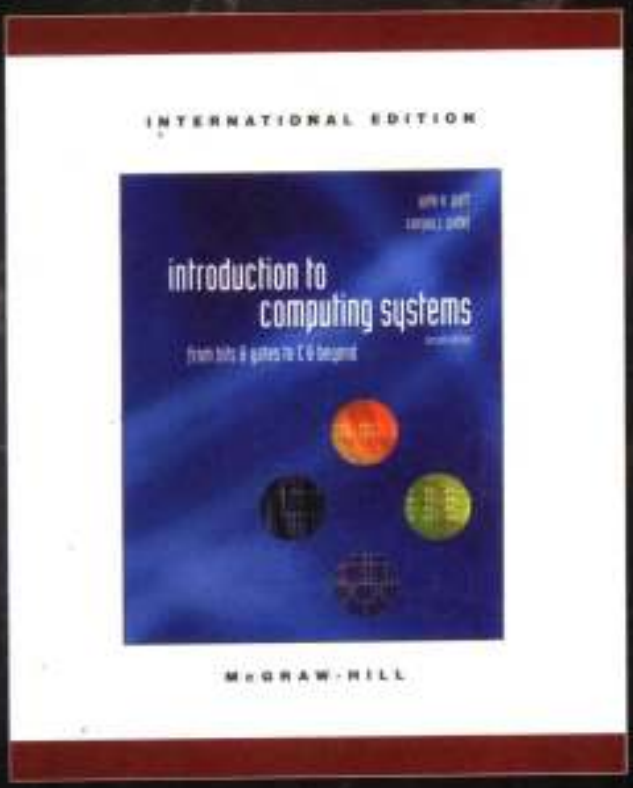
计 算 机 科 学 丛 书

原书第2版

计算机系统概论

(美) Yale N. Patt（得克萨斯大学奥斯汀分校）
Sanjay J. Patel（伊利诺伊大学） 著

梁阿磊 蒋兴昌 林凌
上海交通大学 译



Introduction to Computing Systems

From Bits and Gates to C and Beyond

Second Edition

机械工业出版社
China Machine Press

计算机系统概论（原书第2版）

本书为伊利诺伊大学（UIUC）等众多名校计算机科学的经典基础教材，作者是与Donald E. Knuth齐名的美国计算机界泰斗级作者Yale N. Patt。本书的目的是让学生在一进入大学校门的时候，就对计算机科学有一个深入理解，为以后的课程打下坚实的基础。

本书包括两方面的内容：计算机底层结构，高级语言编程及编程方法学。书中介绍了LC-3体系结构的设计，并配套开发了LC-3模拟器供学生使用，为加深学生对编程及其方法学的理解，本书选用了C语言作为载体，并采用了“驱动式”的自底向上方法进行讲解——先给学生一个整体结构，然后自底向上地建立起相关的知识。同样，在每个子单元中，也采用相同的驱动式教学方法，在每个学习阶段，都在之前已学的知识的基础上介绍新的概念。经验告诉我们，这种学习方法更多地强调理解而不是记忆。通过本书学习，学生的理解能力将获得很大的提高，因为他们循序渐进地了解了构建计算机的全部过程。

本书特色：

- 自底向上的组织：从最底层的MOS晶体管开关器件开始，然后是逻辑门、锁存器、逻辑结构（开关MUX、译码器、全加器、门锁存器等），最后使用这些单元来实现内存。之后，转至有限状态机控制、顺序电路的实现、冯·诺伊曼体系结构、一个简单的计算机（LC-3），以及LC-3的机器和汇编语言、C高级程序设计语言、递归等，最后是基本数据结构。
- 调试技术：从写第一个程序开始，就要求学生采用LC-3的调试工具和相关的调试技术。正因为如此，他们对编程艺术的体会更加深刻。
- LC-3模拟器：学习本书的一个重要过程是亲自操作LC-3模拟器，这是一个专门为学生掌握主要计算机概念而设计的工具。学生可以从本书网站免费下载LC-3模拟器。
- 编程方法学：本书给出了很多例程，其意义在于教会学生怎样分析问题，并通过系统的问题分解转换为计算机可编程的子问题。不论是使用LC-3汇编或C高级程序设计语言，编程思路上都存在相似性。这方面的理解和方法对快速掌握其他语言都有帮助。

本书网站上提供了丰富的辅助阅读材料和教学资料，请感兴趣的读者到http://www.mhhe.com/patt2下载。

作者简介

Yale N. Patt 拥有斯坦福大学电子工程博士学位，目前担任得克萨斯大学奥斯汀分校电子与计算机工程系教授。他是IEEE和ACM会士，曾因在高性能微处理器方面的成就而获得IEEE Emanuel R. Piore奖、IEEE/ACM Eckert-Mauchly奖和IEEE Wallace W. McDowell奖，在教学方面获得过ACM Karl V. Karlstrom杰出教育家奖，得克萨斯优秀教学奖等。



Sanjay J. Patel 拥有密歇根大学计算机科学与工程博士学位，是伊利诺伊大学厄巴纳-尚佩恩分校电子与计算机工程系副教授。他的研究领域包括处理器微体系结构、计算机体系结构、高性能和可靠计算机系统等。



Mc Graw Hill Education

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

上架指导：计算机，计算机科学导论

ISBN 978-7-111-21556-1

9 787111 215561

ISBN 978-7-111-21556-1
定价：49.00 元

计 算 机 科 学 丛 书

计算机系统概论

(美) Yale N. Patt (得克萨斯大学奥斯汀分校)
Sanjay J. Patel (伊利诺伊大学) 著

梁阿磊 蒋兴昌 林凌 译
上海交通大学

本书是计算机科学的经典基础教材。全书以自底向上方法帮助学生理解计算机系统的原理，前半部分阐述了计算机底层结构，后半部分讲解了高级语言编程及编程方法学，主要内容包括数据类型及其运算、数字逻辑、冯·诺伊曼模型、汇编语言、输入和输出、TRAP程序和子程序、C语言编程等内容。

本书可用作高等院校计算机及相关专业学生的入门教材，也可作为的计算机专业人士和高级程序员的参考用书。

Yale N. Patt and Sanjay J. Patel:Introduction to Computing Systems : From Bits and Gates to C and Beyond, Second Edition (ISBN: 0-07-246750-9).

Original language copyright © 2004 by The McGraw-Hill Companies, Inc.

All rights reserved.

Simplified Chinese translation edition published by China Machine Press.

本书中文简体字版由美国麦格劳-希尔教育出版公司授权机械工业出版社出版，未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有McGraw-Hill公司防伪标签，无标签者不得销售。

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：01-2006-3131

图书在版编目（CIP）数据

计算机系统概论（原书第2版）/（美）派特（Patt. Y. N.），（美）派特尔（Patel, S. J.）著；梁阿磊，蒋兴昌，林凌译. -北京：机械工业出版社，2007.7

（计算机科学丛书）

书名原文：Introduction to Computing Systems: From Bits and Gates to C and Beyond

ISBN 978-7-111-21556-1

Ⅰ. 计…　Ⅱ. ①派…　②派…　③梁…　④蒋…　⑤林…　Ⅲ. 计算机系统-教材　Ⅳ. TP30

中国版本图书馆CIP数据核字（2007）第082264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：王　璐

北京慧美印刷有限公司印刷 · 新华书店北京发行所发行

2007年7月第1版第1次印刷

184mm × 260mm · 27印张

定价：49.00元

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

Donald E. Knuth[⊖]: 有关Bottom-Up的论述

（摘自：《The Art of Computer Programming》（Fascicle 1: MMX））

为什么还需要机器语言？

许多读者可能都很困惑："Knuth在替换MIX的时候，为什么还是选择了一个机器语言，而不是某个高级编程语言呢？现在还有谁在使用汇编语言啊？"

或许持这种观点的读者是有他们的道理的，他们当然可以直接跳过书中涉及机器语言的章节。但我仍将坚持我在20世纪60年代早期本书第1卷序言中曾表述的观点：

本书的主要目标之一是：揭示一个高层结构在现实机器上的实现机制，而不是仅仅介绍使用它的方法。其中所阐述的内容将包括：多个相互关联的程序之间的链接、树结构、随机数生成、高精度运算、radix变换、数据封装、组合查找、递归等内容，即一种自底层逐步向上的论述方法（简称自底向上，bottom-up）。

在我的书中，程序例子都非常简短，原因是：我希望能更多地突出关键思想，引发读者思考。我要对那些希望深入了解计算机的读者说：了解底层硬件的工作原理机制是必需的，否则写出来的程序必然是"莫名其妙的"。

正如书中所述，软件的运算过程及其所表现的输出结果都将表明：机器语言在有些场合是非常必要的。

一些基本算法（如排序和搜索）通常都是用机器语言编写的，因为，只有这样才能保证最佳效率。在它们的编写中，要求我们对Cache、RAM大小等硬件结构有足够深入的研究，如内存访问速度、流水、多发（multi-issue）、后援缓冲、Cache块大小等问题。并且，还要分析它们对程序运行性能的影响，从而在各种可能的实现方法之间作出权衡和比较。

⊖ Donald E. Knuth是著名的《计算机程序设计艺术》（*The Art of Computer Programming*）一书的作者，以下论述摘自该书第一卷"MMIX"的内容片段。——译者注

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

本书的翻译工作终于完成了，有机会翻译Patt教授的著作让我们感到非常幸运。尽管本书是为大学一年级学生编写的计算机基础教材，但作者在阐述这些基本概念时所站的高度和论述深度都是无与伦比的。我们几位译者都长期从事计算机专业的教学和开发，但看到书中那些我们“已熟知”的概念时，作者所采用的论述角度和方式却让我们有种顿悟的感觉。

Patt教授在他的主页中说到：他热爱计算机教学和科研，但相比科研，教学是他的“first love”。鉴于他在计算机发展历程中的贡献及在计算机科学教育方面的深刻理解和倾心投入，他被IEEE评为泰斗级人物（与《计算机程序设计艺术》的作者Donald Knuth齐名，全球只有他们两人享此殊荣）。Patt教授在计算机体系结构发展史中占有一席之位：他在芯片集成制造方面提出了WOS概念（1965）；在微结构方面设计了HPS结构（综合了超标量、动态调度、乱序执行、预测跳转等多项高级技术）（1984）；提出了著名的“两级自适应跳转预测”结构（1991）。1996年，他被IEEE/ACM授予“在微处理器的指令级并行（ILP）和超标量设计领域杰出贡献”奖。我们敬仰于他在过去60年间为计算机发展所做的杰出贡献，以及他生生不息的创新能力（在写这本书的时候，他正在研究“2009年的计算机应该是什么样的”这一课题）。

有关本书在教学中的使用方法可参考原书序言。作者在第1版前言中提出了六种教学模式，在第2版前言中又将其归纳为三种教学模式，即“一年级两个小学期全授课”、“一年级两学期先高级语言再回顾深入底层”、“二年级再回顾底层”三种模式。我们的总结是，本书分为两部分，前10章（第一部分）讲述的是计算机的组成原理，后9章（第二部分）讲述C语言。

在前半部分内容中，作者以开发的LC-3机器（仿真机器）为背景，讲述了计算机的基本组成和实现（如运算器、指令编码/译码、I/O）等。学生可以通过LC-3机器和汇编器学习汇编，甚至尝试自己的机器设计方法。

本书的后半部分讲述了C语言内容。但其讲述方法与通常的C语言课本完全不同，它在讲述C语言变量、指针、语句、数据结构等内容的同时，结合前10章的知识，阐释了这些高级语言行为怎样转换为机器语言并运行在硬件系统中。其中，涉及很多长期困惑程序员的问题，如函数调用时栈空间（或函数帧空间）的使用规则、给变量赋值常量的几种方法之间的比较、软件中断指令的执行过程等。此外，即使将后半部分内容看作是单纯的C语言教材也无妨，它没有通篇机械地论述C语言的语法规则，而是通过简单计算器、Hanoi塔、排序等问题的求解，轻松地讲述了C语言编程的要点，且非常自然地传授了C编译器的工作原理、递归编程技巧和注意事项、程序调试的原理和技术、数据结构的重要性等“高级”内容。也正如作者在文中提到的，在语言教学中，编程方法学和实现原理比编程语法更为重要，前者传授的是通过编程解决问题的能力，后者则可以通过练习和自学就达到熟练。

总之，我们认为本书的特点是：深入、透彻、全面、易懂。书中虽然没有涉及更多具体、实际的技术知识（如x86），但读完本书后，再看Intel的技术手册，你一定会体会到“一通百

通”的感觉。

最后，借“译者序”之页，对所有参与和帮助本书翻译的人们表示致谢：本书第1、4、5、6、7、15章及附录A由梁阿磊翻译，第2、3、8、18、19章和附录D及索引由蒋兴昌翻译，第9、10、11、12、13、14、16、17章由林凌翻译，附录B和C由李鸿翻译了初稿。全书由梁阿磊审校。感谢杨建华先生对本书部分章节做出的细致而专业的校对和翻译建议，感谢姜玲燕对部分章节所做的细致校对和修改建议，感谢邹恒明先生为本书所作的序言。您现在看到本书的中文版，也凝结了翻译者的努力。我们尽了最大所能，试图将本书翻译成一本好书。但鉴于能力有限，如有不足和错误之处，敬请谅解和指正。最后，感谢亲人和朋友对本书的热情期待和支持。

译　者

2007年2月26日于上海交大

代　序

梁老师请我为本书写序之初我有些犹豫，原因有二：一是我没有给书写序的经验，二是恐怕自己水平有限，写的序不能充分表述或突出这样一本优秀教材的独特之处而影响该书的推广。不过，在经过慎重考虑后，我还是接受了这个挑战。凡事总有头一回吧，没有经验不是推脱的理由，而且本人在密歇根大学攻读博士学位的三年间耳闻目睹了该书第一作者Yale Patt的过人学识与非凡个性，随后在上海交通大学任教的三年里又与本书的第一译者梁阿磊博士成为好友至交，写起序来总会比不认识作者和译者的人有些优势。至于水平有限，只能请读者包涵了。

Yale Patt曾任美国密歇根大学计算机体系结构实验室主任多年，被*IEEE Spectrum*称为美国计算机界的卓越泰斗（luminary），在美国乃至世界计算机体系结构领域有着广泛的影响力。这本《计算系统概论》不光是密歇根大学计算机专业的经典基础教材，也是美国多所知名大学，如得克萨斯大学、莱斯大学、明尼苏达大学、乔治亚理工学院、伊利诺伊大学和西北大学等校的计算机专业基础教材。

本书第一译者梁阿磊博士是上海交通大学教师，他多年在计算机教学上花费的心血博得了学生的广泛认同，并成为上海交通大学计算机系和软件学院最受学生欢迎的教师之一。梁博士在翻译本书过程中精益求精，既保持了原著的精彩阐述和独特风格，又注入了中文语言所特有的内涵，使译著精辟且顺畅，并成为上海交通大学计算机及相关专业学生的指定教材。

也许你听过这个有启示的故事：在20世纪五六十年代时，美国GE公司的一个大型发电机出了问题。所有的操作人员及工程技术人员面对一大堆的仪表和旋钮一筹莫展。于是他们请来了一位这方面的专家，这位专家看了一下，便拿出一把螺丝刀，将一个旋钮反时针转动了35度，发电机就正常运转了。后来，专家开出的账单为1000美元，GE觉得花两分钟钮动一个旋钮就收取1000美元太多，便要求专家出具一份更详细的说明。两天后，专家寄来了新的账单：

将旋钮反时针方向转动35度：　　0.75美元

知道应转动哪个旋钮及转动幅度：999.25美元

一本好的教科书不应只教导学生转动*A*旋钮35度，而是应教会学生为什么要转动*A*旋钮及为什么只能转动不多不少的35度。这个例子恰恰能印证本书的优点所在。

与多数人的感觉不同，本书不仅简单地讨论计算机组成原理或程序设计语言，而是站在计算机整体系统的高度将软硬件连贯起来进行阐述。本书强调的是理解，而不是死记硬背；强调的是软件与硬件结合，而不是软硬件的分别教授与学习。本书本着将软件硬件教学齐头并进的思路，从硬件的基本构件一直讲到软件的高级程序设计与构造，使学生在学习过程中能够将软硬件融会贯通、相互印证，从而提高学习的广度、深度和效果，并为后续的计算机专业课如操作系统、计算机系统设计与结构、算法设计与分析、高可靠软件工程理论等打下基础。

本书最大的特点是其提倡的层次转换概念，即从问题开始到计算机运算出结果可以分为七个层次。通过七个层次的转换，即可完成从问题到结果的转变。这七个层次及其转换是：问题到算法的转换、算法到程序设计语言的转换、程序到指令集结构（ISA）的转换、指令集结构到微观结构的转换、微观结构到电路的转换和电路到电路组件的转换。该书对每个层次

转换进行了深入的阐述，讲解了为什么需要这些转换以及没有这些转换所带来的困难。美国多个大学的教学实践表明，这种层次转换的概念极大地提高了学生对计算机软硬件的理解。

本书在结构上采取的策略是自底向上。在对计算机的哲学原理进行了简要综述后，从晶体管开始，对逻辑门、触发器、逻辑结构和内存实现分别进行了详细的讨论。在此基础上，再对冯·诺伊曼执行模式进行剖析、讲解如何构建一个简单的LC计算机。然后上升到软件层，对计算机汇编语言和高级程序设计语言C进行讲解，重点将程序语言的各种数据类型和构造与底层的硬件构造联系起来，从而使学生在深层次上理解程序语言里各种构造的来龙去脉，及计算机软硬件之间的有机的、千丝万缕的联系。

本书的使用方式有多种，其中适合中国国情的方式有以下四种：

1. *密歇根模式*：作为计算机软硬件原理的入门教材，无需先修课程。该模式要求一个学期内覆盖本书全部内容，时间安排为本科第一学年的第一学期。该模式工作量大，挑战性高，非常适用于肯钻研、进取心强的学生。
2. *正常模式*：作为软硬件专业入门教材，无需先修课程。该模式要求在一个学期覆盖大部分内容，但可以跳过10.3、10.4节、第16章、第18章和第19章。时间安排为本科第一年的第二学期。该模式较密歇根模式来说负担稍轻，但仍具有相当的挑战性。
3. *第二模式*：在学生已经进行了专业入门教育如程序设计、数据结构和数字电路等课程的学习后使用本书作为第二课程的教材。这种模式在一个学期覆盖本书全部内容，而讲课时间则安排在本科二年级上学期。该模式适合于计算机专业的大多数学生。
4. *分治模式*：将这本书的内容分为两个学期讲授。第一学期讲授第1~10章，第二学期讲授第11~19章。时间安排为本科一年级下学期和二年级上学期，这是使用本书的最优方式。

本书结构紧凑、风格鲜明、内容精辟、条理清晰、文笔流畅、语言优美，读起来引人入胜，爱不释手。本书还引入了大量的典故和比喻（当然，有些典故和比喻需要西方及美国文化背景才能更好地领悟），让人感觉到不是在读一本计算机教科书，而是在读一本幽默有趣的长篇小说，或者借用Donald Knuth的话，它是“一部蓝色的诗歌”。

本人强烈推荐本书给高校计算机专业的师生和所有对计算机软硬件有着浓厚兴趣的人士。读完本书后，你对计算机系统的理解将会达到一个崭新的境界。相信我，你不会失望的。

邹恒明

2007年3月2日于羊庄

第2版前言

本书自第1版发行至今已有三个年头，期间我们收到了大量学生和教师的反馈信息，几乎所有的反馈信息都是“正面的”，这让我们备感欣慰！能有机会出版本书的第2版，实在是让人高兴的一件事情。之所以高兴，是因为大多数人都赞同本书的编写方法，尤其是我们收到的反馈信息都来自学习第一线的读者——它们包含了学习者（学生）的感受和授业者（教师）对本书使用效果的评价，从他们的E-mail中能看出对本书的评价都很高。

但正如本书第1版前言中所提到的，本书仍然在追求不断地完善。另外，在接受赞誉的同时，我们也收到很多建议，这些建议指出怎样使本书写得更好，我们在此向所有关心本书的读者深表感谢。同时，由于第1版出版后我们两人都分别在课堂上讲了两遍以上，也发现了很多需要修改和改进的地方。因而我们决定在第2版中做出较大的修改，但同时也期望新版能沿续第1版编写的初衷，并期待尽快听到读者的反馈。

修订内容

LC-3

本版最大改动之一就是，采用LC-3（Little Computer 3）结构替换了第1版的LC-2机器模型。实际上，新结构保持了LC-2的基本概念，即一个易于描述又有望在短期内掌握的丰富的ISA。指令仍然采用16位宽度，其中操作码为4位。正如我们的一个学生指出的，事实上，子程序返回指令（RET）就是LC-2的JMPR指令的特例，因而在LC3中，我们取消了RET指令作为单独操作码的定义。在LC-3中，我们只定义了15种操作码，预留了一个用于未来应用（或许在第3版中会定义这个操作码）。

其次，我们收到很多反对PC-concatenate寻址模式的建议，特别是针对分支跳转指令。该寻址模式起源于20世纪60年代中期的PDP-8机器，问题表现在当前页中的一条指令访问下一页（或之前页）之时。这个问题很让人头痛，尤其是在接近页边界时出现的前向跳转指令。因而，很多人建议我们采用现代机器中常用的“PC+offset”方式。因此，我们将第1版中所有使用“PC偏移（offset）”之处，都改成了“PC+SEXT（offset）”。

另外，新版还包括以下修改：

- 栈空间增长改为向0地址增长（符合现在的实际惯例）。
- LDR/STR指令中的偏移改成了有符号值（因而可以基于base地址做向前、向后偏移）。
- 操作码1101现在未做定义。
- JSR/JMP的操作码做了轻微改动。
- 最后，条件码扩展为16位处理器状态寄存器（Processor Status Register, PSR，其中包含了特权模式和优先级字段）。

与第1版相同，在附录A中有LC-3结构的完整描述。

扩充内容

在第1版的基础上，新版几乎对所有章节都做了或多或少的修改，有些章节的改动大一些。

其中：

- 第1章中增加了两方面的内容（1）有关抽象方法的重要性及其本质性的描述；（2）硬件和软件之间相关性的讨论。
- 第3章增加了一节，即有关有限状态控制及顺序开关控制的实现方法。因为这些实现方法和技巧已成为计算机科学（CS）或计算机工程（CE）专业学生必须掌握的知识，这些知识的掌握将有助于对第4章冯·诺伊曼模型的深入理解。
- 第4章中增加了对LC-3微结构的概述，相关内容的详细描述参见附录C。
- 第5章，有读者反映内容太扼要，在此我们增加了一些素材，希望新增的这些图和注释会使相关概念更加清晰。
- 第8章和第10章中增加了有关中断驱动I/O的章节内容。
- 第11~14章介绍了C语言（与第1版一样），不同之处是，这些章节更加注重帮助编程初学者掌握语言的基本内容，而将那些特性内容移至该章结尾或附录D之中，使之脱离教材的主线。所有这些章节都增加了更多的例子。

另外，第2版改用“问题解决驱动”方式来讲述编程技术，即通过例子展示为什么新引入的C结构可以在C编程中解决当前的问题（这将有助于读者真正理解编程语言的设计精妙之处）。

- 第14章中还将介绍新的LC-3调用规范，新的规范更接近实际系统中所采用的规范。
- 第15章介绍了有关测试和调试的深入知识。
- 章节顺序的调整：根据来自教学中的经验，我们决定调换“递归”和“指针与数组”两章的顺序。从而使学生在学习“递归”内容（相对较难，目前推后至第17章）之前有更多的机会熟悉和掌握C语言的编程方法。

仿真器

针对新的LC-3结构，Brian Hartman对原先的Windows版仿真器（Simulator）做了修正。Ashley Wise则为UNIX平台编写了LC-3仿真器。在两个版本的仿真器中，都实现了“中断驱动I/O”功能。我们相信没有任何方法比“实际动手”更有助于知识的真正掌握。具备了中断驱动I/O功能的仿真器，使得学生现在可以实验这样一个场景，即通过键盘输入中断一个正在执行的程序，并调用对应的中断服务程序。

本书的使用方法

本书是计算科学相关专业新生的入门教材。我们认为“自底向上”方法是帮助学生理解技术计算原理的最好方法（参考后面第1版前言中的详细论述）。事实和经验表明，学生在掌握了计算机底层工作的原理机制之后，能更加从容地解决以后可能面临的新问题，包括高级编程语言方面的问题。而且，这些学生学习编程语言的方法是“理解式”而不是“记忆式”的，因为一切动作都是明明白白的。从我们的角度来看，本书的最佳使用方法是，开设一个学期的新生主干课程，或是两个学期的课程（节奏慢些）。如果教学的重点是单学期制的高级编程语言，则串行机和中断驱动I/O的内容可以跳过；而如果教学的重点是本书的前半部分，则第15、17、18和19章的内容可以跳过。

通常，这本书的参考使用方法包括：

- 一年级第一学期作为新生主干课程。

第一种方法是在一年级第一学期内分两个小学期完成，这可能是本书最好的使用方法。在第一个小学期，讲授本书的第1~10章，在第二个小学期，讲授第11~19章。这样的进度有些

快，但在两个小学期（即一学期）内是可以讲解完整本书的（注：一个学期（semester）分为两个小学期（quarter））。

• 一年级第二学期作为计算机的第二门课程。

第二种方法还可作为计算的第二门课程在一年级第二学期完成，学生在之前的第一学期已修完了第一门高级编程语言方面的课程，之后在这门课程中，则回头学习初级的数字逻辑、基本的计算机组成和汇编语言编程等底层知识。这种方式下，这一学期的大部分时间都用于学习第1~10章，只留下最后几周学习第11~19章的一些课题，以显示学生从第一门高级程序设计语言课程中学到的机理是如何在底层实现的。其中，通常会讲到函数、活动记录、递归、指针变量和其他基本数据结构等课题。

• 二年级计算机组成课程（Sophomore-Level）。

本书还可用于二年级要上的深入探究计算机实现的课程。在这期间，重点学习第1~10章，有时还需彻底学习附录C（LC-3之微结构和微编程实现）。需要指出的是，本书未讲述计算机体系结构中的一些重要概念，如高速缓存、流水线、虚拟存储器等。我们认同这些概念对于有志成为计算机科学家或计算机工程师的学生至关重要，但同时又感到这些内容更适合放到计算机体系结构与设计的更高级课程中来讲。本书的目的并不在此。

致谢

本书需要向太多的人表示致谢，感谢他们为本书所做的重要贡献。其中，尤其要感谢Brian Hartman和Matt Starolis。

Brian Hartman先生以他的积极热情和技术专长继续为本书新版做出重要贡献。1996年冬，他在密歇根大学作为一年级本科生选修了这门课。他随后在本科期间多次担任这门课的助教，并在硕士学习期间编写开发了Windows版LC-2仿真器（同时这也是他的硕士毕业课题）。最近，他又为LC-3开发了Windows版仿真器。现在，他已离开学校三年多了，但仍在坚持维护该系统。

Matt Starolis在两年前作为UT（得克萨斯大学）一年级学生选修了该课程，并于去年秋季担任该课程助教，在本书第2版的编写中担任了重要角色，为本书提出了许多意见并设计了书中的多幅插图。他还修订了LC-3仿真器的使用手册，以使之与新版仿真器一致。往往在任何事情需要有人去做的时候，他总是第一个站出来。他的热情为本书注入了活力。

目前本书有大约100多位采用者，我们定期地收到来自世界各地的教授们热情洋溢的E-mail。他们是Vijay Pai（Rice）、Richard Johnson（Western New Mexico）、Tore Larsen（Tromso）、Greg Byrd（NC State）、Walid Najjar（UC Riverside）、Sean Joyce（Heidelberg College）、James Boettler（South Carolina State）、Steven Zeltmann（Arkansas）、Mike McGregor（Alberta）、David Lilja（Minnesota）、Eric Thompson（Colorado Denver）、Brad Hutchings（Brigham Young）。

至于我们两人，自第1版出版以来，共使用本书教授了4届学生，并培养了一大批热情、积极的助教和学生。其中，助教包括Kathy Buckheit、Mustafa Erwa、Joseph Grzywacz、Chandresh Jain、Kevin Major、Onur Mutlu、Moinuddin Qureshi、Kapil Sachdeva、Russell Schreiber、Paroma Sen、Santhosh Srinath、Kameswar Subramaniam、David Thompson、Francis Tseng、Brian Ward、Kevin Woley，而Linda Bigelow、Matt Starolis、Lester Guillory等人一开始都是新生，两年后已成为最积极的助教。

Ashley Wise开发了Linux版本的LC-3仿真器，Ajay移植了LCC编译器来产生LC-3代码。

Gregory Muthler和Francesco Spadini对本书后半部分的原稿提出了积极的反馈意见。Brian Fahs为习题提供了解答。

Kathy Buckheit编写了LC-2仿真器的入门材料（因为她觉得这很有必要）。

得克萨斯大学（University of Texas）许多同仁都采用了本书，并与我们分享他们的见解，如ECE专业的Tony Ambler、Craig Chase、Mario Gonzalez、Earl Swartzlander和CS专业的Doug Burger、Chris Edmundson、Steve Keckler。在此表示感谢。

我们还要继续感谢认真负责的编辑们，他们是Betsy Jones和Michelle Flomenhoft。

自第1版以来，我们的书还受益于很多大学教师的审核，感谢他们：Robert Crisp（Arkansas）、Allen Tannenbaum（Georgia Tech）、Nickolas Jovanovic（Arkansas-Little Rock）、Dean Brock（North Carolina-Asheville）、Amar Raheja（Cal State-Pomona）、Dayton Clark（Brooklyn College）、William Yurcik（Illinois State）、Jose Delgado-Frias（Washington State）、Peter Drexel（Plymouth State）、Mahmoud Manzoul（Jackson State）、Dan Connors（Colorado）、Massoud Ghyam（Southern Cal）、John Gray（UMass-Dartmouth）、John Hamilton（Auburn）、Alan Rosenthal（Toronto）、Ron Taylor（Wright State）。

最后，还要感谢那些以独特方式做出贡献的人们（在此不再赘述他们的具体贡献，只是列出他们的名字）：Amanda、Bryan、Carissa Hwu、Mateo Valero、Rich Belgard、Janak Patel、Matthew Frank、Milena Milenkovic、Lila Rhoades、Bruce Shriver、Steve Lumetta、Brian Evans。另外，Sanjay（作者之一）还要感谢Ann Yeung的爱和支持。

结束语

再次重复第1版前言中的话：本书仍然在追求不断地完善，期待你们提出宝贵意见。我们的联系方式是：patt@ece.utexas.edu和sjp@crhc.uiuc.edu。

Yale N. Patt
Sanjay J. Patel
2003年3月

第1版前言

本书内容取自密歇根大学开设的EECS100课程——这是计算机科学（CS）、计算机工程（CE）和电子工程（EE）三个专业的第一门计算机类主修课程。该课程由Kevin Compton教授和本书第一作者于1995年秋季开设。

之所以开设EECS100，是因为多年来计算机科学与工程系的教师认为，他们的本科生对计算机基本概念缺乏深入的理解。例如，学生们无法清楚地解释指针变量，而递归概念对他们来说就像是在“变魔术”，难以理解。

所以，我们在1993年提出对传统教学思路的改革。传统的教学思路是从高级计算机语言开始入手，但是在这种教学方式下，学生仅仅是“记住”了其中的技术细节，并不能真正“理解”其原理、机制。本书的教学思路是“自底向上”（bottom-up）：从MOS晶体管开始，依次介绍逻辑门、锁存器、各种逻辑结构（如MUX、解码器、加法器、门控锁存器），然后通过存储器（或内存）的实现案例，将以上概念有机结合。随后，介绍冯·诺伊曼模型，并以简单的LC-2计算机为背景，介绍LC-2机器语言编程和汇编语言编程；再之后，我们继续上升到高级语言（如C语言），以及递归、指针、数组等概念；最后，引入一些基本的数据结构。

有一种“信息隐藏”（Information Hiding）学习方法，但我们不太赞同这种方法。事实上，“信息隐藏”确实是个很有效的学习方法[⊖]。但我们认为，只有在你真正搞明白它在做什么之后，信息隐藏方法才具有意义。换句话说，我们计划将每个概念的底层动作都暴露、表现出来，消除在上层感受到的神秘感。

需要指出的是，虽然本书采用的是“自底向上”方法，但并不意味着我们反对“自顶向下”（top-down）方法。相反，我们认为“自顶向下”在设计方面是非常正确的方法。在设计中采用的方法，与在学习中采用什么样的方法，是两件不同的事情。就是说，对前者我们主张采用“自顶向下”方法（前提是设计者已深入理解底层构造），而对后者（即学习和理解过程）我们则主张“自底向上”方法。

内容概要

本书内容包括两个部分：一是计算机底层结构（LC-2计算机），二是高级语言编程（C语言）。

LC-2

本书的前半部分偏重计算机底层机制。从底层基础知识开始，逐步上升到操作系统和高级语言程序的接口层，以便能够理解真实计算机的工作原理。

- 第2章：介绍基于位（bit）的算术和逻辑运算操作，以及相关的结构设计（即LC-2计算机的组成）。
- 第3章：存储器设计。从MOS晶体管开始，逐步介绍存储器的实现过程。该存储器很简

⊖ 所谓“信息隐藏”方法，即在学习新概念时，不要过分追究细节。——译者注

单，每字宽度为3-bit，共4个字大小（而不是64MB）。其中，每个部件的设计图都不超过一页大小（以方便阅读）。至此，构建存储器所要求的知识就全部介绍完了。

- 第4章和第5章：介绍冯·诺伊曼模型及LC-2计算机的组成结构。LC-2是一个16-bit结构的计算机，也是冯·诺伊曼模型的一个具体实现。它具备以下功能和特性：
  - 键盘和显示器等物理I/O设备。
  - 调用操作系统服务的TRAP机制。
  - 基于N、Z、P等条件码的条件跳转。
  - 子程序调用和返回机制。
  - 基本的操作指令集合（ADD、AND和NOT）。
  - 各种加载（load）和存储（store）寻址模式，如直接、间接、基址+偏移和立即数等有效地址计算方式。
- 第6章：介绍编程方法学和调试方面的知识。
- 第7章：介绍汇编语言编程，以及我们开发的LC-2仿真器和汇编器。事实上，我们开发了两个版本的仿真器，一个是在Windows平台上运行，另一个是在UNIX平台上运行。Windows版本的仿真器程序可以从网上下载或从本书光盘获取；偏好UNIX的同学，可从网上免费下载UNIX版仿真器，然后安装。

学生使用仿真器的目的是测试和调试用LC-2机器语言和LC-2汇编语言编写的程序。它支持在线调试（即内存检查、单步、断点设置等操作）。该仿真器仅适合于调试简单的LC-2机器语言或汇编语言程序，其主要目的是帮助学生理解和掌握第1~10章中的基本概念。

有关汇编语言，我们仅限于介绍，而不是熟练编程。汇编的编程技术是更高级课程（而不是一年级课程）的任务。之所以在第7章介绍汇编语言知识，是因为它在“自底向上”层次中独占一层。通过它，学生可以观察到从汇编语言到0/1序列的转换过程。并且，通过手工汇编（即手工转换），虽然花费了大量时间，但必将为学生打下坚实的概念基础，令他们受益终生。

所谓汇编语言，可以将它看做是机器指令的一种“友好”表示方式，它的好处在本书后半部分将体现出来。因为从第11章开始，我们将借用汇编语言或机器语言来解释C语言语句的底层含义。当然，汇编语句“ADD R1,R2,R3”显然比机器代码“0001001010000011”具有更好的可读性。

- 第8章：讨论物理设备的输入（键盘）和输出（显示器）。
- 第9章：讨论操作系统的陷入机制（TRAP），以及子程序调用和返回机制。学生将学习用LC-2代码编写的操作系统服务程序，以便生成由TRAP指令调用的物理I/O。
- 第10章：是本书上半部分的总结。通过“计算器程序”例子，讲述栈和数据转换的原理和机制。该程序由1个主程序和11个子程序组成。

C语言

本书下半部分内容是C语言编程，但不是对于C语言的一般性介绍，而是对其内部机制的深入洞察（学生之前已对C语言底层有所了解）。

作为一种高级语言，C语言最适合于我们的“自底向上”方法，因为C语言是各种高级语言中“最低级的”。通过它，能够清晰地表现软件和硬件的接口关系。本书的C语言学习内容，着重于控制结构语句、函数、数组等基本概念；在基本掌握编程概念之后，高级概念（如对象、抽象等）的学习就是轻而易举的事情了。

在学习过程中，每介绍一个C语言结构，我们都将给出其对应的LC-2汇编代码（即编译

器对高级语言的处理结果）。在此，各种C结构内容包括：基本结构（变量、运算符、控制语句和函数）、指针、递归、数组、结构、I/O操作、复杂数据结构和动态分配等。

- 第11章：高级编程语言概述。本章将通过一个简单的C程序，作为学习C语言的“敲门砖”。学生之前学习的汇编技术，将有助于对高级语言要素背后动机的理解。
- 第12、13章：这两章是对C语言的系统介绍。其中，第12章涉及数值、变量、常数、运算符等概念。第13章引入C语言的控制结构。在此，我们给出了很多例子，以讲解这些概念的灵活用法。同时，还将辅以LC-2代码，讲解C语言概念与底层机器之间的关联。
- 第14章：介绍高级语言源代码的调试技术㊀。
- 第15章：介绍C语言的“函数”概念㊁。但本书中的重点不仅是语法层面的介绍，更重要的是其运行时原理的描述。如栈空间运行时的变化过程。
- 第16章：讲述“递归”技术㊃。它涉及之前学过的函数、活动记录、运行时栈空间等概念。
- 第17章：指针和数组㊂。理解这两个概念之间微妙区别的要点在于对内存结构的深入理解。
- 第18章：介绍C语言I/O函数的技术细节，包括：流、可变长参数，以及C语言通过各种“格式”（format specification）控制I/O的效果（参考第8章）。
- 第19章：C语言的总结。并引出结构、动态内存分配和链表等概念。

本书不仅介绍各种概念和定义，同时也注重培养良好的编程风格和方法学，这一点体现在本书的各个例子代码中。通过这些例子，可以加深初级程序员对基本概念的深入理解。

我们在教学过程中发现，学生对“指针变量”的理解毫不费力，这让我们非常意外。这或许就是本书教学方法的效果，因为之前他们已了解内存的概念及其逻辑设计和实现机制，所以能迅速理解地址和数据之间的差异。

“递归”是个难以掌握的概念。但在学习“递归”概念时，理解“递归”所需的基础知识都已具备，如：子程序调用/返回时的栈变化（第10章）、被调用者（called或callee）和调用者（caller）之间的链接方法等，然后引入运行时活动记录、参数传递、动态声明等概念就比较容易了。换句话说，如果你能够理解一个函数是怎样调用另一个函数的，那么一个函数调用它自己的过程（即递归）也就不言而喻了。

本书的教学方法

在过去的两年里，我们研究了多种基于本书的教学方法。下面推荐六种教学方式。

(1) *密歇根大学模式*：在该校，它是本科入学的第一门课程。在不需要任何预备知识的情况下，集中讲解全书内容。我们发现这种模式特别适合那些有天分的、爱动脑筋的同学。

(2) *一般模式*：第一门课程，不需要预备知识。仍然是集中讲解，但略过个别章节，如10.3节、10.4节以及第16章（递归）、第18章（C语言的I/O细节）和第19章（数据结构）。

(3) *第二门课模式*：很多学校认为这种模式效果较好，即在第一门课中先学习“面向对象编程语言”，然后在第二门课中学习本书。在这种模式中，学期的前2/3时间是本

㊀ 在新版中，这部分内容放到了第15章。——译者注

㊁ 在新版中，这部分内容放到了第14章。——译者注

㊂ 在新版中，这部分内容放到了第17章。——译者注

㊃ 在新版中，这部分内容放到了第16章。——译者注

书的前10章内容，后1/3时间是本书的第二部分内容。由于学生在之前已对编程有所了解，所以后半部分内容的学习将结合编程大作业（project）来完成。这种模式的特点是，先学“面向对象语言”，然后反过来学习C语言，以便为后面的高级软件课程（如操作系统）学习做好准备。

(4) *两个小学期模式*：这是一种非常好的模式。无需预备知识，在两个小学期内完成全书。第一个小学期内完成第1~10章，第二个小学期内完成第11~19章。

(5) *两个大学期模式*：这可能是最优模式了。即花费一个学年的两个学期，无需预备知识。第一学期完成前10章和附录C（LC-2的微结构）；第二学期完成第11~19章。课程中配合大量与编程相关的课程设计，促进学生巩固课堂知识。

(6) *二年级硬件课程*：个别大学采用这种模式，其目的在于让二年级学生加深对硬件知识理解。他们计划在一个学期内，让学生掌握“数”（number）的系统定义、数字逻辑、计算机组织、机器语言和汇编，最后以栈、活动记录、递归和链表等概念结束。其思路是，将本课程的前半部分内容与一年级所学的编程课程有机结合，以回顾编程课程中遗留下的概念问题。虽然，我们的建议是将本书的授课时间安排在“面向对象语言”课程之前，但这种方法收到的效果也很好。因为学生在之前的一年级编程学习中曾遇到并遗留下的很多概念上的问题在本课程中都将找到答案，从而起到“豁然开朗”的效果。

经验与体会

理解而不是记忆

由于本课程的学习方法是“自底向上”，所以不存在编程课程所面临的“规则”记忆。这是因为按照本书的知识结构，在接触新概念之前，与之相关的底层实现机制必然都已学过。另外，这种“自底向上”的方法对以后设计类课程的学习也非常有益。因为如何在多种设计方案中做出正确的权衡和决策，完全取决于你对问题的理解和洞察深度。

自己动手

我们经常听到工业界抱怨：“计算机专业毕业的学生不懂得编程”。然而，造成这种现象的部分原因，竟然是“热心”的助教。因为，他们将“实验”准备得太充分了，以至于学生不费力气就能完成题目，从而失去了对编程过程和艺术的“体会”。我们所要求的是：学生必须在没有助教帮助的情况下，尽可能独立地完成编程任务。当然，我们之所以敢这么要求，原因在于：(1) 本书所采用的“自底向上”方法给予学生的是“理解”而不是“记忆”；(2) 本书配套的仿真器工具，学生从第一天学习开始，就要使用它，并用它来调试程序。这种学习顺序的安排，以及基于仿真器的编程调试经验，培养了学生通过分析和实验方法来解决问题的能力（而不是一味地求助于助教，即由助教完成程序的编写工作）。

为未来做准备：深入底层

计算机专业人士都有这样的体会，即系统运行的性能不仅仅取决于他们编写程序的水平。缺乏对系统底层知识的了解，使得他们面对一些性能问题时一筹莫展。这种情况很多，其中不乏资深程序员和工程师。

作为高级程序员，要编写高效率代码，仅仅掌握高级语言本身是不够的。除此之外，他

们还需要了解与设备相关的知识（甚至是引脚定义等）。例如，在一些应用系统中，计算机的作用是从某种测量设备（如天气测量仪或反馈控制系统等）中采样数据，那么工程师要掌握的知识就不仅限于FORTRAN语言了。对于电子工程师来说是如此，在机械、化工、航空等领域亦如此。而在高级语言编程课程中，编译器这个“保护层”也将计算机底层的“丑陋细节都隐藏了。换句话说，如果计算机课程的内容仅限于编程语言，那么培养出来的学生是无法胜任未来工作的。

涟漪效应

本书内容对后续课程将产生涟漪效应（rippling effect）。例如，假设学生对C语言语法和底层结构之间的互动机制已有所了解，那么在以后的语言编程课程中就可以将重点放在问题求解算法及更复杂的数据结构方面。再如，在硬件方面也存在类似的情况：在以后的数字逻辑设计和计算机组织课程中，他们很容易联想到硬件和上层语言之间的交互场景，从而理解底层设计的重点和动机。在计算机组织课程中，刚接触术语“程序计数器”（Program Counter）时，学生通常会问，“为什么要有程序计数器，有什么用？”密歇根大学的教学反馈表明，在EECS 100课程开设前后，学生在后续课程中对该类问题的理解有明显差异。

致谢

本书要感谢太多人的贡献。因而我们一直害怕遗漏了其中的某个人。在此，谨向大家表示致谢。

第一个要感谢的是Kevin Compton教授。早在1993年，在他主持的密歇根大学教学讨论会上，我一提出该教学理念，就获得他的强烈支持。随后，他和本书作者共同创建了一门新课——EECS 100。之后，我们两人还一起执教了三个学期（1995年秋、1996年冬、1996年秋）。Kevin对编程方法学（而不是特定语言）的深入洞察，为新生建立了良好的素质基础。总之，如果不是Kevin，EECS 100这门课绝不会在密歇根大学获得如此之高的声誉。

EECS 100课程创建之初，得到密歇根大学众多同学和教员的无私帮助。在此，特别感谢以下各位教授：David Kieras、Brian Hartman、David Armstrong、Matt Postiff、Dan Friendly、Rob Chappell、David Cybulski、Sangwook Kim、Don Winsor、Ann Ford。

还有就是我们的助教们，他们非常会动脑筋。比如怎样解释才能使学生更容易理解这些概念？这里要特别感谢：Fadi Aloul、David Armstrong、David Baker、Rob Chappell、David Cybulski、Amolika Gurujee、Brian Hartman、Sangwook Kim、Steve Maciejewski、Paul Racunas、David Telehowski、Francis Tseng、Aaron Wagner、Paul Watkins。

我们还要感谢很多出版界对我们手稿的关注。但最终我们还是选择了McGraw-Hill出版社，这主要归因于编辑Betsy Jones女士。在初步交流之后，她就对正在写作中的书稿表示了强烈的信心，她的鉴赏水平和评价能力让人钦佩！同时，我们也很感谢Michelle Flomenhoft女士与本项目的合作，与她一起工作非常愉快。

各大学的教员们赐予本书的评述意见也让我们受益匪浅，感谢：Carl D. Crane III (Florida)、Nat Davis (Virginia Tech)、Renee Elio (University of Alberta)、Kelly Flangan (BYU)、George Friedman (UIUC)、Franco Fummi (Universita di Verona)、Dale Grit (Colorado State)、Thor Guisrud (Stavanger College)、Brad Hutchings (BYU)、Dave Kaeli (Northeastern)、Rasool Kenarangui (UT at Arlington)、Joel Kraft (Case Western Reserve)、

Wei-Ming Lin（UT at San Antonio）、Roderick Loss（Montgomery College）、Ron Meleshko（Grant MacEwan Community College）、Andreas Moshovos（Northwestern）、Tom Murphy（The Citadel）、Murali Narayanan（Kansas State）、Carla Purdy（Cincinnati）、T.N.Rajashekhara（Camden County College）、Nello Scarabottolo（Universita degli Studi di Milano）、Robert Schaefer（Daniel Webster College）、Tage Stabell-Kuloe（University of Tromsoe）、Jean-Pierre Steger（Burgdorf School of Engineering）、Bill Sverdlik（Eastern Michigan）、John Trono（St. Michael's College）、Murali Varansi（University of South Florida）、Montanez Wade（Tennessee State）、Carl Wick（US Naval Academy）。

此外，还有很多人通过其他方式给予了帮助。由于篇幅限制，在此仅列出他们的名字并表示感谢：Susan Kornfield、Ed DeFranco、Evan Gsell、Rich Belgard、Tom Conte、Dave Nagle、Bruce Shriver、Bill Sayle、Steve Lumetta、Dharma Agarwal、David Lilja、Michelle Chapman。

最后，恕我们自信直言：这本书的目标是培养坚实的计算机基础。我们相信，一旦学完本书，学生将不再为概念和细节所困，从而可以集中全力发挥其才智和潜能。感谢我的老师William K. Linvill，是他向我传授了这种理念。35年前，我坐在他的课堂上，他的言传身教都在告诉我，怎样做一个好教授，他是我终生的榜样。

结束语

我们希望你能喜欢本书的写作方法。本书仍然在追求不断地完善，期待你们提出宝贵意见。我们的联系方式是：patt@ece.utexas.edu和sjp@crhc.uiuc.edu。

Yale N. Patt
Sanjay J. Patel
2000年3月

目　录

出版者的话
专家指导委员会
译者序
代序
第2版前言
第1版前言

第1章　欢迎阅读本书 ……1
1.1　我们的目标 ……1
1.2　怎么才能做到 ……1
1.3　两个反复出现的理念 ……2
1.3.1　抽象之理念 ……2
1.3.2　硬件与软件 ……3
1.4　计算机系统简述 ……4
1.5　两个非常重要的思想 ……6
1.6　计算机：通用计算设备 ……6
1.7　从问题描述到电子运转 ……8
1.7.1　问题的提出 ……8
1.7.2　算法 ……8
1.7.3　程序 ……9
1.7.4　指令集结构 ……9
1.7.5　微结构 ……10
1.7.6　逻辑电路 ……10
1.7.7　器件 ……10
1.7.8　小结 ……10
1.8　习题 ……11
第2章　bit、数据类型及其运算 ……14
2.1　bit和数据类型 ……14
2.1.1　bit——信息的基本单位 ……14
2.1.2　数据类型 ……14
2.2　整数数据类型 ……15
2.2.1　无符号整数 ……15
2.2.2　有符号整数 ……15
2.3　补码 ……16
2.4　二进制数与十进制数之间的转换 ……18
2.4.1　二进制数转换为十进制数 ……18
2.4.2　十进制数转换为二进制数 ……19
2.5　bit运算之一：算术运算 ……20
2.5.1　加法和减法 ……20
2.5.2　符号扩展 ……20
2.5.3　溢出 ……21
2.6　bit运算之二：逻辑运算 ……22
2.6.1　“与”运算 ……22
2.6.2　“或”运算 ……23
2.6.3　“非”运算 ……24
2.6.4　“异或”运算 ……24
2.7　其他类型 ……24
2.7.1　位矢量 ……25
2.7.2　浮点数 ……25
2.7.3　ASCII码 ……27
2.7.4　十六进制计数法 ……28
2.8　习题 ……29
第3章　数字逻辑 ……34
3.1　MOS晶体管 ……34
3.2　逻辑门 ……35
3.2.1　非门 ……35
3.2.2　或门、或非门 ……36
3.2.3　与门、与非门 ……37
3.2.4　摩根定律 ……38
3.2.5　多输入门 ……39
3.3　组合逻辑 ……39
3.3.1　译码器 ……40
3.3.2　多路复用器 ……40
3.3.3　全加器 ……41
3.3.4　可编程逻辑阵列 ……43
3.3.5　逻辑完备性 ……43
3.4　存储单元 ……44
3.4.1　R-S锁存器 ……44
3.4.2　门控D锁存器 ……45
3.4.3　寄存器 ……45
3.5　内存的概念 ……46
3.5.1　寻址空间 ……46

3.5.2 寻址能力 ……46
3.5.3 例子：$2^2 \times 3$内存 ……47
3.6 时序电路 ……48
3.6.1 组合密码锁 ……49
3.6.2 状态的概念 ……50
3.6.3 有限状态机 ……51
3.6.4 有限状态机的实现 ……53
3.7 LC-3计算机的数据通路 ……55
3.8 习题 ……56
第4章 冯·诺伊曼模型 ……65
4.1 基本部件 ……65
4.1.1 内存 ……66
4.1.2 处理单元 ……66
4.1.3 输入和输出单元 ……67
4.1.4 控制单元 ……67
4.2 LC-3：一台冯·诺伊曼机器 ……67
4.3 指令处理 ……69
4.3.1 指令 ……69
4.3.2 指令周期 ……70
4.4 改变执行顺序 ……72
4.5 停机操作 ……74
4.6 习题 ……75
第5章 LC-3结构 ……77
5.1 ISA概述 ……77
5.1.1 内存组织 ……77
5.1.2 寄存器 ……77
5.1.3 指令集 ……78
5.1.4 操作码 ……78
5.1.5 数据类型 ……80
5.1.6 寻址模式 ……80
5.1.7 条件码 ……80
5.2 操作指令 ……80
5.3 数据搬移指令 ……83
5.3.1 PC相对寻址 ……83
5.3.2 间接寻址 ……84
5.3.3 基址偏移寻址 ……85
5.3.4 立即数寻址 ……86
5.3.5 一个例子 ……87
5.4 控制指令 ……88
5.4.1 条件跳转指令 ……88
5.4.2 一个例子 ……90
5.4.3 循环控制的两种方法 ……91
5.4.4 例子：哨兵法数组求和 ……92
5.4.5 JMP指令 ……92
5.4.6 TRAP指令 ……93
5.5 例子：字符数统计 ……93
5.6 总结：数据通路（LC-3） ……96
5.6.1 数据通路的基本部件 ……97
5.6.2 指令周期 ……98
5.7 习题 ……98
第6章 编程 ……104
6.1 问题求解 ……104
6.1.1 系统分解 ……104
6.1.2 三种结构：顺序、条件、循环 ……104
6.1.3 实现三种结构的LC-3指令 ……105
6.1.4 回顾字符数统计例子 ……106
6.2 调试 ……109
6.2.1 调试的基本操作 ……110
6.2.2 交互式调试器的使用 ……111
6.3 习题 ……116
第7章 汇编语言 ……120
7.1 汇编语言编程——更上一层 ……120
7.2 一个汇编程序 ……120
7.2.1 指令 ……121
7.2.2 伪操作 ……123
7.2.3 例子：字符数统计程序 ……124
7.3 汇编过程 ……125
7.3.1 概述 ……125
7.3.2 两遍扫描 ……125
7.3.3 第1遍：创建符号表 ……126
7.3.4 第2遍：生成机器语言程序 ……126
7.4 相关知识 ……127
7.4.1 可执行映像 ……128
7.4.2 多目标文件 ……128
7.5 习题 ……129
第8章 输入/输出 ……134
8.1 输入/输出的基本概念 ……134
8.1.1 设备寄存器 ……134
8.1.2 内存映射I/O与专用I/O指令 ……134
8.1.3 异步I/O与同步I/O ……135
8.1.4 中断驱动与轮询 ……135
8.2 键盘输入 ……136

8.2.1 基本输入寄存器 ……136
8.2.2 基本输入服务程序 ……136
8.2.3 内存映射输入的实现 ……136
8.3 显示器输出 ……137
8.3.1 基本输出寄存器（DDR和DSR）…137
8.3.2 基本输出服务程序 ……137
8.3.3 内存映射输出的实现 ……138
8.3.4 例子：键盘回显 ……139
8.4 一个更复杂的输入程序 ……139
8.5 中断驱动I/O ……140
8.5.1 什么是中断驱动I/O ……140
8.5.2 为什么要引入中断驱动I/O ……141
8.5.3 中断信号的产生 ……142
8.6 内存映射I/O的回顾 ……144
8.7 习题 ……144
第9章 TRAP程序和子程序 ……146
9.1 LC-3 TRAP程序 ……146
9.1.1 概述 ……146
9.1.2 TRAP机制 ……146
9.1.3 TRAP指令 ……147
9.1.4 完整机制 ……147
9.1.5 I/O中断处理程序 ……149
9.1.6 HALT中断程序 ……150
9.1.7 寄存器内容的保存和恢复 ……151
9.2 子程序 ……152
9.2.1 调用/返回机制 ……153
9.2.2 JSR（R）指令 ……154
9.2.3 字符输入的TRAP程序 ……154
9.2.4 PUTS：写字符串 ……156
9.2.5 库程序 ……157
9.3 习题 ……160
第10章 栈 ……166
10.1 栈的基本结构 ……166
10.1.1 抽象数据类型：栈 ……166
10.1.2 两个实现例子 ……166
10.1.3 内存中的实现 ……167
10.1.4 小结 ……170
10.2 中断驱动I/O（第二部分） ……171
10.2.1 启动和执行 ……171
10.2.2 中断返回 ……173
10.2.3 例子：嵌套中断 ……173
10.3 基于栈的算术运算 ……175
10.3.1 栈的临时存储作用 ……175
10.3.2 例子：算术表达式 ……175
10.3.3 加、乘和取反 ……176
10.4 数据类型转换 ……180
10.4.1 一个错误结果的例子：2 + 3 = e 181
10.4.2 ASCII/二进制转换 ……181
10.4.3 二进制/ASCII转换 ……183
10.5 模拟计算器 ……184
10.6 习题 ……188
第11章 C语言编程概述 ……192
11.1 我们的目标 ……192
11.2 软硬件结合 ……192
11.3 高级语言翻译 ……194
11.3.1 解释执行 ……194
11.3.2 编译执行 ……194
11.3.3 两种方法的优缺点 ……194
11.4 C编程语言 ……195
11.5 一个简单的C程序 ……197
11.5.1 main函数 ……198
11.5.2 编程风格 ……198
11.5.3 C预处理器 ……199
11.5.4 输入和输出 ……200
11.6 小结 ……202
11.7 习题 ……202
第12章 变量和运算符 ……204
12.1 概述 ……204
12.2 变量 ……204
12.2.1 三种基本数据类型：int、char、double ……204
12.2.2 标识符的选择 ……206
12.2.3 局部变量和全局变量 ……206
12.2.4 更多的例子 ……208
12.3 运算符 ……208
12.3.1 表达式和语句 ……209
12.3.2 赋值运算符 ……209
12.3.3 算术运算符 ……210
12.3.4 算术优先级 ……210
12.3.5 位运算符 ……211
12.3.6 关系运算符 ……212
12.3.7 逻辑运算符 ……213

12.3.8 递增/递减运算符 ……214
12.3.9 运算符混合表达式 ……215
12.4 基于运算符的问题求解 ……215
12.5 编译器处理 ……217
12.5.1 符号表 ……217
12.5.2 变量的空间分配 ……218
12.5.3 完整的例子 ……219
12.6 补充话题 ……221
12.6.1 三种基本类型的变种……221
12.6.2 文字常量、常量和符号值 ……222
12.6.3 存储类型 ……223
12.6.4 更多的C运算符……223
12.7 小结 ……224
12.8 习题 ……224
第13章 控制结构 ……227
13.1 概述 ……227
13.2 条件结构 ……227
13.2.1 if语句 ……227
13.2.2 if-else语句 ……229
13.3 循环结构 ……231
13.3.1 while语句 ……231
13.3.2 for语句 ……233
13.3.3 do-while循环……236
13.4 基于控制结构的问题求解 ……236
13.4.1 问题1：π的近似值求解……236
13.4.2 问题2：找出100以内的质数 ……239
13.4.3 问题3：分析一个E-mail地址……241
13.5 其他C语言控制结构 ……243
13.5.1 switch语句 ……243
13.5.2 break和continue语句……244
13.5.3 简单计算器的例子 ……244
13.6 小结 ……245
13.7 习题 ……246
第14章 函数 ……250
14.1 概述 ……250
14.2 C语言中的函数 ……250
14.2.1 带参数的函数 ……251
14.2.2 求解圆面积 ……253
14.3 C语言中函数的实现 ……254
14.3.1 运行时栈 ……254
14.3.2 实现机制 ……254
14.3.3 汇总 ……259
14.4 问题求解 ……260
14.4.1 例1：大小写转换 ……260
14.4.2 例2：毕达哥拉斯三角形 ……262
14.5 小结 ……263
14.6 习题 ……263
第15章 测试与调试技术 ……268
15.1 概述 ……268
15.2 错误类型 ……269
15.2.1 语法错误 ……269
15.2.2 语义错误 ……269
15.2.3 算法错误 ……270
15.3 测试 ……271
15.3.1 黑盒测试 ……271
15.3.2 白盒测试 ……272
15.4 调试 ……272
15.4.1 特定方法 ……273
15.4.2 源码级调试工具 ……273
15.4.3 断点 ……273
15.5 正确的编程方法 ……274
15.5.1 明确规格说明 ……274
15.5.2 模块化设计 ……275
15.5.3 预防错误式编程 ……275
15.6 小结 ……276
15.7 习题 ……276
第16章 指针和数组 ……280
16.1 概述 ……280
16.2 指针 ……280
16.2.1 声明指针变量 ……281
16.2.2 指针运算符 ……282
16.2.3 指针传递一个引用 ……283
16.2.4 空指针 ……284
16.2.5 语法 ……284
16.2.6 指针例程 ……285
16.3 数组 ……286
16.3.1 数组声明 ……286
16.3.2 数组应用 ……287
16.3.3 数组参数 ……289
16.3.4 C语言的字符串 ……290
16.3.5 数组与指针的关系 ……293
16.3.6 实例：插入排序 ……293

16.3.7 C语言数组的不足 ……295
16.4 小结 ……296
16.5 习题 ……296
第17章 递归 ……299
17.1 概述 ……299
17.2 什么是递归 ……299
17.3 递归与循环 ……300
17.4 汉诺塔 ……301
17.5 斐波纳契数列 ……304
17.6 二分查找 ……307
17.7 整数转换为ASCII字符串 ……309
17.8 小结 ……310
17.9 习题 ……310
第18章 C语言中的I/O ……315
18.1 概述 ……315
18.2 C标准库函数 ……315
18.3 字符I/O操作 ……315
18.3.1 I/O流 ……316
18.3.2 putchar函数 ……316
18.3.3 getchar函数 ……316
18.3.4 缓冲I/O ……316
18.4 格式化I/O ……317
18.4.1 printf ……317
18.4.2 scanf ……318
18.4.3 可变长参数 ……320
18.5 文件I/O ……321
18.6 小结 ……322
18.7 习题 ……323
第19章 数据结构 ……324
19.1 概述 ……324
19.2 结构体 ……324
19.2.1 typedef ……326
19.2.2 结构体在C中的实现 ……326
19.3 结构体数组 ……327
19.4 动态内存分配 ……328
19.5 链表 ……330
19.6 小结 ……336
19.7 习题 ……336
附录A LC-3指令集结构 ……339
附录B 从LC-3到x86 ……353
附录C LC-3的微结构 ……367
附录D C编程语言 ……383
附录E 常用表 ……403

⊖ 附录F部分习题答案请见以下网址http://www.mhhe.com/patt2。

第1章　欢迎阅读本书

1.1　我们的目标

欢迎阅读本书。我们的任务是向你介绍“计算世界”（the world of computing）。本书的主要目标之一，是希望你能认识到计算世界并非如此神秘。相反，计算机（computer）是非常“确定”的一个系统，即在任何时候，在相同的方法、相同的状态下（当然还包括相同的起始条件），同样的问题必然获得相同的结果。其实，计算机并不是什么电子天才，相反，它只是一个电子傻瓜，只会精确地按照我们的要求去执行任务，它本身是没有心智的。

事实上，计算机这样一个复杂的机体（organism），是由一堆简单的部件，经过精心的系统组合而成的。本书首先将介绍那些简单部件的原理和机制，然后一步一步地搭建出一个互连结构，即所谓的“计算机”。这如同一幢房子的建造过程，先是从最底层的“地基”开始，自底向上一层层地“添砖加瓦”，最后形成一个功能完整的计算机。在逐层的讲述过程中，每增加一层，我们都将解释在做什么，以及新出现的概念与其底层组织之间的关系等。我们的目标是，一旦你完成了这本教材的学习，就能够自然地操纵一种语言（如C语言）来编写程序了，并能使用其中的一些高级功能，同时也能理解在程序执行过程中，计算机底层所发生的相应运作。

1.2　怎么才能做到

从第2章开始，是基于这样一种理念，即计算机不过是个电子设备，它由许多电子部件组成，而这些部件又通过导线相连。在任何一个时刻，这些导线要么是高电平、要么是低电平。但是在理解计算机这种电子设备的时候，我们并不关心具体的电压值是多少。换句话说，电压是115V还是118V并不重要，我们所关心的只是“相对于0V电压，它是否足够大”。如果该电压与0V电压相差很小，则在逻辑标识上将它定义为“0”；而如果电压与0V相差很大，则将它定义为逻辑“1”。

同时，通过0和1序列的组合，我们可以表示任何信息。例如，可以将字母a表示为01100001，十进制数35则表示为00100011。后面将详细介绍这种编码体系的原理。

当我们开始习惯了通过0和1编码来表示信息，以及基于这种编码的操作（如“加”操作）之后，就会进入下一个问题：“计算机是怎么工作的？”在第3章中，将介绍怎样用晶体管构建现代微处理器，具体地说，就是怎样使用晶体管来构建能够运算的部件（如“加法器”）和存储信息的记忆部件（如“内存”）；随后，第4章介绍“冯·诺伊曼（Von Neumann）机器”，它是一个描述计算机应该怎样工作的模型；第5章介绍一个简单的机器，即LC-3（Little Computer 3）。从LC-1开始，LC-3已经历了两版修改，它具备现代微处理器应具备的所有特性。所谓现代微处理器，如Intel 8088（用于1981年的IBM PC）、Motorola 68000（用于经典的1984年的Macintosh），以及奔腾IV（它是2003年高性能PC的首选处理器之一）等微处理器产品，而LC-3具备这些真实微处理器产品所具备的所有重要特性，但又不像这些“真家伙”那样复杂，因而很容易理解和掌握。

在理解了LC-3的工作原理之后，下一步就是要对它编程。开始是采用其自身语言即LC-3机器语言编程（第6章）；然后采用汇编语言编程（第7章），当然，汇编语言相对人的自然语言来说，多少还是有点儿让人不习惯；第8章介绍有关LC-3是怎样输入/输出信息的问题；第9章涉及两个很

重要的LC-3机制，即TRAP和子程序调用；第10章将对LC-3编程做全面总结，同时引入两个重要的概念，即“栈”（stack）和“数据转换”（data conversion）。最后，将给出一个稍微复杂的例子，它是一个基于LC-3编程实现的手持计算器。

本书的第二部分（第11~19章）将注意力转向高级编程语言——C语言。我们将深入介绍C语言的很多实现机制，这些内容在一般的入门教材中是不会介绍的。在几乎所有的例程中，我们都会将高层的C结构（construct）和底层的LC-3实现联系起来，以使你明白在使用C程序中的特定结构时，它对底层计算机存在什么要求。

我们讲解C语言的方法是，先介绍基本概念如变量和操作符（第12章）、控制结构（第13章）、函数（第14章）等；之后，是高级话题如C语言调试技术（第15章）、递归（第16章）、指针和数组（第17章）等。

C语言部分的结束篇是两个常见的高层结构，即C语言的输入/输出（第18章）和链表（第19章）。

1.3 两个反复出现的理念

在本书中，有两个理念将反复出现并反复强调：一是“抽象”，二是“在脑子里不要对硬件和软件做任何区分”。这两点非常重要！我们希望每个人都能认识到其中的价值，这也正是我们不断向工程专业和计算机科学专业的学生所灌输的。如果你想成为一个高级工程师或计算机科学家，领会这两点远比你理解计算机是怎样工作或怎样对它编程更为重要。随着对全书学习的不断深入，相信你对它们的理解会越来越清晰。

“抽象”理念（notion of abstraction）非常重要，它是学习的重点，也是在实践中要把握的核心理念。不管你未来是要做数学家、物理学家、工程专家，还是要做商业人士，抽象理念都非常有用，很难想像有哪个学科或知识体系中不需要“抽象”。同样，将硬件和软件做明显的区分也是错误的，这对未来的工作和学习都会造成误导。下面我们就来阐述这两个永恒的理念。

1.3.1 抽象之理念

抽象（abstraction）在生活中普遍存在。当我们搭乘出租车的时候，如果我说“去飞机场”，那么我使用的就是抽象的表达方式。为什么呢？因为我还可以用另一种表述方式，详细告诉他到达目的路线的每一个步骤：“顺这条街道向前过10个街区，左转”，然后当他到达这里之后，我又告诉他“现在顺着这条街道继续5个街区，右转”，如此继续。显然，你知道其中的细节，但这远不如告诉司机你要去机场来得简洁。如果还想进一步细化，你甚至可以将“顺这条路向前10个街区……”这句话分解为“踩油门”、“转方向盘”、“注意过往车辆和人行道”等这样的动作细节，但显然没有这个必要。

学会“抽象”是个重要的进步，它让我们学会站在更高的层次看问题，从而将事物的本质表现出来，而将其中的细节隐藏起来；它让我们学会更有效地使用时间和大脑；它让我们在分析问题时不至于陷入泥潭。

当然，其中存在这样一个假设，即“假设各个方面的细节都是运转正常的”。但是，如果底层细节的工作并不是完全正常呢？这是一个挑战，在这种情况下，要求我们不仅要具备抽象的能力，还要具备“分解抽象”的能力，这样才能保证问题的顺利解决。有人又称之为“解析”过程，即从抽象回到具体的过程。

此刻，让我想起两个小故事：

第一个故事是我很久以前穿越亚利桑那州的一次旅行。那是一个炎热的夏天，我当时的家是在常年温和的帕洛阿尔托（属加利福尼亚州）。为了应对亚利桑那州的炎热天气，我在出发之前去

机械师那里改装车子的制冷系统。注意，我在这里用了一个很抽象的说法——“制冷系统”。然而，我忽略了一个细节，应对帕洛阿尔托的天气所需要的制冷系统远不足以对付亚利桑那州沙漠的炎热。结果你一定猜到了，制冷系统在到达目的地之前出问题了，结果我被迫在Deer Lodge（亚利桑那州人口第三大的城市）待了两天，等待维修所需要的盖板密封圈到货。

第二个故事（可能是杜撰的传闻）发生在电力发电时代的早期。通用电气公司的一个大发电机出故障了，但面对发电机前板上一大堆的仪表盘和旋钮，所有的人都束手无策。大家都知道，调整其中的某些旋钮就可能解决问题，但谁也无法确定是其中的哪些旋钮，以及应该是顺时针还是逆时针旋转、转多少角度？正在这时，请来了电力厂创建初期的一个大师级人物。他看了一眼仪表盘，又仔细地听了一会儿电机的声音。然后，他从口袋里拿出一个螺丝刀，将其中的一个旋钮逆时针旋转了35°，机器正常了！随后，他为自己这两分钟的工作开出了一张1000美元的收费单（这在当时是一笔巨款）。控制中心接到这份账单时很不情愿，于是请求他开一个具体的明细账单，以说明收费理由，新账单的明细如下：

（1）将旋钮逆时针旋转35°： $0.75
（2）知道旋转哪个旋钮以及旋转多少度： $999.25

两个故事所表达的信息是相同的，即“抽象”能提高我们的效率，从而摆脱细节的纠缠。如果事情不存在什么意外，就会一切OK！即如果我不是去亚利桑那州旅游的话，抽象词“制冷系统”就足够了，而当时我却忽略了告诉机械师要穿越亚利桑那州沙漠这个“细节”。同样，如果不是电机发生了意外故障，大师对电机的深刻理解也就派不上用场了。

从这两个故事中，我们获得的启示是：当设计一个由各种门电路组成的逻辑电路时，千万不要深陷门电路的内部原理，因为这会大大拖延设计进度。你应该将其中的每个门电路都看做是现成的、可靠的；而仅当电路不工作的时候，才去研究门电路的内部结构，也只有这样，才能发现问题的症结所在。

再如，当你设计一个复杂的计算机应用程序，如电子表格处理系统、字处理系统或计算机游戏时，你可以将其使用到的每个组件都看做是一个“抽象”。此时，探究每个组件的细节是毫无意义的，那只会让你的工作永远无法结束。但当系统出现问题时，要想发现问题所在，就必须深入到每个组件的实现机制。

抽象技能相当重要。我们的观点是，抽象的层次越高越好，而且它与工作效率成正比。本书的做法是逐步提高抽象层次，我们先是基于晶体管描述逻辑门的实现机制，但一旦你领会了逻辑门的抽象，晶体管将永不再提；随后，就是基于逻辑门来构建更高层次的结构，而一旦我们理解了这些结构，逻辑门又将被丢弃。

结论

“抽象”能提高我们的思考效率。换句话说，忽略抽象之下的细节，会让我们更有效率。希望不要有人一听到“抽象”就反感哦！相反，你应该感谢它，抽象确实能提高我们的效率。

如果我们不需要将一个组件（component）和其他的东西相结合（以构建更大的系统），并且组件内部也不会出问题，那么，将认识停留在抽象层面就万事大吉了。但实际情况是，我们肯定会需要将这些组件拼装成更大的系统，而这些组件结合在一起工作的时候，也难免会出错误。这就意味着，我们既要不断地提高抽象层次，又要注意细节的深入。

1.3.2 硬件与软件

许多计算机科学家或工程师称他们自己是搞硬件的或是搞软件的。硬件通常指一个“物理的”计算机以及和它相关的方方面面；而软件通常指程序，如操作系统UNIX或Windows、数据库系统

Oracle或DB-terrific、应用程序Excel或Word等。他们的这种说法是暗示他们对其中的某一方面相当精通，而对另一方面知之甚少。听起来好像在软件和硬件之间存在一堵很高的墙，硬件是描述有关计算机怎样工作的，而软件的主导则是程序，你要做的就是选择待在墙的哪一边。

当你开始学习和接触计算机的时候，我们希望你抛弃这种观点。因为在我们看来，硬件和软件只是计算机系统中两个组成部分的名称而已；对设计者来说，具体将计算机的某个功能划分给哪部分来实现，以及它们之间如何协同工作，原则只有一个：让计算机工作得最棒（而不是刻意要区分它们）！

处理器的设计者如果懂得运行在处理器之上的程序需求，那么所设计的处理器必然比那些不懂的人所设计的处理器要快。例如，Intel、Motorola等大牌处理器设计厂家，在许多年前就意识到，未来的程序如E-mail、视频游戏、视频电影等，将大量包含视频信息（video clip），未来的处理器必须保证它们的执行性能。结果是，在他们所设计的处理器中，大都内嵌了专用视频处理硬件。如Intel为此提出的MMX指令集及MMX专用执行硬件，而Motorola和Apple也做了类似工作，如AltaVec指令集及其硬件。

软件设计中也有类似的故事。懂硬件特性的软件设计师所设计的程序，其运行性能远高于那些不懂硬件的人所设计的程序。“排序”是一个经典的计算任务，几乎在所有的大型软件中都不可或缺。我们需要将一系列的条目（item）按照一定的顺序排列，如字典中的单词需要按字母排序，学生成绩单是按数字排序的。存在太多有关排序的编程方法（又称算法）。但是，Donald Knuth在他的传世巨著《计算机程序设计艺术》（第3卷）中，竟然花了391页的篇幅专门讲述排序，因为要想做到排序最快，在很大程度上取决于软件设计者对硬件特性的了解。

结论

我们相信，不管你未来的职业取向是计算机软件还是硬件，两者都懂必然会使你更强。本书的宗旨就是让你两者都掌握。有时我们在讲述一个概念的时候，并未特意强调是关于软件或是关于硬件的，但通常是两者都相关的。

当你在学习数据类型（data type）这个软件概念时（第12章），你将理解硬件中字（word）的有限长度，是怎样影响软件中数据类型表示的。

当你学习函数（function）时（第14章），你会联想起硬件的知识，从而明白“函数调用规则”的含义和意义。

当你学习递归（recursion）——一个强大的算法工具时（第16章），结合硬件知识，你将明白为什么花些时间递归执行过程（procedure）是值得的。

当你学习指针（pointer）变量时（第17章），有关计算机内存的知识将更有助于深入理解指针，从而知道什么时候适合使用它，什么时候不适合使用它。

当你学习数据结构（data structure）时（第19章），有关计算机内存的知识，将帮助你理解数据结构在内存中的具体实现，以及有效操作数据结构的窍门。

我们知道，前面的内容中所出现的很多名词让你感到迷惑，不要紧，在本章结束的时候再重读一遍以上内容即可。目前你仅仅需要认识到：软件中的许多重要话题是和硬件中的话题紧密交织的。我们的观点是：无论你更倾向于其中的哪一面，从两方面思考必然会使其更容易。

更重要的是，面对大多数的计算问题，如果解题者具备软、硬件两方面的知识，那么他或她给出的答案会更漂亮。

1.4 计算机系统简述

在前面的章节中，我们已多次使用了“计算机”这个词，但并未直接解释过其定义。它是指这样一种机制，即同时在做着两方面的事情：既控制着信息的处理过程，同时也是信息处理过程

的具体执行者。所谓“控制着信息的处理过程”(directs the processing of information)，指的是它必须决策下一个执行任务是什么，而“处理过程的具体执行者”，意味着它必须具备“加”、“乘”等运算能力以产生执行结果。该机制更准确、更合适的称谓应该是“中央处理器”(Central Processing Unit, CPU)。本书的重点也是围绕CPU及其之上的程序运行而展开的。

在20年前，一个处理器由10个或更多的18英寸电路板组成，每个电路板上包含了大约50个部件（采用集成电路封装，如图1-1所示）。而今天的处理器通常由一个微处理器芯片组成，其大小仅为一片1英寸左右的硅材料，其中包含了大约几百万个晶体管（如图1-2所示）。

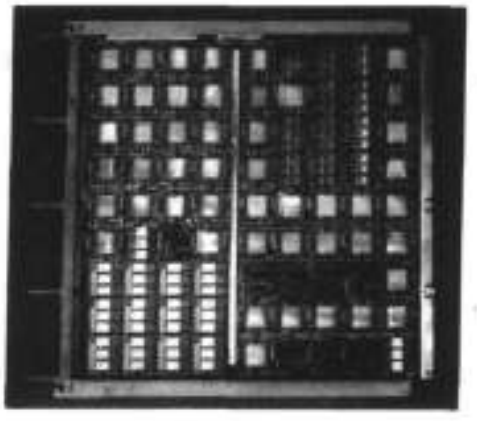


图1-1 处理器板，1980年产

资料提供：Emilio Salgueiro, Unisys Corp.

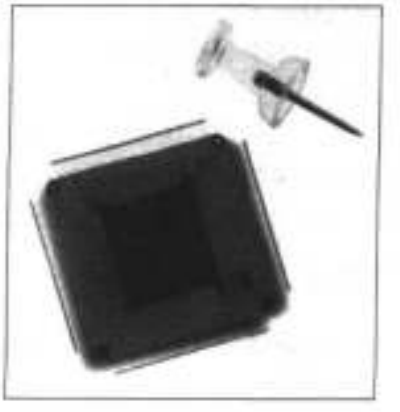


图1-2 微处理器，1998年产

资料提供：Intel Corp.

然而大多数人更熟悉“计算机”这个词，它包含了比处理器更多的意思。一个计算机系统(computer system)由更多的部件组成（如图1-3所示），除了处理器之外，还包括键盘（用来输入命令）、鼠标（用来点击菜单）、显示器（用来显示计算机系统产生的信息）、打印机（用来打印信息的拷贝）、内存（用来临时存储信息）、磁盘和CD-ROM（用来永久存储信息及很多可以执行的程序或软件）。

这些附加的部件更方便了计算机最终用户的使用。例如，如果没有打印机，就只能手抄屏幕上显示出来的信息，而没有鼠标，你将永远手工输入各种命令，不像现在这样，轻轻点击鼠标按键，就可以启动命令。



图1-3 个人计算机（Personal Computer, PC)

资料提供：Dell Corp.

因而，在我们开始本书的旅途之际，需要声明的是，本书的重点落在那1英寸的空间内部，即CPU。但若没有其他这些部件（虽然它们不是本书的重点内容），计算机使用起来就不会那么方便了。

1.5 两个非常重要的思想

在结束第1章之前，我们还将介绍两个非常重要的思想。它们非常重要，论述了计算(computing)的全部内涵，因而我们希望你能了解它们。

第一：所有的计算机（不管是最大的还是最小的、最快的还是最慢的、最贵的还是最廉价的），只要给予足够的时间和内存，它们所能完成的计算任务是相同的。换句话说，最快的计算机能够完成的事情，最慢的计算机也一样能够完成，只是更慢一些而已；而一个便宜的计算机所不能完成的事情（如果有足够内存的话），对于一个更昂贵的计算机来说，同样也是无法完成的。总之，所有的计算机能够完成完全相同的事情。只是有些计算机可能做得更快些，但绝不会做得更多。

第二：我们用英语或其他语言给出了一个问题，然而计算机却能通过电子运转（运行程序）来解决这个问题，太奇妙了！至于怎样把用人类语言描述的问题转换成能够影响电子运转的电压，需要一系列的、系统的转换过程。在计算机的50年历史里，这一转换问题竟然被成功解决了，而且这一复杂的转换任务竟然是由计算机本身完成的。看起来不可思议，但确实如此。

本章后面的内容将详细阐述这两个思想。

1.6 计算机：通用计算设备

一本入门性的教科书先描述计算机是怎么工作的，感觉有些奇怪吧？机械专业的学生是先学物理，然后才是汽车发动机的工作原理；化学工程专业的学生是先学化学，而不是石油提炼。那为什么计算机专业的学生要先学计算机设备呢？

答案是：计算机是特别的。要学习计算机的基本原理，必须先了解计算机是怎样工作的。其原因在于计算机被称为“通用计算设备”(universal computational device)，不理解吗？请看下面的解释。

在现代计算机出现之前，曾经出现了很多能够计算的机器。其中，有些是模拟机(analog machine)——即机器产生的结果是用可测量的模拟量来表示的（如电压、距离等）。例如，滑动计算尺计算乘法的机制，就是滑动其中的一个对数等分尺，然后从第二个尺子上读出其对数“距离”。还有些早期的加法器，其工作原理是采用在秤盘上加重量的方式。模拟机器的缺陷主要是难以提高其精度。

同模拟机相比，数字机(digital machine)通过一组固定的、有限的数字和字符来完成操作，这就是为什么数字机最终主宰计算世界的原因。你应该很熟悉模拟手表和数字手表之间的区别[⊖]。模拟手表有时针、分针，以及秒针，它通过这些针的位置来表达具体的时间；而数字手表通过数字来表达时间。对于数字手表，提高其精度的办法是增加数字数目，如10 : 35 : 16（而不是10 : 35）。但对于机械手表，该怎样表达百分之一秒这样的精度呢？可以的，但你或许需要再加一个更长的秒针（应该比已有的秒针更长些）。本书在提到计算机的时候，讲述的对象是数字计算机。

在现代数字计算机出现之前，在西方最常见的数字机是加法器(adding machine)，而在东方用的则是神奇的算盘。加法器是一种能执行特定“加”功能的机械或机电设备。另外，还有各种各样的其他数字机，有的能够执行整数乘法，有的能对一堆有打孔标识的卡片做字母排序。所有这些机器的局限性在于：它们都只能做一件特定的工作。例如，如果你有一台只能做加法的机器，那么做两个数的乘法运算时，还得用纸和铅笔。

但计算机就不一样了。你需要告诉计算机的是“方法”和“任务”，即怎样做加法，怎样做乘法，怎样对一堆表单做排序或任何其他计算任务。而假若你又想出了新的计算方法，就不需要再

⊖ 生活中我们俗称它们为机械表和电子表，但这种说法已不太准确，如石英表就是机械和电子混合的，但石英表属于模拟表。——译者注

购买或设计新机器了，所要做的惟一事情，就是在原来的计算机上增加一些新的指令（instruction）或程序（program）即可。这就是我们称计算机为“通用计算设备”的原因。计算机科学家相信“任何事情都是可计算的”，换句话说，也就意味着任何事情都是可被计算机所运算的（只要给它足够的时间和内存）。当我们学习计算机的时候，我们要学习计算的基本原理，即计算是什么？怎样实施计算？

通用计算思想的产生要归功于Alan Turing（阿兰·图灵）。1937年，图灵提出一个大胆的假设：任何计算都可以由这样一台机器来完成。这个机器就是图灵机（Turing machine）。他为这类机器给出了一个清晰的数学描述，但没有为之建造一台真正的机器。世界上真正可运行的数字计算机是在1946年才出现的。图灵的兴趣在于解决一个哲学问题：计算的可定义性。于是，他开始观察人们在计算时所采用的各种方法和行为，其中包括：在纸上做标记（mark），按照一定的规则记录符号（symbol）等行为。并将这些行为抽象出来，定义了一种能够表达它们的机制；同时，他还给出了一些例子，来解释基于这种机制是怎样完成一些特定任务的，例如，用图灵机来完成两个整数的加法及乘法。

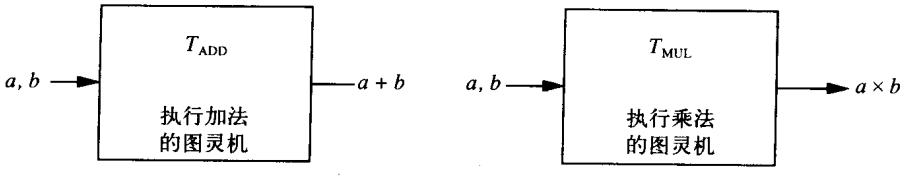


图1-4 图灵机的黑箱模型

图1-4分别是图灵机加法和乘法的“黑箱”模型，要完成的任务描述如箱体内文字所示，被操作数据从箱体输入，操作结果从箱体输出。黑箱模型并不具体说明任务是怎样完成的，但实际上有很多种方法都可以实现两个数的加法或乘法。

图灵提出，任何计算都可以通过某种图灵机来完成，我们称这个理论为“图灵论题”（Turing's thesis）。虽然图灵论题从来没有被严格证明，但众多的证据都表明它是正确的。同时我们发现，任何试图对图灵机做出的改进尝试，最终都可以用图灵机本身来实现。

有关图灵机理论的最好阐述，还要数图灵本人的那份论文。他这样说道：要构建一个比任何形式的图灵机都要强大的机器，则这个机器U必须能仿真所有的图灵机。假设，我们希望机器U能仿真图灵机中两个整数相加的模型，则给定输入，U应该能输出对应的求和结果。随后图灵证明，图灵机也可以做到这点。由此证明了，任何试图发现图灵机所不能做的事情的企图都是失败的。

图1-5进一步阐明了这一点。假设要计算“$g \times (e+f)$”，很简单，你只需要向U提供“加法”图灵机和“乘法”图灵机的描述，以及三个输入参数e、f和g，剩下的事情交给U来做即可。

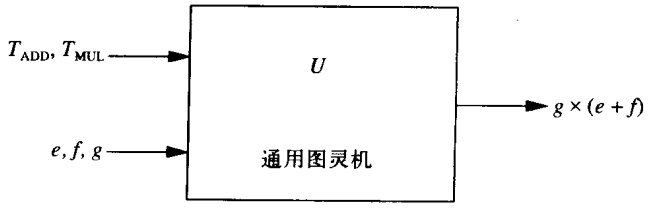


图1-5 通用图灵机的黑箱模型

在通用图灵机U的描述中，图灵第1次深入地阐述了一个论题，即计算机能做什么。换句话说，一台计算机（拥有足够的内存）和一个通用图灵机之间，它们所能完成的事情是一样的。它们中的任意一个，只要给定计算任务的描述及相关数据，都能计算出结果。计算机或图灵机能够计算任何可计算的任务，因为它们都是可编程的。

因此我们说，即使是一台昂贵的大型计算机，它所能完成的事情并不比一台廉价的小计算机多多少。多花的钱只是使你拥有的计算机速度更快、显示器的分辨率更高、声卡系统音质更好。如果你的计算机是一台廉价的个人计算机，你就已经拥有了一台通用（Universal）计算机。

1.7 从问题描述到电子运转

图1-6描述的是要控制电子（器件）按我们的意图工作所经历的整个过程。我们称这个过程的每个步骤为“转换层次”，其中的每一层都有多种实现选择，如果我们忽视其中的任一层，设计完美计算机系统的愿望都会失败。

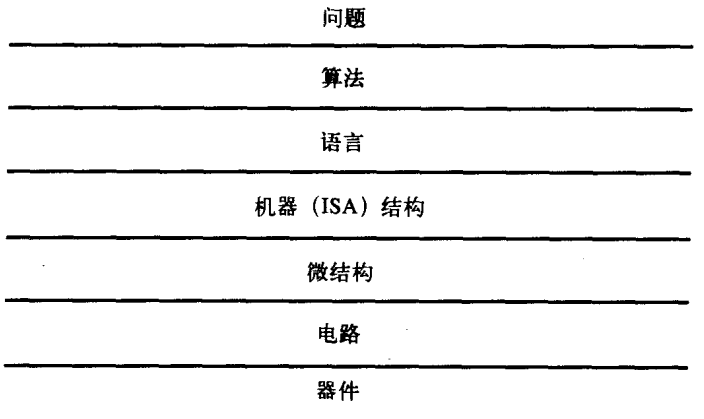


图1-6 转换层次

1.7.1 问题的提出

描述问题的时候，我们采用“自然语言”，即人们所说的语言，如英语、法语、日语、意大利语等。这些语言都经历了几百年的发展，其中包含了太多的不适合作为计算机语言的东西，其中二义性特征最为突出。自然语言中包含了太多的二义性，如不同的说话声调和音量，以及不同的上下文句子内容，都会使得同一段话表现出不同的语义。

英语中有个经典例句：“Time flies like an arrow”（光阴飞似箭）。我们至少可以有三种解释方法，取决于下面三种场景：

（1）一个人正在注意时间的流逝有多快。

（2）一个人正在做昆虫赛跑比赛的相关工作。

（3）一个人正在给昆虫爱好者Abby写信。

其中，第一种情况是一个明喻，表示时间的流逝如箭一般；第二种情况是这个人告诉时间记录员他/她应该像箭一样快速工作；第三种情况是指一种特殊的flies(time flies) like arrow, 而另一种flies(fruit flies) like banana[⊖]。

类似的二义性对于计算机指令来说是不能接受的。计算机是一个电子傻瓜，它只能做你让它做的事情，但如果你给它的命令存在有多种意思（即二义性），它就会茫然不知所措了。

1.7.2 算法

从问题的提出开始，向下转换的第一步是将问题的自然语言描述转换为算法（algorithm）描

⊖ 源自语言学上的一个例子：“Flies like an arrow, fruit flies likes a banana.”（光阴飞似箭，果蝇喜欢香蕉），外形完全一样的句子，结构完全不同，这里是个笑话。——译者注

述，去除那些无用的特性。算法描述的特点是流程化、步骤清晰，并确保该流程能终止。其中每个步骤的定义描述都足够精确，以保证能在计算机上执行。这些特性可以阐述如下：

- 确定性（definiteness）。表明每个操作步骤的描述是清晰的、可定义的。我们认为“制作烙饼的过程说明”是缺乏定义性的，因为它说“拼命搅动直到成为糊状”（stir until lumpy），其中“糊状”（lumpy）的程度表述就是模糊的。
- 可计算性（effective computability）。表示每一步的描述都可被计算机执行。而“取最大的素数”这样一句描述，我们称其不具备可计算性的，因为根本就不存在“最大”的素数。
- 有限性（finiteness）。即过程是会终止的。

任何一个问题都具有多种求解算法。其中，有的算法步骤最少，有的算法允许个别步骤反复执行，等等。值得注意的是，如果一个算法虽然描述出来的步骤很多，但如果这些步骤在一个时刻可以同时被计算机执行（并行），无疑该算法的执行速度是快的。

1.7.3 程序

下一步就是将算法转换为程序，即用编程语言描述。编程语言属于“机械语言”，与自然语言不同的是，机械语言不是设计出来为人所讲的，相反，它被设计成严格的顺序方式，以便让计算机顺序地执行指令序列。换句话说，它不存在二义性问题。

大约有1000多种程序语言。有的是有“特定用途”的（如Fortran是科学计算语言、COBOL是商业数据处理语言），而本书后半部分介绍的C语言是专门为底层硬件操作而设计的语言。

还有用于其他一些目的的语言，如Prolog语言常用来设计专家系统，LISP语言则被那些研究人工智能的人们所青睐，而Pascal语言则成了学生学习计算机语言的教学语言。

计算机语言可以分为高级语言和低级语言两类。高级语言和底层计算机的相关性很弱（距离很远），或称之为“机器无关”语言。前面提到的各种语言都属于高级语言。低级语言则和执行程序的计算机紧密相关，通常一种低级语言只对应一种计算机，我们称之为“某某机器的汇编语言”。

1.7.4 指令集结构

下一步的任务是将程序转换成特定计算机的指令集（instruction set）。指令集结构（Instruction Set Architecture, ISA）是程序和计算机硬件之间接口的一个完整定义。

ISA定义包括：计算机可以执行的指令集合，即计算机所能执行的操作，以及每个操作所需数据是什么，即操作数（operand）；ISA还定义了可接受的（legitimate）操作数表达方式，即数据类型（data type）；ISA还定义了获取操作数的机制，即定位各种操作数的不同方法，我们称之为“寻址模式”（addressing mode）。

不同的ISA定义的操作类型、数据类型和寻址模式的数目都是不同的。有的ISA只有几个操作类型，有的ISA则有上百个；有的ISA只有一种数据类型，有的则有几十种；有的ISA只有一、两种寻址模式，而有的则有20多种。如x86（PC机中的ISA），有100种操作类型、十几种数据类型、二十多种寻址模式。

ISA的设计中，还要折衷考虑计算机内存的大小及每个存储单元的宽度（即能容纳的0和1的数目）。

许多ISA一直延续至今，典型的例子就是x86。该ISA于1979年由Intel公司设计，目前同时被AMD和其他一些公司所采用。其他一些著名的ISA包括：PowerPC（IBM和Motorola）、PA-RISC（Hewlett Packard）、SPARC（Sun Microsystems）等。

将高级语言（如C）翻译为ISA指令（如X86）的过程，通常是由一个被称为“编译器”

(compiler)⊖的程序来完成的。例如，将C语言程序翻译成x86 ISA时，需要一个“x86的C编译器”。就是说，针对不同的高级语言和目标计算机组合，需要一个对应的编译器。

将特定计算机的汇编语言程序翻译为其ISA的过程，则是由汇编器（assembly）来完成的。

1.7.5 微结构

下一步的任务是将ISA转换成对应的实现。实现的具体组织（organization）被称为“微结构”(microarchitecture)。例如，近年来许多处理器都实现了X86这种ISA结构，但每个处理器的实现方法不同，即有自己的微结构。最早的实现是8088（1979年），而最新的则是Pentium IV（奔腾4，2001年）；再如Motorola和IBM已实现了几十种基于Power PC ISA的处理器，每一种也都有自己的微结构，其中最新的两种是Motorola的MPC7455和IBM的Power PC 750FX。

对于设计者来说，每次的新设计都是一次机会，设计者可以重新权衡新机器的性价比。设计永远是一个“权衡”的挑战练习，问题在于在高（或低）成本下，能否实现一个高（或低）性能的计算机。

汽车制造业为ISA和微结构（即ISA的实现）的关系提供了一个很好的比喻：ISA描述的是驾车人在车里看到的一切，几乎所有的汽车都提供了相似的接口（但汽车的ISA接口和轮船、飞机的ISA差别很大），所有的汽车中，三个踏板的定义完全相同，即中间的是刹车、右边的是油门、左边是离合器。ISA表达的是基本功能，其定义还包括：所有的汽车都能够从*A*点移动到*B*点，可前进也可后退，还可左右转向等。

而ISA的实现是指车盖板下的“内容”。所有的汽车其制造和模型都不尽相同，这取决于设计者在制造之前所做的权衡决策，如有的制动系统采用刹车片，有的采用制动鼓；有的是八缸发动机，有的是六缸，还有的是四缸；有的有涡轮增压，有的没有。我们称这些差异性的细节为一个特定汽车的“微结构”，它们反映了设计者在成本和性能之间所做的权衡决策。

1.7.6 逻辑电路

微结构最终是由一组简单的逻辑电路实现的，有多种实现方法可供选择，各种方法之间存在着性能和成本上的差异。例如，仅仅一个“加法器”，就存在着多种实现，它们所表现出来的运算速度（性能）及其成本差异非常大。

1.7.7 器件

最后要说明的是，每个基本的逻辑电路，都是按照特定的器件技术（device technology）来实现的。就是说，CMOS电路中所使用的器件与NMOS电路中所使用的器件是不同的，它们与砷化镓电路中所使用的器件也不同。

1.7.8 小结

综上所述，从用自然语言对问题进行描述开始，直到电子（器件）的实际运转，之间要经历很多层次的转换。如果我们能说“电子”语言，或电子能“听懂”我们说的语言，那么我们只需要走到计算机面前，直接向电子发布命令即可。然而，我们不会说电子语言，电子也听不懂我们

⊖ “编译器”是个广义的说法，有时又称为“compiler driver”。事实上，为完成编译工作，需要很多工具(toolkits)，如：预处理器（pre-processor）、编译器（compiler）、汇编器（assembler）、链接器（linker），以及一些必须的现成工具（bin-utility）如文档处理（archiver）等。——译者注

的语言，我们所能做的事，只能是进行一系列的转换。在转换过程中，每一层的实现又都存在很多选择方案，面对不同的方案，我们的最终决策和选择决定了系统实现的性能和成本。

本书将详细介绍转换过程中的每一个环节，比如晶体管是怎样实现逻辑电路的，逻辑电路是怎样构成微电路的，以及微电路怎样实现一个特定的ISA（我们的ISA是LC-3）。然后，从某个问题出发，讲述其C语言的描述，以及C语言又怎样转换成LC-3的ISA描述的全过程。

我们衷心希望本书能陪你度过一个愉快的学习过程！

1.8 习题

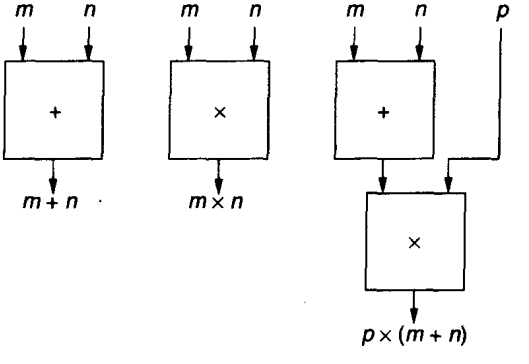
1.1 试解释1.5节中两个重要思想中的第一个。

1.2 试问，同汇编语言相比，高级语言是否能向底层计算机表述更多的计算方式？

1.3 试问，是什么原因使得模拟计算机难以实现，从而使设计者转向采用数字设计？

1.4 列举自然语言的特性之一，说明它为什么不适合直接作为编程语言？

1.5 假设我们有一个“黑箱子”，其输入为两个数字，输出为两数之和，如图1-7a所示；另外还有一个箱子能求两数的乘积，如图1-7b所示。假设我们有无穷多个这样的箱子，通过箱子的互连，我们可以完成如$p \times (m + n)$这样的运算，如图1-7c，请问：怎样通过互连完成如下的计算：



a) 加法　　b) 乘法　　c) 加法和乘法组合

图1-7 黑箱子的功能

a. $ax + b$。

b. 求四个输入w、x、y、z的平均值。

c. $a^2 + 2ab + b^2$（你能只用一个加法箱和一个乘法箱就完成吗？）。

1.6 试用自然语言写一句话，然后给出这句话可能的两种解释。

1.7 1.3.1节有关抽象的讨论给出的结论是：如果“底层所有一切都很好”，我们是没有必要去理解这些部件的组成的。但如果情况并不良好，我们还是要学会分解部件。在出租车的例子中，你并不知道怎样才能到达机场。采用抽象的方法，你只需要简单地告诉司机“带我去飞机场”即可。解释在什么情况下这种方法是有效的，在什么情况下会起到反作用。

1.8 John说：“I saw the man in the park with a telescope.”。这意味着什么？对这句话有多少种可能的解释？请列出来。这句话具备的什么特性，使得它如果出现在程序中是无法接受的？

1.9 试问，自然语言可以表达算法吗？

1.10 给出算法的三个特性，并给出简单的解释。

1.11 针对算法的每个特性，分别给出一个例子，在例子代码中，都缺少这个特性，说明此时为什么不能说它是一个算法。

1.12 请问：以下从a到e的描述是否是算法？如果不是，那么它缺乏算法的哪些特性？

a. 将下面矩阵中的第一行与其他首列不为0的行相加（注意：列是垂直方向的，行是水平方向的）；

$$\begin{bmatrix} 1 & 2 & 0 & 4 \\ 0 & 3 & 2 & 4 \\ 2 & 3 & 10 & 22 \\ 12 & 4 & 3 & 4 \end{bmatrix}$$

b. 为了证明素数的数目和自然数的数目一样多，创建一系列素数和自然数的配对单元，将第一个素数与1（即第一个自然数）配对，第二个素数与2配对，第三个素数与3配对，依次类推。如果结束的时候，每个素数都找到一个与其配对的自然数，即表明素数的数目和自然数的数目相同；

c. 给定两个矢量，每个矢量20个元素，试执行以下操作：取第一个矢量的第一个元素，与第二个矢量的第一个元素相乘；对第二个元素做相同的操作，依次类推。然后将所有的乘积相加（即点积（dot product））；

d. Lynne和Calvin都想带小狗出去散步，互不相让。Lynne于是拿出一个分币，提议抛硬币决定，但Calvin不相信Lynne（怀疑硬币有重量倾向，即总是某一面向上），于是提出一个新的方案：

1）连续抛两次。

2）如果第一次头像面向上，则下次选背面。

3）如果第一次背面向上，则下次选头像面。

4）如果连续两次都是头像面或背面，则重新开始（再抛两次）。

问：Calvin的方法是否符合算法标准？

e. 给定一个数，执行以下步骤：

1）乘以4

2）加4

3）除以2

4）减2

5）除以2

6）减1

7）此时，给计数器加1（表明刚执行了一遍如1~6的操作步骤）；同时判断刚才的减1结果（第6步）；如果为0，记下此时计数器的值，并停止；如果不为0，则从该减过1的数开始，再次执行以上的1~7步。

1.13 两台计算机*A*和*B*，除了*A*具有减法指令外，其他指令完全相同。两者都具有对一个数求负值的指令。试问：*A*和*B*两台计算机，哪一台能解决的问题更多？证明你的结论。

1.14 假设我们试图对一组名字做字母排序（sorting）。有一种算法为“冒泡排序（bubble sort）”；我们用C语言编写这个算法，对其编译并运行在x86 ISA的机器上；x86 ISA可以实现为Pentium IV微结构的方式。我们称这样一个序列为“冒泡排序、C语言编程、x86 ISA、Pentium IV微结构”的转换过程（transformation process）。再假设我们有四种排序算法，且可以用5种语言编程（C、C++、Pascal、Fortran、COBOL）；有面向x86和SPARC两种ISA的编译器，且有三种x86的微结构实现、3种SPARC的微结构实现。请问：

a. 共有多少种转换过程？

b. 列举其中三种转换过程；

c. 如果x86的微结构是2种（而不是3种），SPARC的微结构是4种（而不是3种），那么共有多少种转换过程？

1.15 将高级语言和底层语言相比较，列举一个优点、一个缺点。

1.16 列举至少三个ISA定义所包含的内容。

1.17 简单描述ISA和微结构之间的区别。

1.18 一种微结构可以实现多少种ISA？再反问，一种ISA可以在多少种微结构上实现？

1.19 列出转换过程的所有层次，并在每层中找一个例子。

1.20 图1-6所示的转换层次通常又被称为是不同的抽象层次。你认为这种说法是否合理？试用例子加以解释。

1.21 假设你去商店购买字处理软件。请问该软件通常以什么方式存在？是高级语言方式或是汇编语言？或是与你的计算机ISA兼容的格式？请回答。

1.22 假设给你一个任务，完成图1-6中某一层的转换工作，且只允许转换为下面的一层（即不允许“跨层转换”）。试问，你觉得哪一层的转换工作难度最大？为什么？

1.23 为什么通常会不断改变微结构实现却不改变ISA？例如，Intel公司为什么要确保Pentium III所实现的ISA一定要与之前的Pentium II一样？提示：当你升级计算机（如更换最新的CPU）之后，你希望丢弃原来的软件吗？

第2章 bit、数据类型及其运算

2.1 bit和数据类型

2.1.1 bit——信息的基本单位

如第1章所述，我们知道，计算机是一个包含了多层转换的系统。一个由自然语言（如英语）描述的问题，最终必须转换为计算机内部的电路工作（更具体地说是电子运动），才能得以解决。

在计算机内部，有数以百万计的器件在控制着电子的运动。这些器件随时监测着电路中各处电压的变化，并做出不同的响应控制。它们不仅监测电压的有无，甚至还能测量真正的电压大小。但这无疑会导致控制和检测电路的过度复杂性。如果只需要检测电路中任意两点间是否有电压，而不是测量其精确电压值，电路的复杂性无疑会大大降低。

举一个简单的例子：如果想知道室内插座上的电压，我们需要专用的电压表。这个电压值可能是120V，也可能是115V甚至是118.6V；但是，如果我们只是想知道插座是否有电（而不关心真实电压值究竟是以上三个中的哪一个），问题就简单多了。例如，我们有时在无意间把手指伸入电源插孔里，马上就会缩回来。

在符号层义上，我们采用“1”表示两点间存在电压，而“0”表示两点间不存在电压。我们称这样一个要么是“1”要么是“0”的符号单位为一个“bit[⊖]”，即所谓的二进制表示方法。回顾一下大家从孩童时候就熟悉的阿拉伯数字——0、1、2、3、…、9，由这10个数字，我们可以表示任意的十进制数。同样，表示二进制数时，需要且只需要两个数字（0和1）即可。

确切地说，计算机对有电平（即“1”）和无电平（即“0”）并不做严格的电压值定义。换句话说，“0”并不表示电路中就绝对不存在电压，它仅仅表示当前“0”所代表的电压比“1”所代表的电压更接近于电压值0，同样“1”表示它所代表的电压值与电压0的差值很大。例如，计算机定义2.9V的电压表示“1”，而0V电压表示“0”。但如果电路实际测得的电压值为2.6V，计算机也会将它当做“1”来处理；同样，如果实际电压为0.2V，计算机则将它当做“0”。

当然，计算机需要定义足够大的数值范围才能工作。但事实上，电路只有两个状态，即有电压（用“1”表示）和无电压（用“0”表示）。为了表示更多的数值状态，我们可以将多条线路合并使用，比如8个bit宽度（即8条线路），这样就可以表示256个不同状态(00000000~11111111)。通常，k个bit的组合可以表达（2^k）个不同状态，每个状态分别是k个0和1的bit序列组合。我们称该0和1的序列为编码，每个编码对应一个特定的值或状态。

2.1.2 数据类型

数值的表达方法多种多样，如数值5：十进制符号表示为符号“5”；也有人伸出5个手指头来表示，写出来就是“11111”（每个“1”代表一个指头），这种方法称为单进制；罗马人则采用字母“V”代表数值5。下面，我们将介绍第4种表示方法，也就是计算机采用的二进制表示法，数值5表示为“00000101”。

⊖ 随着计算机知识的普及，bit这个英文单词被越来越多的人所熟知。而中文中，对bit存在多种翻译，如“位”、“比特”，容易让读者感觉混乱。在后面的文字中，有时候不对bit做中文翻译。——译者注

只是能够表达“数值”还是不够的，计算机还必须具备操作这些数值的能力。如果我们不仅定义了数值的表达方式（或编码方式），同时还定义了相关的操作方法，则在定义上称该表达方式为一种数据类型。每个计算机指令集（ISA）都定义了一组数据类型及其相应的操作指令，数据类型的数目可多可少（取决于ISA的设计要求）。本书中，我们将在算术运算中采用补码（2's Complement）编码表示正、负整数，在键盘输入和显示器输出的应用中，采用ASCII码表示字符。关于这两种数据类型，在后面章节中会有详细解释。

除此之外，还存在很多其他数值表示方法。如高中化学课本中的“科学表示法”（scientific notation），即将数值621表示为6.21×10^2。现在，越来越多的计算机也开始支持这种表示方法，并提供相应的操作指令，我们称之为“浮点”（floating point）数据类型。在2.6节将详细介绍它。

2.2 整数数据类型

2.2.1 无符号整数

我们将要介绍的第一种“信息表示方法”或“数据类型”是无符号整数（unsigned integer）。无符号整数在计算机中的应用非常广泛。比如我们希望限定一个任务的执行次数，使用一个无符号整数即可，用它来记录该任务还有多少次执行次数。另外，我们还可以使用无符号整数表示不同内存单元的地址，这与生活中的用法很相似，如第129号大街或第131号大街。

我们可以将一个无符号整数表示为一连串的二进制数字序列（string of binary digits）。为了更好地理解这种表示方法，我们先回顾一下常见的十进制数。

你应该知道十进制数“329”是什么意思吧？这是一种“位值法”（positional notation）表示方法，即“位置”决定其真实“数值”大小。如“329”，不要看其中的数字“3”比数字“9”小（只有其三分之一大），但“3”的权重远比“9”更高，因为在这里“3”代表的是数值300（3×10^2），而“9”只代表9（9×10^0）。

二进制补码的工作原理与此相同，惟一的不同只是使用的数字只能是“0”或“1”，即基（base）为2（而不是10）。如使用5个bit来表示数值“6”，则表示为“00110”，即

$$6 = 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

有k个bit则可以表示2^k个无符号整数（从0到2^k-1）。如上面的5-bit例子，可表示的数值范围是0到31。

2.2.2 有符号整数

然而，在实际算术运算中存在大量的负数。应该怎样表示负数呢？我们可以将2^k个k-bit数一分为二，一半表示正数，另一半表示负数。那么，原先5-bit码所表示的0~+31数值范围，就变成了一半代表正数“+1~+15”，一半代表负数“−1~−15”，共计30个整数。事实上，5个bit可以表示32个码字，余下的两个码字没有指定。如果我们从中选择码字00000来表示数值0，这样还剩一个码字未分配。下面，我们将讨论如何处理这个多余的码字。

但是，我们仍然面临如何“分配”码字的问题，即究竟用哪些码字表示正数、哪些表示负数？

正整数比较好解决，仍然采用“位值法”表示。由于有k个bit，可以表示出2^k个码字，将其中的一半取出，表示（$0 \sim 2^{k-1}-1$）的正整数，我们发现这些正整数的最高位bit都为0。如在$k = 5$的情况中，可表达的最大正数是+15，即01111。

注意图2-1中给出的三种数据类型，它们在表示0和以0开始的正数时都完全一样，那么负数该怎么表示呢（以我们的5-bit为例，为−1至−15）？第一种思路是：以最高bit代表符号，0为正数，1为负数，这种方法称为符号位（Signed Magnitude）表示法；第二种思路，这也正是早期计算机如CDC6600所采用，其方法是：将一个正数的所有bit全部取反，即得到该正数所对应的负数编码，例

如“+5”表示为“00101”，那么“−5”则为“11010”，我们称之为*反码*（1's Complement）表示法。

此时，读者可能认为，至于采用什么编码方法应完全取决于设计者的喜好。不错！但不幸的是，不同的编码方法会导致不同的加法器逻辑复杂度。前面提到的两种编码方法，符号位法和反码，在硬件逻辑设计上都相当复杂。例如，对符号相反的两个数求和，加法器不能对它们直接逐位相加。因此，我们需要设计更适合硬件操作的编码方案。最终的答案是*补码*（2's complement）表示法。目前，几乎所有的计算机都采用这种编码方式。

2.3 补码

如图2-1所示，我们可以看到数值−16~+15的补码（2's Complement）表示。选择这种编码，其中的道理是什么呢？

表 示	被表示的数值		
	符号位表示法	反码	补码
00000	0	0	0
00001	1	1	1
00010	2	2	2
00011	3	3	3
00100	4	4	4
00101	5	5	5
00110	6	6	6
00111	7	7	7
01000	8	8	8
01001	9	9	9
01010	10	10	10
01011	11	11	11
01100	12	12	12
01101	13	13	13
01110	14	14	14
01111	15	15	15
10000	−0	−15	−16
10001	−1	−14	−15
10010	−2	−13	−14
10011	−3	−12	−13
10100	−4	−11	−12
10101	−5	−10	−11
10110	−6	−9	−10
10111	−7	−8	−9
11000	−8	−7	−8
11001	−9	−6	−7
11010	−10	−5	−6
11011	−11	−4	−5
11100	−12	−3	−4
11101	−13	−2	−3
11110	−14	−1	−2
11111	−15	−0	−1

图2-1 有符号整数（signed integer）的三种表示方法[⊖]

⊖ 对于符号位表示法（Signed Magnitude），高位取反即为负数；对于反码（1's Complement）所有位取反；对于补码（2's Complement），负数=2^N-1。——译者注

我们看到，正数的编码方案比较直接。以5-bit为例，整个（2^5）编码空间的一半用于代表数值0和1~2^4-1的正整数。

有关负数表示方法的选择，如前所述，应遵循使逻辑电路尽量简单的原则。几乎所有的计算机都采用相同的实现机制完成加法运算，即*算术逻辑运算单元*（Arithmetic and Logic Unit, ALU）。我们将在第3、4章详细介绍ALU的结构，现在我们只需要知道它有两个输入和一个输出，它的功能就是将两个二进制数输入按位相加，并在输出端给出结果。

仍然以5-bit输入模式（pattern）的二进制数为例，假设两个输入分别为00110和00101，则ALU应输出01011，即

$$\begin{array}{r} 00110 \\ 00101 \\ \hline 01011 \end{array}$$

二进制数的加法规则与十进制加法完全相同。从右向左将两数按列对齐并依次相加；如果前列的加法产生进位，则该进位bit参加其左边列的加法操作。

值得一提的是，ALU不知道也不关心它的两个输入数所代表的任何含义。换句话说，它只会对两个二进制数做加法（ADD）而不考虑其他（如正数、负数等因素）。这意味着我们必须为它选择一个合适的编码方式，以保证无论输入数是什么形式，它都将产生正确结果。⊖

下面我们分析正确执行时对编码方式的要求。首先，绝对值相同但符号相反的两个数之和应该为0。换句话说，对于任意两个非0的整数−A和A，在特定的编码方式下，ALU对它们的加法结果应该为00000。

显然，补码表示法满足以上要求，即补码方式就是按照这个条件要求为每个负数分配码字的。例如，整数+5的码字是00101，则−5的码字就是11011。

另外，补码更重要的一个特征是，如果将数值−15到+15的各个码字顺序排开，相邻两个码字之间差值正好为00001。下面我们用数学的方式表示这个现象：

REPRESENTATION(value + 1) =
REPRESENTATION(value) + REPRESENTATION(1).

其中，REPRESENTATION（value）代表数值value所对应的码字。由此我们发现，只要数值不大于15或小于−16，ALU总能保证正确的加法结果。

注意其中数值0和−1的码字，即11111和00000。当我们知道对数值−1做加00001的操作时，我们看到ALU的输出结果确实为00000，但同时还有一个进位。由于进位对结果不产生影响⊜，所以我们说−1在加00001之后的结果是0，而不是100000。在此，进位bit被丢弃了，而事实上在补码运算中，进位bit始终是被忽略的。

提示：在补码表示中，如果已知一个非零整数A的码字，可以很方便地求得其相应负数−A的码字。换算口诀是：取反加1。即将A的码字按位取反，求得A的反码（即位补码），再对A反码加1，即求得−A的补码。我们对照之前的规则验证一下：对于任意A，它与其反码之和总为11111；如果我们将00001与11111相加，结果显然为00000。从而证明了在补码方式下，A与−A之间的码字换算口诀是正确的。

⊖ 原文是“Since the binary ALU only ADDs and does not CARE”，这就如同我们常说的，它只管做事而不问为什么，或者说它只是一个“机制”，而“策略”由上层制定。——译者注

⊜ 所谓“进位（carry）对结果不产生影响”，可以理解为我们读ALU的输出结果时，只会去读那些有效位（如本例中的5-bit有效位），而“最高位”所产生的进位，通常被用做控制（而不是运算）目的。——译者注

例2-1 求数值-13的补码。
(1) 设定A等于+13，则A的补码表示为01101。
(2) A的反码为10010（按位取反）。
(3) 10010加1的结果为10011，即得-13的补码表示。
验证一下结果，将A和-A的码字相加（ALU运算）：

$$\begin{array}{r} 01101 \\ +\ 10011 \\ \hline 00000 \end{array}$$

通过以上验证，我们看到在5-bit ALU情况下，01101和10011的和为00000，同时ALU还将产生一个进位。就是说，事实上01101和10011的求和结果应该是100000。正如之前所述，在补码运算中，该进位bit将被丢弃。

至此，在我们的5-bit系统中，已解决了数值0和15个正数及其对应的15个负数的补码编码分配问题。总共使用了15+15+1=31个码字。我们知道5-bit可以表示2^5=32个码字。那么，剩下一个码字10000怎么分配值呢？

我们注意到-1的码字是11111，-2是11110，-3是11101，依次类推。最后，我们发现，-15的码字是10001，如果将10001继续减1，则得到10000。所以，顺应ALU的操作方式，我们可以让码字10000对应为数值-16。

在第5章中，我们将介绍一个被称为LC-3（LC代表Little Computer）的计算机，LC-3将采用16-bit补码方式，因此LC-3的数值表示范围为从-32 768到+32 767之间的所有整数。

2.4 二进制数与十进制数之间的转换

由于计算机采用二进制来表示数据，而生活中人们普遍采用十进制数，因此二进制数和十进制数之间的转换就显得非常必要。

2.4.1 二进制数转换为十进制数

从二进制到十进制的转换过程如下：以8-bit数据为例，其补码表示对应的数值范围是-128~+127。可以将一个8-bit二进制数表示如下：

$$a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$$

其中，任意位a_i的值为0或1。转换步骤如下：

(1) 符号检查与转换。如果符号位$a_7=0$，表示该值为正数，则继续；而如果$a_7=1$，表示该值为负数，我们需要先将其转换为绝对值相同的正数的补码（稍后将补一个负号）。

(2) 绝对值的计算很简单，求下面的多项式之和即可：

$$a_6\times 2^6+a_5\times 2^5+a_4\times 2^4+a_3\times 2^3+a_2\times 2^2+a_1\times 2^1+a_0\times 2^0$$

(3) 如果该值为负（参见1），我们只要在绝对值的前面加上负号即可。

例2-2 将二进制数11000111转换为十进制表示。
(1) 最高位为1，判断该数为负，因此，我们必须先求出它对应正数的补码，即00111001。
(2) 00111001的绝对值可以表示为：

$$0\times 2^7+1\times 2^6+1\times 2^5+1\times 2^4+0\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0=57$$

(3) 由于是负数，补上负号即得-57。

2.4.2 十进制数转换为二进制数

从十进制数转换为二进制数，要稍微复杂一点。下面的现象反映了其中的窍门，如果一个二进制数是奇数，那么它的最低位必然是1；如果为偶数，则最低位是0。

仍然以8-bit二进制数为例，一个8-bit二进制数可以表示为如下的多项式：

$$a_7\times 2^7 + a_6\times 2^6 + a_5\times 2^5 + a_4\times 2^4 + a_3\times 2^3 + a_2\times 2^2 + a_1\times 2^1 + a_0\times 2^0$$

我们还是以一个具体例子来讲述它到十进制的转换过程。

假定十进制数为+105，我们将其转换为对应的二进制数。由于该数为正数，所以上述多项式中的a_7必然为0。

剩下的任务就是求解满足下列等式的所有a_i值（i = 0~6）：

$$105 = a_6\times 2^6 + a_5\times 2^5 + a_4\times 2^4 + a_3\times 2^3 + a_2\times 2^2 + a_1\times 2^1 + a_0\times 2^0$$

由于105是奇数，因此a_0的取值为1。然后从等式的两端同时减去1，又得：

$$104 = a_6\times 2^6 + a_5\times 2^5 + a_4\times 2^4 + a_3\times 2^3 + a_2\times 2^2 + a_1\times 2^1$$

将上面等式两端同时除以2，则得：

$$52 = a_6\times 2^5 + a_5\times 2^4 + a_4\times 2^3 + a_3\times 2^2 + a_2\times 2^1 + a_1\times 2^0$$

由于52是偶数，因此最低因子a_1必然为0。

如此迭代，每次都将等式两端同时减去最低位的数值（0或1），然后除以2，即可求出所有的a_i值。下面继续：

$$52 = a_6\times 2^5 + a_5\times 2^4 + a_4\times 2^3 + a_3\times 2^2 + a_2\times 2^1$$

处理后得

$$26 = a_6\times 2^4 + a_5\times 2^3 + a_4\times 2^2 + a_3\times 2^1 + a_2\times 2^0$$

因此，a_2为0。

$$13 = a_6\times 2^3 + a_5\times 2^2 + a_4\times 2^1 + a_3\times 2^0$$

因此，a_3为1。

$$6 = a_6\times 2^2 + a_5\times 2^1 + a_4\times 2^0$$

因此，a_4为0。

$$3 = a_6\times 2^1 + a_5\times 2^0$$

因此，a_5为1。

$$1 = a_6\times 2^0$$

因此，a_6为1。最后求得二进制数为01101001。

总结一下这个过程。假定一个十进制数的绝对值为N，则其二进制数求解过程如下：

（1）先将十进制数N展开为如下等式：

$$N = a_6\times 2^6 + a_5\times 2^5 + a_4\times 2^4 + a_3\times 2^3 + a_2\times 2^2 + a_1\times 2^1 + a_0\times 2^0$$

然后反复执行以下操作过程，直到等式左边变成0：

a. 如果N为奇数，则右边最低位a_i为1。如果N为偶数，则右边最低位a_i为0。

b. 然后将等式两端同时减去1（N为奇数）或0（N为偶数），消除最低位。之后，将等式两端同时除以2。

每执行一次，可以求得一个a_i的值。

（2）最后，如果N值为正，则a_7取值为0，完成。

（3）如果N值为负，则最高位补0，然后求该码字的补码，完成。

2.5 bit运算之一：算术运算

2.5.1 加法和减法

二进制补码的算术运算与我们常用的十进制算术运算非常相似。

加法运算仍然是按位对齐，从右向左依次计算。所不同的是，十进制算术运算是满十进一（因为每位最大数是9），而二进制算术运算则是满二进一（每位最大数是1）。

例2-3 计算11 + 3的值是多少？其中二进制数用5-bit表示。

十进制数11表示为	01011
十进制数3表示为	00011
则两者的和为14	01110

减法可以理解为就是加法，如$A-B$等价于$A+(-B)$。

例2-4 计算14−9的值是多少？其中二进制数用5-bit表示。

十进制数14表示为	01110
十进制数9表示为	01001
转换获得−9的补码	10111
再将−9与14相加	01110
	10111
最后计算结果是5	00101

例2-5 如果将一个数与它自己相加会如何呢？例如，$(x+x)$。

以8-bit编码为例，即数值范围在−128和127之间。假定x等于59，即00111011，如果将59与自己相加，结果为01110110。很奇怪！我们发现所有的位都向左移动了一位。这是偶然现象，还是必然规律呢？

采用多项式表示，则59等于

$$0\times2^6+1\times2^5+1\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0$$

（59+59）之和等价于59×2，即

$$2\times(0\times2^6+1\times2^5+1\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0)$$

也就是

$$0\times2^7+1\times2^6+1\times2^5+1\times2^4+0\times2^3+1\times2^2+1\times2^1$$

我们发现，事实上第一个式子中每个数字左移一位，就是第三个式子（只要总位数足够，不要移出界了）。于是我们得出结论，一个数自己加自己就等于自己左移一位。

2.5.2 符号扩展

在一些情况下，为了减少占用空间，较小的数值会采用较少的bit来表示或存放。例如，数值5与其表示为0000000000000101，还不如用6个bit表示为000101。这个表述可能让你有些困惑了。事实上，我们是想说明这样一个事实，通常在一个数前面补上0，并不会改变它的值。例如，支票上的金额写为$456.78与写为$0000456.78之间没有任何区别。

那么，对于负数呢？我们以数值−5为例，如果我们用6-bit来表示，数值5等于000101，−5则为111011。如果我们用16-bit来表示，数值5为0000000000000101，−5则为1111111111111011。我

们发现，如同在正数前面加0一样，在负数前面加1也不改变其值。

下面，我们看两数相加的操作。我们发现，如果将两个表示长度不等的二进制数直接相加，将存在一些问题。例如，计算13和−5两数的和。其中，将13表示为0000000000001101，而−5表示为111011。那么ALU计算出来的结果是什么呢？

$$\begin{array}{r} 0000000000001101 \\ + \quad\quad 111011 \end{array}$$

问题出现了，我们应该如何处理111011中缺少的位呢？如果我们把那些缺省位填为0，那么与+13相加的数将不再是−5，为什么？因为0000000000111011代表的是+59。这样，计算出来的结果将是+72，显然是错误的。

然而，如果我们意识到数值−5的6-bit表示与16-bit表示之间的差异（即高位那些看来无意义的1），并将其6-bit码字做符号扩展，转换为16-bit，则计算式如下所示：

$$\begin{array}{r} 0000000000001101 \\ + 1111111111111011 \\ \hline 0000000000001000 \end{array}$$

于是，结果正确，即+8。

通过以上例子，我们得出结论：在二进制正数前面添加任意多的0不会改变其值；同样，在负数前面添加任意多的1（即符号扩展）也不会影响其值。我们称这两种操作为“符号扩展”（Sign-Extension），简称SEXT。符号扩展主要应用在两个不同长度的二进制数相加的场合。

2.5.3 溢出

到目前为止，所遇到的运算结果都还未遇到超出码字的最大值或最小值的情况，如果超过，将怎么处理呢？

你一定熟悉汽车仪表盘上的里程表，它的作用是记录汽车已行驶的英里数。在老式的汽车上，如果里程表的当前值为99992，则汽车又行驶100英里后，读数将变为00092。这是因为里程表能够记录的最大值为99999，从而将实际的100092显示为00092，即最高位的进位被丢失了。

这时我们称里程表溢出了。随着技术的进步，现在的汽车大都能行驶到1 000 000英里以上，于是汽车制造商觉得有必要在里程表上再多加一位。也就是说，现在的汽车要行驶到1 000 000英里后才会溢出。

汽车里程表是一个典型的无符号整数运算器，每单位行程之后，它都将累加一个正数。例如，如果当前里程表读数是000129，则行驶50英里之后，其读数将变成000179，只有当最高位进位时，才出现溢出。

相比之下，有符号整数运算的溢出问题就要复杂一些了。仍然以5-bit二进制补码为例，我们已知5-bit码字的表达范围是−16~+15。如果计算(+9)+(+11)的运算结果，+9的二进制形式为01001，+11的二进制形式为01011，则

$$\begin{array}{r} 01001 \\ + 01011 \\ \hline 10100 \end{array}$$

结果是一个负数（最高位或符号位为1），显然出错了。这是因为（+9）+（+11）的结果超出了5-bit补码所能表示的最大正整数0111（或+15），而我们的运算结果超过了这个最大值，导致进位冲入了符号位的bit。结果过大，导致溢出问题，其检测机制很容易实现。因为两个正数相加，

结果必然也是正数，而如果ALU给出的结果是一个负数，显然出错了。在计算机中，我们称这种故障现象为“溢出”。

再看负数的运算。例如，表达式（−12）+（−6）。−12的补码为是10100，−6的补码是11010，即：

$$\begin{array}{r} 10100 \\ +\ 11010 \\ \hline 01110 \end{array}$$

也出现错误了！这是因为（−12）+（−6）的结果小于5-bit补码所能表达的最小值（−16），同样也将产生最高位进位，进而导致符号位被“破坏”。两个负数的和必然也是负数，而溢出后的值却是一个正数。同样，通过运算结果和两个运算数之间的符号位比较，很容易发现溢出问题。

注意，前面的两个溢出例子都发生在同符号数相加的情况下。事实上，也只有在这两种同符号运算情况下，才会发生溢出。一个正数和一个负数相加，永远都不会溢出，参见习题2.25。

2.6 bit运算之二：逻辑运算

前面介绍的是二进制算术运算，如加法、减法。本节将介绍二进制数的另一类重要运算，即*逻辑运算*（Logical Operation）。

逻辑运算符的操作对象是逻辑变量。逻辑变量的取值只有两个：0或1。“*逻辑*”（logical）这个名词的命名是个历史产物；在此，我们只是使用0和1这两个数来代表逻辑值中的“假”（flase）和“真”（true），而逻辑运算与“逻辑”原有的含义已相去甚远。

逻辑运算由几个基本运算组成，大多数ALU都具备这些功能。

2.6.1 “与”运算

“与⊖”（AND）是一个二元逻辑运算，这意味着它需要两个源操作数，且每个操作数的取值要么为0、要么为1。如果两个源操作数同为1，则AND的输出结果为1；否则，结果皆为0。

逻辑操作的功能（或行为）定义的常用描述方式是“*真值表*”（truth table）。真值表的行数和列数分别为2^n和$n+1$。前n列分别代表n个源操作数，由于每个源操作数都是逻辑变量，变量值只可能为0或1，所以n个源操作数的所有可能组合数目为$2n$。真值表中的每一行代表了一种组合，表的最后一列代表该组合方式下的逻辑运算结果。

以两输入AND单元为例，真值表中有两列，分别对应两个源操作数，而行数为4（2^2），对应不同的逻辑组合。

A	*B*	AND
0	0	0
0	1	0
1	0	0
1	1	1

我们也可以对两个长度为m-bit的二进制数做AND运算，即将两个操作数按位对齐，然后对其中的每一对bit进行AND运算。在下面的例子中，a和b都是16-bit二进制数，c是a和b之间相与的结果。我们将这种操作称做*按位*AND操作。

⊖ AND的中文说法是“与”，如“与操作”、“与门”等。但为了避免在字面上出现混淆（如误读做连接词“与”），所以在后面的文字中，有时候直接使用AND符号表示“与”操作或“与”逻辑单元。——译者注

例2-6 如果c是a和b相"与"（AND）的结果，其中a=0011101001101001，b=0101100100100001，那么c的值是多少？

a和b之间的AND运算，实际上是两个数值对应位之间的按位与运算。换句话说，c_i是每对a_i和b_i相AND的结果。例如，由于a_0=1、b_0=1，则c_0等于它们相与的结果，即$c_0 = (a_0 \text{ AND } b_0) = 1$。因而，$c$完整的求解结果如下：

$$\begin{array}{ll} a & 0011101001101001 \\ b & 0101100100100001 \\ c & 0001100000100001 \end{array}$$

例2-7 假设有一个8-bit数A，其最低两位有特殊含义。如果计算机根据A的最低两位值做4种不同的操作，请问如何单独提取这两位的值？

答案是"屏蔽字位"（bit mask）操作。"屏蔽字位"也是一个二进制数，我们可以将其理解为由两部分组成——即我们关心的位和我们忽略的位这两部分。如本例中，屏蔽字位为00000011，它与A相与（AND）产生的结果是：从7到2的最高6位必定为0，而结果的低2位与A原有的低两位完全相同。因此，屏蔽字位能够"屏蔽"掉从7到2的那些位。

例如，A = 01010110，则A与屏蔽字位00000011相与（AND）的结果为00000010，再如A = 11111100，则A和屏蔽字位00000011相与（AND）的结果为00000000。

换句话说，无论A为什么，它与屏蔽字00000011相与（AND）的结果必然是00000000、00000001、00000010或00000011等4个值中的一个。屏蔽字的作用，就是将我们感兴趣的那些位提取出来。

2.6.2 "或"运算

"或"（OR）也是一个二元逻辑运算，它的两个源操作数都是逻辑变量，取值为0或1。仅当两个源操作数都为0时，OR的输出结果才为0，其他情况下，OR输出都为1。换句话说，两个源操作数中任意一个为1，则OR结果必然为1。

OR运算的真值表如下所示：

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

与AND运算类似，我们也可以在两个m-bit的二进制数之间执行按位OR运算。

例2-8 如果c是a和b相"或"（OR）的结果，其中a=0011101001101001，b=0101100100100001，那么c的值是多少？

a和b之间的OR运算，实际上是两个数值对应位之间的按位OR运算。换句话说，c_i是每对a_i和b_i相OR的结果。例如，其中a_0=1、b_0=1，则c_0等于它们之间相OR的结果，即$c_0 = (a_0 \text{ OR } b_0) = 1$。再如，$a_6$=1、$b_6$=0，则$c_6 = (a_6 \text{ OR } b_6) = 1$。$c$完整的求解结果如下：

$$\begin{array}{ll} a & 0011101001101001 \\ b & 0101100100100001 \\ \hline c & 0111101101101001 \end{array}$$

2.6.3 "非"运算

"非"（NOT）运算是一元逻辑运算，这意味着它只需要一个源操作数。该运算又被称为"补"（complement）运算，即输出是输入求补的结果。有时候我们又称该输出是输入的"反（inverting）操作"。如果输入A为1，则输出为0；如果输入A为0，则输出为1。

NOT运算的真值表如下所示：

A	NOT
0	1
1	0

同样，NOT运算也可以像AND和OR运算一样，对m-bit的数进行按位（bit-wise）NOT操作。假设a的取值如前面的例子所示，则c每位的值是a对应位取反的结果，如下所示：

$$\begin{array}{ll} a & 0011101001101001 \\ c & 1100010110010110 \end{array}$$

2.6.4 "异或"运算

"异或"（exclusive-OR），简称XOR，也是一个二元逻辑运算。同样，它也需要两个源操作数，且每个逻辑变量的取值或为0，或为1。如果两个输入值不同，则XOR输出为1；如果两者相同，则XOR输出为0。XOR的真值表如下所示：

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

我们采用和AND运算相同的方法，对两个m-bit二进制数进行按位XOR运算。

例2-9 如果c是a和b的"异或"（XOR）的结果，其中a=0011101001101001，b=0101100100100001，那么c的值是多少？

$$\begin{array}{ll} a & 0011101001101001 \\ b & 0101100100100001 \\ c & 0110001101001000 \end{array}$$

请注意此处XOR的真值表和之前OR真值表之间的差异。在XOR中，如果两个输入都是1，则输出为0（而不是如OR中那样为1）。换句话说，XOR的输出若为1，则如果第一个输入为1，则第二个必须为0，或是第一个输入为0，则第二个必须为1，两者相互制约。之所以称其为"Exclusive"（排他的）OR，是因为两个输入之中有且仅有一个为1，则输出为1。相比较，OR中只要两个输入中有一个为1，则输出为1，我们称这种情况为"Inclusive-OR"（包容的）。

例2-10 如果要判断两个二进制数是否相同，可以通过XOR操作来完成。如果两个数完全相同，则它们之间XOR的输出结果必为0。

2.7 其他类型

除整数类型外，通常我们还会遇到如位矢量（bit vector）、浮点数（float-point）、ASCII码和十六进制（HEX）等数据类型。本节将对它们进行介绍。

2.7.1 位矢量

在很多复杂系统中，包含了许多独立单元，每个单元或忙或空闲。譬如，一个制造车间里的每一台机器，又如出租车网络中的每辆出租车，它们都是系统中独立的组成单元。重要的是，我们必须从各个单元中判断并选取那些空闲的单元，以便我们为其分配任务。

假设存在n个单元，我们可以用一个n-bit的二进制数代表这n个单元。当某个单元空闲时，我们将相应的bit清0；当某个单元忙碌时，我们将相应的单元置1。我们称这个二进制数为“位矢量”(bit vector)。

例2-11 假设要监视8台机器的运行状态，我们可以用一个8-bit矢量BUSYNESS来代表它们，其中每个bit代表一台机器，如果该机器忙，则对应位为0，若空闲则为1。每个bit的编号从右向左，分别是0~7。

位矢量BUSYNESS的含义是，如果BUSYNESS = 11000010，则表示第1、6、7号机器处于空闲状态，从而可以为它们分配任务。

假设我们将任务分配给了第7号机器，则将通过AND运算更新BUSYNESS矢量的bit 7。AND运算的两个输入中，一个是当前BUSYNESS矢量，另一个是屏蔽字01111111。该屏蔽字的目的就是在不影响其他位的情况下，清除BUSYNESS矢量的第7位。操作后的结果是，BUSYNESS矢量从11000010改变为01000010。

回顾一下我们在例2-7中提到的“屏蔽字”概念。它使得我们只与其中的部分bit打交道，而忽略其他的bit。在本例中，它的作用就是只将第7位清除，而不改动其他位。

假设当前第5台机器已完成任务，我们也要更新BUSYNESS矢量的值，即将矢量中的第5位置1，以表示该机器空闲，可以分配新任务了。在此，我们将通过OR运算来解决这个问题。让BUSYNESS矢量与屏蔽字00100000相OR，则得到结果01100010。

2.7.2 浮点数

本书中大多数的例子都是基于整数的运算。在LC-3中，整数的表示方法是16-bit补码，其中15位表示它的数值（绝对值），最高的1位则用来代表该整数的符号。因此，LC-3能表示的数值范围是整数-2^{15}~$2^{15}-1$（即−32 768~32 767），我们称LC-3的数值表示*精度*（precision）是15位，*范围*（range）是2^{15}。但在很多情况下，我们要表示更大的数值，如6.023×10^{23}（摩尔常数[⊖]），显然这个数超过了16-bit二进制数的表达范围。其实，更重要的是能够表达6 023这个4位十进制数。

问题是，我们需要更多的bit才能保证对这个数的表示精度。

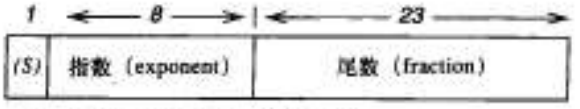
*浮点数*的引入解决了这个问题。在浮点表示法中，除符号位之外，并不是将所有bit都用于精度表示，而是一部分bit用于表达数值范围（多大、多小），另一部分用于表示数值精度。

大多数的指令集都定义了一种或多种浮点数类型。其中之一通常被称做“float”类型，由32-bit组成，各bit的定义如下：

符号：1bit，代表符号（正数或负数）
数值范围：8bit，代表范围（指数，exponent）
数值精度：23bit，代表精度（尾数部分，fraction）

⊖ 摩尔（mole）是一个数量单位，如“1 mole of $H_2O = 6.023 \times 10^{23}$ molecules of H_2O”（参考wei）。该常数又称为Avogadro常数，意大利科学家Amedeo Avogadro（1778-1850）于1811年提出：“在相同温度、压力下，相同体积的不同气体包含相同数目的气体分子，在标准状态下，这个数目就是6.023×10^{23}”。由于分子的英文是molecule，该单位由此而得名。——译者注

几乎所有的计算机制造商都使用如图2-2所示的float格式，该格式也是IEEE浮点运算标准的一部分。



$N=(-1)^s\times 1.\text{尾数}\times 2^{\text{指数}-127}$，$1\leqslant\text{指数}\leqslant 254$

图2-2 浮点数类型

这种浮点数表示法和我们中学课本中的科学计数法（scientific notation）非常相似，如6.023×10^{23}就是科学计数法的例子。在科学计数法中，有三个基本组成：符号（默认为+），有效数字（6.023），指数（23），我们称其中的有效数字为尾数（fraction）。注意，在浮点表示法中，fraction将被正则化（normalized），即小数点左边有且仅有一个非零数（十进制）。

浮点数（如图2-2所示）也由相同的三部分组成。与科学计数法中的4个十进制尾数相比较，浮点数的尾数长度是23个二进制数字。注意，尾数部分是被正则化的，即小数点左边有且仅有一位非0数字，但二进制方式下，这个非0数字只可能是1，所以这个1就不必表示出来了。因而，只要23bit就能表示24位的精度。换句话说，在IEEE标准中，那个必然存在的1被省略了。

关于指数部分，与6.023×10^{23}中的两位十进制指数不同的是，IEEE浮点数使用了23位（二进制）表示指数。两者的不同还表现为：科学计数法采用的基数（radix）为10，而IEEE浮点数的基数为2，8-bit的指数字段可以表示256个不同的指数值。但图2-2却指出指数值范围是1~254，那么剩余两个指数值00000000和11111111代表什么意思呢？图2-2中对此未给出任何解释，我们将在本节最后给出答案。

对浮点数中指数字段的254个取值，解释如下：实际的指数值等于该无符号整数减127之后的结果。例如，如果实际指数值是+8，那么指数部分则表示为10000111（即无符号数135），因为135−127 = 8。如果实际指数值是−125，那么指数部分则表示为00000010（即无符号数2），因为2−127 = −125。

最后一个字段是符号位：0代表正数，1代表负数。公式中，系数$(-1)^s$意味着如果$s=0$，则符号为1，若$s=1$，则为−1。

例2-12 试用IEEE浮点数标准表示$-6\frac{5}{8}$。

首先将$-6\frac{5}{8}$表示成二进制数−110.101，即

$$-6\frac{5}{8}=-(1\times 2^2+1\times 2^1+0\times 2^0+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3})$$

正则化处理之后，即为-1.10101×2^2。

符号位为1，表示$-6\frac{5}{8}$是一个负数。指数部分为10000001，即无符号数129，代表实际指数为+2（129−127 = +2）。尾数部分省略小数点左边的1之后，精度为23位。因而，尾数部分等于10101000000000000000000。由此，$-6\frac{5}{8}$的IEEE浮点数表示为：

1 10000001 10101000000000000000000

例2-13 请问IEEE浮点数00111101100000000000000000000000代表的是什么数？

最左边位为0，表示这是一个正数。随后的8位是无符号数123，将其减去127之后，得实际指数值-4。最后的23位全部为0，所以它代表数值

$$+1.00000000000000000000000\times 2^{-4}$$

即$\frac{1}{16}$。

我们发现，在这32位的解释过程中，指数字段不能是00000000或11111111。在IEEE浮点数标准中，定义了00000000和11111111的含义。

如果指数字段内容是00000000，则代表指数值为-126，且尾数中小数点左边默认数字是0（而不是1）。所以，该浮点数的值应该是：

$$(-1)^{s}\times 0.\text{尾数}\times 2^{-126}$$

再如，IEEE浮点数

0 00000000 00001000000000000000000

的解读过程如下：最高位0代表该数是正数。指数字段为00000000，表示指数值为-126。右边23位事实上代表数值0.00001000000000000000000，即等于2^{-5}。因而，该浮点数代表的数值是$2^{-5}\times 2^{-126}$，即2^{-131}。

通过这种方式，我们发现IEEE浮点数可以表示非常小的数。

例2-14 下面4个例子将进一步解释IEEE标准32-bit浮点数的解读过程。

0 10000011 00101000000000000000000

指数字段的内容是131。由于131-127等于4，所以指数值为+4。在小数点左边补1，并将尾数字段放在小数点右边，则得1.00101。如果将小数点右移+4位，则得10010.1，即18.5。

1 10000010 00101000000000000000000

符号位为1表示负数。指数字段为130，表示指数值等于130-127或+3。在尾数字段前小数点左边添加1，得1.00101。将小数点右移+3位，则得1001.01。所以结果为-9.25。

0 11111110 11111111111111111111111

符号为+。指数值等于254-127或+127。在尾数字段及其小数点左边补1，得1.1111111…1，即大约为2。因此，结果约等于2^{128}。

1 00000000 00000000000000000000001

符号为-。指数字段全为0，表示指数值为-126。注意，此时小数点左边补0（因为指数字段全为0），尾数值为2^{-23}。因而该浮点数等于$-2^{-23}\times 2^{-126}$，即-2^{-149}。

有关IEEE浮点数标准的深入理解，已超出本书要求。事实上，我们连指数字段为11111111的32-bit浮点数的含义是什么都没有讲，也不准备讲了。我们讲述浮点数的意图，只是想让你知道，除了补码整数外，还存在一种非常重要的（几乎所有的ISA都具备的）数据类型。这种数据类型就是浮点数，它既可以表达非常大的数值，也可以表达非常小的数值，只是牺牲一些数值精度而已。

2.7.3 ASCII码

另外一类信息表示方法也是大多数计算机设备厂家都遵循的，它是一种用于在计算机处理单元和输入/输出设备之间传递“字符”（character）的编码标准。这种8-bit编码方式被称做ASCII码。ASCII代表美国信息互换标准码（American Standard Code for Information Interchange），它的存在为计算机厂家、设备厂家（生产键盘、显示器等设备）之间提供了一个定义字符格式的共同标准。

每次在键盘上敲键，都将产生惟一的ASCII码。比如键盘上的数字3，它的ASCII码是

00110011，数字2的ASCII码是00110010，小写*e*是01100101，回车符是00001101。可参考附录E，在图E-3中列出了完整的ASCII码表。当用户在键盘上按下一个键时，键盘传输给计算机的是其ASCII码值。至于传输这个ASCII码所涉及的存储地址和读入机制，将在第8章予以介绍。

键盘上的大多数按键，都对应不止一个ASCII码，如大写*E*的ASCII码是01000101，而小写*e*则是01100101，但这两个字母在键盘上却是同一个按键。键盘通过功能键Shift是否同时按下来，区别两种情况。如果Shift键同时按下，则该键表示大写E；否则代表小写e。

另外，在显示器上输出的字符，也是由计算机通过ASCII编码方式传输给显示器的。该操作机制也将在第8章中介绍。

2.7.4 十六进制计数法

我们已介绍过，信息可以表示为二进制补码、位矢量、浮点数、ASCII码等方式。当然，还存在其他表达方式，我们将它们留给别的教材来介绍（本书将不再深入）。但是，在本书中，还将介绍一种数据类型，它不仅适合计算机操作，更重要的是，它是一种方便人们阅读的表达方式，这就是*十六进制计数法*。它起源于二进制表示法，但更方便表示长串的二进制数。

在LC-3中，我们将使用它来表示单位长度为16位的二进制数。

假设有这样一个16-bit二进制数据：

0011110101101110

我们做个试验，先浏览一遍，然后用手将这些数字遮住。试问，你能将这些数字默写出来么？而我们如果采用十六进制数来表示，则容易得多。

一个16位二进制数可以表示为如下形式：

$$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$$

其中任意a_i的取值为0或1。

如果将该16-bit二进制数看做无符号整数，则可通过下面的公式计算它的值：

$$a_{15}\times 2^{15}+a_{14}\times 2^{14}+a_{13}\times 2^{13}+a_{12}\times 2^{12}+a_{11}\times 2^{11}+a_{10}\times 2^{10}+a_9\times 2^9+a_8\times 2^8$$
$$+a_7\times 2^7+a_6\times 2^6+a_5\times 2^5+a_4\times 2^4+a_3\times 2^3+a_2\times 2^2+a_1\times 2^1+a_0\times 2^0$$

如果将此式简化为：前4项提取公因子2^{12}，第5~8项提取公因子2^8，第9~12项提取公因子2^4，最后4项结合，提取公因子2^0，则得：

$$2^{12}[a_{15}\times 2^3+a_{14}\times 2^2+a_{13}\times 2^1+a_{12}\times 2^0]$$
$$+2^8[a_{11}\times 2^3+a_{10}\times 2^2+a_9\times 2^1+a_8\times 2^0]$$
$$+2^4[a_7\times 2^3+a_6\times 2^2+a_5\times 2^1+a_4\times 2^0]$$
$$+2^0[a_3\times 2^3+a_2\times 2^2+a_1\times 2^1+a_0\times 2^0]$$

注意，每个方括号内的最大值都是15。如果我们用一个符号来代表方括号内的值（从0到15），并将2^{12}替换为16^3，2^8替换为16^2，2^4替换为16^1，2^0替换为16^0，则得：

$$h_3\times 16^3+h_2\times 16^2+h_1\times 16^1+h_0\times 16^0$$

其中h_i表示相应方括号内的值，如符号h_3代表$[a_{15}\times 2^3+a_{14}\times 2^2+a_{13}\times 2^1+a_{12}\times 2^0]$。

由于任意h_i的取值范围是0~15，所以我们用符号0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F等16个字符来代表符号h_i的16个可能取值。即0代表0000、1代表、……、9代表1001、A代表1010、……、F代表1111，我们称这种表示法为十六进制数（hexadecimal），或基（base）为16的表示方法。

试问，如果十六进制数E92F代表的是一个16-bit二进制数，那么该数是正数还是负数？为什么？

现在反问一下，十六进制数表示法有什么好处呢？看起来它仅仅在表示方法上做了些变化，并未带来什么好处啊？让我们回到刚才的那个默写试验。如果将二进制数

0011110101101110

改写为十六进制，即将它分成4段

0011 1101 0110 1110

每段（4个bit）对应的十六进制数是

3 D 6 E

我们会发现，很容易记住3D6E。

所以，十六进制计数法（用来替代二进制数字串）的主要好处是方便记忆和使用。通过它，我们可以表示二进制数、浮点数、ASCII字符序列或位矢量等各种数据类型。同时，这种记忆上的简易性，避免了因二进制数字串过长而引起的誊写错误。

2.8 习题

2.1 一个n-bit数可以表示多少个不同的二进制数？

2.2 如果采用二进制bit串来表示英语中的26个字母，至少需要多少个bit？如果还要区分大小写字母，又需要多少bit？

2.3 a. 假设某班级有400个学生，如果我们为每个学生分配一个惟一的二进制bit串，那么至少需要多少个bit来表示所有的学生？

b. 如果不再增加bit数，这个班最多还能增加多少个学生？

2.4 给定n位（bit），它可以表示多少个不同的无符号整数？范围是多大？

2.5 如果用5-bit二进制串表示数值，写出数值7和−7分别对应的反码、符号位码和补码表示方式。

2.6 试用6-bit反码表示数值−32。

2.7 试列出4-bit二进制补码所能表示的所有整数。

2.8 a. 8-bit二进制补码能表示的最大正整数是多少？分别写出十进制和二进制数。

b. 8-bit二进制补码能表示的具有最大绝对值的负数是多少？分别写出十进制和二进制数。

c. n-bit二进制补码所能表示的最大正整数是多少？

d. n-bit二进制补码所能表示的具有最大绝对值的负数是多少？

2.9 如果用二进制补码方式表示摩尔常数6.0×10^{23}，需要多少个bit？

2.10 将下面的二进制补码转换为十进制数：

a. 1010　　b. 01011010

c. 11111110　　d. 0011100111010011

2.11 将下面的十进制数转换为8-bit二进制补码：

a. 102　　b. 64　　c. 33

d. −128　　e. 127

2.12 二进制补码最后一位如果是0，那么这个数必然是偶数。如果二进制补码的最后两位都为0（例如二进制数01100），那么这种数有什么特点？

2.13 请将下面的二进制数改写为8-bit数，且不允许改变其原有数值。

a. 1010　　b. 011001

c. 1111111000　　d. 01

2.14 计算下列二进制加法，结果仍然表示为二进制数。

a. 1011 + 0001　　b. 0000 + 1010　　c. 1100 + 0011

d. 0101 + 0110　　e. 1111 + 0001

2.15 在本章例2-5中，证明了一个二进制数左移1位等价于将该数值乘2。试问，如果将其右移1位，等价于什么操作呢？

2.16 计算以下8-bit加法的结果，并分别写出结果所对应的二进制和十进制数。

a. 7的反码加上−7的反码；　b. 7的符号位表示加上−7的符号位表示；

c. 7的补码加上−7的补码。

2.17 计算以下二进制补码加法，并将结果转换为十进制形式。

a. 01 + 1011　　b. 11 + 01010101

c. 0101 + 110　　d. 01 + 10

2.18 计算以下无符号二进制数的加法，并将结果转换为十进制形式。

a. 01 + 1011　　b. 11 + 01010101

c. 0101 + 110　　d. 01 + 10

2.19 将十进制数−27分别转换为8-bit补码、16-bit补码和32-bit补码，并阐述符号扩展在这三种表达形式中的应用。

2.20 在以下4-bit补码运算中，其中哪些计算会产生溢出？请将操作数和计算结果分别改写为十进制方式予以验证。

a. 1100 + 0011　　b. 1100 + 0100　　c. 0111 + 0001

d. 1000 - 0001　　e. 0111 + 1001

2.21 试描述在什么情况下，两个补码相加将产生溢出。

2.22 试给出两个16-bit补码相加产生溢出的例子。

2.23 试描述在什么情况下，两个无符号整数相加会产生溢出。

2.24 试给出两个16-bit无符号数相加产生溢出的例子。

2.25 试解释，为什么在补码方式下负数和正数相加不会产生溢出？

2.26 将数值−64表示为二进制补码方式，并回答以下问题：

a. 最少需要多少bit？

b. 这些bit能够表示的最大正整数是多少（分别给出二进制和十进制数）？

c. 这些bit能够表示的最大无符号整数是多少（分别给出二进制和十进制数）？

2.27 LC-3是一个16-bit计算机，如果LC-3对补码0101010101010101和0011100111001111相加的结果为1000111100100100，请问该结果有问题吗？如果有，问题是什么？如果没有，为什么？

2.28 试阐述逻辑AND运算在什么条件下输出为1。

2.29 填写下面逻辑AND运算的真值表。

X	*Y*	*X* AND *Y*	*X*	*Y*	*X* AND *Y*
0	0		1	0	
0	1		1	1	

2.30 求解下列逻辑运算结果，给出二进制表示。

a. 01010111 AND 11010111　　b. 101 AND 110

c. 11100000 AND 10110100　　d. 00011111 AND 10110100

e. (0011 AND 0110) AND 1101　　f. 0011 AND (0110 AND 1101)

2.31 试阐述逻辑OR运算在什么条件下输出为1。

2.32 填写下面的逻辑OR运算的真值表。

X	Y	X OR Y	X	Y	X OR Y
0	0		1	0	
0	1		1	1	

2.33 求解下列逻辑运算结果：

a. 01010111 OR 11010111　　b. 101 OR 110

c. 11100000 OR 10110100　　d. 00011111 OR 10110100

e. (0101 OR 1100) OR 1101　　f. 0101 OR (1100 OR 1101)

2.34 求解下列逻辑运算结果：

a. NOT(1011) OR NOT(1100)　　b. NOT(1000 AND (1100 OR 0101))

c. NOT(NOT(1101))　　d. (0110 OR 0000) AND 1111

2.35 试阐述，在本章例2-11中，屏蔽字的作用是什么？

2.36 参考本章例2-11，回答以下问题：

a. 如果要标识2号机器为“忙碌”，屏蔽字应该是多少？执行什么逻辑操作？

b. 如果要同时标识2、6号两台机器为“空闲”，屏蔽字应该是多少？执行什么逻辑操作（提示：只要一次操作即可）？

c. 如果要标识所有机器都处于“忙碌”状态，屏蔽字是多少？执行什么操作？

d. 如果要表示所有机器都处于“空闲”状态，屏蔽字是多少？执行什么操作？

e. 编写一个程序，专门操作第2位。如果第2位为0，则程序输出00000000；如果第2位为1，则程序输出1000000。换句话说，它等价于这样一种情况，假设BUSYNESS的值为

b7	b6	b5	b4	b3	b2	b1	b0

那么程序输出则为

b2	0	0	0	0	0	0	0

（提示：一个二进制数和本身相加会产生什么情况?）

2.37 假设n、m、s都是4-bit补码数，且s是n和m相加的结果。如果只允许使用2.6节中介绍的逻辑运算，如何判断n和m相加的结果是否溢出？编写一个程序，输入n、m和s的值，如果n和m相加溢出，则输出1000；如果没有溢出，则输出0000。

2.38 如果n、m、s都是4-bit无符号整数，s是n与m相加的结果。如果只允许使用2.6节中介绍的逻辑运算，如何判断n和m相加的结果是否溢出？编写一个程序，输入n、m和s的值，如果n和m相加溢出，则输出1000；如果没有溢出，则输出0000。

2.39 将以下十进制数转换成IEEE浮点数方式。

a. 3.75　　b. $-55\frac{23}{64}$

c. 3.1415927　　d. 64 000

2.40 将以下IEEE浮点数转换为十进制数。

a. 0 10000000 00000000000000000000000

b. 1 10000011 00010000000000000000000

c. 0 11111111 00000000000000000000000

d. 1 10000000 10010000000000000000000

2.41 a. 32-bit IEEE浮点数能表示的最大数值是多少？

b. 32-bit IEEE浮点数能表示的最小数值是多少?

2.42 一个程序员编写了一个两数求和的程序。但是运行后发现，5和8相加的结果是字符“*m*”，请分析造成这种奇怪现象的可能原因。

2.43 将以下十六进制ASCII码改写为8-bit二进制ASCII码。

a. x48656c6c6f21　　b. x68454c4c4f21

c. x436f6d70757465727321　　d. x4c432d32

2.44 如果要将数字3的二进制表示转换成字符“3”的ASCII码，请问如何操作？那么，数字4到字符“4”的转换呢？试问，是否任意数字都可以通过这种操作来转换?

2.45 将以下无符号数从二进制方式转换为十六进制方式。

a. 1101 0001 1010 1111　　b. 001 1111

c. 1　　d. 1110 1101 1011 0010

2.46 将以下十六进制数转换为二进制形式。

a. x10　　b. x801

c. xF731　　d. x0F1E2D

e. xBCAD

2.47 将以下十六进制补码转换为十进制数。

a. xF0　　b. x7FF

c. x16　　d. x8000

2.48 将以下十进制数转换为二进制补码及其十六进制表示。

a. 256　　b. 111

c. 123 456 789　　d. −44

2.49 下面是十六进制表示的补码之间的运算。请问，能否不将它们展开为二进制方式，而直接计算各式的值，结果仍然采用十六进制表示。

a. x025B + x26DE　　b. x7D96 + xF0A0

c. xA397 + xA35D　　d. x7D96 + x7412

e. 对c和d的计算结果有什么补充吗?

2.50 下面是十六进制补码的逻辑运算。请问，能否不将它们展开为二进制方式，而直接计算各式的值，结果仍然采用十六进制表示。

a. x5478 AND Xfdea　　b. xABCD AND x1234

c. NOT(NOT(xDEFA)) AND (NOT(xFFFF))　　d. x00FF XOR x325C

2.51 试写出以下数值的十六进制形式。

a. 25 675　　b. 675.625，IEEE 754浮点数标准形式

c. ASCII字符串：Hello

2.52 假设十六进制数为x434F4D50和x55544552。请填写下表，将这两个数用不同的方式表示。

	x434F4D50	x55544552
无符号二进制整数形式		
二进制反码形式		
二进制补码形式		
IEEE 754浮点数形式		
ASCII 字符串		

2.53 填写以下算式的真值表，并用OR和NOT逻辑表示Q_2。

Q_1 = NOT(A AND B)

Q_2 = NOT(NOT(A) AND NOT(B))

A	B	Q_1	Q_2
0	0	1	0

2.54 填写下面式子的真值表。

Q_1 = NOT(NOT(X) OR (X AND Y AND Z))

Q_2 = NOT((Y OR Z) AND (X AND Y AND Z))

X	Y	Z	Q_1	Q_2
0	0	0	0	1

2.55 我们已学过二进制（base-2）和十六进制（base-16）的数字表示方法。如果是无符号base-4方式，我们称之为四进制（quad）。在四进制中，每个数字的表示可以是0、1、2或3。试问：

a. 3位四进制数能表达的最大无符号数值是多少（请用十进制回答）？

b. n位四进制数能表达的最大无符号数值是多少（提示：答案必须以n为参数）？

c. 计算无符号四进制数023和221相加的结果。

d. 求出十进制数42的四进制表示。

e. 求出将四进制数123.3转换为二进制方式的表示。

f. 将四进制数123.3转换为IEEE浮点数表示。

g. 假设一个黑箱函数，该黑箱的输入是一个m位四进制数，输出是一位四进制数字。试问该黑箱函数有多少种可能的实现（提示：排列组合，对于任意一个特定输入，有4种可能输出）？

2.56 试定义一种新的8-bit浮点数格式，符号位1-bit、指数字段4-bit（校正值为7，即bias=7）、尾数3-bit。试问，十六进制数xE5转换为该8-bit浮点数的结果是什么？最后数值（十进制）是多少？

第3章 数字逻辑

在第1章中，我们曾提到计算机是由非常多的简单逻辑单元组合而成的。如Intel公司2000年推出的奔腾IV处理器，就包含了4200万个MOS晶体管。同样，IBM公司2002年推出的PowerPC 750 FX包含了3800多万个MOS晶体管。在本章里，我们将介绍逻辑单元的基本要素——MOS晶体管的工作原理，以及如何基于MOS晶体管实现逻辑门（Logic Gate），然后是如何通过逻辑门互连构建计算机的组成单元。我们将在第4章介绍计算机及其组成单元，如控制器、ALU、内存等。

下面开始第一个话题，晶体管。

3.1 MOS晶体管

如今大多数的计算机，至少是大多数的微处理器，都是由MOS晶体管构建而成的。“MOS”代表金属氧化物半导体（metal-oxide semiconductor），有关MOS的电气特性，不在本书讲述范围。它在计算机抽象层次的最底层还要靠下，属于电子学课程知识。换句话说，如果任何原因导致MOS晶体管异常，我们是束手无策的。但是，通常情况下MOS晶体管的工作是非常稳定的，不太可能出现问题。尽管如此，我们还是有必要了解p型和n型等两类MOS晶体管的基本工作原理。它们的工作原理和电灯开关非常相似。

图3-1是一个简单的电路，其中包括：电源（本例中的120V是房屋电力系统）、开关（墙上）和电灯（通过插座接电）。如果希望电灯发热发亮，电子就必须流动；为了让电子流动，必须在电源、电灯之间构成回路，即让电子从电源出发，流过电灯并返回电源。通过图中开关的闭合或断开，可以实现该回路的流动或断开，从而实现电灯亮或灭的控制。

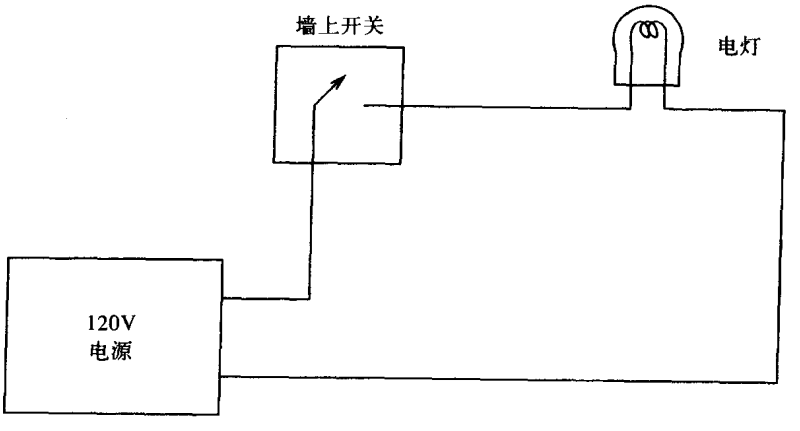


图3-1 电灯开关电路

然而，我们也可以通过一个n型或p型MOS晶体管代替图3-1中的开关，从而控制回路的通断。如图3-2所示，其中图3-2a表示的是一个n-MOS管结构；图3-2b是n-MOS管控制电路通断的示意图。如图3-2a所示，我们看到该晶体管共有三个引脚（或“电极”），它们分别被称做“栅极”（gate）、“源极”（source）和“漏极”（drain）。在本课程中，我们无需关心这些名称的来源，只需要知道MOS晶体管的一个重要特性：对于n-MOS晶体管，如果在栅极接入2.9V电压，则在源极和漏极之

间就会产生一条通路，这时候n-MOS晶体管就等于是一根连通线，专业术语上称“导通”或“闭路”。如果栅极电压是0V，则源极和漏极之间断开，我们称源极和漏极之间“断开”或“开路”(open circuit)。

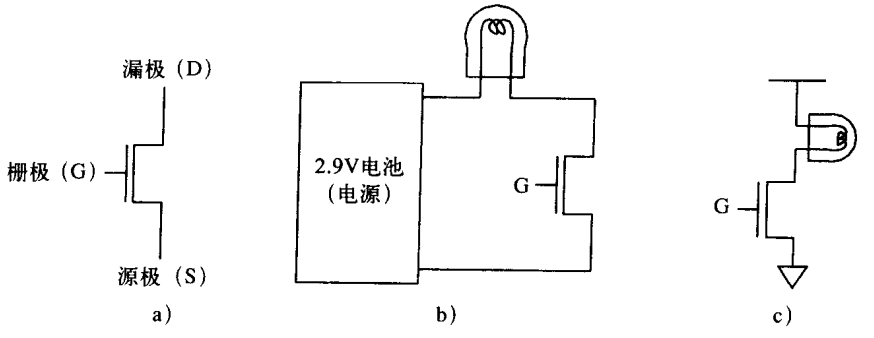


图3-2 n型MOS管

图3-2b是一个由n-MOS管、电池和小灯泡组成的简单电路。如果在n-MOS管的栅极加上2.9V电压，则MOS管导通，从而在电池和灯泡之间构成了闭路，将灯泡点亮。如果栅极电压是0V，则MOS晶体管开路，闭合回路断开，灯泡熄灭。

图3-2c是图3-2b的简化示意图。在这种图中，工程师们通常只将电源的正、负极（地）在器件附近标注出来（而不是让电源线画满整个电路图）。标识方法是：横线代表正极、下三角代表接地。事实上，电源模块及其在整个电路中的连接关系非常简单，易于理解，所以电路原理图中通常不予标识出来。

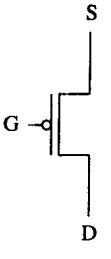


图3-3 p型MOS管

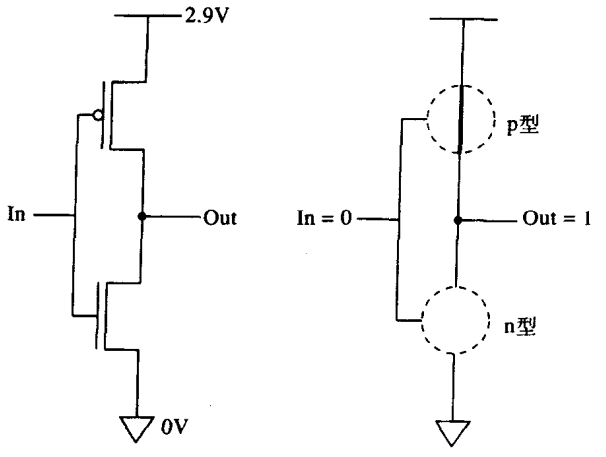
p型MOS管和n型MOS管的工作机制完全相反（或称为互补）。图3-3是p-MOS管的示意图，当p-MOS管的栅极电压为0V时，源极和漏极之间相当于一根导线，即线路导通；而当栅极电压为2.9V时，则p-MOS管开路。由于p-MOS管和n-MOS管的工作方式正好相反，因此电路中如果同时包含p-MOS和n-MOS晶体管，则称该电路为“CMOS电路”，即互补金属氧化物半导体(Complementary Metal-Oxide Semiconductor, CMOS)。

3.2 逻辑门

在第2章中，我们已介绍了AND、OR、NOT等逻辑函数及其真值表，本节将介绍如何使用CMOS电路来实现它们。在电子工程中，通常将实现逻辑函数的CMOS电路称为“逻辑门电路”(Logic Gate Circuit)，如AND、OR、NOT等逻辑所对应的CMOS电路，分别被称为“与门”(AND Gate)、“或门”(OR Gate)和“非门”(NOT Gate)。

3.2.1 非门

图3-4所示是计算机中最简单的一种逻辑结构，它由两个晶体管组成，一个p-MOS管、一个n-MOS管。图3-4a是该结构的原理图，图3-4b则表示该电路在输入为0V时的工作状态。注意，其中p-MOS晶体管处于导通状态，而n-MOS管处于截止状态。因此，输出与2.9V电源连通。与此相反，如果输入电压为2.9V，则p-MOS管不导通而n-MOS管导通，则输出端与地接通（0V）。电路的完整特性可以通过一张表来表示，如图3-4c所示。符号“0”代表0V电压，符号“1”代表2.9V电压，结果这张表与我们在第2章中学过的NOT函数真值表完全相同。



In	Out
0V	2.9V
2.9V	0V

c)

In	Out
0	1
1	0

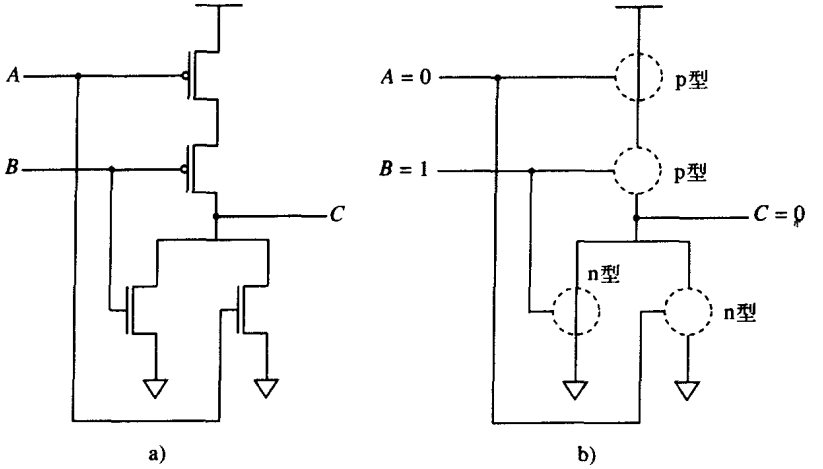
d)

图3-4 CMOS 反相器

换句话说，我们所介绍的这个电路是NOT逻辑（第2章）的具体实现，通常该电路被称为“非门”或“反相器”。

3.2.2 或门、或非门

图3-5所示是或非门。图3-5a是实现或非门的逻辑电路，它由2个p-MOS晶体管和2个n-MOS管组成。



A	*B*	*C*
0V	0V	2.9V
0V	2.9V	0V
2.9V	0V	0V
2.9V	2.9V	0V

c)

A	*B*	*C*
0	0	1
0	1	0
1	0	0
1	1	0

d)

图3-5 或非门

图3-5b表示在*A*端输入0V、*B*端输入2.9V电压时的电路工作状态。在这种情况下，上面的p-MOS管导通，下面的p-MOS晶体管截止，输出端*C*与2.9V电源断开。然而，由于最左边的n-MOS管导通，所以*C*与0V（地）接通。

如果*A*和*B*端输入都为0V，则2个p-MOS管同时导通，因而输出*C*接通2.9V电压。注意，此时不存在二义性（或不确定性），因为底下的2个n-MOS管都是断开的，所以*C*与0V电压之间不存在连接。

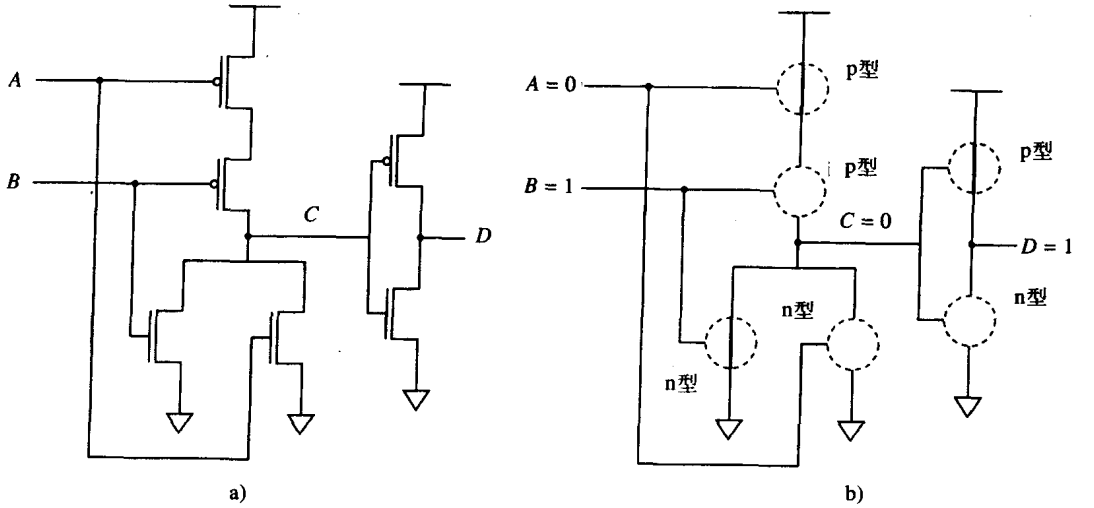
如果*A*和*B*端输入都是2.9V，则对应的2个p-MOS管都处于开路，输出端*C*与2.9V电压完全断开。而输入的2.9V电压将底下的2个n-MOS管打通了，因而*C*端接地（0V）。

图3-5c总结了图3-5a所示电路的可能状态。列出了输入*A*和*B*的4种不同电压组合：

$$A = 0\text{V},\ B = 0\text{V}$$
$$A = 0\text{V},\ B = 2.9\text{V}$$
$$A = 2.9\text{V},\ B = 0\text{V}$$
$$A = 2.9\text{V},\ B = 2.9\text{V}$$

如果将电压值替换为等价的逻辑值，则对应如图3-5d所示的真值表。我们发现其中*C*的输入与OR逻辑完全相反。所以，我们称之为“或非”函数，简称NOR。实现NOR逻辑的电路被称为“或非门”。

如果在图3-5a所示电路的后面加一级反相器，如图3-6a所示，则端口*D*的输出就是OR逻辑。图3-6b是*A*输入为0、*B*输入为1时的电路工作状态。图3-6c是该电路对应的真值表。



A	*B*	*C*	*D*
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

c)

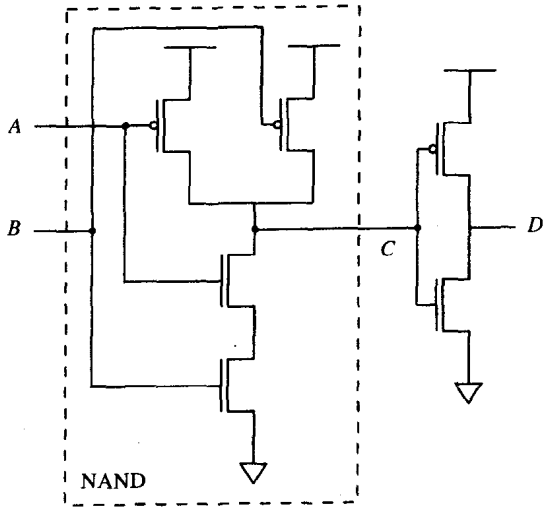
图3-6 或门

3.2.3 与门、与非门

图3-7是与门电路。其中*A*或*B*任一端输入为0V，*C*与2.9V电源直接连通。而*C*为高电平，则意味着它控制的n-MOS管导通并将*D*接地。总之，在图3-7所示的电路中，如果*A*或*B*中任一端输入为0V，则导致*D*端输出为0V。

同样，我们发现该电路中也不存在二义性。对此我们做一个分析：如果*A*和*B*中有任一端输入为0，则*A*和*B*控制的2个n-MOS管中，至少有一个是开路的（断开），结果是*C*与地之间至少是断开的。*C*输出为2.9V，意味着它所控制的p-MOS管是开路的。所以，*D*与2.9V之间是不相通的。

如果*A*和*B*的输入都是2.9V，由*A*、*B*控制的2个p-MOS管都是开路的，2个n-MOS管都导通，因而*C*接地。由于*C*接地，则最右边的p-MOS为闭路，强制*D*为2.9V。



a)

A	B	C	D
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

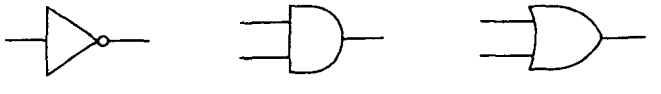
b)

图3-7 与门

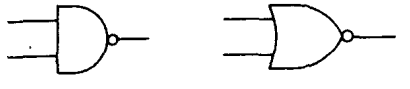
图3-7b是图3-7a电路对应的真值表。注意，该电路是一个“与门”（AND Gate），而虚线内的电路是NOT-AND门（输出为*C*），简称“与非门”（NAND Gate）。

以上介绍的都是在数字逻辑和数字计算机中常用的一些门电路。例如，奔腾IV微处理器中，就包含了几百万个反相器（NOT Gate）。为方便起见，图3-8给出了这些门电路的标准符号。注意，在反相器、与非门和非门等逻辑门的输出端之前，都有一个小圆圈，它表示取反操作，即NOT。

有了这些符号，在以后的电路图中，将不再画出晶体管了，而是采用如图3-8所示的这些符号来代替特定的逻辑功能。



a) 反相器　　b) 与门　　c) 或门



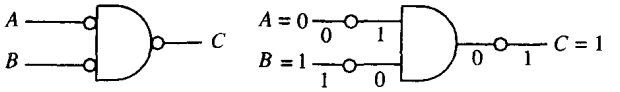
d) 与非门　　e) 或非门

图3-8 基本逻辑门

3.2.4 摩根定律

注意，图3-9a所示输入信号在进入逻辑门之前可以被取反，如图中AND门输入端的两个圆点标识，代表的就是“取反”操作。假设对与门的两个输入*A*和*B*事先取反，同时对其输出也取反。

下面我们看看它所产生的效果。



a) b)

A	B	$\overline{A}$	$\overline{B}$	AND	C
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

c)

图3-9 摩根定律

图3-9b所示是*A*输入为0、*B*输入为1时的电路工作情形。为了表示清晰，我们将与门端口与小圆圈之间拉开了距离，从而方便了对信号经过反相前后的变化情况的了解。

图3-9c是图3-9a电路在各种输入组合下，其输出变化的真值表。其中，$\overline{A}$表示*A*的取反（NOT *A*）。

下面是该电路工作情况的代数表示：

$$\overline{\overline{A}\ AND\ \overline{B}} = A\ OR\ B$$

如果用自然语言来表示，它的意思是：

“不存在*A*和*B*都为假的情况”等价于“*A*和*B*之中至少有一个为真”。

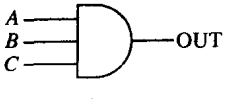
这个等价转换关系被称为摩根定律（De Morgan's law）。试问，如果对一个与门的输入和输出都取反，该等价表达式是否成立?

3.2.5 多输入门

在结束逻辑门话题之前，我们试着将门电路的输入扩展一下。之前介绍的AND、OR、NAND和NOR等4种逻辑门都只有两个输入。我们也可以设计有3个输入的与门或4个输入的或门。例如，对于一个*n*-输入的与门，只有当所有*n*个输入都为1的时候，*n*-AND门的输出才为1；如果其中任意一个输入为0，则*n*-AND的输出为0。同样，对于一个*n*-输入的OR门，如果任意一个输入为1，则*n*-OR门的输出为1；而只有全部输入为0，*n*-OR门输出才为0。

A	B	C	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

a)



b)

图3-10 3输入与门

图3-10所示是一个3输入与门。图3-10a是3输入与门的真值表，图3-10b是3输入与门的符号标识。

试问，能否画出3输入与门的“MOS管实现电路”？以及4输入与门、4输入或门?

3.3 组合逻辑

现在我们已掌握基本逻辑门的工作原理，下一步的任务是用这些基本逻辑门电路构建逻辑结构，就是构建计算机微结构所需要使用的结构单元。

逻辑结构分为两大类，一类是可以存储信息的，另一类是不能存储信息的。前者将在3.4、3.5和3.6节中介绍，后者将在本节讲述。这些结构（不能存储信息的）有时被称做“决策单元”(decision element)。通常它们又被称做“组合逻辑结构”(combinational logic structure)，这是因为

它们的输出状态完全取决于“当前”逻辑结构的输入。换句话说，逻辑结构的输出状态不依赖于任何历史信息（比如记忆在结构内部的），因为这类逻辑结构根本就不具备“记忆”信息的能力。

我们将介绍的3个组合逻辑结构是：译码器（decoder）、多路复用器（mux）和全加器（full adder）。

3.3.1 译码器

图3-11是一个2输入译码器的逻辑门结构，译码器的特点是，在所有输出中有且仅有一个为1，其余皆为0。每个输出端口对应了一种输入模式（input pattern），因而该结构可以用于检测、匹配不同的输入模式。通常，若译码器有n个输入，则有2^n个输出。我们一般这样说：在特定输入模式下，相应输出线被置位。换句话说，对于任一输出线，它仅在一种情况下才为1，其他情况下皆为0。图3-11所示，译码器的两个输入A和B有4种组合，而任一输入组合下，只对应一个输出线为1。例如，图3-11b中译码器在输入为“10”时，将导致第3根输出线被置位。

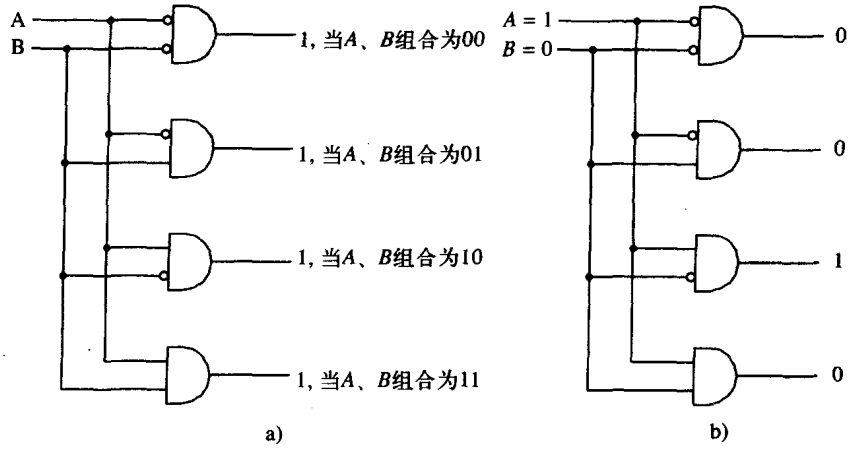


图3-11 2输入译码器

译码器主要用于解释一个二进制数。我们在第5章中将看到，在LC-3中，每条指令的具体操作是由一个被称做“操作码”（opcode）的二进制值决定的，它是一个4-bit数值，是指令的一部分。识别每条指令（或操作码）的4-16译码器，是个非常简单的组合逻辑。

3.3.2 多路复用器

图3-12a是一个2输入多路复用器（multiplexer, MUX）的门电路示意图。MUX的功能就是从多个输入中选择一个，并将其与输出相连。选择信号（图3-12中的S）负责决定究竟选择哪一个。mux的工作原理如图3-12所示，如果S为0（如图3-12b），多路开关最右边一个与门将输出0；现在来看最左边那个与门，若A端输入0，则右边与门输出必然为0（因为与门输入中有一个为0则输出必然为0）。所以，左边与门的输出完全取决于A的值。换句话说，在$S = 0$的情况下，$A = 0$则左边与门输出也为0；$A = 1$则左边与门输出也为1。随后是下面的或门，由于右边与门的输出为0，所以它对或门的输出不起作用。结果是，或门的输出C与与门的输出完全相同。推理的结果是，在$S = 0$的情况下，输出C与输入A完全相同。

相反，如果选择信号$S = 1$，则B与1相与（AND），进而C与B完全相同。

总之，输出C的值或与输入A相同、或与输入B相同，究竟是选择哪一个，完全由选择信号S决定。通常，我们说S将某个MUX输入源（A或B）连至输出端。图3-12c所示是MUX的标准标识符号。

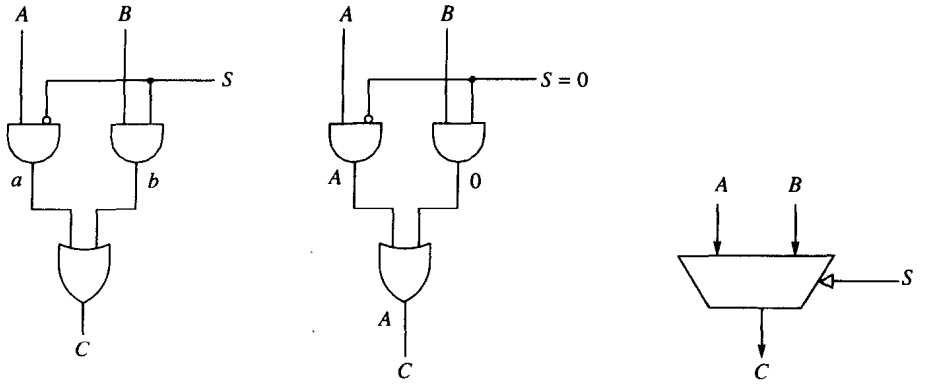


图3-12 2选1多路复用器

多路复用器（mux）通常由2^n个输入、1个输出和n个选择线组成。图3-13a是一个4选1多路复用器的示意图，它由4个输入和2个选择线组成，而图3-13b是4选1多路开关的标识符。

试问，你能画出8选1多路复用器的示意图吗？那么它需要多少个选择线？

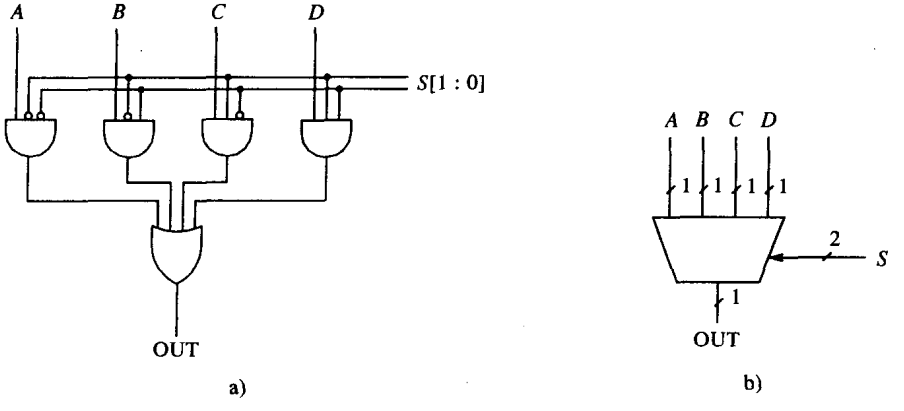


图3-13 4选1多路复用器

3.3.3 全加器

在第2章中，我们已讨论过二进制加法。简单的二进制加法过程与十进制加法过程非常相似，从右向左，每次处理一行，将当前列的两个数字与之前进位等3个数字相加，然后产生当前位结果和传至下一列运算的进位。惟一不同的是，在二进制加法中满2进1，而十进制加法中满10进1。

图3-14所示是n-bit二进制数加法的真值表。真值表表示的是两个n-bit操作数的相加运算。每列[⊖]加法包含3个位相加，即两个二进制数位（a_i和b_i）和1个来自前一列的进位（$carry_i$）。计算结果有两个，一是当前求和位（S_i），二是将传至下一列运算的进位$carry_{i+1}$。例如，如果这3个bit中只有一个为1，则$S_i = 1$、$carry_{i+1} = 0$；如果3个bit中有两个为1，则$S_i = 0$、$carry_{i+1} = 1$；如果3个bit全为1，则$S_i = 1$、$carry_{i+1} = 1$。

⊖ 原文是“At each column”，但此处“每列”的理解是：它不代表真值表中的“列”，而是表示两个二进制数排成两行做加法运算时，每“列”包含两个操作数的相同位置bit及其累加结果。——译者注

a_i	b_i	$carry_i$	$carry_{i+1}$	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

图3-14 二进制加法器的真值表

图3-15是图3-14真值表的门电路描述。在图3-15中，对应（a_i、b_i、$carry_i$）的8种组合，每一种情况下，都有且仅有一个与门输出为1。而C_{i+1}应该只在图3-14真值表中$carry_{i+1}$ = 1的那些组合下，才输出1，这意味着该或门的输入应该来自对应组合下的与门。与此类似，S_i的各个输入端与对应与门输出之间的相连，也将参考图3-14的真值表，即真值表中S_i = 1对应的组合情况。

注意，输入组合000的情况下，在真值表中既不影响$carry_{i-1}$也不影响S_i，因而其对应的AND之输出不与C_{i+1}和S_i中任何一个或门相连。

我们称如图3-15所示的逻辑电路为“全加器”。它对3个输入（a_i、b_i和$carry_i$）相加，产生2个输出（S_i和$carry_{i+1}$）。

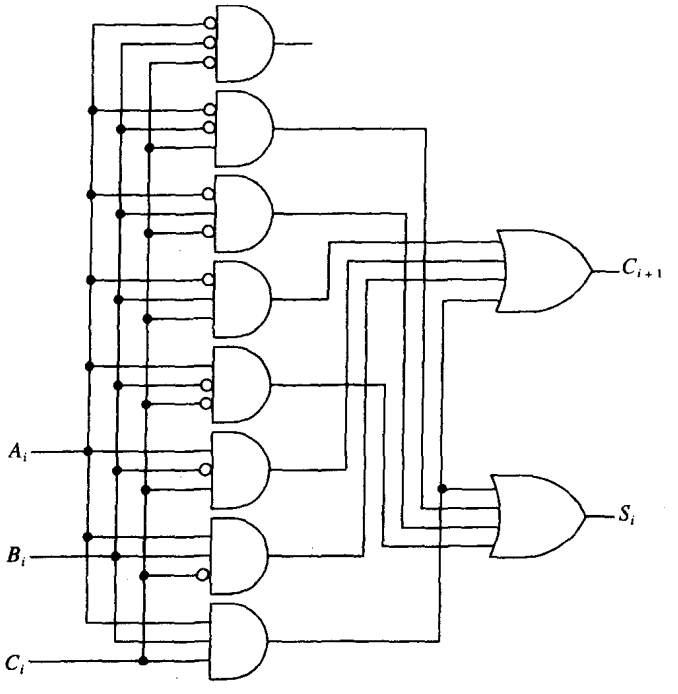


图3-15 全加器的门电路

图3-16是一个4-bit二进制数加法器电路。包含4个全加器模块（如图3-15所示），其中第i个全加器的进位输出是第i+1个全加器的输入。

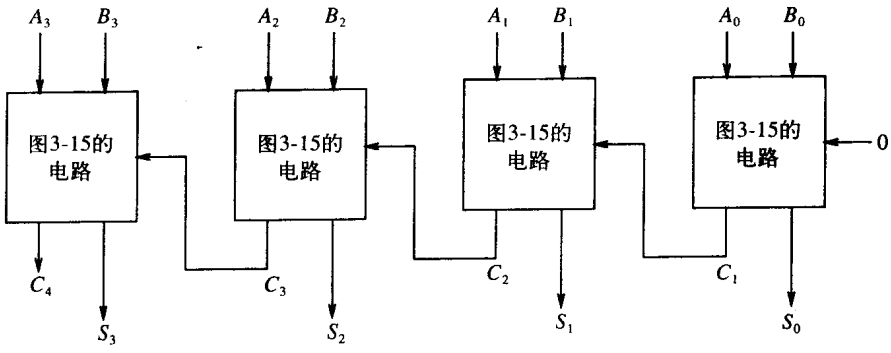


图3-16 4-bit加法器的电路图

3.3.4 可编程逻辑阵列

我们说如图3-15所示的逻辑门单元集合，是一个可以实现任何逻辑函数的可构建模块(building block)。这种可构建的模块又被称为可编程逻辑阵列（Programmable Logic Array, PLA)。它包含一组与门（或称为AND阵列）以及一组或门（或称为OR阵列)。其中，与门的数目对应真值表的输入组合数目。如果逻辑函数的输入数目是n，则PLA需要2^n个AND门。图3-17中包含了2^3个3输入端口的与门；而或门的数目则取决于真值表的输出数目。实现算法是将特定与门的输出与对应或门输入相连，对应关系参考真值表。在真值表中，找出特定输出变量（对应一个或门）列中值为1的那些行（每行对应一个输入组合)，所有这些输入组合对应的与门的输出都将是该或门的输入。这就是“可编程”。换句话说，通过“编程”多个与门（不同的输入组合）与多个或门（不同的输出）之间的连接关系，可以实现任何我们所期望的逻辑函数（或功能)。

我们可以将图3-15理解为就是这样一个“编程”实现，它在8个与门和2个或门之间产生互连，同时实现了两个3输入函数（sum和carry)。图3-17所示是一个3输入4输出PLA。换句话说，通过合适的连线关系（在与门和或门之间)，我们可以同时实现任意4个不同的函数。

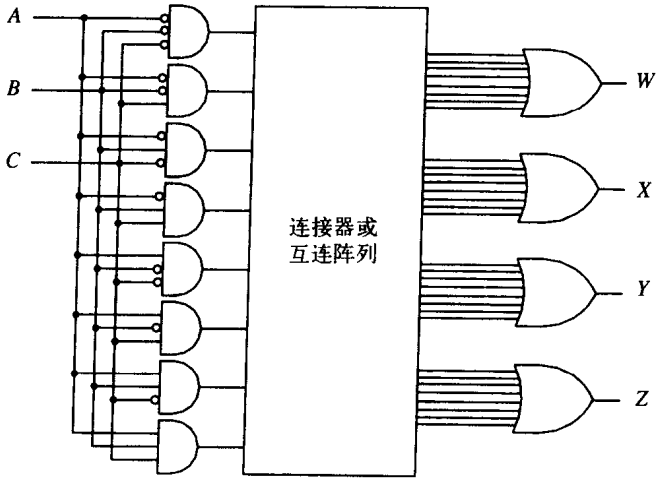


图3-17 可编程逻辑阵列

3.3.5 逻辑完备性

最后，我们将指出逻辑构建中的一个重要概念——逻辑完备性（Logical Completeness)。如

3.3.4小节所表明的，通过PLA可以实现任何形式的逻辑，但PLA却只包括OR、AND和NOT这三种逻辑门。换句话说，只要有足够的AND、OR和NOT门，我们就可以实现任何逻辑函数。因此，我们称{AND，OR，NOT}逻辑门集合是“逻辑完备的”。这意味着，不需要任何其他门电路，通过不同数量的AND、OR、NOT，就可以实现任何真值表。可能这个“数量”非常庞大，但无论如何是可实现的，这就是所谓的“完备性”。

3.4 存储单元

在3.3节中我们曾提到，逻辑电路分为两大类：一类是可以存储信息，另一类不能存储信息。前面给出的三个例子（译码器、多路开关及全加器），都属于不能存储信息的逻辑电路。下面将介绍具有信息存储功能的数字逻辑。

3.4.1 R-S锁存器

最简单的存储单元是R-S锁存器[⊖]（R-S latch）。它能够存储一个位的信息。实现R-S锁存器的方法有多种，最简单的实现方法如图3-18所示。它由两个NAND门互连而成，其中一个NAND门的输出是另一个的输入。

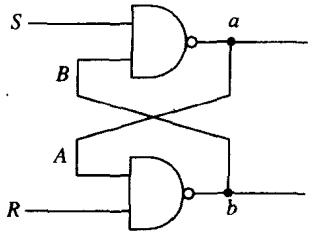


图3-18 R-S锁存器

R-S锁存器的工作机制是：一开始R-S锁存器处于静态（quiescent），即输入R和S都为1。我们假设第一种情况输出a为1，即意味着输入A也为1，且此时输入R也为1（静态），从而由NAND门特性得知，输出b为0。进而输入B为0，导致输出a为1。换句话说，只要保持输入R和S同时为1，则逻辑电路的状态就不会改变。我们称R-S锁存器这种能够保持状态不变的特性为：具有“记忆”或存储能力。本例中R-S保存的信息值为1（输出a的值）。

相反，如果输出a为0，则输入A必为0、输出b必为1。进而输入B为1，结合输入S的值为1（静态），得知输出a必为0。同样，只要输入R和S保持为1不变，则电路状态就不会改变，我们称此时R-S锁存器保存的值为0。

在保持R为1不变的情况下，一旦将S的输入改变为0，则锁存器输出立刻改变为1。同样，在保持R为1的情况下，一旦R输入为1，则锁存器输出为0。在术语中，我们通常称“设置一个变量为0或1”的操作为“置”（set），如“置0”、“置1”。其中，“置0”操作又可称做“清除”（clear）。

如果清除S，将a变为1，进而A为1。而此时R仍然为1，则b必然为0，从而导致B也为0，进而导致a必然为1。如果此时R又恢复为1，我们发现这并不会改变a的输出，因为B此时仍然为0（对于NAND门，只要其输入之一为0则输出必然为1）。所以，我们说即使S恢复为1，锁存器仍然能“记住”刚才的值1。

同样，我们也可以通过瞬态地置R为0来清除锁存器，即设置锁存器输出为0。

值得注意的是，为保证R-S锁存器正常工作，不要同时设置R和S为0。因为，如果S和R同时为0，则a和b的输出也将同时为1。那么，锁存器状态是不确定的，最终的状态将完全取决于MOS管的电气特性而不是逻辑操作了。至于MOS管的电气特性如何决定最后状态，不是本书的内容，你可能会在相关的电子课程中学习它。

⊖ 原文“R-S Latch”被翻译为“R-S锁存器”（而不是触发器）。事实上，锁存器（latch）、触发器（trigger）、寄存器（register）有相同的含义和特性，它们之间没有本质区别。细微的差别在于，latch强调的是它的工作特性，trigger强调的是工作机制（如沿触发或电平触发等），而register则是它在计算机组成中的作用（即n-bit锁存器的组合）。所以，遵循原文，我们在此将其翻译为“R-S锁存器”。——译者注

3.4.2 门控D锁存器

我们有必要对锁存器的置1或清0操作采取一定的控制措施。

图3-19是门控D锁存器的示意图。它由R-S锁存器（如图3-18所示）和控制电路（2个门电路）两部分组成。控制电路的作用是：当且仅当WE有效时（WE = 1），才使得锁存器的值（输出）等于*D*的输入值。WE代表“可写”（Write Enable）。在WE无效时（WE = 0），*S*和*R*的输出都为1。由3.4.1小节的结论我们知道：如果*R*和*S*都为1，则R-S锁存器存储的值保持不变。而当WE瞬间置位（即WE = 1）时，AND门输出*S*和*R*之中有且仅有一个为0。也就是说，此时如果*D*=1，则*S*为0；如果*D*=0，则左下角的NAND输出*R*为0。之前我们已知，如果*S*为0则R-S锁存器为1，如果*R*为0则R-S锁存器为0。所以，此时*D*为1或0将分别导致R-S锁存器的输出为1或0。而当WE恢复为0时，原先的*D*值将被永远存储在R-S锁存器中。

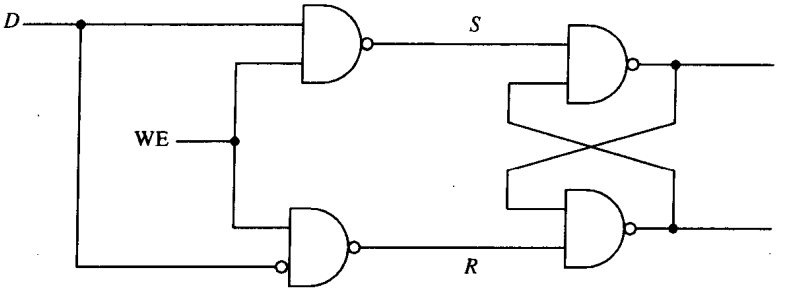


图3-19 门控（gated）D锁存器

3.4.3 寄存器

在第2章中我们看到，计算机处理的数据基本都是由多个bit表示的。例如，将在第5章介绍的LC-3计算机，在大多数情况下它都使用16-bit来表示数据。因此，有必要将这些bit存储为各个独立单元。寄存器（register）就是这样一种结构，它将多个bit组合成一个独立单元。寄存器的bit宽度可大可小，大到需要的任意数，小到只有1个bit。例如，在LC-3计算机中，如图3-33所示，PC、IR和MAR等寄存器的宽度都是16-bit，而*N*、*Z*和*P*等都是1-bit宽度的寄存器。

图3-20所示是一个由4个门控D锁存器组成的4-bit寄存器。我们将该寄存器存储的4个bit分别标识为Q_3、Q_2、Q_1、Q_0。当WE有效时，数值D_3、D_2、D_1、D_0被存入寄存器。

注意：Q_3、Q_2、Q_1、Q_0通常又被标识为$Q[3:0]$。即为每个bit标识一个数字，最右边的bit是bit[0]，依次从右向左递增。如果总共有*n*个bit，则最左边就是bit[*n*-1]。以下面16-bit数为例：

0011101100011110

bit[15]是0，bit[14]是0，bit[13]是1，bit[12]是1，等等。

通常我们还用*Q*[*l*:r]表示一个bit片段，*l*代表left、*r*代表right。我们称这样的bit片段为“字段”（field）。

例如，*A*[15:0]是上面的16-bit数，则

$$A[15:12] = 0011$$
$$A[13:7] = 1110110$$
$$A[2:0] = 110$$
$$A[1:1] = 1$$

还要指出的是，从右至左的bit编号顺序完全是随意的。我们完全可以从左向右编号，让bit[0]

代表最左边一位。事实上，很多人也是这么做的。所以，编号方式是“从左到右”还是“从右到左”并不重要，重要的是保持一致，即一旦确定，必须始终一致。本书遵循的是“从右到左”的方式。

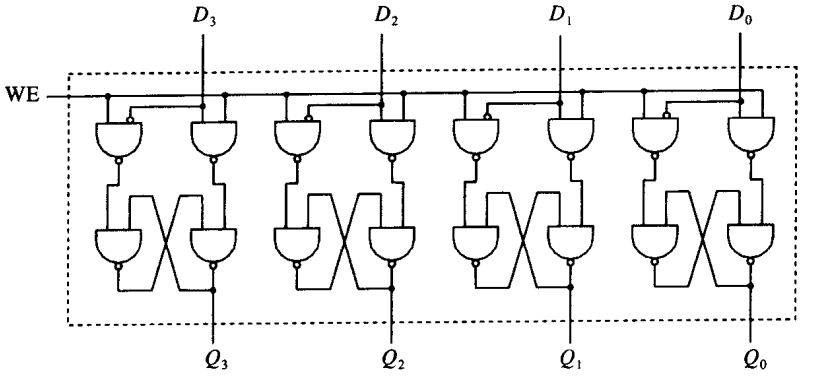


图3-20 4-bit寄存器

3.5 内存的概念

下面我们将介绍电子数字计算机中最重要的结构之一——*内存*。到目前为止，描述内存所需要的工具都已具备。随后，在第4章中会解释内存在一个简要计算机模型中的工作机制，贯穿全书，你必将体会内存概念在计算（computing）过程中的重要性。

内存是由一定数目（通常非常大）的“位置”组成的，其中每个“位置”可以被单独识别并独立存放1个数据。通常，我们称位置识别符为“*地址*”，又称存储在各个位置中的bit数目为“*寻址能力*”（addressability）。

例如，一个PC机的广告商可以这么说：“这个PC机有16MB（兆字节，MegaBytes）的内存”。这句话的意思是（稍后会详细解释），该计算机系统中有16M（million）个内存位置，每个位置能容纳1个字节的信息。

3.5.1 寻址空间

我们称内存中可独立识别的位置总数为内存“*寻址空间*”（address space）。例如，16MB内存指的是，该内存包含1 600万个可独立识别的内存位置。

事实上，1 600万（16M）只是一个近似值。这与我们识别内存位置的方式有关。计算机中所有的东西都采用0、1序列来表示，内存位置的标识方法自然也不例外。表示地址的bit数如果是n，则我们可以识别2^n个存储位置。10-bit的地址可以标识1024个位置，即约等于1000。而如果是20-bit的地址，则有2^{20}个存储位置，这个值大约是1 000 000。那么，16M大小的空间需要24-bit宽度的地址。但是，准确地说，2^{24}地址空间事实上包含16 777 216个位置而不是16 000 000个位置，但口语上，“16兆”说起来更简洁些。

3.5.2 寻址能力

寻址能力是指每个内存位置中包含的bit数目。例如，一个16MB大小的内存，包含16 777 216个内存位置，每个位置存储1个字节（即8-bit）信息。大多数内存都是字节寻址的（byte-addressable）。这是个历史问题，原先的计算机在数据处理或接收键盘输入值的时候，都会将其转换成8-bit的ASCII码（参考第2章）。如果内存是字节寻址的，那么每个ASCII码刚好占用一个存储位置。每个位置都有一个独立的地址标识，无疑方便了读写和修改操作。

许多专用于科学计算的大型计算机采用64-bit寻址。这是因为在科学计算中，数据大都是64-bit浮点数（参考第2章）。正是因为科学计算偏爱64-bit的数据表示形式，很自然，这些计算机的内存系统也被设计成了64-bit寻址规格的。

3.5.3 例子：$2^2 \times 3$内存

图3-21所示是一个$2^2 \times 3$大小的内存。其中，2^2代表内存的地址空间大小为4，3代表寻址能力为3-bit宽度。即该内存的地址空间为4并按3位寻址。大小为2^2的内存地址需要2-bit来表示，寻址能力为3，意味着每个内存位置能存储3-bit信息。内存访问需要对地址进行译码，即译码器将输入$A[1:0]$译码为4个输出线，即4根“字线”（word line）。如图3-21所示，内存中每个字线上包含3个bit（即1个字），这就是“字线”称呼的由来。读取内存时，只要设置地址值$A[1:0]$，则对应的字线被选中输出。内存中每个bit都与对应的字线相与（AND），与其他字线上的相同bit相或（OR）。由于任一时刻有且仅有一个字线被选中，所以这实际上就是一个bit选择的多路开关。3个这样的bit级多路开关并连在一起，就是一个字选开关，即一次读出一个字。

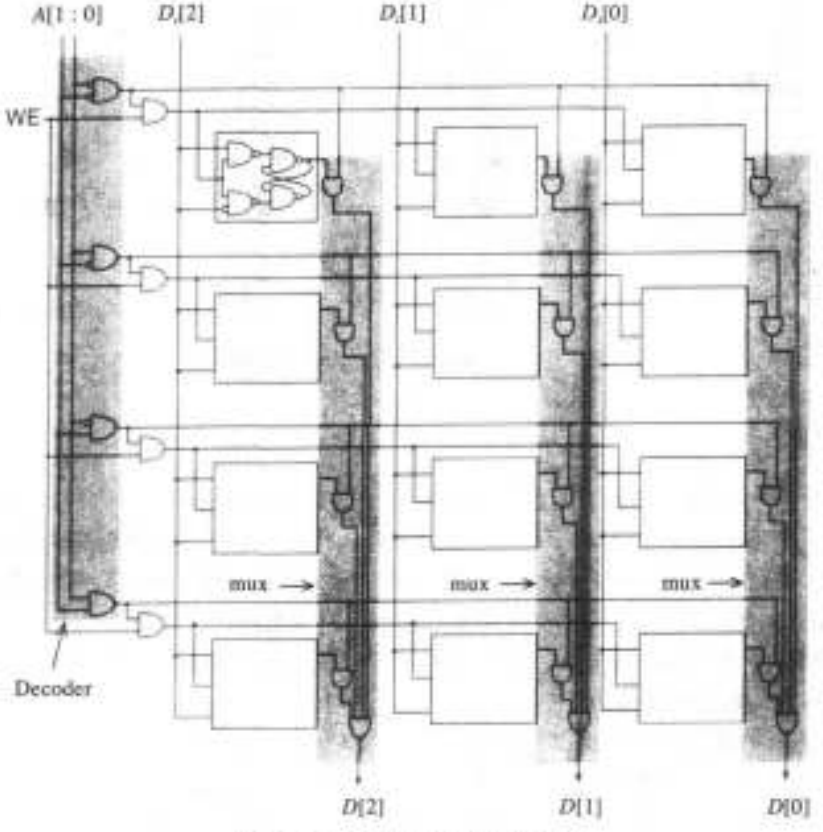


图3-21 $2^2 \times 3$-bit内存的门电路图

图3-22所示是地址3数据读取的流程示意图。数值3的码字是11，所以$A[1:0] = 11$的译码结果是，最下面的字线被选中（其他3根字线未被选中）。该线上3个bit的内容为“101”，所以这3个bit与字线相与（AND）之后，输出为“101”，进一步又输入给OR门。由于其他3个字线信号都为0，所以分别受AND门控制，OR门上其他3个输入必然为0，结果是数据线$D[2:0] = 101$，也就是说，OR门输出的是位置3存储的内容。内存的写操作过程与此类似。地址线$A[1:0]$的内容经过译码器之后，选中对应的字线，然后在WE信号的控制下，将$D_i[2:0]$上已设置的数据写入字线选中的门控锁存器中。

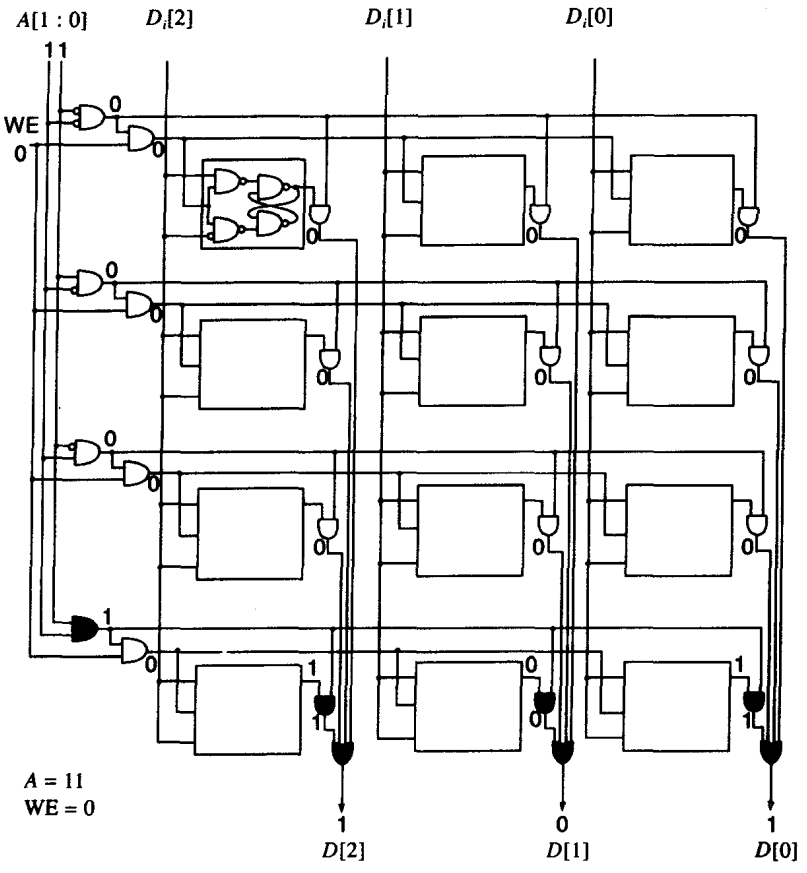


图3-22 从$2^2 \times 3$-bit内存的地址3中读取数据

3.6 时序电路

在3.3节中，我们讨论了多路开关、译码器及全加器等几种逻辑电路，这些逻辑电路有一个共同的特点，就是电路的输出完全取决于电路当前的输入信号，我们称这类逻辑电路为组合逻辑电路（Combinational Logic Circuit）。组合逻辑电路的特点是：它不关心电路以前的状态，电路当前的输出完全取决于电路当前的输入。也就是说，它不能存储电路的“历史”信息。但在3.4和3.5节中讨论的逻辑结构，则能够存储信息，如3.4节的存储单元和3.5节的$2^2 \times 3$-bit大小的内存，但是这些结构单元却不具备数据处理能力。

本节我们将介绍既能处理数据又能存储数据的逻辑电路单元。这种逻辑电路的输出既与当前电路输入相关，更重要的是，它又与之前电路的状态相关，我们称这种电路为“时序逻辑电路”（Sequential Logic Circuit）。时序逻辑电路和组合逻辑电路的不同之处是：时序逻辑电路中具有存储单元，可以跟踪电路以前的状态。图3-23是时序逻辑电路的结构框图。值得注意的是，时序逻辑电路的输出是由电路当前输入信号和存储单元信息共

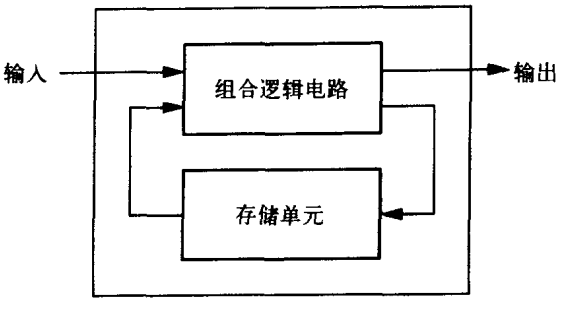


图3-23 时序逻辑电路的结构框图

同决定的，存储单元中的数据反映了电路之前的状态。

时序逻辑电路主要用于实现有限状态机。有限状态机被广泛用于各种工程领域，如电气控制、机械系统、航空等。例如，控制红、黄、绿三种颜色的交通灯控制器就是一个很好的例子，它根据当前颜色（历史信息）及传感器输入信息，设置交通灯的下一个颜色状态。交通传感器可以是道路上的绊绳或光电器等交通监测设备。

在第4章中我们还将介绍，在冯·诺伊曼体系结构中，有限状态控制器是怎样控制整个计算机系统的工作流程的，它将是计算机的心脏。

3.6.1 组合密码锁

通过一个简单例子，我们将看到时序逻辑电路和组合逻辑电路之间的不同。假设某人希望将自行车锁住，但却又不想随身携带钥匙。那么，他可以使用“组合密码锁[⊖]”（Combination Lock）之类的东西。有了密码锁，他只需要记住开锁的“组合码”（combination）即可。图3-24所示是两种常用的密码锁。

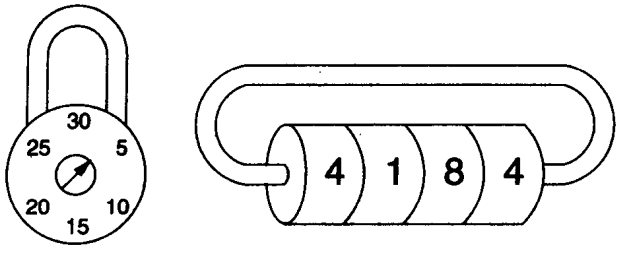


图3-24 两类密码锁的例子

图3-24a所示的密码锁有一个拨号盘，周围刻有0~30等数字。要打开这个锁，必须知道密码组合。假设密码组合是R13-L22-R3，则意味着开锁方法是：右转两满圈[⊖]并继续，直到指针指向13（即R13）；然后左转一满圈并继续，直到指针指向22（即L22）；再右转一满圈并继续，直到指针指向3（即R3）。于是，锁打开了。其中的关键是旋转的“顺序”。如果右转两圈停在20，然后向左转一圈停在22，最后右转一圈指向3。虽然最后停住的数字也是3，但这样锁是打不开的。为什么？因为密码锁能“记住”之前的数字，并将当前输入值（R3）与历史操作结合，最终决定是否将锁打开。这个例子就是典型的“时序结构”。

图3-24b是另一类密码锁。该密码锁有4个轮子，每个轮子上的数字是0~9。分别转动每个轮子，如果4个轮子上的数字排列和设定密码吻合，则锁自动打开。这种锁的特点是：它能否被打开只与4个轮子当前的状态相关，而与轮子旋转过程无关。这个例子则属于“组合结构”。

很奇怪的是，在我们的日常生活中，并不区分这两类锁，统称它们为“组合密码锁”。准确地说，前者应该被称为“时序锁”（Sequential Lock），后者才是真正的“组合密码锁”（Combination Lock）。

⊖ 所谓“组合密码锁”，就是中文俗称的“密码锁”。翻译成“密码锁”可能更直观、形象些，但考虑到文中作者有意区分“组合密码锁”和“时序密码锁”（sequence lock），所以在此将其翻译为“组合密码锁”。——译者注

⊖ 原文是“turning the dial two complete turns to the right”，应该翻译为“右转两满圈”。从原理上来说，1圈或是2圈并不重要。从细节上考虑，在解锁过程的开始（当且仅当）转2圈（而不是1圈），可以起到“复位”的作用，或许原因如此，仅供参考。——译者注

3.6.2 状态的概念

图3-24a所示，在时序密码锁的工作机制中，必须记录每次旋转的数字及其顺序。事实上，“R13-L22-R3”是惟一可以开锁的序列。例如，序列R13-L29-R3、R10-L22-R3都不能开锁。问题是，在任何时候，旋转是锁系统的惟一输入。

下面是图3-24a所示密码锁开锁过程经历的主要状态阶段：

A．闭锁状态，且无外部操作。
B．闭锁状态，刚完成R13操作。
C．闭锁状态，已完成R13-L22操作序列。
D．开锁状态打开。

如果A、B、C、D代表其中的各个阶段，则称它们为密码锁的状态。

在计算机工程中，状态是一个非常重要的概念，它在各类工程技术中都非常重要。系统的某个状态，可以理解为是系统在特定时刻和特定条件下的快照（snapshot）。

换句话说：*系统的状态是系统中所有组成要素在拍照时刻的一个“快照”。*

图3-24a所示的密码锁有四个状态：开锁（状态D）、闭锁且无外部操作（状态A）、闭锁但已接受一个正确序列码（状态B）、闭锁但已接受两个正确序列码（状态C），这些是可能存在的所有状态。试问：果真如此吗？是否存在第5种可能的状态？

可以通过状态来描述系统的例子很多。例如，记分牌可以代表篮球比赛的状态。图3-25中的记分牌表达的信息包括：TEXAS队和OKLAHOMA队当前比分是“73:68”，下半时，比赛剩余时间是“7分38秒”，投篮时限还剩14秒，TEXAS队目前控球，两队分别犯规4次。这就是篮球比赛的一个快照，它描述了比赛当前的状况。假设12秒后TEXAS队投中两分，则记分牌更新。更新后，记分牌显示信息是：TEXAS和OKLAHOMA队比分为“75:68”，下半时剩时“7分26秒”，投篮时限复位为25秒[⊖]，OKLAHOMA队控球。

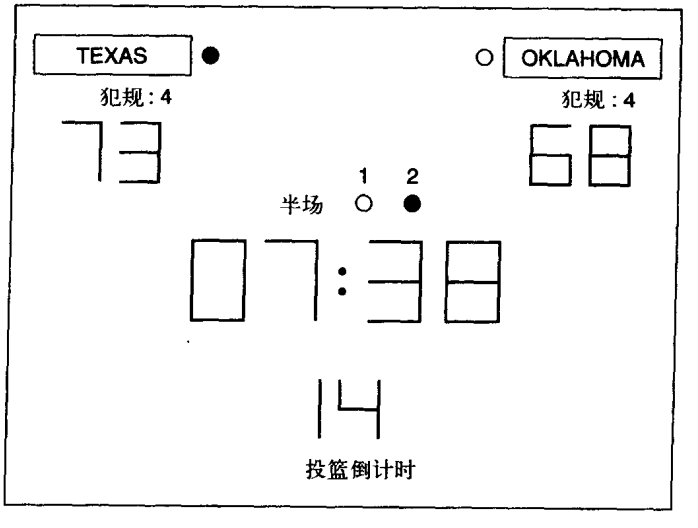


图3-25 一个篮球记分牌状态的例子

游戏“三子连珠”（tic-tac-toe）也可以描述为状态表示方式。这是一个双人游戏（当然，也可以是人和计算机对弈）。这里，状态代表的是，每次计算机等待人走下一步棋的时刻，游戏的快照。游戏规则如下：棋盘是一个3×3方格，人和计算机轮流填充棋盘上的空格，分别填充“X”（人）

⊖ NBA标准规定的“投球时限”（shot clock）是24秒，即24秒内必须投篮，否则违例。——译者注

和“O”（计算机）；人先走；胜者是最先将三个X或O排成一条直线的，这条直线可以是水平的、垂直的或斜线。

初始状态如图3-26a所示，所有方格为空。图3-26b是游戏进行中的一个状态，代表目前轮到人（X）走第二步棋了。状态表明他/她的第一步是在棋盘左上角填了X，而计算机的第一步则是在中心位置填了O。如图3-26c所示状态表明，人在第二步选择了右上角填X，而计算机随后在顶行中间填O。

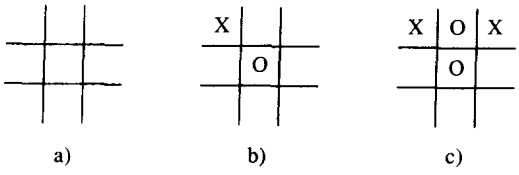


图3-26 三子连珠的三个状态

3.6.3 有限状态机

正如我们所见，状态是系统各相关部件在特定时刻的一个快照。不同的时刻，系统处于不同的状态。所以，通常我们使用“有限状态机”（Finite State Machine, FSM）来描述系统的行为。

有限状态机由五个组成部分：

(1) 状态（有限数目）。
(2) 外部输入（有限数目）。
(3) 对外输出（有限数目）。
(4) 任意状态间迁移（显式注明）。
(5) 对外输出操作（显式注明）。

状态集合表示系统可能处于的所有状态，状态迁移表示从一个状态转换到另一个状态所需要的各种条件。

1. 状态图

有限状态机的常用表达方式是状态图（state diagram）。图3-27是一个状态图的例子。一个状态图由状态节点和节点间的连接线组成。其中，每个状态节点代表系统的一个状态，连线和箭头代表一个状态到另一个状态的转换。术语中，我们称这是一个从“当前状态”到“下一状态”的转移。图3-27所示的状态图中包含3个状态和6个状态转移。注意，从状态*Y*到状态*X*不存在转移条件。

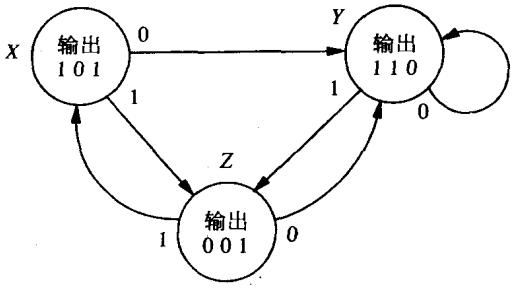


图3-27 一个状态图的例子

通常，当前状态的转移存在多个选择（下一状态）。最终的转移取决于外部输入。例如，在图3-27中，如果当前状态为*X*且输入为0，则下一状态为*Y*；如果当前状态为*X*且输入为1，则下一状态为*Z*。总之，下一状态取决于系统的当前状态和当前输入。

然而，系统输出既可以由当前状态惟一决定，也可以由当前状态和当前输入共同决定。在我们将要学习的所有例子中，系统输出都是由当前状态来决定的。如图3-27所示，当系统状态为*X*时，系统输出101；当系统状态为*Y*时，系统输出110；当系统状态为*Z*时，输出001。

图3-28所示是密码锁系统（见图3-24a）所对应的状态图。假设解锁密码是“R13-L22-R3”；标识为*A*、*B*、*C*、*D*的4个状态代表了“锁的当前状态（开或闭）”以及“至今为止正确操作的次数”等信息；系统输入是可能的密码盘转动操作；系统输出是“开锁”或者“闭锁”。显然，系统输出与每个状态都紧密相关，即状态处于*A*、*B*、*C*时，系统输出为“闭”；处于状态*D*时，输出为“开”。任意状态的所有转移线则代表了在当前状态下的所有可能操作（输入）。例如，当系统处于状态*B*时，所有可能的输入操作可以描述为：“左转至22（L22）”或“除L22之外的任何操作”。因此，对

应这两种可能，从状态*B*出发的转移线为两条。

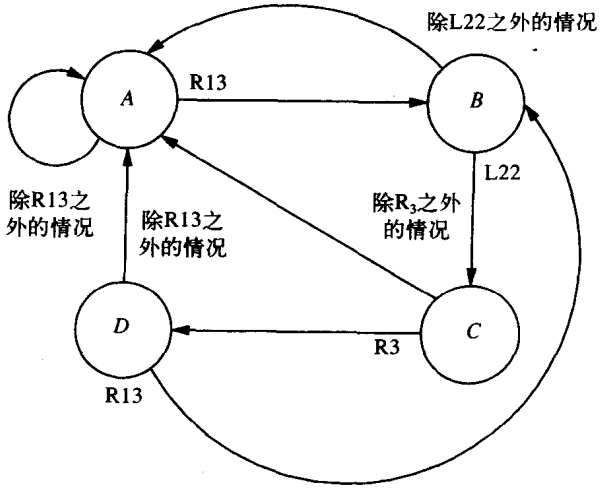


图3-28 对应图3-24的密码锁状态图

同样，我们也可以画出篮球比赛的对应状态图，其中每个状态点代表记分牌内容。引起状态转移的可能情况包括：裁判吹响哨（进球或犯规），或是某方队员得球等。例如，TEXAS投中2分后裁判吹哨，状态就将转移。但是，很快我们就发现，描述篮球比赛所需状态数目很庞大，而状态之间的转移却非常少。系统的输入是自上次转移以来，篮球场上可能出现的任何活动，如TEXAS队投中两分、OKLAHOMA队投中三分、TEXAS队截球、OKLAHOMA队抢篮板成功等等。输出则是最后的比赛结果，有三种可能：比赛正在进行中、TEXAS队获胜或OKLAHOMA队获胜。

试问，你能在TEXAS ：OKLAHOMA比分等于“30:28”和“30:30”等两个状态之间，画出它们之间的状态转移连线吗？可参考习题3.38。

再问，请判断以下情况是否可能？两个状态分别是：TEXAS队领先30:28和两队战平30:30，但两个状态之间却不存在转移线（或转移关系）？可参考习题3.39。

2. 时钟

下面开始讨论有限状态机的一个重要特性——状态转移的触发机制。所谓“触发”，比如在时序锁中，前一轮拨号结束、下一个逆方向拨号开始之前，就是它的转移触发；而在篮球比赛中，裁判的哨声、投篮或对方断球等，是状态转移的触发条件。

通常，状态转移是通过时钟电路来触发的。所谓时钟，是这样一个信号，低电平（逻辑0）和高电平（逻辑1）交替变换。换个数字逻辑的说法是，时钟是0和1交替变换的信号。图3-29是时钟的信号量和时间之间的函数关系，时钟周期（clock cycle）是指时钟信号不断变换的间隔时间。

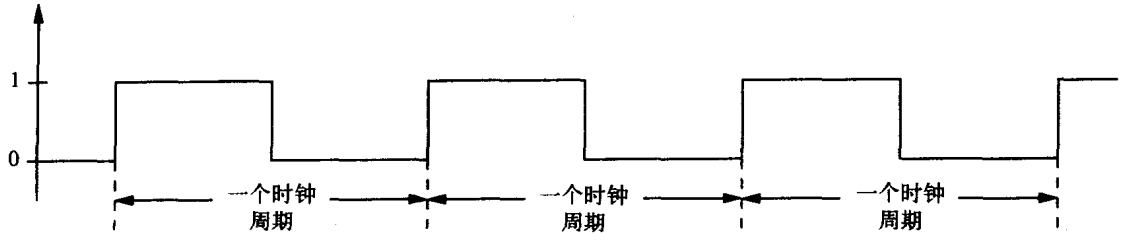


图3-29 一个时钟信号的例子

在电路实现中，有限状态机的状态转移发生在每个时钟周期的起始时刻。

3.6.4 有限状态机的实现

我们将以交通警告牌作为例子，详细介绍有限状态机的原理设计和实现，即交通警告牌控制器的逻辑设计。如图3-30所示，交通牌提示信息包括：危险、右转，以及5个指示灯（用数字1~5表示）。

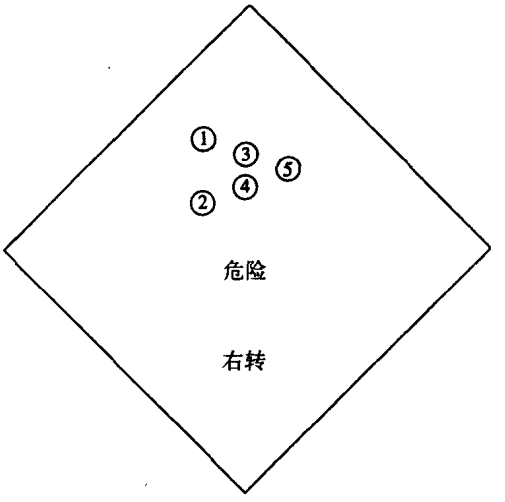


图3-30 交通警告牌

控制器的作用是控制系统的行为，这也是时序逻辑的主要应用之一。在本例中，“系统”是指交通警告牌上各个指示灯，控制器控制这些指示灯开或闭的规则是：在1个周期内，关闭所有的灯；接下来的周期里，打开指示灯1和2；再之后的周期里，打开1、2、3和4；最后，打开所有的灯。随后，重复以上4个阶段：灯灭，1和2，1、2、3和4，全亮。如此反复，其中每个周期持续时间0.5s。

图3-31所示是该交通警告牌的状态机描述。其中，状态机的4个状态分别对应4种相关状况。注意其中状态的迁移条件，如果开关（switch）接通（input = 1），则所有的灯按以上时序亮或灭；如果开关（switch）断开，则状态立刻复位至初始态（所有灯都灭）。

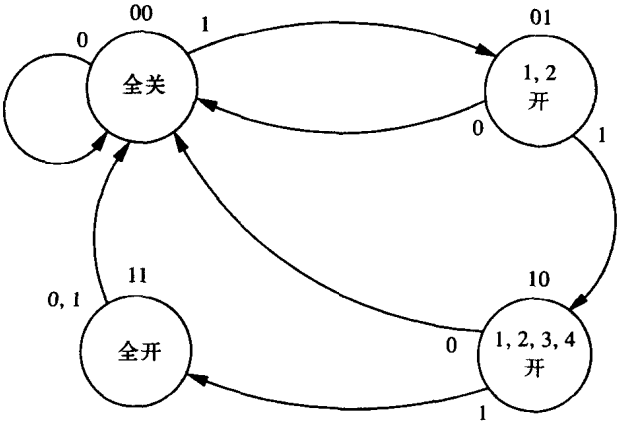


图3-31 交通警告牌控制器的状态图

图3-32所示是有限状态机（见图3-31）的时序逻辑电路。图3-32a是结构框图。注意，该系统包括：一个外部输入，即控制灭或亮的开关（switch）；三个输出，一是控制1和2号灯的点亮，二是控制3、4号灯的点亮，三是控制5号灯。系统内部还包含两个存储单元（element），用于记录控制器的当前状态，它们的值反映了系统之前的状态。最后，注意系统内还包含了一个时钟电路，该时钟以0.5s的周期控制着系统的状态转移。

在存储单元中，需要记录的历史信息（也是惟一相关的）是“之前状态”，即在这之前是从哪一个状态转移过来的。由于本系统只有4个状态，因此2个bit就足以表示它。参见图3-31中4个状态的2-bit标识。

1. 组合逻辑

图3-32b所示是控制器中的组合逻辑部分。组合逻辑电路的输出有两组：一是交通警告牌指示灯的控制信号（外部输出），二是输出给两个存储单元的状态信息（内部输出）。

首先，是指示灯的控制输出。我们之前提过，指示灯控制只需要三个输出即可。其中，5号灯

受控于标识为*X*的AND门输出，因为5号灯点亮的惟一情况是在switch=1且控制器状态为11之时；3、4号灯受控于标识为*Y*的OR门输出，因为这两个灯在两种状态下都将点亮（即状态10和11）；1、2号灯受控于标识为*Z*的OR门，为什么？我们将这个问题作为思考题考考你，参见习题3.42。

然后，是控制存储单元的内部输出。如果系统的下一状态是“11”或“10”，则存储单元1将被设置为1。因为该操作只发生在系统当前状态为“10”或“01”这两种情况下，所以存储单元1受控于标识为W的OR门输出。试问，为什么说存储单元2受控于标识为U的OR门？参见习题3.42。

2. *存储单元*

最后是控制器中（见图3-32a）存储单元的描述。有人可能会问“为什么我们不直接使用3.4节所示的D锁存器来作为存储单元呢？”。其原因是：在当前时钟周期内，存储单元的输出是组合逻辑电路的一个内部输入，而同时组合逻辑电路的输出又是存储单元的输入。该输入应该等到下一个时钟周期才能写入存储单元，如果使用D锁存器，则当前周期开始输入时，就将覆盖存储单元的内容（而不是等到下一周期）。

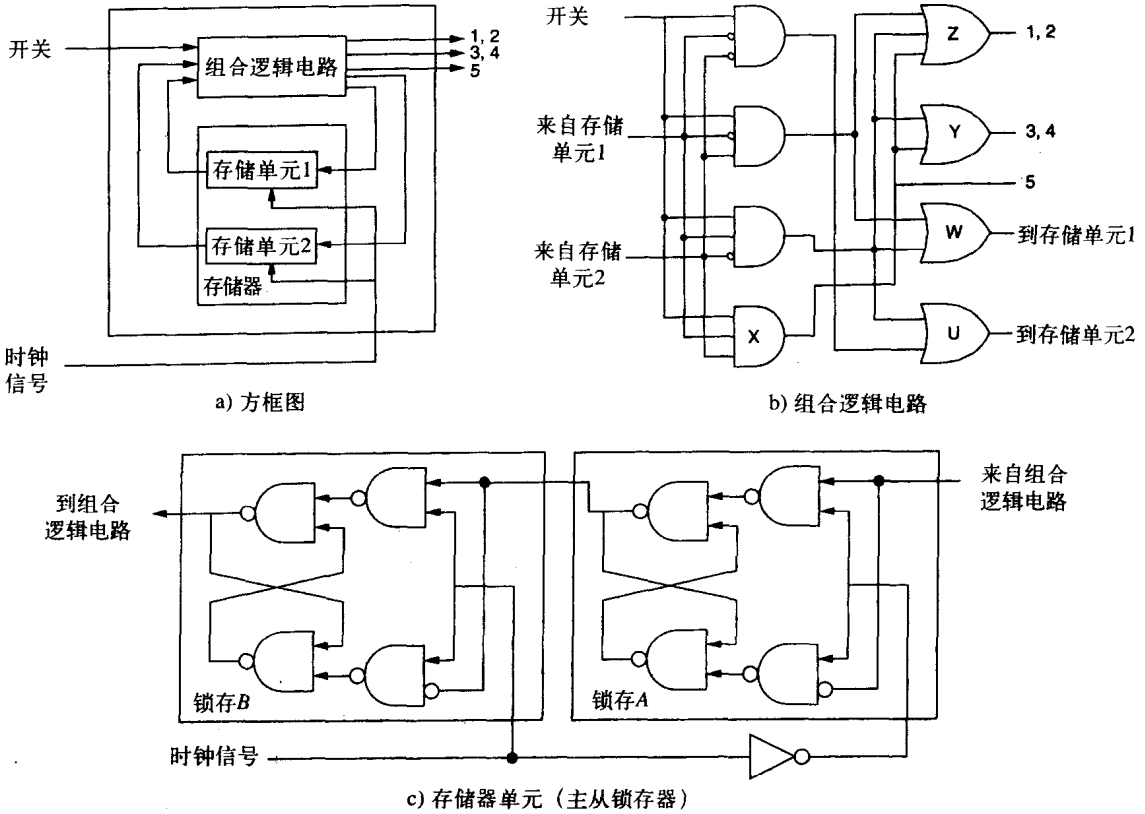


图3-32 用时序逻辑电路实现交通警告牌

为了避免问题的发生，我们采用一种被称为“*主从锁存器*”（master-slave flip-flop）的逻辑电路来实现存储单元。如图3-32c所示，主从锁存器由门限*D*锁存器构成。在时钟的前半周期，锁存器*A*内容不会改变。而锁存器*A*的内容将通过锁存器*B*输出至组合逻辑电路；在时钟的后半周期，锁存器*B*的内容不会改变，即输出至组合逻辑的内容保持不变（即仍然是前半周期的*A*内容）。然而，在后半周期内，锁存器*A*的内容可能发生变化。这就是主从锁存器，它使得当前状态（记忆内容）在整个时钟周期内保持不变，而组合逻辑生成的新状态值，将在后半周期写入锁存器*A*，同时，这个新内容又将在下一周期开始时传递给锁存器*B*，最终输入至组合逻辑电路。

3.7 LC-3计算机的数据通路

在第5章中，我们将介绍一个被称为LC-3的计算机，而且你还可以在它上面编写和运行自己的程序。图3-33所示是LC-3的结构图（即数据通路（data path)），以及控制LC-3的有限状态机。数据通路包含了在计算机内核（core）中所有与处理信息相关的逻辑结构。

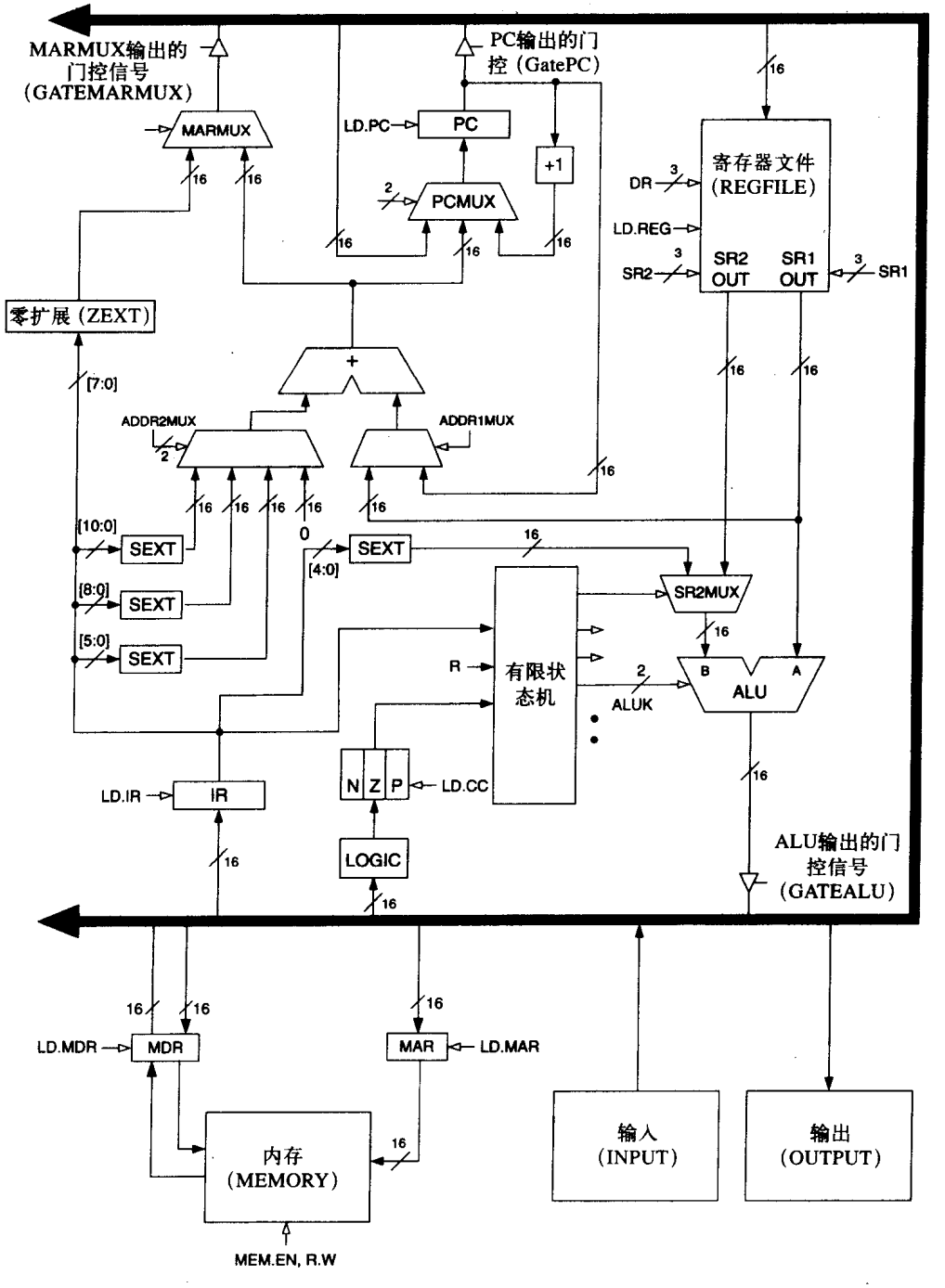


图3-33 LC-3计算机的数据通路

毫无疑问，图3-33看起来非常复杂。现在你不必完全掌握它，因为相关的知识都还未介绍。

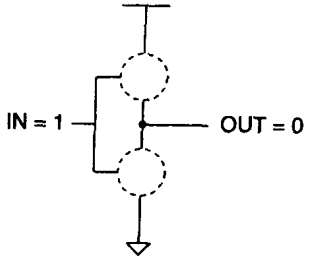
我们将在第5章介绍相关知识及其工作原理。但是，从这张图中，你至少可以看到完成一个计算机所需要的基本结构，我们都已熟悉这些基本结构，同时还了解它们是怎样由门电路实现的。如数据通路中的PC、IR、MAR和MDR等，都是宽度为16-bit的寄存器；用斜线标识的总线（bus）及标识在总线旁边代表总线宽度的数字，如16代表16根线，每条线路包含1个bit信息；再如*N*、*Z*和*P*等是1-bit寄存器，这些寄存器都是“主从锁存器”结构；图中还有5个多路开关，它们是PC寄存器的输入选择、MAR寄存器的输入选择、算术逻辑单元（ALU）B端的输入选择，以及加法器的两个输入选择端。而在第5章，我们将要思考的问题是：为什么以及怎样才能让这些基本结构互连在一起，以使得LC-3能执行程序？目前，让我们仅仅“欣赏”一下这些看起来很眼熟的组成部件。在第4章和第5章，我们将对这些结构单元进一步抽象，然后将它们互连，形成一台真正可运行的计算机。

3.8 习题

3.1 试填写在不同情况下，两种MOS管的通、断状态。

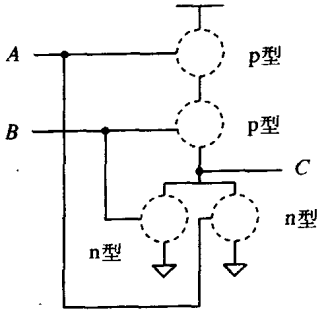
	n型	p型
Gate = 1		
Gate = 0		

3.2 试在图中空缺的地方画上“有连线”还是“没有连线”，条件是输入IN等于逻辑“1”时，输出OUT为逻辑“0”。



3.3 两输入AND门和两输入OR门都是“两输入逻辑”的例子，试问还存在多少种可能的“两输入逻辑”（提示：将2个输入和1个输出的对应关系，而不仅仅是数值，做排列组合。换句话说，就是可能的真值表数目）？

3.4 试在图中空缺的地方画上“有连线”还是“没有连线”，以使得输出*C*为逻辑“1”。再给出输出*C*为逻辑“0”的情况下，输入*A*和*B*的组合。



3.5 给出对应于图3-34所示电路的真值表。

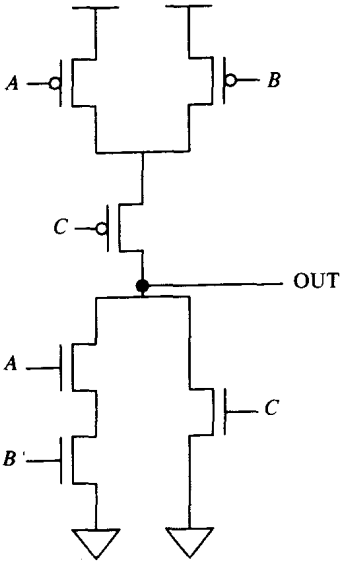


图3-34 习题3.5的结构框图

3.6 填写对应于图3-35所示电路的真值表，并写出Z以A、B为变量的逻辑表达式。

A	B	C	D	Z

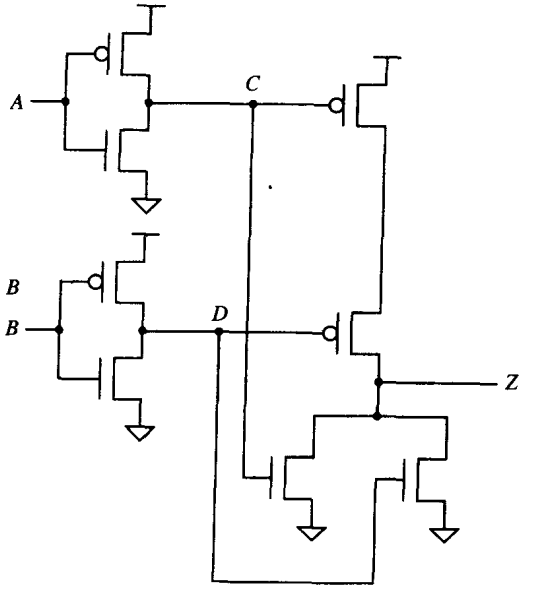
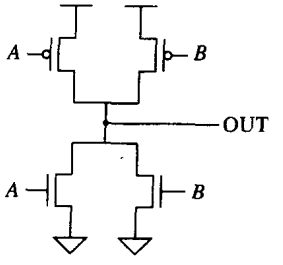
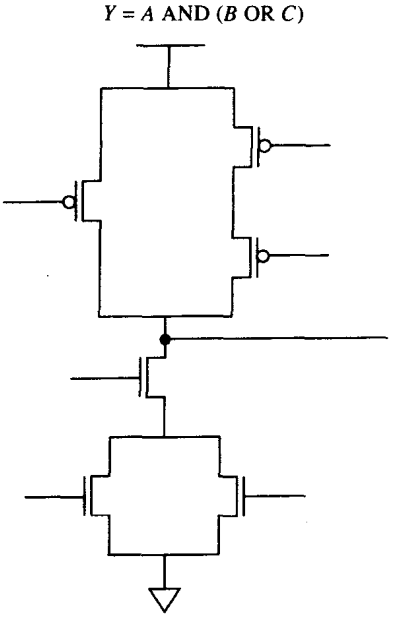


图3-35 习题3.6的结构框图

3.7 试找出下图中的一个严重错误（提示：匹配不同输入下电路的输出）。



3.8 下面电路对应于逻辑表达式Y = NOT(A AND (B OR C))的实现。请为电路标注对应的输入、输出符号。



3.9 填写对应于逻辑表达式NOT(NOT(A) OR NOT(B))的真值表，它对应的是什么逻辑门?

A	B	NOT(NOT(A)OR NOT(B))	A	B	NOT(NOT(A)OR NOT(B))
0	0		1	0	
0	1		1	1	

3.10 填写两输入非门的真值表。

A	B	A NOR B	A	B	A NOR B
0	0		1	0	
0	1		1	1	

3.11 a. 画出3输入与门和或门的MOS晶体管级电路图。提示：参考图3-6a和图3-7a，对其进行扩展。

b. 在问题a的电路图上，对应下列条件，标注电路图中MOS管的导通（连线）或断开（无连线）。

(1) $A = 1, B = 0, C = 0$

(2) $A = 0, B = 0, C = 0$

(3) $A = 1, B = 1, C = 1$

3.12 参考习题3.11a，画出3输入译码器的MOS级电路图。对应译码器的每一个输出，写出其对应的输入组合。

3.13 试问5输入译码器应该有多少根输出线?

3.14 试问16输入多路开关应该有多少根输出线？应该有多少根选择线?

3.15 如果A和B是4-bit无符号整数0111和1011，试用图3-15所示的全加器计算A与B之和S，并填写下面表格。请用十进制加法验证A和B之和，是否与S吻合？为什么?

C_{in}				0
A	0	1	1	1
B	1	0	1	1
S				
C_{out}				

3.16 给定下面真值表，请画出对应的逻辑电路（提示：参考3.3.4节的实现算法）。

A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

3.17 a. 给定4个输入A、B、C、D，1个输出Z，设计一个真值表，要求其中至少有7种输入组合的逻辑输出为1（另：该真值表共有多少行？）。

b. 画出以上真值表对应的门级逻辑电路。参考3.3.4节的实现。

3.18 基于与门、或门和非门，实现以下函数。其中，A和B为输入，F为输出。

a. F为1，当且仅当$A=0$、$B=1$。

b. F为1，当且仅当$A=1$、$B=0$。

c. 基于a和b的答案，实现一个1-bit加法器。该1-bit加法器的真值表如下所示：

A	B	Sum
0	0	0
0	1	1
1	0	1
1	1	0

d. 试问，能否使用4个完全相同的1-bit加法器（如c中的加法器）实现一个4-bit加法器？为什么不可以？缺少什么信息？提示：当A和B都为1时，和为0，其中什么信息被丢失了？

3.19 图3-36所示的逻辑电路（一）有3个输入A、B、C，图3-37的逻辑电路（二）有两个输入A、B，两个电路的输出都是D。但两者之间存在本质的区别是什么？提示：当输入A从逻辑0变为逻辑1时，两个电路有什么不同？

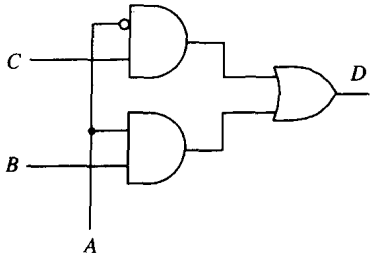


图3-36 习题3.19的逻辑电路（一）

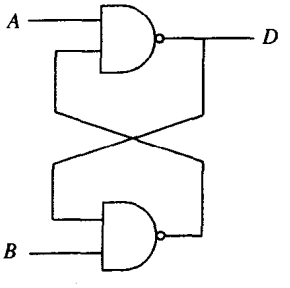


图3-37 习题3.19用到的逻辑电路（二）

3.20 画出下面真值表对应的门级电路图，以及对应的晶体管级电路图，并对照真值表验证晶体管级电路图是否正确。

in_0	in_1	$f(in_0, in_1)$
0	0	1
0	1	0
1	0	1
1	1	1

3.21 我们通常称8-bit为1个字节（byte），4-bit为一个半字节（nibble）。如果一个byte寻址的内存的地址表示宽度是14-bit，试问该内存包含多少个nibble？

3.22 试基于“2选1”多路开关实现一个“4选1”多路开关。注意，“4选1”多路开关包含4个输入、2个选择线、1个输出。写出该电路对应的真值表。

3.23 已知图3-38所示的逻辑电路，填写真值表中Z的值。

A	B	C	Z
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

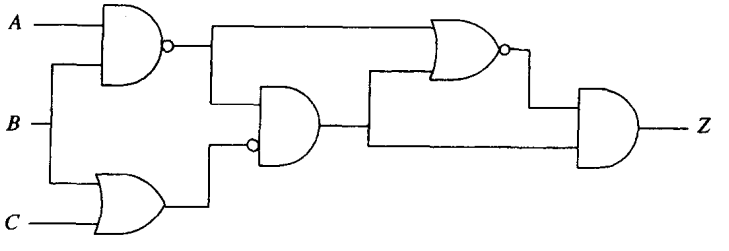


图3-38 习题3.23用到的结构框图

3.24 a. 图3-39所示是在大多数处理器中都用到的逻辑电路，其中每个方框是一个全加器。试问，信号X的作用是什么？或者说，X = 0和X = 1，电路输出有什么不同？

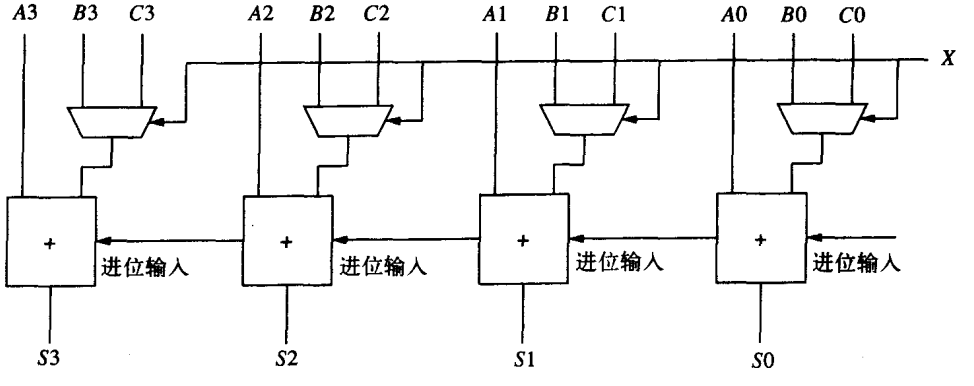


图3-39 结构框图

b. 试设计一个加/减法器。电路能根据X，选择是执行A + B或是A−B。提示：参考图3-39，

并在其基础上实现。

3.25 通常，逻辑结构的执行速度取决于输入和输出端之间的“门”数目。假设AND、OR和NOT门的传输延迟都是一个“门延迟”时间单位。例如，我们说图3-11所示译码器的输入和输出之间延迟时间是2（经过两个逻辑门）。试问：

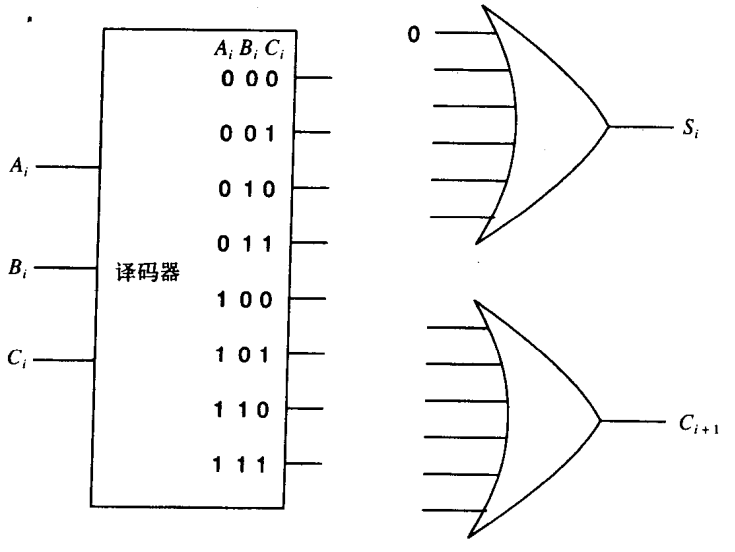
a. 如图3-12，两输入多路开关的传输时延是多少？

b. 如图3-15，1-bit全加器的传输时延是多少？

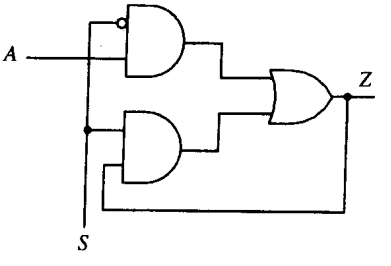
c. 如图3-16，4-bit全加器的传输时延是多少？

d. 假设将4-bit全加器扩展到32位，传输时延又是多少？

3.26 之前介绍过，加法器是由多个独立单bit加法单元（slice）组成的。每个加法单元将A和B两个bit相加，并生成进位（carryin bit）和累加和位（sum bit），我们称之为“全加器”（full adder）。假设我们有1个3-8译码器和2个6输入OR门（如下图所示）。试问，能否基于这3个逻辑单元构建出一个全加器？如果可以，请实现（提示：如果OR门任一输入空缺，可以将其置为输入0，以免产生副作用）。



3.27 看图答题：



a. 选择线$S=0$时，电路输出是多少？或者说，不同的输入A对应的输出Z是多少？

b. 如果将选择线S从0变为1，输出有什么变化？

c. 该逻辑电路是一个存储单元吗？

3.28 假设已有二进制加法器，如果准备设计一个2-bit乘法器。乘法器输入分别是$A[1:0]$和$B[1:0]$，输出是Y（$A[1:0]$和$B[1:0]$相乘的结果），表示为：

$$Y = A[1:0] \times B[1:0]$$

a. $A[1:0]$所能表示的最大值是多少?

b. $B[1:0]$ 所能表示的最大值是多少?

c. Y最大的可能值是多少?

d. 表示Y至少需要多少bit宽度?

e. 画出乘法运算的真值表（4个输入：$A[1]$、$A[0]$、$B[1]$、$B[0]$）。

f. 对照真值表，给出输出$Y[2]$的逻辑实现（只允许使用与、或和非门）。

3.29 假设一个16-bit寄存器原先保存了一个值。如果再向该寄存器写入数值x75A2，试问还能恢复出原先的寄存器内容吗?

3.30 如图3-40和图3-41所示，比较器（comparator）有两个1-bit的输入A和B，3个1-bit输出G（大于）、E（等于）、L（小于）。如果$A>B$，则$G=1$；如果$A=B$，则$E=1$；如果$A<B$，则$L=1$。

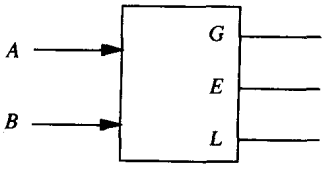


图3-40 习题3.30的结构框图

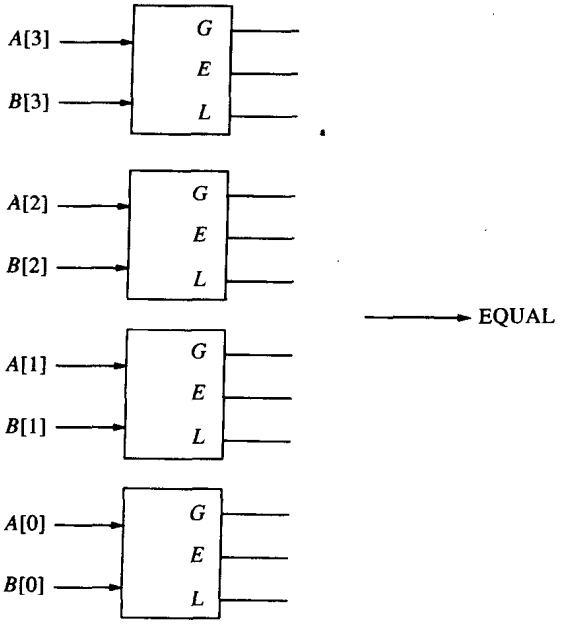


图3-41 习题3.30的结构框图

a. 画出该1-bit比较器对应的真值表。

A	B	G	E	L
0	0			
0	1			
1	0			
1	1			

b. 设计G、E和L的逻辑实现（使用与、或和非门）。

c. 基于1-bit比较器，构建一个4-bit比较器（只判断是否相等）。即如果$A[3:0]=B[3:0]$，则EQUAL输出1。

3.31 如果计算机的寻址能力是8字节，并且需要3-bit访问内存位置，试问该计算机内存的大小是多少（以字节为单位）?

3.32 试解释内存地址（memory address）和内存寻址能力（addressability）之间的区别。

3.33 看图3-21中的$2^2\times 3$-bit内存结构，回答下列问题：

a. 如果读取第4个内存位置（location），则A[1:0]和WE的值分别是多少?

b. 如果将内存单元（entry）数目从4个扩展到60个，问总共需要多少根地址线？扩展之后，内存寻址能力是否发生变化？

c. 如果程序计数器它是CPU中一个特殊的专用寄存器，下一章中将详细介绍它的宽度不小于寻址60个内存位置所需要的地址位数。试问，在不增加程序计数器宽度的情况下，内存还能扩容多少个位置？

3.34 试问，如图3-42所示的内存：

a. 地址空间是多大？

b. 寻址能力是多少？

c. 地址为2的单元内容是什么？

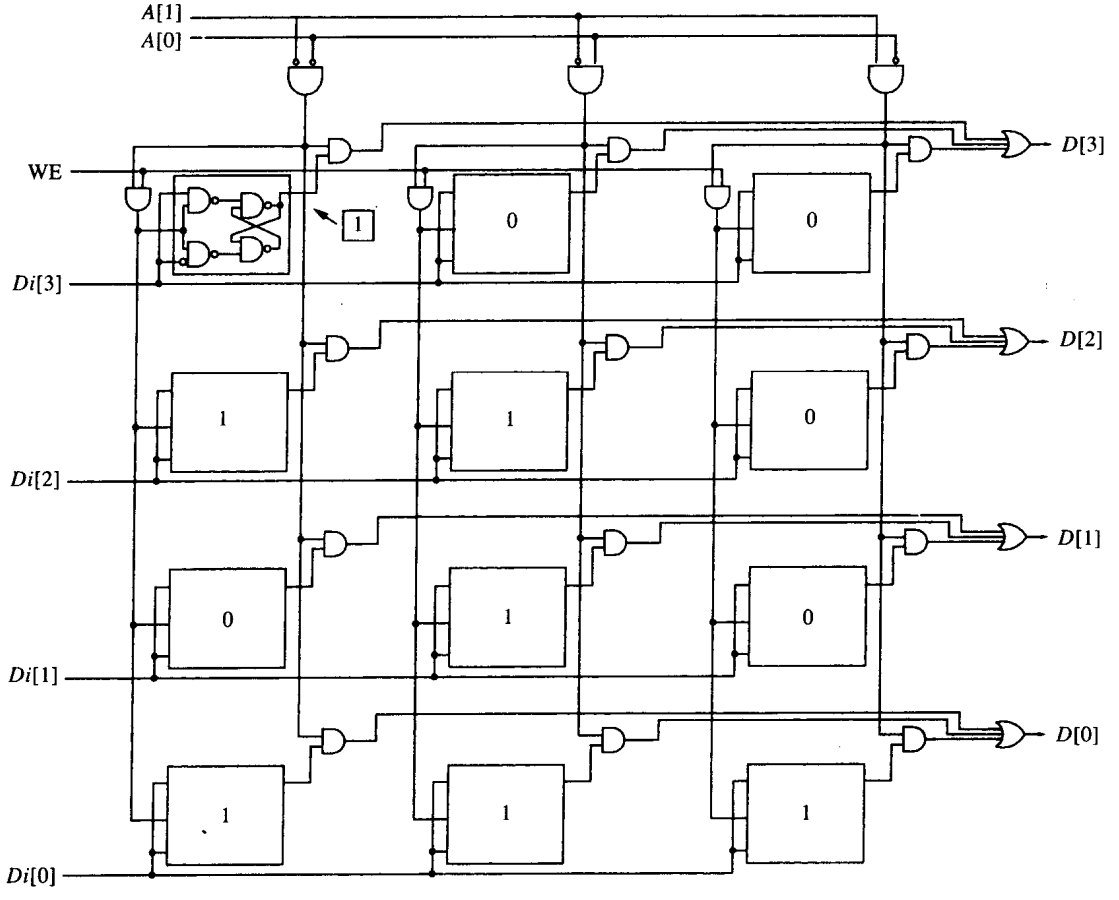


图3-42 习题3.34的内存结构框图

3.35 假设内存是22-bit寻址，且寻址能力为3-bit。试问该内存的总bit容量？

3.36 已知一个两输入组合逻辑。在过去的10个周期内，两个输入的值分别是01、10、11、01、10、11、01、10、11、01。而当前周期中，两个输入值为10。请问之前10个周期的输入值对当前输出是否有影响？

3.37 在3.6.2节中，图3-24a所示的密码锁有4个状态：*A*、*B*、*C*、*D*。即“开锁（*D*）”、“闭锁且无输入（*A*）”、“闭锁且已输入1个正确数字（*B*）”、“闭锁且已输入2个正确数字（*C*）”。这些是密码锁可能出现的所有状态。试问：是否存在第5种状态？

3.38 在3.6.2节的描述中。是否存在状态“TEXAS:OKLAHOMA = 30:28”到状态“比分相同”的状态转移？如果存在，试给出两种状态的记分牌内容（参考图3-25）。

3.39 在3.6.2节描述的篮球比赛中。试问，假设存在两个状态，TEXAS领先“30:28”和平局“30:30”，两者之间是否可能存在转移关系？如果存在这种可能，试画出记分牌予以解释。并标识当时对应的三种可能输出：比赛进行中、得克萨斯队获胜、俄克拉荷马队获胜。

3.40 试画出3.6.2节描述的“三子连珠游戏”的有限状态机。

3.41 在IEEE（美国电气和电子工程师学会）办公楼内，一瓶汽水卖35美分。为方便售货，安装了一个自动售货机。自动售货机内部有一个控制器。自动售货机只接受3种硬币输入：5美分、10美分和25美分硬币。每次用户每投入一枚硬币，就按一下按钮（表示已输入）。如果投入硬币的面值总和不小于35美分，则自动售货机输出一瓶汽水并且找零。请画出该自动售货机的有限状态机，其中每一个状态表示已投入硬币的总额（提示：共有7种状态）。一旦用户投入硬币的总额大于35美分，自动售货机就转移到最后状态，即“输出汽水并找零”（提示：有5种这样的最后状态）。如果已处于最后状态，继续投币将重新开始。

3.42 参考图3-32b。请解释，为什么1号和2号灯会受控于标识为Z的OR门？为什么第2个存储单元的下一状态受控于标识为U的OR门？

3.43 如图3-43所示，有限状态机的输入是X、输出为Z。

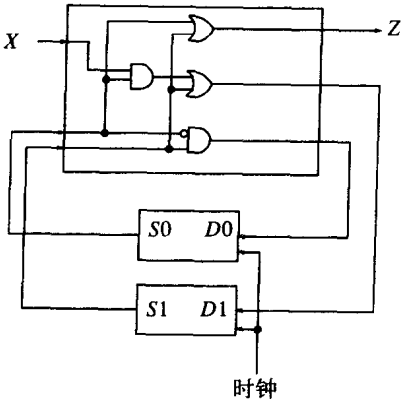


图3-43 对应习题3.43的图

请基于图3-43，回答以下问题：

a. 填写下面的状态转移表。其中：$S0$、$S1$代表当前状态，$D1$、$D0$代表下一状态。

$S1$	$S0$	X	$D1$	$D0$	Z
0	0	0			
0	0	1			
0	1	0			
0	1	1	1	0	1
1	0	0			
1	0	1			
1	1	0			
1	1	1			

b. 画出与状态转移表相对应的状态转移图。

3.44 试证明逻辑门NAND基于自身是逻辑完备的（参考3.3.5节）。换句话说，仅仅使用NAND逻辑就可以实现AND、NOT和OR等逻辑功能。

第4章　冯·诺伊曼模型

下面将进入一个更高的抽象层次。基于第3章所学的决策单元和存储单元，我们可以描述出一个简单的计算机模型，即约翰·冯·诺伊曼（John von Neumann）于1946年提出的“冯·诺伊曼模型”（又称“冯·诺伊曼体系结构”）。

4.1　基本部件

计算机的运行需要两个前提：一是任务描述，即描述计算机所要完成的任务，它以程序（或代码）方式表述；二是计算机本身的运行能力，因为它是任务的具体执行者。

程序是计算机指令的集合，其中每条指令对应计算机的一个基本动作。指令是程序的最小单位（或原子操作），换句话说，一条指令要么完整地执行、要么完全不执行，计算机无法只执行指令的部分功能。

1946年，冯·诺伊曼提出了计算机处理或程序执行的基础模型，图4-1中显示了该模型的基本组成。需要指出的是，该图是我们在冯·诺伊曼那张奇妙的原图上所做的一个润饰修订版。冯·诺伊曼模型包括5个组成部分：*内存*（memory）、*处理单元*（processing unit）、*输入*（input）、*输出*（output）、*控制单元*（control unit）。其中，内存负责存放程序，控制单元负责指令的有序执行。

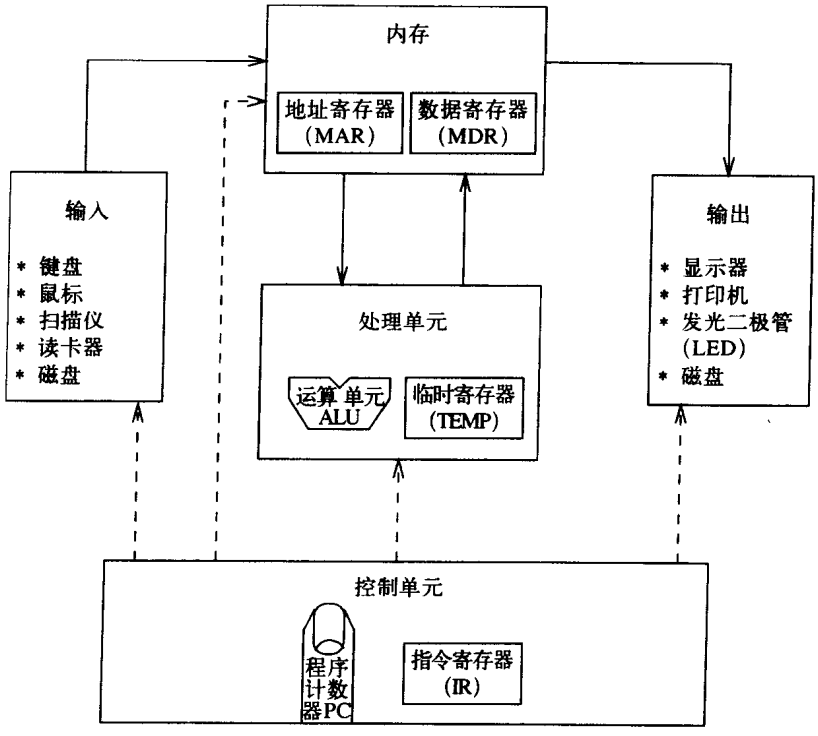


图4-1　冯·诺伊曼模型全局图

下面将详细描述冯·诺伊曼（von Neumann）模型的五个部分。

4.1.1 内存

在第3章中，我们介绍了一个由门电路和锁存器组成的、深度为2^2、宽度为3位（即2^2-by-3-bit）的内存。今天的计算机内存都是2^{28}-by-8-bit模式（2^{28}个存储单元，每个单元可以存储8bit的信息）。换一种专业的说法，我们称该内存的寻址空间为2^{28}，寻址能力为8位，或通常称之为“256兆字节”（256MB）内存。其中256M指的是2^{28}个单元，字节（Byte）指每单元包含8-bit信息。字节是8-bit的又一种简称，类似加仑（gallon）是4夸脱的简称一样。

我们经常说“给定k位，可以表达2^k个单元”。这是因为，要区分和定位2^{28}个不同单元，每个单元必须都具备一个惟一的28位地址。例如第5章中，在LC-3计算机的ISA定义中，LC-3的寻址空间为2^{16}，寻址能力为16位。

访问内存的第一步操作，是向内存提供被访问内存单元的地址（参考第3章）。以读和写操作为例：

- 读操作。在第3章中有关于LC-3的读操作的描述，首先将被访问内存单元的地址放入CPU的内存地址寄存器（Memory Address Register, MAR）；然后发送读信号通知内存。由此，内存将该单元中存放的数据传送至内存数据寄存器（Memory Data Register, MDR）。
- 写操作。首先将被访问内存单元的地址放入MAR寄存器，然后将要写入的数据放入MDR寄存器，最后向内存发送写信号。由此，MDR的内容被写入MAR指向的内存单元。

在结束内存的话题之前，我们再回顾一下“内存单元”的两个重要概念：“单元地址”及其所存储的“单元内容”。图4-2所示是一个包含8个单元的内存，左边数字代表其地址（从0到7），每个单元包含了一个8位信息。其中，地址为4的单元存放的数据内容是6；而地址为6的单元存放的数据内容是4。它们代表的是两种完全不同的情况。

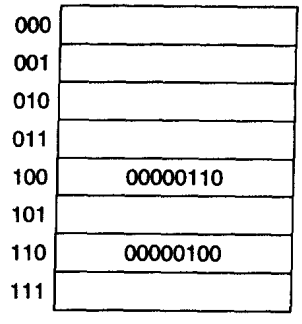


图4-2 单元6存放的是数据4，单元4存放的是数据6

最后，我们以邮局信箱为例，来比喻内存的特性。每个信箱都有一个惟一的编号，编号如同内存单元的地址；而信箱中的信件如同内存单元中存放的数据信息。随着时间的迁移，信箱中的信件在不断地变化，但信箱编号永远不变。内存亦如此，内存每个单元中存放的数据内容会不断地变化，但其地址是不会变化的。

4.1.2 处理单元

处理单元是信息真正被处理的地方。现代计算机的处理单元已发展得非常复杂，它由很多功能单元组成，每个功能单元负责一个功能，如除法操作、求平方根等各种功能单元。其中，ALU是最简单的功能单元（也是冯·诺伊曼模型中惟一的功能单元）。ALU是“算术逻辑运算单元”的缩写。顾名思义，ALU所能完成的功能包括基本运算（如ADD、SUBTRACT）和基本逻辑操作（如按位AND、OR、NOT）等（参见第2章）。在LC-3中，也包含了一个ALU单元，它可以完成ADD、AND和NOT这三个基本操作。

ALU所能处理的量化大小（size of quantity）通常被称为该计算机的“字长”（word length），而量化的基本单位被称为一个“字”（word）。比如LC-3的ALU所能处理的量化大小是16-bit，所以我们称LC-3是一个16位的机器。每类ISA都有自己的字长规定，这取决于该计算机的设计偏好。如今，PC机和工作站（workstation）级别的计算机的字长通常是32位（如Intel公司的奔腾IV）或64位（如Sun公司的SPARC-V9处理器和Intel公司的Itanium处理器）；对那些应用于简单设备的微处理器，8位字长就足够了，如传呼机（pager）、录像机（VCR）、手机或蜂窝电话（cellular

telephone）[⊖]。

通常，在设计中会为ALU在其附近配置少量存储器，以便它存放最近生成的中间计算结果。例如，在（*A*+*B*）·*C*的计算过程中，它可能会将*A*+*B*的结果保存在内存中，但随后与*C*相乘之时，还要再次从内存中读出。显然，相比ADD和MULTIPLY两次运算操作，两次内存访问占据了太多的时间。于是，几乎所有的计算机都为ALU配备了临时存储空间，来存放*A*+*B*的结果，以避免之后的乘法操作引发不必要的内存访问。临时存储器最常见的设计方式就是一组寄存器（见3.4.3节），其中每个寄存器的宽度应该与ALU处理数据的宽度一致，我们说每个寄存器存放了一个字。在LC-3中，有8个这样的寄存器（R0,R1,…，R7），每个的宽度是16-bit。相比之下，SPARC-V9指令集结构则有32个寄存器（R0,R1,…，R31），每个的宽度是64-bit。

4.1.3 输入和输出单元

信息能够被计算机处理的前提是“信息必须事先输入计算机”。同样，信息的处理结果也必须通过显示、打印等方式输出至计算机外部，我们称相关的设备为输入/输出（Input and Output）设备。输入/输出设备的种类很多，在计算机术语中，我们通常称它们为“外围设备”（peripheral）。之所以这么称呼，是因为它们通常是计算机的“附属设备”（accessory），即它们是可有可无的。

LC-3只提供了两个基本的输入/输出设备，其中，输入设备是键盘，输出设备是显示器。

真实的计算机中，还包含很多其他设备，如输入设备包括鼠标、数字扫描仪、软盘等，输出设备包括打印机、LED显示器、磁盘等。在早期，计算机输入和输出是依赖于穿孔卡片（punched card），人们通常要拖动成箱的卡片，但那个时代已经完全过去了。

4.1.4 控制单元

控制单元所扮演的角色如同乐队的指挥，用它来控制其他所有单元之间的协同工作。正如后面我们将看到的那样，在计算机程序的逐步执行过程中，它既负责控制程序执行过程的每一步，又负责控制其中每条指令执行过程的每一步。

控制单元中有几个特殊的寄存器，一是指令寄存器（instruction register），保存的是正在被执行的那条指令；二是PC寄存器（Program Counter, PC），用来指示下一条待处理的指令，控制单元专门准备了这个寄存器。由于历史的原因，该寄存器被命名为“程序计数器”（即PC寄存器），但它更合适的称法应该是“指令指针”（instruction pointer），因为该寄存器的内容实际上是指向下一条待处理指令的地址。有趣的是，Intel公司就是这么命名的（在Intel处理器手册中，该寄存器的简称为IP），而这么好的一个命名，却不被其他处理器所采用。

4.2 LC-3：一台冯·诺伊曼机器

为了将第3章的数据通路（如图3-33所示）以及4.1节介绍的各种结构（structure）有机地结合起来，并衔接在第5章中将展开的LC-3设计细节，本节将概述LC-3的全貌，并将它与冯·诺伊曼模型进行对比。图4-3包括了LC-3的完整数据通路（如图3-33所示），但删除了那些虽然已被实现，但不属于冯·诺伊曼模型所定义5个基本部分的LC-3实现。

注意图4-3中的两种箭头，即实心箭头和空心箭头。实心箭头代表该通路中传送的是数据（被控对象），空心箭头则代表的是控制信号。例如，ALU模块有两个16位的数据输入和一个输出结果，由于输入和输出都是数据，所以它们的标示都是实心箭头；而信号ALUK控制着ALU模块的运算类

⊖ 随着时间的迁移，这种说法也部分过时了，如现在Sony公司的DV+VCR以及大部分手机，采用的都是32位处理器，其性能要求越来越高，甚至采用双核或多核芯片。——译者注

型，属于控制信号，所以它的箭头标示是空心的。

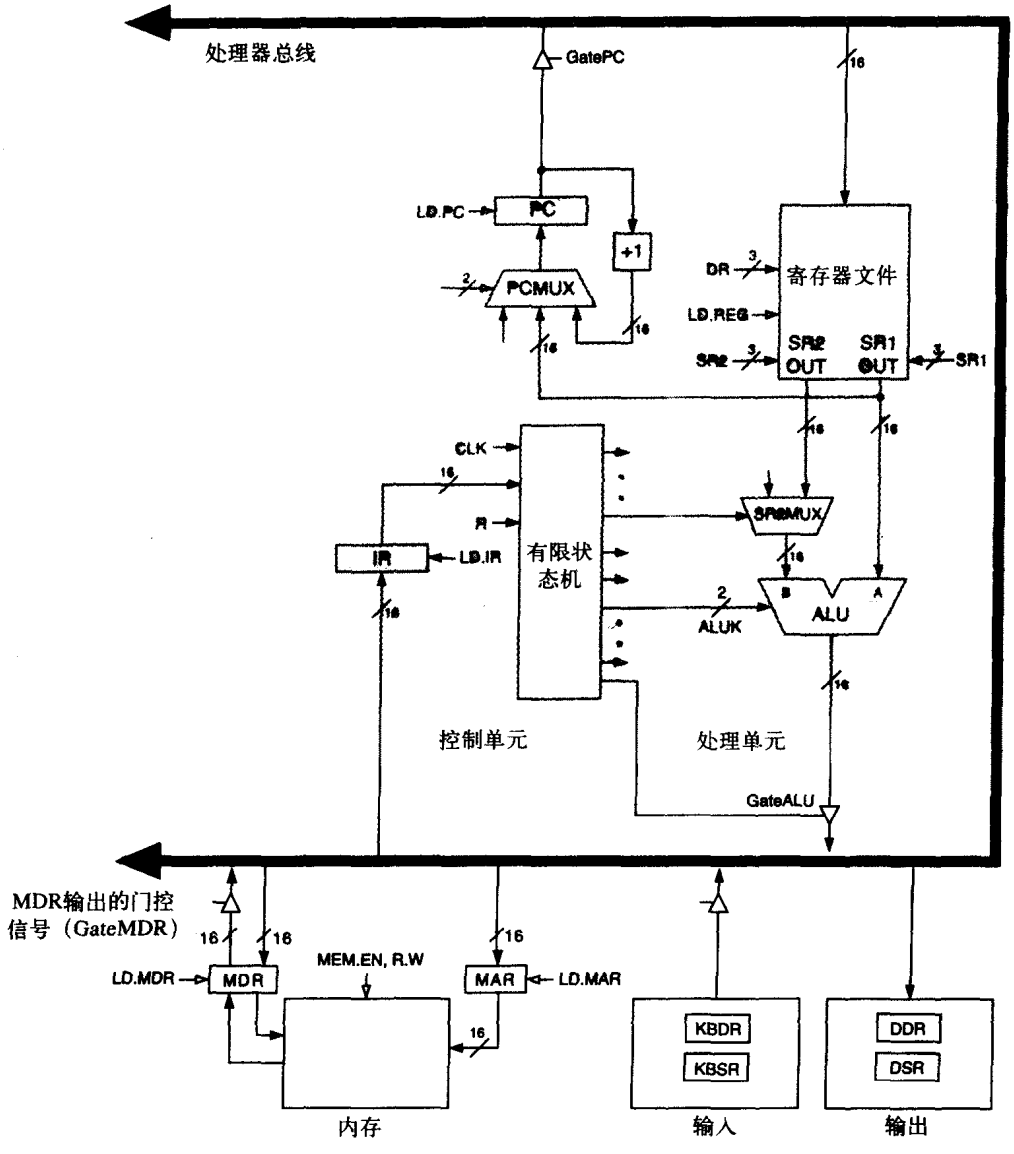


图4-3 LC-3——一个冯·诺伊曼模型的实例

下面描述图4-3的各个主要组成部分：

- 内存（MEMORY）：包括存储单元，以及MAR和MDR寄存器。其中，MAR是16位宽度，意味着LC-3的最大可寻址空间是2^{16}。MDR也是16位，意味着每个存储单元存储了16个bit的信息，或称LC-3是16位可寻址的（addressable）。
- 输入/输出：即键盘和显示器。最简单的键盘需要两个寄存器：一是数据寄存器KBDR，保存键入的ASCII码；二是状态寄存器KBSR，保存键盘敲击时的状态信息。同样，最简单的显示器也需要两个相似的寄存器：一是数据寄存器DDR，保存将要显示字符的ASCII码；二是状态寄存器DSR，保存相关的状态信息。有关这些寄存器的细节，将在第8章讨论。
- 处理单元：包括一个算术逻辑运算单元（ALU）和8个寄存器（R0~R7）。其中，R0~R7作为临时寄存器，用以存放近期将被后面指令使用的操作数。而LC-3的ALU只实现了一种运算

操作（ADD）和两种逻辑操作（按位AND和NOT）。

- 控制单元：包含所有与信息处理控制相关的结构。其中最重要的一个结构就是有限状态机（Finite State Machine, FSM），它控制着系统中的所有活动（有关“有限状态机”，请回顾3.6节）。运算或处理操作是一步一步完成的，具体地说，是一个接一个时钟周期有序操作的（clock cycle by clock cycle）。与有限状态机模块相关的输入/输出如图4-3所示：（1）时钟输入CLK，它定义了每个时钟周期持续的长短；（2）指令寄存器IR，是有限状态机的一个重要输入。由于“输入决定了过程”，所以不同的指令即意味着完全不同的处理流程；（3）程序计数器PC，也是有限状态机的另一个重要输入，它指向当前指令之后下一个要执行指令的位置。

值得注意的是，有限状态机的所有对外输出都是空心箭头，即它们都是控制信号，控制着整个计算机系统的全程活动。例如，输出信号ALUK（2位）控制着ALU当前周期的操作选择（ADD或AND或NOT）；另一个输出GateALU，则决定了ALU的输出在当前周期是否输送至处理器总线。

附录C提供了关于LC-3数据通路、控制信号和有限状态机的更详尽描述。

4.3 指令处理

冯·诺伊曼模型的核心思想是：程序和数据都是以bit流的方式存放在计算机内存中[⊖]，程序在控制单元的控制下，依次完成指令的读取和执行。

4.3.1 指令

指令是计算机执行的最小单位。指令本身又由操作码和操作数两部分组成。操作码表示该指令是做什么的，操作数表示该次操作的对象是哪些。在第5章中我们将介绍，LC-3的指令长度是16位（一个字长），将这16位从左至右依次编号为bit[15]~bit[0]，那么bit[15:12]代表的就是操作码（同时也意味着该字段最多可表达2^4个操作类型），bit[11:0]则对应操作数的表示。

例4-1 ADD指令。

ADD指令需要3个操作数：两个相加数（源操作数）及一个求和结果（目的操作数）。我们曾提到，LC-3有8个用于临时存储的寄存器（R0~R7），ADD指令要求它的两个源操作数中，至少有一个是这些寄存器之一[⊜]，结果也是存放在寄存器中的。由于总的寄存器数目是8，所以指令字段中，代表每个源或目的寄存器的位数是3位[⊝]。据此，LC-3的ADD指令格式（format）如下：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0
ADD				R6			R2						R6		

其中：

bit[15:12] = 0001，代表ADD操作。

bit[11:9] = 110，代表目的寄存器R6。

bit[8:6] = 010，bit[2:0] = 110，分别代表源寄存器R2和R6。

bit[5:3]所代表的含义在本例中暂不做解释，我们将在5.2节介绍它。

上述指令的编码解释为“将寄存器2（R2）和寄存器6（R6）的内容相加，然后将求和的结果存储回寄存器6（R6）”。

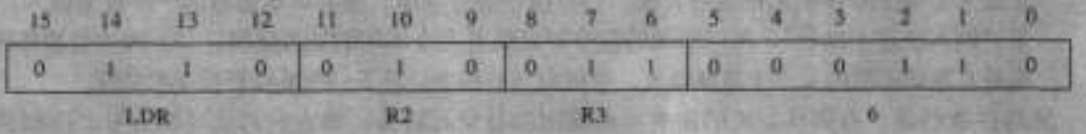
⊖ 换句话说，从存储角度来看，指令和数据之间没有任何区别。——译者注

⊜ 请思考：为什么操作数大多是寄存器而不是内存单元？——译者注

⊝ 请思考：如果操作数是指向内存地址的，则该字段需要占用多少位？——译者注

例4-2 LDR指令。

LDR指令需要两个操作数。LD代表“load”，按照计算机语言（computerese），其意思是“从内存某个地方读取其中的内容，并将其内容存入某寄存器中”。两个操作数分别对应内存地址和目的寄存器。LDR中的“R”代表计算内存读取地址的机制（或称为寻址模式），R代表的是“base+offset”模式。参考LC-3的LDR指令格式：



LDR指令的4-bit操作码是“0110”。

bit[11:9] = 010，代表内存读入数据所存放的目的寄存器R2。

bit[8:0] = 011，代表偏移寄存器R3。

bit[5:0] = 000110，代表基址值6。

bit[8:6]和bit[5:0]的内容与内存地址计算有关。LC-3只支持“基址加偏移”（BASE+offset）一种寻址模式，其计算方法是：将bit[8:6]指定的寄存器的内容和bit[5:0]中的补码数值相加，求得最终地址。因而，本例的解读是“将R3的内容和立即数6相加，求得内存地址；然后将该地址指向的内存单元内容读入，并存放在R2寄存器中”。

4.3.2 指令周期

指令的处理过程是在控制单元的控制下，精确地、一步一步地完成的。我们称这个执行的步骤顺序为指令周期（instruction cycle），其中的每一步称为节拍（phase）。一个指令周期包括6个节拍，大多数计算机的节拍设计都如此（6个节拍），但不是所有的都如此。稍后我们将讨论这个问题。

首先，我们看一下指令周期的6个节拍，即FETCH（取指令）、DECODE（译码）、EVALUATE ADDRESS（地址计算）、FETCH OPERAND（取操作数）、EXECUTE（执行）、STORE RESULT（存放结果）。下面将以图4-3所示的LC-3数据通路为背景，详细介绍每个节拍的操作细节。

1. 取指令

该节拍负责从内存中读取下一条待执行的指令，并将其装入控制单元的指令寄存器IR。顺便回顾一下，一个计算机程序是一组指令的集合，而其中每条指令又表示为一个bit序列集合。在冯·诺伊曼模型中指出，程序执行时整个程序的所有指令必须都已保存在内存中。所以要获取下一条指令，前提是知道它在内存中的准确位置。程序计数器PC负责的就是这个任务，它记录着下一条指令在内存中的地址。

FETCH节拍的具体工作描述如下：

(1) 将PC寄存器的内容装入（load）MAR寄存器。

(2) 该地址对应内存单元的内容（即下一条指令）被装入MDR。

(3) 控制单元将MDR内容装入IR寄存器。

之后，指令就进入译码节拍。但是，由于当前指令周期完成之后还将读入下一条指令，我们希望此时PC内容应修改为是指向下一条指令的地址。所以，在当前FETCH节拍还需要补充一个动作，即修改PC寄存器的内容，我们称之为“PC增量”操作。这样一来，在下一条指令周期的FETCH节拍，下一个内存单元中的指令将被自动装入IR寄存器（只要当前指令不去修改PC的内容）。

以上所有操作都在控制单元掌控之下，这如同我们之前的比喻：乐队的乐器都在乐队指挥的掌控之下（虽然每个乐器是由具体的演奏者所操作）。乐队指挥的每一个动作（stroke）代表一个机器周期（machine cycle），其中一个机器周期对应一个时钟周期（clock cycle，参见4.4.1节）。事

实上，我们对机器周期和时钟周期这两个概念并不做严格区分。

分析如上操作：第一步MAR内容装载（load）将花费1个机器周期；第二步内存访问则花费1个或更多的机器周期，因为它取决于计算机内存访问的速度；第三步MDR和IR的内容拷贝只需1个机器周期。一个机器周期具体是多长时间呢？对于现代计算机来说，这个时间"极其短"，对于主频为3.3GHz的奔腾IV处理器，每秒将完成33亿（3.3billion）机器周期（或时钟周期），即每个机器周期的时间长度只有0.303ns（nanosecond）。对比一下，你头顶的灯泡正在以每秒60次的速率闪烁着[⊖]，这意味着灯泡闪烁一次的瞬间，计算机已完成了5 500万次机器周期！

2. 译码

译码操作的任务是分析、检查指令的类型，并确定对应的微结构操作细节。如第3章所述，LC-3采用"4-16"译码器识别16种不同指令，也就是以IR[15:12]的4位操作码字段为输入，输出16根使能线，但任何时候16根使能线中有且仅有一根是有效的（对应16种不同指令）。同时，基于不同的使能线，剩余的12bit指令信息的解释含义也有所不同。

3. 地址计算

如果指令执行时存在地址计算操作，则在此节拍完成。以例4-2的LDR指令为例，LDR指令应将特定地址内存单元的内容读入寄存器R3，则地址值应该等于R3内容和立即数6之和，该求和操作就是在地址计算（EVALUATE ADDRESS）节拍完成的。

4. 取操作数

该节拍负责读取指令处理所需要的源操作数。仍以LDR指令为例，该节拍包括两个子步骤：(1) 将之前地址计算节拍算出的地址值装入MAR寄存器；(2) 从MDR寄存器获取读自内存的源操作数。

再以例4-1的ADD指令为例，该指令的源操作数分别来自R2和R6寄存器，它们全部来自寄存器而没有内存操作。顺便提一下，在当前的大多数微处理器中，该节拍可以和指令译码节拍同时执行，这类指令的加速处理技巧是个相当吸引人的话题，但受制于篇幅和本书层次，我们不在此展开讨论。

5. 执行

该节拍负责指令的执行操作，不同的操作码在该节拍的操作也多半不同。如ADD指令，本节拍在ALU中执行的就是一个单拍加法操作。

6. 存放结果

这是指令执行的最后节拍。来自之前节拍的执行结果将被写入目的寄存器。

在最后一个节拍（STORE RESULT）完成之后，控制单元复位指令周期，即从"取指令"节拍重新开始。PC寄存器在上个指令周期已自动修改，即已指向下一条指令所在的地址。由于前后指令在存储空间上的存放是顺序的，因而我们又称计算机的执行方式是"顺序执行"，如此执行，直到该顺序流（sequential flow）被打断。

例4-3 ADD [eax], edx。

这是Intel x86指令的例子，该指令周期需要6个节拍完成。

- 同所有指令一样，前两个节拍都是相同的且必需的（即FETCH和DECODE）；
- 基于eax寄存器的内容计算内存地址（EVALUATE ADDRESS）；
- 该地址的内存内容被读入（FETCH OPERAND）；
- 然后将该内容与edx寄存器的内容相加（EXECUTE）；
- 最后将相加的结果写回内存单元，即原先提供第一个源操作数的内存单元（STORE RESULT）。

⊖ 美国的交流电频率是60Hz，而中国的是50Hz。——译者注

例4-4 LC-3的ADD和LDR指令及其节拍数。

LC-3的ADD和LDR指令并不需要全部的6个节拍。如ADD指令不需要地址计算（EVALUATE ADDRESS）节拍，而LDR指令则不需要执行（EXECUTE）节拍。

4.4 改变执行顺序

之前我们所描述的都是顺序的程序执行过程，即第一条指令执行之后，接着就是执行第二条指令，随后就是第三条，如此继续。其中包含的指令类型也仅有两种，即运算指令（operate instruction）和数据搬移指令（data movement instruction）。如ADD指令属于操作类指令，LDR指令则属于数据搬移类指令（即将数据从一个地方搬到另一个地方）。当然有些指令，它们既是操作类也是数据搬移类的（在第5章有关LC-3的介绍中，你会发现这样的指令）。

下面介绍第三类指令，即控制指令（control instruction），其作用是改变程序的执行顺序。例如，如果我们希望循环反复地执行一组指令，即先是第1条、第2条、第3条地执行一遍，然后回过头来，第二遍再次执行第1、2、3条指令，再然后是第3遍，如此反复。如我们所知，任意的指令周期都是从将PC的内容装载到MAR开始的，因而，如果我们希望改变指令执行的顺序，就必须在一条指令的取指令节拍，即把PC加1操作之后与下一条指令的取指令节拍开始之前的这段期间内，修改程序计数器PC的值。控制指令即在指令的执行节拍来修改PC的内容，覆盖之前取指令节拍时PC加1后的值。该控制指令执行完成后，在下一个指令周期开始时，计算机访问PC获得的地址值，是由该控制指令在执行节拍换入的新地址值，而不是之前PC加1后的地址值，从而实现了改变程序的执行顺序的目的。

例4-5 JMP指令。

如下所示是LC-3的JMP指令，假设该指令存放在内存地址x36A2处[⊖]。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0
JMP							R3								

其中：

bit[15:12] = 1100，代表JMP指令操作码；

bit[8:6] = 011，代表R3寄存器，即其中存放了下一条指令地址的寄存器；

所以这条指令的译码结果是“将R3寄存器的内容装入PC寄存器（在执行节拍）”。换句话说，下一条要执行的指令地址来自R3寄存器。

具体处理过程的描述如下：首先在该指令执行之初，PC内容等于x36A2，于是，在取指令（FETCH）节拍整条JMP指令被读入IR寄存器，同时PC自动覆盖为x36A3，假设之前R3内容等于x5446，那么在执行节拍（EXECUTE）PC内容就被更换为x5446。这就导致了在下一个指令周期，被处理的指令来自地址x5446（而不是x36A3）。

指令周期的控制

之前我们介绍过，一个指令周期包括6个节拍（phase），而每个节拍又分解为若干个子步骤（step）。如取指令节拍（FETCH），它需要三个步骤顺序执行（而不可并行）才能完成，一是将PC内容装入MAR寄存器，二是读内存，三是将MDR寄存器内容装入IR寄存器。所有这些取指令节拍的子步骤，完全受控于控制单元中的有限状态机。

⊖ 本书作者表示十六进制数的方式是省略前缀“0”，如写成“x36A2”而不是“0x36A2”。——译者注

图4-4所示是一个有限状态机的状态转移简图，显示的是状态机在一个指令周期内各个节拍的状态。如3.6节介绍的有限状态机示例，每个状态对应一个时钟周期内的活动，每个状态（或节点）的具体操作行为如节点方框内所描述，连接线表示从一个状态至下一状态的迁移。

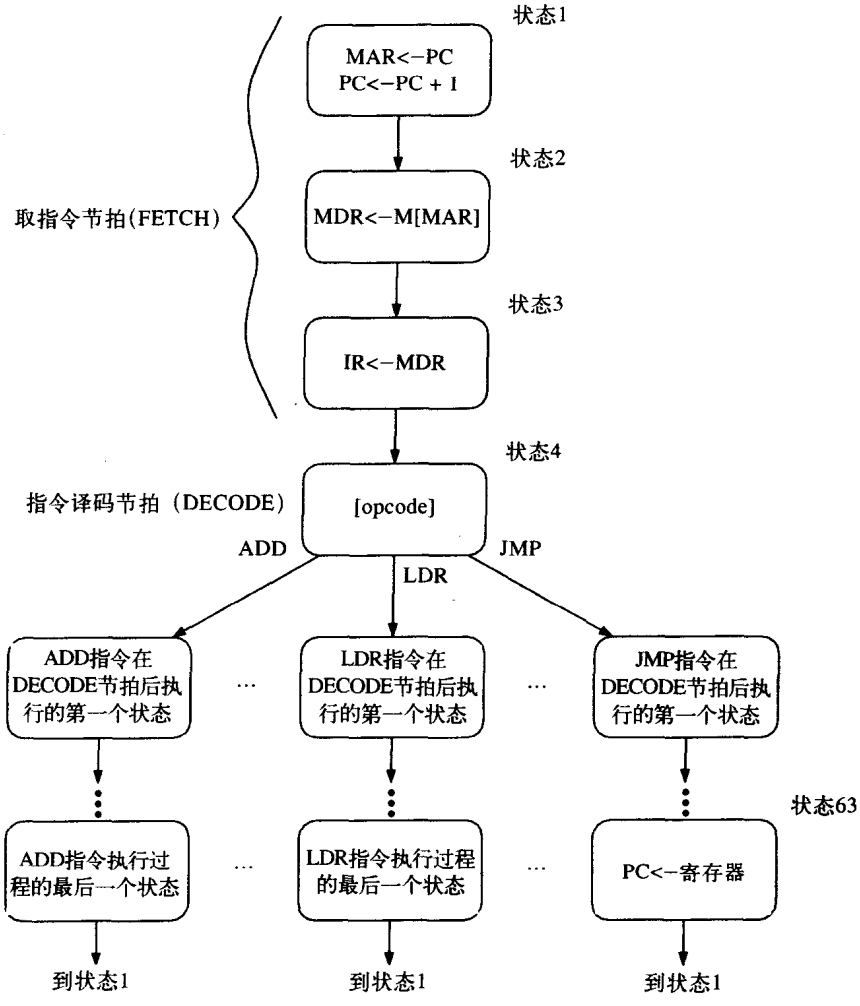


图4-4 LC-3有限状态机简图

取指令节拍需要3个时钟周期：

（1）有限状态机的起始状态是“状态1”，也是指令周期的第1个时钟。本周期内将同时完成“将PC内容装入MAR”和“PC自动增量”两个操作。操作一：为将PC内容装入MAR寄存器（如图4-3所示），有限状态机必须控制GatePC和LD.MAR两个信号。GatePC控制着PC与处理器总线（processor bus）之间的通路，LD.MAR是MAR的可写信号，它确保当前周期结束时，将总线数据“锁”入MAR寄存器（锁存器总是在时钟周期结束时，将输入端数据锁入）。操作二：PC增量操作，如图4-3所示，有限状态机通过PCMUX信号控制标示为“+1”的功能单元输出，而LD.PC信号则控制着PC的输入（或是PCMUX开关单元的输出）。⊖

（2）第2个时钟开始，有限状态机进入“状态2”。在这个周期内完成“读内存”操作，并将内存返回数据（确切地说，这个节拍返回的实际上是“指令”）自动装入MDR。

⊖ 图中PCMUX信号的标示被省略，PCMUX信号即指PCMUX开关单元左侧的2线输入。——译者注

(3) 状态3，将MDR中的数据拷贝至指令寄存器（IR）。有限状态机为此控制GateMDR和LD.IR等两个信号，即在周期结束时，将MDR输出锁入IR。至此，完成全部取指令操作。

译码节拍只需要1个时钟周期。

(4) 在状态4中，基于IR寄存器的输入，尤其是该指令的操作码部分（IR[15:12]），有限状态机的转移出现不同的分支去向。

之后，有限状态机继续，直到整个指令完成，并最后返回到状态1。我们称这个过程为“状态转移”。不同指令之间的差异，主要体现在取指令和译码节拍之后的这些操作上，它们的实现是不同的。

有时候我们希望跳到别的地方去获取下一条要执行的指令（而不是空间顺序上的下一条），我们称这种能够改变程序执行顺序的指令为“控制指令”。控制指令的实现机制非常简单，即在执行节拍修改PC内容即可，如图4-4中的状态63所示。

附录C详细描述了LC-3的设计实现，其中包括了完整的状态图和数据通路。而本章将不深入细节，我们此时的目标只是想告诉你，指令周期级的处理并非那么神奇，完整的状态图描述应该是时钟级别的，能够控制整个指令周期中每一个节拍的动作。由于每个指令周期结束后，都返回到“状态1”，由此可以推论：通过一个周期接一个周期的操作，最终可以完成整个程序（即指令集合）的执行。

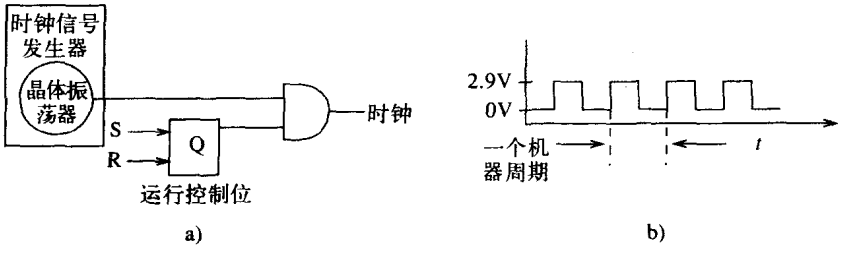


图4-5 时钟电路及其控制部件

4.5 停机操作

之前所描述的一切，都给我们这样一种感觉，即计算机好像永远不会停止！因为它会不停地要求“获取下一条指令”，然后一个周期接一个周期地执行，真令人无法忍受！计算机是永不疲劳的，难道它非得这样永远运行下去，直到有人拔了它的电源插头才能停止？

对于用户程序来说，它通常是受控于操作系统而执行的，如UNIX、DOS、MacOS或Windows NT等，都属于操作系统，即操作系统有办法终止一个用户程序的运行。但是，操作系统本身也是一个计算机程序。另外，对于计算机来说，用户程序和操作系统程序两者之间并没有区别。即它们都是一堆指令的集合，计算机要做的事情，就是一条指令接一条指令地执行。

总之，程序的停止问题对于用户程序来说很好办，只需要在用户程序的结束处“埋放”一条控制指令，该控制指令的任务就是修改PC寄存器，使之跳回操作系统，而操作系统通常也将借助这个时刻启动一个新的用户程序。

但是，从指令层次来看，计算机仍然是“永远动着的”（要么是执行用户程序，要么是执行操作系统的指令）。如果我们希望停止这个无穷的指令周期序列，该怎么做呢？再以乐队演奏为例，计算机的运行相当于指挥棒在以每秒百万次的速率挥动，停止乐队演奏的关键是停止指挥棒。那么谁是计算机中的“指挥棒”呢？答案是“**时钟**”（clock）。时钟就是机器周期，一个机器周期的开始，在当前节拍中意味着下一个操作步骤（next step）的开始，或在当前指令周期中意味着下一节拍的开始。因而，终止了时钟，就意味着终止了指令的执行！

时钟电路的结构如图4-5a所示，主要包括时钟信号发生器（clock generator）和RUN状态开

关[⊖]等。其中：

(1) 时钟信号发生器的主要部件是晶体振荡器 (crystal oscillator，简称“晶振”)，它是一种压电类设备 (piezoelectric device)，即输入“电”，就能转换出“压力”(产生振荡)。我们只需要将“晶振”理解为一个能够产生振荡电压 (如图4-5b所示) 的黑箱 (参见1.4节之“黑箱”定义)，这个振荡电压就等价于指挥手中的指挥棒，在一个机器周期中，前半周电压保持为2.9V，后半周落至0V。

(2) RUN锁存器的状态或输出如果是1 (即Q=1)，则时钟电路的输出和时钟信号发生器的输出一致 (透明输出)；如果RUN部件的输出是0 (Q=0)，则时钟信号发生器到时钟电路的通路被切断，即时钟电路输出为0。

因此，如果要终止指令周期，只需“清零”RUN状态即可。每个计算机都具备类似的机制，在一些老机器上，还提供了HALT指令来完成这个功能。LC-3的做法，如同大多数机器一样，是由操作系统控制完成的 (参考第9章)。

思考题：如果HALT指令能够清零RUN锁存器，则停止指令时钟。那么设计一个什么样的指令，能够通过设置RUN锁存器重启指令周期呢？

4.6 习题

4.1 写出冯·诺伊曼模型的5个组成部分，并写出各个部件的功能和目的。

4.2 简要描述一下内存和处理单元的接口，即内存和处理单元的通信方式。

4.3 *程序计数器* (PC) 的命名可能存在什么误导？为什么说*指令指针* (Instruction Pointer, IP) 的命名更加合适？

4.4 请解释一个计算机的“字长” (word length) 的定义。字长对计算机的工作有什么影响？基于第1章中的知识，试判断以下论断是否正确：字的长度越大，处理的信息越多，因而计算能力[⊜]也更强。

4.5 如下表格代表的是一个很小的内存。请基于以下表格回答问题。

地址			数值	
0000	0001	1110	0100	0011
0001	1111	0000	0010	0101
0010	0110	1111	0000	0001
0011	0000	0000	0000	0000
0100	0000	0000	0110	0101
0101	0000	0000	0000	0110
0110	1111	1110	1101	0011
0111	0000	0110	1101	1001

a. 地址为3的内存单元存放的数值是多少？地址6呢？

b. 每个地址单元的二进制码数值可以有多种解释方法。之前我们学过，它可以是一个无符号数、有符号补码或是浮点数等。

1) 分别写出地址0和地址1数值所对应的补码数；

2) 写出地址4对应的ASCII码；

3) 写出地址6和7对应的IEEE浮点数值 (6和7合起来，表示一个32位浮点数)；

4) 分别写出地址0和地址1对应的无符号整数值。

c. 在冯·诺伊曼模型中，一个内存单元中的数值既可能代表数据也可能代表指令，如果将地

⊖ RUN状态开关，原文为“RUN latch”，表示它的实现方式是一个单bit锁存器，可置位 (set) 或清零 (clear)。——译者注

⊜ 这里说到的“能力”是指所能做的事情，而不是计算速度。——译者注

址0的内容看做是一条指令，问该指令是什么（LC-3）？

d. 一个二进制数还可以表示为一个内存地址。假设地址5单元存放的是一个内存地址，那么它指向的单元地址是多少？该地址中存放的数值又是多少？

4.6 一条指令有哪两个主要的组成部分？试简述这两个组成部分的内容和作用。

4.7 假设一个32位指令的格式如下：

OPCODE	SR	DR	IMM

如果存在60种操作码和32个寄存器，试问立即数部分（IMM）可表达的范围是多大（IMM的编码方式为补码）？

4.8 假设一个32位指令的格式如下：

OPCODE	DR	SR1	SR2	UNUSED

如果操作码的数目是225，寄存器数目是120，试问：

a. 指令中表达操作码部分所需要的最小位数是多少？

b. 表达目的寄存器的最小位数是多少？

c. 表达UNUSED部分可用的最大位数是多少？

4.9 指令周期中取指令节拍要完成两件重要任务，其一是将下面要处理的指令读入指令寄存器，试问另一个任务是什么？

4.10 例4-1、4-2和4-5分别描述了ADD、LDR、JMP三个指令的处理过程。在整个指令周期中，对于不同的指令，在不同的节拍会对寄存器PC、IR、MAR和MDR做出不同的修改，请填写下表（在空格中填入对应的指令操作码）。

	取指令	译码	取数据	计算机地址	执行	存放结果
PC						
IR						
MAR						
MDR						

4.11 说明指令周期的各个节拍，并简述各节拍中发生的各种操作。

4.12 以ADD、LDR、JMP指令为例，分别写出它们的指令周期中各节拍的操作。

4.13 假设读、写内存操作需要100个周期，而读写寄存器和其他节拍操作只需要1个周期。试计算IA-32指令“ADD [eax],edx”（参考例4-3）和LC-3指令“ADD R6,R2,R6”所需要的执行周期数。

4.14 试描述JMP指令的执行情况（假设R3内容为x369C，参考例4-5）。

4.15 如果HALT指令可以清除RUN锁存器，即停止指令周期。试问什么指令可以设置RUN锁存器，即重新启动指令周期？

4.16 a. 假设机器周期的长度是2ns（即2×10^{-9}秒），问每秒钟能产生多少个机器周期？

b. 如果平均每条指令需要8个周期，且计算机每次处理一条指令，问该计算机每秒能处理多少条指令？

c. 为提高计算机每秒执行指令的数目，现代计算机中采用了很多方法。方法之一是类似生产流水线（或装配线）的方法，即指令的每个节拍都被分割成相对独立的一个或多个逻辑部件，每个节拍接着上一个节拍（同时也是上一个周期）的工作继续做。基于这种模式，每个机器周期都可以从内存读入一条新指令，然后在取指令周期结束时，将其传递给后面的译码节拍；而在下一个机器周期中，该指令在被译码的同时，下一条指令又可以被读入。这就是所谓的“装配线”（assembly line）。假设指令是有序地存放在内存中的，且没有任何事情打断这个顺序执行，那么这个“装配线”每秒钟能执行多少条指令？（在以后的高级课程中，在专业上称这种装配线为“流水线——pipeline”，届时还将研究是什么原因使得流水线不能顺利“流动”。）

第5章 LC-3结构

在第4章中，我们讨论了计算机的基本组成，它由以下单元组成：(1) 内存；(2) 处理单元及相关的临时存储（通常指寄存器组）；(3) 输入输出设备；(4) 控制所有单元活动的控制单元（包括控制单元本身）。另外，我们还学习了指令周期的6个节拍：FETCH（取指令）、DECODE（译码）、ADDRESS EVALUATION（地址计算）、OPERAND FETCH（取操作数）、EXECUTE（执行）、STORE RESULT（存放结果）。下面我们将介绍一个“真实的”计算机LC-3，更确切地说，是介绍LC-3的指令集结构（ISA）。之前你应该已听到我们不停地提到LC-3以及它的部分指令，下面将对它进行更系统的描述。

在第1章中，我们描述了这样一个概念，即ISA是软件命令和硬件执行体之间的接口。在本章以及第8、9章中，我们将以LC-3为描述对象，重点讲述ISA的重要作用和特性，你在使用LC-3自己的语言（即LC-3机器语言）编写程序时，将用到这些重要特性。

有关LC-3 ISA的详细描述，请参考附录A。

5.1 ISA概述

ISA定义了软件编程所需要的必要而完整的描述。换句话说，ISA向以机器语言编程的程序员提供有关控制机器所需要的所有必要信息。另外，面向专业的系统编程ISA还将定义更详细的机器相关信息。如高级语言（如C、Pascal、Fortran、COBOL）编译器的开发者需要知道一些“窍门”，才能将高级语言翻译成高效的机器语言。

ISA给出了内存组织方式、寄存器组、指令集（包括操作码、数据类型、寻址模式）等信息。

5.1.1 内存组织

LC-3的可寻址空间大小是2^{16}（或65536），寻址（即读写）基本单位是16位。65536大小的空间并不是全部用于内存，在第8章中将讨论这个问题。由于LC-3中数据处理的基本单位是16位，所以我们在此称这16位为一个“字”（word），同时我们也称LC-3是一个“字寻址”（word-addressable）机器。

5.1.2 寄存器

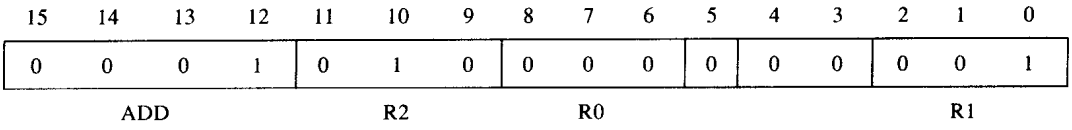
由于从内存中获取数据的速度很慢（不止一个周期），所以LC-3和大多数的机器一样，还提供了临时存储空间，它们的访问速度是一个周期。临时存储空间的最常见实现方式是寄存器，LC-3中提供了一组通用寄存器，其中的每个寄存器称为通用寄存器（General Purpose Register, GPR）。

寄存器和内存的特性是一样的：

(1) 记忆特性。即可以存储信息并可被再次读出。每个寄存器的存储大小是一个“字”，在LC-3中即为16位。

(2) 独立寻址。每个寄存器必须有独立的惟一标识。LC-3中提供了8个GPR（即意味着标识编号需要3位就足够了），用符号表示的话，就是R0，R1，…，R7，图5-1所示是LC-3寄存器组，有时我们又称之为寄存器文件，其中R0，…，R7的内容分别为1、3、5、7、−2、−4、−6和−8。

如下是一条ADD指令，其含义是将R0和R1的内容相加，然后存入R2。



其中ADD指令的两个源操作数分别由bit[8:6]和bit[2:0]表示，即R0和R1；ADD运算结果存放的目的寄存器由bit[11:9]表示，即R2。图5-2所示的是ADD运算之后寄存器文件的内容（对比图5-1，注意其变化）。

Register0	(R0)	0000000000000001
Register1	(R1)	0000000000000011
Register2	(R2)	0000000000000101
Register3	(R3)	0000000000000111
Register4	(R4)	1111111111111110
Register5	(R5)	1111111111111100
Register6	(R6)	1111111111111010
Register7	(R7)	1111111111111000

图5-1 ADD指令执行前的寄存器文件

Register 0	(R0)	0000000000000001
Register 1	(R1)	0000000000000011
Register 2	(R2)	0000000000000100
Register 3	(R3)	0000000000000111
Register 4	(R4)	1111111111111110
Register 5	(R5)	1111111111111100
Register 6	(R6)	1111111111111010
Register 7	(R7)	1111111111111000

图5-2 ADD指令执行之后的寄存器文件

5.1.3 指令集

一条指令分为两个部分：操作码（做什么）和操作数（对谁操作）。一个ISA（或体系结构）的指令集（instruction set）定义包括：操作码的集合、数据类型和寻址模式。其中寻址模式决定了操作数的存放位置。

在刚才的例子中我们看到，其操作码是ADD，寻址模式是“寄存器模式”（register mode）。操作码ADD的意思是请求计算机执行补码加法，而被操作对象（或数据）则来自通用寄存器。

5.1.4 操作码

有的ISA拥有一个庞大的指令操作码集合，其中每一个操作码对应了计算机的各种功能之一；而有的ISA的操作码集合却很小。有些ISA具备特殊科学计算能力（即操作码），例如惠普公司的Precision Architecture具有一条同时操作三个参数并完成乘/加混合操作的指令($A \times B+C$)；还有的计算机具备图像/图片处理专用的指令，如Intel公司的*MMX*（Multi-Media eXtend）指令（常规x86指令集的扩展）。另外，大多数ISA还包括操作系统相关指令，如VAX结构（20世纪80年代的主流机型）就具备保存程序信息的专用指令（用于操作系统切换用户程序之用）。有趣的是，现在的大多数计算机（或ISA）通常却使用一长串的指令来完成信息保存工作（而不是一条指令），这看起来有些“倒退”？这其中是有道理的，有关的话题可能要在以后的课程里才能解释清楚。究竟哪些指令要丢弃、哪些指令要添加，一直是体系结构设计过程中争议最激烈的话题，因为要考虑的因素很多！

LC-3的ISA结构定义了15条指令，每条指令对应一个操作码（指令的bit[15:12]）。细心的读者可能注意到，4个bit的操作码字段可以定义16种指令。在LC-3中，操作码值1101没有定义，我们将其预留（以后再定义）。

所有指令可以分为三类：运算（operate）、数据搬移（data movement）和控制（control）。运算类指令负责处理信息；数据搬移类指令则负责在内存和寄存器之间以及内存/寄存器和输入/输出

设备之间转移信息；控制类指令负责改变指令执行的顺序，即它们能让程序随时跳转至另一个地方继续执行（而不是常规的顺序向下执行）。

图5-3列出了LC-3的所有指令及其格式，其中bit[15:12]标识了各种对应的操作码。有关各种格式的使用，将在5.2、5.3、5.4节详细阐述。

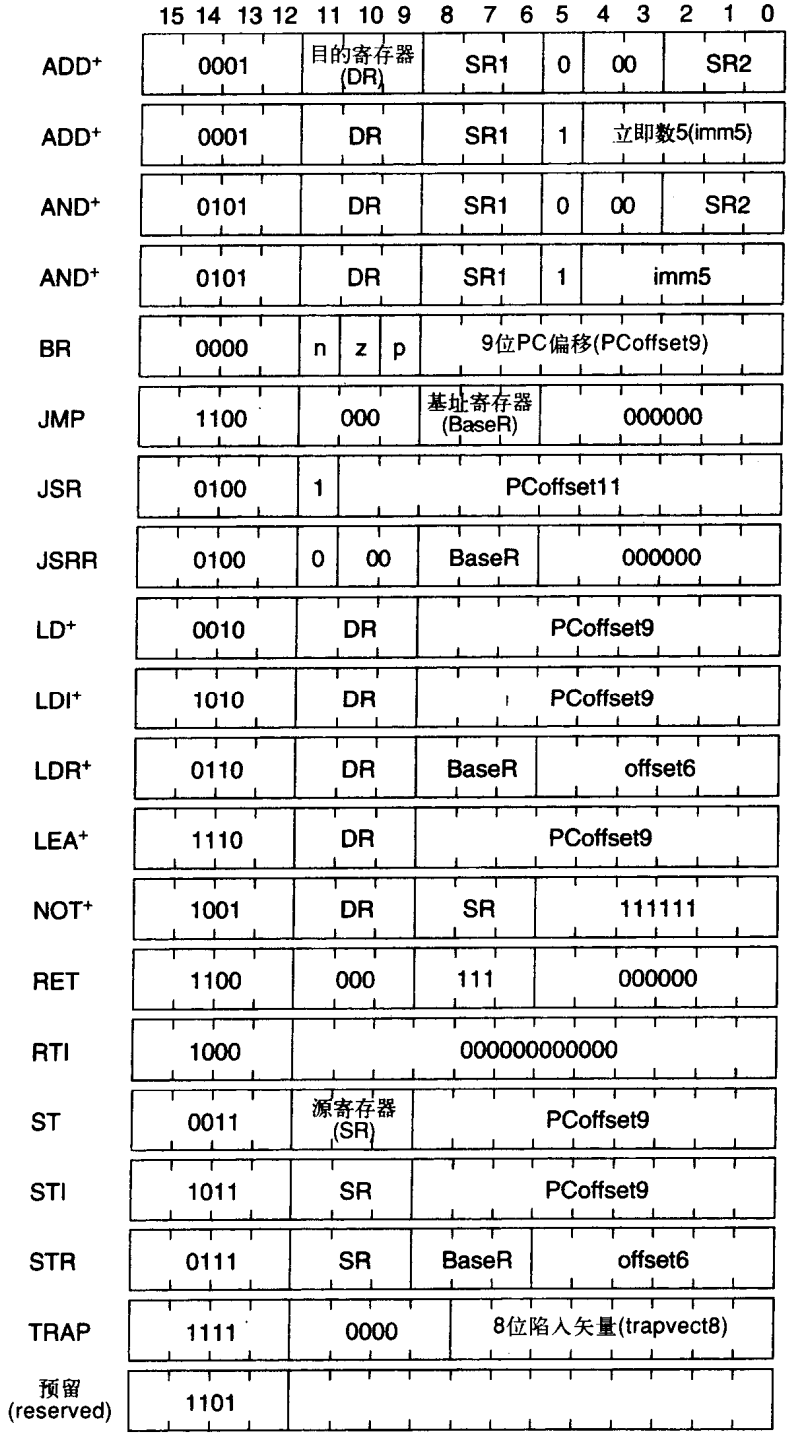


图5-3 LC-3指令集（全部）的格式。注意，+表示该指令将改变条件码

5.1.5 数据类型

*数据类型*是指信息的表达方式，即意味着ISA的操作码是怎样理解这些表达信息的。计算机表达同一个信息的方式有很多种，这一点我们应该不会惊奇，生活中有很多类似的例子。例如，对一个孩子来说，我们问他有几岁时，他伸出三个指头；但他也可能像个大人一样，在纸上写一个数字“3”；而对于CS或CE专业的大学生，他可能会写成“0000000000000011”，即数字3的16位二进制编码；化学专业的学生可能会写成“3.0×10^0”。这四种方式表达了同一个信息实体：3。

如果ISA的操作码能识别/处理某种数据类型表示的信息，我们称该ISA**支持**这种数据类型。第2章中，我们介绍了LC-3的ISA所支持的惟一数据类型：补码整数。

5.1.6 寻址模式

*寻址模式*是指定义操作数位置（或来源）的一种机制。操作数可能存在的地方无非是以下三种之一：内存、寄存器或指令本身。其中，我们称存在于指令之中的操作数为“*字面值*”（literal）或“*立即数*”（immediate）。“*字面值*”说法是指该操作数是由指令的bit从字面上组成的，而“*立即数*”说法是指我们可以从指令中立即获得该操作数，即我们不需要再从别的地方去寻找该操作数。

LC-3共支持5种寻址模式：立即数、寄存器以及三种内存寻址模式，即*相对寻址*（PC-relative）、*间接寻址*（indirect）、*基址偏移*（Base + offset）。在5.2节，我们将学习两种寻址模式：寄存器和立即数，在5.3节的数据搬移指令中，将采用全部5种模式。

5.1.7 条件码

有关LC-3 ISA概述的最后一个内容是“*条件码*”（condition code）。几乎所有的ISA（或机器）都具备“基于之前指令的执行结果来改变指令执行序列”的能力，其中的机制就是“条件码”。LC-3具有三个位寄存器（单bit长度），每当8个通用寄存器中任意一个被修改或写入之时，三个单bit位就会发生对应变化（被置1或清零）。三个位寄存器分别是：*N*、*Z*和*P*，对应的意思是：负数、零、正数。即每当任意GPR寄存器被写入时，根据写入结果是负数、零或正数，分别设置三个条件位（0或1）。例如，如果写入GPR的结果是负数，则*N*位置1、*Z*和*P*清0；如果结果为0，则Z置1、P和N清0；如果写入结果是正数，则P位置1、N和Z清0。

我们称这三个单bit寄存器为“*条件码*”，因为控制指令可根据这些位的“条件（或状态）”决定执行顺序（或方向）是否改变。x86和SPARC机器就是两个例子，它们都基于条件码来控制指令的执行顺序。在5.4节中，我们将介绍LC-3是怎样做的。

5.2 操作指令

操作指令是处理数据的指令。运算操作（如加/减/乘/除，即ADD/SUB/MUL/DIV）和逻辑操作（如与/或/非/异或，即AND/OR/NOT/XOR）都属于典型的操作指令。LC-3只支持三种操作指令：ADD、AND和NOT。

NOT指令（操作码=1001）是惟一的单操作数（unary）指令，即该类指令只需要单个源操作数。NOT指令对16位源操作数做“按位取反”（*bit-wise complement*），并将结果存入目的寄存器。NOT指令对源和目的操作数的访问都是“寄存器寻址模式”。其中bit[8:6]是源寄存器，bit[11:9]是目的寄存器，bit[5:0]为全1。

如果R5寄存器的内容是0101000011110000，则执行以下指令之后，

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	1	0	1	1	1	1	1	1	1
NOT				R3			R5								

R3的内容将是1010111100001111。

如图5-4所示是数据通路中执行NOT有关的关键部分。由于NOT是单操作数指令，所以ALU的输入只有A。ALU的信号表明是NOT操作（按位取反操作），源操作数是R5，运算结果写入R3寄存器。

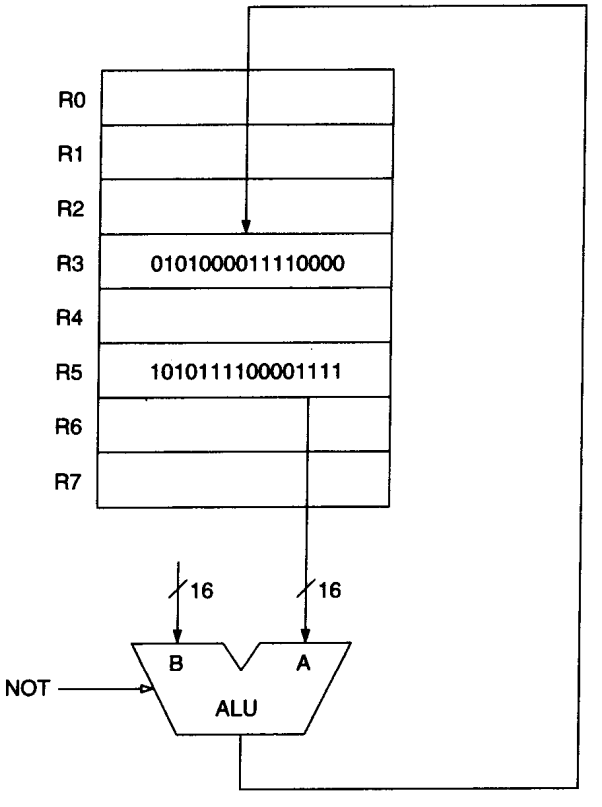


图5-4 执行NOT R3, R5的数据通路

ADD（操作码=0001）和AND（操作码=0101）都是双操作数（binary）指令，即它们都需要两个16位的源操作数。ADD指令执行两个操作数的补码加法；AND指令对两个操作数的16个位中的每个“bit对”做“按位与”（bit-wise AND）操作。同NOT一样，ADD和AND的源操作数之一（bit[8:6]）和目的操作数（bit[11:9]）都是寄存器寻址方式，运算结果写入目的寄存器。

但ADD和AND指令的第二个源操作数寻址方式，则可以是寄存器方式或立即数方式（bit[5]指明是两种模式中的哪一种）。bit[5]=0表明第二个源操作数是寄存器，bit[2:0]则代表该寄存器。这种情况下bit[4:3]=00。

例如，如果R4内容为6，R5为-18，则以下指令执行之后

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	1
ADD				R1			R4						R5		

R1内容为12。

如果bit[5]=1，则表明第二个操作数携带在指令之中，即对bit[4:0]做16位扩展之后再参加运算。图5-5所示是执行指令“ADD R1, R4, #−2”的数据通路中的关键部件。

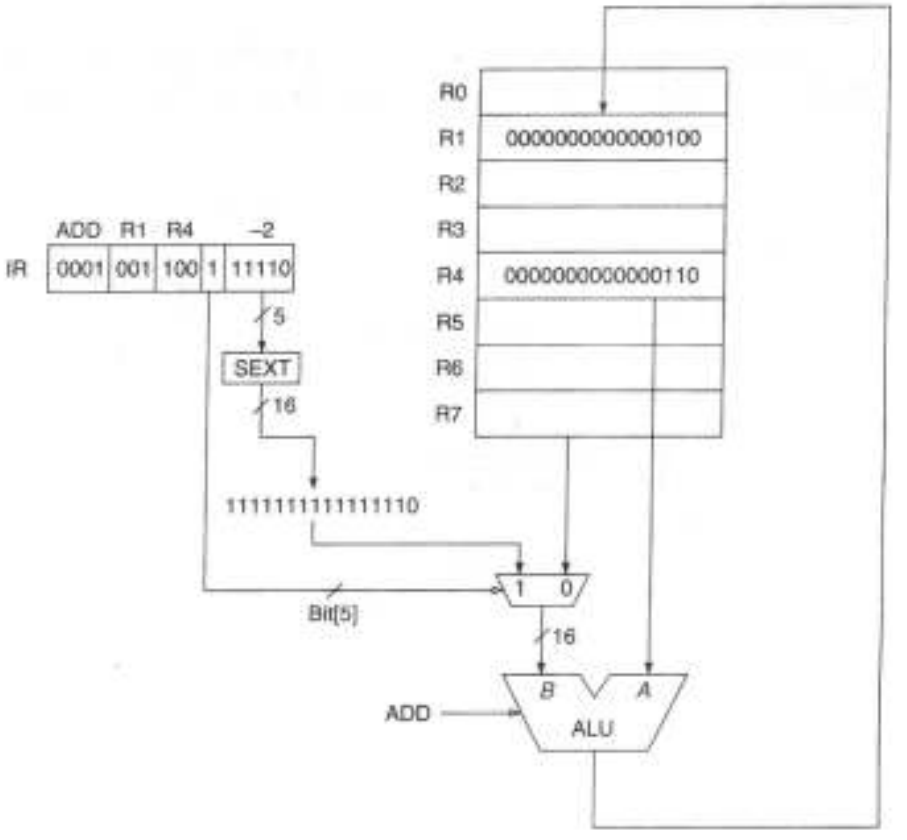


图5-5 执行ADD R1, R4, #-2的相关数据通路

由于在ADD和AND指令中，只有bit[4:0]字段可以表示立即数，所以可表示的操作数（补码形式的）范围是有限的。问：哪些整数是可以的（即哪些整数可被用做立即操作数）？

例5-1 请问：以下指令的执行结果是什么？

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0

答案：寄存器R2被清零。

例5-2 下面指令是什么操作？

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	1	0	1	0	0	0	0	1

答案：寄存器R6增量加（R6←R6 + 1）。

注意，在一条指令中，一个寄存器可以同时扮演源寄存器和目的寄存器，LC-3指令都是如此。

例5-3 回顾补码运算的方法，是对一个数“取反加⊖”。

假设数值*A*和*B*分别存放在R0和R1中，请问：执行“*A*减*B*并将结果写入R2”的三条指令执行序列是什么？

答案：

⊖ 事实上只有负数是这样的。——译者注



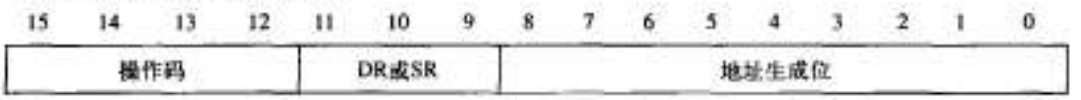
5.3 数据搬移指令

数据搬移指令的任务是在通用寄存器（GPR）和内存之间、寄存器和输入/输出设备之间搬动数据。其中，有关在寄存器和设备之间相互搬移数据的话题，将放在第8章和第9章的一个重要部分展开。本章只讨论内存和通用寄存器之间的数据搬移。

我们称数据从内存移入寄存器为"装载"（load），而从寄存器转入内存为"存储"（store）。注意，在两种情况下，数据源（或源操作数）的信息内容都不会因为此移动而改变，但目的操作数的内容将被刷新，即以前的内容被破坏了。

LC-3有7种搬移指令：LD、LDR、LDI、LEA、ST、STR和STI。

load和store指令的格式如下：



数据搬移指令需要两个操作数：源和目的。源是指将被搬移的数据，目的是指搬移的去处（或目的）。两个操作数之一必然是寄存器，另一个则是内存或输入/输出（I/O）设备。如我们前面说过的，对于第二个操作数，本章只讨论内存，有关输入/输出设备的讨论，留到第8章再讨论。

bit[11:9]标识了操作数之一，即寄存器。如果是load类型指令，则该字段代表DR（即Destination Register，目的寄存器），即指令周期结束时，来自内存的内容将被写入该寄存器；如果是store指令，则SR代表该寄存器内容将被写入内存。

bit[8:0]是"地址生成位"（address generation bit）。这意味着基于bit[8:0]的信息，可以计算出第二个操作数的16位地址。在LC-3中，有4种生成地址（或阐释bit[8:0]）的方法，也就是4种寻址模式。采用哪种寻址模式是由操作码决定的。

5.3.1 PC相对寻址

LD（opcode=0010）和ST（opcode=0011）采用PC相对（PC-relative）寻址模式。该寻址模式之所以这么命名，是因为bit[8:0]代表的是相对当前PC的偏移值。地址计算方法是：先将bit[8:0]内容做16位扩展，然后与PC值（已增量）相加。已增量PC的内容是当前指令在取指令节拍之后的值。如果是load操作，计算出来的地址就是将被读取的内存单元的地址，读取结果被存入bit[11:9]指定的寄存器。

假设下列指令存放在地址x4018中，则被读取内存的地址是x3FC8，结果存放在R2中。

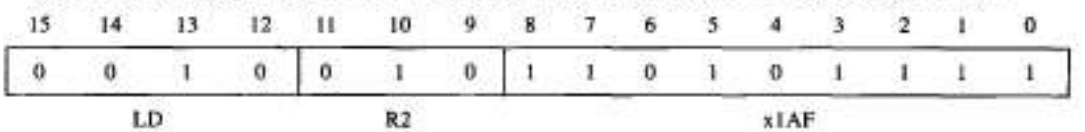


图5-6所示是该指令执行的关键数据通路，其中标注了LD执行的三个步骤：一是将已增量PC（x4019，原先是x4018）与IR[8:0]内容（xFFAF）相加，得出的结果（x3FC8）被装入MAR；二是读取地址为x3FC8的内存单元，其内容暂存入MDR（假设该单元的内容是5）；第三步则是将内容5装入R2，指令周期结束。

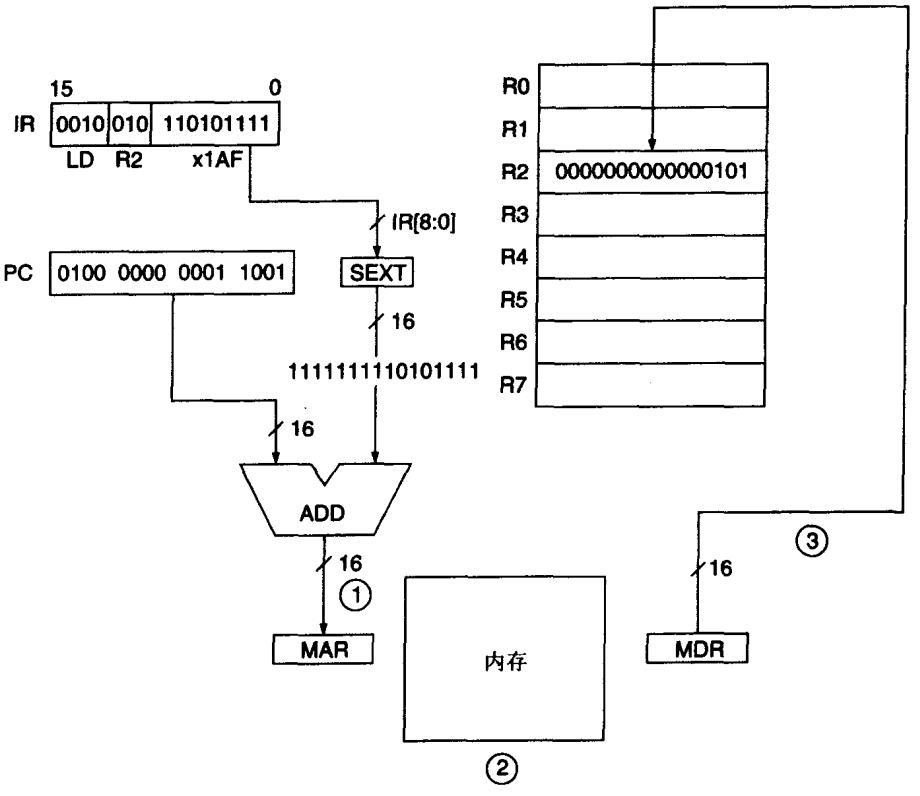


图5-6 LD R2, x1AF执行时的数据通路

注意，内存地址的范围是受限的（不是整个地址空间的范围），即该地址只在相对当前LD或ST指令所在地址的+256和−255范围内（注意，在偏移量相加之前PC已被增量）；另外，指令中bit[8:0]的内容是一个有符号的数值。

5.3.2 间接寻址

LDI（opcode=1010）和STI（opcode=1011）指令采用的是间接寻址模式。首先，采用和LD或ST一样的方法计算出一个地址。但是该地址不是提取或存入操作数所用的地址，事实上，该地址中存放的是另一个地址，后者才是load或store操作数的真正地址。因而，我们称之为“*间接*”模式。值得注意的是，这种方式下可寻址的范围是整个内存空间的任意地方（而不是LD和ST之PC相对模式下的+256~-255受限范围）。同其他load类和store类指令一样，指令的*bit[11:9]*字段标识了LDI的目的寄存器和STI的源寄存器。

假设下列指令的地址是x4A1B，地址x49E8的内容是x2110，该指令的执行结果是R3寄存器被装入内容x2110。

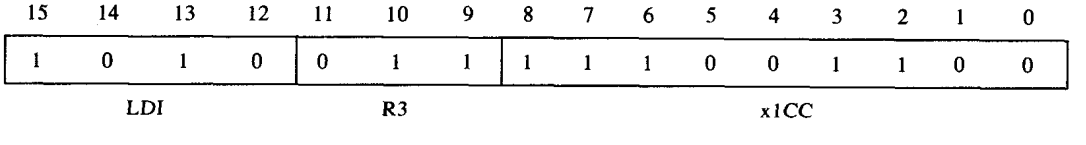


图5-7是与执行该指令相关的关键数据通路。如同LD和ST指令一样，第一步是产生地址，即将PC（x4A1C）与IR[8:0]的符号扩展值（xFFCC）相加，并将结果（x49E8）装入MAR寄存器；第二步是读取内存，将地址x49E8的内容（x2110）读入MDR；第三步中，由于x2110不是操作数（而是操作数所在的地址值），因而它又被装入MAR；第四步是再次读取内存，MDR再次被装入。这一次MDR的内容则是地址x2110的内容。假设x2110的内容是−1，那么在第五步中，MDR的内容（−1）被装入R3，至此，指令周期结束。

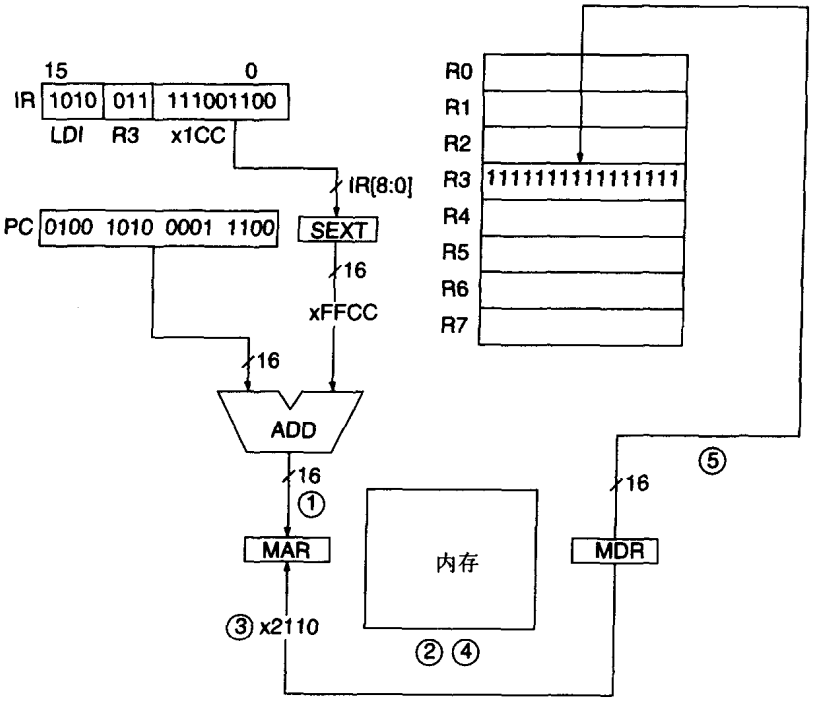


图5-7 LDI R3, x1CC指令执行时的数据通路

5.3.3 基址偏移寻址

LDR（opcode=0110）和STR（opcode=0111）采用的是基址+偏移（Base + offset）的寻址模式。之所以命名为“基址+偏移”模式，是因为操作数的地址是由6-bit偏移量的符号扩展和基址寄存器的内容相加而成的。6-bit的偏移量来源于指令的bit[5:0]字段，而基址寄存器则由指令的bit[8:6]字段标识。

基址偏移寻址中，6-bit补码的表示范围为-32至+31之间，它必须先符号扩展为16位值，然后才能与基址寄存器相加。

假设R2的内容是16位数值x2345，则下列指令的执行会将内容x2362装入R1。

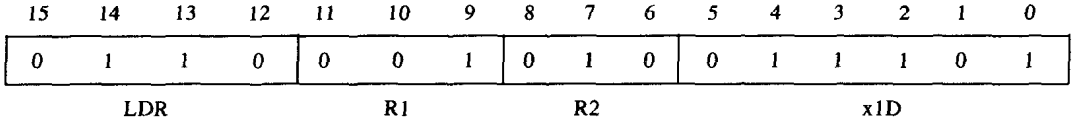


图5-8所示是执行该指令的关键数据通路。首先，R2的内容（x2345）与IR[5:0]内容（x001D）相加，结果（x2362）装入MAR；其次，读内存并将x2362的内容装入MDR。假设内存x2362的内容是x0F0F，则第三步将MDR内容（x0F0F）装入R1。

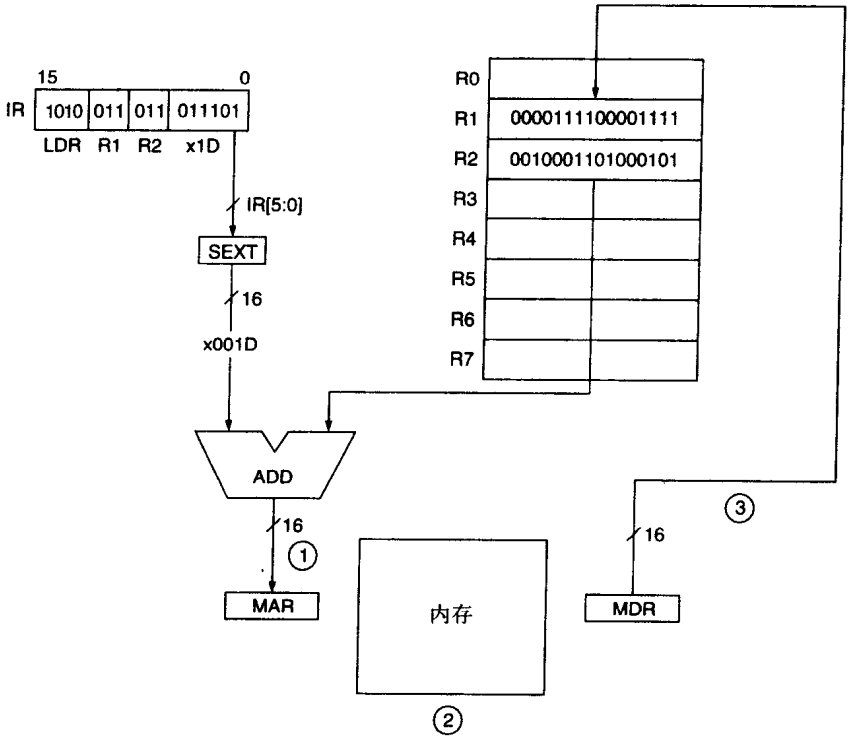


图5-8 LDR R1, R2, X1D指令执行时的关键数据通路

值得一提的是，在基址偏移寻址模式下，操作数的可寻址范围也是任意的（全内存空间）。

5.3.4 立即数寻址

数据搬移指令的第4种也是最后一种寻址模式是立即数寻址模式。该模式只用于有效地址装载指令（load effective address, LEA）。LEA（opcode=1110）将增量PC与bit[8:0]的符号扩展值相加，并装入bit[11:9]指定的寄存器。立即数模式的命名原因是：被装入寄存器的数值的获取是“立即的”（直接从当前指令字段中抽取，而不需要任何内存访问）。

LEA指令的用途是对寄存器做初始化，即向寄存器装入一个地址值，且该地址是与当前指令很相近的一个地址。假设地址x4018处包含指令LEA R5,#-3，即PC内容为x4018，那么下列指令（即x4018处指令）执行之后，R5的内容为x4016。

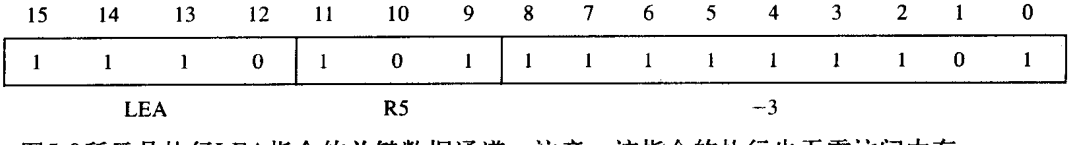


图5-9所示是执行LEA指令的关键数据通道。注意，该指令的执行也无需访问内存。

再次声明一下，LEA指令是“唯一的”，一个无需访问内存的load类型指令。它将由增量PC和指令的地址生成bit内容相加，产生的地址值装入DR（目的寄存器）。

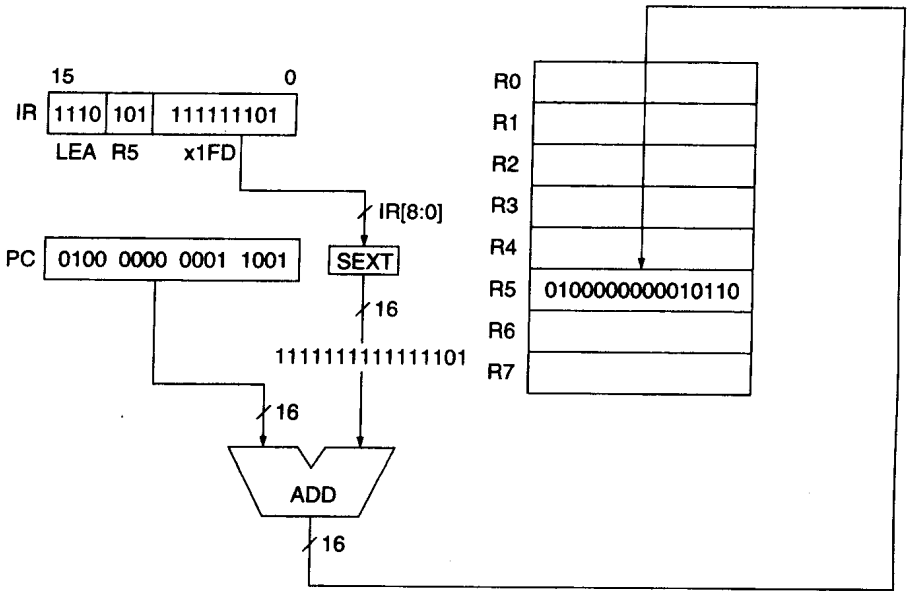


图5-9 LEA R5, #-3指令执行时的关键数据通路

5.3.5 一个例子

下面我们通过一个完整的例子，总结一下学过的各种寻址模式。假设地址x30F6~x30FC范围的内容如图5-10所示，当前PC值为x30F6，试观测7个指令周期之后的运行结果。

地址	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x30F6	1	1	1	0	0	0	1	1	1	1	1	1	1	1	0	1	R1<−PC−3
x30F7	0	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0	R2<−R1+14
x30F8	0	0	1	1	0	1	0	1	1	1	1	1	1	0	1	1	M[x30F4]<−R2
x30F9	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2<−0
x30FA	0	0	0	1	0	1	0	0	1	0	1	0	0	1	0	1	R2<−R2+5
x30FB	0	1	1	1	0	1	0	0	0	1	0	0	1	1	1	0	M[R1+14]<−R2
x30FC	1	0	1	0	0	1	1	1	1	1	1	1	0	1	1	1	R3<−M[M[x3F04]]

图5-10 各种寻址模式的示例

最初PC指向x30F6，或者说第一条将要执行的指令存储在x30F6。其操作码是“1110”——LEA指令，即将源操作数bit[8:0]的符号扩展值和增量PC值相加，然后将求和结果写入目的寄存器bit[11:9]。bit[8:0]的16位符号扩展值为xFFFD，增量PC值等于x30F7；因而LEA指令结束后，R1的内容为x30F4，PC为x30F7。

第2条指令的位置是x30F7。其操作码是“0001”——ADD指令；它将bit[8:6]指定的寄存器与bit[4:0]立即数（因为bit[5]=1）的符号扩展值相加，求和结果存入bit[11:9]寄存器。因为之前的指令已在R1中存入x30F4，立即数符号扩展后的值为x000E，因而R2的装入值为x3102。指令结束后，R1的内容仍然为x30F4，R2为x3102，PC则增量为x30F8。

第3条指令的位置是x30F8。操作码是“0011”——ST指令，即采用PC相对寻址模式将寄存器内容写入当前指令附近的内存单元。该内存单元地址的计算是PC增量值与bit[8:0]符号扩展值相加，然后将bit[11:9]寄存器内容写入该内存单元中。bit[8:0]的符号扩展值是xFFFB，增量PC的值是

x30F9。因而，ST指令结束后，内存x30F4中的内容为x3102，PC为x30F9。

在x30F9，我们发现操作码0101——即AND指令。执行之后，R2为0，PC为x30FA。

在x30FA，我们发现操作码0001——即ADD指令。执行之后，R2为5，PC为x30FB。

在x30FB，我们发现操作码0111——即STR指令。STR指令（同LDR指令一样）采用基址+偏移寻址模式。内存地址是由基址寄存器（bit[8:6]）和偏移（bit[5:0]的符号扩展）求和而得。本例中，bit[8:6]指定了R1，而R1内容仍然是x30F4，偏移（offset）的16位符号扩展值为x000E。由于x30F4 + x000E = x3102，所以内存地址是x3102。因而STR指令将bit[11:9]指定的寄存器（R2）内容存入地址为x3102的内存单元。回顾前面的执行，R2当前的内容应该是5，即意味着指令执行之后，x3102单元的内容为5，PC内容为x30FC。

在x30FC，我们发现操作码1010——即LDI指令。LDI指令（同STI指令一样）采用间接寻址模式。该模式下，它事先采用和PC相对寻址模式相同的计算方法获得一个地址。本例中，bit[8:0]的16位符号扩展值是xFFF7，增量PC为x30FD，则它们求和得x30F4（这只是存放操作数所在的地址的内存地址——“地址的地址”）。内存地址x30F4的内容是x3102，这个地址是最终的操作数所在的地址。LDI将该地址中的内容（本例中为5）装入bit[11:9]指定的寄存器（R3）。执行结果是，R3等于5，PC为x30FD。

5.4 控制指令

控制指令是指那些能够改变指令执行顺序的指令。如果没有控制指令，当前指令完毕后的下一条执行指令（时间顺序）一定也是内存位置顺序的下一条（空间顺序）。这是因为取指令节拍（FETCH）之后，PC总是自动增量加1；但下面我们会看到，在很多情况下，需要打破这种顺序执行。

LC-3有5种操作码可以打破这种顺序：条件跳转、无条件跳转、子程序（又称函数）调用、TRAP、中断返回。在本节中，我们讨论的主要内容是最常用的“条件跳转指令”，无条件跳转和TRAP指令也会有所介绍。其中，TRAP非常有用，因为它使得你可以在无需了解输入/输出设备等复杂的计算机组织细节的情况下，与计算机交互信息，为什么呢？别着急，我们将有关TRAP指令、子程序调用、中断返回等话题，留在了第9、10章。

5.4.1 条件跳转指令

条件跳转指令（opcode=0000）的格式如下所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	N	Z	P	PC 偏移								

其中，bit[11]、[10]、[9]分别对应了5.1.7节中讨论的三个条件码，“**所有**”会对寄存器进行写操作的指令都会设置这三个条件码（之一），这些指令包括ADD、AND、NOT、LD、LDI、LDR和LEA。

条件码的使用方法是：条件跳转指令通过对条件码的判断，来决定是否改变指令流；换句话说，即是否改变正常的顺序（sequential）执行方式（在每条指令的取指令节拍，自动对PC增量）。

在控制指令的指令周期中，取指令（FETCH）和译码（DECODE）节拍没有什么特殊的，取指令节拍之后，PC自动增量；地址计算节拍（EVALUATE ADDRESS）也与LD和ST指令相同，即将增量PC和指令bit[8:0]的16位符号扩展值相加求得地址。

而在执行节拍（EXECUTE），处理器将检测某个条件码（对应指令中指定的条件码）是否为1。如果指令的bit[11]=1，则检查条件码N；如果bit[10]=1，则检查条件码Z；如果bit[9]=1，则检查条件

码P。如果bit[11:9]三个位全是0，则不检查任何条件码。任意一个条件码的状态为1，都将使得PC内容重新修改，即将地址计算节拍（EVALUATE ADDRESS）生成的地址重新装入PC；而如果所有条件码的状态都为0，则PC内容保持不变（意味着在下一个指令周期，将顺序地读取下一条指令）。

例如，假设最近装入寄存器的内容是0，则如下的指令（位置在地址x4027）将对PC装入x4101，即下一条要执行的指令来自x4101（而不是x4028）。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	1
BR				n	z	p	x0D9								

图5-11是执行该指令的关键数据通路。注意其中的判断逻辑，即决定指令流是否是顺序的。本指令的答案是“yes”，因而PC被重新装入x4101，替换原先的x4028（取指令阶段自动装入的增量值）。

如果所有的三个位bit[11:9]都是1，则三个条件码都将被检测。那么，什么情况下需要检查所有的条件码状态呢？在回答之前，我们先分析一下它的执行细节。首先，不论在什么情况下，三个条件码中至少有一个是1，因为之前的寄存器装入内容可能是负数、0或正数（不存在其他的选择）；其次，如果指令检测所有的三个条件状态位，则必定有一个命中（即匹配）。换句话说，PC内容必然要更新，即装入在地址计算阶段得出的地址。现在我们可以回答刚才的问题了，由于这种指令“必定（或无条件）”地要改变已有的PC内容（顺序的），所以我们称这种检查所有条件状态位的指令为“无条件跳转”（unconditional branch）指令。

假设如下的指令所在的地址是x507B，则该指令执行之后，PC内容为x5001。

顺便再问个问题：如果跳转（BR）指令的三个检测位bit[11:9]都是0，将发生什么情况？

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	0	0	0	0	1	0	1
BR				n	z	p	x185								

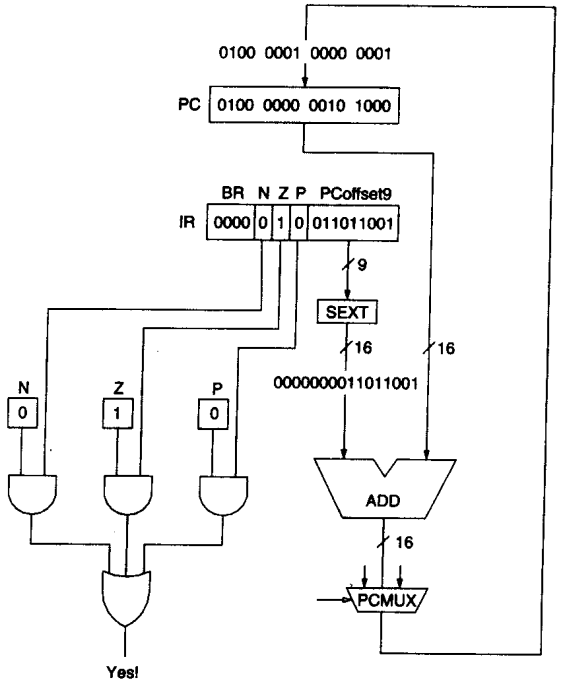


图5-11 执行BRz x0D9指令的关键数据通路

5.4.2 一个例子

下面通过一个简单的例子，讲述控制指令在指令集中的重要价值。

假设在地址x3100~x310B之间包含了12个整数，我们希望写个程序对它们求和。解决这个问题的算法描述如图5-12所示。

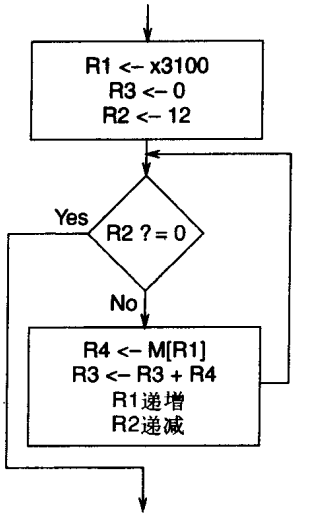


图5-12 对12个整数求和的算法

首先，与所有的算法一样，我们必须“初始化变量”，即要对算法中使用到的变量设置其初始值。在此算法中，涉及三个变量：下一个待求和整数的地址（存放在R1中），求和累计变量（存放在R3），还未计算的整数数目（存放在R2）。这三个变量的初始化内容分别为：R1的初始值为第一个整数的地址；R3因为是不断记录累计和的，所以被初始化为0；R2记录的是还未求和的整数的数目，所以初始值为12。

初始化之后，就开始求和过程。程序重复下面的过程：从12个整数中读取一个新的整数存入R4，然后加入R3。另外，在每执行一次ADD运算的同时，R1被增量（地址）——指向下一个待求和整数，R2则被减量（减1）——告诉我们后面还有多少个整数有待计算。当R2变为0的时候，条件码位Z被置1，通过它我们知道，事情已经完成了。

地址	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	R1<−3100
x3001	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	R3<−0
x3002	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2<−0
x3003	0	0	0	1	0	1	0	0	1	0	1	0	1	1	0	0	R2<−12
x3004	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	BRz x300A
x3005	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	R4<−M[R1]
x3006	0	0	0	1	0	1	1	0	1	1	0	0	0	1	0	0	R3<−R3+R4
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1<−R1+1
x3008	0	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	R2<−R2−1
x3009	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	0	BRnzp x3004

图5-13 实现图5-12算法的程序

图5-13所示是实现该算法的一个小程序（10条指令）。

其中：

- 程序起始于PC=x3000，此第一条指令的任务是，将地址值x3100装入R1寄存器（当前增量PC=x3001，PCoffset的符号扩展值=x00FF，两者求和为x3100）。
- 随后，位于x3001的指令负责将R3清0。R3的任务是记录累计和，所以它必须被清0，即如前面所说的“初始化SUM为0”。
- 位于x3002和x3003的指令负责给R2赋值（=12），即被累计整数的数目。R2的任务是跟踪已有多少整数被加过了，该任务是通过位于地址x3008的指令在每次加法之后都对R2减1实现的。
- 位于x3004的是条件跳转指令。注意，该指令的bit[10]=1，即该指令要检查条件码Z。如果Z状态为1，就意味着之前的R2内容已被减为0了，工作结束了。如果Z状态为0，则意味着工

作还没完成，还得继续。

- 位于x3005的是load类指令，将位于x3100的内容（第一个整数）装入R4；然后由位于x3006的指令将其加入R3。
- 位于x3007和x3008的指令负责一些常务性的收尾（bookkeeping）工作。x3007的指令对R1增量，使之指向下一个新的待加入整数（如地址x3101）；而x3008的指令对R2减量，用于记录还有多少个整数还未加入，同时该指令的执行还影响着条件码*N*、*Z*、*P*的状态值。
- 位于x3009的是一条无条件跳转指令（从其bit[11:9]全为1可以判断出来）。它向PC装入x3004。该指令的执行不会影响条件码，所以，之后将执行的条件跳转指令（x3004）将基于x3008的指令（改变条件码）。

值得注意的是，在转折点的指令执行顺序是x3008→x3009→x3004，但因为x3009的指令不影响条件码，因而事实上是x3008的指令结果（会影响条件码）决定了x3004指令的执行结果（即顺序执行或跳转）。x3008的指令之所以会改变条件码，是因为它对R2减1并写回R2。如果还有整数需要加入，则x3008的ADD指令产生的结果必然是大于0的（即Z条件码是被清0的）。而x3004的条件跳转指令将检查该条件码位Z，如果Z被清0了，则PC不会被修改，即下一个指令周期的指令将从x3005继续获取。

从x3000开始，在条件跳转指令的控制下，指令的执行序列为：x3000, x3001, x3002, x3003, x3004, x3005, x3006, x3007, x3008, x3009, x3004, x3005, x3006, x3007, x3008, x3009, x3004, x3005, ……如此反复直到R2变为0，这样，下次再执行x3004的条件跳转指令时，PC将被修改为x300A，即程序将跳转至x300A继续。

最后，值得一提的是，我们也可以在不使用任何控制指令的情况下，写出一个对12个整数求和的程序。我们仍然需要地址x3000的R1初始化指令；但不需要累计、变量初始化指令（x3001）和剩余整数数目变量初始化代码（x3002和x3003）。我们可以直接将x3100的内容装入R3，然后重复地（通过增量R1、将整数装入R4、将R4累加到R3）加入其他11个整数。在第12个（即最后一个）整数加完之后，我们可以转入下一个任务（如图5-13中x3004处的跳转指令所做的一样）。

不幸的是，没有控制指令（或跳转指令）的新程序代码长达35行指令（而不是原先的10行）。如果我们要求100个整数的和，则这种不采用控制指令的写法编出的代码将长达299条指令（而原先的写法仍然是10条指令就可完成）。由此我们可以看出（如图5-13所示），由于它具有改变指令执行顺序（或方向）的能力，就使得代码得以重用（reuse）——总的代码长度变短了。

5.4.3 循环控制的两种方法

在某种机制的控制下，一遍一遍地被重复执行的指令序列称为“循环”（loop）语句。在12个整数求和的代码中，就包含了一个循环。循环体代码每执行一遍，累计和就加入了一个新整数，计数器也被减1（用于判断剩余整数数目）。我们称循环体每执行一遍为“循环体执行了一轮”。

控制循环体执行轮数的方法有两种：

（1）采用计数器方法（如前所述）。如果需要将一个循环执行*n*遍，我们只需要将计数器初始化为*n*。然后每执行一遍循环，就将计数器减1并判断其是否为0。如果计数器不为0，则重新设置PC至循环的入口，然后开始下一轮执行。

（2）第二种方法是采用“哨兵法”（sentinel）。这种方法在我们“事先不知道要循环多少次”的情况下特别有效。同“计数器方法”不同的是，“哨兵法”通过“对数据内容（而不是计数器余数）的判断”来控制循环是否继续。哨兵法的实现方法是：在被处理数据序列的尾部安放一个“哨兵”——即我们事先就知道绝对不会出现的数值。例如，如果我们正在对一个数字序列做“求

和”运算，那么像“#”或“*”这样的符号，就可以做“哨兵”（因为它们“绝对”不是数字）。所以，每次循环体执行时，都会通过判断哨兵是否出现（即判断当前已读取的数据是否等于“哨兵”字符）来控制循环体是否继续执行，一旦发现“哨兵”，我们就知道任务该结束了。

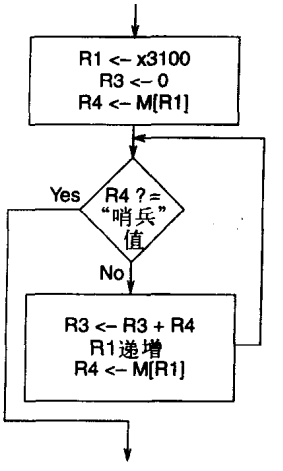


图5-14 基于“哨兵”方法的循环控制算法

5.4.4 例子：哨兵法数组求和

以5.4.2节为例，地址x3100~x310B之间存储的数据全部是正数，那么就可以任意选一个负数作为“哨兵”。假设在内存地址单元x310C放置了一个哨兵“-1”，则该求和计算的相应算法流程图就如图5-14所示，而实现程序则如图5-15所示。

地址	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	R1<--x3100
x3001	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	R3<--0
x3002	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	R4<--M[R1]
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	BRn x3008
x3004	0	0	0	1	0	1	1	0	1	1	0	0	0	1	0	0	R3<--R3+R4
x3005	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1<--R1+1
x3006	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	R4<--M[R1]
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	1	BRnzp x3003

图5-15 对应图5-14算法的程序

与前面的计数器方法一样，x3000的指令向R1装入第一个要加入的数据所在的地址，而x3001的指令将R3（累积和）初始化为0。

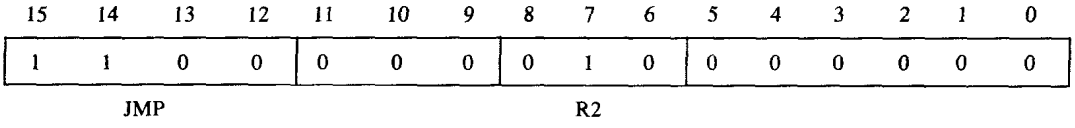
位于x3002的指令真正地将内存中（由R1指向）的数据装入R4。如果此时装入的是“哨兵”数据，则条件码位N必然自动设置为1（因为正常数据都是正数，而哨兵数据是-1）。

位于x3003的条件跳转指令将专门检查N条件码位。如果N位为1，则设置PC为x3008（即进行下一个任务）；如果N位为0，R4必然包含了一个有效数字，则该数字被加入R3（x3004），R1增量指向下一个内存单元地址（x3005），R4装入下一个内存单元的内容（x3006），PC被装入x3003（即无条件跳转至x3003），开始下一轮执行（x3007）。

5.4.5 JMP指令

条件跳转指令存在一个“致命的局限性”，即下一条指令所在位置不能超出“可计算”范围。所谓的“可计算”范围是指PC寄存器和偏移量（bit[8:0]符号扩展值）的求和值。由于bit[8:0]是补码表示，则9个bit的编码所表示的最大地址范围是距当前跳转指令前后+256至-255的地址空间。假如我们要执行的下一条指令是距离当前指令1000的位置，由于9个bit的字段装不下1000这么大的数值，那么条件跳转指令是无法完成这个跳转任务的。

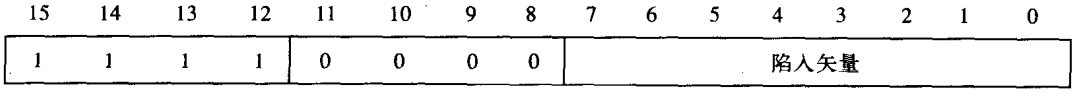
为此LC-3的指令集就提供了JMP指令（操作码=1100）。JMP指令格式如下：



JMP指令的任务就是将寄存器（bit[8:6]标识）的内容装入PC寄存器。假设JMP指令位于x4000，R2的内容为x6600；则JMP指令执行完毕后，下一条执行指令的地址则是x6600。JMP指令可以使得程序执行流跳转至内存空间的任意位置，其中的关键是寄存器（如R2）的宽度是16-bit，它可以表达内存空间的任意地址。

5.4.6 TRAP指令

第9章将详细介绍TRAP指令在计算机数据输入/输出操作中的作用，但由于本章距离第9章还很远，所以在此先做简述。TRAP指令（操作码=1111）的任务是改变PC内容（与JMP指令很相似），使其指向操作系统所在空间内部。换句话说，即以当前程序的身份跳转至操作系统的某个代码入口开始执行。而按照操作系统的术语来说，我们称TRAP指令为激活了操作系统的服务调用（SERVICE CALL）。TRAP指令的bit[7:0]字段表示的是“陷入矢量”（trapvector），我们可以将“矢量（或编号）”理解为程序希望操作系统执行的服务程序的编号。在附录A的表A-2中，给出了LC-3提供的所有服务所对应的陷入矢量。



操作系统在完成服务调用之后，程序计数器（PC）将被设置为TRAP指令后的下一条指令地址。这意味着在程序正常执行的过程中，通过TRAP指令可以随时调用操作系统的服务函数。下面给出几个操作系统服务函数的例子：

- 读取键盘输入（trapvector = x23）。
- 显示器字符输出（trapvector = x21）。
- 终止（HALT）程序（trapvector = x25）。

有关LC-3是怎样实现当前执行程序和操作系统之间交互的机制，是第9章的一个重要话题。

5.5 例子：字符数统计

这是本章的最后一个例子。该例程从键盘读入一个字符，并统计一个文件中该字符出现的次数（计数值）；最后在显示器上显示该计数值。我们先做个假设——文件中任意字符的出现次数很小，即最多只有9次。之所以做这种假设，原因是：我们希望避开做一个复杂转换——二进制数至ASCII码的转换（第10章将介绍此话题）。

图5-16是求解该问题的流程图，其中每个操作步骤都分别用英文和LC-3操作（括号内）予以表示。

第一步是初始化工作，即对R0、R1、R2和R3等4个寄存器赋予初始值。R2代表字符出现的次数，初始值为0；R3是一个指针（pointer），指向文件中的下一个待读字符，初始值等于文件中第一个字符所在的地址；R0的内容则是待检查（或匹配）的字符；R1的任务则是保存当前正在被读取的文件中的字符。

值得注意的是，我们没有要求被检查文件的位置与正在开发程序的距离是近还是远。假设程序的位置是x3000，而文件的位置是x9000，则R3的初始化内容是x9000。

第二步是统计输入字符的出现次数。实现方法是逐个检查文件的每个字符，直到文件结束。

循环体每执行一遍处理一个。回顾5.4.3节介绍的两种控制循环次数的方法，这里我们将采用“哨兵”方法，即采用EOT（End of Text）（00000100）作为哨兵字符。参考附录E中的ASCII字符表。

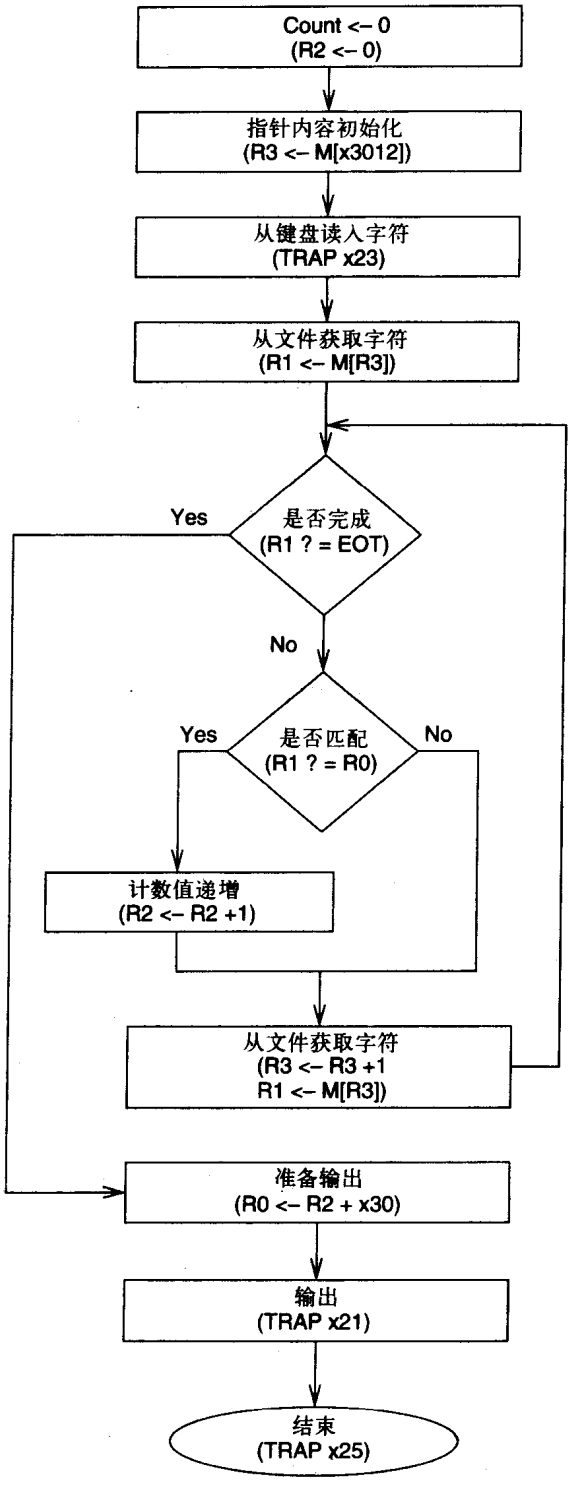


图5-16 统计字符出现次数的算法

每次循环的开始，将*R*1的内容与EOT匹配。如果相同，则循环结束，程序转至最后一步，将计数值显示在屏幕上；如果不相同，则有事情做了。*R*1（当前待检查字符）与*R*0（键盘输入字符）相比。如果相同，则*R*2增量加1。无论如何（相同或不相同），*R*3都增量，指向下一个字符，并将该字符装入*R*1，然后返回循环体的入口，判断该字符是否是哨兵字符——文件结束符。

如果遇到的是文件结束符，即文件所有内容都已检查，则当前*R*2包含的就是键盘输入字符在文件中出现的次数。但这个计数值是二进制数，为了能在屏幕上显示，该数值必须转换成ASCII码。由于事先我们已假设计数值不会超过10，我们只需要在4-bit的计数值前面直接加上0011即可。为什么呢？我们先查看一下附录E的图E-2中十进制数0到9与它们的ASCII码之间的对应关系，很简单，它们的对应关系确实如此——在4-bit二进制编码的前面，再加上4-bit的0011。

最后，程序将转换后的ASCII码（计数值对应的ASCII码）输出到显示器上，程序结束。

图5-17是图5-16的流程图所对应的机器代码程序。

地址	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2<-0
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	R3<-M[x3012]
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	TRAP x23
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1<-M[R3]
x3004	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0	R4<-R1-4
x3005	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	BRz x300E
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1<-NOT R1
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1<-R1+1
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	R1<-R1+R0
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	BRnp x300B
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2<-R2+1
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	R3<-R3+1
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1<-M[R3]
x300D	0	0	0	0	1	1	1	1	1	1	1	1	0	1	1	0	BRnzp x3004
x300E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	R0<-M[x3013]
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	R0<-R0+R2
x3010	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	TRAP x21
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	TRAP x25
x3012	文件的起始地址																
x3013	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	ASCIITEMPLATE

图5-17 实现图5-16算法的机器语言程序

首先是初始化操作。x3000的指令负责对R2清零，即将R2与x0000做AND操作；x3001的指令则将x3012的内容装入R3。这是文件的第一个字符所在的地址。此外，我们可以看到，采用R3寄存器存放地址，使得文件可以存放在内存的任意位置。当然，在x3000指令之前，必须有一序列的指令事先将文件起始地址装入x3012。位于x3002的是一条TRAP指令，即意味着要调用操作系统的服务。该TRAP指令中的8-bit陷入矢量等于00100011（或x23），即从键盘读入一个字符并装入R0。参考附录A的表A-2中列出的所有操作系统服务，其中x23（如表A-2所示）代表读入下一个键入字符，并将其装入R0寄存器。位于x3003的指令将R3所指向内存的内容装入R1。

之后，开始“字符检查”操作。首先是R1减去数值4（EOT的ASCII码），并将结果装入R4。

如果结果是0，则意味着到了文件结尾，是输出计数值的时刻了。位于x3005的指令条件地跳转至x300E，即输出计数值的处理代码段。

如果R4不为0，则意味着R1中的字符是合法的，需要做匹配检查。位于x3006、x3007和x3008的几行代码判断R0和R1的内容是否相同，即完成如下的操作：

$$RO + (NOT(R1)+1)$$

只有当R0和R1的内容完全相同时，该表达式的结果才为0。如果两者不相同，则x3009的条件跳转指令将转至x300B；否则（即相同时），则执行x300A的计数值（R2）增量操作。

位于x300B的指令对R3增量，目的是指向文件中下一个待检查的字符；而位于x300C的指令将字符进一步装入R1；x300D的指令无条件地跳转回x3004入口，开始下一个字符的处理过程。

最终当哨兵字符（EOT）被检查到时，就开始计数值输出操作（位于x300E）。位于x300E的指令将00110000装入R0；而位于x300F的指令将R0的内容和计数值（R2）相加，该操作的目的就是将二进制的计数值（R2）转换成ASCII表示（存入R0）。位于x3010的TRAP指令将调用操作系统，完成对R0内容的输出显示。当事情都完成之后，位于x3011的指令再次通过TRAP方式终止程序执行。

5.6 总结：数据通路（LC-3）

在结束第5章的内容之前，让我们回顾一下数据通路（最初提到它是在第3章的图3-33）。我们将对实现LC-3的ISA的所有结构做一个详细的分析，其中的大多数已在图5-4、图5-5、图5-6、图5-7、图5-8、图5-9、图5-11中出现过，下面将这些结构汇总在一张图中（如图5-18所示）。注意，图中的数据通路有两种箭头：实心的和空心的。实心箭头用于被处理信息，而空心箭头用于控制信号。注意，其中的控制信号都源自标识为“Control”的模块，但模块与控制信号箭头之间的连接线都未画出（以使图显得更简洁）。

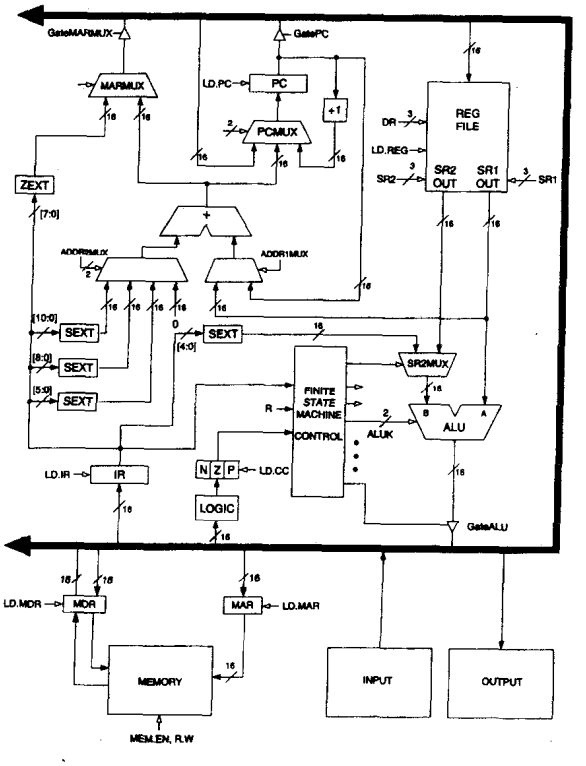


图5-18 LC-3的数据通路

5.6.1 数据通路的基本部件

1. 全局总线

你一定已经注意到了，两端都是箭头的黑粗结构，它代表的就是数据通路的全局总线(Global Bus)。LC-3的全局总线包括16根线和相关的电子部件，它用于一个结构与另一个结构之间传递最多16个bit的信息，当然，每个结构还必须通过一些必要的电子部件才能连接到总线上。总线上每次只能传输一个数据值（即任意时刻最多只有一个发送者）。注意，每个结构在向总线输送数据时，其输入总线箭头的后面都有个三角形，这个三角形即所谓的“三态门”(tri-state device)，它的作用就是保证一个时刻只有一个数据提供者在向总线提供数据；而对于希望从总线上获取这个数据的设备（通常只有一个接收者），则受控于LD.x（load使能）信号（参考3.4.2节的门锁存器）。但并不是所有的计算机都只是单个全局总线（即可能存在多个总线），有关单总线的优缺点，在以后的课程中会有讨论。

2. 内存

不论是什么样的计算机，最重要的部件可能就是内存（Memory）了。内存中包含着指令和数据。访问内存的第一步，是将被访问内存的地址装入MAR寄存器。如果访问操作是load，则向内存传送读信号（RD），结果是内存将数据发送至MDR寄存器；如果是stored操作，则数据事先被存放在MDR寄存器中，然后再向内存传送写信号（WE），以使得MDR输出的数据被装入（MAR）指定的内存单元。

3. ALU和寄存器文件

ALU是信息处理单元。它有两个输入，一个来自寄存器，另一个可能来自寄存器或符号扩展的立即数（指令自身携带）。寄存器文件（R0~R7）最多可以同时传递两个数值，第一个由SR1(3-bit）标识，另一个则由SR2标识。SR1和SR2都是包含在指令中的字段。第2个操作数的选择（寄存器或立即数）是由指令的bit[5]字段决定的。注意，ALU的第2个输入是由一个开关（MUX）控制着的，该开关的控制信号（来自CONTROL模块）事实上就是指令的bit[5]。

ALU的操作结果存放在通用寄存器之一中，会同时改变三个条件码位。ALU向总线提供的是16-bit数据，具体写入哪一个寄存器，是由3-bit的寄存器选择信号DR控制的；同时ALU的输出还将传送给LOGIC模块，判断结果是负数、零还是正数，并由LOGIC对条件码位做相应的设置。

4. PC和PCMUX

PC通过总线向MAR提供地址信息，该地址是下一个指令周期开始（即取指令节拍）要获取的指令所在的地址。而PC本身的输入则是一个3选1PCMUX提供的，复用的选择或切换取决于正在执行的指令类型。我们知道在取指令节拍，PC的内容会自动增量并写回PC，如图中PCMUX的最右端输入。

如果当前指令是一条控制指令，则被选中的PCMUX输入将取决于控制指令的类型（而不是PC增量）。如果当前指令是一条条件跳转指令且条件满足（即跳转），则PC的装入值是增量PC和PCoffset（IR[8:0]的16位符号扩展）的求和。注意，这个求和操作是由专用的加法器（而不是ALU）完成的，其输出结果即为PCMUX的中间输入。PCMUX的第3个输入来自全局总线，在第9和10章介绍过其他控制指令后，就会明白它的使用方法了。

5. MARMUX

如我们所知，内存的访问是通过向MAR寄存器提供地址完成的。而MARMUX控制信号的作用，就是在load、store或TRAP的执行期间，为其在两个地址输入中选择合适的输入。MARMUX右边的输入值是增量PC或基址寄存器与IR字段内容或0的求和。至于究竟是PC还是基址寄存器，以及究竟是哪几位字段信息，则取决于当前指令的操作码。控制信号ADDR1MUX负责选择PC或基址寄存器，ADDR2MUX负责在另外4个数值中做选择。MARMUX的左边输入是trapvector的零

扩展（即在不满16位的数值前填充0，补够16位宽度），它的使用细节将在第9章中讨论。

5.6.2 指令周期

最后，我们找一个指令操作例子，并按照指令周期的执行顺序（时间）将数据通路（空间）复习一遍。假设当前PC内容等于x3456，且地址x3456的内容为0110011010000100；而此时LC-3刚刚完成x3455指令的处理（假设是一条ADD指令），那么下面继续的操作是什么？

1. FETCH

指令执行的第一步是取指令节拍。PC寄存器将提供访问内存（即获取指令）的地址。在第一个时钟周期，PC内容通过全局总线被装入MAR，同时PC自动增量并装回PC。该周期结束的时候，PC内容等于x3457；然后，在下一个周期（假设内存在一个周期内返回信息，而在真正的机器上要数十个周期），内存返回值0110011010000100被装入MDR；再下一个周期，MDR内容被转入IR寄存器。至此，取指令节拍完成。

2. DECODE

随后的一个时钟周期，IR的内容被译码，即由控制逻辑产生对应的一系列控制信号（空心箭头），控制指令的下面操作过程。由于当前操作码是0110（即LDR指令），这意味着将采用“基址+偏移”的寻址模式，从内存读取数据并装入目的寄存器R3。

3. EVALUATE ADDRESS

在下一个时钟周期，开始“基址+偏移”的地址计算。R2（基址寄存器）的内容和IR[5:0]的符号扩展值相加，并通过MARMUX开关将求和结果（地址值）传入MAR。指令中的SR1字段等于010，代表获取基址的寄存器是R2。另外，ADDR1MUX控制信号选择了SR1OUT，ADDR2MUX信号则选择了自右向左的第2个输入源。

4. OPERAND FETCH

在下一个时钟周期（或许更多，如果内存访问的延迟更长的话），该地址内存单元的数据被装入MDR。

5. EXECUTE

LDR指令不需要执行节拍，所以该节拍花费的时间是0个周期（即直接跳过）。

6. STORE RESULT

最后一个周期，MDR的内容被装入R3寄存器。指令的DR字段指定目的寄存器的编号为011(R3)。

5.7 习题

5.1 给定指令ADD、JMP、LEA、NOT，请判断它们分别是操作（或运算）指令，还是数据搬移指令或控制指令？对每一条指令，进一步列出该指令可以采用的寻址模式。

5.2 如果内存可寻址的宽度是64-bit，那么有关MAR和MDR的大小意味着什么？

5.3 有两种终止循环（loop）的方法。一种是采用计数器控制循次数；另一种是采用一个被称为_____的“元素”来控制。适合于做这种元素的字符有什么特性？

5.4 假设内存包含256个位置，每个位置的宽度是16位，问：

a. 表达地址的宽度至少是多少位？

b. 如果采用PC相对寻址模式，并要求控制指令能跳转至20个间隔以内的地址处，那么跳转指令中要留出多少位的字段来标识PC相对偏移（PC-relative offset）？

c. 如果控制指令所在的地址是3，那么在LC-3指令中，它与地址10之间的PC偏移值应该是

多少?

5.5 a. 什么是寻址模式?

b. 列举三种指令操作数可能存在的地方。

c. 列举LC-3的5种寻址模式，并指明各个模式下操作数存在的位置。

d. 5.1.2节中的ADD指令采用的是什么寻址模式?

5.6 回顾2.7.1节中的“机器忙”(machine busy) 例子。假设BUSYNESS位矢量内容存放在R2寄存器中，我们可以采用LC-3指令“0101 011 010 1 00001 (AND R3, R2, #1)”判断机器0是否忙 (busy)。如果该指令的结果是0，则意味着机器0是busy状态。

a. 写一条判断机器2是否busy的LC-3指令；

b. 写一条判断机器2和机器3是否同时busy的LC-3指令；

c. 写一条能够指示没有任何一个机器是busy的LC-3指令；

d. 你能写一个判断机器6是否busy的指令吗? 有什么问题吗?

5.7 LC-3的ADD指令所能表达的最大的正整数是多少?

5.8 如果我们希望将ADD指令能寻址的寄存器数目增加到32个，有什么问题吗? 请解释。

5.9 我们希望设计一个“什么也不做”的指令。许多ISA都有一个什么都不做的操作码，通常称之为NOP指令，即“NO OPERATION”。该指令仍然经历取指令、译码、执行等节拍，只是在执行节拍它什么也不做。下面哪条指令可以扮演NOP的功能，且不影响程序的正常执行?

a. 0001 001 001 1 00000

b. 0000 111 000000001

c. 0000 000 000000000

其中ADD指令与其他两条指令的区别是什么?

5.10 下面的指令*A*和*B*之间的区别是什么? 它们之间的相似性是什么? 差异是什么?

A 0000111101010101

B 0100111101010101

5.11 我们希望通过一条指令的执行来完成：将寄存器R1的内容减20，并将结果保存在R2中。能做到吗? 如果能，写出来；如果不能，请解释为什么。

5.12 在执行完指令ADD R2, R0, R1之后，我们发现R0[15]和R1[15]相同，但与R2[15]不同。已知R0和R1包含的都是无符号 (UNSIGNED) 整数 (即0~65535)，那么在什么情况下，我们可以相信R2中的内容?

5.13 a. 怎样只使用一条指令就能将R2的内容移到R3之中?

b. LC-3没有提供减法指令。怎样只使用3条指令完成下面的运算：$R1 \leftarrow R2-R3$?

c. 能否只使用一条LC-3指令，且不影响任何寄存器的内容，基于R1的内容设置条件码?

d. 是否能通过一组指令序列的执行，使得最后3个条件码分别为N=1、Z=1、P=0? 请解释。

e. 写一条能够清除R2内容的指令。

5.14 LC-3没有提供对应于逻辑OR的操作码。换句话说，不存在能够执行OR操作的LC-3指令。但我们可以通过一组指令来完成等价的OR功能。以下4条指令的任务是将R1和R2的内容相或 (OR)，并将结果保存在R3中。试填写下面空缺的指令：

```
(1) 1001 100 001 111111
(2)
(3) 0101 110 100 000 101
(4)
```

5.15 阅读程序，请写出下面起始于x3100的程序结束后 (HALT)，寄存器R1、R2、R3、R4的内容。

地 址	数 据 值
0011 0001 0000 0000	1110 001 000100000
0011 0001 0000 0001	0010 010 000100000
0011 0001 0000 0010	1010 011 000100000
0011 0001 0000 0011	0110 100 010 000001
0011 0001 0000 0100	1111 0000 0010 0101
:	:
:	:
0011 0001 0010 0010	0100 0101 0110 0110
0011 0001 0010 0011	0100 0101 0110 0111
:	:
:	:
0100 0101 0110 0111	1010 1011 1100 1101
0100 0101 0110 1000	1111 1110 1101 0011

5.16 在以下情况中，LC-3的哪一种寻址模式是最合适的（可能有多个正确答案，所以请为你的答案做出解释）？

a. 读取从间距不超过 $\pm 2^8$ 的地址的内容；

b. 读取从间距超过 2^8 的地址的内容；

c. 读取一个连续存放的数组内容。

5.17 在LD指令的执行过程中，共有多少次读写内存的操作？LDI指令有多少次？LEA指令呢？我们说的操作包括指令周期的所有节拍。

5.18 假设当前PC指向一个LDR指令所在的地址。如果让LC-3处理该指令，则需要多少次内存访问？STI和TRAP呢？

5.19 LC-3的指令寄存器（IR）由16个bit组成，其中bit[8:0]表示LD指令的PC偏移值。如果我们对ISA做出修改，让bit[6:0]表示PC偏移值，则LD指令读取数据的可寻址范围是多大？

5.20 如果我们设计LC-3的ISA，让LD指令只能从距离增量PC不超过±32的地址读取数据，则LD指令中的PC偏移字段需要多少个bit？

5.21 LC-3的ISA最多支持多少个TRAP服务？请解释。

5.22 PC内容为x3010。以下内存单元的内容如下所示：

```
x3050:          x70A4
x70A2:          x70A3
x70A3           xFFFF
x70A4           x123B
```

下面三条指令的执行导致R6装入一个数值，问该数值是多少？

```
x3010           1110 1110 0011 0001
x3011           0110 1000 1100 0000
x3012           0110 1101 0000 0000
```

能否用一条指令就完成上面三条指令所完成的任务？

5.23 假设下面指令被装入x30FF：

```
x30FF           1110 0010 0000 0001
x3100           0110 0100 0100 0010
x3101           1111 0000 0010 0101
x3102           0001 0100 0100 0001
x3103           0001 0100 1000 0010
```

请问程序执行完之后，R2的内容是什么？

5.24 位于x3200的一条LDR指令，采用R4做它的基址寄存器。R4当前的内容是x4011。问该指令读取数据的最大可寻址地址是多少？再假设我们重新定义LDR的偏移值是0扩展的（而不是符号扩展），则该指令最大可寻址地址是多少？最小地址呢？

5.25 写一个LC-3程序，比较R2和R3的内容，并将最大值放入R1寄存器。如果两个数相同，则R1等于0。

5.26 给你一个任务，设计下一代的LC-3。要求是：为ISA增加16个新指令，并将寄存器组的数目从8个增加至16个。机器要具备1个字节的基本寻址能力，且内存寻址空间大小为64K字节。指令的大小仍然是16-bit。指令的格式如原来的16位格式一样，具有相同的5个字段，只是你可以适当地调整各个字段的宽度。问：

a. PC的宽度至少是多少位，才能保证访问整个地址空间？

b. 运算指令中，可表达的最大立即数是多少？

c. 如果我们需要TRAP指令提供128个操作系统服务程序，且通过将trapvector左移5位，直接形成这些服务程序的入口地址，试计算一下这些服务程序所需要占用的最小内存大小。

d. 如果在新的LC-3中，我们将寄存器数目从8个减至4个，但操作码数目仍然保持为16个，试问在新的机器上，ADD指令所能表示的最大立即数值是多少？

5.27 如果在5.3.5节的7行代码的例程执行之前，R2的内容为xAAAA。那么在7条指令的执行过程中，R2的内容共有多少种数值？列举出来。

5.28 事实上我们“确实”没有必要提供“load indirect（1010）”和“store indirect（1011）”指令，因为我们可以通过其他指令的组合来完成相同的任务。试用一组指令序列替换下面程序中的store操作（1011）。

```
x3000          0010 0000 0000 0010
x3001          1011 0000 0000 0010
x3002          1111 0000 0010 0101
x3003          0000 0000 0100 1000
x3004          1111 0011 1111 1111
```

5.29 LC-3有一个指令“LDR DR, BaseR, offset”。在这个指令被译码之后，则产生下面一些操作（或称为“微指令”（microinstruction））：

```
MAR ← BaseR + SEXT(Offset6) ; set up the memory address
MDR ← Memory[MAR] ; read mem at BaseR + offset
DR  ← MDR ; load DR
```

假设LC-3要引入一个新指令“MOVE DR，SR”，即将地址为SR的内存单元内容拷贝至地址为DR的内存单元。问：

a. MOVE指令并不是必需的，因为它能完成的工作通过LC-3现有的一组代码组合也可以完成（又称为仿真（emulate））。试写出（或仿真）“MOVE R0，R1”指令的LC-3代码；

b. 如果LC-3真的添加了MOVE指令，试写出它的微指令序列（从译码完成之后开始）。

5.30 下面表格显示的是LC-3内存的部分内容：

地　址	数 据 值
0011 0001 0000 0000	1001 001 001 111111
0011 0001 0000 0001	0001 010 000 000 001
0011 0001 0000 0010	1001 010 010 111111
0011 0001 0000 0011	0000 010 111111100

如果其中的条件跳转指令又跳转至x3100，那么，你能推测出有关R0和R1的什么信息？

5.31 如图所示是8个寄存器在x1000指令执行之前和执行之后的快照（snapshot），试通过寄存器

的变化对比，推测该指令是一条什么指令（填空）？

	执行前		执行后
R0	x0000	R0	x0000
R1	x1111	R1	x1111
R2	x2222	R2	x2222
R3	x3333	R3	x3333
R4	x4444	R4	x4444
R5	x5555	R5	xFFF8
R6	x6666	R6	x6666
R7	x7777	R7	x7777

0x1000: 0 0 0 1

5.32 假设在下面代码序列执行之前，条件码的值为N=0，Z=0，P=1。问执行之后，它们的值是多少？

```
x3050    0000 0010 0000 0001
x3051    0101 0000 0010 0000
x3052    0001 0000 0010 0001
```

5.33 如果下面代码执行之后R0的内容为5，试推测有关R5的内容会是怎样的。

```
x3000    0101 1111 1110 0000
x3001    0001 1101 1110 0001
x3002    0101 1001 0100 0110
x3003    0000 0100 0000 0001
x3004    0001 0000 0010 0001
x3005    0001 1101 1000 0110
x3006    0001 1111 1110 0001
x3007    0001 0011 1111 1000
x3008    0000 1001 1111 1001
x3009    0101 1111 1110 0000
```

5.34 基于图5-18画出的全部数据通路，指出图5-4中与NOT指令执行相关的功能单元。

5.35 基于图5-18画出的全部数据通路，指出图5-5中与ADD指令执行相关的功能单元。

5.36 基于图5-18画出的全部数据通路，指出图5-6中与LD指令执行相关的功能单元。

5.37 基于图5-18画出的全部数据通路，指出图5-7中与LDI指令执行相关的功能单元。

5.38 基于图5-18画出的全部数据通路，指出图5-8中与LDR指令执行相关的功能单元。

5.39 基于图5-18画出的全部数据通路，指出图5-9中与LEA指令执行相关的功能单元。

5.40 图5-19中所示是LC-3的局部图，试回答标识为A的信号的作用是什么？

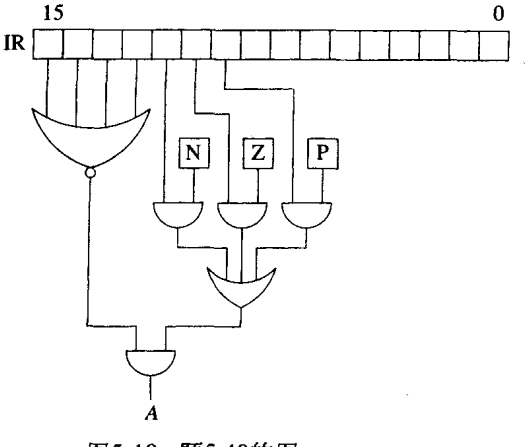


图5-19 题5.40的图

5.41 图5-20所示是LC-3的局部实现逻辑。

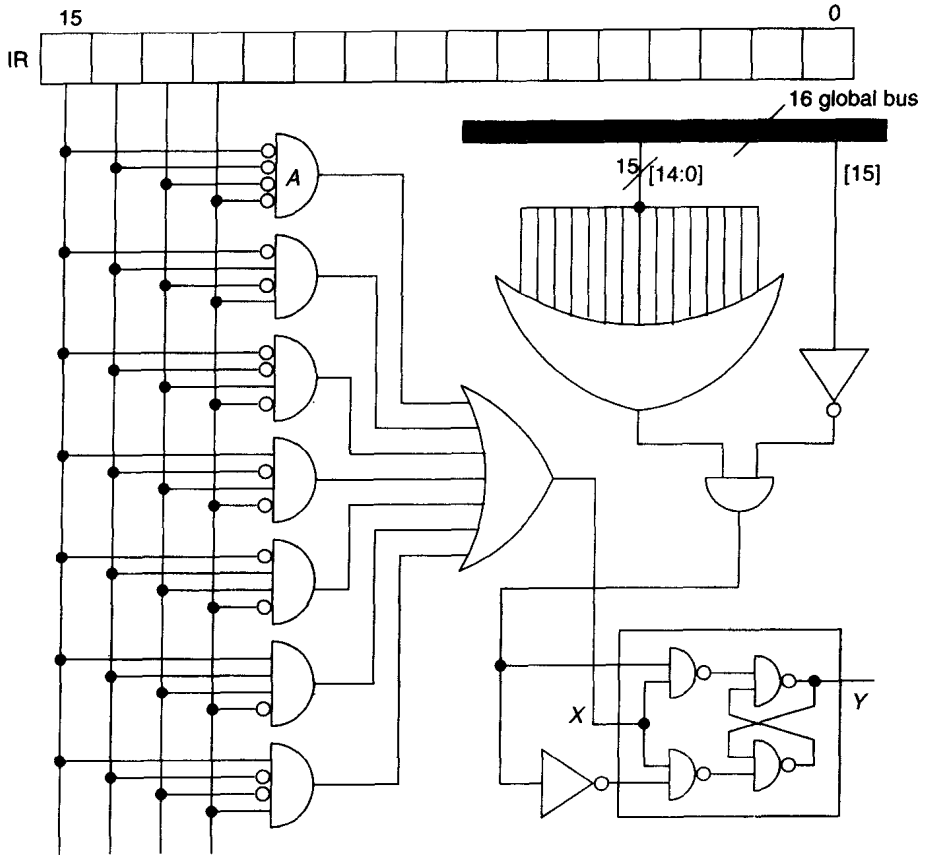


图5-20 题5.41的图

a. 信号*Y*提供的是什么信息?

b. 信号*X*是锁存器*D*的“门控”(gated)信号，请问X信号的产生逻辑是否存在错误?

5.42 LC-3开发组(macho-company)决定利用备用的操作码1101实现一个新指令。他们要在下面的各个候选中，挑选一个最合适的:

a. MOVE Ri,Rj；将Rj的内容拷贝至Ri；

b. NAND Ri,Rj,Rk；Ri是Rj和Rk的按位(bit-wise) NAND结果；

c. SHFL Ri,Rj,#2；Rj内容左移2位的结果存入Ri；

d. MUL Ri,Rj,Rk;

你的答案是什么(你觉得哪个更合适、更有必要)?

第6章 编　　程

现在我们开始学习编程方法，即通过编程方式让计算机为我们解决问题。本章我们要学两个内容：一是构建程序的方法；二是修补这些程序的方法。如果程序刚编写完就运行，难免会出现一些小问题（习惯上，我们称之为“臭虫”或bug），而消除这些错误的操作则被称做“调试”(debugging)。由于引发bug的因素很多，所以调试程序所花费的时间通常会远远大于编写代码所花的时间。

6.1 问题求解

6.1.1 系统分解

在第1章中，我们阐述了这样一个模型：为了让电子（即电子器件和电路）能帮我们解决问题，必须经历从自然语言的问题描述到使控制电路工作的多层转换。例如，首先用自己熟悉的语言（英语或其他你熟悉的语言，如意大利语、汉语（Mandarin）、印度语（Hindi）等）将问题描述清楚；然后，再将它转换成算法（algorithm），即具备了算法的三个定义特性的过程描述：（1）有限性（可终止的）；（2）确定性（每个步骤都是精确的、无二义性的）；（3）可计算性（每个步骤都是计算机可操作的）。

在20世纪60年代后期，提出了“结构化编程”（structured programming）的概念。其目的是使得多个程序员可以共同承担一个复杂问题的求解任务，即将问题系统地分解成多个独立的、足够小的模块（或单元），同时要求这些小单元具备可被每个程序员独立编程实现和运行的特性（即不依赖其他模块，即可独立运行以验证其正确性），我们又称该机制为“系统分解”（system atic decomposition），即一个大任务被分解成了多个子任务的集合。

下面我们将体会到，系统分解模型是用计算机编程求解复杂问题的一个重要方法。

6.1.2 三种结构：顺序、条件、循环

系统分解的原则是：将一个任务即一个完整的工作单元（如图6-1a所示），分割成多个更小的工作子单元，且这些工作子单元合起来所能完成的任务，和原先单个大工作单元是相同的。其思想是从一个很大的、复杂的任务开始，将其分解成若干个子任务；再将每个子任务进一步分解成更小的任务；如此反复，直到足够小，即小到方便编程实现。我们称这个过程为“逐步细化”。

分解后的子任务（模块）将基于特定的“结构”（construct）有机地结合在一起，完成更大的任务目标。最常见的基本构建方法是顺序、条件和循环这三种结构。

顺序结构（sequential construct）的使用场景如图6-1b所示，一个任务被分解成两个子任务，且两个子任务的执行关系是顺序的，即第1个子任务完成之后，再接着做第2个子任务。从另一种角度来解释，即意味着两个子任务都一定要执行，且第2个子任务一旦开始执行，就决不会再返回至第1个子任务。

条件结构（conditional construct）的使用场景如图6-1c所示，一个任务被分解成两个子任务，但两者之中有且仅有一个会被执行。具体是哪一个被执行，取决于“条件”的状态，即若条件为

“真”（true）则执行这个子任务，若条件为“假”（false）则执行另一个子任务。注意，其中的任何一个子任务都可能为空（vacuous），即什么也不做（do nothing）。无论如何，一旦其中的一个子任务执行完毕，程序就离开这里，进入后面的环节，即程序不会回头或重新测试条件。

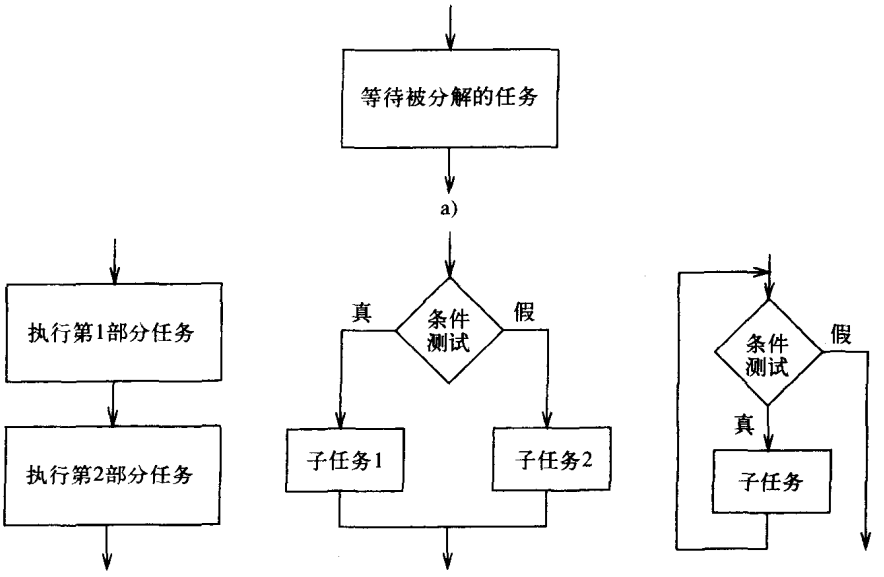


图6-1　结构化编程的基本构造式

循环结构（iterative construct）的使用场景如图6-1d所示，我们希望一个子任务能被重复执行多遍，不断循环，直到特定条件不再为“真”。换句话说，子任务每完成一次，将再次返回并重新测试条件。若条件满足（即为“真”），则再次执行同一任务；若不满足，则程序继续进入下一个环节。

图6-1a中，方框代表一个任务，顶部的箭头代表进入该任务（即任务开始），底部箭头代表任务结束。我们并没有提及方框内的任务究竟是个什么任务，但它总是可以被下面三种之一的结构所表示并替代的。注意，无论是图6-1a还是它的替代者（如图6-1b、c或d），它们都具备一个重要特征，即“*单入口*”（one entrance into the construct）和“*单出口*”（one exit out of the construct）。下面我们将通过例子解释系统分解的方法。

6.1.3　实现三种结构的LC-3指令

在讲解例子之前，我们先介绍LC-3中与三种结构分解方法相关的指令（如图6-2所示）。其中，图6-2b、c、d所示的控制指令分别对应于图6-1b、c、d所示的三种结构。

图6-2中的大写字母*A*、*B*、*C*、*D*代表LC-3代码的不同入口地址。比如在三个例子中都出现的“*A*”，代表的就是第一条执行指令的入口。

图6-2b描述的是顺序类型分解方法的控制流。注意，其中不需要控制指令的存在，因为PC会自动增量（即从B_1转至B_1+1），程序继续执行直到D_1。它不会返回第1个子任务。

图6-2c描述的是条件类型分解方法的控制流。首先是条件的产生，即设置某个条件码。位于地址B_2的条件跳转指令对该条件码做测试，如果条件为真，则PC被设置为地址C_2+1（即执行子任务1）。如果条件为假，则PC（跳转指令的取指令阶段已自动增量）从地址D_2+1读取指令（即执行子任务2）。子任务2的结束指令（地址C_2）是一条无条件跳转指令，跳转至D_2+1（注意：*x*对应的是子任务2的

指令数目，y对应的是子任务1的指令数目）。

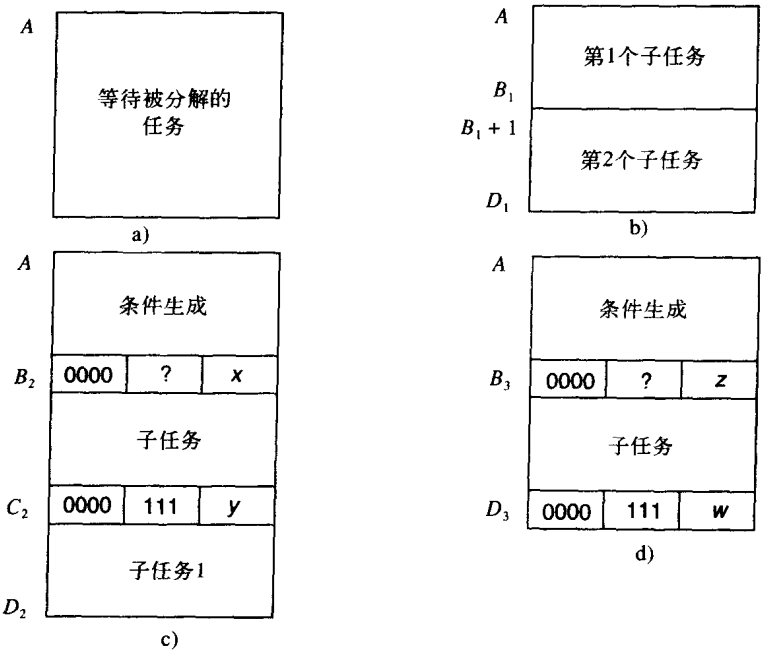


图6-2 与结构化编程相关的LC-3指令

图6-2d描述的是循环类型分解方法的控制流。与条件类型控制流相同的是，开始处是条件生成代码（设置条件码），然后是执行条件跳转指令；在这里，位于地址B_3指令中的条件位被设置了，即意味着如果条件为假，则条件跳转至D_3+1。相反，只要条件持续为真，则PC增量为B_3+1（执行惟一的子任务）。子任务的结束处（地址D_3）是一条无条件跳转指令，即设置PC为A，再次生成和测试条件码（注意：z对应于图6-2d中子任务的指令数目，w对应于图6-2d中整个分解代码的指令数目）。

下面我们开始例子的分析。

6.1.4 回顾字符数统计例子

回顾5.5节中对问题的描述："在一个文件中统计特定字符出现的次数。该特定字符是由键盘输入确定的；统计数结果将回显在屏幕上"。对这个自然语言问题描述的系统分解如图6-3所示，问题的简要描述如图6-3a所示。

为了更好地解决问题，我们必须首先明确问题的目标是什么，以及有哪些方法可以解决这个问题。本例的问题描述告诉我们，从键盘读取我们感兴趣的字符；然后检查某个文件的所有字符；统计我们感兴趣的这个字符在文件中出现的次数；最后将统计结果输出。

另外，我们还需要一个文件扫描机制和计数器（用于每次匹配成功时，增量该计数器）。

同时，我们还需要一些空间或位置，存放下面的信息（与扫描机制和计数器相关的）：

（1）来自键盘的输入字符。

（2）指向被扫描文件当前扫描位置（或偏移）的指针。

（3）文件中正在被扫描的字符值。

（4）被匹配字符出现的次数。

此外，我们还需要一种机制来判断文件是否结束。

因此，该问题很自然地（采用顺序结构）被分解为三个部分（如图6-3b中A、B、C所示）：A

阶段初始化，包括从键盘读取待匹配字符；B阶段统计文件中特定字符出现次数的具体操作；C阶段显示统计结果。

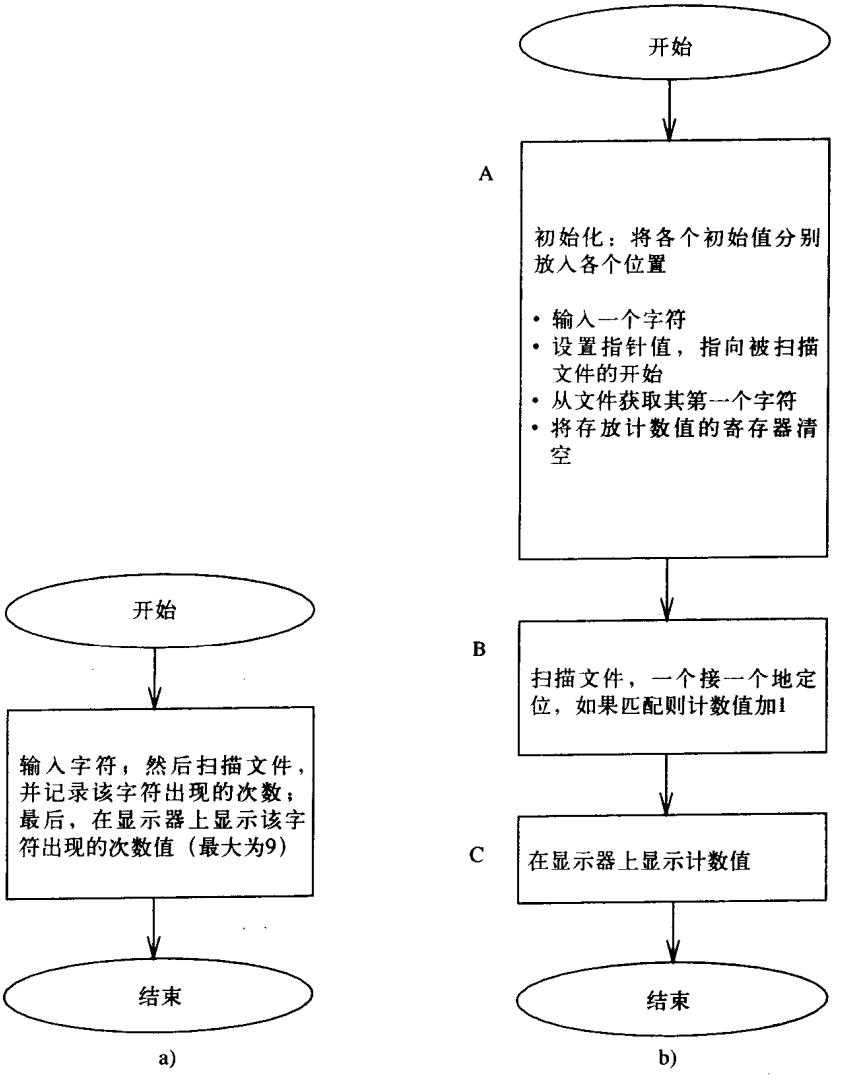


图6-3　字符数统计程序的逐步细化过程

前面的很多例子都表明了初始化过程的重要性。在程序的主体运行之前，必须确保其中的变量具有如我们所期望的初始值（否则会出现意想不到的运行错误）。这些初始值不会自己神奇地出现在GPR寄存器中（注意，无论变量事先是在内存还是在GPR中，最终在运算时刻都会转入GPR）。从另一种角度来看：初始化本身就是算法的一个明确步骤，即所有算法的第一个步骤。

在本算法中（见第5章），初始化工作包括：计数器清0，设置指针指向被检查文件的第一个字符，从键盘读取待匹配字符，以及从文件中读取第一个字符。总体上，这4个操作构成了算法的初始化步骤（如图6-3b中的模块A）。

如图6-3c所示，模块B又被进一步地分解成了“循环结构”。每循环一次，检查文件的一个字符，如此反复。B1描述的是每次循环所做的事情：测试字符并增量计数器（如果匹配的话），然后读入下一个字符，为下次循环做准备。如第5章所述，控制循环次数的方法有两种：哨兵法和计数器法。本程序采用的是哨兵法，即以ASCII字符EOT（End of Text）作为哨兵，一旦发现它，就意

味着文件结束了。换句话说，以文件读入字符是否是EOT字符来判断是否是合法字符。

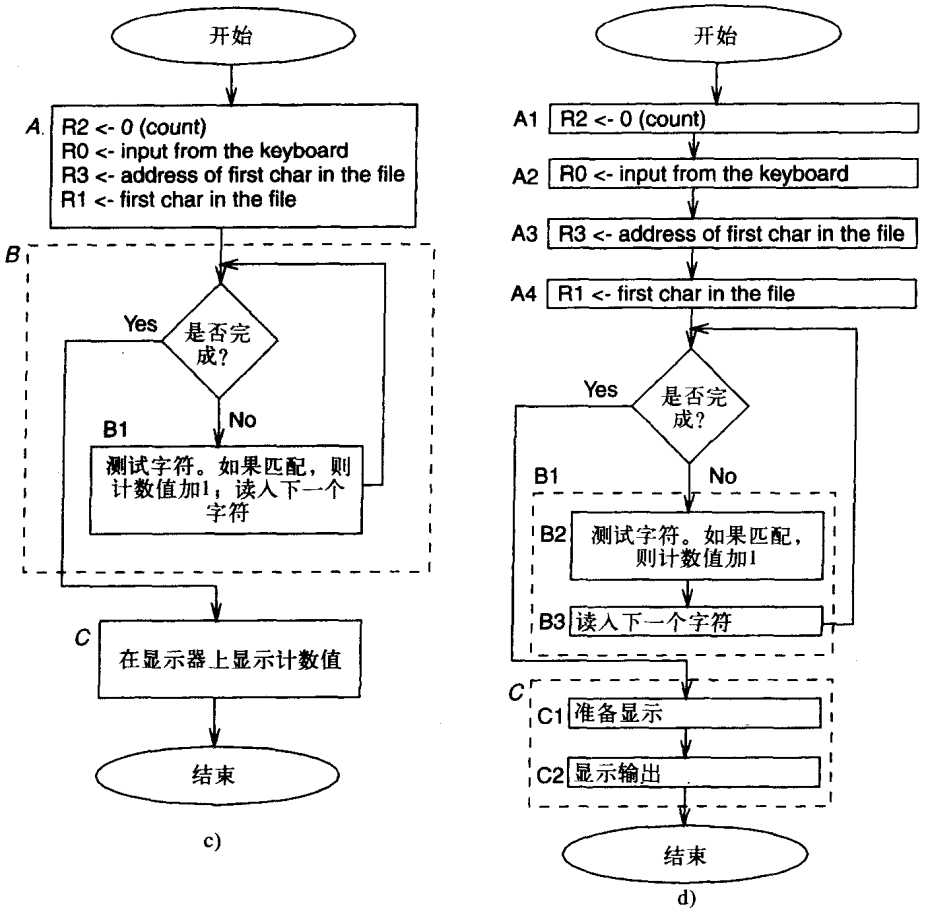


图6-3 字符数统计程序的逐步细化过程（续）

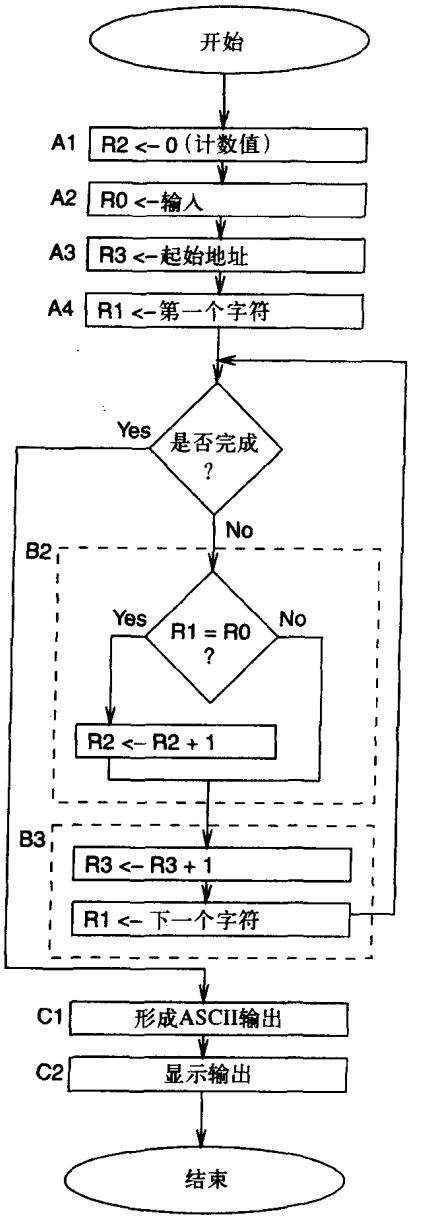
图6-3c给出了子任务A初始化过程的详细描述：LC-3的4个寄存器（R0、R1、R2、R3）分别扮演了算法中的4个重要元素：键盘输入字符、当前被测试字符（来自文件）、计数器、指向下一个待测试字符的指针。

图6-3d采用顺序结构分解了B1和C。其中，B1被分解为B2和B3，B2负责测试当前字符，如果匹配，则增量计数器，B3则负责获取下一个待检查字符。C也被分解成C1和C2，C1负责将计数器值从补码整数转换成ASCII码，C2负责输出ASCII码计数值。

图6-3e完成最后一步分解，即将B2细化成一个条件结构，将B3细化为一个顺序结构（即指针增量，并从文件中装入下一个待检查字符）。

最后一步是最容易的一步，是将图6-3e中每个细化后的方框即子任务写成LC-3的代码。注意，图6-3e本质上和图5-7是完全一样的（惟一不同的，是这个图的形成过程）。

在结束本话题之前值得一提的是，现实中你不可能一下子就能将每一样事情都搞明白，即设计出如图6-3e这样清晰的分解描述。如果遇到这种情况，不要放弃！你可以先从最容易的地方下手，然后逐渐清晰和细化。解决问题就如同猜迷，开始一定很没头绪，但尝试得越多就越清楚，最终就把问题给解决了。当你最终明白“已知哪些”（what is given）、“要做什么”（what is being asked for）以及“该怎样做”（how to proceed）之后，再回到方框图阶段（如图6-3a），然后重新开始问题的系统分解过程。



e)

图6-3　字符数统计程序的逐步细化过程（续）

6.2 调试

“调试”（debugging）可能是再常见不过的事情了（我们称之为“common sense”）。例如，你驱车前往一个从来没有去过的地方，并且在某个路口做了一个错误的拐弯。那么你该怎么办？最普通的“驾驶调试”（driving debugging）方法是漫无目的地乱逛，试图发现返回的道路。如果这样也失败了，你可能会听取旁人的意见，你不停地转圈，最终回到你“知道”的一个来路上；同样，你也可能会取出地图，按照方向指示，对比你的位置（如从窗外的路标上获知）和图上的标识，按照地图指示到达目的地（当然这对很多人来说有难度）。

程序调试和驾驶经历很相似，程序中存在的一个逻辑错误如同你拐错的一个弯。最简单的办

法是通过程序跟踪找到你的位置，跟踪内容包括指令执行顺序及每条指令的执行结果。通过对执行序列的跟踪，可以发现控制流方面的错误（如跳转到不该被执行的代码序列）；对每条指令执行结果的检查，可以发现程序中的逻辑错误。一句话，当你发现程序执行的任何行为和状态与所期望的不同时，那么就知道程序出现bug了。

解决问题的有效办法之一，是将程序分割成多个模块，并在每个模块执行结束时检查计算结果。事实上，结构化的编程方法（6.1节）更方便部署检查点，同样也能更系统地确定故障点，从而将注意力集中在最可能发生错误的地方（而不是漫无目的地查找）。

6.2.1 调试的基本操作

现在已有很多成熟的调试工具，在未来的几年里，你将接触其中的许多工具。在第15章中，我们将学习dbx（一个C语言源码级调试器）所提供的一些调试技术，但现在，我们还是想在机器结构层面（而不是类似C的高级语言层次），借助基本的交互式调试做些事情。所谓“交互式调试”，就是用户坐在键盘和显示器前面，可以向计算机发送命令。在我们的例子中，采用的调试器就是LC-3模拟器（simulator），我们可以通过该模拟器提供的菜单与之交互。

此时需要完成以下重要操作：

（1）对内存单元和寄存器赋值。

（2）执行程序的指令序列。

（3）在任意时刻中止执行。

（4）在任意执行步骤，查看内存和寄存器的内容。

这些操作虽然基本，但基于它们可以完成很多调试任务。

1. 赋值

直接对内存或寄存器赋值的能力非常重要。我们可以跳过之前的代码执行（即直接从需要调试的代码处开始），而直接将内存和寄存器的内容设置为我们期望的（如同执行过之前代码一样）。这样就使得我们可以孤立地测试部分代码，而不需要“麻烦”地执行之前的代码。假设程序中的某个模块首先从键盘获取输入，之后的模块将对该输入进行处理。假设我们希望单独测试第2个模块的正确性，而你已知第1个模块的运算结果，即键盘输入的ASCII码是存放在R0中的，那么你可以直接在R0中放入一个ASCII码，然后控制调试器直接从第2个模块开始执行（而不必从第1个模块开始执行）。

2. 控制指令序列的执行

所谓控制指令序列的执行包括：（1）可以从任意起点执行特定的指令序列（而不一定非要从程序入口处开始，或是执行全部程序）；（2）暂停程序的执行（以查看至此为止内存或寄存器中的程序运算结果）。为此，执行序列操作的三个基本操作是：直接运行（run）、单步或多步跳步执行（step）、设置断点（set breakpoint）。

运行命令就是启动程序的执行，直到必须停止为止，如遇到HALT指令或断点。

跳步命令可以使得程序执行特定数目的指令之后就停止。用户可以通过交互方式输入跳步命令及命令参数（如本次执行指令的数目）。如果数目为1，则执行一条指令后就停止。我们称这种只执行一条指令的方式为“单步执行”（single-stepping）。这种方式使得我们可以在执行一条指令后就查看该指令的执行结果。

设置断点命令可以强制程序执行到特定指令处停止。断点设置的实现机制是：将指定的断点地址添加到模拟器维护的一张“断点列表”中；随后，在每条指令的FETCH节拍，都将当前PC内容与该列表相匹配，如果发现匹配则中止执行。设置断点的效果就是让程序连续执行（而不需要跳步执行），直到PC值与列表中任意一个断点地址相同为止。它的用处是使得我们知道某段代码是否被

执行了。具体使用方法是先执行“断点设置”命令，然后执行“运行”命令，程序将完全自主地运行，直到断点处。用户可以查看此时的程序运行状态（如内存或寄存器的内容）。之后，如果我们希望程序下次不再在此断点处停止，可以通过“清除断点”（Clear Breakpoint）命令将其撤消。

3. 内容查看

当模拟器停止的时候，可以使用“查看”命令检查任意内存单元或寄存器的内容。

6.2.2 交互式调试器的使用

下面将通过4个例子，演示通过调试器发现程序错误的操作方法。我们设计了4种错误：

（1）不正确的循环控制，使得循环体出现了不正确的执行次数（或遍数）。

（2）不正确的load命令使用，原先是希望通过load指令（0010）将内存单元的内容装入寄存器，但由于错误地使用了“load effective address”指令（1110），使得内存的地址（而不是内容）被装入了寄存器。

（3）忽略了设置条件代码指令的存在，导致条件跳转指令的条件判断失误。

（4）忽略了输入值的所有可能性。

例6-1 没有乘法指令的乘法。

图6-4a所示程序的目的是将R4和R5中包含的两个正整数相乘。由于LC-3没有提供乘法指令，所以我们要写一个程序（而不是一条指令）来完成这个乘法任务。

如果将这个程序一条一条指令地看一遍，我们发现这个程序首先将R2清0（或者说将R2初始化为0），然后通过将R4的内容循环多次（R5）地累加至R0，从而完成乘法操作。其中，每做一次加法操作，R5的内容减1，直到R5=0的时候程序结束。

看起来这个程序应该没问题！但执行起来才发现，如果R4的初始值是10，R5初始值为3，程序的结果竟然是40（而不是30），出了什么问题？

我们的第一念头是跟踪一下这个程序。在做这件事情之前，我们发现程序假设R4和R5的内容都是正整数。我们通过“Set Value”命令将数值10放入R4，数值3放入R5。

另外，有必要对每条指令做注释。即每条指令对应的算法和指令的具体操作。这看起来很冗长乏味，对于一个有10000个指令的程序来说，每行都做注释帮助并不大，但是对于一段只有几行指令，且隐藏bug的程序片段，这样做会让情况更清晰。快速地浏览一段代码（quickly eyeballing a sequence of instruction）和认真地为其中的每一行代码做注释，之间是存在很大区别的。在图6-4a中，我们在每条指令的旁边都做了这样的注释。

图6-4b给出的是程序的跟踪记录。它是通过单步执行获得的。标识为PC的列所指示的是每条指令执行之初的PC值，标识R2，R4，R5的列分别表示的是每条指令执行之初（而不是执行之后的结果）的寄存器内容。通过对寄存器的检查，我们发现跳转条件码的内容是错误，即应该是如果R5为正数则跳转，而不是如x3203指令所做的在非负（注意，包括正数和零）情况下则跳转。如x3203的指令将导致一次额外的循环体执行，即数值10被加了4次而不是3次。

修改的方法很简单，即将x3203的指令替换为“BRp x3201”。



我们可能已发现，如果采用断点方式，能节省很多调试工作。即只要在x3203处设置断点，就可以让我们检查每次循环执行的结果，而不需要逐条地跟踪每条指令的执行。图6-4c显示的就是这种方法的跟踪结果，其中的每行对应一次循环。我们发现循环次数是4次而不是3次（本来应该的

循环次数)。

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2<–0
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	R2<–R2 + R4
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	R5<–R5–1
x3203	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	BRzp x3201
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT

a)乘法操作程序

PC	R2	R4	R5
x3201	0	10	3
x3202	10	10	3
x3203	10	10	2
x3201	10	10	2
x3202	20	10	2
x3203	20	10	1
x3201	20	10	1
x3202	30	10	1
x3203	30	10	0
x3201	30	10	0
x3202	40	10	0
x3203	40	10	–1
x3204	40	10	–1
	40	10	–1

b)乘法程序的跟踪

PC	R2	R4	R5
x3203	10	10	2
x3203	20	10	1
x3203	30	10	0
x3203	40	10	–1

c)断点方式的跟踪

图6-4 使用交互式调试技术发现例1中的错误

最后对这个例子做一下评述。在开始程序跟踪之前，先将R4和R5初始化为10和3。初始值的选择对程序测试相当重要。你必须保持头脑清醒。在此，程序声明了只对两个正整数操作有效，所以10和3是OK的。如果这个乘法程序说的是对所有的整数都有效，那么还需要多加几种如下的测试场景（初始值）：–6和3、4和–12、–5和–7。但还是漏了一个最重要的初始值——全0（即0和0），因为“所有”的整数包括了正整数、负整数还有0。这里要指出的是，一个正确的程序应该是在所有的情况下都保证能正确运行的程序，好的测试技术之一，就表现在能对变量做非常正规的初始化（即那些容易被大多数程序员所忽略的），这些初始值被俗称为“死角”。

例6-2 一列数的求和。

图6-5所示程序的目的是：将存储在从x3100开始的10个内存单元的数值加在一起，并将结果存放在R1。图6-6是从x3100开始的10个内存单元的内容。

程序的运算过程如下：

- 初始化（x3000~x3003）。位于x3000的指令对累计和变量R1清0，位于x3001的循环控制变量（计数累加数数目）被初始化为#10，程序每循环一次减1，直到R4为0停止循环；位于x3003的基址寄存器R2被初始化为数据列的起始地址x3100。
- 循环体。从这里开始，每循环一次，就有一个数值被装入R3（x3004），基址寄存器随后被增量（x3005），R3的内容被加入R1（x3006），即运行和（running sum）存放处，计数器减1（x3007），P位被测试。如果为真，则PC被设置为x3004，开始下一轮循环（x3008）。
- 停止。10次之后，R4的内容变为0，P位也相应为0，跳转不再发生，程序结束（x3009）。

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	R1<–0
x3001	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	R4<–0
x3002	0	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0	R4<–R4 + 10
x3003	0	0	1	0	0	1	0	0	1	1	1	1	1	1	0	0	R2<–M[x3100]
x3004	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	R3<–M[R2]
x3005	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2<–R2 + 1
x3006	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	R1<–R1 + R3
x3007	0	0	0	1	1	0	0	1	0	0	1	1	1	1	1	1	R4<–R4–1
x3008	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	1	BRp x3004
x3009	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT

a)10个整数相加的LC-3程序

PC	R1	R2	R4
x3001	0	x	x
x3002	0	x	0
x3003	0	x	#10
x3004	0	x3107	#10

b) 相加程序的前4条指令执行结果

图6-5 使用交互式调试发现例2中的错误

Address	Contents	Address	Contents
x3100	x3107	x310A	x0000
x3101	x2819	x310B	x0000
x3102	x0110	x310C	x0000
x3103	x0310	x310D	x0000
x3104	x0110	x310E	x0000
x3105	x1110	x310F	x0000
x3106	x11B1	x3110	x0000
x3107	x0019	x3111	x0000
x3108	x0007	x3112	x0000
x3109	x0004	x3113	x0000

图6-6 例2中内存x3100~x3113的内容

初看这个程序应该是没问题的。但是，我们执行这个程序并检查R1的内容（x3100~x3109的求和），竟然是x0024（而不是x8135）。哪里出错了呢？

使用调试器看看。图6-5b是前4条指令（初始化操作）执行的情况。注意，x3003指令执行之后，R2的内容是x3107（而不是x3100）。问题出在操作码0010，装入寄存器R2的是x3100的“内容”而不是x3100的“地址”。我们犯的错误是：应该使用操作码1110将x3100的地址（而不是内容）装入R2。修改操作为：将操作码0010替换成1110，之后运行成功。

例6-3 查看内存区域是否包含数字5。

图6-7程序的目的是检查从x3100开始的10个内存单元中是否包含有数值5，如果有则设置R0=1，如果没有则设置R0=0。

程序的运行过程如下：

• 初始化（x3000~x3005）。这6条指令将寄存器初始化为R0=1，R1=–5，R3=10，每个寄存

器的初始化都是先将寄存器清零（寄存器内容和0做“与”（AND）操作），然后将要添入的值与寄存器本身相加（ADD）。例如，位于x3003的指令将-5和R1（已为0）相加，结果存回R1。

- 位于x3006的指令初始化R4的内容为测试内存的起始地址x3100，位于x3007的指令将x3100的“内容”装入R2。
- 判断（x3008-x3009）。判断R2的内容是否是5的方法，是将R2和-5相加，如果结果为0，则跳转至x300F（找到了！）。由于R0的初始值为1（即意味着发现了数值5的存在），因此这里无需再设置R0的内容为1。
- 准备下一轮测试。位于x300A的指令增量R4，准备装入下一个被测试内容；x300B将R3减1，即还有多少数值未被测试；x300C借助R4做指针，真正装入下一个数值至R2；x300D则根据R3的内容是否为0，决定是否跳转回x3008重新此过程。如果R3=0，则测试结束。所以同时还要设置R0=0（X300E）。随后程序结束（x300F）。

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	R0<-0
x3001	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	R0<-R0 + 1
x3002	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	R1<-0
x3003	0	0	0	1	0	0	1	0	0	1	1	1	1	0	1	1	R1<-R1-5
x3004	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	R3<-0
x3005	0	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0	R3<-R3 + 10
x3006	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1	R4<-M[x3010]
x3007	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	R2<-M[R4]
x3008	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1	R2<-R2 + R1
x3009	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	BRz x300F
x300A	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	R4<-R4 + 1
x300B	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	R3<-R3-1
x300C	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	R2<-M[R4]
x300D	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	0	BRp x3008
x300E	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	R0<-0
x300F	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT
x3010	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	x3100

a)检测5的存在的LC-3程序

PC	R1	R2	R3	R4
x300D	-5	7	9	3101
x300D	-5	32	8	3102
x300D	-5	0	7	3013

b) 设置断点x300D跟踪例3程序的执行

图6-7 使用交互式调试发现例3中的错误

我们开始运行这段程序，事先准备好样本数据且在地址x3108处特意放置了一个5。但程序结束的时候，我们发现R0=0，即意味着在x3100~x310A中并没有5。

出什么问题了？再次调试运行一遍，将断点设置在x300D，跟踪结果如图6-7b所示。

第一次运行至断点时，PC=x300D，且x3100的内容已被测试过，R2也装入了数值7（x3101的内容），R3=9，表明还有9个数据未被测试，R4的内容对应于刚装入R2的数值所在的地址（即x3101）。

第二次运行至断点时，仍然是PC=x300D，R2装入了数值32（x3102的内容），R3表明还有8个数据未被测试。

第三次PC又变成x300D的时候，R2的装入数值是0（x3103的内容），R3指示还有7个数据未被测试。但正是这个存放在x3103的数值0，导致x300C的load指令将条件位P清零了，进一步造成x300D的条件跳转没有发生，即循环“中止”了！随后R0被置0（x300E），程序结束（x300F）。

错误的症结在于，我们在x300B指令（计数值减1）和x300D指令（即判断计数如果不为0就跳转回x3008继续测试）之间插入了一个“故障”指令（x300C的load指令）。由于load指令会改变条件码，这使得x300D的条件跳转指令是否跳转完全取决于R2寄存器，而不是R3寄存器。解决这个问题的方法是，删除x300C指令，并修改x300D指令的目的地址为x3007，则程序运行正常。

例6-4　查找字中的第一个1。

最后的这个例子中，隐藏的是最难发现的bug之一。图6-8程序的任务是检查一个内存单元的内容（即某个地址中存放的1个字），逐个bit地检查（从左至右）第一个出现的位值为“1”的bit，并将该bit在该字（word）中的位置保存在R1中。如果没有发现为“1”的位（即全部位为0），则设置R1为“−1”。例如，如果被检查单元的内容是“0010000000000000”，则程序结束时R1=13。如单元的内容是“0000000000000100”，则程序结束时R1=2。

程序的运行过程如下：

- 初始化（像所有的程序一样）。x3000和x3001的指令负责R1的初始化，方法与前面的例子一样，先AND（清0）后ADD，完成对R1的赋值。在此，R1被初始化为15。x3002负责向R2传入x3100内存的内容（将被检查的数值），这里采用的是“间接load”指令，即从x3009获得存放该数值的地址x3100。
- 第1次测试。x3003测试的是该数值的最高位是否为1（等价于判断该数是否为负数），如果是1则跳转至x3008（即程序的结束点），而此时R1=15；如果是0则不跳转，R1减1（x3004），指向下一个被测试的位bit[14]。
- 左移和第2次测试。x3005让R2和自己相加，结果存回R2。事实上这等价于R2乘2，又等价于将R2的内容左移1位，这样bit[14]的内容也就被左移到了bit[15]，使得条件跳转指令可以通过判断该数是否为负，来判断该位的内容。然后，由x3006执行对bit[14]（现在是bit[15]）的内容判断。而如果该位是1，则跳转至程序结束且R1=14。如果该位是0，则继续x3007的执行，x3007无条件地跳转至x3004，重复测试过程。
- 下一次测试。循环体的工作就是每次完成一个bit的测试，即R1减1（x3004），指向下一个被测试位，R2内容左移1位（x3005），然后测试新的bit[15]内容（x3006）。

以上过程重复着，直到发现了第1个“1”。大多数情况下，该程序运行正常。但是如果用我们提供的测试数据，它就出问题了：永远不停止！

图6-8b是我们的测试跟踪过程，断点设置为x3007。跟踪记录显示：每次PC都是x3007，而R1内的数值每次递减1。其中的原因是：R1减1，R2内容左移，bit测试总为0，所以程序继续而不终止。因而R1的内容不断地变化，14、13、12、11、10、9、8、7、6、5、4、3、2、1、0、−1、−2、−3、−4，如此反复。

出现问题的原因是，我们在地址x3100中存放的是x0000。即这个字中永远不会出现“1”，但对于这个程序，至少要存在一个1才可能工作正常。由于x3100的内容全部为0，x3006的跳转永远不会发生，所以程序总是在永不停止地循环着（x3004、x3005、x3006、x3007、x3004、…）。我

们称x3004至x3007的这段序列是一个“环”（loop），因为程序无法打破这个环，所以我们又称它为“无限循环”（infinite loop）。

这是最难检测的一个错误（它大部分时间的运行都是正确的），但同时它们也是最重要的。一个程序在大多数时间里的工作都是正常的，这是不够的；我们必须保证它时时刻刻工作正常，而不管我们交给它的数据是什么。在本书的后面内容中，会介绍更多的类似问题。

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	R1<-0
x3001	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	1	R1<-R1 + 15
x3002	1	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	R2<-M[M[x3009]]
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	BRn x3008
x3004	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1<-R1 - 1
x3005	0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	R2<-R2 + R2
x3006	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	BRn X3008
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	BRnzp x3004
x3008	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT
x3009	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	x3100

a)查找字中第1个为“1”的位的LC-3程序

PC	R1	PC	R1
x3007	14	x3007	4
x3007	13	x3007	3
x3007	12	x3007	2
x3007	11	x3007	1
x3007	10	x3007	0
x3007	9	x3007	−1
x3007	8	x3007	−2
x3007	7	x3007	−3
x3007	6	x3007	−4
x3007	5		

b)设置断点x3007跟踪例4程序的执行

图6-8 使用交互式调试发现例4中的错误

6.3 习题

6.1 请问：一个不具备“算法”特性的过程，能否采用结构化编程的三个基本结构来构建程序？如果可以，请给出一个例子（提示：有关“算法”的定义，参考第1章）。

6.2 LC-3没有提供减法指令。如果要对两个数做减法操作，你必须写一个函数来完成它。给出两个整数减法操作的系统分解过程。

6.3 回顾前几章的“machine busy”例子。假设地址x4000包含了一个整数（范围为0~15），指示刚变为“忙”（busy）状态的机器编号。再假设x4001的内容告诉我们哪些机器正处于忙（busy）状态，哪些机器正处于空闲（idle）状态。试写一个LC-3的机器语言程序，根据x4000的内容（刚处于busy的机器编号）设置x4001的对应位。

例如，假设该程序执行之初x4000的内容为x0005，x4001的内容为x3101，则程序执行之后，x4001的内容应该变为x3121（提示：LC-3没有提供OR操作，你需要使用AND和NOT的

组合操作实现OR操作)。

6.4 试写一个LC-3程序，比较R1和R2的内容并设置R0。如果R1=R2，则R0=0；如果R1>R2，则R0=1；如果R1<R2，则R0=−1。

6.5 下面的两个乘法算法哪一个更好，为什么？88×3=88+88+88或3+3+3+…+3。

6.6 基于习题6.3和6.4的答案，写一个高效率的两个整数的乘法程序，并将结果保存在R3。给出该问题的系统分解过程，从问题描述到最后的程序（提示：所谓的“高效率”，参考习题6.5的讨论)。

6.7 阅读以下LC-3代码，请问该程序的任务是什么？

x3001	1110	0000	0000	1100
x3002	1110	0010	0001	0000
x3003	0101	0100	1010	0000
x3004	0010	0100	0001	0011
x3005	0110	0110	0000	0000
x3006	0110	1000	0100	0000
x3007	0001	0110	1100	0100
x3008	0111	0110	0000	0000
x3009	0001	0000	0010	0001
x300A	0001	0010	0110	0001
x300B	0001	0100	1011	1111
x300C	0000	0011	1111	1000
x300D	1111	0000	0010	0101
x300E	0000	0000	0000	0101
x300F	0000	0000	0000	0100
x3010	0000	0000	0000	0011
x3011	0000	0000	0000	0110
x3012	0000	0000	0000	0010
x3013	0000	0000	0000	0100
x3014	0000	0000	0000	0111
x3015	0000	0000	0000	0110
x3016	0000	0000	0000	1000
x3017	0000	0000	0000	0111
x3018	0000	0000	0000	0101

6.8 6.1.4节的字符数统计程序中，为什么一定要初始化R2？换句话说，如果我们将R2←0的这行代码删除，该程序可能会出什么问题？

6.9 采用循环结构，写一个LC-3程序，在屏幕上显示100个Z字符。

6.10 采用条件结构，写一个LC-3程序，判断存放在R2中的数值是否是“奇数”(odd)。

6.11 写一个LC-3程序，将地址A到地址B之间的所有内存单元的内容加1。假设这些内存单元的内容已被初始化为有意义的数值，而地址A和B的具体地址值分别从x3100和x3101内存单元获得（提示：数据的访问为间接寻址方式)。

6.12 a. 写一个echo程序，即将键盘输入的字符回显在屏幕上。假如刚才的键盘输入为字符R，则程序马上将字符R输出在屏幕上。

b. 将a的问题扩展一下，每次回显一行（而不是一个字符)。例如，用户输入“The quick brown fox jumps over the lazy dog.”，则程序一直等待（无显示输出)，直到用户输入回车键（Enter键，ASCII码为x0A)，才将进行一次性输出。

6.13 已知通过数值的自身相加操作可以实现对该数的左移1位操作。例如，二进制数0011自己加自己，结果为0110（即等价于左移1位)。但是右移1位的操作却不是简单的事情。设计一个

LC-3程序，将x3100的内容右移1位。

6.14 阅读下面的机器语言程序：

x3000	0101	0100	1010	0000
x3001	0001	0010	0111	1111
x3002	0001	0010	0111	1111
x3003	0001	0010	0111	1111
x3004	0000	1000	0000	0010
x3005	0001	0100	1010	0001
x3006	0000	1111	1111	1010
x3007	1111	0000	0010	0101

问：R1的初始值为多少的时候，才能使R2的最终值为3？

6.15 下表显示的是x3010程序执行之前（before）和执行之后（after）的内存和寄存器内容变化。你的任务是：判断存放在x3010的指令是什么指令（注意，以下的信息足以判断出该指令是什么，所以“细心+耐心”）？

	Before（之前）	After（之后）
R0:	x3208	x3208
R1:	x2d7c	x2d7c
R2:	xe373	xe373
R3:	x2053	x2053
R4:	x33ff	x33ff
R5:	x3f1f	x3f1f
R6:	xf4a2	xf4a2
R7:	x5220	x5220
...		
x3400:	x3001	x3001
x3401:	x7a00	x7a00
x3402:	x7a2b	x7a2b
x3403:	xa700	xa700
x3404:	xf011	xf011
x3405:	x2003	x2003
x3406:	x31ba	xe373
x3407:	xc100	xc100
x3408:	xefef	xefef
...		

6.16 地址x3000~x3006之间存放了一段LC-3程序。程序从x3000开始执行。我们在执行过程中跟踪记录MAR的所有变化值，获得以下序列值，我们称这个序列值为“跟踪记录”。

MAR Trace

x3000
x3005
x3001
x3002
x3006
x4001
x3003
x0021

下表显示的是内存x3000~x3006的内容。你的任务是填充表中的空格（0或1），结合上面的MAR跟踪记录信息。

x3000	0	0	1	0	0	0	0									
x3001	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1
x3002	1	0	1	1	0	0	0									
x3003																
x3004	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1
x3005	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
x3006																

6.17　下表显示的是位于x3210的LC-3指令在执行之前和执行之后的内存和寄存器内容变化。你的任务是：判断存放在x3210的指令是什么指令（注意，以下的信息足以判断出该指令是什么，所以“细心+耐心”）？

	Before（之前）	After（之后）
R0:	xFF1D	xFF1D
R1:	x301C	x301C
R2:	x2F11	x2F11
R3:	x5321	x5321
R4:	x331F	x331F
R5:	x1F22	x1F22
R6:	x01FF	x01FF
R7:	x341F	x3211
PC:	X3210	X3220
N:	0	0
Z:	1	1
P:	0	0

6.18　LC-3没有提供除法指令。程序员必须写一个函数才能实现除法操作。给出求解两个“正整数”除法问题的系统分解。要求：LC-3程序从x3000开始执行，被除数位于x4000，除数位于x4001，商存放在x5000，余数存放在x5001。

6.19　有时我们希望将一个消息（message）加密（如不希望好奇者偷看）。消息是由一串ASCII字符组成的，每个ASCII字符占据一个内存单元（且bit[15:8]都为0）；存放是连续的（即相邻的消息字符之间不会有间隔，也不会倒序），且该字符串的结束必定是x0000。

一个没有上过这门课的学生写下这样一个程序：从x4000开始，将每个字符都加4，然后存放在x5000开始的内存中。例如，假设x4000开始的内容是“Matt”，则加密后存放在x5000的内容应该是“Qeyy”。但是，他/她的代码中存在4个bug。找出它们并予以纠正。

x3000	1110	0000	0000	1010
x3001	0010	0010	0000	1010
x3002	0110	0100	0000	0000
x3003	0000	0100	0000	0101
x3004	0001	0100	1010	0101
x3005	0111	0100	0100	0000
x3006	0001	0000	0010	0001
x3007	0001	0010	0110	0001
x3008	0000	1001	1111	1001
x3009	0110	0100	0100	0000
x300A	1111	0000	0010	0101
x300B	0100	0000	0000	0000
x300C	0101	0000	0000	0000

6.20　重做习题6.18，将范围扩展为“所有整数”，而不仅仅是“正整数”。

第7章 汇编语言

可能你早已对0和1有些厌烦了，因为你要记住0001代表ADD，1001代表NOT，等等。如果我们能提供一种符号名方式（而不是16位的数字）来代表内存的地址，你或许不会反对吧？或许你希望我们还能提供一种更简明的方式来描述指令（而不需要记住其各个bit或字段的定义）。好的，我们马上就要开始介绍这样一种语言。

本章我们将介绍汇编语言（assembly language），我们可以将它理解为是上述目标的各种实现机制之一。

7.1 汇编语言编程——更上一层

回顾我们在第1章的图1-6中提到的层次转换问题。算法经过多级转换，最终演变成某种机械语言（mechanical language，与自然语言相比，显得很机械）。这个机械语言可以类似我们在第5章介绍的某个特定计算机的机器语言。换句话说，如果一个程序中的每条指令都符合且对应某个计算机ISA的定义，我们称该程序是用“XXX计算机的机器语言”（machine language）编写的。

但从另一方面来说，我们可以让机器语言变得友好些。机械语言也可以分为高级的（high-level）和低级的（low-level）两类。高级语言如C、C++、Java、Fortran、COBOL、Pascal等，约有上千种，它们的指令几乎（但不完全）都模仿了自然语言的语句结构（尤其是英语）。这意味着，一旦你学会了怎样用C或Pascal、Fortran等语言为一种机器结构（ISA）编写程序，那么，为另一种ISA编写程序则是大同小异。

高级语言编写的程序在能够运行之前，必须先被翻译成特定ISA机器语言格式的代码。通常，高级语言的一条语句会被翻译成多条机器语言指令。关于高级语言，我们将在第11章介绍C语言，在第12~19章，介绍每个C语句和它对应的LC-3指令代码之间的映射关系。但是，本章只计划在第5章的基础上前进一点点。

这个进步就是从机器语言升级到汇编语言。汇编语言属于低级语言。注意，不要试图将低级语言的一个指令和英语中的一个句子相对应。相反，每条汇编指令通常对应了ISA中的一个指令。高级语言通常是“ISA无关的”，而低级语言则是“ISA相关的”。事实上，对于特定的ISA，通常只存在一种对应的汇编语言。

汇编语言的目的是为了让编程过程更加友好（相比第5章介绍的机器语言），但它保留了和机器语言同样的细节控制能力。所不同的是，我们不再需要记住类似“0001是什么操作码”、“1001是什么操作码”，以及“内存单元0011111100001010存放了什么内容”这样枯燥的描述。在汇编语言中，我们采用的是“助记符”（mnemonic）方式来表示ADD和NOT之类的操作码，而内存地址则被类似SUM、PRODUCT之类的“符号名”（symbolic name）所替代。我们看到，SUM和PRODUCT之间的区分要比0011…010和0011…101之间的区分容易，且清晰得多。我们称这种命名方式为“符号地址”（symbolic address）。

我们将看到（从第11章开始），高级语言（如C语言）将使得编程工作更加友好，但同时也将失去对计算机的细节控制权（因为在高级语言中，将不会也不方便提供与ISA相关的控制指令）。

7.2 一个汇编程序

我们将通过一个程序例子开始LC-3汇编程序的讲解。图7-1程序的目的是将最初存放在

NUMBER的整数乘以6，实现方法是将NUMBER这个数反复地加6次。假设，存放在NUMBER的整数是123，则该数乘以6的乘积等于123+123+123+123+123+123。

该程序包含21行，我们在每行代码的开始添加了行号（源程序中是没有行号的，在此只是为了讲解时方便指示，这是教科书的惯例）。

```
01  ;
02  ; Program to multiply an integer by the constant 6.
03  ; Before execution, an integer must be stored in NUMBER.
04  ;
05          .ORIG   x3050
06          LD      R1,SIX
07          LD      R2,NUMBER
08          AND     R3,R3,#0        ; Clear R3. It will
09                                  ; contain the product.
0A  ; The inner loop
0B  ;
0C  AGAIN   ADD     R3,R3,R2
0D          ADD     R1,R1,#-1       ; R1 keeps track of
0E          BRp     AGAIN           ; the iterations
0F  ;
10          HALT
11  ;
12  NUMBER  .BLKW   1
13  SIX     .FILL   x0006
14  ;
15          .END
```

图7-1 一个汇编程序代码

在程序中，有10行的首字符是“;”，这表明这些行是只供人（而不是机器）阅读的，又称“注释行”；有7行汇编指令（06、07、08、0C、0D、0E、10）是将被翻译成LC-3机器指令的，即真正将要运行的；其他4行（05、12、13、15）是伪操作（pseudo-ops），是程序员传递给翻译程序（即汇编器）的信息，用于提示帮助翻译过程。对于汇编语言，该翻译程序被称做“汇编器”(assembler)，翻译过程则被称做“汇编”(assembly)。

7.2.1 指令

与机器指令（如LC-3 ISA的16位0/1序列）不同的是，汇编语言指令的格式包括4个部分，如下所示：

```
LABEL   OPCODE   OPERANDS   ; COMMENTS
```

其中，LABEL和COMMENTS两个字段是可选的。

1. 操作码和操作数

在一个汇编指令的所有字段中，操作码（OPCODE）和操作数（OPERAND）两部分是**必需**的(mandatory)。一个指令必须有操作码（做什么事情）及操作数（被操作的对象）。这一点应该不会奇怪的，第5章的LC-3 ISA机器指令也是如此。

其实，OPCODE就是对应LC-3指令的一个符号名（是字符串表示的，而不是0/1串表示的）。这样做的目的是方便记忆，如ADD、AND、LDR总比0001、0101、0110更容易记忆。图5-3（以及图A-2）列举了LC-3指令的15个OPCODE。

操作数的数目取决于具体的操作。例如ADD指令（第0C行）需要3个操作数（两个相加的源操作数，1个存放结果的目的操作数）。所有这3个操作数都必须显式地在指令中标识出来：

```
AGAIN    ADD    R3,R3,R2
```

寄存器R2和R3是被相加的源操作数，结果被存放在R3寄存器中。我们用R0、R1、…R7代表8个寄存器。

LD指令（第07行）需要两个操作数（被读入数据所在的内存地址和存放该数据的寄存器）。在后面的例子中，我们将直接使用标号来表示内存地址。例如，本例中的NUMBER就是代表内存地址的标号。

```
LD   R2, NUMBER
```

我们在5.1.6节中介绍过，操作数的获取可以来自寄存器、内存或直接来自指令（立即数）。在寄存器操作数方式下，寄存器的表示是显式的（如第0C行中的R2和R3）；在内存操作数方式下，内存地址被显式地表示为符号名（如07行的NUMBER和06行的SIX）；在立即数操作数方式下，其数值被显式地表示出来（如08行的数值0）。

```
AND  R3, R3, #0 ; Clear R3. It will contain the product.
```

立即数的标识符代表了该数值的基，即标识符“#”代表十进制（即base=10），“x”代表十六进制，“b”代表二进制。有时候，即使没有标识符也不会出现二义性，如3F0A（它一定是个十六进制数），尽管如此，我们还是把它写成x3F0A；有的时候则会出现二义性，如1000、x1000等价于十进制数4096，而b1000则代表十进制数8，#1000则代表十进制数1000。

2. 标号

标号（label）是指向内存单元的一个符号名，它可以在程序中直接引用。LC-3汇编语言中，一个标号可以包含1~20个字符（如大写或小写的字母或数字），但首字符必须是字母。NOW、Under21、R2D2、C3PO等都是合法的标号例子。

显式访问内存单元的情况有两种：

（1）该单元的内容是指令，该单元地址是跳转指令的目标（如0C行的AGAIN）。

（2）该单元的内容是load或store指令访问的数值（如12行的NUMBER和13行的SIX）。

标识AGAIN被跳转指令访问的例子，如0E行所示：

```
BRp  AGAIN
```

如果之前的ADD R1,R1,#-1的运算结果是正数（即P位被置1），则跳转指令跳转到由AGAIN显式指向的位置，即下一轮循环的开始。

标识NUMBER被load指令引用的例子如07行所示，即存放在NUMBER指向地址包含的数值被装入R2寄存器。

如果一个内存单元从来不会被程序引用，则没有必要为其做标号。

3. 注释

注释（comment）是那些仅供人阅读的信息。换句话说，这些信息不会影响程序的翻译过程（事实上LC-3汇编器根本就不处理这些信息）。在程序中它们被分号（;）隔开。在一行指令中，分号意味着本行中分号之后的内容都是注释，这些内容都将被汇编器忽略。如果程序中一行的第一个非空字符是分号，则整行被忽略；如果分号是跟在一条指令之后，则分号之后的内容被忽略。

注释的目的是方便程序的阅读。它用来对一条或一组晦涩的指令代码做注解。如08和09行的注释“Clear R3; it will contain the product”告诉读者的是：第8行的指令是为乘法运算做初始化操作，即将R3清零。因为，对一个程序员来说，他/她很清楚“AND R3,R3,#0”这条指令的作用是什么；但是两年后，在他又完成了30000行代码之后，他/她必然完全忘记了这条指令的当初目的（当然通过上下文代码阅读，还是会推理出这条指令的作用的）。还有可能的情况是，两年后这个程序员已经离开公司了，而公司此时要修改/升级该程序。如果任务分给了一个从来没有读过这个代码的人，那么注释将有助于对程序的理解。

值得一提的是，“不要滥用注释”。注释提供的应该是更深的“洞察力”，而不是对显而易见事实的重述。这样做的原因有两个：一是重述浪费时间；二是过多不重要的注释将掩盖那些“重要”的注释信息，让程序变得杂乱无章。例如，第0D行的注释“Decrement R1”就属于废话，它没有提供任何比指令本身更多、更有用的信息，反而把程序版面搞得更杂乱。

注释的另一个目的，同时也是一行中额外的空格的目的，就是使程序更加直观，易于理解。因而，这些注释的作用是将程序的各个代码片段分隔开来，提高程序的可读性。即将相关的指令或代码（即一起工作并产生一个结果的）连续排放，例如注释行0B和0F的作用，就是将0C~0E的代码段与程序其他部分区分开来，而0B和0E行中，除了一个分号别无其他。

这些额外的空格除了起到程序中各元素的对齐作用之外（如指令、注释的列对齐，各代码片段间隔的行对齐），汇编器并不对它们做任何处理。

7.2.2 伪操作

LC-3汇编器本身也是一个程序，只不过它是这样一个程序：输入是以行或字符串为单位的程序（用LC-3汇编语言编写的），而其输出是对LC-3汇编程序的翻译结果（即LC-3 ISA机器代码）。其中，LC-3汇编程序中的伪操作（pseudo-ops）用来对该翻译过程起到辅助作用。

事实上，伪操作的一个更标准的名字是“汇编指令”（assembler directive）。它们之所以被称为伪操作，原因是：它们不代表（被翻译的汇编）程序中的任何操作，或者说程序执行时它们是不产生任何操作的。伪操作可以理解为是程序员传递给汇编器的消息，用于指导汇编器的汇编操作。当汇编器看到这些消息后，就将这些伪操作丢弃。LC-3汇编器具有5种伪操作：.ORIG.、FILL.、BLKW、.STRINGZ、.END。开头的点号“.”代表这是一个伪操作。

1. ORIG

.ORIG告诉汇编器将LC-3程序放在内存的什么位置。如第05行的“.ORIG x3050”是说“从x3050开始”，于是LD R1,SIX指令就被放在地址x3050处。

2. FILL

.FILL告诉汇编器开始（占用）下一个地址，并填充初始值（如其操作数）。如第13行，也即最终LC-3程序的第9个单元，该单元被初始化为数值x0006。

3. BLKW

.BLKW告诉汇编器在程序空间中，开始占用一连串的地址空间（即“a BLocK of Word”）。具体的占用数目由.BLKW伪操作的操作数决定，如第12行指示汇编器占用1个内存空间（很巧的是，该空间还被标识为NUMBER）。

.BLKW特别适用于操作数值不确定的场合。例如，内存中的某个单元是用来存放键盘输入值的，而键盘输入直到程序运行时才能知道，那么它所占用的空间就可以事先通过.BLKW来申请占用。

4. STRINGZ

.STRINGZ告诉汇编器连续占用并初始化n+1个内存单元，其参数（或操作数）是双括号括起来的n个字符。n+1个内存单元的前n个字的内容分别是字符串对应字符的ASCII码的零扩展（zero-extend）值，内存的最后一个字则被初始化为0。最后的这个字符x0000通常为ASCII码字符串的处理提供了哨兵机制。

例如，如下代码片段：

```
        .ORIG    x3010
HELLO   .STRINGZ "Hello, World!"
```

将导致汇编器将内存x3010~x301D初始化为如下内容：

x3010:	x0048
x3011:	x0065
x3012:	x006C
x3013:	x006C
x3014:	x006F
x3015:	x002C
x3016:	x0020
x3017:	x0057
x3018:	x006F
x3019:	x0072

x301A:	x006C
x301B:	x0064
x301C:	x0021
x301D:	x0000

5. END

.END告诉汇编器“程序结束了”。出现在.END之后的任何字符都将被汇编器丢弃。注意：.END并不会停止程序的执行。事实上，最后的程序中不会出现任何.END。它仅仅是一个分隔符（表示源程序结束了）。

7.2.3 例子：字符数统计程序

下面我们讲述一个完整的例子。仍然以5.5节的问题为例，即写一个程序，从键盘读入一个字符，并统计一个文件中该字符出现的次数。按照惯例，首先通过流程图描述算法。在6.1节中，我们学习了怎样先将一个问题系统地分解成多个模块，然后生成如图5-16所示的流程图。事实上，第6章的最后一步生成的如图6-3e的流程图，本质上和图5-16是一样的；之后，就是按照流程图写程序。但这次写程序不用再担心0和1了，因为这次要用LC-3汇编语言完成编程任务。该程序如图7-2所示。

```
01   ;
02   ; Program to count occurrences of a character in a file.
03   ; Character to be input from the keyboard.
04   ; Result to be displayed on the monitor.
05   ; Program works only if no more than 9 occurrences are found.
06   ;
07   ;
08   ; Initialization
09   ;
0A           .ORIG   x3000
0B           AND     R2,R2,#0        ; R2 is counter, initialize to 0
0C           LD      R3,PTR          ; R3 is pointer to characters
0D           TRAP    x23             ; R0 gets character input
0E           LDR     R1,R3,#0        ; R1 gets the next character
0F   ;
10   ; Test character for end of file
11   ;

13   TEST    ADD     R4,R1,#-4       ; Test for EOT
14           BRz     OUTPUT          ; If done, prepare the output
15   ;
16   ; Test character for match.  If a match, increment count.
17   ;
18           NOT     R1,R1
19           ADD     R1,R1,R0        ; If match, R1 = xFFFF
1A           NOT     R1,R1           ; If match, R1 = x0000
1B           BRnp    GETCHAR         ; If no match, do not increment
1C           ADD     R2,R2,#1
1D   ;
1E   ; Get next character from the file
1F   ;
20   GETCHAR ADD     R3,R3,#1        ; Increment the pointer
21           LDR     R1,R3,#0        ; R1 gets the next character to test
22           BRnzp   TEST
23   ;
24   ; Output the count.
25   ;
26   OUTPUT  LD      R0,ASCII        ; Load the ASCII template
27           ADD     R0,R0,R2        ; Convert binary to ASCII
28           TRAP    x21             ; ASCII code in R0 is displayed
29           TRAP    x25             ; Halt machine
2A   ;
2B   ; Storage for pointer and ASCII template
2C   ;
2D   ASCII   .FILL   x0030
2E   PTR     .FILL   x4000
2F           .END
```

图7-2 字符统计的汇编程序

在此，对这个程序做些说明：

在这个程序中，三次通过系统服务调用获得操作系统服务，即TRAP指令方式。TRAP x23是读取键盘输入并将其存放在R0寄存器中（第0D行）；TRAP x21是将R0中存放的ASCII显示在屏幕上（第28行）；TRAP x25是停机操作（第29行）。有关TRAP指令的工作机制，仍然要留在第9章中展开讨论。

有关ASCII码。数字0~9（0000~1001）的对应ASCII码是x30~x39，所以它们的二进制至ASCII的转换方法很简单，只要对原值加x30即可。第2D行的标识“ASCII”指定的内存单元中，存放的就是这个“x0030”。

被统计文件在内存中的存放位置是x4000（第2E行）。通常，这个起始地址是不为写这个程序的程序员所知的，因为我们希望这个程序能处理不同的文件（即存放位置是变化的）。这种情况的处理方法将在7.4节讨论。

7.3 汇编过程

7.3.1 概述

一个LC-3汇编程序不能直接执行，它必须被翻译成机器语言代码，才能被机器识别并执行。所谓机器语言代码或程序，即其中的每条指令都是LC-3 ISA格式的（而不是汇编指令格式）。这个翻译任务是由汇编器完成的。

假设你已获取了现成的汇编器程序，通过执行特定的命令，就能“指挥”该汇编器完成翻译工作。现成的汇编器可通过Web下载获取。汇编器的运行命令是assemble，参数之一是汇编程序的文件名。例如，如果文件名是solution1.asm，则执行命令

```
assemble solution1.asm outfile
```

生成文件outfile，即一个LC-3 ISA格式的机器语言程序文件。

7.3.2 两遍扫描

本节介绍汇编器的工作原理，即从汇编语言至机器语言的翻译过程。我们以图7-2的汇编程序作为这个翻译过程的输入。

你应该还记得，汇编语言中的指令和翻译后的机器语言指令之间是存在“一一对应”关系的。直观上，这意味着从头到尾只需要一遍就可以完成翻译。下面以图7-2的程序为例，汇编器可以直接将第01~09行丢弃，因为它们只是注释（而不是指令），是供人阅读的（而不是执行的），对翻译过程来说毫无作用。于是，汇编器移至第0A行。0A行是一个伪操作，它告诉汇编器“机器程序将从x3000开始执行”；再移至0B行，这是个AND指令，很容易翻译。于是我们翻译出对应的机器指令

```
x3000:  0101010010100000
```

但是，当LC-3汇编器开始翻译下一条指令（0C行）之时，问题出现了！由于汇编器无法理解符号PTR的意思，汇编器卡壳了，翻译过程中止。

为了解决这个问题，汇编过程必须对整个汇编语言程序从头到尾（从开始到.END）多走一遍，为的是在第1遍过程中先确定各符号名（如PTR）所对应的二进制地址，即确定所有符号与地址之间的映射关系。我们称这个映射集合为“符号表”。

总之，两遍扫描的任务分别是，第1遍负责符号表的构建，第2遍负责将所有汇编指令翻译成对应的机器指令。

```
LD R3,PTR
```

那么，汇编器在第2遍过程（即翻译过程）中，再次遇到第0C行（如下所示）时

由于第1遍已确定PTR和x3013之间的映射关系（即建立对应符号表项），所以该指令被成功翻译为

```
x3001:  0010011000010001
```

换句话说，之前“无法确定PTR地址”的问题，通过两编扫描机制予以解决了。

7.3.3 第1遍：创建符号表

符号表并不复杂，不过就是符号名和16-bit地址之间的一张映射表。表项的生成方法是在第1遍扫描过程中，为每条指令以及每个标识顺序分配地址，并记录标识或符号及其所对应的地址。标识所对应的地址通常与跳转指令和内存访问指令（load或store）相关，即要么是跳转指令的目标地址，要么是数据存放的内存地址。

换句话说，只要编程不存在错误，所有的标识都将被分配并确定对应地址，即代码中通过符号或标识代表的内存引用（指令入口地址或数据单元地址），最终都将顺利替换为具体的地址。

之前的“伪操作”介绍中，我们知道伪操作指令.ORIG和.END界定了有效代码的范围，图7-2的汇编程序亦如此。但在7.4节中，我们还将指出包含多个独立代码区段的程序方式，即一个程序由多个代码区段拼装而成，其中每个部分都有自己的.ORIG和.END，并分别被独立汇编。

第1遍扫描的具体操作如下：

（1）首先，直接丢弃01~09行的注释。

（2）之后，最先遇到的应该是一个.ORIG伪操作（或汇编指令），如第0A行所示。它意味着之后将出现该程序的第一条指令，且对应的分配地址如ORIG参数所示。同时，我们还采用了一个地址跟踪计数器LC（Location Counter），LC的初始值由.ORIG指定，在此即为x3000。

（3）每成功识别出一条有效指令，LC增量（加1）。如果该指令行头部存在“标识”字段，则为该标识创建符号表项即二元组[符号，地址值]，其中地址值就是LC计数器的当前内容。

（4）继续识别下一行指令（3），直到.END。

第1个出现标识符的指令行是第13行。由于它是程序中的第15个指令，即LC当前内容是x3004，因而为其创建的表项内容为：

Symbol	Address
TEST	x3004

第2个包含标识符的指令在第20行。此时LC内容已被增量为x300B，因而创建的表项内容为：

Symbol	Address
GETCHAR	x300B

第1遍扫描结束之后，符号表最后的完整内容为：

Symbol	Address
TEST	x3004
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

7.3.4 第2遍：生成机器语言程序

第2遍扫描仍将再次扫描整个汇编程序，只不过这次有了符号表的帮助。在这次扫描中，每个有效指令行（而不是注释行或伪操作行）都将被翻译成一个机器指令。

第2遍扫描的具体操作如下所述：

（1）第01~09行再次被汇编器丢弃（因为它们是注释）。

（2）第0A行是伪操作.ORIG，因而LC被初始化为x3000。

（3）随后，汇编器移至0B行并生成机器指令0101010010100000。

（4）随后移至0C行，这次第0C行将顺利通过，因为事先已知PTR的对应地址为x3013。由于是LD指令，所以机器指令字的操作码字段为0010；目的寄存器字段（R3）填写为011；PCoffset字段则需稍做计算。已知PTR对应地址x3013且当前增量PC值如LC+1所示（即x3002），由于PTR地址值（x3013）等于增量PC（x3002）和符号扩展后PCoffset值之和，因而等式求解推出PCoffset字段内容为x0011。将所有字段拼装，x3001得到机器指令字0010011000010001，同时LC自动增量为x3002。

注意，LD指令中源操作数的地址（即PTR）不能超出LD指令所在位置的+256或−255。换句话说，如果PTR的地址大于LC+1+256或小于LC+1−255，则该偏移值就超出了指令bit[8:0]字段所能表示的范围。当然，如果这种情况出现了，汇编器会判断出来，并将终止汇编过程。幸运的是，本例中PTR和LD指令的间隔距离并不远，所以该指令顺利通过汇编。

随后，继续第2遍扫描，且每推进一步LC增量（加1）。同时，LC指向的地址被填入翻译的机器代码或填入数值（在.FILL情况下）。

直到.END，第2遍扫描结束。翻译后的最终程序如图7-3所示：

Address	Binary
	0011000000000000
x3000	0101010010100000
x3001	0010011000010001
x3002	1111000000100011
x3003	0110001011000000
x3004	0001100001111100
x3005	0000010000001000
x3006	1001001001111111
x3007	0001001001000000
x3008	1001001001111111
x3009	0000101000000001
x300A	0001010010100001
x300B	0001011011100001
x300C	0110001011000000
x300D	0000111111110110
x300E	0010000000000011
x300F	0001000000000010
x3010	1111000000100001
x3011	1111000000100101
x3012	0000000000110000
x3013	0100000000000000

图7-3 图7-2中汇编程序所对应的机器语言程序

但是，即使在心情很好的情况下，这个处理过程也是无聊透顶的。幸运的是，我们并不需要亲自操作这个过程，因为LC-3汇编器程序就是用来完成这个任务的。有了LC-3汇编语言，你就不必再编写枯燥的机器语言程序了；而有了汇编器程序，又省去了繁琐的翻译过程。换句话说，我们用LC-3汇编语言编程，然后让LC-3汇编器将其翻译成机器语言版本的、可以在LC-3计算机上执行的程序。

7.4 相关知识

本章从计算机的ISA层向上又提高了一层，即汇编语言。虽然这距离C或C++等高级语言还很远，但事实上，我们看到汇编语言已省去了不少麻烦。同时，我们还介绍了二次扫描编译器将汇

编语言程序翻译成LC-3 ISA机器语言的原理和过程。

但是，本章并未展开讨论汇编语言编程中的更多高级技术。在本书中，有关汇编语言的讲述并不是研究其技术细节，而是介绍其简洁性的一面。

下面，我们对与汇编语言相关的其他技术内容做一个概述。

7.4.1 可执行映像

当计算机开始程序的执行时，执行实体（entity）又称做“可执行映像”（executable image）。可执行映像通常是由多个相互独立的模块组装在一起形成的，而这些模块通常又是由不同的程序员分别编写的。刚才，我们已模拟LC-3汇编器，走了一遍汇编语言程序的翻译过程；但是，其他模块可能是用C语言编写的，需要借助C编译器（compiler）才能完成翻译过程。另外，这些模块中，有的是由用户编写的，而有的则来自操作系统的库程序（library routine）。

所有这些模块在组装之前，是一个个独立的目标文件（其中包含着机器ISA形式的指令及其相关数据），最后是通过链接操作将这些目标拼装成一个可执行映像的。程序执行的时候，其中的每条指令又被细分为FETCH、DECODE、…等指令周期。这就是从编程语言到机器语言及指令周期的层次细化。

7.4.2 多目标文件

如前所述，通常一个可执行映像由多个模块（或目标）文件组成。事实也如此，大多数的程序都会大量调用操作系统提供的库函数或由其他程序员编写的模块[⊖]，也就是说，最后的可执行映像是由多个目标文件（而不是单个目标文件）拼装而成的。

下面，我们仍以字符数统计程序为例（即统计一个文件中特定字符出现的次数）。假设一个应用程序将该程序作为其模块之一，而将文件输入操作实现在另一个模块中，那么，文件的起始地址（如图7-2的2E行所示）在此程序中将成为一个未知值。例如，我们将第2E行代码改写成如下形式：

```
PTR   .FILL   STARTofFILE
```

这意味着图7-2的程序将无法通过汇编，因为符号STARTofFILE是未知的（第1遍扫描生成的符号表中没有它的对应表项），怎么办呢？

解决办法是，如果LC-3汇编语言中包含如.EXTERNAL的伪操作，则我们可以用符号STARTofFILE声明此类符号，它表明在本程序（或模块）中，没有提供该符号的地址信息。声明方式如下：

```
.EXTERNAL    STARTofFILE,
```

它传递给汇编器的信息是，如果在当前文件中没有发现STARTofFILE的对应地址信息，还不能确定是个编程错误，因为该符号的地址可能在另一个外部（external）模块里定义了。换句话说，我们借用LC-3汇编器提供了伪操作指令“.EXTERNAL”，将STARTofFILE声明为.EXTERNAL类型，因此LC-3汇编器在第1次扫描过程中，会在符号表中为STARTofFILE创建一个符号表项，但在地址值字段填写的不是地址，而是一个特殊标志（标志该符号由另一个模块定义）；在链接阶段（link time）即所有模块合并之际，链接器（Linker）将借助其他模块提供的STARTofFILE符号表项，重新修订第2E行的内容。

有了.EXTERNAL方法，使得一个程序模块事先可以显式地引用另一个模块里的符号，然后由链接器完成最后的修补工作。

⊖ 库函数或其他模块都是目标文件格式，而不是源代码格式。——译者注

7.5 习题

7.1 假设一个汇编语言程序中包含了以下两条指令，汇编器将翻译后的LDI指令放在目标模块的x3025位置。问汇编过程结束后，x3025的内容是什么？

```
PLACE   .FILL   x45A7
        LDI     R3, PLACE
```

7.2 假设汇编语言程序中包含下面的指令

ASCII的符号表项的内容是x4F08，问该指令执行之后，R1的内容是什么？

```
ASCII   LD  R1, ASCII
```

7.3 如果采用字符串AND作为标识（label），会出什么问题？（提示：第1遍扫描会是什么情况？第2遍扫描会是什么情况？）

7.4 试为以下代码写出在汇编器处理下符号表的内容。

```
        .ORIG   x301C
        ST      R3, SAVE3
        ST      R2, SAVE2
        AND     R2, R2, #0
TEST    IN
        BRz     TEST
        ADD     R1, R0, #-10
        BRn     FINISH
        ADD     R1, R0, #-15
        NOT     R1, R1
        BRn     FINISH
        HALT
FINISH  ADD     R2, R2, #1
        HALT
SAVE3   .FILL   X0000
SAVE2   .FILL   X0000
        .END
```

7.5 a. 试阅读以下程序，说明它的作用。

```
        .ORIG   x3000
        LD      R2, ZERO
        LD      R0, M0
        LD      R1, M1
LOOP    BRz     DONE
        ADD     R2, R2, R0
        ADD     R1, R1, -1
        BR      LOOP
DONE    ST      R2, RESULT
        HALT
RESULT  .FILL   x0000
ZERO    .FILL   x0000
M0      .FILL   x0004
M1      .FILL   x0803
        .END
```

b. 程序结束后，RESULT的内容是什么？

7.6 （假设）我们的汇编器崩溃了，需要你的帮助！请创建一个符号表，并将下面程序中标号为D、E、F的指令手工汇编出来。你可以假设在这个模块执行之前，已有另外一个模块将一个正数放入了E中。

```
        .ORIG     x3000
        AND       R0, R0, #0
D       LD        R1, A
        AND       R2, R1, #1
        BRp       B
E       ADD       R1, R1, #-1
B       ADD       R0, R0, R1
        ADD       R1, R1, #-2
F       BRp       B
```

```
        ST      R0, C
        TRAP    x25
A       .BLKW   1
C       .BLKW   1
        .END
```

请用不多于15个字的一句话，解释上面这个程序在做什么。

7.7 试写一个LC-3汇编程序，统计R0寄存器的数值中包含了多少个1，并将结果存入R1。例如，假设R0的内容是0001001101110000，则程序执行之后，R1的内容等于6(0000000000000110)。

7.8 一个工程师正在调试她写的一个程序。当她看到下面代码时，决定在地址0xA404处放置一个断点（breakpoint）。在PC=0xA400时，她将所有寄存器清0，并运行程序直到遇到断点。

```
Code Segment:
...
0xA400  THIS1  LEA    R0, THIS1
0xA401  THIS2  LD     R1, THIS2
0xA402  THIS3  LDI    R2, THIS5
0xA403  THIS4  LDR    R3, R0, #2
0xA404  THIS5  .FILL  xA400
...
```

试写出遇到断点的时候，寄存器文件的内容（用十六进制表示）。

7.9 伪操作.END的作用是什么？它和HALT指令之间的区别是什么？

7.10 下面代码片段中存在一个错误。请找出错误并修复它。

```
        ADD     R3, R3, #30
        ST      R3, A
        HALT
A       .FILL   #0
```

该错误是在被汇编的时候，还是在LC-3上运行的时候，将被发现？

7.11 LC-3汇编器必须能够将ASCII表示的常数转换为相应的二进制数值。例如，x2A将被翻译为00101010，而#12被翻译为00001100。试写一个汇编语言程序，将从键盘读入的十进制或十六进制常数（前缀#代表十进制，前缀x代表十六进制）转换为二进制数表达方式，并打印出来（假设常数表示时，十进制或十六进制的数字不超过两个）。

7.12 阅读以下代码，试解释该程序的目的。

```
        .ORIG   x3000
        AND     R5, R5, #0
        AND     R3, R3, #0
        ADD     R3, R3, #8
        LDI     R1, A
        ADD     R2, R1, #0
AG      ADD     R2, R2, R2
        ADD     R3, R3, #-1
        BRnp    AG
        LD      R4, B
        AND     R1, R1, R4
        NOT     R1, R1
        ADD     R1, R1, #1
        ADD     R2, R2, R1
        BRnp    NO
        ADD     R5, R5, #1
NO      HALT
B       .FILL   xFF00
A       .FILL   x4000
        .END
```

7.13 以下程序的目的是将存放在内存A、B、C中的内容相加，并将结果存入内存。但是，代码中存在两个错误。试找出错误，并分别解释错误会在汇编时还是在运行时被检测出来？

```
Line No.
1               .ORIG  x3000
2       ONE     LD  R0, A
```

```
3                ADD R1, R1, R0
4         TWO    LD  R0, B
5                ADD R1, R1, R0
6         THREE  LD  R0, C
7                ADD R1, R1, R0
8                ST  R1, SUM
9                TRAP  x25
10        A      .FILL x0001
11        B      .FILL x0002
12        C      .FILL x0003
13        D      .FILL x0004
14               .END
```

7.14 a. 汇编以下程序（即写出该程序的机器代码）：

```
          .ORIG   x3000
          STI     R0, LABEL
          OUT
          HALT
LABEL     .STRINGZ "%"
          .END
```

b. 程序员试图将%符号显示在屏幕上，然后停机（halt）。但程序员混淆了操作码（即使用了错误的汇编指令）。请将程序中被错误使用的一个操作码替换为正确的操作码。

c. 原先的程序（如a）的执行会产生奇怪的现象。产生错误现象的原因一部分是由程序员的失误造成的，另一部分归因于R0的内容在程序执行之初是x3000。试解释奇怪的现象是什么样的，以及为什么会这样？

7.15 假设x4000开始的连续内存空间中存放了一个整数序列，且每个内存单元存放一个整数。该序列的结尾数值是x0000。阅读下面的程序，请问该程序的目的是什么？

```
          .ORIG   x3000
          LD      R0, NUMBERS
          LD      R2, MASK
LOOP      LDR     R1, R0, #0
          BRz     DONE
          AND     R5, R1, R2
          BRz     L1
          BRnzp   NEXT
L1        ADD     R1, R1, R1
          STR     R1, R0, #0
NEXT      ADD     R0, R0, #1
          BRnzp   LOOP
DONE      HALT
NUMBERS   .FILL   x4000
MASK      .FILL   x8000
.END
```

7.16 假设x4000开始的连续内存空间中存放了一个非负整数序列，且每个内存单元存放一个整数。每个整数的数值范围是0~30000（十进制），结尾数值是-1。阅读下面的程序，请问该程序的目的是什么？

```
          .ORIG   x3000
          AND     R4, R4, #0
          AND     R3, R3, #0
          LD      R0, NUMBERS
LOOP      LDR     R1, R0, #0
          NOT     R2, R1
          BRz     DONE
          AND     R2, R1, #1
          BRz     L1
          ADD     R4, R4, #1
          BRnzp   NEXT
L1        ADD     R3, R3, #1
NEXT      ADD     R0, R0, #1
          BRnzp   LOOP
DONE      TRAP    x25
NUMBERS   .FILL   x4000
          .END
```

7.17 假设你写了两个独立的汇编语言模块，并用链接器将两者结合在一起。其中，每个模块都使用了标识AGAIN，但没有一个模块中包含了伪操作.EXTERNAL AGAIN。请问在两个模块中，同时使用标识AGAIN是否有问题？为什么？

7.18 下面程序比较两个相同长度的字符串。第一个字符串的起始地址是x4000，第二个字符串的起始地址是x4100，两者都是.STRINGZ格式。如果两个字符串相同，则程序结束时R5=0。请在(a)、(b)、(c)中填充指令，完成该程序。

假设x4000开始的连续内存空间中存放了一个整数序列，且每个内存单元存放一个整数。该序列的结尾数值是x0000。阅读下面的程序，请问它的目的是什么？

```
        .ORIG  x3000
        LD     R1, FIRST
        LD     R2, SECOND
        AND    R0, R0, #0
LOOP    -------------- (a)
        LDR    R4, R2, #0
        BRz    NEXT
        ADD    R1, R1, #1
        ADD    R2, R2, #1
        -------------- (b)
        -------------- (c)
        ADD    R3, R3, R4
        BRz    LOOP
        AND    R5, R5, #0
        BRnzp  DONE
NEXT    AND    R5, R5, #0
        ADD    R5, R5, #1
DONE    TRAP   x25
FIRST   .FILL  x4000
SECOND  .FILL  x4100
        .END
```

7.19 请问下面的程序执行时，标识LOOP处的指令会被执行多少次？

```
        .ORIG   x3005
        LEA     R2, DATA
        LDR     R4, R2, #0
LOOP    ADD     R4, R4, #-3
        BRzp    LOOP        TRAP    x25

DATA    .FILL   x000B
        .END
```

7.20 下面的模块（a）和（b）分别是两个程序员写的LC-3汇编程序，两者的目的都是将数值x0015填入地址x4000。请问两者的本质区别是什么？

a.

```
    .ORIG  x5000
    AND    R0, R0, #0
    ADD    R0, R0, #15
    STI    R0, PTR
    HALT
PTR .FILL  x4000
    .END
```

b.

```
   .ORIG  x4000
   .FILL  x0015
   .END
```

7.21 汇编以下LC-3汇编语言程序（写出机器码）：

```
        .ORIG   x3000
        AND     R0, R0, #0
        LD      R1, MASK
        ADD     R2, R0, #10
        LD      R3, PTR1
LOOP    LDR     R4, R3, #0
        AND     R4, R4, R1
        BRzP    NEXT
        ADD     R0, R0, #1
NEXT    ADD     R2, R2, #-1
        BRp     LOOP
        STI     R0, PTR2
MASK    .FILL   x8001
PTR1    .FILL   x4000
PTR2    .FILL   x5000
        .END
```

请问程序的目的是什么（用不多于20个字来描述）？

7.22 LC-3汇编器必须能将指令的助记（mnemonic）操作码翻译成二进制操作码。例如助记操作码ADD对应的二进制码形式为0001。写一个LC-3汇编语言程序，提示用户输入一个助记操作码（如ADD），然后将其对应的二进制码打印到屏幕上。如果输入的是无效助记操作码，则显示错误信息。

7.23 试写一个LC-3汇编程序，判断一个字符串是否是“回文”（palindrome）。所谓“回文”，就是正读或反读结果都是一样的。例如，字符串“racecar”就是一个回文。假设x4000存放了一个格式为.STRINGZ的字符串。判断该字符串是否是回文，如果是则R5返回内容1；如果不是，则R5返回0。填充(a)~(e)处的指令，完成整个程序。

```
        .ORIG  x3000
        LD     R0, PTR
        ADD    R1, R0, #0
AGAIN   LDR    R2, R1, #0
        BRz    CONT
        ADD    R1, R1, #1
        BRnzp  AGAIN
CONT    --------------(a)
LOOP    LDR    R3, R0, #0
        --------------(b)
        NOT    R4, R4
        ADD    R4, R4, #1
        ADD    R3, R3, R4
        BRnp   NO
        --------------(c)
        --------------(d)
        NOT    R2, R0
        ADD    R2, R2, #1
        ADD    R2, R1, R2
        BRnz   YES
        --------------(e)
YES     AND    R5, R5, #0
        ADD    R5, R5, #1
        BRnzp  DONE
NO      AND    R5, R5, #0
DONE    HALT
PTR     .FILL  x4000
        .END
```

7.24 我们的本意是让以下程序将R3内容左移4个bit，但程序中存在错误，请找出错误并修正。

```
        .ORIG  x3000
        AND    R2, R2, #0
        ADD    R2, R2, #4
LOOP    BRz    DONE
        ADD    R2, R2, #-1
        ADD    R3, R3, R3
        BR     LOOP
DONE    HALT
        .END
```

7.25 伪操作.FILL xFF004有什么问题吗？为什么？

第8章　输入/输出

至目前为止，我们还没有提到计算机的输入/输出部分，但这并不代表它不重要。相反，从第4章内容中，我们获知“输入/输出”是冯·诺依曼模型中的一个重要组成部分。因为我们需要找到一种方法，将待处理的信息输入到计算机系统中，并将计算结果以人们能够理解的形式输出。图4-1列举了各种不同的输入输出设备。

另外，在第5章提到，可以通过TRAP指令请求操作系统来具体完成输入输出操作。图5-17给出了使用例子，即输入（地址0x3002）和输出（地址0x3010）。

本章中，我们准备自己写代码来完成这些输入输出操作。其中，输入设备选择键盘，输出设备选择显示器。选择这两个设备的原因在于，首先，它们是最简单的一类I/O设备（大家都非常熟悉的）；其次，通过它们即可以了解I/O概念和操作特性（而无需陷入一些不必要的硬件细节）。

8.1　输入/输出的基本概念

8.1.1　设备寄存器

通常，我们可能认为I/O设备就是一个单一的外部实体，如键盘和显示器。但是，跟一个I/O设备进行交互，却不是想像中的那么简单。我们需要和多个*设备寄存器*（device register）打交道，即使最简单的I/O设备，也至少包含两个寄存器：一个用来保存跟计算机之间传输的数据；另一个用来指示当前设备的状态信息，如“设备是否空闲”或“最近处理的I/O任务”等。

8.1.2　内存映射I/O与专用I/O指令

指令访问I/O设备寄存器时，需要明确指明目标寄存器。通常有两种实现方法，一些计算机厂家（通常很少）采用专门的I/O指令来访问，即“专用I/O指令”方式；另一种方法（大多数计算机厂家都采用）是采用内存操作指令来完成I/O操作，即“内存映射I/O”方式。

1965年，DEC公司生产的PDP-8就是一台采用“专用I/O指令”的计算机。PDP-8的指令长度为12位，其中前3位表示操作码。如果操作码的值为110，则表示该指令是I/O指令，剩余9位则代表目标寄存器和具体操作。

与专用I/O指令相比，设计者们更偏爱内存指令的I/O操作方式，即内存映射I/O。例如，如果load指令的目标地址是一个设备寄存器，则它就是一条输入指令。又如，一个store指令的目标地址是一个设备寄存器，则它就是一条输出指令。

由于程序员使用与内存访问相同的指令来操作I/O，所以设备的每个输入或输出寄存器，都必须有一个与内存中位置相同的标识方式。为此，在ISA的内存地址空间中，专门划分出一段地址，用于设备寄存器的标识映射。换句话说，I/O设备的寄存器被“映射”到一组地址（这些地址是分配给I/O设备寄存器的，而不是分配给内存位置的），即所谓的“*内存映射*I/O”（memory-mapped I/O）方式。

例如，PDP-11指令集的地址空间是16位的（即2^{16}个地址单元），其中，最高3位为111的地址分配给I/O设备，即可以有2^{13}个地址单元用来标识I/O设备寄存器，剩下的57 344个地址单元则用来标识内存。

LC-3采用内存映射I/O方式。其中，0x0000~0xFDFF的地址空间用于标识内存，0xFE00~0xFFFF则保留给外部设备使用。表A-3列出了LC-3已分配的设备寄存器地址，其他地址预留以后使用。

8.1.3 异步I/O与同步I/O

下面描述打字员的输入字符操作。打字员每次键入一个字符时，该字符的ASCII码值就被存入键盘输入寄存器中。然后CPU执行load指令时，如果目标地址指向键盘输入寄存器，则将该ASCII值读入计算机。

与CPU相比，大多数I/O设备的速度是非常缓慢的。现代微处理器的时钟频率一般都超过了300MHz，假设处理器工作频率是300MHz频率（即一个时钟周期只有3.3ns），CPU执行load指令占用10个时钟周期，则处理器从输入寄存器中读取一次数据仅需33ns。但是，打字员的输入速度远达不到CPU读取数据的速度。*思考题*：假设每个单词的平均长度是6个字符，如果要赶上处理器的读取速度，试问打字员输入单词的速度至少是多少（参考习题8.3）？

我们可以通过硬件设计方法来减小这个速度差异。例如，设计一个系统，该系统每30 000 000个时钟周期读取一次，则打字员每分钟只需键入100个单词即可。但是，这种设计方式下，要求打字员和硬件保持时钟同步，这显然是不可行的，因为打字员的输入速度是随时变化的。

那该怎么办呢？问题的关键在于I/O设备和处理器的工作节奏是不一致的，因此我们称I/O设备和CPU之间是*异步的*（asynchronous）。许多外设和处理器之间的交互都是异步的，如我们提到的键盘和显示器。异步通信方式下，需要通过一定的协议或*握手*（handshaking）机制来控制发送和接收。例如，在键盘输入的例子中，我们采用一个单bit的状态寄存器，称为*标志*（flag），来指示“是否有新的字符输入”。在显示器输出的例子中，我们也使用一个单bit状态寄存器，来表示最近输入的字符是否已经显示在显示器上。

实现同步的最简单的方式是采用标志。键盘状态寄存器中的*标志位*（Ready bit）提供了最简单的一种握手机制：打字员每键入一个字符，该Ready位就被置位，而每次处理器读取之后，都自动将该Ready位清零。所以，在每次读取输入字符之前，处理器都将检查该Ready位，如果Ready位为0，表示没有新的字符输入，处理器将不再读取输入寄存器；否则表示有新字符输入，则处理器执行load指令，读入输入寄存器中的ASCII码值。通过这种握手机制，保证了打字员和处理器之间的数据传输。

另外，如果打字员输入字符的速度是恒定的，我们还可以将硬件设计为定期（如每30 000 000个时钟周期）读取一次输入寄存器。那么，我们就不需要Ready位了，因为处理器知道在30 000 000个时钟周期后，必然有新字符输入。由于打字员和CPU的节奏是一致的，因而不需要额外的同步信息。如果是这样，我们就称打字员和处理器的交互是*同步的*（synchronously）。

8.1.4 中断驱动与轮询

处理器和打字员是两个不同的实体。虽然它们的任务不一样（处理器的任务是执行程序，而打字员的任务是向计算机输入数据），但是它们之间仍然需要交互，即打字员输入的数据必须交给处理器来处理。交互的两种基本方式是*中断驱动*（interrupt-driven）和*轮询*（polling），它们之间的根本区别在于双方是谁控制这个交互。以键盘和CPU之间的交互为例，如果由键盘来控制交互过程，则处理器只顾做自己的事情，当键盘数据准备好时，它会主动通知处理器：“输入字符的ASCII值已存放在输入寄存器中，你可以读取了！”我们称这种由外设控制的交互方式为“*中断驱动*I/O”；而如果该交互过程由处理器控制，则处理器必须不断读取、测试状态寄存器内容，直到Ready位被置位（即有新的字符输入），于是从输入寄存器中读取数据。在这种交互方式下，处理

器必须反复查询，因此称之为“轮询方式的I/O”。

在8.2.2节中，将描述轮询工作方式，在8.5节中，将描述中断驱动I/O方式。

8.2 键盘输入

8.2.1 基本输入寄存器

我们知道，将字符从键盘输入到计算机内部需要两个条件：一是数据寄存器，用来存放键盘输入字符的ASCII值；二是同步机制，即用来告知处理器数据已准备好的方法。其中，同步机制可以通过键盘状态寄存器来实现，如果是这样，键盘就需要两个寄存器，其一是键盘数据寄存器（KBDR），其二是键盘状态寄存器（KBSR）。附录A中的表A-3列出了这两个寄存器的地址，其中，KBDR（数据寄存器）的地址是0xFE02，KBSR（状态寄存器）的地址是0xFE00。

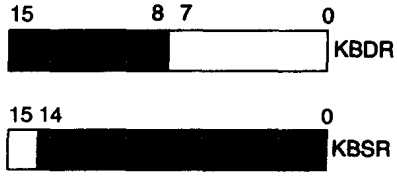


图8-1 键盘的设备寄存器

虽然键盘的数据寄存器只需要8位，而状态寄存器仅需要1位，但我们还是给它们分别分配了两个16位的空间。如图8-1所示，KBDR的第0~7位用以存储输入字符的ASCII值，KBSR的第15位用以存储同步信息（Ready位）。

8.2.2 基本输入服务程序

KBSR[15]的作用是控制处理器（快速）和键盘（慢速）之间的同步。当某个按键被按下时，该键对应的ASCII值在存入KBDR[7:0]的同时，键盘硬件电路自动将KBSR[15]置1；当LC-3读取KBDR时，键盘自动清除KBSR[15]。KBSR[15]=0意味着键盘可以继续输入；KBSR[15]=1，意味着上次输入字符还未被处理器取走，即键盘无法继续输入其他字符。

在轮询方式下（即处理器控制的I/O），程序将反复读取并测试KBSR[15]位，直到它被置为1。如果程序读到KBSR[15]被置位，处理器就将KBDR中的ASCII值拷贝到LC-3内部的一个本地寄存器中。键盘的任一输入字符不应该也不会被重复读取，这是因为在LC-3读取KBDR时，键盘的硬件电路会自动清除KBSR[15]，而程序仅在KBSR[15]=1的情况下，才会读取KBDR的内容。另外，键盘在前一个输入字符被取走之前是被禁用的，这又保证了已输入字符是不会丢失的。由此可见，KBSR[15]的同步机制保证了输入字符有且仅有一次被读取的机会。

如下程序的功能是，将键盘KBDR寄存器中的ASCII值读入本地寄存器R0中，然后跳转至下一个任务：

```
01    START  LDI     R1, A          ; Test for
02           BRzp    START          ; character input
03           LDI     R0, B
04           BRnzp   NEXT_TASK      ; Go to the next task
05    A      .FILL   xFE00          ; Address of KBSR
06    B      .FILL   xFE02          ; Address of KBDR
```

KBSR[15]=0表示自上次读取键盘之后，一直没有新的输入。程序的第1、2行构成KBSR[15]位的测试循环，LDI指令将地址0xFE00处的值载入R1（0xFE00是KBSR寄存器的内存映射地址）。如果KBSR[15]为0，则程序跳回START继续循环。当有人敲击键盘时，则该键的ASCII码被装入KBDR，同时KBSR[15]被置位。这意味着程序将进入第3行代码，即通过LDI指令将地址0xFE02处的内容装入R0（0xFE02是KBDR寄存器的内存映射地址）。于是读取键盘输入过程完毕，程序无条件跳转至NEXT_TASK。

8.2.3 内存映射输入的实现

图8-2所示是I/O内存映射方式下的数据通路。基于第5章的介绍，我们知道数据通路在完成

load指令的EXECUTE节拍时的3个基本步骤为：

（1）在MAR中装入要读取内存的地址。

（2）启动内存访问，并将读出内容存入MDR。

（3）将MDR内容拷贝至目的寄存器DR。

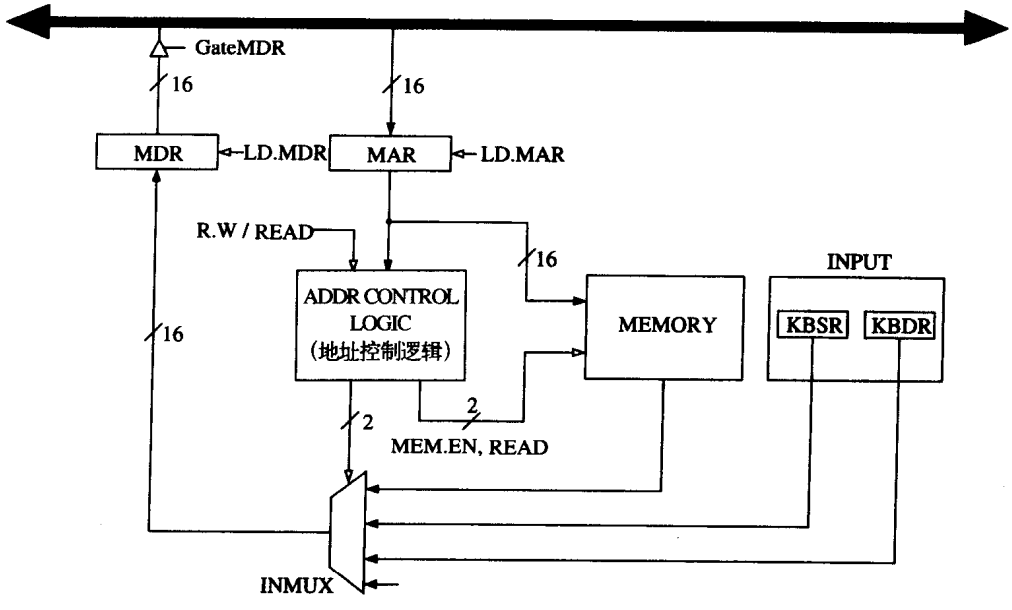


图8-2 内存映射输入

在内存映射方式下，设备寄存器的输入步骤和内存读操作完全一样。只是内存读取时，MAR保存的是内存地址，而内存映射输入时则是设备寄存器地址。同样，内存映射方式下MDR的装入内容来自设备寄存器，而内存读取时则来自内存单元。

8.3 显示器输出

8.3.1 基本输出寄存器（DDR和DSR）

显示器输出和键盘输入的原理相似，只是在显示器工作中，使用的输出数据寄存器和状态寄存器分别是DDR和DSR（而键盘操作使用的是KBDR和KBSR）。在LC-3中，DDR和DSR的内存映射地址分别是0xFE06和0xFE04。

DDR只需要用8-bit来存放数据，DSR仅需要1位，用来同步，这些都与KBDR相同。但为方便起见，我们仍然为DDR和DSR分别分配了16位的空间。其中，DDR的[7:0]位用来输出字符的ASCII值，DSR[15]位用来表示输出同步（Ready 位）。图8-3所示是与显示器相关的两个设备寄存器。

8.3.2 基本输出服务程序

DSR[15]用来同步处理器（快速）和显示器（慢速）。在LC-3将单个字符的ASCII值输出到DDR[7:0]的同时，显示器电路自动清除DSR[15]，然后开始DDR[7:0]内容的处理；当显示器完成字符到屏幕的输出之后，其电路将自动设置DSR[15]位，它表示处理器可以继续下一个字符的输出了（即DDR[7:0]空间空闲了）。DSR[15]=0，表示显示器“忙”（busy），当前仍在

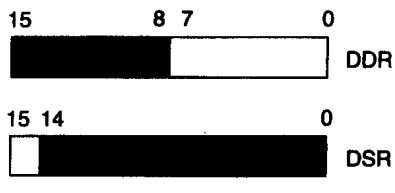


图8-3 显示器的设备寄存器

处理前一个字符；DSR[15]=1，意味着显示器已完成前面的工作，目前空闲，处理器可以进行下一个字符的输出。

如果该输出过程由处理器来控制，则意味着程序需要不断测试DSR[15]位的状态。通常的处理流程是：首先，检测DSR[15]位是否置1；如果为1，则处理器输出字符的ASCII值写入DDR[7:0]中，并启动屏幕显示。

下面这段代码是将R0中的字符输出到显示器上：

```
01      START   LDI     R1, A           ; Test to see if
02              BRzp    START           ; output register is ready
03              STI     R0, B
04              BRnzp   NEXT_TASK
05      A       .FILL   xFE04           ; Address of DSR
06      B       .FILL   xFE06           ; Address of DDR
```

该代码和8.2.2节的键盘输入程序非常相似，第1、2行循环测试DSR[15]位，如果DSR被置位，表示显示器已经完成前一次的字符输出工作。第1行的LDI指令将地址0xFE04处的值装入R1（0xFE04是DSR寄存器的内存映射地址），如果DSR[15]没有置位，程序跳回START重新执行。显示器完成字符输出之后，将自动置位DSR[15]，此时程序将跳到第3行。STI指令将R0中的ASCII值输出到0xFE06地址（DDR的内存映射地址），同时硬件电路将自动清除DSR[15]位，这就保证了只有显示器完成当前字符输出之后，才能接收新的数据。输出完成之后，程序转去执行其他任务。

8.3.3 内存映射输出的实现

图8-4所示是内存映射输出的数据通路。与之前讨论的内存映射输入一样，只需要在原先内存访问数据通路的基础上，添加很少的控制逻辑，就可以实现设备寄存器的访问。

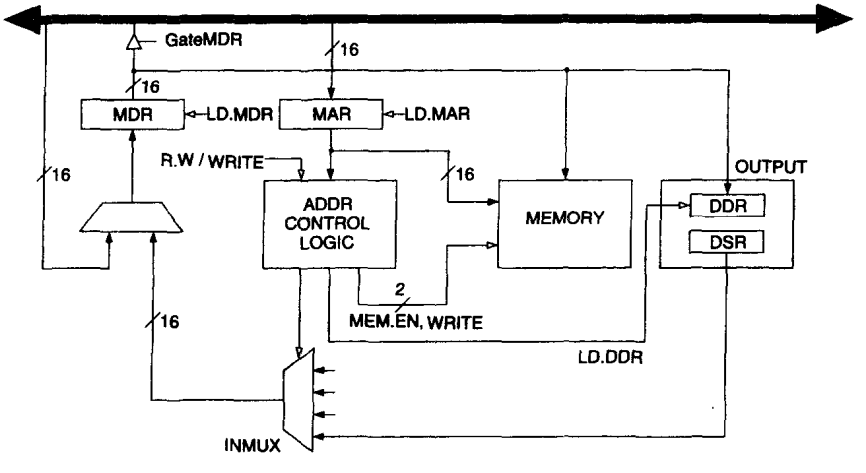


图8-4 内存映射输出

根据第5章学到的知识，我们知道，store指令的EXECUTE节拍执行下面3个步骤：

（1）在MAR中装入要被写入的内存地址。

（2）在MDR中装入数据。

（3）等待内存写操作完成（将MDR内容写入MAR所指向的内存单元）。

内存映射方式下，设备输出的操作与上面代码相似。所不同的是，此时MAR的内容是设备寄存器地址（而不是内存地址），控制逻辑使能（enable）的是设备寄存器而不是内存单元。

图中同时也包括了内存映射的输入部分，这是因为输出操作也需要寄存器读操作。如8.3.2节指出的，只有DSR的Ready位置1时，处理器才能向DDR写入新字符。参见代码的第1、2行测试——读取并测试DSR[15]的值，如果MAR=0xFE04（DSR的内存映射地址），则地址逻辑单元选择DSR作

为MDR的输入，MDR的值被载入R1寄存器并进行测试。

8.3.4 例子：键盘回显

当我们敲键盘的时候，通常希望知道刚才键入的字符是否正确。如果有“回显”（echo）功能，就非常方便了，即将刚才键盘输入的字符，输出在显示器上。下面的程序是已有的键盘输入和显示器输出等两个程序的结合，可以实现回显功能：

```
01     START   LDI     R1, KBSR     ; Test for character input
02             BRzp    START
03             LDI     R0, KBDR
04     ECHO    LDI     R1, DSR      ; Test output register ready
05             BRzp    ECHO
06             STI     R0, DDR
07             BRnzp   NEXT_TASK
08     KBSR    .FILL   xFE00        ; Address of KBSR
09     KBDR    .FILL   xFE02        ; Address of KBDR
0A     DSR     .FILL   xFE04        ; Address of DSR
0B     DDR     .FILL   xFE06        ; Address of DDR
```

8.4 一个更复杂的输入程序

如8.2.2节的输入程序，在实际程序中它只占很小的一部分，事实上还有许多问题需要考虑。例如，假设一个程序需要通过键盘输入数据，坐在机器前面的用户怎样才能知道何时应该敲键盘了呢？坐在那里，用户无法知道程序运行到哪里了，甚至无法知道程序现在是否真的在运行。

为了让用户知道程序正在等待用户的输入，计算机可以在显示器上打印提示信息，这些信息又称*提示符*。它们是由操作系统（显示为%或C:）或编辑器程序（显示为:）等输出的。

图8-5的程序片段，将通过轮询机制读取键盘输入，该例子在8.2.2节已出现过。其中包括了一段提示代码，告诉用户现在可以输入信息了。下面是我们的代码解释。

（1）第13~19行、25~28行的代码大家应该非常熟悉了，这是8.3.4节的键盘字符输入和回显程序。

（2）第1~3、1D~1F和22~24行的代码则暗示了该程序将动用寄存器R1、R2、R3。由于在我们看到的代码中将使用这些寄存器，同时又担心它们在之前可能存储了数值，且这些数值在当前代码结束后还要继续使用，所以第1~3行的代码负责在程序执行前，将R1、R2、R3分别保存到SaveR1、SaveR2、SaveR3等3个内存单元中（参见第22~24行，通过.BLKW伪操作码分配内存的操作）。程序完成之后，则通过第1D~1F行恢复寄存器的原值。

（3）剩下第5~8行、0A~11行、1A~1C行，以及29和2A行代码。这些代码负责输出提示信息input a character>，通知用户可以输入信息了。第5~8行的作用是将ASCII码值0xA输出至显示器。该ASCII码代表“*换行*”（newline）。大多数的ASCII码都是由屏幕可见的。但是，有些ASCII码对应的是控制字符，如0x0A。它们的作用是引发动作，如0xA的作用是控制显示器的光标，将其移动到下一行开始，所以称之为“*换行符*”。当然，即使是输出0xA，之前也需要测试DSR[15]。如果该位被清0，就表示显示器“忙”，程序继续循环（06和07行），直到DSR[15]变为1，跳转条件不满足（07行），之后将0xA写入DDR寄存器。

第0A～11行代码输出提示符“Input a character>”到屏幕。提示符一共18个字符，在第2A行代码处分配空间，每个字符占用一个内存单元，加上结束标识符“0x0000”，一共占用19个内存单元。

0C行代码循环测试字符串结束符（0x0000），如果没有遇到结束符，并且DDR空闲，第0F行代码就输出信息提示符到DDR寄存器。一旦遇到结束符，程序就知道提示符输出完毕，转而跳到第13行执行，等待用户的键盘输入。

当用户从键盘输入一个字符后，程序读取该值并将其在显示屏上回显（第13～19行），最后在输出一个换行符（第13行）后，跳到其他地方执行。

```
01  START   ST      R1,SaveR1    ; Save registers needed
02          ST      R2,SaveR2    ; by this routine
03          ST      R3,SaveR3
04  ;
05          LD      R2,Newline
06  L1      LDI     R3,DSR
07          BRzp    L1           ; Loop until monitor is ready
08          STI     R2,DDR       ; Move cursor to new clean line
09  ;
0A          LEA     R1,Prompt    ; Starting address of prompt string
0B  Loop    LDR     R0,R1,#0     ; Write the input prompt
0C          BRz     Input        ; End of prompt string
0D  L2      LDI     R3,DSR
0E          BRzp    L2           ; Loop until monitor is ready
0F          STI     R0,DDR       ; Write next prompt character
10          ADD     R1,R1,#1     ; Increment prompt pointer
11          BRnzp   Loop         ; Get next prompt character
12  ;
13  Input   LDI     R3,KBSR
14          BRzp    Input        ; Poll until a character is typed
15          LDI     R0,KBDR      ; Load input character into R0
16  L3      LDI     R3,DSR
17          BRzp    L3           ; Loop until monitor is ready
18          STI     R0,DDR       ; Echo input character
19  ;
1A  L4      LDI     R3,DSR
1B          BRzp    L4           ; Loop until monitor is ready
1C          STI     R2,DDR       ; Move cursor to new clean line
1D          LD      R1,SaveR1    ; Restore registers
1E          LD      R2,SaveR2    ; to original values
1F          LD      R3,SaveR3
20          BRnzp   NEXT_TASK    ; Do the program's next task
21  ;
22  SaveR1  .BKLW   1            ; Memory for registers saved
23  SaveR2  .BKLW   1
24  SaveR3  .BKLW   1
25  DSR     .FILL   xFE04
26  DDR     .FILL   xFE06
27  KBSR    .FILL   xFE00
28  KBDR    .FILL   xFE02
29  Newline .FILL   x000A        ; ASCII code for newline
2A  Prompt  .STRINGZ ''Input a character>''
```

图8-5 LC-3的键盘输入例程

8.5 中断驱动I/O

由8.1.4节我们知道，处理器可以通过轮询方式与I/O设备交互，这是一种由处理器控制的交互方式。同样，也可以由I/O设备来控制该交互过程，即“中断”方式。参见8.2、8.3和8.4节中的轮询例子，它们都有一个共同点，即都需要处理器反复测试状态寄存器（Ready位）。直到Ready位被置“1”之后，程序才能执行输入或输出指令。而本节我们将介绍由I/O设备来控制的交互方式。

8.5.1 什么是中断驱动I/O

中断驱动I/O的本质特征是：I/O设备（可能与当前运行程序相关，也可能完全无关）能够实现下列功能：（1）强行中止当前程序的运行，（2）使得处理器执行I/O设备请求；（3）最后恢复被中断程序的执行，并让它感觉好像什么事情都没有发生过一样。中断发生时，指令执行流变化的三个阶段如图8-6所示。

```
                .
                .
                .
   Program A is executing instruction n
   Program A is executing instruction n+1
   Program A is executing instruction n+2
1: Interrupt signal is detected
1: Program A is put into suspended animation
2: The needs of the I/O device start being carried out
2: The needs of the I/O device are being carried out
2: The needs of the I/O device are being carried out
2: The needs of the I/O device are being carried out
2: The needs of the I/O device have been carried out
3: Program A is brought back to life
   Program A is executing instruction n+3
   Program A is executing instruction n+4
                .
                .
                .
```

图8-6 中断驱动I/O方式下指令的执行流程

对于被中断程序A而言，中断发生或没有发生，并不产生任何影响，即中断既不改变程序A的执行流程，也不影响它的执行结果。换句话说，即使发生中断，程序A本身的执行流程仍然如下：

```
                .
                .
                .
   Program A is executing instruction n
   Program A is executing instruction n+1
   Program A is executing instruction n+2
   Program A is executing instruction n+3
   Program A is executing instruction n+4
                .
                .
                .
```

8.5.2 为什么要引入中断驱动I/O

毫无疑问，如果I/O操作采用轮询方式，处理器将会花费大量时间探测Ready标志位。而在中断驱动I/O方式下则不同，如果没有中断发生，处理器可以执行其他程序；当中断发生时，处理器才暂停当前程序，并自动调用（由硬件完成）相应的I/O处理程序。

例8-1设想有一个程序，从键盘读取长度为100的字符串并进行处理。如果用户的键盘输入速度是每分钟80个单词或480个字符（平均每个单词有6个字符），即每0.125s一个字符。再假设处理这100个字符的序列需要12.49999s，试问连续处理1000个这样的字符串，需要多少时间？（为什么选择12.49999呢？这是为了使得结果更直观。）

我们可以采用如8.2节的轮询方式读入每个字符。如果是这样，程序将花费很长时间等待下一个字符的输入，读入100个字符的时间为100×0.125或12.5s。

相反，如果采用中断方式，则无需反复不断地执行LDI和BR指令以等待用户输入。那么，中断到来之前，可以忙着处理上一次读入的字符串。只是在中断到来之时，花很少的时间读入最新的输入字符。假设从I/O寄存器读数据需要10条指令，且每条指令的执行时间是0.00000001秒，则读取一个字符的时间仅为0.0000001s，100个字符的读入时间是0.00001s。用户输入100个字符需要12.5s，但中断方式下处理器读入字符串的时间仅为其中的0.00001s（而不是12.5s），剩下的12.49999s可用来做有用的工作，如处理上一次输入的字符序列。

我们的结论是：采用轮询方式，完成100个字符处理的时间是24.99999s（12.5s用来读入100个字符，12.49999s用来处理字符序列）。采用中断方式时，完成任务的时间是12.5s（0.00001s用来读入，12.49999s用来处理）。如果处理1 000个这样的序列，则轮询方式需要7h，中断方式只需要3.5h。

8.5.3 中断信号的产生

中断驱动I/O包括两部分内容，一是中断使能（enabling）机制，即I/O设备是如何通知处理器的（当设备有数据输入或其输出部件准备就绪时）；二是传输机制，即I/O数据如何在处理器和设备之间传送。这两部分可以简单地描述为：（1）产生中断信号，中止当前执行程序；（2）处理该中断请求。

我们很快将介绍第一部分内容。我们将看到，处理器中断当前工作，转而响应中断请求，需要考虑和执行很多细节工作。

第二部分内容的介绍要推迟到10.2节才会开始。LC-3在处理中断请求的时候，需要用到堆栈（stack）技术，而有关“堆栈”的知识，要在第10章之后才会学习。

设备是否必需，以及是否能够“中断”处理器（也即第一部分内容），必须具备以下几个条件：

（1）I/O设备自身确实需要服务。

（2）设备有请求服务的权限。

（3）设备中断请求的优先级高于当前处理器所运行程序的优先级。

以上三个条件如果同时满足，则处理器中止当前操作，并响应设备的中断请求。

1. 来自设备的中断信号

从I/O设备来说，中断信号的产生条件如上面列出的前两项所示，设备确实有请求服务的需求，且设备有足够的请求权限。

有关第一项条件，我们在轮询方式中已经提到，即KBSR或DSR寄存器的Ready位。换句话说，如果I/O设备是键盘，它表示用户已键入一个字符，需要被服务（即被读取）；如果I/O设备是显示器，则表示之前字符已输出到屏幕，请求继续服务（即请求写入下一个字符）。在这两种情况下，I/O设备都通过设置状态寄存器的Ready位来表示“请求服务”。

有关第二项条件，是指“中断使能标志”。处理器通过设置该标志位，控制I/O设备是否有上报中断的权限。如果该标志置1，表示处理器允许设备发送中断信号；否则，表示处理器禁止接收来自该设备的中断信号。通常，状态寄存器中包含有“中断允许标志”（IE）。如图8-7所示，KBSR和DSR的第14位即为中断允许标志，来自I/O设备的中断请求信号是IE和Ready位的逻辑与（AND）结果。



图8-7 中断使能位及其用法

如果IE标志被清0，则无论Ready标志是否置位，中断信号都不会产生。此时，程序只有通过轮询方式，才能获知设备是否已准备好新数据的传输。

如果IE标志被置1，表示该设备处在中断驱动I/O工作方式下，即允许设备发出中断。一旦设备准备好传输新数据，Ready位就被置1，设备就会产生中断信号。

2. 中断优先级

I/O设备中断的第三项条件是，中断请求比当前运行程序更紧迫。处理器所执行的每条指令都必然处在一定的紧迫级别下，我们称之为优先级。

几乎所有的计算机都定义了一组程序优先级别。如LC-3定义了8个优先级别（PL0~PL7），其数值越大对应的优先级越高。一个程序的PL通常与请求运行该程序的PL（即紧迫度）相同。如果程序当前正运行在一个特定的PL，而此时来了一个更高PL级别的计算机访问请求，则当前这个优先级较低的程序将被挂起，直到更高优先级的程序完成那个请求之后，被挂起的程序才被恢复执行。例如，如果计算机正在执行一个可以通宵运行的工资计算程序，由于它有足够的时间（一整夜）来运行，即紧迫度不高，所以我们可以将它的优先级设置为PL0；而如果正在运行的是一个核电厂发电机的电流浪涌（current surge）控制程序，则运行优先级应该设置为PL6。如果将两个程

序放在一台机器上运行，我们显然希望核电厂的控制程序应该优先于工资计算程序，以免核运转的控制延误而把我们炸成碎片。

若I/O设备能中止处理器的当前运行，并启动中断驱动I/O请求，则它的请求优先级必须高于当前运行程序的优先级。再如，如果是发送E-mail的按键输入，我们则不希望这个键盘输入能够“中断”之前提到的核电力控制程序的运行。

下面我们将看到，当INT信号产生时，处理器是怎样暂停当前运行而去响应中断请求的。如图8-8所示，是INT信号的产生机制以及中断优先级在其中起了作用。图中包含了多个设备的状态寄存器，同时这些设备也处在不同的优先级下。对于任何设备，如果它的状态寄存器的第14、15位同时置位，则该设备将发出中断请求信号。所有的中断信号又将被输入优先级编码器（一个组合逻辑电路），从中选出优先级最高的一个。被选出的优先级如果比当前运行程序的优先级高，则将成功生成INT中断信号，并中止当前程序。

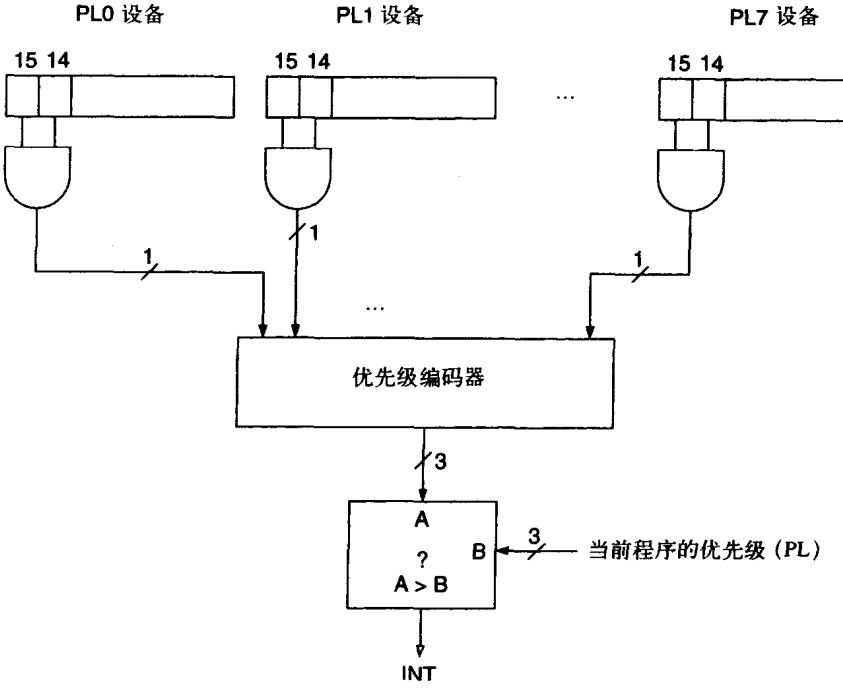


图8-8 INT信号生成电路

3. 中断检测

中断驱动I/O第一部分（即中断产生）的最后一步，是处理器对该中断信号的检测。从第4章的内容中，我们知道指令的执行过程包括6个节拍：取指令（FETCH）、译码（DECODE）、地址计算（EVALUATE ADDRESS）、取操作数（FETCH OPERAND）、执行（EXECUTE）和存放结果（STORE RESULT）。前一条指令完成“存放结果”节拍之后，控制单元将返回第一节拍，即下一条指令的取指操作。

在增加了中断信号检测逻辑之后，原先的最后一步操作——从前一个STORE RESULT节拍总是回到下一个FETCH节拍，将出现变化：在STORE RESULT的同时还将进行中断信号INT的测试。如果INT未被设置，则一切如旧，控制单元将直接返回FETCH节拍；如果INT信号有效，则控制单元在回到FECTH节拍之前，将完成两项工作：一是保存足够的状态信息，以备以后能正确恢复被中断程序的执行环境；二是将即将服务于该设备请求的程序入口地址装入PC寄存器。至于如何实现这些操作，是10.2节的话题，我们将在了解了栈（stack）技术之后，再学习该内容。

8.6 内存映射I/O的回顾

之前在图8-2和8-4中分别给出了内存映射输入和内存映射输出的实现。现在我们知道，如果要支持中断驱动I/O，设备状态寄存器必须是可读、可写的。

图8-9（同附录C中的图C-3）所示的数据通路，已具备了访问I/O设备寄存器所要求的全部特性。其中，地址控制逻辑（Address Control Logic）单元控制着输入、输出操作。注意，该模块有三个输入信号：(1) MIO.EN指示当前指令是内存操作还是I/O操作；(2) R.W信号指示是读操作还是写操作；(3) MAR寄存器中保存的内容则是内存地址（内存操作）或I/O寄存器地址（I/O操作）。基于三个输入信号，地址控制逻辑负责产生相应的控制信号。如果MIO.EN为0，地址控制逻辑将不做任何事情；否则，它将控制MDR和内存或I/O寄存器之间的数据传输。

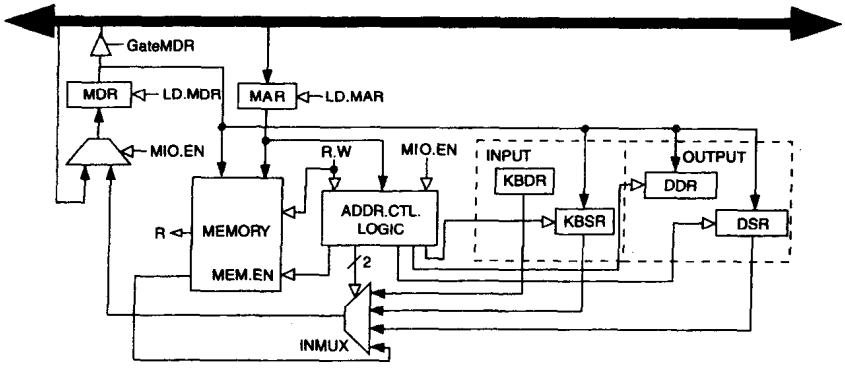


图8-9 内存映射I/O方式下的数据通路

如果R.W指示这是一个读（load）操作，数据的传输方向则是从内存或I/O寄存器至MDR寄存器。同时，地址控制逻辑将产生控制多路开关INMUX的输入源选择信号，即选择是内存还是I/O状态寄存器作为MDR的输入源。如果是读内存操作，除MAR中包含了内存地址之外，地址控制逻辑还将产生内存使能信号（MEM.EN）。

如果R.W指示这是一个写（store）操作，数据的传输方向则是从MDR寄存器至内存或I/O寄存器。地址控制逻辑将基于MAR的内容，向MAR所指向的内存模块或I/O寄存器发送相应的装入使能（load enable）信号。

8.7 习题

8.1 a.什么是设备寄存器？ b.什么是设备数据寄存器？ c.什么是设备状态寄存器？

8.2 为什么同步I/O不需要Ready标志位？

8.3 回顾8.1.3节中的例子，如果CPU工作频率为300MHz，即每33纳秒就能处理一个字符。假设每个单词的平均长度为6个字符(包括单词之间的空格)，求打字员每分钟要输入多少个单词才能跟上CPU的处理速度。

8.4 请问下面的交互方式是异步的还是同步的？

a. 电视和遥控器之间的交互。 b. 收件人和邮递员通过邮箱交互。

c. 鼠标和PC之间的交互。

再问，在什么条件下，它们都是同步的？在什么条件下，它们都是异步的？

8.5 试问，键盘寄存器KBSR中的bit[15]有什么作用？

8.6 如果程序在读取KBDR寄存器之前，不检查KBSR寄存器的Ready位，会出现什么问题？

8.7 下面哪两个描述合在一起，就可以描述8.2.2节描述的系统？

a. 内存映射和中断驱动的I/O。 b. 内存映射和轮询I/O。

c. 专用I/O指令和中断驱动的I/O。 d. 专用I/O指令和轮询I/O。

8.8 试编写一个程序，检查内存地址0x4000的值。如果该内存地址存储的是一个合法的ASCII码，程序就在屏幕上打印这个ASCII字符。如果该内存地址存储的是一个非法的ASCII码，程序就什么也不做。

8.9 如果键盘在向KBDR寄存器写入数据之前，不检查KBSR的Ready位，会发生什么问题？

8.10 如果屏幕输出电路在向DDR寄存器写入数据之前，不检查DSR寄存器的Ready位，会发生什么问题？

8.11 中断驱动I/O和轮询方式I/O相比，哪一个效率更高？为什么？

8.12 Adam H.设计了一个LC-3计算机的变种。该计算机的键盘不需要状态寄存器。相反，它设计一个被称做KBDSR的数据状态寄存器，该寄存器包含了与LC-3的KBDR寄存器完全相同的内容。基于KBDSR寄存器，输入程序只有在KBDSR内容不为0的时候，才从KBDSR中读取数据。这个非0值就是刚才键盘输入字符的ASCII码值。在数据读取之后，程序将清除KBDSR寄存器。请修改8.2.2节的代码，以使之能够适合Adam所设计的方案。

8.13 有一些计算机系学生决定改进LC-3的I/O设计，我们将新的计算机称为LC-4。在LC-4中，它们将键盘状态寄存器KBSR和显示器状态寄存器合并为一个寄存器：IOSR（输入/输出状态寄存器）。IOSR[15]是键盘的Ready位，IOSR[14]是显示设备的Ready位。这样，程序需要如何实现I/O操作？评价这个设计方案。

8.14 一个LC-3的load指令指定的地址是0xFE02。我们怎样才能知道载入的数据是来自KBSR寄存器还是来自某个内存单元？

8.15 关于中断驱动I/O：

a. 下面这段代码完成的是什么任务？

```
        .ORIG    x3000
        LD       R3, A
        STI      R3, KBSR
AGAIN   LD       R0, B
        ST       R0, A
        BRnzp    AGAIN
A       .FILL    x4000
B       .FILL    x0032
KBSR    .FILL    xFE00
        .END
```

b. 假设键盘中断矢量的值为0x34，内存单元0x0134的值为0x1000。键盘中断服务程序如下所示，该中断服务程序完成什么任务？

```
        .ORIG    x1000
        LDI      R0, KBDR
        TRAP     x21
        TRAP     x21
        TRAP     x25
KBDR    .FILL    xFE02
        .END
```

c. 如果程序（a）执行时用户键入一个字符，屏幕上将会出现什么？

8.16 下面这段LC-3程序完成的是什么任务？

```
        .ORIG   x3000
        LD      R0,ASCII
        LD      R1,NEG
AGAIN   LDI     R2,DSR
        BRzp    AGAIN
        STI     R0,DDR
        ADD     R0,R0,#1
        ADD     R2,R0,R1
        BRnp    AGAIN
        HALT
ASCII   .FILL   x0041
NEG     .FILL   xFFB6    ; -x004A
DSR     .FILL   xFE04
DDR     .FILL   xFE06
        .END
```

第9章 TRAP程序和子程序

9.1 LC-3 TRAP程序

9.1.1 概述

在前一章图8-5所示的代码中，程序要从键盘读取输入，程序员必须了解以下内容：

（1）数据寄存器（键盘和显示器）：用于确定显示器的数据输出和读取键盘输入字符的位置。

（2）状态寄存器（键盘和显示器）：用于确定何时可以向显示器输出下一字符，以及键盘是否有按键。

（3）键盘输入的异步工作机制。

对于大多数应用程序员，这些知识通常超出了他们的知识范围。事实上，如果要求一个应用程序员（又称用户程序员）也要了解底层I/O操作，将造成I/O设备减少，从事程序员职业的人也会变少。

赋予用户程序员直接读写KBDR和KBSR等I/O的权限，还将造成另一个问题。I/O操作的设备寄存器通常被很多程序共享。这意味着，如果允许用户级程序员直接读写硬件寄存器，一旦这些寄存器被误操作，必将影响其他用户程序的正常执行。所以，赋予程序员这种权限是个“愚蠢”的决定。通常，我们说硬件寄存器是“*有特权的*”，是指它们只能被那些有合适权限的程序访问。

但是，特权的概念也引发很多复杂问题。在此，我们暂不准备对其展开讨论，有关处理机制将在后面章节论述。目前，我们只需要知道有一些资源是用户程序无法直接访问的，它们只能被一些特权程序所掌控。回到刚才的话题，怎样为用户程序的输入和输出操作找到一种更好的操作办法呢？

一种比较简单和安全的解决方法，是借助于TRAP指令和操作系统。所谓“操作系统”，就是拥有特权权限的程序。

在第5章，我们曾经介绍过TRAP指令及其作用。对于一些特定任务，用户程序可以通过TRAP指令调用操作系统帮助完成。通过这种方法，用户程序员不需要了解之前提到的那些复杂细节，而其他用户程序也不会因为该程序员的失误遭致破坏。

如图9-1所示，用户程序在x4000地址处，将要执行I/O任务。它请求操作系统以该用户程序的身份完成这个任务。操作系统接过控制权，分析并处理TRAP指令传递的服务要求，然后将控制权交还给x4001地址的指令。我们称这种用户程序的请求为“*服务调用*”（service call）或“*系统调用*”（system call）。

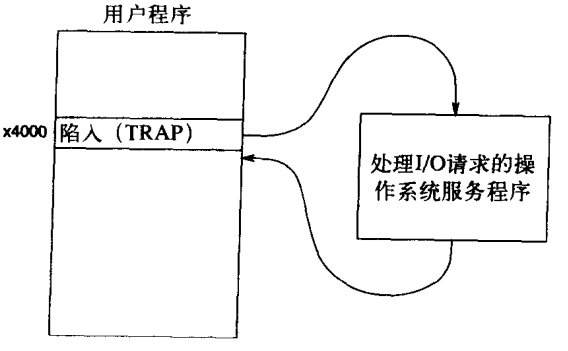


图9-1 使用TRAP指令调用OS的服务程序

9.1.2 TRAP机制

在TRAP机制中，包含以下要素：

（1）服务程序（service routine）集合：由操作系统提供，但以用户身份执行。这些服务程序是操作系统的组成部分，起始于各自固定的内存地址。LC-3最多可以支持256个服务程序。如附录A中表A-2所示，它是完整的LC-3操作系统服务程序列表。

（2）起始地址表：包含256个服务程序的起始地址。该表位于内存地址x0000~x00FF。在不同的操作系统中，该表的称法也不一样。有的称之为“系统控制块”（System Control Block），有的称之为“陷入矢量表”（Trap Vector Table）。图9-2所示是LC-3陷入矢量表的快照，其中给出了各矢量（服务程序）的起始地址。例如，地址x0021的内容是字符输出服务程序的起始地址（x0430），地址x0023的内容是键盘输入服务程序的起始地址（x04A0），地址x0025的内容是机器挂起服务程序的起始地址（xFD70）。

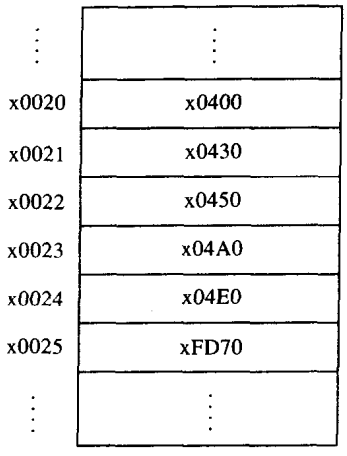


图9-2 陷入（trap）矢量表

（3）TRAP指令：用户程序如果希望操作系统以用户程序身份执行某个特定的服务程序，并在执行结束后将控制权返回，则可以使用TRAP指令。

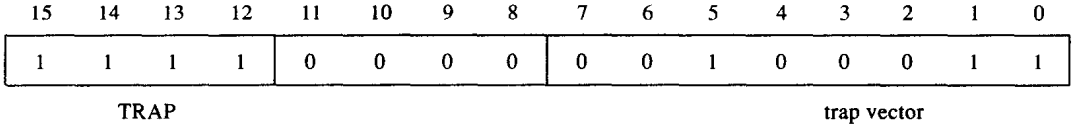
（4）链接（linkage）：通过链接回到用户程序。它意味着操作系统所提供的，从服务程序返回用户程序的机制。

9.1.3 TRAP指令

在执行服务程序之前，TRAP指令需要先完成两件事：

- 根据陷入矢量表项的内容，将PC值修改为对应于服务程序的起始地址。
- 提供一种机制，返回到调用TRAP指令的程序。我们称该“返回”机制为“链接”（linkage）。

“TRAP指令”由两部分组成：操作码1111和陷入矢量编号（bit[7:0]）。位[11:8]必须为0。陷入矢量项标识了用户程序希望操作系统执行的服务程序。如下所示，陷入矢量为x23。



TRAP指令在执行时，要完成4件任务：

（1）将8-bit的陷入矢量零扩展（zero-extend）为16-bit地址，并装入MAR。例如对于陷入矢量x23来说，扩展后的地址就是x0023，它代表某个陷入矢量表项的地址。

（2）陷入矢量表位于内存x0000~x00FF。随后，表项x0023的内容被读入MDR，即x04A0（如图9-2所示）。

（3）将当前PC值存入寄存器R7，以实现返回用户程序的链接机制。

（4）将MDR的内容装入PC。至此，完成TRAP指令。

由于PC的内容目前为x04A0，因而程序将从地址x04A0开始，继续执行。

x04A0指向的是操作系统“键盘读入”服务程序的起始。我们说，该陷入矢量“指向”该TRAP程序的起始。所以，TRAP x23指令的作用是激活操作系统的“键盘读入”服务程序。

TRAP指令能够返回用户程序下一条指令（服务程序结束后）的前提是，必须存在机制以保存下条指令的地址。如上所示，即执行阶段的步骤3实现的链接功能。即在服务程序地址加载到PC之前，先将原PC值存入R7。换句话说，TRAP指令为服务程序返回用户程序提供了所有必需的信息。由于在TRAP指令的预取阶段，PC已经更新，指向下一条指令，所以在trap服务程序开始执行时，R7中包含的就是TRAP指令的下一条指令地址。

9.1.4 完整机制

我们已介绍了TRAP指令激活服务程序的细节（对用户程序请求的响应），以及TRAP指令提供

的服务程序返回用户程序的机制（链接机制）。下面我们将介绍在服务程序中，返回用户程序所使用的指令。回顾第5章介绍的JMP指令。假设在trap服务程序执行过程中，R7的内容没有改变。那么，trap服务程序最后只要执行“JMP R7”，就可以返回用户程序。

图9-3所示是用LC-3 TRAP指令和JMP指令实现调用和返回的示例（参见图9-1）。其中，先是用户程序的*A*（等待键盘输入字符）转移至服务程序*B*，然会返回*C*（已获取键盘读入字符）。

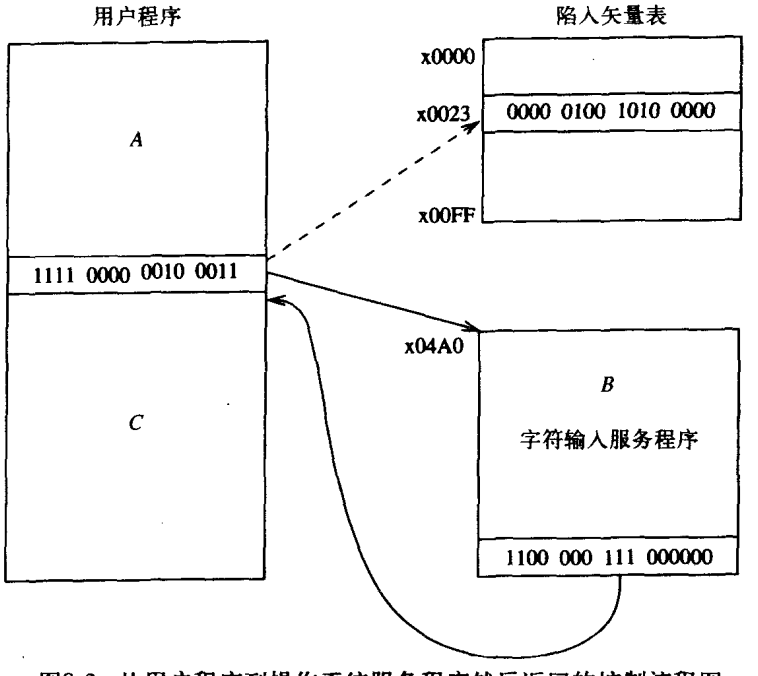


图9-3 从用户程序到操作系统服务程序然后返回的控制流程图

我们将借助计算机指令周期中的概念（取指、译码等），解释TRAP机制的实现细节。我们知道，若要改变程序控制流程，就要在当前指令的“执行”阶段修改PC值。之后的下一个周期，CPU将从新地址处读取指令。

因而，如果要请求字符输入服务，我们只需在用户程序中，调用TRAP指令的第x23号矢量即可。TRAP指令将从内存地址x0023读取其内容（x04A0），并将其加载到PC中，同时将下一条指令的地址装入R7。在图9-3中，虚线指向的是包含该服务程序起始地址的陷入矢量表项。

在下一个指令周期的FETCH节拍，将从x04A0开始执行，即请求（和接受）键盘输入的操作系统服务程序。该服务程序是8.4节键盘输入程序的变种（参见图8-5），其中R0包含的是输入键的ASCII码值。

该trap服务程序的结尾是“JMP R7”指令。它的作用是将R7的内容装入PC。如果R7的内容在服务程序的执行过程中没有改变，则该值就是用户程序中TRAP指令的下一条指令的地址。随后，用户程序继续执行，R0包含的则是刚才输入按键的ASCII码值。

在trap服务程序中，JMP R7指令非常有用（返回用户程序）。为此，在LC-3汇编语言中，专为这个操作定义了一个专用指令字RET（简短、清晰、方便记忆），如下所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
RET															

如下所示是一段使用TRAP指令的代码。它非常简单、有趣，可供4岁孩子玩耍。

例9-1 试编写游戏程序。假设一个人坐在键盘旁边，当他敲一个大写字母时，这个程序就输出对应的小写字母。如果他敲了7，这个程序就终止。

对应的LC-3汇编程序如下所示：

```
01          .ORIG x3000
02          LD    R2,TERM   ; Load -7
03          LD    R3,ASCII  ; Load ASCII difference
04    AGAIN TRAP  x23       ; Request keyboard input
05          ADD   R1,R2,R0  ; Test for terminating
06          BRz   EXIT      ; character
07          ADD   R0,R0,R3  ; Change to lowercase
08          TRAP  x21       ; Output to the monitor
09          BRnzp AGAIN     ; ... and do it again!
0A    TERM  .FILL xFFC9     ; FFC9 is negative of ASCII 7
0B    ASCII .FILL x0020
0C    EXIT  TRAP  x25       ; Halt
0D          .END
```

该程序的执行过程如下：首先，将数值xFFC9和x0020装入寄存器R2和R3。其中，数值xFFC9代表字符7的ASCII码的负值，它用于测试键盘输入字符，判断用户是否希望终止游戏；数值x0020则是同一个字母其大小写之间的ASCII码差值（零扩展后的）。例如，“A”的ASCII码为x41，“a”的ASCII码则为x61。又如，“Z”和“z”的ASCII码分别为x5A和x7A。

随后是TRAP x23指令，调用读入键盘的服务程序。服务程序结束后，返回应用程序的第05行。此时，R0的内容是输入字符的ASCII码。ADD和BRz两条指令的任务是测试输入字符是否为“7”。如果输入字符不是“7”，则将大小写ASCII码差值（x0020）加入输入值，存回R0。之后，TRAP调用（0x21）显示输出服务程序，将该小写字母显示到显示器上。当控制再次返回应用程序时（第09行），则又跳转到AGAIN，开始下一次的键盘读入。

在这个程序中，存在这样一个前提假设：用户只会大写字母或字符“7”。那么，如果用户输入字符“$”会有什么结果呢？对此，我们可以修改例9-1，在程序中检查输入字符，以确保输入字符是字母表中的26个大写字母。如果不是，则执行纠错操作。

思考题：试修改该程序，添加对无效数据检查的操作。换句话说，程序应保证，如果输入是大写字母，则输出对应的小写形式；如果是其他字符，则程序终止。见习题9.6。

9.1.5 I/O中断处理程序

我们只要将如图8-5所示的输入程序稍做修改，就可以将它改造为一个输入服务程序。修改后的程序如图9-4所示。其中的两个修改是：（1）增加了.ORIG和.END这些伪操作。.ORIG指定了该程序的起始地址（x04A0），即陷入矢量表x0023的内容；（2）将图8-5第20行所示的“BR NEXT_TASK”指令替换为“JMP R7”（汇编语言助记为RET）。之所以是JMP R7，是因为该程序是通过TRAP指令调用的（而不是普通的用户程序）。

另外，8.3.2节的输出程序也可以如此修改。修改后的输出服务程序如图9-5所示。结果是，我们可以通过TRAP指令和陷入矢量（编号），调用这些输入（见图9-4）和输出（见图9-5）服务程序。在读取输入的情况中，TRAP x23指令一旦结束，R0包含的就是键盘输入字符的ASCII码；而在输出情况下，用户程序只需将要显示字符的ASCII码装入R0，然后执行TRAP x21即可。

```
01      ;  Service Routine for Keyboard Input
02      ;
03              .ORIG   x04A0
04      START   ST      R1,SaveR1       ; Save the values in the registers
05              ST      R2,SaveR2       ; that are used so that they
06              ST      R3,SaveR3       ; can be restored before RET
07      ;
08              LD      R2,Newline
09      L1      LDI     R3,DSR          ; Check DDR --  is it free?
0A              BRzp    L1
0B              STI     R2,DDR          ; Move cursor to new clean line
0C      ;
0D              LEA     R1,Prompt       ; Prompt is starting address
0E                                      ; of prompt string
1F      Loop    LDR     R0,R1,#0        ; Get next prompt character
10              BRz     Input           ; Check for end of prompt string
11      L2      LDI     R3,DSR
12              BRzp    L2
13              STI     R0,DDR          ; Write next character of
14                                      ; prompt string
15              ADD     R1,R1,#1        ; Increment prompt pointer
16              BRnzp   Loop
17      ;
18      Input   LDI     R3,KBSR         ; Has a character been typed?
19              BRzp    Input
1A              LDI     R0,KBDR         ; Load it into R0
1B      L3      LDI     R3,DSR
1C              BRzp    L3
1D              STI     R0,DDR          ; Echo input character
1E                                      ; to the monitor
1F      ;
20      L4      LDI     R3,DSR
21              BRzp    L4
22              STI     R2,DDR          ; Move cursor to new clean line
23              LD      R1,SaveR1       ; Service routine done, restore
24              LD      R2,SaveR2       ; original values in registers.
25              LD      R3,SaveR3
26              RET                     ; Return from trap (i.e., JMP R7)
27      ;
28      SaveR1  .BLKW   1
29      SaveR2  .BLKW   1
2A      SaveR3  .BLKW   1
2B      DSR     .FILL   xFE04
2C      DDR     .FILL   xFE06
2D      KBSR    .FILL   xFE00
2E      KBDR    .FILL   xFE02
2F      Newline .FILL   x000A           ; ASCII code for newline
30      Prompt  .STRINGZ "Input a character>"
31              .END
```

图9-4 字符输入服务程序

```
01              .ORIG   x0430           ; System call starting address
02              ST      R1, SaveR1      ; R1 will be used to poll the DSR
03                                      ; hardware
04      ; Write the character
05      TryWrite  LDI   R1, DSR         ; Get status
06              BRzp    TryWrite        ; Bit 15 on says display is ready
07      WriteIt   STI   R0, DDR         ; Write character
08
09      ; return from trap
0A      Return    LD    R1, SaveR1      ; Restore registers
0B              RET                     ; Return from trap (JMP R7, actually)
0C      DSR       .FILL xFE04           ; Address of display status register
0D      DDR       .FILL xFE06           ; Address of display data register
0E      SaveR1    .BLKW 1
0F              .END
```

图9-5 字符输出服务程序

9.1.6 HALT中断程序

在4.5节中我们学过，通过RUN锁存门和晶振输出的“与”（AND）逻辑，可以控制计算机的运行。如果RUN锁存的位为0，则“与”门输出为0，即传输给整个计算机系统的时钟停止。

在早期的指令集体系结构中，都有一个HALT指令（清零RUN的内容），可用来停止时钟。但由于该指令的使用频率很低，为它分配一个操作符有些浪费（占用一个指令编码空间）。所以，在现代计算机中，改用TRAP指令方式来清除RUN门。在LC-3中，RUN门的内容对应机器控制寄存器（映射内存地址xFFFE）的第15位。如图9-6所示的trap服务程序，即可停止时钟，终止（halt）处理器。

```
01                  .ORIG   xFD70      ; Where this routine resides
02                  ST      R7, SaveR7
03                  ST      R1, SaveR1 ; R1: a temp for MC register
04                  ST      R0, SaveR0 ; R0 is used as working space
05
06  ; print message that machine is halting
07
08                  LD      R0, ASCIINewLine
09                  TRAP    x21
0A                  LEA     R0, Message
0B                  TRAP    x22
0C                  LD      R0, ASCIINewLine
0D                  TRAP    x21
0E  ;
0F  ; clear bit 15 at xFFFE to stop the machine
10  ;
11                  LDI     R1, MCR    ; Load MC register into R1
12                  LD      R0, MASK   ; R0 = x7FFF
13                  AND     R0, R1, R0 ; Mask to clear the top bit
14                  STI     R0, MCR    ; Store R0 into MC register
15  ;
16  ; return from HALT routine.
17  ; (how can this routine return if the machine is halted above?)
18  ;
19                  LD      R1, SaveR1 ; Restore registers
1A                  LD      R0, SaveR0
1B                  LD      R7, SaveR7
1C                  RET                ; JMP R7, actually
1D  ;
1E  ; Some constants
1F  ;
20  ASCIINewLine    .FILL   x000A
21  SaveR0          .BLKW   1
22  SaveR1          .BLKW   1
23  SaveR7          .BLKW   1
24  Message         .STRINGZ   "Halting the machine."
25  MCR             .FILL   xFFFE      ; Address of MCR
26  MASK            .FILL   x7FFF      ; Mask to clear the top bit
27                  .END
```

图9-6 LC-3的HALT服务程序

其中，第02、03、04行负责保存R7、R1和R0寄存器的内容。之所以保存R1和R0，是因为trap服务程序要借用它们；保存R7，是因为它的内容将被TRAP x21（第09行）覆盖；随后，第08~0D行负责在屏幕上输出提示信息“正在关机”（Halting the machine）；最后是RUN位（MCR[15]）清除操作（第11~14行），即将MCR内容和数值“0111 1111 1111 1111”相“与”（AND）。MCR[14:0]保持不变，MCR[15]变成0。*思考题*：能否设计一个指令（或trap服务程序）来控制时钟的开启呢？

9.1.7 寄存器内容的保存和恢复

在前面的内容中，曾不断地强调：在一些情况下，我们需要显式地保存寄存器的内容：

- 如果该寄存器的内容会被后续操作修改。
- 如果在后续操作中将使用该寄存器。

我们以一个例子讲解它的重要性。假设，程序从键盘读入10个数字，然后将ASCII码转化为二进制形式，存入从地址Binary处开始的10个连续内存空间。如下所示：

```
01              LEA     R3,Binary    ; Initialize to first location
02              LD      R6,ASCII     ; Template for line 05
03              LD      R7,COUNT     ; Initialize to 10
04   AGAIN      TRAP    x23          ; Get keyboard input
05              ADD     R0,R0,R6     ; Strip ASCII template
06              STR     R0,R3,#0     ; Store binary digit
07              ADD     R3,R3,#1     ; Increment pointer
08              ADD     R7,R7,#-1    ; Decrement COUNT.
09              BRp     AGAIN        ; More characters?
0A              BRnzp   NEXT_TASK    ;
0B   ASCII      .FILL   xFFD0        ; Negative of x0030.
0C   COUNT      .FILL   #10
0D   Binary     .BLKW   #10
```

其中，程序的第一步是初始化。首先将存储10个数字的内存起始地址装入R3；然后将ASCII模板（ASCII template）的负值装入R6，它将用做数字减法操作（减x0030）；随后，count的初始值10装入R7。之后是10次循环操作：每次从键盘读入一个字符，减掉ASCII模板值，存储二进制结果，检查是否完成了10次操作。听起来一切正常，但是我们发现该程序并未正常工作，为什么？答案是：每次执行TRAP指令（04行）时，R7的内容都将被修改为其下条指令“ADD R0. R0. R6”所在的地址，这破坏了原先装入R7的count的值（10）。所以，第08和09行代码永远无法结束循环过程。

该例子给我们的启示是：一个寄存器的原内容，如果在修改为其他值之后还要被使用，则在修改操作之前，必须要将原值保存，修改之后再将其恢复。保存的方法是，将寄存器内容存入内存中的某个位置。恢复时，将该值重新装入寄存器即可。如图9-6所示，第03行的ST指令将R1内容存入内存，第11行的LDI指令修改了R1的内容，trap服务程序尾部的第19行LD指令则将调用前的原值装入R1。其中，第22行设置的就是存储R1的内存空间。

有关保存/恢复问题，既可以由调用程序在TRAP之前负责，也可以由被调用者（TRAP执行之后）来负责。在9.2节中我们将看到，同样的问题在另一类调用/被调用程序（子程序）中也存在。

我们称由调用程序负责该问题的方式为“*调用者保存*”（caller-save）方式，而称由被调用者负责的方式为“*被调用者保存*”（callee-save）。选择最合适方式的原则是：“谁（调用者或被调用者）知道谁负责。”换句话说，对于特定的寄存器，谁知道该寄存器的内容会被修改，则由谁来负责保存。

对于被调用者来说，由于它知道自己的程序需要使用哪些寄存器。所以，它可以在执行开始之前，将这些寄存器的内容依次存入内存。执行结束时，再将这些原值恢复至寄存器。为此，我们在内存中预留了空间，以供寄存器值存储。例如，在图9-6中，HALT程序内部将使用R0和R1。所以，第03、04行的ST指令负责这些寄存器值的保存，第19、1A行的LD指令负责恢复，第21、22行是备份寄存器内容的内存空间。

对于调用者来说，也知道在它的操控之下，哪些寄存器的内容会被毁坏。例如，仍以图9-6为例，调用者知道TRAP指令必将修改R7的内容。所以，HALT服务程序在TRAP指令执行之前，就将R7保存；然后，在该TRAP指令执行之后，恢复R7的原值。

9.2 子程序

现在，我们看到，即使程序员不了解I/O的硬件细节，也可以有效地完成很多编程任务。只是，这些都依赖于操作系统所提供的服务程序。另外，我们也曾提到，让操作系统来访问设备寄存器，可以避免因程序员的误操作而产生的麻烦。

请求这些服务程序的方法很简单，仅仅是调用TRAP指令，剩下的事则由操作系统来处理。最后，服务程序通过JMP R7指令将控制权交还给用户程序。

同样，在一个用户程序中，如果某个程序片段被频繁使用。而我们又希望，不要在每次使用时，重复地实现这些细节。换句话说，我们希望代码可以被重复地使用。又如，某个人写的程序中，调用到另一个人所写的代码。

还有一种情况是，程序调用的部分代码是由生产商或第三方软件商开发的。并且，这些代码对用户程序员是可用的，我们称这样的代码集合为“库”（library）。例如，数学库就是一个例子，它包含了**平方根**、**正弦**和**反正切**等运算函数。

以上各种情况都表明，是否存在一种机制，可以帮助我们有效地享用这些代码。我们称这些代码为“子程序”（subroutine）或“过程”（procedure），而C语言的术语则称之为“函数”（function）。它们的使用机制则被称为“调用/返回机制”（Call/Return machanism）。

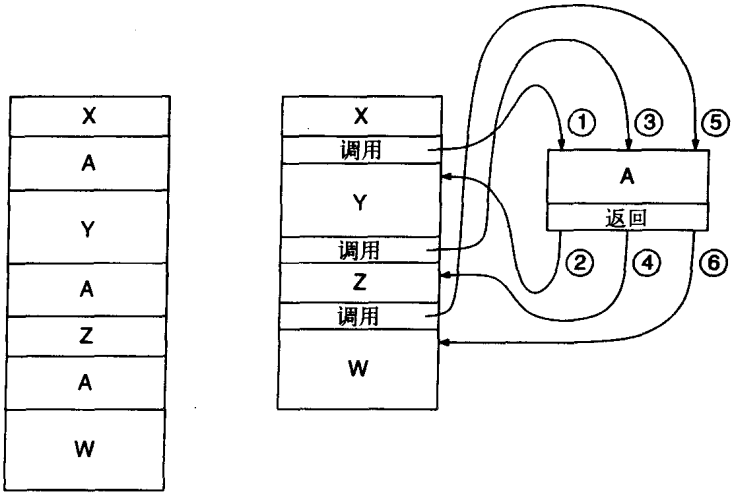
9.2.1 调用/返回机制

在图9-4中，提供了一段简单的程序片段，它在程序中将被多次调用。注意其中从L1标识开始的3条指令，以及从标识L2、L3和L4开始的3条指令。这些指令序列的形式都如下所示：

```
LABEL   LDI     R3,DSR
BRzp    LABEL
STI     Reg,DDR
```

这4段代码非常相似，只是在STI指令行存在区别。其中，2个存储的是R0的内容，另2个存储的是R2的内容，问题不大。如果采用调用/返回机制，我们就可以重复多次地执行这三条指令序列，而在整个程序中只需包含一次即可（子程序方式）。

调用机制的过程是，首先计算子程序的开始地址，并将其装入PC；然后，保存调用返回地址（下一条指令地址）。在返回机制中，将把返回地址再装入PC。图9-7所示是使用和不使用子程序等两种执行过程的比较。



a) 不使用子程序　　b) 使用子程序

图9-7 使用和不使用子程序的指令执行流程

调用/返回机制与TRAP指令的执行过程非常相似，两种情况下，都是先保存返回调用程序的链接地址，然后将程序流跳转至代码段（服务程序或子程序）。其中，PC装入的都是代码段的起始地址，R7装入的则是返回调用者的链接地址；代码段的最后一条语句，无论是在中断服务程序还是子程序中，都是JMP R7指令，即将R7的内容装入PC，从而将控制权返回到调用者。

子程序和服务程序之间的主要区别在于TRAP指令。也许这个话题超出了本课程的内容，不过我们还是简要讲一下。这涉及到被调用代码段的执行特性。在TRAP方式下，服务程序动用了操作系统资源，因而具备访问计算机底层硬件的特权。它们通常由系统程序员编写；相比之下，子程序和调用程序可以由同一个程序员编写（可以是你的同事或来自某个函数库）。但无论什么

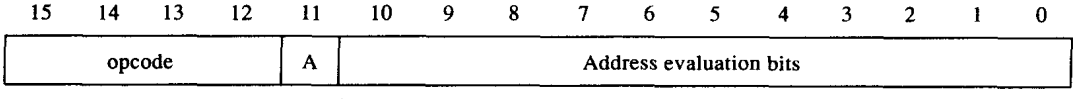
情况下，它们使用的资源都不会将程序的其他代码部分搞混乱，所以我们不将它们看做是用户程序的一部分。

9.2.2 JSR（R）指令

在LC-3中，子程序调用指令的操作码是0100。该指令的寻址模式（计算子程序的起始地址）有两种：PC相对地址寻址和基地址寻址。在LC-3汇编语言中，两种模式的操作码相同，但助记符不同（JSR和JSRR）。

该指令完成两件事：首先是将返回地址存入R7，然后是计算子程序起始地址并装入PC。返回地址是跳转前已经递增的PC值，它指向调用程序中紧随JSR（或JSRR）指令之后的那条指令。

JSR（R）指令由三部分组成：

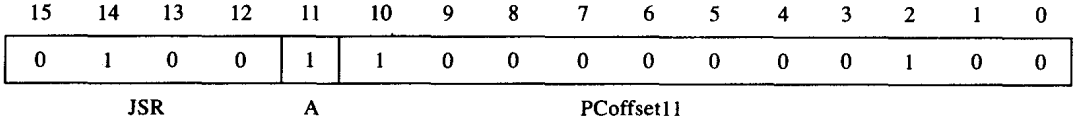


其中，bit[15:12]代表操作码（0010）；bit[11]代表寻址模式，该bit如果为1，代表“PC相对地址寻址”，0则代表“基地址寻址”；bit[10:0]用于子程序起始地址的计算。换句话说，JSR和JSRR之间的惟一区别，是子程序起始地址的寻址模式。

1. JSR

在JSR指令中，子程序目标地址的计算方法是，符号位扩展（16-bit）指令字中的bit[10:0]，然后将扩展后的16-bit与递增后的PC相加。该寻址模式与LD和ST的寻址模式几乎一样，只是PC的偏移量是11-bit，而LD和ST指令使用的是9-bit。

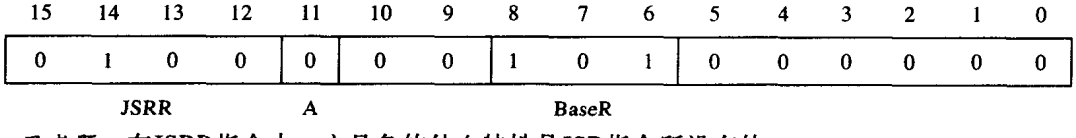
如果JSR指令所在的地址为x4200，那么该指令的执行结果是：PC = x3E05，R7 = x4201。



2. JSRR

JSRR指令和JSR指令相似（只是寻址模式不同）。它们计算子程序起始地址的方式是一样的，只是使用的是bit[8:6]所指定寄存器的内容。

例如，如下所示，假设JSRR指令所在地址为x420A，R5的内容为x3002。那么该JSRR指令的执行结果是，R7 = x420B，PC = x3002。



思考题：在JSRR指令中，它具备的什么特性是JSR指令所没有的？

9.2.3 字符输入的TRAP程序

下面，我们重新看一遍如图9-4所示的键盘输入服务程序。特别注意其中地址标识为L1、L2、L3和L4的三行代码：

```
LABEL   LDI     R3,DSR
        BRzp    LABEL
        STI     Reg,DDR
```

试问，我们可以用JSR/RET机制，将4处代码替换为一个子程序吗？答案“几乎”是肯定的。

图9-8所示是修改后的键盘输入服务程序。其中，第05、0B、11和14行的代码都替换为如下语句：

```
JSR     WriteChar
```

而第1D~20行是新加的4条语句（子程序WriteChar）：

```
WriteChar     LDI     R3,DSR
              BRzp    WriteChar
              STI     R2,DDR
              RET
```

其中，RET指令（实际上是JMP R7）的作用是：结束子程序，返回调用者。

注意前面问题的回答，用到的修饰词是“几乎”。原因是，在从L2和L3开始的代码中，STI指令存入DDR的应该是R0的内容（而不是R2的）。不过，该问题很容易解决，如图9-8的第09行，我们将原来的

```
LDR     R2,R1,#0
```

替换为

```
LDR     R0,R1,#0
```

这样，即将提示信息字符串里的字符装入了R2。然后，在子程序WriteChar中，再将R2中的字符传入DDR。

再看图9-8的第10行，我们插入这样一条指令：

```
ADD     R2,R0,#0
```

它的作用是将键盘输入值（在R0中）装入R2。然后，在子程序WriteChar中，通过R2传入DDR。注意，此时在R0中仍然存有键盘的输入值。另外，由于服务程序中没有修改R0的操作，所以R0将一直存有键盘输入值，直到返回用户程序。

而在图9-8的第13行，我们又插入了指令：

```
LD      R2,Newline
```

它的作用是将“换行”（newline）字符装入R2。然后，通过子程序WriteChar将它送入DDR。

最后，注意该程序与图9-4所示程序的不同。由于在该中断服务程序中，含有很多JSR指令。所以，R7所包含的链接地址，在进入服务程序之后很快就被修改了（如第03行的JSR）。因此，我们在开始执行第一条JSR指令之前（即第02行），保存了R7的内容；直到完成最后一条JSR指令之后（第16行），才恢复R7的内容。

图9-8所示是中断服务程序LC-3键盘读入操作的完整代码。

```
01                .ORIG   x04A0
02   START        ST      R7,SaveR7
03                JSR     SaveReg
04                LD      R2,Newline
05                JSR     WriteChar
06                LEA     R1,PROMPT
07   ;
08   ;
09   Loop         LDR     R2,R1,#0    ; Get next prompt char
0A                BRz     Input
0B                JSR     WriteChar
0C                ADD     R1,R1,#1
0D                BRnzp   Loop
0E   ;
0F   Input        JSR     ReadChar
10                ADD     R2,R0,#0    ; Move char to R2 for writing
11                JSR     WriteChar   ; Echo to monitor
12   ;
13                LD      R2, Newline
14                JSR     WriteChar
15                JSR     RestoreReg
16                LD      R7,SaveR7
17                RET                 ; JMP R7 terminates
18   ;                                  the TRAP routine
19   SaveR7       .FILL   x0000
1A   Newline      .FILL   x000A
1B   Prompt       .STRINGZ  "Input a character>"
1C   ;
1D   WriteChar    LDI     R3,DSR
1E                BRzp    WriteChar
1F                STI     R2,DDR
20                RET                 ; JMP R7 terminates subroutine
```

图9-8 提供字符输入的LC-3 trap服务程序

```
21  DSR         .FILL   xFE04
22  DDR         .FILL   xFE06
23   ;
24  ReadChar    LDI     R3,KBSR
25              BRzp    ReadChar
26              LDI     R0,KBDR
27              RET
28  KBSR        .FILL   xFE00
29  KBDR        .FILL   xFE02
2A   ;
2B   SaveReg      ST      R1,SaveR1
2C                ST      R2,SaveR2
2D                ST      R3,SaveR3
2E                ST      R4,SaveR4
2F                ST      R5,SaveR5
30                ST      R6,SaveR6
31                RET
32   ;
33   RestoreReg   LD      R1,SaveR1
34                LD      R2,SaveR2
35                LD      R3,SaveR3
36                LD      R4,SaveR4
37                LD      R5,SaveR5
38                LD      R6,SaveR6
39                RET
3A  SaveR1       .FILL   x0000
3B  SaveR2       .FILL   x0000
3C  SaveR3       .FILL   x0000
3D  SaveR4       .FILL   x0000
3E  SaveR5       .FILL   x0000
3F  SaveR6       .FILL   x0000
40               .END
```

图9-8 提供字符输入的LC-3 trap服务程序（续）

9.2.4 PUTS：写字符串

有关图9-8所示例子的最后一个内容是，第09~0D行的代码，将字符串“Input a character”输出到显示器上。我们通常称一个字符序列为“字符串”（character string）。这段代码在图9-6所示的代码中也曾出现过，即将字符串“Halting the machine”输出到显示器。由于在用户程序中，经常会将字符串输出到显示器。所以，在LC-3操作系统中，为此专门提供了一个对应的陷入矢量。如果用户程序需要把一个字符串输出到显示器，它只需要提供指向该字符串的首地址（在R0中），然后调用TRAP x22即可。在LC-3汇编语言中，我们称这个TRAP操作为“PUTS”。

PUTS（或TRAP x22）将控制权交给操作系统，然后执行如图9-9所示的代码。注意，只需对图9-8所示代码的第09~0D行代码稍做修改，就可以得到PUTS服务程序了。

```
01    ; This service routine writes a NULL-terminated string to the console.
02    ; It services the PUTS service call (TRAP x22).
03    ; Inputs: R0 is a pointer to the string to print.
04    ;
05                .ORIG   x0450           ; Where this ISR resides
06                ST      R7, SaveR7      ; Save R7 for later return
07                ST      R0, SaveR0      ; Save other registers that
08                ST      R1, SaveR1      ; are needed by this routine
09                ST      R3, SaveR3      ;
0A    ;
0B    ; Loop through each character in the array
0C    ;
0D    Loop        LDR     R1, R0, #0      ; Retrieve the character(s)
0E                BRz     Return          ; If it is 0, done
0F    L2          LDI     R3,DSR
10                BRzp    L2
11                STI     R1, DDR         ; Write the character
12                ADD     R0, R0, #1      ; Increment pointer
13                BRnzp   Loop            ; Do it all over again
14    ;
15    ; Return from the request for service call
16    Return      LD      R3, SaveR3
17                LD      R1, SaveR1
```

图9-9 LC-3的PUTS服务程序

```
18                  LD      R0, SaveR0
19                  LD      R7, SaveR7
1A                  RET
1B      ;
1C      ; Register locations
1D      DSR         .FILL   xFE04
1E      DDR         .FILL   xFE06
1F      SaveR0      .FILL   x0000
20      SaveR1      .FILL   x0000
21      SaveR3      .FILL   x0000
22      SaveR7      .FILL   x0000
23                  .END
```

图9-9 LC-3的PUTS服务程序（续）

9.2.5 库程序

我们注意到，在本节前面的很多地方，都使用了调用/返回机制。并且，我们认为用户程序所调用的库程序（library routine），应该属于计算机系统（而不是应用程序）。之所以提供库程序，是为了方便程序员在不需要知道内部细节的情况下，就可以使用它们所提供的功能，所以我们将它们形象地称为“生产力提高者”（productive enhancer）。例如，用户程序员知道平方根（简称SQRT）操作的含义，而他只需调用函数sqrt(x)，即可求得x的平方根值，而无需编写平方根的计算代码（甚至根本无需知道怎样编写）。

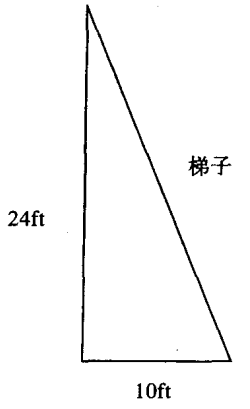


图9-10 求斜边长度的问题

下面，我们看一个简单的例子。假设办公室的钥匙丢了，于是放了一把梯子靠在墙上，使之刚好搭在窗户的下沿，且窗户距离地面的高度是24ft[⊖]。但是，墙边有一个10ft高的花坛，所以梯子必须架在花坛外面。试问，我们需要多长的梯子才能爬入窗户呢？换句话说，直角三角形的两个直角边分别为24ft和10ft，求斜边长度（见图9-10所示）？

这就是中学课本中的毕达哥拉斯（Pythagoras）定理[⊜]：

$$c^2 = a^2 + b^2$$

如果已知a和b，则求a^2和b^2之和的平方根可得c。求和操作比较简单，LC-3的ADD指令可以完成。平方也不难，通过反复的加法可得两数的乘积。但是，怎么计算平方根呢？如图9-11所示，我们给出了问题求解的程序框架。

```
01                  ...
02                  ...
03                  LD      R0,SIDE1
04                  BRz     1S
05                  JSR     SQUARE
06      1S          ADD     R1,R0,#0
07                  LD      R0,SIDE2
08                  BRz     2S
09                  JSR     SQUARE
0A      2S          ADD     R0,R0,R1
0B                  JSR     SQRT
0C                  ST      R0,HYPOT
0D                  BRnzp   NEXT_TASK
0E      SQUARE      ADD     R2,R0,#0
0F                  ADD     R3,R0,#0
10      AGAIN       ADD     R2,R2,#-1
11                  BRz     DONE
12                  ADD     R0,R0,R3
```

图9-11 计算直角三角形斜边的程序片段

⊖ 又称“勾股定理”。——译者注

⊜ 英尺（feet），1ft = 0.3048m。——编者注

```
13              BRnzp   AGAIN
14      DONE    RET
15      SQRT    ...             ; R0 <-- SQRT(R0)
16              ...             ;
17              ...             ; How do we write this subroutine?
18              ...             ;
19              ...             ;
1A              RET
1B      SIDE1   .BLKW   1
1C      SIDE2   .BLKW   1
1D      HYPOT   .BLKW   1
1E              ...
1F              ...
```

图9-11 计算直角三角形斜边的程序片段（续）

其中，缺少子程序SQRT的代码。在没有数学库的情况下，程序员就要拿起数学书（或者请人帮忙），学习Newton-Raphson方法，完成该计算方法的编程。

但是，如果有数学库（Math Libray），这个问题就简单多了。因为，数学库提供了包括SQRT在内的很多子程序。用户程序员不必关心Newton-Raphson之类的内容。惟一需要知道的，就是平方根函数在库程序中的目标地址，以及怎样传递参数x和获取SQRT(x)的计算结果。通过查询数学库的编程手册，我们很容易获得相关信息。

如果库函数在内存中的开始地址是SQRT，库函数要求的参数在R0中，返回结果也是存入R0中。那么，我们可以将图9-11的代码简化为如图9-12所示的代码。

```
01              ...
02              ...
03              .EXTERNAL SQRT
04              ...
05              ...
06              LD      R0,SIDE1
07              BRz     1$
08              JSR     SQUARE
09      1$      ADD     R1,R0,#0
0A              LD      R0,SIDE2
0B              BRz     2$
0C              JSR     SQUARE
0D      2$      ADD     R0,R0,R1  ; R0 contains argument x
0E              LD      R4,BASE
0F              JSRR    R4
10              ST      R0,HYPOT
11              BRnzp   NEXT_TASK
12      SQUARE  ADD     R2,R0,#0
13              ADD     R3,R0,#0
14      AGAIN   ADD     R2,R2,#-1
15              BRz     DONE
16              ADD     R0,R0,R3
17              BRnzp   AGAIN
18      DONE    RET
19      BASE    .FILL   SQRT
1A      SIDE1   .BLKW   1
1B      SIDE2   .BLKW   1
1C      HYPOT   .BLKW   1
1D              ...
1E              ...
```

图9-12 使用库程序解决图9-10问题的程序片段

其中，有两件事情值得一提：

- 第一，在本代码中，程序员不再担心计算平方根的问题了，因为库程序为我们做了。
- 第二，伪操作指令.EXTERNAL的使用。如前面7.4.2节中曾介绍过的，该伪操作指令的作用是告诉汇编器：在本程序中，如出现符号SQRT的引用，它将由外部程序（或模块）提供。例如，第19行的.FILL伪指令中出现的SORT。那么，在最后的可执行映像生成阶段（即链接阶段），编译时负责将本程序与该模块链接在一起。

在生成可执行映像时，将多个模块链接在一起是很常见的情况。图9-13阐述了这个过程。我们在本课程的第二部分，即在介绍C编程语言的工作机制时，将看到更具体的例子。

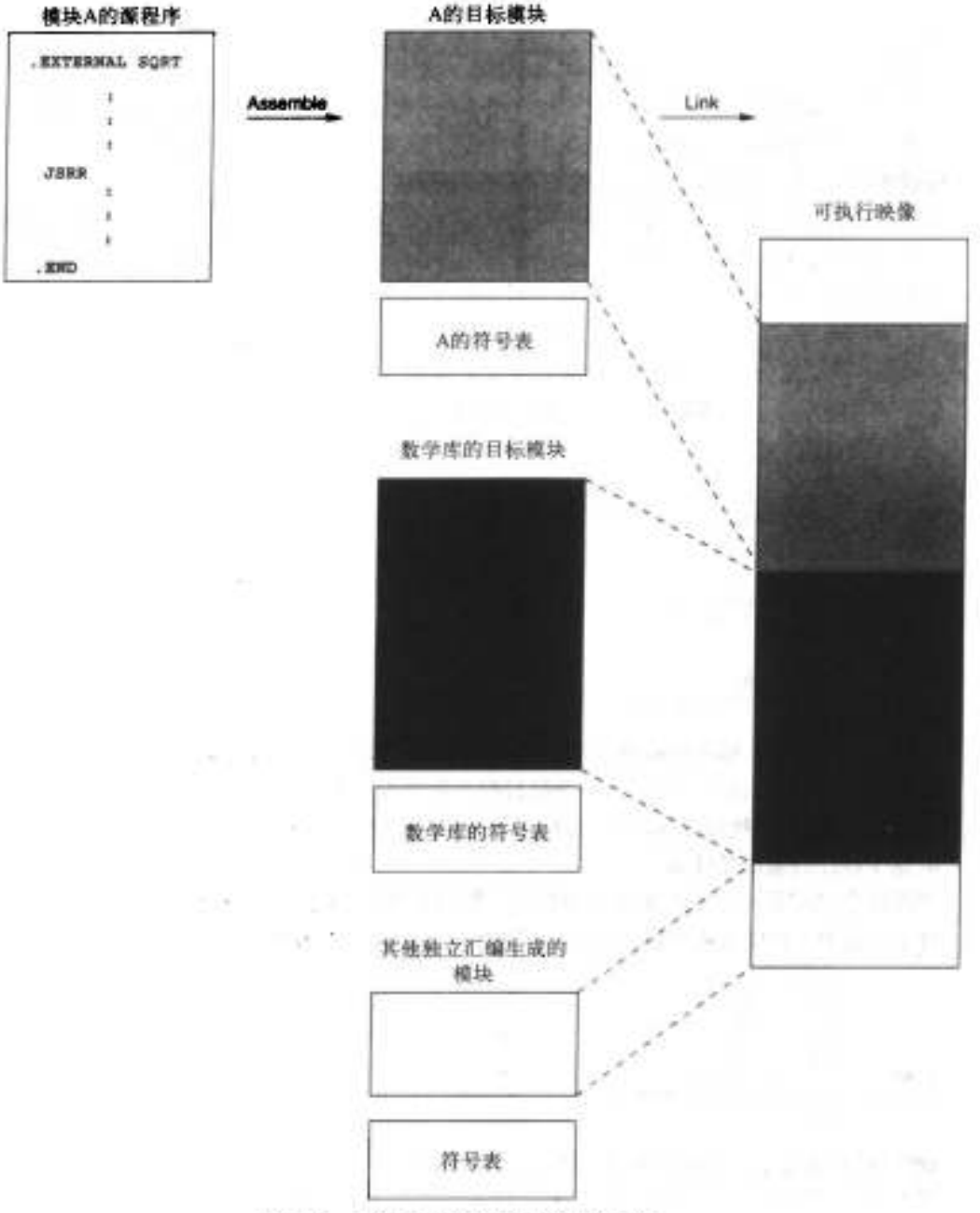


图9-13 由多个文件构成的可执行映像

在很多应用软件中，都将大量使用各种各样的库程序。如果让每个程序员都亲自编写一遍这些库程序，将浪费大量的人力（假设有某个程序员已为这个函数编写了非常优秀的代码）。例如，我们曾提到，在数学库中有很多能够绘制“漂亮”图像的预处理程序。再如其他的一些任务程序，如果让每个程序员都从头编写，显然是没有意义的。简单的方法是，使用库函数，而我们所需要做的只是：(1)用合适的文档清晰地描述库程序与调用程序之间的接口；(2)在源文件中，正确使用如.EXTERNAL等伪指令。最后，让链接器负责，将各个独立的汇编模块链接在一起，生成可执行映像。

9.3 习题

9.1 试阐述通过TRAP替代自己编写I/O操作程序的优点。

9.2 试问：

a. 在LC-3中，最多可有多少个trap服务程序？为什么？

b. 为什么TRAP程序返回时必须使用RET指令？为什么不能是BR（无条件跳转）指令？

c. 在TRAP指令的处理过程中，涉及多少次内存访问（假设TRAP已在IR中）？

9.3 阅读如图9-6所示的HALT服务程序，回答下列问题：

a. 在机器停止之后，怎样启动时钟？（提示：在HALT服务程序中，机器控制寄存器的bit[15]位被清除后，程序怎样返回？）

b. 什么指令可以真正终止机器？

c. 当机器重新启动时，第一条被执行的指令是什么？

d. 在HALT服务程序中，RET指令的返回点是哪里？

9.4 阅读下面这段LC-3汇编语言程序，并回答问题：

```
        .ORIG   x3000
L1      LEA     R1, L1
        AND     R2, R2, x0
        ADD     R2, R2, x2
        LD      R3, P1
L2      LDR     R0, R1, xC
        OUT
        ADD     R3, R3, #-1
        BRz     GLUE
        ADD     R1, R1, R2
        BR      L2
GLUE    HALT
P1      .FILL   xB
        .STRINGZ "HBoeoakteSmtHaotren!s"
        .END
```

a. 这个程序经过汇编并加载到内存后，在地址x3005处的二进制内容是什么？

b. x3005处的指令执行之后，下一条执行指令是什么（给出内存地址）？

c. x3006处的指令执行之前，执行的是哪一条指令（给出内存地址）？

d. 这个程序的输出是什么？

9.5 下面这个LC-3程序，在汇编之后被执行。假设忽略汇编时间和运行时错误，该程序的输出是什么？其中，假定在程序执行之前，所有寄存器的内容都为0。

```
        .ORIG    x3000
        STR      R0, x3007
        LEA      R0, LABEL
        TRAP     x22
        TRAP     x25
LABEL   .STRINGZ "FUNKY"
LABEL2  .STRINGZ "HELLO WORLD"
        .END
```

9.6 例9-1的正确性是，假设坐在键盘旁边的人只敲大写字母和7。但是，如果输入字符“$”，结果如何？更好的程序处理应该是，检查输入字符是否是“大写字母”，如果不是，则执行纠错操作。请修改例9-1所示代码，添加输入检查代码。即编写程序，如果输入为大写字母，则输出对应的小写字母；如果输入是其他字符，则终止执行。

9.7 假设两个学生分别写了两个中断服务程序，且两个服务程序所做的工作是相同的。如果第一个学生的程序结尾没有使用RET指令，而第二个学生正确使用了RET指令。那么，第一个学生的程序中将出现三个错误。请列举其中的任意两个错误。

9.8 假设，在下面这个程序执行前，内存地址A处原有的数据值介于2~32768之间。试描述该程序的执行结果（简要描述，不超过20个字）。

```
            .ORIG  x3000
            AND    R4, R4, #0
            LD     R0, A
            NOT    R5, R0
            ADD    R5, R5, #2
            ADD    R1, R4, #2
            ;
REMOD       JSR    MOD
            BRz    STORE0
            ;
            ADD    R7, R1, R5
            BRz    STORE1
            ADD    R1, R1, #1
            BR     REMOD
            ;
STORE1      ADD    R4, R4, #1
STORE0      ST     R4, RESULT
            TRAP   x25
            ;
MOD         ADD    R2, R0, #0
            NOT    R3, R1
            ADD    R3, R3, #1
DEC         ADD    R2, R2, R3
            BRp    DEC
            RET
            ;
A           .BLKW 1
RESULT      .BLKW 1
            .END
```

9.9 回顾之前的“机器忙”例子。假设表示机器忙或空闲的二进制字所在内存地址为x4001。请分别编写完成以下任务的子程序。

a. 检查是否所有机器都“空闲”，如果是则返回1；

b. 检查是否所有机器都“忙碌”，如果是则返回1；

c. 检查有多少台机器是忙碌的，返回忙碌机器的数量；

d. 检查有多少台机器是空闲的，返回空闲机器的数量；

e. 检查R5内容指定的机器是否忙碌，如果是则返回1；

f. 返回任意一台“空闲”机器的编号。

9.10 trap服务程序的起始地址，保存在TRAP指令所指定的内存位置中。试问，为什么不直接在TRAP指令中指定trap服务程序的起始地址？（假设每个trap服务程序最多包含16条指令。）试修改LC-3 TRAP指令的语义，使陷入矢量直接提供服务程序的起始地址。

9.11 如下是一个待编译（或汇编）的LC-3程序。该程序的任务是，从控制台读入包含若干行的文本输入，并存入缓存中以供字符查找，最后输出该字符在文本中出现的次数。假设文本的结束符为EOT，且文本长度不超过1000个字符。在输入整个文本后，程序读取待统计的字符。其中，标识为COUNT的子程序（即计数程序）由另一个人编写，且其所在地址为x3500。当该子程序被调用时，它默认R5的内容为缓存（buffer）地址，R6的内容为被统计字符。在缓存中，规定以NULL标记文本的结尾。最后，该子程序将计数结果存入R6。

子程序OUTPUT的作用是，将计数结果从二进制形式转换为ASCII字符（数字），并输出显示。该子程序也由另一个人编写，所在地址为x3600。它默认R6的内容是被打印的字符（数字）。

如下所示是读取输入和调用COUNT的代码：

```
        .ORIG  x3000
        LEA    R1, BUFFER
G_TEXT  TRAP   x20          ; Get input text
        ADD    R2, R0, x-4
        BRz    G_CHAR
        STR    R0, R1, #0
        ADD    R1, R1, #1
        BRz    G_TEXT
G_CHAR  STR    R2, R1, #0   ; x0000 terminates buffer
        TRAP   x20          ; Get character to count
        ST     R0, S_CHAR
```

```
        LEA     R5, BUFFER
        LEA     R6, S_CHAR
        LD      R4, CADDR
        JSRR    R4              ; Count character
        LD      R4. OADDR
        JSRR    R4              ; Convert R6 and display
        TRAP    x25
CADDR   .FILL   x3500           ; Address of COUNT
OADDR   .FILL   x3600           ; Address of OUTPUT
BUFFER  .BLKW   1001
S_CHAR  .FILL   x0000
        .END
```

但是，在该代码中存在问题。请找出问题，并给出修正方案（提示，问题不出在子程序COUNT和OUTPUT中）。

9.12 阅读下面这个LC-3汇编语言程序，然后回答问题：

```
        .ORIG   x3000
        LEA     R0,DATA
        AND     R1,R1,#0
        ADD     R1,R1,#9
LOOP1   ADD     R2,R0,#0
        ADD     R3,R1,#0
LOOP2   JSR     SUB1
        ADD     R4,R4,#0
        BRzp    LABEL
        JSR     SUB2
LABEL   ADD     R2,R2,#1
        ADD     R3,R3,#-1
        BRP     LOOP2
        ADD     R1,R1,#-1
        BRp     LOOP1
        HALT
DATA    .BLKW   10 x0000
SUB1    LDR     R5,R2,#0
        NOT     R5,R5
        ADD     R5,R5,#1
        LDR     R6,R2,#1
        ADD     R4,R5,R6
        RET
SUB2    LDR     R4,R2,#0
        LDR     R5,R2,#1
        STR     R4,R2,#1
        STR     R5,R2,#0
        RET
        .END
```

假设，内存地址DATA的内容在程序执行之前就已填充好。试问，在程序执行之前和之后，DATA的前后内容之间是否存在联系？

9.13 如下所示程序的任务是，在屏幕上打印数字5。但它无法正常工作，试问为什么（简要回答，不多于10个字）？

```
        .ORIG   x3000
        JSR     A
        OUT
        BRnzp   DONE
A       AND     R0,R0,#0
        ADD     R0,R0,#5
        JSR     B
        RET
DONE    HALT
ASCII   .FILL   x0030
B       LD      R1,ASCII
        ADD     R0,R0,R1
        RET
        .END
```

9.14 如图9-6所示，服务程序通过清除RUN门（即机器控制寄存器的bit[15]）来终止计算机运行（第14行）。试问，第19~1C行指令的作用是什么？

9.15 假设我们在内存地址x4000处又定义了一个新的服务程序。该程序的任务是，读入一个字符，然后将它显示在屏幕上。假设内存地址x0072中包含数值x4000，且服务程序如下所示，试

回答问题：

```
        .ORIG x4000
        ST R7, SaveR7
        GETC
        OUT
        LD R7, SaveR7
        RET
SaveR7  .FILL x0000
```

a. 给出调用这个程序的指令；

b. 该服务程序能否正常工作？请给出理由。

9.16 我们将下面a和b两段代码分别汇编。但在编译（或链接）时，出现两个错误，请找出问题并阐述原因，说明错误是在编译阶段还是在汇编阶段被检查到的。

a.

```
        .ORIG x3200
SQRT    ADD     R0, R0, #0
        ; code to perform square
        ; root function and
        ; return the result in R0
        RET
        .END
```

b.

```
        .EXTERNAL SQRT
        .ORIG   x3000
        LD      R0,VALUE
        JSR     SQRT
        ST      R0,DEST
        HALT
VALUE   .FILL   x30000
DEST    .FILL   x0025
        .END
```

9.17 如下所示程序的任务是，询问他/她的名字，然后输出字符串“Hello,名字”。该字符串所在位置是标识为HELLO的内存位置。程序假设用户在输入他/她的名字之后，按回车键（ASCII码 = x0A），且名字长度不超过25个字符。

例如，用户在看到提示后，输入名字“Onur”，随后敲入一个回车键，则程序输出如下所示：

```
Please enter your name: Onur
Hello, Onur
```

试在（a）~（d）处填入合适的指令，完成该程序。

```
        .ORIG x3000
        LEA   R1,HELLO
AGAIN   LDR   R2,R1,#0
        BRz   NEXT
        ADD   R1,R1,#1
        BR    AGAIN
NEXT    LEA   R0,PROMPT
        TRAP  x22           ; PUTS
        ------------ (a)
AGAIN2  TRAP  x20           ; GETC
        TRAP  x21           ; OUT
        ADD   R2,R0,R3
        BRz   CONT
        ------------ (b)
        ------------ (c)
        BR    AGAIN2
CONT    AND   R2,R2,#0
        ------------ (d)
        LEA   R0, HELLO
        TRAP  x22           ; PUTS
        TRAP  x25           ; HALT
NEGENTER .FILL xFFF6        ; -x0A
PROMPT  .STRINGZ "Please enter your name: "
HELLO   .STRINGZ "Hello, "
        .BLKW #25
        .END
```

9.18 如下所示程序，执行完成之后的输出内容为：

ABCFGH

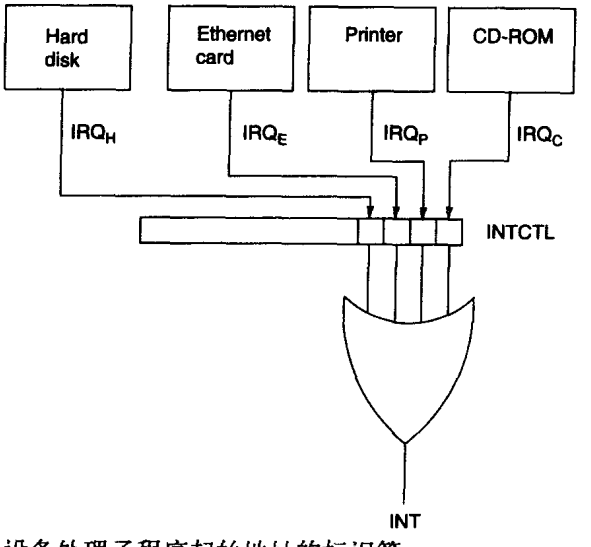
试在（a）~（d）之间插入合适的指令，完成该程序。

```
          .ORIG x3000
          LEA   R1, TESTOUT
BACK_1    LDR   R0, R1, #0
          BRz   NEXT_1
          TRAP  x21
          ------------ (a)
          BRnzp BACK_1
          ;
NEXT_1    LEA   R1, TESTOUT
BACK_2    LDR   R0, R1, #0
          BRz   NEXT_2
          JSR   SUB_1
          ADD   R1, R1, #1
          BRnzp BACK_2
          ;
NEXT_2    ------------ (b)
          ;
SUB_1     ------------ (c)

K         LDI   R2, DSR
          ------------ (d)

          STI   R0, DDR
          RET
DSR       .FILL xFE04
DDR       .FILL xFE06
TESTOUT   .STRINGZ "ABC"
          .END
```

9.19 假设一家公司计划制造一台真正的LC-3计算机。为了让该计算机能工作在网络环境中，我们为它安装4个中断驱动类型的I/O设备链接。如果接收到服务请求，其中一个设备就会向系统发出独立的中断请求信号（IRQ），结果是LC-3的中断控制寄存器INTCTL中的某个bit将被置位（INTCTL的内存映射地址为xFF00）。INTCTL寄存器的字段组成如下所示。当某个设备请求服务时，LC-3数据通路产生INT信号，中断服务程序将判断是哪个设备发出的服务请求，然后调用相应的设备服务子程序处理该请求。如果是多个设备同时发出IRQ信号，则只有优先级最高的设备先被处理。在子程序执行过程中，相应的INTCTL位将被清掉。



如下所示是4个设备处理子程序起始地址的标识符：

HARDDISK ETHERNET PRINTER CDROM

例如，优先级最高的设备“打印机”请求服务，则中断服务程序将通过如下指令调用打印机处理子程序：

JSR PRINTER

按照优先级规则，请填写标号为（a）~（k）处的指令，完成整个LC-3中断服务程序。优先级顺序如下所示（其中，数值越低，代表的设备优先级越高）。

（1）Hard disk;

（2）Ethernet card;

（3）Printer;

（4）CD-ROM。

```
        LDI   R1, INTCTL
DEV0    LD    R2, ------ (a)
        AND   R2, R2, R1
        BRnz  DEV1
        JSR   ---------- (b)
        ---------------- (c)
  ;
DEV1    LD    R2, ------ (d)
        AND   R2, R2,  R1
        BRnz  DEV2
        JSR   ---------- (e)
        ---------------- (f)
  ;
DEV2    LD    R2, ------ (g)
        AND   R2, R2, R1
        BRnz  DEV3
        JSR   ---------- (h)
        ---------------- (i)
  ;
DEV3    JSR  ---------- (j)
  ;
END     ---------------- (k)

INTCTL  .FILL   xFF00
MASK8   .FILL   x0008
MASK4   .FILL   x0004
MASK2   .FILL   x0002
MASK1   .FILL   x0001
```

第10章 栈

到目前为止，我们终于完成了LC-3指令集结构的内容。从下一章（第11章）开始，我们将学习更高层次的抽象——C语言编程。但在本章中，我们仍将花一些时间，介绍一个非常重要的话题：栈（stack）。首先介绍栈的基本结构，然后介绍栈的三种用途：（1）中断驱动I/O（参见8.5节）；（2）一种算术运算机制（通过用栈替代寄存器，实现计算中间结果的临时存储）；（3）二进制补码与ASCII字符串之间的转换算法。在这三个例子中，我们只展示了栈的“冰山一角”（tip of the iceberg）。事实上，栈在计算机科学与工程中占据着非常重要的地位。我们相信，将来你必定会发现更多的栈的新用法。届时，不知本书对栈的介绍是否能给你带来美好回忆。

有关指令集结构（ISA）层的介绍，最后的一个例子是“计算器设计”。这是一个复杂的应用，它将涉及本章的很多知识。

10.1 栈的基本结构

10.1.1 抽象数据类型：栈

在今后的计算机使用或设计中，你必然会接触到一种被称为“栈”（stack）的存储机制。栈机制的实现方法有多种。但是，我们首先要介绍的是栈的概念（而不是它的实现）。所谓“栈”的概念，是指它的访问规则。“栈”的定义是：最后存入的东西，总是第一个被取走的。这是栈与其他结构相区别的主要特点。我们称这种特点为“后入先出”或“LIFO（Last In First Out）”。

按照计算机的术语来说，我们称栈是一种*抽象数据类型*（Abstract Data Type, ADT）。换句话说，抽象数据类型是指其存储机制的操作方式（而不是实现方法）。例如，第19章中将要介绍的链表（linked list），也是一种抽象数据类型。

10.1.2 两个实现例子

例如，汽车扶手边上的硬币盒（coin holder）就是“栈”的一个例子。第一个被取出的硬币，必然是最后放入的硬币。而你放入硬币时，将把先前的硬币又向硬币盒内部压入了一层。

如图10-1所示，是硬币盒的使用示例行为。最开始的情况如图10-1a所示，硬币盒是空的。假设路过第一个收费站的时候，应缴费用是75美分。你交给收费员1美元，找回25美分（1995年的），然后你将它塞入硬币盒。此时，硬币盒的情况将如图10-1b所示。

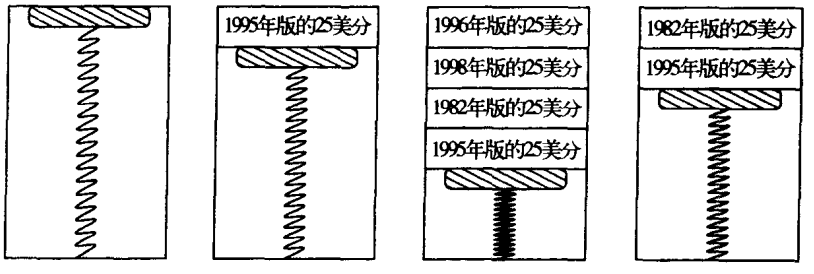


图10-1 一个栈的例子——汽车上的硬币存放器

有关“栈”的插入和删除操作，我们有特定的术语。插入一个元素时，我们称之为“压入”（push）；删除一个元素时，我们称之为“弹出”（pop）。

随后，第二段高速公路费是4.25美元，你交给收费员5美元，找回75美分。你又将这3枚25美分塞入硬币盒：先是1982年的25美分硬币，然后是1998的，最后是1996的。现在，硬币盒的情况将如图10-1c所示。在经过第三段高速公路收费站时，费用是50美分，所以你从硬币盒中取出两枚25美分，它们一个是1996年的，另一个是1998年的。之后，硬币盒的情况将如图10-1d所示。

我们说硬币盒是一个“栈”，更确切地说，它符合LIFO规则。新放入的硬币，总是在最上面；每取走一个硬币，也总是最上面的。最后放入的硬币，总是第一个被取走的，所以我们说它是一个栈。

图10-2是又一个栈实现的例子，我们称之为“硬件栈”（hardware stack）。它的行为与之前的硬币盒很相似。它由一组寄存器组成，每个寄存器可以存放一个元素（element）。如图10-2所示的例子中，包含有5个寄存器。每当一个元素被存入或移出时，栈中其他元素都将一起移动。

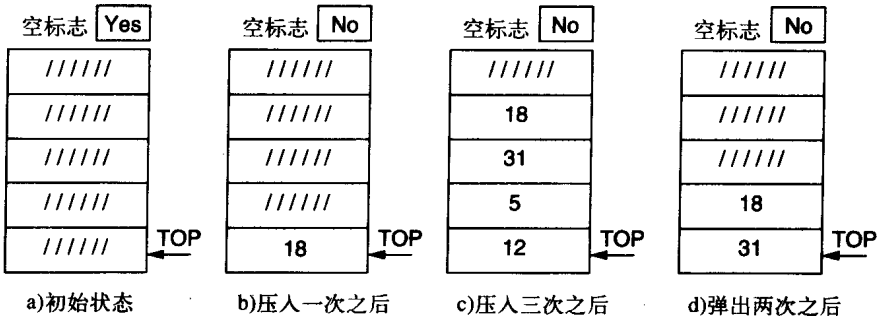


图10-2 一个硬件实现的栈——数据条目需要移动

如图10-2a所示，初始的栈是空的。栈的访问总是针对第一个元素的，我们将该寄存器标识为TOP。例如，数值18被压入栈，则结果如图10-2b所示。随后，依次压入数值31、5和12，则结果如图10-2c所示。再如，从栈中弹出两个元素，则结果如图10-2d所示。图10-2所示的硬件栈，与硬币收集器一样，有一个共同的特点：每压入或弹出一个栈元素，栈中所有元素都将一起“移动”。

10.1.3 内存中的实现

在各种计算机系统中，最常见的栈实现方式如图10-3所示，它是由一段连续内存空间和一个寄存器（栈指针）组成。所谓“栈指针”（stack pointer），是一个寄存器，它的内容是一个地址值，始终指向栈的顶部（即最近被压入的元素）。每个被压入栈中的元素，在内存空间中都占据着一个独立的位置。但是，在入栈和出栈操作时，栈中的其他数据不需要再被移动。

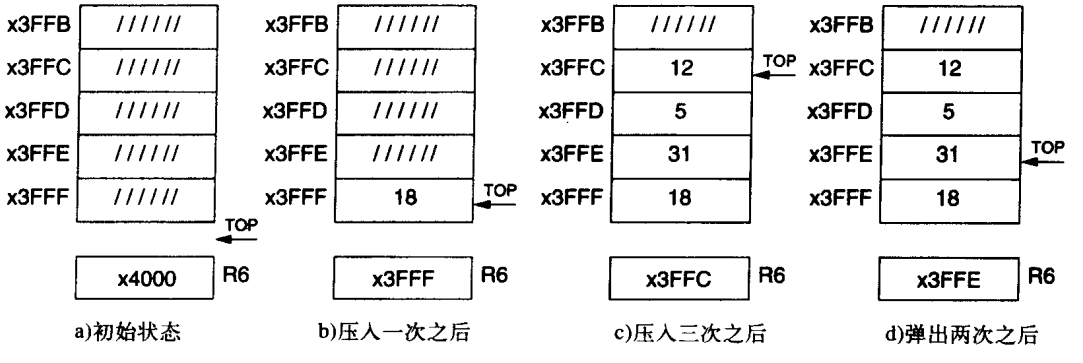


图10-3 一个在内存里实现的栈——数据条目不需要移动

图10-3所示是该栈的组成结构。其内存占用范围（大小为5）是：x3FFF~x3FFB，寄存器R6是栈指针。

其中，图10-3a代表最初的空栈；图10-3b是压入值18后的情况；随后，如图10-3c所示，是顺序压入数值31、5和12之后的情况；最后，如图10-3d所示，是从栈顶弹出两个元素后的情况。注意，两个元素（值5和12）即使已被弹出，其内容仍然存留在内存位置x3FFD和x3FFC。但是，数值5和12已不能再被访问，这是因为该段内存只能通过栈机制来访问。

1. 压入

如图10-3a所示，R6的内容是x4000，即第一个栈位置（基地址）之前的一个位置。它表明此时的栈是空的。图10-3所示的栈，基地址（BASE）等于x3FFF。

首先，我们压入（PUSH）数值18，结果如图10-3b所示。栈指针指向的是最后压入的值，即x3FFF（存储18的位置）。注意，地址x3FFE、x3FFD、x3FFC和x3FFB的内容没有显示出来。我们很快会明白，它们的内容其实是无关的。因为我们并不能通过x3FFF~x3FFB的地址直接访问它们，只能通过栈方式来访问。

我们看到，每压入一个值到栈中，栈指针先递减，然后将数值存入它的地址位置。如下面两条指令序列所示：

```
PUSH    ADD   R6,R6,#-1
        STR   R0,R6,#0
```

其中，这两条指令的任务是将R0的内容压入栈。所以，如果如图10-3b所示，在这两条指令执行之前，R0的内容应该已经为数值18。

随后，我们将31、5和12等三个数值依次装入R0，并调用这两条指令序列，将它们压入栈中。如图10-3c所示，R6（栈指针）的值为x3FFC，这意味着数值12是最后一个被压入栈的元素。

2. 弹出

如果想从栈里弹出（POP）一个元素，要先通过栈指针提供的地址读取该数值，然后递增栈指针。如下面两条指令序列所示：

```
POP     LDR   R0,R6,#0
        ADD   R6,R6,#1
```

其中，栈顶元素的值被弹出，并装入R0。

如图10-3c所示，在两次执行以上指令序列之后，将有两个数值被弹出栈。先是12，然后是5。假设我们弹出两个值的目的是为了使用它们，则在第二次POP调用前，R0原先的内容12应拷贝到别的地方。

图10-3d所示是上述操作执行之后栈的情况。其中，R6的内容是x3FFE（代表当前栈顶元素是31）。注意，内存地址x3FFD和x3FFC的内容仍然是12和5。但是，由于对栈空间操作时，只能通过PUSH压入、POP弹出。所以，对这个空间的操作，只要严格遵循该规则，一切都会正常。我们将这个规则称为“栈协议”（stack protocol）。

3. 下溢出

试问，如果我们现在尝试从栈中弹出三个数值，结果会如何？由于此时栈内只有两个元素，所以问题出现了。试图对一个空栈执行弹出操作，将造成“下溢”（underflow）问题。在下面的例子中，我们将通过地址值x4000与栈指针内容之间的比较，检查是否发生“下溢”现象。其中，x4000是栈为空时，R6的值；标识UNDERFLOW代表下溢处理程序的起始地址。修订后的POP指令序列如下所示：

```
POP     LD      R1,EMPTY
        ADD     R2,R6,R1        ; Compare stack
        BRz     UNDERFLOW       ; pointer with x4000.
        LDR     R0,R6,#0
        ADD     R6,R6,#1
        RET
EMPTY   .FILL   xC000           ; EMPTY <-- -x4000
```

但是，与其让程序控制流在POP不成功时，跳转至UNDERFLOW程序，不如让它返回到调用程序（并将“下溢”信息记录在某个寄存器中）。

常用的做法是，子程序将执行成功或失败的信息记录在某个寄存器中。如图10-4所示，在POP程序的流程中，将成功或失败的信息记录在R5中。

那么，POP程序返回后，调用程序通过R5就可以获知POP执行是否成功（如果成功，R5=0；否则，R5=1）。

注意，由于POP程序将使用R5记录执行状况。所以，R5的原内容（POP执行之前）将丢失。因此，这就要求调用程序在JSR指令执行之前，保存R5的内容。参考9.1.7节中的调用者保存方式。

再次修改之后，POP程序如下所示。注意，RET指令之前的那条指令能影响“条件码”，所以调用程序也可进行条件位Z的测试，判断POP程序的执行是否成功。

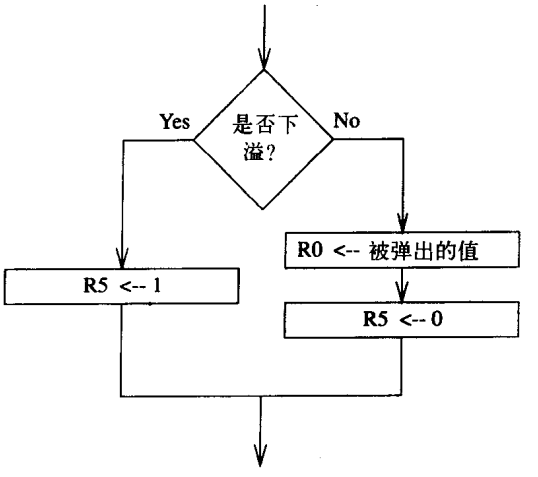


图10-4 包含了“下溢”检查的POP程序

```
POP       LD      R1,EMPTY
          ADD     R2,R6,R1
          BRz     Failure
          LDR     R0,R6,#0
          ADD     R6,R6,#1
          AND     R5,R5,#0
          RET
Failure   AND     R5,R5,#0
          ADD     R5,R5,#1
          RET
EMPTY     .FILL   xC000          ; EMPTY <-- -x4000
```

4. 上溢出

试问，如果在栈空间已被全部占用的情况下，再往栈中压入元素，结果会如何？由于已经没有可用的栈空间，所以我们将遇到被称为“上溢”（overflow）的问题。当然，通过比较栈指针内容与数值x3FFB（如图10-3所示），可以判断是否发生“上溢”。如果两个值相等，则意味着栈空间已无法再存储元素了。假设OVERFLOW是“上溢”处理程序的起始地址，则修改后的PUSH程序如下所示：

```
PUSH      LD      R1,MAX
          ADD     R2,R6,R1
          BRz     OVERFLOW
          ADD     R6,R6,#-1
          STR     R0,R6,#0
          RET
MAX       .FILL   xC005          ; MAX <-- -x3FFB
```

同POP程序一样，如果我们能够记录程序执行状态，并返回调用程序再处理，则要比直接跳到OVERFLOW子程序更有意义。

所以，我们要求PUSH程序在执行成功之时，将数值0存入R5，失败之时则将1存入R5。于是，PUSH程序返回后，调用程序通过检查R5的内容，即可判断PUSH的执行是否成功（R5=0表示成功，R5=1表示失败）。

再次注意，由于R5被用于存放PUSH程序的执行状态（成功或失败）。所以，这又是一个调用者保存的例子。换句话说，为避免R5在PUSH程序执行之前的内容丢失，调用程序要在JSR指令执行之前，保存R5的内容。

同样，因为RET之前的指令设置了条件位，所以调用程序也可以通过条件位Z或P来判断PUSH程序的执行是否成功完成了。PUSH程序如下所示：

```
PUSH    LD      R1,MAX
        ADD     R2,R6,R1
        BRz     Failure
        ADD     R6,R6,#-1
        STR     R0,R6,#0
        AND     R5,R5,#0
        RET
Failure AND     R5,R5,#0
        ADD     R5,R5,#1
        RET
MAX     .FILL   xC005          ; MAX <-- -x3FFB
```

10.1.4 小结

通过子程序POP和PUSH，我们可以将5个内存位置x3FFF~x3FFB，设计为一个栈。例如，如果将一个数值压入栈，只需将该数值装入R0，然后执行“JSR PUSH”即可；如果弹出栈的元素，则执行JSR POP即可（结果在R0中）。如果希望改变栈的空间位置或大小，只需调整代码中BASE和MAX的值。

在话题结束之前，我们还需要注意一些细节。在PUSH和POP子程序中，使用了R1、R2和R5这三个寄存器。如果希望在PUSH或POP程序返回调用程序后，还能够继续使用之前存在它们中的数据，则需要在使用前保存它们的内容。以R1和R2为例，我们可以在PUSH和POP子程序内部，在使用它们之前保存它们，然后在返回调用程序之前恢复它们的原值。这或许是最简单的做法，并且这样一来，调用程序就不需要了解在PUSH和POP中都使用了哪些寄存器，即9.1.7节介绍的“被调用者保存”方式。但是，对于R5则有些不同。事实上，调用程序将通过R5了解调用执行状态（成功或失败）。所以，R5将由调用程序负责保存（如果在子程序执行后还将继续使用R5原值），即“调用者保存”方式。

最终版的PUSH和POP代码，如图10-5所示。

```
01  ;
02  ; Subroutines for carrying out the PUSH and POP functions.  This
03  ; program works with a stack consisting of memory locations x3FFF
04  ; (BASE) through x3FFB (MAX).  R6 is the stack pointer.
05  ;
06  POP             ST      R2,Save2        ; are needed by POP.
07                  ST      R1,Save1
08                  LD      R1,BASE         ; BASE contains -x3FFF.
09                  ADD     R1,R1,#-1       ; R1 contains -x4000.
0A                  ADD     R2,R6,R1        ; Compare stack pointer to x4000.
0B                  BRz     fail_exit       ; Branch if stack is empty.
0C                  LDR     R0,R6,#0        ; The actual "pop"
0D                  ADD     R6,R6,#1        ; Adjust stack pointer.
0E                  BRnzp   success_exit
0F  PUSH            ST      R2,Save2        ; Save registers that
10                  ST      R1,Save1        ; are needed by PUSH.
11                  LD      R1,MAX          ; MAX contains -x3FFB
12                  ADD     R2,R6,R1        ; Compare stack pointer to -x3FFB.
13                  BRz     fail_exit       ; Branch if stack is full.
14                  ADD     R6,R6,#-1       ; Adjust stack pointer.
15                  STR     R0,R6,#0        ; The actual "push"
16  success_exit    LD      R1,Save1        ; Restore original
17                  LD      R2,Save2        ; register values.
18                  AND     R5,R5,#0        ; R5 <-- success.
19                  RET
1A  fail_exit       LD      R1,Save1        ; Restore original
1B                  LD      R2,Save2        ; register values.
1C                  AND     R5,R5,#0
1D                  ADD     R5,R5,#1        ; R5 <-- failure.
1E                  RET
1F  BASE            .FILL   xC001           ; BASE contains -x3FFF.
20  MAX             .FILL   xC005
21  Save1           .FILL   x0000
22  Save2           .FILL   x0000
```

图10-5 栈协议

10.2 中断驱动I/O（第二部分）

之前，我们在8.1.4节介绍了中断驱动I/O的第一部分内容。我们已知，在轮询方式下，处理器将花费太多的执行周期在LDI和BR指令上，直到Ready位被置位。而在中断驱动I/O方式下，处理器不再为LDI和BR指令的重复执行浪费时间。相反，它可以将省出的时间用于其他工作（如执行其他程序），直到I/O设备通知它“已准备好”。

中断驱动I/O包括两个部分：

（1）中断使能机制：在有输入数据要传递或准备好接收输出数据的情况下，I/O设备具备向处理器发出中断的能力。

（2）I/O数据传输的管理能力（即中断处理程序）。

如8.5节所介绍的，所谓“中断处理器的使能机制”，就是发出INT信号的能力。我们看到，通过READY和中断使能位的组合，是怎样输出中断请求信号的。我们也看到，只有在中断请求信号的优先级高于当前执行程序的优先级的情况下，INT信号才能有效。如图8-8所示，我们已经看到该机制的优点（处理器不再在轮询操作上浪费时间）。但在8.5节中，我们并未介绍I/O数据传输的管理程序，这是因为介绍它需要栈的背景知识。现在，我们已学习了栈的知识，可以继续介绍“中断驱动I/O”了。

如图8-6所示，I/O数据传输的管理程序包括三个阶段：

（1）中断服务程序的启动。

（2）中断服务程序的执行。

（3）中断服务程序的返回。

下面将依次阐述它们。

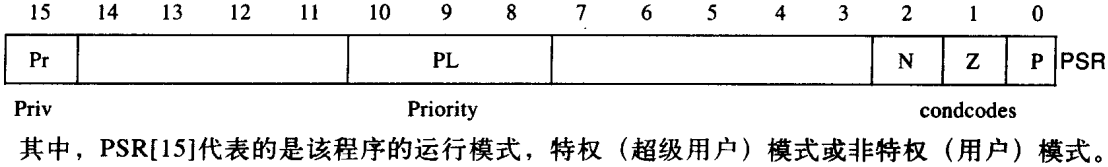
10.2.1 启动和执行

回顾8.5节及图8-8的介绍，当一个I/O设备的优先级高于当前执行程序时，它在发出INT信号时将启动中断。从处理器角度来看，它在每执行完一条指令（周期）之后，都将检查是否存在INT。如果没有，则继续下一条指令的执行；如果检测到INT，则暂停下一条指令的读取。

此时，开始中断执行的准备工作，以启动该I/O设备的中断服务程序。准备工作包括：（1）保存当前执行程序的状态，以使得中断处理程序返回时，能从当前的程序执行；（2）装载中断服务程序的工作状态，开始中断请求的服务。

1. 程序状态

所谓“程序状态”（state of a program），是指该程序运行所涉及资源的快照。它包括该程序的内存空间和通用寄存器的内容，以及两个重要的寄存器：PC和PSR寄存器。有关PC寄存器，相信你已经很熟悉了，它包含的是下一条待执行指令所在的内存地址。PSR寄存器，顾名思义是“处理器状态寄存器”，它包含了与运行程序相关的重要信息。下面是PSR的各字段信息：



其中，PSR[15]代表的是该程序的运行模式，特权（超级用户）模式或非特权（用户）模式。所谓“特权模式”，是指在该模式下，程序能够访问一般用户程序不可访问的资源。我们将看到该模式对中断处理的重要意义；PSR[10:8]代表正在执行程序的优先级别（priority level）。前面已经讲到，一共有8种特权级别，PL0（最低）~PL7（最高）。PSR[2:0]代表条件码，PSR[2] = N，

PSR[1] = Z，PSR[0] = P。

2. 被中断程序的状态保存

中断启动的第一个任务，就是保存正在运行程序的状态，以使得I/O设备请求的服务完成后，程序可以继续运行。换句话说，以LC-3为例，我们需要保存PC和PSR。保存PC，是因为通过它可以获知中断返回后的下一条待执行指令；保存条件码，是因为后续程序可能将依赖它完成条件跳转；保存被中断程序的优先级，是因为它提供被中断程序与其他程序相比的迫切程度，被中断程序恢复执行时，可能还有其他级别的程序试图再次中断它；最后要保存的是被中断程序的特权级别，这是因为它表明了被中断程序可以访问的资源范围。

一般情况下，我们认为通用寄存器的内容是没有必要保存的。因为我们可以假定中断服务程序在动用它们之前，会自觉保存它们的内容，并在返回被中断程序之前恢复原值。

LC-3将这些要保存的信息保存在一个特殊的栈空间中，我们称之为“超级用户栈”(supervisor stack)，该栈空间仅供特权模式下的程序使用。事实上，它们也是内存空间的一部分，只是它与用户程序的用户栈空间是相互隔离的。所有程序都通过R6（栈指针）访问栈空间，超级用户栈也不例外。只不过在超级用户模式下，R6指向超级用户栈空间；在用户模式下，R6指向用户栈空间。因此，在未使用情况下，内部寄存器Saved.SSP和Saved.USP分别用于保存两个栈指针的内容。例如，当特权模式从用户态切换到超级用户态时，原先的R6内容将被存入Saved.USP，然后将Saved.SSP的内容装入R6。

这意味着在中断服务程序开始之前，R6已装入超级用户栈指针的内容。而PC寄存器和被中断程序的PSR寄存器内容将被压入超级用户栈（而不是用户栈）。

3. 中断服务程序的状态装入

在被中断程序的状态信息已被安全存入超级用户栈之后，下一步的任务则是装入中断服务程序的PC和PSR内容。中断服务程序与第9章里介绍的TRAP程序很相似。它们的代码都事先存在于特定的内存地址段中。它们服务的都是中断请求。

大多数处理器采用“矢量中断”(vectored interrupt）的工作方式。在TRAP指令的学习中，你应该已熟悉TRAP矢量的概念。在中断情况下，I/O设备在发出中断时，将向处理器传递一个8-bit的矢量值（以及中断请求信号和设备优先级）。在多个设备同时请求中断处理的情况下，优先级最高的请求被“选中”并传递给处理器。我们称之为INTV。如果该中断被处理器接收，它将该8-bit的中断矢量（INTV）扩展为一个16-bit的地址，即中断矢量表中的某个表项地址。回顾第9章的TRAP矢量表，其内存占用地址是x0000~x00FF，其中每个表项的内容是一个TRAP服务程序的起始地址。类似地，中断矢量表的内存位置是x0100~x01FF，每个表项包含的是一个中断服务程序的起始地址。处理器将扩展后中断矢量INTV地址的内容（即矢量表项的内容）装入PC。

PSR寄存器的装入过程如下：由于在中断服务程序中，还没有指令曾执行过，所以PSR[2:0]的内容初始化为0；而中断服务程序是在特权模式下运行的，所以PSR[15]被设置为0；PSR[10:8]则被设置为中断请求者（即设备）的优先级别。

至此，完成中断服务的启动阶段。之后，可以运行中断服务程序了。

4. 中断服务

此时，PC包含的是中断服务程序的起始地址，所以下一个开始执行的就是中断服务程序了，即I/O设备的请求开始被服务了。

例如，有人在LC-3的键盘上按下某个键，它将“中断”处理器。随后，由键盘中断矢量所指示的处理程序（又称“句柄”(handler)）将被激活（invoke)。处理程序的任务是，将键盘数据寄存器的内容拷贝到内存的某个位置。

10.2.2 中断返回

中断服务程序的最后，是中断返回指令RTI。当处理器遇到RTI指令时，意味着I/O设备的请求已经完成。

RTI指令（操作码为1000）的任务是，将PSR和PC的内容弹出超级用户栈，然后将它们填入处理器中的正确位置。现在，条件码的值已恢复为程序被中断前的内容，如果需要，后续的BR指令即可正确执行了。同样，PSR[15]和PSR[10:8]的内容也被恢复（特权级别和优先级别）；PC也已恢复，指向中断前的下一条待执行指令。

所有的事情看起来都跟中断发生前没有两样，对程序来说，仿佛什么也没发生过一样。

10.2.3 例子：嵌套中断

最后，我们通过一个例子，结束有关中断驱动I/O的讨论。

假设程序*A*正在执行。这时，I/O设备*B*突然请求服务，且*B*的优先级高于*A*，中断开始服务。而在I/O设备*B*的中断服务程序正在执行时，设备*C*又产生中断。

图10-6所示是所发生情况的执行流程。

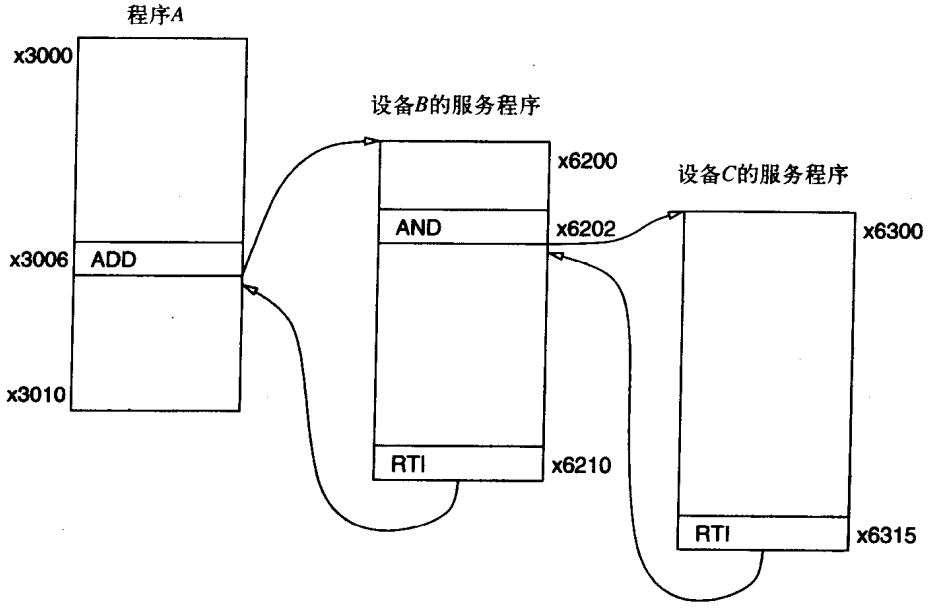


图10-6 中断驱动I/O的执行流程图

其中，程序*A*位于内存地址x3000~x3010，在执行x3006的ADD指令时，设备*B*发出了中断请求信号，导致矢量xF1的INT产生。

设备*B*的中断服务程序位于x6200~x6210。x6210处是RTI指令。当*B*的中断服务程序执行到x6202的AND指令时，设备*C*又发出中断请求（对应的中断矢量是xF2）。由于设备*C*的优先级高于设备*B*，所以INT再次发生。

设备*C*的中断服务程序位于x6300~x6315。x6315处是RTI指令。

让我们看一下处理器的执行过程。图10-7所示是该例子在执行过程中，超级用户栈和PC的内容快照。

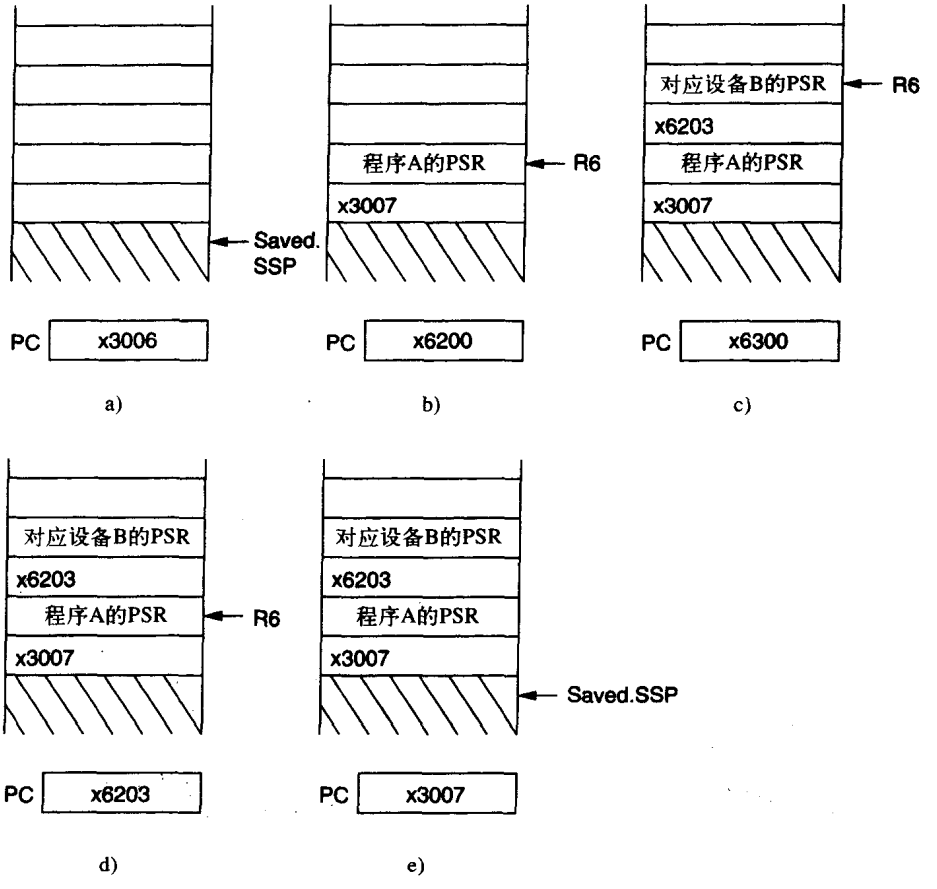


图10-7 中断驱动I/O执行过程中的超级用户栈和PC的内容快照

其中，处理器的执行顺序如下所示：

如图10-7a所示，是程序*A*在读取x3006的指令前的超级用户栈和PC状态值。注意，图中标识的栈指针是Saved.SSP（而不是R6）。由于此时中断尚未发生，所以R6所指向的仍然是用户栈。在x3006指令执行的最后，检测到INT信号（设备*B*的中断）。于是，程序*A*的状态将被保存在超级用户栈。但前提是能开始使用超级用户栈，所以先将当前R6的内容保存入Saved.USP寄存器，然后将Saved.SSP的内容装入R6；随后，PC的内容（程序*A*的下一条指令地址）x3007被压入超级用户栈。同样，ADD指令产生的条件码和PSR信息也被压入超级用户栈；再随后，设备*B*的中断矢量值被扩展为16-bit的x01F1，x01F1的内容（x6200）则被装入PC。图10-7b所示是此时的超级用户栈和PC寄存器状态。

设备*B*的中断服务程序开始执行。在x6202指令执行周期结束时，又检测到一个更高优先级的中断。随后，地址x6203被压入超级用户栈中，同时压入的还有*B*程序的PSR内容（包含AND指令产生的条件码信息）；设备*C*的中断矢量被扩展为16-bit的x01F2，x01F2的内容（x6300）被装入PC。图10-7c所示是此时的超级用户栈和PC寄存器状态。

程序*C*在执行x6315的RTI指令之后，超级用户栈被两次弹出。一是恢复*B*程序的PSR内容（其中包含x6202的AND指令所产生的条件码），二是恢复PC寄存器的内容（x6203）。图10-7*d*所示是此时的超级用户栈和PC寄存器状态。

*B*程序从x6203开始恢复执行，直到x6210的RTI指令结束。超级用户栈再次被弹出两次，恢复的是程序*A*的PSR寄存器内容（包含x3006处的ADD指令所产生的条件码信息），以及PC寄存器的

内容（x3007）。

之后，由于程序*A*运行在用户模式下，所以R6的内容被存入Saved.SSP，更换装入的是Saved.USP的内容。图10-7e所示是此时的超级用户栈和PC寄存器状态。

最后，程序*A*从x3007的指令开始，继续执行。

10.3 基于栈的算术运算

10.3.1 栈的临时存储作用

有些计算机使用栈（而不是通用寄存器）来存放计算过程的中间值。回顾我们的ADD指令：

```
ADD     R0,R1,R2
```

其中，R1和R2是源操作数，相加的结果存入R0。之所以将LC-3称做“3地址机器”（three-address machine），是因为它的三个操作单元（2个源操作数和1个目的操作数）都是显式表示的。而在采用栈方式的计算机中，源和目的操作数的位置都**没有**被显式指明。例如：

```
ADD
```

我们称这类机器为“栈式机”（stack machine）或“零地址机器”（zero-address machine）。硬件默认时将栈顶部的两个元素当做源操作数，并将它们弹出，送给ALU，之后的相加结果又压回栈顶。

在栈式机上的加法操作中，硬件将负责两次弹出、一次加法和一次压入。其中，两次弹出操作将把两个源数据移出栈，加法负责它们的求和，压入则将结果放回栈。注意，弹出、压入和加法都不属于指令集结构的范畴，因此它们不可被程序员所用。它们只是供给硬件来完成具体弹出、压入和加法运算的控制信号。而控制信号属于“微结构”（microarchitecture）的内容，与在第4、5章讨论的使能信号、开关选择信号是同一类事物。LC-3的LD和ST指令与控制信号PCMUX和LD.MDR之间的关系也如此。在栈式机中，程序员只需告诉计算机“执行ADD”，而微结构将负责所有的具体操作。

在有些情况下（如本章最后的例子），使用栈结构来完成算术运算非常方便（即将中间结果存在栈中，而不是存在LC-3的R0~R7寄存器中）。大多数微处理器（包括LC-3）都采用通用寄存器方式，而大多数计算器则采用栈方式。

10.3.2 例子：算术表达式

试计算表达式（*A*+*B*）·（*C*+*D*）的结果，其中*A* = 25、*B* = 7、*C* = 3、*D* = 2。计算结果存入*E*。如果LC-3具备乘法指令（如MUL），则计算过程如下面代码所示：

```
LD    R0,A
LD    R1,B
ADD   R0,R0,R1
LD    R2,C
LD    R3,D
ADD   R2,R2,R3
MUL   R0,R0,R2
ST    R0,E
```

而如果是计算器，则执行如下8步操作即可：

```
(1)    push     25
(2)    push     17
(3)    add
(4)    push     3
(5)    push     2
(6)    add
(7)    multiply
(8)    pop      E
```

最后弹出的是计算结果，即210。图10-8所示是8步操作中每一步操作之后的栈快照。



图10-8 计算(25+17)·(3+2)的程序中栈的使用情况

在10.5节中，我们将编写一个LC-3程序，使其表现得像一个计算器（借助键盘和显示器）。我们说这个LC-3程序“模拟”了一个计算器。

现在，我们先看看在计算器模拟中，将涉及的各种运算操作的子程序。

10.3.3 加、乘和取反

在10.5节的计算器模拟器中，我们可以输入数值，实现加、减、乘等操作，并可显示计算结果。为执行加、减、乘运算，我们需要3个子程序：

（1）加：将栈顶弹出的两个数值相加，然后将结果压回栈。

（2）乘：将栈顶弹出的两个数值相乘，然后将结果压回栈。

（3）取反操作：将栈顶元素弹出，对其取反（补码方式），然后将结果压回栈。

1. 加法运算

图10-9所示是加法运算的流程图。简单地说，先试图从栈顶弹出两个数值，如果成功则相加。如果计算结果在可接受值范围（−999~+999），则将结果压入栈。

有两种情况将造成运算失败：一是栈内元素数目小于2，二是计算结果超出表达范围。那么，在这两种情况下，总是将栈恢复到运算前状态，并在R5中记录数值1（代表执行失败），然后返回调用程序。如果第1个元素的弹出就失败，栈没有变化（因为POP程序没有动栈）；如果是第2个元素弹出失败，那么栈指针回递到前一个位置（即等价于将刚才弹出的值压回栈）；如果是最后的计算结果超出范围，则栈指针回减两个位置（即将两个源操作数都压回栈）。

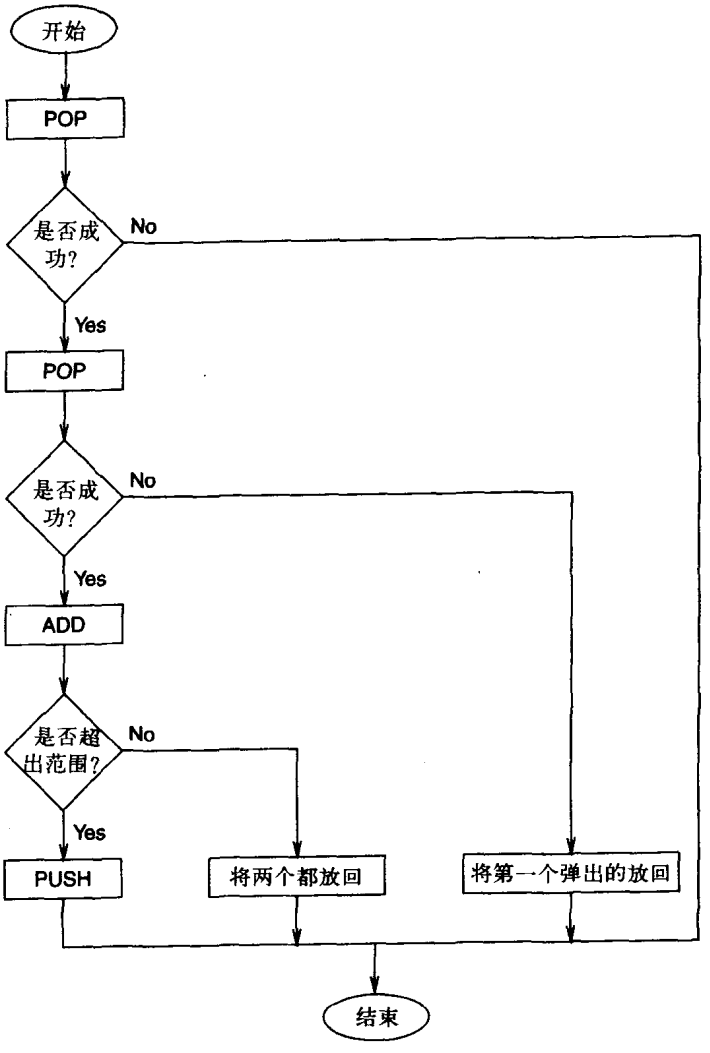


图10-9 加法运算的流程图

图10-10所示是加法运算的程序。

```
01    ;
02    ;        Routine to pop the top two elements from the stack,
03    ;        add them, and push the sum onto the stack.  R6 is
04    ;        the stack pointer.
05    ;
06    OpAdd       JSR     POP             ; Get first source operand.
07                ADD     R5,R5,#0        ; Test if POP was successful.
08                BRp     Exit            ; Branch if not successful.
09                ADD     R1,R0,#0        ; Make room for second operand.
0A                JSR     POP             ; Get second source operand.
0B                ADD     R5,R5,#0        ; Test if POP was successful.
0C                BRp     Restore1        ; Not successful, put back first.
0D                ADD     R0,R0,R1        ; THE Add.
0E                JSR     RangeCheck      ; Check size of result.
0F                BRp     Restore2        ; Out of range, restore both.
10                JSR     PUSH            ; Push sum on the stack.
11                RET                     ; On to the next task...
12    Restore2    ADD     R6,R6,#-1       ; Decrement stack pointer.
13    Restore1    ADD     R6,R6,#-1       ; Decrement stack pointer.
14    Exit        RET
```

图10-10 加法运算

其中，加法运算调用了数值范围检查（RangeCheck）函数，以保证该结果在栈空间中只占用一个单元大小。我们将数值的范围定为−999~+999。该程序已完成，在10.5节的计算器中可直接调用了。图10-11所示是数值范围检查的算法，其对应的LC-3程序如图10-12所示。

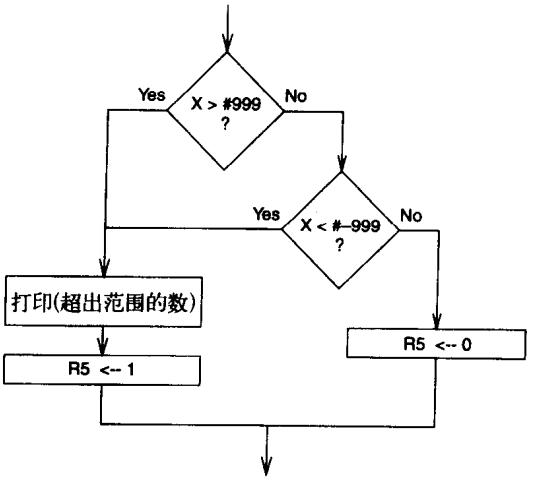


图10-11 范围检查函数的算法流程图

```
01   ;
02   ;     Routine to check that the magnitude of a value is
03   ;     between -999 and +999.
04   ;
05   RangeCheck    LD        R5,Neg999
06                 ADD       R4,R0,R5    ; Recall that R0 contains the
07                 BRp       BadRange    ; result being checked.
08                 LD        R5,Pos999
09                 ADD       R4,R0,R5
0A                 BRn       BadRange
0B                 AND       R5,R5,#0    ; R5 <-- success
0C                 RET
0D   BadRange      ST        R7,Save     ; R7 is needed by TRAP/RET.
0E                 LEA       R0,RangeErrorMsg
0F                 TRAP      x22         ; Output character string
10                 LD        R7,Save
11                 AND       R5,R5,#0    ;
12                 ADD       R5,R5,#1    ; R5 <-- failure
13                 RET
14   Neg999        .FILL     #-999
15   Pos999        .FILL     #999
16   Save          .FILL     x0000
17   RangeErrorMsg .FILL     x000A
18                 .STRINGZ  "Error: Number is out of range."
```

图10-12 范围检查程序

2. *乘法运算*

图10-13所示是乘法运算的流程图，对应的LC-3程序如图10-14所示。同加法运算一样，乘法运算试图从栈里弹出两个值，如果成功则相乘。但是，由于LC-3不具备乘法指令，所以我们需采用以前的做法，即通过一连串的叠加来实现乘法。如图10-14所示，行17~19是实现乘法的代码。最后，如果计算结果在有效范围内，则将它压入栈。

其中，如果两个弹出栈操作中任一个失败，则通过栈指针递减，将已弹出的数值压回栈中。如果计算结果超出可接受范围，则如之前一样，对R5赋值1（表示执行失败），并将栈指针递减两次，以将两个源操作数都压回栈。

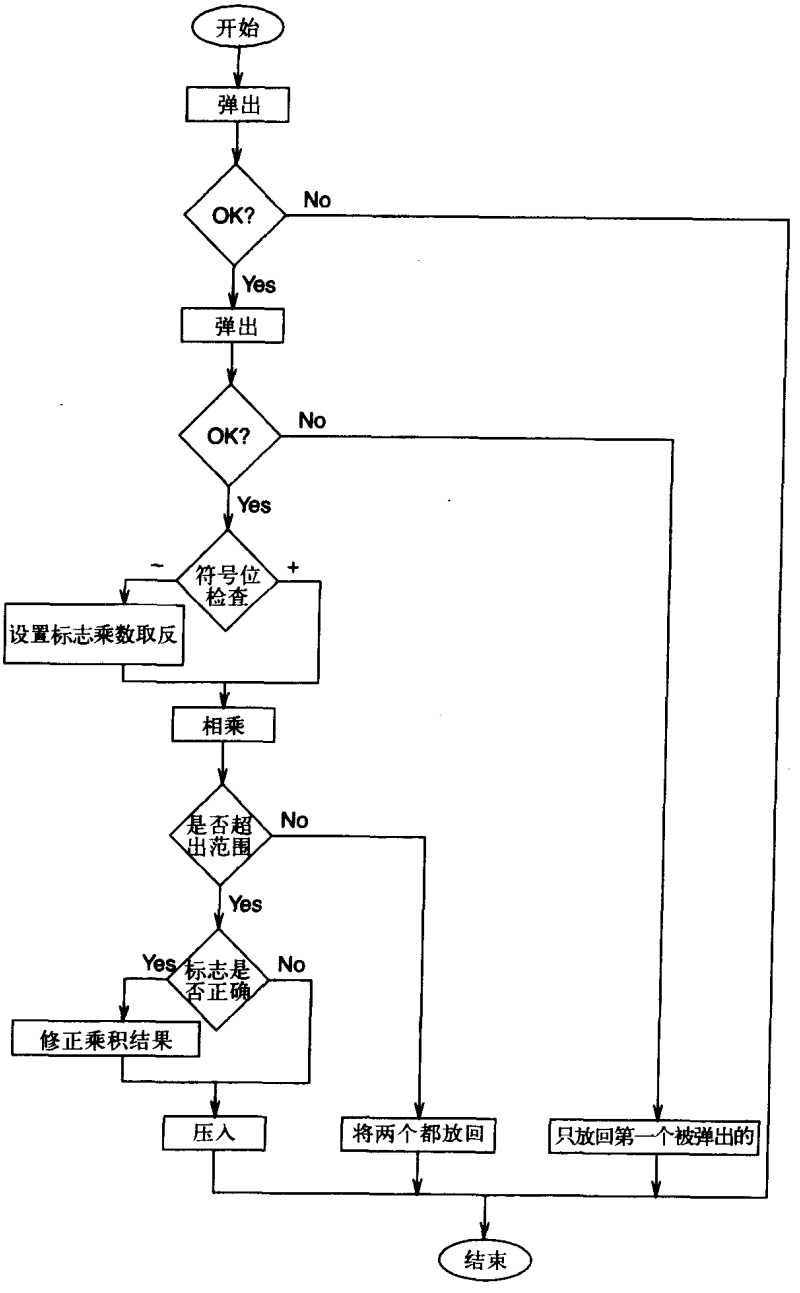


图10-13 乘法运算的流程图

3. 取反运算

之前的加法和乘法操作，都是对栈顶的两个元素进行运算。如果对栈顶的两个元素做减法操作，则只需在运算式中，将栈顶元素替换为它的负数形式，然后再执行加法即可。换句话说，假设栈顶元素为A，栈的第二个元素为B。我们希望弹出的是A和B，压入的是$B-A$。那么，我们可以先对栈顶元素A取反，然后再做加法运算。

图10-15所示是栈顶元素的取反算法（OpNeg）。

```
01  ;
02  ;    Algorithm to pop two values from the stack, multiply them,
03  ;    and if their product is within the acceptable range, push
04  ;    the result onto the stack.  R6 is stack pointer.
05  ;
06  OpMult          AND    R3,R3,#0       ; R3 holds sign of multiplier.
07                  JSR    POP            ; Get first source from stack.
08                  ADD    R5,R5,#0       ; Test for successful POP.
09                  BRp    Exit           ; Failure
0A                  ADD    R1,R0,#0       ; Make room for next POP.
0B                  JSR    POP            ; Get second source operand.
0C                  ADD    R5,R5,#0       ; Test for successful POP.
0D                  BRp    Restore1       ; Failure; restore first POP.
0E                  ADD    R2,R0,#0       ; Moves multiplier, tests sign.
0F                  BRzp   PosMultiplier
10                  ADD    R3,R3,#1       ; Sets FLAG: Multiplier is neg.
11                  NOT    R2,R2
12                  ADD    R2,R2,#1       ; R2 contains -(multiplier).
13  PosMultiplier   AND    R0,R0,#0       ; Clear product register.
14                  ADD    R2,R2,#0
15                  BRz    PushMult       ; Multiplier = 0, Done.
16  ;
17  MultLoop        ADD    R0,R0,R1       ; THE actual "multiply"
18                  ADD    R2,R2,#-1      ; Iteration Control
19                  BRp    MultLoop
1A  ;
1B                  JSR    RangeCheck
1C                  ADD    R5,R5,#0       ; R5 contains success/failure.
1D                  BRp    Restore2
1E  ;
1F                  ADD    R3,R3,#0       ; Test for negative multiplier.
20                  BRz    PushMult
21                  NOT    R0,R0          ; Adjust for
22                  ADD    R0,R0,#1       ; sign of result.
23  PushMult        JSR    PUSH           ; Push product on the stack.
24                  RET
25  Restore2        ADD    R6,R6,#-1      ; Adjust stack pointer.
26  Restore1        ADD    R6,R6,#-1      ; Adjust stack pointer.
27  Exit            RET
```

图10-14 乘法运算的程序

```
01   ; Algorithm to pop the top of the stack, form its negative,
02   ; and push the result onto the stack.
03   ;
04   OpNeg   JSR    POP        ; Get the source operand.
05           ADD    R5,R5,#0   ; Test for successful pop
06           BRp    Exit       ; Branch if failure.
07           NOT    R0,R0
08           ADD    R0,R0,#1   ; Form the negative of source.
09           JSR    PUSH       ; Push result onto the stack.
0A   Exit    RET
```

图10-15 取反运算

10.4 数据类型转换

有关数据类型的讨论，是很久以前的事情了！我们曾接触过多种数据类型，包括地址运算中的无符号整数，整数运算中的“2的补码”，逻辑操作中的16-bit二进制位数，科学计算中的浮点数，以及输入输出设备数据传输中的ASCII码。

对于各种指令，确保其源操作数的数据类型的正确性是非常重要的。例如，ADD指令所需要的操作数是补码形式的整数。而如果给ALU提供浮点型的操作数，则计算结果必然是无意义的。

在高级语言中，也不难找到形式如$A = R + I$的指令，其中R代表浮点数，I代表补码整数。

如果按照浮点加法来执行，则I是个问题。为解决这个问题，我们需要先将I从原数据类型（2

的补码）转换为操作所要求的类型（浮点数）。

在LC-3中，也有这样的数据类型转换问题。假设从键盘读入多个整数，输入的是ASCII字符串。但要对它做算术运算，则必须先将它转换为补码整数。而要将一个补码形式的数值值显示在屏幕上，则又需要将它转换为ASCII字符串。

在本节中，我们将讨论十进制数的ASCII字符串和它的补码二进制数之间的转换程序。

10.4.1 一个错误结果的例子：2 + 3 = e

图10-16所示是数据类型转换的例子。但在编程时，如果不注意数值的数据类型，则会产生如下的一些问题。

```
01   TRAP   x23          ; Input from the keyboard.
02   ADD    R1,R0,#0     ; Make room for another input.
03   TRAP   x23          ; Input another character.
04   ADD    R0,R1,R0     ; Add the two inputs.
05   TRAP   x21          ; Display result on the monitor.
06   TRAP   x25          ; Halt.
```

图10-16 不注意数据类型的加法

其中，假设我们希望从键盘读入两个数，然后相加，并将结果显示在屏幕上。刚开始，我们可能会写出如图10-16所示的代码。如果执行这个程序，结果会如何呢？

假设第一个输入是2，第二位输入是3。试问，程序结束前，屏幕上显示的是什么？输入2后，R0装入的是2的ASCII码（即x0032）；输入3后，R0的内容是3的ASCII码（即x0033）。然后，加法指令运算的是x0032与x0033之和（即结果为x0065）。最后，将该结果显示在屏幕上，由于ASCII码x0065代表小写字母“e”，所以屏幕上显示出“e”。

为什么不是5即2+3的结果呢？原因是：(1) 在计算之前，没有将两个输入字符从ASCII码转换为补码整数；(2) 在输出到屏幕之前，没有将结果转换为ASCII码。

思考题：修改如图10-16所示的程序，使它能正确执行两个个位正整数的相加，并输出一个个位正整数和。假设两数相加的结果也是个位数（没有进位）。

10.4.2 ASCII/二进制转换

我们必须解决多位数的表示问题。图10-17所示是采用ASCII码表示三位数“295”的方式。其中，我们按ASCII字符串方式，将三个数字存放在LC-3内存中标识ASCIIBUFF起始的三个连续地址中。R1记录的是ASCIIBUF起始内存中的有效位数。

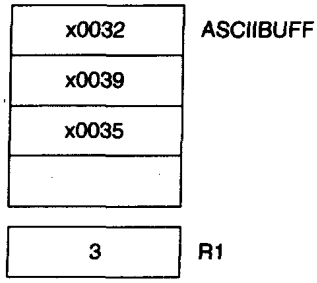


图10-17 以ASCII码表示形式储存在连续内存地址中的数字295

其中，每个ASCII字符都独自占用了一个LC-3字（16-bit）的空间。当然，我们可以（也应该）让每个ASCII字符占用一个字节（8-bit）的空间。但在此，为简化算法，我们将整个字都分配给了一个ASCII字符。

图10-18所示是将ASCII码（如图10-17所示）转化为二进制整数的算法流程图。其中，ASCII码表示数值的范围必须是0~+999之间，即最大是3位数。

在该算法中，我们依次处理每位数字。对于每位数字，只取其二进制值的最低4位。然后，以该值索引对应的数值查找表（不同的数位有不同的表，如LookUp10、Lookup100），每个表项的内容是10个数字对应的数值。然后，将每位数值累加入R0，即得最后结果。

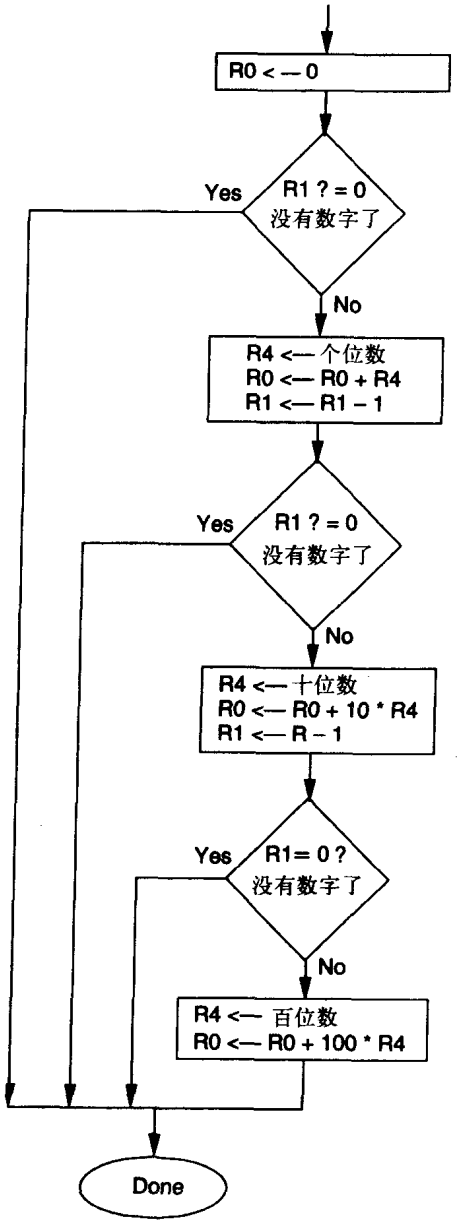


图10-18 将ASCII码转化为二进制的算法流程图

图10-19所示是该算法对应的LC-3程序。

```
01   ;
02   ;  This algorithm takes an ASCII string of three decimal digits and
03   ;  converts it into a binary number.  R0 is used to collect the result.
04   ;  R1 keeps track of how many digits are left to process.  ASCIIBUFF
05   ;  contains the most significant digit in the ASCII string.
06   ;
07   ASCIItoBinary  AND    R0,R0,#0       ; R0 will be used for our result.
08                  ADD    R1,R1,#0       ; Test number of digits.
09                  BRz    DoneAtoB       ; There are no digits.
0A   ;
0B                  LD     R3,NegASCIIOffset  ; R3 gets xFFD0, i.e., -x0030.
0C                  LEA    R2,ASCIIBUFF
```

图10-19 从ASCII码到二进制的转换程序

```
0D                  ADD     R2,R2,R1
0E                  ADD     R2,R2,#-1       ; R2 now points to "ones" digit.
0F  ;
10                  LDR     R4,R2,#0        ; R4 <-- "ones" digit
11                  ADD     R4,R4,R3        ; Strip off the ASCII template.
12                  ADD     R0,R0,R4        ; Add ones contribution.
13  ;
14                  ADD     R1,R1,#-1
15                  BRz     DoneAtoB        ; The original number had one digit.
16                  ADD     R2,R2,#-1       ; R2  now points to "tens" digit.
17  ;
18                  LDR     R4,R2,#0        ; R4 <-- "tens" digit
19                  ADD     R4,R4,R3        ; Strip off ASCII template.
1A                  LEA     R5,LookUp10     ; LookUp10 is BASE of tens values.
1B                  ADD     R5,R5,R4        ; R5 points to the right tens value.
1C                  LDR     R4,R5,#0
1D                  ADD     R0,R0,R4        ; Add tens contribution to total.
1E  ;
1F                  ADD     R1,R1,#-1
20                  BRz     DoneAtoB        ; The original number had two digits.
21                  ADD     R2,R2,#-1       ; R2 now points to "hundreds" digit.
22  ;
23                  LDR     R4,R2,#0        ; R4 <-- "hundreds" digit
24                  ADD     R4,R4,R3        ; Strip off ASCII template.
25                  LEA     R5,LookUp100    ; LookUp100 is hundreds BASE.
26                  ADD     R5,R5,R4        ; R5 points to hundreds value.
27                  LDR     R4,R5,#0
28                  ADD     R0,R0,R4        ; Add hundreds contribution to total.
29  ;
2A  DoneAtoB        RET
2B  NegASCIIOffset  .FILL   xFFD0
2C  ASCIIBUFF       .BLKW   4
2D  LookUp10        .FILL   #0
2E                  .FILL   #10
2F                  .FILL   #20
30                  .FILL   #30
31                  .FILL   #40
32                  .FILL   #50
33                  .FILL   #60
34                  .FILL   #70
35                  .FILL   #80
36                  .FILL   #90
37  ;
38  LookUp100       .FILL   #0
39                  .FILL   #100
3A                  .FILL   #200
3B                  .FILL   #300
3C                  .FILL   #400
3D                  .FILL   #500
3E                  .FILL   #600
3F                  .FILL   #700
40                  .FILL   #800
41                  .FILL   #900
```

图10-19 从ASCII码到二进制的转换程序（续）

思考题（一个非常有挑战的问题）：假设该十进制数的长度是任意的。那么，我们不仅要有十位和百位的数值查找表，还要为千位、万位等数字各准备一个10个表项的查找表。此时内存空间的占用太大了。请设计一个算法，能否不需要查找表就可以完成转换。参见习题10.20。

10.4.3 二进制/ASCII转换

相反，将补码整数转换为ASCII字符也非常必要，例如将结果显示在屏幕上。图10-20所示是该转换的算法。其中，R0存放的是待转换的补码整数，转换后的ASCII字符串存放在ASCIIBUFF起始的4个连续内存位置。R0中数值的范围是−999~+999。地址ASCIIBUFF包含的是该数值的符号

信息，随后的三个地址分别对应三位数字。

```
01  ;
02  ;  This algorithm takes the 2's complement representation of a signed
03  ;  integer within the range -999 to +999 and converts it into an ASCII
04  ;  string consisting of a sign digit, followed by three decimal digits.
05  ;  R0 contains the initial value being converted.
06  ;
07  BinarytoASCII  LEA   R1,ASCIIBUFF  ; R1 points to string being generated.
08                 ADD   R0,R0,#0      ; R0 contains the binary value.
09                 BRn   NegSign       ;
0A                 LD    R2,ASCIIplus  ; First store the ASCII plus sign.
0B                 STR   R2,R1,#0
0C                 BRnzp Begin100
0D  NegSign        LD    R2,ASCIIminus ; First store ASCII minus sign.
0E                 STR   R2,R1,#0
0F                 NOT   R0,R0         ; Convert the number to absolute
10                 ADD   R0,R0,#1      ; value; it is easier to work with.
11  ;
12  Begin100       LD    R2,ASCIIoffset ; Prepare for "hundreds" digit.
13  ;
14                 LD    R3,Neg100     ; Determine the hundreds digit.
15  Loop100        ADD   R0,R0,R3
16                 BRn   End100
17                 ADD   R2,R2,#1
18                 BRnzp Loop100
19  ;
1A  End100         STR    R2,R1,#1   ; Store ASCII code for hundreds digit.
1B                 LD     R3,Pos100
1C                 ADD    R0,R0,R3   ; Correct R0 for one-too-many subtracts.
1D  ;
1E                 LD     R2,ASCIIoffset ; Prepare for "tens" digit.
1F  ;
20  Begin10        LD     R3,Neg10   ; Determine the tens digit.
21  Loop10         ADD    R0,R0,R3
22                 BRn    End10
23                 ADD    R2,R2,#1
24                 BRnzp  Loop10
25  ;
26  End10          STR    R2,R1,#2   ; Store ASCII code for tens digit.
27                 ADD    R0,R0,#10  ; Correct R0 for one-too-many subtracts.
29  Begin1         LD     R2,ASCIIoffset ; Prepare for "ones" digit.
2A                 ADD    R2,R2,R0
2B                 STR    R2,R1,#3
2C                 RET
2D  ;
2E  ASCIIplus      .FILL  x002B
2F  ASCIIminus     .FILL  x002D
30  ASCIIoffset    .FILL  x0030
31  Neg100         .FILL  xFF9C
32  Pos100         .FILL  x0064
33  Neg10          .FILL  xFFF6
```

图10-20 从二进制到ASCII码的转换程序

该算法的工作原理如下所述。首先，判断该数的符号，并表示为对应的ASCII码；然后，将R0的值替换为其绝对值；之后，将R0内容反复减100，直到为负值（判断百位上的数字）。依此类推，测试10位上的数字。最后剩下的就是个位数字。

思考题：在该算法中，无论是什么整数，转换结果都是一个4字符长的字符串。能否设计一个算法，去除不必要的字符（如前缀的“0”或“+”）。参见习题10.22。

10.5 模拟计算器

本章的最后，是一个较复杂的例子：模拟计算器。之所以选这个例子，是因为其中涉及到很多已学过的知识。同时，我们在这个例子中，也将展示一个完善的文档和清晰的编码示例。相比

之前的那些1~2页的简单例子，该设计要复杂得多。在这个计算器模拟程序中，有11个相互独立的程序。我们希望在学习第11章的高级编程语言之前，攻克这个例子。

计算器的工作方式是：通过键盘输入十进制数和操作命令，然后将结果输出在显示器上。算术运算采用的是10.2节中介绍的栈机制。输入和输出数值都限定为3位数，即−999~+999（包括−999和+999）。在这个计数器中，允许的操作命令包括：

X 退出模拟器。
D 显示栈顶元素。
C 清空栈内所有元素。
+ 将栈顶的两个元素求和，并替换栈顶。
* 将栈顶的两个元素求积，并替换栈顶。
− 对栈顶元素取反。
Enter 将键盘输入值压入栈。

图10-21所示是模拟计算器的概要流程图。计算器模拟程序的开始是初始化工作，其中包括设置栈指针R6，清空栈，然后输出信息提示用户输入。

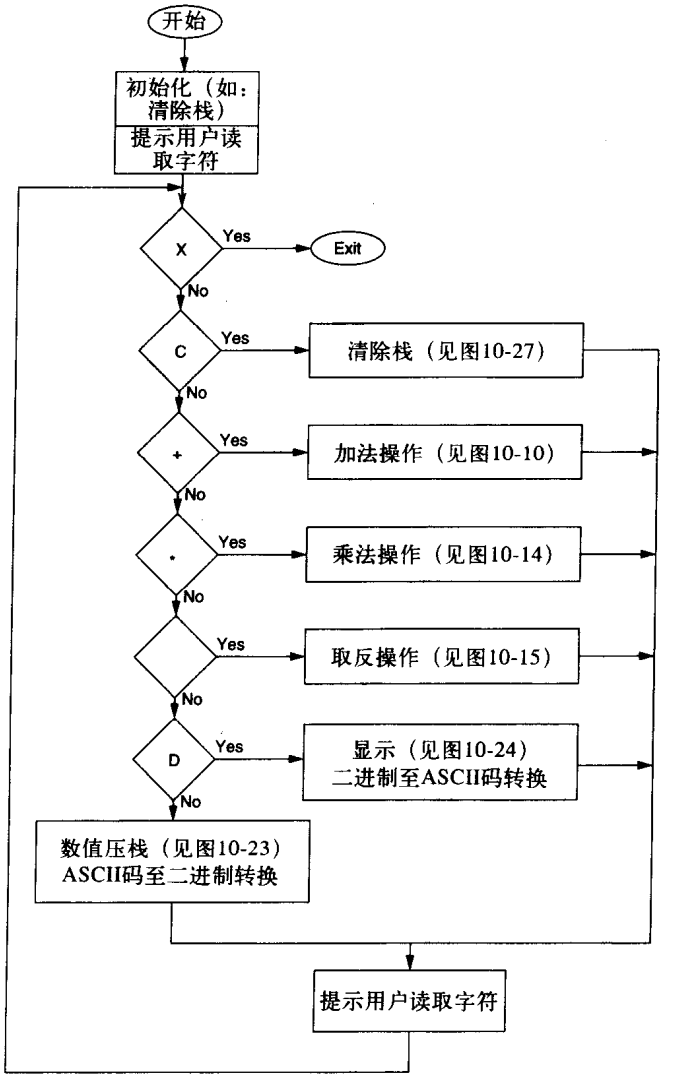


图10-21 计算器的概貌

另外，用户的输入将被回显（echo）。同时，模拟器将分析用户的每个输入。计算器模拟器将根据用户的命令执行相应的操作，然后提示用户输入下一个命令。模拟器不断如此重复，直到用户输入“X”，表示终止模拟器。

整个计算器模拟器由11个程序组成。图10-22所示是主程序；图10-23所示程序的任务是将用户输入的ASCII字符（数字）转换为二进制数，然后压入栈；图10-19所示是ASCII码至二进制的转换程序；如图10-26所示的程序，从栈顶弹出一个元素，并将其转换为ASCII字符串，输出至显示器；图10-20所示是二进制数至ASCII码的转换程序；如图10-10（OpAdd）、图10-14（OpMult）和图10-15（OpNeg）所示的程序，则是基于栈机制实现的各种基本运算方法；如图10-24和图10-25所示的程序，分别是为本程序定制的POP和PUSH程序；最后是如图10-27所示的栈清空程序。

```
01   ;
02   ;  The Calculator, Main Algorithm
03   ;
04              LEA       R6,StackBase  ; Initialize the stack.
05              ADD       R6,R6,#1      ; R6 is stack pointer.
06              LEA       R0,PromptMsg
07              PUTS
08              GETC
09              OUT
0A   ;
0B   ; Check the command
0C   ;
0D   Test       LD        R1,NegX       ; Check for X.
0E              ADD       R1,R1,R0
0F              BRz       Exit
10   ;
11              LD        R1,NegC       ; Check for C.
12              ADD       R1,R1,R0
13              BRz       OpClear       ; See Figure 10.27.
14   ;
15              LD        R1,NegPlus    ; Check for +
16              ADD       R1,R1,R0
17              BRz       OpAdd         ; See Figure 10.10.
18   ;
19              LD        R1,NegMult    ; Check for *
1A              ADD       R1,R1,R0
1B              BRz       OpMult        ; See Figure 10.14.
1C   ;
1D              LD        R1,NegMinus   ; Check for -
1E              ADD       R1,R1,R0
1F              BRz       OpNeg         ; See Figure 10.15.
20   ;
21              LD        R1,NegD       ; Check for D
22              ADD       R1,R1,R0
23              BRz       OpDisplay     ; See Figure 10.24.
24   ;
25   ; Then we must be entering an integer
26   ;
27              BRnzp     PushValue     ; See Figure 10.23.
28   ;
29   NewCommand LEA       R0,PromptMsg
2A              PUTS
2B              GETC
2C              OUT
2D              BRnzp     Test
2E   Exit       HALT
2F   PromptMsg  .FILL     x000A
30              .STRINGZ "Enter a command:"
31   NegX       .FILL     xFFA8
32   NegC       .FILL     xFFBD
33   NegPlus    .FILL     xFFD5
34   NegMinus   .FILL     xFFD3
35   NegMult    .FILL     xFFD6
36   NegD       .FILL     xFFBC
```

图10-22 计算器的主程序

```
01   ; This algorithm takes a sequence of ASCII digits typed by the user,
02   ; converts it into a binary value by calling the ASCIItoBinary
03   ; subroutine, and pushes the binary value onto the stack.
04   ;
05   PushValue        LEA       R1,ASCIIBUFF   ; R1 points to string being
06                    LD        R2,MaxDigits   ; generated.
07   ;
08   ValueLoop        ADD       R3,R0,xFFF6    ; Test for carriage return.
09                    BRz       GoodInput
0A                    ADD       R2,R2,#0
0B                    BRz       TooLargeInput
0C                    ADD       R2,R2,#-1      ; Still room for more digits.
0D                    STR       R0,R1,#0       ; Store last character read.
0E                    ADD       R1,R1,#1
0F                    GETC
10                    OUT                      ; Echo it.
11                    BRnzp     ValueLoop
12   ;
13   GoodInput        LEA       R2,ASCIIBUFF
14                    NOT       R2,R2
15                    ADD       R2,R2,#1
16                    ADD       R1,R1,R2       ; R1 now contains no. of char.
17                    JSR       ASCIItoBinary
18                    JSR       PUSH
19                    BRnzp     NewCommand
1A   ;
1B   TooLargeInput    GETC                     ; Spin until carriage return.
1C                    OUT
1D                    ADD       R3,R0,xFFF6
1E                    BRnp      TooLargeInput
1F                    LEA       R0,TooManyDigits
20                    PUTS
21                    BRnzp     NewCommand
22   TooManyDigits    .FILL     x000A
23                    .STRINGZ "Too many digits"
24   MaxDigits        .FILL     x0003
```

图10-23 计算器的PushValue程序

其中，为使各程序能与如图10-22⊖所示的主程序配合工作，需要对部分代码做出修改。例如，OpAdd、OpMult和OpNeg等程序的结尾，都将更换为如下指令（而不是RET指令）：

```
BRnzp NewCommand
```

此外，有的标识符（label）在多个程序中都出现过。如果这些子程序都是单独汇编的，但这些标识符被标识为.EXTERNAL类型（见9.2.5节），则在多个子程序中使用同名标识符是不会有问题的。但是，如果这些程序最后被链接为一个模块，则标识符重名是不允许的。所以，在这种情况下，我们必须对其中的一些标识符重新命名（如Restore1、Restore2、Save等），以保证它们的惟一性。

```
01   ;  This algorithm POPs a value from the stack and puts it in
02   ;  R0 before returning to the calling program.  R5 is used to
03   ;  report success (R5 = 0) or failure (R5 = 1) of the POP operation.
04   POP              LEA     R0,StackBase
05                    NOT     R0,R0
06                    ADD     R0,R0,#1       ; R0 = -(addr.ofStackBase -1)
07                    ADD     R0,R0,R6       ; R6 = StackPointer
08                    BRz     Underflow
09                    LDR     R0,R6,#0       ; The actual POP
0A                    ADD     R6,R6,#1       ; Adjust StackPointer
0B                    AND     R5,R5,#0       ; R5 <-- success
0C                    RET
0D   Underflow        ST      R7,Save        ; TRAP/RET needs R7.
0E                    LEA     R0,UnderflowMsg
0F                    PUTS                   ; Print error message.
10                    LD      R7,Save        ; Restore R7.
11                    AND     R5,R5,#0
12                    ADD     R5,R5,#1       ; R5 <-- failure
13                    RET
```

图10-24 计算器的POP程序

⊖ 原书中此处为“Figure 10.17”。结合上下文，此处应该是“图10-22”。——译者注

```
14   Save            .FILL   x0000
15   StackMax        .BLKW   9
16   StackBase       .FILL   x0000
17   UnderflowMsg    .FILL   x000A
18                   .STRINGZ "Error: Too Few Values on the Stack."
```

图10-24 计算器的POP程序（续）

```
01   ;  This algorithm PUSHes on the stack the value stored in R0.
02   ;  R5 is used to report success (R5 = 0) or failure (R5 = 1) of
03   ;  the PUSH operation.
04   ;
05   PUSH            ST      R1,Save1        ; R1 is needed by this routine.
06                   LEA     R1,StackMax
07                   NOT     R1,R1
08                   ADD     R1,R1,#1        ; R1 = - addr. of StackMax
09                   ADD     R1,R1,R6        ; R6 = StackPointer
0A                   BRz     Overflow
0B                   ADD     R6,R6,#-1       ; Adjust StackPointer for PUSH.
0C                   STR     R0,R6,#0        ; The actual PUSH
0D                   BRnzp   Success_exit
0E   Overflow        ST      R7,Save
0F                   LEA     R0,OverflowMsg
10                   PUTS
11                   LD      R7,Save
12                   LD      R1, Save1       ; Restore R1.
13                   AND     R5,R5,#0
14                   ADD     R5,R5,#1        ; R5 <-- failure
15                   RET
16   Success_exit    LD      R1,Save1        ; Restore R1.
17                   AND     R5,R5,#0        ; R5 <-- success
18                   RET
19   Save            .FILL   x0000
1A   Save1           .FILL   x0000
1B   OverflowMsg     .STRINGZ "Error: Stack is Full."
```

图10-25 计算器的PUSH程序

```
01   ; This algorithm calls BinarytoASCII to convert the 2's complement
02   ; number on the top of the stack into an ASCII character string, and
03   ; then calls PUTS to display that number on the screen.
04   OpDisplay    JSR     POP             ; R0 gets the value to be displayed.
05                ADD     R5,R5,#0
06                BRp     NewCommand  ; POP failed, nothing on the stack.
07                JSR     BinarytoASCII
08                LD      R0,NewlineChar
09                OUT
0A                LEA     R0,ASCIIBUFF
0B                PUTS
0C                ADD     R6,R6,#-1   ; Push displayed number back on stack.
0D                BRnzp   NewCommand
0E   NewlineChar  .FILL   x000A
```

图10-26 计算器的显示程序

```
01   ;
02   ; This routine clears the stack by resetting the stack pointer (R6).
03   ;
04   OpClear      LEA     R6,StackBase  ; Initialize the stack.
05                ADD     R6,R6,#1      ; R6 is stack pointer.
06                BRnzp   NewCommand
```

图10-27 OpClear程序

10.6 习题

10.1 根据栈的定义，试讲述它的特征。

10.2 试比较如图10-3所示的实现模型，相比图10-2所示的模型，它有什么优点？

10.3 假设我们在LC-3指令集中增加了Push和Pop两条指令。其中，Push Rn的任务是将寄存器n的内容压入栈，Pop Rn的任务则是将栈顶内容移出栈并赋值给Rn。下图所示是LC-3的8个寄存器和6个栈操作，左图和右图分别是6个栈操作指令执行“之前”和“之后”的快照。请比较

两个快照内容的差别，填充指令中标记为（a）~（d）的内容。

	BEFORE			AFTER
R0	x0000	PUSH R4	R0	x1111
R1	x1111	PUSH (a)	R1	x1111
R2	x2222	POP (b)	R0	x3333
R3	x3333	PUSH (c)	R3	x3333
R4	x4444	POP R2	R4	x4444
R5	x5555	POP (d)	R5	x5555
R6	x6666		R6	x6666
R7	x7777		R7	x4444

10.4 试编写一个栈操作函数——peek。Peek的作用是只读取栈顶元素的内容，但并不将它弹出栈。此外，Peek还应具备栈的下溢出错检查。（为什么不做上溢出错检查呢？）

10.5 如果针对如图10-2所示的栈模型，应该怎么检查上溢出和下溢出错误？试重写PUSH和POP程序，实现如图10-2所示的栈模型（即每个栈操作都将引发所有数据的移动）。

10.6 试重写PUSH和POP程序，使栈中每个元素占用两个内存位置。

10.7 试重写PUSH和POP程序，使栈元素的大小可以是任意长度。

10.8 假设我们对栈依次执行以下操作，试问：

```
PUSH A, PUSH B, POP, PUSH  C, PUSH D, POP, PUSH E, POP, POP, PUSH F
```

a. PUSH F执行之后，栈的内容是什么？

b. 什么时候栈中包含的元素最多？假设从此刻开始，继续执行以下操作：

```
PUSH G,PUSH H,PUSH I,PUSH J,POP,PUSH K,POP,POP,POP,PUSH L,POP,POP,PUSH M
```

c. 那么，最后栈中的内容是什么？

10.9 我们称“栈的输入流”为这样一串元素，它们是被压入栈的所有元素，并按其压入栈的顺序排列。同样，我们将“栈的输出流”定义为被弹出栈的所有元素。

假设在习题10.8中，输入流是“ABCDEFGHIJKLM”。试问：

a. 习题10.8中的输出流是什么（提示：BDE…）？

b. 再假设输入流为“ZYXWVUTSR”，试编写一个由PUSH和POP组成的操作序列，使其输出流为“YXVUWZSRT”。

c. 如果输入流为“ZYXW”，请问有多少种不同的输出流形式？

10.10 在中断服务程序的启动阶段，条件位N、Z和P都是存放在栈中的。试给出例子，说明在不保存条件位的情况下，会发生什么错误。

10.11 试问，在10.2.3节所示的例子中，地址x6200和x6300的内容是什么？它们是某个更大的结构体（structure）的组成部分，请给出这个结构体的名字（提示：参考表A-3）。

10.12 假设我们对10.2.3节的例子做一个扩展，使得设备C的中断服务程序在执行x6310指令时，发生一个优先级更高的中断（设备D）。假设，设备D的陷入矢量是xF3，而其中断服务程序在地址x6400~x6412处。试给出程序执行过程中，各关键点的栈内容和PC值快照。

10.13 假设在习题10.12中，设备D的优先级低于设备C，但高于设备B。试在新的假设下，重新回答习题10.12的问题。

10.14 试编写一个满足如下需求的“键盘输入”中断服务程序：在内存空间x4000~x40FE分配一个缓存。在中断处理程序中，读入下一个输入字符，并将它存入下一个缓存空位。内存x40FF的内容是指向缓存中下一个可用空位的指针。如果缓存已满（即地址x40FE已存有字符），则在屏幕上打印消息：“Character cannot be accepted; input buffer full.”

10.15 参考如习题10.14所示的中断服务函数，并对缓存结构做如下修改：缓存空间调整为在内存地

址x4000~x40FC之间。内存x40FF仍然是指向缓存中下一个可用位置的地址。而x40FE包含的是起始字符的地址，x40FD包含的是缓存中字符的个数。其他程序可以从缓存中直接取走字符。试按如下要求修改中断处理函数：如果x4000的字符已被读走（即起始字符的地址后移），则在x40FC被占用之后，下一个可填充的地址就变为x4000。同样，缓存满了之后，中断服务程序则在屏幕上打印："Character cannot be accepted; input buffer full."

10.16 以习题10.15中修改后的中断服务函数，以及一个正在从缓存中取走字符的程序为场景。试问，在同一时刻，中断服务函数正在将字符写入缓存，同时一个程序正在读取缓存，它们能否同时工作，是否存在什么问题？

10.17 试用自己的语言，描述图10-14所示的乘法OpMult的工作原理和过程。该乘法操作总共执行了多少条指令？以乘数n为参数表述该答案。注意：如果同一条指令被执行5次，我们说它执行了5条指令。试编写一个程序，如果乘数值小于25，则它使用更少的指令数来完成该乘法过程。给出它的具体指令数。

10.18 试修改图10-16所示的程序，使得它可以对2个个位正整数相加，并计算出一个个位的正整数结果。其中，假设两个个位正整数相加的结果也是个位正整数。

10.19 试修改图10-16所示的程序，假设输入数字都是个位十六进制正整数，且这两个数相加的结果也是个位十六进制正整数。

10.20 图10-19所示是一个将ASCII字符串转换为二进制值的算法。如果十进制数为任意长度，采用千位数字、万位数字的数值查找表是不合适的。请设计一种更有效的转换算法。

10.21 图10-19所示是一个将ASCII字符串转换为二进制值的算法。试扩展该代码，使其可以将ASCII字符表示的十六进制数转换为二进制数。其中，如果一个数字的前缀为"x"，则认为后续ASCII字符（最多三个）是一个十六进制数；否则将其当做十进制数。

10.22 图10-20所示的算法，总是产生一个4字符的字符串（无论被转换数字的符号和大小如何）。试设计一种算法，它可以将表达式中不必要的字符去除。例如，前缀中的连续0和符号"+"。

10.23 阅读以下LC-3代码，试问它的作用是什么？

```
            .ORIG    X3000
            LEA      R6, STACKBASE
            LEA      R0, PROMPT
            TRAP     x22              ; PUTS
            AND      R1, R1, #0
LOOP        TRAP     x20              ; IN
            TRAP     x21
            ADD      R3, R0, #-10     ; Check for newline
            BRz      INPUTDONE
            JSR      PUSH
            ADD      R1, R1, #1
            BRnzp    LOOP
INPUTDONE   ADD      R1, R1, #0
            BRz      DONE
LOOP2       JSR      POP
            TRAP     x21
            ADD      R1, R1, #-1
            BRp      LOOP2
DONE        TRAP     x25              ; HALT

PUSH        ADD      R6, R6, #-2
            STR      R0, R6, #0
            RET

POP         LDR      R0, R6, #0
            ADD      R6, R6, #2
            RET
PROMPT      .STRINGZ ''Please enter a sentence: ''
STACKSPAC   .BLKW #50
STACKBASE   .FILL #0
            .END
```

10.24 回顾习题8.15的程序，回答以下问题：

```
            .ORIG       x3000
            LD          R3,A
            STI         R3, KBSR
            LD          R0, B
TEST        LDI         R1, DSR
            BRzp        TEST
            STI         R0, DDR
            BRnzp       TEST
A           .FILL       x4000
B           .FILL       x0032
KBSR        .FILL       xFE00
DSR         .FILL       xFE04
DDR         .FILL       xFE06
            .END
```

a. 假设键盘的中断矢量是x34，且键盘中断服务程序的起始地址为x1000，键盘中断服务的代码如下所示。试问该服务程序的操作是什么？

```
            .ORIG       x1000
            LDI         R0,KBDR
            TRAP        x21
            TRAP        x21
            TRAP        x25
KBDR        .FILL       xFE02
            .END
```

b. 假设问题a中的代码正在执行，这时有人又敲了一个按键。试问屏幕上的显示是什么？

第11章 C语言编程概述

11.1 我们的目标

恭喜，你已进入本书的第二部分！通过第一部分的学习，你已对现代计算机系统的底层知识有所了解。基于这个坚实的基础，我们将开始学习高级语言的基础编程。

在本书第二部分中，我们将以C语言为例子，讨论有关高级语言编程的基本概念。在整个循序渐进的学习过程中，每遇到一个高层的新概念，我们都将它与计算机系统的底层相关联。从某种意义上说，我们将破解所有的神秘。这也是为什么采用自底向上方法的理由，我们希望你知道计算机在运行程序（你自己编写的）时都做了些什么？我们的理念是，如果你知道“神秘”背后的活动，对编程的理解也必然会更进一个层次，即达到了一个“真正的”程序员所具有的境界。

我们先简单回顾一下本书第一部分的内容。在前10章中，我们描述了一个简单计算机“LC-3”，它是一个“虚拟机器”，虽然简单，但具备了真实计算机的所有重要特征。LC-3（事实上也是所有现代计算机）的设计理念是：任何复杂的设备都是由基本单元有机地互连而成的。例如，MOS管构成逻辑门，逻辑门构成内存及数据通路（data-path）中的其他部件，而内存和数据通路又构成了整个LC-3。这个理念是计算机系统中的一个重要概念，它不仅存在于硬件设计中，同样也适用于软件设计。也正是基于这样一个简单的设计哲学，我们实现了今天的异常复杂的计算机系统。在介绍完LC-3硬件组成之后，我们开始了基于0和1的机器码语言的机器编程；之后又介绍了LC-3汇编语言，同机器语言相比，它表现得更友好、更清晰；在介绍编程的同时，我们还阐述了“问题分解”方法，即怎样将一个复杂问题分解为用LC-3语言可描述的子问题；随后是I/O操作，即基于TRAP子程序完成输入/输出等任务。“问题分解”和“子程序”是两个重要概念，它们不仅在汇编高级编程中被使用，在本章开始的高级语言编程中也非常重要。在本书中，你将不断看到这些概念的出现。

现在开始本书的第二部分。我们的任务是介绍有关高级语言编程的基础知识（如变量、控制结构、函数、数组、指针、递归和简单数据结构等），以及良好的编程习惯和方法。我们将以C语言为例来讲述这些。当然，本书的目标不是写一个C语言的编程大全。之所以在这里讲述C语言，是希望能通过细节讲解，揭示基本的编程方法和复杂程序的编写技能。读者如果希望了解更多的C语言细节，请参考附录D。

通过本章，我们将完成从汇编语言编程到高级语言编程的过渡。因此，我们将解释高级语言产生的必要性，以及它是怎样与底层系统交互的。最后，将通过一个简单例子认识一下C语言，例子中的细节是C语言编程的入门关键。

11.2 软硬件结合

在计算机硬件越来越快、计算能力越来越强的同时，软件也变得越来越复杂。软件开发面临很多困难。对于一个程序员来说，为维持软件开发过程的简单化，提出了开发过程自动化（即让计算机来做部分开发工作）的需求，这成为一个必然趋势。

正如我们在第5、6、7章所见，LC-3汇编语言编程要比机器语言编程简单得多。例如，在汇编

语言中，我们采用助记符（mnemonic）代替原先的0、1操作码序列，采用符号标识（symbolic label）代表地址。与机器码相比，这种指令与地址的表示方法让我们的阅读感觉更舒服。

另外，如图1-6所示，汇编器在变换层次结构中还起到了衔接“算法层”和“指令集体系结构（ISA，即机器指令）层”的作用。但这还不够，我们希望进一步衔接“语言层”中的间距。于是，高级语言的作用就体现出来了，它的存在使得编程（即算法实现）更简单了。下面，我们来看一下高级语言是怎么做的。

- 数值的符号命名。在机器语言中，表达循环过程中的计数器（counter）需要选定一个内存单元或寄存器来存放计数值。换句话说，这个内存单元或寄存器就是计数器。为了下次访问计数值，我们必须记住存放它的位置（如内存地址或寄存器编号）。在汇编语言中，实现这点已经比较容易了，即为这个计数器地址设置一个标识（label）。但在高级语言（如C语言）中，将非常简单，我们只需要为这个数值分配一个名字（name）及类型（type）。之后，高级语言（更具体地说是编译器）将为它分配合适的位置，并负责相关的数据搬移操作。相比较而言，在汇编语言中需要程序员手工指定位置，并注意不同类型的数据处理，如字节、半字、字等类型。由于程序中会出现大量的数值操作，所以单从这点上，高级语言无疑提高了编程效率。
- 丰富的表现力。在表达多个对象之间的互操作时，人们还是习惯于现实世界中的自然表达方式。相比较而言，数字世界的表达方式则显得“不自然”，如操作时需要注意对象类型是整数、字符还是浮点数等。高级语言中采用的就是自然表达方式，它的“友好”性无疑提高了程序员的“表达”能力。通常，在机器语言中需要花大量代码表达的“复杂”任务，在高级语言中，可能只需要几条语句就可以完成。例如，三角形面积的计算，在高级语言中，只需要一条语句：

```
area = 0.5 * base * height;
```

再如，存在很多条件执行代码，即条件为“真”时做什么事情，条件为“假”时做什么事情。在高级语言中，这类任务可以描述为“类英语”的表达。例如，当条件“阴天”（isItCloudy）为真时，则“取（雨伞）”；否则，“取（太阳眼镜）”。如果用C语言来表示，可以采用“控制结构”（control structure）表示为：

```
if (isItCloudy)
get(Umbrella);
else
get(Sunglasses);
```

- 底层硬件的抽象。高级语言对底层ISA或硬件做了抽象。通常，底层指令不能直接支持某些操作，如LC-3不提供整数乘法指令，解决方法是LC-3的汇编程序员需要编写一大段代码（如循环加）来实现等价的乘法函数。而高级语言通常会向程序员提供完整的操作集合（大于ISA指令集所能提供的），这是因为高级语言程序在生成（或编译为）ISA代码的时候，高级语言负责提供相关的额外代码。这是高级语言的又一特性，它使得程序员可以将精力集中在真正的编程任务上（而不是底层的实现细节）。
- 代码的可读性。由于采用了和英语相似的“控制结构”表示，程序变得更容易被阅读和理解。相比较而言，在汇编语言中，表达“循环”、“条件判断”等结构则要困难得多。如果你没意识到这点，那么很快你就会意识到代码的可读性在编程中的重要性。作为一个程序员，经常会调试别人的代码。如果代码的语言是自然、友好的，那么理解它就将变得容易。
- 可靠性的保证。在高级语言中，存在严格的规则，程序在翻译或执行阶段，将接受规则检查。如果出现规则或条件冲突，则打印出错信息，提示代码中可能存在的BUG。高级语言

的这种特性，能提高程序员的编码效率。

11.3 高级语言翻译

无论采用什么编程语言（汇编或高级语言），它们最终都将被翻译成机器代码，之后才能被底层的机器所识别和执行。如LC-3汇编语言，就要被翻译（或汇编）为机器语言，而高级语言更要经历多次翻译。至于采用什么翻译方法，取决于不同的高级语言。方法一是*解释执行*(interpretation)，负责翻译的程序被称为“*解释器*”(interpreter)，它读入高级语言程序，然后按照高级程序的语义执行相应的操作。真正的执行者不是高级语言代码本身，而是解释器程序；方法二是*编译执行*（compilation），负责翻译的程序被称为“*编译器*”(compiler)，它将高级语言程序读入，然后翻译为机器代码，又称为可执行映像（executable image），该映像是可以直接在机器上执行的。值得一提的是，无论解释器还是编译器，其本身也是一个程序。

11.3.1 解释执行

在解释执行方式中，高级语言程序被看做是一组“命令”的集合。解释器逐个读入每条命令，然后按照语言规范的语义完成命令（执行相应代码）。高级语言程序只是向解释器提供一组“数据”，而不能直接被硬件执行。换句话说，解释器如同是一个*虚拟机*（virtual machine），高级语言是在虚拟机（而不是硬件机器）上被执行的程序。解释器每次处理一个片段，即一行语句或一条命令，甚至是一个子程序。

通常，解释器以行为单位，即每次读入高级语言程序中的一行，然后在实际硬件上运行一段相应的代码，表现出高级语句的执行效果。例如，高级语句的意思如果是：“求变量B的平方根，并将结果存入C”，则解释器将执行一段平方根求解的代码。如此反复，解释器执行完当前行之后，再读入下一行继续执行，直到整个程序结束。

采用解释执行方法的语言包括LISP、BASIC和Perl等。另外，一些专业语言也采用解释方法，如数学语言Matlab。本书的LC-3仿真器也可以认为是一个解释器，还有如UNIX的命令行执行环境（shell）也属于解释器。

11.3.2 编译执行

在编译执行方式中，整个高级语言程序全部被翻译成机器码，然后再在机器上执行。为保证程序翻译的有效性，在翻译之前，编译器会在更大的范围内做分析（通常是整个源文件，而不仅仅是一行代码）。通常程序只需要编译一次，然后就可以多次执行。编译类型的高级语言包括C、C++、FORTRAN等。本书中，LC-3的汇编器就是一个最简单的编译器。编译器是这样一个系统，它能够处理（process）一个或多个包含高级语言程序的文件，生成一个“可执行映像”(executable image)。编译器本身不执行程序，它仅负责翻译任务，即将高级语言转换为机器语言。⊖

11.3.3 两种方法的优缺点

两种翻译技术各有优缺点。解释执行中，程序的开发和调试比较方便。因为解释器每次执行一段（如一行）即输出执行结果，也就是说，程序员可以“在线”(on-the-fly)查看中间结果并修改代码。另外，被解释执行的代码具有很好的可移植性，可以在不同的平台得以执行。但解释执行的缺点也很明显，如执行时间太长，因为存在执行“中介”——即解释器。相比较而言，编译方

⊖ 有些高级编译器确实会执行程序，为的是获取程序的profile信息，优化编译效果。——译者注

式下程序执行速度更快，内存开销也小。编译产生的代码效率更高，所以商业软件开发大都选择编译语言。

11.4 C编程语言

C编程语言是Bell实验室的Dennis Ritchie于1972年发明的。C语言的发明初衷是将它用于编译器和操作系统的开发，因此该语言的设计是偏向底层的。换句话说，C语言可以对底层数据进行直接操作，同时它又兼备高级语言的方便性和表达能力丰富等特性。正因为如此，C语言成为当今使用最广泛的语言，而不仅仅局限在编译器和操作系统的开发。

C语言在编程语言的发展史中占据着重要的地位。图11-1所示是各主要编程语言的发展历史。1954年诞生了历史上的第一个高级编程语言——FORTRAN，之后出现的每个新语言，都是基于前者而产生的。但我们却很难确定一个语言的“父母”，事实上，对任何一种语言来说，之前的各种语言或多或少都对它产生了一定的影响。很明显，C语言对C++和Java有着最直接的影响，而后者是当今最重要的两种语言。同时，Simula语言的面向对象（object-oriented, OO）特性对C++和Java也有重要影响。所以，本书讨论的C语言特性对C++和Java也同样适用。换句话说，掌握本书后半部分内容，对今后的C++和Java学习也非常有帮助，因为它们和C语言非常相似。

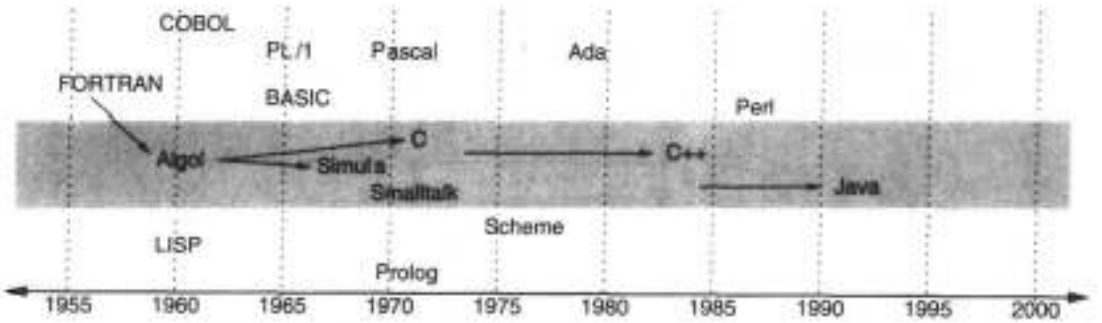


图11-1 编程语言的发展历史：几乎每种新语言与之前的语言之间都存在着一定的关联，C++和Java语言与C之间存在着很密切的关系

鉴于C语言的底层特性及其对现代语言的影响力，它成为本书“自底向上”计算机系统分析过程的首选高级语言。基于C语言，我们很容易在高级语言和底层硬件之间建立关联。至于高层概念中的面向对象等概念，在深入理解C语言的基本概念之后，只需要前进一小步即可理解。

本书采用的C语言遵循ANSI C标准。同其他语言一样，C语言也产生了很多变种。为此，美国国家标准学会（American National Standards Institute, ANSI）于1989年制定了一个“无二义性的、机器无关的C语言定义”（an unambiguous and machine-independent definition of the language C），简称为ANSI C。无论是什么样的C语言变种，它的编译器几乎都支持ANSI C标准。也就是说，无论在什么编译器上编译本书的C程序，只要该编译器是ANSI C兼容的，都可以顺利通过编译和执行。

C编译器

C编译器是一个标准的翻译器典范，它能够将C源程序翻译为一个可执行映像。参考7.4.1节的定义，可执行映像是一个用机器语言表示的、具备内存装载信息的可执行程序。整个编译过程将涉及预处理器（preprocessor）、编译器和链接器（linker）等组件。通常，我们将它们统称为“编译器”（compiler）。编译器在处理过程中，将自动调用预处理器和链接器等组件。图11-2所示是编译过程中各组件之间的协作关系。

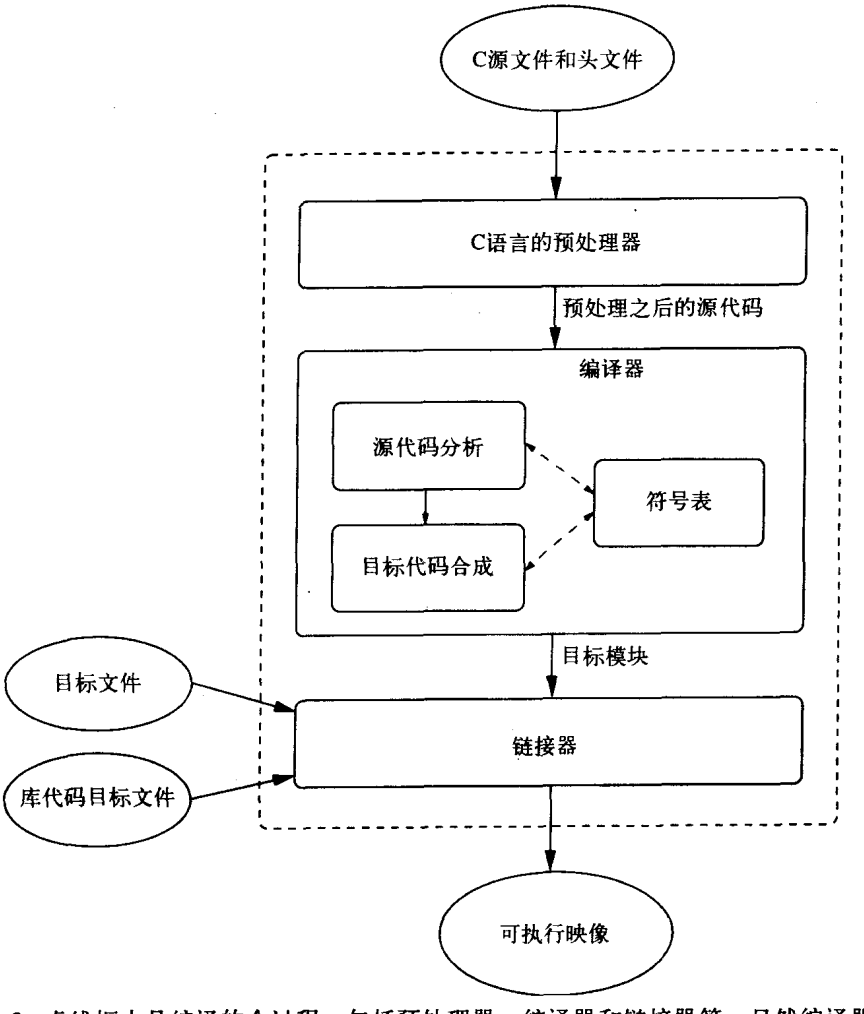


图11-2 虚线框内是编译的全过程，包括预处理器、编译器和链接器等。虽然编译器在其中只是全过程的一部分，我们仍称整个过程为“编译”过程。处理过程的输入是C源程序和头文件及各目标库文件，输出是一个可执行映像

1. 预处理器

顾名思义，C预处理器（preprocessor）的作用就是：在正式启动编译之前，对C程序做“预处理”，然后将预处理结果转交给编译器。C预处理器将扫描所有的C源文件，寻找并执行其中的预处理指令。预处理指令与LC-3汇编语言中的伪操作指令非常相似。预处理器将根据这些指令，执行相关的源文件预处理操作。例如，我们通过预处理指令通知预处理器将代码中所有的字符串“DAYS_THIS_MONTH”替换为数值30；再如，通知预处理器将文件stdio.h的内容拷贝至源文件的当前位置。我们将在后面章节详细讨论它的作用。顺便提一下，在C语言中，预处理器指令行通常以“#”开头。几乎任何C程序都将直接或间接使用预处理器。

2. 编译器

经过预处理之后的程序，将由编译器（compiler）继续翻译为“目标模块”（object module）。参见7.4.2节，目标模块是程序的一个组成部分，是机器语言（或目标）代码形式。编译过程包括两个阶段：一是“分析”（analysis），或称为“语法分析”（parsed），即将源程序分解为更小的组成单元；二是“合成”（synthesis），即生成机器语言代码。分析阶段的任务是读入、分析源代码，

并构建编译器内部格式的数据结构；而合成阶段的任务则是生成机器码，以及根据编译指令优化机器代码。如果将这两个阶段进一步细化，则包括分析（parsing）、寄存器分配（register allocation）、指令调度（instruction scheduling）等子阶段。机器码的生成操作，可以由编译器来做，也可以通过调用汇编器来完成。

符号表（symbol table）是编译器中非常重要的一个内部数据结构，它记录了程序中使用过的所有符号。C编译器的符号表结构和LC-3汇编器中的符号表（参考7.3.3节）非常相似。有关C编译器符号表，将在下一章阐述。

3. *链接器*

所有的目标代码（object code）生成之后，将由链接器完成后续处理。链接器（linker）的任务是将各个目标模块组装成一个可执行映像。可执行映像是一种具有特定格式的机器语言程序，具有可装载和可直接执行这两个特性。例如，在PC机上，当你点击Web浏览器中的图标时，事实上你是在告诉操作系统："从硬盘中读取该图标所对应的执行映像，将其装入内存，然后执行！"

C程序对库函数的依赖非常强。库函数中包含了最常用的一些操作函数（如I/O类），它们以库文件的方式存在。这些库函数通常由系统软件（如操作系统、编译器等）的开发者编写。在用户程序中，如果调用了库函数，则链接器将负责查找该函数在库文件中的位置，并拷贝对应目标代码，然后"链接"入可执行映像。有关程序与库函数目标文件之间的链接操作，你应该不会陌生，因为在9.2.5节的LC-3汇编过程中，也有类似的操作。值得一提的是，在不同的计算机系统中，库文件的存放位置是不同的，如UNIX通常将库文件放在`/usr/lib`目录下。

11.5 一个简单的C程序

下面将介绍C语言编程的概念。针对许多C概念，我们将结合LC-3代码予以解释，即描述C代码在执行时，底层结构中都做了什么事情。在本书中，底层结构是LC-3指令集体系结构，而现实中，你可能面对其他类型的结构（如x86）。例如，如果你的机器是一台Windows-based PC机，则它的体系结构就是x86。也就是说，在你的机器上，C编译器生成的是x86代码（而不是LC-3代码）。

需要声明一下，在本书中，大部分的C语言例程是完整的，你可以直接对它们做编译并运行。但为了使描述更加简洁，有些例程是不完整的，即需要你自己补充代码。因此，为了不引起混淆，以后我们称之为"*代码段*"（而不是程序）。

先看一个简单的C程序。图11-3所示是它的*源代码*。该例程虽然简单，却包含了许多重要的C语言概念。程序意图非常简单：打印提示信息，请用户输入一个十进制正整数，然后对该数字倒计数，直至它变为0。

```
 1  /*
 2   *
 3   *  Program Name : countdown, our first C program
 4   *
 5   *  Description  : This program prompts the user to type in
 6   *  a positive number and counts down from that number to 0,
 7   *  displaying each number along the way.
 8   *
 9   */
10
11  /* The next two lines are preprocessor directives */
12  #include <stdio.h>
13  #define STOP 0
14
15  /* Function    : main                                        */
16  /* Description : prompt for input, then display countdown */
17  int main()
18  {
```

图11-3 该程序提示用户输入一个十进制正整数并将它倒计数直至0

```
19    /* Variable declarations */
20    int counter;          /* Holds intermediate count values */
21    int startPoint;       /* Starting point for count down   */
22
23    /* Prompt the user for input */
24    printf("===== Countdown Program =====\n");
25    printf("Enter a positive integer: ");
26    scanf("%d", &startPoint);
27
28    /* Count down from the input number to 0 */
29    for (counter = startPoint; counter >= STOP; counter--)
30      printf("%d\n", counter);
31  }
```

图11-3 该程序提示用户输入一个十进制正整数并将它倒计数直至0（续）

刚开始时，你可以不必读懂程序中的每一行语句。先尝试着编译一下该程序。在这个程序中，需要强调的概念包括：main函数、代码注释与编程风格、预处理指令、I/O函数调用四个内容。

11.5.1 main函数

从第17行的“int main()”语句开始，至最后一行（第31行）的大括号“}”为止，这几行代码组成了main函数的函数定义。C语言中的“函数”（function）概念与LC-3汇编语言中的“子程序”（subroutine）（第9章）是等价的。函数是C语言中一个非常重要的概念，在第14章将专门介绍这个概念。而main函数又是一个特别的函数：main是任何一个C程序都必备的函数，它代表整个程序执行的起始。在ANSI C中，main的返回值被规定为一个整数，或者称main函数的类型是整数（如第17行代码“int main()”所示）。

在本例中，main函数可分为两个部分：

- 第一部分是“变量声明”（declaration）。所谓变量声明，即对那些在函数中将被使用的变量，如counter和startpoint等，做一个声明。变量是高级语言的又一特性，我们可以将它理解为是数值的符号表示。
- 第二部分是“语句”（statement）。语句表达的是程序的执行动作或行为。对于C程序来说，程序的全部执行过程从main开始，逐句执行，直到main函数的最后一条语句。

下面开始介绍本例中所用到的各个语句。开始几条语句（24~26行）的作用是：打印提示信息（提示用户输入一个正整数）；在接收用户输入的数值之后，程序进入最后一条语句，即for循环语句（一种迭代结构，参见第13章）。循环计数从用户输入值开始，每次倒计数减1，直到计数值为0。例如，刚开始如果用户输入的数值是5，则程序执行的输出结果如下：

```
===== Countdown Program =====
Enter a positive integer: 5
5
4
3
2
1
0
```

我们注意到，每行代码都以分号“；”结尾。在C语言中，分号代表一个声明或语句的结束。编译器将分号理解为“断句符”，它是编译器将程序分解为各个语句的惟一标识。

11.5.2 编程风格

C语言是一种风格自由的编程语言。换句话说，程序中字与字之间、行与行之间的空格数量不影响程序的语义；程序员可以按照自己的意愿，自由组织程序的结构，只要遵守C语言的语法规则即可。基于自由格式的代码具有更强的可读性。例如，在前面的例子程序中，for循环体语句前面的空格（缩进）让这段代码显得非常醒目；再如，在相邻语句之间插入空行，能将整个main函数体分

隔成明显的几个代码段。当然，不必非有这些空行，但有了它们，能在视觉上起到“分隔”代码段的作用。“空行”被经常使用，它将一段复杂的程序，从视觉上分成了相对独立的几个部分。C语言的编程风格有多种流派，本书采用的是最基本的一种C代码规范（如采用缩进、空行等方式）。你完全可以创建自己的风格，但要记住：无论采用什么风格，无非是通过“格式”传达程序的“含义”。

C语言中的注释方式与LC-3汇编语言有所差别。在C语言中，注释信息段以符号“/*”开始，以符号“*/”结束。注释内容可以跨行。例如，在本例中包含了很多注释，有单行的也有多行的。不同的编程语言，其注释方式也总是不同。如C++语言允许注释行以“//”开头。当然，无论哪种注释方式，其目的也都是一样的：提供一种描述机制，使得程序员能够用人类语言（而不是计算机语言）对代码意图做辅助说明。

恰当的代码注释非常重要。漂亮的注释必然能够提高代码的可读性，帮助阅读者快速理解代码。例如，在实际的代码开发中，编程任务通常是由一个团队共同完成，这意味着一段代码将由多人共享。因此，为了保证团队的工作效率，提高代码的可读性，从现在开始，你就应该注重培养对代码做注释的习惯。

一个优秀的注释风格表现为：在每个源文件的开头都会提供一些信息，概述该文件中代码的功能，并记录该文件的修改时间和修改人等历史信息；此外，在每个函数（如main）开始也有注释，描述该函数相关的输入、输出参数；进一步，在代码中也要适当地夹杂“注释”，以辅助对特定代码段的阅读理解。但值得注意的是，“过度注释”并不是件好事，这不仅不会有助于阅读，反而将整个代码文件搞得一团糟，让人抓不住重点。请不要“画蛇添足”，不要写那些可有可无的注释。

11.5.3 C预处理器

之前，在11.4.1节中，我们已对C语言的预处理器做了简要介绍，它是发生在编译器之前，对原始C程序进行转换的一个操作。在前面的例子中，出现了两个常用预处理指令：#define和#include。事实上，预处理指令的种类有很多，但在本书中只涉及这两种。

#define指令的用法虽然简单，但功能却非常强大。它的作用是：指示C预处理器，将程序中出现的所有“文本X”替换为“文本Y”。专业的说法是，将“宏变量X替换为Y”。如在本例中，#define将把文本STOP替换为0。即源代码

```
for (counter = startPoint; counter >= STOP; counter--)
```

在预处理之后（编译之前），将被转化为

```
for (counter = startPoint; counter >= 0; counter--)
```

这有什么用呢？我们在程序中，经常会通过#define创建一些常量，如：

```
#define NUMBER_OF_STUDENTS  25
#define MAX_LENGTH          80
#define LENGTH_OF_GAME      300
#define PRICE_OF_FUEL       1.49
#define COLOR_OF_EYES       brown
```

其中，符号PRICE_OF_FUEL代表石油价格。如果石油价格变化了，我们只要更改此处“宏PRICE_OF_FUEL”的定义即可。如果在程序中会频繁地使用“石油价格”这个数值，那么一旦价格变动，预处理器将为我们完成所有替换，即我们只在宏定义处修改数值，随后代码中所有该石油价格处的数值都同步变化，岂不是非常方便！注意，最后一个例子的用法稍有不同，它是将字符串“COLOR_OF_EYES”替换为另一个字符串（而不是常数）——“brown”。值得一提的是，编程风格的惯例是将宏变量的名字都用大写字母表示。

#include指令的作用是：通过正文方式，指示预处理器在此插入另一个源文件。直观地说，就是将另一个文件的内容拷贝至#include指令所在的地方。这样做有什么意义呢？随着C语言编程的

深入，你将理解C语言中头文件（header file）的用处。例如，可以通过头文件包含#defines宏定义和变量声明，以供多个源文件共享。

以头文件stdio.h为例，它是所有使用I/O函数的C程序都必须包含的一个头文件。在该文件中，定义了库文件中与I/O函数相关的信息。通过预处理指令“#include <stdio.h>”，就在编译之前，将stdio.h文件的内容插入当前源代码文件。

#include指令的表述方式有两种：

```
#include <stdio.h>
#include "program.h"
```

两者的区别是，前者使用尖括号（< >）包含文件名，后者采用双引号（“”）包含文件名。括号表示让预处理器在指定目录中搜索该头文件。指定目录的位置由系统配置决定，该目录中通常包含了系统和库函数相关的头文件（如stdio.h）；然而，有时我们会自己编写头文件（应用相关的）。此时，则可以采用后者，即以双引号指定头文件名，它表示让预处理器在当前C源文件所在的目录中搜索头文件。

再次提醒一下，注意预处理语句的结尾处没有分号（;）。这一点与11.5.1节提醒的不一样。之所以不同，是因为#define和#include是预处理指令，不是C语言语句（或指令）。前者是交给预处理器执行的指令，后者是最终由硬件执行的指令。

11.5.4 输入和输出

本章最后的话题，是有关如何在C程序中使用输入和输出功能。此处有关该话题的描述是高层的，具体的底层细节将在第18章中展开。这是因为只有到那个时候，我们才将积累起理解底层I/O操作所需要的足够知识。任何程序都将或多或少地使用I/O，因此了解C语言的I/O操作机制非常必要。在C语言中，I/O操作由库函数实现，这如同在LC-3中，IN和OUT指令的具体操作是由系统软件所提供的服务程序完成的一样。

在例子程序中有三处代码（第24、25、30行）使用了C库函数提供的printf操作，又称为“格式化打印输出”（print formatted）。printf函数的作用是将信息输出至标准输出设备（如显示器）。输出信息表示为格式化的字符串。所谓“格式化字符串”，是指它包含两方面内容：（1）文本内容的输出；（2）文本中内嵌数值的输出格式说明。例如，下面语句

```
printf("43 is a prime number.");
```

将在输出设备上打印文本

```
43 is a prime number.
```

如果我们希望除文本之外，在输出中还夹杂一些数值，且数值是程序运行时产生的（而不是预定的），那么我们可以在格式字符串中指定数值的输出格式。例如：

```
printf("%d is a prime number.", 43);
```

在格式字符串中包含有格式符“%d”。它表示字符串后面的第一个数值，将以十进制数形式替换格式符%d在字符串中的位置。所以，输出的结果为：

```
43 is a prime number.
```

再例如，下面是printf的其他用法：

```
printf("43 plus 59 in decimal is %d.", 43 + 59);
printf("43 plus 59 in hexadecimal is %x.", 43 + 59);
printf("43 plus 59 as a character is %c.", 43 + 59);
```

在第一个printf中，格式符%d将102（43 + 59的结果）以十进制数方式嵌入输出字符串；在第二个例子中，格式符%x将66（102的十六进制值x66）嵌入字符串；同样，在第三个例子中，格式符%c将数值以ASCII码方式嵌入文本（102等价于小写字母“f”的ASCII码）。所以，第三个输出的结果是：

```
43 plus 59 as a character is f.
```

比较以上三条语句，在字符串尾部提供的数值，其二进制值都是一样的，即0110 0110（十进制数102）。所不同的是，printf的理解方式不同。在第一个语句中，它被解释为一个十进制数，在第二句中它被解释为十六进制数，在第三句中则被解释为ASCII码。参考附录D的表D-6列出的各种printf格式符，其中所有的格式符都以百分号（%）为前缀。

最后，再给出printf的常用方式：

```
printf("The wind speed is %d km/hr.", windSpeed);
```

其中，变量windSpeed的数值是程序运行时产生的。在这里，它将以十进制数方式输出。假设程序执行到当前printf语句时，变量windSpeed等于2，则输出结果为

```
The wind speed is 2 km/hr.
```

但是，如果你写一个程序，将前面5条printf语句连续执行，将发现一个问题——5个输出挤在同一行之中了，即没有换行。如果需要显示换行，可以在格式字符串中需要换行的地方添加换行符。特殊字符如换行符、制表符等，在格式字符串中出现的时候，都以反斜杠符（\）为前缀。例如，重写前面的5条语句，在每个格式字符串后面添加换行符（\n），则每次输出之后都将换行。修改后的printf语句如下所示：

```
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\n", 43 + 59);
printf("43 plus 59 in hexadecimal is %x.\n", 43 + 59);
printf("43 plus 59 as a character is %c.\n", 43 + 59);
printf("The wind speed is %d km/hr.\n", windSpeed);
```

注意其中每个格式字符串后面的换行符（\n）。参考附录D的表D-1中列出的各种特殊字符表示方法。修改后，5条语句的输出结果如下所示：

```
43 is a prime number.
43 plus 59 in decimal is 102.
43 plus 59 in hexadecimal is 66.
43 plus 59 as a character is f.
The wind speed is 2 km/hr.
```

如图11-3所示，例程中有三处调用printf语句。在前两个的输出中，只有文本没有数值（因而没有格式符）。第三个输出中包含变量counter的数值。另外，在一个printf语句中，事实上可以输出多个数值。在字符串中出现的格式符（如%d）数量必须与跟随在字符串后面的数值数量完全相同。

思考题：如果将例程中第三个printf替换为如下形式，结果会如何？

```
printf("%d %d\n", counter, startPoint - counter);
```

在讨论了许多“输出”话题之后，下面看看与之对应的输入函数scanf。scanf函数的任务是：从标准输入设备（如键盘）读入字符。与printf相似，它也包含两部分内容：格式化字符串和变量列表（存放读入的值）。scanf将按照字符串中的格式规范，将读入的字符转换为相应格式的数值，然后存入变量。下面看一个例子。

如图11-3的例程所示，格式规范%d表示将scanf读入的字符转换为一个十进制数。之前我们曾介绍过，在LC-3中接收到的键盘字符都是ASCII编码形式的。而格式规范%d则向scanf函数传达这样一个信息：我们在期待一串数字ASCII键（0~9）的输入。于是当前接收的字符串将被转换为一个十进制数，转换后的二进制值将被存入变量startPoint。注意，其中的类型转换工作是由scanf函数完成的。%d只是scanf众多格式规范中的一个，在附录D的表D-5中，列举了所有的scanf格式规范，如单字符、浮点数、十六进制整数等。

需要提醒的是，那些将由scanf填写的变量（如startPoint），之前都有一个前缀字符（&），这看起来有些神秘，我们将在第16章介绍它的由来。

下面看看更多scanf的例子：

```
/* Reads in a character and stores it in nextChar */
scanf("%c", &nextChar);

/* Reads in a floating point number into radius */
scanf("%f", &radius);

/* Reads two decimal numbers into length and width */
scanf("%d %d", &length, &width);
```

11.6 小结

在本章中，我们介绍了高级编程语言的关键特性，并对C语言编程有了初步认识。下面，我们将对本章的主要话题做个总结。

- 高级编程语言。高级编程语言的任务是简化编程工作，将底层概念与现实对象关联起来。所谓底层概念，是指计算机能处理的位及其位操作。另外，由于计算机只识别机器代码，所以高级语言必须通过翻译或解释的方式转换为机器代码。
- C语言。对于自底向上过程来说，C语言是一个非常理想的语言。这是因为它本质上的底层特性，以及它对现代语言的深远影响。C的编译过程涉及预处理器、编译器和链接器等处理阶段。
- 第一个C程序。我们提供了一个非常简单的程序，并通过它介绍了很多关键的C语言特性，如注释、缩进和编程风格，它们对程序含义的传达非常重要，有助于阅读者更好地理解代码；C程序中通常会使用预处理指令#define和#include等；任何C程序执行的开始都是函数main，它包含了变量声明和语句等组成部分；最后一点是，C的I/O操作可以通过库函数printf和scanf来完成。

11.7 习题

11.1 试列举低级语言编程中存在的问题和不足。

11.2 试问，高级语言是怎样有助于我们减少低级语言编程时所面临的不足的？

11.3 请问高级语言编程存在哪些不足？

11.4 对比解释器执行过程和已编译二进制代码执行过程之间的区别。评价一下解释器方法的性能。

11.5 语言的可移植性是指其代码是否可以在不同的计算机系统（或者说是不同的ISAs）上运行。请阐述为什么解释语言比编译语言有更好的可移植性。

11.6 我们说，UNIX命令行是一个解释器。为什么不说它是一个编译器呢？

11.7 试问，LC-3仿真器属于编译器还是解释器？

11.8 与解释执行相比较，编译执行的另一个优点是，编译器可以对代码做更彻底的优化，这是因为在生成机器代码时，编译器对整个程序进行检查，通过分析程序意图，尽可能地减少程序计算量。

如下所示算法，根据键盘输入值，执行一个简单计算，并将计算结果输出：

(1) 从键盘读入W；

(2) X ← W + W；

(3) Y ← X + X；

(4) Z ← Y + Y；

(5) 输出Z至屏幕。

a. 如果是解释器，则逐句执行整个程序，共有五条语句。请问在该程序执行过程中，解释器

至少要执行多少次算术运算？请详细说明。

b. 如果是编译器，在代码生成之前，将对整个程序做分析，然后优化代码。如果底层ISA具备所有的算术运算能力（如：加、减、乘、除），请问完成该程序至少需要执行多少次操作？请详细说明。

11.9 根据图11-2，回答以下问题。

a. 给出C预处理器的输入；

b. 给出C编译器的输入；

c. 给出链接器的输入。

11.10 如图11-3所示程序，如果将其倒数第二行的语句

```
printf("%d\n", counter);
```

分别替换为如下语句，请问结果如何？

```
a. printf("%c\n", counter + 'A');
b. printf("%d\n%d\n", counter, startPoint + counter);
c. printf("%x\n", counter);
```

11.11 函数scanf的功能是从键盘读入字符，函数printf的功能是把它输出至屏幕。请判断下面两条语句的操作结果：

```
scanf("%c", &nextChar);
printf("%d\n", nextChar);
```

11.12 如果一个C程序中的部分语句如下所示，请问其中每条printf语句的输出结果是什么？

```
#define LETTER 'l'
#define ZERO   0
#define NUMBER 123
printf("%c", 'a');
printf("x%x", 12288);
printf("$%d.%c%d\n", NUMBER, LETTER, ZERO);
```

11.13 请描述一个程序（到目前为止，我们还不要求你能写出可运行的C代码），它能从键盘读入一个十进制数值，然后打印出等价的十六进制数。

第12章　变量和运算符

12.1　概述

本章的主题是高级语言的两个基本概念：变量（variable）和运算符（operator）。变量的作用是存放与程序运行相关的数值，运算符是语言中操作这些数值的各种机制。程序员通过变量和运算符，表达程序要做的各种工作。

下面这行C代码，是一条包含了变量和运算符的语句。该语句的意思是：通过求和运算符（+）将数值3与变量score的数值相加，再通过赋值运算符“=”将运算结果放回score变量。假设在语句执行之前score = 7，那么执行之后变量值就变成了10。

```
score = score + 3;
```

在本章的第一部分中，我们将深入学习C语言的变量。C语言的变量很简单，三种最基本的类型是：整数（integer）、字符（character）和浮点数（floating point number）。在变量之后，是本章的第二部分内容——C运算符，而这部分内容却非常丰富，为此我们提供了大量的讲解例子。在本书中，讲解这两个高层概念的方法也很特别，我们始终将它们与底层实现相关联。而在本章第三部分内容中，我们只是从编译器的角度，讨论在生成机器代码的时候，怎样处理C语言中的变量和运算符。最后，我们将涉及一些问题求解，以及一些与C变量和运算符相关的细节。

12.2　变量

所谓“数值”，泛指程序执行过程中涉及的任何数据项。例如，在循环语句中使用的循环计数器、用户键盘输入的键码值、求和运算中的结果值等都是数值。程序员在这些数值的变化追踪上，要花费大量的精力。

鉴于数值在编程概念中如此重要，高级语言从程序员的角度，对它们的管理过程做了简化设计：高级语言程序员可以采用符号方式描述这些数值，即使用名字（而不是内存地址）来引用数值；当对这些数值做运算操作时，高级语言会自动添加相应的数据搬移操作。换句话说，程序员只需要关注程序的编写，而不需要考虑数据保存在内存的什么地方，以及是将数值放到寄存器还是内存中等细节问题。在高级语言中，我们称这些符号化（有名字）的数值为“变量”。

为了保证变量记录的正确性，高级语言翻译器（如C编译器）需要事先获知变量的很多特征，首先是变量的符号名，其次是变量所含信息的类型，以及变量在程序中的可访问范围等。在大多数的高级编程语言中（包括C语言），是通过变量声明提供这些特征的。

下面看一个例子。如下语句声明了一个名为echo的变量，它可以存储的数值类型是整数。

```
int echo;
```

基于该变量声明，编译器将在内存中为echo预留一个大小为整数的空间（当然，为优化起见，编译器也可将echo存放在寄存器里，而不是内存中。有关话题将在后续的相关课程中介绍）。之后，C代码访问echo变量时，编译器会为之生成相应的机器操作码。

12.2.1　三种基本数据类型：int、char、double

至此，你应该已掌握以下结论：一个特定的二进制数，其含义完全取决于它的数据类型。例

如，二进制码序列0110 0110，既可以代表小写字母f，也可以代表十进制数102。关键在于我们认为该数值的数据类型是ASCII码还是整型。变量声明的作用之一，就是向编译器指明该变量的数据类型。编译器将根据该变量的数据类型，为之分配大小合适的存储空间。另外，数据类型也影响在机器指令级别上操作该变量的运算方式。例如，两个整型变量的加法操作，在LC-3上可以用一条ADD指令完成。但如果是两个浮点数变量，LC-3编译器将生成一组指令来完成它们的加法操作（因为LC-3不具备专门的浮点数加法指令）。

C语言支持整型、字符型和浮点型三种数据类型，C语句中对应的类型说明符分别是int、char和double（双精度浮点数的简称）。

1. int

标识符int所声明的是一个有符号整数变量。整数类型的内部表示及其数值大小的范围，取决于不同的计算机指令集结构（ISA）以及所使用的编译器。例如，在LC-3中，整数类型是指一个16-bit宽度的二进制补码，可表达的数值范围是-32768~+32767。而在一个x86计算机中，整数类型是指一个宽度为32-bit的二进制补码，数值范围是-2147483648~+2147483647。一般情况下，C语言的int与底层ISA的字长（word length）相对应。

如下代码声明的是一个整型变量numberOfSeconds。当编译器看到这个声明的时候，它将为这个变量预留足够的空间大小（以LC-3为例，即一个内存位置）。

```
int numberOfSeconds;
```

整数类型在程序中的使用频度非常高。它们通常用来表示在真实世界中被处理的各种数据。例如，表示时间，以秒为单位，则采用整数类型来表示再合适不过了。再如，在鲸鱼迁徙研究中，我们可以用一个整数变量表示那些正在离开加州海湾的灰鲸数量。同样，在程序控制中，整型变量也非常有用，循环计数器通常也是整数类型的。

2. char

标识符char表示该变量所包含的内容是一个字符类型数据。如下所示是两个char类型变量的声明，一是名为lock的变量，二是名为key的变量。不同的是，在第二个声明语句中包含“初始化”操作。在C语言中，允许变量在声明之时对其赋一个初始值。例如，在本例中，变量key的初始值为大写字母*Q*的ASCII码值。注意，其中字母*Q*是被单引号（‘’）所包围的。单引号在此的作用是，表示这是一个ASCII字符值。那么，试问变量lock的初始值又是多少呢？稍后我们将讨论这个问题。

```
char lock;
char key = 'Q';
```

对于ASCII字符，一个8-bit的空间就足以表示任意ASCII字符。但在本书中，为简洁起见，所有的char变量所占用的空间大小都是16-bit。换句话说，char与int一样，都将独自占用一个内存位置。

3. double

double代表浮点类型变量（参见2.7.2节）。浮点数类型主要用于表示那些包含小数的数值，或者是非常大或非常小的数值。在2.7.2节中曾提到，浮点数的具体表示包括：符号、尾数和指数等三个部分。

以下是double类型变量的三个例子：

```
double costPerLiter;
double electronsPerSecond;
double averageTemp;
```

与int和char相同，浮点类型变量在声明之时也可以被初始化。在介绍浮点变量的初始化操作之前，我们先看一下C语言中浮点数的表示方法。在如下语句中，我们看到，一个浮点数，可能只有尾数部分，也可能只有指数部分，亦或两者都有。字母e或E之后标识的是数值，其值或正或负。指数部分以10为幂，它与e或者E之前的尾数相乘，得到最终的数值。注意，指数部分必须是一个

整数值。有关浮点数更多的描述，参见附录D.2.4。

```
double twoPointOne = 2.1;        /* This is 2.1   */
double twoHundredTen = 2.1E2;    /* This is 210.0 */
double twoHundred = 2E2;         /* This is 200.0 */
double twoTenths = 2E-1;         /* This is 0.2   */
double minusTwoTenths = -2E-1;   /* This is -0.2  */
```

在C语言中，还有一种浮点类型是float。所不同的是，它表示的是单精度浮点变量，而double表示的是双精度浮点变量。回顾在第2章中有关浮点数的描述，一个浮点数的精度取决于其尾数字段的位数。而在C语言中，根据编译器以及指令集结构的不同，double的尾数位数可能大于float的尾数长度，但绝不会小于float的尾数长度。而整个double数所占空间大小也取决于具体的指令集结构和编译器。按照IEEE 754浮点数标准，double类型的长度通常是64位，而float类型的长度是32位。

12.2.2 标识符的选择

对于大多数高级语言，变量名字（又称“标识符”）有各种命名规范可供选择。C语言规定，一个标识符可由字母、数字或下划线字符（_）组成。其中，标识符的起始字符可以是字母和下划线字符。标识符的长度随意，但事实上C编译器只认前31个字符。另外，C编译器对字母的大小写敏感，也就是说，在C语言中，Capital和capital是两个不同的标识符。

下面介绍一些标准的C语言命名规范：

- 起始字符为下划线的变量名（如_index_），通常只用于标准库代码中。
- 全为大写字母的标识符，通常只用于预处理指令#define中（参见11.5.3节），变量名不会全大写。
- 多单词的变量名。从视觉上，程序员通常会对变量名中各个单词做出分隔。在本书中，我们采用大写字母方法（如wordsPerSecond）；而有的程序员则喜欢下划线方法(如words_per_second)。

变量的命名在代码编写中非常重要。变量名应该能反映出该数值的特性，它有助于程序员联想到变量的含义。例如，打字员每秒钟的打字速度，其变量名可以是“wordsPerSecond”。

值得注意的是，在C语言中有许多关键词（reyword），它们具有特殊含义，所以标识符命名不能与它们冲突。在附录D.2.6中，列举了C语言的关键词。例如，之前已接触过的“int”就是一个关键词，这意味着我们不能将任何变量命名为int，否则代码阅读者会因此而困惑。编译器在翻译之时，也会对此出现二义性问题，因为编译器无法确认它是代表“int变量”还是“int类型”。

12.2.3 局部变量和全局变量

如前所述，变量声明的作用是方便编译器管理变量的存储问题。在C语言中，程序中的变量声明是向编译器传递以下三类信息：变量的标识符、类型及其作用域。有关前两者，编译器可直接从变量声明语句中获取；但是，有关作用域，编译器则要通过声明语句所在的位置推理而知。所谓“变量的作用域”，是指有权访问该变量的代码区段（或者说是变量存活区域）。

幸运的是，C语言的变量作用域只有两类：一是“全局的”（global）（在整个程序范围内可访问[⊖]），二是“局部的”（local）（只在当前代码块有效）。

1. 局部变量

在C语言中，任何变量在被使用之前必须先声明。有些变量是在它们所在代码块的开始处声明，我们称之为局部变量（local variable）。这里所说的“代码块”，是指程序中的一段代码，它以左括

⊖ 这个说法有些不严格。事实上，在C语言中，可以标识一个全局变量是在当前源代码文件中有效，还是在整个程序中有效。但这点并不影响我们在此的讨论。

号“{”开始，以右括号“}”结束。局部变量的声明语句通常紧随在左括号后面。

下面的代码是一个简单的C程序，它从键盘读取一个数字，然后将它输出，显示在屏幕上。其中，整数变量echo的声明是在main函数内，这意味着它只在main函数内“可见”，而在main函数之外的代码区域是无法访问该变量的。通常，局部变量是在当前代码块开始处被声明，如本例的echo变量。

```
#include <stdio.h>

int main()
{
   int echo;

   scanf("%d", &echo);
   printf("%d\n", echo);
}
```

但出于特殊目的，我们有时会在同一个函数体内，在不同代码块中，声明两个名字完全相同的变量。例如，在同一个程序内，不同的循环体中，都使用名字为count的计数器变量。C语言允许这种声明方式，但条件是这些声明必须在不同的代码块内。图12-1所示就是一个例子。

```
#include <stdio.h>

int globalVar = 2;            /* This variable is global */

int main()
{
  int localVar = 3;   /* This variable is local to main */

  printf("Global %d Local %d\n", globalVar, localVar);

  {
    int localVar = 4;         /* Local to this sub-block */

    printf("Global %d Local %d\n", globalVar, localVar);
  }

  printf("Global %d Local %d\n", globalVar, localVar);
}
```

图12-1 C程序的嵌套作用域

2. 全局变量

局部变量只能在当前代码块中被访问，与之相比，全局变量则可以在程序的任何地方被访问。换句话说，它们在整个程序的生命周期内，都占有独立的空间及其内容。

在下面这段代码中，main函数同时使用了全局变量和局部变量。

```
#include <stdio.h>

int globalVar = 2;            /* This variable is global */

int main()
{
  int localVar = 3;   /* This variable is local to main */

  printf("Global %d Local %d\n", globalVar, localVar);
}
```

全局变量在特定的编程环境下将非常有用。但初学者会经常听到这样的告诫：“尽量使用局部变量，少用全局变量”。为什么这么说呢？因为全局变量是公用的（public），它可以在程序的任何地方被修改。这意味着代码可能存在隐患，对它的修改和重用需格外小心。在本书的C程序中，几乎只使用局部变量。

下面看一个更复杂的例子。图12-1所示的C程序与之前的程序非常相似，不同之处是我们在main函数中添加了一个子块（sub-block）。而在该子块中，我们又声明了一个localVar变量，它和

main开头声明的局部变量localVar名字完全相同。但你将发现，子块执行时，之前声明的localVar突然不可见了。也就是说，新声明的同名变量替代了原先的同名变量。但是，一旦子块结束，之前的localVar又可见了。这就是所谓的“嵌套作用域”(nested scope)。

3. 变量初始化

至此，我们已介绍了全局变量和局部变量，现在开始回答之前的问题：“如果一个变量没有初始化，那么它的初始值是多少呢？”在C语言中，局部变量的默认初始值是不确定的。因为，为局部变量分配的存储空间并未被特意清除，它包含的是之前存储在此的内容。我们通常称这种情况为：在C语言中，局部变量未被“初始化”(尤其是“自动存储类变量”(automatic storage class))。与之相反，全局变量（以及所有其他“静态存储类变量”(static storage class variable)）在程序执行之前，都将被初始化为0。

12.2.4 更多的例子

下面我们将列举更多的C语言变量声明示例。其中，包含了本章介绍的三种基本数据类型。另外，这些变量声明中，有的未被初始化，有的将被初始化。尤其要注意其中的浮点数和字符的C语言表示方法。

```
double width;
double pType = 9.44;
double mass = 6.34E2;
double verySmallAmount = 9.1094E-31;
double veryLargeAmount = 7.334553E102;
int average = 12;
int windChillIndex = -21;
int unknownValue;
int mysteryAmount;
char car    = 'A';
char number = '4';
```

在C语言中，还可以使用十六进制数，它的表示方法是前缀“0x”。如下例子是3个十六进制整数变量的初始化。

```
int programCounter = 0x3000;
int sevenBits = 0xA1234;
int valueD = 0xD;
```

思考题：在以上声明之后，如果执行语句“printf("%d\n",valueD);”，输出结果是什么？那么在valueD对应的内存中，存放的二进制数又是什么？

12.3 运算符

前面介绍了C语言变量，下面将介绍C语言的运算符。与所有高级语言一样，C语言也支持丰富的运算符集合，即程序员可以对变量实行多种操作。这些运算符中，有数学运算类的，也有逻辑运算类的，或是数值比较类的。有了这些运算符，程序员不用汇编指令，就可以方便简洁地表达各种运算。

编译器所做的工作，则是将C代码转化成底层硬件能够识别的机器码。以C代码转换成LC-3机器码为例，编译器必须把C程序中的各种操作翻译成LC-3指令集支持的指令。显然，这并不容易，因为LC-3支持的操作指令种类非常少。

为了更好的阐述这个观点，我们以一条简单的“两数相乘”语句为例。其中，x、y、z都是整数变量，x和y的乘积结果赋给了z。

```
z = x * y;
```

由于LC-3不支持乘法指令，所以LC-3编译器必须通过一段代码序列，模拟两个整数（甚至可能是负数）的乘法操作。一种可行的解决方法是，将x的值自身叠加y次。这与第10章中计算器例

子的代码非常相似。图12-2所示是LC-3编译器为上面的C语句生成的LC-3机器码程序。假设寄存器5（R5）的内容是变量x的内存地址，而该地址之前的内存位置（即R5-1）存放的是变量y的数值，再之前是变量z。变量的这种地址分配方式看起来有些奇怪，其中的原因将在12.5.2节中予以解释。

```
        AND  R0, R0, #0   ;   R0 <= 0

        LDR  R1, R5, #0   ;   load value of x
        LDR  R2, R5, #-1  ;   load value of y
        BRz  DONE         ;   if y is zero, we're done
        BRp  LOOP         ;   if y is positive, start mult

                          ;   y is negative
        NOT  R1, R1       ;
        ADD  R1, R1, #1   ;   R1 <= -x

        NOT  R2, R2
        ADD  R2, R2, #1   ;   R2 <= -y (-y is positive)

LOOP    ADD  R0, R0, R1   ;   Multiply loop
        ADD  R2, R2, #-1  ;   The result is in R2
        BRp  LOOP

DONE:   STR  R0, R5, #-2  ;   z = x * y;
```

图12-2 对应C乘法语句的LC-3代码

12.3.1 表达式和语句

在深入学习运算符之前，我们先介绍一些C语言的语法知识，这将有助于我们理解C程序。我们将变量、常量值与运算符（如乘法）的组合称为“C表达式”，如前面的“x * y”就是一个表达式。

表达式之间的组合则构成“语句”。例如“z = x * y”就是一个语句。C的语句与英语句子相似，一个完整的句子表达的是一个想法或行为，而一个C语句表达的是计算机可以完成的一个操作。C语句以分号“;”或右大括号“}”结束。语句中分号的作用如同英语句子中的标点符号。C语言还有一个有趣的特点，它可以创造一个不表达任何任务的句子，如仅有一个分号的语句也是合法的，即“空语句”（什么也不做）。

多个基本语句的组合称为“复合”语句或“代码块”（由左右大括号{}包围）。从语法上讲，复合语句等价于基本语句。下一章，我们将介绍各种复合语句的用法。

下面是简单语句、复合语句和空语句的例子比较。

```
z = x * y;   /* This statement accomplishes some work */

{            /* This is a compound statement          */
  a = b + c;
  i = p * r * t;
}

k = k + 1;   /* This is another simple statement      */
;            /* Null statement -- no work done here   */
```

12.3.2 赋值运算符

C语言赋值运算符的符号是“=”（读作“等号”）。它的含义是，先对等号右边的表达式进行运算，然后将右边表达式的结果赋给等号左边的对象。例如，C语句

```
a = b + c;
```

该语句的作用是，将表达式“b + c”的运算结果赋值给变量a。

值得注意的是，虽然数学的等号和C的赋值运算符号完全一样，但它们的含义却完全不同。在数学中，等号“=”用来表示右边和左边的表达式完全相等；但在C语言中，“=”运算符用来表示

赋值（即传递数值），编译器将为之生成合适的代码。

下面是LC-3编译器为C赋值语句生成汇编代码的例子。C语句的作用是对变量x原数值增量4。

```
x = x + 4;
```

与该语句对应的LC-3代码也非常简单。其中，R5的内容是变量x的内存地址。

```
LDR  R0, R5, #0   ;   Get the value of x
ADD  R0, R0, #4   ;   calculate x + 4
STR  R0, R5, #0   ;   x = x + 4;
```

在C语言中，表达式的计算结果是“有类型的”。在这个例子中，表达式“x + 4”的计算结果是整数类型，因为数值4与整型变量（x）之和也是整数，结果也是赋给一个整型变量。那么，如果我们将不同类型的数放在同一个表达式中（即类型混合），如“x + 4.3”，结果如何呢？C语言的处理规则是，将其中的整数转换为浮点数。如果表达式同时包含整型和字符型，则其中的字符型被转换为整型。总之，C语言总是将短类型（shorter type）转换为长类型（longer types）。再问，如果我们将一种类型的表达式结果赋给另一类型的变量，结果如何呢？如“x = x + 4.3”。C语言对此的处理规则是，保持变量类型不变，反之，将表达式结果转换为变量类型。如本例中，浮点表达式“x + 4.3”的结果将被转换为整型（变量x的类型）。在转换过程中，浮点数的尾数部分将被丢弃。例如，浮点数4.3转换为整数后结果为4，若为5.9，则转换为5。

12.3.3 算术运算符

在C语言中，算术运算符相对较易理解，因为大多数运算及其符号我们都很熟悉，与数学课上介绍的运算符完全一样。例如，“+”代表加法，“-”代表减法，“*”代表乘法（稍有不同，主要是为了避免它与字母x冲突），“/”代表除法。另外，与四则运算规则一样，表达式运算也是有顺序要求的。乘除优先，然后是加减。有关运算符的执行顺序，我们称为“优先级”，下一节将详细讨论。下面是几个C算术运算符语句。

```
distance = rate * time;
netIncome = income - taxesPaid;
fuelEconomy = milesTraveled / fuelConsumed;
area = 3.14159 * radius * radius;
y = a*x*x + b*x + c;
```

C语言还有一种算术运算符，你可能不太熟悉，它就是取模（modulus）运算符“%”，又称“整数取余运算符”。我们将以两数相除为例，解释它的用法。我们知道，在C语言中，整数相除时，尾数部分将被丢弃，结果只保留整数部分。例如，表达式“11 / 4”的结果等于2。然而，取模运算符%求解的是除法运算的余数部分。例如，“11 % 4”的结果等于3。换句话说，数值11等于“(11 / 4) * 4 + (11 % 4)”。在如下例子中，变量的类型都是整型。

```
quotient  = x / y;  /* if x = 7 and y = 2, quotient = 3  */
remainder = x % y;  /* if x = 7 and y = 2, remainder = 1 */
```

表12-1是C所支持的算术运算符号。其中，乘法、除法、取模的运算优先级高于加法和减法。

表12-1 C语言里的算术运算符

运算符符号	操作	示例用法	运算符符号	操作	示例用法
*	乘法	x * y	+	加法	x + y
/	除法	x / y	-	减法	x-y
%	取模	x % y			

12.3.4 算术优先级

至此，在继续讨论其他C运算符之前，我们先回答一个重要问题：如下语句中，x的值是多少？

```
x = 2 + 3 * 4;
```

1. 优先级

首先，该运算涉及运算过程的顺序问题。与手工计算一样，表达式的推算过程是“有顺序的”，我们称为运算符的“优先级”(precedence)。例如，在做算术运算时，我们认为乘除的优先级高于加减。C语言的运算优先级规则，与我们的小学算术规则是一致的。所以，该语句的计算结果是，x等于14。换句话说，该表达式等价于“2 + (3 * 4)”。

2. 关联顺序

但是，如果表达式中运算符的优先级相等，该怎样处理呢？即如下语句的结果如何？

```
x = 2 + 3 - 4 + 5;
```

最先，被处理的运算符不同，则结果完全不同。表达式“2 + 3−4 + 5”的结果可能等于6，也可能等于−4。这是因为，在C语言中，加法和减法的优先级相同。因而，我们还必须为C语言定义一个推算规则。尤其是对于优先级相等的运算符，它们之间的关联性（associativity）决定了它们之间的运算顺序。例如，加法和减法之间，它们的关联关系都是“自左向右”，所以“2 + 3−4 + 5”的运算过程等价于“((2 + 3) −4) + 5”。

在本章的表12-5和附录D的表D.4中，都列举了C运算符的优先级和关联性规则。不过，我们不主张你去死记这张表（除非你希望能向朋友炫耀你对C语言细节的熟悉程度）。事实上，你只需要知道有这么一个优先级规则的存在，并大致了解它们背后的逻辑即可。当你不确定特定运算符之间的关系时，再来查阅该表。当然，还有一种最可靠的方法——使用括号。

3. 括号

括号不受任何运算规则的约束，换句话说，它能显式地指示哪些运算符优先，哪些运算符次之。在通常的算术运算中，我们总是从最里的圆括号开始计算。就是说，如果我们希望哪部分表达式先计算，则用圆括号将这个子表达式包围起来。在下面例子中，假设其中的a、b、c、d都为4。语句

```
x = a * b + c * d / 2;
```

等价于

```
x = (a * b) + ((c * d)/4);
```

在两个表达式中，x的值都为20。也就是说，不管是依照运算符优先级规则，还是通过括号显示地指明运算次序，程序总是先匹配括号，从里向外，依次转移至外层。在没有括号的情况下，才动用优先级规则。

试问，如果a、b、c、d都为4，则下面表达式结果如何？

```
x = a * (b + c) * d / 4;
```

另外，括号使得代码的可读性更强。而且，大多数人不一定能记住C语言里的优先级规则。所以，对于复杂的表达式，我们通常使用括号，虽然没有这些括号，代码同样能正常工作。

12.3.5 位运算符

现在，我们继续介绍C语言运算符。C语言中有一组“位运算符”，它们的作用是可以操作数值的任意bit；即位的逻辑操作（如AND、OR、NOT、XOR等）。例如，在C语言中，位运算符“&”的作用等价于LC-3的AND指令，“&”运算符的操作是对两个数中的对应位依次相“与”(AND)。类似，运算符“|”执行按位的“或”操作，运算符“~”执行按位的“非”操作（一元运算符，只需要一个操作数），运算符“^”执行按位的“异或”操作。下面是这些位运算在十六位数值之间的运算例子。

```
0x1234 | 0x5678  /* equals 0x567C */
0x1234 & 0x5678  /* equals 0x1230 */
0x1234 ^ 0x5678  /* equals 0x444C */
-0x1234          /* equals 0xEDCB */
1234 & 5678      /* equals 1026  */
```

另外，C语言的位运算符中，还包含两个“位移”运算符：左移操作“<<”和右移操作“>>”。这两个运算符是二元运算符，即需要两个操作数，一是被位移数值，二是位移位数。左移操作中，右边移空位补0，右移操作中，左边移空位做符号扩展。表达式的值是运算结果，但两个操作数本身的值不变。下面的表达式是十六进制整数的位移运算表达式：

```
0x1234 << 3      /* equals 0x91A0  */
0x1234 >> 2      /* equals 0x048D  */
1234 << 3        /* equals 9872    */
1234 >> 2        /* equals 308     */
0x1234 << 5      /* equals 0x4680  (result is 16 bits)   */
0xFEDC >> 3      /* equals 0xFFDB  (from sign-extension) */
```

下面例子是包含“位运算”表达式的C语句。注意，C的位运算符不支持对浮点数的位操作。例如，下面语句中的变量f、g和h都是整数。

```
h = f & g;      /* if f = 7, g = 8, h will equal 0     */
h = f | g;      /* if f = 7, g = 8, h will equal 15    */
h = f << 1;     /* if f = 7, g = 8, h will equal 14    */
h = g << f;     /* if f = 7, g = 8, h will equal 1024  */
h = ~f | ~g;    /* if f = 7, g = 8, h will equal -1    */
                /* because h is a signed integer       */
```

思考题：假设在特定机器上，整数变量x的长度是16位，其值为1。语句“x = x << 16;”的执行结果如何？理论上，将x移动16位（即它本身的数据宽度）之后，所有位应该都被置0，即x的值应该为0。但为了保持通用性（C语言并不知道特定机器的字宽度是多少），C语言规定位移运算（等于甚至超过数据本身宽度）的结果取决于具体实现。也就是说，在不同的机器上，移动相同位数之后，其结果可能为0也可能不为0，它完全取决于运行系统。

表12-2列出了所有位运算符。运算符的排列以优先级为顺序，其中“取反”（NOT）运算符的优先级最高，然后是“左移”和“右移”（两者优先级相同），随后是“与”（AND）、“异或”（XOR）和“或”（OR）运算符。它们的关联顺序都是自左向右。表12-5是运算符优先级的完整列表。

表12-2 C语言里的位运算符

运算符符号	操　作	示例
~	按位“取反”（NOT）	~x
<<	左移	x << y
>>	右移	x >> y
&	按位“与”（AND）	x & y
^	按位“异或”（XOR）	x ^ y
|	按位“或”（OR）	x | y

12.3.6 关系运算符

C语言中，关系运算符的作用是测试两个数值之间的关系。在下一章中，我们将在“条件结构式”中使用它们。参考6.1.2节中有关系统分解过程中的条件结构式。

相等运算符“==”是一个C关系运算符。该运算符的作用是测试两个数值是否相等。如果它们相等，则表达式值为1；如果不相等，则表达式值为0。下面是两个例子：

```
q = (312 == 83);   /* q will equal 0 */
z = (x == y);      /* z will equal 1 if x equals y */
```

第二个例子中，赋值运算符（=）右边的表达式是“x == y”，它只有两个可选结果（1或0，取决于x和y是否相等）。值得一提的是，其中的括号不是必需的，因为“==”运算符的优先级高于运

算符“=”。但是，显然有了这个括号，表达式的意思看起来更清晰。

与“=”运算符相反，不相等运算符“!=”，在两个操作数不相等时，表达式值为1。另外，还有其他关系运算符，如“大于”(>)、“小于”(<) 等等，我们将在下面例子中予以描述。假设变量f、g和h都是整数，且变量f的值为7，g的值为8。

```
h = f == g;     /* Equal To operator.      h will equal 0  */
h = f > g;      /* Greater Than operator. h will equal 0  */
h = f != g;     /* Not Equal To operator. h will equal 1  */
h = f <= g;     /* Less Than Or Equal To. h will equal 1  */
```

下面的这个例子，表现的是关系运算符的另一种用法。通过对变量的关系测试，可以控制程序流程的改变。我们将在下一章的C语言if语句中，详细介绍它的使用技巧。不过，if结构的内容已不是什么新概念，因为我们在第6章的LC-3编程中，就已接触过这个特别的判断结构式。下面例子的意思是，当变量tankLevel（油箱油位）等于0时，打印输出一个消息“油箱空了”。

```
if (tankLevel == 0)
   printf("Warning: Tank Empty!!\n");
```

表12-3列出了所有关系运算符，以及每个运算符的简单例子。前面4个运算符的优先级高于后面两个。所有运算符的关联顺序都是自左向右。

表12-3 C语言里的关系运算符

运算符符号	操　作	示例用法	运算符符号	操　作	示例用法
>	大于	x > y	<=	小于或等于	x <= y
>=	大于或等于	x >= y	==	等于	x == y
<	小于	x < y	!=	不等于	x != y

12.3.7 逻辑运算符

C语言的逻辑运算符，第一眼看上去似乎与位运算符很相似。事实上，编程新手经常会将两者混淆。为此，我们先介绍一下逻辑值“真”(true) 和“假”(false) 的概念。在C语言的逻辑运算概念中，非零值（即其值不为零的任何数值）被认为是逻辑真，数值零则被认为是逻辑假。这个定义非常重要，因为后面的内容中，我们会反复涉及此概念。

C语言支持三种逻辑运算符：&&、||和！。运算符“&&”表示两个操作数的逻辑与 (AND)，如果两个操作数都为逻辑真（或非零），则逻辑表达式值为1（即逻辑真）；否则，逻辑表达式值为0。例如，表达式“3 && 4”的结果为1，而“3 && 0”结果为0。运算符“||”代表逻辑或 (OR)。例如，如果x或y有一个值为非零，则表达式“x || y”值为1。因此，表示式“3 || 4”的结果为1，表达式“3 || 0”结果也为1。运算符“！”表示对操作数的值逻辑取反。因此，当且仅当数值x为0时，!x才为1；否则，结果为0。

那么，“逻辑运算符”有什么用呢？在程序中，它可被用于构建“逻辑条件”。例如，我们可以结合关系运算符和逻辑运算符，判断变量值是否落入特定范围内。比如对变量x，判断其数值是否在10~20之间（包括10和20），则表达式的形式如下：

```
(10 <= x) && (x <= 20)
```

再如，判断字符变量c是否是一个合法字母（即是否在字母表范围内），表达式如下：

```
(('a' <= c) && (c <= 'z')) || (('A' <= c) && (c <= 'Z'))
```

下面是逻辑运算符的各种示例，其中包含前面介绍过的“位运算符”。之所以将它们列在一起，是为了比较它们之间的差异。与之前的例子相同，变量f、g和h都是整数类型，变量f等于7，g等于8。

```
h = f & g;      /* bitwise operator: h will equal 0  */
h = f && g;     /* logical operator: h will equal 1  */
h = f | g;      /* bitwise operator: h will equal 15 */
h = f || g;     /* logical operator: h will equal 1  */
```

```
h = ~f | ~g;   /* bitwise operator: h will equal -1 */
h = !f && !g;  /* logical operator: h will equal 0  */
h = 29 || -52; /* logical operator: h will equal 1  */
```

表12-4列出了C语言中的逻辑运算及其符号。逻辑非运算符的优先级最高，然后是逻辑与，逻辑或。如果想要知道运算符优先级的完整列表，可以参看表12-5。

表12-4 C语言中的逻辑运算符

运算符符号	操 作	示 例
!	逻辑非	! x
&&	逻辑与	x && y
||	逻辑或	x || y

表12-5 运算符优先级（从高到低排列，括号内为运算符的说明）

优 先 级	关联顺序	运 算 符
1（最高）	自左向右	()（函数调用） []（数组索引） .->
2	自右向左	++ -- （后缀版）
3	自右向左	++ -- （前缀版）
4	自右向左	*（间接引用） &（取地址） +（一元） -（一元） ~ ! sizeof
5	自右向左	(type)（类型转换）
6	自左向右	*（乘） / %
7	自左向右	+（加） -（减）
8	自左向右	<< >>
9	自左向右	< > <= >=
10	自左向右	== !=
11	自左向右	&
12	自左向右	^
13	自左向右	|
14	自左向右	&&
15	自左向右	||
16	自左向右	?:（条件表达式）
17最低	自右向左	= += -= *= 等

12.3.8 递增/递减运算符

变量的增、减操作非常频繁，C语言的设计者为此设计了专门的运算符。运算符“++”代表变量数值的“递增”（increment），运算符“--”代表变量数值的“递减”（decrement）。例如，表达式“x++”表示将整型变量x的值递增1，表达式“x--”表示将整型变量x的值递减1。值得注意的是，它们都是对变量自身值的改变。因此，表达式“x++”和“x = x + 1”的效果是一样的。

运算符“++”和“--”，可以放在变量的任意一边（左或右）。但是，表达式“++x”和“x++”之间存在细微差别。在一个复合表达式中，如果包含“x++”，则“x++”代表的是x递增之前的值；相反，如果是“++x”则代表的是x递增之后的值。我们称运算“++”出现在变量名前面为“前缀”（prefix）形式，而出现在变量名后面为“后缀”（postfix）形式。前缀形式称为“先增”（preincrement）或“先减”，后缀形式则称为“后增”（postincrement）或“后减”。

下面，我们看一对例子：

```
x = 4;
y = x++;
```

此处，虽然变量x的值递增了，但赋给变量y的还是x的原值（即表达式x++的值等于原x的值）。

代码执行之后，变量y为4，x为5。

再看，与前面类似的两行代码：

```
x = 4;
y = ++x;
```

在此，变量x的值也被递增。但是，在这个代码中，表达式“++x”代表的是x递增之后的值。执行结果是，x和y的值都为5。

前缀形式和后缀形式的微小区别，到现在为止理解起来应该不是太难。本书中的大多数例子里，前缀形式运算符和后缀形式运算符都是可以替换的。在附录D.5.6中，有关于这个区别的更详细描述。

12.3.9 运算符混合表达式

至此，我们所列举的表达式中都最多只包含一个或两个运算符。在实际代码中，表达式可能包含多个运算符。多种运算符和操作数组合在一起，可以构建非常复杂的表达式。如下所示是一个混合多种运算符的复杂表达式：

```
y = x & z + 3 || 9 - w % 6;
```

推算该语句运算结果的第一步，是检查运算符的操作顺序。参见表12.5中的C运算符（其中有一些我们目前还未介绍的运算符）及其优先级。参照优先级规则，该语句等价为如下语句：

```
y = (x & (z + 3)) || (9 - (w % 6));
```

再如，下面的表达式中，如果变量age的值在18到25之间，则表达式的结果为1，否则为0。注意，其中的括号不是必需的，但有了它们，可使表达式显得更易于阅读，代码更易于理解。

```
(18 <= age) && (age <= 25)
```

12.4 基于运算符的问题求解

至此，我们已经学习了足够多的C运算符。下面，我们将学习怎样使用它们去求解简单问题。现在，我们将编写一个程序，计算在特定传输速率（每秒字节数）下，网络传输一定数量的字节所需要的时间。问题的难度主要是怎样用“小时/分钟/秒”的形式显示传输时间。

基于第6章介绍的问题分解技术，我们将从程序描述开始，然后将其分解为各个子问题，并分别采用顺序、判断和循环等结构式描述它们，最后为每个结构式或模块编写C代码。我们称这种技术为“自顶向下分解”，因为我们从粗略的算法描述开始，然后将各步骤尽力细化分解，直到它可以被编程实现。对于有经验的程序员，依靠他们对底层系统的理解，能够更准确地决策问题应该怎样分解。同时，为了使用程序方法解决问题，他还必须熟悉了解编程系统的基本要素。到目前为止，这些基本要素就是三种变量类型，以及相关的操作运算符。

在后面的章节里，我们还将陆续介绍几个问题求解例子，以阐述自顶向下分解过程的方法，帮助程序员更好地理解问题求解的过程。

第一件事情（即第0步），就是思考怎样表示数据对象。基于现有知识，我们能够选择的就是从C语言的三种基本类型中，为数据对象选择数据类型：整型、字符型或浮点型。针对网络传输时间问题，可以选用浮点数或整数来表示内部计算量。但由于最终要显示的时间格式是“小时/分钟/秒”的形式，其中的小数部分在此是不需要的。例如，传输时间等于“10.1小时12.7分9.3秒”，是不合适的，正确的表示应该是“10小时18分51秒”。因此，我们决定选用整型而不是浮点数类型（的确，这会产生四舍五入的问题，但针对该问题，我们可以将它们忽略）。

在确定数据类型之后，我们开始逐步细化、分解问题。图12-3是该问题的分解示意图。图中的第1步是勾画问题的基本组成，即包括读取输入、计算、输出结果等三个阶段。其中，第一阶段是获取数据传输总量（字节单位）和网络传输速率（字节每秒单位），第二阶段是执行相关计算，

第三阶段是输出特定格式的结果。

第1步的描述，其细致程度还不足以将它们直接翻译为C代码，所以我们还需要在第2步对它进一步地细化。我们注意到，其中的计算阶段还可以进一步分解。即可以先计算总秒数（基于用户输入，很容易完成该计算），然后调用一个转换子程序，将总时间（秒）转换为“小时/分钟/秒”的格式。

但是对于C语言来说，第2步的部分描述还不够具体。所以。我们决定对其进行第3步细化。在第2步中，大多数内容已可以直接翻译为C代码了，只是“将秒转换为小时/分钟/秒”这个过程还需细化。所以，第3步的任务是将该子过程进一步细化为三个更小的子阶段。首先，我们计算总秒数对应的小时数，然后再计算剩余秒数对应的分钟数，最后是剩余秒数。

现在，如图12-3所示，整个问题在经历了三步细化之后，已足够简单，可以将它翻译成C代码了。最后的C程序如图12-4所示。

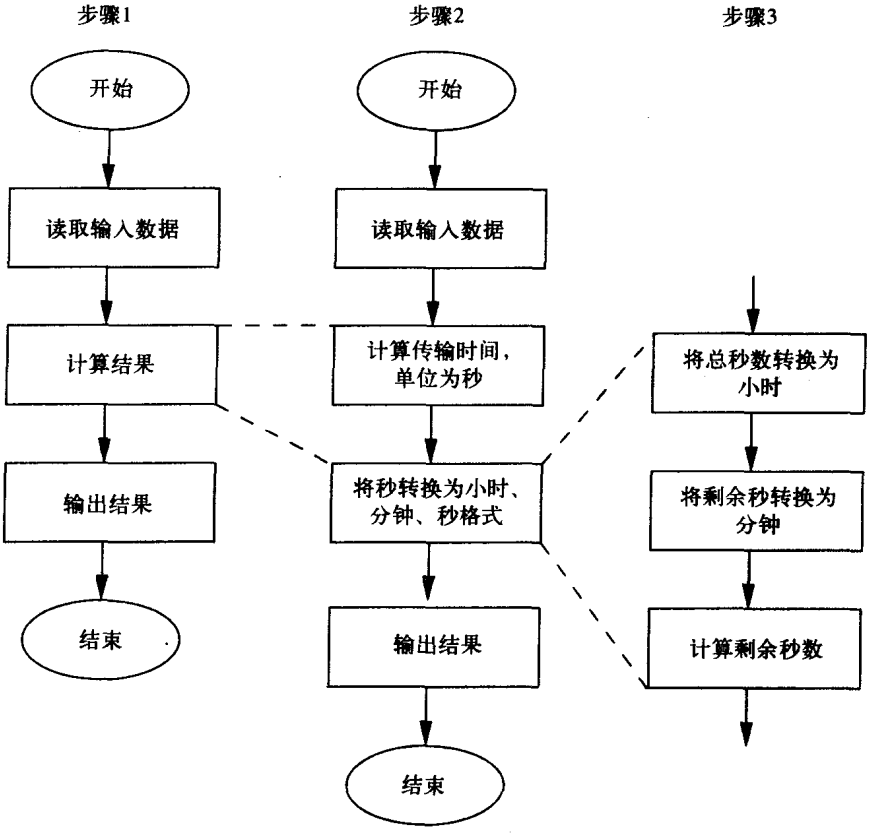


图12-3 计算网络传输时间问题的细化过程

```
#include <stdio.h>

int main()
{
  int amount;    /* The number of bytes to be transferred  */
  int rate;      /* The average network transfer rate      */
  int time;      /* The time, in seconds, for the transfer */

  int hours;     /* The number of hours for the transfer   */
  int minutes;   /* The number of mins for the transfer    */
  int seconds;   /* The number of secs for the transfer    */
```

图12-4 简单的网络速率计算C程序

```
    /* Get input: number of bytes and network transfer rate */
    printf("How many bytes of data to be transferred?  ");
    scanf("%d", &amount);

    printf("What is the transfer rate (in bytes/sec)?  ");
    scanf("%d", &rate);

    /* Calculate total time in seconds                    */
    time = amount / rate;

    /* Convert time into hours, minutes, seconds          */
    hours = time / 3600;          /* 3600 seconds in an hour */
    minutes = (time % 3600) / 60; /* 60 seconds in a minute */
    seconds = ((time % 3600) % 60); /* remainder is seconds */

    /* Output results */
    printf("Time : %dh %dm %ds\n", hours, minutes, seconds);
}
```

图12-4 简单的网络速率计算C程序（续）

12.5 编译器处理

至此，我们已介绍了本书将涉及的所有C基本类型和运算符。在介绍了C的第一组概念之后，我们计划从编译器的角度，来理解一下这些概念。换句话说，就是思考“编译器是怎样翻译C代码中的变量和运算符的？”。编译器在完成翻译工作的时候，需要借助于两个机制。一是符号表。编译器在编译过程中，将与变量相关的信息记录在“*符号表*”（symbol table）中；二是内存的系统分配。即根据变量特性分配内存。编译器将从整个系统层面，为不同类别的对象划分不同的内存区域。在本节中，我们将仔细研究这两个基本机制。

12.5.1 符号表

在第7章里，我们学习了汇编器基于“符号表”记录“标号”（label）的工作原理。与汇编器一样，C编译器也使用符号表来记录程序中遇到的变量。编译器在扫描程序代码的过程中，每遇到一个变量声明语句，就在符号表中为该变量创建一个新表项。表项结构中，将包含很多字段信息，这些字段信息与变量的存储空间管理及变量操作代码生成等都有关。这些字段包括：名字、类型、已分配的内存地址、变量声明域或作用域等信息。

图12-5是对应于图12-4的程序变量符号表。由于程序中包含了6个变量声明，所以编译器为之生成了有6个表项的符号表。注意，编译器采用的是偏移方式，记录变量在内存中的位置。也就是说，它记录的是变量在已分配的内存区域中的相对位置，且大多数偏移值都是负数。

标识	类型	位置（偏移量）	范围	其他信息
amount	**int**	0	**main**	...
hours	**int**	-3	**main**	...
minutes	**int**	-4	**main**	...
rate	**int**	-1	**main**	...
seconds	**int**	-5	**main**	...
time	**int**	-2	**main**	...

图12-5 编译器的符号表（对应图12-4的程序）

这个函数有6[⊖]个局部变量。帧指针R5指向第一个局部变量

12.5.2 变量的空间分配

在C语言中，存放变量内容的内存空间（变量的分配空间）有两种区段（region）：全局数据段（global data section）和运行时栈[⊜]（run-time stack）。全局数据段是内存中存放全局变量的区段，即所有静态类变量所在的地方（参见12.6节）；运行时栈则是局部变量（默认情况下为自动类变量）所在的地方。

符号表中的偏移字段（offset），提供了有关变量在内存中更精确的位置信息，它代表的是变量存储地址距离内存段基地址的偏移（或距离）。

例如，如果全局变量earth距离全局数据段起始地址（0x5000）的偏移量值为4，则变量earth的实际位置就是地址0x5004。在编译器生成的机器代码中，R4是一个专用寄存器，它存放的是全局数据段的基地址（或起始地址），所以R4可以被看做是一个全局指针。如果要将变量earth的内容读入寄存器R3，则LC-3指令如下：

```
LDR R3, R4, #4
```

与之前的情况相反，如果earth是一个局部变量（比如在main函数体内被声明），情况将变得有些复杂。一个函数中所有的局部变量都存放在一个被称做“活动记录”（activation record）或“堆栈帧”（stack frame）的内存模板（memory template）中。在此，我们只介绍该活动记录的结构，而对为什么是这样，暂不展开介绍（其中道理参见第14章）。所谓“活动记录”，是一段连续的内存空间，它包含了当前函数中所有的局部变量。每个函数有自己的活动记录（准确地说，应该是每个被调用的函数，都有一个自己的活动记录）。当我们调用一个函数时，该函数活动记录的最大地址将存放在寄存器R5中。因此，R5又被称做“帧指针”（frame pointer）。图12-6所示是图12-4中main函数的活动记录。注意，活动记录中，变量的排放顺序与它们在程序中的声明顺序是相反的。例如，变量amount在程序中是最先被声明的变量，而它也是距离“帧指针”R5最近的变量。

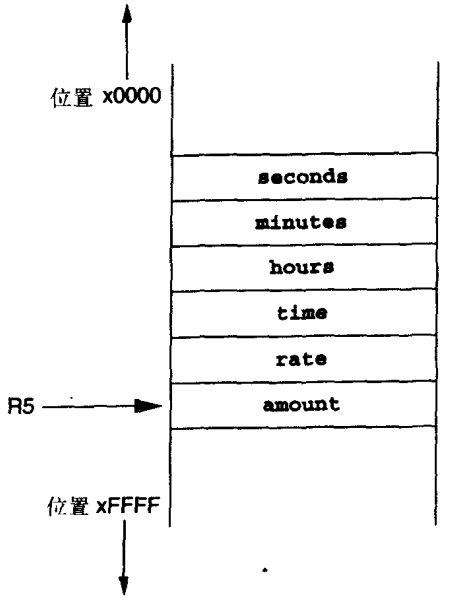


图12-6 LC-3内存中的一个活动记录

当局部变量在程序中被引用时，编译器将参照符号表，按照该变量对应表项的信息，生成合适的机器代码。这主要是因为，表项中的偏移字段能够告诉编译器，该变量在活动记录中的相对位置。例如，访问变量seconds，编译器将生成如下的指令：

```
LDR R0, R5, #-5
```

我们可以预见如下的场景：在C程序中，当我们调用一个函数时（在C语言中，“子程序”和“函数”是同一个概念），该函数的活动记录被“压入”（push）当前栈。同时，R5内容被调整，指向当前栈顶（即记录的基地址）。这意味着，我们可以通过栈指针

⊖ 原文此处为5，应该是个明显的笔误。——译者注

⊜ 在本书的所有例子中，变量都对应于一个内存地址。但在真实的编译器中，出于优化目的，通常会将一些变量分配到寄存器空间中。这是因为访问寄存器所花的时间比访问内存时要少得多，所以会将经常被访问的变量内容存放在寄存器里，从而加速程序的执行速度。

访问函数内的局部变量。当该函数结束，即将把控制权交还给调用者时，活动记录将被“弹出”(pop)当前栈。同样，指针R5的内容也将被修改，指向调用者活动记录所在位置。整个过程中，寄存器R6始终指向运行时栈的顶部，我们称R6为“栈指针”。有关阐述，参见第14章。

图12-7是程序运行时LC-3的内存布局结构。几乎所有的UNIX类操作系统，都采用这种内存空间结构。其中，程序代码占用一个区域；类似地，“运行时栈”和“全局数据段”也有各自的区域。另外，还有一个“堆空间”(heap)(我们将在第19章讨论它)，它存放的是动态分配的数据。栈和堆的大小在程序运行时是可以改变的。例如，当一个函数调用另一个函数时，栈空间会因为新的活动记录的入栈而变大（从大地址向地址x0000方向增长）。与之相反，堆的增长方向是地址0xFFFF。正是因为栈的增长方向是颠倒的（x0000方向），所以其中的活动记录结构也是反向的，即R5指向第一个局部变量，R5-1指向下一个，再之后是R5-2（参见12.5.1节中介绍符号表结构时提出的问题）。

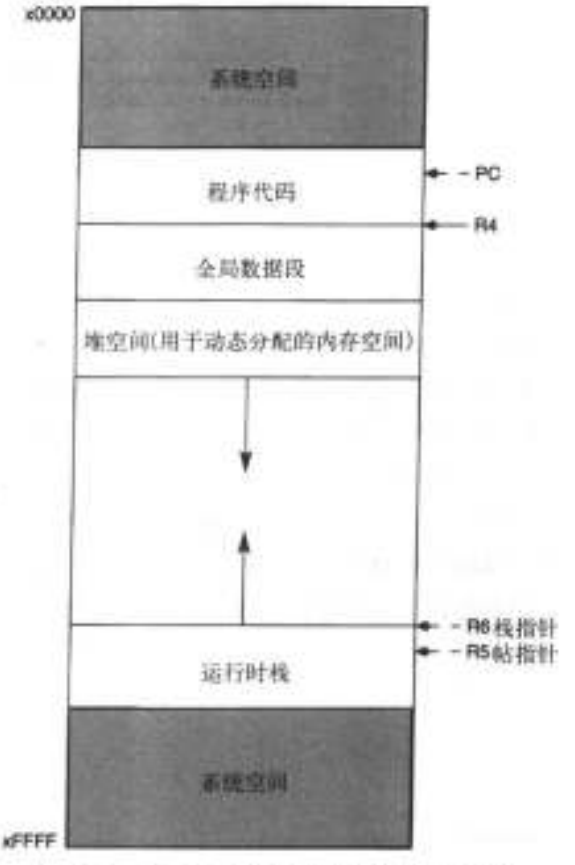


图12-7 LC-3程序执行过程中的内存结构图

在执行过程中，寄存器PC指向程序中当前执行代码，R4指向全局数据段的起始，R5指向栈空间内部，R6则指向栈空间顶部。此外，还存在一些专用内存区域。图12-7中的系统空间(system space)，是为操作系统预留的，其中包括TRAP服务程序、矢量表、I/O寄存器空间、系统引导代码等内容。

12.5.3 完整的例子

前面介绍了有关LC-3编译器的系统内存空间布局，以及局部变量的活动记录结构的处理方法。下面，我们将以一个完整的C程序为例，介绍它的编译过程，以及翻译之后的LC-3代码。

图12-8所示是一个执行了简单计算并将结果打印输出的C程序。该程序包含1个全局变量inGlobal，在main函数内，包含3个局部变量inLocal、outLocalA和outLocalB。

```
/* Include the standard I/O header file */
#include <stdio.h>

int inGlobal;     /* inGlobal is a global variable because */
                  /* it is declared outside of all blocks  */

int main()
{
  int inLocal;    /* inLocal, outLocalA, outLocalB are all */
  int outLocalA; /* local to main                          */
  int outLocalB;
```

图12-8 一个执行简单操作的C程序

```
    /* Initialize */
    inLocal  = 5;
    inGlobal = 3;

    /* Perform calculations */
    outLocalA = inLocal & ~inGlobal;
    outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

    /* Print out results */
    printf("outLocalA = %d,outLocalB=%d\n",outLocalA,outLocalB);
}
```

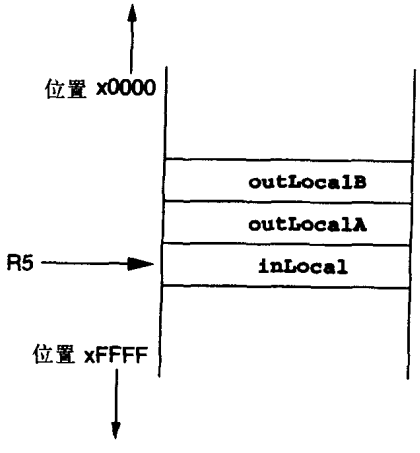
图12-8 一个执行简单操作的C程序（续）

程序的开始，是对inLocal和inGlobal的初始化；之后，执行变量inLocal和inGlobal之间的运算，并将计算结果分别放入变量outLocalA和outLocalB；最后，调用标准库提供的printf函数，将计算结果打印输出（即outLocalA和outLocalB的值）。注意，由于我们在此使用了printf函数，因此必须在程序开始时包含（include）标准I/O库的头文件——stdio.h。

在分析代码时，LC-3 C编译器将全局数据段中的第一个位置（即偏移量为0的空间）分配给变量inGlobal。当分析过程进入main函数体内部时，编译器将main函数活动记录区的偏移0的位置分配给局部变量inLocalA，将偏移-1的位置分配给outLocalA，将偏移-2的位置分配给outLocalB。图12-9所示是编译器的符号表和main函数活动记录区的一个快照。

标识	类型	位置(偏移量)	范围	其他信息
inLocal	int	0	main	...
inGlobal	int	0	global	...
outLocalA	int	-1	main	...
outLocalB	int	-2	main	...

a) 符号表



b) 函数的活动记录

图12-9 图12-8所示程序的编译器符号表和main函数的活动记录

图12-10所示是LC-3 C编译器产生的汇编代码。该程序从标识main处开始执行。

```
 1  main:
 2    :
 3    :
 4    <startup code>
 5    :
 6    :
 7    AND  R0, R0, #0
 8    ADD  R0, R0, #5   ;  inLocal is at offset 0
 9    STR  R0, R5, #0   ;  inLocal = 5;
10
11    AND  R0, R0, #0
12    ADD  R0, R0, #3   ;  inGlobal is at offset 0, in globals
```

图12-10 对应图12-8所示C程序的LC-3代码

```
13    STR  R0, R4, #0  ;  inGlobal = 3;
14
15    LDR  R0, R5, #0  ;  get value of inLocal
16    LDR  R1, R4, #0  ;  get value of inGlobal
17    NOT  R1, R1      ;  ~inGlobal
18    AND  R2, R0, R1  ;  calculate inLocal & ~inGlobal
19    STR  R2, R5, #-1 ;  outLocalA = inLocal & ~inGlobal;
20                     ;  outLocalA is at offset -1
21
22    LDR  R0, R5, #0  ;  get value of inLocal
23    LDR  R1, R4, #0  ;  get value of inGlobal
24    ADD  R0, R0, R1  ;  calculate inLocal + inGlobal
25
26    LDR  R2, R5, #0  ;  get value of inLocal
27    LDR  R3, R4, #0  ;  get value of inGlobal
28    NOT  R3
29    ADD  R3, R3, #1  ;  calculate -inGlobal
30
31    ADD  R2, R2, R3  ;  calculate inLocal - inGlobal
32    NOT  R2
33    ADD  R2, R2, #1  ;  calculate -(inLocal - inGlobal)
34
35    ADD  R0, R0, R2  ;  (inLocal + inGlobal) - (inLocal - inGlobal)
36    STR  R0, R5, #-2 ;  outLocalB = ...
37                     ;  outLocalB is at offset -2
38    :
39    :
40    <code for calling the function printf>
41    :
42    :
```

图12-10 对应图12-8所示C程序的LC-3代码（续）

12.6 补充话题

这是本章的最后一节，将对变量和运算符做一些补充论述。这些内容只是本章之前内容的延伸，是有关C语言的琐碎细节。它们与本书其他内容没有直接关联，只是对C语言知识的一个涵盖，如果你对C语言的变量和运算符确有兴趣，那么接着读吧！

12.6.1 三种基本类型的变种

在C语言中，程序员可以定制三种基本类型int、char、double的大小。例如，在int前面加上前缀long和short，将在int默认大小的基础上扩大和缩小其大小。如“long int”表示的bit宽度是常规int的两倍。基于这种机制，我们就可以在C程序中表示更大范围的整数。同样，double也可以加上前缀long，它将扩展浮点类型的表示范围和精度（需要系统支持）。

修饰词short的作用则是减小变量类型的默认大小。我们可以借用它节省内存空间。下面是使用修饰符long和short声明变量的例子：

```
long double particlesInUniverse;
long int worldPopulation;
short int ageOfStudent;
```

C语言的三种基本类型与底层指令集结构（ISA）密切相关。因此，许多编译器只在底层指令集结构能支持的情况下，才能真正支持修饰词long和short。也就是说，如果底层指令集结构不支持长整数类型，即使将变量声明为“long int”，它实际上和int没有区别。附录D.3.2是有关long和short更多的例子和说明。

int类型的又一变种是“无符号整数”（unsigned integer），它的修饰词是“unsigned”。所谓“无符号整数”，就是在所有bit中不存在符号位，能表达的数也都是非负整数（正数和0）。例如，

在LC-3中，整数的宽度是16-bit，则无符号整数的表示范围为0~65535。当处理一些自然界的对象时，如果其值不为负，则可以使用无符号整数类型。下面是无符号整数的例子：

```
unsigned int numberOfDays;
unsigned int populationSize;
```

以下是三种基本类型的各种变种形式：

```
long int ounces;
short int gallons;
long double veryVeryLargeNumber = 4.12936E361;
unsigned int sizeOfClass = 900;
float oType = 9.24;
float tonsOfGrain = 2.998E8;
```

12.6.2 文字常量、常量和符号值

在C语言中，若一个变量的前缀为const，就表示该变量内容是“常量”。所谓常量，是指其内容在程序执行过程中不可修改。例如，计算圆的面积和周长时，需要一个值为3.14159的浮点常量。图12-11所示就是这样一个程序。

在这个例子中，我们可以学习和区分C语言的三种常量。一是“文字常量”，代表代码中那些没有名字的数值。如本例中的数值2和3.14159，都属于文字常量。注意，C语言中的十六进制文字常量必需前缀0x（如0x1DB），而ASCII文字需用单引号包围（如“R”，它代表的是字母R的ASCII值），浮点文字的表示方式则参见12.2.1节；二是前缀修饰符const的变量（有名字）方式，如程序中的常量变量pi；三是预处理指令#define方式，例如符号数值RADIUS。所有这三种类型的常量值，在整个程序生命期内都不会改变。

```
 1  #include <stdio.h>
 2
 3  #define RADIUS  15.0     /* This value is in centimeters */
 4
 5  int main()
 6  {
 7    const double pi = 3.14159;
 8    double area;
 9    double circumference;
10
11    /* Calculations */
12    area = pi * RADIUS * RADIUS;          /* area = pi*r^2 */
13
14    circumference = 2 * pi * RADIUS;    /* circumference = */
15                                        /*          2*pi*r */
16
17    printf("Area of a circle with radius %f cm is %f cm^2\n",
18           RADIUS, area);
19
20    printf("Circumference of the circle is %f cm\n",
21          circumference);
22  }
```

图12-11 计算圆面积和周长（半径为15cm）的C程序

前缀const方法和#define定义符号数值方法之间，没有本质区别，至于使用哪种方式，只是编程风格的问题。总之，如果我们认为某个东西，它的内容或数值不会改变，则可以将其定义为常量。例如，圆周率常量pi。另外，光速、一星期的天数，都可以表示为常量。

还有一种应用场景，如果一个数值在同一次程序执行过程中是不会改变的，但会随着用户或场合的变化而变化，则可以使用#define方式。我们可以将这种方式定义的常量，理解为是程序参数（每次运行可改变的）。例如，图12-11中RADIUS的定义就是这种方式。每次改变之后，需要重

新编译，然后再执行。

通常，命名常量的方式（如const或#define方式），比直接使用文字常量要受欢迎一些，因为我们可以通过名字传递更多信息（而不仅仅是常量数值本身）。

12.6.3 存储类型

前面介绍了C变量的三个基本属性：标识符、类型和作用域。另外，变量还有一个属性——*存储类*（storage class）。所谓“存储类”，指的是C编译器为变量分配存储空间的方式，以及在代码块执行结束时，该变量内容是否会丢失的问题。C语言支持*静态*（static）和*自动*（automatic）两种存储类。静态变量的内容，在两次启用之间不会改变，而自动变量的内容在代码块结束后就丢失。例如，在C语言中，全局变量就属于静态存储类，即它们的内容将在整个程序运行期内都一直保持；在默认情况下，局部变量属于自动存储类。但如果局部变量在声明时，加上前缀修饰符static，则它就属于静态类。如“static int localVar;”声明语句，说明变量localVar的内容，在函数完成后一直保持。也就是说，如果该函数再次被调用，localVar的内容仍然为上次运行结束时的值。对于加了前缀static的局部变量，编译器将在全局数据段中为它分配空间，但可访问范围仍然限制在当前代码块内。附录D3.3是有关存储类的更多的例子。

12.6.4 更多的C运算符

在C语言中，还包含一些非常规运算符。事实上，它们几乎就是C语言的标志性特征。这些运算符，大多是在基本运算符的基础上组合而成的，但它们的存在使得表达式看起来更简单。对于不熟悉组合运算符的人来说，阅读理解它们确实是件费力的事情。

1. *赋值运算符*

C语言允许某些算术或位运算符与赋值运算符相组合。例如，我们对变量x本身增量29，则可以使用如下的运算符“+=”：

```
x += 29;
```

该语句与如下语句等价。

```
x = x + 29;
```

表12-6所示是C语言具有的一些特殊运算符。其中，后缀运算符的优先级最高，其次是前缀运算符，赋值运算符的优先级最低。关联顺序都是自右向左。

表12-6 C语言的赋值运算符

运算符	操　作	示　例
+=	加后赋值	x += y
−=	减后赋值	x −= y
*=	乘后赋值	x *= y
/=	除后赋值	x /= y
%=	取余后赋值	x %= y
&=	与后赋值	x &= y
|=	或后赋值	x |= y
^=	异或后赋值	x ^= y
<<=	左移后赋值	x <<= y
>>=	右移后赋值	x >>= y

例如：

```
h += g;        /* Equivalent to h = h + g;   */
h %= f;        /* Equivalent to h = h % f;   */
h <<= 3;       /* Equivalent to h = h << 3;  */
```

2. 条件表达式

条件表达式是C语言特有的，它可以在一个表达式中，同时完成判断和赋值这两个操作。条件表达式中的关键符号是问号“？”和冒号“:”。如下所示：

```
x = a ? b : c;
```

该语句的意思是，x的取值可能是b，也可能是c，具体选择取决于表达式a的逻辑值。如果a为非零值（即逻辑真），则x等于b；否则，x等于c。

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5     int maxValue;
 6     int input1;
 7     int input2;
 8
 9     printf("Input an integer: ");
10     scanf("%d", &input1);
11     printf("Input another integer: ");
12     scanf("%d", &input2);
13
14     maxValue = (input1 > input2) ? input1 : input2;
15     printf("The larger number is %d\n", maxValue);
16  }
```

图12-12 C程序中的条件表达式

图12-12所示是一个使用条件表达式求两数中最大值的程序。两个输入中，最大的数值将通过条件表达式赋给变量maxValue，最后由printf打印输出maxValue的内容。

12.7 小结

本章主要讲述了以下三方面内容：

- C变量。C语言支持三种基本变量类型：整型、字符型和浮点型。同其他高级语言一样，C语言允许程序员为变量提供符号名。C变量的声明，可以在某个代码块（如函数）内，也可以是全局可见的。
- C运算符。C的运算符，按照它们的行为和功能来分，包括赋值、算术、位运算、逻辑和关系测试等几类操作。表达式是变量和运算符的组合，表达式的推算过程遵循优先级和关联顺序规则。语句是一个或多个表达式的组合，它表达的是程序要执行的任务。
- C变量和运算符的编译处理（LC-3代码）。在编译过程中，编译器将被声明的变量记录在“变量符号表”中。基于这个符号表，编译器为特定函数内的局部变量分配“活动记录”空间。当该函数被调用时，该函数的活动记录被压入栈区段。程序中，全局变量的分配空间在全局数据段中。

12.8 习题

12.1 写出如下代码的编译器变量符号表。假设其中每个变量占用一个独立的内存空间。

```
{
   double ff;
   char cc;
   int ii;
   char dd;
}
```

12.2 以下是一个变量声明语句：

```
int r;
```

试问：

a. 如果r是局部变量，它的初始值为多少？

b. 如果r是全局变量，它的初始值为多少？

12.3 如果下面两个变量的存储空间都是32-bit，试问它们的数值范围是多少？

```
int plusOrMinus;
unsigned int positive;
```

12.4 请写出如下浮点常数的十进制数值。

a. 111E−11

b. −0.00021E4

c. 101.101E0

12.5 如果使用LC-3 C编译器，请写出如下局部变量声明语句所对应的LC-3汇编代码。

```
char c = 'a';
int  x = 3;
int  y;
int  z = 10;
```

12.6 写出如下代码中，每个printf语句的输出结果。语句执行顺序是A、B、C、D。

```
int t;  /* This variable is global */

{
   int t = 2;

   printf("%d\n", t);      /*  A  */
   {
      printf("%d\n", t);   /*  B  */
      t = 3;
   }
   printf("%d\n", t);      /*  C  */
}

{
   printf("%d\n", t);      /*  D  */
}
```

12.7 已知变量a和b都是正数，它们的值分别为6和9。试问如下表达式的结果是什么？如果a或b的值改变了，请写出它们的新值。

a. `a | b`	b. `a || b`	c. `a & b`	d. `a && b`
e. `!(a + b)`	f. `a % b`	g. `b / a`	h. `a = b`
i. `a = b = 5`	j. `++a + b--`	k. `a = (++b < 3) ? a : b`	l. `a <<= b`

12.8 有关如下字符变量letter的关系测试问题，写出对应的C语言表达式。

a. 测试letter是否为字母或数字；

b. 测试letter是否包含字母或数字之外的字符。

12.9 试回答以下问题：

a. 如下语句的执行结果是什么？其中变量letter是一个字符变量；

```
letter = ((letter >= 'a' && letter <= 'z') ? '!' : letter);
```

b. 修改（a）中的语句，以使得新语句能够将小写字母转换为大写字母。

12.10 试编写程序，从键盘读入整数并进行判断，如果能被3整除，则显示输出“1”；否则，显示输出“0”。

12.11 试解释以下各C语句之间的不同：

a. `j = i++;`　　b. `j = ++i;`　　c. `j = i + 1;`

```
d.i += 1;      e.j = i += 1;
```

f. 以上语句中，谁修改了i的值？谁修改了j的值？如果初始值i = 1，j = 0，试问以上每条语句执行之后，i和j的值是什么？

12.12 假设变量a和b都被声明为“long int”类型的局部变量，试问：

a. 写出表达式“a + b”的LC-3汇编代码。其中，假设“long int”类型占用空间是两个字节，a对应函数活动记录的偏移0位置，b对应活动记录的偏移−1位置；

b. 如问题（a）所示，假设类型“long int”占用4个字节，a为偏移0，b为偏移−2。请写出对应的LC-3汇编代码。

12.13 假设初始情况是：a = 1、b = 1、c = 3、result = 999。试问，执行如下语句之后，各变量的值分别为多少？

```
result = b + 1 | c + a;
```

12.14 回顾第2章中机器“忙”的例子。其中，变量machineBusy记录的是16台机器的忙碌状态。对应bit的值如果为0，代表该机器正在“忙碌”；如果为1，则代表该机器“空闲”。

a. 编写C语句，设置5号机器“忙碌”；

b. 编写C语句，设置10号机器“空闲”；

c. 编写C语句，设置第n号机器“忙碌”。其中，n是一个整型变量。

d. 编写C表达式，检验3号机器是否“空闲”。如果“空闲”，则表达式结果为1，如果“忙碌”，则表达式结果为0。

e. 编写C表达式，计算空闲机器的总数。例如，假设machineBusy的二进制值为1011 0010 1110 1001，则表达式结果应该为9。

12.15 试问，在C语言中，分号的作用是什么？

12.16 假设我们为计算机设计一个新的编程语言，其中包含@、#、$、U等运算符。试问，在如下条件下，表达式“w @ x # y $ z U a”的推算顺序分别是什么？

a. 假设运算符优先级顺序（从高到低）为：@、#、$、U。提示：采用括号表达计算顺序，如（(w@x) ...）；

b. 假设运算符优先级顺序（从高到低）为：#、U、@、$；

c. 假设它们的优先级都相等，关联顺序为自左向右；

d. 假设它们的优先级都相等，关联顺序为自右向左。

12.17 我们都知道，在C语言中，赋值运算符的优先级最低。假设我们设计了一种新的编程语言——*Q*语言。它与C语言工作原理相同。只是在Q语言中，赋值运算符的优先级最高。试问：

a. 如下Q语句的执行结果如何？换句话说，该语句执行之后，x的值是多少？

```
x = x + 1;
```

b. 怎样修改该Q语句，以使得运算结果和对应的C语句结果一样。

12.18 试对第11章的程序（如图11-3所示）做出修改：提示用户输入字符，然后按照ASCII表的顺序，从该字符起，逐个字符打印，直到字符“！”结束。

12.19 试编写一个“交易税计算”程序：提示用户输入“采购金额”（purchase amount）和“税率”（tax rate）；然后，打印输出“交易税”（sales tax）和“交易总额（包括税款）”（提示：交易总额 = 采购金额 + 交易税）。

12.20 假设程序含有两个整型变量x和y，且它们的数值分别为3和4。试编写C语句，交换x和y的值，即语句执行之后，x = 4，y = 3。

a. 第一种方式，允许使用中间变量，写出这个程序；

b. 第二种方式，不允许使用中间变量，写出相应的程序。

第13章 控制结构

13.1 概述

在第6章中，介绍了“自顶向下”方法学，经过系统的细化，一个大问题被分解为多个子任务，然后对每个子任务独立编程。各种子任务的基本编程结构（construct）可以分为顺序、条件和循环三种等。

在前一章的“网络传输时间”问题求解中，我们就采用了这种系统分解方法，只是在那个例子中，我们只用到了顺序结构。如果要求解更复杂的问题，我们还需要掌握另外两种C语言结构的编程技术，即条件结构和循环结构。

本章的第一个内容是C语言的“条件结构”，我们称if和if-else语句为“条件执行语句”。之后，是C语言的“循环结构”，如for、while和do-while语句，它们都可以表达循环操作。随后，我们将介绍LC-3 C编译器怎样为这些语句生成对应的LC-3代码，即这些C结构的底层行为。此外，C语言还提供了诸如switch、break和continue等控制结构，但所有这些都只是为实现特定控制任务提供了更直接的描述方式（参见13.5节）。最后，我们将演示，如何通过“自顶向下”方法，以及这些控制结构，解决一个复杂问题。

13.2 条件结构

条件结构的作用，是使得程序员可以基于“特定条件”选择程序的“动作”。这是一种非常有用的编程结构，几乎每类编程语言都支持这种结构。C语言为此提供了两种基本条件结构：if和if-else。

13.2.1 if语句

if语句非常简单。它的语义是，如果条件为“真”，则执行特定动作。所谓“动作”（action），是一个C语句，当且仅当条件（C表达式）为真时，才执行该“动作”语句。下面，我们以例子来说明它的使用方法。

```
if (x <= 10)
   y = x * x + 5;
```

其中，当且仅当表达式“x <= 10”为真时，语句“y = x * x + 5”才被执行。回顾一下有关运算符“<=”的讨论，它是一个关系运算符。当关系成立时，表示式值为1；否则，表达式值为0。

条件表达式之后的语句，可以是一条单独语句，也可以是一个复合语句（或代码块，即括号{}所包围的代码序列）。所谓复合语句，是由一个或多个简单语句组合而成的单独实体，该实体在程序结构上等价于一个简单语句。基于复合语句，我们可以只做一次条件判断，而连续执行多个语句。例如，如下代码中，如果x小于等于10，则y和z都会被改变。

```
if (x <= 10) {
   y = x * x + 5;
   z = (2 * y) / 3;
}
```

在C语言中，语句的使用格式非常灵活，if语句也不例外。如前面这个例子，分行和缩进就是if语句常用的格式风格。这种格式的特点是，代码阅读者可以快速区分出，哪些语句是条件为真时

要被执行的，哪些语句是条件为假时要被执行的。值得一提的是，格式上的差异并不会影响程序的行为behavior。但是，下面这段代码，虽然与上面代码非常相似（如缩进），但执行时的行为却完全不同。因为，其中的第二个语句“z = (2 * y) / 3;”与if语句之前不存在任何关联，即无论条件是否为真，该语句都被执行。

```
if (x <= 10)
   y = x * x + 5;
   z = (2 * y) / 3;
```

图13-1所示是if语句对应的控制流程图。该图与如下代码形式等价：

```
if (condition)
   action;
```

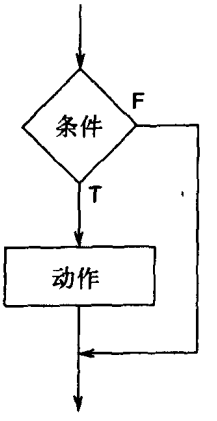


图13-1 C语言里if语句的控制流程图

从语法上来讲，条件表达式必须用圆括号包围，只有这样，编译器才能明确哪部分代码属于条件表达式；而动作部分代表的是一个简单语句或复合语句。

下面是一个if语句的例子，请体会判断（或条件）结构在编程中的作用。

```
if (temperature <= 0)
   printf("At or below freezing point.\n");

if ('a' <= key && key <= 'z')
   numLowerCase++;

if (current > currentLimit)
   blownFuse = 1;

if (loadMAR & clock)
   registerMAR = bus;

if (month == 4 || month == 6 || month == 9 || month == 11)
   printf("The month has 30 days\n");

if (x = 2)    /* This condition is always true.    */
   y = 5;     /* The variable y will always be 5   */
```

在最后一段代码中，存在一个编程错误。这是一个常见的C语言编程错误，甚至很多高级程序员也难免有时会犯。当然，功能强大的C编译器能够检测到类似错误，并输出警告提示。这个错误是，条件表达式中使用的是赋值运算符（=），而不是关系运算符（==）。这样的执行结果是，变量x被赋值为2，且从条件计算角度来说，该条件永远为真（因为，如果表达式是一个赋值运算，则表达式值即为该数值，此处为2）。由于条件永远为真，所以事实上的执行结果是：变量y总被赋为5，x则总被赋为2。

如下是修复后的代码，它与之前的代码非常相似。

```
if (x == 2)
   y = 5;
```

下面是与之对应的LC-3代码。我们的假设是，x和y都是整型变量，且都是局部变量。即R5指向的是变量x，R5-1指向的是变量y。

```
          LDR  R0, R5, #0  ; load x into R0
          ADD  R0, R0, #-2 ; subtract 2 from x
          BRnp NOT_TRUE    ; If condition is not true,
                           ; then skip the assignment

          AND  R0, R0, #0  ; R0 <- 0
          ADD  R0, R0, #5  ; R0 <- 5
          STR  R0, R5, #-1 ; y = 5;

NOT_TRUE  :                ; the rest of the program
          :
```

注意，LC-3汇编语言中，测试的是与原条件相反的条件（即“x不等于2”），然后根据这个反

条件的值，决定是否跳转。对于C编译器来说，这是代码效率最高的处理方法。

值得一提的是，if语句中的动作，也可以是又一个if语句。换句话说，C语言支持“嵌套if”格式。如下面C代码所示，此处由于第一个if之后的if语句属于简单语句，所以第一个if之后不需要大括号包围。

```
if (x == 3)
   if (y != 6) {
      z = z + 1;
      w = w + 2;
   }
```

内层的if语句，当且仅当x等于3时才被执行。这段代码还有一种更简单的表达方法。能否只使用一个if语句呢？参见如下代码。

```
if ((x == 3) && (y != 6)) {
      z = z + 1;
      w = w + 2;
   }
```

13.2.2 if-else语句

如果希望条件为真时执行某个动作，而条件为假时执行另一个动作。那么，它的if语句表达方式如下：

```
if (temperature <= 0)
   printf("At or below freezing point.\n");

if (temperature > 0)
   printf("Above freezing.\n");
```

其中，变量temperature的值或小于等于0，或大于0。与两种情况对应，打印的两个消息也不同。这种条件执行场景（或需求），在实际编程中非常普遍。但前面这种表达方式，看起来有些烦琐。为此，C语言提供了另一种结构：if-else语句。

如下代码的执行效果，与前面的代码等价：

```
if (temperature <= 0)
   printf("At or below freezing point.\n");
else
   printf("Above freezing.\n");
```

其中，当且仅当条件值为假时，关键词else后面的语句才被执行。

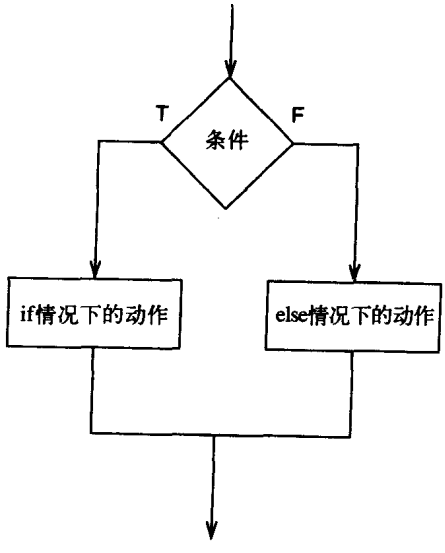


图13-2 C语言里if-else语句的控制流程图

if-else语句对应的控制流程图，如图13-2所示。与之对应的代码结构如下所示：

```
if (condition)
   action_if;
else
   action_else;
```

其中，action_if和action_else所代表的可以是一个复合语句，如下面例子所示：

```
if (x) {
   y++;
   z--;
}
else {
   y--;
   z++;
}
```

其中，如果变量x为非零，即if条件为真，则执行y递增和z递减；否则，执行y递减和z递增。与之对应的LC-3的代码，如图13-3所示。假设条件是，变量x、y和z都是局部声明的整型变量。

```
1          LDR  R0, R5, #0   ;   load the value of x
2          BRz  ELSE         ;   if x equals 0, perform else part
3
4          LDR  R0, R5, #-1  ;   load y into R0
5          ADD  R0, R0, #1
6          STR  R0, R5, #-1  ;   y++;
7
8          LDR  R0, R5, #-2  ;   load z into R0
9          ADD  R0, R0, #-1
10         STR  R0, R5, #-2  ;   z--;
11         BR DONE
12
13  ELSE:  LDR  R0, R5, #-1  ;   load y into R0
14         ADD  R0, R0, #-1
15         STR  R0, R5, #-1  ;   y--;
16
17         LDR  R0, R5, #-2  ;   load z into R0
18         ADD  R0, R0, #1
19         STR  R0, R5, #-2  ;   z++;
20  DONE:  :
21         :
```

图13-3 if-else语句对应的LC-3代码

我们还可以将多个条件结构组合，形成更长的条件测试序列。图13-4所示是由一系列if和if-else语句串联在一起，而组成的复杂条件判断结构（其中没有使用任何其他控制结构）。程序的功能是，读入用户输入的月份数，然后打印输出该月的天数。

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int month;
6
7    printf("Enter the number of the month: ");
8    scanf("%d", &month);
9
10   if (month == 4 || month == 6 || month == 9 || month == 11)
11     printf("The month has 30 days\n");
12   else if (month == 1 || month == 3 || month == 5  ||
13            month == 7 || month == 8 || month == 10 || month == 12)
14     printf("The month has 31 days\n");
15   else if (month == 2)
16     printf("The month has either 28 days or 29 days\n");
17   else
18     printf("Don't know that month\n");
19 }
```

图13-4 输出一个月的天数的程序

在此，我们要强调一下C语言的if和else关联规则：在嵌套（nest）或串联方式下，else总是与它距离最近的if发生关联。以如下代码为例，我们来看一下该语法规则的重要性。

```
if (x != 10)
   if (y > 3)
      z = z / 2;
   else
      z = z * 2;
```

如果没有规则，我们将无法确定代码中的else是与外层的if配对，还是与内层的if配对。注意，两种假设下，执行效果是完全不一样的。

按照规则，这个else应该与内层的if配对，因为它与else之间的距离最短。所以，根据规则，上面代码与如下代码（注意括号的包围）是等价的：

```
if (x != 10) {
   if (y > 3)
      z = z / 2;
   else
      z = z * 2;
}
```

与表达式中的圆括号作用相似（改变运算符关联顺序），大括号在此的作用是改变语句关联的顺序。再假设，如果我们希望该else与外层if相关联，则大括号的用法如下所示：

```
if (x != 10) {
   if (y > 3)
      z = z / 2;
}
else
  z = z * 2;
```

最后，在结束if-else语句之前，我们介绍一个if-else结构在编程中的应用技巧：使用if-else语句，检测程序中存在的错误。如图13-5所示，程序从键盘读入两个数值，然后两者相除。由于除数为0是“非法的”，所以如果检测到用户输入的除数为0，则打印错误提示消息。在此，我们发现if-else语句很适合于该任务。

值得一提的是，在错误检测的例子中，正确的情况（或动作）通常出现在if-else结构的if部分，而错误情况出现在else部分。当然，它们的先后顺序完全可以互换，即先是错误情况再是正确情况。但是，从阅读习惯上来说，人们更习惯于假设先看到的是正常情况（因为错误情况出现概率非常小）。

13.3 循环结构

能够循环（或重复）执行一个计算任务，是计算系统的一个重要能力。几乎所有的程序都含有某种形式的循环。C语言支持三种循环结构，它们之间存在一些细微差别，即while、for和do-while三种语句。

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5    int dividend;
 6    int divisor;
 7    int result;
 8
 9    printf("Enter the dividend: ");
10    scanf("%d", &dividend);
11
12    printf("Enter the divisor: ");
13    scanf("%d", &divisor);
14
15    if (divisor != 0) {
16      result = dividend / divisor;
17      printf("The result of the division is %d\n", result);
18    }
19    else
20      printf("A divisor of zero is not allowed\n");
21  }
```

图13-5 有错误检查代码的程序

13.3.1 while语句

C语言中最简单的循环语句是while语句。它的含义是：“只要”（while）条件为真，则重新执行某条语句。”具体地说，就是每次迭代之前，都再次检查条件。如果条件为逻辑真（非零值），则该语句就再次被执行。

例如，当变量x小于10时，则循环体代码重复执行。该程序的输出结果如下所示：

```
0 1 2 3 4 5 6 7 8 9
```

```
#include <stdio.h>

int main()
{
   int x = 0;

   while (x < 10) {
      printf("%d ", x);
      x = x + 1;
   }
}
```

我们将while语句划分为两部分。一是产生条件结果的表达式，如test，它决定了循环是否需要继续。每次循环体（loop_body）被执行之前，它都将被测试。二是循环体（loop_body），它代表每次循环所要完成的工作。同样，它也可以是一个复合语句。

```
while (test)
   loop_body;
```

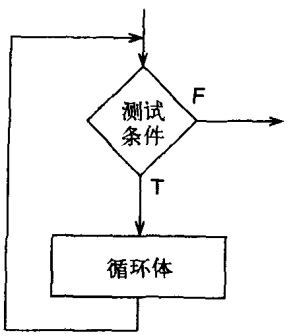


图13-6 while语句的控制流程图

图13-6所示是“系统分解法”中标识while语句的控制流程图。图中存在两个分支：一个退出循环体（条件分支），另一个跳回条件测试语句test（无条件跳转）。

图13-7所示是对应于循环输出0~9的while程序，由编译器生成的LC-3代码。

```
1               AND  R0, R0, #0    ;   clear out R0
2               STR  R0, R5, #0    ;   x = 0;
3
4               ; while (x < 10)
5    LOOP:      LDR  R0, R5, #0    ;   perform the test
6               ADD  R0, R0, #-10
7               BRpz DONE          ;   x is not less than 10
8
9               ; loop body
10              :
11              <code for calling the function printf>
12              :
13              LDR  R0, R5, #0    ;   R0 <- x
14              ADD  R0, R0, #1    ;   x + 1
15              STR  R0, R5, #0    ;   x = x + 1;
16              BR   LOOP          ;   another iteration
17   DONE:      :
18              :
```

图13-7 从1数到9的while循环对应的LC-3代码

在“哨兵（sentinel）条件测试”类的循环中，while语句非常合适。所谓“哨兵条件”，是指循环次数事先是不确定的，程序将无限循环，直到发生某个“事件”（即出现“哨兵”）为止。例如，第5章和第7章的字符统计程序，计数过程持续循环，直到遇到字符EOT（End-Of-Text，ASCII码值为4）为止。如果用C语言（而不是LC-3汇编语言）编写该程序，我们就可以使用while语句，如图13-8所示。请在不执行这个程序的情况下，猜测该程序的运行结果。⊖

在结束while话题之前，我们再介绍一个使用while时常犯的错误。如下程序将永远不会终止，因为循环条件始终没有改变。且该例中，条件总为真，所以循环不会终止。我们称这种循环为“死循环”（infinite loop），且这类错误的原因都是源于程序员。

⊖ 提示：该程序的运行结果和你想像的可能不同。也许你认为，当用户输入字符之后，它会马上被打印出来。但实际上，C语言处理键盘I/O的方法是，如果用户不按回车键，则程序读不到任何输入。我们将在第18章讲述底层I/O时，再解释其原因。

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char echo = 'A';   /* Initialize char variable echo */
6
7    while (echo != '\n') {
8      scanf("%c", &echo);
9      printf("%c", echo);
10   }
11 }
```

图13-8 另一个使用简单while循环的程序

```
#include <stdio.h>

int main()
{
   int x = 0;

   while (x < 10)
      printf("%d ", x);
}
```

13.3.2 for语句

如果说while循环适合于“哨兵条件”控制的循环，则C语言的for语句则非常适合于“计数器”控制的循环。换句话说，for循环是while循环在事先知道循环次数的情况下的一个特例。

for语句最常见的使用形式，是重复执行一个语句，且次数固定。如下所示：

```
#include <stdio.h>

int main()
{
   int x;

   for (x = 0; x < 10; x++)
      printf("%d ", x);
}
```

该程序循环执行10次，运行输出的结果如下：

```
0 1 2 3 4 5 6 7 8 9
```

for语句的语法结构，初看起来让人感觉有些烦琐。for语句包括4个组成部分，如下所示：

```
for (init; test; reinit)
   loop_body;
```

括号中的三个部分是init、test和reinit，它们控制着循环的动作，且之间用分号隔开。最后一个部分是loop_body，代表每次循环中执行的运算。

下面详细解释for循环结构的4个部分：

- init（初始化）：是在第一次循环前要被执行的表达式。通常，它被用做循环初始化。
- test（条件测试）：是每次循环都要被执行的判断表达式，判断循环是否还需要继续。如果test表达式的值为零，则for循环终止，控制权转移至for下面的语句；相反，如果表达式值非零，则继续下一轮loop_body的执行。例如，前面例子中，表达式“x < 10”表示的是“只要x小于10，循环就继续”。
- reinit（重新初始化）：是每次循环结束之后，都要被执行的表达式。它表示为下一次循环所做的准备（所以又称为“reinitialize”）。如前例中，每次下一循环之前，变量x都先被递增1。

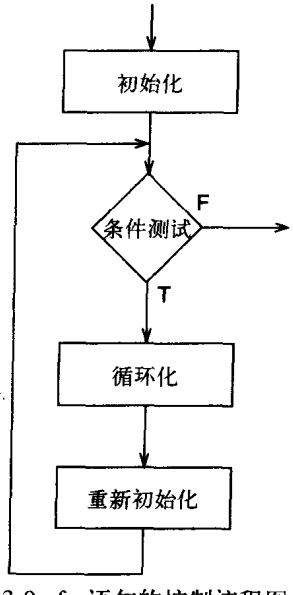


图13-9 for语句的控制流程图

• loop_body（循环体）：是每次循环都要被执行的语句。同样，它可以是一个复合语句。

图13-9所示是for语句的控制流程图。该图中包含4个模块，分别对应for语句的4个部分。其中，test表达式是一个条件分支点，该条件表达式的结果决定了循环过程是继续还是退出。最后的reinit表达式，是一个无条件跳转点，返回test模块。

尽管for语句的灵活性使得循环的控制方法可以多种多样。但事实上，你所遇到的（或自己编写的）for循环都是计数器控制方式的，即循环次数是固定的。如下是计数器方式控制的for循环例子。

```
/* --- What does the loop output?  --- */
for (x = 0; x <= 10; x++)
   printf("%d ", x);

/* --- What does this one output?  --- */
letter = 'a';

for (c = 0; c < 26; c++)
   printf("%c ", letter + c);

/* --- What does this loop do?     --- */
numberOfOnes = 0;

for (bitNum = 0; bitNum < 16; bitNum++) {
   if (inputValue & (1 << bitNum))
      numberOfOnes++;
}
```

再看下面的for循环例子，以及它所对应的LC-3程序。该程序的功能很简单，计算0~9的整数和。

```
#include <stdio.h>

int main()
{
   int x;
   int sum = 0;

   for (x = 0; x < 10; x++)
      sum = sum + x;
}
```

编译器为以上C代码生成的LC-3代码，如图13-10所示。

```
1              AND  R0, R0, #0    ;   clear out R0
2              STR  R0, R5, #-1   ;   sum = 0;
3
4              ; init
5              AND  R0, R0, #0    ;   clear out R0
6              STR  R0, R5, #0    ;   init (x = 0)
7
8              ; test
9   LOOP:      LDR  R0, R5, #0    ;   perform the test
10             ADD  R0, R0, #-10
11             BRpz DONE          ;   x is not less than 10
12
13             ; loop body
14             LDR  R0, R5, #0    ;   get x
15             LDR  R1, R5, #-1   ;   get sum
16             ADD  R1, R1, R0    ;   sum + x
17             STR  R0, R5, #-1   ;   sum = sum + x;
18
19             ; reinit
20             LDR  R0, R5, #0    ;   get x
21             ADD  R0, R0, #1
22             STR  R0, R5, #0    ;   x++
23             BR   LOOP
24
25  DONE:      :
26             :
```

图13-10 for语句对应的LC-3代码

如下for循环代码中，存在一个常见的错误。

```
sum = 0;
for (x = 0; x < 10; x++);
   sum = sum + x;

printf("sum = %d\n", sum);
printf("x = %d\n", x);
```

试问，第一个printf的输出是什么？答案是“sum = 10”，为什么？第二个printf的输出则是“x = 10”，这又是为什么？这两个答案与你推测的是否一样？如果你仔细阅读，会发现问题出在分号的误用上。

for能完成的循环操作，换做while也可以等价实现。反之亦然。既然两种循环语句在某种程度上可互换，那么在编程中选用哪一个更合适呢？这个问题很难回答。但有一定通用规则：while适合“哨兵条件”，而for适合已知循环次数的情况。

循环嵌套

图13-11所示是一个for循环结构，同时它的循环体也是一个for结构。这种一个循环结构（内层）嵌套在另一个循环结构（外层）的形式，称为“循环嵌套”（nested loop）。此处的任务是打印出0到9的乘法表。其中，外层循环每次负责打印一整行，10次循环即意味着10行；内层循环每次打印一个乘积值，每循环10次完成一行打印。注意，printf中的特殊字符“\t”代表一个tab字符，它起到的是对齐作用，使每列输出更整齐。

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5    int multiplicand;    /* First operand of each multiply  */
 6    int multiplier;      /* Second operand of each multiply */
 7
 8    /* Outer Loop */
 9    for (multiplicand = 0; multiplicand < 10; multiplicand++) {
10      /* Inner Loop */
11      for (multiplier = 0; multiplier < 10; multiplier++) {
12        printf("%d\t", multiplier * multiplicand);
13      }
14      printf("\n");
15    }
16  }
```

图13-11 一个打印乘法表的程序

图13-12所示则是一个稍微复杂的例子。其复杂之处是，内层循环次数取决于外层循环中的outer值。即内层循环第一次循环次数为0，随后是1次、2次，以此类推。在该例的基础上，是一个更富挑战性的练习（参见习题13.6）。

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5    int sum = 0;              /* Initial the result variable */
 6    int input;                /* Holds user input            */
 7    int inner;                /* Iteration variables         */
 8    int outer;
 9
10    /* Get input */
11    printf("Input an integer: ");
12    scanf("%d", &input);
13
14    /* Perform calculation */
15    for (outer = 1; outer <= input; outer++)
16      for (inner = 0; inner < outer; inner++) {
17        sum += inner;
```

图13-12 一个使用嵌套for循环的程序

```
18      }
19
20    /* Output result */
21    printf("The result is %d\n", sum);
22  }
```

图13-12 一个使用嵌套for循环的程序（续）

13.3.3 do-while循环

在while循环中，条件测试总是在循环体执行之前先被执行。所以，对while来说，完全存在循环次数仅为0的情况，即循环体一次也不被执行（条件一开始就不为真）。下面，我们介绍一个while语句的变种——do-while，无论条件真假，它至少执行一遍循环体。这是因为，在do-while结构中，每次循环体执行之后才判断条件。下面是do-while的一个例子：

```
x = 0;
do {
   printf("%d \n", x);
   x = x + 1;
} while (x < 10);
```

其中，每次循环之后，测试条件“x < 10”。因此，无论x是否小于10，循环体至少被执行一次。而下一次循环是否能够被执行，则取决于条件表达式的值是否为非零值。该代码的执行输出如下所示：

```
0 1 2 3 4 5 6 7 8 9
```

从语法结构上说，do-while和while相同，都由两部分组成：

```
do
  loop_body;
while (test);
```

其一是循环体（loop_body），是一个简单语句或复合语句；其二是条件表达式（test），决定了下一次循环能否继续。

图13-13所示是do-while循环的控制流程图。注意它与while流程图之间的细微差别，即循环体和条件测试的位置互换。条件分支的返回点是循环体，即下一次循环的开始处。

至此，C语言的三种循环结构已介绍完毕，看起来它们之间的差别很小，对它们的选择似乎很难，但是，一旦你熟悉了它们，并掌握了一定的编程经验之后，做出选择就是很容易的事情。虽然从语义上说，这些结构是等价的，是可以互换的，但在文法上，它们是有差别的，代表的是不同的结构含义，在代码阅读上，也传递了不同的编程意图。

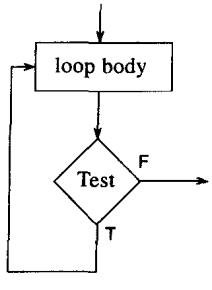


图13-13 do-while语句的流程控制图

13.4 基于控制结构的问题求解

在学习了一种新的控制结构之后，下面我们开始尝试用它们来编程求解问题。本节中，我们仍将使用“自顶向下”方法求解三个问题，其共同之处在于，我们都将借用C的控制结构来编程。

有效编程以解决问题的关键，是了解系统的基本要素。你要在合适的时候，借助这些要素来解决其中的难题。至此，我们已学过的C语言系统要素包括：三种基本变量类型及其运算符、两种判断结构和三种控制结构。

13.4.1 问题1：π的近似值求解

第一个问题是，通过级数展开式，计算π的值：

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \mathrm{L} + (-1)^{n-1}\frac{4}{2n+1} + \mathrm{L}$$

该问题的关键，实际上是根据用户的输入，计算该级数。例如，用户输入3，则求 $4 - \frac{4}{3} + \frac{4}{5}$。该式是一个无穷级数，运算式的项数越多，计算结果就越接近真实π值。

如同第12章的例子，解决问题的第一步任务是：确定合适的数据类型。由于级数展开过程中，涉及小数处理。所以，我们选用double类型浮点数来表示计算中涉及的所有变量。而从问题本身的属性来看，这也是最佳选择。

现在，我们开始问题的细化过程，将粗略的算法分解为C程序。大致过程包括：程序数据的初始化，然后是读取用户输入（级数展开项数），随后是根据给定项数，计算级数展开式的值，最后打印输出结果。如图13-14所示，我们已经将问题从定义转换成一个顺序结构。

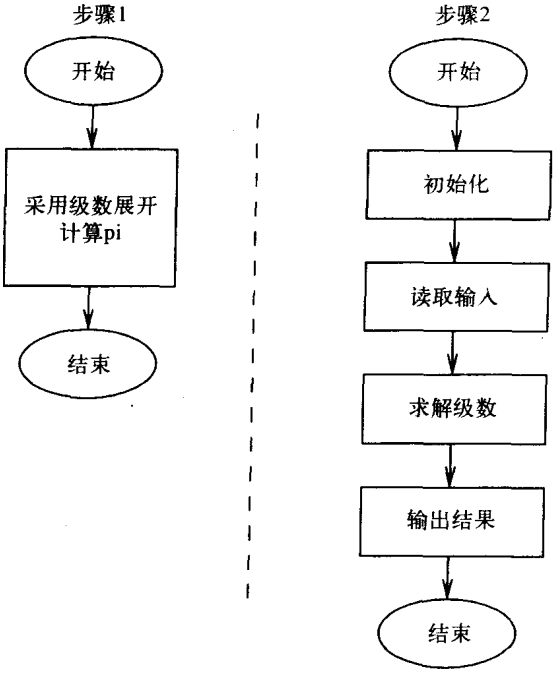


图13-14 根据给定级数展开式项数，计算π近似值的程序的初始分解

如图13-14所示，顺序结构中的各个模块都比较简单，将它们翻译为C代码应该是件容易事。但对于个别结构，还有待进一步细化，比如其中标识为“级数计算”的模块。该模块的要点是，循环处理级数式的每个项数，直到用户输入指定的项数。循环结构可采用计数器控制方式。图13-15所示是进一步的分解图。其中的计数器（counter）控制着循环过程，如果计数值小于用户输入值，则继续循环。注意，细化后的模块结构与for循环的流程图很相似。

现在，距离成功已经不远了，还剩一个模块，“计算下一项”，还未明确。我们发现，在级数展开式中，所有的偶数项都是减操作，而所有的奇数项都是加操作。所以，在该模块的循环处理中，我们需要判断当前子项是偶数项还是奇数项，然后将因子化（1或−1）后的子项累计入近似值结果中。因此，我们将使用如图13-16所示的判断结构。图13-17所示是最后的细化结果所对应的完整代码。

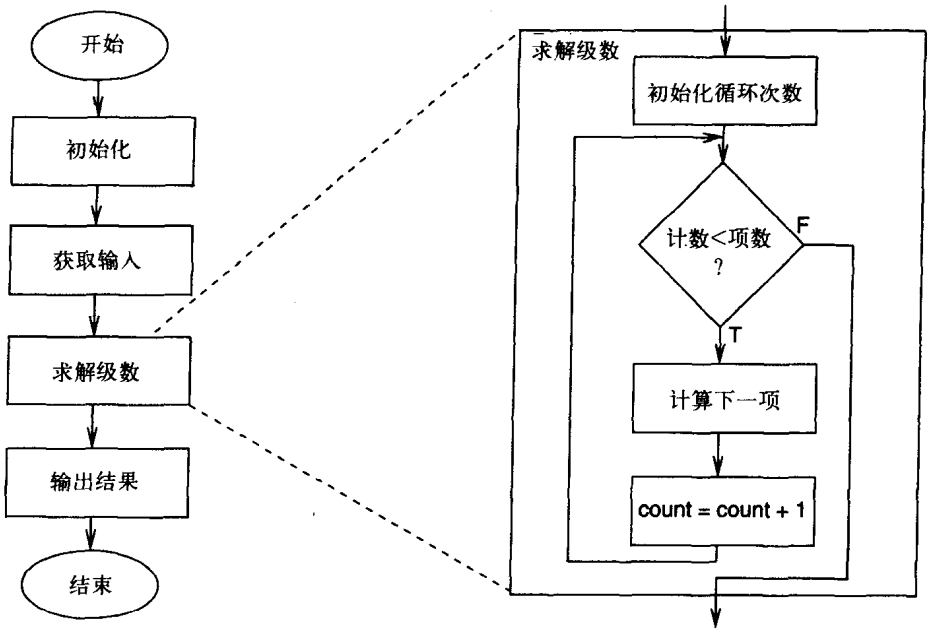


图13-15 “级数计算”模块的循环结构细化。通过循环过程，计算级数展开式得到π

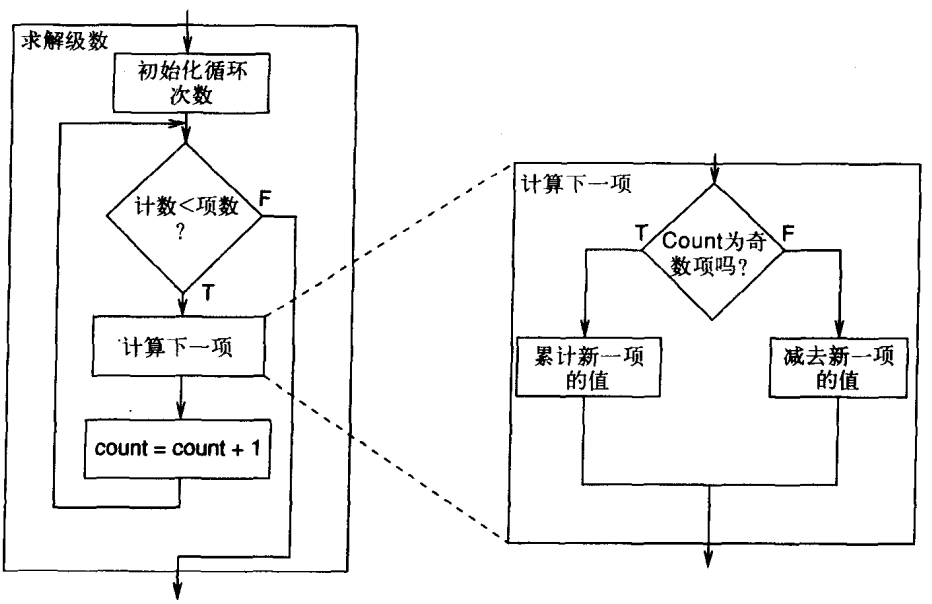


图13-16 根据当前项是为奇数项还是偶数项，来决定是做加法还是减法

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5    int count;        /* Iteration variable                */
 6    int numOfTerms;   /* Number of terms to evaluate       */
 7    double pi = 0;    /* approximation of pi               */
 8
 9    printf("Number of terms (must be 1 or larger) : ");
10    scanf("%d", &numOfTerms);
```

图13-17 计算π值的程序

```
11
12    for (count = 1; count <= numOfTerms; count++) {
13      if (count % 2)
14        pi = pi + (4.0 / (2.0 * count - 1)); /* Odd term  */
15      else
16        pi = pi - (4.0 / (2.0 * count - 1)); /* Even term */
17    }
18
19    printf("The approximate value of pi is %f\n", pi);
20  }
```

图13-17 计算π值的程序（续）

13.4.2 问题2：找出100以内的质数

下面进行第二个问题的求解：找出100以内的所有质数。所谓“质数”（prime），是指只能被1和它本身整除的整数。

同前面例子一样，求解的第一步（步骤0），是为该问题的相关数据选择合适的数据类型。由于质数概念只涉及整数，所以我们选择整数类型。

下一步就是逐步细化，将问题最终转换为C程序。首先，我们将问题描述为一个单独任务（步骤1）；然后，将这个单独任务分解为两个独立的子任务（步骤2），一是“初始化”，二是“计算”。

其中，“计算”子任务是整个程序的关键。“计算”子任务是逐个检查从2到100的数，判断它是否为质数。如果是质数，则打印输出。依此看来，计数器类型的循环控制结构比较合适。随后，我们继续细化“计算”模块。如图13-18所示，细化后的流程图与for结构非常相似。

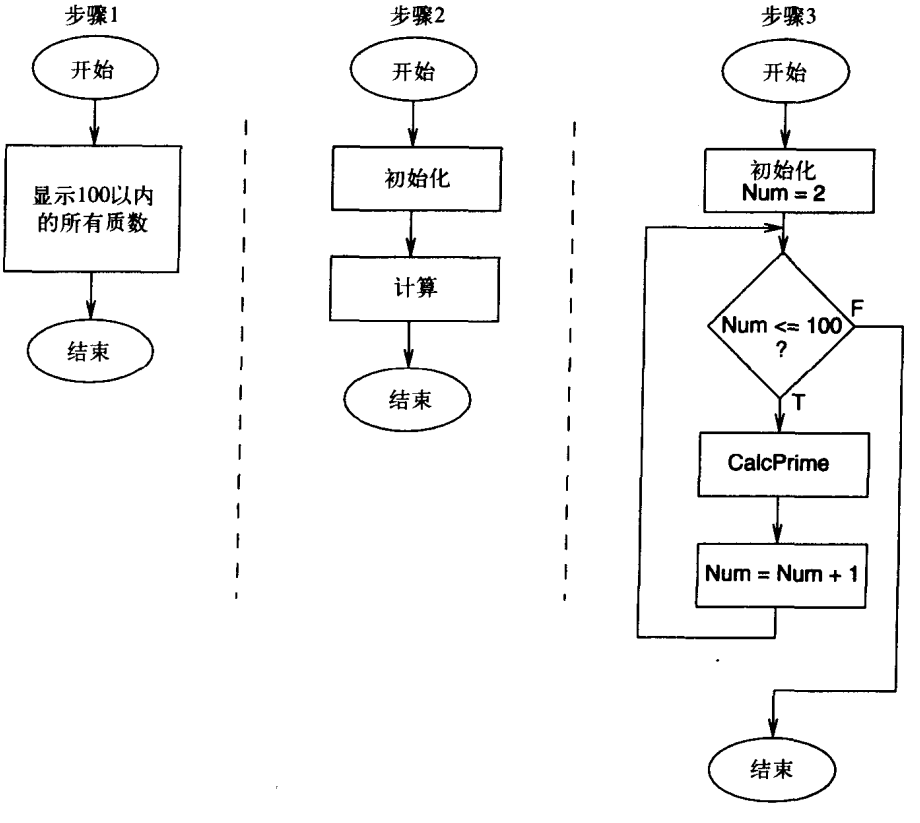


图13-18 找出100以内质数的问题分解：前3步

现在已接近C代码阶段了，还有“判断质数”这个子任务有待细化。该子任务是，判断某个数

是否为质数。在此，我们需要引入一个事实：如果一个介于2~100的整数不是质数，则它必然可以被2~10范围内的某个数整除，且该数不是它本身。基于这个判断规则，我们将该模块细化为如图13-19所示的子模块。对于任意数（2~100范围内），我们只需依次判断2~10范围内的每个数，是否存在能被它整除的数（注意，这个数不能是它本身）。如果不存在，则该数就是质数。

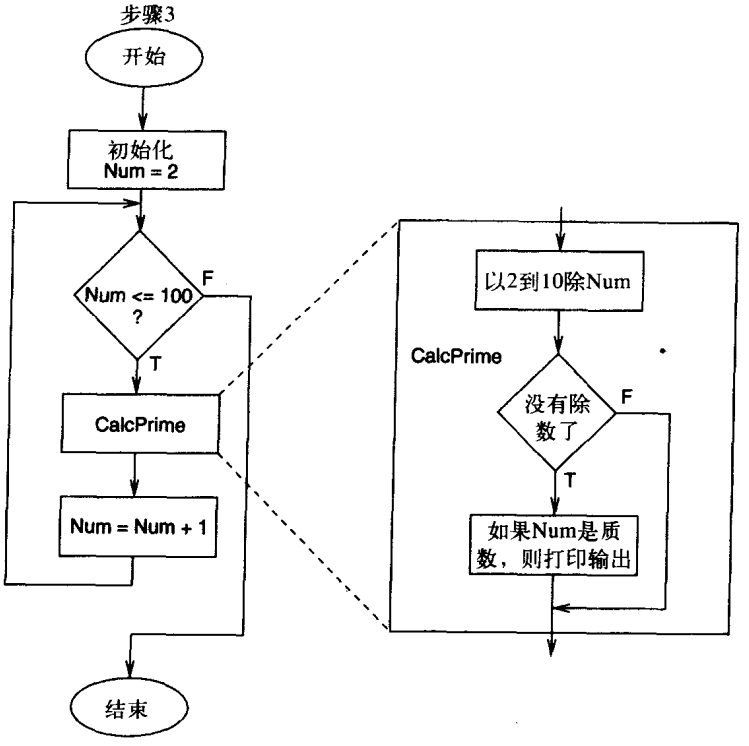


图13-19 分解CalcPrime模块

最后，我们还要对细化后的子任务“除以2~10的数”做进一步的细化。这需要让该数尝试去除以2~10中的每个数。显然，简单的方法又是计数器控制方式，如图13-20所示。

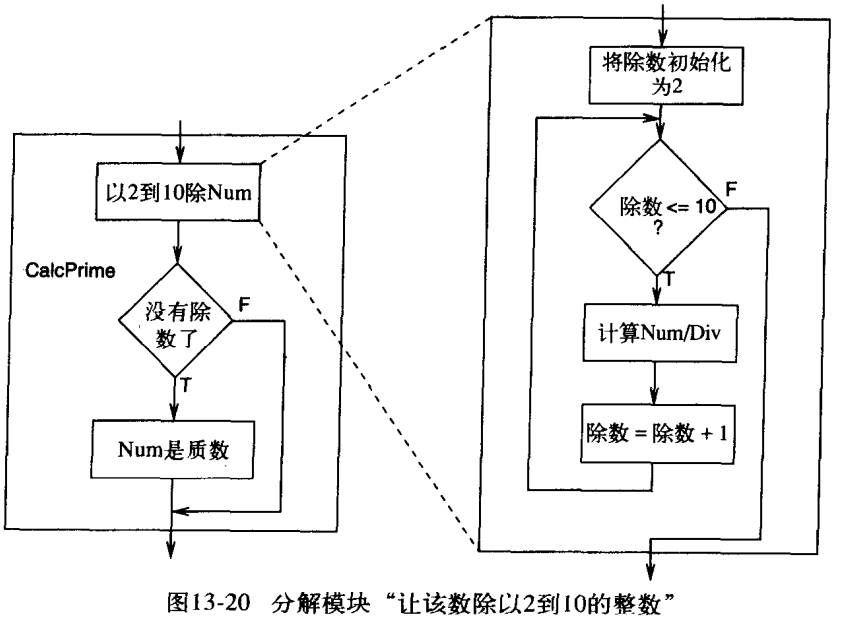


图13-20 分解模块“让该数除以2到10的整数”

至此，我们终于可以开始编程了。如图13-21所示，程序包含两个嵌套的for循环。外层的循环遍历是2~100，它与“计算”子任务的循环对应。内层循环的任务是判断该数是否为质数，它与子任务“除以2~10的数”的循环对应。

```
#include <stdio.h>
#define FALSE 0
#define TRUE  1

int main()
{
  int num;
  int divisor;
  int prime;

  /* Start at 2 and go until 100  */
  for (num = 2; num <= 100; num++) {
    prime = TRUE;   /* Assume the number is prime */

    /* Test if the candidate number is a prime */
    for (divisor = 2; divisor <= 10; divisor++)
      if (((num % divisor) == 0) && num != divisor)
        prime = FALSE;

    if (prime)
      printf("The number %d is prime\n", num);
  }
}
```

图13-21 找出2到100之间所有质数的程序

注意变量prime，在每次内层循环的初始，它的值都为“真”。但在内层循环中，一旦找到能够整除的数（在2~10之间），则标志位改为“假”。换句话说，如果内层循环结束时，标志位保持为真，则意味着找到一个质数。另外，为方便起见，我们还使用了C语言的预处理宏（即#define）来定义符号FALSE（代表0）和TRUE（代表1）。

13.4.3 问题3：分析一个E-mail地址

最后一个问题是，分析从键盘输入的E-mail地址是否是合法地址。所谓“合法地址”，是指在一个E-mail地址字符串中，字符串中必须包含“@”和“.”字符，且“@”符号必须出现在“.”前面。

与前两个问题一样，第一个任务（步骤0）是选择合适的数据类型。在本问题中，由于我们处理的是文本数据，而最基本的文本类型是ASCII码，也即char类型。事实上，对应输入文本的最佳数据类型是字符数组（或称字符串），但我们目前还未介绍过“数组”这种类型（将在第16章介绍）。所以，我们暂用char类型。

下面是问题的细化过程。如图13-22所示，我们先将问题分解为“输入处理”和“结果输出”两个模块（步骤1）。“结果输出”模块相对简单，即将已被检测视为合法的E-mail地址，打印输出。只是“输入处理”模块还有待进一步细化。

有关“输入处理”模块的细化（步骤2），由于我们选用的数据类型是字符类型，所以每次读入一个字符，然后不断循环，直到整个E-mail地址字符串结束。该循环过程可以采用“哨兵条件”的控制方法。如步骤2所示，我们将空格或换行符（\n）当做“哨兵”，即代表E-mail地址的结束标志。

步骤3的细化对象是循环体内的“下一字符处理”模块。在循环体中，对E-mail地址中的每个字符，我们都要检查。注意，我们的检查任务中，有一条就是要确认字符“@”和“.”的先后顺

序。为此，我们使用两个变量记录相关状态。在循环结束之时，这些状态值决定了最后应该输出的消息。

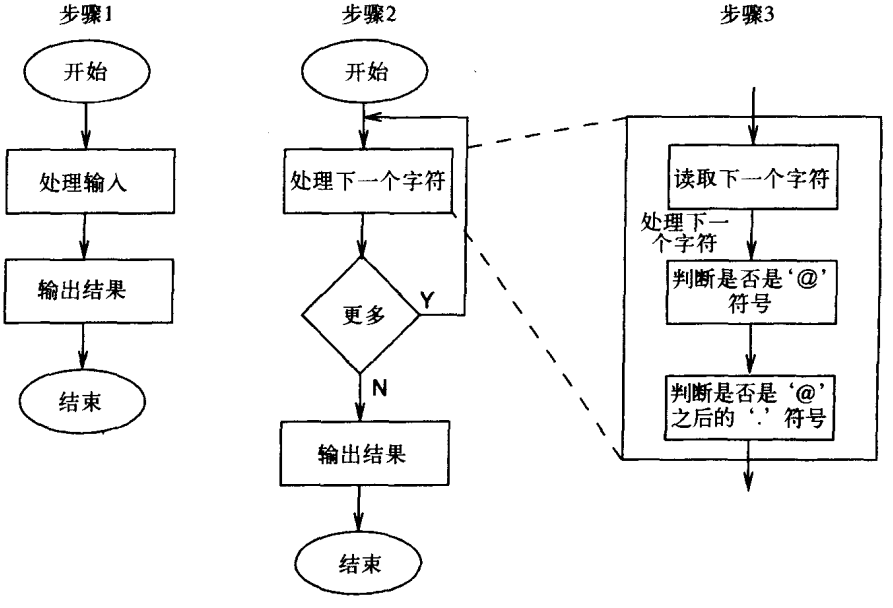


图13-22 分析E-mail地址程序的逐步细化

至此，我们离最后的C代码已经不远了。注意，图13-22中的循环结构与do-while语句的流程图非常相似。图13-23所示是该问题最终对应的C代码。

```
#include <stdio.h>
#define FALSE 0
#define TRUE  1

int main()
{
  char nextChar;   /* Next character in e-mail address */
  int gotAt  = FALSE;  /* Indicates if At @ was found  */
  int gotDot = FALSE;  /* Indicates if Dot . was found */

  printf("Enter your e-mail address: ");

  do {
    scanf("%c", &nextChar);

    if (nextChar == '@')
      gotAt = TRUE;

    if (nextChar == '.' && gotAt == TRUE)
      gotDot = TRUE;
  }
  while (nextChar != ' ' && nextChar != '\n');

  if (gotAt == TRUE && gotDot == TRUE)
    printf("Your e-mail address appears to be valid.\n");
  else
    printf("Your e-mail address is not valid!\n");
}
```

图13-23 判断一个E-mail地址是否合法的C程序

13.5 其他C语言控制结构

最后，我们对C语言的控制结构做一些补充介绍。除了之前已学习过的顺序、条件（if/if-else）、循环（for/while/do-while）等基本控制结构外，下面我们将介绍switch、break和continue这三种语句，它们能为程序员提供特别的程序流程控制功能。之所以在此介绍它们，只是为了内容讲述上的完整性，而在本书其他例子中，都不再使用这三种结构。

13.5.1 switch语句

在编程中，我们时常会遇到一个情况，即对某个值做一系列的测试。例如，在如下代码中，我们将对字符变量keyPress做一系列的测试，看它是否是“a”、“b”、“x”、“y”等特定字符。

```
char keyPress;

if (keyPress == 'a')
   /* statement A */

else if (keyPress == 'b')
   /* statement B */

else if (keyPress == 'x')
   /* statement C */

else if (keyPress == 'y')
   /* statement D */
```

其中，根据变量keyPress的值，我们将分别执行标识为A、B、C、D的那些语句（也可能一个都不执行）。例如，如果keyPress等于字符“a”，则执行语句A；如果为“b”，则执行语句B；依此类推。如果keyPress既不等于a、b，也不等于x或y，则什么语句也不执行。

如果存在多种条件需要检查，如上面代码所示，为找到与变量匹配的条件，最坏情况下，所有的if语句都要被执行一遍。为给编译器一个优化代码的机会，如跳过其中的某些测试，C语言提供了switch语句。如下所示代码，它的行为与上面代码完全一样。不同之处是，这里用的是switch语句，而不是一连串的if-else语句。

```
char keyPress;

switch (keyPress) {
case 'a':
  /* statement A */
  break;

case 'b':
  /* statement B */
  break;

case 'x':
  /* statement C */
  break;

case 'y':
  /* statement D */
  break;
}
```

我们注意到，switch语句包含很多以关键字case开头的行，case之后是一个标签。程序首先执行的是switch语句，计算keyPress的值；然后，确定与该值匹配的case标签；如果匹配，则执行case行随后的语句。

下面仔细研究一下switch语句的结构。关键词switch指示的是一个“判断表达式”，该表达式的值必须是“整数类型”（如int或char）。如果某个case的标签值与该表达式值匹配，则程序控制权转移至该case标签后面的语句或代码块。每个case行之后，可以无任何语句，也可以由多个语句组

成，如同复合语句一样，只是不需要头尾分界的大括号。从整个switch复合语句来看，执行的起始点是标识值与switch表达式值匹配的case行，而不是从第一个case就开始执行。一个switch结构中，每个case标签都必须是惟一的，即不允许出现相同的标签。

另外，case标签的内容必须是常量或常量表达式，即不能是一个不确定的、会随着程序执行而变化的值。例如，如下形式就是不合法的（因为*i*是一个变量）：

```
case      i:
```

每个case之后的代码块，都以break语句结尾。break的作用是退出switch结构，即将程序控制权直接转给switch尾部括号之后的那个语句。break语句的存在，并不是必需的。如果没有break，则程序会顺序执行，最终控制权转移给下一个case。例如，如上面代码所示，将代码块C尾部的break语句删除。那么，如果keyPress与case 'x' 匹配，则代码段C和D都会被执行。这也是一个编程技巧，但通常的用法中，case的尾部总是有一个break。

另外，还有一种默认case类型，即default。它的作用是，当switch表达式值与任何case常量值都不匹配时，则程序控制权就转交给default后面的语句。而如果switch结构中没有给出default行，则当switch表达式值与所有case常量都不匹配之时，任何语句都不执行。

注意，从语法格式上来说，switch的最后一个case后面并不需要break。因为，无论如何，switch执行到此都是要结束的。但是，在最后一个case中包含break，是一个好的编程习惯。例如，你又要添加一个case在最后，则不用挂念着为前一个case后面补上break。所以，保证每个case后面都以break结束，是一个安全的编程习惯。

13.5.2 break和continue语句

在上一节中，我们介绍了在switch结构中，怎样使用break语句。事实上，在循环结构中，偶尔也会使用break语句及continue语句。

break的作用是，让编译器生成一个提前退出循环或switch结构的代码。在循环体内使用break，它会跳出break所在的最内层循环体，即结束本层循环。相比之下，continue语句的作用是，让编译器生成一条跳转指令，提前至下一次循环的开始。这些语句可以出现在循环体内，且仅对最直接包含它们的循环结构有效。总之，break和continue的本质作用是，让编译器生成一个无条件跳转指令，从循环体当前位置直接跳转至代码的另一个位置（如循环体外或循环体开始处）。如下是两个例子（break和continue）：

```
/* This code segment produces the output: 0 1 2 3 4 */
for (i = 0; i < 10; i++) {
   if (i == 5)
      break;
   printf("%d ", i);
}
/* This code produces the output: 0 1 2 3 4 6 7 8 9 */
for (i = 0; i < 10; i++) {
   if (i == 5)
      continue;
   printf("%d ", i);
}
```

13.5.3 简单计算器的例子

如图13-24所示的程序，与第10章的计算器例子相比，功能上相同。先是提示用户输入三个参数：操作数1、运算符、操作数2；然后，对两个操作数做计算（运算符）；最后，输出计算结果。下面这段代码，将采用switch语句，根据不同的运算符进行计算。

```
#include <stdio.h>

int main()
{
  int operand1, operand2;      /* Input values              */
  int result = 0;              /* Result of the operation   */
  char operation;              /* operation to perform      */

  /* Get the input values */
  printf("Enter first operand: ");
  scanf("%d", &operand1);
  printf("Enter operation to perform (+, -, *, /): ");
  scanf("\n%c", &operation);
  printf("Enter second operand: ");
  scanf("%d", &operand2);

  /* Perform the calculation */
  switch(operation) {
  case '+':
    result = operand1 + operand2;
    break;

  case '-':
    result = operand1 - operand2;
    break;

  case '*':
    result = operand1 * operand2;
    break;

  case '/':
    if (operand2 != 0)             /* Error-checking code. */
      result = operand1 / operand2;
    else
      printf("Divide by 0 error!\n");
    break;

  default:
    printf("Invalid operation!\n");
    break;
  }

  printf("The answer is %d\n", result);
}
```

图13-24 计算器程序

13.6 小结

我们总结一下本章的几个关键内容。本章的基本目的是，学习C语言的几种控制结构，以扩展我们用C语言求解问题的方法和能力。

- 判断结构。C语言有两种基本判断语句：if和if-else。它们能根据表达式值的真假，有条件地控制随后要执行的语句。
- 循环结构。C语言支持三种循环语句：while、for和do-while。这些语句的特点是，不断反复地执行某个语句或一段代码，直到特定条件表达式值变为假为止。其中，while和do-while适合“哨兵条件”控制方式，而for则适合计数器控制方式。
- 基于控制结构求解问题。在前几章学过的编程技能的基础上（如C语言的3种基本类型、变量、运算符及printf和scanf等I/O操作），本章又增加了“控制结构”的编程方法。最后，为熟悉这种编程方法，我们练习了三个用控制结构解决问题的例子。

13.7 习题

13.1 试写出对应于图13-24所示“计算器程序”的，由LC-3编译器生成的符号表。

13.2 试回答以下问题：

a. 如下这段代码，预编译器处理之后的结果是什么？

```
#define VERO -2

if (VERO)
   printf("True!");
else
   printf("False!");
```

b. 该代码执行后的输出是什么？

c. 如果修改代码如下，代码执行后与之前的结果有什么不同？为什么？

```
#define VERO -2

if (VERO)
   printf("True!");
else if (!VERO)
   printf("False!");
```

13.3 在C语言中，if-else语句可以替代条件运算符（12.6.3节）。试用if-else语句重写如下的条件运算符语句。

```
x = a ? b : c;
```

13.4 试分别描述如下语句在x=0和x=1两种情况下的动作。

a.

```
if (x = 0)
   printf("x equals 0\n");
else
   printf("x does not equal 0\n");
```

b.

```
if (x == 0)
   printf("x equals 0\n");
else
   printf("x does not equal 0\n");
```

c.

```
if (x == 0)
   printf("A\n");
else if (x != 1)
   printf("B\n");
else if (x < 1)
   printf("C\n");
else if (x)
   printf("D\n");
```

d.

```
int x;
int y;

switch (x) {
case 0:
   y = 3;

case 1:
   y = 4;
   break;

default:
   y = 5;
   break;
}
```

e. 第4部分中，如果x不等于0或1，会发生什么？

13.5 给出对应于习题13.4第4小题的switch语句，由LC-3 C编译器编译之后所生成的LC-3代码。

13.6 图13-12所示是一个嵌套for循环的C程序。试回答以下问题：

a. 试用数学方法，分析该程序的运算级数（或计算复杂度）；

b. 试编写程序，计算如下函数：

$$f(n)=f(n-1)+f(n-2)$$

其初始条件为：$f(0)=1, f(1)=1$

13.7 试问，如下if-else语句可以转换成switch语句吗？如果可以，请转换；如果不可以，请说明理由。

```
if (x == 0)
   y = 3;
else if (x == 1)
   y = 4;
else if (x == 2)
   y = 5;
else if (x == y)
   y = 6;
else
   y = 7;
```

13.8 试问，在如下的结构中，loopBody语句的执行次数分别是多少？

a.

```
while (condition)
   loopBody;
```

b.

```
do
   loopBody;
while (condition);
```

c.

```
for (init; condition; reinit)
   loopBody;
```

d.

```
while (condition1)
   for (init; condition2; reinit)
      loopBody;
```

e.

```
do
   do
      loopBody;
   while (condition1);
while (condition2);
```

13.9 试问，如下各代码段的输出分别是什么？

a.

```
a = 2;
while (a > 0) {
   a--;
}
printf("%d", a);
```

b.

```
a = 2;
do {
   a--;
} while (a > 0)
printf("%d", a);
```

c.

```
b = 0;
for (a = 3; a < 10; a += 2)
   b = b + 1;
printf("%d %d", a, b);
```

13.10 试修改如图13-4所示程序，将其中的if-else语句替换为switch语句。

13.11 试对图13-23所示的“E-mail地址验证”程序做出修改，新的合法性验证要求，在E-mail地址字符串中，符号@前、符号@和.之间、符号.之后，至少存在一个字母。满足这种要求的E-mail地址，才认为是合法的。

13.12 试回答以下问题。其中，x是一个数值为4的整数。

a. 如下代码的输出是什么？

```
if (7 > x > 2)
   printf("True.");
else
   printf("False.");
```

b. 如下代码会无限循环吗？

```
while (x > 0)
   x++;
```

c. 如下代码执行后，x的值为多少？

```
for (x = 4; x < 4; x--) {
   if (x < 2)
      break;
   else if (x == 2)
      continue;
   x = -1;
}
```

13.13 试对如下代码做出修改，将for循环结构替换为do-while结构。

```
int main()
{
   int i;
   int sum;

   for (i = 0; i <= 100; i++) {
      if (i % 4 == 0)
         sum = sum + 2;
      else if (i % 4 == 1)
         sum = sum - 6;
      else if (i % 4 == 2)
         sum = sum * 3;
      else if (i % 4 == 3)
         sum = sum / 2;
   }
   printf("%d\n", sum);
}
```

13.14 试编写C程序，读取整数输入k，然后依次输出各行。其中，第1行是1，第2行是两个2，第3行是三个3，……，直到最后一行的k个k。

例如，输入为5，则输出如下：

```
1
2   2
3   3   3
4   4   4   4
5   5   5   5   5
```

13.15 a. 将如下while循环替换为for结构语句：

```
while (condition)
   loopBody;
```

b. 将如下for循环替换为while结构语句：

```
for (init; condition; reinit)
   loopBody;
```

13.16 试问，如下代码的输出结果是什么？

```
int r = 0;
int s = 0;
int w = 12;
int sum = 0;

for (r = 1; r <= w; r++)
```

```
    for (s = r; s <= w; s++)
       sum = sum + s;

printf("sum =%d\n", sum);
```

13.17 如下代码看起来有些特别，试描述它的输出应该是什么？

```
int i;

scanf("%d", &i);
for (j = 0; j < 16; j++) {
  if (i & (1 << j)) {
    count++;
  }
}

printf("%d\n", count);
```

13.18 试给出如下各代码段的输出结果。

a.

```
int x = 20;
int y = 10;

while ((x > 10) && (y & 15)) {
   y = y + 1;
   x = x - 1;
   printf("*");
}
```

b.

```
int x;

for (x = 10; x ; x = x - 1)
   printf("*");
```

c.

```
int x;

for (x = 0; x < 10; x = x + 1) {
   if (x % 2)
      printf("*");
}
```

d.

```
int x = 0;
int i;

while (x > 10) {
  for (i = 0; i < x; i = x + 1)
    printf("*");
  x = x + 1;
}
```

第14章 函 数

14.1 概述

函数就是子程序，而子程序是现代编程语言的灵魂。函数为程序员提供了一种编程元素的扩展机制，是对程序已有操作符和结构的一个扩展。函数概念非常重要，很久以前它就已成为程序语言必不可少的一部分，所有类型的指令集结构，在硬件结构上都提供了对函数的直接支持（包括LC-3）。

函数也称为过程、子程序或方法，它们都是一个意思。为什么函数如此重要？因为它是一种“抽象”，即通过函数，我们将功能与实现方法相互分离。一旦创建了一个功能模块，并理解了它的结构特性，之后就可以把它当做一个现成构件而直接使用（不必细究它的实现细节）。试想，如果没有抽象，构建计算机这样一个复杂系统，以及开发运行在计算机之上的软件，将是多么困难的事情。

函数对我们来说并不陌生。在LC-3汇编语言中，我们就学习和使用了多种不同的函数。从语法上来说，LC-3的汇编子程序和C的函数有所不同，但其本质是一样的。

C语言是一个非常依赖于“函数”的编程语言。换句话说，本质上，C程序就是函数的集合。其中，每条语句都属于且仅属于一个函数。所有的C程序，都从main函数开始执行，并结束于main。在main函数中，又可以调用其他函数，而这些函数又可以调用更多的函数。但最终的控制权终归会返回到main。main函数的结束，就是程序的终止（假设不存在其他意外情况造成程序的永久终止，如断电、出错等）。

本章将介绍C语言中的函数。我们从一些简单例程开始，首先对C的函数调用语法有个感性认识；然后，讲述函数的实现方法，以及为确保函数正常工作，底层所必须实现的操作；最后，将通过问题求解，感受函数在编程中发挥的作用。

14.2 C语言中的函数

下面是一个非常简单的C函数例程。图14-1中，函数PrintBanner是一个打印横幅（banner）的程序。程序从main函数开始执行，随后调用PrintBanner函数（在显示设备上，打印一整行的“=”字符）。

```
 1  #include <stdio.h>
 2
 3  void PrintBanner();      /* Function declaration */
 4
 5  int main()
 6  {
 7     PrintBanner();        /* Function call        */
 8     printf("A simple C program.\n");
 9     PrintBanner();
10  }
11
12  void PrintBanner()       /* Function definition  */
13  {
14     printf("============================\n");
15  }
```

图14-1 使用函数打印标志信息的C程序

函数PrintBanner非常简单，它不需要从调用者那里获得任何输入参数，也不需要向调用者返

回任何结果（例如，不需要统计打印字符的个数）。在此，我们称main函数为*调用者*（caller），PrintBanner为*被调用者*（callee）。

14.2.1 带参数的函数

在PrintBanner和main之间，没有信息交互，所以接口看起来非常简单。但通常，我们总会在调用者和被调用者之间传递一些信息。所以，在下面这个例子中，我们将介绍C语言中的函数之间怎样传递信息。图14-2所示代码中，有一个带形式参数[⊖]（argument）的函数Factorial。

```
 1  #include <stdio.h>
 2
 3  int Factorial(int n);           /*! Function Declaration !*/
 4
 5  int main()                      /*  Definition for main   */
 6  {
 7    int number;                   /*  Number from user      */
 8    int answer;                   /*  Answer of factorial   */
 9
10    printf("Input a number: ");   /*  Call to printf        */
11
12    scanf("%d", &number);         /*  Call to scanf         */
13
14    answer = Factorial(number);   /*! Call to factorial    !*/
15
16    printf("The factorial of %d is %d\n", number, answer);
17  }
18
19  int Factorial(int n)            /*! Function Definition  !*/
20  {
21    int i;                        /*  Iteration count       */
22    int result = 1;               /*  Initialized result    */
23
24    for (i = 1; i <= n; i++)      /*  Calculate factorial   */
25      result = result * i;
26
27    return result;                /*! Return to caller     !*/
28  }
```

图14-2　计算阶乘的C程序

函数Factorial的任务是计算从1到n之间，所有整数的乘积。其中，n的值由调用者（main函数）提供。将该函数的功能表示为计算公式，如下所示：

$$\text{factorial}(n) = n! = 1 \times 2 \times 3 \times \cdots \times n$$

其中，函数计算结果存入变量result，然后返回给调用者（通过return语句）。我们说，函数Factorial从调用者那里获得了一个整型参数，并且返回一个整型结果给调用者。在此例子中，Factorial的返回值又被赋给了调用者中的answer变量（第14行）。

让我们仔细看一下C语言中函数调用的语法格式。图14-2所示代码中，有4行是我们感兴趣的：第3行的Factorial函数*声明*，第19行的函数*定义*，第14行的Factorial函数调用，第27行的Factorial函数返回。

1. *声明*

第3行是Factorial函数的声明。函数声明的目的是什么？答案是，通过函数声明告诉编译器函数的一些相关属性（如同变量声明一样）。我们又称之为“*函数原型*”（function prototype）。函数

⊖ 所谓“形式参数”（argument），又被翻译为变元、输入参数等，由于它是在被调用时刻被创建的变量，并随着函数的结束而消失（与函数内部的局部变量相同）。与局部变量不同的是，它具有在调用者和被调用者之间传递参数的作用（而局部变量没有此作用），所以我们认为称之为“形式（临时创建的）参数”更直观。——译者注

声明的内容包括：函数名、返回值类型，以及输入参数列表。函数声明语句以分号（;）结尾。

函数返回值类型是声明语句的第一个字段。该类型可以是任何C类型，如int、char、double等，它描述的是函数将产生的输出结果（惟一的）的类型。但是，并不是所有函数都有返回值，如之前例子中的PrintBanner函数。如果一个函数没有返回值，我们称它的返回类型是void，它传达给编译器的意思是，该函数不返回任何结果。

第二个字段是函数名。函数名可以是任何合法的C标识符。函数名的选择应该能反映该函数的一些行为和特性。例如，Factorial这个函数名，一看就知道是执行数学操作“阶乘”（factorial）。另外，一个好的命名规范，函数名和变量名之间应该有所区别。如本书中，函数名的首字母都是大写字母，如Factorial。

在函数的声明中，也描述了函数所需的输入参数（parameter）的类型和顺序，它们也是该函数期望从调用者获取的参数类型和传入顺序。在声明时，我们可以为每个参数指定一个名字（但不是必要的）。例如，函数Factorial的声明语句中，输入参数是一个整数值，它对应的就是函数内部的*n*。有的函数不需要任何输入。例如，PrintBanner函数的参数列表就是空的。

2. 调用

第14行是函数Factorial的调用。该语句表示main函数调用了函数Factorial。但是在Factorial函数执行之前，main函数必须为它传递一个整数值。我们称由调用者传递给被调用者的这些数值为“参数[⊖]”（argument）。参数可以是任何合法的表达式，只是它们的类型必须与被调用者所需要的类型相匹配。这些参数紧随在被调用函数名后面的圆括号里。例如，在本例中，main函数将变量number的值作为形式参数传给被调用者，而函数Factorial的返回值则赋给变量answer。

3. 定义

从第19行开始，是定义Factorial函数的代码。注意，定义中的第一行语句和函数声明几乎一样（除了最后的分号）。函数名后面括号中的内容是函数的正则化参数列表（formal parameter list）。所谓“正则化参数列表”，是一组变量的声明，每个变量的内容都将被初始化为调用者提供的对应数值。例如，本例中，第14行调用函数Factorial时，形式参数*n*的内容将被初始化为main所属变量number的值。在每个函数被调用处，调用者传递的形式参数必须与正则化参数列表中定义的参数类型和顺序相匹配。

之后的大括号内是函数体（function body）。在函数体中，包含的是函数执行时所需要的变量声明和语句。任何在大括号内出现的变量，都是该函数的局部变量。

值得一提的是，在C语言中，函数调用者的局部变量对被调用者来说是不可见的。例如，Factorial（被调用者）无法直接修改（调用者main）内部的变量number。并且，函数调用时，参数传递采用的是“传值”方式。

4. 返回

第27行语句，是将控制权从函数Factorial交还给调用者（如main）。由于函数Factorial要返回一个值，所以关键词return必须跟随一个表达式，且该表达式的类型必须与函数声明的返回值类型相匹配。例如，Factorial中的“return result;”语句，它将result中存储的计算结果返回给调用者。通常，带返回值的函数体内至少存在一个return语句。而没有返回值的函数（声明为void的函数），不需要return语句。换句话说，在这类函数中，return语句是可有可无的，只要最后一条语句执行完，控制权就可以立刻交还给调用者。

那么，main函数如何返回呢？它的返回类型是int（ANSI C标准），但是我们在例子中没有发现它的return语句。事实上，按照严格规范，在前面的所有例子中，我们都应该为main函数补上一

⊖ 在中文中，argument和parameter都翻译为“参数”。事实上，两者的差别很微妙。——译者注

条“return 0;”语句。在C语言中，如果一个带返回值的函数没有显式返回一个值，则将最后一条语句的执行结果返回给调用者。由于我们通常忽略main函数的返回值（main函数因为不能），所以，为简洁起见，我们通常省略这条return语句。

下面，我们总结一下上述各语法：函数声明（或原型）的作用是，向编译器传递函数信息（函数名、参数（来自调用者）的类型和顺序、返回值的类型等）；函数定义是该函数真正的源代码。定义中包含有正则化参数列表，即各个参数的名字和顺序；函数的激活方式是函数调用。被传入的各形式参数值包含在函数调用的圆括号内，它们将按照定义顺序，将返回值传递给函数定义的各参数，即第1个值赋给第1个参数、第2个值赋给第2个参数，依此类推；返回值是函数的输出，传给函数调用者。

14.2.2 求解圆面积

如图14-3所示，我们将进一步讲述C函数的语法。该程序的任务是计算圆环的面积（大圆中间挖掉一个小圆）。换句话说，我们所要做的事情是，计算外半径和内半径这两个圆面积，然后相减。本程序中，我们编写了一个计算给定半径的圆面积的函数AreaOfCircle，该函数有一个double类型参数，返回值也是一个double类型结果。

```
 1  #include <stdio.h>
 2
 3  /* Function declarations */
 4  double AreaOfCircle(double radius);
 5
 6  int main()
 7  {
 8    double outer;                          /* Inner radius */
 9    double inner;                          /* Outer radius */
10    double areaOfRing;                     /* Area of ring */
11
12    printf("Enter inner radius: ");
13    scanf("%lf", &outer );
14
15    printf("Enter outer radius: ");
16    scanf("%lf", &inner );
17
18    areaOfRing = AreaOfCircle(outer) - AreaOfCircle(inner);
19    printf("The area of the ring is %f\n", areaOfRing);
20  }
21
22  /* Calculate area of circle given a radius */
23  double AreaOfCircle(double radius)
24  {
25    double pi = 3.14159265;
26
27    return pi * radius * radius;
28  }
```

图14-3　计算圆环的面积的C程序

再次提醒，注意下面这段论述：当函数AreaOfCircle执行时，它可以看到并修改局部变量pi和参数radius。但是，它不能修改任何main函数体内的变量（返回值除外）。

函数AreaOfCircle与之前例子略有不同。在main函数中，曾多次调用AreaOfCircle。在此，AreaOfCircle扮演的是一个基本运算操作（这也是函数的优点之一）。从更大的层面上来看，实际程序中存在这样一些函数，它们会在上百甚至上千个地方被调用。所以，将如AreaOfCircle这样的基本操作封装成函数，无疑大大缩减了代码占用的空间；并且，这样也大大方便了代码维护，同时也使程序结构更加清晰。与直接内嵌（embedded in-line）方法相比，使用AreaOfCircle方式使得代码意图更加明显。

有人或许记得12.6.2节中有关“常数值”的讨论。在第25行中，我们“应该”将变量pi修饰为const类型，但这里却没有。这是因为，我们希望没有读过第12章的读者也能看懂这个例子。

14.3 C语言中函数的实现

下面介绍C语言函数的底层实现原理。C函数与LC-3中的子程序（第9章）是等价的，它们的核心操作完全相同。在C语言中，函数调用包括三个基本步骤：（1）调用：调用者将参数传递给被调用者，并交出控制权；（2）执行：被调用者执行任务；（3）被调用者返回函数结果，并将控制权交还给调用者。在调用机制的实现中，有一个重要原则，即函数必须是与“*调用者无关的*”。换句话说，一个函数可以被不同的调用者调用（所以函数实现中，不能对调用者附加任何前提假设）。在本节中，我们将以LC-3为例，讲述该过程的实现。

14.3.1 运行时栈

我们需要研究一种机制，在函数被调用时，“激活”（activate）被调用函数。换句话说，在函数被执行前，我们必须在内存中，为该函数的局部变量分配空间。解释如下：

每个函数对应一个存储它的局部变量的内存模板（memory template）。回顾12.5.2节中的论述，函数的活动记录，是指内存中存储函数局部变量的一个模板。函数中被声明的每个局部变量，在活动记录中都占有一个位置。帧指针（frame pointer，R5）标识的是活动记录的起始地址。问题是，活动记录在内存的什么地方？该问题的答案有以下两种选择：

（1）方案1——编译器负责为每个函数分配存放活动记录的地方。例如，函数*A*的活动记录在地址*X*处，函数*B*的活动记录在地址*Y*处，依此类推。当然，在这种方式下，各函数的活动记录之间相互不重叠。这种管理分配方法，看起来简单明了，但存在一系列的限制。例如，在函数*A*中，再次调用它自己（还是函数*A*）时，结果如何？我们称这种情况为“*递归*”（recursion）（详见第17章）。如果函数*A*调用了它自己，显然被调用者*A*的局部变量将覆盖调用者*A*的活动记录空间，结果程序出现异常。所以，对于像C语言这样支持递归的编程语言，方案1是失败的。

（2）方案2——函数每被调用一次，就为它在内存中分配一个活动记录空间。函数返回时，则将该活动记录空间返还，以供其他函数调用时使用。这种方案看起来比方案1复杂，但允许函数递归操作。由于每次调用（invocation）函数时，都将在内存中为其分配一个存储局部变量的空间。所以，如果函数*A*调用它自己，则被调用者*A*有自己的活动记录空间，而该记录空间与调用者*A*的活动记录空间是不一样的。此外，如下机制可进一步降低方案2的复杂性：采用数据结构“栈”，可以方便地跟踪函数调用模式（函数*A*调用函数*B*，*B*又调用*C*），如下例所示。

图14-4所示代码中，包含三个函数：main、Watt和Volta。至于每个函数的任务是什么，对本例来说并不重要（我们所关心的是其调用模式），所以我们删掉了其中的一些细节代码。其中，main调用Watt，Watt又调用Volta；之后，控制权返回main；再之后，main又调用Volta。

```
 1  int main()
 2  {
 3    int a;
 4    int b;
 5
 6    :
 7    b = Watt(a);          /* main calls both     */
 8    b = Volta(a, b);
 9  }
10
11  int Watt(int a)
```

图14-4 体现函数调用的栈特性

```
12  {
13    int w;
14
15    ;
16    w = Volta(w, 10);      /* Watt calls Volta    */
17
18    return w;
19  }
20
21  int Volta(int q; int r)
22  {
23    int k;
24    int m;
25
26    ;                      /* Volta calls no one */
27    return k;
28  }
```

图14-4 体现函数调用的栈特性（续）

在每个函数中，都有自己的活动记录（包括局部变量、暂存信息以及调用者传入的形式参数）。函数被调用时，我们都将采用一种类似栈的形式，在内存中为该活动记录分配一个空间。如图14-5所示。

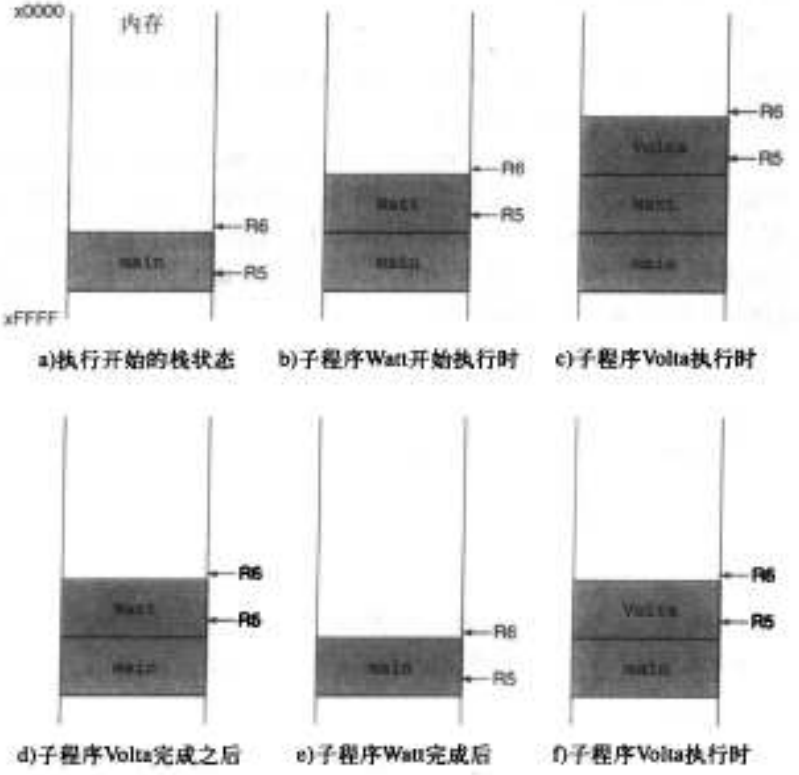


图14-5 图14-4所示程序在运行时的栈空间快照

其中，每个阴影区代表一个特定函数的活动记录。上面这组图示，演示了各函数在调用和返回时，运行时栈空间动态增长和缩小的情况。注意，将数据项压入栈时，栈顶总是向着低的内存地址方向移动，但我们称之为“增长”。

图14-5a所示是程序刚开始的栈快照。由于程序的开始是main，所以栈上分配的是main的活动记录。图14-5b所示是main调用Watt之后的栈快照。注意，活动记录的分配也是栈风格的，即函数调用时，活动记录入栈；函数返回时，活动记录出栈。图14-5c~f所示，是栈在代码运行的不同时

刻的快照。其中，R5总是指向当前活动记录的某个位置（局部变量的起始地址），R6总是指向运行时栈顶（所以，又称它为栈指针）。总之，这些寄存器在运行时栈以及C语言的函数实现中，都扮演着非常关键的角色。

14.3.2 实现机制

很显然，函数调用时，底层有很多事情要做：形式参数传递、活动记录的入栈和出栈、控制权在函数之间的转交。这些工作，有的是由调用者负责，有的是由被调用者负责。

完成这些工作的先后步骤是：(1) 调用者将参数拷贝到被调用者所能访问的内存区域。(2) 被调用函数的开始代码，将活动记录压入栈，并在栈中保存一些备忘（bookkeeping）信息（以保证控制权返回调用者后，调用者的原局部变量和寄存器内容在调用前后没有变化）。(3) 被调用者完成自己的工作。(4) 被调用函数完成之后，将活动记录出栈，并将控制权返回调用者。(5) 调用者重获控制权后，读取被调用者的返回值。

下面，我们来看看执行这些操作的LC-3代码。例如，图14-4中第18行所示的函数调用语句：

```
w = Volta(w, 10);
```

该语句对应的LC-3代码如下所述。

1. 调用

在“w = Volta(w, 10);”语句中，Volta的形式参数有两个，返回值赋给本地整型变量w。在将该语句翻译为LC-3代码的过程中，编译器做了如下工作：

(1) 通过将两个参数值压入运行时栈，向函数Volta传递参数。另外，由于R6总是指向运行时栈的顶部（即当前运行时栈顶数据所在的地址）；所以，每向栈中压入一个数据项，首先是递减R6值，然后将数据存入R6指向的地址。在LC-3结构中，C函数的参数，按照它们在函数调用中的顺序，从右到左依次被压入栈。在此，即意味着先压入数值10（最右边），然后是w的值。

(2) 通过JSR指令，把控制权传给函数Volta。

函数调用操作的LC-3代码如下所示：

```
AND  R0, R0, #0  ; R0 <- 0
ADD  R0, R0, #10 ; R0 <- 10
ADD  R6, R6, #-1 ;
STR  R0, R6, #0  ; Push 10

LDR  R0, R5, #0  ; Load w
ADD  R6, R6, #-1 ;
STR  R0, R6, #0  ; Push w

JSR  Volta
```

图14-6所示是如上指令执行后，运行时栈的变动情况。其中，参数压入的位置是调用者（Watt）的活动记录空间，而被调用者（Volta）的活动记录则在调用者活动记录之上。

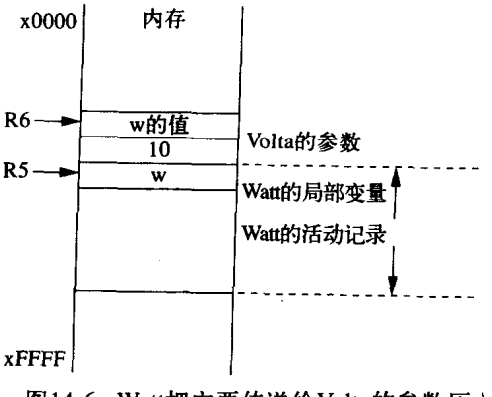


图14-6 Watt把它要传递给Volta的参数压入运行时栈

2. 被调用函数的开始

紧随JSR指令（Watt）之后的第一条指令，是函数Volta中的第一条指令。

被调用函数的初始代码，要完成与调用信息相关的备份操作：

第一个操作就是为返回值预留一个内存位置，即通过栈指针递减，将一个内存空间“压”入栈。之后，被调用函数在返回调用者之前，将返回值填入此内存。

随后，是调用者相关的信息的保存。使得函数完成后，调用者能够顺利重获程序控制权。事实上，就是保存R7中调用者的返回地址（为什么是R7？参考JSR指令），以及R5中调用者的帧指针。备份调用者的帧指针非常重要，我们有时候又称“帧指针”为“动态链”（dynamic link），它是保证调用者在重获控制权后，能恢复对局部变量访问的关键。换句话说，无论是返回地址还是动态链被破坏，都会造成控制权返回给调用者操作的失败。所以，这两个备份信息非常重要。

最后一个操作是，被调用者通过调整R6的值，在栈空间中为它的局部变量分配足够空间，同时设置R5指向这些局部变量的基地址。

总结一下，下面是被调用函数在执行之初所做的各种操作：

（1）为返回值预留空间。返回值在形式参数空间的上方。

（2）将R7的内容（返回地址）压入栈。

（3）将R5的内容（动态链接，即调用者栈指针）压入栈。

（4）在栈空间中，为被调用者自己的局部变量留出空间。设置R5指向局部变量空间的基地址，R6指向栈顶部。

Volta所完成的如上工作，如下面代码所示：

```
Volta:
  ADD R6, R6, #-1   ; Allocate spot for the return value

  ADD R6, R6, #-1   ;
  STR R7, R6, #0    ; Push R7 (Return address)

  ADD R6, R6, #-1   ; Push R5 (Caller's frame pointer)
  STR R5, R6, #0    ; We call this the dynamic link

  ADD R5, R6, #-1   ; Set new frame pointer
  ADD R6, R6, #-2   ; Allocate memory for Volta's locals
```

图14-7所示总结了到目前为止，代码对内存所做的修改。其中，Watt和Volta的活动记录布局非常明显。注意，Volta活动记录中的有些条目是由Watt填写的，即Volta活动记录的参数区。其中，第一个参数是Watt局部变量w的值，第二个参数是数值10。它们入栈的顺序是从右到左，所以w在10的上面。在Volta函数中，将通过名为q和r的变量引用这两个数值。问题是：Volta自己的局部变量的初始值应该是多少？回顾第11章[⊖]所提到的，局部变量的初始值是不确定的。习题14.10就是有关局部变量初始值的。

其中，栈空间中的每个活动记录，其结构布局都相同，即都由函数局部变量、备份信息（返回地址和动态链）、返回值以及参数等内容组成。

3. 被调用函数的结束

在被调用函数完成之后，返回调用者之前，还必须完成一些操作。一是填写返回值，以传递给调用者；二是让出活动记录所占用的内存空间。详细过程如下所示：

（1）如果存在返回值，则填写活动记录的返回值字段。

（2）将局部变量“弹”出栈。

（3）恢复原动态链内容。

⊖ 有关“局部变量的初始值”，应该是在第12章（而不是第11章）。——译者注

(4) 恢复返回地址。

(5) 通过RET指令，返回调用者。

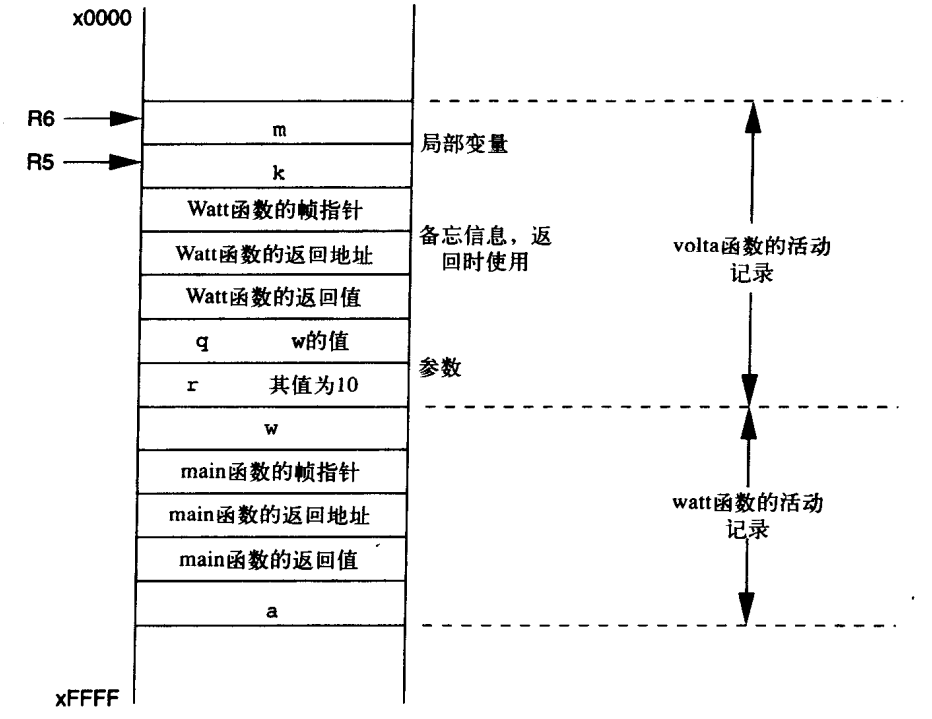


图14-7 Volta的活动记录压入栈后的运行时栈

Volta所完成的以上操作，如下面的LC-3指令所示：

```
LDR  R0, R5, #0  ; Load local variable k
STR  R0, R5, #3  ; Write it in return value slot

ADD  R6, R5, #1  ; Pop local variables

LDR  R5, R6, #0  ; Pop the dynamic link
ADD  R6, R6, #1  ;

LDR  R7, R6, #0  ; Pop the return address
ADD  R6, R6, #1  ;

RET
```

其中，前两条指令的任务是将返回结果（变量k的内容）填入Volta活动记录的返回值字段；然后，通过移动栈指针，“弹”出局部变量；恢复动态链，恢复返回值；最后，返回调用者。

值得一提的是，虽然我们说将Volta的活动记录“弹”出栈，但这些内容仍然留在原内存位置上。

4. 返回调用函数

被调用函数执行RET指令之后，控制权回到调用函数。有关返回值的处理，在有些情况下是没有返回值的（即被调用函数的声明是void类型）；而有些情况下，有返回值，却被忽略；再就是，如我们的程序所示，将返回值由调用函数存入变量（Watt的局部变量w）。

具体来说，执行以下两个操作：

(1) 将返回值（如果有）出栈。

(2) 参数出栈。

JSR之后的LC-3代码如下所示：

```
JSR  Volta

LDR  R0, R6, #0  ; Load the return value
                 ; at the top of stack
STR  R0, R5, #0  ; w = Volta(w, 10);
ADD  R6, R6, #1  ; Pop return value

ADD  R6, R6, #2  ; Pop arguments
```

一旦完成这些代码，则整个函数调用过程就算结束了。之后，调用者恢复原正常执行过程。注意，由于在返回调用者之前，被调用者已恢复调用者的执行环境；所以，对调用者来说，已没有多少工作要做（除了从栈中获取返回值）。

5. 调用规则

最后，我们讨论一个一直避而不谈的话题。在函数执行过程中，R0~R3的作用是存放计算中的临时值，R4~R7默认为系统功能：R4指向全局数据区段，R5为帧指针，R6为栈指针，R7为返回地址。换句话说，函数在执行时，函数调用规则对R4~R7都有明确的使用规定（它们的内容要么不变，要么按照预定的方式改变）。那么寄存器R0、R1、R2和R3怎么处理呢？一般情况下，我们要保证被调用函数不会改变它们的内容。为实现这一点，调用规则可以采用如下两种方法：（1）调用者负责将它们存入自己的活动记录区（保存它们的值），我们称这种方法为“*调用者保存*”规则（参见第9章对该问题的讨论）。函数返回时，调用者再将它们弹出栈，恢复它们的值；（2）另一个方法是，被调用者负责在活动记录区的备份字段添加4个字段，以保存这些寄存器的值，我们称这种方法为“*被调用者保存*”（callee-save）规则。被调用函数在初始化阶段，将R0~R3与R5、R7的内容一起保存在备份字段中，然后在返回前恢复这些寄存器的值。

14.3.3 汇总

如图14-8所示，我们将Watt的函数调用及Volta函数的开始和结尾代码（LC-3），全部放在一起。它们是前几节LC-3代码的汇总，展示了代码的整体结构。只是这段代码比之前的代码稍做了优化，例如，我们将与返回值的压入和弹出相关的栈指针R6的操作，合并到一个指令中。

```
 1  Watt:
 2    ...
 3    AND  R0, R0, #0   ; R0 <- 0
 4    ADD  R0, R0, #10  ; R0 <- 10
 5    ADD  R6, R6, #-1  ;
 6    STR  R0, R6, #0   ; Push 10
 7    LDR  R0, R5, #0   ; Load w
 8    ADD  R6, R6, #-1  ;
 9    STR  R0, R6, #0   ; Push w
10
11    JSR  Volta
12
13    LDR  R0, R6, #0   ; Load the return value at top of stack
14    STR  R0, R5, #0   ; w = Volta(w, 10);
15    ADD  R6, R6, #3   ; Pop return value, arguments
16    ...
17
18  Volta:
19    ADD R6, R6, #-2   ; Push return value
20    STR R7, R6, #0    ; Push return address
21    ADD R6, R6, #-1   ; Push R5 (Caller's frame pointer)
22    STR R5, R6, #0    ; We call this the dynamic link
23    ADD R5, R6, #-1   ; Set new base pointer
24    ADD R6, R6, #-2   ; Allocate memory for Volta's locals
25
26    ...               ; Volta performs its work
27
```

图14-8 C函数调用和返回的LC-3代码

```
26    ...              ; Volta performs its work
27
28    LDR  R0, R5, #0  ; Load local variable k
29    STR  R0, R5, #3  ; Write it in return value slot
30    ADD  R6, R5, #1  ; Pop local variables
31    LDR  R5, R6, #0  ; Pop the dynamic link
32    ADD  R6, R6, #1  ;
33    LDR  R7, R6, #0  ; Pop the return address
34    ADD  R6, R6, #1  ;
35    RET
```

图14-8 C函数调用和返回的LC-3代码（续）

在此，我们按照LC-3的函数调用规则，总结一下函数调用的各个步骤：首先，调用者将各参数顺序压入栈，然后由JSR指令跳转入被调用者；被调用者则在初始化阶段，完成返回值空间分配、保存调用者相关的备份信息、为自己的局部变量分配栈空间等任务；随后，被调用函数完成任务执行；任务完成后，被调用者要将返回值写至为之预留的空间，然后弹出保存的调用者备份信息，返回调用者；最后，调用者读出返回值，并将栈空间中的返回值和参数弹出，然后继续执行。

或许你会想，仅仅是一个函数调用，有必要动用这么多的代码吗？答案是，所有的代码都是必须的，该调用规则已无法再简化了。在调用规则的设计中，存在这样一个要求：任何函数都可以再调用其他函数。这意味着，对于调用者，除了接口信息（即被调用者的返回值类型、参数列表）之外，它不需要知道更多有关被调用者的信息。同样，，被调用者也应该保持与调用者之间的无关性。正因为如此，C函数采用了如上所示的调用规则步骤。

14.4 问题求解

为使函数为我们所用，我们需要将它应用于我们的编程方法学中。在本节中，我们将通过两个例子介绍函数的用法，这两个例子分别展示了函数的不同用法。

在自上而下的算法设计中，函数是个很好的划分点。在问题的分解中，各个“模块”（component）表现为不同的计算任务。每个计算任务都需要一个对应算法，所以各个“模块”自然地对应为各个函数。例如，第一个例子的任务是，将文本的小写转换为大写，该例子非常直观地体现了自顶向下设计中的“模块函数”概念。

函数的另一个作用在于，封装代码中常用的基本操作。我们可以将函数理解为是对编程语言操作符集合的一个扩展（针对特定问题的定制）。所以，第二个例子的任务就是“毕达哥拉斯三角形判断”，我们创建一个计算x^2的基本函数，以辅助整个计算。

14.4.1 例1：大小写转换

本程序的任务是：从键盘读入字符串，将转换结果输出在屏幕上。该程序与第13章中图13-8所示的程序非常相似（输入然后输出），但该程序有个修改：字符在输出之前，先做小写到大写的转换。

我们将以图13-8所示程序为基础，该程序使用一个while循环，不断地从键盘读入字符，然后打印输出。在此框架基础上，添加一段小写字母判断代码（如果是小写，则将它转换为大写）。我们完全可以直接在while循环体内添加这段代码，但考虑到模块的“自包含”（self-contained）特性，我们决定将它包装为一个函数。

转换函数在键盘读入后、屏幕输入前被调用。它的输入是一个字符参数，返回值同样也是字符参数（必定是大写字母或非字母字符）。图14-9所示是该程序的执行流程（flowchart）。我们将其中来自图13-8的流程部分，用阴影部分表示。与原流程相比，这里多了一个负责转换的函数模块（conversion）。

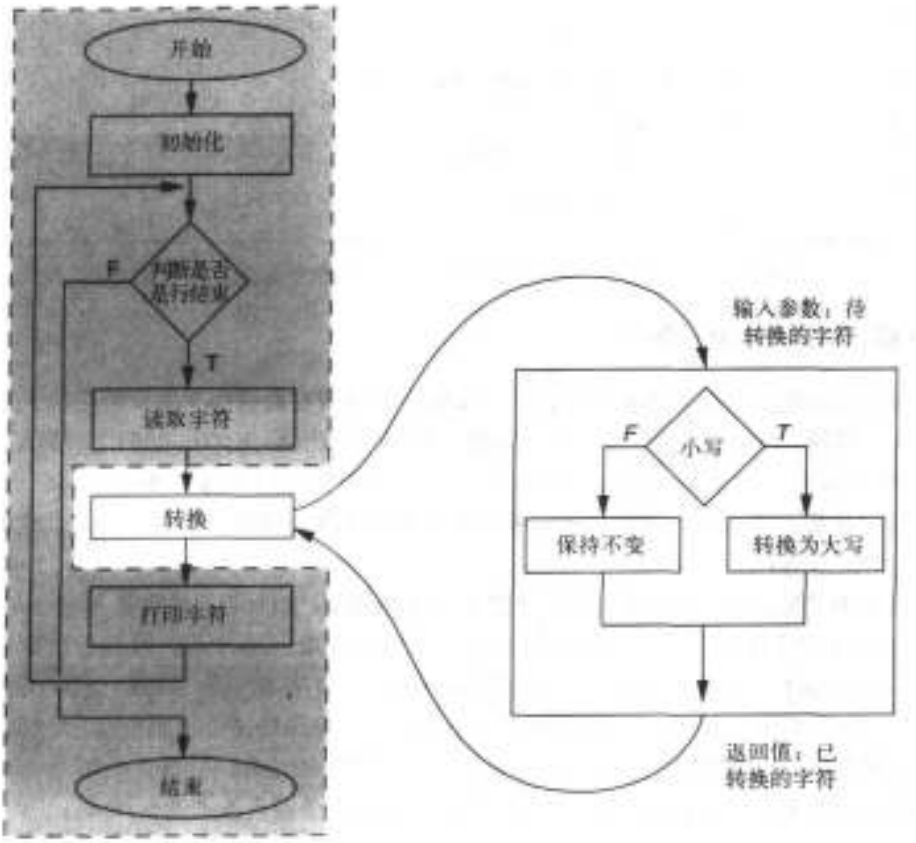


图14-9 字符大写转换的任务分解

图14-10所示是最终的完整C程序。从键盘读入，然后将输入的小写字母转换为大写字母，最后显示输出。如果输入为换行符，程序终止。小写字母转换为大写字母的过程，是由函数ToUpper完成的。注意，函数体内ASCII字符的使用，单引号代表字符（如'A'）的ASCII码值。所以，表达式'a' - 'A'表示字符a的ASCII码值减去字符'A'的ASCII码值。

```
#include <stdio.h>

/* Function declaration */ char ToUpper(char inchar);

/* Function main:                                          */
/* Prompt for a line of text, Read one character,          */
/* convert to uppercase, print it out, then get another    */
int main()
{
  char echo = 'A';          /* Initialize input character */
  char upcase;                  /* Converted character    */

  while (echo != '\n') {
    scanf("%c", &echo);
    upcase = ToUpper(echo);
    printf("%c", upcase);
  }
}

/* Function ToUpper:                                       */
/* If the parameter is lower case return                   */
/* its uppercase ASCII value                               */
char ToUpper(char inchar)
```

图14-10 带有将小写字母转换为大写字母的函数的程序

```
24  {
25    char outchar;
26
27    if ('a' <= inchar && inchar <= 'z')
28      outchar = inchar - ('a' - 'A');
29    else
30      outchar = inchar;
31
32    return outchar;
33  }
```

图14-10 带有将小写字母转换为大写字母的函数的程序（续）

14.4.2 例2：毕达哥拉斯三角形

下面我们尝试通过编程方法求解这样一个问题：计算所有边长小于某个特定输入值的毕达哥拉斯三角形。所谓“毕达哥拉斯三角形”，是指这样3个整数值a、b、c，它们之间满足关系“$c^2 = a^2 + b^2$”。换句话说，a、b、c是直角三角形的三个边（c为斜边）。例如，整数3、4、5就构成一个毕达哥拉斯三角形。我们的问题是，求解所有的毕达哥拉斯三角形（a，b，c），且它们的值都小于用户输入的最大值。

我们将采用“暴力法”寻找所有的三角组合。换句话说，如果用户给定最大值max，我们就检查所有小于max的整数组合，验证它们是否符合毕达哥拉斯三角关系。假设三个数（或边）为sideA、sideB和sideC，将它们遍历1~max（即循环计数）。更准确地说，设置三层for循环，一个遍历sideC，一个遍历sideB，另一个遍历sideA，依次嵌套。循环的核心，是检查三个值是否符合毕达哥拉斯三角关系。如果符合，则将这三个数的组合打印出来。

毕达哥拉斯三角关系的验证方法是，将三个数代入如下关系表达式进行判断：

```
(sideC * sideC == (sideA * sideA + sideB * sideB))
```

其中，我们发现“求平方”是一个基本操作（即代码中很多地方要用到），所以我们将该运算封装为函数Squared，该函数的返回值等于输入参数的平方。因而，将之前的表达式重新表示如下（注意，该代码更清楚地表达了计算意图）：

```
(Squared(sideC) == Squared(sideA) + Squared(sideB))
```

图14-11所示是该问题对应的C程序。事实上，与“暴力法”相比，还存在一种更好的计算方法（你能否修改代码使其更快？）。而在此，我们着重讨论的是函数的使用。

```
1   #include <stdio.h>
2
3   int Squared(int x);
4
5   int main()
6   {
7     int sideA;
8     int sideB;
9     int sideC;
10    int maxC;
11
12    printf("Enter the maximum length of hypotenuse: ");
13    scanf("%d", &maxC);
14
15    for (sideC = 1; sideC <= maxC; sideC++) {
16      for (sideB = 1; sideB <= maxC; sideB++) {
17        for (sideA = 1; sideA <= maxC; sideA++) {
18          if (Squared(sideC) == Squared(sideA) + Squared(sideB))
19            printf("%d %d %d\n", sideA, sideB, sideC);
20        }
```

图14-11 计算毕达哥拉斯三角形的C程序

```
21        }
22      }
23    }
24
25    /* Calculate the square of a number */
26    int Squared(int x)
27    {
28      return x * x;
29    }
```

图14-11　计算毕达哥拉斯三角形的C程序（续）

14.5　小结

本章介绍了C语言中的函数。函数又称为子程序，在很早之前就成为编程语言必备的一个概念。函数的作用，体现为它为编程任务提供了一种创建基本操作的机制。从某种意义上说，它扩展了编程语言的操作和结构。

本章的关键概念如下所示：

- C函数的语法。在C语言中使用函数时，必须先声明函数（通常在代码的最前面），包括函数名、返回值的类型和输入参数的类型及顺序。函数的定义部分是函数的实际代码。函数只在被调用时才执行。函数调用包含形式参数（传递给函数的数值）。
- C函数的底层实现。C函数可以在源文件的任何函数内被调用（甚至是在目标文件中）。为此，我们定义了一种“函数调用规则”，以解决其中存在的一些问题（如函数自身的调用）。这种规则的定义建立在运行时栈的基础上，包括：调用者通过压栈方式传递参数，然后调用函数；参数值写入被调用者活动记录；被调用者完成任务，从栈中弹出活动记录，为调用者留下返回值等。
- 编程中的函数使用。不使用任何函数，也可以完成编程。但这会造成代码难以阅读、维护和扩展等问题，且更容易产生错误。函数为我们提供了一种抽象机制：我们可以为某个特定任务编写一个函数，并对它进行单独调试和测试。

14.6　习题

14.1　试问main函数的重要性是什么？为什么每个程序必须有这个函数？

14.2　根据活动记录的结构，回答下列问题。

a. 动态链的作用是什么？　　b. 返回地址的作用是什么？

c. 返回值的作用是什么？

14.3　根据C函数的语法，回答下列问题。

a. 什么是函数声明？它的作用是什么？　　b. 什么是函数原型？

c. 什么是函数定义？　　d. 什么是形式参数（argument）？

e. 什么是参数（parameter）？

14.4　试指出，如下操作，分别是由调用者还是被调用者来完成的？

a. 将参数值写入活动记录；　　b. 填写返回值；

c. 填写动态链（dynamic link）；

d. 修改R5的内容（指向被调用者的活动记录）。

14.5　试给出如下程序的输出结果，并解释。

```
void MyFunc(int z);

int main()
{
  int z = 2;

  MyFunc(z);
  MyFunc(z);
}

void MyFunc(int z)
{
  printf("%d ", z);
  z++;
}
```

14.6 试问如下程序的输出结果是什么？

```
#include <stdio.h>

int Multiply(int d, int b);

int d = 3;

int main()
{
  int a, b, c;
  int e = 4;

  a = 1;
  b = 2;

  c = Multiply(a, b);
  printf("%d %d %d %d %d\n", a, b, c, d, e);
}

int Multiply(int d, int b)
{
  int a;
  a = 2;
  b = 3;

  return (a * b);
}
```

14.7 阅读如下Bump函数的C代码，并回答问题。

```
int Bump(int x)
{
   int a;

   a = x + 1;

   return a;
}
```

a. 画出Bump的活动记录；

b. 标注如下描述与活动记录的什么字段（entry）相匹配，并解释它们的含义：

（1）局部变量；

（2）形式参数（或输入参数）；

（3）某个指令的地址；

（4）某个数据的地址；

（5）其他。

c. 在Bump的活动记录中，有些字段是由Bump的调用者填写的，有些则由Bump自己填写。请指出Bump自己填写的那些字段。

14.8 试问如下代码的输出结果是什么？阐述其中Swap函数的含义。

```
int main()
{
   int x = 1;
   int y = 2;

   Swap(x, y);
   printf("x = %d   y = %d\n", x, y);
}

void Swap(int y, int x)
{
  int temp

  temp = x;
  x = y;
  y = temp;
}
```

14.9　传递给函数的参数，是在跳转至该函数的JSR指令之前还是之后压入栈中的？请解释其中的道理。

14.10　C语言函数food的LC-3编译代码（部分）如下所示，试回答问题：

```
food:
  ADD R6, R6, #-2   ;
  STR R7, R6, #0    ;
  ADD R6, R6, #-1   ;
  STR R5, R6, #0    ;
  ADD R5, R6, #-1   ;
  ADD R6, R6, #-4   ;
  ...
```

a. 该函数中有多少个局部变量？

b. 该函数有两个整型参数x和y。试写出表达式x + y的运算代码。

14.11　阅读如下代码，回答问题：

```
int main()
{
  int a = 1;
  int b = 2;

  a = Init(a);
  b = Unit(b);

  printf("a = %d   b = %d\n", a, b);
}

int Init(int x)
{
   int y = 2;

   return y + x;
}

int Unit(int x)
{
   int z;

   return z + x;
}
```

a. 该程序的输出是什么？

b. 函数Uint开始执行时，局部变量z的值取决于什么？

14.12　试修改如图14-10所示的代码，使其具备将输入字符转换为小写字母的能力。换句话说，就是要求程序同时输出输入的小写和大写形式。

14.13　试编写函数，打印整数的4进制值（即只使用数字0、1、2、3）。并基于该函数，编写程序，从键盘读入两个整数，然后将两个数及其相加结果以4进制方式显式在屏幕上。

14.14　试编写函数，如果它的第一个输入参数恰好被第二个参数整除，则返回1。然后，基于该函数，编写程序，找出能够被所有小于10的数整除的最小数。

14.15 如下C程序，被编译成LC-3机器语言，且执行前的加载地址为x3000。如果不算跳转至I/O库函数的JSR指令在内，该目标代码中包含3条JSR指令（分别跳入函数f、g和h）。假设，3条JSR指令的地址分别是x3102、x3301和x3304。再假设用户输入为“4、5、6”。试画出当程序从函数f返回时，运行时栈的快照（假设栈的基地址是xEFFF）。

```
#include <stdio.h>

int f(int x, int y, int z);
int g(int arg);
int h(int arg1, int arg2);

int main()
{
   int a, b, c;

   printf("Type three numbers: ");
   scanf("%d %d %d", &a, &b, &c);
   printf("%d", f(a, b, c));
}

int f(int x, int y, int z)
{
   int x1;

   x1 = g(x);
   return h(y, z) * x1;
}

int g(int arg)
{
   return arg * arg;
}

int h(int arg1, int arg2)
{
   return arg1 / arg2;
}
```

14.16 再次引用前面章节介绍的“机器忙”例子，我们以位模式表示16台机器的忙碌状态，即某个bit为0则代表该机器“忙”，为1则代表该机器“空闲”。

a. 试编写函数，根据输入的状态位模式，统计“忙”状态机器的数量。即函数的输入是状态位模式（整型变量），输出是忙碌机器的数量；

b. 试编写函数，根据两个已知状态位模式，判断状态发生变化的机器（如从忙变为空闲，或从空闲变为忙）。函数输出同样是一个简单位模式，相应位为1代表该机器状态发生变化；

c. 试编写程序，从键盘连续读入10个状态位模式，计算平均的忙碌机器数量以及状态发生变化的机器数量。用户输入全1的位模式（所有的机器均为空闲），则代表输入结束。

14.17 a. 试编写一个仿真“4选1开关”（4-*to*-1 *multiplexor*）的C函数。参考如图3-13所示的4选1开关描述；

b. 试编写一个仿真LC-3 ALU行为的C函数。

14.18 在电话机上，标着2，3，4，…，9的按键都有对应的字母。例如，按键2所对应的字母是*A*、*B*和*C*。试编写程序，将一个7位电话号码的所有对应字母序列表示出来。提示：编写一个映射数字和字母的函数。注意：数字1和0没有任何对应字母。

14.19 如下C程序中，同时使用了全局变量和局部变量。试问，它的输出结果是什么？

```
#include <stdio.h>
int t = 1;              /* Global variable */
int sub1(int fluff);
int main ()
{
    int t = 2;
    int z;
```

```
    z = t;
    z = z + 1;
    printf("A: The variable z equals %d\n", z);

    {
        z = t;
        t = 3;

        {
            int t = 4;
            z = t;
            z = z + 1;
            printf("B: The variable z equals %d\n", z);
        }

        z = sub1(z);
        z = z + 1;
        printf("C: The variable z equals %d\n", z);
    }
    z = t;
    z = z + 1;
    printf("D: The variable z equals %d\n", z);
}

int sub1(int fluff)
{
    int i;
    i = t;
    return (fluff + i);
}
```

第15章　测试与调试技术

15.1　概述

1999年12月，美国航空航天局（NASA）控制中心和“火星极地登陆者”（Mars Polar Lander）探测器的通信突然中断，此时探测器即将接近火星表面，但却从此永远失去了联系。火星极地登陆者的任务是探测火星南极区域的土壤和水分。NASA声称探测器极有可能是在着陆过程中与火星表面发生碰撞，而调查者在仔细分析了当时的情况之后，认为原因是控制软件出现了不可恢复的致命性错误，导致探测器在距离火星表面还有40米高空（而不是已着陆）时关闭引擎，从而造成无法挽回的错误。将探测器发射入太空的物理复杂性是惊人的，而控制软件的复杂性也同样如此。软件和机械、电气等子系统有机地组成了整个系统，但软件的正确性更难于把握，因为软件是“不可见的”。它不像其他系统，如推进系统、登陆系统那样易于观察。

今天，软件已无处不在。你的手机、汽车中，甚至本书的排版处理，都是由千万行的软件代码所控制的。由于软件在现今世界中扮演着重要角色，我们必须保证软件按照我们所期望的那样正常运行。程序的设计过程不是自动进行的，在编写过程中难免引入错误。换句话说，程序在编写出来之后，并不意味着它就能正确运行。我们必须在尽可能地测试和调试了所有的可能性之后，才能“假设”它是完整的。

程序员通常花费更多的时间来调试程序，而不是编写程序。专家的调查表明，对一个有经验的程序员而言，调试代码所花的时间远远超过他/她编写代码的时间。代码的编写和测试/调试之间有不可分割的关系，我们将在本章对代码测试和调试中的一些基本概念进行介绍。

测试的目的是“暴露”问题（bug[⊖]），而调试的目的是“解决”问题。测试代码的基本方法，通常是向程序（或局部代码）注入尽可能多的、各种各样的输入条件，以迫使软件暴露bug。以ToUpper函数的测试为例（参考前一章，该函数将输入的字母改为大写并返回）。我们将传入所有可能的ASCII码，然后观察函数的输出结果看看函数是否按规范工作。如果针对特定输入，它产生了错误输出，那么我们就发现了一个bug。我们应该尽可能地在开发阶段就发现bug，而不要让毫无准备的用户在意外情况下遇到错误。仍然以“火星极地登陆者”为例，NASA的工程师们应该让它在地球表面时就出现错误，而不是等到在火星表面40米高度处才出现错误。

基于对程序的阅读和对其执行情况的观察，程序员通常可以确定一些问题的所在。程序调试如同玩拼图游戏，又如同犯罪侦察，程序员必须检查所有可能的线索，才能找出问题的根源。调试的第一任务是收集bug信息，学会怎样“系统地”收集bug信息会对调试大有帮助（如代码中关键变量的执行结果）。

本章将介绍几种在程序中查找和修订bug的技术。首先，我们介绍编程中一些常见错误的分类；然后介绍快速发现这些错误的测试方法；最后，是隔离和修补这些错误的调试技术。同时，我们还将介绍一些“预防”技术，以尽量减小代码出错的可能性。

⊖ bug原意是“臭虫”（如蟑螂等），程序员将程序代码中存在的问题称为bug，称找到错误的过程为“debug”（捉臭虫）。因为这是个非常直观且通俗的称法，且已成为既成事实的“专业名词”，所以本书对bug、debug等不再做翻译。——译者注

15.2 错误类型

如果你了解错误的各种可能类型，无疑可更好地发现和修改错误。下面，我们将遇到的代码错误分为三类：（1）*语法错误*（Syntactic error）最容易处理，因为编译器就能捕捉它们。编译器在将源代码翻译至机器代码时，会提示并准确地指出哪一行出错了；（2）*语义错误*（Semantic error），则很难被修改。语义错误出现的情况是语法上完全正确，但执行结果却与我们期望的不同。语法和语义错误都属于“印刷”错误，即我们在敲键盘的时候出现失误；（3）*算法错误*（Algorithmic error）是指我们解决问题的时候所采用的方法出错了，通常这类问题很难检测，且即使检测到了也很难修改。

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5      int i
 6      int j;
 7
 8      for (i = 0; i <= 10; i++) {
 9          j = i * 7;
10          printf("%d x 7 = %d\n", i, j);
11      }
12  }
```

图15-1 这个程序中存在一个语法错误

15.2.1 语法错误

在C语言中，语法错误（又称为syntax error或parse error）都会被编译器捕获。当编码没有严格遵循C语言规范时，代码在编译之时这些“不规范”之处就会被编译器指出。图15-1的代码中存在一个syntax错误，该错误在编译之时将被标识出来。

变量*i*的声明语句少了分号。作为一个初级C程序员，类似声明语句缺少分号变量未声明这样的错误会经常发生。幸运的是，这类错误很容易发现，因为编译器会检测到并明确指出，所以很容易修改。真正的问题是语法错误修改之后，更难的语义和算法问题却留下了。

15.2.2 语义错误

语义（Semantic）错误和语法错误很类似。错误的原因相同，即我们在编写程序的时候，很难保证头脑和手指完全一致。语义错误并不意味着一定包含了语法错误，即这些程序能顺利地通过编译（即语法是正确的）且能正常执行。但输出的结果分析表明，执行结果与我们期望的不一样。图15-2和图15-1的代码“几乎”完全一样，但存在一个很小的语义错误（而不是语法错误），程序的输出应该是数字7的乘法表。

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5    int i;
 6    int j;
 7
 8    for (i = 0; i <= 10; i++)
 9      j = i * 7;
10      printf("%d x 7 = %d\n", i, j);
11  }
```

图15-2 这段代码中存在一个语义错误

但是，程序执行一遍就暴露问题了，结果只输出了一行（而不是一张表）值。凭着掌握的C语

言知识想想就知道是为什么错。为什么输出是“11 x 7 = 70”？该程序所犯的错被称为“控制流(control flow)错误”，这个问题是程序的控制流即执行顺序与设想的不一致！

再如，图15-3存在一个常见的但非常诡异的语义问题，与局部变量相关。例子与14.2节讨论的求阶乘程序类似。

该程序的目的是：从键盘读取一个值n，然后计算所有小于等于该n的整数之和（如：计算$1 + 2 + 3 + \cdots + n$）。但试着执行一下该程序，会发现执行结果完全出乎所料，为什么它工作不对呢？(提示：请画出该程序执行过程中栈空间的运行变化)

```
 1  #include <stdio.h>
 2
 3  int AllSum(int n);
 4
 5  int main()
 6  {
 7    int in;                      /* Input value              */
 8    int sum;                     /* Value of 1+2+3+..+n      */
 9
10    printf("Input a number: ");
11    scanf("%d", &in);
12
13    sum = AllSum(in);
14    printf("The AllSum of %d is %d\n", in, sum);
15  }
16
17
18  int AllSum(int n)
19  {
20    int result;                  /* Result to be returned    */
21    int i;                       /* Iteration count          */
22
23    for (i = 1; i <= n; i++)     /* This calculates sum      */
24      result = result + i;
25
26    return result;               /* Return to caller         */
27  }
```

图15-3 程序中存在一个涉及局部变量的bug

语义错误特别麻烦，因为：它既不能被编译器检测出来，也很难为程序员所察觉。只有特定输入的触发才会暴露出错误。修改完前面的语义错误之后，再看图15-3代码中的AllSum代码段且传递小于等于0的值或是很大的值给AllSum，此时AllSum返回一个错误结果。因为，这些数值都超出了整型变量result所能表达的范围。改正这个错吧，但输入一个小于1的数，又将会发现另一个bug。

有些错误是在程序运行时，由于非法操作而被系统捕捉到。几乎所有的计算机系统都有安全机制（safeguard）来防止错误扩展，以至于影响其他程序的运行。例如，我们不希望用户程序能够修改存储操作系统的内存空间或直接修改那些会影响其他程序的寄存器（因为这些寄存器会导致整个系统shutdown）。当程序存在这样的非法操作时，操作系统将强制终止该程序，并在屏幕上打印运行时出错（run-time error）信息。例如，将AllSum中的scanf语句改为：

```
scanf("%d", in);
```

注意，本例中“&”符号的作用如第16章所述，是C语言中的一个特殊操作符。如果使用scanf的时候漏了这个“&”符号，就可能引发运行错。因为，它将引发程序去修改它无权访问的内存空间。在后面的章节中，我们会更详细地分析该错误的具体原因。

15.2.3 算法错误

算法错误（Algorithmic）源于程序设计错。即程序本身的运行完全符合设计要求，但设计本

身却存在瑕疵。这类错误相当隐蔽，只有经过大量的测试运行，错误才会暴露。更难的是，即使错误被发现和独立，却很难修改。值得庆幸的是，在设计阶段能合理规划，即写代码前恰当计划，这类错误是可以被减少甚至被消除的。

图15-4的代码中存在一个简单的算法错误，程序任务是输入“年份”，然后判断是否是闰年(leap year)。

第一眼看去，该代码似乎完全正确。每四年出现一次闰年！但事实上不是这样的，每隔一个世纪（100年）就会跳过一个闰年，而每四百年却**不会**如此（如：2000年是一个闰年，但2100年、2200年、2300年却不是闰年）。这段代码在很长一段时间里都表现正常，但特定情况下却出错。我们将这类错归为“算法错误”或设计错。

另一个算法错误的例子是“大名鼎鼎”的2000年问题（Y2K bug）。很多计算机程序会使用尽可能小的内存空间来保存日期值，它们在表达年份的时候只记录年份的最低两个数字，导致2000年和1900年（或1800年、或2100年等）事实上难于区分。于是，在跨越1999年12月31日的这个时刻就存在Y2K问题。例如，你在1999年底从学校图书馆借了一本书，还书期限是2000年初。如果学校图书馆的计算机系统存在Y2K问题，则在还书一刻你将接受重罚。为此，在2000年1月1日前，全世界花费了大量的资金和人力来解决Y2K问题。

15.3 测试

在资深程序员中流传着这么一句话：“任何没有测试过的代码都存在着bug”。这意味着好的测试方法对软件开发是至关重要的。什么是测试？即我们让软件在各种可能的输入模式（模拟真实使用环境）下运行，然后分别检查输出是否正确。事实上，一个软件在发布之前可能要经历几百万次的试运行。

在理想世界中，程序测试方法就是检测程序在所有可能输入下的执行情况，但这几乎是不现实的。例如，一个程序是用来找出处于整数A和B之间的所有素数，如果输入参数A和B都是32位的数值，则共有$(2^{32})^2$种输入组合。假设每秒运行100万个用例（trial），仍然需要50万年才能完成所有测试。显然，测试所有的输入组合不是一个可行的办法。那么，哪些输入组合是需要测试的、哪些又是不必要的呢？如果我们随机选取输入，运气要非常好才能查出程序的bug。通常，软件工程师采用一些系统的方法来测试代码。典型的有，黑盒测试法（black-box）用来测试程序功能是否满足设计规格说明（specification），白盒测试法（white-box）则用来有目的地测试程序实现的各个方面，并保证每行代码都接受了测试。

15.3.1 黑盒测试

黑盒（black-Box）测试关注的是程序的输入和规格说明上的输出是否匹配，不关心其内部的具体实现。换句话说，黑盒测试关心“程序能做什么”（what），而不关心“它是怎么做的”（how）。如图15-3所示，程序AllSum的黑盒测试过程包括：运行程序，输入数据，将程序输出与手工计算结果相比较。如果两个结果不匹配，则意味着要么是程序有错，要么是手算能力有问题。如此进行下去，不停地测试，直到确信它的功能是“可信”的。

然而，对于较大的程序，该测试过程应该是“自动化”（automated）的，这样才能保证在有限时间内能完成足够的测试。这意味着还需要设计一个“测试”程序，该程序能自动地运行被测试程序，并为其提供随机输入，同时匹配输出结果，并重复这项工作。这种自动化方式要比手工测试的效率高得多。

但为了完成黑盒测试的自动化，我们需要一种方法来自动地检查程序的输出是否正确。换句

话说，我们需要开发一个“检查程序器”（checker），它与原程序不同，但却能完成相同的计算任务。当然，如果检查程序和被测程序存在相同的bug，则黑盒法检测必然失败。也正是因为这个原因，编写检测器程序的人是不允许看黑盒里的代码，以保证检测器和被测程序之间的独立性。

15.3.2 白盒测试

然而，对于一个大程序，黑盒测试是不够的。因为在黑盒测试中，无法知道哪些代码被测试过了，哪些还未测试。按照前面的格言所指出的“所有未测试过的代码都可能存在bug”。尤其是有时候在程序的输入及其输出说明不太具体的情况下，黑盒测试变得异常困难执行。例如，黑盒测试一个声音播放器（如MP3播放器）就很难，因为其输出质量的好坏很难明确定义。另外，黑盒测试的前提是整个软件已经开发完成了，这意味着接受测试之前，该程序不仅能通过编译，并且必须已有一些规格说明。

所以，软件工程师们在使用黑盒测试的同时还辅之以白盒（White-Box）测试方法。白盒测试的做法是，将软件内部的各种组件相互独立，然后分别测试各组件是否达到设计要求。例如，根据白盒测试来测试每个函数是否正确执行，此时，程序是实现来划分组件，而不是按它的规格说明划分，我们还可以对函数中的每个循环或其他组成部分使用同样的测试方法。

怎样构建白盒测试呢？大多数测试中，我们可能需要修改原代码。例如，为了测试一个函数是否正确，我们需要添加一些代码；该代码将增加分量运行时间但能重复调用该函数，且每次调用时变换不同的输入和检查相应输出。我们可能在代码中添加大量的printf语句，将内部变量打印出来以观察它的值检查执行是否正确。当代码完成或即将发布的时候，再将这些printf语句删除。

常用的白盒测试技术是在程序中有策略地安放一些检错代码。这些代码将检测能标来程序是否正确运行的条件。当不正常状态被捕捉后，检测代码将打印出一条警告信息，并显示当前状态相关信息，或将程序永久性终止。由于这些检测代码将断定程序执行的（assert）特定条件，因而我们又称这些检测为断言器“assertion”。

例如，assertion可以用来验证代码返回值是否落在规定范围内，如果超出范围，则打印错误信息。以下面的IncomeTax函数为例，该函数的功能是计算“所得税”（Income Tax）是否在规定额度，从这一代码段中可以推论税额是否正确，该函数是按提供给它的特定收入值参数来计算所得税。我们不该付大于收入（很幸运）的所得税，但也不可能付负数值的税，这里假设IncomeTax错断言代码将给出警告信息。

```
tax = IncomeTax(income);

if (tax < 0 || tax > income)
   printf("Error in function IncomeTax!\n");
```

一个完整的测试中，黑盒法和白盒法都是必需的。注意的是，白盒法测试本身是无法测试软件的全部功能。换句话说，即使白盒法测试结果全通过，但仍将存在部分规格说明内容没测试。同样，仅用黑盒法也无法保证每行代码都被测试了。

15.4 调试

发现错误（bug）之后的任务就是修正它，但修正错误通常比发现它更困难。调试一个错需要全面运用我们的能力：观察错误现象，如错误输出，还需要更多其他的信息，如引发错误所在的代码行。然后，基于这些有限的信息，推测出错误的根源。所以，有效地进行调试的关键是收集相关标识错误的信息。这一点和其他行业一样，如侦察犯罪现场收集的证据，或医生在治疗前要充分检查病人的病情以便确诊。

调试中要采用很多方法收集信息，以便诊断出错误。可用方法有简单而快捷的特定方法，也有基于调试工具的系统化技术。

15.4.1 特定方法

最简单的办法是：当你意识到程序中存在错误的时候，仔细检查源代码。有时候根据错误的特征，能很快落实到可能存在错误的代码。这种方法很适合于代码长度较小且你很熟悉的代码。

另一种简单技术则是在代码中插入打印语句。例如，你可以用printf打印出认为比较重要的那些变量的值，以帮助发现bug。你可以在代码不同的地方加入printf语句，查看程序的控制流是否工作正常。例如，若要快速判断一个记数变量控制的循环的循环次数是否正确，那么可以在循环体内放置一条printf语句。对于一个简单程序，类似的特定技术很容易使用也很方便；而对于大程序，错误的数量或原因是错综复杂的，则需要能力更强的复杂技术。

15.4.2 源码级调试工具

特定技术通常无法为bug分析提供足够的信息，此时程序员会选用一个“源码级”（source-level）调试工具来孤立bug。这类工具的特点是：程序可以在受控情况下执行，即借助该工具程序员可以控制/查看程序执行的各种信息。例如，调试工具可以控制程序一条一条语句地单步执行，并随时查看任意一个变量的值（即内存地址和寄存器内容，假如选择查看的话）。事实上，源码级调试工具和第6章介绍的LC-3调试工具很相似，惟一的区别是源码级调试工具操作的是高级语言，而LC-3调试工具操作的是机器语言。

当源码级调试工具用于一个程序上时，被调试的程序必须已编译，同时编译器在编译源码之时会对可执行代码做一些扩充，即在执行代码中加入足够的辅助信息以便协助调试工具工作。在其他时候，调试工具将需要从编译过程获得信息，以便将每个机器语言指令和对应的高级语言程序语句关联；此外，调试工具还需要变量名及其内存位置的相关信息（如符号表），以便程序员可以用源代码中的变量名来查看各个变量的取值。

许多源码级调试工具都很有效且各有各的用户接口。在UNIX和Windows平台上，有很多可用的源码级调试器，各自拥有自己的操作特点。例如，gdb就是一个大多数基于在UNIX平台的、免费源码级调试工具。所有调试器均支持监测程序执行的一组核心的必备操作。其中，许多核心操作其功能和LC-3调试工具的调试特点很类似。本书的重点是介绍其中的核心操作，而不是用户接口，大多数调试器中均支持这些核心操作。

核心调试工具命令可以分为两类：（1）控制操作，用于控制程序的执行。（2）查看操作，用于执行时查看变量或内存内容。

15.4.3 断点

断点（Breakpoints）是指程序执行过程中的一些临时停顿“特殊点”，以便我们检查或修改程序的状态。当代码存在问题的时候一个点对我们检查程序的执行状态非常有用。

例如，我们可以在源代码的某一行或特定函数入口处设置断点。当程序执行到断点处时，整个程序就像“冻结”一样停住了，我们可以查看程序在这一状态中的各种信息。怎样添加断点呢？这取决于调试工具的特定用户接口。有些调试工具只要在代码行上点击一下鼠标即可，而有的调试工具则要求在命令行下输入指定的代码行号来添加断点。

有时候仅当某条件为“真”时暂停程序的执行很有用。这种“条件断点”可用于孤立特定的状态，此时我们猜想有一个bug要发生。例如，我们猜想PerformCalculation函数在输入参数等于16

的时候可能工作不正确，则可以在如下代码中添加条件断点仅在条件为“x = 16”的时候让程序停下来。

```
for (x = 0; x < 100; x++)
   PerformCalculation(x);
```

另外，我们可以在特定条件为“真”的地方设置一个观察点（watch point）来暂停程序的执行。例如，我们可以用一个watch point暂停程序执行，当变量Lastitem等于与4时。该断点可以在执行任意一条使LastItem = 4的语句时触发测试工具，暂停程序的执行。与breakpoint（断点）不同的是，watchpoint不与任何一条代码有关，但可以作用于所有语句行。

2. *单步执行*

一旦调试器遇到断点（或watch point）的时候，它将临时挂起程序，等待我们输入下一个命令。此时，我们可以检查程序的状态，如变量的值，或者继续执行程序。

从一个断点开始一次仅执行一条语句的操作常常很有用，该常用操作过程称为*单步执行*（single-stepping）。LC-3调试工具就具备这么一个命令，它可以一次只执行一条指令，这与源码级调试工具的单步执行功能是类似的。单步命令执行当前语句行之后，则再次挂起程序。大多数调试工具此时会在另一个窗口中将对应的源代码显示出来，（以便我们监视当前程序执行到什么地方了。对一个程序来说单步执行非常有用，特别是在我们猜测bug可能出现的地方设置断点很有用，此时我们可以在猜测位置附近设置一个断点，然后从断点处开始单步执行序，并查看变量的值是否正确。

单步最常见的使用方式是：验证程序控制流是否如我们所希望。例如，可以单步执行一个循环，从检查总的循环次数是否正确；或者单步执行if-else语句，以验证分支条件是否设置正确。

单步执行方式很多，它可以一步跳过一个函数，也可以直接跳至循环的最后一轮。这些单步执行方式使得我们可以快速执行猜想没有错误存在的代码段，并如期暂停在断点处，实现断点和错误点的代码检测。

3. *显示变量内容*

调试艺术更重要是收集信息，以及有逻辑性地减少源代码中的错误。调试工具是当调试大型程序时候能选择要收集的信息的工具。当程序在断点处暂时挂起的时候，我们可以通过查看bug相关的变量内容来收集信息。一般来说，我们可以在断点查看该程序的所有执行状态，包括变量、内存、栈空间、甚至寄存器的内容。怎样做到这一点，由调试工具指定。有些调试工具允许用鼠标指向源代码窗口中的某个变量，触发窗口弹出，以显示该变量的当前值；而有些调试工具则要求你手工输入命令，来指定待查看变量的名字。

我们建议你要多熟悉自己用的源码调试工具。本章结尾我们给出几个问题，以帮助你获得很有用的调试工具使用的若干经验。

15.5 正确的编程方法

懂得怎样测试和调试代码是成为一个优秀程序员的必备条件，但对于一个伟大的程序员来说更重要的是要知道怎样避免错误发生。不良的编程习惯很容易导致bug。学习一些防范出错的编程技术将大大缩短代码的调试时间。记住！与bug的战斗存在于任何一行代码开始编写之前。下面介绍三种在错误发生前就能将错误捕捉住的方法。

15.5.1 明确规格说明

很多bug的起因是程序规格说明（specification）的定义差或不完备。有时候软件规格说明没有完全阐述清楚程序所有可能工作的场景，因此留下了一些条件要程序员自己来解释。例如，第14章的求阶乘的例子：图14-2的代码是求用户输入数字的阶乘。你可以想象该程序的设计说明是这样

描述的“写一个程序，从键盘接入数字，并求其阶乘”。正如所见，这样的说明描述是不完备的，因为如果我们输入一个负数呢？或是输入零呢？或是我们输入一个很大的数字导致溢出呢？面对这些情况，所写出的代码必然不完全正确，因此存在bug。为了改这类错，就需要修改程序规格说明，使得程序在遇到输入为零或小于零的值，或阶乘结果$n!>2^{31}$（即暗示n必须小于等于31）。下面的代码中，我们对第14章的Factorial函数做了修改，添加了输入范围检查代码。一旦输入参数超出正确的操作范围，程序将打印一个警告信息并返回“-1”。

```
1  int Factorial(int n)
2  {
3    int i;                /* Iteration count           */
4    int result = 1;       /* Initialized result        */
5
6    /* Check for legal parameter values */
7    if (n < 1 || n > 31) {
8      printf("Bad input. Input must be >= 1 and <= 31.\n");
9      return -1;
10   }
11
12   for (i = 1; i <= n; i++)  /* Calculates factorial  */
13     result = result * i;
14
15   return result;            /* Return to caller      */
16 }
```

15.5.2 模块化设计

函数是非常利于扩展编程语言的功能。通过函数，我们可以方便地为特定的编程任务添加新的操作和构件。通过这种方式，很自然地使得我们的程序具有模块化的模式。

一旦完成一个函数，我们就可以对它独立测试（如：白盒测试）判断它的工作是否如期完成。相比于完整的程序来说，一个典型的函数仅完成一个小任务。与整个程序相比，它更容易测试。一旦我们独立测试和调试了每个函数，就很高集成它们实现整个程序的设计。

模块化设计概念在系统设计中很重要，即系统的构建可基于一些简单的、已预测试的、工作正常的组件。后面的章节将介绍*库函数*（library）的概念。库函数是一组已预测试的组件的集合。所有程序员可以基于库函数编写自己的程序代码。由于基于模块设计的种种好处，现代编程“严重地”依赖库函数。不仅是在软件设计中而且在电路、硬件、以及计算机系统的各个层面的设计上均使用了模块化设计思想。

15.5.3 预防错误式编程

任何一个老练的程序员都有方法来防止bug“爬入”他们的代码。他们设计代码的方法是在设计阶段就尽量消除能猜到的影响程序的错误发生，即他们边编程边预防错误发生。下面列出常见的预防错误技术，请在编程时尽量采纳它们，以避免错误出现：

- 注释代码：在代码中写上有关它的注释。代码文档的作用不仅仅是告诉别人你的代码是怎么工作的，同时也是你重新审核和反思你自己的代码的过程。在这个过程中，你可能发现你疏漏了一些特例情况或操作条件的处理，而这些疏漏可能酿成大祸。
- 采用统一的编码格式：例如，对齐左括号和右括号，这个习惯会大大减少你犯“因缺少括号引起的错误”。按这一方法，代码中还要注意采用统一的变量名命名方法，变量名应该能传递、暗示变量的内容。
- 不做任何假设：写代码时很容易作简单、无意义假设，但是这些假设最终会引发错误。例如，写函数的时候，我们会假设输入参数总是落在一定的范围内。可如果这个假设没有落实

到设计说明中，则已埋下了错误隐患。写代码时要避免任何假设——或者至少通过断言和现场检查来指明哪些假设不成立。

- 避免全局变量：当有些很有年头的程序员非常喜欢大量使用全局变量时，许多软件工程师大声呐喊尽量不要使用它们。全局变量可能简化一些编程任务，但它们会使代码难以理解和扩展，而且一旦发现bug很难分析出错原因。
- 依赖编译器：大多数好的编译器都提供了一个选项，它可以对你有疑点代码做仔细的检查。例如，是否用赋值操作符“=”代替了相等操作符“==”。当这些检查没通过时，编译器会帮着标识出常见的编译错。(如：变量未初始化）或“对常见的不可用的代码结构进行检查，如果你用的是gcc编译器，可以用“gcc -Wall”使编译器列出所有警告信息。

这里提到的预防技术特别适用于我们之前讨论过的编程概念。在后面的章节中，我们在介绍一些新的编程概念之后，还将讨论如何在编写程序时使用针对于它们的预防技术。

15.6 小结

本章主要介绍了有关发现和修改代码中bug的方法。现代的系统对软件的依赖越来越多，而现代软件也变得非常复杂。为了避免出现因软件bug而发生的诸如“手机死机”、“飞机失控”等问题，软件与规格说明紧密地相一致显得非常重要。本章涉及的关键概念包括：

- 测试：在代码中发现bug并不容易，特别是当程序很大时。软件工程师通常采用系统的测试方法找出软件的错。黑盒法测试法被用来验证程序的执行是否和说明一致；而白盒测试法则被用来有目的地测试程序内部结构，确保每一行代码都经受了一定级别的测试。
- 调试：调试错误需要具备获取信息和推断源代码错误的能力。当一些特定的测试技术允许我们获得少许有关错误的信息时，源码级调试工具已成为软件工程中大多数调试任务的首选调试工具，它使得程序员可以在可控的环境下执行程序，并在执行过程中查看各种数值和状态。
- 正确的编程方法学：有经验的程序员总是在写第一行代码的时候尽量避免bug。但通常bug的真正起源是程序的设计说明，而明确规格说明中各类不严谨的条件，将有效地帮助削减编写出来的代码中的错误，模块化设计使大程序的编写以简单的已预先测试过的函数为基础，从而降低了测试大程序时面临的困难。最后预防错误或编程技术将帮助大量减少导致错误代码出现的情况发生。

15.7 习题

15.1 以下每个程序中都存在一个错误，试做尽可能少的改动来纠正程序中的错误。它们分别通过不同的方法对1到10的整数求和。

a.

```
#include <stdio.h>
int main()
{
   int i = 1;
   int sum = 0;

   while (i < 11) {
      sum = sum + i;
      ++i;
      printf("%d\n", sum);
   }
}
```

b.

```
#include <stdio.h>
int main()
{
   int i;
   int sum = 0;

   for (i = 0; i >= 10; ++i)
      sum = sum + i;
   printf("%d\n", sum);
}
```

c.

```
#include <stdio.h>
int main()
{
   int i = 0;
   int sum = 0;

   while (i <= 11)
      sum = sum + i++;
   printf("%d\n", sum);
}
```

d.

```
#include <stdio.h>
int main()
{
   int i = 0;
   int sum = 0;

   for (i = 0; i <= 10;)
      sum = sum + ++i;
   printf("%d\n", sum);
}
```

15.2 下面的程序存在语法错误，因而编译不通过。假设其中所有变量都事先声明了，试修改程序段中导致编译不通过的错误：

a.

```
i = 0;
j = 0;
while (i < 5);
{
   j = j + 1;
   i = j >> 1;
}
```

b.

```
if (cont == 0)
   a = 2;
   b = 3;
else
   a = -2;
   b = -3;
```

c.

```
#define LIMIT 5;

if (LIMIT)
   printf("True");
else
   printf("False");
```

15.3 下面的C语言程序代码从用户键盘输入的所有正整数中找出最小值。用户通过输入“-1”表示输入结束。数据输入完毕并处理完之后，输出最小值。但是代码中存在一个错误，试找出错误并阐述解决思路和方法。如果需要的话你可以用一个源码级调试工具来帮助查找错误。

```
#include <stdio.h>
int main()
{
   int smallestNumber = 0;
   int nextInput;
```

```
    /* Get the first input number */
    scanf("%d", &nextInput);

    /* Keep reading inputs until user enters -1 */
    while (nextInput != -1) {
       if (nextInput < smallestNumber)
          smallestNumber = nextInput;
       scanf("%d", &nextInput);
    }
    printf("The smallest number is %d\n", smallestNumber);
}
```

15.4 下面的程序从键盘读入一行字符，然后只回显其中的字符和数字(当然空格算字符也回显的)。例如，输入如果是“Let’s meet at 6:00pm”，则回显应该是“Lets meet at 600pm”。但是，程序中存在一个bug，你能发现它并改正该错误吗？

```
#include <stdio.h>
int main()
{
  char echo = '0';

  while (echo != '\n') {
    scanf("%c", &echo);
    if ((echo > 'a' || echo < 'z') &&
        (echo > 'A' || echo < 'Z'))
      printf("%c", echo);
  }
}
```

15.5 试用源码级调试工具监视下面程序的运行：

```
#include <stdio.h>

int IsDivisibleBy(int divisor, int quotient);

int main()
{
   int i;  /* Iteration variable                */
   int j;  /* Iteration variable                */
   int f;  /* The number of factors of a number */


   for (i = 2; i < 1000; i++) {

      f = 0;
      for (j = 2; j < i; j++) {
         if (IsDivisibleBy(i, j))
            f++;
      }
      printf("The number %d has %d factors\n", i, f);
   }
}

int IsDivisibleBy(int divided, int divisor)

{
   if (dividend % divisor == 0)
      return 1;
   else
      return 0;
}
```

并完成以下任务

a. 在函数IsDivisibleBy开始处设置断点，并检查前10次调用的参数值，它们是什么？记录下来。

b. 试给出当内循环for结束后，以及当变量*i*等于660时，变量*f*的值？

c. 试分析，指出该程序如何修改效率更高。提示：监视当函数IsDivisibleBy返回值为1的时候给参数输入的数值。

15.6 使用源码级调试工具，确定在函数Mystery的参数取什么值时，该函数会返回零值？

```
#include <stdio.h>

int Mystery(int a, int b, int c);

int main()
{
   int i;          /* Iteration variable       */
   int j;          /* Iteration variable       */
   int k;          /* Iteration variable       */
   int sum = 0;    /* running sum of Mystery   */

   for (i = 100; i > 0; i--) {
     for (j = 1; j < i; j++) {
       for (k = j; k < 100; k++)
         sum = sum + Mystery(i, j, k);
     }
   }
}

int Mystery(int a, int b, int c)
{
  static max = 1000;
  int out;

  out = 3*a*a + 7*a - 5*b*b + 4*b + 5*c ;

  return out;
}
```

15.7 如下程序是一个小型航线（只有一架飞机）的订票系统，该飞机有SEATS个乘客座位。该程序接收来自网上的订票请示。命令*R*代表“请求座位”，如果有空位则预定成功；如果没有位置了，则订票申请失败。随后，预定成功的旅客通过*P*命令购票。这意味着每个*P*命令之前都必须先有个*R*命令；但反之，每个*R*命令并不一定产生购票操作（即*P*命令）。命令*X*则代表结束程序。但是，在这个程序中存在一个致命的设计错误。请找出错误，并予以纠正。

```
#include <stdio.h>

#define SEATS 10

int main()
{
  int seatsAvailable = SEATS;
  char request = '0';

  while (request != 'X') {
    scanf("%c", &request);

    if (request == 'R') {
      if (seatsAvailable)
        printf("Reservation Approved!\n");
      else
        printf("Sorry, flight fully booked.\n");
    }

    if (request == 'P') {
      seatsAvailable--;
      printf("Ticket purchased!\n");
    }
  }

  printf("Done! %d seats not sold\n", seatsAvailable);
}
```

第16章　指针和数组

16.1　概述

本章将介绍（实际上是再次介绍）两个简单但功能非常强大的编程构件——指针和数组。在编写LC-3汇编程序时，我们就使过指针和数组。现在讨论的则是怎样在C语言里测试它们。

指针是内存对象（如变量）的简单地址。用指针，我们可以间接访问这些对象，它有一些很有用的功能。例如，使用指针，使得函数可以修改调用者传递进来的参数；使用指针，还可以在程序运行过程中，使复杂的数据结构增长或缩小（像运行时栈的变化一样）。

数组是内存中一组用来存储数据序列的连续的空间，例如，在本书前半部的少数LC-3代码中，我们曾用一个有序排列的字符序列代表一个字符文件。我们称这样的字符序列为“字符数组(array)”。为了访问数组中的某个特定数据项，我们还需要指定我们所要的元素。稍后我们将看到，a[4]代表的是a数组的第5个元素（因为起始元素的编号为0）。数组之所以有用，是因为通过它我们可以处理一组数据，如向量、矩阵、列表和字符串等，这些对象自然地表示了真实世界的某些对象。

16.2　指针

我们以一个经典的指针例子开始我们的讨论。图16-1所示的C程序中，函数Swap的作用是交换两个参数的数值。main函数将两个参数valueA和valueB传递给Swap函数，其中valueA = 3，valueB = 4。当控制权从Swap函数返回到main函数时，我们期望valueA和valueB的值已经互换。但是，当我们编译并执行该代码之后，却发现两个变量的值并未改变（没有相互交换）。

```
1  #include <stdio.h>
2
3  void Swap(int firstVal, int secondVal);
4
5  int main()
6  {
7    int valueA = 3;
8    int valueB = 4;
9
10   printf("Before Swap ");
11   printf("valueA = %d and valueB = %d\n", valueA, valueB);
12
13   Swap(valueA, valueB);
14
15   printf("After Swap ");
16   printf("valueA = %d and valueB = %d\n", valueA, valueB);
17 }
18
19 void Swap(int firstVal, int secondVal)
20 {
21   int tempVal;          /* Holds firstVal when swapping */
22
23   tempVal = firstVal;
24   firstVal = secondVal;
25   secondVal = tempVal;
26 }
```

图16-1　Swap函数：试图交换两个参数的值

让我们看看Swap函数执行时，栈内容的变化，或许能分析出其中的问题所在。图16-2所示为运行时栈的状态，表示的是函数完成前栈内容的变化（即第25行语句已执行，但还未真正返回main函数之时）。我们发现，Swap已改变它的活动记录中firstVal和secondVal参数的局部量数值，而当Swap函数最终返回控制权给main函数之时，Swap的活动记录已从栈中弹出，这些被改变过的数据值也随之丢失了。所以，从main函数角度来看，两个值并未被交换。程序有bug。

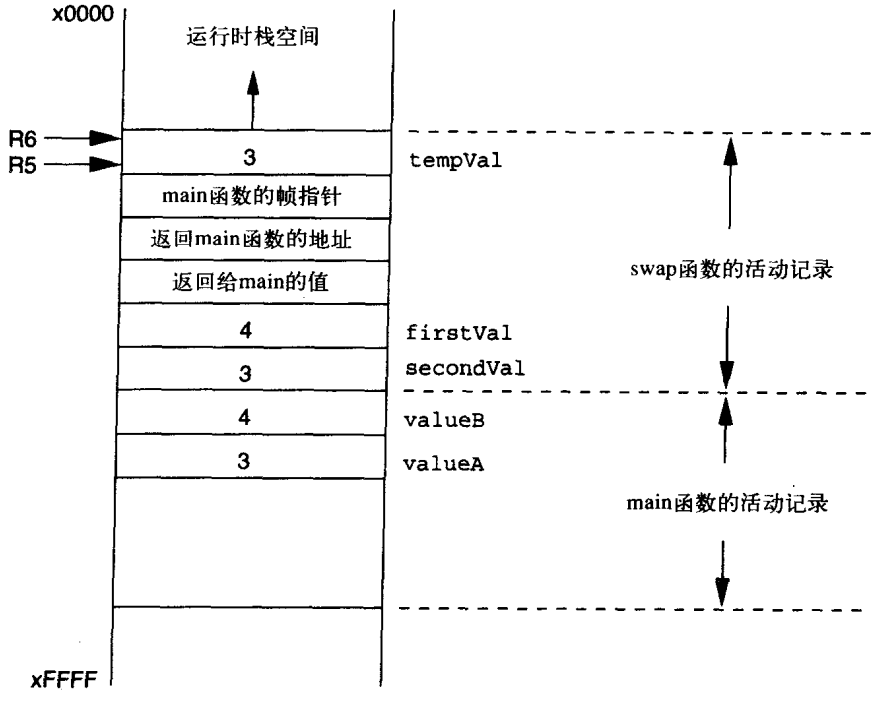


图16-2 Swap函数正要返回控制权到main函数之前运行时的栈快照

在C语言中，调用函数总是把*数据值*传递给被调用函数的参数，C语言将函数调用语句中出现的参数看做是一个表达式，并计算表达式调用之后将计算结果压入运行时的栈实现表达式数值向被调用函数的传递。如果Swap要修改传递给它的参数，它必然需要访问调用函数的活动记录，这意味着它必须访问参数存放空间，以修改它的值。即Swap函数需要访问的是valueA和main函数中的地址（以修改它们的值）。正如在后面几节将看到的，指针和它相关的运算符可以满足这个要求。

16.2.1 声明指针变量

指针变量的内容是一个内存对象的地址（如变量）。或者说，指针指向它的内容指到的变量。与指针变量相关的另一个要素，是它所指向的对象的*类型*（type）。例如，一个整型指针变量指向一个整型变量。在C语言中，指针变量的声明语句如下所示：

```
int *ptr;
```

其中，被声明变量名为ptr，它指向的是一个整数。星号（*）代表之后的标识符是一个指针变量名。C程序员常常说：ptr是int *类型的变量。类似的，我们也可以声明：

```
char *cp;
double *dp;
```

其中，变量cp指向一个字符，dp指向一个双精度浮点数。指针变量的初始化方式和其他类型

变量相似。如果指针变量被声明为局部变量，它将不会被自动初始化。

开始，使用符号“*”声明指针变量的语法看起来有些奇怪。但是，如果看过所有的指针操作符之后，我们就能明白语法背后的原因了。

16.2.2 指针运算符[⊖]

在C语言中，指针相关的运算符有两个：地址运算符（&）、间接引用运算符（*）。

1. 地址运算符（&）

地址运算符（&），生成一个它的操作数的内存地址，该操作数肯定是一个内存对象，如变量。如下代码序列中，指针变量ptr将指向整型变量object。在第二个赋值语句右边的表达式将生成object的内存地址。

```
int object;
int *ptr;

object = 4;
ptr = &object;
```

让我们看看这段LC-3汇编代码。其中，两个声明的变量都是局部变量，可以被放入栈空间。回顾R5，（基地址指针），它指向声明的第一个局部变量，这种情况下是object。

```
AND  R0, R0, #0    ;   Clear R0
ADD  R0, R0, #4    ;   R0 = 4
STR  R0, R5, #0    ;   Object = 4;

ADD  R0, R5, #0    ;   Generate memory address of object
STR  R0, R5, #-1   ;   Ptr = &object;
```

图16-3给出了“ptr = &object;”语句执行之后，函数中代码包含的活动记录的快照。为了让图示更加形象，我们给出了每个内存空间的地址（随意假定起始地址为xEFF0）。而此时，基地指针R5指向的是xEFF2。注意，object的内容是数值4，而ptr的内容则是object的内存地址。

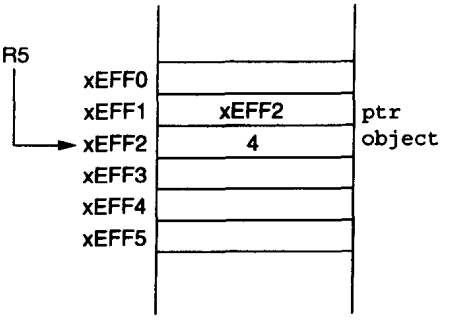


图16-3 语句ptr = &object执行后包含object和ptr的运行时栈结构

2. 间接引用符（*）

第二个指针运算符被称为“间接引用符”（indirection）或“解引用符”（dereference），符号为“*”（发音为“星”（star））。它的作用是允许我们间接地访问内存对象的数值。例如，表达式“*ptr”访问的是指针变量ptr所指向的值。回顾前例：*ptr引用存储在变量object中的值。所以，*ptr和object是可以互换着用，我们可以将前面的代码修改如下：

```
int object;
int *ptr;

object = 4;
ptr = &object;
*ptr = *ptr + 1;
```

也就是说，本质上“*ptr = *ptr + 1;”和“object = object + 1;”是等价的。与其他变量类型一样，*ptr故在赋值运算符的不同位置，代表的含义也不同。在运算符右边代表该地址中的值（这儿是4），在左边代表将被修改的变量的地址空间（这儿是object的地址）。让我们再看代码中最后一

⊖ 有关“operator”的翻译。在中文翻译中，通常称+、−、*、/等符号为“运算符”，而其他如&、~、>>等为“操作符”。而在英文中，它们都被称做“operator”。而本文译者也认为，它们是同类的，都是对变量或数据对象的一种运算。所以，在此将“&”和“*”都译为运算符（而不是操作符）。——译者注

条语句对应的LC-3汇编代码。

```
LDR  R0, R5, #-1  ;   R0 contains the value of ptr
LDR  R1, R0, #0   ;   R1 <- *ptr
ADD  R1, R1, #1   ;   *ptr + 1
STR  R1, R0, #0   ;   *ptr = *ptr + 1;
```

注意，如果C语言的最后一条语句是“object = object + 1;”，所对应的汇编代码是不同的。由于使用了指针，编译器将为等号右边的间接引用符生成为两条LDR指令，一个加载ptr所指的内存地址，另一个则独立存储在那了，地址中的数值。而针对等号左边的间接引用，编译器产生了一条指令“STR R1, R0, #0”。如果这条语句是“object = *ptr + 1;”，编译器则生成“STR R1, R5, #0”指令。

16.2.3 指针传递一个引用

使用地址符和间接引用符，我们可以将如图16-1所示的Swap函数做一个修改，以完成两个输入参数值的互换。图16-4所示是修改后的swap程序，我们将修改后的Swap函数命名为NewSwap。

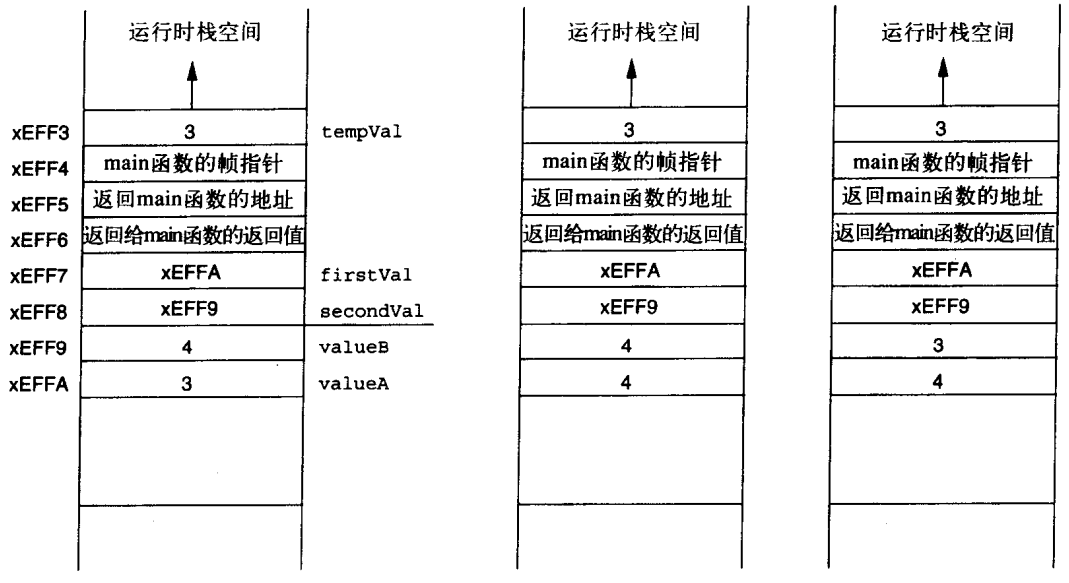
```
 1  #include <stdio.h>
 2
 3  void NewSwap(int *firstVal, int *secondVal);
 4
 5  int main()
 6  {
 7    int valueA = 3;
 8    int valueB = 4;
 9
10    printf("Before Swap ");
11    printf("valueA = %d and valueB = %d\n", valueA, valueB);
12
13    NewSwap(&valueA, &valueB);
14
15    printf("After Swap ");
16    printf("valueA = %d and valueB = %d\n", valueA, valueB);
17  }
18
19  void NewSwap(int *firstVal, int *secondVal)
20  {
21    int tempVal;          /* Holds firstVal when swapping */
22
23    tempVal = *firstVal;
24    *firstVal = *secondVal;
25    *secondVal = tempVal;
26  }
```

图16-4 NewSwap函数：交换了两个参数的值

修改的第一项内容是NewSwap参数的类型，即将它们从int改为指向整型的指针（int *）。换句话说，这的两个参数是两个可以被交换内容的变量的地址。在NewSwap函娄中，我们使用间接引用符*获得指针所指向的数值。

此时，当从main函数调用NewSwap时，我们需要为两个要被交换的变量提供内存地址，而不是它们的数值（前一个版本的代码中给的是数值）。在此关键的是运算符“&”。图16-5所示是执行NewSwap函数时函数的各条指令执行时的栈内容的快照。图16-5a~图16-5c三个子图分别对应第23、24和25行语句执行后的栈内容。

通过设计，C语言用数值从调用函数传递信息给被调用函数，即计算调用语句中的实数表达式，然后把计算结果传给被调用函数。但是，在NewSwap中，我们用地址符“&”为两个实参创建调用时的引用。把参数当成引用值传递时，它的地址被传入被调函数，为保证该效果实现，实参必须是变量或其他内存对象（例如，它必须有一个地址）。被调用函数可以通过间接引用符（*）访问（和修改）该对象的原始值。



a）第23行　　b）第24行　　c）第25行时的运行时栈内容

图16-5　NewSwap函数执行语句

16.2.4　空指针

有时候，我们希望一个指针不指向任何东西。为什么呢？在第19章介绍了动态数据结构（如链表）之后，你将明白其中原委。而现在，我们只介绍一个不指向任何内容的指针。我们称一个不指向任何东西的指针为“空指针”。在C语言中，我们用以下赋值完成这一设计：

```
int *ptr;

ptr = NULL;
```

其中，我们将空值赋给了指针变量ptr。在C语言中，NULL是一个特别定义的预处理宏，它含有一个“非空指针”不可能包含的值。例如，在特定系统中，NULL可以被定义为数值0，因为地址0不存在任何有效的内存对象。

16.2.5　语法

现在可以回答在第11章遗留下的一个问题了。我们曾介绍过I/O函数库中的scanf函数：

```
scanf("%d", &input);
```

其中，scanf函数的任务是，从键盘读取一个十进制数，并赋值给变量input，它所需要的是input的地址（而不是它的值）。为此，我们使用了地址符（&）。换句话说，如果我们删去其中的地址符号，程序的运行将出错而中止。你能给这个出错现象一个合适的解释吗？即为什么传递的不是地址，scanf就不能正常工作呢？

下面，我们正式介绍指针的声明语法。如下所示：

```
type *ptr;
```

其中，type代表任意已定义或自定义的数据类型（如int、char、double等），ptr代表任意合法（而不是C关键词）的变量标识符。该声明语句表示，我们声明了这样一个变量，当对它做*操作时，产生的结果是一个type类型变量。即*ptr的类型是type的。

同样，我们也可以将一个函数的返回值声明为指针类型（后续章节将介绍它的用处）。例如，

我们可以声明这样一个函数：int *MaxSwap()。

与其他C语言运算符一样，地址符和间接引用符也遵循“优先级”和“关联顺序”等规则。有关各种运算符之间的优先级和结合规则，如表12.5所示。值得一提的是，此处的两个指针运算符都拥有很高的优先级别。

16.2.6 指针例程

本节最后，是一个使用指针的示例程序。如果我们要写一个程序，计算整型除法的商（quotient）和余数（remainder）。即计算“被除数/除数”和“被除数%除数”，其中被除数和除数的值都是整数。该程序的结构非常简单，只有一个顺序结构（也就是说，不需要条件、循环等高级结构）。问题的难度在于，我们希望让一个C函数同时返回“商”和“余数”这两个计算结果。

通常的做法可以是，通过函数返回值将计算结果传回调用者。但是，这种方法一次只能返回一个输出值（如“商”）。例如，在计算除法商的函数结尾处，执行语句“return dividend / divisor;”即可传回该值。如果要将多个数值返回给调用者，则可以采用“传址”机制（即指针变量）。

如图16-6所示，函数IntDivide有4个参数：2个整型、2个整型指针。在该函数中，我们将第1个参数x与第2个参数y相除。相除结果的整数部分赋给quoPtr指向的内存位置，余数部分则赋给remPtr指向的内存位置。

```
#include <stdio.h>

int IntDivide(int x, int y, int *quoPtr, int *remPtr);

int main()
{
  int dividend;        /* The number to be divided      */
  int divisor;         /* The number to divide by       */
  int quotient;        /* Integer result of division    */
  int remainder;       /* Integer remainder of division */
  int error;           /* Did something go wrong?       */

  printf("Input dividend: ");
  scanf("%d", &dividend);
  printf("Input divisor: ");
  scanf("%d", &divisor);

  error = IntDivide(dividend,divisor,&quotient,&remainder);

  if (!error)          /* !error indicates no error     */
    printf("Answer: %d remainder %d\n", quotient, remainder);
  else
    printf("IntDivide failed.\n");
}

int IntDivide(int x, int y, int *quoPtr, int *remPtr)
{
  if (y != 0) {
    *quoPtr = x / y;                  /* Modify *quoPtr */
    *remPtr = x % y;                  /* Modify *remPtr */
    return 0;
  }
  else
    return -1;
}
```

图16-6 函数IntDivide：计算整数除法的整数和余数部分。如果除数为0，则返回−1

其中，函数IntDivide的返回值代表的是函数本身的执行状态，例如，如果除数是0，则返回数值−1（告诉调用者函数执行失败）；如果返回0，则告诉调用者计算过程正常。在main函数中，则根据该返回值来判断商和余数里的值是否有效。在调用者和被调用者之间，通过返回值代表被调

用函数的执行状况，是个非常好的“预防式编程”习惯，它可以让我们探知函数调用过程中的各种错误原因。

16.3 数组

假设有这样一个程序，我们用它记录《计算机工程》课程中50位学生的考试成绩。存储成绩数据的最简单方法，就是声明一个对象（如examScore），并将50个不同的整数值都存储在其中。然后，我们可以通过索引方式，访问该对象中任一个数据项（考试成绩）。所谓“索引”，就是该数据项距离起始对象的偏移量。例如，examScore[32]代表第33个学生的成绩（第1个学生的成绩是examScore[0]，examScore是一个整型数组对象）。“数组”，是指连续存储在内存中的一组数据项的集合。在数组内，所有的数据项拥有相同的类型（如int、char等）。

数组尤其擅长表示一组连续的数值序列。在现实中，很多对象都具有这样的性质，如一门课程的学生成绩。无疑，数组在计算机编程中，是一个非常重要的数据结构。例如，一个程序要从键盘依次读入100个数，然后按从小到大的升序排列它们。那么，数组是最佳的内存存储方式。如果使用之前学过的简单变量方式来存储这些数据，几乎是不可能的事情。

16.3.1 数组声明

首先，看一下C语言的数组声明方法。与其他变量一样，数组需要一个相关的“类型”。这里，类型代表存储在数组中各数值的属性。如下所示，声明的是一个包含10个整数的数组：

```
int grid[10];
```

其中，关键词int代表数组中各数据项的类型是整数类型，grid代表数组名，方括号（[]）代表该变量是一个数组，10代表该数组包含的数据项个数（这些数据项在内存中是连续存放的）。图16-7所示是grid在内存中的空间分配情况。其中，第1个元素grid[0]被分配在最低内存地址处，最后一个元素grid[9]被分配在最高内存地址处。

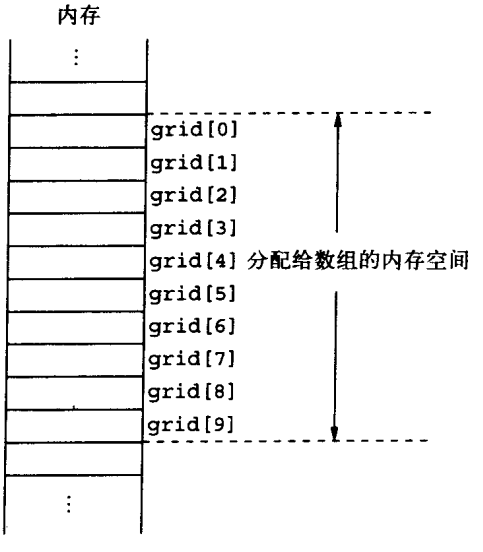


图16-7 grid数组的内存空间布局

其中，如果数组grid是一个局部变量，则该空间对应运行时栈空间。

随后的问题是，怎样存取数组中的不同数据项？如图16.7所示，数组中第一个元素的编号为0，最后一个元素的编号为9。所以，存取数据项时，只需要在方括号中标出该数据项对应的索引编号

即可。例如：

```
grid[6] = grid[3] + 1;
```

该语句表示，读取grid的第4个数据项（记住，0是起始编号），加1，然后将结果存入第7个数据项。下面是该语句的LC-3汇编代码。其中，假设grid是运行时栈中的局部变量（即活动记录基指针R5指向grid[9]）。

```
ADD  R0, R5, #-9  ;  Put the base address of grid into R0
LDR  R1, R0, #3   ;  R1 <-- grid[3]
ADD  R1, R1, #1   ;  R1 <-- grid[3] + 1
STR  R1, R0, #6   ;  grid[6] = grid[3] + 1;
```

其中，第一条指令负责数组基地址的计算（即grid[0]的地址），放入R0。所谓“数组的基地址”，就是该数组第1个元素的地址。之后，通过“基地址+索引”的方式，就可以访问数组中的任意元素。

数组之所以功能强大，原因之一是，索引值（整型）可以是一个C表达式。如下所示：

```
grid[x+1] = grid[x] + 2;
```

该语句对应的LC-3汇编代码如下所示。假设x是栈空间中的另一个整型变量（grid上方）。

```
LDR  R0, R5, #-10 ;  Load the value of x
ADD  R1, R5, #-9  ;  Put the base address of grid into R1
ADD  R1, R0, R1   ;  Calculate address of grid[x]
LDR  R2, R1, #0   ;  R2 <-- grid[x]
ADD  R2, R2, #2   ;  R2 <-- grid[x] + 2

LDR  R0, R5, #-10 ;  Load the value of x
ADD  R0, R0, #1   ;  R0 <-- x + 1
ADD  R1, R5, #-9  ;  Put the base address of grid into R1
ADD  R1, R0, R1   ;  Calculate address of grid[x+1]
STR  R2, R1, #0   ;  grid[x+1] = grid[x] + 2;
```

16.3.2 数组应用

我们先介绍一个简单的C程序，“对两个数组相加”。所谓“两个数组相加”，是将两个数组的对应数据项分别相加。其中，每个数组代表一门课的成绩表。如果要计算每个学生各门课成绩的总和，则执行计算如“Total[i] = Exam1[i] + Exam2[i]”。如图16-8所示的C程序，读入两个数组（10个数据项）的内容，然后将相加结果存入另一个数组（也是10个数据项），最后打印输出结果。

```
 1  #include <stdio.h>
 2  #define NUM_STUDENTS 10
 3
 4  int main()
 5  {
 6    int i;
 7    int Exam1[NUM_STUDENTS];
 8    int Exam2[NUM_STUDENTS];
 9    int Total[NUM_STUDENTS];
10
11    /* Input Exam 1 scores */
12    for (i = 0; i < NUM_STUDENTS; i++) {
13      printf("Input Exam 1 score for student %d : ", i);
14      scanf("%d", &Exam1[i]);
15    }
16    printf("\n");
17
18    /* Input Exam 2 scores */
19    for (i = 0; i < NUM_STUDENTS; i++) {
20      printf("Input Exam 2 score for student %d : ", i);
21      scanf("%d", &Exam2[i]);
22    }
23    printf("\n");
24
```

图16-8 计算两个10元素数组相加的程序

```
    /* Calculate Total Points */
    for (i = 0; i < NUM_STUDENTS; i++) {
      Total[i] = Exam1[i] + Exam2[i];
    }

    /* Output the Total Points */
    for (i = 0; i < NUM_STUDENTS; i++) {
      printf("Total for Student %d = %d\n", i, Total[i]);
    }
}
```

图16-8 计算两个10元素数组相加的程序（续）

其中，有关编程风格上的建议是，注意其中预处理宏NUM_STUDENTS的使用，它是代表输入集合大小的一个常数。这是一种常见的预处理宏用法，我们常在C的源文件（或头文件）开始几行中，看到类似的定义。假设，现在我们要增大数组的大小（比如注册学生数目发生变化），我们所要做的事情，只是修改这个宏定义（只有一处），然后重新编译程序即可。如果没有使用宏，则改变数组大小这样一件事，意味着在代码的各个数组使用处都要做出修改。并且，稍有疏忽，遗漏某个修改点，程序就无法正常工作。所以，此处采用预处理宏来定义数组的大小，又是一个好的编程习惯。

下面我们看一个更复杂的数组例子。如图16-9所示，程序从键盘连续读入一组十进制数（总共有MAX_NUMS个）；然后统计每个数字在序列中出现的次数；最后，打印每个数字及其重复出现次数。

```
#include <stdio.h>
#define MAX_NUMS 10

int main()
{
  int index;              /* Loop iteration variable      */
  int repIndex;           /* Loop variable for rep loop   */
  int numbers[MAX_NUMS];  /* Original input numbers       */
  int repeats[MAX_NUMS];  /* Number of repeats            */

  /* Get input */
  printf("Enter %d numbers.\n", MAX_NUMS);
  for (index = 0; index < MAX_NUMS; index++) {
    printf("Input number %d : ", index);
    scanf("%d", &numbers[index]);
  }

  /* Scan through entire array, counting number of       */
  /* repeats per element within the original array       */
  for (index = 0; index < MAX_NUMS; index++) {
    repeats[index] = 0;
    for (repIndex = 0; repIndex < MAX_NUMS; repIndex++) {
      if (numbers[repIndex] == numbers[index])
        repeats[index]++;
    }
  }

  /* Print the results */
  for (index = 0; index < MAX_NUMS; index++)
    printf("Original number %d. Number of repeats %d\n",
           numbers[index], repeats[index]);
}
```

图16-9 统计数组里每个数值重复次数的C程序

该程序使用了numbers和repeats两个数组（都是MAX_NUMS个整数项）。其中，numbers代表输入数字序列，repeats代表各数字在numbers中重复出现的次数。例如，如果numbers[3] = 115，而在整个输入序列中，曾4次输入“115”（即在数组numbers中有4个115），则repeats[3]等于4。

程序中包含三个大循环体。第一个和最后一个for循环比较简单，分别处理键盘输入和打印输出。中间的循环体是一个*嵌套循环*（见13.3.2节），实际上包含了两层循环。

中间的for嵌套循环负责统计每个元素在全数组中出现的次数。其中，外层循环的计数变量index依次从0~MAX_NUMS，即通过index扫描整个数组，从第1个元素numbers[0]到最后一个元素numbers[MAX_NUMS]；内层循环的计数范围也是0~MAX_NUMS，该层循环的任务也是再次扫描整个数组，统计的是在数组中存在多少个元素，其值与外层循环当前选中的元素（numbers[index]）相同。每匹配到一个相同值（即numbers[repIndex] == numbers[index]），则repeats中对应项的值就加1（即repeats[index]++）。

16.3.3 数组参数

在函数之间传递数组也非常有用，因为它将使得函数具备运算数组的能力。假设，我们要编写一组函数，分别求解整型数组的平均值和中间值，则我们可以：（1）传递数组值；（2）传递该数组的地址。如果该数组的数据项数目很大，则将该数组从一个活动记录区拷贝到另一个活动记录区是件非常耗时的操作。所幸的是，C语言传递数组时，天生采用的就是“传址”方式。如图16-10所示，函数Average需要的是一个数组参数（整数类型）。

```
 1  #include <stdio.h>
 2  #define MAX_NUMS 10
 3
 4  int Average(int input_values[]);
 5
 6  int main()
 7  {
 8    int index;               /* Loop iteration variable  */
 9    int mean;                /* Average of numbers       */
10    int numbers[MAX_NUMS];   /* Original input numbers   */
11
12
13    /* Get input */
14    printf("Enter %d numbers.\n", MAX_NUMS);
15    for (index = 0; index < MAX_NUMS; index++) {
16      printf("Input number %d : ", index);
17      scanf("%d", &numbers[index]);
18    }
19
20    mean = Average(numbers);
21
22    printf("The average of these numbers is %d\n", mean);
23  }
24
25  int Average(int inputValues[])
26  {
27    int index;
28    int sum = 0;
29
30    for (index = 0; index < MAX_NUMS; index++) {
31      sum = sum + inputValues[index];
32    }
33
34    return (sum / MAX_NUMS);
35  }
```

图16-10 参数为数组的函数的例子

其中，main函数在调用Average时，我们要把和数组标识符numbers相关的值传给它。注意，我们这里没有像正常使用数组那样用方括号。在C语言里，数组的名字引用数组首元素的地址。也就是说，名字numbers等于&numbers[0]。numbers的类型和int *很像，它是一个包含了整数的内存空间的地址。

numbers作为函数Average的参数，在调用时numbers的地址被压入Average的栈。而在函数Average中，将该数组地址赋给变量inputValues，并用标准的数组符号方式访问原始数组元素。图16-11所示是Average在return语句（第34行语句）执行之前的运行时栈快照。

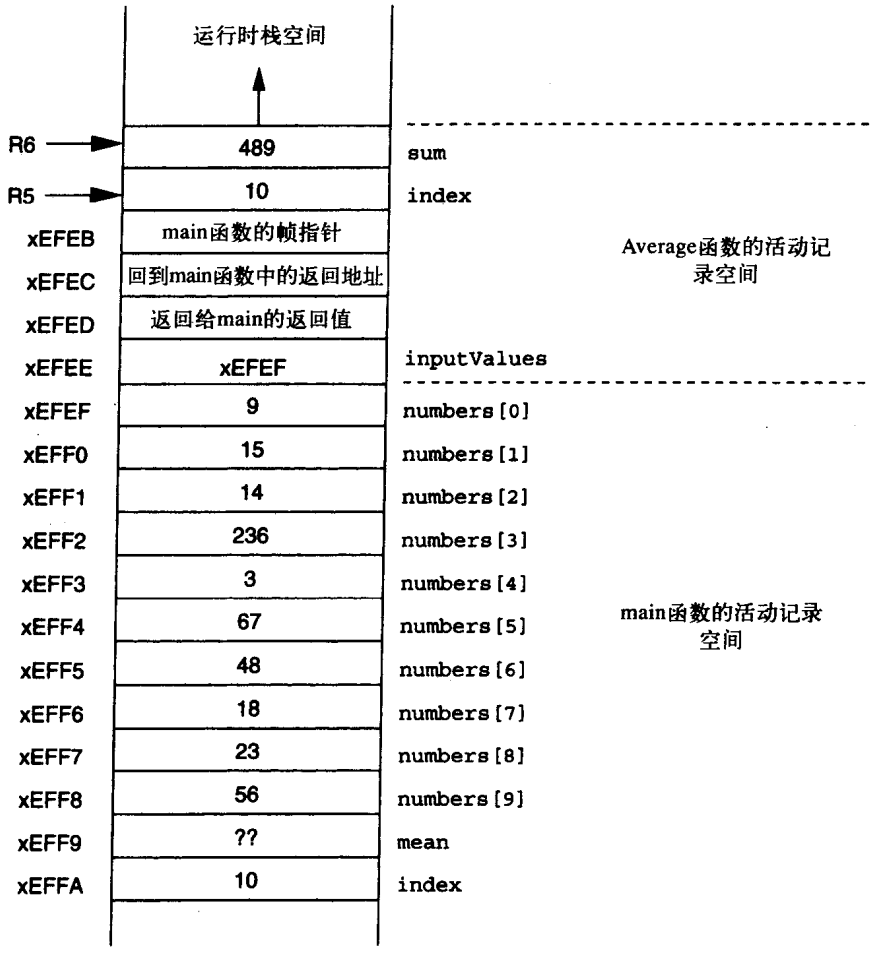


图16-11 函数Average返回前的栈空间内容

注意，其中输入参数inputValues在函数Average中的声明方法。方括号[]意味着，告诉编译器该参数是某数组的基地址。

由于在C语言中，数组采用“传址”方式。所以，在被调用函数内，对数组值所做的任何修改，在控制权返回调用者之后都是可见的。再思考一个问题，如果我们所要传递的参数是某个数据项（而不是整个数组），又该怎样选择是“传值”方式还是“传址”方式呢？

16.3.4 C语言的字符串

在C语言中，字符串是数组的一个重要应用。所谓“字符串”，是表示文本（text）的字符序列，本质上它就是字符数组。例如，

```
char word[10];
```

该语句声明的是一个可以储存10个字符的数组。字符串越长，所需要的数组长度越大。如果该字符串长度小于10，会怎样？在C等现代编程语言中，字符串结尾处是一个空字符（其ASCII码值为0），它代表字符串的结束，这样的字符串也称为无传输字符串。空字符在代码中表示为\0。下面继续之前的声明：

```
char word[10];

word[0] = 'H';
word[1] = 'e';
```

```
word[2] = 'l';
word[3] = 'l';
word[4] = 'o';
word[5] = '\0';

printf("%s", word);
```

在此，我们一个一个地给数组的每个元素赋值。结果是数组的内容包含字符串“Hello”。注意，字符串的结束符本身也是一个字符，它在数组中也占有一个位置。所以，尽管数组的声明长度为10，由于为空字符保留了一个位置，所以该数组能存放的字符串长度最大为9。

在这个例子中，printf使用了一个新的格式规范“%s”。该格式规范的作用是，打印字符串中的所有字符。从参数指定地址开始，直到结束字符‘\0’为止。

ANSI C编译器允许字符串在声明时被初始化。例如，前面例子可以表示如下：

```
char word[10] = "Hello";

printf("%s", word);
```

值得注意的是：（1）字符串和单个字符有所区别。在代码中，字符串由双引号“”包围，而单字符则由单引号包围（如‘A’）；（2）编译器会自动在字符串结尾添加结束符‘\0’。

字符串的例子

如图16-12所示的代码中，包含的是一个有关字符串的简单但却非常有用的操作：计算字符串的长度。我们知道，字符数组的大小并不代表它所包含字符串的长度（它只能代表字符串的最大长度）。所以，我们需要通过检查结束符，才能判断该字符串的实际长度。

```
 1  #include <stdio.h>
 2  #define MAX_STRING 20
 3
 4  int StringLength(char string[]);
 5
 6  int main()
 7  {
 8    char input[MAX_STRING];    /* Input string */
 9    int  length = 0;
10
11    printf("Input a word (less than 20 characters): ");
12    scanf("%s", input);
13
14    length = StringLength(input);
15    printf("The word contains %d characters\n", length);
16  }
17
18  int StringLength(char string[])
19  {
20    int index = 0;
21
22    while (string[index] != '\0')
23      index = index + 1;
24
25    return index;
26  }
```

图16-12 计算字符串长度的程序

判断字符串长度的算法很简单。从字符串的第一个元素开始，直到遇到结束字符为止，所经过的字符数目就是字符串的长度。图16-12所示的StringLength函数就是这么做的。

注意，其中scanf语句所使用的格式规范“%s”，它代表scanf将从键盘读入一连串字符，直到遇到第一个空（white space）字符。在C语言中，空字符包括：空格、制表符（tab）、换行符、回车、垂直制表符（vertical tab）或填表（form-feed）符等。所以，如果程序运行时，用户输入下面这段话（摘自《The New Colossus》作者为Emma Lazarus）：

```
Not like the brazen giant of Greek fame,
With conquering limbs astride from land to land;
```

结果是，只有单词Not被存入数组input，余下的内容则要等待后面的scanf调用来读取。所以，如果我们再一次“scanf(“%s”, &input);”，则input中存入“like”。注意，之间的空字符被“%s”控制符忽略。参考第18章的I/O行为描述。

注意，宏定义规定了字符数组的最大长度是20。那么，如果第一个有效单词的长度大于20，将发生什么情况？由于scanf函数并未指定有关input数组大小的信息，所以它将不断地把输入字符依次填入指定数组（或地址），直到遇到空字符为止。如果第一个单词的长度大于20，则在main函数内，input之后的局部变量空间将被覆盖。只要画出scanf在调用前后活动记录的快照，就可以知道其原因了。在本章习题中有一个题目，其要求就是让你修改该程序，捕捉用户输入字符数大于input数组长度的情况。

下面是一个稍微复杂的例子，用到了前面的StringLength函数。如图16-13所示代码，我们使用scanf从键盘读入字符串，然后调用一个函数“反转”该字符串，然后输出反转后的字符串（所谓“反转”，是指如果原字符串为“Hello”，则反转后结果为“olleH”。）

```
#include <stdio.h>
#define MAX_STRING 20

int StringLength(char string[]);
void CharSwap(char *firstVal, char *secondVal);
void Reverse(char string[]);

int main()
{
  char input[MAX_STRING];            /* Input string */

  printf("Input a word (less than 20 characters): ");
  scanf("%s", input);

  Reverse(input);
  printf("The word reversed is %s.\n", input);
}

int StringLength(char string[])
{
  int index = 0;

  while (string[index] != '\0')
    index = index + 1;

  return index;
}

void CharSwap(char *firstVal, char *secondVal)
{
  char tempVal;   /* Temporary location for swapping */

  tempVal = *firstVal;
  *firstVal = *secondVal;
  *secondVal = tempVal;
}

void Reverse(char string[])
{
  int index;
  int length;

  length = StringLength(string);

  for (index = 0; index < (length / 2); index++)
    CharSwap(&string[index], &string[length - (index + 1)]);
}
```

图16-13 反转字符串的程序

其中，反转函数Reverse要完成两个任务，一是借用StringLength函数获取字符串长度；二是交换字符，第1个与最后一个交换，第2个与倒数第二个交换，以此类推。

为完成交换，我们修改了如图16-4所示的NewSwap函数。在Reverse函数的反转循环中，我们循环调用CharSwap函数，以交换字符串中两个字符的位置。

在C标准函数库中，提供了许多优秀的字符串操作函数。例如，字符串拷贝、字符串合并、字符串比较、字符串长度计算等函数。这些函数的声明，都包含在头文件<string.h>中。可参考附录D.9.2中有关字符串的各种操作函数。

16.3.5 数组与指针的关系

或许你已经注意到，数组与指针存在一定的共性。例如，数组的名字与指向数组的指针变量是等价的，如下面代码所示：

```
char word[10];
char *cptr;

cptr = word;
```

该代码完全合法，并且非常有用。在此，我们赋值指针变量cptr，使之等于数组word的基地址。两者都是指向字符的指针，所以cptr和word之间可以互换。例如，word[3]和*(cptr + 3)，访问的都是字符串中的第4个字符。

两者之间的不同之处是，cptr是可被赋值的变量，而数组标识符word不可被赋值。例如，语句“word = newArray”就是不合法的。对编译器来说，数组标识符一定是指向数组在内存的地址，所以一旦分配该标识符，就“不应该”再被修改了。

表16-1所示是指针与数组符号之间的等价表达式。同一行中的表达式都是相互等价的。

表16-1 指针与数组之间的关系

cptr	word	&word[0]
(cptr + n)	word + n	&word[n]
*cptr	*word	word[0]
*(cptr+n)	*(word + n)	word[n]

16.3.6 实例：插入排序

在学习了数组之后，我们准备借用它来解决一个“难题”，这是一个非常有意思且非常有用的问题：将一个整型数组内的所有元素，按照升序排列。换句话说，就是重新整理数组（a[]），使得新数组满足：a[0]≤a[1]≤a[2]…。

为完成该任务，我们将采用一种被称做“插入排序”（Insertion Sort）的算法。所谓“排序”(sorting)，是计算科学中的一个经典问题，人们已花费很长时间去理解、分析和优化排序问题。所以，现在存在很多排序算法，在后续的计算机课程中，你将学习它们。我们在此采用“插入排序”算法，是因为它与生活中常用的排序方法接近，因而更容易理解。

我们以如下例子讲解插入排序。假设你在整理所收藏的音乐CD盒，并根据CD的艺术家名字的字母顺序，有序排放这些CD。假定采用“插入排序”法。首先，将CD分为两堆，一堆是已整理（排序）的，一堆是还未排序的。当然，一开始“已排序堆”必然是空的。然后，开始排序。每从未排序堆抽取一张CD，则将它按序插入已排序堆。例如，假设已排序堆中现有三张CD，艺术家的名字分别是：Coltrane、Mingus和Monk。现在有一张Davis的CD，则应该将它放在Coltrane和Mingus之间。随后，依次排放，直到未排序堆为空，我们称这种方法为“插入排序”。

那么，我们应该怎样将这种方法应用在数组排序中呢？采用系统分解法分析刚才的算法，我们发现，程序的核心应该是一个对数组的循环操作，将数组中的每个元素，按序插入新数组中。最后，新数组与原数组的元素应该是一样的，只是顺序不同而已。

所以，在该算法的代码实现中，我们需要两个数组，一个是原数组，另一个是已排序数组。其实，我们可以只使用一个数组，就能完成这个任务，这将使代码的内存占用量大大减小（只是对初学者来说，有些费解而已）。采用单个数组的方法是，数组起始部分为已排序元素，剩余部分为未排序元素。每从未排序部分（头部）取出一个元素，则将它插入排序部分的正确位置（数组插入意味着可能存在大量元素的移动）。如此反复，直到数组全部被遍历。

图16-14所示是真正的插入排序程序InsertionSort，它使用了一个嵌套循环。外层循环负责轮流扫描所有未排序项，内层循环扫描的是已排序项（为新元素查找插入位置）。

```
 1  #include <stdio.h>
 2  #define MAX_NUMS 10
 3
 4  void InsertionSort(int list[]);
 5
 6  int main()
 7  {
 8    int index;              /* Iteration variable          */
 9    int numbers[MAX_NUMS];  /* List of numbers to be sorted */
10
11    /* Get input */
12    printf("Enter %d numbers.\n", MAX_NUMS);
13    for (index = 0; index < MAX_NUMS; index++) {
14      printf("Input number %d : ", index);
15      scanf("%d", &numbers[index]);
16    }
17
18    InsertionSort(numbers);  /* Call sorting routine       */
19
20    /* Print sorted list */
21    printf("\nThe input set, in ascending order:\n");
22    for (index = 0; index < MAX_NUMS; index++)
23      printf("%d\n", numbers[index]);
24  }
25
26  void InsertionSort(int list[])
27  {
28    int unsorted;          /* Index for unsorted list items */
29    int sorted;            /* Index for sorted items        */
30    int unsortedItem;      /* Current item to be sorted     */
31
32    /* This loop iterates from 1 thru MAX_NUMS  */
33    for (unsorted = 1; unsorted < MAX_NUMS; unsorted++) {
34      unsortedItem = list[unsorted];
35
36      /* This loop iterates from unsorted thru 0, unless
37         we hit an element smaller than current item */
38      for (sorted = unsorted - 1;
39           (sorted >= 0) && (list[sorted] > unsortedItem);
40           sorted--)
41        list[sorted + 1] = list[sorted];
42
43      list[sorted + 1] = unsortedItem; /* Insert item      */
44    }
45  }
```

图16-14 插入排序

下面我们详解一下插入排序的过程。假设，变量unsorted = 4时（第33~43行），list数组包含的10个元素如下所示：

```
2 16 69 92 15 37 92 38 82 19
```

之后要做的，是将list[4]（即15）插入到list[0]到list[3]之间的已排序堆中。

内层循环的计数变量是sorted，以此遍历list中已排序元素（list[0]~list[3]）。注意该for循环的条件，如果list中某项小于当前要插入值（15），则位置就找到了。

内层循环体（第38~41行）每执行一次，就将已排序部分的一个元素移动到它的下一个位置。例如，list[3]被拷贝到list[4]。所以，内层循环体第一次执行之后，数组list的内容如下所示：

```
2 16 69 92 92 37 92 38 82 19
```

注意，15原来的位置（list[4]）被覆盖了。不用担心，我们已将15备份在unsortedItem变量中了。第二次循环，则对list[2]做同样的操作。所以，第二次循环之后，list的内容变为：

```
2 16 69 69 92 37 92 38 82 19
```

第三次之后，list为

```
2 16 16 69 92 37 92 38 82 19
```

至此，for循环中止，因为条件不再为“真”（即条件list[sorted] > unsortedItem不再满足）。因为当前已排序项list[0]=2，它比unsortedItem（15）小。内层循环中止后，执行语句“list[sorted + 1] = unsortedItem;”。于是，list已排序部分多了一个元素：

```
2 15 16 69 92 37 92 38 82 19
```

通过这种操作，我们可以将整个数组都做排序，即外层循环遍历整个list数组。

16.3.7 C语言数组的不足

与其他现代编程语言相比，C语言未提供数组的越界保护。所以，这成为C编程中出错的一个常见原因。换句话说，C语言对一个数组索引，未判断它是否落在数组范围内。编译器对于表达式a[i]的处理是，只生成计算索引表达式i的代码，而不去（其实是无法）判断它是否可能超越最后一个数组元素。如图16-15所示的代码，说明了这种数组越界问题导致的严重后果。如果为limit输入一个大于数组长度（MAX_SIZE）的数字，该程序将表现出各种可能的怪现象[⊖]。

```
1  #include <stdio.h>
2  #define MAX_SIZE 10
3
4  int main()
5  {
6    int index;
7    int array[MAX_SIZE];
8    int limit;
9
10   printf("Enter limit (integer): ");
11   scanf("%d", &limit);
12
13   for(index = 0; index < limit; index++) {
14     array[index] = 0;
15     printf("array[%d] is set to 0\n", index);
16   }
17 }
```

图16-15 如果用户输入一个太大的数，这个程序会发生一些怪现象

你可以通过程序运行时栈的绘制，分析出现问题的原因。

但正是C语言不对数组做边界检查（较弱的数组访问限制），才使得C代码的执行速度更快。这

⊖ 由所使用的编译器决定。为了可以观察到这个问题，你需要输入一个大于16的数，或者在array之后声明index。

是C语言的特性之一，与其他语言相比，它赋予了程序员更多的控制能力。所以，如果你在编码时不够细心，这个“bare-bones philosophy”（露骨哲学）（C语言的设计哲学是，将所有控制权交给程序员）将让你饱尝调试艰辛。为了避免这类问题，有经验的C程序员在使用数组时，通常会设计一些编程技巧。

C语言的数组存在的另一个缺陷是，数组大小是固定的（特别是静态声明的数组）。换句话说，编译该程序时，就必须确认数组的大小。C语言不支持用变量方式声明数组的大小。例如，如下C代码是不合法的。因为编译器要求，数组temp的大小（num_elements）在源代码分析时刻，必须是已确定的数值。

```
void SomeFunction(int num_elements)
{
  int temp[num_elements];  /* Generates a syntax error */

  :
}
```

对于有经验的C程序员，面对这个问题时的处理办法通常是：仔细分析代码的使用场合，然后分配“足够大”的空间。此外，还可以辅助添加自己设计的边界检查代码，以确认数组大小是否是“足够大”。另一个可选方案是，使用动态内存分配方法，即在运行时分配数组空间（有关内容将在第19章介绍）。

16.4 小结

本章讲述的重点是指针和数组这两个高级编程结构。这两个结构的特点是，都可以间接访问内存。本章的关键内容总结如下：

- 指针。所谓“指针”，是一个包含其他内存对象（如变量）地址的变量。通过指针，我们可以间接存取和操作这些内存对象。指针的一个简单应用是，它可以在函数调用时，通过“传址”方式传递参数。指针的应用还有很多，我们将在后续章节看到它们。
- 数组。所谓“数组”，是一组同类型元素在内存中的连续排列。我们通过“索引”访问数组中的特定元素。“索引”等于该元素距离数组起始位置的偏移量。很多生活中的对象，在计算机程序中很适合于表示为数组，这使得数组成为一个非常重要的数据结构。例如，字符串就是一个文本内容的数组。此外，我们还学习了几种数组操作方法（包括“插入排序”法）。

16.5 习题

16.1 试编写一个C函数，以一个字符串为参数。该字符串是一个长度不定的单词。函数的任务是将英语单词翻译成“Pig Latin”。翻译规则是：将单词字符串的首字母移到最后，然后添加字母“ay”。你可以假设这个数组的长度足够包含这两个额外字母。

例如，如果参数为单词“Hello”，则转换结果为“elloHay”。

16.2 试编写C程序，它将不停地接受用户输入的数字，直到最后两个数字完全相同为止，然后打印输出已接收数字的个数（不包括最后一个），以及这些数字的累加和。程序提示和输出界面如下所示：

```
Number: 5
Number: -6
Number: 0
Number: 45
Number: 45
4 numbers were entered and their sum is 44
```

16.3 试问，如下代码编译后的执行结果如何？

```
int x;

int main()
{
   int *px = &x;
   int x = 7;

   *px = 4;
   printf("x = %d\n", x);
}
```

16.4 试编写一个字符串操作函数，输入参数为两个字符串stringA和stringB。如果两个字符串相同，则返回0；按字典顺序，如果stringA排在stringB之前，则返回1；如果stringB排在stringA之前，则返回2。

16.5 基于习题16.4的函数，修改“插入排序”程序，使它能够对字符串（而不是整数）进行排序。

16.6 试将如下C函数翻译为LC-3汇编代码。

```
int main()
{
   int a[5], i;

   i = 4;
   while (i >= 0) {
      a[i] = i;
      i--;
   }
}
```

16.7 分析如下程序，回答问题。注意，ind是指向指针变量的指针变量（这种结构在C语言里是合法的）。

```
#include <stdio.h>

int main()
{
   int apple;
   int *ptr;
   int **ind;
   ind = &ptr;
   *ind = &apple;
   **ind = 123;

   ind++;
   *ptr++;
   apple++;

   printf("%x %x %d\n", ind, ptr, apple);
}
```

画出“apple++;”语句执行之后，运行时栈的快照。然后分析该程序的行为。

16.8 如下代码调用了函数triple。试问，triple函数的活动记录空间应不小于多大？

```
int main()
{
   int array[3];

   array[0] = 1;
   array[1] = 2;
   array[2] = 3;

   triple(array);
}
```

16.9 试编写程序，删掉数字序列中的重复数据项。例如，如果数字内容为“5，4，5，5，3”，则程序输出结果为“5，4，3”。

16.10 试编写程序，找出一个数字集合的中值（median）。所谓“中值”，是指集合中一半的数比它小，另一半的数比它大。提示：先对该集合排序。

16.11 阅读以下C程序，回答问题。

```
int FindLen(char *);

int main()
{
   char str[10];

   printf("Enter a string : ");
   scanf("%s", str);
   printf("%s has %d characters\n", str, FindLen(str));
}

int FindLen(char * s)
{
   int len=0;

   while (*s != '\0') {
      len++;
      s++;
   }

   return len;
}
```

a. main函数和FindLen函数的活动记录分别为多大？

b. 如果FindLen的输入参数是字符串“apple”，试写出FindLen返回前运行时栈的内容。

c. 如果程序运行时，用户输入的字符串长度大于10，那么这个活动记录内容如何？

16.12 如下代码的任务是：从键盘读入一个字符串，然后将其中的大写字符转换为小写字符，最后输出转换后的字符串。但是，该程序存在错误，请指出。

```
#include <stdio.h>
#define MAX_LEN  10
char *LowerCase(char *s);

int main()
{
   char str[MAX_LEN];

   printf("Enter a string : ");
   scanf("%s", str);

   printf("Lowercase: %s \n", LowerCase(str));
}

char *LowerCase(char *s) {
  char newStr[MAX_LEN];
  int index;

  for (index = 0; index < MAX_LEN; index++) {
     if ('A' <= s[index] && s[index] <= 'Z')
       newStr[index] = s[index] + ('a' - 'A');
     else
       newStr[index] = s[index];
  }

  return newStr;
}
```

16.13 阅读如下声明语句，回答问题。

```
#define STACK_SIZE 100

int  stack[STACK_SIZE];
int  topOfStack;

int  Push(int item);
```

a. 编写函数Push（已声明），将参数值item压入栈（stack数组）顶部。如果栈已满，则不能压入，同时函数返回1；如果item压入成功，则函数返回0。

b. 编写函数Pop，弹出栈顶元素。类似于Push，如果操作失败（即尝试从空栈弹出元素），则函数返回1。如果操作成功，则返回0。请考虑如何将弹出值返回给调用者。

第17章 递 归

17.1 概述

本章将介绍的内容是“*递归*”(recursive)，或许你已非常熟悉这个概念。例如，我们希望在一个已按字母排序的成绩单中，查找某个学生的成绩。通常的做法是，随机地从成绩单中间部分，找出一个学生与我们要找的名字进行匹配。如果不匹配，则比较两个名字的字母顺序关系，在成绩单当前位置之前或之后查找。之后的查找同样是采用这种随机匹配的方法。例如，我们要查找的学生是“Babe Ruth”，而随机看到的是“Mickey Mantle”。那么，下一步查找则是在其后半部分[⊖]中继续查找。如果Babe Ruth存在，则我们应该很快就能找到它。我们在这个有序集合中查找元素，所采用的方法就是*递归*，即查找的集合越来越小。

所谓“递归”，其思想很简单：递归函数通过调用自身，完成更小的子任务。如我们所见，递归是表达循环结构的一种特殊方式。它的优点在于，针对特定任务，它能更清晰地表述程序控制流的执行过程。同样，即使针对实际的编程问题，递归方法通常也可以替代传统的循环方法。在本章中，我们将介绍5个递归编程的例子，并了解在LC-3上实现递归函数的机制。美妙的“栈机制”，将使得递归的实现轻而易举——它使得递归函数的执行过程与普通函数完全一样。本章的目的是，帮助你深入理解递归的原理，分析和解释几个递归程序。因为，理解递归代码是编写递归代码的必要前提。最后，我们希望你能在实际问题的求解中，应用递归这个工具。

17.2 什么是递归

我们称一个调用自身的函数为“递归函数”。如图17-1所示，RunningSum就是一个递归函数。

```
1  int RunningSum(int n)
2  {
3    if (n == 1)
4       return 1;
5    else
6       return (n + RunningSum(n-1));
7  }
```

图17-1 递归函数的例子

该函数的任务是计算输入参数n~1之间所有整数的和。例如，RunningSum(4)所计算的等价于表达式“4 + 3 + 2 + 1”。只是，RunningSum采用的是递归方法。注意，1~4的求和与4与1~3的求和是等价的。同样，1~3的和就是3加上1~2的和。*递归*定义是递归算法的基础。换句话说，

$$\text{RunningSum}(n) = n + \text{RunningSum}(n\text{-}1)$$

在数学上，我们将递归函数描述为如上所示的“*递归等式*”(recurrence equation)。例如，如上所示的就是RunningSum的递归等式。为了完成该等式的计算，我们还需要一个初始值。即在上面这个方程式中，我们需声明：

⊖ 有关“Babe Ruth”和“Mickey Mantle”之间的顺序关系，两者之间比较的是“Ruth”和“Mantle”，所以Ruth在Mantle的后半部。本书其他地方也都沿用这种习惯，即在排序中以英文的姓为序。——译者注

RunningSum(1) = 1

递归等式的演算过程如下所示：

$$\begin{aligned}\text{RunningSum}(4) &= 4 + \text{RunningSum}(3)\\ &= 4 + 3 + \text{RunningSum}(2)\\ &= 4 + 3 + 2 + \text{RunningSum}(1)\\ &= 4 + 3 + 2 + 1\end{aligned}$$

在C语言中，RunningSum函数的运算过程与该递归等式完全相同。在RunningSum(4)的调用执行过程中，RunningSum又以参数3调用了自身（即RunningSum(3)）；而在RunningSum(3)结束之前，又调用了RunningSum(2)；在RunningSum(2)中则又调用RunningSum(1)；但在RunningSum(1)中，它不再递归调用，而是直接返回数值1给RunningSum(2)；然后，RunningSum(2)结束，返回2+1的值给RunningSum(3)；再之后，RunningSum(3)结束，返回3+2+1的值给RunningSum(4)。图17-2所示是RunningSum(4)执行过程的图示。

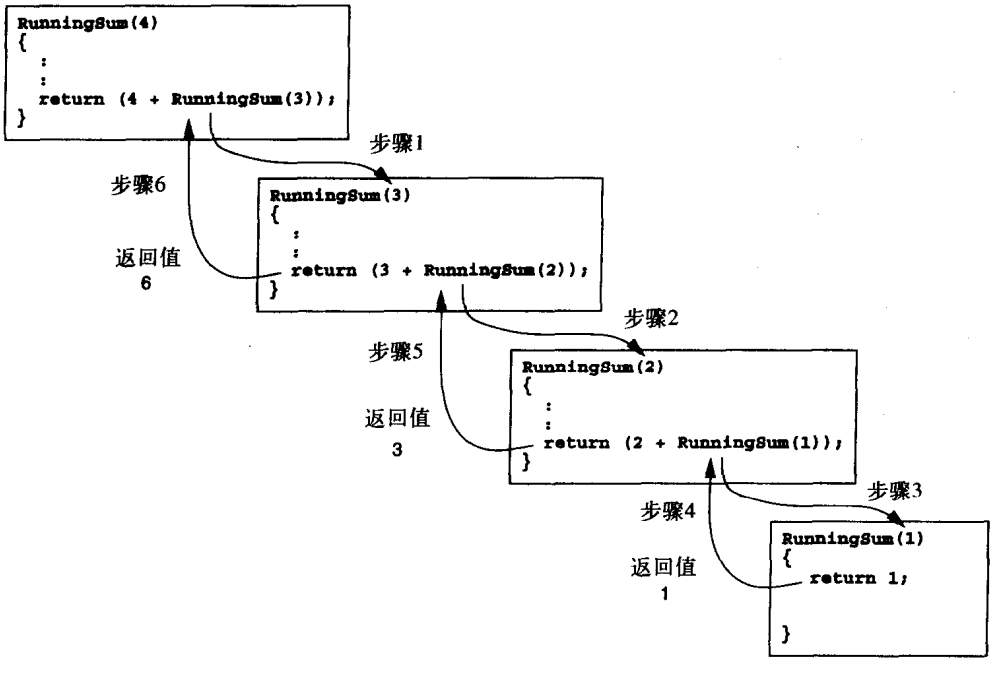


图17-2 RunningSum(4)被调用时的控制流程

17.3 递归与循环

当然，我们也可以使用循环结构（for语句）编写RunningSum函数，并且它的代码将比递归方法更容易理解。但在此，我们仍然采用了递归，目的就是通过简单的例子说明递归调用的原理。

在具体程序中，究竟是使用递归还是使用循环（如for、while）结构？答案是：都可以。所有的递归函数都可以编写为循环结构。但是，在有些问题中，递归将比循环更简捷。例如，那些很容易表示为递归等式的问题，就很适合用递归方法来实现。至于判断一个问题是采用递归方法还是循环方法更容易求解，则属于“计算机编程艺术”的问题了。在有了一些编程经验之后，你自然会清楚这个问题的答案。

在通过编程解决某些问题时，虽然递归的确带来不少方便，但同时也会付出代价。例如，我

们做个实验，编写一个循环版的RunningSum，然后以一个很大的数值n做参数，比较它和递归版程序的执行时间。你可以调用库函数（如gettimeofday）记录程序执行的起始和结束时间。经过几组n值的测试，你将发现递归版的程序运行相对要慢一些（要确认编译器未做递归优化）。我们将在17.5节看到，其中的原因是因为递归方式导致太多的函数调用，而循环方法没有这样的问题。

17.4 汉诺塔

汉诺（Hanoi）塔是个古老的难题，但我们可以用递归方法很容易地解决这个问题。汉诺塔的问题是：如图17-3所示，有三根柱子，其中的一根套着一摞圆盘，自顶向下，从小到大。我们的问题是，将这根柱子上的所有圆盘移至另一根柱子上。但是，移动规则有两条：（1）一次只能移动一个圆盘；（2）大圆盘永远都不能摞在小圆盘之上。在图17-3中，在柱子1上有5个圆盘，请按规则将这5个圆盘移至另一根柱子上。

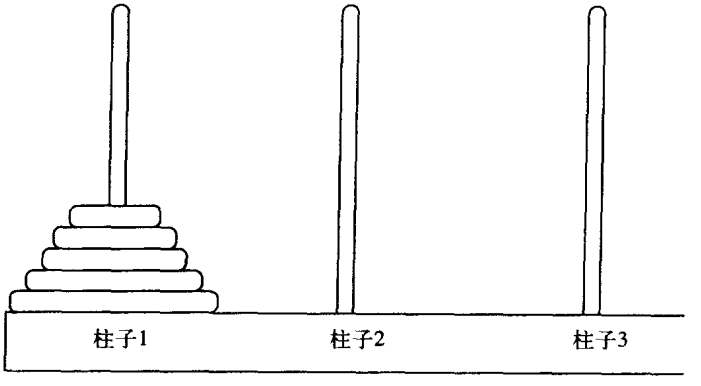


图17-3　汉诺塔谜题

有关这个谜题有个传说：世界诞生时，Brahma寺庙的僧人受命将64个圆盘从一根柱子移至另一根柱子上。一旦他们完成了这个任务，世界末日也就到了。

我们该怎样通过程序求解这个难题呢？我们从结果开始思考，发现：当移动最后一个圆盘时，必须先将该圆盘从柱子1移至目标柱子，比如说柱子3，然后再将其他圆盘（必然是在柱子2中）移至它的上面。所以，之前一定是已将前$n-1$个圆盘从柱子1全部搬移到了中间那根柱子。之后，再将$n-1$个圆盘从中间柱子移至目标柱子上。这样，就完成了该谜题。当然，一次移动$n-1$个圆盘是不允许的，所以问题还没完全解决。但是，我们现在已可以将问题分解为两个更小的子问题。换句话说，如果这两个子问题解决了，那么问题就真正解决了。其中，一是将最大的圆盘移至目标柱子3（这很容易做到，因为柱子3现在是空的），之后我们再也不用关心它了；二是$n-1$个圆盘的移动（从柱子2至柱子3）。现在，第$n-1$个圆盘成为最大的圆盘，所以问题又变成“怎样将最大的圆盘移至目标柱子上”，这就是原来分解之前的问题（只是参数由n变为$n-1$）。因此，我们可以如法炮制地解决（即继续分解问题）。

现在，我们将这个问题定义为如下的递归过程：将该任务表示为Move(n, target)，即将n个圆盘移至目标柱子上。首先要做的是Move($n-1$, intermediate)，即将前$n-1$个圆盘移至中间（intermediate）柱子上，然后将盘n移至target柱子，最后是Move($n-1$,target)，即将盘$n-1$从intermediate移至目标柱子。所以，在Move(n,target)中，将两次递归调用Move操作（都是移动$n-1$个圆盘）。

此外，如同数学中的递归方程式，递归定义中也需要定义递归的结束条件（base case）。在这个问题中，结束条件是最小的圆盘（盘1）被移动至目标柱子。因为，盘1总是在最上面，所以它的移动不涉及任何其他盘的移动，可直接移动。没有结束条件，则递归函数将无穷递归，类似循

环操作的无限循环。

有了前面的分析和定义，将递归定义转化为C代码就相对容易了。图17-4所示是该算法对应的C递归函数。

```
/*
** Inputs
**   diskNumber is the disk to be moved (disk1 is smallest)
**   startPost is the post the disk is currently on
**   endPost is the post we want the disk to end on
**   midPost is the intermediate post
*/
MoveDisk(diskNumber, startPost, endPost, midPost)
{
   if (diskNumber > 1) {
       /* Move n-1 disks off the current disk on          */
       /* startPost and put them on the midPost           */
       MoveDisk(diskNumber-1, startPost, midPost, endPost);

       /* Move the largest disk.                          */
       printf("Move disk %d from post %d to post %d.\n",
               diskNumber, startPost, endPost);

       /* Move all n-1 disks from midPost onto endPost    */
       MoveDisk(diskNumber-1, midPost, endPost, startPost);
   }
   else
       printf("Move disk 1 from post %d to post %d.\n",
               startPost, endPost);
}
```

图17-4 解决汉诺塔谜题的递归函数

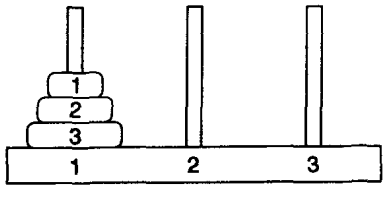


图17-5 初始的汉诺塔谜题

让我们看一下，在只有三个圆盘的情况下，程序的执行过程。如下所示是第一次MoveDisk函数调用，即将要把盘3（最大的圆盘）从柱子1移至柱子3，并以柱子2为中介。换句话说，这是一个如图17-5所示的$n = 3$的Hanoi塔问题。

```
/* diskNumber 3; startPost 1; endPost 3; midPost 2 */
MoveDisk(3, 1, 3, 2)
```

此次调用中，又一次调用MoveDisk，试图将盘1和2从盘3上移开，搬至柱子2（以柱子3为中介）。如下所示（源代码第15行）。

```
/* diskNumber 2; startPost 1; endPost 2; midPost 3 */
MoveDisk(2, 1, 2, 3)
```

而将盘2从柱子1移至柱子2，前提是先将盘1从盘2上移至柱子3（中介）。所以，再次调用MoveDisk（第15行）。

```
/* diskNumber 1; startPost 1; endPost 3; midPost 2 */
MoveDisk(1, 1, 3, 2)
```

之后，由于盘1可直接移动，即执行第二个printf语句[⊖]（第25行），打印信息如下：

⊖ 有关“移动”操作的表示，由于程序还无法显示图形，所以每做一次移动操作，我们就打印一个信息，替代所做的“移动”操作。你可以按照提示，在手边的模型上“按部就班”，即可成功。——译者注

```
Move disk number 1 from post 1 to post 3.
```

图17-6所示是第一个移动操作后的情况。

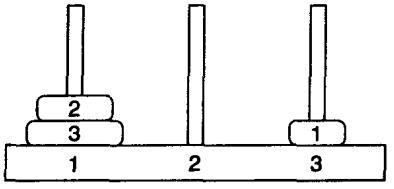


图17-6　第一次移动后的汉诺塔谜题

到目前为止，MoveDisk可以返回调用者了，即MoveDisk(2, 1, 2, 3)。回顾一下，在这个调用者中，我们所期待的是将盘2之上的所有圆盘都移至柱子3。现在，这个任务已完成。所以，下一个操作就是将盘2从柱子1移至柱子2。之后的printf语句（第18行）即执行该任务，它告诉我们，又移动一个圆盘了。

```
Move disk number 2 from post 1 to post 2.
```

再次移动后的情况如图17-7所示。

printf之后是一个MoveDisk调用。此次调用的任务是，将原先在盘2之上的圆盘移回盘2之上（第22行）。

```
/* diskNumber 1; startPost 2; endPost 3; midPost 1 */
MoveDisk(1, 2, 3, 1)
```

盘1之上没有其他圆盘，所以直接移动。我们看到输出信息。

```
Move disk number 1 from post 3 to post 2.
```

移动后的情况见图17-8所示。

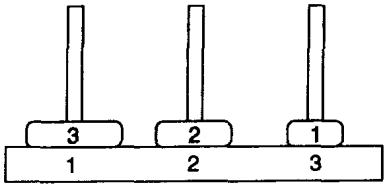


图17-7　第二次移动后的汉诺塔谜题

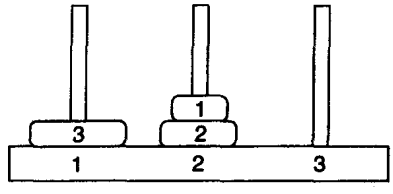


图17-8　第三次移动后的汉诺塔谜题

现在，控制权返回MoveDisk(2, 1, 2, 3)，即已将盘2及之上所有圆盘从柱子1移至柱子2。随后，控制权继续返回至MoveDisk(3, 1, 3, 2)。现在，盘3之上的圆盘都已经移至柱子2，下面要做的是，将盘3从柱子1移至柱子3，即执行printf语句（第15行）。

```
Move disk number 3 from post 1 to post 3.
```

第4次移动后的结果如图17-9所示。

接下来的任务是将盘2及之上的所有圆盘从柱子2移至柱子3（以柱子1为中介）。于是，执行代码第22行的调用。

```
/* diskNumber 2; startPost 2; endPost 3; midPost 1    */
MoveDisk(2, 2, 3, 1)
```

首先是将盘1从柱子2移至柱子1（代码的第15行）。

```
/* diskNumber 1; startPost 2; endPost 1; midPost 3    */
MoveDisk(1, 2, 1, 3)
```

该移动可直接操作。第5次的移动结果如图17-10所示。

```
.Move disk number 1 form post 2 to post 1.
```

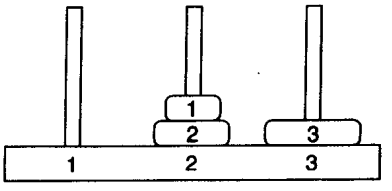


图17-9 第四次移动后的汉诺塔谜题

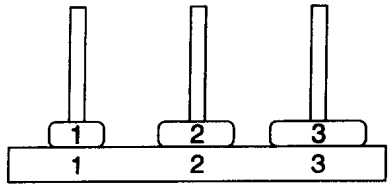


图17-10 第五次移动后的汉诺塔谜题

之后，返回到调用者MoveDisk(2, 2, 3, 1)。下面，可以将盘2移至柱子3。第6次的移动结果如图17-11所示。

```
.Move disk number 2 from post 2 to post 3.
```

最后一个移动操作是，将盘2之上的所有圆盘移回盘2之上。调用代码如下：

```
/* diskNumber 1; startPost 1; endPost 3; midPost 2    */
MoveDisk(1, 1, 3, 2)
```

完成最后一个移动之后的结果如图17-12所示。

```
.Move disk number 1 from post 1 to post 3.
```

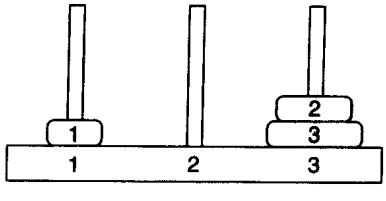


图17-11 第六次移动后的汉诺塔谜题

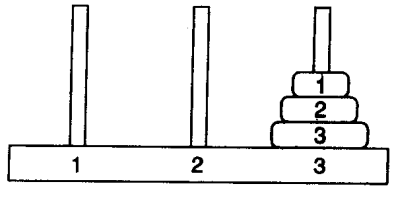


图17-12 完成后的汉诺塔谜题

至此，谜题解出。

让我们回顾一下，在三个圆盘的Hanoi塔问题的求解过程中，MoveDisk函数的递归调用。

```
MoveDisk(3, 1, 3, 2)  /* Initial Call */
MoveDisk(2, 1, 2, 3)
MoveDisk(1, 1, 3, 2)
MoveDisk(1, 2, 3, 1)
MoveDisk(2, 2, 3, 1)
MoveDisk(1, 2, 1, 3)
MoveDisk(1, 1, 3, 2)
```

思考一下，如果采用循环结构求解该问题，程序应该怎么写？思考之后，相信你一定会赞叹递归方法之简洁。回顾Hanoi塔的传说：当寺内僧人完成64个圆盘的谜解后，世界末日将降临。试问，如果每移动一个圆盘需要一秒钟，他们需要多久才能完成？

17.5 斐波纳契数列

下面的这个递归等式是大家非常熟悉的“斐波纳契数列”（Fibonacci Numbers）。该数列具备很多有趣的数学、几何和自然特性。

$f(n) = f(n-1) + f(n-2)$

$f(1) = 1$

$f(0) = 1$

换句话说，在斐波纳契数列中，第n个数是其之前两个数的和，即数列为1、1、2、3、5、8、13…。该数列最早是由意大利数学家Leonardo of Pisa在1200年前后提出的，而他父亲的名字为

Bonacci，所以他称自己为Fibonacci（即filius Bonacci，英文意思是son of Bonacci，Bonacci的儿子）。Fibonacci创建这个数列的目的是，计算兔子的饲养数量。后来，我们发现在自然界中也充满该数列，如贝壳上的螺旋线、花的花瓣模式等。

我们可以为这个递归等式写一个递归函数，计算第*n*个斐波纳契数。其中，Fibonacci(n)的结果可以通过计算“Fibonacci(n-1) + Fibonacci(n-2)”的递归方法获得。该递归的结束条件是：Fibonacci(1)和Fibonacci(0)都直接等于1。图17-13所示是计算第*n*个斐波纳契数的递归代码。

```
#include <stdio.h>

int Fibonacci(int n);

int main()
{
  int in;
  int number;

  printf("Which Fibonacci number? ");
  scanf("%d", &in);

  number = Fibonacci(in);
  printf("That Fibonacci number is %d\n", number);
}

int Fibonacci(int n)
{
  int sum;

  if (n == 0 || n == 1)
     return 1;
  else {
     sum = (Fibonacci(n-1) + Fibonacci(n-2));
     return sum;
  }
}
```

图17-13　Fibonacci是一个递归函数，它用来计算第n个斐波纳契数

我们将通过这个例子，讲述递归过程在计算机系统底层的实现机制。具体地说，就是通过栈机制提供对递归调用的支持。当一个函数被调用时，不管调用的是它自己还是其他函数，总是要在栈空间中为其压入一个新的活动记录。换句话说，函数每次被调用时，都会有一个私有的活动记录拷贝（包括参数和局部变量），相同函数的不同调用，其拷贝是不一样的（相互之间不存在任何关联）。这恰巧也是递归所需要的基本要求。例如，如果在函数中，某个变量在内存中所占的位置是静态分配的，则每次调用Fibonacci函数，都将覆盖该变量在前一次调用中所具有的数值。

下面，我们观察一下以参数3调用Fibonacci函数（即Fibonacci(3)）的情况。最开始，Fibonacci(3)的活动记录被压入栈。图17-14所示是此次调用后的栈状态。

进入Fibonacci(3)函数体之后，由于表达式“Fibonacci(n-1) + Fibonacci(n-2)”的计算顺序是自左向右，所以先计算Fibonacci(3-1)的结果，即Fibonacci(2)。随后，Fibonacci(2)的活动记录又被压入栈（如图17-14的步骤2所示）。

对于Fibonacci(2)来说，参数n = 2，不满足结束条件，所以再次产生Fibonacci(1)调用（如图17-14的步骤3）。该调用是在Fibonacci(2-1) + Fibonacci(2-2)中产生的。

而Fibonacci(1)则不再需要递归调用（因为参数n = 1已满足结束条件）。它将直接返回结果1至Fibonacci(2)。随后是Fibonacci(0)的调用，最后完成“Fibonacci(1) + Fibonacci(0)”的计算（如图17-14的步骤4）。其中，Fibonacci(0)也直接返回1。

至此，Fibonacci(2)调用完成。它再将结果（为2）返回给它的调用者Fibonacci(3)。表达式Fibonacci(2) + Fibonacci(1)的左半部完成之后，Fibonacci(3)继续调用Fibonacci(1)（如图17-14的步

骤5)。其中，Fibonacci(1)直接返回1。于是Fibonacci(3)调用完成，结果为3（如图17-14的步骤6)。

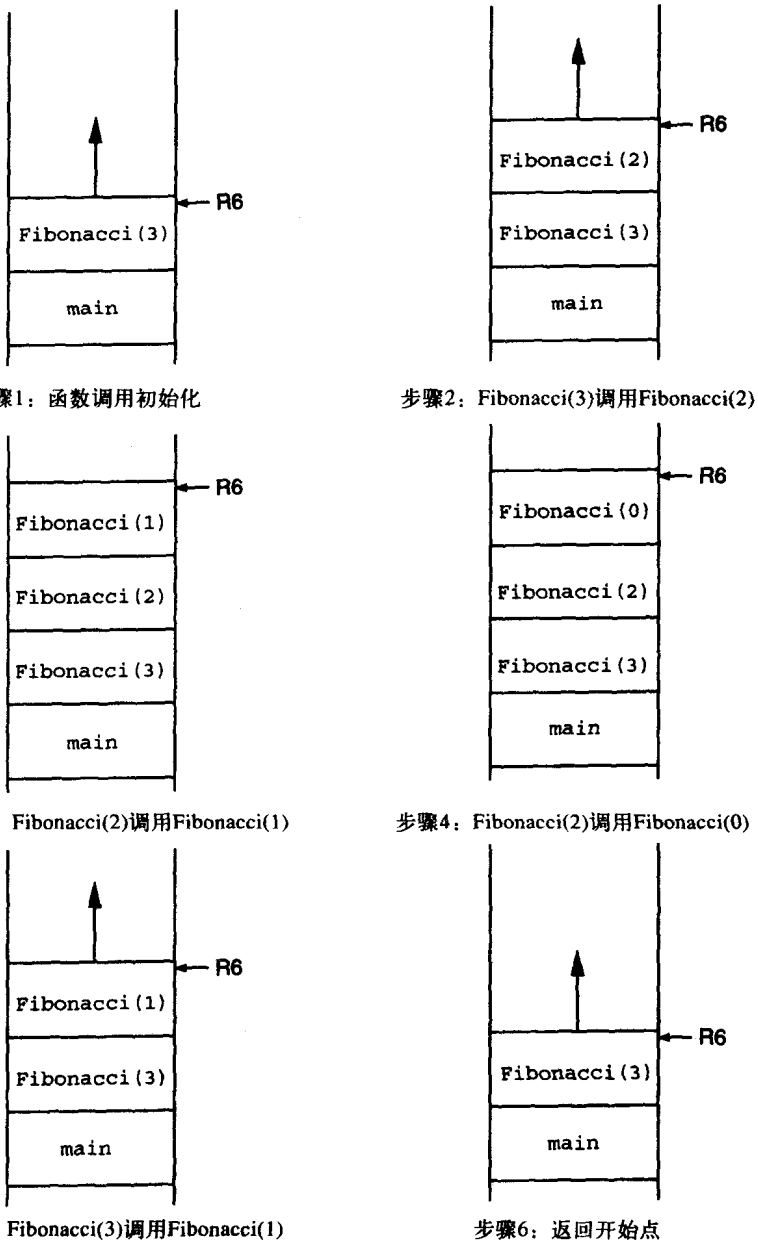


图17-14 Fibonacci(3)调用过程中的运行时栈快照

如果我们将Fibonacci(3)的递归过程表示为代数表达式，则如下所示：

```
Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
             = (Fibonacci(1) + Fibonacci(0)) + Fibonacci(1)
             = 1 + 1 + 1 = 3
```

在Fibonacci(3)的计算过程中，函数调用顺序如下所示：

```
Fibonacci(3)
Fibonacci(2)
Fibonacci(1)
Fibonacci(0)
Fibonacci(1)
```

如果将Fibonacci(4)的计算过程写出来，你将发现Fibonacci(3)的调用顺序包含在Fibonacci(4)的调用顺序之中。原因很简单，因为Fibonacci(4) = Fibonacci(3) + Fibonacci(2)。同样，Fibonacci(4)的调用顺序也是Fibonacci(5)调用顺序的一部分。在本章最后，有一个习题，计算Fibonacci(n)的计算过程中函数调用发生的次数。

图17-15所示是该程序在LC-3 C编译器下生成的代码。注意，在这段代码的生成过程中，编译器未对递归做优化处理。由于都是基于栈机制，所以对于活动函数是递归函数还是普通函数，编译器都是同等对待的。仔细阅读代码，你将发现，编译器为能正确翻译Fibonacci函数的第24行，使用了一个额外的临时变量。很多编译器在翻译较复杂的表达式时，都会使用类似的临时变量。临时变量的空间分配在活动记录中程序声明的局部变量之上。

```
 1  Fibonacci:
 2    ADD  R6, R6, #-2 ; push return value/address
 3    STR  R7, R6, #0  ; store return address
 4    ADD  R6, R6, #-1 ; push caller's frame pointer
 5    STR  R5, R6, #0  ;
 6    ADD  R5, R6, #-1 ; set new frame pointer
 7    ADD  R6, R6, #-2 ; allocate space for locals and temps
 8
 9    LDR  R0, R5, #4  ; load the parameter n
10    BRZ  FIB_BASE    ; n==0
11    ADD  R0, R0, #-1 ;
12    BRZ  FIB_BASE    ; n==1
13
14    LDR  R0, R5, #4  ; load the parameter n
15    ADD  R0, R0, #-1 ; calculate n-1
16    ADD  R6, R6, #-1 ; push n-1
17    STR  R0, R6, #0  ;
18    JSR  Fibonacci   ; call to Fibonacci(n-1)
19
20    LDR  R0, R6, #0  ; read the return value at top of stack
21    ADD  R6, R6, #-1 ; pop return value
22    STR  R0, R5, #-1 ; store it into temporary value
23    LDR  R0, R5, #4  ; load the parameter n
24    ADD  R0, R0, #-2 ; calculate n-2
25    ADD  R6, R6, #-1 ; push n-2
26    STR  R0, R6, #0  ;
27    JSR  Fibonacci   ; call to Fibonacci(n-2)
28
29    LDR  R0, R6, #0  ; read the return value at top of stack
30    ADD  R6, R6, #-1 ; pop return value
31    LDR  R1, R5, #-1 ; read temporary value: Fibonacci(n-1)
32    ADD  R0, R0, R1  ; Fibonacci(n-1) + Fibonacci(n-2)
33    BR   FIB_END     ; branch to end of code
34
35  FIB_BASE:
36    AND  R0, R0, #0  ; clear R0
37    ADD  R0, R0, #1  ; R0 = 1
38
39  FIB_END:
40    STR  R0, R5, #3  ; write the return value
41    ADD  R6, R5, #1  ; pop local variables
42    LDR  R5, R6, #0  ; restore caller's frame pointer
43    ADD  R6, R6, #1  ;
44    LDR  R7, R6, #0  ; pop return address
45    ADD  R6, R6, #1  ;
46    RET
```

图17-15　Fibonacci函数对应的LC-3汇编代码

17.6　二分查找

在本章的概述部分，我们曾描述了这样一种操作：在一个已按字母表排序的成绩集合中，通

过递归方法查找某个同学的成绩。我们称这种方法为“二分查找”(binary search)。这种方法非常适合在有序集合中查找特定元素。下面，我们将在递归和数组的基础上，实现一个二分查找的C递归函数。

假设我们将在一个升序排列的整型数组中，查找某个特定整数。如果查找到了，则函数返回该整数在数组中的索引号；如果未找到，则返回-1。在该任务中，我们可以采用“二分查找”方法：给定一个数组和被查找的整数，我们先匹配数组中间的数，判断被查找整数：(1) 等于该中间数；(2) 小于中间数；(3) 大于中间数。如果相等，则匹配成功；如果小于中间数，则进入数组的前半部，并采用相同方法继续查找；如果大于中间数，则进入数组后半部，并采用同样方法查找。注意，在 (2) 和 (3) 的情况中，我们可以采用递归调用。但是，如果数组中根本就没有我们要找的这个数，情况会如何呢？在递归方式中，每次递归调用总是在比原数组更小的子数组中查找。所以，如果被查找的数不存在，则最终将是对一个空数组（即数组大小为0）进行查找。遇到这种情况时，函数直接返回-1。实际上，这就是该递归的结束条件。

图17-16所示是二分查找算法的C递归代码。注意，为保证在每级调用中都能判断出数组的大小，BinarySearch调用的传入参数中，包含了当前子数组的起始和结束位置。在每次调用时，变量start和end的值都会被重新定义，即在原数组list中越来越小的子数组中查找。

图17-17显示的是这段代码的执行流程。其中，数组list有11个元素。在BinarySearch的最初调用中，传递的参数包括：要查找的元素（item）和被查找数组（回顾第16章，传递的是数组中第一个元素的地址或数组基地址）。此外，还包括数组的查找范围，即起始位置和结束位置。而每一次BinarySearch的递归调用中，该范围将越来越小，直到被查找子集中只有一个或者没有元素为止。这两种情况是递归的结束条件。

反之，如果不借助二分查找技术，我们可以采用直观的顺序查找方法。即依次匹配list[0]、list[1]、list[2]……等，直到匹配成功或数组结尾。但是，在二分查找方法中，匹配次数更少，尤其是在数组足够大的时候，它的执行速度更快。在后续的计算机课程中，你还将学习二分查找算法的分析结果，它的运行时间复杂度是$\log_2 n$（n是数组大小），而顺序查找的复杂度则是n。

```
 1  /*
 2  ** This function returns the position of 'item' if it exists
 3  ** between list[start] and list[end], or -1 if it does not.
 4  */
 5  int BinarySearch(int item, int list[], int start, int end)
 6  {
 7    int middle = (end + start) / 2;
 8
 9    /* Did we not find what we are looking for? */
10    if (end < start)
11      return -1;
12
13    /* Did we find the item? */
14    else if (list[middle] == item)
15      return middle;
16
17    /* Should we search the first half of the array? */
18    else if (item < list[middle])
19      return BinarySearch(item, list, start, middle - 1);
20
21    /* Or should we search the second half of the array? */
22    else
23      return BinarySearch(item, list, middle + 1, end);
24  }
```

图17-16 执行二分查找的递归C函数

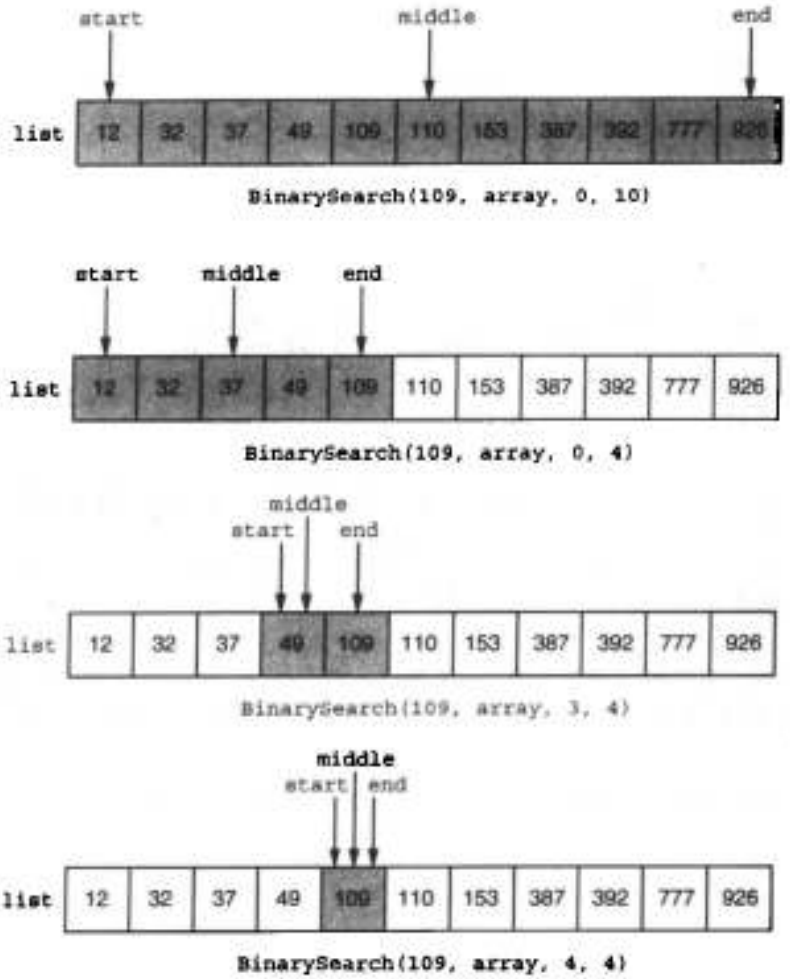


图17-17 在一个含有11个元素的数组里使用二分查找：我们要找的是元素109

17.7 整数转换为ASCII字符串

关于递归函数的最后一个例子，是将一个整数值转换为ASCII字符串。回顾第10章的内容，为在屏幕上显示一个整数值，该整数的每一位都被单独取出，然后转换为对应的ASCII字符，最后输出显示。在第10章中，我们使用的是直接循环方法编写LC-3程序。

下面，我们将采用递归方法来完成：如果被显示整数只有一位，则将它直接转换为ASCII字符并输出（结束条件）；如果被显示的整数有多位，我们对除最低位（最右边）之外的数进行递归调用，递归调用返回时再显示最右边的数。

```
1  #include <stdio.h>
2
3  void IntToAscii(int i);
4
5  int main()
6  {
7    int in;
8
9    printf("Input number: ");
10   scanf("%d", &in);
```

图17-18 递归函数IntToAscii，将一个正整数转化为ASCII字符串

```
11
12    IntToAscii(in);
13    printf("\n");
14  }
15
16  void IntToAscii(int num)
17  {
18    int prefix;
19    int currDigit;
20
21    if (num < 10)                /* The terminal case      */
22      printf("%c", num + '0');
23    else {
24      prefix = num / 10;         /* Convert the number     */
25      IntToAscii(prefix);        /* without last digit     */
26
27      currDigit = num % 10;      /* Then print last digit */
28      printf("%c", currDigit + '0');
29    }
30  }
```

图17-18 递归函数IntToAscii，将一个正整数转化为ASCII字符串（续）

递归函数IntToAscii的工作流程如下：假设要显示的整数是21 669（函数调用IntToAscii(21669)），在该函数中，问题被分为两部分。一是调用递归函数IntToAscii显示“2166”；二是在调用返回后，再显示“9”。

在代码中，我们采用除以10的方法去除参数num的最低位，然后将结果（也是更小的）整数传递给下一级递归调用。如果num只有一位，则将它转换为ASCII字符并显示，即不再继续递归调用。

当前级别的函数返回调用者之后，则将刚才的余数（最低位的数）转换为ASCII字符并显示。为验证该过程，我们给出IntToAscii(12345)调用过程的调用序列，如下所示：

```
IntToAscii(12345)
IntToAscii(1234)
IntToAscii(123)
IntToAscii(12)
IntToAscii(1)
printf('1')
printf('2')
printf('3')
printf('4')
printf('5')
```

17.8 小结

在本章中，我们介绍了递归的概念。所谓“递归”，就是通过递归地调用自身程序，完成问题的子任务。在递归中，我们定义函数f(n)。而表达式f($n-1$)则代表将同样的函数作用于一个比n更小的参数。例如，斐波纳契数列的递归表示如下所示：

```
Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2);
```

为保证递归过程最终能够终止，还必须有一个结束条件。

在编程中，递归，是一个非常有用的工具。在合适的情况下，它可以使问题变得非常简单。例如，通过递归方法，就可以非常简单地解决Hanoi塔问题。相比之下，采用循环方法建立方程则非常困难。在后续的课程里，我们还将学习很多使用了“指针”的数据结构（如树、图），其中，操作它们最简单的方法就是采用递归函数。对于计算机底层来说，递归函数的处理与普通函数没有区别。因为，在栈机制方式下，我们将为任何一个函数的任何一次调用，都独立地分配一个活动记录空间。因而，即使是同一个函数，不同调用的活动记录之间不存在任何冲突。

17.9 习题

17.1 试根据本章给出的例子，回答下列问题。

a. 如果执行RunningSum(10)，试问RunningSum（17.2节）将被调用多少次？

b. 如果调用RunningSum(n)呢？以n为参数表示结果。

c. 在Hanoi塔问题中，如果开始的函数调用是MoveDisk(4, 1, 3, 2)，试问MoveDisk将被调用多少次？

d. 在n个圆盘的Hanoi塔问题中，共有多少次函数调用呢？

e. 如果调用Fibonacci(10)，试问Fibonacci函数（见图17-13）将被调用多少次？

f. 如果计算第n个斐波纳契数，试问有多少次函数调用？

17.2　试问，在递归函数的每次调用中，返回地址都是一样的吗？为什么？

17.3　在如图17-18所示的IntToAscii函数中，如果我们将printf和递归调用之间的位置调换一下，结果会如何？

17.4　阅读如下函数，函数调用count(20)的结果是多少？

```
int count(int arg)
{
   if (arg < 1)
      return 0;

   else if (arg % 2)
      return(1 + count(arg - 2));
   else
      return(1 + count(arg - 1));
}
```

17.5　阅读如下所示C程序，回答问题：

```
#include <stdio.h>

int Power(int a, int b);

int main(void)
{
   int x, y, z;

   printf("Input two numbers: ");
   scanf("%d %d", &x, &y);

   if (x > 0 && y > 0)
     z = Power(x,y);
   else
     z = 0;

   printf("The result is %d.\n", z);
}

int Power(int a, int b)
{
   if (a < b)
      return 0;
   else
      return 1 + Power(a/b, b);
}
```

a. 给定以下输入，给出完整输出。

(1) 4 9

(2) 27 5

(3) −1 3

b. Power函数的功能是什么？

c. 图17-19所示是Power函数调用后的栈快照。其中，显示的是两段活动记录，且部分内容已填写。假设该快照反映的是Power函数中某个return语句执行之前的情况，试填写图中标记为问号（?）的位置内容。如果位置内容是地址值，请用箭头标出该地址所指向的位置。

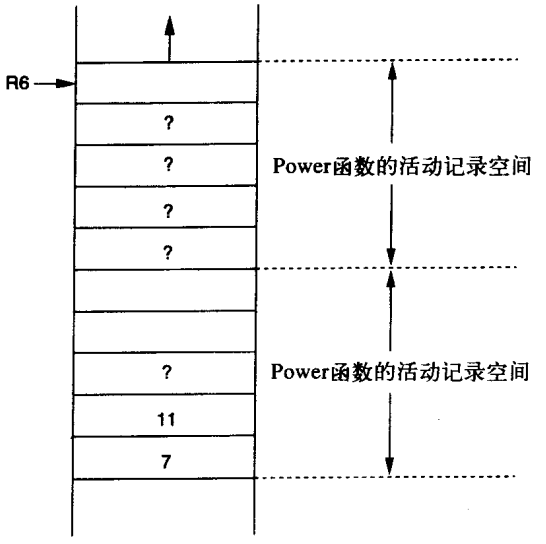


图17-19 Power函数调用后的运行时栈

17.6 阅读如下C函数，回答问题：

```
int Sigma( int k )
{
   int l;

   l = k -1;

   if (k==0)
      return 0;
   else
      return (k + Sigma(l));
}
```

a. 将该递归函数转换为一个非递归函数。假设其中Sigma()的调用参数总是非负的值。

b. 假设有一个1KB大小的连续栈空间，且地址和整型数的宽度都是16-bit。试问，该有限大小的空间能容纳多少次递归调用？临时变量的存储空间不计算在内。

17.7 假设如下C程序编译后在LC-3上运行。程序执行时，栈的起始地址是xFEFF，增长方向是xC000（即栈的最大占用内存大小是16KB）。试回答问题：

```
SevenUp(int x)
{
   if (x == 1)
      return 7;
   else
      return (7 + sevenUp(x - 1));
}

int main()
{
   int a;

   printf("Input a number \n");
   scanf("%d", &a);

   a = SevenUp(a);

   printf("%d is 7 times the number\n", a);
}
```

a. 为保证程序顺利运行，最大的输入值是多少？解释理由。

b. 如果最大栈空间是4KB，则最大可能的输入值又是多少？解释理由。

17.8 试编写循环版本的斐波纳契函数，它可以找出第n个斐波纳契数。然后，在不同的n取值下，比较循环版本和递归版本的运行时间。试问，为何当n足够大时，递归版本的运行时间要慢很多？

17.9 如图17-16所示的二分查找程序，只能查找按升序排序的数组。试重写这段代码，使之能够查找按降序排序的数组。

17.10 如下递归方式实现的算法，其代码表示比循环方式的代码要容易得多。试给出问题a~c的返回值，并回答最后一个问题。

```
int ea(int x, int y)
{
   int a;

   if (y == 0)
      return x;
   else {
      a = x % y;
      return (ea(y, a));
   }
}
```

a. ea(12, 15)

b. ea(6, 10)

c. ea(110, 24)

d. 该函数的作用是什么？试编写该函数的循环版代码。

17.11 阅读如下C代码，试编写一个非递归方式的代码。

```
int main()
{
   printf("%d", M());
}

void M()
{
   int num, x;
   printf("Type a number: ");
   scanf("%d", &num);
   if (num <= 0)
      return 0;
   else {
      x = M();
      if (num > x)
         return num;
      else
         return x;
   }
}
```

17.12 阅读如下递归函数，回答问题：

```
int func (int arg)
{
   if (arg % 2 != 0)
      return func(arg - 1);
   if (arg <= 0)
      return 1;

   return func(arg/2) + 1;
}
```

a. 是否存在一个arg的值，能导致无限递归？如果有，请给出这个值。

b. 如下所示，在main函数中调用func函数。试问该程序执行过程中，func函数共被调用了多少次？

```
int main()
{
   printf("The value is %d\n", func(10));
}
```

17.13 如下所示是一个递归函数，其输入参数是一个不定长的字符串，该函数将判断字符串内出现的圆括号是否成对匹配。函数Balanced的作用是匹配圆括号。如果字符数组中的圆括号成对匹配，则函数返回0，否则返回非0值。函数Balanced的最初调用是Balanced(string, 0, 0)。但是，下面的Balanced函数中缺少一些关键代码，请填写出来。

```
int Balanced(char string[], int position, int count)
{
  if (_____)
    return count;

  else if (string[position] == _____)
    return Balanced( string, ++position, ++count);

  else if (string[position] == _____)
    return Balanced( string, ++position, --count);

  else
    return Balanced( string, ++position, count);
}
```

17.14 试问，如下C程序的输出结果是什么？

```
#include <stdio.h>

void Magic(int in);
int Even(int n);

int main()
{
   Magic(10);
}

void Magic(int in)
{
   if (in == 0)
      return;
   if (Even(in))
      printf("%i\n", in);
   Magic(in - 1);
   if (!Even(in))
      printf("%i\n", in);
   return;
}

int Even(int n)
{
   /* even, return 1; odd, return 0 */
   return (n % 2) == 0 ? 1 : 0;
}
```

第18章　C语言中的I/O

18.1　概述

任何有用的程序，都需要对外输出信息，或是输出至屏幕，或是输出至文件，甚至可能输出至网络上的另一台计算机。另外，很多程序还需要用户输入数据才能运行。但是，如同大多数的现代编程语言一样，C语言本身并不支持输入输出操作。相反，输入/输出（简称I/O）的处理功能是由标准库函数提供的。而ANSI C标准，则对这些库函数的功能和接口做了标准化的定义。

本章将介绍几种标准的I/O库函数。其中，putchar和printf的功能是输出信息，getchar和scanf的功能是读取输入信息。更通用的函数形式，如fprintf和fscanf，则是执行文件I/O，如磁盘上的文件。有关printf和scanf，更是在本书第二部分内容中到处可见。本章将对这些函数的实现细节展开讨论，对其中的可变参数，及其LC-3栈的参数传递机制，做详细的分析研究。

18.2　C标准库函数

标准库函数只是C语言的扩展内容，它负责输出/输出、字符串操作、数学函数、文件存取和系统调用等功能。标准库函数针对的不是某个特定程序。相反，它的目的是向所有程序提供一套通用接口。我们可以将标准库看做是一个仓库，它为用户构建复杂软件提供了必需的部件(component)。现在流行的程序设计语言，如C++和Java，都是通过库函数的形式向用户提供的函数接口。附录D.9中是常用的一些C库函数。库函数也是程序，它们的编写者通常是那些操作系统和编译器的设计者。针对不同的运行系统，编写者通常对这些库函数做了精心优化。

所有的标准库函数，根据它们功能的不同，划分为多个组。而每个功能组，又对应一个头文件(header file或*.h文件)。如果要使用特定组的库函数，则编程者要在自己的程序中包含对应的头文件。例如，如果要使用数学函数sin和tan，则包含math.h头文件；再如，标准输入输出函数，对应的是stdio.h文件。头文件的内容是函数声明及相关的宏定义，但不包含这些函数的真正实现代码。

问题是，如果头文件中没有库函数的源代码，那么最终的程序映像中，从哪里获取这些库函数（如printf）的代码？答案是，函数库中的代码通常都是二进制形式的，存储在系统指定的目标文件中。只有链接程序能访问它们（事实上，只有链接程序能读懂它们的格式）。在程序编译的最后阶段（即链接阶段），由链接程序负责，将库函数的二进制代码和用户程序的二进制代码拼装在一起，形成最终的可执行程序。

顺便说一下，以上场景描述的是程序的静态链接方法，而很多情况下，库函数还可以“动态”地被链接（这种情况下，函数库的格式是动态链接库（dynamically linked libraries, DDL）或共享库）。在动态链接库方式下，库函数的二进制代码不出现在可执行代码中，而是在程序运行时，按需装入内存。

18.3　字符I/O操作

我们先介绍两个最简单的I/O函数：getchar和putchar，它们每次只处理一个字符。类似LC-3的IN和OUT 指令，getchar每次读入一个输入字符的ASCII码值，putchar则输出字符的ASCII码值。

18.3.1 I/O流

概念上，所有基于字符的输入/输出操作都是“流”方式的。例如，从键盘顺序输入的ASCII字符，就是一个流输入的例子。用户每输入一个新字符，该字符都被添加到流的尾部，而键盘读取则是从流的头部获取数据；再如，ASCII字符的打印也是流操作，每次输出的字符也是添加到输出流的尾部。所谓“流”，简单地说，就是操作系统为I/O设备创建的一个FIFO(先进先出)缓冲。基于抽象概念的“流”，我们可以建立起“生产者”（如键盘输入）和“消费者”（如键盘读取程序）的配对角色，并因此解决了输入和读取之间的速度不匹配问题（参见第8章）。以一个字符输出程序为例，如果没有输出流的存在，则程序每输出一个字符之前，都必须确认前一个字符的输出操作是否已完成。而有了输出流，程序只需要直接将字符输出到输出流中即可，而不必关心输出设备的状态。其他语言，如C++，也都提供类似的I/O流抽象。

在C语言中，标准输入流就是stdin。默认情况下，stdin代表的就是键盘。而标准输出流是stdout，默认情况下，它代表显示器。函数getchar的任务，就是从stdin中读取一个字符；函数putchar的任务则是向stdout输出一个字符。

18.3.2 putchar函数

函数putchar和LC-3的OUT指令相似，它的功能是将参数（一个字符）输出到标准输出流中，由于传递给putchar函数的参数值已是一个ASCII码，所以不需要对它做“类型转换”。例如，如下几行putchar函数的代码是等价的。它们的输出结果相同，即字符“h”。putchar和其他函数（如自己编写的函数）没有本质区别，调用方式也一样。惟一不同的是，它是一个标准库函数（在头文件stdio.h中声明），而它的二进制代码也是在程序编译的链接阶段才被加入可执行代码。

```
char c = 'h';

:
putchar(c);
putchar('h');
putchar(104);
```

18.3.3 getchar函数

函数getchar与LC-3的IN指令相似。该函数的返回值，是排在标准输入流（stdin）头部的ASCII码值。默认情况下，stdin对应键盘设备的输入流。如下代码表示的是，getchar返回的是下一个键盘输入字符（ASCII码值），并将该值赋给变量c。

```
char c;

c = getchar();
```

18.3.4 缓冲I/O

如图18-1所示程序，如果运行一遍该程序，你将发现程序的怪异行为。程序提示用户输入一个字符并等待，而用户键入一个字符（如“z”）之后，却发现第二个提示信息始终不输出(getchar函数似乎没有被执行)？但是，如果我们再输入一个回车键，程序立刻恢复执行，getchar函数存在什么问题吗？

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char inChar1;
```

图18-1 缓冲输入的例子

```
 6    char inChar2;
 7
 8    printf("Input character 1:\n");
 9    inChar1 = getchar();
10
11    printf("Input character 2:\n");
12    inChar2 = getchar();
13
14    printf("Character 1 is %c\n", inChar1);
15    printf("Character 2 is %c\n", inChar2);
16  }
```

图18-1 缓冲输入的例子（续）

原因是键盘输入流的缓冲效应。几乎所有的系统都采用缓冲I/O机制，即每个键盘输入都先被操作系统代码捕获，然后放到系统内部的一个缓冲区中。直到用户键入回车键，系统才清空缓冲区，并将缓冲区内容添加到输入流尾部。所以，图18-1所示程序中，用户如果键入一个字符（如“A”），然后再键入“回车符”，则inChar1的内容为“A”（ASCII码65）、inChar2的内容为“回车符”（ASCII码10）。

尽管I/O缓冲会造成以上“误解”，但它却是非常必要的，尤其是用键盘输入时，按下回车键，可让用户确认其输入。在有缓冲的情况下，如果键入错误字符，用户可以通过退格键（backspace）或删除键（del）“擦除”错误，然后重新键入，直到用户最后键入回车键。

同样，输出流也具备类似的缓冲。读者可以尝试图18-2所示的例子，观察输出缓冲时的表现。

```
 1  #include <stdio.h>
 2  #include <unistd.h>
 3
 4  int main()
 5  {
 6    putchar('a');
 7
 8    sleep(5);
 9
10    putchar('b');
11    putchar('\n');
12  }
```

图18-2 一个缓冲输出的例子

其中，该程序使用了一个新函数sleep。sleep的作用是“暂停”程序执行，暂停时间由sleep的参数指定（单位为秒）。如果要使用该函数，程序就要包含头文件unistd.h。程序运行时发现，字符“a”没有立刻输出，而是经过大约5秒后，与字符b同时输出。这是因为，只有字符“\n”能清空输出缓冲区，将字符序列添加到输出流中。我们可以尝试在第6行代码后面，加入一行“putchar('\n');”语句，然后观察前后差别。

尽管存在细微的I/O流缓冲特性，putchar/getchar的底层实现机制其实就是第8章中描述的IN/和OUT TRAP程序，流的缓冲机制则是由围绕IN和OUT服务程序的上层软件实现的。

18.4 格式化I/O

putchar和getchar只能完成简单的字符I/O操作，无法实现“非ASCII码”数据的I/O操作。在C标准库中，函数printf和scanf则支持各种数据类型的I/O，以及更高级的格式化I/O操作。

18.4.1 printf

printf的功能是将格式化的文本输出至输出流中。通过printf函数，我们可以将程序中产生的各种数值，嵌在ASCII文本中，混合打印输出。为此，printf必须具备将各种类型转换为ASCII码的能力。例如，如下代码中，为打印整数变量x，printf必须将整数转换为ASCII字符序列，并将其嵌入

在整个输出字符流中。

```
int x;

printf("The value is %d\n", x);
```

简单地说，就是printf将它的第一个参数写入输出流。printf的第一个参数是“格式串”(format string)，它的类型是字符串（即类型char*），包含的内容是要输出的文本，以及“转换规范”(conversion specifications)。

所谓“转换规范”，其作用是指明“格式串”之后各参数的类型转换方式。转换规范以符号“%”为前缀，“%”后面的字符代表被转换的数据类型。例如，我们已列举的例子中曾使用的“%d”，它表示将一个十进制数转换为ASCII码字符串；再如，转换规范“%x”表示将十六进制数转换为ASCII码字符串，“%b”表示被转换的是二进制数。此外，还有诸如%c、%s、%f等转换规范，%c代表ASCII字符（char），%s代表字符串（string），%f代表浮点数（float）。顺便问个问题，如果“%”表示的是转换规范的开始，那么怎样才能输出字符“%”本身呢？答案是“%%”。可参考附录D列出的各种转换规范及其说明。

在第11章中，我们曾提到，格式串中还可以包含特殊字符，如换行符“\n”。特殊字符的前缀是“\”，它们可以放在格式串的任何位置。例如，“\n”代表换行符（new line），“\t”代表制表符(tab)。那么，如果“\”代表特殊序列的开始，我们怎样输出字符“\”本身呢？答案是“\\”。可参考附录D中表D-1的各种特殊字符。

如下是各种格式转换规范的使用例子：

```
int  a = 102;
int  b = 65;
char c = 'z';
char banner[10] = "Hola!";
double pi = 3.14159;

printf("The variable 'a' decimal : %d\n", a);
printf("The variable 'a' hex : %x\n", a);
printf("The variable 'a' binary : %b\n", a);
printf("'a' plus 'b' as character : %c\n", a + b);
printf("Char %c.\t String %s\n Float %f\n", c, banner, pi);
```

printf从左到右扫描整个格式串。如果当前字符不是“%”或“\”，则将该字符直接添加到输出流中（由于缓冲，输出字符直到遇到换行符，才显示在输出设备上）；如果当前字符是“\”，则将下一个字符看做特殊字符，如“\n”表示换行；如果当前字符是“%”，则下一个字符表示转换规范。例如，如果转换规范为“%d”，且其对应参数的bit值是“0000000001101000”，则该bit串被转换为字符串“104”；如果转换规范是“%c”，则该“bit串”被解释为字符“h”；转换规范如果是“%f”，则结果又将不同（解析过程有些复杂）。值得一提的是，转换规范只表示如何解释后续参数，而与后续参数的原本类型没有任何关系。换句话说，程序员可以选择任何解释方式。*思考题*：如下代码的输出结果如何？

```
printf("The value of nothing is %d\n");
```

其中，格式串中包含有转换规范“%d”，但printf没有提供配对的后续参数。函数执行时，printf将认为后续参数已被放入执行栈中。所以，它会试图从栈中读取与%d相应的数据，然后将转换结果加入输出流。但事实上，我们并未在栈中为它存放数据。结果是，printf从栈中读到一个不确定的数值，所以输出结果也是一个不确定的十进制数。

18.4.2 scanf

函数scanf的作用是读入格式化的输入流中的ASCII数据。scanf的调用方式与printf相似，第一个参数都是格式串（包含解释后续参数的转换规范），之后的参数与格式串中的转换规范一一对应。

不同之处是，scanf中后续参数必须是指针。这一点，我们在第16章曾提到过，因为scanf要为这些变量赋值，所以必须将变量所在的内存位置（即指针）告诉scanf。

scanf采用与printf相同的格式串及转换规范定义，可参考附录D的说明。在scanf中，转换规范代表输入流数据（ASCII码）要被转换的目标格式。例如，转换规范“%d”，表示将输入流中的下一个不包含“非空字符”（注：空格、回车都是“空字符”）组成的数字序列，转换成一个十进制整数。按照这个转换规范所示，scanf从输入流中连续读取一串十进制数字，然后将转换后的数值赋给对应参数。由于scanf将修改参数变量的值，所以传递的是变量的地址。除转换规范之外，格式串中也可以包含普通字符，scanf将它们用于与输入流匹配。如下所示是scanf的使用例子：

```
char name[100];
int month, day, year;
double gpa;

printf("Enter : lastname birthdate grade_point_average\n");
scanf("%s %d/%d/%d %lf", name, &month, &day, &year, &gpa);

printf("\n");
printf("Name : %s\n", name);
printf("Birthday : %d/%d/%d\n", month, day, year);
printf("GPA : %f\n", gpa);
```

其中，scanf的第一个转换规范“%s”表示从输入流读取一个字符串。在输入流中，它对应的范围是第一个非空字符至下一个空字符之间的字符串。读取之后，scanf将在字符串最后，自动添加字符串结束符“\0”，然后将结果拷贝至字符数组name。

第二个转换符是“%d”。这次读取的输入流范围是，自下一个非空字符开始的一个连续数字字符序列。与“%s”不同的是，对应‘%d’的字符串，最后不需要添加结束标志，我们直接将这个序列转换成整数，然后保存在变量month中。

随后，遇到字符“/”。它不是转换规范，所以不需要与任何变量参数匹配，更不需要为任何变量赋值。所以，scanf从输入流中读入该字符后，直接丢弃，然后继续后面的匹配。依此类推，是第三、第四个转换规范的匹配，匹配结果分别保存在变量day和year中。注意，它们之间由字符“/”分隔。

最后一个转换规范是“%lf”。这意味着scanf将输入流对应的数字序列解释为一个双精度浮点数，它的输入格式既可以是十进制小数形式，也可以是E指数形式的科学记数法（参见附录D.2.4）。该字段以一个非数字字符（不包括第一个E、小数点及正负号）或空字符结束。scanf将最后读入的序列转换为浮点数，然后保存在变量gpa中。

以上各匹配完成之后，scanf返回调用值。scanf的返回值是一个正整数，代表已被正确匹配的个数。如在本例中，返回值为5。

例如，在以下输入时，程序的输出为：

```
Enter : lastname birthdate grade_point_average
Mudd 02/16/69 3.02

Name : Mudd
Birthday : 2/16/69
GPA : 3.02
```

如果输入如下所示，由于scanf忽略空字符（white space），因此输出结果不变：

```
Enter : lastname birthdate grade_point_average
Mudd     02
/
16 / 69     3.02

Name : Mudd
Birthday : 2/16/69
GPA : 3.02
```

试问，如果输入格式与scanf指定的格式不匹配，结果会如何呢？例如：

```
Enter : lastname birthdate grade_point_average
Mudd 02 16 69 3.02
```

其中，输入流中没有字符“/”，即输入流格式和指定格式不匹配。结果是，scanf返回值为2（即成功匹配2个参数），所以name和month被正确赋值，day、year和gpa则失败。由于输入流是被缓冲的，所以输入流中的字符“”、“1”、“6”等并未被丢弃。所以，如果我们继续用getchar函数读取输入流的话，还可以将它们读出：

```
a = getchar();
b = getchar();
```

结果是，a = “ ”，b = “1”。

18.4.3 可变长参数

至此，你可能发现函数printf和scanf与其他函数有些不太一样，即它们的参数个数不确定。scanf和printf的参数个数，由它们的第一个参数“格式串”决定。我们称这种函数为“可变长参数函数”（variable argument lists）。

在这种函数中，格式串中的转换规范与之后参数之间存在一对一的映射关系。如下所示：

```
printf("Char %c.\t String %s\n Float %f\n", c, banner, pi);
```

其中，格式串中包含三个转换规范，与此对应，后面也是三个参数。转换规范“%c”对应变量c，“%s”对应变量banner，“%f”对应变量pi。所以，我们说传递给printf的参数是4个，打印输出的数目是3个。如果打印输出的是5个数值，则传递给printf的参数数目必然是6个。

回顾第14章中介绍过的LC-3参数传递规范，参数入栈的顺序是*自右向左*。在printf和scanf函数中，格式串参数位于最左边，所以它最后入栈。当程序执行转入函数时，格式串出现在栈顶，所以通过栈顶格式串内容的分析，就可以推算出栈内参数的个数和类型。而如果函数参数传递顺序是从左到右，则格式串的分析过程就非常困难。图18-3所示是printf执行时的栈情况，图（a）表示自右向左的参数传递方式，图（b）表示的则是自左向右的传递方式。在图（a）中，无论参数个数是多少，格式串所在的栈偏移始终为0；而在图（b）中，格式串的偏移难以确定，它与参数个数密切相关。

另外，格式串是一个字符串常量，与其他嵌在代码中的字符串常量一样，它们位于系统中的一个特殊内存空间（全局数据区），即专门用于存放常量（constant）或*文本值*（literal value）的空间。

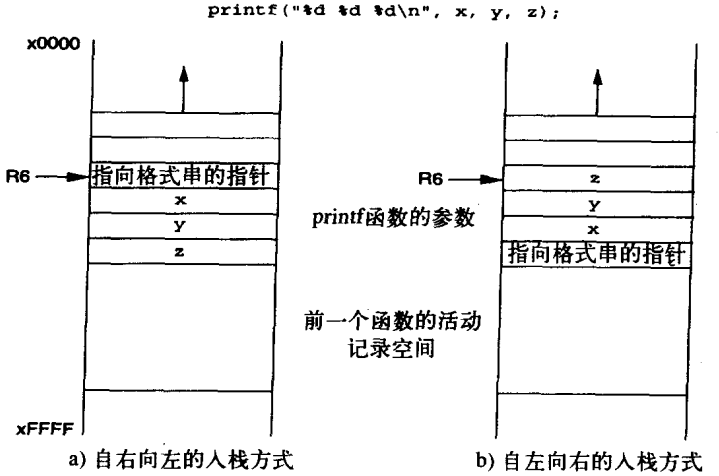


图18-3 入栈方式

18.5 文件I/O

在处理大批量数据的情况下，比如统计20年以来IBM公司股票的日收盘价格。显然，通过手工输入这些数据是不现实的（让用户键盘输入“这么多”数据，是非常“不友好的”（user-unfriendly））。此时，就需要文件I/O方式。数据存放在磁盘文件中，而处理结果也将写入文件。如前所述，C的I/O是面向流的，现在要做的事情是，怎样将流与特定文件相关联。

事实上，前面所介绍的printf和scanf函数，只是许多通用C I/O函数中的特例。换句话说，它们操作的只是两个特殊文件stdin和stdout。而在C语言中，文件stdin代表的是键盘设备，stdout代表的是显示器设备。

更通用版本的printf和scanf函数是fprint和fscanf。它们的用法与printf/scanf相似。不同的是，在printf和scanf中，默认的关联文件分别是stdout和stdin；而在fprintf和fscanf中，关联文件要由程序员显式给出。例如，我们可以指定fprintf将输出内容写入指定的文件中。下面，我们将介绍文件I/O的操作机制。

首先，在文件I/O之前，我们需要声明一个文件指针。如下所示，假设我们声明一个名为infile的文件指针：

```
FILE *infile;
```

其中，类型FILE是在stdio.h头文件中定义的一个数据类型。有关FILE的详细信息，已超出本书讲述范畴，故不再展开。

之后，我们将该文件指针与特定的磁盘文件相关联。关联操作通过fopen函数实现。如下所示，fopen函数有两个参数：一是文件名，二是文件操作类型说明。

```
FILE *infile;

infile = fopen("ibm_stock_prices", "r");
```

其中，第一个参数是文件名“ibm_stock_prices”，第二个参数是我们将对该文件采用的操作模式（mode）。模式有很多类，如“r”代表只读（read-only）；“w”代表写（write）。如果以这种模式打开，则文件的原内容将丢失；“a”代表文件附加（append）。即文件原内容不变，新数据附加在文件结尾；“r+”代表可读可写。

函数fopen的调用如果成功，则返回一个文件指针；否则，函数返回一个空指针。所以，程序员应该培养好习惯，每次调用fopen后，都应该检查返回值即文件指针是否为空。如下面代码所示：

```
FILE *infile;

infile = fopen("ibm_stock_prices", "r");

if (infile == NULL)
   printf("fopen unsuccessful!\n");
```

此时，函数fopen的成功返回，意味着文件指针已与一个物理文件有效“映射”。基于该文件指针，我们就可以通过fscanf和fprintf函数读、写文件了，这与我们通过scanf和print函数读写标准设备的方法一样。如图18-4所示，fscanf和fprintf的第一个参数都是FILE * 类型，指示函数所要操作的流。

这里，程序读取的是一个名为“ibm_stock_prices”的ASCII文本文件，写的是一个名为“*buy_hold_or_sell*”的文件。从代码中，我们可以看出，文件ibm_stock_prices包含的是一组浮点数据，且浮点数之间由空格分隔。尽管该数据文件可能很长，但该程序最多只读入（scanf）10000个数据项。如果文件中不再有数据了，则fscanf会返回一个特殊值。所以，代码中会特地检查这个返回值（预处理器将这个值定义为宏EOF）。while循环的终止条件是：遇到EOF（End-Of-File）字符，或读入字符的数目已超过极限值（10000）。当文件内容被全部读入后（prices[]数组），程序将对它们做出运算处理（程序中省略的部分），最后再将处理结果（answer串）写入文件。

如果fprintf的第一个参数为stdout，则与printf完全等价。同样，如果fscanf的第一个参数为stdin，则与scanf完全等价。

```
#include <stdio.h>
#define LIMIT 10000

int main()
{

  FILE *infile;
  FILE *outfile;
  double prices[LIMIT];
  char answer[10];
  int i = 0;

  infile  = fopen("ibm_stock_prices", "r");
  outfile = fopen("buy_hold_or_sell", "w");

  if (infile != NULL && outfile != NULL) {
    /*  Read the input data */
    while ((fscanf(infile, "%lf", &prices[i]) != EOF) && i < LIMIT)
      i++;

    printf("%d prices read from the data file", i);

    /* Process the data... */
      :
      :


    /* Write the output */
    fprintf(outfile, "%s", answer);
  }
  else {
    printf("fopen unsuccessful!\n");
  }
}
```

图18-4 文件I/O的一个例子

18.6 小结

本章的主题是C的I/O接口使用。与其他语言一样，C本身并不提供I/O能力。相反，它们是由函数库提供给用户调用的。而这些I/O，最终是由底层IN/OUT指令来完成的。

本章的关键内容是：

- 输入/输出流。几乎所有的现代语言，都对I/O进行了抽象，于是形成这样的操作规范，输入/输出的对象是“流”，生产者（producer）将数据写入流的尾部，而消费者（consumer）从流的头部读取数据。生产者和消费者之间，各自做自己的事情，无需相互等待，也不必协商设备的互斥访问问题。例如，程序如果向显示器输出数据，它所要做的事情只是将数据写到输出流即可，而不必在每次写入之前，都确认输出设备是否空闲。
- 四个基本I/O函数。本章非常详细地讨论了C语言中4个基本I/O函数的操作机制：putchar、getchar、printf、scanf。其中，后面两个是变长参数函数。LC-3的参数传递规范，能够很容易地处理不定长的参数数目，因为它的参数入栈顺序是自右向左的。请回顾一下其中的道理。
- 文件I/O。在C语言中，所有的“I/O流”都属于“文件I/O”方式。函数printf和scanf也不例外，只是在printf和scanf中，默认的I/O文件是标准输出和标准输入设备而已。通用的文件I/O函数是：fprintf和fscanf。在这两个函数中，第一个参数指定了操作的文件指针。该文件指针通过fopen函数与一个物理文件相关联（bind）。

18.7 习题

18.1 试调用I/O函数完成以下任务。所有任务每次只能调用一次I/O函数。

a. 在屏幕上，依次打印一个整数、一个字符串和一个浮点数；

b. 在屏幕上，打印输出格式如“(XXX) -XXX-XXXX”的电话号码。提示：电话号码包括区号、交换局号、端号等三部分，分别由三个整数变量表示；

c. 在屏幕上，打印输出格式如“XXX-XX-XXXX”的学生ID，每个ID段是一个长度为3的字符串；

d. 以格式“XXX-XX-XXXX”的方式，从键盘读入学生ID，每个ID字段分别存放在三个整数中；

e. 从键盘读入一连串的个人信息，其中包括姓名、年龄、性别等字段。字段之间由逗号分隔。其中，姓名和性别存储为字符串格式，年龄为整数。

18.2 试问，函数scanf的返回值代表什么含义？

18.3 试问，为什么要对键盘输入流进行缓冲？

18.4 试问，从一个内容为空的输入流中读取数据，结果会如何？

18.5 运行如下代码，会打印输出奇怪的数据（如：1073741824），请解释其中原因。

```
float x = 192.27163;
printf("The value of x is %d\n", x);
```

18.6 运行如下代码。试问，输入字符串“This is not the input you are looking for.”之后，input的值是什么？

```
scanf("%d", &input);
```

18.7 阅读如下代码，回答问题：

```
#include <stdio.h>

int main()
{
  int x = 0;
  int y = 0;
  char label[10];

  scanf("%d %d", &x, &y);
  scanf("%s", label);

  printf("%d %d %s\n", x, y, label);
}
```

a. 如果输入为“46 29 BlueMoon”，程序运行结果如何？

b. 如果输入为“46 BlueMoon”，程序运行结果如何？

c. 如果输入为“111 999 888”，程序运行结果如何？

18.8 试编写程序，这个程序读入一个“C语言的源代码文件”。要求，程序能删除源文件中的空格，并将去除空格后的文件内容写入另一个文件中（文件名设定为“condensed_program”）。

18.9 试编写程序，该程序以一个文本文件为输入，并完成以下计数：

a. 文件中字符串的个数。所谓“字符串”，头部为一个空字符，结尾处也是一个空格；

b. 文件中英文单词的个数。所谓“英文单词”，以字母a~z或A~Z开头，以任意非字母字符结束；

c. 文件中惟一出现的英文单词的个数。所谓“英文单词”，参考问题b的定义。所谓“惟一”，表示该单词在文件中出现且仅出现一次；

d. 统计文件中英文单词出现的频度，并按照频度的高低，有序输出这些单词及其频度。换句话说，就是扫描该文本文件，记录文件中出现的每一个单词，并记录它已出现的次数；最后，将这些单词按照它们出现次数的大小，依次（从大到小）打印输出这些单词及其出现的次数。

第19章 数据结构

19.1 概述

C语言的核心中，只支持三种基本数据类型：整数、字符和浮点数[⊖]。这意味着，C本身支持这些类型变量的内存分配，及其与类型相关的运算符，如加法运算符“+”和乘法运算符“*”等。随着我们对本书第二部分内容的学习，体会到了扩展基本类型的必要性，如指针（pointer）和数组（array）等扩展类型的引入。指针和数组类型源于三种基本类型。例如，指针可指向三种基本类型中的任何一种类型的变量；类似地，我们也可以将数组声明为int、char或double。

编程者的任务是，编写能够解决实际问题的程序。其中，关注的对象包括飞机机翼、人群或迁徙中的鲸鱼。问题是，底层的计算机硬件只能处理如整数、字符或浮点数等数据类型。所以，程序员必须设法解决现实对象和基本数据类型之间的映射问题，这是个繁重的任务。编程语言的任务之一，就是在两者之间进行沟通。提供对现实对象的描述能力，并为之定义操作方法，是面向对象方法的两个基础任务。

所谓“面向对象的编程”，就是让程序设计围绕着“对象”展开（而不是围绕计算机硬件所能提供的基本数据类型），这就是面向对象编程的核心法则。本章我们要做的是，朝着“面向对象”走很小的一步，即学习怎样基于多种基础数据类型构建新的数据类型。在C语言中，我们称这种聚合方式为“结构体”（structure）。结构体的作用是方便对象的描述，一个结构体（或对象）中可以包含不止一个数值。例如，在一个企业数据库程序中，将一个“员工”（employee）对象表示为“结构体”，则包含：姓名（字符串）、职位（字符串）、部门（可能是整数）、工号（整数）等数据。所以，在设计这样的数据库系统时，我们完全有理由使用C结构体。

本章的主题是C语言的高级数据结构。首先我们将介绍，在C程序中如何创建结构体变量，以及如何创建结构体数组；然后，是C中的动态内存分配。虽然动态内存分配与数据结构之间不存在必然联系，但是后面的链表结构中，将涉及有关概念。最后一项内容是“链表”（linked list），它也是一种非常基础和常用的数据组织方式。该结构和数组很相似，都是存储一组数据。不同之处在于数据的组织形式不同。我们还将介绍链表的几种基本操作：添加、删除和查找。

19.2 结构体

对于许多事物来说，最好的描述方法是多个基本类型的聚合。针对这些对象，C语言提供“结构体”的概念。通过“结构体”方法，程序员可以随时定义新的数据类型，它可以包含多个数据项，且这些数据项的类型可以是int、char或double，甚至可以是这些基本类型的指针或数组。结构体变量的声明方式，与基本类型变量的声明方式相同。只是在任何结构体变量被声明之前，该结构体的组织结构及其各成员的类型、命名等，都必须事先定义。

下面，我们以飞机航班的描述为例（在飞行模拟器或航空管理等程序中需要）。首先，我们要确定与应用程序相关的一些飞机属性，如：航班号，用来识别飞机的身份，通常由数字或字符的序列表示；另外，飞行高度、经纬度、航向等属性也很重要，它们可以表示为整数；再就是，飞

⊖ 枚举（enumeration）也是一种基本类型，但与整数类型非常接近。

行速度，可以表示为浮点数。于是，一个航班就可以表述为以下变量的集合：

```
char flightNum[7];  /* Max 6 characters        */
int altitude;       /* in meters               */
int longitude;      /* in tenths of degrees    */
int latitude;       /* in tenths of degrees    */
int heading;        /* in tenths of degrees    */
double airSpeed;    /* in kilometers/hour      */
```

如果程序需要同时描述多个航班飞机，则可以为每个航班拷贝一份上面的代码（适当修改变量名即可），但这样的代码将显得非常冗长。如果采用C语言提供的struct结构体，则可以将所有这些属性聚合成一个类型，如下所示：

```
struct flightType {
  char flightNum[7];  /* Max 6 characters        */
  int altitude;       /* in meters               */
  int longitude;      /* in tenths of degrees    */
  int latitude;       /* in tenths of degrees    */
  int heading;        /* in tenths of degrees    */
  double airSpeed;    /* in kilometers/hour      */
};
```

通过以上声明，我们定义了一种新的数据类型。它包括6个成员（member）。注意，此时我们并未为它申请空间，只是告诉编译器有这样一种新的数据类型，它的类型名称是“flightType”，以及它的成员组成。

为这个新类型声明一个变量的方法，如下所示：

```
struct flightType plane;
```

该语句声明了一个变量plane。事实上，该变量本身包含了6个数据项。除此之外，结构体变量的各种使用方法与传统变量没有什么区别。

对结构体变量内部成员项的访问方式，如下面代码所示：

```
struct flightType plane;

plane.airSpeed = 800.00;
plane.altitude = 10000;
```

其中，对每个成员的访问（读或写），标识方式是“变量名”，然后是连接符“.”，然后是“成员名”。

如果plane是局部变量，则它的分配空间是在“栈”中，且是一段足够大的连续空间（以存放下所有的成员）。如在上面例子中，由于每个基本类型占用一个内存单元，所以变量plane占用的空间总大小是12个内存单元。

结构体变量的空间分配并没有什么特殊之处。结构体变量的空间分配与基本类型相同：局部变量分配在栈空间中，全局变量分配在全局数据空间中。图19-1所示是如下声明代码对应的栈使用情况：

```
int x;
struct airplaneType plane;
int y;
```

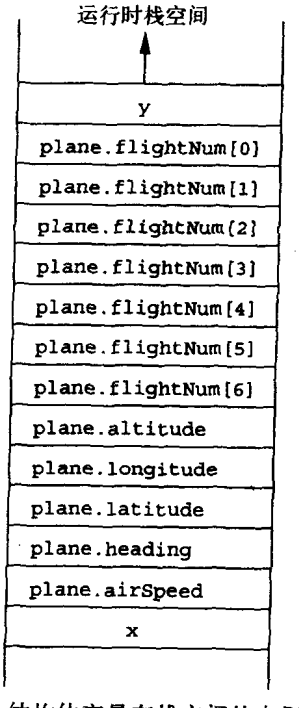


图19-1 结构体变量在栈空间的占用情形

结构体声明的语法格式如下所示：

```
struct tag {
  type1 member1;
  type2 member2;
  ...
  typeN memberN
} identifiers;
```

其中，“tag”代表特定“结构体”，它只是一个标识，

为以后的代码引用提供了一个“抓取”该结构体的“句柄”（handle）；成员列表（member），则定义了该结构体的组织方式。从语法上讲，就是一组声明。每个成员的类型是任意的，甚至可以是一个“结构体”类型；最后，是“标识”（identifier），它们是可选项，代表的是为该结构体类型声明的一个或多个变量。注意，这些变量“标识”的位置是夹在‘}’和‘；’之间的。

19.2.1 typedef

如果说“结构体”为程序员提供了一种任意创建新数据类型的机制；那么，typedef则提供了一种为已有数据类型重新命名的机制。typedef语句的形式如下所示：

```
typedef type name;
```

该语句的作用是，为已有数据类型type创建了一个别名name。其中，type既可以是一个基本数据类型，也可以是由程序员创建的数据类型（如：结构体）。例如：

```
typedef int Color;
```

其中，Color仅仅是int的一个别名。换句话说，代码中出现的Color，都可以直接替换为int，反之亦然。这样，我们就可以使用Color来定义变量了。例如，我们可以这样定义位图像素：

```
Color pixels[500];
```

同样，typedef还可用于程序员自定义的数据类型。例如，我们将之前定义的结构体flightType命名为Flight：

```
struct flightType {
  char flightNum[7];  /* Max 6 characters        */
  int altitude;       /* in meters               */
  int longitude;      /* in tenths of degrees    */
  int latitude;       /* in tenths of degrees    */
  int heading;        /* in tenths of degrees    */
  double airSpeed;    /* in kilometers/hour      */
};

typedef struct flightType Flight;
```

在语法上，Flight和struct flightType完全等价。例如，我们可以使用Flight声明结构体变量plane：

```
Flight plane;
```

该变量声明语句，与之前的“struct flightType plane”声明语句完全等价。

除此之外，typedef并没有更多的作用。C引入typedef的目的，就是使代码更具可读性，尤其是在程序中自定义数据类型较多的情况下。为数据类型选一个好名字，使得该数据类型的含义更加明确（这与好的变量名的作用相同）。

19.2.2 结构体在C中的实现

前面，我们介绍了结构体的声明和变量分配技术，我们的关注点是其成员字段的访问和运算操作，以及通过typedef来自定义数据类型。我们将继续介绍如何操作结构体的成员，例如，如下代码是对类型为Flight的变量的altitude成员的访问操作：

```
int x;
Flight plane;
int y;

plane.altitude = 0;
```

其中，plane是一个Flight类型的变量。由Flight结构体的定义可知，plane包含6个成员。对其成员altitude的访问方法是，在变量名后加操作符“.”，然后是成员名。编译器事先知道该结构体的布局。所以，通过一定的偏移，即可找到变量的某个成员。如图19-1所示，编译器在符号表中，记

录每个变量相对基地址（R5）的偏移量。而如果该变量是结构体类型，符号表中还将记录每个成员在结构体变量中的偏移。例如，“plane.altitude = 0；”语句，编译器要访问的对象是，栈中第2个变量的第2个成员。

如下所示是LC-3编译器为该语句（plane.altitude = 0；）生成的汇编代码：

```
AND  R1, R1, #0    ; zero out R1

ADD  R0, R5, #-12 ; R0 contains base address of plane
STR  R1, R0, #7    ; plane.altitude = 0;
```

19.3 结构体数组

假设我们要写一段代码，判断芝加哥上空是否存在飞机相撞的可能。编写中，我们使用了结构体Flight，并假设同一时刻，芝加哥上空最多有100架飞机。那么，我们可以声明一个数组：

```
Flight planes[100];
```

该声明与简单声明（如 int d[100]；）完全相似。只是，后者声明的是100个整数，而此处声明的是100个结构体（每个结构体又包含6个成员）。例如，plane[12]代表的是，内存中100个结构体中的第13个结构体。每个结构体的单元空间大小应足够保存它的6个成员。

在该数组中，每个数组元素的类型都是Flight，且可以按照标准的数组方式来访问。例如，要访问第一个飞机的属性，则用planes[0]即可。而如果要访问其中的成员，则加上后缀成员名即可。例如，访问第一个飞机的航向信息，则为planes[0].heading即可。如下所示是计算所有飞机的平均速度的代码：

```
int i;
double sum = 0;
double averageAirSpeed;

for (i = 0; i < 100; i++)
   sum = sum + plane[i].airSpeed;

averageAirSpeed = sum / 100;
```

此外，我们还可以创建指向结构体的指针。如下所示定义的指针，包含的是一个类型为Flight的结构体变量所在的地址：

```
Flight *planePtr;
```

结构体指针的赋值操作，与其他指针变量相同。

```
planePtr = &plane[34];
```

如果我们要访问该指针变量所指向的结构体变量的成员，可采用如下表示方式：

```
(*planePtr).longitude
```

其中，做了一个变量planePtr的“间接引用”（dereference）操作。它指向的是类型为Flight的对象，如果对它做间接引用，则访问的就是一个类型为Flight的对象。对其成员的访问，则通过符号“.”实现。我们在后面会看到，通过指针引用一个结构体是个非常常用的操作。但由于该表示方法的非直观性，我们专门为此操作定义了一个运算符。于是，上面的表达式等价为如下表达式：

```
planePtr->longitude
```

在此，表达式“->”等价于间接引用符“*”。不同之处在于，“*”可以用于任何类型的指针，而“->”仅限于对结构体中成员的引用。

下面，我们将之前有关结构体的讨论实际应用起来。要求解的实际例子是，通过对100架飞机的状态分析，检测它们之间存在碰撞危险的可能性。为此，我们要对每架飞机的高度、经度、纬度及航向都做分析，才能判断出相互碰撞的可能。如图19-2所示，在函数PotentialCollisions中，通过调用Collide判断飞机航道之间是否存在交叉（本例中的代码是不完整的，我们将它作为练习，

留给读者。请写出更精确的、判断两机航道之间是否存在交叉的代码)。

值得一提的是，PotentialCollisions传递给Collide的两个参数是两个指针（而不是结构体）。虽然直接传递结构体也是可行的，但传递指针的效率更高。因为，在传递指针的方式下，只需要将两个指针压入栈中（而不必压入两个Flight结构体即24个内存单元）。

```
#include <stdio.h>
#define TOTAL_FLIGHTS 100

/* Structure definition */
struct flightType {
  char flightNum[7];  /* Max 6 characters        */
  int altitude;       /* in meters               */
  int longitude;      /* in tenths of degrees    */
  int latitude;       /* in tenths of degrees    */
  int heading;        /* in tenths of degrees    */
  double airSpeed;    /* in kilometers/hour      */
};

typedef struct flightType Flight;

int Collide(Flight *planeA, Flight *planeB); void
PotentialCollisions(Flight planes[]);

int Collide(Flight *planeA, Flight *planeB)
{
  if (planeA->altitude == planeB->altitude) {

    /** More logic to detect collision goes here **/
  }
  else
    return 0;
}

void PotentialCollisions(Flight planes[])
{
  int i;
  int j;

  for (i = 0; i < TOTAL_FLIGHTS; i++) {
    for (j = 0; j < TOTAL_FLIGHTS; j++) {
      if (Collide(&planes[i], &planes[j]))
        printf("Flights %s and %s are on collision course!\n",
               planes[i].flightNum, planes[j].flightNum);
    }
  }
}
```

图19-2 基于Flight结构体的一个例子

19.4 动态内存分配

在C程序中，内存对象（如变量）在内存中的分配点只有三个可能：栈、全局数据区或堆(heap)。局部变量的默认分配空间是栈；全局变量的分配空间是全局数据区，该区间由程序所有模块访问；动态数据对象（程序运行时创建的）的分配空间则是堆。

在前一个例子中，我们声明了一个能包含100个Flight结构体的数组。但是，如果我们要求程序能处理变化数目的飞机，或2个，或20 000个，应该怎样为这些数据分配空间呢？显然，我们必须按最大数目为之分配空间，即声明一个能包含20 000个Flight结构体的数组。但问题是，通常情况下，飞机数目远小于20 000，所以存在着内存空间的浪费。另一个极端则是，飞机数目如果超过20 000个，则程序崩溃。较好的处理方法应该是，根据当前飞机数目，动态分配存储数据的空间。为此，引入了“动态内存分配”的概念。

所谓“动态内存分配”，原理如下：在系统中，有一个被称做“内存分配器”(allocator)的程

序，管理着一个被称做“堆”（heap）的内存空间。如图19-3所示（这是图12-7的拷贝），它描述的是内存中，各区段（包括“堆”）之间的关系。在执行过程中，程序可以请求内存分配器，申请一段大小确定的连续内存。于是，分配器将这段空间预留下来，并将该段空间的起始地址（指针）返回给申请者。例如，我们可以向分配器申请一段能存储1000个Flight结构体的内存空间。如果堆中剩余的空间足够，则分配器返回指向该地址的指针。注意，在图19-3中，堆和栈是“头对头”增长的。栈的大小取决于函数调用的深度，而堆的大小则取决于分配器接受的“分配申请”数目。

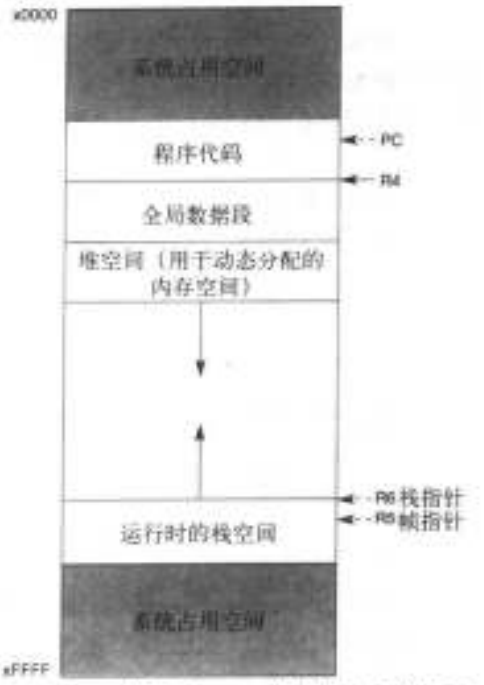


图19-3 “堆”在LC-3内存空间中的位置

顺便提一下，在堆中已被分配的内存空间将永久保留，直到程序主动释放。“内存回收器”（deallocator）的任务，就是接收程序的释放请求，并将该空间“归还”给堆，供下次申请使用。

动态大小的数组

在C语言中，动态内存分配由一个C标准库负责。例如，激活分配器的函数是malloc，参见如下代码，是malloc的使用方法：

```
int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

planes = malloc(24 * airbornePlanes);
```

其中，malloc的任务是，在堆中分配一段连续内存空间，该空间的大小如malloc的参数所指定。如果分配成功，则malloc返回一个指向该空间的指针。

在此，分配大小为“24*airbornePlanes”个字节，airbornePlanes代表飞机数目，那么24代表什么？答案是，它代表Flight结构体的大小，即6个成员的大小总和（1个字符数组（14）、4个整数（8）、1个双精度浮点数（2））。因为，在LC-3中，每单元大小为2个字节，所以该结构体大小为“2*（7+4+1）= 24”个字节。显然，程序员必须小心计算每个结构体的大小，这无疑降低了代码的可读性。为此，C语言又引入了操作符sizeof。sizeof的作用是返回其参数（一个内存对象或类型）所占空间的大小（单位为字节）。例如，sizeof(Flight)的返回值，是结构体Flight的大小（即24）。有了sizeof，程序员可以不必计算各种数据对象的大小；事实上，计算工作将由编译器完成。

如果malloc申请内存时，堆的所有空间都已被分配，则malloc返回NULL。符号NULL是一个预处理宏，代表空指针，它的具体值是多少取决于不同的系统。因此，检查malloc的返回值，对程序员来说是个好习惯。

malloc的返回值是一个指针变量，但该指针的类型是什么呢？例如，在前面的例子中，我们将malloc返回的指针看做是Flight类型的指针。如果我们为int型数组申请空间，则malloc的返回值应该被看做是int指针。为了支持各种类型的内存分配，malloc函数的返回值采用通用指针类型（即void *），而该返回值在使用时要做“强制类型转换”（type cast）。换句话说，无论什么情况下调用malloc，我们都需要告诉编译器，将返回值转换为合适的类型（而不是默认的void *）。

例如，在前一个例子中，我们就将malloc的返回指针转换为另一种类型。其中，我们为指针

planes指定的类型是Flight *，则我们实际上是强制地将malloc返回的void *类型转换为了Flight *类型。当然，这多少造成了一定的移植性方面的问题。所以，大多数编译器会为此输出一个警告信息，告诉我们“程序正在将一种类型的指针值赋给另一种类型的指针变量”。类型转换使得编译器要将一种类型的变量当做另一种类型的变量来处理。对一个数值做类型转换的语法如下所示（将某一种类型数值转换为newType类型）：

```
var = (newType) expression;
```

有关类型转换的细节，可参考附录D.5.11。

了解了类型转换、sizeof以及错误检查的必要性之后，我们将本节前面的代码重新修改如下：

```
int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

/* A more correctly written call malloc */
planes = (Flight *) malloc(sizeof(Flight) * airbornePlanes);
if (planes == NULL) {
   printf("Error in allocating the planes array\n");
   :
   :
}
plane[0].altitude = ...
```

其中，由于malloc所分配的空间在地址上是连续的，所以指针方式或数组方式都是通用的。例如，在如上代码中，我们可以使用planes[29]的表达方式，访问第30个结构体（当然，前提是airbornePlanes必须大于30）。这是C语言的一个重要特性，即指针和数组可以交替互换。也因为这种灵活性，C语言成为最受欢迎的编程语言。其他由C而派生的语言（如C++），也都继承了这个特性。

malloc只是众多动态内存分配函数之一。其他函数，如calloc，不仅分配内存空间，同时还将该空间内容初始化为0；再如，realloc能够尝试着去扩大或缩小已分配内存空间的大小。如果要使用这类内存分配函数，程序中必须包含头文件“stdlib.h”。你能否使用realloc来创建随数据大小而改变大小的数组？例如，写一个函数AddPlane()，当已有planes数组的空间太小时，调用该函数增加planes的空间；反之，让函数DeletePlane()缩减当前planes的空间。

与内存分配函数对应的是内存释放函数。该函数的名称是free，参数是指针，指向被malloc（或calloc、realloc）分配的内存块。一个内存区域一旦被释放，可再次被分配。为什么一定需要释放函数呢？答案是，程序运行时，总有一些数据结构在不断地增长、缩减。对于缩减操作，我们需要将它们的内存空间归还到堆空间，以备以后的分配使用。

19.5 链表

在讨论了结构体和动态内存分配概念之后，我们将介绍计算机中最重要的（也是无处不在的）一个数据结构——*链表*（linked list）。链表和数组相似，都用于存储那些表示为元素数列的数据。不同之处是，在数组中，一个元素(除最后一个)与其下一个元素，在内存空间中也是连续的；而在链表中，每个元素也有其“下一个”，但它们之间不要求在内存空间上是连续的。

我们称链表中的每个元素为“*节点*”（node）。每个节点是一个“单位”（unit）数据，如上一节中，每个航班的所有属性信息的集合就是一个数据“单位”。在链表中，节点之间通过指针互相连接。每个节点中，包含一个指向下一节点的指针。换句话说，给定一个起始节点，我们就可以通过指针，从一个节点“跳”到下一个节点，直至遍历整个链表。如果要创建这些节点，则要用到C结构体。在这个描述节点的结构体中，关键的一个元素就是指向下一节点的指针。在如下代码中，我们将介绍一下链表的使用方法。其中，节点结构体借助于原有的Flight结构体。注意，我们

对它做了一个修改，即在结构体中添加了一个新成员，一个指向同类型结构体的指针。

```
typedef struct flightType Flight;
struct flightType {
  char flightNum[7];  /* Max 6 characters        */
  int altitude;       /* in meters               */
  int longitude;      /* in tenths of degrees    */
  int latitude;       /* in tenths of degrees    */
  int heading;        /* in tenths of degrees    */
  double airSpeed;    /* in kilometers/hour      */
  Flight *next;
};
```

与数组类似，链表也是有头有尾。链表的头（head）节点，由一个“头指针”（head pointer）来指向它。而链表的尾（tail）节点，其指针则指向NULL。图19-4所示是链表的两种表示方法。一个是链表的逻辑表示法，每个节点表示为一个小方块，指针则表示为箭头；另一个则是链表的物理表示法，即链表在内存中的数据结构。

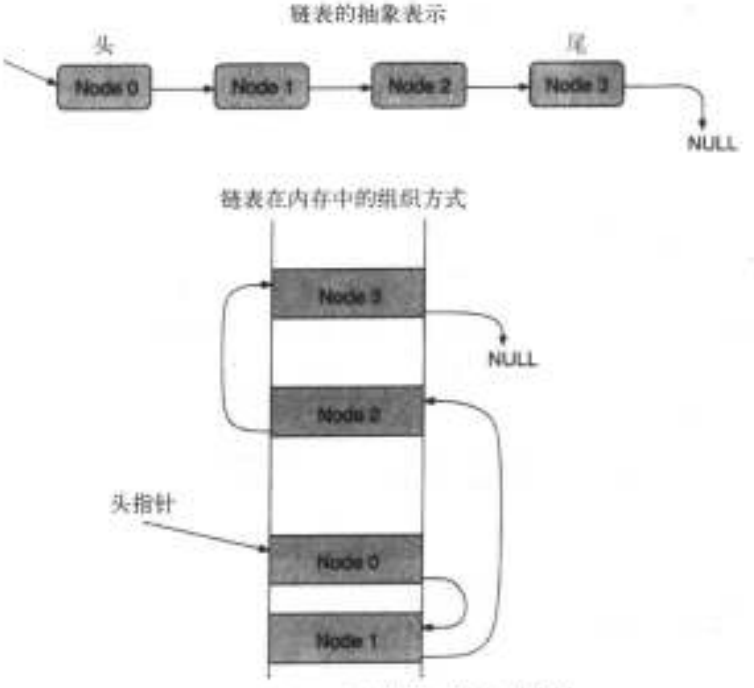


图19-4 链表的两种表示方法

尽管链表和数组存在太多的相似，但它们之间存在本质上的差异。数组的访问顺序是“随机的”（random）。例如，我们可以先访问数组的第4个元素，然后是第911个，然后又是第45个。相比之下，链表的每次访问都必须从“头”开始。例如，要访问第29个节点，则必须先从第0个节点（头节点）开始，然后是第1个节点、第2个节点……最后到达第29个节点。但是，链表具有较好的动态特性，如在链表中，添加或删除节点时无需移动其他节点。而在数组中，每添加或删除一个元素，都需要更多的操作（特别是被操作元素位于数组中间时）。例如，在19.3节的航班管理程序中，当一个飞机着陆后，程序可以删除该飞机的信息，应该怎么做呢？如果是在链表方式中，我们可以将不需要的信息直接删除（即链表的对应节点），空间的分配（为刚起飞的飞机）和回收（已着陆的飞机）都很方便。

示例

假设我们要设计一个“汽车交易库存销售系统”软件。在交易市场，车子不停地进或出，数据库信息也因此而不断地刷新——新车进来时，添加新数据项；车子卖出时，删除该数据项。此外，

这些数据项应该按车牌号顺序存储，以方便销售人员（为了买家）快速查询车辆信息。在数据库中，我们为每辆车定义了如下的记录信息：

```
int  vehicleID;   /* Unique identifier for a car */
char make[20];    /* Manufacturer                */
char model[20];   /* Model name                  */
int  year;        /* Year of manufacture         */
int mileage;      /* in miles                    */
double cost;      /* in dollars                  */

Car *next;        /* Points to a car_node        */
```

为简化起见，我们将车牌号定义为int类型（实际的牌号应该是字符和数字的组合，而不是整数）。

对管理程序而言，最常见的操作就是数据项的添加、删除和查找。如果采用“链表”这种数据结构，这些操作应该非常简单、快捷。链表中，每个节点代表交易场的一辆汽车。为此，我们定义了这样一种节点结构体，并使用typedef将其命名为CarNode：

```
typedef struct carType Car;

struct carType {
  int  vehicleID;   /* Unique identifier for a car */
  char make[20];    /* Manufacturer                */
  char model[20];   /* Model name                  */
  int  year;        /* Year of manufacture         */
  int mileage;      /* in miles                    */
  double cost;      /* in dollars                  */

  Car *next;        /* Points to a car_node        */
};
```

其中，指针next指向一个同类型的结构体对象，即在程序中，它代表链表的下一个节点。如果某节点的next值等于NULL，则意味着该节点为链表的结尾。

```
 1  int main()
 2  {
 3    int op = 0;  /* Current operation to be performed.      */
 4    Car carBase; /* carBase an empty head node              */
 5
 6    carBase.next = NULL;   /* Initialize the list to empty  */
 7
 8    printf("=========================\n");
 9    printf("=== Used car database ===\n");
10    printf("=========================\n\n");
11
12    while (op != 4) {
13      printf("Enter an operation:\n");
14      printf("1 - Car aquired. Add a new entry for it.\n");
15      printf("2 - Car sold. Remove its entry.\n");
16      printf("3 - Query. Look up a car's information.\n");
17      printf("4 - Quit.\n");
18      scanf("%d", &op);
19
20      if (op == 1)
21           AddEntry(&carBase);
22      else if (op == 2)
23           DeleteEntry(&carBase);
24      else if (op == 3)
25           Search(&carBase);
26      else if (op == 4)
27           printf("Goodbye.\n\n");
28      else
29           printf("Invalid option. Try again.\n\n");
30    }
31  }
```

图19-5 二手车数据库程序的main函数

在确定了数据类型的定义及其内存组织方式之后，我们开始程序执行流程的设计。图19-5所

示是主程序main函数的代码。

在图19-5的代码中，我们设计了一个“菜单”（menu）类型的数据库使用接口。在程序中，主要的数据结构体都包含在变量carBase中，它的类型是CarNode，我们称之为“哑”（dummy）头节点。因为，carBase节点中不包含任何数据信息，它的作用只是为了挂接链表。采用哑节点的好处是方便了链表的添加、删除算法，因为不需要考虑链表为空的情况。初始条件下，carBase.next设置为NULL，代表数据库（即链表）中不包含任何数据。值得注意的是，在调用链表的添加（AddEntry）、删除（DeleteEntry）、查找（Search）等函数时，传递的参数都是carBase的地址。

我们还将发现，函数AddEntry、DeleteEntry和Search都要做一个操作：必须扫描链表，才能定位节点。例如，在添加节点操作中，我们必须知道插在什么位置。由于链表是有序排列的（按车牌ID），所以新节点的位置是：保证所有ID值大于该节点ID的节点都在该节点之后。为查找该位置，我们编写了ScanList函数。该函数的任务是从链表头“headPointer”（参数1）开始扫描各节点，查找ID与“vehicleID”（参数2）匹配的节点。如果找到ID匹配节点，则ScanList函数返回指向该节点之前一个节点的指针；如果不存在ID匹配节点，则返回指向最后一个不大于“searchID”节点的指针。顺便问一下，为什么是指向前一个节点呢？答案是，只有这样，随后的添加或删除操作才更方便，参见图19-6所示代码。

```
1  Car *ScanList(Car *headPointer, int searchID)
2  {
3    Car *previous;
4    Car *current;
5
6    /* Point to start of list */
7    previous = headPointer;
8    current = headPointer->next;
9
10   /* Traverse list -- scan until we find a node with a    */
11   /* vehicleID greater than or equal to searchID          */
12   while ((current != NULL) &&
13          (current->vehicleID < searchID)) {
14     previous = current;
15     current  = current->next;
16   }
17
18   /* The variable previous points to node prior to the    */
19   /* node being searched for. Either current->vehicleID   */
20   /* equals searchID or the node does not exist.          */
21   return previous;
22 }
```

图19-6 查找ID匹配节点的函数

下面，我们开始编写节点添加函数AddEntry。该函数的任务是，读取用户输入的新车信息，并在链表中添加一个包含这些信息的节点。如图19-7代码所示，在AddEntry函数中，首先调用malloc函数，在堆区段分配一个CarNode大小的空间（如果分配失败，程序将打印出错信息，并调用exit函数结束程序的执行）。然后，提示用户输入新车信息，并将这些信息存储在CarNode成员变量中。随后，是添加操作。通过ScanList函数查找新节点的插入位置（如果该节点已经存在于链表中，则打印出错信息，并调用free函数释放新节点的空间）。

图19-8所示是将节点插入链表的详细示意图。一旦通过ScanList查找到合适的插入点（prevNode），则将prevNode的next修改为指向新节点（newNode）；同时，将新节点的next指向下一个节点（nextNode）。图中还给出了链表为空时，节点插入的情况。此时，由于prevNode指向的是空的头节点，所以只需将头节点的next指向新节点（newNode）即可。

```
1  void AddEntry(Car *headPointer)
2  {
3    Car *newNode;       /* Points to the new car info       */
4    Car *nextNode;      /* Points to car to follow new one  */
5    Car *prevNode;      /* Points to car before this one    */
6
7    /* Dynamically allocate memory for this new entry.       */
8    newNode = (Car *) malloc(sizeof(Car));
9
10   if (newNode == NULL) {
11      printf("Error: could not allocate a new node\n");
12      exit(1);
13   }
14
15   printf("Enter the following info about the car.\n");
16   printf("Separate each field by white space:\n");
17   printf("vehicle_id make model year mileage cost\n");
18
19   scanf("%d %s %s %d %d %lf",
20         &newNode->vehicleID, newNode->make, newNode->model,
21         &newNode->year, &newNode->mileage, &newNode->cost);
22
23   prevNode = ScanList(headPointer, newNode->vehicleID);
24   nextNode = prevNode->next;
25
26   if ((nextNode == NULL) ||
27       (nextNode->vehicleID != newNode->vehicleID)) {
28     prevNode->next = newNode;
29     newNode->next = nextNode;
30     printf("Entry added.\n\n");
31   }
32   else {
33     printf("That car already exists in the database!\n");
34     printf("Entry not added.\n\n");
35     free(newNode);
36   }
37 }
```

图19-7 数据库添加数据项（新节点）的函数

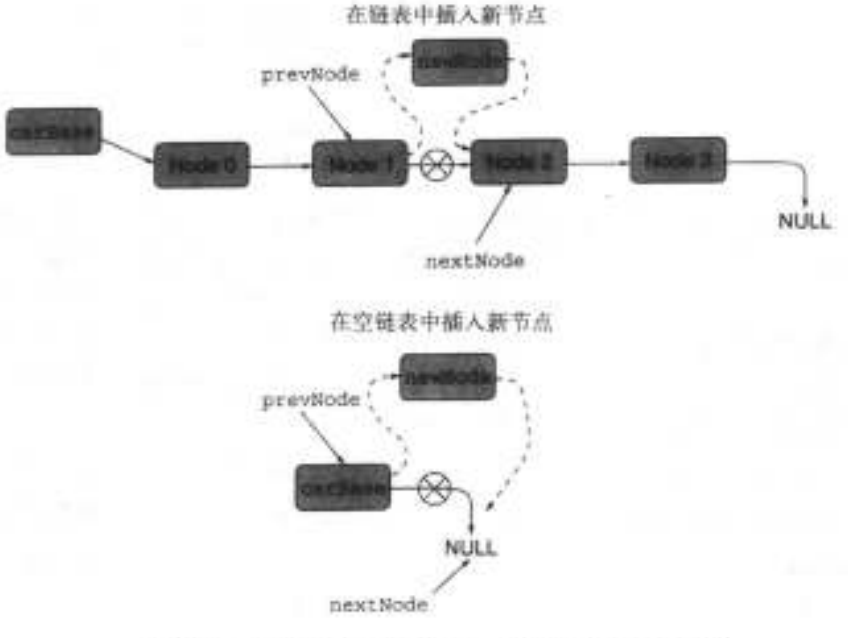


图19-8 在链表中添加新节点。虚线表示新生成的链

删除链表节点的过程，与AddEntry非常相似。首先，用户输入要删除节点的车牌ID，然后，通过ScanList定位节点位置，如果发现该ID节点，则删除该节点。图19-9所示是删除操作的代码。值得注意的是，一旦节点被删除，就能调用free函数将空间归还给堆区段。图19-10所示是删除节点的示意图。

```
1  void DeleteEntry(Car *headPointer)
2  {
3    int vehicleID;
4    Car *delNode;       /* Points to node to delete          */
5    Car *prevNode;      /* Points to node prior to delNode   */
6
7    printf("Enter the vehicle ID of the car to delete:\n");
8    scanf("%d", &vehicleID);
9
10   prevNode = ScanList(headPointer, vehicleID);
11   delNode  = prevNode->next;
12
13   /* Either the car does not exist or                       */
14   /* delNode points to the car to be deleted.               */
15   if (delNode != NULL && delNode->vehicleID == vehicleID) {
16     prevNode->next = delNode->next;
17     printf("Vehicle with ID %d deleted.\n\n", vehicleID);
18     free(delNode);
19   }
20   else
21     printf("The vehicle was not found in the database\n");
22 }
```

图19-9 删除节点的函数

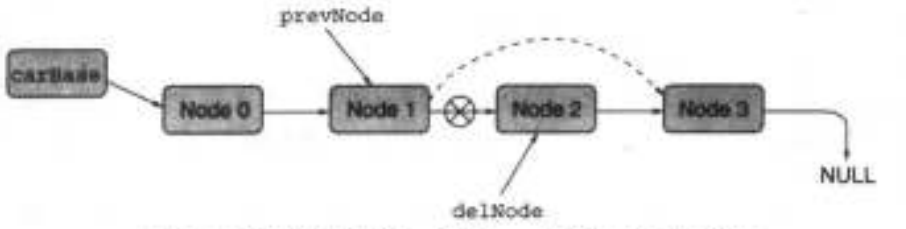


图19-10 从链表中删除一个节点，虚线表示新生成的链

在此，我们将链表和数组的添加、删除方法做个有趣的比较。在链表中，一旦找到节点位置，添加或删除节点的操作仅限于修改几个指针即可。而在数组方式下，删除某个元素，意味着必须将之后的所有元素全部前移一步，以维护数组的顺序性和连续性。如果数组长度非常大，则数据移动的开销是惊人的。结论是，从添加和删除操作来看，链表方式优于数组方式。

最后，我们编写数据库的查找函数Search，它和AddEntry、DelEntry非常相似，只是它不需要修改链表。如图19-11所示，该函数通过ScanList函数，定位被查询节点。

```
1  void Search(CarNode *headPointer)
2  {
3    int vehicleID;
4    CarNode *searchNode;    /* Points to node to delete to follow */
5    CarNode *prevNode;      /* Points to car before one to delete */
6
7    printf("Enter the vehicle ID number of the car to search for:\n");
8    scanf("%d", &vehicleID);
9
10   prevNode = ScanList(headPointer, vehicleID);
11   searchNode  = prevNode->next;
```

图19-11 查询函数

```
11    searchNode  = prevNode->next;
12
13    /* Either the car does not exist in the list or                */
14    /* searchNode points to the car we are looking for.            */
15    if (searchNode != NULL && searchNode->vehicleID == vehicleID) {
16      printf("vehicle ID : %d\n", searchNode->vehicleID);
17      printf("make       : %s\n", searchNode->make);
18      printf("model      : %s\n", searchNode->model);
19      printf("year       : %d\n", searchNode->year);
20      printf("mileage    : %d\n", searchNode->mileage);
21
22      /* The following printf has a field width specification on  */
23      /* %f specification.  The 10.2 indicates that the floating  */
24      /* point number should be printed in a 10 character field   */
25      /* with two units after the decimal displayed.              */
26      printf("cost       : $%10.2f\n\n", searchNode->cost);
27    }
28    else {
29      printf("The vehicle ID %d was not found in the database.\n\n",
30             vehicleID);
31    }
32  }
```

图19-11 查询函数（续）

19.6 小结

本章内容可以总结为以下三个重要概念：

- C结构体。本章最重要的目标，就是介绍“结构体”（structure）的概念。它为程序员创建新的数据类型提供了一种机制。在C语言中，程序员通过“结构体”，可以将多种类型元素组合成一个新类型。“结构体”体现了面向对象的编程思想，即将编程重心围绕在真实世界的对象之上，而不是计算机硬件所支持的几种基本类型。
- 动态内存分配。动态内存分配的概念，是高级C编程的一个基础。在程序执行过程中，我们需要一种内存分配机制，以支持动态数据对象对内存增长和缩减的需求。为此，C提供了如malloc、calloc、realloc和free等内存分配和回收函数。
- 链表。将结构体和动态内存分配这两个概念结合，我们引入了计算机系统中最重要的一个数据结构——链表（linked list）。它与数组的相似之处是，都适用于“数列”（list）的组织管理。链表结构之所以这么重要，是因为它的动态特性。链表大小可以在执行过程中，动态增长和减小。这种特性是数组类型所不具备的（在一些情况下，数组存在空间浪费的问题）。链表的关键是，数据项之间通过指针相连。以后在更高级的数据结构，如哈希表（hash table）、树（tree）或图（graph）中，你还将看到链表的身影。

19.7 习题

19.1 试问，以下程序是否存在bug？请解释。

```
struct node {
  int count;
  struct node *next;
};

int main()
{
  int data = 0;
  struct node *getdata;

  getdata->count = data + 1;
  printf("%d", getdata->count);
}
```

19.2 如下是一个C程序的部分代码：

```
struct node {
  int count;
  struct node *next;
};

main()
{
  int data = 0;
  struct node *getdata;

        :
        :

  getdata = getdata->next;

        :
        :
}
```

请写出编译器为其中的“*getdata = getdata->next;*”语句所生成的LC-3汇编代码。

19.3 如图19-2所示，在PotentialCollisions函数中，有一段配对检查代码，判断某架飞机与其他所有飞机之间是否存在相撞的可能。如果将这段代码稍做修改，即可大大提高程序执行的效率。请问如何修改？

19.4 如果在一个计算机系统中，每一种基本数据类型（指针、字符、整数、浮点数）都只占用一个内存单元。试回答以下编译问题。

```
struct element {
  char  name[25];
  int   atomic_number;
  float atomic_mass;
};

is_it_noble(struct element t[], int i)
{
  if ((t[i].atomic_number==2)  ||
      (t[i].atomic_number==10) ||
      (t[i].atomic_number==18) ||
      (t[i].atomic_number==36) ||
      (t[i].atomic_number==54) ||
      (t[i].atomic_number==86))
    return 1;
  else
    return 0;
}

int main()
{
  int x, y;
  struct element periodic_table[110];

    :
    :
  x = is_it_noble(periodic_table, y);
    :
    :
}
```

a. 函数is_it_noble的活动记录，要占用多少内存单元？

b. 在main函数中，如果x、y、periodic_table是仅有的局部变量，请问main的活动记录中，有多少位置是用于这些局部变量的？

19.5 假设，如下C代码被编译为LC-3机器语言。LC-3运行时栈的起始地址是xEFFF。程序运行时，用户键入“abac”及回车。请回答以下问题：

```
#include <stdio.h>
#define MAX 4

struct char_rec {
```

```
    char ch;
    struct char_rec *back;
};

int main()
{
    struct char_rec *ptr, pat[MAX+2];
    int i = 1, j = 1;

    printf("Pattern: ");
    pat[1].back = pat;
    ptr = pat;

    while ((pat[i].ch = getchar()) != '\n') {

      ptr[++i].back = ++ptr;

      if (i > MAX) break;
    }

    while (j <= i)
      printf("%d ", pat[j++].back - pat);

    /* Note the pointer arithmetic here: subtraction
       of pointers to structures gives the number of
       structures between addresses, not the number
       of memory locations */
}
```

a. 执行结束时，main函数栈空间的内容是什么？

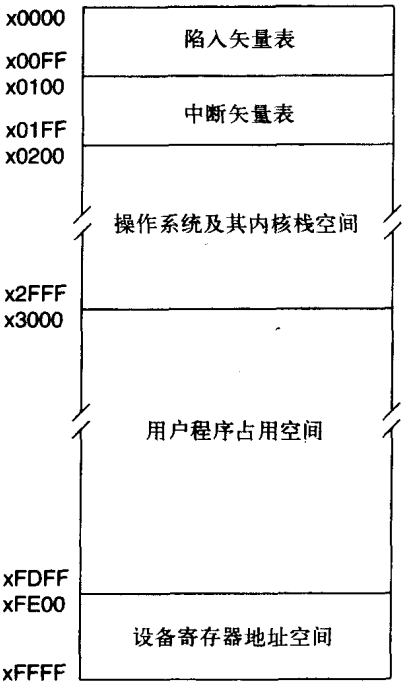
b. 如果输入为“abac”，程序的输出结果如何？

附录A　LC-3指令集结构

A.1　概述

LC-3的指令集结构（Instruction Set Architecture, ISA）定义如下：

- 内存地址空间（Memory Address Space）：16位地址，对应2^{16}个内存单元，每单元包含一个字（word），每字宽度是16位。地址的编号从0（x0000）至65535（xFFFF）。地址是用以识别每个内存单元和内存映射（memory-mapped）的I/O设备寄存器。内存中的部分区段有着特殊用途，如图A-1所示。
- 位编号（Bit Numbering）：量化数（quantity）的每个位也是有编号的。通常，自右向左，最右边位的编号为0（即bit 0），最左边的位编号是15（即bit 15）。
- 指令（Instruction）：每个指令的宽度是16位。其中，bit[15:12]是操作码（operation，即要执行的操作）；bit[11:0]提供指令执行时所需要的信息，具体定义参考A.3节。
- 非法操作码异常（Illegal Opcode Exception）：操作码bit[15:12]=1101没有被定义，即如果一个指令的bit[15:12]的内容是1101，则产生“非法操作码异常”。A.4节将解释这个情况。
- 程序计数器（Program Counter）：16-bit宽度的一个寄存器，其内容是下一条执行指令所在的地址。



图A-1　LC-3内存映像图

- 通用寄存器（General Purpose register）：宽度为16-bit的8个寄存器，编号分别为000~111。
- 条件码（Condition Codes）：1-bit宽度的3个寄存器，分别是N（negative）、Z（zero）、P（positive）。load指令（LD、LDI、LDR、LEA）和运算指令（ADD、AND、NOT）在向任意一个通用寄存器写入数值时，都将改变条件码。按照16位补码方式看待这个数值，它有3种可能，即负数（N=1；Z=0；P=0）、零（Z=1；N=0；P=0）和正数（P=1；N=0；Z=0）。除load和运算指令外，其他LC-3指令不会改变条件码。
- 内存映射I/O（Memory-mapped I/O）：由于设备的输入/输出（即I/O）操作是由load/store指令（LDI/STI、LDR/STR）完成的，因而我们为每个I/O设备寄存器分配了内存地址（xFE00~xFFFF）。图A-1和表A-3列举了LC-3的设备寄存器及其分配的内存地址。
- 中断处理（Interrupt Processing）：I/O设备具备中断处理器的能力。A.4节将详细描述该机制。
- 优先级（Priority Level）：LC-3定义了8个优先级别。PL7（优先级7）最高，PL0最低。PSR[10:8]代表当前执行进程的优先级别。
- 处理器状态寄存器（Processor Status Register, PSR）：16-bit寄存器，包含了当前执行进程的

状态信息。其中，PSR的8个位已做了定义，PSR[15]定义了执行进程的权限模式，PSR[10:8]定义了当前执行进程的优先级别，PSR[2:0]包含的是条件码（PSR[2]=N、PSR[1]=Z、PSR[0]=P）。

- 权限模式（Privilege Mode）：LC-3定义了两种权限模式，即特权模式和用户模式。如中断服务程序的执行状态就是“特权模式”。权限模式由PSR[15]标识，PSR[15]=0代表特权模式，PSR[15]=1代表用户模式。
- 权限模式异常（Privilege Mode Exception）：如果RTI指令执行在特权模式下，但你试图在用户模式下执行RTI指令，则将产生“权限模式异常”。A.4节将对其做详细解释。
- 特权模式栈空间（Supervisor Stack）：又称内核栈空间。在特权模式下、通过SSP指针（Supervisor Stack Pointer）访问该内存区域。事实上，特权模式下（PSR[15]=0），栈指针（R6）代表的就是SSP。
- 用户模式栈空间（User Stack）：在用户模式下，该空间是由USP指针（User Stack Pointer）访问的。当PSR[15]=1时（用户模式），栈指针（R6）就是USP。

A.2 注释

表A-1（见下一页）中的注释，将有助于你更好地理解对LC-3指令的描述（A.3节）。

A.3 指令集

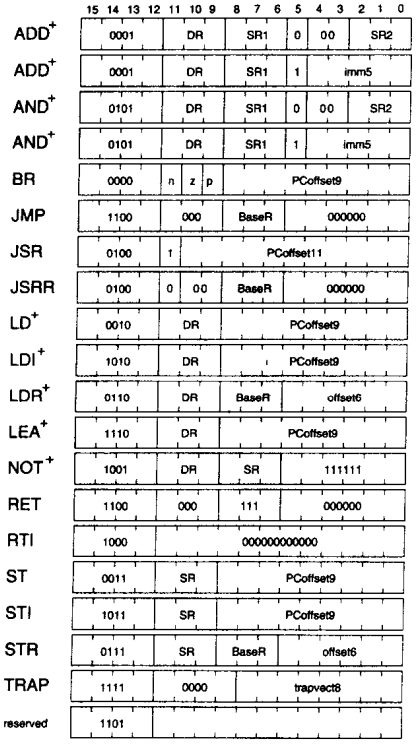
LC-3定义了一组丰富而简洁的指令集。每条指令（16-bit）包含4-bit的操作码（bit[15:12]）以及12位的操作相关信息。图A-2汇总了LC-3的15种操作码，以及其余信息位的使用说明。第16个4-bit操作码未做定义。在下面的篇幅中，将详细描述每条指令的汇编语言表示、16-bit指令格式、指令操作含义及应用实例。

表A-1 指令注释规范（Notational Conventions）

注 释	含 义
xNumber	数值的十六进制（hexadecimal）表示
#Number	数值的十进制（decimal）表示
A[1:r]	数据A的字段取值，左界为bit[l]，右界为bit[r]。例如，PC=0011001100111111，则PC[15:9]=0011001，PC[2:2]=1。如果l和r相同，则可以简写为PC[2]
BaseR	基址寄存器（Base Register）。通常基址寄存器和1个6-bit偏移相加，产生Base+offset地址
DR	目的寄存器（Destination Register）。代表R0~R7之一，指令中用于存放结果的寄存器
imm5	5-bit立即数（immediate value）。当用做立即数时，通常对应指令中的bit[4:0]。补码（2's complement）表示，范围-16~15，使用之前要做16位符号扩展
LABEL	汇编语言的一个构造元素，代表一个地址单元（符号表示，而不是16-bit地址表示）
mem[address]	代表给定地址的内存单元的内容
offset6	6-bit数值，指令的bit[5:0]，用于Base+offset寻址模式。6-bit有符号补码（范围-32~31），所以在计算Base+offset时，要做16位符号扩展
PC	程序计数器（Program Counter）。16位寄存器，指向下一个待获取指令的地址。例如，如果当前指令所在地址是A，则PC的内容是A+1

（续）

注　释	含　义
PCoffset9	9-bit数值，指令的bit[8:0]；用于PC+offset寻址模式。bit[8:0]被看做是一个9-bit的有符号补码（范围是-256~255），符号扩展至16-bit之后，与增量PC相加组成地址
PCoffset11	11-bit数值，指令的bit[10:0]；用于JSR指令计算子程序的入口地址。bit[10:0]被看做是一个11-bit的有符号补码（范围是-1024~1023），符号扩展至16-bit之后，与增量PC相加组成地址
PSR	处理器状态寄存器。16-bit寄存器，包含了当前正在运行进程（process）的状态信息。PSR[15]=权限模式，PSR[2:0]包含的是状态码（PSR[2]=N, PSR[1]=Z, PSR[0]=P）
setcc()	设置条件码N、Z、P（基于写入DR的数值）。如果值为负，则N = 1，Z = 0，P = 0；如果值为0，则N = 0，Z = 1，P = 0；如果值为正，则N = 0，Z = 0，P = 1
SEXT(A)	对A的符号扩展。A的最高位被复制填充前缀位，直至补齐了16位。例如，A=11 0000，则SEXT(A)=1111 1111 1111 0000
SP	当前栈指针。R6就是当前栈指针。存在两个栈，分别对应两种权限模式（特权、用户）。如果PSR[15]=1，SP=USP；如果PSR[15]=0，SP=SSP
SR,SR1,SR2	源寄存器。R0~R7
SSP	特权模式栈指针（Supervisor Stack Pointer）
trapvect8	8-bit值，指令的bit[7:0]。用于TRAP指令确定trap服务程序的入口地址。bit[7:0]被看做是无符号整数，零扩展至16位地址值（范围0~255）。该地址内存单元中存放了TRAP服务程序的入口地址（即间接寻址方式）
USP	用户模式栈指针（User Stack Pointer）
ZEXT(A)	对A的零扩展。A的最左边被填入0，直至16位。例如，如果A=11 0000，则ZEXT(A)=0000 0000 0011 0000



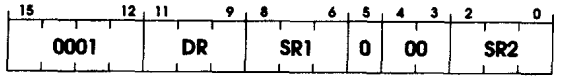
图A-2　LC-3指令集全部指令的格式（注：+表示该指令将修改条件码）

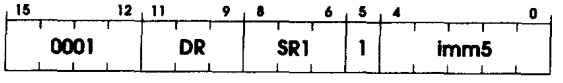
ADD (Addition)

汇编器格式

```
ADD   DR,SR1,SR2
ADD   DR,SR1,imm5
```

编码





操作

```
if (bit[5] == 0)
   DR = SR1 + SR2;
else
   DR = SR1 + SEXT(imm5);
setcc();
```

描述

如果bit[5]是0，则第2个操作数来自SR2；如果bit[5]是1，则第2个操作数来自imm5字段的16位符号扩展值。无论第2个操作数来自哪里，它都将与SR1相加，并将结果存入DR。同时，根据这个数值的结果，设置对应的条件码（N、Z、P）。

例子

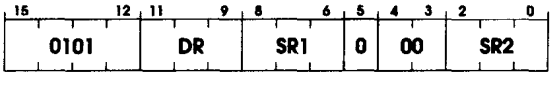
```
ADD   R2,R3,R4          ;R2<-R3+R4
ADD   R2,R3,#7          ;R2<-R3+7
```

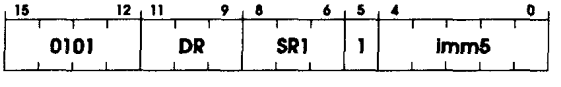
AND (Bit-wise logical AND)

汇编器格式

```
AND   DR,SR1,SR2
AND   DR,SR1,imm5
```

编码





操作

```
if (bit[5] == 0)
   DR=SR1 AND SR2;
else
   DR=SR1 AND SEXT(imm5);
setcc();
```

描述

如果bit[5]是0，则第2个操作数来自SR2；如果bit[5]是1，则第2个操作数来自imm5字段的16位符号扩展值。无论第2个操作数来自哪里，它都将与SR1做“按位与”（bit-wise AND）运算，并将结果存入DR。同时，根据这个数值的结果，设置对应的条件码（N、Z、P）。

例子

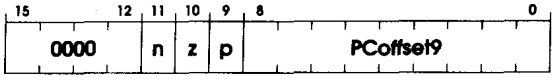
```
AND   R2,R3,R4          ;R2<-R3 AND R4
AND   R2,R3,#7          ;R2<-R3 AND 7
```

BR (Conditional Branch)

汇编器格式

```
BRn   LABEL       BRzp   LABEL
BRz   LABEL       BRnp   LABEL
BRp   LABEL       BRnz   LABEL
BR    LABEL       BRnzp   LABEL
```

编码



操作

```
if ((n AND N) OR (z AND Z) OR (p AND P))
PC = PC⊖ + SEXT(PCoffset9);
```

描述

由bit[11:9]指定的条件码被测试。即如果bit[11]=1，则N被测试；如果bit[11]=0，则N不被测试。Z和P同理。如果任何一个被指定测试的条件码被置位，则程序跳转至PCoffset9字段的符号扩展和增量PC之和的地址处。

例子

```
BRzp   LOOP        ; 如果最后的结果是零 (zero) 或正数 (positive)，跳转至LOOP
BR⊖    NEXT        ; 无条件跳转至NEXT
```

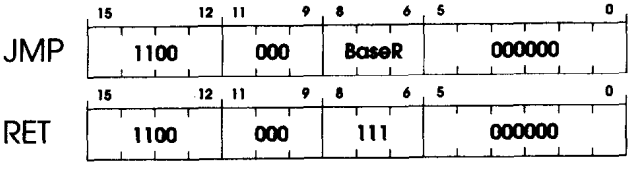
JMP (Jump)

RET

汇编器格式

```
JMP   BaseR
RET
```

编码



操作

```
PC = BaseR
```

描述

程序无条件地跳转至由基址寄存器的内容指定的入口。bit[8:6]代表该基址寄存器的编号。

例子

⊖ 增量PC。

⊖ 汇编语言的BR和BRnzp是完全相同的，即无论哪个条件位满足，都跳转到目标地址（无条件跳转）。

```
JMP    R2          ;PC←R2
RET                ;PC←R7
```

备注

RET指令可以看做是JMP指令的一个特例。PC被装入R7的内容，R7扮演的是链接（linkage）的作用，其内容是指向子程序调用指令之后的指令地址。

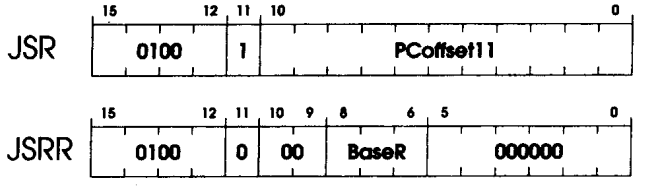
JSR (Jump to Subroutine)

JSRR

汇编器格式

```
JSR    LABEL
JSRR   BaseR
```

编码



操作

```
R7 = PC⊖;
if (bit[11] == 0)
   PC = BaseR;
else
   PC = PC + SEXT(PCoffset11);
```

描述

首先，增量PC值被保存在R7中，这是返回调用代码的链接（linkage）地址；然后PC装入被调用子程序的入口地址，即等价于无条件跳转至该地址。子程序的入口地址来自基址寄存器（如果bit[11]=0），或是bit[10:0]符号扩展值和增量PC之和（如果bit[11]=1）。

例子

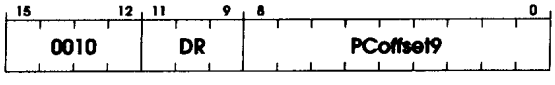
```
JSR    QUEUE       ；将JSR指令之后的地址装入R7，然后跳转至QUEUE
JSRR   R3          ；将JSRR指令之后的地址装入R7，然后跳转至R3内容指定的地址
```

LD (Load)

汇编器格式

```
LD    DR, LABEL
```

编码



操作

```
DR = mem(PC⊖ + SEXT(PCoffset9));
```

⊖ 增量（incremented）PC。

⊖ 增量PC。

```
setcc();
```

描述

由bit[8:0]的16位符号扩展值和增量PC相加，计算出一个地址，然后将该地址的内存单元内容装入DR寄存器。同时，根据装入数值的内容，设置相应的条件码。

例子

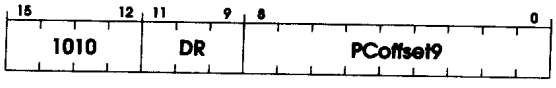
```
LD   R4, VALUE              ; R4←mem[VALUE]
```

LDI (Load indirect)

汇编器格式

```
LDI   DR, LABEL
```

编码



操作

```
DR = mem[mem[PC⊖ + SEXT(PC offset9)]];
setcc();
```

描述

由bit[8:0]的16位符号扩展值和增量PC相加，计算出一个地址，然后再将该地址的内存单元内容作为地址，再次读取内存，并将获取的内容装入DR寄存器。同时，根据装入数值的内容，设置相应的条件码。

例子

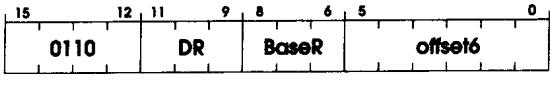
```
LDI   R4, ONEMORE           ; R4<-mem[mem[ONEMORE]]
```

LDR (Load Base+offset)

汇编器格式

```
LDR   DR, BaseR, offset6
```

编码



操作

```
DR = mem[BaseR + SEXT(offset6)];
setcc();
```

描述

由bit[5:0]的16位符号扩展值和bit[8:6]指定寄存器的内容相加，计算出一个地址，然后将该地址的内存单元内容装入DR寄存器。同时，根据装入数值的内容，设置相应的条件码。

例子

```
LDR   R4, R2,#-5            ; R4<-mem[R2-5]
```

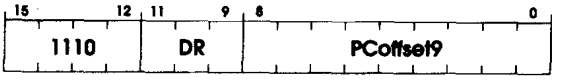
⊖ 增量PC。

LEA (Load Effective Address)

汇编器格式

```
LEA   DR, LABEL
```

编码



操作

```
DR = PC⊖ + SEXT(PCoffset9);
setcc();
```

描述

由bit[8:0]的16位符号扩展值和增量PC相加，计算出一个地址，然后将该地址值装入DR⊜寄存器。同时，根据装入数值的内容，设置相应的条件码。

例子

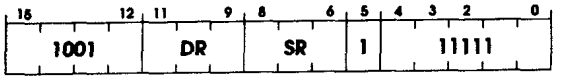
```
LEA   R4, TARTGET        ; R4<-address of TARGET
```

NOT (Bit-wise Complement)

汇编器格式

```
NOT   DR, SR
```

编码



操作

```
DR = NOT(SR);
setcc();
```

描述

将SR的内容按位取补码（bit-wise complement，即反码）存入DR。基于变换出来的数值是否为负数、零、正数，设置相应的条件码。

例子

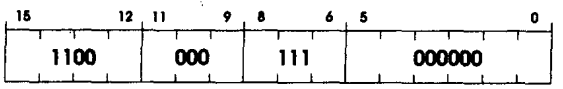
```
NOT   R4, R2      ; R4<-NOT(R2)
```

RET⊜ (Return from Subroutine)

汇编器格式

```
RET
```

编码



⊖ 增量PC。

⊜ LEA指令并不读取内存，只是将计算出来的该内存单元的地址赋值给DR。

⊜ RET指令可以理解为是JMP指令的一个特例，参见JMP。

操作

```
PC = R7;
```

描述

将R7的内容装入PC。由于R7扮演的是linkage的角色（参见JSR/JSRR），所以RET指令产生的效果，就是返回之前JSR调用点的下一条指令。

例子

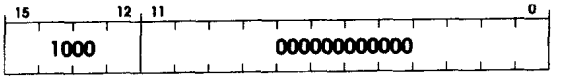
```
RET    ; PC <- R7
```

RTI (Return from Interrupt)

汇编器格式

```
RTI
```

编码



操作

```
if (PSR[15] == 0)
   PC = mem[R6]; R6 is the SSP
   R6 = R6 + 1;
   TEMP = mem[R6];
   R6 = R6 + 1;
   PSR = TEMP;    the privilege mode and condition codes of
   the interrupted process are restored
else
   Initiate a priviledge mode exception;
```

描述

如果当前处理器运行在特权模式，则该指令执行是合法的，特权模式栈（Supervisor Stack）顶部两个单元的内容被弹出（pop），并分别赋值给PC和PSR。如果当前处理器运行在用户模式，则该指令的执行将引发“特权模式冲突”异常。

例子

```
RTI    ; PC,PSR <- top two values popped off stack.
```

备注

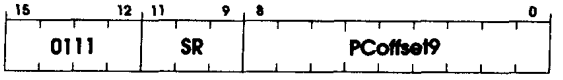
当外部中断（interrupt）或内部异常（exception）发生时，处理器最先做的事，就是将权限模式切换到特权模式（PSR[15]=0）；然后将PSR和PC的内容压入（push）特权模式栈；最后将中断或异常服务程序的入口地址装入PC。中断或异常服务程序一定是运行在特权模式下的，服务程序的最后一条指令一定是RTI（即恢复原PC和PSR内容）。有关PC的内容，在中断情况下，被恢复的（或返回的）PC内容是中断发生时即将执行的指令的地址；而异常情况下，PC的恢复内容可能是引发异常的指令地址（使之重新执行一遍），也可能是其下一条指令的地址（无需重新执行）。有关PSR，异常模式也比较复杂。中断情况的PSR内容恢复为中断发生时的现场值，而异常模式下可能恢复为原现场值，也可能被修改。另外，如果当前处理器运行在用户模式，该指令将引发“权限模式冲突”异常。以上细节的解释参见A.4节的论述。

ST (Store)

汇编器格式

```
ST   SR, LABEL
```

编码



操作

```
mem[PC⊖ + SEXT(PCoffset9)] = SR;
```

描述

由bit[8:0]的16位符号扩展值和增量PC相加，计算出一个地址，然后将SR指定寄存器的内容装入该地址指向的内存单元中。

例子

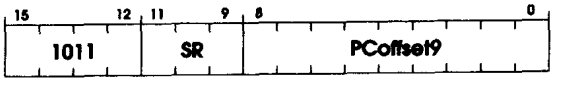
```
ST   R4, HERE              ;mem[HERE]<-R4
```

STI (Store Indirect)

汇编器格式

```
STI   SR, LABEL
```

编码



操作

```
mem[mem[PC⊖ + SEXT(PCoffset9)]] = SR;
```

描述

由bit[8:0]的16位符号扩展值和增量PC相加，计算出一个地址A，然后以该地址A指向的内存单元的内容B作为地址，将SR指定寄存器的内容装入该地址B指向的内存单元中。

例子

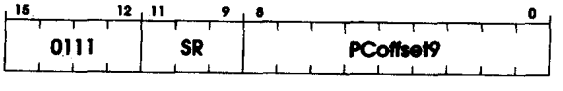
```
STI   R4, NOT_HERE         ; mem[mem[NOT_HERE]]<-R4
```

STR (Store Base+offset)

汇编器格式

```
STR   SR, BaseR, offset6
```

编码



操作

```
mem[BaseR + SEXT(offset6)] = SR;
```

⊖ 增量PC。

⊜ 增量PC。

描述

由bit[5:0]的16位符号扩展值和bit[8:6]指定寄存器的内容相加，计算出一个地址，然后将SR指定寄存器的内容装入该地址的内存单元中。

例子

```
STR   R4, R2, #5  ; mem[R2+5]<-R4
```

TRAP (System Call)

汇编器格式

```
TRAP   trapvector8
```

编码



操作

```
R7 = PC⊖;
PC = mem[ZEXT(trapvect8)];
```

描述

首先将增量PC装入R7（为服务程序返回做准备）。然后将由trapvector8指定的系统调用的入口地址装入PC（等价于跳转）。注意，这个入口地址的计算是间接的，即先将trapvector8零扩展为一个16-bit地址，然后读取该地址单元的内容，作为服务程序的入口地址。

例子

```
TRAP x23   ; Directs the operating system to execute the IN system call
           ; The starting address of this system call is contained in
           ; memory location x0023.
```

备注

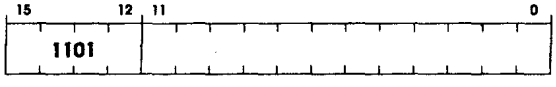
内存单元x0000~x00FF，共256个，用来存放各个陷入矢量（trap vector）所对应的系统服务程序的入口地址，所以该区段又被称为“陷入矢量表”（Trap Vector Table）。表A-2列出了陷入矢量x20~x25所对应的各个服务程序。

Unused Opcode

汇编器格式

```
none.
```

编码



操作

```
Initiate an illegal opcode exception.
```

描述

该操作码所对应的指令未做定义。即意味着如果执行该指令，将引发“非法指令”（illegal

⊖ 增量PC。

opcode）异常。

备注

操作码（1101）是预留以后做定义的。如果当前执行指令的操作码的bit[15:12]=1101，则产生非法指令异常，参见A.4节的描述。

表A-2 Trap服务程序

陷入矢量	汇编器名	描 述
x20	GETC	从键盘读入一个字符。但该字符并不在屏幕上回显（echo）。该字符的ASCII码值被拷贝入R0（R0的高8位被清零）
x21	OUT	将R0[7:0]的字符输出在屏幕上显示
x22	PUTS	向屏幕写一个字符串。所有字符在内存中的存放是连续的，且每个内存单元一个字符。起始地址由R0指定，结束判断由当前内存单元是否为x0000确定
x23	IN	先打印提示（prompt）在屏幕上，然后等待键盘输入一个字符。读入字符回显在屏幕上，读入的ASCII码装入R0，R0高8位清零
x24	PUTSP	向屏幕写一个字符串。所有字符在内存中的存放是连续的，但每个内存单元存放两个字符。输出显示时，先将bit[7:0]输出，然后再将bit[15:8]输出（如果一个字符串的字符个数是奇数个，则最后一个内存单元的内容bit[15:8]为x00）。起始地址由R0指定，结束判断由当前内存单元是否为x0000确定
x25	HALT	停止执行，并在屏幕上输出信息

表A-3 设备寄存器分配

地址	寄 存 器 名	寄存器的功能
xFE00	Keyboard status register	KBSR。Ready位（bit[15]）表示是否已接收到一个新的键盘字符输入
xFE02	Keyboard data register	KBDR。bit[7:0]包含的是最新接收的字符
xFE04	Display status register	DSR。Ready位（bit[15]）表示显示设备目前是否可以接收下一个待显示字符
xFE06	Display data register	DDR。该寄存器低字节（即bit[7:0]）包含的是待显示字符
xFFFE	Machine control register	MCR。bit[15]是时钟使能位，该位清零时，机器停止运行

A.4 中断和异常处理

外部事件（event external）能够中断（interrupt）（而不是halt）当前正在运行的程序。最常见的外部事件就是I/O中断驱动（interrupt-driven I/O）。同样，运行程序本身也可能产生异常事件（exceptional event）中断处理器的运行，如非法指令（即无效操作码）就是一个“内部”事件的例子。

事件（event）处理涉及很多数据结构，中断矢量表（interrupt vector table）就是其一，该表共有256个表项，每个表项对应一种事件（所谓“矢量”就是事件的编号）。中断矢量表的起始地址是x0100，即意味着该表内存占用空间的范围是x0100~x01FF。表项包含了该矢量服务程序的入口地址，所谓“服务程序”，就是由操作系统提供的处理该事件的专用程序，值得提醒的是，服务

程序执行在特权模式下。

该表的前128个表项（x0100~x017F）是程序自身引发事件（即exception）的处理程序入口地址，通常称它们为“异常服务程序”（exception service routine）。之所以称这些事件为异常（exception）而不是中断（interruption），是因为这些事件的发生是“意外的”，会影响程序的正常执行。

该表后半部分的128个表项（x0180~0x01FF）是外部事件的服务程序入口地址，如来自I/O设备的请求事件处理程序，通常称它们为“中断服务程序”（interrupt service routines）。

A.4.1 中断

虽然LC-3提供了能够服务I/O中断的矢量机制，但事实上LC-3计算机系统只提供了惟一一种能产生中断的I/O设备，即键盘（keyboard）。它的中断优先级是PL4，对应的中断矢量是x80。

一个I/O设备如果希望获得服务（即获得“关注”），它将试图中断处理器。如果当前中断使能（Interrupt Enable, IE）位是1，且中断请求的优先级比当前运行程序的优先级高，则中断成功。

下面举例说明中断的处理机制。假设当前程序运行级别小于PL4，且此时有人按动键盘。如果KBSR的IE位为1，则当前指令周期结束之后程序被中断（即程序暂停）。随后，中断服务程序被激活（initiated）：

（1）处理器硬件自动将权限模式设置为特权模式（PSR[15]=0）。

（2）同时，将优先级别设置为PL4，即该设备对应的中断处理优先级（PSR[10:8]=100）。

（3）将特权栈指针（SSP）寄存器的内容装入R6寄存器。

（4）将PSR和PC的内容压入特权栈。

在处理器的硬件自动完成以上操作之后：

（5）键盘设备提供一个8-bit的矢量，即x80。

（6）处理器将该矢量值（x80）扩展为x0180，即该中断矢量表项所在的位置。

（7）将x0180地址内存单元的内容装入PC，即等价于“跳转”入键盘中断服务程序。

在以上硬件操作之后，进入软件运行阶段：

（8）处理器开始了中断服务程序的执行。

中断服务程序的最后一条指令是RTI。当RTI被执行时，硬件又将自动完成以下操作：

（9）特权栈顶部两个单元的内容被弹出，恢复原先PC和PSR寄存器的内容。

（10）根据当前的PSR[15]内容，R6将被装入合适的栈指针值。

在完成以上中断处理之后，原先被中断的程序恢复执行（注意，被中断的程序可能是用户程序，也可能是操作系统）。

A.4.2 异常

LC-3计算机系统只定义了两种异常情况，即“权限模式冲突”和“非法指令”。权限模式冲突的异常情况，是在用户模式下遇到RTI指令；而非法指令的异常情况，是处理器遇到操作码（bit[15:12]）为1101的指令。

对于处理器来说，一旦遇到异常，必须尽快处理！异常服务程序的激活（initiated）方式和中断类似：

（1）处理器切换（设置）权限模式至特权模式（PSR[15]=0）。

（2）R6被装入特权栈指针（SSP）寄存器的内容。

（3）PSR和PC的内容被压入特权栈。

（4）异常处理模块提供它的8-bit的矢量。如果是权限模式冲突，则提供的矢量是x00；如果是

非法指令，提供的矢量是x01。

(5) 处理器将该矢量扩展为x0100或x0101，即对应的中断矢量表项的地址。

(6) 将x0100或x0101地址内存单元的内容装入PC，即异常服务程序的入口地址。

于是，处理器开始异常服务程序的执行。

异常服务程序的处理细节取决于不同的异常类型和操作系统的处理风格。

在一些特定情况下，异常服务程序能够纠正异常问题，并恢复原程序的执行。那么在这种情况下，服务程序的最后一条指令一定是RTI，即将特权栈顶部两个单元的内容弹出，并分别装入PC和PSR寄存器，随后恢复原程序的执行。

但在大多数情况下，异常的出现是灾难性的，即服务程序（操作系统）应该取消原程序的执行。

此外，中断处理和异常处理的区别是服务程序执行的优先级别。中断服务程序运行时，通常将优先级别自动设置为不同设备所对应的级别；而异常处理时，通常我们不改变优先级别。程序原本所具有的优先级别，事实上就代表了该程序本身的“紧迫度”(urgency)，所以在处理它的事件时，也没有必要改变这个“紧迫度”。例如，LC-3 ISA定义的两个异常，即权限模式冲突发生或程序中存在非法指令时，就没有必要改变服务程序运行的优先级。

附录B　从LC-3到x86

正如你所知，LC-3指令集结构（ISA）显式地指定了LC-3机器语言（由程序员编写或由LC-3编译器产生）与LC-3微结构（接受指令并进行处理）之间的接口。指令集的定义范畴包括内存地址空间及其可寻址能力、寄存器数目和大小、指令的格式、操作码、数据编码方式，以及操作数的数据类型及其寻址模式。

同样，在你的PC机上，也存在一个指令集结构，它也指定了（对应于微处理器的）编译器和微结构之间的接口。不同的是，PC机的指令集是x86，而不是LC-3。Intel在1979年设计并实现了该架构（即x86）的第一个成员——8086，它规定了地址和数据的大小都是16位，而今天，地址和数据的标准大小已演变为32位。从8086至今，Intel的系列产品结构一直都遵循着该指令系统集，如80286（1982）、386（1985）、486（1989）、Pentium（1992）、Pentium Pro（1995）、Pentium II（1997）、Pentium III（1999）和Pentium IV（2001）。

相比之下，x86的指令集要比LC-3的指令集复杂得多。X86包含更多的操作码，更多的数据类型，更多的寻址模式，更复杂的内存结构，以及更复杂的0/1编码方式。但本质上，我们说它们（x86和LC-3）具有相同的基本要素。

我们已经花了很多精力去学习、理解与LC-3相关的东西，或许有人会觉得，我们还应该学习一些“真实”的指令集知识。在此要提醒的是，那些在做以及能够做此类事情（即学习研究真实指令集结构）的人，都是诸如Intel公司（大规模生产LC-3类芯片）、Dell公司（基于芯片制造PC机），以及Microsoft公司（为LC-3类指令集结构编译Windows NT操作系统程序）等专业“人士”。当然，还有一种更容易的学习“真实”指令集的方法，即阅读本附录。

下面我们开始介绍x86的有关要素，这是一种非常复杂的指令集。尽管它很复杂，但我们仍然要介绍它，因为它是人们接触最普遍的一种处理器。

当然，我们并不打算介绍完整的x86 ISA规范。因为，要做到这一点，需要整整一本书来描述，同时还涉及到操作系统、编译器和计算机系统在内的各方面知识，这远远超出了目前的知识范畴。如果要详细研究它，我们推荐1997年由Intel公司出版的《Intel架构软件开发者手册》的1、2、3卷。本附录中，我们只讨论与应用程序（而不是操作系统）相关的x86 ISA特性方面的知识。我们的目的是，通过这些内容，让你对x86指令集的丰富性有直观的认识，然后结合我们所熟悉的LC-3指令集，探索、体会这些特性在LC-3的设计和实现中的应用。

B.1　LC-3与x86特性比较

B.1.1　指令集

一个指令集包含了许多不同的指令，每条指令又包含一个操作码和若干个操作数。操作数的具体数目取决于操作码的类型。我们称其中的每个操作数为一个数据元素，并基于数据类型而编码。另外，操作数的来源取决于当前指令指定的寻址模式。

LC-3指令集具备一种数据类型、15个操作码和3种寻址模式，这3种寻址模式是：相对PC寻址（LD/ST）、间接寻址（LDI/STI）和寄存器加偏移寻址（LDR/STR）。而x86指令集则拥有多达12种

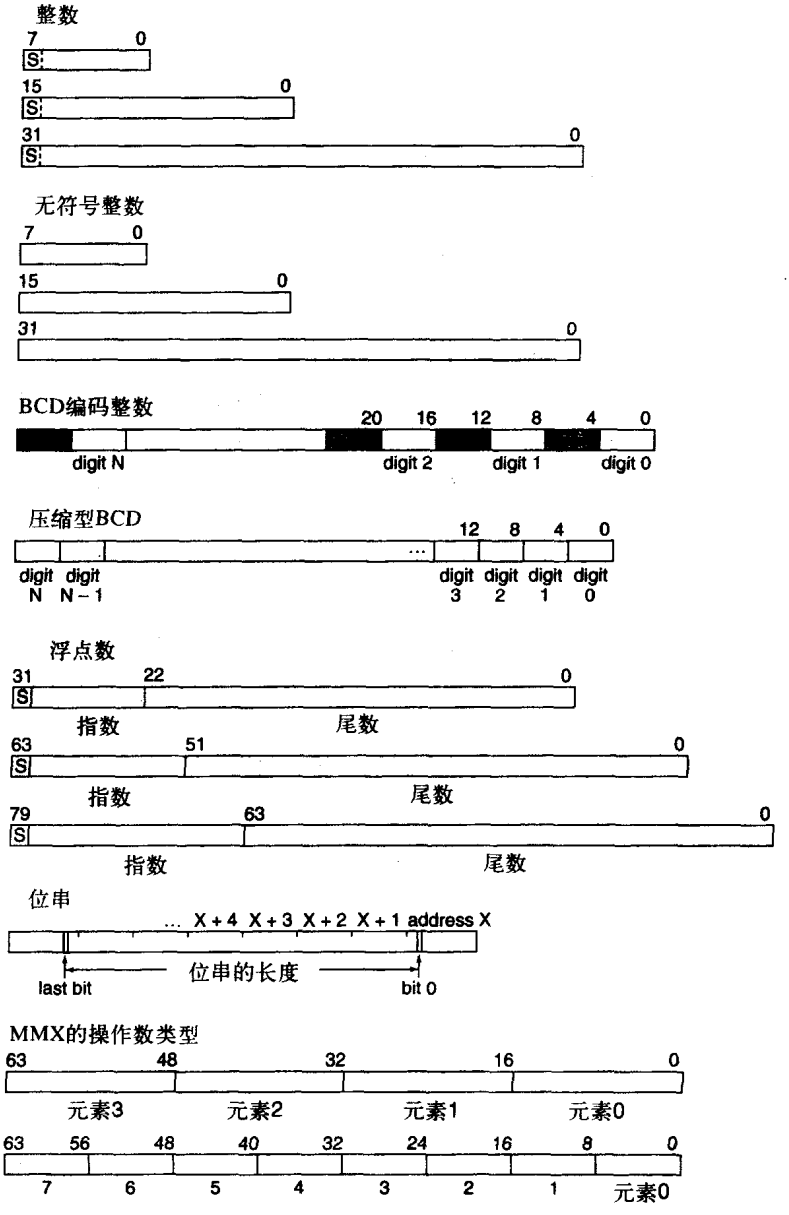
数据类型、200多个操作码和几十种寻址模式。

1. 数据类型

数据类型是指信息的表达形式，特定的指令系统只能处理特定编码类型的操作数。

LC-3只支持一种数据类型，即16位补码整数，当然这远不足以应对真实世界的计算需求，如：科学计算类应用程序需要浮点数类型；多媒体应用程序则需要另一种特定数据类型；一些仍在使用的早期商业程序，还使用一种称做压缩十进制数（packed decimal）的数据类型。总是存在着那么一些应用，它们比普通应用需要更大的数值范围和更高的数值精度。

由于这些需求的存在，x86设计了8-bit整数、16-bit整数、32-bit整数、32-bit浮点数、64-bit浮点数，以及64-bit和128-bit多媒体数等数值类型，及其相应的操作指令。图B-1所示是x86指令系统的各种数据类型。



图B-1 x86的各种数据类型

2. 操作码

LC-3包含15种操作码，而x86指令集则包含200多种操作码。但正如我们之前所论述的，无论一种指令系统的指令数目有多少，其基本类型都只有三类，即运算操作指令、数据搬移指令和控制指令。运算操作是指信息的运算处理，数据搬移是指将数据从一个地方拷贝到另一个地方（包括设备的输入/输出），而控制操作则是改变指令流的流向。

除此之外，我们还应该补充一种现实中必需的类型，即系统管理类指令。由于现实中用户程序并不是独立运行的，即它总是运行在操作系统所掌控的执行环境下的。换句话说，这些指令只能被操作系统（即特权程序）执行，而不能由应用程序执行。它们的功能涉及计算机安全、系统管理、硬件性能监测等一般应用程序无需关注的方面。在本附录中，我们将不讨论这些指令，但要提醒的是，它们确实存在，在今后的学习中，你将遇到它们（如操作系统课程）。

下面我们将重点讨论三种基本指令类型：操作指令、数据搬移指令和控制指令。

3. 运算操作指令

LC-3的运算操作指令有三条：ADD、AND和NOT。其中，ADD是LC-3中惟一的算术运算操作码。如果做减法，只需加该操作数的负值即可。如果想做乘法，则需要写一个循环加的程序。当然，这对于真实的微处理器来说是相当耗时的操作。因此，x86有专门的SUB、MUL、DIV、INC（递增）、DEC（递减）和ADC（带进位加法）等指令。

指令集的一个重要特性是，应该能操作更长的整数。通常的做法是写一个操作长整数的子程序，这其中需要用到一个类似ADC的操作码，即能够将两个操作数与之前加法操作的进位位（Carry）相加。

另外，x86还设计了针对每一种数据类型的操作码集。例如，多媒体指令集（即MMX指令），它与我们通常使用的算术运算不同，它能执行*饱和算术运算*（saturating arithmetic），如PADDS能对两个操作数做饱和算术运算。

饱和算术运算的含义是：假设我们用数字0到9代表一个元素的灰度，0代表白色，9代表黑色。如果我们希望添加一些黑色到已经有灰度的数字上，例如一个元素初始值为灰度7，然后对其添加5级灰度。采用正常的算术运算，结果是7+5=2（带一个进位）。这个结果竟然比7或5还小，一定出什么问题了！但在饱和算术运算方式下，得到的结果却是9，即最大饱和值，且不会产生进位。依次，“7 + 5 = 9”，“9 + n = 9”。饱和算术运算是一种特殊的运算，x86的MMX指令就是这样一种操作码。

此外，科学计算类程序需要的是能够操作浮点数的操作码。如x86指令系统中的FADD、FMUL、FSIN、FSQRT等指令。

有关逻辑运算指令，AND和NOT是LC-3仅有的两个逻辑运算操作码。当然，你可以基于这两个操作码，创建任意的逻辑表达式。然而，同算术运算指令一样，这也是非常耗时的。而x86则拥有更多的专用逻辑运算指令，如OR、XOR、AND-NOT等指令，以及针对不同数据类型的逻辑操作码。

此外，x86有许多其他的指令，可以设置和清除寄存器，将一个数值从一种类型转换成另一种类型，能够实现移位，或者旋转一个数据元素的位，等等。

表B-1列出了一些x86指令集中的操作码。

4. 数据搬移指令

LC-3的数据搬移指令有7条：LD、LDI、ST、STI、LDR、STR和LEA。除LEA是将地址值装入寄存器之外，其他指令都是在内存（或内存映射的设备寄存器）和8个通用寄存器（R0~R7）之间拷贝数据。

而x86除以上指令之外，还包括更多的数据搬移类指令。如XCHG用来交换两个地址中的内容；PUSHA将8个通用寄存器全部压入堆栈；IN和OUT专用于I/O地址空间访问，即在设备输入/输出端口和处理器之间传递数据；CMOVcc是条件执行（C）的数据搬移（MOV）指令，即只在之前计算条件（cc）为真时，才执行MOV操作。

表B-2列举了部分x86指令集中的数据搬移指令。

表B-1 x86的运算操作类指令

指 令	含 义
ADC x,y	x、y和进位位（CF）相加，结果存入x
MUL x	寄存器EAX内容与x相乘，64位结果存放在EDX、EAX之中
SAR x	x算术右移n位，结果存回x。其中n的数值可以是1、立即数或CL寄存器的值
XOR x,y	X和y的异或运算，结果存回x
DAA	两个十进制数相加（两个BCD编码数值事先存放在AL寄存器中）。由于进位门限是15而不是9，所以结果可能不正确。而DAA能纠正两个BCD数字
FSIN	栈顶内容弹出（假设称之为x），之后做sin(x)运算并将结果压回栈
FADD	栈顶的两个元素被弹出，相加并将结果压回栈
PANDN x,y	两个MMX数x和y，与非运算（AND-NOT），将结果存回x
PADDS x,y	两个MMX数x和y，饱和加法运算，结果存回x

表B-2 x86的数据搬移指令

指 令	含 义
MOV x,y	将y的内容拷贝到x中
XCHG x,y	x和y相互交换内容
PUSHA	将所有（ALL）寄存器压入堆栈
MOVS	将DS段偏移ESI处的数据拷贝到ES段偏移EDI处，且在拷贝完成之后，同时增量ESI和EDI
REP MOVS	反复（Repeat）执行MOVS操作。每执行一次MOV操作，ECX减1，直到ECX = 0。（该指令可用于拷贝字符串，将ECX内容初始化为字符串长度）
LODS	将DS段ESI指向的数据装入EAX，并根据当前DF标志，对ESI做递增或递减操作（DF:Descend/Forward）
INS	将DX寄存器指定的端口内容装入EAX寄存器或AX或AL，取决于数据宽度是32、16或8位
CMOVZ x,y	条件数据搬移指令。如果条件位ZF=1，则将y内容拷贝至x；如果ZF=0，则该指令作废（如同执行no-op指令一样）
LEA x,y	将y的地址（而不是内容）存入x。该指令同LC-3的LEA指令相似

表B-3 x86的控制类指令

指 令	含 义
JMP x	将地址x装入IP寄存器。该指令同LC-3的JMP指令相似
CALL x	将IP内容压栈，并将IP内容替换为x
RET	堆栈内容弹出，并将弹出内容的值装入IP
LOOP x	循环指令。将ECX减1，如果ECX非0且ZF = 1，则将x装入IP
INT n	软件中断指令。N代表操作系统中服务程序的索引号。该指令是将第n个服务程序的入口地址装入IP。该指令同LC-3的TRAP指令相似

5. *控制指令*

LC-3有5个控制指令：BR、JSR/JSRR、JMP、RTI和TRAP。相比之下，X86的控制指令数目

更多，表B-3列出了x86指令集的部分控制指令。

6. 两地址与三地址的比较

LC-3属于三地址类型的指令集，即ADD指令所显式要求的操作数数目为三个。加法操作需要两个源操作数（要相加的数）和一个目的操作数（存放结果）。由于在LC-3中，这3个操作数必须显式指定，因此称为三地址指令集。

注意，即使源操作数之一与目标操作数是同一个地址，也需要在指令中明确标注3个地址。例如，在LC-3格式的ADD R1,R1,R2指令中，R1既是源地址也是目标地址。

而x86则是一种两地址类型的指令集。在x86中，加法操作同样需要3个操作数，但源操作数之一的地址必须同时扮演目的地址（即存放结果）。例如，在x86中，相应的ADD指令如ADD EAX,EBX（EAX和EBX是8个通用寄存器中的两个的名字），则EAX和EBX是源操作数，而目的操作数也是EAX。由于操作结果被存储在最初的源地址中，就意味着一旦指令执行之后，该源操作数将被覆盖。所以，如果源操作数在当前指令之后还需要，那么必须在指令执行之前将其妥善保存。

7. 内存操作数

LC-3指令集与x86指令集最主要的区别之一，就是操作数的来源限制。在LC-3运算指令中，操作数必须来自寄存器，且结果也必然写入寄存器。而在x86指令中，操作数既可以来自寄存器也可以来自内存，且允许结果也写回内存。换句话说，x86可以在一条指令中完成从内存中读取数值，然后运算操作该数值，且最终又将结果写回内存等一连串操作（这是LC-3无法做到的）。

若要完成同样任务（如上所述），在LC-3程序中则需要多条指令才能完成。即先是通过load指令将其从内存读入某个寄存器，然后对其运算，并将结果存放在某个寄存器中，最后再通过store指令将结果写回内存。我们称具有这种特性（即对操作数来源有限制）的指令集（如LC-3）为“load-store”类型指令集，显然x86不是load-store类型指令集。

B.1.2 内存空间

LC-3的空间内存有2^{16}个地址，而每个地址中包含的信息位宽度也是16位。之所以称“LC-3具有16位的地址空间”，是因为16位地址可以独立寻址2^{16}个内存位置。之所以称“LC-3的寻址有16位寻址能力”，是因为其中的每个内存地址中包含的信息宽度是16位。

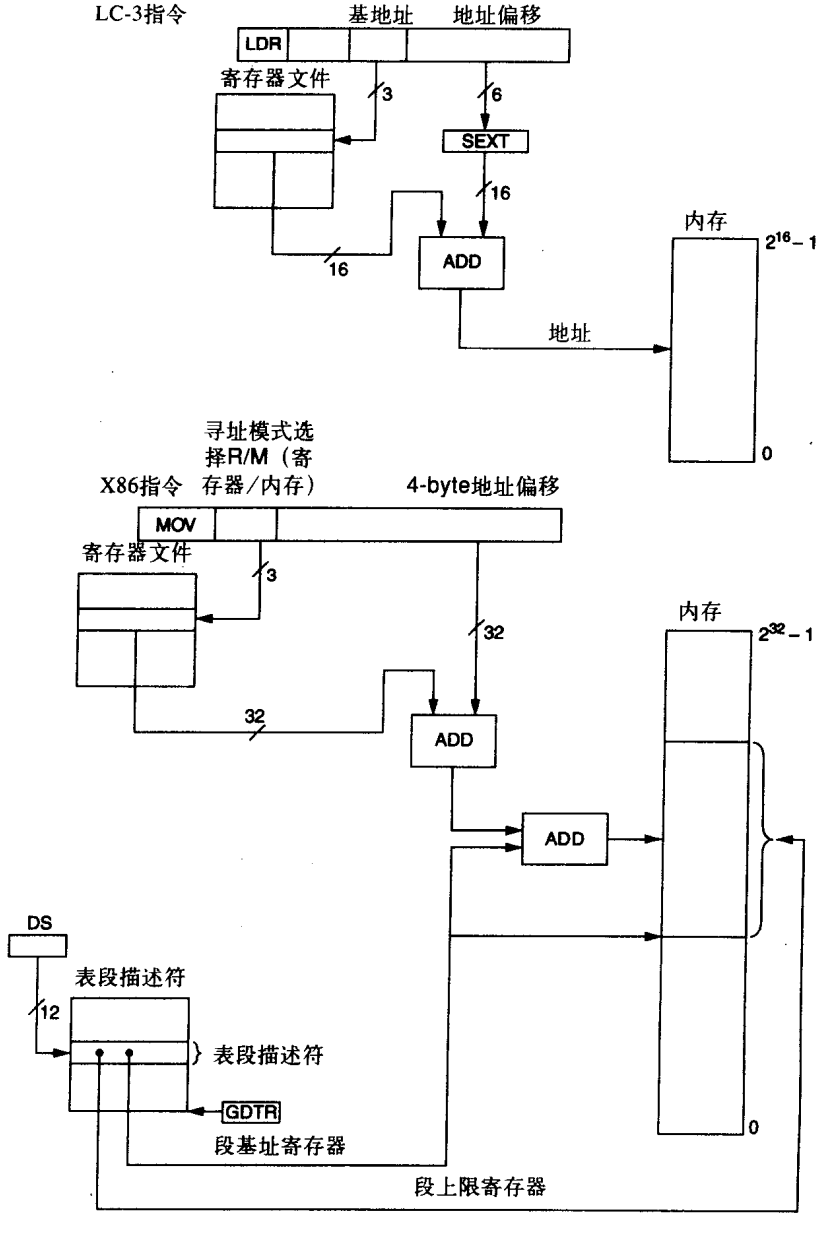
我们称“x86内存拥有32位的地址空间和8位寻址能力”。由于每字节包含 8个bit，我们又称x86是“字节寻址”。由于每字节宽度是8位，即每个地址仅包含8位，那么内存中连续的4个单元可以存放一个32位的数据。假设它们的地址分别为X、X+1、X+2和X+3，我们称该32位数据单元的地址是X。该32位数值中各bit信息的分布是，X仅仅包含其[7:0]位，X+1包含[15:8]位，X+2包含[23:16]位，X+3包含[31:24]位。

基于LC-3的三种寻址模式，我们总可以从LC-3指令的地址信息，推算出实际的内存地址。相比之下，x86指令则提供了多达24种用于推算内存地址的寻址模式（参见B.2节中的x86寻址模式）。

除寻址模式很多之外，x86内存寻址中还存在一种“段（segmentation）机制”，段是一种能保护特定内存地址区域的机制，如能防止非法访问。在这种情况下，特定寻址模式所产生的地址不再是简单的直接地址，而是某个内存段中的偏移地址。对该地址的访问，受控于对应段寄存器的访问控制权限。有关保护机制的工作原理，参见后面章节。

图B-2分别描述了在LC-3和x86段机制下，“寄存器加偏移”（Register + offset）寻址模式的地址计算过程。两者的操作目的都是将内存数据拷贝到通用寄存器，LC-3使用LDR指令，而x86使用MOV指令。在x86的地址计算中，目标地址位于内存的DS区段，即必须通过DS寄存器来访问。通

过一个16位段选择器（selector）索引段描述符表，获取该段对应的段描述符（segment descriptor）。该段描述符中包含了段基址寄存器（segment base register）、段上限寄存器（segment limit register）以及该段的保护信息。直接从指令寻址模式推算出来的内存地址（即偏移）与段基址寄存器内容相加，从而计算出实际内存地址，如图B-2所示。



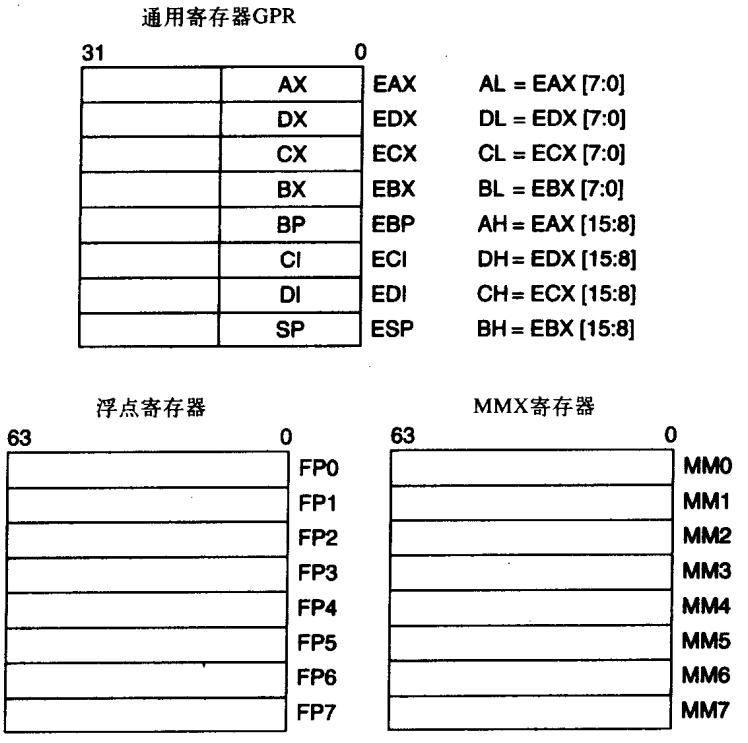
图B-2 LC-3和X86结构的“寄存器+偏移”寻址模式的比较

B.1.3 内部状态

所谓内部状态，是指LC-3中由8个16位通用寄存器（R0到R7）、1个16位PC寄存器和1个16位PSR寄存器（特权模式、优先级和3个N/Z/P）组成的状态集合。同样，x86中，用户可见的内部状态包括应用程序可见的寄存器、指令指针、标志寄存器和段寄存器等。

1. 应用程序可见的寄存器

图B-3显示的是在x86指令集中应用程序可见的部分寄存器（Application-Visible Registers）。



图B-3 x86中应用程序可见的一些寄存器

同LC-3的R0~R7寄存器相似，x86也有8个通用寄存器，即EAX、EBX、ECX、EDX、ESP、EBP、ECI和EDI等。每个寄存器的宽度都是32位，即通常情况下操作数的大小为32位。但是，由于x86还提供了处理16位操作数和8位操作数的指令，按道理它也应该提供16位和8位的寄存器，事实上，指令集对每个32位寄存器的低16位，以及低16位中的低8位和高8位也分别做了标识。换句话说，x86还提供了16位和8位宽度的寄存器。如AX、BX和DI就是16位的寄存器，而AL、BL、CL、DL、AH、BH、CH和DH则是8位寄存器。

另外，x86还提供了一些64位寄存器，用来存放浮点数和MMX计算需要的数值。它们分别是FP0~FP7和MM0~MM7。

2. 系统寄存器

所谓系统寄存器（System level Registers），就是那些只有在特权模式下（如操作系统运行时）才可见的寄存器。LC-3有两个系统级寄存器：PC和PSR。在x86下，用户可见的系统寄存器也有类似的寄存器，甚至更多。

图B-4给出了x86指令集中用户可见的一些系统寄存器。

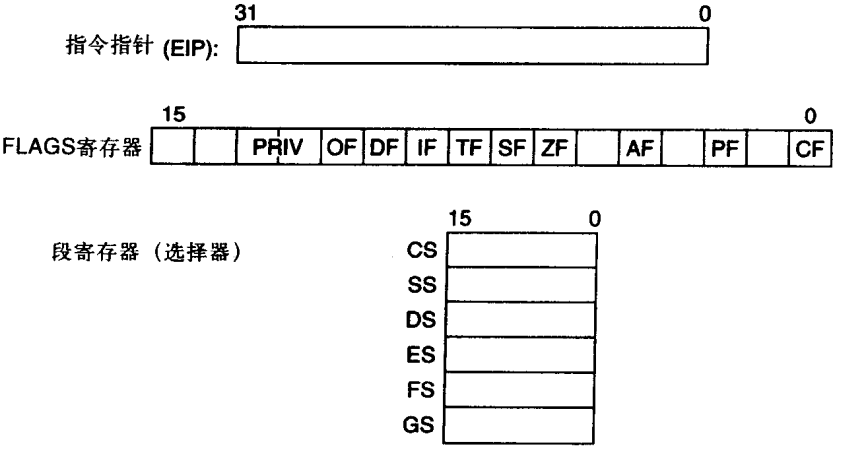
3. 指令指针

x86有与LC-3的16位程序计数器（PC）完全等价的寄存器，x86称之为指令指针（Instruction Pointer, IP）。由于x86的地址空间是32位的，所以IP寄存器也是32位的。

4. 标志寄存器

与LC-3的N、Z、P条件码相比较，x86有1个1位的SF（sign flag）寄存器和1个1位的ZF（zero flag）寄存器。SF和ZF的功能分别对应LC-3的N和Z条件码。在x86中，没有和LC-3的P条件码等价

的寄存器。事实上，P条件码确实是多余的，因为如果知道N和Z的值，自然就知道了P的值。但是，为了方便汇编程序员和编译器开发者，我们还是将其包含在LC-3的指令集中。



图B-4 x86系统寄存器

此外，除了N和Z以外，x86还提供了更多的1位宽度的标志位（flag），它们被包含在一个名为FLAGS的16位标志寄存器（FLAGS Register）中。其他标志位的讨论，参见后面内容。

CF标志位存放的是之前上一个相关指令产生的进位（carry）。正如之前ADC指令描述中所提到的，CF的存在使得软件可以运算处理超过指令集规定长度的大整数。

OF标志则用于存放上一个指令运算产生的超出存储宽度的有效位数，即OF标志存放的是溢出位（overflow）。参见2.5.3中有关溢出的讨论。

DF标志字符串操作的方向（direction）。如果DF = 0，字符串操作从高地址字节开始（即指针字符串，记录下一个被操作字符的指针是递减的）。如果DF = 1，则字符串操作从低地址字节开始（即字符串指针是递增的）。

另外，还有两个通常不被认为属于应用程序状态的标志位：IF（中断）标志和TF（陷入）标志。两者的功能你应该是熟悉的，IF的功能和8.5节的KBSR和DSR寄存器中的IE（interrupt enable）标志类似。如果IF= 1，即允许处理器接收外部中断（比如键盘输入）。如果IF = 0，外部中断将不会“打扰”当前执行的进程，我们称之为“中断关闭”状态。

TF标志和LC-3仿真器中的单步模式（single-step mode）也非常类似，且仅在此方式下，我们才认为它属于指令集定义。如果TF = 1，处理器在每执行完一条指令之后，就会中止（并陷入），借此机会可以检查系统状态。如果TF = 0，处理器就会跳过陷阱（trap），继续下一条指令的执行。

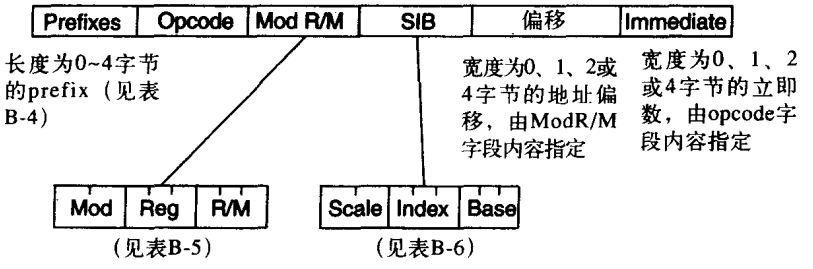
5. 段寄存器

在保护模式（protected mode）下，指令计算出来的地址事实上只是相对某个段起始地址的一个偏移量，而该段起始地址则由对应的段基址寄存器（segment base register）所指定。段基址寄存器是数据段描述符（data segment descriptior）结构的各个字段之一，而“段描述符”又是集中存放在“段描述符表”（segment descriptor table）中的。在任意时刻，所有这些段中总有六个是处于活动状态的，它们分别是代码段（CS）、堆栈段（SS）和4个数据段（DS、ES、FS和GS）。六个活动段分别通过图B-4所示的段寄存器来访问，这些段寄存器中，包含的是指向对应段描述符的指针。

B.2 x86指令的格式和规格说明

LC-3指令的长度是固定的16位。其中，[15:12]位是操作码，余下12位的含义随操作码的不同而不同。

而x86指令的长度是不固定的，不同的指令有不同的字节长度，因而一个x86指令可以包含很多信息。图B-5所示是一个x86指令的格式，该指令长度为15个字节。



图B-5 x86指令格式

x86指令中的两个关键部分是：操作码和ModR/M字节。操作码定义了指令的操作功能，ModR/M字节则规定了如何能够获得操作数。ModR/M字节定义了寻址模式、使用的寄存器和偏移量（1、2或4字节）。有关寄存器的具体信息，被编码在1个SIB字节中。SIB字节和偏移量（如果需要的话）都跟在指令中ModR/M字节的后面。

有些操作码是对立即数操作的，并规定了指令中用来存放立即数的字节数。如果存在立即数，则它的内容被存放在指令的最后。

最后，指令中还定义了如地址大小、操作数大小、所使用的段等默认信息。如果希望改变这些默认信息，可以通过在指令头部添加一个或多个前缀（prefix）的方式，来表示对这些默认信息的修改。

下面将在B.2.1~B.2.6中，详细讨论x86指令的各个字段。

B.2.1 前缀

前缀（Prefix）提供了用于指令处理的额外信息。前缀信息有4类，根据指令需求的不同，一条指令的前缀长度可以是0~4个。换句话说，前缀起到了重新定义（更准确地说，应该是“扩展”）指令的目的。

4类前缀分别是锁与重复（lock & repeat）、段重载（segment override）、操作数重载（operand override）和地址重载（address override）。表B-4是4种前缀的详细描述。

表B-4 x86指令集的指令前缀

重复/锁（Repeat/Lock）	
xF0(LOCK)	该前缀确保在当前指令完成之前，共享内存不能被共享（即独占）
xF2,xF3(REP/REPE/REPNE)	该前缀表示允许当前指令（如字符串操作指令）重复执行，重复次数由ECX寄存器指定。另外，也可以指定该重复执行在特定ZF标志值中止
段重载（Segment override）	
x2E(CS),x36(SS), x3E(DS),x26(ES), x64(FS),x65(GS)	该前缀指定指令使用特定段（而不是默认段）访问内存
操作数重载（Operand size override）	
x66	该前缀指定数据长度。换句话说，如果默认数据长度为32位的数据操作指令，则可以通过该前缀操作16位数。同样，16位数操作指令也可以通过前缀操作32位数据

（续）

地址重载（Address size override） x67	该前缀能改变操作数地址的预定大小。换句话说，默认地址长度为32位的数据访问指令可以使用16位地址。同样，默认地址长度是16位的指令也可以操作32位地址

B.2.2 操作码

操作码（Opcode）字段（1个或2个字节）规定了指令功能等很多信息。操作码字段定义了指令的具体操作功能，以及操作数是从内存还是寄存器中获得、操作数大小、源操作数是否是立即数及立即数大小等信息。

有些操作码还要借用ModR/M字段的位置，即操作码字段和ModR/M字段[5:3]合在一起构成操作码（当然，此时ModR/M字段不再代表地址模式信息）。有关ModR/M字段的描述，参见B.2.3节。

B.2.3 ModR/M字节

如图B-5所示，ModR/M字段在需要时将提供1或2个操作数的寻址模式信息。如果指令需要两个操作数，那么其中一个可能在内存，另一个可能在寄存器中；或者两者都在寄存器中。如果只需要一个操作数，则可能是寄存器也可能是内存。通过ModR/M字段（R/M=Reg/Mem）的标识，以上各种组合方式都可以被识别。

ModR/M字段可以划分为两部分，其中，第一部分由[7:6]和[2:0]位组成，第二部分由[5:3]位组成。

如果[7:6]位 = 00、01或者10，即意味着第一部分标识内存操作数的寻址模式，即[7:6]和[2:0]组合在一起的5位，标识了具体的寻址模式。如果[7:6]位 = 11，则表示没有内存操作数，此时[2:0]位代表的是一个寄存器操作数。

如果指令需要两个操作数，则[5:3]位表示另一个操作数的寄存器编号。如果指令只需要一个操作数，则[5:3]位将作为操作码字段的补充码（sub opcode），用来区分操作码字段内容完全相同的8种指令，参见B.2.2节所述。

表B-5列举了ModR/M字段的含义。

表B-5 ModR/M字段及其例子

Mod	Reg	R/M	Eff.Addr	Reg	含 义
00	011	000	[EAX]	EBX	EAX包含的是内存操作数的地址，EBX是寄存器操作数
01	010	000	disp8[EAX]	EDX	EAX内容和指令中的偏移量字段相加，求得内存操作数的地址，EDX包含寄存器操作数
10	000	100	disp32[-][-]	EAX	将4字节（32位）偏移量和SIB字段指定的内容相加，获得最终的内存操作数地址，EAX是寄存器操作数
11	001	110	ESI	ECX	如果指令需要的两个操作数都来自寄存器（如ESI和ECX），如果指令只需要一个操作数，则它在ESI中，那么此时001(bits[5:3])作为操作码的子码

表B-6 SIB字段（字节）及其例子

Scale	Index	Base	Computation	含 义
00	011	000	EBX+EAX	EBX内容与EAX相加，再将相加的结果同ModR/M指定的内容相加
01	000	001	2×EAX+ECX	EAX乘2与ECX内容相加，再与ModR/M指定的内容相加
01	100	001	ECX	将ECX内容同ModR/M指定的内容相加
10	110	010	4×ESI+EDX	ESI乘4与EDX的内容相加，再与ModR/M指定的内容相加

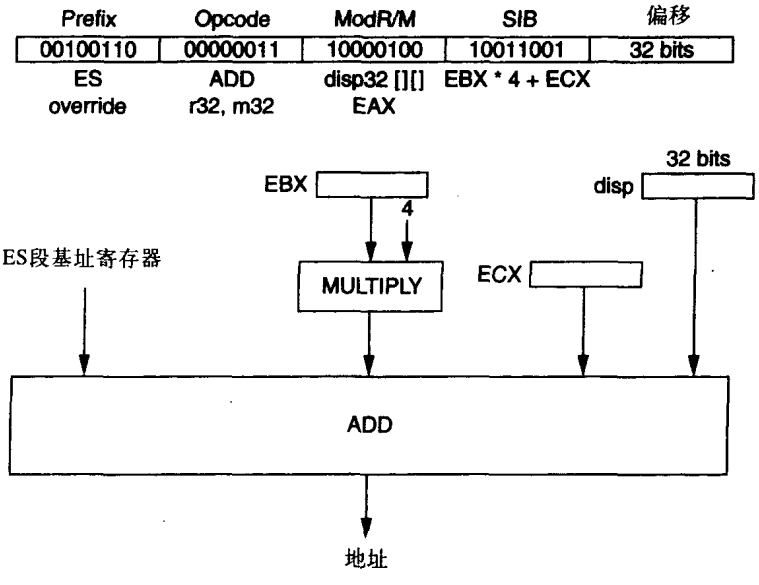
B.2.4 SIB字节

如果指令操作码定义了从内存中读取操作数，则由ModR/M字段指定具体的寻址模式，即有关操作数地址的计算信息。但是，有些寻址模式超出了ModR/M字段所能定义的范围，那么这些操作数的定义（如表B-5中第3个例子）还有赖于SIB字段信息。SIB字段定义了一种更灵活的“比例-索引-基址”（scale-index-base）计算方式，图B-5所示是比例参数、索引寄存器和基址寄存器。SIB的计算方式是“scale × index + base”，其中base和index允许为0，scale允许为1。表B-6是对SIB字段的描述。

B.2.5 偏移量

如果ModR/M字段指明地址计算需要偏移量信息，那么这个偏移量（可能是1、2或4字节）必然包含在当前指令中。偏移量所占的具体字节数，由操作码或ModR/M字段指定。

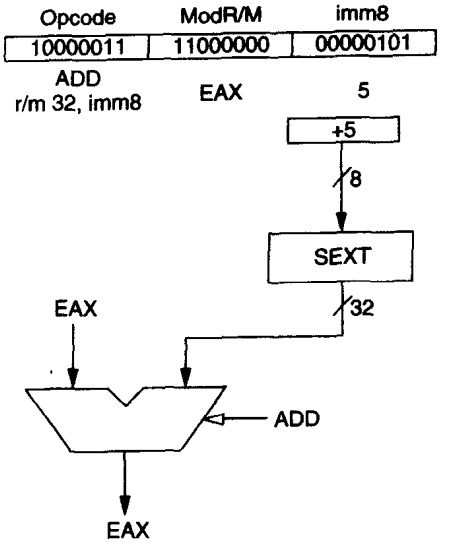
图B-6所示描述了图中例子指令的寻址模式及其地址计算过程。前缀x26重载了段寄存器，即指定使用ES段。这是一条内存操作数和寄存器操作数的求和（ADD）运算指令，其中ModR/M和SIB字节的信息，表示了内存操作数的地址计算模式参数，即SIB（scale × index + base）基址加偏移（displacement）的寻址模式，其中偏移量大小是4个字节（Mod=10），Scale值为4（10），索引寄存器是EBX（011），基址寄存器是ECX（001）。



图B-6 “Base+ScaledIndes+disp32”寻址模式的地址计算方法

B.2.6 立即数

之前我们曾介绍过，LC-3允许在指令中嵌入一个较小的立即数（通过设置指令位inst[5:5]=1）。同样，x86也允许在指令中嵌入立即数。如前面的描述，如果操作码指定源操作数之一是立即数，必然也将指明该立即数的长度，即所占的字节数。因此，在指令中的立即数，可以表示成1、2或者4字节。由于操作码同时也定义了操作数的大小[⊖]，因而实际在指令中内嵌的立即数所占的字节数，可以比操作数大小要小，在真正运算操作之前，可以通过符号扩展（sign-extend）的方式将其扩展为完整大小。图B-7显示了在ADD指令中（ADD EAX,$5），立即数操作数的使用方法，它与我们熟悉的LC-3指令很相似（如ADD R0,R0,#5）。



图B-7 x86指令：ADD EAX,$5

B.3 例子

我们用一个例子来总结这个附录。问题是我们在第14章处理过的。假设输入的字符串由文字、数字和标点符号组成，写一个C程序，转换所有的小写字母为大写字母。图B-8展示了解决这个问题的C程序。图B-9展示了C编译器产生的经过注释的LC-3汇编语言。图B-10展示了相应的经过注释的x86汇编语言。为了可读性，我们展示的是LC-3和x86的汇编语言程序而不是机器码。

```
#include <stdio.h>

void UpcaseString(char inputString[]);

main ()
{
    char string[8];

    scanf("%s", string);
    UpcaseString(string);
}

void UpcaseString(char inputString[])
{
  int i = 0;

  while(inputString[i]) {
    if (('a' <= inputString[i]) && (inputString[i] <= 'z'))
      inputString[i] = inputString[i] - ('a' - 'A');
    i++;
  }
}
```

图B-8 大小写字符转换程序（C语言版）

```
        ; uppercase:  converts lower- to uppercase
                    .ORIG  x3000
                    LEA    R6, STACK
        MAIN        ADD    R1, R6, #3
```

图B-9 大小写字符转换程序（LC-3版）

⊖ 注意“立即数（或操作数）的大小”和“立即数的长度”在此处的区别。前者是指它逻辑上所代表的数值宽度（32位或16位、8位），后者表示它在指令中实际占用的长度或宽度。——译者注

```
            LEA     R6, STACK
MAIN        ADD     R1, R6, #3
READCHAR    IN                      ; read in input string: scanf
            OUT
            STR     R0, R1, #0
            ADD     R1, R1, #1
            ADD     R2, R0, x-A
            BRnp    READCHAR
            ADD     R1, R1, #-1
            STR     R2, R1, #0      ; put in NULL char to mark the "end"
            ADD     R1, R6, #3      ; get the starting address of the string
            STR     R1, R6, #14     ; pass the parameter
            STR     R6, R6, #13
            ADD     R6, R6, #11
            JSR     UPPERCASE
            HALT
UPPERCASE   STR     R7, R6, #1
            AND     R1, R1, #0
            STR     R1, R6, #4
            LDR     R2, R6, #3
CONVERT     ADD     R3, R1, R2      ; add index to starting addr of string
            LDR     R4, R3, #0
            BRz     DONE            ; Done if NULL char reached
            LD      R5, a
            ADD     R5, R5, R4      ; 'a' <= input string
            BRn     NEXT
            LD      R5, z
            ADD     R5, R4, R5      ; input string <= 'z'
            BRp     NEXT
            LD      R5, asubA       ; convert to uppercase
            ADD     R4, R4, R5
            STR     R4, R3, #0
NEXT        ADD     R1, R1, #1      ; increment the array index, i
            STR     R1, R6, #4
            BRnzp   CONVERT
DONE        LDR     R7, R6, #1
            LDR     R6, R6, #2
            RET
a           .FILL   #-97
z           .FILL   #-122
asubA       .FILL   #-32
```

图B-9 大小写字符转换程序（LC-3版）（续）

```
.386P
.model FLAT

_DATA   SEGMENT                 ; The NULL-terminated scanf format
$SG397  DB      '%s', 00H       ; string is stored in global data space.
_DATA   ENDS

_TEXT   SEGMENT

_string$ = -8                   ; Location of "string" in local stack
_main   PROC NEAR
        sub   esp, 8            ; Allocate stack space to store "string"
        lea   eax, DWORD PTR _string$[esp+8]
        push  eax               ; Push arguments to scanf
        push  OFFSET FLAT:$SG397
        call  _scanf

        lea   ecx, DWORD PTR _string$[esp+16]
        push  ecx               ; Push argument to UpcaseString
        call  _UpcaseString

        add   esp, 20           ; Release local stack space
        ret   0
_main   ENDP
```

图B-10 大小写字符转换程序（X86版）

```
_inputString$ = 8              ; "inputString" location in local stack
_UpcaseString PROC NEAR
        mov   ecx, DWORD PTR _inputString$[esp-4]
        cmp   BYTE PTR [ecx], 0
        je    SHORT $L404           ; If inputString[0]==0, skip the loop
$L403:  mov   al, BYTE PTR [ecx]    ; Load inputString[i] into AL
        cmp   al, 97                ; 97 == 'a'
        jl    SHORT $L405
        cmp   al, 122               ; 122 == 'z'
        jg    SHORT $L405
        sub   al, 32                ; 32 == 'a' - 'A'
        mov   BYTE PTR [ecx], al
$L405:  inc   ecx                   ; i++ %$
        mov   al, BYTE PTR [ecx]
        test  al, al
        jne   SHORT $L403           ; Loop if inputString[i] != 0
$L404:  ret   0
_UpcaseString ENDP
_TEXT   ENDS
END
```

图B-10 大小写字符转换程序（X86版）（续）

附录C　LC-3的微结构

从第4章和第5章的学习中我们获知，计算机在处理指令时，每条指令的处理周期事实上包括很多子节拍，而这些节拍及其之间的衔接操作，是由一个微结构（microarchitecture）负责完成的。换句话说，如果将指令集理解为硬件与软件的接口，则微结构就是接口之下的硬件实现。微结构必须在不超过一条指令周期长度的时间内，完成一条指令的所有具体操作。本附录以LC-3为例子，介绍LC-3微结构实现的方法之一及其实现细节。事实上，微结构的许多设计细节、设计方案的选择及其设计理念的知识，已超出入门课程的范围。所以，本书将这部分讨论放在附录中供你参考，它主要介绍了微结构是**怎样**执行LC-3的各种指令的。

C.1　概述

如图C-1所示，数据通路和控制器是指令集结构（ISA）中最重要的两个组成部分。*数据通路*包含了与指令处理相关的所有部件；而*控制器*则根据读入的指令（主要是操作码）产生用于控制数据通路的信号，即“指挥”各个处理单元在“特定时刻”做“特定事情”，直至指令结束。

这里所说的“特定时刻”，即指在每个时钟周期内。在计算机术语中，我们通常以*时钟周期*（clock cycle）为单位（而不是以秒为单位）计量时间。一个周期时间（cycle time），就是相邻周期的间隔时间。在今天的微处理器中，一个周期时间大约是0.5ns，换句话说，每秒钟对应20亿个时钟周期，再换种说法，我们称该微处理器的操作频率是2GHz。

在每个特定时间内——或者每个时钟周期内，有49个控制信号（如图C-1所示）在控制着数据通路的运转，并准备着下一时钟周期的控制信号产生。其中39个信号控制着当前的数据通路，10个信号负责下一时钟周期控制信号的产生。

注意，控制信号的产生和有序操作是一个高度复杂的工作，换句话说，它不是在“真空”（vacuum）完成的。相反，决定一个时钟周期内需要产生什么样的控制信号，取决于以下诸多因素：

（1）当前时钟周期的工作状态。

（2）正在执行的LC-3指令。

（3）正在运行程序的权限模式（privilege）。

（4）如果当前指令是BR，跳转条件（即相关条件码的值）是否满足。

（5）当前是否有外部设备在请求中断处理器。

（6）当前指令的内存访问在本周期内是否能完成。

以上各个问题在图C-1中都有答案，如下所述：

（1）当前时钟周期所需的10个控制信号是J[5:0]、COND[2:0]和IRD。

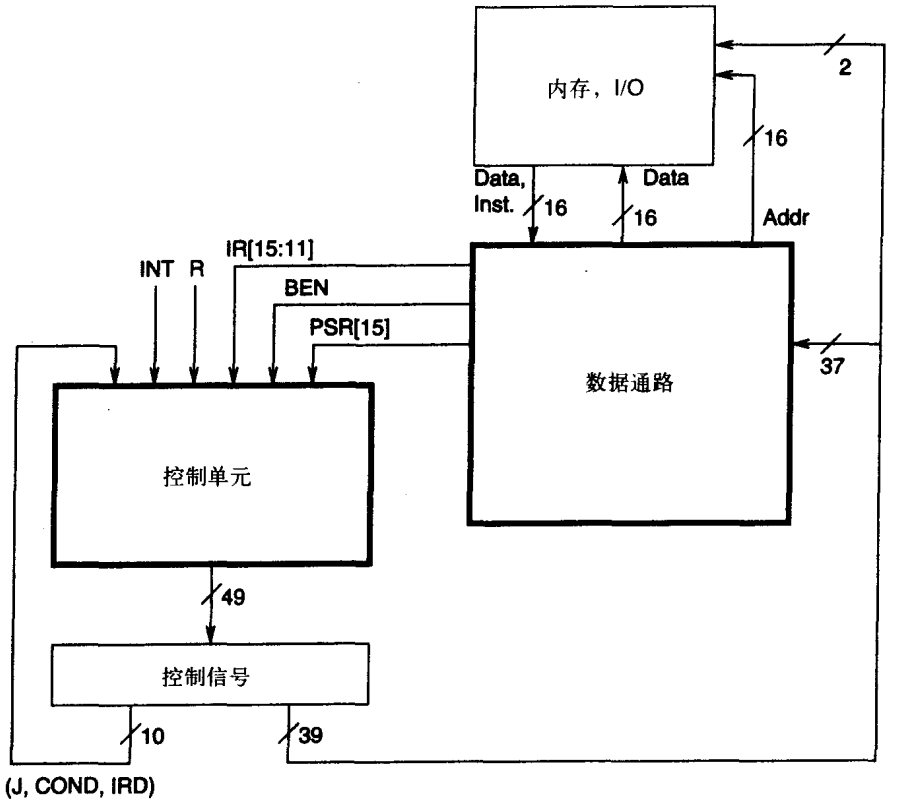
（2）指令中操作码字段（inst[15:12]）确定了具体的指令，而inst[11:11]位则确定了寻址模式，如JSR和JSRR之间，该位值就不同，即寻址子程序地址的方法不同。

（3）PSR[15]即为处理器状态寄存器PSR的bit[15]，标识了当前程序的执行权限（privilege）是特权级（supervisor）还是用户级（user）。

（4）BEN指示跳转（BR）是否执行。

（5）INT表示当前有个优先级比执行进程优先级更高的外部设备，正在请求服务。

（6）R（Ready）表示内存操作结束。



图C-1 LC-3微结构的主要部件

C.2 状态机

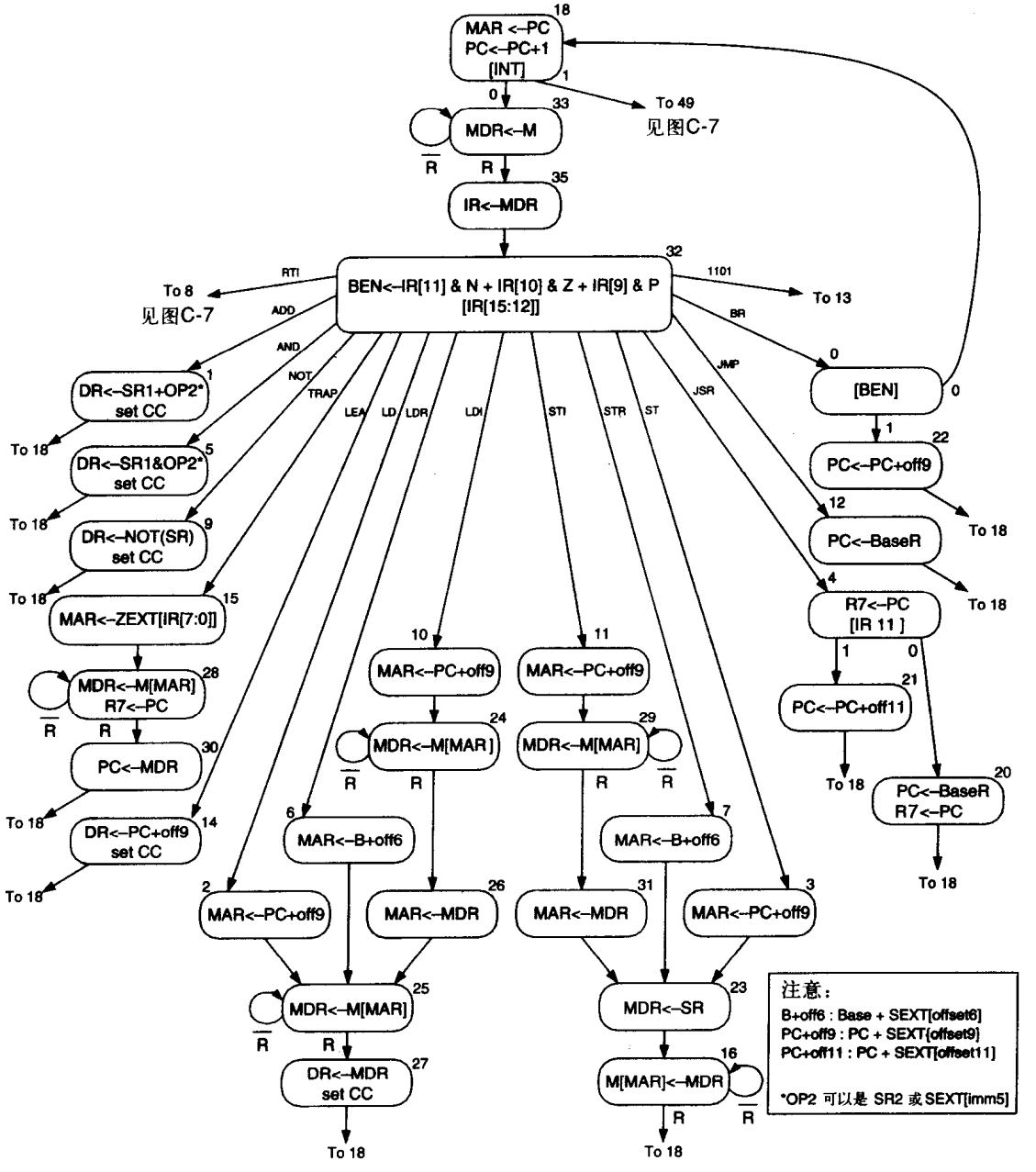
在一个特定的时钟周期内，LC-3微结构的行为完全受控于49个控制信号和9个额外的信息位（即inst[15:11]、PSR[15]、BEN、INT和R），如图C-1所示。我们已介绍过，其中的39个控制信号控制着当前数据通路的运行，而另外10个信号与9个信息位，决定了在下一个时钟周期的控制信号应该是什么样的。

我们称这49个控制信号定义了LC-3微结构之控制结构的状态（state），那么我们可以将LC-3微结构的行为描述为一个有向图（directed graph），由节点（node）和连线（arc）组成。其中，节点即对应了每个状态，连线表示从一个状态转移到下一个状态，我们称这样一张图为状态机（state machine）。

图C-2是LC-3设计实现所对应的状态图。状态机描述了计算机运行时，每个时钟周期内的（处理器）运行状态。每个状态点对应一个时钟周期；反之，每个时钟周期内也只对应一个状态点，即下一个时钟周期开始将转至下一个状态点。事实上，整个状态机描述了一个指令从FETCH状态点开始，逐步（step-by-step或clock-cycle-by-clock-cycle）地操作直至指令完成的全过程（如4.2.2节的描述）。由于一条指令的执行要经历多个时钟周期，所以每个节点反映的是当前周期内处理器的活动状态（是控制信号而不是操作细节），真正的操作事实上是由数据通路部件完成的。

首先，我们回顾一下第4章中所介绍的取指令（FETCH）操作。每个指令周期开始的第一项工作，就是读取由PC指向地址中的指令。参见“状态18”中的注释，将PC内容装入MAR，同时

PC+1，为下一个取指令操作做好准备，然后检查是否存在中断请求（INT标志）。如果没有中断请求（INT=0），则转入下一个状态（33）；如果有中断请求，则跳转至中断服务程序（参见C.6节）。

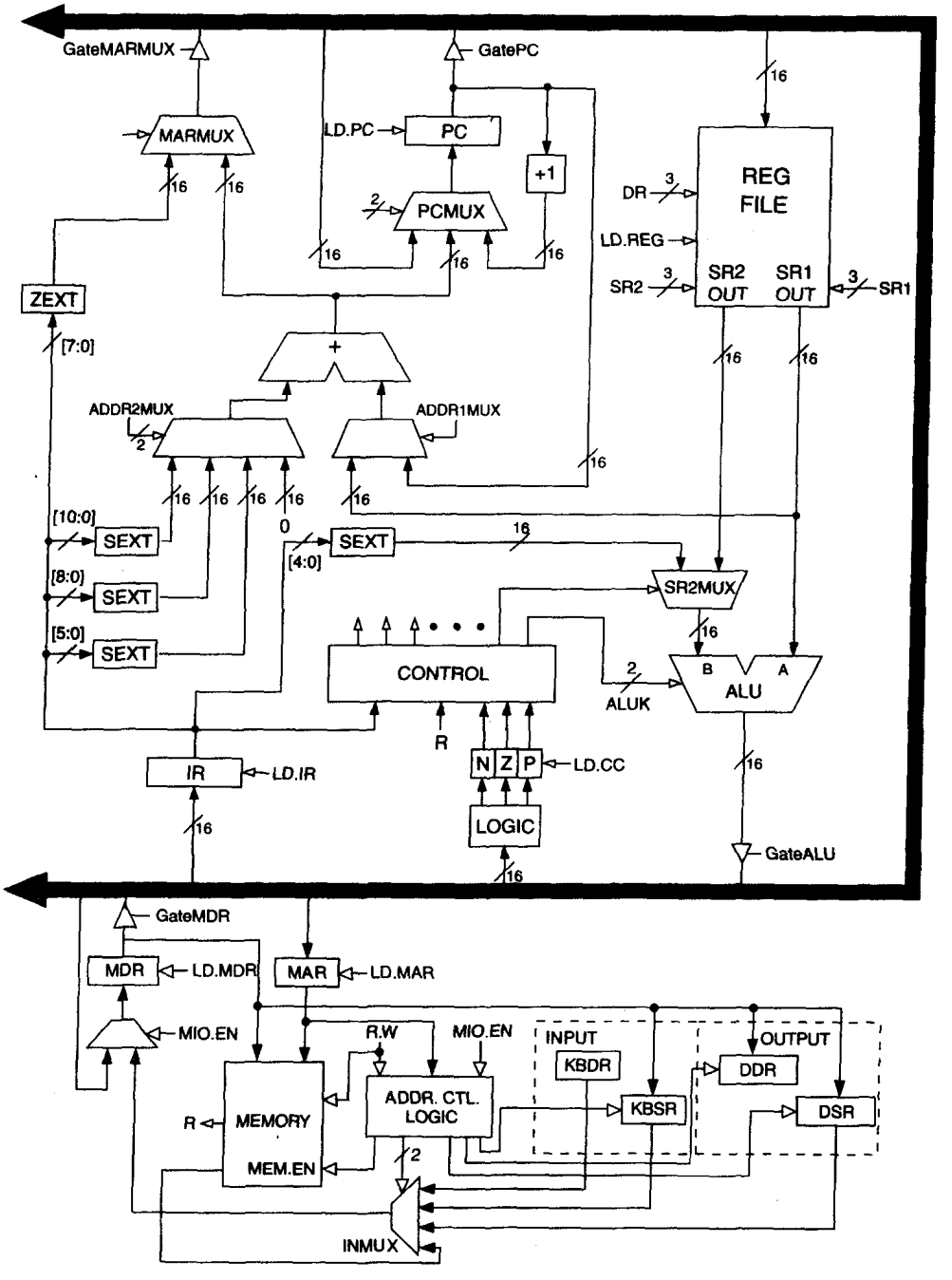


图C-2 LC-3状态机

在进入状态33之前，我们先解释一下该状态机的编号系统，即为什么刚才遇到的两个状态的编号是18和33（而不是别的什么数字）？回顾一下第3章中有限状态机的介绍，首先每个状态都必须有一个惟一的编号，而这个“惟一性”是通过状态变量的方式完成的。LC-3 ISA共有52个状态，图C-2中标注了其中的31个，以及三个指向别处（状态8、13和49）的指针；图C-7标注了另外的18个及其指回图C-2的指针。由于k个逻辑变量（T/F或1/0）能组合的不同数值是2^k个，所以52个条目需要6个状态变量。图C-2中，每个节点旁边的编号是二进制（0/1）状态变量的十进制数，如“状

态18”对应的状态变量是010010。

现在回到状态18结束后的场景。如果此时没有外部设备的中断请求，则状态图转移至状态33。在状态33中，由于MAR寄存器的内容是指令在内存中的地址，所以从MDR中读出的内容就是该指令。内存访问需要一定的延迟，所以该状态不断执行，直到接收到来自内存的R（Ready）信号（表示内存访问结束），即MDR的内容是来自MAR所指向内存单元的内容。于是，状态机转移至状态35，即将MDR中的指令再次搬移至指令寄存器（Instruction Register）。这就是指令周期中“取指令”节拍的全过程。



图C-3 LC-3数据通路

注意状态18的重要性，之前的任意指令执行结束后（即不同指令对应指令周期的最后一个状态），都会跳转至状态18（下一个指令周期的第一个状态）。

表C-1 数据通路控制信号

信号	数值	
LD.MAR/1:	NO,LOAD	
LD.MDR/1:	NO,LOAD	
LD.IR/1:	NO,LOAD	
LD.BEN/1:	NO,LOAD	
LD.REG/1:	NO,LOAD	
LD.CC/1:	NO,LOAD	
LD.PC/1:	NO,LOAD	
LD.Priv/1:	NO,LOAD	
LD.SavedSSP/1:	NO,LOAD	
LD.SavedUSP/1:	NO,LOAD	
LD.Vector/1:	NO,LOAD	
GatePC/1:	NO,YES	
GateMDR/1:	NO,YES	
GateALU/1:	NO,YES	
GateMARMUX/1:	NO,YES	
GateVector/1:	NO,YES	
GatePC-1/1:	NO,YES	
GatePSR/1:	NO,YES	
GateSP/1:	NO,YES	
PCMUX/2:	PC+1	;select pc+1
	BUS	;select value from bus
	ADDER	;select output of address adder
DRMUX/2:	11.9	;destination IR[11:9]
	R7	;destination R7
	SP	;destination R6
SR1MUX/2:	11.9	;source IR[11:9]
	8.6	;source IR[8:6]
	SP	;source R6
ADDR1MUX/1:	PC,BaseR	
ADDR2MUX/2:	ZERO	;select the value zero
	offset6	;select SEXT[IR[5:0]]
	PCoffset9	;select SEXT[IR[8:0]]
	PCoffset11	;select SEXT[IR[10:0]]
SPMUX/2:	SP+1	;select stack pointer+1
	SP-1	;select stack pointer-1
	Saved SSP	;select saved Supervisor Stack Pointer
	Saved USP	;select saved User Stack Pointer
MARMUX/1:	7.0	;select ZEXT[IR[7:0]]
	ADDER	;select output of address adder
VectorMUX/2:	INTV	
	Priv.exception	
	Opc.exception	
PSRMUX/1:	individual settings, BUS	
ALUK/2:	ADD,AND,NOT,PASSA	
MIO.EN/1:	NO, YES	

（续）

信号	数值	
R.W/1:	RD, WR	
Set.Priv/1:	0	;Supervisor mode
	1	;User mode

C.3 数据通路

“数据通路”（data-path）是指在一个指令周期内，信息处理所涉及到的所有部件的总和，即那些处理信息的功能单元（function unit）、寄存器（周期结束时存放处理结果），以及总线（bus）和连线（wire）（在数据通路的各个单元之间传输信息）。图C-3所示的是LC-3数据通路的微结构实现，事实上是图5-9的一个扩展版。

在数据通路中，注意那些与每个部件相连的控制信号。如ALUK信号（事实上是由2个控制信号组成的）连接的是ALU部件。控制信号决定了在一个周期里部件的具体工作。表C-1列出了对应数据通路中各单元的控制信号及其不同取值（为了便于识别，我们标识的是数值的符号名，而不是其真正的二进制值）。例如，ALUK由2个位组成，则它有4种可能值。在特定时钟周期内，其具体数值取决于ALU单元的当前任务是ADD还是AND、NOT，亦或只是简单地将某个输入直通到ALU输出端；再如，PCMUX也是由2个控制信号组成，它决定了多路开关（MUX）的所有输入中，哪一个被选中了；LD.PC则是1位控制信号，取值0（NO）或1（YES），它决定了当前时钟周期内是否装载新的PC内容

任意一个时钟周期只对应状态机中的一个“当前状态”。39个控制信号控制着当前时钟周期内所有数据通路部件的工作，在一个时钟周期内数据通路中所进行的处理工作，如状态机中各个节点内所描述的。

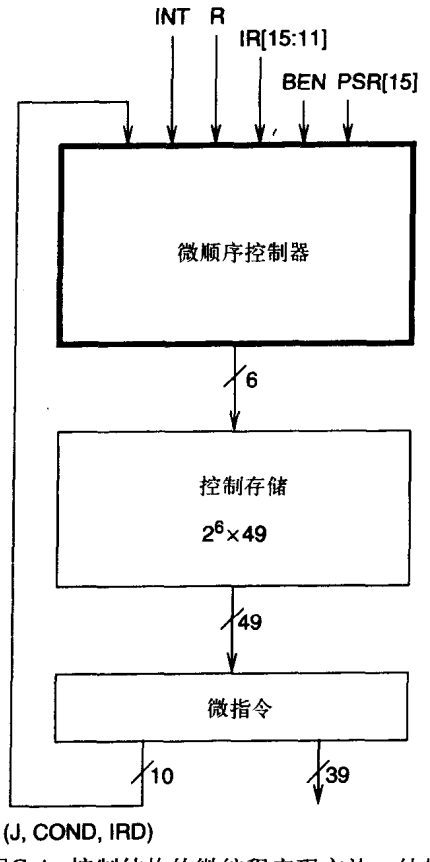
C.4 控制器结构

一个微结构的控制结构对应着状态机的定义。状态机（如图C-2所示）决定了每个时钟周期内的控制信号输出，这些控制信号控制着数据通路的处理过程，并决定着控制流（control flow）从当前状态到下一个状态的跃迁。

图C-4所示是LC-3的控制器结构图。控制器的工作原理如状态机所示，但具体实现方法却有很多种，最终的方案选择，取决于设计时的综合考虑，即计算机体系结构的设计评估。

我们选择的是一种简单的直接微编程（straightforward microprogrammed）方法[⊖]。由于在任意状态下，控制器需要39个信号来控制数据通路的正常工作，另外10个用于确定下一个状态，我们将这49根信号的集合统称为*微指令*（microinstruction）。换句话说，每条微指令的宽度是49位（对应状态机的一个特定状态）。而在控制器内部有一个专用的存储器即“控制存储”（control store），用来存放这些特定的微指令，其中每个存储单元的宽度都是49位。控制存储中共有52个存储单元，对应状态机的52个不同状态，这也意味着，需要一个6-bit的地址来标识每一条微指令。

⊖ 所谓“直接微编程”方法，就是微指令的每个bit对应一个控制信号，与之对应的则是更为复杂一些的“间接”（indirect）或“多级微指令”等方法。如“间接方法”就是指微指令中的部分bit合在一起，再经过一级译码才能产生控制信号（节省空间，但一次只能产生一个控制信号）；再如“多级”方法，微指令格式同上级指令格式一样，包括微操作码和微操作数字段。——译者注



图C-4 控制结构的微编程实现方法：结构图

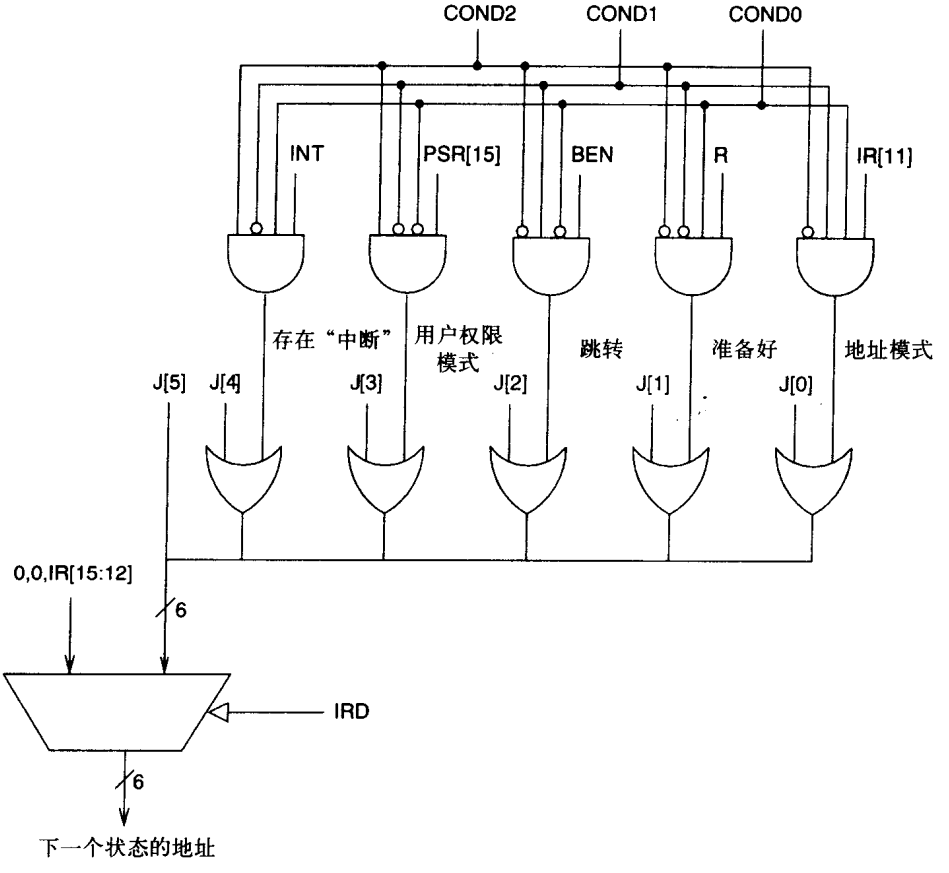
表C-2 微序列器控制信号

信号名	信号值	
J/6:		
COND/3:	COND0	;Unconditional
	COND1	;Memory Ready
	COND2	;Branch
	COND3	;Addressing Mode
	COND4	;Privilege Mode
	COND5	;Interrupt test
IRD/1:	NO,YES	

表C-2所示，是决定状态机下一个状态的10-bit控制信息，及其可能的取值。而图C-5是微序列器（microsequencer）的设计逻辑，微序列器的任务是基于当前的10-bit控制信息，确定状态机的下一个迁移状态，也即下一状态所对应的49位微指令的地址。

例如，图C-2状态机的“状态32”有16个可能的“下一状态”，具体迁移到其中的哪一个状态，完全取决于当前读入的指令。该状态的任务即“解码”（DECODE）工作（如第4章所述），进入该状态意味着控制器切换至对应的微指令（或新的一组控制信号）。如果当前微指令的IRD位（即控制信号）是1，则切换开关（MUX，如图C-5）的输出与左边6位输入（即由“00”和指令IR[15:12]这4位组装而成的6位）相连通。事实上，IR[15:12]对应的就是当前指令的操作码（opcode）字段（即LC-3的15个有效指令和1个未定义指令操作码），那么由它组成的控制存储地址

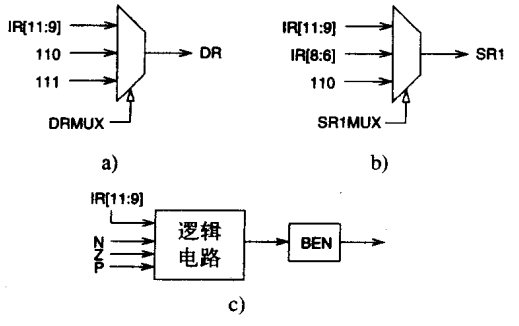
只有16种可能，规定了当前被处理的LC-3指令的操作码，控制存储器的下一个地址将是这16个地址之一，分别对应15个操作码和1个未被使用的操作码（即IR[15:12] = 1101）。换句话说，状态32之后的下一状态只可能是这16个状态之一。假设以ADD为例，由于ADD指令的IR[15:12]=0001，所以它从状态32跳转至的下一状态就是状态1。



图C-5 LC-3微序列器

假设由于种种原因，当前指令的IR[15:12] = 1101（即“无效指令”（unused opcode)），则微结构将跳转至状态13所对应的微指令，即认为它是一个“非法（illegal）指令”（具体细节描述，可参考C.6.3节）。

很多控制数据通路和微序列器的信号未在表C-1和C-2中列出，如DR、SR1、BEN、INT和R，它们需要额外的逻辑来生成，图C-6所示是生成信号DR、SR1和BEN的逻辑电路。



图C-6 控制信号逻辑的附加部件

INT信号是由外部设备提供的，处理器如果收到这个信号，就要中断正在执行的程序，转而处理中断（或称“服务于”该中断）。在微结构中，怎样妥善处理中断是个很重要的问题，有关中断机制的原理，参考第8章，有关微结构中相应的控制方式，参考C.6节。

R信号是一个同步信号，由内存提供。设计该信号的目的，是使得LC-3能正确操作内存。因为内存访问（读或写）所需的时钟周期数目不确定，而内存通过发送R（ready）信号通知内存访问结束。假设内存访问的延迟是5个周期，那么在MAR装入地址，然后向内存发出READ信号之后，需要等待5个周期之后，内存数据才能装入MDR（注意，微指令设置READ操作的实际方法是设置“MIO.EN/YES”和“R.W/RD[⊖]”，参见图C-3）。

回忆一下C.2节中的状态33，它的任务是取指令（FETCH）操作，也即通过内存访问获取指令。换句话说，状态33在转移至“状态35”之前，要连续执行5次。因为每个时钟周期都要有规律地触发一次状态机，所以在MDR获取来自MAR指定内存地址中的有效数据之前（即R信号到达之前），我们设计了让“状态33”循环执行的机制。状态35要做的事情，是将MDR的内容再次搬移到IR寄存器中。由于内存访问的延迟以及操作的异步性，我们将取指令操作分解成状态33和状态35这两个状态（而不是一个）。

R信号（Ready）的作用是确保内存访问操作的有序执行。因为只有内存自己知道需要多久才能完成一次访问（即5个时钟周期），所以R信号是由内存负责生成的。如图C-2所示，如果当前时钟周期内存读操作没有完成，则下一个状态仍然是状态33（地址100001）；如果完成了，则转入状态35（地址100011）。如图C-5所示，生成下一个状态地址的具体工作，是由微序列器完成的。

以状态33为例，在状态33中，微序列器10个控制位的取值如下：

```
IRD/0⊖     ; NO
COND/001   ; Memory Ready
J/100001
```

那么在此情况下，微序列器生成的下一个状态地址应该是什么呢？在状态33的前4个周期执行中，由于R信号都等于0，所以生成的下一个状态地址都是100001，即导致状态33被重复执行。而在第5个周期，如果R = 1，则下一个状态地址等于100011，即LC-3转移至状态35。注意，如图C-5所示，为了让R信号通过，COND0~2必须设置为001（即R是4个输入之一的那个与门）。

C.5 内存映射I/O

如第8章所述，LC-3通过内存映射I/O方式控制输入/输出，也就是说，采用与读写内存一样的数据搬移指令，来读写输入/输出设备。为此，LC-3为每个设备寄存器都分配了一个地址。有关设备的输入/输出操作是，在输入操作中，load指令的有效地址是输入设备的寄存器地址；输出操作中，store指令的有效地址是输出寄存器地址。例如，图C-2左下角状态25的操作，假设MAR中地址内容等于xFE02，即MDR内容来自KBDR，则load读入的数据内容就是最新的键盘输入字符。相反，如果MAR中的地址是一般性的内存地址，则MDR内容就由内存负责提供。

针对内存映射I/O的访问，如图C-2所示的状态机并不需要做什么特殊修改。但是，我们仍然

⊖ MIO.EN/YES和R.W/RD标识的含义是“MIO.EN信号 = YES”和“R.W信号 = RD”。由于之后的很多地方都采用这种标识方法，所以在此没有做更直观的标识改动，而是维持原书的标识方法。——译者注

⊖ 同样，IRD/0的意思是“IRD = 0”，J/100001则代表“J = 100001”。——译者注

需要设计一些逻辑，以判别什么时候内存访问，什么时候访问I/O设备寄存器[⊖]。这就是图C-3所示的地址控制逻辑所做的工作。

表C-3 地址控制逻辑的真值表

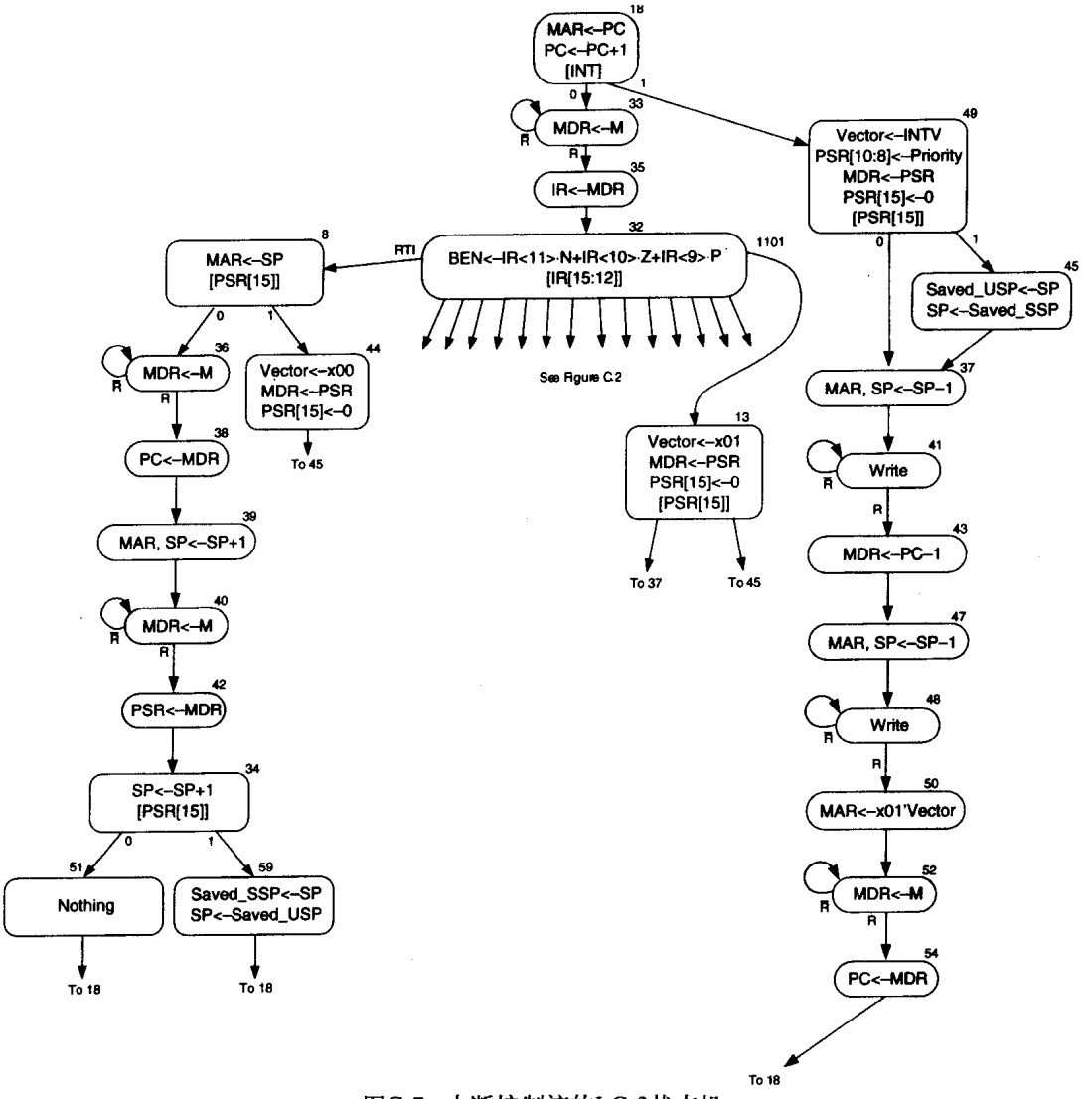
MAR	MIO.EN	R.W	MEM.EN	IN.MUX	LD.KBSR	LD.DSR	LD.DDR
xFE00	0	R	0	x	0	0	0
xFE00	0	W	0	x	0	0	0
xFE00	1	R	0	KBSR	0	0	0
xFE00	1	W	0	x	1	0	0
xFE02	0	R	0	x	0	0	0
xFE02	0	W	0	x	0	0	0
xFE02	1	R	0	KBDR	0	0	0
xFE02	1	W	0	x	0	0	0
xFE04	0	R	0	x	0	0	0
xFE04	0	W	0	x	0	0	0
xFE04	1	R	0	DSR	0	0	0
xFE04	1	W	0	x	0	1	0
xFE06	0	R	0	x	0	0	0
xFE06	0	W	0	x	0	0	0
xFE06	1	R	0	x	0	0	0
xFE06	1	W	0	x	0	0	1
other	0	R	0	x	0	0	0
other	0	W	0	x	0	0	0
other	1	R	1	mem	0	0	0
other	1	W	1	x	0	0	0

表C-3所示是“地址控制逻辑”的真值表，基于下述几点：（1）MAR；（2）当前周期访问对象是内存或I/O（MIO.EN/NO或YES）；（3）所需的是load还是store操作（R.W/Read或Write）的输入信号。注意，在内存映射方式下，MDR的数据来源可以是以下4个之一：内存、KBDR、KBSR和DSR，即地址控制逻辑通过控制INMUX开关选择输入源；同样，在内存映射方式下，MDR数据的写入对象包括：内存、KBSR、DDR和DSR，地址控制逻辑也提供了对应每个模块的使能信号（如MEM.EN、LD.XXX）。

C.6 中断和异常控制

有关LC-3状态机的最后一个内容是中断控制，即中断产生、中断返回（RTI指令）和异常优先问题（即两个异常同时产生的情况）。这里所说的异常，是指“特权模式冲突”和“非法指令”这两个异常。图C-7所示是相关的状态机描述，图C-8则是在补充了中断和异常处理模块之后的完整数据通路图（参考图C-3）。

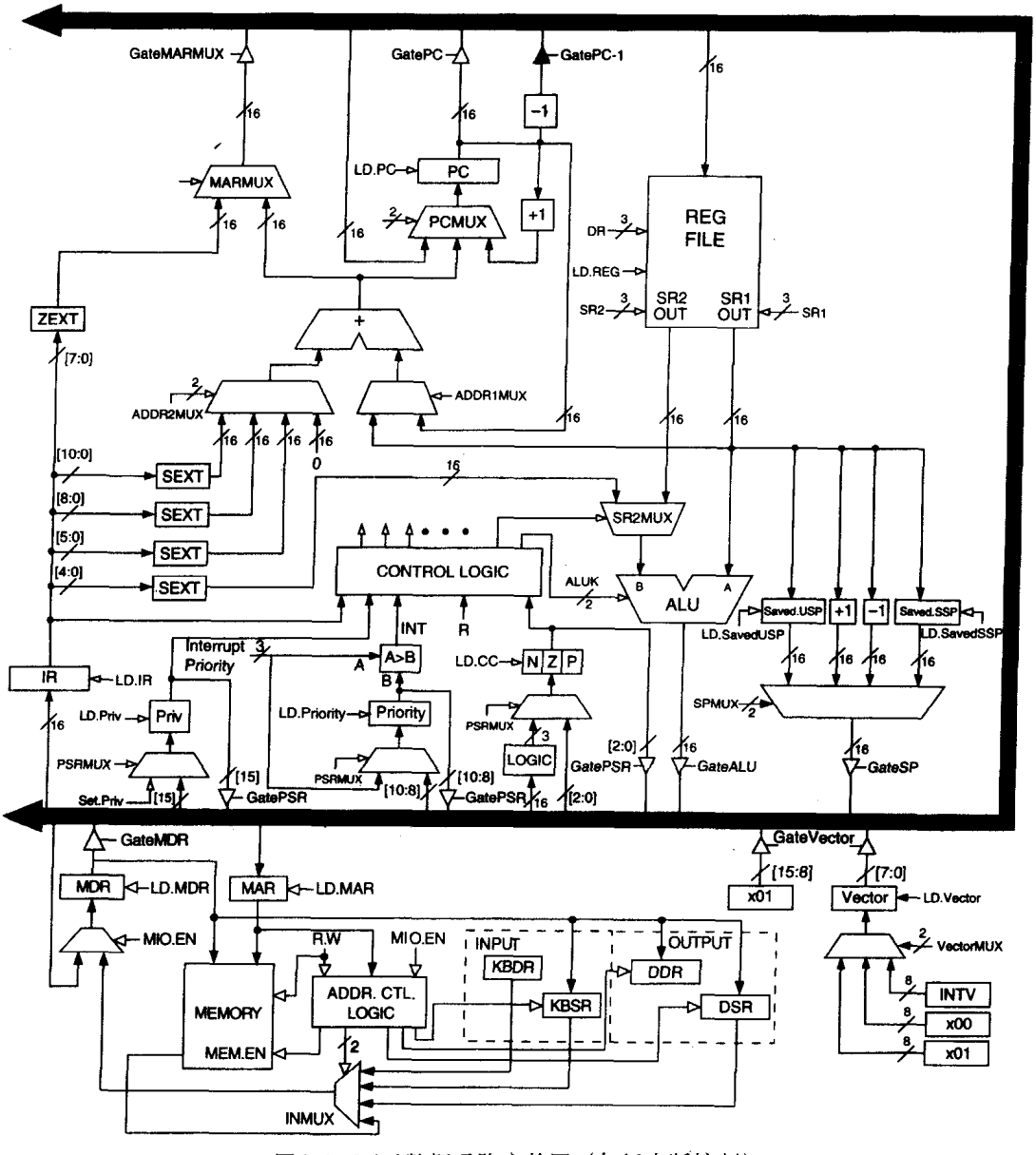
⊖ 请思考：为什么要判断当前访问操作的对象是内存还是I/O设备？不是已经有了不同的地址吗？答案是：“地址控制逻辑”（ADDR_CTRL_LOGIC）为连接在总线（黑色的粗线）上的每个独立的被访问设备或内存模块（而不是每个地址）都准备了一个独立的“使能”（EN）控制信号，也就是我们通常所说的片选（CS）或片使能（CE）信号，比如图中的MEM.EN信号。这是因为地址的解析工作是每个内存模块或设备独立完成的，但只有对应EN信号有效的情况下，该内存模块或设备才能真正“看到”来自总线的地址值，才能继续下一步的地址解析工作。注意，原理是相同的，但在此，LC-3的设计将总线和模块内部的逻辑放在一起做了统一描述，显得更加细致、直观。——译者注



图C-7 中断控制流的LC-3状态机

C.6.1 中断启动

程序执行过程中，外部事件（设备发出的）可以产生中断请求，从而造成正常指令执行的中止，即控制器临时转去处理中断。外部事件通过设置中断请求信号来请求中断，回顾第8章中的描述，如果设备中断请求的优先级高于正在执行程序的优先级，则INT信号有效且将相应的中断矢量装入INTV。微处理器对INT信号的响应就是“启动”（initiating）中断服务过程。即处理器切换至特权模式（supervisor mode），并将当前被中断进程的PSR和PC保存入特权堆栈（俗称“压栈”操作），然后将PC内容替换为中断服务程序的入口地址。注意，被保存的PSR中包含了当前程序（或进程）的权限模式PSR[15]、优先级PSR[10:8]以及条件码PSR[2:0]，这些信息很重要。当处理器结束中断处理后，恢复被中断程序时，必须将PSR中的这些信息恢复到正确的原值，所以需要保存这些信息。



图C-8 LC-3数据通路完整图（包括中断控制）

在LC-3微结构中，启动中断的方式正如图C-7⊖中的状态18所示的那样，在将PC内容装入MAR，PC增量之后，检测INT信号是否有效（如果INT有效，则启动中断处理）。

状态18是处理器检查中断的惟一状态。只在状态18检查中断的原因很简单：LC-3指令一旦开始执行，则让它执行完（即使中途INT信号已经有效）。如果处理器在指令执行中途也能处理中断（即在各个状态检查INT），则要求处理器还要具备保存/恢复指令执行的中间状态（很复杂）。因而，只在状态18检查INT。当前指令周期可以及早中止（其实指令还未被FETCH），并借助当前周期的时间，完成中断的启动工作。

另外，也仅在状态18下，控制信号（COND5）的取值为101（如表C-2所示），即允许INT值通过4输入与门（AND）参与下一状态的地址计算。换句话说，由于COND信号在其他状态下的取值

⊖ 原文中是图C-2（而不是图C-7）。——译者注

不可能是101，所以INT信号在其他状态下不起作用。

在状态18下，微序列器的10个控制位取值如下：

```
IRD/0      ; NO
COND/101   ; Test for interrupts
J/100001
```

如果此时INT = 1，则最左边的AND门（如图C-5所示）的输出为1，则下一状态的地址为110001（即状态49），而不是100001（即状态33）。状态机的中断启动过程由此开始（参见图C-7）。

状态49要完成的工作很多。其中包括：将PSR装入MDR（PSR中包含了被中断程序的特权模式、优先级和条件码等信息），以备将其压入特权堆栈；清除PSR[15]位，即表示将处理器切换到特权模式，因为中断服务程序必须运行在特权模式下；记录由中断设备提供的3-bit优先级和8-bit的中断矢量（INTV），将优先级值装入PSR[10:8]，将INTV值装入内部寄存器Vector；最后，处理器在压入PSR和PC之前，要测试一下原先的PSR[15]位，以判断栈指针R6指向的是哪一个栈（是特权栈还是用户栈）。

如果原先的PSR[15] = 0，表示处理器目前已处于特权模式了，即R6代表SSP（特权栈指针），因而处理器很快转入状态37[⊖]，并将被中断程序的PSR压入特权栈；如果PSR[15] = 1，表示被中断进程是在用户模式下，则必须先将USP（即R6当前值）保存至Saved_USP，而在转入状态37之前，必须保证R6的内容等于Saved_SSP。所有这些都是在状态45内完成的。

从状态49转移至状态37或状态45的控制流，是受控于微序列器的。微序列器的当前控制位取值如下所示：

```
IRD/0      ; NO
COND/101   ; Test PSR[15], privilege mode
J/100001
```

如果PSR[15] = 0，控制流转入状态37（100101）；如果PSR[15] = 1，则控制流转入状态45（101101）。

如果转入状态37，则R6（SSP）递减（为压栈做准备），并将新的SSP值装入MAR寄存器。

随后转入状态41，即执行压栈操作。先是设置内存写相关信号（MIO.EN/YES，R.W/WR），然后等待写操作完成（即R信号= 1）。至此，PSR被压入特权栈，随后控制流转入状态43。

状态43中，PC的内容被装入MDR。注意，在状态43中，MDR装入的是PC-1（而不是PC），为什么呢？因为在状态18，在中断指令执行之前，PC已被递增。而我们需要记录入栈的是中断指令所在的地址，而不是它的下一个指令。

状态47和48所做的事情同状态37和状态56[⊜]类似，只不过此时压入管理栈的内容是被中断进程的PC值。

中断启动的最后一个任务是，将中断服务程序的入口地址装入PC，即状态50、52和54的工作。该操作和TRAP服务程序加载PC的过程类似，发出INT请求的设备提供8-bit中断矢量INTV（同TRAP指令内嵌的8-bit矢量类似），如图C-8所示。

中断矢量表位于内存地址x0100~x01FF之间。在状态50中，将Vector中的矢量值（状态49装入的）与中断矢量表基址（x0100）相加，结果装入MAR；随后，在状态52中，等待内存数据读入（即中断矢量表中存放的、由MAR指向的中断服务程序入口地址值）；当R = 1时，读操作成功，此时MDR中包含的即是中断服务程序的入口地址；最后，在状态54中，将该地址装入PC。至此，

⊖ 原文是“状态37和状态44”，为避免误解，译文将“44”删除。状态44是中断服务程序返回时进入的状态，原文的意图是走“37至44”的这样一个状态转移线路。——译者注

⊜ 在状态机图C-2和图C-7中都没有发现标注为56的状态，但原文是说“状态37和状态56”。——译者注

中断启动工作完成。

值得一提的是，LC-3支持两个堆栈，这一点很重要。在特权模式和用户模式下，各有一个堆栈，且各有一个堆栈指针USP和SSP，两个指针值都存放在R6寄存器中。当权限模式从用户模式切换到特权模式时，R6装入Saved_SSP的内容；而从特权模式转入用户模式时，则装入Saved_USP的内容。同样，当权限模式改变时，R6的当前值必须被保存在合适的“Saved”堆栈指针中，以备权限模式改变回来时使用。

C.6.2 中断返回

中断服务程序通过中断返回（RTI）指令结束中断执行。RTI指令的任务是，将计算机的状态恢复到中断之前的状态，即恢复PSR（包含权限模式、优先级和条件码N、Z、P等信息）及PC的内容，这些都是事先在中断启动时压入堆栈的，它们从堆栈中，以与压入（push）过程相反的顺序被弹出（pop）。

RTI指令被解码（DECODE，状态32）后的第一个状态是状态8。在此，我们将特权栈的栈顶地址装入MAR，其中包含的是最后被压入，且之后一直没有被弹出过的内容，即中断启动时的PC值。另外，我们还将测试PSR[15]的内容。因为，RTI是一个仅在特权模式下才可以被执行的特权指令。如果PSR[15] = 0（即特权模式下），则继续执行RTI剩余的操作；否则，就认为是一个“权限模式异常”（priviledge mode exception）事件。

PSR[15] = 0；RTI继续执行

状态36和状态38负责PC内容的恢复工作。在状态36中，读内存操作。读操作完成之后，MDR包含的应该是指向被中断指令的地址值。状态38则负责将这个地址值装入PC寄存器。

状态39、40、42和34则负责PSR原先内容的恢复（即权限模式、优先级和条件码N、Z、P）。其中，在状态39中，特权栈指针增量（SSP+1），即指向PC弹栈后的栈顶，并将新的栈顶地址装入MAR；状态40等待内存读操作完成，读操作完成后，MDR包含的应该是原先的PSR值；状态42则将MDR内容装入PSR寄存器；最后，状态34进一步增量堆栈指针。

状态34的最后一个工作是，再次检查原先被中断程序的权限模式，目的是判断堆栈指针是否需要做转换操作。在状态34下，微序列器控制位的取值如下：

```
IRD/0      ; NO
COND/101   ; Test PSR[15], privilege mode
J/110011
```

如果PSR[15] = 0，则控制流转入状态51，即下一时钟周期中什么也不做；如果PSR[15] = 1，则转移至状态59，即将当前R6内容存入Saved_SSP，再将Saved_USP内容装入R6。之后，两个状态（51和59）都回到状态18，即继续下一条指令的处理。

PSR[15] = 1；权限模式异常处理

如果PSR[15] = 1，意味着处理器现在遇到了非法特权模式访问，即试图在用户模式下执行RTI这样的特权指令，这是“违法”的。

处理器将通过调用“权限模式异常（或非法权限访问，Privilege Mode Exception）”服务程序来处理这类事件。即将PSR和RTI指令地址压入特权堆栈，然后将处理“权限模式异常”服务程序的入口地址装入PC寄存器。这个过程和中断事件的处理流程非常相似。

首先，在状态44中执行三个功能：（1）将指向中断矢量表中“权限模式异常”表项的矢量值装入Vector寄存器，注意，这只是一个8-bit的矢量值（即x00）；（2）将引起该非法访问的程序的

PSR装入MDR；（3）将PSR[15]设置为0，即切换至特权模式（因为中断服务程序必须在特权模式下执行）。随后，处理器转入状态45，即中断启动流程中的中间状态点。

在随后的处理流程中，同中断启动流程相比，惟一的不同点体现在状态50，即将x01<Vector>装入MAR的操作。在设备中断处理的例子中，Vector寄存器中存放的是中断设备提供的INTV值（在状态49中装入的）；而在“权限模式异常”例子中，Vector的内容是x00（在状态44中装入的）。

还有两个细微的差别，主要体现在状态49的操作中。（1）如果是设备中断处理，则在状态49中，要修改当前执行优先级（priority），即将当前优先级修改为请求中断设备的优先级（即提升优先级），而在权限模式异常处理中，我们不修改优先级，即让异常服务程序执行在与引发异常的程序相同的优先级别下。（2）其次，中断处理时，要测试一下当前的权限模式（以决定是从状态49直接转入状态37，还是转入状态45），而在权限模式异常处理中，则不做测试（因为当前权限模式肯定是用户模式）。

C.6.3 非法操作码异常

在C.6节开始，我们就提到，LC-3指令集定义了两种异常：权限模式异常（Priviledge Mode Exception）和非法操作码异常（Illegal Opcode Exception）。所谓权限模式异常，如之前所描述的，即处理器试图在用户模式下执行RTI指令。而如果被执行指令的操作码字段（IR[15:12]）包含了一个“未定义的”操作码（如1101），则将引发“非法操作码”异常。

处理器处理“非法操作码”异常的方法同处理“权限模式异常”的方法相似。即将程序的PSR和PC压入特权堆栈，然后将“非法操作码”的异常服务程序入口地址装入PC。之后，服务程序将按照事先规定的操作来处理非法操作码（取决于操作系统的设计理念）。

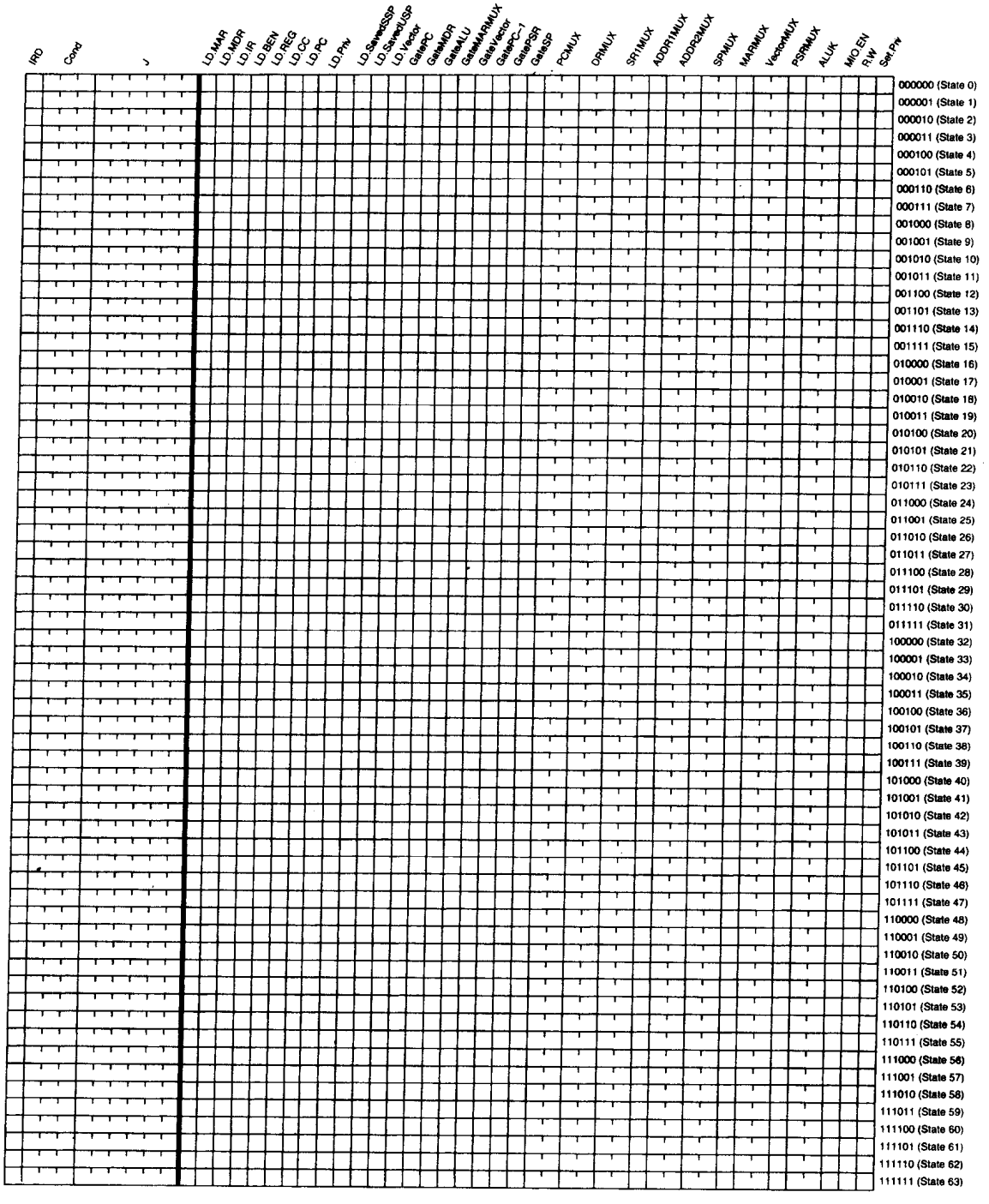
如果处理器进入状态13，则意味着一定是遇到非法操作码了。因为到达状态13的惟一途径是通过状态32（IR的译码状态）。状态13的操作任务，同状态49（中断启动）以及状态44（权限模式异常的启动）的操作非常相似。与它们相同的是，状态13将完成以下操作：（1）将矢量值（x01）装入Vector寄存器；（2）将权限模式设置为特权模式（PSR[15]=0）；（3）将PSR装入MDR，然后压入特权栈。

之后，状态13将与状态44和状态49各不相同。一方面如状态44，它不改变运行程序的优先级，因为它认为处理异常的紧迫性同执行紧迫性应该是一样的；另一方面则如状态49，即需要测试当前的权限模式，以判断当前是特权模式还是用户模式。因为在用户模式下，堆栈指针需要进行转换（如C.6.1节所述）。同样（如状态49），状态13之处将出现局部分支（microbranches），即转至状态37（如果栈指针已指向特权栈）或转至状态45（如果栈指针必须切换）。

再之后的过程，如状态37、41、43等等，则同中断处理（如C.6.1节所述）或权限模式异常处理（如C.6.2节所述）完全一样，即将PSR和PC压入特权堆栈，然后将服务程序入口地址装入PC中，最终启动异常服务程序。

C.7 控制存储器

图C-9是LC-3微编程的完整实现，即控制存储的全部内容。每个地址代表一个状态，其内容则对应当前状态下49个控制信号的取值。但事实上，这些表项的内容都是空着的，其目的是为了让读者享受一下填充的乐趣。标准答案请咨询你的老师。



图C-9 控制单元（微码）存储说明

附录D　C编程语言

D.1　概述

本附录是为C语言初学者提供的参考手册。它涵盖了C语言的各个重要方面，包括本书前面章节没有涉及的部分。本附录的另一个目的是作为C语言的快速参考手册，用户在编程遇到问题时，可以在本附录中找到C语言的各种特性。本附录后面的每个条目下，都会包含C语言某一方面的特性，并在需要的时候给出范例。

D.2　C程序设计规范

我们将通过分析和描述C语言的词法元素，以及C程序员常用的一些编程规范，来介绍C程序设计语言。

D.2.1　源文件

C程序设计规范建议，将程序分为两种类型的文件：源文件（由.c后缀标识）和头文件（由.h标识）。源文件中包含了一组相关函数的定义，比如，管理堆栈数据结构的函数，可能会在文件名为stack.c.的文件中定义。每个.c文件都将被编译成一个目标文件，这些目标文件再由链接程序链接成一个可执行文件。

D.2.2　头文件

头文件中包含函数、变量、结构体和类型的声明，以及一些预处理宏，但一般不包含任何C语句。头文件都是对应于源文件的，在头文件中声明的函数，将在源文件中定义。例如，如果源文件stdio.c中定义了函数printf、scanf、getchar和putchar，那么头文件stdio.h中将包含这些函数的声明。如果其他源程序需要调用stdio.c中定义的这些函数，那么只需要在这些源程序里包含头文件stdio.h就可以了。

D.2.3　注释

C语言里的注释起始于“/*”，终止于“*/”，它可以占据多行。但是注释不支持嵌套，大多数的编译器对于嵌套的注释都会产生一个语法错误。注释也不能位于一个字符串内部，否则注释将被认为是字符串的一部分而失效。虽然ANSI C标准中只允许使用“/*”和“*/”，但是现在很多的编译器都支持C++的注释方式(//)。

D.2.4　文字常量

C语言中的文字常量（literal）可以分为整数常量（integer literal）、浮点数常量（float literal）、字符常量(character literal)、字符串常量（string literal）、枚举常量（enumerator literal）等。文字常量可以用来初始化变量，也可以出现在表达式中，下面将分别给出例子。

1. 整数常量

整数类型的文字常量一般用十进制、八进制或十六进制形式表示。以0作为前缀的整型常量是

八进制形式，以0x作为前缀的整型常量是十六进制形式（因此，它可以由数字0~9，字母a~f或者A~F组成），不带前缀0和0x的整型常量是十进制形式。在整数常量前面加上“-”号，就表示相应的负整数。

在整数常量后面加上后缀“*L*”或“*l*”表示长整型数，加上后缀“*U*”或“*u*”表示无符号整数。请参考D.3.2节对长整型数和无符号整数的介绍。

下面例子中，前面三个表示相同的数字87，最后两个分别表示无符号类型和长整型的数字87：

```
87      /*  87 in decimal     */
0x57    /*  87 in hexadecimal */
0127    /*  87 in octal       */
-24     /* -24 in decimal     */
-024    /* -20 in octal       */
-0x24   /* -36 in hexadecimal */
87U
87L
```

2. 浮点数常量

浮点数常量由三个部分组成：整数部分、小数点和小数部分。其中小数部分和整数部分是可选的，但是这两部分必须至少存在一个。和整型数一样，在前面加上“-”号就表示相应的负数。下面给出浮点数表示的几个例子：

```
1.613123
.613123
1.        /* expresses the number 1.0 */
-.613123
```

浮点数常量还可以用指数形式表示，指数形式的浮点数由一个浮点常量、字符*e*(或*E*)和一个整数组成。*e*(或*E*)后面的整数表示浮点数的指数部分，它可以是负数，也可以没有。如果指数部分存在，则小数点可以省略。指数形式的浮点数表示的值，为前面的浮点常量乘以10的指数次幂，例如：

```
6.023e23        /* 6.023 * 10^23       */
454.323e-22     /* 454.323 * 10^(-22) */
5E13            /* 5.0 * 10^13         */
```

默认情况下，浮点数都是双精度类型。如果要定义单精度类型的浮点数，可以在浮点常量后面加上后缀*f*或*F*。要定义长双精度浮点数（见D.3.2节），则要在浮点常量后面加上“*l*”或“*L*”作为后缀。

3. 字符常量

字符常量由单引号括起来的单个字符表示，例如‘c’，这样就将该字符转换为计算机内部使用的ASCII码值。现在包括LC-3在内的大多数计算机，在内部都是使用ASCII码来表示字符。

表D-1列出了一些特殊的字符，这些字符不能用键盘的某个键表示出来，因此C语言通过一串特殊的字符序列来表示这些字符。表中最后两行给出了表示字符常量的另一种方法，通过转义字符‘\’将字符的ASCII码值转换为相应的字符，其中ASCII值既可以是八进制形式，也可以是十六进制形式。例如，字符‘S’的ASCII码值为83，因此‘S’既可以用‘\0123’也可以用‘\x53’来表示。

表D-1 C语言中的特殊字符

字符	顺序	字符	顺序
换行	\n	反斜杠	\\
横向制表符	\t	问号	\?
纵向制表符	\v	单引号	\'
退格	\b	双引号	\"
回车	\r	八进制数	\0nnn
进纸	\f	十六进制数	\xnnn
警告	\a		

4. 字符串常量

C语言中，用双引号括起来的字符序列表示字符串常量。字符串常量是char *类型，它存放在专门为常量保留的一段特殊内存区段里。字符串常量后面将自动加上“\0”来指示字符串结束。下面是字符串常量的两个例子：

```
char greeting[10] = "bon jour!";
printf("This is a string literal");
```

字符串常量可以用来初始化字符串，也可以用在任何可以使用char *类型的地方。例如，某个函数需要char *类型的参数，那么字符串常量就可以直接传递给这个函数。但要注意的是，字符串常量不能直接传递给字符数组。例如，下面这段代码在C语言里是不合法的：

```
char greeting [10];

greeting = "bon jour!";
```

5. 枚举常量

和枚举类型（见D.3.1节）相关的常量称为枚举器，也称为枚举常量。枚举常量都是整数类型，它们的值是在枚举类型定义的时候确定的。简言之，枚举常量就是一个表示某个特定整数值的符号而已。

D.2.5 程序格式

C语言的格式比较随意，程序员可以任意地在语句间加入空格、制表符、回车符和换行符。但是为了使程序更加清晰可读，程序员常常会在语句间使用缩进，在不相关代码间加入空行，对齐括号等，并且经常在难于理解的代码段旁边加上注释。注释对于那些对程序不熟悉的人来说非常重要。读者可以参考本书在C编程章节中给出的那些例程的格式。

D.2.6 关键字

下面列出了C语言中保留使用的符号——关键字。关键字在C语言里都有特殊的意义，例如用于类型定义、结构控制等，因此程序员不能使用关键字作为程序中的变量名或函数名。

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

D.3 类型

C语言里，表达式、函数、变量等都有相应的类型。以变量的类型为例，它指定了该变量能够表示什么样的值。假设kappa是一个int类型的变量，那么kappa代表的值将被编译器解释为一个有符号整数。C语言中的类型分为两类，一类是*基本数据类型*，即C语言本身提供的；另一类是用户*扩展的类型*，这种类型需要由程序员自己在程序里面定义。

D.3.1 基本数据类型

C语言本身定义了几种基本数据类型：int、float、double和char，所有C语言的实现都要能够支持这些基本数据类型。但是它们占据内存空间的大小和数值范围是与平台相关的。

1. int

int类型在C语言里指有符号整型数，大多数的计算机都使用32位的二进制数来表示int类型变

量。这样，一个整型数的取值范围就是−2 147 483 648 ~ +2 147 483 647。

2. float

float类型指单精度浮点数，大多数（不是所有的）计算机都是依照IEEE单精度浮点数标准来表示的，即用32位来表示一个单精度浮点数。这32位分为符号部分（1位）、指数部分（8位）和小数部分（23位）。想了解更多关于IEEE浮点数标准的读者，可以参考2.7.1节。

3. double

double类型指双精度浮点数，它和float类似，也使用IEEE的标准定义。虽然float和double定义的精度是与系统相关的，但是ANSI C标准规定，double的精度不能低于float。现在大多数的计算机都定义double为64位。

4. char

char类型的变量表示一个字符，在计算机内部，都是使用某种字符编码来表示字符。为了统一，现在大多数计算机都使用ASCII码（见附录E）来表示字符。char类型必须能够包含整个字符集，另外，C语言规定，short int类型长度不小于char类型。

总的来说，int和char（包含枚举类型）都是整数类型，而float和double都是浮点数类型。

5. 枚举类型

通过枚举类型，程序员可以在C语言里定义符号值类型的变量。比如我们想定义一种类型，该类型可以取以下四个符号值：Penguin、Riddler、CatWoman、Joker，我们可以通过枚举类型来实现：

```
/* Specifier */
enum villains { Penguin, Riddler, CatWoman, Joker };

/* Declaration */
enum villains badGuy;
```

这样，badGuy就是枚举类型villains的一个变量，它的取值范围就是在villains类型定义时列出来的四个值：Penguin、Riddler、CatWoman和Joker。枚举类型中定义的符号常量称为枚举常量（见D.2.4节），枚举常量其实就是整数值。

在枚举列表中，第一个枚举常量被自动赋值为0，接下来顺序赋值为1、2……在前面声明的villains中，Penguin的值为0，Riddler为1，CatWoman为2，Joker为3。程序员也可以显式地给枚举常量赋值，例如：

```
/* Specifier */
enum villains { Penguin = 3, Riddler, CatWoman, Joker };
```

这样，Penguin的值就为3，Ridder为4，CatWoman为5，Joker为6。

D.3.2 类型限定词

类型限定词可以用来修饰基本类型，并改变所修饰基本类型的某些特性，比如变量默认的数值范围。

1. signed和unsigned

int和char类型可以被signed和unsigned限定词修饰，默认情况下，int类型是有符号类型，但是char类型在默认情况下，却是和计算机平台相关的。

如果计算机用32位二进制数来表示有符号整数，那么有符号整数的取值范围是−2 147 483 648 ~ +2 147 483 647。在相同的机器上，无符号整数的取值范围就是0 ~ +4 294 967 295。

```
signed int c;      /* the signed modifier is redundant */
unsigned int d;

signed char j;     /* forces the char to be interpreted
                      as a signed value */

unsigned char k;   /* the char will be interpreted as an
                      unsigned value */
```

2. long 和 short

程序员可以使用long和short限定词来指定基本类型的物理长度，例如它们和整数类型搭配，可以定义短整型（short int）和长整型（long int）变量。

要注意的是，C语言并没有严格定义每种整型之间物理长度的差异，它仅仅规定了短整型长度小于或等于整型，整型长度小于或等于长整型。即：

```
sizeof(char)<=sizeof(short int)<=sizeof(int)<=sizeof(long int)
```

在支持64位数据类型的计算机系统里，long int可能是64位整数，而int是32位整数。特定系统的基本类型长度，可以在<limits.h>头文件中找到，类Unix的系统中，该文件存放在/usr/inclcude目录下。

下面列出了几个使用限定词来修饰整数类型的例子：

```
short int q;
long int p;
unsigned long int r;
```

long和short也可以用来限定双精度浮点数类型double，long double可以用来创建比double精度更高的浮点数。ANSI C规定，float长度小于或等于double，double长度小于或等于long double。

```
double x;
long double y;
```

3. 限定词const

在C程序执行过程中，不能被修改的变量可以使用const限定词来定义，例如：

```
const double pi = 3.14159;
```

在程序里不会改变的变量，程序员应该尽量使用const限定词，这样编译器可以对代码进行最大限度的优化。由于在程序执行过程中不能被修改，因此const限定的变量必须在定义的时候进行初始化。

D.3.3 存储类型

C语言中有两种存储类型：静态（static）和自动（automatic）。自动类型变量位于一个代码块（比如函数）中，一旦这个代码块执行完成，自动类型变量的值就丢失了。默认情况下，函数内部的自动类型变量的存储空间，都是在程序的执行栈中分配的（见14.3.1节）。

静态类型变量的值在整个程序的执行过程中都有效。在代码块的外部声明的全局变量是静态类型的，如果想在代码块的内部声明静态类型变量，可以在变量的前面加上static限定词。用static限定的变量，在声明它的函数执行完毕后仍然存在。例如：

```
int Count(int x)
{
  static int y;

  y++;
  printf("This function has been called %d times.", y);
}
```

静态类型变量y的值在函数Count执行完后仍然存在。实际上，静态类型变量都是在全局数据区中分配空间，因此变量的值在程序的执行过程中不会丢失，程序每次调用Count时，都会去更新y的值。

和自动类型变量不同，静态类型变量在定义时，都会自动被初始化为0。自动类型变量则要靠程序员来初始化。

C语言还提供了一个限定词register。register能够限定任何自动类型变量，表明该变量会经常访问，应该为它在寄存器中分配空间，以提高性能。但是编译器仅仅把register限定当做建议而已，是否分配寄存器空间，取决于编译器自己的分析，以及当前是否还有寄存器空间。

函数和变量都可以用extern限定词来修饰，表明该函数或变量的存储空间定义在另一个模块中，编译器会在生成可执行代码的过程中，将这些变量或函数链接在一起。

D.3.4 扩展类型

扩展类型是c语言基本类型的衍生。扩展类型包含指针、数组、结构体和联合体。结构体和联合体允许程序员创建自己定义的新类型，这些新类型由其他基本类型组成。

1. 数组

数组是由相同类型的数据组成的一个序列，它们在内存中顺序存放。即如果类型T数组的第一个元素位于内存地址X，那么该数组的下一个元素就位于内存地址X+sizeof(T)，依此类推。数组的元素都可以通过一个索引值来访问，索引值从0开始依次增加。数组list的第一个元素是list[0]。在声明数组的时候，数组的大小必须是一个整型常量表达式。

```
char string[100]; /* Declares array of 100 characters */
int  data[20];    /* Declares array of 20 integers */
```

数组通过整型表达式作为下标，来访问数组中的某个元素：

```
data[0]         /* Accesses first element of array data */
data[i + 3]     /* The variable i must be an integer */
string[x + y]   /* x and y must be integers */
```

编译器并不检查对数组的访问是否越界，因此程序员必须确保对数组的访问是合法的。例如，使用前面定义的string数组时，引用string[x+y]的下标x+y必须小于100，否则就会发生内存访问越界的错误。

2. 指针

指针是用来存储变量地址的数据类型，在标识符前面加上星号*就声明了指针变量，指针的类型表示指针可以指向的对象类型，例如

```
int *v;           /* v points to an integer */
```

C只允许对指针变量进行一组受限制的操作。例如指针可以出现在算术表达式中，可以用一个指针对同类型的另一个指针赋值，也可以为一个指针赋值0(赋值为0的指针称为空指针)。指针变量还可以加上或减去一个整数值，指针变量之间也可以进行比较或加减运算，这种情况在用指针操作数组时会经常碰到。除了这些之外，C不允许对指针变量进行其余的操作，除非进行强制类型转换。

3. 结构体

程序员可以使用结构体来实现类型的集合。结构体由一组元素组成，每个元素都有自己的类型。下面的代码声明了一个结构体：

```
struct tag_id {
  type1 member1;
  type2 member2;
  :
  :
  typeN memberN;
};
```

这个结构体由N个元素组成，member1是type1类型变量，member2是type2类型变量，memberN是typeN类型变量。组成结构体的元素可以是任何类型，包括其他程序员定义的扩展类型。

程序员在定义时可以为结构体指定一个标签，例如上面例子中的tag_id，这样就可以使用它来声明结构体变量：

```
struct tag_id x;
```

结构体由它的标签确定。例如一个程序里声明了许多结构体，这些结构体有相同的成员元素，而且元素的类型完全相同。但是，只要给它们定义的标签不同，这些结构体就不同。

结构体变量可以在结构体定义的同时进行声明，如下例所示，变量firstPoint在结构体point定义的时候进行声明，数组image则是在结构体定义完后，通过结构体标签point来声明。

```
struct point {
  int x;
  int y;
} firstPoint;


/* declares an array of structure type variables */
struct point image[100];
```

有兴趣的读者，可以参考19.2节关于结构体的介绍。

4. 联合体

结构体由许多各种类型的变量组成。联合体和结构体不同，在不同时刻，联合体只包含多个不同类型变量中的一个变量。例如，下面的代码声明了一个联合体变量joined：

```
union u_tag {
  int    ival;
  double fval;
  char   cval;
} joined;
```

变量joined是一组比特数据，这些比特可以是一个整数、浮点数或者字符数据。例如使用表达式joined.ival，联合体就是一个整数；如果使用joined.fval，联合体就是浮点数；如果使用joined.cval，联合体就成了一个字符变量。编译器将按照联合体中占用空间最大的类型，来为联合体分配空间。例如在上面的例子中，编译器将按照double类型的大小，为联合体分配存储空间。

D.3.5 typedef

C语言允许程序员用typedef为一个已存在的类型创建一个新的类型名，这在程序员定义自己的扩展类型时非常有用。使用typedef的一般形式如下：

```
typedef type name;
```

其中，type可以是任何基本类型、枚举类型或其他扩展类型，name是任意合法的标识符。这样就为类型type取了一个别名name，任何使用type的地方，都可以使用name来代替。一个好的类型名能包含该类型的一些额外信息，这样可以增强程序的可读性。下面是几个例子：

```
typedef enum {coffee, tea, water, soda} Beverage;
Beverage drink;     /* Declaration uses previous typedef */
typedef struct {
   int xCoord;
   int yCoord;
   int color;
} Pixel;

Pixel bitmap[1024*820]; /*Declares an array of pixels*/
```

D.4 声明

C语言里的一个对象（比如变量）就是一段有名字的内存段，C语言规定，对象在使用前必须先声明。对象的声明告诉编译器该对象的类型、名字、存储类别等，这样编译器才能为该对象分配空间，以及为该对象的访问产生正确的机器代码。

函数在使用前也必须先声明。函数的声明告诉编译器它返回值的类型、函数名、参数类型和参数的顺序。

D.4.1 变量声明

变量声明格式如下：

[存储类别] [类型限定] {类型} {标识符} [= 初始化值];

{}中的字段是必需的，[]中的字段可选。

*存储类别*可以是D.3.3中列出的任意存储类别限定词，比如static。

*类型限定*可以是D.3.2中列出的任意类型限定词。

*类型*可以是任何基本类型（int、char、float和double）、枚举类型或衍生类型（数组、指针、结构体、联合体）。

*标识符*是以字母或下划线开头的字母、数字、下划线的任意组合。C语言对标识符的长度没有限制，但是ANSI C编译器只使用最前面31个字符来区分变量的标识符，前面31个字符相同的变量，将被当做同一个变量。另外，C语言是区分大小写的，因此标识符sum和Sum是不同的。最后，标识符不能和关键字相同（见D.2.6节）。下面是合法标识符的例子：

```
blue
Blue1
Blue2
_blue_
bluE
primary_colors
primaryColors
```

只要是能在自动存储类型（见D.3.3节）变量赋值语句之前确定值的表达式，都可以作为自动存储类型变量的*初始化值*，但是只能使用常量表达式来初始化静态存储类型变量或外部变量。

C语言允许在同一行指定多个标识符（和初始化值）来创建多个同种类型的变量，这些变量有相同的存储类别和类型特征。

```
static long unsigned int k = 10UL;
register char l = 'Q';
int list[100];
struct node_type n;  /* Declares a structure variable */
```

变量可以在任何*代码块*的起始处声明（见D.6.2节），也可以在所有函数的外部声明，在代码块的起始处声明的变量，仅仅在该代码块中可见，而在所有函数外部声明的变量，在程序的各个部分均可见，因此这样的变量也称为*全局变量*。参考12.2.3节关于变量声明的详细介绍。

D.4.2 函数声明

函数声明告诉编译器，该函数返回值的类型和调用该函数需要传递的参数类型、参数个数和参数的顺序。函数声明格式如下：

{类型}{函数名}([参数1类型][,参数2类型]……[,参数N类型]);

其中{}中表示必须要的字段，[]中包含的是可选字段。

*类型*表示函数返回值的类型，它可以是基本类型，也可以是程序员自定义的扩展类型（数组除外），甚至是空类型，没有返回值的函数，必须将函数的返回值定义为void类型。

*函数名*可以是任何没有在程序中出现过的有效标识符。

紧跟在*函数名*后面括号里的字段，是函数参数的类型，它们用逗号分隔。每个参数类型后面可以跟变量，例如下面声明的函数可能是用于计算一个数组的平均值：

```
int Average(int numbers[], int howMany);
```

D.5 运算符

这一节描述C语言的运算符，这些运算符都是按照它们的用法来分类。

D.5.1 赋值运算符

C语言提供了多种赋值运算符，其中最基本的是“=”，赋值运算的顺序是从右到左。

下面是一个简单的赋值运算的标准形式：

{左表达式} = {右表达式}

左表达式必须是可以修改的对象，它不能为一个函数、一个const类型的变量或者一个数组（但可以是数组的一个元素）。左表达式也称为左值，它可以是一个结构体或者联合体变量。

当赋值运算执行完毕后，左表达式变量就具有了右表达式的值。在绝大多数赋值运算中，左表达式和右表达式都具有相同的类型，如果它们的类型不同，那么右操作数将转换为左操作数的类型后，再进行赋值运算。

除了“=”外，还有如下的一些赋值运算符：

+= -= *= /= %= &= |= ^= <<= >>=

这些赋值运算符可以看做是简单赋值运算符”=”和相应的算术运算符的合成。即A op= B等价于A = A op (B)。例如x += y和x = x+y的结果是相同的。

其他赋值运算符的例子，可以在12.3.2节中找到。

D.5.2 算术运算符

C语言支持下列基本的二元算术运算符：

+ - * / %

它们分别是加、减、乘、除和取模运算。这些运算符常常使用基本类型（int、double、float和char）作为操作数。如果操作数具有不同类型的值（比如一个浮点数加上一个整数），那么目标表达式将按照转换规则（见D.5.11节）进行转换。但是有一个例外，取模运算符的操作数，必须是一个整数（例如整数类型、字符类型、枚举类型等）。

加、减运算符还可以用指向数组元素的指针作为操作数，这时候，加、减运算也称为指针运算。例如，如果ptr是一个指针类型变量type*，那么表达式ptr+1就等价于ptr+sizeof(type)，即数组下一个元素的地址。

C语言还支持一元运算符+和-，在操作数前面加上负号“-”将得到该操作数的相反数。在操作数前面加上正号“+”即得到该操作数本身。C语言包含“+”仅仅是为了和“-”对应。

更多使用算术运算符的例子，可以参考12.3.3节。

D.5.3 位运算符

C语言支持位操作，下列是位操作的运算符：

& | ^ ~ << >>

位运算符只能以整数作为操作数，浮点数不能用于位操作。

运算符“<<”和“>>”分别表示左移和右移运算。左边的是要移动的操作数，右边的是移动的位数。ANSI C中，没有定义要移动的位数大于左边操作数位长度（比如把一个32位整数移动33位），以及要移动的位数是负数的情况，这两种情况下，得到的结果是不定的。

表D-2给出了这些位操作的例子，并给出了当操作数x＝186、y＝6时的运算结果。

表D-2 C语言中的位操作符

操作符	操作	例子	x=186, y=6
&	按位与	x&y	2
|	按位或	x|y	190
~	按位取反	~x	-187
^	按位异或	x^y	188
<<	左移	x<<y	11904
>>	右移	x>>y	2

D.5.4 逻辑运算符

在C语言里，我们可以使用逻辑运算符将多个逻辑子句连接起来，构成一个逻辑表达式。例如，我们想测试条件A和条件B是否均为真，我们只需要用逻辑“与”将A和B连接起来即可。

逻辑“与”是二元运算符。当且仅当逻辑“与”连接的所有子句均为“真”时，整个表达式才为“真”。

逻辑“或”也是二元运算符。当且仅当逻辑“或”连接的所有子句均为“假”时，整个表达式才为“假”。

逻辑“非”是一元运算符。当操作数是逻辑“真”时，表达式为“假”，反之，表达式为“真”。

逻辑“与”和“或”是“短路”(short-circuit) 操作符，即如果逻辑表达式左边的子句能够决定判断条件，那么表达式右边的子句就不再计算，这将带来“副作用”。例如表达式“x||y++”中，如果x非0，则y++子句将不会执行。

表D-3列出了逻辑运算符的用法。为了和位运算符比较，仍取操作数x＝186，y＝6。

表D-3 C语言中的逻辑操作符

操作符	操 作	例子	x=186, y=6
&&	逻辑与	x&&y	1
||	逻辑或	x||y	1
!	逻辑非	!x	0

D.5.5 关系运算符

下列运算符：

```
==      !=      >      >=      <      <=
```

是C语言的关系运算符，它们用来比较左操作数和右操作数之间的关系，比如“相等”、“不相等”、“大于”等。如果关系成立，则返回逻辑“真”，否则返回逻辑“假”。当左操作数和右操作数类型不匹配时，会发生类型转换，详见D.5.11节。C语言允许指针作为关系运算符的操作数，但是只有在两个指针指向相同对象时才有意义。

D.5.6 增量/减量操作符

C语言使用符号“++”和“--”作为增量和减量操作符，它们都是一元运算符，对操作数做“加1”或“减1”运算。“++”和“--”可以放在操作数的前面，也可以放在操作数的后面。

作为前缀时，操作数先进行“加1”或“减1”运算，然后才将操作数的值作为表达式的值。例如下面代码执行后：

```
int x = 4;
int y;

y = ++x;
```

x和y都等于5；

作为后缀时，先以操作数的值作为表达式的值，然后再对操作数进行“加1”或“减1”运算，例如下面代码运行后：

```
int x = 4;
int y;

y = x++;
```

x等于5，而y等于4。

和加减运算类似，增量和减量运算也可以用指针作为操作数，D.5.2给出了一个例子。

D.5.7 条件表达式

C语言里的条件表达式如下：

```
{表达式 A} ? {表达式 B}: {表达式 C}
```

当表达式A为逻辑“真”时，整个条件表达式的值等于表达式B的值，反之，如果表达式A为逻辑“假”时，整个条件表达式的值就等于表达式C的值。例如，下面的代码段：

```
w = x ? y : z;
```

表达式x ? y : z的值取决于x的值，如果x非0，则y将对w进行赋值，否则z将对w进行赋值。

条件表达式也具有逻辑运算符的“短路”特性，表达式B和表达式C能否得到执行，取决于表达式A的值。如果x非0，表达式z将不会执行，如果x的值为0，表达式y就不会执行。关于“短路”特性，见D.5.4节。

D.5.8 指针、数组和结构体

这里介绍和扩展类型一起使用的与地址相关的操作符。

1. 取址运算符

取址运算符是&，它返回操作数的内存地址，这就要求操作数必须是一个内存对象，例如变量、数组元素或者结构体成员。

2. 间接引用运算符

间接引用运算符是取址运算符的补充。它以地址作为操作数，返回该地址指向的对象。例如下面的代码：

```
int *p;
int x = 5;

p = &x;
*p = *p + 1;
```

表达式*p返回x的值。当*p出现在赋值运算符的左边时，它可以看做一个左值，否则它仅仅返回它指向变量的值。

3. 数组引用

C语言里，括号“[]”里的表达式用于指定数组的下标，“[]”操作符通常和数组名一起使用，来指明访问的数组元素。下面是一个数组引用的例子：

```
int x;
int list [100];

x = list[x + 10];
```

4. 结构体和联合体引用

C语言定义了两个操作符，用于引用结构体和联合体的成员。第一个操作符是“.”，结构体或联合体变量可以直接使用它来引用自己的成员。下面是一个“.”操作符的例子：

```
struct pointType {
  int x;
  int y;
};
typedef pointType Point;

Point pixel;

pixel.x = 3;
pixel.y = pixel.x + 10;
```

pixel是一个结构体变量，可以通过“.”操作符直接引用它的成员。

第二个操作符是“->”，结构体或联合体的指针，可以通过它来直接引用该指针指向变量的成员。下面是一个“->”操作符的例子：

```
Point pixel;
Point *ptr;

ptr = &pixel;
ptr->x = ptr->x + 1;
```

其中，ptr是指向结构体变量pixel的指针。

D.5.9 sizeof操作符

操作符sizeof返回存储一个指定类型的对象所需要的字节数。例如，sizeof(int)返回一个整型数需要占据的字节数，当操作数是一个数组时，sizeof返回数组的大小，如下例所示：

```
int list[45];

struct example_type {
  int    valueA;
  int    valueB;
  double valueC;
};
typedef struct example_type Example;

...

sizeA = sizeof(list);      /* 45 * sizeof(int)  */
sizeB = sizeof(Example);   /* Size of structure */
```

D.5.10 运算顺序

C语言表达式的运算是一个顺序进行的过程。如果表达式中有括号，则首先计算括号内的表达式。如果没有括号，则按照C语言定义的运算符优先级，先计算优先级高的表达式。如果表达式中出现相同优先级的运算符，则按照C语言定义的关联顺序从左到右或从右到左依次计算。

表D-4给出了C语言运算符的优先级和关联顺序，最高优先级的运算符在表的顶部。

表D-4 操作符的优先关系，括号内是该操作符的描述

优先级	关联顺序	操 作 符
1(最高)	从左到右	()(函数调用) []数组下标) . ->
2	从右到左	++ -- (后缀)
3	从右到左	++ --(前缀)
4	从右到左	*(间接引用) &(取地址) +(一元运算) -(一元运算) ~ ! sizeof
5	从右到左	强制类型转换
6	从左到右	*(乘法) / %
7	从左到右	+(加法) -(减法)
8	从左到右	<< >>
9	从左到右	< > <= >=
10	从左到右	== !=
11	从左到右	&(按位与)
12	从左到右	^
13	从左到右	|
14	从左到右	&&
15	从左到右	||
16	从左到右	?:(条件运算符)
17(最低)	从右到左	= += - = *=等

D.5.11 类型转换

考虑一个使用操作符*op*的表达式：

```
A op B;
```

该表达式运算结果的类型取决于：（1）操作数*A*、*B*的类型；（2）操作符*op*的特性。

假如操作数*A*和*B*有相同的类型，而且操作符*op*能够以该类型变量作为操作数，则该表达式运算结果的类型是由操作符*op*决定的。

当表达式中的变量类型各不相同时，C语言将对操作数进行类型转换。一般而言，类型转换都是将占比特数较少的类型转换为占比特数较多的类型，比如将整数类型转换为浮点数类型。假如上面例子中*A*为浮点数类型，而*B*为整数类型，操作结果将是浮点数类型。像字符类型、枚举类型、整数类型等整数类型的值，都转换为整数类型（或者无符号整型，这是和编译器相关的）。下面给出一些类型转换的例子：

```
char   i;
int    j;
float  x;
double y;

i * j    /* This expression is an integer */
j + 1    /* This expression is an integer */
j + 1.0  /* This expression is a float    */
i + 1.0  /* This expression is a float    */
x + y    /* This expression is a double   */
i + j + x + y        /* This is a double */
```

上面（2）提到，一些操作符需要特定的类型作为它的操作数，或者得到的结果是特定的类型。例如取模运算符%只能用整数类型作为它的操作数，这样，其他类型的操作数(比如char类型)都将转换为整数类型。浮点数类型不能转换为整数类型，因此用浮点数作为%的操作数将产生编译错误。

如果将一个浮点数转换为整数类型（这种情况不会发生在自动类型转换时，但是可能发生在强制类型转换时，强制类型转换在下一节介绍），它的小数部分将会丢失，而且如果整数类型不能表示该浮点数的整数部分时，转换的结果是不定的。

强制类型转换

程序员可以通过*强制类型转换*（type casting）来控制类型的转换过程。强制类型转换的一般形式如下：

```
(新类型)表达式
```

这里的表达式将按照前面介绍的转换规则转换为*新的类型*，下面是一个例子：

```
j = (int)x + y;   /*结果的双精度浮点类型将转换为整数类型*/
```

D.6 表达式和语句

C语言里，程序完成的任务是由函数体内的表达式和语句来描述的。

D.6.1 表达式

表达式是由常量、变量、操作符和函数调用组成的序列，该序列能够产生一个特定类型的值。表达式按照操作符的运算优先级和关联顺序进行求值（见D.5.10节），求得值的类型取决于表达式的各个元素和类型转换规则（见D.5.11节）。如果表达式的每个元素都是整数类型，那么这个表达式就是整数类型。下面是几个表达式的例子：

```
a * a + b * b
a++ - c / 3
a <= 4
q || integrate(x)
```

D.6.2 语句

C语言里，简单的语句就是以分号（;）结束的表达式，通常，语句除了对表达式求值外，还有其他副作用，比如对变量值进行修改。一条语句执行完成后，下面一条语句就继续执行。当函数的最后一条语句执行完成后，整个函数就结束了。下面是两个简单语句的例子：

```
c = a * a + b * b;  /* Two simple statements */
b = a++ - c / 3;
```

复合语句（或代码块）由包含在括号“{}”里的几条相关的语句组成。从语法角度上讲，复合语句和简单语句是一样的，它们可以互相替代。例如，下面的复合语句和上面的两条语句是一样的：

```
{                       /* One compound statement */
 c = a * a + b * b;
 b = a++ - c / 3;
}
```

D.7 控制结构

程序员可以通过控制结构来实现程序的条件执行或循环执行。

D.7.1 if语句

if语句的格式如下：

```
if  (expression)
    statement
```

expression可以是任何基本类型、枚举类型或指针类型的表达式。当expression的值不为0时，statement就会执行。statement可以是简单语句，也可以是复合语句。例如：

```
if (x < 0)
   a = b + c; /* Executes if x is less than zero */
```

在13.2.1节，有更多关于if语句的例子。

D.7.2 if-else语句

if-else语句的格式如下：

```
if (expression)
   statement1
else
   statement2
```

同样，expression可以是任何基本类型、枚举类型或指针类型。当expression的值不为0时，就执行statement1语句，否则执行statement2语句。statement1和statement2都可以是简单语句或复合语句。

```
if  (x < 0)
    a = b + c; /* Executes if x is less than zero */
else
    a = b - c; /* Otherwise, this is executed.  */
```

在13.2.2节，有关于if-else语句的更多例子。

D.7.3 switch语句

switch语句的格式如下：

```
switch(expression)  {
case const-expr1:
     statement1A
     statement1B
```

```
        :

    case const-expr2:
        statement2A
        statement2B
        :

        :
        :

    case const-exprN:
        statementNA
        statementNB
        :

    }
```

switch语句由一个整型表达式（见D.3.1节）和紧随其后的复合语句（虽然并不要求是复合语句，但几乎始终是复合语句）组成，这个复合语句中有一个或多个case标志，每个标志后面紧跟一个整型常量表达式（例如上面例子中的const-expr1、const-expr2、const-exprN）。在一个switch语句中，每个case标志后面的常量表达式，必须是互不相同的。

当执行到一个switch语句时，首先计算switch后面的表达式expression。如果该表达式的值和某个case标志后面的常量表达式的值匹配，程序就跳转到该case标志后面的语句开始执行。

default是一个特殊的标志，它用来匹配那些和其他case标志都不匹配的情况。如果switch语句中没有default标志，而且表达式的值和每一个case都不匹配时，switch中的语句都不会执行。

下面是使用switch语句的一个例子。程序执行时如果遇到break，将跳出switch语句。想了解break的更多信息，可以参考D.7.7节。

```
char k;

k = getchar();
switch (k)  {
case '+':
  a = b + c;
  break;       /* break causes control to leave switch */

case '-':
  a = b - c;
  break;

case '*':
  a = b * c;
  break;

case '/':
  a = b / c;
  break;
}
```

13.5.1节有更多关于switch语句的例子。

D.7.4 while语句

while语句的格式如下：

```
while (expression)
  statement
```

while是一个循环控制结构。当表达式expression的值不为0时，就执行语句statement，当statement执行完成后，并不执行后面的语句，而是重新跳转到while语句的开始，并计算expression的值。这个过程重复执行，直到表达式expression的值为0为止，然后才开始执行while后面的一条语句。statement可以是一个简单语句，也可以是复合语句。

下面的例子中，while语句将循环100次。

```
x = 0;
while (x < 100) {
   printf("x = %d\n", x);
   x = x + 1;
}
```

13.3.1节有更多关于while语句的例子。

D.7.5 for语句

for语句的格式如下：

```
for (initializer; term-expr; reinitializer)
  statement
```

for语句也是一个循环结构。表达式initializer在进入循环之前执行，而且仅执行一次。表达式term-expr则是在每次循环开始时执行，如果计算term-expr得到非0值，那么继续执行循环体，否则跳出循环。语句statement在每次循环时都会执行，当statement执行完成后，将执行语句reinitializer。这个过程一直重复，直到term-expr表达式的值等于0时为止。

下面for语句的例子，将循环执行100次：

```
for (x = 0; x < 100; X++) {
   printf("x = %d\n", x);
}
```

13.3.2节有更多关于for循环的例子。

D.7.6 do-while语句

do-while语句的格式如下：

```
do
  statement
while (expression);
```

do-while语句和while语句非常相似。当遇到do-while语句时，首先执行循环体statement，然后计算表达式expression的值。如果计算得到的值非0，那么继续进行下一次循环，否则跳出循环。do-while语句的特点是至少执行一次循环。

下面是循环100次的do-while语句的例子：

```
x = 0;
do {
   printf("x = %d\n", x);
   x = x + 1;
}
while (x < 100);
```

13.3.3节有更多关于do-while语句的例子。

D.7.7 break语句

break语句的格式如下：

```
break;
```

break语句只能用在循环或switch结构中，如果程序在执行过程中遇到break语句，程序将跳出最近一次循环。

下面例子中，break语句将结束for循环：

```
for (x = 0; x < 100; x++) {
  :
  :
  if (error)
      break;
  :
  :
}
```

13.5.2节有关于break语句的更多例子。

D.7.8 continue语句

continue语句的格式如下：

```
continue;
```

continue语句只能用在循环结构中，它立即结束当前的一次循环，并回到循环的开始，重新计算循环条件，以决定是否进行下一次循环。在for语句中，如果遇到continue语句，程序在进行下一次循环前，会先执行reinitializer语句。下面给出一个for循环的例子：

```
for (x = 0; x < 100; x++)  {
  :
  :
  if (skip)
      continue;
  :
  :
}
```

在这段代码中，如果continue语句执行，表达式x++也将会执行，然后再计算循环条件，决定是否进行下一次循环。

13.5.2节有关于continue语句的更多例子。

D.7.9 return语句

return语句的格式如下：

```
return expression;
```

return语句将结束当前函数的执行，并返回到调用函数。当函数执行完最后一条语句后，也会隐式地插入一条return语句，返回到调用程序。

return后面的表达式是函数的返回值，它将转换为函数的返回类型。当函数需要返回值，但函数并没有任何产生返回值的语句时，函数返回的值是不定的。下面是return语句的一个例子：

```
return x+y;
```

D.8 预处理

源代码在编译前，还包含一个修改的过程，这个过程是由程序员控制的，在C语言里称为预处理。

最常用的预处理包括宏替换和头文件包含。宏替换允许程序员用一个文本序列替换另一个序列，头文件包含是指将头文件的内容包含到源文件中。本节后面将详细描述这两个预处理指令。

预处理指令不是C语句，因此它们不需要使用分号来表示一条指令的结束。

D.8.1 宏替换

预处理器将用#define预处理指令将指定的字符序列替换为另一个。例如：

```
#define   A      B
```

这样，源程序里和A匹配的字符序列将被B替换。但是需要注意，只有独立的字符序列才会被替换。例如标识符APPLE中的A不会被B替换，双引号里的字符串常量，例如“A”，也不会被替换。

替换文本包括后续的整行，如果替换文本很长，可以用”\”符号将多行连接在一起。

宏可以带参数，其参数在宏后面的括号里指定。例如：

```
#define REMAINDER(X, Y)          ((X)%(Y))
```

源代码里的宏REMAINDER后面，需要两个表达式作为宏的参数，例如：

```
valueC = REMAINDER(valueA, valueB+15);
```

预处理器将使用#define指定的替换文本来替换该宏，并且使用源代码里出现的参数来替换X和Y。前面的代码在预处理后将生成下面的代码：

```
valueC = ((valueA) % (valueB+15));
```

注意，宏定义时X和Y外面的括号不能省略。否则，由于%比+的优先级高，上面的替换将会产生错误。

带参数的宏在形式上和函数调用很相似，但是宏不会引入一般函数调用时所产生的开销。

D.8.2 头文件包含

预编译指令#include的作用是将一个头文件的内容插入到源代码文件中。C程序的头文件一般都只包含一系列的宏定义和函数的声明。

#include预处理指令有下面两种形式：

```
#include <stdio.h>
#include "program.h"
```

第一种情况是在尖括号“<>”里指明包含的头文件。这种情况下，编译器仅仅搜索环境变量中指定的头文件目录，来查找该头文件，这个头文件目录一般是系统定义好的，在该目录下包含了许多系统相关和其他库相关的头文件。第二种情况是在双引号内指明包含的头文件，这种情况下，编译器将首先搜索包含源文件的目录，如果没有找到，则继续搜索环境变量中定义好的头文件目录。

D.9 标准库函数

ANSI C标准定义了150多个标准库函数，程序可以使用它们来完成不同的任务（例如输入/输出和动态分配内存）。这些函数都是标准的，因此，使用这些函数的程序可以很容易地从一个ANSI C平台移植到另一个ANSI C平台。这一节将介绍一些有用的库函数。

D.9.1 输入/输出函数

使用标准输入/输出函数的代码，必须包含<stdio.h>头文件，下面是几个例子：

1. getchar函数

这个函数的声明如下：

```
int getchar(void)
```

getchar从标准输入设备stdin中读取下一个输入的字符，并且以该字符的值作为函数的返回值。

getchar函数和LC-3中的输入TRAP很相似，只是它不在屏幕上打印输入提示信息。

许多计算机系统都使用缓冲I/O来实现getchar，即操作系统将缓存键盘输入的字符（假定标准输入是键盘）。一旦遇到回车键，缓存的字符序列就加载到输入流中。

2. putchar函数

putchar函数的声明如下：

```
void putchar(int c);
```

函数putchar和LC-3中的TRAP OUT指令相似，它带一个整型数作为参数，将该值代表的字符输出到标准输出流中。

如果标准输出设备是显示器，那么我们就可以在显示器上看到输出的字符。但是，现代计算机都使用了缓冲技术，字符要等到缓冲区清空（flushed）后才能在屏幕上出现，比如在遇到回车字符后。

3. scanf函数

scanf函数的声明如下：

```
int scanf(const char * formatstring,*ptr1,...);
```

scanf以一个格式串和一组指针作为它的参数，这个格式串包含了一系列转换格式声明，这些转换格式声明将控制对输入流的解释。例如，转换格式声明“%d”控制scanf将输入流中下一个非空字符串解释为一个十进制整型数，这个字符串首先转换为整数，然后将其赋值给下一个指针指向的变量。表D-5列出了scanf使用的转换格式声明，格式串后面的指针参数个数，应该和格式串中的转换格式声明个数相同。scanf的返回值表示正确赋值的变量个数。

表D-5 scanf的转换格式声明

转换格式声明	参 数 类 型
%d	有符号十进制数
%i	十进制、八进制（0开头）或十六进制（0x开头）
%o	八进制数
%x	十六进制数
%u	无符号整数
%c	字符
%s	非空字符串
%f %e	浮点数

4. printf函数

printf函数的声明如下：

```
int printf(const char * formatString,...);
```

printf的功能是将格式化字符串写到标准输出流中。如果格式串format string中包含一个转换格式声明，那么printf会将后面的参数解释为相应类型值后填充到输出流中。例如，转换格式声明“%d”告诉printf，将后面的参数解释为一个十进制整数，然后将这个整数插入到输出流中。表D-6列出了printf使用的一些转换格式声明，一般而言，格式串后面的参数，应该和格式串中的转换格式声明相对应。printf的返回值表示写到标准输出流中的字符个数，如果遇到错误，则返回一个负值。

表D-6 printf的转换格式声明

转换格式声明	参 数 类 型	转换格式声明	参 数 类 型
%d %i	有符号整数	%s	字符串
%o	八进制数	%f	十进制形式的浮点数
%x %X	十六进制数	%e %E	指数形式的浮点数
%u	无符号整数	%p	指针
%c	字符		

D.9.2 字符串函数

C标准库里包含了大约15个与字符串相关的标准库函数。要使用这些函数，必须在源文件里包含<string.h>头文件。这一节我们讨论其中的两个函数。

1. strcmp函数

strcmp函数的声明如下：

```
int strcmp(char * stringA, char * stringB);
```

这个函数用来比较两个字符串，如果字符串stringA和stringB相等，函数返回0。如果stringA大于stringB，返回一个正数。如果stringA小于stringB，则返回一个负数。这里的比较是基于字典排序的，即在字典里靠后的字符串大于它前面的字符串。

2. strcpy函数

strcpy的函数声明如下：

```
char * strcpy(char * stringA, char * stringB);
```

这个函数将字符串stringB复制给stringA(包括字符串结束符)，如果没有遇到错误，它将返回一个指向stringA的指针。

D.9.3 数学函数

C标准库函数中包含了一些常用的数值运算，使用它们时需要包含<math.h>头文件。在这一节中，我们列出了一部分关于双精度浮点运算的函数，它们都以double类型作为参数，返回类型也是double:

```
double sin(double x);   /* sine of x, expressed in radians    */
double cos(double x);   /* cosine of x, expressed in radians  */
double tan(double x);   /* tan of x, expressed in radians     */
double exp(double x);   /* exponential function,  e^x         */
double log(double x);   /* natural log of x                   */
double sqrt(double x);  /* square root of x                   */
double pow(double x, double y)   /* x^y -- x to the y power   */
```

D.9.4 其他有用的函数

除了上面列出的函数外，C标准库中还包含其他一些非常重要的函数，例如内存分配、数据转换、排序等。使用这些函数时，只要包含头文件<stdlib.h>即可。

1. malloc函数

正如19.3节所述，malloc函数分配一个固定大小的内存块，它的声明如下：

```
void * malloc(size_t size);
```

函数参数size表示要分配内存的大小，以字节为单位。参数类型是size_t，和操作符sizeof的返回值一样（一般情况下，都是unsigned int类型）。如果分配成功，则返回一个指向该内存块的指针，若分配失败，则返回一个空指针NULL。

2. free函数

free的函数声明如下：

```
void free(void * ptr);
```

函数free释放前面分配的堆（heap）内存区，这个内存区的地址由指针ptr指定。但是需要注意，ptr指向的内存区段必须是前面动态分配的，否则会发生错误。

3. rand和srand函数

C标准库中还包括一个产生随机序列的函数，称为rand。但是它并不能产生真正的随机序列，我们称之为伪随机序列。该序列由初始的种子（seed）值决定，当种子改变时，序列也就改变了。例如，当种子是10时，得到的随机序列将是相同的，但是这个序列和别的种子产生的序列是不同的。

函数rand的声明如下：

```
int rand(void)
```

该函数返回一个伪随机整数，这个随机数的范围在0和RAND_MAX之间。RAND_MAX是系统定义的宏，该值最小为32 767。

为了设置伪随机序列的种子，我们可以使用srand函数。该函数的声明如下：

```
void srand(unsigned int seed);
```

附录E　常用表

E.1　常用数字前缀

表E-1　数字前缀

数值	二进制近似值	前缀	缩写	前缀的起源
10^{24}	2^{80}	yotta	Y	从eight的希腊语okto中转化而来
10^{21}	2^{70}	zetta	Z	从seven的希腊语hepta转化而来
10^{18}	2^{60}	exa	E	从six的希腊语hexa中转化而来
10^{15}	2^{50}	peta	P	从five的希腊语pente中转化而来
10^{12}	2^{40}	tera	T	从monster的希腊语teras中转化而来
10^{9}	2^{30}	giga	G	从giant的希腊语gigas转化而来
10^{6}	2^{20}	mega	M	从large的希腊语megas转化而来
10^{3}	2^{10}	kilo	K	从thousand的希腊语chilioi中转化而来
10^{-3}		milli	m	从thousand的拉丁语milli中转化而来
10^{-6}		micro	μ	从small的希腊语mikros中转化而来
10^{-9}		nano	n	从dwarf的希腊语nanos中转化而来
10^{-12}		pico	p	从little的西班牙语pico中转化而来
10^{-15}		femto	f	从15的德语和挪威语femten中转化而来
10^{-18}		atto	a	从18的德语atten中转化而来
10^{-21}		zepto	z	从seven的希腊语hepta中转化而来
10^{-24}		yocto	y	从eight的希腊语okto中转化而来

E.2　标准ASCII编码

表E-2　标准ASCII 编码

字符	十进制	十六进制	字符	十进制	十六进制	字符	十进制	十六进制	字符	十进制	十六进制
nul	0	00	dle	16	10	sp	32	20	0	48	30
soh	1	01	dc1	17	11	!	33	21	1	49	31
stx	2	02	dc2	18	12	"	34	22	2	50	32
etx	3	03	dc3	19	13	#	35	23	3	51	33
eot	4	04	dc4	20	14	$	36	24	4	52	34
enq	5	05	nak	21	15	%	37	25	5	53	35
ack	6	06	syn	22	16	&	38	26	6	54	36
bel	7	07	etb	23	17	'	39	27	7	55	37
bs	8	08	can	24	18	(	40	28	8	56	38
ht	9	09	em	25	19	)	41	29	9	57	39
if	10	0A	sub	26	1A	*	42	2A	:	58	3A
vt	11	0B	esc	27	1B	+	43	2B	;	59	3B
ff	12	0C	fs	28	1C	'	44	2C	<	60	3C
cr	13	0D	gs	29	1D	−	45	2D	=	61	3D
so	14	0E	rs	30	1E	.	46	2E	>	62	3E
si	15	0F	us	31	1F	/	47	2F	?	63	3F

（续）

字符	十进制	十六进制	字符	十进制	十六进制	字符	十进制	十六进制	字符	十进制	十六进制
@	64	40	P	80	50	`	96	60	p	112	70
A	65	41	Q	81	51	a	97	61	q	113	71
B	66	42	R	82	52	b	98	62	r	114	72
C	67	43	S	83	53	c	99	63	s	115	73
D	68	44	T	84	54	d	100	64	t	116	74
E	69	45	U	85	55	e	101	65	u	117	75
F	70	46	V	86	56	f	102	66	v	118	76
G	71	47	W	87	57	g	103	67	w	119	77
H	72	48	X	88	58	h	104	68	x	120	78
I	73	49	Y	89	59	i	105	69	y	121	79
J	74	4A	Z	90	5A	j	106	6A	z	122	7A
K	75	4B	[	91	5B	k	107	6B	{	123	7B
L	76	4C	\	92	5C	l	108	6C	|	124	7C
M	77	4D	]	93	5D	m	109	6D	}	125	7D
N	78	4E	^	94	5E	n	110	6E	~	126	7E
O	79	4F	–	95	5F	o	111	6F	del	127	7F

E.3 二进制表示范围

表E-3 二进制的表示范围

数值	转化为十进制	近似值
2^1	2	–
2^2	4	–
2^3	8	–
2^4	16	–
2^5	32	–
2^6	64	–
2^7	128	–
2^8	256	–
2^9	512	–
2^{10}	1 024	1K
2^{11}	2 048	2K
2^{12}	4 096	4K
2^{13}	8 192	8K
2^{14}	16 384	16K
2^{15}	32 768	32K
2^{16}	65 536	64K
2^{17}	131 072	128K
2^{18}	262 144	256K
2^{19}	544 288	512K
2^{20}	1 048 576	1M
2^{30}	1 073 741 824	1G
2^{32}	4 294 976 296	4G