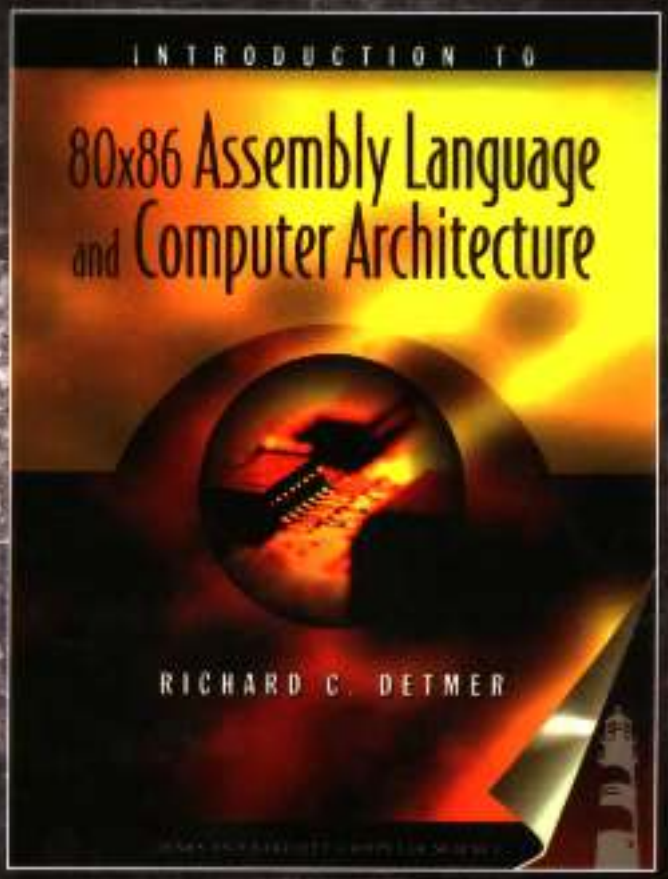
计 算 机 科 学 丛 书

80x86汇编语言与计算机体系结构

(美) Richard C. Detmer 著　郑红　庞毅林　蒋翠玲　译



Introduction to 80x86 Assembly Language and Computer Architecture

机械工业出版社
China Machine Press

本书从计算机的结构层讨论80x86汇编语言与计算机体系结构，并提供了许多汇编语言代码的例子，便于读者在汇编语言层面上学习和掌握计算机体系结构。本书还集中介绍了高级语言中的一些概念以及一些操作系统的功能，并简要描述了在硬件层用到的逻辑门。另外，本书考察了汇编语言如何翻译为机器语言，为读者进一步学习计算机程序设计和体系结构打下基础，有助于用任何编程语言有效地进行编程，激发读者对计算机设计和体系结构进行更进一步的研究，或者更多地了解某个特定计算机系统的详细内容。

本书特点

- 重点介绍了32位平面内存模型
- 强调体系结构，如寄存器、内存编址、硬件功能等
- 增加了高级语言概念
- 初步介绍了汇编语言编程以及Microsoft公司的WinDbg汇编程序
- 实例充分，并有针对性的练习和编程实践

随书光盘内容包括：

Microsoft公司的MASM汇编程序、全屏幕调试器WinDbg和联编器以及完整的源代码和作者自己编写的用于辅助I/O的软件。

作者简介

Richard C. Detmer 于1966年毕业于肯塔基大学，后于威斯康星大学获硕士学位和博士学位，现任中田纳西州大学计算机科学系教授和系主任。

ISBN 7-111-17617-0

9 787111 176176

HZ BOOKS

华章图书

上架指导：计算机科学/嵌入式系统

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

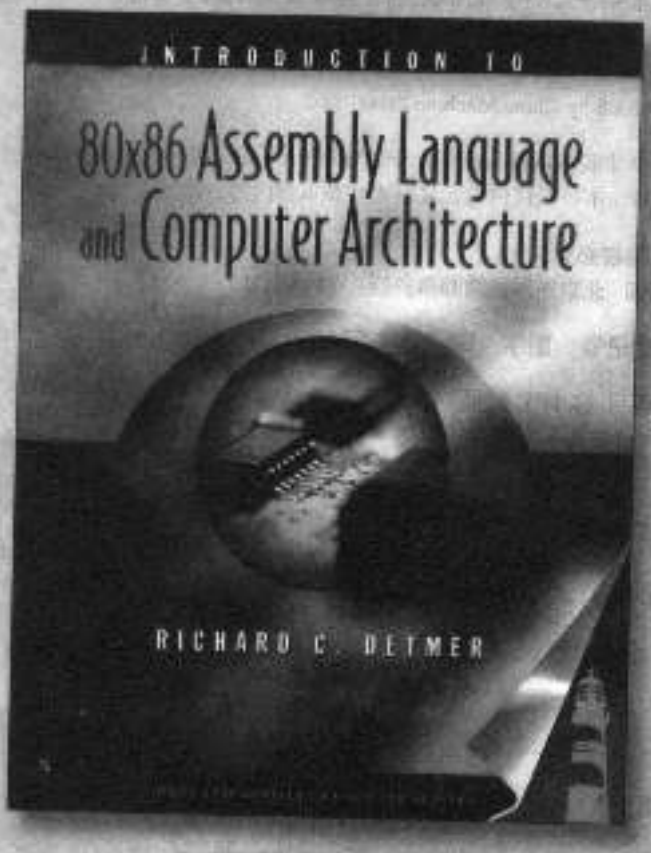
ISBN 7-111-17617-0/TP・4505

定价：49.00 元（附光盘）

计 算 机 科 学 丛 书

80x86汇编语言与计算机体系结构

(美) Richard C. Detmer 著　郑红　庞毅林　蒋翠玲　译



Introduction to 80x86 Assembly Language and Computer Architecture

机械工业出版社
China Machine Press

本书在当前操作系统采用的平面32位地址环境中介绍了80x86汇编语言和计算机体系结构，重点介绍32位平面内存模型，强调了体系结构的概念，如寄存器、内存编址、硬件功能等，涵盖了汇编语言的指令、分支和循环、过程、位运算、汇编过程、输入/输出等重点内容，并增加了高级语言的概念，同时理论结合实例，注重关键知识点练习与编程实践。

本书适合作为高等院校相关专业的教材以及参考书，也可供工程技术人员参考。

Richard C. Detmer: Introduction to 80x86 Assembly Language and Computer Architecture (ISBN 0-7637-1773-8).

Copyright © 2001 by Jones and Bartlett Publishers, Inc.

Original English language edition published by Jones and Bartlett Publishers, Inc., 40 Tall Pine Drive, Sudbury, MA 01776.

All rights reserved. No change may be made in the book including, without limitation, the text, solutions, and the title of the book without first obtaining the written consent of Jones and Bartlett Publishers, Inc. All proposals for such changes must be submitted to Jones and Bartlett Publishers, Inc. in English for his written approval.

Chinese simplified language edition published by China Machine Press.

Copyright © 2005 by China Machine Press.

本书中文简体字版由Jones and Bartlett Publishers, Inc. 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

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：01-2004-5745

图书在版编目（CIP）数据

80x86汇编语言与计算机体系结构 /（美）戴默（Detmer, R. C.）著；郑红等译. －北京：机械工业出版社，2006.1

（计算机科学丛书）

书名原文：Introduction to 80x86 Assembly Language and Computer Architecture

ISBN 7-111-17617-0

Ⅰ.8… Ⅱ.① 戴… ② 郑… Ⅲ.① 汇编语言 ② 计算机体系结构 Ⅳ.①TP313 ② TP303

中国版本图书馆CIP数据核字（2005）第125410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：范运年

北京瑞德印刷有限公司印刷 · 新华书店北京发行所发行

2006年1月第1版第1次印刷

787mm × 1092mm 1/16 · 21.5印张

印数：0 001 - 4 000册

定价：49.00元（附光盘）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

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，还揭橥了学术的源变，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、从业人员较少的现状下，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图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray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、研究及庋藏。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，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步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步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

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，但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。教材的出版只是我们的后续服务的起点。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：

电子邮件：hzjsj@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

译者序

关于这本书的意义和它的主要内容，本书的作者在前言中已经讲得很详细了。我只想简单谈谈在翻译过程中的一些感想。

首先，我很高兴有机会翻译这本书，因为，在过去我们学习计算机课程时，汇编语言课程是使用单独的教材，很少和计算机体系结构结合在一起，学习汇编语言常常令人觉得有些枯燥，并且似乎有些难学易忘；同样，在教授计算机体系结构时，也没有过多地讨论汇编语言程序设计。但这本书很好地将软件设计与硬件结构知识融合在一起，通过一些精选的实例，由浅入深地介绍了汇编语言程序设计的特点以及计算机的工作。因此，通过翻译这本书，不仅让我重温了这两门课程，而且更深层次地理解了计算机的体系结构。

其次，我要感谢机械工业出版社对我的信任，在对书稿的处理过程中，诸位编辑给予了很多帮助，特别是范运年编辑和朱起飞编辑反复征询译者的意见，对本书的译稿提出了许多宝贵的建议。此外，文欣秀老师和朱法枝老师对本书翻译中遇到的个别问题，提出了中肯的意见，在此一并表示感谢。

最后，我要感谢我的家人，他们的支持和鼓励使我能够完成翻译工作。尤其是我的孩子，刚开始翻译时，他尚未出生，他还在孕育中就陪我一起经过了初稿阶段。此后，尽管我常常因为校稿要把他放在一边，减少了对他的照顾，但是，只要我离开电脑向他走去，他总是用最开心、最灿烂的笑容迎接我。

本书的第1章、第4章、第7章由庞毅林翻译，蒋翠玲参与了第9章的翻译，其余章节主要由郑红翻译，全书最后由郑红和庞毅林统稿。由于译者水平所限，加之时间仓促，译文中难免有不妥之处，恳请广大读者不吝批评指正。

译　者

2005年11月

前　言

计算机可以从多种不同的层次来认识。有些人只对字处理或者游戏之类的计算机应用软件感兴趣，但是，计算机程序员通常把计算机作为一个工具，用来编写新的应用软件。通过语言编译器，高级语言程序员更深入地认识了计算机，编译器给人的印象是，计算机的内存地址中存储integer、real和array of char等等对象类型，计算表达式的值，调用过程，执行while循环等等。

然而，事实上计算机是在很低的层次上工作。本书强调计算机的体系结构层，也就是，由机器指令所定义的层次，处理器可以在该层执行。汇编语言指令直接翻译为机器语言指令，这样，当编写一个汇编语言程序时，就可以理解计算机在机器语言级是如何工作的。

尽管本书强调的是计算机操作的汇编语言/机器语言层，但也可从其他层次来认识计算机。本书讨论了高级语言中的一些概念，例如if语句在机器层是如何实现的。本书还讨论了操作系统的一些功能，并简要描述了在硬件层用到的逻辑门。另外，本书考察了汇编语言是如何翻译为机器语言的。

为了在任何层次都可以有效地编程，程序员必须了解在机器层的某些基本原理，它们在大多数的计算机体系结构中都要用到。本书将涉及以下基本概念:

- 存储地址，CPU寄存器及其使用
- 计算机中数值型格式的数据和字符串的表示
- 二进制补码整数的操作指令
- 单个位操作的指令
- 处理字符串的指令
- 分支和循环指令
- 过程编码：控制转移、参数传递、局部变量和调用程序的环境保护

本书中讨论的主要的计算机体系结构是大多数个人计算机所使用的80x86 CPU系列。但是，几乎每章都有其他体系结构，或者不同的计算机层次的信息。用汇编语言编程以及学习本书中的相关概念，有助于用任何编程语言进行有效的编程，激发对计算机设计和体系结构更进一步的研究，或者更多地了解某个特定的计算机系统的详细内容。

本书的组织结构和内容

本书中的大多数素材基于我的前一本书——《Fundamentals of Assembly Language Programming Using the IBM PC and Compatibles》。通过多年对这些素材的教学使我得出这样一个结论：对大多数学生而言，汇编语言课程是介绍计算机体系结构最好的课程。相对于编程而言，本书更多地强调体系结构。本书还重点介绍一些通用的概念，而不是某个特定的计算机系统的细节。

学习这门汇编语言课程要求的前提条件是至少要对高级语言结构有很好的理解。第3章～第6章及第8章是我第一学期课程的核心内容，第1章～第8章的内容我通常讲解得很详细，第9

章速度会快些，根据时间和可利用的资源，选择性讲解第10章～第12章的某些主题。例如，有时，我会通过某个C++程序中的汇编语句行来介绍浮点运算。

风格和教学

本书主要是例证教学。早在第3章本书就给出了一个完整的汇编语言程序，并且在学生能够理解的层次上，仔细地考察了程序的各个部分。随后的章节包含了许多汇编语言代码的例子，同时，对一些新的或者难以理解的概念给出了恰当的解释。

本书使用了大量的图表和例子。给出许多“指令执行前”和“指令执行后”的例子来讲解指令。本书还有一些演示调试程序（debugger）使用的例子。这些例子可以帮助学生深入了解计算机内部的工作。

每章的后面都有练习。答案简短的练习可以加深学生对学过的内容的理解，而且每章后面的编程练习也为学生提供了一个将书中的内容运用到汇编语言编程中的机会。

软件环境

“标准”的80x86汇编器是微软宏汇编器（MASM），版本为6.11。尽管该汇编器生成的代码用于32位的平面内存模式编程，非常适合Windows 95、Windows NT或者32位的微软操作系统环境，但是，与该软件包对应的链接器和调试程序并不适合在这样的系统环境中使用。本书附带一张光盘，包含MASM（ML）的汇编程序、最新的微软链接器、32位的全屏调试程序WinDbg（也来自于微软）以及必要的支持文件。该软件包为生成和调试控制台的应用程序提供了一个良好的环境。

本书配套光盘中不仅有本书的内容，也有可供学生使用的简单的输入/输出设计的软件包。因此，它强调的重点仍然是计算机体系结构而不是操作系统的细节。这个I/O包在本书中广泛使用。最后，该光盘还包含了每个程序的源代码，这些程序都会在书中出现。

致谢

我想感谢我的学生们，他们对本书的最初版本付出了很多努力，让我经常能及时地得到素材。这些学生非常善于捕捉错误。我也要感谢Hong Shi Yuan，在他的汇编语言课程上，他用了本书的最初版本，并提供了有价值的反馈意见。

我还要感谢花了很多时间来检查本书手稿的人们：Houston-Clear Lake大学的Dennis Bouvier、美国空军学院的Barry Fagin、Worcester工艺学院的Glynis Hamel、犹他谷州立大学的Dennis Fairclough、东南路易斯安娜大学的Thomas Higginbotham、Worcester工艺学院的Clifford Nadler。

我的妻子Carol值得称赞。当我在计算机前处理书稿时，经常忽略了她，而她都给予了理解。

Richard C. Detmer

目　录

出版者的话
专家指导委员会
译者序
前言

第1章　计算机中数的表示 ……1
1.1　二进制和十六进制数 ……1
1.2　字符编码 ……4
1.3　有符号整数的二进制补码表示 ……6
1.4　二进制补码数的加减法 ……9
1.5　数的其他表示法 ……13
本章小结 ……15
第2章　计算机系统的组成 ……17
2.1　微机硬件：存储器 ……17
2.2　微机的硬件：CPU ……18
2.3　微机硬件：输入/输出设备 ……22
2.4　PC软件 ……23
本章小结 ……25
第3章　汇编语言的要素 ……26
3.1　汇编语句 ……26
3.2　一个完整的实例 ……28
3.3　程序的汇编、链接和运行 ……33
3.4　汇编器清单文件 ……38
3.5　常数操作数 ……43
3.6　指令中的操作数 ……46
3.7　使用IO.H中宏的输入/输出 ……49
本章小结 ……52
第4章　基本指令 ……54
4.1　复制数据指令 ……54
4.2　整数的加法和减法指令 ……61
4.3　乘法指令 ……69
4.4　除法指令 ……76
4.5　大数的加减 ……84
4.6　其他知识：微代码抽象级 ……86
本章小结 ……87
第5章　分支和循环 ……88
5.1　无条件转移 ……88
5.2　条件转移、比较指令和if结构 ……92
5.3　循环结构的实现 ……103
5.4　汇编语言中的for循环 ……113
5.5　数组 ……118
5.6　其他：流水线 ……123
本章小结 ……124
第6章　过程 ……126
6.1　80x86堆栈 ……126
6.2　过程体、调用和返回 ……131
6.3　参数和局部变量 ……138
6.4　递归 ……145
6.5　其他体系结构：没有堆栈的过程 ……149
本章小结 ……150
第7章　串操作 ……151
7.1　串指令 ……151
7.2　重复前缀和其他串指令 ……156
7.3　字符转换 ……166
7.4　二进制补码整数转换为ASCII码串 ……169
7.5　其他体系结构：CISC和RISC设计 ……172
本章小结 ……173
第8章　位运算 ……174
8.1　逻辑运算 ……174
8.2　移位和循环移位指令 ……181
8.3　ASCII字符串到二进制补码整数的转换 ……190
8.4　硬件级——逻辑门 ……194
本章小结 ……195
第9章　汇编过程 ……197
9.1　两次扫描汇编和一次扫描汇编 ……197
9.2　80x86指令编码 ……200
9.3　宏定义及其展开 ……209

9.4 条件汇编 ……213
9.5 IO.H中的宏 ……218
本章小结 ……221
第10章 浮点数运算 ……222
10.1 80x86浮点数结构 ……222
10.2 浮点型指令编程 ……234
10.3 浮点数的模拟 ……245
10.4 浮点数和嵌入式汇编 ……252
本章小结 ……253
第11章 十进制数运算 ……254
11.1 压缩的BCD码表示 ……254
11.2 压缩的BCD码指令 ……260
11.3 未压缩的BCD码表示和指令 ……266
11.4 其他体系结构：VAX压缩的十进制指令 ……274
本章小结 ……275
第12章 输入/输出 ……276
12.1 使用Kernel32库的控制台输入/输出 ……276
12.2 使用Kernel 32库的连续文件的输入/输出 ……282
12.3 低级输入/输出 ……288
本章小结 ……289
附录A 十六进制/ASCII码的转换 ……291
附录B 常用的MS-DOS命令 ……293
附录C MASM 6.11保留字 ……294
附录D 80x86指令（带助记符） ……298
附录E 80x86指令（带操作码） ……316

第1章　计算机中数的表示

用Java或C++等高级语言编程时，要用到许多不同类型的变量（比如整型、浮点型或者字符型），变量一旦声明，就不需要考虑数据在计算机中是如何表示的。然而，用机器语言编程时，就必须考虑如何存储数据。因此，经常需要将数据从一种表示法转换为另一种表示法。本章将论述微型计算机中数的表示的几种常用方法。第2章概述微机的软件和硬件，第3章介绍如何编写汇编语言程序，由它直接控制计算机机器指令的执行。

1.1　二进制和十六进制数

计算机用位（bit，二进制数制中用不同的电子状态表示0或者1）来表示值。以2为基数，用数字0和1表示二进制数。二进制（binary）数跟十进制数很相似，只不过二进制相应的权（从右到左）依次为1、2、4、8、16（2的更高次幂等），而十进制相应的权为1、10、100、1000、10000（10的幂）。例如，二进制数1101可表示十进制数13。

1		1		0		1		
8	+	4	+	2	+	1	=	13

二进制数很长，因而在读写时很不方便，比如：八位二进制数11111010表示十进制数250，十五位二进制数111010100110000表示十进制数30000。而用十六进制（hexadecimal）（基数16）表示时，只需要用到相应二进制数表示的四分之一长度的位数。十六进制与二进制的转换很容易，因此，十六进制的表示可以缩短二进制的表示。十六进制需要十六个数字：其中0、1、2、3、4、5、6、7、8和9与十进制数相同；A、B、C、D、E、F等同于十进制的10、11、12、13、14和15。另外，这几个字母不论是小写还是大写都可用于表示数。

十六进制数中的权值对应16的幂，权值从右到左依次是1、16、256等等。十六进制数9D7A可如下计算得出表示的是十进制的40314：

```
  9      ×   4096    36864    [4096 = 16³]
+13      ×    256     3328    [D是13, 256 = 16²]
+ 7      ×     16      112
+10      ×      1       10    [A是10]
                    =40314
```

表1-1给出了二进制、十六进制和十进制的关系。记住这张表，或者能够很快地建立这张表是很有必要的。

上面的两个例子显示了二进制数和十六进制数是如何转换为十进制数。那么如何将十进制数转换成二进制数或者十六进制数呢？以及如何将二进制数与十六进制数互相转换呢？随后的内容将介绍如何手工实现不同进制的转换。通常情况下，用一个具有二进制、十进制和十六进制转换功能的计算器很容易实现转换，这样，数制转换只不过是按一两个键而已。这

种计算器可像十进制那样进行二进制和十六进制的数学运算，而且具有很多其他的用途。注意：这种计算器很多都用七个显示段来显示一个数字。比如，显示小写字母b时看起来像数字6，其他的某些字符也有可能很难辨认。

表1-1 十进制、二进制和十六进制数的关系

十进制	十六进制	二进制	十进制	十六进制	二进制
0	0	0	8	8	1000
1	1	1	9	9	1001
2	2	10	10	A	1010
3	3	11	11	B	1011
4	4	100	12	C	1100
5	5	101	13	D	1101
6	6	110	14	E	1110
7	7	111	15	F	1111

不需要用计算器把十六进制数转换为对应的二进制形式。事实上，许多二进制数太长，一般的计算器不易显示。要进行转换，只要将每一个十六进制数用四位二进制数表示即可。其对应关系如表1-1的第3列所示。如果位数不够四位，必要的时候前面用0补充。例如：

$3B8E2_{16} = 11\ 1011\ 1000\ 1110\ 0010_2$

转换数字下标处的16和2表示基数。如果不会造成混淆，这些下标处的数字常可以忽略。二进制数补齐位数是为了增强可读性，如十六进制数2转换为二进制时，最前面用起始位0补齐得到0010。但由于二进制数前面的零不改变该二进制数的值，所以上例中十六进制数的最高位3不需要转换为0011。

把二进制数转换为十六进制数格式，则与上面的步骤正好相反。把二进制数从右向左每四位进行分隔，每四位二进制数用对应的十六进制数表示，例如：

$1011011101001101111_2 = 101\ 1011\ 1010\ 0110\ 1111_2 = 5BA6F_{16}$

前面介绍了如何将二进制数转换为十进制数，但一般不会将很长的二进制数直接转换为十进制数，更快的方法是先将二进制数转换为十六进制数，再将该十六进制数转换为十进制数。以上面的19位二进制数为例：

$$
\begin{aligned}
& 1011011101001101111_2 \\
= \ & 101\ 1011\ 1010\ 0110\ 1111_2 \\
= \ & 5BA6F_{16} \\
= \ & 5 \times 65536 + 11 \times 4096 + 10 \times 256 + 6 \times 16 + 15 \times 1 \\
= \ & 375407_{10}
\end{aligned}
$$

下面将给出十进制数转换为十六进制数的算法，该算法从右到左依次生成十六进制数位。该算法用伪代码来描述，本书中其他的所有算法和程序都将采用伪代码描述。

```
until DecimalNumber = 0 loop
    divide DecimalNumber by 16, getting Quotient and Remainder;
    Remainder (in hex) is the next digit (right to left);
    DecimalNumber := Quotient;
end until;
```

例子

以十进制数5876转换为十六进制数的过程为例：

- 因为这是一个until循环，当第一次执行程序体的时候就进行循环控制条件检查。
- 16整除5876（十进制数）

```
      367   商     新的十进制数的值
16)5876
   5872
      4   余数   最右边的十六进制数位
          当前结果：4
```

- 367不等于0，再用16整除

```
       22   商     新的十进制数的值
16)367
   352
    15   余数   生成的第2个十六进制数位
         当前结果：F4
```

- 22不等于0，用16整除

```
        1   商     新的十进制数的值
16)22
   16
    6   余数   生成的下一个十六进制数位
        当前结果：6F4
```

- 1不等于0，用16整除

```
        0   商     新的十进制数的值
16)1
   0
   1   余数   生成的下一个十六进制数
       当前结果：16F4
```

- 当前的十进制数为0，循环终止，最后结果为$16F4_{16}$。

在计算机中也用到八进制数（octal，基数为8）。八进制数用数字0－7表示，大多数计算器可进行十六进制和八进制运算。把二进制数的每3位转换为一个对应的八进制数，很容易实现二进制数八进制数的转换。同样，将八进制数转换为二进制数时，每一个八进制数用相应的3个二进制数位表示。要实现十进制转换为八进制，可以使用前面的十进制转换为十六进制的算法，只不过在做每一步时，用8而不是用16整除。

练习1.1

请根据给出的每一个数字将表中空白的另外两种进制形式补充完整。

	二进制	十六进制	十进制
1.	100	_____	_____
2.	10101101	_____	_____
3.	1101110101	_____	_____
4.	11111011110	_____	_____
5.	10000000001	_____	_____
6.	_____	8EF	_____
7.	_____	10	_____
8.	_____	A52E	_____
9	_____	70C	_____
10.	_____	6BD3	_____
11.	_____	_____	100
12.	_____	_____	527
13.	_____	_____	4128
14.	_____	_____	11947
15.	_____	_____	59020

1.2 字符编码

字母、数字、标点符号等各种字符在计算机中都是用特定的数值来表示的。字符编码方式有很多，微机中普遍采用的一种字符编码是美国信息交换标准代码（简称为ASCII，其发音为ASK-ee）。

ASCII用七位表示字符，数值从0000000～1111111，包括128个值，可以表示128种字符。也可用十六进制数00～7F或者十进制数0～127[⊖]表示。附录A给出了ASCII的详细列表，在表中可以查到“Computers are fun .”用十六进制表示的ASCII码值；

43	6F	6D	70	75	74	65	72	73	20	61	72	65	20	66	75	6E	2E
C	o	m	p	u	t	e	r	s		a	r	e		f	u	n	.

注意：尽管空格字符不可见，但仍然有一个字符编码（十六进制数20）

数字也可以用字符编码表示，例如用ASCII表示日期“October 21, 1976”：

4F	63	74	6F	62	65	72	20	32	31	2C	20	31	39	37	36
O	c	t	o	b	e	r		2	1	,		1	9	7	6

其中，日期中的数字字符21用ASCII码值32 31表示，1976用31 39 37 36表示，这与上节所讲的二进制表示有所不同，上节中$21_{10} = 10101_2$，$1976_{10} = 11110111000_2$。这两种方法在计算机中都可以表示数字：其中ASCII表示法用于外设输入输出，二进制表示法用于计算机内部计算。

ASCII码看起来似乎是任意指定的，但事实上是遵循某些规范的。大写字母的ASCII码是

⊖ 包括IBM及兼容系统在内的一些计算机，使用扩展的字符集，字符集增加了从十六进制数80～FF（十进制数128～255）的字符，本书中不使用扩展的字符集。

相邻的，同小写字母的ASCII码一样。大写字母的编码与其相对应的小写字母的编码仅仅有一位不同，大写字母的第5位是1，而小写字母的第5位是0，其他各位都相同。（通常计算机用位来表示数时，从右到左，最右边的位从第0位开始。）例如，

- 大写字母M的编码为$4D_{16} = 1001101_2$
- 小写字母m的编码为$6D_{16} = 1101101_2$

打印输出字符（printable character）从20_{16} ~ $7E_{16}$。（空格字符也是打印输出字符。）数字0、1、…、9的ASCII值分别为30_{16}、31_{16}、…、39_{16}。

ASCII码值从00_{16} ~ $1F_{16}$以及$7F_{16}$都是控制字符（control character），例如，ASCII键盘上的ESC键的ASCII码值是$1B_{16}$，简称ESC，表示特殊服务控制，但经常被认为是“escape”的含义。ESC字符经常与其他字符一起传给外部设备，比如，传给一台打印机，让它执行某种指定的操作。因为这样的字符序列没有标准化，所以本书将不作讨论。

本书中使用频率最高的两个ASCII控制字符是$0D_{16}$和$0A_{16}$，分别表示回车（CR）和换行（LF）。当按下ASCII键盘的Return或Enter键时，就会产生编码$0D_{16}$，如果该编码送到ASCII显示器，则使光标移到当前行的开始处而不是到新的一行；如果该编码送到ASCII打印机（至少是早期的一种打印机），则会使打印头移到当前行的开始处。换行码$0A_{16}$在ASCII显示器上会使光标垂直移到下一行或者使打印机将纸向上滚动一行。要想信息从新的一行的开始处显示，需要同时把CR和LF字符传给显示器或者打印机。但是，如果用汇编语言编程来实现这种操作就很麻烦。有时，在命令提示符下输入后，使得光标移开当前行，或者使用几条输出指令将所有输出都显示在一行，这时也可以不选用CR和（或者）LF。

使用较少的控制字符有“form feed”（$0C_{16}$），该字符使打印机退出某页；控制字符“horizontal tab”（09_{16}）在按下键盘的Tab键时生成；“backspace”（08_{16}）在按下键盘的Backspace键时生成；“delete”（$7F_{16}$）在按下键盘的Delete键时生成。注意：Delete键和Backspace键生成的代码不同。响铃（bell）字符（07_{16}）输出到显示器时会听见响铃声，有丰富编程经验的人员在真正需要响铃的时候才使用响铃字符。

许多大型计算机用“扩展二进制编码-十进制信息编码”（Extended Binary Coded Decimal Information Code，简称为EBCDIC，其发音为ib-SEE-dick或者eb-SEE-dick）。本书仅在讨论两种不同编码系统转换时会用EBCDIC编码作为例子。

练习1.2

1. 以下每个十六进制数可表示为一个十进制数或两个字符的ASCII值，请写出这两种表示。
   (a) 2A45　(b) 7352　(c) 2036　(d) 106E
2. 写出下列字符串的ASCII值，不要忘记空格和标点符号。回车和换行字符用CR和LF表示，如果写在一起CRLF（中间没有空格）表示回车换行功能。
   (a) January 1 is New Year's Day. CRLF
   (b) George said, “Ouch!”
   (c) R2D2 was C3P0's friend. CRLF[“0” is the numeral zero]
   (d) Your name? [put two spaces after the question mark]
   (e) Enter value:[put two spaces after the colon]
3. 将下列ASCII序列输出到计算机显示器，会显示什么？
   (a) 62 6C 6F 6F 64 2C 20 73 77 65 61 74 20 61 6E 64 20 74 65 61 72 73

(b) 6E 61 6D 65 0D 0A 61 64 64 72 65 73 73 0D 0A 63 69 74 79 0D 0A

(c) 4A 75 6E 65 20 31 31 2C 20 31 39 34 37 0D 0A

(d) 24 33 38 39 2E 34 35

(e) 49 44 23 3A 20 20 31 32 33 2D 34 35 2D 36 37 38 39

1.3 有符号整数的二进制补码表示

本节详细探讨计算机中数的表示。前面已经介绍了两种表示数的方法，一种是用二进制表示（通常用十六进制表示），另一种是用ASCII码表示，但是这两种表示法有两个问题：（1）表示数的有效位是有限的；（2）如何表示负数并不明确。

第2章将讨论计算机硬件，现在应该知道计算机内存是以字节（Byte）为单位存储的，每一个字节包含8位（bit）[⊖]。假定内存中的数用ASCII码存储，一个ASCII码通常用一个字节存储。而ASCII码长度是7位，附加位（左侧或最高位）置0。为了解决上述表示法中的第二个问题，可在编码中包含一个负数符号。例如：用四个字符的ASCII编码来表示－817，就是2D，38，31和37。为解决第一个问题，通常用固定长度的若干字节数，位数不足时，在ASCII码的左边用0或者空格补充；也可以使用一个长度可变的字节数，但必须规定要表示的数的最后一个数位以ASCII码结束，也就是说用一个非数字字符结束。

计算机内部使用二进制表示数时，需选用某种固定长度位的表示方法。大多数中央处理单元（CPU）能进行变长的二进制数的运算。如Intel 80x86系列，可变长度有8位（1字节）、16位（1个字）[⊜]、32位（双字）和64位（四字）。

以697的字长二进制表示为例：

$697_{10} = 1010111001_2 = 0000001010111001_2$

二进制数表示的前面添加0是为了保证长度是16位，如果将该二进制数用十六进制数简化表示，即：

02	B9

这种表示法将贯穿全书，方框代表字节序列，每个字节的内容用十六进制表示，由于每个十六进制数用四位二进制数表示，因此，每个字节用两个十六进制数表示。如果用双字表示697，则前面需要用0补齐，即：

00	00	02	B9

前面介绍的数的表示法能够很好的表示非负数和无符号（unsigned）数，但不能表示负数。同时，任意给定长度都可表示一个最大无符号数，如字节长度，表示的最大无符号数为FF_{16}或者255_{16}。

二进制补码（2's complement）表示法与前面所讲的无符号数表示法很相似，但前者可以表示负数。二进制补码表示数时，应该明确其长度，所以可用“字长的二进制补码表示法”

⊖ 早期的计算机系统使用字节大小而不是8位。

⊜ 有些其他的计算机体系使用字大小而不是16位。

表示一个数。二进制补码表示非负数与表示无符号数大致相同；也就是说二进制补码表示数时，前面需要补充很多0来确保定长。对于正数的表示有一个附加位，最左边的一位置0。如：用字长二进制补码表示最大的正数就是0111111111111111_2、$7FFF_{16}$或者32767_{10}。

如果根据正数的表示是最左边一位置0，就推测负数的表示是将最左边一位置1，其他各位跟对应正数的表示完全相同，那就大错特错了。因为负数的表示要比正数的表示复杂的多，所以不能简单地将最左边的位由0改为1来表示负数。

有十六进制功能的计算器很容易将一个负的十进制数转换为二进制补码表示。例如，计算器显示－565，如果按“十六进制”转换键，计算器通常会显示FFFFFFFDCB（有可能最前面F的个数不同）。忽略其他数，仅仅保留最后四个十六进制数，则可表示为：

FD	CB

或者二进制1111 1101 1100 1011。（注意：最高位用于表示负数。）如果用双字长表示，则为：

FF	FF	FD	CB

这种表示太长，因此，不方便写成二进制形式。

不用计算器也可实现负数的二进制补码表示。一种方法就是先用十六进制表示这个无符号数，然后用10000_{16}减去这个十六进制数得到该数的字长二进制补码表示，十六进制的被减数由1后面紧跟表示长度所决定的若干个0组成。例如，100000000_{16}就是双字长表示补码中的被减数。（字节长度的二进制补码表示的被减数是多少呢？四字长的二进制补码的被减数又是多少呢？）以二进制表示，被减数0的个数是二进制数位的长度，二进制数是2的幂，所以二进制数做减法有时称为“取补”，因而这也是称为“二进制补码”的原因所在。

长度为字的二进制补码表示十进制数－76：首先转换无符号数76即十六进制数4C，然后用10000减去4C。

```
1 0 0 0 0
    - 4 C
```

因为0不够减C，所以要从1000中借1，剩下FFF，即：

```
F F F 10
  - 4  C
---------
F F B  4
```

借1之后做减法就容易多了，相应的数字变为：

$10_{16} - C_{16} = 16_{10} - 12_{10} = 4$（十进制或十六进制）

并且$F_{16} - 4 = 15_{10} - 4_{10} = 11_{10} = B_{16}$

如果已经了解十六进制数的加减法，则不用将十六进制数转换为十进制数后再做减法。

1后面紧跟适当个数的0作为被减数，减去某个数的操作称为二进制取补（taking the 2's complement）。这个称谓既包含二进制表示的含义，又包含取补操作的含义，二进制取补操作相当于在十六进制计算器上按下符号转换键。

既然一个定长的二进制补码表示是固定长度，那么，显然可以存储一个最大数。对于给定的长度是一个字，则可存储一个最大的正数7FFF，它是最大的16位长的正数，用二进制表示时最高位为0。十六进制数7FFF也就是十进制数32767。正数用十六进制表示的最高位是0到7（即用二进制表示最高位为0），负数用十六进制表示的最高位为8到F，即用二进制表示的最高位是1。

怎样把一个二进制补码表示成对应的十进制数呢？首先，确定二进制补码的符号。将正的二进制补码数转换为十进制数，就像任何无符号二进制数转换成十进制数一样，可以手工也可用有十六进制功能的计算器转换。例如，字长的二进制补码数0D43表示十进制数3395。

二进制数位的最高位是1，即十六进制的8到F表示的二进制补码数，处理这类负数有些复杂。值得注意的是，任何时候对一个数求二进制补码，得到结果，再对该结果求其二进制补码，就得到原数。对于长度为字的数N，通常的代数表达式为:

```
N = 10000 - (10000 - N)
```

例如: 字长的二进制补码数F39E

```
10000 - (10000 - F39E) = 10000 - C62 = F39E
```

这再次说明二进制补码运算与取补运算是一致的。因此，用最高位表示的负数求其二进制补码可得到对应的正数（无符号数）。

字长的二进制补码数E973表示一个负数，因为符号位（最高位）是1（E = 1110）。取补码可得出相应的正数。

$10000 - E973 = 168D = 5773_{10}$

该表达式意味着E973对应的十进制数是－5773。

最高位为1的字长二进制补码的范围从8000～FFFF。转换为十进制数为:

$10000 - 8000 = 8000 = 32768_{10}$

因此，8000表示的是－32768。同样，

```
10000 - FFFF = 1
```

因此，FFFF表示的是－1。由前面得知，字长的二进制补码表示的最大数为十进制正数32767，因此，其表示的十进制数范围是－32768～32767。

用计算器实现将二进制补码表示的负数转换为十进制数是件很棘手的事情。比如，以字长表示FF30，用计算器显示的是10个十六进制数，因此，该数的十个十六进制数的序列为FFFFFFFF30，前面附加了6个F，然后按计算器的“十进制”转换按钮，则显示的为－208。

练习1.3

1. 给出下列每个十进制数的字长的二进制补码:
   (a) 845　(b) 15000　(c) 100　(d) −10　(e) −923
2. 给出下列每个十进制数的双字长的二进制补码:
   (a) 3874　(b) 1000000　(c) −100　(d) −55555
3. 给出下列每个十进制数的字节长的二进制补码:
   (a) 23　(b) 111　(c) −100　(d) −55
4. 给出下列每个字长的二进制补码数所表示的十进制整数:
   (a) 00 A3　(b) FF FE　(c) 6F 20　(d) B6 4A
5. 给出下列每个双字长的二进制补码数所表示的十进制整数:
   (a) 00 00 F3 E1　(b) FF FF FE 03　(c) 98 C2 41 7D
6. 给出下列每个字节长的二进制补码数所表示的十进制整数:
   (a) E1　(b) 7C　(c) FF
7. 给出字节长的二进制补码形式存储的十进制整数范围。
8. 给出双字长的二进制补码形式存储的十进制整数范围。
9. 本节介绍了如何用某个适当的2的幂（如长度为字节则为2的8次幂）的数作被减数来求一个数的二进制补码。另一种方法就是以二进制表示该数（选用恰当的数位作为表示长度），将每位的0变为1，1变为0（也称对1取反），最后，对结果加1（忽略进位）。这两种方法有异曲同工之妙。

1.4 二进制补码数的加减法

计算机内部普遍采用二进制补码表示法存储有符号整数的原因之一，在于这使得计算机硬件实现加减运算非常容易和高效。本小节将讨论二进制补码数的加减运算，并介绍后面内容所用到的进位和溢出。

两个二进制补码数的相加似乎像无符号二进制数的相加那样简单。80x86结构系统对有符号数和无符号数使用相同的加法指令。以后的示例中的数都以字长度表示。

首先做0A07和01D3相加。不管采用的是无符号数还是二进制补码表示，这两个数都是正数。右边表示十进制相加的算式。

```
  0A07        2567
+ 01D3       + 467
  0BDA        3034
```

由$BDA_{16} = 3034_{10}$可得，该计算结果是正确的。

然后，做0206与FFB0相加。很显然，正数如同无符号数，但需用二进制补码的有符号数表示，0206是正数而FFB0为负数，这样就有两种十进制数相加算式的情况，第一种是有符号数相加的算式，第二种是无符号数相加的算式。

```
  0206          518          518
+ FFB0        + (-80)      + 65456
 101B6          438         65974
```

根据上述计算，有符号数的相加与十进制数的相加结果不相等。事实上，101B6是65974的十六进制数表示，但其不能用长度为字的无符号数表示（长度为字的无符号数能表示的最大正数为65535）；如果用有符号数表示并且忽略最左边的附加位1，由101B6可得到01B6，01B6正是十进制数438的二进制补码。

FFE7与FFF6相加。如果用有符号数表示，则这两个数是负数。下面也给出有符号十进制数和无符号十进制数相加的表示。

```
  FFE7          (-25)        65511
+ FFF6        + (-10)      + 65526
 1FFDD           -35        131037
```

相加的和由于太大，仍然不能用两个字节数表示，但是如果不考虑附加位1，则FFDD就是－35的字长的二进制补码表示。

上面两个例子中，后两个加法运算都有一个向高位的进位，除去进位后，其他的数字位恰恰是所求和的正确二进制补码表示，但不是总能得到和的正确二进制补码表示。再看下面两个正数的相加:

```
  483F        18495
+ 645A      + 25690
  AC99        44185
```

这两个相加算式中没有向高位的进位，但有符号数的表示却是错误的，因为AC99表示负数－21351。直观上看，错误的原因在于，求得的和44185比用一个字即两个字节长度存储的最大有符号整数32767大。但是，无符号数求和得到的结果却是正确的。

下面的两个有符号数表示的负数相加会得到明显错误的结果。

```
  E9FF          (-5633)        59903
+ 8CF0        + (-29456)     + 36080
 176EF           -35089        95983
```

两个数相加有进位，但进位外的其他数字76EF却不是和的正确有符号数表示，因为76EF表示的是正数30447，而且，由－32768是以字长度存储的最大负数也可直观判断运算结果出错了。

上面例子出错的原因在于产生了溢出（overflow）。在做加法运算时，由得到错误的有符号数的运算结果，可人为判断产生了溢出，计算机在做二进制加法时，硬件也可判断是否溢出，而且，如果没有溢出，计算机则认为得到的结果是正确的。事实上，计算机做二进制加法时，其运算过程逻辑上从右向左进行各位相加运算，与手动做十进制加法的过程很相似。计算机在逐位相加时，有时会向临近的左边一列产生进位“1”，这个进位会与左边的一列相加的结果一起求和，其他各列相加时依此类推。最特殊的一列是最左边的一列即符号位，可能有进位到该位，也有可能该符号位的进位到“附加位”。符号位进位输出（输出到“附加位”）即前面所谈到的“进位”，以附加的十六进制数1表示。表1-2给出了溢出与非溢出产生的各种情况，从这个表可以总结出：向“附加位”的进位与向符号位的进位同时有或者同时没有时，不会发生溢出，否则发生溢出。

表1-2 加法运算的溢出情况

是否有向符号位的进位?	符号位是否有进位输出?	是否溢出?
无	无	无
无	有	有
有	无	有
有	有	无

上述的加法例子现在用如下二进制数再运算一遍，所有的进位都写在两数的上方。

```
                  111
  0000 1010 0000 0111          0A07
+ 0000 0001 1101 0011        + 01D3
  0000 1011 1101 1010          0BDA
```

该例没有向符号位的进位，也没有符号位进位输出，所以没有溢出。

```
1 1111 11
  0000 0010 0000 0110          0206
+ 1111 1111 1011 0000        + FFB0
1 0000 0001 1011 0110         101B6
```

该例有向符号位的进位，并且有符号位进位输出，所以没有溢出。

```
1 1111 1111 11   11
  1111 1111 1110 0111          FFE7
+ 1111 1111 1111 0110        + FFF6
1 1111 1111 1101 1101         1FFDD
```

同样，该例既有向符号位的进位也有有符号位的进位输出，所以也没有溢出。

```
  1          1111 11
  0100 1000 0011 1111          483F
+ 0110 0100 0101 1010        + 645A
  1010 1100 1001 1001          AC99
```

该例中，有向符号位的进位但符号位没有进位输出，所以发生了溢出。

```
1    1   11 111
  1110 1001 1111 1111          E9FF
+ 1000 1100 1111 0000        + 8CF0
1 0111 0110 1110 1111         176EF
```

该例子因为没有向符号位的进位，但有符号位的进位输出，所以产生了溢出。

在计算机中，a－b这样的减法算式，通常是取b的二进制补码，然后将其与a相加，这就相当于a与－b相加。例如：十进制减法195－618＝－423，

```
  00C3
- 026A
```

可将－026A转换为加上FD96，因为FD96是026A的二进制补码。

```
  00C3
+ FD96
  FE59
```

十六进制有符号数FE59表示十进制数－423。观察上面的加法算式，有

```
          11      11
  0000 0000 1100 0011
+ 1111 1101 1001 0110
  1111 1110 0101 1001
```

要注意，这个加法算式中没有进位，但是做减法时要借位（borrow）。做减法a－b时，如果无符号数b比a大则要借位。计算机硬件通过将减法运算转换为相应的加法运算，根据加法是否产生进位来判断减法是否要借位，如果加法运算没有进位，则对应的减法运算就需要借位，如果加法运算有进位，则对应的减法运算不需要借位。

以另外一个减法运算为例：十进制数985－411＝574，用字长的二进制补码表示为

```
  03D9
- 019B
```

可将减019B转换为加上019B的二进制补码FE65。即：

```
                1 1111 1111 1      1
  03D9            0000 0011 1101 1001
+ FE65          + 1111 1110 0110 0101
 1023E          1 0000 0010 0011 1110
```

不考虑附加位1，十六进制数023E表示十进制数574。这个例子在做加法时有进位，所以相应的减法运算不需要借位。

减法运算也需要定义溢出。如果知道该运算结果已经超出了某种长度（如字长等）表示法所表示的范围，则可以人为断定这个运算结果出错了。而计算机通过对所做的加法运算出现的情况来判断相应的减法运算是否产生了溢出。如果加法运算有溢出，则初始的减法运算也会有溢出；如果加法运算没有溢出，则初始的减法运算也不会有溢出。前面所列举的两个减法运算都没有溢出。如果用字长的二进制补码表示－29123－15447，就会产生溢出。显然，正确的答案－44570已经超出了字长的二进制补码数所表示的范围－32768～32767，而计算机硬件在做如下运算时，

```
  8E3D
- 3C57
```

将减3C57计算转换为加上3C57的二进制补码C3A9。

```
                1    1   11  111   1
  8E3D            1000 1110 0011 1101
+ C3A9          + 1100 0011 1010 1001
 151E6          1 0101 0001 1110 0110
```

由于符号位有进位输出，但没有向符号位的进位，所以产生了溢出。

本小节是以字长的二进制补码来描述数的加减法运算，这种方法也可运用到字节的二进

制补码、双字的二进制补码或者其他长度的二进制补码的加减运算中。

练习1.4

完成下列字长的二进制补码数的操作。给出每个加减运算操作具体的和与差，并判断是否有溢出。对于求和运算，判断是否有进位，对于求差运算，判断是否有借位。将运算的结果转换成十进制数来核对所做的答案是否正确。

1) 003F + 02A4	2) 1B48+ 39E1
3) 6C34 + 5028	4) 7FFE + 0002
5) FF07 + 06BD	6) 2A44 + D9CC
7) FFE3 + FC70	8) FE00 + FD2D
9) FFF1 + 8005	10) 8AD0 + EC78
11) 9E58 - EBBC	12) EBBC - 9E58
13) EBBC - 791C	14) 791C - EBBC

1.5 数的其他表示法

1.2节与1.3节介绍了计算机中数的通常表示方法，如字符编码（通常是ASCII）和二进制补码。本节将介绍另外三种表示法：1位补码表示法、二—十进制编码（BCD）和浮点表示法。1位补码表示法可以作为有符号数的替代表示法，用于Intel 80x86系列以外的其他某些计算机系统中。二—十进制编码和浮点表示法也用于Intel 80x86系列及其他系统中。在以后将详细探讨这些表示法，他们用来描述指令处理的数据。引入这些表示法最主要的原因是对于某些特定的系统，他们是可选的、有效的数的表示方法。

1位补码（1's complement）表示法与二进制补码表示法很相似。在用固定长度表示数时，正数是最简单的二进制数的表示形式，在左边补齐若干个0来确保其长度。要得到一个负数，是对其对应正数的二进制表示的每位取反，即每位的0变为1，1变为0，有时，这种操作也可用于一个数的1位补码表示法。尽管1位补码表示法表示负数比二进制补码表示负数要容易的多，但它有很多不足。最主要的原因是很难设计电路来做加减法运算。因为这种表示法对于0有两种表示（为什么？），这是非常糟糕的。同时，1位补码表示法表示的数的范围比二进制补码表示法的要稍小些。比如，二进制补码表示的数的范围是 - 128 ~ 127，而1位补码表示法表示的数的范围是 - 127 ~ 127。

字节长的1位补码表示十进制数97即0110 0001（十六进制的61），按照每位是0就变成1，是1就变成0的原则，则转换后的结果是1001 1110（十六进制的9E），得到其1位补码表示的结果是十进制的 - 97。

数的1位补码表示与二进制补码表示有非常紧密的联系。对1位补码表示的数加1则得到该数的二进制补码表示。相比前面1.3节所提到的减法运算操作，这种方法用手工实现要容易得多。

二—十进制编码（binary coded decimal, BCD）对每个十进制数用固定长度的一长串位来表示。这些一连串的位组合在一起，就是该数的二—十进制编码表示。使用最多的是每个十进制数用四位二进制数表示。BCD与十进制数的十种对应关系如表1-3所示。

表1-3 二—十进制编码表示

十 进 制	BCD位模式	十 进 制	BCD位模式
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

十进制数926708的用BCD表示就是1001 0010 0110 0111 0000 1000。把每4位二进制数转换为相应的1位十六进制数，再将每两位十六进制数组合成一个字节，则该数的BCD码表示由三个字节组成，即:

92	67	08

注意: BCD表示的数从数字上看起来好像是十进制数。

用BCD编码时，通常使用固定长度的字节数。作为示例，一个BCD码用四字节表示，现在的问题是如何表示被忽略掉的符号。如果不为符号位预留存储空间，那么八位BCD数可用四字节存储。这样，十进制数3691的BCD就可表示为:

00	00	36	91

注意: 如果该数用双字长的二进制补码表示，则为00 00 0E 6B，那么用ASCII码来表示这四个数则应该是33 36 39 31。

计算机进行数的算术运算时，用BCD码表示不如二进制补码高效。如果用ASCII码表示数做算术运算，也是效率极低的。但是迄今为止，用ASCII编码却是表示非整数的惟一方法。例如，78.375用ASCII表示，存储为37 38 2E 33 37 35。浮点表示法可以非常接近地表示非整数。

浮点（floating point）表示法存储数与科学记数法非常接近。下面将举例说明如何将十进制数78. 375转换成32位长的IEEE*单精度格式*（IEEE single format）（Institute of Electrical and Electronics Engineers，IEEE）。浮点表示法由IEEE计算机协会标准委员会提出，得到IEEE标准委员会和美国国家标准协会（ANSI）认可的格式之一，它是Intel 80x86处理器用到的一种浮点格式之一。

首先，78.375必须转换成二进制。在二进制中，二进制“.”右边的位(二进制数的“.”不能确切的说是十进制的“.”)分别对应2的负幂（1/2，1/4，1/8等等），正如十进制中对应的是10的负幂（1/10，1/100等等）。由$0.375 = 3/8 = 1/4 + 1/8 = 0.01_2 + 0.001_2$，$0.375_{10} = 0.011_2$，则78转换为二进制的1001110，因此，

$$78.375_{10} = 1001110.011_2$$

然后，用小数点前面是1的数作为尾数，二进制科学记数法表示为:

$$1001110.011_2 = 1.001110011 \times 2^6$$

二进制的科学记数法的指数与十进制记数一样精确，将二进制的“.”从右向左移动产生尾数来记数，尾数要满足小数点前面是1的条件，同时得到指数6，写成2^6比写成10^{110}更恰当些，

但用十进制表示更方便。最后把它们合在一起就是该数的浮点表示。

- 最左边位为0表示正数（1表示为负数）
- 1000 0101是指数。由原来指数6加上偏值127，求和得到133，133用8位二进制表示得到新指数的二进制数表示1000 0101。
- 001110011 0000 0000 0000 00，去掉最高位1并且右边用0补齐，确保是23位二进制数位而得到。
- 最后得到的完整结果为0 10000101 001110011 0000 0000 0000 00。对其分组后得到0100 0010 1001 1100 1100 0000 0000 0000，用十六进制表示为

42	9C	C0	00

这个结果很容易计算处理，因为，78.375的小数部分0.375可由2的负幂求和得到。但并不是所有的数都如此，通常是用一个二进制小数来近似表示一个十进制数的小数部分，本书不涉及这样的近似表示方法。

总结如下，根据下面的步骤，可以将十进制数转换为IEEE单精度格式。

1. 浮点数的最高位如果为0则表示正数，1表示负数。
2. 用二进制来表示这个无符号数。
3. 用二进制科学记数法表示这个二进制数，如$f_{23} \cdot f_{22} \cdots f_0 \times 2^e$，其中，$f_{23} = 1$。如果小数位已有24位，则其后不用0补充。
4. 将指数e加上偏值127_{10}，得到的和用二进制表示，符号位后的8位就是所要的结果（加偏值127_{10}是将指数转换成有符号数）。
5. 将最高位f_{23}（通常是1）去掉，得到小数位$f_{22}f_{21} \cdots f_0$。

通常，计算机的浮点运算比二进制补码运算要慢，但其优点在于，它能够表示非整数，或者表示二进制补码表示范围以外的较大的或者较小的数。

练习1.5

用字长的1位补码表示下列每个十进制数。

1. 175　　2. －175

3. －43　　4. 43

用BCD码表示下列每个十进制数，要求长度为4字节，并将表示的结果用十六进制表示，按一个字节长度分成两组。

5. 230　　6. 1

7. 12348765　　8. 17195

写出下列十进制数的IEEE单精度格式的浮点表示。

9. 175.5　　10. －1.25

11. －11.75　　12. 45.5

本章小结

计算机用电子信号表示所有数据，这些数据可用二进制数来表示，这种表示法就是数的

二进制表示，也可写成十进制、十六进制和二进制格式。

大多数计算机用ASCII码表示字符，每个字符包括不能打印的控制字符都有一个ASCII码值。

可用预定长度的二进制补码表示整数，正数用二进制数存储（左边用0补充以满足预定长度）；负数可用1后面紧跟若干0（0的个数与预定的长度有关）做被减数，然后减去其对应的正数来表示。负数的二进制补码表示的最高位是1，有十六进制功能的计算器可以方便地进行二进制补码运算。

二进制补码数很容易做加减运算，因为二进制补码的位长度是固定的，在做运算时可能产生进位、借位或溢出。

计算机中数的其他表示法有1位补码表示法、BCD码和浮点法。

第2章　计算机系统的组成

一个实际的计算机操作系统由硬件和软件组成。典型的微机系统的主要硬件部件包括：一个中央处理单元（CPU）、存储单元、输入键盘、监视器或者某些其他的显示设备，专门的输入/输出设备如鼠标、调制解调器或者声卡等，以及一个或多个保存程序和数据的磁盘设备。软件是指硬件执行的程序，包括系统软件和应用软件。

不同的计算机系统有不同的基本的部件。本章讨论在特定的微机如IBM PC以及其他兼容系统的环境下，汇编语言程序员是如何使用存储器和CPU的。这些计算机系统有一个Intel 80x86 CPU，即一个8086或8088、80286、80386、80486或者一个Pentium处理器[⊖]。本书讨论的计算机系统假定具有80386或者更高的处理器，并提供了Windows 95或者Windows NT之类的32位的操作系统。本书的其他部分着重探讨如何用汇编语言在这些系统上编写程序，以及这些系统是如何在硬件级上工作。

2.1　微机硬件：存储器

IBM PC机或兼容微机存储器逻辑上可以看作是一个“条板单元”的集合，每个单元能存储一个字节的指令或者数据。每个存储器字节都有一个32位的助记符，称为*物理地址*（physical address）。一个物理地址通常用一个八位的十六进制数表示。第一个地址为00000000_{16}，最后一个地址可以达到为无符号数的$FFFFFFFF_{16}$。图2-1给出了一台PC中可能的存储器的逻辑图。由$FFFFFFFF_{16}=4\ 294\ 967\ 295$可知，PC微机的存储器字节数可达到4 294 967 295，即4G字节。实际上，大多数用户存储容量比这个要小得多。

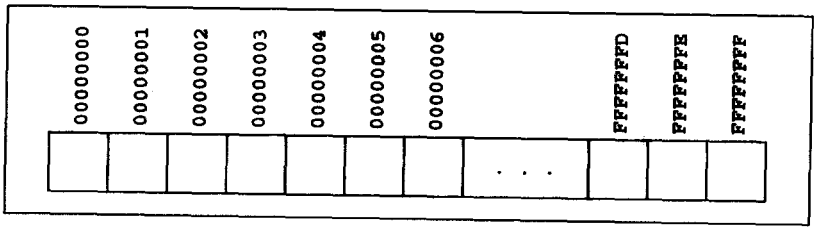


图2-1　PC存储器的逻辑图

在80386之前，Intel 80x86处理器系列仅仅能够直接寻址的存储器为2^{20}字节，使用20位物理地址，通常用5个十六进制数表示，范围从00000～FFFFF。

PC存储器物理上由集成电路（ICs）组成。有些集成电路提供了*随机存取存储器*（random access memory, RAM），程序指令可直接对RAM进行读写。如果关掉计算机电源，RAM的内容会丢失。还有一些集成电路提供了*只读存储器*（read-only memory, ROM），ROM可永久性地保存其内容，但只能对ROM进行读操作，而不能进行写操作。

⊖　INTEL曾经生产过80186 CPU，但它很少用于商业计算机。

本书中的汇编语言程序使用*平面存储模式*（flat memory）。这意味着，逻辑上指向存储数据和指令的存储单元的地址实际上是用32位的地址编码的。

Intel 80x86体系结构还提供了*分段存储模式*（segmented memory），早期的8086/8088 CPU只有这种模式是可用的。在8086/8088中，PC存储器可看作是段的集合，每个段是64K字节长度，以16的倍数作为一个段的开始地址。也就是说，如果一个段的起始地址是00000，另一个段（重叠第一个段）的起始地址就是16（00010_{16}），下一个段的起始地址是32（00020_{16}），其他段的起始地址依此类推。注意，用十六进制表示一个段的起始地址时，其末位始终是0。一个段的*段号*（segment number）由其物理地址的前四个十六进制数组成。

8086/8088微机中的程序并没有直接使用五位的十六进制数地址，事实上，每个存储单元的定位取决于段号和从该段开始处的一个16位的*偏移量*（offset）。通常，程序只写出偏移量，段号可通过上下文中来推断。偏移量是指从段的第一个字节到要定位地址的距离。在十六进制中，偏移量大小从0000到$FFFF_{16}$。用段和偏移量来标识地址：首先是一个四位的十六进制数的段号，接着是一个冒号（:），最后是一个四位的十六进制数的偏移量。

例如，18A3:5B27是指段的起始地址为18A30，偏移量为5B27字节（从段起始地址计有5B27的长度）。段的起始地址加上偏移量就得到五位的十六进制数的地址。

```
  18A30        段号18A3的起始地址
+ 5B27         偏移量
  1E557        五位的十六进制数
```

从80386开始，80x86系列处理器既有16位也有32位的分段存储模式。段号仍是16位长，但不直接指向存储器中的一个段。事实上，段号仅仅是包含真正32位段的起始地址的表的索引。在32位分段模式中，32位的偏移量加上该段的起始地址可得出内存操作数的实际地址。对编程人员而言，段在逻辑上是很有用的：在Intel的分段模式下，编程人员通常为代码、数据和系统堆栈分配不同的内存段。80x86平面存储模式是真正的32位分段模式，所有的段寄存器包含相同的值。

事实上，当程序执行时，由程序产生的32位地址不一定是某个操作数存储的物理地址，另外还有操作系统和Intel 80x86 CPU执行的的存储管理层。分页（paging）机制用于将程序的32位地址映射成物理地址，当程序所产生的逻辑地址超过计算机实际的物理内存空间时，分页机制就非常有用了。如果程序太大而不能全部装入物理内存时，分页机制也可用于将部分程序在需要的时候从磁盘交换到内存中。当用汇编语言编程时，分页机制对程序员是透明的。

练习2.1

1. 假定微机的RAM是32M字节数，则最后一个字节的八位十六进制数的地址是多少？
2. 假定一个微机的视频适配器预定的RAM地址是从000C0000到000C7FFF，则其存储空间的字节数是多少？
3. 假定微机是Intel 8086。请计算下列每对段偏移量所对应的五位十六进制数的物理地址。
   (a) 2B8C:8D21　　(b) 059A:7A04　　(c) 1234:5678

2.2 微机的硬件：CPU

早期的8086/8088 CPU能够执行超过两百种的不同指令。随着80x86系列扩展到80286、

80386、80486以及Pentium处理器，CPU可执行更多的指令。本书探讨如何用这些指令执行程序，从而理解机器级的计算机的性能。其他生产商生产的CPU也执行基本相同的指令集，因此，在80x86上编写的程序在这些CPU上不用改变也可运行。许多处理器系列执行不同的指令集，但是大多数有类似的结构，因此，80x86的基本原理同样适用于这些系统。

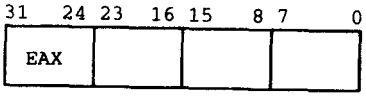
CPU包含许多寄存器（register）。访问每个内部存储地址要比访问RAM快得多。应用寄存器主要跟编程人员有关。一个80x86 CPU（从80386开始）有16个应用寄存器。常用的指令可在寄存器与存储器间传输数据，也可对存储在寄存器或者存储器中的数据进行操作。所有的寄存器都有名字，一些寄存器有着特定的用途。下面将给出这些寄存器的名字，并详述它们的用途，在以后的内容中将会对它们有更多的讨论。

寄存器EAX、EBX、ECX和EDX称为数据寄存器（data register）或者通用寄存器（general register）。EAX有时也是累加器，因为它用于存储许多计算的结果。下面是一条使用EAX寄存器的指令的例子:

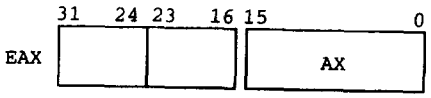
```
add eax, 158
```

将十进制数158（转换成双字长的二进制补码形式）与EAX中已有的数相加，用相加的和取代EAX中原来的数。（加法指令和下面提到的其他指令将在第4章详细讨论。）

寄存器EAX、EBX、ECX和EDX都是32位长，Intel转换是以低位的0开始，从右向左，按数位转换。因此，如果把每个寄存器看成四个字节，则这些位用数表示为:



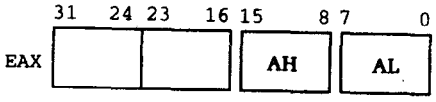
寄存器EAX整体上按照地址可分成若干部分。低位字：位数从0～15，就是大家所知的AX寄存器。



指令 `sub ax, 10`

表示从存储在AX寄存器中的字中减去10，而EAX寄存器的高位数（16～31）没有任何改变。

同样，AX寄存器的低位字节（0～7位）和高位字节（8～15位）分别就是通常所说的AL和AH。



指令 `mov ah, '*'`

复制2A，也就是将星号（*）的ASCII码复制到位8～15，不改变EAX其他的任何位。

寄存器EBX、ECX和EDX也有低位字BX、CX和DX，它们又可按照高位和低位字节划分为BH和BL、CH和CL、DH和DL。BH、BL、CH、CL、DH和DL的每位的改变不会改变相应寄存器的其他位。可能令人奇怪的是，对于EAX、EBX、ECX以及EDX的高位字部分，却没

有对应的命名，所以不能通过名字来独立访问位16～31。

8086到80286处理器有四个16位的通用寄存器，称之为AX、BX、CX和DX。增加的“E”表示“扩展”，将16位扩展成32位的80386寄存器。但是，80386以及后来的体系结构都有效的包含了以前的16位的体系结构。

另外还有四个32位的通用寄存器，分别是ESI、EDI、ESP和EBP。事实上，可以用这些寄存器做算术之类的操作，但通常必须保留它们，以用于特定的用途。ESI和EDI寄存器是*索引寄存器*（index register），其中SI代表源索引，DI代表目的索引。它们的用途之一是，当一串字符从存储器的一个地方复制到另外一个地方时，可指明源地址和目的地址；它们也可实现数组索引。ESI和EDI的低位字SI和DI也可独立使用，但本书很少这样使用。

寄存器ESP是系统栈的*栈指针*（stack pointer），它很少直接通过程序来改变，但当数据入栈或者出栈时会改变。ESP寄存器在堆栈的用途之一就是过程（子例程）调用。过程调用指令地址紧跟在存储于栈中的过程调用指令之后，当调用返回时，这条指令地址就可从堆栈中取出。第6章将会详细的探讨堆栈以及栈指针寄存器。SP可用于ESP的低位字，但本书不这样使用。

寄存器EBP是*基址指针寄存器*（base pointer）。通常，堆栈中被访问的数据项仅仅是存放在栈顶的数据项。然而，EBP寄存器除了标识栈顶位置外，也经常用于标识栈中的某一个固定位置，因此，可以访问这个固定位置附近的数据。EBP也用于过程调用，尤其是带有参数的过程调用。

还有六个16位的*段寄存器*（segment register）：CS、DS、ES、FS、GS和SS。在以前的16位分段存储器模式中，CS寄存器包含有代码段的段号，即存储当前正在执行的指令的存储器区域。由于一个段是64K长，一个程序的指令集通常在64K的范围内；如果一个程序长度超过64K，则该程序在运行时，需要改变CS的值。同样，DS包含数据段的段号，即存储大部分数据的存储器的区域。SS寄存器包含有堆栈段的段号，即保留的栈。ES寄存器包含有附加数据段的段号，该段有多种用途。80386增加了FS和GS，它们便于访问两个附加数据段。

在平面32位存储器模式中，编程人员不太考虑段寄存器。操作系统给每个CS、DS、ES和SS相同的值。回想一下，这是一个表的入口指针，该表包含段的实际起始地址，也包含程序的大小。因此，当程序随意或者故意对另一个区域进行写操作时，操作系统可提示错误。但是，如果根据32位地址来考虑，则这些对编程人员是透明的。

32位*指令指针*（instruction pointer），或称为EIP寄存器，汇编语言的编程人员是不能直接对其进行访问的，CPU必须从存储器中取出要执行的指令，并且，EIP跟踪下一条待取指令的地址。如果是比较老式的、简单的计算机体系结构，下一条待取指令可能也是下一条将要执行的指令。事实上，80x86 CPU在执行前一条指令时，它就取随后要执行的指令了。假设（通常是正确的）：下一次将执行的指令在存储器中是有序紧随上一条指令的。如果这种假设证明是错误的，例如，如果执行一个过程调用，则CPU丢掉已经存储的指令，而且设置EIP包含这个过程的偏移量，这样CPU从新的地址取下一条指令。

另外，对于预取指令，80x86 CPU在完成前一条指令的执行前，实际上，它就开始执行这个预取指令了。*流水线*（pipeline）使用了这种方法，加快了处理器的有效速度。

还有的寄存器称之为*标志寄存器*（flags register），名为EFLAGS指的就是这种寄存器，但是这个助记符在指令中不使用。它的32位中的一些位用于设置80x86处理器的某些特征，其他

的位，称为状态标志位（status flags），表示指令执行的结果，标志寄存器32位中的有些位的名字在表2-1中给出，经常会使用到它们。

表2-1 部分标志位

位	助记符	作用
0	CF	进位标志位
2	PF	奇偶校验标志位
6	ZF	全零标志位
7	SF	符号标志位
10	DF	方向标志位
11	OF	溢出标志位

第11位是溢出标志位（OF），当做加法运算时，没有产生溢出该位置为0，当有溢出产生时，则置1。同样，第0位，即进位标志位（CF），加法运算时，表示在符号位有或者没有进位输出。第7位是符号标志位（SF），是某些运算结果最左边的位，如果运算结果左边的位是0，表示的是非负的二进制补码数；如果左边的位是1，则表示是一个负数。第6位是零标志位（ZF），如果运算结果是0，则该位设置为1；如果运算结果非0（正数或者负数），则设置为0。第2位是奇偶标志位。如果运算结果中1的个数是偶数，则该位设置位1；如果操作结果中1的个数为奇数，则该位置为0。其他的标志位将在以后使用到的时候再详细讨论。

下面的举例说明标志位的设置。还是看这个例子：

```
add   eax, 158
```

这条指令影响标志位CF、OF、PF、SF和ZF。假定在该指令执行前，EAX的内容是字FF FF FF F3，由于158_{10}等同于字00 00 00 9E，这条指令将FF FF FF F3与00 00 00 9E相加，并将相加的和00 00 00 91放入EAX寄存器。由于存在进位，CF的值设为1；由于没有溢出，溢出标志位OF设置为0；符号标志位SF设为0（相加的和的最左边的位是0）；零标志位ZF设为0；因为相加的和不为0；奇偶标志位是0，因为0000 0000 0000 0000 0000 0000 1001 0001包含的1的个数是奇数。

总之，80x86 CPU使用16个内部寄存器存储操作数和运算结果，并且跟踪段选择器和段地址，可执行很多指令，表2-2对这些寄存器进行了总结。

表2-2 80x86寄存器

名字	长度（位）	使用/说明
EAX	32	累加器，通用 低位字AX，可分为AH和AL
EBX	32	通用 低位字BX，可分为BH和BL
ECX	32	通用 低位字CX，可分为CH和CL
EDX	32	通用 低位字DX，可分为DH和DL
ESI	32	源索引；串复制和数组索引的源地址
EDI	32	目的索引；目的或数组索引的地址

（续）

名　字	长度（位）	使用/说明
ESP	32	栈指针；栈顶指针
EBP	32	基址指针；栈中参考位置的地址
CS	16	代码段选择器
DS	16	数据段选择器
ES	16	附加段选择器
SS	16	栈段选择器
FS	16	附加段的选择器
GS	16	附加段的选择器
EIP	32	指令指针；下一条待取指令的地址
EFLAGS	32	标志集；或者状态位

练习2.2

1. 对于下面的每条加法指令，假定在指令执行前，EAX已有给定的内容，给出指令执行后的EAX的值和标志位CF、OF、SF和ZF的值。

指令执行前的EAX	指　令
(a) 00 00 00 45	`add eax, 45`
(b) FF FF FF 45	`add eax, 45`
(c) 00 00 00 45	`add eax, -45`
(d) FF FF FF 45	`add eax, -45`
(e) FF FF FF FF	`add eax, 1`
(f) 7F FF FF FF	`add eax, 100`

2. 在一个8086程序中，假定数据段寄存器DS包含的段号为23D1，并且一条指令在该数据段的偏移量为7B86处取出一个字，则这个取出的字的五位十六进制数地址是多少？
3. 在一个8086程序中，假定代码段寄存器CS 包含的段号为014C，并且指令指针IP的内容为15FE，则下一条待取指令的五位十六进制数的地址是多少？

2.3 微机硬件：输入/输出设备

CPU和存储器组成了计算机，但是如果没有用来获取数据的输入设备，或者没有用来显示或写数据的输出设备，那么计算机就没有多大用处。典型的输入/输出（I/O）设备（input/output device）包括一个用于输入的键盘或者鼠标，一个用于输出和显示的监视器，以及一个用于数据和程序存储的磁盘设备。

汇编语言编程人员可以从多种角度看待I/O设备。在最低级，每个设备在I/O地址空间中使用一个地址集或者端口（port）集。80x86结构有一个64K的端口地址空间，并且有一个典型的I/O设备，使用三到八个端口，这些地址不同于一般的内存地址。程序员可以使用指令或者命令将数据输出到这些端口，或者从这些端口输入数据或状态信息。这样编程是非常枯燥乏味的，而且最后的程序很难在其他不同的计算机系统上重用。

如果计算机系统不是使用单独的端口地址，而是使用部分有序的内存地址作为I/O端口地址，这种设计方法称之为存储器映射输入/输出（memory-mapped input/output）。尽管存储器

映射输入/输出在 80x86中可能用到，但在其他大多数微机中并不使用。

由于对I/O设备进行低级层次的编程比较困难，一个常用的方法是使用程序过程来做繁忙的设备通信工作，同时，编程人员可以从更高级、更逻辑的角度来看待设备。许多例程仍然是相当低级的，例如：用过程将一个字符显示在CRT上，或者从键盘得到一个简单的字符；而一个高级的过程可能是在打印机上打印一串字符。

通过对输入/输出端口和设备的了解，一个汇编语言编程人员可编写输入/输出过程；一些计算机有内置在ROM中输入/输出过程；许多操作系统（见2.4节）也提供输入/输出过程。

练习2.3

假设按前面的讨论有64K的端口地址，则：

1. 64K的端口地址是多少地址（用十进制描述）？
2. 假定第一个地址是0，则最后一个地址是多少？
3. 用十六进制表示这个端口地址的范围。

2.4 PC软件

没有软件，计算机硬件实质上就没什么用处。*软件*（software）指的是由硬件执行的程序或者过程，本小节讨论不同类型的软件。

PC软件：操作系统

一个通用计算机系统需要一个操作系统，以使其他的程序能在其上运行。最初的IBM PC通常运行的操作系统是PC-DOS；类似的兼容操作系统称为MS-DOS。DOS是*磁盘操作系统*（disk operating system）的缩写，所有的MS-DOS由微软公司开发。运行在IBM PC机上的是IBM开发的PC-DOS，而运行在其他的计算机系统的MS-DOS的有些版本是由它们的硬件生产商定制的，以后的PC-DOS版本都是由IBM单独开发的。

DOS操作系统提供给用户*命令行界面*（command line interface），DOS显示一个提示符（如C:\>），并等待用户输入命令，当用户按下Enter（或者Return）键后，DOS将对用户输入的命令进行解释。用户命令可执行DOS已有的功能（如显示磁盘上文件名的目录），或者可以是被装载、运行的程序名。

许多用户更喜欢*图形用户界面*（graphical user interface），其显示图标来表示任务和文件，所以当用户进行选择时，可以用鼠标点击一个图标。Microsoft Windows给微机提供一个图形用户界面，Windows 3.1改进了操作环境，但是仍然需要DOS来运行。Windows 95包含了一个主要修订版本的操作系统，它不再与图形用户界面分开出售。在Windows 95中，尽管命令行界面仍然使用，但图形用户界面已经成为主要的用户界面。

PC软件：文本编辑器

文本编辑器（text editor）是一个程序，它允许用户创建或者修改存储在磁盘上的文本文件。一个文本文件是一个ASCII码集。本书中大多数文本文件是汇编语言源代码文件，这些文件包含汇编语言的语句行。有时，文本编辑器也用于生成数据文件。

后期的MS-DOS版本和Windows 95都提供了*文本编辑器*Edit。Edit在命令行提示下被调用，满屏编辑器用显示器的全部或者部分作为显示文件的窗口。用户可以使用上下键（或者左右

键）移动窗口来显示文件的不同部分。如要对文件进行改动，光标控制键或者鼠标可以将光标移动到需要修改的位置，并且键入所需修改的内容。

Microsoft Windows还有一个称为记事本（Notepad）的文本编辑器，它也是一个满屏编辑器。无论是文本编辑器还是记事本，它们对于写汇编语言源程序都很有用。

字处理器也是文本编辑器，它提供格式化文档和打印文档等服务。例如，如果用文本编辑器，则在每行的结束处必须输入回车键，但如果用字处理器，通常输入时具有自动换行的功能，回车或者其他某些键仅仅在一段的结束处使用。字处理器要考虑把字放在指定的页面设置内的每一行。有时，字处理器也可用作编辑器来编辑汇编语言源代码文件，但有些字处理器把文本的ASCII码连同文件的格式信息一起保存，这些保存的额外信息使文件不适合做汇编语言源代码文件，因此，在创建一个汇编语言源程序时，最好不要使用字处理器。

PC软件：语言翻译器和链接器

语言翻译器是程序，它可以将程序员编写的源代码翻译成可以被计算机执行的格式。

操作系统通常不提供语言翻译器，翻译器可分为：解释器、编译器或者汇编器。

解释器（interpreter）能直接解释一个源程序。为了执行一个程序，解释器先扫描一行源代码，然后执行该行的指令。Basic和Lisp语言程序通常由解释器来执行。尽管解释器自身可能是一个非常有效的程序，但被解释的程序有时执行起来相对比较慢。解释器使用一般很方便，因为它允许一个程序快速修改和运行。解释器本身通常就是一个很大的程序。

编译器（compiler）以源代码开始，并生成CPU可执行指令的目标代码。高级语言如Pascal、Fortran、Cobol、C和C++等通常都需要编译。通常，由编译器生成的目标代码需要链接或者与其他目标代码结合才能生成可装载和执行的程序，因此，需要一个称为*链接器*（linker）的应用程序。一般情况下，编译器提供了链接器。

调试器（debugger）使编程人员能控制程序的执行，在每一条指令后或者预设的中断点处暂停。当程序暂停时，编程人员可以检查高级语言中变量的内容，或者是汇编语言中寄存器、存储器的内容。调试器对于发现错误、“透视”计算机内部以及跟踪计算机程序的执行都是非常有用的。

集成开发环境（integrated development environment）提供单一的界面，通过这个界面可以使用编辑器、编译器和链接器。在开发程序时，能通过它们执行程序，同时还提供一些其他工具，如调试器等。集成开发环境使用起来非常方便，但对于某种特定的程序语言可能不适用。

汇编器（assembler）使用起来更像是编译器，但汇编器是把汇编语言而不是其他高级语言翻译成机器代码。汇编后的文件通常被链接，形成可执行文件。因为汇编语言比高级语言更接近机器代码，所以汇编器比编译器做的事情要简单一些。从历史角度来看，汇编器比编译器出现的时间早。

还是以2.2节所述的汇编语言指令为例，

```
add eax, 158
```

汇编器将该指令翻译成五个字节的05 00 00 00 9E，其中第一个字节是op代码（op code，操作码），该操作码将随后四个字节中存放的数与EAX中的双字数相加，双字数00 00 00 9E是十进制数158的二进制补码表示。

本章小结

本章讨论了组成计算机系统的软件和硬件部件。

计算机系统中主要的硬件部件包括CPU和存储器，其中CPU执行指令，它使用其内部寄存器来存放指令操作数、运算结果，并且确定存放在存储器中的指令和数据的地址。在内存中的数据能够用32位地址数来寻址，在平面存储模式中，这个地址数就是真正有效的地址；在分段模式中，地址是由一个段的起始地址和该段的偏移量计算得出的。

硬件级的输入/输出使用一个独立的地址集，称之为端口。输入/输出通常由操作系统应用程序完成。

操作系统是软件中至关重要的一部分。通过命令行或者图形用户界面，可以解释用户执行命令的请求，或者装载和执行程序。

对汇编语言程序员而言，文本编辑器、汇编器和链接器是必要的软件工具，它们可以是独立的程序，或者是可用的一部分集成开发环境。调试器也是程序员的一个有用的工具。

第3章 汇编语言的要素

第3章讨论如何编写汇编程序。第一部分首先详述了MASM（微软的Macro汇编器）可使用的句子的类型和格式。然后用一个完整的汇编语言程序例子，讨论了如何汇编、链接和执行程序。本章的最后一节详细剖析了这个例子的结构，为后续章节的讨论打下了基础。

3.1 汇编语句

一个汇编语言的源代码文件包含了一系列语句（statement）。大部分语句每行少于80个字符，这样的限制是为了便于打印或者在屏幕上显示。但是，MASM6.1还允许不长于512个字符的句子，在每行的行尾用斜杠“\”（除了句末）将一条语句拆分成多行。

由于汇编语言本身很难读懂，所以需要大量的注释（comment）。注释用句子的形式表示。在注释的开头用分号“;”作标识，直到一行结束都是注释。如果分号是在第一列或者注释是紧跟在一个有效的句子后面，那么整行都是注释。在少数的情况下，可以使用了斜杠“\”将一个注释语句拆分成了几行，紧跟在斜杠后面的那部分也是注释。

汇编语言的语句有3种类型：指令性语句（instruction）、指示性语句（directive）和宏（macro）。指令性语句要通过汇编器翻译成目标代码（机器代码），在运行时执行。每一条指令（instruction）都被惟一地翻译成了80x86 CPU可以执行的操作。例如，在第2章中作为例子出现过的指令：

```
add eax, 158
```

一条指示性语句告诉汇编器做某种操作。这种操作并不会产生机器指令，也不会对目标代码有任何影响。例如：汇编器可以产生一个显示源代码、目标代码和其他信息的列表文件。但是如果在源文件的任何地方出现下面的语句：

```
.NOLIST
```

那么汇编器停止在列表文件中显示源语句，但生成的目标代码有没有.NOLIST都是一样的。（.LIST指示性语句是恢复显示源代码语句清单的。）这两条语句以及其他指示性语句是周期性出现，而其他类型的语句则不一定。

一个宏语句是一系列语句的缩写，它们可以是指令性语句、指示性语句，或者宏。汇编器将一个宏展开为它表示的多条语句，然后再汇编这些语句。本章稍后介绍的例子中会使用到宏。

一条语句不仅仅有简单的注释，它还包括可以表明目的的助记符以及其他三个部分：名字、操作数、注释。这些部分必须以下列顺序排列：

```
名字    助记符    操作数; 注释
```

例如，一个程序包含下面这条语句：

```
ZeroCount: mov eax, 0; 初始化数据为0
```

作为一条普通的指令，名字部分以“:”结尾。如果作为指示性语句，名字后面不跟冒号“:”。语句中的助记符用来标识该语句是特定的指令、指示性语句或者是宏语句。有些语句没有操作数，有些有一个，有些有多个。如果有多个操作数，中间以逗号分隔，也可以加空格。有时候单操作数有多个部分组成，中间用空格作分隔，看起来像有多个操作数。

例如指令:

```
add eax, 158
```

助记符是add，操作数是eax和158。汇编器将add作为执行加法运算指令的一个助记符，操作数提供了汇编器需要的其他信息。第一个操作数eax告诉汇编器用EAX寄存器的双字节数作为被加数，并且相加之和将存储在EAX中。由于第2个操作数是一个数字（不同于指定的寄存器或者存储器的数），汇编器知道这是一个和EAX寄存器中双字节数相加的加数。翻译出来的目标代码就是05 00 00 00 9E，05表示“将跟在05后面的双字节数和存储在EAX中数字相加”。汇编器要将十进制数值158转换成双字节长度的2进制补码0000009E。这个双字节长的数存储于目标代码中，这点通常无须考虑。

名字域的用处之一是在程序汇编和链接后，代表指令在内存中的地址。其他的指令就可以很容易地找到这个被标识的指令。如果上述的加法指令需要在程序中循环执行，那么可以这样写:

```
addLoop: add eax, 158
```

这条指令是jmp（跳转）指令的目的地址，汇编语言版的goto语句:

```
jmp    addLoop;    重复加
```

注意，冒号没有出现在跳转指令名addLoop的结尾处。高级语言中的循环结构，例如while循环或者for循环在机器语言中不能使用，但是它们可通过使用jmp或其他的跳转指令来实现。

有时候一行只包含一个名字的源代码也是很有用的。例如:

```
EndIfBlank:
```

这个标识可作为代码的最后一行，用来实现if-then-else-endif结构。对于跟在该名字后面的指令，这个名字将非常有用。但是，倘若不考虑其后面跟了什么内容，要实现某一结构就更方便了。

用标号描述是非常好的编码习惯。标号addLoop可以使这段汇编语言代码的结构更清晰，说明一个程序循环体的第一条指令包括一个加法运算。其他标号如上面提到的EndIfBlank，可以作为伪代码的相应关键字。

在汇编语言中，名字和其他标识性字符是由字母、数字和特殊字符组成的。允许的特殊字符有下划线（_）、问号（?）、美元符号（$）、at符号（@），本书很少使用特殊符号。名字不能以点开头。标识符最多可以有247个字符，这样比较容易地组成有意义的名字。对于指令性指令助记符、指示性指令助记符、寄存器或者其他对汇编器来说有特定意义的单词，微软的Macro汇编器不允许将其作为名字。附录C包含了保留标识符的清单。

汇编程序代码通常不易阅读，但是，编写的程序终究要被其他人阅读。因此，要考虑程序的可读性。养成良好的编码风格，尽量使用小写字母，这些都有助于提高程序的可读性。

回忆一下汇编语言，包括名字、助记符、操作数和注释部分。一个风格良好的程序从始

至终都包含这几个部分。通常将名字放在第1列，助记符从第12列开始，操作数从第18列开始，注释从第30列开始，保持整体的一致性比强调某一列位置更重要。在汇编语言的源文件中可以出现空白行。用空白行来有效地分割汇编语言的不同结构，就好像把文章分割成自然段一样。

汇编语言可以使用小写字母，也可以使用大写字母。通常情况下，不区分大小写。可以用标识符来区别它们，当然，这种情况只在要使用的编程语言对大小写敏感时才适用。混合大小写的代码比全部用大写或者全部用小写的代码更容易阅读。全部是大写的代码尤其难读。常规做法是大部分代码使用小写字母，只在指示性指令时使用大写字母。本书中的所有程序都遵循这一规则。

练习3.1

1. 说出并描述汇编语言语句的3种类型。
2. 对于下面每一个字母组合，指出其是否是一个合法的名字。如果不是，请给出理由。

(a) repeat	(b) exit	(c) more
(d) EndIf	(e) 2much	(f) add
(g) if	(h) add2	(i) EndOfProcessLoop

3.2 一个完整的实例

本节从程序的伪代码设计开始，讨论一个的完整汇编语言程序。讨论汇编语言的细节容易让人觉得有些难，因此，首先做好设计，然后再涉及详细的汇编语言代码，这样较容易编写代码。下面程序会输入两个数字，然后作加法。本程序的设计如下：

```
输入第一个数字；
输入代表第一个数字的ASCII码字符；
将此字符转化为二进制补码的双字；
存储第一个数字；
输入第二个数字；
输入代表第二个数字的ASCII字符；
将此字符转化为二进制补码的双字；
将第一个数字加到第二个数字上去；
将和转化为ASCII码字符；
显示标号及代表和的字符；
```

代码段3-1列出了使用该设计的完整代码，下面详细介绍各个部分。

代码段3-1 一个完整的汇编语言程序

```
; 两个数相加的汇编程序例子
; 作者:    R. Detmer
; 日期:    1997年7月修正

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

INCLUDE io.h                ; 输入/输出的头文件
```

```
cr      EQU     0dh             ; 回车符
Lf      EQU     0ah             ; 换行

.STACK  4096                    ; 保留4096字节的堆栈

.DATA                           ; 数据保留区
number1 DWORD   ?
number2 DWORD   ?
prompt1 BYTE    "Enter first number:  ", 0
prompt2 BYTE    "Enter second number:  ", 0
string  BYTE    40 DUP (?)
label1  BYTE    cr, Lf, "The sum is "
sum     BYTE    11 DUP (?)
        BYTE    cr, Lf, 0

.CODE                           ; 程序代码开始
_start:
        output  prompt1         ; 提示输入第一个数
        input   string, 40      ; 读取ASCII字符
        atod    string          ; 转换为整型
        mov     number1, eax    ; 存入存储器

        output  prompt2         ; 重复输入第二个数
        input   string, 40
        atod    string
        mov     number2, eax

        mov     eax, number1    ; 第一个数存入EAX寄存器
        add     eax, number2    ; 与第二个数相加，和存入EAX
        dtoa    sum, eax        ; 将和转换为ASCII码字符
        output  label1          ; 输出和

        INVOKE  ExitProcess, 0  ; 退出，返回代码0

PUBLIC _start                   ; 公开入口点

END                             ; 源代码结束
```

该例子从说明程序目的的注释、程序的作者以及编写日期开始。这是任何程序都必须具备的最简单的文档，通常还需要更多的信息。但是，为了节省空间，本书中的程序文件都比较简短。不过，大部分的代码行都有注释。

语句:

```
.386
.MODEL FLAT
```

都是指示性语句。如果没有命令.386，MASM就只会接受8086/8088系列的指令；有了这条语句后，汇编器就还可以接受80186、80286以及80386等处理器的指令。.486和.586指令可以让汇编器处理更多的指令，但是，这里不会用这些指令来编写程序。还有一个.386p指令也允许汇编器识别80386特有的那些指令，但这里也不使用它们。指令.MODEL FLAT告诉汇编器用一个平面存储模式来生成32位的代码。在MASM6.1中，这条指令必须跟在.386指令的后面。

下一条语句:

```
ExitProcess  PROTO  NEAR32  stdcall,   dwExitCode:DWORD
```

是一个指示性语句。PROTO用来生成一个函数体。在本例中，函数的名字是：ExitProcess，用于结束程序的系统函数。它只有一个符号为dwExitCode的双字节参数。

下一条语句:

```
INCLUDE io.h
```

也是一个指示性语句。(尽管如此，但是一个程序并不是只有指示性语句！)当编译该程序时，它指示汇编器拷贝一个文件IO.H到程序中[⊖]。源文件没有经过修改，它仍然包含INCLUDE语句，但是为了编译，汇编器最后会将IO.H插入到INCLUDE语句中。为了被包含，编译时，IO.H文件必须与源文件在同一个目录或者同一个路径下。

IO.H文件包含了章节3.7中定义的大部分宏语句，以及其他一些指示性语句。在列表文件中，来自IO.H文件的语句是.NOLIST、.LIST，和其他一些少量的注释语句。指示性语句.NOLIST语句，如上所述，隐藏了IO.H文件中的大部分语句行。IO.H文件中的最后一条语句是指示性语句.LIST，它要求汇编器恢复列出源代码语句。IO.H文件中的另外一个指示性语句要求汇编器将语句列表用宏扩展表示。这样，程序就更简短。

下面给出两条语句:

```
cr      EQU  0dh;  回车符
Lf      EQU  0ah;  换行符
```

使用了EQU给符号赋值。用EQU后，在随后的代码中，该符号就表示相应的值。用符号来代替数值使用，可以使程序看起来更清晰。本例中，cr赋值为十六进制的0D，它是回车字符的ASCII值；换行符Lf赋值为十六进制的0A，是换行符的ASCII值。使用大写的L，以免和数字1混淆。另起一行时，需要使用回车和换行符，通常，当定义的数据要在屏幕输出或者打印时，也使用这两个字符。

在EQU语句中，汇编器将0dh和0ah作为十六进制数，因为后面有一个h。除非特别说明，在汇编程序中的数一般都作为十进制数。除了十六进制数，其他类型的后缀将在3.5节中介绍。一个十六进制的数都是以数值开头的，而不是十六进制中的a到f，所以汇编器可以将它与名字区分开。

.STACK语句告诉汇编器运行时堆栈要保留多少字节，通常是保留4096个字节。栈通常用于过程调用。IO.H中，每个宏都会生成一个过程调用，去调用另外一个相关的真正执行任务的过程。同样，这些被调用的过程也可以调用其他过程。

.DATA语句是程序的数据段的开始，该数据段中保留了变量的内存空间。在这个程序中，“BYTE”和“DWORD”指示性语句都分别用来存储字节和双字。

语句:

```
number1  DWORD ?
```

保留一个双字的空间，将符号名number1和地址00000000关联起来，因为它是第一个数据项。尽管MASM 6.1将number1视为0，但问号（?）表示这个双字还没有指定初始值。

语句:

```
number2  DWORD ?
```

保留了另一个双字的空间，将符号名number2与下一个可用地址00000004关联起来，因为它是跟在一个已存储的双字后面。程序运行时，number1和number2不仅仅是00000000和00000004地址不同，而且这两个双字要连续地存储在内存中。

⊖ 读者可以使用作者在本书中编写的IO.H、IO.OBJ、IO.ASM文件。

语句:

```
prompt1 BYTE   "Enter first number:  ", 0
```

有两个操作数，字符串“Enter first number”和数字0。它为引号（“”）内的每个字符保留了一个字节长度，为数字0保留了一个字节长度。对于每个字符，该字节存储字符的ASCII码值；而对于数字，存储的只是该数字的二进制补码。因此，这条语句保留了22个字节的存储空间45 6E 74 65 72 20 66 69 72 73 74 20 6E 75 6D 62 65 72 3A 20 20 00。符号名prompt1与地址00000008关联，因为前面的8个字节已经被分配了。

下一条BYTE语句保留了23个字节的存储空间，符号名prompt2从地址0000001E开始。

语句:

```
string    BYTE  40   DUP(?)
```

为字符串string保留了40个未经初始化的字节，DUP运算符要求括号中的项重复。

语句:

```
label1    BYTE  cr, Lf, "The   sum  is "
```

有三个操作数，保留了13个字节的空间。开始的2个字节是0D和0A，它们是cr和Lf两个符号的值。接下来的11个字节都是在引号中字符的ASCII码。注意，对于BYTE指令或下一条指令，这里并没有尾随0操作数，所以结尾也就没有00字节。下一个到最后BYTE指令为sum保留了11个没有初始化的字节，尽管最后BYTE指令没有标号，紧跟在sum的11个字节后面，预留了3个初始化的存储空间。

下一段程序包含了可执行语句，从指示性语句:

```
.CODE
```

开始。

代码行只有一个标号:

```
_start:
```

标识了这个程序的入口点，即第一条要执行语句的地址。程序员可选择取名，但是通常使用_start来标识开始。

现在到了程序真正的处理部分了。因为这个程序主要执行输入输出，所以多数语句是作为宏来实现这些功能的。宏语句:

```
output prompt1
```

显示存储在prompt1的地址中的字符串，使用空字符（00）字节来结束显示。在这个程序中，用户按要求输入第一个数字。由于在提示符后，没有返回值或换行符，所以光标仍然停留在冒号和两个空格的后面的一行。语句:

```
input  string,  40        ; 读ASCII字符
```

是一个函数型的宏，让机器暂停，等待用户从键盘输入字符，直到敲回车来结束输入。第一个操作数(string)定义了这些字符的ASCII代码要存储的位置。第二个操作数（40）定义了输入的字符的最大长度。注意，string保留了40个未初始化的字节存储空间。有关input宏的详细内容将在3.6节中讨论，现在只需要注意需要为输入保留多少空间。

input宏指令输入了ASCII代码，但是CPU只能对二进制补码数做运算。因此atod宏（把ASCII码转换为双字整型数）扫描指定单操作数地址的内存，然后将其中的ASCII代码转换为相应的二进制补码的双字，将结果存储在EAX中。

```
atod    string ;    转换为整型
```

在该程序中，从string开始扫描，跳过开头的空格，注意加号（+）和减号（-），同时将数字转换为ASCII码，遇到任何非数字字符，扫描即停止。

语句:

```
mov     number1, eax    ; 存储在内存中
```

是一条指令性语句。助记符mov代表“move”，但是该指令实际上是完成一个复制操作，如同高级语言中的赋值语句。这个特定的指令把eax寄存器中的数值复制到地址为number1的内存双字中。

接下来的四条语句:

```
output   prompt2            ; 重复操作第二个数
         input    string,  40
         atod     string
         mov      number2, eax
```

重复完成下面的任务：提示输入第二个数字；输入ASCII代码；将ASCII代码转化为二进制补码双字；然后将这个双字拷贝到内存中。注意：输入域是可重复使用的。

接下来的两条指令语句执行加法运算。加法必须在寄存器中执行，因此，第一条指令将数据复制到EAX寄存器中，

```
mov     eax, number1    ; 移动第一个数到AX
```

然后用下面这条指令将第二个数字加到第一个数字上。

```
add     eax, number2    ; 加上第二个数
```

（还有更有效的方法从eax中得到和吗？）

现在相加的和以二进制补码的形式存储在寄存器EAX中。要输出相加和，需要一个ASCII代码来代表这个值。宏dtoa（把双字整型数转换为ASCII码）宏取出第二个操作符定义的双字，并且将它转化为11个字节长的字符串，该字符串存储在第一个操作数定义的目的地址中。在这个程序中，宏:

```
dtoa    sum, eax   ; 转换为ASCII字符
```

使用eax寄存器中的内容作为源操作数，把它对应的ASCII字符存储到11个字节的sum中。对一般的小数字而言，不足部分在开头用空格字符填补。宏:

```
output   label1    ; 输出标号以及和
```

会显示从标号label1开始到空字节（00）字符的存储器的内容。因为sum中的未定义的字节已经赋予了ASCII码值，所以，在这条没有标号的BYTE指令中，内存中的第一个空字节将是跟在回车符和换行符之后的一个字符，总共显示26个字符。

语句:

```
INVOKE   ExitProcess  0 ; 退出并返回0
```

是一个调用过程ExitProcess的指令，在过程体中，参数dwExitCode的值是0。它的功能是终止这个程序，返回0告诉操作系统这个程序已经运行结束。(非0值表示其他错误情况)。

通常在文件中使用的名字只有在文件中可以看到。语句:

```
PUBLIC_start    ; 公开入口点
```

使得这个入口点的地址在文件外也可见，因此当它构成一个可执行程序文件，链接器可以识别执行的第一条指令。这个指示性语句会在后面章节用到，使用它可使单独汇编的过程名在文件外可见。

在这个汇编语言源文件的最后一条语句是指示性语句END，它表示程序的结束，END后面不能跟任何语句。

练习3.2

1. 找出例子程序中出现的三条指示性语句。
2. 找出例子程序中出现的三条宏语句。
3. 找出例子程序中出现的三条指令性语句。
4. 在例子程序中，为什么prompt2地址是0000001E? 这条指令预留的23个字节的内容是什么?

3.3 程序的汇编、链接和运行

本书包含的一张CD提供了一个可以汇编和链接程序的软件，该软件可以安装在计算机上使用。

程序的源代码可以使用任何标准的文本编辑器，例如用记事本或者Edit来编辑。这张CD中不包含文本编辑器。汇编语言的源代码通常存储在.ASM类型的文件中。在本节中，代码段3-1的源文件存储在文件EXAMPLE.ASM中。

使用在MASM6.1中的ML汇编器来汇编程序。为了汇编EXAMPLE.ASM，需要在MS-DOS窗口的DOS提示符下输入:

```
ml /c /coff example.asm
```

如果程序没有什么错误，那么在DOS提示符下会出现如下信息:

```
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp  1981-1993. All rights reserved.

Assembling: example.asm
```

文件EXAMPLE.OBJ将会加到目录中。如果程序有错误，错误信息会显示出来，并且生成no.OBJ的文件。

在调用汇编器时，汇编器有两个转换参数: /c和/coff。ML可以用来汇编和链接，转换参数/c说明只需要编译; 而转换参数/coff要产生一个公共对象文件格式(common object file format)。ML转换参数区分大小写: 必须以小写形式输入。

这里使用的链接器是LINK。例如，在DOS提示符下调用:

```
link /subsystem:console /entry:start /out:example.exe
                                example.obj io.obj kernel32.lib
```

虽然可能不需要输入新的一行，但它作为一个单独的命令输入。同样，如果程序没有错误，在DOS提示符下就会出现:

```
Microsoft (R) 32-Bit Incremental Linker Version 5.10.7303
Copyright (C) Microsoft Corp  1992-1997. All rights reserved.
```

LINK命令将EXAMPLE.OBJ、IO.OBJ，以及KERNEL32.LIB链接起来，产生一个输出文件EXAMPLE.EXE。转换参数/subsystem:console告诉链接器生成一个在DOS环境下运行的控制台程序。转换参数/entry:start标识程序的入口点。注意: 这里并没有使用下划线，虽然_start已经是该程序入口点的实际标识。

在DOS环境下，通过键入一个程序的名字就能执行这个程序。图3-1显示了运行一个EXAMPL.EXE的范例，用户输入的部分用下划线做标识。一旦生成了可执行文件，这个程序不需要再汇编和链接，就可以任意次执行。

```
C:\AsmFiles>example
Enter first number:  98
Enter second number:  -35

The sum is          63

C:\AsmFiles>
```

图3-1 EXAMPLE.EXE的执行

本书的软件包包含微软公司的Windbg，它是一个可以跟踪汇编语言程序执行的调试工具。这是一个有效的查错工具，通过它，也可以了解计算机在机器这一层是如何工作的。

要使用Windbg，必须在ML后敲入/Zi转换参数（大写的Z，小写的i），告诉汇编器在输出时添加调试信息。此时，汇编命令如下:

```
ml /c /coff /Zi example.asm
```

链接器现在多加了一个转换参数/debug，如下:

```
link /debug /subsystem:console /entry:start /out:example.exe
                                example.obj io.obj kernel32.lib
```

在DOS环境下键入Windbg开始调试。此时会出现一个类似于图3-2的窗口。从菜单栏中选择File，然后打开Executable...，选择example.exe文件，或者其他的可执行文件，然后点击OK，返回到类似图3-2的窗口。除此之外，标题栏中增加了example.exe，以及命令窗口中还出现了一些横线。

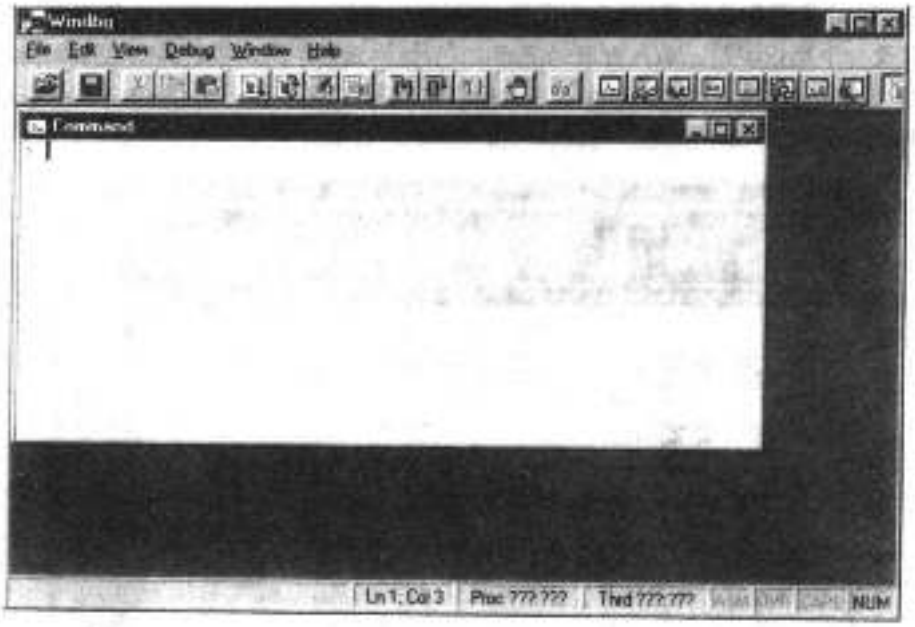


图3-2 Windbg打开屏幕

现在点击“进入”按钮，如图所示：

在信息窗口中点击OK，然后再点击“进入”按钮。现在源代码就出现在Windbg的Command窗口后面的子窗口中。将Command窗口最小化，选择View菜单，然后选择Register子菜单，打开一个窗口，用来显示80x86中寄存器的内容。然后选择View下Memory子菜单，打开一个窗口，用来显示内存的内容。对于该窗口来说，必须输入内存的开始地址。例如，在该例子程序中，使用&number1作为开始地址——C/C++中的取地址符号（&）是用来取得number1的地址，这是数据段的第一项。最后调整一下各个窗口的大小，并且重新排列，让屏幕显示看起来跟图3-3差不多。注意：在Windbg窗口下还是可以看到这个程序的输出窗口的右边缘部分，而桌面的其他部分被运行的汇编器、链接器的窗口以及用户正运行的Microsoft Word的小条带所覆盖。

该例子程序的第一条语句是高亮度显示，点击“进入”按钮，执行这条语句。虽然这是一条宏语句，但可以作为一个单独的指令执行。在输出窗口输出“Enter first number:”（也可以点击输出窗口的边缘，使其置顶）。再次点击“进入”按钮，执行输入宏语句。当输入一个数字，并且按回车键后，Windbg就回到调试窗口，第二条语句高亮显示。第三次点击step into按钮，执行将ASCII代码转换为相应的二进制补码的双字的宏语句，并执行第一个mov指令。现在，Windbg窗口看起来如图3-4所示。

此时，Registers窗口显示EAX值为00000062，它是98的二进制补码的双字形式。数字98是在提示符下输入的。在Memory窗口的第四行可以看到它的ASCII码。Memory窗口的每一行由三部分组成：该行的起始地址，在这些地址中存储字节的十六进制数，可能的话，还有这些字节相应的可打印字符。第四行开始的5个字符是prompt2的串尾，包括：“r”、冒号“:”、两个空格，以及一个null字节等5个字符的ASCII码值。接下来的内存单元中为字符串string预留了40个字节的空间，开始的4个字节是39、38、00和0A，其中39和38是98的ASCII码值，00

和0A是一个null字节和换行符的ASCII码值。当输入98时，操作系统就存入了39和38，一个回车字符以及一个换行字符。输入宏语句用null字节替换回车字符，不过在内存中，仍然可以看到换行符。宏atod扫描这些ASCII代码，然后将值存储在EAX中。Memory窗口中也可以看到number1的值62 00 00 00，但是，通过mov指令复制的值是以相反的次序存放的。

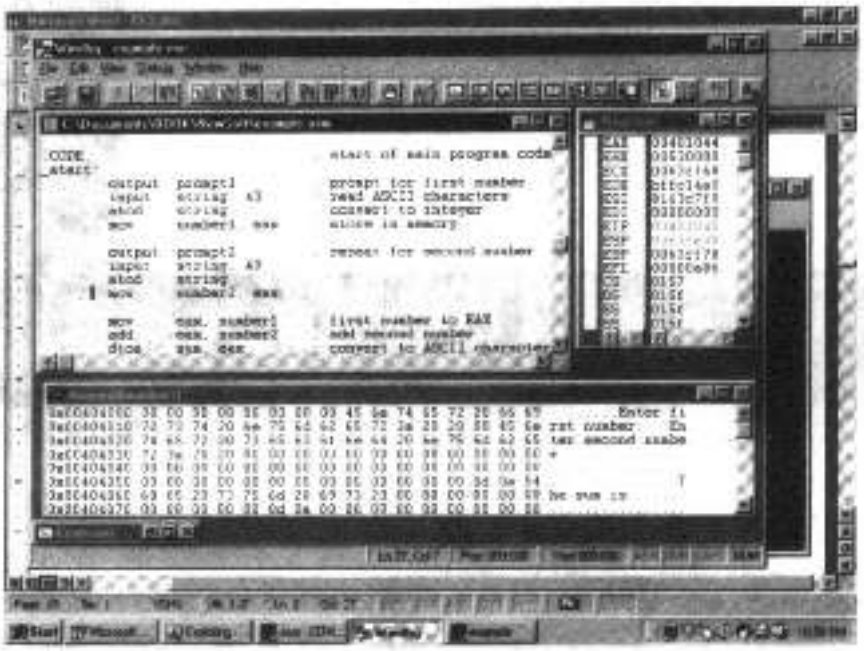


图3-3 Windbg准备跟踪程序

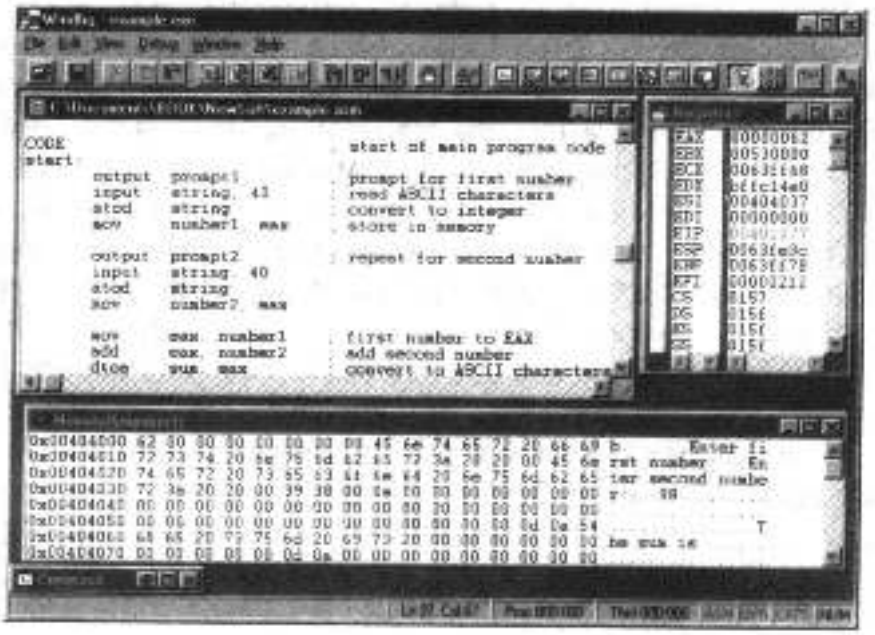


图3-4 Windbg跟踪程序

该程序剩余部分同样可以跟踪。图3-5是程序结束之前的Windbg窗口。滚动Memory窗口，以便显示有输出标识的那部分内容。此时，输入 35作为第二个数字，计算98＋（－35）的和，将结果以二进制的补码形式存储在EAX中，并且由dtoa宏将计算的和转化为11字节长的字符串。这时，内存中，在数字6（ASCII码值为36）和数字3（ASCII码值为33）之前有10个的空格（ASCII码值为20）。

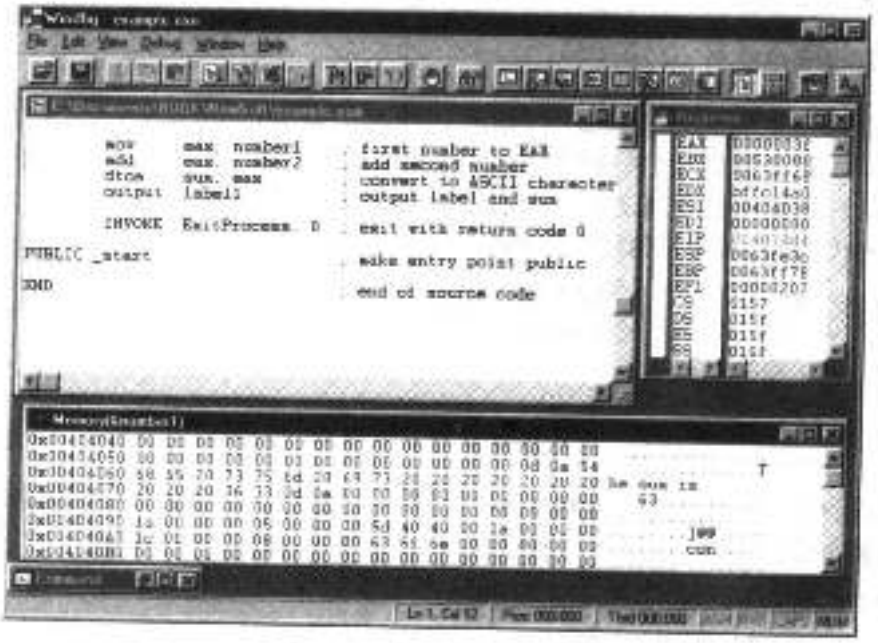


图3-5 程序结束前的Windbg

练习3.3

1. 如果EXAMPLE.ASM根据本章第一条指令性语句来汇编和链接（没有调试），那么汇编器和链接器分别会生成什么文件？
2. 如果EXAMPLE.ASM根据本章第二条指令性语句来汇编和链接（有调试），那么汇编器和链接器分别会生成什么文件？

编程练习3.3

1. 运行本章中给出的例程。使用一个文本编辑器创建源代码文件EXAMPLE.ASM，或者直接从本书带的CD上拷过来，然后汇编、链接和执行该文件，不生成调试代码，使用不同的数据运行几次这个程序。
2. 跟踪本章中给出的例程。使用一个文本编辑器创建源代码文件EXAMPLE.ASM，或者直接从本书带的CD上拷过来，然后汇编、链接和执行该文件，生成调试代码，使用不同的数据跟踪几次这个程序。
3. 修改本章给出的例程：提示、输入，实现三个数相加，将这个源代码文件命名为ADD3.ASM，按本章给出的步骤汇编、链接这个程序，生成ADD3.EXE文件。使用不同的数据运行ADD3。如果有问题或者想跟踪这个程序的执行，可以使用调试程序。

4. 指令性语句:

```
sub   eax, label
```

用存储在寄存器EAX中的值减去存储在label中的值。修改本章所给的例程：提示和输入两个数字，然后用第一个数字减去第二个数字。源代码文件命名为SUBTRACT.ASM。按照本章所给的步骤汇编、链接程序，产生一个SUBTRACT.EXE文件。使用不同的数据运行SUBTRACT.EXE。

3.4 汇编器清单文件

ML汇编器在汇编过程中会生成一个清单文件.LST。这个.LST文件显示源代码、要转换的目标代码以及附加信息。检查这个清单文件有助于理解汇编的过程。如果源文件中包含错误，.LST文件能够在出错的地方显示错误信息，帮助定位错误语句。

现在将代码段3-1中的范例程序EXAMPLE.ASM稍作修改，将:

```
atod  string          ; 转换为整型
mov   number1, eax    ; 存入存储器
```

改成:

```
atod   eax, string    ; 转换为整型
mov    number1, ax    ; 存入存储器
```

修改后的指令有两个错误：宏指令atod只允许一个操作数，并且mov指令的源操作数和目标操作数的长度不同。把修改后的文件保存为EXAMPLE1.ASM.

汇编时，需要额外附加转换器/Fl（大写字母F，小写字母l）生成清单文件:

```
ml /c /coff /Fl example1.asm
```

当在DOS提示符下敲入以上命令后，控制台会显示:

```
        Assembling: example1.asm
example1.asm(32): error A2022: instruction operands must be the same size
example1.asm(31): error A2052: forced error : extra operand(s) in
ATOD
 atod(7): Macro Called From
          example1.asm(31): Main Line Code
```

这些错误提示信息很清楚地指出了在源代码的32行与31行存在错误，而且说明了是什么错误。不过，如果查看EXAMPLE1.LST文件中相应的部分，在出错的语句下也能看到错误信息，如下所示。因此，经常检查清单文件能使查找错误变得更容易。

代码段3-2没有任何错误的源例子程序EXAMPLE.ASM的清单文件，检查该文件能更好地

```
00000000                    _start:
                                   output  prompt1        ; prompt for first number
                                   input   string, 40     ; read ASCII characters
                                   atod    eax, string    ; convert to integer
                        1                .ERR <extra operand(s) in ATOD>
example1.asm(31): error A2052: forced error : extra operand(s) in ATOD
 atod(7): Macro Called From
  example1.asm(31): Main Line Code
```

```
                                mov     number1, ax     ; store in memory
example1.asm(32): error A2022: instruction operands must be the same size

                                output  prompt2         ; repeat for second number
```

理解汇编过程。

代码段3-2 EXAMPLE.LST程序清单

在源代码文件的开头，首先列出文件的注释和指令，在INCLUDE指示性语句后是来自文

```
Microsoft (R) Macro Assembler Version 6.11                    08/04/97 21:21:16
example.asm                                                    Page 1 - 1

                                ; 两个数相加的汇编语言程序例子
                                ; 作者: R. Detmer
                                ; 日期: 1997年7月

                                .386
                                .MODEL FLAT

                                ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

                                INCLUDE io.h                ; 输入/输出的头文件
                              C ; IO.H-I/O宏的头文件
                              C ; 32位平面存储模式
                              C ; R. Detmer  1997年7月
                              C .NOLIST                     ; 在列表中不显示
                              C .LIST                       ; 显示列表
                              C

= 0000000D                      cr      EQU     0dh         ; 回车符
= 0000000A                      Lf      EQU     0ah         ; 换行符

                                .STACK  4096                ; 保留4096字节堆栈

00000000                        .DATA                       ; 数据保留区
00000000 00000000               number1 DWORD   ?
00000004 00000000               number2 DWORD   ?
00000008 45 6E 74 65 72         prompt1 BYTE    "Enter first number:  ", 0
         20 66 69 72 73
         74 20 6E 75 6D
         62 65 72 3A 20
         20 00
0000001E 45 6E 74 65 72         prompt2 BYTE    "Enter second number:  ", 0
         20 73 65 63 6F
         6E 64 20 6E 75
         6D 62 65 72 3A
         20 20 00
00000035  00000028 [            string  BYTE    40 DUP (?)
           00
          ]
0000005D 0D 0A 54 68 65         label1  BYTE    cr, Lf, "The sum is "
         20 73 75 6D 20
         69 73 20
0000006A  0000000B [            sum     BYTE    11 DUP (?)
           00
          ]
00000075  0D 0A 00                      BYTE    cr, Lf, 0

00000000                        .CODE                       ; 主程序代码开始
00000000                        _start:
```

```
                                        output  prompt1          ; 输入第一个数
                                        input   string, 40       ; 读取ASCII字符
                                        atod    string           ; 转换为整型
0000002E  A3 00000000 R                 mov     number1, eax     ; 存入存储器

                                        output  prompt2          ; 重复输入第二个数
                                        input   string, 40
                                        atod    string
00000061  A3 00000004 R                 mov     number2, eax

00000066  A1 00000000 R                 mov     eax, number1     ; 第一个数存入EAX寄存器
0000006B  03 05 00000004 R              add     eax, number2     ; 加上第二个数，和放入EAX
                                        dtoa    sum, eax         ; 将和转换为ASCII字符
                                        output  label1           ; 输出和

                                        INVOKE  ExitProcess, 0   ; 退出，返回代码口

                                PUBLIC _start                    ; 公开入口点

                                END                              ; 源代码结束

Microsoft (R) Macro Assembler Version 6.11                  08/04/97 21:21:16
example.asm                                                  Symbols 2 - 1


Macros:

                N a m e                 Type

atod . . . . . . . . . . . . . .        Proc
atoi . . . . . . . . . . . . . .        Proc
dtoa . . . . . . . . . . . . . .        Proc
input . . . . . . . . . . . . .         Proc
itoa . . . . . . . . . . . . . .        Proc
output . . . . . . . . . . . . .        Proc


Segments and Groups:

                N a m e                 Size     Length   Align   Combine Class

FLAT . . . . . . . . . . . . . .        GROUP
STACK . . . . . . . . . . . . .         32 Bit   00001000 Dword   Stack   'STACK'
_DATA . . . . . . . . . . . . .         32 Bit   00000078 Dword   Public  'DATA'
_TEXT . . . . . . . . . . . . .         32 Bit   00000097 Dword   Public  'CODE'


Procedures,  parameters and locals:

                N a m e                 Type     Value    Attr

ExitProcess . . . . . . . . . .         P Near   00000000 FLAT  Length= 00000000
                                                                External STDCALL

Symbols:

                N a m e                 Type     Value    Attr

@CodeSize . . . . . . . . . . .         Number   00000000h
@DataSize . . . . . . . . . . .         Number   00000000h
@Interface . . . . . . . . . . .        Number   00000000h
@Model . . . . . . . . . . . . .        Number   00000007h
```

```
@code . . . . . . . . . . . . . .      Text     _TEXT
@data . . . . . . . . . . . . . .      Text     FLAT
@fardata? . . . . . . . . . . . .      Text     FLAT
@fardata . . . . . . . . . . . . .     Text     FLAT
@stack . . . . . . . . . . . . . .     Text     FLAT
Lf . . . . . . . . . . . . . . . .     Number   0000000Ah
_start . . . . . . . . . . . . . .     L Near   00000000 _TEXT Public
atodproc . . . . . . . . . . . . .     L Near   00000000 FLAT  External
atoiproc . . . . . . . . . . . . .     L Near   00000000 FLAT  External
cr . . . . . . . . . . . . . . . .     Number   0000000Dh
dtoaproc . . . . . . . . . . . . .     L Near   00000000 FLAT  External
inproc . . . . . . . . . . . . . .     L Near   00000000 FLAT  External
itoaproc . . . . . . . . . . . . .     L Near   00000000 FLAT  External
label1 . . . . . . . . . . . . . .     Byte     0000005D _DATA
number1 . . . . . . . . . . . . .      Dword    00000000 _DATA
number2 . . . . . . . . . . . . .      Dword    00000004 _DATA
outproc . . . . . . . . . . . . .      L Near   00000000 FLAT  External
prompt1 . . . . . . . . . . . . .      Byte     00000008 _DATA
prompt2 . . . . . . . . . . . . .      Byte     0000001E _DATA
string . . . . . . . . . . . . . .     Byte     00000035 _DATA
sum . . . . . . . . . . . . . . .      Byte     0000006A _DATA

          0 Warnings
          0 Errors
```

件IO.H的几行，这些以字母C标识的行显示它们来自一个包含文件。特别是，还会发现禁止大部分的IO.H文件显示的.NOLIST指示性语句，而.LIST则恢复列出余下的源文件。

汇编器的每条EQU语句显示哪些标识符是由8位十六进制数字表示的。例如，0000000D表示cr，0000000A表示Lf。

清单中每一栏的最左边显示每一个指示或指令与该段开始处的偏移量，用多个字节表示，下面这行:

```
00000000  00000000  number1  DWORD ?
```

由于这条语句是该数据段的开始，因此，这一行语句的偏移量为00000000。然后，汇编程序显示双字节的目标代码，它的值为00000000。因为DWORD预留了4个字节，因此，下一条语句偏移量为00000004。

```
00000004  00000000  number2  DWORD ?
```

接下来的4个字节保存的值为00000000。

现在已经有8个字节被预留了，因此接下来的偏移量为00000008。下面两句表示用BYTE指示性语句给prompt1与prompt2赋值。

```
00000008 45  6E  74  65  72      prompt1 BYTE   "Enter first"
         20  66  69  72  73
         74  20  6E  75  6D
         62  65  72  3A  20
         20  00
0000001E 45  6E  74  65  72      prompt2 BYTE   "Enter second"
         20  73  65  63  6F
         6E  64  20  6E  75
         6D  62  65  72  3A
         20  20  00
```

prompt2的偏移量是prompt1的偏移量00000008加上prompt1所占用的字节22（十六进制中的16），结果为0000001E。同样，因为第二个提示符占用字节数为23（十六进制中的17），因此下一条语句的偏移量为0000001E + 17 = 00000035。

下面这句:

```
00000035   00000028   [      string BYTE   40 DUP(?)
            00
           ]
```

表明为BYTE分配十六进制的28（十进制的40）个字节，每个字节初始化为00。这个数据段剩下的语句都没有什么新内容。

该代码段的汇编的清单用十六进制显示了每条指令的偏移量与目标代码，某些汇编器还显示宏语句的偏移量，也就是显示宏的第一条指令的地址。每条指令的机器代码的第一个字节称之为操作码。当程序执行的时候，通过操作码，80x86知道将要执行何种类型的操作，以及判断该指令是否还有更多字节。在目标代码中，单指令的长度为1 ~ 16位。

下面这句:

```
0000002E  A3  00000000 R      mov  number1, eax
```

表示这条指令起始的偏移量为0000002E，同时目标代码有5个字节，以操作码A3开头。操作码A3告诉80x86，复制EAX寄存器的内容到该指令接下来的4个字节单元中去。符号R表示这是个重定位（relocatable）地址，也就是说，在实际运行时，指令中用00000000代替该地址，因为链接器与装配器必须确定运行时number1的具体地址。图3-4显示在该程序的某次运行时，number1的地址为00404000。当然，程序每次运行，这个地址都不一定相同。

加法语句:

```
0000006B  03  05  00000004 R     add  eax,  number2
```

开始的偏移量为0000006B，其操作码为03，不同的操作码表示不同的加法指令。03操作码可用来做不同种形式的加法运算，CPU必须看后面的字节才能决定该做哪种操作。字节05告诉80x86，EAX寄存器是用来存放和（及一个源数据）的目的地址，另一个源数据存放在内存单元中接下来的4个字节。第9章将会更详细地讨论有关80x86的指令，以及它们是如何汇编的。

汇编清单文件的最后部分显示了所有在程序中用到的标识符。开始的几行列出了在IO.H中定义的宏的名字，尽管程序中有些宏并没有用到过。接下来列出的是段名和过程名，然后是保留字。这份清单包含了常见的标识符，比如Lf、number2以及_start等等。清单同时还显示了一些以@开头的标识符，它们给出了有关汇编过程的信息。剩下的一些符号是过程名，它们在IO.H中称之为宏，例如，atoiproc被称作为宏atod。

练习3.4

根据代码段3-2的汇编清单，回答下面的问题。

1. 字符串“The sum is”的ASCII代码是什么？
2. 在数据段中标号sum的偏移量是什么？
3. 如果下面的语句增加到数据段的结尾（只在.CODE前），那么在汇编清单中偏移量和值是

多少?

```
extra    DWORD 999
label2   BYTE "The End", Cr, Lf, 0
```

(提示: 可用ASCII/十六进制转换表来解此题)

4. 在示例程序中，第一到第三条语句共生成多少字节的目标代码（宏output、input和atod）?

3.5 常数操作数

这一节讨论在BYTE、WORD、DWORD指示性语句中用到的常数操作数的格式。无论是指示性语句还是指令性语句，常数的写法都是一样的，因此，其格式也适用于指令性语句。

数字操作数可用十进制、十六进制、二进制以及八进制表示。通常，汇编器默认为一个数为十进制数，除非这个数已经注明进制的下标或者被.RADIX标识符（本书没有用到过）修改了默认的进制。常用下标如下表所示。

下　标	基　数	进　制
H	16	十六进制
B	2	二进制
O或Q	8	八进制
没有	10	十进制

任何一种下标都可以写成大写或者小写字母。八进制并不经常使用，如果用到八进制的话，常用Q表示八进制，因为Q比O更容易区别，尽管这两种形式都可以。

语句:

```
mask   BYTE   01111101b
```

预留了一个字节的内存空间，并且初始化为7D。上面那句也可以用下面三条指示性语句表示:

```
mask   BYTE   7dh
mask   BYTE   125
mask   BYTE   175q
```

这是因为 $(1111101)_2 = (7D)_{16} = (125)_{10} = (175)_8$。进制的选择往往取决于要使用的常数。如果用一串8个单独的位来表示数，那么使用二进制就比较合适，比如逻辑运算。(将在第8章讨论)。

一条BYTE指示性语句为数据预留了一个或多个字节的存储空间。如果数据是数字的，那么可以被看作为有符号数或者无符号数。无符号数能表示的数值范围为0 ~ 255，有符号数能表示的数值范围为 - 128 ~ 127。尽管汇编器允许更大或者更小的数，但是，通常BYTE表达的数值范围为 - 128 ~ 255。下面的例子中的注释部分给出了预留的字节初始化的值。

```
byte1    BYTE    255      ; 数值为FF
byte2    BYTE    127      ; 数值为7F
byte3    BYTE    91       ; 数值为5B
byte4    BYTE    0        ; 数值为00
byte5    BYTE    -1       ; 数值为FF
```

```
byte6    BYTE    -91       ; 数值为A5
byte7    BYTE    -128      ; 数值为80
```

DWORD和WORD语句的情况类似。DWORD指示性语句预留了一个双字空间，由于8个字节能存储的有符号数的范围是－2 147 483 648～2 147 483 647，能存储的无符号数的范围是0～4 294 967 295，因此，操作数的范围要限制在－2 147 483 648～4 294 967 295之内。同样，WORD指示性语句中的操作数的范围限制在－32 768～65 535之内。下面的例子给出了预留的双字或字的值。

```
double1    DWORD   4294967295     ; 数值为FFFFFFFF
double2    DWORD   4294966296     ; 数值为FFFFFC18
double3    DWORD   0              ; 数值为00000000
double4    DWORD   -1             ; 数值为FFFFFFFF
double5    DWORD   -1000          ; 数值为FFFFFC18
double6    DWORD   -2147483648    ; 数值为80000000

word1      WORD    65535          ; 数值为FFFF
word2      WORD    32767          ; 数值为7FFF
word3      WORD    1000           ; 数值为03E8
word4      WORD    0              ; 数值为0000
word5      WORD    -1             ; 数值为FFFF
word6      WORD    -1000          ; 数值为FC18
word7      WORD    -32768         ; 数值为8000
```

在上述的例子中值得注意的地方是，不同的操作数存储的数值可能是相同的。例如，在WORD指示性语句中，操作数65535与－1的得到的双字数都是FFFF。这个数值既可以被看作是无符号数的65535，也可以被看作是有符号数的－1，这要取决于所在的上下文环境。

就上述情况而言，字与双字的字节是反向存储的。举例来说，上面的word6的初始值实际上是18FC。本书将注重于逻辑实际值，而不是它们存储的方式。

BYTE指示性语句允许操作数为单个字符，或者是由很多字符组成的字符串。单引号（'）与双引号（"）都可以用来表示字符或者是字符串。它们必须成对出现，而不能在左边用单引号而右边使用双引号。如果字符串的划界符是单引号，那么字符串中可以包含双引号字符；同样，如果字符串的划界符是双引号，那么字符串中可以包含单引号字符。这样，字符串就能包含这些特殊的字符了。除非特别说明，否则本书将用单引号定界单个字符，而用双引号定界一个字符串。

下面的几个BYTE指示性语句都是允许的。

```
char1      BYTE    'm'        ; 数值为6D
char2      BYTE    6dh        ; 数值为6D
string1    BYTE    "Joe"      ; 数值为4A 6F 65
string2    BYTE    "Joe's"    ; 数值为4A 6F 65 27 73
```

如果要保存字母m，而不是数字$(6D)_{16}$，那么不需要查询ASCII码，直接将6dh输入到char2中，因为汇编器内置了ASCII码表。值得注意的是定界符，无论是字中的单引号定界符，还是字符串中的双引号定界符，它们本身都不被存储。

汇编器可限制DWORD或者WORD指示性语句中字符操作数的用法。但是，这样做没什么意义。

在上面的BYTE指示性语句的例子中，多个操作数可用逗号分隔。同样，DOWRD与WORD也允许多个操作数。指示性语句

```
words     WORD  10, 20, 30, 40
```

预留了四个字的存储空间，并初始化为000A、0014、001E与0028。DUP运算符可用来生成多个字节或字，这些字或字节可以是确定的值，也可以是没有初始化的值。DUP的作用是约束预留存储空间的BYTE、DWORD、WORD以及其他指示性语句。指示性语句

```
DblArray   DWORD   100   DUP   (999)
```

预留了100个双字空间，每个都被初始化成000003E7。对于初始化数组元素，这是一种有效的方法。如果一个字符串由50个乘法符号组成，那么可以使用

```
Stars   BYTE   50   DUP   ('*')
```

这样一条语句来实现。如果想要25个乘法符号，它们中间用空格相隔，这样就预留49个字节的存储单元，并且按要求指定了初始化值。那么可以使用

```
starsAndSpaces   BYTE   24   DUP   ("*"), '*'
```

来实现。

BYTE、DWORD、WORD以及其他的语句中的操作数可以是一个包含算术符或者其他运算符的表达式。在汇编时，而不是运行时，汇编器会对这个表达式求值，然后将结果用于汇编。通常情况下，没有必要用一个表达式来代替常数。但是，有时候使用表达式可能使代码更清楚。下面的几条语句是相等的，每条语句都保存了一个字，其初始化值为十六进制的0090。

```
gross     WORD   144
gross     WORD   12*12
gross     WORD   10*15-7 + 1
```

在每条BYTE、DWORD或者WORD语句中定义的标识符都与长度相关。汇编器会注意其长度，并检查以确保指令中用到的标识符是合适的。例如，如果

```
char   BYTE   'x'
```

用在

```
mov   ax,   char
```

中，汇编器就会生成错误信息，因为寄存器AX是一个字长，而char的存储空间却是一个单字节。

微软的汇编器认可为预留存储空间使用的其他指示性语句。其中QWORD用来预定4个字的空间，TBYTE用来预定10字节的整数存储空间，REAL4用来预定4个字节的浮点数存储空间，REAL8、REAL10分别用来预定8个及10个字节的浮点数存储空间。此外，还有一些指示性语句可区分有符号字节、字以及无符号双字。不过，很少使用这些指示性语句。

练习3.5

请给出汇编器根据下面指示性语句生成的初始值。

```
1. byte1        BYTE      10110111b
2. byte2        BYTE      33q
3. byte3        BYTE      OB7h
4. byte4        BYTE      253
5. byte5        BYTE      108
6. byte6        BYTE      -73
7. byte7        BYTE      'D'
8. byte8        BYTE      'd'
9. byte9        BYTE      "John's program"
10. byte10      BYTE      5 DUP("<>")
11. byte11      BYTE      61 + 1
12. byte12      BYTE      'c' - 1
13. dword1      DWORD     1000000
14. dword2      DWORD     1000000b
15. dword3      DWORD     1000000h
16. dword4      DWORD     1000000q
17. dword5      DWORD     -1000000
18. dword6      DWORD     -2
19. dword7      DWORD     -10
20. dword8      DWORD     23B8C9A5h
21. dword9      DWORD     0, 1, 2, 3
22. dword10     DWORD     5 DUP(0)
23. word1       WORD      1010001001011001b
24. word2       WORD      2274q
25. word3       WORD      2274h
26. word4       WORD      0ffffh
27. word5       WORD      5000
28. word6       WORD      -5000
29. word7       WORD      -5, -4, -3, -2, -1
30. word8       WORD      8 DUP(1)
31. word9       WORD      6 DUP(-999)
32. word10      WORD      100/2
```

3.6 指令中的操作数

指令的操作数有三种基本的类型：常数、指定的CPU寄存器以及存储器操作数。定位内存地址有多种方法，本节讨论其中较为简单的两种，更多复杂的方法将在本书后面需要的时候介绍。

许多指令有两个操作数。通常，第一个操作数给出了操作的目的地址，虽然它还可以是指定的源数据之一。第二个操作数为该操作提供源数据（或一个数据源），但绝对不会是目的

地址。例如:

```
mov   al,   '/'
```

该句执行的时候，字节2F（斜线“/”的ASCII码值)会被放入AL寄存器，覆盖该寄存器以前的内容。第二个操作数‘/’表明这是个常量数据源。又例如:

```
add   eax,   number1
```

执行这条语句时，声明为双字节数的number1与EAX寄存器里面的值相加，其结果保存在EAX寄存器里面，并且覆盖了EAX寄存器以前的内容。因此，第一个操作数EAX既是一个双字数据源，又是和的目的地址，而number1则是双字加法的另外一个加数。

表3-1列出了Intel80x86微处理器中的寻址方式，并给出了每种寻址方式下，数据所在的地址。内存地址可以通过几种方式计算，表3-2列出了最常用的两种。

表3-1 80x86寻址方式

方　式	数据所在地址
立即数	指令操作数就是该数据
寄存器	数据放在某一寄存器中
存储器	数据放在某一内存地址中

表3-2 两种80x86内存寻址方式

存储寻址方式	数据所在地址
直接寻址	在一内存地址中，其偏移量为指令中的操作数
寄存器间接寻址	寄存器的值作为数据的地址

对于*立即数寻址方式*（immediate mode）的操作数，在运行之前，数据就已经写在指令里面了。通常情况下，如果是常数[⊖]，通常数据是由汇编器放置的。不过，根据数据值是在哪一阶段确定的，数据也能由链接器或者装配器插入。程序员编写指令的时候可以直接用实际的值，也可以用标识符来代替一个常数。在*寄存器寻址方式*（register mode）下，需要的数据在寄存器内。要得到一个寄存器寻址方式的操作数，程序员只需要简单地写上寄存器的名字。寄存器寻址方式的操作数还能指定一个寄存器作为目的地址，但是，立即寻址方式的操作数则不能作为目的地址。

下面的例子中，第一个操作数都是寄存器寻址方式，而第二个操作数都是立即寻址方式。在汇编程序的清单文件中，目标代码以注释形式显示。如:

```
mov   al,   '/'   ;   B0   2F
```

汇编器将指令中斜线“/”的ASCII码2F作为第二个操作数，并且由汇编器放在寄存器al中。如:

```
add   eax,   135   ;   05   00000087
```

该句中135的双字节长度的二进制补码汇编时放入指令的最后4个字节单元中。

任何一种内存寻址方式的操作数要么在内存中指定使用数据，要么在内存中指定一个目

⊖ 可以用自修改方式编码，也就是说指令在执行时代码发生改变，这是一种非常差的编程经验。

的地址。直接寻址方式（direct mode）操作数在指令中是一个32位地址。通常情况下，在数据段中程序员用一个标识符表示相应的单字节、双字和字；或者在代码段中，用标识符表示指令。由于标识符的对应的地址需要重新定位，因此，汇编清单上显示汇编时的地址以后可能会调整。

```
add   eax,   number2   ;   05   00000004
```

这条语句来自代码段3-1的示例，第一个操作数是寄存器寻址方式，第二个操作数为直接寻址方式，该存储器操作数地址编码为32位地址00000004，也就是number2在该数据段中的偏移量。

```
add   eax,   [edx]   ;   03   02
```

这句中第一个操作数为寄存器寻址方式，第二个操作数为寄存器间接寻址方式（register indirect mode）。本书将在后面讨论汇编器是如何得到该指令的目标代码03和02的。但是，值得注意的是，02没有足够的空间存储一个32位地址，事实上，02表示用寄存器EDX中的值作为地址来定位一个内存中的双字节数，把这个数和已经在寄存器EAX中的数相加。换句话说，第二个操作数不在EDX寄存器中，但是它的地址在EDX寄存器中。中括号（[]）表示MASM6.11的间接寻址方式，图3-6说明了在这个例子中寄存器是如何进行间接寻址方式的。

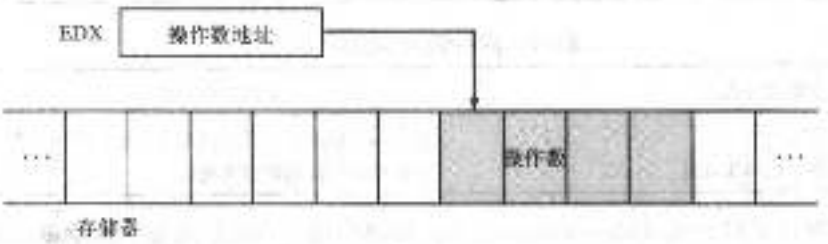


图3-6 寄存器间接寻址

任何一个普通的寄存器，比如EAX、EBX、ECX、EDX或者索引寄存器ESI、EDI都能作为寄存器间接寻址方式来使用。基址指针寄存器EBP也能用于该方式，但是地址是在堆栈中使用，而不是在数据段使用。尽管栈指针ESP在某些特定的环境中也能用作寄存器间接寻址方式，不过没有必要这样做。

在寄存器间接寻址方式中，从高级语言角度来看，寄存器的使用就像是指针变量。寄存器存放的是指令中数据的地址，而不是数据本身。如果内存地址的长度不确定，那么必须使用PTR操作符来告诉汇编器其长度。

```
mov   [ebx],   0
```

上句中，汇编器会给出错误信息，因为它没有说明目的地址是单字节，还是双字节，或者是双字。如果它是单字节，可以使用下句：

```
mov   BYTE   PTR   [ebx],   0
```

如果目的地址是字，或者双字，可以分别使用WORD PTR或者DWORD PTR。在下句中：

```
add   eax,   [edx]
```

这里没有必要使用DWORD PTR [edx]，因为目的地址EAX是双字，所以汇编器假定它是

双字。

某些指令没有操作数，还有一些指令只有一个操作数。有时，一条没有操作数的指令不需要任何数据；有时只有一个操作数的指令仅仅需要一个值。此外，有时一个或者多个操作数的地址是隐含的，没有写出来。比如80x86的乘法指令为mul，它可以写成：

```
mul   bh
```

在该句中，只给出了一个操作数，其他乘数通常保存在AL寄存器中。（下章将会详细介绍这条指令）

练习3.6

在下面的指令中，给出每个操作数的寻址方式，假定这些指令是在包含如下代码的程序中。

```
cr      EQU     0dh
.DATA
value   DWORD   ?
char    BYTE    ?
```

```
1. mov value, 100
2. mov ecx, value
3. mov ah, cr
4. mov eax, [esi]
5. mov [ebx], ecx
6. mov char, '*'
7. add value, 1
8. add WORD PTR [ecx], 10
```

3.7 使用IO.H中宏的输入/输出

一个有用的程序，必须能够输入输出数据，操作系统提供了一些例程帮助完成这些任务。一个典型的输入例程应该能接收从键盘输入的一个字符，然后返回该字符的ASCII码，并存放到一个寄存器中。而一个输出例程则能够在终端显示一串字符，直到碰到某些结束符，比如$符号。

高级语言通常提供数字、字符以及字符串等数据的输入或输出。在高级语言中，数字的输入例程能接收表示一个数字的字符串代码，然后把这些字符转换成二进制补码或者浮点形式，并且把数值存入与某一变量名关联的内存地址中。相反，高级语言中数字的输出例程是把某一内存地址中的二进制补码或者浮点数转换成一个表示该数的字符串，然后输出这个字符串。通常，操作系统并不提供这些服务。因此，汇编语言程序员必须自己编写代码。

IO.H文件提供了一组宏定义，用来实现输入、输出以及简单明了的数值转换。每个宏看起来像一条80x86指令，但实际上，宏展开为多条指令，其中包括调用一个完成大多数工作的外部过程。这些外部过程的源代码放在IO.ASM文件中，相应的汇编文件为IO.OBJ。在本书稍后的章节会考察IO.ASM文件中的代码。

表3-3列出了在IO.H文件中定义的宏，并简要地描述了它们，下面会对它们做更多的解释。在随后的章节里，一些程序将用到这些宏。

表3-3 IO.H中的宏

宏 名	参 数	功 能	受影响的标志位
dtoa	目的操作数，源操作数	把源操作数（寄存器或存储器）中的双字节数转换成一个11个字节长的ASCII码，并存入目的操作数	无
atod	源操作数	扫描源操作数中以“+”或“-”号起始后字符串中的数字，将这些数字字符作为一个整数，将这个整数的2进制补码形式的数存入EAX寄存器。非数字字符存入ESI寄存器。如果输入错，0存入EAX寄存器。如果该数超出-2 147 483 647到2 147 483 647的范围，那么输入错误	如果输入错误，OF置1，否则OF置0；其他标志的值根据结果存入EAX寄存器
itoa	目的操作数，源操作数	把源操作数（寄存器或存储器）中的单字节数转换成一个6字节长的ASCII码，并存入目的操作数	无
atoi	源操作数	与atod类似，只是转换的结果存入AX寄存器，并且该数允许的范围是-32 768到32 767	与atod相同
output	源操作数	输出源操作数中的字符串，该字符串必须以空字符结束	无
input	目的操作数，长度	输入一个长度为length的字符串，并将该字符串存入目的操作数	无

输出宏用于将字符串输出到显示器上。它的源操作数是指数据段的一个地址，通常是以BYTE命名的。字符串从这个地址开始显示，直到一个遇到空字符才中止显示，空字符中止输出。源字符串必须包括要输出显示的字符的ASCII码值，这一点非常重要。大部分字符是可以打印的，如果回车、换行或者一些其他的特殊字符也能打印的话，那就更有意义了。如果要使用输出宏来显示非ASCII码数据，比如2进制补码形式的双字节整型数据，那么其结果将会出乎意料。

宏output不会修改任何寄存器的内容，包括标志寄存器。

宏input用于从键盘输入一个字符串。它包含两个参数，即目标操作数和长度。目标操作数是指数据段中的字节串；长度操作数是指到该字符串的字符个数。在输入回车后，考虑到操作系统要加上回车与换行符，因此，目标字符串比实际输入的字符串至少要多两个字节的长度。输入宏会用空字节来填补回车符，这样，一个以空字符结尾的字符串就保存在目的地址中。

宏input仅仅修改了内存中指定的目的地址的值，而不会修改任何寄存器的内容，包括标志寄存器。

宏dtoa（double to ASCII），顾名思义，它的作用是把双字节数转换成ASCII码。它把一个包含2进制补码的双字长度的整型数转换成一个相同的十进制整数，该整数由11个ASCII字符组成。源操作数通常是寄存器或者存储器操作数。目的地址通常是数据段中由BYTE指示性语句保留的11个字节的数据区。如果产生的数字位数小于11，那么前面用空格填充。如果是一个负数，那么负号将放在数字之前。因为2位补码的十进制数据范围为是-2 147 483 648～2 147 483 647。因此，不用担心生成的位数太多，11位字节足够长了。正数至少有一个空格。

宏dtoa仅仅修改了目的地址的ASCII代码的11个字节的内存区，没有修改任何寄存器的内容，包括标志寄存器。

宏atod（ASCII to double）在很多方面都和宏dtoa相反。它只有一个操作数，就是存储的ASCII码组成的字符串的地址。atod检查表示十进制数的字符的内存域，如果发现由数字字符组成的十进制数是在－2 147 483 648～2 147 483 647范围内，那么它将把字符串转换成2进制补码形式的数，并把这个数保存在EAX寄存器中。

源字符串可以用多个空格开头，atod将跳过空格。然后可能出现ASCII码的“－”或“+”符号。如果一个数字前面没有任何符号，那么该数被认为是一个正数，接下来是从0～9的任一数字。如果遇到的是数字码，atod将继续检查下一个码，直到下一个码不是数字码为止。

使用宏atod可能会出现一些问题。如果在负号与第一个数字之间存在一个空格，或者数据源字符串是以一个非数字码的字符开头，宏将找不到数字码；此外，如果目标数太大，不能存储在一个2进制补码的双字空间内；这样，atod就可能检查不到一个数字码。上面任何一种情况发生，都会将00000000放入EAX寄存器，并把溢出标志位OF置为1。

如果宏atod能成功转换ASCII字符串，那么溢出标志位OF置0。而SF、ZF以及PF标志位则根据EAX内的值视情况而定：

- 如果是负数，SF置1；否则置0
- 如果是0，ZF置1；不是0，则ZF置0
- PF则由EAX中的数的奇偶性确定

此外，CF置0，DF不变，除了EAX寄存器与标志寄存器外，别的寄存器都不改变。

宏atod经常紧跟在宏input后，宏input生成一个尾部为空字符的ASCII码字符串。当宏atod检查这个字符串时，尾部的空字符可以用来终止对该字符串的检查。如果宏atod检查的字符串不是input宏产生的字符串时，程序员必须确保该字符串是以非数字字符结尾的，可以终止对字符串的检查。

宏atoi（ASCII to integer）与宏itoa（integer to ASCII）是宏atod与dtoa的字长版本，处理字长的参数。宏atoi检查字符串，将其转换成相应字长的2进制补码，并放在AX寄存器中。宏itoa则将一个字长中的2进制补码数转换为一个十进制数，该数是由6个字符组成的字符串。如果数值范围是在－32 768～32 767之间，这两个宏都是有用的。

练习3.7

1. 为什么宏dtoa不设计为可以生成很多位ASCII编码的宏，为什么数字11对宏特别重要。
2. 为什么宏itoa不设计为可以生成很多位ASCII编码的宏，为什么数字6对宏特别重要。
3. 假定

数据段如下定义：

```
response1 BYTE 10 DUP(?)
```

代码段有宏

```
input response1, 10
```

(a) 如果运行时，输入－578，那么数据段将存储什么ASCII编码？

(b) 假如input宏后紧跟

```
atod response1
```

那么EAX寄存器内是什么，OF、SF以及ZF标志位的值是多少？

4. 假定

数据段如下定义:

```
response2 BYTE 10 DUP(?)
```

代码段有宏

```
input response2,10
```

(a) 如果运行时，输入123456，那么数据段将存储什么ASCII编码?

(b) 假如input宏后紧跟

```
atoi response2
```

那么AX寄存器内是什么，OF、SF以及ZF标志位的值是多少?

5. 假设程序包含如下定义的数据段

```
value1      DWORD ?
result1     BYTE  11 DUP(?)
            BYTE  'sum', 0dh, 0ah, 0
```

代码段有宏

```
dtoa result1,value1
```

(a) 如果运行时，value1指向的双字数是FFFFFF1A，dtoa宏调用后，result1的值是多少?

(b) 假如dtoa宏后紧跟

```
output result1
```

那么显示器将显示什么?

6. 假设程序中包含如下数据段定义

```
result2  BYTE  6  DUP(?)
         BYTE     'total', 0dh, 0ah, 0
```

代码段有宏

```
itoa      result2,  BX
```

(a) 假定运行时BX寄存器值为1AFF，那么在itoa宏调用后，results存储的值是多少?

(b) 假如if itoa宏后紧跟

```
output result5
```

那么显示器将显示什么?

本章小结

第3章介绍了使用微软MASM汇编器80x86汇编语言。

一条汇编注释语句总是以“;”开始，汇编语句有如下格式:

```
名字      助记符      操作数      ; 注释
```

其中一些项是可选的。

汇编语句有3中类型:

- 指令性语句——每一条语句对应于一条CPU指令
- 指示性语句——告诉汇编器需要做什么
- 宏——展开为其他语句

汇编语言程序主要包括数据段和代码段，其中在数据段中定义变量，代码段中包含运行时执行的语句。为了获得可执行程序，编程人员必须使用汇编器将程序汇编为目标代码，使用链接器将目标代码转换为可执行文件，可执行文件可以使用类似Windbg的调试器进行跟踪。

BYTE、DWORD以及WORD指示性语句用于在存储区域预留字节、双字或者字，也可用于按指定的值进行初始化。

指令中的操作数有三种模式:

- 立即数——指令中含有数据
- 寄存器——数据在寄存器中
- 存储器——数据在存储区域

得到存储器操作数有几种方式，其中的两种方式如下:

- 直接寻址——地址在指令中给出
- 寄存器间接寻址——数据的地址在寄存器中

在文件IO.H中定义了输入输出宏，这些宏调用了一些汇编到了IO.OBJ中的过程，有以下的宏:

- output——在显示器中输出一字符串
- input——从键盘输入一字符串
- atod——转换一字符串为一双字长的2进制补码数
- dtoa——转换一双字长的2进制补码数为一字符串
- atoi——转换一字符串为一字长的2进制补码数
- itoa——转换一字长的2进制补码数为一字符串

第4章 基本指令

本章介绍了将数据从一个位置复制到另外一个位置的指令和整数运算指令。本章尤其强调各种指令允许什么类型的操作数，介绍了时间效率和空间效率的概念。最后，本章给出了在一些需要的操作数类型受限制而不能使用时，可以实现同等操作的一些方法。通过本章的学习，可以了解到如何在存储器和CPU寄存器之间复制数据，以及如何在两个寄存器之间传送数据。而且，还将了解到如何使用80x86的加、减、乘、除指令，以及这些指令的执行是如何影响标志位的。

4.1 复制数据指令

大多数计算机程序都能将数据从一个位置复制到另外一个位置。对于80x86机器语言，复制工作由mov（move）指令完成，每条mov指令格式如下:

```
mov 目的操作数，源操作数
```

可以从源操作数地址把一个字节、字或双字复制到目的操作数地址，存储在源地址中的值不会改变。目的地址必须要求与源地址大小一致。一条mov指令与一条高级语言中的简单赋值语句十分相似。例如，Pascal和Ada赋值语句:

```
Count:= Number
```

对应的汇编语言指令如下

```
mov Count, ecx; Count:= Number
```

这里假设ECX寄存器中的值是Number，而且Count指向内存中的一个双字。但是，高级语言的赋值语句与mov指令不能因此类推。例如，赋值语句

```
Count:= 3*Number + 1
```

就不能用一条简单的mov指令来实现。在运算结果的值被复制到目的地址前，需要多条指令来计算等式右边的表达式。

80x86体系结构的局限之一是：并不是所有的源操作数和目的操作数的任意组合都是合理的。特别是，源操作数和目的操作数不能同时在存储器中。指令

```
mov  Count, Number     ; 不合法的两个操作数
```

如果Count和Number都指向内存地址，那么这条指令是非法指令。

所有80x86 mov指令使用同样的助记符来编码。通过察看操作数和助记符，汇编器选择正确的操作码和其他字节的机器码。

表4-1列出了mov指令，这些指令的源操作数是立即数，目的操作数在寄存器中，并且列出了80386、80486和Pentium处理器执行每一条指令所需的时钟周期数。虽然只有少数程序用汇编语言编写，但是汇编语言却能编写高效的过程。时间效率通常由执行程序的时间长度来衡量，

而这点取决于处理器执行每一条指令所需要的时钟周期数。空间效率涉及程序代码的大小，例如，如果程序必须存储在ROM中，小的可执行文件十分重要。表4-1列出了每条指令的字节数。

表4-1 立即数-寄存器mov指令

目的操作数	源操作数	时钟周期数			字节数	操作码
		386	486	Pentium		
8位寄存器	字节立即数	2	1	1	2	
AL						B0
CL						B1
DL						B2
BL						B3
AH						B4
CH						B5
DH						B6
BH						B7
16位寄存器	字节立即数	2	1	1	3（包括前缀字节）	
AX						B8
CX						B9
DX						BA
BX						BB
SP						BC
BP						BD
SI						BE
DI						BF
32位寄存器	字节立即数	2	1	1	5	
EAX						B8
ECX						B9
EDX						BA
EBX						BB
ESP						BC
EBP						BD
ESI						BE
EDI						BF

执行一条指令所需的时间长度由时钟周期数来衡量。为了确定实际的执行时间，必须知道处理器实际的时钟频率。在早期的IBM PC中，Intel 8088处理器的时钟频率为4.77MHz，也就是说一秒钟有4 770 000周期。现在，许多80x86个人计算机可以高于200MHz工作，也就是说一秒钟有200 000 000周期。早期的IBM PC每个时钟周期是210ns（ns＝纳秒，1/1 000 000 000s），而对于200MHz的计算机来说，每个时钟周期是5ns。微型计算机速度的提高不仅仅是因为更高的时钟频率，而且对于处理器系列的新成员，执行同样的指令需要更少的时钟周期。

对于80386、80486和Pentium等处理器，由于目标代码相同，所以每条指令所占用的字节数也相同，这一点同样适用于8086、8088、80186和80286处理器。但是，8086、8088、80186和80286处理器不支持32位寄存器，因此，表4-1中对于32位寄存器的指令不适用。

对于mov指令，将字或双字立即数复制到寄存器，其操作码都是一样的。80x86处理器为每一个活动段提供一个*段描述符*（segment descriptor）。该描述符的一位决定操作数是16位还

是32位（默认为32位），对于本书中用到的汇编指示性语句和连接器的选项，该位置为1表示32位操作数。因此，例如，B8操作码的意思是复制操作码后的双字立即数到EAX，而不是一个单字立即数到AX。如果编写16位指令：

```
mov ax, 0
```

那么，汇编器将插入前缀字节（prefix byte）66到目标代码B8 0000前，因此，实际上生成的代码为66 B8 0000。通常，前缀字节66告诉汇编器，将前缀后的一条指令从默认的操作数长度（32位或16位）转换为可选的长度（32位或16位）。

正如第2章所讨论的，指令有时会影响EFLAGS寄存器中不同的标志位。通常，一条指令可能有如下3种影响：

- 标志位不改变
- 根据指令的执行结果对特定的标志位赋值
- 某些标志位可能被改变，但是它们的设置无法预测

所有mov指令属于第一种情况。没有一条mov指令会改变任何标志位。

表4-2列出了源操作数为立即数，目标操作数在存储器中的mov指令。80486和Pentium处理器执行这些指令需要一个时钟周期，而80386需要两个时钟周期。与早期8088相比，有一定提高，因为执行同样一条指令，8088至少需要14个时钟周期。

表4-2 立即数-存储器mov指令

目的操作数	源操作数	时钟周期数			字节数	操作码
		386	486	Pentium		
存储器字节	字节立即数	2	1	1		C6
直接					7	
寄存器间接					3	
其他					3～8	
存储器字	字立即数	2	1	1		C7
直接					8	
寄存器间接					4	
其他					4～9	
存储器双字	双字立即数	2	1	1		C7
直接					10	
寄存器间接					6	
其他					6～11	

存储器操作数所占用的字节数由操作数的类型决定。一个直接操作数必须用32位地址编码，共四个字节。寄存器间接操作数由第二个目标代码的三位表示。以后章节将考察其他类型的存储器操作数。16位操作数还需要前缀字节66，从技术角度来看，它不是指令的一部分，所以没在表4-1中列出。

表4-2中列出了操作码C6和C7，用于立即数到存储器的复制，也可用来实现从立即数到寄存器的复制。但是，这些形式需要额外一个字节的目标代码，通常，汇编器选择表4-1中给出的较短的形式。

表4-3列出了其他的80x86 mov指令。这张表引入了一些新的术语。32位寄存器（Register 32）是指32位寄存器EAX、EBX、ECX、EDX、EBP、ESI、EDI或ESP中的一个寄存器；同样，

16位寄存器（Register 16）是指16位寄存器AX、BX、CX、DX、SP、BP、SI或DI中的一个寄存器；同时8位寄存器（Register 8）是指8位寄存器AL、AH、BL、BH、CL、CH、DL或DH中的一个寄存器。

注意：相同的操作码有时用于不同的指令，例如，从一个8位寄存器到8位寄存器的数据传送和一个存储器字节数到一个8位寄存器的数据传送，它们有相同的操作码。在这样的情况下，指令中的第二个字节不仅决定了目的寄存器，而且决定了源寄存器的编码或指出了源存储字节的模式。该字节的结构将在第9章详细讨论。

用两条不同的指令都可以复制一个存储器操作数到累加器，例如：操作码A1或8B都可以用来对指令 mov eax, Number编码。两者的区别在于8B指令操作码还可以用来复制双字到其他的目标寄存器，而A1操作码只能把累加器作为目标寄存器。汇编器通常使用A1，因为它比8B少一个字节。

注意，尤其是以前的处理器，指令访问存储器的速度比访问寄存器中数据的速度要慢。还需注意的是，访问存储器的指令可能需要的时钟周期比表4-3列出的更多。原因之一可能是存储器的响应速度不够快，在这种等待的状态下，浪费掉的时钟周期也被算在其内，直到存储器响应。即使是高速存储器，当访问一个字或双字时，如果字或双字在存储器中不是连续存储的，就是说它存储的地址分别是2或者4的倍数，就需要额外的时钟周期。编写程序时应该尽可能将常用的数据存储在寄存器中。

表4-3 额外的mov指令

目的操作数	源操作数	时钟周期数			字节数	操作码
		386	486	Pentium		
8位寄存器	8位寄存器	2	1	1	2	8A
16位寄存器	16位寄存器	2	1	1	2	8B
32位寄存器	32位寄存器	2	1	1	2	8B
8位寄存器	存储器字节	4	1	1	2～7	8A
16位寄存器	存储器字	4	1	1	2～7	8B
32位寄存器	存储器双字	4	1	1	2～7	8B
AL	直接存储器字节	4	1	1	5	A0
AX	直接字	4	1	1	5	A1
EAX	直接双字	4	1	1	5	A1
存储器字节	8位寄存器	2	1	1	2～7	88
存储器字	16位寄存器	2	1	1	2～7	89
存储器双字	32位寄存器	2	1	1	2～7	89
直接存储器字节	AL	2	1	1	5	A2
直接字	AX	2	1	1	5	A3
直接双字	EAX	2	1	1	5	A3
段寄存器	16位寄存器	2	3	1	2	8E
16位寄存器	段寄存器	2	3	1	2	8C
段寄存器	存储器字	2	3+	2+	2～7	8E
存储器字	段寄存器	2	3	1	2～7	8C

在系统编程中，对特定寄存器进行读写的mov指令，本书不做讨论。首先考察表4-1至表4-3中全部的mov指令，并不能使用它们把任何源数据复制到目标位置。许多看上去合理的组合不一定可用，其中包括：

- 源操作数和目的操作数都在存储器中的数据传送
- 源为立即数，目的操作数为段寄存器
- 源操作数和目的操作数都在标志寄存器中的数据传送
- 任何向指令指针寄存器的数据传送
- 从一个段寄存器到另一个段寄存器的数据传送
- 操作数长度不一致的数据传送
- 一次数据传送多个对象

有时，可能需要做上述的某些操作，下面将讨论如何具体实现。

尽管没有一个mov指令从存储器到存储器间进行数据复制，但是，可以使用两条立即数/寄存器指令来实现这样的工作。例如，对于Count和Number指向的双字长数，如下指令不合法:

```
mov Count, Number                ; 非法是因为两个均为存储器操作数
```

可以替换为

```
mov eax, Number ; Count:= Number
mov Count, eax
```

每条语句都用了EAX累加器和一个直接存储器操作数。除了EAX外，也可以使用其他寄存器。但是，使用累加器的每条指令需要5个字节，而使用其他相应寄存器的指令需要6个字节，从空间效率来考虑，选择EAX更好。

为了将一个立即数复制到一个段寄存器，首先，将立即数复制到16位寄存器，然后，将16位寄存器复制到段寄存器。当使用段存储模式编码时，这样的复制顺序用来初始化数据段寄存器DS。

尽管标志寄存器和指令指针寄存器都不能由mov指令来设置，但其他的指令却可以改变它们的值。当取新指令后，指令指针寄存器会更新；当jump、Call和返回指令执行后，指令指针寄存器会自动改变。通过各种指令，可设置每一个标志位的值，有时，可能需要将标志寄存器的所有位赋特定的值，稍后将介绍其实现的一些方法。

将一个数据的长度从字转换为字节是合法的，例如，将一个字复制到16位寄存器，然后只将高位字节或低位字节复制到目的地址。按照另一种方法，可以将一个16位寄存器的高位字节和低位字节组合成一个字，并将其复制到某一目的地址。这些方法有时很有用，其他的方法将在第8章中继续讨论。有时，需要将一个字节的数据扩展成一个字长或双字长的数据，或将一个字长的数据扩展成4个字节的数据，这样的指令将在4.4节中讨论。

假设源地址和目的地址的声明如下:

```
source DWORD 4 DUP(?)
dest DWORD 4 DUP(?)
```

而且希望将全部的四个双字长数据从源地址复制到目的地址，一种方法是使用以下四组指令

```
mov eax, source     ; 复制第一个双字
mov dest, eax
mov eax, source+4   ; 复制第二个双字
mov dest + 4, eax
mov eax, source+8   ; 复制第三个字节
```

```
mov dest + 8, eax
mov eax, source+12 ; 复制第四个字节
mov dest + 12, eax
```

source + 4的地址指的是source地址起始后的第四个字节（一个双字长）的地址。由于这四个双字长数在source地址起始后的存储器单元中连续存储，因此，source + 4也就指向了第二个双字长数。当需要复制40或400个双字长数据时，用这种编码方法，空间效率显然不高。第5章将介绍如何设置一个循环去复制多个对象，第7章将讨论如何使用字符串操作来复制大的数据块。

80x86有一个非常有用的xchg指令，它可将两个不同地址的数据进行交换，只需要一条简单的指令完成，但是高级语言却需要三条指令。假设交换Value1和Value2，在高级语言的设计中，可使用如下语句完成:

```
Temp:= Value1;        {交换Value1和Value2的数据}
Value1:= Value2;
Value2:= Temp;
```

假定Value1的值存在寄存器EAX中，而Value2的值存在寄存器EBX中，那么上面的数据交换可以写成如下形式:

```
xchg   eax, ebx   ; 交换Value1和Value2的数据
```

如果不使用xchg指令，还可以用如下代码

```
mov ecx, eax      ; 交换Value1和Value2的数据
mov eax, ebx
mov ebx, ecx
```

每条mov指令需要一个时钟周期和两个字节，因此，上述代码总共需要三个时钟周期和六个字节的编码。但是，xchg指令只需要一个字节和两个时钟周期（在Pentium上）。另外，写一条指令远比写三条指令容易得多，而且代码更容易理解。

表4-4列出了xchg指令的各种形式。16位和32位指令是一样的，区别仅在于前缀字节，表4-4将二者一起列出。尽管表中没有列出，如果第二个操作数是寄存器，那么，第一个操作数可以是存储器操作数；汇编器能有效地交换操作数的顺序，并可使用表中所列出的形式。

表4-4 xchg指令

操作数1	操作数2	时钟周期数			字节数	操作码
		386	486	Pentium		
8位寄存器	8位寄存器	3	3	3	2	86
8位寄存器	存储器字节	5	5	3	2～7	86
EAX/AX	32/16位寄存器	3	3	2	1	
	ECX/CX					91
	EDX/DX					92
	EBX/BX					93
	ESP/SP					94
	EBP/BP					95
	ESI/SI					96
	EDI/DI					97
32/16位寄存器	32/16位寄存器	3	3	3	2	87
32/16位寄存器	32/16位存储器	5	5	3	2～7	87

在计算机体系结构中，xchg指令再次说明了累加器有时有着特定的作用。有一些特殊的指令，它们用于其他寄存器与累加器之间的数据交换，但其速度比相应的通用寄存器与寄存器之间的数据交换要快，而且所用字节数少。这些指令也可以把累加器作为第二个操作数。

注意，不能使用xchg指令交换两个存储器操作数。通常，80x86 指令不允许两个存储器操作数。

类似于mov指令，xchg指令不改变任何状态标志位，也就是说，在xchg指令执行后，EFLAG寄存器的内容与指令执行前相同。

练习4.1

1. 对于如下问题的每部分，在给定的mov指令执行时，假设已有执行“执行前”的值，给出该指令执行“执行后”所需的值。

执 行 前	指 令	执 行 后
(a) BX: FF 75		
CX: 01 A2	`mov bx, cx`	BX, CX
(b) AX: 01 A2	`mov ax, 100`	AX
(c) EDX: FF 75 4C 2E		
值: DWORD - 1	`mov edx, Value`	EDX, Value
(d) AX: 01 4B	`mov ah, 0`	AX
(e) AL: 64	`mov al, -1`	AL
(f) EBX: 00 00 3A 4C		
值: DWORD?	`mov Value, ebx`	EBX, Value
(g) ECX: 00 00 00 00	`mov ecx, 128`	ECX

2. 给出练习题1中每条指令的opcode。
3. 对于如下问题的每部分，在给定的xchg指令执行时，假设已有执行“执行前”的值，给出该指令执行“执行后”所需的值。

执 行 前	指 令	执 行 后
(a) BX: FF 75		
CX: 01 A2	`xchg bx, cx`	BX, CX
(b) AX: 01 A2		
Temp: WORD - 1	`xchg Temp, ax`	AX, Temp
(c) DX: FF 75	`xchg dl, dh`	DX
(d) AX: 01 4B		
BX: 5C D9	`xchg ah, bl`	AX, BX
(e) EAX: 12 BC 9A 78	`xchg eax, edx`	EAX, EDX
EDX: 56 DE 34 F0		

4. 给出练习题3中每条指令的操作码。
5. 假定number指的是某个程序的数据段中的一个双字，需要将该双字的内容与EDX寄存器的内容交换，有两种可能的方法:

```
xchg    edx, number
```

和

```
mov  eax, edx
mov  edx, number
mov  number, eax
```

(a) 假定使用的是Pentium计算机，则每种方法所需的总时钟数和总的字节数为多少？如果使用的是80386计算机则又为多少？

(b) 如果使用的是166 MHz Pentium计算机，则执行每个指令集需要多少纳秒？如果使用的是20MHz的80386计算机，则执行每个指令集需要多少纳秒？

(c) 在上述第二种方法的三个“mov”指令语句中，如果使用的是EBX寄存器而不是累加器EAX，答案会有什么不同？

6. 注意，xchg不能交换寄存器中的两个值。写出存储在Value1和Value2中的双字在用mov 和（或者）xchg交换时的顺序。假定使用的任意的32位寄存器是可用的，并且代码有尽可能的足够的时间和空间。

7. 假定是Pentium系统，下列指令需要多少时钟周期和字节？

```
mov  dx, [ebx]          ; 复制表入口
```

4.2 整数的加法和减法指令

Intel 80x86微处理器有add和sub指令，能完成字节、字或双字长操作数的加减运算。操作数可以解释为无符号数或2进制补码的有符号数。80x86体系结构也提供了inc和dec指令，用来对单操作数进行加1或减1操作，以及neg指令用来取单操作数的补码。

本节讲到的指令和4.1节讲到的mov、xchg指令有所不同，add、sub、inc、dec和neg指令都会对EFLAG寄存器的标志位进行更新。根据操作数的结果来设置SF、ZF、OF、AF标志位的值。例如，如果操作数的结果是负的，那么，符号标志位SF将设置为1；如果操作数的结果为0，那么，零标志位ZF将设置为1。除了add和sub指令，其他指令也会影响进位标志位CF。

加法指令有以下形式:

```
add 目的操作数，源操作数
```

执行加法指令时，源操作数中的整数和目的操作数中的整数相加，相加的和将取代目的操作数中原来的值。减指令有如下形式:

```
sub 目的操作数，源操作数
```

减法指令执行时，目的操作数中的整数减去源操作数中的整数，相减的差将取代目的操作数中原来的值。对于减法，注意，计算得到的差是:

```
目的操作数 - 源操作数
```

或者“操作数1－操作数2”。add和sub指令都不会改变源（第二个）操作数的值。下面举例说明这些指令的执行。

执行前	指令执行	执行后
AX: 00 75	`add ax, cx`	AX [02][17]
CX: 01 A2		CX [01][A2]
		SF 0 ZF 0 CF 0 OF 0
EAX: 00 00 00 75	`sub eax, ecx`	EAX [FF][FF][FE][D3]
ECX: 00 00 01 A2		ECX [00][00][01][A2]
		SF 1 ZF 0 CF 0 OF 0
AX: 77 AC	`add ax, cx`	AX [C2][E1]
CX: 4B 35		CX [4B][35]
		SF 1 ZF 0 CF 0 OF 1
EAX: 00 00 00 75	`sub ecx, eax`	EAX [00][00][00][75]
ECX: 00 00 01 A2		ECX [00][00][01][2D]
		SF 0 ZF 0 CF 0 OF 0
BL: 4B	`add bl, 4`	BL [4F]
		SF 0 ZF 0 CF 0 OF 0
DX: FF 20	`sub dx, value`	DX [00][00]
Value中的字: FF 20		Value [FF][20]
		SF 0 ZF 1 CF 0 OF 0
EAX: 00 00 00 09	`add eax, 1`	EAX [00][00][00][0A]
		SF 0 ZF 0 CF 0 OF 0
Dbl中的双字: 00 00 01 00	`sub Dbl, 1`	Dbl [00][00][00][FF]
		SF 0 ZF 0 CF 0 OF 0

加法指令和减法指令设置SF标志位与结果的最高位相同，因此，当这些指令用来进行2进制补码整数的加减运算时，如果结果为负，则SF标志位为1。如果结果为0，那么零标志位ZF为1；如果结果为非零，则零标志位的值为0。对于进位标志位CF，它表示加法时向高位的进位以及减法时的借位。溢出标志位OF表示溢出，这在第2章中讨论过。

使用2进制补码数表示有符号数，是因为它不需要为加减指令准备特殊的硬件。同样的电路可以用来对无符号数和2进制补码数进行加法运算。根据操作数的类型不同，标志位会有不同的意义。例如，如果对两个大的无符号整数相加，运算结果的最高位的值为1，那么，SF标志位也将置为1，但它并不表示结果是一个负值，只不过是表示一个相对较大的和而已。对于无符号数相加，如果CF标志位的值为1，则表明结果太大而不能存储在目的位置上；对于有符号数相加，如果OF标志位为1，则表示空间不够而导致溢出。

表4-5列出了加法和减法指令。对于每一条加法指令，都有一个相对应的减法指令，它们有相同的操作数类型，相同的时钟周期数，相同长度的目标代码，因此，不必将加法指令和减法指令分开列出。

表4-5 加减指令

目的操作数	源操作数	时钟周期数			字节数	操作码	
		386	486	Pentium		add	sub
8位寄存器	8位寄存器	2	1	1	3	80	80
16位寄存器	8位寄存器	2	1	1	3	83	83
32位寄存器	8位寄存器	2	1	1	3	83	83
16位寄存器	16位立即数	2	1	1	4	81	81
32位寄存器	32位立即数	2	1	1	6	81	81
AL	8位立即数	2	1	1	2	04	2C
AX	16位立即数	2	1	1	3	05	2D
EAX	32位立即数	2	1	1	5	05	2D
存储器字节	8位立即数	7	3	3	3+	80	80
存储器字	8位立即数	7	3	3	3+	83	83
存储器双字	8位立即数	7	3	3	3+	83	83
存储器字	16位立即数	7	3	3	4+	81	81
存储器双字	32位立即数	7	3	3	6+	81	81
8位寄存器	8位寄存器	2	1	1	2	02	2A
16位寄存器	16位寄存器	2	1	1	2	03	2B
32位寄存器	32位寄存器	2	1	1	2	03	2B
8位寄存器	存储器字节	6	2	2	2+	02	2A
16位寄存器	存储器字	6	2	2	2+	03	2B
32位寄存器	存储器双字	6	2	2	2+	03	2B
存储器字节	8位寄存器	7	3	3	2+	00	28
存储器字	16位寄存器	7	3	3	2+	01	29
存储器双字	32位寄存器	7	3	3	2+	01	29

从表4-5可以清楚地看出，当两个操作数都在寄存器中，加法指令和减法指令是最快的；而当两个操作数都在存储器中则是最慢的。值得注意的是，一个存储器中的操作数与寄存器中的数相加比一个寄存器中的数与存储器中的数相加要快，这是因为存储器在后者中必须访问两次，一次是从存储器取加数，另一次是将和写回存储器。使用80x86体系，仅仅允许存储器中最多有一个操作数。有些计算机体系没有目的操作数是存储器操作数的运算指令，但也有一些处理器允许两个存储器操作数的运算指令。

对于add和sub指令，累加器还有特殊的指令，当目的操作数是EAX、AX或AL而源操作数是立即数时，这些指令不会比其他“立即数—寄存器”的指令执行得快，但却有更少字节的目标代码。

对于表4-5中带“+”号的项，一旦知道存储器操作数的类型，就可算出指令的目标代码的总字节数。特别是对直接模式，如果是32位地址则需加4字节。对于寄存器间接模式，则不需要额外的字节。

注意，即便目的操作数是一个字或双字，立即数的源操作数也可以是一个字节。因为立即操作数通常比较小，这使得目标代码更紧凑。在执行加减指令前，字节长度的操作数被带

符号扩展（sign-extended）为字或者双字，如果原来的操作数是负数（可看作二进制补码数），那么，用一个或三个FF字节扩展成相应的字或双字长的值。一个非负操作数可以简单的用一个或三个00字节扩展成相应的字或双字长的值。这两种情况都需复制原来操作数的符号位，使高8位或24位与符号位相同。

一些add和sub指令有同样的操作码，在这种情况下，第二个指令字节的某一个字段用来区分加减指令。本书后面将涉及到这些具有同样操作码的指令。

inc（加）和dec（减）指令是有特定用途的加、减指令。通常1作为默认的源操作数，它们具有以下形式:

```
inc 目的操作数
```

和

```
dec 目的操作数
```

类似add和sub这样的指令，inc和dec指令成对出现，也要考虑操作数类型、时钟周期、目标代码，表4-6对其进行了总结。

表4-6 inc和dec指令

目的操作数	时钟周期数			字节数	操作码	
	386	486	Pentium		inc	dec
8位寄存器	2	1	1	2	FE	FE
16位寄存器	2	1	1	1		
AX					40	48
CX					41	49
DX					42	4A
BX					43	4B
SP					44	4C
BP					45	4D
SI					46	4E
DI					47	4F
32位寄存器	2	1	1	1		
EAX					40	48
ECX					41	49
EDX					42	4A
EBX					43	4B
ESP					44	4C
EBP					45	4D
ESI					46	4E
EDI					47	4F
存储器字节	6	3	3	2+	FE	FE
存储器字	6	3	3	2+	FF	FF
存储器双字	6	3	3	2+	FF	FF

inc和dec指令，把目的操作数看作无符号整数，就像加减指令一样，运算结果将影响OF，SF和ZF标志位，但是不会改变进位标志位CF。下面的例子显示了inc和dec指令是如何执行的。

执行前	指令执行	执行后
ECX: 00 00 01 A2	inc ecx	ECX [00][00][01][A3] SF 0 ZF 0 OF 0
AL: F5	dec al	AL [F4] SF 1 ZF 0 OF 0
Count中的字: 00 09	inc Count	Count [00][0A] SF 0 ZF 0 OF 0
BX: 00 01	dec bx	BX [00][00] SF 0 ZF 1 OF 0
EDX: 7F FF FF FF	inc edx	EDX [80][00][00][00] SF 1 ZF 0 OF 1

inc和dec指令在计数器做加1或减1时十分有用，相对于其他加减指令，有时它们需要较少的代码以及较少的时钟周期。例如，指令:

```
add cx, 1; 循环计数器的增加
```

以及

```
inc cx; 循环计数器的增加
```

这两条指令具有相同的功能，add指令需要三个字节（因为立即数只需占用一个字节，所以占用三个字节而不是四个），而inc指令只需要一个字节。不管是哪条指令，在80386上执行都需两个时钟周期，在 80486或Pentium上执行需要一个时钟周期，所以它们的执行时间都是一样的。

从表4-6可以看出，对于存储在寄存器中的字或双字操作数，单字节的inc和dec指令的执行速度更快，因此，如果要预留一个寄存器的话，最好把计数器的值存放在寄存器中。

neg取补指令或者取2进制数补码数，它是单操作数。如果正数取补，其结果为负数；对负数取补其结果为正数，而零仍然还是零；neg指令都有如下形式:

```
neg 目的操作数
```

表4-7列出了neg指令可用的操作数。

表4-7 neg指令

目的操作数	时钟周期数			字节数	操作码
	386	486	Pentium		
8位寄存器	2	1	1	2	F6
16位寄存器	2	1	1	2	F7
32位寄存器	2	1	1	2	F7
存储器字节	6	3	3	2+	F6
存储器字	6	3	3	2+	F7
存储器双字	6	3	3	2+	F7

下面列举了neg指令是如何执行的4个例子。每种情况下，取补“后”的值正好是取补“前”的二进制补码数。

执行前	指令执行	执行后
BX: 01 A2	neg bx	BX [FE][5E] SF 1 ZF 0
DH: F5	neg dh	DH [0B] SF 0 ZF 0
Flag: 00 01	neg Flag	Flag [FF][FF] SF 1 ZF 0
EAX: 00 00 00 00	neg eax	EAX [00][00][00][00] SF 0 ZF 1

本节最后给出了使用这些指令的一个完整例子程序，这个程序要求输入x，y，z三个数，计算表达式$-(x+y-2z+1)$的值，并显示其结果。设计实现如下：

```
提示输入x的值
将x从ASCII码转换成二进制补码数
表达式:= x
提示输入y的值
将y从ASCII码转换成二进制数
将y加到表达式里，给出x + y
提示输入z的值
将z从ASCII码转换成二进制数
计算2*z当作（z + z）
将表达式里减去2*z，给出x + y - 2*z
给表达式 + 1，给出x + y - 2*z + 1
为表达式取补，给出 -(x + y - 2*z + 1)
将结果从二进制数转换成ASCII码
显示结果
```

写汇编语言程序时，需要安排如何使用寄存器和存储器。在该程序中，在计算完含有x，y，z的表达式后就不再需要保存它们的值了，因此，不需将它们保存于存储器中。假设数字不是很大，可以用字长来存储，因为某些运算使用AX计算更快，所以，表达式的值逻辑上最好存放在累加器AX中。但由于atoi宏使用AX作为目的操作数，因此，这样的做法行不通。这样，通用寄存器剩下BX、CX和DX，该程序选择使用DX。由于在编写汇编程序时，很容易用完寄存器，即使访问存储器的速度慢，也得用存储器来存放数值。有时，必须在存储器和寄存器之间传送数据。

代码段4-1给出了程序源代码，本程序遵循与代码段3-1中的例子同样的模式，在提示符下，注意使用回车符CR、换行符LF，LF跳转到新的一行并留出一空行。没有必要再键入一个CR，因为在回车显示后，光标已经在下一行开始处。2*z的值通过z加z得出，下一节将介绍乘法，

但是计算2*z使用加法计算效率更高。注意，本程序中的注释不仅仅是简单地重复指令助记符，它还有助于对该程序的理解。

代码段4-1 计算－(x + y－2z + 1) 的程序

```
; 该程序要求输入x, y及z的值
; 并计算表达式-(x + y - 2z + 1)
; 作者: R. Detmer
; 日期: 1997年8月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

include io.h                ; 包含有input/output宏的文件

cr      equ     0dh         ; 回车符
Lf      equ     0ah         ; 换行符

.STACK  4096                ; 预留4096字节的堆栈
.DATA                       ; 数据保留区
Prompt1     BYTE    "This program will evaluate the expression",cr,Lf,Lf
            BYTE    "     - (x + y - 2z + 1)",cr,Lf,Lf
            BYTE    "for your choice of integer values.",cr,Lf,Lf
            BYTE    "Enter value for x:  ",0
Prompt2     BYTE    "Enter value for y:  ",0
Prompt3     BYTE    "Enter value for z:  ",0
Value       BYTE    16 DUP (?)
Answer      BYTE    cr,Lf,"The result is "
Result      BYTE    6 DUP (?)
            BYTE    cr,Lf,0

.CODE                                   ; 主程序代码开始
_start:
            output  Prompt1             ; 提示输入x
            input   Value,16            ; 读入ASCII字符
            atoi    Value               ; 转换为整数
            mov     dx,ax               ; 将x保存到dx

            output  Prompt2             ; 提示输入y
            input   Value,16            ; 读入ASCII字符
            atoi    Value               ; 转换为整数
            add     dx,ax               ; 计算x+y

            output  Prompt3             ; 提示输入z
            input   Value,16            ; 读入ASCII字符
            atoi    Value               ; 转换为整数
            add     ax,ax               ; 计算2*z
            sub     dx,ax               ; 计算x+y-2*z

            inc     dx                  ; 计算x+y-2*z+1
            neg     dx                  ; 计算-(x+y-2*z+1)

            itoa    Result,dx           ; 将最后结果转换为ASCII字符集
```

```
            output Answer            ; 输出提示和结果

            INVOKE ExitProcess, 0    ; 退出并返回代码0

PUBLIC _start                        ; 公开程序入口点
END                                  ; 源代码结束
```

图4-1给出了本程序的运行情况，和前面的例子一样，划线部分由用户输入。

```
This program will evaluate the expression

    - (x + y - 2z + 1)

for your choice of integer values.

Enter value for x:  10
Enter value for y:  3
Enter value for z:  5

The result is     -4
```

图4-1 程序运行示例

练习4.2

1. 对于每条指令，给出在Pentium机上运行所需的操作码、目标代码的字节数和时钟周期数。假定Value指的是寄存器中的字并且Double指的是双字。

(a) `add ax, Value`	(b) `sub Value, ax`
(c) `sub eax, 10`	(d) `add Double, 10`
(e) `add eax, [ebx]`	(f) `sub [ebx], eax`
(g) `sub dl, ch`	(h) `add bl, 5`
(i) `inc bx`	(j) `dec al`
(k) `dec Double`	(l) `inc BYTE PTR [esi]`
(m) `neg eax`	(n) `neg bh`
(o) `neg Double`	(p) `neg WORD PTR [ebx]`

2. 对于如下问题的每部分，假设已有指令执行“执行前”的值，给出该指令执行“执行后”所需的值。

执行前	指令	执行后
(a) EBX: FF FF FF 75 ECX: 00 00 01 A2	`add ebx, ecx`	EBX, ECX, SF, ZF, CF, OF
(b) EBX: FF FF FF 75 ECX: 00 00 01 A2	`sub ebx, ecx`	EBX, ECX, SF, ZF, CF, OF
(c) BX: FF 75 CX: 01 A2	`sub cx, bx`	BX, CX, SF, ZF, CF, OF
(d) DX: 01 4B	`add dx, 40h`	DX, SF, ZF, CF, OF

(e) EAX: 00 00 00 64	sub	eax, 100	EAX, SF, ZF, CF, OF
(f) AX: 0A 20			Value
Value: FF20	add	ax, Value	AX, SF, ZF, CF, OF
(g) AX: 0A 20			Value
Value: FF 20	sub	Value, ax	AX, SF, ZF, CF, OF
(h) CX: 03 1A	inc	cx	CX, SF, ZF
(i) EAX: 00 00 00 01	dec	eax	EAX, SF, ZF
(j) Count: 00 99	inc	Count	Count, SF, ZF
(k) Count: 00 99	dec	Count	Count, SF, ZF
(l) EBX: FF FF FF FF	neg	ebx	EBX, SF, ZF
(m) CL: 5F	neg	cl	CL, SF, ZF
(n) Value: FB 3C	neg	Value	Value, SF, ZF

编程练习4.2

对于完整的程序，输入提示必须解释什么是待输入的，同时，输出必须正确地标识出来。

1. 写一个完整的80x86汇编语言程序，提示输入x，y，z的值，并显示表达式$x-2y+4z$的值。其中，输入值为16位的整型值。
2. 写一个完整的80x86汇编语言程序，提示输入x，y，z的值，并显示表达式$2(-x+y-1)+z$的值。其中，输入值为32位的整型值。
3. 写一个完整的80x86汇编语言程序，提示输入一个矩形的长和宽，并显示它的周长（2×长+2×宽）。

4.3 乘法指令

80x86体系结构有两条乘法指令，imul指令把操作数作为有符号数，乘积结果的符号由有符号数的乘法规则决定。mul指令将操作数作为无符号二进制数，乘积结果也是无符号的。如果是非负数进行乘法运算，通常使用mul而不是imul，这是因为mul的速度较快一些。

mul比imul指令种类少，所以首先介绍mul。有一个单操作数指令的mul指令式如下：

```
mul 源操作数
```

源操作数可以是字节、字或双字，而且它也可以在存储器或寄存器中。另外，一个被乘数总是在累加器中，如果是字节长源操作数，则放在AL中，如果是字长源操作数，则放在AX中，如果是双字长源操作数，则放在EAX中。如果源操作数是字节操作数，那么它将和AL中的字节相乘，其结果是16位长，存放在AX寄存器中；如果源操作数是字操作数，它将和AX中的字相乘，结果是32位长，其中低16位存放在AX寄存器中，高16位存放在DX中；如果源操作数是双字操作数，那么它将和EAX中的双字相乘，得到64位的结果，其低32位存放在EAX中，高32位存放在EDX中。对于字节乘法，AX中原有的值由乘积替换掉；对于字乘法，AX:DX中的数由乘积替换掉；类似地，对于双字乘法，EAX:EDX中的数也都由乘积替换掉。在以上各种情况下，源操作数中的值都不会改变，除非它是目的寄存器长度的一半。

乘积长度是乘数和被乘数的两倍，似乎有些奇怪。但是，在一般的十进制乘法中，这种

情况也可能出现。例如，两个4位数相乘，结果可能是7位或8位数。计算机做乘法操作时，为了避免目的地址空间太小，所以，乘积结果以两倍操作数的空间存放。

即使为乘积提供双倍长的存储空间，对于判断乘积是否与源操作数长度一致，也是十分有用的，也就是说，假如乘积的高位部分是0，使用mul指令，如果乘积的高位部分不为零，那么进位标志位CF和溢出标志位OF将置为1，否则就置为0。乘法指令执行后仅影响AF、PF、SF和ZF标志位，它们的值会重新设置。在第5章中将提到一些检查标志位值的指令，根据标志位，乘积结果的高位部分可能被忽略掉。

在表4-8中，总结了mul指令允许的操作数类型，mul指令中不允许有立即数操作数。注意，乘法指令所需的时钟周期数明显大于加减指令。对于80386和80486而言，实际的时钟周期数由运算中的操作数决定。

表4-8 mul指令

操作数	时钟周期数			字节数	操作码
	386	486	Pentium		
8位寄存器	9 ~ 14	13 ~ 18	11	2	F6
16位寄存器	9 ~ 22	13 ~ 26	11	2	F7
32位寄存器	9 ~ 38	13 ~ 42	10	2	F7
存储器字节	12 ~ 17	13 ~ 18	11	2+	F6
存储器字	12 ~ 25	13 ~ 26	11	2+	F7
存储器双字	12 ~ 41	13 ~ 42	10	2+	F7

下面的例子详细说明了乘法指令是如何执行的。

执行前	指令执行	执行后
AX: 00 05	mul bx	DX [00][00]
BX: 00 02		AX [00][0A]
DX: ?? ??		CF, OF 0
EAX: 00 00 00 0A	mul eax	EDX [00][00][00][00]
EDX: ?? ?? ?? ??		EAX [00][00][00][64]
		CF, OF 0
AX: ?? 05	mul Factor	AX [04][FB]
Factor字节: FF		CF, OF 1

第一个例子给出了AX和BX中字的乘法，DX的内容在乘法运算中没有使用到，但是，DX被32位乘结果0000000A的高16位替换掉了。因为DX的内容为0000，进位标志位和溢出标志位全部清零。第二个例子是EAX和自身做乘法，说明了显式的源操作数可以和另外一个相同的隐式乘数相乘。最后一个例子是AL中字节的乘法，另外一个乘数是存储器中一个字节Factor，值相当于十进制数255，得到乘积是16位无符号数04FB，由于高位部分不为零，所以

CF和OF同时置为1。

有符号数乘法指令的助记符是imul，有三种格式，每种格式都有不同的操作数。第一种格式如下:

```
imul 源操作数
```

这种格式和mul一样，用源操作数中的内容作为一个乘数，累加器作为另一个乘数，源操作数不能是立即数。根据源操作数大小，目的寄存器可能是AX、DX:AX或EDX:EAX，如果高位字节是有意义的，进位和溢出标志位置为1，否则置为0。注意，如果乘积是负数，高位部分全为1。表4-9中总结了单操作数imul指令，注意，表4-9和表4-8是一样的。即使mul与单操作数的imul指令有相同的操作码，也会因为指令的第二个字节的不同而使两者有较大差别。

表4-9 imul指令（单操作数格式）

操作数	时钟周期数			字节数	操作码
	386	486	Pentium		
8位寄存器	9～14	13～18	11	2	F6
16位操作数	9～22	13～26	11	2	F7
32位寄存器	9～38	13～42	10	2	F7
存储器字节	12～17	13～18	11	2+	F6
存储器字	12～25	13～26	11	2+	F7
存储器双字	12～41	13～42	10	2+	F7

imul指令的第二种格式如下:

```
imul 寄存器，源操作数
```

源操作数可以在寄存器中、存储器中或是立即数，另一个乘数在寄存器中，它也作为目的地址。操作数必须是字或双字，而不能是字节，乘积必须与乘数的长度一致，如果是这种情况，CF和OF清零，否则都置1。

表4-10总结了双操作数的imul指令，有一些指令有双字节长的操作码，注意，立即数操作数可以是目的寄存器的大小或者是一个字节长。在乘法运算前，单字节操作数根据符号位扩展——也就是说，起始位和符号位相同，扩展后16位或32位的值同样表示原来的8位有符号操作数。

表4-10 imul指令（双操作数格式）

操作数1	操作数2	时钟周期数			字节数	操作码
		386	486	Pentium		
16位寄存器	16位寄存器	9～22	13～26	11	3	0F AF
32位寄存器	32位寄存器	9～38	13～42	10	3	0F AF
16位寄存器	存储器字	12～25	13～26	11	3+	0F AF
32位寄存器	存储器双字	12～41	13～42	10	3+	0F AF
16位寄存器	字节立即数	9～14	13～18	11	3	6B
16位寄存器	字立即数	9～22	13～26	11	4	69
32位寄存器	字节立即数	9～14	13～18	11	3	6B
32位寄存器	双字立即数	9～38	13～42	10	6	69

imul指令的第三种格式如下:

```
imul 寄存器, 源操作数, 立即数
```

在这个指令格式中，第一个操作数是寄存器，仅用来存放乘积，第二个操作数可能在寄存器中或者在存储器中，第三个操作数为立即数。第一个操作数和第二个操作数长度相同，都是16位或都是32位。如果乘积和目的寄存器长度一致，那么CF和OF清零，否则置1。表4-11总结了三个操作数的imul指令。

表4-11 imul指令（三操作数格式）

目的寄存器	源操作数	立即数操作数	时钟周期数			字节数	操作码
			386	486	Pentium		
16位寄存器	16位寄存器	字节	9~14	13~18	10	3	6B
16位寄存器	16位寄存器	字	9~22	13~26	10	4	69
16位寄存器	16位存储器	字节	12~17	13~18	10	3+	6B
16位寄存器	16位存储器	字	12~25	13~26	10	4+	69
32位寄存器	32位寄存器	字节	9~14	13~18	10	3	6B
32位寄存器	32位寄存器	双字	9~38	13~42	10	6	69
32位寄存器	32位存储器	字节	12~17	13~18	10	3+	6B
32位寄存器	32位存储器	双字	12~41	13~42	10	6+	69

下面的例子有助于理解imul指令。

执行前	指令执行	执行后
AX: 00 05	`imul bx`	DX [00 | 00] AX [00 | 0A] CF, OF 0
BX: 00 02 DX: ?? ??		
AX: ?? 05 Factor字节: FF	`imul Factor`	AX [FF | FB] CF, OF 0
EBX: 00 00 00 0A	`imul ebx, 10`	EBX [00 | 00 | 00 | 64] CF, OF 0
ECX: FF FF FF F4 Double中的双字: FF FF FF B2	`imul ecx, Double`	ECX [00 | 00 | 03 | A8] CF, OF 0
Value中的字: 08 F2 BX: ?? ??	`imul bx, Value, 1000`	BX [F1 | 50] CF, OF 1

前两个例子是单操作数格式，乘积是操作数长度的两倍，第一个例子给出了AX中的字（隐式操作数）与BX相乘，其结果存于DX:AX。第二个例子说明了AL中的5与存储器中的Factor（值为-1）相乘，得到一个字长的乘积，其值为-5，存于AX中。第三个例子给出了

双操作数格式，EBX中的10和立即数10相乘，得到的乘积100存放于EBX中。第四个例子是两个负数相乘，得到的乘积为正。在最后一个例子中，乘积是16进制的22F150，因为值太大而不能存放于BX中，所以CF，OF全被置为1，表明结果太大，低位部分存放于BX中。

对于代码段4-1中的示例程序，已经讨论过计算2z用z加z得出要快于用乘法指令。对于前者，z存放于AX寄存器中，于是

```
add ax, ax     ; 计算2z
```

完成2z的计算。该指令有两个字节长，在80486或Pentium系统上需占用一个时钟周期。同样，也可用乘法指令实现

```
imul ax, 2     ; 计算2z
```

该指令（从表4-10可以得到）是三字节长，由于立即操作数2比较短，足以单字节存储，它占用了13～18个80486的时钟周期或10个Pentium的时钟周期，这都远远长于加法指令所占用的时钟周期。

本节用一个程序例子做总结。该程序要求输入矩形的长和宽，并计算矩形的面积（长*宽）。（很明显，相比用汇编语言或者其他语言编写计算机程序来计算，手算更适合）代码段4-2给出了程序的源代码。注意，由于长和宽都是正数，所以该程序使用mul而不是imul来计算乘积。如果输入长或宽为负数，或者长或宽太大（比方为200和300），程序将出错。为什么？遗憾的是，这样的错误在程序中很普遍。

代码段4-2 计算长方形面积的程序

```
; 计算矩形面积
; 作者: R. Detmer
; 日期: 1997年9月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

INCLUDE io.h

cr          EQU     0dh     ; 回车
LF          EQU     0ah     ; 换行

.STACK  4096                ; 预留4096字节的堆栈

.DATA                       ; 数据保留区
prompt1     BYTE    "This program will find the area of a
                      rectangle",cr,Lf,Lf
            BYTE    "Width of rectangle?  ",0
prompt2     BYTE    "Length of rectangle?  ",0
value       BYTE    16 DUP (?)
answer      BYTE    cr,Lf,"The area of the rectangle is "
area        BYTE    11 DUP (?)
            BYTE    cr,Lf,0

.CODE                       ; 主程序代码开始
```

```
_start:
Prompt:     output prompt1          ; 提示输入宽度
            input  value,16         ; 读取ACSII字符
            atod   value            ; 转换为整数
            mov    ebx,eax          ; 将宽度值保存到ebx

            output prompt2          ; 提示输入长度
            input  value,16         ; 读取ACSII字符
            atod   value            ; 转换为整数
            mul    ebx              ; 计算长×宽

            dtoa   area,eax         ; 将面积结果转换为字符集
            output answer           ; 输出提示和最后得到的结果

            INVOKE ExitProcess, 0   ; 退出并返回0
PUBLIC _start                       ; 公开程序入口前
            END
```

从本节内容可了解到，80x86体系结构包含三种乘法格式。也注意到，乘积不能是存储器操作数，这看起来有很大局限性，但一些处理器甚至有更多的限制。实际上，包括Intel 8086（译者注：原书为8080，但译者认为是8086）在内的大多数8位处理器都没有乘法指令，任何乘法指令都需用软件来实现。

练习4.3

1. 对于如下问题的每部分，假设已有指令执行“执行前”的值，给出该指令所需的执行“执行后”值。

执　行　前	指　　令	执　行　后
(a) EAX: FF FF FF E4 EBX: 00 00 00 02	mul ebx	EAX, EDX, CF, OF
(b) AX: FF E4 Value: FF 3A	mul Value	AX, DX, CF, OF
(c) AX: FF FF	mul ax	AX, DX, CF, OF
(d) AL: 0F BH: 4C	mul bh	AX, CF, OF
(e) AL: F0 BH: C4	mul bh	AX, CF, OF
(f) AX: 00 17 CX: 00 B2	imul cx	AX, DX, CF, OF
(g) EAX: FF FF FF E4 EBX: 00 00 04 C2	imul ebx	EAX, EDX, CF, OF
(h) AX: FF E4 Value: FF 3A	imul Value	AX, DX, CF, OF
(i) EAX: FF FF FF FF	imul eax	EAX, EDX, CF, OF
(j) AL: 0F BH: 4C	imul bh	AX, CF, OF

(k) AL: F0
BH: C4 `imul bh` AX, CF, OF

2. 给出练习1中每条指令的操作码。
3. 对于如下问题的每部分，假设已有指令执行“执行前”的值，给出该指令所需的执行“执行后”值。

执行前	指令	执行后
(a) BX: 00 17		
CX: 00 B2	`imul bx, cx`	BX, CF, OF
(b) EAX: FF FF FF E4		
EBX: 00 00 04 C2	`imul eax, ebx`	EAX, CF, OF
(c) AX: 0F B2	`imul ax, 15`	AX, CF, OF
(d) ECX: 00 00 7C E4		
Mult:	`imul ecx, Mult`	ECX, CF, OF
00 00 65 ED		
(e) DX: 7C E4		
BX: 49 30	`imul dx, bx`	DX, CF, OF
(f) DX: 0F E4		
Value: 04 C2	`imul dx, Value`	DX, CF, OF
(g) EBX: 00 00 04 C2	`imul ebx, -10`	EBX, CF, OF
(h) ECX: FF FF FF E4	`imul ebx, ecx, 5`	EBX, CF, OF
(i) DX: 00 64	`imul ax, dx, 10`	AX, CF, OF

4. 给出练习3中每条指令的操作码。
5. 假定x的值存储在AX寄存器中，并且从AX寄存器中得到$5x$的值。比较这些指令在Pentium机上执行时的时钟周期，以及下列每种方法中目标代码的字节数。

```
mov  bx, ax          ; 复制x的值
add  ax, ax          ; x + x得2x
add  ax, ax          ; 2x + 2x得4x
add  ax, bx          ; 4x + x得5x
```

和

```
imul ax, 5           ; 5x
```

6. 假定对于x的某个值，需要计算下列多项式的值:

$p(x) = 5x^3 - 7x^2 + 3x - 10$

如果以直观的方法计算该多项式的值，如:

$5*x*x*x - 7*x*x + 3*x - 10$

则有六次乘法运算和三次加/减法运算。基于Horner方法计算该多项式的值，则可等价于如下形式:

$((5*x - 7)*x + 3)*x - 10$

这个表达式中仅有三次乘法运算。

假定x的值在EAX寄存器中。

(a) 若用最“直观的”方法计算$p(x)$ 多项式的值，写出80x86汇编语言的描述，并将计算所得的值存放在EAX中。

(b) 若用Horner方法计算$p(x)$ 多项式的值，写出80x86汇编语言的描述，并再把运算结果存放在EAX中。

(c) 假定是Pentium系统，比较（a）和（b）中执行的时钟周期数和代码段所需的目标代码字节数。

7. 80x86体系结构对于有符号数和无符号数的乘法有着不同的指令，而对于有符号数和无符号数的加法却没有各自的指令。为什么对于乘法需要有不同的指令，而对于加法却没有呢？

编程练习4.3

1. 编写一个完整的80x86汇编语言程序提示输入一个箱子的长、宽、高，并显示它的体积（长×宽×高）。
2. 编写一个完整的80x86汇编语言程序提示一个箱子的长、宽、高，并显示它的表面积2×(长×宽＋长×高＋宽×高)。
3. 假定某人有一定数量的硬币(便士、五分硬币、一角硬币、二角五分、五十分和美元硬币)，并且需要知道总的硬币值和有多少个硬币。编写一个程序解决这个问题。特别地，要求遵循下列设计：

```
提示并输入便士的个数;
币值总数:=便士的个数
硬币的个数:=便士的个数
提示并输入五分硬币的个数;
币值总数:=前面币值累计总数＋5×五分硬币的个数;
将五分硬币的个数加到硬币个数总和;
提示并输入一角的个数;
币值总数:=前面币值累计总数+10×一角硬币的个数;
将一角硬币的个数加到硬币个数总和;
提示并输入二角五分硬币的个数;
币值总数:=前面币值累计总数+25×二角五分硬币的个数;
将二角五分硬币的个数加到硬币个数总和;
提示并输入五十分硬币的个数;
币值总数:=前面币值累计总数+50×五十分硬币的个数;
将五十分硬币的个数加到硬币个数总和;
提示并输入一美元硬币的个数;
币值总数:=前面币值累计总数+100×一美元硬币的个数;
将一美元硬币的个数加到硬币个数总和;
最后显示"总共有"，硬币总数;
以及显示"有"，币值总数div100，"美元以及"，币值总数mod100，"分币";
注意：需要显示币值总数的美元数和分币数，并且假定所有的数值都是双字长。
```

4.4 除法指令

Intel 80x86的除法指令和单操作数乘法指令很相似，指令idiv用于有符号二进制补码整数的除法，指令div用于无符号整数的除法。注意到，单操作数乘法指令用乘数和被乘数相乘，

并得到一个两倍长的乘积。而除法指令用一个两倍长的数作为被除数，用一个单倍长度的数作为除数，最后得到单倍长度的商和单倍长度的余数。在开始除法之前，80x86用指令来产生一个双倍长的被除数。

除法指令有如下格式:

```
idiv 源操作数
```

以及

```
div 源操作数
```

源操作数就是除数。除数可以存放在寄存器或存储器中，但不能是立即数。idiv和div使用隐式的被除数（该操作数被除）。如果源操作数是一个字节长度，那么双倍长度的被除数是字长度，并假定它存放在寄存器AX中；如果源操作数是一个字长，那么双倍长度的被除数是双字长度，并假定它的低16位存放在AX寄存器中，且高16位在DX寄存器中；如果源操作数是一个双字的长度，那么被除数应是四字长度（64位），并假定它的低32位在EAX寄存器，高32位在EDX寄存器中。

表4-12总结了80x86除法指令中被除数、除数、商和余数的各种情况。

表4-12 80x86除法指令的操作数和结果

源操作数（除数）大小	第二个操作数（被除数）	商	余 数
字节	AX	AL	AH
字	DX:AX	AX	DX
双字	EDX:EAX	EAX	EDX

除法指令不会改变源操作数（除数）。AX中字长的被除数被字节长的除数进行除操作后，商将放入寄存器AL中，余数将放入寄存器AH中；DX和AX中的双字被字长的除数进行除操作后，商将存入AX寄存器中，而余数存入寄存器DX中；EDX和EAX中的四字长数被双字长的除数进行除操作后，商将被存入EAX寄存器，而余数将被存入EDX寄存器。

对于所有的除法操作数，被除数、除数、商、余数必须满足下面的等式

```
被除数 = 商 * 除数 + 余数
```

对于无符号数的div指令，被除数、除数、商和余数都被当作非负数看待。对于有符号数除法指令idiv，商的符号使用一般的符号规则，由被除数和除数决定，余数的符号总是和被除数的相同。

除法指令不对任何标志位赋值。它们可能会改变AF、CF、OF、PF、SF和ZF预先设置的值。下面的例子说明了除法指令是如何执行的。

执行前	指令执行	执行后				
EDX: 00 00 00 00	div ebx	EDX	00	00	00	09
EAX: 00 00 00 64		EAX	00	00	00	07
EBX: 00 00 00 0D						

DX: 00 00	idiv cx	DX	00	09
AX: 00 64		AX	00	07
CX: 00 0D				
AX: 00 64	div Divisor	AX	09	07
Divisor的字节: 0D				

在如下的每一个例子中，被除数100都被13除。由

$$100 = 7 * 13 + 9$$

而得到的商是7，同时余数是9。对于双字长的除数，商在EAX中而余数在EDX中。对于字长的除数，商在AX中余数在DX中。对于字节长除数，商在AL中余数在AH中。

对于被除数或者除数是负数的操作，有与上面类似的等式

$$100 = (-7) * (-13) + 9$$
$$-100 = (-7) * 13 + (-9)$$
$$-100 = 7 * (-13) + (-9)$$

注意，在以上每种情况下，余数的符号与被除数的符号都相同。下面的例子给出了当上述的除数13或－13为字长时的等式。

执行前	指令执行	执行后		
DX: 00 00	idiv cx	DX	00	09
AX: 00 64		AX	FF	F9
CX: FF F3				
DX: FF FF	idiv cx	DX	FF	F7
AX: FF 9C				
CX: 00 0D		AX	FF	F9
DX: FF FF				
AX: FF 9C	idiv cx	DX	FF	F7
CX: FF F3		AX	00	07

在第二个和第三个例子中，被除数－100就是在DX和AX寄存器中的32位数FF FF FF 9C。

最后，下面这两个例子有助于说明有符号数除法和无符号数除法的差别。

执行前	指令执行	执行后		
AX: FE 01	idiv bl	AX	E1	0F
BL: E0				
AX: FE 01	div bl	AX	00	FF
BL: FF				

对于有符号数除法，－511被－32除，得到商15和余数－31；对于无符号数除法，65025被255除，得到商255和余数0。

对于乘法，在每个单操作数格式中，双倍长的目的地址以确保能存放乘积——这样在单操作数乘法运算中就不会出错。但是，在除法中就有可能有错，一个显而易见的原因就是除数为0，另外一个不太明显的原因是商太大而不能存入单倍长的目的地址中。比如，00 02 46 8A被2除，商12345太大而不能存入AX寄存器中。如果在除法操作时出错，80x86将发生异常(exception)。处理该异常的程序或者中断控制器可能因系统的不同而不同。在Pentium系统的WIN95中会弹出一个窗口，提示消息为“非法操作，程序将中止”。如果按下“详细”按钮，将显示“测试引发除法错误”。

表4-13列出了idiv指令允许的操作数类型，表4-14列出了div指令允许的操作数类型。这两张表的惟一区别是时钟周期所在列的时钟周期数不同；div的运算速度稍快于idiv的运算速度。

表4-13 idiv指令

操作数	时钟周期数			字节数	操作码
	386	486	Pentium		
8位寄存器	19	19	22	2	F6
16位寄存器	27	27	30	2	F7
32位寄存器	43	43	48	2	F7
存储器字节	22	20	22	2+	F6
存储器字	30	28	30	2+	F7
存储器双字	46	44	48	2+	F7

表4-14 div指令

操作数	时钟周期数			字节数	操作码
	386	486	Pentium		
8位寄存器	14	16	17	2	F6
16位寄存器	22	24	25	2	F7
32位寄存器	38	40	41	2	F7
存储器字节	17	16	17	2+	F6
存储器字	25	24	25	2+	F7
存储器双字	41	40	41	2+	F7

当用给定长度的操作数作除法运算时，在做除法前，被除数必须被转换成双倍长。对于无符号数除法，如果被除数为双字长，通过将EDX寄存器的高位置0，将该被除数转换为四字长。实现的方法有很多种，下面的是其中的两种方法：

```
mov edx, 0
```

和

```
sub edx, edx
```

在做无符号数除法前，如果被除数是字操作数，可以用类似的指令将DX的高位置0，如果被除数是字节操作数，则可将AH的高位置0。

对于有符号数除法，这样做比较复杂。正的被除数高位必须用0扩展，负的被除数高位必须用1扩展，80x86有三个指令可以实现这项工作。与以前的指令不同，这三条指令cbw、cwd和cdq，它们都没有操作数。其中cbw指令是用AL作为源操作数，AX作为目的操作数；cwd用AX作为源操作数，DX和AX作为目的操作数；cdq是用EAX作源操作数，EDX和EAX作为目的操作数。源寄存器的值不会改变，但它们会作为有符号数扩展到AH、DX、或者EDX中。表4-15集中总结了这些指令，表中也给出cwde指令，它扩展AX中的字到EAX，使AX和EAX中的有符号数相等。

表4-15 cbw和cwd指令

指令	时钟周期数			字节数	操作码
	386	486	Pentium		
cbw	3	3	3	1	98
cwd	2	3	2	1	99
cdq	2	3	2	1	99
cwde	3	3	3	1	98

cbw指令（将字节转换为字）将AL寄存器中的二进制补码数扩展为AX中的字长。cwd（将字转换为双字）将AX寄存器中的二进制数扩展为DX和AX中的双字。cdq（将双字转换为四字）将EAX中的双字扩展为EDX和EAX 的四字。

cwde指令（将字转换成双字）将AX中的字扩展为EAX中的双字，这个指令不是通常意义上使用的除法指令。每条指令都复制源数据的符号位到运算结果的高位部分中的每一位，这些指令都不影响标志位。

执行前	指令执行	执行后				
AX: 07 0D DX: ?? ??	cwd	DX	00	00		
		AX	07	0D		
EAX: FF FF FA 13 EDX: ?? ?? ?? ??	cdq	EDX	FF	FF	FF	FF
		EAX	FF	FF	FF	13
AL: 53	cbw	AX	00	53		
AL: C6	cbw	AX	FF	C6		
AX: FF 2A	cwde	EAX	FF	FF	FF	2A

两条“传送”指令在一定程度上像上面的“转换”指令。这些指令将一个8位或16位源操作数复制到16位或32位的目的位置，扩展了源操作数的值。movzx指令总是用位为0来扩展源操作数。有如下格式:

```
movzx 寄存器，源操作数
```

movsx指令用复制符号位来扩展源操作数，有如下格式:

```
movsx 寄存器，源操作数
```

表4-16中给出了这些指令的相关数据。对于任意一条指令，源操作数可以在寄存器中或存储器中，但却没有哪条指令改变任何标志位的值。

表4-16 movsx和movzx指令

目的位置	源位置	时钟周期数			字节数	操作码	
		386	486	Pentium		movsx	movzx
16位寄存器	8位寄存器	3	3	3	3	0F BE	0F B6
32位寄存器	8位寄存器	3	3	3	3	0F BE	0F B6
32位寄存器	16位寄存器	3	3	3	3	0F BF	0F B7
16位寄存器	存储器字节	6	3	3	3+	0F BE	0F B6
32位寄存器	存储器字节	6	3	3	3+	0F BE	0F B6
32位寄存器	存储器字	6	3	3	3+	0F BF	0F B7

下面的例子说明了这些指令是如何实现的。

执行前	指令执行	执行后				
Value: 07 0D	movsx ecx, value	ECX	00	00	07	0D
Value: F7 0D	movsx ecx, value	ECX	FF	FF	F7	0D
Value: 07 0D	movzx ecx, value	ECX	00	00	07	0D
Value: F7 0D	movzx ecx, value	ECX	00	00	F7	0D

本节用另一个简单的程序作总结，该程序用于将摄氏温度转换成华氏温度。代码段4-3给出了源代码。转换用到的公式为:

$$F = (9/5) * C + 32$$

其中F是华氏温度，C是摄氏温度。由于目前介绍的运算指令只进行整数运算，程序得到的是对小数部分进行取整的结果。在除以5之前将9和C相乘非常重要，因为9/5的整数商为1。如果C先被5除，再与9相乘，则这种方法比第一种方法产生的误差更大。为什么？为了得到取整运算的正确结果，在除之前将除数的一半加到被除数中。由于公式中的除数是5，取舍后的2加到被除数中。注意，cwd指令在做除法前用来扩展被除数。

代码段4-3 转换摄氏温度为华氏温度

```
; 转换摄氏温度为华氏温度
; 使用公式F=(9/5)*C + 32
; 作者: R. Detmer
; 日期: 1997年7月
```

```
.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

INCLUDE io.h

cr          EQU     0dh     ; 回车
Lf          EQU     0ah     ; 换行

.STACK  4096                ; 预留4096字节的堆栈

.DATA                       ; 数据保留区
Prompt1     BYTE    CR,LF,"This program will convert a Celsius "
            BYTE    "temperature to the Fahrenheit scale",cr,Lf,Lf
            BYTE    "Enter Celsius temperature:  ",0
Value       BYTE    10 DUP (?)
Answer      BYTE    CR,LF,"The temperature is"
Temperature BYTE    6 DUP (?)
            BYTE    "   Fahrenheit",cr,Lf,0

.CODE                                   ; 主程序代码开始
_start:
Prompt:     output  Prompt1             ; 提示输入摄氏温度
            input   Value,10            ; 读取ASCII字符
            atoi    Value               ; 转换为整数

            imul    ax,9                ; 计算C*9
            add     ax,2                ; 加取整因子2到被除数
            mov     bx,5                ; 除数5
            cwd                         ; 扩展被除数
            idiv    bx                  ; 计算C*9/5
            add     ax,32               ; 计算C*9/5 + 32

            itoa    Temperature,ax      ; 将得到结果转换为ASCII
            output  Answer              ; 输出提示和得到的最后结果

            INVOKE  ExitProcess, 0      ; 退出并返回0
PUBLIC _start                           ; 公开程序入口点
            END
```

练习4.4

1. 对于如下问题的每部分，假设已有指令执行“执行前”的值，给出该指令所需的执行“执行后”值。这些指令中有些指令会造成除法运算出错，识别这样的指令。

执行前	指令	执行后
(a) EDX: 00 00 00 00 EAX: 00 00 00 9A EBX: 00 00 00 0F	idiv ebx	EDX, EAX
(b) AX: FF 75 Count: FC	idiv Count	AX
(c) AX: FF 75 Count: FC	div Count	AX

(d) DX: FF FF
AX: FF 9A
CX: 00 00 `idiv cx` DX, AX

(e) EDX: FF FF FF FF
EAX: FF FF FF 9A
ECX: FF FF FF C7 `idiv ecx` EDX, EAX

(f) DX: 00 00
AX: 05 9A
CX: FF C7 `idiv cx` DX, AX

(g) DX: 00 00
AX: 05 9A
CX: 00 00 `idiv cx` DX, AX

(h) EDX: 00 00 00 00
EAX: 00 00 01 5D
EBX: 00 00 00 08 `idiv ebx` EDX, EAX

2. 给出练习1中每条指令的操作码。
3. 在做无符号数除法之前，本节给出了两种置EDX为0的方法。使用

```
mov  edx, 0
```

或

```
sub  edx, edx
```

哪条指令的编码更简洁？在Pentium中，哪条指令执行所需的时钟周期更少？

4. 华氏温度与摄氏温度转换程序（代码段4-3）可用于摄氏温度，该摄氏温度可以非常大，也可以是正数或负数。假如限定摄氏温度的范围在0 ~ 100度，对应的华氏温度的范围为32 ~ 212。如果考虑温度受限的范围，则该程序如何修改？

编程练习4.4

1. 将华氏温度转换为摄氏温度的公式为:

$C=(5/9)*(F-32)$

写一个完整的80x86汇编语言程序，该程序提示输入华氏温度，并显示其对应的摄氏温度。

2. 写一个完整的80x86汇编语言程序，该程序提示输入四个成绩，然后并显示这四个成绩的总和（sum）以及四个成绩的平均分（sum/4）。
3. 写一个完整的80x86汇编语言程序，该程序提示输入四个成绩。假如最后一个成绩是期末考试成绩，它与其他三个成绩一样计算了两次。显示成绩的总和sum（最后一个成绩加了两次）和平均成绩（sum/5）。
4. 写一个完整的80x86汇编语言程序，该程序提示输入四组成绩和权系数。每个权系数表示其对应的成绩应该在sum中计数的次数。加权总和为:

WeightedSum = Grade1 * Weight1
+ Grade2 * Weight2
+ Grade3 * Weight3

+ Grade4 * Weight4

加权总和为SumOfWeights = Weight1 + Weight2 + Weight3 + Weight4

显示加权的和、权的和以及加权的平均值。(WeightedSum/SumOfWeights)。

可如下的例子显示:

```
grade 1 ?  88
weight 1?   1

grade 2 ?  77
weight 2?   2

grade 3 ?  94
weight 3?   1

grade 4 ?  85
weight 4?   3

weighted sum:  591
sum of weights:  7
weighted average:  84
```

5. 写一个完整的80x86汇编语言程序，该程序提示输入四个成绩，然后，以ddd.dd的格式（十进制小数点前三位，十进制小数点后两位）显示这四个成绩的总和（sum）以及四个成绩的平均分（sum/4）。
6. 写一个简短的程序，该程序用于当除数为0时，80x86系统如何做出中断处理。

4.5 大数的加减

在4.2节中提到了加法和减法指令，其操作数是字节长、字长或双字长。尽管双字的值域已经很大，－2 147 483 648（80000000_{16}）～2 147 483 647（$7FFFFFFF_{16}$），但有时还需要作更大的数的运算，非常大的数加减法可以一次用一组二进制数位的加减来实现。

下面介绍两个64位长的大数的加法。方法是先用通常的add指令对每个大数的低32位相加，如果向高位有进位，置进位标志位CF为1；如果没有向高位的进位，则置CF为0。另外的高32位用一个特殊的加法指令adc（进位加法）。两个数的高32位做正常的加法，如果在相加之前CF为1，则在相加的和送到目的位置之前要加1。adc指令也会改变CF的值。因此，这个过程还可以继续做更多位的加法。

假定在数据段中，两个数进行相加，其中每个数用两个双字长表示。

```
Nbr1Hi DWORD ?    ; Nbr1的高32位
Nbr1Lo DWORD ?    ; Nbr1的低32位
Nbr2Hi DWORD ?    ; Nbr2的高32位
Nbr2Lo DWORD ?    ; Nbr2的低32位
```

下面的代码片段将Nbr1和Nbr2相加，将相加的和保存到预留给Nbr1的双字中。

```
mov eax, Nbr1Lo   ; Nbr1的低32位
add eax, Nbr2Lo   ; 加上Nbr2的低32位
mov Nbr1Lo, eax   ; 结果转到目标位置
mov eax, Nbr1Hi   ; Nbr1的高32位
adc eax, Nbr2Hi   ; 加Nbr2的高32位和进位数
mov Nbr1Hi, eax   ; 结果存入目标位置
```

这段代码中除了add和adc，其间还有mov指令，它不会改变进位标志位。如果add和adc之间有其他指令改变了CF的值，则得到的结果不正确。

当CF为1时，相加的和需再加1，除此之外，adc指令和add指令都一样。对于减法，除了CF位为1时，相减的差需再减1之外，sbb(借位减法)指令与sub指令都一样。大数可从右向左进行一组位的减法。表4-17列出了adc和sbb指令允许的操作数类型。表4-17与表4-5除了部分操作码不同外其他的都相同。

表4-17 adc和abb指令

目的操作数	源操作数	时钟周期数			字节数	操作码	
		386	486	Pentium		adc	sbb
8位寄存器	8位立即数	2	1	1	3	80	80
16位寄存器	8位立即数	2	1	1	3	83	83
32位寄存器	8位立即数	2	1	1	3	83	83
16位寄存器	16位立即数	2	1	1	4	81	81
32位寄存器	32位寄存器	2	1	1	6	81	81
AL	8位立即数	2	1	1	2	14	1C
AX	16位寄存器	2	1	1	3	15	1D
EAX	32位寄存器	2	1	1	5	15	1D
存储器字节	8位立即数	7	3	3	3+	80	80
存储器字	8位立即数	7	3	3	3+	83	83
存储器双字	8位立即数	7	3	3	3+	83	83
存储器字	16位立即数	7	3	3	4+	81	81
存储器双字	32位立即数	7	3	3	6+	81	81
8位寄存器	8位寄存器	2	1	1	2	12	1A
16位寄存器	16位寄存器	2	1	1	2	13	1B
32位寄存器	32位寄存器	2	1	1	2	13	1B
8位寄存器	存储器字节	6	2	2	2+	12	1A
16位寄存器	存储器字	6	2	2	2+	13	1B
32位寄存器	存储器双字	6	2	2	2+	13	1B
存储器字节	8位寄存器	7	3	3	2+	10	18
存储器字	16位寄存器	7	3	3	2+	11	19
存储器双字	32位寄存器	7	3	3	2+	11	19

同样的方法也可适用于更长的数，通常使用相同的循环指令集。即使CF在循环前已知为0，第一次加法也可用adc。80x86有三条指令使编程人员可以操作进位标志位。表4-18中进行了总结。在80386、80486和Pentium处理器上，指令都占用了两个时钟周期数，所以没有单独地列出不同处理器的时钟周期。

表4-18 进位标志位的控制

指令	操作	时钟周期数	字节数	操作码
clc	清空进位标志位 (CF :=0)	2	1	F8
stc	设置进位标志位 (CF :=1)	2	1	F9
cmc	对进位标志位取补 (if CF = 0 then CF := 1 else CF := 0)	2	1	F5

大数的乘法和除法运算比其加减法用到的更多。通常，用大数的加减法实现乘除算法，该算法类似于小学十进制数的乘除法过程。

如果用到更长的数字，它可能需要更多的算术过程，可能还需要像itoa和atoi这样的过程，这些过程用于实现大数和ASCII字符之间的转换。

练习4.5

1. 假如两个96位的长整数相加:
   (a) 给出这三个数如何存储在一个程序的数据段中。
   (b) 给出80x86的部分代码，该代码实现第二个数与第一个数相加，并将相加的和保存在第一个数的地址处。
   (c) 给出80x86的部分代码，该代码实现第二个数与第一个数相加，并将相加的和保存在第三个数的地址处。
2. 假如两个64位的整数以本节所示的例子存储。给出80x86的部分代码，该代码实现Nbr1与Nbr2相减，并把差值保存在Nbr1的地址处。
3. 对于如下的每个问题，假定“执行前”的值在指令执行前已经给定。给出指令执行后的“执行后”值。

执行前	指令	执行后
(a) EAX: 00 00 03 7D ECX: 00 00 01 A2 CF: 0	`adc eax, ecx`	EAX, CF
(b) EAX: 00 00 03 7D ECX: 00 00 01 A2 CF: 1	`adc eax, ecx`	EAX, CF
(c) EAX: FF 49 00 00 ECX: 03 68 00 00 CF: 0	`adc eax, ecx`	EAX, CF
(d) EAX: FF 49 00 00 ECX: 03 68 00 00 CF: 1	`adc eax, ecx`	EAX, CF
(e) EAX: 00 00 03 7D ECX: 00 00 01 A2 CF: 0	`sbb eax, ecx`	EAX, CF
(f) EAX: 00 00 01 A2 ECX: 00 00 03 7D CF: 1	`sbb eax, ecx`	EAX, CF

4.6 其他知识：微代码抽象级

在计算机科学技术中，可从很多层面来考察计算机和计算。当使用像字处理包或游戏这样的应用程序时，仅仅想知道它们的各种工作特性，而并不关心程序是如何编写的。当使用

高级语言编写程序时，可以把计算机看作一台Ada机器或一台C++机器，并且通常不考虑各种语言结构是如何实现的。*应用程序级*（application level）和*高级语言级*（high-level language level）是两种不同的*抽象级*。正如所使用的英文单词“abstraction”，可将其理解为“忽略细节”的意思。

本书主要处理*机器语言级*（machine-language level）的抽象。本书的主要目标之一是将机器语言级与高级语言抽象级联系起来。对于一个硬件设计师，将机器语言级与更低的抽象级相联系尤其重要。

什么是更低的抽象级呢？很显然，计算机硬件必须执行像add或者imul这样的指令。尽管可从晶体管构造这一更低等级的视角看待机器的硬件级，但它经常被看作一个逻辑电路的集合。对于相对简单的结构，可设计电子电路来直接实现各种可能的指令。

对于更复杂的指令集，有另外一个抽象级——*微代码级*（microcode level），它介于用户可见的机器语言级与机器数字电路级之间。微代码级由实际执行指令的程序集组成。微指令通常存储在CPU的永久存储器中。使用微代码的CPU有内部暂存寄存器集和诸如加法器的简单电路，其中，内部暂存寄存器不与用户直接接触。一个机器语言指令通过一系列的微指令来实现，这些微指令实际上访问内部暂存寄存器。微代码类似机器语言，但有很多不同点。微指令有直接控制电路的位。通常，微指令没有程序计数器——每条指令包含下一条指令的地址。总的来说，微程序比汇编语言程序更加复杂。

本章小结

Intel 80x86 mov指令用来将数据从一个位置复制到另一个位置。几乎所有的源位置和目的位置的逻辑组合都是允许的。xchg指令交换存储在两个地址处的数据。

80x86体系结构有很多用于字节长、字长和双字长整数计算的指令集。add和sub指令用来做加法和减法；inc和dec分别做加1和减1操作。neg指令对操作数取补。

有两个乘法指令和两个除法指令。imul和idiv指令假定它们的操作数是有符号的二进制补码数；mul和div假定它们的操作数是无符号的。许多乘法指令使用单倍长度的操作数，并产生一个双倍长度的乘积；其他格式产生一个和乘数相同长度的乘积。除法指令总是以一个双倍长度的被除数和一个单倍长度的除数开始；运算结果是一个单倍长度的商和一个单倍长度的余数。cbw、cwd、cdq指令在做有符号数除法前，辅助产生一个双倍长度的被除数。设置标志位来提示乘法操作时可能会发生的错误；除法操作时发生的错误会产生一个硬件异常，该异常触发一个过程来处理错误。

操作数在寄存器中的指令通常比操作数在存储器中的指令执行速度要快。乘法和除法指令相对于加减法指令执行得要慢。

adc和sbb指令使得比双字更长的数的加减法成为可能，可以一次用一组二进制数位的相加减来实现，该组二进制数位的加减法可能会产生向其左边一组数位的进位或借位。进位或借位由进位标志位CF记录。80x86的clc、stc和cmc指令使得编程人员能够在需要的时候清除、设置和处理进位标志位。

机器语言级仅仅是把计算机看成多个抽象级中的一个级。在机器语言级之上的是高级语言级和应用程序级。在机器语言级之下的是微代码级和硬件级。

第5章　分支和循环

计算机强大的处理能力不仅在于它可选择性地执行代码，还在于它可高速地执行重复的算法。用高级语言编写的程序，如Ada、C++或者Pascal语言，可用if-then、if-then-else和case结构进行编码；并且选择性地运用循环结构重复执行代码，例如while（先测试条件）循环、until（后测试条件）循环以及for（计数控制）循环等等。一些高级语言对于无条件分支结构使用goto语句。一些早期的语言（如BASIC以前的版本）依靠简单的if语句和大量的goto语句，执行选择性的分支和循环结构。

用80x86汇编语言编程和以前用BASIC语言编程相似。80x86微处理器能执行一些比for语句简单的指令，但是，对于大多数分支和循环，80x86是用那些比if或goto语句更简单、甚至更原始的语句来完成。本章旨在论述if-then、if-then-else、while、until和for等语言结构在机器上是如何实现的。

5.1　无条件转移

80x86 jmp（jump）指令与高级语言的goto语句类似，如果用汇编语言编写代码，jmp语句格式如下：

```
jmp  StatementLabel
```

*StatementLabel*语句标号与其他汇编语言程序中语句的名字段一致。回顾一下，当使用标号标识一个可执行语句时，是冒号（:）跟在名字段后，但jmp语句中标号不用冒号。例如，如果在程序应该终止的情况下，有两个选择条件，则代码可以包含如下内容：

```
jmp   quit                                  ; 退出程序
        .
        .
quit: INVOKE ExitProcess, 0                 ; 退出，并返回代码0
        .
        .
```

代码段5-1给出了一个完整的例子：一个重复输入数的程序，在每个数输入之后，显示迄今为止所有数的个数、累计和以及平均值。下面是该例的伪代码结构设计。

```
显示指令;
sum:= 0;
count:= 0;
无条件循环
    提示输入数字;
    输入数字的ASCII码;
    将数字转换成二进制补码形式;
    将数字加到sum中
    将count加1;
```

```
    将count转换成ASCII码;
    显示标号和count;
    将sum转换成ASCII码;
    显示标号和sum;
    average: = sum/count;
    显示标号和average;
重复循环;
```

代码段5-1 无限循环结构的程序

```
; 程序要求输入数字并显示运行中的平均数和总和
; 作者: R. Detmer
; 日期: 1997年9月

.386
.MODEL FLAT

INCLUDE io.h

cr          EQU     0dh     ; 回车
Lf          EQU     0ah     ; 换行

.STACK  4096                ; 保留4096字节的堆栈

.DATA                       ; 为数据保留存储空间
sum         DWORD     ?
explain     BYTE      cr,Lf,"As you input numbers one at a time, this",cr,Lf
            BYTE      "program will report the count of numbers so far,",cr,Lf
            BYTE      "the sum so far, and the average.",cr,Lf,Lf,0
prompt      BYTE      "number?  ",0
number      BYTE      16 DUP (?)
countLabel  BYTE      "count",0
sumLabel    BYTE      "        sum",0
avgLabel    BYTE      "        average",0
value       BYTE      11 DUP (?), 0
nextPrompt  BYTE      cr,Lf,Lf,"next ",0

.CODE                                   ; 主程序代码开始
_start:
            output  explain             ; 初始化指令
            mov     sum,0               ; sum为0
            mov     ebx,0               ; count为0

forever:    output  prompt              ; 提示输入数
            input   number,16           ; 读ASCII码
            atod    number              ; 转换为整数

            add     sum,eax             ; 把数加到sum中
            inc     ebx                 ; count加1

            dtoa    value,ebx           ; 将count转换为ASCII码
            output  countLabel          ; 显示count的标号
            output  value               ; 显示count
```

```
            dtoa    value,sum          ; 将sum转换为ASCII码
            output  sumLabel           ; 显示sum的标号
            output  value              ; 显示sum

            mov     eax,sum            ; 取出sum的值
            cdq                        ; 将sum扩展为64位
            idiv    ebx                ; sum/count
            dtoa    value,eax          ; 将average转换为ASCII码
            output  avgLabel           ; 显示average的标号
            output  value              ; 输出average

            output  nextPrompt         ; 转移，开始下一个立即数
            jmp     forever            ; 重复

PUBLIC _start                          ; 公开入口点
            END
```

该程序必须保存*count*和*sum*的值，并且除了EBX和ECX外，其他所有的寄存器都被input宏/output宏以及（或者）除法指令所使用。*count*的值保存在EBX寄存器中，*sum*的值以双字长数保存在数据段中。注意：初始化时，*sum*是直接通过DWORD指令初始化为0，而不是用mov语句；这样，实现时，代码与设计更一致，但由于*sum*只须初始化一次，因此在时间和空间上有些浪费。

该程序存在一些错误，其中的一个小缺陷是它不能完全四舍五入计算平均值。但是，其主要错误是包含了一个无法退出的*死循环*。事实上，一个程序通常不含结束代码，因为程序无论如何都不会执行到那里。但有一种不用关闭计算机或者重启计算机就能结束这个程序的方法：当提示输入一个数字时，只要按control-C即可。这个方法之所以有用，是因为input宏使用了Kernel32服务（32内核函数）来输入。这个函数为control-C提供了特殊功能。图5-1是这个程序的一次简单运行。

```
As you input numbers one at a time, this
program will report the count of numbers so far,
the sum so far, and the average.

number?  75
count          1        sum        75        average        75

next number?  93
count          2        sum       168        average        84

next number?  78
count          3        sum       246        average        82

next number?  (control-C pressed)
```

图5-1 程序运行实例

代码段5-1程序中的jmp转移控制到jmp语句本身之前，这称为*向后引用*（backward reference），

```
jmp   quit                  ; 退出程序
        .
        .
quit: INVOKE ExitProcess, 0  ; 退出，并返回代码0
```

上述代码说明了向前引用（forward reference）。

80x86 jmp指令分为两类，这些指令改变指令指针寄存器EIP中的值，因此，要执行的下一条指令来自于一个新的地址，而不是紧跟在当前指令后的地址。指令跳转可以发生在段与段之间（intersegment），从而改变代码段寄存器CS和EIP内容。但是，在平面存储模式的编程中不会出现这种情况。因此，这里不讨论这些指令。表5-1列出了段内（intrasegment）转移指令，其中前两个最为常用。

表5-1 jmp指令

类型	时钟周期			字节数	操作码
	386	486	Pentium		
near跳转	7+	3	1	5	E9
short跳转	7+	3	1	2	EB
寄存器间接	10+	5	2	2	FF
存储器间接	10+	5	2	2+	FF

每条相对的转移指令包含相对jmp指令本身的目标位移量，这个位移量加上下一条指令的地址就是转移的目标地址。偏移量是一个有符号数，对于向前引用，它是正数；而对于向后引用，它是负数。对于相对短的转移指令，只存储一个单字节的偏移量。在做加法之前，这个偏移量首先扩展为双字。这种格式还包括32位偏移量。

8位位移量是一个相对较短的转移，可以在jmp指令之前，转移到128字节以内的目标地址；或者在jmp指令之后，转移到127字节以内的目标地址。位移量是通过jmp本身的目标代码后面的字节数来计算的，这是因为在执行一条指令时，逻辑上EIP包含了下一条将要执行的指令的地址。32位位移量是在一个相对近的转移，在jmp指令之前，可转移到2 147 483 648字节内的目标地址；或在jmp指令之后，转移到2 147 483 647字节以内的目标地址。

无论是相对短（short）的还是相对近的（near）转移指令在编写代码时没有什么区别。为了让代码紧凑，如果目标地址范围不大，汇编器使用短的转移指令。如果目标地址超过128个字节，汇编器自动使用近的转移指令。

间接转移指令用32位地址作为目标地址，而不是用位移量。但是这个地址并没有写在指令中，而是保存在寄存器或双字的存储器中。因此，其格式为：

```
jmp   edx
```

指转移到存储于EDX中的地址处。存储器间接形式可使用任一有效的双字内存地址。如果Target在数据项中申明为双字，那么：

```
jmp   Target
```

将转移到存储于Target的双字地址处，而不是指数据项中的这个双字。用寄存器间接寻址，可以用指令：

```
jmp   DWORD  PTR [ebx]
```

转移到一个保存在双字单元的地址处，该双字单元的地址是保存在EBX中，但是，这些间接寻址形式都很少用。

练习5.1

1. 如果执行下面的语句

```
hardLoop:    jmp     hardLoop
```

继续执行“死循环”语句。那么这个语句的目标代码是什么?

2. 指出下列代码段中的每条jmp指令的类型（近跳、短跳、间接寄存器或间接存储器转移）。

```
.DATA
            ...
addrStore   DWORD   ?
            ...
.CODE
            ...
doAgain:
            ... (3条指令)
            jmp  doAgain
            ... (200条指令)
            jmp    doAgain
            ...
            jmp  addrStore
            ...
            jmp  eax
            ...
            jmp  [edi]
```

编程练习5.1

修改代码段5-1中的程序，根据提示输入数字。即，将图5-1中运行的例子改为:

```
As you input numbers one at a time, this program
will report the sum so far and the average.
            number          1  ?   10
            sum            10      average      10
            number          2  ?   50
            sum            60      average      30
```

5.2 条件转移、比较指令和if结构

在80x86机器语言中，条件转移指令可以实现if结构、其他选择结构以及循环结构。条件转移指令有很多，其每条指令格式如下:

```
j-    目标语句
```

其中助记符的最后一部分，定义了执行转移的条件。如果条件满足，则发生转移；否则执行下一条指令（条件转移后面的那一条指令）。

但有些条件转移指令的“条件”是由标志寄存器中标识来设置的，如jcxz/jecxz指令，它们将在5.4节中介绍。例如，指令:

```
jz   endWhile
```

如果零标志ZF为1，则转移到有标号endWhile的语句；否则执行后面的语句。

条件转移指令不改变任何标志位，只根据已设置的标志值做出响应。回顾一下，标志寄存器中的标志位是如何取值的。有些指令（如mov）保持部分或所有标志位不变；有些指令（如add）根据结果值设置某些标志位；其他还有些指令（如div），对某些标志位的改变无法预测，因此也就不能确定标志位的值。

设想一下，将EAX寄存器中的值加到表示账户余额总和的balance中，根据相加后的balance值是负数、零，还是正数，需要考虑三种不同的情况。其伪代码设计：

```
将值加到balance中;

if balance < 0
then
... {balance为负时的程序}
elseif balance = 0
then
... {balance为零时的程序}
else
... {balance为正时的程序}
end if;
```

下面是实现该设计的80x86代码段。

```
            add   balance, eax      ; 将值加到balance中
            jns   elseIfZero        ; balance不为负则转移
            ...                     ; balance为负时的代码
            jmp   endBalanceCheck
elseIfZero: jnz   elsePos           ; balance不为零则转移
            ...                     ; balance为零时的代码
            jmp   endBalanceCheck
elsePos:    ...                     ; balance为正时的代码

endBalanceCheck:
```

add指令设置或清除适当的标志位。在上述代码段中，没有其他指令可以改变标志位。这段程序首先检查（balance<0），下面这条指令完成这项工作：

```
jns   elseIfZero
```

这条指令的意思是，如果符号标志位为0，则转移到elseIfZero；也就是说，如果（balance<0）不成立，则转移到elseIfZero。跟在这条指令后的代码，与该伪代码设计中的第一个then后面的语句是一致的，这段代码的最后一条语句：

```
jmp   endBalanceCheck
```

这条语句是必需的，这样CPU才能跳过适合其他情况的语句。如果第一个条件转移语句转到elseIfZero，那么balance必须是非负的。这样做的目的是检查balance的值是否为0。指令：

```
elseIfZero: jnz   elsePos
```

当零标志位ZF为0时，转移到elsePos。设置标志位的最后一条指令是开头的add指令，因此，balance不为0时发生转移。当Balance为0时，程序必须无条件转移到endBalanceCheck，这时程序再次结束。最后，程序中与else对应的代码在elsePos。代码的最后一块，无须用转移指令来到达endBalanceCheck，因为程序最终会执行到此处。

以上的80x86代码是直接按照程序中语句的顺序来执行的。如果用汇编语言编程，那么，较好的方法是：首先仔细设计，开始编码，然后进行检查，看看是否需要做必要的修改，使之更加有效，这与用许多高级语言编译器的情况一致。很多机器语言程序与翻译过来的高级语言的语句的顺序是一致的。最后，编译程序优化代码，为了提高效率，再重新排列一些语句的顺序。

在前面的代码中，标号endBalanceCheck本身占用一行。技术上，这个标号会指向任何跟在它后面的语句的地址。但是，把它作为当前设计结构中的一部分，而不考虑接下来要做什么，这样更简单。因为，即使改变该结构后面的内容，这个结构的代码仍保持不变。如果下一条语句需要另外的标号，那也完全没有问题——多个标号仍能指向存储器中的同一位置。标号不是目标代码的一部分，因此多余的标号，并不会增加目标代码的长度，也不会增加运行时间。

在根据设计编写代码时，通常想使用if、then、else和endif等作为标号。但是IF、ELSE和ENDIF都是MASM指令，因此，它们不能用做标号。除此之外，IF1、IF2和其他一些可能想到的符号也是备用指令。一个解决办法是用有较长描述符号如上例中的elseIfZero来做标号。由于任何保留字都不能包含下划线，因此，还有一个解决办法是，当原程序中含有并列的关键字时，使用像if_1和if_2为这样的符号作为标号。

术语set（设置）或reset（重置）分别是指对一个标志位置1或0（有时clear代替reset使用）。有许多指令可为标志位置1或清0，不过，使用cmp（比较）指令为标志位赋值可能是最常用的方法。

每条cmp指令都对两个操作数进行比较，并为AF、CF、OF、PF、SF和ZF标志位置1或0。一条cmp指令的惟一任务就是确定标志位的值，这不只是某些功能的一个伴随作用。每条cmp指令形式如下：

```
cmp    操作数1，操作数2
```

通过计算操作数1减去操作数2的值来进行比较，就像一条sub指令。根据差值以及作减法时是否产生借位，确定标志位的设置。cmp指令和sub指令的不同之处在于，cmp指令中，位于操作数1的值不会改变。本书主要讨论的标志位是CF、OF、SF和ZF。减法中有借位时，将进位标志位CF置1；没有借位时，将其清0。有溢出时，将溢出标志位置1；否则清0。如果差是一个负的二进制补码数，则将符号标志位SF置1，否则清0。最后，如果差为0，则零标志位ZF置1；如果差不为0，则将ZF清0。

下面举例说明对一些常见的表示字节长度的数进行比较时，如何设置标志位。回顾一下，减法运算无论对于无符号数还是有符号数（二进制补码）来说，都是一样的。就像一个只有一位的数，它的形式既可以用无符号数表示，也可以用有符号数表示。但对于标志位而言，无论是对无符号数还是有符号数进行比较后，可能会有不同的解释。表5-2列出了有符号数和

无符号数比较时操作数之间的关系。

标志位的值表示了一种什么关系呢？是相等？小于？还是大于？相等的情况比较简单：不管是有符号数还是无符号数，当且仅当操作数1（operand1）和操作数2（operand2）的值相等时，ZF标志位置1。表5-2中的例1就是这种情况，小于和大于的情况则需要进一步的分析。

表 5-2

	操作数1	操作数2	差	标志				转换	
				CF	OF	SF	ZF	有符号数	无符号数
1	3B	3B	00	0	0	0	1	$op1=op2$	$op1=op2$
2	3B	15	26	0	0	0	0	$op1>op2$	$op1>op2$
3	15	3B	DA	1	0	1	0	$op1<op2$	$op1<op2$
4	F9	F6	03	0	0	0	0	$op1>op2$	$op1>op2$
5	F6	F9	FD	1	0	1	0	$op1<op2$	$op1<op2$
6	15	F6	1F	1	0	0	0	$op1>op2$	$op1<op2$
7	F6	15	E1	0	0	1	0	$op1<op2$	$op1>op2$
8	68	A5	C3	1	1	1	0	$op1>op2$	$op1<op2$
9	A5	68	3D	0	1	0	0	$op1<op2$	$op1>op2$

首先考虑小于的情况。当操作数1小于操作数2时，看起来似乎有借位，应该将进位标志位置1。如果操作数是无符号数，这样做逻辑上是正确的。上例中3、5、6和8都是把操作数作为无符号数，且操作数1小于操作数2，这样的情况下，确实是CF = 1。因此，对于无符号数，CF = 0意味着操作数1大于等于操作数2。对于无符号数，严格意义上的不相等是指CF = 0且ZF = 0，也就是操作数1 > 操作数2，而且操作数1 ≠ 操作数2。

例3、5、7和9，把操作数1和操作数2当成有符号数，且操作数1 < 操作数2，这时SF ≠ OF。在剩余例子中，SF = OF，而且操作数1和操作数2是有符号数，操作数1 > 操作数2。对于无符号数，严格意义上的不相等是指SF = OF，而且ZF = 0。也就是操作数1 > 操作数2，而且操作数1 ≠ 操作数2。

表5-2列出了cmp指令。回顾一下表4-5，可以看到在不同的列的内容和sub指令几乎相同。如果第一个操作数在存储器中，由于其结果不需要保存，cmp指令和相应的sub指令相比，需要较少的时钟周期。对某些操作数组合还有一些可选的操作码，表5-3列出了一些MASM6.11使用的操作码。

表5-3 cmp指令

目的操作数	源操作数	时钟周期			字节数	操作码
		386	486	Pentium		
8位寄存器	8位立即数	2	1	1	3	80
16位寄存器	8位立即数	2	1	1	3	83
32位寄存器	8位立即数	2	1	1	3	83
16位寄存器	16位立即数	2	1	1	4	81
32位寄存器	32位立即数	2	1	1	6	81
AL	8位立即数	2	1	1	2	3C
AX	16位立即数	2	1	1	3	3D
EAX	32位立即数	2	1	1	5	3D

（续）

目的操作数	源操作数	时钟周期			字节数	操作码
		386	486	Pentium		
字节存储器	8位立即数	5	2	2	3+	80
字存储器	8位立即数	5	2	2	3+	83
双字存储器	8位立即数	5	2	2	3+	83
字存储器	16位立即数	5	2	2	4+	81
双字存储器	32位立即数	5	2	2	6+	81
8位寄存器	8位寄存器	2	1	1	2	38
16位寄存器	16位寄存器	2	1	1	2	3B
32位寄存器	32位寄存器	2	1	1	2	3B
8位寄存器	字节存储器	6	2	2	2+	3A
16位寄存器	字存储器	6	2	2	2+	3B
32位寄存器	双字存储器	6	2	2	2+	3B
字节存储器	8位寄存器	5	2	2	2+	38
字存储器	16位寄存器	5	2	2	2+	39
双字存储器	32位寄存器	5	2	2	2+	39

还有一些操作码和立即操作数有关，这些可以根据个人选择来编码。假设*pattern*在数据段中指向一个字，那么下面任何一条指令都是允许的:

```
cmp   eax, 356
cmp   pattern, 0d3a6h
cmp   bh, '$'
```

注意: 立即数必须是第二个操作数。指令:

```
cmp   100, total        ; 非法
```

是不允许的，因为第一个操作数是立即数。

最后，表5-4列出了一些条件转移指令。这些指令中，有许多有可供选择的助记符，但这些助记符能生成完全一样的机器代码；并且可用不同的方式，描述同样的设置条件。在同一给定的设计中，通常使用一个助记符比使用不同助记符更加自然。

表5-4 条件转移指令

对有符号操作数比较后的固定用法

助记符	描述	转移标志	操作码	
			短	近
ja	大于则转移	CF = 0而且ZF = 0	77	OF87
jnbe	不小于或等于则转移			
jae	大于或等于则转移	CF = 0	73	OF83
jnb	不小于则转移			
jb	小于则转移	CF = 1	72	OF82
jnae	不大于或等于则转移			
jbe	小于或等于则转移	CF = 1或ZF = 1	76	OF86
jna	不大于则转移			

（续）

对有符号操作数比较后的固定用法

助记符	描述	转移标志	操作码 短	操作码 近
jg	大于则转移	SF = OF而且ZF = 0	7F	OF8F
jnle	不小于或等于则转移			
jge	大于或等于则转移	SF = OF	7D	OF8D
jnl	不小于则转移			
jl	小于则转移	SF ≠ OF	7C	OF8C
jnge	不大于或等于则转移			
jle	小于或等于则转移	SF ≠ OF或ZF = 1	7E	OF8E
jng	不大于则转移			

其他条件转移

助记符	描述	转移标志	操作码 短	操作码 近
je	等于则转移	ZF = 1	74	OF84
jz	为0则转移			
jne	不等于则转移	ZF = 0	75	OF85
jnz	不为0则转移			
js	符号位为1则转移	SF = 1	78	OF88
jns	符号位不为1则转移	SF = 0	79	OF89
jc	有进位则转移	CF = 1	72	OF82
jnc	无进位则转移	CF = 0	73	OF83
jp	为偶数则转移	PF = 1	7A	OF8A
jpe	为偶数则转移			
jnp	为奇数则转移	PF = 0	7B	OF8B
jpo	为奇数则转移			
jo	溢出则转移	OF = 1	70	OF80
jno	无溢出则转移	OF = 0	71	OF81

条件转移指令总是将第一个操作数与第二个操作数进行比较。比如，对于指令jg，也就是“大于则转移”，是指如果操作数1大于操作数2，则转移。

每一条条件转移指令执行时占用一个时钟周期。

任何条件转移指令都不会改变任一标志位的值。每一条指令都有短和近两种转移形式。和短的无条件转移指令一样，一条短的条件转移可使用单个字节的偏移量，可以控制转移到指令本身后面的127字节的地址，或之前的128字节。一个短的条件转移指令需要两个字节的目标代码：一个给操作码、一个给偏移量。一个近的条件转移指令可使用32位的偏移量，以及2个字节的操作码，因此，其总长度为6个字节。它能把控制转移到向后高达2 147 483 648字节的地址，或向前高达2 147 483 647字节的地址。表5-4列出了条件转移指令中所需的字节数和时钟周期数。

再举一些例子来说明有符号数和无符号数的比较之后，条件转移指令用法的区别。假设在EAX中存储了一个值，当这个值大于100时，需要采取一些措施。如果这个值是无符号数，

那么可以用如下代码:

```
cmp     eax, 100
ja      bigger
```

任何大于00000064_{16}的值，包括大的无符号数80000000_{16}和负的二进制补码数$FFFFFFFF_{16}$之间的值，都可执行转移。如果EAX中的值是有符号数，那么下面的指令:

```
cmp     ax, 100
jg      bigger
```

是合适的。只有当值在00000064和7FFFFFFF之间时，而不是值为负的二进制补码时，才会执行转移。

表5-5 jump需要的时钟周期数和字节数

	时钟周期			字节数
	386	486	Pentium	
short条件跳转	7+, 3	3, 1	1	2
near条件跳转	7+, 3	3, 1	1	6

对80386和80486，执行转移时用较长的时间，不转移时用较短的时间。

现在来看看实现的if结构的三个例子，它们与高级语言的编译器的情况是一致的。首先考虑设计:

```
if value < 10
then
        加1到smallCount;
else
        加1到largeCount;
end if;
```

假设*value*存储在EBX中，并且*smallCount*和*largeCount*指向存储器中的字。下列80x86代码可实现该设计:

```
            cmp   ebx, 10            ; value < 10?
            jnl   elseLarge
            inc   smallCount         ; 加1到small_count
            jmp   endValueCheck
elseLarge:  inc   largeCount         ; 加1到large_count
endValueCheck:
```

注意: 这段代码本身已经很完整了，至于这部分设计之前或之后的整个设计是什么，不需要了解。可是，为了避免完全相同的字和保留字，要注意标号的使用。编译器通常会生成跟在一个顺序数字后包含一个字母的一种标号，但是，大多数情况下，人工编码更好。

现在考虑设计:

```
if(total>100) or (count=10)
then
      把value加到total中
end if;
```

假设*total*和*value*是存储器中的双字，*count*存储在CX寄存器中。实现该设计的汇编语言代码如下:

```
            cmp  total, 100      ; total >= 100?
            jge  addValue
            cmp  cx, 10          ; count = 10?
            jne  endAddCheck
addValue:   mov  ebx, value      ; 复制value
            add  total, ebx      ; 把value加到total
endAddCheck:
```

注意: 这段程序中的or需要两条cmp指令。如果其中任一条件满足，都能执行相加指令。(为什么要用两个语句完成相加过程? 为什么不用add total, value?) 这段代码用了一条快捷的or路径——如果第一个条件成立，那么根本不会检查第二个条件。如果用某些语言来设计代码，即使第一个条件成立，代码还是对一个or操作的两个操作数进行检查。

最后考虑以下设计:

```
if (count >0) and (ch = 退格键)
then
      从count中减去1;
end if;
```

对第三个例子，假设*count*在CX寄存器中，*ch*在AL寄存器中，并且退格键等于它的ASCII码08_{16}。具体程序如下:

```
            cmp  cx, 0           ; count > 0?
            jng  endCheckCh
            cmp  al, backspace   ; ch退格键?
            jne  endCheckCh
            dec  count           ; 从count中减去1
endCheckCh:
```

这个复合条件用了and，因此两个条件必须同时成立才能执行这段程序。这段代码用了一条快捷的and路径——如果第一个条件不成立，那么根本不会检查第二个条件。如果用某些语言来设计代码，即使第一个条件不成立，代码还是对一个and操作的两个操作数都进行检查。

最后以一个简单的游戏程序为例。计算机要求第一个玩家输入一个数字，数字输入后清屏; 然后另一个玩家来猜这个数字，每猜测一次，计算机都提示是“太小了”还是“太大了”，或者是“答对了”。猜对之后，猜过的数字的数目，会显示在屏幕上，并且询问玩家，是否要玩另一个游戏。图5-2给出了该游戏的伪代码设计。

该游戏程序的汇编语言源代码如代码段5-2所示。注意: 第24行换行符的作用是清屏。程序中的循环和选择结构与图5-2的设计紧紧相扣。前面曾讲过until循环是先测试循环，下一节将详细讨论如何实现until和while循环。

如果玩家对提问用“N”或“n”来响应，那么，这个游戏程序的外部until循环就终止。input宏用来获取输入的数字或退出循环的响应。由于多字节目标的地址是第一个字节的地址，因此指令

```
cmp    stringIn,  'n'    ; 响应 = 'n'?
```

是将输入的第一个（很有可能仅此一个）字符和字母“n”进行比较，这并不是对两个字符串的比较。

```
until response='N' or response='n' loop

    一个玩家为target;
    输入target并转换为补码;
    清屏;
    count为0;

    until guess=target loop

        加1到count;
        第二个玩家猜数;
        输入所猜数字并转换为补码;

        if猜对
        then
            显示“答对了”;
        elseif猜的数小
        then
            显示“太小了”;
        else
            显示“太大了”;
        end if;

    end until; {guess=target}

    将count转换为ASCII码;
    显示count;
    显示“是否要再玩一遍？”;
    输入响应;

end until; { response='N' or response='n'}
```

图5-2 游戏程序的设计

代码段5-2 游戏程序

```
; 猜数字游戏的程序
; 作者: R. Detmer
; 日期: 1997年9月

.386
.MODEL FLAT

INCLUDE io.h

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

cr          EQU     0dh     ; 回车
Lf          EQU     0ah     ; 换行
```

```
.STACK  4096                ; 保留4096字节的堆栈

.DATA                       ; 为数据保留存储空间
prompt1     BYTE   cr,Lf,Lf,"Player 1, please enter a number:  ", 0
target      DWORD  ?
clear       BYTE   24 DUP (Lf), 0
prompt2     BYTE   cr,Lf,"Player 2, your guess?   ", 0
stringIn    BYTE   20 DUP (?)
lowOutput   BYTE   "too low", cr, Lf, 0
highOutput  BYTE   "too high", cr, Lf, 0
gotItOutput BYTE   "you got it", cr, Lf, 0
countLabel  BYTE   Lf, "Number of guesses:"
countOut    BYTE   6 DUP (?)
            BYTE   cr, Lf, Lf, Lf, "Do you want to play again?  ",0

.CODE                                   ; 开始主程序代码
_start:

untilDone:  output prompt1              ; 要求第一个玩家输入目标数
            input  stringIn, 20         ; 取数
            atod   stringIn             ; 转换为整数
            mov    target,eax           ; 存储目标数
            output clear                ; 清屏
            mov    cx, 0                ; 记数为0

untilMatch: inc    cx                   ; 递增猜的次数
            output prompt2              ; 要求第二个玩家猜数
            input  stringIn, 20         ; 取数
            atod   stringIn             ; 转换为整数

            cmp    eax, target          ; 比较所猜的数字和目标数
            jne    ifLess               ; 猜的数=目标数?
equal:      output gotItOutput          ; 显示“答对了”
            jmp    endCompare
ifLess:     jnl    isGreater            ; 猜的数<目标数
            output lowOutput            ; 显示“太小了”
            jmp    endCompare
isGreater:  output highOutput           ; 显示“太大了”
endCompare:
            cmp    eax, target          ; 比较所猜的数字和目标数
            jne    untilMatch           ; 再次询问是否猜的数=目标数?

            itoa   countOut, cx         ; 将count转换为ASCII码
            output countLabel           ; 显示标号, count和提示符
            input  stringIn, 20         ; 取出响应
            cmp    stringIn, 'n'        ; 响应='n'?
            je     endUntilDone         ; 是, 则退出
            cmp    stringIn, 'N'        ; 响应='N'?
            jne    untilDone            ; 不是, 则重复
endUntilDone:

            INVOKE ExitProcess, 0       ; 退出, 返回代码0
```

```
PUBLIC _start                    ; 公开代码入口点
        END                      ; 结束
```

练习5.2

1. 假设本题中的每一小题，EAX寄存器包含0000004F，*value*指向的双字是FFFFFF38。确定每个条件转移语句是否可转移到*dest*。

```
(a) cmp  eax, value        (b) cmp  eax, value
    jl   dest                  jb   dest
(c) cmp  eax, 04fh         (d) cmp  eax, 79
    je   dest                  jne  dest
(e) cmp  value, 0          (f) cmp  value, -200
    jbe  dest                  jge  dest
(g) add  eax, 200          (h) add  value, 200
    js   dest                  jz   dest
```

2. 本题中的每一小题都用了if结构，并且给出了汇编语言程序中变量存储方式。写出一段汇编语言代码实现这个设计。

(a) 设计:

```
if count = 0
then
    count: = value;
end if;
```

假设：*count*在ECX中；*value*是存储器中的双字。

(b) 设计:

```
if count>value
then
    count: = 0;
end if;
```

假设：*count*在ECX中；*value*是存储器中的双字。

(c) 设计:

```
if a + b = c
then
    check: = 'Y';
else
    check: = 'N';
end if;
```

假设：其中*a*、*b*和*c*都是存储器中的双字；*check*在AL寄存器中。

(d) 结构:

```
if (value<-1000)or(value>1000)
then
    value: = 0;
end if;
```

假设：value在EDX中。

(e) 结构：

```
if (ch>'a')and(ch<'z')
then
    add 1 to lowerCount;
else
    if (ch>'A')and(ch<'Z')
    then
        add 1 to upperCount;
    else
        add 1 to otherCount;
    end if;
end if;
```

假设：*ch*在AL中；每个*lowerCount*，*upperCount*和*otherCount*都是存储器中的双字。

编程练习5.2

1. 修改代码段5-2的游戏程序，只允许玩家输入0到100之间的数字。新代码段的设计是：

```
until (value>0)and(value<1000) loop
    输入值并转换为二进制补码形式;
    if (value<0)or(value>1000)
    then
        显示"输入0到1000之间的值";
    end if;
end until;
```

2. 修改游戏程序，对于第一个玩家输入的数，只允许第二个玩家猜五次。如果第五次猜数也不正确，则显示“对不起，正确的数是xxx”，并且询问玩家，是否要玩另一个游戏。

5.3 循环结构的实现

大多数程序都有循环结构，常用的循环结构有while、until和for循环。本节讨论如何用80x86汇编语言实现这三种结构。下一节讨论其他一些实现for循环的指令。

一个while循环能用下面的伪代码语言来描述：

```
while 循环条件 loop
    ...{循环体}
end while;
```

首先检查Boolean表达式的循环条件，如果条件成立，则执行循环体。然后再检查循环条件。当Boolean表达式不成立时，执行end while后面的语句。

用80x86实现while循环，大多采用以下形式：

```
while:      .           ; 用来检查Boolean表达式的代码
            .
            .
body:       .           ; 循环体
```

```
                .
                .
            jmp  while    ; 再次检查条件
endWhile:
```

通常要用多条语句来检查Boolean表达式的值。如果确定确定循环条件不成立，则转移到endWhile。否则，要么执行循环体，要么转移到它的标号处。注意：要用jmp语句重新检查循环条件，以决定是否结束循环体。有两个常见的错误，要么忽略这个转移，要么转移到循环体。

由于while是MASM中的保留字，因此，在实际代码中标号while并不允许使用。事实上，MASM 6.11有一条while指示性语句，它简化了while循环代码。但本书没有用到这条语句，因为本书所关心的是在机器语言级如何实现循环结构。

例如，用80x86汇编语言对以下设计进行编码：

```
while (sum < 1000) loop
   ...{循环体}
end while;
```

假设*sum*是存储器中的双字，一种可能的实现方式是：

```
whileSum:    cmp     sum, 1000        ; sum<1000?
             jnl     endWhileSum      ; 如果不是则退出循环
              .                       ; 循环体
              .
              .
             jmp     whileSum         ; 再次检查条件
endWhileSum:
```

语句：

```
jnl     endWhileSum
```

直接实现这个结构。还有一种方法是：

```
jge     endWhileSum
```

如果$sum \geqslant 1000$，则将控制转移到循环的最后。这是由于$sum < 1000$不成立时，恰好不等式$sum \geqslant 1000$成立。但是jnl助记符不用转换不等式，就可以很容易地实现这个结构。

用一个小例子来显示一个完整的循环体。假设判断一个整数，它是一个正的以2为底的对数，其最大整数x使得$2^x \leqslant number$成立。实现该设计的程序如下：

```
x: = 0;
twoTox: = 1;
while twoTox≤number
    twoTox乘以2;
    加1到x;
end while;
从x中减去1;
```

假设*number*是存储器中的双字，下列80x86代码实现这个设计。*twoTox*用EAX寄存器，x用CX寄存器。

```
            mov     cx, 0          ; x: = 0
            mov     eax, 1         ; twoTox: = 1
whileLE:    cmp     eax, number    ; twoTox<=number?
            jnle    endWhileLE     ; 不成立则退出
body:       add     eax, eax       ; twoTox乘以2
            inc     cx             ; 加1到x
            jmp     whileLE        ; 再次检查条件
endWhileLE:
            dec     cx             ; 从x中减去1
```

通常while中的循环条件是复合的，用Boolean运算符and或者or连接两部分。对于and运算，其运算符的两边必须同时为真，循环条件才成立。对于or运算，只有两边的运算同时为假，循环条件才不成立。

修改前面的例子，使其包含一个复合条件。假设对下列结构编码：

```
while (sum<1000)and(count<24) loop
    ...{循环体}
end while;
```

假设*sum*是存储器中的双字，并且*count*的值在CX中。一种实现是：

```
whileSum:   cmp     sum, 1000      ; sum<1000?
            jnl     endWhileSum    ; 不成立则退出
            cmp     cx, 24         ; count< = 24
            jnle    endWhileSum    ; 不成立则退出
             .                     ; 循环体
             .
             .
            jmp     whileSum       ; 再次检查条件
endWhileSum:
```

用or，而不是and，对该例子再次进行修改。

```
while (sum<1000)or(flag = 1) loop
    ...{循环体}
end while;
```

这次，假设*sum*在EAX寄存器中，flag在DH寄存器中，有且只有一个字节。下面是实现这个结构的80x86代码：

```
whileSum:   cmp     eax, 1000      ; sum<1000?
            jl      body           ; 成立则执行循环
            cmp     dh, 1          ; flag = 1?
            jne     endWhileSum    ; 不成立则退出
body:        .                     ; 循环体
             .
             .
            jmp     whileSum       ; 再次检查条件
endWhileSum:
```

注意上面两个例子的区别，对于and，只要复合条件中的任一运算不成立，即退出循环。对于or，只要复合条件中有一个运算成立，即执行循环体。

有时，遇到正常的值时，循环继续；遇到一些条件值时，循环停止。如果要从键盘输入数据，那么可以这样写：

```
从键盘取值;
while (不是条件值) loop
    ...{循环体}
    从键盘取值;
end while;
```

在一些高级语言中，实现代码必须和这个设计完全对应。汇编语言的优点之一是具有更大的灵活性。一个相等的设计是：

```
while (从键盘输入的值不是条件值) loop
    ...{循环体}
end while;
```

该设计不需要用两条不同的指令输入数据。它能用一些高级语言编码，也能用80x86汇编语言编码。

在实现这个设计的具体的例子中，假设把从键盘输入的非负数相加，并把输入的任何负数都看作条件值，那么，该设计可以是：

```
sum: = 0;
while (输入的值不是负数) loop
    把number加到sum中;
end while;
```

假设在数据段中有合适的定义，则80x86代码可以是：

```
              mov     ebx, 0          ; sum: = 0
whileNotNeg:  output  prompt          ; 输入提示符
              input   number, 10      ; 从键盘取数
              atod    number          ; 转换为二进制补码
              js      endWhile        ; 负数，则退出
              add     ebx, eax        ; 把number加到sum中
              jmp     whileNotNeg     ; 取下一个数
endWhile:
```

回顾一下atod宏对符号标志位SF的影响。如果ASCII码转换成EAX寄存器中的负数，则将SF置1，否则清0。

for循环是一个计数器控制循环，在给定范围内，每执行一次循环，计数一次。在一些高级语言中，循环计数不仅可以是整数，还可以是其他类型。在汇编语言计算中，计数通常是整数。一个for循环可以用下面的伪代码描述：

```
for index: = 最初值 to 结束值 loop
    ...{循环体}
end for;
```

一个for循环很容易就能转换到while结构，如下：

```
index : = 最初值;
while index<最初值loop
    ...{循环体}
```

```
        加1到index;
    end while;
```

这样，while结构就能用80x86汇编语言来编码了。

例如，假设要将一个数字集合相加，并且不把任何值当作标记。那么就需要问用户，有多少数字要输入，并且要循环多少次。该设计可以是:

```
提示输入数字的个数;
提示要输入的数字个数tally;
sum: = 0
for count: = 1 to tally loop
    数字提示符;
    输入数字;
    把number加到sum中;
end for;
```

直接定义数据段，下面是该设计的80x86程序:

```
            output   prompt1      ; 提示要输入多少数字
            input    value, 20    ; 取出tally(ASCII)
            atoi     value        ; 转换为二进制补码
            mov      tally, ax    ; 存储tally
            mov      edx, 0       ; sum: = 0
            mov      bx, 1        ; count: = 1
forCount:   cmp      bx, tally    ; count< = tally?
            jnle     endFor       ; 不成立则退出
            output   prompt2      ; 数字提示符
            input    value, 20    ; 取数(ASCII)
            atod     value        ; 转换到二进制补码
            add      edx, eax     ; 把number加到sum中
            inc      bx           ; 加1到count
            jmp      forCount     ; 重复
endFor:
```

对于for循环体，至少能执行一次（例如，最初值 < 最终值），因为计数值的检查是在循环体的最后，而不是在循环体之前。其他一些有关实现for循环的指令将在5.4节中讨论。

前面曾讨论过一些until循环的例子。一般来说，一个until循环可用下列伪代码表示:

```
until 结束条件 loop
    ...{循环体}
end until;
```

循环体至少执行一次，然后检查结束条件。如果不成立，则再次执行循环体。如果成立，则继续执行end until后面的语句。

一个until循环的80x86实现，通常代码段如下:

```
until:      .           ; 循环体开始
            .           .
            .           .
            .           ; 检查结束条件的代码
endUntil:
```

如果检查结束条件，判断其条件值为假，那么就会转移到*until*。如果判断其值为真，则要么转移到*endUntil*，要么转移到*endUntil*标号处。

代码5-2中的游戏程序，包含两个简单的until循环，其中一个含有复合结束条件。其设计为:

```
count: = 0;
until (sum>1000)or(count = 100) loop
    ...{循环体}
    加1到count;
end until;
```

下列80x86代码给出了一种实现方式。假设*sum*是数据段中的字，并且*count*存储在CX中。

```
            mov   cx, 0        ; count: = 0
until:       .                 ; 循环体
             .
             .
            inc   cx           ; 加1到count
            cmp   sum, 1000    ; sun>1000?
            jg    endUntil     ; 如果sun>1000，则退出
            cmp   cx, 100      ; count = 100?
            jne   until        ; 如果count不等于100则继续
endUntil:
```

其他循环结构也能用汇编语言编码，无限循环通常很有用。如果它出现在伪代码中，它总有一条exit loop（退出循环）语句，以转移控制到循环结束，这个循环通常是条件循环——也就是说，要用if语句。下面是一段典型的代码设计。

```
forever loop
    .
    .
    .
    if (response = 's') or (response = 'S')
    then
        exit loop;
    end if;
    .
    .
    .
end loop;
```

假设*response*的值在AL寄存器中，则能用下列80x86汇编语言实现:

```
forever:    .
            .
            .
           cmp    al, 's'        ; response='s'
           je     endLoop        ; 如果是，则退出循环
           cmp    al, 'S'        ; response='S'?
           je     endLoop        ; 如果是，则退出循环
```

```
            .
            .
            .
            jmp     forever          ; 重复循环体
endLoop:
```

练习5.3

1. 本题中的每一小题都含一个while循环。假设*sum*是数据段中的双字，*count*的值在ECX寄存器中。给出相应的80x86代码。

(a)
```
sum: = 0;
count: = 1;
while (sum<1000) loop
  加count到sum中;
  加1到count中;
end while;
```

(b)
```
sum: = 0;
count: = 1;
while (sum<1000)and(count<50) loop
  加count到sum中;
  加1到count中;
end while;
```

(c)
```
sum: = 0;
count: = 100;
while (sum<1000)or(count>0) loop
  加count到sum中;
  从count中减1;
end while;
```

2. 本题中的每一小题都含一个until循环。假设*sum*是数据段中的双字，*count*的值在ECX寄存器中。给出相应的80x86代码，实现以下结构。

(a)
```
sum: = 0;
count: = 1;
until (sum>5000) loop
  加count到sum中;
  加1到count中;
end until;
```

(b)
```
sum: = 0;
count: = 1;
until (sum>5000)or(count = 40) loop
  加count到sum中;
  加1到count中;
end until;
```

(c)
```
sum: = 0;
count: = 1;
until (sum>5000)and(count>40) loop
```

```
        加count到sum中;
        加1到count中;
    end until;
```

3. 本题中的每一小题都含一个for循环。假设*sum*是数据段中的双字，*count*的值在ECX寄存器中。给出相应的80x86代码，实现以下结构。

(a)
```
sum: = 0;
for count: = 1 to 100 loop
    加count到sum中;
end for;
```

(b)
```
sum: = 0;
for count: = -10 to 50 loop
    加count到sum中;
end for;
```

(c)
```
sum: = 1000;
for count: = 100 downto 50 loop
    从sum中减去2*count;
end for;
```

编程练习5.3

1. 写出一段完整的80x86汇编语言程序，从键盘输入的数字，并报告这些数字中的最小值和最大值。实现下列设计，并对输出加上合适的标号。

```
显示"第一个数?";
输入数字;
最小值: = number;
最大值: = number;
while (对"另一个数?"响应是'Y'或'y') loop
    输入数字;
    if (number<最小值)
    then
        最小值: = number;
    end if;
    if (number>最大值)
    then
        最大值: = number;
    end if;
end while;
显示最小值;
显示最大值;
```

2. 写出一段完整的80x86汇编语言程序，使其接收从键盘输入的数字，并报告这些数的总和以及平均值。输入数字的个数事先未知，用-999999作为结束输入的标记值。实现下面的设计，并对输入加上适当的提示，对输出加上合适的标号。

```
sum: = 0;
count: = 0;
```

```
while (从键盘输入的数字≠-999999) loop
    加number到sum中;
    加1到count中;
end while;

if  (count = 0)
then
    显示"没有数字输入";
else
    average: = sum/count;
    显示sum和average;
end if;
```

3. 写出一段完整的80x86汇编语言程序，用来分析考试成绩。程序将输入一个未知的考试成绩，如果输入分数为负，则结束输入。然后报告As（90～100）、Bs（80～89）、Cs（70～79）、Ds（60～69）和Fs（60以下）的数目。实现下列设计，给输入加上适当的提示。

```
ACount: = 0;
BCount: = 0;
CCount: = 0;
DCount: = 0;
FCount: = 0;

while(从键盘输入的成绩>0) loop
    if  (成绩>90)
    then
        加1到ACount;
    elseif (成绩>80)
    then
        加1到BCount;
    elseif (成绩>70)
    then
        加1到CCount;
    elseif (成绩>60)
    then
        加1到DCount;
    else
        加1到FCount;
    end if;
end while;

display "Number of As", ACount;
display "Number of Bs", BCount;
display "Number of Cs", CCount;
display "Number of Ds", DCount;
display "Number of Fs", FCount;
```

4. 两个非负整数的最大公约数是这两个数的最大除数。下面的算法找出*number*1和*number*2的最大公约数。

```
gcd: = number1;
remainder: = number2;

until (remainder = 0) loop
    dividend: = gcd;
    gcd: = remainder;
    remainder: = dividend mod gcd;
end until;
```

写出一段完整的80x86汇编语言程序，实现下面的设计，给输入适当的提示，并给输出合适的标号。

```
until (number1>0) loop
    input number1;
end until;

until (number2>0) loop
    input number2;
end until;

find gcd of number1 and number2; (见以上设计)
display gcd;
```

5. 给出一段完整的80x86汇编语言程序，用来模拟一个简单的计数器。这个计数器能做加法和减法运算，并能清除累加值或退出。实现下列设计。

```
total: = 0;

forever loop
    显示"数字?";
    输入数字;

    显示"运算 (+, -, c或q)?";
    输入运算;

    if (运算 = '+')
    then
        加number到total中;
    elseif (运算 = '-')
    then
        从total中减去number;
    elseif (运算 = 'c') or (运算 = 'C')
    then
        total: = 0;
    elseif (运算 = 'q') or (运算 = 'Q')
    then
        退出循环;
    else
        显示"未知运算";
        end if;

        显示"total", total;
    end loop;
```

5.4 汇编语言中的for循环

通常要执行的循环体的次数是已知的，可以在写程序时，用常量来表示循环次数；或者在执行循环之前，用已赋值的变量值表示循环次数。对编写这样的循环，for循环结构是最理想的。

上一节提到如何将一个for循环转换为一个while循环。这个方法很有用，并且也是实现for循环最常用的方法。但是，80x86微处理器还有一些指令，它们可使某些for循环编码变得更简单。

考虑下面两个for循环，第一个向上记数，第二个向下记数。

```
for index: = 1 to count loop
    ...{循环体}
end for;
```

第二个for循环

```
for index: = count downto 1 loop
    ...{循环体}
end for;
```

每个循环体执行的次数是*count*。如果*index*的值不需要在循环体内显示或计算，那么向下记数的循环和向上记数的循环是一样的，尽管向下循环的设计有些不自然，向下记数的for循环很容易用80x86汇编语言中的loop指令实现。

循环指令格式如下:

```
loop     statementLabel
```

其中*statementLabel*1是距离loop指令较短偏移量的语句标号（向后128字节或向前127字节）。loop指令执行下列动作:

- ECX中的值减少
- 如果ECX中新的值是0，那么继续执行循环指令下面的语句
- 如果ECX中新的值不是0，那么执行转移指令到*statementLabel*1

除了loop指令，还有两种不太常用的条件循环指令。表5-6总结了这三种循环指令的特点，每种都需要2个字节的目标代码，第一个字节是操作码，第二个字节是到目标语句的偏移量。80486和Pentium的指令有两个时钟数。第一个表示不发生转移时，需要的时钟周期数；第二个表示发生转移时，需要的时钟周期数。80386的情况比较复杂，但也有两段独立的执行时间。这些指令都不会改变任何标志位。

表5-6 loop指令

助记符	时钟周期			字节数	操作码
	386	486	Pentium		
loop	11+	6/7	5/6	2	E2
loope/loopz	11+	6/9	7/8	2	E1
loopne/loopnz	11+	6/9	7/8	2	E0

虽然ECX寄存器是一个通用寄存器，但是在loop指令和其他一些指令中，它作为一个计

数器有特殊地位。在这些指令中，没有任何寄存器可以替代ECX。实际上，这意味着编写循环代码时，ECX要么不能用作其他用途，要么在执行循环指令前，把计数器的值放在ECX中。否则的话，应该把计数器的值保存在其他地方，以释放ECX，以便其他的循环体可以使用它。

向下记数的for循环结构:

```
for count: = 20 downto 1 loop
    ...{循环体}
end for;
```

用80x86汇编语言编码如下:

```
            mov     ecx, 20         ; 重复数
forCount:    .                      ; 循环体
             .
             .
            loop    forCount        ; 重复循环体20次
```

第一次执行循环体时，ECX寄存器中的计数值是20，通过loop指令减到19。19不等于0，因此，控制转移到标号为forCount的循环体开头。第二次执行循环体时，ECX寄存器中的值还是19。最后一次执行循环体时，ECX中的值是1。loop指令将这个值减为0，循环不再转移到forCount处。

标志for循环体的明显的记号为for，但是，它是MASM中的保留字，用来简化for循环编码的指令。再强调一次，本文重点是了解计算机如何在机器层工作，因此，不必用这条指令。

现在，假设*number*指向的存储器中的双字是循环的执行次数，实现向下记数的for循环的80x86代码可以是:

```
            mov     ecx, number     ; 重复数
forIndex:    .                      ; 循环体
             .
             .
            loop    forIndex        ; 重复循环体number次
```

只有当*number*中的值不为0时，这段代码才是可靠的。如果*number*中的值为0，则执行循环体时，0被减为FFFFFFFF（做这个减法需要一个借位），再执行一次循环体，FFFFFFFF减为FFFFFFFE，以此类推。在ECX中的值回到0之前，循环体要执行4 294 967 296次！为了避免这样的问题，代码可以写为:

```
            mov     ecx, number     ; 重复数
            cmp     ecx, 0          ; number = 0?
            je      endFor          ; 如果number = 0，则跳过循环
forIndex:    .                      ; 循环体
             .
             .
            loop    forIndex        ; 重复循环体number次
endFor:
```

如果*number*中的数是一个有符号数，并且是负的，那么:

```
jle    endFor        ; 如果number< = 0，则跳过循环
```

是一个更合适的条件转移。

还有一种方法保证for循环可靠，那就是当ECX中的值为0时，循环不会执行。80x86指令，设置了一条jecxz条件转移指令，如果ECX寄存器中的值为0，则转移到它的目的地。使用jecxz指令时，上面的例子可以这样编码:

```
            mov     ecx, number     ; 重复数
            jecxz   endFor          ; 如果number = 0, 则跳过循环
forIndex:    .                      ; 循环体
             .
             .
            loop    forIndex        ; 重复循环体number次
endFor:
```

还有一条jcxz指令是用来检查CX寄存器，而不是ECX寄存器。两条指令都是2个字节长。操作码E3加上一个字节的偏移量；前缀字节67用来区分是16位长度还是32位长度。和其他条件转移指令一样，jcxz/jecxz不影响任何标志位的值。它们执行时，需要较长的时间。在发生转移时，Pentium需要6个时钟周期（如果ECX中的值是0）；否则，执行下一条指令，需要5个时钟周期。

当循环体长度大于127字节时，jecxz指令可用于实现向下记数的for循环，但对于loop指令的一个字节的偏移量来说，循环体长度太长了。例如，下列结构:

```
for counter: = 50 downto 1 loop
    ...{循环体}
end for;
```

能够这样编码:

```
            mov     ecx, 50         ; 重复数
forCounter:  .                      ; 循环体
             .
             .
            dec     ecx             ; 循环计数器递减
            jecxz   endFor          ; 如果counter = 0, 则退出
            jmp     forCounter      ; 否则执行循环体
endFor:
```

但是，由于dec指令将零标志ZF置1或清0，所以可以使用相对较快的条件转移:

```
jz    endFor
```

来代替jecxz指令。

即使for循环计数值增加，且必须在循环体内使用，用一个loop语句来实现for循环还是很方便的。当一个独立的计数器用做循环计数器时，loop语句用ECX控制循环次数。

例如，实现以下for循环:

```
for index: = 1 to 50 loop
    ...{使用index的循环体}
end for;
```

ECX寄存器可用来存储从1到50的*index*计数，同时ECX寄存器从50向下计数，直到1。

```
            mov     ebx, 1        ; index: = 1
            mov     ecx, 50       ; 循环的重复次数
forNbr:      .
             .                    ; 把EBX中的值用做index
             .
            inc     ebx           ; 加1到index
            loop    forNbr        ; 重复
```

表5-6列出了loop循环指令的两种变体：loopz/loope和loopnz/loopne。其中每种都以递减ECX中的计数值来完成循环工作。但是，这些指令都要检查零标志位ZF的值和ECX寄存器中新的值，根据检查结果判断是否需要转移到目的地址。对于loopz/loope指令，当ECX中新的值为非0时，且零标志ZF为1时，发生转移；而对于loopnz/loopne指令，当ECX中新的值为非0时，而且零标志被清0（ZF = 0）时，发生转移。

在某些特定情况下，loopz和loopnz指令很有用。一些编程语言允许如下形式的循环结构，如：

```
for year: = 10 downto 1 until balance = 0 loop
    ...{循环体}
end for;
```

这种结构形式令人困惑，因为无论用哪个循环控制都可以终止循环。也就是说，循环体要执行10次（for year = 10, 9, …, 1）。但如果循环体最底部的条件*balance* = 0成立，那么，循环执行不到10次就结束了。如果*balance*的值在EBX寄存器中，则能使用下面的80x86代码：

```
            mov       ecx, 10        ; 重复的最大值
forYear:     .                       ; 循环体
             .
             .
            cmp       ebx, 0         ; balance = 0?
            loopne    forYear        ; 如果balance不为0，则重复10次
```

练习5.4

1. 本题中的每小题，都是用loop循环语句实现for循环。每个循环体执行多少次？

(a)

```
            mov     ecx, 10
forA:        .
             .                    ; 循环体
             .
            loop    forA
```

(b)

```
            mov     ecx, 1
forB:        .
             .                    ; 循环体
             .
            loop    forB
```

(c)

```
            mov     ecx, 0
```

```
forC:    .
         .                    ; 循环体
         .
         loop   forC
```

(d)

```
         mov    ecx, -1
forD:    .
         .                    ; 循环体
         .
         loop   forD
```

2. 本题中的每小题，都有一个for循环。假设*sum*是数据段中的双字。给出一段用loop语句实现该设计的80x86代码，用宏dtoa和output显示。假设数据段包含:

```
ASCIIcount    BYTE    11 DUP (?)
ASCIIsum      BYTE    11 DUP (?)
              BYTE    13, 10, 0        ; 回车，换行
```

(a)
```
sum: = 0;
for count: = 50 downto 1 loop
    add count to sum;
    display count, sum;
end for;
```

(b)
```
sum: = 0;
for count: = 1 to 50 loop
    add count to sum;
    display count, sum;
end for;
```

(c)
```
sum: = 0;
for count: = 1 to 50 loop
    add (2*count - 1) to sum;
    display count, sum;
end for;
```

编程练习5.4

1. 写出一段完整的80x86程序，实现输入一个正整数值N后，用两栏格式显示从1到N和它们的平方。如:

数字	平方
1	1
2	4
3	9
4	16
5	25

2. 一个毕达哥拉斯三角形的三个边是由三个正整数A、B和C组成，从而$A^2 + B^2 = C^2$。例如，数字3、4、5，由于9 + 16 = 25，而形成一个毕达哥拉斯三角形。写出一段完整的80x86程序，实现输入一个值给C，然后显示值为C时，所有可能的毕达哥拉斯三角形。例如，如果输入5作为C的值，那么输出可以是:

```
A        B        C
3        4        5
4        3        5
```

5.5 数组

程序常用数组存储数据值的集合，通常用循环控制数组中的数据。这一节讨论一种用80x86汇编语言访问一维数组的方法，其他方法将在第9章讨论存储器寻址模式时再考虑。

本节提供了一个实现以下设计的完整程序。该程序首先从键盘输入一组正数集，计算输入的个数，并将它们存储在一个数组中。然后，浏览存储在数组中的数，计算这些数的平均值，把累加和放在*sum*中。最后，再次浏览数组中的数，显示比平均数大的数。当然，前两个循环可以结合在一起，在输入数字的同时，可以计算总和。但通用的编程思想是任务分隔得越独立，代码就越清晰。因此，只有当需要节约运行时间或目标代码的字节数时，才将它们结合在一起。

```
nbrElts: = 0;          {将数字输入数组}
取出数组中第一项的地址;

while (从键盘输入的数字>0) loop
    将数字转化为二进制补码;
    在数组中的地址处存储数字;
    加1到nbrElts;
    取出数组中下一项的地址;
end while;

sum: = 0;              {找到总和及平均数}
取出数组中第一项的地址;

for count: = nbrElts downto 1 loop
    加数组中地址处的双字到sum;
    取出数组中下一项的地址;
end for;

平均数: = 总和/nbrElts;
显示平均数;

取出数组中第一项的地址;    {列出最大的数}

for count: = nbrElts downto 1 loop
    if数组的双字>平均数
    then
        转换双字为ASCII码;
        显示值;
    end if;
    取出数组中下一项的地址;
end for;
```

该伪代码设计有一些奇怪的指令，如“取出数组中第一项的地址”和“取出数组中下一

项的地址”，这些指令反映了特殊的汇编语言实现方式，如果现有的任务要求顺序移动数组中的数，那么用这种方式能很好地完成任务。有关80x86寄存器间接寻址的特点在第3章已经讨论过了，本例使用EBX寄存器保存当前访问的字的地址。[ebx]是EBX寄存器中存储的地址所指的双字，而不是EBX寄存器本身存储的双字。80x86中，任何通用寄存器EAX、EBX、ECX、EDX以及索引寄存器EDI和ESI都可作为指针来使用。其中ESI和EDI寄存器常为串所用，串通常是字符数组，有关串的操作会在第7章讨论。该程序实现如代码段5-3所示。

代码段5-3 数组程序

```
; 输入一组数字集合
; 报出它们的平均数以及在平均数以上的数
; 作者: R. Detmer
; 日期: 1997年9月

.386
.MODEL FLAT

INCLUDE io.h

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

cr          EQU     0dh     ; 回车
Lf          EQU     0ah     ; 换行
maxNbrs     EQU     100     ; 数组的大小

.STACK      4096

.DATA
directions  BYTE     cr, Lf, 'You may enter up to 100 numbers'
            BYTE     ' one at a time.',cr,Lf
            BYTE     'Use any negative number to terminate
                     input.',cr,Lf,Lf
            BYTE     'This program will then report the average and
                     list',cr,Lf
            BYTE     'those numbers which are above the
                     average.',cr,Lf,Lf,Lf,0
prompt      BYTE     'Number?  ',0
number      BYTE     20 DUP (?)
nbrArray    DWORD    maxNbrs DUP (?)
nbrElts     DWORD    ?
avgLabel    BYTE     cr,Lf,Lf,'The average is'
outValue    BYTE     11 DUP (?), cr,Lf,0
aboveLabel  BYTE     cr,Lf,'Above average:',cr,Lf,Lf,0

.CODE
_start:
; 将数字输入数组

            output directions     ; 显示立即数
            mov    nbrElts,0      ; nbrElts:=0
            lea    ebx,nbrArray   ; 取出nbrArray的地址
```

```
whilePos:   output prompt          ; 数字提示符
            input  number,20       ; 取数
            atod   number          ; 转换为整数
            jng    endWhile        ; 如果不是正数，则退出
            mov    [ebx],eax       ; 存储数组中的数字
            inc    nbrElts         ; 加1到nbrElts
            add    ebx,4           ; 取出数组中下一项的地址
            jmp    whilePos        ; 重复
endWhile:

; 找出总和及平均数

            mov    eax,0           ; sum:=0
            lea    ebx,nbrArray    ; 取出nbrArray的地址
            mov    ecx,nbrElts     ; count:=nbrElts

            jecxz  quit            ; 不是数字则退出
forCount1:  add    eax,[ebx]       ; 加number到sum中
            add    ebx,4           ; 取出数组中下一项的地址
            loop   forCount1       ; 重复nbrElts次

            cdq                    ; 扩展sum到四倍字长
            idiv   nbrElts         ; 计算平均数
            dtoa   outValue,eax    ; 将平均数转换为ASCII码
            output avgLabel        ; 输出标号和平均数
            output aboveLabel      ; 输出大的数字的标号

; 显示在平均数以上的数字
            lea    ebx,nbrArray    ; 取出nbrArray的地址
            mov    ecx,nbrElts     ; count:=nbrElts

forCount2:  cmp    [ebx],eax       ; 双字>平均数
            jng    endIfBig        ; 如果平均数不小，则继续
            dtoa   outValue,[ebx]  ; 将数组中的值转换为ASCII码
            output outValue        ; 显示值
endIfBig:
            add    ebx,4           ; 取出数组中下一项的地址
            loop   forCount2       ; 重复

quit:       INVOKE ExitProcess, 0  ; 返回代码0，退出

PUBLIC _start                      ; 公开程序入口点
            END                    ; 源代码结束
```

伪代码语句“取出数组中第一项的地址”可以用下面的80x86语句实现：

```
lea    ebx, nbrArray
```

助记符lea，代表的是“取有效地址”。lea指令格式如下：

```
lea    目的地址，源数据
```

目的地址通常是一个32位的通用寄存器，源数据是存储器的值。源地址放在寄存器中（mov指令与此相反，mov目的地址，源数据，其中源地址中的值复制到目的地址）。lea指令的操作码是8D，Pentium处理器中占用一个时钟周期，80486中占用1或2个时钟周期，80386中占用2

个时钟周期。

语句“取出数组中下一项的地址”，用下面的80x86语句实现:

```
add        ebx, 4
```

由于每个双字占用4个字节的存储空间，因此，数组中当前元素的地址加4后就是数组中的下一个元素的地址。

如果想用高级语言为这段程序编码，那么前两个循环可以设计为:

```
nbrElts: = 0;            {输入数字到数组中}
while 从键盘输入的数字>0 loop
  加1到nbrElts;
  将数字存储到nbrSrray[nbrElts];
end while;

sum: = 0;                {找到总和及平均数}
for count: = 1 to nbrElts loop
  加nbrArray[count]到sum;
end for;
```

该设计利用了数组的一个主要特点，即只需给出数组元素的索引，那么任何元素可在任何时候访问，数组元素无需顺序访问。这种随机访问可用寄存器间接寻址来完成。例如，“add nbrArray[count] to sum”，可用下面的语句实现，假设同一个寄存器的用法和以前一样: *count*用ECX寄存器，*sum*用EAX寄存器。

```
mov  edx, ecx          ; count
dec  edx               ; count-1
add  edx, edx          ; 2* (count-1)
add  edx, edx          ; 4* (count-1)
lea  ebx, nbrArray     ; 开始数组的寻址
add  ebx, edx          ; nbrArray[count]的寻址
add  eax, [ebx]        ; 加array[count]到sum
```

上述方法计算所要访问的数组元素前的字节数，并将这个数加到起始地址上。还有很多有效的方法可用来直接访问数组中的元素，这些将在以后的章节中讨论。

练习5.5

1. 修改代码段5-3中的程序，加上一个循环: 显示数组中比平均数小的元素（和平均数相等的数在任何循环中都不显示）。
2. 修改代码段5-3中的程序，显示所有与平均数的差值在±5之间（包含±5）的数，用这样一个循环，代替最后一个循环。
3. 修改代码段5-3中的程序，加上一个循环: 按从大到小的顺序显示所有的数。（提示: 找到*nbrArray*[*nbrElts*]的地址。首先显示这个地址上的元素，然后重复减4，直到所有的元素都已显示。）
4. 修改代码段5-3中的程序，以保证用户最多给出*maxNbr*个值。

编程练习5.5

1. 通常需要在一个数组中搜索一个给定的值。写出一段完整的程序，要求输入整数集，然后顺序查找存储在数组中的值，实现以下设计。

```
nbrElts: = 0;
取出数组中第一项的地址;
while (从键盘输入的数字>0) loop
  将数字转换为二进制补码;
  存储数组中地址处的数字;
  加1到nbrElts;
  取出数组中下一项的地址;
end while;
until(响应 = 'N')或(响应 = 'n')
  显示"寻找?";
  输入关键值;
  将关键值转换为二进制代码;
  取出数组中第一项的地址;
  count:= 1;
  forever loop
    if count>nbrElts
    then
      显示关键值, "不在数组中";
      exit loop;
    end if;
    if关键值为数组中的当前值
    then
      显示关键值, "是元素", 计数;
      exit loop;
    end if;
    加1到count;
    取出数组中下一项的地址;
  end loop;
  显示"寻找另一个数字?";
  输入响应;
end until;
```

2. 上面的编程练习1，给出了一种查找数组的方法。另一种方法是将需要查找的数放在数组的最后。是否能查找到这个值，取决于这个值是在*nbrElts*位置之前还是之后找到。用这个方法，写出一段完整的程序。该设计除了查找循环体以外，和练习1是一样的。查找循环体部分可替换如下。

```
until(响应 = 'N')或(响应 = 'n')
  显示"寻找?";
  输入关键值;
  将关键值转换为二进制代码;
  存储数组中nbrElts + 1地址处的关键值;
  取出数组中第一项的地址;
  count:= 1;
```

```
    while关键值不等于当前数组中的元素loop
      加1到count;
      取出数组中下一项的地址;
    end while;

    if count>nbrElts
    then
      显示关键值, "不在数组中";
      exit loop;
    else
      显示关键值, "是元素", 计数;
      exit loop;
    end if;

    显示"寻找另一个数字?";
    输入响应;
  end until;
```

3. 有很多方法可以判断素数。下面是一个找出前100个素数的设计。用80x86汇编语言实现这个设计。

```
prime[1]为2; {第一个素数数字}
prime[2]为3; {第二个素数数字}
PrimeCount:= 2;
candidate:= 4; {新素数的第一个candidate}
while primeCount<100 loop
  index:= 1;
  while (index<primeCount并且(prime[index]不能除尽candidate)loop
    加1到index;
  end while;
  if (index>primeCount)
  then {不存在素数能除尽candidate, 所以它是一个新的素数}
  加1到primeCount;
    prime[primeCount]: = candidate;
  end if;
  加1到候选值;
end while;

显示"Prime Numbers";
for index:= 1 to 100 loop{每行显示5个数字}
  显示prime[index];
  if index能被5除尽, 则跳到新的一行;
  end if;
end for;
```

5.6 其他: 流水线

第2章讨论了中央处理单元的基本操作周期:

- 从存储器中取出一条指令

• 对这条指令解码

• 执行这条指令

一个CPU必须有实现这些功能的电路系统。为了提高CPU的运行速度，计算机设计者所做的工作之一是设计多阶段CPU，使每阶段几乎都能独立执行这些（或其他）操作。

CPU的第一阶段完成从存储器中取出下一条指令的工作，也许是只有充分解码后，才能知道指令的字节数，并且更新程序计数器PC。第一阶段传递信息到第二阶段，第二阶段的工作是完成指令的解码，可能还计算一些操作数地址。同时，第一阶段可以从存储器中取出下一条指令。第二阶段可将一条完全解码的指令传送到第三阶段运行。与此同时，第一阶段可能已把它的第二条指令传递给第二阶段，这样第一阶段就能取第三条指令了，这种设计称为流水线。如果流水线一直是满的话，那么，相比每条指令在执行下一条指令之前必须要完成整个取指令——解码——执行而言，整个CPU的运行速度要快三倍。

表5-7说明了流水线的操作。正在处理的指令分三行水平排列，并用标号1、2、3表示阶段数，水平轴表示时间。在任何给定的时间内，这三条指令中总有部分在执行。

表5-7 流水线中的指令

CPU阶段	正在执行的指令										
1	1	2	3	4	5	6	7	8	9	10	11
2		1	2	3	4	5	6	7	8	9	10
3			1	2	3	4	5	6	7	8	9
时间间隔	1	2	3	4	5	6	7	8	9	10	11

一个流水线的CPU并不像上述那样简单。如果出现这样一个问题：第二阶段需要计算一个地址，这个地址是基于以前的指令在第三阶段修改的一个寄存器的内容；该寄存器也许不包含正确的地址。为避免这样的问题，通常在设计CPU时，在流水线中设置一个“洞”。

当CPU执行条件转移指令时，会出现更严重的问题。处理条件转移时，对于要执行的两种指令顺序，在条件本身还没有被最后一个阶段处理前，CPU不能确定到底要按哪种顺序执行。前面的阶段可能正在执行一组指令流，但CPU不得不放弃当前的工作，从另一组指令流的开始重新装满流水线。

本章小结

本章讨论了能用来实现许多高级设计或语言的80x86指令，包括if语句以及各种不同的循环结构和数组。

jmp指令无条件将控制转移到目标语句，它有几种形式，包括短跳，是指在jmp指令之前，转移到128字节以内的目标地址；或者在jmp指令之后，转移到127字节以内的目标地址。还包括到距离目标地址32位偏移量的近跳。jmp指令可用在不同循环结构之中。典型的是将控制转移到循环的开始。jmp指令还可用在if-then-else结构中，在then代码的最后，将控制转移到endif。这样，就不会执行else代码了。jmp语句相当于大多数高级语言中的goto语句。

条件转移语句检查标志寄存器中一个或多个标志位的设置，然后根据标志值判断是转移到目标语句，还是执行下面的指令。条件转移指令按照偏移量，有短跳和近跳两种形式。条件转移指令有很多，在if语句和循环中，它常和比较指令一起检查Boolean条件。

cmp（比较）指令的惟一目的就是将EFLAGS寄存器中的标志位置1或清0。每条指令比较两个操作数，并设置标志位的值。通过从第一个操作数中减去第二个操作数，来完成这两个数的比较。虽然它用了sub指令，但并不保留差值。比较指令通常用在条件转移指令之前。

循环结构如while、until和for循环，能够用比较、转移和条件转移指令实现。loop指令还提供了另外一种方法实现for循环。为了使用loop指令，在循环开始之前，把一个计数器放在ECX寄存器中。loop指令本身是在循环体的底部，它递减ECX中的值。如果新的值不是零，则转移控制到目的地（通常是循环体的第一条语句），循环体执行的次数始终放在ECX寄存器中。当计数器的初值为0时，条件转移指令jecxz可用来防止执行循环。

程序中数据段的DUP指令可为一个数组预留存储空间。数组中第一个元素的地址是在寄存器中，第一个元素的地址重复加上数组中元素的长度就可以取出下一个元素。这样，就能顺序访问数组中的元素。当前的元素用寄存器间接寻址获得。通常用lea（取有效地址）指令取出数组的初始地址。

流水线是CPU用多阶段完成的，多阶段是指在同一时间可处理多条指令：在取一条指令的同时，对另一条指令解码，同时执行第三条指令。这样大大提高了CPU的运行速度。

第6章 过　　程

80x86体系结构能够实现那些类似于高级语言的过程，过程有时要使用硬件堆栈。本章首先讨论80x86的堆栈，接着介绍一些重要的过程概念，包括过程调用、返回、参数传递、局部数据和递归等等，最后详述在一个没有硬件堆栈的体系结构中如何实现过程。

6.1 80x86堆栈

本书中的程序用如下的代码分配堆栈:

```
.STACK    4096
```

这条.STACK指令告诉汇编器预留4096个字节的未初始化的存储空间。操作系统初始化ESP指向堆栈中4096个字节的第一个字节的地址。分配堆栈的大小取决于程序的需要。

堆栈常用于压入（push）或弹出（pop）一些字或双字。作为执行调用和返回指令的一部分（见6.2节），压入或弹出操作是自动完成的，它们也可以通过pust和pop指令手工完成。本章将讨论pust和pop指令的机制，考察它们是如何影响堆栈的内容的。

push指令的源代码语法如下:

```
push    源操作数
```

源操作数可以是一个16位寄存器、一个32位寄存器、一个段寄存器、存储器中的一个字、存储器中的一个双字、一个字节的立即数、一个字的立即数或一个双字的立即数。只有一个字节的操作数是立即数，因此，多字节是以一个字节立即数的形式压入堆栈的。表6-1列出了允许使用的立即数的类型。入栈指令常用的助记符是push，但是，如果操作数的大小不明确（只是一个小的立即数），可以用助记符pushw和pushd来分别指定字或双字的操作数。

表6-1　push指令

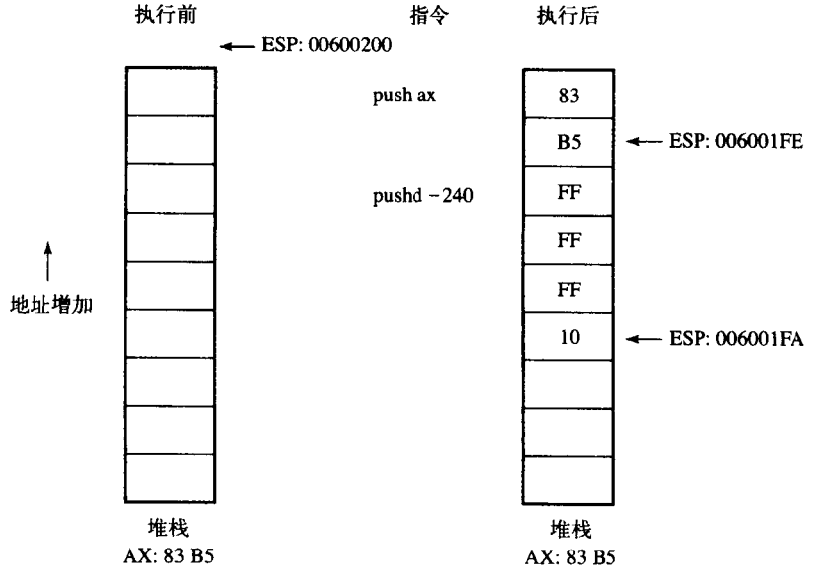
操作数	时钟周期			字节数	操作码
	386	486	Pentium		
寄存器	2	1	1	1	
EAX或AX					50
ECX或CX					51
EDX或DX					52
EBX或BX					53
ESP或SP					54
EBP或BP					55
ESI或SI					56
EDI或DI					57
段寄存器	2	3	1		
CS				1	0E

（续）

操作数	时钟周期			字节数	操作码
	386	486	Pentium		
DS				1	1E
ES				1	06
SS				1	16
FS				2	0FA0
GS				2	0FA8
存储字	5	4	2	2+	FF
存储双字	5	4	2	2+	FF
字节立即数	2	1	1	2	6A
字立即数	2	1	1	3	68
双字立即数	2	1	1	5	68

当一个字的操作数入栈时，堆栈指针ESP将减去2。回顾一下，最初ESP指向的只是被分配的空间的最上面字节的地址。减去2后，ESP指向堆栈的顶部的字。然后，这个操作数被存到了ESP指向的地址，也就是堆栈的顶部。如果是一个双字操作数入栈，执行情况类似，但是堆栈指针ESP将减去4。对于一个字节的立即数的存储需要注意，虽然，指令中是一个单字节，但是，它扩展符号位变为双字，实际上以双字的形式存储在堆栈内。一个字节的操作数节省了三个字节的目标代码，但没有节省执行时的堆栈空间。

这里给出一个执行两条push指令的例子。假设ESP的初始值是00600200。执行第一条push指令后，ESP指向006001FE，然后AX寄存器的内容存放在ESP指向的地址。注意：低字节和高字节在存储器中已预留空间。执行第二条入栈指令后，ESP指向006001FA，并将FFFFFF10（-240_{10}）存入该地址。



随着其他操作数入栈，堆栈指针ESP递减，新的值存入，push指令不影响任何标志位。

注意：堆栈是“向下递减”的，这与常见的软件堆栈相反。还要注意的是，堆栈中最容易得到的值是最后一个被压入堆栈的，存放在ESP所指的地址处。此外，堆栈指针ESP在入栈和过程调用时经常改变。6.3节将考察用EBP寄存器，如何在堆栈中间建立一个固定的参考点。这样，在参考点附近的值能够被读取，而不用弹出要读取值上面所有的值。

出栈指令与入栈指令作用相反，pop指令格式如下：

pop　目的操作数

其中目的操作数指向存储器中一个字或双字、任意一个16位寄存器、任意一个32位寄存器，或者除了CS外的任意一个段寄存器（push指令可用CS寄存器）。对于单字出栈操作，pop指令复制ESP所指的地址中的字，并将它存放到目的地址中，然后ESP加2。双字出栈操作情况类似，但是，ESP加4。表6-2列出了不同的目的操作数的出栈指令。

这个例子说明了出栈指令是如何工作的。对于双字的出栈操作，先把ESP所指地址的内容复制到ECX，然后ESP加4。出栈的操作只是改变了数据在逻辑上的存放位置，并没有改变其在物理上的存放位置。注意：在80x86体系结构中，双字中的字节在存储器中是低字节优先，不过在寄存器ECX中是高字节优先。

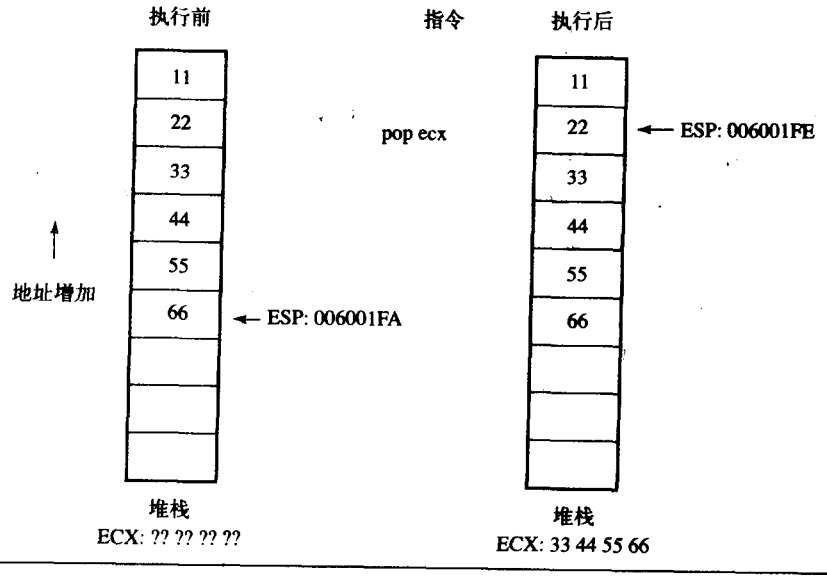


表6-2　pop指令

操作数	时钟周期			字节数	操作码
	386	486	Pentium		
寄存器	4	1	1	1	
EAX或AX					58
ECX或CX					59

（续）

操作数	时钟周期			字节数	操作码
	386	486	Pentium		
EDX或DX					5A
EBX或BX					5B
ESP或SP					5C
EBP或BP					5D
ESI或SI					5E
EDI或DI					5F
段寄存器	7	3	3		
DS				1	1F
ES				1	07
SS				1	17
FS				2	0FA1
GS				2	0FA9
存储字	5	6	3	2+	8F
存储双字	5	6	3	2+	8F

push和pop指令可用来将寄存器中的内容暂时存放在堆栈中。因为之前就注意到：在编程时，寄存器资源并不是很多。例如，如果寄存器EDX已用来存放了某个过程变量，但是，在做除法运算时，被除数必须先存放在EDX-EAX中，为了避免丢失EDX中存储的数据值，必须先将EDX中存放的数据值入栈。

```
push  edx       ; 保存变量
cdq             ; 扩展被除数为双字
idiv  Divisor   ; 除数
pop   edx       ; 恢复变量
```

该例中假定不需要将除法运算的余数存入EDX。如果需要余数的话，在移出堆栈中的数据前，该余数要先复制放到其他某个地方，然后再存入EDX。

如上例所示，push和pop指令经常是成对使用的。在使用堆栈传递参数给过程时，如果堆栈中数据没有被复制到目的地址，那么这些数据就会丢失。

除了普通的push和pop指令之外，还有特定助记符的入栈和出栈标志寄存器，包括pushf（pushfd用于扩展的标志寄存器）和popf（popfd用于扩展的标志寄存器），如表6-3所示，它们一般用于过程代码。显然，popf和popfd指令能够改变标志位的数值；它们是惟一能改变标志位的入栈或出栈指令。

表6-3　pushf和popf指令

指令	时钟周期			字节数	操作码
	386	486	Pentium		
pushf pushfd	4	4	3	1	9C
popf popfd	5	9	4	1	9D

在80x86结构体系中，有些可以使用单一的指令压入或取出所有通用寄存器的内容。pushad指令能依次将数据压入EAX、ECX、EDX、EBX、ESP、EBP、ESI和EDI。压入到ESP的值是所有寄存器被压入前的地址。popad指令按照相反的顺序依次从同样的寄存器中提取数据，只有ESP的值被丢失。按相反顺序取出这些寄存器中的值，以保证pushad和popad是成对使用的，这样，每一个寄存器（除了ESP外）都能恢复它的初值。表6-4列出了所有的入栈和出栈指令，包括压入和取出16位寄存器的pusha和popa指令。

表6-4 所有的push和pop指令

指令	时钟周期			字节数	操作码
	386	486	Pentium		
pusha pushad	18	11	5	1	60
popa popad	24	9	5	1	61

最后，值得注意的是，尽管Intel体系结构允许16位或者32位的数压入堆栈，但是有些操作系统（包括微软的Windows NT）要求在系统调用中所用的参数是双字，也就是说，参数地址必须是4的倍数。堆栈要以双字开始，但是要保持对齐，在系统调用前，应该先将双字压入堆栈。（参见第12章系统调用的例子。）

练习6.1

1. 对下面每条指令，给出操作码、执行的时钟周期数以及目标代码字节数。假设Double是存储器中的双字，时钟周期数采用Pentium系统。

   (a) `push ax`　　(b) `pushd 10`

   (c) `pusha`　　(d) `pop ebx`

   (e) `pop Double`　　(f) `popad`

   (g) `pushf`

2. 下面每一小题，假定给出了指令执行前的值。请给出指令执行后的值。画出堆栈示意图，跟踪指令执行过程。

	执行前	指令	执行后
(a)	ESP: 06 00 10 00 ECX: 01 A2 5B 74	push ecx pushw 10	ESP, ECX
(b)	ESP: 02 00 0B 7C EBX: 12 34 56 78	pushd 20 push ebx	ESP, EBX
(c)	ESP: 00 00 F8 3A EAX: 12 34 56 78	push eax pushw 30 pop bx pop ecx	ESP, EAX, BX, ECX

3. 许多微处理器没有等价于xchg功能的指令。对于这类的系统，下面的指令序列能够用于两个寄存器间数据的交换。

```
push  eax
push  ebx
pop   eax
pop   ebx
```

解释为什么这段指令序列可交换EAX和EBX寄存器的内容。比较执行该指令序列与指令xchg eax, ebx所需的目标代码字节数和时钟周期数。

4. 还有一种可替代xchg指令的方法是:

```
push  eax
mov   eax, ebx
pop   ebx
```

解释为什么这段指令能交换EAX和EBX寄存器的内容。比较执行该指令序列与指令xchg eax, ebx所需的目标代码字节数和时钟周期数。

6.2 过程体、调用和返回

“过程”（procedure）在高级语言中可描述为一个自包含的子程序，主程序或者其他子程序通过包含某些语句能够调用过程，该语句包含了过程名以及与过程形参有关的参数列表。

许多高级语言将执行某个动作的过程和带有返回值的“函数”（function）区分开来。函数与过程类似，但是，函数能够通过在表达式中使用函数名和参数表来调用。此外，它返回一个与函数名有关的值，并且这个值可以被表达式使用。在C/C++语言中，所有的子程序都是函数，而且允许函数没有返回值。

在汇编语言和一些高级语言中，术语“过程”用来描述两种类型的子程序，一种有返回值，一种没有返回值，本书中的过程是基于上述两种类型的。

同高级语言一样，过程在汇编语言中也是很有用的。过程不仅能将程序划分成多个容易处理的任务，而且可以分开写代码，这样可以在单个程序中多次使用这些代码，或者将它们保存起来，为其他程序重用。

本节详述如何编写80x86过程，以及如何使用微软提供的软件进行汇编和链接，包括如何定义一个过程，如何将执行控制传递给过程，以及如何返回调用程序。要考察堆栈是如何保存寄存器内容的，这样，当过程返回到调用者时，几乎所有寄存器的内容都没有改变。同过程有关的还有一些重要概念需要考虑，包括如何传递参数给过程，以及如何在过程体中实现局部变量等等，这些将在以后章节中讨论。

过程的代码总是出现在一条.CODE指令后。每个过程体都包含两条指令：PROC和ENDP，每条指令都有一个标号，标识该过程的名字。用微软的宏汇编器，PROC指令允许指定一些属性；这里只使用其中的一个，即NEAR32。该属性表明过程位于和调用代码同样的代码段内，并且使用32位的地址，这对于平面的32位内存模式编程是很正常的。代码段6-1列出了包含过程*Initialize*的一个程序的相关部分，该过程的主要工作是初始化一些变量，调用程序只给出了一个框架，但是过程*Initialize*的代码很完整。

在代码段6-1中，过程*Initialize*定义在PROC和ENDP内。距离属性NEAR32声明此过程是near（近程）过程。尽管这个例子给出的过程体是放在主程序代码前，但它也可以放在主代码后。回顾一下，程序的执行并不一定是从代码段的第一句开始，标号_start表示第一条要执行的指令。

代码段6-1 过程结构

```
; 过程结构示例
; 作者: R. Detmer
; 日期: 1997年10月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

.STACK  4096                        ; 保留4096字节堆栈

.DATA                               ; 数据保留区
Count1      DWORD   11111111h
Count2      DWORD   22222222h
Total1      DWORD   33333333h
Total2      DWORD   44444444h
;           其他数据

.CODE                               ; 程序代码

Initialize  PROC    NEAR32
            mov     Count1,0        ; Count1为0
            mov     Count2,0        ; Count2为0
            mov     Total1,0        ; Total1为0
            mov     Total2,0        ; Total2为0
            mov     ebx,0           ; 余额为0, 存入ebx寄存器
            ret                     ; 返回
Initialize  ENDP

_start:                             ; 程序入口
            call    Initialize      ; 初始化变量

; - 其他程序任务

            call    Initialize      ; 再初始化变量

; - 更多的程序任务

            INVOKE ExitProcess, 0   ; 退出返回代码0
PUBLIC _start                       ; 公开入口点

END                                 ; 程序结束
```

过程*Initialize*的大部分语句是常用的mov指令。在这个主程序中，有两个地方用了call语句，使用过程可以使主程序代码更简短、更清晰。该过程影响了主程序数据段中定义的双字和EBX寄存器；它没有局部变量。

当主程序执行时，指令：

```
call Initialize
```

将从主程序转去调用过程。主程序两次调用该过程；通常，一个过程可以被调用任意次。返

回指令：

```
ret
```

将控制返回给调用者；在过程中，至少要有一条ret指令，也可以有多条ret指令。假如只有一条ret指令，那么通常这条指令就是该过程的最后一条指令，因为没有这条指令的话，调用语句后的其他指令就无法执行。虽然call指令必须说明它的目的地址，但ret指令不需要，它会将控制转移返回到最近的call指令后的那条指令。80x86使用堆栈来保存返回地址。

在对代码段6-1所示的例程进行汇编、链接和执行后，没有什么可视化信息输出。然而，如果使用类似WinDbg这样的工具跟踪执行的话，就可获得很多信息。图6-1是过程调用前WinDbg的初始窗口信息，注意：ESP的值是0063FE3C。打开的存储器窗口是从地址0063FE30处开始，并且将12字节压入堆栈。EIP寄存器的值是0040103E，这是要执行的第一条指令的地址（第一次调用）。图6-2是这条指令执行后的状态，现在，EIP寄存器的值是00401010，它是过程Initialize中第一条语句的地址。把4个字节压入堆栈后，ESP的值是0063FE38，在这个地址上的存储器中的值为43 10 40 00——也就是00401043，这比第一次调用的地址增加了5个字节。通过检查该程序的代码，可以发现每一条调用指令占用了5个字节的目标代码，因此，00401043就是紧跟在第一条call指令后的指令的地址。

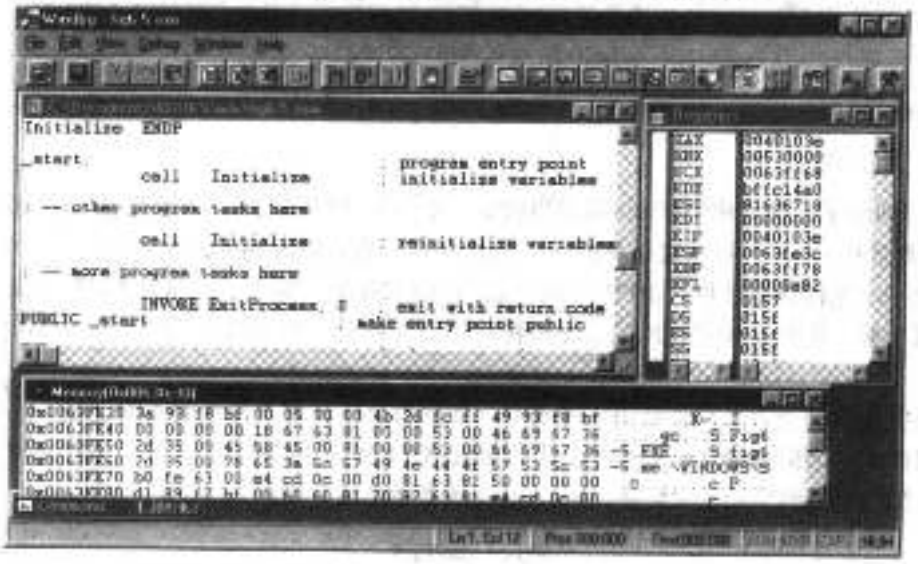


图6-1　过程调用前的状态

通常，调用指令将下条指令（紧跟在调用指令后的指令）的地址压入堆栈，然后转去调用过程代码。近程调用指令将EIP压入堆栈，然后改变EIP的内容，让它指向被调用过程的第一条指令的地址。

从过程返回到主程序，其执行顺序与调用过程的顺序相反，ret指令取出EIP中的数据，这样，下一条要执行的指令就是曾经压入堆栈保存地址的指令。

80x86程序可以使用平面存储模式，也可用分段存储模式。使用分段存储模式，过程和调用它的程序代码可以不在同一个段内，事实上，对于16位分段编程，段长最大为65536字节。所以，过程经常放在不同的段内。80x86体系结构采用远程调用（farcall），转去调用不同内存

段的过程：远程调用将EIP和CS压入堆栈，远程返回从堆栈中取出EIP和CS。对于32位平面内存模式编程，一般采用近程调用的方式。

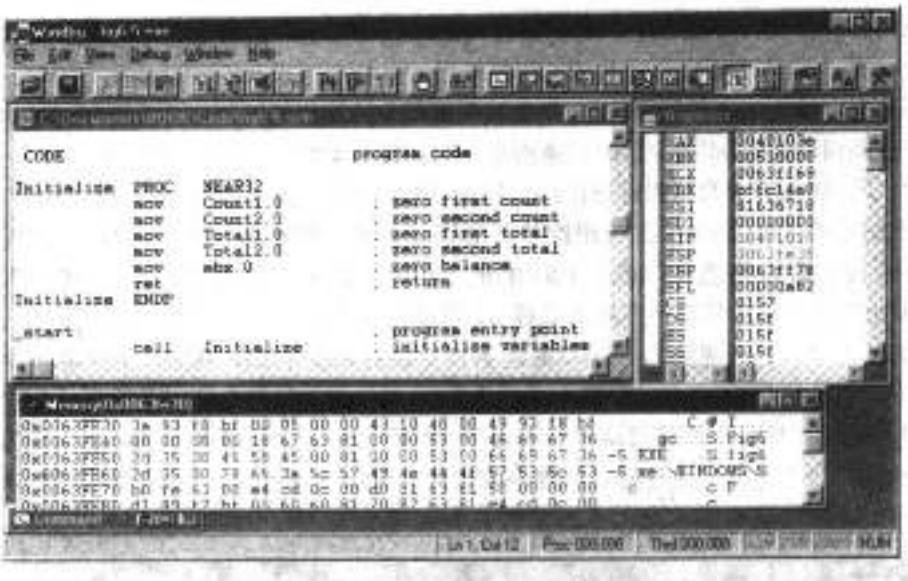


图6-2 过程调用后的状态

80x86调用语句的语法如下：

```
call 目的过程
```

表6-5列出了一些可用的80x86的调用指令，不包括16位形式和以前的系统编程形式。在80386处理器中，"+"符号表示根据下一条指令判断是否需要额外的时钟周期，图6-1中的程序包含一个近程调用过程，由PROC操作数NEAR32指派。通常，通过PROC指令或其他指令以及操作数，汇编器确定目的地址是否指向一个近程或远程调用，调用指令不修改任何标志位。

本书中使用的全部过程都是第一种类型，即近程关系。对于近程关系的过程，汇编器计算32位目的地址的偏移量，E8操作码加上这个偏移量组成了指令的5个字节。过程调用时的控制转移类似于相对跳转指令，但是，EIP中原来的数据要压入堆栈。

表6-5 call指令

操作数	时钟周期			字节数	操作码
	386	486	Pentium		
近程相对	7+	3	1	5	E8
近程间接					FF
使用寄存器	7+	5	2	2	
使用存储器	10+	5	2	2+	
远程直接	17+	18	4	7	9A
远程间接	22+	17	5	6	FF

近程间接调用使用32位寄存器或存储器中的双字。当执行调用时，寄存器或双字的内容

就是被调用过程的地址。这样，一条call指令能够在不同时间调用不同的过程。

所有远程调用必须提供新的CS和EIP中的值，使用远程直接调用时，CS和EIP中的值要写在指令中，操作码加上6个字节。使用远程间接调用时，设置了一个6字节的内存块，该内存块的地址在指令中给出。

返回指令ret用于将控制从过程体返回到调用点。它的基本操作很简单，取出以前存放在堆栈中的地址，然后将该地址放入指令指针寄存器EIP。由于堆栈保存了跟在调用指令后的下一条指令的地址，因此，程序会在该地址处继续执行。近程返回必须恢复EIP的内容，远程返回与远程调用步骤相反，恢复EIP和CS的内容，从堆栈中取出EIP和CS的值。

返回指令有两种形式。较常用的形式是无操作数，其简单代码如下:

```
ret
```

还有一种形式是有一个操作数，代码如下:

```
ret count
```

在完成返回进程（恢复EIP内容，对于远程过程，还要恢复CS内容）的其他步骤后，操作数count加上ESP内容，并将结果存放到ESP中。对于过程调用而言，如果其他值（特别是参数）已经保存在堆栈里，并且在过程结束时，逻辑上清除这些值，那么，这将是很有用的。（参数将在下一节做进一步讨论。）表6-6列出了ret指令的不同形式。

如果一个过程的PROC指示性语句有操作数NEAR32，那么汇编器生成过程的近程调用，并且从该过程近程返回结束。微软的宏汇编器还有用于强制near或far返回的retn（near返回）和retf（far返回）助记符，但这里没有用到这些助记符。

为大型程序创建内存区，需要汇编一个过程或者通过调用不同过程组成的过程组；也就是说，过程和调用程序可以在不同的文件中。这就需要一些额外的步骤。首先，必须汇编这些过程，这样，这些过程名就可在包含这些过程的文件的外部可见。其次，必须让调用程序知道有关外部过程的必要信息。最后，必须链接另外的.OBJ文件以获得一个可执行程序。

表6-6　ret指令

类　型	操 作 数	时钟周期			字 节 数	操 作 码
		386	486	Pentium		
near	无	10+	5	2	1	C3
near	立即数	10+	5	3	3	C2
far	无	18+	13	4	1	CB
far	立即数	18+	14	4	3	CA

PUBLIC指示性语句用于让过程名在包含它们的文件外部可见，这与曾经用于使_start标号（symbol）可见是一样的。通常，语法如下:

```
PUBLIC symbol1 [, symbol2]...
```

一个文件可能包含多条PUBLIC指示性语句。

EXTRN指示性语句提供给调用程序有关外部标号（symbol）的信息。它可以有多个候选项，包括:

```
EXTRN symbol1:type [, symbol2:type]
```

一个文件可能包含多条EXTRN指示性语句。代码段6-2说明给出了带有指示性语句PUBLIC和EXTR的两个过程：Procedure 1和Procedure 2，这两个过程可在主程序的不同文件中汇编。注意，这里需要.386和.MODEL FLAT指示性语句，同时，还可能需要INCLUDE指示性语句。

代码段6-2 外部过程代码

包含过程定义的文件

```
PUBLIC      Procedure1, Procedure2

.CODE

Procedure1  PROC  NEAR32
            ...
Procedure1  ENDP

Procedure2  PROC  NEAR32
            ...
Procedure2  ENDP

            END
```

包含过程调用的文件

```
EXTRN Procedure1:NEAR32, Procedure2:NEAR32
            ...
.CODE
            ...
            call  Procedure1
            ...
            call  Procedure2
            ...
            END
```

上述文件可像主程序一样汇编，每一次汇编产生一个.OBJ文件。为了链接这些文件，只要在链接命令中列出所有的.OBJ文件——用已经单独汇编过的IO.OBJ文件来链接程序。

本节最后以一个计算正整数Nbr平方根的过程为例，该过程需要求出满足SqRt*SqRt≤Nbr的最大整数SqRt。该过程代码如代码段6-3所示，这不是一个能够汇编的完整文件，但该过程代码可以用表6-6中的指示性语句分别进行汇编，或者可以将它放在调用程序的文件中。

代码段6-3 求整数平方根的过程代码

```
; 计算Nbr平方根的过程
; Nbr存入寄存器EAX中
; 返回平方根SqRt存入寄存器EAX中
; 其他寄存器保持不变
; 作者：R. Detmer
; 日期：1997年10月

Root        PROC  NEAR32
            push  ebx           ; 保存寄存器
            push  ecx
```

```
            mov     ebx, 0          ; 平方根SqRt为0
WhileLE:    mov     ecx, ebx        ; 复制SqRt
            imul    ecx, ebx        ; 计算SqRt*SqRt
            cmp     ecx, eax        ; SqRt*SqRt小于等于Nbr吗?
            jnle    EndWhileLE      ; 如果不是，退出
            inc     ebx             ; 否则SqRt加1
            jmp     WhileLE         ; 重复
EndWhileLE:
            dec     ebx             ; SqRt减1
            mov     eax, ebx        ; 返回SqRt，并存入寄存器AX中
            pop     ecx             ; 恢复寄存器
            pop     ebx
            ret                     ; 恢复寄存器
Root        ENDP
```

过程Root实现如下:

```
Sqrt: = 0;
while Sqrt*Sqrt<Nbr loop
加1到SqRt;
end while;
从Sqrt减1;
```

算法采用不断地逼近整数*SqRt*的方法；当尝试值超过正确值时，就取上一个计算的结果作为最后的结果。这不是一个非常有效的算法，但易于实现。

调用程序必须将*Nbr*的值放在EAX寄存器中，下一节讨论更常用的将参数传递给过程的方法。求平方根的过程Root将返回*SqRt*的值，并把它放在EAX寄存器中。由此可见，返回一个整型值的函数常常使用累加器。

除了实现该算法外，该过程在开始时还包含两条push指令，调用返回前，还有两条相应的pop指令。这些指令用于保存EBX和ECX寄存器的内容；也就是说，在调用过程*Root*前，将寄存器的原有数据返回给调用程序。这样，过程与调用程序相互独立，在使用过程*Root*时不必担心出现意外的结果，这一点将在下一节进一步讨论。

练习6.2

1. 假设NEAR32过程Exercise1被指令call Exercise1调用，如果这个调用语句的地址是00402000，调用前ESP的内容是00406000。那么，Exercise1过程的第一条指令执行后，堆栈返回地址是多少？ESP中的值又是多少？
2. 为什么分别编译过程时，要使用PUBLIC指示性语句？为什么分别编译过程时，要使用EXTRN指示性语句？

编程练习6.2

1. 编写一个主程序，输入一个整数，调用过程*Root*（代码段6-3），求该整数的平方根，并显示平方根的值。要求主程序和过程*Root*在同一文件中，并且一起汇编。
2. 在不同的文件中汇编过程*Root*和主程序，重复练习1。
3. 编写一个过程*GetValue*，提示输入介于0到特定最大值*MaxValue*之间的整数。主程序将EAX寄存器中*MaxValue*传送给过程。过程*GetValue*返回这个整数，并把它放在EAX寄存器中，

过程*GetValue*重复提示输入，直至输入的值是在一个特定范围内。编写过程*GetValue*，在返回到调用程序时，EAX是惟一改变的寄存器，其他标志寄存器不能改变。

6.3 参数和局部变量

在高级语言中，过程定义常常包括参数、和变元有关的形参或调用过程的实参。当过程被调用时，过程的（值传递）参数以及实参值（可以是表达式）被复制到参数上。然后，这些值在过程中通过过程中的本地名来引用，这些本地名是定义参数的标识符。输入输出（通过地址或变量传递）参数将一个参数标识符与一个单变量的变元关联在一起，该参数标识符在调用者与过程之间相互传递值。本节讨论了参数传递的常用方法，该方法可为输入参数传递字或双字型的值，或者在为调用程序中为输入输出参数传递数据地址。

尽管简单的过程可以只用寄存器传递参数，但是大多数过程要用堆栈来传递参数。堆栈通常还用来存储局部变量。后面将看到，参数和局部变量使用堆栈的方法是紧密相关的。

下面用一个简单的例子来说明如何用堆栈传递参数。假定一个NEAR32过程*Add2*的工作是将两个双字整型数相加，并将相加的和返回到EAX寄存器。如果通过把参数压入堆栈来传递参数调用程序，那么代码如下:

```
push Value1        ; 第一个变量值
push ecx           ; 第二个变量值
call Add2          ; 调用子过程找到总和
add  esp, 8        ; 从堆栈移除参数
```

在考虑如何从堆栈访问参数值之前，要注意的问题是，执行调用指令后，这些参数是如何从堆栈中取出的。这里不需要将从堆栈中弹出参数放到某一目的地址，仅仅在堆栈指针上加8，ESP就可以跳过参数了。从堆栈中移出参数很重要，因为重复的过程调用可能会耗尽堆栈空间。更严重的是，如果过程调用是嵌套的，并且内部调用在堆栈中留下了参数，那么外部返回在堆栈中将找不到正确的返回地址。一个可选的方法就是在调用程序的堆栈指针上加n，在过程中使用ret n指令，在取出返回地址后，这种形式的返回指令将ESP内容加n。这两种形式本书都会举例说明。

代码段6-4说明了过程*Add2*是如何将两个参数值从堆栈中取出的，过程代码使用基地址模式。在这种模式下，存储器地址是基地址寄存器的内容加上指令中的偏移量的和。微软汇编器允许一个基地址用多种符号表示，本书使用[寄存器＋数字]的形式，例如，[ebp＋6]。任何通用的寄存器（如EAX、EBX、ECX、EDX、ESI、EDI、EBP或ESP）都可以作为基地址寄存器，EBP通常用于访问堆栈中的值。

代码段6-4 堆栈中使用参数值传递

```
Add2        PROC NEAR32                 ; 通过堆栈传递将两个数字相加
                                        ; 返回的和存入寄存器EAX中
            push   ebp                  ; 保存寄存器
            mov    ebp,esp              ; 建立堆栈
            mov    eax,[ebp+8]          ; 复制第二个参数值
                                        ;
            add    eax,[ebp+12]         ; 加上第一个参数值
            pop    ebp                  ; 恢复寄存器EBP
            ret                         ; 返回
Add2        ENDP
```

传递实参值的方法如下。在调用过程前，堆栈内容如图6-3中左图所示。

```
push ebp            ; 保存EBP
mov ebp, esp        ; 建立堆栈
```

这两条过程指令执行后，堆栈内容如图6-3中右图所示。

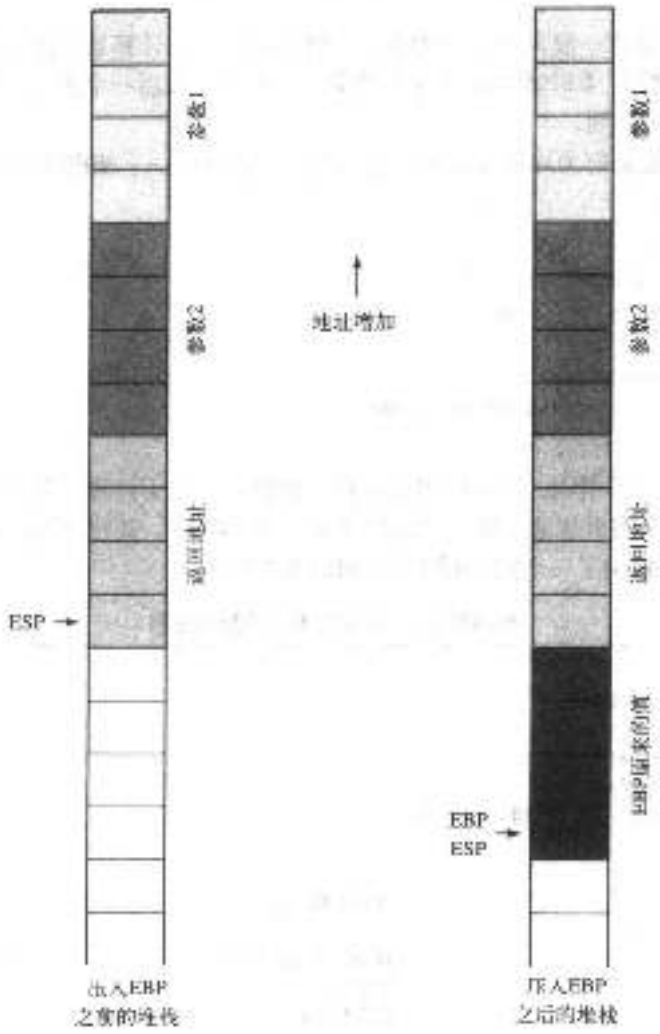


图6-3　堆栈内的局部参数

在EBP（和ESP）中的地址和第2个参数值之间存储了8个字节。因此，参数2的地址是[bp + 8]，第一个参数值在堆栈的高4字节，其地址为[bp + 12]，代码

```
mov eax, [bp + 8]        ; 复制第二个参数
add eax, [bp + 12]       ; 添加第一个参数
```

使用堆栈中存储器地址的值来计算要求的和。

这里为什么要使用EBP，为什么不只使用ESP作为基地址寄存器呢？其主要原因是ESP的内容很可能会改变，但指令mov ebp, esp将堆栈中固定的参考点放入EBP中。即使堆栈用于其

他目的，如压入另外的寄存器值或调用其他过程，这个固定参考点都不会改变。

有些过程需要为局部变量分配堆栈空间，大多数过程需要保存寄存器的内容，如代码段6-3所示。实现这些任务的指令和下面的两条指令一起组成一个过程的入口代码（entry code）。

```
push ebp          ; 保存EBP
mov ebp, esp      ; 建立堆栈
```

但是，这两条指令是第一段入口代码指令。通过它们，可以计算最后的参数是存储在EBP的参考指针之上的8字节。EBP寄存器本身总是第一个压入，最后一个弹出，这样返回到调用程序的值和调用前的值相同。

现在考察堆栈是如何为局部变量分配空间的。首先回顾一下编程练习5.3中计算两个整数的最大公约数的算法。

```
gcd : = number 1;
remainder : = number 2;
until(remainder = 0) loop
    dividend : = gcd;
    gcd : = remainder;
    remainder : = dividend mod gcd;
end until;
```

如代码段6-5所示，作为一个NEAR32过程，该设计实现了计算两个双字整型数的最大公约数的算法，这两个整数值通过堆栈传递给过程，返回GCD，并将它放到EAX中，除了这个过程外，代码段6-5提供了一个完整的文件，可以独立汇编。

代码段6-5 计算最大公约数的过程

```
PUBLIC GCD
; 计算两数最大公约数的过程
; 通过堆栈传送双字整型参数
; 返回最大公约数到EAX
; 其他寄存器及标志位不改变
; 作者: R. Detmer        日期: 1997年10月

GCD     PROC    NEAR32
        push    ebp                     ; 建立堆栈
        mov     ebp,esp
        sub     esp,4                   ; 存放局部变量的空间
        push    edx                     ; 保存EDX
        pushf                           ; 保存标志位

        mov     eax,[ebp+8]             ; 取出第一个数Number1
        mov     [ebp-4],eax             ; GCD := Number1
        mov     edx,[ebp+8]             ; 余数 := Number2
until0: mov     eax,[ebp-4]             ; 被除数 := GCD
        mov     [ebp-4],edx             ; GCD := 余数
        mov     edx,0                   ; 将被除数扩展为双字
        div     DWORD PTR [ebp-4]       ; 余数存入EDX
        cmp     edx, 0                  ; 余数 = 0?
        jnz     until0                  ; 如果不是，重复

        mov     eax,[ebp-4]             ; 将GCD存入EAX
        popf                            ; 恢复标志位
```

```
        pop     edx             ; 恢复EDX
        mov     esp, ebp        ; 恢复ESP
        pop     ebp             ; 恢复EBP
        ret     0               ; 返回，释放参数
GCD     ENDP
        END
```

在这个过程中，gcd存储在堆栈中，直到返回值放入EAX中，指令：

```
sub esp, 4          ; 给一局部双字释放空间
```

使堆栈指针向下移动4位，在存储EBP的位置下方，以及在存储其他寄存器的上方预留一个双字的空间，在EDX和标志寄存器压入堆栈后，堆栈的内容如图6-4所示，现在局部变量*gcd*可以在[ebp-4]地址处访问，这个地址是EBP中的固定参考下的4字节。

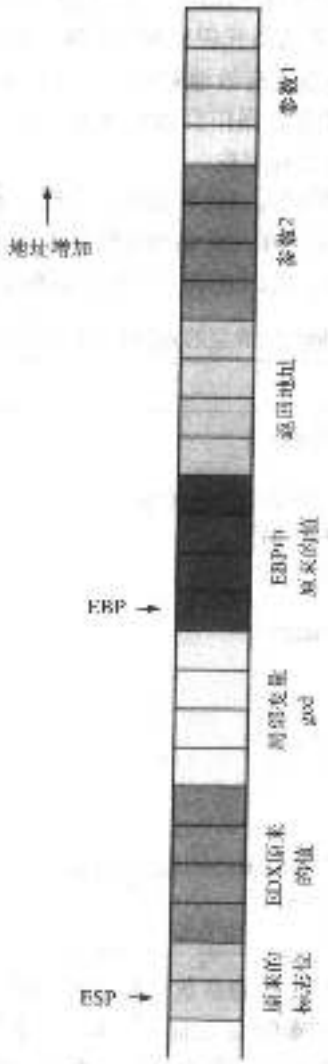


图6-4 使用局部变量的堆栈

该过程的其余部分的设计很容易实现。在这个例子中，寄存器可用于存储gcd，但很多过程都有太多的局部变量，不能将它们都存储在寄存器中。因此，可以在堆栈中存储一些局部变量，在[ebp－offset]地址处访问。注意，在预留局部变量的空间后，要保存寄存器的内容，这样被存储的寄存器个数不会影响变量的偏移量。还要注意的是，如果寄存器内容在返回调用程序后都没有改变，那么大部分过程要保存超过两个寄存器的内容。

最后考虑过程的出口代码（exit code）。

```
popf            ; 恢复标志位
pop edx         ; 恢复EDX
mov esp, ebp    ; 恢复ESP
pop ebp         ; 恢复EBP
ret 8           ; 返回，释放参数
```

前面的两条pop指令只是恢复标志寄存器和EDX，这些指令的出栈顺序与入栈的时候相反。下一条指令应该是add sp, 4，以消除在入口代码处相应的减法的影响。但是，无论为局部变量分配多少空间，指令mov esp, ebp都能更有效地做到这一点，而且像add指令一样，这条指令不会改变标志位。最后，使用了ret指令，操作数为8，这样，对于这个过程而言，调用程序不必从堆栈中移出参数，这个任务由该过程完成。

代码段6-6总结了一个过程常用的入口代码和出口代码，高级语言的编译器为子程序生成类似的代码。事实上，通常可用这种方式来编写高级语言调用的汇编语言过程。编写过程的方法有很多，因此，在写过程之前，可以查看一下编译器的参考资料。

代码段6-6 典型的入口与出口代码过程

入口代码：

```
push    ebp             ; 建立堆栈
mov     ebp, esp
sub     esp, n          ; n个字节的局部变量参数
push    ...             ; 保存寄存器
...
push    ...
pushf                   ; 保存标志位
```

出口代码：

```
popf                    ; 恢复标志位
pop     ...             ; 恢复寄存器
...
pop     ...
mov     esp, ebp        ; 如果局部变量已使用过，恢复ESP
pop     ebp             ; 恢复EBP
ret                     ; 返回
```

如何让一种高级语言实现多种多样的参数？如何将大的参数如数组、字符串或记录有效地传递给过程？这些可以通过将实参的地址而不是实参值本身传递给过程来实现，然后，过程可以使用在这个地址的数值，或是在这个地址存入新值。代码段6-7给出了一个过程，该过程实现了Pascal过程。

```
PROCEDURE Minimum (A : IntegerArray ;
          Count : INTEGER;
          VAR Min : INTEGER);
(*  Set Min to smallest value in A[1], A[2], ..., A[Count]  *)
```

上述代码中，对应于变量*A*和*Min*的地址传到了过程*Minimum*。这段过程使用了寄存器间接寻址的方式，首先检查每个数组元素，最后保存最小值。

代码段6-7　使用地址参数的过程

```
; 找寻数组A中最小值的过程
; 参数:   (1) 数组A的地址
;         (2) Count值（字）
;         (3) Min的地址（最小值的目标地址）
; 无寄存器可变。标志不可变
Minimum PROC   NEAR32
        push   ebp              ; 建立堆栈
        mov    ebp,esp
        pushad                  ; 保存所有寄存器
        pushf                   ; 保存标志

        mov    ebx,[ebp+14]     ; 获得数组A的地址
        mov    ecx, 0           ; 保证升序，0存入ECX
        mov    cx,[ebp+12]      ; 获得数值
        mov    eax,7fffffffh    ; 到目前为止的最小（MaxInt）
        jecxz  endForCount      ; 当无元素检查时退出
forCount:
        cmp    [ebx],eax        ; 元素 < 到目前为止最小的元素
        jnl    endIfLess        ; 如果不是则退出
        mov    eax,[ebx]        ; 新的最小值
endIfLess:
        add    ebx,4            ; 下一个数组元素的地址
        loop   forCount         ; 重复
endForCount:
        mov    ebx,[ebp+8]      ; 得到Min的地址
        mov    [ebx],eax        ; 移动最小到Min
        popf                    ; 恢复标志位
        popad                   ; 恢复寄存器
        pop    ebp              ; 恢复EBP
        ret                     ; 返回
Minimum ENDP
```

指令pushad和popad保存和恢复所有通用寄存器的内容。这些指令使用方便，但是如果过程生成的值要返回到一个寄存器中，那么它们就不能使用。注意，由于Count参数是以字为单位的，那么第一个参数的地址是在固定的基址上的14个字节，其中EBP占用4个字节，返回地址占用4字节，Min的地址占用4个字节，Count的值占用2个字节。（画一下堆栈图。）

调用过程Minimum的调用代码如下：

```
lea   eax, Array     ; 参数1: Array的地址
push  eax
push  Count          ; 参数2: Count的值
lea   eax, Min       ; 参数3: Min的地址
push  eax
call  Minimum        ; 调用过程
add   esp, 10        ; 释放参数
```

这段代码执行后，数组中的最小值保存在Min指向的地址中。

对于一个过程而言，将局部变量存在数据段中是十分合理的。.DATA指示性语句可放在一个由几个过程各自独立汇编的文件中。事实上，一个程序中可以有多条.DATA指示性语句，虽然，通常没有必要这样做。变量可放在堆栈或者是数据段部分，尽可能保持变量只在局部起作用，该数据段仅在包含定义的文件汇编时可见。即使一个过程和调用程序在同一文件中汇编，也应该避免直接引用过程中调用代码的变量。

通常，80x86指令是编译器的输出，因此80x86体系结构提供了一些便于过程实现的指令。enter（输入）指令语法如下:

```
enter localBytes, nestingLevel
```

当*nestingLevel*为0时，这条指令的功能和下面几条指令一样:

```
push  ebp
mov   ebp, esp
sub   esp, localBytes
```

也就是说，这条指令建立了一个堆栈区，并且预留了局部变量所需的存储空间。如果*nestingLevel*>0，这条enter指令也将堆栈指针从*nestingLevel*-1层压回到新指针的位置之上，这使得过程容易访问嵌套过程的变量。如果使用enter指令，那么这条enter指令通常会是过程中的第一条指令。

leave指令和enter指令作用相反。具体而言，它的功能和如下两条指令一样:

```
mov esp, ebp     ; 恢复ESP
pop ebp          ; 恢复EBP
```

通常这条指令之后会紧跟一条返回指令。本书中的过程没有用到enter或leave指令。

本书中的每个程序都有退出语句:

```
INVOKE ExitProcess, 0     ; 返回代码0的exit
```

INVOKE不是一条指令——MASM称它为指示性语句。然而，它的作用更像是一个宏。事实上，如果指示性语句.LISTALL在上面的代码行之前，那么它将扩展为:

```
push + 00000000h
call ExitProcess
```

显然，这是用一个双字参数0在调用过程*ExitProcess*。

练习6.3

1. 假设过程NEAR32开始于:

```
push ebp            ; 保存EBP
mov  ebp, esp       ; 新基址指针
push ecx            ; 保存寄存器
push esi
...
```

假设这个过程有3个参数传递到堆栈：（1）一个双字，（2）一个字和（3）第二个字。画出上面的代码执行后的堆栈图，图中包括参数、返回地址以及EBP和ESP指向的字节。并说明每个参数是如何引用的。

2. 给出过程NEAR32的入口代码（代码段6-6），该过程入口代码为局部变量预留了8个字节的堆栈存储空间，假设这个空间被2个双字占用，那么每个局部变量是如何引用的？
3. 解释为什么在将返回值放入寄存器EAX的过程中不能使用pushad和popad指令。

编程练习6.3

写一个NEAR32过程，该过程实现下面的具体任务。对于每一个过程，使用堆栈传递参数给过程。除非具体说明要返回一个值放入寄存器中，否则寄存器中的值在过程中应该始终不变。也就是说，在过程中使用的寄存器（包括标志寄存器）应该在过程的开始就被保存起来，在返回过程前恢复。根据局部变量分配堆栈空间，使用不带操作数的ret指令。对于下面的问题，分别编写一个汇编测试驱动程序，一个输入合适的值的简单的主程序，该主程序调用过程并输出结果。主程序必须从堆栈中移出变量。链接并运行每个完整的程序。

1. 写一个过程*Min*2，该过程可找出两个一个字长的整型参数中的最小值，并将这个最小值放在AX寄存器中。
2. 写一个过程*Max*3，该过程可找出3个双字长的整型参数中的最大值，并将这个最大值放在EAX寄存器中。
3. 写一个过程*Avg*，该过程可找出一个数组中双字长的整数的平均值。过程*Avg*有3个参数：
   （1）数组的地址
   （2）数组中整数的个数（以双字传递）
   （3）一个用于保存结果的双字的地址
4. 写一个过程*Search*，该过程可从双字长的数组中找到某个特定的值。如果在数组中找到这个值，返回这个值在数组中的位置，并将返回值放入EAX寄存器中；如果没有找到，则返回0。过程Search有3个参数：
   （1）要搜索的值（一个双字）
   （2）数组的地址
   （3）数组中双字长的数的个数N（以双字形式传递）

6.4 递归

递归（recursive）过程或函数是指直接或间接调用它自己的过程或函数。处理数据结构的最好算法是递归。如果一种编程语言不支持递归的话，要用这样的语言实现某些算法通常是十分困难的。

用80x86汇编语言编写一个递归程序几乎和编写普通程序一样简单。如果参数传到堆栈中，

并且局部变量也存在堆栈中，那么过程的每一个调用为其参数和局部变量分配新的存储空间。由于每个调用都有自己的堆栈，因此，传递给一个过程调用的参数就不会和其他调用混淆。如果寄存器的内容被正确地保存和恢复，那么过程的每次调用都可使用同样的寄存器。

本小节给出了一个用80x86汇编语言编写的递归过程的例子，它解决了汉诺塔问题，图6-5中有4个盘子。这个问题的目标是把所有的盘子从原柱A移到目标柱B，每次一个，不能将大的盘子放在小盘子上。盘子可以放到备用柱C上。举例说，如果只有两个盘，小盘可以先从原柱A移到C，大的盘子可以从A移到B，最后将小盘子的再从C移到B。

图6-5 汉诺塔问题

通常，汉诺塔问题可分两种情况来解决。如果仅有一个盘子，那么，单盘可以简单地从原柱移到目标柱。如果盘数NbrDisks大于1，则顶上的（NbrDisks－1）个稍小的盘子从目标柱移到备用柱，最大的盘子移到目标柱，最后，（NbrDisks－1）个稍小的盘子从备用柱移到目标柱。每次（NbrDisks－1）个盘子被移之后，除了原柱、目标柱和备用柱的角色发生了变化之外，其他步骤将被重复执行。图6-6给出了该算法的伪代码。

```
procedure Move(NbrDisks, Source, Destination, Spare);
begin
        if NbrDisks = 1
        then
                display "Move disk from ", Source, " to ", Destination
        else
                Move(NbrDisks – 1, Source, Spare, Destination);
                Move(1, Source, Destination, Spare);
                Move(NbrDisks – 1, Spare, Destination, Source);
        end if;
end procedure Move;
begin {main program}
        prompt for and input Number;
        Move(Number, 'A', 'B', 'C');
end;
```

图6-6 汉诺塔问题过程伪代码

代码段6-8给出了用80x86汇编语言实现的汉诺塔问题的代码。堆栈用来传递参数给过程*Move*，该过程是一个只为输出引用数据段的NEAR32过程。建立一个标准的堆栈区，该过程所使用的寄存器需要保存和恢复，这段代码是直接根据伪代码编写的。下面的语句需要操作符DWORD PTR:

```
cmp  DWORD PTR  [bp + 14], 1
```

因此，汇编器能够比较字或字节长的操作数。同样，有几个地方使用了助记符pushw，汇编器就可以压入字长的参数。注意，这个递归调用的实现和主程序调用是同一种方式，通过将4个参数压入堆栈，调用过程Move，然后，从堆栈中移出参数。但是，在该主程序中备用柱的参数是不变的，保存在一个字的低字节部分，因为单个的字节不能压入到80x86堆栈。

代码段6-8　汉诺塔问题的解决方案

```
; 打印解决"汉诺塔"问题指令的程序
; 作者: R. Detmer        日期: 1997年10月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

include io.h                ; 输入/输出的头文件

cr      equ     0dh         ; 回车符
Lf      equ     0ah         ; 换行

.STACK  4096                ; 保留4096字节堆栈

.DATA                       ; 数据保留区
prompt      BYTE    cr,Lf,'How many disks?  ',0
number      BYTE    16 DUP (?)
message     BYTE    cr,Lf,'Move disk from spindle '
source      BYTE    ?
            BYTE    ' to spindle '
dest        BYTE    ?
            BYTE    '.',0

.CODE
Move        PROC NEAR32
; Procedure  Move (NbrDisks: integer; {需要移动的盘子个数})
;       Source, Dest, Spare: Character{使用的柱}
; 参数以字为单位传到栈中

            push    ebp                     ; 保存基址指针
            mov     ebp,esp                 ; 复制堆栈指针
            push    eax                     ; 保存寄存器
            push    ebx

            cmp     WORD PTR [ebp+14],1     ; 判断盘的个数是否为1
            jne     elseMore                ; 大于1时跳转
            mov     bx,[ebp+12]             ; 源柱
            mov     source,bl               ; 将记录复制到输出函数
            mov     bx,[ebp+10]             ; 目的柱
            mov     dest,bl                 ; 将记录复制到输出函数
            output  message                 ; 输出
            jmp     endIfOne                ; 返回
elseMore:   mov     ax,[ebp+14]             ; 得到盘的个数
            dec     ax                      ; 盘的个数-1
```

```
            push    ax              ; 参数1: 盘的个数-1
            pushw   [ebp+12]        ; 参数2: 源柱不改变
            pushw   [ebp+8]         ; 参数3: 备用柱为新的目的柱
            pushw   [ebp+10]        ; 参数4: 原目的柱为新的备用柱
            call    Move            ; 调用Move函数
            add     esp,8           ; 从栈中将参数移回

            pushw   1               ; 参数1: 1
            pushw   [ebp+12]        ; 参数2: 源柱不改变
            pushw   [ebp+10]        ; 参数3: 目的柱不改变
            pushw   [ebp+8]         ; 参数4: 备用柱不改变
            call    Move            ; 调用Move函数
            add     esp,8           ; 从栈中移出参数

            push    ax              ; 参数1: 盘的个数-1
            pushw   [ebp+8]         ; 参数2: 源柱为初始状态下的备用柱
            pushw   [ebp+10]        ; 参数3: 初始目的柱
            pushw   [ebp+12]        ; 参数4: 备用柱为初始状态下的源柱
            call    Move            ; 调用Move函数
            add     esp,8           ; 从栈中取出参数
endIfOne:
            pop     ebx             ; 恢复寄存器
            pop     eax
            pop     ebp             ; 恢复基址指针
            ret                     ; 返回
Move        ENDP

_start:     output  prompt          ; 询问盘的个数
            input   number,16       ; 读取ASCII字符
            atoi    number          ; 转换为整型
            push    ax              ; 参数1: Number
            mov     al,'A'          ; 参数2: 'A'
            push    ax
            mov     al,'B'          ; 参数3: 'B'
            push    ax
            mov     al,'C'          ; 参数4: 'C'
            push    ax
            call    Move            ; 调用Move函数
            add     esp,8           ; 从栈中取出参数

            INVOKE  ExitProcess, 0  ; 退出，并返回代码0

PUBLIC _start                       ; 公开程序入口点
END                                 ; 源代码结束
```

练习6.4

1. 假如在过程*Move*开始时，EAX没有被保存；并且在该过程结束时，EAX没有被恢复，那么汉诺塔的程序会出现什么错误？
2. 假设执行汉诺塔程序时盘的数目是2，通过主程序中的指令add esp, 8，从主程序的第一次入栈开始，画出堆栈变化图。

编程练习6.4

1. 阶乘函数定义如下，其中n是一个非负整数

$$\text{factorial}(n)=\begin{cases}1 \text{ if } n=0\\ n\times\text{factorial}(n-1) \text{ if } n>0\end{cases}$$

用汇编语言编写一个名为*Factorial*的过程，实现阶乘函数的递归定义。使用堆栈传递单个双字整型参数；函数返回的值放在EAX寄存器中；调用程序从堆栈移出参数。通过主程序输入一个整数调用*Factorial*，测试阶乘函数，显示函数所返回的值。解释为什么在这个函数中用双字长比字长整数更好？

2. 两个正的实整数*m*和*n*的最大公约数（GCD）可通过下面的伪代码递归计算得出：

```
function GCD(m, n :integer) :integer:
if n = 0
then
    return m;
else
    Remainder : = m mod n;
    return GCD(n, Remainder);
end if;
```

用汇编语言实现这个递归定义。使用堆栈传递两个双字整型参数，函数返回的值放在EAX寄存器中。过程应当从堆栈中移出参数。通过主程序输入两个整数，调用最大公约数GCD函数，测试该函数，显示返回的值。

6.5 其他体系结构：没有堆栈的过程

并不是所有的计算机体系结构都提供硬件堆栈，有些计算机体系结构可通过留出一块存储单元实现软件堆栈；软件堆栈也可作为堆栈，保持栈顶位置为一个变量，通过将数据复制到堆栈或从堆栈移出数据来实现数据的入栈和出栈。但是，它没有80x86体系结构方便，因为后者可根据入栈或出栈、调用过程以及过程返回自动调整栈顶。

显然，堆栈在80x86过程的实现中扮演了一个重要角色。如何在一个没有堆栈的体系结构上合理地实现过程呢？本节将简要讨论该问题。以常用的IBM主机为例，这种机器的体系结构起源于20世纪60年代第一次引入的System/360（S/360）系统。

S/360体系结构包括16个32位的通用寄存器（GPR），编号为0～15。地址为24位长，一个地址可存储在任一个寄存器中，这种体系结构包括直接寻址、寄存器间接寻址和变址寻址。

通常，将过程地址放入通用寄存器15（GPR 15）来调用该过程，然后，将下一条指令的地址复制到通用寄存器14（GPR 14）后，执行一条分支和链接指令，将程序转移到该过程处。这样，调用返回时只需简单地跳转到GPR 14中的地址处。

参数传递有些难。通常GPR 1用来传递参数地址列表的地址。这是一个32位存储单元的列表（在S/360体系结构中，32位被称为一个字），第一个字存储第一个参数的地址，第二个字存储第二个参数的地址，依此类推。为取出一个字长的参数，必须通过参数地址列表来得到它的地址，然后在这个地址处复制该字。

每次调用一个过程时，同样的通用寄存器一般用于同样的任务。但当一个过程调用另一个过程时，可能会出现一些问题。例如，第二个过程调用将移出第一个返回地址，把第二个返回地址存入GPR 14。为了避免这个问题，主程序和每个过程为寄存器存储域分配了一块存储单元，并把它的地址在过程调用前放入GPR 13。然后，该过程将通用寄存器0～12、14和15保存在调用程序的寄存器存储域，而GPR 13保存在它自己的寄存器存储域。这种系统比使用堆栈要相对复杂一些，除了递归过程调用外，它和使用堆栈的过程一样。由于每个过程只有一个寄存器存储域，因此在不修改系统的情况下，要实现递归过程调用是不可能的。

练习6.5

1. 假设将IBM S/360参数传递方案转换为80x86汇编语言，调用程序的代码如下：

```
Double1     DWORD  ?
...
Value1      DWORD  ?
Value2      DWORD  ?
...
AddrList    DWORD   OFFSET  Value1     ; 参数1地址
            DWORD   OFFSET  Value2     ; 参数2地址
            DWORD   OFFSET  Double1    ; 参数3地址
...
            lea     ebx, AddrList      ; 获得AddrList的地址
            call    Proc1
```

注意，参数不需要连续的存储空间，但它们的地址在AddrList中是连续的字。写出代码，显示过程*Proc*1如何访问3个参数。

2. 如果使用本节的系统调用递归过程，会出现什么情况？

本章小结

本章讨论了在80x86体系下实现过程的技术。在过程实现时，堆栈有几个重要的作用。当调用一个过程时，在控制转移到过程的第一条指令前，下一条指令的地址被存储在堆栈中。为了返回到调用程序中的正确地址，返回指令从堆栈中取出地址。参数值或它们的地址能被压入到堆栈传递给一个过程；这时，基指针EBP和基地址为访问过程中的参数值提供了一种便捷的机制。堆栈可用来为过程的局部变量提供存储空间。堆栈总是用来“保护环境”，例如，当一个过程开始和在返回调用程序时，寄存器内容可被压入堆栈。这样，调用程序不必担心寄存器的内容会被过程改变。

递归算法在许多计算应用中很常见。在80x86体系中，递归过程并不比非递归过程更困难。

有些计算机体系结构没有硬件堆栈。使用寄存器存储地址，当一个过程调用另一个过程时，用存储单元保存寄存器，可实现非递归过程。

第7章 串 操 作

计算机常用于处理数值型数据和字符串数据。在数据处理时，应用程序名、地址等都必须要存储起来，有时还需重新组织。文本编辑器和字处理程序都需具有查找和移动字符串的功能。汇编语言程序必须能解释汇编语言中的语句元素，鉴别出要保留的助记符。即使当处理基本数值型计算时，比如从键盘输入一个数字，也必须将一个字符串转换为（计算机）内部的数值型格式，或者将某种内部格式转换为字符串来显示输出。

80x86微处理器处理字符串的指令。同一种指令能处理双字长的串或字长的串。本章涵盖了80x86中用于串操作的指令，重点是字符串的操作指令。本章将给出多种串的应用，包括与一些高级语言中类似的像宏dtoa的子程序。

7.1 串指令

80x86有5个用于串操作的指令：movs（串传送）、cmps（串比较）、scas（串扫描）、stos（存入串）和lods（从串取）。movs指令把一个串从存储器的一个位置复制到另外一个位置。cmps指令用于比较两个串的内容。scas指令用来查找串中某个特定的值。stos指令能存储一个新的值到串的某个位置。最后一条指令lods能复制出串的某个位置的值。

80x86中的串（string）是指存储器中连续的字节、字或双字的一个集合。串通常使用如下的指令在程序的数据段中定义：

```
response     BYTE    20   DUP  (?)
label1       BYTE    'The results are ', 0
wordString   WORD    50 DUP (?)
arrayD       DWORD   60 DUP (0)
```

注意：串和数组表面上不同，实际上它们是相同的。

每一条串指令需要一个源串、一个目的串，或者两者都需要。串指令处理串时，一次处理串中的一条字节、字或者双字。寄存器间接寻址用来定位单独的字节、字或双字元素。80x86指令使用源索引寄存器ESI的地址来访问源串中的元素，使用目的索引寄存器EDI的地址来访问目的串中的元素。如果程序使用的是分段存储模式，则应该知道源串的元素在数据段中（地址为DS：ESI），同时，目的串的元素在附加段（地址为ES：EDI）中。在平面存储模式的程序中，段寄存器有相同的段号，各个段之间无差别。

由于源串和目的串元素的地址通常分别由ESI和EDI给出，因此，不需要操作数来确定它们的位置。然而，如果没有任何操作数，汇编器不能判断出串元素的长度。例如，串传送时，可以是移动一个字节、字或者双字。微软的宏汇编器对此提供了两种方法，第一种方法是使用目的操作数和源操作数；除非MASM知道它们的类型（两个操作数必须是同一类型），并使用该元素长度进行操作，否则操作数将被忽略。第二种方法是使用特殊的助记符号来定义元素的长度，字节操作串指令使用b后缀，字长操作的串指令使用w后缀，双字的串指令使用d后

缀。例如，movsb用来字节串传送，movsw是字串传送，movsd是双字串传送。这些指令中的任意一个都像movs指令一样汇编，并且都不需要操作数，因为汇编程序从助记符中知道每个元素的长度。本书对于串指令使用后缀b、w和d的助记符，而不采用操作数表示。

串指令虽然一次只能操作一个串元素，但已对下一个元素的操作做好了准备。因此，串指令改变源索引寄存器ESI和（或者）目的索引寄存器EDI，使其指向串中下一个元素的地址。当处理元素是字节长时，寄存器加（减）1，当处理元素是字长时，ESI和EDI寄存器加（减）2；处理元素是双字长时，ESI和EDI寄存器加（减）4。80x86能在一个串中向前移动，即从低地址向高地址移动，也可以向后移动，即从高地址向低地址移动。移动的方向由方向标志DF的值决定，即由EFLAGS寄存器位10决定。如果DF置为1，则串指令使EDI和ESI中的地址值递减，从右向左对串操作。如果DF清0，则串指令使ESI和EDI的值递增，即从左向右处理串。

80x86有两条指令，它们的惟一作用就是对方向标志DF置0或置1。cld指令将DF清0，这样串指令使ESI和EDI的值递增，从左向右处理串。std指令将DF置1，这样可以从右向左处理串。这两条指令都只影响DF标志位。这些指令的技术参数如表7-1所示。

表7-1 cld和std指令

指令	时钟周期			字节数	操作码
	386	486	Pentium		
cld	2	2	2	1	FC
std	2	2	2	1	FD

下面详细讨论串指令。串传送指令movs把一个串的元素（字节、字或双字）从源串传送到目的串，即地址DS: ESI指向的源元素被复制到地址ES: EDI。串元素复制完后，两个索引寄存器的值根据每个元素的长度（1、2或者4）而改变，如果方向标志DF是0就增加一个元素的长度，如果DF是1就减少一个元素的长度。移动指令movs不影响任何标志位，具体有movsb、movsw和movsd三种形式。表7-2给出了每种形式的相关信息。

表7-2 movs指令（使用EDI和ESI）

助记符	元素长度	时钟周期			字节数	操作码
		386	486	Pentium		
movsb	字节	7	7	4	1	A4
movsw	字	7	7	4	1	A5
movsd	双字	7	7	4	1	A5

代码段7-1给出了使用MOV指令的一个例程。该例中最重要的部分是过程*strcopy*，该过程有两个通过栈传递的参数，这两个参数给出了字节串或者字符串的源地址和目的地址。假设源串是以空字节作结束符，过程*strcopy*在目的地址复制了源串，以空字节作为结束符。

该过程仅使用了EDI和ESI寄存器，在堆栈中保存了它们的值和标志寄存器，因而，过程返回时它们的值不会发生改变。索引寄存器EDI和ESI必须被初始化为待处理串的第一个字节的地址。ESI和EDI作为参数入栈，方向标志清0，从左至右复制串。

代码段7-1 串复制程序

```
; 测试过程strcopy
; 作者: R. Detmer 日期: 1997年10月

.386
.MODEL FLAT
ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
INCLUDE io.h                ; input/output的头文件
cr      equ     0dh         ; 回车符
Lf      equ     0ah         ; 换行符

.STACK  4096                ; 保留4096字节的栈

.DATA                       ; 数据保留区
prompt      BYTE    cr, Lf, "Original string?  ",0
stringIn    BYTE    80 DUP (?)
display     BYTE    cr, Lf, "Your string was...", cr, Lf
stringOut   BYTE    80 DUP (?)

.CODE
_start:     output prompt               ; 请求输入
            input  stringIn, 80         ; 输入源串
            lea    eax, stringOut       ; 目的地址
            push   eax                  ; 第一个参数
            lea    eax, stringIn        ; 源地址
            push   eax                  ; 第二个参数
            call   strcopy              ; 调用字符串复制函数
            output display              ; 打印结果
            INVOKE ExitProcess, 0       ; 退出返回代码0
PUBLIC _start                           ; 公开程序入口点

strcopy     PROC NEAR32

; 复制串的程序（从起始串的首字节到最后的空字节结束）
; 说明: 目的位置有足够的空间来复制

; 通过堆栈传递参数:
;     (1)  目的地址
;     (2)  源地址
            push   ebp                  ; 保存基指针
            mov    ebp,esp              ; 复制栈指针

            push   edi                  ; 保存寄存器和标志位
            push   esi
            pushf

            mov    esi,[ebp+8]          ; 初始化源地址
            mov    edi,[ebp+12]         ; 初始化目的地址
            cld                         ; 清除方向标志位
whileNoNull:
            cmp    BYTE PTR [esi],0     ; 是否是源串最后的空字节
            je     endWhileNoNull       ; 如果是空停止复制
            movsb                       ; 复制一个字节
            jmp    whileNoNull          ; 检查下一个字节
```

```
endWhileNoNull:
            mov     BYTE PTR [edi],0      ; 结束目的串

            popf                          ; 恢复标志位和寄存器
            pop     esi
            pop     edi
            pop     ebp
            ret     8                     ; 退出程序，参数出栈
strcopy     ENDP

            END
```

初始化以后，该过程实现了如下的伪代码:

```
while下一个源字节非空时;
    从源字节复制到目的串;
    源地址加1;
    目的地址加1;
end while;
在目的字节的最后输入空字节;
```

要检查下一个源串字节是否为空，使用的语句为:

```
cmp  BYTE  PTR  [esi], 0  ; 源字节是否是空字节?
```

回顾一下，符号[esi]表明是寄存器间接寻址，这样就可以访问ESI指向地址的值，也就是源串的当前字节。MASM需要操作符BYTE PTR来判断操作数[ESI]和0是需要字节、字还是双字参加比较。指令movsb可实现复制源串的字节，并且两个索引寄存器的值递增。最后，

```
mov  BYTE  PTR  [edi], 0  ; 结束目的串
```

该语句用来在目的串的末尾加上一个空字节，因为在源串的最后一个字节复制到目的串后，EDI的值已经递增了。另外，操作符BYTE PTR告诉MASM目的地址是字节而不是字或双字。

测试过程*strcopy*的程序只需从键盘输入一个串，调用*strcopy*，将输入的串复制到某一位置，最后显示复制的串。代码中最值得注意的是调用过程所需的指令集，因为该过程完成了串复制，而不需从堆栈中取出参数。

通常，movs指令不会使源串覆盖目的串。但是，有时候这一点很有用。假设将“*/”重复40次来初始化一个80个字符长的字符串starSlash，可用以下的代码实现。

```
starSlash   BYTE  80  DUP (?)
            ...
            mov   starSlash, '*'        ; 第一个*号
            mov   starSlash + 1, '/'    ; 第一个/号
            lea   esi, starSlash        ; 源地址
            lea   edi, starSlash + 2    ; 目的地址
            cld                         ; 从左向右处理
            mov   ecx, 78               ; 需复制的字符数
forCount:   movsb                       ; 复制下一个字符
            loop  forCount              ; 循环复制
```

在这个例子中，第一次执行movsb，从串的第一个位置得到一个“*”，将其复制到第三个

位置。下一步重复将从第二个位置得到的“/”复制到第四个位置。第三次执行，将从第三个位置得到的“*”复制到第五个位置，依次类推。下一节将介绍一种更简单的方法实现循环执行串传送指令。

练习7.1

1. 请给出下面程序的输出结果。

```
.386
.MODEL FLAT
ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
INCLUDE io.h                        ; 输入或输出头文件
cr      equ     0dh                 ; 回车符
Lf      equ     0ah                 ; 换行符

.STACK  4096                        ; 保留4096字节的堆栈
.DATA                               ; 全局数据
string  BYTE    'ABCDEFGHIJ'
        BYTE    cr, Lf, 0
.CODE
setup1  PROC    NEAR32
        lea     esi, string         ; 串的开始
        lea     edi, string + 5     ; 'F'的地址
        cld                         ; 地址增量
        ret
setup1  ENDP

_start: call    setup1              ; 置源、目的和方向
        movsb                       ; 移动4字节
        movsb
        movsb
        movsb
        output string               ; 显示处理过的串
        INVOKE  ExitProcess, 0      ; 退出和返回代码0
PUBLIC _start                       ; 公开程序入口点
        END
```

2. 用以下的程序代替问题1中的过程*setup*1，请给出程序的输出结果。

```
setup2  PROC NEAR32
        lea     esi, string         ; 栈开始
        lea     edi, string + 2     ; 'C'的地址
        cld                         ; 地址增量
        ret
setup2  ENDP
```

3. 用以下的程序代替问题 1 中的过程*setup*1，请给出程序的输出结果。

```
setup3  PROC NEAR32
        lea     esi, string + 9     ; 栈结束
        lea     edi, string + 4     ; 'E'的地址
```

```
        std                         ; 地址减量
        ret
setup3  ENDP
```

4. 用以下的程序代替问题 1 中的过程*setup*1，请给出程序的输出结果。

```
setup4  PROC NEAR32
        lea   esi, string + 9       ; 栈结束
        lea   edi, string + 7       ; 'H'的地址
        std                         ; 地址减量
        ret
setup4  ENDP
```

编程练习7.1

写一个程序，该程序每次从键盘读入一个串，然后将该串复制到一个大的存储区中以备处理。特别要注意的是，使用宏input来输入串，然后把串复制到数据段中预留的1024个字节的存储块中（回顾一下：输入宏input生成一个以空字符结束的串）。在存储区中，串后面紧跟回车和换行符。重复处理其他的串，复制其后的每个串到存储区，因此，最后一个串后紧跟的是换行符。当源串的第一个字符是“$”时，则退出循环——该串不复制到存储区。在最后一个串的换行符后面置一个空字节，退出循环。最后，使用宏output来显示数据区中的所有字符，显示输入的串，并且每行显示一个串。

7.2 重复前缀和其他串指令

每条80x86串指令每次操作一个串元素，但是80x86体系结构还有三种重复前缀，这样，串指令可以按照给定的次数自动重复执行，或者自动重复执行直到满足某种条件为止。这三个重复前缀实际上等同于两种不同的单字节编码，它们本身不是指令，而是扩展原有字符串指令的机器代码，生成新的指令。

代码段7-2给出了两个程序段，每个程序段从sourceStr复制一个定长的字符串到destStr中。字符数通过count存入到ECX寄存器中。程序代码a部分使用了一个循环，由于字符的个数count可能为0，因此，用jecxz指令控制循环。循环体用movsb一次复制一个字符，loop循环指令需要注意循环重复的次数。程序代码b部分在功能上和程序a部分相同，在count值存入ECX后，重复前缀rep与movsb指令一起使用；rep movsb的作用等同于程序a中的最后4行。

代码段7-2 复制一个串中指定长度的字符

```
        lea   esi, sourceStr   ; 源串
        lea   edi, destStr     ; 目的串
        cld                    ; 向前复制
        mov   ecx, count       ; 计数要复制的字符数
        jecxz endCopy          ; 如果count值为0，跳出循环
copy:   movsb                  ; 复制一个字符
        loop  copy             ; count减1和继续循环
endCopy:
```

(a) 一循环中重复使用movsb指令

```
lea   esi, sourceStr   ; 源串
lea   edi, destStr     ; 目的串
cld                    ; 向前复制
mov   ecx, count       ; 统计要复制的字符数
rep movsb              ; 复制字符
```

(b) 使用重复前缀的movsb指令

Rep前缀通常与movs指令及stos指令（下面将讨论）一起使用。执行如下的程序代码:

```
while countin ECX>0 loop
    执行循环体主指令集;
    ECX减1;
end while;
```

注意，这是一个while循环。如果ECX是0，循环体主指令集（primitive instruction）根本就不会执行。这里不需要重复串指令，因为常常使用loop指令实现for循环。

其他的两个重复前缀是repe（等同于助记符repz）和 repne（等同与助记符repnz）。助记符repe表示“相等的时候重复”，repz表示“为0的时候重复”。类似地，repne和repnz分别表示“不相等的时候重复”和“不为0的时候重复”。这些重复前缀都适合和串指令cmp和scas一起使用，这两条指令将影响零标志ZF。

这些助记符的名字部分地说明了它们的行为，每一条指令的工作原理与rep一样，当ECX不为0时，重复执行指令。但是，在串指令执行后，每次都要检查ZF标志位。当ZF为1时，repe和repz继续重复执行，如同跟在比较两个操作数是否相等的指令后。当ZF为0时，repne和repnz继续重复执行，如同跟在比较两个操作数是否不相等的指令后。重复前缀不会影响任何标志位。表7-3总结了三种重复前缀。注意: rep和repz（repe）会产生相同的操作码。

表7-3 重复前缀

助记符	循环条件	字节数	操作码
rep	ECX>0	1	F3
repz/repe	ECX>0 and ZF = 1	1	F3
repnz/repne	ECX>0 and ZF = 0	1	F2

如表7-3所示，当while循环条件为真时，前缀repz和repnz不会退出过程。在循环体主指令集第一次重复之前，首先检查ECX的值，如同使用while循环。但是，在重复循环体主指令集完成后才会检查ZF。如果循环次数为0，则不会执行循环体，程序员不必专门在重复指令前初始化ZF。

表7-4显示了重复前缀rep是如何结合movs指令一起使用的。在时钟周期列中，每一次重复都会有一个“启动”时间加上每次的循环时间。表中的n代表重复的次数。例如，一条rep movs指令传送5个字节，需占用33（13 + 4*5）个Pentium处理器时钟周期。（当n = 1或者n = 0时，486和Pentium有专门的计时方式，所以表中的列并不是严格准确的。）

表7-5总结了cmp指令，比较了源串和目的串的元素。第5章讨论过cmp指令，该指令通过两个操作数相减的差值设置标志位，但两个操作数都没有改变。同样，cmps使两个串中的元

素相减，并根据其差值设置标志位，但两个操作数都没有改变。如果在一个循环中使用一条cmps指令，根据要完成的设计，可以在该指令后使用任何条件跳转指令。

表7-4 rep movs指令

助记符	元素大小	时钟周期			字节数	操作码
		386	486	Pentium		
rep movsb	字节	7 + 4*n*	12 + 3*n*	13 + 4*n*	2	F3 A4
rep movsw	字					F3 A5
rep movsd	双字					F3 A5

表7-5 cmp比较指令

助记符	元素大小	时钟周期			字节数	操作码
		386	486	Pentium		
cmpsb	字节	10	8	5	1	A6
cmpsw	字					A7
cmpsd	双字					A7
repe cmpsb	字节	5 + 9*n*	7 + 7*n*	9 + 4*n*	2	F3 A6
repe cmpsw	字					F3 A7
repe cmpsd	双字					F3 A7
repne cmpsb	字节	5 + 9*n*	7 + 7*n*	9 + 4*n*	2	F2 A6
repne cmpsw	字					F2 A7
repne cmpsd	双字					F2 A7

重复前缀通常与cmps指令一起使用。事实上，为了判断两个串是否相同，那么repe前缀和cmps最好一起使用。表7-5总结了所用的cmps指令，包括带有重复前缀的cmps指令。另外，对于486和Pentium CPU，这里给定的时间并不是严格精确的；对rep cmps来说，当n＝0时，会有特定的时钟周期。

有时需要判断一个串是否是另外一个串的子串。假如该子串存在的话，要找出该子串在另外一个串中的位置。如何确定该子串在另外一个串中的位置，可用下列简单的算法实现：

```
position:=1;
while(position<=(targetLength - keyLength + 1))  loop
    If在position处找到匹配的子串Key then
        报告查找成功;
        退出查找处理;
    end if;
    position加1;
end While;
报告查找失败;
```

该算法在目的串的每个可能的位置检查子串是否与目的串的某部分匹配。使用80x86寄存器，按照如下代码检查是否匹配：

```
ESI:=key串地址;
EDI:= address of target + position - 1;
ECX:=子串key长度;
```

```
forever loop
    if ECX=0 then
        退出循环体;
        end if;
        根据比较[ESI]和[EDI]结果置ZF;
        ESI加1;
        EDI加1;
        ECX减1;
        If ZF =0 then
            退出循环体;
        end if;
    end loop;

if ZF=1
then
    匹配查找成功;
end if;
```

该循环其实就是完成重复串指令repe cmpsb所做的事情。当ECX = 0或者ZF = 0，循环终止，必须确定最后比较的一对字符是否相等，因此，在代码的最后使用了if结构。代码段7-3给出了实现这个设计的完整程序。

串扫描指令scas用来在串中扫描某个特定的串元素是否在串中存在。被扫描的串是目的串，也就是说，被扫描的串元素地址存放在目的索引寄存器EDI中。用scasb指令，则需要查找的串元素以字节存放在寄存器AL中；用scasw指令，则需要查找的串元素以字存放在寄存器AX中；用scasd指令，则需要查找的串元素以双字存放在EAX中。通过助记符可以判断串元素的长度，所以，以上三条指令没有使用操作数。表7-6总结了scas指令，与前面的重复指令一样，486和Pentium机在n = 0时有特定的时钟周期。

表7-6 scac指令（使用EDI）

助记符	元素大小	时钟周期			字节数	操作码
		386	486	Pentium		
scasb	字节	7	6	4	1	AE
scasw	字					AF
scasd	双字					AF
repe scasb	字节	5 + 8*n*	7 + 5*n*	9 + 4*n*	2	F3AE
repe scasw	字					F3AF
repe scasd	双字					F3AF
repne scasb	字节	5 + 8*n*	7 + 5*n*	9 + 4*n*	2	F2AE
repne scasw	字					F2AF
repne scasd	双字					F2AF

代码段7-4中的程序要求输入一个串和一个字符，并运用repne和scasb来定位该字符在串中第一次出现的位置，然后显示从该字符开始到串结束的剩余部分。串的长度可用代码段7-3中的过程*strlen*来计算。现在假定*strlen*是单独编译的，lea指令用来载入待查找的串的偏移量，cld保证向前查找。

代码段7-3 串查找程序

```
查找插入到其他串中的子串的程序
; 作者: R. Detmer     日期: 1997年10月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
INCLUDE io.h

cr          EQU    0dh    ; 回车符
Lf          EQU    0ah    ; 换行符

.STACK  4096              ; 保留4096字节的堆栈

.DATA
prompt1     BYTE  "String to search?  ", 0
prompt2     BYTE  cr, Lf, "Key to search for?  ", 0
target      BYTE  80 DUP (?)
key         BYTE  80 DUP (?)
trgtLength  DWORD ?
keyLength   DWORD ?
lastPosn    DWORD ?
failure     BYTE  cr,Lf,Lf,"The key does not appear in the string.",cr,Lf,0
success     BYTE  cr,Lf,Lf,'The key appears at position'
position    BYTE  11 DUP (?)
            BYTE  "   in the string.", cr, Lf, 0

PUBLIC _start                          ; 公开程序入口点
.CODE

_start:     output prompt1             ; 提示输入
            input  target,80           ; 输入目的字串
            lea    eax, target         ; 得到目的字串长度
            push   eax                 ; 长度参数
            call   strlen
            mov    trgtLength,eax      ; 保存目的串长度
            output prompt2             ; 提示输入
            input  key,80              ; 输入key串
            lea    eax, key            ; 得到key串长度
            push   eax                 ; 参数长度
            call   strlen
            mov    keyLength,eax       ; 保存key串长度

; 计算查找目的串的最后一个字符的位置
            mov    eax,trgtLength
            sub    eax,keyLength
            inc    eax                 ; trgtLength - keyLength + 1
            mov    lastPosn, eax
            cld                        ; 从左到右比较

            mov    eax,1               ; 起始位置
whilePosn:  cmp    eax,lastPosn        ; position < = last_posn?
            jnle   endWhilePosn        ; 如果超过最后的位置, 则退出
```

```
            lea     esi,target          ; 目的串的地址
            add     esi,eax             ; 加上位置偏移
            dec     esi                 ; 检查的起始位置
            lea     edi,key             ; key串的地址
            mov     ecx,keyLength       ; 需要比较的长度
            repe cmpsb                  ; 比较
            jz      found               ; 比较成功则退出
            inc     eax                 ; 移到下一个位置
            jmp     whilePosn           ; 下一次比较
endWhilePosn:
            output failure              ; 查找失败
            jmp     quit                ; 退出

found:      dtoa    position,eax        ; 将position中的字符转为ASCII码
            output success              ; 查找成功
quit:
            INVOKE  ExitProcess, 0      ; 返回0并退出

strlen      PROC    NEAR32
; 得到栈内传送过来的字串的长度
; 由EAX返回
            push    ebp                 ; 建立堆栈
            mov     ebp, esp
            pushf                       ; 保存标志位
            push    ebx                 ; 保存ebx
            sub     eax, eax            ; 清零
            mov     ebx, [ebp+8]        ; 子串的地址
whileChar:  cmp     BYTE PTR [ebx], 0   ; 是否为空(NULL)字符?
            je      endWhileChar        ; 如果是空字符就退出
            inc     eax                 ; length + 1
            inc     ebx                 ; 指向下一个字符
            jmp     whileChar           ; 重复循环
endWhileChar:
            pop     ebx                 ; 恢复寄存器和标志位
            popf
            pop     ebp
            ret     4                   ; 返回并释放参数
strlen      ENDP

            END
```

代码段7-4 在字串中查找字符的程序

```
;在串中查找字符的程序
;显示从被查到的字符到串尾的部分
;作者: R. Detmer        日期: 1997年10月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
INCLUDE io.h
EXTRN strlen:NEAR32
PUBLIC _start
```

```
cr          EQU     0dh     ; 回车符
Lf          EQU     0ah     ; 换行符

.STACK  4096                ; 保留4096字节堆栈

.DATA
prompt1     BYTE    "String?  ", 0
prompt2     BYTE    cr, Lf, Lf, "Character?  ", 0
string      BYTE    80 DUP (?)
char        BYTE    5 DUP (?)
label1      BYTE    cr, Lf, Lf, "The rest of the string is-", 0
crlf        BYTE    cr, Lf, 0

.CODE
_start:     output prompt1          ; 提示输入串
            input  string,80        ; 输入串
            lea    eax, string      ; 找到串的长度
            push   eax              ; 长度参数
            call   strlen
            mov    ecx, eax         ; 保存串的长度
            inc    ecx              ; 串长度包括空字节

            output prompt2          ; 提示输入字符
            input  char,5           ; 输入字符
            mov    al, char         ; 把字符放入AL

            lea    edi, string      ; 串的偏移地址
            cld                     ; 向前查找
            repne scasb             ; 如果找不到字符则继续扫描
            dec    edi              ; 回到空字节或者匹配字符位置

            output label1           ; 输出label1
            output [edi]            ; 输出串
            output crlf             ; 跳向新行

            INVOKE  ExitProcess, 0  ; 退出并返回0代码
            END
```

在查找后，无论标志位是否为1，串指令总是会增加索引寄存器的值，所以，目的索引寄存器EDI的值会比预期的值大。如果查找成功，EDI将包含与AL相匹配的字符后的下一个字符的地址；如果ECX的值减到0，则EDI指向串结束后的字符的地址。指令dec edi会考虑上述两种情况，如果是第一种情况，即匹配成功，则EDI倒退到该匹配字符的位置；否则，指向该串结束的空字节。串的长度加了1，因此，空字节应该包含在查找的串中。output宏显示了串的最后部分，其地址在EDI中。

保存串指令stos将AL、AX或者EAX中的一个字节、字或者双字复制为目的串的某个元素。stos指令不影响标志位，因而，它可用rep重复，将同样的值复制到目的串的连续的位置。例如，以下代码将空格保存到串的前30个字节的空间。

```
mov   ecx, 30           ; 30个字节
mov   al,  ' '          ; 要存储的字符
lea   edi, string       ; 串的地址
```

```
cld                     ; 递增
rep stosb               ; 存储空格
```

表7-7给出了stos指令的信息。与前面的重复的串指令一样，486和Pentium机在n = 0时有特定的时钟周期。

表7-7 stos指令（使用EDI）

助记符	元素大小	时钟周期			字节数	操作码
		386	486	Pentium		
stosb	字节	4	5	3	1	AA
stosw	字					AB
stosd	双字					AB
rep stosb	字节	5 + 5*n*	7 + 4*n*	9*n*	2	F3 A6
rep stosw	字					F3 A7
rep stosd	双字					F3 A7

最后介绍的串指令是从串取指令lods。该指令按照串元素的长度复制源串中的元素到AL、AX和EAX寄存器中。lods指令不设置标志位。在lods前可使用rep前缀，但是这样没有多大意义——因为，除了最后一个串元素外，所有的串元素值都将被复制到目的寄存器中的连续的值所取代。在循环中，lods指令非常有用，在处理串元素时，该指令易于每次处理一个串元素。表7-8总结了lods指令，由于本书中没有在lods指令前使用rep重复前缀，所以表7-8没有列出相应的指令形式。

表7-8 lods指令（使用EDI）

助记符	元素大小	时钟周期			字节数	操作码
		386	486	Pentium		
lodsb	字节	5	5	2	1	AC
lodsw	字					AD
lodsd	双字					AD

练习7.2

下面的每一小题，假定数据段内容如下：

```
source  BYTE  "brown"
dest    BYTE  "brine"
```

1. 假如执行如下指令：

```
lea   esi, source
lea   edi, dest
cld
mov   ecx, 5
repne cmpsb
```

假定ESI初始值为00010000、EDI初始值为00010005，在repne cmpsb指令执行后，ESI和EDI中存储的值是多少？ECX中的值是多少？

2. 假如执行如下指令:

```
lea   esi, source
lea   edi, dest
cld
mov   ecx, 5
repe cmpsb
```

假定ESI初始值为00010000、EDI初始值为00010005，在repe cmpsb指令执行后，ESI和EDI中存储的值是多少？ECX中的值是多少？

3. 假如执行如下指令:

```
mov    al, 'w'
lea    edi, dest
cld
mov    ecx, 5
repe scasb
```

假定EDI初始值为00010005，在repe cmpsb指令执行后，EDI中存储的值是多少？ECX中的值是多少？

4. 假如执行如下指令:

```
mov   al, 'n'
lea   edi, dest
cld
mov   ecx, 5
repne scasb
```

假定EDI初始值为00010005，在repne cmpsb指令执行后，EDI中存储的值是多少？ECX中的值是多少？

5. 假如执行如下指令:

```
mov  al, '*'
lea  edi, dest
cld
mov  ecx, 5
rep stosb
```

假定EDI初始值为00010005，在rep stosb指令执行后，EDI中存储的值是多少？ECX中的值是多少？目的串的值是什么？

6. 假如执行如下指令:

```
      lea   esi, source
      lea   edi, dest
      cld
      mov   ecx, 5
for6: lodsb
      inc   al
      stosb
      loop  for6
endFor6:
```

假定ESI的初始值是00010000，EDI的初始值是00010005，在循环以后ESI和EDI的值分别是多少？ECX的值是多少？目的串的内容是什么？

7. 假如执行如下指令：

```
lea   esi, source
lea   edi, dest
cld
mov   ecx, 3
rep   movsb
```

假定ESI的初始值是00010000，EDI的初始值是00010005，在rep movsb指令执行后，ESI和EDI的值分别是多少？ECX的值是多少？目的串的内容是什么？

8. 假如执行如下指令：

```
lea   esi, source + 4
lea   edi, dest + 4
std
mov   ecx, 3
rep   movsb
```

假定ESI的初始值是00010010，EDI的初始值是00010015，在rep movsb指令执行后，ESI和EDI的值分别是多少？ECX的值是多少？目的串的内容是什么？

编程练习7.2

1. 写一个NEAR32的过程*index*，该过程在一个以空字符结束的串中找出某个字符第一次出现的位置。该过程必须有两个参数：（1）要查找的字符（2）串在数据段中的地址。使用堆栈来传递参数：对字符来说，字符存放在一个完整的字的低字节。返回该字符在串内的位置，并放入EAX寄存器；如果在串中找不到该字符，则返回0。其他寄存器的内容不改变。过程*index*不会从堆栈中取出参数。
2. 写一个NEAR32的过程*append*，该过程能把一个以空字符结束的串追加到另一个串的尾部。该过程必须有两个参数：（1）string1在数据段中的地址（2）string2在数据段中的地址。使用堆栈来传递参数。该过程应该把string2复制到string1的尾部，其中，string2的第一个字符取代string1尾部的空字节，依次类推。（注意：在数据部分中，在第一个串的空字节后必须预留足够的空间，以保存第2个串的字符。）该过程所使用的所有寄存器应该保存和恢复。过程*append*不从堆栈中取出参数。
3. 写一个完整的程序，该程序提示并按照“LastName, FirstName”（即“先名后姓”）的格式输入一个人的名字，按照“FirstName LastName”（即“先姓后名”）的格式建立一个新的串。开始时，用逗号和空格来隔开不同的人名，并且，在“First Name”后面除了空字符没有其他字符；在新串中只用一个空格来隔开不同的人名。在内存中生成新的串，并在屏幕上将其显示出来。
4. 写一个完整的程序，该程序提示并按照“Last Name, First Name”（即“先名后姓”）的格式输入一个人的名字，按照“First Name Last Name”（即“先姓后名”）的格式建立一个新的串。开始时，用一个或多个空格来隔开不同的人名，并且，在“First Name”后面可能跟有多个空格字符。在新串中仅用一个空格符来隔开不同的名字。在内存中生成新的串，并在

屏幕上将其显示出来。

5. 写一个完整的程序，该程序提示输入一个串和一个单个的字符。构造一个新串，该新串是由输入串删除输入的单个字符后得到的串。在内存中生成新的串，并在屏幕上将其显示出来。
6. 写一个完整的程序，该程序提示并输入一个句子和一个单词。构造一个新句子，该新句子由输入的句子删除输入的一个单词后得到。在内存中生成这个新句子，并在屏幕上将其显示出来。
7. 写一个完整的程序，该程序提示并输入一个句子和两个单词。构造一个新句子，在提示输入的句子中，每次出现第一个单词都将用第二个单词取代，从而得到新句子。在内存中生成新的句子，并在屏幕上将其显示出来。

7.3 字符转换

有时，字符型数据使用某种格式表示，但在处理时又需要另外一种格式，例如：当在两个计算机系统之间传输字符时，一个系统使用ASCII字符编码，而另一个系统使用EBCDIC字符编码。另外，当传输字符到某个设备，而这个设备不能处理所有的编码时，则需要转换字符编码。有时候，用易接受的编码来代替不合适的编码比完全删除它们要容易。

80x86指令集包括xlat指令，该指令可将一种字符转换为另一种字符，与其他串处理指令一起使用时，xlat指令能容易地转换串中所有的字符。

xlat指令仅仅需要一个字节的目标代码，操作码为D7。该指令在80386中需要5个时钟周期来完成，而在80486或者Pentium机中需要4个时钟周期。在指令执行之前，要转换的字符需存放在AL寄存器中。指令执行时，需要使用数据段中的一个转换表，利用该表来查找AL中转换的字节。该转换表一般包含256个字节的数据，每个字节可能是AL中8位的值。表中偏移量为0的字节，即第一个字节——该字符转换为00。偏移量为1的字节转换为01。通常，xlat用被转换的字符作为偏移量放在表中，并且用偏移量处的字节取代AL中的字符。

xlat指令没有操作数。EBX寄存器必须包含转换表的地址。

代码段7-5给出了一个小程序，该程序在原地转换串中每一个字符；也就是说，利用原有的空间，通过转换来取代原来的每个字符。程序的关键是转换表和指令的顺序。

```
            mov   ecx, strLength    ; 串的长度
            lea   ebx, table        ; 转换表的地址
            lea   esi, string       ; 串的地址
            lea   edi, string       ; 目的串和源串同一地址
forIndex:   lodsb                   ; 复制下一个字节到AL
            xlat                    ; 转换字符
            stosb                   ; 转换后的字符复制到源串
            loop  forIndex          ; 循环处理所有的字符
```

代码段7-5 转换程序

```
; 将大写字母转换为小写，小写字母不改变
; 字母和数字的情况，将其他字符转换为空格
; 作者：R. Detmer    日期：1997年10月

.386
.MODEL FLAT
```

```
ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
INCLUDE io.h
PUBLIC _start
cr          EQU     0dh     ; 回车符
Lf          EQU     0ah     ; 换行符

.STACK  4096                ; 保留4096字节的堆栈

.DATA
string      BYTE    'This is a #!$& STRING',0
strLength   EQU     $ - string - 1
label1      BYTE    'Original string ->',0
label2      BYTE    cr, Lf, 'Translated string ->',0
crlf        BYTE    cr, Lf, 0
table       BYTE    48 DUP (' '), '0123456789', 7 DUP (' ')
            BYTE    'abcdefghijklmnopqrstuvwxyz', 6 DUP (' ')
            BYTE    'abcdefghijklmnopqrstuvwxyz', 133 DUP (' ')

.CODE
_start:     output label1            ; 显示label1
            output string
            output crlf
            mov     ecx, strLength  ; 串的长度
            lea     ebx, table      ; 转换表的地址
            lea     esi, string     ; 串的地址
            lea     edi, string     ; 目的地址和源串相同
forIndex:   lodsb                   ; 复制下一个字节到AL
            xlat                    ; 转换字符
            stosb                   ; 从转换后的字符复制到源串
            loop    forIndex        ; 循环

            output label2           ; 显示label2
            output string
            output crlf

            INVOKE  ExitProcess, 0
            END
```

这些指令实现下面的一个for循环:

```
for index:=1 to stringLength loop
    取源串字符到AL;
    转换AL中的字符;
    将转换后AL中的字符复制到源串
end for;
```

这个程序的一个新特点是使用了地址计数器符号$。回顾一下汇编器是如何计算地址的，起始地址从00000000开始，每生成目标代码时，计数器就加1。在汇编时，如果遇到$符号，则$符号就是计数器的值。在该程序中，地址计数器符号$指的是该串空字节的前一个字节的地址。由于符号string实际上表示其首地址，表达式string-$就是串的长度（包括空字节）。strLength的值等于$ - string - 1（不包括空字节）。汇编过程将在第9章讨论。

该程序将一个ASCII码转换成另外一个ASCII码，其中，大写字母转换成小写字母，小写

字母和数字保持不变，其他字符转换成空格。构建这样一个表需要查看ASCII代码表（见附录A）。该程序中可如下定义转换表:

```
table       BYTE 48 DUP (' '), '0123456789', 7 DUP(' ')
            BYTE 'abcdefghijklmnopqrstuvwxyz', 6 DUP(' ')
            BYTE 'abcdefghijklmnopqrstuvwxyz', 133 DUP(' ')
```

注意，这里的确定义了256个字节。回顾一下，一条BYTE指示性语句用来存放每一个ASCII码的字符操作数。表中的前48个字节包含了空格字符的ASCII码，即20_{16}。因此，如果存放在AL寄存器中的代码表示任意的前48个ASCII码字符，比如，控制字符或者是从20_{16}（空格）到$2F_{16}$（/）之间的任意一个可打印的字符都将转换为空格。

注意，一个字符可以转换为它自身。比如，对于数字处理时，即ASCII码30_{16}到39_{16}是数字0到9，出现在偏移量为30_{16}到39_{16}。ASCII码表中的@前（包括@在内）的7个字符，其中的每一个字符都会转换为空格。@后的ASCII码字符是大写字母，其后的下一列是小写字母的ASCII码。例如，表中包含了在偏移量41_{16}处的61_{16}，因此，是大写字母A（A的ASCII码是41_{16}），它将转换为小写字母a（a的ASCII码是61_{16}）。接下去的6个空格在偏移量91_{16}（[）到96_{16}（'），因此，其中的每一个字符转换成空格。每一个小写字母的ASCII码在和它ASCII码值相等的偏移量处汇编，因此每一个小写字母转换为它自身。最后，转换表包括了133个空白的ASCII码，其中的五个是{、|、}、~、del的转换码，其余的128个没有相应的ASCII码。

图7-1给出了代码段7-5中程序的输出。注意，“strange”字符没有被删除，而是被空白取代了。

```
Original string   ->This is a #!$& STRING

Translated string ->this is a      string
```

图7-1 转换程序的输出

练习7.3

1. 下面是一个十六进制/EBCDIC的部分转换表:

```
81a  C1 A  40 space
82b  C2 B  4B.
83c  C3 C  6B,
84d  C4 D
85e  C5 E  F0 0
86f  C6 F  F1 1
87g  C7 G  F2 2
88h  C8 H  F3 3
89i  C9 I  F4 4
91j  D1 J  F5 5
92k  D2 K  F6 6
93l  D3 L  F7 7
94m  D4 M  F8 8
95n  D5 N  F9 9
96o  D6 O
```

```
97p  D7 P
98q  D8 Q
99r  D9 R
A2s  E2 S
A3t  E3 T
A4u  E4 U
A5v  E5 V
A6w  E6 W
A7x  E7 X
A8y  E8 Y
A9z  E9 Z
```

请给出一个转换表，该转换表可用xlat指令把EBCDIC代码中的字母、数字、空格、点和逗号转换成相应的ASCII码，并把其他的每个EBCDIC代码转换成空字符。

2. 请给出一个转换表，该转换表可用xlat指令把小写字母转换成相应的大写字母，其余的字母不变。
3. 下面的指令也可实现xlat指令的功能。

```
movzx   eax, al            ; 清除EAX中高位
mov     al, [ebx + eax]    ; 从表中复制新字符到AL
```

假设[ebx + eax]指的是EBX和EAX内容的和在内存中的地址，解释为什么这两个指令和单独使用xlat指令是一样的。

编程练习7.3

在美国，十进制数的书写是用十进制小数点分开小数部分和整数部分，整数部分从小数点起每3个位置用一个逗号隔开。但在很多欧洲国家，十进制数的书写是用小数点和逗号分开的，但小数点和逗号的作用相反。比如，数字1,234,567.89会被写成1.234.567,89。写一个程序，能交换逗号和点，转换一个串的字符，将其从一种格式转换为另外一种格式。使用带转换表的xlat指令，将点转换为逗号，逗号转换为点，数字转换为自身，其他的字符转换成空格。提示并输入要转换的数字，然后转换该串，并用一个适当的标号来显示新的数字格式。

7.4 二进制补码整数转换为ASCII码串

宏dtoa和宏itoa用于将二进制补码的整数转换为可输出的ASCII码字符串，这些转换的代码非常类似，本节考察代码相对简短的宏itoa。

宏itoa展开为如下的一系列指令:

```
push  ebx              ; 保存EBX
mov   bx, source
push  bx               ; 源参数
lea   ebx, dest        ; 目的地址
push  ebx              ; 目的参数
call  itoaproc         ; 调用过程itoaproc
pop   ebx              ; 恢复EBX
```

在源值和目的地址入栈后，这些指令调用过程*itoaproc*。扩展宏中所用到的是真正的源值和目的地址，而不是source和dest。因此，不必担心任何寄存器的内容发生改变，EBX最初保存在堆栈中，并在这些指令结束后被恢复。跟在call指令后的指令add esp, 6可能会改变标志位，所以，过程*itoaproc*从堆栈中移出参数。

过程*itoaproc*实现将二进制补码整数转换为ASCII码。该过程汇编时包含在文件IO.OBJ中。代码段7-6给出了来自IO.ASM文件的源代码。该过程开始时，把所有要改变的寄存器的值保存在堆栈中，同时保存标志寄存器，这样，调用过程*itoaproc*将不改变标志的设置，在过程返回前，立即恢复标志寄存器和其他寄存器的值。

代码段7-6 整数转换成ASCII码的过程

```
; itoaproc(source, dest)
; 转换整型数为6个字符长的串，并保存在destination地址处

itoaproc    PROC    NEAR32
            push    ebp                     ; 保存基址指针
            mov     ebp, esp                ; 建立堆栈
            push    eax                     ; 保存过程使用的寄存器
            push    ebx                     ; 使用
            push    ecx                     ; 过程
            push    edx
            push    edi
            pushf                           ; 保存标志位

            mov     ax, [ebp+12]            ; 第一个参数（整数）
            mov     edi, [ebp+8]            ; 第二个参数（目的地址）
ifSpecial:  cmp     ax,8000h                ; 特殊处理-32768
            jne     EndIfSpecial            ; 如果不是-32768，则作一般处理
            mov     BYTE PTR [edi],'-'      ; 人工存入-32768的ASCII
            mov     BYTE PTR [edi+1],'3'    ; -32768
            mov     BYTE PTR [edi+2],'2'
            mov     BYTE PTR [edi+3],'7'
            mov     BYTE PTR [edi+4],'6'
            mov     BYTE PTR [edi+5],'8'
            jmp     ExitIToA                ; 特殊处理
EndIfSpecial:

            mov     dx, ax                  ; 保存源数

            mov     al,' '                  ; 存空格于
            mov     ecx,5                   ; 前五个
            cld                             ; 字节
            rep stosb                       ; 目的地址

            mov     ax, dx                  ; 取得源数
            mov     cl,' '                  ; 赋值默认符号位(空格表示正号+)
IfNeg:      cmp     ax,0                    ; 检查源数符号位
            jge     EndIfNeg                ; 如果非负数则跳过
            mov     cl,'-'                  ; 赋值负数的符号位
            neg     ax                      ; 现在AX中的数>0
EndIfNeg:
```

```
            mov     bx,10               ; 除数
WhileMore:  mov     dx,0                ; 扩展源数为双字长度
            div     bx                  ; 被10除
            add     dl,30h              ; 转换余数为字符
            mov     [edi],dl            ; 将字符存入字符串
            dec     edi                 ; 向前移动，指向下一个位置
            cmp     ax,0                ; 检查商
            jnz     WhileMore           ; 如果商不为0则继续转换

            mov     [edi],cl            ; 插入表示符号的空格''或者"-"
ExitITOA:   popf                        ; 恢复标志位和其他寄存器
            pop     edi
            pop     edx
            pop     ecx
            pop     ebx
            pop     eax
            pop     ebp
            ret     6                   ; 退出
itoaproc    ENDP
```

该过程的基本思想是将一个数字重复地用10来整除，从左到右创建字符串，用余数确定最右边的字符。例如，用10整除数字2895（$0B4F_{16}$）得到余数5，商是289（0121_{16}），用新得到的商289再次重复该过程。这种方法对正数非常有效，但对于一个负数，在开始做这种除法循环之前，要将该负数取绝对值。对于更复杂的情况，8000_{16}是负数$-32\ 768_{10}$，但是+32 768不能用一个16位字长的2进制补码表示。

在标准入口代码后，参数值复制到AX，目的地址复制到EDI，然后，该过程判断是否是特殊的情况，即要转换的数为8000_{16}。如果是这种情况，那么，-32 768的ASCII编码一次一位地移到目的地址，该目的地址在EDI中。负的符号放在EDI寄存器中，因此，寄存器间接寻址能用于将该符号存放到正确的内存单元中。字符3的地址是EDI包含的地址值指向的下一个字节，其地址可用[edi + 1]定位。余下的4个字符类似存放在合适的地方，然后该过程退出。

接着，该过程将起始的5个空白字符存放在6字节长的目的地址域中，该过程使用一条指令rep stosb来实现，其中EDI指向目的地址中连续的字节。注意，EDI从左开始，一直指向目的地址中最后一个字节。

然后，该过程在CL寄存器中存储正确的“符号”。对一个大于或者等于0的数，使用一个空格符号；对于一个负数，使用负数符号（-）。负数取反，它的绝对值以后处理。

最后，执行该过程的核心部分。除数10存放在BX寄存器中。把0放入DX中，将非负数扩展为双字。用BX中的10来整除得到0到9的余数，并将余数，即该数的最后一位十进制数存放在DX中。该余数与30_{16}相加，得到其对应的ASCII码；回顾一下，数字0到9的ASCII码就是30_{16}到39_{16}。mov指令使用寄存器间接寻址，把字符存放到目的串中对应的位置，同时EDI减1，指向左边字符存储的下一位置。

该过程重复执行，直到商为0。最后，存储在CL 的“符号”（空格或-）复制到最后一位数字代码的左边。如果左边有空位，先前已经用空格补齐了。

练习7.4

1. 为什么过程*itoaproc*使用6字节长的目的串？

2. 假设负数在过程*itoaproc*做除法循环之前没有发生改变，并且在循环中使用idiv指令，而不是div指令。回顾一下，当一个负数被正数整除，商和余数都是负的。例如，－1273 = 10*(－127) + (－3)。修改该除法循环的其他部分，以便对正数和负数都能得出正确的ASCII码。

编程练习7.4

1. 重写过程*itoaproc*，增加一个长度参数。新的itoaNEW将是一个NEAR32过程，有3个参数，通过堆栈传递：
   (1) 要转换为ASCII码字符的2进制补码数（字长）
   (2) ASCII码串的地址（双字长）
   (3) ASCII码串的长度（字长）

   从数据段的偏移量处开始，将该数转换为ASCII码字符串。正数前不使用空格。如果长度小于要显示数字的实际字符的个数，其他部分用符号#来填充。如果长度远比数字字符数需要的大，在数字的左边用空格补齐。该过程将从堆栈中移出参数，并且不修改寄存器的值。
2. 写一个NEAR32的过程*hexString*，将一个32位的整数转换为一个8字符的字符串，该字符串表示其对应的16进制数。（也就是说，输出的字符是0～9和A～F，没有空字符。）该过程有两个参数，通过堆栈传递：
   (1) 32位的整数
   (2) 目的串地址

   该过程将从堆栈中移出参数，并不修改寄存器的值。（用16整除得到的余数对应其十六进制数的最右边的一位。）
3. 写一个NEAR32的过程*binaryString*，将32位的整数转换为一个32个字符长的字符串，该字符串表示对应的2进制数。该过程有两个参数，通过堆栈传递：
   (1) 32位整数
   (2) 目的串地址

   该过程将从堆栈中移出参数，并不修改寄存器的值。（用2整除后得到的余数对应其二进制数的最右边一位。）

7.5 其他体系结构：CISC和RISC设计

早期的数字计算机有非常简单的指令集。在20世纪60年代，设计者开始用微代码编写指令，这使得更复杂的指令成为可能。同时，高级编程语言开始流行，但是，语言编辑器还相当简单，这使得迫切需要机器语言语句能直接实现高级语言语句，因而，需要具有更多复杂指令的计算机体系结构。

Intel 80x86机器使用复杂指令集计算机（CISC）的设计。像本章讨论的串指令，没有出现在早期的计算机中。CISC机器也有多种形式的内存寻址模式。尽管迄今为止只介绍了其中的一些模式，但在这方面，80x86系列是典型的代表。通常，CISC指令执行时需要占用几个时钟周期。

精简指令集计算机（RISC）设计开始出现于20世纪80年代。这些机器的指令和内存寻址的模式相对较少。这些指令很简单，任何一条指令都能在一个时钟周期内完成。随着编译器

技术的改进，使得生成RISC机器的有效代码成为可能。当然，要实现一个给定的高级语言语句，RISC机器比CISC机器需要更多的指令来完成。但是，执行所有的操作时，RISC机器比CISC机器会更快，因为其指令都是单个执行的。

在RISC体系中，所有指令都是同样的格式，也就是说，相同数量的字节数用相同的模式编码，这与CISC体系不一样。如果80x86芯片是用RISC设计，那么本书就不会提出诸如“有多少时钟周期”或者“有多少字节”这样的问题了。第9章将介绍多种80x86指令格式读者可能更向往RISC机器的简单性了。

RISC设计的一个显著的特点是寄存器集相对较大（有时超过500个），但一次可见的仅是其中的一小部分（经常是32个）。寄存器用来传递参数给过程，并且，在程序调用中存储变量的寄存器覆盖在过程中接收参数值的寄存器。对于调用的程序和过程之间的通讯，这提供了一个简单而有效的方法。

CISC和RISC的设计各有优缺点，很难说哪一种更好。但是，常用的Intel 80x86和Motorola 680x0系列都是CISC设计。因此，至少在不远的将来，会更多涉及到CISC系统。

本章小结

字串指的是在内存中连续的字节、字或者双字集。80x86指令集包括5个串操作指令：movs（从源地址传送或复制到目的地址）、cmps（比较两个串）、scas（扫描串中某个特定的元素）、stos（在串中存储一个给定的值）和lods（复制串中元素到EAX、AX或者AL中）。每个助记符都是以b，w或者d结尾的形式来给出串元素的大小。

一条串指令一次只能操作一个串元素。当涉及到源串时，源索引寄存器ESI包含了该串元素的地址。当涉及到目的串时，目的索引寄存器EDI包含了该串元素的地址。在访问串时，根据方向标志位DF是0还是1，决定索引寄存器的值是递增还是递减；cld和std指令可用于设置方向标志位的值。

一些串指令使用重复前缀rep、repe（repz）和repne（repnz）来自动重复执行。执行指令的重复次数保存在ECX寄存器中。有条件的重复次数存放在ECX中，但是，如果ZF标志位被置为满足条件的值，也将终止指令的执行。用cmps和scas指令将ZF置1或置0。

xlat指令用来转换串中的字符。该指令要求有一个256字节长的转换表，源字节00转换到目的地址开始处，源字节FF转换到目的地址结束处。应用xlat指令可实现诸如将ASCII码转换为EBCDIC码，或者在给定字符的编码系统下，改变字符的大小写。

宏itoa扩展的代码调用过程itoaproc。该过程基本上是用10来重复整除一个非负数，并用余数来得到目的串中最右边的字符。

80x86芯片是复杂指令集计算机体系（CISC）的代表。它包括很多复杂的指令，并提供很多不同的寻址模式。精简指令集计算机体系（RISC）执行相对较少和较简单的指令，并且寻址方式有限。尽管RISC计算机需要更多的指令来完成一个任务，由于RISC计算机简单的指令执行更快，所以执行速度通常更快些。

第8章 位 运 算

一台计算机有许多集成电路，这些电路能使计算机完成其功能。每个芯片由少许到上千数量不等的逻辑门组成，每个基本电路可执行由电子状态表示的位的与、或、异或、非等布尔运算，CPU通常是PC中最复杂的集成电路。

上一章已经介绍了一些80x86微处理器指令，其中包括数据传送、运算操作、串操作、分支和子程序调用等指令。80x86（以及其他大多数CPU）一次也能执行多对位的布尔运算指令。本章定义了布尔运算，并详述了实现这些运算的80x86指令，包括引起位模式变化的指令，如在字节、字、或双字中的移位和循环移位指令，或者从一个地址单元转移到另一个单元指令。虽然位运算指令非常简单，但是，由于它们能够提供一些在高级语言中很少用的控制，因此，在汇编语言设计中位运算指令被广泛应用。本章给出了几个应用实例，包括一个命名为atoi的宏过程，该过程在一些地方使用了位运算指令。

8.1 逻辑运算

许多高级语言允许使用布尔类型的变量，也就是说，变量可以存储true或false值。实际上，所有的高级语言都允许在条件语句（if）中使用带布尔值的表达式。在汇编语言中，布尔值true用位值1表示，布尔值false用位值0表示。图8-1给出了用位值做操作数的布尔运算的定义。或（or）运算有时称为“包含或”，以便与“异或”（xor）区分。or和xor的惟一区别在于两个1运算时：1or 1得到1；1 xor 1得到0。也就是说，异或值为1相当于两个操作数中有一个为真，但不是两个都为真。

(a) and操作

bit1	*bit2*	*bit1* and *bit2*
0	0	0
0	1	0
1	0	0
1	1	1

(b) or操作

bit1	*bit2*	*bit1* or *bit2*
0	0	0
0	1	1
1	0	1
1	1	1

(c) xor操作

bit1	*bit2*	*bit1* xor *bit2*
0	0	0
0	1	1
1	0	1
1	1	0

(d) not操作

bit	not *bit*
0	1
1	0

图8-1 逻辑运算的定义

80x86有与and、或or、异或xor和非not指令来实现逻辑运算。这些指令的形式如下:

```
and  目的操作数，源操作数
or   目的操作数，源操作数
xor  目的操作数，源操作数
nor  目的操作数
```

前三条指令可用于两个双字、两个字或两个字节的操作数之间，对两个操作数相应的位进行逻辑运算。例如，当执行指令 and bx, cx时，寄存器BX中的第0位和寄存器CX中的第0位进行与运算，寄存器BX中的第1位和寄存器CX中的第1位进行与运算，如此类推，直到寄存器BX和寄存器CX中的第15位进行与运算，所有16位and运算的结果按位存入目的地址。

not指令只有一个操作数。它的作用是把操作数的每一位由0改为1，由1改为0。例如，如果AH寄存器存储的数为10110110，那么执行not ah指令后，AH内寄存器中的内容变为01001001。有时，not运算也称为“对操作数取补”。

not指令不影响任何标志位，但是，其他三条布尔指令都影响CF、OF、PF、SF、ZF和AF等标志位。其中进位标志位CF和溢出标志位OF都置0；辅助进位标志位AF可能被改变，但是没有定义；根据运算的结果，奇偶校验位PF、符号标志位SF和零标志位ZF置1或者置0。例如，如果运算结果的每一位都是0，那么ZF置1；如果任意一位不为0，那么ZF将置0。

and、or和xor指令都支持相同类型的操作数，执行时需要相同的时钟周期，并且需要相同字节的目标代码，表8-1对这些指令进行了总结。表8-2列出了not指令。

表8-1 and、or和xor指令

目的操作数	源操作数	时钟周期			字节数	操作码		
		386	486	Pentium		and	or	xor
8位寄存器	8位立即数	2	1	1	3	80	80	80
16位寄存器	8位立即数	2	1	1	3	83	83	83
32位寄存器	8位立即数	2	1	1	3	83	83	83
16位寄存器	16位立即数	2	1	1	4	81	81	81
32位寄存器	32位立即数	2	1	1	6	81	81	81
AL	8位立即数	2	1	1	2	24	0C	34
AX	16位立即数	2	1	1	3	25	0D	35
EAX	32位立即数	2	1	1	5	25	0D	35
字节存储器	8位立即数	7	3	3	3+	80	80	80
字存储器	8位立即数	7	3	3	3+	83	83	83
双字存储器	8位立即数	7	3	3	3+	83	83	83
字存储器	16位立即数	7	3	3	4+	81	81	81
双字存储器	32位立即数	7	3	3	6+	81	81	81
8位寄存器	8位寄存器	2	1	1	2	22	0A	32
16位寄存器	16位寄存器	2	1	1	2	23	0B	33
32位寄存器	32位寄存器	2	1	1	2	23	0B	33
8位寄存器	字节存储器	6	2	2	2+	22	0A	32
16位寄存器	字存储器	6	2	2	2+	23	0B	33
32位寄存器	双字存储器	6	2	2	2+	23	0B	33
字节存储器	8位寄存器	7	3	3	2+	20	08	30
字存储器	16位寄存器	7	3	3	2+	21	09	31
双字存储器	32位寄存器	7	3	3	2+	21	09	31

表8-2 not指令

目的操作数	时钟周期			字节数	操作码
	386	486	Pentium		
8位寄存器	2	1	1	2	F6
16位寄存器	2	1	1	2	F7
32位寄存器	2	1	1	2	F7
字节存储器	6	3	3	2+	F6
字存储器	6	3	3	2+	F7
双字存储器	6	3	3	2+	F7

值得注意的是，表8-1几乎和表4-5是一样的，表4-5列举的是add和sub指令。同样，表8-2和表4-7相似，而表4-7列举的是neg指令。在以上两种情况下，可使用的操作数类型相同，执行时间相同，甚至许多操作码也一样（回顾一下，当操作码相同时，add、sub、or和xor指令之间的区别）。

下面举例子说明逻辑指令是怎样工作的。要计算结果，首先必须把每个十六进制的数变为二进制的数，相对应的位对之间进行逻辑运算，最后再把结果转换为十六进制，大多数十六进制计算器可以直接进行逻辑运算。

执行前	指令	按位运算	执行后
AX: E2 75 CX: A9 D7	and ax, cx	1110 0010 0111 0101 1010 1001 1101 0111 1010 0000 0101 0101	AX [A0 | 55] SF 1 ZF 0
DX: E2 75 value: A9 D7	or dx, value	1110 0010 0111 0101 1010 1001 1101 0111 1110 1011 1111 0111	DX [EB | F7] SF 1 ZF 0
BX: E2 75	xor bx, 0a9d7h	1110 0010 0111 0101 1010 1001 1101 0111 0100 1011 1010 0010	BX [4B | A2] SF 0 ZF 0
AX: E2 75	not ax	1110 0010 0111 0101 0001 1101 1000 1010	AX [1D | 8A]

每条逻辑指令都有很多用途，and指令应用之一是对目的地址中的内容选定的位清零。注意：任意一位值和1进行与运算，其结果仍是原值；任意一位值与0进行与运算，其结果必定为0。因此，要把一个字节或者字中的选定的位清零，只要让它和一个要清零的对应位为0，不需改变的位为1的数进行与运算就可以了。

例如，除寄存器EAX内的数最后四位外，将其余所有位清零，可以用以下的指令：

```
and    eax, 0000000fh        ;  将EAX中的前28位清零
```

如果EAX中原来的内容是4C881D7B，那么与运算的结果就是0000000B:

```
0100 1100 1000 1000 0001 1101 0111 1011 4C881D7B
0000 0000 0000 0000 0000 0000 0000 1111 0000000F
0000 0000 0000 0000 0000 0000 0000 1011 0000000B
```

在0000000fh 的7个0中只有一个是必要的，但是编码7个0可以帮助理解这个操作数的用途。最后的十六进制数f对应的是二进制数的1111，4个1保证EAX中的后四位不会改变。

在逻辑指令中用来改变位值的值通常称为掩码。微软汇编器MASM支持十进制、十六进制、二进制和八进制的数值。掩码通常使用十六进制和二进制表示，因为这样的位模式和二进制值对应，也容易得出相应的十六进制值。

如上所述，and指令可将字节或字中的选定位清零。在不改变其他位的情况下，or指令可将一个字节或字中的选定位置1。显然，1无论是与0还是1进行or运算，结果还是1；如果一个操作数为0时，or运算的结果是另一个操作数。

在不改变其他位的情况下，异或指令可对一个字节或字中选定的位按位取补，0 xor 1得到1，1 xor 1得到0；也就是说，任一操作数与1进行xor运算，其结果是对该操作数取反。

逻辑指令的第二个应用是实现高级语言的布尔运算。存储器中的一个字节可以存放8个布尔值。设flags为这个字节，那么语句:

```
and flags, 11011101b        ;  flag5:= false;  flag1:= false
```

将false赋值给第一位和第五位，其他位的值不变（位数是从右到左计数，从最右边的第0位开始）。

如果存储器中的flags存储了8个布尔值，or指令可以给它的任意选定的位赋值true。例如，语句:

```
or  flags, 00001100b        ; flag3:= true;  flag2:= true
```

将第二位和第三位设置为true，而其他位保持不变。

如果存储器中的flags存储了8个布尔值，xor指令可以对选定的位取反。例如，要实现flag6: = NOT flag6；可以用以下语句实现:

```
xor flags, 01000000b        ; flag6 := not flag6
```

逻辑指令的第三个应用是执行一些算术运算。假设寄存器EAX的值是一个无符号整数。表达式（value mod 32）可以用下面的几条指令计算:

```
mov   edx, 0      ; 把值扩展到4个字
mov   ebx, 32     ; 除数
div   ebx         ; 值除以32
```

执行上述指令，余数（value mod 32）将放在寄存器EDX中。下面是另一种方法把同样的结果放在寄存器EDX，但是，商没有放在EAX中。

```
mov   edx, eax            ; 把值复制到DX
and   edx, 0000001fh      ; 计算模32值
```

后一种方法比前一种方法的效率更高（参见练习2），因为EDX中的值是一个二进制数，计算如下:

$$bit31 \times 2^{31} + bit30 \times 2^{30} + \cdots + bit2 \times 2^2 + bit1 \times 2 + bit0$$

因为从bit31 × 2^{31}到bit5 × 2^5的每一项都可以被32(2^5) 约分，除以32的余数是由后五位表示的组合；这五位的左边用0000001F屏蔽掉。其他相似的指令也可以这样计算，只要mod的第二个操作数是2的幂。

逻辑指令的第四个应用是处理ASCII码。回顾一下，ASCII码值30_{16}表示阿拉伯数字的0，值31_{16}表示1，依此类推，39_{16}表示9。假设寄存器AL存放的是一个数字的ASCII码，对应的整数值必须放到寄存器EAX中。如果EAX中的值的高24位已知为0，那么指令:

```
sub  eax, 00000030h     ; 把ASCII码转变为整数
```

就可以将这个ASCII码转变为对应的整数。若EAX中的值的高24位未知，则用指令:

```
and  eax, 0000000fh     ; 把ASCII码转变为整数
```

更为安全。它能确保EAX中的值，除后四位外，其余所有位清零。例如，如果寄存器EAX中的值为5C3DF036，高位任意，AL中存放的是6的ASCII码，那么执行指令and eax, 0000000fh，其结果00000006存入EAX。

or指令可以把0到9之间的整型值转变为相应的ASCII码。例如，如果整数放在BL中，那么下面的指令可以把BL中的值转变为相应的ASCII码。

```
or  bl, 30h     ; 把整数变为ASCII码
```

如果BL的内容是04，则or指令的结果是34:

```
0000 0100     04
0011 0000     30
0011 0100     34
```

在80x86处理器中，指令add bl, 30h可以用同样的时钟数周期数和目标代码完成or指令的上述功能。但是，对有些CPU而言，or运算比加法指令的效率更高。

xor指令可以用来改变ASCII码字母的大小写。假设寄存器CL内有大写或小写的ASCII码的字母。ASCII码的大写字母和相应的小写字母的区别仅仅在于第五位的值。如，大写字母S的码是53_{16} (01010011_2)，小写字母s的代码是73_{16} (01110011_2)。指令

```
xor  cl, 00100000b     ; 改变CL内的字母的大小写
```

把寄存器CL内的第五位取反，从而对ASCII码的值进行大小写的转换。

80x86指令集包括test指令，test指令除了不改变目的操作数之外，它的功能与and指令相同。也就是说，指令test惟一的任务是设置标志位（要记住，cmp指令实质上相当于设置标志位的sub指令，但是cmp指令不改变目的操作数）。test指令的应用之一是检查一个字节或字的特定位的值。下面的指令检查寄存器DX的第13位的值:

```
test  dx, 2000h     ; 检查13位
```

值得注意的是，十六进制的2000等于二进制0010 0000 0000 0000，它的第13位等于1。通常，指令test后面都会跟随jz或者jnz指令，其作用是根据第13位是0还是1，跳转到相应的目的地。

test指令也可以用来得到寄存器中的值的信息，例如:

```
test  cx, cx     ; 设置cx内标志位的值
```

寄存器CX的内容和自己进行与运算的结果还是原值（任意位和它自己执行and操作的结果都是本身）。标志位的设置取决于CX内的值。指令

```
and  cx, cx     ; 设置cx内标志位的值
```

可以达到相同的目的，并且效率也相当。但是使用test可以使指令的作用一目了然，即仅仅用于测试。

表8-3列出了各种形式的test指令。它们和指令and、or以及xor几乎相同。当源操作数是在存储器中时，目的操作数只能是累加器。但是，MASM允许指定任意寄存器为目的操作数，并且MASM允许调换操作数，将存储器操作数作为目的操作数。

表8-3 指令test

目的操作数	源操作数	时钟周期			字节数	操作码
		386	486	Pentium		
8位寄存器	8位立即数	2	1	1	3	F6
16位寄存器	16位立即数	2	1	1	4	F7
32位寄存器	32位立即数	2	1	1	6	F7
AL	8位立即数	2	1	1	2	A8
AX	16位立即数	2	1	1	3	A9
EAX	32位立即数	2	1	1	5	A9
字节存储器	8位立即数	5	2	2	3+	F6
字存储器	16位立即数	5	2	2	4+	F7
双字存储器	32位立即数	5	2	2	6+	F7
8位寄存器	8位寄存器	2	1	1	2	84
16位寄存器	16位寄存器	2	1	1	2	85
32位寄存器	32位寄存器	2	1	1	2	85
字节存储器	8位寄存器	5	2	2	2+	84
字存储器	16位寄存器	5	2	2	2+	85
双字存储器	32位寄存器	5	2	2	2+	85

练习8.1

1. 下面每小题，假定给出指令执行前的值，求执行后的值。

	执行前	指令	执行后
(a)	BX: FA 75 CX: 31 02	and bx, cx	BX, SF, ZF
(b)	BX FA 75 CX 31 02	or bx, cx	BX, SF, ZF
(c)	BX FA 75 CX 31 02	xor bx, cx	BX, SF, ZF
(d)	BX FA 75	not bx	BX,
(e)	AX FA 75	and ax, 000fh	AX, SF, ZF
(f)	AX FA 75	or ax, 0fff0h	AX, SF, ZF

(g) AX FA 75 xor ax, 0ffffh AX, SF, ZF

(h) AX FA 75 test ax, 0004h AX, SF, ZF

2. 当EAX寄存器中value是无符号整数时，本节给出了value mod 32的两种计算方法:

```
mov  edx, 0            ; 把值扩展为4字长
mov  ebx, 32           ; 除数
div  ebx               ; 值除以32
```

和

```
mov  edx, eax          ; 把值复制到DX
and  edx, 0000001fh    ; 计算模32的值
```

分别计算每种方法在Pentium处理器上执行指令时所需的时钟周期数和目标代码所需的字节数。

3. 假设EAX寄存器中value是无符号整数。给出适当的指令计算value mod 8，并把结果放到寄存器EBX中，不改变EAX寄存器内容。

4. 假设双字长flags的每一位表示一个布尔值，0位对应flag0，依此类推，31位对应flag31。根据下面每条语句，给出一条80x86指令实现这条语句。

(a) `flag2 := true;`

(b) `flag5 := false ; flag16 := false ; flag19 := false ;`

(c) `flag12 := NOT flag12`

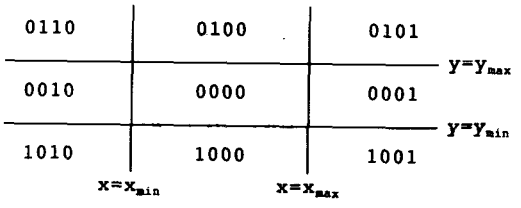
5. (a) 假设寄存器AL中有一个大写字母的ASCII码。给出一条逻辑指令（除了xor外）把AL中的内容改为相应的小写字母。

(b) 假设寄存器AL中有一个小写字母的ASCII码。给出一条逻辑指令（除了xor外）把AL中的内容改为相应的大写字母。

编程练习8.1

1. Pascal程序设计语言中的预定义函数odd有一个双字整数参数和一个返回值，如果是奇数时，该返回值为true，而为偶数时返回值为false。用汇编语言编写一个NEAR32过程实现这个功能，EAX寄存器中返回值－1代表true，0代表false。除了EAX外，该过程不能改变的其他寄存器内容。用适当的逻辑指令产生返回值，该过程必须能够取出堆栈中的参数。

2. 用二维平面制图法设计一个平面矩形区域显示在显示器上；区域外的点要忽略掉。用x = Xmax，x = Xmin，y = Ymin，y = Ymax这四条线来把这块区域划分如下:



图中每个（x，y）坐标点对应一个outcode（或者说区域码）。这个四位码按以下规则赋值:
如果坐标点超出区域的右边，（最右边）第0位的值为1，即$x > x_{max}$；否则，第0位的值为0。
如果坐标点在区域的左边（$x < x_{min}$），第1位的值为1。

如果坐标点高出区域（$y<y_{man}$），第2位的值为1。

如果坐标点低于区域（$y<y_{min}$），第3位的值为1。

以上的图标给出了图中九个区域的区域码。

(a) 假设坐标点（x_1，y_1）的区域码放在AL的低四位，坐标点（x_2，y_2）的区域码放在BL的低四位，寄存器的其他位都为0。给出一条80x86语句，使得当这两个坐标点都在区域内时，设置ZF为1，否则ZF为0，AL或BL中的值可以改变。

(b) 假设坐标点（x_1，y_1）的区域码放在AL的低四位，坐标点（x_2，y_2）的区域码放在BL的低四位，寄存器的其他位都为0。给出一条80x86语句，使得当这两个点都在矩形区域的同一边时，设置ZF为0（“在同一边”意味着，都在X = Xmax的右边，或都在X = Xmin的左边，或都在Y = Ymax上边，或都在Y = Ymin下边）。AL和BL中的值可以改变。

(c) 编写一个NEAR32过程setcode，把区域码还原成坐标点（x，y）。特别是，过程setcode有六字长的整型参数：x、y、x_{max}、x_{min}、y_{max}和y_{min}。这些参数按给定顺序入栈。返回区域码放入寄存器AL中低四位，将0赋给EAX中的高位。

8.2 移位和循环移位指令

上一节介绍的逻辑指令能够用汇编语言设置或清除寄存器或存储器中的字或字节的位。移位和循环移位指令能够改变一个双字、字或者字节内位的位置，本节详细介绍移位和循环移位指令，并给出了一些应用的实例。

移位指令对目的操作数所给定地址内的位进行左移或右移。移位的方向是由指令助记符的最后一位决定的，sal和shl是左移位；sar和shr是右移位。移位指令可分为两类：逻辑移位和算术移位，指令shl和shr是逻辑移位，指令sal和sar是算术移位。下面将解释算术和逻辑移位的区别，图8-2列出了移位指令的助记符。

	左移	右移
逻辑移位	shl	shr
算术移位	sal	sar

图8-2 移位指令

移位指令的格式是：

```
s- 目的地址，count
```

操作数count有三种形式。这个操作数可以是数字1，或是一个字节的立即操作数，还可以是寄存器CL，最初的8086/8088只有第一和第三两种形式，只能是数字1或寄存器CL。

如果指令如下：

```
s- 目的地址，1
```

那么，其作用是使目的地址的内容移动1位。而指令：

```
s- 目的地址，8位立即数
```

可以对0到255之间的一个立即操作数编码。然而，大多数80x86系列用00011111_2与这个操作数作掩码；也就是说，在移位前对它进行模32运算。这是因为对一个不超过一个双字节长的操作数来说，做超过32位的移位操作是没有意义的。最后一种形式的移位指令：

```
s- 目的地址，cl
```

其中寄存器CL中是无符号计数操作数。同样，对于大多数80x86CPU而言，在移位前先把计数操作数转换为32的模。

算术左移和逻辑左移是一样的，助记符sal和shl产生同样的目标代码。当执行左移位时，目的操作数的每一位向左移动，最右边的一位置0。移出左边的位被丢失，除了最后移出的一位，它被保存在进位标志位CF中。根据目的地址中最后的结果，符号标志位SF、零标志位ZF和奇偶校验标志位PF将赋相应的值。在多位移位中，溢出标志位OF未定义；在一位移位（*count*=1）中，如果结果的符号位的值和原操作数符号位的值相同，则OF设置为0，反之，则置1。辅助进位标志位AF未定义。

算术右移和逻辑右移不一样。对这两者而言，目的操作数每一位向右移动，除了最后移出的一位，它被保存在CF中，其他移到右边的位被丢失。对于逻辑右移（shr），最左边的一位置0。而对于算术右移（sar），最左边的一位用原操作数的符号位填充。因此，对算术右移而言，如果原操作数是负的二进制补码数，那么新的操作数中最左边将一直用1填充，并且新的操作数仍为负。和左移一样，算术右移后标志位SF、ZF和PF的值取决于运算的结果，AF未定义，多位移位中，溢出标志位OF未定义。对于一位逻辑右移shr，如果结果的符号位和原操作数的符号位的值相同，OF置0；反之，则置1（注意：这相当于把原操作数的符号位的值赋给OF）。对于一位算术右移sar，OF为0，因为原操作数和新的操作数的符号位通常是相同的。

有些十六进制计算器可以直接进行移位运算。要求用二进制表示操作数，移位或重组位（用适当的0或1填入），再把结果转换为十六进制。移动四位或四的倍数位的多位移位相对简单一些，在这样的情况下，每四位一组对应一个十六进制数，从而把移位变为移位十六进制数。下面举例说明移位指令的执行，每个例子都从一个包含十六进制的A9D7（1010 1011 1010 1112）的字开始。移出的位用一条线和原值分开，新增的位在新值中用粗体表示。

执行前	指令	二进制操作	执行后
CX: A9D7	`sal cx, 1`	1010 1001 1101 0111 0101 0011 1010 111**0**	CX [53 | AE] SF0 ZF 0 CF1 OF 1
AX: A9D7	`shr ax, 1`	1010 1001 1101 011**1** **0**101 0100 1110 1011	AX [54 | EB] SF0 ZF 0 CF1 OF 1
BX: A9 D7	`sar bx, 1`	1010 1001 1101 011**1** **1**101 0100 1110 1011	BX [D4 | EB] SF1 ZF 0 CF1 OF 0
ace: A9 D7	`sal ace, 4`	1010 1001 1101 0111 1001 1101 0111 **0000**	ace [9D | 70] SF1 ZF 0 CF0 OF ?

```
DX: A9 D7      shr dx, 4      1010 1001 1101 0111      DX  0A  9D
                                  ──────►
                              0000 1010 1001 1101      SF0  ZF 0
                                                       CF0  OF ?

AX: A9 D7      sar ax, cl     1010 1001 1101 0111      AX  FA  9D
CL: 04                            ──────►
                              1111 1010 1001 1101      SF1  ZF 0
                                                       CF0  OF ?
```

对于不同的操作数类型，表8-4给出了移位指令所需的时钟周期数和操作数字节数。到目前为止，已经讨论了移位指令的四种类型，和后面讨论的循环移位指令共用操作码，目的操作数的大小和计数操作数的类型隐含在操作码中。正如其他指令，目标代码的第二字节用来选择移位运算或循环移位运算的不同类型，以及运算是否在寄存器和存储器之间进行。值得注意的是，一位移位运算比多位移位运算要快——通常使用几条一位移位指令比使用一条多位移位指令更具有时效性。

表8-4 移位和循环移位指令

目的操作数	计数操作数	时钟周期			字节数	操作码
		386	486	Pentium		
8位寄存器	1	3	3	1	2	D0
16/32位寄存器	1	3	3	1	2	D1
字节存储器	1	7	4	3	2+	D0
字/双字存储器	1	7	4	3	2+	D1
8位寄存器	8位立即数	3	2	1	3	C0
16/32位寄存器	8位立即数	3	2	1	3	C1
字节存储器	8位立即数	7	4	3	3+	C0
字/双字存储器	8位立即数	7	4	3	3+	C1
8位寄存器	CL	3	3	4	2	D2
16/32位寄存器	CL	3	3	4	2	D3
字节存储器	CL	7	4	4	2+	D2
字/双字存储器	CL	7	4	4	2+	D3

移位指令很简单，但是它们应用广泛。应用之一是进行乘法和除法运算。实际上，对于没有乘法指令的处理器，移位指令是做乘法运算的一个关键部分。即使对80x86结构体系而言，有些乘法计算用移位运算比用乘法指令更快。

在一个乘数为2的乘法运算中，被乘数左移一位的结果作为乘积放在初始地址，除非进位标志位OF设置为1，该乘积是正确的。显然，该运算对无符号数有效；每位左移一位，新的数就是原数二进制表示的2的高次幂，一个一位左移等于一个有符号操作数乘2。实际上，在十六进制计算器中，一位左移的结果可用乘2来得到。

一位右移可用于计算一个无符号操作数除2。例如，寄存器EBX内有一个无符号操作数，逻辑右移 shr ebx, 1把EBX内的每一位移到相应的2的下一个低次幂，得到初始值的一半。初始单元的位复制到进位标志位CF，这就是除法的余数。

如果EBX内有一个有符号操作数，那么算术右移指令sar ebx, 1和idiv指令在除数为2时结果几乎是一样的。不同的是，如果被除数是一个负的奇数，商就被取整；就是说，右移得出的值可能比用idiv指令得出的值要小。举一个具体的例子来说，假设寄存器DX内容为FFFF，寄存器AX内容为FFF7，那么DX-AX表示是双字长度的二进制补码表示的－9。同时，假定有CX的内容为0002。那么idiv cx得到结果在AX中为FFFC，在DX中为FFFF；也就是说，商为－4，余数为－1。然而，如果把FFFFFFF7放在EBX中，sar ebx, 1得到结果是EBX内容为FFFFFFFB，CF内容为1，商为－5，余数为＋1。两种结果的商和余数都同时满足以下的等式：

被除数＝商×除数＋余数

但是，当商为－5，余数为＋1时，余数的符号和被除数的符号是相反的，这违反了idiv指令的规则。

一个操作数乘2，可以用自己加上自己的加法运算实现，也可以用左移位运算实现。有时，移位运算的效率比加法高一些。不过，这两种运算的效率都比乘法高得多。一个操作数除2，右移位是替代除法的惟一选择，而且速度更快；但是当被除数是负数时，对于除2运算，右移位和除法有很大的不同。无论是一个操作数乘或者除以4、8以及其他2的幂，都可以用重复执行一位移位指令或用一条多位移位指令来实现。

移位可以和其他逻辑指令一起使用，把不同组的位组合成一个字节或字，或者把一个字节或字内的位分解成几个不同的组。代码段8-1的程序提示输入一个整数，用atod宏把这个整数转换为二进制的补码形式，并存入寄存器EAX，然后用8个十六进制数字显示寄存器EAX中的字。为此，必须从EAX的值中取出八组四位一组的数。每个组四位代表一个0到15的十进制值，每组必须转换为字符显示出来。这些字符是数字0到9表示整数0（0000_2）到9（1001_2），或是字母A到F表示整数10（1010_2）到15（1111_2）。

代码段8-1 用十六进制形式显示一个整数

```
; 显示8个十六进制数的程序
; 作者:   R. Detmer
; 日期:   1997年10月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

include io.h                                ; input/output的头文件

cr      equ     0dh                         ; 回车符
Lf      equ     0ah                         ; 换行符

.STACK  4096                                ; 保留4096字节的堆栈
.DATA                                       ; 保留数据存储空间
prompt      BYTE    "Enter a number:  ",0
number      BYTE    20 DUP (?)
result      BYTE    cr,Lf,"The 2's complement representation is "
hexOut      BYTE    8 DUP (?),cr,Lf,0

.CODE                                       ; 主程序代码的开始
```

```
_start:
            output prompt              ; 输入数字
            input  number,20           ; 读入ASCII字符
            atod   number              ; 转换为数字

            lea    ebx,hexOut+7        ; 最后一个字符的地址
            mov    ecx,8               ; 字符个数
forCount:   mov    edx,eax             ; 复制格式
            and    edx,0000000fh       ; 除最后一个十六进制数外均置0
            cmp    edx,9               ; 是否数字
            jnle   elseLetter          ; 不是则为字母
            or     edx,30h             ; 转换为字符
            jmp    endifDigit
elseLetter: add    edx,'A'-10          ; 转换为字母
endifDigit:
            mov    BYTE PTR [ebx],dl   ; 把字符复制到存储器
            dec    ebx                 ; 指向下一个字符
            shr    eax,4               ; 右移一个十六进制数
            loop   forCount            ; 循环

            output result              ; 输出标号和十六进制值

            INVOKE  ExitProcess, 0     ; 退出并返回代码0
PUBLIC _start                          ; 公开程序入口点
END                                    ; 结束源代码
```

生成的八个字符按从右到左的顺序存入存储器内的相邻字节；寄存器EBX用来指向每个字符的目的地址。程序设计思想是：

```
for count: = 8 递减 1 loop
    将EAX的内容复制到EDX;
    清除EDX中除低四位外的内容;

    if EDX中的值小于等于9;
    then
        将EDX中的值转换为0到9中的一个数字;
    else
        将EDX中的值转换为A到F中的一个字母;
    end if;

    将存储器中字符的地址存入EBX;
    EBX减1, 向左指向下一个位置;
    将EAX中的数值右移4位;
end for;
```

为了实现该设计，指令

```
and   edx, 0000000fh      ; 除最后一位十六进制位外，所有位置0
```

除了最后4位，屏蔽掉EDX中其他所有的位。If语句由以下指令实现：

```
cmp   edx, 9              ; 是否数字?
jnle  elseLetter          ; 不是则为字母
```

```
            or      edx, 30h            ; 转换为字符
            jmp     endifDigit
elseLetter: add     edx, 'A'-10         ; 转换为字母
endifDigit:
```

用or指令把一个0到9之间的数字转换为ASCII码；这里也可以使用add edx, 30h指令。为了把数字0A到0F转换为字母A到F，其对应的ASCII码是41到46，原值要加上‘A’－10，实际上就是加上十进制数的55，但这样写比add edx, 55更清晰。指令shr把寄存器EAX内的值右移四位，移出的十六进制数正好已经被转换为一个字符。

7.4节中的编程练习2要求编写的过程和代码段8-1给出的程序功能相似。那个过程是用除以16的余数来得到对应于最右边十六进制数的值。注意，这个例子程序中使用的shr 和and指令更易于编写代码，并且更有效率。

上面讨论的移位指令是把一个操作数的位移到适当的位置，只是有一位要影响到进位标志位。80x86体系结构还有两个双移位指令：shld 和shrd，其形式如下：

```
sh-d 目的操作数，源操作数，计数
```

其中目的操作数可以是寄存器或存储器中的一个字或者双字，源操作数是寄存器中的一个字或双字，计数是立即数或者是寄存器CL中的数。shld指令和shl指令类似，把目的操作数左移，不同之处在于被移动的位是从源操作数最左边一位开始，源操作数并不改变。shrd指令就像shr指令，把目的操作数右移，不同之处在于被移的位从是源操作数最右边一位开始。对这两种双移位，最后移出的一位移入CF，根据目的地址内的结果，SF、ZF和PF赋相应的值。双移位中，溢出标志位OF未定义。

下面举两个例子来说明双移位指令。shld移位把ECX中的前三位十六进制数（12位）移出，用EAX中的最左边三位十六进制数从右边填补。因为最后一个移出的位是3（0011_2）的最右边的一位为1，所以进位标志位CF为1。本例中shrd把ECX最后两个十六进制数（共8位）移出，用EAX的最右边两个十六进制数从左边填补。由于最后一个移出的位是7（0111_2）的最左边的一位为0，因此标志位CF为0。

执行前	指令	执行后				
ECX: 12 34 56 78	shld ecx, eax, 12	ECX	45	67	89	0A
EAX: 90 AB CD EF		EAX	90	AB	CD	EF
		CF 1	ZF 0	SF 0		
ECX: 12 34 56 78	shrd ecx, eax, cL	ECX	EF	12	34	56
EAX: 90 AB CD EF		EAX	90	AB	CD	EF
CL: 08		CF 0	ZF 0	SF 1		

表8-5列出了多种双移位指令，源操作数没有给出，因为它和目的操作数的字节数相同，

要么是一个16位寄存器，要么是一个32位寄存器。

表8-5 双移位指令

目的操作数	计数操作数	时钟周期			字节数	操作码	
		386	486	Pentium		shld	shrd
16/32位寄存器	8位立即数	3	2	4	4	0F 04	0F AC
字/双字存储器	8位立即数	7	4	4	4+	0F 04	0F AC
16/32位寄存器	CL	3	3	4	3	0F 05	0F AD
字/双字存储器	CL	7	4	5	3+	0F 05	0F AD

如果代码段8-1给出的程序用双移位指令的话，那么该程序的代码会更清晰。下面的程序是从左到右生成十六进制数，而不是从右到左生成。每循环一次，shld把EAX中第一个十六进制数复制到EDX中。

```
            lea   ebx, hexOut          ; 首字符地址
            mov   ecx, 8               ; 字符数
forCount:   shld  edx, eax, 4          ; 得到首十六进制数
            and   edx, 0000000fh       ; 除最后一个十六进制数外均置0
            cmp   edx, 9               ; 是否数字?
            jnle  elseLetter           ; 不是则为字母
            or    edx, 30h             ; 转换为字符
            jmp   endifDigit
elseLetter: add   edx, 'A'-10          ; 转换为字母
endifDigit:
            mov   BYTE PTR [ebx], dl   ; 复制字符到存储器
            inc   ebx                  ; 指向下一个字符
            shl   eax, 4               ; 左移一个十六进制数
            loop  forCount             ; 循环
```

循环移位指令类似于一位移位指令。在移位指令中，位从一端移出，在另一端空出的位置用0填空（当对一个负数进行算术右移时，用1填空）。在循环移位指令中，从目的操作数一端移出的位用来填补到另一端的空位。

循环移位指令的格式和单移位指令相同。一位循环移位指令的格式如下:

```
r- 目的操作数, 1
```

它还有两个多位循环移位形式:

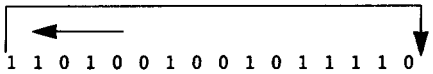
```
r- 目的操作数, 8位立即数
r- 目的操作数, cl
```

rol指令（循环左移位）和ror指令（循环右移位）的操作数可以是寄存器或存储器中的字节、字或者双字。每一位从一端移出的位，被复制到目的操作数的另一端。另外，最后移出的位被复制到另一端，同时也被复制到进位标志位CF，溢出标志位OF也受循环移位指令的影响。但对于多位循环移位，OF未定义，以及大家熟知的一位移位循环指令，本书没有列出。

例如，假设寄存器DX中的数是D25E，执行指令:

```
rol dx, 1
```

用二进制表示的话，其运算如下:



结果是1010 0100 1011 1101或A4BD。由于最左边的一位1循环移位到右边，因此，进位标志位CF置1。

循环移位指令所需的时钟周期数和操作码与移位指令相同，在表8-4已经列出。

代码段8-1的程序也可使用循环移位指令来实现。下面的过程要求十六进制数按照从左到右的顺序循环移位，并且每次移动4位，八次循环移位后，最后EAX中的值不改变，循环移位后，所有的位都转回到原来的位置。

```
            lea   ebx, hexOut          ; 首字符地址
            mov   ecx, 8               ; 字符数
forCount:   rol   eax, 4               ; 把第一个十六进制数循环移位到最后
            mov   edx, eax             ; 复制所有的数
            and   edx, 0000000fh       ; 除最后一个十六进制数外全置0
            cmp   edx, 9               ; 是否数字?
            jnle  elseLetter           ; 否则是字母
            or    edx, 30h             ; 转换为字符
            jmp   endifDigit
elseLetter: add   edx, 'A'-10          ; 转换为字母
endifDigit:
            mov   BYTE PTR [ebx], dl   ; 把字符复制到存储器
            inc   ebx                  ; 指向下一个字符
            loop  forCount             ; 循环
```

此外，还有一对循环移位指令rcl（带进位的循环左移位）和rcr(带进位的循环右移位)。这两条指令都带进位CF执行，把进位CF作为目的操作数的一部分。这就是说，指令rcl eax, 1执行的结果是将EAX中的0到30位左移一位；将第31位的原值复制给CF，CF中的原值复制到EAX的第0位。显然，带进位的循环移位指令影响CF，也影响OF，但不影响其他标志位。带进位的循环移位指令的操作码和相应的移位指令一样，在表8-4中已列出。但是，它们所需的时钟周期数不同，本书没有给出。

练习8.2

1. 下面每小题，假定给出指令执行前的值，求执行后的值。(与练习8-1类似)

执 行 前	指 令	执 行 后
(a) AX:A8 B5	shl ax, 1	AX, CF, OF
(b) AX:A8 B5	shr ax, 1	AX, CF, OF
(c) AX:A8 B5	sar ax, 1	AX, CF, OF
(d) AX:A8 B5	rol ax, 1	AX, CF
(e) AX:A8 B5	ror ax, 1	AX, CF
(f) AX:A8 B5 CL:04	sal ax, cl	AX, CF

(g) AX:A8 B5	shr ax, 1	AX, CF
(h) AX:A8 B5 CL:04	sar ax, cl	AX, CF
(i) AX:A8 B5 CL:04	rol ax, cl	AX, CF
(j) AX:A8 B5	ror ax, 4	AX, CF
(k) AX:A8 B5 CF:1	rcl ax, 1	AX, CF
(l) AX:A8 B5 CF:0	rcr ax, 1	AX, CF
(m) AX:A8 B5 CX:FE 40	shrd ax, cx, 4	AX, CF
(n) AX:A8 B5 CX:FE 40	shld ax, cx, 4	AX, CF

2. 计算寄存器EAX中的无符号整数除以32，使用Pentium处理器的时钟周期，比较的不同程序所用的时钟周期数和目标代码字节数。

```
(a) mov  edx, 0     ; 把值扩展为双字
    mov  ebx, 32    ; 除数
    div  ebx        ; 值除以32
(b) shr  eax, 1     ; 除以2
    shr  eax, 1     ; 除以2
    shr  eax, 1     ; 除以2
    shr  eax, 1     ; 除以2
    shr  eax, 1     ; 除以2
(c) shr  eax, 5     ; 除以32
```

3. 计算寄存器EAX中的值乘以32，使用Pentium处理器的时钟周期，比较的不同程序的所用的时钟周期数和目标代码字节数。

```
(a) mov  ebx, 32    ; 乘数
    mul  ebx,       ; 值 × 32
(b) imul eax, 32    ; 值 × 32
(c) shl  eax, 1     ; 乘以2
    shl  eax, 1     ; 乘以2
    shl  eax, 1     ; 乘以2
    shl  eax, 1     ; 乘以2
    shl  eax, 1     ; 乘以2
(d) shl  eax, 5     ; 乘以32
```

4. 假设value1、value2和value3在存储器中各占一个字节，每个字节存储一个无符号整数。假设第一个值不大于31，这样它最多有五个有效位，开头最少有三个0。同样，假设第二个值不大于15（四个有效位），第三个值不大于127（七个有效位）。

(a) 写出代码，把这三个数压缩为一个16位的字存放在寄存器AX中，把value1的低五位按顺序存入AX的11－15位，value2的低四位按顺序存入AX的7－10位，value3的低七位存入AX的0－6位。

(b) 写出代码，把寄存器AX中的这个16位数分解为五位、四位和七位的三个数，每个数的左边空余位填0，以补足为8位，结果分别存放到value1、value2和value3中。

5. 下面的指令:

```
mov   ebx, eax     ; 值
shl   eax, 1       ; 2×值
add   eax, ebx     ; 3×值
```

是将EAX中的值乘以3。使用移位和加法指令，编写和上述指令序列类似的代码，计算EAX中的值乘以5、7、9和10。

编程练习8.2

1. 编写一个NEAR32过程，把一个32位整数转换为一个32位字符的字符串，用二进制数表示它的值。该过程有两个参数，通过堆栈传递。

(1) 数

(2) 目的字符串的地址

过程可以从堆栈中取出参数，并且不修改寄存器内容。用循环移位指令一次取出一位，从左到右，提示：可用jc或jnc指令查询进位标志位（这道练习和7.4节的编程练习3相似，除了取出位的方法不同）。

2. 一个8位数可以用三个八进制数来表示。第7位和第6位确定左边的一个八进制数字，这个八进制数不能大于4，第5位、第4位和第3位是中间的一个八进制数字，第2位、第1和第0位是右边的一个八进制数字。例如，110101102是11 010 1102或326_8。一个16位数用八进制数表示的方法是用2－3－3模式，分别表示高八位和低八位字节。编写一个NEAR32过程，把一个16位整数拆成由六个字符组成的字符串，该字符串表示八进制数值。这个过程有两个参数，通过堆栈传递:

(1) 数

(2) 目的字符串的地址

该过程序可以从堆栈中取出参数，并且不修改寄存器内容。

8.3 ASCII字符串到二进制补码整数的转换

atoi宏和atod宏用于扫描一个ASCII码表示的整数的内存区，并生成同样字长的二进制补码整数，放在寄存器EAX中。宏调用和过程调用类似，本节以atod为例。

atod宏可扩展为下面的指令行:

```
lea   eax, source        ; 源地址到EAX
push  eax                ; 堆栈中的源参数
call  atodproc           ; 调用atodproc(源)
```

这些指令只使用的一个参数，即扫描ASCII码字符串的地址，调用atodproc过程。这个宏没有保存寄存器EAX的内容，因为宏调用的结果要返回到寄存器EAX中。这个宏扩展使用了实际的源标识符，而不是名字source。

真正的ASCII码到二进制补码整数的转换是由过程atodproc完成的，该过程汇编后放在文件IO.OBJ中。过程atodproc的源代码如代码段8-2所示，它以标准入口代码开始。在注释时没有明确地置1还是置0的标志位要保存起来，这样调用返回时，原值依然保持不变，在AToDExit中的指令popf和pop恢复这些值；但是，atodproc过程会改变由popf指令取出的堆栈中的字，这一点以后讨论。

代码段8-2 ASCII到双字整数的转换

```
; atodproc(源)
; 程序用来扫描从源地址开始的数据段，把ASCII字符转换为整数值返回到EAX.

; 段首的空部分跳过. 可以由"-"或"+"符号开始. 数字必须紧跟在符号后面 (如果有的话).
; 存储器扫描在没有任何数字后结束，结束字符的地址返回ESI.

; 以下的标志位受到影响:
;   AF未定义
;   PF, SF, ZF由返回到EAX的数的符号决定
;   CF置"0"
;   OF为"1"表示出错. 允许出错的条件是:
;       - 没有数字输入
;       - 值超出-2,147,483,648到2,147,483,647的范围
;   如果OF为"1", (EAX)为0

atodproc    PROC    NEAR32
            push    ebp                     ; 保存基指针
            mov     ebp, esp                ; 建立堆栈
            sub     esp, 4                  ; 符号的局部空间
            push    ebx                     ; 保存寄存器内容
            push    ecx
            push    edx
            pushf                           ; 保存标志位

            mov     esi,[ebp+8]             ; 得到参数 (源addr)

WhileBlankD:cmp     BYTE PTR [esi],' '      ; 是否为空?
            jne     EndWhileBlankD          ; 如果不为空退出
            inc     esi                     ; 字符指针加1
            jmp     WhileBlankD             ; 再执行一次
EndWhileBlankD:

            mov     eax,1                   ; 默认符号乘数
IfPlusD:    cmp     BYTE PTR [esi],'+'      ; 负符号-1
            je      SkipSignD               ; 起始是否为+
IfMinusD:   cmp     BYTE PTR [esi],'-'      ; 是, 跳出
            jne     EndIfSignD              ; 起始是否为-
            mov     eax,-1                  ; 不是, 保存默认+
SkipSignD:  inc     esi                     ; 转移符号位
EndIfSignD:
            mov     [ebp-4],eax             ; 保存带符号乘数
            mov     eax,0                   ; 数字被累加?
            mov     cx,0                    ; 计数开始
```

```
WhileDigitD:cmp     BYTE PTR [esi],'0'      ; 下一个字符和“0”比较
            jl      EndWhileDigitD          ; 小于“0”不是数字
            cmp     BYTE PTR [esi],'9'      ; 和“9”比较
            jg      EndWhileDigitD          ; 大于“9”不是数字
            imul    eax,10                  ; 原数字乘以10
            jo      overflowD               ; 如果结果太大，退出
            mov     bl,[esi]                ; ASCII字符移入BL
            and     ebx,0000000Fh           ; 转换为单数字整数
            add     eax,ebx                 ; 加上总数
            jc      overflowD               ; 若总数太大，退出
            inc     cx                      ; 计数加1
            inc     esi                     ; 字符指针加1
            jmp     WhileDigitD             ; 检测下一个字符
EndWhileDigitD:

            cmp     cx,0                    ; 不是数字?
            jz      overflowD               ; 不是，则置溢出标志位为1
; 如果值是80000000h，且符号是“-”，要返回80000000h（-2^32）

            cmp     eax,80000000h           ; 是否80000000h?
            jne     TooBigD?
            cmp     DWORD PTR [ebp-4],-1    ; 是否乘数-1?
            je      ok1D                    ; 若是，返回8000h

TooBigD?:   test    eax,eax                 ; 检查符号标志位
            jns     okD                     ; 若数>2^32-1，符号位为“1”

overflowD:  pop     ax                      ; 得到标志位
            or      ax,0000100001000100B    ; 溢出标志位、零标志位、奇偶标志位均为1
            and     ax,1111111101111110B    ; 设置符号位和进位标志位为1
            push    ax                      ; 新的标志位值入栈
            mov     eax,0                   ; 返回零标志位值
            jmp     AToDExit                ; 退出

okD:        imul    DWORD PTR [ebp-4]       ; 产生有符号数
ok1D:       popf                            ; 弹出原标志位
            test    eax,eax                 ; 设置新数的标志位为1
            pushf                           ; 保存标志位
AToDExit:   popf                            ; 弹出标志位
            pop     edx                     ; 恢复寄存器
            pop     ecx
            pop     ebx
            mov     esp, ebp                ; 删除局部变量
            pop     ebp
            ret     4                       ; 退出，移出参量
atodproc    ENDP
```

如果字符串是以空格起始的话，过程atodproc的第一个任务就是跳过这些空格，这个直接用while循环实现。值得注意的是，BYTE PTR[esi]使用寄存器间接寻址来访问源字符串中的一个字节。在下面的while循环中，ESI指向一个非空字符。

该过程的主要思想是计算整数的值，用下面从左到右扫描的算法来实现：

```
value :=0;
while代码中指向一个数字时loop
```

```
        value乘以10;
        将ASCII码值转换为整型;
        将此整型值与value值相加，结果放入value中;
        指向存储器下一个字节;
    end while;
```

这个算法适用于无符号数；设置一个专门的乘数为最后有符号结果给出正确的符号。这个过程的第二个任务是跳过空格后，保存该乘数，正数为1，负数为－1。乘数是堆栈中的局部变量，默认情况下，它为1；如果第一个非空字符为负号的话，则乘数为－1。若第一个非空字符是正号或者是负号，ESI中的地址加1，跳过该符号位。

现在执行主程序，累加值放在寄存器EAX中。如果乘以10产生溢出的话，说明结果值太大，不能存放在寄存器EAX中。这时，指令jc overflowD转去执行 overflowD代码，由它处理所有的错误。

为了把一个字符转换为数字，该字符要存入寄存器BL。执行指令and ebx, 0000000Fh后，寄存器EBX中除最低四位外，其他所有位都被清零。因此，若ASCII码37_{16}表示的数字7在寄存器EBX中就是00000007。如果这个值和EAX中累加值相加产生进位，这个相加的和超过EAX的范围，指令jc跳转到overflowD。

如果ESI指向任何一个非数字字符，主循环终止。因此，一个整数可以由一个空格、逗号、字母以及其他任何非数字结束。为了确定输入的是否为有效的整数，主循环在寄存器CX中存有一个十进制的计数器。循环终止后，检查计数器。如果它为0，说明没有数字输入，指令jz跳转至overflowD，进行错误处理。如果数字个数太多，就不需要检查，这种情况下，主循环可能因为溢出而终止。

若寄存器AX中的累加值大于80000000_{16}（无符号数2 147 483 648），以至于超出双字长二进制补码形式表示数的范围。若该累加值等于80000000_{16}，那么乘数必须为－1，因为－2 147 483 648可用二进制补码表示（等于80000000_{16}），而＋2 147 483 648超出范围。下一段代码检查EAX中的80000000_{16}，乘数为－1；在这种情况下，程序继续运行。否则，指令test eax, eax查看累加值是否大于80000000_{16}；如果是，符号位为1。

如果有错误发生，开始执行overflowD的指令。最初的标志位出栈，放入寄存器AX中。溢出标志位置为1，表示有错误发生，值00000000返回到EAX；根据这个0值，其他标志位置1或置0。指令：

```
or ax, 0000100001000100b      ; 设置溢出、零和奇偶标志位为1
```

设置第11位（溢出标志位）、第6位（零标志位），第2位（奇偶标志位）为1。由于返回的结果为零，因此，零标志位为1；值00000000_{16}有偶数个1，因此，奇偶标志位为1。指令：

```
and ax, 1111111101111110b     ; 符号位和进位标志位清零
```

第7位清零（符号标志位），因为00000000不是负数；第0位（进位标志位）也被清零。执行指令or和and后得到的值压入栈，并在程序退出前由popf指令出栈，放入标志寄存器中。

如果没有异常情况，指令imul得出无符号数的乘积，乘以乘数（＋1或－1）后，得出正确的有符号的结果。正常情况下，标志位的值通过popf指令，恢复为最初的标志位值；test eax, eax指令对标志位CF和OF清零，并为PF、SF和ZF赋值。然后，新的标志位值用一个pushf指令入栈，在代码结束前，由popf指令恢复。指令test对标志位AF没有定义，因此，在该过程开始的注释部分就提到了AF。

练习8.3

1. 过程atodproc代码用下面的语句检查符号标志位:

```
TooBigD?:  test   eax, eax        ; 检查符号标志位
           jns    okD             ; 如果数>2^32-1, 置1
```

还有一种方法是:

```
TooBigD?:  cmp eax, 80000000h     ; EAX<2 147 483 648
           jb  okD                ; 是的话ok
```

比较test指令和cmp指令所需的时钟周期数和目标代码字节数。

2. 过程atodproc检查转换的数中0的个数，但不适用于位数太多的数。解释为什么没有必要用最小可能的11位数100 000 000 000来写代码。(另一个不限制数字个数的原因是，任何一个以0（可能不止一个0）开始的数也应该是有效的。)

编程练习8.3

写一个NEAR32过程hexToInt，它有一个由堆栈传递的参数——字符串的地址。该过程和atodproc相似，但它是把一个表示的无符号十六进制数的字符串转换为双字长的二进制补码整数，并存入EAX。这个过程能够跳过开头的空格，并把值累加起来，直到遇到的字符不表示十六进制数为止（有效字符是0到9，A到F，以及a到f)。如果没有十六进制数，或结果值太大不能存入寄存器EAX，那么返回0，并设置OF为1，这是惟一可能的错误。如果没有错误发生，OF位清0。任何情况下，根据返回到EAX中的值，置SF、ZF和PF为1，CF清0。

8.4 硬件级——逻辑门

数字计算机包含许多集成电路，这些电路上的很多元件是逻辑门。一个逻辑门执行一个基本的逻辑运算，8.1节介绍过一些逻辑运算，包括：and、or、xor和not运算。每种逻辑门用一个简单的图形表示它的功能，如图8-3所示，左边是输入，右边是输出。

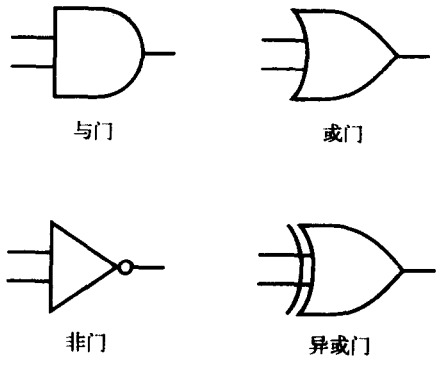
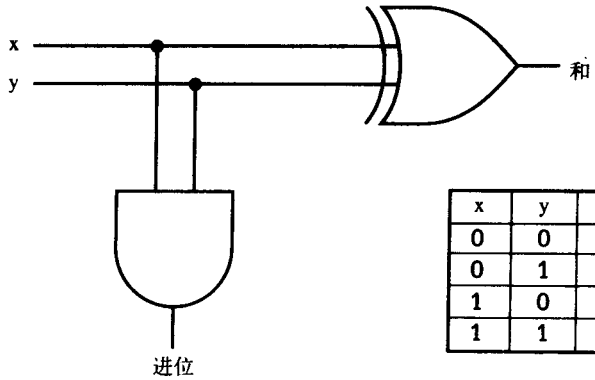


图8-3 逻辑门

通过输入逻辑0或1，输出正确的值，可以做一些简单的电路操作。例如，如果or电路有两个输入，分别为0和1，那么它的输出将为1。逻辑值0和1通常用两个不同的电平来表示。

这些简单的电路组合起来就形成了复杂的电路，可以执行计算机的运算。例如，图8-4是

一个半加电路。该电路输入端x和y的逻辑值可以看作两位相加，想要的结果是：0+0=0，1+0=1，0+1=1，进位标志位为0；1+1=0，进位标志位为1，这些是半加电路得出的正确结果。



x	y	进 位	和
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

图8-4 半加电路

练习8.4

多位数的加法的执行类似于小学里学十进制加法；从最右边一对开始每两位相加，但是除了第一对，其他对还必须加上上一位的进位。要实现多位数加法，需要一系列全加电路。一个全加电路有三个输入端：x、y和进位in，以及两个输出端：和sum和进位out。

1. 画出和图8-4中的表相似的描述一个全加电路的输入和输出的表。这个表要有五列（x、y、进位in、sum、进位out），首行下有八行。
2. 画出一个全加电路。提示：用2个半加电路和一个或门组合成进位输出。
3. 用三个全加和一个半加电路来画出一个电路，该电路能够计算两个四位数的加法。这个电路有八个输入（四对位）和五个输出（四个和位和一个进位标志位）。为了简单起见，可以把每个加法或半加画成一个模块图，不用画出全部的逻辑门。

本章小结

本章探讨了对一个字节、字、或双字操作数中的位进行运算的各种80x86指令。逻辑指令and、or和xor对源操作数和目的操作数的对应的位进行布尔运算。这些指令的功能包括对目的操作数中所选中的位置1或清0。指令not对目的操作数的每一位取反，把每位由0变为1，由1变为0。指令test和指令and相同，但test指令只影响标志位，目的操作数不改变。

移位指令把目的操作数的位左移或右移。移位指令分为一位移位指令和多位移位指令。一位移位指令中，1是第二个操作数；多位移位指令中，用CL或立即数作为第二个操作数，并且指定目的操作数移动的位数。除了算术右移负数填1外，其余所有的一位移位运算的空出位都填0。如果用移位指令来计算乘以或除以2、4、8或2的更高次幂，则是一种更有效、方便的方法。双移位指令用于移位源寄存器内容。

循环移位指令同移位指令相似。但是，循环移位指令中从目的操作数一端移出的位填补到另一端的空位。移位或循环移位指令可以和逻辑指令一起用于取出一个存储地址中的一组

位或把多个值压缩为一个字节或字。

宏atod生成调用*atodproc*过程的代码。该过程跳过开始的空格，扫描存储器内的字符串，注意符号（如果有的话），并累加遇到的数字个数，用ASCII码表示这个双字整数累加值，该程序有几处用到了逻辑指令。

逻辑门是数字计算机电路的最简单的模块，每个门执行一个基本的布尔运算。

第9章 汇编过程

汇编器的工作是将汇编语言源代码转变为目标代码。在较简单的计算机系统中，目标代码就是准备调入内存并且被执行的机器代码。在更复杂的系统中，汇编器生成的目标代码必须在执行前被链接器和（或者）加载器“修正”。本章第一节描述了典型的汇编器的汇编过程，并且给出了关于微软宏汇编器的一些特殊实例。第二节具体介绍了80x86微处理器系列，并详细阐述了它们的机器语言的结构。第三节和第四节分别讨论了宏和条件汇编。大多数汇编器有这样的功能，这些章节描述了MASM如何实现这些功能的。最后一节描述了头文件IO.H中的宏。

9.1 两次扫描汇编和一次扫描汇编

使用汇编语言而不是机器语言的原因之一是，汇编语言允许使用标识符或符号来访问数据段中的数据和代码段中的指令。使用机器语言编码，程序员必须知道数据和指令运行时的地址。汇编器有一个*符号表*，该表将标识符和各种属性相联系。其中一个属性就是一个地址，该地址与一个段的起始地址有关，但有时在运行时也使用绝对地址。另一个属性是符号的类型，符号的类型可能包括数据或指令的标号，符号等同于常量、程序名、宏名和段名。有些汇编器用符号表开始汇编源程序，该符号表包括所使用语言的助记符、所有的寄存器名以及其他保留使用的符号。

汇编器的另一个主要工作是输出目标代码，目标代码很接近于程序运行时所执行的机器语言代码。*两次扫描汇编器*（two-pass assembler）是指第一次扫描源代码产生一个符号表，第二次扫描的时候产生目标代码。*一次扫描汇编器*（one-pass assembler）只扫描源代码一次，但是必须经常在扫描过程中修正所生成的目标代码。以下这个简单的例子可以看出其原因：

```
        jmp endLoop
        add  eax, ecx
endLoop:
```

如果扫描该段，汇编器在jmp指令中找到一个前向变量*endLoop*，这时候，汇编器不能判断出*endLoop*的地址，更不用说判断目的地址是short类型（add指令的地址在2^7字节以内）还是near类型（2^{32}字节以内）。第一种情况使用一个EB操作码和一个字节的位移量。第二种情况选择使用一个E9操作码和一个双字节的位移量。很明显，最后的代码至少要等到汇编器汇编到了标号*endLoop*的源代码行才可确定。

典型的汇编器使用两次性汇编，事实上，有些使用三次或更多次的汇编。微软宏汇编器是一次性汇编器。本书不打算去探讨如何生成目标代码的细节，在汇编列表文件的最后会看到部分MASM的符号表。本节其他的部分将集中关注一个典型的符号表，以第3章中出现的程序和列表文件作为引例。

如果符号是一个数据标号，那么符号表可以包括数据的大小。例如，代码段3-1的程序包

含指示性语句:

```
number2  DWORD ?
```

以及列表文件中的对应行（代码段3-2），即

```
number2  . . . . . . . . . . . . .  Dword   00000004 _DATA
```

上述例子表明*number*2的大小被标识为一个双字节。有了这样的标识长度，MASM能够发现符号的使用错误——通过*number*2的定义，MASM将指出指令的错误:

```
mov  bh, number2
```

由于BH寄存器是单字节大小，而符号表定义的number2却是双字节，因此，类型不匹配而出错。除了长度之外，如果一个符号和多个对象相关，则该符号表应包括与其相关的对象的个数或者与符号相关的字节的总数。MASM的符号列表不显示这些信息。

如果一个符号和值相等同，那么这个值通常存储在符号表中。当汇编器在随后的代码中遇到这个符号时，它将替换保存在符号表中的值。在如下的示例程序中，源代码行:

```
cr   EQU   0dh    ; 传递返回的字符
```

在列表文件行中显示为:

```
cr  . . . . . . . . . . . . . . . .  Number  0000000Dh
```

如果一个符号是一个数据或指令的标号，那么，它的值通常保存在符号表中。汇编器将用一个地址计数器来计算位置值。在一个典型的汇编器中，地址计数器在程序开始或者每个子过程的开始被置为0。微软宏汇编器在每个段的开始处将地址计数器置0。当汇编器扫描源代码时，在语句被汇编之前，每一个数据或指令的地址就是这个地址计数器的值。当前汇编的语句所需要的字节数和地址计数器相加，从而得到下一条语句的地址。再看这一行:

```
number2   DWORD   ?
```

列表文件显示:

```
number2  . . . . . . . . . . . . .  Dword   00000004 _DATA
```

值这一列为00000004，它就是地址计数器在数据段中遇到*number*2时的值。这个值是00000004，因为*number*2之前只有*number*1这一项，并且number1占用了四个字节，所以地址计数器的值为00000004。

当汇编指令的时候，地址计数器使用同样的方法。假设MASM到达代码段9-1中所示的代码段时，地址计数器的值是0000012E，则符号while1的地址将是0000012E。指令cmp需要三个字节的目标代码（9.2节将具体介绍如何确定80x86指令中的目标代码）。因此，当MASM到达指令jnle时，地址计数器的值将是00000131。指令jnle将需要两个字节的目标代码，所以，对于第一个add指令，它占用两个字节的目标代码，其地址计数器的值将增加到00000133。当MASM到达第二个add指令时，地址计数器的值将增加到00000135。add ebx, 2占用三个字节，所以指令inc的地址计数器的值为00000138。由于指令inc占用一个字节，所以，jmp指令的地址计数器的值是00000139。而指令jmp需要两个字节，当汇编器到达符号*endWhile*1时，地址计数器的值为0000013B。因此，*endWhile*1的地址在符号表中是0000013B。

代码段9-1 带前向引用的编码

```
while1: cmp   ecx, 100        ; 循环次数是否<= 100?
        jnle  endWhile1       ; 如果不是，则退出循环
        add   eax, [ebx]      ; 累计和
        add   ebx, 4          ; 地址增加，指向下一个数值
        inc   ecx             ; 循环次数加1
        jmp   while1
endWhile1:
```

符号的地址有多种用途。假设MASM遇到如下语句:

```
mov   eax, number
```

number是在数据部分用DWORD定义的一个标号。由于number的寻址模式是直接寻址，所以汇编器需要知道number在目标代码中的偏移量；这个偏移量恰好就是number在符号表中的地址。

汇编器的主要任务是生成目标代码。但是，一个典型的汇编器也完成其他的任务。任务之一是预留存储区。如下列语句:

```
WORD   20  DUP(?)
```

预留20个字的存储空间。这种保留存储空间的方法通常可以采用如下两种方法中:

- 汇编器可用某个已知的值（如00）向目标文件写入40个字节;
- 汇编器可插入一条命令，最终使得程序被加载到内存中时加载器跳过40个字节。

对于后一种方法，存储区将保留其他程序执行时所留下的任何值。

除了预留存储区，汇编器也能用给定的值来初始化预留的存储区。如MASM语句:

```
WORD   10, 20, 30
```

不仅仅预留了三个字的存储区，而且初始化的第一个字为000A，第二个字和第三个字分别为0014和001E。在MASM和大多数其他的汇编器中，初始值可用多种方式来表达。数字可以用不同的数制系统来表示，通常采用二进制、八进制、十进制和十六进制表示。汇编器把字符值转换为相应的ASCII码或EBCDIC字符编码。汇编器通常允许表达式为初始值。微软宏汇编器通常允许加、减、否、乘、除、非、与、或、异或、移位和相关运算的表达式。这样的表达式在汇编时求值，产生的值实际上就是目标代码中所使用的值。

大多数汇编器能产生一个列表文件，该列表文件可显示最初的源代码以及与目标代码相对应的一些报告。汇编器的另一个任务就是当源代码有错误时产生错误信息。一般的汇编器对于每个错误仅显示行号和错误代码。比较简单的汇编器另用一页生成行号和错误信息。当有错误时，大多数汇编器在错误产生的地方由列表文件包含错误信息。微软宏汇编器可选择在列表文件中包含错误信息，并在控制台显示该信息。

除了列表文件可显示源代码和目标代码外，汇编器也能产生程序中所使用的符号列表文件。这个列表文件包括每个符号的属性信息——属性可从汇编器的符号表中得到，也可由交叉引用得到，交叉引用提示符号所定义的行以及在每行中引用的位置。

有些汇编器用地址计数器来设置一个特殊的实际的内存地址以开始汇编指令，并且生成要加载到这个地址的目标代码。这是一些较简单的系统产生目标代码的惟一方法。通常来说，这样生成的代码不需链接就可以加载并运行。

一个文件能够引用另一个文件中的对象。注意，EXTRN指示性语句有助于MASM实现这个功能。链接器把分散的多个目标代码文件合并成一个单一的文件。如果一个文件引用了另一个文件中的对象，链接器把引用由“待决定”改变为合并文件中的位置。

大多数汇编器产生的目标代码可重定位；也就是说，它能在任何地址被加载。要想这样做的一种方法是，在目标代码文件中设置一个映射，该映射记录程序中每个必须被修改的地址的位置。修改地址通常由加载器完成。加载器最终生成机器代码以备执行。

得到重定位代码的另一种方法是仅用相对的引用来编写代码，因此，每个指令只是引用一个与它自身有一定距离的对象，而不是一个固定的地址。在80x86系统中，大多数跳转指令是相对的，所以，如果一个程序将数据存储在寄存器或栈中，编写这样的程序相当容易。

通过MASM，程序员实际上可以直接使用$符号来引用地址计数器。代码段9-1中的代码段可以改写为:

```
cmp   ecx, 100      ; 循环次数是否< = 100 ?
jnle  $+10          ; 如果不是，则退出循环
add   eax, [ebx]    ; 累计和
add   ebx, 4        ; 地址增加，指向下一个数值
inc   ecx           ; 循环次数加1
jmp   $-11
```

由于地址计数器$的值就是jnle语句被汇编时的首地址，所以该代码段是正确的。要退出循环，必须跳过随后的两个语句的两个字节和八个字节。同样，向后引用也必须跳过inc语句和四个其他的语句以回到cmp语句的开始处，则总共是11个字节。尽管MASM允许使用$来引用地址计数器，但是很明显，这样可能会生成混淆代码，通常应该避免这样做。

练习9.1

1. 说出目标代码和机器语言的不同。
2. 假设在汇编语言程序中的符号引用都是向后引用。一次性扫描的汇编器是否必须“修正”它所产生的代码？请解释原因。
3. 汇编以下的代码片段:

```
Array   DWORD  10 DUP(?)
ArrSize EQU    SIZE Array
```

   ArrSize的值等于多少？对于MASM是否记录某个属性，而该属性跟踪与变量相关的字节数，从中可得出什么结论？
4. 本节介绍了类似WORD指示性语句的存储区预留，它可以通过在目标文件中设置某个已知字节值的恰当的数来实现，或者插入一个命令，该命令最终导致加载器跳过某个恰当的字节数。阐述以上每种方法中的一个优点与一个缺点。

9.2 80x86指令编码

本节将描述80x86机器语言的结构。从这些描述的信息中，人们几乎可以自己手工来汇编一个80x86汇编语言程序。但是，现在的主要目的是想要更好地了解80x86微处理器系列的性能及其局限性。

一条80x86指令由若干域组成，表9-1对此进行了总结。有些指令仅仅有一个操作码，而其他一些指令需要包括域。任何包括的域总是按顺序出现。下面将讨论每个域。

表9-1 80x86的指令域

域	字节数	含义
指令前缀	0/1	$F3_{16}$表示rep、repe或者repz $F2_{16}$表示REPNE或REPNZ $F0_{16}$表示LOCK
地址大小	0/1	如果67_{16}存在，则说明位移量是16位地址，而不是32位地址
操作数大小	0/1	如果66_{16}存在，说明存储器操作数在32位模式下是16位，在16位模式下是32位
段覆盖	0/1	说明操作数不在默认段中
操作码	1/2	操作编码
寄存器-寄存器/存储器模式	0/1	表明是寄存器或者存储器操作数，使用寄存器编码
比例变址基址字节	0/1	其他变址和寄存器信息
位移量	0-4	位移地址值
立即数	0-4	立即数值

第7章已经介绍过串指令的重复前缀，从中学习了在一个基本的字符串指令上加一个重复前缀，可有效地将其变为一个能够自动重复某个基本操作的新指令。重复前缀编码在指令的前缀字节中，基本字符串指令的操作码在操作码字节中。重复前缀字节只能与基本字符串指令一起编码。

LOCK前缀在本书的代码里将不举例说明。LOCK前缀可与一些有选择的指令一起使用，并且在指令执行时会锁定系统总线。锁定总线确保80x86处理器能独占共享的内存。

本书中所有的代码都使用32位存储地址。在32位地址环境下，指令可以只包含16位地址。

当一个地址大小域编码为67_{16}时，位移域中使用的位移是双字节而非四字节。在本书中，该前缀字节不会出现在由汇编语言代码生成的机器代码中。

另一方面，操作数大小域字节通常由本书中的汇编语言代码生成。80x86的CPU有一个状态位决定操作数是16位还是32位。通过使用汇编和链接选项，设置该状态位为1，则说明是32位的操作数。每次对一个字长的操作数编码时，产生的指令包括66_{16}前缀字节以表示是16位的操作数。本书中没有使用到的其他的汇编和链接选项，默认的操作数长度为16位；在这种情况下，66_{16}前缀字节表示一个32位操作数。

字节长的操作数是如何表示的呢？不同的操作码可表示字节长的操作数。为什么16位和32位的操作数不使用截然不同的操作码呢？这样的设计是Intel做出的。最初的8086处理器有16位寄存器，并且用不同的操作码表示8位和16位的操作数长度；而没有指令使用32位的操作数。当80386采用32位寄存器时，可选择“共享”的操作码以表示16位和32位的操作数长度，没有引入更多的新的操作码。

寄存器-寄存器/存储器模式（mod-reg-r/m）字节对不同的指令有不同的使用。目前，它通常有三个域：两位的mod域（“mode”），三位的reg域（用于寄存器，但是有时也用于其他用途），还有一个三位的r/m域（用于寄存器/存储器）。后面的内容将考察mod-reg-r/m字节。

操作码域可完全识别许多指令，但是，有些需要额外的信息——例如，要确定操作数的类

型或者确定其操作本身，得预先看后面的位置。例如，每条add、or、adc、sbb、and、sub、xor和cmp指令，它们都有一个在寄存器或者存储器中的字节长的操作数和一个立即操作数，每条指令使用的操作码为80。选择这个指令中的哪一个是由mod-reg-r/m字节的reg域决定的。对于操作码为80这个特例，reg域是000表示add，001表示or，010表示adc，011表示sbb，and是100，101是sub，xor是110，cmp是111。

操作码80是寄存器-寄存器/存储器模式字节中的reg域所对应的十二个操作码中的一个，这些操作码实际决定了相应的指令。其他的操作码是81、82、83、D0、D1、D2、D3、D6、F6、F7、FE和FF。表9-2给出了最常用指令的reg域的信息。

表9-2 指定操作码的reg域

	寄存器域								
		000	001	010	011	100	101	110	111
操作码	80, 81, 82, 83	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
	D0, D1, D2, D3	ROL	ROR	RCL	RCR	SHL	SHR		SAR
	F6, F7	TEST		NOT	NEG	MUL	IMUL	DIV	IDIV
	FE, FF	INC	DEC					PUSH	

每一对双操作数，如果都不是立即数，80x86要求至少有一个是寄存器操作数。reg域包含了该寄存器的编码。表9-3给出了八个可能的寄存器的编码是如何分配的。根据操作数的大小和指令，reg代码的含义不同，因此，同样的编码可用于ECX和CL。无论何时寄存器的信息都可以编写在指令代码中，而不管该信息是在reg域还是在其他地方。

表9-3 80x86寄存器代码

寄存器代码	32位寄存器	16位寄存器	8位寄存器	段寄存器
000	EAX	AX	AL	ES
001	ECX	CX	CL	CS
010	EDX	DX	DL	SS
011	EBX	BX	BL	DS
100	ESP	SP	AH	FS
101	EBP	BP	CH	GS
110	ESI	SI	DH	
111	EDI	DI	BH	

mod域也用来确定一条指令所拥有的操作数的类型。通常，同一个操作码用于一条指令，该指令有两个寄存器操作数，或者有一个寄存器操作数和一个存储器操作数。选择mod = 11，则表示该指令是一个寄存器到寄存器的操作，或者是一个立即数到寄存器的操作。对于寄存器到寄存器的操作，目的寄存器在reg域中编码，而源寄存器在r/m域中编码。使用这两个寄存器的编码如表9-3所示。对于立即数到寄存器的操作，操作的编码如表9-2所示，目的寄存器在r/m域中编码。如果mod为其他的值，则编码的情况更复杂，它既取决于mod域又取决于r/m域。如果r/m＝100，它还依赖于变址基址字节（SIB）。

SIB字节由三个域组成，一个两位的比例域，一个三位的变址寄存器域和一个三位的基址寄存器域。

比例值是00代表比例因子1，01代表比例因子2，
10代表比例因子4，11代表比例因子8。

变址寄存器和基址寄存器的编码如表9-3所示。因为ESP不是变址寄存器，所以除了100不能在变址寄存器域显示外，其他的都可以显示。表9-4给出了不同的编码。这种格式下的mod域可判断位移量中有多少字节。值00意味着机器代码中没有位移量，除了r/m＝101，这时有且只有一个位移量，但这种特殊的情况经常用于直接存储器寻址。mod值为01意味着机器码中有一个字节的位移量，该字节被看作为一个有符号数，并且，在当其与基址寄存器和/或变址寄存器的值相加之前，这个字节将被展开为双字。mod值为10的含义是在机器码中有一个双字的位移量，该双字与基址寄存器和/或比例变址寄存器的值相加。比例因子可为变址寄存器中值的整数倍。

表9-4 80x86的指令编码

mod域	r/m域	基于SIB	操作数（根据SIB的比例和变址）
00	000		DS:[EAX]
	001		DS:[ECX]
	010		DS:[EDX]
	011		DS:[EBX]
	100	000	DS:[EAX + (比例*变址)]
	(使用*SIB*)	001	DS:[ECX + (比例*变址)]
		010	DS:[EDX + (比例*变址)]
		011	DS:[EBX + (比例*变址)]
		100	SS:[ESP + (比例*变址)]
		101	DS:[*displacement*32 + (比例*变址)]
		110	DS:[ESI + (比例*变址)]
		111	DS:[EDI + (比例*变址)]
	101		DS:32位位移量
	110		DS:[ESI]
	111		DS:[EDI]
01	000		DS:[EAX + 8位位移量]
	001		DS:[ECX + 8位位移量]
	010		DS:[EDX + 8位位移量]
	011		DS:[EBX + 8位位移量]
	100	000	DS:[EAX + (比例*变址) + 8位位移量]
	(使用*SIB*)	001	DS:[ECX + (比例*变址) + 8位位移量]
		010	DS:[EDX + (比例*变址) + 8位位移量]
		011	DS:[EBX + (比例*变址) + 8位位移量]
		100	SS:[ESP + (比例*变址) + 8位位移量]
		101	SS:[EBP + (比例*变址) + 8位位移量]
		110	DS:[ESI + (比例*变址) + 8位位移量]
		111	DS:[EDI + (比例*变址) + 8位位移量]
	101		SS:[EBP + 8位位移量]
	110		DS:[ESI + 8位位移量]
	111		DS:[EDI + 8位位移量]

（续）

mod域	r/m域	基于SIB	操作数（根据SIB的比例和变址）
10	000		DS:[EAX + 32位位移量]
	001		DS:[ECX + 32位位移量]
	010		DS:[EDX + 32位位移量]
	011		DS:[EBX + 32位位移量]
	100	000	DS:[EAX + (比例*变址) + 32位位移量]
	（使用SIB）	001	DS:[ECX + (比例*变址) + 32位位移量]
		010	DS:[EDX + (比例*变址) + 32位位移量]
		011	DS:[EBX + (比例*变址) + 32位位移量]
		100	SS:[ESP + (比例*变址) + 32位位移量]
		101	SS:[EBP + (比例*变址) + 32位位移量]
		110	DS:[ESI + (比例*变址) + 32位位移量]
		111	DS:[EDI + (比例*变址) + 32位位移量]
	101		SS:[EBP + 32位位移量]
	110		DS:[ESI + 32位位移量]
	111		DS:[EDI + 32位位移量]
mod域	寄 存 器	r/m域	操 作 数
11	目的	源	源寄存器，目的寄存器
	操作数	目的	目的寄存器，立即数操作数

下面举例。如下的第一个例子是本书经常用到的一类指令。

```
add ecx, value
```

假设在执行的时候，value引用存储器中地址为1B27D48C的双字的值。从表4-5或附录D中可知，add指令的操作码是03。直接地址仅由32位的位移量组成——没有使用变址寄存器或基址寄存器。因此，mod-reg-r/m字节的组成是mod＝00，reg＝001（用于ECX）和r/m＝101（用于直接寻址），在重新组合并且将其转变为对应的十六进制后，值为00 001 101或0D。指令的最后部分是位移量，所以，整个指令的编码为03 0D 1B27D48C（地址的字节实际上被反向存储）。

考虑指令：

```
add ecx, eax
```

该指令的操作码同样为03。因为有两个寄存器操作数，所以mod域为11。reg域指定了目的寄存器为ECX，reg域值为 001。r/m域给出了源寄存器是EAX，r/m域值为000。因此，该指令的mod-reg-r/m字节为11 001 000，或者用十六进制表示为C8，则该指令的机器码为03 C8。

考虑下一条指令：

```
mov edx, [ebx]
```

表4-3或附录D给出mov指令的操作码为8B。由于操作数[ebx]是间接寻址，没有使用位移量，所以mod域是00。reg域包含EDX的代码010。mod = 00组的第四行显示地址为DS:[EBX]，也就是说，数据段的间接寻址使用EBX中的地址。因此，r/m域是011。把这些域放在一起，得出mod-reg-r/m的字节是00 010 011或13，并且整个指令汇编成8B 13。

再看指令:

```
xor ecx, [edx + 2]
```

表8-1或附录D给出xor指令的操作码是33。内存操作数使用间接寻址并且使用的位移量为2，该位移足够的小，但编码仍为单字节02。因此，mod域是01。reg域包含的ECX为001。表9-4给出的r/m域为010。把这些合在一起则得出一个mod-reg-r/m的字节为01 001 010 或4A，所以，这个指令的机器码为33 4A 02。

考虑下一条使用比例的指令:

```
add eax, [ebx + 4*ecx]
```

这种类型的指令对于处理数组非常有用，如同高级语言中处理数组一样。可以把数组的首地址保存在EBX中，并且数组的索引也保存在ECX中（假设索引从0开始）。索引值和比例因子4（双字的大小）相乘，乘积与基地址相加得到数组元素的地址。表4-5给出的操作码是03。对于无位移量的情况，mod-reg-r/m字节是00 000 100或04，使用了目的寄存器EAX和SIB字节。由于指令既包括基址寄存器也包括变址寄存器，所以需要SIB字节。SIB字节域的比例因子是4，所以比例值为10，index为001代表ECX，base为011代表EBX，所以得到SIB字节为10 001 011或8B。因此，目标代码是03 04 8B。

接下来看:

```
sub ecx, value [ebx + 2*edi]
```

在这条指令中，value引用了数据段中的地址。这条减法指令的操作码是2B。地址可看作是32位的位移量，并且有一个基址寄存器和一个变址寄存器。因此，mod = 10，reg = 001（ECX），并且r/m = 100（需要SIB）。SIB字节的域是01（比例因子为2），111（变址寄存器EDI）和011（基址寄存器EBX）。双字位移量将包含运行时的value的地址。因此，机器码是2B 8C 7B xxxxxxxx，其中x表示value的地址。

如果上述最后一个例子中的第二个操作数改变为value[EBX + 2*EDI + 10]，那么，位移量/地址（以上xxxxxxxx所表示）的值只是简单的增加10。也就是说，汇编器把位移量10和value相对应的位移量结合在一起了。

注意，表9-4中的第一组没有显示如何编码操作数[ebp]，该操作数编码为[ebp + 0]，使用了一个字节长度的位移量00。例如，

```
mov eax, [ebp]
```

编码为8B 45 00，操作码为8B，mod-reg-r/m字节为01 000 101（1字节的位移地址，目的地址EAX，基址寄存器EBP）和位移00。

表9-4还可指出使用ESP和EBP的间接寻址是在堆栈段中，而不是在数据段中。很少会跨段使用。但是，如果想引用附加段中的数据，那么可以如下编码一条指令:

```
cmp  ax,  WORD  PTR  es : [edx  +  2*esi  +  512]
```

选用的这个例子几乎包含了一条80x86指令寻址的所有可能的组成。由于使用的是字长操作数，所以该例使用了操作数长度前缀WORD PTR，使用了一个跨段前缀ES。该指令使用基址寄存

器和变址寄存器以及一个32位的位移量。生成的代码是66 3B 84 72 00000200，操作数长度前缀为66，段越位为26（对于ES），操作码3B，mod-reg-r/m字节为84，SIB为72和位移00000200。表9-5给出了可能的跨段越位字节。

表9-5 跨段前缀

前 缀	段	前 缀	段
2E	CS	36	SS
3E	DS	64	FS
26	ES	65	GS

操作码的分配看起来是完全随机的，但事实上其分配有几种方式。例如，假定value引用一个双字操作数，存储器到寄存器指令mov eax，value的操作码是A1，寄存器到存储器指令mov value，eax的操作码是A3，在二进制中，其区别仅仅在位1，即倒数第二位。位1经常用作*方向标志位*，当第一个操作数在存储器中时，该位值为1，当第一个操作数在寄存器中时其值为0。

同样，对于双字操作数和字节操作数的相应指令，通常，其操作码的不同仅在最后一位，即第0位。例如，假定bVal引用一个字节操作数和dVal引用一个双字操作数，那么，cmp bVal, dl 的操作码是38，cmp dVal, edx的操作码是39。第0位经常用作*类型位*，值为1表示双字（或者字）操作数，值为0表示字节操作数。

某些单字节的指令会有另一种情况，同一条指令可适用于每个寄存器——操作数以恰当的寄存器编码结束。例如，对于32位寄存器操作数（表4-6）的inc指令，其操作码从40到47，最后的三位是000到111，对应寄存器的寄存器编码是递增的。也可用其他方法来考虑这种情况，这类inc指令的操作码可由40和寄存器编码相加而获得。

练习9.2

1. 为什么80x86没有能够指定两个存储器操作数的汇编语言指令？
2. 假设有以下的语句，给出每条指令的机器码。

```
dbl  DWORD  ?   ; run-time location 1122AABB
wrd  WORD   ?   ; run-time location 3344CCDD
byt  BYTE   ?   ; run-time location 5566EEFF
```

(a) add dbl, ecx
(b) add wrd, cx
(c) add byt, cl
(d) add edx, ebx
(e) add dx, bx
(f) add dl, bh
(g) push ebp
(h) cmp ecx, dbl
(i) cmp al, byt
(j) inc ecx
(k) inc cx
(l) pop eax

(m) push dbl
(n) or al, 35
(o) sub dbl, 2 (byte-size immediate operand)
(p) and ebx, 0ff000000h (doubleword-size immediate operand)
(q) xchg ebx, ecx
(r) xchg eax, ecx (note accumulator operand)
(s) cwd
(t) shl edx, 1
(u) neg WORD PTR [EBX]
(v) imul ch
(w) div dbl
(x) dec DWORD PTR [ebx + esi]
(y) and ecx, [ebx + 4*edi]
(z) sub ebx, dbl [4*eax]

编程练习9.2

1. 假设数组arr[0..nbr]包含一个递增顺序的双字集合。以下的过程描述了二分查找关键值keyValue，如果在矩阵中找到关键值keyValue，则返回keyValue的索引；如果不在矩阵中，则返回－1。

```
procedure binarySearch(arr:array, nbr:integer, keyValue:integer):integer
topIndex: = nbr;
bottomIndex: = 0;
while(bottomIndex<topIndex)loop
  midIndex: = (bottomIndex + topIndex) div 2;
  if(keyValue = arr[midIndex])
  then
        return midIndex;
  elseif(keyValue<arr[midIndex])
  then
        topIndex: = midIndex-1;
  else
        bottomIndex: = midIndex + 1;
  end if;
end loop;
return -1;
```

用带有三个参数的80x86 NEAR32过程*binarySearch*来实现这个设计，参数1——双字数组的地址，参数2——双字数nbr，参数3——双字keyValue。将正确的结果返回到EAX。该程序除了EAX外不改变其他寄存器的值，并且通过栈取出参数。使用比例变址寻址方式来寻址数组元素，写一个简短的测试程序来测试过程binarySearch。

2. 初始的nbrElts的值在数组a[1..maxIndex] 中，运用选择排序算法（selection sort）实现将其递增排序。

```
procedure selectionSort(arr:array, nbr:integer)
for position: = 1 to nbrElts-1 loop
   smallSpot: = position;
```

```
    smallValue: = a[position];
    for i: = position + 1 to nbrElts loop
          if a[i]<smallValue
          then
                smallSpot: = i;
                smallValue: = a[i];
          end if;
    end for;
    a[smallSpot]: = a[position];
    a[position]: = smallValue;
end for;
```

用一个带有两个参数的NEAR32过程selectionsort来实现这个算法，参数1——双字整型数数组的地址，参数2——双字数nbrElts。该程序不改变寄存器的值，并且它将通过栈来访问参数。比例变址寻址方式来寻址数组元素。注意，编写这个算法的起始变址是1而不是0。写一个简短的测试程序来测试该过程。

3. 快速排序（quick sort）算法将一个矩阵a[leftEnd..rightEnd]分片以实现递增排序。该算法的思想是通过在矩阵和待移动的矩阵元素中确定一个中间值，以使中间值左边的所有元素比该中间值小，而中间值右边的所有元素比该中间值大。然后，程序对（以中间值为参照）所划分的左右两部分进行递归调用来排序。当待排序的划分只有一个或较少的元素时，递归中止。以下是其设计。

```
procedure quickSort(a:array, leftEnd:integer, rightEnd:integer)
if leftEnd<rightEnd
then
  left: = leftEnd;
  right: = rightEnd;

  while left<right loop
        while(left<right)and (a[left]≤a[right]) loop
              add 1 to left;
        end while;
        swap a[left] and a [right];

        while(left<right) and (a[left]≤a[right]) loop
              subtract 1 from right;
        end while;
        swap a[left] and a[right];
  end while;

  quickSort(a, leftEnd, left-1);
  quickSort(a, right + 1, rightEnd);
end if;
```

用带有三个参数的NEAR32过程quicksort来实现这个算法：（1）矩阵地址是双字整型；（2）双字leftEnd；（3）双字rightEnd。程序将不改变寄存器的值，并且负责它清除堆栈中的参数。使用比例变址寻址方式来正确地寻址矩阵元素。写一个简短的测试驱动程序来测试所编写的过程。

9.3 宏定义及其展开

在第3章，宏可定义为一个语句行，该语句行是其他语句系列的简写。汇编器可将一个宏展开为它所表示的语句，然后汇编这些新的语句。前面许多章节已经广泛使用了IO.H文件中定义的宏。本节将介绍如何写一个宏定义，并且阐述MASM如何使用这些定义将宏展开为其他语句。

宏定义类似高级语言中的过程定义。第一行给出了定义的宏名字和一系列参数；宏定义的主要部分由一个语句的集合组成，这些语句根据参数来描述宏的作用。宏调用也非常类似高级语言的过程，调用时，宏的名字后跟着一系列的参数。

宏与高级语言中过程只是表面上相似。高级语言中的过程调用通常被编译成一系列的指令，并将参数压入堆栈，过程后面紧跟一个call指令；然而，宏调用实际上是展开宏定义中的语句，用实参来代替宏定义中的参数。每次调用宏时，宏中的代码都重复展开，但对于过程仅仅只有代码的一个拷贝。通常，宏的执行要比过程调用更快，因为它没有传递参数，或者call和ret指令而带来的开销，但是，它却常常需要更多字节的目标代码。

每一个宏定义由指示性语句MACRO和ENDM组成。其格式为:

```
name    MACRO    list of parameters (参数列表)
        assembly language statements (汇编语言语句行)
        ENDM
```

MACRO中的参数是普通的符号，用逗号隔开。汇编语言语句可以像使用寄存器、立即操作数或宏以外定义的符号那样使用参数。汇编语言语句行甚至可以包括宏调用。

只要满足宏定义在第一个调用该宏的语句之前，则宏定义可以在汇编语言源代码文件中的任何地方出现。一个很好的编程习惯是将宏定义放在靠近源文件开始的地方。

本节剩下的内容将给出宏定义和宏调用的一些例子。假设一个程序设计需要几次暂停，提示用户按回车键。则不用每次重复这样的代码或者使用过程，而可采用定义一个宏pause来实现。代码段9-2给出了宏pause的定义。

代码段9-2 pause宏定义

```
pause   MACRO
; 提示用户，等待按下回车键
        ouput pressMsg       ; "Press [Enter]"
        input stringIn, 5    ; 输入
        ENDM
```

由于宏pause没有参数，所以，调用宏pause就是将其展开为宏定义中的相同语句。如果语句:

```
pause
```

包括在随后的源代码中，那么汇编器将把这个宏调用展开为语句:

```
output pressMsg      ; "Press [Enter]"
input stringIn, 5    ; 输入
```

当然，每一个语句本身就是一个宏调用，并且将其展开为其他的语句行。注意，宏pause没有包含自身；它引用了数据段中的两个域:

```
pressMsg BYTE "Press [Enter] to continue", 0
stringIn BYTE 5 DUP(?)
```

注意到，宏pause的定义和展开都没有包含ret语句。尽管宏看起来很像过程，但是在汇编时，宏调用被展开并产生嵌入的代码。

代码段9-3给出了一个宏add2的定义。该宏定义用于得到两个参数相加的和，并将结果放入AX寄存器中，宏的两个参数是nbr1和nbr2。参数标号对于宏定义来说都是局部的。同样的标号名也可在程序中用于其他的用途，当然，这会造成一些混淆。

代码段9-3 求两个整数相加和的宏

```
add2        MACRO    nbr1, nbr2
; 将两个双字参数相加的和放入EAX中
            mov      eax, nbr1
            add      eax, nbr2
            ENDM
```

宏add2所展开的语句依赖于调用时所使用的实参。例如，宏调用:

```
add2  value, 30              ; ualue + 30
```

展开为:

```
; 将两个双字参数相加的和放入EAX中
mov     eax, value
add     eax, 30
```

语句:

```
add2     value1, value2      ; value1 + value2
```

展开为:

```
; 将两个双字参数相加的和放入EAX中
mov     eax, value1
add     eax, value2
```

宏调用:

```
add2     eax, ebx            ; 两个值的和
```

展开为:

```
; 将两个双字参数相加的和放入EAX中
mov     eax, eax
add     eax, ebx
```

指令mov eax, eax即使它什么都没实现，但它是合法的。

在上述的每个例子中，第一个实参替代了第一个参数nbr1，第二个实参替代了第二个参数nbr2。每个宏调用展开为两个mov指令，但由于实参的类型不同，因而目标代码也将不同。

如果缺少了某一个参数，宏仍将被展开。例如，语句:

```
add2   value
```

展开为:

```
; 将两个双字参数相加的和放入EAX中
mov     eax, value
add     eax,
```

实参value代替了nbr1，并且用一个空的字符串代替了nbr2。汇编器将报告出错，该错误是由宏展开所产生的非法add指令引起的，而不是因为缺少了变量。

同样地，宏调用:

```
add, value
```

展开为:

```
; 将两个双字参数相加的和放入EAX中
mov     eax,
add     eax, value
```

宏调用中的逗号将缺少的第一个实参和第二个实参value隔开。一个空的实参代替了参数nbr1。汇编器将再次报告出错，这次出错是由于非法的mov指令产生的。

代码段9-4给出了宏swap的定义，该宏用于交换存储器中两个双字的内容。它非常类似80x86的xchg指令，但xchg指令不能使用两个存储器操作数。

代码段9-4 交换存储器字的宏

```
swap      MACRO  dword1, dword2
; 交换存储器中的两个双字
          push   eax
          mov    eax, dword1
          xchg   eax, dword2
          mov    dword1, eax
          pop    eax
          ENDM
```

如同宏add2，调用宏swap所产生的代码取决于所使用的实参。例如，调用

```
swap   [ebx], [ebx + 4]   ; 交换矩阵中相邻的字元素
```

展开为:

```
; 交换存储器中的两个双字
        push  eax
        mov   eax, [ebx]
        xchg  eax, [ebx + 4]
        mov   [ebx], eax
        pop   eax
```

swap宏使用了EAX寄存器，这一点对用户来说不是很明显，因而，宏中的push和pop指令保护EAX寄存器的内容，避免用户意外地丢失EAX的值。

代码段9-5给出了宏min2的定义，该宏用来找出两个双字有符号整数中较小的一个，并把较小的整数放入EAX寄存器中。该宏的代码必须使用一个if语句来实现，并且至少需要一条有标号的汇编语言语句。如果使用一个常规标号，则该标号在每次min2宏调用被展开时出现，这样汇编器将因为重复的标号而报告出错。解决的办法是使用一个LOCAL指示性语句来定义

符号endIfMin，该符号是宏min2的局部符号。

代码段9-5 找出存储器中两个字中较小一个的宏

```
min2    MACRO   first, second
        LOCAL   endIfMin
; 将两个双字中的较小一个放入EAX寄存器中
        mov     eax, first
        cmp     eax, second
        jle     endIfMin
        mov     eax, second
endIfMin:
        ENDM
```

LOCAL指示性语句只能在宏定义内部使用，并且必须是MACRO指示性语句后的第一条语句。（甚至注释也不能分开MACRO和LOCAL。）LOCAL列出了用逗号隔开的一个或多个在宏定义中使用的符号。每次宏被展开并且需要某一个符号时，它就以两个问号开始和四个十六进制数字结束的符号（如??0000，??0001，等等）来代替该符号。在宏调用的某个特殊展开中，相同的符号??dddd在局部符号被使用的地方来取代该局部符号。相同的符号可以在不同宏定义的LOCAL中列表出来，或者也可在宏定义外部的代码中作为常规的符号使用。

宏调用:

```
min2    [ebx], ecx    ; 找出两个值中较小的一个
```

可能展开为下面的代码:

```
LOCAL  endIfMin
; 将两个双字值中的较小一个放入EAX寄存器中
        mov    eax, [ebx]
        cmp    eax, ecx
        jle    ??000C
        mov    eax, ecx
??000C:
```

在这里，endIfMin在宏定义中两次出现的地方在宏展开中被??000C取代。同一个宏的另一个展开可在问号后使用不同的十六进制数字。

MASM汇编器有一些指示性语句来控制宏和其他语句在.LST文件中如何显示。最常用的有:

- .LIST 将使语句行包含在列表文件中;
- .NOLIST 将完全禁止所有语句包含在列表文件中;
- .NOLISTMACRO 当允许原始语句行包含在列表文件中时，可有选择性地禁止宏展开。

IO.H文件以一个.NOLIST指示性指令开始的语句结束，因此，宏定义不会使列表文件混乱。同样地，IO.H以.NOLISTMACRO和.LIST指示性语句结束，因此，宏展开列表文件不会模糊程序员的代码，但是，原始语句行将被列出。

练习9.3

1. 运用代码段9-3给出的add2宏定义，写出以下每个宏调用展开时所对应的语句序列。

(a) `add2 25, ebx`

(b) `add2 ecx, edx`

(c) `add2` ; 没有参数

(d) `add2 value1, value2, value3`

(提示：由于没有匹配的参数，所以第三个变量被忽略。)

2. 运用代码段9-4给出的swap宏定义，写出以下每个宏调用展开时所对应的语句序列。

(a) `swap value1, value2`

(b) `swap temp, [ebx]`

(c) `swap value`

3. 运用代码段9-5给出的min2宏定义，写出以下每个宏调用展开时所对应的语句序列。

(a) `min2 value1, value2`

(假设本地的符号计数器是在000A)

(b) `min2 cx, value`

(假设本地的符号计数器是在0019)

编程练习9.3

1. 写一个add3宏定义，该宏有三个双字整型数的参数，并将三个数的相加和放入寄存器EAX。
2. 写一个max2宏定义，该宏有两个双字整型数的参数，并将两个数的最大值放入寄存器EAX。
3. 写一个min3宏定义，该宏有三个双字整型数的参数，并将三个数中最小的放入寄存器EAX。
4. 写一个toUpper宏定义，该宏有一个参数，参数为存储器中一个字节的地址。该宏代码将检查这个字节，如果其ASCII码是小写字母，则用对应的大写字母的ASCII码来代替。

9.4 条件汇编

微软宏汇编器能查看各种条件，在汇编时检查这些条件，并根据这些条件来汇编源代码。例如，根据一个常量的定义，可能汇编或者跳过一段代码段。这种条件汇编在宏定义中特别有用。例如，根据给出的操作数的个数，两个使用相同助记符的宏可以展开成不同的语句行序列。本节描述了使用条件汇编的一些方法。

代码段9-6给出了宏addAll的定义，该宏实现一到五个双字的整数相加，并将相加的和放入EAX寄存器。它使用了条件汇编IFNB（“如果不为空”），尽管IFNB在开放代码（宏以外的常规代码）中是合法的，但该指示性语句通常用于宏定义中。当addALL宏调用被展开并且遇到某一个IFNB时，MASM检查宏参数的值，参数名包含在“<”和“>”之间。如果该参数有一个相应的实参传递给它，那么它就是“非空”，则宏展开中包括用于该实参的add指令。如果一个参数没有对应的实参，那么，add指令将不被汇编。

代码段9-6 使用条件汇编的addAll宏

```
addAll MACRO  nbr1, nbr2, nbr3, nbr4, nbr5
; 将5个双字整数相加，和放入EAX
        mov    eax, nbr1     ; 第1个操作数
        IFNB   <nbr2>
        add    eax, nbr2     ; 第2个操作数
```

```
ENDIF
IFNB  <nbr3>
add   eax, nbr3     ; 第3个操作数
ENDIF
IFNB  <nbr4>
add   eax, nbr4     ; 第4个操作数
ENDIF
IFNB  <nbr5>
add   eax, nbr5     ; 第5个操作数
ENDIF
ENDM
```

假定宏调用:

```
addAll   ebx, ecx, edx, number, 1
```

五个宏参数中的每一个参数都有一个对应的实参，所以宏展开为:

```
mov  eax, ebx       ; 操作数1
add  eax, ecx       ; 操作数2
add  eax, edx       ; 操作数3
add  eax, number    ; 操作数4
add  eax, 1         ; 操作数5
```

宏调用:

```
addAll  ebx, ecx, 45   ; value1 + value2 + 45
```

仅有三个实参。实参ebx表示为参数nbr1的值，ecx替代nbr2，45替代nbr3，但是，参数nbr4和nbr5是空值。因此，宏展开语句为:

```
mov  eax, ebx    ; 操作数1
add  eax, ecx    ; 操作数2
add  eax, 45     ; 操作数3
```

一般很少这样做，除了尾部实参外，其他的实参都可以忽略。例如，宏调用语句:

```
addAll  ebx,    , ecx
```

有ebx与nbr1相对应，ecx与nbr3相对应，所有其他的参数是空值。因此，宏展开为:

```
mov  eax, ebx    ; 操作数1
add  eax, ecx    ; 操作数2
```

如果在addAll宏调用中第一个实参被忽略，则该宏仍将被展开。但其结果是语句序列中将包含一个缺少操作数的mov指令，这样，该语句将使得MASM产生一个错误信息。例如，宏调用:

```
addAll   , value1, value2
```

展开为:

```
mov  eax,           ; 操作数1
add  eax, value1    ; 操作数2
add  eax, value2    ; 操作数3
```

通过调用来举例说明addAll宏的一个特殊使用:

```
addAll  value, eax, eax, value, eax   ; 10 * value
```

展开为:

```
mov  eax, value   ; 操作数1
add  eax, eax     ; 操作数2
add  eax, eax     ; 操作数3
add  eax, value   ; 操作数4
add  eax, eax     ; 操作数5
```

注释“10*value”解释了这个调用的目的。

微软汇编器提供了一些条件汇编的指示性语句。IFNB指示有一个对应的指示性语句IFB(“如果为空”),用于检查宏参数是否为空。

IF和IFE指示性语句可以检查一个表达式在汇编时所确定的值。对于IF,如果表达式的值不为0,那么,MASM汇编条件代码;对于IFE,如果表达式的值是0,那么,MASM汇编条件代码。

IFDEF和IFNDEF类似于IF和IFE。IFDEF和IFNDEF检查符号,MASM汇编条件代码取决于该符号是否在先前的程序中被定义过。

每个条件汇编块以ENDIF指示性语句结束。可以使用ELSEIF和ELSE指示性语句来提供可选择的代码。通常,条件汇编代码块看起来像:

```
IF...  [操作数]
语句
ELSEIF ...
语句
ELSE
语句
ENDIF
```

操作数随着IF的类型而变化,但并不是与所有的类型都能使用。跟在IF后的ELSEIF和语句行是可选择的,跟在IF后的ELSE和语句行也是。在IF之后可有多个ELSEIF,但最多只能有一个ELSE。

以上的语法非常类似于许多高级语言中所出现的语法。值得注意的是,这些指示性语句在汇编时被使用,而不是在执行时被使用。也就是说,这些指示性语句控制语句的汇编并在后来执行该语句,而不是控制语句执行的顺序。

使用指示性语句EXITM可使一些宏定义更易于编写和理解。当MASM在处理一个宏调用并且发现了指示性语句EXITM时,MASM立即停止展开宏,忽略在该宏定义中EXITM后面的任何语句。有如下设计:

```
if 条件为真
then
    处理满足条件为真的汇编语言语句;
else
    处理条件为假的汇编语言语句;
end if;
```

可选用另一种设计:

```
if 条件为真
then
    处理满足条件为真的汇编语言语句;
    终止宏展开;
end if;

处理条件为假的汇编语言语句
```

假设在这些拟定的设计后没有宏定义的语句，二者是相当的。这些可选用的设计能通过以下来实现:

```
IF... [操作数条件为真]
处理满足条件为真的汇编语言语句
ELSE
处理条件为假的汇编语言语句
ENDIF
```

和

```
IF... [操作数条件为真]
处理满足条件为真的汇编语言语句
EXITM
ENDIF
处理条件为假的汇编语言语句
```

注意，当使用了ELSE时，则不需要EXITM。代码段9-7给出了使用了EXITM的宏定义。

代码段9-7 改进后的宏min2

```
min2        MACRO   value1,value2,extra
            LOCAL   endIfLess
; 求value1和value2的最小数，并放入到EAX

            IFB     <value1>
            .ERR    <first argument missing in min2 macro>
            EXITM
            ENDIF

            IFB     <value2>
            .ERR    <second argument missing in min2 macro>
            EXITM
            ENDIF

            IFNB    <extra>
            .ERR    <more than two arguments in min2 macro>
            EXITM
            ENDIF
```

```
            mov   eax, value1    ;; value1放入EAX
            cmp   eax, value2    ;; value1<= value2?
            jle   endIfLess      ;; 如果是则执行
            mov   eax, value2    ;; 否则value2<value1
    endIfLess:
            ENDM
```

前面章节的例子说明了宏调用由于缺少实参而被展开为非法语句，这样的非法语句可由MASM在随后的汇编时而不是宏展开时发现。设计宏定义时，要有安全措施来确保宏调用中包含正确的实参个数，或者用其他有效的方法调用。条件汇编指示性语句可能做到这一点。但是，如果靠避免产生非法语句来消除汇编错误，那么，用户可能不知道宏调用何时出错了，它需要额外的措施来告知用户出错。指示性语句.ERR就是一种有效的实现方法。.ERR指示性语句在汇编时产生一个强制性的错误，如果有错误信息，就将信息显示在控制台和列表文件中。该指示性语句也确保汇编时没有.obj文件生成。.ERR指示后常跟着一个“<”和“>”包含的串，这个字符串也包含在错误信息中。

代码段9-7中的min2宏定义结合了安全措施，以确保宏被调用时带有正确的参数个数。条件块:

```
IFB  <value1>
.ERR <first argument missing in min2 macro>
EXITM
ENDIF
```

检查第一个实参。如果缺少第一个实参，那么，.ERR指示显示信息“first argument missing in min2 macro.”。注意，条件代码块以EXITM结束，所以，如果缺少了第一个实参，这个宏不会做进一步的展开。另一个可选的方法也可避免其他宏的展开，对于第一个条件块，将宏定义的其他部分嵌入在ELSE和ENDIF之间。

条件块:

```
IFB <value2>
.ERR  <second argument missing in min2 macro>
EXITM
ENDIF
```

检查第二个实参，如果第二个实参缺少，则报告出错。条件块:

```
IFNB  <extra>
.ERR  <more than two arguments in min2 macro>
EXITM
ENDIF
```

告诉MASM去检查第三个实参是否在展开的宏调用列表中。在这里，应当没有第三个实参，所以，如果实参不为空，则产生错误信息。

练习9.4

使用代码段9-7中给出的宏定义min2，写出以下每个宏调用展开时所对应的语句序列。

(a) `min2  nbr1, nbr2`
(假设局部符号计数器在0004)
(b) `min2,  value`
(假设局部符号计数器在0011)
(c) `min2  ecx`
(假设局部符号计数器在000B)
(d) `min2  nbr1, nbr2, nbr3`
(假设局部符号计数器在01D0)

编程练习9.4

1. 重写代码段9-4中给出的宏swap定义，使宏swap调用必须有且仅有两个实参；如果缺少实参或有多余的实参，使用.ERR提示相关错误信息。
2. 写一个宏定义min3，它有且仅有三个双字整型数的参数，把三个数中最小的数放入EAX中。如果在调用min3中缺少实参或者有多余的实参，使用.ERR提示相关错误信息。

9.5 IO.H中的宏

IO.H文件中的宏用来提供简单、安全地访问标准输入输出设备。代码段9-8给出了IO.H文件的内容，本节余下的部分将讨论IO.H文件中的指示性语句和宏。

代码段9-8 IO.H

```
; I/O宏的IO.H头文件
; 平面存储模式的32位版本
; R. Detmer    日期: 2000年8月
.NOLIST      ; 关闭列表
.386

            EXTRN  itoaproc:near32, atoiproc:near32
            EXTRN  dtoaproc:near32, atodproc:near32
            EXTRN  inproc:near32, outproc:near32

itoa        MACRO  dest,source,xtra      ;; 转换整数到ASCII字符串

            IFB    <source>
            .ERR <missing operand(s) in ITOA>
            EXITM
            ENDIF

            IFNB   <xtra>
            .ERR <extra operand(s) in ITOA>
            EXITM
            ENDIF

            push   ebx                   ;; 保存EBX
            mov    bx, source
            push   bx                    ;; 源数形参
            lea    ebx,dest              ;; 目的地址
            push   ebx                   ;; 目的地址形参
```

```
        call    itoaproc                ;; 调用itoaproc(source, dest)
        pop     ebx                     ;; 恢复EBX值
        ENDM

atoi    MACRO   source,xtra             ;; 转换ascii字符串为AX中的整数
                                        ;; ESI中存放结束字符的偏移地址

        IFB     <source>
        .ERR <missing operand in ATOI>
        EXITM
        ENDIF

        IFNB    <xtra>
        .ERR <extra operand(s) in ATOI>
        EXITM
        ENDIF

        push    ebx                     ;; 保存EBX
        lea     ebx,source              ;; 源地址赋值给EBX
        push    ebx                     ;; 源地址参数传递到堆栈
        call    atoiproc                ;; 调用atoiproc(source)
        pop     ebx                     ;; ret移除参数
        ENDM

dtoa    MACRO   dest,source,xtra        ;; 转换双精度数为ASCII字符串

        IFB     <source>
        .ERR <missing operand(s) in DTOA>
        EXITM
        ENDIF

        IFNB    <xtra>
        .ERR <extra operand(s) in DTOA>
        EXITM
        ENDIF

        push    ebx                     ;; 保存EBX
        mov     ebx, source
        push    ebx                     ;; 源参数
        lea     ebx,dest                ;; 取得目的地址
        push    ebx                     ;; 目的参数
        call    dtoaproc                ;; 调用dtoaproc(source, dest)
        pop     ebx                     ;; 恢复EBX值
        ENDM

atod    MACRO   source,xtra             ;; 转换ascii字符串为EAX中的整数
                                        ;; ESI中存放结束字符的偏移地址

        IFB     <source>
        .ERR <missing operand in ATOD>
        EXITM
        ENDIF

        IFNB    <xtra>
        .ERR <extra operand(s) in ATOD>
```

```
            EXITM
            ENDIF

            lea     eax,source          ;; 源地址存放到EAX
            push    eax                 ;; 源地址参数传递到堆栈
            call    atodproc            ;; 调用atodproc(source)
                                        ;; ret移出参数
            ENDM

output      MACRO   string,xtra         ;; 显示字符串

            IFB     <string>
            .ERR <missing operand in OUTPUT>
            EXITM
            ENDIF

            IFNB    <xtra>
            .ERR <extra operand(s) in OUTPUT>
            EXITM
            ENDIF

            push    eax                 ;; 保存EAX
            lea     eax,string          ;; 字符串地址
            push    eax                 ;; 地址参数传递到堆栈
            call    outproc             ;; 调用outproc(string)
            pop     eax                 ;; 恢复EAX值
            ENDM

input       MACRO   dest,length,xtra    ;; 从键盘读取字符串

            IFB     <length>
            .ERR <missing operand(s) in INPUT>
            EXITM
            ENDIF

            IFNB    <xtra>
            .ERR <extra operand(s) in INPUT>
            EXITM
            ENDIF

            push    ebx                 ;; 保存EBX
            lea     ebx,dest            ;; 目的地址
            push    ebx                 ;; 目的地址参数传递到堆栈
            mov     ebx,length          ;; 缓存区长度
            push    ebx                 ;; 堆栈中参数长度
            call    inproc              ;; 调用inproc(dest, length)
            pop     ebx                 ;; 恢复EBX值
            ENDM

.NOLISTMACRO                            ; 阻止宏扩展列表
.LIST                                   ; 开始列表
```

大多数IO.H文件由宏定义组成，当使用IO.H文件时，生成代码以调用外部的过程。同时，IO.H文件也包含了其他指示性语句，以.NOLIST指示性语句开始，特别地，在IO.H的内容中，它使得所有源代码不出现在列表文件中。接着，后面有EXTRN指示性语句，该指令声明被宏

调用的外部过程。IO.H文件以.NOLISTMACRO指示性语句结束，以此语句来禁止任何宏展开在列表文件中，并且还有一个.LIST指示性语句，以使得跟在INCLUDE io.h后面的用户语句可再次显示在列表文件中。

IO.H文件由宏itoa、atoi、dtoa、atod、output和宏input的定义组成。这些宏定义的结构很相似。每个宏都使用IFB和IFNB来检查宏调用是否有正确的实参个数。如果实参的个数不正确，则使用.ERR来生成强制的错误信息及相关信息。事实上，这种检查不是非常完整的。

假设宏调用的参数是正确的，并且input/output宏调用展开为一个指令序列，该指令序列调用了外部过程。例如，宏itoa调用外部过程itoaproc。通过堆栈传递参数，但是，有些代码序列使用了寄存器来临时存储一个值，通过push和pop指令来确保宏调用后这些寄存器的值不发生改变。

练习9.5

注意，如果缺少一个或两个实参，宏itoa将产生一个错误信息。重定义宏itoa以提供全部的实参检测。也就是说，分别检测是否缺少实参source和dest，并为每个缺少的实参生成详细的信息。考虑两个实参可能都缺少的情况。

本章小结

本章讨论了汇编的过程。典型的两次性汇编器两次扫描汇编语言程序，它使用了地址计数器，在第一遍扫描时构建一个符号表，并在第二次扫描时完成汇编。符号表包含了程序中使用的每个标识符的信息，包括其类型、大小和地址。如果解决了前向引用，即修正了目标代码，那么，汇编器也可以一次性汇编。

一个机器指令可以有一个或更多个前缀字节。但是，对于每条80x86指令，机器码的主要字节是其操作码。有些指令是单字节长，但是大多数指令由多字节组成，其下一个字节通常是mod reg r/m格式，其中reg表示源寄存器或目的寄存器，并且与其他两个域结合起来描述寻址方式。其他的指令字节包括附加的寻址信息、立即数或者存储器操作数的地址。

宏由MACRO和ENDM来定义。宏可以使用参数，该参数与宏调用时的实参相对应。汇编时展开宏调用。宏调用的展开语句是宏定义中的语句，并用实参来代替参数。宏定义可以声明局部标号，因而MASM对于不同的宏调用可展开为不同的符号。

条件汇编可用在常规代码中，也可用在宏定义中，根据汇编时检查到的条件，生成不同的语句。IFB和IFNB用在宏中以检查是否缺少实参或者存在实参。还有一些其他的条件汇编语句，包括IF、IFE、IFDEF和IFNDEF。ELSE语句可用于提供两个可选择的代码块，并且ENDIF用于结束该条件汇编块。

如果汇编器在展开宏定义时遇到了一个EXITM，它立即中止该宏的展开。.ERR触发一个强制错误，以使MASM显示一个错误信息，并且不为该汇编器生成.OBJ文件。

IO.H文件包含对宏input/output等的定义及其他一些指示性语句。这些宏定义使用条件汇编来判断是否缺少实参或者有多余的实参，并且生成调用外部过程的代码。

第10章　浮点数运算

本书集中了以二进制补码为主的整型数的多种表示方法，因为所有80x86微处理器都有一些处理二进制补码的指令。许多80x86微处理器系统（包括所有的Pentium处理器系统、486DX的系统以及其他系统）都配备了一个浮点数协微处理器，它具有处理浮点数形式存储的数据的能力。

本书前面的1.5节描述了用32位存储浮点数的IEEE格式（Institute of Electrical and Electronics Engineers，美国电气与电子工程师学会）。MASM汇编器的一些指令允许十进制操作数，并且可使用IEEE格式初始化存储器。要在一台PC机上实现浮点数算法有两种方法：要么微处理器内置浮点型单元，要么配置浮点数协微处理器，这样就可以直接使用浮点型指令。否则，就要使用实现加法，乘法等运算的过程集。

10.1节描述了80x86浮点数的结构。10.2节论述了如何实现浮点数和ASCII码以及其他数值表示形式之间的相互转化。10.3节给出了浮点数加法、减法、乘法、除法、取反和比较运算的模拟例程，对于在一个没有内置浮点型指令的80x86微处理器而言，这些例程对进行浮点数运算将是非常有用的。在10.3节将给出一些用汇编语言实现算法的实例，这些算法复杂度适中、并且很有用。同时，还例举了一些本书先前未提及的方法。10.4节简要的介绍了如何在C++代码中嵌入汇编语言，用C++实现输入输出操作，用汇编语言来实现浮点数运算。但是，嵌入汇编语言代码并不仅限于浮点型指令。

10.1　80x86浮点数结构

如上所述，一些80x86微处理器没有内置浮点数功能，它们依赖一个浮点数协处理器芯片来执行浮点型指令。芯片中的浮点运算器（floating-point unit，FPU）几乎全部都是独立于其他芯片的。它有自己的内部寄存器，与其他80x86寄存器完全分离。浮点型协处理器芯片执行指令，以实现浮点数算术运算，包括加法和乘法等常规运算，以及一些如超越函数求值的之类复杂运算。浮点数协处理器芯片不仅可以和存储器之间传递浮点型操作数，还可以在协微处理器之间传递整数或者BCD操作数。当非浮点数放入浮点型寄存器时，它将转换成浮点数格式；一个内部浮点型的数值放入存储器时，这个浮点数将转换成整数或者BCD格式。

一个FPU有八个数据寄存器，每个80位长。数据在这些寄存器中（根据IEEE标准）以十字节的浮点数形式存放。这些寄存器基本上是作为堆栈组织在一起的，例如：如果使用fld（floating load）指令将一个数值从存储器传递到浮点型单元，那么该数值被放入堆栈顶端的寄存器，数据存储在栈顶，同时其他寄存器依次下推。然而，只有某些指令可以访问这八个寄存器中的任意一个，因此，这样的组织不是一个“纯”堆栈。

八个浮点型寄存器是：

- ST，栈顶，也称为ST(0)，
- ST(1)，仅居于顶栈之下的寄存器，

- ST(2)，居于ST(1)之下的寄存器，
- ST(3)，ST(4)，ST(5)，ST(6)，和
- ST(7)，是居于堆栈底部的寄存器。

除了这八个寄存器外，浮点型单元还有几个16位的控制寄存器。状态字的某些位是通过浮点数比较指令来赋值的，而且必须检查这些位，80x86才能执行基于浮点数比较的条件转移指令。FPU中控制字的位有时必须设置，以确保某些取整模式。

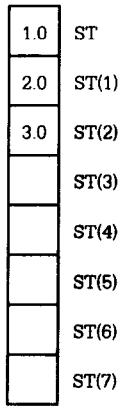
在考虑浮点型指令前，要注意：每一个浮点数助记符都是由字母F开始的，而非浮点型指令不会以字母F开始。多数浮点型指令作用于栈顶ST，其他操作数存放在另一个浮点型寄存器或存储器中。浮点型指令不能在一个普通寄存器（如EAX）和一个浮点型寄存器之间传递数据，要实现这样的传递，必须要使用一个存储器作为中间转存单元（但是，寄存器AX中要有指令存储状态字或者控制字）。

浮点型指令是以段为单位进行验证的，指令的开始是将操作数进栈，表10-1列出指令的助记符。

表10-1 浮点数载入指令

助记符	操作数	功能
fld	memory(real)	将存储器中的实型数值压入堆栈
fild	memory(integer)	将存储器的整型数值转化为浮点数并压入堆栈
fbld	memory(BCD)	将存储器的BCD码转化为浮点数并压入堆栈
fld	st(*num*)	将浮点型寄存器中的数压入堆栈
fld1	(none)	1.0压入堆栈
fldz	(none)	0.0压入堆栈
fldpi	(none)	π(pi) 压入堆栈
fldl2e	(none)	$\log_2(e)$ 压入堆栈
fldl2t	(none)	$\log_2(10)$ 压入堆栈
fldlg2	(none)	$\log_{10}(2)$ 压入堆栈
fldln2	(none)	$\log_e(2)$ 压入堆栈

下面举例说明这些指令是如何工作的。假设浮点型寄存器堆栈状态如下：

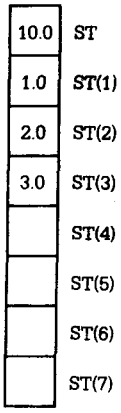


其中的数值是用十进制表示的，而不是IEEE浮点数形式。如果数据段包含：

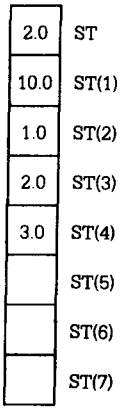
```
fpValue     REAL4     10.0
intValue    DWORD     20
```

```
bcdValue    TBYTE       30
```

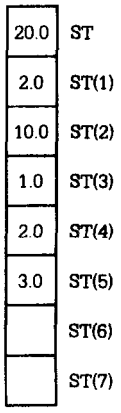
那么这些数值汇编后，fpValue的值为41200000，intValue的值为00000014，bcdValue的值为00000000000000000030。如果执行指令fld fpValue，那么寄存器堆栈将是:



原本堆栈中的数值都被下推了一个位置。在这样的初始值情况下，若执行指令fld st(2)，寄存器堆栈将是:



注意: 来自ST(2) 的数值2.0被推到栈顶，但它并未从原来的堆栈中移出。此时执行指令fild intValue，则:



32位的数值00000014被转化成一个80位的浮点型数值，但这样的转化看起来并不明显。一个合法的整型操作数必须是一个字长、双字长或者四字长，字节长的整型操作数也允许。本章没有介绍浮点型指令的操作码。

如果执行指令fbld bcdValue，堆栈中的数值将是:

30.0	ST
20.0	ST(1)
2.0	ST(2)
10.0	ST(3)
1.0	ST(4)
2.0	ST(5)
3.0	ST(6)
	ST(7)

其中，80位的BCD码被转换成为了完全不同的80位浮点数格式。最后，如果执行指令fldz，那么寄存器堆栈将成为:

0.0	ST
30.0	ST(1)
20.0	ST(2)
2.0	ST(3)
10.0	ST(4)
1.0	ST(5)
2.0	ST(6)
3.0	ST(7)

现在，这个堆栈满了。如果不从堆栈中弹出一些数值或清空堆栈，那么堆栈中将不能放入后继的数值。指令finit可以初始化浮点型单元，并清空八个寄存器的内容。通常，用浮点型单元的程序都会包括这样的语句:

```
finit        ; 初始化算术处理器
```

这条语句出现在代码的开始部分，程序中可能需要对浮点型单元重新初始化，但通常没有这个必要，因为从堆栈中弹出的值不会继续堆积在堆栈中。

使用Windbg程序可以跟踪浮点数的运算。图10-1在屏幕的左边方框中给出了执行的代码段，在右边方框中显示了浮点数窗口。

表10-2列出了一些浮点型指令，它们可用于将栈顶数据复制到存储器或到其他浮点型寄存器。这些指令大多都是成对的：每对指令中的一条仅仅是复制ST到目的地，另一条指令类似，它也复制ST到它的目的地，但是，它还将ST从寄存器堆栈中取出来。

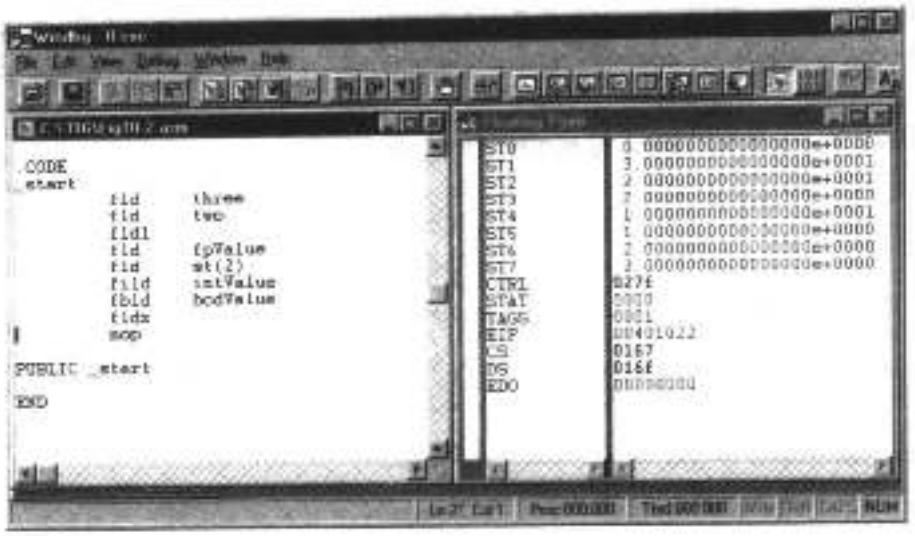


图10-1 Windbg窗口显示的浮点数指令的执行

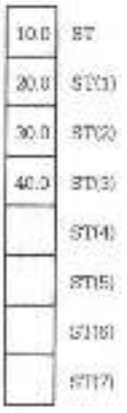
表10-2 浮点数数据存储指令

助记符	操作数	功能
fst	st(num)	复制ST的值来替换ST(num)的内容；只有ST(num)是受影响的
fstp	st(num)	复制ST的值来替换ST(num)的内容；ST出栈
fst	memory(real)	复制ST的值为实型数，存入存储器；堆栈不受影响
fstp	memory(real)	复制ST的值为实型数，存入存储器；ST出栈
fist	memory(integer)	复制ST的值，并转换为整型数存入存储器
fistp	memory(integer)	复制ST的值，并转换为整型数存入存储器；ST出栈
fbstp	memory(BCD)	复制ST的值，并转换为BCD码存入存储器；ST出栈

下面举例说明这些指令的不同作用。假设在数据段中有如下指令：

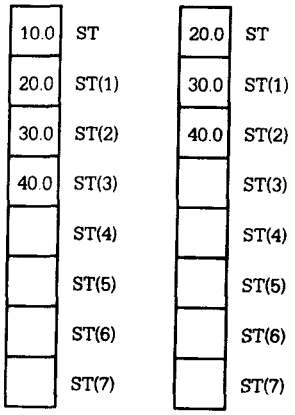
```
intValue    DWORD ?
```

设浮点型寄存器的状态如下：

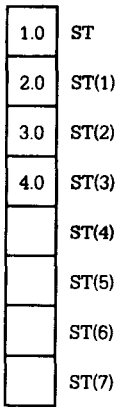


左下图显示了指令fist intValue执行后的堆栈情况，右下图显示了指令fistp intValue执行后的堆

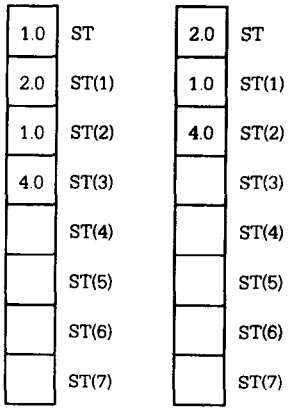
栈的情况。在这两种情况下，intValue中的值都是0000000A，这是浮点数10.0的双字长二进制补码表示的整型数。



如果目的地址是一个浮点型寄存器，那么情况就有点复杂。假设在执行的时候，浮点型寄存器的内容如下:



左下图显示了执行fst st(2) 后的堆栈内容，右下图显示了执行fstp st(2) 后的堆栈情况。在第一种情况下，复制的ST已经保存在ST(2) 中，在第二种情况下，复制ST后，然后ST的内容被弹出。



除了前面列出的装载和存储指令之外，浮点型单元还有一条fxch指令，用来实现两个浮点型寄存器内容的互换。

如果没有操作数，指令：

```
fxch            ; 互换ST和ST(1)
```

互换了在栈顶的ST和仅次于ST的ST(1)的内容。如果有操作数，例如：

```
fxch    st(3)   ; 互换ST和ST(3)
```

将ST和指定的寄存器互换。

表10-3列出了浮点型加法的指令。这里有多种加法形式：将ST中的内容加到其他寄存器中、将任一寄存器中的内容加到ST中、将存储器中的一个实型数加到ST中、或者将存储器中的一个整型数加到ST中。没有使用BCD码数的加法形式。在将栈顶内容加到另一个寄存器中后，指令faddp将它从栈顶取出，这样两个操作数都改变了。

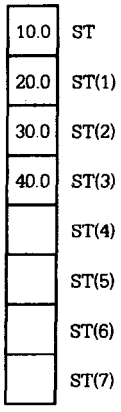
表10-3 浮点型加法指令

助记符	操作数	功能
fadd	(none)	将ST和ST(1)出栈；将两个值相加；并将它们的和入栈
fadd	st(*num*), st	将ST(*num*)和ST相加；用和替换ST(*num*)
fadd	st, st(*num*)	将ST(*num*)和ST相加；用和替换ST
fadd	memory(real)	将ST和存储器中的实型数相加；用和替换ST
fiadd	memory(integer)	将ST和存储器中的整型数相加；用和替换ST
faddp	st(*num*), st	将ST(*num*)和ST相加；用和替换ST(*num*)；将ST出栈

下面举一些例子说明浮点型加法指令是如何执行的。假设数据定义段包含如下指令：

```
fpValue     REAL4   5.0
intValue    DWORD   1
```

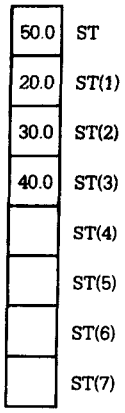
并且，浮点型寄存器堆栈内容如下：



在执行指令

```
fadd    st, st(3)
```

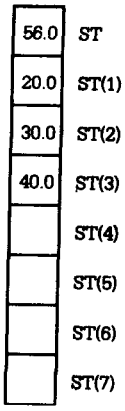
之后，堆栈内容为：



在此基础上，执行下列两条指令

```
fadd    fpValue
fiadd   intValue
```

之后，堆栈内容为:



最后，如果执行指令

```
faddp    st(2), st
```

之后，堆栈内容将是:

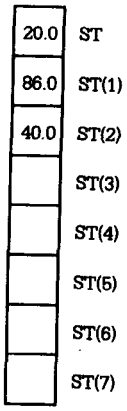
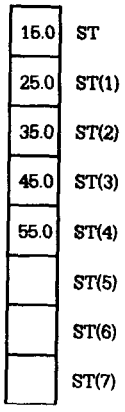


表10-4列出了减法指令。前六条指令和相应加法指令相似。后六条减法指令的操作数是以相反的顺序被减的，这样很方便，因为减法中的被减数和减数的顺序是不可交换的。

表10-4 浮点型减法指令

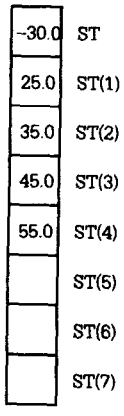
助 记 符	操 作 数	功 能
fsub	(none)	将ST和ST(1)出栈；计算ST(1)减ST的值；将差值入栈
fsub	st(*num*), st	计算ST(*num*) 减ST的值；用差值替换ST(*num*)
fsub	st, st(*num*)	计算ST减ST(*num*) 的值；用差值替换ST
fsub	memory(real)	计算ST减存储器中的实型数的值；用差值替换ST
fisub	memory(integer)	计算ST减存储器中的整型数的值；用差值替换ST
fsubp	st(*num*), st	计算ST(*num*) 减ST的值；用差值替换ST(*num*)；将ST出栈
fsubr	(none)	将ST和ST(1) 出栈；计算ST减ST(1) 的值；将差值入栈
fsubr	st(*num*), st	计算ST－ST(*num*) 的值；用差值替换ST(*num*)
fsubr	st, st(*num*)	计算ST(*num*) 减ST的值；用差值替换ST
fsubr	memory(real)	计算存储器中实型数值减ST的值；用差值替换ST
fisubr	memory(integer)	计算存储器中整型数值减ST的值；用差值替换ST
fsubpr	st(*num*), st	计算ST减ST(*num*) 的值；用差值替换ST(*num*)；将ST出栈

下面举例说明这些类似的减法指令在功能上的区别。若浮点型寄存器堆栈内容如下：

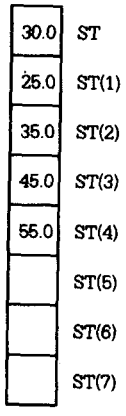


下面的两张图分别显示了执行指令fsub st, st(3) 和指令fsubr st, st(3) 后的结果：

执行fsub st, st(3)后



执行fsubr st, st(3)后



乘法和除法指令分别如表10-5和表10-6所示。乘法指令和表10-3中的加法指令形式相同，除法指令和表10-4中的减法指令形式相同。也就是说，R交换了操作数的被除数和除数的位置。

表10-5 浮点型乘法指令

助记符	操作数	功能
fmul	(none)	将ST和ST(1) 出栈；并将它们的值相乘；乘积入栈
fmul	st(*num*), st	将ST(*num*) 和ST相乘；用乘积来替换ST(*num*)
fmul	st, st(*num*)	将ST和ST(*num*) 相乘；用乘积来替换 ST
fmul	memory(real)	将ST和存储器中的实型数相乘；用乘积来替换ST
fimul	memory(integer)	将ST和存储器中的整型数相乘；用乘积来替换ST
fmulp	st(*num*), st	将ST(*num*) 和ST相乘；用乘积来替换ST(*num*)；并将ST出栈

表10-6 浮点型除法指令

助记符	操作数	功能
fdiv	(none)	将ST和 ST(1)出栈；计算ST(1)/ST的值；并将商入栈
fdiv	st(*num*), st	计算ST(*num*)/ST的值；用商来替换ST(*num*)
fdiv	st, st(*num*)	计算ST/ST(*num*) 的值；用商来替换ST
fdiv	memory(real)	计算ST/存储器中的实型数；用商来替换ST
fidiv	memory(integer)	计算ST/存储器中的整型数；用商来替换ST
fdivp	st(*num*), st	计算ST(*num*)/ST的值；用商来替换ST(*num*)；并将ST出栈
fdivr	(none)	将ST和ST(1) 出栈；计算ST/ST(1) 的值；并将商入栈
fdivr	st(*num*), st	计算ST/ST(*num*) 的值；用商来替换ST(*num*)
fdivr	st, st(*num*)	计算ST(*num*)/ST的值；用商来替换ST
fdivr	memory(real)	计算存储器中的实型数/ST；用商来替换ST
fidivr	memory(integer)	计算存储器中的整型数/ST；用商来替换ST
fdivpr	st(*num*), st	计算ST/ST(*num*) 的值；用商来替换ST(*num*)；并将ST出栈

表10-7列出了4条附加的浮点型指令，用于计算正切函数、反正切函数、指数函数和对数函数，本书没有涉及这些内容。

表10-7 附加的浮点型指令

助记符	操作数	功能
fabs	(none)	ST := |ST| (绝对值)
fchs	(none)	ST := －ST (相反数)
frndint	(none)	对ST取整
fsqrt	(none)	用ST的平方根来替换ST

浮点型单元提供了比较栈顶ST和第二操作数的指令集，表10-8列举出了这些指令。

表10-8 浮点型比较指令

助记符	操作数	功能
fcom	(none)	比较ST和ST(1)
fcom	st(*num*)	比较ST和ST(*num*)
fcom	memory(real)	比较ST和存储器中的实型数

（续）

助记符	操作数	功能
ficom	memory(integer)	比较ST和存储器中的整型数
ftst	(none)	比较ST和0.0
fcomp	(none)	比较ST和ST(1)；然后出栈
fcomp	st(*num*)	比较ST和ST(*num*)；然后出栈
fcomp	memory(real)	比较ST和存储器中的实型数；然后出栈
ficomp	memory(integer)	比较ST和存储器中的整型数；然后出栈
fcompp	(none)	比较ST和ST(1)；然后出栈两次

回想一下，浮点型单元有一个称为状态字的16位控制寄存器。比较指令可以给这个状态字的第14位、第10位和第8位赋值。这些“条件码”位分别被称为C3、C2和C0，它们的设置如下：

比较结果	C3	C2	C0
ST>第二操作数	0	0	0
ST<第二操作数	0	0	1
ST = 第二操作数	1	0	0

另外还有一种可能是两个操作数不可比。如果其中一个操作数是IEEE表示的无穷大或者不是一个数值（NaN）时，那么，这种情况就会出现。此时，这三位“条件码”位都设置为“1”。

如果做比较是为了确定程序的分支，那么只在状态字中简单设置标志位是没有什么作用的。在80x86中，条件转移指令是参考标志寄存器的某些位，而不是浮点型单元中的状态字。因此，在用80x86指令（如test指令）测试状态字的一些位之前，状态字必须复制到存储器或者AX寄存器中。浮点型单元有两条指令可以用来存储状态字，表10-9对它们进行了总结，其中的表还列出了存储和设置控制字的指令。

表10-9 混合浮点型指令

助记符	操作数	功能
fstsw	memory word	复制状态寄存器到存储器字
fstsw	AX	复制状态寄存器到AX寄存器
fstcw	memory word	复制控制字寄存器到存储器
fldcw	memory word	复制存储器字到控制字寄存器

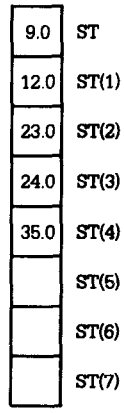
实际上，80x86浮点型和整型单元能够并发执行指令，因此，在某些情况下，用汇编语言编程要特别小心，本书不对此展开讨论。

练习10.1

1. 假设一个程序的数据段内容如下：

```
fpValue     REAL4    0.5
intValue    DWORD    6
```

相关代码还没有执行，程序还没有修改它们的数值。浮点型寄存器堆栈内容如下：



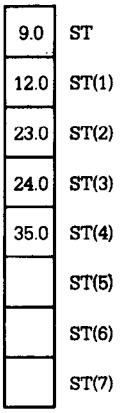
假设这些数值在下列各指令执行之前都是正确的。而且下列指令都是独立执行的，不存在哪条指令先执行，哪条指令后执行的问题。请给出执行下列指令后，fpValue和intValue的浮点型寄存器堆栈的内容。

(a) `fld st(2)`
(b) `fld fpValue`
(c) `fild intValue`
(d) `fldpi`
(e) `fst st(4)`
(f) `fstp st(4)`
(g) `fst fpValue`
(h) `fistp intValue`
(i) `fxch st(3)`
(j) `fadd`
(k) `fadd st(3), st`
(l) `fadd st, st(3)`
(m) `faddp st(3), st`
(n) `fsub fpValue`
(o) `fisub intValue`
(p) `fisubr intValue`
(q) `fsubp st(3), st`
(r) `fmul st, st(4)`
(s) `fmul`
(t) `fmul fpValue`
(u) `fdiv`
(v) `fdivr`
(w) `fidiv intValue`
(x) `fdivp st(2), st`
(y) `fchs`
(z) `fsqrt`

2. 假设某程序数据段内容如下:

```
fpValue     REAL4    1.5
intValue    DWORD    9
```

相关代码还未执行，程序还没有修改它们的数值。假设浮点型寄存器堆栈内容如下:



假设这些数值在下列各指令执行之前都是正确的。给出下列指令执行后状态字标志位C3、C2和C0的内容。

(a) `fcom`

(b) `fcom  st(3)`

(c) `fcom  fpValue`

(d) `ficom  intValue`

对于下面后继的两条指令，请给出指令执行后堆栈的内容。

(e) `fcomp`

(f) `fcompp`

10.2 浮点型指令编程

本节给出了三个使用浮点型指令编程的例子。第一个例子是计算两个数的平方和的平方根。虽然该例中没给出任何易于浮点数输入/输出的过程，但可以在Windbg中看到FPU的操作。第二和第三个例子提供了浮点数输入/输出的过程。

代码段10-1列出了第一个例子，value1和value2是浮点型数值。第一条指令将存储器中的value1复制给ST。第二条指令将value1从ST复制到ST，下推第一个堆栈条目到了ST(1)。第三条指令将value1*value1的值赋给ST，ST(1)为空。(当然，每个浮点型寄存器中总是有某个值。)对value2进行操作的指令顺序同上。图10-2给出的Windbg窗口显示的仅仅是第二条fmul指令执行之前的CPU。此时，ST和ST(1)中都有value2的副本，ST(2)中有value1*value1的值。在ST中算出结果后，该结果被存储在sqrt中，并使它出栈，从而堆栈回到最初的状态。

值得注意的是，在图10-2中数值1.2显示为1.2000000476837158e + 0000，那是因为小数点后有非零数字。数值1.2作为浮点数，没有一个精确的表示方法。因此，使用32位的REAL4指令得到近似值，保留了17位的精度。还可以通过使用一个REAL8或者REAL10指令得到一个

更好的近似值，但这要浪费更多的存储字节。

代码段10-1 浮点型运算

```
; 查找两个浮点数的平方和的平方根
; 作者:  R. Detmer
; 日期:  1998年4月

.386
.MODEL FLAT

.STACK  4096                    ; 预留4096个字节的堆栈空间

.DATA                           ; 定义初始化数据段
value1  REAL4   0.5
value2  REAL4   1.2
sqrt    REAL4   ?

.CODE
_start:
        fld     value1          ; value1放入ST
        fld     st              ; value1放入ST和ST(1)
        fmul                    ; value1 *value1的值放入ST
        fld     value2          ; value2放入ST(value1 *value1 in ST(1))
        fld     st              ; value2放入ST和ST(1)
        fmul                    ; value2 *value2放入ST
        fadd                    ; 平方和放入ST
        fsqrt                   ; 平方和的平方根放入ST
        fstp    sqrt            ; 保存结果

PUBLIC  _start
END
```

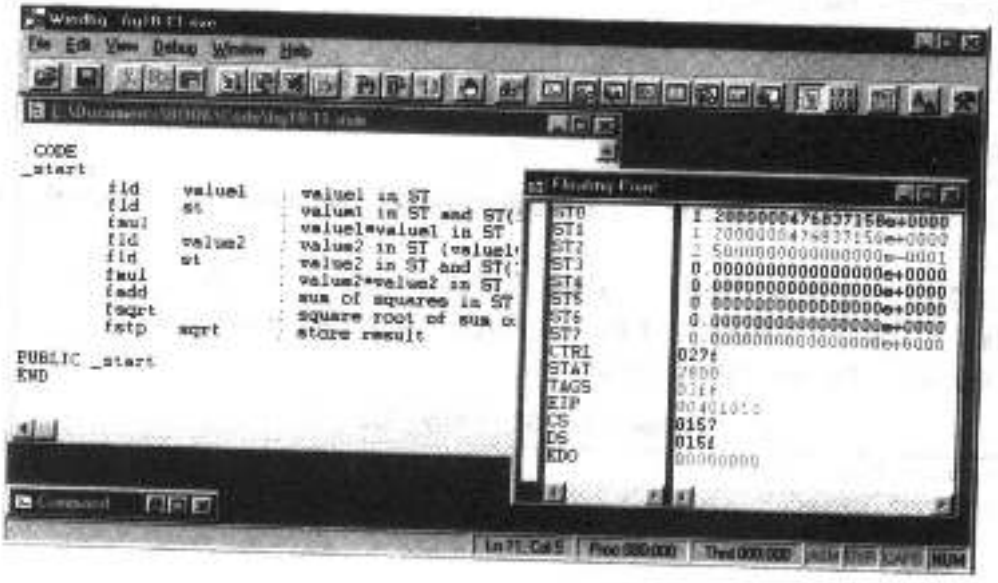


图10-2 浮点数运行实例

第二个例子是实现简单ASCII码到浮点数的转换的算法，如代码段10-2所示。

代码段10-2 ASCII码转换成浮点数的算法

```
value := 0.0;
divisor := 1.0;
point := false;
minus := false;

指向源串的第一个字符;
If 源字符 = '-'
then
    minus := true;
    指向源串的下一个字符;
end if;

while(源字符是一个阿拉伯数字或者小数点) loop
    if 源字符= '.'
    then
        point := true;
    else
        将ASCII码转换成2的补码;
        value := 10*value + float(digit);
        if point
        then
            divisor*10;
            end if;
    end if;
    指向源串的下一个字符;
end while;

value := value/divisor;

if minus
then
    value := -value;
end if;
```

该算法与宏指令atoi和atod很相似，它通过参数扫描所给地址的存储器，将字符解释成一个浮点数。

该算法在一个NEAR32过程（一段程序）atofproc中执行。这个过程有一个参数：字符串的地址。它返回的浮点数值放在ST中。这里没有标志位说明非法的情况，比如是否有多个负号或小数点等，代码如代码段10-3所示。

代码段10-3 ASCII码到浮点数的转换

```
ASCII码转换成浮点数
; 作者: R. Detmer
; 日期: 1998年4月

.386
.MODEL FLAT
```

```
PUBLIC atofproc

false      EQU  0
true       EQU  1

.DATA
ten        REAL4  10.0
point      BYTE   ?
minus      BYTE   ?
digit      WORD   ?

.CODE

atofproc  PROC NEAR32  ; 将ASCII码字符串转换成浮点数
; 参数传递到堆栈: ASCII码源串的地址
; 起始位可以是负号，其他位只能是数字 0-9 和小数点
; 如果是其他字符，扫描终止
; 返回浮点数放入SP

          push ebp                    ; 初始化堆栈
          mov  ebp, esp
          push eax                    ; 保存寄存器内容
          push ebx
          push esi

          fld1                        ; divisor := 1.0
          fldz                        ; value := 0.0
          mov  point, false           ; 小数点未被找到
          mov  minus, false           ; 负号未被找到
          mov  esi, [ebp+8]           ; 源字符的首地址

          cmp  BYTE PTR [esi], '-'    ; 是否为“-”?
          jne  endifMinus             ; 如果不是“-”就跳出
          mov  minus, true            ; “-”被找到
          inc  esi                    ; 指向源操作数下一个字符
endifMinus:

whileOK:  mov  bl, [esi]              ; 源操作数下一个字符
          cmp  bl, '.'                ; 是否小数点?
          jne  endifPoint             ; 如果不是，转移
          mov  point, true            ; 找到小数点
          jmp  nextChar
endifPoint:
          cmp  bl, '0'                ; 是否数字?
          jl   endwhileOK             ; 如果小于'0'，转换
          cmp  bl, '9'
          jg   endwhileOK             ; 如果大于'9'，转换
          and  bx, 000fh              ; 将ASCII码转换成相应整数
          mov  digit, bx              ; 把整数存入存储器
          fmul ten                    ; value := value * 10
          fiadd digit                 ; value := value + digit
          cmp  point, true            ; 是否已经找到小数点?
          jne  endifDec               ; 如果不是，转移
          fxch                        ; 将divisor压入ST, value压入ST(1)
          fmul ten                    ; divisor := divisor * 10
          fxch                        ; value压入ST; divisor压入 ST(1)
```

```
endifDec:
nextChar: inc  esi              ; 指向源操作数下一个字符
          jmp  whileOK
endwhileOK:

          fdivr                 ; value := value/divisor
          cmp  minus, true      ; 是否有负号?
          jne  endifNeg
          fchs                  ; value := -value
endifNeg:
          pop  esi              ; 恢复寄存器内容
          pop  ebx
          pop  eax
          pop  ebp
          ret  4
atofproc  ENDP
          END
```

在ASCII码到浮点数转换算法的程序实现中，divisor使用ST(1)，value使用ST，但是在一个短数据段中，为了修改divisor，divisor使用了ST，而value使用了ST(1)。输入代码后，指令

```
fldl    ; divisor: = 1.0
fldz    ; value: = 0.0
```

对两个变量进行了初始化。注意：divisor的值1.0一直在ST(1)中，因为它被指令fldz下推入栈。

设计：

```
value : = 10*value + float(digit);
```

由如下代码实现：

```
fmul    ten       ; value : = value*10
fiadd   digit     ; value : = value + digit
```

注意：一个字长的二进制补码整数形式的digit存储在存储器中，浮点型单元将把它转换成浮点数，这是fiadd指令的一部分。

为了实现“divisor乘10”，被乘数必须放在ST中。指令

```
fxch            ; divisor放入ST, value放入ST(1)
fmul ten        ; divisor : = divisor * 10
fxch            ; value返回ST; divisor返回ST(1)
```

交换了divisor和value，实现了乘法，并将结果放入ST，然后再交换回来。

接下来是实现“value: = value/divisor”的指令：

```
fdivr           ; value : = value/divisor
```

从栈ST中取出value，ST(1)中取出divisor，计算商，然后将它放回ST。注意：这里如果使用指令fdiv计算“divisor/value”，那么结果是不正确的。除法指令执行后，ST(1)将不再被该过程使用。如果ASCII码字符串是以负号开始的话，指令fchs将改变value的符号。

用一个简单的测试驱动程序来测试atofproc，如代码段10-4所示。这个过程的“输出”可以使用Windbg来观察。

代码段10-4 atofproc的测试驱动程序

```
; atofproc测试驱动程序
; 作者: R. Detmer
; 日期: 1998年4月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
EXTRN atofproc:NEAR32
.STACK  4096                ; 预置4096个字节的堆栈空间

.DATA                       ; 定义初始化数据段
String      BYTE    "435.75", 0

.CODE                       ; 程序段

_start:
            pushd  NEAR32 PTR String
            call   atofproc
            INVOKE ExitProcess, 0
PUBLIC _start
END
```

最后，考察一个将浮点数参数转换成“E记数法”的程序。这个程序生成一个12位长的ASCII码字符串，它包括:

- 一位起始位的负号位或者一位空格
- 一位数字
- 一位小数点
- 五位整数
- 字母E
- 一个加号或者一个减号
- 两位数字

这个字符串代表了十进制数的科学记数法。例如，对于十进制小数145.8798，该程序将它转换成字符串b1.458880E + 02。其中，b代表空格。注意，这个ASCII码字符串有个取整的数值。

代码段10-5给出了将浮点数转换成ASCII码的过程。在起始位的空格或者负号生成后，在剩余字符真正转换前，还必须先取出这些字符。数值被重复地乘以或除以10，直到它的值大于等于1.0且小于10.0为止。如果数值初始化小于1，要用乘法，乘的次数是科学记数法中10的负的若干次方。如果数值初始化大于10，要用除法，除的次数是科学记数法中10的正的若干次方。

代码段10-5 浮点数转换成ASCII码的算法

```
指向目的地址第一个字节;

if value > 0
then
    将空格放入目的串;
```

```
else
    将负号放入目的串;
    value := -value;
end if;
指向目的地址下一个字节;

exponent := 0;
if value ≠0
then
    if value > 10
    then
        until value < 10 loop
            value除以10;
            指数exponent加1;
        end until;
    else
        while value < 1 loop
            value乘以10;
            指数exponent减1;
        end while;
    end if;
end if;

value 加0.000005; {取整}
if value > 10
then
    value除以10;
    指数exponent加1;
end if;

digit := int(value); {截去小数部分, 取整}
将数字转换为ASCII码, 并存入目的串;
指向下一个目的字节;
将小数点存入目的串;
指向下一个目的字节;

for i := 1 to 5 loop
    value := 10 * (value - float(digit));
    digit := int(value);
    将数字转换为ASCII码, 并存入目的串;
    指向下一个目的字节;
end for;
将E存入目的串;
指向下一个目的字节;
if exponent > 0
then
    将正号+放入目的串;
else
```

```
        将负号-放入目的串;
        exponent := -exponent;
    end if;
    指向下一个目的字节;

    将指数转换为两位小数;
    两位小数的指数转换为ASCII码;
    将指数部分存入目的串;
```

小数点后只能显示五位。在1.0和10.0之间的数被通过加0.000005来四舍五入。如果第六位小数位上的数是5或者比5大，那么向高位进一。这时候，有可能产生的和是10.0或者比10.0更大，那么这个数要再被10除一次，指数加一。

对于大于等于1.0但小于10.0的数，在小数点前将其截断，只取一位整数。将这位整数和小数点连接起来。然后由原数值减去之前所保留的整数部分，再用10乘以剩下的小数部分，截取新数的整数部分，这样重复操作，直到最后获得小数点后的五位整数。

在产生了ASCII码字符串的“小数部分”之后，生成字母E，一个指数的加号或者减号以及指数。指数最多包括两位数字。单一的IEEE记数可以表示的最大数为2^{128}，它小于10^{39}。

代码段10-6给出了过程名为ftoaproc的设计代码。该过程有两个参数：第一个是要被转换的浮点数，第二个是目的字符串的地址。

代码段10-6 浮点数转换成ASCII码的实现过程

```
; 浮点数转换成ASCII码
; 作者: R. Detmer
; 日期: 1998年4月

.386
.MODEL FLAT

PUBLIC ftoaproc

C3 EQU 0100000000000000b
C2 EQU 0000010000000000b
C0 EQU 0000000100000000b

.DATA
value      REAL4  ?
ten        REAL4  10.0
one        REAL4  1.0
round      REAL4  0.000005
digit      WORD   ?
exponent   WORD   ?
controlWd  WORD   ?
byteTen    BYTE   10

.CODE
ftoaproc   PROC NEAR32   ; 将浮点数转换成ASCII码串
; 参数传递到堆栈:
;     (1) 32-位的浮点数
;     (2) ASCII码目的串的地址
; 生成的ASCII码串格式为[blank/-]d.dddddE[+/-]dd的格式
```

```
; (串长度为12个字符)
            push ebp                    ; 初始化堆栈
            mov  ebp, esp
            push eax                    ; 保存寄存器
            push ebx
            push ecx
            push edi

            fstcw controlWd             ; 取得控制字
            push controlWd              ; 保存控制字
            or   controlWd, 0000110000000000b
            fldcw controlWd             ; 设置控制字标号
            mov  edi, [ebp+8]           ; 目的串地址
            mov  eax, [ebp+12]          ; value转换
            mov  exponent, 0            ; exponent:= 0
            mov  value, eax             ; 通过存储器，将value压入ST
            fld  value
            ftst                        ; value >= 0?
            fstsw  ax                   ; 复制状态寄存器字到AX
            and  ax, C0                 ; 检查C0
            jnz  elseNeg                ; 如果value为负，转移
            mov  BYTE PTR [edi], ' '    ; 如果value为正，填入空格
            jmp  endifNeg
elseNeg:    mov  BYTE PTR [edi], '-'    ; value为负填入"-"号
            fchs                        ; 使数字成为正数
endifNeg:
            inc  edi                    ; 指向目的操作数的下一个字节

            mov  exponent, 0            ; exponent := 0
            ftst                        ; value = 0?
            fstsw ax                    ; 复制状态寄存器字到AX寄存器
            and  ax, C3                 ; 检查C3
            jne  endifZero              ; 如果为零，转移
            fcom ten                    ; value > 10?
            fstsw ax                    ; 复制状态寄存器字到AX寄存器
            and  ax, C3 or C2 or C0     ; 检测C3 = C2 = C0 = 0?
            jnz  elseLess               ; 如果value <=10，转移
untilLess:
            fdiv ten                    ; value := value/10
            inc  exponent               ; 指数加1
            fcom ten                    ; value < 10
            fstsw ax                    ; 复制状态寄存器字到AX
            and  ax, C0                 ; 检查C0
            jnz  untilLess              ; 继续执行，直到value < 10
            jmp  endifBigger            ; 转移
elseLess:
whileLess:
            fcom one                    ; value < 1
            fstsw ax                    ; 复制状态寄存器字到AX寄存器
            and  ax, C0                 ; 检查C0
            jz   endwhileLess           ; 如果不小于0，转移
            fmul ten                    ; value := 10 * value
            dec  exponent               ; 指数减1
            jmp  whileLess              ; 继续执行，直到 value < 1
```

```
endwhileLess:
endifBigger:
endifZero:

          fadd round                  ; 加上整数value
          fcom ten                    ; value > 10?
          fstsw ax                    ; 复制状态寄存器字到 AX寄存器
          and  ax, C3 or C2 or C0     ; C3 = C2 = C0 = 0?(value > 10?)
          jnz  endifOver              ; 如果不是，转移
          fdiv ten                    ; value := value/10
          inc  exponent               ; 指数加1
endifOver:

; 1.0<=value<10.0
          fist digit                  ; 保存整数部分
          mov  bx, digit              ; 复制整数到BX寄存器
          or   bx, 30h                ; 把数字转换成字符
          mov  BYTE PTR [edi], bl     ; 将字符存储到目的地址
          inc  edi                    ; 指向目的操作数下一个字节
          mov  BYTE PTR [edi], '.'    ; 小数点
          inc  edi                    ; 指向目的操作数下一个字节

          mov  ecx, 5                 ; 计算剩余数字
forDigit: fisub digit                 ; 减去整数部分
          fmul ten                    ; 乘以10
          fist digit                  ; 存储整数部分
          mov  bx, digit              ; 复制整数到BX寄存器
          or   bx, 30h                ; 将数字转换成字符
          mov  BYTE PTR [edi], bl     ; 将字符存储到目的地址
          inc  edi                    ; 指向目的操作数下一个字节
          loop forDigit               ; 重复执行5次

          mov  BYTE PTR [edi], 'E'    ; 指数指示器
          inc  edi                    ; 指向目的操作数下一个字节
          mov  ax, exponent           ; 取得指数
          cmp  ax, 0                  ; 指数 >= 0?
          jnge NegExp
          mov  BYTE PTR [edi], '+'    ; 非负指数
          jmp  endifNegExp
NegExp:   mov  BYTE PTR [edi], '-'    ; 负指数
          neg  ax                     ; 将指数转换成正数
endifNegExp:
          inc  edi                    ; 指向目的操作数下一个字节

          div  byteTen                ; 将指数转换成2进制数
          or   ax, 3030h              ; 将数字转换成ASCII码
          mov  BYTE PTR [edi+1], ah   ; 保存字符到目的地址
          mov  BYTE PTR [edi], al

          pop  controlWd              ; 恢复控制字
          fldcw controlWd
          pop  edi                    ; 恢复寄存器内容
          pop  ecx
          pop  ebx
          pop  eax
```

```
            pop  ebp
            ret  8
ftoaproc    ENDP
            END
```

程序通过名字引用控制位的指令开始。C3、C2和C0的控制位分别在第14、10和8位，并且都置“1”。

```
C3 EQU 0100000000000000b
C2 EQU 0000010000000000b
C0 EQU 0000000100000000b
```

在常规的过程入口语句后，FPU控制字被复制到存储器中，并且让它进栈。这样，在过程的结束后，它能被恢复，直到控制字的第10、11位用于控制取整。接下来的两条指令将它们设为11。这样，当一个浮点数存入整型存储器时，该数值的小数部分会舍去。

```
fstcw controlWd          ; 获取控制字
push controlWd           ; 保存控制字
or controlWd, 0000110000000000b
fldcw controlWd          ; 设置舍入控制字
```

该过程中的大部分代码都是直接实现设计的，很容易理解。不过，浮点型比较运算需要多做些解释。第一段是:

```
ftst              ; value > = 0?
fstsw ax          ; 复制状态字至AX
and ax, C0        ; 检测C0
jnz elseNeg       ; 如果设置，跳转（value负）
```

指令ftst比较value和0，在状态字中设置标志位。为了测试这些位，将这个状态字复制到AX中。仅当ST<0时，C0标志位被置“1”。除C0所对应的位外，and指令屏蔽了其余所有的位。如果剩余位是非0，表示value是负的，那么执行jnz指令。

判断是否“value>10”的程序和上述程序段类似，但更复杂，其代码如下:

```
fcom ten                  ; value>10?
fstsw ax                  ; 复制状态字至AX
and ax, C3 or C2 or C0    ; 测试是否C3 = C2 = C0 = 0
jnz elseLess              ; 如果value不大于10，跳转
```

如果ST>操作数，那么C3 = C2 = C0 = 0，三个控制位都为零。程序屏蔽C3、C2或C0，可写为0100010100000000。在汇编时而不是在执行时，or运算将操作数组合在一起。

这里用了一种新方法将指数转换成两个ASCII码字符。执行下列指令时，在AX中的指数是非负的，并且小于40。

```
div byteTen                  ; 将指数转换成两个数字
or ax, 3030h                 ; 将两个数字转换成ASCII码
mov BYTE PTR [edi + 1], ah   ; 将字符存储到目的地址
mov BYTE PTR [edi], al
```

将指数除以10，商（高位）放入AL，余数（低位）放入AH。由or指令将商和余数同时转换成ASCII码，并保存在目的字符串中。

编程练习10.2

1. 编写一个完整的程序，该程序可以根据提示输入十进制数表示的圆半径，然后计算并显示出（合适的标号）圆的周长和面积。要求使用宏指令input和output输入输出字符串，用过程atofproc和ftoaoproc实现浮点数和ASCII码之间的转换，用FPU指令实现浮点数运算。
2. 编写一个NEAR32的过程ftoaproc1，实现浮点数到固定小数点格式的ASCII码字符串的转换。要特别说明的是，这个过程必须将下列四个参数压入堆栈:
   - 一个32位浮点数值
   - 目的字符串的地址
   - 一个字，该字表示生成的字符串的字符总个数n
   - 一个字，该字表示生成的小数点后的数字的个数d

   输出的字符串包含一个起始位的负号或空格、占用n-d-2位（开始必须加入空格）的数值的整数部分、一个小数点，以及四舍五入后的数值存储了d位的小数部分。这个过程将保存所有寄存器，并从堆栈中取出参数。
3. 下列算法是求实数x的立方根。

```
root : = 1.0;
until (|root-oldRoot| < smallValue) loop
    oldRoot : = root;
    root : = (2.0*root + x/(root*root)) /3.0;
end until;
```

请编写一个NEAR32过程cuberoot，实现这个设计，使用0.001作为smallValue的值。

假设堆栈中有一个参数传递，是x的值。计算的结果返回到ST。该过程保存所有寄存器，并从堆栈中取出参数。

为该过程编写一个短的测试驱动程序，通过WinDbg来观察结果。

10.3 浮点数的模拟

一些80x86计算机系统没有浮点型单元，但这样的系统还是可以实现浮点数的运算。在这样的系统中，软件通过使用存储器和通用寄存器例程来实现浮点数运算，而不是用浮点型单元。这一节详述了实现浮点数的乘法和加法的过程，它们对于浮点数的模拟将是非常有用的，而且，它们还有助于更好地理解浮点数的表示。

本节中的程序按照IEEE单精度格式对浮点数进行了处理。回顾1.5节介绍的“二进制科学记数法”表示数:

- 一个完整的数的起始位为符号位，0表示正，1表示负。
- 一个8位偏移指数（或阶）。这是实际的指数加上127_{10}的偏移量。
- 23位小数（或尾数），起始位1被移去。

这就是REAL4指令的显示格式。

每个过程结合它的参数部分在结构fp3中生成结果。通常，这个结果不是标准化的。也就是说，没有精确到24位的小数位。NEAR过程normalize调整小数和指数位为标准格式。

注意：在使用标准IEEE格式表示0.0的时候存在一个问题，没有二进制小数点前以一位1开始的“二进制科学记数法”表示的零。最接近的表示方式是1.0×2^{-127}，它是一个很小的数，

但是它还不是零。根据先前给出的标准，这个值应该有一个32位0的IEEE表示。然而，以31个零为结束的两位模式被认为是特例，每个都被解释为0.0，而不是正的或者负的1.0×2^{-127}。下面的乘法和加法程序将考虑这些特例。

除了在表示0.0时有一个特别的位之外，IEEE标准还列出其他三种特殊的情况。如下面的模式:

```
s  11111111  00000000000000000000000
```

（符号位s，偏移指数255，小数0）代表正或负的无穷大，这些值可用来表示一个非零数被零除时的商；另一个特别情况被称为NaN（not a number，非数），它由偏移指数255和一个非零的小数的任意位模式来表示，例如，0/0的商应该是NaN；最后一个特殊情况是非标准化的数字，当偏移指数是零，而尾数是非零的时候，不再假定尾数的起始位为1，这样可以表示极小的数字。本节的浮点数程序在必要的地方注意特别的零的表示方法，但忽略了其他特别数的表示。

浮点数的符号、指数和尾数经常需要取出，因此，使用宏指令expand。这个宏指令有四个参数:

1. 一个32位浮点数
2. 一个字节，用来表示符号（0为正，1为负）
3. 一个字，用来表示非偏移（实际上的）指数
4. 一个双字，用来表示小数，包括非零数的起始位1

代码段10-7给出了宏指令expand的代码。

代码段10-7 宏指令expand

```
expand    MACRO  source, sign, exponent, fraction
LOCAL  addOne, endAddOne
; 取得32位的源浮点数并扩展成
; 单独的几个片断
;    符号位：字节
;    指数：字(非偏移)
;    小数：双字(起始位1)
        push eax                ; 保存EAX寄存器内容
        mov  eax, source        ; 取出源操作数
        rol  eax, 1             ; 符号位放到第0位
        mov  sign, 0            ; 符号位清0
        mov  sign, al           ; 得到有符号位的一个字节
        and  sign, 1            ; 覆盖除标志位的所有值
        rol  eax, 8             ; 指数移到0到7位
        mov  exponent, ax       ; 取出偏移指数的字
        and  exponent, 0ffh     ; 覆盖除了指数的所有值
        sub  exponent, 127      ; 减去偏移
        shr  eax, 9             ; 小数右移
        test eax, eax           ; 小数是否为零?
        jnz  addOne             ; 如果非零，起始位加1位
        cmp  exponent, -127     ; 初始指数是否为零?
```

```
            je    endAddOne          ; 如果为零，转移到endAddOne
AddOne:     or    eax, 800000h       ; 起始位置1
endAddOne:
            mov   fraction, eax      ; 保存小数部分
            pop   eax                ; 恢复EAX寄存器内容
            ENDM
```

扩展的宏指令代码阐述了位运算的用处。通过将符号位循环左移到第0位，该符号位被隔离，保存了包含它的字节，用1（= 00000001b）将除符号位之外的所有位都清0。然后，其他的八个指数位循环移位到EAX的右端，在起始位被屏蔽之前，将这八个指数位作为一个字保存起来。偏移量127被减去以获得正确的有符号的指数。最后，尾数被移回EAX的右边。在它被保存之前，要检查它是否是IEEE规定的0.0的表示。如果原始数据不是0.0，用or运算将科学记数法表示的起始位置为1。

如果浮点数可拆分为“符号-尾数-指数”的形式，那么，可通过处理各个部分来实现运算，然后将“符号-尾数-指数”三部分的结果组合在一起，得到浮点数表示的结果。这个将各个部分组合的运算由宏combine来实现，宏combine的代码如代码段10-8所示。

代码段10-8 宏combine

```
combine  MACRO  destination, sign, exponent, fraction
LOCAL  endZero
; 取出独立的部分：
;    符号：字节
;    指数：字(非偏移)
;    尾数：双字(起始位1)
; 取得浮点数把它们组合成一个32位
; IEEE的结果存入目的操作数
          push eax                  ; 保存EAX寄存器内容
          push ebx                  ; 保存EBX寄存器内容
          mov  eax, 0               ; 结果为零
          cmp  fraction, 0          ; 是否为零?
          je   endZero              ; 如果为零，转移
          mov  al, sign             ; 取出符号
          ror  eax, 1               ; 循环右移到符号位
          mov  bx, exponent         ; 取出指数
          add  bx, 127              ; 加偏移量
          shl  ebx, 23              ; 左移指数位
          or   eax, ebx             ; 和符号组合
          mov  ebx, fraction        ; 得到尾数
          and  ebx, 7fffffh         ; 起始位移动一位
          or   eax, ebx             ; 组合符号和指数
endZero:
          mov  destination, eax     ; 保存结果
          pop  ebx                  ; 恢复寄存器内容
          pop  eax
          ENDM
```

宏combine假定浮点数的每个部分的表示都是合法的，只考虑尾数为零的特殊情况。在这样的前提下，尾数将被标准化，即，第24位是1，并且最左边没有位为1。执行运算后，可能获得一个未标准化的结果，需要第三个宏对浮点数的表示进行标准化。代码见代码段10-9。它实现了下列设计：

```
if尾数为0 then停止;end if;
while尾数部分最左边不为0 loop;
    将尾数向右移动一位;
    指数加1;
end loop;
while 第23位不为1 loop
    将尾数向左移动一位;
    指数减1;
end loop;
```

代码段10-9 宏normalize

```
normalize  MACRO sign, exponent, fraction
LOCAL  endZero, while1, while2, endWhile1, endWhile2
; 描述浮点数的格式的几个部分:
; 符号: 字节
; 指数: 字(不带偏移)
; 尾数: 双字(起始位1)
          push eax                ; 保存EAX寄存器内容
          cmp  fraction, 0        ; 尾数是否为零?
          je   endZero            ; 如果为零, 转移
while1:   mov  eax, fraction      ; 复制尾数
          and  eax, 0ff000000h    ; 起始位非零?
          jz   endWhile1          ; 如果为零, 转移
          shr  fraction, 1        ; 指数右移
          inc  exponent           ; 指数加1
          jmp  while1             ; 重复操作
endWhile1:
while2:   mov  eax, fraction      ; 复制尾数
          and  eax, 800000h       ; 检测第23位
          jnz  endWhile2          ; 如果为1, 转移
          shl  fraction, 1        ; 尾数左移
          dec  exponent           ; 尾数减1
          jmp  while2             ; 重复操作
endWhile2:
endZero:
          pop  eax                ; 保存EAX寄存器内容
          ENDM
```

乘法是最容易实现的浮点数运算。它基于普通的乘法法则，参与运算的数值用科学记数法表示：

- 乘数的尾数部分相乘得到结果的尾数部分
- 乘数的指数部分相加得到结果的指数
- 根据符号惯例规则得到结果的符号

这些规则由代码段10-10所示的代码实现。过程fMultProc有三个进栈的参数：两个操作数和一

个存放结果的地址。符号由操作数符号的异或运算决定。指数的相加非常直观，容易理解。尾数部分的乘法通过对低23位的移位来实现：每一个尾数在逻辑上是一个1，跟在二进制小数点后，共有23位二进制的小数位。这样两个数相乘的结果将得到一个46位的尾数，多余的23位必须舍去。

代码段10-10 过程fMultProc

```
; 过程fMultProc (Operand1, Operand2 : float; Result : address of float)
; 参数以双字入栈
; 参数由过程取出
; 作者: R. Detmer
; 日期: 1998年4月

.DATA
sign1       BYTE  ?
exponent1   WORD  ?
fraction1   DWORD ?
sign2       BYTE  ?
exponent2   WORD  ?
fraction2   DWORD ?
sign3       BYTE  ?
exponent3   WORD  ?
fraction3   DWORD ?

.CODE
fMultProc   PROC NEAR32
            push    ebp                 ; 保存基地址指针
            mov     ebp,esp             ; 复制堆栈指针
            push    eax                 ; 保存寄存器内容
            push    edx

            expand  [ebp+16], sign1, exponent1, fraction1
            expand  [ebp+12], sign2, exponent2, fraction2
            mov     al, sign1           ; 组合符号位
            xor     al, sign2
            mov     sign3, al           ; 保存
            mov     ax, exponent1       ; 指数相加
            add     ax, exponent2
            mov     exponent3, ax       ; 保存
            mov     eax, fraction1      ; 尾数相乘
            mul     fraction2
            shrd    eax, edx, 23        ; 丢弃多余位
            mov     fraction3, eax      ; 保存

            normalize sign3, exponent3, fraction3
            mov     edx, [ebp+8]        ; 保存结果的地址
            combine [edx], sign3, exponent3, fraction3

            pop     edx                 ; 恢复寄存器的内容
            pop     eax
            pop     ebp                 ; 恢复基地址指针
            ret     12                  ; 返回，取出参数
fMultProc   ENDP
```

过程fMultProc调用的宏在以前的代码段中已列出。宏的使用很方便，但是要注意，这里也有一些危险。例如，不能够使用下列指令来组合各部分的结果:

```
mov     eax, [ebp + 8]      ; 结果的地址
combine [eax], sign3, exponent3, fraction3
```

因为组合的宏是在内部使用EAX寄存器，所以，expand、combine和normalize作为过程而不是宏来实现会更安全。

下面给出浮点数的加法的算法，它要比乘法复杂一些。但是，它和用两个科学记数法表示的数的加法有相同的过程，即，先调整两个数的指数，让它们指数部分相同，然后将它们的尾数相加。如果有一个是负数，这样的加法有些复杂，负数的尾数必须先取反，然后再让它和其他尾数相加。代码段10-11中的代码是实现了上述算法。

代码段10-11 过程fAddProc

```
; 过程fAddProc (Operand1, Operand2 : float; Result : address of float)
; 参数以双字入栈
; 参数由过程取出
; 作者: R.Detmer
; 日期: 1998年4月

.DATA
sign1       BYTE  ?
exponent1   WORD  ?
fraction1   DWORD ?
sign2       BYTE  ?
exponent2   WORD  ?
fraction2   DWORD ?
sign3       BYTE  ?
exponent3   WORD  ?
fraction3   DWORD ?

.CODE
fAddProc    PROC NEAR32
            push    ebp                 ; 保存基地址指针
            mov     ebp,esp             ; 复制堆栈指针
            push    eax                 ; 保存寄存器内容
            push    edx
            expand  [ebp+16], sign1, exponent1, fraction1
            expand  [ebp+12], sign2, exponent2, fraction2
            mov     ax, exponent1       ; 复制exponent1
while1:     cmp     ax, exponent2       ; exponent1 < exponent2?
            jnl     endWhile1           ; 如果不是，转移
            inc     ax                  ; 将exponent1加上1
            shr     fraction1,1         ; fraction1右移1位
            jmp     while1              ; 重复操作
endWhile1:  mov     exponent1, ax       ; 将exponent1存入存储器
            mov     ax, exponent2       ; 复制exponent2
while2:     cmp     ax, exponent1       ; exponent2< exponent1?
            jnl     endWhile2           ; 不是，则转移
            inc     ax                  ; 将exponent1加上1
            shr     fraction2,1         ; fraction2右移1位
            jmp     while2              ; 重复操作
```

```
endWhile2: mov     exponent2, ax   ; 将exponent2存入存储器
           mov     exponent3, ax   ; 保存公有指数
           cmp     sign1, 1        ; sign1为负?
           jne     notNeg1         ; 如果不是，转移
           neg     fraction1       ; fraction1取反
notNeg1:
           cmp     sign2, 1        ; sign1为负?
           jne     notNeg2         ; 如果不是，转移
           neg     fraction2       ; fraction2取反
notNeg2:
           mov     eax, fraction1  ; 尾数相加
           add     eax, fraction2
           mov     fraction3, eax  ; 尾数相加的和保存在fraction3
           mov     sign3, 0        ; sign3为正
           cmp     eax, 0          ; fraction3 < 0?
           jnl     notNegResult    ; 如果不是，转移
           mov     sign3, 1        ; sign3为负
           neg     fraction3       ; 将fraction3变成正数
notNegResult:
           normalize sign3, exponent3, fraction3
           mov     edx, [ebp+8]    ; 保存结果的地址
           combine [edx], sign3, exponent3, fraction3
           pop     edx             ; 恢复寄存器内容
           pop     eax
           pop     ebp             ; 恢复基地址指针
           ret     12              ; 返回，取出参数
fAddProc   ENDP
```

```
扩展每一个数的符号位、指数和尾数部分;
while exponent1 < exponent2 loop
    exponent1加1
    fraction1右移1位
end while;
while exponent2 < exponent1 loop
    exponent2加1
    fraction2右移1位
end while;
exponent3 := exponent1; {尾数相等}
if sign1为负 then fraction1取反;end if;
if sign2为负then fraction2取反;end if;
fraction3 := fraction1 + fraction2;
sign3为正;
if fraction3 < 0
then
    sign3为负;
    fraction3取反;
end if;
```

编程练习10.3

不使用浮点型指令，编写下列练习的程序。

1. 编写一个NEAR32过程fDivProc。它有三个参数：Operand1、Operand2和Result。每个操作数是一个32位的浮点数，Result是一个32位浮点数表示的地址。如果Operand2≠0.0，将Operand1/Operand2的值存入Result给出的地址中。如果第二个操作数是零，那么结果就用是正的或负的无穷大的IEEE表示（是正数还是负数由Operand1的符号决定）。该过程从堆栈中取出参数，并且不改变寄存器内容。
2. 编写一个NEAR32过程fSubProc。它有三个参数：Operand1、Operand2和Result。每个操作数是一个32位的浮点数，Result是一个32位浮点数表示的地址。将Operand1－Operand2的值存入Result给出的地址中。这个过程将从堆栈中取出参数，并且不改变寄存器内容。（可以调用fAddProc来实现这个功能，但是请编写一个完整的过程来替代它。）
3. 编写一个NEAR32过程fNegProc。它有两个参数：Operand和Result。Operand是一个32位的浮点数，Result是一个32位浮点数表示的浮点数表示的地址。将－Operand1的值存入Result给出的地址中。这个过程将从堆栈中取出参数，并不改变寄存器内容。
4. 编写一个NEAR32过程fCmpProc。它有两个参数：Operand1和Operand2。每个操作数是一个32位的浮点数。这个过程将Operand1和Operand2比较的结果返回到EAX中：如果Operand1＝Operand2，返回0；如果Opeand1＜Operand2，返回－1；如果Operand1＞Operand2，返回＋1。这个过程将从堆栈中取出参数，除EAX外，不改变其他寄存器内容。

10.4 浮点数和嵌入式汇编

高级语言编译器有时候可翻译包含*嵌入式汇编代码*（in-line assembly code）的程序，它允许程序的大部分代码用高级语言编写，而小部分代码用汇编语言编写，这些汇编语言部分主要用于需要临界优化的地方或者实现一些低级算法，它们可能很难或者不可能用高级语言来编码。

本节给出了一个简单程序例子，它是用Microsoft Visual C++编写的，实现了和代码段10-1中代码所做的相同的运算，就是求两个浮点数的平方和的平方根。但是，这个例子要求输入数据和输出结果，其中输入和输出语句用C++语言编写。代码见代码段10-12。

代码段10-12 嵌入汇编语言

```
// 求两个数的平方和的平方根
#include <iostream.h>
void main()
{
  float value1;
  float value2;
  float sum;

  cout << "First value? ";
  cin >> value1;
  cout << "Second value? ";
  cin >> value2;

  __asm
  {
    fld    value1
    fld    st
```

```
        fmul
        fld     value2
        fld     st
        fmul
        fadd
        fsqrt
        fstp  sum
      }
        cout << "The sum is " << sum << endl;
    }
```

注意：对于这个编译程序，嵌入式汇编代码是以开头是两个下划线的关键字__asm标志开始的，而且汇编语言部分要用大括号括起来。另外，汇编语言部分可以调用在C++语言部分中声明的变量。最后，虽然这些汇编语言都是浮点型指令，但基本上任何语句都可用在嵌入式汇编代码中，包括语句标号。

编程练习10.4

1. 编写一个可以根据提示输入圆半径（十进制小数），然后计算并显示（用合适的标号）出圆的周长和面积的完整程序。输入和输出用C++语言编写，浮点数的计算用浮点数嵌入汇编语言的指令来编写。
2. 下面是求实数x的立方根的算法：

```
root : = 1.0;
until (|root-oldRoot| < smallValue) loop
    oldRoot : = root;
    root : = (2.0*root + x/(root*root))/3.0;
end until;
```

用C++语言声明变量，输入x的值，输出显示立方根，用浮点数嵌入汇编语言来实现立方根的计算，用0.001作为smallValue的值。

本章小结

Intel 80x86浮点型单元（FPU）有八个80位数据的数据寄存器，它们共同组成了一个堆栈。这个堆栈可以执行各种载入、存储的指令、算术运算以及复杂的超越函数运算。比较指令可用来设置FPU状态寄存器中的位，这个状态字必须被复制到AX寄存器或者存储器，这样才能检查比较的结果。

浮点数和ASCII码之间的转换类似于前面讨论的ASCII码和整数的转换。最容易读的ASCII码格式是简单的十进制格式，而要生成最简单的ASCII码格式是E记数法。

在没有浮点型单元的情况下，浮点型指令可以被模拟。它的基本方法是首先将浮点数的表示分割成符号、指数和尾数三个部分。然后分别对这些部分进行处理，最后再将各部分的结果组合成浮点数的表示形式。

一些高级语言编译器可以翻译嵌入的汇编语言代码，其应用之一是和浮点型指令一起，用C++这样的高级语言实现输入输出，而用汇编语言实现计算。而且，嵌入式汇编语言对其他关键或难以实现的应用也很有用。

第11章　十进制数运算

1.5节简要介绍了整数的BCD码（binary coded decimal，BCD）表示。BCD码对于存储包含多位数字的整数非常有用，比如在财务记录中可能就要用到BCD码数。BCD码数与ASCII码之间的转换比二进制补码与ASCII码之间的转换要容易得多。但是，只有一小部分80x86指令可用于BCD码数的运算。

本章介绍BCD码的表示规则，以及使用BCD码数的80x86指令。本章还给出了BCD码数与相应的ASCII码之间转换的程序，以及一些实现BCD码算法的过程。

11.1　压缩的BCD码表示

BCD码编码方式分为两大类：压缩的和未压缩的。此外，根据所用字节数和数值的符号的表示方式还有一些形式。本节和11.2节讨论压缩的BCD码，11.3节将会讨论未压缩的BCD码。

压缩的BCD码表示每字节存储两个十进制数字，高四位存储一个数字，低四位存储另外一个数字。例如：01101001代表十进制的69，其中0110代表6，1001代表9。对于压缩的BCD码有一个容易混淆的情况，那就是这个表示和十六进制的69的表示是一样的。但是，如果将01101001看作一个BCD码数，那么它表示十进制数69；如果将它看作为一个有符号或者无符号的二进制整数的话，那么01101001表示的是十进制的105。由此可见，一个给定的位模式可能解释为不同的数字，甚至还可能是非数字。

如果用单字节表示压缩的BCD码，那么可以存储0到99的十进制数，这一点不是很有用，所以需要用几个字节来存储一个数字。有许多可能的表示方式，有些使用固定的字节数；有些长度是可变的，将长度合并为一个字段表示其中的一部分；用位模式来表示一个数时，常用一位或多位表示数的符号。

第10章提到的微软的宏汇编程序提供了一条DT指令，它用来定义一个十字节的压缩的十进制数。虽然其他表示方法同样有效，但本书主要使用这种表示方法。指令

```
DT 123456789
```

预留了初始值（十六进制）的十个字节的存储空间：

```
89 67 45 23 01 00 00 00 00 00
```

注意：字节的存储是逆向的：低位存在高位字节中，但是每个字节内，单个的十进制数的存储是正向的，这和二进制补码整数的高位和低位字节的存储是一致的。这种表示方式的第十个字节用来表示整个数的符号：如果是正数，那么第十个字节就是00；如果是负数，那么它就是80。因此，DT指令

```
DT -1469
```

执行的结果为：

```
69 14 00 00 00 00 00 00 00 80
```

注意，对负数而言，负数的表示只有符号位和正数的表示不同，其他位都相同。

因为要用一个完整的字节来表示符号，所以只剩下九个字节可以用来存放十进制数。因此，如果压缩的BCD码使用DT指令来存储一个带符号十进制数，那么这个数最长为18位。用MASM6.11汇编，大于18位的数字会被截去，而且不会给出任何警告。

虽然DT指令可用在汇编语言程序中初始化压缩的BCD码数据，并且下一节将介绍一些有助于BCD数的运算的指令。但是，压缩的BCD码还是很少使用，除非需要将它们显示出来。代码段11-1给出了过程*ptoaProc*的代码，该过程将一个压缩的BCD码数据转换为对应的ASCII码字符串。对压缩BCD码数，该过程实现了和过程*itoaProc*以及*dtoaProc*对二进制补码整数的同样的功能。

代码段11-1 压缩的BCD码转换成ASCII码

```
ptoaProc  PROC NEAR
; 10字节长的BCD码转换为19字节长的ASCII码字符串
; 参数1: BCD码的地址
; 参数2: 目的操作数地址
; 作者: R. Detmer      日期: 1998年5月
          push ebp                    ; 初始化堆栈
          mov  ebp, esp
          push esi                    ; 保存寄存器内容
          push edi
          push eax
          push ecx
          mov  esi, [ebp+12]          ; 源操作数地址
          mov  edi, [ebp+8]           ; 目的操作数地址
          add  edi, 18                ; 指向目地操作数的最后一个字节
          mov  ecx, 9                 ; 计算要处理字节数
for1:     mov  al, [esi]              ; 一个字节存放两个BCD码
          mov  ah, al                 ; 复制到AX寄存器的高位
          and  al, 00001111b          ; 屏蔽高位
          or   al, 30h                ; 转换为ASCII码字符
          mov  [edi], al              ; 保存低位字
          dec  edi                    ; 向左指向目的操作数的下一个字节
          shr  ah, 4                  ; 移出低位字
          or   ah, 30h                ; 转换为ASCII码
          mov  [edi], ah              ; 保存高位字
          dec  edi                    ; 向左目的操作数的下一个字节
          inc  esi                    ; 指向下一个字节
          loop for1                   ; 继续循环9次

          mov  BYTE PTR [edi], ' '    ; 为正数，则填入空格
          and  BYTE PTR [esi], 80h    ; 检验符号字节
          jz   nonNeg                 ; 非负，则跳转
          mov  BYTE PTR [edi], '-'    ; 负数符号
nonNeg:
          pop ecx                     ; 恢复寄存器内容
          pop eax
          pop esi
          pop edi
          pop ebp
          ret 8                       ; 返回，取出参数
ptoaProc  ENDP
```

过程*ptoaProc*有两个参数：一个10字节长的压缩的BCD码源数据，一个19字节长的ASCII码的目的字符串，每个都通过地址来传递。19个字节长的目的字符串包括表示符号的1个字节和表示数字的18个字节。用空格来表示正数，减号表示负数。在数字的表示中，起始位是用0来代替空格。此过程实现了下列的设计：

```
复制源操作数地址到ESI;
复制目的操作数地址到EDI;
EDI加18指向目的串最后一个字节;

for count := 9 直到1 loop {处理包含两个数字的字节}
    复制下一个源字节到AL;
    复制源字节到AH;
    屏蔽AL中的高位数;
    将AL中的低位数转换为ASCII码;
    将低位数的ASCII码保存到目的串;
    EDI减1，向左指向目的串的下一个字节;
    将AH右移4位，得到高位数;
    将AH中的高位数转换为ASCII码;
    将高位数的ASCII码保存到目的串;
    EDI减1，向左指向目的串的下一个字节;
    ESI加1，向右指向下一个源操作数;
end for;
移到目的串的第一个字节;
if源数是负的
then
    将负号放入目的串的第一个字节;
end if;
```

该设计和代码中最令人感兴趣的部分是将一个源字节分成两个目的字节。源字节复制了两份，分别存放在AL和AH中。AL中的字节被转换成了ASCII码的低位，用and指令屏蔽掉了左边的四位，用or指令将0011（十六进制的3）放入被屏蔽掉的左边四位的位置。然后用相同的方法产生高位。shr指令丢弃了AH中的低位，将高位的数字放入了右边的四位，左边四位为零。再用一条or指令将ASCII码放入高位。

一旦一个压缩的BCD码数转换为一个ASCII码字符串后，该BCD码数就可以使用宏指令output或者其他的方法来显示。由于在金融计算中常用到BCD码数据，因此，除了用过程*ptoaProc*将BCD数转换为ASCII码外，还需要一些其他的ASCII码表示，本节最后的练习给出了一些可供选择的方法。

有时，需要将ASCII码字符串转换成对应的压缩的BCD码值，代码段11-2所给出的过程*atopProc*可实现这一任务，但它对ASCII码有一定的限制。这个过程有两个参数：ASCII码的源字符串的地址和一个10字节长的BCD码的目的字符串的地址。ASCII码的源字符串受到限制：它只能由表示数字的以空字节结束的ASCII码组成，除此之外，不能有符号和空格，也不允许其他的任何字符。

代码段11-2 ASCII码转换为压缩的BCD码

```
atopProc  PROC NEAR32
; ASCII码字符串转换为10字节压缩的BCD码
; 参数1: ASCII码串地址    参数2: BCD码地址
; 只包含ASCII码表示的数字，以空字节结束源串
; 作者: R. Detmer     日期: 1998年5月
          push ebp                    ; 初始化堆栈
          mov  ebp, esp
          push esi                    ; 保存寄存器内容
          push edi
          push eax
          push ecx
          mov  esi, [ebp+12]          ; 源操作数地址
          mov  edi, [ebp+8]           ; 目的操作数地址
          mov  DWORD PTR [edi], 0     ; BCD码目的操作数置0
          mov  DWORD PTR [edi+4], 0
          mov  WORD PTR [edi+8], 0
; 查找源串的长度，并移动ESI到尾部的空位
          mov  ecx, 0                 ; count := 0
while1:   cmp  BYTE PTR [esi], 0      ; 没有到串末时（非空），循环
          jz   endwhile1
          inc  ecx                    ; count加1
          inc  esi                    ; 指向下一个字符
          jmp  while1                 ; 再次检查
endwhile1:
; 每次将字符配对处理
while2:   cmp  ecx, 0                 ; 当count> 0时
          jz   endwhile2
          dec  esi                    ; 从右端指向下一个ASCII码字节
          mov  al, BYTE PTR [esi]     ; 取字节
          and  al, 00001111b          ; 转换为BCD码
          mov  BYTE PTR [edi], al     ; 保存BCD码
          dec  ecx                    ; count减1
          jz   endwhile2              ; 如果没有源数字，则跳出循环
          dec  esi                    ; 从右端指向下一个ASCII字节
          mov  al, BYTE PTR [esi]     ; 取字节
          shl  al, 4                  ; 左移，并转换为数字
          or   BYTE PTR [edi], al     ; 和其他BCD码组合起来
          dec  ecx                    ; count减1
          inc  edi                    ; 指向下一个目的操作数字节
          jmp  while2                 ; 重复所有源字符
endwhile2:
          pop ecx                     ; 恢复寄存器内容
          pop eax
          pop esi
          pop edi
          pop ebp
          ret 8                       ; 返回，取出参数
atopProc  ENDP
```

过程atopProc的设计和atodProc（代码段8-2）的设计完全不同，后者是将ASCII码字符串转换成了一个双字长的整数。ASCII码到双字的转换程序从左到右扫描源字符，一次扫描一个字符。而ASCII码到压缩的BCD码的转换程序是从右到左扫描源字符，一次扫描两个字符，这是为了将两位的十进制数字压缩到一个字节中，这个过程必须从字符串的右端开始。如果源字符的个数是奇数的话，最后一个BCD码的字节中将只保存一个字符。atopProc的设计如下：

```
复制源操作数地址到ESI;
复制目的操作数地址到EDI;
将目的串的10个字节初始化为0;
计数器置0;
while ESI没有指向源ASCII串尾部的空字节时 loop
    counter加1;
    ESI加1指向源串的下一个字节;
end while;
while counter > 0 loop
    ESI减1, 从右端指向源串的下一个字节;
    将源字节复制到AL;
    将ASCII码转换为数字, 最左边四位置0;
    将低位数保存到目的串;
    counter减1;
    if counter = 0
    then
        停止循环;
    end if;
    ESI减1, 从右端指向源串的下一个字节;
    将源字节复制到AL;
    将AL左移4位, 得到高4位数;
    AL与目的串的低4位结合;
    counter减1;
    EDI加1, 指向下一个目的串字节;
end while;
```

设计中的第一个while循环实现从左到右扫描源字符串，计算前面所读到的阿拉伯数字，一直读到空字节为止。虽然这个设计只允许表示数字的ASCII码，但可以再编写一个循环，该循环跳过起始位的空格以及表示正负号的符号位。（这些以及其他要增加的功能将在程序练习中说明。）

第二个while循环对在第一个循环计算过的表示数字的ASCII码进行处理，要将有效的两个数字压缩到一个目的字节中。这个循环每次至少处理一个源字节，所以第一个字符放入AL中，将ASCII码转换为一个阿拉伯数字，并将它存储到目的字符串的地址中。（通过用源字符减去30_{16}，也可将ASCII码转换为数字。）如果源字符用完了，那么while循环也就结束了。否则，将第二个ASCII码字符放入到AL中，用左移指令将它转换成数字，并放入AL的左边四位。再用or指令将它和已经存放在目的字符串的存储器中的右边的数合并。

过程atopProc可用来转换宏指令input获得的字符串，如果是通过其他方式获得的字符串，要保证该字符串是以一个空字节来结尾的。

练习11.1

1. 写出下列DT指令执行后MASM产生的初始值。
   (a) `DT 123456`
   (b) `DT -123456`
   (c) `DT 345`
   (d) `DT -345`
   (e) `DT 102030405060708090`
   (f) `DT -102030405060708090`
2. 说明如何用浮点型指令将一个存储为二进制补码双字长整数的数转换成一个相等的10字节长的压缩的十进制数。如果是将压缩的BCD码转换成双字长整数，该如何处理？
3. 定义一个宏ptoa，该宏和9.5节中介绍的宏指令itoa类似。它有两个参数：dest和source。其中dest是一个19个字节长的ASCII码字符串地址，source是存储器中的一个10个字节长的压缩的BCD码字符串的地址。请考虑安全措施，确保一次调用中参数的个数是正确的，这个宏代码将调用*ptoaProc*。

编程练习11.1

1. 修改过程*ptoaProc*的代码，让它的起始位是空格而不是零。如果有负号的话，负号放在第一个非零数字的最左边。如果这个数是零，最右边一位的零不被替换为空格。该字符串的总长度是19个字符，这个过程将从堆栈中取出参数。
2. 修改过程*ptoaProc*的代码，让它生成一个可表示货币值的22个字节长的ASCII码字符串。在前16个字符中，起始位用空格来代替零（如果有的话）。第17个字符是固定的小数点。第18、19个字符是阿拉伯数字，即使它们的值是零。第20个字符是一个空格。如果源数值是正数，那么第21、22个字符是“CR”的ASCII码表示。如果是负数，那么第21、22个字符是“DB”的ASCII码表示。这个过程将从堆栈中取出参数。
3. 编写程序，实现如下过程：
   (a) 修改过程*atopProc*的代码，使得它可以跳过源字符串起始位的空格，允许第一个数字前可以有正负号（+或-），并且在扫描字符串时，遇到任何非数字（而不是只有一个空字节），就停止扫描。如果遇到一个负号，BCD码表示的符号字节设为80_{16}。这个过程将从堆栈中取出参数。
   (b) 请定义一个和9.5节中介绍的宏atoi相似的宏atop。它有两个参数：dest和source。其中dest是存储器中的一个10个字节长的压缩的BCD码字符串的地址，source是一个19个字节长的ASCII码字符串的地址。请考虑安全措施，确保一次调用的参数个数是正确的，这个宏的代码将调用在（a）修改的过程*atopProc*。
4. 编写一个过程*editProc*，它有两个参数：（1）一个模板字符串的地址；（2）一个10个字节长的压缩的BCD码数的地址。这个过程有选择地将模板字符串中的一些字符替换成空格，或者替换成从BCD码中提取的ASCII码表示的数字。除了一个作为结束的空字节外，模板字符串还允许有下列字符：井号（#）、逗号（,）、小数点（.）。小数点通常不能替换。每一个#号可替换为一个阿拉伯数字。最多可以有18个#字符。如果少于18的话，那么只使用BCD码的低位。结果字符串起始位的零全部替换为空格，但是，如果零是跟在是小数点后

面的话，那么这些零就要保留。逗号不需要改变，除非它和空格相邻，这时，逗号要用空格替换。下面的例子（b表示空格）说明了editProc是如何替换的。请注意，原来的模板字符串被过程改变了，这个过程将从堆栈中取出参数。

用模板前	BCD值	用模板后
##, ###.##	123456	bl, 234.56
##, ###.##	12345	bbb123.45
##, ###.##	1	bbbbbb.01

11.2 压缩的BCD码指令

压缩的BCD码数的加法和减法运算和多位数的二进制数的加法和减法相似（见4.5节）。两个操作数的相对应的字节相加，而进位加法是加到下一个字节。BCD码操所数没有专门的加法指令，通常使用一般的指令add和adc。但是，这些指令是为二进制数设计的，而不是针对BCD码数的，所以对于多操作数相加，其结果可能是错误的。

80x86体系结构中有一条daa（decimal adjust after addition）指令，在执行加法指令后，daa指令可以对和进行调整。本节讨论了daa指令以及和它对应的减法das指令，并给出了可用于非负的压缩的BCD码数的加法和减法的过程，还给出了一个通用的加法程序。

下面举例说明对BCD码操作数使用二进制加法会产生的问题。在AF列，给出了辅助进位标志的值，其作用将在后面讨论。

执 行 前		执行add al, bl后		
AL	BL	AL	AF	CF
34	25	59	0	0
37	25	5C	0	0
93	25	B8	0	0
28	39	61	1	0
79	99	12	1	1

如果是两个无符号的二进制整数相加的和，那么上述每个答案都是正确的。但是，如果是BCD码数的和，那么只有第一个答案是正确的。第二和第三个答案中有BCD码表示中不用的位模式：第二个例子中的C_{16}和第三个例子中的B_{16}。虽然最后两个和没有不合法的阿拉伯数字，但是作为十进制数的和，显然它们是不正确的。

指令daa用于一条加法指令之后，它的功能是将一个二进制的和转换成一个压缩的BCD码的和，该指令没有操作数，需要转换的和必须存放在AL寄存器中。一条daa指令测试并设置进位标志位CF和辅助进位标志位AF（EFLAGS寄存器的第四位）为“1”。回顾一下，两个八位数相加，如果最左位产生进位，那么进位标志位CF置为1。同样，执行add或adc指令时，如果两个操作数的低四位相加产生进位，那么AF置为1，也就是说，两个低位的十六进制数的和大于F_{16}。

指令daa先测试AL中的二进制和的右边的第一个十六进制数，如果这个数超过9（也就是A到F），那么，在整个和加6，并且AF置为1。注意：在上述第二个例子中，通过：5C + 6 = 62来调整结果，62是压缩的BCD码37加25的和。在执行指令daa时，如果AF = 1，那么可以用同样方法调整结果。因此，在第四个例子中，61 + 6 = 67。

在调整了右边的数字后，指令daa测试AL中左边的数字。操作过程很相似：如果左边的数字超过9或者CF = 1，那么在整个和上加上60_{16}。如果调整和的话，进位标志位CF置为1。在第三个例子中，B8 + 60 = 18，进了一位。

在最后一个例子中，两个数都要调整：12 + 6 = 18，18 + 60 = 78（因为CF = 1）。下表完善了上述例子，假设执行下列两条指令：

```
add al, bl
daa
```

执行前	执行add后	执行daa后
AL: 34	AL: 59	AL: 59
BL: 25	AF: 0 CF: 0	AF: 0 CF: 0
AL: 37	AL: 5C	AL: 62
BL: 25	AF: 0 CF: 0	AF: 1 CF: 0
AL: 93	AL: B8	AL: 18
BL: 25	AF: 0 CF: 0	AF: 0 CF: 1
AL: 28	AL: 61	AL: 67
BL: 39	AF: 1 CF: 0	AF: 1 CF: 0
AL: 79	AL: 12	AL: 78
BL: 99	AF: 1 CF: 1	AF: 1 CF: 1

指令das（decimal adjust after subtraction）用在一条sub或sbb指令之后。它的执行和指令daa差不多，但是要从AL中的值中减去6或60_{16}，而不是加上。下面举例说明在执行指令sub al, bl之后，指令das是如何执行的。在第一个例子中，CF和AF都被置为1，因为减法中两个数位都要求借位。BC将减去6和60_{16}，结果是56，并且CF和AF还是1。25 − 69 = 56（借位使得25成为125），这个结果是正确的。

执行前	执行add后	执行daa后
AL: 25	AL: BC	AL: 56
BL: 69	AF: 1 CF: 1	AF: 1 CF: 1
AL: 37	AL: 12	AL: 12
BL: 25	AF: 0 CF: 0	AF: 0 CF: 0
AL: 93	AL: 6E	AL: 68
BL: 25	AF: 1 CF: 0	AF: 1 CF: 1
AL: 92	AL: 59	AL: 53
BL: 39	AF: 1 CF: 0	AF: 1 CF: 0
AL: 79	AL: E4	AL: 84
BL: 95	AF: 0 CF: 1	AF: 0 CF: 1

每条daa和das指令都是用单字节编码。指令daa的操作码是27，指令das的操作码是2F。对于Pentium处理器，每条指令的执行需要三个时钟周期。为了修正AL中的最后值，除了要修改AF和CF之外，还要通过daa或者das指令对SF、ZF和PF标志位置“1”或置“0”，溢出标志位OF没有定义，其他标志位不受影响。

本节的第一个BCD码运算过程是将两个非负的10个字节长的数相加。该过程有两个参数：目的数地址和源数的地址。每个都作为一个操作数，计算的和将存入目的操作数，这和普通加法指令使用目的操作数是一致的，这里不再考虑标志位的设置。本小节的练习中给出了较

全面地对SF、ZF和CF赋值的程序段。下面是该过程的设计，用过程addBcd1（见代码段11-3）实现。

```
指向源操作数和目的操作数的第1个字节;
for count := 1 to 9 loop
    将目的操作数字节复制到AL;
    将源操作数字节和AL相加，结果存入AL;
    将相加的和调整为BCD码;
    保存AL到目的操作数;
    指向下一个源操作数和目的操作数字节;
end for;
```

代码段11-3 无符号压缩的BCD码加法

```
addBcd1  PROC NEAR32
; 两个无符号压缩BCD码相加
; 参数1: 操作数1的地址 (目的操作数)
; 参数2: 操作数2的地址
; 作者: R. Detmer        日期: 1998年5月
          push ebp                ; 初始化堆栈
          mov  ebp, esp
          push esi                ; 保存寄存器内容
          push edi
          push ecx
          push eax
          mov  edi, [ebp+12]      ; 目的操作数地址
          mov  esi, [ebp+8]       ; 源操作数地址

          clc                     ; 进位标志清零
          mov  ecx, 9             ; 计算要处理字节数
forAdd:   mov  al, [edi]          ; 取一个操作数字节
          adc  al, [esi]          ; 与另一个操作数字节相加
          daa                     ; 调整为BCD码
          mov  [edi], al          ; 保存和
          inc  edi                ; 指向下一个操作数字节
          inc  esi
          loop forAdd             ; 所有9个字节重复执行

          pop  eax                ; 恢复寄存器内容
          pop  ecx
          pop  edi
          pop  esi
          pop  ebp
          ret  8                  ; 返回调用

addBcd1   ENDP
```

十字节长的压缩的BCD码的减法过程比较难，操作数要求是非负数。从目的数（地址在参数1中）中减去源数（地址在参数2中），如果源数比目的数大，减法的结果是一个负数。设计如下:

```
指向源操作数和目的操作数的第1个字节;
for count := 1 to 9 loop
```

```
        将目的操作数字节复制到AL;
        AL减去源操作数字节，结果存入AL;
        将相减的差调整为BCD码;
        保存AL到目的串;
        指向下一个源操作数和目的操作数字节;
    end for;

    if源操作数>目的操作数
    then
        指向目的操作数第1个字节;
        for count :=1 to 9 loop
            将0放入寄存器 AL;
            AL减去源操作数字节，结果存入AL;
            将相减的差调整为BCD码;
            保存AL到目的串;
            DI加1;
        end for;
        将符号字节80移到目的串;
    end if;
```

该设计的第一部分几乎和加法的设计一样。在第一个循环之后，如果进位标志位被设置为“1”，那么源数大于目的数为真，通过用零减去结果来调整不一致的地方。如果不调整的话，那么3－7的结果将是999999999999999996，而不是－4。代码段11-4中的过程subBcd1实现了这个设计。

代码段11-4 非负压缩的BCD码数的减法

```
subBcd1    PROC NEAR32
; 两个非负压缩BCD码数的减法
; 参数1: 操作数1的地址（目的操作数）
; 参数2: 操作数2的地址
; 操作数1 减操作数2的差值保存到目的操作数
; 作者: R. Detmer        日期: 1998年5月
          push ebp                ; 初始化堆栈
          mov  ebp, esp
          push esi                ; 保存寄存器内容
          push edi
          push ecx
          push eax
          mov  edi, [ebp+12]      ; 目的操作数地址（操作数1）
          mov  esi, [ebp+8]       ; 源操作数地址（操作数2）
          clc                     ; 进位标志清零
          mov  ecx, 9             ; 计算要处理字节数
forSub:   mov  al, [edi]          ; 取一个操作数字节
          sbb  al, [esi]          ; 减去另一个操作数字节
          das                     ; 调整为BCD码
          mov  [edi], al          ; 保存相减的差
          inc  edi                ; 指向下一个操作数字节
          inc  esi
          loop forSub             ; 重复执行9个字节

          jnc  endIfBigger        ; 如果目的操作数>=源操作数则完成
```

```
            sub  edi, 9                    ; 指向目地操作数的开始端
            mov  ecx, 9                    ; 计算要处理字节数
forSub1:    mov  al, 0                     ; 目的操作数减去0
            sbb  al, [edi]
            das
            mov  [edi], al
            inc  edi                       ; 下一个字节
            loop forSub1
            mov  BYTE PTR [edi], 80h       ; 结果为负
endIfBigger:
            pop  eax                       ; 恢复寄存器内容
            pop  ecx
            pop  edi
            pop  esi
            pop  ebp
            ret  8                         ; 返回调用
subBcd1     ENDP
```

如果有了兼有非负操作数的过程*addBcd*1和*subBcd*1，构造常规的压缩的BCD码的加法和减法过程就不是很难了。加法的设计如下:

```
if operand1 > 0
then
    if operand2 > 0
    then
        addBcd1(operand1, operand2);
    else
        subBcd1(operand1, operand2);
    end if;
else {operand1 < 0}
    if (operand2 < 0)
    then
        addBcd1(operand1, operand2);
    else
        改变operand1的符号字节;
        subBcd1(operand1, operand2);
        改变operand1的符号字节;
    end if;
end if
```

对负数的operand1的设计有些难，当operand2也是负数的时候，结果将是负数。因为*addBcd*1不影响目的数（operand1）的符号字节，和operand2相加，如果不要求特别的调整，那么结果应该是负数。如果和非负的operand2相加，结果可能是正数，也可能是负数。希望读者能够验证这个设计，并且根据代码调整结果的符号。过程*addBcd*实现了这个设计，如代码段11-5所示。常规的减法过程留作练习。

代码段11-5 通用的加法过程

```
addBcd  PROC NEAR32
; 两个任意压缩的BCD码数相加
; 参数1: 操作数1的地址（目的操作数）
; 参数2: 操作数2的地址
; 作者: R. Detmer            日期: 1998年5月
          push ebp                          ; 初始化堆栈
          mov  ebp, esp
          push esi                          ; 保存寄存器内容
          push edi
          mov  edi, [ebp+12]                ; 目的操作数地址
          mov  esi, [ebp+8]                 ; 源操作数地址
          push edi                          ; 准备下一次调用参数1
          push esi                          ; 准备下一次调用参数2
          cmp  BYTE PTR [edi+9], 80h        ; 操作数1 >= 0?
          je   op1Neg
          cmp  BYTE PTR [esi+9], 80h        ; 操作数2 >= 0?
          je   op2Neg
          call addBcd1                      ; 调用BCD码加法 (>= 0, >= 0)
          jmp  endIfOp2Pos
op2Neg:   call subBcd1                      ; 调用BCD码减法 (>= 0, < 0)
endIfOp2Pos:
          jmp  endIfOp1Pos                  ; 完成
op1Neg:   cmp  BYTE PTR [esi+9], 80h        ; 操作数2 < 0?
          jne  op2Pos
          call addBcd1                      ; 调用BCD码加法(< 0, < 0)
          jmp  endIfOp2Neg
op2Pos:   xor  BYTE PTR [edi+9], 80h        ; 替换符号字节
          call subBcd1                      ; 调用BCD码减法(< 0, >= 0)
          xor  BYTE PTR [edi+9], 80h        ; 替换符号字节
endIfOp2Neg:
endIfOp1Pos:
          pop  edi                          ; 恢复寄存器内容
          pop  esi
          pop  ebp
          ret  8                            ; 返回调用

addBcd    ENDP
```

练习11.2

1. 下列每部分中，假设执行指令

```
add al, bl;
daa;
```

给出AL寄存器中的数值、进位标志位CF和辅助标志位AF的值。

(a) AL: 35 BL: 42

(b) AL: 27 BL: 61

(c) AL: 35 BL: 48
(d) AL: 47 BL: 61
(e) AL: 35 BL: 92
(f) AL: 27 BL: 69
(g) AL: 75 BL: 46
(h) AL: 00 BL: 61
(i) AL: 85 BL: 82
(j) AL: 89 BL: 98
(k) AL: 76 BL: 89
(l) AL: 27 BL: 00

2. 在和练习1相同的条件下执行指令

```
sub al, bl
das
```

请给出AL寄存器中的数值、进位标志位CF和辅助标志位AF的值。

编程练习11.2

1. 修改过程*addBcd*，设置SF、ZF和CF为“1”。根据和的符号来决定符号位的设置。如果结果是零，设置ZF为1。如果和多于18个数字，设置进位标志位CF为“1”。
2. 设计并编写一个常规减法过程*subBcd*。它有两个参数：（1）operand1的地址，（2）operand2的地址。将operand1 – operand2的差存储在operand1的地址中，这个过程将从堆栈中取出参数。

11.3 未压缩的BCD码表示和指令

未压缩的BCD码不同于压缩的BCD码，它的每个字节存储一个数字而不是两个，每个字节的左半个字节的位模式是0000。本节介绍未压缩的BCD码的定义，并讨论未压缩的BCD码与ASCII码之间的相互转换，以及如何使用80x86指令对未压缩的BCD码数的进行算术运算。

未压缩的BCD码表示没有标准的长度。本书中，每个值存储为八个字节长，高位在左边，低位在右边（和通过DT指令存储压缩的BCD码的方式相反）。没有表示符号的字节，因此，只能表示非负数。一条普通的指令BYTE可以用来初始化未压缩的BCD码。例如，语句

```
BYTE  0, 0, 0, 5, 4, 3, 2, 8
```

预留了八个字节的存储空间，包括：00 00 00 05 04 03 02 08。是54328的未压缩的BCD码表示。指令

```
BYTE  8 DUP (?)
```

建立了一个八字节长的区域，可用来存储一个未压缩的BCD码数据。

要将未压缩的BCD码转换为ASCII码很简单。假设一个程序的数据段包括如下指令：

```
ascii      DB    8  DUP  (?)
unpacked   DB    8  DUP  (?)
```

如果unpacked中已经存放了一个未压缩的BCD码数，下列代码段运行后将产生与它一致的ASCII码表示，存放在ascii中。

```
        lea  edi, ascii          ; 目的
        lea  esi, unpacked       ; 源
        mov  ecx, 8              ; 计数次数
for8:   mov  al, [esi]           ; 得到位
        or   al, 30h             ; 转换成ASCII
        mov  [edi], al           ; 保存ASCII码字符
        inc  edi                 ; 增量指针
        inc  esi
        loop for8                ; 对所有字节重复操作
```

将一个ASCII码字符串转换成未压缩的BCD码表示也很简单。可以用相同的循环将EDI和ESI颠倒，不用or指令，而是用and指令，屏蔽高四位。

```
and al, 0fh   ; 将ASCII码转换成未压缩的BCD码
```

如果“在原来的位置”进行ASCII码和未压缩的BCD码的之间的转换将会更容易（见练习3）。

80x86结构体系有四条用于未压缩的BCD码数的算术运算的指令，每个助记符以“aa”起始代表“调整ASCII码”。Intel使用ASCII码字来描述未压缩的BCD码表示，尽管用ASCII码来表示一个数字时，左半字节为0011；用未压缩的BCD码表示时，左半字节为0000。这四条指令分别是：aaa、aas、aam和aad，有关这些指令的信息见表11-1。

表11-1 未压缩的BCD码的指令

指令	助记符	字节数	操作码	时钟周期（pentium）
加法后，和调整为ASCII码	aaa	1	37	3
减法后，差调整为ASCII码	aas	1	3F	3
乘法后，积调整为ASCII码	aam	2	D4 0A	18
除法后，商调整为ASCII码	aad	2	D5 0A	10

指令aaa与aas和对应的压缩的BCD码指令daa和das很相似。对于加法而言，使用一条add或adc指令来组合未压缩的BCD码的字节，并将和的结果放在AL寄存器中。如果需要的话，aaa指令可调整AL中的结果。指令aaa设置标志位，并且影响AH的内容。回想一下，daa指令只影响AL和标志位。下列算法描述了aaa是如何工作的。

```
if(AL中的低位数 > 9) 或 (AF=1)
then
    AL中的数与6相加，并存入AL;
    AH加1;
    AF := 1;
end if;

CF := AF;
AL中的高位数置0;
```

指令aas与此类似。If中的前两个运算由下列指令来代替。

```
subtract 6 from AL;
decrement AH;
```

指令aaa和aas未定义OF、PF、SF和ZF标志位。

下面举例说明add和aaa是如何一起使用的。在每个例子中，假设执行下列指令:

```
add al, ch
aaa
```

执行前	执行add后	执行aaa后
AX: 00 04	AX: 00 07	AX: 00 07
CH: 03	AF: 0	AF: 0 CF: 0
AX: 00 04	AX: 00 0B	AX: 01 01
CH: 07	AF: 0	AF: 1 CF: 1
AX: 00 08	AX: 00 11	AX: 01 07
CH: 09	AF: 1	AF: 1 CF: 1
AX: 05 05	AX: 05 0C	AX: 06 02
CH: 07	AF: 0	AF: 1 CF: 1

再举一组例子说明sub和aas是如何区分不同的单字节未压缩的BCD码操作数的。假设执行下列指令:

```
sub al, dl
aas
```

执行前	执行sub后	执行aas后
AX: 00 08	AX: 00 05	AX: 00 05
DL: 03	AF: 0	AF: 0 CF: 0
AX: 00 03	AX: 00 FC	AX: FF 06
DL: 07	AF: 1	AF: 1 CF: 1
AX: 05 02	AX: 05 F9	AX: 04 03
DL: 09	AF: 1	AF: 1 CF: 1

代码段11-6列出了过程addUnp，该过程将两个八字节未压缩的BCD码数相加，这两个数的地址作为参数传递。该过程与代码段11-3列出的过程addBcd1相似，但比它更简单。这里不需要设置重要的标志位，因为低位数字存储在右边，所以字节的处理是从右到左的。(编程练习1给出了相应的减法过程。)

代码段11-6 两个八字节长的未压缩的BCD码数的加法

```
addUnp   PROC NEAR32
; 两个八字节长未压缩的BCD码数相加
; 参数1: 操作数1和目的操作数的地址
; 参数2: 操作数2的地址
; 作者: R. Detmer        日期: 1998年5月
        push ebp            ; 初始化堆栈
        mov  ebp, esp
        push esi            ; 保存寄存器内容
        push edi
        push eax
        push ecx
        mov  edi, [ebp+12]  ; 目的操作数地址
        mov  esi, [ebp+8]   ; 源操作数地址
        add  esi, 8         ; 指向源操作数后的字节
        add  edi, 8         ; 目的操作数后的字节
```

```
            clc                 ; 进位标志清零
            mov  ecx, 8         ; 计算要处理字节数
forAdd:     dec  edi            ; 向左指向操作数
            dec  esi
            mov  al, [edi]      ; 取一个操作数字节
            adc  al, [esi]      ; 与另一个操作数字节相加
            aaa                 ; 调整为未压缩的BCD码
            mov  [edi], al      ; 保存和
            loop forAdd         ; 所有8个字节重复操作
            pop  ecx            ; 恢复寄存器内容
            pop  eax
            pop  edi
            pop  esi
            pop  ebp
            ret  8              ; 返回，释放参数
addUnp      ENDP
```

过程addUnp有一个值得注意的特点，它将给出正确的八字节的ASCII码数（不是未压缩的BCD码数）相加的和，这个和是用未压缩的BCD码表示的，Intel在未压缩的BCD码助记符中使用ASCII码是有道理的。该过程也可以作用于ASCII码字符串，因为aaa指令仅依赖于add对低位数字的操作，而且aaa将AL中的高位数设置为0。但是，即使操作数是真正的ASCII码字符串，那么相加的和也不是ASCII码，而是未压缩的BCD码。

两个单字节的未压缩的BCD码操作数的乘法使用一个普通的mul指令，相乘得到的结果放在AX寄存器中。当然，这个结果应该是一个调整了的二进制数，而不是通常的BCD码数。指令aam可以实现将AX中的乘积结果转换为两个未压缩的BCD码数字，分别放在AH和AL中的。实际上，指令aam将AL中的数除以10，将商放入AH，将余数放入AL中。下面举例说明这两条指令，假设执行指令

```
mul     bh
aam
```

执行前	执行mul后	执行aam后
AX: 00 09	AX: 00 51	AX: 08 01
BH: 06		
AX: 00 05	AX: 00 1E	AX: 03 00
BH: 06		
AX: 00 06	AX: 02 2A	AX: 04 02
BH: 07		

有些标志位受到了指令aam的影响，标志位PF、SF和ZF的值取决于AX中最后的值，标志位AF、CF和OF未定义。

一个数字的乘法不是很有用。代码段11-7所给的过程mulUnp1是一个八字节的未压缩的BCD码数和一个只有一位数字的未压缩的BCD码数的乘法运算。该过程有三个参数：1）目的地址，2）BCD码源数据的地址，3）包含了一数字的未压缩的BCD码数的一个字，其中BCD码是这个字的低字节。

代码段11-7 未压缩的BCD码数的乘法

```
mulUnpl  PROC NEAR32
; 一个八字节的未压缩的BCD码数和一个只有一字节的未压缩的BCD码数的乘法
; 参数1: 目的操作数地址
; 参数2: 8字节未压缩BCD码的地址
; 参数3: 包含了一位数的未压缩的BCD码数的一个字，其中BCD码是这个字的低字节
         push ebp              ; 初始化堆栈
         mov  ebp, esp
         push esi              ; 保存寄存器内容
         push edi
         push eax
         push ebx
         push ecx
         mov  edi, [ebp+14]    ; 目的操作数地址
         mov  esi, [ebp+10]    ; 源操作数地址
         mov  bx, [ebp+8]      ; 乘数
         add  esi, 8           ; 指向源操作数后的字节
         add  edi, 8           ; 目的操作数后的字节
         mov  bh, 0            ; lastCarry :=0
         mov  ecx, 8           ; 计算要处理字节数
forMul:  dec  esi              ; 指向操作数字节的左端
         dec  edi              ; 指向目的操作数字节的左端
         mov  al, [esi]        ; 数字由八位数
         mul  bl               ; 乘以单个字节
         aam                   ; 调整为未压缩的BCD码
         add  al, bh           ; 加上lastCarry
         aaa                   ; 调整为未压缩的BCD码
         mov  [edi], al        ; 保存单位数
         mov  bh, ah           ; 保存lastCarry
         loop forMul           ; 所有8个字节重复操作
         pop  ecx              ; 恢复寄存器内容
         pop  ebx
         pop  eax
         pop  edi
         pop  esi
         pop  ebp
         ret  10               ; 返回，释放参数
mulUnpl  ENDP
```

该算法的实现本质上和小学生所用的乘法一样。这个只有一位的BCD码数不断地和多位数的低位相乘，保存部分积的个位数，将部分积的十位数作为进位和高位的乘积相加。在循环开始前，先将所有的八位乘积通过初始化变量lastCarry都置为零。以下就是真正实现的设计:

```
{X7X6X5X4X3X2X1X0乘以Y得到Z7Z6Z5Z4Z3Z2Z1Z0}
lastCarry := 0;
for i := 0 to 7 loop
    xi乘以Y;
    add lastCarry;
    Zi:= units digit;
```

```
    lastCarry := tens digit;
end for;
```

在mulUnp1代码中，lastCarry值存储在BH寄存器中。当八字节的未压缩的BCD码中的一位数字和BL中的一位数相乘后，乘积的结果转换成未压缩的BCD码，并加上lastCarry，然后，必须将这个和转换成未压缩的BCD码。

指令aad基本上和指令aam的作用是互逆的，通过将AH中的数乘以10，再加上AL中的数，指令aad将一个存放在AH和AL中的两位数的未压缩的BCD码值组合成一个二进制数，并存放在AX中。寄存器AH被清除为00，标志位PF、SF和ZF的值根据结果而定，AF、CF和OF未定义。

在指令div执行前，执行指令aad，这和那些在算术指令执行之后，才执行的ASCII码调整指令的顺序相反。假设执行下列指令:

```
aad
div   dh
```

执 行 前	执行aad后	执行div后
AX: 07 05	AX: 00 4B	AX: 03 09
DH: 08	DH: 08	
AX: 06 02	AX: 00 3E	AX: 02 0F
DH: 04	DH: 04	
AX: 09 03	AX: 00 5D	AX: 01 2E
DH: 02	DH: 02	

在第一个例子里，执行完div指令后，商和余数以BCD码的形式分别存放在AL和AH中。但是第二和第三个例子表明并不是所有的情况都是这样。AH中的余数始终是正确的，因为除以一个小于等于9的数之后，余数是一个二进制数，并且，这个余数肯定在0～8之间，而这个区间内的数的二进制表示和未压缩的BCD码表示是一致的。AL中的商显然也是一个二进制数，而不是BCD码，为了将它转换成未压缩的BCD码，需要在div指令后面加一条aam指令。在第二个例子中，AX中的值应该是62÷4的商: 01 05。在第三个例子中，aam将AX中的值变为正确结果04 06。值得注意的是，除法的余数丢失了。因此，如果需要的话，在执行aam指令前，应该先将余数复制到AH中。

注意: 如果AH中的初始数字比DH中的除数小，那么上述例子中提到的问题就不会出现。简单的一个一位数去除一个多位数的算法是: 从被除数的左边除到右边，用除数来除一个两位数，这个两位数的第一位是上一次除法的余数，它一定小于除数。下列设计实现了该简单算法。

```
{X7X6X5X4X3X2X1X0除以Y得到Z7Z6Z5Z4Z3Z2Z1Z0}
lastRemainder : = 0;
for i : = 7 downto 0 loop
    dividend : = 10*lastRemainder + Xi;
    dividend除以Y, 得到quotient和lastRemainder;
    Zi: = quotient;
end for;
```

代码段11-8给出了实现这一设计的代码。寄存器AH很适合存储lastRemainder，因为一个16位二进制数除以一个8位数的最后的余数就在这里。

代码段11-8 未压缩的BCD码的除法

```
divUnpl   PROC  NEAR32
; 参数1: 目的操作数地址
; 参数2: 8字节未压缩BCD码数的地址
; 参数3: 一位数的BCD码作为低位字节
; 作者: R. Detmer                    日期: 1998年5月
          push ebp                  ; 初始化堆栈
          mov  ebp, esp
          push esi                  ; 保存寄存器内容
          push edi
          push eax
          push ebx
          push ecx
          mov  edi, [ebp+14]        ; 目的操作数地址
          mov  esi, [ebp+10]        ; 源操作数地址
          mov  bx, [ebp+8]          ; 除数
          mov  ah, 0                ; lastRemainder := 0
          mov  ecx, 8               ; 计算要处理字节数
forDiv:   mov  al, [esi]            ; 八位被除数
          aad                       ; 调整为二进制
          div  bl                   ; 用单个字节除
          mov  [edi], al            ; 保存商
          inc  esi                  ; 指向被除数的下一个字
          inc  edi                  ; 指向下一个目的操作数字节
          loop forDiv               ; 重复所有8个字节的操作
          pop  ecx                  ; 恢复寄存器内容
          pop  ebx
          pop  eax
          pop  edi
          pop  esi
          pop  ebp
          ret  10                   ; 返回，释放参数
divUnpl   ENDP
```

练习11.3

1. 下列每小题中，假设执行指令

```
add al, bl
aaa
```

请分别给出下列情况下：（1）执行add后，aaa执行前，（2）执行aaa后，寄存器AX中的值，以及进位标志位CF和辅助标志位AF的值。

(a) AX: 00 05 BL: 02

(b) AX: 02 06 BL: 03

(c) AX: 03 05 BL: 08

(d) AX: 00 07 BL: 06

(e) AX: 00 09 BL: 08

(f) AX: 02 07 BL: 09

(g) AX: 04 01 BL: 09

(h) AX: 00 00 BL: 01

2. 将练习1中的指令改为

```
sub al, bl
aas
```

其他相同。

3. 在下列两种情况中，假设有如下定义:

```
value    BYTE    8   DUP(?)
```

(a) 设value中的值是ASCII码表示的0 ~ 9的数字。编写一段代码，实现“在原位置”（不将字节复制到其他位置）将这些字节替换为对应的未压缩的BCD码数。

(b) 设value中的值是八字节长的未压缩的BCD码数。请编写一个代码段，实现“在原位置”（不将字节复制到其他位置）将这些字节替换为对应的ASCII码表示的0 ~ 9的数字。

4. 下列每小题中，假设执行指令

```
mul    ch
aam
```

请分别给出:（1）执行mul后，aam执行前，（2）执行aam后，寄存器AX中的值。

(a) AL: 05 CH: 02

(b) AL: 06 CH: 03

(c) AL: 03 CH: 08

(d) AL: 07 CH: 06

(e) AL: 09 CH: 08

(f) AL: 07 CH: 09

(g) AL: 04 CH: 09

(h) AL: 08 CH: 01

5. 下列每小题中，假设执行指令

```
aad
div   dl
aam
```

请分别给出:（1）执行add后，div执行前，（2）执行div后，aam执行前，（3）执行aam后，寄存器AX中的值。

(a) AX: 07 05 DL: 08

(b) AX: 05 06 DL: 09

(c) AX: 02 07 DL: 08

(d) AX: 04 07 DL: 06

(e) AX: 05 09 DL: 06

(f) AX: 03 07 DL: 07

(g) AX: 07 04 DL: 03
(h) AX: 05 00 DL: 04

编程练习11.3

1. 编写一个过程*subUnp*，找出两个八字节未压缩的BCD码数的区别。该过程有两个参数：（1）operand1和destination的地址，（2）operand2的地址。operand1 - operand2的值保存在destination中。如果源数大于destination中的值，那么将CF设置为“1”，否则置为“0”。其他标志位的值不变，这个过程将从堆栈中取出参数。
2. 有一种可变长度的表示多字节未压缩的BCD码数的方法。第一个字节中的无符号二进制数用来说明这个未压缩的BCD数中共有多少个十进制数字，数字从右到左存储（低位到高位）。例如：十进制数1234567890存储为0A 00 09 08 07 06 05 04 03 02 01，这种表示方法最多可以存储255个数字的十进制数。

   编写一个过程*addVar*，实现两个可变长的未压缩的BCD码数的加法。该过程有两个参数：（1）operand1和destination的地址（2）operand2的地址。operand1 + operand2的值保存在destination中，这两个数不需要一样长。相加的和的长度与较长操作数的长度相等，或比它长一个字节。假设在目的地址预留了足够的空间可存储相加的和，即使operand1是比较短的操作数。这个过程将从堆栈中取出参数。

11.4 其他体系结构：VAX压缩的十进制指令

由于80x86体系结构对压缩的十进制运算提供的支持非常有限，因此，为了使用压缩的十进制数，必需要有一个很大的过程库。其他一些体系结构对压缩的十进制数提供了广泛的硬件支持。本节简要介绍了在VAX体系结构中定义的压缩十进制指令，虽然它们没有必要在所有的VAX机器都能够执行。

VAX体系结构根据十进制数的长度和起始地址定义了一个压缩十进制字符串。长度给出了字符串中十进制数字的个数，而不是字节的个数。最后四位（半个字节）是一个符号标识，通常，C_{16}表示正数，D_{16}表示负数。因为十进制数字压缩为每字节存放两个数字，所以一个压缩的十进制字符串的长度（字节长度）大约是阿拉伯数字个数的一半。更准确地说，对于一个有n个十进制数字的数，如果n是奇数，那么压缩的十进制字符串的长度为 $(n + 1)/2$。如果n是偶数，那么压缩的十进制字符串的长度为 $(n + 2)/2$。

VAX体系结构中有一套完整的指令，用来执行压缩十进制数的运算：ADDP（add packed）、DIVP（divide packed）、MULP（multiply packed）和SUBP（subtract packed）。其中每条指令至少有四个操作数，用来说明每个压缩的十进制字符串的长度和地址。如果只指定了两个字符串，其中的一个同时作为源数据和目的数据。此外还有六个操作数的指令，其中源数据和目的数据的格式是分别指定的。（MULP和DIVP指令只有六个操作数的形式。）指令MOVP（move packed）可以将一个压缩的十进制字符串从一个地址复制到另一个地址。通过设置条件码（标志位），指令CMPP（compare packed）可以比较两个压缩的十进制字符串。

回想一下，压缩十进制数与其他形式的数相互转换是很困难的，VAX体系结构为此提供了六条不同的指令，有些可用来实现压缩的十进制字符串和32位二进制补码整数之间转换，还有一些可用来实现压缩十进制字符串和以其他记数方式表示的字符串（包括ASCII码）之间

转换。此外，还有一条EDIT指令，在转换过程中执行一些可能的编辑操作，这条指令可以将一个压缩的十进制字符串转换成一个字符型字符串。（与编程练习11.1 #4很相似，但是要简单的多。）

COBOL语言直接支持压缩的十进制类型和运算。如果为VAX体系结构编写一个COBOL编译器，那么压缩的十进制指令可以极大地简化这项工作。和用软件程序对每个压缩的十进制运算模拟相比，这样的编译器生成的代码更简洁、有效。

本章小结

整数可以用二进制编码的十进制形式存储在计算机中，而不用无符号或者二进制补码的形式。有两种基本的BCD码形式：压缩的和未压缩的。压缩的BCD码数每字节存储两个十进制数字，未压缩的BCD码数每个字节存储一个十进制数字。

二进制表示比BCD码表示更紧凑。80x86处理器有很多指令可实现二进制数的运算。但是，在存储很大的整数时，BCD码表示就比较方便，而且可以容易实现BCD码和ASCII码之间的相互转换。

BCD码表示可以使用一个可变的或者固定的字节数，也可以存储或者不存储符号标识。MASM汇编器提供了一条DT指令，该指令可生成一个十字节有符号的压缩的BCD码数。使用BYTE指令，可初始化未压缩的BCD码数。

通过使用普通的二进制指令将两个操作数组成字节对，可实现BCD码数的运算。然后，将二进制的结果转换为BCD码。压缩的十进制表示使用指令daa和das，这些指令和二进制运算指令一起使用，可实现压缩的BCD码的运算。

下列四条指令常用于未压缩的BCD码的运算的实现：aaa、aas、aam和aad。指令aad和其他指令略有不同，在执行div指令之前，指令aad要将BCD码的结果转换为二进制形式。

还有其他一些体系结构提供了一套更为完整的压缩的十进制数的指令，特别是，VAX系统结构还提供了算术运算、数据传送、数据比较和数据转换等指令。

第12章 输入/输出

前面章节中介绍的程序用宏input从PC控制台键盘输入数据，并用宏output将数据输出显示到终端上。汇编语言程序的输入和输出受到键盘和显示器的限制。本章讨论了通过input宏和output宏对底层操作系统的调用，然后考查了可将连续文件读写到辅助存储器的类似的操作系统的调用。本章还介绍了80x86实际用于输入输出操作的指令，并讨论了其他的输入/输出（I/O）方案，包括存储器映射和中断驱动的输入/输出（I/O）。

12.1 使用Kernel32库的控制台输入/输出

代码段12-1给出了一个简单的例子，该例子说明了kernel 32函数是如何输出简单信息的。总的来说，这个例子与本书前面章节中所举的例子很相似。但是，该例没有“标准”指令INCLUDE io.h。除了熟悉的ExitProcess函数模型外，该例中还有两个新的函数模型，要把信息输出到控制台，就必须用到这两个函数。

代码段12-1 使用kernel32函数的控制台输出

```
; 该程序输出一段简单的信息
; 作者: R. Detmer
; 日期: 1998年6月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

GetStdHandle PROTO NEAR32 stdcall,
    nStdHandle:DWORD

WriteFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToWrite:DWORD,
    lpNumberOfBytesWritten:NEAR32, lpOverlapped:NEAR32

STD_OUTPUT EQU -11

cr      EQU     0dh     ; 回车
Lf      EQU     0ah     ; 换行

.STACK
.DATA

OldProg BYTE    "Old programmers never die.", cr, lf
        BYTE    "They just lose their byte.", cr, lf
msgLng  DWORD   56      ; 上述信息中的字符数
written DWORD   ?
hStdOut DWORD   ?
```

```
.CODE
_start:
        INVOKE GetStdHandle,        ; 获得控制台输出的句柄值
          STD_OUTPUT
        mov    hStdOut, eax

        INVOKE WriteFile,
          hStdOut,                  ; 要显示的文件的句柄
          NEAR32 PTR OldProg,       ; 字符串的地址
          msgLng,                   ; 字符串的长度
          NEAR32 PTR written,       ; 已写的字节
          0                         ; 重写模式

        INVOKE  ExitProcess, 0      ; 退出，并返回代码0

PUBLIC _start
END
```

Windows 95/98/NT操作系统和其他的系统有些地方很相似，比如，输入/输出设备和磁盘文件都用相同的方式对待。注意，在代码段12-1中，WriteFile调用就是用来在控制台显示信息的，它同样可以用来写磁盘文件。输入/输出所用的设备或文件是由句柄决定的，在汇编语言程序中，它定义为双字，在调用WriteFile之前，必须先获得句柄值。对于控制台文件，多种方法可用来获得句柄，如GetStandardHandle 就是一种很简单的方法。

任何GetStdHandle调用都有一个参数：数字值，与句柄不同，该参数用来指明特定的设备。有三种标准设备：一个用来输入，一个用来输出，还有一个用来报告错误（一般情况下，它和标准输出设备一样）。每一个设备号都有一个等同的符号，这些符号常用在代码中，表12-1列出了输入和输出设备的号码和名称。GetStdHandle是一个函数，将标准的输入/输出（I/O）设备的句柄值返回到寄存器EAX。句柄值通常储存在存储器中，以备将来使用。在这个例子程序中，返回值立刻被复制到hStdout所指的的双字中。

表12-1 标准设备号

助 记 符	等 同 值
STD_INPUT	-10
STD_OUTPUT	-11

有五个参数的WriteFile调用比较复杂。第一个参数是标识文件的句柄，该句柄通过GetStdHandle返回，但并不是设备号。第二个参数是字符串的地址，注意NEAR32 PTR操作数的使用，在这个例子中，该操作数告诉汇编器使用OldProg的地址，而不是存储在该地址的值。第三个参数是一个双字，它包含了要显示的字节数。第四个参数用于给调用程序返回一个值，这个值说明了实际被写入的字节数，当向控制台输出时，只要没有错误发生，这个返回值就是信息的长度。第五个参数也是最后一个参数，在本书的例子中，该参数是0，它用来说明对某些文件的非连续访问，但本书只处理连续访问。

控制台的输入和输出一样简单。代码段12-2给出了一个例子程序，该程序输入一个字符串，把每一个大写字母转化成小写字母，并输出最终的字符串。

代码段12-2 使用kernel32函数的控制

```
; 该程序输入一段简单的信息，并且用小写字母输出
; 作者：R. Detmer
; 日期：1998年6月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

GetStdHandle PROTO NEAR32 stdcall,
    nStdHandle:DWORD

ReadFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToRead:DWORD,
    lpNumberOfBytesRead:NEAR32, lpOverlapped:NEAR32

WriteFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToWrite:DWORD,
    lpNumberOfBytesWritten:NEAR32, lpOverlapped:NEAR32

STD_INPUT  EQU -10
STD_OUTPUT EQU -11

.STACK
.DATA

prompt  BYTE    "String to convert? "
CrLf    BYTE    0ah, 0dh
StrIn   BYTE    80 DUP (?)
read    DWORD   ?
written DWORD   ?
hStdIn  DWORD   ?
hStdOut DWORD   ?

.CODE
_start:
        INVOKE GetStdHandle,        ; 获得控制台输出的句柄值
          STD_OUTPUT
        mov    hStdOut, eax

        INVOKE WriteFile,
          hStdOut,                  ; 屏幕的文件句柄
          NEAR32 PTR prompt,        ; 输入地址
          19,                       ; 输入长度
          NEAR32 PTR written,       ; 写入的字节
          0                         ; 重写模式

        INVOKE GetStdHandle,        ; 得到控制台输入句柄
          STD_INPUT
        mov    hStdIn, eax

        INVOKE ReadFile,
          hStdIn,                   ; 键盘的文件句柄
```

```
            NEAR32 PTR StrIn,                   ; 字符串地址
            80,                                 ; 读入的最大数字
            NEAR32 PTR read,                    ; 读入的字节
            0                                   ; 重写模式

        mov     ecx, read                       ; 建立转换循环
        lea     ebx, StrIn                      ; 起始地址
forCh:  cmp     BYTE PTR [ebx], 'A'             ; char<'A'?
        jl      endIfUpper                      ; 如果是，转移
        cmp     BYTE PTR [ebx], 'Z'             ; char>'Z'?
        jg      endIfUpper                      ; 如果是，转移
        add     BYTE PTR [ebx], 'a' - 'A'       ; 转换为小写字母
endIfUpper:
        inc     ebx                             ; 指向下一个字符
        loop    forCh                           ; 重复执行

        mov     ecx, read                       ; 输出的长度
        add     ecx, 2                          ; 换行符和回车符

        INVOKE WriteFile,
            hStdOut,                            ; 屏幕的文件句柄
            NEAR32 PTR crLf,                    ; 首先输出换行符和回车符
            ecx,                                ; 输出的长度
            NEAR32 PTR written,                 ; 写入的字节数
            0                                   ; 重写模式

        INVOKE  ExitProcess, 0                  ; 结束，返回代码0

PUBLIC _start
END
```

在这个例子中的新函数是Readfile，它与WriteFile非常相似。但是，它的第二个参数有输入缓冲区的地址，第三个参数给出了能读入字符的最大数，第四个参数返回实际读入的字符数。

一般情况下，实际读入的字符个数要小于缓冲区能接收的字符数。否则，在输入缓冲区之后的存储器中的数值会被破坏。对于控制台的输入，还需要考虑输入字符后要加上回车和换行符。也就是说，如果输入六个字符后，然后按回车键，那么实际上会有八个字符存入输入缓冲区——六个字符加上换行符和回车符。

在代码段12-2所示的程序中，在小写字符输出之前，输出了一个空行，这是因为在输入缓冲区之前，CR/LF（换行符和回车符）已经在存储器中的，输出的起始地址包括这两个字符。因此，字符个数也要加上2。由于原始字符数包含读入的字符后的换行符和回车符，因此，在字符显示后也会跳过一行。

本书用到的宏input和output大多扩展为过程调用，这些过程调用使用kernel32控制台输入/输出函数。代码段12-3列出了文件IO.ASM中的相关部分。

代码段12-3　IO.ASM中输入/输出过程

```
STD_OUTPUT EQU -11
STD_INPUT  EQU -10

GetStdHandle PROTO NEAR32 stdcall,
    nStdHandle:DWORD
```

```
ReadFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToRead:DWORD,
    lpNumberOfBytesRead:NEAR32, lpOverlapped:NEAR32

WriteFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToWrite:DWORD,
    lpNumberOfBytesWritten:NEAR32, lpOverlapped:NEAR32

.DATA

written     DWORD  ?
read        DWORD  ?
strAddr     DWORD  ?
strLength   DWORD  ?
hStdOut     DWORD  ?
hStdIn      DWORD  ?

.CODE

; outproc(source)
; 该过程输出结尾为空的字符串
; 没有寄存器改变; 标志位不受影响

outproc     PROC   NEAR32
            push   ebp                    ; 保存基地址指针
            mov    ebp, esp               ; 初始化堆栈
            pushad
            pushfd                        ; 保存标志位

            mov    esi,[ebp+8]            ; 源地址
            mov    strAddr, esi

; 找出字符串长度
            mov    strLength, 0           ; 初始化字符串长度
WhileChar:  cmp    BYTE PTR [esi], 0      ; 字符为空吗?
            jz     EndWhileChar           ; 如果是, 退出
            inc    strLength              ; 字符数加1
            inc    esi                    ; 指向下一个字符
            jmp    WhileChar
EndWhileChar:

            INVOKE GetStdHandle,          ; 得到控制台输出句柄
              STD_OUTPUT
            mov    hStdOut, eax

            INVOKE WriteFile,
              hStdOut,                    ; 屏幕文件句柄
              strAddr,                    ; 字符串地址
              strLength,                  ; 字符串长度
              NEAR32 PTR written,         ; 写入的字节数
              0                           ; 重写模式

            popfd                         ; 恢复标志位
            popad                         ; 恢复寄存器
            pop    ebp
```

```
            ret     4                       ; 退出，释放参数
outproc     ENDP

; inproc(dest, length)
; 该过程从键盘输入一个字符串。
; 该字符串存放在给定的目标地址。
; 变量长度提供给用户的缓存，假定有空间可以存放该字符串和一个空字节。
; 字符串将以空字符(00h)结尾。
; 不改变标志位。

inproc      PROC    NEAR32
            push    ebp                     ; 保存基址指针
            mov     ebp, esp                ; 初始化堆栈
            pushad                          ; 保存所有寄存器
            pushfd                          ; 保存标志位

            INVOKE GetStdHandle,            ; 获得控制台句柄
              STD_INPUT
            mov     hStdIn, eax
            mov     ecx, [ebp+8]            ; 字符串长度
            mov     strLength, ecx

            mov     esi, [ebp+12]           ; 源地址
            mov     strAddr, esi

            INVOKE ReadFile,
              hStdIn,                       ; 键盘文件句柄
              strAddr,                      ; 字符串地址
              strLength,                    ; 字符串长度
              NEAR32 PTR read,              ; 读入的字节
              0                             ; 重写模式

            mov     ecx, read               ; 读入的字节数
            mov     BYTE PTR [esi+ecx-2],0  ; 用结尾的空字符替代CR/LF

            popfd                           ; 恢复标志位
            popad                           ; 恢复寄存器
            pop     ebp
            ret     8                       ; 退出，并释放参数
inproc      ENDP
```

IO.ASM中的输入/输出代码没什么特别之处，它的起始指令和前面两个例子所用的起始指令一样。数据域没有包括输入缓冲区，因为这将放在用户的调用程序中。数据域用变量strAddr本地保存输入或输出缓冲区地址，该地址作为参数来传递。输出过程outproc把这个地址看作是空字符结尾的字符串的地址。在标准过程入口代码后，该过程计算字符串的长度，然后得到控制台的句柄，并输出到控制台，就像代码段12-1所示的例子一样。

输入过程*inproc*也很简单。在标准过程入口代码后，它获得控制台的句柄，并将两个参数（长度和字符串地址）复制到局部变量。ReadFile调用执行实际输入操作，惟一复杂的地方是inproc过程允许以空字符结尾的字符串，并且由ReadFile读入的字符串是以换行符/回车符（CR/LF）结束。下面的语句

```
mov     ecx, read
mov     BYTE PTR [esi + ecx-2], 0
```

把空字节放在字符串的结尾处，实际上是用空字符代替了回车符。这样做是因为字符串的起始地址是在ESI寄存器中，因此，如果字符个数放在ECX中，那么ESI + ECX－2就指向是输入缓冲区中最后字符的前一个字符的地址。

现在可以再考虑6.1节中所提到的要注意的内容：有的Microsoft操作系统函数可能要求栈必须是双字设置的。在过程中使用这些函数时，只能将双字值压入栈。因此，尽管pushf可以保存对大多程序都有意义的所有的标志位的值，但代码段12-3中所示的程序中还包含了一条pushfd指令。

编程练习12.1

1. 编写一个程序，只用kernel32函数，不用书上I/O的包，根据提示从键盘以先姓后名的方式输入姓名：last, first（即姓、逗号、名）。然后，用合适的标号按先名后姓的方式将输入的姓名显示出来：last first（即名、空格、姓）。
2. 编写一个程序，只用kernel32函数，不用书上I/O的包，根据提示从键盘输入短语，并报告它是否是一个回文（即，是否与短语反过来写一样）。

12.2 使用Kernel 32库的连续文件的输入/输出

文件处理通常涉及打开文件、读文件或写文件，以及最后关闭文件等等。在Kernal32库中，打开文件意味着为文件得到一个句柄。关闭已读文件来释放文件，以便其他用户可以访问这个文件。一个已经写好的文件必须要关闭，这样，操作系统才能保存最后的字符。本节将考察连续磁盘文件的操作，这些文件操作通常更适合用高级语言来执行。因此，本节旨在讨论如何用高级语言执行文件操作。

代码段12-4给出了一个例子程序，它可以提示输入一个文件名，然后把这个文件的内容在控制台上显示出来。该程序包含了两个新的Kernel32函数原型：CreateFileA和CloseHandle。CreateFileA用于打开已存在的文件或者创建一个新文件。CloseHandle则用于关闭文件。

代码段12-4 用kernel32库连续文件输入

```
; 输入连续文件，并在终端显示
; 作者：R. Detmer
; 日期：1998年6月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

STD_OUTPUT     EQU  -11
STD_INPUT      EQU  -10
GENERIC_READ   EQU  80000000h
OPEN_EXISTING  EQU  3

GetStdHandle PROTO NEAR32 stdcall,
    nStdHandle:DWORD
```

```
ReadFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToRead:DWORD,
    lpNumberOfBytesRead:NEAR32, lpOverlapped:NEAR32

WriteFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToWrite:DWORD,
    lpNumberOfBytesWritten:NEAR32, lpOverlapped:NEAR32

CreateFileA PROTO NEAR32 stdcall,
    lpFileName:NEAR32, access:DWORD, shareMode:DWORD,
    lpSecurity:NEAR32, creation:DWORD, attributes:DWORD, copyHandle:DWORD

CloseHandle PROTO NEAR32 stdcall,
    fHandle:DWORD

.DATA
written     DWORD  ?
read        DWORD  ?
fileName    BYTE   60 DUP (?)
hStdOut     DWORD  ?
hStdIn      DWORD  ?
hFile       DWORD  ?
buffer      BYTE   64 DUP (?)
prompt      BYTE   "File name?  "

.CODE
_start:
          INVOKE GetStdHandle,                  ; 控制台输出句柄
            STD_OUTPUT
          mov    hStdOut, eax

          INVOKE GetStdHandle,                  ; 控制台输入句柄
            STD_INPUT
          mov    hStdIn, eax

          INVOKE WriteFile,
            hStdOut,                            ; 屏幕的文件句柄
            NEAR32 PTR prompt,                  ; 提示输入地址
            12,                                 ; 提示输入长度
            NEAR32 PTR written,                 ; 写入的字节数
            0                                   ; 重写模式

          INVOKE ReadFile,
            hStdIn,                             ; 键盘输入文件的句柄
            NEAR32 PTR fileName,                ; 文件名地址
            60,                                 ; 最大长度
            NEAR32 PTR read,                    ; 读取的字节数
            0                                   ; 重写模式

          mov    ecx, read                      ; 读取的字节数
          mov    BYTE PTR fileName[ecx-2],0     ; 结尾加上空字符

          INVOKE CreateFileA,                   ; 打开文件
            NEAR32 PTR fileName,                ; 文件名
            GENERIC_READ,                       ; 访问
```

```
                0,                          ; 没有共享
                0,                          ; 没有预定义安全
                OPEN_EXISTING,              ; 当且仅当文件存在时打开
                0,                          ; 没有特殊的属性
                0                           ; 没有复制句柄
              mov     hFile, eax            ; 文件句柄

readLoop:     INVOKE ReadFile,
                hFile,                      ; 文件句柄
                NEAR32 PTR buffer,          ; 输入的地址
                64,                         ; 缓冲区长度
                NEAR32 PTR read,            ; 读取的字节数
                0                           ; 重写模式

              INVOKE WriteFile,
                hStdOut,                    ; 屏幕的文件句柄
                NEAR32 PTR buffer,          ; 输出的地址
                read,                       ; 输出与输入同样的数
                NEAR32 PTR written,         ; 已写的字节
                0                           ; 重写模式

              cmp     read, 64              ; 是否读取64个字符
              jnl     readLoop              ; 如果是，继续

              INVOKE  CloseHandle,          ; 关闭文件句柄
                hfile

              INVOKE  ExitProcess, 0        ; 退出，并输出0代码

PUBLIC _start                               ; 公开入口点
END                                         ; 结束
```

CreateFileA返回一个由它打开或创建的文件的句柄，如果打开文件失败的话，则返回－1（$FFFFFFFF_{16}$）。CreateFileA有七个参数：

1）一个空字符结尾的文件名字符串的地址。

2）一个提供所期望存取的双字。只使用GENERIC_READ（80000000_{16}）和GENERIC_WRITE（40000000_{16}）。

3）一个说明文件如何可以共享的双字。0表示文件不能共享。

4）用来说明文件是否可以被子程序使用的参数。0表示不能使用。

5）一个包含标志位的双字，用来说明如果文件不存在的话，应执行的操作。当打开一个已经存在的文件时，用OPEN_EXISTING(3)；若文件不存在，CreateFileA函数将无法使用。在创建一个新文件时，用CREATE_NEW(1)；若文件已经存在，CreateFileA函数将无法使用。在其他应用中，CREATE_ALWAYS(2)是比较合适的，如果文件不存在，它创建一个新文件，如果文件存在，它就重写这个已存在的文件。

6）用来设定文件的多种属性的参数。0表示没有特定的属性。

7）最后一个参数，用来表示模板文件的句柄，这个模板文件的属性将只用于新创建的文件。0表示没有模板。

如果使用CreateFileA，就要指定参数1、2和5，并将其他四个参数设为0。

CloseHandle函数非常简单。它只有一个参数，就是要关闭的文件的句柄。

在代码段12-4所示的程序中，读循环用Readfile一次从源文件中读取64个字符。如果实际读入的字符数是64，表示可能还没有读到文件的结尾。如果不到64，说明已经到文件的结尾了。但是要注意，读到的字符将先显示出来，因此，不要丢失最后部分缓冲区的字符。

此例中，数字64是2的整数次方，除此之外没有特别之处。对于磁盘文件的存取，多数操作系统有自己的缓冲区，并且，这样的缓冲区的大小都是2的整数次方，因此，在这个例子程序中，它的缓冲区大小的设置是有道理的。

代码段12-5给出了一个程序，它可以从控制台输入，创建一个磁盘文件。首先它提示并输入文件名，创建文件，如果这个文件已存在，则无法创建。然后将从控制台键盘输入的文字行复制到文件，直到用户以%%开始一行的输入为止。在一般的文本中，不太可能选择合并一个字符，如“%%”这样作为一行的开始。

代码段12-5 从控制台输入创建一个文件

```
; 从控制台输入创建连续文件
; 作者: R. Detmer
; 日期: 1998年6月

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

STD_OUTPUT     EQU  -11
STD_INPUT      EQU  -10
GENERIC_WRITE  EQU  40000000h
CREATE_NEW     EQU  1

GetStdHandle PROTO NEAR32 stdcall,
    nStdHandle:DWORD

ReadFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToRead:DWORD,
    lpNumberOfBytesRead:NEAR32, lpOverlapped:NEAR32

WriteFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToWrite:DWORD,
    lpNumberOfBytesWritten:NEAR32, lpOverlapped:NEAR32

CreateFileA PROTO NEAR32 stdcall,
    lpFileName:NEAR32, access:DWORD, shareMode:DWORD,
    lpSecurity:NEAR32, creation:DWORD, attributes:DWORD,
copyHandle:DWORD

CloseHandle PROTO NEAR32 stdcall,
    fHandle:DWORD

.DATA

written    DWORD  ?
```

```
read        DWORD   ?
fileName    BYTE    60 DUP (?)
hStdOut     DWORD   ?
hStdIn      DWORD   ?
hFile       DWORD   ?
buffer      BYTE    128 DUP (?)
prompt1     BYTE    "File name?  "
prompt2     BYTE    "Enter text.  Start a line with %% to stop", 0dh, 0ah

.CODE
_start:
            INVOKE GetStdHandle,                ; 输出控制台句柄
              STD_OUTPUT
            mov     hStdOut, eax

            INVOKE GetStdHandle,                ; 输入控制台句柄
              STD_INPUT
            mov     hStdIn, eax

            INVOKE WriteFile,
              hStdOut,                          ; 屏幕的文件句柄
              NEAR32 PTR prompt1,               ; 输入的地址
              12,                               ; 输入长度
              NEAR32 PTR written,               ; 写入的字节数
              0                                 ; 重写模式

            INVOKE ReadFile,
              hStdIn,                           ; 键盘的文件句柄
              NEAR32 PTR fileName,              ; 文件名的地址
              60,                               ; 最大长度
              NEAR32 PTR read,                  ; 读取的字节
              0                                 ; 重写模式

            mov     ecx, read                   ; 读取的字节数
            mov     BYTE PTR fileName[ecx-2],0  ; 结尾加上空字符

            INVOKE CreateFileA,                 ; 打开文件
              NEAR32 PTR fileName,              ; 文件名
              GENERIC_WRITE,                    ; 访问
              0,                                ; 没有共享
              0,                                ; 没有预定义安全
              CREATE_NEW,                       ; 如果文件存在，退出
              0,                                ; 没有特殊属性
              0                                 ; 没有复制句柄
            mov     hFile, eax                  ; 文件的句柄

            INVOKE WriteFile,
              hStdOut,                          ; 屏幕的文件句柄
              NEAR32 PTR prompt2,               ; 输入的地址
              43,                               ; 输入长度
              NEAR32 PTR written,               ; 已写的字节数
              0                                 ; 重写模式

readLoop:   INVOKE ReadFile,
              hStdIn,                           ; 从控制台读取
```

```
                NEAR32 PTR buffer,        ; 输入的地址
                128,                      ; 缓冲区长度
                NEAR32 PTR read,          ; 读取的字节数
                0                         ; 重写模式

              cmp     buffer, "%"         ; 第1个字符是%吗?
              jne     continue            ; 如果不是，继续
              cmp     buffer+1, "%"       ; 如果是，第2个字符是%吗?
              je      endRead             ; 如果是，退出
continue:
              INVOKE WriteFile,
                hfile,                    ; 文件句柄
                NEAR32 PTR buffer,        ; 输出的地址
                read,                     ; 输出与输入同样的数
                NEAR32 PTR written,       ; 写入的字节数
                0                         ; 重写模式

              jmp     readLoop            ; 如果是，继续
endRead:

              INVOKE  CloseHandle,        ; 关闭文件句柄
                hfile

              INVOKE  ExitProcess, 0      ; 退出，返回代码0

PUBLIC _start                             ; 公开程序入口点
END                                       ; 源代码结束
```

这个例子中有一些新的内容。用GENERIC_WRITE调用CreateFileA，CREATE_NEW创建新文件。主循环从键盘读入一个字符串，该字符串最多可有128个字符，并将这个字符串写入文件。在将字符串写入文件前，首先检查字符串的前两个字符，判断循环控制是否终止。

练习12.2

1. 本节所给的例子没有检查以确保文件打开是否成功。代码段12-4中，即使文件没有成功打开，为什么程序仍旧正常运行了？修改代码段12-4的程序，如果文件没有打开，程序将给出一个警告信息，并结束运行。
2. 本节所给的例子没有检查以确保文件打开成功。如果运行代码段12-5中的程序，会出现什么情况？特别是，如果输出文件已存在，又会出现什么情况？修改代码段12-5的程序，如果文件没有打开，程序将给出一个警告信息，并结束运行。

编程练习12.2

1. 文件堆程序按照两字符十六进制代码输出文件的每个字节，如果有相应的可打印字符形式，则输出该打印字符。只使用kernel32库（不用IO.ASM），写一个堆程序，该程序输入文件名，然后用下列格式输出文件：

   每行显示16个字符，首先每个字节用一对十六进制形式输出，每对之间插入一个空格，这样一共占据48个位置，然后对于普通字符，用句点代替非打印字符，中间不用空格。常见的一行输出形式如下：

```
50 72 6F 67  72 61 6D 6D 69 6E 67 20 0D 0A 69 73 Programming...is
```

在显示器上显示20行之后，用户根据提示“m[ore] or q[uit]?”决定继续输出20行，还是响应后退出运行。

2. 编写一个程序，将源文件复制到目的文件。值得注意的是，程序必须提示输入源文件名，打开源文件，如果不能打开，则给出错误信息并退出。如果源文件能打开，用户将根据提示输入目的文件名。如果目的文件已经存在，则决定是否用CREATE_NEW来打开该文件，用户应该回答是否用CREATE_ALWAYS破坏旧的文件，若用户的回答是否定，则程序应终止运行。如果目的文件不存在，在文件复制前不需要警告。只用kernel32库中的输入/输出函数，不要用IO.H的宏。
3. 编写一个程序，将源文件复制到目的文件。并将其中的大写字母转换成小写字母，其他字符保持不变。程序必须提示源文件名和目的文件名。在文件复制前，即使目的文件已存在，也不需要给出警告。只用kernel32库中的输入/输出函数，不要用IO.H的宏。
4. 编写一个程序，处理文件RECORDS.DAT中的固定格式记录集。文件的每一行由如下形式的ASCII码组成:
   - 1 ~ 20列显示人名
   - 21 ~ 25列显示整数，右对齐

   文件的每一行以回车符和换行符结束，一行的总长度是27个字符。这样的文件可用标准文本编辑器生成。这个程序必须显示数据行并且报告
   - 记录的数字
   - 数字的总和
   - 最大数所对应的人名

   只用kernel32库中的输入/输出函数，不要用IO.H的宏，但可以使用IO.H的atod和dtoa宏指令。

12.3 低级输入/输出

本书的前面的部分都是用IO.H中的宏指令进行输入和输出操作。在这一章，输入和输出都是采用kernel32库的函数调用，它是一种较低级的方式，高级语言可提供较高级的输入/输出。本节讨论的输入/输出方式比kernel32库的函数调用更低级，它在Intel 80x86和其他的结构中使用。由于低级输入/输出日益受到操作系统的限制，因此，本节没有提供实际的代码。

正如第2章中所讨论的，Intel 80x86的体系结构的存储器地址是从00000000_{16}到$FFFFFFFF_{16}$。它还有独立的I/O地址空间，其端口地址是从0000_{16}到$FFFF_{16}$。本书的很多指令都用到了存储器地址，但是，只有小部分指令使用端口地址。其中最常用的是in和out指令，它们用于指定的端口地址和累加器（如AL、AX或EAX等）之间的数据传送。因此，这两条指令类似于受限的mov指令。

在IBM兼容的PC机上，通用I/O设备都有标准的端口分配地址。例如，LPT1作为并行打印机端口，它占用了3个端口地址：0378、0379和037A。其中第一个端口用于向打印机传送字符，第二个端口用于确定打印机的状态，第三个用于向打印机发送控制信息。串行端口一般由串行输入/输出（SIO）芯片控制，它们也需要若干端口地址。

80x86体系结构还可使用存储器映射I/O。用存储器映射I/O，一些普通的存储器地址也可用于输入输出，并且，常规的数据的传送指令可用于实现存储器与外设之间的数据传递。设计系统时，硬件设计者可以选择是否采用存储器映射I/O，或者独立的I/O地址空间。其他一些

体系结构，如摩托罗拉680x0，只采用了存储器映射I/O。

不管I/O设备地址是如何分配的，有一个问题必须考虑：什么时候程序可以从设备接收字符，或者反过来说，如何保证设备已经准备从程序接收字符了。如果要将一个字符传送到一个过时的、反应慢的、机械式打印机上的话，显然，计算机发送字符的速度比打印机打印字符的速度要快得多。这时，可采用轮流检测技术，也就是说，程序重复检查设备端口的状态，直到它得到设备已准备好接收字符的通知，程序再发送字符。设计如下：

```
forever
    从状态端口得到状态;
    if 清0，传送字符 then退出循环;
end loop;
传送字符到数据端口;
```

这个设计中的循环叫做忙等待循环。除非计算机可处理多个任务，否则在等待设备接收字符期间，计算机无法做其他的工作。

中断驱动的I/O依赖硬件中断通知CPU某个设备状态的变化。中断是设备产生的硬件信号，并且该信号可被CPU接收。当CPU接收到这样的信号时，它会正常终止当前正在运行的指令，并转去执行中断过程，这和常规的过程调用很相似。

Intel 80x86系统提供了多达256种不同的中断，每个中断过程的地址来自存储器最底部的地址表。0到1024_{10}的存储器地址单元包含了256个地址，对应0到255的中断类型。通常，对于中断类型t，中断过程的地址是存储在4*t的位置上。

当用户按下某个键时，计算机系统可产生一个中断。相关的中断过程将获取这个字符，在中断返回前，该字符会被存入一个缓冲区中，以备后用，计算机可继续执行它当时正在执行的程序。

80x86体系结构包含了一条int 指令，可使一个程序调用一个中断过程。不是所有的中断类型都能用于硬件设备，有些是用于操作系统的，特别是Microsoft DOS，使用int指令调用操作系统函数。

80x86中断0到4是预设好的。当除数为0时，中断类型0自动被80x86的CPU调用。对于一个包含了指令int 0的简单的程序而言，它也可调用除数为0的中断句柄，这说明特定的80x86系统可以处理除法错误，而不需要实际执行一个除法运算。

中断类型4的句柄也有特定的用途，处理由指令产生的溢出。这个中断句柄不是由80x86自动调用的。它可用int 4调用，但更常用into（interrupt on overflow）指令调用。这是一个有条件的调用：只有当溢出标志位OF被置为1时，这个溢出中断句柄才能被调用，否则继续执行下一条指令。通常，在一条有可能导致溢出的指令后，有一条into指令。

练习12.2

1. 存储器映射I/O的优点是什么？为I/O使用单独地址的优点是什么？
2. 80x86系统中哪个地址存放着中断15_{10}的中断过程地址？

本章小结

输入和输出可在许多层面上实现，从高级语言过程到一般的in 和out指令。Kernel32 库给

出了在操作系统层实现I/O的实例，该库具有获取文件或设备的句柄，从文件或设备读出，向文件或设备写入，以及释放文件或设备等功能。

从硬件角度来看，I/O可为外部设备使用独立的端口地址，也可使用存储器映射I/O，为外部设备分配一部分常规的存储器空间，而不是存储器。

设备可用轮流检测来访问，或者用中断驱动输入/输出，这是一种更有效的方法。80x86体系结构提供了多达256种不同的中断，尽管这些中断的分配常常不是用于I/O请求。

附录A　十六进制/ASCII码的转换

00	NUL	(null)
01	SOH	
02	STX	
03	ETX	
04	EOT	
05	ENQ	
06	ACK	
07	BEL	(bell)
08	BS	(backspace)
09	HT	(tab)
0A	LF	(line feed)
0B	VT	
0C	FF	(form feed)
0D	CR	(return)
0E	SO	
0F	SI	
10	DLE	
11	DC1	
12	DC2	
13	DC3	
14	DC4	
15	NAK	
16	SYN	
17	ETB	
18	CAN	
19	EM	
1A	SUB	
1B	ESC	("escape")
1C	FS	
1D	GS	
1E	RS	
1F	US	
40	@	
41	A	
42	B	
43	C	
44	D	
45	E	
46	F	
47	G	
48	H	
49	I	
4A	J	

20	space
21	!
22	"
23	#
24	$
25	%
26	&
27	'
28	(
29	)
2A	*
2B	+
2C	,
2D	-
2E	.
2F	/
30	0
31	1
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9
3A	:
3B	;
3C	<
3D	=
3E	>
3F	?
60	`
61	a
62	b
63	c
64	d
65	e
66	f
67	g
68	h
69	i
6A	j

4B	K
4C	L
4D	M
4E	N
4F	O
50	P
51	Q
52	R
53	S
54	T
55	U
56	V
57	W
58	X
59	Y
5A	Z
5B	[
5C	\
5D	]
5E	^
5F	_
6B	k
6C	l
6D	m
6E	n
6F	o
70	p
71	q
72	r
73	s
74	t
75	u
76	v
77	w
78	x
79	y
7A	z
7B	{
7C	|
7D	}
7E	~
7F	DEL

附录B 常用的MS-DOS命令

MS-DOS（和Windows）使用类似Unix的分级文件结构。MS-DOS的文件通过驱动器（C:、A:等等）区分，驱动器后紧跟着路径以区分目录（文件夹），最后是文件名本身。例如，一个完整的文件名：A:\asm\project1\example.asm。符号“\”用于分隔路径中的组成部分与根目录的名字（顶级）。大多数MS-DOS系统设置显示当前的驱动器，以及作为提示符部分的路径（例如，C:\WINDOWS>）。

如果不具体指定路径中的驱动器或者目录，默认（*Default*）指的是当前使用的驱动器或者目录。如要改变默认（当前）驱动器，只需敲一个新的驱动器字母和冒号就可以了。

如要改变默认（当前）目录，可用CD命令，符号“..”为当前目录的父目录的快捷方式。例如，如果当前目录为\Windows\Desktop，那么CD..将改变当前目录到\Windows。（注意：MS-DOS不区分大小写，cd同样可实现该功能。）

MD命令创建一个新的目录。要在当前目录中创建一个新的目录，可用MD *目录名*来实现。

DIR命令显示当前文件夹中的文件目录。另外，DIR命令也可给出所想要的目录的路径。如，DIR C:\projects。可用*作为一个通配符。例如，DIR s*.*查找所有名字以字母s开头的文件。

COPY命令将文件从一个目录复制到另外一个目录。格式为COPY *源文件 目的文件*。如果不具体指定一个目的文件的名字，那么，目的文件名将采用源文件名。也可用COPY命令在同一个目录中创建一个文件的副本，但要用不同的文件名。COPY命令允许使用通配符*来复制一组文件。

EDIT命令用于创建或者修改一个文本（text）文件。EDIT *文件名*调用一个简单的文本编辑器，如果该文件名的文件存在，则打开该文件。如果该文件名的文件不存在，则创建该文件。EDIT自身带有很多信息的帮助系统，这些信息要比所需求的多得多。

REN命令用来给文件重命名。格式为REN *旧文件名 新文件名*。

DOSKEY命令装入扩展名到命令处理器，允许使用“↑”（up）键来返回到前一个命令，从而可以再次执行或者编辑。

通过敲入命令“/?”，可获得大多数命令的更多信息。

注意：如果在MS-DOS下正在做某件事情，这并不意味着不能使用其他的Windows工具。可用“我的电脑”或者“Explorer”来创建目录、复制文件、重命名文件等等。也可用记事本来编辑文件，但是要注意的是，记事本通常对每一个文件加上扩展名TXT。如program.asm.txt.这样的文件名结束的话，就可能让人混淆。通常，要避免使用字处理程序来编辑像汇编语言源代码文件那样的文本文件。

附录C　MASM 6.11保留字

AAA	BTR	COMM	DUP
AAD	BTS	COMMENT	DWORD
AAM	BX	COMMON	DX
AAS	BYTE	CONST	EAX
ABS	CALL	.CONTINUE	EBP
ADC	CARRY?	@Cpu	EBX
ADD	CASEMAP	.CREF	ECHO
AH	CATSTR	CS	ECX
AL	@CatStr	@CurSeg	EDI
ALIGN	CBW	CWD	EDX
.ALPHA	CDQ	CWDE	ELSE
AND	CH	CX	ELSEIF
AND	CL	DAA	ELSEIFDIF
ARPL	CLC	DAS	ELSEIFIDN
ASSUME	CLD	.DATA	EMULATOR
AT	CLI	@data	END
AX	CLTS	.DATA?	ENDIF
BH	CMC	@DataSize	.ENDIF
BL	CMP	@Date	ENDM
BOUND	CMPS	DEC	ENDP
BP	CMPSB	DH	ENDS
.BREAK	CMPSD	DI	ENDW
BSF	CMPSW	DIV	ENTER
BSR	CMPXCHG	DL	@Environment
BSWAP	.CODE	.DOSSEG	EPILOGUE
BT	@code	DOTNAME	EQ
BTC	@CodeSize	DS	EQU

ERR
.ERRB
ERRDEF
.ERRDIF
.ERRE
.ERRIDN
.ERRNB
ERRNDEF
.ERRNZ
ESI
ES
ESP
EVEN
.EXIT
EXITM
EXPORT
EXPR16
EXPR32
EXTERN
EXTERNDEF
@F
F2XM1
FABS
FADD
FADDP
FARDATA
@fardata
FARDATA?
@fardata?
FBLD
FBSTP
FCHS
FCLEX
FCOM
FCOMP
FCOMPP
FCOS
FDECSTP
FDISI
FDIV
FDIVP
FDIVR
FDIVRP
FENI
FFREE
FIADD
FICOM
FICOMP
FIDIV
FIDIVR
FILD
@FileCur
@FileName
FIMUL
FINCSTP
FINIT
FIST
FISTP
FISUB
FISUBR
FLAT
FLD
FLD1
FLDCW
FLDENV
FLDENVD
FLDENVW
FLDL2E
FLDL2T
FLDLG2
FLDLN2
FLDPI
FLDZ
FMUL
FMULP
FNCLEX
FNDISI
FNENI
FNINIT
FNOP
FNSAVE
FNSAVED
FNSAVEW
FNSTCW
FNSTENV
FNSTENVD
FNSTENVW
FNSTSW
FOR
FORC
FORCEFRAME
FPATAN
FPREM
FPREM1
FPTAN
FRNDINT
FRSTOR
FRSTORD
FRSTORW
FS
FSAVE
FSAVED
FSAVEW
FSCALE
FSETPM
FSIN
FSINCOS
FSQRT
FST
FSTCW
FSTENV
FSTENVD
FSTENVW
FSTP
FSTSW
FSUB
FSUBP
FSUBR
FSUBRP
FTST
FUCOM
FUCOMP
FUCOMPP
FWAIT
FWORD
FXAM
FXCH
FXTRACT
FYL2X
FYL2XP1
GE
GOTO
GROUP
GS
GT
HIGH
HIGHWORD
HLT
IDIV
IF
.IF
IFB
IFDEF
IFDIF
IFDIFI
IFE
IFIDN
IFIDNI

IFNB
IFNDEF
IMUL
IN
INC
INCLUDE
INCLUDELIB
INS
INSB
INSD
INSTR
@InStr
INSW
INT
INTO
INVD
INVLPG
INVOKE
IRET
IRETD
JA
JAE
JB
JBE
JC
JCXZ
JE
JECXZ
JG
JGE
JL
JLE
JMP
JNA
JNAE
JNB
JNBE
JNC
JNE
JNG
JNGE
JNL
JNLE
JNO
JNP
JNS
JNZ
JO
JP
JPE
JPO
JS
JZ
LABEL
LAHF
LANGUAGE
LAR
LDS
LE
LEA
LEAVE
LENGTHOF
LES
LFS
LGDT
LGS
LIDT
@Line
.LIST
.LISTALL
.LISTIF
.LISTMACRO
.LISTMACROALL
LJMP
LLDT
LMSW
LOADDS
LOCAL
LOCK
LODS
LODSB
LODSD
LODSW
LOOP
LOOPD
LOOPW
LOW
LOWWORD
LROFFSET
LSL
LSS
LT
LTR
M510
MACRO
MASK
MEMORY
MOD
.MODEL
@Model
MOV
MOVS
MOVSB
MOVSD
MOVSW
MOVSX
MOVZX
MUL
NE
NEG
.NO87
.NOCREF
NODOTNAME
NOKEYWORD
.NOLIST
.NOLISTIF
.NOLISTMACRO
NOLJMP
NOM510
NOP
NOREADONLY
NOSCOPED
NOSIGNEXTEND
NOT
OFFSET
OPTION
OR
ORG
OUT
OUTS
OUTSB
OUTSD
OUTSW
OVERFLOW?
PAGE
PARA
PARITY?
POP
POPA
POPAD
POPCONTEXT
POPF
POPFD
PRIVATE
PROC
PROLOGUE
PROTO
PTR

PUBLIC
PURGE
PUSH
PUSHA
PUSHAD
PUSHCONTEXT
PUSHD
PUSHF
PUSHFD
PUSHW
QWORD
.RADIX
RCL
RCR
READONLY
REAL10
REAL4
REAL8
RECORD
REP
REPE
REPEAT
REPNE
REPNZ
REPZ
RET
RETF
RETN
ROL
ROR
SAHF
SAL
SAR
SBB
SBYTE
SCAS
SCASB
SCASD
SCASW
SCOPED
SDWORD
SEG
SEGMENT
.SEQ
SET
.SETIF2
SGDT
SHL
SHL
SHLD
SHORT
SHR
SHR
SHRD
SI
SIDT
SIGN?
SIZEOF
SIZESTR
@SizeStr
SLDT
SMSW
SP
SS
.STACK
@stack
.STARTUP
STC
STD
STDCALL
STI
STOS
STOSB
STOSD
STOSW
STR
STRUCT
SUB
SUBSTR
@SubStr
SUBTITLE
SWORD
SYSCALL
TBYTE
TEST
TEXTEQU
.TFCOND
THIS
@Time
TITLE
TYPE
TYPEDEF
UNION
.UNTIL
USE16
USE32
USES
VERR
@Version
VERW
WAIT
WBINVD
WHILE
.WHILE
WIDTH
WORD
@WordSize
XADD
XCHG
XLAT
XOR
ZERO?

附录D 80x86指令（带助记符）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
aaa	none	AF,CF SF,ZF,OF,PF ?	37	1	4	3	3
aad	none	SF,ZF,PF OF,AF,CF ?	D5 0A	2	19	14	10
aam	none	SF,ZF,PF OF,AF,CF ?	D4 0A	2	17	15	18
aas	none	AF,CF SF,ZF,OF,PF ?	3F	1	4	3	3
adc	AL,imm8	SF,ZF,OF,CF,PF,AF	14	2	2	1	1
adc	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	15	3 5	2	1	1
adc	reg8,imm8	SF,ZF,OF,CF,PF,AF	80	3	2	1	1
adc	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	81	4 6	2	1	1
adc	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	83	3	2	1	1
adc	mem8,imm8	SF,ZF,OF,CF,PF,AF	80	3+	7	3	3
adc	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	81	4+ 6+	7	3	3
adc	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	83	3+	7	3	3
adc	reg8,reg8	SF,ZF,OF,CF,PF,AF	12	2	2	1	1
adc	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	13	2	2	1	1
adc	reg8,mem8	SF,ZF,OF,CF,PF,AF	12	2+	6	2	2
adc	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	13	2+	6	2	2
adc	mem8,reg8	SF,ZF,OF,CF,PF,AF	10	2+	7	3	3
adc	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	11	2+	7	3	3
add	AL,imm8	SF,ZF,OF,CF,PF,AF	04	2	2	1	1
add	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	05	3 5	2	1	1

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
add	reg8,imm8	SF,ZF,OF,CF,PF,AF	80	3	2	1	1
add	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	81	4 6	2	1	1
add	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	83	3	2	1	1
add	mem8,imm8	SF,ZF,OF,CF,PF,AF	80	3+	7	3	3
add	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	81	4+ 6+	7	3	3
add	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	83	3+	7	3	3
add	reg8,reg8	SF,ZF,OF,CF,PF,AF	02	2	2	1	1
add	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	03	2	2	1	1
add	reg8,mem8	SF,ZF,OF,CF,PF,AF	02	2+	6	2	2
add	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	03	2+	6	2	2
add	mem8,reg8	SF,ZF,OF,CF,PF,AF	00	2+	7	3	3
add	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	01	2+	7	3	3
and	AL,imm8	SF,ZF,OF,CF,PF,AF	24	2	2	1	1
and	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	25	3 5	2	1	1
and	reg8,imm8	SF,ZF,OF,CF,PF,AF	80	3	2	1	1
and	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	81	4 6	2	1	1
and	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	83	3	2	1	1
and	mem8,imm8	SF,ZF,OF,CF,PF,AF	80	3+	7	3	3
and	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	81	4+ 6+	7	3	3
and	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	83	3+	7	3	3
and	reg8,reg8	SF,ZF,OF,CF,PF,AF	22	2	2	1	1
and	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	23	2	2	1	1
and	reg8,mem8	SF,ZF,OF,CF,PF,AF	22	2+	6	2	2
and	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	23	2+	6	2	2

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
and	mem8,reg8	SF,ZF,OF,CF,PF,AF	20	2+	7	3	3
and	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	21	2+	7	3	3
call	rel32	none	E8	5	7+	3	1
call	reg32 (near indirect)	none	FF	2	7+	5	2
call	mem32 (near indirect)	none	FF	2+	10+	5	2
call	far direct	none	9A	7	17+	18	4
call	far indirect	none	FF	6	22+	17	5
cbw	none	none	98	1	3	3	3
cdq	none	none	99	1	2	3	2
clc	none	CF	F8	1	2	2	2
cld	none	DF	FC	1	2	2	2
cmc	none	CF	F5	1	2	2	2
cmp	AL,imm8	SF,ZF,OF,CF,PF,AF	3C	2	2	1	1
cmp	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3D	3 5	2	1	1
cmp	reg8,imm8	SF,ZF,OF,CF,PF,AF	80	3	2	1	1
cmp	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	81	4 6	2	1	1
cmp	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	83	3	2	1	1
cmp	mem8,imm8	SF,ZF,OF,CF,PF,AF	80	3+	5	2	2
cmp	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	81	4+ 6+	5	2	2
cmp	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	83	3+	5	2	2
cmp	reg8,reg8	SF,ZF,OF,CF,PF,AF	38	2	2	1	1
cmp	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	3B	2	2	1	1
cmp	reg8,mem8	SF,ZF,OF,CF,PF,AF	3A	2+	6	2	2
cmp	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	3B	2+	6	2	2
cmp	mem8,reg8	SF,ZF,OF,CF,PF,AF	38	2+	5	2	2
cmp	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	39	2+	5	2	2

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
cmpsb	none	none	A6	1	10	8	5
cmpsw cmpsd	none	none	A7	1	10	8	5
cwd	none	none	99	1	2	3	2
cwde	none	none	98	1	3	3	3
daa	none	SF,ZF,PF,AF OF ?	27	1	4	2	3
das	none	SF,ZF,PF,AF OF ?	2F	1	4	2	3
dec	reg8		FE	2	2	1	1
dec	AX EAX	SF,ZF,OF,PF,AF	48	1	2	1	1
dec	CX ECX	SF,ZF,OF,PF,AF	49	1	2	1	1
dec	DX EDX	SF,ZF,OF,PF,AF	4A	1	2	1	1
dec	BX EBX	SF,ZF,OF,PF,AF	4B	1	2	1	1
dec	SP ESP	SF,ZF,OF,PF,AF	4C	1	2	1	1
dec	BP EBP	SF,ZF,OF,PF,AF	4D	1	2	1	1
dec	SI ESI	SF,ZF,OF,PF,AF	4E	1	2	1	1
dec	DI EDI	SF,ZF,OF,PF,AF	4F	1	2	1	1
dec	mem8	SF,ZF,OF,PF,AF	FE	2+	6	3	3
dec	mem16 mem32	SF,ZF,OF,PF,AF	FF	2+	6	3	3
div	reg8	SF,ZF,OF,PF,AF ?	F6	2	14	16	17
div	reg16 reg32	SF,ZF,OF,PF,AF ?	F7	2	22 38	24 40	25 41
div	mem8	SF,ZF,OF,PF,AF ?	F6	2+	17	16	17
div	mem16 mem32	SF,ZF,OF,PF,AF ?	F7	2+	25 41	24 40	25 41
idiv	reg8	SF,ZF,OF,PF,AF ?	F6	2	19	19	22
idiv	reg16 reg32	SF,ZF,OF,PF,AF ?	F7	2	27 43	27 43	30 48

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
idiv	mem8	SF,ZF,OF,PF,AF ?	F6	2+	22	20	22
idiv	mem16 mem32	SF,ZF,OF,PF,AF ?	F7	2+	30 46	28 44	30 48
imul	reg8	OF,CF SF,ZF, PF,AF ?	F6	2	9-14	13-18	11
imul	reg16 reg32	OF,CF SF,ZF, PF,AF ?	F7	2	9-22 9-38	13-26 13-42	11 10
imul	mem8	OF,CF SF,ZF, PF,AF ?	F6	2+	12-17	13-18	11
imul	mem16 mem32	OF,CF SF,ZF, PF,AF ?	F7	2+	12-25 12-41	13-26 13-42	11 10
imul	reg16,reg16 reg32,reg32	OF,CF SF,ZF, PF,AF ?	0F AF	3	9-22 9-38	13-26 13-42	11 10
imul	reg16,mem16 reg32,mem32	OF,CF SF,ZF, PF,AF ?	0F AF	3+	12-25 12-41	13-26 13-42	11 10
imul	reg16,imm8 reg32,imm8	OF,CF SF,ZF, PF,AF ?	6B	3	9-14	13-18	10
imul	mem16 mem32	OF,CF SF,ZF, PF,AF ?	F7	4 6	9-22 9-38	13-26 13-42	11 10
imul	reg16,reg16,imm8 reg32,reg32,imm8	OF,CF SF,ZF, PF,AF ?	6B	3	9-14	13-18	10
imul	reg16,reg16,imm16 reg32,reg32,imm32	OF,CF SF,ZF, PF,AF ?	69	4 6	9-22 9-38	13-26 13-42	10 10
imul	reg16,mem16,imm8 reg32,mem32,imm8	OF,CF SF,ZF, PF,AF ?	6B	3+	9-17	13-18	10
imul	reg16,mem16,imm16 reg32,mem32,imm32	OF,CF SF,ZF, PF,AF ?	69	4+ 6+	12-25 12-41	13-26 13-42	10 10
inc	reg8	SF,ZF,OF,PF,AF	FE	2	2	1	1
inc	AX EAX	SF,ZF,OF,PF,AF	40	1	2	1	1
inc	CX ECX	SF,ZF,OF,PF,AF	41	1	2	1	1
inc	DX EDX	SF,ZF,OF,PF,AF	42	1	2	1	1
inc	BX EBX	SF,ZF,OF,PF,AF	43	1	2	1	1
inc	SP ESP	SF,ZF,OF,PF,AF	44	1	2	1	1

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
inc	BP EBP	SF,ZF,OF,PF,AF	45	1	2	1	1
inc	SI ESI	SF,ZF,OF,PF,AF	47	1	2	1	1
inc	DI EDI	SF,ZF,OF,PF,AF	48	1	2	1	1
inc	mem8	SF,ZF,OF,PF,AF	FE	2+	6	3	3
inc	mem16 mem32	SF,ZF,OF,PF,AF	FF	2+	6	3	3
ja jnbe	rel8	none	77	7+,3	3,1	1	2
ja jnbe	rel32	none	0F 87	7+,3	3,1	1	6
jae jnb	rel8	none	73	7+,3	3,1	1	2
jae jnb	rel32	none	0F 83	7+,3	3,1	1	6
jb jnae	rel8	none	72	7+,3	3,1	1	2
jb jnae	rel32	none	0F 82	7+,3	3,1	1	6
jbe jna	rel8	none	76	7+,3	3,1	1	2
jbe jna	rel32	none	0F 86	7+,3	3,1	1	6
jc	rel8	none	72	7+,3	3,1	1	2
jc	rel32	none	0F 82	7+,3	3,1	1	6
je jz	rel8	none	74	7+,3	3,1	1	2
je jz	rel32	none	0F 84	7+,3	3,1	1	6
jecxz	rel8	none	E3			6,5	2
jg jnle	rel8	none	7F	7+,3	3,1	1	2
jg jnle	rel32	none	0F 8F	7+,3	3,1	1	6
jge jnl	rel8	none	7D	7+,3	3,1	1	2

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
jge jnl	rel32	none	0F 8D	7+,3	3,1	1	6
jl jnge	rel8	none	7C	7+,3	3,1	1	2
jl jnge	rel32	none	0F 8C	7+,3	3,1	1	6
jle jng	rel8	none	7E	7+,3	3,1	1	2
jle jng	rel32	none	0F 8E	7+,3	3,1	1	6
jmp	rel8	none	EB	2	7+	3	1
jmp	rel32	none	E9	5	7+	3	1
jmp	reg32	none	FF	2	10+	5	2
jmp	mem32	none	FF	2+	10+	5	2
jnc	rel8	none	73	7+,3	3,1	1	2
jnc	rel32	none	0F 83	7+,3	3,1	1	6
jne jnz	rel8	none	75	7+,3	3,1	1	2
jne jnz	rel32	none	0F 85	7+,3	3,1	1	6
jno	rel8	none	71	7+,3	3,1	1	2
jno	rel32	none	0F 81	7+,3	3,1	1	6
jnp jpo	rel8	none	7B	7+,3	3,1	1	2
jnp jpo	rel32	none	0F 8B	7+,3	3,1	1	6
jns	rel8	none	79	7+,3	3,1	1	2
jns	rel32	none	0F 89	7+,3	3,1	1	6
jo	rel8	none	70	7+,3	3,1	1	2
jo	rel32	none	0F 80	7+,3	3,1	1	6
jp jpe	rel8	none	7A	7+,3	3,1	1	2
jp jpe	rel32	none	0F 8A	7+,3	3,1	1	6
js	rel8	none	78	7+,3	3,1	1	2
js	rel32	none	0F 88	7+,3	3,1	1	6
lea	reg32,mem32	none	8D	2+	2	1	1
lodsb	none	none	AC	1	5	5	2

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
lodsw lodsd	none	none	AD	1	5	5	2
loop	none	none	E2	11+	6,7	5,6	2
loope loopz	none	none	E1	11+	6,9	7,8	2
loopne loopnz	none	none	E0	11+	6,9	7,8	2
mov	AL, imm8	none	B0	2	2	1	1
mov	CL, imm8	none	B1	2	2	1	1
mov	DL, imm8	none	B2	2	2	1	1
mov	BL, imm8	none	B3	2	2	1	1
mov	AH, imm8	none	B4	2	2	1	1
mov	CH, imm8	none	B5	2	2	1	1
mov	DH, imm8	none	B6	2	2	1	1
mov	BH, imm8	none	B7	2	2	1	1
mov	AX, imm16 EAX, imm32	none	B8	3 5	2	1	1
mov	CX, imm16 ECX, imm32	none	B9	3 5	2	1	1
mov	DX, imm16 EDX, imm32	none	BA	3 5	2	1	1
mov	BX, imm16 EBX, imm32	none	BB	3 5	2	1	1
mov	SP, imm16 ESP, imm32	none	BC	3 5	2	1	1
mov	BP, imm16 EPB, imm32	none	BD	3 5	2	1	1
mov	SI, imm16 ESI, imm32	none	BE	3 5	2	1	1
mov	DI, imm16 EDI, imm32	none	BF	3 5	2	1	1
mov	mem8, imm8	none	C6	3+	2	1	1
mov	mem16,imm16 mem32,imm32	none	C7	4+ 6+	2	1	1
mov	reg8,reg8	none	8A	2	2	1	1
mov	reg16,reg16 reg32,reg32	none	8B	2	2	1	1
mov	AL, direct	none	A0	5	4	1	1

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
mov	AX, direct EAX, direct	none	A1	5	4	1	1
mov	reg8,mem8	none	8A	2+	4	1	1
mov	reg16,mem16 reg32,mem32	none	8B	2+	4	1	1
mov	mem8,reg8	none	88	2+	2	1	1
mov	mem16,reg16 mem32,reg32	none	89	2+	2	1	1
mov	direct ,AL	none	A2	5	2	1	1
mov	direct, AX direct, EAX	none	A3	5	2	1	1
mov	sreg, reg16	none	8E	2	2	3	1
mov	reg16, sreg	none	8C	2	2	3	1
mov	sreg,mem16	none	8E	2+	2	3*	2*
mov	mem16,sreg	none	8C	2+	2	3	1
movsb	none	none	A4	1	7	7	4
movsw movsd	none	none	A5	1	7	7	4
movsx	reg16,reg8 reg32,reg8	none	0F BE	3	3	3	3
movsx	reg16,mem8 reg32,mem8	none	0F BE	3+	6	3	3
movsx	reg32,reg16	none	0F BF	3	3	3	3
movsx	reg32,mem16	none	0F BF	3+	6	3	3
movzx	reg16,reg8 reg32,reg8	none	0F B6	3	3	3	3
movzx	reg16,mem8 reg32,mem8	none	0F B6	3+	6	3	3
movzx	reg32,reg16	none	0F B7	3	3	3	3
movzx	reg32,mem16	none	0F B7	3+	6	3	3
mul	reg8	OF,CF SF,ZF, PF,AF ?	F6	2	9-14	13-18	11
mul	reg16 reg32	OF,CF SF,ZF, PF,AF ?	F7	2	9-22 9-38	13-26 13-42	11 10
mul	mem8	OF,CF SF,ZF, PF,AF ?	F6	2+	12-17	13-18	11
mul	mem16 mem32	OF,CF SF,ZF, PF,AF ?	F7	2+	12-25 12-41	13-26 13-42	11 10

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
neg	reg8	SF,ZF,OF,CF,PF,AF	F6	2	2	1	1
neg	reg16 reg32	SF,ZF,OF,CF,PF,AF	F7	2	2	1	1
neg	mem8	SF,ZF,OF,CF,PF,AF	F6	2+	2	1	1
neg	mem16 mem32	SF,ZF,OF,CF,PF,AF	F7	2+	2	1	1
not	reg8	none	F6	2	2	1	1
not	reg16 reg32	none	F7	2	2	1	1
not	mem8	none	F6	2+	6	3	3
not	mem16 mem32	none	F7	2+	6	3	3
or	AL,imm8	SF,ZF,OF,CF,PF,AF	0C	2	2	1	1
or	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	0D	3 5	2	1	1
or	reg8,imm8	SF,ZF,OF,CF,PF,AF	80	3	2	1	1
or	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	81	4 6	2	1	1
or	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	83	3	2	1	1
or	mem8,imm8	SF,ZF,OF,CF,PF,AF	80	3+	7	3	3
or	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	81	4+ 6+	7	3	3
or	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	83	3+	7	3	3
or	reg8,reg8	SF,ZF,OF,CF,PF,AF	0A	2	2	1	1
or	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	0B	2	2	1	1
or	reg8,mem8	SF,ZF,OF,CF,PF,AF	0A	2+	6	2	2
or	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	0B	2+	6	2	2
or	mem8,reg8	SF,ZF,OF,CF,PF,AF	08	2+	7	3	3
or	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	09	2+	7	3	3
pop	AX EAX	none	58	1	4	1	1
pop	CX ECX	none	59	1	4	1	1

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
pop	DX EDX	none	5A	1	4	1	1
pop	BX EBX	none	5B	1	4	1	1
pop	SP ESP	none	5C	1	4	1	1
pop	BP EBP	none	5D	1	4	1	1
pop	SI ESI	none	5E	1	4	1	1
pop	DI EDI	none	5F	1	4	1	1
pop	DS	none	1F	1	7	3	3
pop	ES	none	07	1	7	3	3
pop	SS	none	17	1	7	3	3
pop	FS	none	0F A1	2	7	3	3
pop	GS	none	0F A9	2	7	3	3
pop	mem16 mem32	none	8F	2+	5	6	3
popa popad	none	none	61	1	24	9	5
popf popfd	none	none	9D	1	5	9	4
push	AX EAX	none	50	1	2	1	1
push	CX ECX	none	51	1	2	1	1
push	DX EDX	none	52	1	2	1	1
push	BX EBX	none	53	1	2	1	1
push	SP ESP	none	54	1	2	1	1
push	BP EBP	none	55	1	2	1	1
push	SI ESI	none	56	1	2	1	1
push	DI EDI	none	57	1	2	1	1

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
push	CS	none	0E	1	2	3	1
push	DS	none	1E	1	2	3	1
push	ES	none	06	1	2	3	1
push	SS	none	16	1	2	3	1
push	FS	none	0F A0	2	2	3	1
push	GS	none	0F A8	2	2	3	1
push	mem16 mem32	none	FF	2+	5	4	2
push	imm8	none	6A	2	2	1	1
push	imm16 imm32	none	68	3 5	2	1	1
pusha pushad	none	none	60	1	18	11	5
pushf pushfd	none	none	9C	1	4	4	3
rep repz repe	none (string instruction prefix)	none	F3	1			
rep movsb	none	none	F3 A4	2	7+4n	12+3n	13+4n
rep movsw rep movsd	none	none	F3 A5	2	7+4n	12+3n	13+4n
rep stosb	none	none	F3 A6	2	5+5n	7+4n	9n
rep stosw rep stosd	none	none	F3 A7	2	5+5n	7+4n	9n
repe cmpsb	none	none	F3 A6	2	5+9n	7+7n	9+4n
repe cmpsw repe cmpsd	none	none	F3 A7	2	5+9n	7+7n	9+4n
repe scasb	none	none	F3 AE	2	5+8n	7+5n	9+4n
repe scasw repe scasd	none	none	F3 AF	2	5+8n	7+5n	9+4n

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
repne cmpsb	none	none	F2 A6	2	5+9n	7+7n	9+4n
repne cmpsw repne cmpsd	none	none	F2 A7	2	5+9n	7+7n	9+4n
repne scasb	none	none	F2 AE	2	5+8n	7+5n	9+4n
repne scasw repne scasd	none	none	F2 AF	2	5+8n	7+5n	9+4n
repnz repne	none (string instruction prefix)	none	F2	1			
ret (far)	none	none	CB	1	18+	13	4
ret (far)	imm16	none	CA	3	18+	14	4
ret (near)	none	none	C3	1	10+	5	2
ret (near)	imm16	none	C2	3	10+	5	3
rol ror	reg8	SF,ZF,OF,CF,PF AF ?	D0	2	3	3	1
rol ror	reg16 reg32	SF,ZF,OF,CF,PF AF ?	D1	2	3	3	1
rol ror	mem8	SF,ZF,OF,CF,PF AF ?	D0	2+	7	4	3
rol ror	reg16 reg32	SF,ZF,OF,CF,PF AF ?	D1	2+	7	4	3
rol ror	reg8, imm8	SF,ZF,OF,CF,PF AF ?	C0	3	3	2	1
rol ror	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF AF ?	C1	3	3	2	1
rol ror	mem8, imm8	SF,ZF,OF,CF,PF AF ?	C0	3+	7	4	3
rol ror	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF AF ?	C1	3+	7	4	3
rol ror	reg8, CL	SF,ZF,OF,CF,PF AF ?	D2	2	3	2	1
rol ror	reg16,CL reg32,CL	SF,ZF,OF,CF,PF AF ?	D3	2	3	2	1

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
rol ror	mem8, CL	SF,ZF,OF,CF,PF AF ?	D2	2+	7	4	4
rol ror	mem16,CL mem32,CL	SF,ZF,OF,CF,PF AF ?	D3	2+	7	4	4
sbb	AL,imm8	SF,ZF,OF,CF,PF,AF	1C	2	2	1	1
sbb	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	1D	3 5	2	1	1
sbb	reg8,imm8	SF,ZF,OF,CF,PF,AF	80	3	2	1	1
sbb	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	81	4 6	2	1	1
sbb	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	83	3	2	1	1
sbb	mem8,imm8	SF,ZF,OF,CF,PF,AF	80	3+	7	3	3
sbb	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	81	4+ 6+	7	3	3
sbb	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	83	3+	7	3	3
sbb	reg8,reg8	SF,ZF,OF,CF,PF,AF	1A	2	2	1	1
sbb	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	1B	2	2	1	1
sbb	reg8,mem8	SF,ZF,OF,CF,PF,AF	1A	2+	6	2	2
sbb	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	1B	2+	6	2	2
sbb	mem8,reg8	SF,ZF,OF,CF,PF,AF	18	2+	7	3	3
sbb	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	19	2+	7	3	3
scasb	none	none	AE	1	7	6	4
scasw scasd	none	none	AE	1	7	6	4
shl/sal shr sar	reg8,1	SF,ZF,OF,CF,PF AF ?	D0	2	3	3	1
shl/sal shr sar	reg16,1 reg32,1	SF,ZF,OF,CF,PF AF ?	D1	2	3	3	1
shl/sal shr sar	mem8,1	SF,ZF,OF,CF,PF AF ?	D0	2+	7	4	3
shl/sal	reg16,1	SF,ZF,OF,CF,PF	D1	2+	7	4	3

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
shr sar	reg32,1	AF ?					
shl/sal shr sar	reg8, imm8	SF,ZF,OF,CF,PF AF ?	C0	3	3	2	1
shl/sal shr sar	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF AF ?	C1	3	3	2	1
shl/sal shr sar	mem8, imm8	SF,ZF,OF,CF,PF AF ?	C0	3+	7	4	3
shl/sal shr sar	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF AF ?	C1	3+	7	4	3
shl/sal shr sar	reg8, CL	SF,ZF,OF,CF,PF AF ?	D2	2	3	2	1
shl/sal shr sar	reg16,CL reg32,CL	SF,ZF,OF,CF,PF AF ?	D3	2	3	2	1
shl/sal shr sar	mem8, CL	SF,ZF,OF,CF,PF AF ?	D2	2+	7	4	4
shl/sal shr sar	mem16,CL mem32,CL	SF,ZF,OF,CF,PF AF ?	D3	2+	7	4	4
shld	reg16,reg16,imm8 reg32,reg32,imm8	SF,ZF,CF,PF OF,AF ?	0F 04	4	3	2	4
shld	mem16,reg16,imm8 mem32,reg32,imm8	SF,ZF,CF,PF OF,AF ?	0F 04	4+	7	4	4
shld	reg16,reg16,CL reg32,reg32,CL	SF,ZF,CF,PF OF,AF ?	0F 05	3	3	3	4
shld	mem16,reg16,CL mem32,reg32,CL	SF,ZF,CF,PF OF,AF ?	0F 05	3+	7	4	5
shrd	reg16,reg16,imm8 reg32,reg32,imm8	SF,ZF,CF,PF OF,AF ?	0F AC	4	3	2	4
shrd	mem16,reg16,imm8 mem32,reg32,imm8	SF,ZF,CF,PF OF,AF ?	0F AC	4+	7	4	4
shrd	reg16,reg16,CL reg32,reg32,CL	SF,ZF,CF,PF OF,AF ?	0F AD	3	3	3	4

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
shrd	mem16,reg16,CL mem32,reg32,CL	SF,ZF,CF,PF OF,AF ?	0F AD	3+	7	4	5
stc	none	CF	F9	1	2	2	2
std	none	DF	FD	1	2	2	2
stosb	none	none	AA	1	4	5	3
stosw stosd	none	none	AB	1	4	5	3
sub	AL,imm8	SF,ZF,OF,CF,PF,AF	2C	2	2	1	1
sub	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	2D	3 5	2	1	1
sub	reg8,imm8	SF,ZF,OF,CF,PF,AF	80	3	2	1	1
sub	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	81	4 6	2	1	1
sub	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	83	3	2	1	1
sub	mem8,imm8	SF,ZF,OF,CF,PF,AF	80	3+	7	3	3
sub	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	81	4+ 6+	7	3	3
sub	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	83	3+	7	3	3
sub	reg8,reg8	SF,ZF,OF,CF,PF,AF	2A	2	2	1	1
sub	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	2B	2	2	1	1
sub	reg8,mem8	SF,ZF,OF,CF,PF,AF	2A	2+	6	2	2
sub	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	2B	2+	6	2	2
sub	mem8,reg8	SF,ZF,OF,CF,PF,AF	28	2+	7	3	3
sub	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	29	2+	7	3	3
test	AL,imm8	SF,ZF,OF,CF,PF,AF	A8	2	2	1	1
test	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	A9	3 5	2	1	1
test	reg8,imm8	SF,ZF,OF,CF,PF,AF	F6	3	2	1	1
test	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	F7	4 6	2	1	1
test	mem8,imm8	SF,ZF,OF,CF,PF,AF	F6	3+	5	2	2
test	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	F7	4+ 6+	5	2	2

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
test	reg8,reg8	SF,ZF,OF,CF,PF,AF	84	2	2	1	1
test	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	85	2	2	1	1
test	mem8,reg8	SF,ZF,OF,CF,PF,AF	84	2+	5	2	2
test	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	85	2+	5	2	2
xchg	AX, CX EAX, ECX	none	91	1	3	3	2
xchg	AX, DX EAX, EDX	none	92	1	3	3	2
xchg	AX, BX EAX, EBX	none	93	1	3	3	2
xchg	AX, SP EAX, ESP	none	94	1	3	3	2
xchg	AX, BP EAX, EBP	none	95	1	3	3	2
xchg	AX, SI EAX, ESI	none	96	1	3	3	2
xchg	AX, DI EAX, EDI	none	97	1	3	3	2
xchg	reg8,reg8	none	86	2	3	3	3
xchg	reg8,mem8	none	86	2+	5	5	3
xchg	reg16,reg16 reg32,mem32	none	87	2	3	3	3
xchg	reg16,mem16 reg32,mem32	none	87	2+	5	5	3
xlat	none	none	D7	1	5	4	4
xor	AL,imm8	SF,ZF,OF,CF,PF,AF	34	2	2	1	1
xor	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	35	3 5	2	1	1
xor	reg8,imm8	SF,ZF,OF,CF,PF,AF	80	3	2	1	1
xor	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	81	4 6	2	1	1
xor	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	83	3	2	1	1
xor	mem8,imm8	SF,ZF,OF,CF,PF,AF	80	3+	7	3	3
xor	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	81	4+ 6+	7	3	3

（续）

助记符	操作数	受影响的标志位	操作码	字节数	时钟周期数		
					386	486	Pentium
xor	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	83	3+	7	3	3
xor	reg8,reg8	SF,ZF,OF,CF,PF,AF	32	2	2	1	1
xor	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	33	2	2	1	1
xor	reg8,mem8	SF,ZF,OF,CF,PF,AF	32	2+	6	2	2
xor	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	33	2+	6	2	2
xor	mem8,reg8	SF,ZF,OF,CF,PF,AF	30	2+	7	3	3
xor	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	31	2+	7	3	3

* timing varies

附录E　80x86指令（带操作码）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
00	add	mem8,reg8	SF,ZF,OF,CF,PF,AF	2+	7	3	3
01	add	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	2+	7	3	3
02	add	reg8,reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
02	add	reg8,mem8	SF,ZF,OF,CF,PF,AF	2+	6	2	2
03	add	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	2	2	1	1
03	add	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	2+	6	2	2
04	add	AL,imm8	SF,ZF,OF,CF,PF,AF	2	2	1	1
05	add	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3 5	2	1	1
06	push	ES	none	1	2	3	1
07	pop	ES	none	1	7	3	3
08	or	mem8,reg8	SF,ZF,OF,CF,PF,AF	2+	7	3	3
09	or	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	2+	7	3	3
0A	or	reg8,reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
0A	or	reg8,mem8	SF,ZF,OF,CF,PF,AF	2+	6	2	2
0B	or	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	2	2	1	1
0B	or	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	2+	6	2	2
0C	or	AL,imm8	SF,ZF,OF,CF,PF,AF	2	2	1	1
0D	or	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3 5	2	1	1
0E	push	CS	none	1	2	3	1
0F 04	shld	reg16,reg16,imm8 reg32,reg32,imm8	SF,ZF,CF,PF OF,AF ?	4	3	2	4
0F 04	shld	mem16,reg16,imm8 mem32,reg32,imm8	SF,ZF,CF,PF OF,AF ?	4+	7	4	4
0F 05	shld	reg16,reg16,CL reg32,reg32,CL	SF,ZF,CF,PF OF,AF ?	3	3	3	4
0F 05	shld	mem16,reg16,CL mem32,reg32,CL	SF,ZF,CF,PF OF,AF ?	3+	7	4	5

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
0F 80	jo	rel32	none	7+,3	3,1	1	6
0F 81	jno	rel32	none	7+,3	3,1	1	6
0F 82	jb jnae	rel32	none	7+,3	3,1	1	6
0F 82	jc	rel32	none	7+,3	3,1	1	6
0F 83	jae jnb	rel32	none	7+,3	3,1	1	6
0F 83	jnc	rel32	none	7+,3	3,1	1	6
0F 84	je jz	rel32	none	7+,3	3,1	1	6
0F 85	jne jnz	rel32	none	7+,3	3,1	1	6
0F 86	jbe jna	rel32	none	7+,3	3,1	1	6
0F 87	ja jnbe	rel32	none	7+,3	3,1	1	6
0F 88	js	rel32	none	7+,3	3,1	1	6
0F 89	jns	rel32	none	7+,3	3,1	1	6
0F 8A	jp jpe	rel32	none	7+,3	3,1	1	6
0F 8B	jnp jpo	rel32	none	7+,3	3,1	1	6
0F 8C	jl jnge	rel32	none	7+,3	3,1	1	6
0F 8D	jge jnl	rel32	none	7+,3	3,1	1	6
0F 8E	jle jng	rel32	none	7+,3	3,1	1	6
0F 8F	jg jnle	rel32	none	7+,3	3,1	1	6
0F A0	push	FS	none	2	2	3	1
0F A1	pop	FS	none	2	7	3	3
0F A8	push	GS	none	2	2	3	1
0F A9	pop	GS	none	2	7	3	3
0F AC	shrd	reg16,reg16,imm8 reg32,reg32,imm8	SF,ZF,CF,PF OF,AF ?	4	3	2	4
0F AC	shrd	mem16,reg16,imm8 mem32,reg32,imm8	SF,ZF,CF,PF OF,AF ?	4+	7	4	4
0F AD	shrd	reg16,reg16,CL reg32,reg32,CL	SF,ZF,CF,PF OF,AF ?	3	3	3	4

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
0F AD	shrd	mem16,reg16,CL mem32,reg32,CL	SF,ZF,CF,PF OF,AF ?	3+	7	4	5
0F AF	imul	reg16,reg16 reg32,reg32	OF,CF SF,ZF, PF,AF ?	3	9-22 9-38	13-26 13-42	11 10
0F AF	imul	reg16,mem16 reg32,mem32	OF,CF SF,ZF, PF,AF ?	3+	12-25 12-41	13-26 13-42	11 10
0F B6	movzx	reg16,reg8 reg32,reg8	none	3	3	3	3
0F B6	movzx	reg16,mem8 reg32,mem8	none	3+	6	3	3
0F B7	movzx	reg32,reg16	none	3	3	3	3
0F B7	movzx	reg32,mem16	none	3+	6	3	3
0F BE	movsx	reg16,reg8 reg32,reg8	none	3	3	3	3
0F BE	movsx	reg16,mem8 reg32,mem8	none	3+	6	3	3
0F BF	movsx	reg32,reg16	none	3	3	3	3
0F BF	movsx	reg32,mem16	none	3+	6	3	3
10	adc	mem8,reg8	SF,ZF,OF,CF,PF,AF	2+	7	3	3
11	adc	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	2+	7	3	3
12	adc	reg8,reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
12	adc	reg8,mem8	SF,ZF,OF,CF,PF,AF	2+	6	2	2
13	adc	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	2	2	1	1
13	adc	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	2+	6	2	2
14	adc	AL,imm8	SF,ZF,OF,CF,PF,AF	2	2	1	1
15	adc	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3 5	2	1	1
16	push	SS	none	1	2	3	1
17	pop	SS	none	1	7	3	3
18	sbb	mem8,reg8	SF,ZF,OF,CF,PF,AF	2+	7	3	3
19	sbb	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	2+	7	3	3
1A	sbb	reg8,reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
1A	sbb	reg8,mem8	SF,ZF,OF,CF,PF,AF	2+	6	2	2
1B	sbb	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	2	2	1	1
1B	sbb	reg16,mem16	SF,ZF,OF,CF,PF,AF	2+	6	2	2

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
		reg32,mem32					
1C	sbb	AL,imm8	SF,ZF,OF,CF,PF,AF	2	2	1	1
1D	sbb	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3 5	2	1	1
1E	push	DS	none	1	2	3	1
1F	pop	DS	none	1	7	3	3
20	and	mem8,reg8	SF,ZF,OF,CF,PF,AF	2+	7	3	3
21	and	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	2+	7	3	3
22	and	reg8,reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
22	and	reg8,mem8	SF,ZF,OF,CF,PF,AF	2+	6	2	2
23	and	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	2	2	1	1
23	and	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	2+	6	2	2
24	and	AL,imm8	SF,ZF,OF,CF,PF,AF	2	2	1	1
25	and	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3 5	2	1	1
27	daa	none	SF,ZF,PF,AF OF ?	1	4	2	3
28	sub	mem8,reg8	SF,ZF,OF,CF,PF,AF	2+	7	3	3
29	sub	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	2+	7	3	3
2A	sub	reg8,reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
2A	sub	reg8,mem8	SF,ZF,OF,CF,PF,AF	2+	6	2	2
2B	sub	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	2	2	1	1
2B	sub	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	2+	6	2	2
2C	sub	AL,imm8	SF,ZF,OF,CF,PF,AF	2	2	1	1
2D	sub	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3 5	2	1	1
2F	das	none	SF,ZF,PF,AF OF ?	1	4	2	3
30	xor	mem8,reg8	SF,ZF,OF,CF,PF,AF	2+	7	3	3
31	xor	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	2+	7	3	3
32	xor	reg8,reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
32	xor	reg8,mem8	SF,ZF,OF,CF,PF,AF	2+	6	2	2
33	xor	reg16,reg16	SF,ZF,OF,CF,PF,AF	2	2	1	1

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
		reg32,reg32					
33	xor	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	2+	6	2	2
34	xor	AL,imm8	SF,ZF,OF,CF,PF,AF	2	2	1	1
35	xor	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3 5	2	1	1
37	aaa	none	AF,CF SF,ZF,OF,PF ?	1	4	3	3
38	cmp	reg8,reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
38	cmp	mem8,reg8	SF,ZF,OF,CF,PF,AF	2+	5	2	2
39	cmp	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	2+	5	2	2
3A	cmp	reg8,mem8	SF,ZF,OF,CF,PF,AF	2+	6	2	2
3B	cmp	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	2	2	1	1
3B	cmp	reg16,mem16 reg32,mem32	SF,ZF,OF,CF,PF,AF	2+	6	2	2
3C	cmp	AL,imm8	SF,ZF,OF,CF,PF,AF	2	2	1	1
3D	cmp	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3 5	2	1	1
3F	aas	none	AF,CF SF,ZF,OF,PF ?	1	4	3	3
40	inc	AX EAX	SF,ZF,OF,PF,AF	1	2	1	1
41	inc	CX ECX	SF,ZF,OF,PF,AF	1	2	1	1
42	inc	DX EDX	SF,ZF,OF,PF,AF	1	2	1	1
43	inc	BX EBX	SF,ZF,OF,PF,AF	1	2	1	1
44	inc	SP ESP	SF,ZF,OF,PF,AF	1	2	1	1
45	inc	BP EBP	SF,ZF,OF,PF,AF	1	2	1	1
47	inc	SI ESI	SF,ZF,OF,PF,AF	1	2	1	1
48	dec	AX EAX	SF,ZF,OF,PF,AF	1	2	1	1
48	inc	DI EDI	SF,ZF,OF,PF,AF	1	2	1	1
49	dec	CX	SF,ZF,OF,PF,AF	1	2	1	1

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
		ECX					
4A	dec	DX EDX	SF,ZF,OF,PF,AF	1	2	1	1
4B	dec	BX EBX	SF,ZF,OF,PF,AF	1	2	1	1
4C	dec	SP ESP	SF,ZF,OF,PF,AF	1	2	1	1
4D	dec	BP EBP	SF,ZF,OF,PF,AF	1	2	1	1
4E	dec	SI ESI	SF,ZF,OF,PF,AF	1	2	1	1
4F	dec	DI EDI	SF,ZF,OF,PF,AF	1	2	1	1
50	push	AX EAX	none	1	2	1	1
51	push	CX ECX	none	1	2	1	1
52	push	DX EDX	none	1	2	1	1
53	push	BX EBX	none	1	2	1	1
54	push	SP ESP	none	1	2	1	1
55	push	BP EBP	none	1	2	1	1
56	push	SI ESI	none	1	2	1	1
57	push	DI EDI	none	1	2	1	1
58	pop	AX EAX	none	1	4	1	1
59	pop	CX ECX	none	1	4	1	1
5A	pop	DX EDX	none	1	4	1	1
5B	pop	BX EBX	none	1	4	1	1
5C	pop	SP ESP	none	1	4	1	1
5D	pop	BP EBP	none	1	4	1	1

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
5E	pop	SI ESI	none	1	4	1	1
5F	pop	DI EDI	none	1	4	1	1
60	pusha pushad	none	none	1	18	11	5
61	popa popad	none	none	1	24	9	5
68	push	imm16 imm32	none	3 5	2	1	1
69	imul	reg16,reg16,imm16 reg32,reg32,imm32	OF,CF SF,ZF, PF,AF ?	4 6	9-22 9-38	13-26 13-42	10 10
69	imul	reg16,mem16,imm16 reg32,mem32,imm32	OF,CF SF,ZF, PF,AF ?	4+ 6+	12-25 12-41	13-26 13-42	10 10
6A	push	imm8	none	2	2	1	1
6B	imul	reg16,imm8 reg32,imm8	OF,CF SF,ZF, PF,AF ?	3	9-14	13-18	10
6B	imul	reg16,reg16,imm8 reg32,reg32,imm8	OF,CF SF,ZF, PF,AF ?	3	9-14	13-18	10
6B	imul	reg16,mem16,imm8 reg32,mem32,imm8	OF,CF SF,ZF, PF,AF ?	3+	9-17	13-18	10
70	jo	rel8	none	7+,3	3,1	1	2
71	jno	rel8	none	7+,3	3,1	1	2
72	jb jnae	rel8	none	7+,3	3,1	1	2
72	jc	rel8	none	7+,3	3,1	1	2
73	jae jnb	rel8	none	7+,3	3,1	1	2
73	jnc	rel8	none	7+,3	3,1	1	2
74	je jz	rel8	none	7+,3	3,1	1	2
75	jne jnz	rel8	none	7+,3	3,1	1	2
76	jbe jna	rel8	none	7+,3	3,1	1	2
77	ja jnbe	rel8	none	7+,3	3,1	1	2
78	js	rel8	none	7+,3	3,1	1	2
79	jns	rel8	none	7+,3	3,1	1	2
7A	jp jpe	rel8	none	7+,3	3,1	1	2

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
7B	jnp jpo	rel8	none	7+,3	3,1	1	2
7C	jl jnge	rel8	none	7+,3	3,1	1	2
7D	jge jnl	rel8	none	7+,3	3,1	1	2
7E	jle jng	rel8	none	7+,3	3,1	1	2
7F	jg jnle	rel8	none	7+,3	3,1	1	2
80	adc	reg8,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
80	adc	mem8,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
80	add	reg8,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
80	add	mem8,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
80	and	reg8,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
80	and	mem8,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
80	cmp	reg8,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
80	cmp	mem8,imm8	SF,ZF,OF,CF,PF,AF	3+	5	2	2
80	or	reg8,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
80	or	mem8,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
80	sbb	reg8,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
80	sbb	mem8,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
80	sub	reg8,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
80	sub	mem8,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
80	xor	reg8,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
80	xor	mem8,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
81	adc	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	4 6	2	1	1
81	adc	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	4+ 6+	7	3	3
81	add	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	4 6	2	1	1
81	add	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	4+ 6+	7	3	3
81	and	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	4 6	2	1	1
81	and	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	4+ 6+	7	3	3
81	cmp	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	4 6	2	1	1

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
81	cmp	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	4+ 6+	5	2	2
81	or	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	4 6	2	1	1
81	or	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	4+ 6+	7	3	3
81	sbb	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	4 6	2	1	1
81	sbb	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	4+ 6+	7	3	3
81	sub	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	4 6	2	1	1
81	sub	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	4+ 6+	7	3	3
81	xor	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	4 6	2	1	1
81	xor	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	4+ 6+	7	3	3
83	adc	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
83	adc	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
83	add	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
83	add	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
83	and	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
83	and	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
83	cmp	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
83	cmp	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	3+	5	2	2
83	or	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
83	or	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
83	sbb	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
83	sbb	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
83	sub	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
83	sub	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
83	xor	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
83	xor	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF,AF	3+	7	3	3
84	test	reg8,reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
84	test	mem8,reg8	SF,ZF,OF,CF,PF,AF	2+	5	2	2
85	test	reg16,reg16 reg32,reg32	SF,ZF,OF,CF,PF,AF	2	2	1	1
85	test	mem16,reg16 mem32,reg32	SF,ZF,OF,CF,PF,AF	2+	5	2	2
86	xchg	reg8,reg8	none	2	3	3	3
86	xchg	reg8,mem8	none	2+	5	5	3
87	xchg	reg16,reg16	none	2	3	3	3
87	xchg	reg16,mem16	none	2+	5	5	3
88	mov	mem8,reg8	none	2+	2	1	1
89	mov	mem16,reg16 mem32,reg32	none	2+	2	1	1
8A	mov	reg8,reg8	none	2	2	1	1
8A	mov	reg8,mem8	none	2+	4	1	1
8B	mov	reg16,reg16 reg32,reg32	none	2	2	1	1
8B	mov	reg16,mem16 reg32,mem32	none	2+	4	1	1
8C	mov	reg16, sreg	none	2	2	3	1
8C	mov	mem16,sreg	none	2+	2	3	1
8D	lea	reg32,mem32	none	2+	2	1	1
8E	mov	sreg, reg16	none	2	2	3	1
8E	mov	sreg,mem16	none	2+	2	3*	2*
8F	pop	mem16 mem32	none	2+	5	6	3
91	xchg	AX, CX EAX, ECX	none	1	3	3	2
92	xchg	AX, DX EAX, EDX	none	1	3	3	2
93	xchg	AX, BX EAX, EBX	none	1	3	3	2

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
94	xchg	AX, SP EAX, ESP	none	1	3	3	2
95	xchg	AX, BP EAX, EBP	none	1	3	3	2
96	xchg	AX, SI EAX, ESI	none	1	3	3	2
97	xchg	AX, DI EAX, EDI	none	1	3	3	2
98	cbw	none	none	1	3	3	3
98	cwde	none	none	1	3	3	3
99	cdq	none	none	1	2	3	2
99	cwd	none	none	1	2	3	2
9A	call	far direct	none	7	17+	18	4
9C	pushf pushfd	none	none	1	4	4	3
9D	popf popfd	none	none	1	5	9	4
A0	mov	AL, direct	none	5	4	1	1
A1	mov	AX, direct EAX, direct	none	5	4	1	1
A2	mov	direct ,AL	none	5	2	1	1
A3	mov	direct, AX direct, EAX	none	5	2	1	1
A4	movsb	none	none	1	7	7	4
A5	movsw movsd	none	none	1	7	7	4
A6	cmpsb	none	none	1	10	8	5
A7	cmpsw cmpsd	none	none	1	10	8	5
A8	test	AL,imm8	SF,ZF,OF,CF,PF,AF	2	2	1	1
A9	test	AX,imm16 EAX,imm32	SF,ZF,OF,CF,PF,AF	3 5	2	1	1
AA	stosb	none	none	1	4	5	3
AB	stosw stosd	none	none	1	4	5	3
AC	lodsb	none	none	1	5	5	2
AD	lodsw lodsd	none	none	1	5	5	2
AE	scasb	none	none	1	7	6	4
AE	scasw	none	none	1	7	6	4

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
	scasd						
B0	mov	AL, imm8	none	2	2	1	1
B1	mov	CL, imm8	none	2	2	1	1
B2	mov	DL, imm8	none	2	2	1	1
B3	mov	BL, imm8	none	2	2	1	1
B4	mov	AH, imm8	none	2	2	1	1
B5	mov	CH, imm8	none	2	2	1	1
B6	mov	DH, imm8	none	2	2	1	1
B7	mov	BH, imm8	none	2	2	1	1
B8	mov	AX, imm16 EAX, imm32	none	3 5	2	1	1
B9	mov	CX, imm16 ECX, imm32	none	3 5	2	1	1
BA	mov	DX, imm16 EDX, imm32	none	3 5	2	1	1
BB	mov	BX, imm16 EBX, imm32	none	3 5	2	1	1
BC	mov	SP, imm16 ESP, imm32	none	3 5	2	1	1
BD	mov	BP, imm16 EPB, imm32	none	3 5	2	1	1
BE	mov	SI, imm16 ESI, imm32	none	3 5	2	1	1
BF	mov	DI, imm16 EDI, imm32	none	3 5	2	1	1
C0	rol ror	reg8, imm8	SF,ZF,OF,CF,PF AF ?	3	3	2	1
C0	rol ror	mem8, imm8	SF,ZF,OF,CF,PF AF ?	3+	7	4	3
C0	shl/sal shr sar	reg8, imm8	SF,ZF,OF,CF,PF AF ?	3	3	2	1
C0	shl/sal shr sar	mem8, imm8	SF,ZF,OF,CF,PF AF ?	3+	7	4	3
C1	rol ror	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF AF ?	3	3	2	1
C1	rol ror	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF AF ?	3+	7	4	3
C1	shl/sal shr	reg16,imm8 reg32,imm8	SF,ZF,OF,CF,PF AF ?	3	3	2	1

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
	sar						
C1	shl/sal shr sar	mem16,imm8 mem32,imm8	SF,ZF,OF,CF,PF AF ?	3+	7	4	3
C2	ret (near)	imm16	none	3	10+	5	3
C3	ret (near)	none	none	1	10+	5	2
C6	mov	mem8, imm8	none	3+	2	1	1
C7	mov	mem16,imm16 mem32,imm32	none	4+ 6+	2	1	1
CA	ret (far)	imm16	none	3	18+	14	4
CB	ret (far)	none	none	1	18+	13	4
D0	rol ror	reg8	SF,ZF,OF,CF,PF AF ?	2	3	3	1
D0	rol ror	mem8	SF,ZF,OF,CF,PF AF ?	2+	7	4	3
D0	shl/sal shr sar	reg8	SF,ZF,OF,CF,PF AF ?	2	3	3	1
D0	shl/sal shr sar	mem8	SF,ZF,OF,CF,PF AF ?	2+	7	4	3
D1	rol ror	reg16 reg32	SF,ZF,OF,CF,PF AF ?	2	3	3	1
D1	rol ror	reg16 reg32	SF,ZF,OF,CF,PF AF ?	2+	7	4	3
D1	shl/sal shr sar	reg16 reg32	SF,ZF,OF,CF,PF AF ?	2	3	3	1
D1	shl/sal shr sar	reg16 reg32	SF,ZF,OF,CF,PF AF ?	2+	7	4	3
D2	rol ror	reg8, CL	SF,ZF,OF,CF,PF AF ?	2	3	2	1
D2	rol ror	mem8, CL	SF,ZF,OF,CF,PF AF ?	2+	7	4	4
D2	shl/sal shr sar	reg8, CL	SF,ZF,OF,CF,PF AF ?	2	3	2	1
D2	shl/sal shr sar	mem8, CL	SF,ZF,OF,CF,PF AF ?	2+	7	4	4
D3	rol ror	reg16,CL reg32,CL	SF,ZF,OF,CF,PF AF ?	2	3	2	1

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
D3	rol ror	mem16,CL mem32,CL	SF,ZF,OF,CF,PF AF ?	2+	7	4	4
D3	shl/sal shr sar	reg16,CL reg32,CL	SF,ZF,OF,CF,PF AF ?	2	3	2	1
D3	shl/sal shr sar	mem16,CL mem32,CL	SF,ZF,OF,CF,PF AF ?	2+	7	4	4
D4 0A	aam	none	SF,ZF,PF OF,AF,CF ?	2	17	15	18
D5 0A	aad	none	SF,ZF,PF OF,AF,CF ?	2	19	14	10
D7	xlat	none	none	1	5	4	4
E0	loopne loopnz	none	none	11+	6,9	7,8	2
E1	loope loopz	none	none	11+	6,9	7,8	2
E2	loop	none	none	11+	6,7	5,6	2
E3	jecxz	rel8	none			6,5	2
E8	call	rel32	none	5	7+	3	1
E9	jmp	rel32	none	5	7+	3	1
EB	jmp	rel8	none	2	7+	3	1
F2	repnz repne	none (string instruction prefix)	none	1			
F2 A6	repne cmpsb	none	none	2	5+9n	7+7n	9+4n
F2 A7	repne cmpsw repne cmpsd	none	none	2	5+9n	7+7n	9+4n
F2 AE	repne scasb	none	none	2	5+8n	7+5n	9+4n
F2 AF	repne scasw repne scasd	none	none	2	5+8n	7+5n	9+4n
F3	rep repz repe	none (string instruction prefix)	none	1			
F3 A4	rep movsb	none	none	2	7+4n	12+3n	13+4n

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
F3 A5	rep movsw rep movsd	none	none	2	7+4n	12+3n	13+4n
F3 A6	rep stosb	none	none	2	5+5n	7+4n	9n
F3 A6	repe cmpsb	none	none	2	5+9n	7+7n	9+4n
F3 A7	rep stosw rep stosd	none	none	2	5+5n	7+4n	9n
F3 A7	repe cmpsw repe cmpsd	none	none	2	5+9n	7+7n	9+4n
F3 AE	repe scasb	none	none	2	5+8n	7+5n	9+4n
F3 AF	repe scasw repe scasd	none	none	2	5+8n	7+5n	9+4n
F5	cmc	none	CF	1	2	2	2
F6	div	reg8	SF,ZF,OF,PF,AF ?	2	14	16	17
F6	div	mem8	SF,ZF,OF,PF,AF ?	2+	17	16	17
F6	idiv	reg8	SF,ZF,OF,PF,AF ?	2	19	19	22
F6	idiv	mem8	SF,ZF,OF,PF,AF ?	2+	22	20	22
F6	imul	reg8	OF,CF SF,ZF, PF,AF ?	2	9-14	13-18	11
F6	imul	mem8	OF,CF SF,ZF, PF,AF ?	2+	12-17	13-18	11
F6	mul	reg8	OF,CF SF,ZF, PF,AF ?	2	9-14	13-18	11
F6	mul	mem8	OF,CF SF,ZF, PF,AF ?	2+	12-17	13-18	11
F6	neg	reg8	SF,ZF,OF,CF,PF,AF	2	2	1	1
F6	neg	mem8	SF,ZF,OF,CF,PF,AF	2+	2	1	1
F6	not	reg8	none	2	2	1	1
F6	not	mem8	none	2+	6	3	3
F6	test	reg8,imm8	SF,ZF,OF,CF,PF,AF	3	2	1	1
F6	test	mem8,imm8	SF,ZF,OF,CF,PF,AF	3+	5	2	2
F7	div	reg16 reg32	SF,ZF,OF,PF,AF ?	2	22 38	24 40	25 41
F7	div	mem16 mem32	SF,ZF,OF,PF,AF ?	2+	25 41	24 40	25 41

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
F7	idiv	reg16 reg32	SF,ZF,OF,PF,AF ?	2	27 43	27 43	30 48
F7	idiv	mem16 mem32	SF,ZF,OF,PF,AF ?	2+	30 46	28 44	30 48
F7	imul	reg16 reg32	OF,CF SF,ZF, PF,AF ?	2	9-22 9-38	13-26 13-42	11 10
F7	imul	mem16 mem32	OF,CF SF,ZF, PF,AF ?	2+	12-25 12-41	13-26 13-42	11 10
F7	imul	mem16 mem32	OF,CF SF,ZF, PF,AF ?	4 6	9-22 9-38	13-26 13-42	11 10
F7	mul	reg16 reg32	OF,CF SF,ZF, PF,AF ?	2	9-22 9-38	13-26 13-42	11 10
F7	mul	mem16 mem32	OF,CF SF,ZF, PF,AF ?	2+	12-25 12-41	13-26 13-42	11 10
F7	neg	reg16 reg32	SF,ZF,OF,CF,PF,AF	2	2	1	1
F7	neg	mem16 mem32	SF,ZF,OF,CF,PF,AF	2+	2	1	1
F7	not	reg16 reg32	none	2	2	1	1
F7	not	mem16 mem32	none	2+	6	3	3
F7	test	reg16,imm16 reg32,imm32	SF,ZF,OF,CF,PF,AF	4 6	2	1	1
F7	test	mem16,imm16 mem32,imm32	SF,ZF,OF,CF,PF,AF	4+ 6+	5	2	2
F8	clc	none	CF	1	2	2	2
F9	stc	none	CF	1	2	2	2
FC	cld	none	DF	1	2	2	2
FD	std	none	DF	1	2	2	2
FE	dec	reg8		2	2	1	1
FE	dec	mem8	SF,ZF,OF,PF,AF	2+	6	3	3
FE	inc	reg8	SF,ZF,OF,PF,AF	2	2	1	1
FE	inc	mem8	SF,ZF,OF,PF,AF	2+	6	3	3
FF	call	reg32 (near indirect)	none	2	7+	5	2
FF	call	mem32 (near indirect)	none	2+	10+	5	2
FF	call	far indirect	none	6	22+	17	5
FF	dec	mem16 mem32	SF,ZF,OF,PF,AF	2+	6	3	3

（续）

操作码	助记符	操作数	受影响的标志位	字节数	时钟周期数		
					386	486	Pentium
FF	inc	mem16 mem32	SF,ZF,OF,PF,AF	2+	6	3	3
FF	jmp	reg32	none	2	10+	5	2
FF	jmp	mem32	none	2+	10+	5	2
FF	push	mem16 mem32	none	2+	5	4	2

* timing varies