 位固定端口号（p8）出现在地址总线 $A_7 \sim A_0$ 上，此时 $A_{15} \sim A_8$ 为 00000000_2。对 I/O 指令来说，A_{15} 以上的地址线没有定义。16 位可变端口号（DX）出现在地址线 $A_{15} - A_0$ 上，这意味着开始的 256 个 I/O 端口地址（00H ～ FFH）由固定的和可变的 I/O 指令访问，但 0100H ～ FFFFH 中的任一 I/O 地址只能由可变的 I/O 指令访问。在许多专用任务系统中，只有最右边 8 位地址被译码，以减少译码所需电路的数量。在 PC 机中，所有 16 个地址总线位被译码为 0000H ～ 03XXH，这些地址是 PC 机内部 **ISA**（**industry standard architecture**，**工业标准结构**）总线用于 I/O 设备的 I/O 地址。

INS 和 OUTS 指令使用 DX 寄存器来寻址 I/O 设备，但不像 IN 和 OUT 指令那样在累加器和 I/O 设备之间传送数据，而是在存储器和 I/O 设备之间传送数据。存储器地址在 INS 指令中由 ES：DI 定位，在 OUTS 指令中由 DS：SI 定位。正如其他串指令一样，指针中内容根据方向标志（DF）的状态加 1 或减 1。INS 与 OUTS 指令均可加 REP 前缀，从而允许多于一个字节、一个字或一个双字长度的数据在 I/O 与存储器之间传送。

Pentium 4 与 Core2 的 64 位模式操作有相同的 I/O 指令。但在 64 位模式中并没有 64 位的 I/O 指令，主要原因是大部分的 I/O 依然是 8 位的，而且很可能会使用相当长一段时间。

11.1.2 独立编址 I/O 与存储器映像 I/O

有两种不同的方法连接 I/O 与微处理器：**独立编址 I/O**（**isolated I/O**）与**存储器映像 I/O**（**memory-mapped I/O**）。在独立编址 I/O 方案里，IN、INS、OUT 及 OUTS 指令在微处理器的累加器或存储器与 I/O 设备之间传送数据。在存储器映像 I/O 方案里，任一涉及存储器的指令均可完成数据传送。独立编址 I/O 与存储器映像 I/O 都在使用，因此这两种方法本章都将讨论。PC 机不采用存储映像 I/O。

独立编址 I/O

在基于 Intel 微处理器的系统中使用最普遍的 I/O 传送技术是独立编址 I/O。术语“独立编址”描述了 I/O 存储单元是如何与存储系统隔离，而存在于一个独立的 I/O 地址空间里的（图 11-1 给出了任一 Intel 80X86 或 Pentium ～ Core2 微处理器的独立编址与存储器映像地址空间）。独立编址 I/O 设备的地址称为端口，与存储器是隔离的。由于端口是隔离的，所以用户可扩展存储器到最大容量而不必为 I/O 设备留出存储器空间。独立编址 I/O 的一个缺点是，在 I/O 与微处理器之间传送的数据必须由 IN、INS、OUT 及

OUTS指令存取。I/O空间独立的控制信号由M/$\overline{IO}$和W/$\overline{R}$产生,它们指示一次I/O读($\overline{IORC}$)或一次I/O写($\overline{IOWC}$)操作。这些信号还表明,出现在地址总线上的I/O端口地址被用于选择I/O设备。在PC机中,

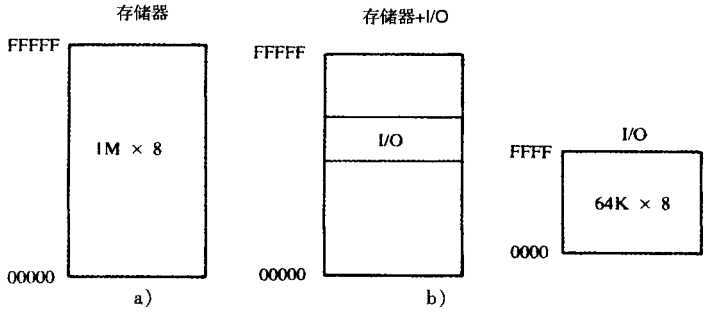


图11-1 8086/8088微处理器的存储器与I/O映像

a)独立编址I/O b)存储器映像I/O

独立编址I/O端口用于控制外围设备。一个8位端口地址用于访问系统板上的设备,如定时器和键盘接口,而一个16位端口用于访问串行和并行端口以及视频和磁盘驱动系统。

存储器映像I/O

不同于独立编址I/O,存储器映像I/O不使用IN、INS、OUT或OUTS指令。相反,它使用任一在微处理器与存储器间传送数据的指令。存储器映像I/O设备被视为存储器映像中的一个存储单元。存储器映像I/O的主要优点在于任何存储器传送指令都可用来访问I/O设备。主要缺点在于一部分存储器被用作I/O映像,这样就减少了可用存储器的数量。另一优点是$\overline{IORC}$和$\overline{IOWC}$信号在存储器映像I/O系统中不起作用,从而可减少译码所需电路的数量。

11.1.3 PC机I/O映像

PC机使用部分I/O映像用于专用功能,图11-2示出了PC机的I/O映像。注意,在端口0000H和03FFH之间的I/O空间通常留给计算机系统和ISA总线,位于0400H～FFFFH的I/O端口一般用于用户应用、主板功能及PCI总线。注意,80287算术协处理器使用I/O地址00F8H～00FFH进行通信,因此Intel保留I/O端口00F8H～00FFH。80386～Core2使用I/O端口800000F8H～800000FFH与协处理器通信。I/O端口0000H～00FFH通过固定端口I/O指令被访问,00FFH以上的端口通过可变I/O端口指令被访问。

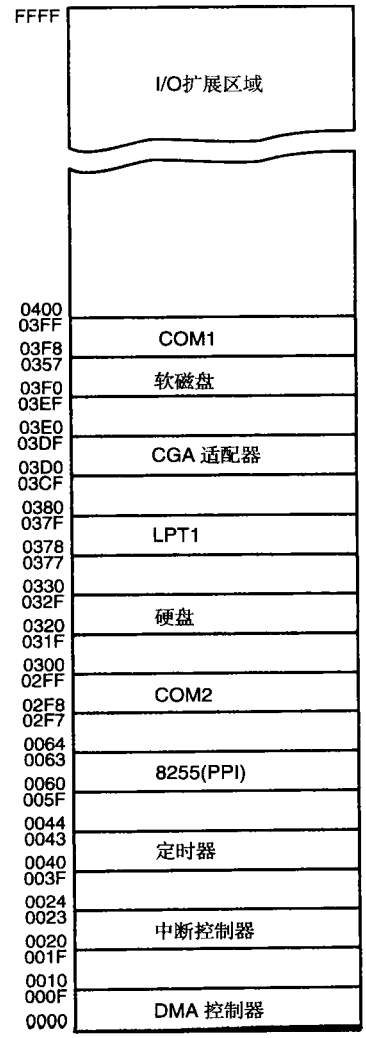


图11-2 有许多固定I/O区域的PC机I/O映像

11.1.4 基本输入输出接口

基本输入设备是一组三态缓冲器,基本输出设备是一组数据锁存器。术语"输入"指将数据从I/O设备移入微处理器中,术语"输出"指将数据从微处理器中移出并送给I/O设备。

基本输入接口

三态缓冲器用于构造8位输入端口,如图11-3所示。外部TTL数据(本例中是简单切换开关)被连到缓冲器的输入上,缓冲

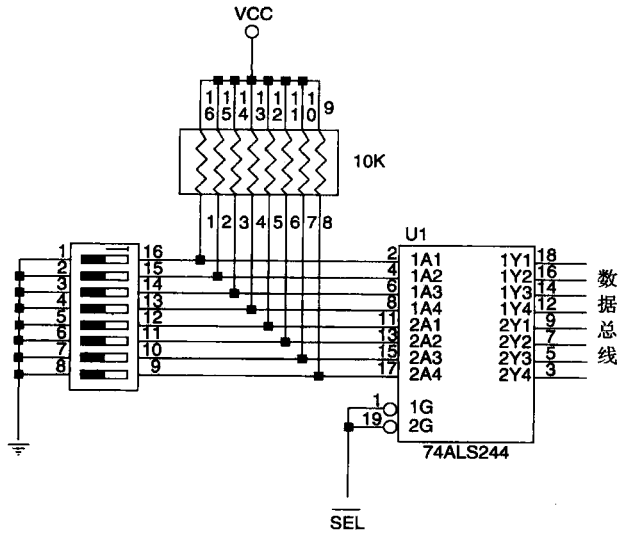


图 11-3 与 8 个开关相连的基本输入接口

注:74ALS244 是一个三态缓冲器,控制将开关数据加到数据总线上。

器输出与数据总线相连。具体的数据总线宽度取决于微处理器的型号。例如,8088 的数据总线为 $D_7 \sim D_0$,80486 的数据总线为 $D_{31} \sim D_0$,Pentium ~ Core2 的数据总线为$D_{63} \sim D_0$。图 11-3的电路允许在选择信号$\overline{SEL}$变为逻辑 0 时,微处理器读取与任意 8 位数据总线相连的 8 个开关的内容。因此,一旦执行 IN 指令,开关的内容就被复制到 AL 寄存器中。

当微处理器执行 IN 指令时,I/O 端口地址被译码,在$\overline{SEL}$上产生逻辑 0。74ALS244 缓冲器的输出控制引脚($\overline{1G}$和$\overline{2G}$)被置为 0,使得数据输入线(A)与数据输出线(Y)相连。若$\overline{1G}$和$\overline{2G}$置为 1,则器件进入三态高阻抗模式,从而有效地将开关与数据总线断开。

这个基本输入电路并不是可有可无的,只要输入数据要接到微处理器上,就必须有此电路。有时它作为电路的独立部分出现,如图 11-3 所示;有时它被构造在一个可编程 I/O 设备的内部。

16 位或 32 位数据也可与各种型号的微处理器相连,但这不像使用 8 位数据那么普遍。为连接 16 位数据,图 11-3 中的电路要增加一倍,即包含 2 个 74ALS244 缓冲器,将 16 位输入数据与 16 位数据总线相连。为连接 32 位数据,则该电路要扩展到 4 倍。

基本输出接口

基本输出接口从微处理器接收数据,而且通常必须为某个外部设备保持该数据。其锁存器或触发器,就像输入设备中的缓冲器一样,经常建造在 I/O 设备内部。

图 11-4 显示了 8 个简单发光二极管(LED)是如何通过八数据锁存器与微处理器相连的。锁存器存储微处理器从数据总线上输出的数据,使 LED 可显示任意 8 位二进制数。需要用锁存器来保持数据,是因为当微处理器执行 OUT 指令时,数据存在于数据总线上的时间不到 1.0μs,所以如果没有锁存器,观察者将看不到 LED 发光。

当输出指令执行时,来自 AL、AX 或 EAX 的数据通过数据总线传送给锁存器。这里,74ALS374 八锁存器的 D 输入与数据总线相连,以捕获输出数据,锁存器的 Q 输出连接到 LED 上。当一个 Q 输出变为逻辑 0 时,相应的 LED 发光。每次 OUT 指令执行时,锁存器的$\overline{SEL}$信号被激活,捕获从任意 8 位数据总线上输出给锁存器的数据,且数据一直保持到下一条 OUT 指令执行。因此,一旦对此电路执行输出指令,则来自 AL 寄存器的数据就会出现在 LED 上。

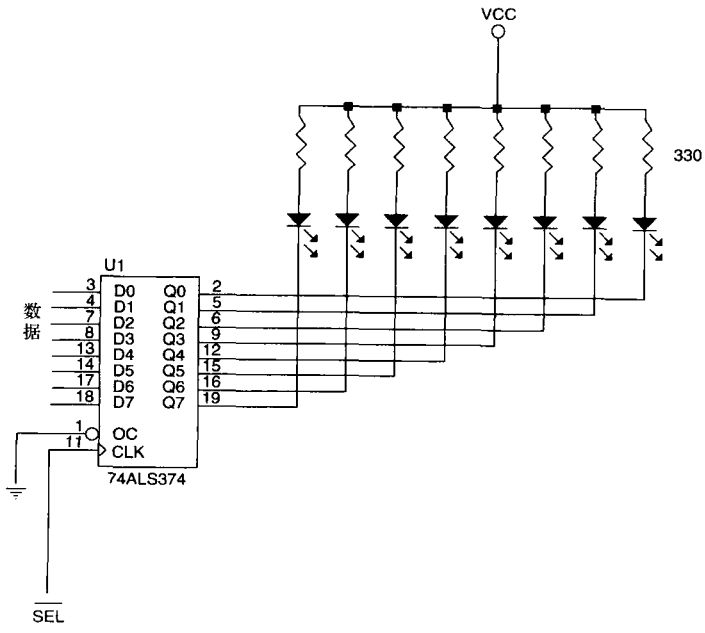


图 11-4 与一组 LED 显示器相连的基本输出接口

11.1.5 握手

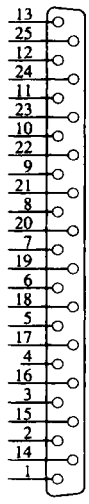
许多 I/O 设备接收或发送信息的速度比微处理器慢得多。I/O 控制的另一种方法被称为**握手**(**handshaking**)或**查询**(**polling**),可使 I/O 设备与微处理器同步。一个需要握手的设备例如并行打印机,它每秒打印 100 个字符(CPS)。显然微处理器可以超过 100CPS 的速度发送数据给打印机,所以必须设计一种方法来降低微处理器的速度去匹配打印机。

图 11-5 给出了打印机的典型输入输出引脚。这里,数据通过一组数据线($D_7 \sim D_0$)传送,BUSY 指示打印机"忙"状态,$\overline{STB}$是一个时钟脉冲,用于发送数据给打印机进行打印。

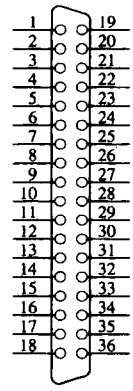
要打印的 ASCII 数据被置于 $D_7 \sim D_0$ 上,然后把一个脉冲加到$\overline{STB}$引脚上,该选通信号发送数据给打印机,进行打印。一旦打印机接收到数据,它就将 BUSY 引脚置 1,表明打印机正忙于打印数据。微处理器软件查询或测试 BUSY 引脚以决定打印机是否"忙"。若打印机"忙",则微处理器就等待,否则微处理器就发送下一个 ASCII 字符给打印机。这种查询打印机或任何类似于打印机的异步设备的过程称为握手或查询。例 11-1 给出了一个简单程序,测试打印机 BUSY 标志,并在打印机不忙时发送数据给打印机。PRINT 程序只有在 BUSY 标志为逻辑 0 即指示打印机不忙时,打印 BL 中 ASCII 编码的内容。每调用一次该程序,打印一个字符。

例 11-1

```
; 一个打印 BL 中 ASCII 内容的汇编语言程序

PRINT PROC   NEAR

      .REPEAT                        ;测试忙标志位
            IN AL,BUSY
            TEST AL,BUSY_BIT
      .UNTIL ZERO
      MOV AL,BL                      ;取 BL 中的数据
      OUT PRINTER,AL                 ;把数据送到打印机
      RET

PRINT ENDP
```



连接



连接器CENT36

DB25 引脚号	CENT36 引脚号	功能	DB25 引脚号	CENT36 引脚号	功能
1	1	数据选通	12	12	纸已空
2	2	Data 0 (D0)	13	13	选择
3	3	Data 1 (D1)	14	14	Afd
4	4	Data 2 (D2)	15	32	$\overline{\text{Error}}$
5	5	Data 3 (D3)	16	—	$\overline{\text{RESET}}$
6	6	Data 4 (D4)	17	31	选择输入
7	7	Data 5 (D5)	18—25	19—30	地
8	8	Data 6 (D6)	—	17	帧地
9	9	Data 7 (D7)	—	16	地
10	10	$\overline{\text{Ack}}$	—	33	地
11	11	忙			

图 11-5 计算机上的 DB25 连接器与打印机上用于 Centronics 并行打印机接口的 Centronics 36 引脚连接器

11.1.6 关于接口电路的注释

接口电路的特定部分需要一些电子学方面的知识。这部分内容介绍了与电子接口有关的一些情况。在一个电路或设备与微处理器连接之前，必须了解微处理器的终端特性及其相关的接口部件（在第 9 章开始介绍过）。

输入设备

输入设备可能本身就是 TTL 或与 TTL 兼容的电路，因此可与微处理器及其接口部件相连；它们也可能是基于开关的设备。大多数基于开关的设备要么是断开的，要么是接通的，它们不是 TTL 电平——TTL 电平为逻辑 0（0.0～0.8V）或逻辑 1（2.0～5.0V）。

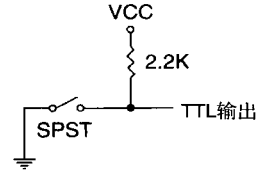


图 11-6 将一个单刀单掷开关作为 TTL 设备连接

一个开关型设备要用作 TTL 兼容的输入设备，必须做一些调整。图 11-6 说明了一个简单的切换开关是如何正确连接以用作输入设备的。注意，这里使用了一个上拉电阻以确保当开关断开时，输出信号是逻辑 1；当开关闭合时，开关接地，

从而产生一个有效的逻辑0电平。上拉电阻的阻值不是很严格——它只要保证信号在逻辑1电平即可。上拉电阻的标准值范围通常在1 kΩ～10 kΩ。

机械开关闭合时其触点会自然反跳(又叫抖动),因此当开关用作数字电路的定时信号时会产生问题。为防止抖动问题,可构造图11-7给出的两个电路之一。电路图11-7a是典型的教科书里的去抖动电路,电路图11-7b是一个更实用的电路。因为电路图11-7a成本更高,而电路图11-7b不需要上拉电阻,只需要2个反相器取代2个与非门,所以在实际中将会使用电路图11-7b。

可以注意到图11-7的两个电路都是异步触发器。电路图11-7b以如下方式工作:假定开关现在处于位置Q,如果它向$\overline{Q}$方向闭合但还未接触上$\overline{Q}$,则电路的$\overline{Q}$输出为逻辑0,逻辑0状态由反相器记忆。反相器B的输出与反相器A的输入相连,由于反相器B输出为逻辑0,所以反相器A输出为逻辑1。反相器A的逻辑1输出使反相器B的输出维持在逻辑0,触发器保持在此状态,直到正在移动的开关触点首次接触到$\overline{Q}$。一旦开关的$\overline{Q}$输入变为逻辑0,它就改变了触发器状态,如果触点离开$\overline{Q}$输入弹回,则触发器记忆此状态,没有出现变化,所以消除了抖动。

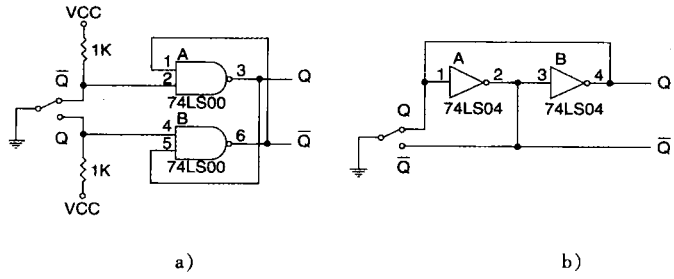


图11-7 开关触点的去抖动电路

a)传统的去抖动电路 b)实用的去抖动电路

输出设备

输出设备与输入设备大不相同,但许多设备以统一的方式连接。在连接任何输出设备之前,必须了解来自微处理器或TTL接口部件的电压和电流是多少。来自微处理器或接口元件的电压是TTL兼容的(逻辑0 = 0.0～0.4V,逻辑1 = 2.4～5.0V)。微处理器和许多微处理器接口部件的电流小于标准TTL部件的电流(逻辑0 = 0.0～2.0mA,逻辑1 = 0.0～400μA)。

一旦知道了输出电流,现在就可以将一个设备与微处理器的一个输出相连。图11-8显示如何将一个简单LED与微处理器外围引脚相连。注意图11-8a使用了一个晶体管驱动器,图11-8b使用了一个TTL反相器。TTL反相器(标准型号)在逻辑0电平时提供最大16mA电流,足以驱动一个标准LED,因为一个标准LED发光只需要10mA正向偏置电流。在这两个电路中,假设LED上的压降大约为2.0V,数据手册上一个LED的额定压降为1.65V,但根据经验来看压降在1.5V～2.0V之间,这意味着限流电阻的值为3.0V/10mA,即300Ω。由于300Ω不是标准电阻值,所以选用330Ω的电阻。

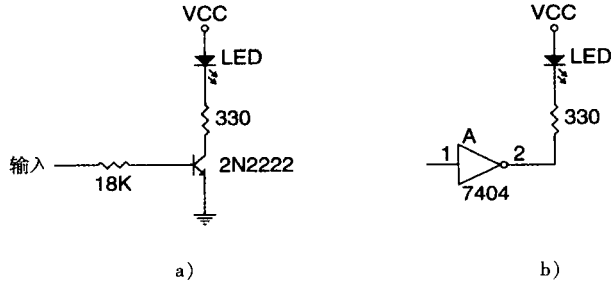


图11-8 连接一个LED

a)使用一个晶体管 b)使用一个反相器

在图11-8a的电路中,我们选择使用一个开关晶体管代替TTL缓冲器。2N2222是一个很好的通用开关晶体管,其最小增益为100。此电路中,集电极电流为10mA,因此基极电流将是集电极电流的1/100,即为0.1mA。为确定基极限流电阻的值,我们利用电阻上0.1mA电流和1.7V的压降。TTL输入信号的最小值为2.4V,在发射极－基极上压降为0.7V,二者之差为1.7V,即为限流电阻上的压降。故电阻值为1.7V/0.1mA,即17kΩ。由于17kΩ不是标准电阻值,所以选用18kΩ的电阻。

假设我们需要将一个12V直流电机与微处理器相连,且电机电流为1A。显然不能使用TTL反相器,原因有两个:12V信号会烧坏反相器,而且电机的电流值远远超过了反相器的16mA最大电流。也不能

使用2N2222晶体管，因为其电流最大值是250～500mA，具体值取决于封装类型。解决办法是使用一个达林顿复合晶体管，例如TIP120。它价值254并且可以使用适当的散热片处理当前4A的电流。

图11-9给出了电机与达林顿复合晶体管的连接。达林顿复合晶体管的最小电流增益为7000，最大电流为4A。基极电阻值的计算与LED驱动器中所用方法一样，通过此电阻的电流为1A/7000，即大约0.143mA。此电阻上的压降为0.9V，这是因为有2个二极管压降而不是1个。偏置电阻值为0.9V/0.143mA，即6.29kΩ，但在电路中使用6.2kΩ的标准值。由于流过达林顿复合晶体管的电流很大，所以它必须经过散热；另外还必须有一个二极管，以防达林顿复合晶体管被电机的感应回程所损坏。此电路还可用于连接机械继电器或任何需要大电流或改变电压的设备。

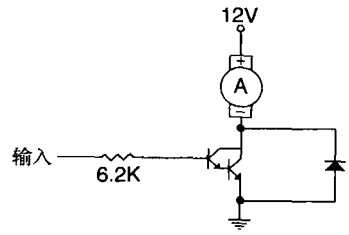


图11-9 使用达林顿复合晶体管将直流电机与系统相连

11.2 I/O端口地址译码

I/O端口地址译码与存储器地址译码非常相似，尤其是存储器映像I/O设备。事实上，我们不讨论存储器映像I/O译码，就因为它与存储器的译码方式相同（除了因没有IN或OUT指令而未使用$\overline{IORC}$和$\overline{IOWC}$外）。是否使用存储器映像I/O常取决于存储系统的容量以及系统中I/O设备的布局。

存储器译码与独立编址I/O译码间的主要区别在于与译码器相连的地址引脚数目不同。存储器译码的是A_{31}～A_0、A_{23}～A_0或A_{19}～A_0，而独立编址I/O译码的是A_{15}～A_0。有时如果I/O设备只使用固定I/O寻址，则只译码A_7～A_0。在PC机系统中，总是译码所有16个I/O端口地址位。另一区别是，使用$\overline{IORC}$和$\overline{IOWC}$激活I/O设备执行一次读或写操作。在早期微处理器中，使用IO/$\overline{M}$ = 1与$\overline{RD}$或$\overline{WR}$激活I/O设备，而在最新微处理器中，使用M/$\overline{IO}$ = 0与W/$\overline{R}$激活I/O设备。

11.2.1 译码8位I/O地址

前面提到，固定I/O指令使用出现在A_{15}～A_0上的8位I/O端口地址，地址为0000H～00FFH。如果一个系统包含肯定不超过256个的I/O设备，则常常只译码地址引脚A_7～A_0作为8位I/O端口地址，因此可以忽略地址引脚A_{15}～A_8。嵌入式系统常使用8位端口地址。请注意DX寄存器也可寻址I/O端口的00H～FFH。如果地址被译码为8位地址，则绝不能包括使用16位I/O地址的I/O设备。PC机从来不使用或译码8位I/O地址。

图11-10给出了一个74ALS138译码器，它译码8位I/O端口F0H～F7H（假定此系统只将I/O端口00H～FFH用于此译码器）。除只将地址位A_7～A_0与译码器输入相连之外，此译码器与存储器地址译码器相同。图11-11给出了PLD型译码器，即使用一个GAL22V10作为译码器。PLD是一个更好的译码器电路，因为集成电路的数目已被减少到只有一个器件。PLD的VHDL代码描述如例11-2所示。

例11-2

```
-- 图11-11中译码器的VHDL代码描述

library ieee;
use ieee.std_logic_1164.all;

entity DECODER_11_11 is

port (
       A7, A6, A5, A4, A3, A2, A1, A0: in STD_LOGIC;
       D0, D1, D2, D3, D4, D5, D6, D7: out STD_LOGIC
);

end;

architecture V1 of DECODER_11_11 is

begin

      D0 <= not( A7 and A6 and A5 and A4 and not A3 and not A2 and not A1 and
            not A0 );
```

```
    D1 <= not( A7 and A6 and A5 and A4 and not A3 and not A2 and not A1 and A0 );
    D2 <= not( A7 and A6 and A5 and A4 and not A3 and not A2 and A1 and not A0 );
    D3 <= not( A7 and A6 and A5 and A4 and not A3 and not A2 and A1 and A0 );
    D4 <= not( A7 and A6 and A5 and A4 and not A3 and A2 and not A1 and not A0 );
    D5 <= not( A7 and A6 and A5 and A4 and not A3 and A2 and not A1 and A0 );
    D6 <= not( A7 and A6 and A5 and A4 and not A3 and A2 and A1 and not A0 )
    D0 <= not( A7 and A6 and A5 and A4 and not A3 and A2 and A1 and A0 );
end V1;
```

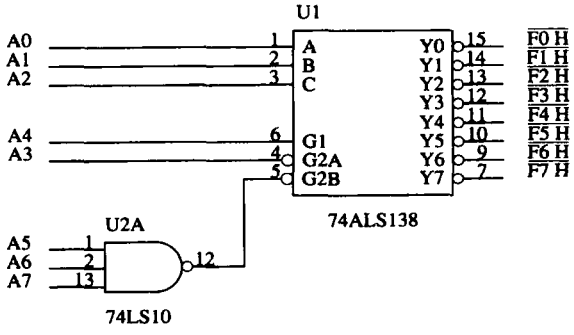


图 11-10 译码 8 位 I/O 端口的端口译码器,为端口 F0H ~ F7H 产生低有效输出

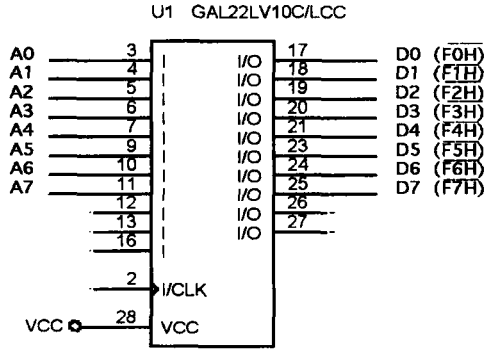


图 11-11 产生端口选择信号 F0H ~ F7H 的 PLD

11.2.2 译码 16 位 I/O 地址

PC 机系统向来用 16 位 I/O 地址,而在嵌入式系统中相对很少见 16 位端口地址。译码 16 位 I/O 地址与译码 8 位 I/O 地址的主要区别在于另外 8 个地址线(A_{15}~ A_8)必须被译码。图 11-12 给出了一个电路,它包含一个 PLD 和一个用于译码 I/O 端口 EFF8H ~ EFFFH 的 4 输入与非门。

与非门只译码部分地址(A_{15},A_{14},A_{13}与 A_{11}),因为 PLD 没足够的地址输入引脚。与非门的输出引脚连接到 PLD 的输入引脚 Z,并被译码为 I/O 端口地址的一部分。PLD 还为 I/O 端口$\overline{\text{EFF8H}}$~$\overline{\text{EFFFH}}$产生地址选通信号。PLD 的 VHDL 代码描述如例 11-3 所示。

例 11-3

```
-- 图 11-12 中译码器的 VHDL 代码描述
library ieee;
use ieee.std_logic_1164.all;
entity DECODER_11_12 is

port (
        Z, A12, A10, A9, A8, A7, A6, A5, A4, A3, A2, A1, A0: in STD_LOGIC;
        D0, D1, D2, D3, D4, D5, D6, D7: out STD_LOGIC
);
end;
architecture V1 of DECODER_11_12 is
begin
     D0 <= not (not Z and not A12 and A10 and A9 and A8 and A7 and A6 and A5 and A4 and A3
          and not A2 and not A1 and not A0);
     D1 <= not (not Z and not A12 and A10 and A9 and A8 and A7 and A6 and A5 and A4 and A3
          and not A2 and not A1 and    A0);
     D2 <= not (not Z and not A12 and A10 and A9 and A8 and A7 and A6 and A5 and A4 and A3
          and not A2 and A1 and not A0);
     D3 <= not (not Z and not A12 and A10 and A9 and A8 and A7 and A6 and A5 and A4 and A3
          and not A2 and A1 and A0);
     D4 <= not (not Z and not A12 and A10 and A9 and A8 and A7 and A6 and A5 and A4 and A3
          and A2 and not A1 and not A0);
```

```
        D5 <= not (not Z and not A12 and A10 and A9 and A8 and A7 and A6 and A5 and A4 and A3
              and A2 and not A1 and A0);
        D6 <= not (not Z and not A12 and A10 and A9 and A8 and A7 and A6 and A5 and A4 and A3
              and A2 and A1 and not A0);
        D7 <= not (not Z and not A12 and A10 and A9 and A8 and A7 and A6 and A5 and A4 and A3
              and A2 and A1 and A0);
end V1;
```

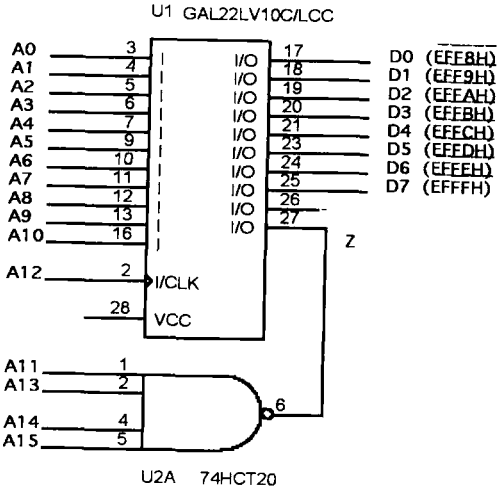


图 11-12 译码 16 位 I/O 端口 EFF8H～EFFFH 的 PLD

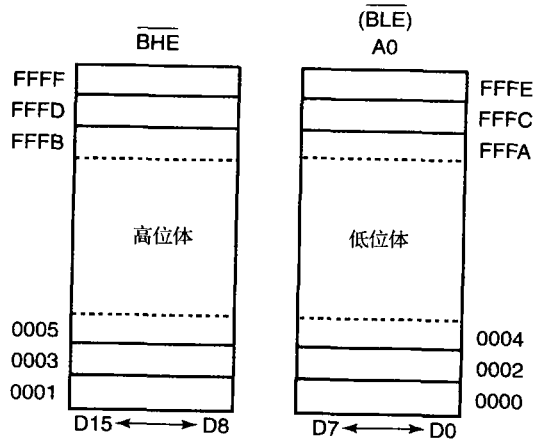


图 11-13 8086、80186、80286 及 80386SX 中的 I/O 体

11.2.3 8 位与 16 位 I/O 端口

既然明白了 I/O 端口的编址和寻址 I/O 端口比寻址存储器译码更简单(由于位数少),那么我们来解释微处理器与 8 位或 16 位 I/O 设备间的接口。在像 80386SX 的 16 位微处理器中,数据传送到一个I/O 存储体中的 8 位 I/O 设备。有 64K 个不同的 8 位端口,但只有 32K 个不同的 16 位端口,因为一个 16 位端口要用两个 8 位端口。正如存储器一样,I/O 系统包含 2 个 8 位 I/O 体。如图 11-13 所示,给出了诸如 80386SX 16 位系统的独立 I/O 体。

由于存在 2 个 I/O 体,所以任意 8 位 I/O 写操作需要一个独立的写选通才能正确操作。I/O 读操作不需要独立的读选通,正如存储器一样,微处理器只读它"期望"的那个字节而忽略另一字节。读操作惟一可能引起问题的时候是当 I/O 设备错误地响应一个读操作时。在 I/O 设备响应读操作是来自错误 I/O 体的情况下,可能需要独立的读信号。这种情况将在本章后面讨论。

图 11-14 给出了一个系统,它包含 2 个不同的 8 位输出设备,分别位于 8 位 I/O 地址 40H 和 41H 处。由于它们是 8 位设备且出现在不同的 I/O 体中,所以要产生独立的 I/O 写信号。注意,所有 I/O 端口均使用 8 位地址,因此端口 40H 和 41H 每个都可作为独立的 8 位端口被寻址,或者一起作为一个 16 位端口被寻址。用于图 11-14 的 PLD 译码器的程序见例 11-4。

例 11-4

```
-- 图 11-14 中译码器的 VHDL 代码描述

library ieee;
use ieee.std_logic_1164.all;

entity DECODER_11_14 is

port (
        BHE, IOWC, A7, A6, A5, A4, A3, A2, A1, A0: in STD_LOGIC;
        D0, D1: out STD_LOGIC
```

```
);
end;

architecture V1 of DECODER_11_14 is

begin

        D0 <= BHE or IOWC or A7 or not A6 or A5 or A4 or A3 or A2 or A1 or A0;
        D1 <= BHE or IOWC or A7 or not A6 or A5 or A4 or A3 or A2 or A1 or
              not A0;

end V1;
```

当选择16位宽的I/O设备时，$\overline{BLE}$(A_0)与$\overline{BHE}$引脚不起作用，因为2个I/O体被一起选中。尽管16位I/O设备相对少见，但确实存在一些模/数和数/模转换器，以及一些视频和磁盘存储器接口为16位I/O。

图11-15给出了一个16位输入器件，其8位I/O地址为64H和65H。注意，PLD译码器没有地址位$\overline{BLE}$(A_0)和$\overline{BHE}$引脚，因为这些信号在16位宽的I/O器件上不用。PLD译码器的程序见例11-5，它说明了用作输入器件的三态缓冲器(74HCT244)的使能信号是如何产生的。

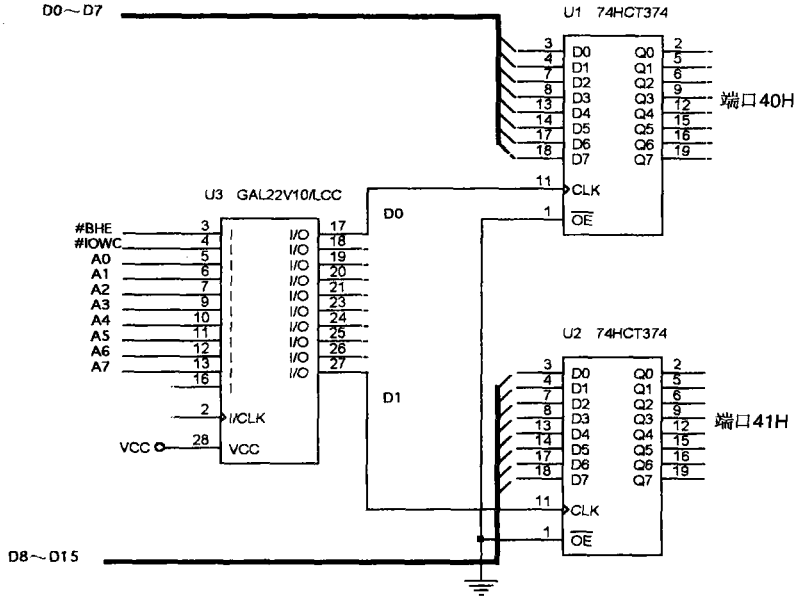


图11-14 选择端口40H和41H输出数据的I/O端口译码器

例11-5

```
-- 图11-15中译码器的VHDL代码描述

library ieee;
use ieee.std_logic_1164.all;

entity DECODER_11_15 is

port (
        IORC, A7, A6, A5, A4, A3, A2, A1: in STD_LOGIC;
        D0: out STD_LOGIC
);

end;
architecture V1 of DECODER_11_15 is

begin

      D0 <= IORC or A7 or not A6 or not A5 or A4 or A3 or not A2 or A1;

end V1;
```

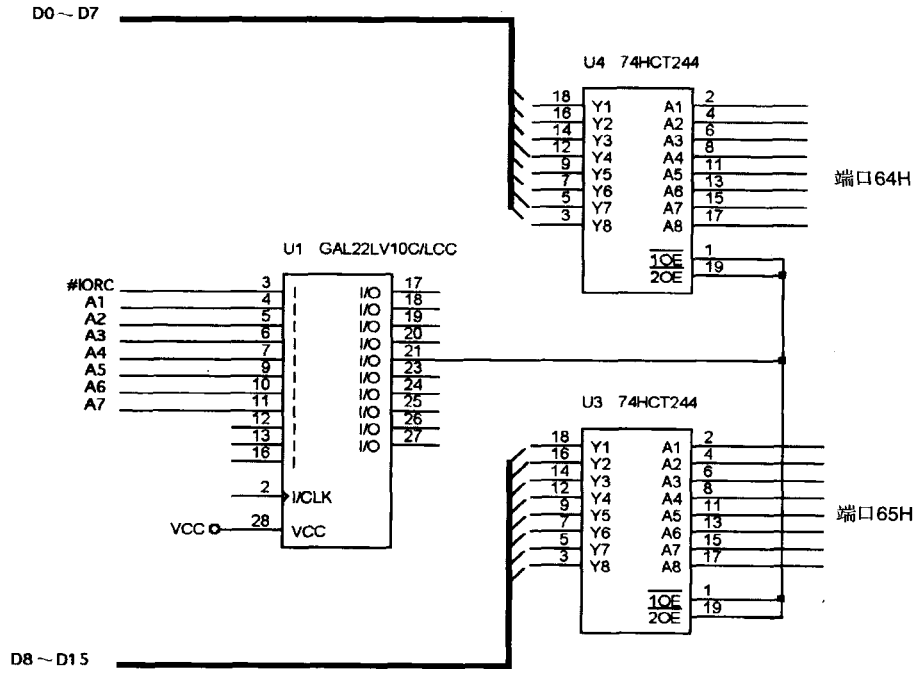


图 11-15 I/O 地址为 64H 和 65H 的一个 16 位 I/O 端口

11.2.4 32 位 I/O 端口

尽管 32 位宽的 I/O 端口不普遍,但它们也许最终会因为计算机系统新总线的出现而变得常见起来。曾经很有希望的 EISA 系统总线支持 32 位 I/O,VESA 局部总线和现在的 PCI 总线也是如此,但没有多少 I/O 设备是 32 位宽的。

图 11-16 中电路给出了 80386DX～80486DX 微处理器的一个 32 位输入端口。与早期的接口一样,此电路使用一个 PLD 译码 I/O 端口,使用 4 个 74HCT244 缓冲器将 I/O 数据与数据总线相连。此电路译码的 I/O 端口为 8 位端口 70H～73H,如例 11-6 中的 PLD 程序所示。再次提醒,这里只译码 8 位 I/O 端口地址。当编写软件来访问此端口时,对 32 位输入使用地址 70H,正如指令 IN EAX,70H 一样。

例 11-6

```
-- 图 11-16 中译码器的 VHDL 代码描述
library ieee;
use ieee.std_logic_1164.all;

entity DECODER_11_16 is

port (
       IORC, A7, A6, A5, A4, A3, A2: in STD_LOGIC;
       D0: out STD_LOGIC
);
end;
architecture V1 of DECODER_11_16 is

begin

      D0 <= IORC or A7 or not A6 or not A5 or not A4 or A3 or A2;

end V1;
```

对于 Pentium～Core2 微处理器及其 64 位数据总线,I/O 端口出现在不同的 I/O 体中,具体由I/O端

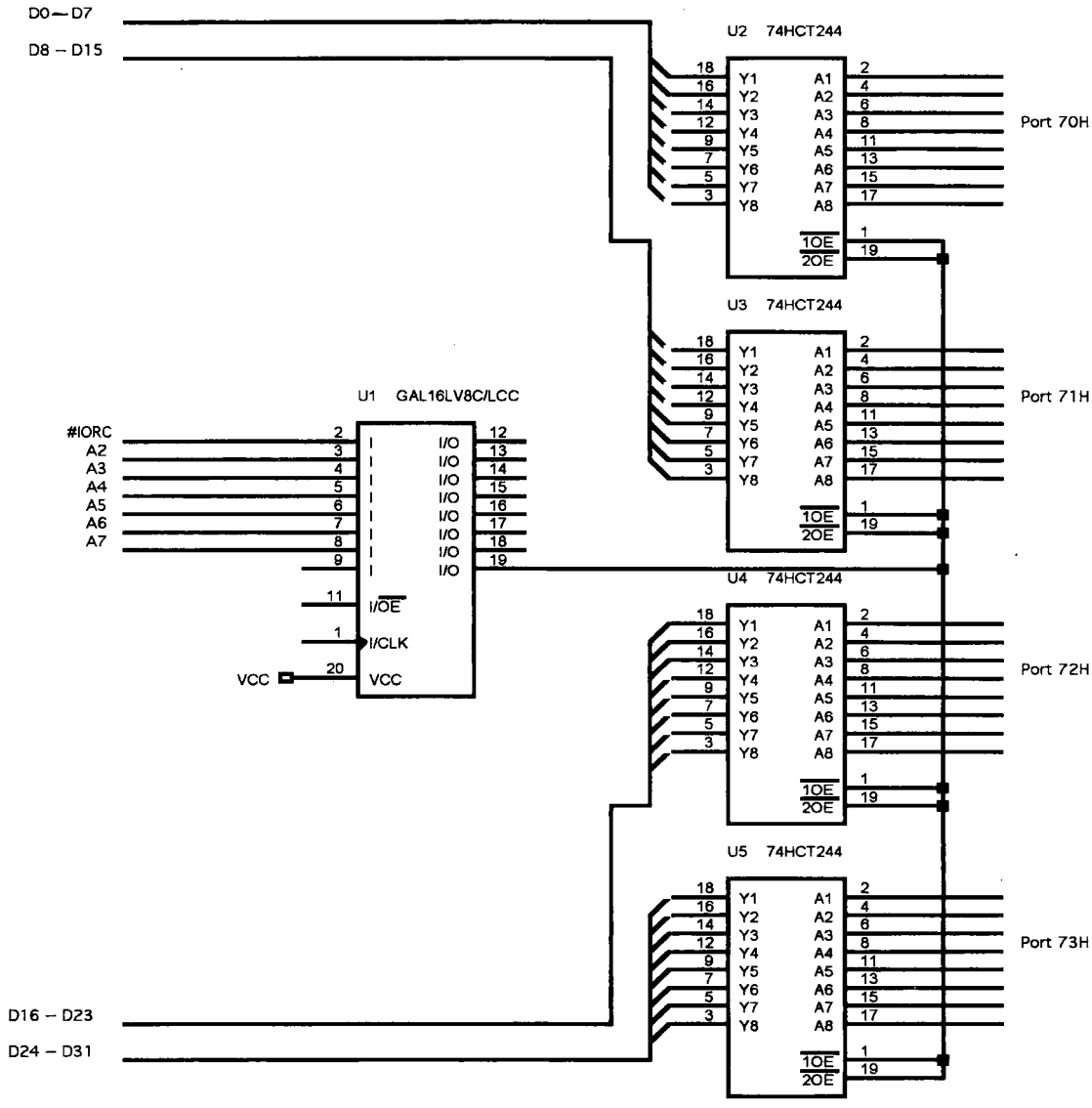


图 11-16 对 80486DX 微处理器地址为 70H～73H 的一个 32 位输入端口译码

口地址确定。例如,8 位 I/O 端口 0034H 出现在 Pentium 的 I/O 体 5 中,而 16 位 I/O 端口0034H～0035H 出现在 Pentium 的 I/O 体5 和6 中。Pentium 系统的32 位 I/O 访问可以出现在任意4 个连续的 I/O 体中,例如 32 位 I/O 端口 0100H～0103H 出现在 I/O 体 0～3 中。I/O 地址范围必须从最右两位为 0 的地址开始。例如,0100H～0103H 可以,但是 0101H～0104H 就不可以。

那么一个 64 位 I/O 设备是如何接口的呢? 最宽的 I/O 传送是 32 位,现在还没有 64 位 I/O 指令支持 64 位传送。这对于 Pentium 4 或 Core2 的 64 位模式是真实的情况。

假定我们需在 IO 端口 2000H 与 2001H 接入一个简单的 16 位输出端口。最低端口 2000H 地址最右端三位是 000。这意味着端口 2000H 在存储器的 0 地址块。同理,端口 2001H 的最右端三位是 001,这意味着端口 2001H 处于存储器地址块 1 中。图 11-17 给出了连接图,且在例 11-7 中给出了 PLD 的 VHDL 代码。

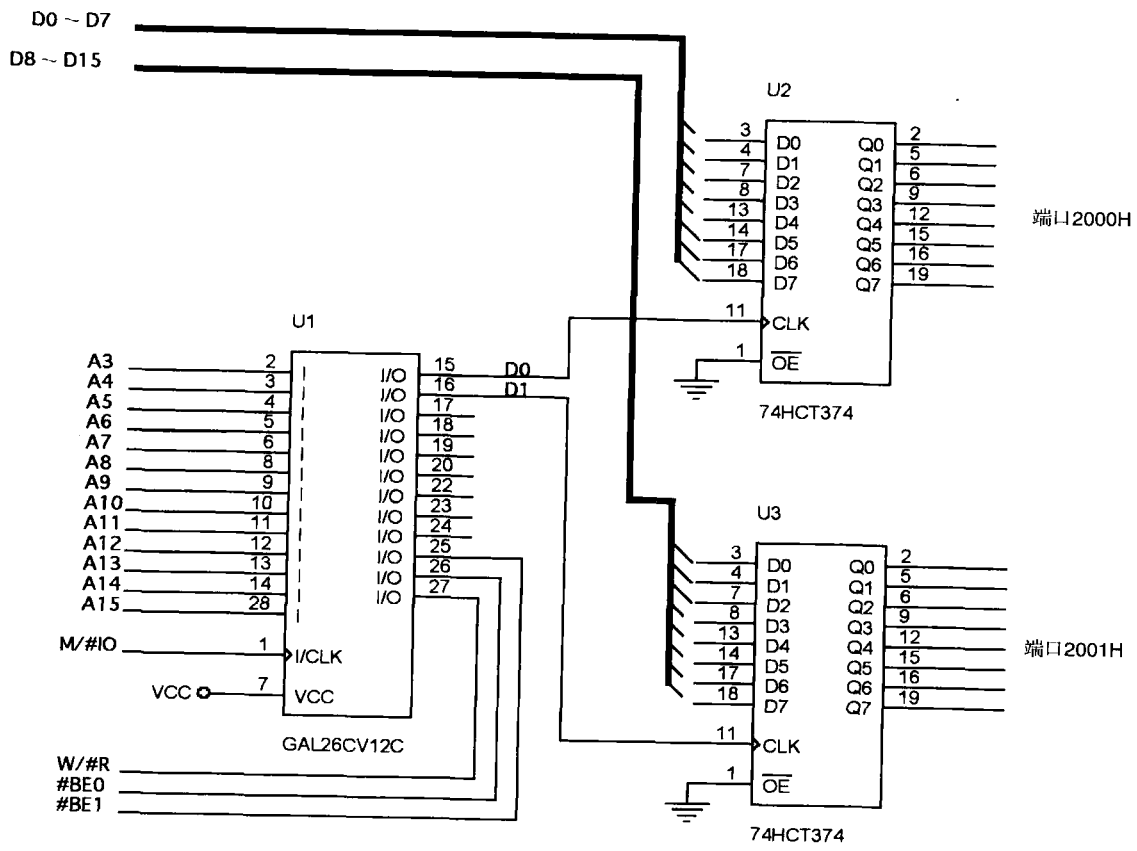


图 11-17 Pentium 4 接口到 16 位 I/O 端口,端口地址在 2000H 和 2001H

控制信号 $M/\overline{IO}$与 $W/\overline{R}$ 必须组合为锁存器产生一个 I/O 写信号,并且$\overline{BE0}$与$\overline{BE1}$块使能信号被用来控制写信号以校准地址 2000H(块 0)与 2001H(块 1)的锁存时钟。在连接中可能出现的惟一问题是,当 I/O 端口扩展到 64 位时,例如:16 位宽的端口(其端口地址是 2007H、2008H),在这种情形下,端口 2007H 与 2008H 分别使用的是地址块 7 与块 0,但二者被译码的地址是不同的:2007H 被译为 0010 0000 0000 0XXX,而 2008H 则被译为 0010 0000 0000 1XXX。因而最好避免这种情形发生。

例 11-7

```
-- 图 11-17 中译码器的 VHDL 代码描述

library ieee;
use ieee.std_logic_1164.all;

entity DECODER_11_17 is

port (
       MIO, BE0, BE1, WR, A15, A14, A13, A12, A11, A10, A9, A8, A7, A6, A5, A4,
             A3: in STD_LOGIC;
       D0, D1: out STD_LOGIC
);
end;

architecture V1 of DECODER_11_17 is

begin
```

```
    D0 <= MIO or BE0 or not WR or A15 or A14 or not A13 or A12 or A11 or A10
          or A9 or A8 or A7 or A6 or A5 or A4 or A3;
    D1 <= MIO or BE1 or not WR or A15 or A14 or not A13 or A12 or A11 or A10
          or A9 or A8 or A7 or A6 or A5 or A4 or not A3;

end V1;
```

11.3 可编程外围设备接口

82C55 **可编程外围设备接口**(**programmable peripheral interface**,**PPI**)是一个非常流行且低成本的接口器件。该 PPI 器件有 24 个引脚可用于 I/O,每组 12 个引脚可进行编程,以 3 种不同的操作方式工作。82C55 可将任一 TTL 兼容的 I/O 设备与微处理器相连接。82C55(CMOS 型)如果与使用高于 8MHz 时钟的微处理器一起工作,则需要插入等待状态。它还可以为每个输出提供至少 2.5mA 的吸收(逻辑 0)电流,最大为 4mA。由于 I/O 设备本来就慢,所以在 I/O 传送期间使用等待状态并不会显著影响系统的速度。82C55 甚至在最新的基于 Pentium 4 的计算机系统中仍有应用(尽管它可能不作为单独的 82C55 出现在系统中,但编程是兼容的)。在许多 PC 机中 82C55 被用作键盘和并行打印机端口的接口,但它还用在一个接口芯片集里,这个芯片集同时还控制定时器,并从键盘接口中读取数据。

我们可以用一个低成本的带有 8255 的实验板,将其插入 PC 机的并口,由此,便可访问实验板上的 8255。通过实验板自带的驱动程序即可用汇编语言或 VC++ 对 8255 编程。具体信息读者可访问网站:http://www.microdigitaled.com/hardware/mdelpt/MDELPT.htm。

11.3.1 82C55 基本描述

图 11-18 给出了 82C55 的引脚输出图。它的 3 个 I/O 端口(标识为 A、B 和 C)按组进行编程,A 组由

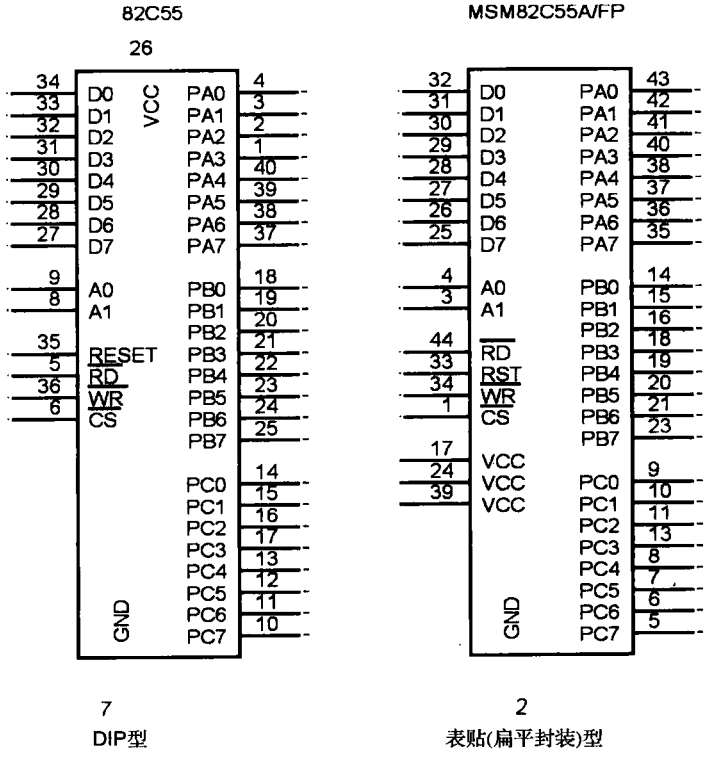


图 11-18 82C55 外围设备接口适配器(PPI)的引脚输出图

端口 A($PA_7 \sim PA_0$)和端口 C 的高半部分($PC_7 \sim PC_4$)组成,B 组由端口 B($PB_7 \sim PB_0$)和端口 C 的低半部分($PC_3 \sim PC_0$)组成。82C55 由$\overline{CS}$引脚选中,进行端口编程、读或写。寄存器选择是通过输入引脚 A_1 和 A_0 实现的,A_1 和 A_0 选择一个内部寄存器进行编程或操作。表 11-2 给出了用于编程和访问 I/O 端口的 I/O 端口分配。在 PC 机中,82C55 或其等价器件被译码在 I/O 端口 60H ~ 63H 及端口 378H ~ 37BH。

表 11-2 82C55 的 I/O 端口分配

A_1	A_0	功 能
0	0	端口 A
0	1	端口 B
1	0	端口 C
1	1	命令寄存器

82C55 是一个相当简单的,与微处理器相连接及编程的器件。为读或写 82C55,$\overline{CS}$输入必须为逻辑 0,而且正确的 I/O 地址必须加到 A_1 和 A_0 引脚上。其余的端口地址对 82C55 而言为无关项,而且它们是在外部译码的,以便选择 82C55。

图 11-19 给出了与 80386SX 相连的一个 82C55,其 8 位 I/O 端口地址为 C0H(端口 A)、C2H(端口 B)、C4H(端口 C)及 C6H(命令寄存器)。该接口使用了 80386SX I/O 映像的低位 I/O 体。注意,从该接口到所有 82C55 引脚均直接与 80386SX 相连,除$\overline{CS}$引脚外。$\overline{CS}$引脚是由 74ALS138 译码器译码并选择的。

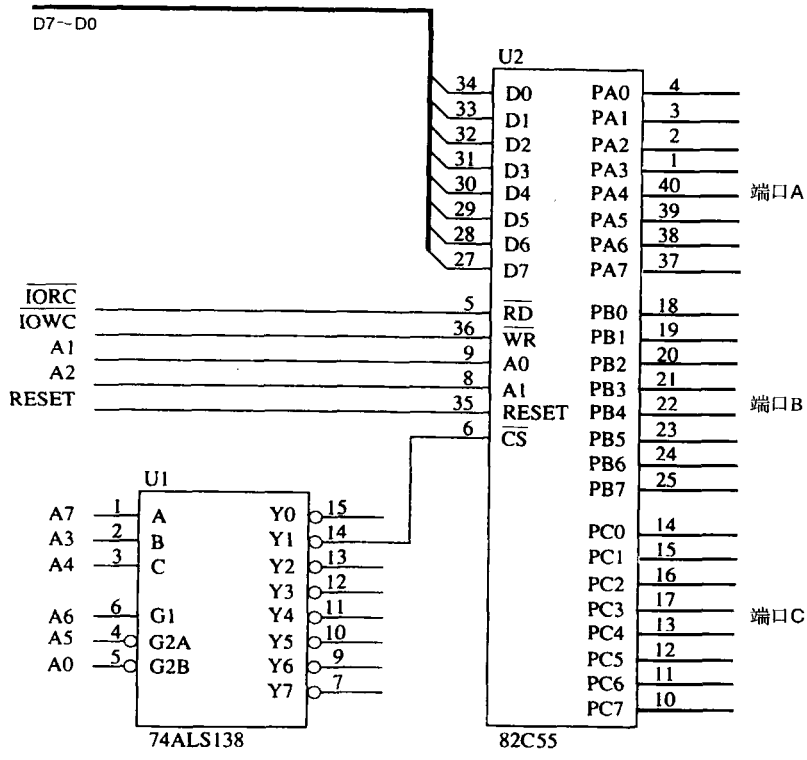


图 11-19 与 80386SX 微处理器的低位 I/O 体连接的 82C55

一旦微处理器复位,输入到 82C55 的 RESET 信号就使器件初始化,使得所有端口设置成为用方式 0 操作的简单输入端口。由于端口引脚在 RESET 后被内部编程为输入引脚,所以当系统加电时避免了损坏。在 RESET 之后,只要 82C55 所有 3 个端口都用作输入,就不需要其他命令来编程 82C55。注意 82C55 连接到 PC 机上时其端口地址为 60H ~ 63H,用于控制键盘、扬声器、定时器以及其他内部设备,如存储器扩充。与此同时,I/O 端口 378H ~ 37BH 也用于并行打印机的端口。

11.3.2 82C55 编程

如图 11-20 所示,82C55 是通过 2 个内部命令寄存器进行编程的。注意,位 7 选择命令字节 A 或命令

字节B。命令字节A对A组和B组的功能进行编程，而命令字节B只有在82C55被编程为方式1或方式2时对端口C进行置位(1)或复位(0)。

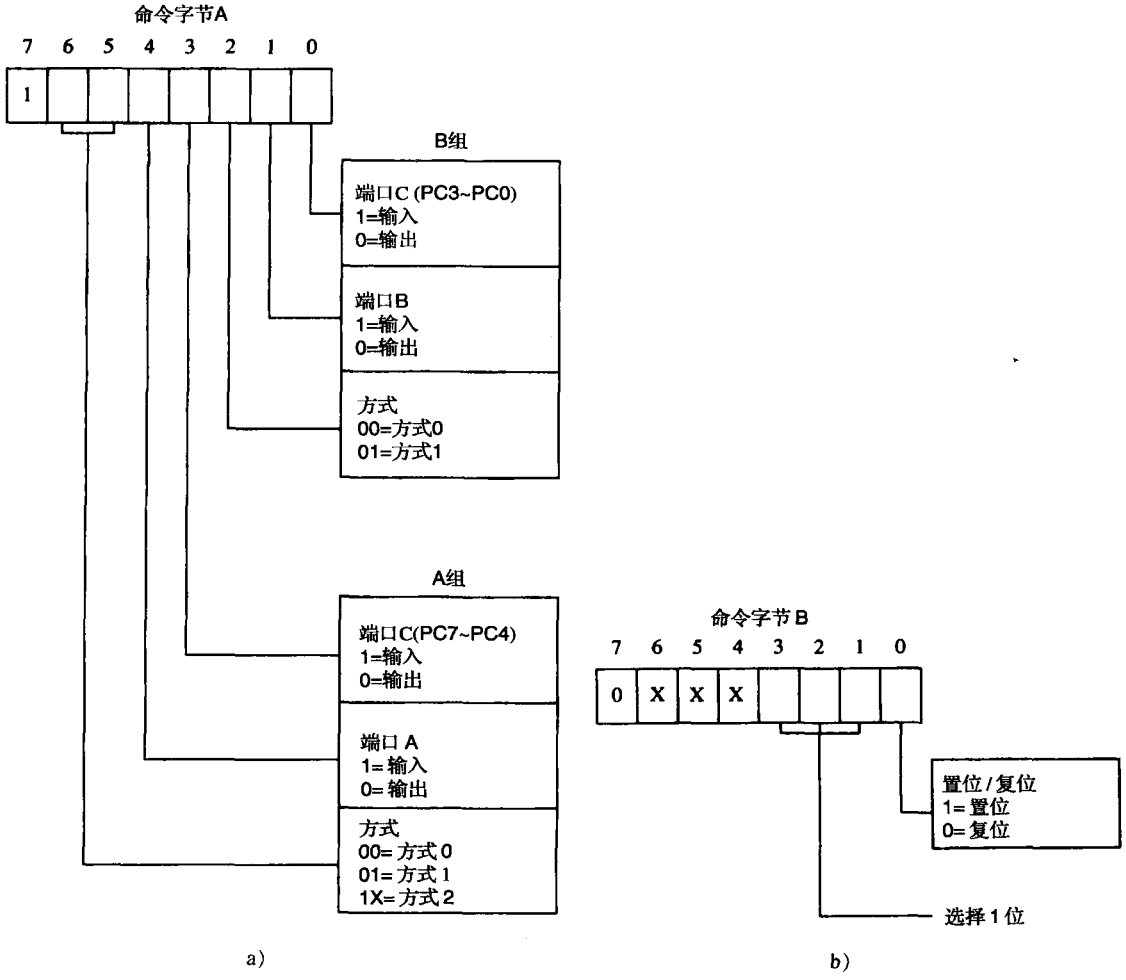


图11-20 82C55中命令寄存器的命令字节

a)编程端口A、B和C b)对选择的位进行置位或复位

B组的引脚(端口C的低位部分及端口B)可编程为输入或输出引脚，B组可工作在方式0或方式1下。方式0是基本输入/输出方式，它允许B组的引脚被编程为简单输入和锁存的输出引脚。方式1操作是B组的选通操作方式，数据通过端口B传送，握手信号由端口C提供。

A组的引脚(端口C的高位部分及端口A)可编程为输入或输出引脚，与B组的区别在于：A组可工作在方式0、方式1和方式2下。方式2操作是端口A的双向操作方式。

如果命令字节的第7位被置0，则选中命令字节B。当82C55工作在方式1或方式2时，此命令允许端口C的任意一位被置位(1)或复位(0)，否则不使用该命令字节进行编程。在控制系统中常使用位的置位/复位功能来置位或清除端口C的控制位。位的置位/复位功能可防止误操作，这意味着在位的置位/复位命令期间，端口C的其他引脚不会改变。

11.3.3 方式0操作

方式0操作使82C55或者作为一个经过缓冲的输入设备，或者作为一个经过锁存的输出设备工作。这与本章第一节中讨论的基本输入和输出电路是相同的。

图 11-21 给出了 82C55 与一组 8 个 7 段码 LED 显示器相连的电路。这是标准的 LED,但通过修改电阻值可以连接有机 LED(OLED)或高亮度 LED。此电路中,端口 A 和 B 均被编程为(方式 0)简单的锁存输出端口。端口 A 提供段数据输入给显示器,端口 B 给多路复用显示器提供一次选择一个显示位置的方式。82C55 通过 PLD 与 8088 微处理器相连接,其 I/O 端口号为0700H～0703H。对 PLD 的编程在例 11-8 中给出。该 PLD 器件译码 I/O 地址,并为 82C55 的$\overline{WR}$引脚产生写选通信号。

例 11-8

```
-- 图 11-21 中译码器的 VHDL 代码描述

library ieee;
use ieee.std_logic_1164.all;

entity DECODER_11_21 is

port (
       IOM, A15, A14, A13, A12, A11, A10, A9, A8, A7, A6, A5, A4, A3,
              A2: in STD_LOGIC;
       D0: out STD_LOGIC
);

end;

architecture V1 of DECODER_11_17 is

begin

       D0 <= not IOM or A15 or A14 or A13 or A12 or A11 or not A10
       or not A9 or not A8 or A7 or A6 or A5 or A4 or A3 or A2;

end V1;
```

选择图 11-21 中的电阻阻值,使段电流为 80mA。当显示器多路复用时,要求产生的每段平均电流为 10mA。一个 6 位数字显示器要用 60mA 的段电流,每段平均 10mA。在这种显示系统中,在任何给定时刻 8 个显示位置只有一个位置发光。8 位数字显示器的阳极峰值电流为 560mA(7 段 × 80mA),但平均阳极电流为 80mA。在 6 位数字显示器中,其峰值电流将为 420mA(7 段 ×60mA)。一旦显示器多路复用,我们就把段电流从 10mA(显示器使用每段 10mA 作为额定电流)增加为显示位数与 10mA 的乘积。这意味着一个 4 位数字显示器使用每段 40mA 电流,一个 5 位数字显示器使用每段 50mA 电流,依次类推。

在这个显示器里,段负载电阻上通过 80mA 电流并约有 3.0V 的电压降。LED 额定电压为 1.65V,通过阳极开关和段开关后会降低十分之几伏,因此需要 3.0V 电压通过段负载电阻。电阻值为3.0V/80mA = 37.5Ω。则图 11-21 中使用最接近的标准电阻值 39Ω 作为段负载电阻。

考虑与段开关基极串联的电阻,假定晶体管的最小增益是 100,那么基极电流为 80mA/100 = 0.8mA。通过基极电阻的电压大约是 3.0V(82C55 逻辑 1 的最小电压电平)减去经过发射极 - 基极的压降(0.7V),即为 2.3V。则基极电阻值为 2.3V/0.8mA = 2.875kΩ。最接近的标准电阻值是 2.7kΩ,但本电路选择的是 2.2kΩ。

阳极开关在其基极有一个电阻,由于晶体管的最小增益是 100,所以通过此电阻的电流为 560mA/100 = 5.6mA。它超过了 82C55 的最大电流 4.0mA,但还是比较小,工作时不会出现问题。假如你正使用端口引脚作为另一电路的 TTL 输入,则需要考虑最大电流的问题。如果电流值超过 8.0mA～10.0mA,那么就需要适当的电路(或者是达林顿复合晶体管,或者是另一种晶体管开关)。这里,通过基极电阻的电压是 5.0V 减去发射极 - 基极的压降(0.7V),再减去端口引脚为逻辑 0 电平时的电压(0.4V),则电阻值为 3.9V/5.6mA = 696Ω。最接近的标准电阻值是690Ω,本例选择了该值。

在检查显示器软件的运行情况之前,必须首先编程 82C55。这是由例 11-9 中列出的几条短指令实现的。这里,端口 A 和端口 B 均被编程为输出。

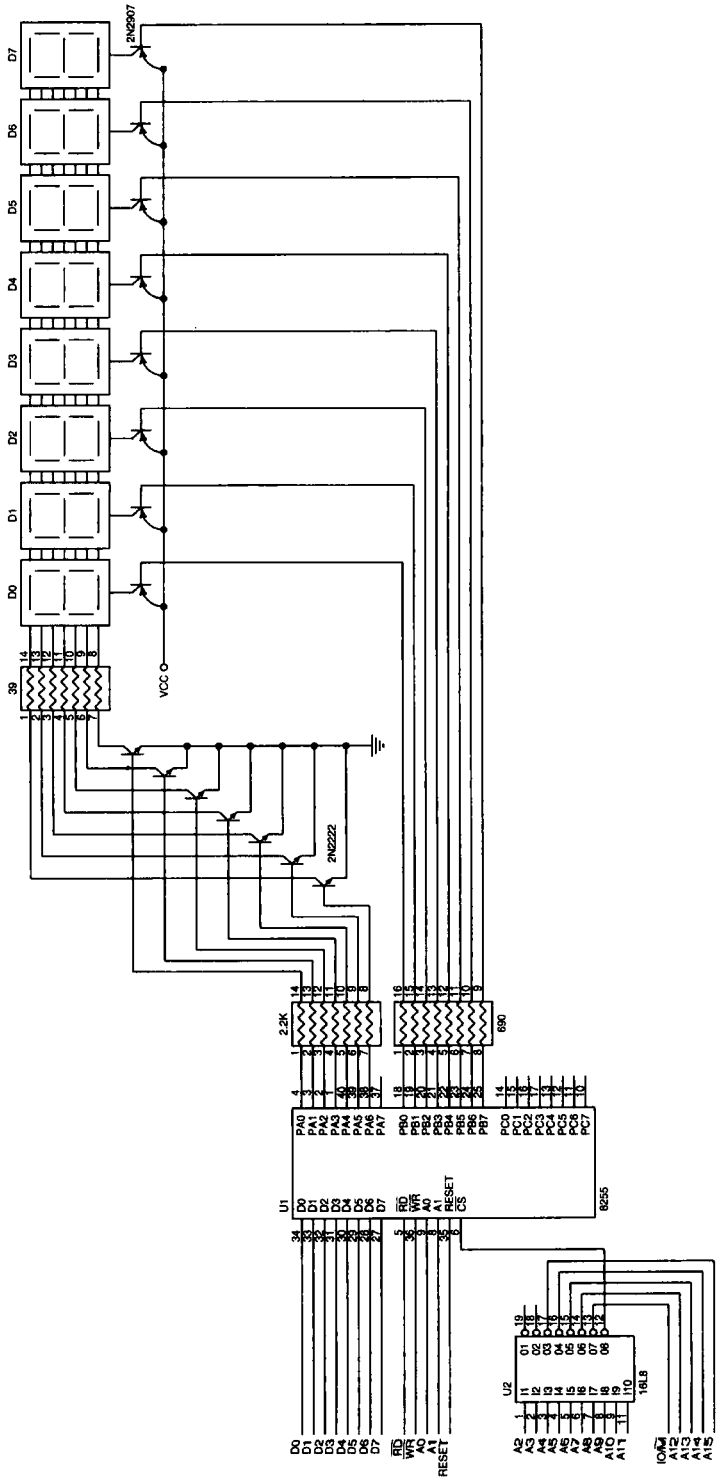


图11-21 一个8位LED显示器通过82C55的PIA连接到8088微处理器上

例 11-9

```
;_编程设置82C55 PIA

        MOV   AL,10000000B      ;命令
        MOV   DX,703H           ;地址端口703H
        OUT   DX,AL             ;发送命令到端口703H
```

驱动多路复用显示器的过程如例 11-10 所示，既可以使用汇编语言，也可以使用 C++ 和汇编语言。为使显示器系统正确工作，必须经常调用此过程。注意，此过程调用了产生 1ms 时间延迟的另一过程(DELAY)。本例中没有描述时间延迟，但它被用来允许每个显示位置发光的时间。LED 显示器的制造商建议显示器闪烁频率在 100Hz 到 1500Hz 之间。使用 1ms 的时间延迟，每 1ms 显示一个数字，总的显示器闪烁速率为 1000/8Hz，即所有 8 个数字闪烁速率为 125Hz。

例 11-10

```
; 多路复用 8 位 LED 显示的汇编语言程序；
; 此程序必须经常调用以保证正常显示。


DISP    PROC  NEAR USES AX BX DX SI

        PUSHF
        MOV   BX,8                  ;装入计数器
        MOV   AH,7FH                ;装入选择模板
        MOV   SI,OFFSET MEM-1       ;寻址显示数据
        MOV   DX,701H               ;寻址端口 B

;display all 8 digits

        .REPEAT
              MOV   AL,AH           ;将选择模式发送到端口 B
              OUT   DX,AL
              DEC   DX
              MOV   AL,[BX+SI]      ;将数据发送到端口 A
              OUT   DX,AL
              CALL  DELAY           ;等待 1 毫秒
              ROR   AH,1            ;调整选择模式
              INC   DX
              DEC   BX              ;计数器减 1
        .UNTIL BX == 0

        POPF
        RET

DISP    ENDP

// 使用字符数组 MEM 多路复用 8 位显示器的 C++ 函数


void Disp()
{
        unsigned int *Mem = &MEM[0];        // 指向数组 0 单元
        for ( int a = 0; a < 8; a++ )
        {
              unsigned char b = 0xff ^ ( 1 << a );        // 形成选择模式
              _asm
              {
                    mov   al,b
                    mov   dx,701H
                    out   dx,al      ;将选择的模式发到端口 B
                    mov   al,Mem[a]
                    dec   dx
                    out   dx,al      ;将数据发送到端口 A
              }
```

```
            Sleep(1);                ; 等待 1.0ms
        }
}
```

显示过程(DISP)寻址一个存储器区域,该区域中存储了称为 MEM 的 8 个显示器数字的 7 段码数据。AH 寄存器装入一个选择代码(7FH),开始寻址最高有效显示位置。一旦该位置被选中,则存储单元 MEM +7 被寻址,其中的内容被发送给最高有效数字。然后调整选择码以选择下一显示数字,地址也做同样的调整。这一过程重复 8 次,从而在 8 个显示数字上显示存储单元 MEM 到 MEM +7 中的内容。

可以通过写一个使用系统时钟决定每条指令执行时间的过程来产生 1.0 ms 的延时。例 11-11 中列出这个过程,通过执行一定次数的 LOOP 循环指令来产生一段时间的延时。这里先用 XXXX 表示,当讨论了一些因素后,再给出具体的值。LOOP 指令的执行需要一定数目的时钟,具体需要的数值,可以在附录 B 中找到。假定这个接口使用了 20MHz 时钟的 80486 微处理器,附录 B 中指出 LOOP 指令执行要用 7/6 个时钟周期。第一个数值是指跳转到 D1 所需要的时钟周期数,第二个数值是无跳转所需要的时钟周期数。对于 20MHz 的时钟频率,一个周期是 1/20MHz = 50ns。在这种情形下,LOOP 指令需要 350ns 执行所有的迭代(最后一个迭代除外)。要确定实现 1.0ms 延时所需的 XXXX 值,只要用 1.0ms 除以 350ns,XXXX = 2.857。如果需要一个更大值的 XXXX,可以使用 LOOPD 指令和 ECX 寄存器。执行 MOV CX,XXXX 来获得所需的时间,RET 指令通常可以忽略不计。

例 11-11

```
;延时的等式
;
;
;XXXX = 延迟时间
        LOOP 迭代执行时间;
;
;

DELAY   PROC   NEAR USES CX

        MOV CX,XXXX
D1:
        LOOP D1
        RET

DELAY   ENDP
```

假定使用 2.0GHz 时钟的 Core2 来产生这个延时。这里一个时钟周期是 0.5ns,每一个 LOOP 迭代需要 5 个时钟,那么 XXXX 的值大约为 400 000,这时就可以将 LOOPD 结合 ECX 来使用。

如果程序是为 Windows 环境所写,比如在 Windows 的嵌入式系统中应用,那么时间延迟就可以使用定时器。定时器可以精确到毫秒,这样,在嵌入式 Windows 中,延时就可以得到保证。

11.3.4 与 82C55 接口的 LCD 显示器

LCD(liquid crystal display,液晶显示器)在许多应用上取代了 LED 显示器,但它的惟一缺点是在光线较弱的情形下难以看清,因此在这种情形下仍然使用 LED。例如视力差的老年人使用的医疗设备。如果 OLED(有机发光二极管显示器)价格降到一定程度,LCD 显示器将会淘汰。目前一家德国公司生产的 OLED 显示屏售价低于 10 美元。

图 11-22 给出了 Optrex DMC - 20481 LCD 显示器与 82C55 的连接。DMC - 20481 是一个 4 行 ×20 个字符/行的显示器,它接收 ASCII 码作为输入数据,还接收对它进行初始化和控制其应用的命令。正如图 11-22 所示的,LCD 显示器引脚不多。与 82C55 端口 A 相连的数据引脚用于输入显示数据和从显示器中读出信息。图中给出的是一个 8 位的接口。如果需要一个 4 位的接口,那么只要 D4 ~ D7 引脚用作数据线,需要注意的是数据的格式必须是高 4 位在前,低 4 位在后。此外,少数新型的 OLED 设备还包括一个串口,在此串口中数据使用单一引脚。

该显示器上有 4 个控制引脚。V_{EE}引脚用于调节 LCD 显示器的对比度,而且通常连接到一个 10kΩ

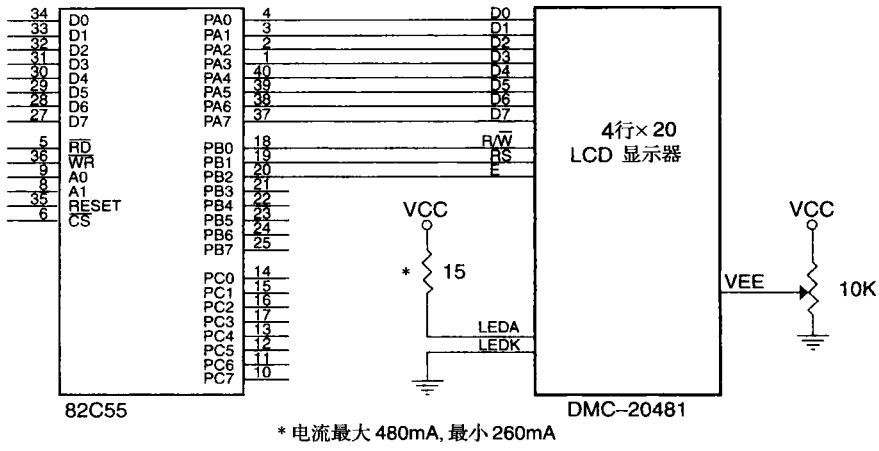


图 11-22 与 82C55 连接的 DMC-20481 LCD 显示器

的电位器上,如图 11-22 所示。RS(寄存器选择)输入用于选择数据(RS = 1)或指令(RS = 0)。DMC - 20481 读或写信息时 E(允许)输入必须为逻辑 1。最后,R/$\overline{W}$ 引脚选择一次读或写操作。通常 RS 引脚被置为 1 或 0,R/$\overline{W}$ 引脚被置位或清零,数据被置于数据输入引脚上,然后 E 引脚被加以脉冲,从而存取 DMC - 20481。此显示器还有 2 个输入用于背景发光的 LED 二极管,图中没有给出。

为编程 DMC - 20481,首先必须对它进行初始化。任何使用 HD44780(日立公司)显示驱动器集成电路的显示器都应先初始化。Optrex 的小显示面板上的一整行以同样的方式被编程。初始化是通过下列步骤完成的:

1)在 V_{CC} 上升到 5.0V 后至少等待 15ms。

2)输出功能设置命令(30H),并至少等待 4.1ms。

3)第 2 次输出功能设置命令(30H),并至少等待 100μs。

4)第 3 次输出功能设置命令(30H),并至少等待 40μs。

5)第 4 次输出功能设置命令(38H),并至少等待 40μs。

6)输出 08H 禁止显示器,并至少等待 40μs。

7)输出 01H 使光标返回原位置并清除显示,然后至少等待 1.64ms。

8)输出允许显示器光标关闭(0CH),并至少等待 40μs。

9)输出 06H 选择自动加 1,移动光标,并至少等待 40μs。

完成 LCD 显示器初始化的软件在例 11-12 中给出。该程序很长,但显示控制器需要这样长的初始化对话过程。注意,这里没有给出 3 次时间延迟的程序。如果与 PC 机连接,那么可使用在 Pentium 那章讨论的 RDTSC 指令作为时间延迟。如果你正为其他应用开发此接口,则必须写 3 个独立的时间延迟,以提供初始化对话中指出的延迟时间。在 C++ 中可以使用计时器获得时间延迟。

例 11-12

```
PORTA_ADDRESS    EQU 700H          ;设置端口地址
PORTB_ADDRESS    EQU 701H
COMMAND_ADDRESS EQU 703H

;宏指令发送命令或数据到 LCD 显示器中
;
SEND    MACRO   PORTA_DATA, PORTB_DATA, DELAY

        MOV   AL,PORTA_DATA        ;送入端口 A
        MOV   DX,PORTA_ADDRESS
        OUT   DX,AL
        MOV   AL,PORTB_DATA        ;送入端口 B
        MOV   DX,PORTB_ADDRESS
        OUT   DX,AL
```

```
        OR    AL,00000100B          ;设置E位
        OUT   DX,AL                 ;送入端口B
        AND   AL,11111011B          ;E位清零
        NOP                         ;产生一个小的延迟
        NOP
        OUT   DX,AL                 ;送入端口B
        MOV   BL,DELAY              ;BL=延迟时间数
        CALL  MS_DELAY              ;时间延迟(ms)
        ENDM

;初始化LCD显示器的程序

START:
        MOV   AL,80H                ;对82C55进行编程
        MOV   DX,COMMAND_ADDRESS
        OUT   DX,AL

        MOV   AL,0
        MOV   DX,PORTB_ADDRESS      ;端口B清零
        SEND  30H, 2, 16            ;送30H延时16 ms
        SEND  30H, 2, 5             ;送30H延时5 ms
        SEND  30H, 2, 1             ;送30H延时1 ms
        SEND  38H, 2, 1             ;送38H延时1 ms
        SEND  8, 2, 1               ;送8延时1ms
        SEND  1, 2, 2               ;送1延时2 ms
        SEND  0CH, 2, 1             ;送0CH延时 1 ms
        SEND  6, 2, 1               ;送6延时1 ms
```

在SEND宏中加入了NOP指令,从而确保E位保持逻辑1的时间足够长,以激活LCD显示器。这一进程在大多数时钟频率下的大多数系统中都可正常工作,但在某些情况下可能需要增加NOP指令来延长这个时间。

在编程送给显示器数据之前,必须解释一下初始化对话中使用的命令。参见表11-3,它是LCD显示器的命令或指令的完整列表。试将初始化程序中发送给LCD显示器的命令与表11-3进行比较。

表11-3 大多数LCD显示器的指令

指 令	代 码	说 明	执行时间
清除显示	0000 0001	清除显示,并使光标返回原位置	1.64ms
光标返回原位置	0000 0010	使光标返回原位置	1.64ms
登录模式设置	0000 00AS	设置光标移动方向(A=1加1)及移动(S=1移动)	40 μs
显示器开/关	0000 1DCB	设置显示器开/关(D=1开)(C=1光标亮)(B=1光标闪烁)	40 μs
光标/显示移位	0001 SR00	设置光标移动和显示移位(S=1移位显示)(S=0移动光标)(R=1向右)	40 μs
功能设置	001L NF00	编程LCD电路(L=1 8位接口)(N=1 2行)(F=1 5×10字符,F=0 5×7字符)	40 μs
设置CGRAM地址	01XX XXXX	设置字符发生器RAM的地址	40 μs
设置DRAM地址	10XX XXXX	设置显示RAM的地址	40 μs
读"忙"标志	B000 0000	读"忙"标志(B=1忙)	0
写数据	Data	写数据给显示器或字符发生器RAM	40 μs
读数据	Data	从显示器或字符发生器RAM读出数据	40 μs

一旦LCD显示器被初始化,就需要几个过程来显示信息和控制显示器。初始化后,当给显示器发送数据或许多命令时,就不再需要时间延迟。清除显示命令仍然需要时间延迟,这是因为该命令未使用"忙"标志。与时间延迟不同,测试"忙"标志以检查显示器是否已完成一次操作。测试"忙"标志的过程如例11-13所示。BUSY过程测试LCD显示器,而且只有当显示器完成了前一个指令时才返回。

例 11-13

```
PORTA_ADDRESS    EQU 700H     ;设置端口地址
PORTB_ADDRESS    EQU 701H
COMMAND_ADDRESS  EQU 703H

BUSY    PROC   NEAR USES DX AX

        PUSHF
        MOV   DX,COMMAND_ADDRESS
        MOV   AL,90H              ;设置端口 A 为 IN 模式
        OUT   DX,AL
        .REPEAT
                MOV   AL,5        ;选择从 LCD 读取数据
                MOV   DX,PORTB_ADDRESS
                OUT   DX,AL       ;和脉冲 E
                NOP
                NOP
                MOV   AL,1
                OUT   DX,AL
                MOV   DX,PORTA_ADDRESS
                MOV   AL,DX       ;读取忙指令
                SHL   AL,1
        .UNTIL  !CARRY?           ;直到不忙
        NOV   DX ,COMMAND_ADDRESS
        MOV   AL ,80H
        OUT   DX ,AL              ;设置端口 A 为 OUT 模式
        POPF
        RET

BUSY    ENDP
```

一旦有了 BUSY 过程,通过编写另一个被称为 WRITE 的过程就可发送数据给显示器。在试图给显示器写新数据之前,WRITE 过程使用 BUSY 进行测试。例 11-14 给出了 WRITE 过程,它将 ASCII 字符从 BL 寄存器中传送到显示器的当前光标位置。注意,初始化对话按自动加 1 方法移动光标,所以如果调用 WRITE 超过一次,则写给显示器的字符就会一个挨一个地显示,就像在视频显示器上显示一样。

例 11-14

```
WRITE  PROC    NEAR
       MOV   AL,BL                    ;将 BL 送入端口 A 中
       MOV   DX,PORTA_ADDRESS
       OUT   DX,AL
       MOV   AL,0                     ;写 ASCII
       MOV   DX,PORTB_ADDRESS
       OUT   DX,AL
       OR    AL,00000100B             ;设置 E 位
       OUT   DX,AL                    ;送入端口 B
       AND   AL,11111011B             ;E 位清零
       NOP                            ;小延时
       NOP
       OUT   DX,AL                    ;送入端口 B
       CALL BUSY                      ;等待完成
       RET
WRITE  ENDP
```

基本显示操作还需要的一个过程是清屏和使光标返回原位置的过程 CLS,如例 11-15 所示。使用 CLS 及前面提到的过程,可以在显示器上显示任何信息、清除它并显示另一信息以及对显示器进行基本的操作。正如前面提到的,清除命令需要一个时间延迟(至少 1.64ms)而不是调用 BUSY 以进行正确的操作。

例 11-15

```
CLS     PROC    NEAR
        SEND    1, 2, 2
        RET
CLS     ENDP
```

可以设计另外的过程来指向一个显示 RAM 的位置。显示 RAM 的地址始于 0，并随着显示不断递增，直到第 1 行的最后一个字符的地址为存储单元 19，存储单元 20 是第 2 行的头一个字符的显示位置，依此类推。一旦可以移动显示地址，就可以改变显示器上的每个单独的字符，甚至从显示器上读取数据。如果需要，这些过程可由读者自行设计完成。

这里谈谈关于在 LCD 显示器内显示 RAM 的内容。LCD 包含 128 字节的存储器，地址为 00H～7FH。并非该存储器的所有存储单元都被使用。例如，1 行×20 个字符的显示器只使用存储器的开始 20 个字节(00H～13H)。任何显示器的第 1 行总是始于地址 00H。由 HD44780 驱动的任一显示器的第 2 行总是开始于地址 40H。例如，一个 2 行×40 个字符的显示器使用地址 00H～27H 来存储第 1 行的 ASCII 编码数据。第 2 行数据存储于地址 40H～67H 中。在 4 行×20 个字符的显示器中，第 1 行地址是 00H，第 2 行是 40H，第 3 行是 14H，最后一行是 54H。使用 HD44780 的最大的显示器件是一个 2 行×40 个字符的显示器，4 行×40 个字符的显示器使用 1 个 M50530 或 2 个 HD44780。关于这些器件的信息很容易在 Internet 上查到，因此这里不做介绍。

与 82C55 连接的步进电机

常常连接到计算机系统上的另一设备是步进电机。步进电机是数字电机，因为当它在 360 内旋转时，是不连续地步进移动的。一个普通的、便宜的步进电机大约每步移动 15°，而一个较贵的、高精度的步进电机每步移动 1°。所有步进电机都是通过许多磁极与/或传动装置来移动的。注意，图 11-23 中 2 个线圈被通电，如果所需功率较低，则每次可以只给一个线圈通电，使电机以每步 45°、135°、225°和 315°移动。

图 11-23 给出了使用一个单极电枢的 4 线圈步进电机。注意，步进电机随着电枢（永久磁铁）旋转到 4 个不同的位置被显示了 4 次。这是通过给线圈通电完成的，如图所示。该图是对全步进方式运转的描述。步进电机是通过使用 NPN 达林顿放大器给每个线圈提供大电流驱动的。

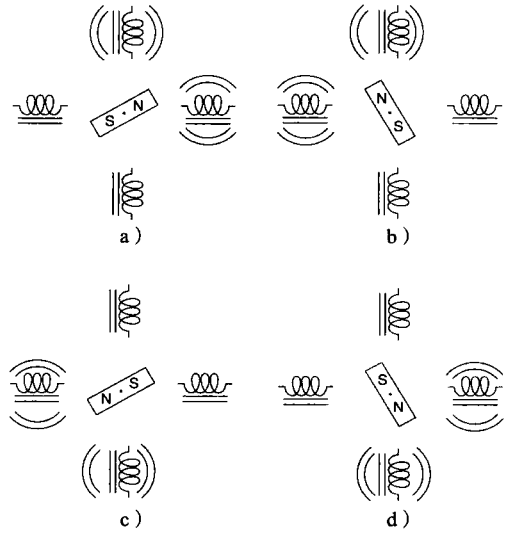


图 11-23 显示全步进运转的步进电机
a)45° b)135° c)225° d)315°

图 11-24 给出了可驱动该步进电机的一个电路，它有 4 个线圈。此电路使用 82C55 提供驱动信号，使

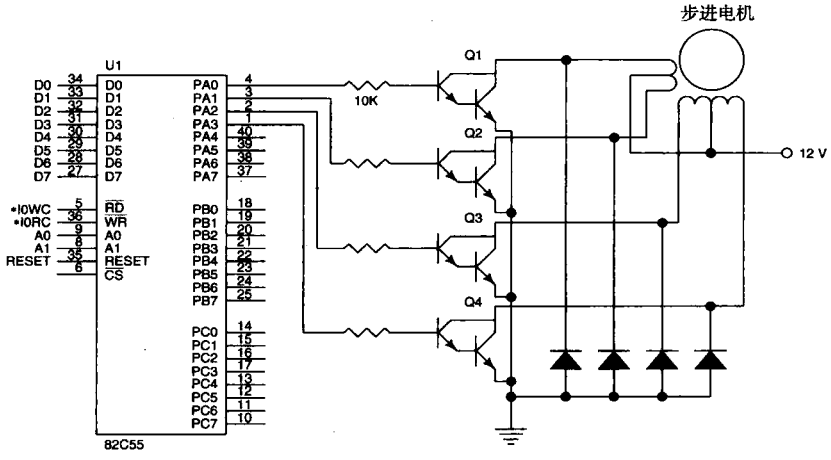


图 11-24 与 82C55 接口的步进电机（本图未给出译码器）

* 低有效。

电机的电枢按右手方向或左手方向旋转。

驱动该电机（假定端口 A 被编程为方式 0，作为一个输出设备）的简单过程见例 11-16，包括用汇编语言和 C++ 函数。该子程序被调用时，CX 保持步进数和旋转方向。如果 CX > 8000H，则电机按右手方向旋转；若 CX < 8000H，则电机按左手方向旋转。例如，步进数是 0003H，则电机向左转 3 步；如果步进数是 8003H，则电机向右转 3 步。去掉 CX 最左边一位，其余 15 位是步进数。注意，该过程使用一个时间延迟（未给出）来产生 1ms 的时间延迟。这个时间延迟允许步进电机电枢有足够的时间移动到下一个位置。

例 11-16

```
       PORT    EQU    40H

; 一个控制步进电机的汇编语言过程

STEP   PROC    NEAR USES CX AX

       MOV    AL,POS                          ;获得位置
       OR     CX,CX                           ;设置标志位
       IF   !ZERO?
              .IF !SIGN?                      ;如果无信号
                     .REPEAT
                         ROL   AL,1           ;向左旋转
                         OUT   PORT,AL
                         CALL DELAY           ;等待 1ms
                     .UNTILCXZ
              .ELSE
                     AND CX,7FFFH            ;使 CX 为正数
                     .REPEAT
                         ROR   AL,1          ; 向右旋转
                         OUT   PORT,AL
                         CALL DELAY          ; 等待 1ms
                     .UNTILCXZ
              .ENDIF
        .ENDIF
        MOV POS,AL
       RET

STEP   ENDP

//控制步进电机的 C++ 函数

char Step(char Pos, short Step)
{
       char Direction = 0;
       if (Step < 0)
       {
              Direction = 1;
              Step =& 0x8000;
       }
       while (Step)
       {
              if (Direction)
              {
                    if ((Pos & 1) == 1)
                    {
                          Pos = (Pos >> 1) | 0x80;
                    }
                    else
                    {
                          Pos >>= 1;
                    }
              }
              else
              {
```

```
                if ((Pos & 0x80) == 0x80)
                {
                        Pos = (Pos << 1) | 1;
                }
                else
                {
                        Pos <<= 1;
                }
        }
        _asm
        {
                mov al,Pos
                out 40h, al
        }
    }
    return Pos;
}
```

当前位置存储在存储单元 POS 中，它必须初始化为 33H、66H、0CCH 或 99H，这就允许用一个简单的 ROR（右步进）或 ROL（左步进）指令使二进制位模式旋转为下一步。

C++ 函数有两个参数：Pos 是指步进电机的当前位置，Step 是指如前所述的步数。新的 Pos 值被函数返回而不是存储在变量中。

步进电机也可以按半步模式运转，这样每个序列需要 8 步。这是通过使用半步描述的全步序列来实现的，半步是通过给全步之中的一个线圈通电获得的。半步允许电枢位于 0°、90°、180°和 270°，半步位置码为 11H、22H、44H 和 88H。一个完整的 8 步序列是 11H、33H、22H、66H、44H、0CCH、88H 和 99H。该序列既可从查找表输出，也可由软件产生。

键盘矩阵接口

键盘有各种大小，从与微处理器连接的标准 101 键的 QWERTY 键盘到只有 4～16 键的小型专用键盘。本节重点介绍预先组装好的或可由单个键盘开关构造的较小键盘。

图 11-25 给出了包含 16 个开关的一个小键盘矩阵，它与 82C55 的端口 A 和端口 B 相连接。在此例中，开关形成一个 4 × 4 矩阵，但可以使用其他矩阵，如 2 × 8。注意键是如何组织成 4 行（ROW_0～ROW_3）和 4 列（COL_0～COL_3）的。每行通过一个 10KΩ 上拉电阻与 5.0V 相连，以确保在没有按钮开关闭合时该行被拉到高电平。

对于 8088 微处理器，82C55 的 I/O 端口地址为 50H～53H（这里没有给出 PLD 程序）。端口 A 被编程为输入端口以读取行，端口 B 被编程为输出端口以选择一列。例如，若 1110 被输出给端口 B 的引脚 PB_3～PB_0，则第 0 列为逻辑 0，因此第 0 列中的 4 个键被选中。注意，PB_0 为逻辑 0 时，能置端口 A 为逻辑 0 的开关只有开关 0～3。若开关 4～F 闭合，则相应的端口 A 引脚保持为逻辑 1。同样，如果 1101 被输出给端口 B，则开关 4～7 被选中，依此类推。

从键盘矩阵读一个键并去除键抖动的软件流程图如图 11-26 所示。键必须去抖动，一般需要 10～20ms的短时间延迟。该流程图包括 3 个主要部分。第一部分等待键的释放，这似乎是多此一举，但在微处理器中软件执行非常快，有可能在一个键释放之前该程序就回到了程序的顶部，所以我们必须首先等待释放。下一部分等待一次键击。一旦键击被检测到，则在流程图的最后部分计算键的位置。

该软件使用了一个称为 SCAN 的过程扫描键盘和另一个称为 DELAY10（例子中未给出）的过程延迟 10ms 用于去抖动。主要的键盘过程称为 KEY，它与其他过程一起在例 11-17 中列出。例 11-17 同时还列出了完成键盘读操作的 C++ 函数。注意 KEY 过程是通用的，它可处理从 1 × 1 矩阵到 8 × 8 矩阵的任意配置的键盘。改变程序开始的两个等式（ROWS 和 COLS）将改变软件对任意大小键盘的配置。还应注意这里未给出初始化 82C55，使端口 A 为输入端口，端口 B 为输出端口所需的步骤。

由于某些键盘不符合键扫描的方式，这时就需要一个查找表来将原始的键盘码（由 KEY 返回的）转换为与键盘相一致的键盘码。查找程序放置在从 KEY 返回之前的位置，也就是紧跟 XLAT 的 MOV BX，OFFSET TABLE 这行代码。

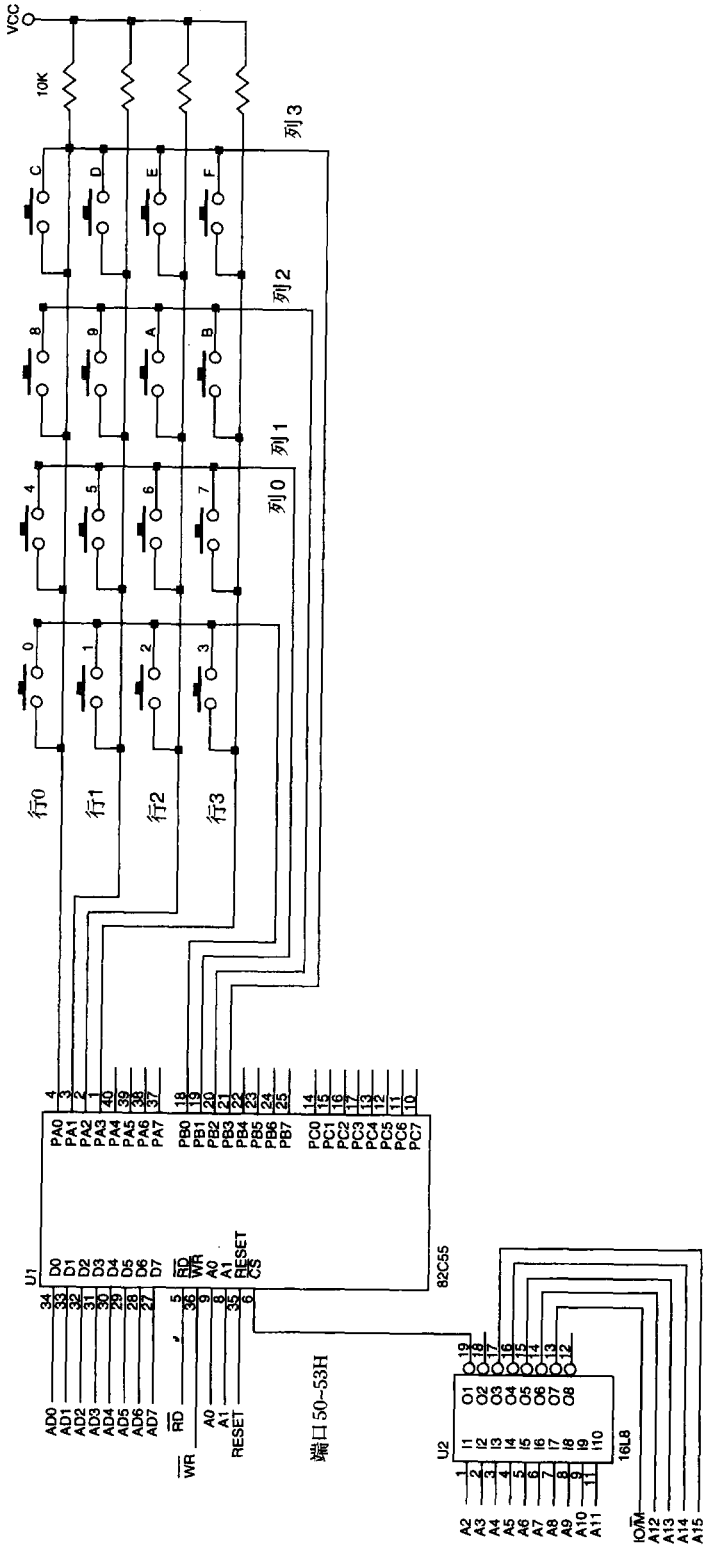


图 11-25 通过 82C55 的 PIA 与 8088 微处理器相连接的 4 × 4 键盘矩阵

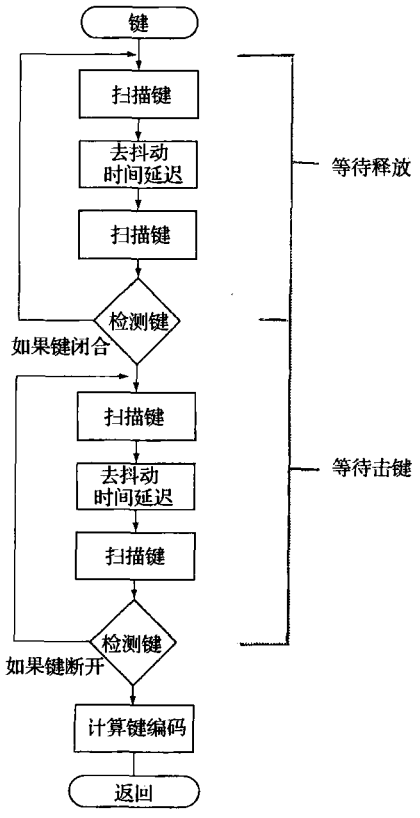


图 11-26 键盘扫描程序的流程图

例 11-17a

```
;汇编语言版本
;KEY 扫描键盘并且将键盘代码从 AL 中返回

        COLS    EQU   4
        ROWS    EQU   4
        PORTA   EQU   50H
        PORTB   EQU   51H

KEY     PROC    NEAR USES CX BX

        MOV   BL,FFH                ;计算行屏蔽值
        SHL   BL,ROWS

        MOV   AL,0
        OUT   PORTB,AL              ;将 0 置入端口 B 中

        .REPEAT                     ;等待键释放
              .REPEAT
                   CALL SCAN
              .UNTIL  ZERO?
              CALL DELAY10
              CALL SCAN
        .UNTIL ZERO?
        .REPEAT                     ;等待键值
              .REPEAT
                    CALL SCAN

              .UNTIL !ZERO?
              CALL DELAY10
```

```
        CALL SCAN
    .UNTIL !ZERO?
    MOV  CX,00FEH
    .WHILE 1                        ;找到所在的列
        MOV  AL,CL
        OUT  PORTB,AL
        CALL SHORTDELAY             ;见后面的说明
        CALL SCAN
        .BREAK !ZERO?
        ADD CH,COLS
        ROL  CL,1
    .ENDW
    .WHILE 1                        ; 找到所在的行
        SHR  AL,1
        .BREAK .IF !CARRY?
        INC  CH
    .ENDW
    MOV  AL,CH                      ;获取按键码
    RET

KEY ENDP

SCAN PROC  NEAR

    IN   AL,PORTA                   ;读取行
    OR   AL,BL
    CMP  AL,0FFH                    ; 测试是否按键
    RET

SCAN        ENDP
```

例 11-17b

```
// 键盘扫描软件的 C++ 语言版本

#define ROWS 4
#define COLS 4
#define PORTA 50h
#define PORTB 51h

char Key()
{
      char mask = 0xff << ROWS;
      _asm
      {
            mov al,0                        ;选择所有的列
            out PORTB,al
      }
      do
      {                                     //等待释放
            while (Scan(mask));
            Delay();
      }
      while (Scan(mask));
      do
      {                                     //等待按键
            while (!Scan(mask));
            Delay();
      }
      while (!Scan(mask));
      unsigned char select = 0xfe;
      char key = 0;
      _asm
      {
            mov  al,select
            out  PortB,al
      }
      ShortDelay();
      while(!Scan(mask))
```

```
        {                                          //计算键盘码
              _asm
              {
                    mov  al,select
                    rol  al,1
                    mov  select,al
                    out  PortB,al
              }
              ShortDelay();
              key += COLS;
        }
        _asm
        {
              in   al,PortA
              mov  select,al
        }
        while ((Select & 1) != 0)
        {
              Select <<= 1;
              key ++;
        }
        return key;
}

bool Scan(mask)
{
        bool flag;
        _asm
        {
              in   al,PORTA
              mov  flag,al
        }
        return (flag | mask);
}
```

SHORTDELAY 过程是必须的，因为计算机以一个非常高的速率改变端口 B。这个短暂的延时可以使得发送到端口 B 的数据达到最终稳定状态。在大多数情形下，如果扫描速度（输出指令之间的时间）不超过 30KHz，则不需要这个延时。如果扫描频率更高的话，设备将产生无线电干扰。否则，美国通信委员会（Federal Communications Commission，FCC）将不会批准其应用于任何他们认可的系统中。没有 FCC 的 A 类或 B 类证书，系统是不允许出售的。

11.3.5 方式 1 选通输入

方式 1 操作使端口 A 或端口 B 作为锁存输入设备工作。这就允许外部数据被存储在端口中，直到微处理器准备好去取它。端口 C 也可按方式 1 操作使用——但不传送数据，而是操作控制或握手信号，辅助端口 A 或端口 B 实现选通输入。图 11-27 给出了两个端口是如何构造用于方式 1 选通输入操作的，

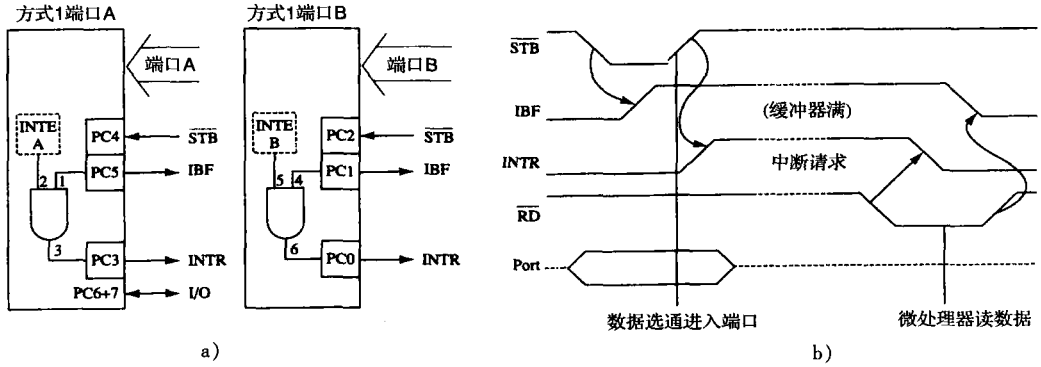


图 11-27 82C55 的选通输入操作（方式 1）

a）内部结构 b）时序图

并给出了它们的时序图。

选通输入端口在选通信号（$\overline{STB}$）被激活时从端口引脚上捕获数据。注意选通信号是在0到1跳变时捕获端口数据的。$\overline{STB}$信号使数据捕获到端口中，它还激活 **IBF**（**input buffer full，输入缓冲器满**）和 **INTR**（**interrupt request，中断请求**）信号。一旦微处理器通过软件（IBF）或硬件（INTR）注意到数据已被选通进入端口，它就执行一条 IN 指令读取该端口（$\overline{RD}$）。读端口的操作将 IBF 和 INTR 恢复到无效状态，直到下一个数据被选通进入端口。

方式1 选通输入的信号定义

$\overline{STB}$ **选通**输入将数据装入端口锁存器，该信息保持到由 IN 指令输入给微处理器。

IBF **输入缓冲器满**是一个输出信号，表明输入缓冲器已装入信息。

INTR **中断请求**是一个输出信号，它请求一次中断。INTR 引脚在$\overline{STB}$输入回到逻辑1时变为逻辑1；在微处理器从端口输入数据时被清零。

INTE **中断允许**信号既不是输入也不是输出，它是通过端口 PC_4（端口A）或 PC_2（端口B）编程设置的内部位。

PC_7 和 PC_6 端口C的引脚7和引脚6是通用 I/O 引脚，用途广泛。

选通输入实例

选通输入设备的一个很好的例子是键盘。键盘编码器去除键开关的抖动，并在一个键被压下时提供一个选通信号，这时数据输出包含 ASCII 编码的键代码。图 11-28 给出了一个与选通输入端口 A 相连的键盘。每次在键盘上按下一个键时，$\overline{DAV}$（data available，数据有效）被激活并维持 1.0 μs。由于$\overline{DAV}$与端口 A 的$\overline{STB}$输入相连，所以数据被选通进入端口 A。因此每次按下一个键，它就被选通进入 82C55 的端口 A 中。$\overline{STB}$输入还激活 IBF 信号，表明数据是在端口 A 中。

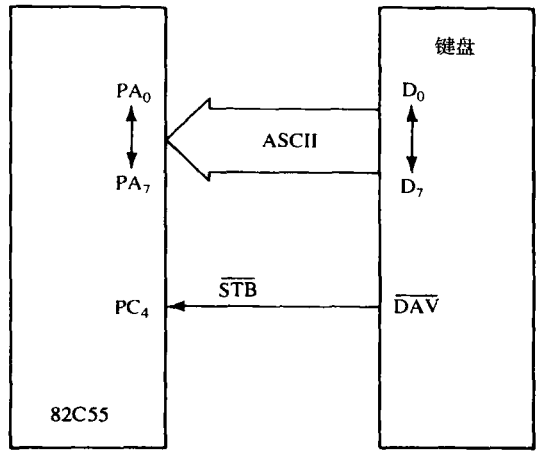


图 11-28 使用 82C55 用作键盘的选通输入操作

例 11-18 给出了每次按下一个键时从键盘读取数据的一个过程。该过程从端口 A 读取键值并返回存于 AL 中的 ASCII 码。为检测一个按键，要读取端口 C，测试 IBF 位（PC_5 位），以检查缓冲器是否满。如果缓冲器为空（IBF = 0），则该过程反复测试 IBF，等待从键盘上键入一个字符。

例 11-18

```
;读取键盘的编码，并从 AL 中返回其 ASCII 码的过程
;

BIT5    EQU   20H
PORTC   EQU   22H
PORTA   EQU   20H

READ    PROC  NEAR

        .REPEAT                          ;查询 IBF 位
              IN    AL,PORTC
              TEST  AL,BIT5
        .UNTIL !ZERO?
        IN    AL.PORTA                   ;取得 ASCII 数据
        RET

READ    ENDP
```

11.3.6 方式1选通输出

图11-29给出了当82C55作为一个选通输出设备在方式1下工作时，其内部结构和时序图。选通输出操作与方式0输出操作类似，只是它包括控制信号以提供握手操作。

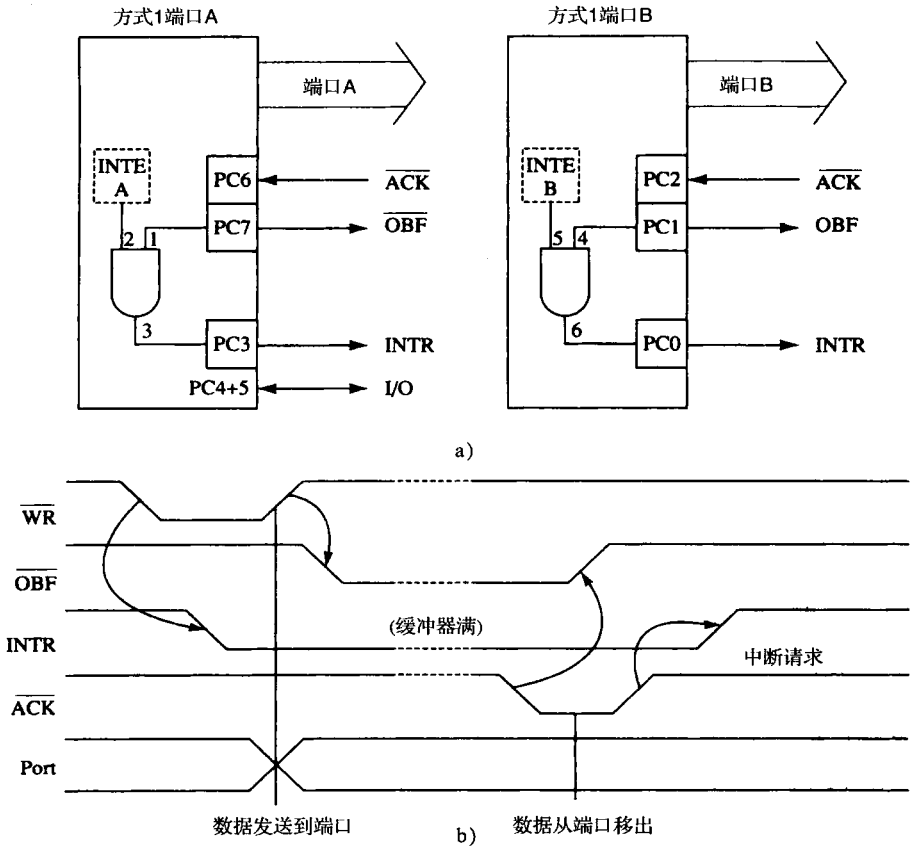


图11-29 82C55的方式1选通输出操作

a）内部结构 b）时序图

一旦数据被写入选通输出端口，$\overline{\text{OBF}}$（**output buffer full，输出缓冲器满**）信号就变为逻辑0，表明数据已出现在端口锁存器中。此信号表明对于一个外部I/O设备，数据是有效的。外部I/O设备是通过选通端口的$\overline{\text{ACK}}$（acknowledge，响应）输入信号来移走数据的。$\overline{\text{ACK}}$信号使$\overline{\text{OBF}}$信号回到逻辑1，表明缓冲器未满。

方式1选通输出的信号定义

$\overline{\text{OBF}}$ **输出缓冲器满**是一个输出信号，一旦数据输出（OUT）给端口A或端口B的锁存器，它就变为低电平；一旦外部设备返回$\overline{\text{ACK}}$脉冲，它就被设置为逻辑1。

$\overline{\text{ACK}}$ **响应信号**使$\overline{\text{OBF}}$引脚回到逻辑1电平。ACK是来自外部设备的一个响应信号，表明它已接收到来自82C55端口的数据。

INTR **中断请求**信号常常在外部设备通过$\overline{\text{ACK}}$信号接收数据时中断微处理器，该引脚受内部INTE位（interrupt enable，中断允许）的限制。

INTE **中断允许**既不是输入也不是输出，它是被编程为允许或禁止INTR引脚的一个内部位。

INTE A 位被编程为 PC_6，INTE B 位被编程为 PC_2。

PC_4 和 PC_5 端口 C 的引脚 5 和引脚 4 为通用 I/O 引脚，位设置与复位命令可用来设置或复位这两个引脚。

选通输出实例

11.1 节中讨论过的打印机接口在这里用来说明如何在打印机和 82C55 之间达到选通输出同步。图 11-30 描述端口 B 与一个并行打印机相连，该打印机有 8 个数据输入用于接收 ASCII 编码的数据，$\overline{\text{DS}}$（**data strobe，数据选通**）输入用于选通数据进入打印机，$\overline{\text{ACK}}$输出用于响应接收到的 ASCII 字符。

此电路中，没有信号产生$\overline{\text{DS}}$信号给打印机，所以软件用 PC_4 产生$\overline{\text{DS}}$信号。从打印机返回的$\overline{\text{ACK}}$信号响应数据的接收，并与 82C55 的$\overline{\text{ACK}}$输入相连。

例 11-19 列出了将 AH 中的 ASCII 编码字符发送给打印机的程序。该过程首先测试$\overline{\text{OBF}}$以确定打印机是否已从端口 B 移走数据。如果没有，该过程就等待从打印机返回的$\overline{\text{ACK}}$信号；若$\overline{\text{OBF}}$ = 1，则该过程通过端口 B 将 AH 中的内容发送给打印机，同时还发送$\overline{\text{DS}}$信号。

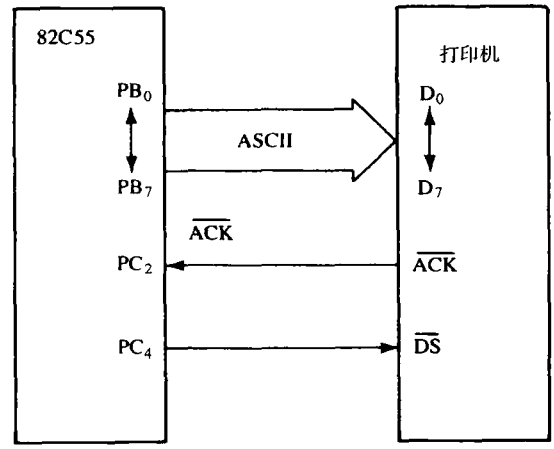


图 11-30 82C55 与并行打印机的接口，描述了 82C55 的选通输出方式操作

例 11-19

```
;这个过程将 ASCII 字符从 AH 送入连接到端口 B 的打印机中
;
BIT1    EQU   2
PORTC   EQU   63H
PORTB   EQU   61H
CMD     EQU   63H

PRINT   PROC  NEAR

        .REPEAT                         ;等待打印机准备好
              IN    AL,PORTC
              TEST  AL,BIT1
        .UNTIL !ZERO?
        MOV   AL,AH                     ;发送 ASCII 码
        OUT   PORTB,AL
        MOV   AL,8                      ;脉冲数据选通
        OUT   CMD,AL
        MOV   AL,9
        OUT   CMD,AL
        RET

PRINT   ENDP
```

11.3.7 方式 2 双向操作

方式 2 只允许 A 组采用，此时端口 A 变为双向，允许数据在同一组 8 条线上发送和接收。双向总线数据在连接两台计算机时很有用，它还用于 IEEE－488 并行高速 **GPIB**（**general purpose interface bus，通用接口总线**）接口标准。图 11-31 给出了方式 2 双向操作的内部结构和时序图。

双向方式 2 的信号定义

INTR **中断请求**是一个输出信号，用于在输入和输出情况下中断微处理器。

$\overline{\text{OBF}}$ **输出缓冲器满**是一个输出信号，表明输出缓冲器包含给双向总线的数据。

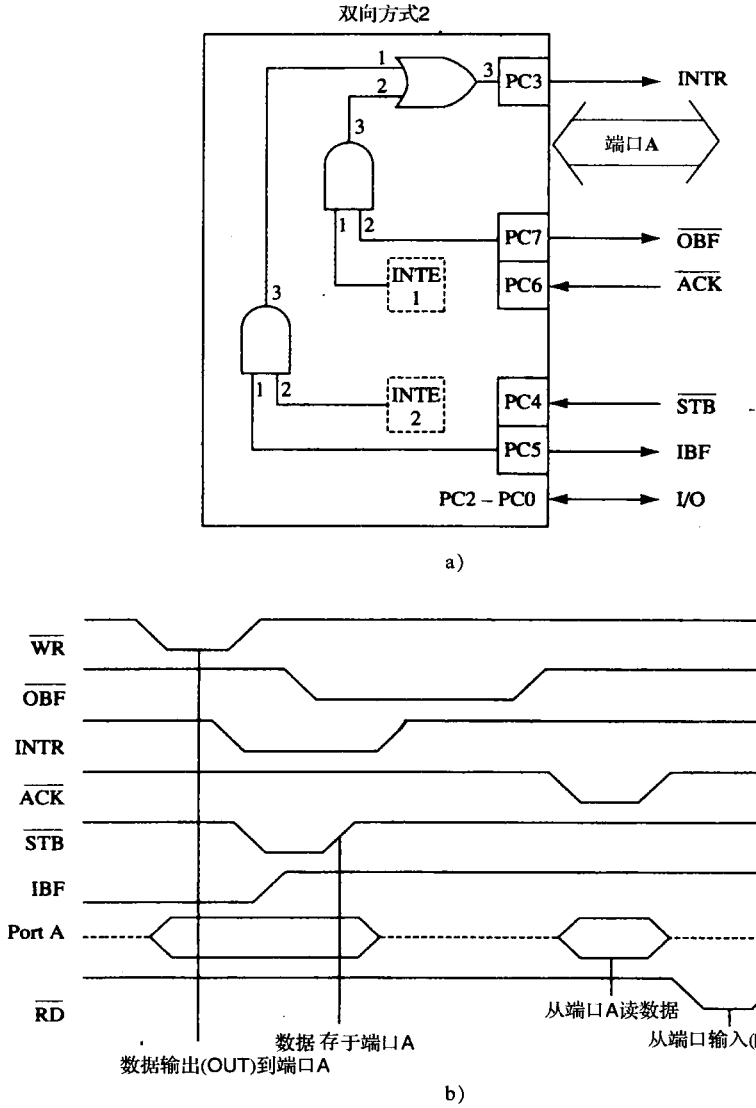


图 11-31 82C55 的方式 2 操作

a）内部结构 b）时序图

$\overline{ACK}$ **响应**输入信号允许三态缓冲器，使数据可以出现在端口 A。如果 ACK 为逻辑 1，则端口 A 的输出缓冲器处于高阻抗状态。

$\overline{STB}$ **选通**输入将来自双向端口 A 总线上的外部数据装入端口 A 的输入锁存器。

IBF **输入缓冲器满**是一个输出信号，表明输入缓冲器已包含外部双向总线的数据。

INTE **中断允许**是允许 INTR 引脚的内部位（INTE1 和 INTE2）。INTR 引脚的状态通过端口 C 的 PC_6 位（INTE1）和 PC_4（INTE2）控制。

PC_0、PC_1 和 PC_2 这些引脚在方式 2 下为通用 I/O 引脚，由位设置与复位命令控制。

双向总线

用 IN 和 OUT 指令访问端口 A 时使用双向总线。为通过双向总线发送数据，程序首先测试$\overline{OBF}$信号以确定输出缓冲器是否为空。若为空，则数据由 OUT 指令发送给输出缓冲器。外部电路也监视$\overline{OBF}$信号以决定微处理器是否已将数据发送给总线。一旦输出电路检测到$\overline{OBF}$为逻辑 0，它就从输出缓冲器

移走数据并返回$\overline{ACK}$信号。$\overline{ACK}$信号置位$\overline{OBF}$并使能三态输出缓冲器，从而可读出数据。例 11-20 列出了通过双向端口 A 发送 AH 寄存器中内容的一个过程。

例 11-20

```
;该过程通过双向总线传送 AH

BIT7   EQU  80H
PORTC  EQU  62H
PORTA  EQU  60H

TRANS  PROC NEAR

       .REPEAT                          ;测试 OBF
            IN   AL,PORTC
            TEST AL,BIT7
       .UNTIL !ZERO?
       MOV  AL,AH                       ;发送数据
       OUT  PORTA,AL
       RET

TRANS  ENDP
```

为通过双向端口 A 的总线接收数据，程序测试 IBF 位以确定数据是否已被选通进入端口。如果 IBF = 1，则用 IN 指令输入数据。外部接口通过使用$\overline{STB}$信号将数据送入端口，当$\overline{STB}$被激活时，IBF 信号变为逻辑 1，端口 A 的数据被保持在端口内部的锁存器中。当执行 IN 指令时，IBF 位被清零，端口中的数据被移入 AL。例 11-21 列出了从端口读取数据的过程。

例 11-21

```
;该过程从双向总线将数据读入 AL 中

BIT5  EQU  20H
PORTC EQU  62H
PORTA EQU  60H

READ  PROC NEAR

      .REPEAT                           ;测试 IBF
            IN   AL,PORTC
            TEST AL,BIT5
      .UNTIL !ZERO?
      IN   AL,PORTA
      RET

READ ENDP
```

INTR（中断请求）引脚可被通过总线的来自两个方向的数据流激活。如果两个 INTE 位都使能 INTR，则输出缓冲器空和输入缓冲器满都产生中断请求。这种情况出现在使用$\overline{STB}$将数据选通进入缓冲器的时候，或使用 OUT 指令写数据的时候。

11.3.8 82C55 方式小结

图 11-32 汇总了 82C55 的 3 种操作方式。方式 0 提供简单 I/O，方式 1 提供选通 I/O，方式 2 提供双向 I/O。正如前面提及过的，这些方式通过 82C55 的命令寄存器来选择。

		方式0		方式1		方式2
Port A		IN	OUT	IN	OUT	I/O
Port B		IN	OUT	IN	OUT	未用
Port C	0	IN	OUT	$INTR_B$	$INTR_B$	I/O
	1			IBF_B	$\overline{OBF_B}$	I/O
	2			$\overline{STB_B}$	$\overline{ACK_B}$	I/O
	3			$INTR_A$	$INTR_A$	INTR
	4			$\overline{STB_A}$	I/O	$\overline{STB}$
	5			IBF_A	I/O	IBF
	6			I/O	$\overline{ACK_A}$	$\overline{ACK}$
	7			I/O	$\overline{OBF_A}$	$\overline{OBF}$

图 11-32 82C55 PIA 端口连接汇总

11.3.9 串行EEPROM接口

在第10章图10-23中给出了一个串行的EEPROM，但是并没有将I/O设备用作接口。假设使用82C55的C口连接此接口，则还需要软件驱动此接口。假定引脚PC4连到SCL输入端，PC0连到SDA（串行数据连接）。PC4被编程为输出引脚以提供时钟信号。PC0被编程为输出发送数据，而被编程为输入则接收EEPROM的数据。

参照图10-24，软件读或写EEPROM数据的数据格式，在例11-22中给出。这些程序采用C语言和汇编语言编写，当然也可以完全采用汇编语言编写。命令寄存器的I/O端口地址是0x1203，C口寄存器的地址是0x1202。400KHz的数据速率对应的时延应为1.25μs。注意，此处并没有给出时延程序，这里使用while循环来等待写操作后的一个ACK信号。

例11-22

```
unsigned char void PC0in(unsigned char bit)
{
      _asm
      {
            mov dx,1203h
            mov al,81h
            out dx,al
            dec dx
            mov al,bit
            out dx,al
      }
      Delay();
      _asm
      {
            mov dx,1202h
            in al,dx              ;返回al
      }
}

void PC0out(unsigned char bit)
{
      _asm
      {
            mov dx,1203h
            mov al,80h
            out dx,al
            dec dx
            mov al,bit
            out dx,al
      }
      Delay();
}

unsigned char void SendByte(unsigned char data)
{
      for (int a = 7; a >= 0; a--)
      {
            PC0out((data >> a) & 0xef);
            PC0out((data >> a) | 0x10);
      }
      PC0in(0xef);                //ACK位
      return PC0in(0x10);
}

unsigned char GetByte()
{
      unsigned char temp = 0;
      for (int a = 7; a >= 0; a--)
      {
            PC0in(0xef);
            temp |= PC0in(0x10) << a;
      }
```

```
        PC0in(0xef);                    //ACK位
        PC0in(0x10);
        return temp;
}

void SendStart()
{       // start is one
        PC0out(0xef);
        PC0out(0x10);
}

void SendStop()
{       // stop is zero
        PC0out(0xee);
        PC0out(0x10);
}

void SendData(char device, short address, unsigned char data)
{
        Char c = 0;
        SendStart();
        SendByte(0xa0 | device << 1);
        SendByte(address >> 8);
        SendByte(address);
        SendByte(data);
        while (c == 0)
        {       // 等位ACK = 1;
                C = SendByte(0xa0 | device << 1);
        }
        SendStop();
}

unsigned char ReadData(char device, short address)
{
        SendStart();
        SendByte(0xa0 | device << 1);
        SendByte(address >> 8);
        SendByte(address);
        SendByte(0xa1 | device << 1);
        unsigned char temp = GetByte();
        SendStop();
        return temp;
}
```

11.4 8254 可编程间隔定时器

8254 可编程间隔定时器由 3 个独立的 16 位可编程计数器（**定时器**）组成。每个计数器可按二进制或二进制编码的十进制（BCD）计数。任一计数器的最高允许输入频率是 10MHz。此器件在微处理器必须控制实时事件的处理时非常有用，一些使用实例包括实时时钟、事件计数器以及电机速度和方向控制等。

该定时器还出现在 PC 机中，被译码为端口 40H～43H，完成以下工作：

1）产生一个基本定时器中断，其发生频率约为 18.2Hz。

2）刷新 DRAM 存储系统。

3）为内部扬声器和其他设备提供定时源。该定时器在 PC 机中是 8253 而不是 8254。

11.4.1 8254 功能描述

图 11-33 给出了 8254 的引脚图，它是 8253 的高速型号，还给出了 3 个计数器。每个定时器包含一个 CLK 输入、一个门控输入以及一个输出（OUT）引脚。CLK 输入为定时器提供了基本的操作频率，门控引脚控制工作在某些方式下的定时器，从 OUT 引脚可获得定时器的输出。

与微处理器相连的信号有数据总线引脚（D_7～D_0）、$\overline{RD}$、$\overline{WR}$、$\overline{CS}$以及地址输入 A_1 和 A_0。地址

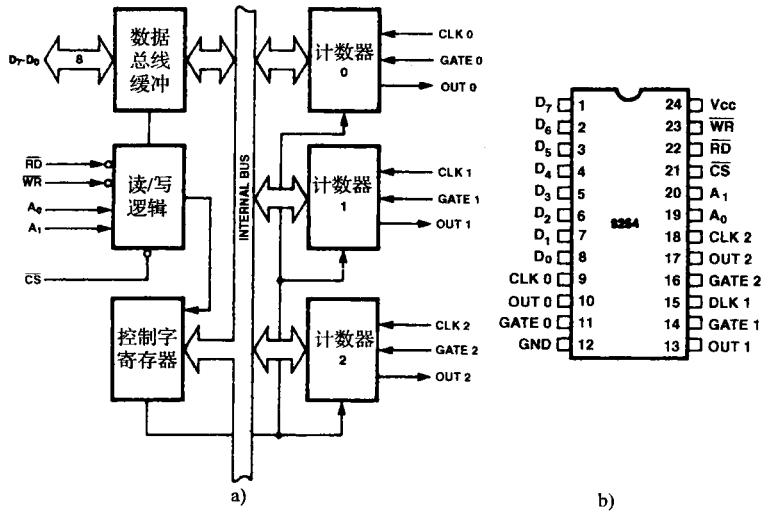


图 11-33 8254 可编程间隔定时器

a）内部结构 b）引脚输出

输入用于选择 4 个内部寄存器中的任何一个，这些寄存器用来对定时器进行编程：读或写。PC 机包含一个 8253 定时器或与其功能相同的器件，其 I/O 端口地址为 40H～43H。对定时器 0 编程，产生一个 18.2Hz 的时钟信号，以中断向量 8 去中断微处理器。该时钟信号常用于定时程序和事件。对定时器 1 编程，产生一个 15 μs 信号，用于 PC/XT 计算机申请一次 DMA 操作从而刷新动态 RAM。对定时器 2 编程，产生 PC 机扬声器上的双音频信号。

引脚定义

A_1 和 A_0 **地址输入**选择 8254 中 4 个内部寄存器中的一个，参见表 11-4 中 A_1 和 A_0 地址位的功能。

CLK **时钟**输入是每个内部定时器的定时源。该输入常与来自微处理器系统总线控制器的 PCLK 信号相连。

$\overline{CS}$ **片选**允许 8254 对计数器进行编程：读或写。

G **门控输入**控制计数器在某些操作方式下的操作。

GND **接地引脚**与系统接地总线相连。

OUT 在**计数器输出引脚**上可得到定时器产生的波形。

$\overline{RD}$ **读引脚**使数据从 8254 中读出，它常与$\overline{IORC}$信号相连。

V_{CC} **电源**与 +5.0V 电源相连。

$\overline{WR}$ **写引脚**使数据写入 8254，它常与写选通$\overline{IOWC}$信号相连。

表 11-4 8254 的地址选择输入

A_1	A_0	功 能
0	0	计数器 0
0	1	计数器 1
1	0	计数器 2
1	1	控制字

11.4.2 8254 编程

每个计数器通过写一个控制字和计数初值被单独编程。图 11-34 列出了 8254 的程序控制字结构。控制字允许编程人员选择计数器、操作方式以及操作类型（读/写）。控制字还可用于选择二进制计数还是 BCD 计数。每个计数器可用计数值 1 到 FFFFH 进行编程。计数值 0 等于 FFFFH + 1（65 536）或 BCD 码 10 000。最小计数值 1 应用于除了方式 2 和方式 3 以外的其他操作方式，方式 2 和方式 3 的最小计数值为 2。用于 PC 机中的定时器 0 被 64K（FFFFH）计数值分频后产生 18.2Hz（18.196Hz）的中断时钟信号。定时器 0 的时钟输入频率为 4.77MHz ÷ 4，即 1.1925MHz。

控制字使用 BCD 位来选择 BCD 计数（BCD = 1）或二进制计数（BCD = 0）。M_2、M_1 和 M_0 位选择计数器的 6 种不同操作方式（000～101）中的一种；RW_1 和 RW_0 位确定数据如何从计数器中读出或写入计数器；SC_1 和 SC_0 位选择一个计数器或特殊的读回操作方式，这将在本节后面讨论。

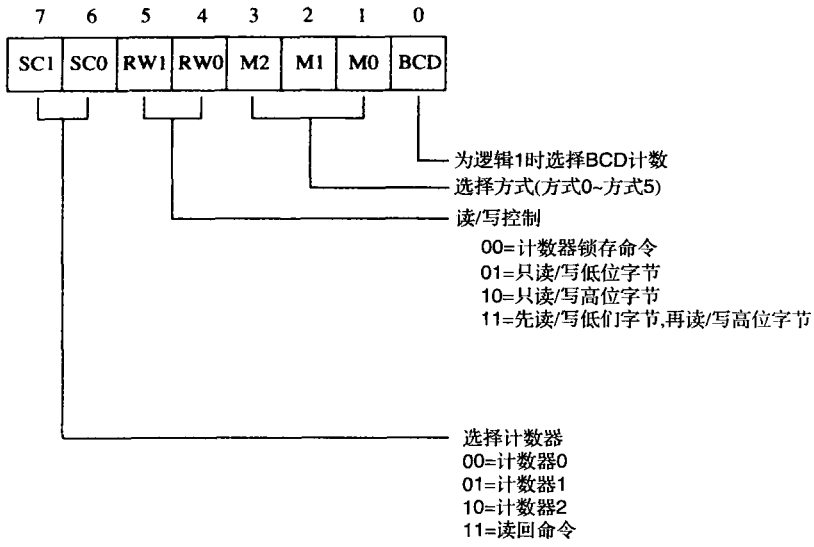


图 11-34 8254-2 定时器的控制字

每个计数器有一个程序控制字用于选择计数器操作方式。如果把 2 个字节编程到计数器里，那么第 1 个字节（LSB）将停止计数，第 2 个字节将以新的计数值启动计数。对每个计数器来说，编程顺序非常重要，但为了更好地进行控制，不同计数器的编程可以交叉存取。例如，对于单独的编程，控制字可以在计数之前发送给每个计数器。例 11-23 给出了编程计数器 1 和 2 的几种方法。第 1 种方法编程两个控制字，然后为每个计数器编程其计数值的 LSB，它使计数器停止计数，最后编程计数值的 MSB 部分，以新计数值启动这两个计数器；第 2 种方法是先编程一个计数器，再编程另一个计数器。

例 11-23

```
PROGRAM CONTROL WORD 1 PROGRAM CONTROL WORD 2 PROGRAM LSB 1
PROGRAM LSB 2
PROGRAM MSB 1
PROGRAM MSB 2

;设置计数器1
;设置计数器2

;停止计数器1,编程LSB
;停止计数器2,编程LSB;编程计数器1的MSB并启动它
;编程计数器2的MSB并启动它

      或

PROGRAM CONTROL WORD 1 PROGRAM LSB 1
PROGRAM MSB 1
PROGRAM CONTROL WORD 2 PROGRAM LSB 2
PROGRAM MSB 2

;设置计数器1
;停止计数器1,编程LSB;编程计数器1的MSB并启动它
;设置计数器2
;停止计数器2,编程LSB;编程计数器2的MSB并启动它
```

操作方式

每个 8254 计数器有 6 种操作方式（方式 0 到方式 5）。图 11-35 显示了每种方式是如何与 CLK 输入、门控（G）信号以及 OUT 信号一起工作的。各种方式描述如下：

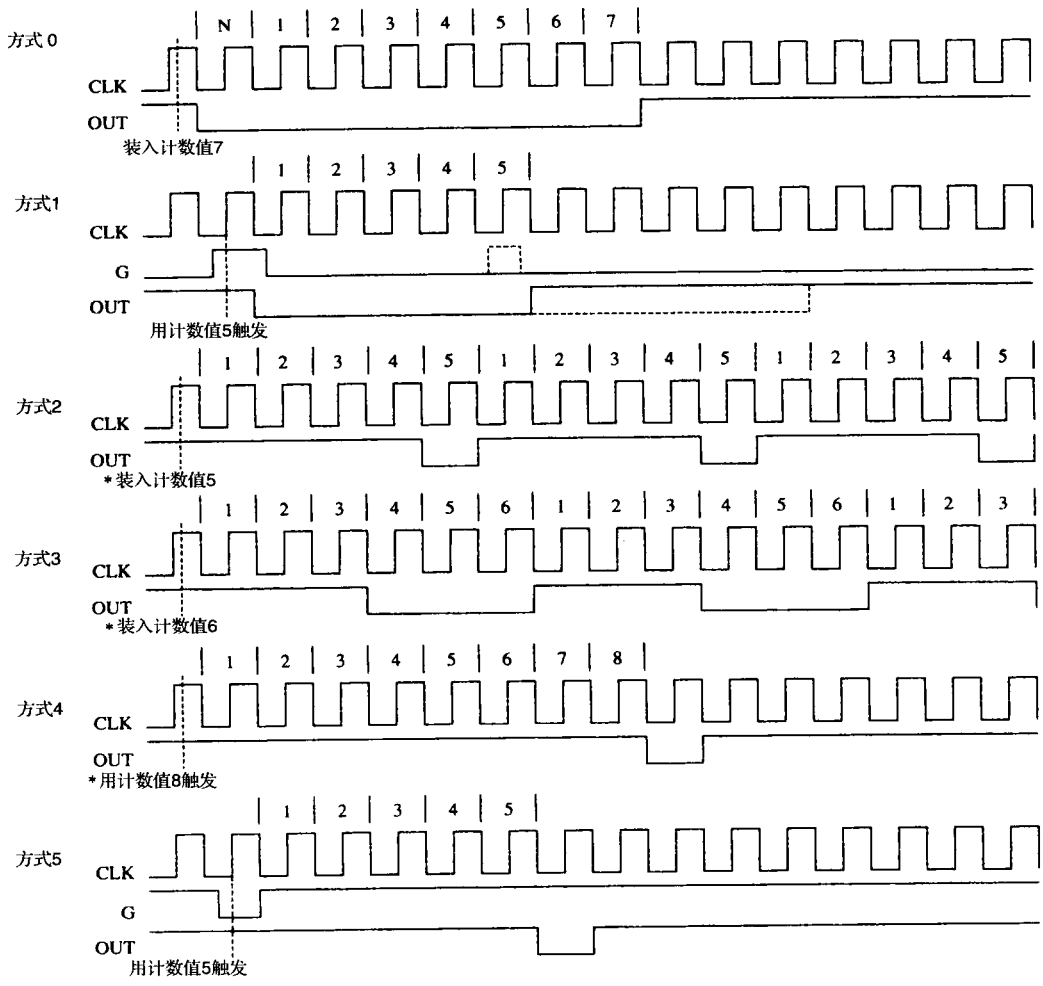


图 11-35　8254-2 可编程间隔定时器的 6 种操作方式 * 在方式 2、3 和 4 下 G 输入为 0 时停止计数

方式 0　允许 8254 计数器用作事件计数器。在此方式中，当写入控制字时，输出变为逻辑 0，并一直维持到 N 加上编程的计数值。例如，若计数值 5 被编程，则输出将一直维持逻辑 0，直到开始于 N 的计数值 6。注意门控（G）输入必须为逻辑 1 以允许计数器计数。如果 G 在计数中变为逻辑 0，则计数器将停止计数，直到 G 又变为逻辑 1。

方式 1　使计数器作为可重触发的单稳多谐振荡器（单脉冲）。在此方式中，G 输入触发计数器，使其在 OUT 引脚产生一个脉冲，此脉冲在计数期间为逻辑 0。若计数值为 10，则 OUT 引脚在被触发时变为逻辑 0 并维持 10 个时钟周期。如果 G 输入出现在输出脉冲期间，则计数器再次装入计数值，OUT 引脚维持低电平为计数值总长度的时间。

方式 2　允许计数器产生一系列连续的脉冲，脉冲宽度为一个时钟脉冲的宽度。脉冲之间的间隔取决于计数值。例如，若计数值为 10，则输出为逻辑 1 的时间是 9 个时钟周期，而为逻辑 0 的时间是 1 个时钟周期。重复此循环直到计数器被编程为一个新的计数值，或 G 引脚被置为逻辑 0 电平。对于该方式，为产生连续的一系列脉冲，G 输入必须为逻辑 1。

方式 3　G 引脚为逻辑 1 时在 OUT 引脚产生一个连续的方波。如果计数值为偶数，则输出为高电平的时间是计数值的一半，为低电平的时间也是计数值的一半。如果计数值为奇数，则输出为高电平的时间比输出为低电平的时间长一个时钟周期。例如，如果计数器被编程为计数值 5，则

输出为高电平的时间是3个时钟，而输出为低电平的时间是2个时钟。

方式4 允许计时器在输出端产生单个脉冲。如果计时器被编程为10，则输出为高电平的时间是10个时钟周期，而为低电平的时间是1个时钟周期。这一循环只有在计数器装入完整的计数值后才会开始。此方式以软件触发单脉冲方式操作。与方式2和方式3一样，此方式也使用G输入使能计数器。计数器以这3种方式操作时，G输入都必须为逻辑1。

方式5 硬件触发单脉冲方式，与方式4相似，只是它由G引脚上的一个触发脉冲启动而不是靠软件启动。此方式也类似于方式1，因为它是可重触发的。

用8254产生波形

图11-36给出了8254与80386SX微处理器的连接图，其I/O端口地址为0700H、0702H、0704H和0706H。使用PLD译码地址，并为8254产生一个写选通信号，8254与低8位数据总线相连。PLD还为微处理器产生等待信号，在8254被访问时产生2个等待状态。与微处理器相连的等待状态发生器实际上用来控制插入时序中的等待状态的个数。这里未给出PLD的程序，因为它与前面的许多例子相同。

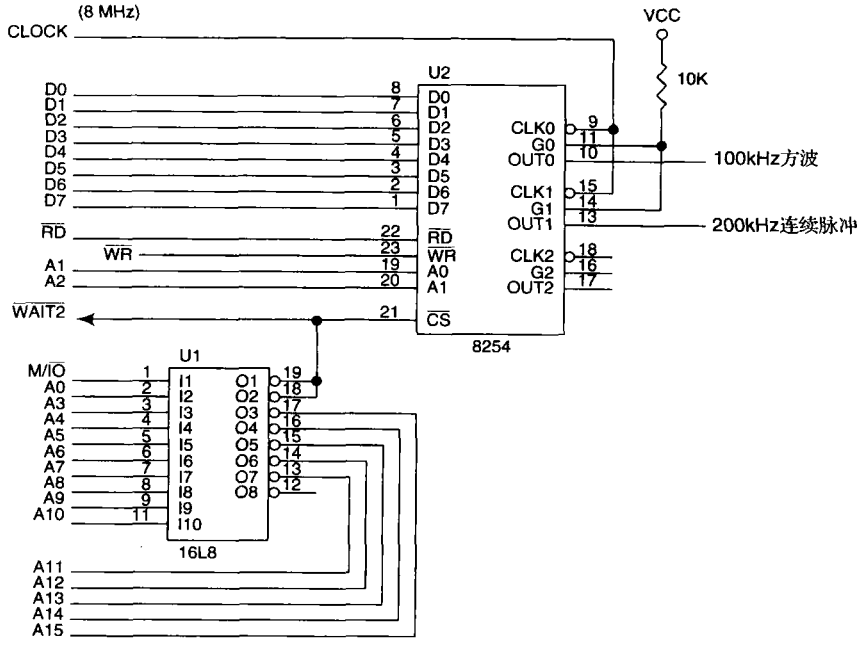


图11-36 8254与8MHz 8086相连，在OUT0引脚产生100KHz方波，在OUT1引脚产生200KHz连续脉冲

例11-24列出了一个程序，它在OUT0引脚产生100KHz的方波，在OUT1引脚产生200KHz的连续脉冲。计数器0使用方式3，计数器1使用方式2。编程到计数器0中的计数值为80，计数器1的计数值为40。这2个计数值对于8MHz输入时钟产生期望的输出频率。

例11-24

```
;该过程对8254定时器进行编程，其功能如图11-34所示

TIME    PROC    NEAR USES AX DX

        MOV   DX,706H              ;编程计数器0为方式3
        MOV   AL,00110110B
        OUT   DX,AL
        MOV   AL,01110100B         ;编程计数器1为方式2
        OUT   DX,AL
```

```
        MOV   DX,700H              ;编程计数器0置为80
        MOV   AL,80
        OUT   DX,AL
        MOV   AL,0
        OUT   DX,AL

        MOV   DX,702H              ;编程计数器1置为40
        MOV   AL,40
        OUT   DX,AL
        MOV   AL,0
        OUT   DX,AL

        RET

TIME    ENDP
```

读计数器

每个计数器有一个内部锁存器，它由读计数器端口操作进行读取。这些锁存器通常跟着计数值变。如果需要计数器的内容，那么锁存器可以通过编程计数器锁存控制字（参见图 11-37）来记住计数值，这个控制字使得计数器的内容被保持在锁存器中直到它被读出。一旦编程读锁存器或计数器时，锁存器就跟踪计数器的内容。

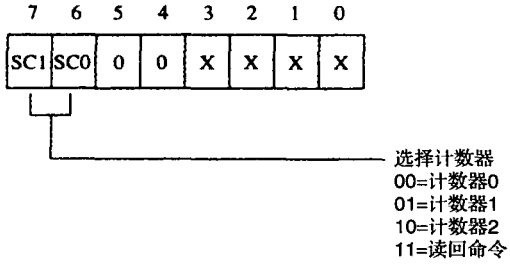


图 11-37 8254-2 计数器锁存控制字

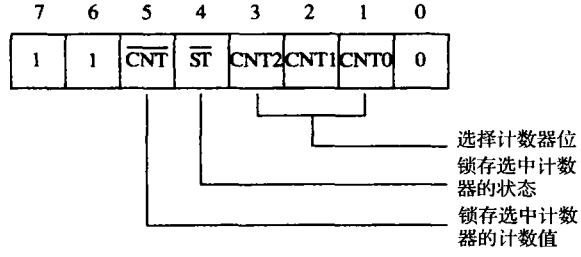


图 11-38 8254-2 计数器读回控制字

在有必要同时读出多于一个计数器的内容时，使用图 11-38 中的读回控制字。对于读回控制字，$\overline{\text{CNT}}$位为逻辑 0 时锁存由 CNT0、CNT1 及 CNT2 选中的计数器。如果要锁存状态寄存器，则将$\overline{\text{ST}}$位置为逻辑 0。图 11-39 给出了状态寄存器，它表明输出引脚的状态：计数器是否处于无效状态（0）以及如何编程计数器。

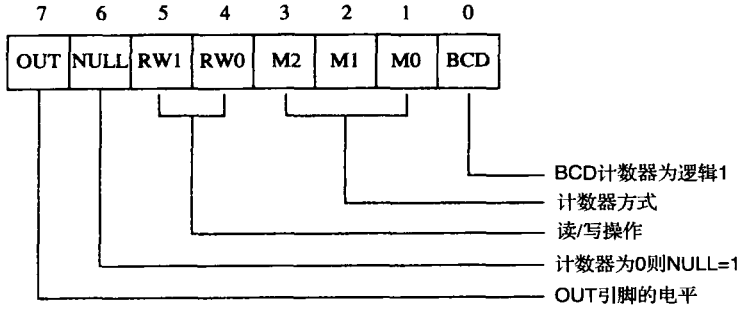


图 11-39 8254-2 状态寄存器

11.4.3 直流电机速度与方向控制

8254 定时器的一个应用是作为直流电机的电机速度控制器。图 11-40 给出了电机的示意图及其相关的驱动器电路。它还给出了 8254、触发器以及电机和驱动器之间的互连。

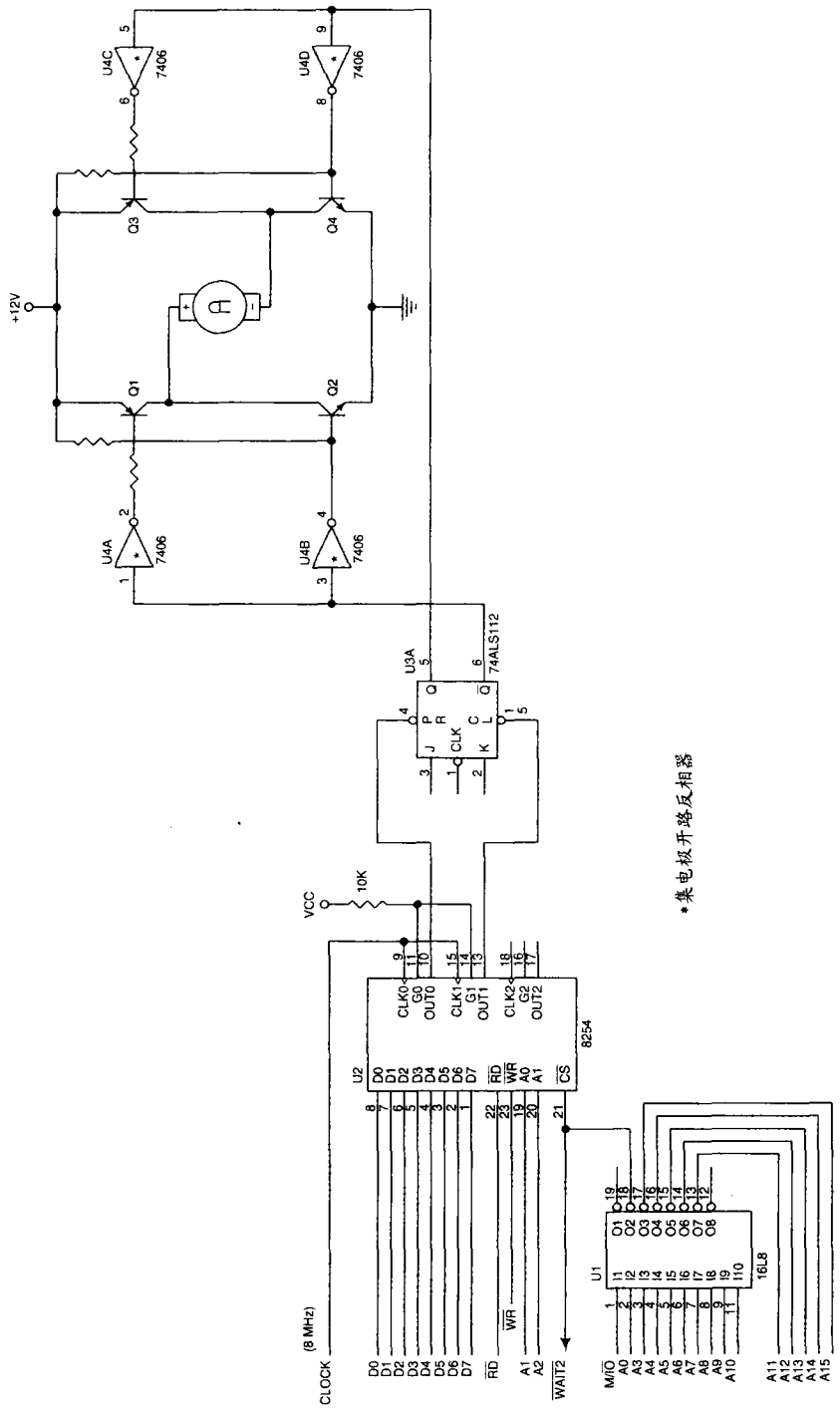


图11-40 使用8254定时器控制电机速度和方向

电机驱动器电路的工作比较简单。如果74ALS112的Q输出为逻辑1，则Q_2基极通过基极上拉电阻被拉到+12V，Q_1基极为断路。这意味着Q_1断开时Q_2导通，电机的正端接地。Q_3和Q_4的基极通过反相器被拉为低电平，使得Q_3导通而Q_4断开，从而使电机的负端接+12V。因此触发器的Q输出为逻辑1时使电机的负端接+12V而正端接地。这种连接使电机向前旋转。如果触发器Q输出的状态变为逻辑0，那么晶体管通断的条件正好相反，电机的正端接+12V而负端接地，这使得电机按相反方向旋转。

如果触发器的输出在逻辑1和0之间交替变化，则电机在每个方向上以不同的速度旋转。如果Q输出的占空比为50%，则电机根本不会旋转，且展示出某个保持扭矩，这是因为有电流流过电机。图11-41给出了几个时序图及其对电机速度和方向的影响。注意每个计数器是如何在不同位置产生脉冲从而改变触发器Q输出占空比的，这个输出也被称为**脉冲宽度调制**（**pulse width modulation**）。

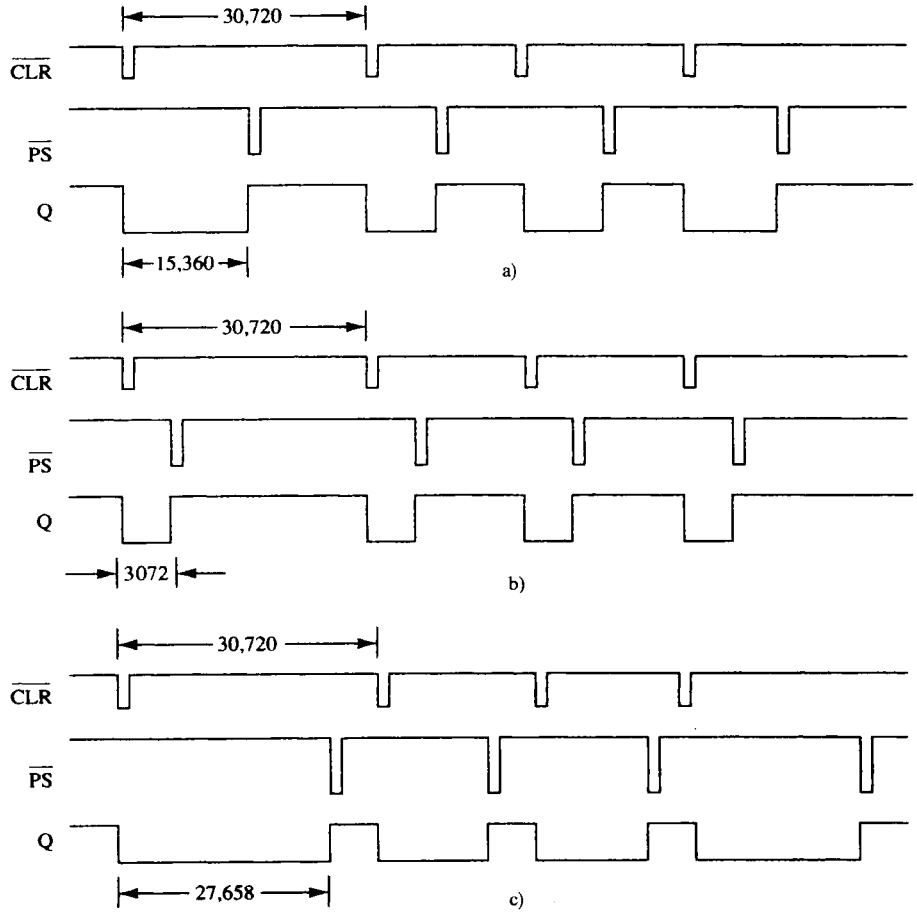


图11-41 图11-40中电机速度和方向控制电路的时序图

a）不旋转 b）反向高速旋转 c）正向高速旋转

为产生这些波形，计数器0和1均被编程为对输入时钟（PCLK）进行30720分频。通过改变计数器1相对计数器0开始计数的时间来改变Q的占空比，这样就改变了电机的方向和速度。但为何用30720来分频8MHz时钟呢？因为分频比30720可被256除尽，所以设计一个短程序就可允许256种不同的速度，也为电机产生了一个大约260Hz的基本工作频率，这么低的频率足以驱动电机。工作频率保持在60Hz～1000Hz之间是非常重要的。

例11-25列出了控制电机速度和方向的一个过程。调用该过程时，速度由AH中的值控制。由于用

8位数表示速度，那么停止电机的50%占空比所需计数值为128。调用该过程时，通过改变AH中的值可以调整电机速度，改变AH中的数可以使电机在每个方向上增速。当AH中的值接近00H时，电机开始反向增加速度；当AH中的值接近FFH时，电机正向增加速度。

例11-25

```
;控制图11-40中电机的转速与方向的过程
;
;AH决定电机的速度与方向,其取值范围为00H~FFH

CNTR    EQU   706H
CNT0    EQU   700H
CNT1    EQU   702H
COUNT   EQU   30720

SPEED   PROC  NEAR USES BX DX AX

        MOV   BL,AH                    ;计算次数
        MOV   AX,120
        MUL   BL
        MOV   BX,AX
        MOV   AX,COUNT
        SUB   AX,BX
        MOV   BX,AX

        MOV   DX,CNTR
        MOV   AL,00110100B             ;编程控制字
        OUT   DX,AL
        MOV   AL,01110100B
        OUT   DX,AL

        MOV   DX,CNT1                  ;编程计数器1
        MOV   AX,COUNT                 ;产生一次清0
        OUT   DX,AL
        MOV   AL,AH
        OUT   DX,AL

        .REPEAT                        ;等待计数器1
              IN    AL,DX
              XCHG  AL,AH
              IN    AL,DX
              XCHG  AL,AH
        .UNTIL BX == AX

       MOV   DX,CNT0                   ;编程计数器0
       MOV   AX,COUNT                  ;产生一次置位
       OUT   DX,AL
       MOV   AL,AH
       OUT   DX,AL

       RET

SPEED  ENDP
```

该过程首先计算计数器0相对计数器1启动的计数值来调整Q的波形，这是通过30720减去AH与120的乘积来实现的。需要这么做是因为计数器为减法计数器，它在重新启动前从编程的计数值开始计数，一直到0。接着，计数器被编程为计数值30720，并开始为触发器产生清除波形。计数器1启动后，它被读取并与计算的计数值相比较，一旦达到此值，则计数器0以30720的计数值开始启动计数。从此时开始，两个计数器连续产生清除和设置波形，直到该过程再次被调用，以调整电机的速度和方向。

11.5 16550可编程通信接口

美国国家半导体公司的PC16550D是一种可以连接几乎任何类型串行接口的可编程通信接口。它

是一个通用异步接收器/发送器（UART），与 Intel 微处理器完全兼容。它能以 0～1.5M 的波特率工作。波特率是每秒传送的位数，包括起始位、停止位、数据和奇偶校验位。16650 还包括一个可编程波特率发生器以及用于输入输出数据以减轻微处理器负担的独立 FIFO。每个 FIFO 包含 16 字节的存储量。16550 是目前基于微处理器的设备（包括 PC 机和许多调制解调器）中最普遍的通信接口。

11.5.1 异步串行数据

异步串行数据无需时钟或定时信号就可发送和接收。图 11-42 给出了两帧异步串行数据，每帧包含一个起始位、7 个数据位、一个奇偶校验位以及一个停止位。该图说明一帧包含一个 ASCII 字符，共 10 位。大多数拨号通信系统，如 CompuServe、Prodigy 及 America Online，都使用具有偶校验的 10 位异步串行数据。大多数 Internet 和电子公告牌服务也使用 10 位，但它们一般不使用奇偶校验位，而是传送 8 位数据，用一个数据位取代了奇偶校验位，这样使得非 ASCII 数据的字节传送更易实现。

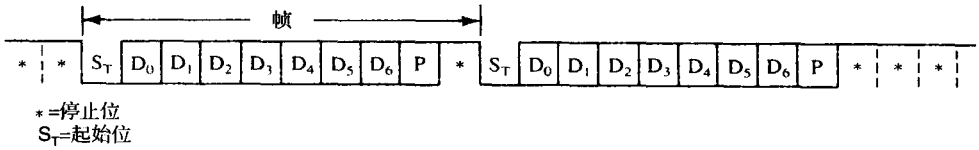


图 11-42 异步串行数据

11.5.2 16550 功能描述

图 11-43 给出了 16550 UART 的引脚图。该器件可以是 40 引脚 **DIP**（**dual in-line package，双列直插封装**）或 40 引脚 **PLCC**（**plastic lead-less chip carrier，塑料无引线芯片载体**）。两个完全独立的部分，即接收器和发送器负责数据通信。由于每个部分是互相独立的，所以 16550 能够以单工、半双工或全双工方式工作。16550 的主要特性之一是它的内部接收器和发送器 FIFO 存储器。由于每个存储器为 16 字节深，所以 UART 在接收到 16 字节数据后才要求微处理器来处理，它还在微处理器必须等待发送器之前保持 16 字节数据。FIFO 使得这种 UART 成为与高速系统接口的理想器件，因为只需较少的时间对它服务。

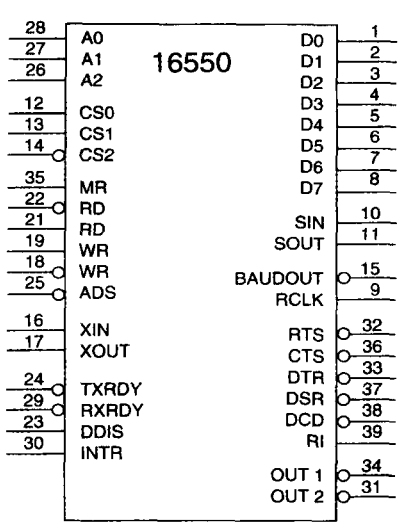


图 11-43 16550 UART 的引脚输出

单工（**simplex**）系统的一个例子是 **FM**（**frequency modulation，调频**）无线电台，它只能独自用发送器或者接收器。**半双工**（**half-duplex**）系统的一个例子是 **CB**（**citizens band，民用波段**）无线通信，它可以发送和接收，但二者不能同时进行。**全双工**（**full-duplex**）系统允许在两个方向上同时进行发送和接收，这样的一个例子是电话。

16550 可控制**调制解调器**（**modulator/demodulator，modem**），调制解调器将串行数据的 TTL 电平转换成可通过电话系统的音频信号。16550 上有 6 个引脚用于调制解调器控制：$\overline{\text{DSR}}$（**data set ready，数据装置就绪**），$\overline{\text{DTR}}$（**data terminal ready，数据终端就绪**），$\overline{\text{CTS}}$（**clear-to-send，允许发送**），$\overline{\text{RTS}}$（**request-to-send，请求发送**），$\overline{\text{RI}}$（**ring indicator，响铃指示**）以及$\overline{\text{DCD}}$（**data carrier detect，数据载波检测**）。调制解调器用作数据装置，16550 用作数据终端。

16550 引脚功能

A_0、A_1 和 A_2 **地址输入**用于选择内部寄存器进行编程和数据传送。参见表 11-5 列出的地址输入的每种组合及选中的寄存器。

表 11-5 由 A_0、A_1 和 A_2 选择的寄存器

A_2	A_1	A_0	功 能
0	0	0	接收器缓冲（读）与发送器保持（写）
0	0	1	中断允许
0	1	0	中断识别（读）与 FIFO 控制（写）
0	1	1	线路控制
1	0	0	调制解调器控制
1	0	1	线路状态
1	1	0	调制解调器状态
1	1	1	暂存

$\overline{ADS}$ **地址选通**输入用于锁存地址线和片选线。如果不需要（如在 Intel 系统中），则将此引脚接地。$\overline{ADS}$引脚被设计用于 Motorola 微处理器。

$\overline{BAUDOUT}$ 在**波特输出**引脚上可得到由发送器部分的波特率发生器产生的时钟信号。它常与 RCLK 输入相连，以产生与发送器时钟相等的接收器时钟。

CS_0、CS_1 和$\overline{CS_2}$ **片选**输入必须都有效才能使能 16550 UART。

$\overline{CTS}$ **允许发送**（如果为低）表明调制解调器或数据装置准备交换信息。该引脚常用于半双工系统中改变传输方向。

$D_7 \sim D_0$ **数据总线**引脚与微处理器数据总线相连。

$\overline{DCD}$ **数据载波检测**输入用于调制解调器发信号给 16550，通知它有载波出现。

DDIS **禁止驱动器**输出变为逻辑 0 时表明微处理器正从 UART 读取数据。DDIS 可用于改变数据流通过缓冲器的方向。

$\overline{DSR}$ **数据装置就绪**是给 16550 的输入信号，表明调制解调器或数据装置准备操作。

$\overline{DTR}$ **数据终端就绪**是一个输出信号，表明数据终端（16550）准备工作。

INTR **中断请求**信号是一个给微处理器的输出信号，一旦 16550 有一个接收器错误、已接收到数据以及发送器为空时，就申请一次中断（INTR = 1）。

MR **主复位**信号初始化 16550，应与系统 RESET 信号相连。

$\overline{OUT1}$和$\overline{OUT2}$ 用户定义的输出引脚，在系统需要时可为调制解调器或任何其他设备提供信号。

RCLK **接收器时钟**是 UART 的接收器部分的时钟输入，此输入总等于 16×理想的接收器波特率。

RD 和$\overline{RD}$ **读**输入（两个都可以用）使数据从由 UART 地址输入指定的寄存器中读出。

$\overline{RI}$ **响铃指示**输入由调制解调器置为逻辑 0，表明电话在响。

$\overline{RTS}$ **请求发送**是给调制解调器的信号，表明 UART 希望发送数据。

SIN 和 SOUT 这些为串行数据引脚。SIN 接收串行数据而 SOUT 发送串行数据。

$\overline{RXRDY}$ **接收器就绪**信号用于传送通过 DMA 技术（参见本书）接收到的数据。

$\overline{TXRDY}$ **发送器就绪**信号用于传送通过 DMA 技术（参见本书）发送的数据。

WR 和$\overline{WR}$ 写信号（两个都可以用）与微处理器的写信号相连，传送命令和数据给 16550。

XIN 和 XOUT 这些是**主时钟引脚**，一个晶体通过这些引脚连接到 16550，形成晶体振荡器，或者 XIN 与外部定时源相连。

11.5.3 16550 编程

16550 的编程比较简单，尽管可能比本章描述的其他一些可编程接口稍复杂一些。编程分为两部分过程，包括初始化对话与操作对话。

*在 PC 机中使用 16550 或与其功能相同的可编程接口时，其 I/O 端口地址被译码为：COM0 端口为 3F8H～3FFH，COM2 端口为 2F8H～2FFH。*尽管本节中给出的例子不是专门针对 PC 机而写的，但可以利用这些程序，通过改变端口号来控制 PC 机上的 COM 端口。

初始化 16550

初始化对话出现在硬件或软件复位后，它由两部分组成：编程线路控制寄存器与波特率发生器。线路控制寄存器选择数据位数、停止位个数以及奇偶校验位（是偶校验还是奇校验，或者校验位是作为 1 还是 0 发送）。波特率发生器由一个确定发送器波特率的除数进行编程。

图 11- 44 给出了线路控制寄存器，它通过输出信息给 I/O 端口 011（A_2、A_1 和 A_0）进行编程。其最右边 2 位选择发送的数据位数（5、6、7 或 8）。停止位位数由线路控制寄存器中的 S 来选择：如果S = 0，则使用 1 位停止位；如果 S = 1，对于 5 个数据位，使用 1. 5 位停止位，对于 6、7 或 8 个数据位，使用 2 位停止位。

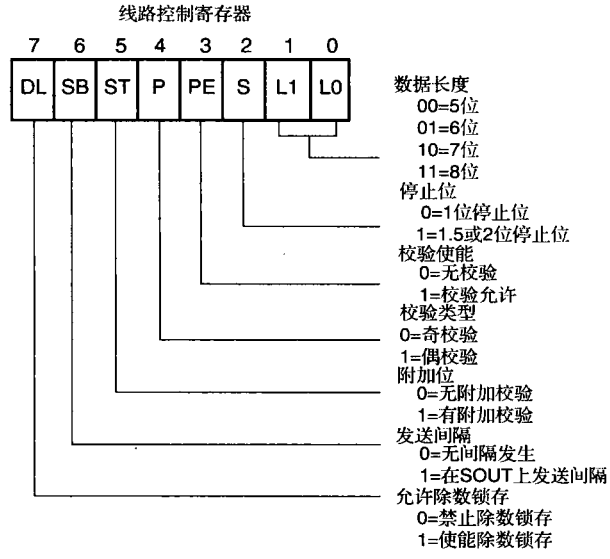


图 11-44　16550 线路控制寄存器的内容

接下来的 3 位一起用于发送偶校验或奇校验、发送无奇偶校验或在奇偶校验位位置发送 1 或 0。为发送偶校验或奇校验，**ST**（**stick，附加**）位必须置为逻辑 0，且奇偶校验允许必须置为逻辑 1，奇偶校验位的值确定偶校验或奇校验。为发送无奇偶校验（在 Internet 连接中很普遍），ST = 0，奇偶校验允许也为 0，在这种情况下发送和接收数据都没有奇偶校验位。最后，如果对所有数据必须发送和接收一个 1 或 0 到校验位位置，则 ST = 1且奇偶校验允许也为 1。为发送一个 1 到校验位位置，则将奇偶校验位置为 0；为发送一个 0，则将奇偶校验位置为 1（参见表 11- 6 奇偶校验位及附加位的操作）。

表 11-6　ST 及奇偶校验位的操作

ST	P	PE	功　能
0	0	0	无奇偶校验位
0	0	1	奇校验
0	1	0	无奇偶校验位
0	1	1	偶校验
1	0	0	未定义
1	0	1	发送/接收 1
1	1	0	未定义
1	1	1	发送/接收 0

表 11-7　18. 432MHz 晶体的波特率发生器使用的除数与常见波特率

波特率	除数值
110	10 473
300	3840
1200	920
2400	480
4800	240
9600	120
19 200	60
38 400	30
57 600	20
115 200	10

线路控制寄存器的其余 2 位用于发送一个间隔符，以及选择对波特率除数进行编程。如果线路控制寄存器第 6 位为逻辑 1，则发送一个间隔符。只要此位为 1，则 SOUT 引脚就发送间隔符。一次间隔定义为至少 2 帧逻辑 0 数据。系统软件负责间隔符发送的定时。为结束间隔，线路控制寄存器的第 6 位回到逻辑 0 电平。波特率除数只在第 7 位为逻辑 1 时是可编程的。

编程波特率

波特率发生器被编程在 I/O 地址 000 和 001（A_2、A_1、A_0）。端口 000 用于保持 16 位除数的最低有效部分，端口 001 用于保持最高有效部分。除数的值取决于外部时钟或晶体频率。表 11-7 给出了 18. 432MHz 晶体用作定时源时可得到的常见波特率，还给出了编程到波特率发生器中获得这些波特率

的除数值。被编程到波特率发生器中的实际数值使波特率发生器产生一个时钟，为16与理想波特率的乘积。例如，如果把240编程为波特率除数，则波特率为18.432MHz／（16×240）＝4800波特。

初始化举例

假定一个异步系统需要7位数据位、奇校验、9600的波特率及1位停止位。例11-26列出了初始化16550使其以此种方式工作的过程。图11-45给出了与8088微处理器的接口，使用一个PLD译码8位端口地址为F0H～F7H（未给出PLD程序）。这里，端口F3H存取线路控制寄存器，F0H和F1H存取波特率除数寄存器。例11-26的最后部分描述了在下面几段里将介绍的FIFO控制寄存器的功能。

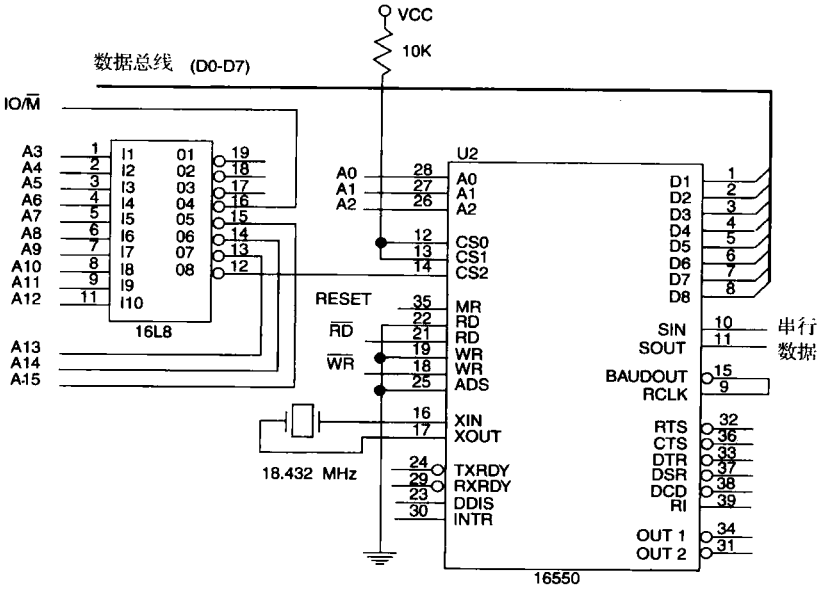


图11-45 与8088微处理器连接，端口地址为00F0H～00F7H的16550

例11-26

```
;图11-45的初始化对话
;波特率为9600,7位数据,奇校验,一个停止位

LINE    EQU   0F3H
LSB     EQU   0F0H
MSB     EQU   0F1H
FIFO    EQU   0F2H

INIT    PROC  NEAR

        MOV   AL,10001010B      ;装入波特率除数
        OUT   LINE,AL

        MOV   AL,120            ;编程设置波特率9600
        OUT   LSB,AL
        MOV   AL,0
        OUT   MSB,AL

        MOV   AL,00001010B      ;7位数据,奇校验
        OUT   LINE,AL           ;一个停止位

        MOV   AL,00000111B      ;使能发送器和
        OUT   FIFO,AL           ;接收器

        RET

INIT    ENDP
```

在线路控制寄存器和波特率除数被编程到16550中后，16550仍不能工作，还必须编程FIFO控制寄存器，它位于图11-45电路中的端口F2H。

图11-46给出了16550的FIFO控制寄存器。该寄存器允许发送器和接收器（位0 = 1），并清除发送器和接收器的FIFO，它还提供了对16550中断的控制，这将在第12章讨论。请注意在例11-26的最后部分将7装入FIFO控制寄存器中，从而允许发送器和接收器，并清除二者的FIFO。16550现在准备工作，但没有中断，因为当系统RESET信号将MR（主复位）输入引脚置为逻辑1时，中断已被自动禁止了。

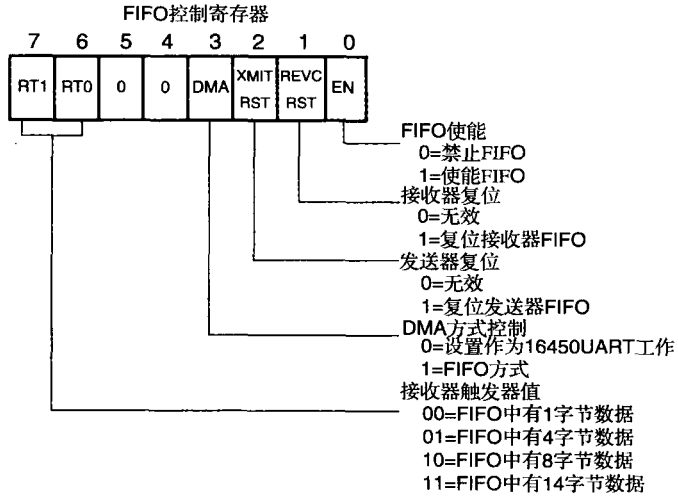


图11-46 16550 UART的FIFO控制寄存器

发送串行数据

通过16550发送或接收串行数据之前，需要了解线路状态寄存器（参见图11-47）的功能。线路状态寄存器包含错误状态信息及发送器和接收器的状态，该寄存器在发送或接收一字节数据之前被测试。

假定一个过程（见例11-27）将AH的内容发送给16550并通过其串行数据引脚（SOUT）输出。程序查询TH位以确定发送器是否已准备好接收数据。该过程使用图11-45的电路。

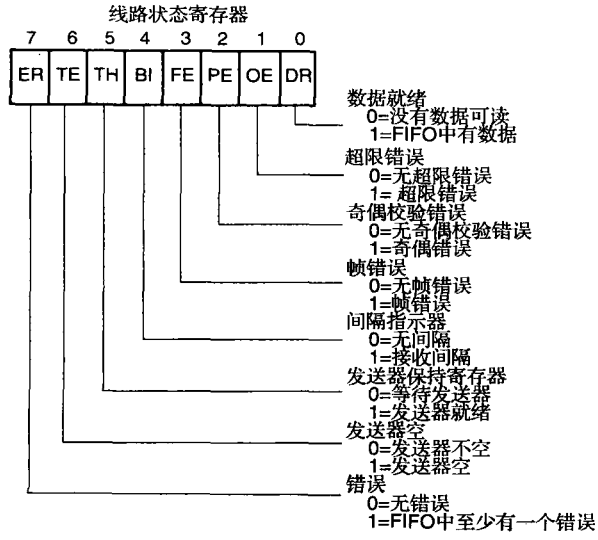


图11-47 16550 UART的线路状态寄存器的内容

例 11-27

```
;该过程通过16550 UART发送AH内容

LSTAT   EQU    0F5H
DATA    EQU    0F0H

SEND    PROC   NEAR USES AX

        .REPEAT                     ;测试TH位
                IN    AL,LSTAT
                TEST AL,20H
        .UNTIL !ZERO?

        MOV   AL,AH                 ;发送数据
        OUT   DATA,AL
        RET

SEND    ENDP
```

接收串行数据

为从 16550 中读取接收到的信息，需测试线路状态寄存器的 DR 位。例 11-28 列出了一个过程，它测试 DR 位以确定 16550 是否已接收到数据。在接收数据时，过程测试有无错误。若检测到一个错误，则过程在 AL 中返回一个 ASCII 码“?”；若未出现错误，则过程在 AL 中返回的是接收字符。

例 11-28

```
;该过程从16550 UART接收数据并返回AL中

LSTAT   EQU    0F5H

DATA    EQU    0F0H

REVC    PROC   NEAR

        .REPEAT
                IN    AL,LSTAT      ;测试DR位
                TEST AL,1
        .UNTIL !ZERO?

        TEST AL,0EH                 ;测试有无任何错误
        .IF ZERO?                   ;无错误
                IN    AL,DATA
        .ELSE                       ;有错误
                MOV   AL,'?'
        .ENDIF
        RET

RECV    ENDP
```

UART 错误

16550 检测到的错误类型是奇偶校验错误、帧错误和超限错误。**奇偶校验错误**（**parity error**）表明接收到的数据包含错误的奇偶校验位，**帧错误**（**framing error**）表明起始位和停止位不在正确的位置上，**超限错误**（**overrun error**）表明数据已超出内部接收器 FIFO 缓冲器。这些错误在正常操作中是不应出现的。如果出现奇偶校验错误，则表明在接收中有噪声干扰。如果接收器正以一个不正确的波特率接收数据，那么就会出现帧错误。只有在接收器 FIFO 满之前程序错误从 UART 读数据时才会出现超限错误。此例没有测试 BI（间隔指示位）以检测间隔状态。注意，一次间隔是指 UART 的 SIN 引脚上的两帧连续为逻辑 0。其余的用于中断控制和调制解调器控制的寄存器，将在第 12 章介绍。

11.6 模/数转换器（ADC）与数/模转换器（DAC）

模/数转换器（ADC）与数/模转换器（DAC）用于微处理器与模拟世界之间的接口。微处理器监视和控制的许多事件都是模拟事件，常常包括监视各种形式的事件甚至是语音、控制电机及类似设备

为将微处理器与这些事件连接起来，我们必须了解 ADC 和 DAC 的接口与控制，它们在模拟量和数字量之间进行转换。

11.6.1 DAC0830 数/模转换器

一种相当普遍且低成本的数/模转换器是 DAC0830（国家半导体公司产品）。该器件是一个 8 位转换器，它将一个 8 位二进制数转换成模拟电压。其他转换器可将 10 位、12 位或 16 位二进制数转换成模拟电压。由转换器产生的电压阶梯数目等于二进制输入组合的数目，因此，一个 8 位转换器可以产生 256 种不同电压值，一个 10 位转换器可以产生 1024 种不同电压值，依次类推。DAC0830 是一种中速转换器，它在大约 1.0μs 内将一个数字量输入转换成模拟量输出。

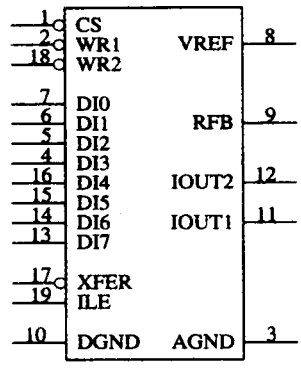


图 11-48 DAC0830 数/模转换器的引脚图

图 11-48 给出了 DAC0830 的引脚图。该器件有一组 8 位数据总线用于输入数字码，一对标识为 IOUT1 和 IOUT2 的模拟输出被设计作为外部运算放大器的输入。由于这是一个 8 位转换器，所以它的输出阶梯电压定义为 $-V_{REF}$（参考电压）除以 255 所得的商。例如，若参考电压为 -5.0V，则其输出阶梯电压为 +0.0196V。注意，输出电压与参考电压极性相反。如果输入 1001 0010_2 加到该器件上，则输出电压将是阶梯电压与 1001 0010_2 的乘积，本例为 +2.862V。将参考电压改为 -5.1V，则阶梯电压变为 +0.02V。阶梯电压也常被称为转换器的**分辨率**（**resolution**）。

DAC0830 的内部结构

图 11-49 给出了 DAC0830 的内部结构。注意该器件包含 2 个内部寄存器，第一个是保持寄存器，而第二个与 R-2R 内部 T 型网络转换器相连。两个锁存器允许保持一个字节的同时转换另一字节。在多数情况下，我们禁止第一个锁存器，只使用第二个锁存器将数据输入转换器中，这是通过将 ILE 置为逻辑 1 并将$\overline{CS}$（片选）置为逻辑 0 来实现的。

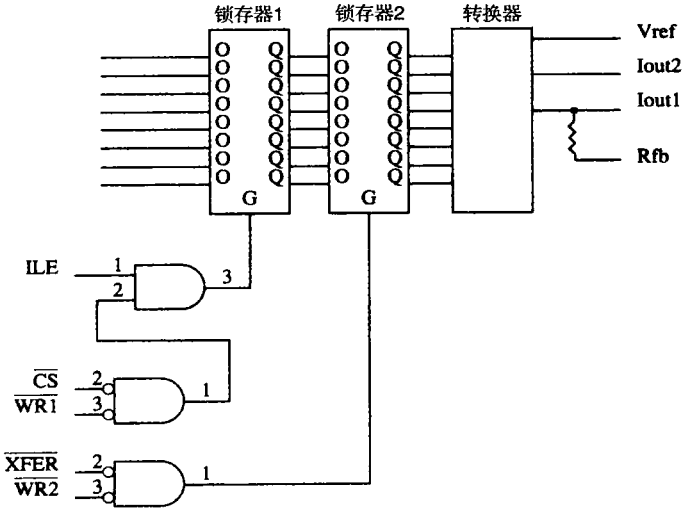


图 11-49 DAC0830 的内部结构

DAC0830 中的两个锁存器均为透明锁存器，即当锁存器的 G 输入为逻辑 1 时，数据通过锁存器，而当 G 输入变为逻辑 0 时，数据被锁存或保持。转换器有一个参考输入引脚（V_{REF}）建立满量程电压。如果 -10V 被加在 V_{REF}引脚上，则满量程（11111111_2）输出电压为 +10V。转换器内部 R-2R T 型网络的输出出现在 IOUT1 和 IOUT2 上，这两个输出被设计成用于 741 运算放大器或类似的器件。

DAC0830 与微处理器的连接

DAC0830 与微处理器的连接如图 11-50 所示。这里，PLD 用来译码 DAC0830，其 8 位 I/O 端口地址为 20H。一旦执行 OUT 20H，AL 指令，则数据总线 $AD_0 \sim AD_7$ 的内容传送给 DAC0830 中的转换器。741 运算放大器与 -12V 齐纳参考电压一起产生 +12V 的满量程输出电压。运算放大器的输出提供给一个驱动器，以驱动一个 12V 直流电机，此驱动器为达林顿放大器，用于大功率电机。本例显示了驱动电机的转换器，但也可驱动其他设备。

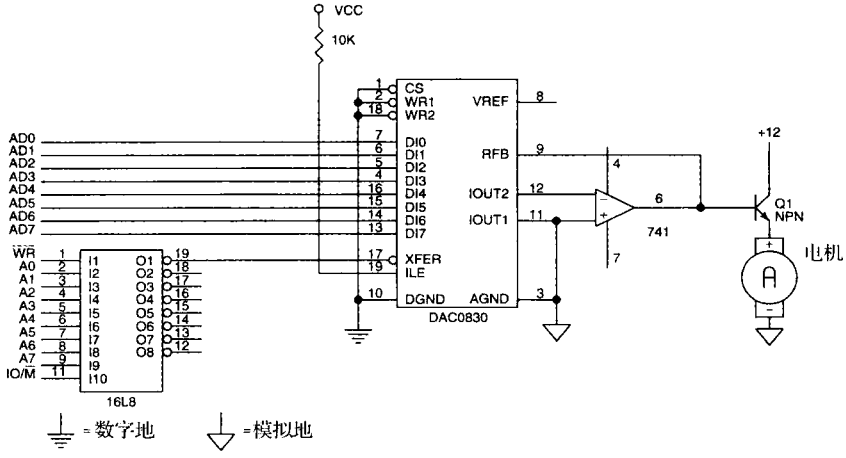


图 11-50 与微处理器连接的 DAC0830，8 位 I/O 地址为 20H

11.6.2 ADC080X 模/数转换器

一种普遍使用且成本较低的 ADC 是 ADC0804，它属于除精度外所有特性都相同的一种转换器系列。该器件与大多数微处理器如 Intel 系列兼容。尽管还有更快的 ADC，并且有比 8 位更高的分辨率，但该器件在许多不需要高精度的应用中仍是理想的。ADC0804 最多需要 100μs 将模拟输入电压转换为数字输出码。

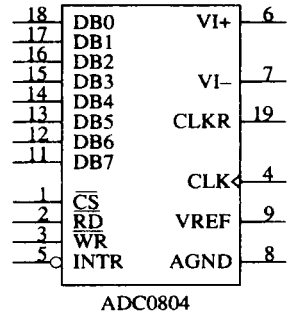


图 11-51 ADC0804 模/数转换器的引脚图

图 11-51 给出了 ADC0804 转换器（国家半导体公司产品）的引脚图。为使转换器工作，$\overline{WR}$引脚加负脉冲，同时$\overline{CS}$接地，从而启动转换进程。由于此转换器需要相当长的时间进行转换，所以用标识为 INTR 的引脚发信号通知转换结束。参见图 11-52 的时序图，它显示了控制信号的交互作用。可以看出，这里用$\overline{WR}$脉冲启动转换器，等待 INTR 回到逻辑 0 电平后，才从转换器读出数据。如果采用

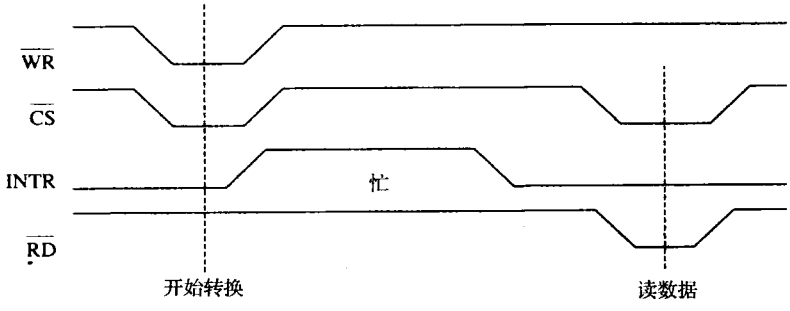


图 11-52 ADC0804 模/数转换器的时序图

至少100μs的时间延迟，则无需测试INTR引脚。另一选择是将INTR引脚与一个中断输入相连，从而在转换完成后产生一个中断。

模拟输入信号

在ADC0804与微处理器连接之前，必须理解模拟输入。ADC0804有2个模拟输入：VIN（+）和VIN（-）。这2个输入与一个内部运算放大器相连，且为差动输入，如图11-53所示。差动输入由运算放大器求和，产生一个信号给内部模/数转换器。图11-53给出了使用这些差动输入的几种方式。第1种方式（见图11-53（a））使用可在0V～+5.0V间变化的单一输入，第2种方式（见图11-53（b））显示加在VIN（-）引脚上的是一个可变电压，从而可对VIN（+）的0参考电压进行调整。

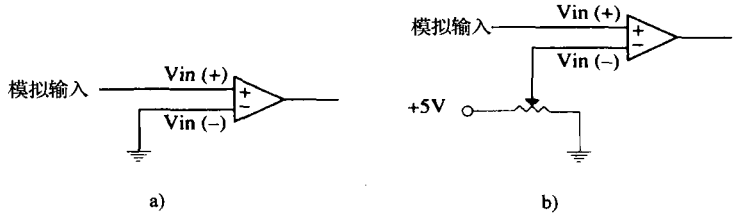


图11-53 ADC0804转换器的模拟输入

a）检测0V～+5.0V输入 b）检测与接地电压有一偏移量的输入

产生时钟信号

ADC0804需要一个时钟源才能工作。时钟可以是加在CLK IN引脚上的一个外部时钟，也可以由一个RC电路产生。时钟频率的允许范围为100KHz～1460KHz，使用尽可能接近1460KHz的频率是最为理想的，这样转换时间可达到最小。

如果时钟由RC电路产生，则将CLK IN和CLK R引脚与RC电路相连，如图11-54所示。使用这种连接，时钟频率由下式计算：

$$Fclk = 1/(1.1RC)$$

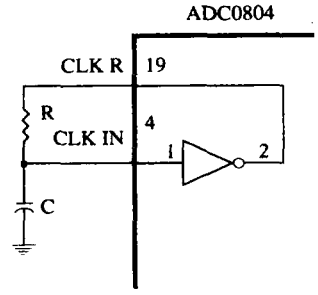


图11-54 RC电路与ADC0804的CLK IN和CLK R引脚的连接

ADC0804与微处理器的连接

ADC0804与微处理器的连接如图11-55所示。注意，V_{REF}信号不与任何信号相连是正常的。假定ADC0804被译

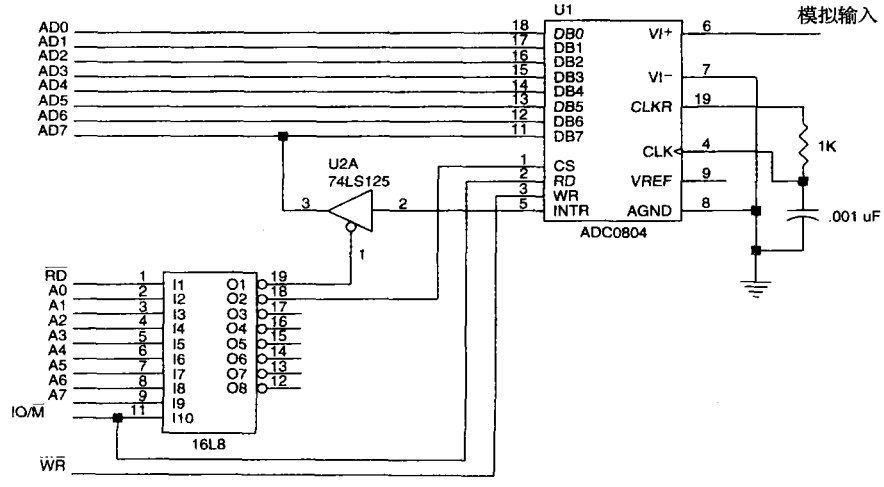


图11-55 ADC0804与微处理器的连接

码，其8位I/O端口地址40H用于数据，端口地址42H用于INTR信号，需要一个过程来启动ADC并从ADC中读出数据。此过程在例11-29中列出。注意INTR位被查询，如果它变为逻辑0，则该过程结束，同时AL中包含转换的数字代码。

例11-29

```
ADC     PROC    NEAR

        OUT   40H,AL
        .REPEAT                 ;测试 INTR
              IN    AL,42H
              TEST AL,80H
        .UNTIL ZERO?
        IN AL,40H
        RET

ADC     ENDP
```

11.6.3 使用ADC0804和DAC0830的实例

本节给出了一个使用ADC0804和DAC0830来捕获和重放音频信号或语音信号的实例。过去，我们常使用语音合成器产生语音信号，但语音质量很差。对于人类的语音，可以使用ADC0804捕获音频信号，并将它存储在存储器中，以后再通过DAC0830重放。

图11-56给出了将ADC0804连接在I/O端口0700H和0702H所需的电路。DAC0830的I/O端口地

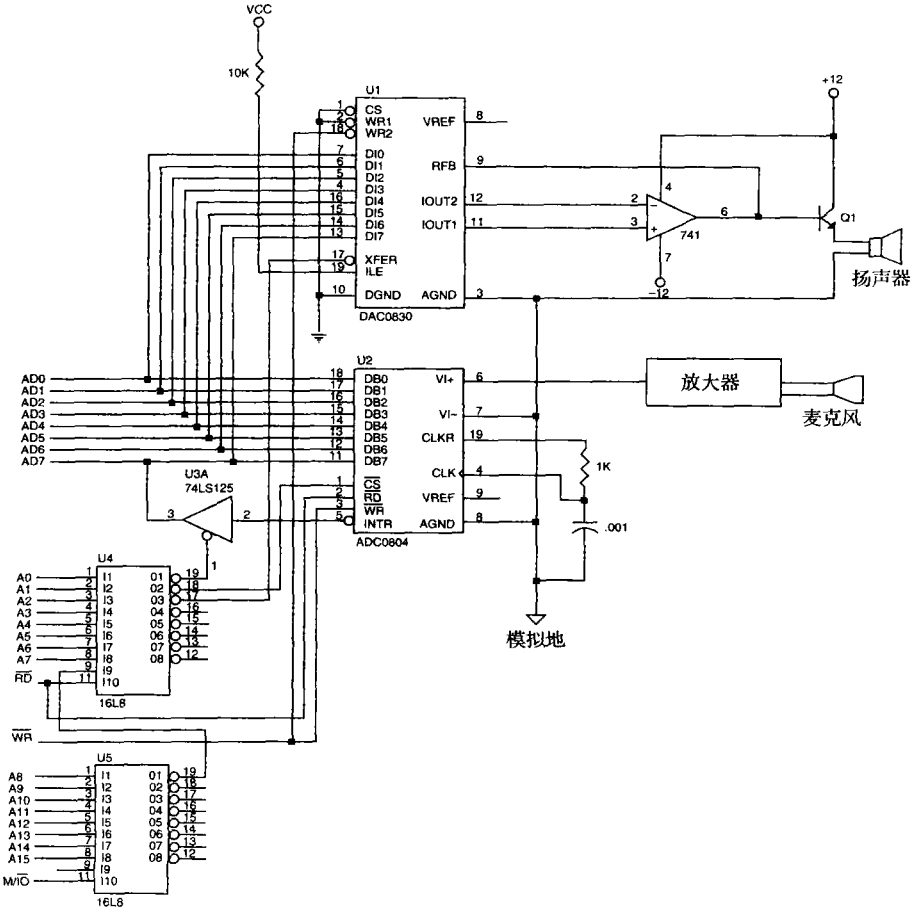


图11-56 存储语音并通过扬声器重放的电路

址为0704H。这些I/O端口位于16位微处理器如8086或80386SX的低位存储体中。用于运行这些转换器的程序列于例11-30中。该软件读1秒的语音信号，然后重放10次。一个过程调用READS读语音，另一个过程调用PLAYS重放该语音。语音被采样和存储在称为WORDS的存储器段中。选择采样率为每秒钟采样2048次，从而提供可接受的语音信号。

例11-30

```
; 软件记录1秒钟语音信号，并把它重放10次
; 假定时钟频率为20MHz（80386EX微处理器）

READS   PROC  NEAR USES ECX DX

        MOV   ECX,2048                  ;次数为2048
        MOV   DX,700H                   ;寻址端口700H
        .REPEAT
              OUT   DX,AL               ;开始转换
              ADD   DX,2                ;访问状态端口
                .REPEAT                 ;等待转换
                        IN    AL,DX
                        TEST  AL,80H
                .UNTIL ZERO?
                SUB   DX,2              ;访问数据端口
                IN    AL,DX             ;取得数据
                MOV   WORDS[ECX-1]
                CALL  DELAY             ;等待1/2048秒
         .UNTILCXZ
        RET

READS   ENDP

PLAYS   PROC  NEAR USES DX ECX

        MOV   ECX,2048                  ;次数为2048
        MOV   DX,704H                   ;寻址DAC
        .REPEAT
              MOV   AL,WORDS[EAX-1]
              OUT   DX,AL               ;发送数据到DAC
              CALL  DELAY               ;等待1/2048秒
         .UNTILCXZ
        RET
PLAYS   ENDP

DELAY   PROC    NEAR USES CX

        MOV   CX,888

        .REPEAT                         ;等待1/2048秒
        .UNTILCXZ
        RET

DELAY     ENDP
```

11.7 小结

1）8086～Core2微处理器有两种基本类型的I/O指令：IN和OUT。IN指令将来自外部I/O设备的数据输入到AL（8位）或AX（16位）寄存器，IN指令有固定端口指令、可变端口指令以及串指令（80286～Pentium 4）INSB或INSW。OUT指令从AL或AX输出数据到外部I/O设备，它也有固定端口指令、可变端口指令以及串指令OUTSB或OUTSW。固定端口指令使用8位I/O端口地址，而可变端口和串指令使用DX寄存器中的16位端口地址。

2）独立编址I/O有时称为直接I/O，使用独立的I/O空间映射表，从而释放整个存储器给程序使用。独立编址I/O使用IN和OUT指令在I/O设备与微处理器之间传送数据。I/O映射表的控制结构使用$\overline{IORC}$（I/O读控制）和$\overline{IOWC}$（I/O写控制），加上存储体选择信号$\overline{BHE}$和$\overline{BLE}$（8086和80286上的A_0）来影响I/O传送。早期8086/8088使用M/$\overline{IO}$(IO/$\overline{M}$)信号与$\overline{RD}$和$\overline{WR}$产生I/O控制信号。

3）存储器映像 I/O 使用部分存储器空间进行 I/O 传送，这就减少了可用存储器空间，但它无须使用$\overline{IORC}$和$\overline{IOWC}$信号进行 I/O 传送。另外，使用任何寻址方式寻址存储单元的任一指令，都可用于在微处理器与 I/O 设备之间使用存储器映像 I/O 传送数据。

4）所有输入设备都经过缓冲，这样在执行 IN 指令期间 I/O 数据只与数据总线相连。缓冲器既可以构造于一个可编程外围设备内部，也可以是独立的部分。

5）所有输出设备在执行 OUT 指令期间都使用锁存器来捕获输出数据。这是必要的，因为对于一条 OUT 指令，数据出现在数据总线上的时间不到 100ns，而大多数输出设备需要数据保持的时间更长一些。在许多情况下，锁存器构造在外围设备内部。

6）握手或查询是用几条控制线使两个独立设备同步的方式。例如，计算机通过输入来自打印机的 BUSY 信号询问打印机是否“忙”。如果不忙，则计算机输出数据给打印机，并用一个数据选通（$\overline{DS}$）信号通知打印机数据有效。这种在计算机与打印机之间的通信就是一种握手或查询。

7）对于大多数基于开关的输入设备和大多数不是 TTL 兼容的输出设备，都需要使用接口。

8）对于固定端口 I/O 指令，I/O 端口号出现在地址总线 $A_7 \sim A_0$ 上；对于可变端口 I/O 指令，I/O 端口号出现在 $A_{15} \sim A_0$ 上（注意，对于 8 位端口，$A_{15} \sim A_8$ 均为 0）。在这两种情况下，超过 A_{15} 的地址位没有定义。

9）由于 8086/80286/80386SX 微处理器包含一个 16 位数据总线，而 I/O 地址访问的是以字节为单位的存储单元，所以 I/O 空间也组织成存储体，如存储系统一样。为将一个 8 位 I/O 设备与 16 位数据总线相连，常需要独立的 I/O 写选通（高位和低位）进行 I/O 写操作。同样，80486 和 Pentium ~ Pentium 4 也将 I/O 组织成存储体。

10）I/O 端口译码器与存储器地址译码器非常相似，不同于译码整个地址，I/O 端口译码器对可变端口指令只译码 16 位地址，而对固定 I/O 指令常常只译码 8 位端口号。

11）82C55 是一种可编程外围设备接口（PIA），它有 24 个可编程引脚，分为 2 组（A 组和 B 组），每组 12 个引脚。82C55 有 3 种操作方式：简单 I/O（方式 0）、选通 I/O（方式 1）以及双向 I/O（方式 2）。当 82C55 与工作在 8MHz 下的 8086 相连时，我们插入 2 个等待状态，这是因为微处理器的速度高于 82C55 可处理的速度。

12）LCD 显示器需要相当长的程序来控制，但它可以显示 ASCII 编码信息。

13）8254 是可编程间隔定时器，它有 3 个以二进制或二进制编码的十进制（BCD）计数的计数器。每个计数器相互独立，并按 6 种不同方式操作。计数器的 6 种方式是：①事件计数器；②可重触发单稳多谐振荡器；③脉冲发生器；④方波发生器；⑤软件触发脉冲发生器；⑥硬件触发脉冲发生器。

14）16550 是可编程通信接口，能够接收和发送异步串行数据。

15）DAC0830 是一个 8 位数/模转换器，它在 1.0μs 内将一个数字信号转换为模拟电压。

16）ADC0804 是一个 8 位模/数转换器，它在 100μs 内将一个模拟信号转换为数字信号。

11.8 习题

1. 解释 IN 指令和 OUT 指令的数据流动方式。
2. 固定 I/O 指令的 I/O 端口号存储在何处？
3. 可变 I/O 指令的 I/O 端口号存储在何处？
4. 串 I/O 指令的 I/O 端口号存储在何处？
5. 16 位 IN 指令将数据输入到哪个寄存器？
6. 描述 OUTSB 指令的操作。
7. 描述 INSW 指令的操作。
8. 比较存储器映像 I/O 系统与独立编址 I/O 系统。
9. 什么是基本输入接口？
10. 什么是基本输出接口？
11. 解释术语“握手”应用于计算机 I/O 系统时的情形。
12. 一个偶数 I/O 端口地址出现在 8086 微处理器的_____ I/O 存储体中。
13. 在 Pentium 4 中，哪块存储体包含有 I/O 端口号 000AH？
14. 在 Pentium 4 或 Core2 微处理器中，有多少个 I/O 存储体？
15. 给出一个可以产生高低 I/O 写选通的电路图。
16. 触点抖动消除器的作用是什么？
17. 设计一个正确驱动继电器的接口，继电器电压为 12V，需要 150mA 的线圈电流。
18. 设计一个继电器线圈驱动器，使之能控制 5V 的继电器，且其线圈电流为 60mA。
19. 为 16 位的微处理器设计一个 I/O 端口译码器，使用一个 74ALS138，为下列 8 位 I/O 端口地址 10H、12H、14H、16H、18H、1AH、1CH 及 1EH 产生低位存储体 I/O 选通信号。
20. 设计一个 I/O 端口译码器，使用一个 74ALS138，为下列 8 位 I/O 端口地址 11H、13H、15H、17H、19H、1BH、1DH 及 1FH 产生高位存储体 I/O 选通信号。
21. 设计一个 I/O 端口译码器，使用一个 PLD，为下列 16 位 I/O 端口地址 1000H ~ 1001H、1002H ~ 1003H、1004H ~ 1005H、1006H ~ 1007H、1008H ~ 1009H、100AH ~ 100BH、100CH ~ 100DH 及 100EH ~ 100FH 产生 16 位 I/O 选通信号。
22. 设计一个 I/O 端口译码器，使用一个 PLD，产生下列

低位存储体 I/O 选通信号：00A8H、00B6H 和 00EEH。

23. 设计一个 I/O 端口译码器，使用一个 PLD，产生下列高位存储体 I/O 选通信号：300DH、300BH、1005H 和 1007H。
24. 为什么$\overline{BHE}$和$\overline{BLE}$(A_0)在一个 16 位端口地址译码器中均可被忽略？
25. I/O 端口地址为 0010H 的一个 8 位 I/O 设备，在 Pentium 4 中与哪些数据总线引脚相连？
26. I/O 端口地址为 100DH 的一个 8 位 I/O 设备，在 Core2 中与哪些数据总线引脚相连？
27. 82C55 有多少个可编程 I/O 引脚？
28. 列出 82C55 中属于 A 组和 B 组的引脚。
29. 哪 2 个 82C55 引脚完成内部 I/O 端口地址选择？
30. 82C55 上的$\overline{RD}$引脚与 8086 系统的哪个控制总线引脚相连？
31. 使用一个 PLD，将一个 82C55 连接到 8086 微处理器上，使其 I/O 地址为 0380H、0382H、0384H 和 0386H。
32. 当 82C55 被复位时，其 I/O 端口均被初始化为_____。
33. 82C55 有哪 3 种操作方式？
34. 82C55 的选通输入操作中$\overline{STB}$信号有什么用途？
35. 设计一个时间延迟的过程，使之能使 2.0GHz 的 Pentium 4 延时 80μs。
36. 设计一个时间延迟的过程，使之能使 3.0GHz 的 Pentium 4 延时 12ms。
37. 解释简单 4 线圈步进电机的工作。
38. 在 82C55 的选通输入操作中用什么来置位 IBF 引脚？
39. 写出在选通输入操作期间将 82C55 的 PC_7 引脚置为逻辑 1 的程序。
40. 在 82C55 的选通输入方式操作中是如何允许中断请求引脚的？
41. 在 82C55 的选通输出操作中，$\overline{ACK}$信号的用途是什么？
42. 在 82C55 的选通输出操作中用什么来清除$\overline{OBF}$信号？
43. 写出一个程序，确定当 82C55 工作在选通输出方式下时 PC_4 是否为逻辑 1。
44. 在 82C55 的双向操作期间使用哪一组引脚？
45. 在 82C55 的方式 2 操作期间哪些引脚是通用 I/O 引脚？
46. 描述使用 LCD 显示器时是如何清除显示的？
47. 在 LCD 显示器中是如何选择显示位置的？
48. 写一个短程序，使空 ASCII 字符串显示在 LCD 显示器的显示位置 6。
49. 在 LCD 显示器中如何测试“忙”标志？
50. 如何修改图 11-25，使之与 3 行 5 列的键盘矩阵一起工作？
51. 通常用于键盘去抖动的时间是多少？
52. 设计一个 3×4 的电话键盘。在设计中，你需要用一个查找表来进行适当的键码转换。
53. 8254 间隔定时器工作在从 DC 到_____Hz 之间。
54. 8254 的每个计数器可以用多少种不同方式工作？
55. 连接 8254 使其工作在 I/O 端口地址 XX10H、XX12H、XX14H 和 XX16H。
56. 写一个程序编程计数器 2，使其在 CLK 输入为 8MHz 时产生一个 80KHz 的方波。
57. 为计数 300 个事件，编程到 8254 计数器中的计数值是多少？
58. 如果一个 16 位计数值被编程到 8254 中，那么首先编程哪一字节的计数值？
59. 解释 8254 中读回控制字是如何工作的。
60. 编程 8254 的计数器 1 使其产生一个连续的系列脉冲，高电平时间为 100μs，低电平时间为 1μs，明确指出本任务需要的 CLK 频率。
61. 在本章给出的电机速度和方向控制电路中，为什么 50% 的占空比使电机保持静止？
62. 什么是异步串行数据？
63. 什么是波特率？
64. 编程 16550，使其使用 6 个数据位、偶校验、一个停止位，使用 18.432MHz 时钟的 19 200 波特率（假定 I/O 端口为 20H 和 22H）。
65. 如果 16550 要产生一个波特率为 2400 波特的串行信号，且波特率除数编程为 16，那么信号源的频率是多少？
66. 解释下列术语：单工、半双工和全双工。
67. 16550 是如何被复位的？
68. 写一个过程，使 16550 从由 SI（SI 已由外部装入数据）寻址的数据段（DS 已由外部装入数据）里的小缓冲器中发送 16 字节数据。
69. DAC0830 在大约_____时间内将一个 8 位数字输入转换成一个模拟输出。
70. 如果参考电压为 -2.55V，那么 DAC0830 输出的阶梯电压是多少？
71. 把 DAC0830 连接到 8086 上，使其 I/O 端口为 400H。
72. 为第 71 题的接口设计一个程序，使 DAC0830 产生一个三角形电压波形，该波形的频率必须大约为 100Hz。
73. ADC080X 需要大约_____时间将一个模拟电压转换成数字代码。
74. ADC080X 上的$\overline{WR}$引脚有什么用途？
75. ADC080 上的 INTR 引脚的作用是什么？
76. 连接 ADC080X，使其 I/O 端口 0260H 用于数据，0270H 用于测试 INTR 引脚。
77. 为第 76 题的 ADC080X 设计一个程序，使它每 100ms 读一个输入电压，并将结果存储在 100H 字节长的存储器数组中。
78. 使用 C++ 语言内嵌汇编代码的方式重写例 11-29 的代码。

第 12 章　中　　断

引言

本章将扩充前面的基本 I/O 及可编程外围设备接口内容，介绍一种称为中断处理 I/O 的技术。中断是一个由硬件激发的过程，它中断当前正在执行的任何程序。本章对整个 Intel 系列微处理器的中断结构提供一些例子并进行详细解释。

目的

读者学习完本章后将能够：

1）解释 Intel 系列微处理器的中断结构。
2）解释软件中断指令 INT、INTO、INT 3 和 BOUND 的操作。
3）解释中断允许标志位（IF）是如何修改中断结构的。
4）描述陷阱中断标志位（TF）的功能及产生陷阱跟踪的操作。
5）设计控制较低速度外围设备的中断服务过程。
6）使用 8259A 可编程中断控制器及其他技术扩展微处理器的中断结构。
7）解释实时时钟的用途与操作。

12.1　基本中断处理

本节将讨论基于微处理器系统的中断功能，以及 Intel 系列微处理器现有中断的结构和特性。

12.1.1　中断的目的

在与以相对较低数据传输速率提供或接收数据的 I/O 设备接口时，中断特别有用。例如在第 11 章中，列举了使用 82C55 选通输入操作的键盘实例。在该例中，软件查询 82C55 及其 IBF 位以确定数据是否可从键盘得到。如果键盘使用者每秒键入一个字符，那么在每次键击之间，82C55 的软件需要整整 1 秒时间来等待键入另一个键，这种处理方式很浪费时间，因此设计者开发了另一种处理方式，即用中断处理来处理这种情况。

与查询技术不同，中断处理允许微处理器在键盘操作者正在考虑下一个键入什么键时执行其他程序。一旦按下一个键，键盘编码器就会去除开关抖动，并产生一个中断微处理器的脉冲。这样，微处理器一直执行其他程序，直到确实又有键按下时才读此键并返回被中断的程序。结果是，当操作者正在键入一个文件并考虑下一个键入什么时，微处理器可以打印报告或完成任何其他任务。

图 12-1 给出了一条时间线，表明打字员在键盘上键入数据，打印机从存储器取出数据以及一个正在执行的程序。该程序是主程序，它被每次键击和每个要在打印机上打印的字符所中断。注意，由键盘中断调用的键盘中断服务程序以及打印机中断服务程序只占用极少的执行时间。

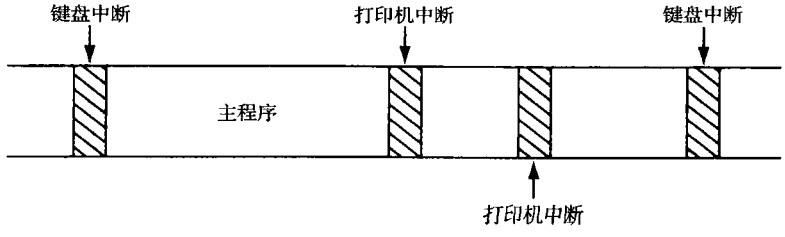


图 12-1　在一个典型系统中表明中断使用情况的一条时间线

12.1.2　中断

整个 Intel 系列微处理器的中断包括 2 个申请中断的硬件引脚（INTR 和 NMI），和 1 个响应 INTR 中断申请的硬件引脚（INTA）。除这些引脚外，微处理器还有软件中断 INT、INTO、INT3 和 BOUND，还有用在中断结构中的 2 个标志位 IF（interrupt flag，中断标志）和 TF（trap flag，陷阱标志）和一个特殊的返回指令 IRET（在 80386、80486 或 Pentium ～ Pentium 4 中的 IRETD）。

中断向量

中断向量与向量表对于理解硬件和软件中断是非常重要的。**中断向量表（interrupt vector table）**位于存储器的前 1024 字节中，地址为 000000H ～ 0003FFH。它包含 256 个不同的 4 字节中断向量。**中断向量（interrupt vector）**包含中断服务程序的地址（段和偏移）。

图 12-2 给出了微处理器的中断向量表。前 5 个中断向量在 8086 ～ Pentium 的所有 Intel 系列微处理器中都是相同的。其他中断向量存在于 80286 及向上兼容的 80386、80486 和 Pentium ～ Core2 中，但不向下兼容 8086 或 8088。Intel 保留前 32 个中断向量为 Intel 各种微处理器系列成员专用，最后 224 个向量可用作用户中断向量。每个向量为 4 字节长，包含中断服务程序的**入口地址（starting address）**。向量的前 2 个字节包含偏移地址，后 2 个字节包含段地址。

下面描述微处理器每个专用中断的功能：

类型 0　**除法出错**——发生在除法结果溢出或试图除以零时。

类型 1　**单步或陷阱**——若陷阱标志位被置位，则在每条指令执行后发生中断。一旦接受此中断，则 TF 位被清除，使中断服务程序以全速执行（关于此中断的更多细节将在本节后面部分介绍）。

类型 2　**非屏蔽硬件中断**——是将微处理器的 NMI 输入引脚置为逻辑 1 的结果。此输入是非屏蔽的，这意味着它不能被禁止。

类型 3　**一字节中断**，一个特殊的单字节指令（INT 3）使用该向量访问中断服务程序。INT3 指令常用于调试时存储程序的断点。

类型 4　**溢出**，INTO 指令的专用向量。如果由溢出标志（overflow flag，OF）反映的溢出情况存在，则 INTO 指令中断正执行的程序。

类型 5　**边界**，边界指令将寄存器与存储于存储器中的边界值相比较。如果寄存器的内容大于或等于存储器的第一个字，并小于或等于第二个字，则不发生中断，因为寄存器的内容在边界之内。如果寄存器的内容超出边界，则发生类型 5 中断。

类型 6　**无效操作码**，一旦在程序中遇到未定义的操作码时发生此中断。

类型 7　**协处理器不存在**，正如机器状态字（machine status word，MSW）的协处理器控制位所指示的，当在一个系统中未找到协处理器时发生此中断。如果执行了 ESC 或 WAIT 指令且未找到协处理器，则发生类型 7 异常（或中断）。

类型 8　**双中断出错**，在同一指令期间发生 2 个独立的中断时激活此中断。

类型 9　**协处理器段超限**，如果 ESC 指令（协处理器操作码）的存储器操作数扩展超出偏移地址 FFFFH，则发生该中断。

类型 10　**无效任务状态段**，在保护模式下，由于段限制区域不是 002BH 或更高，则 TSS 为无效，此时发生该中断。大多数情况下是由于 TSS 未被初始化而引起的。

类型 11　**段不存在**，当保护模式描述符中的 P 位（P = 0）指示段不存在或无效时发生该中断。

类型 12　**堆栈段超限**，如果保护模式中堆栈段不存在（P = 0）或者堆栈段超限，则发生该中断。

类型 13　**一般性保护**，在 80286 ～ Core2 保护模式系统中，若违反了大多数保护模式，则发生此中断（这些错误在 Windows 中表现为一般性保护错）。这些保护违规列出如下：

1）描述符表边界超限。

2）违反特权规则。

3）装入了无效的描述符段类型。

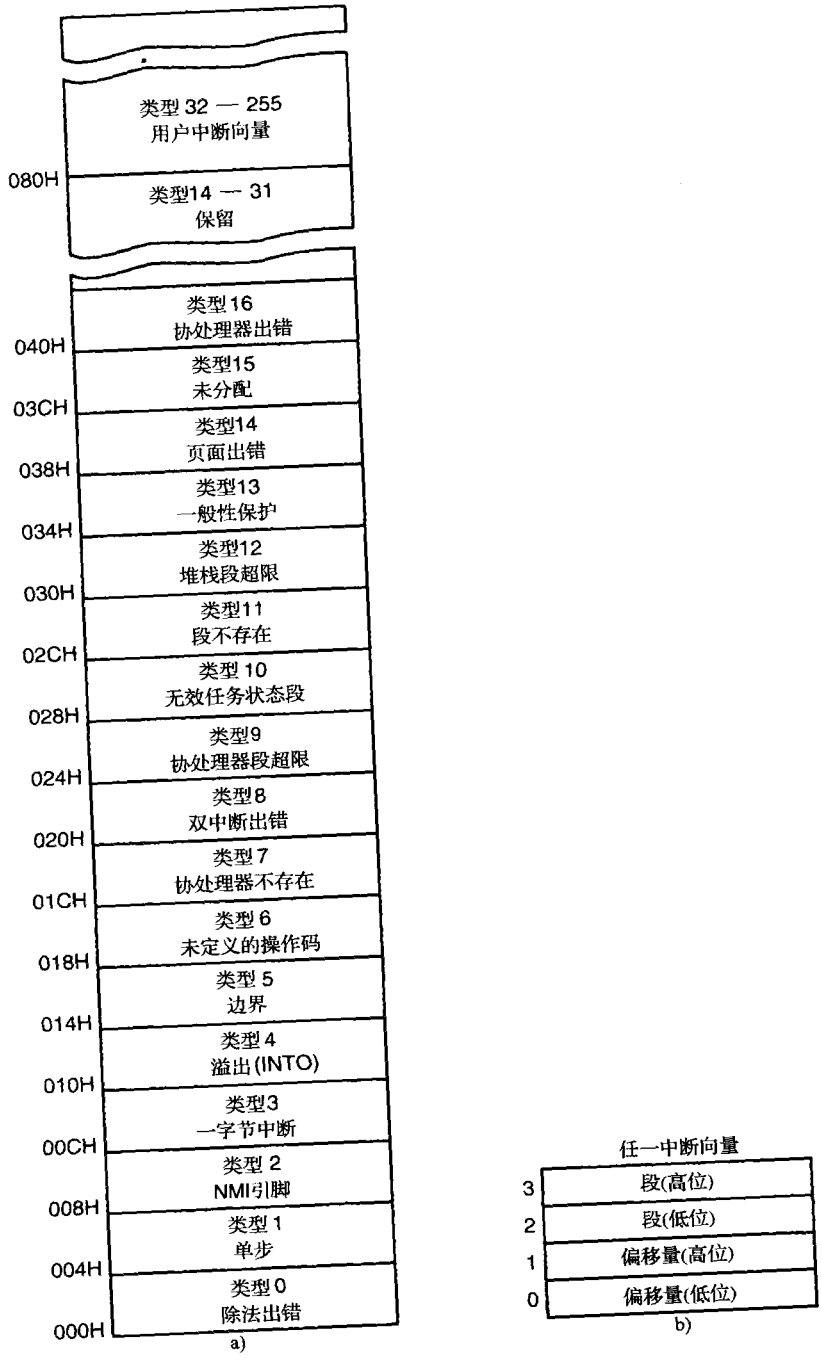


图 12-2 微处理器的中断向量表和中断向量内容

a）微处理器的中断向量表 b）中断向量的内容

4）对被保护代码段的写操作。

5）从只能执行的代码段读取数据。

6）对只读数据段的写操作。

7）段边界超限。

8）当执行 CTS、HLT、LGDT、LIDT、LLDT、LMSW 或 LTR 时，CPL = IOPL。

9）当执行 CLI、IN、INS、LOCK、OUT、OUTS 和 STI 时，CPL > IOPL。

类型 14　页面出错，在 80386、80486 和 Pentium ～ Core2 微处理器中，访问页面出错的存储器或代码时发生此中断。

类型 16　协处理器出错，一旦 80386、80486 和 Pentium ～ Core2 微处理器的 ESCape 或 WAIT 指令发生协处理器错误（ERROR = 0）时发生此中断。

类型 17　对齐检查，指示字和双字数据存储在奇数存储单元（或一个双字存储在不正确的存储单元）。此中断只在 80486 和 Pentium ～ Core2 微处理器中有效。

类型 18　机器检查，在 Pentium ～ Core2 微处理器中激活一个系统存储管理模式中断。

12.1.3　中断指令：BOUND、INTO、INT、INT 3 和 IRET

在微处理器现有的 5 个软件中断指令中，INT 和 INT 3 非常相似，BOUND 和 INTO 是条件中断，IRET 是一个专用的中断返回指令。

BOUND 指令有 2 个操作数，它比较寄存器和 2 个字的存储器数据。例如，若执行指令“BOUND AX，DATA”，则 AX 与 DATA 和 DATA + 1 中内容相比较，还与 DATA + 2 和 DATA + 3 中内容相比较。如果 AX 小于 DATA 和 DATA + 1 中内容，则发生类型 5 中断。如果 AX 大于 DATA + 2 和 DATA + 3 中内容，也发生类型 5 中断。如果 AX 在这 2 个存储器字的范围内，则不发生中断。

INTO 指令检查溢出标志（OF）。如果 OF = 1，则 INTO 指令调用入口地址存于中断向量类型 4 中的过程。如果 OF = 0，则 INTO 指令不执行任何操作，程序中的下一指令顺序执行。

INT n 指令调用入口地址存于向量号 n 中的中断服务程序。例如，一个 INT 80H 或 INT 128 指令调用入口地址存于向量类型号 80H（000200H ～ 000203H）中的中断服务程序。为确定向量地址，只要将向量类型号（n）乘以 4，它给出了 4 字节长中断向量的起始地址。例如，INT 5 = 4 × 5 即 20（14H），INT 5 向量从地址 000014H 开始，直到 000017H 为止。每个 INT 指令存储在 2 字节的存储器中：第一个字节包含操作码，第二个字节包含操作中断类型号。惟一例外的是 INT 3 指令。这是一个单字节指令，常用作断点中断，这是因为很容易在程序中插入一个一字节指令。断点常用于调试有故障的软件。

IRET 指令是一个特殊的返回指令，用于软件和硬件中断的返回。IRET 指令与远返回很相似，因为它从堆栈检索返回地址。但它不同于近返回，因为它还从堆栈取回标志寄存器的复本。一个 IRET 指令从堆栈中移出 6 个字节：2 个 IP 字节、2 个 CS 字节以及 2 个标志字节。

在 80386 ～ Core2 中还有一个 IRETD 指令，因为这些微处理器可将 EFLAG 寄存器（32 位）压入堆栈，也将保护模式下的 32 位 EIP 压入堆栈。如果工作在实模式下，我们使用 80386 ～ Core2 微处理器的 IRET 指令。如果 Pentium 4 工作在 64 位模式下，使用 IRETQ 指令从中断处返回。IRETQ 指令将 EFLAG 指令寄存器弹给 RFKAGS，同时将 64 位的返回地址放在 RIP 寄存器中。

12.1.4　实模式中断操作

当微处理器执行完当前指令，它按给出的顺序检查下列条件来确定一个中断是否有效：1）指令执行；2）单步；3）NMI；4）协处理器段超限；5）INTR；6）INT 指令。如果一个或多个中断条件出现，则按以下顺序引发事件：

1）标志寄存器的内容压入堆栈。

2）清除中断标志（IF）和陷阱标志（TF）。这样就禁止了 INTR 引脚和陷阱或单步功能。

3）代码段寄存器（CS）的内容压入堆栈。

4）指令指针（IP）的内容压入堆栈。

5）取出中断向量内容，然后送入 IP 和 CS 中，使下一指令执行由中断向量寻址的中断服务程序。

一旦接收到一个中断，微处理器就将标志寄存器、CS 和 IP 的内容压入堆栈；清除 IF 和 TF；跳转到由中断向量寻址的服务程序。在标志被压入堆栈后，IF 和 TF 被清除。当在中断服务程序的末尾遇

到 IRET 指令时，这 2 个标志回到中断前的状态。因此，如果在中断服务程序之前允许中断，那么通过中断服务程序末尾的 IRET 指令可自动再次允许中断。

返回地址（在 CS 和 IP 中）在中断期间压入堆栈。有时，返回地址指向程序中的下一指令，有时指向程序中发生中断的地方。中断类型 0、5、6、7、8、10、11、12 和 13 压入堆栈的返回地址指向错误指令，而不是指向程序中的下一指令，这就使得中断服务程序在某些错误情况下有可能重新执行该指令。

一些保护模式中断（类型 8、10、11、12 和 13）将错误代码紧跟返回地址压入堆栈。错误代码识别引起中断的选择器，如果不包括选择器，则错误代码为 0。

12.1.5 保护模式中断操作

保护模式下的中断与实模式几乎完全相同，但中断向量表不同。保护模式使用一组存储在**中断描述符表（interrupt descriptor table，IDT）**中的 256 个中断描述符取代中断向量。中断描述符表为 256×8（2K）字节长，每个描述符包含 8 个字节。中断描述符表由中断描述符表地址寄存器（IDTR）定位于系统中任何存储单元。

IDT 中每一项包含中断服务程序的地址，该地址形式为段选择符和 32 位偏移地址。IDT 还包含 P 位（表示描述符有效）和描述中断优先级的 DPL 位。图 12-3 给出了中断描述符的内容。

实模式中断向量可被转换成保护模式中断，这通过复制中断向量表中的中断服务程序地址，并将其转换成存储于中断描述符中的 32 位偏移地址来实现。一个选择符和段描述符可被置于全局描述符表中，该表将存储器的前 1MB 定义为中断段。

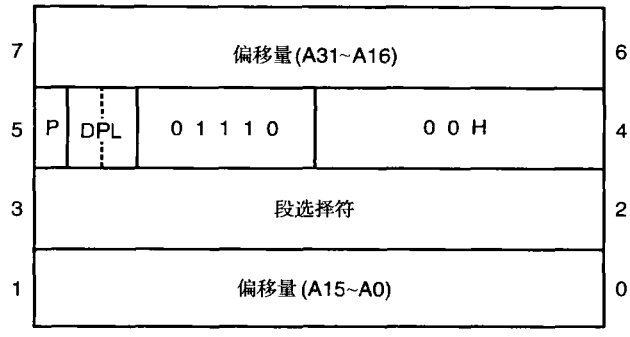


图 12-3 保护模式中断描述符

除了 IDT 和中断描述符外，保护模式中断功能与实模式中断相似，都是通过使用 IRET 或 IRETD 指令从中断返回。惟一区别在于，在保护模式下微处理器访问 IDT 而不是中断向量表。在 Pentium 4 和 Core2 的 64 位模式下，必须用 IRETQ 从中断返回。这就是 64 位模式下有不同的驱动程序和操作系统的一个原因。

12.1.6 中断标志位

中断标志（IF）和陷阱标志（TF）均在中断期间的标志寄存器内容压入堆栈后被清除。图 12-4 描述了标志寄存器的内容及 IF 和 TF 的位置。当 IF 被置位，它允许 INTR 引脚产生一个中断；当 IF 被清除，它阻止 INTR 引脚产生中断。当 TF = 1，它在每条指令执行之后产生一个陷阱中断（类型 1），这就是为什么我们常称陷阱为单步的原因；当 TF = 0，程序正常执行。该标志位用于调试，具体将在第 17 章至第 19 章中详述 80386 ～ Core2 时再做介绍。

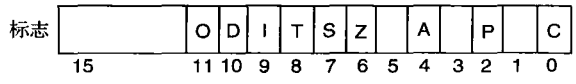


图 12-4 标志寄存器

中断标志分别由 STI 和 CLI 指令来置位和清除。没有特殊的指令来置位和清除陷阱标志。例 12-1 给出了一个中断服务程序，该程序通过置位程序内部堆栈的陷阱标志位来打开单步跟踪。例 12-2 给出了另一个中断服务程序，这个程序通过清除程序内部堆栈的陷阱标志位来关闭单步跟踪。

例 12-1

```
;该过程置位 TRAP 标志位以允许跟踪

TRON  PROC   FAR USES AX BP

      MOV  BP,SP              ;取 SP
      MOV  AX[BP+8]           ;从栈中获取标志
```

```
        OR    AH,1              ;设置跟踪标志
        MOV   [BP+8].AX
        IRET

TRON    ENDP
```

例 12-2

```
;该过程清除 TRAP 标志位以禁止跟踪

TROFF PROC    FAR USES AX BP

        MOV   BP,SP             ;取 SP
        MOV   AX,[BP+8]         ;从栈中获得标志
        AND   AH,0FEH           ;清除跟踪标志
        MOV   [BP+8],AX
        IRET

TROFF ENDP
```

在这两个例子中，使用 BP 寄存器从堆栈中取出标志寄存器，默认情况下 BP 寻址堆栈段。找到标志后，在从中断服务程序返回前将 TF 位置位（TRON）或者清除（TROFF）。IRET 指令以新的陷阱标志状态恢复标志寄存器。

跟踪过程

假定 INT 40H 指令访问 TRON，INT 41H 指令访问 TROFF，例 12-3 跟踪一个紧跟在 INT 40H 指令后的程序。例 12-3 中给出的中断服务程序对应于中断类型 1 或陷阱中断。每次跟着 INT 40H 执行一条指令后发生一个陷阱中断——TRACE 过程把所有 32 位微处理器寄存器的内容保存在被称为 REGS 的数组中。如果在数组中保存了寄存器的内容，这为在 INT 40H（TRON）和 INT 41H（TROFF）之间的所有指令提供了寄存器跟踪。

例 12-3

```
REGS    DD    8 DUP(?)          ;寄存器空间

TRACE   PROC  FAR USES EBX

        MOV   EBX,OFFSET REGS
        MOV   [EBX],EAX         ;保存 EAX
        POP   EAX
        PUSH  EAX
        MOV   [EBX+4],EBX       ;保存 EBX
        MOV   [EBX+8],ECX       ;保存 ECX
        MOV   [EBX+12],EDX      ;保存 EDX
        MOV   [EBX+16],ESP      ;保存 ESP
        MOV   [EBX+20],EBP      ;保存 EBP
        MOV   [EBX+24],ESI      ;保存 ESI
        MOV   [EBX+28],EDI      ;保存 EDI
        IRET

TRACE   ENDP
```

12.1.7　将一个中断向量存入向量表

为装入一个中断向量，有时被称为中断**钩链**（**hook**），汇编程序必须寻址绝对存储器。例 12-4 说明了一个新向量是如何通过使用汇编程序和 DOS 功能调用被加到中断向量表中的。这里，中断向量 INT 40H（中断过程 NEW40），被安装到存储器实模式向量位置 100H～103H。此过程首先要做的是保存旧的中断向量以防止卸载这个向量。如果无需卸载中断，则可以跳过此过程。

INT 21H 的函数 AX = 3100H 是 DOS 的存取函数，它将 NEW 40 过程装入内存中，在没有关机之前一直有效。DX 中的数是按段（16 字节）计算的程序长度。DOS 函数的详细资料参见附录 A。

需要注意的是，INT40 中断在 ENDP 之前有一条 IRET 指令，这是必须的，因为汇编编译器无法判

定 FAR 程序是否是一个中断程序。正常的 FAR 程序无需返回指令，但是中断程序确实需要一个 IRET 指令返回。中断必须定义为 FAR 程序。

例 12-4

```
.MODEL TINY
.CODE
.STARTUP
        JMP     START
OLD     DD      ?       ;旧向量的空间

NEW40   PROC    FAR     ; 必须是 FAR

;
;Interrupt software for INT 40H
;
        IRET            ; 必须有一个 IRET
NEW40   ENDP

;start installation

START:
        MOV   AX,0                      ;寻址0000H段
        MOV   DS,AX
        MOV   AX,DS:[100H]              ;得到INT 40H的偏移地址
        MOV   WORD PTR CS:OLD,AX        ;保存
        MOV   AX,DS:[102H]              ;得到INT40H的段地址
        MOV   WORD PTR CS:OLD+2,AX      ;保存
        MOV   DS:[100H],OFFSET NEW40    ;保存偏移量
        MOV   DS:[102H],CS              ;保存段地址

        MOV   DX,OFFSET START
        SHR   DX,4
        INC   DX
        MOV   AX,3100H                  ;使NEW40驻留
        INT   21H

END
```

12.2 硬件中断

微处理器有 2 个硬件中断输入：**非屏蔽中断**（**non-maskable interrupt**，**NMI**）和**中断请求**（**interrupt request**，**INTR**）。一旦激活 NMI 输入，就发生类型 2 中断，因为 NMI 是内部译码的。INTR 输入必须外部译码，以选择一个向量，INTR 引脚可选择任何中断向量，但通常只使用 20H ～ FFH 之间的中断类型号。Intel 保留 00H ～ 1FH 之间的中断用作内部和将来扩展。$\overline{\text{INTA}}$信号也是微处理器上的一个中断引脚，但它是用于响应 **INTR** 输入的一个输出引脚，它将向量类型号加载到数据总线 $D_7 \sim D_0$ 上。图 12-5 给出了微处理器上的 3 个用户中断引脚。

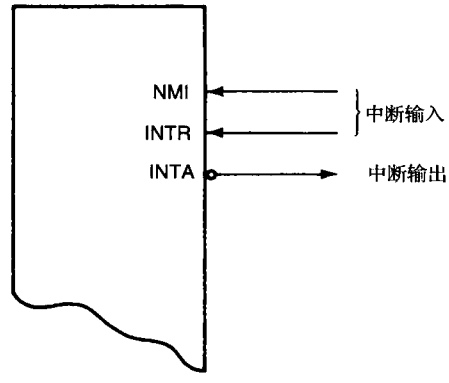


图 12-5 所有型号 Intel 微处理器上的中断引脚

NMI 是边沿触发输入，在上升沿（0 到 1 跳变）申请中断。在上升沿之后，NMI 引脚必须保持逻辑 1 直到微处理器识别它。注意，在上升沿被识别之前，NMI 引脚必须保持逻辑 0 至少 2 个时钟周期。

NMI 输入常用于奇偶校验错误和其他主要系统故障，如掉电。掉电通过监视 AC 电源线很容易被检测到，一旦 AC 电源掉电，则产生一个 NMI 中断。响应这种类型的中断时，微处理器将所有内部寄存器存于使用电池的备份存储器或 EEPROM 中。图 12-6 给出了一个掉电检测电路，一旦 AC 电源被中断，则它给 NMI 输入提供逻辑 1。

在此电路中，一个光隔离器提供与 AC 电源线的隔离。隔离器的输出由施密特触发器反相器整形，该反相器给 74LS122 可重触发单稳多谐振荡器的触发器输入端提供一个 60Hz 的脉冲。选择 R 和 C 的值使 74LS122 的有效脉冲宽度为 33ms 或 2 个 AC 输入周期。由于 74LS122 是可重触发的，所以只要 AC 电源一加上，Q 输出被触发总维持在逻辑 1，而 $\overline{Q}$ 维持在逻辑 0。

如果 AC 电源出现故障，74LS122 不再从 74ALS14 接收触发脉冲，这意味着 Q 回到逻辑 0 而 $\overline{Q}$ 回

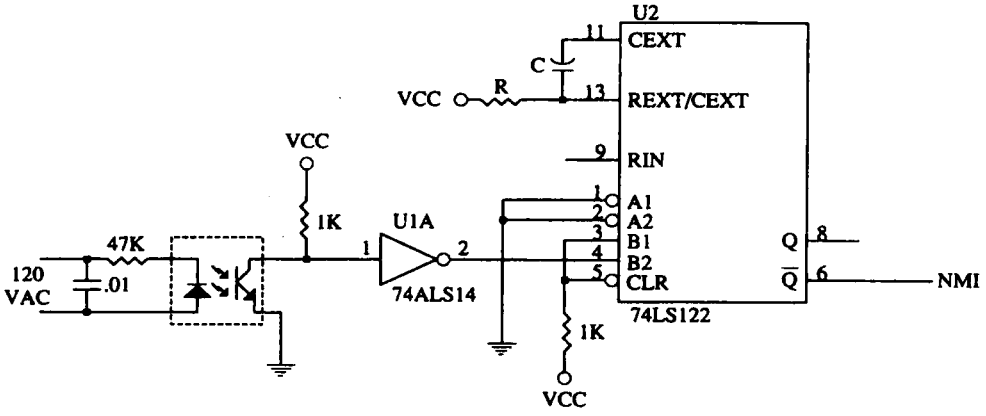


图 12-6　掉电检测电路

到逻辑 1，从而通过 NMI 引脚中断微处理器。中断服务程序（这里未给出）将所有内部寄存器的内容和其他数据存入电池备份存储器中。本系统假定系统电源有一个足够大的滤波电容器，在 AC 电源掉电后至少可提供电能 75ms 的时间。

图 12-7 给出了一个电路，它在 DC 电源出故障后给存储器提供电源。这里，二极管用于将电源电压从 DC 电源切换到电池。由于存储器电路的电源被提升为 +5.0V 到 +5.7V，所以使用的二极管为标准硅二极管。通过电阻来对电池涓流充电，电池可以是镍镉、锂或凝胶体电池。

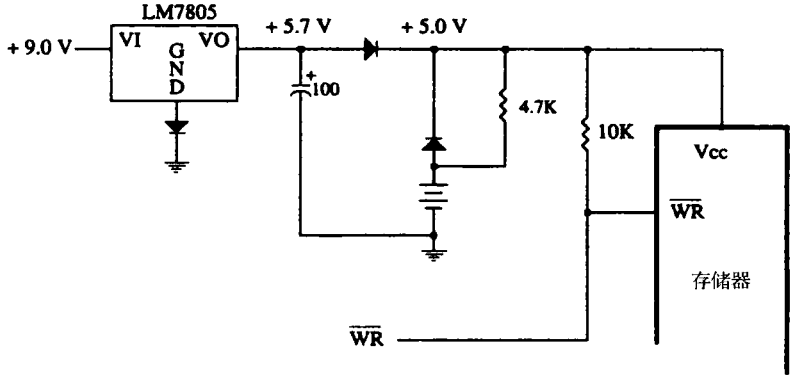


图 12-7　使用镍镉、锂或凝胶体电池的电池备份存储系统

当 DC 电源出故障时，电池给存储器件的 V_{CC} 引脚提供一个降低的电压。大多数存储器件在 V_{CC} 电压低到 1.5V 时仍能保持数据，所以电池电压无须为 +5.0V。$\overline{WR}$ 引脚在电源停电期间被拉升到 V_{CC}，所以没有数据会写入存储器。

12.2.1　INTR 和 $\overline{INTA}$

中断请求输入（INTR）是电平敏感的，这意味着它必须保持逻辑 1 电平直到被识别为止。INTR 引脚由外部事件置位，并在中断服务程序内部被清除。该输入一旦被微处理器接收到则自动被禁止，

并由中断服务程序末尾的 IRET 指令再次使能。80386 ～ Core2 在保护模式操作下使用 IRETD 指令。在 64 位模式中 IRETQ 在保护模式操作下使用。

若微处理器要在数据总线 D_7 ～ D_0 上接收一个中断向量类型号，则通过给$\overline{\text{INTA}}$输出加脉冲来响应 INTR 输入。图 12-8 给出了微处理器的 INTR 和$\overline{\text{INTA}}$引脚的时序图。系统产生了 2 个$\overline{\text{INTA}}$脉冲，用于在数据总线上插入中断向量类型号。

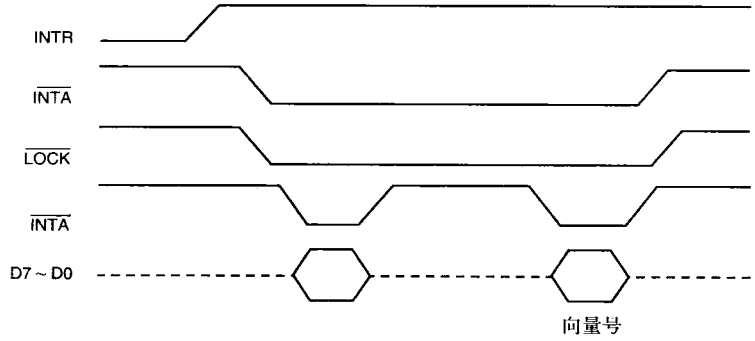


图 12-8 INTR 输入和$\overline{\text{INTA}}$输出的时序图

注：不理会这部分数据总线，通常会有向量号。

图 12-9 给出了一个简单电路，它将中断向量类型号 FFH 加到数据总线上以响应 INTR。注意，在此电路中没有连接$\overline{\text{INTA}}$引脚。由于使用电阻将数据总线 D_7 ～ D_0 拉高，所以微处理器自动用向量类型号 FFH 响应 INTR 输入。这可能是响应微处理器上 INTR 引脚的一种最经济的方式。

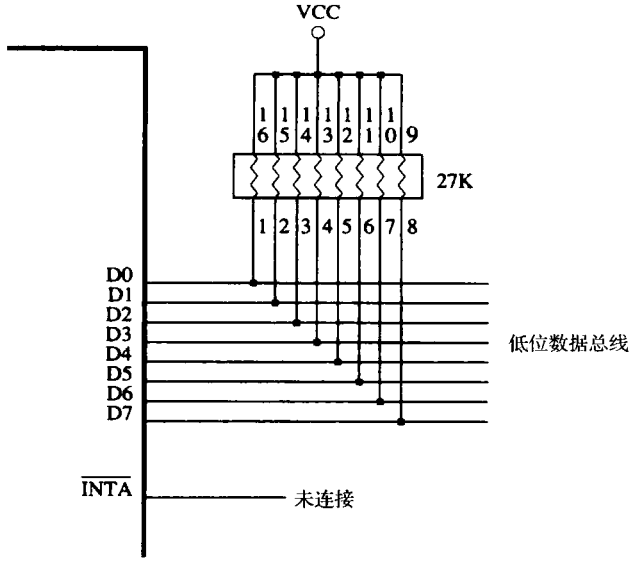


图 12-9 产生中断向量类型号 FFH 以响应 INTR 的简单方法

$\overline{\text{INTA}}$使用三态缓冲器

图 12-10 显示了中断向量类型号 80H 是如何加到数据总线 D_0 ～ D_7 上以响应 INTR 的。为响应 INTR，微处理器输出$\overline{\text{INTA}}$用于允许一个 74ALS244 三态 8 位缓冲器。8 位缓冲器将中断向量类型号加到数据总线上以响应$\overline{\text{INTA}}$脉冲。通过本图所示的 DIP 开关很容易改变中断向量类型号。

使 INTR 输入为边沿触发

我们常常需要一个边沿触发输入而不是电平敏感输入。使用 D 型触发器可将 INTR 输入转换为边沿

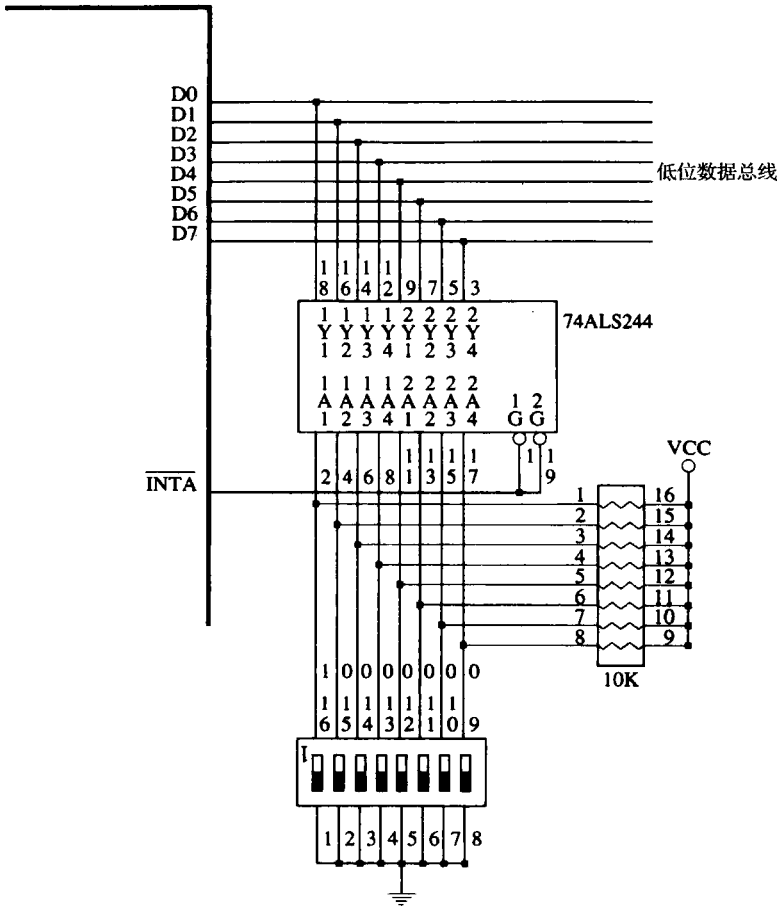


图 12-10　应用任一中断向量类型号响应$\overline{\text{INTA}}$的电路，本电路使用类型号 80H

触发输入，如图 12-11 所示。这里，时钟输入变成了边沿触发的中断请求输入，清除输入用于在微处理

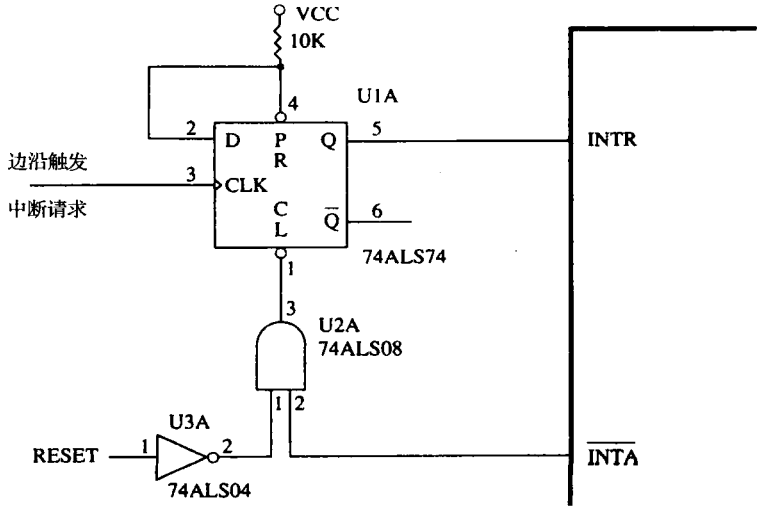


图 12-11　将 INTR 转换为边沿触发中断请求输入

器输出$\overline{INTA}$信号时清除请求。RESET 信号一开始就清除触发器，使得在系统刚上电时不申请中断。

12.2.2 82C55 键盘中断

第 11 章给出了 INTR 输入操作和中断的一个简单键盘实例。图 12-12 给出了 82C55 与微处理器和键盘的互连，它还说明了 74ALS244 8 位缓冲器在$\overline{INTA}$脉冲期间，是如何给微处理器提供中断向量类型号 40H 以响应键盘中断的。

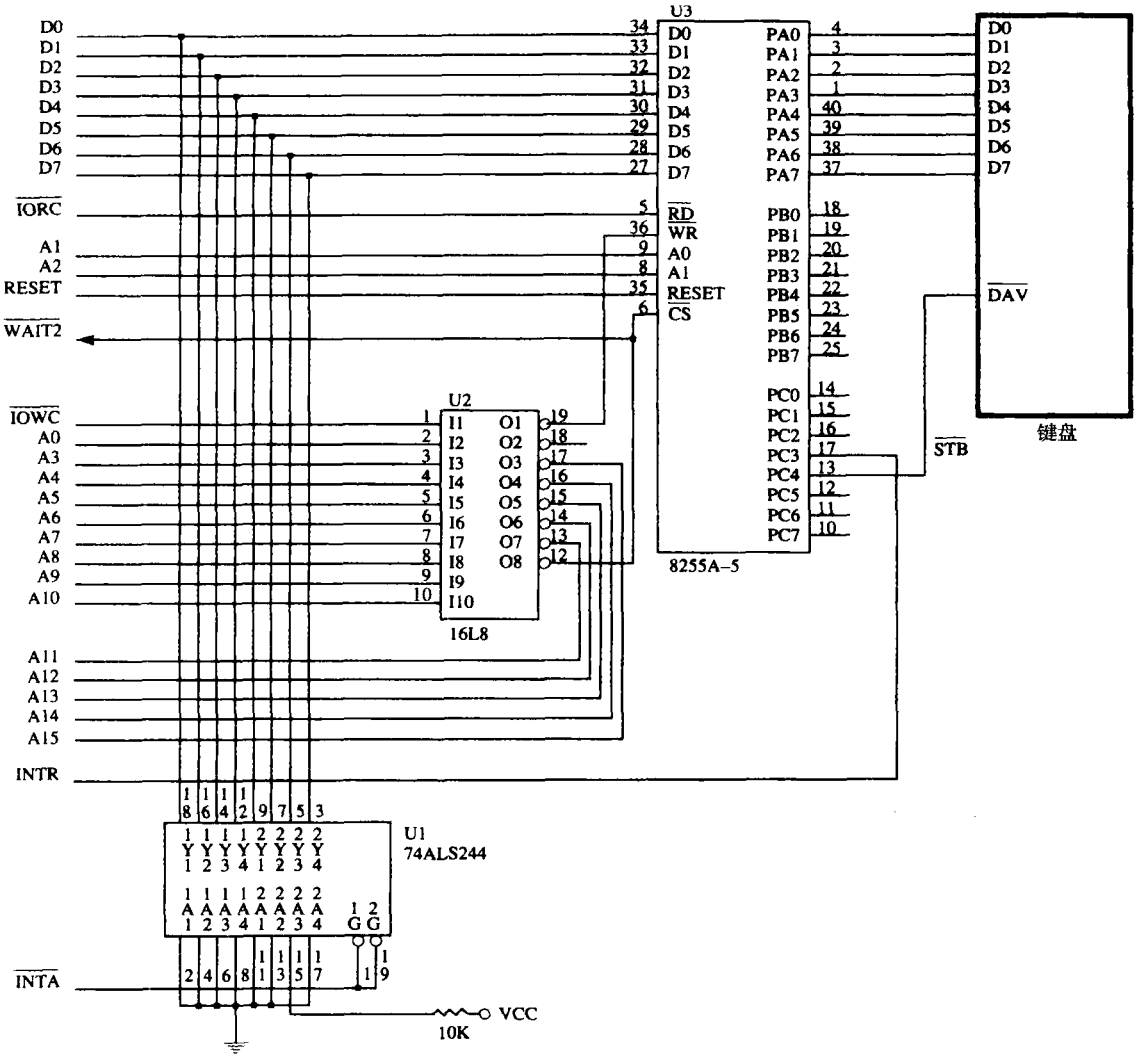


图 12-12 82C55 使用中断向量 40H 与微处理器的键盘互连

82C55 由 PLD（程序未给出）译码，对于 80386SX，其 I/O 端口地址为 0500H、0502H、0504H 和 0506H。82C55 工作在方式 1（选通输入方式）下，所以一旦键入一个键，INTR 输出（PC3）就变为逻辑 1，并通过微处理器上的 INTR 引脚申请一次中断。INTR 引脚维持高电平直到从端口 A 读出 ASCII 数据。换句话说，每次键入一个键，82C55 就通过 INTR 引脚申请一个类型 40H 的中断。来自键盘的$\overline{DAV}$信号使数据被锁存到端口 A，并使 INTR 变为逻辑 1。

例 12-5 给出了键盘的中断服务程序。在使用所有被中断影响的寄存器之前保存它们是非常重要的。在初始化 82C55 的软件中（这里未给出），初始化 FIFO 使两个指针相等，通过 82C55 内部的 INTE

位允许 INTR 请求引脚，并编程操作方式。

例 12-5

```
;从图12-12中描述的键盘读取键值的中断服务程序

PORTA   EQU    500H
CNTR    EQU    506H

FIFO    DB     256 DUP(?)              ;队列

INP     DD     FIFO                    ;输入指针
OUTP    DD     FIFO                    ;输出指针

KEY     PROC   FAR USES EAX EBX EDX EDI

        MOV    EBX,CS:INP              ;获取指针
        MOV    EDI,CS:OUTP

        INC    BL
        .IF BX == DI                   ;如果队列是满的
               MOV    AL,8
               MOV    DX,CNTR
               OUT    DX,AL           ;禁止82C55 中断
        .ELSE                          ;如果队列不满
               DEC    BL
               MOV    DX,PORTA
               IN     AL,DX           ;读取键码
               MOV    CS:[BX]         ;把数据存到队列中
               INC    BYTE PTR CS:INP
        .ENDIF
        IRET
KEY     ENDP
```

该程序很短，因为 80386SX 在调用该程序时已知道键盘数据是可用的。数据从键盘输入，然后存储在 FIFO（先进先出）缓冲器中。大多数键盘接口包含一个 FIFO，它至少有 16 字节长。本例中 FIFO 是 256 字节的，对于一个键盘接口来说足够了。注意 INC BYTE PTR INP 是如何用来给输入指针加 1 的，还应注意它总是寻址队列中的数据。

该程序首先检查 FIFO 是否满。当输入指针（INP）在输出指针（OUTP）下面一个字节时指示 FIFO 满。如果 FIFO 为满，则用 82C55 的置位/复位命令禁止中断，然后从中断返回。如果 FIFO 未满，则数据从端口 A 输入，且输入指针在返回之前加 1。

例 12-6 给出了从 FIFO 移出数据的过程。该过程首先通过比较 2 个指针以确定 FIFO 是否为空。如果指针相等，则 FIFO 为空，软件在 EMPTY 循环处等待，不断地测试指针。EMPTY 循环由键盘中断来中断，键盘中断将数据存入 FIFO 使其不再为空。该过程在寄存器 AH 中返回字符。

例 12-6

```
;该过程从例12-5的队列中读数据，并将它放在AH中返回
READQ   PROC   FAR USES EBX EDI EDX

        .REPEAT
               MOV     EBX,CS:INP          ;装载输入指针
               NOV     EDI,CS:OUTP
        .UNTIL EBX == EDI                  ;为空时

        MOV    AH,CS:[EDI]                 ;得到数据
        MOV    AL,9
        MOV    DX,CNTR                     ;允许82C55中断
        OUT    DX,AL
        INC    BYTE PTR CS:OUTP
        RET

READQ   ENDP
```

12.3 扩展中断结构

本书介绍扩展微处理器中断结构的3种最常见的方法。这一节我们将解释怎样用软件以及对图12-10中电路做一些硬件修改，就有可能扩展INTR输入，使其能接收7个中断输入。我们还将解释如何通过软件查询构成**菊花链**（**daisy-chain**）中断。下一节将介绍第3种技术，通过8259A可编程中断控制器将中断输入增加到63个。

12.3.1 使用74ALS244扩展

把图12-10修改为图12-13的电路，就可提供最多7个额外的中断输入。惟一的硬件改变是增加了一个8输入与非门，它在任一$\overline{IR}$输入变为有效时给微处理器提供INTR信号。

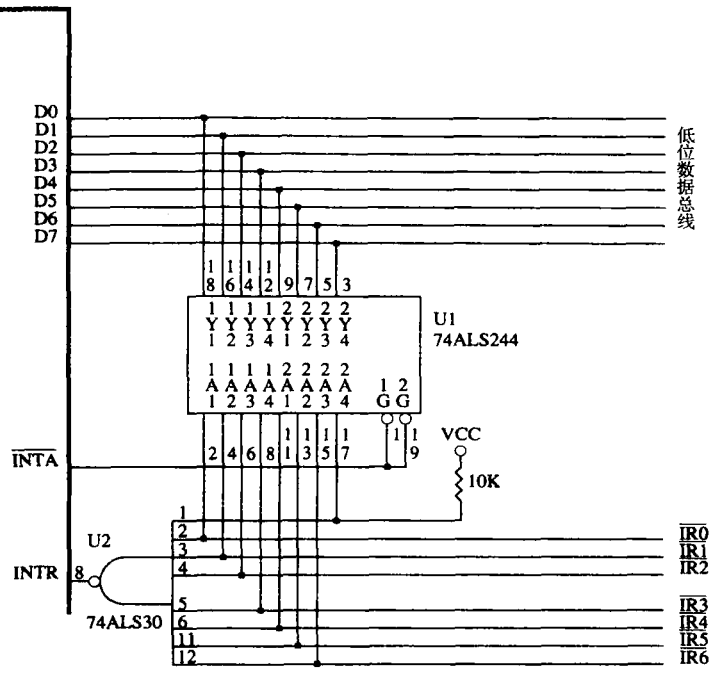


图12-13 将INTR输入从1条中断请求线扩展到7条

操作

如果任一$\overline{IR}$输入变为逻辑0，那么与非门的输出变为逻辑1，并通过INTR输入申请一次中断。在$\overline{INTA}$脉冲期间取得哪一个中断向量取决于哪一条中断申请线变为有效。表12-1给出了单个中断请求输入使用的中断向量。

表12-1 图12-13的单个中断请求

$\overline{IR6}$	$\overline{IR5}$	$\overline{IR4}$	$\overline{IR3}$	$\overline{IR2}$	$\overline{IR1}$	$\overline{IR0}$	中断向量
1	1	1	1	1	1	0	FEH
1	1	1	1	1	0	1	FDH
1	1	1	1	0	1	1	FBH
1	1	1	0	1	1	1	F7H
1	1	0	1	1	1	1	EFH
1	0	1	1	1	1	1	DFH
0	1	1	1	1	1	1	BFH

注：尽管没有说明，但IR输入均为低有效。

如果两个或多个中断请求输入同时有效，则产生一个新的中断向量。例如，若$\overline{IR1}$和$\overline{IR0}$均有效，则产生的中断向量为 FCH（252）。此存储单元要解决优先权问题。如果 IR0 输入有较高的优先权，则 IR0 的向量地址存储在向量单元 FCH 中。向量表的整个上半部分及其 128 个中断向量必须用来提供这 7 个中断请求输入的所有可能情况。这看起来似乎很浪费，但在许多专门应用中它是一种低成本的中断扩展方法。

12.3.2　菊花链中断

菊花链中断扩展在许多方面优于使用 74ALS244 的中断扩展，这是因为它只需要一个中断向量。确定优先权的任务留给了中断服务程序。为菊花链设置优先权确实需要额外的软件执行时间，但一般来说这是扩展微处理器中断结构的更好方法。

图 12-14 给出了一组 2 个 82C55 外围设备接口，它有 4 个 INTR 输出菊花链，并与微处理器的单个 INTR 输入相连。如果任一中断输出变为逻辑 1，则微处理器的 INTR 输入也变为逻辑 1，从而产生一次中断。

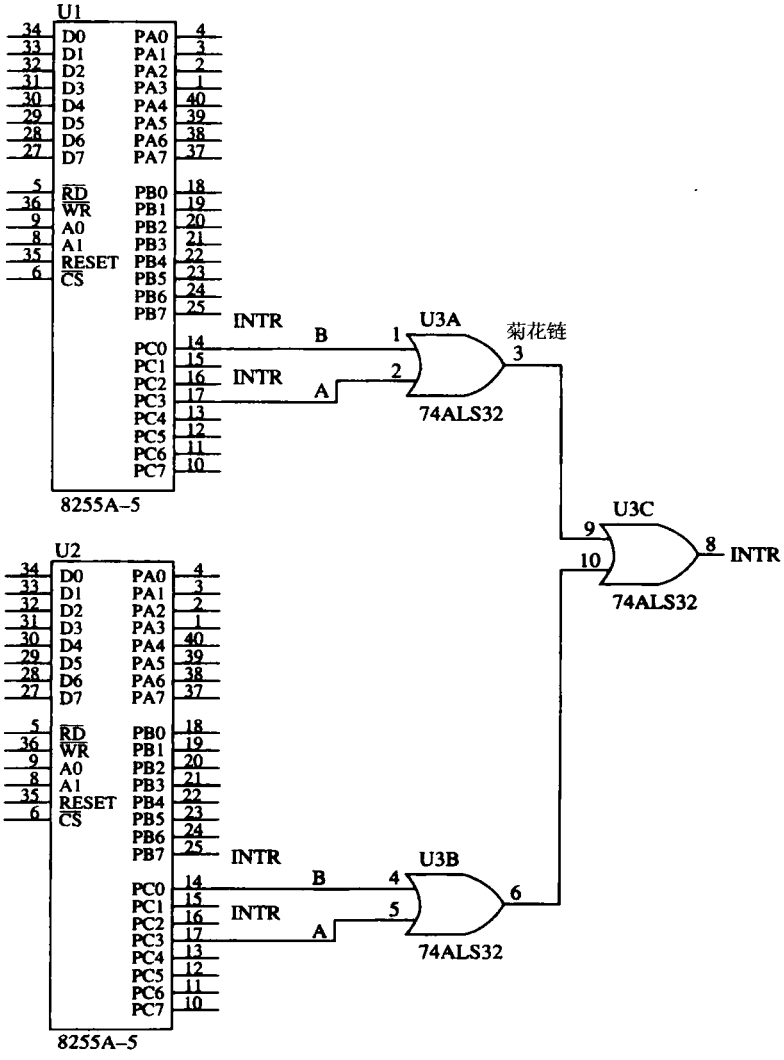


图 12-14　2 个 82C55 PIA 与 INTR 输出的连接为菊花链，它产生一个 INTR 信号

当菊花链用于申请中断时，最好使用上拉电阻将数据总线（$D_0 \sim D_7$）拉高，使中断向量 FFH 用于菊花链。任一中断向量都可用于响应菊花链。此电路中，来自 2 个 82C55 的 4 个 INTR 输出中的任意一个都将使微处理器的 INTR 引脚变为高，从而申请一次中断。

当具有菊花链的 INTR 引脚确实变为高时，硬件没有直接指示哪个 82C55 或哪个 INTR 输出产生中断。定位哪个 INTR 输出变为有效的任务交给中断服务程序完成，它必须查询 82C55 以确定哪个输出引起了中断。

例 12-7 给出了响应菊花链中断请求的中断服务程序。该程序查询每个 82C55 和每个 INTR 输出以决定使用哪个中断服务程序。

例 12-7

```
;该过程执行图12-14的菊花链中断服务

C1      EQU   504H                  ;第一个 82C55
C2      EQU   604H                  ;第二个  82C55
MASK1   EQU   1                     ;INTRB
MASK2   EQU   8                     ;INTRA

POLL    PROC  FAR    USES EAX EDX

        MOV   DX,C1                 ;寻址第一个 82C55
        IN    AL,DX
        TEST  AL,MASK1              ;测试  INTRB
        .IF !ZERO?

              ;LEVEL 1 中断软件

        .ENDIF
        TEST   AL,MASK2             ;测试. INTRA

        .IF !ZERO?

              ;LEVEL 2 中断软件

        .ENDIF
        MOV   DX,C2                 ;寻址第二个 82C55
        TEST  AL,MASK1              ;test INTRB
        .IF !ZERO?

              ;LEVEL 3 中断软件

        .ENDIF

              ;LEVEL 4 中断软件

POLL    ENDP
```

12.4 8259A 可编程中断控制器

8259A 可编程中断控制器（PIC）给微处理器增加了 8 个向量优先权编码中断。该控制器无需增加硬件即可被扩展，最多可接收 64 个中断请求。这种扩展需要 1 个主 8259A 和 8 个从 8259A。Intel 及其他厂商的最新芯片集中仍然采用这样一对 8259A 控制器，其编程方式如下所述。

12.4.1 8259A 概述

图 12-15 给出了 8259A 的引脚图。8259A 很容易与微处理器连接，因为除了$\overline{CS}$引脚必须被译码，$\overline{WR}$引脚必须有一个 I/O 存储体写脉冲以外，其他所有引脚都是直接与微处理器连接的。8259A 的各个引脚描述如下：

$D_0 \sim D_7$　双向**数据引脚**通常与 80386SX 的高位或低位数据总线相连，或与 8088 的数据总线相连。如果使用 80486 或 Pentium ～ Pentium 4，那么它们与任一 8 位存储体相连。

$IR_0 \sim IR_7$　**中断请求输入**用于申请一次中断，与具有多个 8259A 系统的一个从 8259A 相连。

$\overline{WR}$　**写输入**与微处理器上的写选通信号（$\overline{IOWC}$）相连。

$\overline{RD}$　**读输入**与$\overline{IORC}$信号相连。

INT　**中断输出**引脚，主 8259A 的 INT 与微处理器的 INTR 引脚相连，从 8259A 的 INT 与主 8259A 的 IR 引脚相连。

$\overline{INTA}$　**中断响应**输入与系统的$\overline{INTA}$信号相连。在具有主从 8259A 的系统中，只有主 8259A 的$\overline{INTA}$信号被连接。

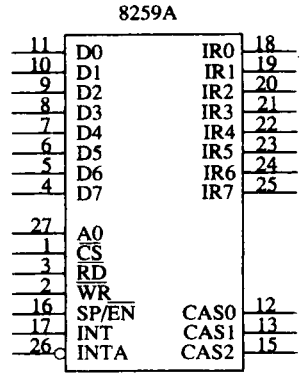


图 12-15　8259A 可编程中断控制器（PIC）的引脚图

A_0　**A_0 地址输入**选择 8259A 内部不同的命令字。

$\overline{CS}$　**片选**使能 8259A 进行编程与控制。

SP/$\overline{EN}$　**从片编程/允许缓冲器**是一个双功能引脚。当 8259A 工作于缓冲方式下时，它是一个输出引脚，控制一个基于微处理器的大系统中的数据总线收发器。当 8259A 工作于非缓冲方式下时，该引脚编程 8259A 为主片（SP/$\overline{EN}$ = 1）或从片（SP/$\overline{EN}$ = 0）。

$CAS_0 \sim CAS_2$　**级联线**用于从主 8259A 输出到从 8259A，从而级联系统中的多个 8259A。

12.4.2　连接单个 8259A

图 12-16 给出了单个 8259A 与 8086 微处理器的连接图。这里 SP/$\overline{EN}$引脚接高电平，表明它是主

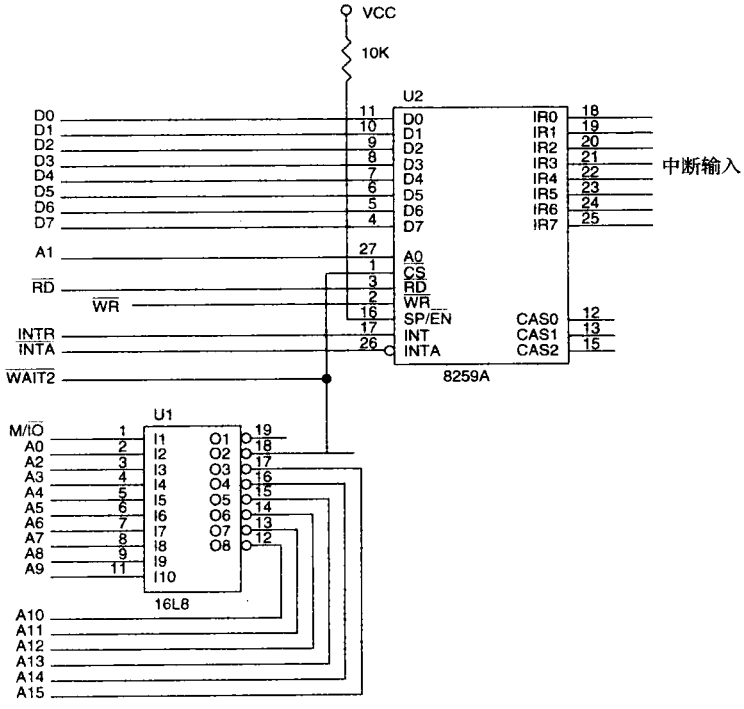


图 12-16　一个 8259A 与 8086 微处理器的连接

8259A。8259A 由 PLD 译码（程序未给出），其 I/O 端口地址为0400H 和0402H。与第 11 章讨论的其他外围设备一样，8259A 需要 4 个等待状态以与 16MHz 80386SX 一起正常工作，与 Intel 微处理器系列的一些其他型号连接时则需要更多的等待状态。

12.4.3 级联多个 8259A

图 12-17 给出了 2 个 8259A 与 80386SX 微处理器的连接，这种连接方式常见于 AT 型计算机，它有 2 个 8259A 用于中断。XT 或 PC 型计算机使用一个 8259A 控制器，中断向量为 08H ～ 0FH。AT 型计算机使用中断向量 0AH 作为来自第二个 8259A 的级联输入，第二个 8259A 位于向量 70H ～ 77H。附录 A 包含一张表，它列出了所有已用中断向量的功能。

此电路使用向量 08H ～ 0FH 以及 I/O 端口 0300H 和 0302H 用于 U1，即主 8259A；使用向量 70H ～ 77H以及 I/O 端口 0304H 和0306H 用于 U2，即从8259A。注意，其中还包括了数据总线缓冲器，以说明 8259A 上 SP/$\overline{EN}$引脚的用法。这些缓冲器只用在有许多器件连接到其数据总线上的大系统中，实际上很少使用这些缓冲器。

12.4.4 8259A 编程

8259A 由初始化命令字和操作命令字进行编程。**初始化命令字**（**initialization command word，ICW**）在 8259A 能够工作之前被编程，它规定了 8259A 的基本操作。**操作命令字**（**operation command word，OCW**）在正常操作过程中被编程，它控制 8259A 的操作。

初始化命令字

当 A_0 引脚为逻辑 1 时，8259A 有 4 个初始化命令字（ICW）可选择。当 8259A 刚上电时，它必须有 ICW_1、ICW_2 和 ICW_4 发送进来。如果 8259A 由 ICW_1 编程为级联方式，那么还必须编程 ICW_3。所以如果单个 8259A 用于系统中，则 ICW_1、ICW_2 和 ICW_4 必须被编程。如果级联方式用于系统中，那么所有 4 个 ICW 必须被编程。参见图 12-18 全部 4 个 ICW 格式。每个 ICW 描述如下：

ICW_1 编程 8259A 的基本操作。为编程此 ICW 用于 8086 ～ Pentium 4 操作，要将位 IC_4 置为逻辑 1。位 AD_1、A_7、A_6 和 A_5 对于微处理器操作为无关项，它们仅用于 8259A 与 8 位 8085 微处理器一起使用时（本书未讨论）。此 ICW 通过编程 SNGL 位选择单个或级联操作。如果选择了级联操作，则还必须编程 ICW_3。LTIM 位确定中断请求输入是上升沿触发还是电平触发。

ICW_2 选择用于中断请求输入的向量号。例如，如果编程 8259A 使之工作在向量 08H ～ 0FH，则将 08H 装入此命令字。同样，如果编程 8259A 使之工作在向量 70H ～ 77H，则将 70H 装入此 ICW。

ICW_3 仅用于 ICW_1 指示系统工作于级联方式下时。此 ICW 表明从 8259A 在何处与主 8259A 相连。例如，在图 12-18 中一个从 8259A 与 IR_2 相连。为编程 ICW_3 用于此连接，在主 8259A 和从 8259A 中，均将 04H 写入 ICW_3。假定有 2 个从 8259A 使用 IR_0 和 IR_1 与主 8259A 相连，则主 8259A 被编程为 ICW_3 = 03H，一个从 8259A 被编程为 ICW_3 = 01H，另一个从 8259A 被编程为 ICW_3 = 02H。

ICW_4 被编程用于 8086 ～ Pentium 4 微处理器，而在 8085 微处理器的系统中不被编程。最右边一位必须为逻辑 1，以选择与 8086 ～ Pentium 4 微处理器一起工作，其余位编程如下：

$\overline{SFNM}$ 如果此位被置为逻辑 1，则选择 8259A 的特殊完全嵌套操作方式。这样就允许主 8259A 正在处理来自从 8259A 的一个中断时，可以识别从 8259A 的另一更高优先权中断请求。一般方式下，一次只处理一个中断请求，其他中断请求均被忽略，直到完成这次中断处理。

$\overline{BUF}$和 $\overline{M/S}$ 缓冲器和主/从一起配合使用，以选择缓冲操作，或是 8259A 作为主片或从片时的非缓冲操作。

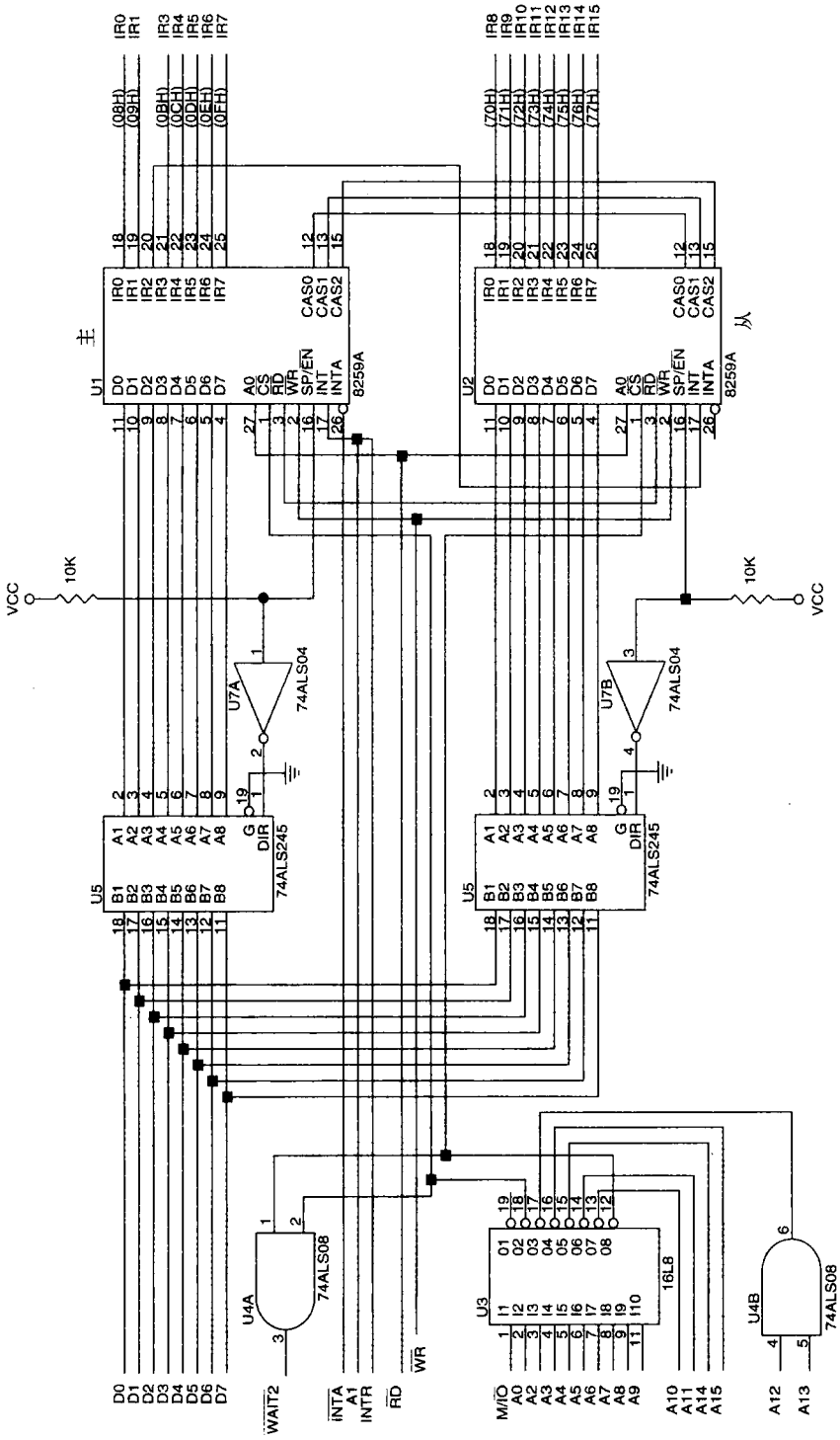


图 12-17　2个8259A与80386 SX微计理器的连接

注：8259AI/O端口为0300H和0302H,从8259AI/O端口为0304H和0306H。

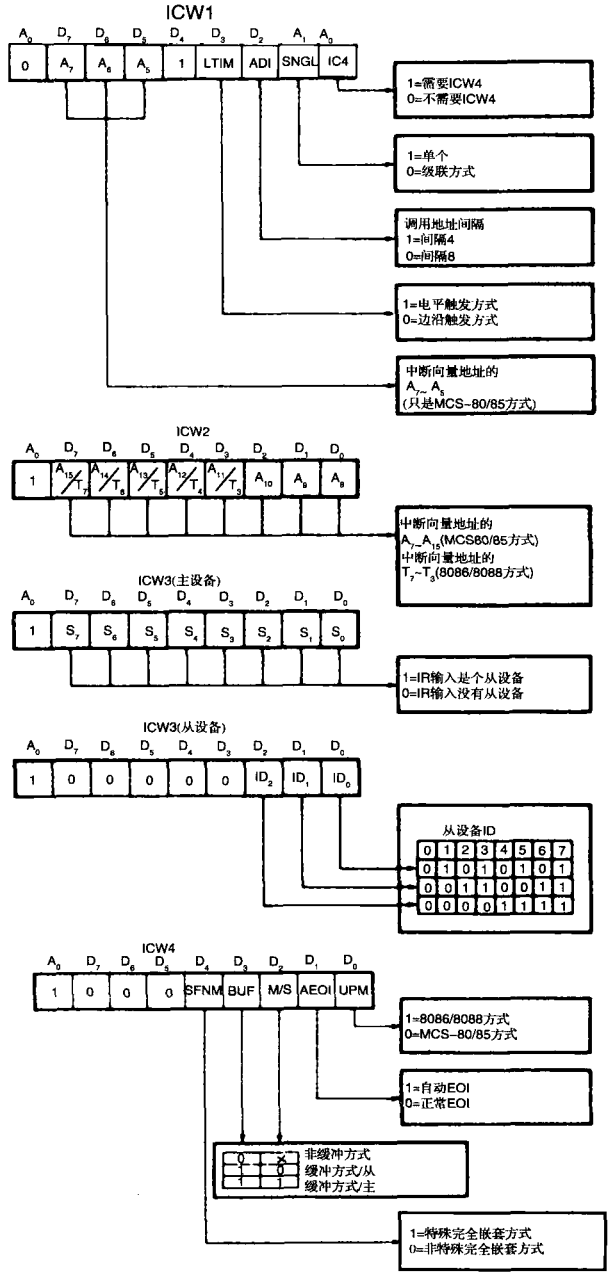


图 12-18 8259A 初始化命令字（ICW）

注：从片 ID 等于相应的主片 IR 输入。

AEOI 选择中断的自动结束或正常结束（在操作命令字中讨论得更完整）。OCW_2 的 EOI 命令只用在 ICW_4 未选择 AEOI 方式时。如果选择了 AEOI，则中断自动复位中断请求位且不修改优先权。这是 8259A 首选的操作方式，它减少了中断服务程序的长度。

操作命令字

一旦用 ICW 编程了 8259A，操作命令字（OCW）就用于控制 8259A 的操作。当 A_0 引脚为逻辑 0 时，除 OCW_1 外，其他 OCW 被选中；OCW_1 在 A_0 引脚为逻辑 1 时被选中。图 12-19 列出了 8259A 的所有 3 个操作命令字的二进制位模式。每个 OCW 的功能描述如下：

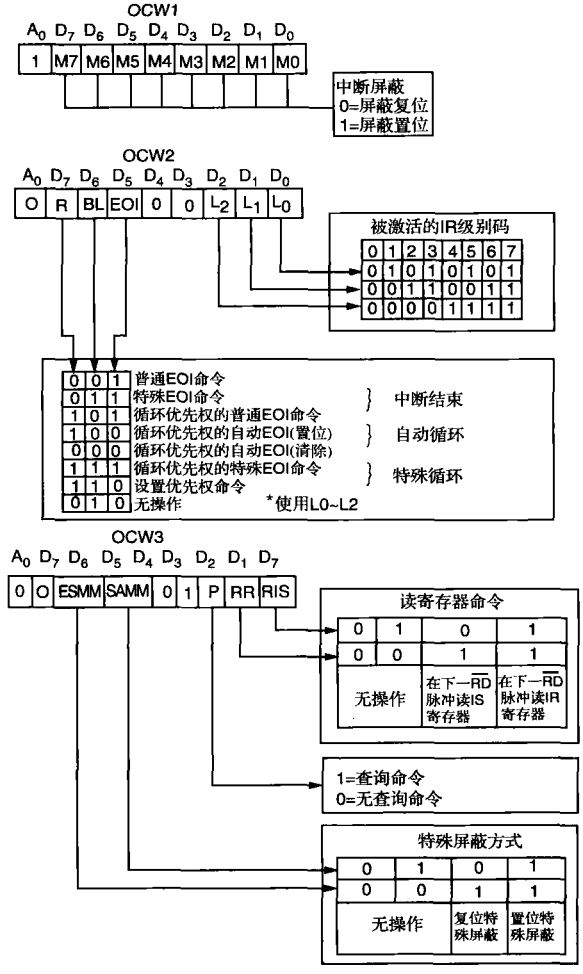


图 12-19　8259A 的操作命令字（OCW）

OCW_1　用于设置和读取中断屏蔽寄存器。当一个屏蔽位被置位时，它将关闭（屏蔽）相应的中断输入。当读 OCW_1 时屏蔽寄存器就被读出。由于在 8259A 刚初始化时屏蔽位的状态是未知的，所以在初始化编程 ICW 之后必须编程 OCW_1。

OCW_2　仅当 8259A 未选择 AEOI 方式时被编程。在这种情况下，此 OCW 选择 8259A 响应中断的方式。各方式列出如下：

普通中断结束命令——由中断服务程序发出的命令，标志中断的结束。8259A 自动确定哪一个中断电平有效并复位中断状态寄存器的正确位。复位状态位允许中断再次发生或较低优先权的中断生效。

特殊中断结束命令——使能一个特殊中断请求被复位的命令。其精确位置由 OCW_2 的 $L_2 \sim L_0$ 位确定。

循环优先权的普通 EOI 命令——功能与普通中断结束命令非常相似，只是它在复位中断状态

寄存器位之后循环中断优先权。由此命令复位的中断变为最低优先权中断。例如，如果 IR_4 刚刚被此命令服务过，它就变为最低优先权中断输入，而 IR_5 变为最高优先权。

循环优先权的自动 EOI 命令——此命令选择具有循环优先权的自动 EOI。只在需要此方式时才将此命令发送给 8259A。如果必须关闭此方式，则使用清除命令。

循环优先权的特殊 EOI 命令——功能与特殊 EOI 相同，除了它选择循环优先权外。

设置优先权命令——允许编程人员使用 $L_2 \sim L_0$ 位设置最低优先权中断输入。

OCW_3 选择要读的寄存器、特殊屏蔽寄存器的操作以及查询命令。如果选择查询，则 P 位必须被置位并输出给 8259A，下一次读操作将读出查询字。查询字的最右边 3 位指示具有最高优先权的有效中断请求；最左边一位指示是否有中断，必须检查这一位，以确定最右边 3 位是否包含有效信息。

状态寄存器

在 8259A 中有 3 个状态寄存器是可读的：**中断请求寄存器（interrupt request register，IRR）**、**服务寄存器（in-service register，ISR）**和**中断屏蔽寄存器（interrupt mask register，IMR）**（参见图 12-20，它对 3 个状态寄存器都可进行说明，因为它们均有相同的位结构）。IRR 是一个 8 位寄存器，它指示哪些中断请求输入有效。ISR 是一个 8 位寄存器，它包含正被服务的中断级。IMR 也是一个 8 位寄存器，它保持中断屏蔽位并指示哪些中断被屏蔽了。

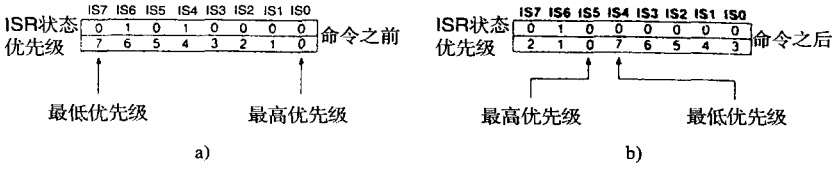


图 12-20 8259A 服务寄存器（ISR）

a）在 IR_4 被接收前 b）在 IR_4 被接收后

IRR 和 ISR 均通过编程 OCW_3 读出，而 IMR 通过 OCW_1 读出。为读 IMR，需使 $A_0 = 1$；为读 IRR 或 ISR，需使 $A_0 = 0$。OCW_3 的位 D_0 和 D_1 选择 $A_0 = 0$ 时哪个寄存器（IRR 或 ISR）被读出。

12.4.5 8259A 编程实例

图 12-21 给出了 8259A 可编程中断控制器与 16550 可编程通信控制器的连接。此电路中，来自 16550 的 INTR 引脚与可编程中断控制器的中断请求输入 IR_0 相连接。IR_0 发生在：（1）发送器准备发送另一字符；（2）接收器已接收到一个字符；（3）在接收数据时检测到一个错误；（4）发生调制解调器中断。注意，16550 被译码在 I/O 端口 40H～47H，8259A 被译码在 8 位 I/O 端口 48H 和 49H。两个器件均与 8088 微处理器的数据总线相连。

初始化软件

该系统的软件开始部分必须编程 16550 和 8259A，然后允许 8088 的 INTR 引脚，使中断可以生效。例 12-8 列出了编程这 2 个器件和使能 INTR 的软件。该软件使用了 2 个 FIFO 存储器为发送器和接收器保持数据。每个 FIFO 存储器 16KB 长并由一对指针（输入和输出）寻址。

例 12-8

```
;图12-21电路中16550和8259A的初始化软件

PIC1    EQU  48H              ;8259A 控制  A0 = 0
PIC2    EQU  49H              ;8259A 控制  A0 = 1
ICW1    EQU  1BH              ;8259A ICW1
ICW2    EQU  80H              ;8259A ICW2
ICW4    EQU  3                ;8259A ICW4
OCW1    EQU  0FEH             ;8259A OCW1
```

```
LINE      EQU  43H                    ;16650 线路寄存器
LSB       EQU  40H                    ;16650 波特率除数 LSB
MSB       EQU  41H                    ;16650 波特率除数 MSB
FIFO      EQU  42H                    ;16650 FIFO 寄存器
ITR       EQU  41H                    ;16650 中断寄存器

INIT PROC  NEAR

;
;setup 16650
;
        MOV  AL,10001010B             ;使能波特率除数
        OUT  LINE,AL

        MOV  AL,120                   ;编程波特率 9600
        OUT  LSB,AL
        MOV  AL,0

        OUT  MSB,AL

        MOV  AL,00001010B             ;7位数据、奇校验
        OUT  LINE,AL                  ;一个停止位

        MOV  AL,00000111B             ;允许发送和接收
        OUT  FIFO,AL                  ;
;
;program 8259A
;
        MOV  AL,ICW1                  ;编程 ICW1
        OUT  PIC1,AL
        MOV  AL,ICW2                  ;编程 ICW2
        OUT  PIC2,AL
        MOV  AL,ICW4                  ;编程 ICW4
        OUT  PIC2,AL
        MOV  AL,OCW1                  ;编程 OCW1
        OUT  PIC2,AL
        STI                           ;允许中断
;
;允许16650 中断
;
        MOV  AL,5
        OUT  ITR,AL                   ;允许中断
        RET

INIT    ENDP
```

该程序的开始部分（START）编程 16550 UART，使其具有 7 位数据位、奇校验、1 位停止位以及 9600 的波特率时钟。FIFO 控制寄存器还使能发送器和接收器。

该程序的第二部分用 3 个 ICW 和 1 个 OCW 编程 8259A。设置 8259A 工作在中断向量 80H ～ 87H 及自动 EOI 方式下。OCW 允许 16550 UART 的中断，还通过使用 STI 指令允许微处理器的 INTR 引脚。

软件的最后部分通过中断控制寄存器使能接收器中断和 16550 UART 的错误中断。发送器中断直到数据可以发送时才被使能。参见图 12-22 16550 UART 的中断控制寄存器的内容。注意，控制寄存器可使能或禁止接收器中断、发送器中断、线路状态（错误）中断及调制解调器中断。

处理 16550 UART 中断请求

由于 16550 对于各种中断只产生一个中断请求，所以中断管理程序必须查询 16550，确定发生了什么类型的中断。这是通过检查中断识别寄存器来实现的（见图 12-23）。注意，中断识别寄存器（只读）与 FIFO 控制寄存器（只写）共享同一 I/O 端口。

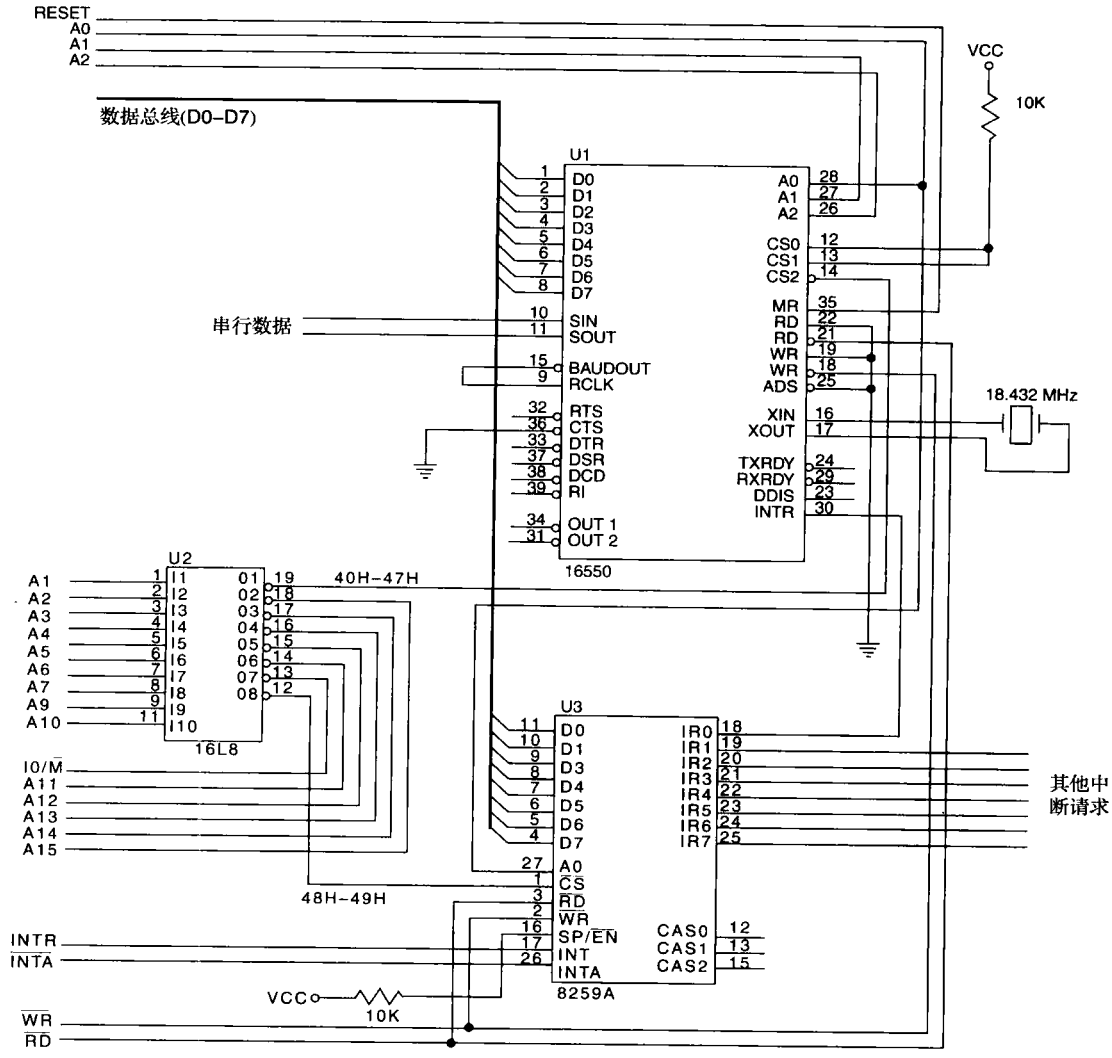


图 12-21 16550 UART 通过 8259A 与 8088 微处理器的连接

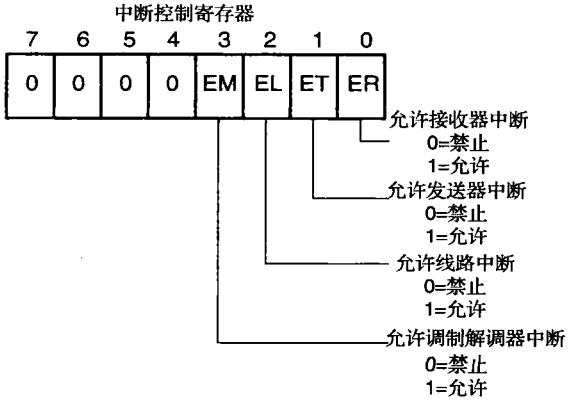


图 12-22 16550 中断控制寄存器

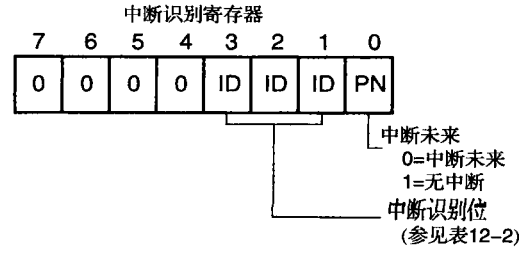


图 12-23 16550 中断识别寄存器

中断识别寄存器指示是否有中断、中断的类型以及发送器和接收器 FIFO 存储器是否被允许。参见表 12-2 中断控制位的内容。

表 12-2　16550 的中断控制位

位 3	位 2	位 1	位 0	优先级	类型	复位控制
0	0	0	1	—	没有中断	—
0	1	1	0	1	接收器错误（奇偶校验、帧、超限或中断）	通过读寄存器复位
0	1	0	0	2	接收器数据就绪	通过读数据复位
1	1	0	0	2	字符超时，至少在 4 个字符时间内未从接收器 FIFO 中移出字符	通过读数据复位
0	0	1	0	3	发送器空	通过写发送器复位
0	0	0	0	4	调制解调器状态	通过读 MODEM 状态复位

注：1 是最高优先级，而 4 是最低优先级。

中断服务程序必须检查中断识别寄存器的内容，以确定是什么事件引起了中断，并将控制转移到与该事件对应的过程。例 12-9 给出了中断管理程序的开始部分，它将控制转移给接收器数据中断的 RECV、发送器数据中断的 TRANS 以及线路状态错误中断的 ERR。注意，调制解调器状态在本例中未测试。

例 12-9

```
;图12-21中16550 UART的中断处理程序

INT80 PROC FAR USES AX BX DI SI

      IN    AL,42H                    ;读中断号
      .IF AL == 6

            ;处理接收错误

      .ELSEIF AL == 2
            ;处理发送器为空时的情况

            JMP     TRAN    ;例 12-13

        .ELSEIF AL == 4

            ;处理接收器准备好

            JMP     RECV    ;例 12-11

        .ENDIF

        IRET

INT80   ENDP
```

从 16550 接收数据需要 2 个过程。一个过程在每次 INTR 引脚申请中断时读 16550 的数据寄存器，并将其存储到 FIFO 存储器中。另一过程从主程序的 FIFO 存储器中读数据。

例 12-10 列出了从主程序的 FIFO 存储器中读数据的过程。该过程假定已在系统初始化时初始化了指针 IIN 和 IOUT（未给出）。READ 过程返回 AL，它包含从存储器 FIFO 读出的一个字符。如果存储器 FIFO 为空，则该过程返回设置为逻辑 1 的进位标志位。如果 AL 包含一个有效字符，则进位标志位在从 READ 返回时被清除。

一旦地址超出 FIFO 起始地址加上 16K 的和时，请注意如何从 FIFO 顶部到底部改变地址从而重新使用 FIFO。还应注意，万一有中断被 RECV 中断服务程序的 FIFO 存储器满条件禁止，则在本过程的末尾使能这些中断。

例 12-10

```
;该过程从 FIFO 读一个字符，并将它返回 AL
;如果 FIFO 为空，则用进位位（1）返回

READC   PROC    NEAR USES BX DX

        MOV   DI,IOUT                ;得到指针
        MOV   BX,IIN
        .IF   BX == DI               ;如果为空
          STC                        ;设置进位

        .ELSE                                ;如果不为空
          MOV  AL,ES:[DI]           ;取得数据
          INC  DI                   ;指针加1
          .IF DI == OFFSET FIFO+16*1024
                MOV  DI,OFFSET FIFO
          .ENDIF
        MOV IOUT,DI
        CLC
        .ENDIF
        PUSHF                                ;允许接收中断
        IN   AL,41H
        OR   AL,5
        OUT  41H,AL
        POPF
        RET

READC   ENDP
```

例 12-11 列出了 RECV 中断服务程序，微处理器在每次从 16550 接收一个字符时调用它。本例中，中断使用向量类型号 80H，它必须访问例 12-9 中的中断管理程序。每次发生此中断时，由中断管理程序从 16550 读出一个字符，然后访问 RECV 服务程序。RECV 服务程序将该字符存入 FIFO 存储器。如果 FIFO 存储器已满，则 16550 内部的中断控制寄存器禁止接收器中断。这也许会导致丢失数据，但至少不会引起中断，而超越已存于存储器 FIFO 中的有效数据。由 16550 检测到的任何错误情况都将“?”（3FH）存入存储器 FIFO 中。注意，错误由中断管理程序的 ERR 部分（未给出）检测。

例 12-11

```
;例 12-9 中断处理程序的 RECV 部分

RECV:
      MOV   BX,IOUT               ;取指针
      MOV   DI,IIN
      MOV   SI,DI
      INC   SI
      .IF   SI == OFFSET FIFO+16*1024
        MOV SI,OFFSET FIFO
      .ENDIF
      .IF SI == BX                ;如果 FIFO 为满
        IN    AL,41H              ;禁止接收
        AND   AL,0FAH
        OUT   41H,AL
       .ENDIF
       IN AL,40H                  ;读数据
       STOSB
       MOV  IIN,SI
       MOV  AL,20H                ;8259A EOI 命令
       OUT  49H,AL
       IRET
```

把数据发送给 16550

把数据发送给 16550 的方式与接收相同，除了中断服务程序是从第二个 16KB FIFO 存储器移出发送数据以外。

例 12-12 列出了填充输出 FIFO 的过程，它与例 12-10 中的过程类似，不同的是它确定 FIFO 是否为满，而不是测试是否为空。

例 12-12

```
;该过程把数据放在 FIFO 中，以便发送中断将它发送出去
;AL = 要发送的字符

SAVEC  PROC  NEAR USES BX DI SI

       MOV  SI,OIN                    ;装载指针
       MOV  BX,OOUT
       MOV  DI,SI
       INC  SI
       .IF SI == OFFSET OFIFO+16*1024
            MOV  SI,OFFSET OFIFO
       .ENDIF
       .IF BX == SI                   ;如果 OFIFO 为满
            STC
       .ELSE
            STOSB
            MOV  OIN,SI
            CLC
       .ENDIF
       PUSHF
       IN   AL,41H                    ;允许发送器
       OR   AL,1
       OUT  41H,AL
       RET

SAVEC  ENDP
```

例 12-13 列出了 16550 UART 发送器的中断服务子程序。该过程是例 12-9 中给出的中断管理程序的延续，与例 12-11 的 RECV 过程相似，不同的是它确定 FIFO 是否为空，而不是为满。注意，这里没有包括中止中断或任何错误处理的中断服务过程。

例 12-13

```
;16550 发送器的中断服务过程

TRAN:
      MOV BX,OIN                    ;装载指针
      MOV DI,OOUT
      .IF BX == DI                  ;如果为空
            IN   AL,41H
            AND  AL,0FDH            ;禁止发送
            OUT  41H,AL
      .ELSE                         ;如果不为空
            MOV  AL,ES:[DI]
            OUT  40H,AL             ;发送数据
            INC  DI
            .IF DI == OFFSET OFIFO+16*1024
                 MOV  DI,OFFSET OFIFO
            .ENDIF
            MOV  OFIFO,DI
      .ENDIF
      MOV AL,20H                    ;发送 EOI 给 8259A
      OUT 49H,AL
      IRET
```

16550 还包含一个临时寄存器，它是一个通用寄存器，必要时编程人员可将它用在任何方面。16550 内部还包含一个调制解调器控制寄存器和一个调制解调器状态寄存器。这些寄存器允许调制解调器产生中断，并控制带有一个调制解调器的 16550 的操作。参见图 12-24 调制解调器状态寄存器和控制寄存器的内容。

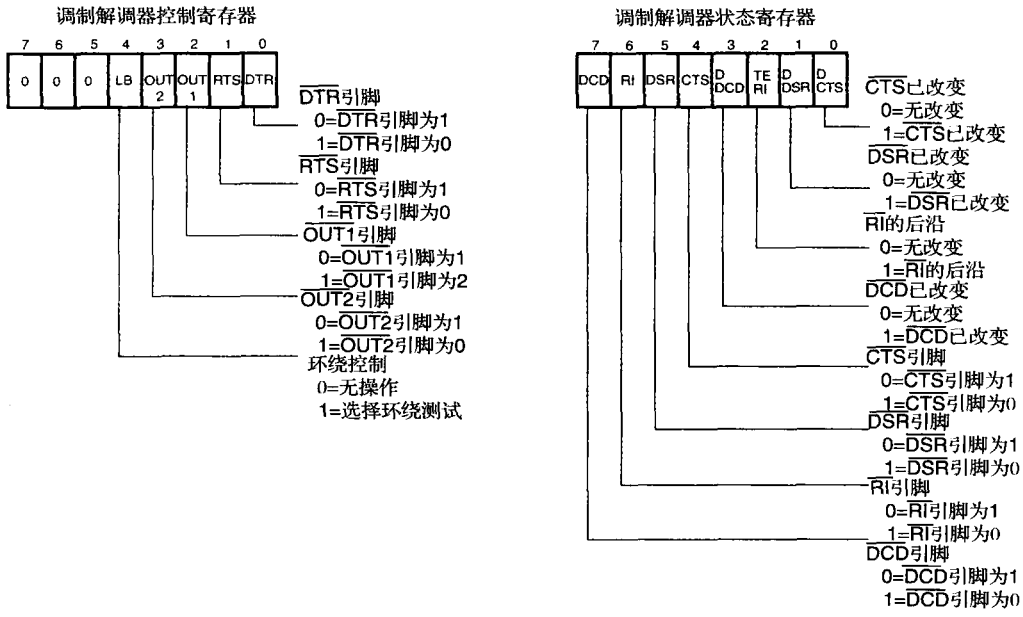


图 12-24 16550 调制解调器控制寄存器和状态寄存器

调制解调器控制寄存器使用位0～3控制16550上的不同引脚。位4使能内部环绕测试。调制解调器状态寄存器使能测试调制解调器引脚的状态；还允许检查调制解调器引脚是否有变化，比如RI的后沿。

图12-25给出了16550 UART与RS-232C接口的连接，RS-232C接口常用来控制调制解调器。在这个接口电路中包括线路驱动器和接收器电路，用来在16550上的TTL电平与接口的RS-232C电平之间进行转换。注意，RS-232C电平通常是逻辑0为 +12V，而逻辑1为 -12V。

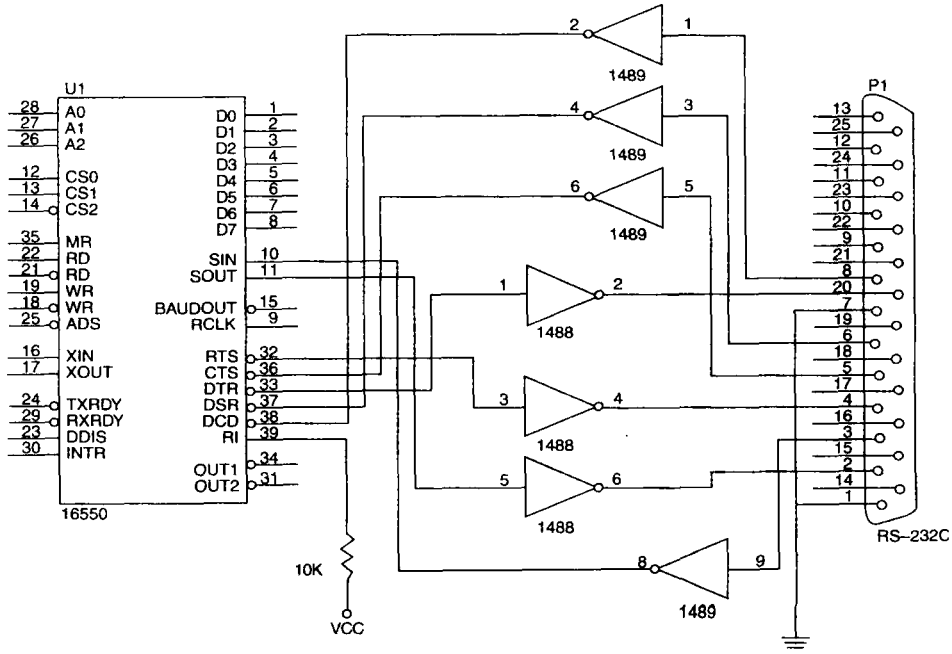


图 12-25 16550 使用1488线路驱动器和1489线路接收器与RS-232C连接

为通过调制解调器发送或接收数据，先激活$\overline{\text{DTR}}$引脚（逻辑0），然后UART等待调制解调器的$\overline{\text{DSR}}$引脚变为逻辑0，指示调制解调器已就绪。一旦这个信号交换完成，UART就给调制解调器的$\overline{\text{RTS}}$引脚发送逻辑0。当调制解调器已就绪，它就返回$\overline{\text{CTS}}$信号（逻辑0）给UART。现在就可以开始通信了。调制解调器的$\overline{\text{DCD}}$信号指示调制解调器已检测到一个载波，该信号还必须在通信开始之前被测试。

12.5　中断实例

本节给出一个实时时钟和一个中断处理键盘作为中断应用的实例。**实时时钟**（**real-time**，**RTC**）以真实时间计时，也就是说，以小时和分钟计时。本例以小时、分钟、秒和1/60秒计时，使用4个存储单元来保持一天的BCD时间。中断处理键盘使用周期性中断来扫描键盘的键。

12.5.1　实时时钟

图12-26给出了一个简单电路，它使用60Hz AC电源线为NMI中断输入引脚产生一个周期性中断请求信号。尽管使用来自AC电源线的信号在频率上不时地稍有变化，但在联邦交易委员会（Federal Trade Commission，FTC）限制的时间周期内是准确的。

该电路使用来自120V AC电源线的信号，在加到NMI中断输入引脚上之前由一个施密特触发器反相器进行了调节。注意，必须确保图中电源线的接地线与系统接地线相连接。电源中线（白色线）是电源线上的大扁平引脚，一边窄扁平引脚是火线（黑色线）；另一边窄扁平引脚是120V AC端。

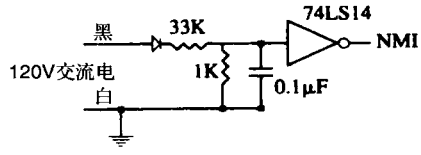


图12-26　将AC电源线转换为用于NMI输入的60Hz TTL信号

实时时钟软件包含一个每秒调用60次的中断服务程序，以及一个更新存于4个存储单元中的计数值的过程。例12-14列出了这2个程序，以及用于保持一天BCD时间的4字节存储器。TIME的存储单元在系统内存中的段地址为SEGMENT，偏移地址为TIME，在TIMEP过程中SEGMENT第一次被装入。模数或每个计数器的查找表（LOOK）与过程一起存储在代码段。

例12-14

```
TIME    DB      ?               ;1/60秒计数器（÷60）
        DB      ?               ;秒计数器（÷60）
        DB      ?               ;分计数器（÷60）
        DB      ?               ;小时计数器（÷24）

LOOK    DB      60H, 60H, 60H, 24H

TIMEP   PROC    FAR USES AX BX DS

        MOV   AX,SEGMENT                ;装载TIME的段地址
        MOV   DS,AX
        MOV   BX,0                      ;初始化指针

        .REPEAT                         ;启动时钟
              MOV   AL,DS:TIME[BX]
              ADD   AL,1                ;计数器加1
              DAA                       ;调整为BCD码
              .IF AL == BYTE PTR CS:LOOK[BX]
                    MOV AL,0
              .ENDIF
               MOV   DS:TIME[BX],AL
               INC   BX
        .UNTIL AL !=  0 || BX == 4
        IRET

TIMEP   ENDP
```

另一种处理时间的方法是使用一个单独的计数器将时间存储在内存中，然后用软件来决定真实的

时间。比如，可以用一个32位的计数器存储时间（一天有5 184 000个1/60秒），计数器的0可以表示12:00:00:00 AM，5 183 999可以表示11:59:59:59 PM。例12-15给出了这种实时时钟（RTC）的中断过程。这种实时时钟需要执行的时间最少。

例12-15

```
TIME    DD      ?                       ;模数为5184000的计数器

TIMEP   PROC    FAR USES EAX

        MOV   AX,SEGMENT
        MOV   DS,AX

        INC   DS:TIME
        .IF DS:TIME == 5184000
              MOV  DWORD PTR DS:TIME,0
        .ENDIF
        IRET

TIMEP   ENDP
```

软件将模数为5 184 000的计数器中的数字转化为小时、分钟和秒。例12-16将给出这个过程。BL返回小时（0～23），BH返回分钟，AL返回秒，但不返回1/60秒。

例12-16

```
;返回时间BL = 小时,BH = 分钟和AL = 秒

GETT   PROC    NEAR ECX EDX

        MOV   ECX,216000        ;除以216000
        MOV   EAX,TIME
        SUB   EDX,EDX           ;清EDX
        DIV   ECX               ;得到小时数
        MOV   BL,AL
        MOV   EAX,EDX
        MOV   ECX,3600          ;除以3600
        DIV   ECX               ;得到分钟数
        MOV   BH,AL
        SUB   EAX,EDX
        MOV   ECX,60            ;除以60
        DIV   ECX
        RET

GETT    ENDP
```

假设需要时间延迟，那么在例12-25中使用RTC可以得到1/60秒到24小时间的任何时间延迟。例12-17给出了利用实时时钟来实现用EAX寄存器传递延迟秒数的过程。这可以是给一天的时间增加1秒延时。延迟的精度是1/60秒，即RTC的分辨率。

例12-17

```
SEC    PROC     NEAR USES EAX EDX

       MOV    EDX,60
       MUL    EDX              ;得到按1/60秒计算的秒数
       ADD    EAX,TIME         ;TIME提前加在EAX中
       .IF    EAX >= 51840000
                SUB EAX,5184000
       .ENDIF
       .REPEAT                 ;等待TIME赶上
       .UNTIL  EAX == TIME
       RET

SEC    ENDP
```

12.5.2 中断处理键盘

中断处理键盘通过周期性中断来扫描键盘上的键。每次中断发生，中断服务程序都测试一个键或为此键去抖动。一旦检测到一个有效的键，中断服务程序就会将此键代码存入一个键盘队列，以供系统稍后读出。该系统的基础是一个周期性中断，它可由一个定时器或系统中其他器件产生。注意，大多数系统已有一个用于实时时钟的周期性中断。本例中，假定该中断每 10ms 调用一次中断服务程序；如果 RTC 的时钟频率为 60Hz，也可以每 16.7ms 调用一次。

图 12-27 给出了键盘与 82C55 的连接。它未给出每 10ms 或 16.7ms 调用一次中断所需的定时器或其他电路（编程 82C55 的软件也未给出）。必须对 82C55 进行编程，使端口 A 为输入端口，端口 B 为输出端口，初始化软件必须在端口 B 存储一个 00H。该接口使用代码段存储器存储键盘扫描过程之后的一个队列和几个字节。例 12-18 列出了键盘的中断服务程序。

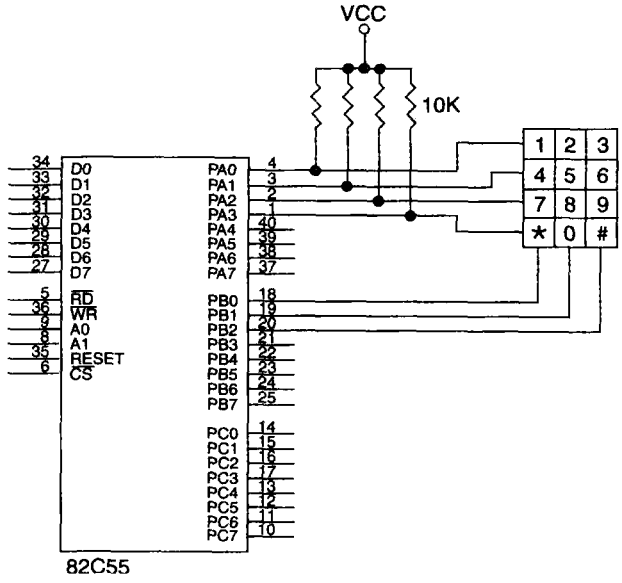


图 12-27　电话式键盘与 82C55 的连接

例 12-18

```
; 图 12-27 中键盘的中断服务程序

PORTA   EQU    1000H
PORTB   EQU    1001H

DBCNT   DB     0                        ;去抖动计数器
DBF     DB     0                        ;去抖动标志
PNTR    DW     QUEUE                    ;队列输入指针
OPNTR   DW     QUEUE                    ;队列输出指针
QUEUE   DB     16 DUP(?)                ;16 字节的队列

INTK    PROC   FAR USES AX BX DX

        MOV    DX,PORTA                 ;测试任意键
        IN     AL,DX
        OR     AL,0F0H
        .IF AL != 0FFH                  ;如果有键按下
              INC   DBCNT               ;抖动计数加 1
              .IF DBCNT == 3            ;如果键按下 >20ms
                    DEC   DBCNT
```

```
                .IF  DBF == 0
                     MOV  DBF,1
                     MOV  BX,00FEH
                     .WHILE 1        ;找到键
                            MOV  AL,BL
                            MOV  DX,PORTB
                            OUT  DX,AL
                            ROL  BL,1
                            MOV  DX,PORTA
                            IN   AL,DX
                            OR   AL,0F0H
                            .BREAK .IF AL != 0
                            ADD  BH,4
                     .ENDW
                     MOV  BL,AL
                     MOV  AL,0
                     MOV  DX,PORTB
                     OUT  DX,AL
                     DEC  BH
                     .REPEAT
                            SHR  BL,1
                            INC  BH
                     .UNTIL !CARRY?
                     MOV   AL,BH
                     MOV   BX,PNTR
                     MOV   [BX],AL          ;键码送到队列
                     INC     BX
                     .IF BX == OFFSET QUEUE+16
                            MOV  DX,OFFSET QUEUE
                     .ENDIF
                     MOV  PNTR,BX
                .ENDIF
          .ENDIF
     .ELSE                              ;若没有键按下
          DEC  DBCNT                    ;抖动计数器减1
          .IF SIGN?                     ;若低于0
               MOV  DBCNT,0
               MOV  DBF,0
          .ENDIF
     .ENDIF
     IRET

INTK ENDP
```

键盘中断发现键并将键代码存于队列中。存于队列中的代码是原始码，它不表示键号。例如，“1”键的键代码是00H，“4”键的键代码是01H等。此软件没有预防队列溢出，应加上相应的程序，但几乎在所有情况下，一次键入都难以超出16字节队列。

例12-19给出了从键盘队列移出数据的一个过程。该过程不是中断驱动的，它仅在一个程序需要来自键盘的信息时被调用。例12-20给出了键盘过程的调用程序。

例12-19

```
LOOK   DB      1,4,7,10              ;查找表
       DB      2,5,8,0
       DB      3,6,9,11

KEY    PROC    NEAR USES BX

       MOV  BX,OPNTR
       .IF BX == PNTR                ;如果队列为空
            STC
       .ELSE
            MOV AL,[BX]              ;获取队列数据
            INC BX
```

```
            .IF BX == OFFSET QUEUE+16
                  MOV BX,OFFSET QUEUE
            .ENDIF
            MOV OPNTR,BX
            MOV BX,LOOK
            XLAT
            CLC
      .ENDIF
      RET

KEY   ENDP
```

例 12-20

```
.REPEAT
      CALL KEY
.UNTIL !CARRY?
```

12.6 小结

1）中断是硬件或软件激发的一次调用，它在任何时刻中断当前正在执行的程序并调用一个过程。该过程由中断处理器或中断服务程序调用。

2）当一个低数据传输率 I/O 设备只是偶尔需要服务时，中断是很有用的。

3）微处理器有 5 条指令用于中断：BOUND、INT、INT 3、INTO 和 IRET。INT 和 INT 3 指令用存储在中断向量中的地址来调用过程，中断向量的类型由指令指出。BOUND 指令是一个条件中断，它使用中断向量类型号 5。INTO 指令也是一个条件中断，它只有在溢出标志被置位时中断一个程序。最后，IRET 指令用于从中断服务程序返回。

4）微处理器有 3 个引脚应用于硬件中断结构：INTR、NMI 和$\overline{\text{INTA}}$。中断输入为 INTR 和 NMI，它们用于申请中断。$\overline{\text{INTA}}$是一个输出引脚，用于响应 INTR 的中断请求。

5）实模式中断通过向量表被引用，向量表占据存储单元 00000H ~ 003FFH。每个中断向量为 4 字节长，包含中断服务程序的偏移地址和段地址。在保护模式下，中断引用包含 256 个中断描述符的中断描述符表（IDT）。每个中断描述符包含一个段选择符和一个 32 位偏移地址。

6）有 2 个标志位用于微处理器的中断结构：陷阱（TF）和中断允许（IF）。IF 标志位允许 INTR 中断输入，在每条指令执行后，只要 TF 有效，TF 标志位就引起中断。

7）前 32 个中断向量单元保留给 Intel 使用，有许多已在微处理器中预先确定了。最后 224 个中断向量供用户使用，可完成任何需要的功能。

8）一旦检测到一个中断，就会发生如下事件：（1）标志被压入堆栈；（2）IF 和 TF 标志位均被清除；（3）IP 和 CS 寄存器均被压入堆栈；（4）中断向量从中断向量表中取出，并通过向量地址访问中断服务子程序。

9）跟踪或单步通过设置 TF 标志位来实现，这使得每条指令执行后引起一次中断，从而便于调试。

10）非屏蔽中断输入（NMI）调用其地址存于中断向量类型 2 中的过程。此输入为上升沿触发。

11）INTR 引脚不像 NMI 引脚一样被内部译码，相反，$\overline{\text{INTA}}$用于在$\overline{\text{INTA}}$脉冲期间将中断向量类型号加到数据总线 $D_0 \sim D_7$上。

12）在$\overline{\text{INTA}}$脉冲期间将中断向量类型号加到数据总线 $D_0 \sim D_7$ 上的方法各不相同。一种方法是使用电阻将中断向量类型号 FFH 加到数据总线上，另一种方法是使用一个三态缓冲器来加送任何中断向量类型号。

13）8259A 可编程中断控制器（PIC）给微处理器增加了至少 8 个中断输入。如果需要更多的中断，该器件可级联以提供最多 64 个中断输入。

14）编程 8259A 分 2 步处理。首先，给 8259A 发送一系列初始化命令字（ICW），然后发送一系列操作命令字（OCW）。

15）8259A 包含 3 个状态寄存器：IMR（中断屏蔽寄存器）、ISR（在服务寄存器）及 IRR（中断请求寄存器）。

16）一个实时时钟用于以真实时间计时。在大多数情况下，时间以二进制或 BCD 形式存储于几个存储单元中。

12.7 习题

1. 一个中断所中断的是什么？
2. 定义术语：中断。
3. 中断调用的是什么？
4. 中断为什么给微处理器节约了时间？
5. 列出微处理器上的中断引脚。
6. 列出微处理器的 5 个中断指令。

7. 什么是中断向量?
8. 中断向量位于微处理器存储器中的什么地方?
9. 在中断向量表中有多少个不同的中断向量?
10. Intel 保留了哪些中断向量?
11. 解释类型 0 中断是如何产生的。
12. 保护模式操作的中断描述符表位于什么地方?
13. 每个保护模式中断描述符包含什么信息?
14. 描述保护模式中断与实模式中断之间的区别。
15. 描述 BOUND 指令的操作。
16. 描述 INTO 指令的操作。
17. 哪些存储单元包含 INT 44H 指令的向量?
18. 解释 IRET 指令的操作。
19. IRETQ 指令用在哪里?
20. 中断向量类型 7 的用途是什么?
21. 列出当一个中断变为有效时所发生的事件。
22. 解释中断标志（IF）的用途。
23. 解释陷阱标志（TF）的用途。
24. IF 是如何被清除和置位的?
25. TF 是如何被清除和置位的?
26. NMI 中断输入通过哪个向量类型号自动获得向量?
27. 激活$\overline{\text{INTA}}$信号是为了 NMI 引脚吗?
28. INTR 输入是_____敏感的。
29. NMI 输入是_____敏感的。
30. 当$\overline{\text{INTA}}$信号变为逻辑 0 时，它表明微处理器正在等待一个中断_____号置于数据总线（$D_0 \sim D_7$）上。
31. 什么是 FIFO?
32. 设计一个电路，将中断类型号 86H 置于数据总线上以响应 INTR 输入。
33. 设计一个电路，将中断类型号 CCH 置于数据总线上以响应 INTR 输入。
34. 解释为什么 $D_0 \sim D_7$ 上的上拉电阻使微处理器响应$\overline{\text{INTA}}$脉冲期间的中断向量类型号 FFH。
35. 什么是菊花链?
36. 为什么在一个菊花链中断系统中必须查询中断设备?
37. 什么是 8259A?
38. 为具有 64 个中断输入，需要多少个 8259A?
39. 8259A 上的 $IR_0 \sim IR_7$ 引脚的用途是什么?
40. 何时使用 8259A 上的 $CAS_2 \sim CAS_0$ 引脚?
41. 在一个级联系统中，从 8259A 的 INT 引脚连接到主 8259A 的什么地方?
42. 什么是 OCW?
43. 什么是 ICW?
44. 中断向量类型号存储于 8259A 的什么地方?
45. 当 8259A 作为单个主 8259A 工作在系统中时，需要多少个 ICW 编程 8259A?
46. ICW_1 的用途是什么?
47. IR 引脚的有效极性被编程在 8259A 的什么地方?
48. 解释 8259A 中的优先权循环。
49. 什么是普通 EOI?
50. 在 PC 机中，主 8259A PIC 处于哪些 I/O 端口?
51. 8259A 中的 IRR 的用途是什么?
52. 在 PC 机中，从 8259A 处于哪些 I/O 端口?

第 13 章　直接存储器存取及 DMA 控制 I/O

引言

前面的章节讨论了基本 I/O 和中断处理 I/O。现在我们转而讨论最后一种形式的 I/O，称为**直接存储器存取（direct memory access，DMA）**。在微处理器临时被禁止时 DMA I/O 技术提供直接对存储器的存取。它允许数据在存储器与 I/O 设备之间以某种速率传输，该速率仅受系统中存储器件或 DMA 控制器的速度限制。在当今高速 RAM 存储器件的支持下，DMA 传输速率可达 33 ～ 150MB/s。

DMA 传输有很多用途，但更常见的是 DRAM 刷新、视频显示刷新屏幕以及磁盘存储系统读写。DMA 传输还用于高速存储器到存储器之间的传输。

本章还解释常用 DMA 处理的磁盘存储系统和视频系统的操作。磁盘存储器包括软盘、硬盘以及光盘存储。视频系统包括数字显示器和模拟显示器。

目的

读者学习完本章后将能够：

1）描述 DMA 传输。
2）解释 HOLD 和 HLDA 直接存储器存取控制信号的操作。
3）解释 8237 DMA 控制器用于 DMA 传输时的功能。
4）编程 8237 以实现 DMA 传输。
5）描述 PC 机系统中的磁盘标准。
6）描述 PC 机中的各种视频接口标准。

13.1　基本 DMA 操作

在基于微处理器的系统中，有两个控制信号用于请求和响应直接存储器存取（DMA）传输。HOLD 引脚为输入引脚，用于请求 DMA 操作；HLDA 引脚为输出引脚，用于响应 DMA 操作。图 13-1 给出了这两个 DMA 控制引脚的典型时序图。

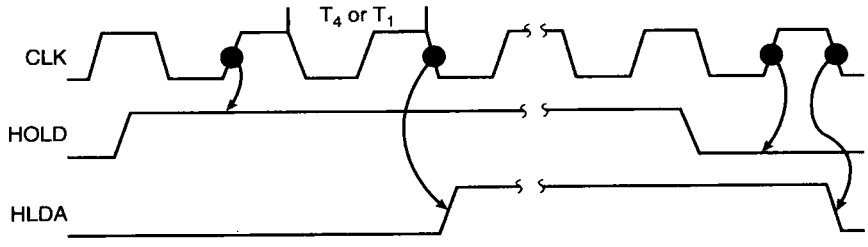


图 13-1　微处理器的 HOLD 和 HLDA 时序

一旦 HOLD 输入被置为逻辑 1，则请求 DMA 操作。微处理器在几个时钟内响应，它将正在执行的程序挂起，并将其地址、数据和控制总线置为高阻抗状态。这使微处理器看起来似乎被从插座里拔走了。这种状态允许外部 I/O 设备或其他微处理器获得对系统总线的访问权，因此存储器可直接被存取。

正如时序图所示，HOLD 在每个时钟周期的中间被采样，因此，HOLD 可以在微处理器指令集里任意一个指令处于执行状态的任何时刻生效。一旦微处理器识别出 HOLD 信号，它就停止执行程序，并进入 HOLD 周期。注意 HOLD 输入比 INTR 或 NMI 中断输入的优先级更高。中断在指令的末尾生效，

而 HOLD 在指令的中间生效。微处理器惟一比 HOLD 优先级更高的引脚是 RESET 引脚。注意，HOLD 输入在 RESET 期间不会有效，否则不能保证复位。

HLDA 信号变为有效，以指示微处理器确实将其总线置为高阻抗状态，如时序图所示。注意，在 HOLD 变化和直到 HLDA 变化的时刻之间有几个时钟周期。HLDA 输出信号给外部请求 DMA 操作的设备，通知它们微处理器已放弃它对存储器和 I/O 空间的控制权。HOLD 输入可称为 DMA 请求输入，而 HLDA 输出可称为 DMA 允许信号。

基本 DMA 定义

直接存储器存取通常发生在 I/O 设备与存储器之间，而与微处理器无关。**DMA 读（DMA read）**将数据从存储器传输给 I/O 设备，**DMA 写（DMA write）**将数据从 I/O 设备传输给存储器。在两种操作中，存储器和 I/O 设备同时被控制，这也是为什么系统包含独立的存储器与 I/O 控制信号的原因。微处理器的这种特殊控制总线结构允许 DMA 传输。DMA 读使$\overline{\text{MRDC}}$和$\overline{\text{IOWC}}$信号同时被激活，从而从存储器传输数据给 I/O 设备。DMA 写使$\overline{\text{MWTC}}$和$\overline{\text{IORC}}$信号同时被激活。除 8086/8088 系统外的 Intel 系列所有微处理器都具有这些控制总线信号，8086/8088 需要用系统控制器或诸如图 13-2 所示的电路来产生这些信号。DMA 控制器为存储器提供其地址以及来自控制器的一个信号（$\overline{\text{DACK}}$），用于在 DMA 传输期间选择 I/O 设备。

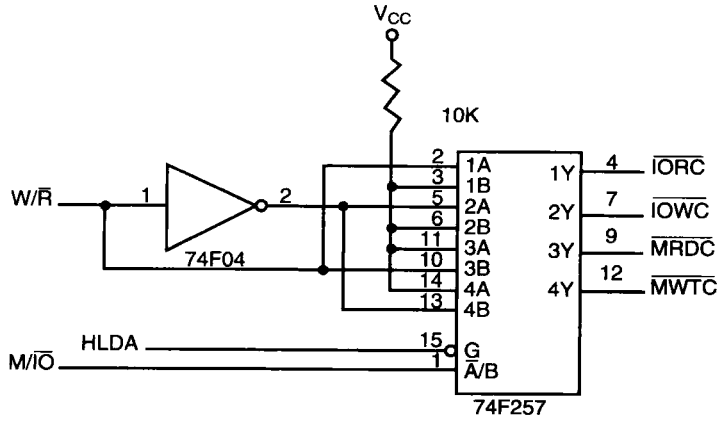


图 13-2 在 DMA 环境中产生系统控制信号的电路

数据传输速度由存储器件或常用来控制 DMA 传输的 DMA 控制器的速度决定。如果存储器速度为 50ns，则 DMA 传输以最多 1/50ns（即每秒 20MB）的速率进行。如果系统中 DMA 控制器以最大 15MHz 速率工作，且我们仍使用 50ns 存储器，则最大传输速率为 15MHz，因为 DMA 控制器比存储器速度要慢。在许多情况下，当进行 DMA 传输时，DMA 控制器降低了系统速度。

由于现代计算机系统数据传输向串行数据传输方式的转变，DMA 也变得不再那么重要了。PCI Express 总线便是串行传输，其数据传输速率已超过 DMA，甚至磁盘驱动的 SATA（串行 ATA）接口采用串行传输速率就可达 300Mbps，以至于其替代了用于硬盘驱动的 DMA 传输。主板上使用串行技术的部件之间的串行传输对于 PCI Express 连接可达到 20Gbps 的传输速率。

13.2 8237 DMA 控制器

8237 DMA 控制器给存储器和 I/O 提供 DMA 传输期间的控制信号及存储器地址信息。8237 实际上是一个特殊用途的微处理器，其工作是在存储器与 I/O 之间进行高速数据传输。图 13-3 给出了 8237 可编程 DMA 控制器的引脚和框图。尽管该器件不会作为一个分立器件出现在现代微处理器系统中，但它却出现在大多数系统的系统控制器芯片组中。尽管由于其复杂性而在这里没有描述它们，但芯片组

(ISP 或集成系统外围控制器) 及组成它的 2 个 DMA 控制器均可像 8237 一样被编程。ISP 还为系统提供一对 8259A 可编程中断控制器。

8237 是一个 4 通道器件，它与 8086/8088 微处理器兼容。8237 可扩展为包含任意数目的 DMA 通

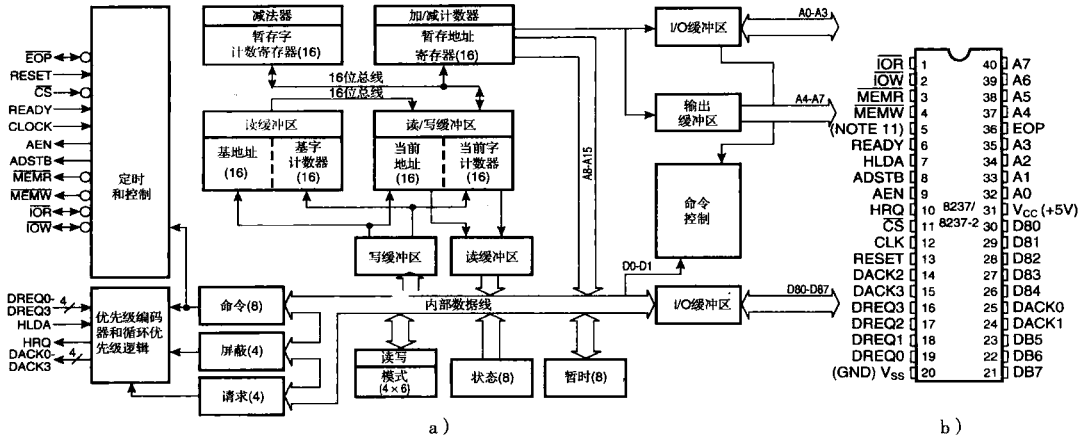


图 13-3 8237A-5 可编程 DMA 控制器

a) 框图 b) 引脚图

道，尽管 4 个通道对许多小系统来说似乎是足够了。8237 能以最大 1.6MB ps 的速率进行 DMA 传输。每个通道能够寻址一个完整的 64KB 存储器段，并且一次编程可传输最多 64KB 数据。

引脚定义

CLK **时钟** (**clock**) 输入与系统时钟信号连接，只要该信号为 5MHz 或更低。在 8086/8088 系统中，时钟必须反相，以保证 8237 正确工作。

$\overline{CS}$ **片选** (**chip select**) 引脚使能 8237 进行编程。$\overline{CS}$引脚通常与译码器的输出相连。译码器不使用 8086/8088 控制信号 IO/$\overline{M}$ (M/$\overline{IO}$)，因为它有新的存储器与 I/O 控制信号 ($\overline{MEMR}$、$\overline{MEMW}$、$\overline{IOR}$和$\overline{IOW}$)。

RESET **复位** (**reset**) 引脚清除命令、状态、请求以及临时存储器。它还清除高/低触发器并设置屏蔽寄存器。该输入初始化 8237，所以常被禁止，直到被另外编程。

READY 把逻辑 0 加到**就绪** (**ready**) 输入上则使 8237 进入等待状态，以等待较慢的存储器或 I/O 器件。

HLDA **保持响应** (**hold acknowledge**) 通知 8237，微处理器已放弃对地址、数据及控制总线的控制权。

$DREQ_3 \sim DREQ_0$ **DMA 请求** (**DMA request**) 输入用于为 4 个 DMA 通道中的每一个请求 DMA 传输。由于这些输入的极性是可编程的，所以它们可以是高有效输入也可以是低有效输入。

$DB_7 \sim DB_0$ **数据总线** (**data bus**) 引脚与微处理器的数据总线相连，并在 DMA 控制器编程期间使用。

$\overline{IOR}$ **I/O 读** (**I/O read**) 是一个双向引脚，用在编程及 DMA 写周期期间。

$\overline{IOW}$ **I/O 写** (**I/O write**) 是一个双向引脚，用在编程及 DMA 读周期期间。

$\overline{EOP}$ **过程结束** (**end-of-process**) 是一个双向信号，用作输入时终止 DMA 过程，用作输出时通知 DMA 传输的结束。该输入常用于在 DMA 周期的末尾中断 DMA 传输。

$A_3 \sim A_0$ 这些**地址引脚** (**address pin**) 在编程期间选择内部寄存器，还在 DMA 操作期间提供部分 DMA 传输地址。

$A_7 \sim A_4$ 这些地址引脚为输出引脚，在 DMA 操作期间提供部分 DMA 传输地址。

HRQ **保持请求** (**hold request**) 输出与微处理器的 HOLD 输入相连，以请求 DMA 传输。

$DACK_3 \sim DACK_0$ **DMA 通道响应（DMA channel acknowledge）**输出，响应一个通道的 DMA 请求。这些输出可编程为高有效或低有效信号。DACK 输出常用于在 DMA 传输期间选择 DMA 控制的 I/O 设备。

AEN **地址使能（address enable）**信号使能 DMA 地址锁存器与 8237 的 $DB_7 \sim DB_0$ 引脚相连。它还用于禁止系统中任何与微处理器相连的缓冲器。

ADSTB **地址选通（address strobe）**与 ALE 功能相同，只是它用于 DMA 控制器在 DMA 传输期间锁存地址位 $A_{15} \sim A_8$。

$\overline{MEMR}$ **存储器读（memory read）**是一个输出信号，它使存储器在 DMA 读周期期间读出数据。

$\overline{MEMW}$ **存储器写（memory write）**是一个输出信号，它使存储器在 DMA 写周期期间写入数据。

内部寄存器

CAR **当前地址寄存器（current address register）**用来保持用于 DMA 传输的 16 位存储器地址，每个通道都有用于此目的的当前地址寄存器。当在 DMA 操作中传输一字节数据时，CAR 加 1 或减 1，这取决于它是如何被编程的。

CWCR **当前字计数寄存器（current word count register）**用于编程 DMA 操作中一个通道所传输的字节数（最多 64K）。装入此寄存器的数比所传输的字节数少 1。例如，如果 10 被装入 CWCR，则在 DMA 操作中传输 11 字节数据。

BA 和 BWC **基地址（base address，BA）**和**基字计数（base word count，BWC）**寄存器用于通道选择了自动初始化模式时。在自动初始化模式下，这些寄存器用于在 DMA 操作完成后对 CAR 和 CWCR 再装入。这就允许使用同一计数和地址从同一存储器区域传输数据。

CR **命令寄存器（command register）**编程 8237 DMA 控制器的操作。图 13-4 给出了命令寄存器的功能。

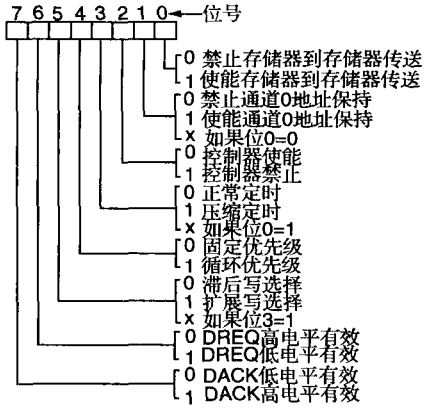


图 13-4 8237A-5 命令寄存器

命令寄存器使用位 0 选择存储器到存储器的 DMA 传输模式。存储器到存储器 DMA 传输使用 DMA 通道 0 存放源地址，使用 DMA 通道 1 存放目标地址（这类似于 MOVSB 指令的操作）。一个字节从由通道 0 访问的地址读出，并存入 8237 内部的暂存寄存器中。然后，8237 启动一个存储器写周期，此时暂存寄存器的内容被写入由 DMA 通道 1 选择的地址中。所传输的字节数由通道 1 计数寄存器决定。

通道 0 地址保持使能位（位 1）编程通道 0，使之用于存储器到存储器的传输。例如，如果必须用数据填充一个存储器区域，则通道 0 可保持在同一地址，而通道 1 改变，采用存储器到存储器传输。这样就将由通道 0 寻址的内容复制到由通道 1 存取的存储体中。

控制器使能/禁止位（位 2）打开和关闭整个控制器。正常和压缩位（位 3）决定一个 DMA 周期包含 2 个（压缩）还是 4 个（正常）时钟周期。位 5 用于正常时序中扩展写脉冲，使得在需要一个更宽写脉冲的 I/O 设备的时序中，写脉冲提前一个时钟出现。

位 4 选择 4 个 DMA 通道 DREQ 输入的优先级。在固定优先级方案中，通道 0 有最高优先级而通道 3 有最低优先级。在循环优先级方案中，最近刚服务过的通道优先级最低。例如，如果通道 2 刚进行了 DMA 传输，则它呈现最低优先级，而通道 3 呈现最高优先级。循环优先级给所有通道以相等的优先级。

其余 2 位（位 6 和位 7）编程 DREQ 输入和 DACK 输出的极性。

MR **模式寄存器（mode register）**编程一个通道的操作模式。注意，每个通道有其自己的模

式寄存器（见图 13-5），由位 1 和位 0 选择通道。模式寄存器的其余各位选择操作类型、自动初始化、加 1/减 1 以及通道操作方式。校验操作产生 DMA 地址，但不产生 DMA 存储器和 I/O 控制信号。

操作方式包括请求传输模式，单字节传输模式，块传输模式以及级联传输模式。请求传输模式传输数据，直到输入一个外部 EOP 信号，或 DREQ 输入变为无效时为止。单字节传输模式在传输每字节数据后释放 HOLD 信号，如果 DREQ 引脚保持有效，则 8237 通过连到微处理器 HOLD 输入上的 DRQ 线再次请求 DMA 传输。块传输模式自动传输由通道的计数寄存器指示的字节数，在块传输模式中 DREQ 无须保持有效。级联传输模式用于系统中存在不止一个 8237 时。

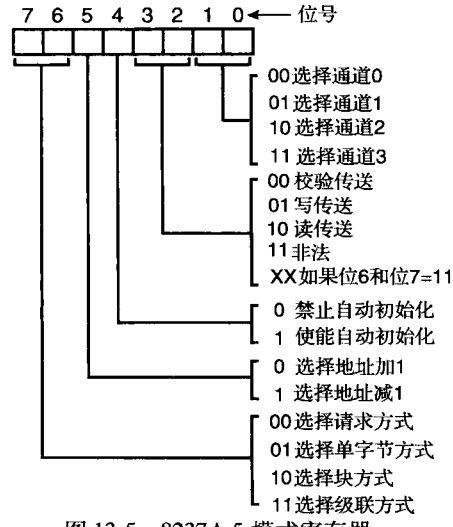


图 13-5 8237A-5 模式寄存器

BR **请求寄存器（bus request register）** 用于通过软件请求 DMA 传输（见图 13-6）。这在存储器到存储器传输中非常有用，因为在这种情况下不能用外部信号启动 DMA 传输。

MRSR **屏蔽寄存器置位/复位（mask register set/reset）** 用来设置或清除通道屏蔽，如图 13-7 所示。如果置位屏蔽位，则该通道的请求被禁止。回忆一下，RESET 信号置位所有通道的屏蔽位，以禁止它们的 DMA 请求。

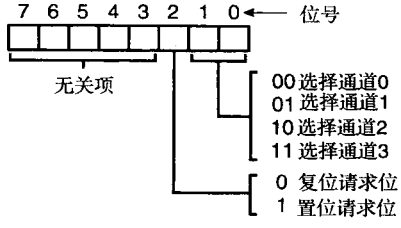


图 13-6 8237A-5 请求寄存器

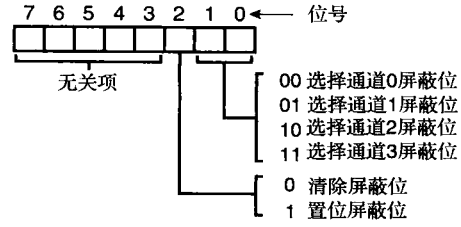


图 13-7 8237A-5 屏蔽寄存器置位/复位模式

MSR **屏蔽寄存器（mask register）** 见图 13-8，用一条命令清除或置位所有的屏蔽位，而不是像 MRSR 那样只对单独的通道操作。

SR **状态寄存器（status register）** 显示了每个 DMA 通道的状态（见图 13-9）。TC 位指示通道是否已达到其终点计数值（即传输完所有字节）。一旦达到终点计数值，则大多数操作模式的 DMA 传输被终止。请求位指示对于某给定通道的 DREQ 输入是否有效。

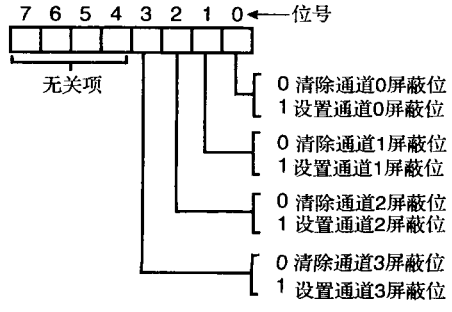


图 13-8 8237A-5 屏蔽寄存器

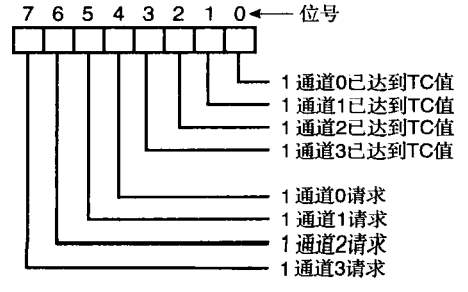


图 13-9 8237A-5 状态寄存器

13.2.1 软件命令

有3条软件命令用于控制8237的操作。这些命令与8237内部的各种控制寄存器不同，它们没有二进制位模式。一个对正确端口号的简单输出即发出软件命令。图13-10给出了访问所有寄存器和软件命令的I/O端口分配。

软件命令的功能解释如下：

1）**清除高/低触发器**：清除8237内部的高/低（F/L）触发器。F/L触发器选择在当前地址寄存器和当前计数寄存器中哪一个字节（低位或高位）被读/写。如果F/L = 0，则选中低位字节；如果F/L = 1，则选中高位字节。对地址寄存器和计数寄存器的任何读写都将自动翻转F/L触发器。

2）**主清除**：与8237的RESET信号作用完全相同。像RESET信号一样，该命令禁止所有通道的DMA请求。

3）**清除屏蔽寄存器**：使能所有4个DMA通道。

信号						操作
A3	A2	A1	A0	$\overline{IOR}$	$\overline{IOW}$	
1	0	0	0	0	1	读状态寄存器
1	0	0	0	1	0	写命令寄存器
1	0	0	1	0	1	非法
1	0	0	1	1	0	写请求寄存器
1	0	1	0	0	1	非法
1	0	1	0	1	0	写单字节屏蔽寄存器位
1	0	1	1	0	1	非法
1	0	1	1	1	0	写模式寄存器
1	1	0	0	0	1	非法
1	1	0	0	1	0	清除字节指针触发器
1	1	0	1	0	1	读暂存寄存器
1	1	0	1	1	0	主清除
1	1	1	0	0	1	非法
1	1	1	0	1	0	清除屏蔽寄存器
1	1	1	1	0	1	非法
1	1	1	1	1	0	写所有屏蔽寄存器位

图13-10 8237A-5命令和控制端口分配

13.2.2 地址寄存器和计数寄存器编程

图13-11给出了编程每个通道的计数和地址寄存器的I/O端口地址。注意F/L触发器的状态决定

通道	寄存器	操作	信号							内部触发器	数据总线DB0～DB7
			$\overline{CS}$	$\overline{IOR}$	$\overline{IOW}$	A3	A2	A1	A0		
0	基和当前地址	写	0	1	0	0	0	0	0	0	A0～A7
			0	1	0	0	0	0	0	1	A8～A15
	当前地址	读	0	0	1	0	0	0	0	0	A0～A7
			0	0	1	0	0	0	0	1	A8～A15
	基和当前字计数	写	0	1	0	0	0	0	1	0	W0～W7
			0	1	0	0	0	0	1	1	W8～W15
	当前字计数	读	0	0	1	0	0	0	1	0	W0～W7
			0	0	1	0	0	0	1	1	W8～W15
1	基和当前地址	写	0	1	0	0	0	1	0	0	A0～A7
			0	1	0	0	0	1	0	1	A8～A15
	当前地址	读	0	0	1	0	0	1	0	0	A0～A7
			0	0	1	0	0	1	0	1	A8～A15
	基和当前字计数	写	0	1	0	0	0	1	1	0	W0～W7
			0	1	0	0	0	1	1	1	W8～W15
	当前字计数	读	0	0	1	0	0	1	1	0	W0～W7
			0	0	1	0	0	1	1	1	W8～W15
2	基和当前地址	写	0	1	0	0	1	0	0	0	A0～A7
			0	1	0	0	1	0	0	1	A8～A15
	当前地址	读	0	0	1	0	1	0	0	0	A0～A7
			0	0	1	0	1	0	0	1	A8～A15
	基和当前字计数	写	0	1	0	0	1	0	1	0	W0～W7
			0	1	0	0	1	0	1	1	W8～W15
	当前字计数	读	0	0	1	0	1	0	1	0	W0～W7
			0	0	1	0	1	0	1	1	W8～W15
3	基和当前地址	写	0	1	0	0	1	1	0	0	A0～A7
			0	1	0	0	1	1	0	1	A8～A15
	当前地址	读	0	0	1	0	1	1	0	0	A0～A7
			0	0	1	0	1	1	0	1	A8～A15
	基和当前字计数	写	0	1	0	0	1	1	1	0	W0～W7
			0	1	0	0	1	1	1	1	W8～W15
	当前字计数	读	0	0	1	0	1	1	1	0	W0～W7
			0	0	1	0	1	1	1	1	W8～W15

图13-11 8237A-5 DMA通道I/O端口地址

是LSB还是MSB被编程。如果不知道F/L触发器的状态，则计数和地址寄存器的编程就不会正确。DMA通道在其计数和地址寄存器被编程之前必须被禁止，这一点也很重要。

编程8237需要4个步骤：1）使用清除F/L命令来清除F/L触发器；2）禁止通道请求；3）编程地址寄存器的LSB，然后是MSB；4）编程计数寄存器的LSB和MSB。一旦这4个操作完成，则该通道被编程，并准备使用。在通道被允许和开始前，需要另外编程来选择操作模式。

13.2.3 8237与80X86微处理器相连

图13-12给出了包含8237 DMA控制器的基于80X86的系统。

8237的地址使能（AEN）输出信号控制锁存器的输出引脚及74LS257（E）的输出。在正常的80X86操作期间（AEN = 0），锁存器A和C以及多路器（E）提供地址总线位A_{19}～A_{16}与A_7～A_0。只要是80X86控制系统，多路器就给系统提供控制信号。在DMA操作期间（AEN = 1），锁存器A和C与多路器（E）一起被禁止。锁存器D和B现在提供地址总线位A_{19}～A_{16}与A_{15}～A_8。地址总线位A_7～A_0直接由8237提供，它们是DMA传输地址的一部分。控制信号$\overline{MEMR}$、$\overline{MEMW}$、$\overline{IOR}$和$\overline{IOW}$由DMA控制器提供。

8237的地址选通输出（ADSTB）在DMA操作期间将地址（A_{15}～A_8）同步地输入锁存器D，使整个DMA传输地址出现在地址总线上。地址总线位A_{19}～A_{16}由锁存器B提供，在允许控制器进行DMA传输之前，锁存器B必须编程这4个地址位。8237的DMA操作被限制为只传输处于同一个64KB存储器段中的不超过64KB的数据。

译码器（F）选择8237进行编程，同时把4位锁存器（B）作为地址位的最高4位。PC机中的锁存器称为DMA页面寄存器（8位），它用来存放DMA传输的地址位A_{16}～A_{23}。还有一个高页面寄存器，但其地址是随芯片而定的。DMA页面寄存器的端口号列在表13-1中（这些用于Intel ISP）。本系统的译码器允许8237的I/O端口地址为XX60H～XX7FH，I/O锁存器（B）的端口地址为XX00H～XX1FH。注意，译码器输出与$\overline{IOW}$信号组合在一起为锁存器（B）产生一个高电平有效的时钟。

表13-1 DMA页面寄存器端口

通 道	端口号（A_{16}～A_{23}）	端口号（A_{24}～A_{31}）
0	87H	487H
1	83H	483H
2	81H	481H
3	82H	482H
4	8FH	48FH
5	8BH	48BH
6	89H	489H
7	8AH	48AH

在正常的80X86操作期间，DMA控制器与集成电路B和D被禁止。在一次DMA操作中，集成电路A、C和E被禁止，使8237可获得地址、数据和控制总线而得到对系统的控制权。

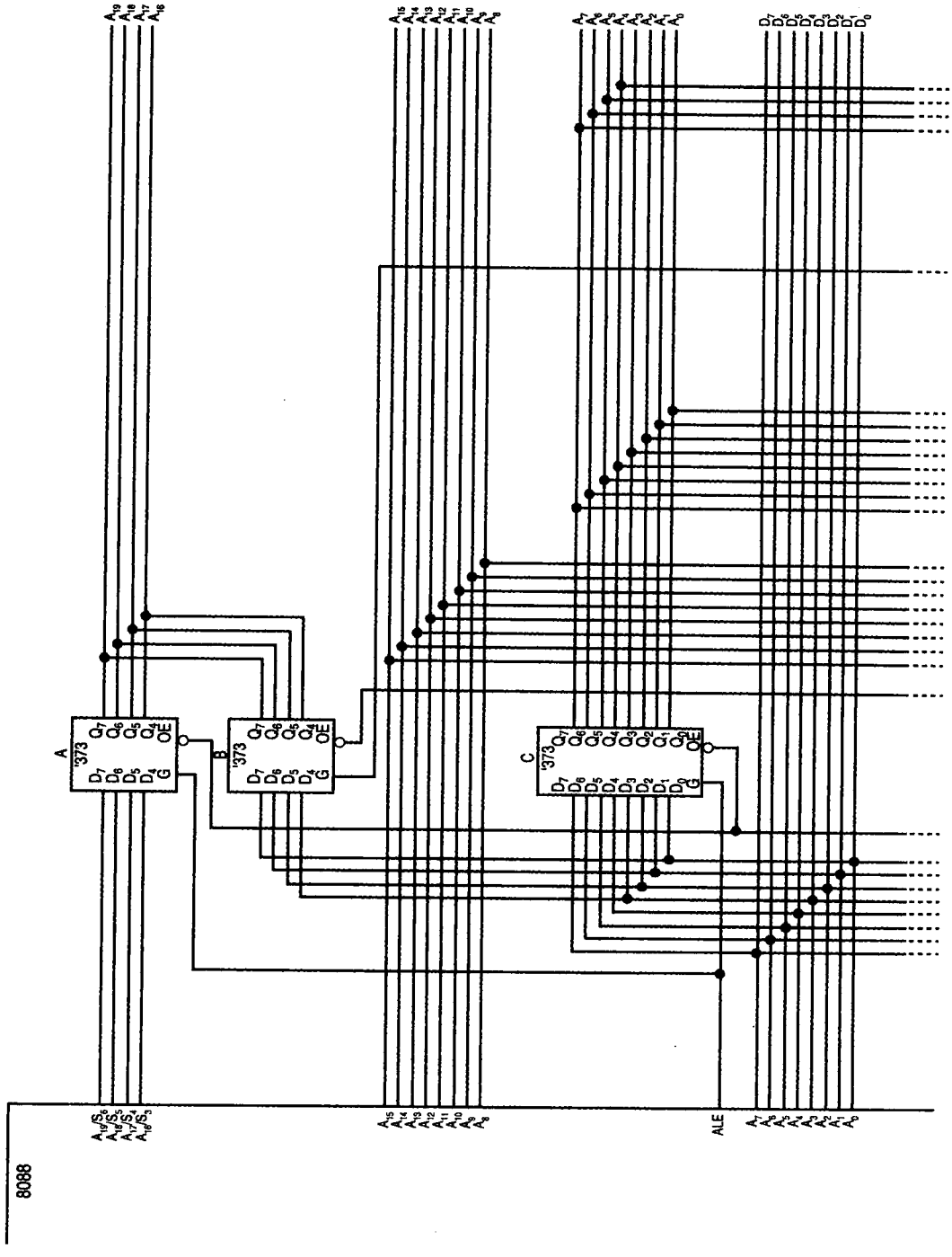
在PC机中，2个DMA控制器被编程为，I/O端口0000H～000FH用于DMA通道0～3，端口00C0H～00DFH用于DMA通道4～7。注意，第2个控制器只能在偶地址编程，所以通道4的基址和当前地址寄存器编程在I/O端口00C0H，而通道4的基址和当前计数值寄存器编程在I/O端口00C2H。页面寄存器保持DMA地址位A_{23}～A_{16}，它位于I/O端口0087H（CH-0）、0083H（CH-1）、0081H（CH-2）、0082H（CH-3）、（无通道4）、008BH（CH-5）、0089H（CH-6）及008AH（CH-7）。页面寄存器的功能与本书中许多例子所描述的地址锁存器相同。

13.2.4 用8237进行存储器到存储器传输

存储器到存储器传输甚至比自动重复的MOVSB指令功能更强大。（注意，大多数现代芯片组不支持存储器到存储器传输）。虽然指令表说明重复的MOVSB指令对8088每字节需4.2 μs，而8237每字节仅需2.0 μs，这比软件数据传输要快2倍多，但如果系统使用80386、80486或Pentium～Pentium 4，则情况并非如此。

存储器到存储器DMA传输

假定存储单元10000H～13FFFH中的内容要传输到存储单元14000H～17FFFH中，这可以用一条重复串移动指令来实现，或者用DMA控制器以更快的速率来实现。


A19 A18 A17 A16
A15 A14 A13 A12 A11 A10 A9 A8
A7 A6 A5 A4 A3 A2 A1 A0
D7 D6 D5 D4 D3 D2 D1 D0
A
B
C
'373
D7 D6 D5 D4 G OE
Q7 Q6 Q5 Q4
Q3 Q2 Q1 Q0
A19/S6 A18/S5 A17/S4 A16/S3
ALE
8088

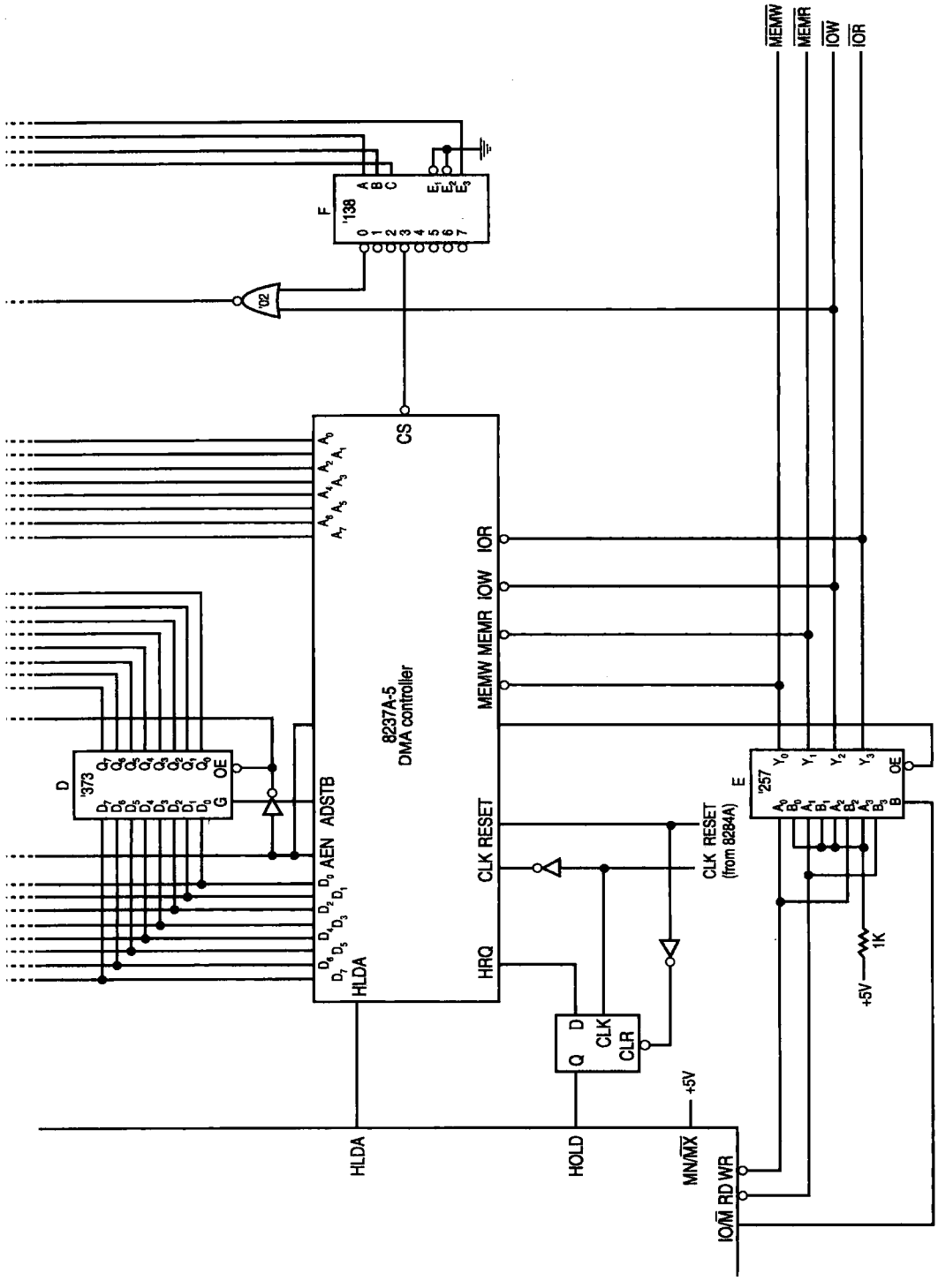


图13-12 完整的8086最小模式DMA系统

例 13-1 给出了初始化 8237 以及编程图 13-12 中用于 DMA 传输的锁存器 B 所需软件。该软件用于嵌入式应用。为使其用于 PC 机中（如果芯片组支持该特性），必须使用表 13-1 所示的页面寄存器的端口地址。

例 13-1

```
;该过程用图13-12中8237A DMA控制器传送数据块,这是一种存储器至存储器的块传送
;调用参数:
;    SI = 源地址
;    DI = 目的地址
;    CX = 计数
;    ES = 源和目的的段

LATCHB EQU   10H
CLEARF EQU   7CH
CH0A   EQU   70H
CH1A   EQU   72H
CH1C   EQU   73H
MODE   EQU   7BH
CMMD   EQU   78H
MASKS  EQU   7FH
REQ    EQU   79H
STATUS EQU   78H

TRANS  PROC  NEAR USES AX

    MOV   AX,ES                ;编程锁存器 B
    MOV   AL,AH
    SHR   AL,4
    OUT   LATCHB,AL
    OUT   CLEARF,AL            ;清除 F/L 触发器

    MOV   AX,ES                ;编程源地址
    SHL   AX,4
    ADD   AX,SI
    OUT   CH0A,AL
    MOV   AL,AH
    OUT   CH0A

    MOV   AX,ES                ;编程目的地址
    SHL   AX,4
    ADD   AX,DI
    OUT   CH1A,AL
    MOV   AL,AH
    OUT   CH1A,AL

    MOV   AX,CX                ;编程计数
    DEC   AX
    OUT   CH1C,AL
    MOV   AL,AH
    OUT   CH1C,AL

    MOV   AL,88H               ;编程模式
    OUT   MODE,AL
    MOV   AL,85H
    OUT   MODE,AL

    MOV   AL,1                 ;允许块传送
    OUT   CMMD,AL
    MOV   AL,0EH               ;0通道解除屏蔽
    OUT   MASKS,AL

    MOV   AL,4                 ;启动DMA传送
    OUT  REQ,AL

    .REPEAT                   ;等待直到DMA完成
          IN   AL,STATUS
```

```
    .UNTIL AL & 1
    RET

TRANS ENDP
```

编程 DMA 控制器需要几个步骤，如例 13-1 所示。5 位地址的最左边 1 位被送给锁存器 B，然后，在清除 F/L 触发器后对通道进行编程。注意，在存储器到存储器传输中使用通道 0 作为源，通道 1 作为目的。接着用比将要传输的字节数小 1 的数作为计数值进行编程。下一步，对每个通道的模式寄存器编程，命令寄存器选择块移动，使能通道 0，并启动软件 DMA 请求。在从过程返回之前，测试状态寄存器的终点计数值 TC。回忆一下，TC 标志表明 DMA 传输已完成。TC 还禁止了通道请求，以防止另外的传输。

使用 8237 进行存储器填充

为使用同样的数据填充一段存储器区域，通道 0 的源寄存器被编程为在传输过程中指向同一地址，这是用通道 0 保持模式实现的。控制器将这一存储单元的内容复制到由通道 1 寻址的整个存储体中。这个功能非常有用。

例如，假设一个 DOS 视频显示必须被清除。这一操作可以由 DMA 控制器用通道 0 保持模式和存储器到存储器传输完成。如果视频显示器包含 80 列 25 行，则它所有 2000 个显示位置都必须被设置为 20H（ASCII 空格）以清屏。

例 13-2 给出了一个过程，它清除由 ES：DI 寻址的一段存储器区域。CX 寄存器把要清除的字节数传输给 CLEAR 过程。注意该过程与例 13-1 几乎相同，只是命令寄存器被编程使其保持通道 0 地址。源地址被编程为与 ES：DI 地址相同，而目标地址被编程为比 ES：DI 地址高 1。还应注意该程序被设计成与图 13-12 中硬件一起工作，除非有相同的硬件，否则它在 PC 机中将不起作用。

例 13-2

```
;该过程用图13-12的8237A DMA控制器清除DOS模式视频屏幕

;调用参数:
;       DI = 被清除区的偏移地址
;       ES = 被清除区的段地址
;       CX = 被清除的字节数

LATCHB EQU   10H
CLEARF EQU   7CH
CHOA   EQU   70H
CH1A   EQU   72H
CH1C   EQU   73H
MODE   EQU   7BH
CMMD   EQU   78H
MASKS  EQU   7FH
REQ    EQU   79H
STATUS EQU   78H
ZERO   EQU   0

CLEAR  PROC  NEAR USES AX

    MOV    AX,ES                ;编程锁存器B
    MOV    AL,AH
    SHR    AL,4
    OUT    LATCHB,AL
    OUT    CLEARF,AL            ;清除F/L触发器

    MOV    AL,ZERO              ;把0保存到第一个字节

    MOV    ES:[DI],AL

    MOV    AX,ES                ;编程源地址
    SHL    AX,4
    ADD    AX,SI
```

```
    OUT   CH0A,AL
    MOV   AL,AH
    OUT   CH0A

    MOV   AX,ES                ;编程目的地址
    SHL   AX,4
    ADD   AX,DI
    OUT   CH1A,AL
    MOV   AL,AH
    OUT   CH1A,AL

    MOV   AX,CX                ;编程计数
    DEC   AX
    OUT   CH1C,AL
    MOV   AL,AH
    OUT   CH1C,AL

    MOV   AL,88H               ;编程模式
    OUT   MODE,AL
    MOV   AL,85H
    OUT   MODE,AL

    MOV   AL,03H               ;允许块保持传送
    OUT   CMMD,AL

    MOV   AL,0EH               ;允许0通道
    OUT   MASKS,AL

    MOV   AL,4                 ;启动DMA传送
    OUT   REQ,AL

    .REPEAT
        IN   AL,STATUS
    .UNTIL AL &  1
    RET

CLEAR  ENDP
```

13.2.5 DMA处理的打印机接口

图13-13在图13-12的基础上增加了一些硬件，使之成为DMA控制的打印机接口。它只增加了很少的附加电路就可用于与Centronics型并行打印机的接口。锁存器用于DMA传输期间当数据发送给打印机时捕获数据，在DMA操作期间传给锁存器的写脉冲还产生给打印机的单脉冲数据选通（$\overline{DS}$）信号。在每次打印机准备好接收另外的数据时，打印机返回$\overline{ACK}$信号。此电路中，$\overline{ACK}$信号用于通过触发器请求DMA操作。

注意，并不是通过译码地址总线上的地址选择I/O设备的。在DMA传输期间，地址总线包含存储器地址，但不能包含I/O端口地址。来自8237的$\overline{DACK3}$输出信号取代I/O端口地址，通过一个或门选通写脉冲，从而选择锁存器。

控制该接口的软件非常简单，因为只需编程数据的地址及要打印的字符数。一旦编程，则通道请求被允许。每次接口接收到打印机的$\overline{ACK}$信号时，则DMA操作一次传输一字节数据给打印机接口。

例13-3给出了打印当前数据段中数据的过程。该过程编程8237，但实际上并不打印任何数据，打印由DMA控制器和打印机接口完成。

例13-3

```
;该过程通过图13-13中的打印机接口打印数据

;调用参数:
;       BX = 打印数据的编移地址
;       DS = 打印数据的段地址
```

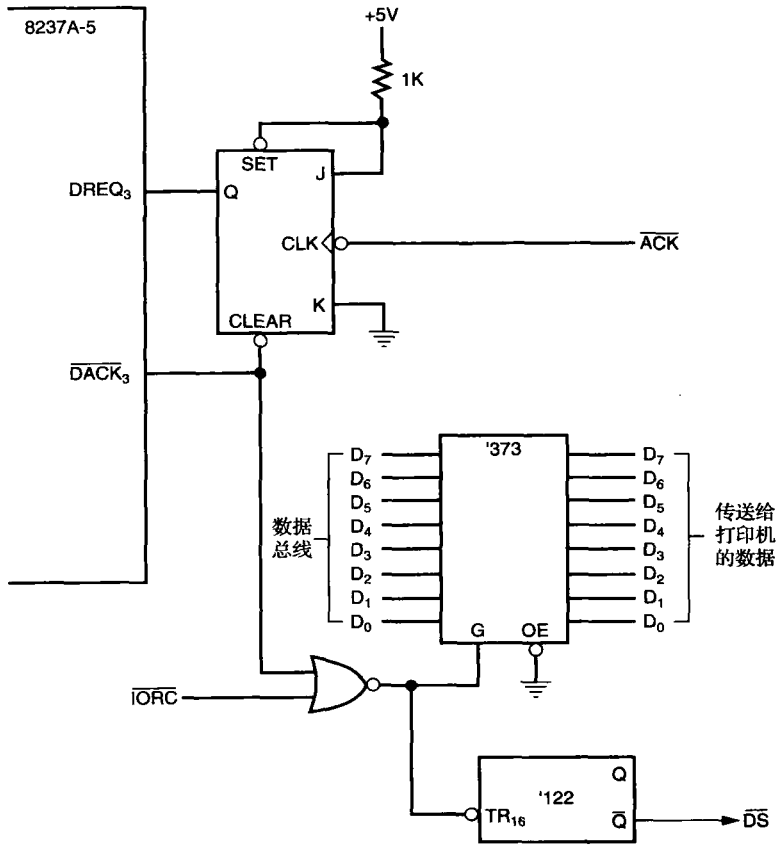


图 13-13 DMA 处理的打印机接口

```
;       CX = 要打印的字节数

LATCHB  EQU     10H
CLEARF  EQU     7CH
CH3A    EQU     76H
CH1C    EQU     77H
MODE    EQU     7BH
CMMD    EQU     78H
MASKS   EQU     7FH
REQ     EQU     79H

PRINT . PROC    NEAR USES AX CX BX

    MOV     EAX,0
    MOV     AX,DS               ;编程锁存器B
    SHR     EAX,4
    PUSH    AX
    SHR     EAX,16
    OUT     LATCHB,AL

    POP     AX                  ;编程地址
    OUT     CH3A,AL
    MOV     AL,AH
    OUT     CH3A,AL

    MOV     AX,CX               ;编程计数
    DEC     AX
    OUT     CH3C,AL
```

```
        MOV     AL,AH
        OUT     CH3C,AL

        MOV     AL,0BH              ;编程模式
        OUT     MODE,AL

        MOV     AL,00H              ;允许块模式传送
        OUT     CMMD,AL

        MOV     AL,7                ;允许通道3
        OUT     MASKS,AL
        RET

PRINT   ENDP
```

另外，还需要第二个过程以确定DMA操作是否完成。例13-4列出了测试DMA控制器以确定DMA传输是否完成的这个过程。在编程DMA控制器之前需调用TESTP过程以检查前一次传输是否完成。

例13-4

```
;该过程测试DMA操作是否完成

STATUS  EQU   78H

TESTP   PROC  NEAR USES AX

        .REPEAT
              IN    AL,STATUS
        .UNTIL AL & 8
        RET

TESTP   ENDP
```

被打印的数据可以经过双重缓冲，首先将要打印的数据装入缓冲区1，然后调用PRINT过程开始打印缓冲区1中的数据。由于编程DMA控制器只占用很少时间，所以在通过打印机接口和DMA控制器打印第1个缓冲区（缓冲区1）中的数据时，第2个缓冲区（缓冲区2）可用新的打印数据来填充。重复该过程直到所有数据打印完毕。

13.3 共享总线操作

现在复杂的计算机系统有许多任务要完成，因此在一些系统中使用不止一个微处理器来完成这些工作。这样的系统称为**多处理**（**multiprocessing**）系统，有时也称为**分布式**（**distributed**）系统。完成不止一个任务的系统被称为**多任务**（**multitasking**）系统。在多处理系统中，必须设计和使用一些控制方法。在一个分布式、多处理器、多任务的环境中，每个微处理器访问2种总线：1）**局部总线**（**local bus**）；2）**远程总线**（**remote bus**）或**共享总线**（**shared bus**）。

本节描述了8086和8088微处理器使用8289总线仲裁器的共享总线操作。80286使用82289总线仲裁器，80386/80486使用82389总线仲裁器。Pentium～Pentium 4直接支持多用户环境，正如第16章至第18章中介绍的那样。这些系统过于复杂，这里难以详述，但它们的术语和操作本质上与8086/8088相同。

局部总线与存储器和I/O设备相连，它们由单个微处理器直接访问，无须任何特殊协议或访问规则。远程（共享）总线包含被系统中任意微处理器访问的存储器和I/O设备。图13-14用几个微处理器描述了这一概念。注意PC机也是按与图13-14中系统同样的方式配置的。PC机中的主微处理器是总线主控设备。PC机中的局部总线在本图中为共享总线，ISA总线是作为PC机微处理器的从属设备工作的，连到共享总线上的任何其他设备也是如此。PCI总线可作为从设备或者作为主控设备。

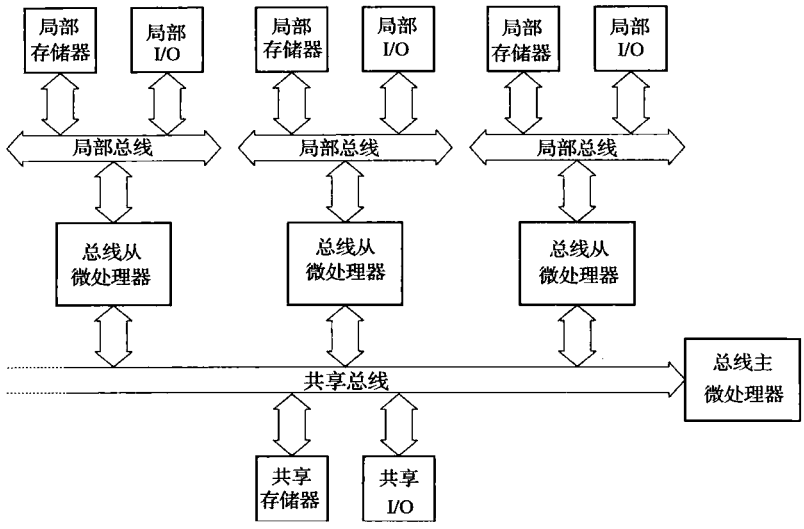


图 13-14 共享总线和局部总线的框图

13.3.1 定义的总线类型

局部总线是微处理器常驻的总线，它包含驻留的或局部的存储器和 I/O 设备。迄今为止本书所研究的所有微处理器均被认为是局部总线系统。局部存储器和局部 I/O 设备可被直接与它们相连的微处理器访问。

共享总线是与系统中所有微处理器相连的总线，用于在系统中的微处理器之间交换数据。共享总线可包含被系统中所有微处理器访问的存储器和 I/O 设备。对共享总线的访问由总线仲裁器控制，总线仲裁器只允许单个微处理器访问系统的共享总线空间。正如前面提及的，PC 机中的共享总线常在 PC 机中被称为局部总线，因为它对 PC 机中的微处理器而言是局部的。

图 13-15 表示一个 8088 微处理器被连接作为远程总线主控设备。术语**总线主控设备**（**bus master**）是指那些设备（微处理器或其他设备），它们可控制包括存储器和 I/O 设备的总线。本章前面提到的 8237 DMA 控制器就是远程总线主控设备的一个例子。DMA 控制器获得对系统存储器和 I/O 设备的访问权，以进行数据传输。同样，远程总线主控设备出于同样的目的获得对共享总线的访问权。不同的是，远程总线主控设备微处理器可执行不同的软件，而 DMA 控制器只能传输数据。

对于 DMA 控制器，对共享总线的访问是通过使用微处理器上的 HOLD 引脚来实现的；而对于远程总线主控设备，访问共享总线是通过**总线仲裁器**（**bus arbiter**）实现的，总线仲裁器起决定总线主控设备之间的优先权的作用，它一次只允许一个设备访问共享总线。

注意，在图 13-15 中，8088 微处理器有一个与局部（即驻留）总线和共享总线都相连的接口。这种配置允许 8088 访问局部存储器和 I/O 设备，或通过总线仲裁器和缓冲器访问共享总线。分配给微处理器的任务可能是数据通信，在从通信接口采集了一批数据后，微处理器可将它们传送给共享总线和共享存储器，使得连接到该系统的其他微处理器也可访问这些数据。这样就允许许多微处理器共享公共数据。以同样的方式，多个微处理器在系统中也可分配不同的任务，从而彻底提高吞吐量。

13.3.2 总线仲裁器

在完全理解图 13-15 之前，必须掌握总线仲裁器的操作。8289 总线仲裁器控制总线主控设备与共享总线的接口。尽管 8289 不是惟一的总线仲裁器，但它被设计与 8086/8088 一起工作，所以这里对它加以介绍。每个总线主控设备或微处理器都需要一个总线仲裁器与共享总线接口，Intel 称共享总线为**多总线**（**Multi bus**），IBM 称之为**微通道**（**Micro Channel**）。

共享总线只用于将信息从一个微处理器传送到另一微处理器，而总线主控设备通过使用它们各自

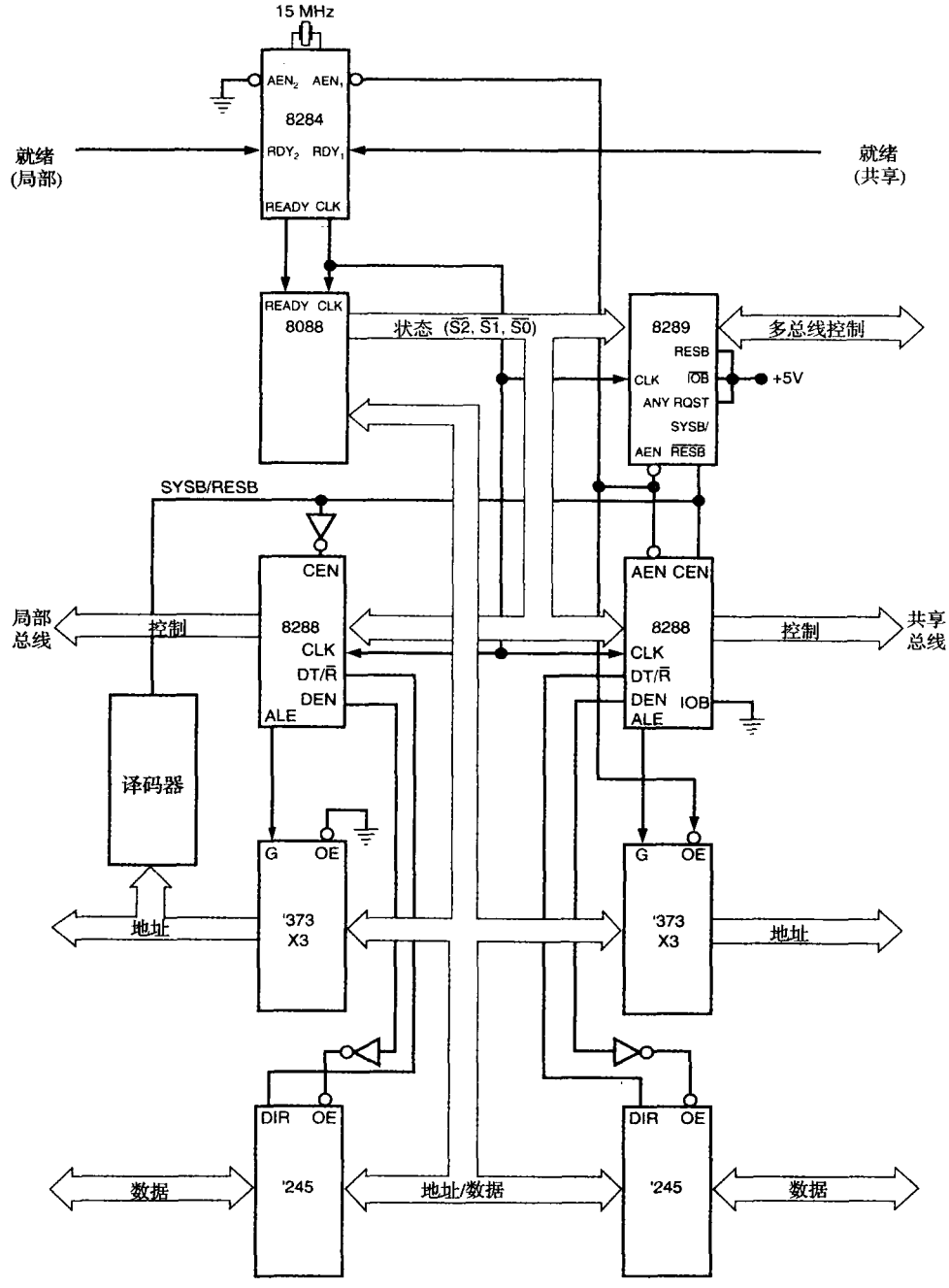


图 13-15 工作在远程模式下的 8088，描述了局部总线和共享总线的连接

的局部程序、存储器以及 I/O 空间，在它们各自的局部总线模式中起作用。连接在这种系统中的微处理器常称为**并行**（**parallel**）或**分布式**（**distributed**）处理器，因为它们可并行执行软件和完成任务。

8289 结构

图 13-16 给出了 8289 总线仲裁器的引脚图和框图。框图左边描述了对微处理器的连接，右边表示了与共享总线或多总线的连接。

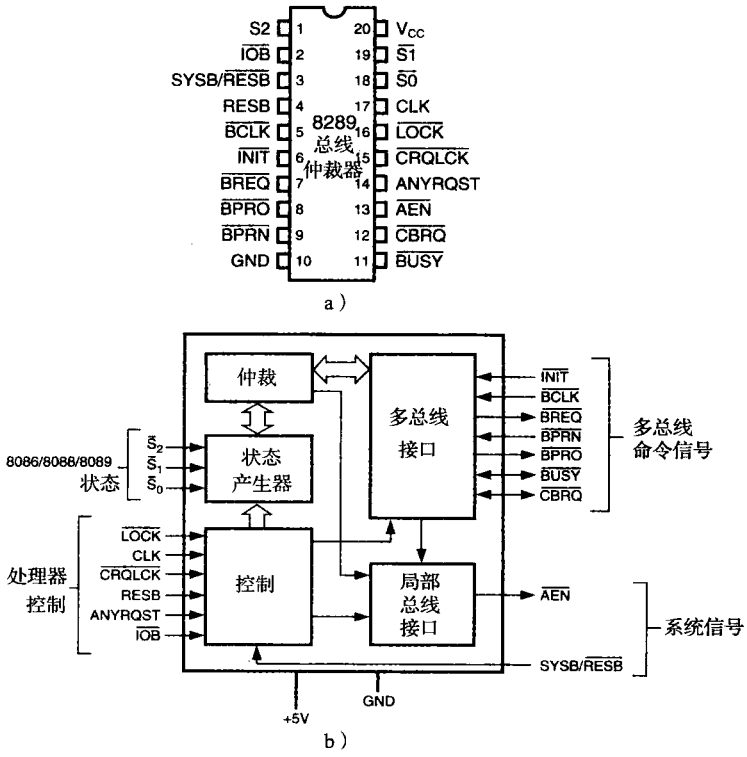


图 13-16 8289 的引脚输出和框图

在微处理器访问共享总线被拒绝时，8289 控制器通过使微处理器的 READY 输入变为逻辑 0（未就绪）来控制共享总线。一旦另一微处理器正在访问共享总线，就会出现**阻塞**（**blocking**）。结果是，微处理器访问共享总线的请求被加在其 READY 输入上的逻辑 0 所阻塞。当 READY 引脚为逻辑 0 时，微处理器及其软件一直等待，直到访问共享总线的请求被仲裁器允许。在这种方式下，一次只有一个微处理器获得对共享总线的访问权。8289 总线仲裁器无须特殊的指令进行总线仲裁，因为仲裁完全由硬件完成。

引脚定义

AEN **地址使能**（**address enable**）输出使系统中的总线驱动器转换到第三态，即高阻抗状态。

ANYRQST **任意请求**（**any request**）输入是一个跳线选择，防止较低优先级的微处理器获得对共享总线的访问权。如果连到逻辑 0 上，则发生正常仲裁，如果此时CBRQ也是逻辑 0，则较低优先级的微处理器可获得对共享总线的访问权。

BCLK **总线时钟**（**bus clock**）输入使所有共享总线主控设备同步。

BPEN **总线优先级输入**（**bus priority input**）允许 8289 在BLCK信号的下一个下降沿获得共享总线。

BPRO **总线优先级输出**（**bus priority output**）信号用于在一个包含多个总线主控设备的系统中决定优先级。

BREQ **总线请求输出**（**bus request output**）用于请求对共享总线的访问。

BUSY **忙输入/输出**（**busy input/output**），作为输出时表示 8289 已获得共享总线；作为输入时，用于检测另一 8289 是否已获得共享总线。

CBRQ **公共总线请求**（**common bus request**）输入/输出用在一个较低优先级微处理器正在请求使用共享总线时。作为输出信号时，一旦 8289 请求共享总线，则CBRQ变为逻辑 0，并

维持低电平，直到8289获得对共享总线的访问为止。

CLK **时钟（clock）**输入由8284A时钟产生器产生，给8289提供内部定时源。

$\overline{\text{CRQLCK}}$ **公共请求锁定（common request lock）**输入防止8289将共享总线交给系统中其他任一8289。该信号与$\overline{\text{CBRQ}}$引脚一起工作。

$\overline{\text{INIT}}$ **初始化（initialization）**输入复位8289，通常与系统RESET信号相连。

$\overline{\text{IOB}}$ **I/O总线（I/O bus）**输入选择8289在共享总线中（如果由RESB选择）是与I/O设备（$\overline{\text{IOB}}$ = 0）一起工作，还是与存储器和I/O设备（$\overline{\text{IOB}}$ = 1）一起工作。

$\overline{\text{LOCK}}$ **锁定（lock）**输入防止8289允许任一其他微处理器获得对共享总线的访问权。包含一个LOCK前缀的8086/8088指令将防止其他微处理器访问共享总线。

RESB **驻留总线（resident-bus）**输入是一个跳线连接，它允许8289在有共享总线的系统或驻留总线系统中工作。如果RESB为逻辑1，则8289被配置为共享总线主控设备；如果RESB为逻辑0，则8289被配置为局部总线主控设备。当被配置为共享总线主控设备时，需要通过SYSB/$\overline{\text{RESB}}$输入引脚访问共享总线。

S_0、S_1和S_2 **状态（status）**输入初始化共享总线的请求和交还。这些引脚与8288系统总线控制器的状态引脚相连。

SYSB/$\overline{\text{RESB}}$ **系统总线/驻留总线（system bus/resident bus）**输入在置为逻辑1时选择共享总线系统，在置为逻辑0时选择驻留局部总线。

通用8289操作

正如引脚描述所介绍的，8289可在3种基本模式下操作：1）I/O外围总线模式；2）驻留总线模式；3）单总线模式。参见表13-2，它描述了8289工作在这3种模式下所需的引脚连接。在**I/O外围总线模式（I/O peripheral bus mode）**中，局部总线上的所有设备（包括存储器）均被看做I/O设备，并由I/O指令访问。所有存储器访问共享总线，而所有I/O设备访问驻留局部总线。**驻留总线模式（resident bus mode）**允许存储器和I/O设备访问局部总线和共享总线。最后，**单总线模式（single-bus mode）**将微处理器连接到共享总线上，但微处理器没有局部存储器或局部I/O设备。在许多系统中，只设置一个微处理器作为共享总线主控设备（单总线模式）来控制共享总线，并变为共享总线主控设备。共享总线主控设备（shared-bus master）通过共享存储器和I/O设备来控制系统。其他微处理器被接到共享总线上，作为驻留或I/O外围总线主控设备。这些另外的总线主控设备一般执行独立的任务，这些任务是通过共享总线报告给共享总线主控设备的。

表13-2 8289操作模式

模 式	引脚链接
单总线	$\overline{\text{IOB}}$ = 1和RESB = 0
驻留总线	$\overline{\text{IOB}}$ = 1和RESB = 1
I/O总线	$\overline{\text{IOB}}$ = 0和RESB = 0
I/O总线和驻留总线	$\overline{\text{IOB}}$ = 0和RESB = 1

描述单总线和驻留总线连接的系统

单总线操作将微处理器连接到共享总线上，这个共享总线包含被其他微处理器共享的I/O和存储器资源。图13-17给出了3个8088微处理器，每个均连接到共享总线上。其中2个微处理器工作在驻留总线模式，而第3个微处理器工作在单总线模式。图中微处理器A工作在单总线模式且没有局部总线。此微处理器只访问共享存储器和I/O空间。微处理器A常被称为系统总线主控设备（system bus master），因为它负责协调主存储器和I/O任务。其余2个微处理器（B和C）被连接成驻留总线模式，允许它们访问共享总线及各自的局部总线。这些驻留总线微处理器用于执行独立于系统总线主控设备的任务。事实上，系统总线主控设备中断其正在执行的任务的惟一时刻，是当2个驻留总线微处理器中的一个需要在它自己和共享总线之间传输数据时。这种连接允许所有3个微处理器同时执行任务，然而在需要时，数据可在微处理器之间共享。

在图13-17中，总线主控设备（A）允许用户用一个视频终端操作，该视频终端允许程序的执行和对系统总体的控制。微处理器B处理所有电话通信并将信息以块的形式传送给共享存储器。这意味着

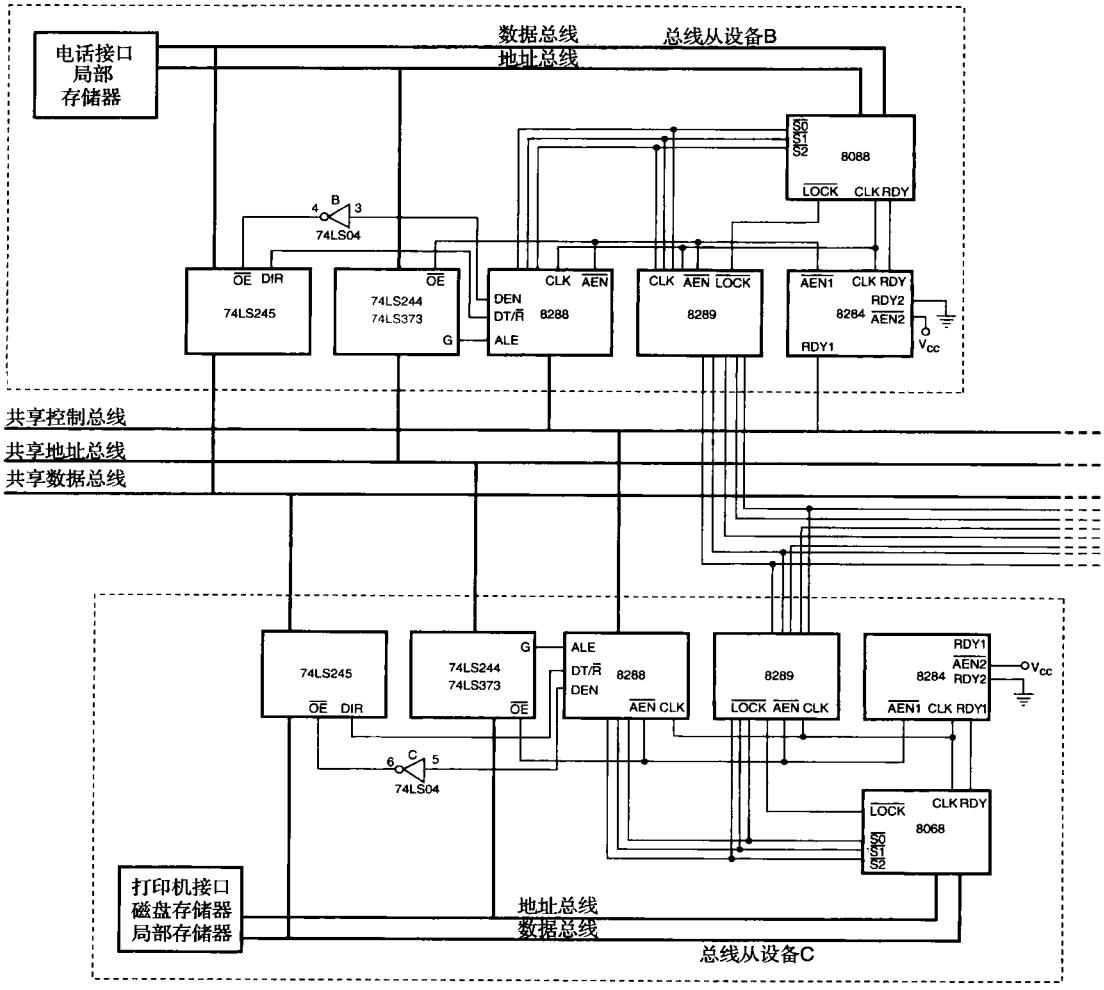


图 13-17 共享一个公共总线系统的 3 个 8088 微处理器

微处理器 B 等待每个字符的发送或接收，并控制用于传输的协议。例如，假设 1KB 的数据块通过电话接口以每秒 100 个字符的速率发送，这意味着整个传输需要 10 秒钟时间。但这并不需要占用总线主控设备 10 秒钟，而是微处理器 B 从其自己的局部存储器和局部通信接口执行数据传输。这样就使总线主控设备可以执行其他任务。微处理器 B 中断总线主控设备的惟一时刻，是在共享存储器和微处理器 B 的局部存储系统之间传输数据时。这种在微处理器 B 和总线主控设备之间的数据传输，只需要几百微秒时间。

微处理器 C 被用作一个打印假脱机系统，它的惟一任务是在打印机上打印数据。一旦总线主控设备需要打印输出，则它将任务传输给微处理器 C。然后微处理器 C 访问共享存储器，捕获要打印的数据，并将其存储在自己的局部存储器中。数据就从局部存储器中打印出来，从而释放总线主控设备去执行其他任务。这就允许该系统用总线主控设备执行程序，通过与微处理器 B 的通信接口传输数据，并用微处理器 C 在打印机上打印信息。这些任务均同时执行。使用这一技术对于连接到系统的微处理器的数目或同时执行任务的个数并没有限制，惟一的限制是系统设计和设计者的独创性。美国 California 州的 Lawrence Livermore 实验室拥有一个包含 4096 个 Pentium 处理器的系统。

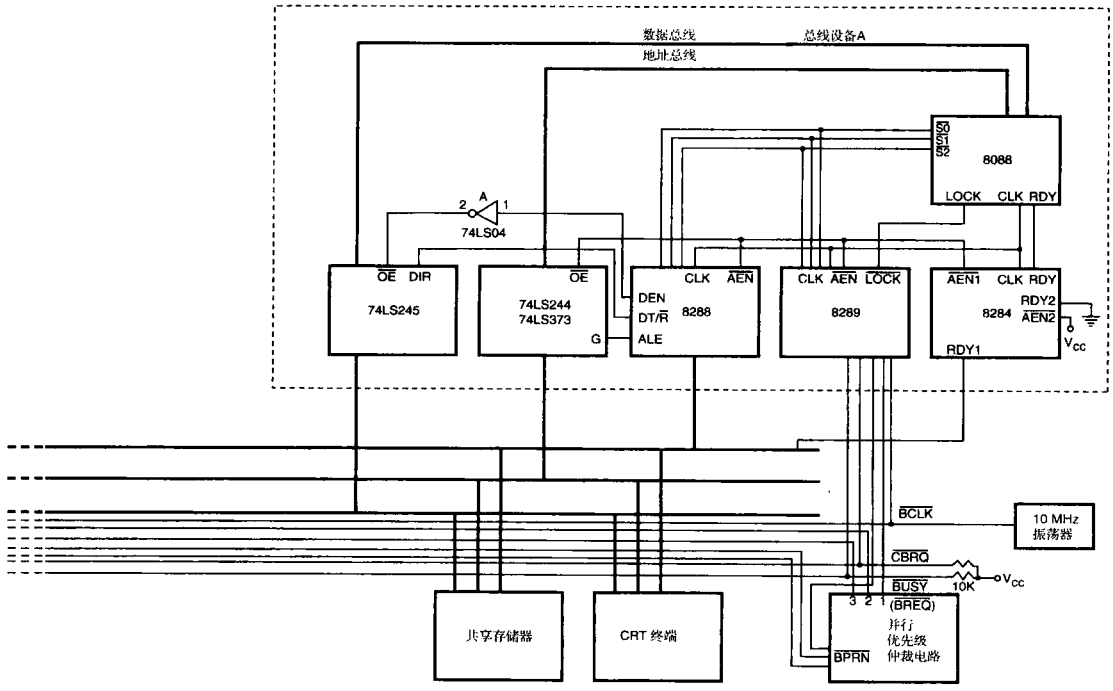


图 13-17（续）

注：1. 微处理器 A 为总线主控设备，控制共享存储器和 CRT 终端。
2. 微处理器 B 为总线从属设备，控制其局部电话接口和存储器。
3. 微处理器 C 也是总线从属设备，控制打印机、磁盘存储系统和局部存储器。

13.4 磁盘存储系统

磁盘存储器用于长期存储数据。现在有许多类型的磁盘存储系统，它们使用磁性介质，除了光盘存储器是在塑料盘片上存储数据之外。光盘存储器可以是 **CD-ROM**（**compact disk/read only memory，压缩盘-只读存储器**），它只能读出不能写入，也可指 **WORM**（**write once/read many，写一次/读多次存储器**），它大多数时候是只读的，但可用激光光束写一次。可读写多次的光盘存储器也已问世，但允许写操作的次数仍然有限。最新的光盘技术被称为 **DVD**（**digital versatile disk，数字通用磁盘**）。DVD（8.5GB）是用于高清晰度的视频和数据存储的版本例如蓝光（50GB）和 HD-DVD（30GB）。本节介绍用于计算机系统中的磁盘存储系统，还详细介绍了它们的操作。

13.4.1 软盘存储器

磁盘存储器最常见和最基本的形式是软盘。如今软盘逐渐消失，也许很快就彻底不见了，人们更喜欢 USB 笔式驱动器。这种磁性记录介质有 3 种尺寸：8 英寸**标准软盘**（**8″standard**），5.25 英寸**小型软盘**（**5.25″mini-floppy**），以及 3.5 英寸**微型软盘**（**3.5″micro-floppy**）。现在，8 英寸标准软盘已经消失了，而让位于小型软盘和微型软盘。8 英寸磁盘太大，难以处理和存放。为解决这一问题，开发商研制出 5.25 英寸小型软盘。现在，微型软盘在较新系统中已取代了小型软盘，因为它尺寸缩小了，易于存储，而且经久耐用。尽管如此，但少数系统仍然同时具有小型软盘和微型软盘的驱动装置。

所有磁盘有几个共同点。它们都是有组织的，使数据存储在磁道中。一个**磁道**（**track**）是在磁盘表面存储数据的一个同心圆。图 13-18 给出了 5.25 英寸小型软盘的表面，可以看出磁道被划分成许多扇区。一个**扇区**（**sector**）是一个磁道的一部分，被设计用来保存合理数量的数据。在许多系统中，一个扇区保存 512 或 1024 个字节数据。扇区的大小可以不同，可从 128 字节到整个磁道的长度。

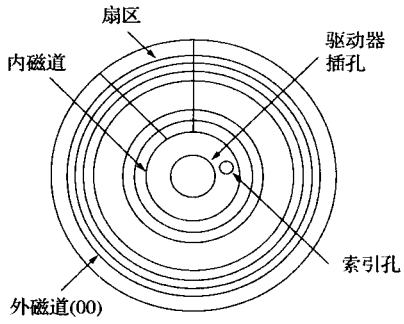


图 13-18 5.25 英寸软磁盘的格式

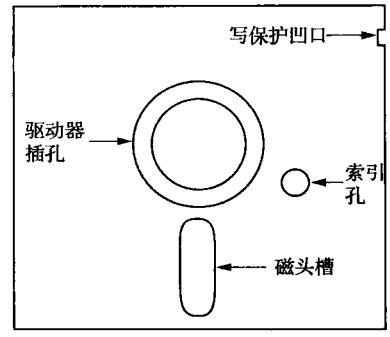


图 13-19 5.25 英寸小型软盘

注意，图 13-18 中磁盘上有一个孔被标识为索引孔。**索引孔**（**index hole**）设计用于使读盘的电子系统可以找到磁盘的起始位置及它的第一个扇区（00）。磁道被编号为从磁道 00（即最外面的磁道）开始，越靠近中心或最里面的磁道，编号越大。扇区常被编号为从最外面磁道上的扇区 00 开始，一直到最里面的磁道和最后一个扇区为止。

5.25 英寸小型软盘

现在，5.25 英寸软盘可能是较老的微处理器系统中使用得比较多的磁盘尺寸。图 13-19 给出了这种小型软盘。软盘在其半硬的塑料套中以 300r/m 的速度旋转。软盘驱动器中的磁头机械装置使磁头和磁盘表面有物理接触，最终将引起磁盘的磨损和损坏。

现在，大多数小型软盘为双面盘。这意味着数据被写在磁盘的正面和反面。一组磁道被称为一个**柱面**（**cylinder**），它由一个正面磁道和一个反面磁道组成。例如，柱面 00 由最外面的正面和反面磁道组成。

软盘数据以双密度格式存储，它使用称为 **MFM**（**modified frequency modulation，改进调频制**）的记录技术来存储信息。**双面、双密度**（**double-sided，double-density，DSDD**）磁盘通常被组织成磁盘每面存储 40 个磁道的数据。典型双密度磁盘被分为 9 个扇区，每个扇区包含 512 字节信息。这意味着一个双密度、双面磁盘的总容量为 40 磁道/面 ×2 面 ×9 扇区/磁道 ×512 字节/扇区，即 368 640 字节（360KB）信息。

现在还有一种常见的**高密度**（**high-density，HD**）小型软盘，它每面包含 80 个磁道的信息，每个磁道有 8 个扇区，每个扇区包含 1024 字节信息。因此，5.25 英寸高密度、小型软盘的总容量为 80 磁道/面 ×2 面 ×15 扇区/磁道 ×512 字节/扇区，即 1 228 800 字节（大约 1.2MB）信息。

用于在磁盘表面存储数据的磁性记录技术被称为**不归零制**（**non-return to zero，NRZ**）记录。使用 NRZ 记录，加在磁盘表面的磁通量决不会回到零。图 13-20 给出了存储在一部分磁道上的信息，它还显示了磁场是如何编码数据的。注意图中的箭头表示存储在磁盘表面上的磁场的极性。

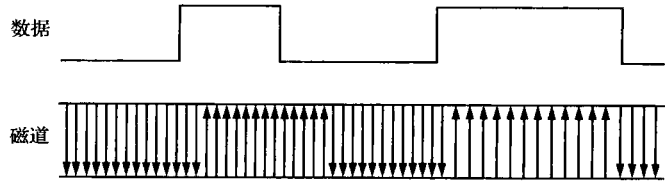


图 13-20 不归零制（NRZ）记录技术

选择这种形式的磁编码的主要原因是，在记录新的信息时可自动擦除旧的信息。如果使用其他技术，则需要一个独立的擦除磁头。独立的擦除磁头与独立的读/写磁头的机械定位事实上是不可能的。NRZ 信号的磁通量密度太大，以至于完全磁化了磁盘表面，从而擦除了以前的数据。它还确保噪声不会影响到信息，因为磁场的幅度不表示信息。信息存储在变化的磁场中。

在现代软磁盘系统中，数据以 MFM 的形式存储。MFM 记录技术以图 13-21 所示的形式存储数据。

注意，双密度磁盘上每位时间为2.0 μs宽，这意味着以每秒500 000位的速率记录数据。每个2.0 μs位时间被分为两部分：指定一部分保持一个时钟脉冲，另一部分保持一个数据脉冲。如果时钟脉冲出现，则它是1 μs宽，数据脉冲也是1 μs宽。时钟脉冲和数据脉冲在一个位周期中决不会在同一时刻出现（注意，高密度磁盘驱动器只需要这些时间的一半，所以位时间为1.0 μs，时钟脉冲或数据脉冲为0.5 μs宽，这也使传输速率提高了一倍，达到每秒1百万位，即1Mb/s）。

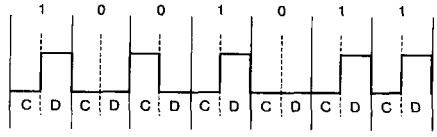


图13-21 用于磁盘存储器的改进调频制（MFM）

如果数据脉冲出现，则位时间表现为逻辑1；如果没有数据或时钟出现，则位时间表现为逻辑0；如果时钟脉冲出现而没有数据脉冲出现，则位时间仍表现为逻辑0。使用MFM存储数据的规则如下：

1）对于一个逻辑1，总是存储一个数据脉冲。

2）在一个逻辑0串中，第1个逻辑0不存储数据和时钟。

3）第2个和后来的一行逻辑0包含一个时钟脉冲，但没有数据脉冲。

一个时钟被插入第2个和后来的一行逻辑0中的原因是为了在数据从磁盘读出时维持同步。用于从磁盘驱动器中取回数据的电子设备，使用一个锁相环来产生一个时钟和一个数据窗口。锁相环需要一个时钟或数据来维持同步操作。

3.5英寸微型软盘

一种非常流行的磁盘尺寸是3.5英寸微型软盘。近来这种尺寸的软盘已经开始被在传播媒体中占统治地位的笔式USB所取代。微型软盘是前面介绍的小型软盘的改进型。图13-22给出了3.5英寸微型软盘的外形。

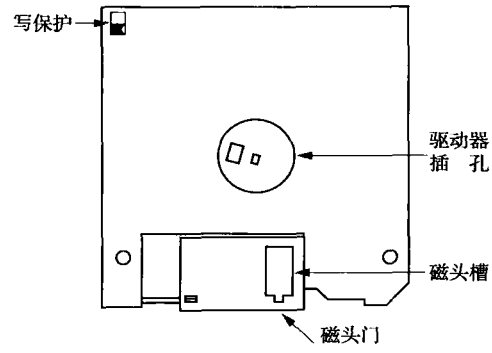


图13-22 3.5英寸微型软盘

在作为8英寸标准软盘按比例缩小版本的小型软盘推出后不久，磁盘设计者就注意到它的一些缺点。小型软盘的最大问题在于它被封装在一个易弯曲的半硬塑料套中，而微型软盘被封装在一个不易弯曲的硬塑料套中，这给套中的磁盘提供了很大程度的保护作用。

小型软盘的另一问题是，磁头凹槽一直将磁盘表面暴露在污染物中。这一问题在微型软盘中也得到了解决，因为微型软盘的结构有一个带弹簧的滑动磁头门。磁头门一直是关闭的，直到磁盘插入驱动器时才打开。一旦插入驱动器，驱动器机械装置就滑开磁头门，从而将磁盘表面暴露在读/写磁头下。这样就大大保护了微型软盘的表面。

另一个改进是微型软盘上的滑动塑料写保护装置。在小型软盘上，一片胶带被粘在塑料套的凹口处以防止写操作。这个塑料胶带在磁盘驱动器中很容易被弄掉，从而带来问题。在微型软盘上，一个集成的塑料滑块取代了胶带写保护装置。为写保护（防止写）微型软盘，移动塑料滑块，打开盘套上的一个小孔，使光照到一个传感器上，从而禁止了写操作。

还有一个改进是用一个不同的驱动装置取代了索引孔。小型软盘上的驱动装置允许磁盘驱动器在任意一点抓住磁盘，这就需要一个索引孔，使电子设备可以找到磁道的起始位置。索引孔会造成另一个麻烦，因为它会聚集污垢和灰尘。微型软盘是上栓的驱动装置，因此它在磁盘驱动器内部只适于一种方式。于是这种上栓的驱动装置，就不再需要索引孔了。由于有滑动磁头装置以及不存在索引孔，所以微型软盘没有地方会聚集灰尘或污垢。

有两种类型的微型软盘被广泛应用：双面双密度（DSDD）和高密度（HD）软盘。双面双密度微型软盘每面有80个磁道，每个磁道包含9个扇区，每个扇区包含512字节的信息。这样就允许80磁道/面×2面×9扇区/磁道×512字节/扇区，即737 280字节（720KB）数据存储在一个双密度双面的

微型软盘上。

高密度双面微型软盘可存储更多的信息。高密度版本每面有 80 个磁道，但每个磁道包含 18 个扇区，这种格式仍然是每个扇区包含 512 字节信息，正如双密度格式一样。高密度双面微型软盘上的字节总数为 80 磁道/面 ×2 面 ×18 扇区/磁道 ×512 字节/扇区，即 1 474 560 字节（1.44MB）信息。

13.4.2 笔式驱动器

笔式驱动器通常也被称为闪存（flash drive）。它使用 Flash memory 来存储数据，是软盘的替代品。尽管闪存没有磁道和扇区，但是作为 Windows（除了 Windows 98）一部分的驱动程序，使闪存就像软盘一样具有磁道和扇区。和软盘一样，FAT 系统用于文件结构。采用这种存储类型的驱动器是串行存储器。当闪存连接在 USB 总线上时，操作系统会识别它，并且允许数据在闪存与计算机之间传输。

使用 USB 2.0 总线规范，新型闪存的传输速率要远远大于 USB1.1 规范的速率。USB 1.1 的读速率是 750KB/s，写速率是 450KB/s。USB 2.0 闪存的传输速率可以达到约 48MB/s。目前，闪存的存储量可以达到 1GB，擦除次数可以高达 1 000 000 次，其价格相比于软盘也更合理。

13.4.3 硬盘存储器

更大的磁盘存储器是**硬盘驱动器**（**hard disk drive**）。硬盘驱动器常被称为**固定磁盘**（**fixed disk**），因为它不像软磁盘那样是可移动的。硬盘也常被称为**硬磁盘**（**rigid disk**）。术语**温彻斯特驱动器**（**Winchester drive**）也被用于描述硬盘驱动器，但现在已不很常见了。硬盘存储器的容量比软盘存储器大得多，现有的硬盘存储器的容量超过了 1TB 数据。普遍的、低廉（每 MB 小于 1 美元）的硬盘存储器容量为 20GB ～ 300GB。

在软盘和硬盘存储器之间有几个区别。硬盘存储器使用浮动磁头存储和读取磁盘表面的数据。浮动磁头非常小而轻，它不接触磁盘表面。它在一层空气薄膜表面的上方飞行，这层空气薄膜在磁盘旋转时在磁盘的表面产生。硬盘典型的旋转速度为 3000 ～ 15 000RPM，比软盘速度快许多倍。这一较高的旋转速度允许磁头在磁盘表面上方飞行（正如飞机飞行一样）。这是一个非常重要的特性，因为硬盘表面没有磨损，而软盘则不同。

浮动磁头也会产生问题。其中一个问题是磁头碰撞。如果电源突然中断或硬盘驱动器受到振动，则磁头会撞上磁盘表面，这将损坏磁盘表面或磁头。为防止碰撞，一些驱动器制造商在驱动器中包括了一个当电源中断时可自动停泊磁头的系统。这种磁盘驱动器有自动停泊磁头。当磁头停泊时，它们被移到一个安全着陆区（未用的磁道），此时电源是切断的。一些驱动器不是自动停泊的，它们通常需要一个程序在电源切断前将磁头停泊在最里面的磁道上。最里面的磁道是安全着陆区，因为它是最后被磁盘驱动器填充的磁道。在这种磁盘驱动器中停泊磁头是操作者的责任。

在软盘驱动器和硬盘驱动器之间的另一区别是磁头与磁盘表面的数目。软盘驱动器有 2 个磁头，一个用于上表面，另一个用于下表面。硬盘驱动器最多可以有 8 个磁盘表面（4 个磁盘片），每个表面最多有 2 个磁头。每次通过移动磁头组合来获得新的柱面，在磁头下有 16 个新的磁道。参见图 13-23 给出的硬盘系统。

磁头通过使用步进电机或音圈从一个磁道移动到另一个磁道。步进电机速度慢且有噪声，而音圈机械装置速度快且安静。在使用步进电机定位磁头的系统中，需要每个柱面一步来移动磁头组合。在使用音圈的系统中，一个扫描动作可使磁头移动许多个柱面。这使得磁盘驱动器在寻找新的柱面时，速度更快。

音圈系统的另一优点是，一个伺服机构可监视来自读磁头的信号幅度，并对磁头位置进行轻微的调整。对于步进电机这是不可能的，因为它严格依赖机械装置来定位磁头。步进电机型磁头定位机械装置在使用时常常变得失调，而音圈机械装置可纠正任何失调。

硬盘驱动器常在 512 字节长的扇区中存储信息。数据在 8 个或更多扇区的簇（cluster）中被寻址，在大多数硬盘驱动器中一个簇包含 4096 字节（或更多）信息。硬盘驱动器使用 MFM 或 RLL 存储信息。MFM 在软盘驱动器中已介绍过，这里介绍游程长度受限（run-length limited，RLL）驱动器。

一个典型的早期 MFM 硬盘驱动器使用每个磁道 18 个扇区，使得每个磁道可存储 18KB 的数据。如

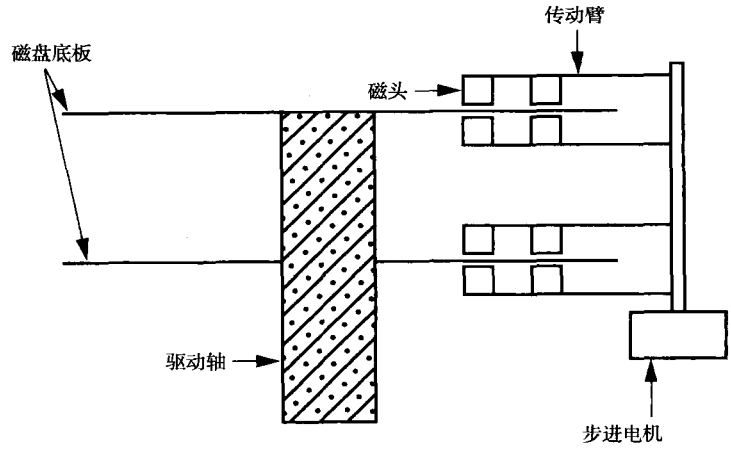


图 13-23 每个盘片使用4个磁头的硬盘驱动器

果一个硬盘驱动器的容量为40MB，则它包含大约2280个磁道。如果磁盘驱动器有2个磁头，这意味着它包含1140个柱面；如果包含4个磁头，则有570个柱面。这些规格因磁盘驱动器的不同而不同。

RLL 存储

游程长度受限（run-length limited，RLL）磁盘驱动器使用与MFM不同的方法编码数据。术语RLL意味着零的游程（一行中的零）是有限的。当今常用的RLL编码方案是RLL2，7。这意味着零的个数总是在2～7之间。表13-3给出了标准RLL的编码。

在数据被送到驱动器电子设备以存储到磁盘表面上之前，首先使用表13-3进行编码。由于采用这种编码技术，就有可能使磁盘驱动器上的数据存储量比MFM增加50%。主要区别在于RLL驱动器常常包含27个磁道，而MFM驱动器上只包含18个磁道（一些RLL驱动器还使用每磁道35个扇区）。

表 13-3 标准 RLL2，7 的编码

输入数据流	RLL 输出
000	000100
10	0100
010	100100
0010	00100100
11	1000
011	001000
0011	00001000

在大多数情况下，RLL编码不需要改变驱动器电子设备或磁盘表面。使用RLL的惟一区别是脉冲宽度的稍稍减小，它需要在磁盘表面上更细小的氧化物粒子。磁盘制造商们测试磁盘表面，将磁盘驱动器分级为MFM编码的或RLL编码的驱动器。除了分级以外，在磁盘驱动器结构和涂在磁盘表面的磁性材料方面没有区别。

图13-24给出了MFM数据和RLL数据的比较。注意，与MFM相比，存储RLL数据所需的时间（空间）减少了。这里一个101001011被编码为MFM和RLL形式，以便对这两种标准进行比较。注意，RLL信号的宽度减小了，使得3个脉冲的空间相当于MFM的一个时钟脉冲和一个数据脉冲。一个40MB的MFM磁盘可保存60MB的RLL编码数据。除了能保存更多信息外，RLL驱动器还可以以更高的速度进行读和写。

所有硬盘驱动器都使用MFM或RLL编码。当今正在使用的有许多磁盘驱动器接口。最早是ST-506接口，它使用MFM或RLL数据。使用此接口的磁盘系统也被称为MFM或RLL磁盘系统。现在又有较新的标准出现，包括ESDI、SCSI和IDE。所有这些较新标准均使用RLL，尽管它们一般都没注意到这一点。它们的主要区别在于计算机和磁盘驱动器之间的接口。IDE系统正在成为标准硬盘存储器接口。

增强型小磁盘接口（enhanced small disk interface，ESDI）系统已经消失，在它和计算机之间以大约每秒10MB的速度传输数据。ST-506接口可接近每秒860KB的传输速度。

小型计算机系统接口（small computer system interface，SCSI）系统也在使用中，因为它允许最

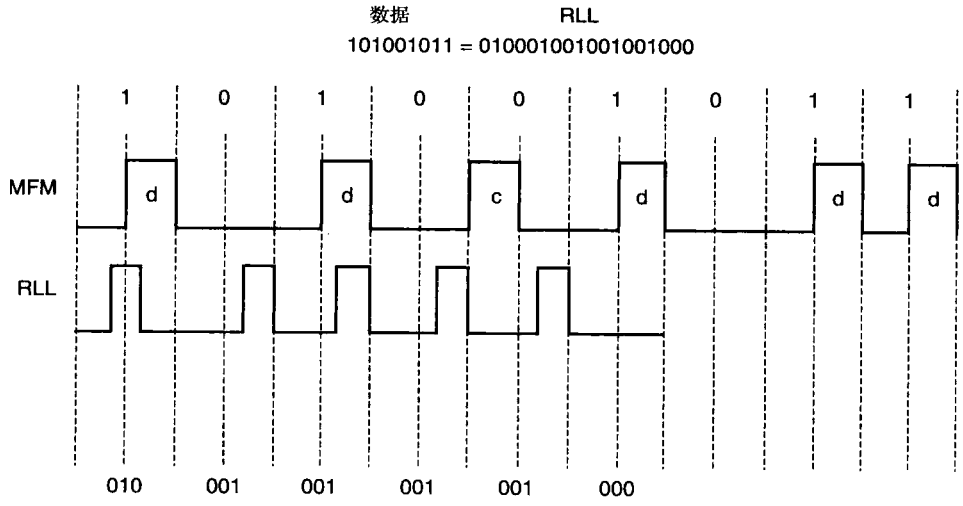


图 13-24 MFM 和 RLL 使用数据 101001011 的比较

多 7 个不同磁盘或其他接口通过同一接口控制器连到计算机上。SCSI 出现在一些 PC 型计算机以及 Apple Macintosh 系统中。一个改进版本即 SCSI-Ⅱ，已开始出现在一些系统中。将来，在大多数应用中此接口也许会被 IDE 所取代。

最新的系统是**集成驱动电子设备**（**integrated drive electronics，IDE**），它将磁盘控制器合并到磁盘驱动器中，并通过一条小接口电缆将磁盘驱动器接到宿主机系统上。这就允许许多磁盘驱动器接到一个系统上而无须担心总线冲突或控制器冲突。IDE 驱动器出现在较新的 IBM PS-2 系统中和许多兼容机中，甚至苹果机系统也开始用 IDE 驱动器取代早期苹果机中的 SCSI 驱动器。IDE 接口还能驱动除硬盘以外的其他 I/O 设备。该接口通常还包含至少一个 256KB ～ 2MB 的高速缓冲存储器供磁盘数据使用，高速缓冲存储器加速了磁盘数据的传输。IDE 驱动器的存取时间一般小于 8ms，而软盘的存取时间大约为 200ms。

IDE 有时也被称为 ATA，是 **AT attachment** 的缩写，此处的 AT 表示先进的计算机技术。最新的系统是串行 ATA 接口或 SATA，这种接口传输串行数据的速率可以达到 150MB/s（SATA2 是 300MB/s），比任何 IDE 接口都要快。还未发布的 SATA3 的速率可以达到 600MB/s。传输速率更高，这是因为逻辑 1 电平不再是 5.0V。对于 SATA 接口，逻辑 1 电平是 0.5V。由于信号升到 0.5V 的时间要少于升到 5V 的时间，所以数据的传输速率就会大大提高。这种硬盘接口的传输速率可达到 600MB/s，如 SATA3。

13.4.4 光盘存储器

光盘存储器（参见图 13-25）通常有两种形式：CD-ROM（compact disk/read only memory，压缩磁盘/只读存储器）和 WORM（write once/read many，写一次/读多次）。CD-ROM 是最便宜的光盘类型，但它速度不快。CD-ROM 的典型存取时间为 300ms 或更长，大致与软盘一样（注意，较慢的CD-ROM 设备在市场上还有，不推荐购买）。硬盘磁性存储器的存取时间可小到 11ms。一个 CD-ROM 存储 660MB 的数据，或数据与音乐通道的组合。随着系统的不断开发并使之更具可视性、灵活性，CD-ROM驱动器的使用将更为普及。

WORM 驱动器比 CD-ROM 更看好商业应用。由于 WORM 本身特性的缘故其应用在特定领域。由于数据只可以写入一次，所以主要应用在银行业、保险业以及其他大型的数据存储机构中。通常用 WORM 形成一个事务处理的审计跟踪文件并将其存于 WORM 中，只在审计期间才被检索。因此可以称 WORM 为存档设备。

许多 WORM 和读/写光盘存储系统，通过使用硬盘存储器所用的 SCSI 或 ESDI 接口标准来接到微处理器上。区别在于目前的光盘驱动器并不比大多数软盘驱动器快。一些 CD-ROM 驱动器通过与其他

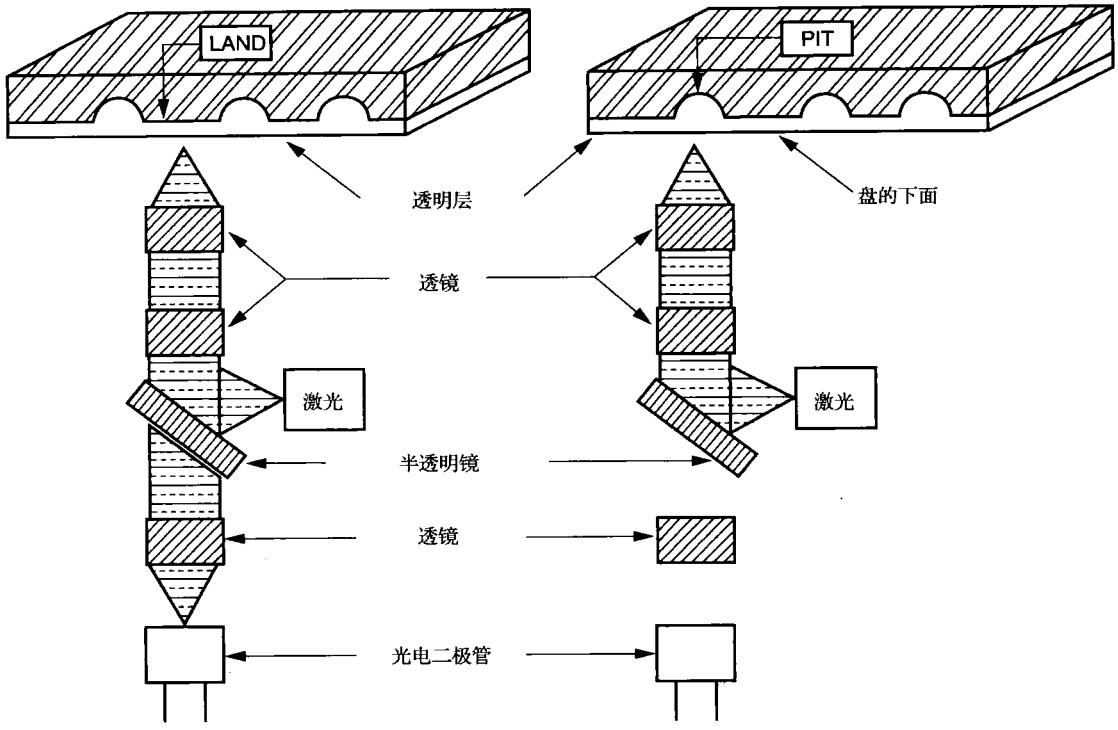


图 13-25 CD-ROM 光盘存储系统

磁盘驱动器中不兼容的专有接口接到微处理器上。

光盘的主要优点是它的耐久性。由于使用一束固态激光光束从磁盘中读取数据，而且聚焦点在保护性塑料涂层下，所以磁盘表面即使有小的划痕和尘粒，仍能正确读出。这一特性允许不必像对待软盘那样小心地保护光盘，破坏光盘上数据的惟一方式大概是折断光盘或深深地划伤它。

读/写 CD-ROM 驱动器已经出现，而且其价格正迅速下降。在不久的将来，应该可以看到读/写 CD-ROM 驱动器取代软盘驱动器。读/写 CD-ROM 的主要优点在于其巨大的存储容量。不久，这种格式将会改变，将可以做到保存许多 GB 的数据。一种称为 DVD 的新型通用读/写 CD-ROM 在 1996 年末 1997 年初出现。DVD 与 CD-ROM 功能大致相同，只是位密度更高一些。CD-ROM 存储 660MB 的数据，而当前类型的 DVD 存储 4.7GB 或 9.4GB 的数据，这取决于使用的标准。我们期待 DVD 最终将完全取代 CD-ROM 格式，起码是用于计算机数据存储，而不是用作音响。

采用这种技术的新产品主要有 Sony 公司的蓝光 DVD 和 Toshiba 公司的 HD-DVD，它们容量大小分别是 50GB 和 30GB。至于哪种格式将最终成为标准尚存争议，但不管怎样，获益最大的就是视频了，因为高分辨率的 HD 视频（1080p）既可以存储在蓝光 DVD 上也可以存储在 HD-DVD 上。然而传闻将来会出现更高分辨率的视频标准，甚至蓝光 DVD 和 HD-DVD 也可能会被某些其他的技术所取代。早期 DVD 与新技术之间的主要变化恐怕就是从红色激光到蓝色激光的转变了。蓝色激光频率较高，这就意味着它从 DVD 中每秒能读取更多的信息，因此有较高的存储密度。

13.5 视频显示器

现代视频显示器是 **OEM（original equipment manufacturer，原始设备制造商）**设备，它们通常被买来合并在一个系统里。现在，有许多不同类型的视频显示器，在这些显示器中，有彩色和单色两种类型。

单色显示器通常使用琥珀色、黄色或纸白色显示信息。纸白色显示器在许多应用中正变得极为普遍。最常见的应用是桌面出版和计算机辅助绘图（computer-aided drafting，CAD）。

彩色显示器则大不相同。彩色显示器系统可用来接收复合视频信号，就像家用电视机一样，还可接收TTL电压电平信号（0V或5V）以及模拟信号（0～0.7V）。复合视频显示器正在逐步消失，因为其分辨率太低。现在，许多应用需要高分辨率图形，不能在诸如家用电视接收机的复合显示器上显示。早期的复合视频显示器出现在Commodore 64、Apple 2以及类似的计算机系统中。

13.5.1 视频信号

图13-26给出了发送给显示器的复合视频信号。此信号由这种类型显示器所需要的几个部分组成。图中信号表示发送给彩色复合视频显示器的信号。注意，这些信号不仅包括视频信号，还包括同步脉冲、同步消隐脉冲电平和颜色脉冲。这里未给出音频信号，因为经常不存在音频信号。复合视频信号中不包括音频信号，音频信号是在计算机内产生并从计算机机箱内部的扬声器输出。音频信号还可由音响系统产生，并以立体声向外部扬声器输出。复合视频显示器的主要缺点是分辨率和颜色的限制。复合视频信号被设计用来仿真电视视频信号，使得家用电视接收机可作为视频显示器工作。

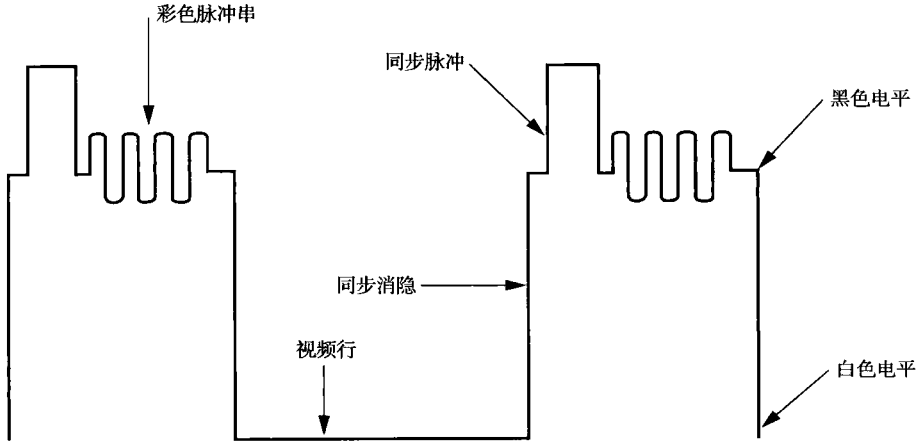


图13-26 复合视频信号

大多数现代视频系统使用直接的视频信号，视频信号与独立的同步信号一起产生。在直接视频系统中，视频信息通过一条电缆传送给显示器，此电缆对视频信号和同步脉冲分别使用单独的线。回想一下，这些信号组合起来就是一个复合视频信号。

单色（一种颜色）显示器使用一条线传送视频信号，一条线传送水平同步信号，另一条线用于传送垂直同步信号。这些信号线是最常见的。彩色视频显示器使用3种视频信号，分别表示红、绿、蓝。这样的显示器常称为RGB显示器，其视频光的原色为红（R）、绿（G）、蓝（B）。

13.5.2 TTL RGB显示器

RGB显示器既可以是模拟显示器，也可以是TTL显示器。RGB显示器使用TTL电平信号（0或5V）作为视频输入，并使用称为亮度的第4条线来允许改变亮度。RGB视频TTL显示器可以显示总共16种不同的颜色。TTL RGB显示器用在早期计算机系统的**CGA（color graphics adapter，彩色图形适配器）**系统里。

表13-4列出了这16种颜色以及产生它们的

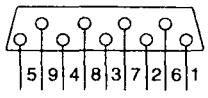
表13-4 CGA显示器中的16种颜

亮度	红	绿	蓝	颜色
0	0	0	0	黑色
0	0	0	1	蓝色
0	0	1	0	绿色
0	0	1	1	青色
0	1	0	0	红色
0	1	0	1	品红色（紫色）
0	1	1	0	棕色
0	1	1	1	白色
1	0	0	0	灰色
1	0	0	1	亮蓝色
1	0	1	0	亮绿色
1	0	1	1	亮青色
1	1	0	0	亮红色
1	1	0	1	亮紫色
1	1	1	0	黄色
1	1	1	1	亮白色

TTL 信号。其中 8 种颜色以高亮度产生，而另外 8 种以低亮度产生。3 种视频颜色为红、绿、蓝。它们是光的原色。次混合色是青色、品红色和黄色。青色是蓝色和绿色视频信号的组合，即蓝绿色。品红色是蓝色和红色视频信号的组合，即紫色。

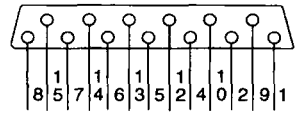
黄色（高亮度）和棕色（低亮度）均为红色和绿色视频信号的组合。如果想得到其他颜色，则通常不使用 TTL 视频。一种方案是使用低、中彩色 TTL 视频信号，它可提供 32 种颜色，但这种方案并没有被广泛应用。

图 13-27 给出了 TTL RGB 显示器或 TTL 单色显示器常用的连接器。图中的连接器为 9 个引脚。其中 2 个引脚用于接地，3 个引脚用于视频信号，2 个引脚用于同步或回扫信号，1 个引脚用于亮度。注意引脚 7 被标识为标准视频，此引脚用在单色显示器上作为亮度信号。单色 TTL 显示器使用与 RGB TTL 显示器相同的 9 引脚连接器。



引脚	功能
1	地
2	地
3	红色视频信号
4	绿色视频信号
5	蓝色视频信号
6	亮度
7	标准视频信号
8	水平回扫
9	垂直回扫

图 13-27 TTL 显示器上的 9 引脚连接器



引脚	功能
1	红色视频信号
2	绿色视频信号(单色视频)
3	蓝色视频信号
4	地
5	地
6	红色地
7	绿色地(单色地)
8	蓝色地
9	封锁为键
10	地
11	颜色检测(彩色显示器上的地)
12	单色检测(单色显示器上的地)
13	水平回扫
14	垂直回扫
15	地

图 13-28 模拟显示器上的 15 引脚连接器

13.5.3 模拟 RGB 显示器

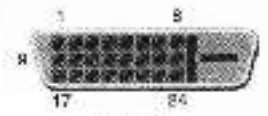
为显示多于 16 种的颜色，需要一个模拟视频显示器，通常称之为模拟 RGB 显示器。模拟 RGB 显示器也有 3 个视频输入信号，但没有亮度输入。由于视频信号为模拟信号，而不是 2 种电平的 TTL 信号，所以它们为 0.0V ～0.7V 之间的任意电压值，这就可以允许显示无限种颜色。这是因为在最小值和最大值之间可以产生无数个电压值。但实际上，只有有限个电平产生，通常对应 256K、16M 或 24M 种颜色，具体情况取决于所用的标准。

图 13-28 给出了模拟 RGB 或模拟单色显示器所用的连接器。注意，连接器有 15 个引脚，支持 RGB 和单色模拟显示器。数据在模拟 RGB 显示器上的显示方式取决于显示器所用的接口标准。引脚 9 是一个锁脚，这意味着在相应的插座上不存在此引脚的插孔。

另一种名为 **DVI-D**（**digital visual interface**）的模拟 RGB 显示器的连接器正逐渐流行。-D 代表数字化，是这种类型中最为常见的接口。图 13-29 给出了较新的显示器和显卡上的插座。电视机与视频设备中都有 HDMI（high-definition multimedia interface，高清晰度多媒体接口）连接器。这种连接器还没有进入数字视频卡，但是也许将来会出现。最终所有的视频设备都将采用 HDMI 连接器进行连接。

大多数模拟显示器使用数/模转换器（DAC）来产生每种颜色的视频电压。一个常见标准使用一个 8 位 DAC，为每个视频信号产生 0.0V ～0.7V 之间的 256 种不同的电压电平。有 256 个不同的红色视频电平，256 个不同的绿色视频电平，以及 256 个不同的蓝色视频电平。这样就允许显示 256 × 256 × 256，即 16 777 216（16M）种颜色。

图 13-30 给出了用在许多通用视频标准中的视频产生电路，如昙花一现的 **EGA**（**enhanced graphics adapter，增强型图形适配器**）和 **VGA**（**video graphics array，视频图形阵列适配器**），它们曾用于 IBM PC 中。此电路用于产生 VGA 视频。注意每种颜色是用一个 18 位数字码产生的。18 位中的 6 位被加到一个 6 位 DAC 的输入上时，用来产生每种视频颜色电压。



DVI-D
插孔连接器

针脚	信号分配	针脚		针脚	信号分配
1	Data2–	9	Data1–	17	Data0–
2	Data2+	10	Data1+	18	Data0+
3	Data2/4 屏蔽	11	Data1/3 屏蔽	19	Data0/5 屏蔽
4	Data4–	12	Data3–	20	Data5–
5	Data4+	13	Data3+	21	Data5+
6	DDC Clock	14	+5V电源	22	Clock 屏蔽
7	DDC Data	15	地(+5V)	23	Clock+
8	不连接	16	热插拔检测	24	Clock–

表13-29 较新的显示器和显卡上的 DVI-D 接下

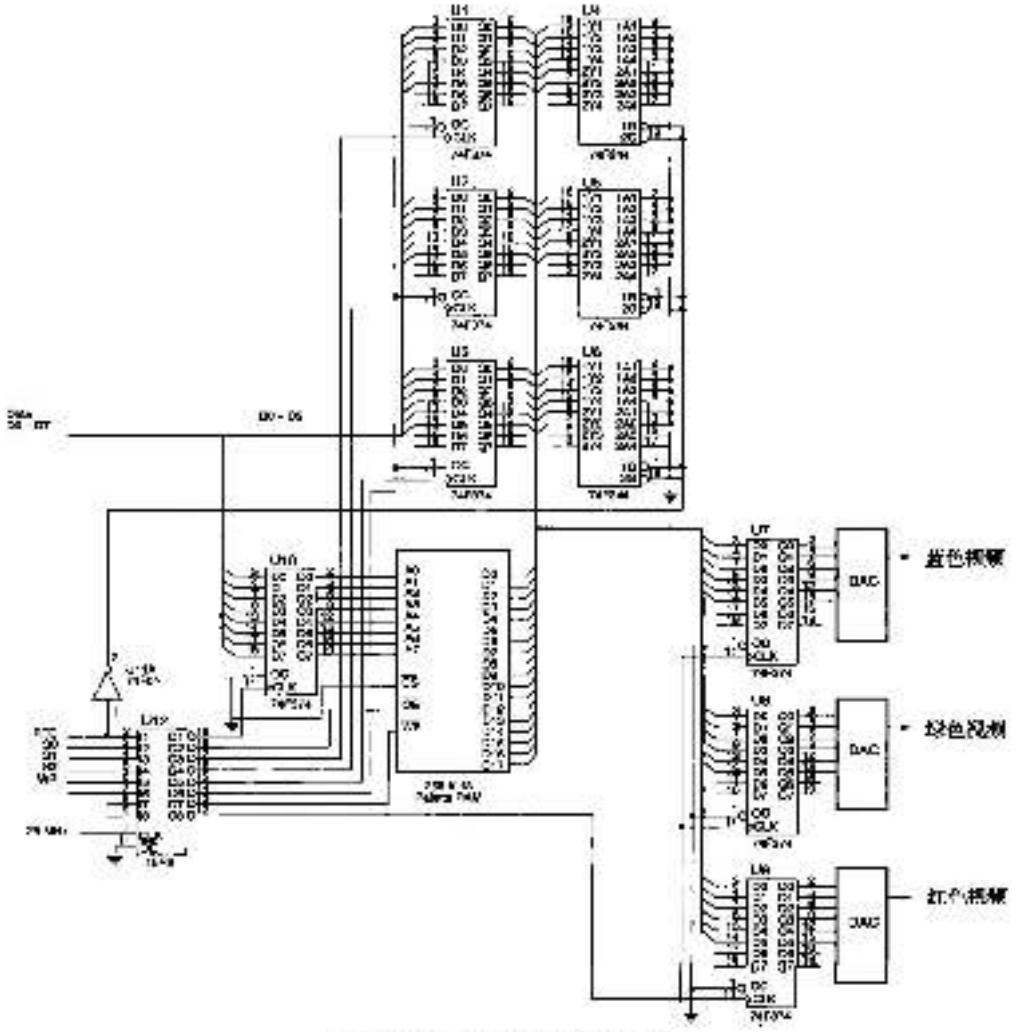


图 13-30 VGA 视频信号的产生

一个高速调色板 SRAM（存取时间小于 40ns）用于存储 256 种不同的 18 位码，代表 256 种不同的颜色。这个 18 位码被加到数/模转换器上。SRAM 的地址输入选择 256 种颜色中的一种，这 256 种颜色以 18 位二进制码的形式存储。此系统允许 256K 种可能颜色中的 256 种同时显示。为选择 256 种颜色中的任意一种，存储在计算机的视频显示 RAM 中的一个 8 位码用来指定一个像素的颜色。如果系统中使用了更多的颜色，则这个二进制码必须加宽。例如，显示 256K 种颜色中的 1024 种颜色的系统，需要一个 10 位码来寻址包含 1024 个存储单元的 SRAM，每个存储单元包含一个 18 位颜色码。一些较新的系统使用更大的调色板 SRAM 来存储最多 64K 种不同的颜色码。

一旦一种颜色显示在视频显示器上，RTC 为逻辑 0，则系统将表示颜色的 8 位码发送给 $D_0 \sim D_7$ 引脚。然后 PLD 为 U_{10}产生一个时钟脉冲，从而锁存这个颜色码。40ns 以后（一个 25MHz 时钟），PLD 为 DAC 锁存器（U_7、U_8 和 U_9）产生一个时钟脉冲。调色板 SRAM 需要这段时间来查找由 U_{10}选择的存储单元的 18 位内容。一旦颜色码（18 位）被锁存到 $U_7 \sim U_9$ 中，则 3 个 DAC 将它转换为 3 个给显示器的视频电压。对于每个要显示的 40ns 宽的像素（pixel），重复这一过程。像素为 40ns 宽是因为系统使用的是 25MHz 时钟。如果系统使用更高的时钟频率，则可得到更高的分辨率。

如果必须改变存储在 SRAM 中的颜色码（18 位），则总是在 RTC 为逻辑 1 的回扫期间完成。这样就防止了视频噪声中断显示在显示器上的图像。

为改变颜色，系统使用了 PLD 的 S_0、S_1 和 S_2 输入来选择 U_1、U_2、U_3 或 U_{10}。首先，要改变颜色的地址被发送给锁存器 U_{10}，U_{10}寻址调色板 SRAM 中的一个存储单元。然后，每种新的视频颜色被装入 U_1、U_2 和 U_3 中。最后，PAL 为 SRAM 的$\overline{WE}$输入产生一个写脉冲，将新的颜色码写入调色板 SRAM 中。

对于一个 640×480 的显示器，垂直方向回扫是 70.1 次/秒，水平方向回扫是 31 500 次/秒。在回扫期间，发送给显示器的视频信号电压必须为 0V，这使得在回扫期间显示黑色。回扫本身的作用是：垂直回扫时将电子束移到显示屏的左上角，水平回扫时将电子束移到显示屏的左边界。

图中电路使 $U_4 \sim U_6$ 缓冲器被使能，使得 0000 被加到每个 DAC 锁存器上以进行回扫。DAC 锁存器捕获此码，并为每个视频颜色信号产生 0V 电压，使显示屏为空白。0V 被定义为视频的黑色电平，而 0.7V 被定义为视频彩色信号的全亮度。

显示器的分辨率，例如 640×480，决定了视频接口卡所需的内存容量。如果此分辨率用于 256 种颜色的显示器（每个像素 8 位颜色码），那么需要 640×480 字节的内存（307 200）来存储显示器的所有像素。更高分辨率的显示器是可能的，但需要更多的内存。一个 640×480 的显示器有 480 条视频光栅行，每行有 640 个像素。**光栅行（raster line）**是显示在显示器上的视频信息的水平线。**像素（pixel）**是水平线的最小单位。

图 13-31 给出的视频显示器说明了视频线和回扫。图中每条视频线的倾斜被大大夸张了，每条线之间的间隔也是如此。该图示出了垂直方向和水平方向上的回扫。正如前面所介绍的，对于 VGA 显示器，垂直回扫是 70.1 次/秒，水平回扫是 31 500 次/秒。

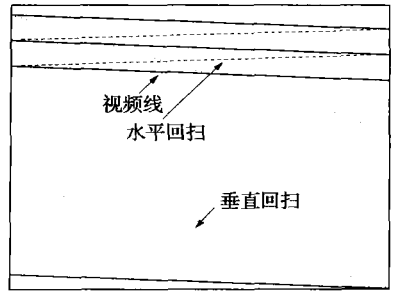


图 13-31 描绘光栅线和回扫的视频显示屏

为在一行中产生 640 个像素，需要 40ns × 640 = 25.6 μs 时间。一个 31 500Hz 的水平速率允许一条水平线的时间为 1/31 500 = 31.746 μs。这两个时间之差就是显示器允许的回扫时间（Apple Macintosh 的水平线时间为 28.57 μs）。

由于垂直回扫的重复速率为 70.1 Hz，所以产生的行数由垂直时间除以水平时间来确定。在 VGA 显示器中（640×400 的显示器），为 449.358 行。这些行中只有 400 行用来显示信息，其余行在回扫期间丢失了。由于在回扫期间丢失了 49.358 行，所以回扫时间为 49.358 × 31.746 μs = 1567 μs。正是在这段相对长的时间内，颜色调色板 SRAM 被改变或者显示器内存系统被更新为新的视频显示。在 Apple Macintosh 计算机（640×480）中，产生的行数为 525 行。总行数中有 45 行在垂直回扫期间丢失了。

其他显示器分辨率为 800×600 和 1024×768。800×600 SVGA（超级 VGA）显示器最适于 14 英寸彩色显示器，而 1024×768 EVGA 或 XVGA（扩展 VGA）最适于用在 CAD 系统中的 21 英寸或 25 英寸显示器。这些分辨率听起来只是一组数字，但是要认识到，一般家用电视接收机的分辨率大约只有 400×300。计算机系统中可得到的高分辨率显示器比家用电视机的清楚得多。1024×768 的分辨率已接近 35 mm 胶片。在计算机屏幕上视频显示的惟一缺点就是一次显示的颜色数目有限，但随着时间的推移，这种情况一定会改善的。由于一幅真正高品质的、逼真的图像需要精细的渲染，所以颜色的增加将使得图像看起来更逼真。

如果一个显示器系统以 60Hz 的垂直扫描速率和 15 600Hz 的水平扫描速率工作，则产生的行数为 15 600/60 = 260 行。此系统中可用的行数最多为 240 行，因此在垂直回扫期间丢失了 20 行。显然，扫描线的数目是通过改变垂直扫描率和水平扫描率来调整的。垂直扫描率必须大于或等于 50Hz，否则显示屏将会出现闪烁。垂直扫描率必须不高于大约 75Hz，否则垂直偏转线圈可能会出现问题。显示器中的电子束是由电磁场来定位的，而电磁场由磁轭中缠绕在显像管颈上的线圈产生。由于磁场由线圈产生，所以加在线圈上的信号频率是有限制的。

水平扫描率也受磁轭中线圈的物理设计的限制。由于这一点，通常加在水平线圈上的频率只限于一个窄的范围内。一般为 30 000Hz～37 000Hz 或 15 000Hz～17 000Hz。一些较新的显示器被称为多同步显示器，因为偏转线圈是被分接的，所以可由不同的偏转频率来驱动。有时，垂直线圈和水平线圈均被分接，以适应不同的垂直和水平扫描率。

高分辨率显示器使用隔行扫描或逐行扫描。逐行扫描系统用在除了最高标准外的所有标准中。在隔行扫描系统中，视频图像是这样形成的：首先由所有的奇扫描行画出一半图像，然后由偶扫描行画出另一半。显然，此系统更复杂也更有效，因为在隔行扫描系统中，扫描频率减小了 50%。例如，一个视频系统使用 60Hz 的垂直扫描频率和 15 720Hz 的水平扫描频率，以 60 帧/秒的速率产生 262（15 720/60）行视频。如果水平频率稍做改变，变为 15 750Hz，则将产生 262.5（15 750/60）行，所以两次完整的扫描就可以画出一幅 525 条视频行的完整图形。注意水平频率上的轻微变化是如何使光栅行数目加倍的。

13.6 小结

1）HOLD 输入用于申请一次 DMA 操作，HLDA 输出通知 HOLD 已生效。当 HOLD 输入置为逻辑 1 时，微处理器将：①停止执行程序；②将其地址、数据和控制总线置为高阻抗状态；③通过在 HLDA 引脚上置逻辑 1，通知 HOLD 已生效。

2）DMA 读操作将数据从一个存储单元中传输到外部 I/O 设备。DMA 写操作将数据从 I/O 设备传输到存储器中。另外还有存储器到存储器的传输，它通过使用 DMA 技术，允许在两个存储单元之间传输数据。

3）8237 直接存储器存取（DMA）控制器是一个 4 通道的器件，它可被扩展增加另外的 DMA 通道。

4）磁盘存储器就像 3.5 英寸微型软盘那样以软磁盘存储的形式出现。磁盘为双面双密度（DSDD）或高密度（HD）存储设备。DSDD 3.5 英寸磁盘存储 720KB 的数据，而 HD3.5 英寸磁盘存储 1.44MB 的数据。

5）软盘存储器使用 NRZ（不归零制）记录来存储数据。此方法用磁能的一个极性把磁盘磁化到饱和作为逻辑 1，用相反的极性作为逻辑 0。在任何情况下，磁场决不会归零。这种技术不需要单独的擦除磁头。

6）通过使用改进调频制（MFM）或游程长度受限（RLL）编码方案将数据记录到磁盘上。MFM 方案对于逻辑 1 记录为一个数据脉冲，对于 0 字符串的第一个逻辑 0 没有数据或时钟脉冲，对于 0 字符串的第二个和后来的逻辑 0 记录为一个时钟脉冲。与 MFM 方案相比，使用 RLL 方案进行数据编码可在同一磁盘区域多存入 50% 的信息。大多数现代磁盘存储系统均使用 RLL 编码方案。

7）视频显示器为 TTL 或模拟显示器。TTL 显示器使用 2 个离散的电压电平 0V 和 5.0V。模拟显示器使用 0.0V 和 0.7V 之间的无数个电压电平。模拟显示器可以显示无限数量的视频电平，而 TTL 显示器只限于 2 个视频电平。

8）彩色 TTL 显示器显示 16 种不同的颜色。这是通过 3 个视频信号（红、绿、蓝）和一个亮度输入来实现的。模拟彩色显示器通过其 3 个视频输入可显示无数种颜色。在实际应用中，最常见的彩色模拟显示器系统（VGA）可显示 16M 种不同的颜色。

9）当今的视频标准包括 VGA（640×480）、SVGA（800×600）和 EVGA 或 XVGA（1024×768）。在所有这 3 种情况下，视频信息可以是总共 16M 种颜色。

13.7 习题

1. 哪些微处理器引脚用于请求和响应 DMA 传输?
2. 解释一旦将逻辑 1 置于 HOLD 输入引脚上，将发生什么情况。
3. 一次 DMA 读操作将数据从_____传输到_____。
4. 一次 DMA 写操作将数据从_____传输到_____。
5. DMA 控制器通过什么总线信号选择用于 DMA 传输的存储单元?
6. DMA 控制器通过哪个引脚选择 DMA 传输期间所用的 I/O设备?
7. 什么是存储器到存储器 DMA 传输?
8. 描述当 HOLD 和 HLDA 引脚为逻辑 1 时对微处理器和 DMA 控制器的作用。
9. 描述当 HOLD 和 HLDA 引脚为逻辑 0 时对微处理器和 DMA 控制器的作用。
10. 8237 DMA 控制器是一个_____通道 DMA 控制器。
11. 如果 8237 DMA 控制器被译码在 I/O 端口 2000H ～ 200FH，那么哪些端口用于编程通道 1?
12. 8237 DMA 控制器的哪个寄存器被编程以初始化控制器?
13. 8237 DMA 控制器可以传输多少字节的数据?
14. 写一系列指令，使用8237 DMA 控制器的通道 2，将数据从存储单元 21000H ～ 210FFH 中传输到存储单元 20000H ～200FFH 中。必须初始化 8237，并使用 12.1 节中介绍的锁存器来保持 A_{19}～A_{16}。
15. 写一段指令，使用8237 的通道 3，将数据从存储器传输到外部 I/O 设备中。要传输的数据位于存储单元 20000H ～20FFFH 中。
16. 什么是笔式驱动器?
17. 3.5 英寸磁盘被称为_____软盘。
18. 数据被记录在磁盘表面上被称为_____的同心圆中。
19. 磁道被分成的数据区叫做_____。
20. 在双面磁盘上，正面和反面的磁道一起被称为_____。
21. 为什么磁盘存储系统使用 NRZ 记录?
22. 画出使用 MFM 编码写 1001010000 的时序图。
23. 画出使用 RLL 编码写 1001010000 的时序图。
24. 什么是浮动磁头?
25. 为什么硬盘上的磁头必须停泊?
26. 音圈磁头定位机械装置与步进电机磁头定位机械装置之间的区别是什么?
27. 什么是 WORM?
28. 什么是 CD－ROM?
29. 普通 DVD、HD-DVD 和蓝光 DVD 分别可以存储多少数据?
30. TTL 显示器与模拟显示器之间的区别是什么?
31. 光的 3 种原色是什么?
32. 光的 3 种次混合色是什么?
33. 什么是像素?
34. 分辨率为 1280×1024 的视频显示器包含_____行的视频信息，每行被分成_____个像素。
35. 解释一个 TTL RGB 显示器怎样显示 16 种不同颜色。
36. 什么是 DVI 和 HDMI 连接器?
37. 解释一个模拟 RGB 显示器怎样显示无数种颜色。
38. 如果一个模拟 RGB 视频系统使用 8 位 DAC，则它可产生_____种不同颜色。
39. 如果一个视频系统使用 60Hz 的垂直频率和 32 400Hz 的水平频率，则可产生多少光栅行?

第 14 章　算术协处理器、MMX 和 SIMD 技术

引言

Intel 系列的算术协处理器包括 8087、80287、80387SX、80387DX 以及与 80486SX 微处理器共同使用的 80487SX。80486DX ～ Core2 微处理器均有自己的内置算术协处理器。但某些兼容的 80486 微处理器（由 IBM 和 Cyrix 生产）内部并不包含算术协处理器。对于各种协处理器，指令系统和编程几乎完全相同，主要区别是每种协处理器被设计成与 Intel 的不同型号的微处理器共同工作。本章详尽地介绍整个 Intel 系列的算术协处理器。由于协处理器是 80486DX ～ Core2 微处理器的一个组成部分，而且这些微处理器很普遍，所以现在许多程序都需要或至少得益于一个协处理器。

标识为 80X87 的协处理器系列可以实现乘法、除法、加法、减法、求平方根、部分正切、部分反正切和对数运算。数据类型包括 16 位、32 位和 64 位带符号的整数、18 位 BCD 数据以及 32 位、64 位和 80 位浮点数。应用 80X87 执行的操作通常比使用微处理器常用指令系统写出的最有效程序来执行同等的操作快许多倍。使用改进的 Pentium 协处理器，其运算速度比同等时钟频率下 80486 微处理器执行速度快 5 倍。注意，Pentium 微处理器常常可以同时执行一条协处理器指令和两条整型指令。Pentium Pro ～ Core2 协处理器与 Pentium 协处理器的操作类似，但增加了两条新的指令，即 FCMOV 和 FCOMI。

对 Pentium ～ Core2 的**多媒体扩展**（**multimedia extension**，**MMX**）是共享算术协处理器寄存器组的一些指令。MMX 扩展是一种特殊的内部处理器，被设计用来为外部多媒体设备高速执行指令。本章将介绍 MMX 的指令系统与规范。SIMD（single-instruction，multiple data）扩展被称为 **SSE**（**streaming SIMD extensions**），与 MMX 指令集类似，但是它采用浮点数而非整型数，并且不像 MMX 指令那样使用协处理器寄存器空间。

目的

读者学习完本章后将能够：

1）在十进制数据与带符号的整数、BCD 数据以及浮点数据之间进行数据转换，用于算术协处理器，MMX 和 SIMD 技术。

2）解释 80X87 算术协处理器，MMX 和 SSE 部件的操作。

3）解释算术协处理器 MMX 和 SSE 每条指令的操作和寻址方式。

4）应用算术协处理器 MMX 和 SIMD 编程来解决复杂的数值运算问题。

14.1　算术协处理器的数据格式

本节介绍所有算术协处理器系列成员所使用的数据类型（参见表 14-1 列出的所有的 Intel 微处理器及其对应的协处理器）。这些数据类型包括带符号的整数、BCD 数据和浮点数。每种数据在系统中都有特殊的用途，而许多系统需要所有这 3 种数据类型。注意，对协处理器进行汇编语言编程常局限于修改由诸如 C/C++ 等高级语言生成的代码。为了实现这样的修改，就必须了解本章中介绍的指令系统和一些基本的编程概念。

表 14-1　微处理器和协处理器兼容性

微处理器	协处理器
8086/8088	8087
80186/80188	80187
80286	80287
80386	80387
80486SX	80487SX
80486DX ～ Core2	内置于微处理器中

14.1.1　带符号的整数

协处理器使用的带符号的整数基本上与第 1 章中介绍的带符号的整数相同。算术协处理器中使用

的带符号整数有16位（字型）、32位（短整型）和64位（长整型）。对于协处理器，长整型是新的数据格式，在第1章中没有介绍过，但原理是一样的。十进制格式和带符号的整数之间的转换方式与第1章中带符号整数的转换完全一样。读者也许还记得，正数是以原码形式存储，最左边的符号位为0；负数则以2的补码形式存储，符号位为1。

字型整数的取值范围是 $-32768 \sim +32767$，双字型的取值范围是 $\pm 2 \times 10^{+9}$，四字型的取值范围是 $\pm 9 \times 10^{+18}$。整数型数据可以在一些使用算术协处理器的应用中见到。图14-1给出了带符号整数的这3种形式。

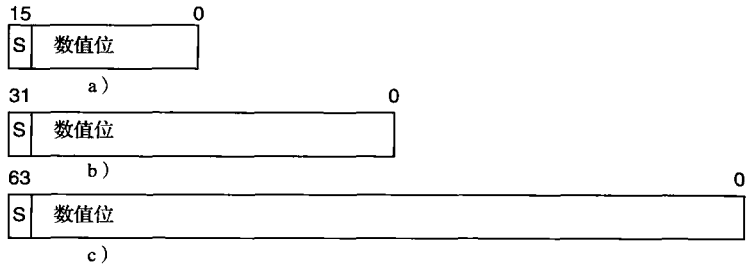


图14-1 80X87系列算术协处理器的整数格式

a）字型 b）短整型 c）长整型

注：S=符号位。

可以使用前几章中介绍的同样的汇编伪指令将数据存储在内存中。DW定义字型，DD定义短整型，DQ定义长整型。例14-1给出了以上几种不同长度的带符号整数的定义方式，以供汇编程序和算术协处理器使用。

例14-1

```
0000 0002              DATA1   DW    2        ;16位整型
0002 FFDE              DATA2   DW    -34      ;16位整型
0004 000004D2          DATA3   DD    1234     ;32位整型
0008 FFFFFF9C          DATA4   DD    -100     ;32位整型
000C 0000000000005BA0  DATA5   DQ    23456    ;64位整型
0014 FFFFFFFFFFFFFF86  DATA6   DQ    -122     ;64位整型
```

14.1.2 二进制编码的十进制（BCD）

二进制编码的十进制（BCD）形式需要80位的内存。每个数以一个18个数位的压缩整数形式存储，共占用9个字节，每个字节有2个数位，第10个字节只包含18数位带符号的BCD数据的符号位。图14-2给出了算术协处理器使用的BCD码数的格式。注意，正数和负数都是以原码形式存储的，而不是以10的补码形式存储。DT伪指令将BCD数据存储在内存中，如例14-2所示。这种格式很少用，因为它惟一用于Intel协处理器。

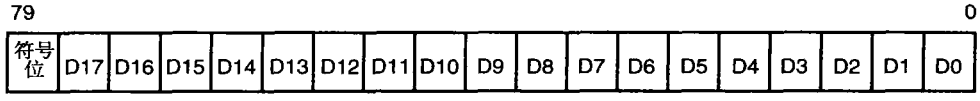


图14-2 80X87系列算术协处理器的BCD数据格式

例14-2

```
0000 00000000000000000200    DATA1  DT    200    ;定义10字节
000A 80000000000000000010    DATA2  DT    -10    ;定义10字节
0014 00000000000000010020    DATA3  DT    10020  ;定义10字节
```

14.1.3 浮点数

浮点数通常称为实数，因为它们包含有带符号的整数、分数和混合数。一个浮点数包括3个部分：符号

位、阶码和有效数字。浮点数是通过科学二进制计数法来表示的。Intel 系列算术协处理器支持3 种类型的浮点数：单格式浮点数（32 位）、双格式浮点数（64 位）和临时浮点数（80 位），参见图 14-3这 3 种形式的浮点数。注意，单格式浮点数又称为单精度浮点数，双格式浮点数又称为双精度浮点数，有时 80 位的临时浮点数也可称为扩展精度浮点数。浮点数以及算术协处理器对它们的操作都遵循 IEEE-754 标准，该标准已为所有主要的 PC 机软件生产商所接受，其中也包括 Microsoft 公司，Microsoft 公司已于 1995 年停止支持 Microsoft 公司自己的浮点格式和在许多大型计算机系统中非常流行的 ANSI 浮点标准。

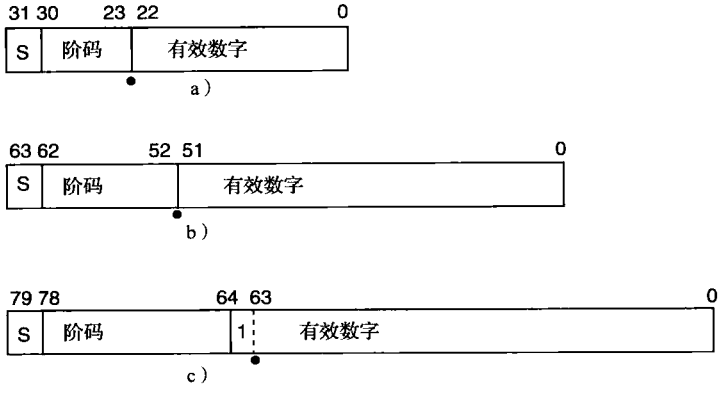


图 14-3　80X87 系列算术协处理器的浮点数（实数）格式

a）短（单精度）浮点数，偏移量为 7FH

b）长（双精度）浮点数，偏移量为 3FFH

c）临时（扩展精度）浮点数，偏移量为 3FFFH

注：S = 符号位。

在 Visual C++2008 或企业版中，float 型、double 型和 decimal 型用于表示三种数据类型。float 与 double 类型分别是 32 位和 64 位的，decimal 类型是专门为 Visual studio 开发的一种用于银行交易或其他需要高精确度场合的精确浮点数据类型。decimal 类型的变量形式在 Visual Studio 2005 和 2008 中都是新加入的。

转换为浮点数形式

将一个十进制数转换为浮点数形式很简单，步骤如下：

1）将十进制数转换为二进制数。

2）规格化二进制数。

3）计算出阶码。

4）以浮点数格式存储该数。

例 14-3 用以上 4 个步骤将十进制数 100.25_{10}转换为一个单精度（32 位）的浮点数。

例 14-3

步骤	结果
1	100.25 = >1100100.01
2	1100100.01 = >1.10010001×2^6
3	110 +01111111 = >10000101
4	符号位 = >0
	阶码 = >10000101
	有效数字 = >10010001000000000000000

在例 14-3 的第 3 步中，阶码等于指数 2^6，即 110 加上一个偏移量 01111111（7FH），得到 10000101（85H）。所有单精度浮点数使用的偏移量为 7FH，双精度浮点数使用的偏移量为 3FFH，而扩展精度浮点数使用的偏移量为 3FFFH。

在第 4 步中，将前几步中得到的信息组合起来形成一个浮点数。最左边的位是数的符号位，本例

中，由于 $+100.25_{10}$是正数，所以符号位为0。阶码跟在符号位的后面，有效数字是带有一个隐含1位的23位数。注意，数1.XXXX的有效数字是XXXX部分，1.是一个**隐含1位（implied one-bit）**，它只有以浮点数的扩展精度形式存储时才是可见的1位。

少数几个数具有特殊的规则。例如，数字0存储时除了符号位外所有其他数位都为0，而符号位可以为逻辑1，代表一个负0。正无穷和负无穷存储为：阶码为全1，有效数字为全0，符号位表示正或负。一个NAN（非数据）表示一个无效浮点数结果，其阶码为全1，而有效数字不为全0。

将浮点数转换为十进制数

浮点数转换为十进制数的步骤总结如下：

1）分离符号位、阶码和有效数字。

2）通过减去偏移量，将阶码转换为真正的指数。

3）将此数写为规格化的二进制数形式。

4）将规格化二进制数转换为非规格化二进制数。

5）将非规格化二进制数转换为十进制数。

按以上5个步骤将一个单精度浮点数转换为十进制数，如例14-4所示。注意符号位1是如何使十进制数的结果为负的。还应注意，在第3步中，隐含的1位被加到了规格化二进制数中。

例14-4

```
步骤        结果
1           符号位 = >1
            阶码 = >10000011
            有效数字 = >10010010000000000000000
2           100 =10000011 -01111111
3           1.1001001 ×2^4
4           11001.001
5           -25.125
```

将浮点数据存入内存中

在用汇编语言存储浮点数时，使用DD伪指令存储单精度数，用DQ存储双精度数，用DT存储扩展精度数。例14-5给出了一些浮点数据存储的例子。作者发现微软6.0版本的宏汇编语言中有一个错误，它不允许正浮点数使用加号。例如+92.45必须被定义为92.45，否则汇编程序就不能正常运行。微软已声明，在MASM 6.11版本中此错误已得到纠正，即使用REAL4、REAL8或REAL10伪指令取代DD、DQ和DT来定义浮点数据。如果读者的系统中没有带协处理器的微处理器，则汇编语言提供了一个可以访问的8087仿真器。此仿真器存在于微软的所有高级语言中，或者作为诸如EM87的共享程序。此仿真器是通过在程序中的.MODEL语句后面加上OPTION EMULATOR语句进行访问的。要记住，此仿真器不仿真某些协处理器指令。如果系统中有协处理器，则不要使用此仿真器。任何情况下，必须使用.8087、.80187、.80287、.80387、.80487或.80587开关来使能产生协处理器指令。

例14-5

```
0000 C377999A                DATA7  DD       -247.6      ;定义单精度
0004 40000000                DATA8  DD       2.0         ;定义单精度
0008 486F4200                DATA9  REAL4    2.45E+5     ;定义单精度
000C 4059100000000000        DATAA  DQ       100.25      ;定义双精度
0014 3F543BF727136A40        DATAB  REAL8    0.001235    ;定义双精度
001C 400487F34D6A161E4F76    DATAC  REAL10   33.9876      ;定义扩展精度
```

14.2 80X87的结构

80X87被设计成与微处理器协同工作。注意，80486DX～Core2微处理器中包含其自己内置的和

80387 完全兼容的协处理器。对于其他的 Intel 系列微处理器，协处理器是并联在微处理器上的外部集成电路。80X87 执行 68 条不同的指令。微处理器执行所有的常规指令，而 80X87 只执行算术协处理器指令。微处理器和协处理器可以同时或并行地执行各自的指令。算术协处理器是一个特殊用途的微处理器，专门是为有效地执行算术或超越函数的运算而设计的。

微处理器截取和执行常规指令系统中的指令，而协处理器只截取和执行协处理器指令。协处理器指令实际上是换码（ESC）指令，微处理器使用这些指令为协处理器产生一个内存地址，使得协处理器可以执行协处理器指令。

80X87 的内部结构

图 14-4 给出了算术协处理器的内部结构，它可分为两个主要部分：控制单元和数字执行单元。

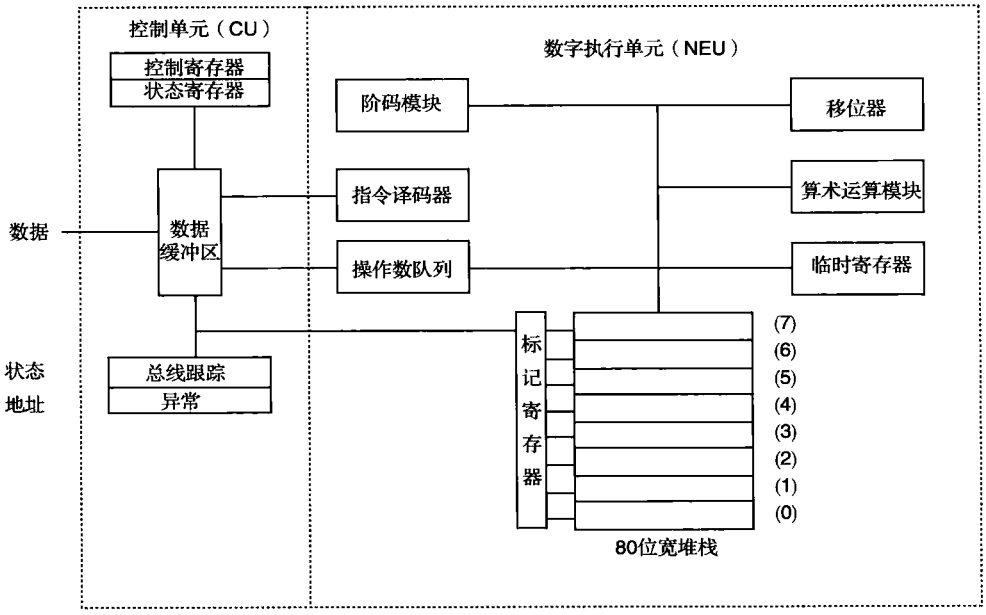


图 14-4　80X87 算术协处理器的内部结构

控制单元（control unit，CU）将协处理器连接到微处理器系统数据总线上。微处理器和协处理器均监视指令流，如果是 ESC（协处理器）指令，则由协处理器予以执行，否则由微处理器执行。

数字执行单元（numeric execution unit，NEU）负责执行所有协处理器指令。NEU 有一个 8 寄存器的堆栈，用于存储算术指令的操作数和结果。指令或者寻址在指定堆栈数据寄存器中的数据，或者使用一种压入和弹出机制在栈顶存储和取回数据。NEU 中的其他寄存器分别为状态、控制、标记和异常指针寄存器。很少有指令在协处理器和微处理器的 AX 寄存器之间传输数据，FSTSW AX 指令是协处理器允许通过 AX 寄存器和微处理器直接通信的惟一指令。注意 8087 中不包含 FSTSW AX 指令，但所有新协处理器都包含该指令。

协处理器中的堆栈包含 8 个寄存器，每个为 80 位宽。这些堆栈寄存器中总是包含一个 80 位的扩展精度浮点数。数据只有驻留在内存时才可能是任何其他格式。当数据从内存中移到协处理器的寄存器堆栈中时，协处理器将这些带符号的整数、BCD 数、单精度或双精度数转换为扩展精度浮点数。

状态寄存器

状态寄存器（见图 14-5）反映协处理器所有指令的运行情况。执行 FSTSW 指令就可以访问状态寄存器，此指令将状态寄存器中的内容存入内存的一个字单元中。在 80187 或 80187 以上的协处理器中，FSTSW AX 指令可将状态寄存器中的内容直接复制到微处理器的 AX 寄存器中。一旦状态寄存器的状态被存储到内

存或AX寄存器中，则可以使用常规软件检测状态寄存器中的各位。协处理器和微处理器之间的通信在80187和80287中是通过I/O端口00FAH～00FFH实现的，而在80387～Pentium 4中是通过I/O端口800000FAH～800000FFH实现的。注意，不要使用这些I/O端口来连接I/O设备到微处理器。

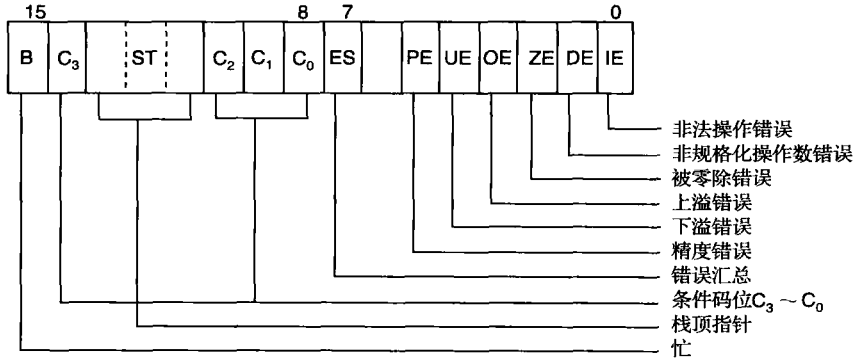


图14-5 80X87算术协处理器的状态寄存器

更新型的协处理器（80187及更高型号）使用状态位6（SF）来指示堆栈上溢或下溢错误，下面列出了除SF以外所有的状态位及其应用。

B **忙位（busy bit）**表明协处理器正忙于执行一项任务，通过检测状态寄存器或者使用FWAIT指令均可测试忙位。由于较新的协处理器自动与微处理器同步，所以在执行其他协处理器任务之前不必测试忙标志。

C_3～C_0条件码位（condition code bit），表明了协处理器的条件（参见表14-2，它列出了这些位的每种组合及其功能）。注意，这些位对于不同的指令有不同的含义，如表中所示。此表中，栈顶缩写为ST。

表14-2 80X87状态寄存器的条件码位

指　令	C_3	C_2	C_1	C_0	意　义
FTST，FCOM	0	0	X	0	ST > 操作数
	0	0	X	1	ST < 操作数
	1	0	X	1	ST = 操作数
	1	1	X	1	ST 不可比较
FPREM	Q1	0	Q0	Q2	商的最右3位
	?	1	?	?	未完成
FXAM	0	0	0	0	+ unnormal
	0	0	0	1	+ NAN
	0	0	1	0	- unnormal
	0	0	1	1	- NAN
	0	1	0	0	+ normal
	0	1	0	1	+ ∞
	0	1	1	0	- normal
	0	1	1	1	- ∞
	1	0	0	0	+0
	1	0	0	1	空
	1	0	1	0	-0
	1	0	1	1	空
	1	1	0	0	+ denormal
	1	1	0	1	空
	1	1	1	0	- denormal
	1	1	1	1	空

注：unnormal = 有效数字的前面位为0，即0. XXX；denormal = 阶码是最大的负值；normal = 标准浮点形式；NAN（非数据）= 阶码为全1，有效数字不为0，FTST的操作数为0。

TOP **栈顶（top-of-stack，ST）**位表示当前寻址为栈顶的寄存器，通常是寄存器ST（0）。

ES **错误汇总（error summary）**位，当任何一个非屏蔽的错误位（PE、UE、OE、ZE、DE或IE）被置位时，则ES被置位。在8087协处理器中该位也可引起协处理器中断。但从80187开始，

不再有协处理器中断。

PE **精度错误（precision error）**表明结果或操作数超过了设定的精度范围。

UE **下溢错误（underflow error）**表明一个非 0 的结果太小，以致不能用由控制字选择的当前精度来表示。

OE **上溢错误（overflow error）**表明结果太大而不能被表示出来，如果此错误被屏蔽，则协处理器对上溢错误就会产生一个无穷大。

ZE **被零除错误（zero error）**表明当被除数是非无穷大和非零时，除数是零。

DE **非规格化操作数错误（denormalized error）**表明至少有一个操作数是非规格化的。

IE **非法操作错误（invalid error）**表明堆栈有上溢或下溢错误，是不确定的形式（0÷0、+∞和-等），或者使用了 NAN 作为操作数。此标志表明诸如对负数开平方等类似的错误。

一旦使用 FSTSW AX 指令将状态寄存器的内容移入 AX 寄存器中，则可以有两种方法测试状态寄存器的各位。一种方法是使用 TEST 指令来测试状态寄存器的各位，另一种方法是使用 SAHF 指令将状态寄存器中最左边的 8 位传送到微处理器的标志寄存器中。例 14-6 描述了这两种方法。此例使用 DIV 指令用栈顶除以 DATA1 中的内容，使用 FSQRT 指令来求栈顶的平方根。此例还使用 FCOM 指令来比较栈顶与 DATA1 中的内容。注意，条件跳转指令和 SAHF 指令共同用来测试表 14-3 中列出的条件。尽管 SAHF 指令和条件跳转指令不能测试协处理器的所有可能的运行条件，但它们仍有助于减小特定测试条件的复杂程度。注意，SAHF 将 C_0 置入进位标志位，C_2 置入奇偶校验标志位，C_3 置入零标志位。

表 14-3 例 14-6 中在 FCOM 或 FTST 之后用条件跳转指令和 SAHF 测试的协处理器条件

C_3	C_2	C_0	条　件	跳转指令
0	0	0	ST > 操作数	JA（若 ST 大则跳转）
0	0	1	ST < 操作数	JB（若 ST 小则跳转）
1	0	0	ST = 操作数	JE（等于 ST 则跳转）

如果 Pentium 4 或 Core2 工作在 64 位模式下，那么 SAHF指令就不起作用。在 64 位模式下，需要其他的方法来测试协处理器标志，比如测试 AX 的每一比特位 C_0、C_2和 C_3（见例 14-6）。

例 14-6

```
; 测试被 0 除错误

    FDIV  DATA1
    FSTSW AX              ;将状态寄存器内容复制到 AX 中
    TEST  AX,4            ;测试 ZE 位
    JNZ   DIVIDE_ERROR

;测试 FSQRT 指令后的无效操作

    FSQRT
    FSTSW AX
    TEST  AX, 1           ;测试 IE
    JNZ   FSQRT_ERROR

;使用 SAHF 指令测试,使条件跳转指令可以执行

    FCOM  DATA1
    FSTSW AX
    SAHF                  ;将协处理器标志复制到标志寄存器中
    JE    ST_EQUAL
    JB    ST_BELOW
    JA    ST_ABOVE
;在 Pentium 4 或 Core2 的 64 位模式下
;需要下面的代码来测试先决条件
;因为 SAHF 指令在 64 位模式下不起作用

;测试条件
```

```
FCOM    DATA1
FSTSW   AX              ;复制状态寄存器到 AX 中
TEST    AX,100H
JNZ     ST_BELOW
TEST    AX,4000H
JNZ     ST_EQUAL
JMP     ST_ABOVE
```

当执行 FXAM 和 FSTSW AX 指令后，又接着执行 SAHF 指令时，则零标志位将包含 C_3。通过使用跟在 FXAM、FSTSW AX 和 SAHF 指令之后的 JZ 指令，FXAM 指令可用来在进行被零除运算之前测试除数。

控制寄存器

控制寄存器如图 14-6 所示。控制寄存器包括精度控制、舍入控制和无穷大控制，它也可以屏蔽或不屏蔽与状态寄存器最右边 6 位对应的异常位。FLDCW 指令用于给控制寄存器赋值。

以下是控制寄存器中各位及各个组合位的功能：

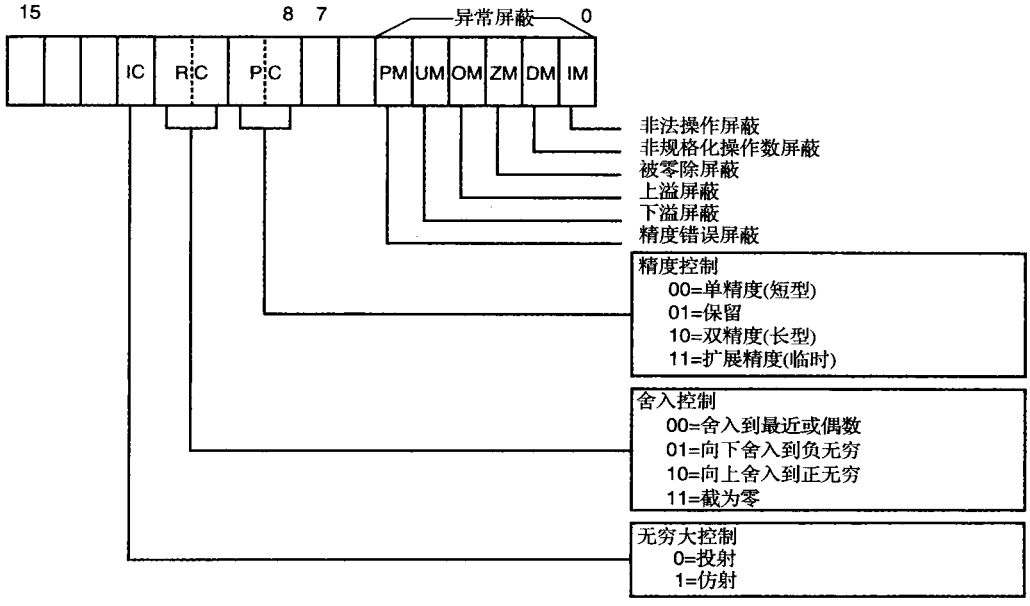


图 14-6 80X87 算术协处理器的控制寄存器

IC **无穷大控制（infinity control）**，选择是仿射无穷大还是投射无穷大。仿射允许正无穷大和负无穷大，而投射则假定无穷大为无符号的数。

RC **舍入控制（rounding control）**，确定舍入的类型，如图 14-6 所示。

PC **精度控制（precision control）**，设置结果的精度，如图 14-6 所示。

Exception Masks **异常屏蔽**，决定异常错误是否影响状态寄存器的错误位。如果其中一个异常控制位被置为逻辑 1，则相应的状态寄存器位被屏蔽。

标记寄存器

标记寄存器（tag register）表明协处理器堆栈中每个单元的内容。图 14-7 给出了标记寄存器以及每个标记指示的状态。标记表明寄存器内容是否合法、是否为零、是否不合法或为无穷大以及是否为空。通过程序查看标记寄存器的惟一方法是使用 FSTENV、FSAVE 或 FRSTOR 指令来存储协处理器操作环境。其中每条指令均可将标记寄存器与其他协处理器数据一起存储。

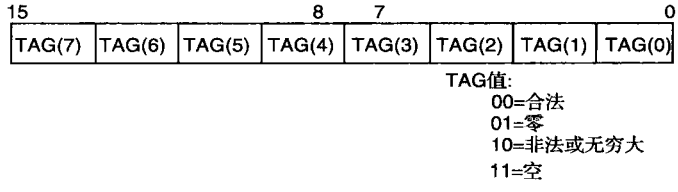


图 14-7 80X87 算术协处理器的标记寄存器

14.3 指令系统

算术协处理器可以执行超过 68 条不同的指令。一旦协处理器指令要访问内存，则微处理器自动给指令产生内存地址。协处理器在协处理器指令期间使用数据总线传送数据，而微处理器则在常规指令期间使用数据总线传送数据。注意，80287 使用 Intel 自身保留的 I/O 端口 00F8H ～ 00FFH（尽管协处理器只使用端口 00FCH ～ 00FFH）在协处理器和微处理器之间进行通信。这些端口主要由 FSTSW AX 指令使用。80387 ～ Core2 则使用 I/O 端口 800000F8H ～ 800000FFH 进行通信。

本节介绍了每条指令的功能，并列出了它们的汇编语言形式。由于协处理器使用微处理器的存储器寻址方式，所以这里并没有列出每条指令的所有可能形式。每次当汇编程序遇到协处理器的一个助记操作码时，就会把它转换为机器语言的 ESC 指令。ESC 指令代表协处理器的操作码。

14.3.1 数据传送指令

有 3 种基本的数据传送：浮点数传送、带符号整数传送和 BCD 数据传送。数据只有在内存中才以带符号整数形式或 BCD 形式出现；在协处理器内部，数据总是以 80 位扩展精度浮点数形式存储。

浮点数据传送

在协处理器指令系统中，有 4 条传统的浮点数据传送指令：FLD（装入实数）、FST（存储实数）、FSTP（存储并弹出实数）和 FXCH（交换数）。Pentium Pro ～ Core2 中增加了一条称为条件浮点传送指令的新指令，该指令使用操作码 FCMOV 和一个浮点条件。

FLD 指令将内存浮点数据装入由 ST 指向的内部栈顶。此指令将数据存储于栈顶，然后将堆栈指针减 1。装入栈顶的数据来自于任何存储单元，或来自另一个协处理器的寄存器。例如，FLD ST（2）指令将寄存器 2 中的内容复制到栈顶（ST）。当协处理器复位或初始化时，栈顶为寄存器 0。又例如，FLD DATA7 指令将存储单元 DATA7 中的内容复制到栈顶。传送的数据长度自动由汇编程序通过伪指令决定，如 DD 或 REAL4 表示单精度数据，DQ 或 REAL8 表示双精度数据，而 DT 或 REAL10 则表示扩展精度数据。

FST 指令将栈顶的内容复制到存储单元或由操作数指示的协处理器寄存器中。在存储过程中，内部的扩展精度浮点数舍入成为由控制寄存器指定的浮点数长度。

FSTP（浮点数存储并弹出）指令将栈顶内容复制到内存或任一协处理器寄存器中，然后从栈顶弹出该数据。可将 FST 指令理解为复制指令，而将 FSTP 理解为移动指令。

FXCH 指令用来交换由操作数指定的寄存器和栈顶中的内容。例如，FXCH ST（2）指令将栈顶数据与寄存器 2 中的数据进行交换。

整数传送指令

协处理器支持 3 条整数传送指令：FILD（装入整数）、FIST（存储整数）和 FISTP（存储并弹出整数）。这 3 条指令功能与 FLD、FST 和 FSTP 一样，只不过传送的数据类型为整数而不是浮点数。协处理器自动将内部的扩展精度浮点数转换为整数。数据长度由汇编语言程序中用 DW、DD 或 DQ 定义标识的方法来决定。

BCD 数据传送指令

有 2 条指令用来装入或存储 BCD 带符号的整数：FBLD 指令将内存 BCD 数据装入栈顶，FBSTP 指令存储并弹出栈顶数据。

Pentium Pro ～ Pentium 4 的 FCMOV 指令

Pentium Pro ～ Pentium 4 包含了一个称为 FCMOV 的新指令。此指令还包含一个条件，如果条件为真，则 FCMOV 指令将源内容复制到目标单元中。由 FCMOV 测试的条件以及 FCMOV 使用的操作码见表 14-4。注意，这些条

表 14-4 FCMOV 指令的变化及测试条件

指 令	条 件
FCMOVB	低于则传送
FCMOVE	等于则传送
FCMOVBE	低于或等于则传送
FCMOVU	无顺序则传送
FCMOVNB	不低于则传送
FCMOVNE	不等于则传送
FCMOVNBE	不低于或等于则传送
FCMOVNU	不是无顺序则传送

件或者是顺序检验，或者是非顺序检验。FCMOV 不检验 NAN（非数据）和非规格化操作数。

例 14-7 表明，当 ST（2）的内容低于栈顶（ST）的内容时，如何使用 FCMOVB（低于则传送）指令将 ST（2）的内容复制到栈顶（ST）。注意，FCOM 指令必须用于执行比较，而且状态寄存器的内容仍需复制到标志寄存器中，这样，FCOM 才能正常工作。更多的时候，FCMOV 指令和 FCOMI 指令同时出现，FCOMI 指令也是 Pentium Pro ～ Core2 微处理器的新指令。

例 14-7

```
FCOM   ST(2)       ;比较 ST 与 ST(2)
FSTSW  AX          ;将浮点标志寄存器内容复制到 AX
SAHF               ;将浮点标志寄存器内容复制到标志位
FCMOVB ST(2)       ;如果 ST(2)低则将 ST(2)的内容复制到 ST
                或
FCOMI  ST(2)
FCMOVB ST(2)
```

14.3.2 算术运算指令

协处理器的算术运算指令包括加法、减法、乘法、除法和求平方根指令。与算术运算相关的指令包括比例运算、舍入运算、求绝对值运算以及改变符号等指令。

表 14-5 给出了算术运算的基本寻址方式。每种寻址方式均以 FADD（实数相加）指令为例加以说明。所有算术运算都是浮点运算，除非使用内存数据作为操作数。

表 14-5 算术运算的寻址方式

方 式	形 式	例 子
堆栈	ST（1），ST	FADD
寄存器	ST，ST（n）	FADD ST，ST（1）
	ST（n），ST	FADD ST（4），ST
寄存器弹出	ST（n），ST	FADDP ST（3），ST
存储器	操作数	FADD DATA3

注：堆栈寻址固定为 ST（1），ST，也包括弹出，只有结果保留在栈顶。n = 寄存器号 0～7。

寻址操作数的传统堆栈形式（堆栈寻址）使用栈顶作为源操作数，次栈顶作为目的操作数。最后，弹出操作从堆栈移走栈顶的源数据，只有目的寄存器中的结果保留在栈顶。使用这种寻址方式，程序中的指令无需任何操作数，例如 FADD 或 FSUB 指令。FADD 指令将 ST 中的数据加上ST（1）中的数据，并将结果存储在栈顶中；它还通过弹出将原来的 2 个数据移出堆栈。尤其注意FSUB指令从ST(1)中的数据减去ST中的数据，并将差存储于 ST。因此，反向减法指令（FSUBR）从 ST 中的数据减去 ST（1）中的数据，并将差存储于 ST（注意，在 Intel 公司的文件中，包括 Pentium 数据手册，在说明一些反向指令的操作时有一个错误）。反向操作的另一个用途是求倒数（1/X）。它的实现步骤如下：如果 X 在栈顶，用 FLD1 将 1.0 装入 ST，接着使用 FDIVR 指令。FDIVR 指令实现 ST 中的数据除以 ST（1）中的数据，也即 1 除以 X，并将倒数（1/X）存于 ST。

寄存器寻址方式使用 ST 作为栈顶，ST（n）作为另一存储单元，其中 n 为寄存器号。使用这种方式，一个操作数必须是 ST，另一个操作数必须是 ST（n）。注意，为使栈顶内容加倍，必须使用 FADD ST，ST（0）指令，其中 ST（0）也寻址栈顶。在寄存器寻址方式中，2 个操作数之一必须是 ST，而另一个必须是 ST（n），其中 n 代表堆栈寄存器 0～7。对许多指令而言，ST 或 ST（n）均可以作为目的操作数。十分重要的是栈顶为 ST（0），这是在程序使用 ST（0）栈顶之前，通过复位和初始化协处理器就已完成了的。寄存器寻址的另一个例子是 FADD ST（1），ST，这里 ST 中的内容被加到 ST（1）上，结果存于 ST（1）中。

由于协处理器是面向堆栈的机器，所以栈顶总被存储器寻址方式用作目的操作数。例如，FADD DATA 指令将存储单元 DATA 中的实数加到栈顶。

算术运算操作

操作码中的字母 P 指定在操作结束后由寄存器弹出（试比较 FADDP 与 FADD）。操作码中的字母

R（只在减法和除法中出现）表示反向模式。反向模式对于内存数据十分有用。因为通常总是从栈顶减去内存数据，而一个反向减法指令是从内存数据减去栈顶数据，并将结果存于栈顶。例如，如果栈顶包含一个 10，而存储单元 DATA1 包含一个 1，则 FSUB DATA1 指令的结果为 +9，存储到栈顶。FSUBR 指令的结果为 -9。另一个例子是 FSUBR ST，ST（1），它从 ST（1）减去 ST，并将结果存于 ST 中。FSUBR ST（1），ST 则不同，它从 ST 减去 ST（1），结果存于 ST（1）中。

操作码中第 2 个字母 I 表明内存操作数是整数。例如，FADD DATA 指令是浮点加法，而 FIADD DATA 指令是整数加法，它将存储单元 DATA 中的整数加到栈顶的浮点数中。同样的规则也适应于 FADD、FSUB、FMUL 和 FDIV 指令。

与算术运算相关的操作

其他的算术运算包括 FSQRT（求平方根）、FSCALE（对一个数进行比例运算）、FPREM/FPREM1（求部分余数）、FRNDINT（舍入为整数）、FXTRACT（提取阶码和有效数字）、FABS（求绝对值）和 FCHS（改变符号）。这些指令及其功能描述如下所示：

FSQRT 求栈顶中数据的**平方根**，并将平方根存于栈顶。若对负数求平方根，则会出现非法错误。因此，一旦有非法结果出现，则应测试状态寄存器的 IE 位。IE 状态位是通过用 FSTSW AX 指令将状态寄存器中内容装入 AX 中，接着用 TEST AX，1 来测试的。

FSCALE 将 ST（1）中内容（被认为是整数）加到栈顶的指数中，**FSCALE** 能快速地乘以或除以 2 的幂。ST（1）中的值必须在 2^{-15} 与 2^{+15} 之间。

FPREM/FPREM1 完成 ST 对 ST（1）**取模**，结果余数放在栈顶，且余数与原来的被除数有相同的符号。注意，取模结果只有余数，没有商。还应注意，8086 和 80287 支持 FPREM，而在更新的协处理器中，则应该使用 FPREM1。

FRNDINT 对栈顶的数进行**舍入**运算，使之成为整数。

FXTRACT 将栈顶的数分成 2 个独立部分，分别代表无偏移的阶码和有效数字。所提取的有效数字存于栈顶，而无偏移的阶码存于 ST（1）中。此指令常用于将浮点数转化为可以作为混合数打印的格式。

FABS 将栈顶中数的符号变成正号。

FCHS 将正数转换为负数；或者将负数转换为正数。

14.3.3 比较指令

比较指令用于比较栈顶的数据与另一单元的内容，并将比较结果返回到状态寄存器中的条件码 $C_3 \sim C_0$。协处理器允许的比较指令包括 FCOM（浮点数比较）、FCOMP（浮点数比较并弹出）、FCOMPP（浮点数比较并 2 次弹出）、FICOM（整数比较）、FICOMP（整数比较并弹出）、FTST（测试）和 FXAM（检查）。Pentium Pro 中的新指令 FCOMI 指令进行浮点数比较，并将结果移到标志寄存器中。这些指令及其功能的描述如下所示：

FCOM **比较**栈顶浮点数与寄存器或内存中的操作数，如果操作数默认，则下一堆栈单元 ST（1）中内容将与栈顶浮点数相比较。

FCOMP/FCOMPP 这 2 条指令执行的功能与 FCOM 指令相同，但它们还要从堆栈弹出 1 个或 2 个数据。

FICOM/FICOMP 将栈顶的数据与存于内存中的整数比较。除比较外，FICOMP 指令还将栈顶的数据弹出。

FTST 对照 0 **测试**栈顶内容，比较结果被编码于状态寄存器中的条件码位。如表 14-2 中的状态寄存器所示。表 14-3 给出了使用 SAHF 及使用 FTST 指令的条件跳转指令。

FXAM **检测**栈顶，并修改条件码位来指示栈顶内容是否为正数、负数或规格化数等等。参见表 14-2 中的状态寄存器。

FCOMI/FUCOMI 对 Pentium Pro 和 Pentium 4 而言是新指令，该指令同 FCOM 指令完全一样地执行比

较，但增加了一个特性：它将浮点数标志移入标志寄存器中，正如例 14-8 中 FNSTSW AX 和 SAHF 指令所做的操作一样。Intel 已将 FCOM、FNSTSW AX 和 SAHF 这 3 个指令组合成 FCOMI 指令，另外还可以得到无序比较指令 FUCOMI。这 2 条新指令都可在操作码的后面加上 P 实现弹出功能。

14.3.4 超越运算指令

超越运算指令包括 FPTAN（求部分正切）、FPATAN（求部分反正切）、FSIN（求正弦值）、FCOS（求余弦值）、FSINCOS（求正弦值和余弦值）、F2XM1（计算 2^X-1）、FYL2X（计算 $Y\log_2X$）和 FYL2XP1（计算 $Y\log_2(X+1)$），下面列出这些指令的详细功能。

FPTAN 求 $Y/X=\tan\theta$ 的**部分正切值**。θ 值位于栈顶，对于 8087 和 80287，其取值范围为 0～π/4 弧度；对于 80387、80486/7 和 Pentium～Core2 微处理器，则必须小于 2^{63}。结果为比率 Y/X，其中 ST=X，ST（1）=Y。若该值超出允许范围，则出现非法错误，正如状态寄存器的 IE 位所指示的一样。还应注意，ST（7）必须为空，此指令才能正常工作。

FPATAN 求**部分反正切值**，θ=ARCTAN X/Y，其中 X 在栈顶，而 Y 在 ST（1）中。X 和 Y 值必须为 $0\leqslant Y<X<\infty$。指令将数据弹出堆栈而将结果 θ 值存于栈顶。

F2XM1 求**函数 2^X-1**，其中 X 取自栈顶，函数结果返回栈顶。若求 2^X，只需在栈顶结果上加 1。X 值必须在 -1 到 +1 之间。F2XM1 指令用于导出表 14-6 中列出的函数。注意，其中常数 $\log_2 10$ 和 $\log_2 e$ 为协处理器的内置标准值。

表 14-6 指数函数

函 数	等 价 于
10^Y	$2^Y\times\log_2 10$
ε^Y	$2^Y\times\log_2\varepsilon$
X^Y	$2^Y\times\log_2 X$

FSIN/FCOS 求栈顶 ST 中以弧度表示（360°=2π 弧度）的参数的**正弦**或**余弦值**，结果存于 ST 中。ST 中的值必须小于 2^{63}。

FSINCOS 求 ST 中以弧度表示的参数的正弦和余弦值，结果分别存于 ST（=正弦值）和 ST（1）（=余弦值）中。与 FSIN 或 FCOS 一样，ST 的初值必须小于 2^{63}。

FYL2X 求 **$Y\log_2X$** 的值。其中 X 值取自 ST，Y 取自 ST（1）。结果在栈顶弹出 X 后存于栈顶。X 的取值范围为（0，∞），Y 的取值范围为（-∞，+∞）。一个以任意正数为底的对数可以用等式 $\log_bX=(\log_2b)^{-1}\times\log_2X$ 实现。

FYL2XP1 求 **$Y\log_2(X+1)$** 的值。其中 X 取自 ST，Y 取自 ST（1）。结果在栈顶弹出 X 后存于栈顶。X 的取值范围为（0，$1-\sqrt{2}/2$），Y 的取值范围为（-∞，+∞）。

14.3.5 常数操作指令

协处理器指令系统包括返回常数到栈顶的操作码，这些指令如表 14-7所示。

表 14-7 常数操作

指 令	压入 ST 的常数
FLDZ	+0.0
FLD1	+1.0
FLDPI	π
FLDL2T	$\log_2 10$
FLDL2E	$\log_2\varepsilon$
FLDLG2	$\log_{10}2$
FLDLN2	$\log_e 2$

14.3.6 协处理器控制指令

协处理器的控制指令用于初始化、异常处理和任务切换。控制指令有两种形式。例如，FINIT 和 FNINIT 都实现对协处理器的初始化，区别在于，FNINIT 不产生任何等待状态，而 FINIT 则产生等待状态。微处理器通过测试协处理器上的 BUSY 引脚来等待 FINIT 指令。所有控制指令都有这两种形式，以下是每条控制指令及其功能的描述。

FINIT/FNINIT 此指令执行对算术协处理器的**复位**操作（参见表 14-8 中的复位条件）。协处理器在**初始化**或复位时，执行投射闭包（无符号的无穷大）操作，舍入方式为最近舍入或偶数舍入，并使用扩展精度。此指令同时设置寄存器 0 为栈顶。

表 14-8 协处理器复位或初始化后的状态

域	值	条 件
无穷大	0	投射闭包
舍入	00	就近舍入
精度	11	扩展精度
错误屏蔽	11111	错误位禁止
忙	0	不忙
$C_3 \sim C_0$	????	未知
TOP	000	寄存器 000 或 ST (0)
ES	0	无错误
错误位	00000	无错误
所有标记	11	空
寄存器	ST (0) ～ST (7)	不变

FSETPM 改变协处理器的寻址模式为**保护寻址模式**。此方式用于微处理器也工作在保护模式下时。与微处理器一样，只有通过硬件复位才能退出保护模式，而对于 80386 ～ Pentium 4，可以通过改变控制寄存器来退出保护模式。

FLDCW 将由操作数寻址的字装入控制寄存器中。

FSTCW 将控制寄存器中的内容存入长度为一个字长的内存操作数中。

FSTSW AX 将控制寄存器中的内容复制到 AX 寄存器中，在 8087 协处理器中没有此指令。

FCLEX 清除状态寄存器的“错误”标志和“忙”标志。

FSAVE 将机器的全部状态写入内存。图 14-8 给出了此指令的内存分布图。

FRSTOR 从内存复原机器状态。此指令用来恢复由 FSAVE 保存的信息。

FSTENV 存储协处理器的环境，如图 14-9 所示。

FLDENV 重装由 FSTENV/FNSTENV 保存的环境参数。

FINCSTP 堆栈指针加 1。

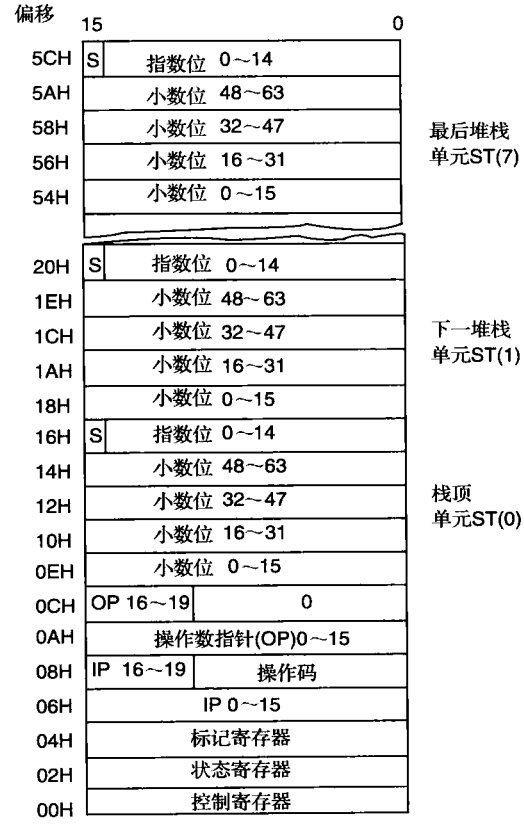


图 14-8 使用 FSAVE 指令保存 80X87 寄存器时的内存格式

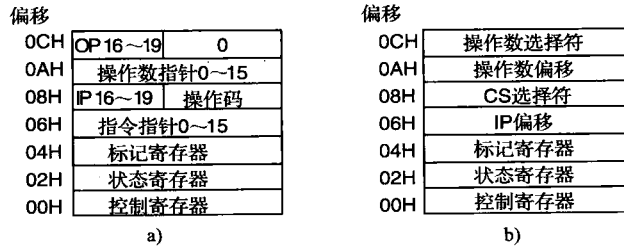


图 14-9 使用 FSTENV 指令时的内存格式
a) 实模式 b) 保护模式

FDECSTP 堆栈指针减1。

FFREE 通过把目的寄存器的标记改变为空来释放该寄存器，但不影响寄存器中的内容。

FNOP 浮点协处理器的NOP。

FWAIT 使微处理器等待协处理器完成一个操作。FWAIT指令应该用在微处理器访问被协处理器影响的内存数据之前。

14.3.7 协处理器指令

尽管本章没有讨论微处理器的电路，但却讨论了协处理器的指令系统及其与其他型号协处理器的区别。近期的协处理器包含与早期的协处理器一样的基本指令，另外还增加了几条新的指令。

80387、80486、80487SX和Pentium～Core2均包含如下的增补指令：FCOS（求余弦）、FPREM1（求部分余数）、FSIN（求正弦）、FSINCOS（求正弦和余弦）以及FUCOM/FUCOMP/FUCOMPP（无序比较）。正弦和余弦指令是指令系统中最重要的新增部分。在早期的协处理器中，正弦和余弦值是从正切中计算出来的。Pentium Pro～Core2包含2条新的浮点指令：FCMOV（条件传送）和FCOMI（比较并传送到标志寄存器）。

表14-9列出了各种型号协处理器的指令系统，同时列出了每条指令执行时所需的时钟周期数。其中列出了8087、80287、80387、80486、80487和Pentium～Core2的执行时间（Pentium～Pentium 4的时序完全相同，因为在这些微处理器内的协处理器是相同的）。指令执行时间为时钟周期和表中列出的执行时间的乘积。例如FADD指令对于80287需要70～143个时钟周期，假设80287使用8MHz的时钟，则时钟周期为1/8μs，即125ns，FADD指令执行时间为8.75μs～17.875μs。如果使用33MHz（33ns）的80486DX2芯片，则此指令需要0.264μs～0.66μs的执行时间。在Pentium协处理器中，FADD指令需要1～7个时钟周期，所以如果工作在133MHz（7.52ns）下，则FADD需要0.00752μs～0.05264μs的执行时间。而Pentium Pro～Core2则比Pentium更快。3GHz的Pentium 4时钟周期是0.333ns，FADD要占用0.333ns～2.333ns。

表14-9中使用了一些速记符来代表位移量，对于使用存储器寻址方式的指令，可以需要也可以不需要位移量。表中还使用缩写mmm表示寄存器/存储器寻址方式，使用rrr表示浮点协处理器寄存器ST（0）～ST（7）中的一个。出现在某些指令的操作码中的d（目标）位定义了数据流方向，例如FADD ST，ST（2）或FADD ST（2），ST中的情况。d位为逻辑0，表示数据流流向ST，例如，在FADD ST，ST（2）中ST保存相加的和。d位为逻辑1，表示数据从ST流出，例如，在FADD ST（2），ST中ST（2）保存相加的和。

还应注意，一些指令允许选择是否插入一个等待。例如，FSTSW AX指令将状态寄存器中的内容复制到AX寄存器中，FNSTSW AX实现同样的功能，但没有插入等待。

表14-9 算术协处理器的指令系统

F2XM1 $2^{ST}-1$		
11011001 11110000		
示例		时钟周期数
F2XM1	8087	310～630
	80287	310～630
	80387	211～476
	80486/7	140～279
	Pentium～Core2	13～57

（续）

FABS　ST 的绝对值

11011001　11100001

示例		时钟周期数
FABS	8087	10～17
	80287	10～17
	80387	22
	80486/7	3
	Pentium～Core2	1

FADD/FADDP/FIADD　加法

11011000　oo000mmm disp	32 位存储器（FADD）
11011100　oo000mmm disp	64 位存储器（FADD）
11011d00　11000rrr	FADD ST，ST（rrr）
11011110　11000rrr	FADDP ST，ST（rrr）
11011110　oo000mmm disp	16 位存储器（FIADD）
11011010　oo000mmm disp	32 位存储器（FIADD）

格式	示例		时钟周期数
FADD FADDP FIADD	FADD DATA FADD ST，ST（1） FADDP FIADD NUMBER FADD ST，ST（3） FADDP ST，ST（2） FADD ST（2），ST	8087	70～143
		80287	70～143
		80387	23～72
		80486/7	8～20
		Pentium～Core2	1～7

FCLEX/FNCLEX　清除错误

11011011　11100010

示例		时钟周期数
FCLEX FNCLEX	8087	2～8
	80287	2～8
	80387	11
	80486/7	7
	Pentium～Core2	9

FCOM/FCOMP/FCOMPP/FICOM/FICOMP　比较

11011000　oo010mmm disp	32 位存储器（FCOM）
11011100　oo010mmm disp	64 位存储器（FCOM）
11011000　11010rrr	FCOM ST（rrr）
11011000　oo011mmm disp	32 位存储器（FCOMP）
11011100　oo011mmm disp	64 位存储器（FCOMP）
11011000　11011rrr	FCOMP ST（rrr）
11011110　11011001	FCOMPP
11011110　oo010mmm disp	16 位存储器（FICOM）
11011010　oo010mmm disp	32 位存储器（FICOM）
11011110　oo011mmm disp	16 位存储器（FICOMP）
11011010　oo011mmm disp	32 位存储器（FICOMP）

（续）

格式	示例		时钟周期数
FCOM FCOMP FCOMPP FICOM FICOMP	FCOM ST (2) FCOMP DATA FCOMPP FICOM NUMBER FICOMP DATA3	8087	40～93
		80287	40～93
		80387	24～63
		80486/7	15～20
		Pentium～Core2	1～8

FCOMI/FUCOMI/COMIP/FUCOMIP　比较并装入标志寄存器

11011011	11110rrr	FCOMI ST (rrr)
11011011	11101rrr	FUCOMI ST (rrr)
11011111	11110rrr	FCOMIP ST (rrr)
11011111	11101rrr	FUCOMIP ST (rrr)

格式	示例		时钟周期数
FCOM FUCOMI FCOMIP FUCOMIP	FCOMI ST (2) FUCOMI ST (4) FCOMIP ST (0) FUCOMIP ST (1)	8087	—
		80287	—
		80387	—
		80486/7	—
		Pentium～Core2	—

FCMOVcc　条件传送

11011010	11000rrr	FCMOVB ST (rrr)
11011010	11001rrr	FCMOVE ST (rrr)
11011010	11010rrr	FCMOVBE ST (rrr)
11011010	11011rrr	FCMOVU ST (rrr)
11011011	11000rrr	FCMOVNB ST (rrr)
11011011	11001rrr	FCMOVNE ST (rrr)
11011011	11010rrr	FCMOVENBE ST (rrr)
11011011	11011rrr	FCMOVNU ST (rrr)

格式	示例		时钟周期数
FCMOVB FCMOVE	FCMOVB ST (2) FCMOVE ST (3)	8087	—
		80287	—
		80387	—
		80486/7	—
		Pentium～Core2	—

FCOS　ST 的余弦

11011001	11111111	

示例		时钟周期数
FCOS	8087	—
	80287	—
	80387	123～772
	80486/7	193～279
	Pentium～Core2	18～124

（续）

FDECSTP　堆栈指针减 1

11011001　11110110

示例		时钟周期数
FDECSTP	8087	6～12
	80287	6～12
	80387	22
	80486/7	3
	Pentium～Core2	1

FDISI/FNDISI　禁止中断

11011011　11100001

（在 80287、80387、80486/7 和 Pentium～Core2 上忽略）

示例		时钟周期数
FDISI FNDISI	8087	2～8
	80287	—
	80387	—
	80486/7	—
	Pentium～Core2	—

FDIV/FDIVP/FIDIV　除法

11011000	oo110mmm	disp	32 位存储器（FDIV）
11011100	oo100mmm	disp	64 位存储器（FDIV）
11011d00	11111rrr		FDIV ST，ST（rrr）
11011110	11111rrr		FDIVP ST，ST（rrr）
11011110	oo110mmm	disp	16 位存储器（FIDIV）
11011010	oo110mmm	disp	32 位存储器（FIDIV）

格式	示例		时钟周期数
FDIV FDIVP FIDIV	FDIV　DATA FDIV　ST，ST（3） FDIVP FIDIV　NUMBER FDIV　ST，ST（5） FDIVP ST，ST（2） FDIV ST（2），ST	8087	191～243
		80287	191～243
		80387	88～140
		80486/7	8～89
		Pentium～Core2	39～42

FDIVR/FDIVRP/FIDIVR　反向除法

11011000	oo110mmm	disp	32 位存储器（FDIVR）
11011100	oo111mmm	disp	64 位存储器（FDIVR）
11011d00	11110rrr		FDIVR ST，ST（rrr）
11011110	11110rrr		FDIVRP ST，ST（rrr）
11011110	oo111mmm	disp	16 位存储器（FIDIVR）
11011010	oo111mmm	disp	32 位存储器（FIDIVR）

（续）

格式	示例		时钟周期数
FDIVR FDIVRP FIDIVR	FDIVR DATA FDIVR ST, ST (3) FDIVRP FIDIVR NUMBER FDIVR ST, ST (5) FDIVRP ST, ST (2) FDIVR ST (2), ST	8087	191～243
		80287	191～243
		80387	88～140
		80486/7	8～89
		Pentium～Core2	39～42

FENI/FNENI 禁止中断

11011011 11100000

（在80287、80387、80486/7、Pentium～Core2上忽略）

示例		时钟周期数
FENI FNENI	8087	2～8
	80287	—
	80387	—
	80486/7	—
	Pentium～Core2	—

FFREE 释放寄存器

11011101 11000rrr

格式	示例		时钟周期数
FFREE	FFREE FFREE ST (1) FFREE ST (2)	8087	9～16
		80287	9～16
		80387	18
		80486/7	3
		Pentium～Core2	1

FINCSTP 堆栈指针加1

11011001 11110111

示例		时钟周期数
FINCSTP	8087	6～12
	80287	6～12
	80387	21
	80486/7	3
	Pentium～Core2	1

FINIT/FNINIT 初始化协处理器

11011001 11110110

示例		时钟周期数
FINIT FNINIT	8087	2～8
	80287	2～8
	80387	33
	80486/7	17
	Pentium～Core2	12～16

（续）

FLD/FILD/FBLD　装入数据到 ST（0）

11011001　oo000mmm　disp　32 位存储器（FLD）
11011101　oo000mmm　disp　64 位存储器（FLD）
11011011　oo101mmm　disp　80 位存储器（FLD）
11011111　oo000mmm　disp　16 位存储器（FILD）
11011011　oo000mmm　disp　32 位存储器（FILD）
11011111　oo101mmm　disp　64 位存储器（FILD）
11011111　oo100mmm　disp　80 位存储器（FBLD）

格式	示例		时钟周期数
FLD FILD FBLD	FLD DATA FILD DATA1 FBLD DEC_DATA	8087	17～310
		80287	17～310
		80387	14～275
		80486/7	3～103
		Pentium～Core2	1～3

FLD1　装入 +1.0 到 ST（0）

11011001　11101000

示例		时钟周期数
FLD1	8087	15～21
	80287	15～21
	80387	24
	80486/7	4
	Pentium～Core2	2

FLDZ　装入 +0.0 到 ST（0）

11011001　11101110

示例		时钟周期数
FLDZ	8087	11～17
	80287	11～17
	80387	20
	80486/7	4
	Pentium～Core2	2

FLDPI　装入 π 到 ST（0）

11011001　11101011

示例		时钟周期数
FLDPI	8087	16～22
	80287	16～22
	80387	40
	80486/7	8
	Pentium～Core2	3～5

（续）

FLDL2E　装入 $\log_2 e$ 到 ST（0）

11011001　11101010

示例		时钟周期数
FLDL2E	8087	15～21
	80287	15～21
	80387	40
	80486/7	8
	Pentium～Core2	3～5

FLDL2T　装入 $\log_2 10$ 到 ST（0）

11011001　11101001

示例		时钟周期数
FLDL2T	8087	16～22
	80287	16～22
	80387	40
	80486/7	8
	Pentium～Core2	3～5

FLDLG2　装入 $\log_{10} 2$ 到 ST（0）

11011001　11101000

示例		时钟周期数
FLDLG2	8087	18～24
	80287	18～24
	80387	41
	80486/7	8
	Pentium～Core2	3～5

FLDLN2　装入 $\log_e 2$ 到 ST（0）

11011001　11101101

示例		时钟周期数
FLDLN2	8087	17～23
	80287	17～23
	80387	41
	80486/7	8
	Pentium～Core2	3～5

FLDCW　装入控制寄存器

11011001　oo101mmm　disp

格式	示例		时钟周期数
FLDCW	FLDCW　DATA FLDCW STATUS	8087	7～14
		80287	7～14
		80387	19
		80486/7	4
		Pentium～Core2	7

（续）

FLDENV　装入环境

11011001　oo100mmm disp

格式	示例		时钟周期数
FLDENV	FLDENV　ENVIRON FLDENV　DATA	8087	35～45
		80287	25～45
		80387	71
		80486/7	34～44
		Pentium～Core2	32～37

FMUL/FMULP/FIMUL　乘法

11011000	oo001mmm	disp	32 位存储器（FMUL）
11011100	oo001mmm	disp	64 位存储器（FMUL）
11011d00	11001rrr		FMUL　ST，ST（rrr）
11011110	11001rrr		FMULP ST，ST（rrr）
11011110	oo001mmm	disp	16 位存储器（FIMUL）
11011010	oo001mmm	disp	32 位存储器（FIMUL）

格式	示例		时钟周期数
FMUL FMULP FIMUL	FMUL DATA FMUL ST，ST（2） FMUL ST（2），ST FMULP FIMUL DATA3	8087	110～168
		80287	110～168
		80387	29～82
		80486/7	11～27
		Pentium～Core2	1～7

FNOP　空操作

11011001　11010000

示例		时钟周期数
FNOP	8087	10～16
	80287	10～16
	80387	12
	80486/7	3
	Pentium～Core2	1

FPATAN　ST（0）的部分反正切

11011001　11110011

示例		时钟周期数
FPATAN	8087	250～800
	80287	250～800
	80387	314～487
	80486/7	218～303
	Pentium～Core2	17～173

（续）

FPREM 部分余数

11011001 11111000

示例		时钟周期数
FPREM	8087	15～190
	80287	15～190
	80387	74～155
	80486/7	70～138
	Pentium～Core2	16～64

FPREM1 部分余数（IEEE）

11011001 11110101

示例		时钟周期数
FPREM1	8087	—
	80287	—
	80387	95～185
	80486/7	72～167
	Pentium～Core2	20～70

FPTAN ST（0）的部分正切

11011001 11110010

示例		时钟周期数
FPTAN	8087	30～450
	80287	30～450
	80387	191～497
	80486/7	200～273
	Pentium～Core2	17～173

FRNDINT 舍入 ST（0）为整数

11011001 11111100

示例		时钟周期数
FRNDINT	8087	16～50
	80287	16～50
	80387	66～80
	80486/7	21～30
	Pentium～Core2	9～20

FRSTOR 恢复状态

11011101 oo110mmm disp

格式	示例	时钟周期数	
FRSTOR	FRSTOR DATA FRSTOR STATE FRSTOR MACHINE	8087	197～207
		80287	197～207
		80387	308
		80486/7	120～131
		Pentium～Core2	70～95

（续）

FSAVE/FNSAVE 保存机器状态			
11011101 oo110mmm disp			
格式	示例		时钟周期数
FSAVE FNSAVE	FSAVE STATE FNSAVE STATUS FSAVE MACHINE	8087	197～207
		80287	197～207
		80387	375
		80486/7	143～154
		Pentium～Core2	124～151
FSCALE 比例运算			
11011001 11111101			
示例			时钟周期数
FSCALE		8087	32～38
		80287	32～38
		80387	67～86
		80486/7	30～32
		Pentium～Core2	20～31
FSETPM 设置保护方式			
11011011 11100100			
示例			时钟周期数
FSETPM		8087	—
		80287	2～18
		80387	12
		80486/7	—
		Pentium～Core2	—
FSIN ST（0）的正弦			
11011001 11111110			
示例			时钟周期数
FSIN		8087	—
		80287	—
		80387	122～771
		80486/7	193～279
		Pentium～Core2	16～126
FSINCOS 求 ST（0）的正弦和余弦			
11011001 11111011			
示例			时钟周期数
FSINCOS		8087	—
		80287	—
		80387	194～809
		80486/7	243～329
		Pentium～Core2	17～137

（续）

FSQRT ST（0）的平方根

11011001 11111010

示例		时钟周期数
FSQRT	8087	180～186
	80287	180～186
	80387	122～129
	80486/7	83～87
	Pentium～Core2	70

FST/FSTP/FIST/FISTP/FBSTP 存储

11011001	oo010mmm disp	32位存储器（FST）
11011101	oo010mmm disp	64位存储器（FST）
11011101	11010rrr	FST ST（rrr）
11011011	oo011mmm disp	32位存储器（FSTP）
11011101	oo011mmm disp	64位存储器（FSTP）
11011011	oo111mmm disp	80位存储器（FSTP）
11011101	11001rrr	FSTP ST（rrr）
11011111	oo010mmm disp	16位存储器（FIST）
11011011	oo010mmm disp	32位存储器（FIST）
11011111	oo011 mmm disp	16位存储器（FISTP）
11011011	oo011mmm disp	32位存储器（FISTP）
11011111	oo111mmm disp	64位存储器（FISTP）
11011111	oo110mmm disp	80位存储器（FBSTP）

格式	示例		时钟周期数
FST FSTP FIST FISTP FBSTP	FST DATA FST ST（3） FST FSTP FIST DATA2 FBSTP DATA6 FISTP DATA9	8087	15～540
		80287	15～540
		80387	11～534
		80486/7	3～176
		Pentium～Core2	1～3

FSTCW/FNSTCW 存储控制寄存器

11011001 oo111mmm disp

格式	示例		时钟周期数
FSTCW FNSTCW	FSTCW CONTROL FNSTCW STATUS FSTCW MACHINE	8087	12～18
		80287	12～18
		80387	15
		80486/7	3
		Pentium～Core2	2

（续）

FSTENV/FNSTENV 存储环境

11011001 oo110mmm disp

格式	示例		时钟周期数
FSTENV FNSTENV	FSTENV CONTROL FNSTENV STATUS FSTENV MACHINE	8087	40～50
		80287	40～50
		80387	103～104
		80486/7	58～67
		Pentium～Core2	48～50

FSTSW/FNSTSW 存储状态寄存器

11011101 oo111mmm disp

格式	示例		时钟周期数
FSTSW FNSTSW	FSTSW CONTROL FNSTSW STATUS FSTSW MACHINE FSTSW AX	8087	12～18
		80287	12～18
		80387	15
		80486/7	3
		Pentium～Core2	2～5

FSUB/FSUBP/FISUB 减法

11011000 oo100mmm disp	32 位存储器（FSUB）
11011100 oo100mmm disp	64 位存储器（FSUB）
11011d00 11101rrr	FSUB ST，ST（rrr）
11011110 11101rrr	FSUBP ST，ST（rrr）
11011110 oo100mmm disp	16 位存储器（FISUB）
11011010 oo100mmm disp	32 位存储器（FISUB）

格式	示例		时钟周期数
FSUB FSUBP FISUB	FSUB DATA FSUB ST，ST（2） FSUB ST（2），ST FSUBP FISUB DATA3	8087	70～143
		80287	70～143
		80387	29～82
		80486/7	8～35
		Pentium～Core2	1～7

FSUBR/FSUBRP/FISUBR 反向减法

11011000 oo101mmm disp	32 位存储器（FSUBR）
11011100 oo101mmm disp	64 位存储器（FSUBR）
11011d00 11100rrr	FSUBR ST，ST（rrr）
11011110 11100rrr	FSUBRP ST，ST（rrr）
11011110 oo101mmm disp	16 位存储器（FISUBR）
11011010 oo101mmm disp	32 位存储器（FISUBR）

格式	示例		时钟周期数
FSUBR FSUBRP FISUBR	FSUBR DATA FSUBR ST，ST（2） FSUBR ST（2），ST FSUBRP FISUBR DATA3	8087	70～143
		80287	70～143
		80387	29～82
		80486/7	8～35
		Pentium～Core2	1～7

（续）

FTST 比较 ST（0）与 +0.0		
11011001 11100100		
示例		时钟周期数
FTST	8087	38～48
	80287	38～48
	80387	28
	80486/7	4
	Pentium～Core2	1～4

FUCOM/FUCOMP/FUCOMPP 无序比较			
11011101 11100rrr FUCOM ST, ST（rrr）			
11011101 11101rrr FUCOMP ST, ST（rrr）			
11011101 11101001 FUCOMPP			
格式	示例		时钟周期数
FUCOM	FUCOM ST, ST（2）	8087	—
FUCOMP	FUCOM	80287	—
FUCOMPP	FUCOMP ST, ST（3）	80387	24～26
	FUCOMP	80486/7	4～5
	FUCOMPP	Pentium～Core2	1～4

FWAIT 等待		
10011011		
示例		时钟周期数
FWAIT	8087	4
	80287	3
	80387	6
	80486/7	1～3
	Pentium～Core2	1～3

FXAM 检查 ST（0）		
11011001 11100101		
示例		时钟周期数
FXAM	8087	12～23
	80287	12～23
	80387	30～38
	80486/7	8
	Pentium～Core2	21

14.4 算术协处理器编程

本节提供了算术协处理器的几个编程实例。每个实例都说明了协处理器的编程技巧。

14.4.1 计算圆的面积

第1个编程实例给出了寻址协处理器堆栈的简单方法。首先回忆一下，计算圆面积的公式为 A =

πR^2。完成该计算的程序如例 14-8 所示。注意，程序从数组 RAD 中提取测试数据，RAD 包含 5 个半径的采样值。而这 5 个半径对应的 5 个面积被存于名为 AREA 的第 2 个数组中。此程序没有使用 AREA 数组中的数据。

尽管这是一个简单程序，但它却说明了堆栈的操作。为更好地理解堆栈的操作，图 14-10 给出了例 14-8中每条指令执行后堆栈中的内容。注意，由于程序中计算了 5 个圆面积，而每一过程均完全相同，所以图 14-10 中只给出一个循环。

指令	ST(0)	ST(1)
FLDPI	π	
FLD RAD[ECX*4]	2.34	π
FMUL ST,ST(0)	5.4756	π
FMUL ST,ST(1)	17.202	π
FSTP AREA[ECX*4]	π	

图 14-10 例 14-8 中堆栈的操作，注意图中表示的是指令执行后的堆栈

例 14-8

```
;该过程用于求 5 个圆的面积,其半径值存于数组 RAD 中

RAD    DD    2.34                    ;半径数组
       DD    5.66
       DD    9.33
       DD    234.5
       DD    23.4

AREA   DD    5 DUP(?)                ;面积数组

FINDA  PROC  NEAR

       FLDPI                         ;装载常数 π
       MOV ECX,0                     ;初始化指针
       .REPEAT
            FLD  RAD [ECX* 4]        ;取半径
            FMUL ST,ST(0)            ;求半径值的平方
            FMUL ST,ST(1)            ;半径的平方乘以 π
            FSTP AREA [ECX* 4]       ;保存面积值
            INC  ECX                 ;指向下一个半径
       .UNTI ECX = 5                 ;重复 5 次
       FCOMP                         ;从协处理器堆栈中清 π
       RET

FINDA  ENDP
```

第 1 条指令将 π 装入栈顶，然后将内存单元 RAD［ECX＊4］的内容，即数组中的一个元素装载到栈顶。这就把 π 推到了 ST（1）。之后，FMUL ST，ST（0）对栈顶的半径求平方。FMUL ST，ST（1）指令求出了面积。最后栈顶被存到面积数组中，并且将结果出栈为下一次迭代做好准备。

一定要注意随时移出所有堆栈数据。在 RET 之前的最后一条指令把 π 弹出堆栈。因为如果在程序结束后数据仍然存于栈中，则栈顶将不再是寄存器 0，这样就会引发许多问题，因为程序总是假定栈顶为寄存器 0。另外一种确保协处理器被初始化的方法是将 FINIT（初始化）指令放在程序的开始部分。

14.4.2 求谐振频率

电子学中的一个常用方程式是确定 LC 电路谐振频率的公式。例 14-9 给出了解方程式 $Fr=\frac{1}{2\pi\sqrt{LC}}$ 的程序。此例中用 L1 作为电感 L，C1 作为电容 C，RES 作为谐振频率的结果。

例 14-9

```
RES  DD      ?              ;谐振频率
```

```
L1    DD      0.0001          ;1mH 电感值
C1    DD      47E-6           ;47μF 电容值

FR    PROC    NEAR

      FLD     L1              ;取 L
      FMUL    C1              ;求 LC
      FSQRT                   ;求 LC 平方根
      FLDPI                   ;取 π
      FADD ST,ST(0)           ;求 2π
      FMUL                    ;求 2π 乘 LC 平方根
      FLD1
      FDIVR                   ;求倒数
      FSTP    RES
      RET

FR    ENDP
```

注意，此程序用简单的方式就解出了方程式。由于协处理器内部堆栈的使用，所以几乎不需要额外的数据操作。还应注意，DIVRP 是如何使用传统的堆栈寻址方式来求倒数的。如果读者有一个逆波兰输入计算器，如 Hewlett-Packard 生产的计算器，则应该熟悉堆栈寻址方式。如果没有，使用协处理器将增加读者对这类输入的经验。

14.4.3 使用一元二次方程求根

此例给出了如何通过解一元二次方程求一个多项式（$ax^2 + bx + c = 0$）的根。一元二次方程的解为：

$$b \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$

例 14-10 给出了求一元二次方程根（R_1 和 R_2）的程序。3 个常数分别存于存储单元 A_1、B_1 和 C_1 中。注意，如果根是虚数，则不要求求解。此例对是否有虚根进行测试，如果有，则返回 DOS，并将根（R_1 和 R_2）赋予零值。实际上，虚根是可以求解的，并可存储在一系列独立的结果存储单元中。

例 14-10

```
; 该过程用一元二次方程求多项式方程的根,若根 1 (R1) 和根 2 (R2) 均为 0,则表明为虚根.

FOUR  DW    4               ;整数 4
A1    DD    ?               ;a 值
B1    DD    ?               ;b 值
C1    DD    ?               ;c 值
R1    DD    ?               ;根 1
R2    DD    ?               ;根 2

ROOTS PROC NEAR

      FLDZ                  ;得到 0.0
      FST   R1              ;清除根
      FSTP  R2
      FLD   A1              ;求 2a
      FADD  ST,ST(0)
      FILD  FOUR            ;取 4
      FMUL  A1              ;求 4ac
      FMUL  C1
      FLD   B1              ;求 b²
      FMUL  ST,ST(0)
```

```
        FSUBR               ;求 b² -4ac
        FTST                ;测试结果是否为0
        SAHF
        .IF ! ZERO?
             FSQRT                ;求 b² -4ac 的平方根
             FSTSW AX             ;测试非法错误(负数)
             TEST AX,1
             .IF  ! ZERO?
                  FCOMPP          ;清除堆栈
                  RET
             .ENDIF
        .ENDIF
        FLD  B1
        FSUB ST,ST(1)
        FDIV ST,ST(2)
        FSTP R1             ;保存根1
        FLD  B1
        FADD
        FDIVR
        FSTP R2             ;保存根2
        RET

ROOTS ENDP
```

14.4.4 使用内存数组存储结果

下一个编程实例说明了内存数组的使用方法，以及如何用比例变址寻址方式访问此数组。例 14-11 给出了计算 100 个感应电抗值的程序。感应电抗的计算公式为 XL = 2πFL，本例中 F 的频率范围为 10Hz～1000Hz，电感值为 4mH。注意，指令 FSTP XL[ECX * 4 + 4]是如何用来存储每个频率的电抗的，即首先存储最大频率 1000Hz 的电抗值，而最后存储 10Hz 的电抗值。同时注意，FCOMP 指令是如何用于在 RET 指令之前清除堆栈的。

例 14-11

```
;该过程用于计算L的感抗值,频率范围为10Hz～1000Hz,频率存于数组XL中,注意按10Hz递增.

XL    DD     100 DUP(?)              ;XL数组
L     DD     4E-3                    ;L=4mH
F     DW     10                      ;F为整型10
XLS   PROC   NEAR

      MOV    ECX,100                 ;装入计数值=100
      FLDPI                          ;取π
      FADD   ST,ST(0)                ;形成2π
      FMUL   L                       ;形成2πL
      .REPEAT
             FILD  F                 ;取F
             FMUL  ST,ST(1)          ;求XL
             FSTP  XL [ECX* 4 +4]    ;存结果
             MOV   AX,F
             ADD   AX,10             ;F+10
             MOV   F,AX
      .UNTILCXZ
      FCOMP                          ;清堆栈
      RET
```

```
XLS   ENDP
```

14.4.5 将单精度浮点数转换为字符串

本节给出了如何取出32位单精度浮点数中的内容，并将它保存为一个ASCII码字符串。此过程将浮点数转换成一个混合数，其中包括整数部分和小数部分，中间用小数点分开。为简化此过程，这里限定所显示的混合数长度，即整数部分为32位二进制数（±2G），而小数部分为24位二进制数（1/16M），对于太大或太小的数，此过程将不能正常工作。

例14-12列出了这样一个程序，它调用一个过程将存储单元NUMB中内容转换成存储在STR数组中的字符串。该过程首先测试数的符号位，对于负数则显示减号。显示减号后，如果需要，可以用FABS指令将此数变为正数。然后，这个数被分成整数和小数部分，并分别存储在WHOLE和FRACT存储单元中。注意，FRNDINT指令是如何被用来舍入栈顶的数（使用截断方式），从而形成NUMB的整数部分的。原来的数减去整数部分得到小数部分，这是通过用FSUB指令从ST中的内容减去ST（1）中的内容而实现的。

例14-12

```
;该过程将浮点数转换成ASCII字符串

STR     DB      40 DUP(?)               ;存储字符串
NUMB    DD      -2224.125               ;测试数据
WHOLE   DD      ?
FRACT   DD      ?
TEMP    DW      ?                       ;放CW
TEN     DW      10                      ;整数10

FTOA    PROC    NEAR USES EBX ECX EDX

        MOV     ECX,0                   ;初始化指针
        FSTCW   TEMP                    ;保存当前控制字
        MOV     AX, TEMP                ;将四舍五入改为截断
        PUSH    AX
        OR      AX,0C00H
        MOU     TEMP,AX
        FLDCW   TEMP
        FTST    NUMB                    ;测试NUMB
        FSTSW   AX
        AND     AX,4500H                ;取C0、C2和C3
        .IF AX  ==100H                  ;如果为负
                MOV STR[ECX],'-'
                INC ECX
                FABS                    ;使为正

        .ENDIF
        FRNDINT                         ;舍入为整数
        FIST    WHOLE                   ;保存整数部分
        FLD     NUMB                    ;计算并保存小数
        FABS
        FSUBR
        FSTP    FRACT
        MOV     EAX,WHOLE               ;转换整数部分
        MOV     EBX,10
        PUSH    EBX
        .REPEAT
                MOV EDX,0
```

```
            DIV EBX
            ADD DL,30H                  ;转换成 ASCII 码
            PUSH EDX
        .UNTIL EAX = =0
        POP    EDX
        MOV    AH,3                     ;逗号计数
        .WHILE EDX ! = =10              ;整数部分为 ASCII 码
            POP    EBX
            DEC    AH
            .IF AH = =0&&EBX ! =10
                MOV STR[ECX],'-'
                INC ECX
                MOV AH,3
            .ENDIF
            MOV    STR[ECX],DL
            INC    ECX
            MOV    EDX,EBX
        .ENDW
        MOV    STR[ECX],'-'             ;保存小数点
        INC    ECX
        POP    TEMP                     ;重新存储原始的 CW
        FLDCW  TEMP
        FLD    FRACT                    ;转换小数部分
        .REPEAT
            FIMUL   TEN
            FIST    TEMP
            MOV     AX,TEMP
            ADD     AL,30H
            MOV     STR[ECX],AL
            INC     ECX
            FISUB   TEMP
            FXAM
            SAHF
        .UNTIL ZERO?
        FCOMP                           ;消除堆栈
        MOV    STR [ECX],0              ;存储 null
        RET

FTOA    ENDP
```

14.5 MMX 技术简介

MMX[⊖]（**multimedia extensions，多媒体扩展**）技术在 Pentium ～ Pentium 4 微处理器的指令系统中增加了 57 条新指令。MMX 技术还引入了新的通用指令。新的 MMX 指令被设计应用于动态视频、组合图形与视频、图像处理、音频合成、语音合成与压缩、电话、视频会议、2D 图形以及 3D 图形等方面。这些新指令（1995 年开始在 Pentium 中使用）与算术协处理器的运算指令并行执行。

14.5.1 数据类型

MMX 体系结构引入了新的压缩数据类型，它们是：8 个压缩的、连续的 8 位字节，4 个压缩的、连续的 16 位字以及 2 个压缩的、连续的 32 位双字。这种多字节格式中的字节具有连续的内存地址，

⊖ MMX 是 Intel 公司的注册商标。

而且像其他 Intel 数据那样使用从小到大的形式。参见图 14-11 中这些新的数据类型格式。

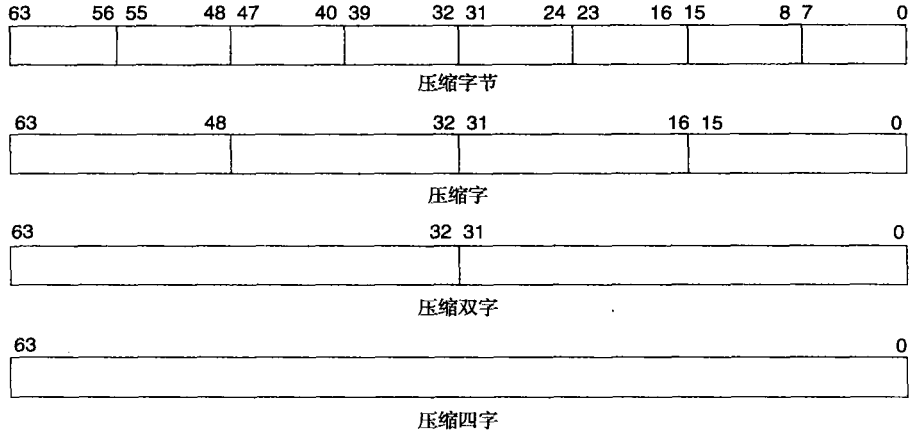


图 14-11 存储于 MMX 寄存器中的数据结构

MMX 技术寄存器与 64 位宽内存有相同的格式，它有两种数据存取方式，即 64 位存取方式和 32 位存取方式。大多数指令使用 64 位存取方式进行 64 位内存与寄存器之间的传送，使用 32 位存取方式进行 32 位内存与寄存器之间的传送。32 位传送出现在微处理器的寄存器之间，而 64 位传送出现在浮点协处理器的寄存器之间。

图 14-12 给出了 MMX 技术扩展的内部寄存器组，并说明了如何使用浮点协处理器的寄存器组。此技术被称为**别名使用**（**aliasing**），因为浮点寄存器被共享为 MMX 寄存器，也即 MMX 寄存器（$MM_0 \sim MM_7$）与浮点寄存器是相同的。注意 MMX 寄存器组为 64 位宽，并使用浮点寄存器组的最右边 64 位。

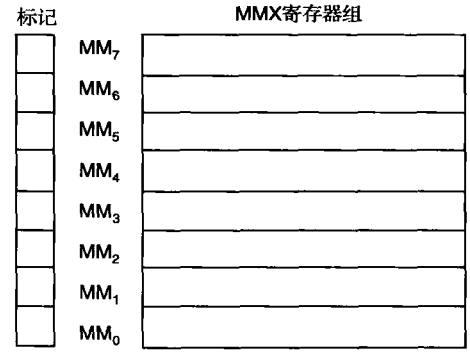


图 14-12 MMX 寄存器组的结构

注：MM_0 和 FP_0 一直到 MM_7 和 FP_7 可以相互交换。

14.5.2 指令系统

MMX 技术的指令包括算术运算、比较、转换、逻辑运算、移位以及数据传送等指令。尽管这些指令类型与微处理器的指令系统很相似，但主要区别在于 MMX 指令使用图 14-11 所示的数据类型，而不是微处理器所使用的常规数据类型。

算术运算指令

算术运算指令包括加法、减法、乘法以及特殊的带加法的乘法指令。有 3 条加法指令。PADD 和 PSUB 指令加上或减去压缩的带符号的字节或者是压缩的不带符号的字节、压缩字或压缩的双字数据。加法指令后附加字母 B、W 或 D，以选择数据长度。例如，PADDB 选择字节，PADDW 选择字，而 PADDD 选择双字。PSUB 指令也是如此。PMULHW 和 PMULLW 指令完成 4 对 16 位操作数的乘法，并产生 32 位的结果。PMULHW 指令进行高 16 位的乘法，而 PMULLW 指令进行低 16 位的乘法。PMADDWD 完成乘法和加法。在进行乘法后，4 个 32 位的结果相加，从而产生 2 个 32 位双字结果。

MMX 指令与整数或浮点数指令一样使用操作数，区别在于寄存器名称（$MM_0 \sim MM_7$）不同。例如，*PADDB* MM_1，MM_2 指令将 MM_2 的整个 64 位内容按字节与 MM_1 中的内容相加，结果存入 MM_1 中。当每 8 位数据相加时，所产生的任何进位都将会丢失。例如，字节 A0H 加上 70H 得到和 10H，而真实的和是 110H，进位丢失了。注意，第 2 个操作数或源数据可以是包含 64 位压缩源数据的存储单元，

或是 MMX 寄存器。可以说此指令完成的功能与 8 个独立的字节长的 ADD 指令所完成的相同！如果应用于实际中，一定会提高指令运行速度。与 PADD 一样，PSUB 也没有进位或借位。区别在于发生上溢或下溢，对于上溢，差变为 7FH（+127）；而对于下溢，差变为 80H（-128）。Intel 称之为饱和（saturation），因为这两个值代表了最大和最小的带符号字节。

比较指令

有两条比较指令：PCMPEQU（相等）和 PCMPGT（大于）。正如 PADD 和 PSUB 一样，每条比较指令均有 3 种形式。例如，PCMPEQUB（比较字节）、PCMPEQUW（比较字）和 PCMPEQUD（比较双字）。这些指令并不改变微处理器的标志位，相反，在条件为真时结果为全 1，而在条件为假时结果为全 0。例如，执行 PCMPEQUB MM_2，MM_3 指令，而 MM_2 和 MM_3 的最低有效字节分别为 10H 和 11H，则位于 MM_2 中的最低有效字节结果为 00H，它表明 2 个最低有效字节不相等。如果最低有效字节结果为 FFH，则表明 2 个字节相等。

转换指令

有两条基本的转换指令：PACK 和 PUNPCK。PACK 又分为 PACKSS（带符号饱和）和 PACKUS（不带符号饱和）。PUNPCK 又分为 PUNPCKH（解压缩高位数据）和 PUNPCKL（解压缩低位数据）。与前面的指令类似，它们可附加字母 B、W 或 D，分别表示字节、字和双字压缩或解压缩；但它们必须以 WB（字到字节）或 DW（双字到字）组合形式使用。例如，PACKUSWB MM_3，MM_6 指令将 MM_6 中的字压缩为字节存入 MM_3 中。如果不带符号的字不适合一个字节（因为太大），则目标字节将变为 FFH。对于带符号的饱和，我们使用在加法中介绍过的同样的值。

逻辑运算指令

逻辑运算指令有 PAND（与）、PANDN（与非）、POR（或）和 PXOR（异或）。这些指令没有长度扩展，它们对数据的所有 64 位进行按位运算。例如，POR MM_2，MM_3 指令将 MM_3 中的全部 64 位与 MM_2 中的全部 64 位相或。逻辑和在或运算后存入 MM_2 中。

移位指令

移位指令包含逻辑移位和算术右移指令。逻辑移位指令为 PSLL（左移）和 PSRL（右移）。根据数据长度又分为字（W）、双字（D）和四字（Q）。例如，PSLLQ $MM_{3,2}$ 指令将 MM_3 中的所有 64 位左移 2 位。另一例子是 PSLLD $MM_{3,2}$ 指令，它将 MM_3 中的 2 个 32 位双字各左移 2 位。

PSRA（算术右移）指令与逻辑移位工作方式相同，但它保留符号位。

数据传送指令

有 2 条数据传送指令：MOVED 和 MOVEQ。它们允许在寄存器之间及寄存器与内存之间传送数据。MOVED 指令在一个整型寄存器或整型存储单元与一个 MMX 寄存器之间传送 32 位数据。例如，MOVED ECX，MM_2 指令将 MM_2 中的最右边 32 位数复制到 ECX 中。没有用于传送 MMX 寄存器的最左边 32 位数的指令，但可以在 MOVED 指令进行传送之前将数据右移。

MOVEQ 指令将 MMX 寄存器中的 64 位全部复制到内存或另一 MMX 寄存器中。MOVEQ MM_2，MM_3 指令将 MM_3 中的 64 位全部传送到 MM_2 中。

EMMS 指令

EMMS（空 MMX 状态）指令置位（11）浮点单元中的所有标记，所以浮点寄存器表为空。必须在任何 MMX 过程的末尾执行返回指令之前执行 EMMS 指令，否则后来的浮点运算将产生一个浮点中断错误，从而导致 Windows 或任何其他应用软件崩溃。如果想要在 MMX 过程内部使用浮点指令，则必须在执行浮点指令之前使用 EMMS 指令。所有其他 MMX 指令则清除标记，表明所有浮点寄存器正在使用中。

指令列表

表 14-10 列出了带机器码的所有 MMX 指令，所以这些指令可以被汇编语言使用。目前，MASM 还不支持这些新的指令，除非将其更新到最新的 6.15 版本。可以在 Windows Driver Development Kit（Windows DDK）中找到最新的版本，但是要付费给 Microsoft 公司。在 Visual Studio. NET2003

（ML. EXE）也可以找到该软件。通过使用内嵌汇编器，VC++ 支持所有的 MMX 指令。

表 14-10 MMX 指令系统扩展

EMMS 空的 MMX 状态	
0000 1111 0111 1111	
示例	
EMMS	
MOVED 传送双字	
0000 1111 0110 1110 11 xxx rrr	reg →xreg
示例	
MOVED MM3, EDX MOVED MM4, EAX	
0000 1111 0111 1110 11 xxx rrr	xreg →reg
示例	
MOVED EAX, MM3 MOVED EBP, MM7	
0000 1111 0110 1110 oo xxx mmm	mem →xreg
示例	
MOVED MM3, DATA1 MOVED MM5, BIG_ONE	
0000 1111 0111 1110 oo xxx mmm	xreg →mem
示例	
MOVED DATA2, MM3 MOVED SMALL_POTS, MM7	
MOVEQ 传送四字	
0000 1111 0110 1111 11 xxx1 xxx2	xreg2 →xreg1
示例	
MOVEQ MM3, MM2 ; 把 MM2 复制到 MM3 MOVEQ MM7, MM3	
0000 1111 0111 1111 11 xxx1 xxx2	xreg1 →reg2
示例	
MOVEQ MM3, MM2 ; 把 MM3 复制到 MM2 MOVEQ MM7, MM3	
0000 1111 0110 1111 oo xxx mmm	mem →xreg
示例	
MOVEQ MM3, DATA1 MOVEQ MM5, DATA3	
0000 1111 0111 1111 oo xxx mmm	xreg →mem
示例	
MOVEQ DATA2, MM0 MOVEQ SMALL_POTS, MM3	
PACKSSDW 将带符号双字压缩为字	
0000 1111 0110 1011 11 xxx1 xxx2	xreg2 →xreg1
示例	
PACKSSDW MM1, MM2 PACKSSDW MM7, MM3	
0000 1111 0111 1011 oo xxx mmm	mem →xreg
示例	
PACKSSDW MM3, BUTTON PACKSSDW MM7, SOUND	

（续）

PACKSSWB　将带符号字压缩为字节	
0000 1111　0110 0011　11 xxx1 xxx2	xreg2 →xreg1
示例	
PACKSSWB MM1，MM2 PACKSSWB MM7，MM3	
0000 1111　0111 0011　oo xxx mmm	mem →xreg
示例	
PACKSSWB MM3，BUTTON PACKSSWB MM7，SOUND	
PACKUSWB　将不带符号字压缩为字节	
0000 1111　0110 0111　11 xxx1 xxx2	xreg2 →xreg1
示例	
PACKUSWB MM1，MM2 PACKUSWB MM7，MM3	
0000 1111　0111 0111　oo xxx mmm	mem →xreg
示例	
PACKUSWB MM3，BUTTON PACKUSWB MM7，SOUND	
PADD　带截断的加法	字节、字和双字
0000 1111　1111 11gg　11 xxx1 xxx2	xreg2 →xreg1
示例	
PADDB MM1，MM2 PADDW MM7，MM3 PADDD MM3，MM4	
0000 1111　1111 11gg　oo xxx mmm	mem →xreg
示例	
PADDB MM3，BUTTON PADDW MM7，SOUND PADDD MM3，BUTTER	
PADDS　带符号饱和的加法	字节和字
0000 1111　1110 11gg　11 xxx1 xxx2	xreg2 →xreg1
示例	
PADDSB MM1，MM2 PADDSW MM7，MM3	
0000 1111　1110 11gg　oo xxx mmm	mem →xreg
示例	
PADDSB MM3，BUTTON PADDSW MM7，SOUND	
PADDUS　不带符号饱和的加法	字节和字
0000 1111　1101 11gg　11 xxx1 xxx2	xreg2 →xreg1
示例	
PADDUSB MM1，MM2 PADDUSW MM7，MM3	
0000 1111　1101 11gg　oo xxx mmm	mem →xreg
示例	
PADDUSB MM3，BUTTON PADDUSW MM7，SOUND	

（续）

PAND 与	
0000 1111 1101 1011 11 xxx1 xxx2	xreg2 →xreg1
示例	
PAND MM1, MM2 PAND MM7, MM3	
0000 1111 1101 1011 oo xxx mmm	mem →xreg
示例	
PAND MM3, BUTTON PAND MM7, SOUND	
PANDN 与非	
0000 1111 1101 1111 11 xxx1 xxx2	xreg2 →xreg1
示例	
PANDN MM1, MM2 PANDN MM7, MM3	
0000 1111 1101 1111 oo xxx mmm	mem →xreg
示例	
PANDN MM3, BUTTON PANDN MM7, SOUND	
PCMPEQU 比较是否相等	字节、字和双字
0000 1111 0111 01gg 11 xxx1 xxx2	xreg2 →xreg1
示例	
PCMPEQUB MM1, MM2 PCMPEQUW MM7, MM3 PCMPEQUD MM0, MM5	
0000 1111 0111 01gg oo xxx mmm	mem →xreg
示例	
PCMPEQUB MM3, BUTTON PCMPEQUW MM7, SOUND PCMPEQUD MM0, FROG	
PCMPGT 比较是否大于	字节、字和双字
0000 1111 0110 01gg 11 xxx1 xxx2	xreg2 →xreg1
示例	
PCMPGTB MM1, MM2 PCMPGTW MM7, MM3 PCMPGTD MM0, MM5	
0000 1111 0110 01gg oo xxx mmm	mem →xreg
示例	
PCMPGTB MM3, BUTTON PCMPGTW MM7, SOUND PCMPGTD MM0, FROG	
PMADD 乘法和加法	
0000 1111 1111 0101 11 xxx1 xxx2	xreg2 →xreg1
示例	
PMADD MM1, MM2 PMADD MM7, MM3	
0000 1111 1111 0101 oo xxx mmm	mem →xreg
示例	
PMADD MM3, BUTTON PMADD MM7, SOUND	

（续）

PMULH　高位乘法	
0000 1111　1110 0101　11 xxx1 xxx2	xreg2 →xreg1
示例	
PMULH MM1，MM2 PMULH MM7，MM3	
0000 1111　1110 0101　oo xxx mmm	mem →xreg
示例	
PMULH MM3，BUTTON PMULH MM7，SOUND	
PMULL　低位乘法	
0000 1111　1101 0101　11 xxx1 xxx2	xreg2 →xreg1
示例	
PMULL MM1，MM2 PMULL MM7，MM3	
0000 1111　1101 0101　oo xxx mmm	mem →xreg
示例	
PMULL MM3，BUTTON PMULL MM7，SOUND	
POR　或	
0000 1111　1110 1011　11 xxx1 xxx2	xreg2 →xreg1
示例	
POR MM1，MM2 POR MM7，MM3	
0000 1111　1110 1011　oo xxx mmm	mem →xreg
示例	
POR MM3，BUTTON POR MM7，SOUND	
PSLL　左移	字、双字和四字
0000 1111　1111 00gg　11 xxx1 xxx2	xreg2 →xreg1
示例	
PSLLW MM1，MM2 PSLLD MM7，MM3 PSLLQ MM6，MM5	
0000 1111　1111 00gg　oo xxx mmm	mem →xreg 在存储器中移位计数
示例	
PSLLW MM3，BUTTON PSLLD MM7，SOUND PSLLQ MM2，COUNT1	
0000 1111　0111 00gg　11 110 mmm　data8	按计数移位 xreg 移位计数是 data8
示例	
PSLLW　MM3，2 PSLLD MM0，6 PSLLQ MM7，1	

（续）

PSRA 算术右移	字、双字和四字
0000 1111 1110 00gg 11 xxx1 xxx2	xreg2 →xreg1
示例	
PSRAW MM1, MM2 PSRAD MM7, MM3 PSRAQ MM6, MM5	
0000 1111 1110 00gg oo xxx mmm	mem →xreg 在存储器中移位计数
示例	
PSRAW MM3, BUTTON PSRAD MM7, SOUND PSRAQ MM2, COUNT1	
0000 1111 0111 00gg 11 100 mmm data8	按计数移位 xreg 移位计数是 data8
示例	
PSRAW MM3, 2 PSRAD MM0, 6 PSRAQ MM7, 1	
PSRL 右移	字、双字和四字
0000 1111 1101 00gg 11 xxx1 xxx2	xreg2 →xreg1
示例	
PSRLW MM1, MM2 PSRLD MM7, MM3 PSRLQ MM6, MM5	
0000 1111 1101 00gg oo xxx mmm	mem →xreg 在存储器中移位计数
示例	
PSRLW MM3, BUTTON PSRLD MM7, SOUND PSRLQ MM2, COUNT1	
0000 1111 0111 00gg 11 010 mmm data8	按计数移位 xreg 移位计数是 data8
示例	
PSRLW MM3, 2 PSRLD MM0, 6 PSRLQ MM7, 1	
PSUB 带截断的减法	字节、字和双字
0000 1111 1111 10gg 11 xxx1 xxx2	xreg2 →xreg1
示例	
PSUBB MM1, MM2 PSUBW MM7, MM3 PSUBD MM3, MM4	
0000 1111 1111 10gg oo xxx mmm	mem →xreg
示例	
PSUBB MM3, BUTTON PSUBW MM7, SOUND PSUBD MM3, BUTTER	

（续）

PSUBS　带符号饱和的减法	字节、字和双字
0000 1111　1110 10gg　11 xxx1 xxx2	xreg2 →xreg1
示例	
PSUBSB MM1，MM2 PSUBSW MM7，MM3 PSUBSD MM3，MM4	
0000 1111　1110 10gg　oo xxx mmm	mem →xreg
示例	
PSUBSB MM3，BUTTON PSUBSW MM7，SOUND PSUBSD MM3，BUTTER	
PSUBUS　不带符号饱和的减法	字节、字和双字
0000 1111　1101 10gg　11 xxx1 xxx2	xreg2 →xreg1
示例	
PSUBUSB MM1，MM2 PSUBUSW MM7，MM3 PSUBUSD MM3，MM4	
0000 1111　1101 10gg　oo xxx mmm	mem →xreg
示例	
PSUBUSB MM3，BUTTON PSUBUSW MM7，SOUND PSUBUSD MM3，BUTTER	
PUNPCKH　解压缩高位	字节、字和双字
0000 1111　0110 10gg　11 xxx1 xxx2	xreg2 →xreg1
示例	
PUNPCKH MM1，MM2 PUNPCKH MM3，MM4	
0000 1111　0110 10gg　oo xxx mmm	mem →xreg
示例	
PUNPCKH MM7，WATER PUNPCKH MM2，DOGGY	
PUNPCKL　解压缩低位	字节、字和双字
0000 1111　0110 00gg　11 xxx1 xxx2	xreg2 →xreg1
示例	
PUNPCKL MM1，MM2 PUNPCKL MM3，MM4	
0000 1111　0110 00gg　oo xxx mmm	mem →xreg
示例	
PUNPCKL MM7，WATER PUNPCKL MM2，DOGGY	
PXOR　异或	字节、字和双字
0000 1111　1110 1111　11 xxx1 xxx2	xreg2 →xreg1
示例	
PXOR MM2，MM3 PXOR MM4，MM7 PXOR MM0，MM1	

（续）

0000 1111 1110 1111 oo xxx mmm	mem →xreg
示例	
PXOR MM2, FROGS PXOR MM4, WALTER	
注：gg=00（字节）、=01（字）、=10（双字）和=11（四字）；xxx=MMX寄存器号，或MOVED指令的32位寄存器号；oo=模式；mmm=r/m域。	

编程实例

例14-13给出了一个MMX指令的简单程序，如果使用一般的微处理器指令完成这个任务将会耗时8倍之多。在这个例子中，一个1000字节数据的数组（BLOCKA）被加到第二个1000字节的数组（BLOCKB）上。结果被存储到第三个数组BLOCKC中。例14-13a采用传统汇编语言的方式来完成加法；例14-13b则使用MMX指令来完成加法。

例14-13a

```
; 该过程把 BLOCKA0 与 BLOCKB 相加,然后把和保存在 BLOCKC 中

BLOCKA DB     1000  DUP(?)
BLOCKB DB     1000  DUP(?)
BLOCKC DB     1000  DUP(?)

SUM    PROC   NEAR

       MOV    ECX,1000
       .REPEAT
              MOV     AL,BLOCKA [ECX-1]
              ADD     AL,BLOCKB [ECX-1]
              MOV     BLOCKC [ECX-1]
       .UNTILCXZ
       RET

SUM    ENDP
```

例14-13b

```
;该过程把 BLOCKA0 和 BLOCKB 相加,然后把和保存在 BLOCKC 中

BLOCKA DB      1000  DUP(?)
BLOCKB DB      1000  DUP(?)
BLOCKC DB      1000  DUP(?)

SUMM   PROC    NEAR
       MOV     ECX,125
       .REPEAT
              MOVEQ  MM0,QWORD PTR BLOCKA [ECX-8]
              PADDB  MM0,QWORD PTR BLOCKB [ECX-8]
              MOVEQ  QWORD PTR BLOCKC [ECX-8],MM0
       .UNTILCXZ
       RET

SUM    ENDP
```

如果仔细比较这两个程序，会发现MMX的版本执行了125遍LOOP循环中的3条指令；而传统的方式则执行了1000次，所以使用MMX速度会快7倍。这是因为一次会加8个字节（QWORD）。

14.6 SSE 技术概述

添加到 Pentium 4 指令系统中的最新指令类型是 **SIMD**（**Single-instruction, multiple data**）。就像其名字所指的一样，SIMD 就是一条指令操作多个数据，这和 MMX 指令的工作方式一样。MMX 指令集的对整型数进行操作，而 SIMD 指令集对浮点数和整型数进行操作。SIMD 扩展指令集第一次出现在 Pentium Ⅲ处理器上，称为 **SSE**（**streaming SIMD extensions**）。之后 SSE2 被加入到 Pentium 4 中，Pentium 4（90 纳米 E 型）采用的是更新的 SSE3 指令。SSE3 扩展同样存在于 Core2 微处理器中。

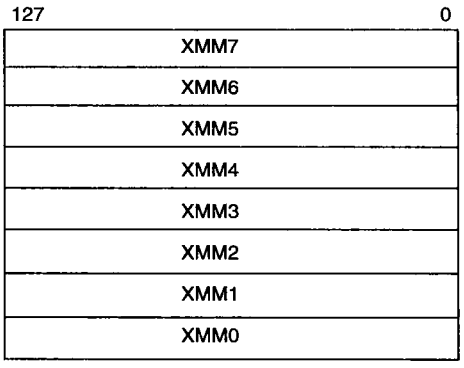


图 14-13 SSE 指令使用的 XMM 寄存器

MMX 与算术协处理器共享寄存器，而 SSE 指令集使用全新的、独立的寄存器组操作数据。图 14-13 给出了 8 个 128 位数据宽的寄存器组，与 SSE 指令一起工作。这些新的寄存器被称为 **XMM 寄存器**（$XMM_0 \sim XMM_7$），表示扩展的多媒体寄存器。一个新的关键字 OWORD 被引入，以容纳 128 位宽的数据，在 SSE 指令集的 OWORD PTR 中，一个 **OWORD**（**octalword**）指定了一个 128 位的变量。有时也用**双四字**（**quadword**）指定一个 128 位数字。

正如 MMX 寄存器能够容纳多种数据类型一样，SSE 部件的 XMM 寄存器也可以容纳多种数据类型。图 14-14 给出了多种 SSE 指令的 XMM 寄存器中能够出现的数据类型。一个 XMM 寄存器可以容纳 4 个单精度浮点数或 2 个双精度浮点数。XMM 寄存器也可以容纳 16 个 8 位整型、8 个 16 位整型、4 个 32 位整型或 2 个 64 位整型，这相当于两倍的 MMX 寄存器存储能力，因此使用 XMM 寄存器和 SSE 指令，整型操作的执行速度也提高了一倍。对于在 Pentium 4 或更新的微处理器上执行的新的应用，人们已经使用 SSE 指令集来代替 MMX 指令集。因为目前并不是所有的机器都是 Pentium 4，所以仍然要使用 MMX 指令集以兼容旧版本的机器。

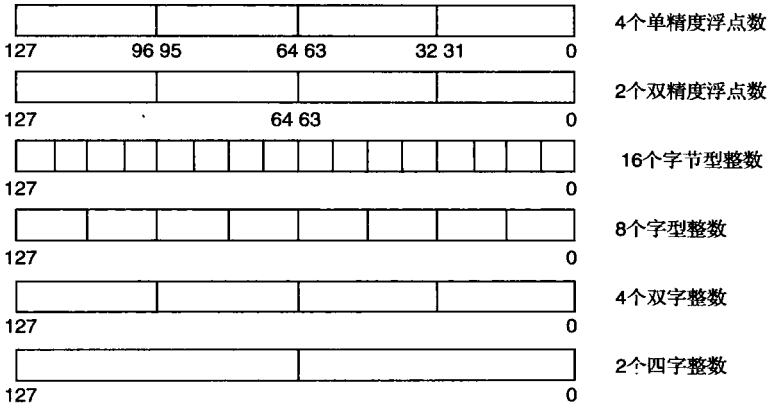


图 14-14 SSE2 和 SSE3 指令集的数据格式

14.6.1 浮点数

浮点数可以以分组或标量的形式进行操作，可以是单精度的也可以是双精度的。分组操作即同时对所有段执行；而标量形式仅对寄存器内容的最右段执行。图 14-15 给出了对 XMM 寄存器中的 SSE 数据进行分组和标量操作的两种方式。标量形式可以与算术协处理器完成的操作相比拟。操作码增加了 PS（packed single）、SS（scalar single）、PD（packed double）或 SD（scalar double）以实现所需要的指令。比如，乘法操作的操作码是 MUL，而 packed double 的操作码是 MULPD；scalar double 的操作码是 MULSD。单精度乘法是 MULPS 和 MULSS。也就是说，只要明白两个扩展字母的含义就可以轻松掌握

SSE 指令集。

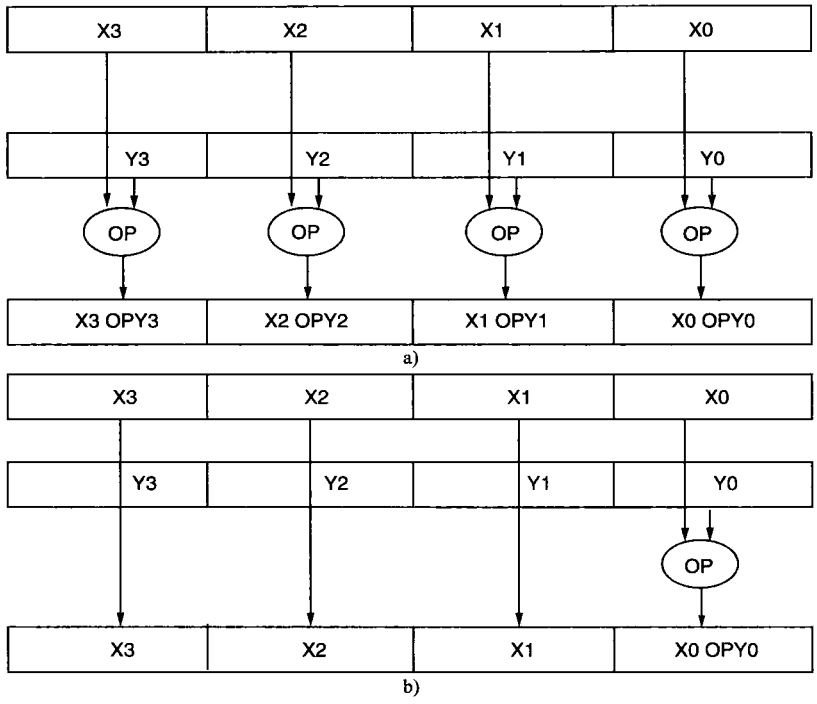


图 14-15 单精度浮点数的分组

a）操作 b）标量操作

14.6.2 指令集

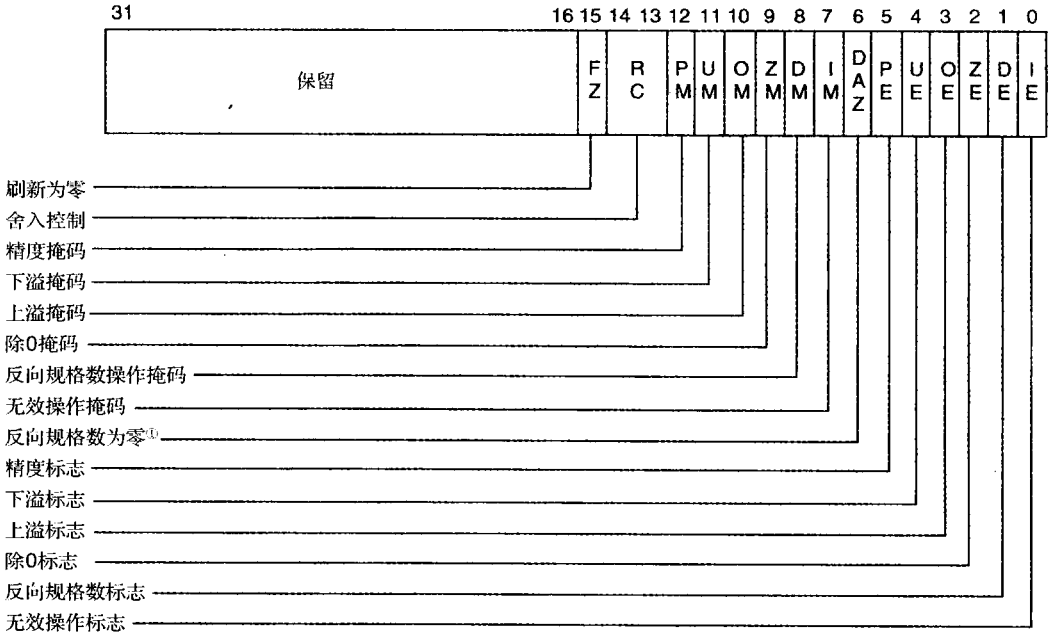
SSE 指令集中加入了一些新的指令。求倒数的指令常用于解决复杂的方程，但是浮点单元中没有倒数指令。在 SSE 指令集中新增了倒数（$1/n$）指令 RCP，它有 RCPPS、RCPSS、RCPPD、RCPSD 这些形式。还有求平方根倒数（$1/\sqrt{n}$）指令 RSQRT，它的形式有：RSQRTPS、RSQRTSS、RSQRTPD 和 RSQRTSD。

SSE 部件中的其他指令除了少数外基本上和微处理器及 MMX 单元的指令一样。附表 B 中列出了这些指令，但是没有加上扩展后缀（PS、SS、PD 和 SD）。再一次提醒，SSE2 和 SSE3 包含双精度操作，而 SSE 没有。以 P 开头的指令用于操作以字节、字、双字或四字为单位的整数。比如，PADDB XMM_0，XMM_1 指令用于将 XMM_0 中 16 字节大小的整数与 XMM_1 中 16 字节大小的整数相加。PADDW 用于 16 位整数相加，PADDD 用于双字相加，PADDQ 用于四字相加。因为 Intel 公司没有提供执行的时间，所以附录中没有关于这些指令执行时间的信息。

14.6.3 控制/状态寄存器

SSE 还包含控制/状态寄存器，即 MXCSR。图 14-16 给出了 SSE 部件的 MXCSR。注意，这个寄存器与在前面章节介绍的算术协处理器控制/状态寄存器非常类似。这个寄存器像控制寄存器为算术协处理器设置精度和四舍五入方式一样，为协处理器设置精度和四舍五入方式，并且提供了 SSE 部件的操作信息。

使用 LDMXCSR 或 FXRSTOR 指令将 SSE 控制/状态寄存器从内存之中装载，并使用 STMXCSR 或 FXSAVE 指令将 SSE 控制/状态寄存器存入内存。假设四舍五入控制（图 14-16 给出了四舍五入控制位的状态）要被更改为四舍五入到正无穷（RC = 10）。例 14-14 给出了改变控制/状态寄存器的四舍五入控制位的程序。



① 在Pentium 4 和Intel Xeon处理器中引入反向规格数为零的标志。

图 14-16 SSE 部件的控制/状态寄存器 MXCSR

例 14-14

```
;把四舍五入控制改为 10
    STMXCSR  CONTROL      ;保存控制/状态寄存器的内容
    BTS  CONTROL,14       ;设置位 14
    BTR  CONTROL,13       ;清除位 13
    LDMXCSR  CONTROL      ;重新装载控制/状态寄存器
```

14.6.4 编程实例

需要用一些程序例子来说明如何使用 SSE 部件。就像前面所提到的，SSE 部件允许对多个数据执行浮点或整型操作，假设一个电路有一个 1.0μF 的电容，其频率在 100Hz 和 10000Hz 之间，步长为 100Hz。计算电容容抗的公式为：

$$XC = \frac{1}{2\pi FC}$$

例 14-15 给出了使用 SSE 部件和单精度浮点数的过程，该过程根据以上的公式计算出了 100 个输出结果。例 14-15a）用 SSE 部件实现一次迭代计算 4 个 *XC*；例 14-15b）使用浮点协处理器一次迭代计算一个 *XC*；例 14-15c）则使用 C++ 。仔细查看每个例子的循环，第一个例子执行了 25 次循环；而第二个例子执行了 100 次循环。例 14-15a）中每次循环执行了 7 条指令（25 × 7 = 175），整个程序花费了 175 个指令周期；例 14-15b）每次循环迭代执行 8 条指令（100 × 8 = 800），整个程序需要 800 个指令周期；由于 SSE 采用了并行化，所以它完成计算所需要的时间比任何其他方法要少许多。例 14-15c）的 C++ 版本中在每个变量之前使用伪指令 `_declspec (align (16))` 来确定它们是否在内存中对齐。如果缺少它，程序将无法正常运行，因为 SSE 内存中的变量必须至少在四字边界（16）对齐。此最终版的运行速度比例 14-15b）快 4.5 倍。

例 14-15a)

```
;使用 SSE 单元

XC      DD      100 DUP(?)
CAP     DD      1.0E-6, 1.0E-6, 1.0E-6, 1.0E-6
F       DD      100.0, 200.0, 300.0, 400.0
INCR    DD      400.0, 400.0, 400.0, 400.0
PI      DD      4 DUP(?)

FXC     PROC    NEAR

        MOV     ECX,0
        FLDPI                                   ;取π
        FADD    ST,ST(0)                        ;2π
        FST     PI                              ;存储 4 个 2π
        FST     PI+4
        FST     PI+8
        FSTP    PI+12
        MOVUPS  XMM0,OWORD PTR PI               ;取 4 个 2πs
        MOVUPS  XMM1,OWORD PTR INCR             ;取累加值
        .REPEAT
              MOVUPS    XMM2,OWORD PTR F        ;装载频率
              MULPS     XMM2,XMM0               ;产生 2πF
              MULPS     XMM2,CAPS               ;2πFC
              RCPPS     XMM3,XMM2               ;取倒数
              MOVUPS    OWORD PTR XC[ECX],XMM3  ;存储 4 个 XC
              ADD       ECX,16                  ;移动指针
              ADDPS     OWORD PTR F,XMM1        ;累加 F
        .UNTIL ECX == 100
        RET

FXC     ENDP
```

例 14-15b)

```
;使用协处理器

XC      DD      100 DUP(?)
CAP     DD      1.0E-6
F       DD      0
INCR    DD      100.0

FXC1    PROC    NEAR

        FLDPI                                   ;取π
        FADD  ST,ST(0)                          ;求 2π
        FMUL  ST,CAP                            ;求 2πC
        MOV   ECX,0
        .REPEAT
              FLD   F                           ;取频率
              FADD  INCR                        ;加累加值
              FST   F                           ;为下一次保存
              FMUL  ST,ST(1)                    ;2πFC
              FLD1                              ;求倒数
              FDIVR
              FSTP  XC[ECX*4]                   ;保存 XC
              INC   ECX
        .UNTIL ECX == 100
        FCOMP                                   ;清除协处理器堆栈
        RET

FXC1    ENDP
```

例 14-15c)

```
void FindXC()
{
      //使用 C++ 和内嵌汇编程序进行浮点数计算

      __declspec(align(16)) float f[4] = {-300,-200,-100,0};
      __declspec(align(16)) float pi[4];
      __declspec(align(16)) float caps[4] = {1.0E-6, 1.0E-6, 1.0E-6, 1.0E-6      };
      __declspec(align(16)) float incr[4] = {400, 400, 400, 400};
      __declspec(align(16)) float Xc[100];
      _asm
      {
            Fldpi                                    ;求 2π
            fadd     st,st(0)
            fst      pi
            fst      pi+4
            fst      pi+8
            fstp     pi+12
            movaps   xmm0,oword ptr pi
            movaps   xmm1,oword ptr incr
            movaps   xmm3,oword ptr f
            mulps    xmm0,oword ptr caps             ;2πC
            mov      ecx,0
LOOP1:
            movaps   xmm2,xmm3
            addps    xmm2,xmm1
            movaps   xmm3,xmm2
            mulps    xmm2,xmm0
            rcpps    xmm2,xmm2                       ;倒数
            movaps   oword ptr Xc[ecx],xmm2
            add      ecx,16
            cmp      ecx,400
            jnz      LOOP1
      }
}
```

虽然例 14-15 利用浮点数实现乘法计算，但是 SSE 部件也支持整型操作。例 14-16 利用整型操作实现 BlockA 加 BlockB，其结果存入 BlockC。每一个块包含 4000 个 8 位数。例 14-16a）列出了一个汇编语言编写的过程，它利用微处理器的标准整型单元来实现求和操作，整个过程需要 4000 次迭代。

例 14-16a)

```
;该过程得到 4000 个 8 位数的和

SUMS   PROC   NEAR

       MOV    ECX,0
       .REPEAT
              MOV  AL,BLOCKA[ECX]
              ADD  AL,BLOCKB[ECX]
              MOV  BLOCKC,[ECX]
              INC  ECX
       .UNTIL ECX == 4000
       RET

SUMS   EMDP
```

例 14-16b)

```
;该过程使用 SSE 单元得到 4000 个 8 位数的和

SUMS1  PROC   NEAR
       MOV    ECX,0
       .REPEAT
              MOVDQA XMM0,OWORD PTR BLOCKA[ECX]
```

```
        PADDB   XMM0,OWORD PTR BLOCKB[ECX]
        MOVDQA  OWORD PTR BLOCKC[ECX]
        ADD     ECX,16
    .UNTIL ECX == 4000
    RET

SUMS1   ENDP
```

两个程序都产生4000个和。但第二个程序使用SSE部件执行了250次循环；而第一个程序却执行了4000次循环。因此SSE部件使得第2个程序要比第1个快15倍。注意MMX单元的PADDB指令是如何与SSE部件一起使用的。除了寄存器不同，SSE部件使用的命令与MMX相同。MMX使用64位宽的MM寄存器；SSE使用128位宽的XMM寄存器。

14.6.5 优化

Visual C++的编译器能够优化SSE部件，但是它不能优化本章中的例子。如果可以用SSE部件实现一个等式，那么编译器会优化语句中的单一等式。在上边的例子中，优化器看不到块操作的程序，虽然块操作也可以被优化。要想能够优化这类并行操作，还要等待编译器进一步的发展和扩充，高效程序还需要手写的汇编语言以达到优化的目的。这一点对于SSE部件特别重要。

14.7 小结

1）算术协处理器和微处理器并行工作，这意味着微处理器和协处理器可以同时执行各自的指令。

2）协处理器使用的数据类型包括带符号的整数、浮点数以及二进制编码的十进制数（BCD）。

3）协处理器使用3种类型的整数：字型（16位）、短整型（32位）和长整型（64位）。对于带符号的整数，正数以原码形式表示，负数则以2的补码形式来表示。

4）BCD数存储为一个18位数，占用10个字节的内存空间，最高有效字节包含符号位，其余9个字节存储一个18数位压缩的BCD数。

5）协处理器支持3种类型的浮点数：单精度型（32位）、双精度型（64位）和扩展精度型（80位）。浮点数由3部分组成：符号、阶码和有效数字。在协处理器中，指数有一个常数偏置，规格化数的整数位不存储于有效数字中，而扩展精度形式例外。

6）十进制数转换为浮点数的步骤如下：

a）转换为二进制数。

b）规格化二进制数。

c）指数加上偏移量。

d）以浮点形式存储该数。

7）浮点数转换为十进制数的步骤如下：

a）从阶码中减去偏移量。

b）将此数非规格化。

c）将其转换为十进制数。

8）80287使用I/O空间来执行一些指令。此空间对于程序是不可见的，但可以由80286/80287系统内部使用。在包含80287的系统中，这些16位I/O端口地址（00F8H～00FFH）不能用于I/O数据传送。80387、80486/7、Pentium～Core2则使用I/O端口地址800000F8H～800000FFH。

9）协处理器包含一个状态寄存器，它的状态值表示协处理器忙或空闲、紧跟在比较或测试指令后的条件、栈顶位置以及错误位的状态。FSTSW AX指令，后带SAHF指令，经常同条件跳转指令一起用于测试协处理器的某些条件。

10）协处理器的控制寄存器的控制位用于选择无穷大控制、舍入控制、精度控制和错误屏蔽控制。

11）以下伪指令常与协处理器一起用于存储数据：DW（定义字）、DD（定义双字）、DQ（定义四字）和DT（定义10字节）。

12）协处理器使用堆栈在它自己与内存之间传送数据。通常，数据被装入栈顶或者从栈顶移出到内存中存储。

13）协处理器的所有内部数据总是按80位扩展精度浮点形式表示。数据以其他任何形式出现的惟一时刻是被存入内存或从内存中取出时。

14）协处理器的寻址方式包括传统的堆栈寻址、寄存器寻址、带弹出的寄存器寻址以及存储器寻址方式。堆栈寻址意味着ST中的数据为源操作数，ST（1）中的数据为目的操作数，当ST中原值弹出后结果值存入ST中。

15）协处理器的算术运算包括：加法、减法、乘法、除法及求平方根等运算。

16）协处理器指令系统中有超越函数指令，它们是：求部分正切、求部分反正切、2^X-1、$Ylog_2X$ 和 $Ylog_2(X+1)$ 运算。80387、80486/7 和 Pentium～Core2 中还包括正弦和余弦函数。

17）存储于协处理器内部的常数包括 +0.0、+1.0、π、$\log_2 10$、$\log_2 \varepsilon$、$\log_{10} 2$ 和 $\log_e 2$ 等。

18）80387 与80386 微处理器并用，80487SX 与80486SX 微处理器并用，但80486DX 和 Pentium～Core2 拥有内置的算术协处理器。早期版本的协处理器所执行的指令在这些协处理器上仍然可以使用。除这些指令外，80387、80486/7 和 Pentium～Core2 还有求正弦和余弦函数的指令。

19）Pentium Pro～Core2 中包含两条新的浮点指令：FCMOV 和 FCOMI。FCMOV 指令为条件传送指令，FCOMI 指令执行与 FCOM 同样的任务，但它还将浮点标志置于系统的标志寄存器中。

20）MMX 扩展使用算术协处理器的寄存器 MM_0～MM_7。因此协处理器软件和 MMX 软件不要同时使用这些寄存器，这一点是非常重要的。

21）MMX 扩展指令按字节（一次8 个字节）、字（一次4 个字）、双字（一次2 个双字）以及四字来执行算术和逻辑运算。所执行的运算包括加法、减法、乘法、与、或、异或以及与非运算。

22）MMX 部件和 SSE 部件都采用了 SIMD 技术来完成用一条指令对多个数据的并行操作。SSE 部件完成对整数和浮点数的操作。SSE 部件中的寄存器为 128 位宽，可以同时容纳（SSE2 或更新的 SSE 部件）16 字节或4 个单精度浮点数。SSE 部件包括寄存器 XMM_0～XMM_7。

23）新的 Pentium 4 应用程序应该包括 SSE 指令，以取代 MMX 指令。

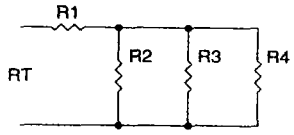
24）OWORD 指针寻址 128 位宽的数，这个数被称为八字或双四字。

14.8 习题

1. 列出协处理器从内存中装入或存储于内存中的3 种数据类型。
2. 列出3 种整数类型、存储范围以及每种数据类型分配的位数。
3. 协处理器是如何将 BCD 数存入内存的?
4. 列出协处理器使用的3 种浮点数类型，以及分配给每种类型的二进制位数。
5. 将下列十进制数转换为单精度浮点数：
   （a）28.75
   （b）624
   （c）-0.615
   （d）+0.0
   （e）-1000.5
6. 将下列单精度浮点数转换为十进制数：
   （a）11000000 11110000 00000000 00000000
   （b）00111111 00010000 00000000 00000000
   （c）01000011 10011001 00000000 00000000
   （d）01000000 00000000 00000000 00000000
   （e）01000001 00100000 00000000 00000000
   （f）00000000 00000000 00000000 00000000
7. 说明当执行常规的微处理器指令时，协处理器进行什么操作。
8. 说明当协处理器执行指令时，微处理器进行什么操作。
9. 状态寄存器中 C_3～C_0 位的作用是什么?
10. FSTSW AX 指令实现什么样的操作?
11. 状态寄存器中 IE 位的作用是什么?
12. SAHF 和条件跳转指令如何决定栈顶（ST）中数据是否等于寄存器 ST（2）中的数据?
13. 在80X87 中是如何选择舍入模式的?
14. 哪条协处理器指令使用微处理器的 AX 寄存器?
15. 哪些 I/O 端口是留给80287 协处理器使用的?
16. 数据是如何存储在协处理器内部的?
17. 什么是 NAN?
18. 一旦协处理器被复位，则栈顶寄存器号为_____。
19. 关于控制寄存器的舍入控制位，术语“截断”（chop）是什么含义?
20. 仿射无穷大控制和投射无穷大控制的区别是什么?
21. 哪条微处理器指令为协处理器形成操作码?
22. FINIT 指令对所有协处理器操作选择_____精度。
23. 使用汇编语言伪操作码写出实现以下要求的语句：
   （a）将23.44 存入名为 FROG 的双精度浮点存储单元中。
   （b）将-123 存入名为 DATA3 的32 位带符号整型存储单元中。
   （c）将-23.8 存入名为 DATA1 的单精度浮点存储单元中。
   （d）保留一个名为 DATA2 的双精度存储单元。
24. 描述 FST DATA 指令是如何操作的，假设 DATA 定义为64 位的存储单元。
25. FILD DATA 指令实现什么功能?
26. 选择一条指令将寄存器3 中的内容加到栈顶。
27. 描述 FADD 指令的操作。
28. 选择一条指令，实现栈顶内容减去寄存器2 中的内容，并将结果存于寄存器2 中。
29. FBSTP DATA 指令的功能是什么?
30. 正向除法和反向除法有什么区别?
31. Pentium Pro 中的 FCOMI 指令有什么作用?
32. Pentium Pro 中的 FCMOVB 指令实现什么功能?
33. 执行 FCMOV 指令的前提条件是什么?
34. 设计一个过程来求出单精度浮点数的倒数，此数在

EAX 中传给该过程，并且必须将倒数返回 EAX 中。

35. FTST 指令与 FXAM 指令的区别是什么？
36. 解释 F2XM1 指令进行什么计算。
37. 执行 FSQRT 指令后，协处理器状态寄存器的哪一位应该被测试？
38. 哪条协处理器指令将 π 压入栈顶？
39. 哪条协处理器指令将 1.0 压入栈顶？
40. 执行 FFREE ST（2）将实现什么功能？
41. 哪条指令存储环境参数？
42. FSAVE 指令存储什么？
43. 编写一个过程，求矩形面积 A = L × W。存储单元为单精度浮点单元 A、L 和 W。
44. 编写一个过程，求容抗 $XC = 1/(2\pi FC)$，存储单元为单精度浮点单元 *XC*、*F* 和 *C*。
45. 编写一个过程，产生一个从整数 2 到 10 的平方根表，结果必须为单精度浮点数，存入名为 ROOTS 的数组中。
46. 在程序中，何时使用 FWAIT 指令？
47. FSTSW 和 FNSTSW 指令有什么区别？
48. 如图 14-17 所示，已知串/并联电路图及方程，编写一段程序求总电阻值，其中 R_1、R_2、R_3 和 R_4 为单精度数，并将结果存入单精度单元 RT 中。



$$RT = R1 + \frac{1}{\frac{1}{R2}+\frac{1}{R3}+\frac{1}{R4}}$$

图 14-17 串/并联电路图

49. 编写一个过程，求单精度浮点数的余弦值。角度以度来表示，存于 EAX 中，并将余弦值返回 EAX 中。回忆 FCOS 指令求以弧度表示的角的余弦的过程。
50. 已知 2 个双精度浮点数据数组（$ARRAY_1$ 和 $ARRAY_2$），每个数组包含 100 个元素。编写一个过程，求 $ARRAY_1$ 中元素和 $ARRAY_2$ 中元素的乘积，并将双精度浮点数结果存入第 3 个数组 $ARRAY_3$ 中。
51. 编写一个过程，将寄存器 EBX 中的单精度浮点数乘以 π，并将单精度浮点数结果存入寄存器 EBX 中。要求使用存储器实现此任务。
52. 编写一个过程求 X 的 Y 次幂。参数由 EAX = X 和 EBX = Y 传给过程，结果存入 ECX 中。
53. 已知 $LOG_{10}X = (LOG_2 10)^{-1} \times LOG_2 X$，编写一个名为 LOG_{10}的过程，求栈顶中的数的以 10 为底的对数值，在过程的末尾返回对数值存于栈顶。
54. 应用 53 题中的过程求解等式：分贝增益 = $20LOG_{10}(V_{OUT}/V_{IN})$。其中 V_{OUT}和 V_{IN}均为包含 100 个单精度数的数组，分贝增益值存入第 3 个数组 DBG 中。
55. 什么是 Pentium ~ Core2 微处理器的 MMX 扩展？
56. EMMS 指令的作用是什么？
57. MM_0 ~ MM_7 寄存器在微处理器中的什么地方？
58. 什么是带符号的饱和？
59. 什么是不带符号的饱和？
60. 如何用一条指令将所有 MMX 寄存器中的内容存入内存？
61. 编写一个简短的程序，使用 MMX 指令实现两个字长度的数相乘，两个数分别位于各包含 256 个字的数组中，并将 32 位结果存入第三个数组中。
62. 什么是 SIMD 指令？
63. 什么是 SSE 指令？
64. XMM 寄存器是_____位宽？
65. 单 XMM 寄存器可以容纳_____单精度浮点数？
66. 单 XMM 寄存器可以容纳_____字节整数？
67. 什么是 OWORD？
68. 算术协处理器的浮点数指令可以和 SSE 指令同时执行吗？
69. 编写一个 C++ 函数（使用内嵌汇编代码），该函数计算（使用标量 SSE 指令和浮点数指令）并且返回一个表示振荡频率的单精度数，L 和 C 作为参数传给该函数，计算公式如下：

$$Fr = \frac{1}{2\pi\sqrt{LC}}$$

第15章 总线接口

引言

许多应用需要了解位于PC机内部的总线系统。有时，PC机的主板在工业应用中被用作核心系统。这些系统常需要连接到主板的某一条总线上的用户接口。本章介绍ISA（industry standard architecture，工业标准结构）总线、VESA局部总线、PCI（peripheral component interconnect，外围部件互联）总线、USB（universal serial bus，通用串行总线）以及AGP（advanced graphics port，高级图形端口）。同时对其中的许多总线系统提供了一些简单接口作为设计向导。

虽然未来的PC机可能不使用并口和串行通信口，但是在以下的章节中我们仍然对它们进行了阐述。它们是个人计算机的第一代I/O端口并且占据历史相当长的时间，但是通用串行总线几乎已经取代了它们。

目的

读者学习完本章后将能够：

1）详述并行端口和串行端口、ISA、AGP、PCI及PCI Express总线的引脚和信号总线。

2）设计与并行端口、串行端口、ISA总线及PCI总线相连的简单接口。

3）编程置于主板上的接口，使主板与ISA总线与PCI总线相连。

4）描述USB的操作并设计一些短程序来传送数据。

5）解释AGP如何提高图形子系统的效率。

15.1 ISA总线

ISA总线（industry standard architecture，工业标准结构总线）在与IBM兼容的PC机系统刚起步时就已出现了（约1982年）。实际上，任何早期PC机中的功能卡均可插入最先进的基于Pentium 4的计算机中，并且正常使用，只要这台计算机拥有ISA插槽。这是因为在所有这些计算机中都有ISA总线接口，从而与早期的PC机仍然兼容。ISA总线几乎在家用PC机上消失了，但是在许多工业应用上仍可以看到它的身影，这也是这里要介绍ISA总线的原因。仍然在工业中应用的主要原因是接口的成本低和现存的接口卡槽的数量。这种状况最终将改变。

15.1.1 ISA总线的发展

ISA总线已经不同于它的早期版本。这些年以来，ISA总线已从最初的8位标准总线发展为当今在一些系统中使用的16位标准总线。最后一代使用ISA总线的是Pentium Ⅲ计算机，随着Pentium 4的诞生，ISA总线也逐渐寿终正寝。在发展过程中，甚至出现过一种被称为**EISA（extended ISA，扩展ISA）**总线的32位标准总线，但现在已基本消失了。现在，在大多数PC机中仍保留的是主板上的一个ISA插槽，既可以插入8位ISA卡，又可以插入16位ISA印刷电路板。32位的印刷电路板则经常是PCI卡，或是早一点的基于80486机器的VESA卡。ISA总线在家用电脑上几乎全部消失了，但是作为一种特殊的要求，它仍出现在大多数的主板上。如今在许多工业应用上仍可以看到它的影子，但它的使用范围非常有限。

15.1.2 8位ISA总线输出接口

图15-1给出了存在于所有PC机主板上的8位ISA连接器（也许和16位

计算机背面

引脚#	焊接面	元件面
1	GND	IO CHK
2	RESET	D7
3	+5V	D6
4	IRQ9	D5
5	-5V	D4
6	DRQ2	D3
7	-12V	D2
8	OWS	D1
9	+12V	D0
10	GND	IO RDY
11	$\overline{\text{MEMW}}$	AEN
12	$\overline{\text{MEMR}}$	A19
13	$\overline{\text{IOW}}$	A18
14	$\overline{\text{IOR}}$	A17
15	$\overline{\text{DACK3}}$	A16
16	DRQ3	A15
17	$\overline{\text{DACK1}}$	A14
18	DRQ1	A13
19	$\overline{\text{DACK0}}$	A12
20	CLOCK	A11
21	IRQ7	A10
22	IRQ6	A9
23	IRQ5	A8
24	IRQ4	A7
25	IRQ3	A6
26	$\overline{\text{DACK2}}$	A5
27	T/C	A4
28	ALE	A3
29	+5V	A2
30	OSC	A1
31	GND	A0

图15-1 8位ISA总线

连接器组合在一起)。ISA 总线连接器包括完整的经多路分离的地址总线（A_{19}～A_0），专为 1MB 的 8088 系统使用；8 位数据总线（D_7～D_0）以及 4 个控制信号$\overline{MEMR}$、$\overline{MEMW}$、$\overline{IOR}$和$\overline{IOW}$，用于控制印刷电路板上的 I/O 端口和存储器。由于 ISA 卡工作频率是 8MHz，所以现在很少将存储器加在 ISA 总线卡上。可能在一些 ISA 卡上使用 EPROM 或闪速存储器来存储初始化信息，但决不会使用 RAM 来存储。

对 I/O 接口有用的其他信号线是**中断请求线（interrupt request line）**IRQ_2～IRQ_7。注意，在现代的系统中，IRQ_2 又被接到 IRQ_9 上，如图 15-1 中连接器上的标识所示。此连接器还提供 DMA 通道 0～3 的控制信号，**DMA 请求输入（DMA request input）**被标识为 DRQ_1～DRQ_3，**DMA 响应输出（DMA acknowledge output）**被标识为$\overline{DACK0}$～$\overline{DACK3}$。注意，DRQ_0 输入引脚并不存在，因为在早期 PC 机中，$DACK_0$ 输出作为一个刷新信号用来刷新可能位于 ISA 卡上的任何 DRAM。现在，此输出引脚是一个 15.2μs 的时钟信号。连接器上的其余引脚用于电源和复位。

假使有一组 4 个 8 位锁存器需要连接到 PC 机上用于传输 32 位并行数据，则这个任务可以通过向维克多电子公司或其他公司购买 ISA 接口卡（产品号 4713-1）来完成。此卡不仅提供 ISA 总线使用的边缘连接器，其背面还为接口连接器准备了空间。可把一个 37 引脚的超小型的 D 型连接器插在卡的背面，用来把 32 位数据传输到外部设备。

图 15-2 给出了一个提供 32 位并行 TTL 数据的 ISA 总线的简单接口。此例说明了一些关于任一系

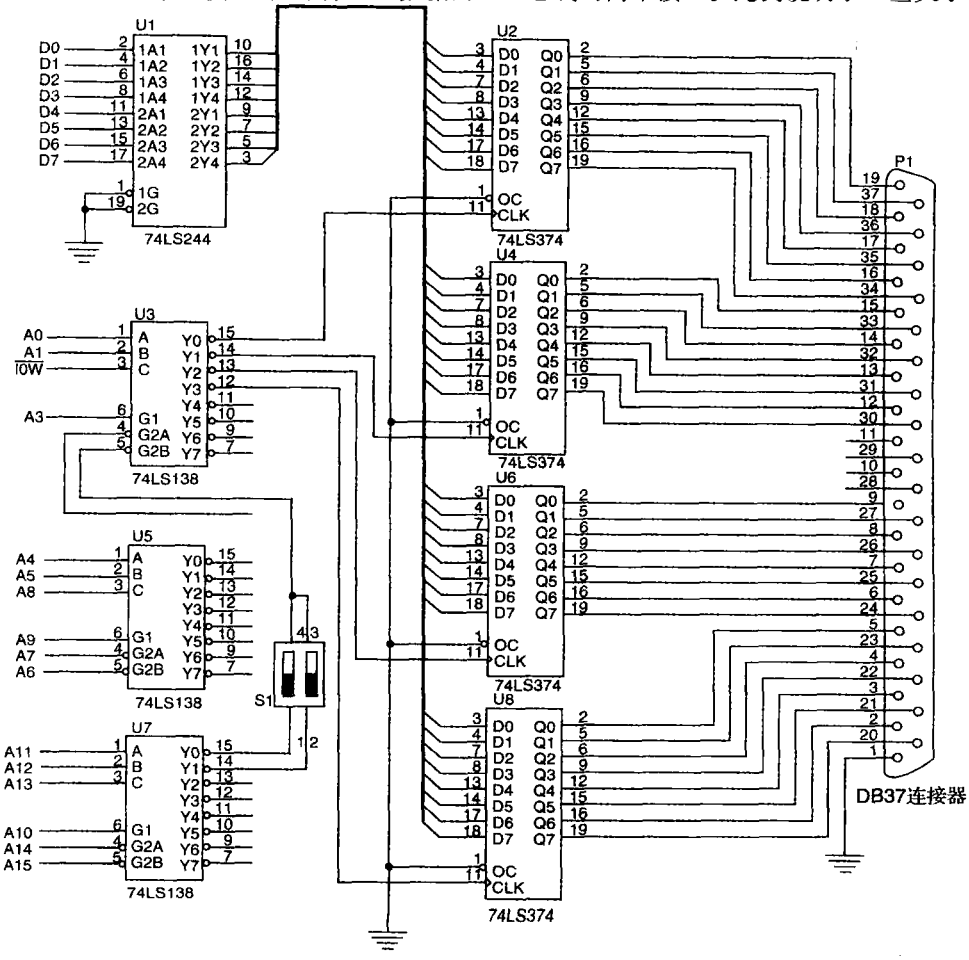


图 15-2 连接到 8 位 ISA 总线上的 32 位并行端口

统接口的重要注意点。首先，非常重要的是ISA总线的负载必须是低功耗（LS）TTL负载。在此电路中，74LS244缓冲器用于减轻数据总线上的负载。如果没有74LS244缓冲器，则系统将给数据总线加上4个负载。如果所有总线卡均提供如此沉重的负载，则系统将无法正常工作（或者根本不能工作）。

此电路中ISA卡的输出由标识为P_1的带37个引脚的连接器提供。电路的输出连到P_1上，P_1的地线已经接好。必须给外界提供地线，否则并行端口上的TTL数据将不起作用。如果需要，每个74LS374锁存器的输出控制引脚（$\overline{OC}$）也可以不与地线相连，而是接到P_1的4个剩余引脚上，这样就允许外部电路控制锁存器的输出。

一个小的DIP开关被接到U_7的2个输出上。这样，如果与其他的卡发生地址冲突，则可以改变地址。但这种情况一般不太可能发生，除非打算在同一系统中使用2个ISA卡。此系统中地址线A_2没有被译码，因此这里它是一个无关项。参见表15-1中每个锁存器的地址及S_1的每一位置。注意，在同一时间，2个开关中只能有一个闭合，而且对于每个开关设置，每个端口有2个可能的地址，这是因为A_2没有连接。

表15-1 图15-2的I/O端口分配

DIP开关	锁存器U_2	锁存器U_4	锁存器U_6	锁存器U_8
1-4闭合	0608H或060CH	0609H或060DH	060AH或060EH	060BH或060FH
2-3闭合	0E08H或0E0CH	0E09H或0E0DH	0E0AH或0E0EH	0E0BH或0E0FH

在PC机中，ISA总线被设计工作在I/O地址0000H～03FFH。ISA卡有的可以，有的不可以超出这个地址范围工作，这取决于主板的型号以及主板的制造商。较新的系统常允许ISA的I/O端口地址高于03FFH，而较早的系统则不允许。此例中的端口对于某些系统可能需要经过修改。一些较早的卡只译码I/O端口地址为0000H～03FFH，如果03FFH以上的端口地址冲突，则可能发生地址冲突。此例中，由3个74LS138译码器完成端口的译码。如果使用可编程逻辑器件，则端口的译码将更有效和更经济。

图15-3中的电路对图15-2进行了修改，使用PLD为系统译码地址。注意，地址位A_{15}～A_4由PLD译码，开关接到PLD的2个输入上。这一修改允许每个锁存器有4个不同的I/O端口地址，使得电路更加灵活。表15-2给出了由开关1-4和开关2-3所选择的端口号。例15-1对于PLD的编程实现了表15-2中的端口分配。

表15-2 图15-3中的端口分配

S_2	S_1	U_3	U_4	U_5	U_6
闭合	闭合	0300H	0301H	0302H	0303H
闭合	断开	0304H	0305H	0306H	0307H
断开	闭合	0308H	0309H	030AH	030BH
断开	断开	030CH	030DH	030EH	030FH

注：On为开关闭合（0），Off为断开（1）。

例15-1

```
--图15-3译码器的VHDL代码
library ieee;
use ieee.std_logic_1164.all;
entity DECODER_15_3 is
port(
     IOW,A14, A13, A12, A11, A10, A9, A8, A7,
          A6, A5, A4, A3, A2, A1, A0, S1,S2: in STD_LOGIC;
     U3, U4, U5, U6:out STD_LOGIC
);
end;
```

```
architecture V1 of DECODER_15_3 is
begin
    U3 < = IOW or A14 or A13 or A12 or A11 or A10 or not A9 or not A8 or A7 or A6 or A5 or A4 or A1 or A0
          or( S2 or S1 or A3 or A2 ) and(S2 or not S1 or A3 or not A2) and (not S2 or S1 or not A3 or
          A2) and (not S2 or not S1 or not A3 or not A2);
    U4 < = IOW or A14 or A13 or A12 or A11 or A10 or not A9 or not A8 or A7 or A6 or A5 or A4 or A1 or
          not A0 or( S2 or S1 or A3 or A2 ) and(S2 or not S1 or A3 or not A2) and (not S2 or S1 or not
          A3 or A2) and (not S2 or not S1 or not A3 or not A2);
    U5 < = IOW or A14 or A13 or A12 or A11 or A10 or not A9 or not A8 or A7 or A6 or A5 or A4 or not A1
          or A0 or( S1 or S2 or A3 or A2 ) and(S2 or not S1 or A3 or not A2) and (not S2 or S1 or not
          A3 or A2) and (not S2 or not S1 or not A3 or not A2);
    U6 < = IOW or A14 or A13 or A12 or A11 or A10 or not A9 or not A8 or A7 or A6 or A5 or A4 or not A1
          or not A0 or( S1 or S2 or A3 or A2 ) and(S2 or not S1 or A3 or not A2) and (not S2 or S1 or
          not A3 or A2) and (not S2 or not S1 or not A3 or not A2);
end V1;
```

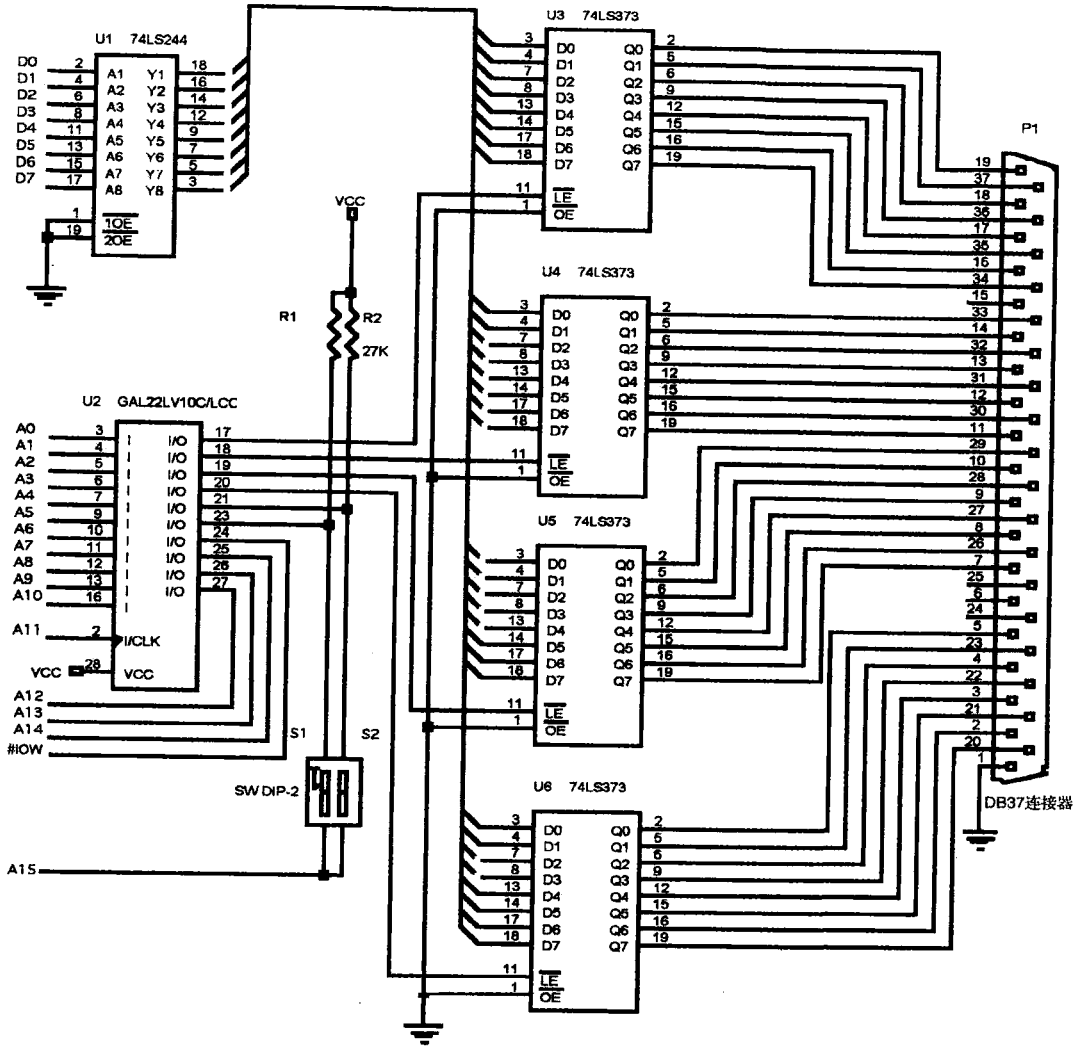


图 15-3 ISA 总线的 32 位并行端口

在例 15-1 中，对于 I/O 端口 0300H，当 2 个开关都处于断开位置时，请注意第 1 个乘积项（U_3）是如何在译码器的输出端产生逻辑 0 的。它还根据开关的设置状态，为 U_3 产生一个用于 I/O 端口 304H、308H 或 30CH 的时钟信号。根据开关的设置状态，乘积项（U_4）对 I/O 端口 301H、305H、309H 或 30DH 有效。再次参考表 15-2 对不同开关进行设置时全部的端口分配。A_{15} 与开关的底部连接，因它没有被译码，此电路还将触发其他 I/O 地址的锁存器，I/O 地址 830XH 也将为该锁存器产生时钟信号。

例 15-2 给出了两个 C++ 函数，它们将一个整型数传送给 32 位端口。这两个函数都发送数据给端口，第一个函数效率更高，但第二个函数也许可读性更强（例 15-2c 给出了例 15-2b 的反汇编形式）。有两个参数传递给函数：一个是要发送给端口的数据，另一个是端口基地址。基地址为 0300H、0304H、0308H 或 030CH，它们必须与图 15-3 的开关设置相匹配。

例 15-2a

```
void OutPort(int address,int data)
{
      _asm
      {
            mov  edx,address
            mov  eax,data
            mov  ecx,4
OutPort1:
            out  dx,al              ;输出 8 位
            shr  eax,8              ;取下一个 8 位段
            inc  dx                 ;寻址下一个端口
            loop OutPut1            ;重复 4 次
      }
}
```

例 15-2b

```
Void OutPut(int address ,int data)
{
      for(  int a = address; a < address +4; a++)
      {
            _asm
            {
                  mov edx,a
                  mov eax,data
                  out dx,al
            }
            data > > =8;                  //取下一个 8 位段
      }
}
```

例 15-2c

```
//例 15-2b 的反汇编形式
            for (int a =address; a <address +4; a++)
00413823  mov       eax, dword ptr [address]
00413826  mov       dword ptr [a], eax
00413829  jmp       CSSEDlg:: OutPrt +54h (413834h)
0041382B  mov       eax, dword ptr [a]
0041382E  add       eax, 1
00413831  mov       dword ptr [a], eax
00413834  mov       eax, dword ptr [address]
```

```
00413837  add       eax, 4
0041383A  cmp       dword ptr [a], eax
0041383D  jge       CSSEDlg:: OutPrt +71h (413851h)
      {
          _ asm
          {
                mov edx, a
0041383F  mov       edx, dword ptr [a]
                mov eax, data
00413842  mov       eax, dword ptr [data]
                out dx, al
00413845  out       dx, al
          }
          data > > =8;              //取下一个 8 位段
00413846  mov       eax , dword  ptr [data]
00413849  sar       eax, 8
0041384C  mov       dword ptr [data], eax
      }
    0041384F  jmp       CSSEDlg:: OutPrt +4Bh (41382Bh)
```

15.1.3 8位ISA总线输入接口

为说明ISA总线的输入接口，在图15-4中，一对ADC804模/数转换器被接到ISA总线上。一个9引脚的DB_9连接器与转换器相连。译码I/O端口地址的任务更加复杂，因为每个转换器需要一个写脉冲来启动转换；而且一旦信号由模拟输入数据转换为数字信号，还需要一个读脉冲去读这个数字信号；另外还需要一个脉冲选择$\overline{\text{INTR}}$输出。注意，$\overline{\text{INTR}}$输出被接到数据总线D_0位，当$\overline{\text{INTR}}$被输入给微处理器，AL最右位即被测试，以检查转换器是否处于“忙”状态。

像以前一样，要特别注意，连接到ISA总线上的负载为一个。表15-3中给出在例15-3中由PLD译码的I/O端口分配。在下面的例子中假设使用标准的ISA总线，仅包括地址线$A_0 \sim A_9$。

表15-3 图15-4的I/O端口分配

设 备	端 口 号
开始ADC（U_3）	0300H
读ADC（U_3）	0300H
读INTR（U_3）	0301H
开始ADC（U_4）	0302H
读SDC（U_4）	0302H
读INTR（U_4）	0303H

例15-3

```
- -图 15-4 译码器的 VHDL 代码

library ieee;
use ieee.std_logic_1164.all;

entity DECODER_15_4 is

port(
     IOW,IOR, A9, A8, A7, A6, A5, A4, A3, A2, A1, A0, :in STD_LOGIC;
     A,B,C,D,E,F:out STD_LOGIC
);

end;

architecture V1 of DECODER_15_4 is

begin
     A < = not A9 or not A8 or A7 or A6 or A5 or A4 or A3 or A2or A1 or A0 or IOR;

     B < = not A9 or not A8 or A7 or A6 or A5 or A4 or A3 or A2or A1 or A0 or IOW;

     C < = not A9 or not A8 or A7 or A6 or A5 or A4 or A3 or A2or A1 or not A0 or IOR;

     D < = not A9 or not A8 or A7 or A6 or A5 or A4 or A3 or A2or not A1 or not A0 or IOR;
```

```
    E < = not A9 or not A8 or A7 or A6 or A5 or A4 or A3 or A2or not A1 or A0 or IOR;
    F < = not A9 or not A8 or A7 or A6 or A5 or A4 or A3 or A2or not A1 or A0 or IOW;
end V1;
```

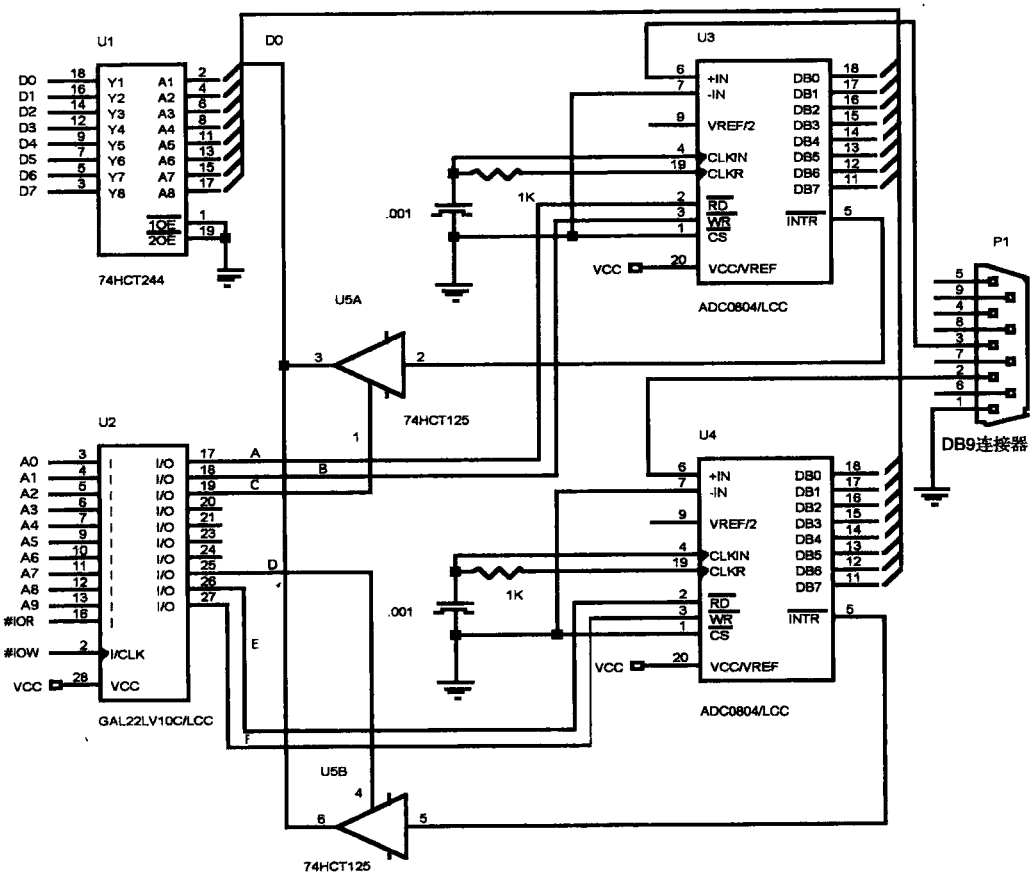


图 15-4 接到 ISA 总线上的一对模/数转换器

例 15-4 列出了一个可以用来读 ADC U_3 或 U_4 的函数。对于 U_3，给函数传递一个 0；对于 U_4，给函数传递一个 1 作为参数，以产生地址。这个函数通过对转换器的写操作来启动转换器，然后等待，直到$\overline{\text{INTR}}$引脚回到逻辑 0 为止，表明在数据被读出并被函数返回一个字符之前，转换已经完成。

例 15-4

```
char ADC(int address)
{
      char temp =1;
      if(address)
         address =2;
      address + =0x300;
      _asm
      {                    ;开始转换
           mov  edx,address
           out  dx,al
      }
      while(temp)          //忙则等待
```

```
    {
        _asm
        {
                mov edx,address
                inc edx
                in al,dx
                mov temp,al
                and al,1
        }
    }
    _asm
    {                       ;取数据
        mov edx,address
        in al,dx
        mov temp,al
    }
    return temp;
}
```

15.1.4 16位ISA总线

16位与8位ISA总线的惟一区别在于：8位连接器的后面增加了一个额外的连接器。16位的ISA卡包括两个边缘连接器：一个插入原来的8位连接器，另一个插入新的16位连接器。图15-5给出了增加的16位连接器的引脚图，以及它在计算机中相对于8位连接器的位置。除非在ISA卡上另外增加存储器，否则额外的地址线 A_{23}～A_{20}对I/O操作将不起任何作用。现在最常用的新增特性是增加了一些中断请求输入和DMA请求信号。在一些系统中，16位I/O使用新增的8条数据总线（D_8～D_{15}），但是当今应用更广泛的是将PCI总线用于比8位宽的外围设备上。大概ISA总线仅有的新接口是少数的调制解调器和声卡。

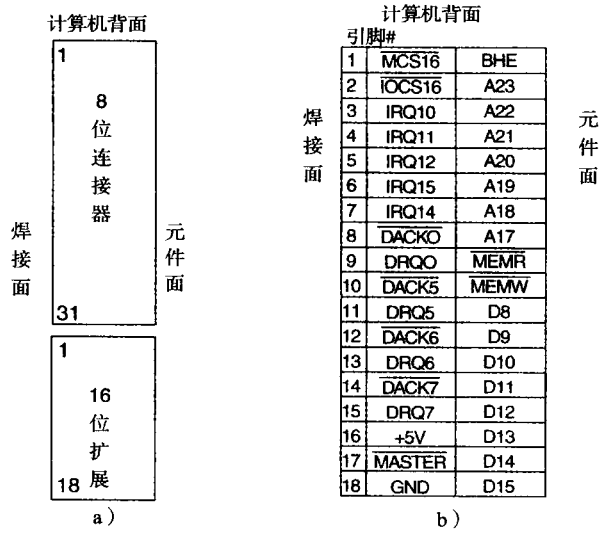


图15-5 16位ISA总线

a)8位和16位连接器 b)16位连接器的引脚图

15.2 外围部件互连（PCI）总线

PCI（peripheral component interconnect，外围部件互连）总线实际上是最新的Pentium 4系统和几乎所有Pentium系统中惟一都在使用的总线。尽管在所有较新的系统中，ISA总线仍然存在，但它只是作为早期8位和16位接口卡的接口。许多新系统中只有2个ISA总线插槽，还有的根本没有ISA插槽。也许有一天ISA总线会消失，但它对于许多应用来说仍是一种重要的接口。PCI总线已取代了VESA局部总线，一个原因是PCI总线具有即插即用的特性，而且能够在64位数据总线上工作。一个PCI接口包括一系列的寄存器，位于PCI接口上的一个小的存储器件中，其中包含了主板的信息。这一相同的存储器可以为ISA总线或任何其他总线提供即插即用特性。这些寄存器中的信息允许计算机自动配置PCI卡。这个特性被称为**即插即用（plug-and-play，PnP）**特性，这也许是PCI总线在最新的计算机系统中变得如此流行的主要原因。

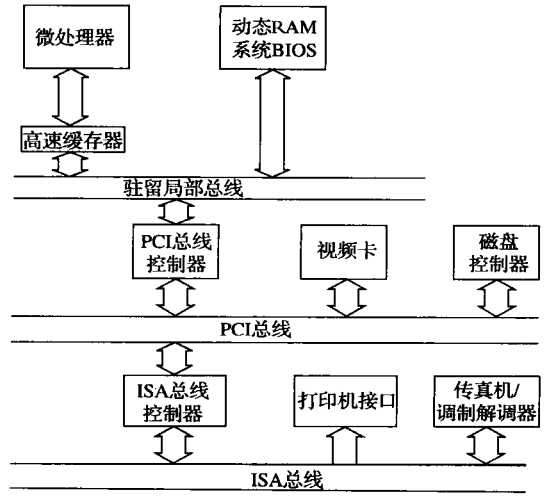


图15-6 包含PCI总线的PC机的系统框图

图15-6给出了PC机系统中PCI总线的系统

结构。注意，微处理器的总线是单独的并独立于 PCI 总线。微处理器通过被称为 PCI 桥的集成电路与 PCI 总线相连。这意味着只要系统设计了 PCI 控制器或 **PCI 桥（PCI bridge）**，那么实际上任何微处理器都可以接到 PCI 总线上。将来，所有的计算机系统也许会使用同一种总线。甚至连 Apple 公司的 Macintosh 系统也正转向 PCI 总线。常驻局部总线（resident local bus）经常被称为前端总线（front side bus）。

15.2.1 PCI 总线的引脚图

如本章中描述的其他总线一样，PCI 总线包含所有的系统控制信号。同其他总线不同，PCI 总线与 32 位或 64 位的数据总线以及一个完全的 32 位地址总线共同工作。另一区别是地址总线和数据总线是多路复用的，以减小边缘连接器的尺寸。这些多路复用引脚在连接器上标识为 $AD_0 \sim AD_{63}$。32 位卡只有连接端 1～62，而 64 位卡则有所有的 94 个连接端。如果将来某一天需要，64 位卡可以容纳一个 64 位的地址。图 15-7 给出了 PCI 总线的引脚图。

正如其他总线系统一样，PCI 总线常用于将 I/O 器件连接到微处理器上。也可以连接存储器，但它只能在 33MHz 的频率下与 Pentium 一起工作，这个速度仅仅是 Pentium～Pentium 4 系统中驻留局部总线速度 66MHz 的一半。PCI 总线的最新版本（与 2.1 版兼容）既可以工作在 66MHz 下，对于早期的接口卡又可以工作在 33MHz 下。Pentium 4 系统使用 200MHz 的系统总线速度（尽管经常号称是 800MHz），但目前还没有进一步修改 PCI 总线速度的计划。

15.2.2 PCI 总线的地址/数据线

PCI 的地址线标识为 $AD_0 \sim AD_{31}$，与数据线是多路复用的。在某些系统中，有一条 64 位的数据总线，但只使用 $AD_{32} \sim AD_{63}$ 来传送数据。将来，这些引脚可用于将地址扩展到 64 位。图 15-8 给出了 PCI 总线的时序图，它说明了地址与数据多路复用的方式以及用于多路复用的控制信号。

在第一个时钟周期中，存储单元和 I/O 单元的地址出现在 AD 总线上，对 PCI 外围设备的命令出现在 C/$\overline{BE}$引脚上。表 15-4 给出了 PCI 总线的总线命令。

INTA 序列 在中断响应序列期间，访问中断控制器（产生中断的控制器），得到中断向量。字节宽度的中断向量在字节读操作期间返回。

特殊周期 特殊周期用于将数据传输给所有的 PCI 器件。在这个周期中，数据总线最右边的 16 位如果是 0000H 表示处理器关闭，如果是 0001H 表示处理器挂起，如果是 0002H 则表示 80X86 的特殊编码或数据。

计算机背面

引脚#	焊接面	元件面
1	-12V	TRST
2	TCK	+12V
3	GND	TMS
4	TDO	TDI
5	+5V	+5V
6	+5V	INTA
7	INTB	INTC
8	INTD	+5V
9	PRSNT 1	
10		+VI/O
11	PRSNT 2	
12	KEY	KEY
13	KEY	KEY
14		
15	GND	RST
16	CLK	VI/O
17	GND	VNT
18	REQ	GND
19	+V IO	
20	AD31	AD30
21	AD29	+3.3V
22	GND	AD28
23	AD27	AD26
24	AD25	GND
25	+3.3V	AD24
26	C/BE3	IDSEL
27	AD23	+3.3V
28	GND	AD22
29	AD21	AD20
30	AD19	GND
31	+3.3V	AD18
32	AD17	AD16
33	C/BE2	+3.3V
34	GND	FRAME
35	IRDY	GND
36	+3.3V	TRDY
37	DEVSEL	GND
38	GND	STOP
39	LOCK	+3.3V
40	PERR	SDONE

引脚#	焊接面	元件面
41	+3.3V	SBO
42	SERR	GND
43	+3.3V	PAR
44	C/BE1	AD15
45	AD14	+3.3V
46	GND	AD13
47	AD12	AD11
48	AD10	GND
49	GND	AD9
50	KEY	KEY
51	KEY	KEY
52	AD8	C/BE0
53	AD7	+3.3V
54	+3.3V	AD6
55	AD5	AD4
56	AD3	GND
57	GND	AD2
58	AD1	AD0
59	+V IO	+V IO
60	ACK64F	REQ64
61	+5V	+5V
62	+5V	+5V
63		GND
64	GND	C/BE7
65	C/BE6	C/BE5
66	C/BE4	+V IO
67	GND	PAR64
68	AD63	AD62
69	AD61	GND
70	+V IO	AD60
71	AD59	AD58
72	AD57	GND
73	GND	AD56
74	AD55	AD54
75	AD53	+V IO
76	GND	AD52
77	AD51	AD50
78	AD49	GND
79	+V IO	AD48
80	AD47	AD46
81	AD45	GND
82	GND	AD44
83	AD43	AD42
84	AD41	+VI/O
85	GND	AD40
86	AD39	AD38
87	AD37	GND
88	+VI/O	AD36
89	AD35	AD34
90	AD33	GND
91	GND	AD32
92		
93		GND
94	GND	

注：1. 引脚63~94只存在于64位PCI卡上。
2. +VI/O在3.3V主板上为+3.3V，在5V主板上为 +5V。
3. 空白引脚为保留引脚。

图 15-7 PCI 总线的引脚图

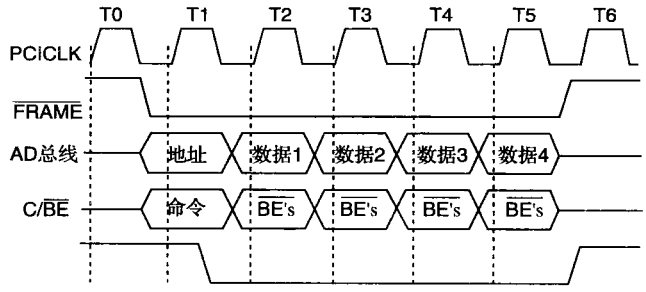


图 15-8 PCI 总线系统的基本突发模式时序图

注：它既可以传送 4 个 32 位数（32 位 PCI），也可以传送 4 个 64 位数（64 位 PCI）。

I/O 读周期 从使用 I/O 地址（出现在引脚 AD_0 ～ AD_{15} 上）的 I/O 设备中读出数据，但 I/O 设备并不支持猝发读出。

I/O 写周期 同 I/O 读周期一样，此周期也访问 I/O 设备，但是它将数据写入 I/O 设备。

存储器读周期 从位于 PCI 总线上的存储器件中读出数据。

存储器写周期 同存储器读周期一样，访问位于 PCI 总线上的设备，将数据写入其中的一个单元。

配置读出 使用配置读周期读出 PCI 设备中的配置信息。

配置写入 此配置写命令允许数据写入 PCI 设备的配置区域，注意地址由配置读命令指定。

存储器多路访问 同存储器读周期很相似，只是此命令通常用于访问多个数据，而不是一个数据。

双寻址周期 用于将地址信息传送给只包含 32 位数据通路的 64 位 PCI 设备。

线性存储器访问 用于从 PCI 总线上读出 2 个以上的 32 位数据。

存储器写并无效操作 同线性存储器访问一样，但只用于写操作。此写操作旁路了高速缓存的回写功能。

表 15-4 PCI 总线命令

C/$\overline{BE3}$-C/$\overline{BE0}$	命 令
0000	INTA 序列
0001	特殊周期
0010	I/O 读周期
0011	I/O 写周期
0100～1001	保留
1010	存储器读周期
1011	存储器写周期
1000～1001	保留
1010	配置读出
1011	配置写入
1100	存储器多路访问
1101	双寻址周期
1110	线性存储器访问
1111	无效存储器写操作

15.2.3 配置空间

PCI 接口包含 256 字节的配置内存空间，允许计算机访问 PCI 接口。这个特性允许系统自动地为 PCI 插板配置系统本身。Microsoft 公司称之为即插即用（PnP）。图 15-9 给出了配置内存及其内容。

第 1 个 64 字节的配置内存包含关于 PCI 接口信息的标题。其中第 1 个 32 位的双字包含了设备识别（ID）码和销售商识别码。设备识别码是一个 16 位的数（D_{31}～D_{16}），如果此设备没有安装，则为 FFFFH；如果安装了，则为 0000H 和 FFFEH 之间的一个数。类别符号识别 PCI 接口的类别。类别符号位于配置内存 08H 单元的 D_{31}～D_{16}位。注意，D_{15}～D_0 由制造商定义。当前的类别符号列于表 15-5 中，并由 PCI SIG 分配，PCI SIG 是 PCI 总线接口标准的主管组织。销售商识别

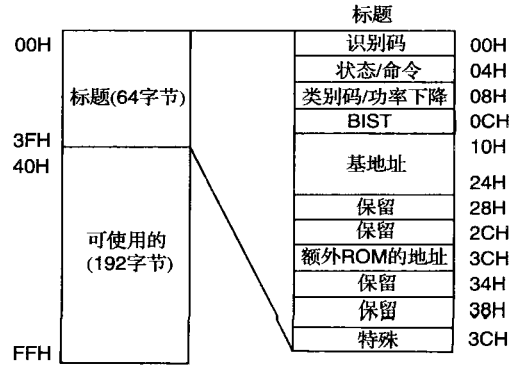


图 15-9 PCI 扩展板上配置内存的内容

码（D_{15}～D_0）也是由 PCI SIG 来分配的。

表 15-5 类别符号

类别符号	功能	类别符号	功能
0000H	早期的非 VGA 设备（非即插即用）	0401H	音频多媒体
0001H	早期的 VGA 设备（非即插即用）	0480H	其他多媒体控制器
0100H	SCSI 控制器	0500H	RAM 控制器
0101H	IDE 控制器	0501H	FLASH 存储器控制器
0102H	软盘控制器	0580H	其他存储器桥控制器
0103H	IPI 控制器	0600H	主机桥
0180H	其他硬盘/软盘控制器	0601H	ISA 桥
0200H	以太网控制器	0602H	EISA 桥
0201H	令牌网控制器	0603H	MCA 桥
0202H	FDDI	0604H	PCI-PCI 桥
0280H	其他网络控制器	0605H	PCMIA 桥
0300H	VGA 控制器	0680H	其他桥
0301H	XGA 控制器	0700H～FFFEH	保留
0380H	其他视频控制器	FFFFH	不在以上类别中的设备
0400H	视频多媒体		

状态字被装入配置内存单元 04H 的 D_{31}～D_{16}位，而命令字位于单元 04H 的 D_{15}～D_0 位。图15-10给出了状态寄存器和命令寄存器的格式。

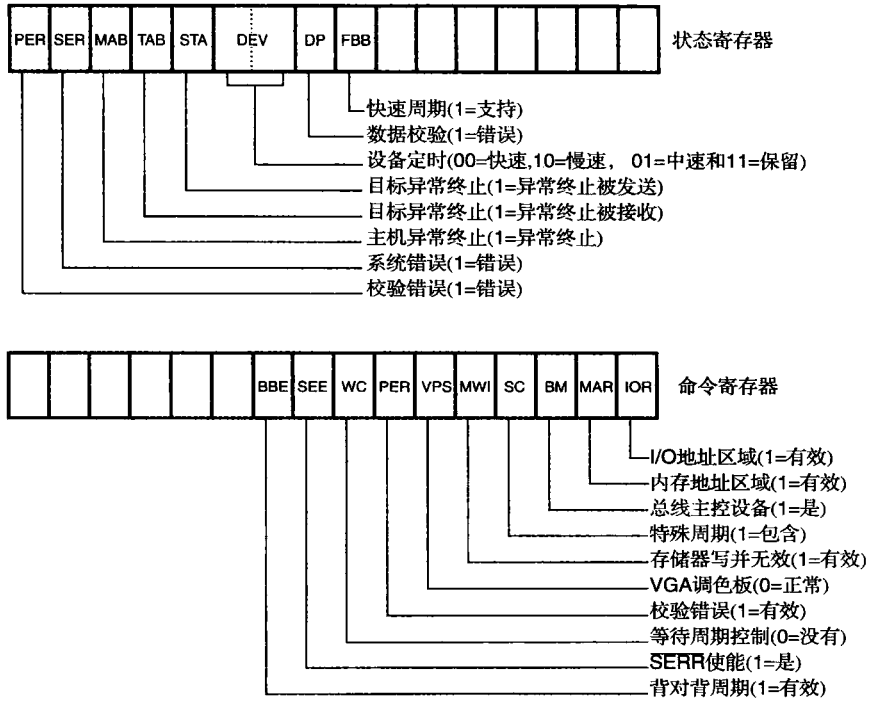


图 15-10 在配置内存中状态字和控制字的内容

基地址空间由内存基地址、I/O 空间地址和扩展 ROM 地址组成。基地址空间中的前 2 个双字包含内存的32 位基地址或64 位基地址，此内存存在于 PCI 接口中。第3 个双字包含 I/O 空间的基地址。注意，尽管 Intel 微处理器只使用 16 位 I/O 地址，但却预留了空间，可以将 I/O 地址扩展到 32 位。这样就允许使用 680X0 系列和 PowerPC 的系统访问 PCI 总线，因为它们确实有通过 32 位地址访问的 I/O 空间。680X0 和 PowerPC 使用存储器映像 I/O，这在第 11 章的开始部分讨论过。

15.2.4 PCI 总线的 BIOS

大多数现代的 PC 机包含 PCI 总线以及对普通系统 BIOS 进行扩展、支持 PCI 总线。这些较新的系统通过中断向量 1AH 访问 PCI 总线，表 15-6 列出了通过 INT 1AH 指令和 AH =0B1H 得到的 PCI 总线功能。

表 15-6 PCI 总线的 BIOS INT 1AH 功能

01H	BIOS 可否使用?
入口	AH =0B1H AL =01H
出口	AH =00H，如果 PCI BIOS 扩展可以使用 BX = 版本号 EDX = ASCII 字符 'PCI' Carry = 1，如果不存在 PCI 扩展
02H	**搜索 PCI 设备**
入口	AH =0B1H AL =02H CX = 设备 DX = 制造商 SI = 索引
出口	AH = 结果代码（参见注释） BX = 总线和设备号 Carry = 1，表示出错
注释	结果代码为： 00H = 搜索成功 81H = 不支持此功能 83H = 无效的制造商 ID 码 86H = 未找到设备 87H = 无效的寄存器号
03H	**搜索 PCI 类别符号**
入口	AH =0B1H AL =03H ECX = 类别符号 SI = 索引
出口	AH = 结果代码（参见功能 02H 的注释） BX = 总线和设备号 Carry = 1，表示出错
06H	**启动特殊周期**
入口	AH =0B1H AL =06H BX = 总线和设备号 EDX = 数据
出口	AH = 结果代码（参见功能 02H 的注释） Carry = 1，表示出错
注释	在地址区间，传送到 EDX 中的值传送给 PCI 总线
08H	**读字节宽度的配置信息**
入口	AH =0B1H AL =08H BX = 总线和设备号 DI = 寄存器号
出口	AH = 结果代码（参见功能 02H 的注释） CL = 来自配置寄存器的数据 Carry = 1，表示出错

（续）

09H	读字宽度的配置信息
入口	AH = 0B1H AL = 08H BX = 总线和设备号 DI = 寄存器号
出口	AH = 结果代码（参见功能02H的注释） CX = 来自配置寄存器的数据 Carry = 1，表示出错
0AH	**读双字宽度的配置信息**
入口	AH = 0B1H AL = 08H BX = 总线和设备号 DI = 寄存器号
出口	AH = 结果代码（参见功能02H的注释） ECX = 来自配置寄存器的数据 Carry = 1，表示出错
0BH	**写字节宽度的配置信息**
入口	AH = 0B1H AL = 08H BX = 总线和设备号 CL = 要写入配置寄存器的数据 DI = 寄存器号
出口	AH = 结果代码（参见功能02H的注释） Carry = 1，表示出错
0CH	**写字宽度的配置信息**
入口	AH = 0B1H AL = 08H BX = 总线和设备号 CX = 要写入配置寄存器的数据 DI = 寄存器号
出口	AH = 结果代码（参见功能02H的注释） Carry = 1，表示出错
0DH	**写双字宽度的配置信息**
入口	AH = 0B1H AL = 08H BX = 总线和设备号 ECX = 要写入配置寄存器的数据 DI = 寄存器号
出口	AH = 结果代码（参见功能02H的注释） Carry = 1，表示出错

例15-5说明了BIOS是如何被用来决定是否可以使用PCI总线扩展的。一旦建立了BIOS，则使用BIOS功能就可以读出配置内存中的内容。注意，BIOS不支持计算机与PCI接口之间的数据传输。数据传输由和接口同时提供的驱动器来控制。这些驱动器控制微处理器与PCI接口上的器件之间的数据流。

例15-5

```
;该 DOS 程序确定 PCI 是否存在
.MODEL SMALL
```

```
.DATA
        MES1 DB "PCI BUS IS PRESENT $"
        MES2 DB "PCI BUS IS NOT FOUND $"
.CODE
.STARTUP
        MOV  AH,0B1H            ;访问 PCI BIOS
        MOV  AL,1
        INT  1AH
        MOV  DX,OFFSET MES2
        .IF CARRY?              ;如果 PCI 存在
              MOV  DX,OFFSET MES1
        .ENDIF
        MOV  AH,9               ;显示 MES1 或 MES2
        INT  21H
        .EXIT
END
```

15.2.5 PCI 接口

PCI 接口比较复杂，为接口到 PCI 总线通常要用一个集成的 PCI 总线控制器。它需要存储器（EPROM）存储销售商信息及其他信息，正如本节前面部分介绍的那样。PCI 接口的基本结构如图 15-11 所示。该框图的内容给出了 PCI 接口正常工作所需的器件，但未给出接口本身。寄存器、奇偶校验模块、启动程序、目标程序以及销售商 ID EPROM 是任何 PCI 接口都需要的器件。如果构造一个 PCI 接口，由于此接口的复杂性，则常常要使用 PCI 控制器。PCI 控制器提供了如图 15-11 所示的结构。

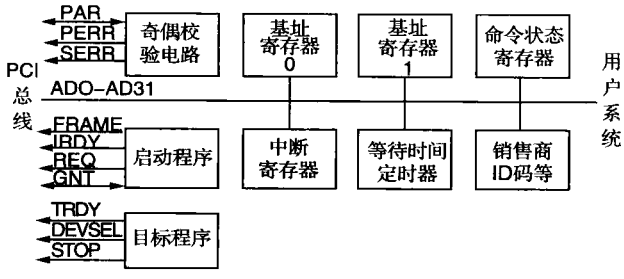


图 15-11 PCI 接口的框图

15.2.6 PCI Express 总线

PCI Express 以 2.5GHz 的频率向先前的 PCI 应用串行地传输数据，PCI Express 接口的数据链路速度可从 250MBps 增加到 8GBps。相比之下，标准的 PCI 总线传输数据的速度是 133MBps。较大的改善就是主板，采用串行方式互连，频率为 2.5GHz。PCI Express 总线上的每一个串行连接称为一个通道。主板上的插槽是单通道插槽，总传输速度达 1GBps。PCI Express 视频卡连接器目前有 16 个通道，传输速度达 4GBps。标准的视频卡最多允许 32 个通道，但是目前视频卡上最宽的插槽也不过 16 个通道。目前大多数主板有 4 个单通道插槽用于外设，有 1 个 16 通道插槽用于视频卡，少数较新的主板则有两个 16 通道的插槽。相信不久，标准的 PCI 插槽将全部被成本较低的 PCI Express 连接器所取代。

PCI Express 2 总线于 2007 年下半年发布，其传输速度是 PCI Express 总线的 2 倍。也就是说，每通道的速度从 250MBps 增加到 500MBps。

表 15-7 PCI Express 单通道引脚（PCI X1）

引脚号	A 边名字	A 边的描述	B 边名字	B 边的描述
1	PRSNT1	当前	+12V	正 12V 电源
2	+12V	正 12 伏电源	+12V	正 12V 电源
3	+12V	正 12 伏电源	保留	未使用
4	GND	地	GND	地
5	JTAG2	TCK	SMCLK	SMBus 时钟
6	JTAG3	TDI	SMDAT	SMBus 数据
7	JTAG4	TDO	GND	地

（续）

引脚号	A 边名字	A 边的描述	B 边名字	B 边的描述
8	JTAG5	TMS	+3.3V	正 3.3V 电源
9	+3.3V	正 3.3V 电源	JTAG1	+TRST#
10	+3.3V	正 3.3V 电源	+3.3V	正 3.3V 电源
11	PWRGD	电源正常	WAKE#	链路激活
12	GND	地	Reserved	未使用
13	REFCLK +	参考时钟	GND	地
14	REFCLK -	参考时钟	HSOp（0）	Lane0 输出数据 +
15	GND	地	HSOm（0）	Lane0 输出数据 -
16	HSIp（0）	Lane0 输入数据 +	GND	地
17	HSIm（0）	Lane0 输入数据 -	PRSNT2	当前
18	GND	地	GND	地

由于 PCI Express 总线具有更快的速度优势，这种新的 PCI 总线正在逐步取代 AGP 端口上的视频卡。这种串行技术可以让主板制造商使用更少的主板空间实现互连从而降低主板制造成本。此外，连接器的变小也降低了连接器的成本。而且，PCI Express 总线与 PCI 总线使用的软件相同，因此，无需为 PCI Express 总线开发新的驱动程序。

表 15-7 列出了支持大多数通用接口连接器、单通道连接器的 PCI Express 的引脚，图 15-12 是一个 36 引脚的连接器。加在 PCI Express 总线上的信号采用 3.3V 的 180 度不同相位的信号。通道由一对数据管道组成，一个用作输入，另一个用作输出。

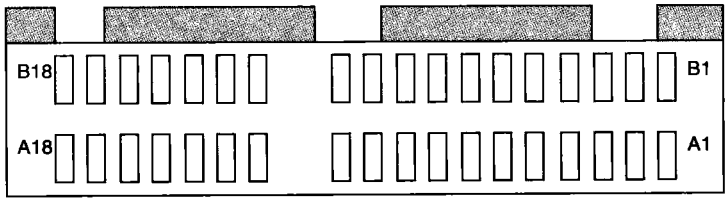


图 15-12 单通道 PCI Express 连接器

15.3 并行打印机接口（LPT）

并行打印机接口（LPT）位于 PC 机的后面，既然它是 PC 机的一部分，就可用作与 PC 机的接口。LPT 用于行式打印机。此打印机接口使用户能够访问可被编程为接收或发送数据的 8 条线。

15.3.1 端口介绍

并行端口（LPT_1）常位于 DOS 或 Windows 驱动程序里的 I/O 端口地址 378H、379H 及 37AH。第 2 个端口 LPT_2（如果存在的话）位于 I/O 端口地址 278H、279H 及 27AH。以下的信息对这两个端口都适用，但这里始终使用 LPT_1 端口地址。

由并口实现的 Centronics 接口使用了两个连接器，即在 PC 机后面的 25 脚 D 型连接器和打印机后面的 36 脚 Centronics 连接器。这两个连接器的引脚描述如表 15-8 所示，连接器如图 15-13 所示。

表 15-8 并行端口的引脚

信　号	描　述	25 脚	36 脚
#STR	选通打印机	1	1
D_0	数据位 0	2	2
D_1	数据位 1	3	3
D_2	数据位 2	4	4
D_3	数据位 3	5	5
D_4	数据位 4	6	6
D_5	数据位 5	7	7

（续）

信　号	描　述	25 脚	36 脚
D_6	数据位 6	8	8
D_7	数据位 7	9	9
#ACK	打印机响应	10	10
BUSY	打印机忙	11	11
PAPER	缺纸	12	12
ONLINE	打印机联机	13	13
#ALF	若打印机在 CR 之后发出 LF，则为低	14	14
#ERROR	打印机错误	15	32
#RESET	复位打印机	16	31
#SEL	选择打印机	17	36
+5V	打印机的 5V	—	18
保护地	大地	—	17
信号地	信号地	所有其他引脚	所有其他引脚

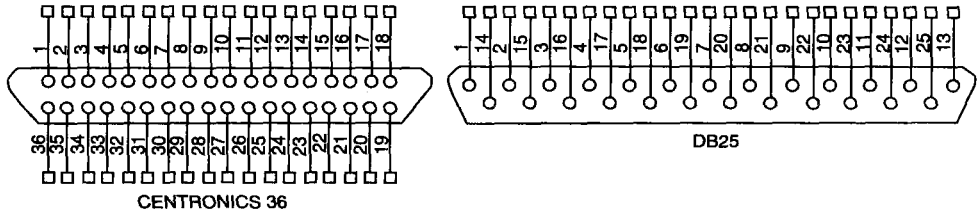


图 15-13　用于并行端口的连接器

并行端口在其数据引脚（$D_0 \sim D_7$）上既可作为一个接收器，也可作为一个发送器工作。这就允许打印机以外的其他设备如 CD-ROM，也可通过并口与 PC 机相连并被使用。任何可通过 8 位接口接收和/或发送数据的设备，都能够而且确实是与 PC 机的并口（LPT_1）相连。

图 15-14 给出了数据端口（378H）、状态寄存器（379H）以及附加状态端口（37AH）的内容。其中一些状态位为逻辑 0 时为真。

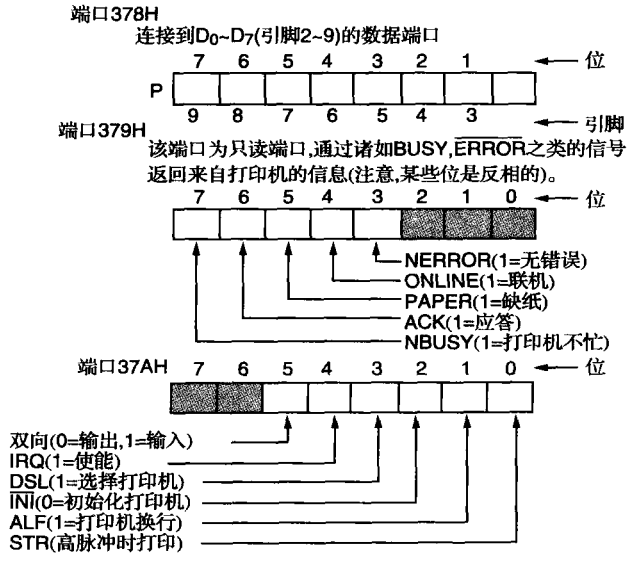


图 15-14　并行端口使用的端口 378H、379H 和 37AH

15.3.2 使用并行端口而不需要 ECP 支持

在大多数系统中由于 IBM 放弃了 PS/2，所以基本可以根据图 15-14 中的信息使用并行端口而不需要 ECP 支持。为读取某端口，首先必须通过发送 20H 给寄存器 37AH 来初始化该端口，如例 15-6 所示。如图 15-14 所示，这样使双向位置 1，并行端口选择输入操作；若该位清 0，则选择输出操作。

例 15-6

```
MOV  AL,20H
MOV  DX,37AH
OUT  DX,AL
```

一旦端口被编程置位，则可以读取该端口，如例 15-7 所示。一旦并行端口作为读取端口被编程置位，可以通过访问位于 378H 数据端口很容易完成读操作。

例 15-7

```
MOV  DX,378H
IN   AL ,DX
```

为向某端口写入数据，要重新编程命令寄存器，向地址 37AH 写 00H，使双向位为 0。一旦编程了双向位，就可以通过数据端口地址 378H 向并口发送数据。例 15-8 给出了如何向并口发送数据的程序。

例 15-8

```
MOV  DX,378H
MOV  AL,WRITE_DATA
OUT  DX,AL
```

在早期（基于 80286）的系统中，接口没有双向位，为从并口读入信息，必须对端口 378H 写 0FFH，然后才可以读取端口。这些老的系统没有位于地址 37AH 的寄存器。

如果使用的是 Windows 2000 或 Windows XP，通过 Windows 操作系统访问打印机端口是很困难的，因为必须要写驱动。在 Windows 98 或 Windows ME 中，访问打印机端口的操作就像本节所介绍的一样。

有一种方式可以通过 Windows 2000 或 Windows XP 访问并行端口而无需写驱动。一种被称为 UserPort 的驱动（在互联网上很容易得到）可以打开 Windows 中受保护的 I/O 端口，这样就允许了在VC++中使用端口 378H，通过汇编模块直接访问并行端口。也允许访问 0000H ～ 03FFH 之间的任何 I/O 端口。另一个有用的工具是 30 天试用版本的 Jungo 工具，可以在 www. jungo . com 上找到。Jungo 工具是一个驱动开发工具，它提供了大多数子系统的许多驱动参考实例。

15.4 串行 COM 端口

在旧系统或有任何数量端口的现代系统中，串行通信端口是 COM_1 ～ COM_8，但是大部分的计算机只安装了 COM_1 和 COM_2，有些只有一个串行通信端口 COM_1。在第 11 章中讲述了在 DOS 环境下通过 16550 串行接口器件控制和访问串行端口的方式，在以下的章节中将不会再重复这部分内容。在本节将介绍如何通过 Windows API 函数使用 16550 串行接口操作 COM 端口。USB 设备经常使用 HID（human interface device，人机接口设备）作为 COM 端口。这样允许标准串行软件访问 USB 设备。

通信控制

使用几个系统 Windows API 函数可以通过任何版本的 Windows 操作系统和 Visual C++ 访问串行端口。例 15-9 使用 Visual Studio. net 2003 给出了一个短小的访问串口的 C++ 函数，该函数名为 WriteComPort，它有 2 个参数：第一个参数是端口，指定了 COM_1 和 COM_2 等；第二个参数是通过该端口传送的字符。传送成功则返回 true，否则返回 false。如果要通过 COM_1 传送字母 A，那么就调用 WriteComPort（“COM_1”，“A”）。该函数只能通过串行 COM 端口传送 1 个字节，但也可以修改它以传送字符串。通过 COM_2 传送 00H（其他的数字不能用这种方式），则调用 WriteComPort（“COM_2”，0x00）。注意，

COM 端口的波特率被设定为 9600 波特，可以通过修改 CBR_9600 来容易地修改波特率值。表 15-9 给出了允许的波特率。

CreateFile 结构创建了一个 COM 端口的句柄，使用该句柄将数据写入端口。在得到端口状态并且根据波特率要求改变端口状态后，WriteFile 函数发送数据到端口。WriteFile 函数的参数是文件句柄 hPort，将要写入的字符串形式的数据，字节数（本例中为 1），以及实际写入端口的字节的存储位置。

表 15-9 COM 端口所允许的波特率

关 键 字	位/秒
CBR_110	110
CBR_300	300
CBR_600	600
CBR_1200	1200
CBR_2400	2400
CBR_4800	4800
CBR_9600	9600
CBR_14400	14400
CBR_19200	19200
CBR_38400	38400
CBR_56000	56000
CBR_57600	57600
CBR_115200	115200
CBR_128000	128000
CBR_256000	256000

例 15-9

```
bool WriteComPort(CString PortSpecifier, CString data)
{
    DCB dcb;
    DWORD byteswritten;
    HANDLE hPort = CreateFile(PortSpecifier,
    GENERIC_WRITE,
    0,
    NULL,
    OPEN_EXISTING,
    0,
    NULL);
    if (! GetCommState(hPort,&dcb)){
          return false;
    }
    dcb.BaudRate = CBR_9600;              //9600 波特
    dcb.ByteSize = 8;                     //8 位数据
    dcb.Parity = NOPARITY;                //无奇偶位
    dcb.StopBits = ONESTOPBIT;            //1 个停止位
    if (! SetCommState(hPort,&dcb))
          return false;
    bool retVal = WriteFile(hPort,data,1,& byteswritten,NULL);
    CloseHandle(hPort);                   //关闭句柄
    return retVal;}
```

通过 COM 口接收数据有些难度，因为它比传输过程更易发生错误。也有许多类型的错误可以被检测到，经常向用户报告。例 15-10 给出了名为 ReadByte 的 C++ 函数，它用于从串口中读取一个字符。该函数返回从端口读到的字符，当端口无法打开时返回错误代码 0×100，当接收器检测到错误时返回 0×101。如果没有数据接收，则该函数挂起，因为没有设置超时。

例 15-10

```
int ReadByte(CString PortSpecifier)
{
      DCB dcb;
      int retVal;
      BYTE Byte;
      DWORD dwByteTransferred;
      DWORD dwCommModemStatus;
```

```
        HANDLE hPort =CreateFile(PortSpecifier,
        GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
        0,
        NULL);

        if (! GetCommState(hPort,&dcb))
              return 0x100;

        dcb.BaudRate = CBR_9600;          //9600 波特
        dcb.ByteSize = 8;                 //8 位数据
        dcb.Parity = NOPARITY;            //无奇偶位
        dcb.StopBits = ONESTOPBIT         //1 个停止位

        if (! SetCommState(hPort,&dcb))
              return 0x100;

        SetCommMask (hPort, EV_RXCHAR | EV_ERR);        //接收字符事件
        WaitCommEvent(hPort, &dwCommModemStatus, 0);  //等待字符

        if (dwCommModemStatus & EV_RXCHAR)
             ReadFile (hPort,&Byte, 1, &dwByteTransferred, 0);  //读1
        else if (dwCommModemStatus & EV_ERR)
             retVal =0x101;
        retVal =Byte;
        CloseHandle(hPort);
        return retVal;
}
```

如果使用的是 Visual Studio Express，那么可以利用工具箱中的串口控件访问任何 COM 端口。由于某些原因，Visual Studio 5 中曾有串口控件，后来 Visual Studio 5 和 Visual Studio. net 又去掉了该控件，而后在 2005 企业版中又加入了串口控件。很多 USB 设备类似 COM 口，通过串口控件就可以访问，同传统的 COM 口一样。HID USB 设备是使微软向 Visual Studio 中增加串口控件的主要原因。

一旦在程序中加入了串行端口控制，其属性中就要为通信设置一些参数，当接收到数据时使用一个事件句柄，例 15-11 中列出了发送数据时的函数。

例 15-11

```
private: System::Void SendPort(String^ portDataString)
{
       serialPort1->WriteLine(portDataString);
}
```

为了接收数据，需要为接收到的数据设置句柄。每当从串口收到数据信息，就会调用数据接收事件对信息进行处理。例 15-12 显示了数据接收函数。此处没有列出的部分是端口必须利用串口类中的 Open 函数打开发送或接收信息。

例 15-12

```
private: System::Void serialPort1_DataReceived(System::Object^ sender,
   ystem::IO::Ports::SerialDataReceivedEventArgs^ e)'
{
       String^ receivedString = serialPort1->ReadLine();

       // process the line read from the port
}
```

15.5 通用串行总线（USB）

通用串行总线（USB）解决了PC机系统的一个问题。目前PCI声卡使用PC内部电源，它会产生极大的噪声。由于USB允许声卡有其自己的电源，所以与PC机电源有关的噪声可以消除掉，从而得到高保真度的音质。其他优点是易于进行用户连接，而且通过一条4线串行电缆可以访问最多127个不同的USB设备。此接口对于键盘、声卡、简单的图像－检索设备以及调制解调器均是理想的接口。对于全速USB2.0操作，数据传输速度为480Mb/s，USB1.1应允传输速度为12 Mb/s，而对于慢速操作则为1.5Mb/s。

对于全速接口，电缆长度被限制为最长5米，而对于慢速接口则限制为最长3米。这些电缆可使用的功率为最大100mA电流，电压为5.0V。如果电流值超过100mA，则Windows将在该设备旁显示一个黄色的惊叹号，以表示电流过载。

15.5.1 连接器

图15-15给出了USB连接器的引脚图。规定了2种连接器，且均在使用中。每种连接器有4个引脚，包含表15-10所指示的信号。正如前面提及的，只要每个设备的电流值不超过100mA，则+5.0V和地信号就能够用于驱动与总线相连的设备。数据信号为双相信号，当正数据为+5.0V时，则负数据为0V，反之亦然。

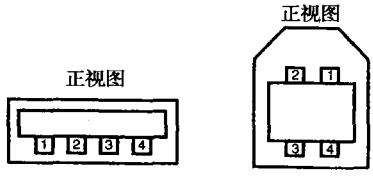


图15-15 USB连接器的两种常见类型的正视图

表15-10 USB引脚配置

引脚号	信号
1	+5.0V
2	负数据
3	正数据
4	地

15.5.2 USB数据

USB的数据信号为双相信号，它通过使用如图15-16所示的电路来产生。图15-16中还给出了线路接收器。与一对发送端相连的为噪声抑制电路，此电路可从Texas Instruments（SN75240）上得到。一旦收发器处于适当位置，则与USB的连接就完成了。Texas Instruments的75773集成电路芯片同此图中的差分线路驱动器与接收器的功能相同。

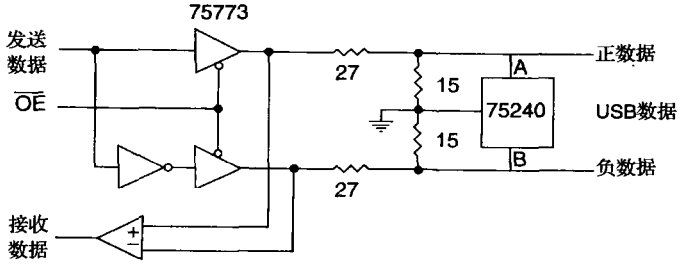


图15-16 使用一对CMOS缓冲器的与USB连接的接口

下一步了解信号是如何在USB上相互作用的。这些信号允许数据从主机系统中发送与接收。USB使用NRZI（non-return to zero, inverted，反向不归零制）数据编码来传送数据包。这一编码方法在传送逻辑1时不改变信号电平，但每当信号由1变为0时，信号电平要反向。图15-17给出了数字式数据流以及使用该编码方法产生的USB信号。

所传送的实际数据包含同步位，它们是使用称为位填充（bit stuffing）的方法产生的。如果在一行

中传送的逻辑 1 多于 6 位，则位填充技术在一行 6 个连续的逻辑 1 后增加一个额外的位（逻辑 0）。由于这加长了数据流的长度，所以被称为位填充。图 15-18 给出了经过位填充的串行数据流，以及用来从原始的数字式串行数据产生这个数据流的运算法则。位填充确保了对于长的逻辑 1 串，接收器可以维持同步。数据传送总是首先从最低有效位开始，接着传送后来那些位。

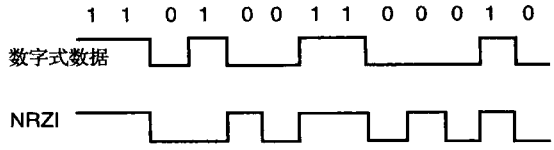


图 15-17 USB 使用的 NRZI 编码

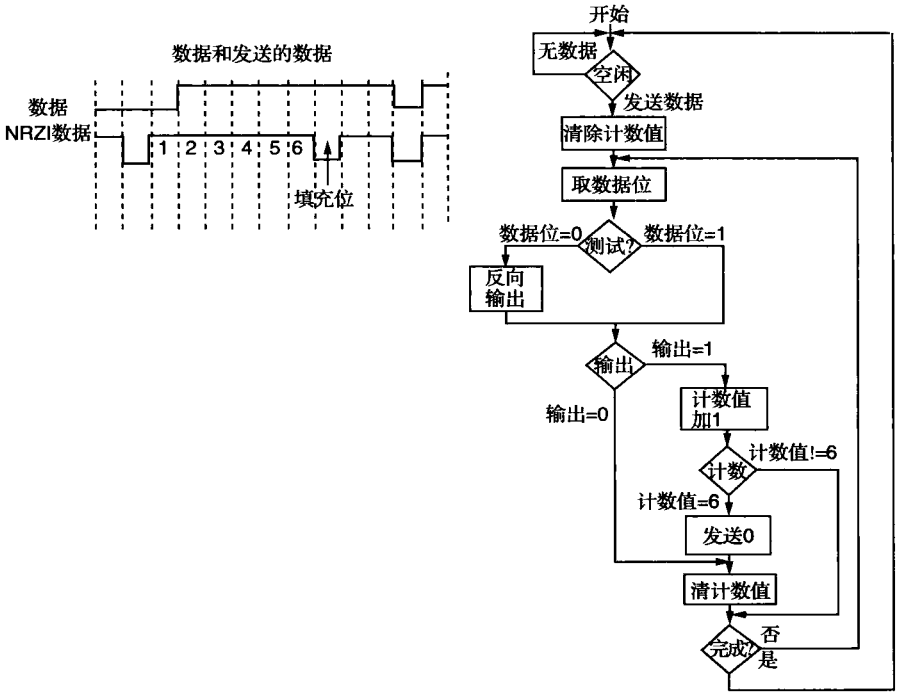


图 15-18 数据流以及用于产生 USB 数据的流程图

15.5.3 USB 命令

既然已经了解了 USB 的数据格式，那么现在开始讨论用于传输数据和选择接收器的命令。为开始通信，首先传送同步字节（80H），接着传送数据包识别字节（PID）。PID 包含 8 位，但只有最右边的 4 位包含所跟数据包的类型。PID 的最左边 4 位是最右边 4 位的 1 的补码。例如，如果发送命令 1000，则发送给 PID 的实际字节为 0111 1000。表 15-11 给出了可得到的 4 位 PID 及它们的 8 位代码。注意，有的 PID 用作标记指示器、数据指示器以及握手。

表 15-11 PID 代码

PID	名 称	类 型	描 述	PID	名 称	类 型	描 述
E_1	OUT	标记	主机→功能事务处理	5A	NAK	握手	接收器不接收数据
D_2	ACK	握手	接收器接收数据包	4B	Data1	数据	数据包（PID 奇数）
C_3	Data0	数据	数据包（PID 偶数）	3C	PRE	特殊	主机前同步信号
A_5	SOF	标记	帧起始	2D	Setup	标记	设置命令
69	IN	标记	功能→主机事务处理	1E	Stall	标记	停止

图 15-19 列出了 USB 中出现的数据、标记、握手以及帧起始数据包的格式。在标记数据包中，ADDR（地址域）包含 USB 设备的 7 位地址。正如前面提到的，一次可以有最多 127 个设备出现在 USB 上。ENDP（端点）是一个由 USB 使用的 4 位数。端点 0000 用于初始化，而其他端点数对于每个 USB 设备是惟一的。

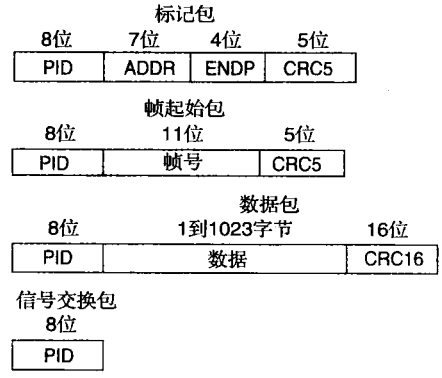


图 15-19 USB 中出现的数据包类型及内容

在 USB 上使用了两种 **CRC（cyclic redundancy check，循环冗余校验）**：一种是 5 位 CRC，另一种是 16 位 CRC（用于数据包）。5 位 CRC 用多项式 $X^5 + X^2 + 1$ 来产生，16 位 CRC 用多项式 $X^{16} + X^{15} + X^2 + 1$ 来产生。构造电路来产生或检测 CRC 时，加号代表异或电路。注意 CRC 为串行校验机制。当使用 5 位 CRC 码时，如果在所有 5 位 CRC 及所有数据位中没有错误，则接收到剩余数 01100；对于 16 位 CRC，没有错误时剩余数为 1000000000001101。

USB 使用 ACK 和 NAK 标记来协调数据包在主机系统和 USB 设备之间的传输。一旦一个数据包从主机传输给 USB 设备，则 USB 设备发回 ACK（响应）或 NAK（不响应）标记给主机。如果数据和 CRC 被正确接收，则发送 ACK；如果接收不正确，则发送 NAK。如果主机接收到 NAK 标记，则它重发送数据包，直到接收器最后正确接收到此数据包为止。这种数据传输的方法通常被称为**停止并等待数据流控制（stop and wait flow control）**。在传输其他数据包之前，主机必须等待客户发送 ACK 或 NAK。

15.5.4 USB 总线节点

美国国家半导体（National Semiconductor）公司生产出的 USB 接口能够非常方便地与微处理器进行连接。图 15-20 给出了 USBN9604 USB 节点。这个设备使用非 DMA 访问与系统连接在一起，通过把数据总线与 $D_0 \sim D_7$ 相连，分别连接控制信号 $\overline{RD}$、$\overline{WR}$、$\overline{CS}$，以及将一个 24MHz 的基础晶振与 X_{in} 和 X_{out} 引脚相连就可以完成连接。USB 总线信号与 D - 和 D + 相连。把两个模式输入接地是最简单的接口模式，这使设备处于非多路的并行模式。在这种模式下，使用 A_0 来选择地址（1）或数据（0）。图 15-21 给出了 USBN9604 与微处理器相连接的方式，其中 I/O 端口译码地址为：0300H（数据）和 0301H（地址）。

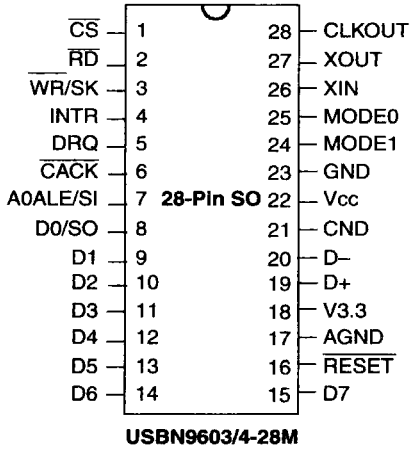


图 15-20 美国国家半导体公司的 USB 总线节点

USBN9604 是一种能接收和传送 USB 数据的 USB 总线收发器。它提供了一个至少节省 2 美元的连接到 USB 总线的接口点。

15.5.5 USBN9604/3 编程

以下的函数都是根据图 15-21 而编写，其中主机系统的驱动程序没有给出。例 15-13 给出了初始化 USB 控制器的代码。过程 USBINT 将 USB 控制器设置成使用端点 0 来传输数据。

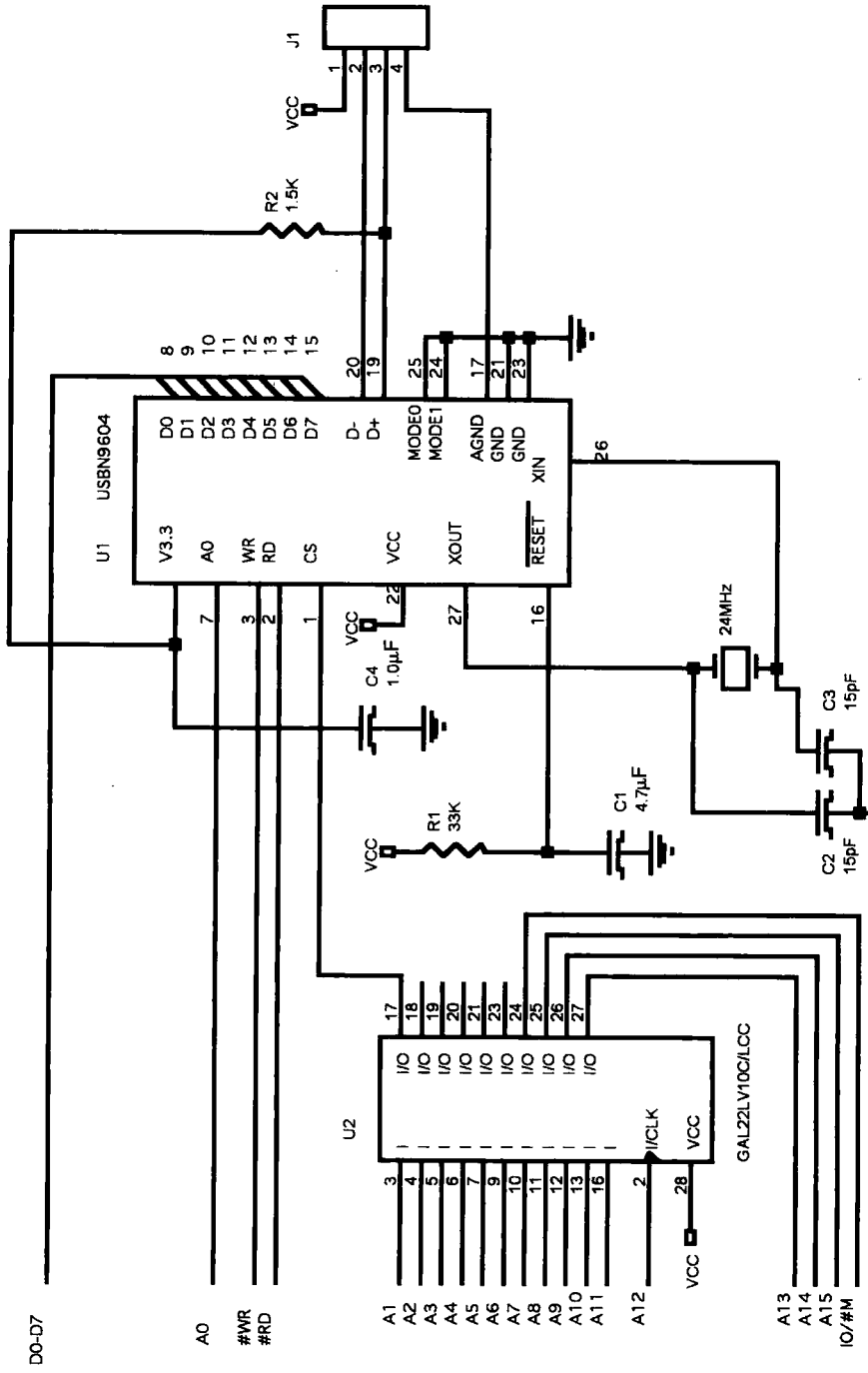


图15-21 USBN9604与微处理器相连，I/O端口地址为300H和301H

例 15-13

```
SEND   MACRO  ADDR, DATA
       MOV  DX,301H
       MOV  AL,ADDR
       OUT  DX,AL
       MOV  DX,300H
       MOV  AL,DATA
       OUT  DX,AL
       ENDM

USBINT PROC NEAR

       SEND 0,5                ;关闭中断,软件复位 USB
       SEND 0,4                ;清除 reset
       CALL DELAY1             ;等待 1ms
       SEND 9,40H              ;使能复位检查
       SEND 0DH,3              ;使能 EP0 以接收数据
       SEND 0BH,3              ;使能 EP0 以发送数据
       SEND 20H,0              ;EP0 控制使得无默认地址
       SEND 4,80H              ;设置 FAR 接受默认地址
       SEND 0,8CH              ;USB 就绪可以接收或发送数据

USBINIT ENDP
```

一旦 USB 控制器初始化完毕，就可以通过 USB 接收或发送数据给主机系统。为完成数据传输，调用例 15-14 的过程，使用 TXD0 FIFO 来发送一个字节包。该过程使用了例 15-13 中的 SEND 宏，通过 USB 将 BL 中的字节传给主机系统。

例 15-14

```
TRANS  PROc  NEAR

       SEND 21H,BL          ;将 BL 传给 FIFO
       SEND 23H,3           ;传送字节数据

TRANS  ENDP
```

从 USB 接收数据需要两个函数。一个用来检测数据是否到达，另一个从 USB 中读取一个字节并将其放入 BL 寄存器。例 15-15 给出了这两个函数。STATUS 过程检查数据是否已经在接收 FIFO 中，如果数据已在，则进位位被设为返回，否则被清空。READS 过程重新取回接收 FIFO 中的字节并将其返回给 BL。

例 15-15

```
READ   MACRO  ADDR

       MOV  DX,301H
       MOV  AL,ADDR
       OUT  DX,AL
       MOV  DX,300H
       IN   AL,DX
       ENDM

STATUS PROC   NEAR

       SEND   6
       SEND   6
       SHL    AL,2
       RET
```

```
STATUS ENDP
READS  PROC   NEAR
       READ   25H
       RET
READS ENDP
```

15.6 加速图形端口（AGP）

现在许多计算机系统新增加了**加速图形端口**（**accelerated graphics port**，**AGP**），直到 PCI Express 接口可用于视频。AGP 工作在微处理器的总线时钟频率下，被设计用来使视频卡与系统存储器之间的数据传输可按最大速度进行，AGP 最大可以 2Gbps 的速率传输数据。这种端口除视频卡外也许绝不会用于其他任何设备，所以不用很大篇幅去讨论它。因为 PCI Express 视频卡使用 8 条通道，x16 PCI Express 视频卡以 4Gbps 的速率传输数据。

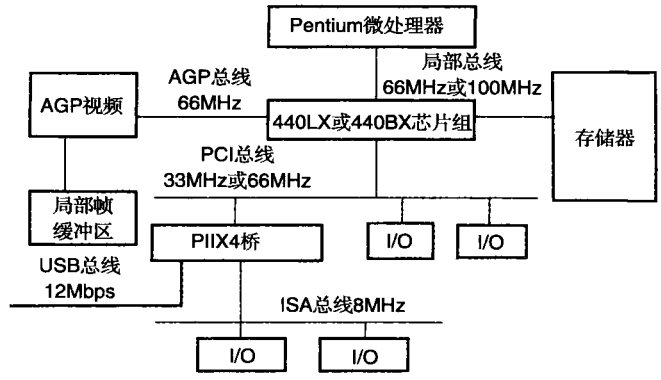


图 15-22 现代计算机的结构（给出了所有总线）

图 15-22 给出了 AGP 与 Pentium 4 系统的连接图以及系统中的其他总线。AGP 总线优于 PCI 总线的地方主要是：AGP 可以以最高 4Gbps 的速度（使用标称 8X 的系统）持续进行数据传输，4X 系统以超过 1Gbps 的速率传输数据，而 PCI 总线最大传输速度只有大约133MB/s。AGP 专门设计用来允许在视频卡帧缓冲器与系统存储器之间通过芯片组进行高速数据传输。

将来 PCI 总线可能会被淘汰，USB 将被合并到芯片组中。甚至芯片组会被集成到微处理器中。现在的系统使用 865 或 875 芯片组，如果主板上需要定制 ISA 总线的话，还会需要 ISA 桥。

15.7 小结

1）总线系统（ISA、PCI 和 USB 总线）允许 I/O 接口和存储系统连接到 PC 机上。

2）ISA 总线为 8 位或 16 位总线，支持存储器或 I/O 接口在 8MHz 的频率下进行数据传输。

3）PCI（外围部件互连）总线支持在 PC 机与存储器或 I/O 接口之间以 33MHz 频率进行 32 位或 64 位的数据传输。这种总线实际上允许任何微处理器通过桥接口接到 PCI 总线上。

4）大多数计算机中的 PCI Express 总线采用单通道或 16 通道端口形式，其中单通道端口与 I/O 设备接口，而 16 通道端口则向视频卡提供接口以取代 AGP。

5）即插即用（PnP）接口是一个包含一段内存的设备，这段内存中保留系统的配置信息。

6）LPT_1 并行端口用来并行传输 8 位数据到打印机和其他设备。

7）COM 串口用于串行数据传输，Windows VC++应用程序使用 Windows API 实现通过 COM 口的串行数据传输。

8）通用串行总线（USB）在大多数高级系统中一定会取代 ISA 总线。USB 有 3 种数据传输率：1.5Mbps、12Mbps 和 480Mbps。

9）USB 使用 NRZI 系统来编码数据，传送超过 6 位长的逻辑 1 使用位填充技术。

10）高速图形端口（AGP）是存储系统与视频图形卡之间的高速连接。

15.8 习题

1. ISA 是什么短语的缩写？
2. ISA 总线系统支持多少位数据传输？
3. ISA 总线接口是否常用于存储器扩展？
4. 设计一个 ISA 总线接口，在地址 310H～313H 译码，此接口必须包含通过这些端口地址访问的 8255（记住缓冲 ISA 总线卡的所有输入）。

5. 设计一个 ISA 总线接口，译码端口 0340H ～ 0343H 来控制一个 8254 定时器。
6. 设计一个 32 位 PCI 总线接口，增加一个 27C 256 EPROM，其存储器地址为 FFFF0000H ～ FFFF7FFFH。
7. 给定一个 74LS244 缓冲器和一个 74LS374 锁存器，设计一个 ISA 总线接口，使它包含一个 I/O 地址为 308H 的 8 位输入端口和一个 I/O 地址为 30AH 的 8 位输出端口。
8. 设计一个 ISA 总线接口，允许 4 个通道的模拟输出信号，每个通道输出 0V ～5.0V。这 4 个通道必须在 I/O 地址 300H、310H、320H 和 330H 译码，并且设计程序支持这 4 个通道。
9. 重做第 8 题，但不是 4 个输出通道，而是使用 4 个 ADC 在与第 8 题相同的地址上产生 4 个模拟输入通道。
10. 使用一个或多个 8254 定时器，在 ISA 总线卡上设计一个暗室定时器，在 1/100 秒～ 5 分钟之间每隔 1/100 秒的间隔产生一个逻辑 0。使用 8MHz 的系统时钟作为定时源。所设计的程序必须允许用户从键盘上选择时间。定时器的输出信号在用户选择时间期间必须为逻辑 0。此输出信号必须通过一个反相器，以允许一个控制相片放大机的固态继电器。
11. 将 16550 UART 接到 ISA 总线接口上的 PC 机上。设计程序，以 300、1200、9600 和 19 200 的波特率发送和接收数据。UART 必须响应 I/O 端口 1E3XH。
12. ISA 总线可以以 8MHz 频率传输_____宽的数据。
13. 描述如何从 PCI 总线上捕获地址。
14. PCI 总线接口上的配置内存的作用是什么？
15. 定义术语“即插即用”。
16. PCI 总线系统上的 C/$\overline{BE}$线的用途是什么？
17. PCI BIOS 扩展是如何测试 BIOS 的？
18. 设计一个小程序，使用 BIOS 扩展访问 PCI 总线。此程序读出配置寄存器 08H 的 32 位内容。在这里假定总线和设备号为 0000H。
19. PCI 总线在什么方面优于 ISA 总线？
20. PCI Express 传输串行数据的速率有多大？
21. PCI Express 接口中的通道指的是什么？
22. 在 PC 机中，并行端口被译码到哪些 I/O 地址？
23. 可以从并口中读数据吗？
24. 计算机后面的并口连接器有_____个引脚？
25. 大部分的计算机至少有一个串行通信端口，这个端口被称为_____。
26. 开发一个 C++ 程序，通过串口传送字符 ABC，直到该串口返回 ABC 时程序才停止。给出完成这个功能的所有函数，包括初始化。
27. 修改例 15-9，使其能够传输任意长的字符串。
28. 在网上查资料，写一篇在可视化编程环境中所用到的变量的报告。
29. USB 设备可以作为 COM 设备出现吗？
30. USB 使用的数据传输率为多少？
31. 在 USB 上数据如何编码？
32. 对于 USB，可用电缆的最大长度为多少？
33. USB 总线会取代 ISA 总线吗？
34. 在 USB 上可以有多少设备地址？
35. 什么是 NRZI 编码？
36. 什么是填充位？
37. 如果以下原始数据被发送到 USB 上，试画出 USB 上出现的信号波形：(11001100001100110110 10)。
38. 在 USB 上一个数据包可以有多长？
39. USB 上 NAK 和 ACK 标记有什么作用？
40. 描述 PCI 总线与 AGP 在数据传输率上的区别。
41. 使用 AGP 8X 视频卡时系统的数据传输速率是多少？
42. PCI Express 16X 视频卡的传输速率是多少？
43. 从网上查找几个显卡生产商，看其 AGP 显卡的内存为多少，列一个清单，给出生产商及其显存大小。
44. 在网上查资料，写一篇关于任一 USB 控制器的报告。

第 16 章　80186、80188 及 80286 微处理器

引言

Intel 80186/80188 及 80286 是早期的 80X86 微处理器家族的增强型号。80186/80188 及 80286 都是向上兼容 8086/8088 的 16 位微处理器。甚至这些微处理器的硬件也与早期型号相似。本章对每种处理器进行了简要介绍，并指出了每个型号的不同或改进之处。本章首先描述 80186/80188 微处理器，最后介绍 80286 微处理器。

本书这一版扩大了 80186/80188 系列的范围。Intel 公司在其微处理器系列中，给每一种嵌入式控制器都添加了四个新的型号，它们都是 CMOS 型的，用两个字母后缀来区分：XL、EA、EB 和 EC。其中，80C186XL 和 80C188XL 与早期的 80186/80188 最为相似。

目的

读者学习完本章后将能够：

1）描述 80186/80188 和 80286 微处理器相对于 8086/8088 在软件和硬件方面的改进。
2）详述 80186 和 80188 嵌入式控制器各型号之间的区别。
3）了解 80186/80188 和 80286 与存储器和 I/O 之间的接口。
4）利用这些微处理器所提供的增强功能开发软件。
5）描述 80286 微处理器内部的存储管理单元（MMU）的运作。
6）定义和详述实时操作系统（RTOS）的作用。

16.1　80186/80188 的结构

类似于 8086 和 8088 的关系，80186 和 80188 之间也非常相似，它们之间惟一的区别在于数据总线的宽度。80186（类似 8086）数据总线是 16 位，而 80188（类似 8088）的数据总线则是 8 位。80186/80188 的内部寄存器结构与 8086/8088 完全相同。惟一的区别在于 80186/80188 包含了附加的预留中断向量和一些非常强大的内建 I/O 功能。由于 80186 和 80188 的主要应用是作为控制器，而不是作为基于微处理器的计算机，所以它们常常被称为**嵌入式控制器**（**embedded controller**）。

16.1.1　80186/80188 的型号

如前所述，80186 和 80188 有四种不同的型号，它们都是 CMOS 微处理器。表 16-1 列出了每个型号及其主要特征。80C186XL 和 80C188XL 是 80186 和 80188 的最基本的型号，而 80C186EC 和 80C188EC 是最高级的型号。本书详述了 80C186XL/80C188XL，并在此基础上对其他型号的附加特性和增强功能进行了描述。

表 16-1　80186/80188 嵌入式控制器的四种型号

特　性	80C186XL 80C188XL	80C186EA 80C188EA	80C186EB 80C188EB	80C186EC 80C188EC
类 8259 指令集	√	√	√	√
节能（绿色方式）	√	√		√
停机方式		√	√	√
80C187 接口	√	√	√	√
ONCE 方式	√	√	√	√
中断控制器	√	√	√	√
				类 8259
定时器单元	√	√	√	√
片选单元	√	√	√	√
			增强	增强
DMA 控制器	√	√		√
	2 通道	2 通道		4 通道
串行通信单元			√	√
刷新控制器	√	√	√	√
			增强	增强
看门狗定时器				√
I/O 端口			√	√
			16 位	22 位

16.1.2 80186 基本结构框图

图 16-1 给出了 80186 微处理器各型号的基本结构框图，它不包括表 16-1 中列出的附加特性和增强功能。请注意，这个微处理器比 8088 多了很多内部电路。除了预取队列在 80188 中为 4 个字节而在 80186 中为 6 个字节以外，80186 的结构框图与 80188 的相同。类似 8088，80188 同样也含有总线接口单元（BIU）和执行单元（EU）。

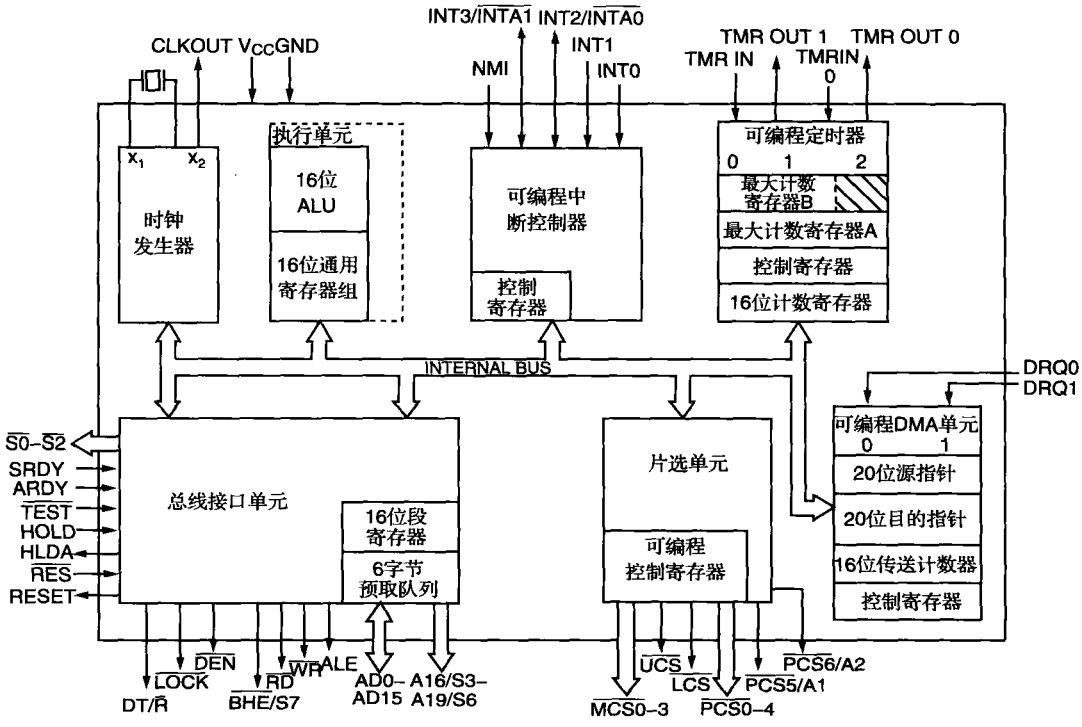


图 16-1 80186 微处理器的结构框图

注：除了没有$\overline{BHE}$/S7 及 AD_{15}～AD_8 被另标为 A_{15}～A_8 以外，80188 的结构框图与本图相同（由 Intel 公司提供）。

除了 BIU 和 EU 外，80186 和 80188 系列还含有时钟发生器、可编程中断控制器、可编程定时器、可编程 DMA 控制器和可编程片选单元。这些增强功能提高了 80186/80188 的通用性，减少了实现系统所需的外围器件。许多 PC 上流行的子系统都用 80186/80188 微处理器作为磁盘高速缓存控制器、局域网（LAN）控制器等。80186/80188 还用在蜂窝电话网络的交换器中。

如果不包括存储管理指令，80186/80188 的软件和 80286 微处理器一样。这也意味着类 80286 的一些指令，如立即数乘、立即数移位计数、串 I/O、PUSHA、POPA、BOUND、ENTER 和 LEAVE 都能在 80186/80188 微处理器上运行。

16.1.3 80186/80188 基本特征

这一节将介绍 80186/80188 微处理器或嵌入式控制器的增强功能，除非特别指出，这些增强功能适用于所有型号。但我们没有提供严格的覆盖范围，有关每个增强功能的操作细节和每个高级型号的细节将在本章的后面部分给出。

时钟发生器

内部时钟发生器取代了和 8086/8088 微处理器一起使用的外部时钟发生器 8284A，从而减少了系统中的元器件数目。

内部时钟发生器有 3 个引脚：X_1、X_2 和 CLKOUT（有些型号中为 CLKIN、OSCOUT 和 CLKOUT）。

X_1（CLKIN）和 X_2（OSCOUT）引脚连接到共振频率是微处理器工作频率两倍的晶体上。在 8MHz 型号的 80186/80188 中，X_1（CLKIN）和 X_2（OSCOUT）引脚接 16MHz 的晶体。80186/80188 有 6MHz、8MHz、12MHz、16MHz 和 25MHz 等型号。

CLKOUT 引脚提供了系统时钟信号，该信号的频率为晶体频率的一半，占空比为 50%。CLKOUT 引脚可以驱动系统中的其他器件，可以为系统中其他的微处理器提供定时源。

除了这些外部引脚外，时钟发生器还提供了用于同步 READY 输入引脚的内部定时信号，而在 8086/8088 系统中，READY 的同步由时钟发生器 8284A 提供。

可编程中断控制器

可编程中断控制器（Programmable Interrupt Controller，简称 PIC）仲裁内部或外部的中断，并最多可控制两片外部的 8259A PIC。当连接了外部的 8259 时，80186/80188 微处理器作为主控制器，而外部 8259 作为从控制器。80C186EC 和 80C188EC 中含有一个兼容 8259A 的中断控制器，替代了这里所描述的其他型号（XL、EA 和 EB）中所使用的中断控制器。

在没有外部的 8259 时，PIC 有 5 个中断输入：INT0 ～ INT3 和 NMI。请注意，可用的中断数依赖于微处理器的型号：EB 型号有 6 个中断输入而 EC 型号有 16 个。这是对 8086/8088 微处理器上的两个中断输入的扩充。在许多系统中，5 个中断输入就足够了。

定时器

定时器部分包含了三个完全可编程的 16 位定时器。定时器 0 和定时器 1 产生外部使用的波形，它们由 80186/80188 主时钟或外部时钟驱动。这两个定时器也可用来对外部事件计数。第三个定时器即定时器 2，是一个内部定时器，以主时钟作为它的时钟。定时器 2 的输出可以用来在指定的时钟周期后产生中断，也可以用来给其他定时器提供时钟。由于定时器 2 可以编程为在确定的时间后中断微处理器，因此定时器 2 可以用作看门狗定时器。

80C186EC 和 80C188EC 中附加了一个称为看门狗（watchdog）的定时器。看门狗定时器是一个 32 位的计数器，在内部以 CLKOUT 信号（晶体频率的一半）作为时钟。计数器每计数到 0 时，就会再装入计数值并在 WDTOUT 引脚产生一个宽度为 4 个 CLKOUT 周期的脉冲。这个输出信号可以有许多用途：可以连接到 RESET 输入引起复位或者连接到 NMI 输入上，引起中断。注意，如果 WDTOUT 连接到 RESET 或 NMI 输入上，就要周期性地对看门狗定时器重新编程，从而使其不会计数到 0。看门狗定时器可以在软件出问题时复位或中断系统。

可编程 DMA 单元

可编程 DMA 单元包含 2 个或 4 个（在 80C186EC/80C188EC 中）DMA 通道。每个通道可以在存储器之间、在存储器和 I/O 之间或者在 I/O 设备之间传送数据。DMA 控制器与第 13 章中讨论过的 DMA 控制器 8237 类似，主要的区别是 DMA 控制器 8237 中有 4 个 DMA 通道，如同 EC 型号中的 DMA 控制器。

可编程片选单元

片选单元是内置的可编程的存储器和 I/O 译码器。在 XL 和 EA 型号中，有 6 个用于存储器选通的输出线，有 7 个用于 I/O 选通的输出线；在 EB 和 EC 型号中，有 10 个输出线用于存储器或者 I/O 选通。

在 XL 和 EA 型号中，存储器选通线分为三组，分别用来选择 80186/80188 存储映射的主要区域。低端存储器片选信号用来选择存放中断向量的存储区。中间存储器片选信号可以选通 4 个位于中间存储区的器件。低端存储器边界起始于 00000H，高端存储器边界结束于 FFFFFH。存储区域的大小是可编程的，并且可以给选通的存储区自动插入等待状态（0 ～ 3 个等待）。

在 XL 和 EA 型号中，每个可编程 I/O 片选信号可寻址 128 个字节的 I/O 块。可编程 I/O 区域起始于由用户编程设定的 I/O 基地址，并且所有 7 个 128 字节的块都是连续的。

在 EB 和 EC 型号中，有 1 个高端存储器片选引脚、1 个低端存储器片选引脚及 8 个通用存储器或I/O 片选引脚。另一个区别在于在这两种型号的 80186/80188 嵌入式控制器中等待状态可被编程为 0 ～ 15 个。

节能/停机特性

节能（power save）功能可以使系统时钟被 4、8 或 16 分频，从而降低电源消耗。该功能由软件启

动，并且由中断等硬件事件终止。停机（power down）功能将完全停止时钟，但在 XL 型号中没有这项功能。通过执行 HLT 指令便可以进入停机方式，遇到中断后会终止并退出停机方式。

刷新控制单元

刷新控制单元以编程设置的时间间隔产生刷新行地址。刷新控制单元没有多路复用 DRAM 的地址——这仍是系统设计者的工作。在可编程的刷新时间间隔的结尾，伴随着$\overline{\text{RFSH}}$控制信号，刷新地址被提供给存储系统。存储系统必须在$\overline{\text{RFSH}}$激活时间内执行刷新周期。有关存储器和刷新的更多信息将在对片选单元进行讲解时给出。

16.1.4 引脚

图 16-2 给出了 80C186XL 微处理器的引脚图。80C186XL 有**无引线芯片载体（leadless chip carrier，LCC）**和**引脚栅格阵列（pin grid array，PGA）**两种封装，如图 16-3 所示。

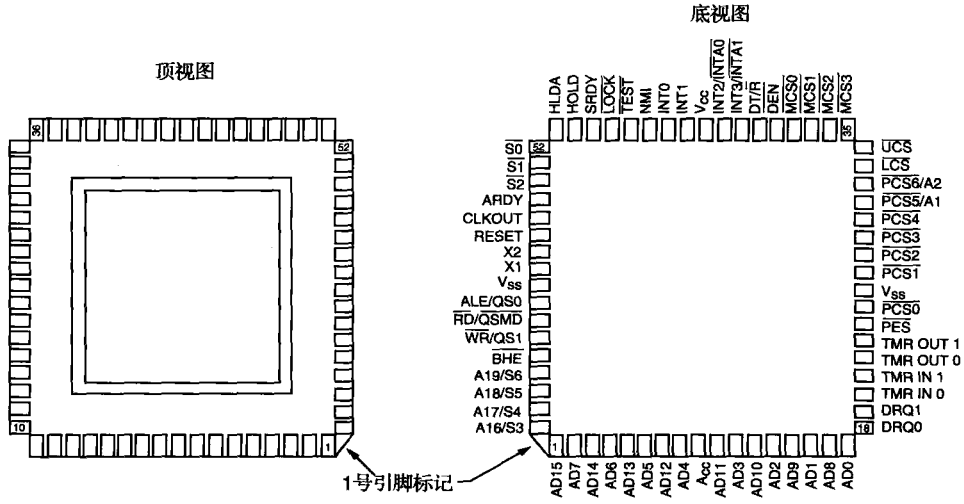


图 16-2 80186 微处理器的引脚图

（由 Intel 公司提供）

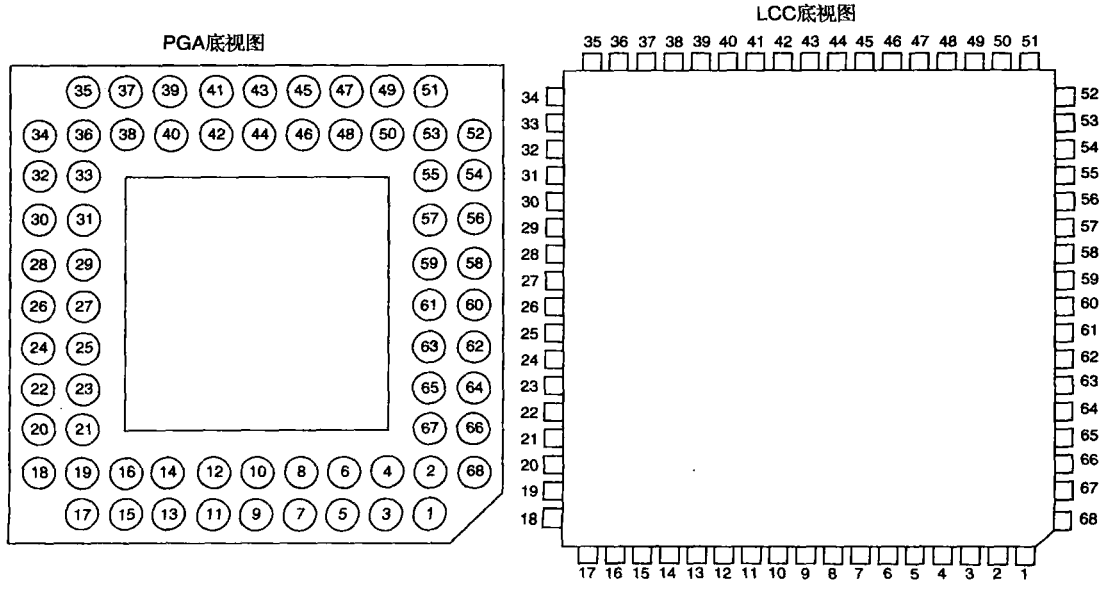


图 16-3 80C188XL 微处理器的 PGA 和 LCC 封装的底视图

引脚定义

下面列出了80C186XL每个引脚的定义，并说明了80C186XL和80C188XL微处理器之间的区别。增强型号将在本章的后面部分中讲述。

V_{CC} 系统+5.0V，±10%**电源**（**power**）接点。

V_{SS} 系统**地**（**ground**）接点。

X_1 和 X_2 **时钟引脚**（**clock pin**）。这两个引脚通常连接到一个基本模式并联谐振的晶体上，由它操纵内部晶体振荡器。外部时钟信号可以连接到 X_1 引脚。内部的主时钟频率是外部晶体或时钟输入信号的一半。注意，在80186/80188的有些型号中，这两个引脚被标记为CLKIN（X_1）和OSCOUT（X_2）。

CLKOUT **时钟输出**（**clock out**）**引脚**。为外围设备提供时序信号，其频率为时钟频率的一半，占空比为50%。

$\overline{RES}$ **复位输入**（**reset input**）**引脚**。用于复位80186/80188。为了正确复位，在上电后必须至少保持50ms的低电平。这个引脚通常连接到产生上电复位信号的RC电路。复位地址与8086/8088微处理器相同，为FFFF0H。

RESET **复位输出**（**reset output**）**引脚**（高电平有效）。它连接到系统外围设备上，每当RES输入为低电平时对这些外围设备进行初始化。

$\overline{TEST}$ 该引脚连接到80187协处理器的BUSY输出。该引脚由FWAIT或WAIT指令来查询。

$T_{in}0$ 和 $T_{in}1$ 这两个引脚作为定时器0和1的**外部时钟源**（**external clocking source**）。

$T_{out}0$ 和 $T_{out}1$ 这两个引脚提供了定时器0和1的**输出信号**（**output signal**），这两个定时器可以被编程来产生方波或脉冲信号。

DRQ0 和 DRQ1 这两个引脚是DMA通道0和1的高电平触发的**DMA请求**（**DMA request**）线。

NMI **非屏蔽中断**（**non-maskable interrupt**）输入引脚。由上升沿触发并始终有效，当NMI被激活时，使用中断向量2。

INT_0、INT_1、INT_2/$\overline{INTA0}$和 INT_3/$\overline{INTA1}$ **可屏蔽中断**（**maskable interrupt**）**输入引脚**。高电平有效，可被编程为电平或边沿触发。这些引脚在没有外部8259的情况下可以被配置为4个中断输入；如果有外部8259，可以被配置为2个中断输入。

A_{19}/$\overline{ONCE}$、A_{18}、A_{17}和 A_{16} 这些引脚是提供地址（A_{19}～A_{16}）和状态（S_6～S_3）的**多路复用的地址状态接点**（**multiplexed address/status connections**）。地址引脚 A_{18}～A_{16}上的状态位没有系统功能，在生产过程中用来测试。A_{19}引脚在复位时作为功能的输入引脚。如果在复位时保持低电平，则微处理器进入测试模式。

AD_{15}～AD_0 **多路复用的地址/数据总线接点**（**multiplexed address/data bus connections**）。在 T_1 周期，80186将地址信号 A_{15}～A_0 输出到这些引脚，在 T_2、T_3 和 T_4 周期，80186又将这些引脚用作数据总线 D_{15}～D_0。注意，80188有 AD_7～AD_0 和 A_{15}～A_8 引脚。

$\overline{BHE}$ **总线高字节使能**（**bus high enable**）引脚，表明（当为逻辑0时）数据总线 D_{15}～D_8 上传送的是有效数据。

ALE **地址锁存使能**（**address latch enable**）。这是一个多路复用的输出引脚，包含了地址锁存允许（ALE）信号，它比8086的ALE早半个时钟周期。用于分离地址/数据和地址/状态总线的多路复用（即使系统不使用 A_{19}～A_{16} 上的状态位，也必须分离复用）。

$\overline{WR}$ **写**（**write**）引脚，该引脚使数据被写入存储器或I/O。

$\overline{RD}$ **读**（**read**）引脚，该引脚使数据从存储器或I/O中读出。

ARDY **异步就绪**（**asynchronous READY**）输入引脚，该输入通知80186/80188，存储器或I/O设备已经就绪，可以读写数据。如果这个引脚接到+5.0V上，则处理器正常工作；如果接地，则微处理器进入等待状态。

SRDY 同步就绪（**synchronous READY**）输入引脚，该输入被系统时钟同步，因此，这个就绪输入的时序要求不是很严格。和 ARDY 一样，SRDY 在不需要等待状态时接到 +5.0V 上。

$\overline{LOCK}$ 由 LOCK 前缀控制的一个输出引脚。如果某条指令加有 LOCK 前缀，则在这个被锁定的指令执行期间，$\overline{LOCK}$引脚为逻辑 0。

S_2、S_1 和 S_0 这 3 个**状态位**（**status bits**）规定了系统正在进行的总线传输的类型。状态位所表示的状态参见表 16-2。

表 16-2 S_2、S_1 和 S_0 的状态位

S_2	S_1	S_0	功 能
0	0	0	中断响应
0	0	1	I/O 读
0	1	0	I/O 写
0	1	1	停机
1	0	0	取操作码
1	0	1	存储器读
1	1	0	存储器写
1	1	1	未定义

$\overline{UCS}$ **高端存储器片选**（**upper-memory chip select**）引脚，用来选通高端存储区。这个输出引脚是可编程的，可以使能大小为 1 KB ～256KB 的结束于 FFFFFH 的存储区。注意，这个引脚的功能在 EB 和 EC 型号中有所不同，可以选通 1KB ～1MB 的存储器。

$\overline{LCS}$ **低端存储器片选**（**lower-memory chip select**）引脚，用来选通起始于 00000H 的存储区。该引脚可以被编程为选择 1 KB ～256KB 大小的存储器。注意，这个引脚的功能在 EB 和 EC 型号中有所不同，可以选通 1KB ～1MB 的存储器。

$\overline{MCS0}$～$\overline{MCS3}$ **中间存储器片选**（**middle-memory chip select**）引脚，用于选通 4 个中间存储器件。这些引脚是可编程的，可以选择 8 KB ～512KB 的包含 4 个器件的存储器块。注意，在 EB 和 EC 型号中没有这些引脚。

$\overline{PCS0}$～$\overline{PCS4}$ 5 个不同的**外围设备片选线**（**peripheral selection lines**）。注意，在 EB 和 EC 型号中没有这些引脚。

$\overline{PCS5}/A_1$ 和$\overline{PCS6}/A_2$ 这些引脚可以被编程作为外围设备片选线或作为内部锁存地址线 A_1 和 A_2。注意，在 EB 和 EC 型号中没有这些引脚。

DT/$\overline{R}$ **数据传送或接收**（**data transmit/receive**）引脚，用于控制连接到系统中的数据总线缓冲器的收发方向。

$\overline{DEN}$ **数据总线使能**（**data bus enable**）引脚，用于选通外部数据总线缓冲器。

16.1.5 直流工作特性

在与微处理器接口或对其进行操作之前，必须要了解它的直流工作特性。80C186/80C188 微处理器需要 42 ～63mA 的电源电流。每个输出引脚上表示逻辑 0 时提供 3.0mA 的电流，表示逻辑 1 时提供 -2.0mA 的电流。

16.1.6 80186/80188 时序

图 16-4 给出了 80186 的时序图。80188 的时序除了多路复用地址的连接（是 AD_7 ～ AD_0 而不是 AD_{15}～ AD_0）和$\overline{BHE}$（80188 无此信号）外，与 80186 完全相同。

80186/80188 的基本时序由 4 个时钟周期组成，类似于 8086/8088。8MHz 的主频对应的总线周期为 500ns，16MHz 对应的为 250ns。

80186/80188 和 8086/8088 的时序差别很小，最突出的差别在于地址锁存允许（ALE）信号在 80186/80188 中早半个时钟周期出现。

存储器存取时间

存储器存取时间是微处理器时序图的要素之一。80186/80188 的存取时间的计算与 8086/8088 相同。我们再回顾一下，存取时间是指微处理器给存储器和 I/O 发出地址之后，分配给存储器和 I/O 的用于为微处理器提供数据的时间。

仔细观察时序图就会发现，从 T_1 周期开始经 T_{CLAV}时间后地址出现在地址总线上。T_{CLAV}在 8MHz 型号

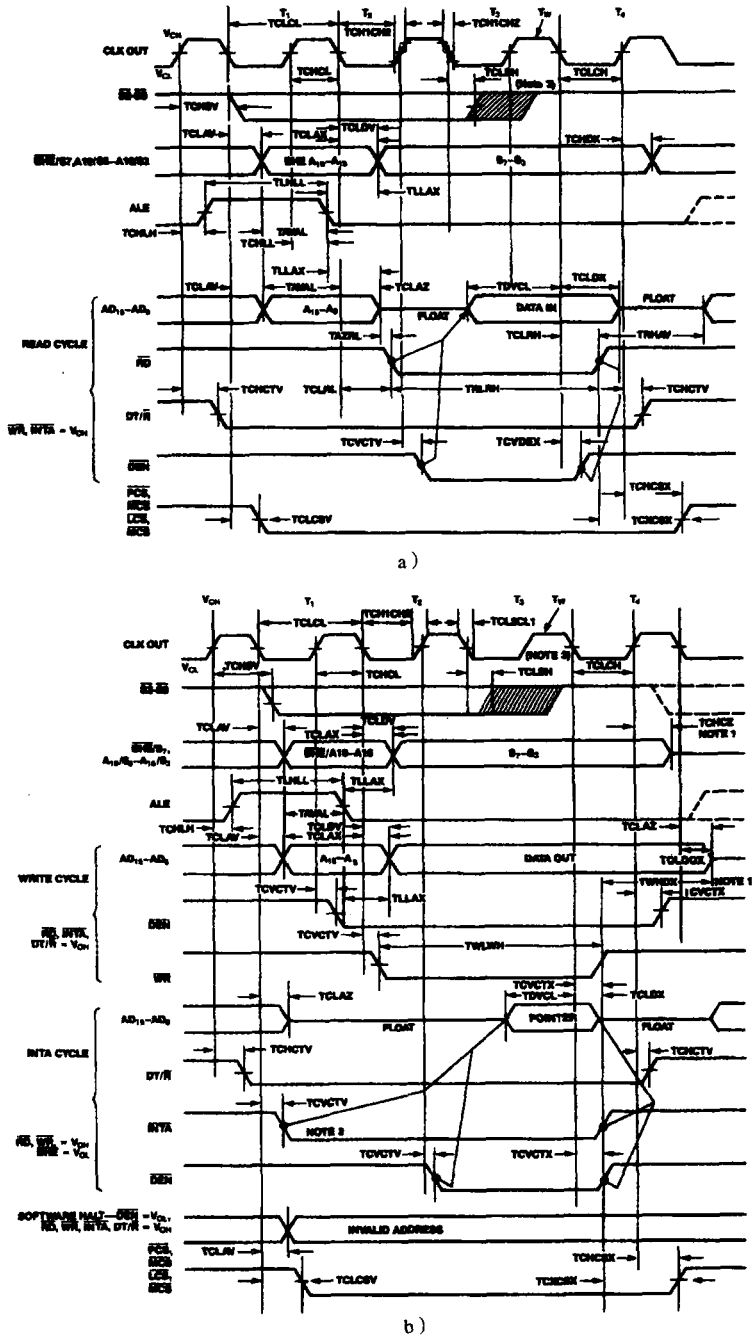


图 16-4 80186/80188 的时序（由 Intel 公司提供）

a）读周期时序 b）写周期时序

的微处理器中为 44ns（见图 16-5）。在 T_3 周期结束时，微处理器从数据总线上采样数据，但在该时钟确立为 T_{DVCL}之前应该有一段建立时间。图 16-5 中列出的两种类型的微处理器的 T_{DVCL}时间都为 20ns。因此，存取时间就等于三个时钟周期减去 T_{CLAV}和 T_{DVCL}。8MHz 微处理器的存取时间为 375ns - 44ns - 20ns =

311ns。16MHz 微处理器的存取时间以同样方式计算，只不过对应的 T_{CLAV} 和 T_{DVCL} 分别是 25ns 和 15ns。

80186 设备接口时序响应

		800188（8MHz）		80188-6（6MHz）			
符号	参数	最小值	最大值	最小值	最大值	单位	测试条件
T_{CLAV}	Address Valid Delay	5	44	5	63	ns	C_L =20～200pF all outputs
T_{CLAX}	Address Hold	10		10		ns	
T_{CLAZ}	Address Float Delay	T_{CLAX}	35	T_{CLAX}	44	ns	
T_{CHCZ}	Command Lines Float Delay		45		56	ns	
T_{CHCV}	Command Lines Valid Delay (after float)		55		76	ns	
T_{LHLL}	ALE Width	$T_{CLCL-35}$		$T_{CLCL-35}$		ns	
T_{CHLH}	ALE Active Delay		35		44	ns	
T_{CHLL}	ALE Inactive Delay		35		44	ns	
T_{LLAX}	Address Hold to ALE Inactive	$T_{CHCL-25}$		$T_{CHCL-30}$		ns	
T_{CLDV}	Data Valid Delay	10	44	10	55	ns	
T_{CLDOX}	Data Hold Time	10		10		ns	
T_{WHDX}	Data Hold after WR	$T_{CLCL-40}$		$T_{CLCL-50}$		ns	
T_{CVCTV}	Control Active Delay1	5	70	5	87	ns	
T_{CHCTV}	Control Active Delay2	10	55	10	76	ns	
T_{CVCTX}	Control Inactive Delay	5	55	5	76	ns	
T_{CVDEX}	$\overline{DEN}$Inactive Delay (Non-Write Cycle)		70		87	ns	
T_{AZRL}	Address Float to $\overline{RD}$ Active	0		0		ns	
T_{CLRL}	$\overline{RD}$Active Delay	10	70	10	87	ns	
T_{CLRH}	$\overline{RD}$Inactive Delay	10	55	10	76	ns	
T_{RHAV}	$\overline{RD}$ Inactive to Address Active	$T_{CLCL-40}$		$T_{CLCL-50}$		ns	
T_{CLHAV}	HLDA Valid Delay	10	50	10	67	ns	
T_{RLRH}	$\overline{RD}$ Width	$2T_{CLCL-50}$		$2T_{CLCL-50}$		ns	
T_{WLWH}	$\overline{WR}$ Width	$2T_{CLCL-40}$		$2T_{CLCL-40}$		ns	
T_{AVAL}	Address Valid to ALE Low	$T_{CLCH-25}$		$T_{CLCH-45}$		ns	
T_{CHSV}	Status-Active Delay	10	55	10	76	ns	
T_{CLSH}	Status Inactive Delay	10	55	10	76	ns	
T_{CLTMV}	Timer Output Delay		60		75	ns	最大 100pF
T_{CLRO}	Reset Delay		60		75	ns	
T_{CHQSV}	Queue Status Delay		35		44	ns	

80186 片选时序响应

符号	参数	最小值	最大值	最小值	最大值	单位	测试条件
T_{CLCSV}	Chip-Select Active Delay		66		80	ns	
T_{CXCSX}	Chip-Selct Hold from Command Inactive	35		35		ns	
T_{CHCSX}	Chip-Select Inactive Delay	5	35	5	47	ns	

符号	参数	最小值	最大值	单位	测试条件
TDVCL	Data in Setup（A/D）	20		ns	
TCLDX	Data in Hold（A/D）	10		ns	
TARYHCH	Asynchronous Ready（AREADY）active setup time*	20		ns	
TARYLCL	AREADY inactive setup time	35		ns	
TCHARYX	AREADY hold time	15		ns	
TSRYCL	Synchronous Ready（SREADY）transition setup time	35		ns	
TCLSRY	SREADY transition hold time	15		ns	
THVCL	HOLD Setup*	25		ns	
TINVCH	INTR, NMI, TEST, TIMERIN, Setup*	25		ns	
TINVCL	DRQ0, DRQ1, Setup*	25		ns	

* 为确保下一时钟周期的识别。

图 16-5 80186 的交流特性（由 Intel 公司提供）

16.2 80186/80188 增强功能编程

本节给出了80186/80188所有型号（XL、EA、EB和EC）的增强功能的编程和操作细节。下一节将详述80C188EB在一个系统中的运用，该系统将用到许多这里所讨论过的增强功能。本节惟一没有讨论的新功能是时钟发生器，在上一节体系结构的介绍中已经描述过它。

16.2.1 外设控制块（PCB）

所有的片内外设由一组位于外设控制块（PCB）的16位宽的寄存器控制。PCB（如图16-6所示）是位于I/O或存储器空间的一组寄存器，共256个。注意，这组寄存器适用于XL和EA型号。本节后面的部分将定义和描述EB和EC型号的PCB。

80186/80188每次复位，外设控制块都会自动定位到I/O空间的最高端（I/O地址为FF00H～FFFFH）。大多数情况下，PCB都固定在I/O空间的这个区域，但PCB可以随时重定位到任意的存储器或I/O空间。可以通过改变偏移地址为FEH和FFH的重定位寄存器（参见图16-7）的值实现重定位。

重定位寄存器在80186/80188复位后的默认值为20FFH，将PCB定位到地址为FF00H～FFFFH的I/O空间。如果要重定位PCB，用户只需向I/O地址FFFEH处OUT一个新的位模式。例如，要将PCB重定位到存储器地址20000H～200FFH处，只需向I/O地址FFFEH发送1200H。要注意，M/$\overline{IO}$为逻辑1表示重定位到存储器空间，200H表示以存储器地址20000H作为PCB的基地址。对PCB的所有访问必须以字为单位，因为这些寄存器都是按16位宽度组织的。例16-1给出了将PCB重定位到存储器地址20000H～200FFH上所需的程序。对80186编程，既可以使用8位也可以使用16位的输出，而在80188中不要使用OUT DX，AX指令，因为该指令的执行需要附加的时钟周期。

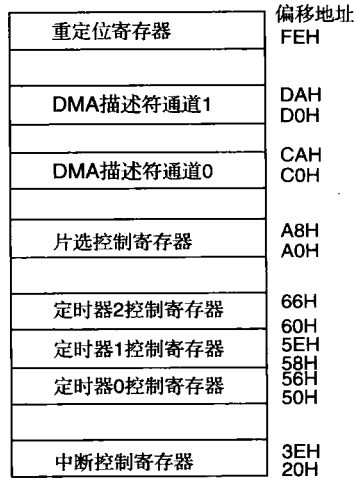


图16-6 80186/80188的外设控制块（PCB）（由Intel公司提供）

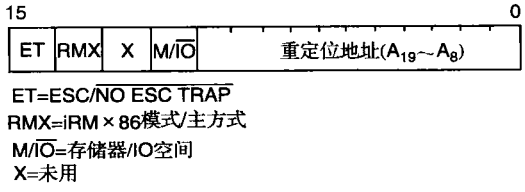


图16-7 外设控制寄存器

例16-1

```
MOV  DX,0FFFEH          ;寻址重定位寄存器
MOV  AX,1200H           ;新PCB定位码
OUT  DX,AL              ;也可以是OUT DX,AX
```

在EB和EC型号中，用于PCB单元的编程地址有所不同。这两个型号都将PCB重定位信息存放在偏移地址为XXA8H的存储单元中，而不是XL和EA型号中的XXFEH。这两个型号的位模式除了没有RMX位，其余位与XL和EA型号相同。

16.2.2 80186/80188的中断

除去80186/80188中为内部设备定义的附加的中断向量以外，80186/80188中断与8086/8088中断相同。表16-3列出了所有保留的中断向量。前5个与8086/8088相同。

如果变址寄存器边界超过了存储器的设定值，数组的BOUND指令就会产生中断。如果80186/80188执行了未定义的操作码，就会产生未定义操作码中断。在程序跑飞时，这个功能非常有用。未

定义操作码中断可以用指令访问，但汇编器没有把这条指令包括在指令集中。在 Pentium Pro ～ Pentium 4及早期的 Intel 微处理器中，0F0BH 或 0FB9H 指令会引起未定义操作码中断服务程序的调用。

表 16-3 80186/80188 中断向量

名　称	类　型	地　址	优 先 级
除法错	0	00000 ～00003	1
单步断点	1	00004 ～00007	1A
NMI 引脚	2	00008 ～0000B	1
断点	3	0000C ～0000F	1
溢出	4	00010 ～00013	1
BOUND 指令	5	00014 ～00017	1
未用操作码	6	00018 ～0001B	1
ESC 操作码	7	0001C ～0001F	1
定时器 0	8	00020 ～00023	2A
保留	9	00024 ～00027	—
DMA0	A	00028 ～0002B	4
DMA1	B	0002C ～0002F	5
INT_0	C	00030 ～00033	6
INT_1	D	00034 ～00037	7
INT_2	E	00038 ～0003B	8
INT_3	F	0003C ～0003F	9
80187 协处理器	10	00040 ～00043	1
保留	11	00044 ～00047	—
定时器 1	12	00048 ～0004B	2B
定时器 2	13	0004C ～0004F	2C
串行接收器	14	00050 ～00053	3A
串行发送器	15	00054 ～00057	3B

注：优先级 1 最高，优先级 9 最低。某些中断优先级相同。只有 EB 和 EC 型号有串行口。

执行 ESC 操作码 D8H ～ DFH 会产生 ESC 操作码中断。这种情况仅在重定位寄存器的 ET（escape trap）位被置位时才会发生。如果出现 ESC 中断，该中断存放在堆栈上的地址指向 ESC 指令或段超越前缀（如果使用了段超越前缀）。

内部的硬件中断必须由 I 标志位使能并且功能不可屏蔽。I 标志用 STI 设置（使能中断），用 CLI 清除（禁止中断）。其余的内部译码中断将在本节后面的定时器和 DMA 控制器中进行讨论。

16.2.3 中断控制器

80186/80188 内部的中断控制器是一个复杂的部件。它有多个中断输入，分别来自 5 个外部中断、DMA 控制器和 3 个定时器。图 16-8 是 80186/80188 中断控制器的中断结构框图。XL、EA 和 EB 型号中都含有这种中断控制器，而在 EC 型号中含有 2 个与第 12 章中所介绍的 8259A 完全相同的中断控制器。在 EB 型号中，DMA 的中断输入被串行收发单元的中断输入所替代。

中断控制器有两种工作模式：主模式和从模式。工作模式由中断控制寄存器（EB 和 EC 型号）中的 CAS 位选择，如果 CAS 位为 1，中断控制器就连接了外部的 8259A 可编程中断控制器（如图 16-9 所示）；如果 CAS 位为 0，那么就选择了内部的中断控制器。在大多数情况下，80186/80188 内部的中断就足够了，因此一般不用从模式。在 XL 和 EA 型号中，主、从模式的选择在偏移地址为 FEH 的外设控制寄存器中进行。

这部分内容没有详述中断控制器的编程，相反，仅对中断控制器内部结构进行了讨论。中断控制器的编程和应用将在描述定时器和 DMA 控制器的章节中讨论。

中断控制器中的寄存器

中断控制器中的寄存器如图 16-10 所示。这些寄存器位于外设控制块，起始偏移地址为 22H。在 EC 型号中，它的中断控制器与 8259A 兼容，其中主中断控制器的端口偏移地址为 00H 和 02H，从中断控制器的端口地址为 04H 和 06H。在 EB 型号，中断控制器编程的偏移地址为 02H。还要注意 EB 型号

中有一个附加的中断输入（INT4）。

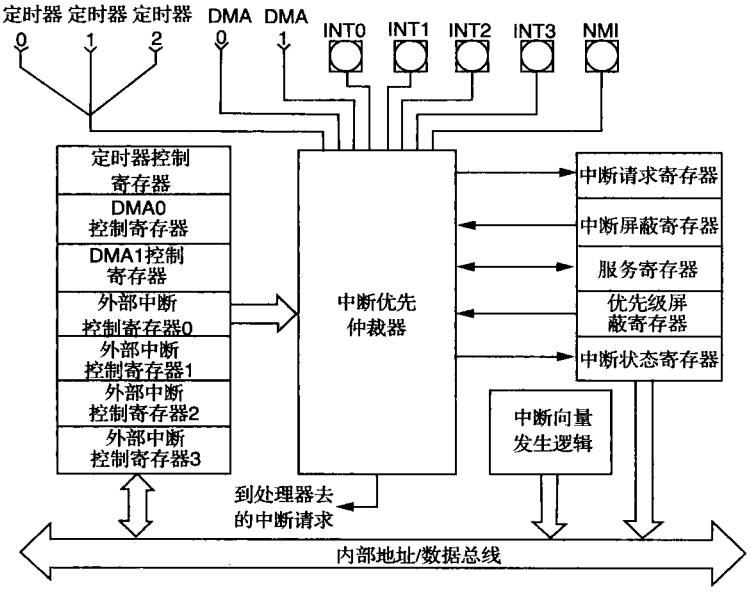


图 16-8　80186/80188 可编程中断控制器（由 Intel 公司提供）

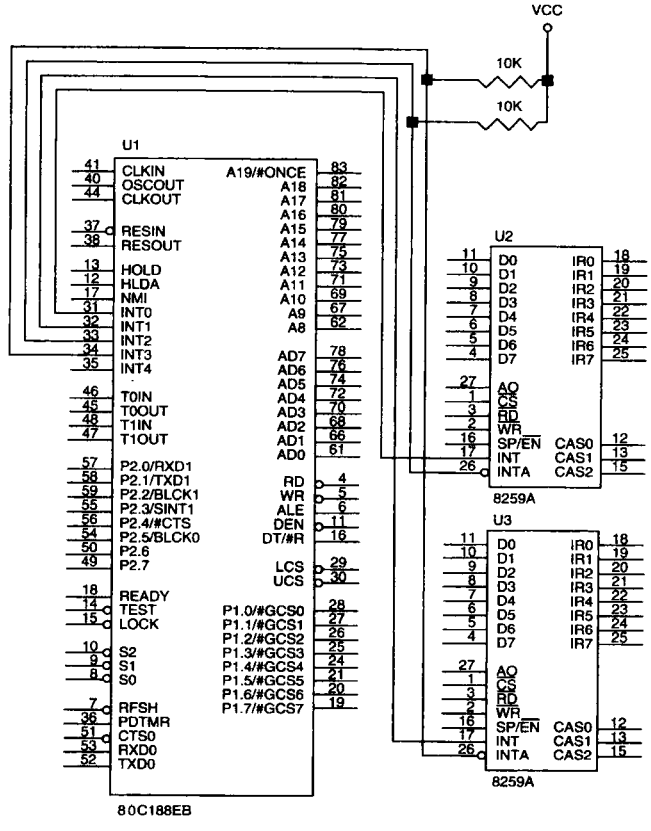


图 16-9　80C188EB 与两个可编程中断控制器间的互连（由 Intel 公司提供）

注：只给出对应接口必需的连接。

从模式

当中断控制器工作在从模式时，它可以使用两个以上的外部 8259A 可编程中断控制器扩充中断输入。图 16-9 举例说明了如何把外部中断控制器接到 80186/80188 的中断输入引脚上并以从模式工作。这里，INT_0 和 INT_1 的输入作为外部 8259 的中断请求，$\overline{INTA0}$（INT_2）和$\overline{INTA1}$（INT_3）被用作外部中断控制器的响应信号。

中断控制寄存器

在两种工作模式下，中断控制器都有一组中断控制寄存器，这些寄存器各自控制着一个中断源。图 16-11 描述了每个中断控制寄存器的二进制位格式。屏蔽位用来使能（为 0 时）或禁止（为 1 时）控制字所代表的中断输入，优先级字段用来设置该中断的优先级。最高优先级为 000，最低为 111。CAS 位用来使能从模式或级联模式（0 为从模式）。SFNM 位用来选择**特定全级联方式（special full nested mode, SFNM）**，SFNM 使得 8259A 的优先级结构被保持。

	XL和EA型号		EB型号
3EH	INT3控制寄存器	1EH	INT3控制寄存器
3CH	INT2控制寄存器	1CH	INT2控制寄存器
3AH	INT1控制寄存器	1AH	INT1控制寄存器
38H	INT0控制寄存器	18H	INT0控制寄存器
36H	DMA1控制寄存器	16H	INT4控制寄存器
34H	DMA0控制寄存器	14H	串行控制寄存器
32H	定时器控制寄存器	12H	定时器控制寄存器
30H	中断状态	10H	中断状态
2EH	请求	0EH	请求
2CH	中断服务	0CH	中断服务
2AH	优先级屏蔽	0AH	优先级屏蔽
28H	中断屏蔽	08H	中断屏蔽
26H	POLL状态	06H	POLL状态
24H	POLL	04H	POLL
22H	EOI	02H	EOI

图 16-10 中断控制单元 I/O 端口偏移值分配

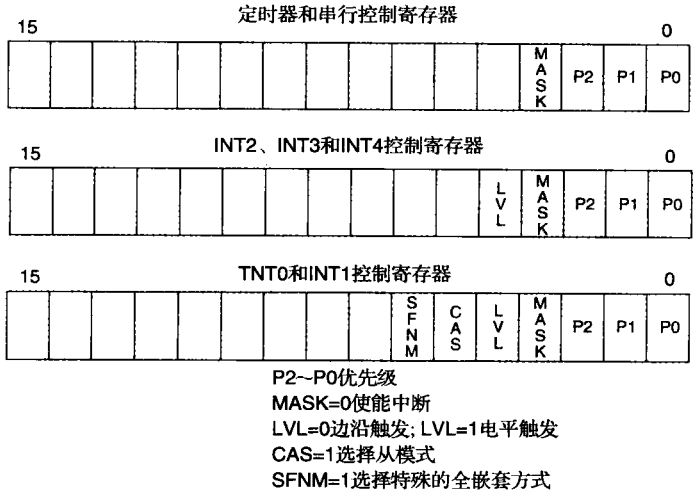


图 16-11 中断控制寄存器

中断请求寄存器

中断请求寄存器包含各种工作模式下中断源的映像。一旦有中断请求，即使该中断已被屏蔽，对应的中断请求位也会被置为逻辑 1。当 80186/80188 响应该中断后，请求位会自动清除。图 16-12 给出了中断请求寄存器的二进制位格式，它适用于主模式和从模式。

屏蔽寄存器和优先级屏蔽寄存器

中断屏蔽寄存器与图 16-12 中的中断请求寄存器的格式相同。如果某中断源被屏蔽（禁止），中断屏蔽寄存器中相应的位应设置为 1；如果被使能，则相应的位设置为 0。读中断屏蔽寄存器可以确定哪

些中断源被屏蔽而哪些未被屏蔽。一个中断源可以通过设置中断屏蔽寄存器中该中断源对应的屏蔽位实现屏蔽。

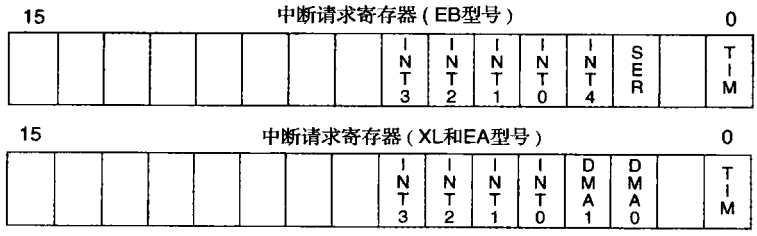


图 16-12 中断请求寄存器

优先级屏蔽寄存器如图 16-13 所示。优先级屏蔽寄存器表明了 80186/80188 当前正在处理的中断的优先级。该中断的级别由优先级位 $P_2 \sim P_0$ 来表示。这些位可以防止在该中断处理过程中低优先级中断的发生。当 80186/80188 发出中断处理结束命令时，这些位会自动设置为下一优先级。如果当前没有等待处理的中断，这些位就被置为 111，允许响应所有优先级的中断。

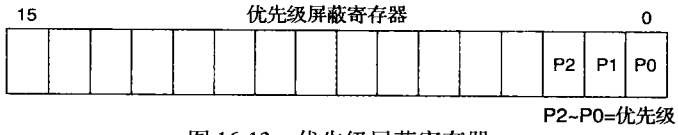


图 16-13 优先级屏蔽寄存器

中断服务寄存器

中断服务寄存器和图 16-12 中所示的中断请求寄存器有相同的二进制位格式。如果 80186/80188 正在响应某个中断，该中断源对应的位就会被置位；在中断结束时，该位被复位。

轮询和轮询状态寄存器

中断轮询寄存器和中断轮询状态寄存器都具有如图 16-14 所示的二进制位模式。这些寄存器都有 1 位（INT REQ）指示有无中断挂起。如果接收到一个具有足够优先级的中断，INT REQ 位就被置位；当中断响应时，它就被复位。VT4 ～ VT0 位用来指示悬而未决的最高优先级的中断向量类型号。

由于所包含的信息相同，轮询寄存器和轮询状态寄存器看上去是一样的。然而它们的功能不同，当中断轮询寄存器被读时，中断就被响应；而当中断轮询状态寄存器被读时，并不响应中断。这两个寄存器只在主模式下使用，不能用在从模式下。



图 16-14 轮询和轮询状态寄存器

中断结束（EOI）寄存器

当程序向中断结束（EOI）寄存器中写数时，会引起中断的终止。图 16-15 所示的是主模式和从模式下 EOI 寄存器的内容。

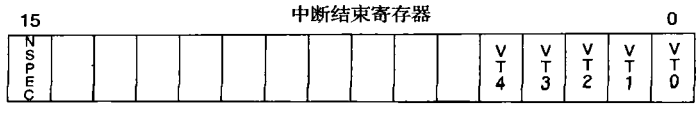


图 16-15 中断结束（EOI）寄存器

在主模式下，对 EOI 寄存器写数，可以结束指定优先级（向量号）的中断，或结束当前正在处理（未指定优先级）的中断而不论它是哪一优先级。在未指定优先级模式下，要结束未指定优先级的中断，在向 EOI 寄存器送数之前，必须先将 NSPEC 位置位。未指定优先级的 EOI 将清除中断服务寄存器中最高优先级的中断位。指定优先级的 EOI 将清除中断服务寄存器中所选定的位，通知微处理器那个中断已被服务，可以接收其他同样类型的中断。除了需要不同中断响应顺序的特定情况外，都用未指定模式。如果需要指定的 EOI，向量号就放在 EOI 命令中。例如清除定时器 2 中断，EOI 命令是 13H（定时器 2 的向量）。

在从模式下，把要终止的中断的优先级送到 EOI 寄存器。从模式不允许未指定方式的 EOI。

中断状态寄存器

中断状态寄存器的格式如图 16-16 所示。在主模式下，$T_2 \sim T_0$ 位指出哪个定时器（定时器 0、定时器 1 或定时器 2）引起了中断。由于 3 个定时器有相同的中断优先级，因此这一点非常必要。当定时器请求中断时，对应位被置位。当中断被响应后，对应位被清除。DHLT（DMA 停止）位只在主模式下使用；当置位时终止 DMA 的工作。需注意，EB 型号中的中断状态寄存器不同。

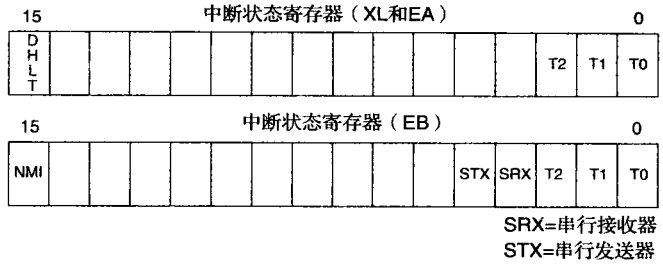


图 16-16 中断状态寄存器

中断向量寄存器

中断向量寄存器只在 XL 和 EA 的从模式下才有效，其偏移地址为 20H。它用来指定中断向量类型号的最高 5 位。图 16-17 说明了这个寄存器的格式。

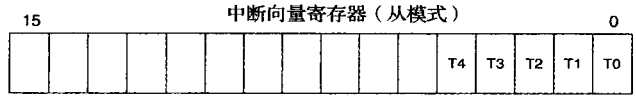


图 16-17 中断向量寄存器

16.2.4 定时器

80186/80188 微处理器中含有三个完全可编程的 16 位定时器，这三个定时器完全独立。其中两个定时器（定时器 0 和定时器 1）有输入和输出引脚，可以用来对外部事件进行计数或产生波形。第三个定时器（定时器 2）接到 80186/80188 的时钟上，它可用作 DMA 请求源或作为其他定时器的时钟，还可用作看门狗定时器。

图 16-18 所示的是定时器部件的内部结构。从图中可以看出，定时器部件中有一个计数单元，它负责对三个计数器的更新。每个定时器实际上就是一个由计数单元不断重写的寄存器（计数单元是一个实现从定时器寄存器读取值然后加 1 回写的电路）。计数单元还负责产生 $T0_{OUT}$ 和 $T1_{OUT}$ 引脚的输出以及读取 $T0_{IN}$ 和 $T1_{IN}$ 引脚的输入，如果定时器 2 被编程用作 DMA 请求，那么计数单元还可以由定时器 2 的终止计数（TC）引起 DMA 请求。

定时器寄存器操作

定时器由外设控制块中的一组寄存器来控制（见图 16-19）。每个定时器都有一个计数寄存器，一个或多个最大计数值寄存器和一个控制寄存器。这些寄存器可以随时被读写，因为 80186/80188 微处理器能够确保在读写过程中内容不变。

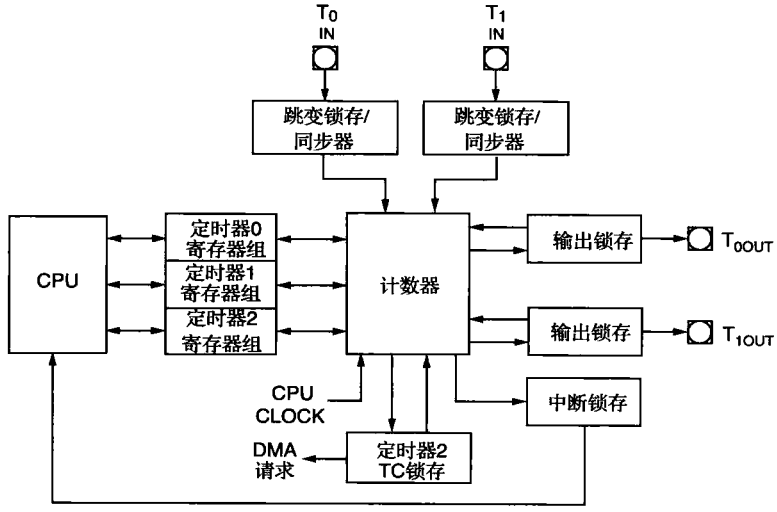


图 16-18 80186/80188 定时器内部结构（由 Intel 公司提供）

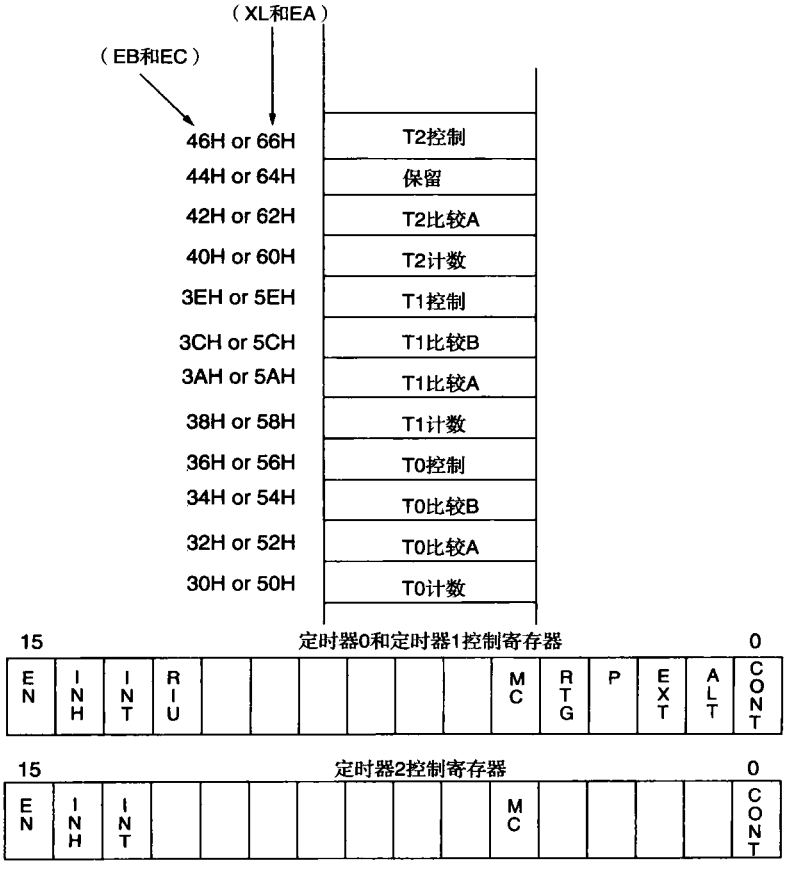
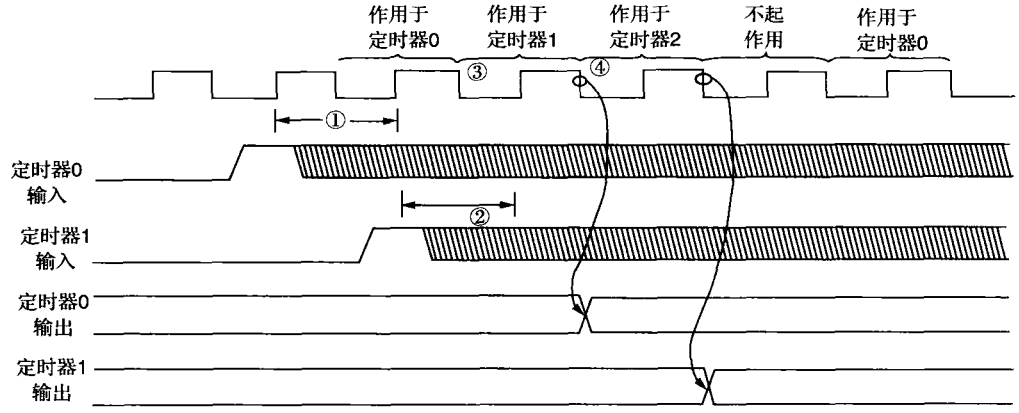


图 16-19 定时器控制寄存器的偏移地址和内容

定时器计数寄存器包含了一个16位的数，每当定时器有输入时，这个数就会自动加1。在外部输入引脚上有上升沿信号时，或每隔4个80186/80188时钟，或定时器2输出触发时，定时器0和定时器1被加1。定时器2每4个80186/80188时钟计时一次，它没有其他的定时信号源。这意味着在主频为8MHz的80186/80188中，定时器2工作频率为2MHz，定时器0和定时器1的最大计数频率为2MHz。图16-20描述了这4个时钟周期，它们与总线时序无关。



1. 定时器0计数。
2. 定时器1计数。
3. 已修改的计数值写入80186定时器0的计数寄存器。
4. 已修改的计数值写入80186定时器1的计数寄存器。

图16-20 80186/80188定时器时序（由Intel公司提供）

每个定时器至少有一个最大计数值寄存器，称为比较寄存器（在定时器0和定时器1中称为比较寄存器A），该寄存器装有计数寄存器产生输出的最大计数值。注意，定时器是向上计数的计数器。当计数寄存器的计数值等于比较寄存器的值时，计数寄存器被清零。当最大值为0000H时，计数器将计数65 536次。如果最大值为其他值，定时器就按真正的计数值计数。例如，如果最大计数值为0002H，那么计数器将从0计数到1然后清零（一个模2计数器只有2种状态）。

定时器0和定时器1各自还有第2个最大值比较寄存器（比较寄存器B），它由控制寄存器来选择。只用比较寄存器A还是同时使用比较寄存器A和B，这由定时器控制寄存器中的ALT位来确定。当两个最大值比较寄存器都使用时，定时器先计数到最大值比较寄存器A中的值后清零，接着计数到最大值比较寄存器B中的值后又清零，就这样一直反复着。同时使用两个最大值比较寄存器可以使定时器计数到131 072。

每个定时器的控制寄存器（参见图16-19）都是16位，用于指定定时器的操作。每个控制位定义如下：

EN **使能（enable）**位。决定是否允许定时器开始计数。若EN位被清零，定时器将不能计数；若EN位被置位，则开始计数。

INH **禁止（inhibit）**位。决定对定时器控制寄存器的写操作是否影响使能（EN）位。如果INH被置位，则EN位可以被置位或清零以控制计数。如果INH被清零，则对定时器控制寄存器的写操作将不会影响EN位。这使得在不改变定时器的使能或禁止状态的情况下，改变定时器的其他特征。

INT **中断（interrupt）**位。决定是否允许定时器产生中断。如果INT位被置位，则每当计数值达到两个最大值比较寄存器中任意一个的值时，就会产生一个中断。如果INT位被清零，就不会产生中断。当产生一个中断请求后，即使EN位被清零，INT位仍然有效。

RIU **正在使用寄存器（register in use）**位。它表示当前正在使用哪一个最大值比较寄存器。如果

RIU 为逻辑 0，则最大值比较寄存器 A 正在被使用。这是一个只读位，写操作不会影响它。

MC **最大计数值（maximum count）**位。它指示定时器是否已经达到它的最大值。当定时器达到最大值时，MC 位就变成逻辑 1 并保持下去，直到被写入逻辑 0。这使得可以用软件来检查是否达到最大计数值。

RTG **重新触发（retrigger）**位。RTG 位只对内部时钟（EXT = 0）有效。RTG 位只用在定时器 0 和定时器 1 中，以选择输入引脚（$T0_{IN}$和 $T1_{IN}$）的作用。如果 RTG 为逻辑 0，外部输入为逻辑 1 时将使定时器开始计数，如果外部输入为逻辑 0，定时器将保持计数值（停止计数）；如果 RTG 为逻辑 1，在外部的输入引脚的每个上升沿，定时器的计数值将被清为 0000H。

P **预定标器（prescaler）**位。用来选择定时器 0 和定时器 1 的时钟信号源。如果 EXT = 0 且 P = 0，则以系统时钟频率的 1/4 作为时钟信号源；如果 EXT = 0 且 P = 1，则以定时器 2 的输出作为时钟信号源。

EXT **外部（external）**位。用来选择是内部定时（EXT = 0）还是外部定时（EXT = 1）。如果 EXT = 1，采用 $T0_{IN}$或 $T1_{IN}$引脚的输入作为定时信号源。在这种方式下，在输入引脚的每个上升沿，定时器加 1。如果 EXT = 0，则选择内部的一个时钟源作为定时时钟。

ALT **交替（alternate）**位。如果为逻辑 0，则选择单个最大计数值方式（只选择最大值比较寄存器 A）；如果为逻辑 1，则选择交替最大计数值方式（选择最大值比较寄存器 A 和 B）。

CONT **连续（continuous）**位。如果为逻辑 1，选择连续工作方式。在连续工作方式下，计数器在计数到最大值以后会自动重新开始计数。如果 CONT 位为逻辑 0，定时器将自动停止计数并将 EN 位清零。注意，当 80186/80188 复位时，定时器自动被禁止。

定时器输出引脚

定时器 0 和定时器 1 有一个用于产生方波或脉冲的输出引脚。若要产生脉冲，定时器需运行在单个最大计数值方式下（ALT = 0）。在这种方式下，当计数器达到最大值时，输出引脚就会输出长度为一个时钟周期的低电平信号。通过控制控制寄存器中的 CONT 位，可以产生单个或连续脉冲。

若要产生方波或不同占空比的波形，则需要选择交替方式（ALT = 1）。在这种方式下，在最大值比较寄存器 A 控制着定时器时，输出引脚为逻辑 1；在最大值比较寄存器 B 控制着定时器时，输出引脚为逻辑 0。与单个最大计数值方式一样，定时器既可以产生单个方波也可以产生连续方波。ALT 和 CONT 控制位的功能参见表 16-4。

表 16-4 定时器控制寄存器中的 ALT 和 CONT 位的功能

ALT	CONT	方式
0	0	单脉冲
0	1	连续脉冲
1	0	单方波
1	1	连续方波

在交替方式下，几乎可以产生任意占空比的波形。例如，假如要在输出引脚产生 10% 的占空比波形，在最大值比较寄存器 A 中装入 10，在最大值比较寄存器 B 中装入 90，即可产生 10 个时钟的逻辑 1 和 90 个时钟的逻辑 0。这样也就将定时信号源被 100 分频。

实时钟举例

许多系统需要用到日时钟，通常称为**实时钟（real-time clock）**。80186/80188 内的定时器可以为维护日时钟的软件提供定时信号源。

这个应用所需的硬件没有用图说明。只需要将 $T1_{IN}$引脚通过上拉电阻接到 +5.0V，使定时器 1 工作。在这个例子中，定时器 1 和定时器 2 用来产生 1 秒的中断，作为软件的定时信号源。

实现实时钟所需的软件如例 16-2 和例 16-3 所示。例 16-2 是定时器的初始化软件。例 16-3 是一个中断服务程序，用来走时对准。在例 16-3 中还有另外一个程序，用于增加 BCD 模计数器。这里没有举例说明中断向量和日时钟安装或显示所需的软件。

例 16-2

```
;该软件为 80186/80188EB 写的,初始化并启动定时器 1 和定时器 2
;地址为
```

```
T2_CA  EQU   0FF42H            ;定时器2比较寄存器A
T2_CON EQU   0FF46H            ;定时器2控制寄存器
T2_CNT EQU   0FF40H            ;定时器2计数寄存器
T1_CA  EQU   0FF3AH            ;定时器1比较寄存器A
T1_CON EQU   0FF38H            ;定时器1控制寄存器
T1_CNT EQU   0FF3EH            ;定时器1计数寄存器

       MOV  AX,20000           ;编程定时器2为10ms
       MOV  DX,T2_CA
       OUT  DX,AX

       MOV  AX,100             ;编程定时器1为1s
       MOV  DX,T1_CA
       OUT  DX,AX

       MOV  AX,0               ;清除计数寄存器
       MOV  DX,T2_CNT
       OUT  DX,AX
       MOV  DX,T1_CNT
       OUT  DX,AX

       MOV  AX,0C001H          ;使能定时器2并启动它
       MOV  DX,T2_CON
       OUT  DX,AX

       MOV  AX,0E009H          ;使能带中断的定时器并启动它
       MOV  DX,T1_CON
       OUT  DX,AX
```

定时器2被编程除以20 000。这使得它的时钟（假定在8MHz的80186/80188中为2MHz）被分频为每10ms产生一个脉冲。内部定时器1的时钟输入则来自定时器2的输出。定时器1被编程除以100，这样每秒钟可以产生一个脉冲。通过对定时器1的控制寄存器进行编程，便可以使每秒一次的脉冲能够在内部产生中断。

中断服务程序每秒调用一次以实现走时。这个中断服务程序将存储单元SECONDS中的值加1。每过60秒，将下一个存储单元（SECONDS+1）中的值加1。最后，每过一小时，将存储单元SECONDS+2中的值加1。时间以BCD码形式保存在这3个连续的存储单元中，因此系统软件可以很容易地访问时间。

例16-3

```
SECONDS      DB  ?
MINUTES      DB  ?
HOURS        DB  ?

INTRS  PROC  FAR  USES  DS  AX  SI

       MOV   AX,SEGMENT_ADDRESS
       MOV   DS,AX
       MOV   AH,60H                        ;装载模数60
       MOV   SI,OFFSET  SECONDS            ;寻址时钟
       CALL  UPS                           ;秒计数加1
       .IF  ZERO?                          ;如果秒计数变为0
            CALL UPS                       ;分计数加1
            MOV  AH,24H                    ;装载模数24
            .IF ZERO?                      ;如果分计数变为0
                 CALL  UPS                 ;小时数加1
            .ENDIF
       .ENDIF
       MOV  DX,0FF02H                      ;清除中断
       MOV  AX,8000H
       OUT  DX,AX
```

```
        RET
INIRS   ENDP
UPS     PROC  NEAR
        MOV  AL,[SI]
        ADD  AL,1                          ;计数器加1
        DAA                                ;转换为 BCD 码
        INC   SI
        .IF   AL = =AH                     ;测试模数
              MOV  AL,0
        .ENDIF
        MOV  [SI-1],AL
        RET
UPS     ENDP
```

16.2.5 DMA 控制器

80186/80188 内的 DMA 控制器有两个完全独立的 DMA 通道，每个通道都有一套自己的 20 位地址寄存器，所以 DMA 传输可以访问任意内存或 I/O 空间。另外，每个通道都可以将源寄存器或者目的寄存器编程为自动加或自动减方式。在 EB 或 EC 型号中没有这个控制器。EC 型号含有一个修改过的 4 个通道的 DMA 控制器，而 EB 型号中没有 DMA 控制器。本文不描述 EC 型号的 DMA 控制器。

图 16-21 显示了 DMA 控制器的内部寄存器结构。这组寄存器位于外设控制块，偏移地址为C0H ～ DFH。

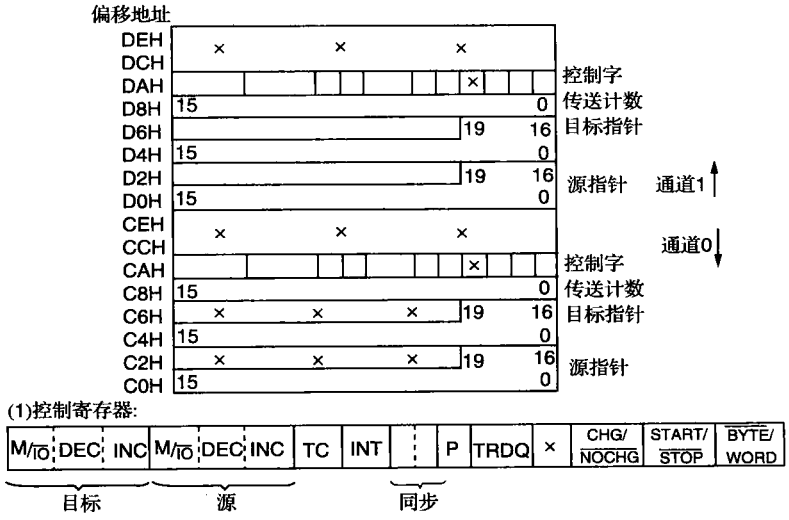


图 16-21 80186/80188 DMA 控制寄存器结构（由 Intel 公司提供）

注意，两个 DMA 通道的寄存器组是相同的。每个通道都有一个控制字、一个源指针、一个目的指针和一个传送计数器。传送计数器是一个 16 位的寄存器，允许 DMA 以字节方式（80186/80188）或字方式（仅限于 80186）自动传输。每传输一个字节或一个字，计数器就会自动减 1，直到减到 0000H——计算终点。

源指针和目的指针都是 20 位的寄存器，因此，DMA 传输可以在任意内存和 I/O 地址空间进行，而不必关心段和偏移地址。如果源或目的地址是 I/O 端口，A_{19}～A_{16}必须为 0000，否则会出错。

通道控制寄存器

每个 DMA 通道都有自己的通道控制寄存器（见图 16-21），用来定义通道的操作。最左边的 6 位指定了源和目的寄存器的操作类型。M/$\overline{IO}$位用于表示是存储器还是 I/O 空间，DEC 位置 1 可以使指针递减，INC 位置 1 可以使指针递加。如果 INC 位和 DEC 位都被置 1，每次 DMA 传输后指针寄存器的值将保持不变。需要注意的是，用 DMA 控制器可以实现存储器到存储器的传输。

TC（终值）位用来确定 DMA 通道在计数寄存器减到 0000H 时是否停止传输。如果这一位被置位为逻辑 1，DMA 控制器在到达终值 0000H 后还将继续传输数据。

INT 位用来使能到中断控制器的中断。如果被置位，INT 位在通道计数器计数到终值时，就会产生中断。

SYN 位用来选择通道的同步类型：00 = 非同步，01 = 源同步，10 = 目的同步。选择非同步或源同步方式时，数据将以 2MB/S 的速率传输，这两种同步允许不中断地传输。如果选择目的同步，那么传输率就比较低（1.3MB/S），DMA 控制器在每次 DMA 传输后都将释放控制权给 80186/80188 处理器。

P 位用来选择通道的优先级。如果 P = 1，通道就有最高优先级。如果两个通道有相同的优先级，两个通道将交替传输。

TRDQ 位使能 DMA 控制器由定时器 2 启动传输。如果这一位置 1，DMA 请求就由定时器 2 发起。这种方式可以防止 DMA 在传输时占用所有微处理器的时间。

CHG/$\overline{\text{NOCHG}}$位决定是否可以对 START/$\overline{\text{STOP}}$位进行写。START/$\overline{\text{STOP}}$位用于启动或停止 DMA 传输。要启动 DMA 传输，CHG/$\overline{\text{NOCHG}}$和 START/$\overline{\text{STOP}}$位必须同时为逻辑 1。

BYTE/$\overline{\text{WORD}}$位用来选择是字节传输还是字传输。

存储器到存储器传输的例子

内置 DMA 控制器可以完成存储器到存储器的传输。例 16-4 给出了用于 DMA 控制器编程和启动传输的例程。

例 16-4

```
.MODEL SMALL
.186
.CODE
;
;存储器至存储器用 DMA 传输
;
;MOVES 调用序列:
;
;     DS:DI = 源地址
;     ED:DI = 目的地址
;     CX = 字节数
;
GETA  MACRO SEGA, OFFA, DMAA
      MOV   AX,SEGA          ; 获得段地址
      SHL   AX,4             ; 段地址左移 4 位
      ADD   AX,OFFA          ; 加上偏移地址
      MOV   DX,DMAA          ; 寻址 DMA 控制器
      OUT   DX,AX            ; 编程地址
      PUSHF
      MOV   AX,SEGA
      SHR   AX,12
      POPF
      ADC   AX,0
      ADD   DX,2
      OUT   DX,AX
      ENDM
MOVES PROC  NEAR
      GETA  DS,SI,0FFC0H     ;编程源地址
      GETA  ES,DI,0FFC4H     ;编程目的地址
      MOV   DX,0FFC8H        ;编程计数器
      MOV   AX,CX
      OUT   DX,AX
      MOV   DX,0FFCAH        ;编程 DMA 控制器
      MOV   AX,0B606H
      OUT   DX,AX            ;启动传送
      RET
MOVES ENDP
```

例 16-4 的子程序可以将位于 DS：SI 中的数据传输到 ES：DI 中。传输的字节数存放在 CX 寄存器中。这个操作与 REP MOVSB 类似，但通过使用 DMA 控制器，它的执行速度要快得多。

16.2.6 片选单元

片选单元简化了存储器和 I/O 到 80186/80188 的接口电路。这个单元包括了可编程片选逻辑。在一些中小规模的系统中，外部不需要附加任何译码器就可以选通存储器或 I/O。对于大规模的系统，可能还需要附加一些外部的译码器。片选单元有两种形式：一种是在 XL 和 EA 型号中，另一种在 EB 和 EC 型号中，这两种是不同的。

存储器片选

在小规模或中规模的基于 80186/80188 的系统中，可以有 6 个（XL 和 EA 型号）或 10 个（EB 和 EC 型号）片选引脚用来选择外部的存储器部件。**UCS**（**upper chip select**）引脚可以选通位于高端存储空间的存储器件，这个空间常常由板上的 ROM 占用。这个可编程引脚可以指定 ROM 的地址空间和需要插入的等待状态的个数。注意，ROM 的结束地址为 FFFFFH。**LCS**（**low chip select**）引脚可以选择从 00000H 起始的存储器件（通常为 RAM）。与UCS引脚一样，LCS 的存储空间大小和等待状态的个数都是可编程的。剩余的 4 个或 8 个片选引脚可以选择中间的存储器件。XL 和 EA 型号中的 4 个引脚（MCS3～MCS0）对起始（基）地址和存储器大小都是可编程的。要注意所有器件的容量必须相同。EB 和 EC 型号中的 8 个引脚（GCS7～GCS0）的起始地址和大小也是可编程的，并能表示存储器件或 I/O 器件。

外设片选

80186/80188 用PCS6～PCS0引脚（在 XL 和 EA 型号中）最多可寻址 7 个外设。在 EB 和 EC 型号中，可以用GCS引脚选通最多 8 个存储器件或 I/O 器件。I/O 的基地址可以编程到 1KB 内的端口地址，该端口地址对应的块长度为 128 字节（在 EB 和 EC 型号上为 64 字节）。

XL 和 EA 型号片选单元的编程

存储器和 I/O 各部分等待状态的数目都是可编程的。80186/80188 微处理器有内置的等待状态发生器，可以插入 0～3 个等待状态（XL 和 EA 型号）。表 16-5 列出了选择不同数目的等待状态时每个可编程寄存器中 R_2～R_0 位所要求的逻辑值。这三条线可以选择是否需要外部 READY 信号来产生等待状态。如果选择了 READY，外部的 READY 信号与内部的等待状态发生器将并行工作。例如，如果选择了 READY，且 READY 有 3 个时钟周期是逻辑 0，内部等待状态发生器却被编程为插入 2 个等待状态，此时，将插入 3 个等待状态。

表 16-5 等待状态控制位 R_2、R_1 和 R_0（XL 和 EA 型号）

R_2	R_1	R_0	等待状态数	是否需要 READY 信号
0	×	×	—	需要
1	0	0	0	不需要
1	0	1	1	不需要
1	1	0	2	不需要
1	1	1	3	不需要

假设 64KB 的 EPROM 在存储系统的顶端，正确运行需要 2 个等待状态。要在这部分地址空间上选通这个器件，UCS引脚被编程为存储器 F0000H～FFFFFH，并且有 2 个等待状态。图 16-22 列出了所有存储器和 I/O 片选的控制寄存器，这些控制寄存器在外设控制块（PCB）中，偏移地址为 A_0H～A_9H。注意，这些寄存器最右边的 3 位来自表 16-5。高端存储器区域的控制寄存器位于 PCB 中，偏移地址为 A_0H。这个 16 位的寄存器可以用起始地址（在本例中为 F0000H）和等待状态的个数来编程。还需注意，地址的最高两位必须为 00，只有 A_{17}～A_{10}地址位可以在控制寄存器中编程设置。表 16-6 举例说明了不同存储器大小对应的控制寄存器的值。由于我们的例子中需要两个等待状态，除了最右边 3 位是 *110* 而不是 *100* 外，基本地址与表中 *64KB* 器件对应的值相同。送到高端存储器控制寄存器的数据应该是 3006H。

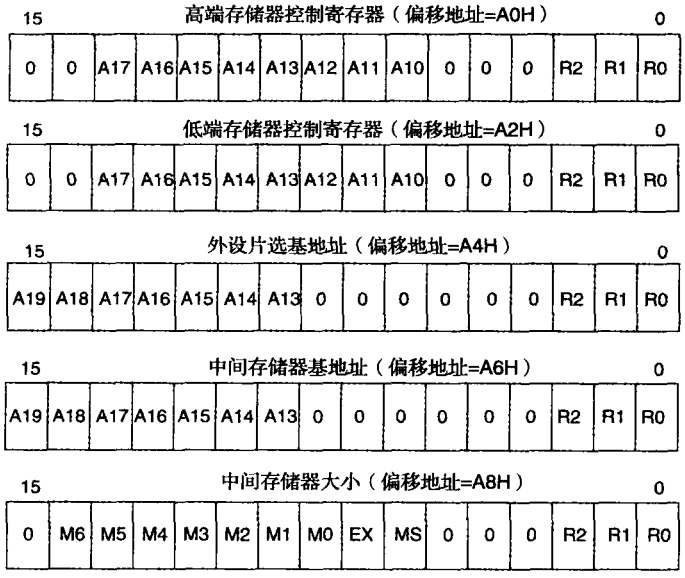


图 16-22 80186/80188（XL 和 EA 型号）中的片选寄存器

表 16-6 高端存储器控制寄存器 A_0H 编程（XL 和 EA 型号）

起始地址	块大小	无等待、无 READY 信号的值
FFC00H	1KB	3FC4H
FF800H	2KB	3F84H
FF000H	4KB	3F04H
FE000H	8KB	3E04H
FC000H	16KB	3C04H
F8000H	32KB	3804H
F0000H	64KB	3004H
E0000H	128KB	1004H
C0000H	256KB	0004H

表 16-7 低端存储器控制寄存器 A_2H 编程（XL 和 EA 型号）

终止地址	块大小	无等待、无 READY 信号的值
003FFH	1KB	0004H
007FFH	2KB	0044H
00FFFH	4KB	00C4H
01FFFH	8KB	01C4H
03FFFH	16KB	03C4H
07FFFH	32KB	07C4H
0FFFFH	64KB	0FC4H
1FFFFH	128KB	1FC4H
3FFFFH	256KB	3FC4H

假如有一块 32KB 的 SRAM 位于内存空间的最低端，既不需要内部插入等待状态，也不需要外部的 READY 输入，可以编程引脚$\overline{\text{LCS}}$来选通这个器件，A_2H 寄存器的加载与 A_0H 寄存器相同。在这个例子中，A_2H 寄存器被送入 07FCH。表 16-7 列出了低端存储器片选所对应的控制寄存器的值。

存储器的中间部分的片选可以用 A_6H 和 A_8H 这两个寄存器编程设置。A_6H 设定中间存储器的片选线$\overline{\text{MCS3}}$～$\overline{\text{MCS0}}$和等待状态数。A_8H 定义存储器块的大小和每个存储芯片的大小（参见表 16-8）。除了块大小，外设的等待状态的数目也可以和其他存储器区域一样编程。EX（第 7 位）和 MS（第 6 位）用来指定外设选择线，我们将简短地讨论一下。

表 16-8 中间存储器控制寄存器 A_8H 编程（XL 和 EA 型号）

块大小	芯片大小	无等待、无 READY 信号且 EX = 0，MS = 1 时的值
8KB	2KB	0144H
16KB	4KB	0344H
32KB	8KB	0744H
64KB	16KB	0F44H
128KB	32KB	1F44H
256KB	64KB	3F44H
512KB	128KB	7F44H

例如，假设有 4 块 32KB 的 SRAM 要加到系统的中间存储区域，起始地址为 80000H，结束地址为

9FFFFH，没有等待状态。要对中间部分存储器选择线进行编程，寄存器 A_6H 的最左边的 7 位装入地址位，使 8～3 位为逻辑 0，最右边的 3 位包含了 READY 控制位。在此例中，寄存器 A_6H 中装入 8004H，寄存器 A_8H 中装入 1F44H，假定 EX = 0 和 MS = 1，并且外设不需要等待状态和 READY 信号。

寄存器 A_4H 与寄存器 A_8H 中的 EX 和 MS 位共同对外设片选引脚（$\overline{PCS6}$～$\overline{PCS0}$）进行编程。寄存器 A_4H 用来保存外设选择线的起始地址或基地址。外设可以位于存储器空间或 I/O 空间。如果位于 I/O空间，端口号的 A_{19}～A_{16}必须为0000。一旦起始地址被编程到任意1KB 的I/O 地址边界，那么相邻的$\overline{PCS}$引脚的所选通地址间隔为 128 字节。

例如，如果寄存器 A_4H 被设置为 0204H，没有等待状态和 READY 同步信号，存储器或 I/O 起始地址为 2000H。在这种情况下，I/O 端口是：$\overline{PCS0}$ = 2000H，$\overline{PCS1}$ = 2080H，$\overline{PCS2}$ = 2100H，$\overline{PCS3}$ = 2180H，$\overline{PCS4}$ = 2200H，$\overline{PCS5}$ = 2280H，$\overline{PCS6}$ = 2300H。

寄存器 A_8H 的 MS 位用来确定外设片选引脚是作为存储器空间的片选还是作为 I/O 空间的片选。如果 MS 位是逻辑 0，引脚$\overline{PCS}$将按照存储器地址译码；如果是逻辑 1，则引脚$\overline{PCS}$将按照 I/O 地址译码。

EX 位用来选择引脚$\overline{PCS5}$和$\overline{PCS6}$的功能。如果 EX = 1，这两个引脚可作为 I/O 设备的片选；如果 EX = 0，这两个引脚为系统提供被锁存的地址线 A_1 和 A_2。有些 I/O 设备利用 A_1 和 A_2 来选择内部寄存器，为此要提供地址线 A_1 和 A_2。

EB 和 EC 型号片选单元的编程

前面已经提到，EB 和 EC 型号有不同的片选单元。这两个较新型的 80186/80188 和早期的型号一样有高端和低端片选引脚，但没有中间存储器片选和外设片选引脚。在 EB 和 EC 型号中有 8 个通用片选引脚（$\overline{GCS7}$～$\overline{GCS0}$），它们取代了中间存储器片选和外设片选引脚。通用片选引脚既可用来选通存储器，又可用来选通 I/O 设备。

由于这些片选引脚各自都包含一个起始地址寄存器和一个终止地址寄存器，所以 EB 和 EC 型号的片选单元的编程与其他型号不同。每个片选引脚的偏移地址及起始和结束寄存器的内容参见图 16-23。

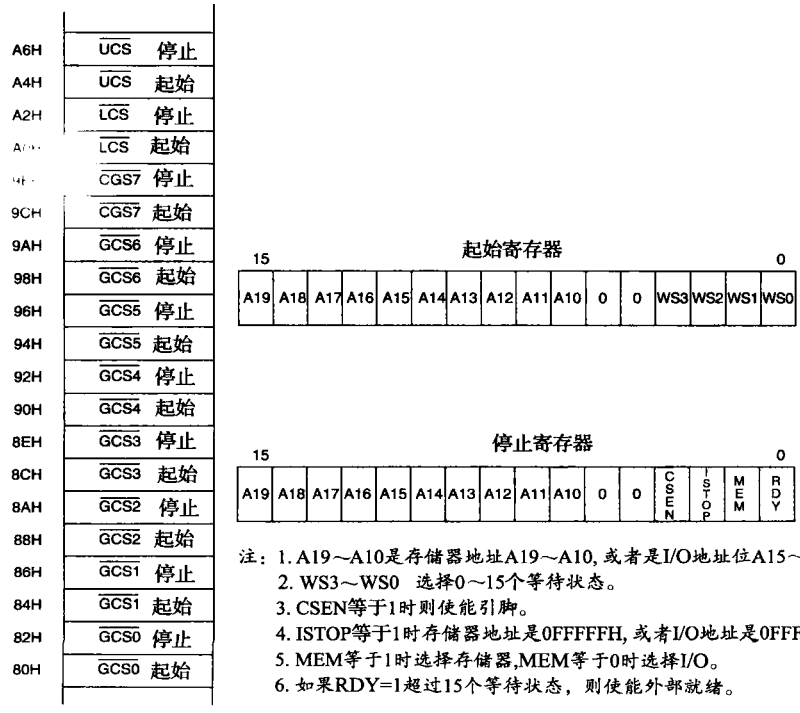


图 16-23 80186/80188（EB 和 EC 型号）中的片选单元

EB 和 EC 型号的编程比早期的 XL 和 EA 型号要简单。例如，要把$\overline{\text{UCS}}$引脚编程到起始地址为 F0000H，结束地址为 FFFFFH 的范围内（64KB）。起始地址寄存器（偏移地址 = A_4H）被设置为 F002H，表示起始地址为 F0000H 并且有两个等待状态；结束地址寄存器（偏移地址 = A_6H）被设置为 000EH，表示结束地址是 FFFFFH 并且没有外部 READY 同步的存储器空间的选择。其他片选引脚以相同方式来编程。

16.3 80C188EB 接口举例

由于 80186/80188 微处理器被设计成嵌入式控制器，所以本节给出一个这样应用的例子。这个例子说明了简单的存储器和 I/O 如何连接到 80C188EB 微处理器上，同时给出了系统复位后对 80C188EB 和内部寄存器编程所需要的软件。图 16-24 所示为 80C188EB 型 80188 的引脚输出。请注意，这个型号与前面提到的 XL 型号有所不同。

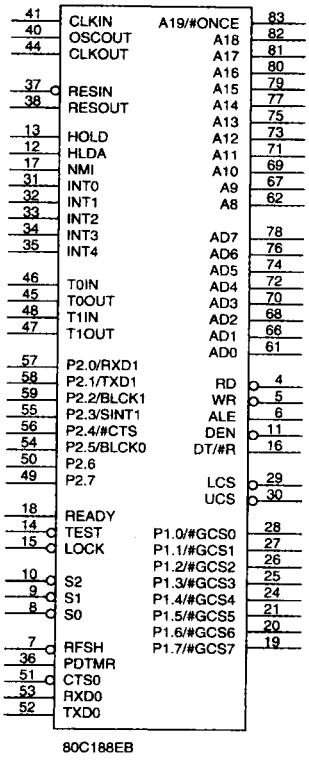


图 16-24 80188EB 型的 80188 微处理器的引脚图

80C188EB 中包含了一些早期型号所没有的新特性，这些新特性包括两个与其他功能共享的 I/O 端口 P_1 和 P_2 以及两个内置的串行通信接口。这个型号没有类似于 XL 型号中的 DMA 控制器。

80C188EB 可以和一个被设计作为微处理器教练板（trainer）的小系统接口。本文举例说明的教练板用了 1 片 27256 EPROM 存放程序，1 片 62256 SRAM 存放数据，1 片 8255 与键盘和 LCD 显示接口。图 16-25 所示的是一个小型的基于 80C188EB 微处理器的微处理器教练板。

存储器的片选由片选信号来完成，其中 EPROM 27256 的片选用$\overline{\text{UCS}}$引脚，SRAM 62256 的片选选用$\overline{\text{LCS}}$引脚。8255 的片选用$\overline{\text{GCS0}}$。系统软件的 EPROM 地址空间为 F8000H ～ FFFFFH；SRAM 的地址为 00000H ～ 07FFFH；8255 的 I/O 端口在 0000H ～ 003FH（软件用端口 0、1、2 和 3）。和一般情况下一样，在这个系统中，我们没有修改外设控制块（PCB）的地址，它的地址位于 FF00H ～ FFFFH。

例 16-5 是 80C188EB 微处理器初始化所需的程序。这个例子对 80C188EB 和整个系统作了完整的编程。程序将在本章的下一节详述。

例 16-5

```
; 简单的80188EB实时操作测试系统
;
; INT 40H 延迟 BL 毫秒（范围：1~99）
; INT 41H 延迟 BL 秒（范围：1~59）
; 注意，延迟时间必须写成十六进制
; 例如，15 毫秒是 15H
; INT 42H在 LCD 上显示字符串
;         ES:BX 寻址以 NULL（空）结束的串
;         AL = 显示哪里（80H 为第一行，C0H为第二行）
; INT 43H  清除 LCD
; INT 44H  从键盘读键；AL = 键码

.MODEL TINY
.186                    ;转到 80186/80188 指令集
.CODE
.STARTUP

;80188EB 微处理器教练板程序
;使用 MASM 6.11
;命令行 = ML /AT FILENAME.ASM
```

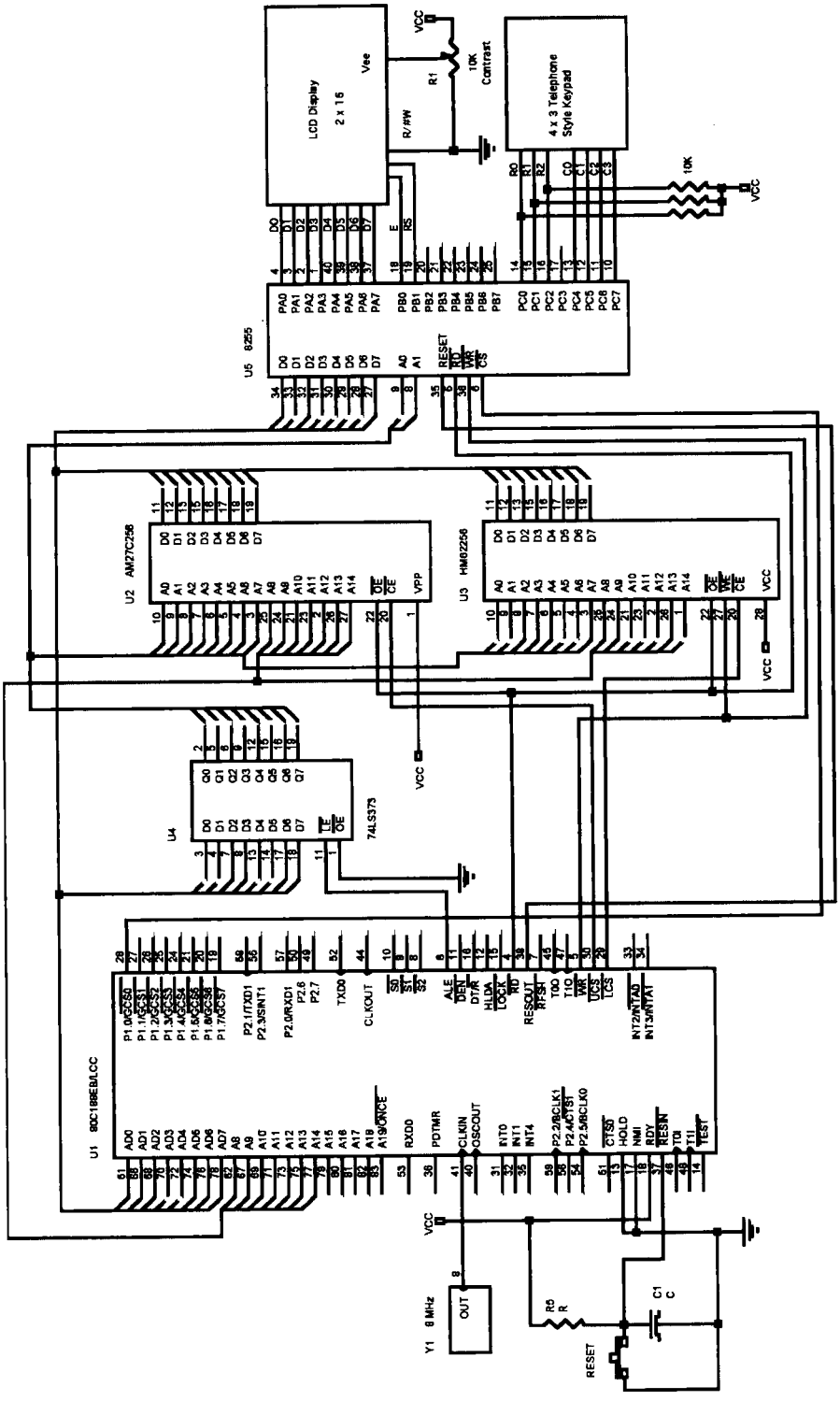


图16-25 一个基于80C188EB的简单的微处理器系统

```
;;;
```

```
        OUT     1,AL                    ;使 LCD 的 E 信号处于 0
        STI                             ;允许中断
        CALL    INIT                    ;初始化 LCD

;;;
```

```
        .ENDIF
        IRET

TIM2    ENDP

MODU    DB      0                           ;模 100
        DB      10H                         ;模 10
        DB      60H                         ;模 60
        DB      60H                         ;模 60
        DB      24H                         ;模 24

DISPLAY PROC NEAR       ;显示一天的时间（一次一秒）

        .IF     BYTE PTR DS:[40FH] != 0             ;如果显示时间不是0
                STI                                 ;开中断
                MOV     BX,3F0H
                MOV     SI,40EH                     ;寻址时钟
                MOV     AL,[SI]                     ;取小时数
                .IF AL > 12H                        ;AM / PM
                        SUB         AL,12H
                        DAS
                .ELSEIF AL == 0
                        MOV         AL,12H
                .ENDIF
                CALL    STORE
                MOV     BYTE PTR [BX],':'
                INC     BX
                MOV     AL,[SI]                     ;取分钟数
                CALL    STORE
                MOV     BYTE PTR [BX],':'
                INC     BX
                MOV     AL,[SI]                     ;取秒数
                CALL    STORE
                MOV     DL,'A'                      ;AM / PM
                .IF     BYTE PTR DS:[40EH] > 11H
                        MOV       DL,'P'
                .ENDIF
                MOV     BYTE PTR [BX],' '
                MOV     [BX+1],DL
                MOV     BYTE PTR [BX+2],'M'
                MOV     BYTE PTR [BX+3],0           ;字符串结束
                MOV     BX,3F0H                     ;显示缓冲区
                MOV     AL,0C2H                     ;启动 LCD
                INT     42H
        .ENDIF
        RET

DISPLAY ENDP

STORE   PROC    NEAR

        PUSH    AX
        SHR     AL,4
        MOV     DL,AL
        POP     AX
        AND     AL,15
        MOV     DH,AL
        ADD     DX,3030H
        MOV     [BX],DX
        ADD     BX,2
        DEC     SI
        RET

STORE   ENDP

DELAYM  PROC    FAR USES DS BX AX

        STI                         ;开中断
        MOV     AX,0
        MOV     DS,AX
```

```
        MOV     AL,DS:[40AH]        ;获取毫秒计数器
        ADD     AL,BL               ;BL =毫秒数
        DAA
        .REPEAT
        .UNTIL AL == DS:[40AH]
        IRET

DELAYM  ENDP

DELAYS  PROC    FAR USES DS BX AX

        STI                         ;开中断
        MOV      AX,0
        MOV      DS,AX
        MOV      AL,DS:[40CH]       ;获取秒数
        ADD      AL,BL
        DAA
        .IF AL >= 60H
                SUB       AL,60H
                DAS
        .ENDIF
        .REPEAT
        .UNTIL AL == DS:[40CH]
        IRET

DELAYS  ENDP

INIT    PROC    NEAR

        MOV      BL,30H             ;等待 30ms
        INT      40H
        MOV      CX,4
        .REPEAT
                SEND       38H,0,6
        .UNTILCXZ
        SEND     8,0,2
        SEND     1,0,2
        SEND     12,0,2
        SEND     6,0,2
        RET

INIT_LCD ENDP

STRING  PROC    FAR USES BX AX    ;显示字符串

        STI                                     ;开中断
        SEND     AL,0,1                         ;发送启动位
        .REPEAT
                SEND      BYTE PTR ES:[BX],2,1
                INC       BX
        .UNTIL BYTE PTR ES:[BX] == 0
        IRET

STRING  ENDP

CLEAR   PROC    FAR USES AX BX    ;清 LCD

        STI                                     ;开中断
        SEND     1,0,2
        IRET
CLEAR   ENDP

KEY     PROC     FAR USES BX          ;读键值

        STI
        MOV  AL,0           ;清 C0～C3
        OUT  2,AL
        .REPEAT             ;等待按键释放
                .REPEAT
                        BUT
                .UNTIL ZERO?
```

```
                MOV   BL,12H            ;时间延迟
                INT   40H
                BUT
          .UNTIL ZERO?
          .REPEAT                 ;等待键按F
                .REPEAT
                                  BUT
                .UNTIL    !ZERO?
                MOV   BL,   12H         ;时间延迟
                INT   40H
                BUT
          .UNTIL !ZERO?
         MOV  BX,0FDEFH
         .REPEAT
                MOV   AL,BL
                OUT   2,AL
                ADD   BH,3
                ROL   BL,1
                BUT
        .UNTIL !ZERO?
         .WHILE 1
                SHR   AL,1
                .BREAK .IF !CARRY?
                INC   BH
         .ENDW
         MOV  BX,OFFSET LOOK-100H
         XLAT CS:LOOK       ;指定代码段  (EPROM)

         IRET

KEY      ENDP

LOOK     DB       3,2,1
         DB       6,5,4
         DB       9,8,7
         DB       10,0,11

;;;
```

的系统任务，提供系统调用或函数，组成嵌入式系统。数据和程序部分拥有操作系统要用的所有的程序和静态数据。

复位部分

例 16-5 中的软件的最后部分给出了 RTOS 的复位块。ORG 语句将 reset 指令放到存储器倒数 16 个字节处。既然这样，EEROM 是 32K 就意味着从 0000H 开始到 7FFFH 结束。记得 32K 的器件有 15 个地址引脚。$\overline{CS}$输入将系统 F8000H 到 FFFFFH 单元选择给 EPROM。程序中的 ORG 语句将复位部分的起点放在 80F0H，因为在所有 TINY 模式（.COM）中，即使程序的第一个字节是文件中存储的第一个字节程序都要从偏移地址 100H 开始汇编。由于偏差，EPROM 上的所有地址需要像 ORG 语句用 100H 调整。

由于系统中的复位单元是 FFFF0H，只有 16 字节的存储器存放复位指令。在这个例子中，仅有在跳到 EPROM 起始位置前将$\overline{UCS}$起始地址编程为 F8000H 的地址。TINY 模式下不允许长跳转，因此强行存储长跳转实际的十六进制操作码（EAH）。

初始化部分

例 16-5 的初始化部分开始于复位块，在 EPROM 开始的地方继续。如果查看初始化部分，可以看到系统中所有的可编程设备都被编程并且段寄存器也被加载。初始化部分也对定时器 2 进行编程，每毫秒给 TIM_2 程序产生一次中断。TIM_2 中断服务程序每秒更新一次时钟，这也是精确的程序延时的基础。

内核

因为系统不完整且仅作为一个试验系统，因此例 16-5 中的内核代码非常短。在这个例子中，所有要做的事情就是显示一个签到消息并在 LCD 的第二行显示日时间。一旦完成，系统就在 WHILE 语句上无限循环。所有系统程序如果没有损坏都应该是无限循环的。

16.4.2 实例系统

图 16-27 举例说明了一个简单的基于 80C188EB 嵌入式微处理器的嵌入式系统。该原理图只描绘了添加到图 16-25 用于从 LM-70 读温度的部分。该系统包含了 2 行 × 16 字符/行的 LCD 显示器，可以显示时间和温度。系统本身存放在 32K × 8 的 EPROM 中。含有 32K × 8 的 SRAM，充当堆栈以存储时间。数据库保存了最近的温度以及采集温度时对应的时间。

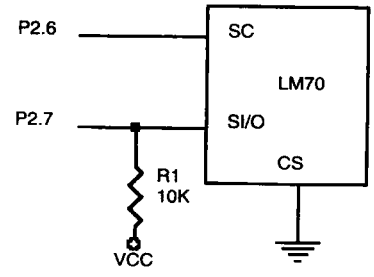


图 16-27 图 16-25 用于读取温度时的附加电路

温度传感器位于 LM_{70} 数字温度传感器中，该传感器由美国国家半导体公司生产，售价不到 1 美元。以串行格式与微处理器的接口，转换器的分辨率是 10 位，包括了 1 位符号位。图 16-28 列出了 LM_{70} 温度传感器的引脚。

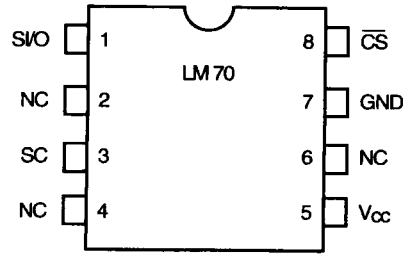


图 16-28 LM_{70} 温度传感器

LM_{70} 通过 SIO 引脚到微处理器或从微处理器传送数据，该引脚是双向串行数据引脚。信息通过 SIO 引脚由 SC（时钟）引脚同步。LM_{70} 有三个 16 位寄存器：配置寄存器、温度传感器寄存器和标识寄存器。配置寄存器选择关闭模式（XXFFH）或连续转换模式（XX00）。温度寄存器在 16 位数据字的最左边 11 位中包含了有符号的温度。如果温度是负的，它就是补码形式。标识寄存器读的时候为 8100。

从 LM_{70} 读温度时，以摄氏度读的，每个步长为 0.25℃。例如，如果温度寄存器是 0000 1100 100X XXXX 即十进制数的 100，则表示的温度为 25℃。

例 16-6 举例说明了添加到例 16-5 所列的操作系统的软件。系统每分钟采样一次温度，并将温度连同日期、时、分计时存储到循环队列中。日期是当系统初始化后从 0 开始的数。队列大小为 16KB，可

以存储最近的4096个测量值。这个例子没有用键盘，但是用到了一些系统调用，在显示器第一行显示温度。通过实时时钟来确定每分钟的开始，进行采样。列出的程序取代例16-5中“；System software placed here”那部分代码。这个软件替换了例子中的WHILE无限循环。

LM_{70}通过给它输出16位0初始化，然后就可以读16位温度。LM_{70}的读由软件通过TEMP过程完成，初始化通过INITT过程完成。

例16-6

```
;系统软件每分钟采样一次温度,
;并将温度存储到位于0500H~44FFH的循环队列中

        MOV   WORD PTR DS:[4FCH],500H     ;进队列指针 = 500H
        MOV   WORD PTR DS:[4FEH].500H     ;出队列指针 = 500H
        CALL  INITT           ;初始化 LM70
        .while 1
              .IF DS:[40CH] == 0 && DS:[40BH] == 0 && DS:[40AH] == 0
                    CALL  TEMP            ;每分钟一次
                    CALL  ENQUE           ;温度进队列
                    CALL  DTEMP           ;显示温度
              .ENDIF
        .endw

INITT   PROC    NEAR           ;送 0000H 到 LM70 ,复位它

        IO   0FF58H,003FH              ;p2.6和p2.7 设置为输出
        MOV  CX,16                     ;位计数为16
        MOV  DX,0FF5EH                 ;寻址端口2锁存器
        .REPEAT
                    MOV  AL,40H
                    OUT  DX,AL
                    MOV  AL,0
                    OUT  DX,AL
        .UNTILCXZ
        RET

INITT   ENDP

TEMP    PROC    NEAR           ;读温度

        IO   0FF58K,00BFH              ;p2.7 为输入
        MOV  CX,16
        MOV  BX,0
        .REPEAT                        ;读取16位
              IO  0FF5EH,0C0H
              MOV DX,0FF5AH
              IN  AL,DX                ;读1位
              SHR AL,1
              RCR BX,1                 ;放进 BX
              IO  0FF5EH,40H
        .UNTILCXZ
        MOV  AX,BX
        SAR  AX,6                      ;转换为整数温度值
        RET

TEMP    EMDP

ENQUE   PROC     NEAR USES AX          ;温度进队，不检查是否满

        MOV  BX,DS:[4FCH]
        MOV  [BX],AX                   ;保存温度
        ADD  BX,2
        MOV  AX,DS:[40DH]              ;取时间 HH:MM
        MOV  [BX],AX
        ADD  BX,2
        .IF BX == 4500H
              MOV BX,500H
```

```
        .ENDIF
        RET

ENQUE   ENDP

DTEMP   PROC

        MOV  BX,410H          ;寻址字符串缓存区
        OR   AX,AX

        .IF SIGN?                    ;如果为负
             MOV  BYTE PTR [BX],'-'
             NEG  AX
             INC  BX
        .ENDIF
        AAM                          ;转换为 BCD码
        .IF AH != 0
            ADD  AH,30H
            MOV  [BX],AH
            INC  BX
        .ENDIF
        ADD  AL,30H
        MOV  [BX],AL
        INC  BX
        MOV  BYTE PTR [BX],'°'
        MOV  BYTE PTR [BX+1],'C'
        MOV  AX,DS
        MOV  ES,AX
        MOV  AL,86H
        MOV  BX,410H
        INT  42H                     ;在第一行上显示温度

DTEMP   ENDP
```

16.4.3 线程系统

有时候需要实现可以处理多线程的操作系统。操作系统内核用实时钟中断处理多线程。在小的RTOS 中，一种进程调度的方法就是在不同任务间用时间片切换。基本时间片可以是任意宽度，在某些程度上依赖于微处理器的执行速度。例如，在 100MHz 时钟的系统中，现代微处理器上的许多指令都在 1 个或 2 个时钟内执行结束，假设机器每两个时钟执行一条指令，如果选择 1ms 的时间片，那么每个时间片内机器可以执行 50 000 条指令，对于大部分系统足够了。如果使用了较低的时钟频率，那么时间片就选 10ms 甚至 100ms。

每个时间片由定时器中断激活。中断服务程序查看队列中是否有要执行的任务，如果有，它将启动执行新任务；如果没有出现新任务，它将继续执行旧任务或者进入到空闲状态等待新任务。队列是循环的，可以包含很多系统任务直至上限。例如，只有 10 个入口的小系统可能是一个小队列。队列的大小由预期的整个系统的需求来确定，可以很大或很小。

每个调度队列入口必须包含进程的指针（CS：IP）和机器的整个上下文状态。调度队列内容还可以包括其他一些形式的入口，例如万一发生死锁时的解除死锁时间入口、优先级入口和延长时间片激活时间入口。在下面的例子没有用到优先级入口和允许程序延长连续时间片数量的入口。进程进入队列时，内核将按照线性原则或者循环法严格地维护进程。

要实现嵌入式系统的调度器，需要实现一些过程或宏，用它们启动新的应用程序，当应用程序完成时终止它，如果访问 I/O 需要时间则暂停该应用程序。每个宏将在一个有效地址上，如 0500H，访问位于存储系统内的调度队列。调度队列将使用例 16-7 中的数据结构，使得创建队列相当容易，它有 10 个入口的空间。该调度队列允许每次启动最多 10 个进程。

例 16-7

```
PRESENT    DB    0          ;0 = 不存在
DUMMY1     DB    ?
RAX        DW    ?
RBX        DW    ?
RCX        DW    ?
RDX        DW    ?
RSP        DW    ?
RBP        DW    ?
RSI        DW    ?
RDI        DW    ?
RFLAG      DW    ?
RIP        DW    ?
RCS        DW    ?
RDS        DW    ?
RES        DW    ?
RSS        DW    ?
DUMMY2     DW    ?          ;记录的大小为32字节
```

在系统初始化期间，例 16-7 所示的数据结构在内存中复制了 10 次，完成了队列的结构，此时，在初始化，它包含没有激活的进程。队列指针初始化为 500H。在这个例子中，队列指针存放在 4FEH 单元。例 16-8 提供了一种可能的初始化。在 RAM 里 500H 开始的地方存储了 10 个数据结构的备份。该软件假定 32MHz 的系统时钟操作定时器 2，定时器 2 作为预先引入比例因子，把 4MHz 时钟输入（系统时钟除 8）除以 40 000，这使得定时器 2 的输出为 1kHz（1.0ms）。定时器 1 被编程为定时器 2 的时钟信号除以 10，即每 10ms 产生一次中断。

例 16-8

```
;初始化线程队列

        PUSH  DS
        MOV   AX,0
        MOV   DS,AX
        MOV   SI,500H
        MOV   BX,OFFSET PRESENT-100H     ;清空所有队列入口
        MOV   CX,10
        .REPEAT
          MOV   BYTE PTR DS:[SI],0
          ADD   SI,32
        .UNTILCXZ
        MOV   DS:[4FEH],500H             ;设置队列指针
        POP   DS
        MOV   DX,0FF42H                  ;Timer 2 CMPA = 40000
        MOV   AX,40000
        OUT   DX,AL
        MOV   DX,0FF32H                  ;Timer 1 CMPA = 10
        MOV   AX,10
        OUT   DX,AL
        MOV   AX,0                       ;清除定时器计数寄存器
        MOV   DX,0FF30H
        OUT   DX,AL
        MOV   DX,0FF40H
        OUT   DX,AL
        MOV   DX,0FF46H
        MOV   AX,0C001H                  ;启动定时器 2
        OUT   DX,AL
        MOV   DX,0FF36H                  ;启动定时器 1
        MOV   AX,0E009H
        OUT   DX,AL
```

NEW 过程（例 16-9 中放在 INT 60H）增加一个进程到队列中，在 10 个记录中搜索，直到找到第一个字节（PRESENT）为 0 的，这表示该入口为空。如果找到空的入口，它将进程的起始地址放到 RCS 和 RIP 中，并将 0200H 放到 RFLAG 单元。RFLAG 中 200H 确保当该进程启动时中断是允许的，以

防止系统崩溃。如果 10 个进程都已经调度了，那么 NEW 过程将一直等待到某个进程结束。每个进程也被分配了 256 字节的堆栈空间，从偏移地址 7600H 开始分配的，因此最低进程的堆栈空间为 7500H～75FFH，接着的堆栈空间为 7600H～76FFH 等。堆栈区分配可以由存储管理算法实现。

例 16-9

```
INT60   PROC    FAR USES DS AX DX SI

        MOV   AX,0
        MOV   DS,AX                     ;寻址 0000 段
        STI                             ;开中断

        .REPEAT                         ;在这里插入正好
              MOV    SI,500H
              HLT                       ;与RTC 中断同步
              .WHILE BYTE PTR DS:[SI] != 0 && SI != 660H
                     ADD   SI,32
              .ENDW
        .UNTIL BYTE PTR DS:[SI] == 0

        MOV   BYTE PTR DS:[SI],0FFH ;激活进程
        MOV   WORD PTR DS:[SI+18],200H      ;标志
        MOV   DS:[SI+20],BX             ;保存 IP
        MOV   DS:[SI+22],DX             ;保存 CS
        MOV   DS:[SI+28],SS             ;保存 SS
        MOV   AX,SI
        AND   AX,3FFH
        SHL   AX,3
        ADD   AX,7500H
        MOV   DS:[SI+10],AX             ;保存 SP
        IRET

INT60   ENDP
```

最后的控制程序（KILL）位于中断向量 61H，如例 16-10 所示，要终止应用程序，通过将队列数据结构中的 PRESENT 置 00H，将其从调度队列中删除来实现。

例 16-10

```
INT61   PROC   FAR USES DS AX DX SI

        MOV  AX,0
        MOV  DS,AX
        MOV  SI,DS:[4FEH]           ;获取队列指针
        MOV  BYTE PTR DS:[SI],0     ;结束线程
        JMP  INT12A

INT61   ENDP
```

PAUSE 过程只调用了时间片过程（INT 12H），它跳出进程并将控制返回给时间片过程，及早地结束了进程的时间片。这种早跳出使得其他进程在返回到当前进程之前继续。

在例 16-11 中给出使用 10ms 时间片的 80C188EB 的时间片中断服务程序。由于这是中断服务程序，要注意使它尽可能高效率。例 16-11 说明了位于中断向量 12H 的时间片程序，该中断向量对应 80C188EB 微处理器中的定时器 1。虽然没有说明，本软件假定定时器 2 用作预先引入比例因子，定时器 1 使用来自定时器 2 的信号以产生 10ms 中断。同时软件假定系统中没有使用其他中断。

例 16-11

```
INT12  PROC   FAR USES DS AX DX SI

       MOV   AX,0
       MOV   DS,AX              ;寻址0000段
       MOV   SI,DS:[4FEH]       ;获取队列指针
       CALL  SAVES              ;保存状态
```

```
INT12A:                          ;从结束线程来
       ADD   SI,32               ;获得下一个进程
       .IF SI == 660H            ;使队列循环
             MOV  SI,500H
       .ENDIF
       .WHILE BYTE PTR DS:[SI] != 0FFH    ;找下一个进程
              ADD  SI,32
              .IF SI == 660H
                    MOV  SI,500H
              .ENDIF
       .ENDW
       MOV   DX,0FF30H           ;获得完整的10ms时间片
       MOV   AX,0
       OUT   DX,AL
       MOV   DX,0FF02H           ;清除中断
       MOV   AX,8000H
       OUT   DX,AX
       JMP   LOADS               ;加载下一个进程

INT12  ENDP
```

系统启动（置于系统初始化之后）必须是内部为无限循环等待的单进程，如例 16-12 所示。

例 16-12

```
SYSTEM_STARTUP:

;产生一个 WAITS 线程

      MOV   BX,OFFSET WAITS-100H    ;线程的偏移地址
      MOV   DX,0F800H               ;线程的段地址
      INT   60H                     ;启动等待线程
      STI

;其他进程可以在这里启动

WAITS:                           ;系统空闲循环
      .WHILE 1
             INT  12H            ;放弃
      .ENDW
```

最后，用 SAVES 和 LOADS 过程在从一个进程切换到另外一个进程时加载和保存机器的上下文。这些过程由时间片中断（INT 12H）和启动过程（INT 60H）调用，例 16-13 列出了这些过程。

例 16-13

```
SAVES   PROC   NEAR

        MOV  DS:[SI+4],BX          ;保存 BX
        MOV  DS:[SI+6],CX          ;保存 CX
        MOV  DS:[SI+10],SP         ;保存 SP
        MOV  DS:[SI+12],BP         ;保存 BP
        MOV  DS:[SI+16],DI         ;保存 DI
        MOV  DS:[SI+26],ES         ;保存 ES
        MOV  DS:[SI+28],SS         ;保存 SS
        MOV  BP,SP                 ;获得 SP
        MOV  AX,[SP+2]             ;获得 SI
        MOV  DS:[SI+14],AX         ;保存 SI
        MOV  AX,[BP+4]             ;获得 DX
        MOV  DS:[SI+8],AX          ;保存 DX
        MOV  AX,[BP+6]             ;获得 AX
        MOV  DS:[SI+2],AX          ;保存 AX
        MOV  AX,[BP+8]             ;获得 DS
        MOV  DS:[SI+24],AX         ;保存 DS
        MOV  AX,[BP+10]            ;获得 flags
        MOV  DS:[SI+18],AX         ;保存 flags
        MOV  AX,[BP+12]            ;获得 CS
        MOV  DS:[SI+22],AX         ;保存 CS
```

```
        MOV  AX,[BP+14]              ;获得 IP
        MOV  DS:[SI+20],AX           ;保存  IP
        RET

SAVES   ENDP

LOADS   PROC   FAR

        MOV  SS,DS:[SI+28]           ;获得 SS
        MOV  SP,DS:[SI+10]           ;获得 SP
        PUSH WORD PTR DS:[SI+20]     ;PUSH IP
        PUSH WORD PTR DS:[SI+22]     ;PUSH CS
        PUSH WORD PTR DS:[SI+18]     ;PUSH Flags
        PUSH WORD PTR DS:[SI+24]     ;PUSH DS
        PUSH WORD PTR DS:[SI+2]      ;PUSH AX
        PUSH WORD PTR DS:[SI+8]      ;PUSH DX
        PUSH WORD PTR DS:[SI+14]     ;PUSH SI
        MOV  BX,DS:[SI+4]            ;获得 BX
        MOV  CX,DS:[SI+6]            ;获得 CX
        MOV  BP,DS:[SI+12]           ;获得 BP
        MOV  DI,DS:[SI+16]           ;获得 DI
        MOV  ES,DS:[SI+26]           ;获得 ES
        POP  SI
        POP  DX
        POP  AX
        POP  DS
        IRET

LOADS   ENDP
```

16.5 80286 简介

80286 微处理器是 8086 微处理器的高级型号，是为多用户和多任务环境设计的。80286 通过它的存储管理系统可寻址 16MB 的物理存储器和 1GB 的虚拟存储器。本节将介绍 80286 微处理器，它曾用在早期计算机市场上一度流行的 AT 型的 PC 机上。80286 基本上是一个优化了指令执行周期的 8086。由于 80286 含有存储管理单元，所以它又是一个增强的 8086。现在，80286 在 PC 机中已经找不到了，但在一些控制系统中还在作为嵌入式控制器应用。

16.5.1 硬件特性

图 16-29 是80286 微处理器内部结构框图。注意，与 80186/80188 不同，80286 没有内置外围设备，它增加了一个存储管理单元（MMU），在结构框图中称为**寻址单元（address unit）**。

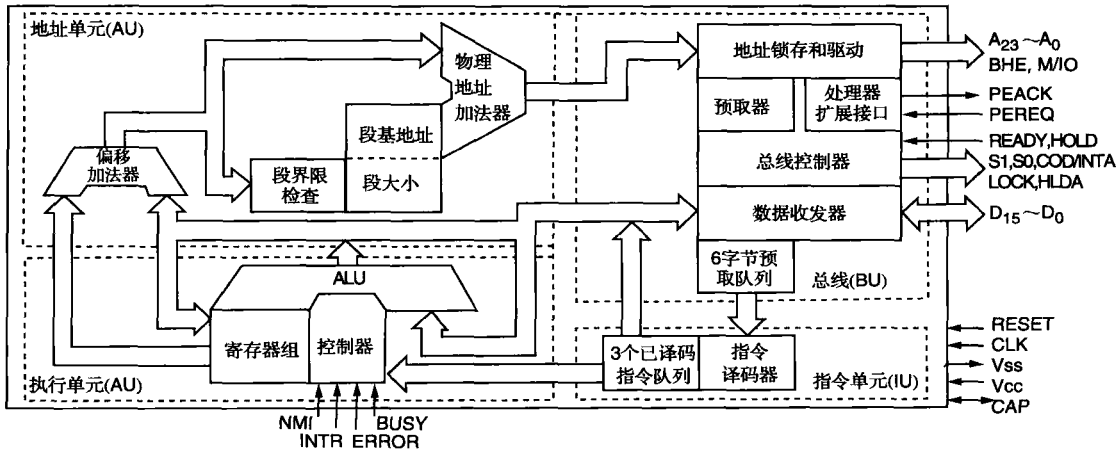


图 16-29 80286 微处理器的结构框图（由 Intel 公司提供）

仔细研究框图可以发现，地址引脚 A_{23}～A_0、$\overline{BUSY}$、CAP、$\overline{ERROR}$、$\overline{PEREQ}$和$\overline{PEACK}$是8086微处理器上所没有的新增加的引脚。引脚$\overline{BUSY}$、$\overline{ERROR}$、$\overline{PEREQ}$和$\overline{PEACK}$上的信号用于微处理器的扩展或协处理器，80287就是协处理器的一个例子（注意，TEST引脚现在用来表示$\overline{BUSY}$引脚）。地址总线是24位，对应16MB的物理存储器。CAP引脚接一个0.047μF，±20%的电容，该电容作为12V的滤波器接地。为了便于比较，图16-30列出了8086和80286的引脚。注意，80286没有多路复用地址/数据总线。

正如在第1章中提到的，80286可以在实模式和保护模式两种模式下运行。在实模式下，80286可以寻址1MB的存储空间，实际上与8086相同；在保护模式下，80286可以寻址16MB的存储空间。

图16-31说明了基于80286微处理器的基本系统。时钟由82284时钟发生器（类似于8284A）提供，系统控制信号由82288系统总线控制器（类似于8288）提供。此外要注意没有8086地址/数据总线复用的锁存电路。

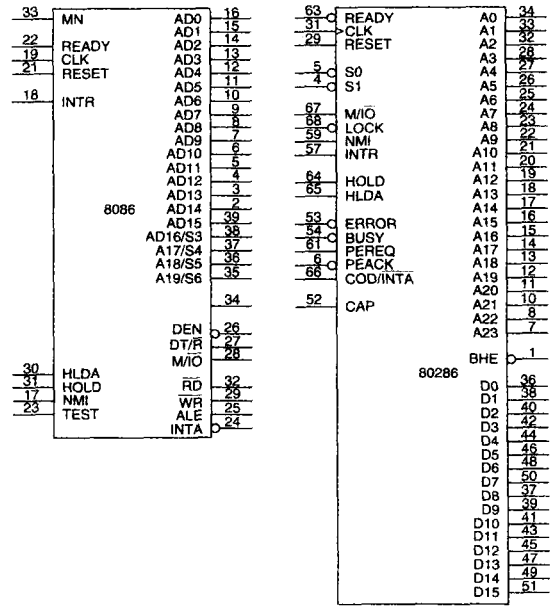


图16-30 8086和80286微处理器的引脚

注：80286没有多路复用地址/数据总线。

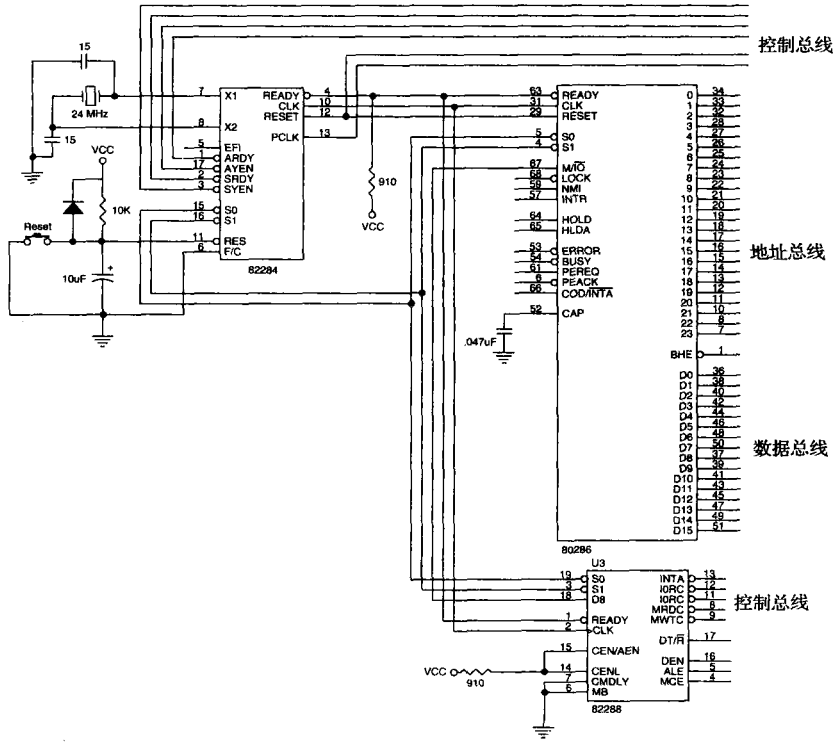


图16-31 80286微处理器、82284时钟发生器和82288系统总线控制器的互连

16.5.2 新增指令

80286 比早期的 8086、80186/80188 有了更多的指令。这些新加的指令可以通过 80286 的存储管理器来控制虚拟存储系统。表 16-9 列出了 80286 新加的指令及对每条指令用途的注释。这些指令仅是 80286 新增加的指令。注意，80286 也包含了 80186/80188 新添加的指令，如 INS、OUTS、BOUND、ENTER、LEAVE、PUSHA、POPA、立即乘、立即移位和循环计数。

表 16-9 新增加的 80286 指令

指　令	用　途
CLTS	清除任务切换标志位
LDGT	装载全局描述符表寄存器
SGDT	保存全局描述符表寄存器
LIDT	装载中断描述符表寄存器
SIDT	保存中断描述符表寄存器
LLDT	装载局部描述符表寄存器
SLDT	保存局部描述符表寄存器
LMSW	装载机器状态字
SMSW	保存机器状态字
LAR	装载访问权限
LSL	装载段界限
SAR	保存访问权限
ARPL	调整请求优先级
VERR	检验是否可读
VERW	检验是否可写

以下是对存储管理一节中没有解释的指令的描述。这些指令是一些特殊指令，只能在特定的条件下使用。

CLTS **清除任务切换标志（clear task-switched flag）**指令。该指令用来将 TS（任务切换）标志位清 0。如果 TS 标志位为逻辑 1 并且 80287 算术协处理器被任务使用，就会产生中断（其向量类型为 7）。这就使协处理器的功能可以用软件仿真。因为 CLTS 指令只能在保护模式下优先级为 0 时执行，CLTS 指令用在系统级并被认为是一个特权指令。没有设置 TS 位的指令，可以用 LMSW 指令向程序状态字（MSW）的第 3 位（TS）写逻辑 1 来设置 TS 位。

LAR **装载访问权（load access right）**指令。该指令用来读取段描述符并将访问权限字节的副本放到一个 16 位的寄存器中。例如，LAR AX，BX 指令就是把 BX 值作为选择子，读取对应的描述符，并将描述符中的访问权限字节装载到 AX 中。由于该指令可以取得访问权限，因此可以用来在程序使用描述符所描述的存储段之前检查访问权限。

LSL **装载段界限（load segment limit）**指令。该指令把段界限值装载到用户指定的寄存器中。例如，LSL AX，BX 指令可以把段界限装载到 AX，该段是由 BX 中的选择子所选择的描述符来描述的。该指令可用来测试段的界限。

ARPL **调整请求优先级（adjust requested privilege level）**指令。该指令用于检测选择子，以确保所请求的选择子的优先级没有冲突。例如 ARPL AX，CX 指令，AX 中含有请求的优先级，CX 中含有用来访问描述符的选择子的值。如果所请求的优先级低于正在检测的描述符的优先级，则零标志被置位。这要求程序调整优先级或指示优先级冲突。

VERR **读访问检验（verify for read access）**指令。该指令用来检验段是否可读。回顾第 1 章，数据段是可以读保护的。如果数据段可以被读，则零标志位就被置位。

VERW **写访问检验（verify for write access）**指令。该指令用来检验段是否可写。回顾第 1 章，数据段是可以写保护的。如果数据段可以被写，则零标志位就被置位。

16.5.3 虚拟存储机

虚拟存储机（virtual memory machine）是一种将大的存储空间（在 80286 中为 1GB）映射到比它小得多的物理存储空间（在 80286 中为 16MB）的机构，这样就可以在较小的物理存储系统上运行非常大的系统。这是通过在固定磁盘存储系统和物理存储器之间的数据和程序交换来实现的。80286 微处理器通过描述符可以寻址 1GB 存储空间。每个 80286 描述符描述了一个 64KB 的存储段，80286 最多可以有 16K 个描述符，因此可以为系统描述最多 1GB（64KB×16K）的存储空间。

如第 1 章所述，在保护模式下描述符描述存储器的段地址。80286 有代码段描述符、数据段描述符、堆栈段描述符、中断描述符、过程描述符以及任务描述符。在保护模式下描述符的存取是通过在段寄存器中装入一个选择子来实现的。有关描述符及其应用的更多细节在第 1 章以及第 17 章、第 18 章和第 19 章中给出。要详细了解保护模式下存储管理系统，请参考这些章节。

16.6 小结

1）除去一些新增加的指令，80186/80188 微处理器和 8086/8088 微处理器有相同的基本指令系统。因此 80186/80188 是 8086/8088 的增强型号。新增加的指令有 PUSHA、POPA、INS、OUTS、BOUND、ENTER、LEAVE、立即乘和立即移位/循环计数。

2）80186/80188 在硬件上增加了时钟发生器、可编程中断控制器、三个可编程定时器、可编程 DMA 控制器、可编程片选逻辑单元、看门狗定时器、动态 RAM 刷新逻辑电路以及 80186/80188 各种不同型号所附加的功能。

3）时钟发生器使得 80186/80188 可以由外部 TTL 电平时钟源或者连接在 X_1（CLK_{IN}）和 X_2（OSC_{OUT}）引脚上的晶体提供时钟运行。晶体的振荡频率是微处理器运行频率的两倍。80186/80188 可以在 6～20MHz 的速度下运行。

4）可编程中断控制器仲裁所有的内部和外部中断请求。它还可以与两个外部 8259A 中断控制器一起使用。

5）80186/80188 内部有三个可编程定时器。每个定时器都是一个完全可编程的 16 位计数器，用于产生波形和事件计数。定时器 0 和定时器 1 有外部输入和输出引脚。定时器 2 由系统时钟提供定时信号，并用来为其他两个定时器提供时钟信号或发出 DMA 请求。

6）可编程 DMA 控制器是一个完全可编程的双通道控制器。DMA 可以在存储器和 I/O 之间、I/O 和 I/O 之间或存储器和存储器之间进行传送。DMA 请求可以由软件、硬件或定时器 2 的输出来产生。

7）可编程片选单元是一个内部译码器，它最多提供 13 个输出引脚来选择存储器（有 6 个引脚）和 I/O（有 7 个引脚）。它还可以插入 0～3 个等待状态，可以有或没有外部 READY 同步信号。在 EB 和 EC 型号中有 10 个片选引脚，等待状态的数目可以被编程为 0～15 个。

8）80186/80188 的时序和 8086/8088 的时序的惟一区别在于 80186/80188 中的 ALE 信号比 8086/8088 中提前半个时钟出现。否则，二者的时序完全相同。

9）6MHz 型号的 80186/80188 允许 417ns 的存储器访问时间，8MHz 型号为 309ns。

10）内部的 80186/80188 外设要通过外设控制块（PCB）编程，PCB 被初始化在 I/O 端口 FF00H～FFFFH。通过改变初始 I/O 地址为 FFFEH 和 FFFFH 的重定位寄存器的值，可以将 PCB 重定位到任意的存储器空间或 I/O 空间。

11）80286 是一个包含了存储管理单元（MMU）的增强型 8086。由于有 MMU，80286 可以寻址 16MB 的物理存储空间。

12）除了少数用于控制存储管理单元的附加指令外，80286 包含与 80186/80188 相同的指令系统。

13）通过存储管理单元，80286 微处理器可以寻址由存储在两个描述符表中的 16K 个描述符所确定的 1GB 虚拟存储空间。

16.7 习题

1. 列出 8086/8088 与 80186/80188 微处理器之间的区别。
2. 80186/80188 在硬件上有哪些 8086/8088 中所没有的改进？
3. 80186/80188 有哪些封装方式？
4. 如果在 X_1 和 X_2 引脚上连接一个 20MHz 的晶体，那么 CLK_{OUT} 引脚上的信号的频率是多少？
5. 描述 80188 嵌入式控制器的 80C188XL 和 80C188EB 两种型号之间的区别。
6. 对于逻辑 0，80186/80188 引脚的扇出是_____。
7. 80186/80188 总线周期中包含了几个时钟周期？
8. 8086/8088 和 80186/80188 在时序上的主要区别是什么？
9. 存储器存取时间有何重要性？
10. 以 6MHz 时钟运行的 80186/80188 允许多长的存储器存取时间？
11. 80186/80188 复位后，外设控制块位于什么地址？
12. 编写一个程序，将外设控制块映射到存储器地址 10000H～100FFH 上。
13. 80186/80188 微处理器中 INT_0 引脚使用哪个中断向量？
14. 80186/80188 微处理器中的中断控制器有多少中断向量可用？
15. 中断控制器有哪两种工作模式？
16. 中断控制寄存器有何用处？
17. 当一个中断源被屏蔽时，中断屏蔽寄存器中对应的屏蔽位为逻辑_____。
18. 中断轮询寄存器和中断轮询状态寄存器之间有何区别？
19. 中断结束（EOI）寄存器有何用途？
20. 80186/80188 中有多少个 16 位定时器？
21. 哪个定时器有输入和输出引脚连接？
22. 哪个定时器连接到系统时钟上？
23. 如果定时器中用到了两个最大值比较寄存器，解释该定时器是如何工作的？
24. 定时器控制寄存器中的 INH 位有何用途？

25. 定时器控制寄存器中的 P 位有何用途?
26. 定时器控制寄存器中的 ALT 位为定时器 0 和定时器 1 选择哪种工作类型?
27. 解释定时器输出引脚如何使用。
28. 编一个程序使定时器 1 产生一个连续周期性信号，该信号在前 123 次计数中保持为逻辑 1，而后在 23 次计数过程中保持为逻辑 0。
29. 编一个程序使定时器 0 在它的输入引脚上出现 105 个时钟脉冲时输出一个单脉冲。
30. 80C186XL 中的 DMA 控制器控制着多少个 DMA 通道?
31. DMA 控制器的源地址寄存器和目标地址寄存器均为_____位宽。
32. 如何用软件启动 DMA 通道?
33. 片选单元（XL 和 EA 型号中）有_____个引脚用于选择存储器芯片。
34. 片选单元（XL 和 EA 型号中）有_____个引脚用于选择外围设备。
35. 引脚$\overline{\text{UCS}}$选通的高端存储器块的结束地址是____。
36. 中间存储器片选引脚（XL 和 EA 型号中）可以编程_____地址和模块大小。
37. 引脚$\overline{\text{LCS}}$选通的低端存储器块的起始地址是_____。
38. 内部等待状态发生器（EB 和 EC 型号）可以插入 0 到_____个等待状态。
39. 对寄存器 A_8H（XL 和 EA 型号）进行编程，使中间存储器块的大小为 128KB。
40. 寄存器 A_8H 中的 EX 位有何用途?
41. 编写一个程序，要求对引脚$\overline{\text{GCS3}}$编程，使其选择 20000H～2FFFFH 的存储器单元并插入 2 个等待状态。
42. 编写一个程序，要求对引$\overline{\text{GCS4}}$脚编程，使其选择 1000H～103FH 的 I/O 设备端口并插入 1 个等待状态。
43. 80286 微处理器可以寻址_____字节的物理存储器。
44. 如果使用存储管理，80286 可以寻址_____字节的虚拟存储器。
45. 80286 和_____的指令系统相同，存储管理指令除外。
46. VERR 指令有何用途?
47. LSL 指令有何用途?
48. RTOS 是什么?
49. RTOS 是如何处理多线程的?
50. 在 Internet 搜索至少两个不同的 RTOS，写一个短的比较报告。

第 17 章　80386 和 80486 微处理器

引言

80386 微处理器是较早期的 16 位微处理器 8086/80286 的一个完全 32 位的版本，代表了体系结构的重要进步——从 16 位体系结构过渡到 32 位体系结构。在增大字长的同时，还增加了许多增强功能和附加特征。80386 微处理器主要特征是支持多任务、存储管理、虚拟存储（分页或非分页）、软件保护、大的存储空间。80386 向上兼容早期为 8086/8088 和 80286 写的所有软件。80386 所支持的寻址空间也从 8086 的 1MB 和 80286 的 16MB 增加到 4GB。80386 不需要复位就可以实现保护模式和实模式之间的相互切换。这种切换在早期的 80286 上是一个问题，因为它需要硬件复位。

80486 微处理器是 80386 微处理器的增强型号，它的许多指令可以在一个时钟周期内完成。80486 含有 8KB 的 cache 存储器和改进型的 80387 算术协处理器（注意，80486DX4 含有 16KB 的 cache 存储器）。当 80486 工作在与 80386 相同的工作频率时，它的执行速度比 80386 大约提高 50%。在第 18 章中，我们将看到 Pentium 和 Pentium Pro 微处理器，它们都有 16KB 的 cache 存储器，它们的执行速度比 80486 微处理器快两倍以上，并且 Pentium 和 Pentium Pro 微处理器都含有比 80486 算术协处理器快 5 倍的改进的算术协处理器。第 19 章将介绍 Pentium Ⅱ 到 Pentium 4 微处理器的改进之处。

目的

读者学习完本章后将能够：

1）对比 80386、80486 微处理器与早期的 Intel 微处理器有何不同。
2）描述 80386 和 80486 微处理器的存储管理单元和分页单元的操作。
3）切换保护模式与实模式。
4）定义 80386/80486 附加指令和寻址方式的操作。
5）解释 cache 存储系统的操作。
6）详述 80386/80486 的中断结构和直接内存访问（DMA）的结构。
7）对比 80486 与 80386 微处理器。
8）解释 80486 cache 的操作。

17.1　80386 微处理器简介

在将 80386 或其他微处理器用在某系统之前，首先应该掌握该微处理器每个引脚的功能。本节详述了 80386 每个引脚的功能，同时还详述了外部存储系统和 I/O 结构。

图 17-1 展示了 80386DX 微处理器的引脚。80386DX 采用的是 132 引脚的 PGA（引脚栅格阵列）封装。80386 通常有两个型号，一个是 80386DX，本章将对它进行图解和说明；另一个是 80386SX，它是一个缩减总线型的 80386。80386 还有一个新的型号——80386EX，它包括了 AT 总线系统、动态 RAM 控制器、可编程片

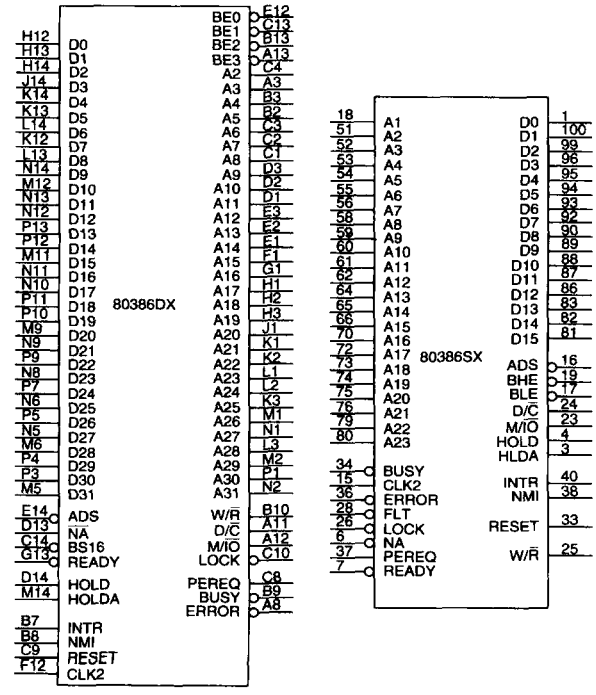


图 17-1　80386DX 和 80386SX 微处理器的引脚图

选逻辑、26 位地址引脚、16 位数据引脚及 24 个 I/O 引脚。图 17-2 为 80386EX 嵌入式 PC。

80386DX 通过 32 位的数据总线和 32 位的地址总线可以寻址 4GB 的存储空间。80386SX 更像 80286，它通过 24 位地址总线和 16 位的数据总线可寻址 16MB 的存储空间。80386SX 是在 80386DX 之后专门为那些不需要完整的 32 位总线的应用开发的。在许多早期使用与 80286 相同设计的主板的 PC 中可以找到 80386SX。在 80386SX 流行时，许多应用，包括 Windows3.11，对内存的要求小于 16MB，所以 80386SX 是一个非常流行和廉价的 80386 微处理器。即使可以很便宜地将系统升级为 80486，80386 仍然在许多应用中使用。例如，80386EX 似乎不在计算机系统中露面，但在嵌入式应用中非常流行。

正如 Intel 系列微处理器中其他的早期型号，运行 80386 需要 +5.0V 单电源。80386 的电源电流在 25MHz 的型号中平均为 550mA，在 20MHz 型号中平均为 500mA，在 16MHz 型号中平均为 450mA。还有一种 33MHz 的型号，需要 600mA 的电源电流。80386EX 工作在 33MHz 时，它的电源电流为 320mA。注意，在某些正常操作期间，电源电流可以超过 1.0A，这就意味着电源和功率的分布网络必须能够提供这些大电流。这个器件有多个 V_{CC} 和 V_{SS} 接点，为了正确运行，它们必须都连接到 +5.0V 和地上。有些引脚被标注为 N/C（无连接），这些引脚绝不能连接。80386SX 和 80386EX 的增强型号可以用 +3.3V 电源。这些型号常常用于便携式笔记本或膝上型计算机中，它们通常为表面贴装器件。

每个 80386 输出引脚能够输出 4.0mA（地址和数据接点）或 5.0mA（其他接点）的电流。与早期 8086、8088 和 80286 输出引脚的 2.0mA 相比，80386 的驱动电流有所提高。80386EX 的大部分输出引脚的输出电流都为 8.0mA。每个输入引脚代表了一个只需 ±10μA 电流的小负载。在某些系统中（不包括最小系统），这些电流需要总线缓冲。

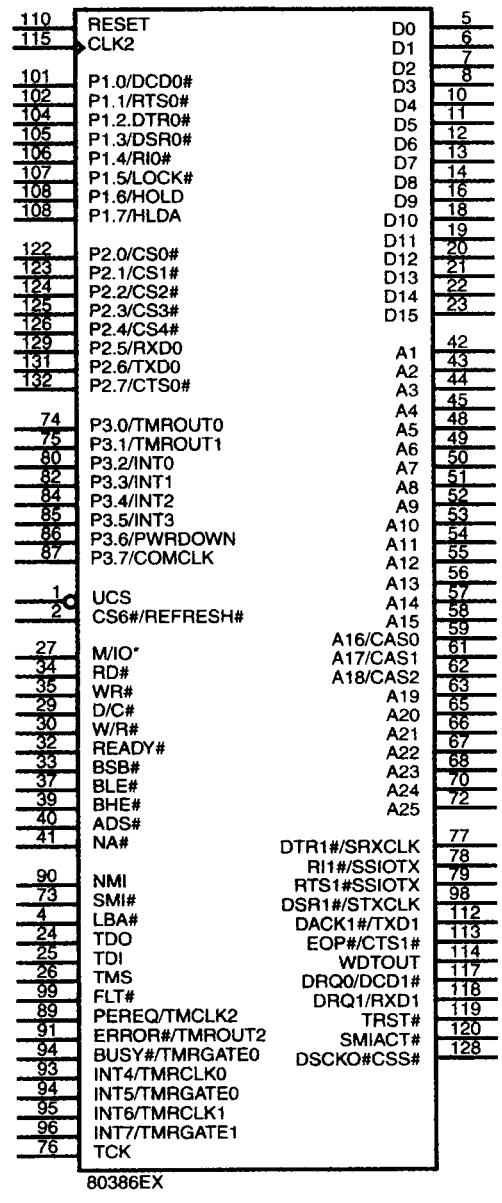


图 17-2 80386EX 嵌入式 PC

80386DX 引脚的功能如下：

A_{31}～A_2 **地址总线接点（address bus connection）**。用来寻址 80386 存储系统中 1G×32（4GB）存储器单元。注意，A_0 和 A_1 被编码为总线使能信号（$\overline{BE3}$～$\overline{BE0}$），以便选择 32 位存储单元中 4 个字节的任何一个或几个。还要注意，因为 80386SX 用 16 位数据总线取代了 80386DX 的 32 位数据总线，所以 80386SX 上只有 A_1，同时体选择信号由$\overline{BHE}$和$\overline{BLE}$来代替。$\overline{BHE}$信号选通数据总线的高 8 位；$\overline{BLE}$信号选通数据总线的低 8 位。

D_{31}～D_0 **数据总线接点（data bus connection）**。用于微处理器和存储器及 I/O 之间传送数据。注意，80386SX 只有 D_{15}～D_0。

$\overline{BE3}$～$\overline{BE0}$ **体使能信号（bank enable signal）**。选择字节、字或双字的数据。这些信号由微处理器内部根据地址线 A_0 和 A_1 产生。在80386SX中这些引脚被$\overline{BHE}$、$\overline{BLE}$和 A_1 取代。

M/$\overline{IO}$ **存储器/IO（memory/IO）**引脚。它为逻辑1时选择存储器设备，为逻辑0时选择I/O设备。在I/O操作期间，地址总线包含了16位的地址，在 A_{15}～A_2 地址连接线上。

W/$\overline{R}$ **写/读（write/read）**引脚。该信号为逻辑1时表明当前总线周期是写周期，为逻辑0时则表明当前总线周期是读周期。

$\overline{ADS}$ **地址数据选通（address data strobe）**引脚。当80386发出有效的存储器地址或I/O地址时，该信号激活。该信号与W/$\overline{R}$信号组合产生早期的基于8086～80286微处理器的系统中所出现的独立的读和写信号。

RESET **复位（reset）**引脚。该信号用来初始化80386。使其从内存FFFFFFF0H处开始执行软件，80386被复位为实模式，最左边的12位地址线保持逻辑1（FFFH），直至执行一个远程调用或远跳转。这使它与早期微处理器兼容。

CLK_2 **2倍频时钟（clock times 2）**引脚。它由两倍于80386工作频率的时钟信号驱动。例如，要使80386以16MHz频率工作，该引脚上应加上32MHz时钟。

$\overline{READY}$ **就绪（read）**引脚。该引脚控制为延长存储器访问时间而插入的等待状态的个数。

$\overline{LOCK}$ 总线锁定（lock）引脚。当指令带有LOCK前缀时，该引脚变为逻辑0，这经常用于DMA访问期间。

D/$\overline{C}$ **数据/控制（data/control）**输出引脚。为逻辑1表示数据总线含有来自或到存储器或I/O的数据。为逻辑0则表明微处理器正在执行中断响应或已停机。

$\overline{BS16}$ **16位总线（bus size 16）**输入引脚。该引脚用来选择32位（$\overline{BS16}$=1）或16位（$\overline{BS16}$=0）数据总线。在大多数情况下，如果80386DX工作在16位数据总线方式时，我们通常选用80386SX，它含有16位数据总线。在80386EX中用$\overline{BS8}$引脚选择8位总线。

$\overline{NA}$ **下条地址（next address）**，该信号使80386在当前总线周期输出下一条指令或数据的地址。这个引脚通常用在流水线操作中。

HOLD 总线保持（hold）。该信号用来请求DMA操作。

HLDA 总线保持响应（hold response），该信号用来表明80386当前处于总线保持状态。

$\overline{PEREQ}$ **协处理器请求（coprocessor request）**信号。该信号要求80386放弃总线控制并直接连到80387算术协处理器上。

$\overline{BUSY}$ **忙（busy）**输入信号。由WAIT或FWAIT指令使用，等待协处理器变为可用。可以直接将80387和80386连接。

$\overline{ERROR}$ **错误（error）**信号。向微处理器指明协处理器检测到错误。

INTR **中断请求（interrupt request）**信号。外部电路用该信号来请求中断。

NMI **非屏蔽中断（non-maskable interrupt）**输入引脚。请求非屏蔽中断，和早期微处理器的作用一样。

17.1.1 存储系统

80386DX的物理存储系统的空间为4GB并按此空间进行寻址。如果采用虚拟寻址，存储管理单元和描述符可以将64TB的虚拟空间映射到4GB的物理空间（注意，只要找到一种与大容量硬盘交换的方法，虚拟寻址就允许程序大于4GB）。图17-3显示了80386DX的物理存储系统的组织结构。

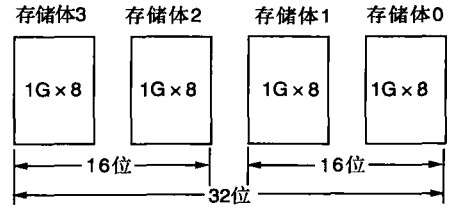


图17-3 80386微处理器的存储系统

存储器分成了四个8位宽的存储体，每个存储体最多包含1GB的存储容量。这种32位宽的存储器组织结构允许以字节、字或双字为单位直接存取存储器中的数据。80386DX可以在一个存储周期中传送一个32位的数

字，而完成相同的功能，早期的8088需要四个存储周期，80286和80386SX需要两个存储周期。如今，数据宽度已非常重要，尤其对32位的单精度浮点数来说。高级软件一般用浮点数来存储数据，所以高级软件在对存储单元存取时可以直接利用这种32位宽度优势来提高执行速度。

如Intel家族中以前的型号一样，存储器的每个字节以十六进制编号。不同的是80386DX使用32位的存储地址，存储器按字节编号为00000000H～FFFFFFFH。

8086、80286和80386SX系统中的两个存储体可以利用$\overline{BLE}$（在8086和80286中为A_0）和$\overline{BHE}$来分别访问。在80386DX中，存储体通过四个体选择信号$\overline{BE3}$～$\overline{BE0}$来访问。这样可以使微处理器在激活一个体选择信号时只对一个字节进行存取，在激活两个体选择信号时寻址一个字。在大多数情况下，一个字被寻址到体0和体1或者被寻址到体2和体3。存储单元00000000H在体0中，00000001H在体1中，00000002H在体2中，00000003H在体3中。80386DX没有地址线A_1和A_0，因为它们被编码为体选择信号。同样，在80386SX中也没有地址A_0引脚，因为它被编码到$\overline{BLE}$和$\overline{BHE}$信号中。在$\overline{BS8}$ = 1时，80386EX使用16位存储系统，可以在两个存储体中任意访问数据；当$\overline{BS8}$ = 0时，使用8位存储系统，只在一个存储体中访问数据。

缓冲系统

图17-4显示连接了缓冲器的80386DX，这些缓冲器提高了80386DX的地址、数据及控制连接的扇出。该处理器工作频率为25MHz，由一个50MHz的集成振荡器模块提供时钟输入。在现代基于微处理器的设备中，通常用振荡器模块来提供时钟。在使用DMA（直接存储器访问）的系统中，用HLDA作为所有缓冲器的使能信号。否则，在没有DMA的系统中，缓冲器的使能引脚可以直接接地。

流水线和高速缓冲存储器（cache）

cache存储器是一个缓冲存储器，它能够使80386在DRAM速度较低的情况下高效运行。**流水线（pipeline）**是一种特殊的处理存储访问的方法，它使得存储器有额外的时间来存取数据。16MHz的80386所允许的存储器存取时间为50ns或更短。显然，目前几乎没有能够以这个速度访问的DRAM，实际上目前所使用的最快的DRAM的访问周期为40ns或更长。这就意味着，必须找到某种与这种比微处理器所要求的访问速度低的存储芯片进行接口的技术。可采用的技术有三种：交叉存取、高速缓冲和流水线技术。

因为80386微处理器支持以流水线方式进行存储器访问，流水线技术是与存储器接口的推荐方式。流水线技术使得存储器有额外的时钟周期来存取数据，在以16MHz时钟运行的80386中，这个额外的时钟周期将存取时间由50ns延长为81ns。所谓的**管道（pipe）**是由微处理器准备的。当从存储器中取出一条指令时，在下一条指令取出之前，微处理器有额外的时间，在这段时间里，下条指令的地址预先从地址总线上送出。这段额外时间（一个时钟周期）使得可以对较慢的存储器部件有附加的访问时间。

并非所有存储器访问都可以利用流水线，也就是说，有些存储器周期不是流水线方式的。如果正常的流水线存储器周期不需要等待状态，那么这些非流水线存储器周期则要有等待状态。总之，流水线是一个节省费用的功能部件，用在低速系统中可减少访问存储系统所需的时间。

并非所有的系统都能利用流水线，这些系统典型地运行在20MHz、25MHz或33MHz下。在这些高速系统中，必须使用另外的技术来提高存储系统的速度。对那些需要多次访问的数据，**cache**存储系统改善了存储系统的整体性能。注意，80486包含一个称为**一级cache（level 1 cache）**的内部cache，而80386只拥有称为**二级cache（level 2 cache）**的外部cache。

cache是一个高速的存储器（SRAM）系统。它位于微处理器和DRAM存储系统之间。cache存储器通常为存取时间小于10ns的静态RAM器件。在许多情况下，我们见到的二级cache存储系统的大小为32KB到1MB之间。cache存储器的大小更多地是由应用程序而不是由微处理器决定。如果程序较小并很少访问存储器数据，建议使用小的cache存储器。如果程序很大并要访问大块的内存数据，建议

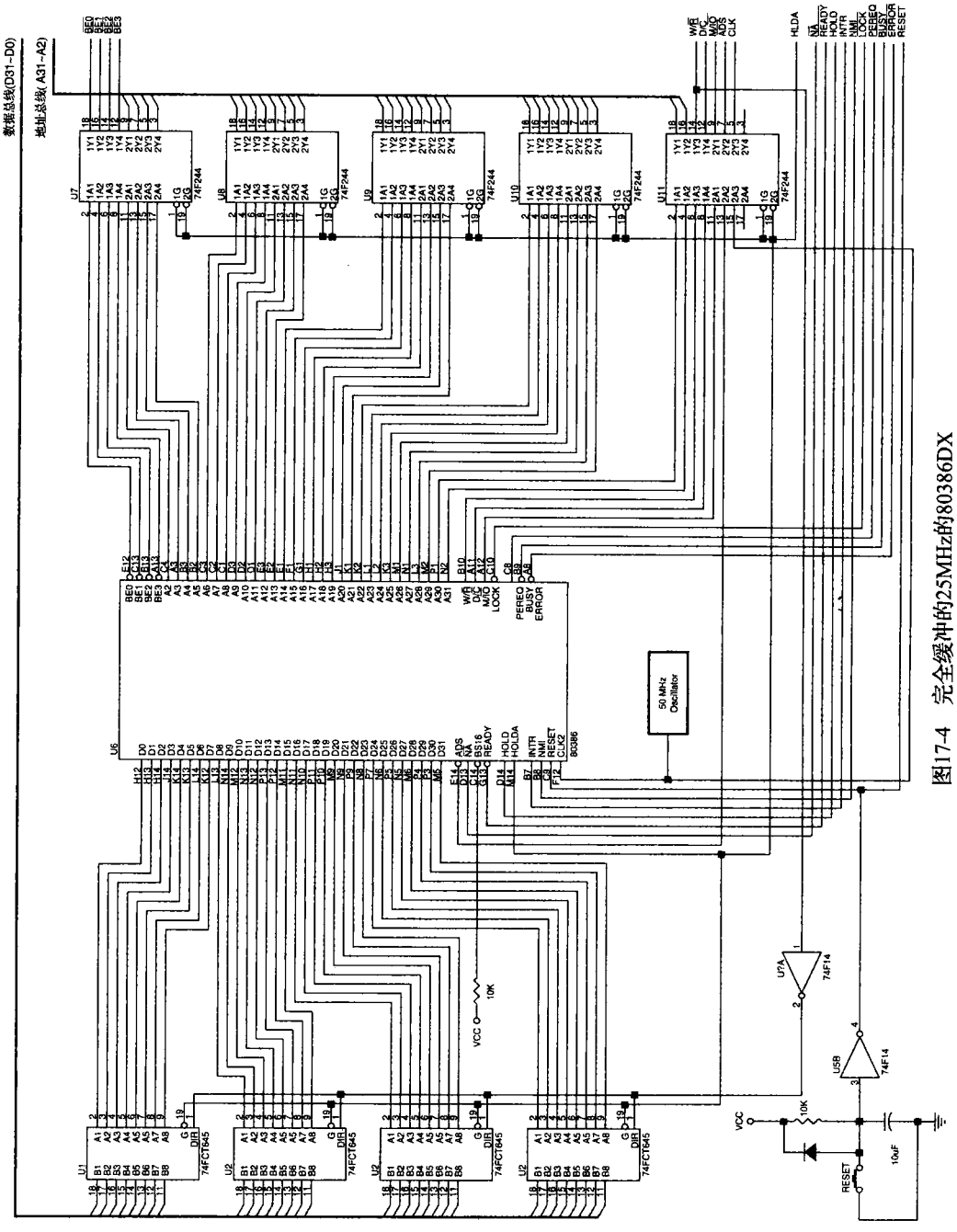


图17-4 完全缓冲的25MHz的80386DX

使用尽可能大的 cache 存储器。在许多情况下，64KB 的 cache 存储器能充分提高执行速度。但最佳状态通常是采用 256KB 的 cache 存储器。我们会发现，在含有 80386 微处理器的系统中，当 cache 存储器大小远远超过 256KB 时，执行速度并没有比 256KB 时提高多少。

交叉存储系统

交叉存储系统（interleaved memory system）是提高系统速度的另一种方法。它的惟一缺点是由于结构决定了需使用相当多的存储器。有些系统采用了交叉存储系统，因而存储器存取时间无需插入等待状态便可以加长。在有些系统中，交叉存储器仍然需要等待状态，但可以减少它们的数目。交叉存储系统需要两组或更多组完整的地址总线和一个为每组地址总线提供地址的控制器。使用两组完整总线的系统，称为**二路交叉（two-way interleave）**系统；使用四组完整总线的系统，称为**四路交叉（four-way interleave）**系统。

交叉存储器分成两个或四个部分。例如，如果交叉存储系统是为 80386SX 微处理器设计的，一部分包含 16 位存储单元地址 000000H ～ 000001H、000004H ～ 000005H 等，而另一部分包含地址 000002H ～ 000003H、000006H ～ 000007H 等。当微处理器访问 000000H ～ 000001H 单元时，交叉控制逻辑产生 000002H ～ 000003H 单元的地址选通信号。当微处理器处理完 000000H ～ 000001H 单元的字时，就可以对 000002H ～ 000003H 单元进行选择和访问了。这个过程切换了访问的存储器部分，这样提高了存储系统的性能。

因为在微处理器访问存储单元之前，地址就已经产生并用来选择存储器了，因此交叉存储延长了提供给存储器的存取时间。这是因为微处理器以流水线方式处理存储器地址，在从上一个地址中读入数据之前发送下一个存储地址。

交叉存储技术存在一个问题（尽管不是主要的），即被访问的存储器的地址在每个存储区交替寻址。而程序的执行并不总是如此。一般程序在执行过程中，平均大约 93% 的时间内，微处理器交替访问存储器区，而在剩余 7% 的时间内，微处理器在同一存储区中访问数据，这意味着由于存取时间的缩短有 7% 的存储器访问必须插入等待状态。存取时间的缩短是由于存储器在得到其地址之前必须等待前一个数据被传送完毕。这给存储器留下较少的访问时间，因此在同一个存储体中访问时需要插入等待状态。

微处理器地址引脚上的地址时序图参见图 17-5。这个时序图显示了在当前数据被存取之前下一个地址是如何输出的。图中还显示了如何通过使用交叉存储器寻址每个存储区来增加访问时间，并与需要一个等待状态的非交叉存储的访问进行了比较。

图 17-6 给出了一个交叉控制器。很显然，这是一个非常复杂的逻辑电路，需对它做些解释。首先，如果 SEL 输入（用于选定当前存储体）为无效（逻辑 0），则$\overline{\text{WAIT}}$信号为逻辑 1。而且，用于选通存储区地址的 ALE0 和 ALE1 都为逻辑 1，使得连接到它们的锁存器变为透明。

一旦 SEL 输入变为逻辑 1，这个电路就开始工作。A1 输入用于确定哪个锁存器（U2B 还是 U5A）变为逻辑 0，来选择一个存储区。同样，变为逻辑 1 的 ALE 也与其前一状态进行比较。如果对同一个存储区第二次访问，$\overline{\text{WAIT}}$信号变为逻辑 0，请求插入等待状态。

图 17-7 展示了一个使用图 17-6 所示电路的交叉存储系统。请注意，ALE_0 和 ALE_1 信号是如何被用来锁定两个存储区中的地址的。每个存储体都是 16 位宽。如果需访问内存中的 8 位数据，则在大多数情况下，系统会产生等待状态。随着程序的执行，80386SX 一般从顺序的存储单元中取指，每次读取 16 位。大多数情况下程序的执行使用交叉存储方式。如果一个系统将要在大多数情况下访问 8 位数据，交叉存储技术是否会减少等待状态数值得怀疑。

采用 16MHz 系统时钟时，图 17-7 所示的交叉存储系统允许的存取时间从 69ns 增加到 112ns（如果插入一个等待状态，则在 16MHz 系统时钟下存取时间为 136ns。这意味着交叉存储系统和有一个等待状态的系统速率大约相同）。如果系统时钟增加到 20MHz，交叉存储系统的存取时间为 89.6ns，而标准的非交叉存储系统只为 48ns。在这种较高时钟频率下，如果存储地址交叉，则 80ns 的 DRAM 能够正确地工作，而不需要等待状态。如果访问同一区域，则需插入一个等待状态。

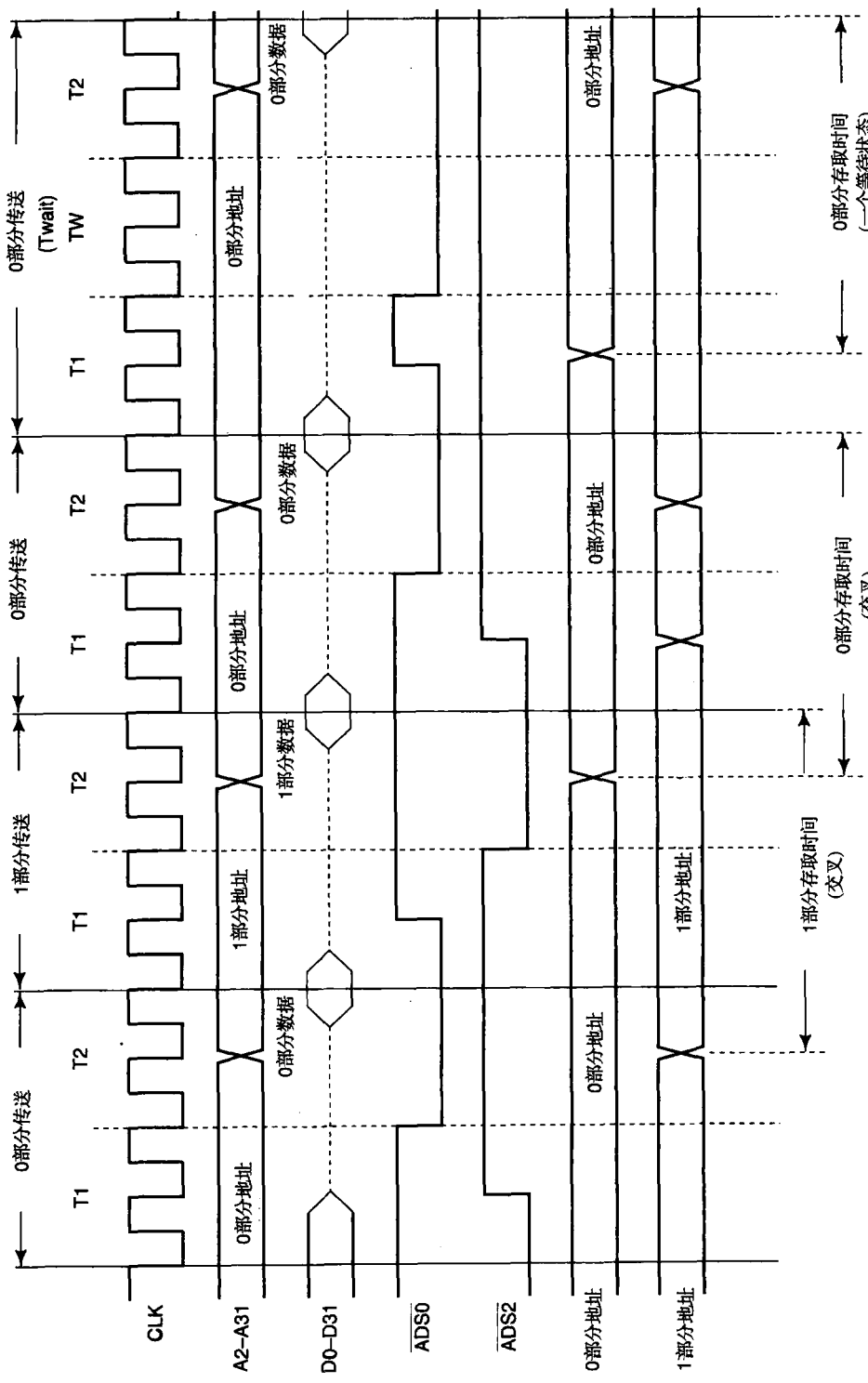


图17-5 交叉存储器系统的时序图，图中显示了两部分存储器的访问时间和地址信号

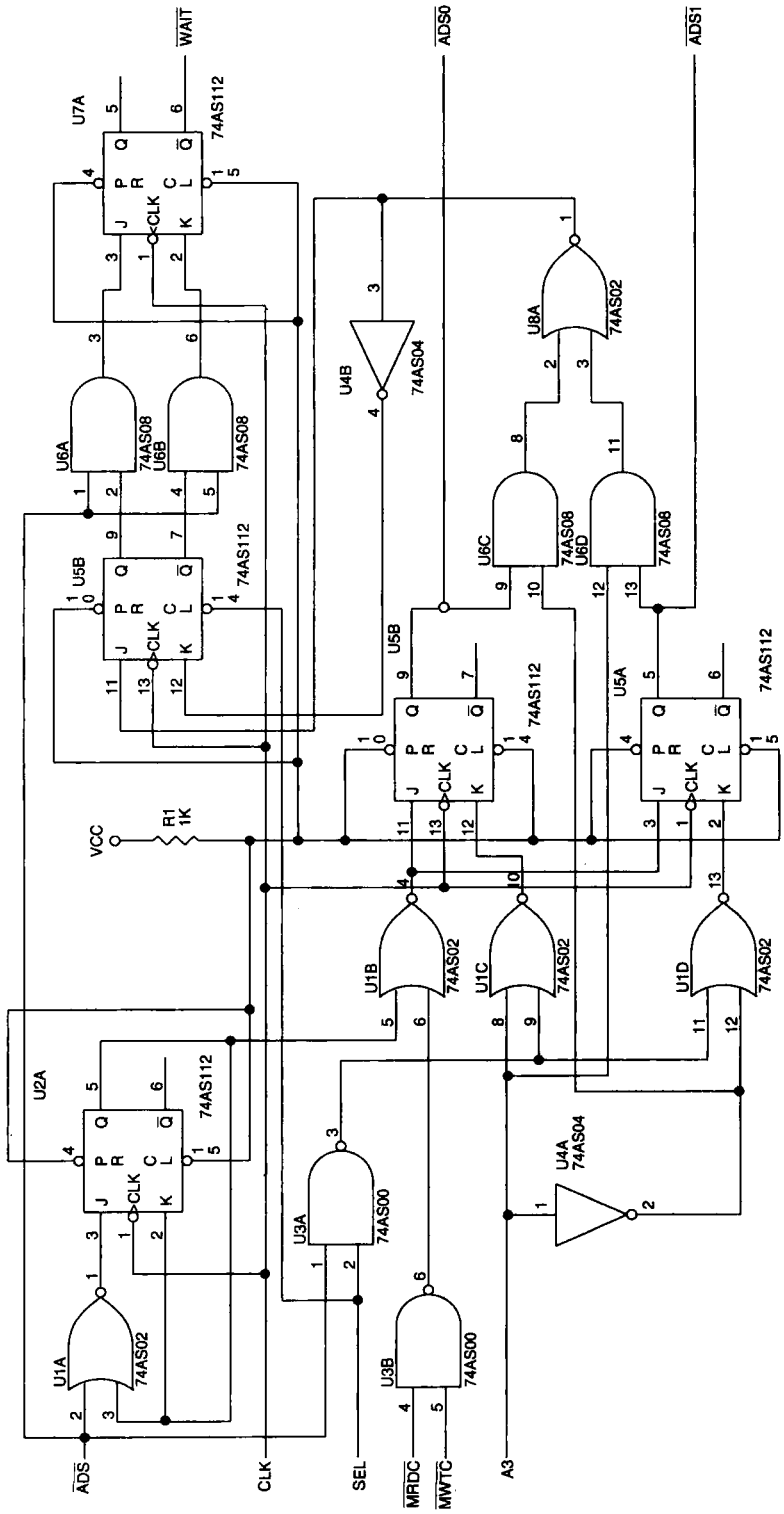


图17-6 交叉控制逻辑,产生控制交叉存储器的独立ADS信号和$\overline{\text{WAIT}}$信号

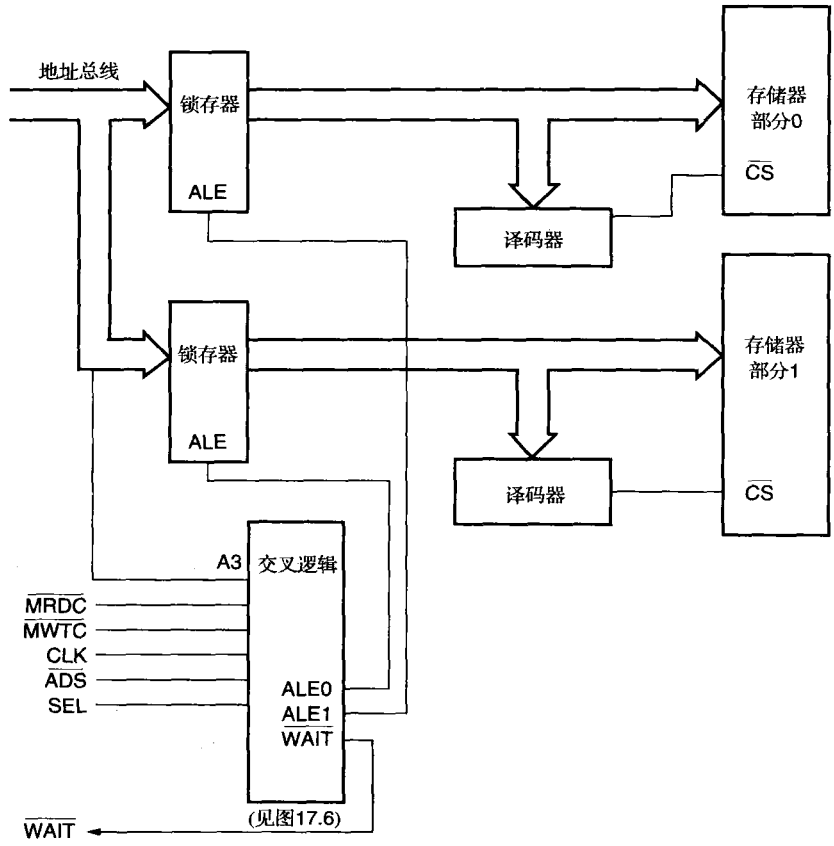


图 17-7 具有地址锁存和交叉逻辑电路的交叉存储系统

17.1.2 输入/输出系统

80386 中的 I/O 系统和任何基于 Intel 8086 系列微处理器系统中的 I/O 系统一样。如果采用独立的 I/O，系统中有 6KB 的 I/O 空间可用。采用独立的 I/O 空间，IN 和 OUT 指令用来在微处理器和 I/O 设备间传送 I/O 数据。I/O 端口地址在地址总线 A_{15}～A_2 上，并用$\overline{BE3}$～$\overline{BE0}$来选择传送一个字节、一个字或一个双字的 I/O 数据。如果采用存储器映射的 I/O，那么 I/O 空间的大小可以达到 4GB。对于存储器映射的 I/O，用于在微处理器和存储系统中传送数据的指令都可以用来进行 I/O 传送，因为 I/O 设备被当成存储器设备。由于在保护模式下 80386 提供了 I/O 保护方案，因此几乎所有的 80386 系统都采用了独立编址的 I/O 空间。

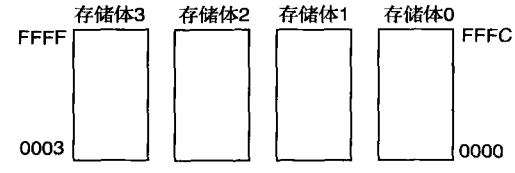


图 17-8 80386 微处理器独立的 I/O 编址。此处 4 个 8 位的存储体均被用来寻址 64K 个不同的 I/O 单元。I/O 单元被编号为 0000H～FFFFH

图 17-8 展示了 80386 微处理器的 I/O 地址映射。与早期 Intel 微处理器的 16 位宽的 I/O 地址映像不同，80386 使用了完全 32 位的被分为四个体的 I/O 系统。这一点与被分成四个存储体的存储器相同。我们通常用 ASCII 码（一种 7 位编码）在微处理器和打印机与键盘间传送字母和数字数据，所以大多数 I/O 传送的都为 8 位。如果统一码（Unicode，一种 16 位的字母数字编码）变得通用并取代 ASCII 码，这种情况或许会有变化。近来，已出现了一些 16 位甚至 32 位的 I/O 设备，如磁盘存储器和图像显示接口等。与 8 位相比，这些更宽的 I/O 通道增加了微处理器

到 I/O 设备间的数据传输率。

I/O 单元被编号为 0000H ～ FFFFH。I/O 地址映像的一部分供 80387 协处理器使用。虽然协处理器的端口号比一般的 I/O 地址映像要高得多，但是译码的 I/O 空间重叠时，将其考虑在内是非常重要的。协处理器利用 I/O 地址 800000F8H ～800000FFH 来与 80386 进行通信。为 80286 设计的协处理器 80287 则用 I/O 地址 00F8H ～00FFH 与 80286 通信。由于我们经常只对地址线 A_{15}～A_2 进行译码来选择 I/O 设备，所以应当注意，除非地址线 A_{31} 也被译码，否则协处理器将激活设备 00F8H ～00FFH。这应该不会出现什么问题，因为一般不用 I/O 端口 00F8H ～00FFH。

在 I/O 方面 80386 惟一新增加的特征是当 80386 运行在保护模式下时，在任务状态段 TSS 的尾端添加了 I/O 特权信息。如同有关存储管理章节中所述的一样，在保护模式下，I/O 单元会被封锁或禁止。如果被封锁的 I/O 单元被寻址，便会产生中断（类型 13，常规错误）。由于增加了这种机制，在多用户环境下，I/O 访问可以被禁止。和优先级一样，封锁也是保护模式的一个扩充。

17.1.3 存储器和 I/O 控制信号

存储器和 I/O 由各自的控制信号来控制。M/$\overline{IO}$信号指明当前的数据传送是在微处理器和存储器之间（M/$\overline{IO}$ = 1）还是在微处理器和 I/O 之间（M/$\overline{IO}$ = 0）进行。除了 M/$\overline{IO}$信号，存储器和 I/O 系统必须要读或写数据。W/$\overline{R}$ 信号为逻辑 0 时表示读操作，为逻辑 1 时表示写操作。$\overline{ADS}$信号用来限定 M/$\overline{IO}$和 W/$\overline{R}$ 控制信号。这和早期的 Intel 微处理器略有不同，在早期的 Intel 微处理器中不用$\overline{ADS}$信号限定。

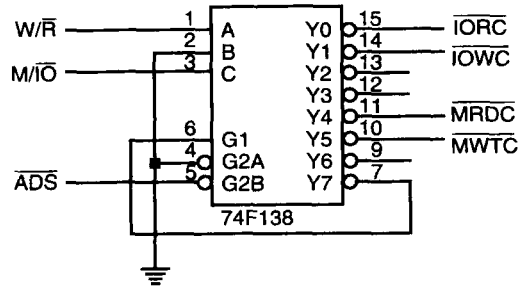


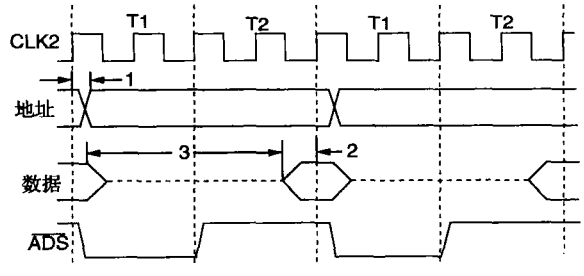
图 17-9 80386、80486 和 Pentium 的存储器和 I/O控制信号产生电路

图 17-9 是一个产生四个存储器和 I/O 设备控制信号的简单电路。注意，其中两个控制信号是为存储器控制（$\overline{MRDC}$和$\overline{MWTC}$）设计的，另两个是为 I/O 控制（$\overline{IORC}$和$\overline{IOWC}$）设计的。这些信号和 Intel 微处理器早期的型号中产生的存储器和 I/O 控制信号是一致的。

17.1.4 时序

时序对于理解 80386 微处理器与存储器和 I/O 设备的接口是非常重要的，图 17-10 为非流水线存储器读周期的时序图。注意，该时序以 CLK_2 的输入信号为基准，并且每个总线周期包含四个时钟周期。

每个总线周期包含两个时钟状态（T_1 和 T_2），每个时钟状态又包含两个时钟周期。请注意，在图 17-10 中，存取时间是标记为 3 的时间段。在 16MHz 型号的非流水线操作方式中，在插入等待状态以前存储器存取时间为 78ns。要选择非流水线方式，$\overline{NA}$引脚应置为逻辑 1。



	33MHz	25MHz	20MHz	16MHz
时间1:	4–15 ns	4–21 ns	4–30 ns	4–36 ns
时间2:	5 ns	7 ns	11 ns	11 ns
时间3:	46 ns	52 ns	59 ns	78 ns

图 17-10 80386 微处理器非流水线方式读时序

图 17-11 给出了 80386 运行在流水线方式下的读时序。请注意，由于地址被提前送出，这就为访问存储器中的数据提供了附加的时间。流水线方式通过在$\overline{NA}$引脚上置 0 来选定并利用地址锁存器来捕获流水线地址。加到地址锁存器上的时钟脉冲来自于$\overline{ADS}$信号。和交叉存储体系统一样，在流水线系统中也必须使用地址锁存器。两个和四个这种最小数量的交叉存储体已经成功地运用了。

注意，在流水线方式下，地址出现比非流水线方式下早一个完整的时钟状态。在 16MHz 的 80386

中，这使得对存储访问可以有 62.5ns 的附加时间；在非流水线系统中，允许的存储器存取时间为 78ns，在流水线方式的系统中，允许的存取时间可以为 140.5ns。流水线方式的优点在于它不需要等待状态（在许多但非所有的总线周期中），并且许多低速的存储器件也可以连接到微处理器上。它的缺点在于需要以交叉方式组织存储器来使用流水线，流水线要求有附加的电路，并且偶尔需要等待状态。

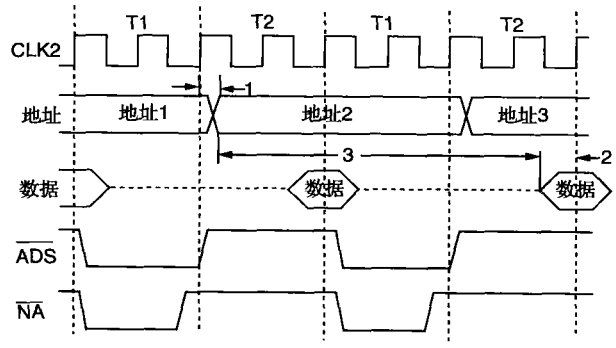


图 17-11 80386 微处理器流水线方式读时序

17.1.5 等待状态

如果存储器的存取时间比 80386 允许的存储器访问时间要长，那么就需要插入等待状态。在 33MHz 的非流水线方式的系统中，存储器存取时间仅为 46ns。目前，仅有少数存取时间为 46ns 的 DRAM 存储器。这意味着，必须引入等待状态来访问 DRAM（60ns 的 DRAM 需要 1 个等待状态）和访问存取时间为 100ns 的 EPROM（需要 2 个等待状态）。请注意，这种等待状态已嵌入主板之中，不能去掉。

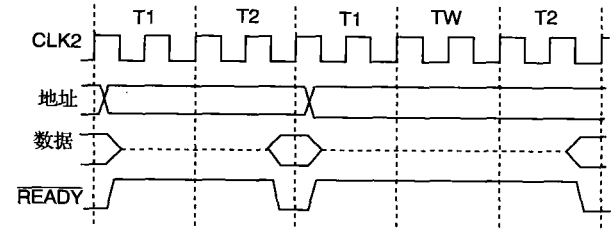


图 17-12 有 0 个或 1 个等待状态的非流水线方式的 80386 时序图

READY输入信号控制着是否向时序中插入等待状态。80386 中的READY信号是在每个总线周期中都必须被激活的动态输入信号。图 17-12 显示了两个总线周期，一个为正常总线周期（不含等待状态），另一个包含一个等待状态。注意如何控制READY产生 0 或 1 个等待状态。

在总线周期结束时对READY信号进行采样确定当前时钟周期是 T_2 还是 T_W。如果这时READY为 0，表明当前总线周期结束或者说是 T_2。如果在时钟周期结尾READY为 1，则当前时钟周期为 T_W，并且微处理器将继续检测READY直到其为 0，总线周期才结束。

在非流水线系统中，每当ADS变为逻辑 0，如果READY等于 1 就插入一个等待状态。在ADS变为逻辑 1 后，用时钟的上升沿进行计数来产生READY信号。在第一个时钟之后，如果不插入等待状态，则READY信号应变为逻辑 0；如果要插入 1 个等待状态，则READY信号必须保持至少两个时钟周期的逻辑 1。如果希望有更多的等待状态，那么READY应保持更长时间的逻辑 1。这样实质上可以往时序中插入任意个等待状态。

图 17-13 显示了一个对不同的存储器地址可插入 0 到 3 个等待状态的电路。此例中，访问 DRAM 时产生一个等待状态，访问 EPROM 时产生两个等待状态。每当ADS为低电平而 D/C 为高电平时，74F164 被清零。当ADS变为逻辑 1 之后，74F164 开始移位。随着移位寄存器的移位，该寄存器的值 00000000 开始从 QA 到 QH 逐位被逻辑 1 填充。四个不同的输出连接到一个反向多路器上，来产生低电平有效的READY信号。

17.2 特定的 80386 寄存器

80386 中有一系列早期的 Intel 微处理器所没有的新寄存器，如控制寄存器、调试寄存器及测试寄存器。控制寄存器 $CR_0 \sim CR_3$ 控制各种功能，调试寄存器 $DR_0 \sim DR_7$ 方便了调试，而测试寄存器 TR_6 和 TR_7 用于测试分页和高速缓存。

17.2.1 控制寄存器

除了以前讲过的 EFLAGS 和 EIP，80386 中还有其他的控制寄存器。控制寄存器 0（CR_0）和 80286

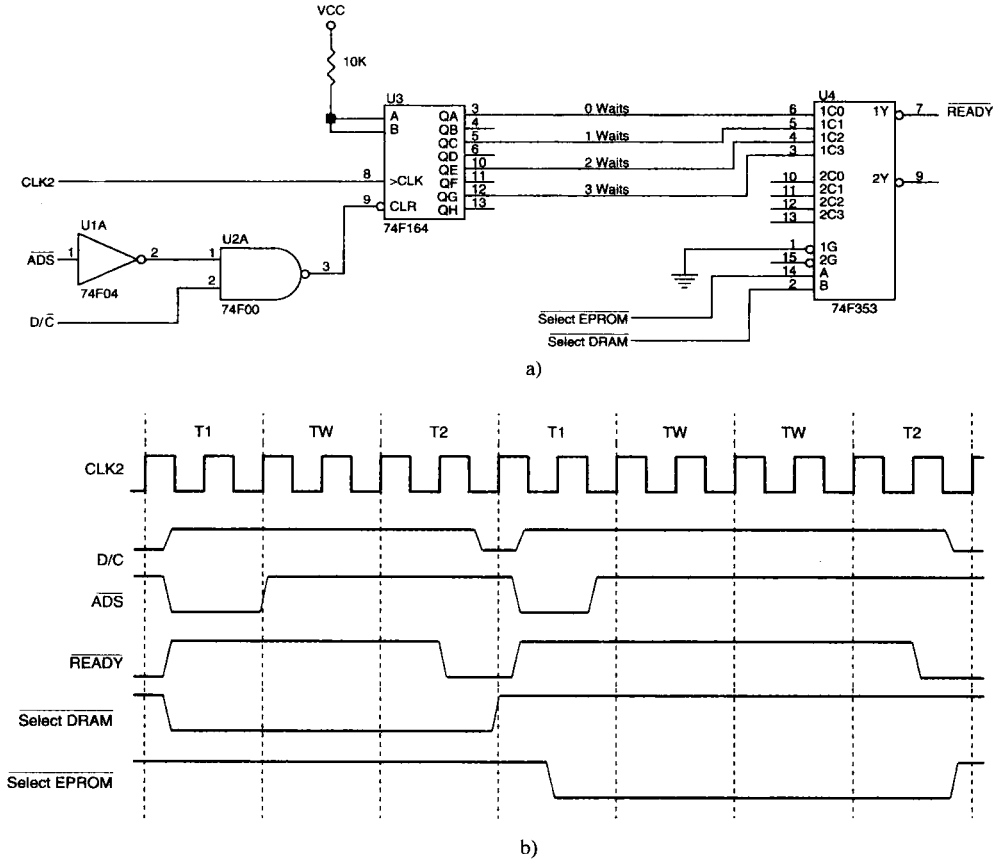


图 17-13 DRAM 选择 1 个等待状态、EPROM 选择 2 个等待状态的
a）电路图 b）时序图

中的 MSW（机器状态字）类似，只不过前者为 32 位而后者为 16 位。其他的控制寄存器还有 CR_1、CR_2 和 CR_3。

图 17-14 说明了 80386 控制寄存器的结构。控制寄存器 CR_1 在 80386 中没有使用，它是为后续产品保留的。控制寄存器 CR_2 保存着页故障中断之前所访问的最后一页的线性页地址。最后，控制寄存器 CR_3 保存着页目录的基地址。32 位页表地址的最右边 12 位总为 0，它们与其他位一起来确定 4K 长页表的起始地址。

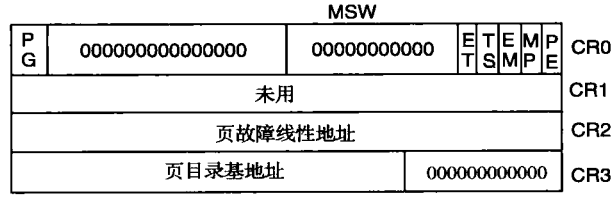


图 17-14 80386 微处理器控制寄存器结构

80386 中寄存器 CR_0 包含了许多特定的控制位，定义如下：

PG 当 PG = 1 时，选择线性地址到物理地址的**页表转换**（**page table trans/ation**）。页表转换可以把线性地址分配到物理存储单元。

ET 当 ET = 0 时，选择 **80287 协处理器**（***coprocessor***）；当 ET = 1 时，选择 **80387 协处理器**。设置此控制位是因为 80386 刚出现时还没有 80387。在多数系统中，ET 位被置位表明系统中有 80387。

TS 表明80386已经**切换任务**（**switched task**）。在保护模式下，改变TR的内容将TS位置1。如果TS=1，协处理器指令将引起7号（协处理器不存在）中断。

EM **仿真**（**emulate**）位，设置该位可以使每条ESC指令引起7号中断（ESCape指令用来对80387协处理器指令编码）。通常通过这个中断用软件来仿真协处理器的功能。仿真的方法可以降低系统的成本，但执行仿真的协处理器指令通常至少需要100倍的时间。

MP 该位被设置则表明系统中**有协处理器**（**coprocessor is present**）。

PE 设置该位来选择80386的**保护模式**（**protected mode**）。它也可以被清除而重新进入实模式。在80286中，该位只能被置位。80286只有硬件复位才能回到实模式，这使得在大多数采用保护模式的系统中，不能使用实模式。

17.2.2 调试和测试寄存器

图17-15显示了调试和测试寄存器组。前四个调试寄存器包含32位线性断点地址（线性地址是由微处理器指令产生的32位地址，它可以和物理地址相同也可以不相同）。指向指令或数据的断点地址不断与程序产生的地址进行比较，如果调试寄存器DR_6和DR_7指向该断点地址，一旦出现地址匹配，80386将引起1号中断（TRAP或调试中断）。这项功能是对早期Intel微处理器上1号中断提供的基本陷阱或跟踪机制的很大扩充。断点地址在调试软件错误中非常有用。DR_6和DR_7的控制位定义如下：

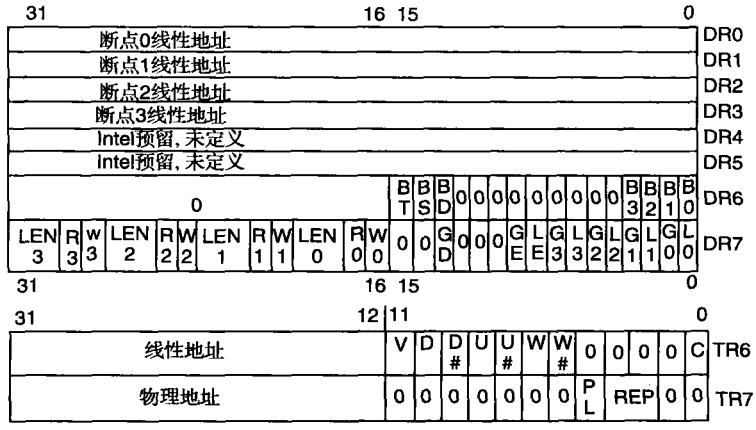


图17-15 80386的调试和测试寄存器（由Intel公司提供）

BT 如果置位（1），任务切换将引起调试中断。

BS 如果置位，标志寄存器中的TF位将引起调试中断。

BD 如果置位，读GD位被设置的调试寄存器将引起调试中断。GD位可以保护对调试寄存器的访问。

$B_3 \sim B_0$ 指示四个调试断点地址中哪个引起调试中断。

LEN 每四个长度的域对应了四个存储在$DR_0 \sim DR_3$的四个断点地址。这些位进一步定义了断点地址处访问的长度，如00（字节）、01（字）或11（双字）。

RW 每四个域的读写对应了四个存储在$DR_0 \sim DR_3$的四个断点地址。RW域选择选通断点地址的原因，如00（访问指令）、01（数据写）或11（数据读写）。

GD 如果置位，GD将通过产生调试中断来防止对调试寄存器的读写操作。

GE 如果置位，四个调试断点地址寄存器选择全局断点地址。

LE 如果置位，四个调试断点地址寄存器选择局部断点地址。

测试寄存器TR_6和TR_7用来测试**转换后备缓冲区**（**translation look-aside buffer，TLB**）。TLB与80386中的分页单元一起使用。TLB中保存了最常用的页表地址转换，减少了在页转换表中查找页转

换地址所需的存储器读次数。TLB 中保存了页表中最常用的 32 项，由 TR_6 和 TR_7 进行测试。

测试寄存器 TR_6 保存 TLB 的标志域（线性地址），TR_7 保存 TLB 的物理地址。给 TLB 中写入一项，需要以下步骤：

1）给 TR_7 写入期望的物理地址、PL 及 REP 值。

2）给 TR_6 写入线性地址，使 C = 0。

从 TLB 中读出一项，需要以下步骤：

1）给 TR_6 写入线性地址，使 C = 1。

2）读 TR_6 和 TR_7。如果 PL 位表示命中，则 TR_6 和 TR_7 的期望值就表示了 TLB 的内容。

TR_6 和 TR_7 中的位表示的条件如下：

V 表示 TLB 中的项有效。

D 表示 TLB 中的项无效或被修改过。

U TLB 的一位。

W 表示 TLB 寻址的区域是可写的。

C 选定往 TLB 中写（为 0 时）或立即查 TLB（为 1 时）。

PL 为逻辑 1 时表示命中。

REP 选定 TLB 中的哪一块被写入。

关于 TLB 功能的更多细节请参考有关存储管理和分页单元的章节。

17.3 80386 存储管理

80386 中的**存储管理单元**（**memory management unit**，**MMU**）与 80286 中的 MMU 相似，只是 80386 中包含了分页单元而 80286 中没有。MMU 完成将程序中输出的线性地址转换为物理地址的任务。80386 利用分页机制将物理地址分配到逻辑地址。因此，如果分页是激活的，即使程序指令要访问 A0000H 单元，而实际的物理地址也可能是 100000H 存储单元或其他单元。实际上这个特点使得对任意存储单元操作的软件都能运行在 80386 上，因为任何线性地址都可以变换为任意物理地址。早期的 Intel 微处理器没有这种灵活性。DOS 利用分页机制将 80386 和 80486 在 FFFFFH 以上的内存重定位到位于 ROM 的 D0000H ～ DFFFFH 之间的区域和其他可用区域。ROM 之间的区域通常称为高端内存，FFFFFH 以上的区域称为扩展内存。

17.3.1 描述符和选择子

在讨论存储分页单元之前，我们先看看 80386 微处理器的描述符和选择子。80386 使用描述符的风格与 80286 非常相似。在这两种微处理器中，**描述符**（**descriptor**）是描述和定位存储器段的连续的 8 个字节。**选择子**（段寄存器）用来从描述符表中检索描述符。80286 和 80386 之间的主要区别在于后者有两个新增加的选择子（FS 和 GS），并且 80386 中对描述符的两个非常重要的字节进行了定义。另一个区别在于 80386 的描述符使用 32 位的基地址和 20 位的界限域，而不是 80286 中的 24 位基地址和 16 位界限域。

80286 使用 24 位基地址寻址 16MB 的存储空间，由于 16 位界限域，它的段长度为 64KB。80386 用 32 位的基地址寻址 4GB 的存储空间，由于具有两种不同方式的 20 位界限域，它的段长度可以为 1MB 或 4GB。如果粒度位（G）为 0，20 位的界限域可以访问长度为 1MB 的段；如果 G 为 1，20 位的界限域允许的段长度为 4GB。

在 80386 的描述符中有一粒度位 G。如果 G = 0，界限域中的数值直接被解释为界限，取值范围为 00000H ～ FFFFFH；如果 G = 1，界限域中存储的数值被解释为 00000XXXH ～ FFFFFXXXH，这里 XXX 指的是 000H ～ FFFH 之间的任意一个值。这使得段界限范围为 0 ～ 4GB，步长为 4KB。界限值为 00001 时，当 G = 1，则界限大小为 4KB；当 G = 0，则界限大小为 1 个字节。例如，某段的起始物理地址为 100000000H，如果界限值为 00001H 并且 G = 0，则该段始于 10000000H，结束于 10000001H；如果 G = 1 而界限值不变（00001H），则该段始于 10000000H，结束于 10001FFFH。

图 17-16 说明了 80386 在保护模式下如何用选择子和描述符来寻址存储段。注意，这与 80286 段寻址的方法相同。不同之处在于 80386 所访问的段的大小不同。选择子用它的最左边的 13 位从描述符表中选择一个描述符，TI 位指示是局部（TI = 1）描述符表还是全局（TI = 0）描述符表。选择子的最右边两位用来定义要求的访问优先级别。

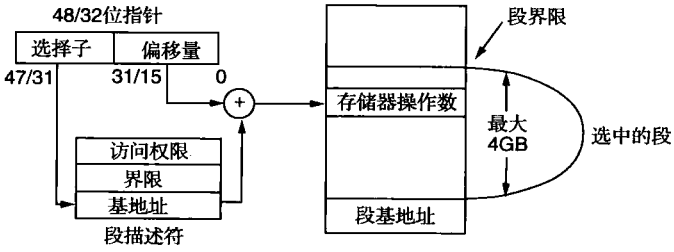


图 17-16 用段寄存器作为选择子的保护模式寻址（由 Intel 公司提供）

由于选择子用 13 位的代码访问描述符，所以在每个局部或全局描述符表中最多可以有 8192 个描述符。由于每个段（在 80386 中）可以为 4GB，而我们可以用两个描述符表访问 16 384 个段，这就使得 80386 可以访问的虚拟存储器为 64TB（1TB = 1024GB）。当然，存储系统中实际存在的存储器只能有 4GB。如果某时刻某程序需要多于 4GB 的存储器，可以将它在存储系统和磁盘驱动器或其他形式的大容量存储设备之间进行交换。

80386 有全局描述符表（GDT）和局部描述符表（LDT）。第三个描述符表是为中断（记为 IDT）描述符或门设计的。描述符的前 6 个字节与 80286 的相同，这使 80286 的软件与 80386 向上兼容（80286 描述符中的两个最重要的字节的值为 00H）。80286 和 80386 的描述符见图 17-17。80386 中的基地址为 32 位，界限域为 20 位，G 位选择界限的倍数（1 或 4K 倍）。80386 的描述符中各域的定义如下：

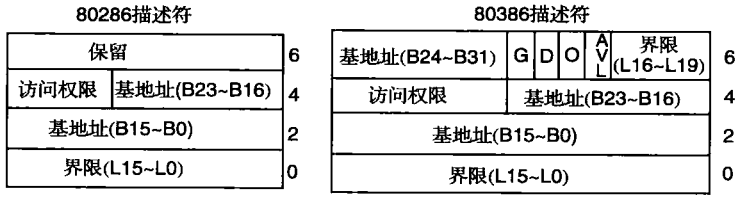


图 17-17 80286 和 80386 微处理器的描述符

Base（B_{31}～B_0） 定义 80386 微处理器在 4GB 物理地址空间内的 32 位起始段地址。

Limit（L_{19}～L_0） 定义段的界限，如果 G = 0，以字节为单位；如果 G = 1，以 4KB 为单位。这使得在 G = 0 时，段的长度可以为 1 个字节到 1MB；如果 G = 1，段的长度可以为 4KB 到 4GB。界限指示了段中的最后一个字节。

Access Rights 用来确定优先级及段的其他信息。不同类型的描述对应的这个字节也不相同，并且在每个描述符中都很复杂。

G 粒度位用来为界限域选择 1 或 4K 倍的倍数。如果 G = 0，则倍数为 1；如果 G = 1，则倍数为 4K。

D 选择默认寄存器宽度。如果 D = 0，寄存器为 16 位宽度，如同 80286；如果 D = 1，寄存器的宽度为 32 位，如同 80386。该位决定访问 32 位的数据或变址寄存器时是否需要加前缀。如果 D = 0，需要访问 32 位的寄存器和使用 32 位指针的前缀；如果 D = 1，需要访问 16 位的寄存器和使用 16 位指针的前缀。在汇编语言中 SEGMENT 语句后附加 USE_{16} 和 USE_{32} 伪指令可以控制设置 D 位。在实模式下，总是假定寄存器为 16 位宽，因此引用 32 位寄存器或指针的指令必须加前缀，DOS 的当前版本假定 D = 0。

AVL 这一位由操作系统以适当的方式使用，通常用来表示描述符所描述的段是可用的。

80386 微处理器中描述符有两种形式：段描述符和系统描述符。段描述符定义数据、堆栈和代码段；系统描述符定义系统中有关表、任务和门的信息。

段描述符

图 17-18 给出了段描述符。该描述符适合图 17-17 中所示的一般格式，只是访问权限位被用来指示该描述符所描述的数据段、堆栈段及代码段如何工作。访问权限字节的第 4 位用来决定该描述符是数据或代码段描述符（S = 1）还是系统描述符（S = 0）。注意，这些位的标记在不同版本的 Intel 文献中可能有所不同，但它们作用是相同的。

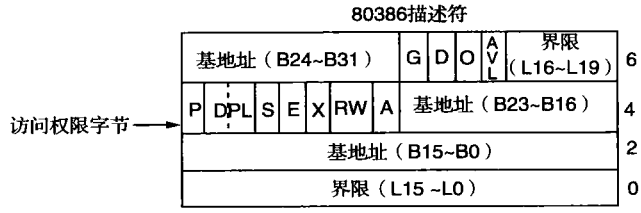


图 17-18 80386 段描述符的格式

以下内容描述了段描述符中的访问权限位和它们的功能：

P **存在（present）**位，为逻辑 1 时表示该段存在，如果 P = 0 并且通过描述符访问该段，则会产生 11 号中断。该中断表明所访问的段在系统中不存在。

DPL 描述符优先级（**descriptor privilege level**）设置描述符的优先级，这里 00 是最高优先级，11 为最低优先级，用来对段的访问进行保护。如果用一个比 DPL 低（数字较大）的优先级去访问该段，就会发生越权中断。在多用户系统中优先级用于防止访问系统存储区。

S **段（segment）**位，用来指示是数据或代码段描述符（S = 1）还是系统段描述符（S = 0）。

E **可执行（executable）**位，选择数据（堆栈）段（E = 0）或代码段（E = 1）。E 位还用来定义接下来的两位（X 和 RW）的功能。

X 如果 E = 0，则 X 位指示数据段的**扩展（expansion）**方向。若 X = 0，该段像数据段一样向上扩展；若 X = 1，该段像堆栈段一样向下扩展。如果 E = 1，X 位指示代码段的优先级将被忽略（X = 0）或要遵守（X = 1）。

RW 如果 E = 0，**RW 位（read/write bit）**指示数据段允许写（RW = 1）或不允许写（RW = 0）；如果 E = 1，RW 位指示代码段允许读（RW = 1）或不允许读（RW = 0）。

A **已访问（accessed）**位，微处理器每次访问段时，该位被置位。操作系统有时候用该位来跟踪那些已经被访问的段。

系统描述符

系统描述符如图 17-19 所示。80386 系统中有 16 种可能的系统描述符类型（如表 17-1 所示），但并非都在 80386 微处理器中使用。有些类型的描述符是为 80286 定义的，使 80286 上的软件与 80386 兼容。有些类型的描述符是新定义并且是 80386 所独有的。有些类型还没有定义，它们是为 Intel 后续产品保留的。

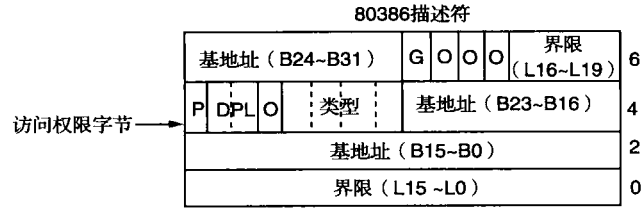


图 17-19 80386 系统描述符的一般格式

表 17-1 80386 系统描述符类型

类　型	用　途
0000	无效
0001	可用的 80286 TSS（任务状态段）
0010	LDT
0011	正在执行的 80286 TSS
0100	80286 调用门
0101	任务门（80386 及更高型号微处理器）

（续）

类　型	用　途
0110	80286 中断门
0111	80286 陷阱门
1000	无效
1001	可用的 80386 及更高型号微处理器 TSS
1010	保留
1011	正在执行的 80386 及更高型号微处理器 TSS
1100	80386 及更高型号微处理器调用门
1101	保留
1110	80386 及更高型号微处理器中断门
1111	80386 及更高型号微处理器陷阱门

17.3.2 描述符表

描述符表定义了 80386 保护模式下用到的所有的段。共有三种类型的描述符表：全局描述符表（GDT）、局部描述符表（LDT）和中断描述符表（IDT）。80386 用来寻址这三个表的寄存器分别称为全局描述符表寄存器（GDTR）、局部描述符表寄存器（LDTR）和中断描述符表寄存器（IDTR）。这三个寄存器分别用指令 LGDT、LLDT 和 LIDT 来加载。

描述符表（descriptor table）是一个不定长的数组，每项保存了 8 字节长的描述符。局部和全局描述符表各自最多可以有 8192 项，而中断描述符表最多可有 256 项。局部或全局描述符表中描述符的检索是利用段寄存器中的选择子来进行的。图 17-20 显示了保护模式下段寄存器及其中的选择子。最左边的 13 位检索描述符，TI 位选定是局部（TI = 1）还是全局（TI = 0）描述符表；而 RPL 位指示请求优先级。

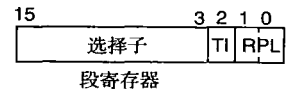


图 17-20　段寄存器中的选择子、TI 位及请求优先级（RPL）位

每当新的选择子被放到某个段寄存器时，80386 就会访问描述符表中的某一项，并自动将对应的描述符加载到该段寄存器的程序不可见的 cache 部分。只要该选择子与段寄存器中的选择子保持相同，不需要对该描述符表有额外的访问。从描述符表中取出一条新描述符的操作不可见是因为保护模式下每当段寄存器的内容被修改，微处理器会自行完成这个过程。

图 17-21 说明了存放在地址 00010000H 的全局描述符表（GDT）如何通过段寄存器和它的选择子被访问。这个表共有四项。第一项是一个空（0）描述符。描述符 0 必须总为空描述符。其他项寻址 80386 保护模式下存储系统中各种段。在此图中，数据段寄存器值为 0008H，这意味着选择子检索的是全局描述符表（TI = 0）里的 1 号描述符，请求优先级为 00。1 号描述符地址比描述符表基地址高 8 个字节，从 00010008H 开始。位于这个存储单元的描述符就可以寻址基址为 00200000H、限界为 100H 的存储单元，这也就意味着该描述符可寻址存储单元 00200000H ～ 00200100H。由于这是一个数据段（DS）寄存器，所以数据段位于存储系统中的这些位置。如果数据访问超出这个范围，则引发一个中断。

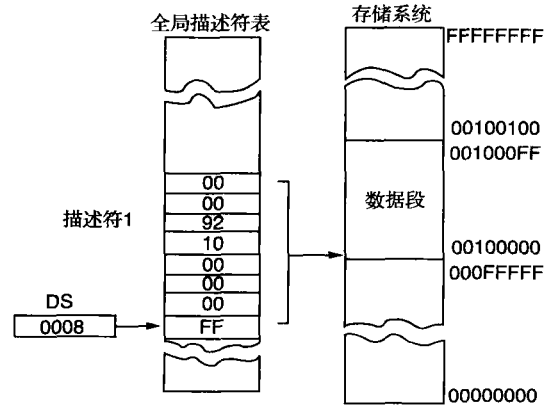


图 17-21　用 DS 寄存器从全局描述符表中选择描述符。在本例中，DS 寄存器访问数据段中 00100000H ～ 001000FFH 存储单元

局部描述符表（LDT）和全局描述符表（GDT）访问方式相同。惟一的区别在于访问全局描述符

表时 TI 位被清零而访问局部描述符表时 TI 位被置位。如果检查全局和局部描述符表寄存器，还存在另外一个区别：全局描述符表寄存器（GDTR）包含全局描述符表的基地址和界限，局部描述符表寄存器（LDTR）仅包含一个 16 位宽的选择子。LDTR 的内容寻址 0010 类型的包含 LDT 的基地址和界限的系统描述符。这种方案使得多个任务可以共用一个全局描述符表，并且如果需要，每个任务可以有一个或多个局部描述符表。全局描述符描述了系统的存储器，而局部描述符描述了应用程序或任务的存储器。

类似 GDT，中断描述符表 IDT 通过将基地址和界限存入中断描述符表寄存器来进行寻址。GDT 和 IDT 之间的主要区别在于 IDT 只包含了中断门。GDT 和 LDT 包含了系统和段描述符，但不包含中断门。

图 17-22 给出了门描述符，这是一种特殊格式的系统描述符，在前面我们已经介绍过了（不同的门描述符类型参见表 17-1）。注意，门描述符包含一个 32 位的偏移地址、一个字计数和一个选择子。32 位的偏移地址指向中断服务程序或其他程序入口。字计数指示已有多少字从调用者堆栈传送到调用门所调用过程的堆栈上。这个从调用者堆栈中传送数据的特性对于实现如 C/C++ 等高级语言非常有用。注意，字计数域不用于中断门。选择子用来指示任务状态段（TSS）在 GDT 中的位置；如果它是一个局部过程，则指示在 LDT 中的位置。

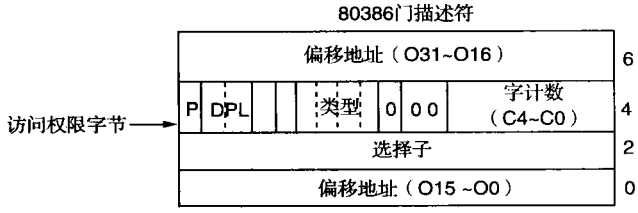


图 17-22 80386 微处理器中的门描述符

每当一个门被访问，选择子的内容就被加载到任务寄存器（TR）中，引起一个任务切换。门的接收取决于优先级。返回指令（RET）终止门调用过程，而中断返回指令（IRET）终止中断门过程。任务通常用 CALL 或 INT 指令来访问，调用指令寻址描述符表中的调用门和中断描述符中的调用地址。

实模式中断和保护模式中断之间的区别是在保护模式下中断向量表为 IDT。IDT 仍包含 256 级中断，但每个中断都是通过中断门而不是中断向量来调用的。这样，中断类型 2（INT 2）位于 IDT 中的 2 号描述符的位置，比 IDT 的基地址高 16 个字节。这也意味着存储器的前 1KB 不再像实模式下那样是中断向量表。IDT 可以位于存储系统的任意位置。

17.3.3 任务状态段（TSS）

任务状态段（task state segment，TSS）描述符和其他任何描述符一样，包含了任务状态段的位置、大小和优先级别。而它们之间的区别在于 TSS 描述符所描述的 TSS 段不包含数据和代码。它包含了任务状态和联系，以使任务可以被嵌套（第一个任务可以调用第二个任务，而第二个又可以调用第三个，如此类推）。TSS 描述符由**任务寄存器（task register，TR）**寻址。TR 的内容由 LTR 指令或保护模式下运行的程序中的远程 JMP 或 CALL 指令来改变。LTR 指令用来在系统初始化过程中首次访问一项任务。在初始化之后，CALL 或 JUMP 指令通常对任务进行切换。大多数情况下，我们用 CALL 指令来初始化一项新任务。

TSS 段如图 17-23 所示。可以看到，TSS 段是存储器中一个非常重要的数据结构，包含许多不同类型的信息。TSS 的第一个字标记为**返回链（back-link）**。它是一个选择子，由返回指令（RET 或 IRET）将其装入 TR 寄存器从而回到前一个 TSS。下一个字的值必须为 0。第二到第七双字中包含优先级 0～2 的 ESP 和 ESS 值。当前任务被中断时要用这些值来对优先级 0～2 的堆栈进行寻址。第八个双字（偏移量为 1CH）包含 CR_3 的内容，CR_3 中保存前一个状态的页目录寄存器的基地址。如果分页有效，则必须保存这项信息。此后的 17 个双字的值被装入指定的寄存器。每当任务切换时，处理器当前所有状态信息（所有寄存器）被保存在 TSS 的这些单元中，然后又从新任务的 TSS 的相同单元装入新值。最后一个字（偏移量 66H）中包含 I/O 允许位图基地址。

I/O 允许位图（I/O permission bit map）使 TSS 可以通过一个 I/O 允许拒绝中断来封锁对已经禁止的 I/O 端口地址的操作。允许拒绝中断是类型为 13 的一般错误保护中断。I/O 允许位图基地址为相对

于 TSS 起始位置的偏移地址。这使得同一个允许位图可以被多个 TSS 使用。

每个 I/O 允许位图大小为 64Kb（8KB），起始地址为 I/O 允许位图基地址所表示的偏移地址。I/O 允许位图的首字节为 I/O 端口 0000H～0007H 的许可位，最右端的一位为端口 0000H 的许可位，最左一位为端口 0007H 的许可位。位图中每位和 I/O 端口号的对应关系就是这样，一直到图最后一个字节的最左一位与最后一个端口 FFFFH 对应。I/O 允许位图中某位为逻辑 0 则对应 I/O 端口地址开放，为逻辑 1 则对应 I/O 端口地址被禁止。目前，只有 Windows NT、Windows 2000 和 Windows XP 使用 I/O 允许机制禁止与应用程序或用户相关的 I/O 端口。

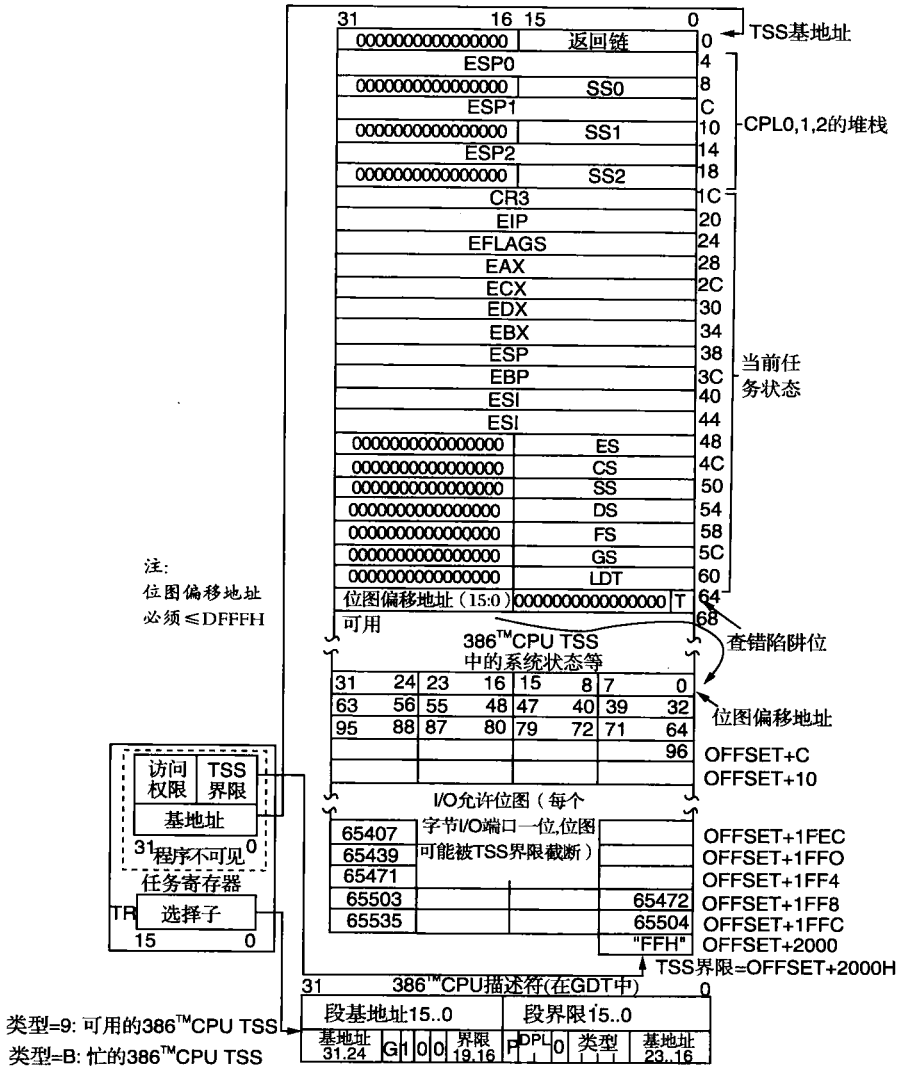


图 17-23 任务状态段（TSS）描述符（由 Intel 公司提供）

为了回顾任务切换操作，列出以下步骤，这个过程在 80386 微处理器上执行仅需 17μs：

1）门包含任务切换中过程地址或要跳转到的地址。另外，还包含 TSS 描述符的选择子号和参数传递中从调用者传送到用户栈上的字的数目。

2）选择子从门中装入 TR 寄存器（这一步由与有效的 TSS 描述符相关的 CALL 或 JMP 指令来完成）。

3）TR 选择 TSS。

4）将处理器当前状态保存在当前 TSS 中，然后从新任务的 TSS 中装入新的状态（所有寄存器值）。当前状态根据 TR 中当前的 TSS 选择子来保存。一旦存储完毕，一个新的 TSS 选择子的值（根据 CALL 或 JMP 指令）装入 TR 并从新的 TSS 中加载新的状态信息。

任务返回由以下几步完成：

1）将处理器当前状态存入当前 TSS。

2）将返回链选择子装入 TR 来访问前一个 TSS 以使得微处理器可以恢复到前一个状态。返回到调用 TSS 由 IRET 指令来完成。

17.4 向保护模式转换

为了将 80386 从实模式转换到保护模式，必须遵循几个步骤。在硬件复位或将 CR_0 中的 PE 位变为逻辑 0 后，处理器进入实模式。通过给 CR_0 寄存器中的 PE 位置 1，微处理器将进入保护模式，但在进行该操作之前，必须对其他方面做好初始化。下面的步骤将完成从实模式到保护模式的切换：

1）初始化中断描述符表，使其包含至少前 32 种中断类型的有效的中断门。IDT 可以（并且通常）拥有最多 256 个 8 字节的中断门，可以定义所有 256 个中断类型。

2）初始化全局描述符表（GDT），使其描述符 0 为一个空描述符，并且使其至少包含一个有效的代码段描述符、一个有效的堆栈段描述符、一个有效的数据段描述符。

3）将 CR_0 中的 PE 位置位，切换到保护模式。

4）执行一条段间（远）JMP 指令清除内部指令队列。

5）将初始选择子的值装入到所有的数据选择子（段寄存器）中。

6）现在 80386 已运行在保护模式下，正在使用 GDT 和 IDT 中定义的段描述符。

图 17-24 显示了利用以上 1～6 步所设置的保护系统存储器映像。完成此任务的软件在例 17-1 中给出。该系统包含了一个数据段描述符和一个代码段描述符，每个段被设置为 4GB 大。这是可能的最简单的保护模式下的系统——**平展模型（flat model）**，除了代码段寄存器以外，其余的段寄存器中都装入了与 GDT 中相同的数据段描述符。优先级被初始化为 00，即最高优先级。这个系统常用于用户对微处理器有访问权并且需要访问整个存储空间的场合。这个程序被设计用在那些没有用到 DOS 或从 Windows 到 DOS 的 SHELL 的系统中。本节后面将展示如何在 DOS 环境下进入到保护模式（请注意，例 17-1 中的软件是为独立的系统设计的，例如 80386EX 嵌入式处理器，而不是为 PC 设计的）。

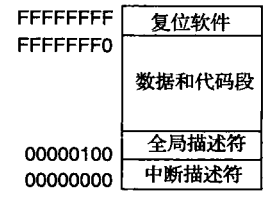


图 17-24 例 17-1 的存储器映像

例 17-1 中没有在中断描述符表内存放任何中断向量，因为在例子中没用。如果要用中断向量，必须包含将中断服务过程地址装入 IDT 的软件。软件必须分开两个部分生成，然后连接在一起写入 ROM 中，如按照说明的那样把第一部分写成实模式，而第二部分（见程序列表的注释）使用 32 位平展模型由汇编程序调整为保护模式代码。在 PC 机上这个软件不能执行，因为它是为在嵌入式系统上执行而写的。它被汇编后，必须在写入 ROM 之前用 EXE2BIN 转换成二进制文件。

例 17-1

```
.MODEL SMALL
.386P
ADR     STRUC
        DW      ?               ;地址结构
        DD      ?
ADR     ENDS
.DATA
IDT     DQ      32 DUP(?)       ;中断描述表
GDT     DQ      8               ;全局描述表
```

```
DESC1   DW      0FFFFH              ;代码描述符
        DW      0
        DW      0
        DW      9EH
        DW      8FH
        DW      0
DESC2   DW      0FFFFH              ;数据段描述符
        DW      0
        DW      0
        DW      92H
        DW      8FH
        DW      0
IDTR     ADR        <0FFH,IDT>     ;IDTR 数据
GDTR     ADR        <17H,GDT>      ;GDTR 数据
JADR     ADR        <8,PM>         ;远跳转指令JMP数据
.CODE
.STARTUP
        MOV   AX,0                  ;初始化     DS
        MOV   DS,AX

        LIDT  IDTR                  ;初始化     IDTR
        LGDT  GDTR                  ;初始化     GDTR

        MOV   EAX,CR0               ;设置 PE
        OR    EAX,1
        MOV   CR0,EAX

         JMP   JADR                 ;远跳转到     PM
PM::                                ;强制远标号

;下列软件必须单独用汇编语言开发，以便产生32位保护模式代码

;  即:       .MODEL  FLAT

       MOV   AX,10H                 ;装载段寄存器
       MOV   DS,AX                  ;进入保护模式
       MOV   ES,AX
       MOV   SS,AX
       MOV   FS,AX
       MOV   GS,AX
       MOV   SP,0FFFF000H

;其他初始化放在这里
end
```

在越来越复杂的系统中（不可能出现在嵌入式系统中），在保护模式下初始化系统所需的步骤也就越复杂。通常对于多用户的复杂系统，可以利用任务状态段（TSS）来加载寄存器。对于更复杂的系统，利用任务切换来使 80386 进入到保护模式所需的步骤如下：

1）初始化中断描述符表，以便用 IDT 中的至少 32 个描述符提供有效的中断描述符。

2）初始化全局描述符表（GDT），以便使其最少有一个任务状态段（TSS）描述符和初始任务所需要的初始代码及数据段描述符。

3）初始化任务寄存器（TR），使它指向一个 TSS。

4）用一条段间（远）跳转指令清掉内部指令队列，切换到保护模式下。将当前的 TSS 选择子装入到 TR 寄存器中并初始化任务。

5）现在 80386 在第一个任务控制下运行于保护模式。

例 17-2 显示初始化系统和利用任务切换转换到保护模式所需的软件。初始系统任务运行在最高保护级别（00）并控制着整个 80386 运行环境。在许多情况下，该任务用来在多用户环境中引导（加载）允许多用户访问系统的软件。与例 17-1 一样，这个软件不能在 PC 机上运行，被设计只能运行在嵌入式系统上。

例 17-2

```
.MODEL SMALL
.386P
.DATA
ADR     STRUC                       ;48位地址结构
        DW      ?                   ;选择子
        DD      ?                   ;偏移
ADR     ENDS
DESC    STRUC                       ;描述符结构
        DW      ?
        DW      ?
        DB      ?
        DB      ?
        DB      ?
        DB      ?
DESC    ENDS
TSS     STRUC                       ;TSS 结构
        DD      18 DUP(?)
        DD      18H                 ;ES
        DD      10H                 ;CS
        DD      4 DUP(18H)
        DD      28H                 ;LDT
        DD      IOBP                ;IO 特权位图
TSS     ENDS
GDT     DESC    <>                              ;null
        DESC    <2067H,TS1,0,89H,90H,0>         ;TSS 描述符
        DESC    <-1,0,0,9AH,0CFH,0>             ;代码段
        DESC    <-1,0,0,92H,0CFH,0>             ;数据段
        DESC    <0,0,0,0,0,0>                   ;TSS 的 LDT
LDT     DESC    <>                  ;null
IOBP    DB      2000H DUP(0)        ;允许所有I/O
IDT     DQ      32 DUP(?)           ;IDT
TS1     TSS     <>                  ;形成 TSS
IDTA    ADR     <0FFH,IDT>          ;IDTR
GDTA    ADR     <27H,GDT>           ;GDTR
JADR    ADR     <10H,PM>            ;跳转地址
.CODE
.STARTUP
        MOV   AX,0
        MOV   DS,AX
        LGDT  GDTA
        LIDT  IDTA
        MOV   EAX,CR0
        OR    EAX,1
        MOV   CR0,EAX
        MOV   AX,8
        LTR   AX
        JMP   JADR
PM:
        ;保护模式
END
```

例 17-1 和例 17-2 不是为在 PC 机环境中运行而编写的。PC 机环境需要使用由 DOS 中的 HIMEM. SYS 驱动程序提供的**虚拟控制程序接口（virtual control program interface，VCPI）**驱动程序，或者使用由 Windows 进入 DOS shell 时提供的 **DOS 保护模式接口（DOS protected mode interface，DPMI）**驱动程序。例 17-3 说明如何使用 DPMI 切换到保护模式，然后显示存储器任意区域中的内容，包括扩展寄存器和其他地址的内容。该 DOS 应用程序可以以十六进制方式显示任意存储单元的内容，包括在存储系统中位于开始 IMB 以上的存储单元。

例 17-3

```
                        ;该程序显示存储区的内容，包括扩展存储器
                        ;
                        ;*** 命令行语法***
                        ;EDUMP XXXX,YYYY     这里，XXXX 是起始地址，YYYY 是结束地址

                        ;注意，该程序必须在Windows下执行
                        ;
                                .MODEL SMALL
                                .386
                                .STACK 1024             ;1024字节的堆栈区
0000                            .DATA
0000  00000000           ENTRY DD   ?                   ;DPMI 入口点
0004  00000000           EXIT  DD   ?                   ;DPMI 出口点
0008  00000000           FIRST DD   ?                   ;第一个地址
000C  00000000           LAST1 DD   ?                   ;最后一个地址
0010  0000               MSIZE DW   ?                   ;DPMI 需要的内存
0012  0D 0A 0A 50 61     ERR1  DB   13,10,10,'Parameter error.$'
      72 61 6D 65 74
      65 72 20 65 72
      72 6F 72 2E 24
0026  0D 0A 0A 44 50     ERR2  DB   13,10,10,'DPMI not present.$'
      4D 49 20 6E 6F
      74 20 70 72 65
      73 65 6E 74 2E
      24
003B  0D 0A 0A 4E 6F     ERR3  DB   13,10,10,'Not enough real memory.$'
      74 20 65 6E 6F
      75 67 68 20 72
      65 61 6C 20 6D
      65 6D 6F 72 79
      2E 24
0056  0D 0A 0A 43 6F     ERR4  DB   13,10,10,'Could not move to protected mode.$'
      75 6C 64 20 6E
      6F 74 20 6D 6F
      76 65 20 74 6F
      20 70 72 6F 74
      65 63 74 65 64
      20 6D 6F 64 65
      2E 24
007B  0D 0A 0A 43 61 ERR5  DB   13,10,10,'Cannot allocate selector.$'
      6E 6E 6F 74 20
      61 6C 6C 6F 63
      61 74 65 20 73
      65 6C 65 63 74
      6F 72 2E 24
0098  0D 0A 0A 43 61 ERR6  DB   13,10,10,'Cannot use base address.$'
      6E 6E 6F 74 20
      75 73 65 20 62
      61 73 65 20 61
      64 64 72 65 73
      73 2E 24
00B4  0D 0A 0A 43 61 ERR7  DB   13,10,10,'Cannot allocate 64K to limit.$'
      6E 6E 6F 74 20
      61 6C 6C 6F 63
      61 74 65 20 36
      34 4B 20 74 6F
      20 6C 69 6D 69
      74 2E 24
00D5  0D 0A 24           CRLF  DB   13,10,'$'
00D8  50 72 65 73 73     MES1  DB   'Press any key...$'
      20 61 6E 79 20
      6B 65 79 2E 2E
      2E 24
```

```
                        ;
                        ;存放DPMI功能0300H的寄存器数组
                        ;
00E9   = 00E9           ARRAY EQU  THIS BYTE
00E9   00000000         REDI  DD   0                    ;EDI
00ED   00000000         RESI  DD   0                    ;ESI
00F1   00000000         REBP  DD   0                    ;EBP
00F5   00000000               DD   0                    ;保留
00F9   00000000         REBX  DD   0                    ;EBX
00FD   00000000         REDX  DD   0                    ;EDX
0101   00000000         RECX  DD   0                    ;ECX
0105   00000000         REAX  DD   0                    ;EAX
0109   0000             RFLAG DW   0                    ;标志
010B   0000             RES   DW   0                    ;ES
010D   0000             RDS   DW   0                    ;DS
010F   0000             RFS   DW   0                    ;FS
0111   0000             RGS   DW   0                    ;GS
0113   0000             RIP   DW   0                    ;IP
0115   0000             RCS   DW   0                    ;CS
0117   0000             RSP   DW   0                    ;SP
0119   0000             RSS   DW   0                    ;SS
0000                          .CODE
                              .STARTUP
0010   8C C0                  MOV  AX,ES
0012   8C DB                  MOV  BX,DS                ;找到程序和数据大小
0014   2B D8                  SUB  BX,AX
0016   8B C4                  MOV  AX,SP                ;找到堆栈大小
0018   C1 E8 04               SHR  AX,4
001B   40                     INC  AX
001C   03 D8                  ADD  BX,AX                ;BX = 按段计算的长度
001E   B4 4A                  MOV  AH,4AH
0020   CD 21                  INT  21H                  ;修改已分配的内存块
0022   E8 00D1                CALL GETDA                ;取命令行信息
0025   73 0A                  JNC  MAIN1                ;如果参数正确
0027   B4 09                  MOV  AH,9                 ;参数错误
0029   BA 0012 R              MOV  DX,OFFSET ERR1
002C   CD 21                  INT  21H
002E   E9 00AA                JMP  MAINE                ;返回到 DOS
0031                    MAIN1:
0031   E8 00AB                CALL ISDPMI               ;加载了DPMI吗?
0034   72 0A                  JC   MAIN2                ;如果有 DPMI
0036   B4 09                  MOV  AH,9
0038   BA 0026 R              MOV  DX,OFFSET ERR2
003B   CD 21                  INT  21H                  ;显示没有DPMI
003D   E9 009B                JMP  MAINE                ;返回到  DOS
0040                    MAIN2:
0040   B8 0000                MOV  AX,0                 ;表示需要0内存
0043   83 3E 0010 R 00        CMP  MSIZE,0
0048   74 F6                  JE   MAIN2                ;如果DPMI不要内存
004A   8B 1E 0010 R           MOV  BX,MSIZE             ;得到数量值
004E   B4 48                  MOV  AH,48H
0050   CD 21                  INT  21H                  ;为DPMI分配内存
0052   73 09                  JNC  MAIN3
0054   B4 09                  MOV  AH,9                 ;如果没有足够的内存
0056   BA 003B R              MOV  DX,OFFSET ERR3
0059   CD 21                  INT  21H
005B   EB 7E                  JMP  MAINE                ;返回到 DOS
005D                    MAIN3:
005D   8E C0                  MOV  ES,AX
005F   B8 0000                MOV  AX,0                 ;16位应用
0062   FF 1E 0000 R           CALL DS:ENTRY             ;切换到保护模式
0066   73 09                  JNC  MAIN4
0068   B4 09                  MOV  AH,9                 ;如果切换失败
006A   BA 0056 R              MOV  DX,OFFSET ERR4
006D   CD 21                  INT  21H
006F   EB 6A                  JMP  MAINE                ;返回到 DOS
```

```
                        ;
                        ; 保护模式
                        ;
0071                    MAIN4:
0071  B8 0000                 MOV  AX,0000H              ;获得局部选择子
0074  B9 0001                 MOV  CX,1                  ;只需要1个
0077  CD 31                   INT  31H
0079  72 48                   JC   MAIN7                 ;如果出错
007B  8B D8                   MOV  BX,AX                 ;保存选择子
007D  8E C0                   MOV  ES,AX                 ;把选择子装入ES
007F  B8 0007                 MOV  AX,0007H              ;设置基地址
0082  8B 0E 000A R            MOV  CX,WORD PTR FIRST+2
0086  8B 16 0008 R            MOV  DX,WORD PTR FIRST
008A  CD 31                   INT  31H
008C  72 3D                   JC   MAIN8                 ;如果出错
008E  B8 0008                 MOV  AX,0008H
0091  B9 0000                 MOV  CX,0
0094  BA FFFF                 MOV  DX,0FFFFH             ;设置界限为64K
0097  CD 31                   INT  31H
0099  72 38                   JC   MAIN9                 ;如果出错
009B  B9 0018                 MOV  CX,24                 ;装入行计数
009E  BE 0000                 MOV  SI,0                  ;装入偏移
00A1                    MAIN5:
00A1  E8 00F4                 CALL DADDR                 ;如果需要，显示地址
00A4  E8 00CE                 CALL DDATA                 ;显示数据
00A7  46                      INC  SI                    ;指向下一个数据
00A8  66| A1 0008 R           MOV  EAX,FIRST             ;测试结束否
00AC  66| 3B 06 000C R        CMP  EAX,LAST1
00B1  74 07                   JE   MAIN6                 ;如果结束
00B3  66| FF 06 0008 R        INC  FIRST
00B8  EB E7                   JMP  MAIN5
00BA                    MAIN6:
00BA  B8 0001                 MOV  AX,0001H              ;释放描述符
00BD  8C C3                   MOV  BX,ES
00BF  CD 31                   INT  31H
00C1  EB 18                   JMP  MAINE                 ;返回到DOS
00C3                    MAIN7:
00C3  BA 007B R               MOV  DX,OFFSET ERR5
00C6  E8 0096                 CALL DISPS                 ;显示不能分配选择子
00C9  EB 10                   JMP  MAINE                 ;返回到DOS
00CB                    MAIN8:
00CB  BA 0098 R               MOV  DX,OFFSET ERR6
00CE  E8 008E                 CALL DISPS                 ;显示不能使用基地址
00D1  EB E7                   JMP  MAIN6                 ;释放描述符
00D3                    MAIN9:
00D3  BA 00B4 R               MOV  DX,OFFSET ERR7
00D6  E8 0086                 CALL DISPS                 ;显示不能分配64K界限
00D9  EB DF                   JMP  MAIN6                 ;释放描述符
00DB                    MAINE:
                              .EXIT
                        ;
                        ;ISDPMI过程测试DPMI是否存在
                        ;***退出参数***
                        ;carry = 1; 如果DPMI 存在，进位位=1
                        ;carry = 0; 如果DPMI 不存在，进位位=0
                        ;
00DF                    ISDPMI PROC NEAR

00DF  B8 1687                 MOV  AX,1687H              ;取DPMI状态
00E2  CD 2F                   INT  2FH                   ;DOS多路功能调用
00E4  0B C0                   OR   AX,AX
00E6  75 0D                   JNZ  ISDPMI1               ;如果无DPMI
00E8  89 36 0010 R            MOV  MSIZE,SI              ;保存所需内存总数
00EC  89 3E 0000 R            MOV  WORD PTR ENTRY,DI
00F0  8C 06 0002 R            MOV  WORD PTR ENTRY+2,ES
00F4  F9                      STC
00F5                    ISDPMI1:
00F5  C3                      RET
```

```
00F6                    ISDPMI  ENDP
                        ;
                        ; GETDA 过程获取命令行参数,
                        ; 以便按十六进制显示存储器
                        ;FIRST = 命令行的第一个地址
                        ;LAST1 = 命令行的最后地址
                        ;***返回参数***
                        ;如果有错, 进位位 (carry) =1
                        ;如果无错, 进位位 (carry) =0
                        ;
00F6                    GETDA   PROC NEAR

00F6  1E                        PUSH DS
00F7  06                        PUSH ES
00F8  1F                        POP  DS
00F9  07                        POP  ES                 ;ES 与 DS 变换
00FA  BE 0081                   MOV  SI,81H             ;寻址命令行
00FD                    GETDA1:
00FD  AC                        LODSB                   ;跳过空格
00FE  3C 20                     CMP  AL,' '
0100  74 FB                     JE   GETDA1             ;如为空格
0102  3C 0D                     CMP  AL,13
0104  74 1E                     JE   GETDA3             ;如果入口有错误
0106  4E                        DEC  SI                 ;调整 SI
0107                    GETDA2:
0107  E8 0020                   CALL GETNU              ;取第一个数
010A  3C 2C                     CMP  AL,','
010C  75 16                     JNE  GETDA3             ;如果是逗号则错误
010E  66| 26: 89 16 0008 R      MOV  ES:FIRST,EDX
0114  E8 0013                   CALL GETNU              ;取第二个数
0117  3C 0D                     CMP  AL,13
0119  75 09                     JNE  GETDA3             ;如果有错
011B  66| 26: 89 16 000C R      MOV  ES:LAST1,EDX
0121  F8                        CLC                     ;表示无错
0122  EB 01                     JMP  GETDA4             ;返回无错
0124                    GETDA3:
0124  F9                        STC                     ;指示出错
0125                    GETDA4:
0125  1E                        PUSH DS                 ;交换 ES 和 DS
0126  06                        PUSH ES
0127  1F                        POP  DS
0128  07                        POP  ES
0129  C3                        RET
012A                    GETDA ENDP
                        ;
                        ; GETNU 过程从命令行中提取数,
                        ; 把它放在 EDX 中返回
                        ;并且把命令行最后一个字符作为定界符放在 AL 中返回
012A                    GETNU   PROC NEAR

012A  66| BA 00000000          MOV  EDX,0              ;清除结果
0130                    GETNU1:
0130  AC                        LODSB                   ;从命令行取得数字
                                .IF  AL >= 'a' && AL <= 'z'
0139  2C 20                        SUB  AL,20H          ;转换为大写字母
                                .ENDIF
013B  2C 30                     SUB  AL,'0'             ;从 ASCII 码转换为数
013D  72 12                     JB   GETNU2             ;如是不是一个数
                                .IF  AL > 9             ;把 ASCII 码转换为 A~F
0143  2C 07                        SUB  AL,7
                                .ENDIF
0145  3C 0F                     CMP  AL,0FH
0147  77 08                     JA   GETNU2             ;如果不在 0~F 之间
0149  66| C1 E2 04             SHL  EDX,4
014D  02 D0                     ADD  DL,AL              ;把数字加到 EDX
014F  EB DF                     JMP  GETNU1             ;取下一位数
```

```
0151                    GETNU2:
0151  8A 44 FF                MOV   AL,[SI-1]          ;取定界符
0154  C3                      RET
0155                    GETNU  ENDP
                        ;
                        ;DISPC过程显示AL寄存器中得到的ASCII字符
                        ;*** 使用 ***
                        ;INT21H
                        ;
0155                    DISPC  PROC NEAR

0155  52                      PUSH DX
0156  8A D0                   MOV  DL,AL
0158  B4 06                   MOV  AH,6
015A  E8 0084                 CALL INT21H              ;完成实模式 INT 21H
015D  5A                      POP  DX
015E  C3                      RET

015F                    DISPC ENDP
                        ;
                        ;DISPS过程显示保护模式下由DS:EDX寻址的字符串
                        ;*** 使用 ***
                        ;DISPC
                        ;
015F                    DISPS  PROC NEAR

015F  66| 81 E2 0000FFFF    AND  EDX,0FFFFH
0166  67& 8A 02             MOV  AL,[EDX]            ;取得字符
0169  3C 24                 CMP  AL,'$'              ;检测是否结束
016B  74 07                 JE   DISP1               ;如果结束
016D  66| 42                INC  EDX                 ;寻址下一个字符
016F  E8 FFE3               CALL DISPC               ;显示字符
0172  EB EB                 JMP  DISPS               ;重复直到出现 $
0174                    DISP1:
0174  C3                      RET
0175                    DISPS  ENDP
                        ;
                        ;DDATA过程显示由ES: SI寻址的单元中的一个字节,
                        ;字节后面跟着一个空格
                        ;*** 使用***
                        ;DIP and DISPC
                        ;
0175                   DDATA PROC  NEAR

0175  26: 8A 04              MOV  AL,ES:[SI]          ;取得字节
0178  C0 E8 04               SHR  AL,4
017B  E8 000C                CALL DIP                 ;显示第一位
017E  26: 8A 04              MOV  AL,ES:[SI]          ;取得字节
0181  E8 0006                CALL DIP                 ;显示第二位
0184  B0 20                  MOV  AL,' '              ;显示空格
0186  E8 FFCC                CALL DISPC
0189  C3                     RET

018A                    DDATA ENDP
                        ;
                        ;DIP过程将AL中的右半字节显示成为一位十六进制数
                        ;*** 使用 ***
                        ;DISPC
                        ;

018A                   DIP   PROC NEAR

018A  24 0F                   AND  AL,0FH              ;获得右边 4 位组
```

```
018C  04 30                ADD  AL,30H              ;转换为ASCII码
                           .IF  AL > 39H            ;如果是A~F
0192  04 07                    ADD  AL,7
                           .ENDIF
0194  E8 FFBE              CALL DISPC               ;显示数
0197  C3                   RET

0198                DIP    ENDP
                    ;
                    ;DADDR过程显示DS：FIRST中的十六进制地址，
                    ;如果它是小段边界
                    ;***使用***
                    ;DIP, DISPS, DISPC, and INT21H
                    ;
0198                DADDR PROC NEAR

0198  66| A1 0008 R        MOV  EAX,FIRST           ;取得地址
019C  A8 0F                TEST AL,0FH              ;测试是否为XXXXXXX0
019E  75 40                JNZ  DADDR4              ;如果不是，则不显示地址
01A0  BA 00D5 R            MOV  DX,OFFSET CRLF
01A3  E8 FFB9              CALL DISPS               ;显示CR和LF
01A6  49                   DEC  CX                  ;行计数加1
01A7  75 18                JNZ  DADDR2              ;如果不是一页结束
01A9  BA 00D8 R            MOV  DX,OFFSET MES1      ;如果是一页结束
01AC  E8 FFB0              CALL DISPS               ;显示press any key
01AF                DADDR1:
01AF  B4 06                MOV  AH,6                ;获得任意键，不回显
01B1  B2 FF                MOV  DL,0FFH
01B3  E8 002B              CALL INT21H              ;完成实模式INT21H
01B6  74 F7                JZ   DADDR1              ;如果没有击键
01B8  BA 00D5 R            MOV  DX,OFFSET CRLF
01BB  E8 FFA1              CALL DISPS               ;显示CRLF
01BE  B9 0018              MOV  CX,24               ;复位行计数器
01C1                DADDR2:
01C1  51                   PUSH CX                  ;保存行计数
01C2  B9 0008              MOV  CX,8                ;装入位计数
01C5  66| 8B 16 0008 R     MOV  EDX,FIRST           ;获得地址
01CA                DADDR3:
01CA  66| C1 C2 04         ROL  EDX,4
01CE  8A C2                MOV  AL,DL
01D0  E8 FFB7              CALL DIP                 ;显示数
01D3  E2 F5                LOOP DADDR3              ;重复8次
01D5  59                   POP  CX                  ;获取行计数
01D6  B0 3A                MOV  AL,':'
01D8  E8 FF7A              CALL DISPC               ;显示冒号
01DB  B0 20                MOV  AL,' '
01DD  E8 FF75                CALL DISPC             ;显示空格
01E0                DADDR4:
01E0  C3                     RET

01E1                DADDR  ENDP
                    ;
                    ;INT21H过程在保持参数完整前提下，
                    ;获得实模式DOS INT21H指令访问
                    ;
01E1                INT21H PROC NEAR

01E1  66| A3 0105 R        MOV  REAX,EAX            ;保存寄存器
01E5  66| 89 1E 00F9 R     MOV  REBX,EBX
01EA  66| 89 0E 0101 R     MOV  RECX,ECX
01EF  66| 89 16 00FD R     MOV  REDX,EDX
01F4  66| 89 36 00ED R     MOV  RESI,ESI
01F9  66| 89 3E 00E9 R     MOV  REDI,EDI
01FE  66| 89 2E 00F1 R     MOV  REBP,EBP
0203  9C                   PUSHF
0204  58                   POP  AX
0205  A3 0109 R            MOV  RFLAG,AX
0208  06                   PUSH ES                  ;完成DOS中断
0209  B8 0300              MOV  AX,0300H
020C  BB 0021              MOV  BX,21H
020F  B9 0000              MOV  CX,0
0212  1E                   PUSH DS
0213  07                   POP  ES
0214  BF 00E9 R            MOV  DI,OFFSET ARRAY
```

```
0217  CD 31                  INT   31H
0219  07                    POP   ES
021A  A1 0109 R             MOV   AX,RFLAG          ;恢复寄存器
021D  50                    PUSH  AX
021E  9D                    POPF
021F  66| 8B 3E 00E9 R      MOV   EDI,REDI
0224  66| 8B 36 00ED R      MOV   ESI,RESI
0229  66| 8B 2E 00F1 R      MOV   EBP,REBP
022E  66| A1 0105 R         MOV   EAX,REAX
0232  66| 8B 1E 00F9 R      MOV   EBX,REBX
0237  66| 8B 0E 0101 R      MOV   ECX,RECX
023C  66| 8B 16 00FD R      MOV   EDX,REDX
0241  C3                    RET

0242                INT21H ENDP
                    END
```

或许读者注意到了 DOS INT 21H 功能调用在保护模式下必须进行不同的处理。例 17-3 的结尾部分给出了调用 DOS INT 21H 功能的过程，因为这段代码非常长并且很耗时，所以在 Windows 应用程序中应该尽量避免使用 DOS 中断。开发 Windows 软件的最好办法就是使用 C/C++，同时对于一些关键任务可以用汇编语言子程序。

17.5 虚拟 8086 模式

虚拟 8086 模式是一个特殊工作模式，至此我们还没有对其进行讨论。这种特殊运行模式的设计使得多个 8086 实模式的应用软件可以同时运行。利用 DOS 模拟器 cmd.exe（命令提示），PC 机允许 DOS 应用程序运行在这种模式下。图 17-25 举例说明了两个 8086 应用程序如何采用虚拟模式映射到 80386 中。操作系统允许多个应用程序执行，通常利用称为**时间片**（**time-slicing**）的技术。操作系统为每个任务分配一定的时间。例如，如果有三个任务在执行，操作系统为每个任务分配 1ms，这就意味着每过 1ms 就会发生一个任务到另一个任务的切换。在这种方式下，每个任务都得到一部分微处理器的运行时间，使得系统看上去就像在同时执行多个任务。任务占用微处理器的时间比例可以任意调整。

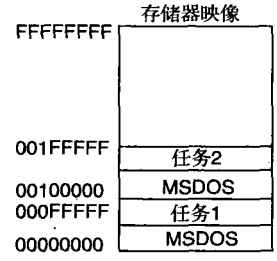


图 17-25 两个驻留在 80386 中以虚拟 8086 方式运行的任务

脱机打印程序就采用了这种技术。脱机打印程序可以运行在一个 DOS 分区上，拥有 10% 的访问时间片。这使系统可以使用脱机打印程序来打印，并且由于它仅占用了 10% 的系统时间，因此不会对系统有很大影响。

80386 保护模式和虚拟 8086 模式之间的主要区别在于微处理器对段寄存器的解释方式。在虚拟 8086 模式下，段寄存器与在实模式下的使用方式相同：用作能寻址从 00000H 到 FFFFFH 的 1MB 存储空间的段地址和偏移地址。通过下一节中将要介绍的分页单元，使得访问多个虚拟 8086 模式的系统成为可能。通过分页，程序仍然访问的是 1MB 以内的存储器，而微处理器可以访问存储系统中 4GB 范围内的任意物理存储单元。

通过将 EFLAG 寄存器中的 VM 位置为逻辑 1 就可以进入虚拟 8086 模式。如果优先级为 00，可以用 IRET 指令进入该模式。其他任何方式都不能对 VM 位进行设置。对 1MB 范围以外的存储单元的访问会引发 13 号中断。

虚拟 8086 模式通过对内存分区，每个用户都有一个 DOS 分区，可以使多个用户共享一个微处理器。分配给用户 1 的内存可能是 01000000H～01FFFFFFH，分配给用户 2 的内存可能是 02000000H～02FFFFFFH，以此类推。位于 00000000H～000FFFFFH 的系统软件则可以通过在用户间进行切换来将处理器时间分配给它们。通过这种方式，一个微处理器可以由多个用户共享。

17.6 内存分页机制

分页机制可以将程序产生的线性（逻辑）地址放入分页机制产生的物理存储页。**线性存储页**（**linear memory page**）是在实模式或保护模式下用选择子和偏移地址来寻址的页。**物理存储页**（**physical memory page**）则是实际存在的物理存储单元的页。例如，线性存储单元 20000H 通过分页机构可映射到物理存储单元 30000H 或其他任何单元。这意味着访问 20000H 单元时，实际上访问的是 30000H 单元。

每个 80386 存储页的大小为 4KB。利用分页机制对存储器进行分页，可以把系统软件放在任何物理地址。有三个组成部分用于页地址转换：页目录、页表和实际的物理存储页。注意，扩展存储管理器 EMM386. EXE 使用分页机制来在扩展内存中模拟扩充内存，并在系统 ROM 之间生成高端存储器块。

17.6.1 页目录

页目录最多包含 1024 个页转换表的地址，每个页转换表将一个逻辑地址转换为物理地址。页目录存储在内存中并通过页描述符地址寄存器（CR_3）（见图 17-14）来访问。控制寄存器 CR_3 保存着页目录的基地址，该基地址起始于任意 4KB 的边界。指令 MOV CR3，reg 用来对 CR_3 寄存器进行初始化。在虚拟 8086 方式的系统中，每个 8086 DOS 分区都有其自己的页目录。

页目录最多包含 1024 项，每项为四个字节。页目录自身占用一个 4KB 的存储页。页目录中每项（见图 17-26）转换存储地址的最左 10 位。线性地址的这 10 位部分用于在不同页表中查找不同页表项。存储在页目录项中的页表地址（A_{32}～A_{12}）可访问 4KB 长的页转换表。完全将线性地址转换成物理地址需要 1024 张大小为 4KB 的页表，另外还得加一个 4KB 大小的页目录。这种转换机制要求多达 4MB + 4KB 的存储空间来完成全地址转换。只有最大的操作系统才能支持如此大范围的地址转换。常见的操作系统在分页机制使能时只转换存储系统前 16MB 如 Windows。这种地址转换机制在页目录中需要 4 项（16 字节）并需要 4 个完整的页表（16KB）。

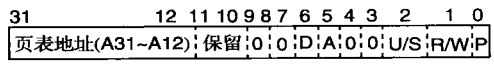


图 17-26 页表目录项

图 17-26 中所示的页目录项中各控制位的功能如下：

D **修改**（**dirty**）位。80386 对页目录项的这一位未做定义，它供操作系统使用。

A **访问**（**accessed**）位。每当微处理器访问该页目录时，访问位被置为逻辑 1。

R/W 和 U/S **读/写**（**read/write**）和**用户/超级用户**（**user/supervisor**）位均用于保护机制。如表 17-2 所示。这两位联合起来实现最低用户级（即第 3 级）的分页优先级的保护。

P **存在**（**present**）位，如果 P 为逻辑 1，表明地址转换中可以使用该项；如果 P 为逻辑 0，则不能用于转换。不存在项有其他用处，如表明该页当前位于磁盘上，如果 P = 0，该项的其他位可以用来指示该页在磁盘存储系统中的位置。

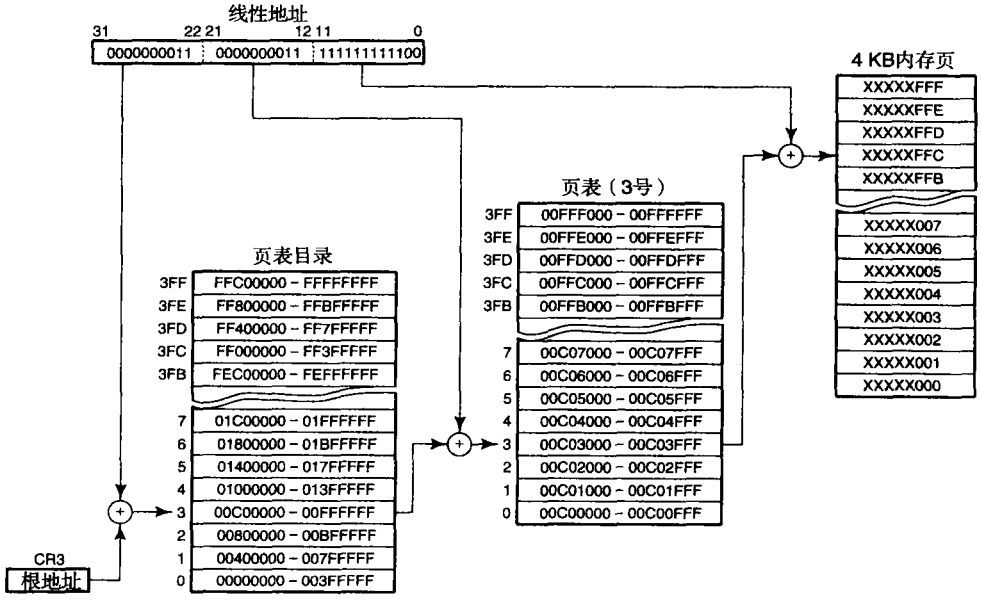
表 17-2 U/S 和 R/W 对优先级 3 的保护

U/S	R/W	优先级 3 的访问
0	0	无
0	1	无
1	0	只读
1	1	读/写

17.6.2 页表

页表中包含 1024 个物理页地址，用来将线性地址转换为物理地址。每张页表将线性存储器的一块 4MB 的区域转换为物理存储器中的 4MB。页表项的格式和页目录项的格式完全相同（参见图 17-26）。主要区别在于目录项包含页表的物理地址，而页表项则包含一个大小为 4KB 的物理页地址。另一个区别是在页目录项中 D（修改位）位没有定义，而在页表中该位用来指示对应物理页已被修改。

图 17-27 给出了 80386 微处理器的分页机制。这里，程序中产生的线性地址 00C03FFCH 经过分页机构被转换成物理地址 XXXXXFFCH（注意：XXXXX 是任意 4KB 大小的物理页地址）。分页机制的工作方式如下：



注：1. 图示页目录和页表中的地址范围表示选中的线性地址范围，而不是它们的内容。
2. 记在存储器页里的地址（XXXXX）由页表项选择。

图 17-27 线性地址 00C03FFCH 到物理存储地址 XXXXXFFCH 的转换，XXXXX 的值由页表项确定（此处未给出）

1）4KB 长的页目录存储在由 CR_3 所指定的物理地址。此地址常称为**根地址**（**root address**）。系统中，同时只有一个页目录。在虚拟 8086 模式中，每个任务都有其自己的页目录，可以将物理存储器的不同区域分配给不同的虚拟 8086 任务。

2）由本章前面讲过的描述符或实地址决定的线性地址的高 10 位（位 31 ～位 22）在分页机制中用来选择页目录项。这使页目录项与线性地址的最左 10 位对应起来。

3）页表通过页目录中的项进行寻址。在采用全地址转换系统中最多允许 4K 的页表。

4）接下来的 10 位（位 21 ～位 12）线性地址用于寻址页表中的项。

5）页表项中包含 4KB 大小的物理存储页的实际物理地址。

6）线性地址的最右边 12 位（位 11 ～位 0）用于选定物理存储页中的单元。

通过分页机制物理存储地址可以与任何线性地址相对应。例如，假定程序要选择线性地址 20000000H，但在物理存储系统中不存在这个单元。程序所指的页面大小为 4KB，地址为 20000000H ～ 20000FFFH。由于物理存储器中没有这部分，操作系统可能将诸如 12000000H ～ 12000FFFH 这样的物理存储器中存在的物理页赋给上面的线性地址区域。

在地址转换处理中，该线性地址的最左 10 位选定页目录项 200H，该项在页目录中的偏移地址为 800H。该页目录项包含线性地址 20000000H ～ 203FFFFFH 所对应页表的地址。线性地址位（21 ～ 12）在该页表中选择与 4KB 存储页的页对应的一项。对线性地址 20000000H ～ 20000FFFH，选中的是页表中的第一项（项 0）。该项中包含实际存储页的物理地址，此例中为 12000000H ～ 12000FFFH。

以一个典型的基于 DOS 的计算机系统为例，该系统的存储器映像如图 17-28 所示。注意，图中有尚未使用的空间，这一空间可以被分页并映射到不同的区域，为 DOS 实模式下的应用程序提供了更多的存储空间。正常

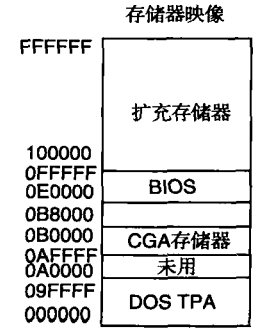


图 17-28 AT 风格的兼容产品中的内存分配

的 DOS 存储系统始于 00000H 单元，终于 9FFFFH 单元，该空间大小为 640KB。9FFFFH 单元以上的空间用于视频卡驱动、磁盘驱动以及系统 BIOS ROM。在此例中，9FFFFH 以上相邻的区域（A0000H～AFFFFH）未用。这个区域可由 DOS 使用，从而使得总的应用程序区域为 704KB 而非 640KB。将 A0000H～AFFFFH 域用作附加 RAM 时需要小心，因为在 12H 和 13H 方式中视频卡的驱动程序用该区域存储位图。

这段存储区可以被映射到位于 102000H～11FFFFH 的扩展内存单元。完成这个转换以及初始化页表目录和初始化必须设置内存的页表的软件如例 17-4 所示。注意，这个程序初始化了页表目录和页表，并加载了 CR_3 寄存器，它没有切换到保护模式，也没有启用分页。请注意实模式存储器操作中的分页功能。

例 17-4

```
.MODEL SMALL
.386P
.DATA

;页目录

PDIR   DD     4

;页表0

TAB0   DD     1024 dup(?)

.CODE
.STARTUP
       MOV  EAX,0
       MOV  AX,CS
       SHL  EAX,4
       ADD  EAX,OFFSET TAB0
       AND  EAX,0FFFFF000H
       ADD  EAX,7
       MOV  PDIR,EAX              ;寻址页目录
       MOV  ECX,256
       MOV  EDI,OFFSET TAB0
       MOV  AX,DS
       MOV  ES,AX
       MOV  EAX,7
       .REPEAT                    ;重映射00000H~9FFFFH
              STOSD               ;到00000H-9FFFFH
              ADD EAX,4096
       .UNTILCXZ
       MOV  EAX,102007H
       MOV  ECX,16
       .REPEAT                    ;重映射A0000H-AFFFFH
              STOSD               ;到102000H-11FFFFH
              ADD  EAX,4096
       .UNTILCXZ
       MOV  EAX,0
       MOV  AX,DS
       SHL  EAX,4
       ADD  EAX,OFFSET PDIR       ;装入CR3
       MOV  CR3,EAX

;重映射其他存储区另外的软件

       END
```

17.7 80486 微处理器简介

80486 微处理器是一个高度集成的器件，它含有 120 万个晶体管。在这个芯片内包含一个存储管理单元（MMU）、一个与 80387 兼容的完备的数字协处理器、一个含有 8KB 的一级 cache 以及一个与 80386 向上兼容的完全 32 位的微处理器。80486 目前有 25MHz、33MHz、50MHz、66MHz 及 100MHz 的

器件。注意，66MHz 型号是二倍频，100MHz 型号是三倍频。1990 年 Intel 公司为《计算机设计》杂志演示了 100MHz 型号（非二倍频）的 80486，但这个型号尚未发行。AMD 公司已经生产出 40MHz 型号的 486 微处理器，也有相应的 80MHz（二倍频）和 120MHz（三倍频）两种型号。80486 有 80486DX 和 80486SX 两类。这两类的惟一区别在于 80486SX 不包含算术协处理器，从而降低了价格。80487SX 算术协处理器可作为 80486SX 微处理器的一个独立部件使用。

本节详述 80486 和 80386 微处理器之间的区别，正如将要看到的，它们之间的区别很少，最显著的区别在于 cache 系统和奇偶校验生成器。

17.7.1 80486DX 和 80486SX 微处理器的引脚

图 17-29 给出了 80486DX 微处理器引脚图，一个 168 脚 PGA 封装的图。80486SX 引脚图也采用 168 引脚的 PGA 封装，它与 80486DX 之间只存在一些很小的差别，因此没有给出它的引脚图。请注意，80486DX 上 B_{15} 引脚作为 NMI 引脚，而在 80486SX 上 A_{15} 引脚作为 NMI 引脚。其余差别是，在 80486DX 上 A_{15} 引脚作为 $\overline{\text{IGNNE}}$（忽略数字错误）引脚，在 80486SX 上无此引脚；在 80486DX 上 C_{14} 引脚作为 $\overline{\text{FERR}}$（浮点错误）引脚，而在 80486SX 上 B_{15} 和 C_{14} 引脚没有连接。

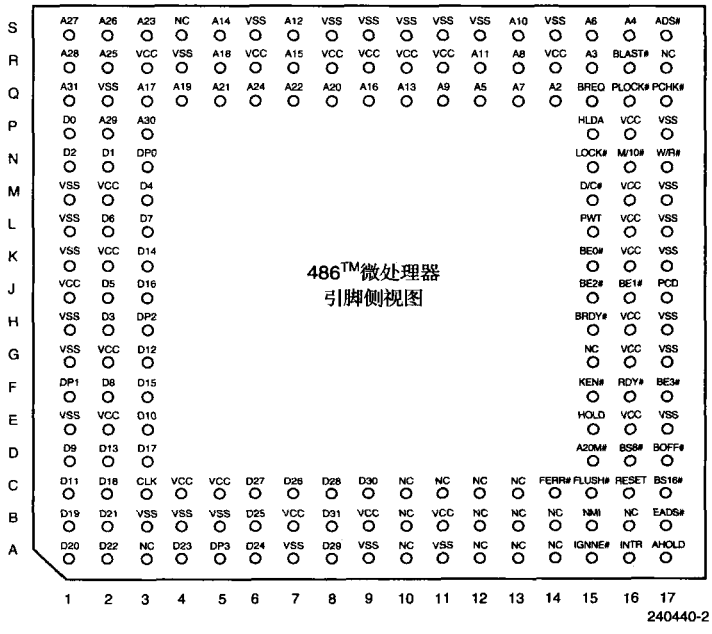


图 17-29 80486 的引脚图（由 Intel 公司提供）

在连接 80486 微处理器时，所有的 V_{CC} 和 V_{SS} 引脚必须正确地连接到电源上。电源必须提供 +5.0V ±10% 电压，在 33MHz 型号中浪涌电流可达 1.2A。33MHz 型号的平均电源电流为 650mA。Intel 公司也生产了 3.3V 的这种微处理器，在三倍频（即 100MHz）的速度下平均电流为 500mA。逻辑 0 输出最高电流为 4.0mA，逻辑 1 输出最高电流为 1.0mA。如果需要较大的电流，就必须对 80486 进行缓冲。图 17-30 给出了一个被缓冲的 80486 系统。在所示的电路图中，仅对地址、数据和奇偶校验信号进行了缓冲。

引脚定义

A_{31}～A_2 **地址输出（address output）**引脚。在正常运行过程中，A_{31}～A_2 提供存储器和 I/O 地址；在 cache 行无效期间，A_{31}～A_4 用于驱动微处理器。

$\overline{\text{A20M}}$ **地址位 20 屏蔽（address bit 20 mask）**引脚。该引脚可以使得 80486 将其地址放置在 000FFFFFH 到 00000000H 之间，就像 8086 微处理器一样。这使得存储系统可以像 8086 微处理器的 1MB 实模式存储系统一样工作。

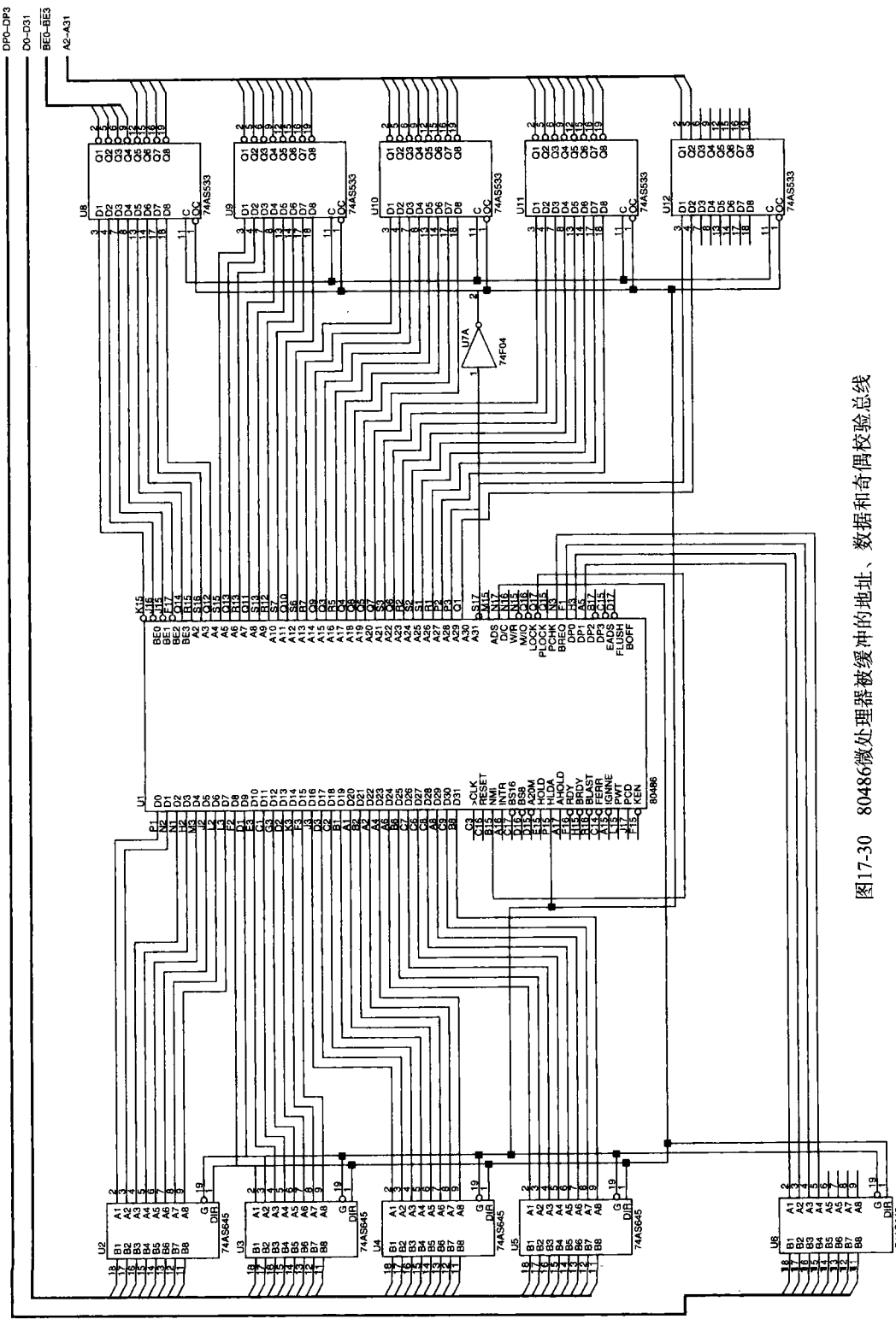


图17-30 80486微处理器被缓冲的地址、数据和奇偶校验总线

$\overline{ADS}$ **地址数据选通（address data strobe）**引脚。该引脚变为逻辑 0 时表示地址总线包含有效的存储器地址。

AHOLD **地址保持（address hold）**输入引脚。该引脚使得微处理器将其地址总线置于高阻态，而其他总线处于激活状态。该引脚常常被总线上的其他主控设备用来获得对无效 cache 周期的访问。

$\overline{BE3}$～$\overline{BE0}$ **字节使能（byte enable）**输出引脚。当微处理器与存储器或 I/O 空间传送信息时，这些引脚用来选择一个存储体。$\overline{BE3}$信号使能 D_{31}～D_{24}，$\overline{BE2}$使能 D_{23}～D_{16}，$\overline{BE1}$使能 D_{15}～D_8，$\overline{BE0}$使能 D_7～D_0。

$\overline{BLAST}$ **猝发持续（burst last）**输出引脚。用来表示当下一个$\overline{BRDY}$信号激活时猝发周期结束。

$\overline{BOFF}$ 总线**挂起（back-off）**输入引脚。该引脚可使微处理器在下一个时钟周期将其总线置于高阻态。微处理器一直处于总线挂起状态，直到$\overline{BOFF}$引脚被置为逻辑 1。

$\overline{BRDY}$ **猝发就绪（burst ready）**输入引脚。该引脚用于通知微处理器猝发周期已结束。

BREQ **总线请求（bus request）**输出引脚。该引脚用于表明 80486 微处理器产生了一个内部总线请求。

$\overline{BS8}$ **8 位总线宽（bus size 8）**输入引脚。该引脚可使 80486 用其数据总线中的 8 位来访问存储器或 I/O 组件中的字节单元。

$\overline{BS16}$ **16 位总线宽（bus size 16）**输入引脚。该引脚可使 80486 用其数据总线中的 16 位来访问存储器或 I/O 组件中的字节单元。

CLK **时钟（clock）**输入引脚。该引脚向 80486 提供基本时序信号。时钟输入信号与 TTL 电平兼容，80486 工作在 25MHz 时该信号频率为 25MHz。

D_{31}～D_0 数据总线（data bus）。用于在微处理器与存储器或 I/O 系统传送数据。在中断响应周期中，D_7～D_0 还用来接收中断向量类型号。

D/$\overline{C}$ **数据/控制（data/control）**输出引脚。该引脚用来表明当前操作是数据传送周期还是控制传送周期。参见表 17-3 中有关 D/$\overline{C}$、M/$\overline{IO}$及 W/$\overline{R}$ 的功能。

表 17-3 总线周期的鉴别

M/$\overline{IO}$	D/$\overline{C}$	W/$\overline{R}$	总线周期类型
0	0	0	中断响应
0	0	1	停机/特定
0	1	0	I/O 读
0	1	1	I/O 写
1	0	0	取操作码
1	0	1	保留
1	1	0	存储器读
1	1	1	存储器写

DP_3～DP_0 **数据奇偶校验（data parity）**输入/输出引脚。它们为写操作提供偶校验信息，为读操作提供奇偶校验检测。如果在读操作过程中发现奇偶校验错误，输出引脚$\overline{PCHK}$变为逻辑 0，指示出现了一个奇偶校验错误。如果系统中不使用奇偶校验，则必须将这些引脚上的电压上拉到 +5.0V；若系统使用 3.3V 的电源，则上拉到 +3.3V。

$\overline{EADS}$ **外部地址选通（external address strobe）**输入引脚。它和 AHOLD 引脚一起来表示正在使用外部地址，当前为 cache 无效周期。

$\overline{FERR}$ **浮点错误（floating-point error）**输出引脚。用来表明浮点协处理器发现一个错误状态。它用来保持 DOS 软件的兼容性。

$\overline{FLUSH}$ **清空（flush）** cache 输入引脚。它使微处理器将 8KB 的内部 cache 的内容清空。

HLDA **保持响应（hold acknowledge）**输出引脚。它表明微处理器正处于活跃状态，并且微处理器已将其总线置为高阻态。

HOLD **保持（hold）**输入引脚。该信号用来请求 DMA 操作，使地址总线、数据总线和控制总线被置为高阻态。并且该信号一旦被许可，就使 HLDA 变为逻辑 0。

$\overline{IGNNE}$ **忽略数字错误（ignore number error）**输入引脚。它使协处理器忽略浮点错误并继续处理数据。该信号不影响 FERR 引脚的状态。

INTR **中断请求**（**interrupt request**）输入引脚。该信号与 Intel 家族其他成员一样，用来请求可屏蔽中断。

$\overline{\text{KEN}}$ **cache 选通**（**cache enable**）输入引脚。它使得当前总线数据被存储在内部 cache 中。

$\overline{\text{LOCK}}$ **锁定**（**lock**）输出引脚。任何带锁定前缀的指令使其输出变为逻辑 0。

M/$\overline{\text{IO}}$ **存储器/IO**（**memory/IO**）引脚。它指出地址总线上是存储器地址还是 I/O 端口号。它还可与 W/$\overline{\text{R}}$ 信号一起来生成存储器与 I/O 读写控制信号。

NMI **非屏蔽中断**（**non-maskable interrupt**）输入引脚。它请求 2 号中断。

PCD **页高速缓冲器禁止**（**page cache disable**）输出引脚。它反映了页表项或页目录项中 PCD 属性位的状态。

$\overline{\text{PCHK}}$ **奇偶校验检查**（**parity check**）输出引脚。该信号有效时表明读操作过程中在 DP_3 ～ DP_0 引脚上发现奇偶校验错误。

$\overline{\text{PLOCK}}$ **伪锁**（**pseudo-lock**）输出引脚。它指明当前操作需要多于一个总线周期才能完成。在协处理器存取 64 或 80 位的存储器数据的操作中该信号变为逻辑 0。

PWT **页直写**（**page write through**）输出引脚。它指明页表或页目录项中 PWT 属性位的状态。

$\overline{\text{RDY}}$ **就绪**（**ready**）输入引脚。它指出非猝发总线周期已结束。RDY 信号必须被返回，否则微处理器将不断往时序中插入等待状态，直到 RDY 被断定有效。

RESET **复位**（**reset**）输入引脚。与 Intel 家族中的其他成员一样，该信号用来初始化 80486。表 17-4 列出了 80486 微处理器中的 RESET 输入引脚的影响。

W/$\overline{\text{R}}$ **写/读**（**write/read**）引脚。它指出当前总线周期是读周期还是写周期。

表 17-4 RESET 后微处理器的状态

寄 存 器	带自检的初始值	不带自检的初始值
EAX	00000000H	?
EDX	00000400H + ID①	00000400H + ID①
EFLAGS	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H
ES	0000H	0000H
CS	F000H	F000H
DS	0000H	0000H
SS	0000H	0000H
FS	0000H	0000H
GS	0000H	0000H
IDTR	Base = 0, limit = 3FFH	Base = 0, limit = 3FFH
CR_0	60000010H	60000010H
DR_7	00000000H	00000000H

① 由 Intel 提供的微处理修正 ID 号。

17.7.2 80486 的基本结构

80486DX 的结构几乎与 80386 的完全一样。在 80386 结构的基础上，80486DX 增加了一个算术协处理器和一个 8KB 的一级 cache。80486SX 几乎与一个有 8KB 的 cache 并且不带协处理器的 80386 一样。

扩展标志寄存器（EFLAG）如图 17-31 所示。为了与 X86 系列中其他微处理器兼容，最右边的标志位实现相同的功能。惟一的新标志位是 **AC**（***alignment check***，**地址对齐检测**）位，用于指示微处理器已存取一个位于奇地址的字或一个位于非双字边界的双字。软件的高效运行要求数据存储在字或双字的边界上。

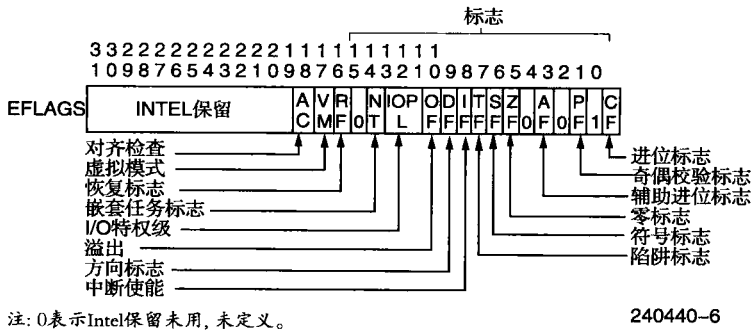


图 17-31 80486 的 EFLAG 寄存器（由 Intel 公司提供）

17.7.3 80486 的存储系统

80486 的存储系统与 80386 的存储系统相同。80486 的存储空间为 4GB，起始于 00000000H，结束于 FFFFFFFFH。80486 存储系统的主要改变在其内部的 8KB 高速缓冲存储器（cache），它加速指令执行和数据读取的速度。另外，80486 微处理器内置了奇偶校验器/奇偶发生器。

奇偶校验器/奇偶发生器

奇偶校验通常用来确定从存储单元读取的数据是否正确。为了方便，Intel 在 80486 微处理器内部包含了奇偶校验发生器/检测器。80486 在每个写周期中产生奇偶校验。所产生的校验是偶校验，为每个内存字节都提供了校验位。奇偶校验的校验位出现在 $DP_0 \sim DP_3$ 引脚上，它们既作为输入又作为输出。通常，在每个写周期中这些校验位被存储到存储器中；在每个读周期中从存储器中读取校验信息。

在读周期中，微处理器检查奇偶，如果发现奇偶校验错误，就会在$\overline{\text{PCHK}}$引脚上产生奇偶校验错误信号。除非用户将$\overline{\text{PCHK}}$信号用作一个中断输入，否则奇偶校验错误不会对数据处理引起任何变化。在基于 DOS 的系统中，常常用中断来通知奇偶校验错误。图 17-32 说明了 80486 存储系统的组织结构，包括了奇偶校验位的存储。注意，除了奇偶校验位的存储以外，它的组织结构与 80386 相同。如果不用奇偶校验，Intel 建议将 $DP_0 \sim DP_3$ 引脚上拉到 +5.0V。

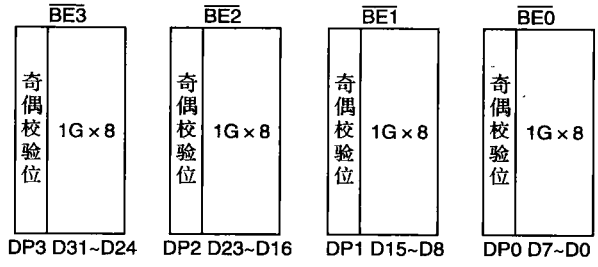


图 17-32 80486 存储器及奇偶校验位的组织结构（由 Intel 公司提供）

cache 存储器

cache 存储系统用于高速缓冲存储程序即将使用的数据或即将执行的指令。cache 采用 4 路成组联想式组织结构，每个单元（行）包含 16 个字节或 4 个双字的数据信息，其操作采用直写（write through）方式。注意，只有在未命中时，cache 中的内容才发生变化。这意味着不先写缓存写入到存储单元的数据是不写入 cache 中的。在很多情况下，程序活动部分完全位于 cache 中。这使得程序中许多常用的指令可以在一个时钟周期内执行完。使这些高效的指令执行速度降低的惟一情形是当微处理器必须填充高速缓冲中的行。数据也被存储在 cache 之中，但这对程序的执行速度影响不大，因为程序中的许多部分并不是不停地存取数据。

控制寄存器 0（CR_0）用两个 80386 微处理器中所没有的位来控制 cache（有关 80486 微处理器中的 CR_0 寄存器，参见图 17-33）。这两个新的控制位是 CD（cache 关闭）位和 NW（非 cache 直写存储器）位，它们用于控制 8KB 的 cache。如果 CD 位为逻辑 1，所有的 cache 操作都被禁止。只有在软件调试时才对 CD 位进行置位，通常它都被清零。NW 位用来禁止 cache 直写存储器操作。只有在测试过

程中才用 CD 位禁止 cache 直写存储器操作。正常程序执行过程中 CD 位和 NW 位的值均为 0。

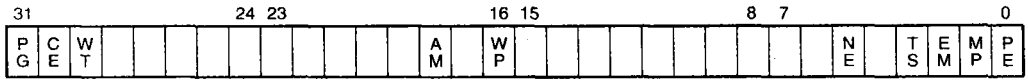


图 17-33 80486 微处理器控制寄存器 0（CR0）

因为 cache 是 80486 微处理器中新出现的，而且它又是利用了 80386 中所没有的猝发周期来填充，所以有必要详细了解总线填充周期。当总线行被填充时，80486 必须从存储系统中读取 4 个 32 位数字以填充 cache 中的一行。填充过程占用一个猝发周期。猝发周期是一个特殊的存储周期，共包含五个时钟周期，在猝发周期中，从存储系统中读出四个 32 位数字。这里，假定存储器速度足够快，不需要等待状态。如果 80486 的时钟频率为 33MHz，我们可以在 167ns 内填充一个 cache 行，鉴于普通的、非猝发的 32 位存储器读操作需要两个时钟周期，因此这种方式非常高效。

存储器读时序

图 17-34 显示了 80486 非猝发存储器操作的读时序。注意，这里用两个时钟周期传送数据。在时钟周期 T_1 中产生存储器地址和控制信号。在时钟周期 T_2 中，微处理器和存储器间传送数据。注意，$\overline{RDY}$位必须变为逻辑 0 使数据被传送并结束总线周期。非猝发访问的时间为两个时钟周期减去使地址信号出现在地址总线上所需的时间及数据总线的建立时间。对以 20MHz 运行的 80486，两个时钟周期为 100ns，减去 28ns 的地址建立时间及 6ns 的数据建立时间，就得出非猝发存取时间为 100ns－34ns＝76ns。当然，如果包括译码时间和延迟时间，无等待存取时间就更短了。而且系统中使用的是更高频率的 80486，存取时间还要短。

33MHz、66MHz 和 100MHz 的 80486 微处理器都以 33MHz 的速率访问总线上的数据。换句话说，微处理器可能工作在 100MHz 状态下，而系统总线却工作在 33MHz 状态下。请注意，33MHz 系统总线的非猝发存取时间为 60ns－24ns＝36ns。显然，如果使用标准 DRAM 存储设备则要插入等待状态。

图 17-35 说明了利用猝发方式用 4 个 32 位数填充 cache 行的时序图。注意，地址（A_{31}～A_4）在时钟周期 T_1 中出现并在整个猝发周期中保持不变。还要注意的是 A_2 和 A_3 在第一个 T_2 之后，在每个 T_2 中都发生变化来寻址存储系统中 4 个连续的 32 位数字。正如以前提到的，使用猝发方式填充 cache 行仅需 5 个时钟周期（1 个 T_1 和 4 个 T_2）。假定系统中没有延迟，20MHz 的 80486 填充第二个及其后的双字的存取时间为 50ns－28ns－5ns＝17ns。要实现猝发方式的传送，必须使用高速的存储器。因为 DRAM 存储器的存取时间最快为 40ns，我们只好使用 SRAM 来进行猝发方式传送。33MHz 系统允许第二个及其后的双字的存取时间为 30ns－19ns－5ns＝6ns。如果用一个外部计数器来取代地址位 A_2 和 A_3，则无须 19ns 的地址建立时间，于是存取时间变为 30ns－5ns＝25ns，即使将最慢的 SRAM 插入系统作为 cache 也足够用了。如果在系统中 SRAM 用作 cache，我们通常称其为同步猝发方式 cache。注意，$\overline{BRDY}$引脚确认猝发方式传送的结束，而不是确认普通方式传送结束的$\overline{RDY}$引脚。

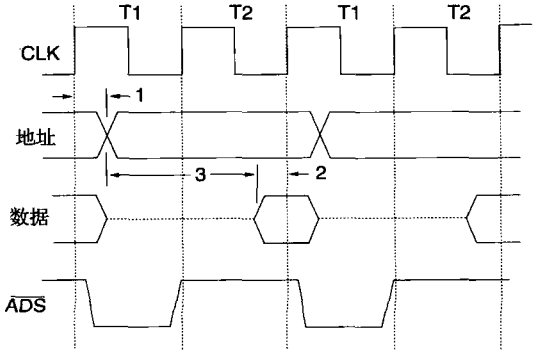


图 17-34 80486 微处理器非猝发读时序

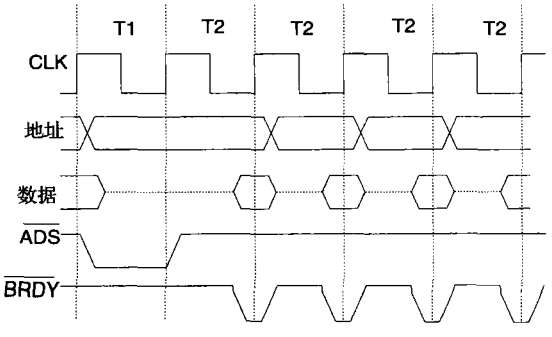


图 17-35 猝发周期在 5 个时钟周期中读入 4 个双字

PWT 位用来控制外部 cache 写操作中的 cache 工作方式，不能控制内部 cache 的写操作。它的逻辑值可以在 80486 微处理器的 PWT 引脚上找到。该信号可用来指定外部 cache 的直写策略。

PCD 位用于控制片内 cache。如果 PCD = 0，片内 cache 可用于当前页存储页。注意，如果要使能 cache，80386 页表项中的 PCD 位必须为逻辑 0。如果 PCD = 1，片内 cache 被禁止。高速缓存的禁止不受$\overline{\text{KEN}}$、CD 和 NW 条件的影响。

17.8 小结

1）80386 微处理器是 80286 微处理器的增强版，它有增强的存储管理单元（MMU），提供内存分页。80386 含有 32 位的扩展寄存器以及 32 位的数据总线和地址总线。80386SX 是 80386DX 的一个压缩版，具有 16 位数据总线和 24 位地址总线。80386EX 是一个具有完全 AT 风格的单片式 PC 机。

2）80386 的物理存储空间是 4GB，可以访问 64TB 的虚拟存储空间。80386 的存储器数据宽度是 32 位，可以按字节、字、双字访问。

3）当 80386 工作在流水线方式时，可以在当前指令执行完成之前送出下一个指令或存储器数据的地址。这样使存储系统在当前指令完成之前就可以预取下一条指令或数据。增加了存取时间，从而降低了存储速度。

4）cache 系统是高速半导体存储器，可以存放那些要频繁读取的数据，从而减少了对这些数据的访问时间。数据写入内存的同时也写入 cache 中，因此，最新的数据总在 cache 中。

5）80386 在保护模式下通过存储在 TSS 中的 I/O 位图保护表可以禁止 I/O，除此之外 80386 的 I/O 结构几乎与 80286 相同。

6）80386 微处理器扩充了中断系统，包括在中断向量表中增加了附加的预定义中断。这些中断用于存储管理系统。

7）80386 的存储管理与 80286 的类似，只是 MMU 产生的物理地址不是 24 位宽而是 32 位宽。80386 的 MMU 还支持分页。

8）80386 被复位时运行在实模式（8086 模式）下。实模式允许处理器访问存储器前第一个 1MB 的数据。在保护模式下，80386 可寻址 4GB 的物理地址空间。

9）描述符是一个 8 字节的串，用来指定 80386 如何使用代码段和数据段。描述符由存储在段寄存器中的选择子来选择。描述符仅用在保护模式下。

10）存储管理是通过存储在描述符表中的一系列描述符来实现的。为了便于存储管理，80386 使用了三个描述符表：全局描述符表（GDT）、局部描述符表（LDT）以及中断描述符表（IDT）。GDT 和 LDT 每个最多可包含 8192 个描述符，IDT 最多可包含 256 个描述符。GDT 和 LDT 描述代码段和数据段以及任务。IDT 通过中断门描述符描述 256 个不同的中断级。

11）TSS（任务状态段）含有当前任务以及前一个任务的信息。TSS 的结尾附加了 I/O 位图保护，可禁止被选中的 I/O端口地址。

12）存储分页机制允许将任意 4KB 的物理存储页映射到任意 4KB 的线性存储页。例如，通过分页机制可以将物理地址为 00A00000H 的存储单元映射到线性地址 A0000000H 上。页目录和页表用来将物理地址分配给线性地址。分页机制用于保护模式和虚拟模式。

13）80486 微处理器是 80386 的改进型，它含有一个 8KB 的 cache 及一个 80387 算术协处理器；它的许多指令可以在一个时钟周期内完成。

14）80486 可执行一些新指令。这些指令控制内部 cache，可实现交换加（XADD）、比较交换（CMPXCHG）和字节交换（BSWAP）等功能。除了这些少数新增加的指令外，80486 是 100% 向上兼容 80386 及 80387 的。

15）80486 新增了内置自检测（build-in self-test，BIST）功能，在复位时可以对微处理器、协处理器及 cache 进行检测。如果 80486 通过了检测，EAX 寄存器应该为零。

17.9 习题

1. 80386 微处理器在保护模式下可寻址_____字节的存储器。
2. 80386 微处理器通过存储管理单元（MMU）可以寻址_____字节的虚拟存储器。
3. 描述 80386DX 和 80386SX 之间的区别。
4. 画出 80386 工作在以下模式时的存储器映射图：（a）保护模式；（b）实模式。
5. 80386 各输出引脚上的电流有多大？将这些电流与 8086 微处理器的输出电流进行比较。
6. 描述 80386 的存储系统并解释体选择信号的用途及工作原理。
7. 解释 80386 硬件复位时地址总线接线的操作。

8. 解释在基于80386微处理器的系统中，流水线是如何将存储器访问时间延长了许多。
9. 简单描述cache系统是如何工作的。
10. 80386中的I/O端口地址起始于I/O地址_____并扩展到I/O地址_____。
11. 80386和80387之间用什么I/O端口进行数据通信。
12. 将80386的存储器的地址连接与早期的微处理器的连接进行比较。
13. 如果80386要运行在20MHz下，那么CLK_2引脚上的时钟频率为多少？
14. 80386微处理器中的$\overline{BS16}$引脚有何用途？
15. 说明80386的每个控制寄存器（CR_0、CR_1、CR_2和CR_3）的作用。
16. 说明80386的每个调试寄存器的作用。
17. 调试寄存器引发哪一级别的中断？
18. 什么是测试寄存器。
19. 选择一条可以把控制寄存器0复制到EAX的指令。
20. 描述CR_0中PE位的用途。
21. 设计一条指令，使其能够访问FS段由DI寄存器间接寻址的内存单元的数据，指令应能将EAX中的内容存储到该单元中。
22. 什么是比例变址寻址？
23. 下面的指令是否合法？MOV AX，[EBX + ECX]。
24. 解释下面的指令如何计算存储器地址：
    (a) ADD [EBX + 8 * ECX], AL
    (b) MOV DATA [EAX + EBX], CX
    (c) SUB EAX, DATA
    (d) MOV ECX, [EBX]
25. 7号中断类型的作用是什么？
26. 保护特权冲突时哪个中断向量类型被激活？
27. 什么是双中断错误？
28. 在保护模式下发生中断时，中断向量是怎样定义的？
29. 什么是描述符？
30. 什么是选择子？
31. 选择子如何选择局部描述符表？
32. 哪个寄存器用来寻址全局描述符表？
33. GDT中可存储多少个全局描述符？
34. 解释当物理存储器只有4GB时，80386如何访问64TB的虚拟存储空间？
35. 段描述符和系统描述符的区别是什么？
36. 什么是任务状态段（TSS）？
37. 如何寻址TSS？
38. 描述80386如何从实模式切换到保护模式？
39. 描述80386如何从保护模式切换到实模式？
40. 什么是80386微处理器的虚拟8086模式操作？
41. 80386如何寻址分页目录？
42. 存储页中有多少字节？
43. 说明如何利用80386的分页单元将线性地址D0000000H分配到物理存储地址C0000000H。
44. 80386和80486微处理器之间有什么区别？
45. 80486微处理器中$\overline{FLUSH}$输入引脚有何作用？
46. 将80386的寄存器组与80486微处理器的进行比较。
47. 与80386微处理器相比，80486中的标志有哪些不同？
48. 80486微处理器中哪些引脚用作奇偶校验？
49. 80486微处理器中采用_____校验。
50. 80486微处理器的内部cache是_____KB。
51. 通过从存储系统中读_____字节来填充cache行。
52. 什么是80486的猝发方式？
53. 定义术语“cache直写”（cache write-through）。
54. 什么是BIST？

第 18 章　Pentium 和 Pentium Pro 微处理器

引言

Pentium 微处理器对 80486 微处理器的体系结构进行了改进。这些改进包括优化的高速缓存结构、更宽的数据总线、更快的算术协处理器、双整型处理器以及分支预测逻辑。高速缓存被重新组织成两个，每个 8KB，一个用于数据缓存，另一个用于指令缓存。数据总线宽度从 32 位增加到 64 位。算术协处理器的速度则比 80486 的协处理器快 5 倍。双整型处理器通常允许每个时钟周期执行两条指令。最后，分支预测逻辑使分支程序执行更加有效。请注意，这些变化是 Pentium 内部的，这使得 Pentium 的软件与早期的 Intel 80X86 微处理器向上兼容。Pentium 新的改进是增加了 MMX 指令。

Pentium Pro 是一种比 Pentium 更快的微处理器，它具有改进的内部体系结构，能够调度 5 条指令执行，并具有更快的浮点运算单元。除了 16KB 的一级高速缓存（8KB 用于数据缓存，8KB 用于指令缓存）外，Pentium Pro 还包含 256KB 或 512KB 的二级高速缓存。Pentium Pro 含有纠错电路（ECC），可以纠正 1 位错误，识别两位错误。Pentium Pro 还增加了四条地址线，这使它能够访问惊人的直接可寻址的 64GB 存储空间。

目的

读者学习完本章后将能够：

1）对比 Pentium、Pentium Pro 与 80386 和 80486 微处理器。
2）描述 64 位宽的 Pentium 存储系统的组织和接口及其变化。
3）相对于 80386 和 80486 微处理器，对比存储管理单元和分页单元的变化。
4）详述 Pentium 微处理器新增加的指令。
5）解释用超标量双整型单元如何改进 Pentium 微处理器的性能。
6）描述分支预测逻辑的操作。
7）详述 Pentium Pro 相对于 Pentium 的改进。
8）解释 Pentium Pro 的动态执行体系结构如何运行。

18.1　Pentium 微处理器简介

在将 Pentium 处理器或其他微处理器用于系统之前，必须了解每个引脚的功能。本章这一节详述了每个引脚的功能以及 Pentium 微处理器的外部存储系统和 I/O 结构。

图 18-1 列出了 237 脚 PGA（pin grid array）封装的 Pentium 微处理器的外部引脚。Pentium 微处理器有两个版本：全功能型 Pentium 和称为 Pentium OverDrive 的 P24T 型。P24T 型具有 32 位的数据总线，可以兼容地插到有 P24T 插座的 80486 机器中。P24T 型 Pentium 有一个风扇。与早期 80486 微处理器相比，Pentium 引脚的最大特点在于它具有 64 个数据总线连接点而不是 32 个，这就允许有更大的物理空间。

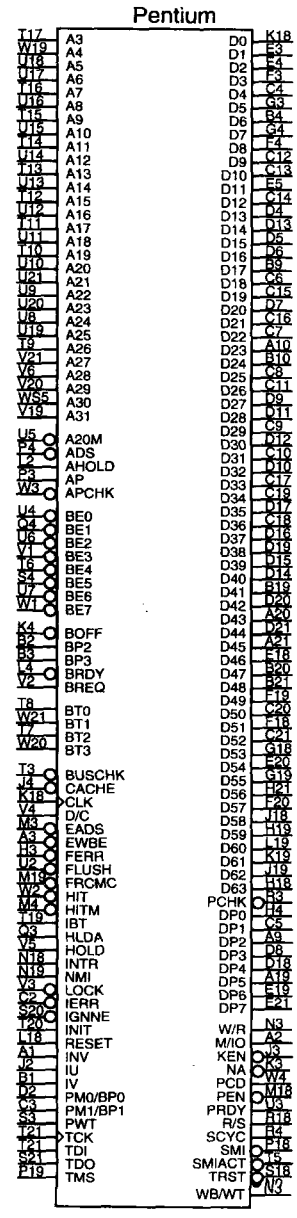


图 18-1　Pentium 微处理器的外部引脚

与早期的 Intel 微处理器家族的成员一样，Pentium 的早期型号需要 +5.0V 单电源，对于 66MHz 的 Pentium，电源电流的平均值为 3.3A，对于 60MHz 的 Pentium 则为 2.91A。由于电流较大，所以这些微处理器的功耗也很大：66MHz 的 Pentium 是 13W，而 60MHz 的 Pentium 是 11.9W。当前型号的 Pentium（90MHz 及以上的）都采用 3.3V 电源供电，以减少电流消耗。目前，Pentium 需要一个具有良好通风的散热器来保证 Pentium 芯片不会过热。Pentium 具有多个 Vcc 和 Vss 连接点，为了正常运行，这些接点必须都接到对应的 +5.0V 或 +3.3V 电源上和地上。某些标记为 N/C（不连）的引脚必须不连接。新型 Pentium 在降低功耗方面做了改进，例如，233MHz 的 Pentium 只需要 3.4A 的电流，仅比 66MHz 的 Pentium 所需的 3.3A 略微大了一点。

每个 Pentium 的输出引脚在逻辑 0 时提供 4.0mA 电流，在逻辑 1 时提供 2.0mA 电流。这表示 Pentium 的驱动电流与早期的 8086、8088 和 80286 上 2.0mA 的驱动电流相比增加了。每个 Pentium 输入引脚仅需要 15μA 的负载电流。在一些系统中（除了最小系统）这些电流需要总线缓冲。

每个 Pentium 引脚组的功能如下：

$\overline{A20}$ **地址 A20 屏蔽（address A20 mask）**输入引脚。该引脚用于在实模式中通知 Pentium 进行地址回绕，就像在 8086 微处理器中一样，该引脚供 HIMEM.SYS 驱动程序使用。

A_{31}～A_3 **地址总线（address bus）**连接引脚。它们用于寻址 Pentium 存储系统中任何 512K×64 存储空间，注意 A_0、A_1 和 A_2 在总线使能（$\overline{BE7}$～$\overline{BE0}$）中被编码用于选择 64 位宽存储单元中的任意或全部 8 个字节。

$\overline{ADS}$ **地址数据选通（address data strobe）**信号。当 Pentium 发出一个有效的存储器地址或 I/O 地址，该信号就变为有效。这个信号与 W/$\overline{R}$ 和 M/$\overline{IO}$信号一起产生早期基于 8086～80286 微处理器的系统中出现过的独立的读写信号。

$\overline{AHOLD}$ **地址保持（address hold）**输入引脚。该引脚用于使 Pentium 为下一个时钟周期保持地址和 AP 信号。

AP **地址校验（address parity）**引脚。该引脚为所有的 Pentium 存储和 I/O 传送器提供偶校验。在整个查询周期中 AP 引脚必须在与$\overline{EADS}$信号相同的时钟周期内由偶校验信息驱动。

$\overline{APCHK}$ **地址校验检查（address parity check）**引脚。当 Pentium 检查到地址校验错时该信号变为逻辑 0。

$\overline{BE_7}$～$\overline{BE_0}$ **体使能信号（bank enable signal）**。该信号用于选择访问单字节、字、双字或四字数据。这些信号在微处理器内由地址 A_0、A_1 和 A_2 产生。

$\overline{BOFF}$ **Back-off** 输入信号。该信号用来中止所有未完成的总线周期，并使 Pentium 的总线悬浮直到$\overline{BOFF}$为负，当$\overline{BOFF}$为负后，Pentium 就重新启动所有中止的总线周期。

BP_3～BP_0 **断点（breakpoint）**引脚。当调试寄存器被编程来监测匹配时，断点引脚组 BP_3～BP_0 用来指示断点匹配。

PM_1～PM_0 **性能监控（performance monitoring）**引脚。PM_1 和 PM_0 用以指示调试模式控制寄存器的性能监控位的设置。

$\overline{BRDY}$ **猝发就绪（burst ready）**信号。该信号通知 Pentium 处理器外部系统已从数据总线连接中取得数据。此信号可用于向 Pentium 时序中插入等待状态。

BREQ **总线请求（bus request）**输出信号。该信号指示 Pentium 已产生了一个总线请求。

BT_3～BT_0 **分支跟踪（branch trace）**输出引脚。提供分支目标的线性地址的 2～0 位，并在 BT_3 上提供默认操作数长度。这些输出信号在分支跟踪特殊消息周期中有效。

$\overline{BUSCHK}$ **总线检查（bus check）**输入引脚。它允许系统向 Pentium 发送信号告知总线传送失败。

$\overline{CACHE}$ **cache** 输出信号。该信号指示当前 Pentium 周期可对数据进行缓存。

CLK **时钟（clock）**引脚。它由具有当前 Pentium 工作频率的时钟信号驱动，例如，为了使 Pentium 工作于 66MHz，我们将 66MHz 时钟加于此引脚。

D_{63}～D_0 **数据总线（data bus）**连接。在微处理器与内存和 I/O 系统间进行字节、字、双字和四字

数据的传送。

$\overline{D/C}$ **数据/控制（Data/Control）**信号。为逻辑1时表明数据总线上含有来自存储器或I/O或写往存储器或I/O的数据。如果$\overline{D/C}$为逻辑0，则表明微处理器或者处于停机状态或者正在执行一个中断请求。

$DP_7 \sim DP_0$ **数据奇偶校验（data parity）**信号。由Pentium产生并通过这些连接点检查8个存储体的数据。

$\overline{EADS}$ **外部地址选通（external address strobe）**输入信号。该信号指示地址总线包含一个请求周期的地址。

$\overline{EWBE}$ **外部写缓存空（external write buffer empty）**输入信号。该信号指示写周期在外部系统中处于挂起状态。

$\overline{FERR}$ **浮点错（floating-point error）**信号。该信号与80386中的$\overline{ERROR}$信号兼容，用来指示内部协处理器出现错误。

$\overline{FLUSH}$ **清cache（flush cache）**输入信号。该信号使cache清除所有回写行并使内部cache无效。如果在进行复位时$\overline{FLUSH}$为0，则Pentium进入cache测试模式。

$\overline{FRCMC}$ **功能性冗余检查（functional redundancy check）**信号。在复位时该信号设置Pentium为主模式（0）或检查模式（1）。

$\overline{HIT}$ **命中（hit）**信号。该信号表明在查询方式中内部cache包含了有效数据。

$\overline{HITM}$ **命中修改（hit modified）**信号表明在查询周期中发现了一个修改过的cache行，此输出信号用于在已修改过的cache行回写到存储器之前禁止其他单元访问数据。

HOLD **保持（hold）**信号。该信号请求一个DMA操作。

HLDA **保持响应（hold acknowledge）**信号。该信号指示Pentium当前处于保持状态。

IBT **指令分支采用（instruction branch taken）**信号。该信号表示Pentium采用了一个指令分支。

$\overline{IERR}$ **内部错（internal error）**输出信号。该信号表明Pentium已检测到一个内部的奇偶校验错或功能性冗余错。

$\overline{IGNNE}$ **忽略数字错（ignore numeric error）**输入信号。该信号使Pentium忽略协处理器错误。

INIT **初始化（initialization）**输出信号。该信号执行不初始化cache、回写缓冲区和浮点寄存器的复位操作，该信号不能取代加电后的RESET信号对处理器的复位。

INTR **中断请求（interrupt request）**信号。外部电路用该信号进行中断请求。

INV **无效（invalidation）**输入信号。该信号决定一个查询后的cache行状态。

IU **U-管道指令完成（U-pipe instruction complete）**输出信号。该信号表明U-管道中的指令已完成。

IV **V-管道指令完成（V-pipe instruction complete）**输出信号。该信号表明V-管道中的指令已完成。

$\overline{KEN}$ **cache使能（cache enable）**输入信号。使能内部高速缓存。

$\overline{LOCK}$ **锁定（LOCK）**信号。指令带有LOCK：前缀时该信号变为逻辑0。该信号通常用在DMA访问中。

M/$\overline{IO}$ **存储器/IO（memory/IO）**信号。该信号为逻辑1时选择存储器设备，而在为逻辑0时选择I/O设备，I/O操作过程中，地址总线在$A_{15} \sim A_3$上包含16位的I/O地址。

$\overline{NA}$ **下一地址（next address）**信号。该信号表明外部存储系统已准备好接收新的总线周期。

NMI **非屏蔽中断（non-maskable interrupt）**信号。该信号请求一个非屏蔽中断，这与早期版本的微处理器相同。

PCD **页缓存禁止（page cache disable）**输出信号。该信号通过反射CR_3的PCD位的状态来表

示内部的页缓存被禁止。

$\overline{\text{PCHK}}$ **奇偶校验检查（parity check）**输出信号。该信号表明从存储器或 I/O 读数据时出现奇偶校验检查错误。

$\overline{\text{PEN}}$ **奇偶校验使能（parity enable）**输入信号。该信号使能机器检查中断或异常。

PRDY **探针就绪（probe ready）**输出信号。该信号表明已为调试准备好探针方式。

PWT **页直写（page write-through）**输出信号。该信号显示 CR_3 上 PWT 位的状态。

R/S 此引脚与 Intel 调试端口一起使用来产生一个中断。

RESET **复位（reset）**信号。该信号初始化 Pentium，使其从内存 FFFFFFF0H 处开始执行软件，Pentium 被复位为实模式，最左边的 12 条地址线保持逻辑 1（FFFH），直至执行一个远程调用或远跳转。这使它与早期微处理器兼容。硬件复位后 Pentium 的状态请参见表 18-1。

表 18-1 RESET 后的 Pentium 状态

寄 存 器	RESET 值	RESET + BIST 值
EAX	0	0（如果测试成功）
EDX	0500XXXXH	0500XXXXH
EBX、ECX、ESP、EBP、ESI 和 EDI	0	0
EFLAGS	2	2
EIP	0000FFF0H	0000FFF0H
CS	F000H	F000H
DS、ES、FS、GS 和 SS	0	0
GDTR 和 TSS	0	0
CR_0	60000010H	60000010H
CR_2、CR_3 和 CR_4	0	0
$DR_0 \sim DR_3$	0	0
DR_6	FFFF0FF0H	FFFF0FF0H
DR_7	00000400H	00000400H

SCYC **分割周期（split cycle）**输出信号。该信号指示一个未对准的（misaligned）锁定的总线周期。

$\overline{\text{SMI}}$ **系统管理中断（system management interrupt）**输入信号。该信号使 Pentium 进入系统管理运行模式。

$\overline{\text{SMIACT}}$ **系统管理中断激活（system management interrupt active）**输出信号。该信号表明 Pentium 正工作在系统管理模式。

TCK **可测试性时钟（testability clock）**输出信号。根据 IEEE 1149.1 边界检测接口选择时钟操作。

TDI **测试数据输入（test data input）**信号。该信号用来测试由 TCK 信号输入 Pentium 的数据。

TDO **测试数据输出（test data output）**信号。该信号用来获得由 TCK 移出的 Pentium 的测试数据和指令。

TMS **测试方式选择（test mode select）**输入信号。该信号在测试模式中控制 Pentium 操作。

$\overline{\text{TRST}}$ **测试复位（test reset）**输入信号。该信号使测试模式被复位。

W/$\overline{\text{R}}$ **写/读（write/read）**表明当前总线周期在逻辑 1 时为写，或在逻辑 0 时为读。

WB/$\overline{\text{WT}}$ **回写/直写（write-back/write-through）**信号。该信号为 Pentium 数据 cache 选择相应操作。

18.1.1 存储系统

Pentium 微处理器的存储系统大小为 4GB，与 80386DX 和 80486 微处理器的一样。它们之间的差别在于存储器数据总线的宽度。Pentium 使用 64 位数据总线来寻址 8 个存储体，每个存储体包含 512MB 的数据，Pentium 物理存储系统的组织如图 18-2 所示。

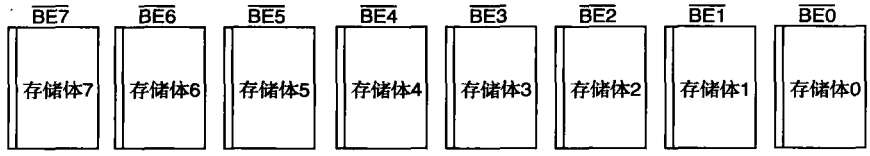


图 18-2 Pentium 微处理器的 8 字节宽存储体

Pentium 存储系统被分为 8 个存储体，每个存储体都有一个检验位，每 8 个存储体就可以用一个字节存放校验位。与 80486 一样，Pentium 采用内部校验发生和检查逻辑来获得存储系统的数据总线信息（注意，多数 Pentium 系统不使用校验检查，因为 ECC 是可用的）。64 位宽的存储器对于双精度浮点型数据是很重要的，因为双精度浮点型数据正好是 64 位宽。因为数据总线变为 64 位宽，Pentium 可以在一个读周期里得到浮点数据。这使得 Pentium 比 80486 的吞吐量更高。与早期的微处理器相似，Pentium 存储系统也是以字节方式从 00000000H 到 FFFFFFFFH 计数的。

存储器选择由体允许信号（$\overline{BE7}$～$\overline{BE0}$）来完成，这些单独的存储器体使 Pentium 在一个存储器传送周期中可以存取单个字节、字、双字或四字的数据。与早期的存储器选择逻辑一样，通常产生 8 个独立的写脉冲向存储器中写数据。

Pentium 所添加的一个新特性是能够在特定操作中为地址总线（A_{31}～A_5）检查和产生奇偶校验。AP 引脚为系统提供奇偶校验信息，$\overline{APCHK}$指示地址总线出现一个错误的奇偶校验检查。当检测到一个地址奇偶校验错误时，Pentium 并不采取任何措施，此错误必须由系统获得，如果需要可由系统采取适当措施处理（例如中断）。

18.1.2 输入/输出系统

Pentium 的 I/O 系统完全与早期的 Intel 微处理器兼容。I/O 端口号出现在地址线 A_{15}～A_3，和体使能信号一起选择实际用于 I/O 传送的存储体。

从 80386 微处理器开始，当 Pentium 在保护模式下操作时，I/O 特权信息被添加到 TSS 段，注意，这使得 I/O 端口可以被有选择地禁止。如果一个锁定的 I/O 地址被访问，Pentium 就产生一个 13 号中断来指示 I/O 特权冲突。

18.1.3 系统时序

和所有的微处理器一样，在与微处理器连接时我们必须了解系统时序信号。这里通过时序图详述了 Pentium 的操作，并说明了如何确定存储器的访问时间。

基本的 Pentium 非流水线存储器周期包括两个时钟周期：T_1 和 T_2。基本的非流水线读周期参见图 18-3。从时序图中可以看出，66MHz 的 Pentium 每秒可完成 3300 万次存储器传送。这里我们假定存储器可以此速度进行操作。

同样从时序图可以注意到，如果在时钟周期的上升沿（T_1 末端），$\overline{ADS}$为逻辑 0 时，W/$\overline{R}$ 信号变为有效，必须用该时钟确定是读周期还是写周期。

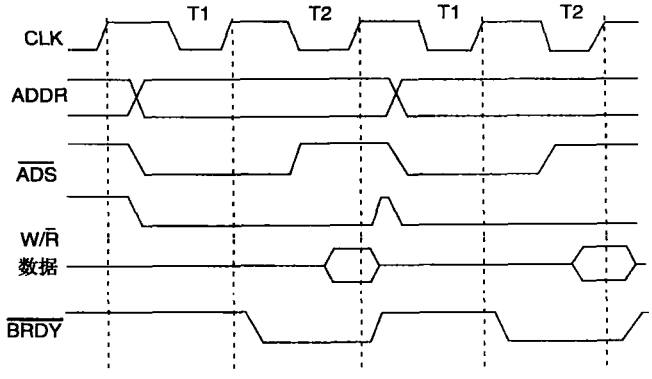


图 18-3 Pentium 微处理器的非流水线读周期

在 T_1 周期，微处理器发出$\overline{ADS}$、W/$\overline{R}$、地址和 M/$\overline{IO}$信号。为了确定 W/$\overline{R}$ 信号并产生正确的$\overline{MRDC}$和$\overline{MWTC}$信号，我们采用触发器来产生 W/$\overline{R}$ 信号，然后使用二选一的多路器来产生存储器和 I/O 控制信号。为 Pentium 微处理器产生存储器

和 I/O 控制信号的电路参见图 18-4。

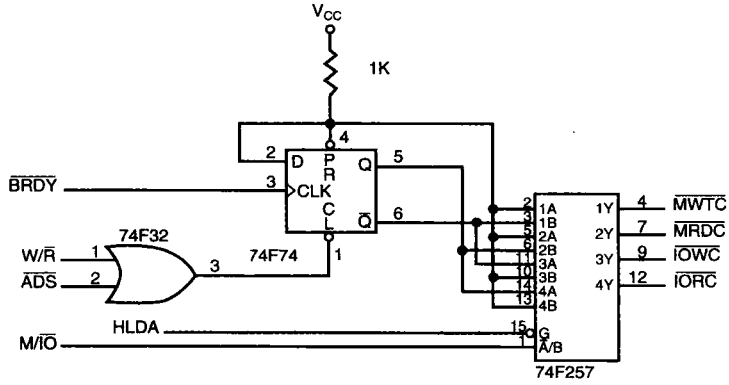


图 18-4 用于产生存储器和I/O控制信号的电路

在 T_2 周期，数据总线在 T_2 末端时钟上升沿被同步采样。时钟之前的建立时间是 3.8ns，而在时钟后的保持时间是 2.0ns，这意味着在此时钟边缘有 5.8ns 的数据窗口。在 T_1 开始后最多 8ns 后地址信号出现，这就是说，66MHz 的 Pentium 允许的访问时间为 30.3ns（两个时钟周期）减去 8.0ns 的地址延迟再减去 3.8ns 的数据准备时间。没有等待状态的存储器访问时间是 30.3 - 8.0 - 3.8，即 18.5ns，这个时间对于访问 SRAM 足够了，但如果不在时序中插入等待状态，这么短的时间对于任何 DRAM 都是不够的。SRAM 通常用在外部的二级高速缓存中。

通过控制 Pentium 的$\overline{\text{BRDY}}$输入信号可以向时序中插入等待状态，在 T_2 结束之前$\overline{\text{BRDY}}$信号必须变为逻辑 0，否则多余的 T_2 状态就会插入到时序中。图 18-5 给出了用于较慢存储器且包含了等待状态的读周期时序图。向时序中插入等待状态的结果延长了时序，以便存储器有较多的时间访问数据。在所示的时序中，访问时间被延长到可以使用标准的 60ns DRAM。注意，这需要加入 4 个 15.2ns（一个时钟周期）的等待状态，将访问时间延长到 79.5ns。这段时间对于 DRAM 和译码器的工作都足够了。

$\overline{\text{BRDY}}$信号是由系统时钟产生的同步信号，图 18-6 显示了一个可用于产生$\overline{\text{BRDY}}$信号的电路，可以向 Pentium 时序图中添加任意个等待状态。你或许会记得在 80386 微处理器中也有一个类似的电路用于插入等待状态。$\overline{\text{ADS}}$信号通过 74F161 移位寄存器被延迟 0～7 个时钟周期，以产生$\overline{\text{BRDY}}$信号，确切的等待状态数目由 74F151 八选一多路转换器选择。在这个例子中，多路器从移位寄存器中选择了 4 个等待状态的输出。

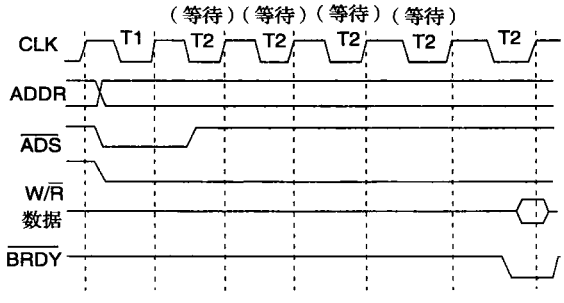


图 18-5 插入了 4 个等待状态，访问时间为 79.5ns 的 Pentium 时序图

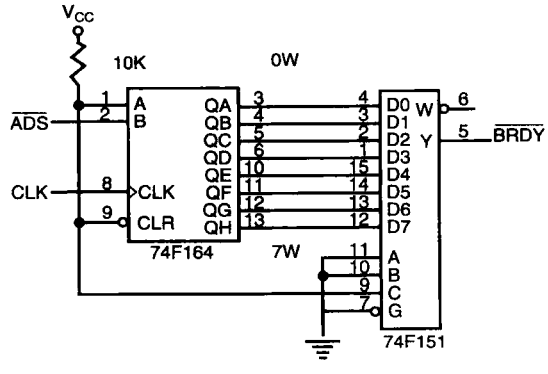


图 18-6 通过延迟$\overline{\text{ADS}}$产生等待状态的电路（此电路产生 4 个等待状态）

读存储器数据的更有效方法是使用猝发（burst）周期。Pentium 在一个猝发周期里的 5 个时钟周期中可传送 4 个 64 位数。没有等待状态的猝发周期，需要存储系统每 15.2ns 传送一次数据。如果有二级高速缓存，获得这个速度是没有问题的，只要从高速缓存读取数据即可。如果在高速缓存中没有包含所需数据，那么就必须加入等待状态，这将会降低系统的吞吐量。图 18-7 所示为 Pentium 没有等待状态的猝发周期传送时序。与前面所述相同，等待状态的插入会使存储系统有更多的时间来访问数据。

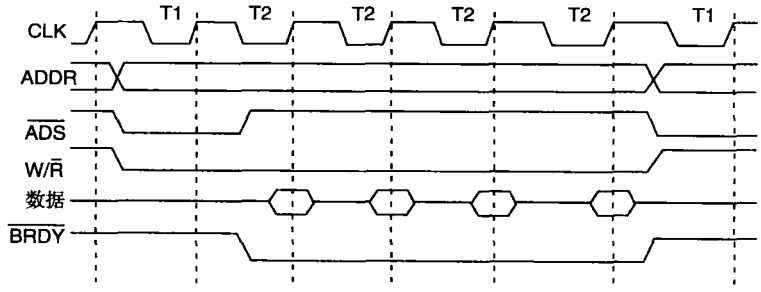


图 18-7 在微处理器和存储器之间传送 4 个 64 位数据的 Pentium 猝发周期操作

18.1.4 分支预测逻辑

Pentium 微处理器采用分支预测逻辑来减少由于分支导致的时间消耗。由于微处理器在遇到分支指令（仅限于短转移或近转移）时在分支地址处进行了指令预取，因而减少了时间消耗。这些指令被装入到指令缓存中，因而出现分支时，指令就已经存在了，从而使分支可以在一个时钟周期内执行。如果由于某种原因分支预测逻辑出错，那么分支就需要再多 3 个时钟周期来执行。在大多数情况下，分支预测逻辑是正确的，因此不会有延迟发生。

18.1.5 高速缓存结构

Pentium 的高速缓存结构与 80486 微处理器中的高速缓存结构不同。Pentium 包括两个 8KB 高速缓存而 80486 只有一个。Pentium 有一个 8KB 的数据缓存和一个 8KB 的指令缓存。指令缓存只存储指令，而数据缓存只存储指令所需的数据。

在只有统一高速缓存的 80486 中，一个数据密集的程序很快就会占满缓存，几乎没有空间用于指令缓存，这就降低了 80486 微处理器的执行速度。在 Pentium 中就不会发生这种情况，因为有单独的指令缓存。

18.1.6 超标量体系结构

Pentium 微处理器由三个执行单元组成，一个执行浮点指令，而另两个（U 管道和 V 管道）执行整型指令。这意味着 Pentium 可同时执行 3 条指令。例如，指令 FADD ST，ST（2），指令 MOV EAX，10H 和指令 MOV EBX，12H 可同时执行，因为它们并不互相依赖。FADD ST，ST（2）指令由协处理器执行，MOV EAX，10H 指令由 U 流水线执行，而 MOV EBX，12H 指令由 V 流水线执行。由于浮点部件也用于 MMX 指令，如果可用，Pentium 可以同时执行 2 条整数指令和 1 条 MMX 指令。

在编写软件时应充分利用这个特性，通过查看程序中的指令，对那些相互依赖而又可以分解为非依赖的指令进行修改。修改后，在某些软件中可能会获得近 40% 的执行速度的提高。一定要在新的编译器和应用程序包中使用这种超标量体系结构特性。

18.2 Pentium 的特定寄存器

除了控制寄存器组有一些新增加的特性和变化外，本质上 Pentium 与 80386 和 80486 微处理器相同。这一节着重介绍 80386 和 Pentium 在控制寄存器结构和标志寄存器上的差异。

18.2.1 控制寄存器

图 18-8 显示了 Pentium 微处理器的控制寄存器结构。请注意，控制寄存器阵列中增加了一个新的

控制寄存器 CR_4。

本节只介绍控制寄存器中新的 Pentium 部件，对 80386 控制寄存器的描述和图解请参阅图 17-14。以下是对新的控制位和新的控制寄存器 CR_4 的描述。

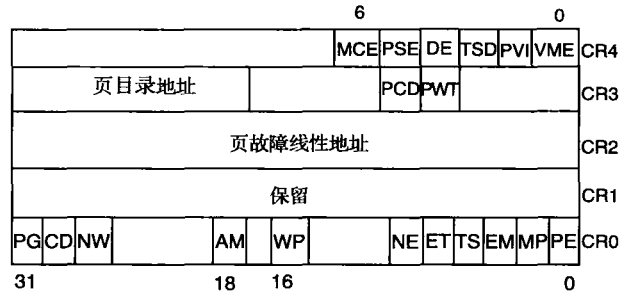


图 18-8 Pentium 控制寄存器结构

CD **cache 禁止（cache disable）**。该位用来控制内部 cache。如果 CD = 1，cache 将不再为未命中的 cache 填入新数据，但对于命中的 cache 还将继续起作用。如果 CD = 0，未命中将会导致新数据的填入。

NW **未直写（not write-through）**。该位用来选择数据 cache 的操作模式。若 NW = 1，数据 cache 将被禁止 cache 直写。

AM **地址对齐屏蔽（alignment mask）**。该位置位时允许对齐检查，请注意仅当在保护模式下用户处于优先级 3 时才发生对齐检查。

WP **写保护（write protect）**。该位保护用户级页使其不接受超级用户的写操作，当 WP = 1 时超级用户可以向用户级段进行写操作。

NE **数字错（numeric error）**。该位使能标准的算术协处理器错误检测，如果 NE = 1，$\overline{\text{FERR}}$引脚将激活响应算术协处理器错；如果 NE = 0，则任何协处理器错误都将被忽略。

VME **虚拟方式扩展（virtual mode extension）**。在保护模式下该位使能对虚拟中断标志的支持。如果 VME = 0，则虚拟中断支持被禁止。

PVI **保护模式虚拟中断（protected mode virtual interrupt）**。在保护模式下使能对虚拟中断标志的支持。

TSD **时间戳禁止（time stamp disable）**。该位控制 RDTSC 指令。

DE **调试扩展（debugging extension）**。设置时该位使能 I/O 断点调试扩展。

PSE **页尺寸扩展（page size extension）**。置位时该位使能 4MB 存储页。

MCE **机器检查使能（machine check enable）**。该位允许机器检查中断。

Pentium 包含一些由 CR_4 和 CR_0 中的一些位控制的新特性，这些新特性稍后再进行介绍。

18.2.2 EFLAG 寄存器

扩展标志寄存器（EFLAG）在 Pentium 微处理器中有所变化。图 18-9 描述了 EFLAG 寄存器的内容。注意，Pentium 增加了 4 个新的标志位，用于控制和指示一些关于 Pentium 新特性的条件，以下是这四个新的标志位及它们的功能：

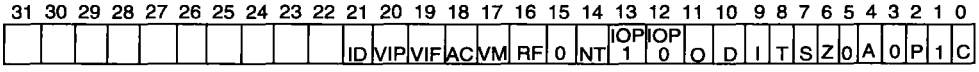


图 18-9 Pentium EFLAG 寄存器结构

注：标志寄存器中的空位保留未用，必须不定义这些位。

ID **标识（identification）**位，用于 CPUID 指令的检测。如果程序可以设置和清除 ID 位，则该处理器支持 CPUID 指令。

VIP **虚拟中断挂起（virtual interrupt pending）**位，指示一个虚拟中断处于挂起状态。

VIF **虚拟中断（virtual interrupt）**位，与 VIP 一起使用的虚拟中断标志 IF 的镜像。

AC **地址对齐检查（alignment check）**位，指示控制寄存器 0 中 AM 位的状态。

18.2.3 内置自检（BIST）

内置自检（BIST）通过在加电时当 RESET 引脚从 1 变到 0 时给 INIT 置逻辑 1 来实现。大约在

150μs 内 BIST 检查 Pentium 的 70% 的内部结构。BIST 结束后，Pentium 在 EAX 寄存器中报告测试结果。若 EAX = 0，则 BIST 通过，Pentium 开始工作；若 EAX 中包含其他值，则 Pentium 有故障。

18.3 Pentium 的存储管理

Pentium 内的存储管理单元与 80386 和 80486 微处理器的存储单元是向上兼容的，许多早期微处理器的特性在 Pentium 中基本上都没有什么改变，最主要的变化在于分页单元和新的系统存储管理模式。

18.3.1 分页单元

页管理机制工作于 4KB 页或 Pentium 新扩展的 4MB 页。正如在第 1 章和第 17 章所描述的，在一个具有很大存储器的系统中页表的尺寸可能会变得很大。记得为了给 4GB 存储器完全重新分页，早期的微处理器大约需要 4MB 的内存来存储页表。在 Pentium 中，由于新的 4MB 分页特性，只需要单一的一个页目录，没有页表，从而大大地减少了内存用量。新的 4MB 页的大小可以由控制寄存器 0 的 PSE 位选择。

4KB 页和 4MB 页的主要区别在于在 4MB 页方式中没有线性地址的页表入口，Pentium 微处理器的 4MB 分页系统请参考图 18-10。请注意在该模式下是如何使用线性地址的，线性地址的最左 10 位在页目录中选择一个入口（与 4KB 一样）。与 4KB 页不同的是，这里没有页表，而是使用页目录来寻址 4MB 内存页。

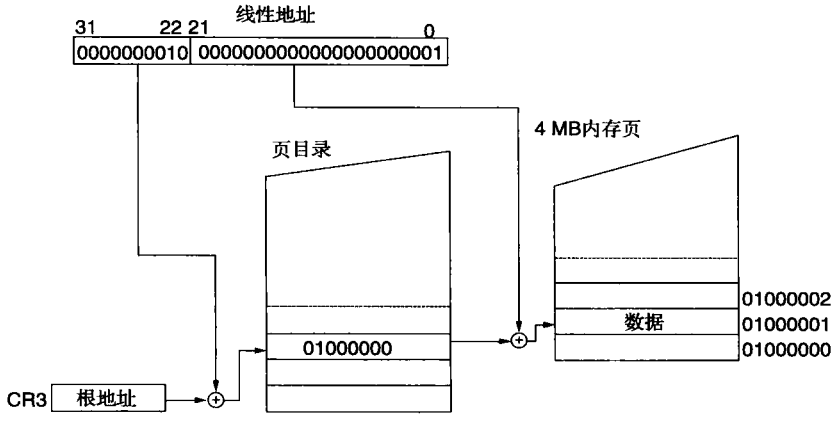


图 18-10 线性地址 00200001H 在 4MB 页中重新分页到 01000002H

注：没有页表。

18.3.2 存储管理模式

系统存储管理模式（SMM）与保护模式、实模式和虚拟模式处于同一级别，但 SMM 用作管理者。SMM 不是用作一个应用程序或一个系统级特性的。它的目的是高层的系统功能，如电源管理和安全性，Pentium 在运行时会用到这些功能，但是这些由操作系统控制。

对 SMM 的访问是通过应用于 Pentium 的$\overline{\text{SMI}}$引脚的新的外部硬件中断来实现的。当硬件中断激活时，微处理器开始执行位于称为系统管理 RAM（或 SMMRAM）的内存区域的系统级软件。$\overline{\text{SMI}}$中断禁止所有其他一般由用户应用程序和操作系统处理的中断，SMM 中断的返回由一个称为 RSM 的新指令完成。RSM 指令可以从存储管理方式中断返回到被中断程序的中断点处。

SMM 中断将调用初始地址为 38000H，CS = 3000H 和 EIP = 8000H 的软件，这个初始态可以通过一个跳转指令跳到第一个 1MB 内存中的任意地址。管理模式中断进入一个与实模式存储器寻址相类似的环境，但又有所不同，因为 SMM 方式不仅能寻址第一个 1MB 内存，而且是把整个存储系统视为一个平展式的 4GB 系统。

除了执行始于 38000H 处的软件外，SMM 中断还把 Pentium 的状态保存到一个**转储记录（dump re-**

cord）中，转储记录存储在3FFA8H至3FFFFH和由Intel保留的3FE00H至3FEF7H。转储记录可以使基于Pentium的系统进入睡眠模式并且使系统返回到中断点。这就要求在睡眠期间SMMRAM必须有电。许多膝上型电脑有独立的电池，在睡眠模式期间可以为SMMRAM供电很长时间。表18-2列出了转储记录的内容。

当通过RSM指令退出SMM模式时，会用到停机自动重启和I/O陷阱重启。这些数据使RSM指令返回到停机状态或返回到中断I/O指令。如果在进入SMM模式时，停机操作和I/O操作都没有起作用。则RSM指令从状态转储记录中重新装入机器状态并返回到中断点。

在通常的操作系统装入内存和执行之前就可以使用SMM模式。假如38000H～3FFFFH处没有普通软件，SMM模式也可以用来周期性地管理系统。如果在引导普通操作系统之前系统重新定位了SMRAM，那么除了普通系统外，它也可用。

SMM模式的SMMRAM的基地址，可以通过在第一次存储管理方式中断之后修改状态转储记录基地址寄存器（位于3FEF8H～3F3FBH）的值来改变。当执行第一个RSM指令时，控制就返回到中断系统。这些位置的新值改变了SMM中断的基地址，以备将来使用。例如，若状态转储基地址变为000E8000H，那么所有接下来的SMM中断都使用E8000H～EFFFFH单元来存储Pentium状态转储记录。这些单元与DOS和Windows兼容。

表18-2 Pentium SMM状态转储记录

偏移地址	寄存器
FFFCH	CR_0
FFF8H	CR_3
FFF4H	EFLAGS
FFF0H	EIP
FFECH	EDI
FFE8H	ESI
FFE4H	EBP
FFE0H	ESP
FFDCH	EBX
FFD8H	EDX
FFD4H	ECX
FFD0H	EAX
FFCCH	DR_6
FFC8H	DR_7
FFC4H	TR
FFC0H	LDTR
FFBCH	GS
FFB8H	FS
FFB4H	DS
FFB0H	SS
FFACH	CS
FFA8H	ES
FF04H～FFA7H	保留
FF02H	停机自动重启
FF00H	I/O陷阱重启
FEFCH	SMM修正标识符
FEF8H	状态转储基址
FE00H～FEF7H	保留

注：这些偏移地址起始位于基地址00003000H。

18.4 Pentium的新指令

Pentium仅包含一个用于正常系统软件的新指令，其他的新指令用于控制存储管理模式特性和串行化指令。表18-3列出了Pentium指令集中新增加的指令。

CMPXCHG8B指令是80486指令集中CMPXCHG指令的扩展。CMPXCHG8B指令用于对存储在EDX和EAX中的64位数和存储器中的64位数，或寄存器对中的数进行比较。例如CMPXCHG8B $DATA_2$，该指令对存储于内存地址$DATA_2$处的8个字节和存于EDX和EAX的64位数进行比较，如果$DATA_2$等于EDX：EAX，那么存储于ECX：EBX的64位数就存到内存地址$DATA_2$处；如果不相等，则$DATA_2$地址处的内容被存到EDX：EAX。注意，零标志位指示了EDX：EAX处的内容是否与$DATA_2$处的内容相等。

表18-3 新的Pentium指令

指令	功能
CMPXCHG8B	比较并交换8个字节
CPUID	返回CPU标识码
RDTSC	读时间戳计数器
RDMSR	读特定模式寄存器
WRMSR	写特定模式寄存器
RSM	从系统管理中断中返回

CPUID指令从Pentium中读取CPU标识码和其他一些信息。表18-4显示了CPUID指令根据不同的EAX输入值返回的不同信息。要执行CPUID指令，必须先为EAX装入输入值，然后再执行，返回信息存储于表中所示的寄存器中。

如果在执行CPUID指令前给EAX置0，那么微处理器就在EBX、EDX和ECX中返回销售商标识，例如，Intel Pentium返回ASCII码“GenuineIntel”，其中EBX中为“Genu”，EDX中为“inel”，ECX中

为“ntel”。若在执行 CPUID 指令前给 EAX 中置 1，则 EDX 寄存器返回相应信息。

表 18-4 CPUID 指令信息

输入值（EAX）	CPUID 执行后的结果
0	EAX = 1（对于所有处理器）
	EBX – EDX – ECX = 厂商标识
1	EAX（位 3～0）= 版本标识
	EAX（位 7～4）= 型号
	EAX（位 11～8）= 系列
	EAX（位 13～12）= 类型
	EAX（位 31～14）= 保留
	EDX（位 0）= CPU 中包含 FPU
	EDX（位 1）= 支持增强的 8086 虚拟模式
	EDX（位 2）= 支持 I/O 断点
	EDX（位 3）= 支持分页扩展
	EDX（位 4）= 支持时间戳计数器（TSC）
	EDX（位 5）= 支持 Pentium 风格的 MSR
	EDX（位 6）= 保留
	EDX（位 7）= 支持机器检查异常
	EDX（位 8）= 支持 CMPXCHG8B
	EDX（位 9）= 3.3V 微处理器
	EDX（位 10～31）= 保留

例 18-1 描述了一个用 CPUID 指令读供应商信息的小程序。这个程序放到简单对话框应用程序的 OnInitDialog 函数的 TODO：区。然后在如图 18-11 所示的 ActiveX 标签上显示。CPUID 指令既可以在实模式下又可以在保护模式下运行，可以用于任意的 Windows 应用程序。

例 18-1

```
CString temp;
int a, b, c;
_asm
{
        mov  eax,0
        cpuid
        mov  a,ebx
        mov  b,edx
        mov  c,ecx
}
for (int d = 0; d < 4; d++ )
{
       temp += (char)a;
       a >>= 8;
}
for (d = 0; d < 4; d++ )
{
       temp += (char)b;
       b >>= 8;
}
for (d = 0; d < 4; d++ )
{
       temp += (char)c;
       c >>= 8;
}
Label1.put_Caption(temp);
```

RDTSC 指令把时间戳计数器读入 EDX：EAX。时间戳计数器从微处理器复位时开始计数 CPU 时钟，这里时间戳计数器被初始化为不确定的值。由于它是 64 位计数器，一个 1GHz 的 Pentium 处理器可计数 580 年。此指令只能运行在实模式或保护模式优先级 0 下。

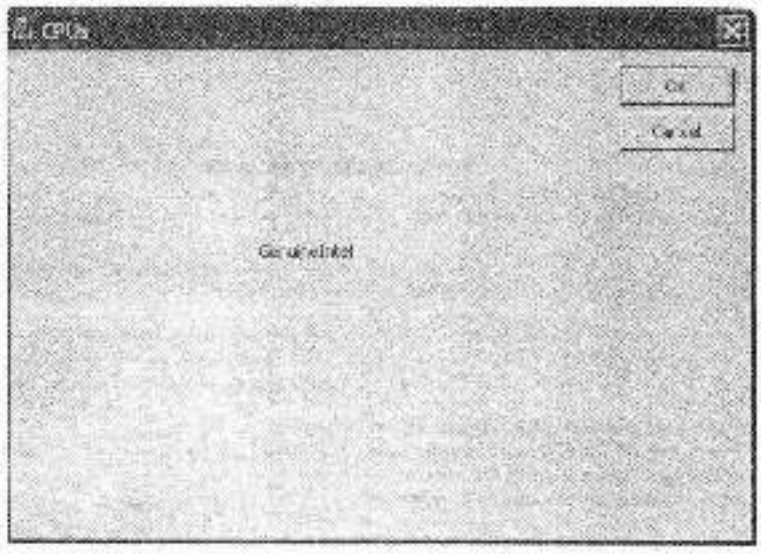


图 18-11 例 18-1 使用 CPUID 指令的程序的屏幕截图

例 18-2 给出了一个 Windows 的类，提供了用于精确延时的成员函数以及测量软件执行时间的成员函数。该类的添加是通过在项目名上击鼠标右键插入名称为 TimeD 的 MFC 普通类来实现。它有三个成员函数，分别为 Start，Stop 和 Delay。

Start() 函数用来启动测量，Stop() 结束时间测量，Stop() 函数返回一个双精度浮点数，该返回值是 Start() 和 Stop() 之间以微秒计的数。

Delay 函数基于时间戳计数器产生一个精确的延时。传给 Delay 函数的参数是毫秒，这意味着 Delay (1000) 产生 1000ms 延迟。

在程序中初始化时，TimeD 用 RegOpenKeyEx 函数打开 Windows 注册表文件，然后用 RegQueryValueEx 函数从注册表文件中以 MHz 单位读出微处理器的频率。微处理器时钟频率被返回到类的变量 MicroFrequency 中。

例 18-2

```
#include "StdAfx.h"
#include ".\timed.h"

int MicroFrequency;                    //频率:MHz
__int64 Count;

TimeD::TimeD(void)
{
        HKEY hKey;
        DWORD dataSize;
                    // 获取处理器频率

        if ( RegOpenKeyEx (HKEY_LOCAL_MACHINE,
                "Hardware\\Description\\System\\CentralProcessor\\0",
                0, KEY_QUERY_VALUE, &hKey) == ERROR_SUCCESS )
        {
                RegQueryValueEx (hKey, _T("~MHz"), NULL, NULL,
                        (LPBYTE)&MicroFrequency, &dataSize);
                RegCloseKey (hKey);
        }
}

TimeD::~TimeD(void)
```

```
{
}

void TimeD::Start(void)
{
        _asm
        {
                rdtsc                   ; 取 TSC (时间戳计数器) 并保存
                mov  dword ptr Count,eax
                mov  dword ptr Count+4,edx
        }
}

double TimeD::Stop(void)
{
        _asm
        {
                rdtsc
                sub  eax,dword ptr Count
                mov  dword ptr Count,eax
                sbb  edx,dword ptr Count+4
                mov  dword ptr Count+4,edx
        }
    return (double)Count/MicroFrequency;
}
void TimeD::Delay(__int64 milliseconds)
{

        milliseconds *= 1000;                   //转换为微秒
        milliseconds *= MicroFrequency;         //转换为原计数
        _asm {
                mov   ebx, dword ptr milliseconds       ;64 位延时: 毫秒 (ms)
                mov   ecx, dword ptr milliseconds+4
                rdtsc                           ;取计数
                add   ebx, eax
                adc   ecx, edx                  ;延时比计数提前
Delay_LOOP1:                                    ;等待计数赶上

                rdtsc
                cmp   edx, ecx
                jb    Delay_LOOP1
                cmp   eax, ebx
                jb    Delay_LOOP1
        }
}
```

如果需要，我们可以为该类添加另外的产生以微秒为单位延时的 Delay，但是应该加以限制，因为从时间戳计数器给计数器加时间本身也需要花费时间，所以延迟时间不要小于 2 到 3 微秒。

例 18-3 显示了简单对话框应用程序点击按钮后在改变 ActiveX 标签前景颜色前用 Delay（）等待 1 秒的例子。例子中没有出现在对话框类的开始位置出现的#include "TimeD. h" 语句。软件本身在 OnInitDialog 函数的 TODO:区。

例 18-3

```
void CRDTSCDlg::OnBnClickedButton1()
{
      TimeD timer;
      timer.Delay(1000);
      Label1.put_ForeColor(0xff0000);
}
```

RDMSR 和 WRMSR 指令用于读或写特定模式寄存器。特定模式寄存器是 Pentium 特有的，用于跟踪和检查性能、测试和检查机器错误。这两条指令都使用 ECX 给微处理器传递寄存器号，使用 EDX：EAX为 64 位宽的读或写服务。注意寄存器地址是 0H ～ 13H。Pentium 特定模式寄存器及其内容

参见表 18-5。与 RDTSC 指令一样，这两条指令也只能在实模式或保护模式优先级 0 下运行。

表 18-5 Pentium 特定模式寄存器

地址（ECX）	大 小	功 能
00H	64 位	机器检查异常地址
01H	5 位	机器检查异常类型
02H	14 位	TR_1 奇偶校验反向测试寄存器
03H	—	—
04H	4 位	TR_2 指令 cache 结束位
05H	32 位	TR_3 cache 数据
06H	32 位	TR_4 cache 标志
07H	15 位	TR_4 cache 控制
08H	32 位	TR_6 TLB 命令
09H	32 位	TR_7 TLB 数据
0AH	—	—
0BH	32 位	TR_9 BTB 标志
0CH	32 位	TR_{10} BTB 目标
0DH	12 位	TR_{11} BTB 控制
0EH	10 位	TR_{12}新特征控制
0FH	—	—
10H	64 位	时间戳计数器（可写）
11H	26 位	事件计数器选择和控制
12H	40 位	事件计数器 0
13H	40 位	事件计数器 1

使用 RDMSR 和 WRMSR 指令之前不要在 ECX 中使用未定义的值。在读或写特定模式寄存器前，如果 ECX = 0，则返回到 EDX:EAX 的值是机器检查异常地址（EDX:EAX 所寻址单元就是写或读特定模式寄存器时存放数据的位置）。如果 ECX = 1，则该位是机器检查异常类型，如果 ECX = 0EH，则测试寄存器 12（TR_{12}）被访问。注意这些都是用于内部测试的内部寄存器，这些寄存器的内容都是 Intel 私有的，在通常的编程中不能用。

18.5 Pentium Pro 微处理器简介

在 Pentium Pro 或其他任何微处理器用于系统之前，必须要了解每个引脚的功能。本章的这一节详述了每个引脚的功能及 Pentium Pro 微处理器的外部存储系统和 I/O 结构。图 18-12 给出了 Pentium Pro 微处理器的外部引脚图，这些引脚都封装在一个很大的 387 个引脚 PGA（引脚栅格阵列）中。目前，Pentium Pro 有两个型号，其中之一包含了 256KB 的二级 cache，另一种包含了 512KB 的二级 cache。Pentium Pro 与 Pentium 引脚相比最大区别在于 Pentium Pro 提供36 位地址总线，这使得 Pentium Pro 能够访问 64GB 的存储器。这是为了将来打算的，因为当前没有任何系统能拥有这么大的内存。

与多数当前型号的微处理器类似，Pentium Pro 的运行需要 +3.3V 或 +2.7V 的电源。150MHz 的 Pentium Pro 的最大电源电流为 9.9A，最大耗电量为 26.7W。目前，需要用一个通风良好的散热器使 Pentium Pro 保持较低温度。与 Pentium 一样，Pentium Pro 有多个 V_{CC} 和 V_{SS} 连接点。为了正确运行，这些接点都必须正确连接。Pentium Pro 的 $V_{CC}P$（主 V_{CC}）引脚组连到 +3.1V 上。$V_{CC}S$（次 V_{CC}）引脚组连到 +3.3V 上。而 $V_{CC}5$（标准 V_{CC}）引脚组连到 +5.0V 上。也有一些标为 N/C（非连接）的引脚不能连接。

每个 Pentium Pro 输出引脚能在逻辑 0 时提供 48.0mA 的电流，与早期微处理器的输出引脚（驱动电流为 2.0mA）相比，该驱动电流有了显著的增加。而每个输入引脚仅需要 15μA 的电流驱动。由于

每个输出引脚有48.0mA的驱动电流，只有非常巨大的系统中才需要总线缓冲。

18.5.1 Pentium Pro的内部结构

Pentium Pro的结构与早期微处理器不同。早期微处理器包含一个执行单元和一个总线接口单元，并有一个小的cache用于为总线接口单元缓冲执行单元。这种结构在后来的微处理器中有所修改，不过这些修改只是在微处理器中附加的升级。Pentium Pro体系结构同样对此做了改进，不过它比早期的微处理器做了更重要的改进。图18-13显示了Pentium Pro微处理器内部结构的框图。

与内存和I/O通信的系统总线和内部二级cache相连，这个二级cache在其他微处理器中通常都是在主板上。Pentium Pro的这个二级cache是256KB或512KB。它的集成加速了操作并减少了系统部件数。

总线接口单元（BIU）通过二级cache控制对系统总线的访问，这与其他微处理器相同。同样，区别在于在Pentium Pro中二级cache是集成于芯片内部的。BIU生成存储器地址和控制信号并向一级数据cache或一级指令cache传送或获取数据或指令。这两个cache当前都是8KB，但在将来的微处理器型号中可能会有所增大。早期型号的Intel微处理器包含一个通用的cache，它既可用于存储数据又可用于存储指令，而分离后的cache提高了系统性能，因为数据密集型的程序不会再把cache填满数据。

指令cache被连接到取指令和译码单元（IFDU），虽然在图中没有显示，但IFDU确实包含3个独立的指令译码器，用于同时对3条指令进行译码。一旦译码完毕，3个译码器的输出就传递到指令池中，一直保留在那里，直到处理和执行单元或弃置单元获得它们。IFDU中也包含一个分支预测逻辑部件，用于在条件跳转指令中预测执行代码序列。如果遇到一个条件跳转，分支预测逻辑就会试图决定程序流中下条将要执行的指令。

译码之后的指令被传送到指令池，在这里指令等待处理。指令池是一个按内容寻址的存储器，Intel从来没有在文献中说明它的大小。

调度和执行单元（DEU）从指令池中取得译码后的指令，然后执行它们。DEU的内部结构如图18-14所示。注意DEU包含3个指令执行单元，两个用于执行整型指令，另一个用于执行浮点指令。这意味着Pentium Pro可以同时处理两个整型指令和一个浮点指令。Pentium也包含3个执行单元，但组织结构有所不同，因为它不包含Pentium Pro所包含的跳转执行单元或地址生成单元。保留站（RS）可以预先安排5个要执行的事件并且同时可处理4个。注意有两个站部件被连接到一个地址生成单元，在图18-14中没有给出它的描述。

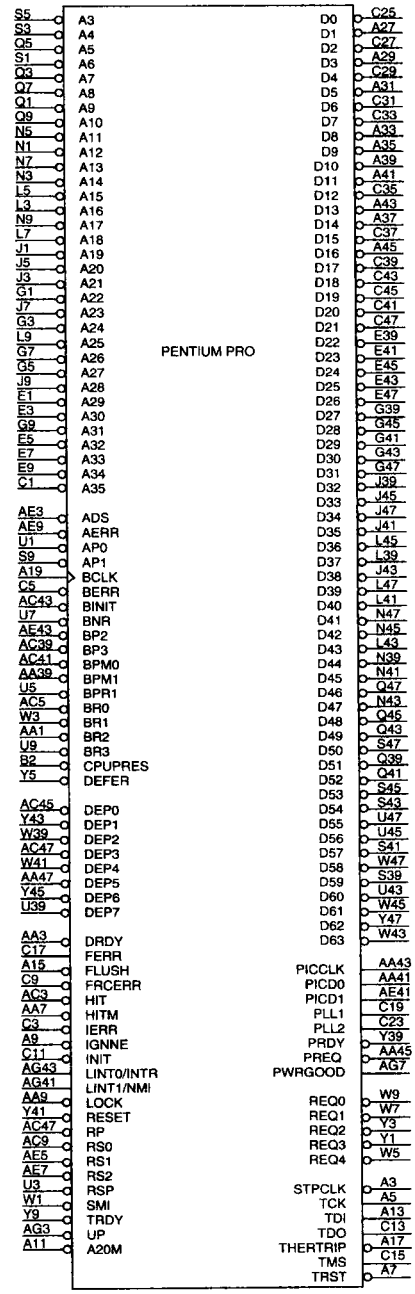


图18-12 Pentium Pro微处理器的外部引脚

Pentium Pro的最后一个内部结构是弃置单元（RU）。RU检查指令池并删除已执行过的译码指令。RU在每个时钟脉冲中可删除3条被译码的指令。

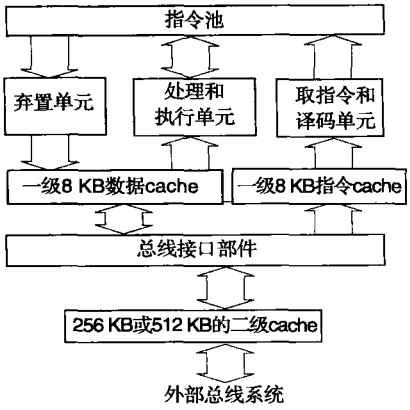


图 18-13 Pentium Pro 微处理器的内部结构

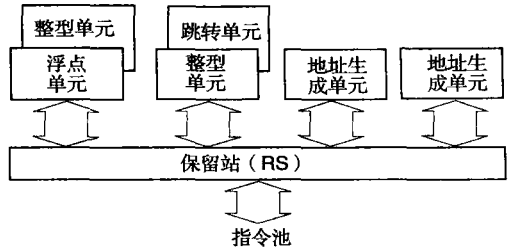


图 18-14 Pentium Pro 的调度与执行单元

18.5.2 引脚连接

Pentium Pro 引脚数从 Pentium 的 237 个增加到 387 个，以下是每个引脚或引脚组的描述：

$\overline{\text{A20M}}$ **地址 A_{20} 屏蔽（address A_{20} mask）**输入引脚，用于在实模式中通知 Pentium Pro 进行地址回绕，就像在 8086 微处理器中一样，它供 HIMEM. SYS 驱动程序使用。

$\overline{\text{A35}}$～$\overline{\text{A3}}$ **地址总线（address bus）**连接引脚，用于寻址 Pentium Pro 存储系统中任何 8G×64 存储单元。

$\overline{\text{ADS}}$ **地址数据选通（address data strobe）**信号。当 Pentium Pro 发出一个有效存储器地址或 I/O 地址时该信号有效。

$\overline{\text{AP0}}$和$\overline{\text{AP1}}$ **地址校验（address parity）**引脚。为所有 Pentium Pro 初始化的内存和 I/O 传送提供偶校验。$\overline{\text{AP0}}$输出引脚为地址线 A_{23}～A_3 提供校验，而$\overline{\text{AP1}}$输出引脚为地址线 A_{35}～A_{24}提供校验。

表 18-6 由 ASZ 引脚确定的存储器大小

$\overline{\text{ASZ1}}$	$\overline{\text{ASZ0}}$	存储器大小
0	0	0～4GB
0	1	4G～64GB
1	×	保留

$\overline{\text{ASZ1}}$和$\overline{\text{ASZ0}}$ **地址宽度（address size）**输入，被驱动用来选择存储器访问的大小。表 18-6 描述了 Pentium Pro 的这两个输入引脚上的二进制位组合所确定的存储器访问大小。

BCLK **总线时钟（bus clock）**输入引脚，确定 Pentium Pro 微处理器的工作频率。例如，若 BCLK 是66MHz，根据表 18-7 中引脚的逻辑电平可确定不同的内部时钟速度。66MHz 的 BCLK 频率使系统总线工作于该频率。

表 18-7 BCLK 信号和它对 Pentium Pro 时钟速度的影响

LINT1/NMI	LINT0/INTR	$\overline{\text{IGNNE}}$	$\overline{\text{A20M}}$	比 率	BCLK = 50MHz	BCLK = 66MHz
0	0	0	0	2	100MHz	133MHz
0	0	0	1	4	200MHz	266MHz
0	0	1	0	3	150MHz	200MHz
0	0	1	1	5	250MHz	333MHz
0	1	0	0	5/2	125MHz	166MHz
0	1	0	1	9/2	225MHz	300MHz
0	1	1	0	7/2	175MHz	233MHz
0	1	1	1	11/2	275MHz	366MHz
1	1	1	1	2	100MHz	133MHz

$\overline{\text{BERR}}$ **总线错（bus error）**输入/输出引脚。发出总线错或者由外部设备声明总线错，从而引起机器检查中断或非屏蔽中断。

BINIT 总线初始化（**bus initialization**）信号。在加电后激活以初始化总线系统。

BNR 锁定下一请求（**block next request**）信号。在多处理器系统中用该信号暂停系统。

BP3和BP2 断点状态（**break point status**）输出信号，指示 Pentium Pro 中断点的状态。

BPM1和BPM0 断点监视器（**breakpoint monitor**）输出信号，指示断点和可编程计数器的状态。

BPRI 优先级代理总线请求（**priority agent bus request**）输入信号，使微处理器停止总线请求。

BR3~BR0 总线请求（**bus request**）输入，最多允许 4 个 Pentium Pro 同时存在于同一个总线系统。

BREQ3~BREQ0 总线请求信号（**bus request signal**），用于具有同一系统总线的多处理器系统。

D63~D0 数据总线（**data bus**）连接引脚，用于在微处理器与存储器及 I/O 系统之间传送字节、字、双字和四字数据。

DBSY 数据总线忙（**data bus busy**）信号，用于指示数据总线正在传送数据。

DEFER 延迟（**defer**）输入信号，在探测阶段被确认，以指示事务不能被保证按顺序完成。

DEN 延迟使能（**defer enable**）信号，在请求阶段的第二个阶段被驱动到总线上。

DEP7~DEP0 数据总线 ECC 保护信号（**data bus ECC protection signal**）为改正单个位错误或是发现双位错误提供纠错码。

FERR 浮点错（**floating-point error**）信号，与 80386 中的 ERROR 兼容，表示内部协处理器出错。

FLUSH 清 cache（**flush cache**）输入信号，使 cache 清除所有的回写行并使内部 cache 无效。如果在复位时，FLUSH输入信号为逻辑 0，Pentium 将进入它的测试模式。

FRCERR 功能性冗余检查错（**functional redundancy check error**）。两个 Pentium Pro 微处理器被配置成一对时使用该信号。

HIT 命中（**hit**）信号，表明在查询方式中内部 cache 包含了合法数据。

HITM 命中修改（**hit modified**）信号，表明在查询周期中发现了一个修改过的 cache 行，此输出信号用于在已修改过的 cache 行回写到存储器之前禁止其他单元访问数据。

IERR 内部错（**internal error**）输出信号，表明 Pentium Pro 已发现一个内部校验错或功能性冗余错。

IGNNE 忽略数字错（**ignore numeric error**）输入信号，使 Pentium Pro 忽略协处理器错误。

INIT 初始化（**initialization**）输出信号，执行不初始化 cache、回写缓冲区和浮点寄存器的复位操作。该信号不能取代加电后的 RESET 信号对处理器的复位。

INTR 中断请求（**interrupt request**）信号。外部电路用该信号请求中断。

LEN1和LEN0 长度信号（位 0 和位 1），指示数据传送的长度，如表 18-8 所示。

表 18-8 LEN位表示的数据传送的长度

LEN1	LEN0	数据传送长度
0	0	0～8 字节
0	1	16 字节
1	0	32 字节
1	1	保留

LINT1和LINT0 局部中断（**local interrupt**）输入信号用作 NMI 和 INTR，并当复位时用来设置时钟分频器频率。

LOCK 锁定信号。在指令带有 LOCK：前缀时该信号为逻辑 0。该信号在 DMA 访问中很常用。

NMI 非屏蔽中断（**non-maskable interrupt**）信号请求一个不可屏蔽中断，这与早期版本的微处理器相同。

PICCLK 时钟信号（**clock signal**）输入用于同步数据传送。

PICD 处理器接口串行数据（**processor interface serial data**）用于在多个 Pentium Pro 微处

理器之间传送双向串行数据。

PWRGOOD 电源正常（**power good**）输入信号，当电源和时钟稳定时为逻辑 1。

$\overline{\text{REQ4}}$～$\overline{\text{REQ0}}$ 请求（**request**）信号（0～4 位）定义数据传送操作的类型，如表 18-9 和表 18-10 所示。

表 18-9 第一个时钟脉冲请求信号的功能

$\overline{\text{REQ4}}$	$\overline{\text{REQ3}}$	$\overline{\text{REQ2}}$	$\overline{\text{REQ1}}$	$\overline{\text{REQ0}}$	功 能
0	0	0	0	0	延迟回答
0	0	0	0	1	保留
0	1	0	0	0	情况 1①
0	1	0	0	1	情况 2①
1	0	0	0	0	I/O 读
1	0	0	0	1	I/O 写
×	×	0	1	0	存储器读
×	×	0	1	1	存储器写
×	×	1	0	0	存储器代码读
×	×	1	1	0	存储器数据读
×	×	1	×	1	存储器写

① 情况 1 和情况 2 的第二个时钟脉冲的功能参见表 18-10。

表 18-10 情况 1 和情况 2 请求输入的功能

情 况	$\overline{\text{REQ4}}$	$\overline{\text{REQ3}}$	$\overline{\text{REQ2}}$	$\overline{\text{REQ1}}$	$\overline{\text{REQ0}}$	功 能
1	×	×	×	0	0	中断响应
1	×	×	×	0	1	特殊处理
1	×	×	×	1	×	保留
2	×	×	×	0	0	分支跟踪消息
2	×	×	×	0	1	保留
2	×	×	×	1	×	保留

$\overline{\text{RESET}}$ **复位**（**reset**）信号初始化 Pentium Pro，使其从内存 FFFFFFF0H 处开始执行软件，Pentium Pro 被复位为实模式，最左边的 12 条地址线保持逻辑 1（FFFH），直至执行一个远程调用或远跳转，这使它与早期微处理器兼容。

$\overline{\text{RP}}$ **请求校验**（**request parity**）信号提供了一种请求 Pentium Pro 进行奇偶校验检查的手段。

$\overline{\text{RS2}}$～$\overline{\text{RS0}}$ **响应状态**（**response status**）输入信号使 Pentium Pro 执行表 18-11 所列的功能。

表 18-11 响应状态输入的操作

$\overline{\text{RS2}}$	$\overline{\text{RS1}}$	$\overline{\text{RS0}}$	功 能	$\overline{\text{HITM}}$	$\overline{\text{DEFER}}$
0	0	0	空闲状态	×	×
0	0	1	重试	0	1
0	1	0	延迟	0	1
0	1	1	保留	0	1
1	0	0	硬件失效	×	×
1	0	1	正常，没有数据	0	0
1	1	0	隐含回写	1	×
1	1	1	正常有数据	0	0

$\overline{\text{RSP}}$ **响应校验**（**response parity**）输入从外部校验检查器中获得校验错信号。

$\overline{\text{SMI}}$ **系统管理中断**（**system management interrupt**）输入信号使 Pentium Pro 进入系统管理操作模式。

$\overline{\text{SMMEM}}$ **系统存储管理模式（system memory-management mode）**信号。当 Pentium Pro 运行于系统存储管理模式中断和地址空间时，该信号为逻辑 0。

$\overline{\text{SPCLK}}$ **分割锁定（split lock）**信号。当该信号置为逻辑 0 电平时表明传送包括 4 个锁定的事务。

$\overline{\text{STPCLK}}$ **停止时钟（stop clock）**信号。该信号为逻辑 0 时使 Pentium Pro 进入低功耗状态。

TCK **可测试性时钟（testability clock）**输出信号。根据 IEEE 1149.1 边界检测接口选择时钟操作。

TDI **测试数据输入（test data input）**信号用来测试由 TCK 信号打入 Pentium Pro 的数据。

TDO **测试数据输出（test data output）**信号用来获得由 TCK 移出的 Pentium Pro 的测试数据和指令。

TMS **测试方式选择（test mode select）**输入信号，在测试模式中控制 Pentium Pro 操作。

$\overline{\text{TRDY}}$ **目标就绪（target ready）**输入信号。目标准备好数据传送操作时声明该信号。

18.5.3 存储系统

Pentium Pro 微处理器的存储系统大小是4GB，与80386DX～Pentium 的相同，但它可以用新增加的地址信号 A_{32}～A_{35}，可访问4～64GB 的地址空间。Pentium Pro 使用64 位数据总线来访问由 8 个存储体组成的、每个存储体又包含了 8GB 数据的存储器。注意，附加的存储器由 CR_4 的第 5 位来使能，并且仅当 2MB 分页使能时才能被访问。2MB 分页是 Pentium Pro 所特有的，用于访问 4GB 以上的内存。本章后面会对 Pentium Pro 的分页有更详尽的描述，Pentium Pro 物理存储系统的组织参见图 18-15。

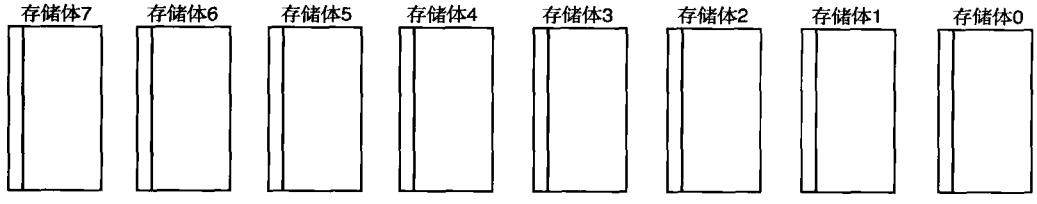


图 18-15 Pentium Pro 系统中的 8 个存储体

注：每个存储体是 8 位宽且 8GB 长（如果使能 36 位寻址）。

Pentium Pro 存储系统分成 8 个存储体，每个存储体都有一位校验位，需要一个字节来保存。绝大多数基于 Pentium 和 Pentium Pro 的系统都不再使用奇偶校验位。与 80486 和 Pentium 一样，Pentium Pro 采用内部奇偶校验生成器和检查逻辑为存储器数据总线生成信息。64 位宽的存储器对于双精度浮点数据是很重要的，因为双精度浮点型数正好是 64 位宽。与早期的 Intel 微处理器一样，存储系统按字节从 000000000H 到 FFFFFFFFH 计数。这个 9 位十六进制的地址用于寻址 64GB 内存的系统中。

存储器选择是由体选通信号（$\overline{\text{BE7}}$～$\overline{\text{BE0}}$）完成的。在 Pentium Pro 微处理器中，体选通信号在存储器或 I/O 访问的第二个时钟周期时出现在地址总线（A_{15}～A_8）上，这些信号必须从地址总线上抽取出来，以访问存储体。这种独立的存储体使 Pentium Pro 在一个存储器传送周期内可访问单字节、字、双字或四字数据。与早期的存储器选择逻辑一样，通常为存储系统写产生 8 个独立的写选通脉冲。存储器写信息在存储器或 I/O 访问的第二个时钟阶段由微处理器的请求线提供。

Pentium 和 Pentium Pro 的一个新增特性是能够在某些操作中检查地址总线和产生奇偶校验。$\overline{\text{AP}}$引脚（Pentium）或$\overline{\text{AP}}$引脚组（Pentium Pro）给系统提供奇偶校验信息，$\overline{\text{APCHK}}$（Pentium）或$\overline{\text{AP}}$引脚组（Pentium Pro）指示地址总线奇偶校验错。当检查出地址校验错误时，Pentium Pro 并不采取任何措施，该错误必须由系统来维护，如果需要，系统可以采取必要的行动（如中断）。

Pentium Pro 新的特点就是内置了一个纠错电路（即 ECC），它可以纠正一位错误，并可以检测两

位错。要实现检错或纠错，存储系统必须要为每个 64 位数提供一个空间，以存储附加的 8 位数。这附加的 8 位用来存储错误纠正码，该纠正码使得 Pentium Pro 可以自动地纠正任何 1 位的错误。64M 不带 ECC 的 SDRAM 为 1M×64，带 ECC 的 SDRAM 为 1M×72。ECC 比老的奇偶校验方案可靠得多，在现代系统中已经很少使用奇偶校验了。ECC 方案的惟一缺点就是增加了 SDRAM 的成本，它为 72 位宽。

18.5.4 输入/输出系统

Pentium Pro 的 I/O 系统与早期 Intel 微处理器的 I/O 系统完全兼容。I/O 端口号出现在地址线 A_{15}～A_3，体选通信号用来选择实际用于 I/O 传送的存储体。

18.5.5 系统时序

与其他所有微处理器一样，为了与微处理器进行接口，必须要理解系统时序信号。本节内容详述 Pentium Pro 操作的系统时序图，并指出如何确定存储器访问的时序。

基本的 Pentium Pro 存储周期由两部分组成：寻址阶段和数据阶段。在寻址阶段，Pentium Pro 把地址（T_1）和控制信号（T_2）送到存储器和 I/O 系统，控制信号包括 ATTR 线（A_{31}～A_{24}）、DID 线（A_{23}～A_{16}）、体选通信号（A_{15}～A_8）和 EXF 线（A_7～A_3）。基本的时序周期参见图 18-16。存储周期的类型出现在请求引脚上。在数据阶段，4 个 64 位宽的数被读取或写到存储器中。这是最普通的操作，因为主存储器的数据是在内部 256KB 或 512KB 回写 cache 和存储系统之间传送。进行字节、字或双字的写操作时（例如 I/O 传送），要使用体选通信号并且在数据传送阶段只有一个时钟周期。从时序图中可以看出，66MHz 的 Pentium Pro 每秒可进行 3300 万次存储器数据传送（假设存储器可以工作于该速度）。

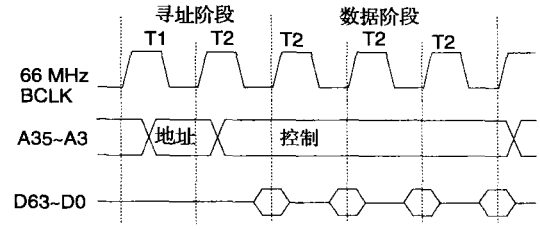


图 18-16 Pentium Pro 的基本时序

时钟前的建立时间是 5.0ns，时钟后的保持时间是 1.5ns。这意味着在此时钟周围的数据窗口有 6.5ns。地址在 T_1 开始最多 8.0ns 后出现。这也意味着工作在 66MHz 下的 Pentium Pro 微处理器所允许的访问时间为 30ns（两个时钟周期）减去 8.0ns 的地址延迟再减去 5.0ns 的数据建立时间。没有等待状态的存储器访问时间是 30－8.0－5.0，即 17.0ns。这段时间对于访问 SRAM 足够了，但如果不在时序中加入等待状态，这么短的时间对于任何 DRAM 都是不够的。

通过控制 Pentium Pro 的输入信号$\overline{\text{TRDY}}$可以向时序中插入等待状态。在 T_2 结束之前，该信号必须变为逻辑 0，否则多余的 T_2 状态就会插入到时序中。注意，60ns 的 DRAM 需要加入 4 个 15ns（一个时钟周期）的等待状态将存取时间延长至 77ns，这段时间对于 DRAM 和系统中的译码器的工作足够了。由于许多 EFROM 存储器件需要 100ns 的存取时间，此时 EFROM 需要加入 7 个等待状态，将存取时间延长至 122ns。

18.6 Pentium Pro 的特性

Pentium Pro 除了增加一些特性以及控制寄存器组发生了一些变化以外，与 80386、80486 和 Pentium 本质上是相同的。本节重点突出 80386 控制寄存器结构与 Pentium Pro 控制寄存器的不同之处。

控制寄存器 4

图 18-17 显示了 Pentium Pro 微处理器的控制寄存器 4。注意 CR_4 有两个新的控制位被添加到控制寄存器阵列中。

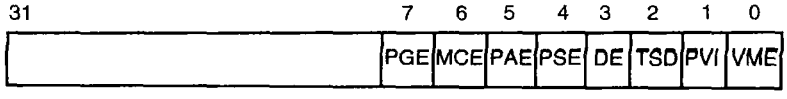


图 18-17 Pentium Pro 微处理器中新的控制寄存器 4（CR_4）

本文这一节只介绍了控制寄存器4中两个新的Pentium Pro组件（查看Pentium控制寄存器的描述和图解请参考图18-9）。以下是Pentium CR_4 位的描述和Pentium Pro控制寄存器 CR_4 中新控制位的描述：

VME **虚拟模式扩展（virtual mode extension）**，在保护模式下使能对虚拟中断标志的支持。如果VME=0则虚拟中断支持被禁止。

PVI **保护模式虚拟中断（protected mode virtual interrupt）**，在保护模式下使能对虚拟中断标志的支持。

TSD **时间戳禁止（time stamp disable）**控制RDTSC指令。

DE **调试扩展（debugging extension）**。该位设置时使能I/O断点调试扩展。

PSE **页尺寸扩展（page size extension）**。在Pentium中，该位置位时使能4MB存储页；在Pentium Pro中，该位置位时使能2MB存储页。

PAE **页地址扩展（page address extension）**。当使用Pentium Pro中由PSE控制的新的寻址方式时使能使用地址线 A_{35}～A_{32}。

MCE **机器检查使能（machine check enable）**。激活机器检查中断。

PGE **页大小扩展（page size extension）**。当与PAE和PSE一起设置时用于控制新的、更大的64G寻址方式。

18.7 小结

1）Pentium微处理器与早期的80386和80486几乎相同。主要区别在于Pentium被改进为内部包含双cache（指令cache和数据cache）和双整型处理单元的微处理器。Pentium的工作时钟也可以高于66MHz。

2）66MHz的Pentium需要3.3A的电流，60MHz的Pentium需要2.91A的电流。电源必须是+5.0V，变化范围±5%。新型的Pentium可以用3.3V和2.7V的电源供电。

3）Pentium的数据总线为64位宽，有8个字节宽度的存储体，由体选通信号（$\overline{BE0}$～$\overline{BE7}$）选通。

4）66MHz的Pentium无等待状态的存储器存取时间大约只有18ns。大多数情况下，需要通过控制Pentium的$\overline{BRDY}$输入给这个短的访问时间插入等待状态。

5）Pentium的超标量结构有3个独立的处理单元：一个浮点处理器和两个被Intel标为U和V的整型处理单元。

6）Pentium的cache结构做了修改，它含有两个cache，一个8K×8的cache作为指令cache，另一个8K×8的cache作为数据cache。数据cache可以作为直写或回写cache。

7）Pentium增加了一个新的工作模式，称为系统存储管理模式（SMM）。SMM模式通过作用于$\overline{SMI}$输入引脚的系统存储管理中断来访问。作为对$\overline{SMI}$的响应，Pentium开始执行始于38000H的程序。

8）Pentium新指令有CMPXCHG8B、RSM、RDMSR、WRMSR和CPUID。CMPXCHG8B指令与80486的CMPXCHG指令类似。RSM指令用于从系统存储管理中断中返回。RDMSR和WRMSR指令用来读或写特定模式寄存器。CPUID用来读取CPU的标识码。

9）内置自检（BIST）可以使Pentium在系统第一次加电时被测试。一般的带电复位激活了Pentium的RESET输入。而BIST加电复位首先要激活INIT，接着才使RESET引脚无效。在BIST通过后，EAX等于00000000H。

10）Intel对于页单元所做的修改允许4MB的存储页而不是4KB页。这是通过用页目录来寻址1024个页实现的，其中每页包含4MB的存储器。

11）Pentium Pro是Pentium微处理器的增强型号，它不仅包含了Pentium内置的一级cache，而且包含大多数主板上才有的256KB或512KB二级cache。

12）Pentium Pro使用与Pentium和80486相同的66MHz的总线速度来运行。它使用内部时钟发生器可以使总线速度按不同的因子加倍，以获得较高的内部执行速度。

13）Pentium Pro与早期微处理器上软件的主要区别在于Pentium Pro增加了FCMOV和CMOV指令。

14）Pentium Pro与早期微处理器上硬件的主要区别在于Pentium Pro增加了2M的分页和4条新增加的地址线，可访问的存储器空间为64GB。

15）Pentium Pro增加了纠错码（ECC），可以纠正1位错误，检测2位错误。

18.8 习题

1. Pentium 微处理器可以访问多大的存储器空间?
2. Pentium Pro 微处理器可以访问多大的存储器空间?
3. Pentium 的存储器总线宽度是_____。
4. Pentium 的 $DP_0 \sim DP_7$ 引脚有何用途?
5. 如果 Pentium 工作频率为 66MHz，那么 CLK 引脚输入的时钟频率是多少?
6. Pentium 微处理器的$\overline{\text{BRDY}}$引脚有何作用?
7. Pentium 微处理器的 AP 引脚有何作用?
8. Pentium 工作在 66MHz 时，没有等待状态的存储器访问时间是多少?
9. Pentium 的哪个引脚用来给时序中插入等待状态?
10. 一个等待状态是附加的_____时钟周期。
11. 解释 Pentium 的两个整型单元如何同时执行两个不相关的指令。
12. Pentium 中有多少高速缓冲存储器? 大小是多少?
13. Pentium 存储器读操作的数据采样窗口有多宽?
14. Pentium 可以同时执行 3 条指令吗?
15. $\overline{\text{SMI}}$引脚有什么作用?
16. 什么是 Pentium 的系统存储管理工作模式?
17. 系统存储管理模式如何退出?
18. $\overline{\text{SMI}}$中断时 Pentium 从哪里开始执行服务程序?
19. 怎样修改系统存储管理单元的转储地址?
20. 解释 CMPXCHG8B 指令的操作。
21. EAX 初始值为 0，执行 CPUID 指令后 EAX 返回什么信息?
22. Pentium 微处理器增加了哪些新的标志位?
23. Pentium 微处理器增加了什么新的控制寄存器?
24. 描述 Pentium 如何访问 4MB 页。
25. 解释时间戳时钟怎样运行以及它怎样用于时间事件。
26. 比较 Pentium 和 Pentium Pro 微处理器。
27. Pentium Pro 微处理器的存储体选通信号在哪里?
28. Pentium Pro 系统有多少地址线?
29. Pentium Pro 的 CR_4 有何变化? 为什么?
30. 比较 Pentium 系统和 Pentium Pro 系统的存储器存取时间。
31. 什么是 ECC?
32. 使用 ECC 必须购买什么样的 SDRAM?

第 19 章　Pentium Ⅱ、Pentium Ⅲ、Pentium 4 和 Core2 微处理器

引言

随着 Intel 的 Itanium[⊖] 和 ItaniumⅡ微处理器的出现，PentiumⅡ、PentiumⅢ、Pentium 4 和 Core2 微处理器标志了 32 位体系结构发展的结束。Itanium 是 64 位体系结构的微处理器。PentiumⅡ、PentiumⅢ、Pentium 4 和 Core2 体系结构是 Pentium Pro 体系结构的扩展，它们有一些区别。最大的区别在于 Pentium Pro 体系结构中的内部 cache 在 Pentium Ⅱ微处理器中被移到了外部。另外一个较大的改变就是 Pentium Ⅱ微处理器没有集成电路封装形式，变为被称为封装盒（Cartridge）的小的插入式印刷电路板形式，板上带有二级 cache 芯片。Pentium Ⅱ有多种型号，其中 Celeron[⊖] 型号在 Pentium Ⅱ电路板上没有包含二级 cache。Xeon[⊜] 是 Pentium Ⅱ的一种增强型号，在电路板上最多可以有 2MB 的 cache。

与 Pentium Ⅱ相似，早期 PentiumⅢ微处理器封装在封装盒里而不是集成电路。比较新的型号，如 Coppermine，又封装在集成电路（370 引脚）中。PentiumⅢ的 Coppermine 与 Pentium Pro 一样，内部含有 cache。Pentium 4 封装在更大的集成电路中，有 423 或 478 个引脚。最新的 Pentium 4 和 Core2 采用 775 个引脚触点阵列封装技术制造。Pentium 4 采用的是物理上更小的晶体管，使其比 PentiumⅢ更小更快。到目前为止，Intel 已经发布了工作频率在 3GHz 以上的 Pentium 4 和 Core2 处理器，在未来也许能达到 10GHz。Pentium 4 和 Core2 具有 2MB 高速缓存的极大型号和具有 4MB 高速缓存的再版极大型号。与早期使用 0.13μs 制程的 Pentium 4 相比，Pentium 4 和 Core2 型号现在都是 65nm（0.065μs）制程。最新的是 Core2 Duo 和 Core2 Quad 型号，它们采用的是 45nm 技术并且是双核或四核。

目的

读者学习完本章后将能够：

1）详述 PentiumⅡ、PentiumⅢ、Pentium 4 和 Core2 与前面的 Intel 微处理器之间的区别。
2）解释 PentiumⅡ、PentiumⅢ、Pentium 4 和 Core2 的体系结构如何提高系统速度。
3）说明使用 PentiumⅡ、PentiumⅢ、Pentium 4 和 Core2 微处理器时，计算机系统的基本体系结构如何改变。
4）详述 CPUID 指令和特定模型寄存器的变化。
5）描述 SYSENTER 和 SYSEXIT 指令的操作。
6）描述 FXSAVE 和 FXRSTOR 指令的操作。

19.1　Pentium Ⅱ微处理器简介

在将 Pentium Ⅱ微处理器或其他微处理器用于系统之前，必须了解每个引脚的功能。本章这一节详述每个引脚的功能以及 Pentium Ⅱ微处理器的外部存储系统和 I/O 结构。

图 19-1 说明了 Pentium Ⅱ微处理器 slot1 连接器的基本外形以及用于与芯片组接口的信号。图 19-2 显示了盒式封装组件的简图以及在典型 Pentium Ⅱ系统中 Pentium Ⅱ盒式结构和总线组件的布局简图。

⊖ Itanium 是 Intel 公司注册的商标。
⊖ Celeron 是 Intel 公司注册的商标。
⊜ Xeon 是 Intel 公司注册的商标。

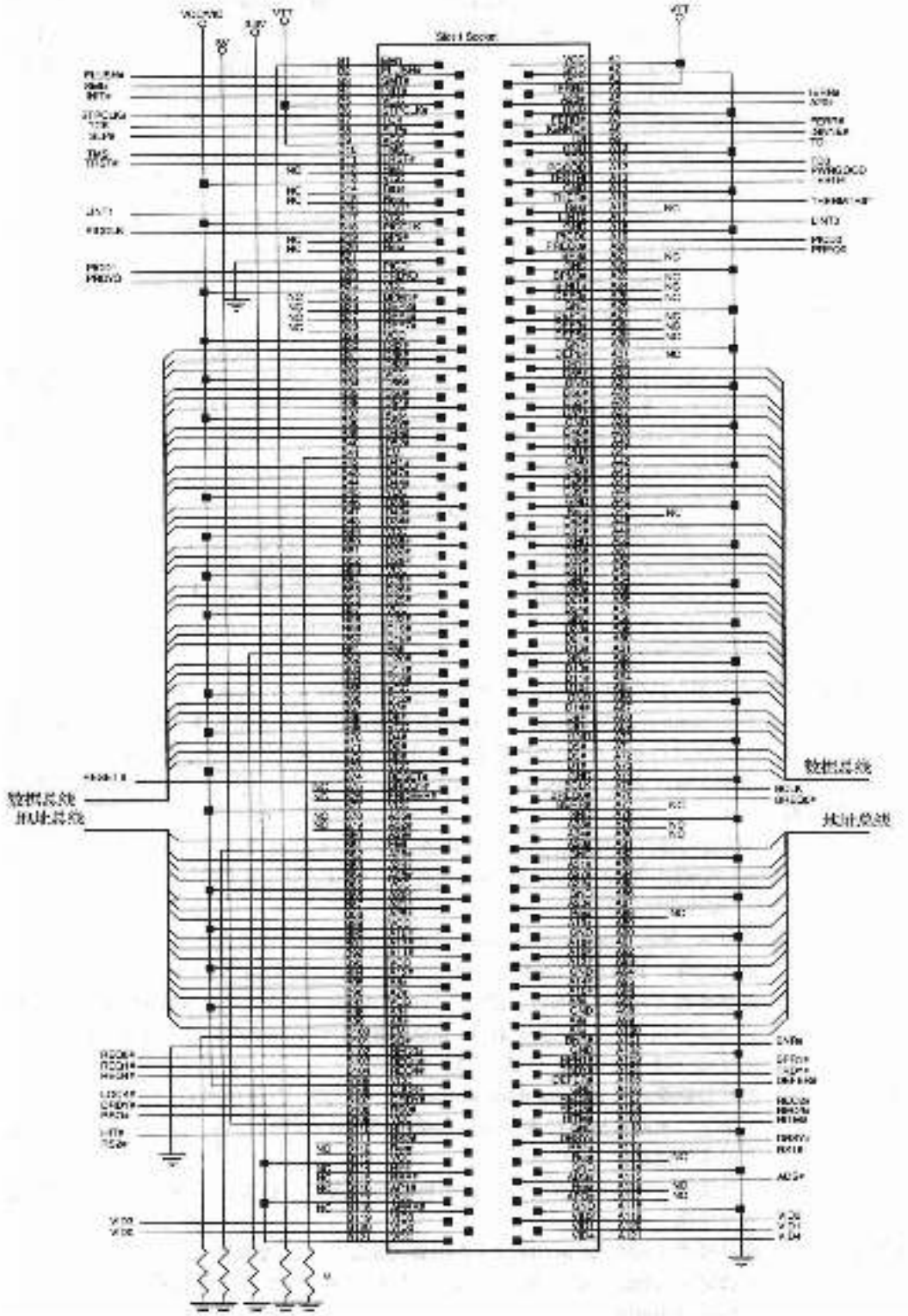


图 19-1 slot 1 连接器的引脚及其与系统的连接

微处理器 slot1 连接器有 242 个引脚（这些连接减少了 Pentium 和 Pentium Ⅱ微处理器上所见到的引脚数）。Pentium Ⅱ不再像过去的 Intel 微处理器被封装在集成电路里，而是被封装在印刷电路板上。它的一级 cache 与 Pentium Pro 相同，是 32KB，但二级 cache 不再放在集成电路内。Intel 改变了体系结构，因此二级 cache 可以放得离微处理器很近。这个改变使微处理器成本降低并且使得二级 cache 有效工作。Pentium Ⅱ的二级 cache 工作频率是微处理器时钟频率的一半，而不是 Pentium 微处理器的 66MHz。400MHz 的 Pentium Ⅱ的二级 cache 速度为 200MHz。Pentium Ⅱ有三种型号。第一种型号是功能完整的 Pentium Ⅱ，也就是我们通常所说的 Pentium Ⅱ，它采用了 slot1 连接器。第二种型号是 Celeron，它除了在 slot1 电路板中没有包含二级 cache 外，与第一种型号（Pentium Ⅱ）相似，Celeron 系统中的二级 cache 位于系统主板上，工作在 66MHz 下。最新的型号是 Xeon，由于使用了 512KB、1MB 或 2MB 的二级 cache，它代表了 Pentium Ⅱ有效的速度改进。Xeon 的二级 cache 可以以微处理器的时钟频率工作。400MHz 的 Xeon 的二级 cache 速度为 400MHz，这速度是常规 Pentium Ⅱ的两倍。

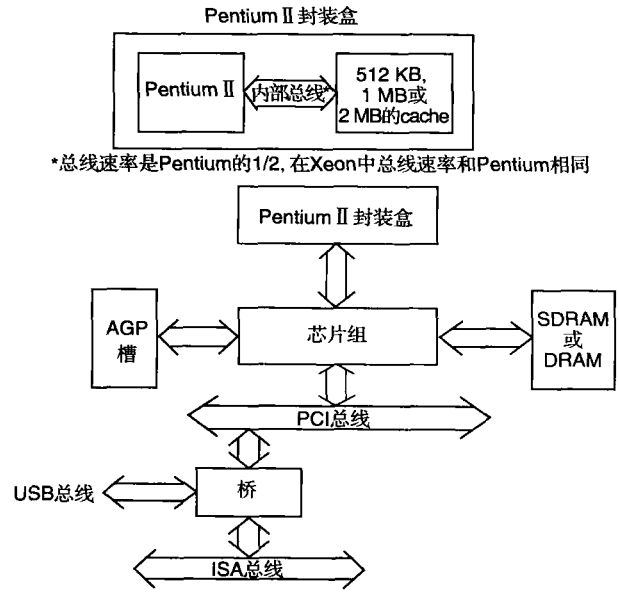


图 19-2 Pentium Ⅱ盒式结构和 Pentium Ⅱ系统结构图

早期的 Pentium Ⅱ型号需要 +5.0V、+3.3V和可变电压的电源。微处理器上主可变电源的电压可以从 3.5V 低到 1.8V。这就是大家知道的微处理器核电压。根据 Pentium Ⅱ微处理器的工作频率和电源电压，电源电流平均为 14.2A 到 8.4A。由于这些电流非常大，所以 Pentium Ⅱ微处理器的功耗也很大。目前，Pentium Ⅱ需要用靠气流散热的散热片来降低温度。非常幸运，散热片和风扇可以做在 Pentium Ⅱ封装盒上。最新的 Pentium Ⅱ的型号已经对功耗问题做了改进。

每个 PentiumⅡ封装盒的输出引脚能够在信号连接点为逻辑 0 时提供至少 36.0mA 的电流。一些输出控制信号仅提供 14mA 的电流。PentiumⅡ的另一个改变就是它的输出是漏极开路，需要外部的上拉电阻。

各组 Pentium Ⅱ引脚的功能如下：

$\overline{\text{A20}}$ **地址 A_{20}屏蔽（address A_{20} mask）**输入引脚用于在实模式中给 Pentium Ⅱ发信号进行地址回绕，就像在 8086 微处理器中一样，它供 HIMEM.SYS 驱动程序使用。

$\overline{\text{A35}}$～$\overline{\text{A3}}$ **地址总线（address bus）**。该信号低电平有效，用于寻址 Pentium Ⅱ存储系统中任意存储器单元。注意 A_0、A_1 和 A_2 在总线使能（$\overline{\text{BE7}}$～$\overline{\text{BE0}}$）中被编码用于选择 64 位宽存储空间中的任意一个或全部 8 个字节。

$\overline{\text{ADS}}$ **地址数据选通（address data strobe）**输入引脚，当该信号有效时，给 Pentium Ⅱ发出指示，表明系统已经做好执行存储器或 I/O 操作的准备。该信号使得微处理器给系统提供地址。

$\overline{\text{AERR}}$ **地址错误（address error）**输入引脚。如果该信号被激活，Pentium Ⅱ就会检查地址校验错误。

$\overline{\text{AP1}}$和$\overline{\text{AP0}}$ **地址校验（address parity）**输入引脚用来指示地址校验错误。

$\overline{\text{BCLK}}$ **总线时钟（bus clock）**输入信号。该信号用来设定总线时钟频率。在 Pentium Ⅱ中为 66MHz 或 100MHz。

$\overline{\text{BERR}}$ **总线错误（bus error）**信号。该引脚激活时表明总线系统上有错误出现。

$\overline{\text{BINIT}}$	**总线初始化（bus initialization）**信号。当系统复位或初始化时该引脚为逻辑 0。该引脚作为输入指示总线出现错误并需要重新初始化。
$\overline{\text{BNR}}$	**总线未就绪（bus not ready）**输入引脚用来给 Pentium Ⅱ 时序插入等待状态。该引脚为逻辑 0 时，Pentium Ⅱ 将进入迟延状态或等待状态。
$\overline{\text{BP3}}$、$\overline{\text{BP2}}$、$\overline{\text{PM1}}/\overline{\text{BP1}}$和$\overline{\text{PM0}}/\overline{\text{BP0}}$	当调试寄存器被编程来监视匹配断点时，**断点引脚（breakpoint pin）**$\overline{\text{BP3}}$~$\overline{\text{BP0}}$用来指示匹配的断点。**性能监视（performance monitoring）**引脚$\overline{\text{PM1}}$和$\overline{\text{PM0}}$用来指示调试模式控制寄存器的性能监视位的设置。
$\overline{\text{BPRI}}$	**总线优先权请求（bus priority request）**输入用于从 Pentium Ⅱ 请求系统总线。
$\overline{\text{BR1}}$和$\overline{\text{BR0}}$	**总线请求（bus request）**引脚。该信号表明 Pentium Ⅱ 已产生了一个总线请求。在初始化时，$\overline{\text{BR0}}$引脚必须被激活。
BSEL	**总线选择（bus select）**引脚。Pentium Ⅱ 目前没有用到该信号，必须将该引脚接地。
$\overline{\text{D63}}$~$\overline{\text{D0}}$	**数据总线（data bus）**引脚。在微处理器与内存和 I/O 系统间传送字节、字、双字和四字的数据。
$\overline{\text{DEFER}}$	**延期（defer）**信号。用来指示外部系统没有完成总线周期。
$\overline{\text{EP7}}$~$\overline{\text{EP0}}$	**数据 ECC 引脚（Data ECC pin）**。这组信号用在 Pentium Ⅱ 的纠错方案中，一般接到附加的 8 位存储器上，这意味着 ECC 存储器模块为 72 位而不是 64 位宽。
$\overline{\text{DRDY}}$	**数据就绪（Data Ready）**引脚。该信号激活时表明系统给 PentiumⅡ发出了有效数据。
EMI	**电磁干扰（Electro magnetic interference）**引脚。该信号必须接地防止 Pentium Ⅱ 产生或接收干扰。
$\overline{\text{FERR}}$	**浮点错（floating-point error）**引脚。该信号与 80386 中的 ERROR 信号类似，它用来指示内部协处理器出现错误。
$\overline{\text{FLUSH}}$	**清 cache（flush cache）**输入引脚。该信号使 cache 清除所有写回行并使内部 cache 无效。如果在进行复位操作期间$\overline{\text{FLUSH}}$为逻辑 0，则 Pentium Ⅱ 进入到 cache 测试模式。
FRCERR	**功能性冗余检查（functional redundancy check）**引脚，在复位时采样用来配置 Pentium 为主模式（0）或检验者模式（1）。
$\overline{\text{HIT}}$	**命中（hit）**引脚。该信号指示在查询方式中内部 cache 包含有效数据。
$\overline{\text{HITM}}$	**命中修改（hit modified）**引脚。该信号表明在查询周期中发现了一个修改过的 cache 行。在已修改过的 cache 行写回存储器之前该输出信号用于禁止其他主单元访问数据。
$\overline{\text{IERR}}$	**内部错（internal error）**输出引脚。该信号用来表明 Pentium Ⅱ 已检测到一个内部错或功能性冗余错。
$\overline{\text{IGNNE}}$	**忽略数字错（ignore numeric error）**输入引脚，使 Pentium Ⅱ 忽略协处理器错误。
$\overline{\text{INIT}}$	**初始化（initialization）**输入引脚。该信号完成不初始化 cache、回写缓冲区和浮点寄存器的复位操作，此信号在加电后不能用于代替 RESET 信号进行微处理器的复位。
INTR	**中断请求（interrupt request）**引脚。外部电路使用该信号可请求中断。
LINT_1 和 LINT_0	**局部 APIC 中断（local APIC interrupt）**信号，这两个信号必须连接到所有适用的 APIC 总线代理引脚上。当 APIC 被禁止时，LINT_0 信号变为 INTR，即可屏蔽中断请求信号；LINT_1 信号则变为 NMI，即不可屏蔽中断。
$\overline{\text{LOCK}}$	当指令带有 LOCK：前缀时，$\overline{\text{LOCK}}$变为逻辑 0，该信号在 DMA 访问中非常有用。
NMI	**非屏蔽中断（non-maskable interrupt）**请求引脚。该信号用来请求非屏蔽中断，这与早期版本的微处理器相同。
PICCLK	**时钟信号（clock signal）**必须为$\overline{\text{BCLK}}$的四分之一频率。

$PICD_1$ 和 $PICD_0$ **处理器接口串行数据（processor interface serial data）**用于 Pentium Ⅱ 和 APIC 之间的串行消息。

PRDY **探测就绪（probe ready）**输出引脚。该信号表明已进入调试探测模式。

$\overline{PREQ}$ **探测请求（probe request）**。该信号用来请求调试。

PWRGOOD **电源正常（power good）**指示系统电源工作的输入信号。

$\overline{REQ4}$～$\overline{REQ0}$ **请求信号（request signal）**用于总线控制器和 Pentium Ⅱ 的命令通信。

$\overline{RESET}$ **复位（reset）**信号。该信号使 Pentium Ⅱ 初始化，使其开始执行位于存储单元 FFFFFFF0H 或 000FFFF0H 的程序。在复位时，A_{35}～A_{32}地址位被置为 0。Pentium Ⅱ 被复位为实模式，最左边 12 位地址线保持逻辑 1（FFFH），直到执行到远跳转（far jump）或远调用（far call）。这使得 Pentium Ⅱ 与早期的微处理器兼容。Pentium Ⅱ 硬件复位后的状态参见表 19-1。

表 19-1 Pentium Ⅱ 复位后的状态

寄 存 器	复 位 值	复位 + BIST 值
EAX	0	0（如果通过测试）
EDX	0500XXXXH	0500XXXXH
EBX、ECX、ESP、EBP、ESI 和 EDI	0	0
EFLAGS	2	2
EIP	0000FFF0H	0000FFF0H
CS	F000H	F000H
DS、ES、FS、GS 和 SS	0	0
GDTR 和 TSS	0	0
CR_0	60000010H	60000010H
CR_2、CR_3 和 CR_4	0	0
DR_3 ～ DR_0	0	0
DR_6	FFFF0FF0H	FFFF0FF0H
DR_7	00000400H	00000400H

注：BIST = built-in self-test（内建自测试），XXXX = Pentium Ⅱ 的版本号。

$\overline{RP}$ **请求奇偶校验（request parity）**。该信号用来请求奇偶校验。

$\overline{RS2}$～$\overline{RS0}$ **请求状态（request status）**输入引脚，用来请求当前的 Pentium Ⅱ 状态。

$\overline{RSP}$ **响应奇偶校验（response parity）**输入引脚。该信号被激活时请求奇偶校验。

$\overline{SLOTOCC}$ **slot 被占用（slot occupied）**输出信号。该信号为逻辑 0 时，表明插槽插有 Pentium Ⅱ 或者等效终端负载。

$\overline{SLP}$ **睡眠（sleep）**输入引脚。当在停机状态下插入该信号时，Pentium Ⅱ 进入睡眠状态。

$\overline{SMI}$ **系统管理中断（system management interrupt）**输入引脚使 Pentium Ⅱ 进入系统管理工作模式。

$\overline{STPCLK}$ **停止时钟（stop clock）**输入引脚。该信号使 Pentium Ⅱ 进入低功耗准予停机状态。

TCK **可测试性时钟（testability clock）**输出引脚，根据 IEEE1149.1 边界检测接口选择时钟操作。

TDI **测试数据（test data）**输入引脚。该信号用来测试由 TCK 信号同步打入 Pentium Ⅱ 的数据。

TDO **测试数据（test data）**输出引脚。该信号用来获得由 TCK 移出 Pentium Ⅱ 的测试数据和指令。

TESTHI **高电平测试（test high）**输入引脚。为了使 Pentium Ⅱ 正常工作，该引脚通过一个 1KΩ～10KΩ 的电阻接到 +2.5V 上。

$\overline{THERMTRIP}$ **温度传感器开关（thermal sensor trip）**。当 Pentium Ⅱ 的温度超过 130℃时该输出变为 0。

TMS **测试方式选择（test mode select）**输入引脚，在测试模式中控制 Pentium 操作。

$\overline{\text{TRDY}}$ **目标就绪（target ready）**输入引脚，用来使 Pentium Ⅱ执行回写操作。

$\overline{\text{VID4}}$～$\overline{\text{VID0}}$ **电压数据（voltage data）**。这组引脚为地/开输出引脚，指示 Pentium Ⅱ需要什么电源电压。电源必须根据 Pentium Ⅱ实际要求的电压提供，如表 19-2 所示。

表 19-2 $\overline{\text{VID}}$引脚要求的加到 V_{cc}上的电源电压

$\overline{\text{VID4}}$	$\overline{\text{VID3}}$	$\overline{\text{VID2}}$	$\overline{\text{VID1}}$	$\overline{\text{VID0}}$	V_{cc}
0	0	0	0	0	2.05V
0	0	0	0	1	2.00V
0	0	0	1	0	1.95V
0	0	0	1	1	1.90V
0	0	1	0	0	1.85V
0	0	1	0	1	1.80V
0	0	1	1	0	—
0	0	1	1	1	—
0	1	0	0	0	—
0	1	0	0	1	—
0	1	0	1	0	—
0	1	0	1	1	—
0	1	1	0	0	—
0	1	1	0	1	—
0	1	1	1	0	—
0	1	1	1	1	—
1	0	0	0	0	3.5V
1	0	0	0	1	3.4V
1	0	0	1	0	3.3V
1	0	0	1	1	3.2V
1	0	1	0	0	3.1V
1	0	1	0	1	3.0V
1	0	1	1	0	2.9V
1	0	1	1	1	2.8V
1	1	0	0	0	2.7V
1	1	0	0	1	2.6V
1	1	0	1	0	2.5V
1	1	0	1	1	2.4V
1	1	1	0	0	2.3V
1	1	1	0	1	2.2V
1	1	1	1	0	2.1V
1	1	1	1	1	—

19.1.1 存储系统

Pentium Ⅱ微处理器的存储系统大小为 64GB，与 Pentium Pro 微处理器的一样。这两种微处理器都可以用 36 位地址总线来访问宽度为 64 位的存储系统。大多数 Pentium Ⅱ系统都采用工作在 66MHz 或 100MHz 的 SDRAM。66MHz 的系统中，SDRAM 的访问时间为 10ns；100MHz 的系统中，SDRAM 的访问时间为 8ns。本章没有说明连接到芯片组上的存储系统。要查看不含 ECC 的 64 位宽存储系统的组织，可参见前面章节。

Pentium Ⅱ的存储系统被分为 8 个或 9 个存储体，每个存储体都存储一个字节的数据。如果有第 9 个字节，那么它将用来存放错误校正码（ECC）。Pentium Ⅱ与 80486～Pentium Pro 一样，采用内部奇偶校验发生和检查逻辑来获得存储系统的数据总线信息（注意，多数 Pentium Ⅱ系统不使用奇偶校验）。如果使用了奇偶校验，每个存储体都包含一个第 9 位。64 位宽的存储器对于双精度浮点型数据是很重要的，因为双精度浮点型数据正好是 64 位宽。与 Pentium Pro 相似，Pentium Ⅱ的存储系统也是以字节方式计数的，从字节 00000000H 到字节 FFFFFFFFH。请注意，目前还没有芯片组支持多于 1GB 的系统

存储器，所以额外的地址连接是为将来系统扩展保留的。图 19-3 展示了采用 AGP 显示适配器的 Pentium Ⅱ 系统的基本存储器映射。

除了 AGP 区使用的存储区以外，Pentium Ⅱ 的存储器映射与前面有关章节中举例说明的存储器映射相似。AGP 区使得视频卡和 Windows 可以在线性地址空间访问视频信息。这与 DOS 下标准 VGA 显示卡不同，DOS 下只有 128KB 窗口。由于 AGP 显示卡不需要通过 128KB 的 DOS 显示存储器分页访问，因而视频更新更快。

Pentium Ⅱ 与存储器之间的传送由 440LX 或 440BX 芯片组控制。Pentium Ⅱ 和芯片组之间的数据传送的宽度是 8 个字节。芯片组通过 5 个$\overline{\text{REQ}}$信号与微处理器通信，如表 19-3 所示。从本质上看，芯片组控制着 Pentium Ⅱ，这与传统上把微处理器直接连接到系统中并直接连接存储器的方法完全不同。

Pentium Ⅱ 只直接连接 cache，cache 在 Pentium Ⅱ 封装盒上。我们已经讲过，Pentium Ⅱ 的 cache 工作频率为微处理器时钟频率的一半。因此，对于 400MHz 的 Pentium Ⅱ，它的 cache 将工作在 200MHz 下。Pentium Ⅱ Xeon 的 cache 工作频率与微处理器的时钟频率一样，这就意味着具有 512KB、1MB 或 2MB cache 的 Pentium Ⅱ Xeon 性能将超过标准的 Pentium Ⅱ。

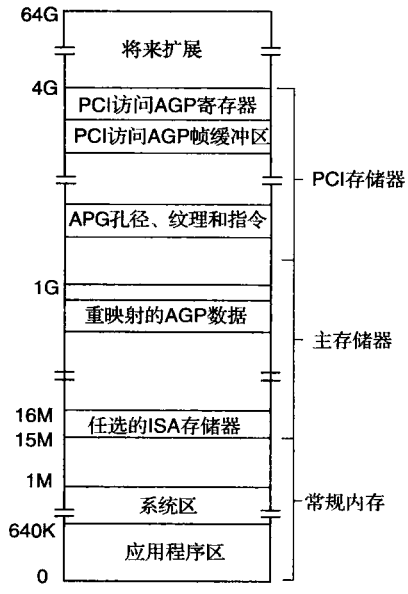


图 19-3 基于 Pentium Ⅱ 微处理器的计算机系统的存储器映射

表 19-3 Pentium Ⅱ 的请求信号（$\overline{\text{REQ}}$）

$\overline{\text{REQ4}}$-$\overline{\text{REQ0}}$	名 称	说 明
00000	延期回答	对先前延期的事务发出延期回答
00001	保留	—
00010	存储器读并无效	从 DRAM 中读存储器或从 PCI 到 DRAM 写
00011	保留	—
00100	存储器代码读	存储器读周期
00101	存储器回写	存储器回写周期
00110	存储器数据读	存储器读周期
00111	存储器写	正常的存储器写周期
01000	中断响应或特殊周期	PCI 总线中断响应周期
01001	保留	—
10000	I/O 读	I/O 读操作
10001	I/O 写	I/O 写操作

19.1.2 输入/输出系统

Pentium Ⅱ 的输入/输出系统与早期的 Intel 微处理器完全兼容。I/O 端口号出现在地址线 A_{15}～A_3，与体使能信号一起用来选择实际用于 I/O 传输的存储体。传输由芯片组控制，这与 Pentium Ⅱ 之前的标准微处理器体系结构不同。

从 80386 微处理器开始，当 Pentium Ⅱ 工作在保护方式下时，I/O 特权信息被添加到 TSS 段，注意，这使得 I/O 端口可以被有选择地禁止。如果一个被锁定的 I/O 地址被访问，Pentium Ⅱ 就产生 13 号中断来指示 I/O 特权冲突。

19.1.3 系统时序

与所有的微处理器一样，为了与微处理器接口，必须要理解系统时序信号。由于 Pentium Ⅱ 被设计成由芯片组控制，因此微处理器与芯片组之间的时序信号成为 Intel 公司专利信息。

19.2 Pentium Ⅱ软件变化

Pentium Ⅱ微处理器的核心是Pentium Pro。这意味着Pentium Ⅱ和Pentium Pro本质上是具有相同软件的器件。本节列出了CPUID指令的变化，并列出了SYSENTER、SYSEXIT、FXSAVE和FXRSTORE指令（这些指令是Pentium Ⅱ在软件方面的惟一修改之处）。

19.2.1 CPUID指令

表19-4列出了Pentium Ⅱ和CPUID指令间传递的值，与早期型号的Pentium微处理器相比这些值有一些变化。

表19-4 CPUID指令

EAX输入	输出寄存器	说明
0	EAX	CPUID指令输入到EAX中的最大值
0	EBX	"uneG"
0	ECX	"Ieni"
0	EDX	"letn"
1	EAX	版本号
1	EDX	特征信息
2	EAX	cache数据
2	EBX	cache数据
2	ECX	cache数据
2	EDX	cache数据

EAX为逻辑0时执行CPUID指令后，版本信息就被返回到EAX中。其中，系列标识返回到8～11位，型号标识返回到4～7位。版本标识返回到0～3位。对于Pentium Ⅱ，型号数为6，系列标识为3。版本编号指的是更新编号数。版本编号越高，表示该型号越新。

EAX为0时执行CPUID指令后，EDX寄存器指示处理器的特征。Pentium Ⅱ只有两个新的特征返回到EDX寄存器中。第11位表示该微处理器是否支持SYSENTER和SYSEXIT这两个新的快速调用指令。第23位表示该微处理器是否支持第14章中所介绍的MMX指令集。其余位与早期型号相同因此没有描述。其中，第16位表示该微处理器是否支持页属性表或PAT。第17位表示该微处理器是否支持Pentium Pro和Pentium Ⅱ中所见到的页大小扩充。页扩充使得高于4G直到64G的存储器都可以被访问。最后，第24位表示是否实现了快速浮点保存（FXSAVE）和快速浮点恢复（FXRSTOR）指令。

19.2.2 SYSENTER和SYSEXIT指令

SYSENTER和SYSEXIT指令使用了Pentium Ⅱ引入的快速调用机制。请注意，这些指令只在保护模式下0级（特权级0）起作用。Windows工作在0级，但不允许应用程序访问0级。由于这些新指令在其他特权级下不起作用，因此这些指令对操作系统软件很有意义。

SYSENTER指令用特定模型寄存器保存CS、EIP和ESP，以执行由特定模型寄存器定义的过程的快速调用。快速调用与常规调用不同，因为它不像常规调用那样将返回地址入栈。表19-5说明了SYSENTER和SYSEXIT指令用到的特定模型寄存器。请注意，特定模型寄存器可以用RDMSR指令读取，用WRMSR指令写入。

表19-5 SYSENTER和SYSEXIT用到的特定模型寄存器

名称	编号	功能
SYSENTER_CS	174H	SYSENTER目标代码段
SYSENTER_ESP	175H	SYSENTER目标堆栈指针
SYSENTER_EIP	176H	SYSENTER目标指令指针

使用RDMSR和WRMSR指令时，要将寄存器号放入ECX寄存器。如果要使用WRMSR，将新数据放在EDS：EAX单元中。对于SYSENTER指令，只需要EAX寄存器，但EDS寄存器为0。如果程序中使用RDMSR指令，数据就被返回到EDX：EAX寄存器对中。

要使用SYSENTER指令，首先要把系统入口点地址装载到特定模型寄存器SYSENTER_CS、SYSENTER_ESP和SYSENTER_EIP中。这一般是操作系统如Windows 2000或Windows XP的入口地址或堆栈区。请注意，这个指令可作为系统指令访问0级的代码或软件。堆栈段寄存器加载的是SYSENTER_CS加8的值。换句话说，由SYSENTER_CS选择子的值所寻址的选择子对被装载到CS和SS。堆栈的偏移值被装载到SYSENTER_ESP中。

SYSEXIT指令把由SYSENTER_CS加16和24所寻址的选择子对装载到CS和SS中。表19-6说明

了来自全局选择子表的选择子，由 SYSENTER_CS 寻址。除了代码段和堆栈段以及它们所表示的存储系统外，SYSEXIT 指令还将 EDX 中的值传给 EIP 寄存器，将 ECX 中的值传给 ESP 寄存器。SYSEXIT 指令将控制返回到应用程序 3 级。我们已经提到，这些指令设计用来快速进入 PC 机上 Windows 或 Windows NT 操作系统，或从操作系统快速返回。

表 19-6 SYSENTER_CS 值所访问的选择子

SYSENTER_CS（MSR 174H）	功 能
SYSENTER_CS 值	SYSENTER 代码段选择子
SYSENTER_CS 值 +8	SYSENTER 堆栈段选择子
SYSENTER_CS 值 +16	SYSEXIT 代码段选择子
SYSENTER_CS 值 +24	SYSEXIT 堆栈段选择子

在使用 SYSENTER 和 SYSEXIT 指令时，SYSENTER 指令必须将返回地址传递给系统，这可以通过将返回地址的偏移地址装载到 EDX 寄存器并将返回地址的段地址放到位于 SYSENTER_CS + 16 的全局描述符表中来完成。堆栈段的传递是通过将堆栈段选择子装载到 SYSENTER_CS + 24 并将 ESP 装载到 ECX 中来完成的。

19.2.3 FXSAVE 和 FXRSTOR 指令

FXSAVE 和 FXRSTOR 是 Pentium Ⅱ 增加的最后两个指令，它们与第 14 章中详述的 FSAVE 和 FRSTOR 几乎相同，主要的区别在于 FXSAVE 设计用来完整保存 MMX 处理机的状态，而 FSAVE 用来完整保存浮点协处理器的状态；FSAVE 指令保存了整个标志域，而 FXSAVE 指令只保存了标志域中的有效位。当 FXRSTOR 指令执行时，有效的标志域用来重构恢复标志域。这意味着，如果要保存机器的 MMX 状态，就要用 FXSAVE 指令；如果要保存机器的浮点状态，就要用 FSAVE 指令。对于一些新的应用，建议使用 FXSAVE 和 FXRSTOR 指令保存机器的 MMX 状态和浮点状态。在新的应用中不要再使用 FSAVE 和 FRSTOR 指令。

19.3 Pentium Ⅲ

Pentium Ⅲ微处理器是 Pentium Ⅱ微处理器的改进型。虽然型号比 Pentium Ⅱ新，但仍然是基于 Pentium Pro 体系结构。

Pentium Ⅲ有两种型号：一种型号具有 512KB 非阻塞 cache，封装在 slot1 封装盒中；另一种型号具有 256KB 预传送 cache，封装在集成电路中。slot1 型的 cache 运行速度是处理器速度的一半，集成 cache 以处理器的时钟频率运行。正如许多 cache 性能的标准测试程序（benchmark）所显示的，cache 大小从 256K 字节增加到 512K 字节时，性能的改善仅有几个百分点。

19.3.1 芯片组

Pentium Ⅲ的芯片组与 Pentium Ⅱ的不同。Pentium Ⅲ使用 Intel 的 810、815 或者 820 芯片组。在比较新的使用 Pentium Ⅲ的系统中最常见的是 815。其他几个厂家的芯片组也可以使用，但新外设（例如视频卡）的驱动问题已经被报道。840 芯片组也是为 Pentium Ⅲ开发的，但 Intel 没有生产。

19.3.2 总线

Coppermine 型号的 Pentium Ⅲ将总线速度增加到 100MHz 或 133MHz。这个加快型号使微处理器和存储器之间可以以更高的速度传输数据。Pentium Ⅲ发行的最后的版本是具有 133MHz 总线的 1GHz 的微处理器。

假如你有 1GHz 的微处理器，使用的是 133MHz 的存储器总线，你也许会认为加快存储器的总线速度能提高性能，我们同意你的看法。然而，微处理器和存储器之间的接线却阻碍了使用更高速度访问存储器。如果我们决定使用 200MHz 的总线速度，我们必须考虑到 200MHz 时的波长为 300 000 000/200 000 000即 3/2 米。天线是波长的 1/4。在 200MHz 时，天线为 14.8 英寸。为了在 200MHz 下不放射能量，要保持印刷电路板上的连线短于 1/4 波长。在实践中，要保持连线不超过 1/4 波长的 1/10。这就意味着 200MHz 的系统中的连线应该不超过 1.48 英寸长。为 200MHz 的存储系统放置插座时，这个尺寸将给主板制造商带来问题。200MHz 总线系统可能是技术的极限了。如果调整总线，可能有办法提

高频率，时间将确定这是否可能。现在做的只是在广告中玩弄字眼，如800MB/s评价总线（由于每次传送64位（8字节），所以800MB每秒实际上是100MHz）。

存储系统有可能接近或超过200MHz吗？如果我们开发出新的微处理器、芯片组和存储器之间的互连技术，那么回答是肯定的。目前，每次读主存储器时，存储器以4个64位数的猝发串方式操作。这32字节的猝发串被读到cache中。在100MHz时，主存储器访问第一个64位数需要3个等待状态，接着剩余的三个64位数每个都只要0个等待状态，总体需要7个100MHz的总线时钟。这意味着以每字节70ns/32 = 2.1875ns的速度读数据，总线速度为457MB/s。这个速度比1GHz的微处理器要慢，但是，大部分的程序都是循环的，并且指令已存放到内部的cache中，因此能够并且经常接近微处理器的工作频率。

19.3.3 引脚

图19-4显示了socket 370型的Pentium Ⅲ微处理器的引脚图。该集成电路封装在370引脚的PGA中。它被设计成与Intel的一个芯片组一起工作。除了完全版的Pentium Ⅲ，还有使用66MHz存储器总线的Celeron。Intel同样制造了Pentium Ⅲ的Xeon，为服务器应用程序提供大的Cache。

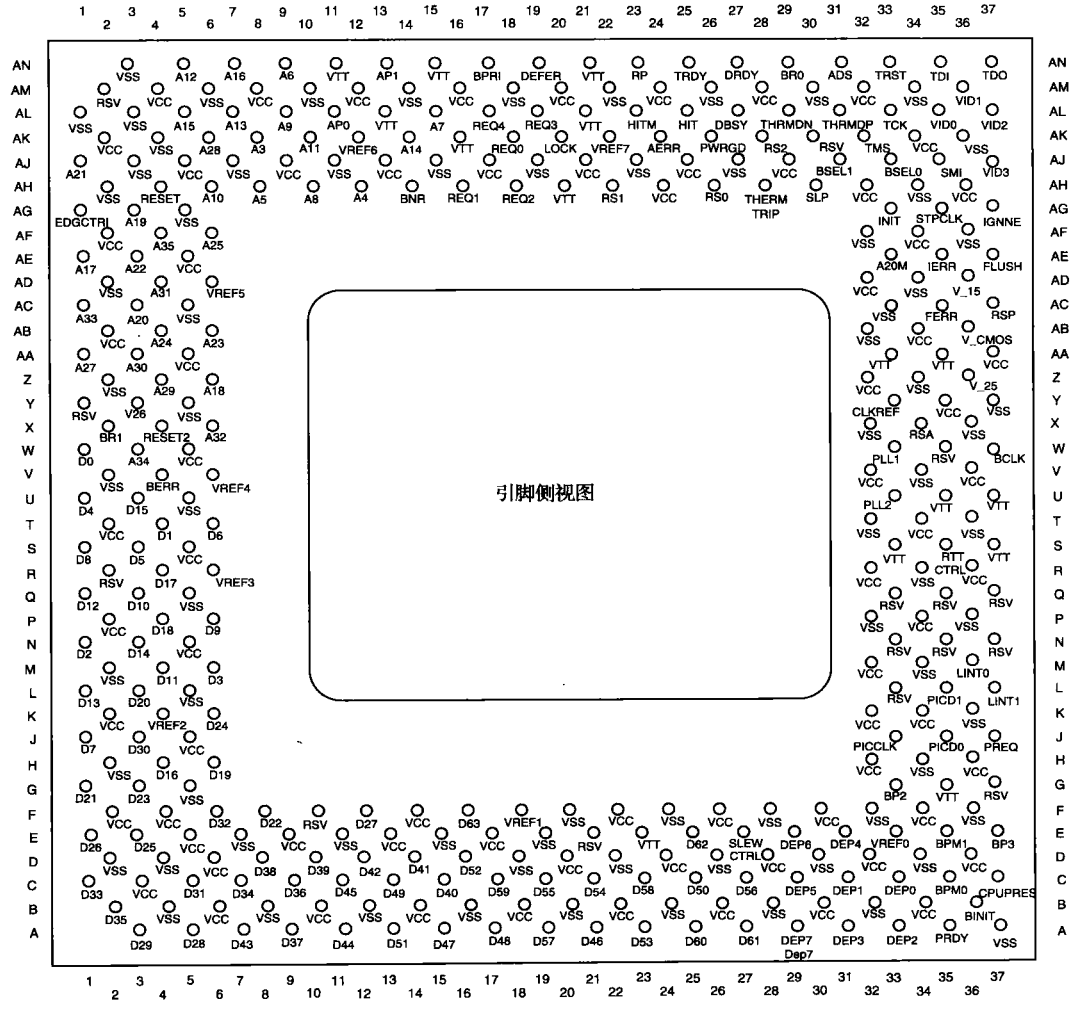


图19-4 socket 370型Pentium Ⅲ微处理器引脚图（由Intel公司提供）

19.4 Pentium 4 和 Core2

Intel 最新型号的 Pentium Pro 体系结构的微处理器是 Pentium 4 和最近的 Core2 微处理器。迄今为止，Pentium Ⅱ、Pentium Ⅲ、Pentium 4 和 Core2 都是 Pentium Pro 体系结构的版本。Pentium 4 最初在 2000 年 11 月发行，速度是 1.3GHz。目前它的速度可达到 3.8GHz。这种集成的微处理器有两种封装，423 引脚的 PGA 封装和 478 引脚的 FC-PGA2 封装。这两种型号都采用 0.18 微米的制造工艺。最新型号用到了 0.13 微米技术或 90nm（0.09 微米）技术。最新型号的 Pentium 4 使用 775 引脚 LGA 封装技术并且具有 775 个引脚。Intel 正在为将来的产品开发 45nm 技术。和早期的 Pentium 型号一样，Pentium 4 使用 100MHz 的存储器总线速度，但由于四倍加速，总线速度可以接近 400MHz。最近的型号使用 133MHz 总线，由于四倍加速，标称为 533MHz，或者 200MHz 总线标称为 800MHz。一些最新的型号使用 1033MHz 或 1333MHz 总线，最新型号 Xeon 中已经使用一种称为 LGA 771 的封装技术。图 19-5 给出了 423 引脚 PGA 封装的 Pentium 4 微处理器的引脚图。

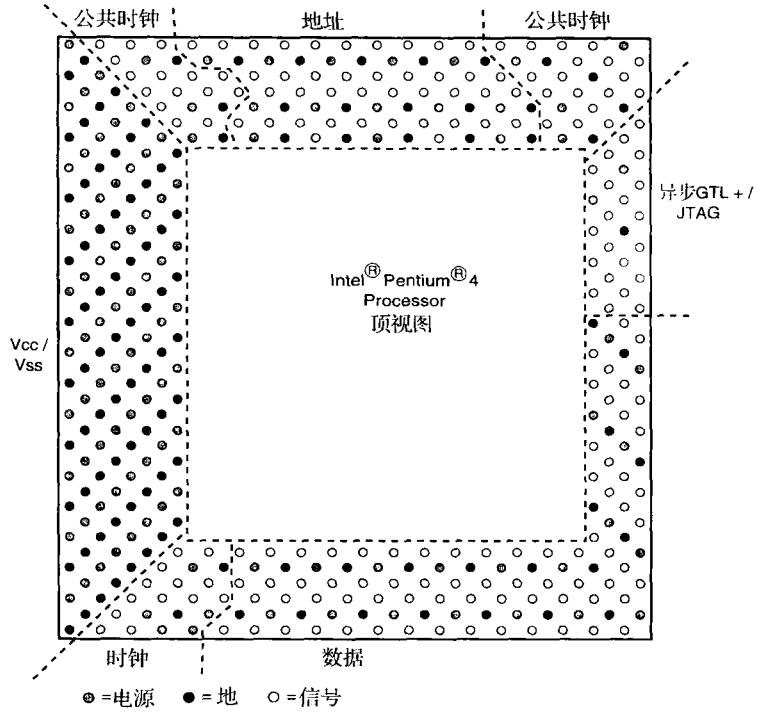


图 19-5 423 引脚 PGA 封装的 Pentium 4 的引脚图（由 Intel 公司提供）

19.4.1 存储器接口

存储器与 Pentium 4 的接口典型使用 Intel 945、965 或 975 芯片组。这些微处理器提供了双通道的存储器总线，每个总线通道与 32 位宽度的存储器接口。两个通道一起组成 64 位宽的到微处理器的数据通道。由于双通道排列，存储器必须由工作在 600MHz、800MHz、1033MHz 的 DDR2 存储器件对构成。根据 Intel DDR2 所述，这种排列与由 PC-100 存储器构成的存储器相比，可以提高 300% 的速度。

Intel 在 965 和 975 芯片组中已经放弃 RDRAM 而使用 DDR2（double data rate）存储器。显然声称 RDRAM 速度 300% 的增加缺乏实际的证明。除了支持 DDR2 外，还增加了支持 SATA 磁盘接口的存储器。

像 945 和 965 这样的新型芯片组包含 PCI-Express 接口而没有 AGP 接口。AGP 接口被 PCI-E 接口所取代以支持视频。IDE 提供对 HDD、CD-ROM 和 DVD 驱动这样的老设备的接口支持。

19.4.2 寄存器组

除了 MMX 寄存器与浮点寄存器分开之外，Pentium 4 和 Core2 的寄存器组几乎与其他型号的 Pentium 相同。另外，增加了 8 个 128 位宽的 XMM 寄存器，用于第 14 章中说明的 SIMD（single instruction multiple data）指令和扩展的 128 位满双精度浮点数。

可以把 XMM 寄存器看做两倍宽度的 MMX 寄存器，它能够保存一对 64 位的双精度浮点数或者 4 个单精度浮点数。正如 MMX 寄存器可以保存 8 字节宽的数一样，同样，XMM 寄存器可以保存 16 字节宽的数。XMM 寄存器是两倍宽度的 MMX 寄存器。

如果从 Microsoft 下载了 MASM 6.15 新补丁，程序就可以用 MMX 和 XMM 指令汇编。ML.EXE 程序也可以在 Microsoft Visual Studio.NET 2003 下找到。汇编包含 MMX 指令的程序时，用 .MMX 开关。对于包含 SIMD 指令的程序，则用 .XMM 开关。例 19-1 给出了用 MMX 指令将两个 8 字节宽的数加在一起的非常简单的程序。注意如何用 .MMX 开关来选择 MMX 指令集。MOVQ 指令用来在存储器和 MMX 寄存器间传送数据。MMX 寄存器从 MM_0 到 MM_7 编号。如果从 Microsoft 下载了 Visual Studio 6.0 版的最新补丁或使用更新的 Visual Studio，那么就可以在 Microsoft Visual C++ 中利用内嵌汇编使用 MMX 和 SIMD 指令。这里推荐使用 Visual Studio Express，其中包括用于软件开发的补丁。

例 19-1

```
.MMX
.DATA
        DATA1   DQ      1FFH
        DATA2   DQ      101H
        DATA3   DQ      ?
.CODE
        MOVQ    MM0,DATA1
        MOVQ    MM1,DATA2
        PADDB   MM0,MM1
        MOVQ    DATA3,MM0
```

同样地，XMM 软件在程序中用 .XMM 开关。最新的程序用 XMM 寄存器和 XMM 指令集实现多媒体和其他高速操作。例 19-2 给出的短程序举例说明了一些 XMM 指令的使用。这个程序将每组由四个单精度浮点数组成的两组数相乘并将四个结果存放在 ANS 指向的四个双字中。为了能够访问八个字（128 位宽的数），我们使用了 OWORD PTR 伪指令。此外要注意 FLAT 模式与 C 剖面（profile）一起用。由于 SIMD 指令只能在保护模式下执行，因此我们将程序定义为 FLAT 模式的格式。这就意味着 .686 和 .XMM 开关必须在 MODEL 语句之前。

例 19-2

```
.686
.XMM
.MODEL FLAT,C
.DATA
        DATA1   DD  1.0         ;为DATA1定义4个浮点数
                DD  2.0
                DD  3.0
                DD  4.0
        DATA2   DD  6.3         ;为DATA2定义4个浮点数
                DD  4.6
                DD  4.5
                DD  -2.3
        ANS     DD  4 DUP(?)
.CODE
        MOVAPS XMM0,OWORD PTR DATA1
        MOVAPS XMM1,OWORD PTR DATA2
        MULPS  XMM0,XMM1
        MOVAPS OWORD PTR ANS,XMM0

;其他代码放在这里

END
```

19.4.3 超线程技术

被称为**超线程**（**hyper-threading**）的技术是 Pentium 最新的创新。这个重大的进步将两个微处理器封装合并为单个。为了理解这项新技术，参见图 19-6，它给出了传统的双处理器系统和超线程系统。

超线程处理器含有两个执行单元，每个单元含有一整套寄存器，能够独立或并发地运行软件。这两个独立的机器上下文共享一个公共总线接口单元。在机器运行期间，每个处理器可以独立运行一个线程（进程），提高了用多线程编写的应用程序的执行速度。总线接口单元含有 2 级和 3 级高速缓存以及到存储器和系统 I/O 结构的接口。当任意一个微处理器需要访问存储器或 I/O 时，必须共享总线接口单元。

总线接口单元用来访问存储器，但由于存储器以填充高速缓存的猝发方式访问，所以经常是空闲的。因此，当第一个处理器正忙于执行指令时，第二个处理器可以利用这个空闲时间访问存储器。那么系统的速度是否加倍了呢？是也不是。只要不访问存储器同一区域，线程就能相互独立运行。如果每个线程都访问存储器的同一区域，具有超线程技术的机器实际上运行得更慢。这并不经常发生，因此，在大多数情况下超线程系统几乎达到了与双处理器系统相同的性能。

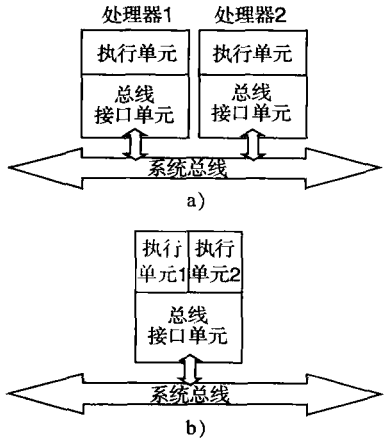


图 19-6 图解说明双处理器和超线程处理器系统
a）双处理器系统处理器 b）超线程系统

最终大部分机器都将使用超线程技术，这意味着要更加关注多线程软件的开发。在有双处理器或超线程处理器的系统中，每一个线程运行在不同的处理器上，从而提高了性能。在将来，系统结构可以包含更多的处理器处理额外的线程。

19.4.4 多核技术

Pentium 4 和 Core2 的大多数新型号要么是双核要么是四核。每个核都是一个单独的微处理器，独立执行不同的任务。目前有三种型号：Pentium D，包含双核，每个核带有不同的缓存；Core2 Duo，它共享缓存但是有双核；以及包含 4 个核的四核型号。Intel 看来为多核微处理器提供了共享缓存。最近的一份来自 Intel 的报告中指出，在将来，Pentium 或者无论叫做什么的芯片都将可能包含多达 80 个核。Core2 Duo 的缓存是 2MB 或者 4MB，主频为 3GHz。这样看来，速度竞争结束了，时钟频率稳定在 3GHz ~4GHz。这是否意味着 5GHz 的型号在将来不会出现了呢？是这样的，至少目前来看，不可能出现更高频率的型号。如此看来，使用线程应用的多核技术将成为发展前景。硅技术似乎达到了其顶峰。这意味着，高效的编程将会成为提高计算机系统速度的途径。

19.4.5 CPUID

和早期 Pentium 型号一样，CPUID 指令访问表示处理器类型的信息以及被微处理器支持的特征信息。在不断进化的微处理器系列中，能够访问这些信息是非常重要的，这样，可以编写出高效的软件并能够运行在诸多不同型号的微处理器上。

表 19-7 列出了 CPUID 指令最新的特征。要访问这些特征，给 EAX 寄存器装入表中列出的数字，接着执行 CPUID 指令。在实模式或保护模式下，CPUID 指令通常将信息返回到 EAX、EBX、ECX 和 EDX 寄存器。正如表中资料显示的，与以前的型号相比，CPUID 指令增加了附加的特征。

表 19-7 Pentium 4 CPUID 指令

EAX 输入值	输出寄存器	注 释
0	EAX = 最大输入值	“GenuineIntel” 按从小到大格式返回
	EBX = “uneG”	
	ECX = “Ieni”	
	EDX = “letn”	

（续）

EAX 输入值	输出寄存器	注　释
1	EAX = 版本信息	特征信息
	EBX = 特征信息	
	ECX = 扩展特征信息	
	EDX = 特征信息	
2	EAX、EBX、ECX 和 EDX	Cache 和 TLB 信息
3	ECX 和 EDX	只是 Pentium Ⅲ才有的序列号
4	EAX、EBX、ECX 和 EDX	确定性 cache 参数
5	EAX、EBX、ECX 和 EDX	监视/等待信息
80000000H	EAX	扩展功能信息
80000001H	EAX	保留
80000002H、80000003H 和 80000004H	EAX、EBX、ECX 和 EDX	处理器商标字符串
80000006H	ECX	Cache 信息

在第 18 章开发了 EAX = 1 时调用 CPUID 指令后读和显示可用数据的软件。这里我们处理读处理器商标字符串并在 Visual C++ 函数中显示。如果支持，商标字符串包含了微处理器保证工作的频率和真实的 Intel 关键字。BrandSring 函数（见例 19-3）返回一个 CString，该对象含有存储在 CPUID 成员 80000002H ～ 80000004H 的信息。该软件需要 Pentium 4 系统正确运行来测试 BrandString 函数。Convert 函数从返回参数寄存器 EAX、EBX、ECX 和 EDX 中读取内容，并将其转换成 CString 返回。作者的系统显示的商标字符串是：

“Intel（R）Pentium（R）4 CPU 3.06GHz”

例 19-3

```
int getCPU(int EAXvalue)
{
      int temp;
      _asm
      {
            mov  eax,EAXvalue
            cpuid
            mov  temp1,eax
      }
      return temp;
}

private: System::String^ BrandString(void)
{
      String^ temp;
      int temp1 = getCpu(0x8000000);
      if ( temp1 >= 0x80000004 )     // 如果商标串存在
      {
            temp += Convert(0x80000002);        // 读寄存器 80000002H
            temp += Convert(0x80000003);        // 读寄存器 80000003H
            temp += Convert(0x80000004);        // 读寄存器 80000004H
      }
      return temp;
}

private: System::String^ Convert(int EAXvalue)
{
      CString temp ="                    ";  // 必须是 16 个空格
      int temp1, temp2, temp3, temp4;
      _asm
      {
            mov  eax,EAXvalue
            cpuid
            mov  temp1,eax
            mov  temp2,ebx
            mov  temp3,ecx
```

```
            mov  temp4,edx
        }
        for ( int a = 0; a <4; a++ )
        {
            temp.SetAt(a, temp1);
            temp.SetAt(a + 4, temp2);
            temp.SetAt(a + 8, temp3);
            temp.SetAt(a + 12, temp4);
            temp1 >>= 8;
            temp2 >>= 8;
            temp3 >>= 8;
            temp4 >> =8;
        }
        return temp;
}
```

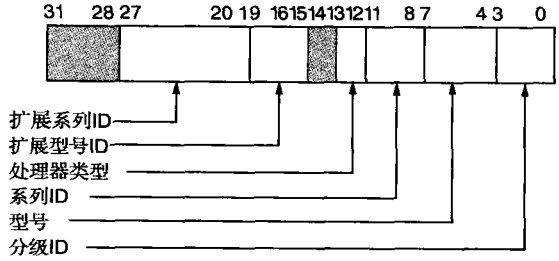


图 19-7 执行 CPUID 指令后 EAX 显示的版本信息

在 EAX 装入 1 并执行 CPUID 后，其他有关系统的信息返回到 EAX、EBX、ECX 和 EDX 中。EAX 寄存器包含版本信息，如型号、系列、分级信息，如图 19-7 所示。EBX 寄存器包含关于高速缓存的信息，如 15～8 位是 CFLUSH 指令清空的高速缓存行大小，31～24 位为复位时分配给局部的 APIC 的 ID，23～16 位表示有多少内部处理器可用于超线程（当前的 Pentium 4 处理器是 2 个）。例 19-4 展示了在超线程 CPU 中确定处理器个数的函数，并用字符串返回结果。如果最终多于 9 个处理器增加到微处理器上，例 19-4 中的软件就需要作修改。

例 19-4

```
CString CCPUIDDlg::GetProcessorCount(void)
{
        CString temp = "This CPU has ";
        char temp1;
        _asm
        {
                mov  eax,1
                cpuid
                mov  temp1,31h
                bt   edx,28            ;检查超线程
                jnc   GetPro1          ;如果没有超线程，则temp1 = 1
                bswap ebx
                add  bh,30h
                mov  temp1,bh
GetPro1:        }
        return temp + temp1 + " processors.";
}
```

微处理器的特征信息返回到 ECX 和 EDX 中，如图 19-8 和图 19-9 所示。如果特征出现，对应那一位为 1。例如，如果应用程序需要超线程，通过测试 EDX 中的第 28 位看是否支持超线程。该功能的代码与读超线程微处理器中的处理器个数的代码一起出现在例 19-4 中。BT 指令测试指定位并将结果放在进位标志中。如果该位测试为 1，作为结果的进位标志为 1；如果该位测试为 0，作为结果的进位标志为零。

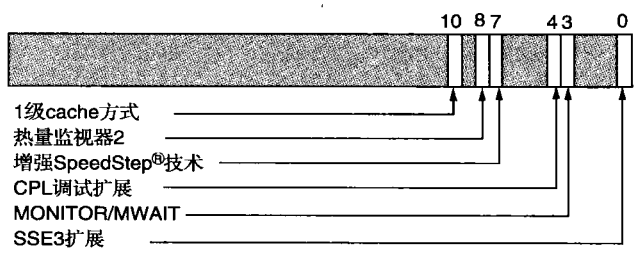


图 19-8 执行 CPUID 指令后 ECX 显示的版本扩展信息

注：1 表示支持扩展。

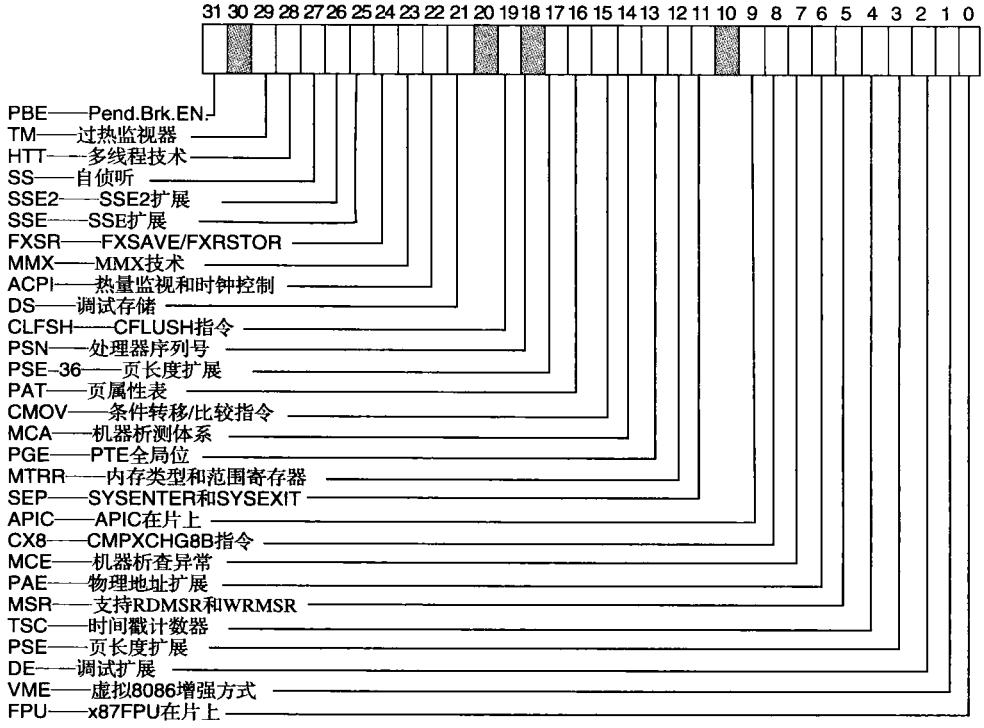


图 19-9 执行 CPUID 指令后 EDX 显示的版本扩展信息

注：1 表示支持扩展。

19.4.6 特定模型寄存器

与早期的 Pentium 处理器一样，Pentium 4 和 Core2 也有特定模型寄存器，读用 RDMSR 指令，写用 WRMSR 指令。Pentium 4 和 Core2 有 1743 个特定模型寄存器，从 0H 到 6CFH 编号。Intel 没有提供它们的全部信息。未确定的寄存器由 Intel 保留或用于一些没有文档说明的特征或功能。

读和写特定模型寄存器指令以同样方式工作。ECX 加载要访问的寄存器编号，数据通过 EDX：EAX 寄存器对作为 64 位数传递，EDX 是 32 高有效位，EAX 是低有效位。这些寄存器必须在实模式（DOS）或保护模式的 0 级下访问。这些寄存器通常由操作系统访问，一般的 Visual C++ 程序不能访问。

19.4.7 性能监视寄存器

Pentium 4 的另外一个特点是有一组性能监视寄存器（PMR），如同特定模型寄存器一样，只能用在实模式或者保护模式的 0 级下。用户软件可以直接访问的寄存器仅有时间戳计数器，这是一个性能

监视寄存器。剩下的 PMR 用 RDPMR 访问。这条指令与 RDMSR 指令相似，用 ECX 指定寄存器号，结果出现在 EDX：EAX 中。PMR 没有写指令。

19.4.8 64 位扩展技术

Intel 已经发布了它的支持 Intel 32 位架构系列的绝大多数最新的 64 位扩展技术。指令集和体系结构向下与 8086 兼容，这意味着指令和寄存器组保持兼容。唯一不兼容的是一些逻辑指令以及一些处理 AH、BH、CH 和 DH 的指令。所变的就是寄存器组在宽度上取代了现在的 32 位宽的寄存器而变宽到 64 位。64 位模式下 Pentium 4 和 Core2 的编程模式参见图 19-10。

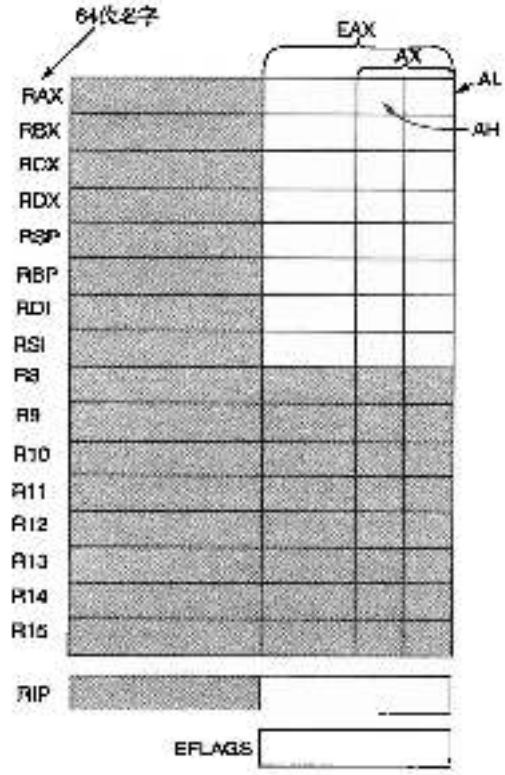


图 19-10 64 位模式的 Pentium 4 微处理器的整数寄存器组

注：阴影部分为 Pentium 4 处在 64 位模式下的寄存器。

注意现在寄存器组包括了 16 个 64 位宽的通用寄存器，RAX，RBX，RCX，RDX，RSP，RBP，RDI，RSI，$R_8 \sim R_{15}$。指令指针也变成 64 位宽。使处理器用 64 位存储器地址寻址存储器。这使得处理器可以寻址的存储器与处理器地址引脚指定实现的一样多。

寄存器可以按 64 位、32 位、16 位或者 8 位寻址。一个例子，R_8（64 位）、R8D（32 位）、R8W（16 位）和 R8L（8 位）。没有办法寻址编号寄存器的高字节（如 BH），编号寄存器只有低字节可以被访问。原先的如 MOV AH，AL 的访问工作正常，但是寻址一个原先的高字节和编号低字节寄存器是不允许的。换句话说，MOV AH，R9L 是不允许的，但是 MOV AL，R9L 是允许的。如果 MOV AH，R9L 指令包含在程序中，不会出现错误，取代的是，指令将被改变为 MOV BPL，R9L。AH、BH、CH 和 DH 分别变为 BPL，SPL，DIL，SIL 低 8 位（L 是低 8 位）。其他方面，原先的寄存器能与新的编号寄存器 $R_8 \sim R_{15}$ 混合使用，如 MOV R_{11}，RAX，MOV R11D，ECX 或 MOV BX，R14W。

在体系结构上另外增加了一组附加的 SSE 寄存器，编号为 $XMM_8 \sim XMM_{15}$。这些寄存器通过 SSE，SSE_2 和 SSE，指令访问。在其他方面，SSE 部件没有变化。控制和调试寄存器被扩展到 64 位宽。在地址 C0000080H 处增加了一个新的特定模型寄存器，以控制扩展特征。图 19-11 描述了控制寄存器的扩展特征：

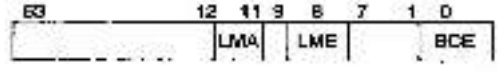


图 19-11 扩展特征特定模型寄存器的内容

SCE 系统调用使能（system CALL enable）位，置位使能 64 位模式下 SYSCALL 和 SYSRET 指令。

LME 模式使能（mode enable）位，置位使能微处理器使用 64 位扩展模式。

LMA 模式激活（mode active）位。该位显示微处理器正工作在 64 位扩展模式下。

在扩展的 64 位模式下，保护模式的描述符表寄存器也被扩展，GDTR、LDTR、IDTR 和任务寄存器（TR）含有 64 位基地址，替代 32 位基地址。最大的变化就是不必关注基地址和段描述符限制。系统为代码段用一个基地址 0000000000000000H，DS、ES 和 SS 段则忽略基地址。

分页也作了修改，包括了支持 64 位线性地址到 52 位物理地址转换的分页单元。Intel 声明在 64 位 Pentium 第一个型号中，线性地址将是 48 位，物理地址将是 40 位。这就意味着 40 位地址支持从 256T（tera）字节的线性空间转换到 1T 字节的物理存储器。52 位地址可寻址 4P（peta）存储器字节。64 位

线性地址寻址 16E（exa）字节的存储器。转换由分页单元的附加表来完成。64 位扩展分页单元用四层页表取代了两表（页目录和页表）。

19.5 小结

1）Pentium Ⅱ与早期的微处理器不同，因为它不再以集成电路形式出现了。Pentium Ⅱ封装在插入式的封装盒即印刷电路板上。

2）PentiumⅡ的二级 cache 安装在封装盒内，Celeron 除外，它没有二级 cache。cache 的速度是PentiumⅡ时钟速度的一半，Xeon 除外，它与 PentiumⅡ的速度相同。所有型号的 PentiumⅡ内部都含有一级 cache，cache 中可以存储 32KB 的数据。

3）Pentium Ⅱ是 Intel 的第一个由外部总线控制器控制的微处理器。早期的微处理器读写信号都是由微处理器产生的，而 Pentium Ⅱ是通过外部总线控制器命令读写信息的。

4）Pentium Ⅱ工作的时钟频率为 233MHz 到 450MHz，总线速度为 66MHz 或 100MHz。第二级 cache 的大小可以为 512KB、1MB 或 2MB。Pentium Ⅱ含有 64 位的数据总线和 36 位的地址总线，可以访问最多 64GB 的存储器。

5）SYSENTER、SYSEXIT、FXSAVE 和 FXRSTOR 是 Pentium Ⅱ新增加的指令。

6）优选 SYSENTER 和 SYSEXIT 命令用来从特权级 3 访问处于特权级 0 的操作系统。这些指令的操作速度比任务切换甚至比调用和返回的组合还要快很多。

7）优选 FXSAVE 和 FXRSTOR 指令用来完整保存 MMX 技术单元和浮点协处理器的状态。

8）Pentium Ⅲ微处理器是 Pentium Pro 体系结构的扩展，增加了使用 XMM 寄存器的 SIMD 指令集。

9）Pentium 4 和 Core2 微处理器是 Pentium Pro 体系结构的扩充，由于采用了 0.13 微米和最新的 45nm 的制造工艺，它的工作时钟频率比以前最高时钟频率还要高。

10）为了在系统中正常工作，Pentium 4 和 Core2 微处理器需要搭配新的 ATX 电源和机箱。

11）MASM 的 6.15 版本和 Visual Studio 6 用 .686 开关与 .MMX 和 .XMM 开关支持新的 MMX 和 SIMD 指令。

12）Pentium Ⅱ、Pentium Ⅲ、Pentium 4 和 Core2 微处理器是 Pentium Pro 微处理器的变种。

13）据称未来的 Pentium 4 和 Core2 微处理器将对 32 位体系机构使用 64 位扩展。在系统中采用多于 4GB 的内存将会很重要。

19.6 习题

1. Pentium Ⅱ微处理器中第一级 cache 的大小是多少？
2. Pentium Ⅱ微处理器中第二级 cache 的大小是多少？列出所有型号。
3. 基于 Pentium 的系统和基于 Pentium Ⅱ的系统中的第二级 cache 有何区别？
4. Pentium Pro 和 Pentium Ⅱ中的第二级 cache 有何区别？
5. Pentium Ⅱ Xeon 中的二级 cache 的速度是 Pentium Ⅱ（不包括 Celeron）的_____倍。
6. Pentium Ⅱ可寻址多大的存储器空间？
7. 有以集成电路形式封装的 Pentium Ⅱ吗？
8. Pentium Ⅱ的封装盒有多少引脚接点？
9. PICD 控制信号有何用途？
10. Pentium Ⅱ的读写引脚有何变化？
11. Pentium Ⅱ运行的总线速度是多少？
12. 连接到总线速度为 100MHz 的 Pentium Ⅱ系统的 SDRAM 有多快？
13. 如果使用 ECC，Pentium Ⅱ的存储器有多宽？
14. Pentium Ⅱ微处理器增加了哪些新的特定模型寄存器（MSR）？
15. Pentium Ⅱ微处理器增加了哪些新的 CPUID 识别信息？
16. 特定模型寄存器是怎样被寻址的？用什么指令可以读它？
17. 编写往特定模型寄存器 175H 中存储 12H 的软件。
18. 编写一小段程序，确定微处理器是否含有 SYSENTER 和 SYSEXIT 指令。如果有该指令，该程序将进位标志置位并返回；否则，清除进位标志位并返回。
19. 当用 SYSENTER 指令时，返回地址是怎样传递给系统的？
20. 当用 SYSEXIT 指令返回到应用程序时，返回地址是如何重新得到的？
21. SYSENTER 指令将控制移交给什么优先级的软件？
22. SYSEXIT 指令将控制移交给什么优先级的软件？
23. FSAVE 和 FXSAVE 指令之间有何区别？
24. Pentium Ⅲ是_____体系结构的扩展。
25. 在 Pentium Ⅲ微处理器上出现的而在 Pentium Pro 微处理器上未出现的新指令有哪些？
26. Pentium 4 或 Core2 微处理器要求电源有什么变化？
27. 写一个短程序，读取 Pentium Ⅲ的微处理器的序列号并在屏幕上显示。
28. 开发一个短的 C++ 函数判断 Pentium 4 是否支持超线程技术，如果支持返回布尔值 true，否则返回布尔值 false。
29. 开发一个短的 C++ 函数判断 Pentium 4 或 Core2 是否支持 SSE、SSE_2 和 SSE_3，如果支持返回布尔值 true，否则返回布尔值 false。
30. 用自己的话比较超线程与双处理器操作，假定包括超过四个以上的处理器是可能的。
31. Core2 处理器指的是什么？

附录 A　汇编程序、Visual C ++ 和 DOS

本附录介绍在 DOS 环境下和在 Visual C++ 环境下如何用汇编程序开发程序。DOS 环境实质上已退出历史舞台（只有 Windows98 仍然还在使用），但是它凭借 Microsoft 公司 Windows 的附件文件夹中被称为 CMD. EXE 的控制台程序仍然在发挥作用。可能有人为 DOS 的离开而流泪，但是想一想 DOS 环境有那么多麻烦事，许多人在它上面花费多年进行编程。DOS 仅有 1MB 存储系统，它的驱动软件也是个问题，尤其是在最近几年。Microsoft 公司从来没有提供像样的保护模式 DOS。DOS 在显示文本信息方面表现很好，但由于 DOS 视频存储结构存在问题和缺乏驱动程序的支持，图形显示则不尽如人意。

Windows 解决了许多折磨人的 DOS 问题，并迎来了 GUI 时代，它是在 DOS 基于文本应用的基础上的巨大改进。Windows 恰恰是要让个人非常容易地去使用和控制计算机。作者记得以前不得不写很多批处理文件，这样妻子才能使用他的计算机。由于有了 Windows，现在她已成为一位真正的计算机专家。为什么呢？——她可以在网上冲浪，现在她真的很专业了。她可以发送一封邮件，无需粘贴图片或其他任何附件，Windows 是一个巨大的易于使用的系统。

A. 1　汇编程序

尽管汇编程序不经常作为独立操作的程序设计工具使用，但在开发链接到 Visual C++ 的程序模块（见第 7 章）时仍会应用它。程序本身由 Visual C++ 提供，在 C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\bin 目录下，文件名是 ML. EXE。在同一目录里还有建库程序 LIB. EXE 和用于链接目标模块的 LINK 程序。

例 A-1 说明如何汇编一个用汇编语言写的程序。这个例子用一个称作 WOW. TXT 的文件（它不需要用 . ASM 扩展名，尽管 . ASM 经常用在汇编语言模块中）。编译文件 WOW. TXT 时，用错误开关小写的 c（/c）和生成 WOW. LST 列表文件开关（/Fl）。如果需要其他开关就在命令提示符处键入：ML/?，显示一张开关表。也可以键入：ML/c/coff WOW. TXT。包含/coff（C 目标文件格式）开关，以便产生目标文件可以链接到 Visual C++ 程序。

例 A-1

```
ML /c /FlWOW. LST WOW. TXT
```

表 A-1　常用于汇编程序的模型

类　型	说　明
. TINY（微）型	所有数据及代码装入同一个 64KB 代码段内。此模型的程序按 DOS . COM 文件格式汇编，要求程序代码从地址 0100H 处开始存放
. SMALL（小）型	这种模型包含两个段：一个数据段和一个代码段。此模型的程序产生 DOS. EXE 文件，且从 0000H 开始存放
. FLAT（平展）型	平展型模型使用单一 4GB 长的存储段，该模型只工作在 Windows 中，程序从 00000000H 开始存放

如果使用 Visual C++ 的 LINK 程序，因为它是一个 32 位的链接器，不能产生 DOS 兼容的执行文件，DOS 的 16 位链接器不在 Visual Studio 软件包中。如果必须开发 DOS 软件，要从 Microsoft 公司获得 Windows 驱动程序开发套件（Windows DDK）。DDK 包含开发 DOS 应用所必须的 16 位链接器。该链接器位于 DDK 文件夹，C:\WINDDK\2600. 1106\bin\win_me\bin16。除了以 CL. EXE 出现的面向 DOS 的

16 位版本 C++ 语言的链接器以外，这里还提供一些应用程序。

例 A-2 显示如何去链接一个由汇编产生的程序。这里假设正在使用 16 位 DOS 实模式链接器程序。32 位链接器通常通过 Windows 应用的 Visual C++ 来使用。这里由例 A-1 产生的目标程序被链接生成名为 WOW. EXE 的可执行程序，或者，如果是 TINY 模型的话，实际生成的是 WOW . COM。

例 A-2

```
LINK WOW. OBJ
```

A. 2 汇编存储模型

尽管平展模型经常用于 Visual C++，但也还有其他存储器模型用于 DOS 应用和嵌入式程序开发。对于这些应用，表 A-1 列出最常用的模型。在 DOS 程序中，开始存放位置由 . STARTUP 伪指令设置，而在平展程序中是自动设置。

表 A-2 列出表 A-1 中各个模型的缺省信息。如果需要有关模型的其他信息，请访问 Microsoft 公司的网站并查找汇编模型。

表 A-2 . MODEL 伪指令所用的默认值

存储模型	伪指令	段名	对齐类型	组合类型	类别	组别
. TINY	. CODE	_TEXT	Word	PUBLIC	'CODE'	DGROUP
	. FARDATA	FAR _DATA	Para	Private	'FAR_DATA'	
	. FARDATA?	FAR_BSS	Para	Private	'FAR_BSS'	
	. DATA	_DATA	Word	PUBLIC	'DATA'	DGROUP
	. CONST	CONST	Word	PUBLIC	'CONST'	DGROUP
	. DATA?	_BSS	Word	PUBLIC	'BSS'	DGROUP
. SMALL	. CODE	_TEXT	Word	PUBLIC	'CODE'	
	. FARDATA	FAR_DATA	Para	Private	'FAR_DATA'	
	. FARDATA?	FAR_BSS	Para	Private	'FAR_BSS'	
	. DATA	_DATA	Word	PUBLIC	'DATA'	DGROUP
	. CONST	CONST	Word	PUBLIC	'CONST'	DGROUP
	. DATA?	_BSS	Word	PUBLIC	'BSS'	DGROUP
	. STACK	STACK	Para	STACK	'STACK'	DGROUP
. FLAT	. CODE	_TEXT	Dword	PUBLIC	'CODE'	
	. FARDATA	_DATA	Dword	PUBLIC	'DATA'	
	. FARDATA?	_BSS	Dword	PUBLIC	'FBSS'	
	. DATA	_DATA	Dword	PUBLIC	'DATA'	DGROUP
	. CONST	CONST	Dword	PUBLIC	'CONST'	DGROUP
	. DATA?	_BSS	Dword	PUBLIC	'BSS'	DGROUP
	. STACK	STACK	Dword	STACK	'STACK'	DGROUP

A. 3 选择 DOS 功能调用

并不是所有 DOS 功能调用都包括进来了，因为很难都用得到它们。DOS 最近版本的功能调用是从功能 00H 到功能 6CH。本文只列出用于简单应用的功能调用。从 DOS 版本 1. 0 开始，其中的许多功能调用已经过时许多年，而另外一些用于访问磁盘的功能调用在 Visual C++ 中仍在使用。

为了在 DOS 程序中使用 DOS 功能调用，如表 A-3 显示的那样要把功能号放在 AH 中，并将另外一些可能必要的数据放在另外的寄存器中。例 A-3 表示一个 DOS 功能调用 01H 的例子。该功能读取 DOS 键盘，并在 AL 中返回一个 ASCII 字符。一旦装载了某个功能，执行 INT 21H 指令，就完成这项任务。

例 A-3

```
MOV  AH, 01H;              ; 装载 DOS 功能号
INT  21H;                  ; 访问 DOS
; 返回, AL = ASCII 键码
```

表 A-3 DOS 功能调用

00H	**结束程序**
入口	AH = 00H CS = 程序段前缀地址
出口	已进入 DOS
01H	**读键盘**
入口	AH = 01H
出口	AL = ASCII 字符
注释	如果 AL = 00H，功能调用需再次激活，以读入一个扩展的 ASCII 字符，参见第 1 章表 1-9，该表列出了扩展的 ASCII 键盘代码，本功能调用将键入的字符在屏幕上自动回显出来
02H	**写标准输出设备**
入口	AH = 02H DL = 要显示的 ASCII 字符
注释	本功能调用通常用来在视频显示器上显示数据
03H	**从 COM1 口读入字符**
入口	AH = 03H
出口	AL = 从通信口所读到的 ASCII 字符
注释	本功能调用从串行通信口读取信息
04H	**向 COM1 口写**
入口	AH = 04H DL = 要发送到 COM1 的字符
注释	本功能调用通过串行通信口发送数据。COM 口的配置可以更改，可用 DOS MODE 命令把 COM1 再分配为其他 COM 口，以便其他 COM 口使用 03H 和 04H 功能调用
05H	**向 LPT1 口写字符**
入口	AH = 05H DL = 待打印的 ASCII 码字符
注释	向连在 LPT1 的行式打印机输出 DL 中的字符，注意，LPT1 可由 DOS MODE 命令修改
06H	**直接控制台读/写**
入口	AH = 06H DL = 0FFH 或 ASCII 码字符
出口	AL = ASCII 码
注释	如果入口时，DL = 0FFH，这是读控制台功能。如果 DL = ASCII 码，则这个功能调用是在控制台（CON）屏幕上显示 ASCII 码字符。如果是从键盘上读字符，零标志（ZF）指示是否键入字符。零条件表示未键入，非零条件表示 AL 中存放的是 ASCII 码或 00H。如果 AL 为 00H，该功能调用必须再次激活，以从键盘上读入一个扩展的 ASCII 码。注意，此键码不在屏幕上回显
07H	**无回显直接控制台输入**
入口	AH = 07H
出口	AL = ASCII 码字符
注释	此功能很像 DL = 0FFH 的 06H 功能，但它要等到有键键入时才返回

（续）

08H	读标准输入设备（不回显）
入口	AH = 08H
出口	AL = ASCII 字符
注释	除了是读标准输入设备外，实现类似于07H 号功能调用。标准输入设备可以指定为键盘或 COM 口。这个功能也对 Ctrl-Break 有响应，而06H 和07H 就没有此功能。Ctrl-Break 将导致 INT 23H 的执行。在默认情况下，此功能调用同07H 号功能调用相同
09H	**显示一个字符串**
入口	AH = 09H DS：DX = 字符串的地址
注释	字符串必须以 ASCII 码 $ （24H）结束。字符串长度任意，可包括如回车（0DH）、换行（0AH）这样的控制字符
0AH	**带缓冲区的键盘输入**
入口	AH = 0AH DS：DX = 键盘输入缓冲区地址
注释	缓冲区第一字节包括缓冲区的长度（最大 255）。第二字节表示为返回时实际键入字符的个数。从第三个字节到缓冲区的末尾为键入的字符串，末尾可能是回车符（0DH）。这个功能不断读键盘输入（并随时显示输入的数据），一直到输入指定数目的字符或键入回车符为止
0BH	**测试标准输入设备的状态**
入口	AH = 0BH
出口	AL = 输入设备的状态
注释	这个功能调用测试标准输入设备的状态，以确定是否有数据输入。如果 AL = 00，无输入，如果 AL = 0FFH，则有输入数据并且必须通过调用08H 功能读入
0CH	**清除键盘缓冲区并调用键盘功能**
入口	AH = 0CH AL = 01H、06H、07H 或 0AH
出口	见01H、06H、07H 或0AH 号功能
注释	当程序执行其他任务时，键盘缓冲区保存有键盘的输入字符。这个功能将置空或清除缓冲区，然后执行 AL 中的键盘输入功能

A.4 使用 Visual C++

本书中许多新加的例子都使用 Visual C++. net 2003，用汇编语言写程序很少。如果使用汇编语言，通常出现在 C++ 程序里要完成一项特殊的任务，或者要增加一段程序。

不是每个人都熟悉 C++ 环境，因此本书增加这一节，作为一个在用汇编语言与 Visual C++ 加载程序方面的指南。为此，最容易的应用程序类型是使用 Microsoft 公司通用语言运行库（CLR）的基于对话框的应用程序。

创建一个对话框应用程序

打开 Visual C++ Express，那么屏幕上将会如图 A-1 所示。点击 Create 右侧的 Project 来创建一个新的 C++ 项目。那么屏幕会如 A-2 所示。选择 CLR 中的 Windows Forms Application，并在名称框给它取一个惟一的有意义的名字，单击 OK。

这时，你应该看到画面如图 A-3 所示，应用程序显示了空白窗体。

总结一下创建一个基于对话框的应用程序的过程：

1）打开 Visual C++ Express。

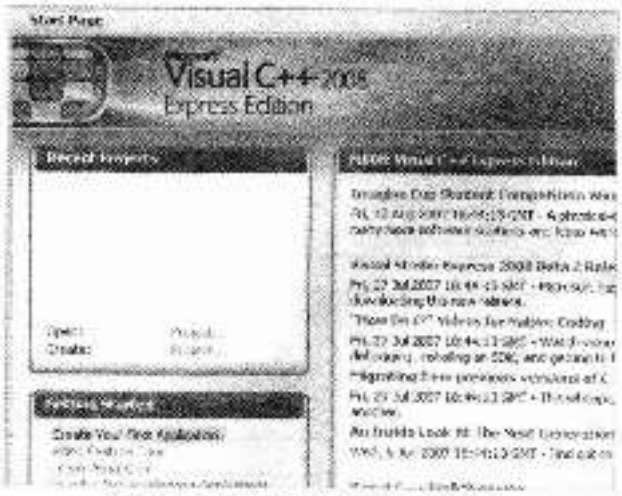


图 A-1 Visual C++ Express 开始界面

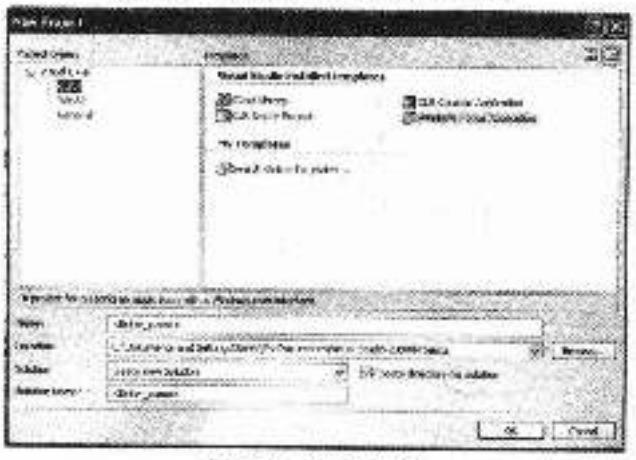


图 A-2 新项目界面

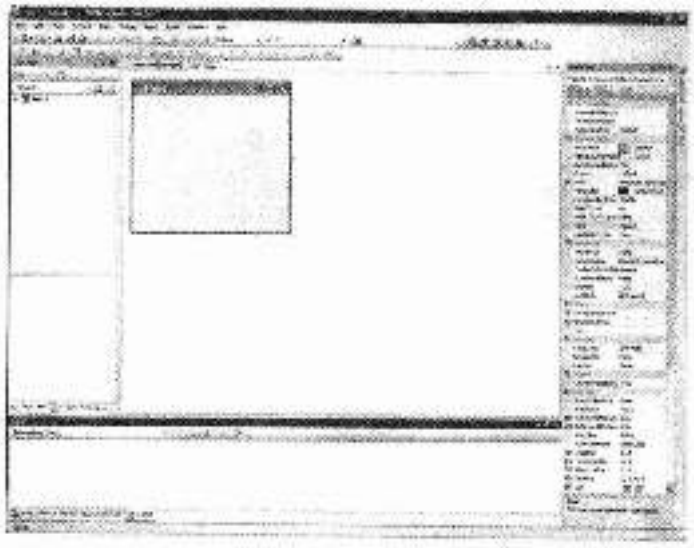


图 A-3 新项目设计界面

2）单击 Create 右侧的 Project。

3）在 CLR 中选择 Windows Forms Application，为其命名，单击 OK。

如图 A-3 所显示的屏幕是资源编辑器中基于窗体的应用程序。这时你可以准备开始往对话框窗体中放置对象。你所见到的屏幕可能会与图中所示的有些不同。如何显示取决于工具菜单，通过单击 Customize 可以改变显示。

在图 A-3 中左上角的窗口的上侧有很多符。这些制表符是用来在程序中选择不同的页，比如开始页。点击图 A-3 中 Test1 左边的那个加号（在 Class View 窗口的左上侧）来展开类。本文的许多程序中，Form1_Load 函数用来设置窗体。

图 A-4 给出了 Form1_Load 函数中的软件。为添加 Form1_Load 函数，可以双击窗体，Design 窗口会切换为 Code 视图，并且在合适的位置上显示出 Form1_Load 的程序。如果想查看一下，可以在 Class 视图上双击 Form1_Load，在屏幕上显示窗体前，CLR 架构调用 Form1_Load 函数。任何添加的用于初始化的软件都会放在图 A-4 中 Form1_Load 函数的底部。这样，你可以输入并执行任何用文本文件写的有汇编程序的 Visual C++。

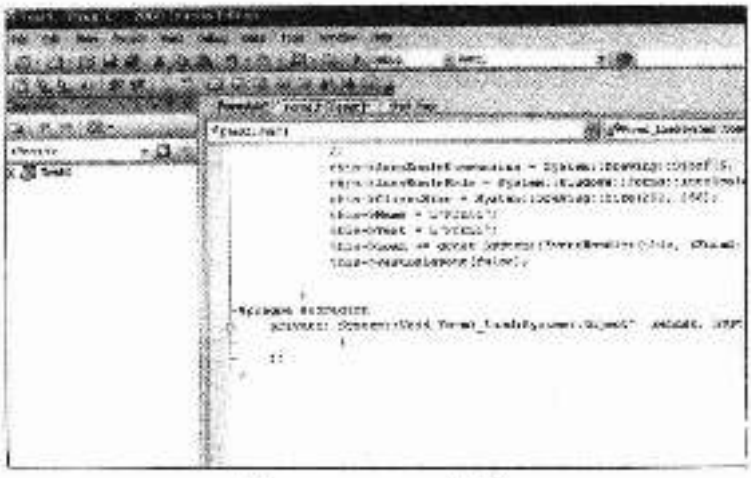


图 A-4　Form1_Load 函数

附录 B　指令系统一览

本附录按字母顺序完整地列出了 8086 ～ Pentium 4 微处理器的全部指令。在第 14 章中给出的协处理器和 MMX 的指令未重复收入此附录中。在主要指令系统一览之后，列出了 SIMD 指令。

指令表对每条指令都列出了助记符操作码和有关此指令用途的简短描述，还列出了每一条指令的二进制机器语言代码，以及构成该指令所必需的其他数据，如位移量或者立即数。在每一条指令的二进制机器语言代码的右边，都列出了各标志位及该指令对它们的影响。标志位以如下方式描述：空白符表示无影响或无变化，? 表示一个不可预测结果的变化，* 表示一个可预测结果的变化，1 表示将此标志置 1，0 表示将此标志清 0。如果一条指令未对标志位 ODITSZAPC 进行任何说明，则它不改变这些标志位。

在列指令表之前，有必要先说明一下如何设置指令的二进制机器代码中的某些位。表 B-1 给出了指令表的机器代码中 oo 编码修饰位的赋值。

表 B-1　指令表中 oo 编码修饰位的赋值

oo	功　能
00	如果 mmm = 110，那么位移量在操作码后面，否则没有使用位移量
01	操作码后面是 8 位有符号的位移量
10	操作码后面是 16 位或 32 位有符号的位移量
11	mmm 指定一个寄存器而不是一种寻址方式

表 B-2 列出了当使用寄存器字段编码 mmm 时，有效的存储器寻址方式。只要工作于 16 位指令模式，这个表适用于所有型号的微处理器。

表 B-2　16 位寄存器/存储器（mmm）字段描述

mmm	16 位寄存器
000	DS：[BX + SI]
001	DS：[BX + DI]
010	SS：[BP + SI]
011	SS：[BP + DI]
100	DS：[SI]
101	DS：[DI]
110	SS：[BP]
111	DS：[BX]

表 B-3 列出了指令中 rrr 字段所选择的寄存器。此表包括了对 8 位、16 位和 32 寄存器的选择。

表 B-4 列出了对应于 MOV、PUSH 和 POP 指令中段寄存器位（rrr）的配置。

当使用 80386 ～ Core2 微处理器时，表 B-1 到表 B-3 中的定义会有些变化。关于它们应用于 80386 ～ Core2 微处理器时的变更情况参见表 B-5 和表 B-6。

表 B-3　寄存器字段（rrr）的分配

rrr	W = 0	W = 1（16 位寄存器）	W = 1（32 位寄存器）
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX

（续）

rrr	W = 0	W = 1（16 位寄存器）	W = 1（32 位寄存器）
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

表 B-4　对段寄存器的寄存器字段（rrr）的分配

rrr	段寄存器
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS

表 B-5　80386～Core2 工作于 32 位模式时 rrr 所定义的变址寄存器

rrr	变址寄存器
000	DS：[EAX]
001	DS：[ECX]
010	DS：[EDX]
011	DS：[EBX]
100	无变址（见表 B-6）
101	SS：[EBP]
110	DS：[ESI]
111	DS：[EDI]

表 B-6　80386～Core2 微处理器用 32 位寻址方式时 oo、mmm 和 rrr 字段可能的组合

oo	mmm	rrr（基于比例变址字节）	寻址方式
00	000	—	DS：[EAX]
00	001	—	DS：[ECX]
00	010	—	DS：[EDX]
00	011	—	DS：[EBX]
00	100	000	DS：[EAX + 比例变址]
00	100	001	DS：[ECX + 比例变址]
00	100	010	DS：[EDX + 比例变址]
00	100	011	DS：[EBX + 比例变址]
00	100	100	SS：[ESP + 比例变址]
00	100	101	DS：[32 位位移量 + 比例变址]
00	100	110	DS：[ESI + 比例变址]
00	100	111	DS：[EDI + 比例变址]
00	101	—	DS：32 位位移量
00	110	—	DS：[ESI]
00	111	—	DS：[EDI]
01	000	—	DS：[EAX + 8 位的位移量]
01	001	—	DS：[ECX + 8 位的位移量]
01	010	—	DS：[EDX + 8 位的位移量]
01	011	—	DS：[EBX + 8 位的位移量]

（续）

oo	mmm	rrr（基于比例变址字节）	寻址方式
01	100	000	DS：[EAX + 比例变址 +8 位的位移量]
01	100	001	DS：[ECX + 比例变址 +8 位的位移量]
01	100	010	DS：[EDX + 比例变址 +8 位的位移量]
01	100	011	DS：[EBX + 比例变址 +8 位的位移量]
01	100	100	SS：[ESP + 比例变址 +8 位的位移量]
01	100	101	SS：[EBP + 比例变址 +8 位的位移量]
01	100	110	DS：[ESI + 比例变址 +8 位的位移量]
01	100	111	DS：[EDI + 比例变址 +8 位的位移量]
01	101	—	SS：[EBP +8 位的位移量]
01	110	—	DS：[ESI +8 位的位移量]
01	111	—	DS：[EDI +8 位的位移量]
10	000	—	DS：[EAX +32 位的位移量]
10	001	—	DS：[ECX +32 位的位移量]
10	010	—	DS：[EDX +32 位的位移量]
10	011	—	DS：[EBX +32 位的位移量]
10	100	000	DS：[EAX + 比例变址 +32 位的位移量]
10	100	001	DS：[ECX + 比例变址 +32 位的位移量]
10	100	010	DS：[EDX + 比例变址 +32 位的位移量]
10	100	011	DS：[EBX + 比例变址 +32 位的位移量]
10	100	100	SS：[ESP + 比例变址 +32 位的位移量]
10	100	101	SS：[EBP + 比例变址 +32 位的位移量]
10	100	110	DS：[ESI + 比例变址 +32 位的位移量]
10	100	111	DS：[EDI + 比例变址 +32 位的位移量]
10	101	—	SS：[EBP +32 位的位移量]
10	110	—	DS：[ESI +32 位的位移量]
10	111	—	DS：[EDI +32 位的位移量]

为了使用表 B-6 列出的带比例因子的变址寻址方式，应把 oo 和 mmm 编于操作码的第二个字节。这种带比例因子的变址字节通常在第三个字节且包含三个字段。它用最左边的两位决定比例因子（00 = ×1，01 = ×2，10 = ×4，11 = ×8），向右数三位包含了带比例因子的变址寄存器号（这可以从表 B-5 中得到）。最右面三位是列于 B-6 中的 rrr 字段。例如，指令 MOV AL，[EBX + 2 * ECX] 有一个带比例因子的变址字节 01001011。其中 01 = X_2，001 = ECX，011 = EBX。

有些指令通过添加前缀来改变默认段或超越指令模式。表 B-7 列出了段和指令模式超越前缀，当它们用于构成指令时，应附加在指令的前面。如，指令 MOV AL，ES:[BX]，由于使用了段超越前缀 ES:，因而它使用附加段。

在 8086 和 8088 微处理器中，计算有效地址还需要额外的时钟周期，这些额外的时间（列于表 B-8 中）需要加到指令系统一览表中的时间上去。在 80286 ～ Core2 中不需要增加这些时间。注意，指令系统一览表不包括 Pentium Pro ～ Core2 的时钟周期数，Intel 不公开这些时间，并确定对于通常的应用可以用 RDTSC 指令给微处理器计算时钟数，尽管这些新型微处理器的定时未公开，但它们很像 Pentium，可以用 Pentium 作为指南。

表 B-7 超越前缀

前缀字节	作 用
26H	ES：段超越前缀
2EH	CS：段超越前缀
36H	SS：段超越前缀
3EH	DS：段超越前缀

（续）

前缀字节	作　用
64H	FS：段超越前缀
65H	GS：段超越前缀
66H	对存储器操作指令模式超越
67H	对寄存器操作指令模式超越

表 B-8　8086 和 8088 微处理器中有效地址的计算

类　型	时钟周期数	例　子
基址或变址	5	MOV CL，[DI]
位移量	3	MOV AL，DATA1
基址加变址	7	MOV AL，[BP＋SI]
位移量加基址或变址	9	MOV DH，[DI＋20H]
基址加变址加位移量	11	MOV CL，[BX＋DI＋2]
段超越	ea＋2	MOV AL，ED：[DI]

B.1　指令系统一览表

AAA　　加法后对 AL 进行 ASCII 调整

00110111　　O D I T　　S Z A P C

?　　? ? * ? *

例子

微处理器	时钟周期数
8086	8
8088	8
80286	3
80386	4
80486	3
Pentium ～ Core2	3

AAD　　除法前对 AX 进行 ASCII 调整

11010101 00001010　　O D I T　　S Z A P C

?　　* * ? * ?

例子

微处理器	时钟周期数
8086	60
8088	60
80286	14
80386	19
80486	14
Pentium ～ Core2	10

AAM　　乘法后对 AX 进行 ASCII 调整

11010100 00001010　　O D I T　　S Z A P C

?　　* * ? * ?

例子

微处理器	时钟周期数
8086	83
8088	83
80286	16
80386	17
80486	15
Pentium ～ Core2	18

（续）

AAS	减法后对 AL 进行 ASCII 调整		
00111111		O D I T	S Z A P C
		?	? ? * ? *
例子		微处理器	时钟周期数
AAS		8086	8
		8088	8
		80286	3
		80386	4
		80486	3
		Pentium ~ Core2	3

ADC	带进位加法		
000100dw oorrrmmm disp		O D I T	S Z A P C
		*	* * * * *
格式	例子	微处理器	时钟周期数
ADC reg, reg	ADC AX, BX	8086	3
	ADC AL, BL	8088	3
	ADC EAX, EBX	80286	3
	ADC CX, SI	80386	3
	ADC ESI, EDI	80486	1
		Pentium ~ Core2	1 或 3
ADC mem, reg	ADC DATAY, AL	8086	16 + ea
	ADC LIST, SI	8088	24 + ea
	ADC DATA2 [DI], CL	80286	7
	ADC [EAX], BL	80386	7
	ADC [EBX +2* ECX], EDX	80486	3
		Pentium ~ Core2	1 或 3
ADC reg, mem	ADC BL, DATA1	8086	9 + ea
	ADC SI, LIST1	8088	13 + ea
	ADC CL, DATA2 [SI]	80286	7
	ADC CX, [ESI]	80386	6
	ADC ESI, [2* ECX]	80486	2
		Pentium ~ Core2	1 或 2
100000sw oo010mmm disp data			
格式	例子	微处理器	时钟周期数
ADC reg, imm	ADC CX, 3	8086	4
	ADC DI, 1AH	8088	4
	ADC DL, 34H	80286	3
	ADC EAX, 12345	80386	2
	ADC CX, 1234H	80486	1
		Pentium ~ Core2	1 或 3
ADC mem, imm	ADC DATA4, 33	8086	17 + ea
	ADC LIST, 'A'	8088	23 + ea
	ADC DATA3 [DI], 2	80286	7
	ADC BYTE PTR [EBX], 3	80386	7
	ADC WORD PTR [DI], 669H	80486	3
		Pentium ~ Core2	1 或 3

（续）

ADC acc，imm	ADC AX，3 ADC AL，1AH ADC AH，34H ADC EAX，2 ADC AL，'Z'	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1

ADD 加法

000000dw oorrrmmm disp

O D I T S Z A P C

* * * * * *

格式	例子	微处理器	时钟周期数
ADD reg，reg	ADD AX，BX ADD AL，BL ADD EAX，EBX ADD CX，SI ADD ESI，EDI	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ～ Core2	1 或 3
ADD mem，reg	ADD DATAY，AL ADD LIST，SI ADD DATA6 [DI]，CL ADD [EAX]，CL ADD [EDX +4* ECX]，EBX	8086	16 + ea
		8088	24 + ea
		80286	7
		80386	7
		80486	3
		Pentium ～ Core2	1 或 3
ADD reg，mem	ADD BL，DATA2 ADD SI，LIST3 ADD CL，DATA2 [DI] ADD CX，[EDI] ADD ESI，[ECX +200H]	8086	9 + ea
		8088	13 + ea
		80286	7
		80386	6
		80486	2
		Pentium ～ Core2	1 或 2

100000sw oo000mmm disp data

格式	例子	微处理器	时钟周期数
ADD reg，imm	ADD CX，3 ADD DI，1AH ADD DL，34H ADD EDX，1345H ADD CX，1834H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1 或 3
ADD mem，imm	ADD DATA4，33 ADD LIST，'A' ADD DATA3 [DI]，2 ADD BYTE PTR [EBX]，3 ADD WORD PTR [DI]，669H	8086	17 + ea
		8088	23 + ea
		80286	7
		80386	7
		80486	3
		Pentium ～ Core2	1 或 3
ADD acc，imm	ADD AX，3 ADD AL，1AH ADD AH，34H ADD EAX，2 ADD AL，'Z'	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1

（续）

AND 逻辑“与”			
001000dw oorrrmmm disp		O D I T 0	S Z A P C * * ? * 0
格式	例子	微处理器	时钟周期数
AND reg, reg	AND CX, BX AND DL, BL AND ECX, EBX AND BP, SI AND EDX, EDI	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ~ Core2	1 或 3
AND mem, reg	AND BIT, AL AND LIST, DI AND DATAZ [BX], CL AND [EAX], BL AND [ESI +4* ECX], EDX	8086	16 + ea
		8088	24 + ea
		80286	7
		80386	7
		80486	3
		Pentium ~ Core2	1 或 3
AND reg, mem	AND BL, DATAW AND SI, LIST AND CL, DATAQ [SI] AND CX, [EAX] AND ESI, [ECX +43H]	8086	9 + ea
		8088	13 + ea
		80286	7
		80386	6
		80486	2
		Pentium ~ Core2	1 或 2
100000sw oo100mmm disp data			
格式	例子	微处理器	时钟周期数
AND reg, imm	AND BP, 1 AND DI, 10H AND DL, 34H AND EBP, 1345H AND SP, 1834H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ~ Core2	1 或 3
AND mem, imm	AND DATA4, 33 AND LIST, ‘A’ AND DATA3 [DI], 2 AND BYTE PTR [EBX], 3 AND DWORD PTR [DI], 66H	8086	17 + ea
		8088	23 + ea
		80286	7
		80386	7
		80486	3
		Pentium ~ Core2	1 或 3
AND acc, imm	AND AX, 3 AND AL, 1AH AND AH, 34H AND EAX, 2 AND AL, ‘r’	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ~ Core2	1

（续）

ARPL 调整请求优先级（RPL）

01100011 oorrrmmm disp　　O D I T　　S Z A P C

*

格式	例子	微处理器	时钟周期数
ARPL reg，reg	ARPL AX，BX ARPL BX，SI ARPL AX，DX ARPL BX，AX ARPL SI，DI	8086	—
		8088	—
		80286	10
		80386	20
		80486	9
		Pentium ～ Core2	7
ARPL mem，reg	ARPL DATAY，AX ARPL LIST，DI ARPL DATA3［DI］，CX ARPL［EBX］，AX ARPL［EDX＋4＊ ECX］，BP	8086	—
		8088	—
		80286	11
		80386	21
		80486	9
		Pentium ～ Core2	7

BOUND 数组边界检查

01100010 oorrrmmm disp

格式	例子	微处理器	时钟周期数
BOUND reg，mem	BOUND AX，BETS BOUND BP，LISTG BOUND CX，DATAX BOUND BX，［DI］ BOUND SI，［BX＋2］	8086	—
		8088	—
		80286	13
		80386	10
		80486	7
		Pentium ～ Core2	8

BSF 从右向左位扫描

00001111 10111100 oorrrmmm disp　　O D I T　　S Z A P C

?　　? ＊ ? ? ?

格式	例子	微处理器	时钟周期数
BSF reg，reg	BSF AX，BX BSF BX，SI BSF EAX，EDX BSF EBX，EAX BSF SI，DI	8086	—
		8088	—
		80286	—
		80386	10＋3n
		80486	6～42
		Pentium ～ Core2	6～42
BSF reg，mem	BSF AX，DATAY BSF SI，LIST BSF CX，DATA3［DI］ BSF EAX，［EBX］ BSF EBP，［EDX＋4＊ ECX］	8086	—
		8088	—
		80286	—
		80386	10＋3n
		80486	7～43
		Pentium ～ Core2	6～43

（续）

BSR 从左向右位扫描

00001111 10111101 oorrrmmm disp

O D I T
?

S Z A P C
? * ? ? ?

格式	例子	微处理器	时钟周期数
BSR reg, reg	BSR AX, BX BSR BX, SI BSR EAX, EDX BSR EBX, EAX BSR SI, DI	8086	—
		8088	—
		80286	—
		80386	10 + 3n
		80486	6 ~ 103
		Pentium ~ Core2	7 ~ 71
BSR reg, mem	BSR AX, DATAY BSR SI, LIST BSR CX, DATA3 [DI] BSR EAX, [EBX] BSR EBP, [EDX + 4 * ECX]	8086	—
		8088	—
		80286	—
		80386	10 + 3n
		80486	7 ~ 104
		Pentium ~ Core2	7 ~ 72

BSWAP 字节交换

00001111 11001rrr

格式	例子	微处理器	时钟周期数
BSWAP reg32	BSWAP EAX BSWAP EBX BSWAP ECX BSWAP EDX BSWAP ESI	8086	—
		8088	—
		80286	—
		80386	—
		80486	1
		Pentium ~ Core2	1

BT 位测试

00001111 10111010 oo100mmm disp data

O D I T

S Z A P C
*

格式	例子	微处理器	时钟周期数
BT reg, imm8	BT AX, 2 BT CX, 4 BT BP, 10H BT CX, 8 BT BX, 2	8086	—
		8088	—
		80286	—
		80386	3
		80486	3
		Pentium ~ Core2	4
BT mem, imm8	BT DATA1, 2 BT LIST, 2 BT DATA2 [DI], 3 BT [EAX], 1 BT FROG, 6	8086	—
		8088	—
		80286	—
		80386	6
		80486	3
		Pentium ~ Core2	4

00001111 10100011 disp

（续）

格式	例子	微处理器	时钟周期数
BT reg，reg	BT AX，CX	8086	—
	BT CX，DX	8088	—
	BT BP，AX	80286	—
	BT SI，CX	80386	3
	BT EAX，EBX	80486	3
		Pentium ～ Core2	4 或 9
BT mem，reg	BT DATA4，AX	8086	—
	BT LIST，BX	8088	—
	BT DATA3［DI］，CX	80286	—
	BT［EBX］，DX	80386	12
	BT［DI］，DI	80486	8
		Pentium ～ Core2	4 或 9

BTC 位测试并求反

00001111 10111010 oo111mmm disp data　　O D I T　　S Z A P C

*

格式	例子	微处理器	时钟周期数
BTC reg，imm8	BTC AX，2	8086	—
	BTC CX，4	8088	—
	BTC BP，10H	80286	—
	BTC CX，8	80386	6
	BTC BX，2	80486	6
		Pentium ～ Core2	7 或 8
BTC mem，imm8	BTC DATA1，2	8086	—
	BTC LIST，2	8088	—
	BTC DATA2［DI］，3	80286	—
	BTC［EAX］，1	80386	7 或 8
	BTC FROG，6	80486	8
		Pentium ～ Core2	8

00001111 10111011 disp

格式	例子	微处理器	时钟周期数
BTC reg，reg	BTC AX，CX	8086	—
	BTC CX，DX	8088	—
	BTC BP，AX	80286	—
	BTC SI，CX	80386	6
	BTC EAX，EBX	80486	6
		Pentium ～ Core2	7 或 13
BTC mem，reg	BTC DATA4，AX	8086	—
	BTC LIST，BX	8088	—
	BTC DATA3［DI］，CX	80286	—
	BTC［EBX］，DX	80386	13
	BTC［DI］，DI	80486	13
		Pentium ～ Core2	7 或 13

BTR 位测试并清零

00001111 10111010 oo110mmm disp data　　O D I T　　S Z A P C

*

（续）

格式	例子	微处理器	时钟周期数
BTR reg, imm8	BTR AX, 2	8086	—
	BTR CX, 4	8088	—
	BTR BP, 10H	80286	—
	BTR CX, 8	80386	6
	BTR BX, 2	80486	6
		Pentium ~ Core2	7 或 8
BTR mem, imm8	BTR DATA1, 2	8086	—
	BTR LIST, 2	8088	—
	BTR DATA2 [DI], 3	80286	—
	BTR [EAX], 1	80386	8
	BTR FROG, 6	80486	8
		Pentium ~ Core2	7 或 8
00001111 10110011 disp			
格式	例子	微处理器	时钟周期数
BTR reg, reg	BTR AX, CX	8086	—
	BTR CX, DX	8088	—
	BTR BP, AX	80286	—
	BTR SI, CX	80386	6
	BTR EAX, EBX	80486	6
		Pentium ~ Core2	7 或 13
BTR mem, reg	BTR DATA4, AX	8086	—
	BTR LIST, BX	8088	—
	BTR DATA3 [DI], CX	80286	—
	BTR [EBX], DX	80386	13
	BTR [DI], DI	80486	13
	BTC [DI], DI	Pentium ~ Core2	7 或 13
BTS	位测试并置位		
00001111 10111010oo101mmm disp data		O D I T	S Z A P C *
格式	例子	微处理器	时钟周期数
BTS reg, imm8	BTS AX, 2	8086	—
	BTS CX, 4	8088	—
	BTS BP, 10H	80286	—
	BTS CX, 8	80386	6
	BTS BX, 2	80486	6
		Pentium ~ Core2	7 或 8
BTS mem, imm8	BTS DATA1, 2	8086	—
	BTS LIST, 2	8088	—
	BTS DATA2 [DI], 3	80286	—
	BTS [EAX], 1	80386	8
	BTS FROG, 6	80486	8
		Pentium ~ Core2	7 或 8
00001111 10101011 disp			

（续）

格式	例子	微处理器	时钟周期数
BTS reg，reg	BTS AX，CX BTS CX，DX BTS BP，AX BTS SI，CX BTS EAX，EBX	8086	—
		8088	—
		80286	—
		80386	6
		80486	6
		Pentium ～ Core2	7 或 13
BTS mem，reg	BTS DATA4，AX BTS LIST，BX BTS DATA3［DI］，CX BTS［EBX］，DX BTS［DI］，DI	8086	—
		8088	—
		80286	—
		80386	13
		80486	13
		Pentium ～ Core2	7 或 13
CALL	过程（子程序）调用		
11101000 disp			
格式	例子	微处理器	时钟周期数
CALL label （近）	CALL FOR_FUN CALL HOME CALL ET CALL WAITING CALL SOMEONE	8086	19
		8088	23
		80286	7
		80386	3
		80486	3
		Pentium ～ Core2	1
10011010 disp			
格式	例子	微处理器	时钟周期数
CALL label （远）	CALL FAR PTR DATES CALL WHAT CALL WHERE CALL FARCE CALL WHOM	8086	28
		8088	36
		80286	13
		80386	17
		80486	18
		Pentium ～ Core2	4
11111111 oo010mmm			
格式	例子	微处理器	时钟周期数
CALL reg （近）	CALL AX CALL BX CALL CX CALL DI CALL SI	8086	16
		8088	20
		80286	7
		80386	7
		80486	5
		Pentium ～ Core2	2
CALL mem （近）	CALL ADDRESS CALL NEAR PTR［DI］ CALL DATA1 CALL FROG CALL ME_NOW	8086	21 + ea
		8088	29 + ea
		80286	11
		80386	10
		80486	5
		Pentium ～ Core2	2
11111111 oo011mmm			

（续）

格式	例子	微处理器	时钟周期数
CALL mem （远）	CALL FAR_LIST［SI］ CALL FROM_HERE CALL TO_THERE CALL SIXX CALL OCT	8086	16
		8088	20
		80286	7
		80386	7
		80486	5
		Pentium ～ Core2	2

CBW 字节转换到字（AL⇒AX）

10011000

例子	微处理器	时钟周期数
CBW	8086	2
	8088	2
	80286	2
	80386	3
	80486	3
	Pentium ～ Core2	3

CDQ 双字转换到四字（EAX⇒EDX：EAX）

11010100 00001010

例子	微处理器	时钟周期数
CDQ	8086	—
	8088	—
	80286	—
	80386	2
	80486	2
	Pentium ～ Core2	2

CLC 将进位标志清零

11111000

O D I T	S Z A P C
	0

例子	微处理器	时钟周期数
CLC	8086	2
	8088	2
	80286	2
	80386	2
	80486	2
	Pentium ～ Core2	2

CLD 将方向标志清零

11111100

O D I T	S Z A P C
0	

例子	微处理器	时钟周期数
CLD	8086	2
	8088	2
	80286	2
	80386	2
	80486	2
	Pentium ～ Core2	2

CLI 将中断标志清零

11111010

O D I T	S Z A P C
0	

（续）

例子	微处理器	时钟周期数
CLI	8086	2
	8088	2
	80286	3
	80386	3
	80486	5
	Pentium ～ Core2	7

CLTS 清任务切换标志（CR0）

00001111 00000110

例子	微处理器	时钟周期数
CLTS	8086	—
	8088	—
	80286	2
	80386	5
	80486	7
	Pentium ～ Core2	10

CMC 进位标志求反

10011000

O D I T S Z A P C

*

例子	微处理器	时钟周期数
CMC	8086	2
	8088	2
	80286	2
	80386	2
	80486	2
	Pentium ～ Core2	2

CMOVcondition 条件传送

00001111 0100cccc oorrrmmm

格式	例子	微处理器	时钟周期数
CMOVcc reg, mem	CMOVNZ AX, FROG CMOVC EAX, [EDI] CMOVNC BX, DATA1 CMOVP EBX, WAITING CMOVNE DI, [SI]	8086	—
		8088	—
		80286	—
		80386	—
		80486	—
		Pentium ～ Core2	—

条件码	助记符	标志位	说明
0000	CMOVO	O = 1	若溢出则传送
0001	CMOVNO	O = 0	若无溢出则传送
0010	CMOVB	C = 1	若低于则传送
0011	CMOVAE	C = 0	高于或等于则传送
0100	CMOVE	Z = 1	相等/为零则传送
0101	CMOVNE	Z = 0	不相等/不为零则传送
0110	CMOVBE	C = 1 + Z = 1	低于或相等则传送
0111	CMOVA	C = 0 · Z = 0	高于则传送
1000	CMOVS	S = 1	符号为负则传送
1001	CMOVNS	S = 0	符号为正则传送
1010	CMOVP	P = 1	奇偶标志 P = 1，则传送
1011	CMOVNP	P = 0	奇偶标志 P = 0，则传送
1100	CMOVL	S · O	小于则传送
1101	CMOVGE	S = 0	大于或等于则传送
1110	CMOVLE	Z = 1 + S · O	小于或等于则传送
1111	CMOVG	Z = 0 + S = O	大于则传送

（续）

CMP 比较			
001110dw oorrrmmm disp		O D I T *	S Z A P C * * * * *
格式	例子	微处理器	时钟周期数
CMP reg，reg	CMP AX，BX CMP AL，BL CMP EAX，EBX CMP CX，SI CMP ESI，EDI	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ~ Core2	1 或 2
CMP mem，reg	CMP DATAY，AL CMP LIST，SI CMP DATA6 [DI]，CL CMP [EAX]，CL CMP [EDX +4* ECX]，EBX	8086	9 + ea
		8088	13 + ea
		80286	7
		80386	5
		80486	2
		Pentium ~ Core2	1 或 2
CMP reg，mem	CMP BL，DATA2 CMP SI，LIST3 CMP CL，DATA2 [DI] CMP CX，[EDI] CMP ESI，[ECX +200H]	8086	9 + ea
		8088	13 + ea
		80286	6
		80386	6
		80486	2
		Pentium ~ Core2	1 或 2
100000sw oo111mmm disp data			
格式	例子	微处理器	时钟周期数
CMP reg，imm	CMP CX，3 CMP DI，1AH CMP DL，34H CMP EDX，1345H CMP CX，1834H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ~ Core2	1 或 2
CMP mem，imm	CMP DATAS，3 CMP BYTE PTR [EDI]，1AH CMP DADDY，34H CMP LIST，'A' CMP TOAD，1834H	8086	10 + ea
		8088	14 + ea
		80286	6
		80386	5
		80486	2
		Pentium ~ Core2	1 或 2
0001111w data			
格式	例子	微处理器	时钟周期数
CMP acc，imm	CMP AX，3 CMP AL，1AH CMP AH，34H CMP EAX，1345H CMP AL，'Y'	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ~ Core2	1
CMPS 串比较			
1010011w		O D I T *	S Z A P C * * * * *

（续）

格式	例子	微处理器	时钟周期数
CMPSB CMPSW CMPSD	CMPSB CMPSW CMPSD CMPSB DATA1, DATA2 REPE CMPSB REPNE CMPSW	8086	32
		8088	30
		80286	8
		80386	10
		80486	8
		Pentium ~ Core2	5

CMPXCHG 比较并交换

00001111 1011000w 11rrrrrr

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

格式	例子	微处理器	时钟周期数
CMPXCHG reg, reg	CMPXCHG EAX, EBX CMPXCHG ECX, EDX	8086	—
		8088	—
		80286	—
		80386	—
		80486	6
		Pentium ~ Core2	6

0001111w data

格式	例子	微处理器	时钟周期数
CMPXCHG mem, reg	CMPXCHG DATAD, EAX CMPXCHG DATA2, EDI	8086	—
		8088	—
		80286	—
		80386	—
		80486	7
		Pentium ~ Core2	6

CMPXCHG8B 比较并交换 8 字节

00001111 11000111 oorrrmmm

O	D	I	T	S	Z	A	P	C
				*				

格式	例子	微处理器	时钟周期数
CMPXCHG8B mem64	CMPXCHG8B DATA3	8086	—
		8088	—
		80286	—
		80386	—
		80486	—
		Pentium ~ Core2	10

CPUID CPU 标识码

00001111 10100010

例子	微处理器	时钟周期数
CPUID	8086	—
	8088	—
	80286	—
	80386	—
	80486	—
	Pentium ~ Core2	14

CWD 字到双字的转换（AX⇒DX：AX）

10011000

（续）

例子	微处理器	时钟周期数
CWD	8086	5
	8088	5
	80286	2
	80386	2
	80486	3
	Pentium ～ Core2	2

CWDE 字到扩展双字的转换（AX⇒EAX）

10011000

例子	微处理器	时钟周期数
CWDE	8086	—
	8088	—
	80286	—
	80386	3
	80486	3
	Pentium ～ Core2	3

DAA 加法后对 AL 进行十进制调整

00100111

O D I T
?

S Z A P C
* * * * *

例子	微处理器	时钟周期数
DAA	8086	4
	8088	4
	80286	3
	80386	4
	80486	2
	Pentium ～ Core2	3

DAS 减法后对 AL 进行十进制调整

00101111

O D I T
?

S Z A P C
* * * * *

例子	微处理器	时钟周期数
DAS	8086	4
	8088	4
	80286	3
	80386	4
	80486	2
	Pentium ～ Core2	3

DEC 减 1

1111111w oo001mmm disp

O D I T
*

S Z A P C
* * * *

格式	例子	微处理器	时钟周期数
DEC reg8	DEC BL	8086	3
	DEC BH	8088	3
	DEC CL	80286	2
	DEC DH	80386	2
	DEC AH	80486	1
		Pentium ～ Core2	1 或 3

（续）

DEC mem	DEC DATAY DEC LIST DEC DATA6［DI］ DEC BYTE PTR［BX］ DEC WORD PTR［EBX］	8086	15 + ea
		8088	23 + ea
		80286	7
		80386	6
		80486	3
		Pentium ～ Core2	1 或 3

01001rrr			
格式	例子	微处理器	时钟周期数
DEC reg16 DEC reg32	DEC CX DEC DI DEC EDX DEC ECX DEC BP	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ～ Core2	1

DIV 除法

1111011w oo110mmm disp		O D I T ?	S Z A P C ? ? ? ? ?
格式	例子	微处理器	时钟周期数
DIV reg	DIV BL DIV BH DIV ECX DIV DH DIV CX	8086	162
		8088	162
		80286	22
		80386	38
		80486	40
		Pentium ～ Core2	17～41
DIV mem	DIV DATAY DIV LIST DIV DATA6［DI］ DIV BYTE PTR［BX］ DIV WORD PTR［EBX］	8086	168
		8088	176
		80286	25
		80386	41
		80486	40
		Pentium ～ Core2	17～41

ENTER 建一个堆栈帧

11001000 data			
格式	例子	微处理器	时钟周期数
ENTER imm，0	ENTER 4，0 ENTER 8，0 ENTER 100，0 ENTER 200，0 ENTER 1024，0	8086	—
		8088	—
		80286	11
		80386	10
		80486	14
		Pentium ～ Core2	11
ENTER imm，1	ENTER 4，1 ENTER 10，1	8086	—
		8088	—
		80286	12
		80386	15
		80486	17
		Pentium ～ Core2	15

（续）

ENTER imm，imm	ENTER 3，6 ENTER 100，3	8086	—
		8088	—
		80286	12
		80386	15
		80486	17
		Pentium ～ Core2	15 +2n
ESC	换码（已过时，见协处理器）		
HLT	暂停		
11110100			
例子		微处理器	时钟周期数
HLT		8086	2
		8088	2
		80286	2
		80386	5
		80486	4
		Pentium ～ Core2	变数
IDIV	带符号的整数除法		
1111011w oo111mmm disp		O D I T	S Z A P C
		?	? ? ? ? ?
格式	例子	微处理器	时钟周期数
IDIV reg	IDIV BL IDIV BH IDIV ECX IDIV DH IDIV CX	8086	184
		8088	184
		80286	25
		80386	43
		80486	43
		Pentium ～ Core2	22 ～46
IDIV mem	IDIV DATAY IDIV LIST IDIV DATA6 [DI] IDIV BYTE PTR [BX] IDIV WORD PTR [EBX]	8086	190
		8088	194
		80286	28
		80386	46
		80486	44
		Pentium ～ Core2	22 ～46
IMUL	有符号乘法		
1111011w oo101mmm disp		O D I T	S Z A P C
		*	? ? ? ? *
格式	例子	微处理器	时钟周期数
IMUL reg	IMUL BL IMUL CX IMUL ECX IMUL DH IMUL AL	8086	154
		8088	154
		80286	21
		80386	38
		80486	42
		Pentium ～ Core2	10 ～11
IMUL mem	IMUL DATAY IMUL LIST IMUL DATA6 [DI] IMUL BYTE PTR [BX] IMUL WORD PTR [EBX]	8086	160
		8088	164
		80286	24
		80386	41
		80486	42
		Pentium ～ Core2	10 ～11

（续）

011010s1 oorrrmmm disp data			
格式	例子	微处理器	时钟周期数
IMUL reg，imm	IMUL CX，16 IMUL DI，100 IMUL EDX，20	8086	—
		8088	—
		80286	21
		80386	38
		80486	42
		Pentium ～ Core2	10
IMUL reg，reg，imm	IMUL DX，AX，2 IMUL CX，DX，3 IMUL BX，AX，33	8086	—
		8088	—
		80286	21
		80386	38
		80486	42
		Pentium ～ Core2	10
IMUL reg，mem，imm	IMUL CX，DATAY，99	8086	—
		8088	—
		80286	24
		80386	38
		80486	42
		Pentium ～ Core2	10
00001111 10101111 oorrrmmm disp			
格式	例子	微处理器	时钟周期数
IMUL reg，reg	IMUL CX，DX IMUL DI，BX IMUL EDX，EBX	8086	—
		8088	—
		80286	—
		80386	38
		80486	42
		Pentium ～ Core2	10
IMUL reg，mem	IMUL DX，DATAY IMUL CX，LIST IMUL ECX，DATA6［DI］	8086	—
		8088	—
		80286	—
		80386	41
		80486	42
		Pentium ～ Core2	10
IN	从端口输入数据		
1110010w 端口号			
格式	例子	微处理器	时钟周期数
IN acc，pt	IN AL，12H IN AX，12H IN AL，0FFH IN AX，0A0H IN EAX，10H	8086	10
		8088	14
		80286	5
		80386	12
		80486	14
		Pentium ～ Core2	7
1110110w			

（续）

格式	例子	微处理器	时钟周期数
IN acc，DX	IN AL，DX IN AX，DX IN EAX，DX	8086	8
		8088	12
		80286	5
		80386	13
		80486	14
		Pentium ～ Core2	7

INC 加1

1111111w oo000mmm disp

O D I T
*

S Z A P C
* * * *

格式	例子	微处理器	时钟周期数
INC reg8	INC BL INC BH INC AL INC AH INC DH	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ～ Core2	1 或 3
INC mem	INC DATA3 INC LIST INC COUNT INC BYTE PTR [DI] INC WORD PTR [ECX]	8086	15 + ea
		8088	23 + ea
		80286	7
		80386	6
		80486	3
		Pentium ～ Core2	1 或 3
INC reg16 INC reg32	INC CX INC DX INC BP INC ECX INC ESP	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ～ Core2	1

INS 从端口输入串

0110110w

格式	例子	微处理器	时钟周期数
INSB INSW INSD	INSB INSW INSD INS DATA2 REP INSB	8086	—
		8088	—
		80286	5
		80386	15
		80486	17
		Pentium ～ Core2	9

INT 中断

11001101 type

格式	例子	微处理器	时钟周期数
INT type	INT 12H INT 15H INT 21H INT 2FH INT 10H	8086	51
		8088	71
		80286	23
		80386	37
		80486	30
		Pentium ～ Core2	16 ～ 82

（续）

INT 3 中断 3

11001100

例子	微处理器	时钟周期数
INT 3	8086	52
	8088	72
	80286	23
	80386	33
	80486	26
	Pentium ～ Core2	13 ～56

INTO 溢出中断指令

11001110

例子	微处理器	时钟周期数
INTO	8086	53
	8088	73
	80286	24
	80386	35
	80486	28
	Pentium ～ Core2	13 ～56

INVD 使数据缓存无效

00001111 00001000

例子	微处理器	时钟周期数
INTVD	8086	—
	8088	—
	80286	—
	80386	—
	80486	4
	Pentium ～ Core2	15

IRET/IRETD 中断返回

11001101 data

O D I T
* * * *

S Z A P C
* * * * *

格式	例子	微处理器	时钟周期数
IRET IRETD	IRET IRETD IRET 100	8086	32
		8088	44
		80286	17
		80386	22
		80486	15
		Pentium ～ Core2	8 ～27

Jcondition 条件转移

0111cccc disp

格式	例子	微处理器	时钟周期数
Jcnd label （8 位偏移）	JA ABOVE JB BELOW JG GREATER JE EQUAL *JZ ZERO*	8086	16/4
		8088	16/4
		80286	7/3
		80386	7/3
		80486	3/1
		Pentium ～ Core2	1

（续）

00001111 1000cccc disp

格式	例子	微处理器	时钟周期数
Jcnd label （16 位偏移）	JNE NOT_MORE JLE LESS_OR_SO	8086	—
		8088	—
		80286	—
		80386	7/3
		80486	3/1
		Pentium ～ Core2	1

条件码	助忆符	标志	说明
0000	JO	O = 1	溢出则转移
0001	JNO	O = 0	无溢出则转移
0010	JB/NAE	C = 1	低于则转移
0011	JAE/JNB	C = 0	高于或等于则转移
0100	JE/JZ	Z = 1	相等/为零则转移
0101	JNE/JNZ	Z = 0	不相等/非零则转移
0110	JBE/JNA	C = 1 + Z = 1	低于或等于则转移
0111	JA/JNBE	C = 0 · Z = 0	高于则转移
1000	JS	S = 1	符号为负则转移
1001	JNS	S = 0	符号为正则转移
1010	JP/JPE	P = 1	奇偶标志 P = 1 则转移
1011	JNP/JPO	P = 0	奇偶标志 P = 0 则转移
1100	JL/JNGE	S · O	小于转移
1101	JGE/JNL	S = 0	大于或等于则转移
1110	JLE/JNG	Z = 1 + S · O	小于或等于则转移
1111	JG/JNLE	Z = 0 + S = O	大于则转移

JCXZ/JECXZ 若 CX（ECX）= 0 则转移

11100011

格式	例子	微处理器	时钟周期数
JCXZ label JECXZ label	JCXZ ABOVE JCXZ BELOW JECXZ GREATER JECXZ EQUAL JCXZ NEXT	8086	18/6
		8088	18/6
		80286	8/4
		80386	9/5
		80486	8/5
		Pentium ～ Core2	6/5

JMP 无条件转移

11101011 disp

格式	例子	微处理器	时钟周期数
JMP label （短）	JMP SHORT UP JMP SHORT DOWN JMP SHORT OVER JMP SHORT CIRCUIT JMP SHORT JOKE	8086	15
		8088	15
		80286	7
		80386	7
		80486	3
		Pentium ～ Core2	1

11101001 disp

（续）

格式	例子	微处理器	时钟周期数
JMP label （近）	JMP VERS JMP FROG JMP UNDER JMP NEAR PTR OVER	8086	15
		8088	15
		80286	7
		80386	7
		80486	3
		Pentium ～ Core2	1
11101010 disp			
格式	例子	微处理器	时钟周期数
JMP label （远）	JMP NOT_MORE JMP UNDER JMP AGAIN JMP FAR PTR THERE	8086	15
		8088	15
		80286	11
		80386	12
		80486	17
		Pentium ～ Core2	3
11111111 oo100mmm			
格式	例子	微处理器	时钟周期数
JMP reg （近）	JMP AX JMP EAX JMP CX JMP DX	8086	11
		8088	11
		80286	7
		80386	7
		80486	3
		Pentium ～ Core2	2
JMP mem （近）	JMP VERS JMP FROG JMP CS:UNDER JMP DATA1 [DI+2]	8086	18 + ea
		8088	18 + ea
		80286	11
		80386	10
		80486	5
		Pentium ～ Core2	4
11111111 oo101mmm			
格式	例子	微处理器	时钟周期数
JMP mem （远）	JMP WAY_OFF JMP TABLE JMP UP JMP OUT_OF_HERE	8086	24 + ea
		8088	24 + ea
		80286	15
		80386	12
		80486	13
		Pentium ～ Core2	4
LAHF	标志寄存器的低 8 位装入 AH 中		
10011111			
例子		微处理器	时钟周期数
LAHF		8086	4
		8088	4
		80286	2
		80386	2
		80486	3
		Pentium ～ Core2	2

（续）

LAR	装入访问权限字节		
00001111 00000010 oorrrmmm disp		O D I T	S Z A P C *
格式	例子	微处理器	时钟周期数
LAR reg，reg	LAR AX，BX	8086	—
	LAR CX，DX	8088	—
	LAR ECX，EDX	80286	14
		80386	15
		80486	11
		Pentium ～ Core2	8
LAR reg，mem	LAR CX，DATA1	8086	—
	LAR AX，LIST3	8088	—
	LAR ECX，TOAD	80286	16
		80386	16
		80486	11
		Pentium ～ Core2	8
LDS	将一个远指针装入 DS 和指令指定的寄存器		
11000101 oorrrmmm			
格式	例子	微处理器	时钟周期数
LDS reg，mem	LDS DI，DATA3	8086	16 + ea
	LDS SI，LIST2	8088	24 + ea
	LDS BX，ARRAY_PTR	80286	7
	LDS CX，PNTR	80386	7
		80486	6
		Pentium ～ Core2	4
LEA	装入有效地址		
10001101 oorrrmmm disp			
格式	例子	微处理器	时钟周期数
LEA reg，mem	LEA DI，DATA3	8086	2 + ea
	LEA SI，LIST2	8088	2 + ea
	LEA BX，ARRAY_PTR	80286	3
	LEA CX，PNTR	80386	2
		80486	2
		Pentium ～ Core2	1
LEAVE	退出过程且释放为其分配的堆栈空间		
11001001			
例子		微处理器	时钟周期数
LEAVE		8086	—
		8088	—
		80286	5
		80386	4
		80486	5
		Pentium ～ Core2	3
LES	将一个远指针装入 ES 和指令指定的寄存器		
11000100 oorrrmmm			

（续）

格式	例子	微处理器	时钟周期数
LES reg，mem	LES DI，DATA3 LES SI，LIST2 LES BX，ARRAY_PTR LES CX，PNTR	8086	16 + ea
		8088	24 + ea
		80286	7
		80386	7
		80486	6
		Pentium ～ Core2	4

LFS 将一个远指针装入 FS 和指令指定的寄存器

00001111 10110100 oorrrmmm disp

格式	例子	微处理器	时钟周期数
LFS reg，mem	LFS DI，DATA3 LFS SI，LIST2 LFS BX，ARRAY_PTR LFS CX，PNTR	8086	—
		8088	—
		80286	—
		80386	7
		80486	6
		Pentium ～ Core2	4

LGDT 加载全局描述符表

00001111 00000001 oo010mmm disp

格式	例子	微处理器	时钟周期数
LGDT mem64	LGDT DESCRIP LGDT TABLED	8086	—
		8088	—
		80286	11
		80386	11
		80486	11
		Pentium ～ Core2	6

LGS 将一个远指针装入 GS 和指令指定的寄存器

00001111 10110101 oorrrmmm disp

格式	例子	微处理器	时钟周期数
LGS reg，mem	LGS DI，DATA3 LGS SI，LIST2 LGS BX，ARRAY_PTR LGS CX，PNTR	8086	—
		8088	—
		80286	—
		80386	7
		80486	6
		Pentium ～ Core2	4

LIDT 装入中断描述符表

00001111 00000001 oo011mmm disp

格式	例子	微处理器	时钟周期数
LIDT mem64	LIDT DATA3 LIDT LIST2	8086	—
		8088	—
		80286	12
		80386	11
		80486	11
		Pentium ～ Core2	6

LLDT 装入局部描述符表

00001111 00000000 oo010mmm disp

（续）

格式	例子	微处理器	时钟周期数
LLDT reg	LLDT BX LLDT DX LLDT CX	8086	—
		8088	—
		80286	17
		80386	20
		80486	11
		Pentium ~ Core2	9
LLDT mem	LLDT DATA1 LLDT LIST3 LLDT TOAD	8086	—
		8088	—
		80286	19
		80386	24
		80486	11
		Pentium ~ Core2	9
LMSW	装入机器状态字（仅用于80286）		
00001111 00000001 oo110mmm disp			
格式	例子	微处理器	时钟周期数
LMSW reg	LMSW BX LMSW DX LMSW CX	8086	—
		8088	—
		80286	3
		80386	10
		80486	2
		Pentium ~ Core2	8
LMSW mem	LMSW DATA1 LMSW LIST3 LMSW TOAD	8086	—
		8088	—
		80286	6
		80386	13
		80486	3
		Pentium ~ Core2	8
LOCK	锁定总线		
11110000			
格式	例子	微处理器	时钟周期数
LOCK：inst	LOCK:XCHG AX，BX LOCK:ADD AL，3	8086	2
		8088	3
		80286	0
		80386	0
		80486	1
		Pentium ~ Core2	1
LODS	从串中取数据		
1010110w			
格式	例子	微处理器	时钟周期数
LODSB LODSW LODSD	LODSB LODSW LODSD LODS DATA3	8086	12
		8088	15
		80286	5
		80386	5
		80486	5
		Pentium ~ Core2	2
LOOP/LOOPD	循环直到 CX = 0 或 ECX = 0		
11100010 disp			

（续）

格式	例子	微处理器	时钟周期数
LOOP label LOOPD label	LOOP NEXT LOOP BACK LOOPD LOOPS	8086	17/5
		8088	17/5
		80286	8/4
		80386	11
		80486	7/6
		Pentium ～ Core2	5/6

LOOPE /LOOPED 相等则循环

11100001 disp

格式	例子	微处理器	时钟周期数
LOOPE label LOOPED label LOOPZ label LOOPZD label	LOOPE AGAIN LOOPED UNTIL LOOPZ ZORRO LOOPZD WOW	8086	18/6
		8088	18/6
		80286	8/4
		80386	11
		80486	9/6
		Pentium ～ Core2	7/8

LOOPNE /LOOPNED 不等则循环

11100000 disp

格式	例子	微处理器	时钟周期数
LOOPNE label LOOPNED label LOOPNZ label LOOPNZD label	LOOPNE FORWARD LOOPNED UPS LOOPNZ TRY_AGAIN LOOPNZD WOO	8086	19/5
		8088	19/5
		80286	8/4
		80386	11
		80486	9/6
		Pentium ～ Core2	7/8

LSL 装段界限

00001111 00000011 oorrrmmm disp　　O D I T　　S Z A P C

*

格式	例子	微处理器	时钟周期数
LSL reg，reg	LSL AX，BX LSL CX，BX LSL EDX，EAX	8086	—
		8088	—
		80286	14
		80386	25
		80486	10
		Pentium ～ Core2	8
LSL reg，mem	LSL AX，LIMIT LSL EAX，NUM	8086	—
		8088	—
		80286	16
		80386	26
		80486	10
		Pentium ～ Core2	8

LSS 将一个远指针装入 SS 和指令指定的寄存器

00001111 10110010 oorrrmmm disp

（续）

格式	例子	微处理器	时钟周期数
LSS reg，mem	LSS DI，DATA1 LSS SP，STACK_TOP LSS CX，ARRAY	8086	—
		8088	—
		80286	—
		80386	7
		80486	6
		Pentium ~ Core2	4

LTR 加载任务寄存器

00001111 00000000 oo001mmm disp

格式	例子	微处理器	时钟周期数
LTR reg	LTR AX LTR CX LTR DX	8086	—
		8088	—
		80286	17
		80386	23
		80486	20
		Pentium ~ Core2	10
LTR mem16	LTR TASK LTR NUM	8086	—
		8088	—
		80286	19
		80386	27
		80486	20
		Pentium ~ Core2	10

MOV 传送数据

100010dw oorrrmmm disp

格式	例子	微处理器	时钟周期数
MOV reg，reg	MOV CL，CH MOV BH，CL MOV CX，DX MOV EAX，EBP MOV ESP，ESI	8086	2
		8088	2
		80286	2
		80386	2
		80486	1
		Pentium ~ Core2	1
MOV mem，reg	MOV DATA7，DL MOV NUMB，CX MOV TEMP，EBX MOV [ECX]，BL MOV [DI]，DH	8086	9 + ea
		8088	13 + ea
		80286	3
		80386	2
		80486	1
		Pentium ~ Core2	1
MOV reg，mem	MOV DL，DATA8 MOV DX，NUMB MOV EBX，TEMP + 3 MOV CH，TEMP [EDI] MOV CL，DATA2	8086	10 + ea
		8088	12 + ea
		80286	5
		80386	4
		80486	1
		Pentium ~ Core2	1

1100011w oo000mmm disp data

格式	例子	微处理器	时钟周期数
MOV mem，imm	MOV DATAF，23H MOV LIST，12H MOV BYTE PTR [DI]，2 MOV NUMB，234H MOV DWORD PTR [ECX]，1	8086	10 + ea
		8088	14 + ea
		80286	3
		80386	2
		80486	1
		Pentium ~ Core2	1

（续）

1011wrrr data 格式	例子	微处理器	时钟周期数
MOV reg，imm	MOV BX，22H MOV CX，12H MOV CL，2 MOV ECX，123456H MOV DI，100	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1
101000dw disp 格式	例子	微处理器	时钟周期数
MOV mem，acc	MOV DATAF，AL MOV LIST，AX MOV NUMB，EAX	8086	10
		8088	14
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1
MOV acc，mem	MOV AL，DATAE MOV AX，LIST MOV EAX，LUTE	8086	10
		8088	14
		80286	5
		80386	4
		80486	1
		Pentium ～ Core2	1
100011d0 oosssmmm disp 格式	例子	微处理器	时钟周期数
MOV seg，reg	MOV SS，AX MOV DS，DX MOV ES，CX MOV FS，BX MOV GS，AX	8086	2
		8088	2
		80286	2
		80386	2
		80486	1
		Pentium ～ Core2	1
MOV seg，mem	MOV SS，STACK_TOP MOV DS，DATAS MOV ES，TEMP1	8086	8 + ea
		8088	12 + ea
		80286	2
		80386	2
		80486	1
		Pentium ～ Core2	2 或 3
MOV reg，seg	MOV BX，DS MOV CX，FS MOV CX，ES	8086	2
		8088	2
		80286	2
		80386	2
		80486	1
		Pentium ～ Core2	1
MOV mem，seg	MOV DATA2，CS MOV TEMP，DS MOV NUMB1，SS MOV TEMP2，GS	8086	9 + ea
		8088	13 + ea
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1

（续）

00001111 001000d0 11rrrmmm			
格式	例子	微处理器	时钟周期数

MOV reg, cr	MOV EBX, CR0
	MOV ECX, CR2
	MOV EBX, CR3
		8086	—

		8088	—
		80286	—
		80386	6
		80486	4
		Pentium ~ Core2	4

MOV cr, reg	MOV CR0, EAX
	MOV CR1, EBX
	MOV CR3, EDX
		8086	—

		8088	—
		80286	—
		80386	10
		80486	4
		Pentium ~ Core2	12 ~ 46
00001111 001000d1 11rrrmmm			
格式	例子	微处理器	时钟周期数

MOV reg, dr	MOV EBX, DR6
	MOV ECX, DR7
	MOV EBX, DR1
		8086	—

		8088	—
		80286	—
		80386	22
		80486	10
		Pentium ~ Core2	11

MOV dr, reg	MOV DR0, EAX
	MOV DR1, EBX
	MOV DR3, EDX
		8086	—

		8088	—
		80286	—
		80386	22
		80486	11
		Pentium ~ Core2	11
00001111 001001d0 11rrrmmm			
格式	例子	微处理器	时钟周期数
MOV reg, tr	MOV EBX, TR6 MOV ECX, TR7	8086	—
		8088	—
		80286	—
		80386	12
		80486	4
		Pentium ~ Core2	11
MOV tr, reg	MOV TR6, EAX MOV TR7, EBX	8086	—
		8088	—
		80286	—
		80386	12
		80486	6
		Pentium ~ Core2	11
MOVS	串数据传送		
1010010w			

（续）

格式	例子	微处理器	时钟周期数
MOVSB MOVSW MOVSD	MOVSB MOVSW MOVSD MOVS DATA1，DATA2	8086	18
		8088	26
		80286	5
		80386	7
		80486	7
		Pentium ～ Core2	4

MOVSX 带符号扩展的传送

00001111 1011111w oorrrmmm disp

格式	例子	微处理器	时钟周期数
MOVSX reg，reg	MOVSX BX，AL MOVSX EAX，DX	8086	—
		8088	—
		80286	—
		80386	3
		80486	3
		Pentium ～ Core2	3
MOVSX reg，mem	MOVSX AX，DATA34 MOVSX EAX，NUMB	8086	—
		8088	—
		80286	—
		80386	6
		80486	3
		Pentium ～ Core2	3

MOVZX 带零扩展的传送

00001111 1011011w oorrrmmm disp

格式	例子	微处理器	时钟周期数
MOVZX reg，reg	MOVZX BX，AL MOVZX EAX，DX	8086	—
		8088	—
		80286	—
		80386	3
		80486	3
		Pentium ～ Core2	3
MOVZX reg，mem	MOVZX AX，DATA34 MOVZX EAX，NUMB	8086	—
		8088	—
		80286	—
		80386	6
		80486	3
		Pentium ～ Core2	3

MUL 无符号乘法

1111011w oo100mmm disp

O D I T
*

S Z A P C
? ? ? ? *

格式	例子	微处理器	时钟周期数
MUL reg	MUL BL MUL CX MUL EDX	8086	118
		8088	143
		80286	21
		80386	38
		80486	42
		Pentium ～ Core2	10 或 11

（续）

MUL mem	MUL DATA9 MUL WORD PTR [ESI]	8086	139
		8088	143
		80286	24
		80386	41
		80486	42
		Pentium ~ Core2	11

NEG 求补

1111011w oo011mmm disp

O D I T
*

S Z A P C
* * * * *

格式	例子	微处理器	时钟周期数
NEG reg	NEG BL NEG CX NEG EDI	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ~ Core2	1 或 3
NEG mem	NEG DATA9 NEG WORD PTR [ESI]	8086	16 + ea
		8088	24 + ea
		80286	7
		80386	6
		80486	3
		Pentium ~ Core2	1 或 3

NOP 空操作

10010000

例子	微处理器	时钟周期数
NOP	8086	3
	8088	3
	80286	3
	80386	3
	80486	3
	Pentium ~ Core2	1

NOT 求反

1111011w oo010mmm disp

格式	例子	微处理器	时钟周期数
NOT reg	NOT BL NOT CX NOT EDI	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ~ Core2	1 或 3
NOT mem	NOT DATA9 NOT WORD PTR [ESI]	8086	16 + ea
		8088	24 + ea
		80286	7
		80386	6
		80486	3
		Pentium ~ Core2	1 或 3

（续）

OR	逻辑“或”		
000010dw oorrrmmm disp		O D I T 0	S Z A P C * * ? * 0
格式	例子	微处理器	时钟周期数
OR reg, reg	OR AX, BX OR AL, BL OR EAX, EBX OR CX, SI OR ESI, EDI	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ~ Core2	1 或 2
OR mem, reg	OR DATAY, AL OR LIST, SI OR DATA2 [DI], CL OR [EAX], BL OR [EBX +2* ECX], EDX	8086	16 + ea
		8088	24 + ea
		80286	7
		80386	7
		80486	3
		Pentium ~ Core2	1 或 3
OR reg, mem	OR BL, DATA1 OR SI, LIST1 OR CL, DATA2 [SI] OR CX, [ESI] OR ESI, [2* ECX]	8086	9 + ea
		8088	13 + ea
		80286	7
		80386	6
		80486	2
		Pentium ~ Core2	1 或 3
100000sw oo001mmm disp data			
格式	例子	微处理器	时钟周期数
OR reg, imm	OR CX, 3 OR DI, 1AH OR DL, 34H OR EDX, 1345H OR CX, 1834H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ~ Core2	1 或 3
OR *mem, imm*	OR DATAS, 3 OR BYTE PTR [EDI], 1AH OR DADDY, 34H OR LIST, ‘A’ OR TOAD, 1834H	8086	17 + ea
		8088	25 + ea
		80286	7
		80386	7
		80486	3
		Pentium ~ Core2	1 或 3
0000110w data			
格式	例子	微处理器	时钟周期数
OR acc, imm	OR AX, 3 OR AL, 1AH OR AH, 34H OR EAX, 1345H OR AL, ‘Y’	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ~ Core2	1
OUT	向端口输出数据		
1110011w port#			

（续）

格式	例子	微处理器	时钟周期数
OUT pt，acc	OUT 12H，AL OUT 12H，AX OUT 0FFH，AL OUT 0A0H，AX OUT 10H，EAX	8086	10
		8088	14
		80286	3
		80386	10
		80486	10
		Pentium ～ Core2	12 ～26
1110111w			
格式	例子	微处理器	时钟周期数
OUT DX，acc	OUT DX，AL OUT DX，AX OUT DX，EAX	8086	8
		8088	12
		80286	3
		80386	11
		80486	10
		Pentium ～ Core2	12 ～26
OUTS	向端口输出一串数据		
0110111w			
格式	例子	微处理器	时钟周期数
OUTSB OUTSW OUTSD	OUTSB OUTSW OUTSD OUTS DATA2 REP OUTSB	8086	—
		8088	—
		80286	5
		80386	14
		80486	10
		Pentium ～ Core2	13 ～27
POP	从堆栈弹出数据		
01011rrr			
格式	例子	微处理器	时钟周期数
POP reg	POP CX POP AX POP EDI	8086	8
		8088	12
		80286	5
		80386	4
		80486	1
		Pentium ～ Core2	1
10001111 oo000mmm disp			
格式	例子	微处理器	时钟周期数
POP mem	POP DATA1 POP LISTS POP NUMBS	8086	17 + ea
		8088	25 + ea
		80286	5
		80386	5
		80486	4
		Pentium ～ Core2	3
00sss111			
格式	例子	微处理器	时钟周期数
POP seg	POP DS POP ES POP SS	8086	8
		8088	12
		80286	5
		80386	7
		80486	3
		Pentium ～ Core2	3

（续）

00001111 10sss001

格式	例子	微处理器	时钟周期数
POP seg	POP FS POP GS	8086	—
		8088	—
		80286	—
		80386	7
		80486	3
		Pentium ～ Core2	3

POPA /POPAD 从堆栈弹出所有寄存器

01100001

例子	微处理器	时钟周期数
POPA POPAD	8086	—
	8088	—
	80286	19
	80386	24
	80486	9
	Pentium ～ Core2	5

POPF/POPFD 从堆栈弹出标志寄存器

10010000 O D I T S Z A P C

* * ** * * * * * *

例子	微处理器	时钟周期数
POPF POPFD	8086	8
	8088	12
	80286	5
	80386	5
	80486	6
	Pentium ～ Core2	4 或 6

PUSH 将数据压入堆栈

01010rrr

格式	例子	微处理器	时钟周期数
PUSH reg	PUSH CX PUSH AX PUSH EDI	8086	11
		8088	15
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1

11111111 oo110mmm disp

格式	例子	微处理器	时钟周期数
PUSH mem	PUSH DATA1 PUSH LISTS PUSH NUMBS	8086	16 + ea
		8088	24 + ea
		80286	5
		80386	5
		80486	4
		Pentium ～ Core2	1 或 2

00ss110

（续）

格式	例子	微处理器	时钟周期数
PUSH seg	PUSH ES	8086	10
	PUSH CS	8088	14
	PUSH DS	80286	3
		80386	2
		80486	3
		Pentium ～ Core2	1
00001111 10sss000			
格式	例子	微处理器	时钟周期数
PUSH seg	PUSH FS	8086	—
	PUSH GS	8088	—
		80286	—
		80386	2
		80486	3
		Pentium ～ Core24	1
011010s0 data			
格式	例子	微处理器	时钟周期数
PUSH imm	PUSH 2000H	8086	—
	PUSH 53220	8088	—
	PUSHW 10H	80286	3
	PUSH ‘,’	80386	2
	PUSHD 100000H	80486	1
		Pentium ～ Core2	1
PUSHA /PUSHAD 将所有寄存器压入堆栈			
01100000			
例子		微处理器	时钟周期数
PUSHA		8086	—
PUSHAD		8088	—
		80286	17
		80386	18
		80486	11
		Pentium ～ Core2	5
PUSHF /PUSHFD 将标志寄存器压入堆栈			
10011100			
例子		微处理器	时钟周期数
PUSHF		8086	10
PUSHFD		8088	14
		80286	3
		80386	4
		80486	3
		Pentium ～ Core2	3 或 4
RCL /RCR /ROL /ROR 循环移位			
1101000w ooTTTmmm disp		O D I T	S Z A P C
		*	*
TTT = 000 = ROL，TTT = 001 = ROR，TTT = 010 = RCL 和 TTT = 011 = RCR			

（续）

格式	例子	微处理器	时钟周期数
ROL reg, 1 ROR reg, 1	ROL CL, 1 ROL DX, 1 ROR CH, 1 ROL SI, 1	8086	2
		8088	2
		80286	2
		80386	3
		80486	3
		Pentium ～ Core2	1 或 3
RCL reg, 1 RCR reg, 1	RCL CL, 1 RCL SI, 1 RCR AH, 1 RCR EBX, 1	8086	2
		8088	2
		80286	2
		80386	9
		80486	3
		Pentium ～ Core2	1 或 3
ROL mem, 1 ROR mem, 1	ROL DATAY, 1 ROL LIST, 1 ROR DATA2 [DI], 1 ROR BYTE PTR [EAX], 1	8086	15 + ea
		8088	23 + ea
		80286	7
		80386	7
		80486	4
		Pentium ～ Core2	1 或 3
RCL mem, 1 RCR mem, 1	RCL DATA1, 1 RCL LIST, 1 RCR DATA2 [SI], 1 RCR WORD PTR [ESI], 1	8086	15 + ea
		8088	23 + ea
		80286	7
		80386	10
		80486	4
		Pentium ～ Core2	1 或 3
1101001w ooTTTmmm disp			
格式	例子	微处理器	时钟周期数
ROL reg, CL ROR reg, CL	ROL CH, CL ROL DX, CL ROR AL, CL ROR ESI, CL	8086	8 + 4n
		8088	8 + 4n
		80286	5 + n
		80386	3
		80486	3
		Pentium ～ Core2	4
RCL reg, CL RCR reg, CL	RCL CH, CL RCL SI, CL RCR AH, CL RCR EBX, CL	8086	8 + 4n
		8088	8 + 4n
		80286	5 + n
		80386	9
		80486	3
		Pentium ～ Core2	7 ～ 27
ROL mem, CL ROR mem, CL	ROL DATAY, CL ROL LIST, CL ROR DATA2 [DI], CL ROR BYTE PTR [EAX], CL	8086	20 + 4n
		8088	28 + 4n
		80286	8 + n
		80386	7
		80486	4
		Pentium ～ Core2	4
RCL mem, CL RCR mem, CL	RCL DATA1, CL RCL LIST, CL RCR DATA2 [SI], CL RCR WORD PTR [ESI], CL	8086	20 + 4n
		8088	28 + 4n
		80286	8 + n
		80386	10
		80486	9
		Pentium ～ Core2	9 ～ 26

（续）

1100000w ooTTTmmm disp data			
格式	例子	微处理器	时钟周期数
ROL reg, imm ROR reg, imm	ROL CH, 4 ROL DX, 5 ROR AL, 2 ROL ESI, 14	8086	—
		8088	—
		80286	5 + n
		80386	3
		80486	2
		Pentium ～ Core2	1 或 3
RCL reg, imm RCR reg, imm	RCL CL, 2 RCL SI, 12 RCR AH, 5 RCR EBX, 18	8086	—
		8088	—
		80286	5 + n
		80386	9
		80486	8
		Pentium ～ Core2	8 ～27
ROL mem, imm ROR mem, imm	ROL DATAY, 4 ROL LIST, 3 ROR DATA2 [DI], 7 ROR BYTE PTR [EAX], 11	8086	—
		8088	—
		80286	8 + n
		80386	7
		80486	4
		Pentium ～ Core2	1 或 3
RCL mem, imm RCR mem, imm	RCL DATA1, 5 RCL LIST, 3 RCR DATA2 [SI], 9 RCR WORD PTR [ESI], 8	8086	—
		8088	—
		80286	8 + n
		80386	10
		80486	9
		Pentium ～ Core2	8 ～27
RDMSR	读专用模式寄存器		
00001111 00110010			
例子		微处理器	时钟周期数
RDMSR		8086	—
		8088	—
		80286	—
		80386	—
		80486	—
		Pentium ～ Core2	20 ～24
REP	重复前缀		
11110011 1010010w			
格式	例子	微处理器	时钟周期数
REP MOVS	REP MOVSB REP MOVSW REP MOVSD REP MOVS DATA1, DATA2	8086	9 + 17n
		8088	9 + 25n
		80286	5 + 4n
		80386	8 + 4n
		80486	12 + 3n
		Pentium ～ Core2	13 + n
11110011 1010101w			

（续）

格式	例子	微处理器	时钟周期数
REP STOS	REP STOSB REP STOSW REP STOSD REP STOS ARRAY	8086	9 + 10n
		8088	9 + 14n
		80286	4 + 3n
		80386	5 + 5n
		80486	7 + 4n
		Pentium ~ Core2	9 + n
11110011 0110110w			
格式	例子	微处理器	时钟周期数
REP INS	REP INSB REP INSW REP INSD REP INS ARRAY	8086	—
		8088	—
		80286	5 + 4n
		80386	12 + 5n
		80486	17 + 5n
		Pentium ~ Core2	25 + 3n
11110011 0110111w			
格式	例子	微处理器	时钟周期数
REP OUTS	REP OUTSB REP OUTSW REP OUTSD REP OUTS ARRAY	8086	—
		8088	—
		80286	5 + 4n
		80386	12 + 5n
		80486	17 + 5n
		Pentium ~ Core2	25 + 4n
REPE /REPNE	条件重复前缀		
11110011 1010011w			
格式	例子	微处理器	时钟周期数
REPE CMPS	REPE CMPSB REPE CMPSW REPE CMPSD REPE CMPS DATA1，DATA2	8086	9 + 22n
		8088	9 + 30n
		80286	5 + 9n
		80386	5 + 9n
		80486	7 + 7n
		Pentium ~ Core2	9 + 4n
11110011 1010111w			
格式	例子	微处理器	时钟周期数
REPE SCAS	REPE SCASB REPE SCASW REPE SCASD REPE SCAS ARRAY	8086	9 + 15n
		8088	9 + 19n
		80286	5 + 8n
		80386	5 + 8n
		80486	7 + 5n
		Pentium ~ Core2	9 + 4n
11110010 1010011w			
格式	例子	微处理器	时钟周期数
REPNE CMPS	REPNE CMPSB REPNE CMPSW REPNE CMPSD REPNE CMPS ARRAY，LIST	8086	9 + 22n
		8088	9 + 30n
		80286	5 + 9n
		80386	5 + 9n
		80486	7 + 7n
		Pentium ~ Core2	8 + 4n
11110010 1010111w			

（续）

格式	例子	微处理器	时钟周期数
REPNE SCAS	REPNE SCASB REPNE SCASW REPNE SCASD REPNE SCAS ARRAY	8086	9 + 15n
		8088	9 + 19n
		80286	5 + 8n
		80386	5 + 8n
		80486	7 + 5n
		Pentium ～ Core2	9 + 4n

RET 过程返回

11000011

例子	微处理器	时钟周期数
RET （近）	8086	16
	8088	20
	80286	11
	80386	10
	80486	5
	Pentium ～ Core2	2

11000010 data

格式	例子	微处理器	时钟周期数
RET imm （近）	RET 4 RET 100H	8086	20
		8088	24
		80286	11
		80386	10
		80486	5
		Pentium ～ Core2	3

11001011

例子	微处理器	时钟周期数
RET （远）	8086	26
	8088	34
	80286	15
	80386	18
	80486	13
	Pentium ～ Core2	4 ～ 23

11001010 data

格式	例子	微处理器	时钟周期数
RET imm （远）	RET 4 RET 100H	8086	25
		8088	33
		80286	11
		80386	10
		80486	5
		Pentium ～ Core2	4 ～ 23

RSM 恢复系统管理方式

00001111 10101010	O D I T	S Z A P C
	* * * *	* * * * *

（续）

例子	微处理器	时钟周期数
RSM	8086	—
	8088	—
	80286	—
	80386	—
	80486	—
	Pentium ～ Core2	83

SAHF 将 AH 的内容存入标志寄存器的低 8 位

10011110

O D I T S Z A P C

* * * * *

例子	微处理器	时钟周期数
SAHF	8086	4
	8088	4
	80286	2
	80386	3
	80486	2
	Pentium ～ Core2	2

SAL /SAR /SHL /SHR 移位

1101000w ooTTTmmm disp

O D I T S Z A P C

* * ? * *

TTT = 100 = SHL/SAL，TTT = 101 = SHR 和 TTT = 111 = SAR

格式	例子	微处理器	时钟周期数
SAL reg, 1 SHL reg, 1 SHR reg, 1 SAR reg, 1	SAL CL, 1 SHL DX, 1 SAR CH, 1 SHR SI, 1	8086	2
		8088	2
		80286	2
		80386	3
		80486	3
		Pentium ～ Core2	1 或 3
SAL mem, 1 SHL mem, 1 SHR mem, 1 SAR mem, 1	SAL DATA1, 1 SHL BYTE PTR [DI], 1 SAR NUMB, 1 SHR WORD PTR [EDI], 1	8086	15 + ea
		8088	23 + ea
		80286	7
		80386	7
		80486	4
		Pentium ～ Core2	1 或 3

1101001w ooTTTmmm disp

格式	例子	微处理器	时钟周期数
SAL reg, CL SHL reg, CL SAR reg, CL SHR reg, CL	SAL CH, CL SHL DX, CL SAR AL, CL SHR ESI, CL	8086	8 + 4*n*
		8088	8 + 4n
		80286	5 + n
		80386	3
		80486	3
		Pentium ～ Core2	4
SAL mem, CL SHL *mem*, CL SAR mem, CL SHR mem, CL	SAL DATAU, CL SHL BYTE PTR [ESI], CL SAR NUMB, CL SHR TEMP, CL	8086	20 + 4n
		8088	28 + 4n
		80286	8 + n
		80386	7
		80486	4
		Pentium ～ Core2	4

（续）

1100000w ooTTTmmm disp data			
格式	例子	微处理器	时钟周期数
SAL reg, imm SHL reg, imm SAR reg, imm SHR reg, imm	SAL CH, 4 SHL DX, 10 SAR AL, 2 SHR ESI, 23	8086	—
		8088	—
		80286	5 + n
		80386	3
		80486	2
		Pentium ~ Core2	1 或 3
SAL mem, imm SHL mem, imm SAR mem, imm SHR mem, imm	SAL DATAU, 3 SHL BYTE PTR [ESI], 15 SAR NUMB, 3 SHR TEMP, 5	8086	—
		8088	—
		80286	8 + n
		80386	7
		80486	4
		Pentium ~ Core2	1 或 3

SBB 带借位的减法

000110dw oorrrmmm disp		O D I T *	S Z A P C * * * * *
格式	例子	微处理器	时钟周期数
SBB reg, reg	SBB CL, DL SBB AX, DX SBB CH, CL SBB EAX, EBX SBB ESI, EDI	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ~ Core2	1 或 2
SBB mem, reg	SBB DATAJ, CL SBB BYTES, CX SBB NUMBS, ECX SBB [EAX], CX	8086	16 + ea
		8088	24 + ea
		80286	7
		80386	6
		80486	3
		Pentium ~ Core2	1 或 3
SBB reg, mem	SBB CL, DATAL SBB CX, BYTES SBB ECX, NUMBS SBB DX, [EBX + EDI]	8086	9 + ea
		8088	13 + ea
		80286	7
		80386	7
		80486	2
		Pentium ~ Core2	1 或 2

100000sw oo011mmm disp data			
格式	例子	微处理器	时钟周期数
SBB reg, imm	SBB CX, 3 SBB DI, 1AH SBB DL, 34H SBB EDX, 1345H SBB CX, 1834H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ~ Core2	1 或 3

（续）

SBB mem，imm	SBB DATAS，3 SBB BYTE PTR［EDI］，1AH SBB DADDY，34H SBB LIST，'A' SBB TOAD，1834H	8086	17 + ea
		8088	25 + ea
		80286	7
		80386	7
		80486	3
		Pentium ～ Core2	1 或 3

0001110w data

格式	例子	微处理器	时钟周期数
SBB acc，imm	SBB AX，3 SBB AL，1AH SBB AH，34H SBB EAX，1345H SBB AL，'Y'	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1

SCAS 串扫描

1010111w

O D I T
*

S Z A P C
* * * * *

格式	例子	微处理器	时钟周期数
SCASB SCASW SCASD	SCASB SCASW SCASD SCAS DATAF REP SCASB	8086	15
		8088	19
		80286	7
		80386	7
		80486	6
		Pentium ～ Core2	4

SETcondition 条件置位

00001111 1001cccc oo000mmm

格式	例子	微处理器	时钟周期数
SETcnd reg8	SETA BL SETB CH SETG DL SETE BH SETZ AL	8086	—
		8088	—
		80286	—
		80386	4
		80486	3
		Pentium ～ Core2	1 或 2
SETcnd mem8	SETE DATAK SETAE LESS_OR_SO	8086	—
		8088	—
		80286	—
		80386	5
		80486	3
		Pentium ～ Core2	1 或 2

条件码	助记符	标志	说明
0000	SETO	O = 1	溢出则置位
0001	SETNO	O = 0	无溢出则置位
0010	SETB/SETAE	C = 1	低于则置位
0011	SETAE/SETNB	C = 0	高于或等于则置位
0100	SETE/SETZ	Z = 1	等于或零则置位
0101	SETNE/SETNZ	Z = 0	不等或非零则置位
0110	SETBE/SETNA	C = 1 + Z = 1	低于或等于则置位
0111	SETA/SETNBE	C = 0 · Z = 0	高于则置位
1000	SETS	S = 1	符号为负则置位

（续）

条件码	助记符	标志	说明
1001	SETNS	S=0	符号为正则置位
1010	SETP/SETPE	P=1	奇偶标志P为1则置位
1011	SETNP/SETPO	P=0	奇偶标志P为0则置位
1100	SETL/SETNGE	S·O	小于则置位
1101	SETGE/SETNL	S=0	大于或等于则置位
1110	SETLE/SETNG	Z=1+S·O	小于或等于则置位
1111	SETG/SETNLE	Z=0+S=O	大于则置位

SGDT/SIDT/SLDT 存储描述符表寄存器内容

00001111 00000001 oo000mmm disp

格式	例子	微处理器	时钟周期数
SGDT mem	SGDT MEMORY SGDT GLOBAL	8086	—
		8088	—
		80286	11
		80386	9
		80486	10
		Pentium ~ Core2	4

00001111 00000001 oo001mmm disp

格式	例子	微处理器	时钟周期数
SIDT mem	SIDT DATAS SIDT INTERRUPT	8086	—
		8088	—
		80286	12
		80386	9
		80486	10
		Pentium ~ Core2	4

00001111 00000000 oo000mmm disp

格式	例子	微处理器	时钟周期数
SLDT reg	SLDT CX SLDT DX	8086	—
		8088	—
		80286	2
		80386	2
		80486	2
		Pentium ~ Core2	2
SLDT mem	SLDT NUMBS SLDT LOCALS	8086	—
		8088	—
		80286	3
		80386	2
		80486	3
		Pentium ~ Core2	2

SHLD/SHRD 双精度移位

00001111 10100100 oorrrmmm disp data	O D I T	S Z A P C
	?	* * ? * *

（续）

格式	例子	微处理器	时钟周期数
SHLD reg，reg，imm	SHLD AX，CX，10 SHLD DX，BX，8 SHLD CX，DX，2	8086	—
		8088	—
		80286	—
		80386	3
		80486	2
		Pentium ～ Core2	4
SHLD mem，reg，imm	SHLD DATAQ，CX，8	8086	—
		8088	—
		80286	—
		80386	7
		80486	3
		Pentium ～ Core2	4
00001111 10101100 oorrrmmm disp data			
格式	例子	微处理器	时钟周期数
SHRD reg，reg，imm	SHRD CX，DX，2	8086	—
		8088	—
		80286	—
		80386	3
		80486	2
		Pentium ～ Core2	4
SHRD mem，reg，imm	SHRD DATAZ，DX，4	8086	—
		8088	—
		80286	—
		80386	7
		80486	2
		Pentium ～ Core2	4
00001111 10100101 oorrrmmm disp			
格式	例子	微处理器	时钟周期数
SHLD reg，reg，CL	SHLD BX，DX，CL	8086	—
		8088	—
		80286	—
		80386	3
		80486	4
		Pentium ～ Core2	4 或 5
SHLD mem，reg，CL	SHLD DATAZ，DX，CL	8086	—
		8088	—
		80286	—
		80386	7
		80486	3
		Pentium ～ Core2	4 或 5
00001111 10101101 oorrrmmm disp			
格式	例子	微处理器	时钟周期数
SHRD reg，reg，CL	SHRD AX，DX，CL	8086	—
		8088	—
		80286	—
		80386	3
		80486	3
		Pentium ～ Core2	4 或 5

（续）

SHRD mem, reg, CL	SHRD DATAZ, DX, CL	8086	—
		8088	—
		80286	—
		80386	7
		80486	3
		Pentium ~ Core2	4 或 5

SMSW	存机器状态字（只用于80286）		
00001111 00000001 oo100mmm disp			
格式	例子	微处理器	时钟周期数
SMSW reg	SMSW AX SMSW DX SMSW BP	8086	—
		8088	—
		80286	2
		80386	10
		80486	2
		Pentium ~ Core2	4
SMSW mem	SMSW DATAQ	8086	—
		8088	—
		80286	3
		80386	3
		80486	3
		Pentium ~ Core2	4

STC	设置进位标志		
11111001		O D I T 1	S Z A P C
例子		微处理器	时钟周期数
STC		8086	2
		8088	2
		80286	2
		80386	2
		80486	2
		Pentium ~ Core2	2

STD	设置方向标志		
11111101		O D I T 1	S Z A P C
例子		微处理器	时钟周期数
STD		8086	2
		8088	2
		80286	2
		80386	2
		80486	2
		Pentium ~ Core2	2

STI	置中断标志		
11111011		O D I T 1	S Z A P C

（续）

例子	微处理器	时钟周期数
STI	8086	2
	8088	2
	80286	2
	80386	3
	80486	5
	Pentium ～ Core2	7

STOS 存串数据

1010101w

格式	例子	微处理器	时钟周期数
STOSB STOSW STOSD	STOSB STOSW STOSD STOS DATA_LIST REP STOSB	8086	11
		8088	15
		80286	3
		80386	40
		80486	5
		Pentium ～ Core2	3

STR 存任务寄存器

00001111 00000000 oo001mmm disp

格式	例子	微处理器	时钟周期数
STR reg	STR AX STR DX STR BP	8086	—
		8088	—
		80286	2
		80386	2
		80486	2
		Pentium ～ Core2	2
STR mem	STR DATA3	8086	—
		8088	—
		80286	2
		80386	2
		80486	2
		Pentium ～ Core2	2

SUB 减法

000101dw oorrrmmm disp

O D I T
*

S Z A P C
* * * * *

格式	例子	微处理器	时钟周期数

SUB reg, reg	SUB CL, DL
	SUB AX, DX
	SUB CH, CL
	SUB EAX, EBX
	SUB ESI, EDI
		8086	3

		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ～ Core2	1 或 2

SUB mem, reg	SUB DATAJ, CL
	SUB BYTES, CX
	SUB NUMBS, ECX
	SUB [EAX], CX
		8086	16 + ea

		8088	24 + ea
		80286	7
		80386	6
		80486	3
		Pentium ～ Core2	1 或 3

（续）

SUB reg，mem	SUB CL，DATAL	8086	9 + ea
	SUB CX，BYTES	8088	13 + ea
	SUB ECX，NUMBS	80286	7
	SUB DX，[EBX + EDI]	80386	7
		80486	2
		Pentium ～ Core2	1 或 2
100000sw oo101mmm disp data			
格式	例子	微处理器	时钟周期数
SUB reg，imm	SUB CX，3	8086	4
	SUB DI，1AH	8088	4
	SUB DL，34H	80286	3
	SUB EDX，1345H	80386	2
	SUB CX，1834H	80486	1
		Pentium ～ Core2	1 或 3
SUB mem，imm	SUB DATAS，3	8086	17 + ea
	SUB BYTE PTR [EDI]，1AH	8088	25 + ea
	SUB DADDY，34H	80286	7
	SUB LIST，'A'	80386	7
	SUB TOAD，1834H	80486	3
		Pentium ～ Core2	1 或 3
0010110w data			
格式	例子	微处理器	时钟周期数
SUB acc，imm	SUB AL，3	8086	4
	SUB AX，1AH	8088	4
	SUB EAX，34H	80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1
TEST	测试操作数（逻辑比较）		
1000001w oorrrmmm disp		O D I T 0	S Z A P C * * ? * 0
格式	例子	微处理器	时钟周期数
TEST reg，reg	TEST CL，DL	8086	5
	TEST BX，DX	8088	5
	TEST DH，CL	80286	2
	TEST EBP，EBX	80386	2
	TEST EAX，EDI	80486	1
		Pentium ～ Core2	1 或 2
TEST mem，reg reg，mem	TEST DATAJ，CL	8086	9 + ea
	TEST BYTES，CX	8088	13 + ea
	TEST NUMBS，ECX	80286	6
	TEST [EAX]，CX	80386	5
	TEST CL，POPS	80486	2
		Pentium ～ Core2	1 或 2
1111011sw oo000mmm disp data			

（续）

格式	例子	微处理器	时钟周期数
TEST reg，imm	TEST BX，3 TEST DI，1AH TEST DH，44H TEST EDX，1AB345H TEST SI，1834H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1 或 2
TEST mem，imm	TEST DATAS，3 TEST BYTE PTR［EDI］，1AH TEST DADDY，34H TEST LIST，'A' TEST TOAD，1834H	8086	11 + ea
		8088	11 + ea
		80286	6
		80386	5
		80486	2
		Pentium ～ Core2	1 或 2
1010100w data			
格式	例子	微处理器	时钟周期数
TEST acc，imm	TEST AL，3 TEST AX，1AH TEST EAX，34H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1

VERR /VERW 校验读/写

00001111 00000000 oo100mmm disp		O D I T	S Z A P C *
格式	例子	微处理器	时钟周期数
VERR reg	VERR CX VERR DX VERR DI	8086	—
		8088	—
		80286	14
		80386	10
		80486	11
		Pentium ～ Core2	7
VERR mem	VERR DATAJ VERR TESTB	8086	—
		8088	—
		80286	16
		80386	11
		80486	11
		Pentium ～ Core2	7
00001111 00000000 oo101mmm disp			
格式	例子	微处理器	时钟周期数
VERW reg	VERW CX VERW DX VERW DI	8086	—
		8088	—
		80286	14
		80386	15
		80486	11
		Pentium ～ Core2	7
VERW mem	VERW DATAJ VERW TESTB	8086	—
		8088	—
		80286	16
		80386	16
		80486	11
		Pentium ～ Core2	7

（续）

WAIT 等待协处理器

10011011

例子	微处理器	时钟周期数
WAIT	8086	4
FWAIT	8088	4
	80286	3
	80386	6
	80486	6
	Pentium ～ Core2	1

WBINVD 写回缓存并使数据缓存无效

00001111 00001001

例子	微处理器	时钟周期数
WBINVD	8086	—
	8088	—
	80286	—
	80386	—
	80486	5
	Pentium ～ Core2	2000 +

WRMSR 写专用方式寄存器

00001111 00110000

例子	微处理器	时钟周期数
WRMSR	8086	—
	8088	—
	80286	—
	80386	—
	80486	—
	Pentium ～ Core2	30 ～45

XADD 交换并相加

00001111 1100000w 11rrrrrr

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

格式	例子	微处理器	时钟周期数
XADD reg，reg	XADD EBX，ECX	8086	—
	XADD EDX，EAX	8088	—
	XADD EDI，EBP	80286	—
		80386	—
		80486	3
		Pentium ～ Core2	3 或 4

00001111 1100000w oorrrmmm disp

格式	例子	微处理器	时钟周期数
XADD mem，reg	XADD DATA5，ECX	8086	—
	XADD [EBX]，EAX	8088	—
	XADD [ECX +4]，EBP	80286	—
		80386	—
		80486	4
		Pentium ～ Core2	3 或 4

（续）

XCHG 交换

1000011w oorrrmmm

格式	例子	微处理器	时钟周期数
XCHG reg，reg	XCHG CL，DL XCHG BX，DX XCHG DH，CL XCHG EBP，EBX XCHG EAX，EDI	8086	4
		8088	4
		80286	3
		80386	3
		80486	3
		Pentium ～ Core2	3
XCHG mem，reg reg，mem	XCHG DATAJ，CL XCHG BYTES，CX XCHG NUMBS，ECX XCHG［EAX］，CX XCHG CL，POPS	8086	17 + ea
		8088	25 + ea
		80286	5
		80386	5
		80486	5
		Pentium ～ Core2	3

10010reg

格式	例子	微处理器	时钟周期数
XCHG acc，reg reg，acc	XCHG BX，.AX XCHG AX，DI XCHG DH，AL XCHG EDX，EAX XCHG SI，AX	8086	3
		8088	3
		80286	3
		80386	3
		80486	3
		Pentium ～ Core2	2

XLAT 换码

11010111

例子	微处理器	时钟周期数
XLAT	8086	11
	8088	11
	80286	5
	80386	3
	80486	4
	Pentium ～ Core2	4

XOR 异或

000110dw oorrrmmm disp

O D I T S Z A P C

0 * * ? * 0

格式	例子	微处理器	时钟周期数
XOR reg，reg	XOR CL，DL XOR AX，DX XOR CH，CL XOR EAX，EBX XOR ESI，EDI	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium ～ Core2	1 或 2
XOR mem，reg	XOR DATAJ，CL XOR BYTES，CX XOR NUMBS，ECX XOR［EAX］，CX	8086	16 + ea
		8088	24 + ea
		80286	7
		80386	6
		80486	3
		Pentium ～ Core2	1 或 3

（续）

XOR reg, mem	XOR CL, DATAL XOR CX, BYTES XOR ECX, NUMBS XOR DX, [EBX + EDI]	8086	9 + ea
		8088	13 + ea
		80286	7
		80386	7
		80486	2
		Pentium ～ Core2	1 或 2
100000sw oo110mmm disp data			
格式	例子	微处理器	时钟周期数
XOR reg, imm	XOR CX, 3 XOR DI, 1AH XOR DL, 34H XOR EDX, 1345H XOR CX, 1834H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1 或 3
XOR mem, imm	XOR DATAS, 3 XOR BYTE PTR [EDI], 1AH XOR DADDY, 34H XOR LIST, 'A' XOR TOAD, 1834H	8086	17 + ea
		8088	25 + ea
		80286	7
		80386	7
		80486	3
		Pentium ～ Core2	1 或 3
0010101w data			
格式	例子	微处理器	时钟周期数
XOR acc, imm	XOR AL, 3 XOR AX, 1AH XOR EAX, 34H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium ～ Core2	1

B.2 SIMD 指令系统表

SIMD（单指令流、多数据流）指令和一些新的特性使微处理器用于完成多媒体应用和操作。XXM 寄存器编号为 XMM_0 到 XMM_7，每个寄存器的宽度是 128 位，存储在 XXM 寄存器中用于 SIMD 指令的数据格式如图 B-1 所示。

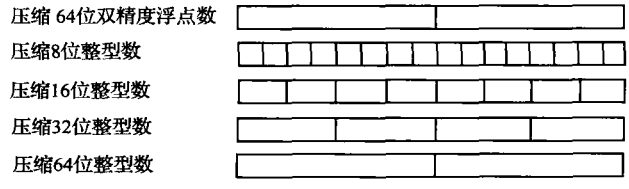


图 B-1 Pentium Ⅲ和 Pentium 4 微处理器中 128 位宽数据格式

数据在存储器中必须存储在连续的 16 个字节存储单元中，当该数据被一条指令访问时，要用带 OWORD PTR 超越前缀来寻址。OWORD PTR 超越前缀用于寻址 8 字宽的数据或者 16 字节宽的数据。SIMD 指令可以操作压缩的（Packed）双精度数和标量（Scalar）双精度数。两种数据格式的操作如图 B-2 所示，图中表示了压缩数的乘法和标量数的乘法。注意，标量数乘法不用源操作数寄存器中靠左边的数来乘，只复制目的寄存器中靠左边的双精度数。标量指令表示与浮点协处理器指令兼容。

本附录这一节详细介绍众多的 SIMD 指令，举例说明它们的用法。

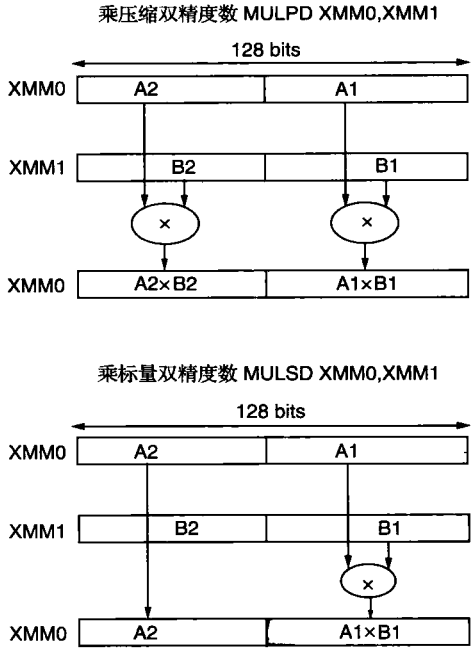


图 B-2 压缩双精度浮点数和标量双精度浮点数

B.3 数据传送指令

MOVAPD 传送对齐压缩双精度数，数据必须按 16 字节边界排列对齐
例子 MOVAPD XMM0, OWORD DATA3 ; 把 DATA3 复制到 XMM0 MOVAPD OWORD PTR DATA4, XMM2; 把 XMM2 复制到 DATA4
MOVUPD 传送非对齐压缩双精度数
例子 MOVUPD XMM0, OWORD DATA3 ; 把 DATA3 复制到 XMM0 MOVUPD OWORD PTR DATA4, XMM2; 把 XMM2 复制到 DATA4
MOVSD 传送标量压缩双精度数到低四字
例子 MOVSD XMM0, DWORD DATA3 ; 把 DATA3 复制到 XMM0 MOVSD DWORD PTR DATA4, XMM2 ; 把 XMM2 复制到 DATA4
MOVHPD 传送压缩双精度数到高四字
例子 MOVHPD XMM0, DWORD DATA3 ; 把 DATA3 复制到 XMM0 MOVHPD DWORD PTR DATA4, XMM2 ; 把 XMM2 复制到 DATA4
MOVLPD 传送压缩双精度到低四字中
例子 MOVLPD XMM0, DWORD DATA3 ; 把 DATA3 复制到 XMM0 MOVLPD DWORD PTR DATA4, XMM2 ; 把 XMM2 复制到 DATA4
MOVMSKPD 传送压缩双精度掩码

（续）

例子
MOVMSKPD EAX，XMM1；把2个符号位复制到通用寄存器中

MOVAPS 传送4个对齐压缩单精度数，数据按16字节边界排列对齐

例子
```
MOVAPS XMM0，OWORD DATA3      ；把DATA3复制到XMM0
MOVAPS OWORD PTR DATA4，XMM2；把XMM2复制到DATA4
```

MOVUPS 传送4个非对齐压缩单精度数

例子
```
MOVUPS XMM0，OWORD DATA3      ；把DATA3复制到XMM0
MOVUPS OWORD PTR DATA4，XMM2；把XMM2复制到DATA4
```

MOVLPS 传送2个压缩单精度数到低端四字

例子
```
MOVLPS XMM0，OWORD DATA3      ；把DATA3复制到XMM0
MOVLPS OWORD PTR DATA4，XMM2；把XMM2复制到DATA4
```

MOVHPS 传送压缩单精度数到高端四字

例子
```
MOVHPS XMM0，OWORD DATA3      ；把DATA3复制到XMM0
MOVHPS OWORD PTR DATA4，XMM2；把XMM2复制到DATA4
```

MOVAPD 传送对齐的压缩双精度数，数据必须已按16字节边界对齐

例子
```
MOVAPD XMM0，OWORD DATA3      ；把DATA3复制到XMM0
MOVAPD OWORD PTR DATA4，XMM2；把XMM2复制到DATA4
```

MOVLHPS 传送2个压缩单精度数从低端四字到高端四字

例子
```
MOVLHPS XMM0，XMM1            ；把XMM1低端四字复制到XMM0高端四字
MOVLHPS XMM3，XMM2            ；把XMM2低端四字复制到XMM3高端四字
```

MOVHLPS 传送2个压缩单精度数，从高端四字到低端四字

例子
```
MOVHLPS XMM0，XMM2            ；把XMM2高端四字复制到XMM0低端四字
MOVHLPS XMM4，XMM5            ；把XMM5高端四字复制到XMM4低端四字
```

MOVMSKPS 传送4个压缩单精度数的4个符号位到通用寄存器

例子
```
MOVMSKPS EBX，XMM0            ；把XMM0的符号位复制到EBX
MOVMSKPS EDX，XMM2            ；把XMM2的符号位复制到EDX
```

B.4 算术指令

ADDPD 加压缩双精度数据

例子
```
ADDPD  XMM0，OWORD DATA3      ；把DATA3加到XMM0
ADDPD  XMM2，XMM3             ；把XMM3加到XMM2
```

ADDSD 加标量双精度数据

（续）

例子
```
ADDSD   XMM0, OWORD DATA3      ; 把 DATA3 加到 XMM0
ADDSD   XMM4, XMM2             ; 把 XMM2 加到 XMM4
```

ADDPS 加 2 个压缩单精度数字

例子
```
ADDPS   XMM0, QWORD DATA3      ; 把 DATA3 加到 XMM0
ADDPS   XMM3, XMM2             ; 把 XMM2 加到 XMM3
```

ADDLS 加标量单精度数

例子
```
ADDLS   XMM0, DWORD DATA3      ; 把 DATA3 加到 XMM0
ADDLS   XMM7, XMM2             ; 把 XMM2 加到 XMM7
```

SUBPD 减压缩双精度数

例子
```
SUBPD   XMM0, OWORD DATA3      ; 从 XMM0 中减去 DATA3
SUBPD   XMM2, XMM3             ; 从 XMM2 中减去 XMM3
```

SUBSD 减标量双精度数

例子
```
SUBSD   XMM0, OWORD DATA3      ; 从 XMM0 中减去 DATA3
SUBSD   XMM4, XMM2             ; 从 XMM4 中减去 XMM2
```

SUBPS 减 2 个压缩单精度数字

例子
```
SUBPS   XMM0, QWORD DATA3      ; 从 XMM0 中减去 DATA3
SUBPS   XMM3, XMM2             ; 从 XMM3 中减去 XMM2
```

SUBLS 减标量单精度数

例子
```
SUBLS   XMM0, DWORD DATA3      ; 从 XMM0 中减去 DATA3
SUBLS   XMM7, XMM2             ; 从 XMM7 中减去 XMM2
```

MULPD 乘压缩双精度数

例子
```
MULPD   XMM0, OWORD DATA3      ; XMM0 乘以 DATA3
MULPD   XMM3, XMM2             ; XMM3 乘以 XMM2
```

MULSD 乘标量双精度数

例子
```
MULSD   XMM0, OWORD DATA3      ; XMM0 乘以 DATA3
MULSD   XMM3, XMM6             ; XMM3 乘以 XMM 6
```

MULPS 乘 2 个压缩单精度数字

例子
```
MULPS   XMM0, QWORD DATA3      ; XMM0 乘以 DATA3
MULPS   XMM0, XMM2             ; XMM0 乘以 XMM 2
```

MULSS 乘一个单精度数字

例子
```
MULSS   XMM0, DWORD DATA3      ; XMM0 乘以 DATA3
MULSS   XMM1, XMM2             ; XMM1 乘以 XMM2
```

（续）

DIVPD 除压缩双精度数
例子 DIVPD XMM0, OWORD DATA3 ; XMM0 除以 DATA3 DIVPD XMM3, XMM2 ; XMM3 除以 XMM2
DIVSD 除标量双精度数
例子 DIVSD XMM0, OWORD DATA3 ; XMM0 除以 DATA3 DIVSD XMM3, XMM6 ; XMM3 除以 XMM6
DIVPS 除 2 个压缩单精度数字
例子 DIVPS XMM0, QWORD DATA3 ; XMM0 除以 DATA3 DIVPS XMM0, XMM2 ; XMM0 除以 XMM2
DIVSS 除一个单精度数字
例子 DIVSS XMM0, DWORD DATA3 ; XMM0 除以 DATA3 DIVSS XMM1, XMM2 ; XMM1 除以 XMM2
SQRTPD 求压缩双精度数的平方根
例子 SQRTPD XMM0, OWORD DATA3 ; 求 DATA3 的平方根，结果放在 XMM0 中 SQRTPD XMM3, XMM2 ; 求 XMM2 的平方根，结果放在 XMM3 中
SQRTSD 求标量双精度的平方根
例子 SQRTSD XMM0, OWORD DATA3 ; 求 DATA3 的平方根，结果放在 XMM0 中 SQRTSD XMM3, XMM6 ; 求 XMM6 的平方根，结果放在 XMM3 中
SQRTPS 求 2 个压缩单精度数字的平方根
例子 SQRTPS XMM0, QWORD DATA3 ; 求 DATA3 的平方根，结果放在 XMM0 中 SQRTPS XMM0, XMM2 ; 求 XMM2 的平方根，结果放在 XMM0 中
SQRTSS 求单精度数的平方根
例子 SQRTSS XMM0, DWORD DATA3 ; 求 DATA3 的平方根，结果放在 XMM0 中 SQRTSS XMM1, XMM2 ; 求 XMM2 的平方根，结果放在 XMM1 中
RCPPS 求一个压缩单精度数的倒数
例子 RCPPS XMM0, OWORD DATA3 ; 求 DATA3 的倒数，结果放在 XMM0 中 RCPPS XMM3, XMM2 ; 求 XMM2 的倒数，结果放在 XMM3 中
RCPSS 求一个单精度数字的倒数
例子 RCPSS XMM0, OWORD DATA3 ; 求 DATA3 的倒数，结果放在 XMM0 中 RCPSS XMM3, XMM6 ; 求 XMM6 的倒数，结果放在 XMM3 中
RSQRTPS 求各压缩单精度数平方根的倒数

（续）

例子
```
RSQRTPS  XMM0, OWORD DATA3    ; 求 DATA3 的平方根的倒数
RSQRTPS  XMM3, XMM2           ; 求 XMM2 的平方根的倒数
```

RSQRTSS 求标量单精度数平方根的倒数

例子
```
RSQRTSS  XMM0, OWORD DATA3    ; 求 DATA3 的平方根的倒数
RSQRTSS  XMM3, XMM6           ; 求 XMM6 的平方根的倒数
```

MAXPD 比较并返回最大的压缩双精度浮点数值

例子
```
MAXPD  XMM0, OWORD DATA3      ; 比较 DATA3 中的各数，将最大的数放入 XMM0 中
MAXPD  XMM3, XMM2             ; 比较 XMM2 中的各数，将最大的数放入 XMM3 中
```

MAXSD 比较标量双精度数并返回大者

例子
```
MAXSD  XMM0, OWORD DATA3      ; 比较 DATA3 中的数，将最大的数放入 XMM0 中
MAXSD  XMM3, XMM6             ; 比较 XMM6 中的数，将最大的数放入 XMM3 中
```

MAXPS 比较并返回最大的压缩单精度数

例子
```
MAXPS  XMM0, QWORD DATA3      ; 比较 DATA3 中的各数值，将最大的数放入 XMM0 中
MAXPS  XMM0, XMM2             ; 比较 XMM2 中的各数值，将最大的数放入 XMM0 中
```

MAXSS 比较标量单精度数值并返回最大者

例子
```
MAXSS  XMM0, DWORD DATA3      ; 比较 DATA3 中的各数，将最大的数放入 XMM0 中
MAXSS  XMM1, XMM2             ; 比较 XMM2 中的各数，将最大的数放入 XMM1 中
```

MINPD 比较并返回最小的压缩双精度浮点数

例子
```
MINPD  XMM0, OWORD DATA3      ; 比较 DATA3 中的数，将较小的数放入 XMM0 中
MINPD  XMM3, XMM2             ; 比较 XMM2 中的数，将较小的数放入 XMM3 中
```

MINSD 比较标量双精度数并返回最小者

例子
```
MINSD  XMM0, OWORD DATA3      ; 比较 DATA3 中的数，将较小的数放入 XMM0 中
MINSD  XMM3, XMM6             ; 比较 XMM6 中的数，将较小的数放入 XMM3 中
```

MINPS 比较并返回最小的压缩单精度数值

例子
```
MINPS  XMM0, QWORD DATA3      ; 比较 DATA3 中的各数，将最小的数放入 XMM0 中
MINPS  XMM0, XMM2             ; 比较 XMM2 中的各数，将最小的数放入 XMM0 中
```

MINSS 比较标量单精度数值并返回最小者

例子
```
MINSS  XMM0, DWORD DATA3      ; 比较 DATA3 中的各数，将最小的数放入 XMM0 中
MINSS  XMM1, XMM2             ; 比较 XMM2 中的各数，将最小的数放入 XMM1 中
```

B.5 逻辑运算指令

ANDPD 压缩双精度数‘与’

（续）

例子
ANDPD XMM0, OWORD DATA3 ; DATA3 和 XMM0 “与”
ANDPD XMM2, XMM3 ; XMM3 和 XMM2 “与”

ANDNPD 压缩双精度数‘与非’

例子
ANDNPD XMM0, OWORD DATA3 ; DATA3 和 XMM0 “与非”
ANDNPD XMM4, XMM2 ; XMM2 和 XMM4 “与非”

ANDPS 双压缩单精度数‘与’

例子
ANDPS XMM0, QWORD DATA3 ; DATA3 和 XMM0 “与”
ANDPS XMM3, XMM2 ; XMM2 和 XMM3 “与”

ANDNPS 双压缩单精度数‘与非’

例子
ANDNPS XMM0, DWORD DATA3 ; DATA3 和 XMM0 “与非”
ANDNPS XMM7, XMM2 ; XMM2 和 XMM7 “与非”

ORPD 压缩双精度数‘或’

例子
ORPD XMM0, OWORD DATA3 ; DATA3 和 XMM0 “或”
ORPD XMM2, XMM3 ; XMM3 和 XMM2 “或”

ORPS 双压缩单精度数‘或’

例子
ORPS XMM0, OWORD DATA3 ; DATA3 和 XMM0 “或”
ORPS XMM3, XMM2 ; XMM2 和 XMM3 “或”

XORPD 压缩双精度数据‘异或’

例子
XORPD XMM0, OWORD DATA3 ; DATA3 和 XMM0 “异或”
XORPD XMM2, XMM3 ; XMM3 和 XMM2 “异或”

XORPS 压缩双精度数‘异或’

例子
XORPS XMM0, OWORD DATA3 ; DATA3 和 XMM0 “异或”
XORPS XMM2, XMM3 ; XMM3 和 XMM2 “异或”

B.6 比较指令

CMPPD 压缩双精度数比较

例子
CMPPD XMM0, OWORD DATA3 ; DATA3 与 XMM0 比较
CMPPD XMM2, XMM3 ; XMM3 与 XMM2 比较

CMPSD 标量双精度数比较

例子
CMPSD XMM0, QWORD DATA3 ; DATA3 与 XMM0 比较
CMPSD XMM3, XMM2 ; XMM2 与 XMM3 比较

CMPISD 标量双精度比较设置 EFLAGS

例子
CMPISD XMM0, OWORD DATA3 ; DATA3 与 XMM0 比较
CMPISD XMM2, XMM3 ; XMM3 与 XMM2 比较

UCOMISD 比较标量无序双精度数并改变 EFLAGS

（续）

例子
UCOMISD XMM0, QWORD DATA3 ; DATA3 和 XMM0 比较
UCOMISD XMM3, XMM2 ; XMM2 和 XMM3 比较
CMPPS 比较压缩单精度数
例子
CMPPS XMM0, OWORD DATA3 ; DATA3 和 XMM0 比较
CMPPS XMM2, XMM3 ; XMM3 和 XMM2 比较
CMPSS 双压缩单精度数比较
例子
CMPSS XMM0, QWORD DATA3 ; DATA3 和 XMM0 比较
CMPSS XMM3, XMM2 ; XMM2 和 XMM3 比较
COMISS 比较标量单精度数并改变 EFLAGS
例子
COMISS XMM0, OWORD DATA3 ; DATA3 与 XMM0 比较
COMISS XMM2, XMM3 ; XMM3 与 XMM2 比较
UCOMISS 比较无序单精度数值并改变 EFLAGS
例子
UCOMISS XMM0, QWORD DATA3 ; DATA3 与 XMM0 比较
UCOMISS XMM3, XMM2 ; XMM2 与 XMM3 比较

B.7 数据转换指令

SHUFPD 重组压缩双精度数
例子
SHUFPD XMM0, OWORD DATA3 ; 重组 DATA3 和 XMM0
SHUFPD XMM2, XMM2 ; XMM2 中的高四字和低四字交换
UNPCKHPD 高双精度数解压缩
例子
UNPCKHPD XMM0, OWORD DATA3 ; DATA3 解压缩，放入 XMM0 中
UNPCKHPD XXM3, XMM2 ; XMM2 解压缩，放入 XMM3 中
UNPCKLPD 低双精度数解压缩
例子
UNPCKLPD XMM0, OWORD DATA3 ; DATA3 解压缩，放入 XMM0 中
UNPCKLPD XMM3, XMM2 ; XMM2 解压缩，放入 XMM3 中
SHUFPS 重组压缩单精度数
例子
SHUFPS XMM0, QWORD DATA3 ; DATA3 和 XMM0 重组
SHUFPS XMM2, XMM2 ; XMM2 中的高四字和低四字交换
UNPCKHPS 低双精度数解压缩
例子
UNPCKHPS XMM0, QWORD DATA3 ; DATA3 解压缩，放入 XMM0 中
UNPCKHPS XMM3, XMM2 ; XMM2 解压缩，放入 XMM3 中
UNPCKLPSD 低双精度数解压缩
例子
UNPCKLPSD XMM0, QWORD DATA3 ; DATA3 解压缩，放入 XMM0 中
UNPCKLPSD XMM3, XMM2 ; XMM2 解压缩，放入 XMM3 中

附录 C　标志位的变化

该附录只列出了引起标志位变化的那些指令，没有列出不影响任何标志位的指令。

指　令	O	D	I	T	S	Z	A	P	C
AAA	?				?	?	*	?	*
AAD	?				*	*	?	*	?
AAM	?				*	*	?	*	?
AAS	?				?	?	*	?	*
ADC	*				*	*	*	*	*
ADD	*				*	*	*	*	*
AND	0				*	*	?	*	0
ARPL						*			
BSF						*			
BSR						*			
BT									*
BTC									*
BTR									*
BTS									*
CLC									0
CLD		0							
CLI			0						
CMC									*
CMP	*				*	*	*	*	*
CMPS	*				*	*	*	*	*
CMPXCHG	*				*	*	*	*	*
CMPXCHG8B						*			
DAA	?				*	*	*	*	*
DAS	?				*	*	*	*	*
DEC	*				*	*	*	*	
DIV	?				?	?	?	?	?
IDIV	?				?	?	?	?	?
IMUL	*				?	?	?	?	*
INC	*				*	*	*	*	
IRET	*	*	*	*	*	*	*	*	*
LAR						*			
LSL						*			
MUL	*				?	?	?	?	*
NEG	*				*	*	*	*	*
OR	0				*	*	?	*	0
POPF	*	*	*	*	*	*	*	*	*
RCL/RCR	*								*
REPE/REPNE						*			
ROL/ROR	*								*
SAHF					*	*	*	*	*
SAL/SAR	*				*	*	?	*	*
SHL/SHR	*				*	*	?	*	*

（续）

指　令	O	D	I	T	S	Z	A	P	C
SBB	*				*	*	*	*	*
SCAS	*				*	*	*	*	*
SHLD/SHRD	?				*	*	?	*	*
STC									1
STD		1							
STI			1						
SUB	*				*	*	*	*	*
TEST	0				*	*	?	*	0
VERR/VERW						*			
XADD	*				*	*	*	*	*
XOR	0				*	*	?	*	0

附录 D　偶数号习题的答案

第 1 章

2. Herman Hollerith
4. Konrad Zuse
6. ENIAC
8. Augusta Ada Byron
10. 那种将它的程序存储在存储系统中的机器。
12. 2 亿
14. 16MB
16. 1993
18. 2000
20. 每秒执行百万条指令（Millions of instructions per second）。
22. 1 或 0
24. 1024KB
26. 1024
28. 临时程序区（TPA 区）和系统区
30. 640KB
32. 1MB
34. 80386、80486、Pentium、Pertium Pro、P Ⅱ、P Ⅲ、P4 和 Core2
36. 基本 I/O 系统
38. 8088 和 8086 采用 XT 系统，从 80286 开始使用 AT 系统。
40. 8 位和 16 位
42. 高级图形端口（AGP）用于视频显示卡。
44. 串行 ATA 接口用于硬盘驱动存储器。
46. 64KB
48. 见图 1-6
50. 地址、数据和控制总线
52. $\overline{\text{MRDC}}$
54. 存储器读操作
56. (a) 8 位有符号数 (b) 16 位有符号数 (c) 32 位有符号数 (d) 32 位浮点数 (e) 64 位浮点数
58. (a) 156.625 (b) 18.375 (c) 4087.109375 (d) 83.578125 (e) 58.90625
60. (a) 10111_2, 27_8, 17_{16} (b) 1101011_2, 153_8, 6B (c) 10011010110_2, 2326_8, $4D6_{16}$ (d) 1011100_2, 134_8, $5C_{16}$ (e) 10101101_2, 255_8, AD
62. (a) 0010 0011 (b) 1010 1101 0100
(c) 0011 0100. 1010 1101 (d) 1011 1101 0011 0010
(e) 0010 0011 0100. 0011
64. (a) 0111 0111 (b) 1010 0101 (c) 1000 1000
(d) 0111 1111
66. 字节是 8 位二进制数，字是 16 位二进制数，双字是 32 位二进制数。
68. Enter（回车键）是 0DH，用于将显示器的光标或打印机头返回到屏幕或页的最左边。
70. LINE1 DB 'What time is it?'
72. (a) 0000 0011 1110 1000 (b) 1111 1111 1000 1000
(c) 0000 0011 0010 0000 (d) 1111 0011 0111 0100
74. char Fred1 = -34
76. "从小到大"的数其最低有效字节总是存储在最低地址存储单元中，而最高有效字节存储在高地址存储单元；"从大到小"的数则依照最低地址单元存储最高位有效数字的方式存储数据
78. (a) 压缩 = 00000001 00000010;
非压缩 = 00000001 00000000 00000010
(b) 压缩 = 01000100; 非压缩 = 00000100 00000100
(c) 压缩 = 00000011 00000001;
非压缩 = 00000011 00000000 00000001
(d) 压缩 = 00010000 00000000;
非压缩 = 00000001 00000000 00000000 00000000
80. (a) 89 (b) 9 (c) 32 (d) 1
82. (a) +3.5 (b) -1.0 (c) +12.5

第 2 章

2. 16 位
4. EBX
6. 用于保存程序下一条指令的偏移地址。
8. 不会。+1 加 -1 等于 0，是有效数。
10. 中断标志（I）
12. 在实模式下，段寄存器定位一 64KB 存储器段的起始地址。
14. (a) 12000H (b) 21000H (c) 24A00H (d) 25000H (e) 3F12DH
16. DI
18. SS 加 SP 或 SS 加 ESP。
20. (a) 12000H (b) 21002H (c) 26200H (d) A1000H (e) 2CA00H
22. 所有 16MB。
24. 段寄存器中含有一个选择子，该选择子用于从描述表中选择一个描述符。段寄存器中还设有请求优先级以及选定局部或全局描述表。
26. A00000H～A01000H
28. 00280000H～00290FFFH
30. 3

32. 64KB

34.

0000 0011	1101 0000
1001 0010	0000 0000
0000 0000	0000 0000
0010 1111	1111 1111

36. 通过存储在全局描述表中的描述符。

38. 对程序不可见的寄存器是指段寄存器、GDTR 寄存器、LDTR 寄存器和 IDTR 寄存器的 cache 区。

40. 4KB

42. 1024

44. 页目录 000H 和页表项 200H

46. TLB 高速缓存最近来自分页机构的存储器访问。

第 3 章

2. AH，AL，BH，BL，CH，CL，DH 和 DL

4. EAX，EBX，ECX，EDX，ESP，EBP，ESI 和 EDI

6. CS、DS、ES、SS、FS 和 GS

8. 寄存器的长度不匹配。

10. (a) MOVAL，12H (b) MOV AX，123AH (c) MOV CL，0CDH (d) MOV RAX，1000H (e) MOV EBX，1200A2H

12. 选择一种汇编语言编程模型，它像 .COM 程序那样包含一个单一的段。

14. 标识符是表示存储地址的。

16. 标识符可以以一个字母或者其他特殊的字符开始，但不能以数字开始。

18. . TINY 模型创建了一个 .COM 程序。

20. 偏移量表示的是偏移值。在 MOV DS：[2000H]，AL 中，偏移量 2000H 加上 DS 内容乘以 10H 后的值来形成内存地址。

22. (a) 3234H (b) 2300H (c) 2400H

24. MOV BYTE PTR [2000H]，6

26. MOV DWORD PTR DATA1，5

28. MOV BX，DATA 将位于 DATA 存储单元的内容复制到 BX，而 MOV BX，OFFSET DATA 指令则将 DATA 的偏移地址复制到 BX。

30. 这条指令没有错误，是 MOV AL，[BX+SI]的替换形式。

32. (a) 11750H (b) 11950H (c) 11700H

34. BP 或 EBP

36.
```
FIELDS    STRUC
F1        DW   ?
F2        DW   ?
F3        DW   ?
F4        DW   ?
F5        DW   ?
FIELDS    ENDS
```

38. 直接、相对和间接。

40. 段间跳转是一个远跳转，允许转至存储系统内的任意位置，而段内跳转是一个近跳转，只允许访问当前代码段内的任何位置。

42. 32 位

44. 短的

46. JMP BX

48. 2

50. AX，CX，DX，BX，SP，BP，DI 和 SI，就以这个顺序存放。

52. PUSHFD

第 4 章

2. D 位指示数据的流向（REG 向 R/M，或 R/M 向 REG）；W 位指示数据的长度（字节或者字/双字）。

4. DL

6. DS：[BX + DI]

8. MOV AL，[BX]

10. 8B 77 02

12. REX 前缀用于 64 位平展模式，它是一个在一条指令中允许寻址 64 位寄存器的寄存器扩展。

14.
```
MOV AX，1000H
MOV DS，AX
```

16. PUSH RAX

18. AX、CX、DX、BX、SP、BP、SI 和 DI

20. (a) PUSH AX 指令将 AX 内容压入堆栈。
(b) POP ESI 指令将一个 32 位数字弹出堆栈并放入 ESI。
(c) PUSH [BX] 指令将数据段中由 BX 寻址的存储单元的字压入堆栈。
(d) PUSHFD 指令将 EFLAGS 寄存器压入堆栈。
(e) POP DS 指令将一个 16 位数字弹出堆栈写入 DS。
(f) PUSHD 4 指令将 32 位数字 4 压入堆栈。

23. PUSH EAX 指令将 EAX 的 31～24 位存入 020FFH 存储单元，23～16 位存入 020FEH 存储单元，15～8 位存入 020FDH 存储单元，7～0 位存入 020FCH 存储单元。然后将 SP 的内容加 4，即 SP 的结果为 00FCH。

24. 一种可能的组合是两个寄存器均为 200H，SP = 0200H；SS = 0200H。

26. 两条指令均将 NUMB 的地址装入 DI，不同是 MOV DI，OFFSET NUMB 以立即数传送指令汇编；而 LEA DI，NUMB 以装入有效地址指令汇编。

28. 这条指令将存于数据段存储单元 NUMB 中的字装入 BX，并将数据段中由 NUMB + 2 寻址的字装入 DS 。

30.
```
MOV  BX,NUMB
MOV  DX,BX
MOV  SI,DX
```

32. CLD 指令清除方向标志，而 STD 指令使方向标志置 1。

34. LODSB 指令将由 SI 寻址的数据段存储单元的字节数据装入 AL，然后如果方向标志是清 0 的则 SI 加 1。

36. OUTSB 指令将数据段由 SI 寻址的存储单元的内容输出到由 DX 寻址的 I/O 端口，然后如果方向标志是清 0 的则 SI 加 1。

38.
```
MOV  SI,OFFSET SOURCE
MOV  DI,OFFSET DEST
MOV  CX,12
REP  MOVSB
```

40. XCHG EBX, ESI
42. LAHF 和 SAHF 指令用于具有算术协处理器的非 64 位应用程序。
44. XLAT 指令将 AL 中内容与 BX 中内容相加形成一数据段中的偏移地址，然后将此地址中内容复制到 AL 中。
46. OUT DX, AX 指令表示把 AX 中的 16 位内容复制到数据段由 DX 寄存器寻址的存储单元。
48. MOV AH, ES: [BX]
50. 汇编语言伪指令是给汇编器的特殊命令，指示它生成代码和数据并将其存储到存储器中，或者不生成代码和数据。
52. 伪指令 DB、DW、DD 分别用于定义一个字节、一个字和双字的内存。
54. EQU 伪指令允许一个内存单元可以和另一个内存单元相同。
56. MODEL 伪指令指定一个程序所使用的内存模式的类型。
58. 完整段定义
60. PROC 表示过程的开始，而 ENDP 表示过程的结束。
62.
```
STORE  PROC   NEAR
       MOV    [DI],AL
       MOV    [DI+1],AL
       MOV    [DI+2],AL
       MOV    [DI+3],AL
       RET
STORE  ENDP
```
64.
```
COPY  PROC   FAR
      MOV    AX, CS: DATA4
      MOV    BX, AX
      MOV    CX, AX
      MOV    DX, AX
      MOV    SI, AX
      RET
COPY  ENDP
```

第 5 章

2. 两个寄存器的长度不匹配。
4. AX = 3100H, C = 0, A = 1, S = 0, Z = 0, O = 0
6.
```
ADD  AX,BX
ADD  AX,CX
ADD  AX,DX
ADD  AX,SP
```
8.
```
MOV  DI,AX
MOV R12, RCX
ADD  R12,RDX
ADD  R12,RSI
```
10. INC SP
12. (a) SUB CX, BX (b) SUB DH, OEEH (c) SUB SI, DI (d) SUB EBP, 3322H (e) SUB CH, [SI] (f) SUB DX, [SI + 10] (g) SUB FROG, AL (h) SUB R10, R9
14.
```
MOV  BX, AX
SUB  BX, DI
SUB  BX, SI
SUB  BX, BP
```
16. 从由 DI - 4 寻址的数据段 16 位存储单元中减去 DX 的内容和借位标志，把结果放到 DX 中。
18. AH（最高有效位）和 AL（最低有效位）
20. O 和 C 标志包含乘积中最高有效位部分的状态。如果乘积中最高有效位的部分是零，那么 C 和 O 也是零。
22.
```
MOV  DL,5
MOV  AL,DL
MUL  DL
MUL  DL
```
24. BX = DX × 100H
26. AX
28. 在除法中可以检测到两种错误：一种是除法溢出，一种是试图除以 0。
30. AH
32.
```
MOV    AH, 0
MOV    AL, BL
DIV    CL
ADD    AL, AL
MOV    DL, AL
MOV    DH, 0
ADC    DH, 0
```
34. AAM 指令把 AX 转换为 BCD 码，通过用 10 来除 AX，与一般除法不同的是，把十进制数 0 ~ 99 转换为非压缩 BCD 码，余数放在 AL 里面，而商放入 AH 中。
36.
```
PUSH  DX
PUSH  CX
MOV   CX, 1000
DIV   CX
MOV   [BX], AL
MOV   AX, DX
POP   CX
POP   DX
PUSH  AX
AAM
MOV   [BX+1], AH
MOV   [BX+2], AL
POP   AX
MOV   AL, AH AAM
MOV   [BX+3], AH
MOV   [BX+4], AL
```
38. 64 位模式下既没有 BCD 也没有 ASCII 指令函数。
40.
```
MOV   BH, DH
AND   BH, 1FH
```
42.
```
MOV    SI,DI
OR     SI,1FH
```
44.
```
OR     AX,0FH
AND    AX,1FFFH
XOR    AX,0380H
```
46. TEST CH, 4
48. (a) SHR DI, 3 (b) SHL AL, 1 (c) ROL AL, 3 (d) RCR EDX, 1 (e) SAR DH, 1
50. 附加段
52. 当 CX 值不为 0 或者比较得出一个相等条件，则重复 SCASB 指令操作。
54. CMPSB 指令比较数据段由 SI 寻址的字节内容和附加段由 DI 寻址的字节内容。

56. 在 DOS 中显示字母 C。

第 6 章

2. 近跳转
4. 远跳转
6. (a) 近 (b) 短 (c) 远
8. EIP/IP 寄存器
10. JMP AX 指令跳转至 AX 中的偏移地址，这是一次近跳转。
12. JMP [DI] 指令执行一次近跳转，转向的目标地址在当前数据段由 DI 寻址的存储单元中。而 JMP FAR PTR [DI] 指令执行一次远跳转，转向数据段由 DI + 2 寻址的存储单元来寻址的新数据段，新偏移地址在数据段由 DI 寻址的存储单元中。
14. JA 指令测试算术或逻辑的条件确定结果是否高于，当目的操作数比源操作数高时 JA 指令执行跳转；否则不跳转。
16. JNE，JE，JG，JGE，JL 和 JLE
18. JA 和 JBE
20. SETZ 或 SETE
22. ECX
24.
```
        MOV  DI,OFFSET DATAZ
        MOV  CX,150H
        CLD
        MOV  AL,00H
    L1: STOSB
        LOOP L1
```
26.
```
        CMP  AL,3
        JNE  @ C0001
        ADD  AL,2
    @ C0001:
```
28.
```
        MOV  SI,OFFSET BLOCKA
        MOV  DI,OFFSET BLOCKB
        CLD
        .REPEAT
              LODSB
              STOSB
        .UNTIL AL = = 0
```
30.
```
        MOV  AL,0
        MOV  SI,OFFSET BLOCKA
        MOV  DI,OFFSET BLOCKB
        CLD
        .WHILE AL ! = 12H
              LODSB
              ADD  AL, [DI]
              MOV  [DI],AL
              INC  DI
        .ENDW
```
32. 过程是一个可重复使用的指令组，以 RET 结束。
34. RET
36. 通过在 PROC 伪指令的右边使用 NEAR 或 FAR。
38.
```
CUBE  PROC  NEAR USES AX DX
      MOV   AX,CX
      MUL   CX
      MUL   CX
      RET
CUBE  ENDP
```
40.
```
SUMS  PROC  NEAR
      MOV   EDI,0
      ADD   EAX,EBX
      ADD   EAX,ECX
      ADD   EAX,EDX
      ADC   EDI,0
      RET
SUMS  ENDP
```
42. 中断是一种硬件驱动的功能调用。
44. INT 0 到 INT 255
46. 该中断向量用于确定并响应除法错误。
46. RET 指令从堆栈中弹出返回地址，而 IRET 指令不但从堆栈中弹出返回地址还弹出标志寄存器内容。
48. IRETD 是一个 32 位返回指令并将返回地址弹给 EIP。
50. 当溢出标志置 1 时，发生 INT 0 并中断程序。
52. CLI 和 STI
54. 当目的操作数寄存器或存储单元中的值被测试出高于或低于存储在由源操作数寻址的存储器里的上、下限时，产生中断。
56. BP

第 7 章

2. 不能，在 C++ 中字节必须用 char 定义。
4. EAX，EBX，ECX，EDX 和 ES
6. 浮点协处理器堆栈
8. 指令把 SI 作为指向 string1 的一个字符的指针，把字符复制到 DL 寄存器中。
10. 如果 C++ 程序没有包含头文件库，它会短很多。
12. 不能，INT 21H 是 16 位 DOS 系统调用，不能用于 Windows32 位环境。
14.
```
#include "stdafx.h"
#include <conio.h>
int _tmain (int argc, _ TCHAR* argv [])
{
     char a = 0;
     while (a ! = '@ ')
     {
          a = _getche ();
          _putch (a);
     }
     return 0;
}
```
16. 指令_putch (10) 显示新的一行，指令_putch (13) 使光标返回到显示器的最左边。
18. 单独汇编模块最灵活。
20. 必须与 C 原型程序一起用的平展模型，像. MODEL FLAT、C 和链接到 C ++ 必须构成公有的函数。
22. 用 short 伪指令定义 16 位字。
24. 事件的例子有：移动鼠标、按下了键等，事件管理程序捕获这些事件，以便在程序中能够使用他们。
26. 可以，C ++ 编辑屏幕可以编辑汇编语言模块，但模块必须用 . TXT 扩展名替换 . ASM。

28.
```
#define RDTSC _asm _emit 0x0f _asm _emit 0x31
```

30.
```
;
;把字节循环左移 3 位的外部函数
;
.586                    ;选择 Pentium,32 位模型
.model flat,C           ;选择与 C/C++相关的平展模型
.stack 1024             ;分配堆栈空间
.code                   ;代码段开始

public RotateLeft3  ;定义 RotateLeft3 为公有函数

RotateLeft3 proc    ;定义过程
Rotatedata:byte     ;定义字节
     mov    al,Rotatedata
     rol    al,3
     ret
RotateLeft3  endp
```

32.
```
;转换函数
;
.model  flat,C
.stack  1024
.code
Public  Upper
Upper   proc
Char:byte

       mov  al,Char
       .if  al >= 'a' && al <= 'z'
                  sub al,30h
       .endif
       ret
Upper  endp
```

34. Properties 包含有关对象的信息，如前台彩色和背景颜色等。

36.
```
_asm inc ptr;
```

第 8 章

2. TEST. ASM 文件被汇编后，生成 TEST. OBJ 文件，如果没有选择命令文件开关，就生成 TEST. EXE 文件。
4. PUBLIC 声明一个标号对其他模块是公有的。
6. EXTRN
8. MACRO 和 ENDM
10. 将参数放在“MACRO”关键字后面（在同一行），使其传送到宏序列。
12. LOCAL 伪指令定义局部变量，LOCAL 伪指令必须紧跟在 MACRO 语句行后。

14.
```
ADDM  MACRO LIST,LENGTH
      PUSH  CX
      PUSH  SI
      MOV   CX,LENGTH
      MOV   SI,LIST
      MOV   AX,0
      .REPEAT
           ADD  AX, [SI]
           INC  SI
     .UNTILCXZ
     POP   SI
     POP   CX
     ENDM
```

16.
```
private: System::Void textBox1_KeyDown
(System::Object^ sender,System::
Windows::Forms::KeyEventArgs^ e)
{
      //这是第一位调用
      keyHandled = true;
      if (e->KeyCode >= Keys::D0 &&
            e->KeyCode <= Keys::D9 &&
            e->Shift == false)
      {
            keyHandled = false;
      }
}
```

18.
```
private: System::Void textBox1_KeyDown
(System::Object^ sender,System::Windows::
Forms::KeyEventArgs^ e)
{
    {
        RandomNumber++;   // 全局变量
        if ( RandomNumber == 63 )
            RandomNumber = 9;
    }
```

20.
```
bool direction;

private: System::Void button1_Click(System
::Object^ sender,System::EventArgs^ e)
{
      label1->Text = "Shift Left = ";
      shift = true;
      data1 = 1;
      label2->Text = "00000001";
      timer1->Start();
}

private: System::Void button1_Click(System::
Object^ sender,System::EventArgs^ e)
{
      label1->Text = "Rotate Left = ";
      shift = false;
      data1 = 1;
      label2->Text = "00000001";
      timer1->Start();
}

private: System::Void radiobutton1_Click
(System::Object^ sender,System::EventArgs^ e)
{
      // 左键
      direction = true;  // 新布尔变量
      if ( shift )
            label1->Text = "Shift Left = ";
      else
            label1->Text = "Rotate Left = ";
}

private: System::Void radiobutton2_Click
(System::Object^ sender,System::EventArgs^ e)
{
      // 左键
      direction = false; // 新布尔变量
      if ( shift )
            label1->Text = "Shift Right = ";
      else
            label1->Text = "Rotate Left = ";
}

private: System::Void timer1_Tick(System::
Object^ sender,System::EventArgs^ e)
{
      String^ temp = "";
      char temp1 = data1;
      if ( shift )
                  if ( direction )
                        _asm shl temp1,1;
                  else
                        _asm shr temp1,1;
        else

                  if ( direction )
                        _asm rol temp1,1;
                  else
                        _asm ror temp1,1;
        data1 = temp1;
        for (int a = 128; a > 0; a>>=1)
        {
                  if ( ( temp1 & a ) == a )
                        temp += "1";
                  else
                        temp += "0";
        }
        label2->Text = temp;
  }
```

22. MouseDown 中的 MouseEventArgs 包含按钮状态，这一状态在程序中通过:: mouses:: MouseButtons:: Right 拦截右鼠标按钮可以对其测试。

24.
```
private: System::Void Form1_MouseDown
(System::Object^ sender,System::Windows::
Forms::MouseEventArgs^ e)
{
      if (e->Button ==
       ::mouses::MouseButtons::Left &&
       e->Button  = ::mouses::MouseButtons::
       Right)
      {
                  //left and right
      }
}
```

26. ForeColor 用来设置一个控件中文本或字符的颜色。
28. 重复地除以 10，然后将余数保存作为 BCD 数的有效位。
30. 30H

32.
```
public GetMultiple()
{
    String[] temp;
    int result = 1;
    char sampl;
    temp = (val[0]).ToCharArray();
    for ( int a = 0; a < temp->Length; a++ )
    {
        sampl = temp.at(a);
        __asm
        {
            sbb result,1
            mov eax,0
            mov al,sampl
            sub al,100
            mul result,eax
        }
    }
    return result;
}
```

34.
```
char Up (char temp)
{
    if ( temp > = 'a' && temp < = 'z' )
        _asm sub temp,32;
    return temp;
}
```

36. 引导扇区中含有一个引导加载程序，将操作系统载入内存。FAT（文件分配表）中存有数字，标明哪些扇区是空的，哪些损坏了，哪些被占用的。如果已被占了，数据 FFFFH 指示文件结尾。其他数据是在文件链中的下一个簇号。目录保存有关文件或文件夹的信息。

38. 扇区

40. 禁止一个扇区。

42. 4GB

44. 8

46. 256

48.
```
File::SysOpen("TEST.DAT", SYSRW | 0x0, SYSRW |
DAK2);
// creates test.dat and OPEN.DAK. Deletes
TEST.DAT
```

50. 控件是公用的元素，可以用于 Visual 程序发开语言里。

52. 输出见图 D-1（注 ListBox 包含输出）

```
// 单击 OnInitDlg 函数响应代码
int temp=1, i;
for ( i=0; i < 5; i++ )
{
    CString temp = "I'm " + i;
    temp.SetAt (2,i   + 0x30);
    temp = GetNum( temp+1);
    List.InsertString (m,temp);
    temp++ = 2;
}
String CMyDialog::GetNum (int temp)
{
    char num [10];
    _asm
    {
        mov eax,temp
        mov ecx,10
        push eax
        mov edx,0
        div ebx
        push eax
        cmp eax,0
        jne loop1
loop1:
        mov ecx,0
        mov eax,ecx
        pop ebx
        add eax,30h
        mov num[ecx],al
        inc ecx
        jmp loop2
loop2:
        mov byte ptr num[ecx],0
    }
    return num;
}
```

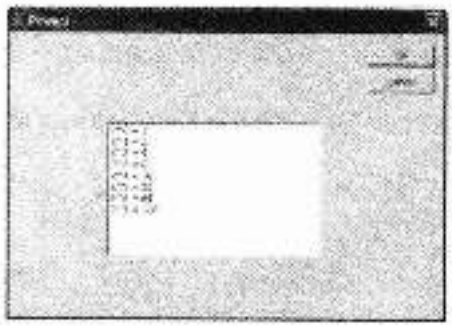


图 D-1

54.
```
private: System::Void Clear()
{
    panel1->Visible = false;
    panel2 =Visible = false;
    panel3->Visible = false;
    panel4 =Visible = false;
    panel5->Visible = false;
    panel6->Visible = false;
    panel7 =Visible = false;
}
private: System::Void Form1_KeyDown(System::
Object^ sender,System::Windows::Forms::
KeyEventArgs^ e)
{
    char LookUp[] = {0x3f, 6, 0x5b, 0x4f,
        0x66, 0x6d, 0x7d, 7, 0x7f,0x6f,
        0x77, 0x7c, 0x39, 0x5e, 0x79,
        0x71};
    if (e->KeyCode >= Keys::D0 &&
        e->KeyCode <= Keys::D9)
    {
        Show0Digit(LookUp
            [e->KeyValue -
            0x30]);  //display
                     the digit
    }
    else if (e->KeyCode >= Keys::A &&
        e->KeyCode <= Keys::F)
    {
        Show0Digit(LookUp
            [e->KeyValue
            0x37]);  //display
                     letter
    }
}
private: System::Void Show0Digit(unsigned
char value)
{
    Clear();
    if (( code & 1 ) == 1)  //test a
                            segment
        panel1->Visible = true;
    if (( code & 2 ) == 2)  //test b
                            segment
        panel4->Visible = true;
    if (( code & 4 ) == 4)  //test c
                            segment
        panel5->Visible = true;
    if (( code & 8 ) == 8)  //test d
                            segment
        panel3->Visible = true;
    if (( code & 16 ) == 16)  //test e
                            segment
        panel2->Visible = true;
    if (( code & 32 ) == 32)  //test f
                            segment
        panel7->Visible = true;
```

```
        if (( code & 64 ) == 64)  //test g
                                    segment
                    panel2->Visible = true;
}
private: System::Void Form1_Load(System::
Object^ sender,System::EventArgs^ e)
{
        Clear();
}
```

第9章

2. 只要逻辑0驱动电流不超过2.0mA并且噪声容限不超过350mV它们就是TTL兼容的。
4. 地址位 $A_7 \sim A_0$。
6. 读操作。
8. 占空比为33%的方波。
10. 出现一个写操作。
12. 数据总线正在向存储器或I/O传送数据。
14. IO/$\overline{M}$、DT/$\overline{R}$ 和$\overline{SS0}$
16. 队列状态位为协处理器指示微处理器内部队列的状态。
18. 3
20. 14MHz/6 = 2.33MHz
22. 地址总线 $A_0 \sim A_{15}$
24. 74LS373八位透明锁存器。
26. 如果有过多的存储器和I/O设备连接到系统总线就需要缓冲器。
28. 4
30. 取和执行
32. (a) 与ALE一起输出地址。
    (b) 存储器访问允许时间并且在此采样READY输入。
    (c) 发出读或写信号。
    (d) 传送数据并释放读或写。
    (e) 等待允许增加存储器访问时间。
36. 为READY选择一个或两个同步周期。
38. 最小模式操作经常用于嵌入式应用；最大模式操作经常用于早期PC机。

第10章

2. (a) 256 (b) 2K (c) 4K (d) 8K (e) 1M
4. 选择存储器。
6. 执行一次写操作。
8. 5MHz微处理器允许存储器用460ns的时间来存取数据，但由于连接到存储器时还有延时，因此在这样的系统里450ns的存储器不能很好地工作，除非加一个等待状态。
10. 静态RAM。
12. 250ns
14. 大多数DRAM的地址输入是多路复用的，因而一个地址输入访问两个不同的地址位，对于DRAM存储器需要减少地址引脚数。
16. 一般时间值相当于一个读周期，在现代存储系统中这只是一个很短的时间。
18. 见图D-2。

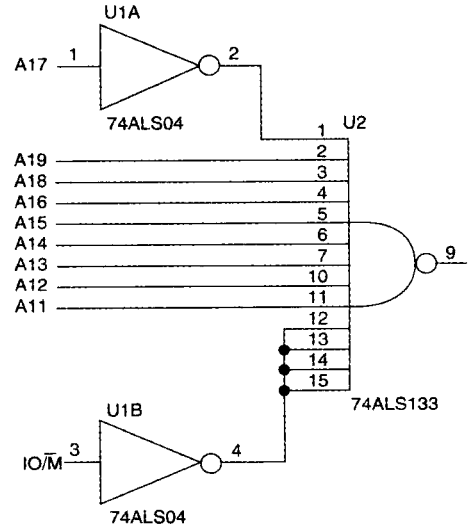


图 D-2

20. 由地址输入指示的8个输出中，有一个输出变成逻辑0。
22. 见图D-3。

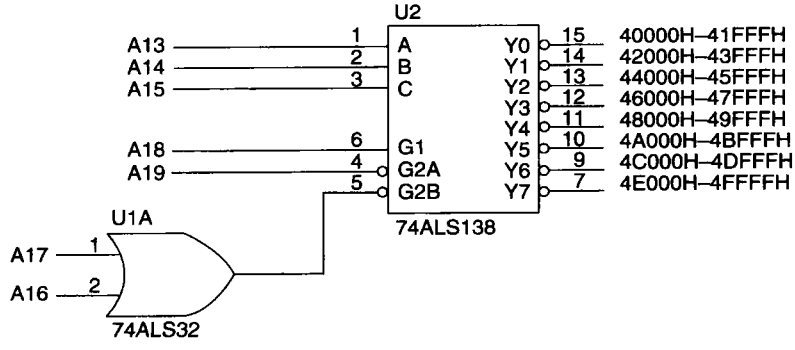


图 D-3

24. Verilog 硬件描述语言。
26. begin 和 end 之间的结构模块。
28. $\overline{MRDC}$和$\overline{MWTC}$
30. 见图 D－4。

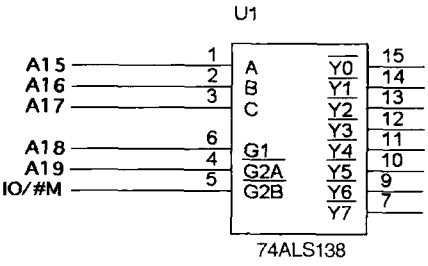


图 D-4

32. 5
34. 1
36. $\overline{BHE}$选择高位存储体，A_0 选择低位存储体。
38. 独立译码器或独立写控制信号
40. 低位存储体
42.
```
library ieee
use ieee.std_logic_1164.all;
entity DECODER_10_28 is
port (
      A23,A22,A21,A20,A19,A18,A17,
      A16,A0,BHE,MWTC: in STD_LOGIC;
      SEL,LWR,HWR: out STD_LOGIC
);
  end;
architecture V1 of DECODE_10_28 is

begin
   SEL < = A23 or A22 or A21 or A20 or A19
    or A18 or (not A17) or (not A16);
    LWR < = A0 or MWTC;
    HWR < = BHE or MWTC;
end V1;
```
44. 见图 D-5。
48. 可以，只要不访问 DRAM 上的存储单元。
50. 128 位宽

第 11 章

2. 固定 I/O 端口存储在指令的第二个字节里。
4. 寄存器 DX。
6. OUTSB 指令将 SI 寻址的数据段存储单元内容传送到 DX 寻址的 I/O 端口，然后 SI 的值加 1。
8. 存储器映像 I/O 用任何数据传送指令在存储器与 I/O 之间传送数据，而独立编址 I/O 要求用 IN 和 OUT 指令。
10. 基本输出接口是一个锁存器，它获得并为输出设备保持输出数据。
12. 低位存储体
14. 4
16. 消除开关的机械抖动。
18. 见图 D-6。
20. 见图 D-7。
22. 见图 D-8。
24. 如果端口是 16 位宽，高、低两半均不需要允许输入。
26. D_{47}～D_{40}
28. A 组是 A 口和 PC_4～PC_7 的引脚，而 B 组是 B 口和 PC_3～PC_0 的引脚。
30. $\overline{RD}$
32. 输入
34. 选通输入锁存输入数据，将缓冲器满标志和中断请求置为 1。
36.
```
DELAY  PROC  NEAR USES ECX

       MOV   ECX,7272727
D1:
       LOOPD D1
       RET
DELAY  ENED
```
38. 选通信号（$\overline{STB}$）
40. 由 PC_4（A 口）或 PC_2（B 口）的 INTE 位置 1 来允许中断请求引脚 INTR。
42. 当数据输出到端口时$\overline{OBF}$变为 0，当$\overline{ACK}$发送到端口时$\overline{OBF}$变为 1。
44. A 组即 A 口包含双向数据。
46. 把 01H 命令发送给 LCD 显示器。
48.
```
;显示由 DS:BX 寻址的空 ASCII 字符串,
;用宏调用 SEND 发送数据到显示器。
;
DISP  PROC  NEAR  USES  BX
      SEND  86H,2,1      ;移动光标到位置 6
     .WEILE BYTE PTR [BX] ! =0
            SEND [BX], 0, 1
            INC BX
     . ENDW
     RET
DISP  ENDP
```
50. 惟一要作的修改是把 4 行换成 3 行并把 3 个上拉电阻连到 A 口，以及把 5 列连到 B 口。当然软件也需要作少许修改。
54. 6
58. 最低有效字节。
62. 异步串行数据是无须时钟脉冲即被发送的数据。
64.
```
LINE  EQU  023H
LSB   EQU  020H
MSB   EQU  021H
FIFO  EQU  022H

      MOV  AL,10001010B;使能波特率除数
      OUT  LINE,AL

      MOV  AL,60        ; 编程波特率
      OUT  LSB,AL
      MOV  AL,0
      OUT  MSB,AL

      MOV  AL,00011001B;编程为 7 位数据，奇校验
      OUT  LINE,AL      ; 一个停止位

      MOV  AL, 00000111B; 允许发送器和接收器
      OUT  FIFO, AL
```

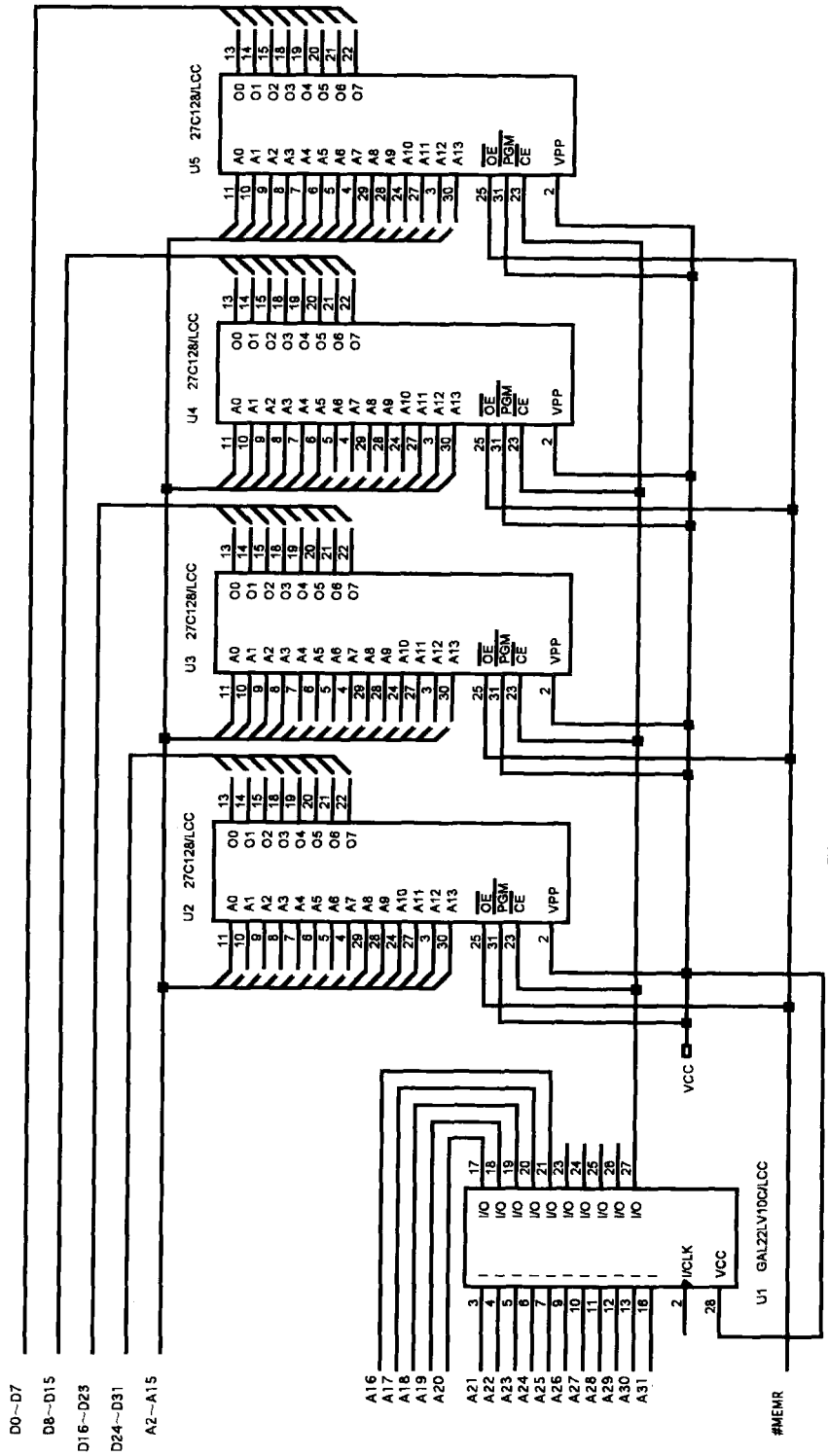

U5 27C128/LCC
U4 27C128/LCC
U3 27C128/LCC
U2 27C128/LCC
U1 GAL22LV10C/LCC
D0~D7
D8~D15
D16~D23
D24~D31
A2~A15
A16
A17
A18
A19
A20
A21
A22
A23
A24
A25
A26
A27
A28
A29
A30
A31
VCC
#MEMR

图 D-5

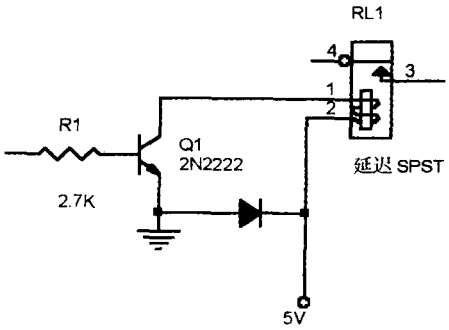


图 D-6

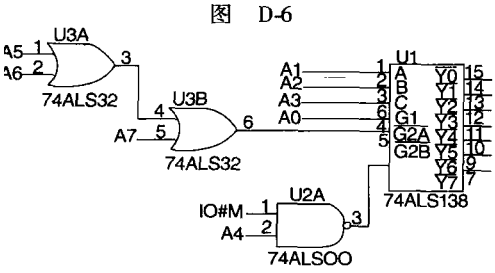


图 D-7

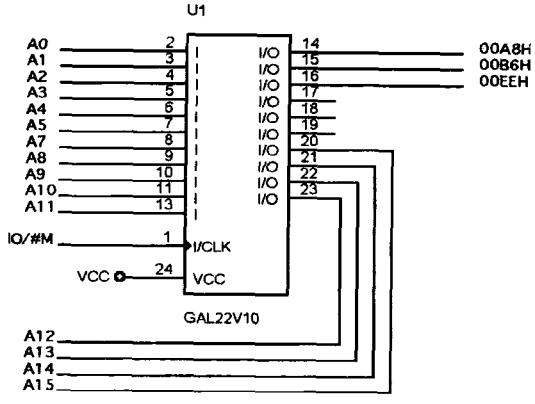


图 D-8

66. 单工 = 接收数据或者发送数据。半双工 = 发送与接收数据，但一次只能一个方向。双工 = 同时发送与接收数据。

68.
```
SENDS  PROC  NEAR

MOV     CX,16
        .REPEAT
            .REPEAT
                IN    AL,LSTAT  ;取行状态寄存器
                TEST  AL,20H    ;测试 TH 位
            .UNTIL !ZERO?
            LODSB               ;取数据
            OUT  DATA,AL        ;传送数据
        .UNTILCXZ
    RET
SENDS ENDP
```

70. 0.01V

72.
```
        .MODEL TINY
        .CODE
        .STARTUP
                MOV     DX,400H
                .WHILE 1
                MOV     CX,256
                MOV     AL,0
                .REPEAT
                    OUT    DX,AL
                    INC    AL
                    CALL   DELAY
                .UNTILCXZ
                MOV    CX,256
                .REPEAT
                     OUT      DX,AL
                     DEC      AL
                     CALL     DELAY
                .UNTILCXZ
                .ENDW
DELAY           PROC    NEAR

;39 微秒时延

DELAY  ENDP
       END
```

74. INTR 引脚指示转换器已经完成转换。

76. 见图 D-9。

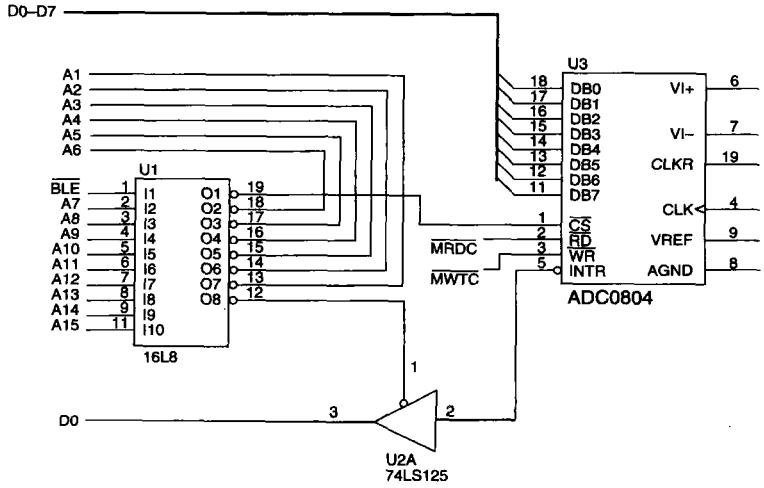


图 D-9

第 12 章

2. 中断是一次硬件或软件触发的子程序调用。
4. 中断只有在中断有效时才使用处理器时间。
6. INT，INT_3，INTO，CLI，STI
8. 在实模式中是在存储系统的第一个 1KB 字节，在保护模式中可在任何地方。
10. 00H ～ 1FH
12. 可位于存储系统的任何位置。
14. 实模式中断把 CS，IP 和 FLAGS 压入堆栈，而保护模式中断把 CS，EIP 和 EFLAGS 压入堆栈。
16. 如果置位溢出标志则发生 INTO 中断。
18. IRET 指令从堆栈弹出标志和返回地址。
20. 把中断的现场存到堆栈，这样返回时再恢复它，中断标志位和跟踪标志位均被清 0。
22. IF 标志控制 INTR 引脚被允许或禁止。
24. 通过使用 STI 或者 CLI 指令允许或禁止 IF 标志。
26. 2
28. 电平有效
30. 向量
32. 见图 D-10。

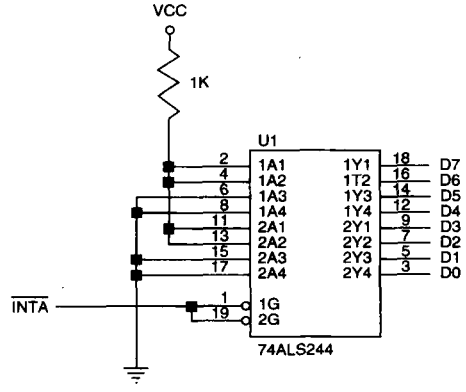


图 D-10

34. 当中断应答是 FFH 时，上拉电阻用来保证从数据总线获取的向量数目。
36. 因为中断请求信号（INTR）的产生是由所有的请求一块提供，那么软件必须查询每一个设备，来决定是哪个设备产生了中断请求。
38. 9
40. CAS 引脚是级联引脚，用来在多于 8 个中断输入时 8259 芯片的级联。
42. OCW 是 8259 的操作命令字。
44. ICW_2
46. 编程触发方式和单个或多个 8259。
48. 开始服务后将最近的中断请求置为最低中断优先级。
50. INT 8 到 INT 0FH

第 13 章

2. 一旦 HOLD 被置为逻辑 1，程序停止执行，地址、数据和控制总线置为高阻状态。
4. 从 I/O 到存储器
6. DACK
8. 微处理器处于它的 HOLD 状态，而 DMA 控制器控制总线。
10. 4
12. 命令寄存器
16. 笔式驱动器一种 USB 设备，把闪速存储器当做存储设备用。
18. 磁道
20. 柱面
22. 见图 D-11。

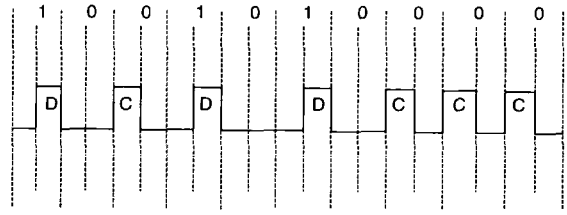


图 D-11

24. 硬盘驱动器内的磁头按流线型设计，在磁盘旋转产生的空气垫上飞行，因此也叫飞行磁头。
26. 步进电机定位机构有噪音而且定位不是很精确，而音圈电机定位机构安静而且定位很精确，因为它的安放可以连续调整。
28. CD-ROM 是一种存储音乐或数据的光学设备，容量大约有 660M 或 700M（80 分钟）字节。
30. TTL 显示器用 TTL 信号产生显示，而模拟显示器用模拟信号产生显示。
32. 青色、紫色、和黄色。
34. 1024 行，每行包含 1280 个像素。
36. DVI-D 和 HDMI 是最新的数字视频输入连接器，支持所有类型视频装置。
38. 1600 万种颜色。

第 14 章

2. 字（16 位，±32K），双字（32 位，±2G），和四字（64 位，$\pm 9 \times 10^{18}$）。
4. 单精度（32 位）、双精度（64 位）和临时精度（80 位）。
6. (a) −7.75 (b) +0.5625 (c) 76.5
   (d) 2.0 (e) 10.0 (f) 0.0
8. 当协处理器执行浮点指令时，微处理器继续执行微处理器的（整数）指令。
10. 把协处理器状态寄存器复制到 AX 中。
12. 比较两个寄存器，然后把状态字传送到 AX。如果 SAHF 指令后紧接着执行 JZ 指令就可以用来测试协处理器比较指令的结果。
14. FSTSW AX
16. 数据总是以 80 位临时精度形式存储。
18. 0
20. 仿射允许正、负无穷大，而投射假定无穷大是无符号数。
22. 扩展精度（临时精度）数
24. 把数据从堆栈顶的内容复制到 DATA 存储单元，形成浮点数。
26. FADD ST，ST（3）

28. FSUB ST (2), ST

30. 前向除法是堆栈顶的内容除以存储单元的内容，并把商返回到堆栈顶；而反向除法是存储单元的内容除以堆栈顶内容，并把商返回到堆栈顶。如果没有操作数，前向除法由 ST (1) 除以 ST (0)。而反向除法用 ST (0) 除以 ST (1)。

32. 如果小于，该指令完成传送到 ST 操作。

34.
```
RECIP  PROC    NEAR
       MOV  TEMP,EAX
       FLD  TEMP
       FLD1
       FDIVR
       FSTP TEMP
       MOV  EAX,TEMP
RECIP  ENDP
TEMP   DD  ?
```

36. 求函数 2^X-1。

38. FLDPI

40. 它指示释放 ST (2) 寄存器。

42. 机器的状态

44.
```
CAPR  PROC  NEAR
      FLDPI
      FADD ST,ST (1)
      FMUL F
      FMUL C1
      FLD1
      FDIVR
      FTSP XC
      RET
CAPR  ENDP
```

46. 在当今软件中从来不用它。

48.
```
TOT  PROC  NEAR
     FLD  R2
     FLD1
     FDIVR
     FLD  R3
     FLD1
     FDIVR
     FLD  R4
     FLD1
     FDIVR
     FADD
     FADD
     FLD1
     FDIV
     FADD  R1
     FSTP  RT
     RET
TOT  ENDP
```

50.
```
PROD PROC  NEAR
     MOV  ECX,100
     .REPEAT
            FLD  ARRAY1 [ECX* 8 -8]
            FUML ARRAY2 [ECX* 8 -8]
            FSTP ARRAY3 [ECX* 8 -8]
      .UNTILCXZ
      RET
PROD  ENDP
```

52.
```
POW   PROC  NEAR
      MOV  TEMP,EBX
      FLD  TEMP
      F2XM1
      FLD1
      FADD
      MOV  TEMP,EAX
      FLD  TEMP
      FYL2X
      FSTP  TEMP
      MOV  ECX,TEMP
      RET
POW   ENDP
```

54.
```
GAIN PROC  NEAR
     MOV  ECX, 100
     .REPEAT
           FLD  DWORD PTR VOUT [ECX* 4 -4]
           FDIV  DWORD PTR VIN [ECX* 4 -4]
           CALL  LOG10
           FIMUL  TWENTY
           FSTP  DWORD PTR DBG [ECX* 4 -4]
     .UNTILCXZ
     RET
TWENTY DW   20
GAIN   ENDP
```

56. EMMS 指令清除协处理器的堆栈来表明使用协处理器寄存器组已经完成了 MMX 扩展。

58. 对于字节型数据，当数据相加出现上溢而值为 7FH，或者出现下溢而值为 80H 的时候，则产生带符号的饱和。

60. FSAVE 指令把所有 MMX 寄存器存入内存中。

62. 单指令多数据流指令

64. 128 位

66. 16

68. 可以

第 15 章

2. 8 位或 16 位，取决于插座配置。

4. 参见图 D-12。

6. 参见图 D-13。

8. 参见图 D-14。

12. 16 位

14. 配置内存标识供应商和有关中断的信息。

16. 这是个命令/总线使能信号，高表示 PCI 总线上有个命令；低表示 PCI 总线上是个数据。

18.
```
MOV  AX,0B108H
MOV  BX,0
MOV  DI,8
INT  1AH
```

20. 2.5GHz

22. 可以

24. COM_1

30. 1.5Mbps，12Mbps，和 480Mbps

32. 5 米

34. 127

36. 是填入数据流中额外加的一位，如果数据流多于六个 1，则发送一个低。

38. 1 到 1023 个字节。

40. PCI 以 33MB/s 传送数据，AGP 以 2GB/s (8 ×) 传送数据。

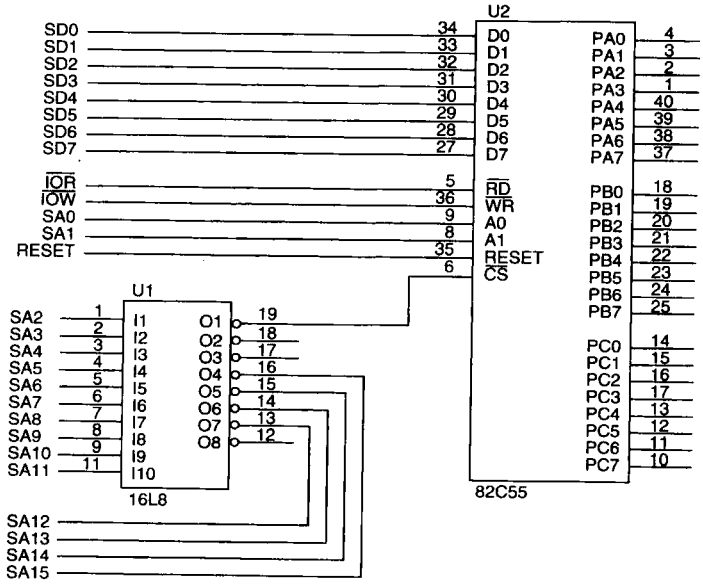

U2
SD0
SD1
SD2
SD3
SD4
SD5
SD6
SD7
IOR
IOW
SA0
SA1
RESET
D0
D1
D2
D3
D4
D5
D6
D7
RD
WR
A0
A1
RESET
CS
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
PC0
PC1
PC2
PC3
PC4
PC5
PC6
PC7
82C55
U1
SA2
SA3
SA4
SA5
SA6
SA7
SA8
SA9
SA10
SA11
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
O1
O2
O3
O4
O5
O6
O7
O8
16L8
SA12
SA13
SA14
SA15

图 D-12

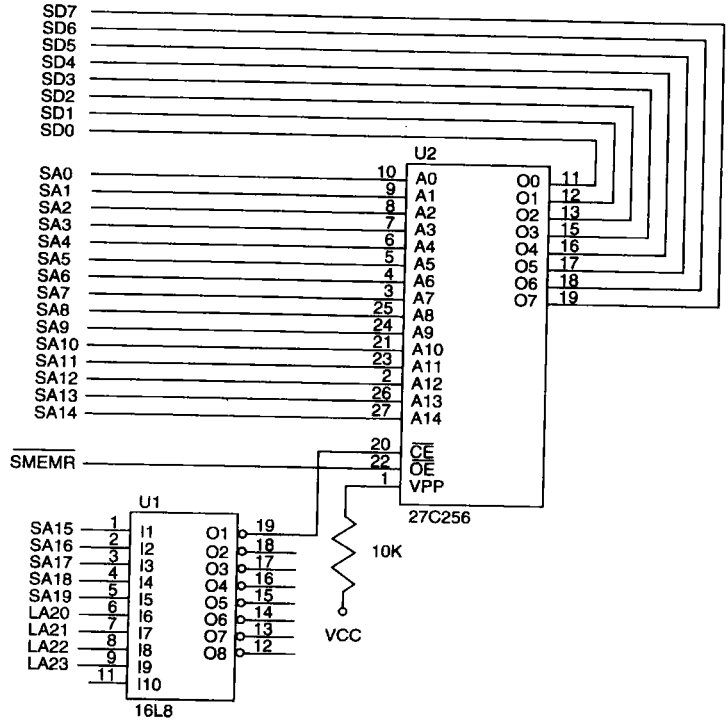

SD7
SD6
SD5
SD4
SD3
SD2
SD1
SD0
U2
SA0
SA1
SA2
SA3
SA4
SA5
SA6
SA7
SA8
SA9
SA10
SA11
SA12
SA13
SA14
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
O0
O1
O2
O3
O4
O5
O6
O7
SMEMR
CE
OE
VPP
27C256
10K
VCC
U1
SA15
SA16
SA17
SA18
SA19
LA20
LA21
LA22
LA23
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
O1
O2
O3
O4
O5
O6
O7
O8
16L8

图 D-13

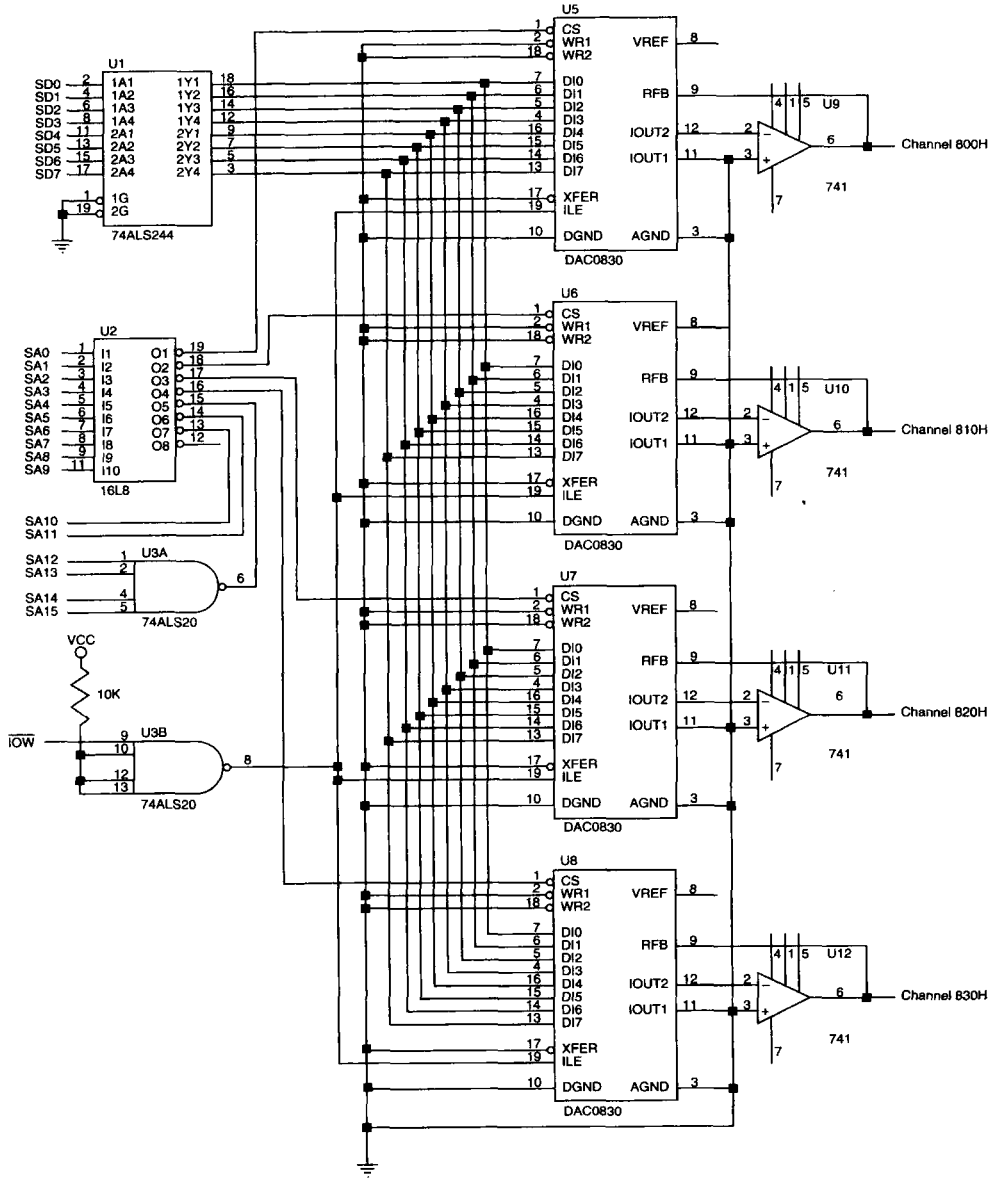


图 D-14

第 16 章

2. 增强的硬件包括：内部时钟、附加中断输入、片选逻辑、串行通信口、并行口管脚、DMA 控制器和一个中断控制器。
4. 10MHz
6. 3.0mA
8. 地址出现的时间点上
10. 16MHz 处理器工作在 10MHz 时存储器访问时间是 260ns
12.
```
MOV  AX,1000H
MOV  DX,0FFFEH
OUT  DX,AL
```
14. 在 80186/80188 大多数型号中中断为 10 个，包括所有内部中断。
16. 每个中断控制寄存器控制一个中断信号。
18. 中断轮询寄存器响应中断，而中断轮询状态寄存器则不响应中断。
20. 3
22. 定时器 2

24. INH 位决定是否影响使能计数位。
26. 两个比较寄存器都选择 ALT 位，这样可以编程逻辑 1 和逻辑 0 的持续时间。
28.
```
MOV  AX,123
MOV  DX,0FF5AH
OUT  DX,AX
MOV  AX,23
ADD  DX,2
OUT  DX,AX
MOV  AX,0C007H
MOV  DX,0FF58H
OUT  DX,AX
```
30. 2
32. 将控制寄存器的 CHG/$\overline{\text{NOCHG}}$位和 START/$\overline{\text{STOP}}$位都置为逻辑 1。
34. 7
36. 片
38. 15
40. 它用于选定$\overline{\text{PCS5}}$和$\overline{\text{PCS6}}$引脚的功能。
42.
```
MOV  AX,1001H
MOV  DX,0FF90H
OUT  DX,AX
MOV  AX,1048H
OUT  DX,AX
```
44. 1G
46. 确认是否可读。
48. RTOS 是实时操作系统，对线程访问具有可预测的保证时间。

第 17 章

2. 64T
4. 见图 D-15。

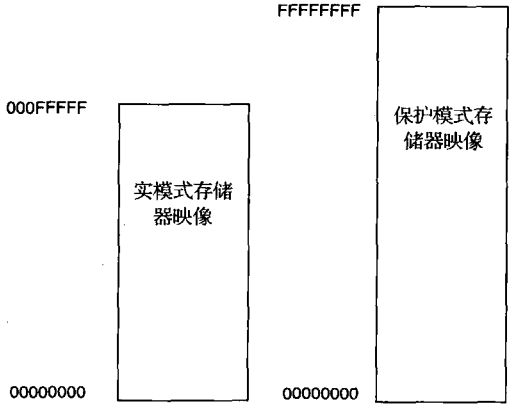


图 D-15

6. 80386 存储系统包含 4GB，体使能信号选择一个或多个 8 位宽的存储体。
8. 流水线允许微处理器在前一个操作数据被取出之前发出下一个地址，这样就容许存储器增加访问数据的时间。
10. 0000H ～ FFFFH
12. 像早期微处理器一样，有同样多的 I/O 地址。区别是通过体使能信号把 I/O 布置成由 4 个 8 位体组成的 32 位宽空间。
14. $\overline{\text{BS16}}$引脚配置微处理器，使其以 16 位的数据总线工作。
16. 前四个调试寄存器（DR_0 ～ DR_3）包含四个断点地址；寄存器 DR_4 和 DR_5 保留给 Intel 使用；DR_6 和 DR_7 用于控制调试。
18. 测试寄存器用于测试转换后备缓冲区。
20. 如果 PE 位置 1，将使微处理器切换到保护模式；如果 PE 位清为 0，将使微处理器切换到实模式。
22. 比例变址寻址使变址寄存器乘以比例因子 1、2、4 或 8，按比例寻址字节、字、双字或四字。
24. (a) 地址在数据段以 EBX 乘 8 加 ECX 为指针指向的单元中。
    (b) 地址在数据段以 EAX 加 EBX 之和为指针指向的数组中。
    (c) 地址在数据段 DATA 单元中。
    (d) 地址在数据段 EBX 指向的单元中。
26. 类型 13（0DH）
28. 中断描述符表和中断描述符。
30. 选择子出现在段寄存器中，它从描述符表中选择一个描述符，还包含要求的请求优先级。
32. 全局描述符表寄存器
34. 因为一个描述符寻址可达 4GB 存储单元，有 8K 个局部描述符和 8K 个全局描述符，所以可寻址 4GB × 16K = 64TB。
36. 任务状态段 TSS 保持任务链和任务寄存器，使任务可以更有效的切换。
38. 当逻辑 1 置在 CR_0 的 PE 位时发生切换。
40. 虚拟模式模拟保护模式下的 DOS，设置最高 1M 存储边界可工作于实模式。
42. 4K
44. 80486 有一个 8K 的内部 cache，还包含一个协处理器。
46. 寄存器组实际上完全一样。
48. $\overline{\text{PCHK}}$和 DP_0 ～ DP_3
50. 8K
52. 猝发是当 4 个 32 位数字在 cache 和存储器之间的读或写。
54. 内建自检测

第 18 章

2. 64GB
4. 这些引脚既产生每字节的奇偶校验第 9 位，又检查奇偶校验。
6. 猝发就绪管脚用于向总线周期插入等待状态。
8. 18.5ns
10. T_2
12. 一个 8KB 的数据 cache 和一个 8KB 的指令 cache。

14. 可以，如果一个是协处理器指令和整数指令不是相互依赖的。
16. 系统存储管理模式在大部分系统中用于电源管理。
18. 38000H
20. CMPXCH8B 指令比较存储于内存的 64 位数和存放在 EDX:EAX 中的 64 位数。如果它们相等则将 ECX:EBX 存入内存；如果它们不相等则将内存内容传送到 EDX:EAX。
22. ID、VIP、VIF 和 AC
24. 为访问 4M 页，放弃页表并只用 22 位偏移地址的页目录。
26. Pentium Pro 是 Pentium 的改进型，包含 3 个整数部件、一个 MMX 部件和一套 36 位地址总线。
28. $A_3 \sim A_{35}$ 36 位地址（$A_0 \sim A_2$ 编码为体选择信号）
30. 66MHz 的 Pentium 的存取时间是 18.5ns；66MHz 的 Pentium Pro 的存取时间是 17ns。
32. 为应用 ECC 存储器替换 64 位宽存储器要购买 72 位宽的 SDRAM。

第 19 章

2. 512KB、1MB 或 2MB
4. Pentium Pro 二级 cache 在主板上，而 Pentium Ⅱ的二级 cache 是在盒式封装里并以更高速度工作。
6. 64GB
8. 242
10. 代替微处理器，读写信号是由芯片组来开发。
12. 存取第一个四字后 8ns，为存取第一个四字还需要 60ns。
14. 已经增加的特定寄存器型号有：SYSENTER_CS、SYSENTER_SS 和 SYSENTER_ESP
16. 当 RDMSR 指令执行时 ECX 寄存器寻址 MSR 寄存器号，执行了 RDMSR 指令后，EDX:EAX 包含寄存器的内容。
18.
```
TESTS  PROC    NEAR
       CPUID
       BT      EDX,800H
       RET
TESTS  ENDP
```
20. EDX 存入 EIP 寄存器，ECX 的数值送到 ESP 寄存器。
22. 环 3
24. Pentium Pro
26. Pentium 4 需要一个 12V 电源，用辅助连接器连到主板上，必须使用 Pentium 4 辅助电源。
28.
```
bool Hyper()
{
     _asm
     {
          bool State = ture;
          mov eax,1
          cpuid
          mov  templ,31h
          bt    edx,28    ;检查超线程
          jc    Hyperl
          mov   State,0
Hyperl:
     }
     return State;
}
```

一本打开的书，
一扇开启的门，
通向科学圣殿的阶梯，
托起一流人才的基石。

华章经典 服务中国教育



计算机硬件及组成原理
作 者：[美] Arnold S Berg
译 者：[illegible]
中文版：[illegible]定价：[illegible]元
英文版：[illegible]定价：[illegible]元



计算机系统结构
作 者：[美] Rob Williams
译 者：[illegible]
中文版：[illegible]定价：[illegible]元
英文版：[illegible]定价：[illegible]元



数字设计和计算机体系结构
作 者：[美] David Harris
译 者：[illegible]
中文版：[illegible]定价：[illegible]元
英文版：[illegible]定价：[illegible]元



计算机组成和汇编语言原理
作 者：[美] Patrick Jones
译 者：[illegible]
中文版：[illegible]定价：[illegible]元
英文版：[illegible]定价：[illegible]元



逻辑和计算机设计基础，第4版
作 者：[美] M Morris Mano
英文版：[illegible]定价：[illegible]元



计算机组成与设计：硬件/软件接口
（英文版·第4版·ARM版）
作 者：[美] David A. Patterson 译者
ISBN：[illegible]
定价：[illegible]



欲了解更多华章计算机图书中国动态，敬请您访问华章IT官方博客：http://blog.csdn.net/hzbooks

延伸阅读



作者：Muhammad Ali Mazidi;Janice Gillispie Mazidi;Rolin D. McKinlay
译者：卢健永 译
ISBN 978-7-111-21524-0
定价：65.00元



作者：[印] Raj Kamal 著
译者：张威 等译
ISBN：978-7-111-27090-0
定价：75.00



作者：Wayne Wolf
译者：李仁发 等
中文版 2/e 2008
ISBN：978-7-111-27066-5 定价：59.00元
英文版 2/e 2008
ISBN：978-7-111-25080-0 定价：75.00元



作者：Wayne Wolf
ISBN：978-7-111-20416-6
定价：65.00元

教师服务登记表

尊敬的老师：

您好！感谢您购买我们出版的 _____ 教材。

机械工业出版社华章公司本着为服务高等教育的出版原则，为进一步加强与高校教师的联系与沟通，更好地为高校教师服务，特制此表，请您填妥后发回给我们，我们将定期向您寄送华章公司最新的图书出版信息。为您的教材、论著或译著的出版提供可能的帮助。欢迎您对我们的教材和服务提出宝贵的意见，感谢您的大力支持与帮助！

个人资料（请用正楷完整填写）

<table>
<tr><td>教师姓名</td><td></td><td>□先生
□女士</td><td>出生年月</td><td></td><td>职务</td><td></td><td colspan="2">职称：□教授 □副教授
□讲师 □助教 □其他</td></tr>
<tr><td>学校</td><td colspan="3"></td><td>学院</td><td colspan="2"></td><td>系别</td><td></td></tr>
<tr><td rowspan="2">联系
电话</td><td colspan="3" rowspan="2">办公：
宅电：
移动：</td><td colspan="2">联系地址
及邮编</td><td colspan="3"></td></tr>
<tr><td colspan="2">E-mail</td><td colspan="3"></td></tr>
<tr><td>学历</td><td></td><td colspan="2">毕业院校</td><td colspan="2"></td><td colspan="2">国外进修及讲学经历</td><td colspan="2"></td></tr>
<tr><td>研究领域</td><td colspan="9"></td></tr>
<tr><td colspan="3">主讲课程</td><td colspan="3">现用教材名</td><td>作者及
出版社</td><td>共同授
课教师</td><td colspan="2">教材满意度</td></tr>
<tr><td colspan="3">课程：
□专 □本 □研
人数： 学期：□春□秋</td><td colspan="3"></td><td></td><td></td><td colspan="2">□满意 □一般
□不满意 □希望更换</td></tr>
<tr><td colspan="3">课程：
□专 □本 □研
人数： 学期：□春□秋</td><td colspan="3"></td><td></td><td></td><td colspan="2">□满意 □一般
□不满意 □希望更换</td></tr>
<tr><td colspan="10">样书申请</td></tr>
<tr><td colspan="4">已出版著作</td><td colspan="3"></td><td colspan="2">已出版译作</td><td></td></tr>
<tr><td colspan="5">是否愿意从事翻译/著作工作 □是 □否</td><td>方向</td><td colspan="4"></td></tr>
<tr><td>意见和建议</td><td colspan="9"></td></tr>
</table>

填妥后请选择以下任何一种方式将此表返回：（如方便请赐名片）

地 址：北京市西城区百万庄南街1号 华章公司营销中心 邮编：100037

电 话：(010) 68353079 88378995 传真：(010)68995260

E-mail:hzedu@hzbook.com markerting@hzbook.com 图书详情可登录http://www.hzbook.com网站查询