计 算 机 科 学 丛 书

原书第4版

计算机系统

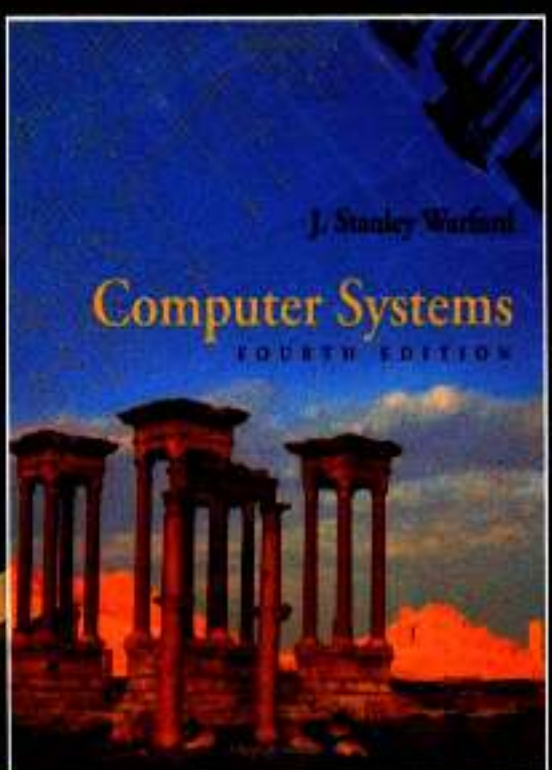
核心概念及软硬件实现

[美] J. 斯坦利·沃法德（J. Stanley Warford）
钍珀代因大学 著

龚奕利
武汉大学 译

Computer Systems

Fourth Edition



机械工业出版社
China Machine Press

计算机系统 核心概念及软硬件实现 原书第4版

Computer Systems Fourth Edition

本书清晰、详细、一步一步地介绍了计算机组成、汇编语言和计算机体系结构中的核心概念。它指导学生用自上而下的方法探索计算机系统各个抽象层次的方方面面，通过说明不同的抽象层次和其他层次之间的关系，本书帮助学生将计算机系统及其组成部分看作一个统一的概念。

本书基于Pep/8汇编器和模拟器，用来讲授经典马·诺依曼机器的基本知识。Pep/8现在包括新的符号跟踪特性，能够在学生单步跟踪程序时，实时显示全局变量和运行时栈的情况。

作者Warford教授从教30余年，他贯穿全书强调了掌握基本计算机概念的重要性，是理解当前和未来技术的基础，他还强调了解决问题能力的重要性。本书覆盖了ACM-IEEE计算机科学课程体系指导意见中体系结构和组成原理中所有的核心概念。

作者简介

J. 斯坦利·沃法德（J. Stanley Warford）现为美国佩珀代因大学（Pepperdine University）大学计算机科学系教授。Warford教授在进入学术界前是航空工程师，他任教30余年，曾担任佩珀代因大学计算机科学系主任。他从伦斯勒理工学院获得硕士学位，从加州大学洛杉矶分校获得博士学位。作为杰出的教学成员，Warford教授获得了Luckman奖。

译者简介

龚奕利 本科毕业于武汉大学，在中国科学院计算技术研究所获得博士学位，曾在美国印第安纳大学从事博士后工作和美国密歇根大学从事访问学者工作，现为武汉大学计算机学院副教授。主要研究方向为高性能计算和分布式系统，包括云计算和广域文件系统。翻译过《深入理解计算机系统》（第一、二版）等计算机专业书籍。

JONES & BARTLETT LEARNING

ISBN 978-7-111-50783-3

9 787111 507833 >

定价：79.00元

图书在版编目（CIP）数据

计算机系统：核心概念及软硬件实现（原书第4版）/（美）沃尔福德（Warford, J. S.）著；袁春风等译. —北京：机械工业出版社，2015.7
（计算机科学丛书）
书名原文：Computer Systems, Fourth Edition

ISBN 978-7-111-50783-3

Ⅰ. 计… Ⅱ. ①沃… ②袁… Ⅲ. 计算机系统 Ⅳ. TP30

中国版本图书馆CIP数据核字（2015）第150951号

本书版权登记号：图字：01-2013-5990

Original English language edition published by Jones & Bartlett Learning, LLC, 5 Wall Street, Burlington, MA 01803

J. Stanley Warford: Computer Systems, Fourth Edition (ISBN 978-0-7637-7144-7).

Copyright © 2010 by Jones & Bartlett Learning, LLC.

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2015 by China Machine Press.

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

本书基于虚构的计算机Pep/8，简明、详细、循序渐进地介绍了计算机组成、汇编语言和计算机体系结构中的核心思想，贯穿7个抽象层次组织内容，详细介绍了计算机系统的6层视图：高级语言层、汇编层、操作系统层、指令集架构层、微代码层和逻辑门层。本书有完整的程序示例，理论和实践相结合，实质和深度相结合，提供了有益的见解，使读者对计算机系统有深入理解。

本书可作为高等院校计算机科学专业本科生的教材，也可作为相关专业人员学习计算机基础知识的参考书。

出版发行：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：路晓迪　曲　熠　　责任校对：殷虹翠
印　刷：三河市宏图印务有限公司　　版　次：2015年7月第1版第1次印刷
开　本：185mm×260mm　1/16　　印　张：31
书　号：ISBN 978-7-111-50783-3　　定　价：79.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

版权所有·侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光／邹晓东

文艺复兴以来，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；也正是这样的优势，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，不仅奠划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

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

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

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，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，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

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

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

华章科技图书出版中心

It has been several years since the publication of the Fourth Edition of *Computer Systems*.

During that time computer technology has continued to advance. The natural question for this translation is, Why is *Computer Systems* still current after these years since its English publication? There are two reasons why it is current and will remain current for many years.

First, *Computer Systems* is still current because it teaches the fundamentals of computer systems that have been constant for decades. All computer systems consist of software and hardware, and all commercial hardware systems are based on the ubiquitous von Neumann cycle.

On the software side, all high-order languages manipulate data with iterative and recursive algorithms. All languages must be translated to lower level machine language to be executed or interpreted. On the hardware side, all physical machines are combinational and sequential circuits built from logic gates. No matter what the technology, the design principles behind these software and hardware systems do not change.

Second, *Computer Systems* is still current because it teaches the above design principles with the Pep/8 virtual machine. The Pep/8 machine will always be current because it is not subject to obsolescence, but rather is built to teach the fundamentals of computer systems that do not change. Specifically, it presents a computer system as seven levels of abstraction:

- Applications
- High-order languages
- Assembly
- Operating system
- Instruction set architecture
- Microcode
- Logic gate

The book illustrates these levels of abstraction using C++ as the high-order language and Pep/8 assembly language and machine language at the lower levels. The strength of this approach is that the central concepts of computer science are taught without getting entangled in the many irrelevant details that often accompany courses that focus on current technology rather than computing fundamentals. Students who learn the fundamentals are better equipped to master whatever new technology they will encounter in their future computing careers than they would be if they studied only the technology that is current when they are students.

本书第4版完成已经有几年的时间了，这期间计算机技术持续发展。要出版中文版，一个很自然的问题就是：为什么英文版出版多年之后本书的内容仍然没有过时呢？它现在没有过时，今后许多年也不会过时，原因有两点。

首先，本书讲授计算机系统的基础知识，这些基础知识数十年都没有改变过。所有的计算机系统都是由软件和硬件组成的，所有的商用硬件系统都是基于普适的冯·诺依曼周期。

从软件方面来说，所有的高级语言都以迭代或递归算法来处理数据。所有的语言都必须翻译成更低级的机器语言才能执行或解释。从硬件方面来说，所有的物理机器都是由逻辑门构成的组合或时序电路。无论技术如何发展，这些软件和硬件系统背后的设计原理没有改变。

其次，本书以Pep/8虚拟机为例来讲授上述设计原理。Pep/8机器不会过时，因为它不受时间限制，设计它的初衷就是讲授计算机系统中不会变化的基础知识。具体来说，它展现的是计算机系统在7个抽象层次上的样子：

- 应用层
- 高级语言层
- 汇编层
- 操作系统层
- 指令集架构层
- 微代码层
- 逻辑门层

本书在描述各个抽象层次时使用C++作为高级语言，在较低层上使用Pep/8汇编语言和机器语言。这种方法的好处就是讲授计算机科学的核心概念，而又不会纠结于无关的细节，很多把重点放在当前技术而不是计算基础的课程通常都会有这种问题。比起只学习当前技术的学生，学习了基础知识的学生在以后的工作中能更好地掌握遇到的新技术。

J. Stanley Warford

译者序

Computer Systems，Fourth Edition

本书第1版出版于2010年。在不断感叹计算机技术发展太快的时候，2010年的内容会不会太“老”太“旧”了呢？答案显然是否定的，尤其在读完了整本书之后。本书展示了计算机系统的7个抽象层次：应用层、高级语言层、汇编层、操作系统层、指令集架构层、微代码层和逻辑门层。本书的特色之一就是着眼于计算机软件和硬件系统背后的设计原理，而这些原理数年来都未曾改变过。而且，去除那些眼花缭乱的新技术的表象，能够更好地看清和理解系统的本质。

本书的另一个特色是建立了一个虚构的计算机系统Pep/8，借助于这个示例系统，能够让读者/学生更具体地了解计算机系统中的各个组成部分，进而了解核心概念，而不是通过抽象的描述学习抽象的概念；同时又不必拘泥于现实系统的实现细节。本书覆盖广泛，但又重点突出，强调了硬件及其相关软件的实现。本书文字简洁明了，是非常合适的计算机系统入门教材。

与所有计算机书籍的翻译一样，翻译过程中充满了艰难的术语选择，因为我们越来越习惯于在日常技术工作中使用英文术语，有时候使用它的中文翻译反而显得有些陌生和别扭。比如token，中文一般译作“语言符号”，但是实际上它有终结符字符串的含义，如果翻译成中文，很容易失去这些意义。所以我选择保留英文术语，相信不会影响读者阅读或增加阅读难度。

在此感谢王文杰帮助我一起讨论翻译中遇到的问题，还要感谢机械工业出版社华章公司的编辑们给了我很多理解和支持，使得本书得以完成。

在翻译过程中，我尽量做到认真仔细，但还是难以避免出现错误和不尽如人意的地方。在此欢迎广大读者批评指正，我也会把勘误表及时在网上更新，便于大家阅读。

吴英杰

2015年5月于安娜堡

本书清晰、详细、循序渐进地展示了计算机组成、汇编语言和计算机体系结构中的核心思想。本书的很大一部分是建立在一个虚构的计算机 Pep/8 基础上的，用它来讲解经典的冯·诺依曼机器的基本概念。这种方法的好处是能够讲解计算机科学的核心概念，而又不必拘泥于此类课程中常见的许多不相关的细节。这种方法还能鼓励学生思考计算机科学底层的原理。本书的范围也比较广泛，重点强调了与硬件及其相关软件的处理有关而少有提及的计算机科学主题。

内容摘要

计算机运行在一些抽象层上，在高级抽象层上编程只是一部分。基于图 1 的层次结构，本书展示了计算机系统的一个统一的概念。



图 1 典型计算机系统的层次结构

对应于图 1 的 7 层，本书也分为 7 个部分：

App7 层　应用层
HOL6 层　高级语言层
ISA3 层　指令集架构层
Asmb5 层　汇编层
OS4 层　操作系统层
LG1 层　逻辑门层
Mc2 层　微代码层

本书主要是按照从上到下、从最高层到最低层的顺序来书写。ISA3 层在 Asmb5 层之前以及 LG1 层在 Mc2 层之前讲解是出于教学的目的。在这两种情况下，暂时同相反的从下至上的方法来讲解更自然，有了低层的构造模块就很容易完成上层的构建。

App7 层　App7 层是单独一章，介绍了应用程序。本章展示了抽象层次的概念，建立本书剩下部分的框架。还介绍了一些关系数据库的概念，作为典型计算机应用的例子。同时，还假设学生对文字编辑器或文字处理器有一定的经验。

HOL6 层　HOL6 层也是一章，复习了 C++ 编程语言。本章假设学生具有某种命令语言的经验，不一定是 C++，可以是 Java 或 C。书中避免了 C++ 的高级特性，包括面向对象的概念。如果有必要，教师可以把 C++ 例子翻译成其他 HOL6 层的语言。

本章着重介绍了 C++ 内存模型，包括全局变量和局部变量、函数参数以及动态分配的变量。也介绍了递归的问题，因为它依赖于运行时栈上的内存分配机制。还相当详细地解释了函数调用的内存分配过程，因为本书后面还会在较低抽象层次上分析这个机制。

ISA3 层　ISA3 层是指令集架构层，包括两章，描述了一个用于说明计算机概念的虚构的 Pep/8 计算机。Pep/8 是经典的冯·诺依曼机器。CPU 包含一个累加器、一个变址寄存器、一个程序计数器、一个栈指针和一个指令寄存器。有 8 种寻址方式：立即数、直接、间接、栈相对、栈相对间接、变址、栈变址和栈变址间接。在模拟的只读存储器（ROM）中，Pep/8 操作系统能从学生的文本文件中装入和执行十六进制格式的程序。学生可以在 Pep/8

模拟器上运行小程序，学习不会改变内存值的 ROM 存储指令。

学生能学习到位层的信息表示和计算机组成的知识。因为本书的中心主题是层次之间的关系，所以有关 Pep/8 的章节展示了 ASCII 表示（ISA3 层）和类型为 char 的 C++ 变量（HOL6 层）之间的关系。还展示了补码表示（ISA3 层）和类型为 int 的 C++ 变量（HOL6）之间的关系。

Asmb5 层 Asmb5 层是汇编层，书中介绍了汇编器的概念（汇编器是汇编层和机器层之间的翻译器），还介绍了 Asmb5 层的符号和符号表。

这里是统一的方法派上用场的地方。第 5 章和第 6 章中的编译器是高级语言到汇编语言的翻译器。前面，学生学习了一种具体的 HOL6 层语言 C++ 和一种具体的冯·诺依曼机器 Pep/8。接下来的章节将继续介绍层次之间的关系，讲述下面这样一些对应关系：(a) HOL6 层的赋值语句和 Asmb5 层的装入 / 存储指令；(b) HOL6 层的循环和 if 语句与 Asmb5 层的分支指令；(c) HOL6 层的数组和 Asmb5 层的变址寻址；(d) HOL6 层的过程调用和 Asmb5 层的运行时栈；(e) HOL6 层的函数和过程参数与 Asmb5 层的栈相对寻址；(f) HOL6 层的 switch 语句和 Asmb5 层的转移表；(g) HOL6 层的指针和 Asmb5 层的地址。

统一方法之美就在于可以在较低层次上实现 C++ 章节中的例子。例如，第 2 章的递归例子中描述的运行时栈直接对应于 Pep/8 主存中的硬件栈。学生可以用手动方式直接在两层之间翻译，以便更好地理解编译的过程。

这种方法为讨论计算机科学中的核心问题提供了一种很自然的环境。例如，本书介绍了 HOL6 层的结构化编程，可以和 Asmb5 层的非结构化编程的可能性进行对比。书中讨论了 goto 争议、结构化编程 / 效率之间的折中，给出了两个层次上语言的实际例子。

第 7 章向学生介绍了计算机科学理论。既然学生对如何把高级语言翻译成汇编语言已经有了感性的认识，那么我们就提出所有计算中最基本的问题：什么是能够被自动化的？这里介绍理论是很自然的，因为学生现在知道了编译器（自动化翻译器）必须做什么。他们通过识别 C++ 和 Pep/8 汇编语言的语言符号来学习语法分析和有限状态机——确定性的和非确定性的。本章包括两种小语言之间的自动翻译器，说明了词法分析、语法分析和代码生成。词法分析器是有限状态机的实现。还有什么比这样更自然的介绍理论的方法呢？

OS4 层 OS4 层讲述操作系统，分为两章。第 8 章讲述进程管理，包括两节，一节讲装载器，一节讲陷阱处理程序，说明了 Pep/8 操作系统的概念。有 5 条指令具有产生软件陷阱的未实现操作码。操作系统把用户正在运行的进程的进程控制块存储在系统栈上，中断服务例程解释该指令，通过具体实现一个挂起进程来强化操作系统中运行和等待进程的经典状态转移图。本章结尾描述了并发进程和死锁。第 9 章描述存储管理，包括主存和磁盘存储器。

LG1 层 LG1 层用两章来介绍组合电路和时序电路。从布尔代数的定理开始，第 10 章重点介绍计算机科学的数学基础的重要性，展示布尔代数和逻辑门之间的关系，然后介绍一些常见的 SSI 和 MSI 逻辑设备，包括 Pep/8 ALU 的完整的逻辑设计。第 11 章通过介绍时序电路的状态转移图，描述有限状态机的基本概念。最后描述常见的计算机子系统，比如双向总线、内存芯片和双端口存储器体。

Mc2 层 第 12 章描述 Pep/8 CPU 的微程序设计控制区，给出了一些示例指令和寻址方式的控制序列，还提供了有关其他指令和寻址方式的大量练习。本章还介绍了装入 / 存储体系结构的概念，对比了 MIPS 的 RISC 机器和 Pep/8 CISC 机器。最后描述了高速缓存、流水线、动态分支预测和超标量机器，介绍了一些性能问题。

在课程中的使用

本书覆盖的内容广泛，教师在设计课程时可以省略一些内容。第 1 ～ 5 章可以看作核心，第 6 ～ 12 章可以有所取舍。

本书第 1 ～ 5 章必须顺序讲授，第 6 章和第 7 章可以按任意顺序讲授。我通常会省略第 6 章而直接讲第 7 章，开始一个大的软件项目——为 Pep/8 汇编语言的一个子集写一个汇编器，这样学生在一学期中有足够的时间完成它。第 11 章显然依赖于第 10 章，但是它们都不依赖于第 9 章，所以第 9 章可以省略。图 2 是一个章节依赖图，图中总结了可以省略那些章节。

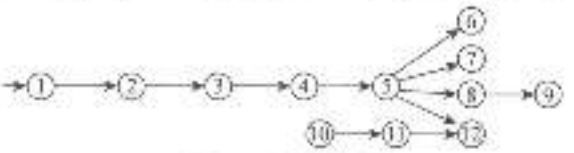


图 2　章节依赖图

辅助资料

下面列出的辅助资料可以从出版社网站获得：

http://www.jbpub.com/catalog/9780763771447/

Pep/8 汇编器和模拟器　Pep/8 机器在 MS Windows、Mac OS X 和 UNIX/Linux 系统上都有。汇编器的特性包括：

- 集成的文字编辑器；
- 在源代码中发现错误的地方插入红色字体的错误消息；
- 对学生友好的、十六进制格式的机器语言目标代码；
- 能够直接以机器语言编写代码，跳过汇编器；
- 能够重定义触发同步陷阱的未实现操作码的助记符。

模拟器的特性包括：

- 模拟的 ROM，装入指令不能修改；
- 在模拟的 ROM 中烧入了一个小的操作系统，包括一个装载器和一个陷阱处理程序；
- 一个集成的调试器，允许设置断点、单步执行、CPU 跟踪和内存跟踪；
- 用户定义的、从无限循环恢复的语句执行计数的上限；
- 能够通过为未实现操作码设计新的陷阱处理程序来修改操作系统。

Pep/8 CPU 模拟器　CPU 模拟器，有 MS Windows、Mac OS X 和 UNIX/Linux 系统版本，可以用在计算机组成课程中。CPU 模拟器的特性包括：

- 颜色编码的展示通路，根据控制信号跟踪复用器的数据流；
- 操作的单周期模式，用 GUI 输入每个控制信号，立即展示信号的效果；
- 操作的多周期模式，学生可以在集成的文字编辑器中编写 Mc2 微代码序列并执行它们以便实现 ISA3 指令。

课程课件　每章有 50 ～ 125 页的课程幻灯片，有 Keynote 和 PDF 格式。幻灯片包括课本中所有的图和总结信息，通常以标号的形式给出。不过其中没有太多的例子，给教师展示示例和教师指导讨论留出了空间。

考试题目　提供有一组考试题目，包括参考信息，例如 ASCII 表、指令集表等，供考试和自学之用。这些对用本书作为教材的教师开放。

数字电路实验　有一组 6 个数字电路实验，能够让学生在物理实验电路板上亲身体验。这些实验说明了第 10 章和第 11 章的组合和时序设备，使用许多本书中没讲到的电路。学生

可以自学实际的电子设计和实现概念。这些超出了本书的讲述范围，它们可以按照书中讨论的主题顺序，从组合电路开始，然后是时序电路和 ALU。

答案手册 附录中有部分练习的答案。剩下练习的答案对用本书作为教材的教师开放。出于安全原因，答案直接从出版社获取。相关信息请联系 Jones 和 Bartlett Publishers Representative，电话 1-800-832-0034。

第 4 版所做的修改

第 4 版对第 3 版做了大量修改，包括使用了 Pep/8，它是对 Pep/7 架构的彻底重新设计。前几版的用户已经接受了 Pep/8 的教学特性，第 4 版中还是保留了 Pep/8 架构。本版的每一章都有改进，下面只列出了其中一些主要的：

- *改进了 C++ 回顾*——扩展了第 3 版中引入的 C++ 内存模型，更系统地从头开始描述。内存分配图更现实，与主函数一致，显示了对主函数返回地址和返回值的分配。重命名了所有以前命名为 i 的变量。当程序翻译成 Pep/8 汇编语言后容易有误会，Pep/8 汇编语言用字母 i 表示立即数寻址。
- *改进了字符编码的内容*——讲述了 Unicode 字符集，代替了 EBCDIC。
- *跟踪标签*——Pep/8 汇编器和模拟器包括一个新的符号跟踪特性，当用户单步跟踪程序时，能够实时显示全局变量和运行时栈。使用这个新特性要求程序员在某些汇编语言语句的注释字段放置跟踪标签。翻译器将忽略这些标签，但是调试器将使用它们。跟踪标签的一个巨大好处是迫使程序员做好文件说明。要使用调试器，学生必须在注释字段精确说明哪些变量要在运行时栈上分配以及分配的顺序。汇编器验证分配的字节数是否与变量列表要求的字节数匹配。跟踪标签的文件说明优点很大，现在这个版本中描述了跟踪标签语法，书中和答案手册中的每个汇编语言程序中都包括了跟踪标签。
- *改进了语言翻译的内容*——前面版本中，第 7 章讲述的语言翻译原理假设学生没有面向对象知识。本版假设学生已经学习过基本的面向对象设计原理，展示的语法分析程序使用了对象组合、继承和多态调度以及 UML 图。
- *新的项目问题*——这一版本有两个项目问题，一个是新的，在第 6 章中，要求写一个 Pep/8 机器模拟器；另一个是改进的，在第 7 章中，要求写一个 Pep/8 汇编器。这两个项目要求开发上千行代码的程序，它们都有多个部分，每一个都往应用程序中增加了更多的功能。项目的目的有两个：1）让学生获得编写较复杂程序的经验；2）增强对这门课程问题域中计算机系统概念的理解。
- *改进了 RAID 的内容*——这个版本介绍了更广泛的 RAID 磁盘系统内容，扩展了 RAID 等级 01 和 10 的区别，增加了新的图和新的量化分析习题。
- *改进了 MIPS 的内容*——扩展了 MIPS 的内容，更系统地比较了 CISC 架构的 Pep/8 和 RISC 架构的 MIPS。新的 MIPS 章节用新的指令集表描述了 5 种寻址方式。MIPS 机器的数据区的图包括了伪直接寻址方式所要求的数据通路和复用器。明确命名的控制信号使用与 Pep/8 控制信号相同的语法，提供了 MIPS 指令实现更简洁而详细的描述。

独特的特性

本书有几个独特的特性，使之有别于其他计算机系统、汇编语言和计算机组成的书。

- 概念的方法——许多教科书试图跟上领域的变化，包括最新的技术发展。例如，最新外围设备的通信协议规范。这类书通常通篇是“设备如何工作”的描述性解释。本书避开了这类资料，而只选择基础的计算概念，掌握了这些就有了理解当前和未来技术的基础。例如，以数字电路设计问题来说，让学生掌握空间/时间折中的概念比简单地阅读通用描述更重要。再举一个例子，通过学习如何在ISA指令的微代码实现中合并周期来掌握硬件并行的概念，这样才是最好的。
- 强调问题解决——如果只听说或者读到某个主题，学生能记住的很少；但当他们体验到时，才会记住很多。本书强调问题解决，每章后面有近400道练习，其中很多有多个部分。这些练习不会让学生重复课本中的原话，而是要求量化地解答、分析或者设计系统某个抽象层次上的一个程序或电子电路。
- 一致的机器模型——Pep/8机器是一个小型的CISC计算机，是描述系统所有层次的载体。学生可以清晰地看到抽象层次之间的关系，因为他们要在所有的层次上为这个机器编程或者设计电子电路。例如，当在LG1层设计ALU组件时，他们知道ALU在ISA3层的实现中应该在哪个位置。通过像编译器那样把C++程序翻译成汇编语言，他们学到优化编译器和非优化编译器之间的差别。在不同层次上都使用同样的机器模型做工作在效率上有很大的优势，因为模型从上至下都是一致的。不过本书也讲述了MIPS机器，对比了RISC设计原理和微程序设计的CISC设计。
- 完整的程序示例——许多计算机组成和汇编语言的书会受到代码片段综合征的影响。Pep/8的内存模型、寻址方式和输入/输出特性使得学生能写出完整的程序，容易执行和测试，而不只是代码片段。真实的机器，特别是RISC机器，有复杂的函数调用协议，涉及寄存器分配、寄存器溢出和内存对齐限制之类的问题。Pep/8是少数几种教学机之一（有可能是唯一一个），允许学生书写具有输入/输出的完整程序，可以使用全局变量和局部变量、全局数组和局部数组、传值调用和传引用调用、数组参数、使用转移表的开关语句、递归、使用指针的链式结构和堆。写完整程序的作业进一步实现了通过动手来学习的目标，而不是通过读代码片段来学习。
- 理论和实践的结合——有些读者注意到了，讲述语言翻译原理的第7章在计算机系统书中不常见。这种现象可悲地说明了计算机科学课程体系甚至计算机科学领域中理论和实践之间的鸿沟。因为本书讲述了HOL6层的C++语言、Asmb5层的汇编语言和ISA3层的机器语言，而且都有一个目标，即理解层次之间的关系，一个更好的问题是：“为什么不能包括讲述语言翻译原理的一章呢？”本书尽可能地加入理论以支持实践。例如，介绍布尔代数作为一个公理系统，配合练习来证明定理。
- 宽度和深度——第1～6章中的内容对计算机系统或汇编语言编程的书来说是很典型的。第8～12章对计算机组成的书来说是很典型的。在一本书中包括这么广泛的内容是很独特的，而且它还在一个完整机器的各个抽象层次上使用一个一致的机器模型。数字电路LG1层内容的深度也是很特别的，它使得CPU的组成部分不再神秘。例如，本书描述了Pep/8 CPU的复用器、加法器、ALU、寄存器、内存子系统和双向总线的实现。学生学习逻辑门层的实现，没有概念上的空洞，而如果只是泛泛地描述，就只能选择相信而不能完全理解。

本书回答了这个问题：“汇编语言编程和计算机组成在计算机科学课程体系中的位置是什么？”它提供了对无处不在的冯·诺依曼机器架构的深入理解。本书的目标是提供本领域

内所有主要知识域的综合概述，包括软件和硬件的结合，理论和实践的结合。

计算机课程体系 2001

ACM 和 IEEE 计算机学会建立了计算机科学的“课程体系 2001”指导原则。该指导原则给出了知识体的分类和具体核心。本书适用于体系结构和组成（Architecture and Organization，AR）类别，几乎包含 AR 知识体中所有的核心主题。初期报告中的 AR 核心域以及本书中覆盖这些域的章节如下所示：

AR1. 数字逻辑和电子系统，第 10、11、12 章。

AR2. 数据的机器级表示，第 3 章。

AR3. 汇编层机器组成，第 4、5、6 章。

AR4. 内存系统组成和体系结构，第 9、11 章。

AR5. 接口与通信，第 8、9 章。

AR6. 功能组成，第 11、12 章。

AR7. 多处理和其他体系结构，第 8 章。

致谢

Pep/1 有 16 条指令、一个累加器和一种寻址方式。Pep/2 增加了变址寻址。John Vannoy 用 ALGOL W 语言写了 2 个模拟器。Pep/3 有 32 条指令，用 Pascal 语言编写，是学生软件项目，由 Steve Dimse、Russ Hughes、Kazuo Ishikawa、Nancy Brunet 和 Yvonne Smith 完成。Harold Stone 在早期审阅中提出许多对 Pep/3 架构的改进意见，后来被加到 Pep/4 中，并延续到后续的机器中。Pep/4 有特殊的栈指令，模拟 ROM 和软件陷阱。Pep/5 有更正交的设计，允许任何指令使用任何寻址方式。John Rooker 写了 Pep/4 系统和早期的 Pep/5 版本。Gerry St. Romain 实现了一个 MacOS 版本和一个 MS-DOS 版本。Pep/6 简化了变址寻址方式，也包括了一组完整的条件分支指令。John Webb 用 BlackBox 并发系统编写了跟踪功能。Pep/7 把安装的内存从 4 MB 增加到了 32 MB。Pep/8 把寻址方式的数量从 4 增加到 8，安装的内存增加到 64MB。Pep/8 汇编器和模拟器的 GUI 版本由一组学生用 Qt 开发系统和 C++ 实现和维护，小组成员包括 Deacon Bradley、Jeff Cook、Nathan Counts、Stuartt Fox、Dave Grue、Justin Haight、Paul Harvey、Hermi Heimgartner、Matt Hightfield、Trent Kyono、Malcolm Lipscomb、Brady Lockhart、Adrian Lomas、Ryan Okelberry、Thomas Rampelberg、Mike Spandrio、Jack Thomason、Daniel Walton、Di Wang、Peter Warford 和 Matt Wells。Ryan Okelberry 也参与编写了 Pep/8 CPU 模拟器。Luciano d'Ilori 编写了汇编器的命令行版本。

Tanenbaum 的《Structured Computer Organization》比其他任何一本书都更大地影响了本书的编写。本书扩展了 Tanenbaum 书的层次结构，在上面增加了高级语言层和应用层。

下面这些书稿审阅者和前一版本的用户极大地影响了本版本的终稿，他们是：Wayne P. Bailey、Jim Bilitski、Fadi Deek、William Decker、Peter Drexel、Gerald S. Eisman、Victoria Evans、David Garnick、Ephraim P. Glinert、Dave Hanscom、Michael Hennessy、Michael Johnson、Andrew Malton、Robert Martin、Richard H. Mercer、Randy Molmen、John Motil、Peter Ng、Bernard Nudel、Carolyn Oberlink、Wolfgang Pelz、James F. Peters III、James C. Pleasant、Eleanor Quinlan、Glenn A. Richard、David Rosser、Gerry St. Romain、Harold S.

Stone、J. Peter Weston 和 Norman E. Wright。Joe Piasentin 提供了艺术咨询。有两个人极大地影响了 Pep/8 的设计，一位是 Myers Foreman，有关指令集的很多想法都来自于他；另一位是 Douglas Harms，他提出很多改进意见，其中之一是 MOVSPA 指令，使得可以用传引用方式传递局部变量。

Jones and Bartlett Publishers 的责任编辑 Tim Anderson、产品主管 Amy Rose 和编辑助理 Melissa Potter 提供了宝贵的支持，很高兴与他们一起工作。Kristin Parker 设计的吸引人的封面正符合本书的风格。

我很幸运在一所致力于在本科教学中追求卓越的学校。佩珀代因（Pepperdine）大学的 Ken Perrin，提供了富有创造性的环境和专业的支持，正是在这种环境中，本书得到孕育。我的妻子 Ann 给予我无尽的支持，我要为本书占用的时间向她道歉，并送上我由衷的感谢。

Stan Warford
Malibu，California

目 录

Computer Systems, Fourth Edition

出版者的话
中文版序
译者序
前言

第一部分 应用层（第7层）

第1章 计算机系统……2
1.1 抽象层次……2
1.1.1 艺术中的抽象……3
1.1.2 文档中的抽象……4
1.1.3 组织中的抽象……5
1.1.4 机器中的抽象……6
1.1.5 计算机系统中的抽象……6
1.2 硬件……8
1.2.1 输入设备……9
1.2.2 输出设备……11
1.2.3 主存储器……12
1.2.4 中央处理单元……13
1.3 软件……13
1.3.1 操作系统……14
1.3.2 软件分析和设计……15
1.4 数据库系统……16
1.4.1 关系……17
1.4.2 查询……18
1.4.3 语言结构……19
总结……20
练习……21

第二部分 高级语言层（第6层）

第2章 C++……24
2.1 变量……24
2.1.1 C++编译器……24
2.1.2 机器无关性……24
2.1.3 C++的内存模型……25
2.1.4 全局变量和赋值语句……26
2.1.5 局部变量……28
2.2 控制流……29
2.2.1 if/else 语句……29
2.2.2 switch 语句……30
2.2.3 while 循环……30
2.2.4 do 循环……31
2.2.5 数组和 for 循环……31
2.3 函数……32
2.3.1 空函数和传值调用的参数……32
2.3.2 函数的例子……33
2.3.3 传引用调用的参数……34
2.4 递归……36
2.4.1 阶乘函数……37
2.4.2 递归的思考方式……39
2.4.3 递归加法……40
2.4.4 二项式系数函数……41
2.4.5 逆转数组元素顺序……43
2.4.6 汉诺塔……45
2.4.7 相互递归……48
2.4.8 递归的成本……48
2.5 动态内存分配……49
2.5.1 指针……49
2.5.2 结构……50
2.5.3 链式数据结构……51
总结……52
练习……53
问题……54

第三部分 指令集架构层（第3层）

第3章 信息的表示……58
3.1 无符号二进制表示……58
3.1.1 二进制存储器……58
3.1.2 整数……59
3.1.3 基本转换……60

3.1.4 无符号整数的范围 …… 61
3.1.5 无符号加法 …… 62
3.1.6 进位位 …… 62
3.2 补码二进制表示 …… 63
3.2.1 补码的表数范围 …… 65
3.2.2 基数转换 …… 66
3.2.3 数轴 …… 66
3.2.4 溢出位 …… 68
3.2.5 负数和零位 …… 69
3.3 二进制运算 …… 69
3.3.1 逻辑运算符 …… 69
3.3.2 寄存器传送语言 …… 70
3.3.3 算术运算符 …… 70
3.3.4 循环移位运算符 …… 72
3.4 十六进制和符号表示 …… 72
3.4.1 十六进制 …… 72
3.4.2 基数转换 …… 73
3.4.3 字符 …… 75
3.5 浮点数表示 …… 77
3.5.1 二进制小数 …… 77
3.5.2 余码表示 …… 78
3.5.3 隐藏位 …… 79
3.5.4 特殊值 …… 80
3.5.5 IEEE 754 浮点数标准 …… 83
3.6 跨层的表示方法 …… 85
3.6.1 另一种表示 …… 87
3.6.2 模型 …… 88
总结 …… 90
练习 …… 90
问题 …… 95

第 4 章 计算机体系结构 …… 97

4.1 硬件 …… 97
4.1.1 中央处理单元 …… 98
4.1.2 主存储器 …… 98
4.1.3 输入设备 …… 99
4.1.4 输出设备 …… 99
4.1.5 数据和控制 …… 100
4.1.6 指令格式 …… 100
4.2 直接寻址 …… 102
4.2.1 停止指令 …… 102
4.2.2 装入指令 …… 103
4.2.3 存储指令 …… 103
4.2.4 加法指令 …… 104
4.2.5 减法指令 …… 105
4.2.6 与和或指令 …… 105
4.2.7 按位取反和取负指令 …… 106
4.2.8 装入字节和存储字节指令 …… 107
4.2.9 字符输入和输出指令 …… 108
4.3 冯·诺依曼机器 …… 109
4.3.1 冯·诺依曼执行周期 …… 109
4.3.2 一个字符输出程序 …… 110
4.3.3 冯·诺依曼漏洞 …… 113
4.3.4 一个字符输入程序 …… 113
4.3.5 十进制转换为 ASCII …… 113
4.3.6 一个修改自身的程序 …… 114
4.4 ISA3 层编程 …… 115
4.4.1 只读内存 …… 117
4.4.2 Pep/8 操作系统 …… 117
4.4.3 使用 Pep/8 系统 …… 119
总结 …… 119
练习 …… 120
问题 …… 121

第四部分 汇编层(第 5 层)

第 5 章 汇编语言 …… 124

5.1 汇编程序 …… 124
5.1.1 指令助记符 …… 124
5.1.2 伪操作 …… 126
5.1.3 .ASCII 和 .END 伪操作 …… 126
5.1.4 汇编器 …… 127
5.1.5 .BLOCK 伪操作 …… 128
5.1.6 .WORD 和 .BYTE 伪操作 …… 129
5.1.7 使用 Pep/8 汇编器 …… 129
5.1.8 交叉汇编器 …… 130
5.2 立即数寻址和陷阱指令 …… 131
5.2.1 立即数寻址 …… 131
5.2.2 DECI、DECO 和 BR 指令 …… 131
5.2.3 STRO 指令 …… 133
5.2.4 解移位模式 …… 134
5.2.5 反汇编器 …… 135

5.3 符号……137
5.3.1 带符号的程序……137
5.3.2 一个码·语依量示例……138
5.4 从 HOL6 层翻译……139
5.4.1 cout 语句……139
5.4.2 变量和类型……140
5.4.3 全局变量和赋值语句……141
5.4.4 类型提高……143
5.4.5 Pep/8 符号跟踪器……144
5.4.6 算术移位和循环移位指令……145
5.4.7 常量和 EQUATE……147
5.4.8 指令和数据的放置……149
总结……149
练习……150
问题……152

第 6 章 编译到汇编层……155

6.1 栈寻址和局部变量……155
6.1.1 栈相对寻址……155
6.1.2 访问运行时栈……156
6.1.3 局部变量……158
6.2 转移指令和控制流……159
6.2.1 翻译 if 语句……160
6.2.2 优化编译器……161
6.2.3 翻译 if/else 语句……162
6.2.4 翻译 while 循环……163
6.2.5 翻译 do 循环……164
6.2.6 翻译 for 循环……165
6.2.7 面条代码……166
6.2.8 早期语言中的控制流……168
6.2.9 结构化编程定律……169
6.2.10 goto 争论……169
6.3 函数调用和参数……171
6.3.1 翻译函数调用……171
6.3.2 用全局变量翻译传值调用参数……173
6.3.3 用局部变量翻译传值调用参数……176
6.3.4 翻译非空函数调用……178
6.3.5 用全局变量翻译传引用调用参数……180
6.3.6 用局部变量翻译传引用调用参数……183
6.3.7 翻译布尔类型……186
6.4 变址寻址和数组……188
6.4.1 翻译全局数组……189
6.4.2 翻译局部数组……191
6.4.3 翻译作为参数传递的数组……193
6.4.4 翻译 switch 语句……198
6.5 动态内存分配……200
6.5.1 翻译全局指针……200
6.5.2 翻译局部指针……204
6.5.3 翻译结构……207
6.5.4 翻译链式数据结构……210
总结……214
练习……214
问题……215

第 7 章 语言翻译原理……222

7.1 语言、语法和语法分析……222
7.1.1 连接……223
7.1.2 语言……223
7.1.3 语法……224
7.1.4 C++ 标识符的语法……225
7.1.5 有符号整数的语法……226
7.1.6 上下文相关的语法……227
7.1.7 语法分析问题……227
7.1.8 表达式的语法……228
7.1.9 C++ 语法的一部分……229
7.1.10 C++ 的上下文相关性……232
7.2 有限状态机……233
7.2.1 用 FSM 来分析标识符……233
7.2.2 简化的有限状态机……234
7.2.3 非确定性有限状态机……234
7.2.4 具有空转移的状态机……235
7.2.5 语言符号识别器……237
7.3 实现有限状态机……239
7.3.1 查找表分析器……240
7.3.2 直接编码分析器……241
7.3.3 输入缓冲区类……244
7.3.4 多 token 分析器……249

7.4 代码生成……249
7.4.1 语言翻译器……249
7.4.2 语法分析器特性……259
总结……260
练习……260
问题……262

第五部分 操作系统层（第4层）

第8章 进程管理……266
8.1 装载器……266
8.1.1 Pep/8 操作系统……266
8.1.2 Pep/8 装载器……267
8.1.3 程序的终止……269
8.2 陷阱……269
8.2.1 陷阱机制……269
8.2.2 RETTR 指令……270
8.2.3 陷阱处理程序……271
8.2.4 陷阱寻址方式断言……273
8.2.5 陷阱操作数地址计算……274
8.2.6 空操作陷阱处理程序……277
8.2.7 DECI 陷阱处理程序……277
8.2.8 DECO 陷阱处理程序……282
8.2.9 STRO陷阱处理程序和OS向量……284
8.3 并发进程……286
8.3.1 异步中断……286
8.3.2 操作系统中的进程……287
8.3.3 多处理……288
8.3.4 并发处理程序……289
8.3.5 临界区……290
8.3.6 第一次尝试实现互斥……291
8.3.7 第二次尝试实现互斥……291
8.3.8 Peterson 互斥算法……292
8.3.9 信号量……293
8.3.10 带信号量的临界区……295
8.4 死锁……296
8.4.1 资源分配图……296
8.4.2 死锁策略……298
总结……298
练习……299
问题……302

第9章 存储管理……305
9.1 内存分配……305
9.1.1 单道程序设计……305
9.1.2 固定分区多道程序设计……306
9.1.3 逻辑地址……306
9.1.4 可变分区多道程序设计……308
9.1.5 分页……310
9.2 虚拟内存……312
9.2.1 大程序的行为……312
9.2.2 虚拟内存……313
9.2.3 按需取页……315
9.2.4 替换页……315
9.2.5 页替换算法……316
9.3 文件管理……318
9.3.1 磁盘驱动器……318
9.3.2 文件抽象……319
9.3.3 分配技术……319
9.4 错误检测和纠错码……321
9.4.1 错误检测码……321
9.4.2 编码要求……322
9.4.3 纠正一位错编码……324
9.5 RAID 存储系统……325
9.5.1 RAID 0 级：非冗余条带化……326
9.5.2 RAID 1 级：镜像……326
9.5.3 RAID 01 级和 10 级：条带化和镜像……327
9.5.4 RAID 2 级：内存风格的 ECC……328
9.5.5 RAID 3 级：位交叉奇偶校验……329
9.5.6 RAID 4 级：块交叉奇偶校验……329
9.5.7 RAID 5 级：块交叉分布奇偶校验……330
总结……331
练习……331

第六部分 逻辑门层（第1层）

第10章 组合电路……334
10.1 布尔代数和逻辑门……334
10.1.1 组合电路……335
10.1.2 真值表……335
10.1.3 布尔代数……336

10.1.4 布尔代数定理……337
10.1.5 互补证明……338
10.1.6 逻辑图……339
10.1.7 其他表达方式……341
10.2 组合分析……341
10.2.1 布尔表达式和逻辑图……342
10.2.2 真值表和布尔表达式……343
10.2.3 两级电路……345
10.2.4 无所不在的 NAND……346
10.3 组合设计……347
10.3.1 范式……348
10.3.2 三变量卡诺图……349
10.3.3 四变量卡诺图……353
10.3.4 对偶卡诺图……355
10.3.5 无关条件……356
10.4 组合设备……356
10.4.1 视角……356
10.4.2 复用器……358
10.4.3 二进制译码器……358
10.4.4 多路分配器……359
10.4.5 加法器……359
10.4.6 加法器 / 减法器……361
10.4.7 算术逻辑单元……362
10.4.8 LG1 层的抽象……367
总结……368
练习……368

第 11 章 时序电路……374

11.1 锁存器和时钟触发器……374
11.1.1 SR 锁存器……374
11.1.2 钟控 SR 触发器……376
11.1.3 主 – 从 SR 触发器……377
11.1.4 基本触发器……380
11.1.5 JK 触发器……381
11.1.6 D 触发器……382
11.1.7 T 触发器……383
11.1.8 激励表……384
11.2 时序分析和设计……384
11.2.1 时序分析问题……385
11.2.2 预设置和清除……387
11.2.3 时序设计……387
11.2.4 一个时序设计问题……388
11.3 计算机子系统……390
11.3.1 寄存器……390
11.3.2 总线……391
11.3.3 内存子系统……392
11.3.4 地址译码……396
11.3.5 双端口寄存器堆……400
总结……401
练习……402

第七部分 微代码层（第 2 层）

第 12 章 计算机组成……406

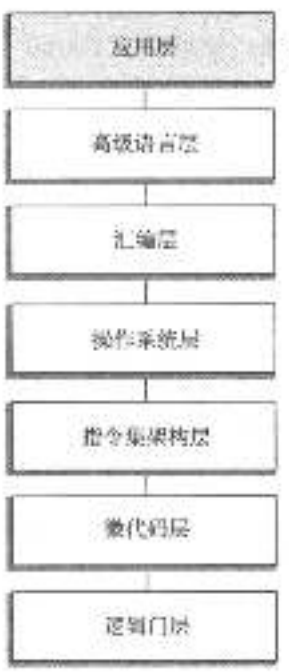
12.1 构造 ISA3 层机器……406
12.1.1 中央处理单元……406
12.1.2 冯·诺依曼周期……408
12.1.3 实现存储字节指令……412
12.1.4 实现加法指令……412
12.1.5 实现装入指令……413
12.1.6 实现算术右移指令……415
12.2 性能问题……416
12.2.1 总线宽度……417
12.2.2 特殊的硬件单元……419
12.2.3 3 个优化领域……421
12.2.4 微代码……423
12.3 MIPS 机器……425
12.3.1 装入 / 存储体系结构……425
12.3.2 指令集……427
12.3.3 高速缓存……431
12.3.4 MIPS 的计算机组成……437
12.3.5 流水线……439
12.4 结论……445
12.4.1 模型简化……446
12.4.2 更大的景象……446
总结……447
练习……448

附录 Pep/8 体系结构……451

部分练习参考答案……457

索引……468

应用层（第7层）



第 1 章 |
Computer Systems, Fourth Edition

计算机系统

计算机科学的基础问题是：什么能够被自动化？就像工业革命中研制出的机器使得手工劳作自动化了，计算机实现了信息处理的自动化。电子计算机是在 20 世纪 40 年代研发出来的，设计者构建计算机是为了自动求解数学问题。不过从那时起，计算机就被运用到了各种问题上，例如金融会计、航空预订、文字处理以及图形图像。计算机传播如此迅猛，几乎每天都有新的计算机自动化领域出现。

本书的目的是展示计算机是如何自动信息处理的。从原则上来说，计算机所能做的一切，你都能做。计算机和人执行任务的主要区别在于计算机执行任务非常迅速。不过要利用它的速度，人们必须指导计算机，也就是对计算机进行编程。

要想理解计算机的本质，最好的方法是学习如何对机器编程。编写程序要求学习编程语言。在投入学习编程语言的细节之前，本章首先介绍抽象的概念（本书就是基于抽象这一主题），然后描述计算机系统的硬件和软件组成，最后描述一个典型应用——一个数据库系统。

1.1 抽象层次

抽象层次这个概念在艺术以及自然和应用科学中都广泛存在。抽象的完整定义是多面的，对我们要讲述的领域来说，包括下面这样一些部分：

- 隐藏细节以展示物质的本质。
- 概要结构。
- 通过一连串的命令划分责任。
- 将一个系统细分成较小的子系统。

本书的主题就是把抽象应用到计算机科学中。不过，我们先从考虑其他一些非计算机科学领域中的抽象层次开始。从这些领域中得出的类比会扩展到上述抽象定义中的 4 个部分，再应用到计算机系统上。

抽象层次的 3 种常见图形化表示是：(a) 层次图；(b) 嵌套图；(c) 层次结构或树状图。现在，我们分析每种抽象的表示，说明它们是如何类比的。这三张图也适用于贯穿本书的计算机系统中的抽象层次。

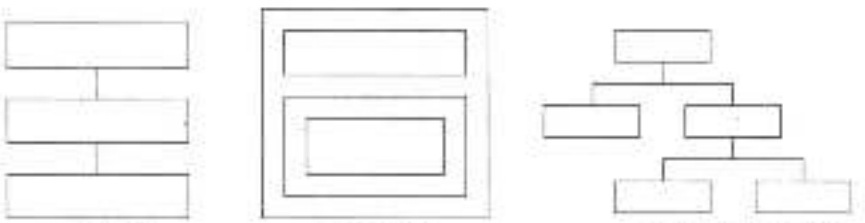
层次图，如图 1-1a 所示，是一组垂直排列的方框。最顶端的方框表示最高层次的抽象，最底端的方框表示最低层次的抽象。抽象的层数取决于所描述的系统。这张图表示一个三层抽象的系统。

图 1-1b 是嵌套图。与层次图一样，嵌套图也是一组方框。它总是包括一个大的外层方框，其余的方框都嵌套在里面。在这张图中，外层的大方框里直接嵌套了两个方框。这两个方框中，下面那个里面还嵌套了一个方框。嵌套图中最外层的方框对应于层次图中的最顶层方框。嵌套的方框对应于层次图中较低层次的方框。

在嵌套图中，方框是不重叠的。也就是说，嵌套图中绝不会有边界互相交叉的方框。一

个方框总是完全包含另一个方框。

抽象层次的第三种图形化表示是层次结构或树状图，如图 1-1c 所示。在树中，大树枝从树干分支出来，较小的树枝从较大的树枝中分支出来，以此类推。树叶在链路的末端，附着在最小的树枝上。图 1-1c 这样的树状图，主干在顶端而不是底部。每个方框称为一个结点，唯一的顶端结点称为根。没有更低层相连的结点是叶子。图 1-1c 是一棵有着一个根结点和三个叶子结点的树。层次结构图中顶端的结点对应于层次图中的顶端方框。

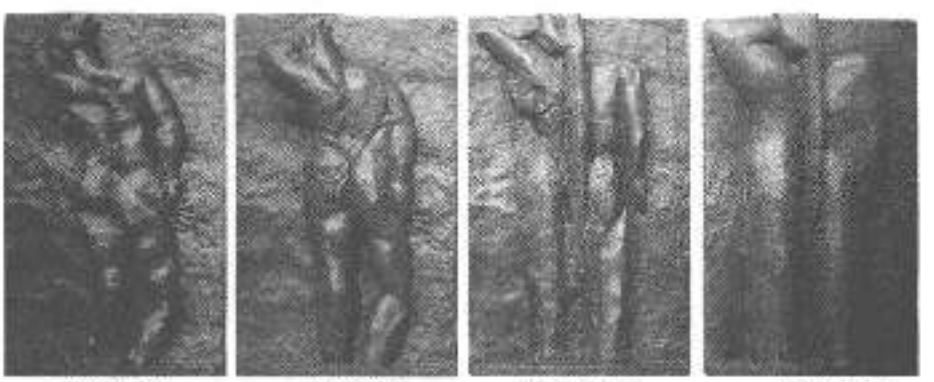


a）层次图　　b）嵌套图　　c）层次结构或树状图

图 1-1　三种抽象层次的图形化描述

1.1.1　艺术中的抽象

亨利·马蒂斯（Henri Matisse）是现代艺术史上的一位重要人物。1909 年，他制作了一尊女人后背的铜雕塑，名为《The Back I》。4 年后，他又创作了一尊同样主题的作品，不过用了更简单的处理方法，并将此作品命名为《The Back II》。又过了 4 年，他创作了《The Back III》，13 年后又创作了《The Back IV》。图 1-2 展示了这 4 尊雕塑。



The Back I
1909

The Back II
1913

The Back III
1917

The Back IV
1930

图 1-2　马蒂斯的铜雕塑。每件作品的表现手法越来越抽象

这些作品的一个显著特征是，在一件作品到另一件作品的发展中，艺术家不断地去除细节。第二尊雕塑中背部轮廓变得不怎么清晰，第三尊雕塑中右手手指被隐去，而在最为抽象的第四尊雕塑中，几乎不能识别出臀部。

马蒂斯追求表达性，他故意隐藏视觉上的细节而去展示主题的实质。1908 年他写道[⊖]：

> 在一幅画中，每个可见的部分都扮演着赋予它的角色，无论是主要的还是次要的。画中所有那些没用的部分都是有害的。艺术作品必须在整体上是和谐的，因为

⊖　Alfred H. Barr, Jr., Matisse: His Art and His Public（纽约：现代艺术博物馆，1951）。

在观赏者脑海中，多余的细节会使占主要元素的位置。

隐藏细节是抽象层次这一概念的必要组成部分，它直接适用于计算机科学。在计算机科学术语中，《The Back IV》抽象层次最高，《The Back I》层次最低。图 1-3 的层次图展示了这些层次的关系。



图 1-3　马蒂斯雕塑的抽象层次。《The Back IV》的抽象层次最高

就像艺术家一样，计算机科学家必须认识到要素和细节之间的区别。按时间进程，马蒂斯的“The Back”系列是从最注重细节演变到最抽象。然而，在计算机科学中，解决问题的进程应该是从最抽象到最详细。本书的目标之一就是教你如何进行抽象思维，如何在对一个问题制定解决方案时忽略无关的细节。并不是说在计算机科学中细节不重要！细节是最重要的。不过在计算问题中，人们自然倾向于在开始阶段非常注重很多细节；而在计算机科学中解决问题时，要素应该比细节优先考虑。

4
|
5

1.1.2　文档中的抽象

在编撰书面文档的架构时，抽象层次的作用也是显而易见的。以美国宪法为例，它包括 7 章，每章又分为多节。下面的大纲展示了条款和章节的标题，它并不是宪法本身的一部分[⊖]，只不过是总结了各个部分的内容。

第一条　立法部门
　第一款　国会
　第二款　众议院
　第三款　参议院
　第四款　参议员和众议员的选举——国会会议
　第五款　国会两院的权利与职责
　第六款　参议员和众议员的报酬、特权及任职限制
　第七款　法案通过的方式
　第八款　国会权利
　第九款　对联邦权利的限制
　第十款　对各州的限制
第二条　行政部门
　第一款　总统
　第二款　总统的职权
　第三款　总统的责任
　第四款　行政和文职官员的免职
第三条　司法部门
　第一款　联邦法院的司法权
　第二款　联邦法院的管辖范围
　第三款　叛国罪

⊖　加利福尼亚州参议员 J. A. Beak，参议院秘书长，《加利福尼亚州州宪法》，《美国宪法和相关文献》（萨克拉门托，1967）。

第四条 各州和联邦政府
　第一款 各州的官方法律
　第二款 各州的公民
　第三款 新的州
　第四款 对各州保护的保证
第五条 修正案
第六条 总体保障
第七条 宪法的批准

宪法作为一个整体是最高层次的抽象。每个特定的条款，例如第三条（司法部门），处理整体的一部分。这一条的某一款，例如第二款（联邦法院的管辖范围）讲述一个特定的主题，抽象层次最低。这个大纲从逻辑上对这些主题进行组织。

图 1-4 的嵌套图展示了宪法大纲的结构，外层的大方框代表整个宪法，它里面嵌套的 7 个小一点的方框代表“条”，每“条”里面的方框代表“款”。

这种对文档组织列大纲的方法在计算机科学中也是很重要的。按照大纲格式组织程序和信息的技术称为结构化程序设计（structured programming）。这和语文作文老师教你在写作细节之前先按照大纲的格式组织好报告是一样的道理。软件设计者先组织好程序大纲，再填写编程细节。

1.1.3 组织中的抽象

公司组织是另一个用到抽象层次概念的领域。例如，图 1-5 是一个假想的教材出版公司以层次结构图表示的部分组织结构。公司总裁在最高的层次，对整个公司的成功运营负责。4 个副总裁向总裁汇报，每个副总裁负责一个主要的运营部门。在每个经理和副总裁下面还有更多的层次，本图中没有展示出来。



图 1-4 美国宪法的嵌套图



图 1-5 一个假想的出版公司的简化组织图

组织结构图中的层级对应组织中相应的责任和权力。总裁代表整个公司的利益，给那些向她汇报的人授予相应的责任和权力。那些人再运用他们的权力去管理组织中其负责的部门，还可以向员工授予相应的责任。在企业中，每个人拥有的实际权力可能并不能由他们在官方结构图中的位置直接反映。而在其他的一些组织中，例如美国军队，有严格的命令链。图1-6是一个层次图，它展示了美军中的权力线，以及每个军官要对哪个单位负责。



图1-6　美国军队中的命令链。四星上将位于抽象的最高层

（来源：Henry Mintzberg，《The Structuring of Organizations》（组织的架构），1979，P27，由Prentice-Hall出版公司（Englewood Cliffs，NJ.）许可进行了修改）

类比组织结构图反映组织结构功能，计算机系统功能也可以按照类似的关系图反映出来。就像一个大的组织，一个大的计算机系统一般是按照层次结构来组织的。计算机系统中任何一部分都接收来自层次结构图中它的直接上级的命令。然后，它把需要执行的指令分发给层次结构中它的直接下级。

1.1.4　机器中的抽象

另一个抽象层次的例子是和计算机系统非常类似的汽车。与计算机系统一样，汽车是一台人造机器。它由发动机、变速器、电气系统、制冷系统和底盘组成。每部分都进一步细分，其中电气系统包括电池、前照灯、稳压器以及其他一些部件。

和汽车相关的人位于不同的抽象层次。驾驶者位于最高的抽象层次，驾驶者了解怎样驾驶汽车，例如，怎样启动，怎样加速，怎样刹车。

抽象的下一层次是初级机械师。与一般的司机相比，他们知道更多发动机盖下面的细节，知道怎么换机油和火花塞，但他们不需要知道有关开汽车的细节知识。

抽象的再下一层次是高级机械师，他们可以完全地拆卸发动机，拆解、修理并把它再组装到一起。如果只是简单地更换机油并不需要知道这些细节知识。

同样的道理，与计算机相关的人也位于不同的抽象层次，使用计算机不需要完全领悟计算机的每个抽象层次。你不需要成为机械师才能开车。同样，如果只想使用文字处理软件，你不需要成为经验丰富的程序员。

1.1.5　计算机系统中的抽象

图1-7展示了一个典型的计算机系统的层次结构。图中的每一层都有它自己的语言：

第7层（App7）：与应用程序有关的语言

第6层（HOL6）：与机器无关的编程语言

第5层（Asmb5）：汇编语言

第4层（OS4）：操作系统调用

第3层（ISA3）：机器语言

第2层（Mc2）：微指令和寄存器传输

第1层（LG1）：布尔代数和真值表

用这些语言编写的程序指示计算机执行一定的操作。一个执行特定任务的程序可以用

图 1-7 中的任一层次的语言编写。与汽车一样，一个用某一层语言写程序的人，不必懂得任何更低层的语言。



图 1-7 一个典型的计算机系统的层次结构。有些系统没有第 2 层

当发明计算机时，只有 LG1 和 ISA3 这两层。人们要想与这些机器通信，就必须在指令集架构层上用机器语言（machine language）对它们编程。机器语言对于机器来说很好，但是对于人类程序员来说单调乏味的且很不方便。于是 Asmb5 层的汇编语言（assembly language）应运而生，以便帮助人类程序员。

最早的计算机巨大且昂贵。当一个程序员占用计算机时，其他的用户就必须排队等候，这浪费了大量的时间。慢慢地，人们开发出了 OS4 层的操作系统（operating system），这样许多用户可以同时接入计算机。对于今天的个人计算机来说，即使只服务一个用户，操作系统仍然是管理程序和数据所必需的。

早期，每次当一家公司介绍一款新发布的计算机模型时，程序员就不得不学习那个模型的汇编语言。他们为旧机器所写的所有程序都不能在新机器上使用。于是，人们发明了 HOL6 层的高级语言（high-order language），这样程序可以不怎么改动就从一种计算机转移到另一种计算机上，而且用高级语言编程比用低级语言编程容易。你可能熟悉下面这些流行的 HOL6 层的语言。

- FORTRAN　公式转换器（FORmula TRANslator）。
- BASIC　初学者的通用符号指令代码（Beginner's All-purpose Symbolic Instruction Code）。
- LISP　列表处理。
- C++　流行的通用语言。
- Java　适合于万维网浏览器。

计算机系统的广泛使用促使人们开发出了许多 App7 层的应用程序。编写应用程序（application program）是为了解决特定类型的问题，比如打印工资支票、输入文档或者统计分析数据。这使得你可以把计算机当作工具来使用而无需知道更低层上的操作细节。

最底层 LG1 由称作逻辑门（logic gate）的电子组件组成。在向更高层发展的过程中，人们发现在逻辑门层上设置一个新的层次是非常有用的，它能够帮助设计者构建 ISA3 层的机器。在现在的一些计算机系统中，使用 Mc2 层的微程序设计（microprogramming）来实现 ISA3 层的机器。Mc2 层是发明手持式计算器的一种重要工具。

学习本书的目的是与计算机进行有效的沟通。为了实现这个目标，你必须学习计算机语言。较高层的语言比低层的语言更人性化，更易于理解，这也正是人们发明它们的原因。

大多数人最开始都是通过使用别人写的程序来学习 App7 层的计算机。准备输入公司工资单程序的办公人员，还有游戏玩家，都属于这一类。在用户手册中，通常有关于 App7 层的应用程序说明，描述应该怎样运行某个特定的程序。

在学习本书时，你会通过不断审视较低层的抽象来了解计算机系统内部的工作原理。进入的层越低，你就会发现越多的在更高层中被隐藏的细节。在不断学习的过程中，你要牢

记图 1-7。你要掌握大量貌似琐碎的细节。不过你要记住：计算机科学的美不在于细节的多样，而在于概念的统一。

1.2　硬件

我们构建计算机是为了解决问题。早期的计算机主要解决数学和工程问题，后来的计算机强调商业应用的信息处理。今天，计算机也控制各种诸如汽车发动机、机器人和微波炉之类的机器。计算机系统通过接收输入、处理输入并生成输出来解决这些领域的问题。图 1-8 说明了计算机系统的功能。

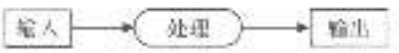


图 1-8　计算机系统的 3 个活动

计算机系统由硬件和软件组成。硬件（hardware）是系统的物理组成部分，一旦设计好，更改它会很困难且昂贵。软件（software）是一组程序，它指示硬件进行工作，比硬件容易修改。与只能解决一种问题的专用机器相比，计算机的价值在于它是可以解决许多不同问题的通用机器。通过给系统提供不同的指示，即不同的软件，用相同的硬件可以解决不同的
10 问题。

每台计算机有 4 个基本的硬件组件：

- 输入设备。
- 输出设备。
- 主存储器。
- 中央处理单元（CPU）。

图 1-9 以框图的形式展示了这些组件。方框之间的线代表信息流。总线是一组连接组件的线路，信息通过总线（bus）从一个组件流向另一个组件。处理过程是在 CPU 和主存储器（简称主存）中进行的。图 1-9 中的各个组件通过总线互连的结构是常见的，但是也有其他可能的结构。

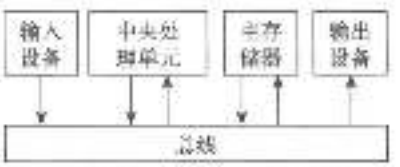


图 1-9　计算机系统 4 个组件的方框图

通常根据计算机硬件的相对物理大小对其进行分类：

- 小　个人计算机。
- 中　工作站。
- 大　大型主机。

对于大型主机，仅仅 CPU 就能占据一个整个的机柜，它的输入/输出（I/O）设备和存储器可能会填满一个房间。个人计算机可以小到放在桌子上或者公文包里。随着技术的发展，过去可能只能在大型计算机上做的处理变得可能在更小的计算机上进行了。现在，个人计算机可以做很多过去只有工作站或大型主机才能做的工作。

上述分类是基于计算机的物理大小，而不是存储器大小。计算机系统用户通常更关心存储器的大小，因为它是一个更直观的指标，指明硬件可以执行的有效工作量。计算速率是另一个对用户来说很重要的特性。一般来说，用户希望他的计算机 CPU 运算速率快并且存储容量大，而输入/输出设备和主存占用的物理空间又要小。

因此，当计算机科学家研究问题时，他们关心的是空间和时间——计算机系统内存储问题所需的空间和解决问题所需的时间。他们通常用如图 1-10a 所示的公制前缀来表示空间或
11 时间的大小数量。

倍 数	前 缀	缩 写
10^{15}	peta-	P
10^{12}	tera-	T
10^{9}	giga-	G
10^{6}	mega-	M
10^{3}	kilo-	k
10^{-3}	milli-	m
10^{-6}	micro-	μ
10^{-9}	nano-	n
10^{-12}	pico-	p

a) 表示 10 的幂的前缀

前 缀	计算机科学中的数值
peta-	$2^{50} = 1\,125\,899\,906\,842\,624$
tera-	$2^{40} = 1\,099\,511\,627\,776$
giga-	$2^{30} = 1\,073\,741\,824$
mega-	$2^{20} = 1\,048\,576$
kilo-	$2^{10} = 1\,024$

b) 表示较大数的前缀在计算机科学中的解释

图 1-10 科学表示法中的前缀

例 1.1 在图 1-9 中，假设通过总线从一个组件传送一些信息到另外一个组件需要 4.5 微秒（μs）。(a) 该传送需要多少秒？(b) 1 分钟内能发生多少次传送？

(a) 从图 1-10a 可知，时间 4.5 微秒是 4.5×10^{-6} 或者 0.000 004 5 秒。(b) 因为 1 分钟等于 60 秒，所以 1 分钟内可发生的传送次数是（60 秒）/（0.000 004 5 秒/传送），即 13 300 000 次传送。注意，由于题目给出的原始数据是 2 位有效数字，所以结果的有效数字不能超过 2 位或 3 位。 □

图 1-10a 说明在公制系统中，前缀 kilo- 表示 1000，mega- 表示 1 000 000。但是，在计算机科学中，kilo- 表示 2^{10} 或者 1024。1000 比 1024 小不到 3%，因此你可以把计算机科学中的 kilo- 理解为 1000，尽管它稍微大一点。这种方式同样适用于 mega- 和 giga-，如图 1-10b 所示。这时，近似值的偏差更大一些，但是对于 mega-，偏差仍在 5% 以内。采用这些看起来奇怪的惯例与在指令集架构层（ISA3 层）的信息表示有关。

1.2.1 输入设备

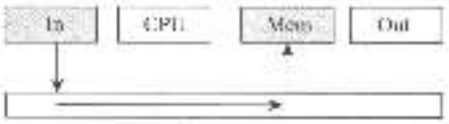
输入设备（input device）把从外部世界来的信息传送到计算机的存储器。图 1-11 展示了数据通过总线从输入设备到存储器的路径。输入设备的类型有很多种，包括：

- 键盘。
- 磁盘驱动器。
- USB 闪存驱动器。
- 鼠标设备。
- 条形码读码器。

当在键盘上敲击一个键时，你将一个字符发送到主存。这个字符以 8 位电子信号序列的形式存储在存储器中。序列中的每一位信号称为一个二进制数字（binary digit）（或者比特，bit（有时也译作“位”））。这个信号可以是高电压，用符号 1 表示，或是低电压，用符号 0 表示。8 位信号序列组成的字符叫作 1 字节（byte），如图 1-12 所示。

办公室文员在一台计算机键盘上进行输入是在应用层（App7 层），这是计算机系统结构中的最高抽象层次。他们不需要知道他们输入的每字符的位模式（bit pattern）。而指令集架构层（ISA3 层）的程序员必须知道位模式。现在，你应该记住数据的一个字节对应键盘的

一个字符。



In	输入设备
CPU	中央处理单元
Mem	主存储器
Out	输出设备

图 1-11　输入的数据通路。信息通过总线从输入设备流到主存储器



a）一个8比特字节的存储表示

b）字母“k”的位模式

图 1-12　1字节的信息。当在键盘上敲下“k”时，信号01101011会传送到总线上，然后存储到主存储器上

例 1.2　一个打字员以每分钟35个单词的速度在一台计算机键盘上输入文本。如果平均每个单词的长度是7个字符，那么每秒有多少比特被传送到主存呢？一个空格键是一个字符，假设平均每个单词后有一个空格。

包括空格，每个单词是8个字符，每秒输入的字符数等于（35单词/分钟）×（8字符/单词）×（1分钟/60秒）= 4.67字符/秒。因为一个字节存储一个字符，一个字节包含8比特，所以比特率是（4.67字符/秒）×（1字节/字符）×（8比特/字节）= 37.4比特/秒。　□

字节的缩写是大写字母B，比特的缩写是小写字母b。因此，例1.2中的结果可以写作37.4 b/s。再举一个例子，你可以把12 000字节写作12 KB。

磁盘驱动器（disk drive）是计算机的一部分，我们从磁盘提取数据或者把数据写到磁盘上。磁盘驱动器包括一个使磁盘旋转的发动机，一个轴或轮毂夹用于把磁盘固定在发动机上，一个或多个用来探测磁盘表面单个比特的读/写头。

硬盘被永久固定密封在磁盘驱动器中。通常，个人计算机的存储容量为250 GB～1 TB之间。工作站的存储容量为1 T～100 TB。大型主机的容量大于100 TB。硬盘驱动器通过在一个轴上堆叠数个盘片的方式来实现高容量，每个磁盘表面有一个专门的读/写头。

光盘最初是作为音频压缩盘流行起来的，不过很快就用于为计算机存储数据。光盘的记录技术是基于激光的，激光可以形成高度聚焦的单色光束。光盘上有嵌入式的纹理，它从中心螺旋状地盘出，上面有激光束记录的凹凸序列，每个凹或者凸代表被光束反射所探测到的比特信息。一张CD的典型存储容量是650 MB。DVD当初设计用来存储多通道伴音的视频信息，现在也被计算机工业采用，一张DVD的典型存储容量是4.7 GB。

12
13

例 1.3　要想把硬盘上20 GB信息传送到一组CD上，需要多少张CD？

硬盘上确切的字节数是20×10 737 412 824，每片CD上的字节数是650×1 048 576。然而，如果你满足于近似值，你可以估计硬盘上是20×10^9字节，每片CD是650×10^6字节。需要的CD数量是$(20\times10^9)/(650\times10^6)=31$张。　□

USB闪存驱动器（flash drive），也称为拇指驱动器，其实并不是一个磁盘驱动器。它是一个没有移动部件的固态设备，用来模仿硬盘驱动器的行为。缩写USB表示通用串行总线（Universal Serial Bus），它定义了许多硬盘驱动器和计算机系统之间的连接协议。当你把一个拇指驱动器插在计算机上时，对计算机来说，它就好像一个硬盘驱动器一样，因为拇指驱动器没有移动部件，所以它比硬盘驱动器要耐用，也不像硬盘驱动器，它是可移动的，存储容量可以达到大约16 GB。

鼠标（mouse）是一种非常常用的手持输入设备。光学鼠标内部是一个小的发光二极管，它向下在桌面或者鼠标垫上投射光束，光反射回一个每秒采样 1 500 次的传感器。鼠标里有一个数字信号处理器，它像一个被编程为只处理一个任务的微小计算机：探测桌面或者鼠标垫图像中的模式，并确定从上次采样到现在它们移动了多远。鼠标中的这个处理器从模式中计算鼠标的方向和速度，然后把这些数据发送到个人计算机，计算机再在显示器上显示光标图像。

条形码读码器（bar code reader）是另一种有效的输入设备。最常见的条形码也许就是超市商品上的通用产品代码（Universal Product Code，UPC）（见图 1-13）。UPC 符号中的每个数字有 7 个垂直的数据元素，每个数据元素是白色或者黑色。条形码读码器中的光电管检测明暗区域并把它们转换为比特，白色元素被读为 0，黑色元素被读为 1。图 1-14 展示了图 1-13 中 UPC 符号右半部分中两位数字的比特与黑白区域之间的对应关系。



图 1-13　一包麦片上的 UPC 符号，左边 5 位数字标识制造商，右边 5 位标识产品。桂格（Quaker）公司的代码是 30000，100% 天然麦片的代码是 06700

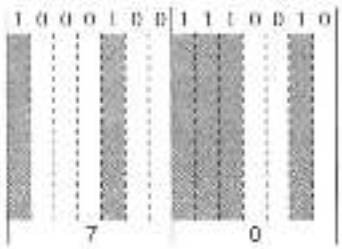


图 1-14　桂格麦片盒子上 UPC 符号的片段。视觉上，相邻的黑色区域组看上去就像粗的竖条

图 1-15 给出了 UPC 中十进制数值和二进制数值之间的对应关系，左半部分和右半部分中的字符代码是不一样的。一个黑色的竖条是由 1～4 个相邻的黑色区域构成，每个十进制数字都含有两个黑条和 2 个白色的空间。左半部分的字符都始于白条终于黑条，右半部分的字符都始于黑条终于白色空间；每个左半部分的字符都有奇数个 1，而每个右半部分的字符有偶数个 1。

超市的收银员在最高的抽象层次工作，他们不需要知道 UPC 符号的细节，只知道要录入 UPC 符号。如果没有听到确认的“哔”声，那就是发生了录入错误，必须重新扫描条形码。然而在较低层次的程序员必须知道编码的细节。例如，他们的程序必须检测左半部分每个字符的 1 或者黑色元素的个数，如果左半部分的字符 1 的个数是偶数，那么程序就不能发生确认的哔声。

十进制数值	左边的字符	右边的字符
0	0001101	1110010
1	0011001	1100110
2	0010011	1101100
3	0111101	1000010
4	0100011	1011100
5	0110001	1001110
6	0101111	1010000
7	0111011	1000100
8	0110111	1001000
9	0001011	1110100

图 1-15　UPC 符号中十进制数值的位模式

1.2.2　输出设备

输出设备（output device）从计算机的存储器传送信息到外部世界。图 1-16 展示了数据从主存到输出设备的通路。输出时，数据经过与输入设备使用的一样的总线，输出设备包括：

- 磁盘驱动器。
- USB 闪存驱动器。

- 显示屏（screen）。
- 打印机。

需要注意的是，磁盘驱动器和USB闪存驱动器既可以作为输入设备也可以作为输出设备。当磁盘用于输入时，这个过程叫作读；当磁盘用于输出时，这个过程叫作写。

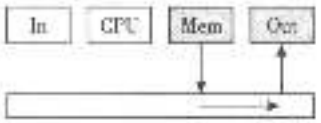
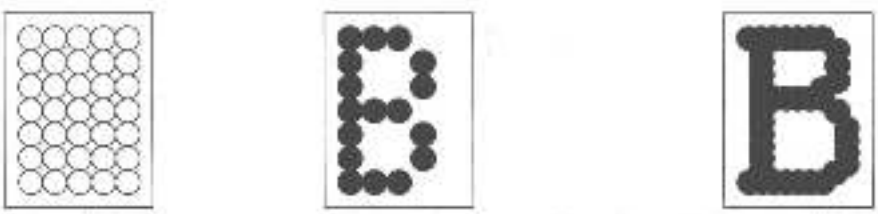


图1-16　输出的数据通路。信息从主存储器通过总线流到输出设备

显示屏是一种类似于电视机屏幕的视觉显示设备。它可以是阴极射线管（CRT）或平板。显示器是与键盘、CPU分开包装的。终端（terminal）是显示器加键盘。终端虽然好像是但其实并不是一台完备的通用个人计算机。终端与工作站和大型主机通信，没有这些后台设备，终端是没有用的。另一方面，个人计算机是完备的，就算不连接到更大的机器也能处理信息。个人计算机也能像终端一样和其他机器通信。在计算机发展的早期，一台标准终端显示屏有24行文本，一行最多80个字符。随着图形用户界面的出现，显示屏的大小不再是固定的文本行数，因为窗口和对话框的大小可以是各种各样的。不过，个人计算机上的终端仿真器程序在代表终端的窗口中有时遵循24行80字符的老标准。

显示屏上的单个字符实际上是由一个矩形点阵网格组成，每个点称为像素（pixel），它代表图像的元素。在黑白显示屏上，一个像素或明或暗。矩形网格中亮像素的模式形成了字符的图像。图1-17显示了一个形成字符‘B’的5列7行的像素网格。更高质量的显示屏在矩形网格中有更多的像素来形成更平滑的字符图像。你看在9×13这样的区域中，‘B’的图像是多么清晰。



a）5×7的像素网格　b）‘B’在5×7的像素网格上的图像　c）‘B’在9×13的像素网格上的图像

图1-17　矩形网格上的图像元素（像素）。b）中的像素与c）中的像素有相同的直径

打印机（printer）在性能和价格方面的差别很大。喷墨打印机的工作原理和显示屏上显示像素是一样的。当打印头移动通过纸张时，在适当的时刻向纸面喷射小的墨滴，形成所期望的图像。由计算机程序来控制释放墨水的时机。和显示屏一样，形成单个字符的点的数量越多，打印出来的图像质量就越高。许多打印机有多种运行模式，从低质量高速度到高质量低速度。

页面打印机是一种高质量的输出设备。大多数页面打印机用激光束在页面上形成图像。页面打印机的成像系统也使用像素，但是像素间隔足够紧密到不易察觉。一台常用的桌面激光打印机每英寸有600或1 200像素，一台每英寸600像素的打印机每平方英寸有600×600（即360 000）像素，而商用排版机器每英寸有2 400像素或者更多。

1.2.3　主存储器

主存储器（main memory，简称为主存）存储被处理的数据和处理数据的程序。和磁盘

一样，主存也是用字节来度量其容量的。小的个人计算机通常有 1 GB 主存，大的能够达到 8 GB；工作站通常有大于 8 GB 的主存；而大型主机有数百 GB 的主存。

主存的一个重要特性是，它是易失的。即，如果计算机的供电被有意或无意地中断，那么主存中的信息就会丢失，而磁盘不会这样。你可以从一台计算机上拔掉 USB 闪存，关掉机器，第二天回来时信息仍然在你的闪存里。

主存的另一个重要特性是，它的访问方式是随机的。实际上，组成主存的电子组件经常称作随机访问存储器（Random Access Memory，RAM）电路。不像硬盘驱动器，如果你刚从主存的一端获得一些信息，你可以不用经过中间的信息立刻随机地从另一端得到信息。

1.2.4　中央处理单元

中央处理单元（Central Processing Unit，CPU）包括控制计算机所有其他部分的电路。它有自己的存储器，称为寄存器（register）。CPU 电路中也永久固化有一组指令。这些指令完成的工作有：从存储器获取信息到寄存器、加、减、比较、把信息从寄存器存储回存储器等。指令执行的顺序不是固定不变的，这是由第三层的机器语言编写的程序所决定的。 14 ~ 16

按照人的标准，机器指令的执行是很快的。CPU 的速度一般用 GHz（即千兆赫兹）来度量。1 Hz 是每秒 1 条指令，因此，1 GHz 就是每秒 10 亿条指令。

例 1.4　假设 CPU 的频率是 2.5 GHz，执行一条指令的平均时长是多少？

2.5 GHz 表示每秒 2.5×10^9 条指令，那么每条指令的时长是 $1/(2.5\times10^9)=0.4\times10^{-9}$ 秒或 400 皮秒。 □

为了处理存储在主存中的数据，CPU 必须首先把数据放到自己本地的寄存器中，然后处理这些数据，再把结果返回到主存。最终，数据必须传送到输出设备才能为用户所用。图 1-18 展示了一个完整作业的数据流。

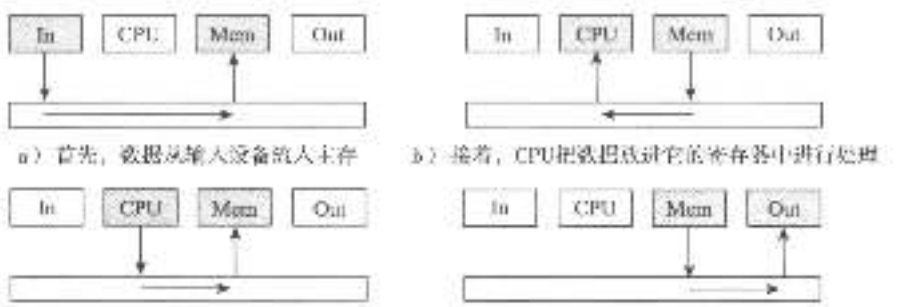


图 1-18　一个完整作业的数据流。步骤 b）和 c）通常会重复多次

1.3　软件

算法（algorithm）是一组指令，按照适当的顺序执行，在有限的时间内解决一个问题。算法并不一定需要计算机。图 1-19 是一个用中文表达的算法，解决制作 6 份奶油冻的问题。 17

这份食谱说明了算法的两个重要性质——有限数量的指令和在有限时间内执行。这个算法有 7 条指令——混合、搅拌、烹调、关火、冷却、添加和放凉。7 是一个有限的数量。算法中指令的数量不能是无限的。

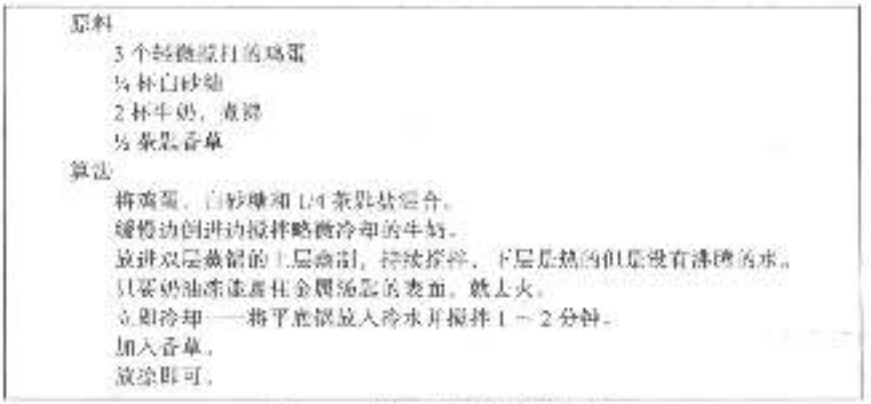


图1-19　制作搅拌奶油冻的算法

（来源：选自《Better Homes and Gardens New Cook Book》（更好的家和花园系列之新烹饪大全），Meredith公司版权所有，1981）

尽管奶油冻算法的指令数量是有限的，但是它的执行有一个潜在的问题。食谱告诉我们一直煮到蛋奶能够覆住金属汤匙，但是如果它一直不能覆住汤匙怎么办呢？那么如果严格遵照指令，我们就要一直煮下去！一个有效的算法绝不能无休止地执行。它必须提供一个在有限时间内解决问题的方案。假设奶油总是能覆住汤匙，那么这个食谱就是一个真正的算法了。

程序（program）是在计算机上执行的算法。程序不能用自然语言编写，必须用计算机系统的7层之一的语言编写。

通用计算机可以解决不同种类的问题，从计算公司的工资表到修正备忘录中的拼写错误。能够对硬件编程让它做不同的事情，硬件的多功能正是来源于此。控制计算机的程序称为软件（software）。

18

软件分为两大类：

- 系统软件。
- 应用软件。

系统软件（system software）使应用设计者可以接入计算机。同样的道理，应用软件（applications software）使位于App7层的用户可以使用计算机。一般来说，系统软件工程师在HOL6层和更低层设计程序，这些程序处理计算机系统中应用程序员不想费心的许多细节。

1.3.1　操作系统

计算机最重要的软件是操作系统，操作系统（operating system）是使硬件可用的系统程序。每个通用计算机系统都包括硬件和一个操作系统。

为了有效地学习本书，你必须能够使用一台有操作系统的计算机。一些常用的商用操作系统包括Microsoft Windows、Mac OS X、UNIX和Linux。不幸的是，每个操作系统都有自己独特的命令。本书不可能解释怎样使用所有不同的操作系统。你必须从你的老师或其他资源那里学习你的操作系统的详细内容。

操作系统有3个通用功能：

- 文件管理。

- 存储器管理。
- 处理器管理。

在3个功能中，对用户来说文件管理是最直观的。计算机新手要学会的第一件事情就是如何操作操作系统中的信息文件。

操作系统中的文件类似于办公室中的文件，它们包括应要求被检索和处理的信息。在办公室中，文件柜存储文件；在操作系统中，外围存储器设备存储文件。尽管磁带和磁盘都能存储文件，但是下面的讨论集中于磁盘。

在办公室中，每个文件放在一个文件夹中。办公室工作人员给每个文件命名，并且在文件夹的标签上标注文件名。文件名指示文件夹的内容，也便于从文件柜中取出某个特定的文件。在操作系统中，每个文件也有名字，这个文件名起到与标注在文件夹上的名字同样的作用——便于从磁盘上找出某个文件。

当计算机用户创建一个文件时，操作系统会要求给这个文件起一个名字。根据系统不同，通常对文件名长度和其中允许使用的字符有一些限制。有时，系统会自动给名字附加一 [19]
个前缀或者后缀。除了用户创建的文件外，其他的文件由系统生成并由它自动命名。

文件可以包含3种类型的信息：

- 文档。
- 程序。
- 数据。

文档可以是公司备忘录、信件和报告等。文件也可以存储计算机执行的程序。要执行程序时，首先要把它们从磁盘加载到主存中。一个正在执行的程序的输入数据可以来自文件，而输出数据也可以发送到文件。

物理上文件散布在磁盘的表面。为了记录所有这些信息文件，操作系统维护文件的目录。目录是磁盘上所有文件的一个列表，目录中的每个条目都包括文件名字、大小、在磁盘上的物理位置和其他任何操作系统管理文件所需的信息。目录本身也存储在磁盘上。

操作系统向用户提供了一种操作磁盘上文件的方法。一些典型的操作系统命令包括：

- 列出目录中的文件名。
- 从磁盘删除文件。
- 更改文件名。
- 打印文件内容。
- 执行应用程序。

要解答本书中的问题，你需要学习你的操作系统上的这些命令。

你的操作系统是由一组系统程序员针对你的计算机编写的一个程序。当你发出从磁盘删除一个文件的命令时，系统程序会执行这条命令。你，作为用户，正在使用其他人，即系统程序员，写的一个程序。

1.3.2 软件分析和设计

软件，无论系统软件还是应用软件，和文学有很多共同点。两者都是人写的。尽管计算机也能读并执行程序，但人能阅读这两样东西。小说家和程序员都很有创造性，因为他们提出的解决方案并不是唯一的。当小说家想要传达某些东西时，总是有不止一种的表达方式。好小说和坏小说之间的差别不仅体现在要表达的主题上，还体现于表达的方式。同样，程序 [20]

员要解决一个问题时，解决方案总有不止一种编程方法。好程序和坏程序之间的差别不仅体现在解决方案对问题解答的正确性上，还体现在程序的其他特性上，比如清晰程度、执行速度和对存储器的要求。

文学专业的学生要从事两项活动——阅读和写作。阅读是分析，读其他人写的东西并分析它的内容。写是设计或综合。要表达一个观点，你需要做的是有效地传达这个观点。大多数人觉得写比读难多了，因为它要求有更多的创造力。这也正是大众中读者比作家多的原因。

类似地，作为一个软件专业的学生，你将分析和设计程序。要牢记程序的3个活动是输入、处理和输出。在分析时，给出的是输入和处理指令，问题是确定输出。在设计时，给出的是输入和预期的输出，问题是写出处理指令，这就是软件设计。图1-20展示了分析和设计之间的不同。

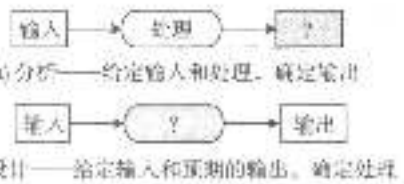


图1-20　分析与设计的不同

如同文学中的阅读和写作，设计优秀软件比分析优秀软件困难很多。计算机专业的学生最熟悉的抱怨是"我能理解概念，但是我不会写程序。"这是一个很自然的抱怨，因为它反映了综合相对于分析的困难之处。我们的终极目标是让你能够分析软件，同样能够设计软件。下面的几章将教你一些具体的软件设计技术。

不过，首先你得熟悉求解问题的一般性指导原则，它们同样适用于软件设计：

- 理解问题。
- 概括解决方案。
- 解决概括出的问题的各个部分。
- 手工验证你的解决方案。
- 在计算机上验证你的解决方案。

当面对一个软件设计问题时，通过写下一些样本输入和相应的输出，可以检测你对问题的理解。如果你不知道怎么手工解决一个问题，就不可能通过计算机解决这个问题。要概述问题的解决方案，你必须把问题分解成几个子问题，因为子问题比原始问题小，更容易解决。如果你对程序的正确性有疑问，应该在把它输入计算机之前手工验证它。可以用在第一步中写出的样本输入来对它运行测试。

很多学生认为这些步骤对入门性教材中的小程序不是很必要。如果对你来说问题很简单，那么在编写解决方案之前，也可以不在纸上组织思路。不过，这种情况下，你还是要在心里按照这些步骤进行思考。另一方面，你终究会碰到大的设计问题，那么这些解决问题的步骤将是不可或缺的。

1.4　数据库系统

数据库系统是App7层最常见的应用之一。数据库（database）是包括相关联信息的文件汇集，而数据库系统（database system）（也称作数据库管理系统，DBMS）是一个让用户在数据库中添加、删除和修改记录的程序，它也允许对数据库进行查询。查询（query）是对信息的请求，所请求的信息通常来自数据库的不同部分。

这里有一个数据库的例子，一个家具制造商利用数据库维护有关它的库存、零件供应商和货运的信息。查询可能是请求一个报告，显示仓库中制造某一款沙发所需部件的数量。为

了生成这个报告，数据库系统将文自数据库中不同部分的信息组合起来。在这个例子中信息来自库存文件和制造该沙发所需原料的文件。

数据库系统主要分为3类：层次型系统、网状型系统和关系型系统。在这3种类型的数据库系统中，层次型系统是最快的，但对用户来说是最受限制的。如果你能自然地把数据库中的信息组织成像层次型那样的结构，这个系统就是很适合的。网状型系统比层次型系统更灵活，但是对用户来说，它比关系型数据库系统更难一些。

关系型系统是三者中最常见的，它是App7层上最灵活也最容易使用的。但是在计算机系统中，没有免费的午餐，它获得高灵活性的代价是，相比于其他数据库系统，它的速度更低。这一节描述关系型DBMS背后的基本思想。

1.4.1 关系

关系型数据库系统（relational database system）把信息存储在文件中，对外呈现表结构。每个表有固定的列数和可变的行数。图1-21是一个关系型数据库中信息的示例。每个表有一个名字。名为Sor的表包含姐妹会成员的信息，名为Frat的表包含兄弟会成员的信息。位于App7层的用户先固定每个表中垂直的列数，再在表体中输入信息。水平方向的行数是可变的，这样能够在表中增加或者删除人员。 22

Sor

S.Name	S.Class	S.Major	S.State
Beth	Soph	Hist	TX
Nancy	Jr	Math	NY
Robin	Sr	Hist	CA
Allison	Soph	Math	AZ
Luiwa	Sr	CompSci	CA

Frat

F.Name	F.Major	S.State
Emile	PolySci	CA
Sam	CompSci	WA
Ron	Math	OR
Mehdi	Math	CA
David	English	AZ
Jeff	Hist	TX
Craig	English	CA
Gary	CompSci	CA

图1-21 一个关系型数据库的示例。这个数据库包括两个关系：Sor和Frat

在关系型数据库术语中，表称作关系（relation），列为属性（attribute），行叫元组（tuple，与couple同前）。在图1-21中，Sor和Frat是关系。（Nancy，Jr，Math，NY）是Sor的一个四元组，因为它有4个元素。而F.Major是Frat的一个属性。属性的域（domain）是该属性所有可能值的集合。S.Major和F.Major的域是集合{Hist，Math，CompSci，PolySci，English}。

Edgar Codd



Edgar Codd于1923年出生在英格兰多塞特郡波特兰市，是7个孩子中最小的一个。他曾经就读于牛津大学，主修数学和化学专业。第二次世界大战期间他是皇家空军的飞行员。1948年，他移居纽约，就职于IBM。不满于参议院的麦卡锡对所谓的共产主义同情者的攻击，他搬到圣何塞居住，20世纪50年代早期生活在这里。

Codd 最终在密歇根大学安娜堡分校获得了计算机科学专业博士学位，然后迁居加利福尼亚州圣何塞市，就职于 IBM 研究实验室。1970 年，他写出了里程碑式的论文“A Relational Model of Data for Large Shared Data Banks”（大型共享数据库的关系数据模型）。在该论文发表时，数据库系统的用户接口是在较低的抽象层次上。要执行一个查询，用户不得不使用复杂的查询语言，该语言依赖于数据在磁盘上存储方式的细节。Codd 的关系型数据库语言把用户放在了较高的抽象层次上，隐藏了原有语言中用户进行查询所需要知道的细节。Codd、Don Chamberlin 和 Ray Boyce 一起发明了结构化查询语言（Structured Query Language，SQL），SQL 语言已经成为关系型数据库查询的标准语言。

对 Codd 来说不幸的是，IBM 并没有像它的竞争对手那样迅速认识到 Codd 工作的商业价值。直到 Larry Ellison 以 Codd 的研究为基础创建了一家公司，就是后来的 Oracle（甲骨文）。1973 年，IBM 开始了 System R 计划，验证 Codd 的关系型理论。最终在 1978 年，Codd 论文发表整整 8 年后，IBM 开始构建商用关系型数据库产品。

Edgar Codd 被广泛认为是关系型数据库的发明者。1981 年，由于他在数据库管理系统理论和实现方面基础而持续的贡献，他被授予图灵奖。Codd 于 2003 年在佛罗里达州威廉姆斯岛的家中逝世，享年 79 岁。

1.4.2　查询

请求 Ron 的家乡州，请求姐妹会中所有二年级学生的名字，这些都是从该数据库进行查询的例子。请求一个列表，列出所有具有相同专业的兄弟会和姐妹会成员以及该专业的名字，是另一个查询的例。

在这个小例子中，你可以手工搜索数据库来确定每个查询的结果。Ron 的家乡州是 OR，Beth 和 Allison 是姐妹会中二年级学生。第二个查询制成表格稍微有点难度：Beth 和 Jeff 是历史专业；Nancy 和 Ron 是数学专业，Nancy 和 Mehdi 也是；Robin 和 Jeff 是历史专业等。

有趣的是，每个查询结果都可以以表格的形式列出来（见图 1-22）。第一个查询的结果是一个 1 列 1 行的表格，第二个查询的结果是一个 1 列 2 行的表格，而第三个查询的结果是一个 3 列 8 行的表格。关系型数据库是关系的汇集，而一个对关系型数据库查询的结果本身也是一个关系！

Result1

F.State
OR

Result2

S.Name
Beth
Allison

Result3

S.Name	F.Name	Major
Beth	Jeff	Hist
Nancy	Ron	Math
Nancy	Mehdi	Math
Robin	Jeff	Hist
Allison	Ron	Math
Allison	Mehdi	Math
Lulwa	Sam	CompSci
Lulwa	Gary	CompSci

图 1-22　对图 1-21 所示的数据库进行 3 次查询的结果。每个结果都是一个关系

查询结果本身就是关系这一事实是关系型数据库系统一个强大的理念。Apps7 层的用户把数据库看作关系的汇集，用户的查询就是请求从数据库现有的关系中衍生出另一个关系。

记住每层都有自己的语言。Apps7 层关系型 DBMS 的语言是一组对现有关系进行组合或者修改并产生新关系的命令。Apps7 层的用户使用这些命令生成想要的结果。图 1-23 展示了数据库、查询和结果之间的关系。数据库是输入，查询是一组 Apps7 层语言写的命令。就像在计算机系统中所有层上一样，三者之间的关系都是一样的形式：输入、处理和输出。

本章不描述市场上每种关系型数据库系统的每种语言，而是描述一种具备这样一些系统典型特征的简化语言。大多数关系型DBMS语言有许多强大的命令，但是3个命令是最基础的——select、project和join。

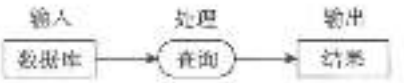


图1-23 数据库、查询和结果之间的关系

select和project语句类似，因为它们都是对单个关系进行操作，生成一个修改的关系。select语句是从一个特定的表中提取满足语句中指定条件的行。project语句是根据语句中指定的属性从一个特定的表中提取一组列。图1-24说明了语句

```
select Frat where F.Major = English giving Temp1
```

和

```
project Sor over S.Name giving Temp2
```

的结果。

Temp2

S.Name
Beth
Nancy
Robin
Allison
Lulwa

Temp3

S.Class	S.State
Soph	TX
Jr	NY
Sr	CA
Soph	AZ

Temp1

F.Name	F.Major	F.State
David	English	AZ
Craig	English	CA

a) select Frat where F.Major=English giving Temp1

b) project Sor over S.Name giving Temp2

c) project Sor over (S.Class, S.State) giving Temp3

图1-24 select和project操作符

project语句可以指定多列，此时属性用圆括号括起来，并用逗号分隔。例如，

```
project Sor over (S.Class, S.State) giving Temp3
```

从Sor关系中选出两个属性。

注意，在图1-24c中（Sr, CA）对是关系Sor（见图1-21）中的四元组（Robin, Sr, Hist, CA）和（Lulwa, Sr, CompSci, CA）都有的，但是这一对在关系Temp3中不重复出现。关系的一个基本性质就是在任何表中不能有重复的行。project操作符会检测重复行，不允许它们存在。从数学上说，关系就是元组的集合，集合中的元素不能有重复。

23 / 25

join与select和project不同，它的输入是两个表，而不是一个。第一个表的一列和第二个表的一列被指定作为join列。每个表的join列必须有相同的域。两个表join的结果是一个更宽的表，除了join列只出现一次以外，它的列和两个表中原始的列完全相同；结果表的行就是在两个原始表中在join列有相同元素的那些行。

例如，在图1-21中，S.Major列和F.Major有相同的域。语句

```
join Sor and Frat over Major giving Temp4
```

指定Major作为join列，关系Sor和Frat在这一列上进行合并。图1-25显示

S.Name	S.Class	S.State	Major	F.Name	F.State
Beth	Soph	TX	Hist	Jeff	TX
Nancy	Jr	NY	Math	Ron	OR
Nancy	Jr	NY	Math	Mehdi	CA
Robin	Sr	CA	Hist	Jeff	TX
Allison	Soph	AZ	Math	Ron	OR
Allison	Soph	AZ	Math	Mehdi	CA
Lulwa	Sr	CA	CompSci	Sam	WA
Lulwa	Sr	CA	CompSci	Gary	CA

图1-25 join操作符，该关系来自于语句join Sor and Frat over Major giving Temp4

了两个表 join 后的行是那些专业相同的行。Sor 的四元组 (Robin, Sr, Hist, CA) 和 Frat 的三元组 (Jeff, Hist, TX) 合并在 Temp4 中，因为它们的专业（Hist）是一样的。

1.4.3 语言结构

App7 层语言的语句有如下格式：

```
select 关系 where 表达 giving 关系
project 关系 over 属性 giving 关系
join 关系 and 关系 over 属性 giving 关系
```

语言的保留字包括

select	project
join	and
where	over
giving	

[26]

正如前面的例子展示的那样，每个保留字在语言中都有特定的含义。在语言中识别对象的单词，例如 Sor 和 Temp2 用于标识关系，而 F.state 用于标识属性，都不是保留字。它们是 App7 层的用户随意生成的，称为标识符（identifier）。保留字和用户定义的标识符在典型计算机系统的所有层语言中都是很常见的。

你知道怎样用 select，project 和 join 语句来生成图 1-22 中的查询结果吗？第一个询问 Ron 的家乡州的查询语句是

```
select Frat where F.Name = Ron giving Temp5
project Temp5 over F.State giving Result1
```

第二个询问姐妹会中所有二年级学生名字的查询语句是

```
select Sor where S.Class = Soph giving Temp6
project Temp6 over S.Name giving Result2
```

第三个请求查询一个列表，表中列出专业一样的兄弟会和姐妹会的成员以及他们共同的专业，该查询的语句是

```
join Sor and Frat over Major giving Temp4
project Temp4 over (S.Name, F.Name, Major) giving  Result3
```

总结

计算机科学的基础问题是：什么能够被自动化？计算机把信息处理自动化了。本书的主题是计算机系统中的抽象层次。抽象包括隐藏细节以展示物质的本质，概要结构，通过一连串的命令划分责任，将一个系统细分成较小的子系统。一个典型计算机系统的 7 个抽象层次是

第 7 层（App7）：应用层

第 6 层（HOL6）：高级语言层

第 5 层（Asmb5）：汇编层

第 4 层（OS4）：操作系统层

第 3 层（ISA3）：指令集架构层

第 2 层（Mc2）：微代码层

第 1 层（LG1）：逻辑门层

每层都有自己的语言，目的是隐藏更低层的细节。

计算机系统由硬件和软件组成。硬件的 4 个组成部分是输入设备、中央处理单元、主存储器和输出设备。控制计算机的程序叫作软件。

算法是一组指令，依照适当的顺序执行，在有限的时间内解决问题。程序是计算机上执行的算法。程序输入信息，处理信息并输出结果。

数据库系统是 App7 层最常见的一种应用。关系型数据库系统把信息存储在呈现为表结构的文件中，这个表称作关系。关系型数据库系统中的查询结果本身就是关系。关系型数据库系统中最基本的 3 个操作是 select、project 和 join。查询是这 3 种操作的组合。

练习

本书每章的最后都有一组练习和问题。请在纸上手工做这些练习。带星号练习的答案在本书的后面（对一些有多个部分的练习，可能只有部分答案）。问题是要输入计算机中的程序，本章仅包括练习。

1.1 节

1. （a）请画一个对应美国宪法的层次结构图；（b）依照图 1-5，画一个对应假定的出版公司的组织结构层次图。
2. 成吉思汗把他的战士以 10 人为一组组成十人队，由“什长”率领；10 个“什长”由 1 个“百夫长”率领；10 个“百夫长”由 1 个“千夫长”率领。
   *（a）如果成吉思汗在最低层有 10 000 名战士，那么他手下总共有多少人？
   （b）如果成吉思汗在最低层有 5763 名战士，那么他手下总共有多少人？假设如果可能，有 10 组每组应该 10 人，但是每层可能有一组可以包含少一些。
3. 《圣经》中，《出埃及记》第 18 章讲述了作为以色列唯一法官的摩西由于要处理大量琐碎的案件而疲惫不堪。他的岳父叶特罗向他推荐了上诉法院的分层体系，在这个体系中，最底层的法官负责 10 个市民，5 个管理 10 个市民的法官将他们不能解决的疑难案件交由一个负责 50 个市民的法官处理；2 个负责 50 个市民的法官在一个负责 100 个市民的法官手下工作；10 个能负责 100 个市民的法官在一个负责 1000 个市民的法官手下工作，能负责 1000 个市民的法官向摩西汇报，这样摩西只处理最棘手的案件。*（a）如果市民人口正好是 2000 人（不包括法官），画出分层体系最上面的三层。（b）在（a）中，包括摩西、所有法官和市民的总人口是多少？（c）如果市民人口正好是 10 000 人（不包括法官），包括摩西、所有法官和市民的总人口是多少？
4. 完全二叉树是一种叶子都在同一层的树，而每个非叶子结点下面正好有 2 个结点。图 1-26 是一个三层的完全二叉树。*（a）请画出一个四层的完全二叉树。*（b）一个五层的完全二叉树总共有多少个结点？（c）六层的呢？（d）N 层的呢？

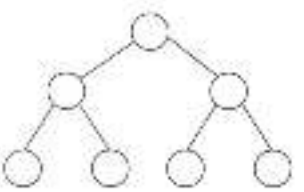


图 1-26　练习 4：一个三层的完全二叉树

1.2 节

*5. 一个打字员以每分钟 40 个单词的速度在键盘上输入文本。如果每个单词的平均长度是 5 个字符，那么每秒有多少比特传送到主存呢？一个空格也是一个字符，假定平均每个单词后面有一个空格。
6. 一个打字员以每分钟 30 个单词的速度在键盘上输入文本。如果每个单词的平均长度是 6 个字符，那么每秒有多少比特传送到主存呢？一个空格也是一个字符，假定平均每个单词后面有一个空格。
7. 你有 2300 首数字歌曲，平均每首歌的存储空间是 4.6MB。（a）如果把你收藏的所有歌曲都烧录到

650 MB 大小的 CD 上，需要多少张 CD？(b) 如果用 4.7 GB 的 DVD，需要多少张？

8. 你有平均每张需要 75KB 存储空间的照片。(a) 一张 650 MB 的 CD 上能放下多少张照片？(b) 一张 4.7 GB 的 DVD 能放下多少张照片？

*9. 一个显示屏上每个字符对应一个 8×10 的像素矩形网格，它能显示 24 行 80 列的字符。(a) 这个显示屏总共有多少像素？(b) 如果每个像素以 1 比特存储，那么存储一个显示屏画面需要多少 KB？

10. 一个显示屏上每个字符对应一个 5×7 的像素矩形网格，它能显示 24 行 80 列的字符。(a) 这个显示屏有多少像素？(b) 如果每个像素以 1 比特存储，那么存储一个显示屏画面需要多少 KB？

11. 一台桌面激光打印机的分辨率是每英寸 300 像素，如果每个像素存储在 1 比特内存中，那么存储一张 $8\frac{1}{2}\times 11$ 英寸纸面大小的完整图像，需要多少字节的内存？

12. 一本中等大小的书包含大约 100 万个字符。*(a) 在一台每秒打印 15 个字符的信函级质量打印机上打印这本书需要多少小时？(b) 假定每行平均 55 个字符，在一台每分钟打印 600 行的打印机上打印它需要多少小时？

13. 图 1-27 中的 UPC 符号代表哪两个十进制数字？

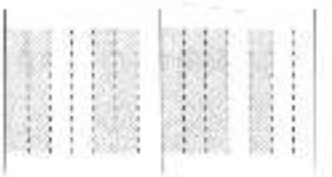


图 1-27　练习 13：数字是 UPC 符号右半部分的字符

1.3 节

14. 回答下列有关操作系统文件名的问题。(a) 对文件名的字符数有限制吗？如果有，限制是什么？(b) 特定的字符不允许做文件名吗？如果允许，有何限制？(c) 你的操作系统对文件名区分大小写字母吗？

15. 在你的操作系统中，怎样执行下面的步骤。(a) 如果你的机器是一台大型主机或者小型计算机，登录系统；如果是一台个人计算机，则启动系统。(b) 列出目录中的文件名。(c) 删除磁盘上的一个文件。(d) 更改一个文件的名字。(e) 复制一个文件。(f) 打印一个文件的内容。

1.4 节

*16. 根据本章 1.4 节中的讨论，写出关系 Temp5 和 Temp6。

17. 写出对图 1-21 中的数据库进行下列查询的语句。*(a) 找出 Beth 的家乡州。(b) 列出英语专业的兄弟会成员。(c) 列出有相同家乡州的兄弟会和姐妹会成员以及家乡州的名字。

28
:
30

18. (a) 写出生成图 1-22 中 Result2 的语句，但是要先用 project 命令，再用 select 命令。(b) 写出生成图 1-22 中 Result3 的语句，但是最后一条语句需是 join。

高级语言层（第6层）



C++

程序输入信息，处理信息并输出结果。本章展示了一个 C++ 程序怎样输入、处理和输出数值。本章讲述 HOL6 层的编程，我们假定你已经有一些用高级语言编程的经验，不一定非要是 C++，例如可以是 C、Java 或 Ada。因为本书表达的概念对所有这些语言都是通用的，所以虽然 C++ 与你熟悉的语言可能有所不同，但是你仍然能够看懂本章的讨论。

2.1 变量

计算机只能执行 ISA3 层（指令集架构层）上的机器语言语句。因此 HOL6 层的语句必须先被翻译到 ISA3 层，然后才能执行。图 2-1 展示了编译器的功能，它执行从 HOL6 层语言到 ISA3 层语言的翻译工作。这个图展示了到第 3 层的翻译，有些编译器是从第 6 层翻译到第 5 层，然后再要求从第 5 层翻译到第 3 层。



图 2-1 编译器的功能，把用第 6 层语言编写的程序翻译为等价的、较低层语言描述的程序

2.1.1 C++ 编译器

为了执行本书中的程序，需要有一个 C++ 编译器。运行程序分为 3 个步骤：

- 在文本编辑器中用 C++ 语言写程序，这个版本叫作源程序。
- 调用编译器把源程序从 C++ 翻译成编译为机器语言，机器语言的版本叫作目标程序。
- 执行目标程序。

有些系统允许用一条命令去指定后面的两个步骤，通常叫作“运行”命令。无论是否分开编译和执行，HOL6 层的程序在执行前必须进行翻译。

当你写源程序时，它像其他文本文件一样保存在磁盘文件中。编译器将生成另一个称为代码文件的目标程序文件。编译后目标程序在你的文件目录中是否可见取决于你的编译器。

如果想执行一个之前编译过的程序，就不需要再翻译它，只需直接执行目标程序即可。如果删掉了磁盘上的目标程序，总是可以通过再编译源程序来恢复它。但是翻译只能从高层到低层。如果删除了源程序，那么不能从目标程序恢复它。

C++ 编译器是软件，不是硬件。它存储在磁盘上的文件中。像所有的程序一样，编译器有输入、处理和生成输出这 3 个过程。从图 2-2 中可以看到编译器的输入是源程序，而输出是目标程序。



图 2-2 编译器是一个程序

2.1.2 机器无关性

ISA3 层语言是与机器相关的。使用 ISA3 层语言写的用于在 X 品牌计算机上执行的程

序，是不能在 Y 品牌计算机上运行的。HOL6 层语言的一个重要性质就是它们与机器无关。如果用 HOL6 层语言写了一个程序用于在 X 品牌计算机上执行，那么只用稍加修改，它就可以在 Y 品牌计算机上运行。

图 2-3 展示了 C++ 怎样实现它的机器无关性。假设用 C++ 写了一个做统计分析的应用程序。既想把它卖给有 X 品牌计算机的人，也想把它卖给有 Y 品牌计算机的人。只有当这个统计程序是机器语言格式时才能执行。因为机器语言是与机器相关的，所以需要 2 个机器语言的版本：X 品牌一种，Y 品牌一种。因为 C++ 是一种常用的高级语言，所以应该有 X 品牌机器的 C++ 编译器和 Y 品牌机器的 C++ 编译器。如果这样的话，那么只需在一台机器上调用 X 品牌的 C++ 编译器生成 X 品牌机器语言版本，在另一台机器上调用 Y 品牌的 C++ 编译器生成 Y 品牌机器语言版本，而只需写一个 C++ 程序。

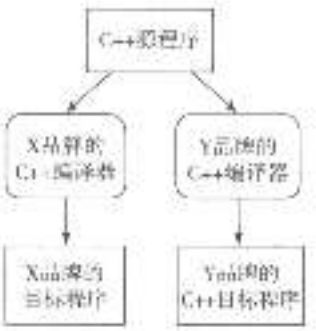


图 2-3 HOL6 层语言的机器无关性 [34]

2.1.3 C++ 的内存模型

C++ 编程语言有 3 种不同类型的变量：全局变量、局部变量和动态分配变量。变量的值存储在计算机的主存中，但是变量存储的方式取决于变量的类型。3 种类型的变量分别对应存储器中 3 个特定的区域：

- 全局变量存放在存储器中的固定位置。
- 局部变量存放在运行时栈上。
- 动态分配变量存放在堆上。

全局变量的声明在所有函数的外面，在程序的执行过程中位置保持不变。局部变量在函数中声明，函数被调用时它们出现，函数结束时它们消失。动态分配变量随着 new 运算符的执行出现，随着 delete 运算符的执行消失。

栈是一个值的容器，通过压入（push）操作存入值，通过弹出（pop）操作取出值。存入和取出值的原则是后进先出，即当从栈中弹出一个值时，取出的是最后一个进入栈的值。正因如此，有时候栈称为 LIFO 表，LIFO 是“Last In, First Out”（后进先出）的首字母缩写。

每条执行的 C++ 语句是一个函数的一部分。C++ 函数有一个返回类型、一个名字和一个参数表。程序包括一个名为 main 的特殊函数。通过执行 main 函数中的语句来执行程序。main 函数中的语句有可能调用另一个函数。当执行一个函数时，按照如下顺序对运行时栈的空间进行分配：

- 压入返回值的存储空间。
- 压入参数。
- 压入返回地址。
- 压入局部变量的存储空间。

当函数结束时，按照相反的顺序释放运行时栈的存储空间：

- 释放局部变量。
- 弹出返回地址，根据返回地址确定要执行的下一条语句。
- 释放参数。

35

- 弹出返回值，按照调用语句指定方式进行使用。

不管一个函数是 main 函数，还是在另一个函数中被一条语句调用的函数，都会执行上述这些步骤。

本章的程序说明 C++ 编程语言的内存模型。后面的章节将展示编译器把同样的程序翻译到 Asmb5 层后的目标代码。

2.1.4　全局变量和赋值语句

每个 C++ 变量有 3 个属性：

- 名字。
- 类型。
- 值。

变量名是程序员任意确定的标识符，变量类型指定变量值的可能类型。图 2-4 展示的程序声明了两个全局变量，输入它们的值，对它们的值进行操作，然后输出结果。这个程序没有实际的意义，它的唯一目的就是说明 C++ 程序的一些特点。

图 2-4 的前两行是注释，编译器会忽略注释。C++ 源程序中的注释以两条斜杠 // 开始直到本行结束。程序接下来的一行是

```
#include <iostream>
```

它是一个编译器指令（compiler directive），使得程序能够使用函数库。在这个例子中，库文件 iostream 包含程序后面要用到的输入函数 >> 和输出函数 <<。所有要使用 >> 和 << 的程序都需要这条指令或者类似的指令。语句

```
using namespace std;
```

是使用标识符 cin 和 cout 所必需的，这两个标识符是在 namespace std 中定义的。如果不用 using 语句，使用 cin 和 cout 时就必须用完全限定名（fully qualified name）。例如，main() 中的第一行就会写作

```
std::cin >> ch >> j;
```

程序中接下来的两行

```
char ch;
int j;
```

```
// Stan Warford
// A nonsense program to illustrate global variables.

#include <iostream>
using namespace std;

char ch;
int j;

int main () {
   cin >> ch >> j;
   j += 5;
   ch++;
   cout << ch << endl << j << endl;
   return 0;
}
```

输入

```
H 419
```

输出

```
I
424
```

图 2-4　HOL6 层和 Asmb5 层的全局变量赋值语句

声明了两个全局变量。第一个变量的名字是 ch，它的类型是字符型，是由变量名前面的关键字 char 来指定的。和大多数变量一样，声明并不能确定它的值，而必须从一个输入语句获得值。第二个变量的名字是 j，类型是整型，由 int 指定。每个 C++ 程序都有一个包含可执行语句的主函数。在图 2-4 中，因为变量是在主程序外声明的，所以它们是全局变量。

程序中接下来的一行

```
int main () {
```

声明了主程序是一个返回一个整数的函数。C++编译器必须生成能在特定操作系统上执行的代码，由操作系统来解释返回值。标准范例是，返回值为0表示程序执行中没有发生错误。如果发生了执行错误，则程序中断，然后返回一些非零的值，不会执行到main()最后一条可执行语句。这种情况下，如何处理取决于操作系统和错误的类型。本书中所有的C++程序都遵循通常的惯例：返回0作为main函数的最后一条可执行语句。

表2-4中第一条可执行语句是

```
cin >> ch >> j;
```

这条语句用输入运算符>>连接cin，cin表示标准输入设备。标准输入设备可以是键盘或者磁盘文件。在UNIX环境下，默认的输入设备是键盘。执行程序时，可以把输入重定向到一个磁盘文件。这条输入语句把输入流中的第一个值赋给ch，第二个值赋给j。

36 / 37

第二条可执行语句是：

```
j += 5;
```

C++中的赋值运算符是=，在英语中读作“gets”（意为获得，在中文里一般读作“等于”）。上面这条语句和下面这条是等价的：

```
j = j + 5;
```

英文中读作“j gets j plus five”（意为j的值为j加上5）。

和某些编程语言不同，C++把字符当作整数，可以对它们进行运算。下面这条可执行语句

```
ch++;
```

用增量运算符对ch加1，它等价于赋值语句

```
ch = ch + 1;
```

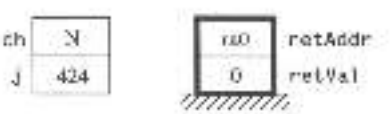
C++编程语言是C语言（它本身是从B语言发展而来）的一个扩展。语言设计者在确定C++这个名字时，就使用了增量运算符。

接下来的可执行语句是

```
cout << ch << endl << j << endl;
```

这条语句用输出运算符<<连接cout，cout表示标准输出设备，标准输出设备可以是显示屏也可以是磁盘文件。在UNIX环境下，默认的输出设备是显示屏，执行程序时，可以把输出重定向到一个磁盘文件。endl代表“end line”（意为行结束）。这条输出语句把变量ch的值传送到输出设备，然后把光标移到下一行的开始位置，把变量j的值传送到输出设备，再把光标移到下一行的开始位置。

图2-5展示了图2-4所示程序在结束前的内存模型。全局变量ch和j的存储位置是在内存的固定位置上分配的，如图2-5a所示。



a) 固定位置　　b) 运行时栈

图2-5　图2-4所示程序的内存模型

38

记住，当一个函数被调用时，运行时栈上分配了4项：返回值、参数、返回地址和局部变量。由于这个程序的主函数没有参数和局部变量，所以在运行时栈上仅分配了标记为retVal的返回值和标记为retAddr的返回地址的存储空间，如图2-5b所示。图中显示的返回地址值是ra0，这是操作系统中程序结束时要执行的指令的地址。对在HOL6层的我们来说，OS4层操作系统的

细节是隐藏的。

2.1.5　局部变量

全局变量在主存中的固定位置进行分配，而局部变量在运行时栈上进行分配。在C++程序中，局部变量在主程序内声明。图2-6所示的程序声明了一个常量和3个局部变量，3个局部变量分别表示一门课程的两次考试分数和一个总分，总分是两次考试分数的平均分加上奖励分。

```
#include <iostream>
using namespace std;

int main () {
   const int bonus = 5;
   int exam1;
   int exam2;
   int score;
   cin >> exam1 >> exam2;
   score = (exam1 + exam2) / 2 + bonus;
   cout << "score = " << score << endl;
   return 0;
}
```

输入

```
68 84
```

输出

```
score = 81
```

图2-6　处理3个局部整型值的C++程序

第一个变量之前的是常量bonus。与变量一样，常量有名字、类型和值。不过，与变量不同的是，常量的值不会改变。初始化运算符=将这个常量的值指定为5。

图2-6中的第一条可执行语句是

```
cin >> exam1 >> exam2;
```

它把输入流中的第一个值赋给exam1，第二个值赋给exam2。第二条可执行语句是

```
score = (exam1 + exam2) / 2 + bonus;
```

它把exam1和exam2的值相加得到的和除以2获得它们的平均值，再加上奖励分，接着把这个值赋给变量score。因为exam1、exam2和2都是整数，所以除法运算符/代表整数除法。如果exam1和exam2二者之一被声明为浮点型，或者除数写作2.0而不是2，那么除法运算符就代表浮点数除法。整数除法会截掉余数，而浮点数除法会保留小数部分。

例2.1　如果图2-6所示程序的输入是

```
58 85
```

那么输出仍然是

```
score = 81
```

考试分数和是153。如果用153除以2.0，得到浮点数值76.5。但是，如果用153除以2，运算符/代表整数除法，小数部分被丢掉，或者说砍掉，得到76。□

例2.2　如果把score声明为双精度浮点型，如下所示

```
double score;
```

并且如果通过将2改为2.0把除法强制为浮点数除法，如下所示

```
score = (exam1 + exam2) / 2.0 + bonus;
```

那么当输入是68和85时，输出是

```
score = 81.5
```

□

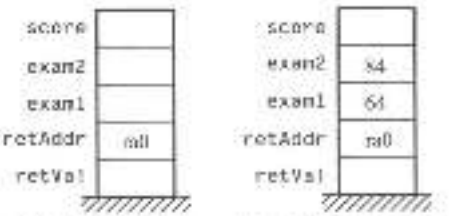
两个数的浮点除法仅生成一个值，即商。然而，整数除法生成两个值——商和余数，两者都是整型。可以用C++的模运算符%计算整

表达式	值	表达式	值
15 / 3	5	15 % 3	0
14 / 3	4	14 % 3	2
13 / 3	4	13 % 3	1
12 / 3	4	12 % 3	0
11 / 3	3	11 % 3	2

图2-7　一些整数除法和模运算的例子

型除法的余数。图 2-7 展示了一些整型除法和模运算的例子。

图 2-8 展示了图 2-6 所示程序中的局部变量的内存模型。计算机在运行时栈上给所有的局部变量分配存储空间。当 main() 执行时，返回值、返回地址、局部变量（exam1、exam2 和 score）被压入栈中。因为 bonus 不是变量，所以它不会入栈。



a) 输入语句执行之前　　b) 输入语句执行之后

图 2-8　图 2-6 所示程序中的局部变量的内存模型

2.2　控制流

程序是按照一条语句接着一条语句的方式顺序执行的。有两种方式可以改变控制流，进而改变程序顺序：选择和循环。C++ 有 if 和 switch 语句用于选择，while、do 和 for 语句用于循环。每一条语句都执行一个可能改变控制流顺序的测试。最常见的测试是用图 2-9 所示的 6 种关系运算符之一进行的。

2.2.1　if/else 语句

图 2-10 给出了一个 C++ if 语句的简单用法，用大于或等于关系运算符 >= 执行测试。程序输入整型变量 num 的值，然后把它和整型常量 limit 进行比较，如果 num 的值大于或等于 limit 的值（100），则输出单词 high，否则输出 low。没有 else 部分的 if 语句是合法的。

可以用图 2-11 所示的布尔运算符把数个关系测试结合起来。两个 & 号（&&）是 AND 运算符，两个竖线（||）是 OR 运算符，感叹号（!）是 NOT 运算符。

运 算 符	含　义
==	等于
<	小于
<=	小于或者等于
>	大于
>=	大于或者等于
!=	不等于

图 2-9　关系运算符

```
#include <iostream>
using namespace std;

int main () {
   const int limit = 100;
   int num;
   cin >> num;
   if (num >= limit) {
      cout << "high";
   }
   else {
      cout << "low";
   }
   return 0;
}
```

输入

75

输出

low

图 2-10　C++ if 语句

含　义	符　号
AND	&&
OR	||
NOT	!

图 2-11　布尔运算符

例 2.3　如果年龄、收入和婚姻是整型变量，if 语句

```
if ((age < 21) && (income <= 4000)) {
   tax = 0;
}
```

表示，如果年龄小于21且收入少于$4000，则激税值设置为0。□

在图2-10中的if语句，每个选择只有一条语句。如果在一个选择中需要执行多于一条的语句，那就必须用花括号{}把这些语句括起来，否则括号是可选的。

例2.4　图2-10中的if语句可以这样写

```
if (num >= limit)
   cout << "High";
else
   cout << "low";
```

输出语句不用花括号括起来。□

2.2.2　switch语句

图2-12中的程序使用C++的switch语句和用户玩一个竞猜小游戏，要求用户猜一个数字，然后根据输入的数字输出相应的消息。

也可以用if语句获得和switch语句相同的结果，然而等价的if语句不如switch语句效率高。

例2.5　图2-12中的switch语句可以用逻辑上等价的嵌套的if语句写为

```
if (guess == 0) {
   cout << "Not close";
}
else if (guess == 1) {
   cout << "Close";
}
else if (guess == 2) {
   cout << "Right on";
}
else if (guess == 3) {
   cout << "Too high";
}
```

然而，这个代码不如switch语句效率高。使用这个代码，如果用户猜3，那么所有4个测试都要执行。使用switch语句，如果用户猜3，程序立即跳到“Too high”语句，而不必用0、1和2与guess比较。

```
#include <iostream>
using namespace std;

int main () {
   int guess;
   cout << "Pick a number 0..3: ";
   cin >> guess;
   switch (guess) {
      case 0: cout << "Not close"; break;
      case 1: cout << "Close"; break;
      case 2: cout << "Right on"; break;
      case 3: cout << "Too high";
   }
   cout << endl;
   return 0;
}
```

交互的输入/输出

```
Pick a number 0..3: 1
Close
```

图2-12　C++ switch语句

2.2.3　while循环

图2-13中的程序是一个没有什么实际意义的程序，它唯一的目的是说明C++的while循环。程序的输入是以星号（*）结尾的一个字符串，输出不包括星号的所有字符。有经验的C++程序员不会用这个技巧。图2-13和本章中所有的程序都是为了在后面的章节中能够对它们在更低的抽象层次上进行分析。

```
#include <iostream>
using namespace std;

char letter;

int main () {
   cin >> letter;
   while (letter != '*') {
      cout << letter;
      cin >> letter;
   }
   return 0;
}
```

输入

```
happy*
```

输出

```
happy
```

图2-13　C++ while循环

在进入循环前，程序给全局变量 letter 输入第一个字符的值。语句

```
while (letter != '*')
```

将 letter 的值和星号字符进行比较。如果它们不相等，那么执行循环体，输出这个字符，然后输入下一个字符。接着，控制流返回到循环顶部的条件判断。

如果 letter 是局部变量而不是全局变量，那么程序的输出还是一样。把变量声明为全局的还是局部的是一个软件设计问题。经验法则是：总是把变量声明为局部变量，除非有很好的原因不这么做。局部变量能增强软件系统的模块性，让长程序更容易阅读和调试。图 2-4 和图 2-13 中的全局变量不代表好的软件设计。以这种方式呈现是为了说明 C++ 的内存模型。后面的章节会展示 C++ 编译器怎样翻译本章中给出的程序。

```
#include <iostream>
using namespace std;

int cop;
int driver;

int main () {
   cop = 0;
   driver = 40;
   do {
      cop += 25;
      driver += 20;
   }
   while (cop < driver);
   cout << cop;
   return 0;
}
```

输出

```
200
```

图 2-14 C++ do 循环

2.2.4 do 循环

图 2-14 中的程序说明 do 语句的使用。这个程序比较特殊，因为它没有输入，每次程序执行都生成同样的结果。这也是一个没有实际意义程序，只是为了说明控制流。

一个警官的初始位置在 0 单位处，然后开始追一个初始位置在 40 单位的司机。每执行循环体一次代表一个时间间隔。在此期间，警官行进 25 个单位，司机行进 20 个单位。语句

```
cop += 25;
```

是 C++ 中语句

```
cop = cop + 25;
```

的缩写形式。

与图 2-13 中的循环不一样，do 语句是在循环的底部进行测试，因此循环体能保证至少被执行一次。当执行语句

```
while (cop < driver);
```

执行时，它比较 cop 的值和 driver 的值。如果 cop 小于 driver，则控制流转到 do，于是循环体重复。

2.2.5 数组和 for 循环

图 2-15 中的程序用来说明 for 循环和数组。程序分配了一个 4 个整数的局部数组，把值输入数组，然后以相反的顺序输出值。

语句

```
int vector[4];
```

声明变量 vector，该变量是一个由 4 个整数组成的数组。在 C++ 中，所有数组的第一个

```
#include <iostream>
using namespace std;

int main () {
   int vector[4];
   int j;
   for (j = 0; j < 4; j++) {
      cin >> vector[j];
   }
   for (j = 3; j >= 0; j--) {
      cout << j << ' ' << vector[j] << endl;
   }
   return 0;
}
```

输入

```
2 26 -3 9
```

输出

```
3 9
2 -3
1 26
0 2
```

图 2-15 C++ 数组 for 循环

索引都是0。因此，这个声明为数组元素

```
vector[0] vector[1] vector[2] vector[3]
```

分配存储空间。

声明中的数字指定要分配多少个元素。它总是比最后一个元素的索引大1。在这个程序中，元素个数是4，比最后一个元素的索引3大1。

每个for语句有一对圆括号，它里面分为3个部分，各部分之间用分号隔开。第一个部分赋初值，第二个部分是测试，第三个部分是增量递增。在这个程序中，for语句

```
for (j = 0; j < 4; j++)
```

中，j=0是赋初值，j<4是测试，j++是递增。

当程序进入循环时，j设置为0。因为测试在循环的顶部，所以j的值和4进行比较。由于j小于4，所以循环体

```
cin >> vector[j];
```

执行。输入流中的第一个整数值被读入v[0]。控制返回到for语句，因为第三个部分的表达式是j++，所以j递增1。接着j的值和4比较，重复处理过程。

递减表达式

```
j--
```

是C++中j=j-1的缩写，所以第二个循环是以相反的顺序打印值。

2.3　函数

C++有两种函数：一种返回值为空，另一种返回值为非空类型。函数main()返回整型，不是空值。操作系统根据这个整数来确定程序是否正常结束。返回值为空的函数（简称空函数）完成处理，不返回任何值。返回值为空的函数的常见用法是输入或输出一组值。

2.3.1　空函数和传值调用的参数

图2-16中的程序使用空函数打印一个数值的柱状图。程序把第一个值读入整型变量numPts。在主程序中，全局变量j控制for循环执行numPts次。每次执行循环都调用空函数printbar。图2-17展示了图2-16中程序开始执行时的轨迹。

当调用一个空函数时，运行时栈中的分配按照以下顺序进行：

- 压入实际参数（简称实参）。
- 压入返回地址。
- 压入局部变量的存储空间。

图2-17c是图2-16所示程序分配过程的开始，程序压入形式参数（简称形参）n的值value。在图2-17f中，压入返回地址。图2-17g中压入局部变量的存储空间。分配过程完成后，列表中的最后一个局部变量k在栈的顶部。

被压入运行时栈的所有项目的集合称为栈帧（stack frame）或活跃记录（activation record）。在图2-16的程序中，该空函数的栈帧由3项组成：n、返回地址和k。图中用ra1标识的返回地址是主程序中for语句结束处的地址。main函数的栈帧由两项组成：返回值和返回地址。

空函数打印柱状图的一根柱子后，控制返回到主程序。运行时栈上的项目按照与分配顺

序相反的顺序进行释放。过程如下：

- 释放局部变量的存储空间。
- 弹出返回地址。
- 释放实参。

```
#include <iostream>
using namespace xin;

int numPts;
int value;
int j;

void printBar (int n) {
   int k;
   for (k = 1; k <= n; k++) {
      cout << "*";
   }
   cout << endl;
}

int main () {
   cin >> numPts;
   for (j = 1; j <= numPts; j++) {
      cin >> value;
      printBar (value);
   } // ral
   return 0;
}
```

输入

```
12 5 15 17 34 27 23 25 29 16 10 0 2
```

输出

```
***
*************
*****************
*********************************
***************************
***********************
*************************
*****************************
****************
**********

**
```

图 2-16 打印柱状图的程序。该空函数打印一根柱子

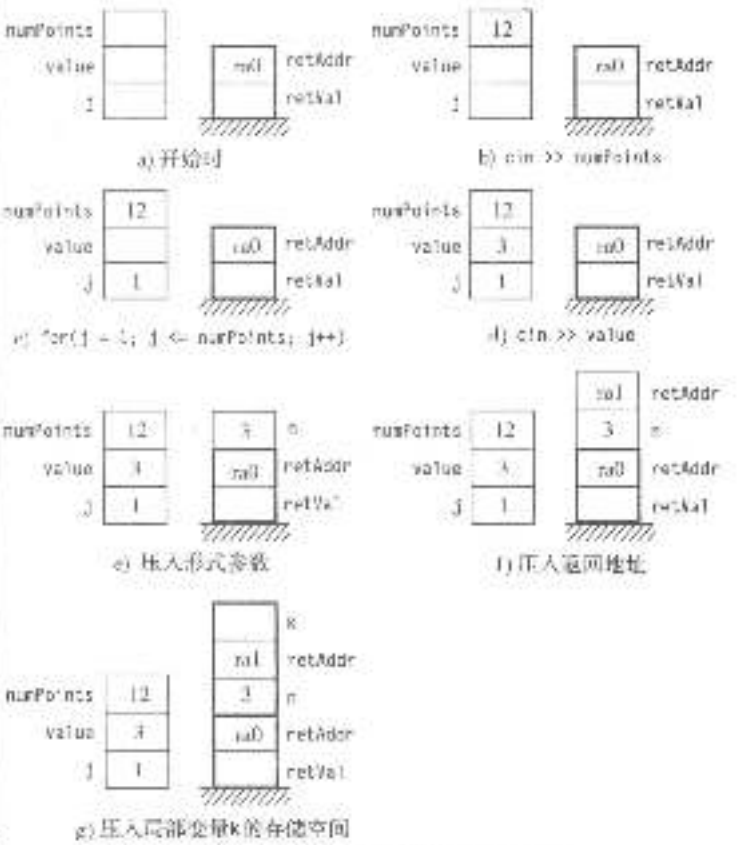


图 2-17 图 2-16 所示程序的运行时栈

程序通过返回地址知道，在执行完空函数的最后一条语句后，接下来去执行主程序中的那条语句。在主程序代码中，那条语句被标识为 ral，是过程调用后面的一条语句。

2.3.2 函数的例子

图 2-18 中的程序用函数计算一个整数的阶乘值。程序提示用户输入一个小的整数，把它作为参数传递给函数 fact。

图 2-19 展示了图 2-18 中函数的分配过程，该函数返回实参的阶乘。图 2-19c 表明首先压入返回值的存储空间。图 2-19d 展示压入形参 n 的值 num。图 2-19e 中压入返回地址。图 2-19f 和 g 压入局部变量 f 和 j 的存储空间。

这个函数的栈帧有5项。图中标记为ra1的返回地址表示主程序中cout语句的地址。控制从该函数返回到调用该函数的语句（简称调用语句）。这与空函数是不同的。在空函数的调用中，控制是返回调用语句后面的那条语句。

```
#include <iostream>
using namespace std;

int num;
int fact (int n) {
   int f, j;
   f = 1;
   for (j = 1; j <= n; j++) {
      f *= j;
   }
   return f;
}

int main () {
   cout << "Enter a small integer: ";
   cin >> num;
   cout << "Its factorial is: " << fact (num) << endl; // ra1
   return 0;
}
```

交互式输入/输出

```
Enter a small integer: 3
Its factorial is: 6
```

图2-18　用函数计算整数阶乘的程序

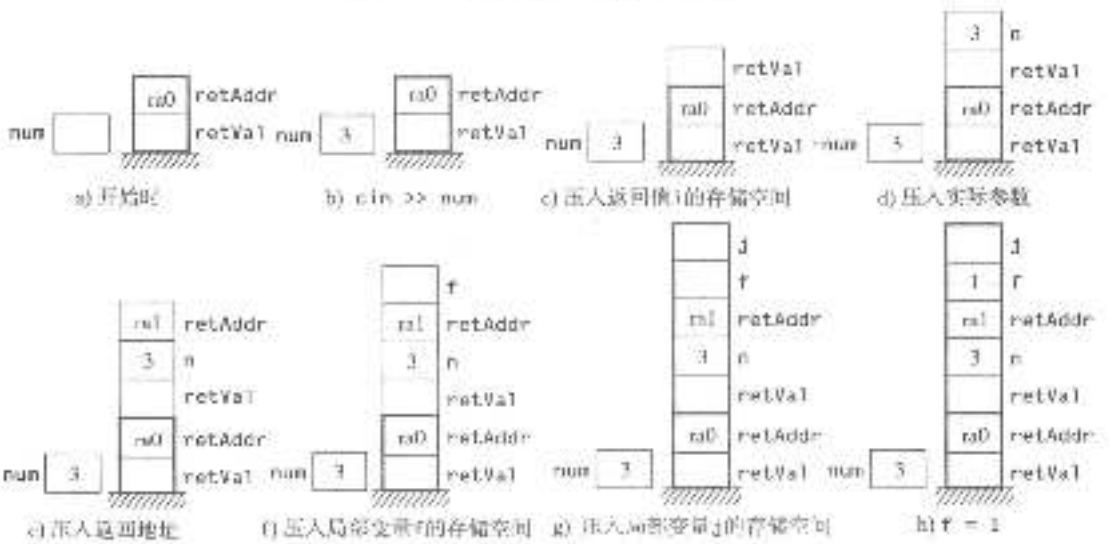


图2-19　图2-18所示程序的运行时栈

2.3.3　传引用调用的参数

前面讲述程序中的过程和函数都是通过值传递参数的。在传值调用中，形参获得实参的值。如果被调用的过程改变了它的形参值，调用过程中相应的实参值不变。被调用过程所做

的任何改变都是对运行时栈上的值进行的，当栈帧释放后，任何被改变的值也随之释放。

如果一个过程意在改变调用程序中实参的值，那么就要使用传引用调用而不是传值调用。在传引用调用中，形参获得的是对实参的一个引用。如果被调用过程改变了它的形参的值，那么调用程序中相应的实参也会改变。要指定一个参数为传引用调用，就要在参数列表中该参数的类型后加一个 & 符号。如果没有这个 & 符号，编译器会假定该参数是传值调用的。

图 2-20 中的程序用传引用调用改变实参的值。它提示用户输入两个整数，排序输入。它有一个空函数 order，调用另一个空函数 swap。图 2-21 展示了整个程序的分配和释放的顺序。

48 ⋮ 52

```
#include <iostream>
using namespace std;

int a, b;

void swap (int& r, int& s) {
   int temp;
   temp = r;
   r = s;
   s = temp;
}

void order (int& x, int& y) {
   if (x > y) {
      swap (x, y);
   } // ra2
}

int main () {
   cout << "Enter an integer: ";
   cin >> a;
   cout << "Enter an integer: ";
   cin >> b;
   order (a, b);
   cout << "Ordered they are: " << a << ", " << b << endl; // ra1
   return 0;
}
```

交互式输入/输出

```
Enter an integer: 6
Enter an integer: 2
Ordered they are: 2, 6
```

图 2-20 对两个数排序的程序。空函数使用传引用方式传递参数

图 2-21c 中 order 的栈帧有 3 项。形参 x 和 y 是传引用调用的，有箭头从运行时栈上的 x 指向主程序中的 a，这表明 x 引用 a。类似地，从运行时栈上的 y 指向主程序中 b 的箭头表明 y 引用 b。ra1 标识的返回地址是 cout 语句的地址，该语句在主函数中对 order 的调用之后。

图 2-21d 中 swap 的栈帧有 4 项，r 引用 x，x 引用 a，因此 r 引用 a。箭头从运行时栈上的 r 指到 a，箭头同样从 x 指向 a。类似地，箭头从运行时栈上的 s 指向 b，箭头同样从 y 指向 b。ra2 标识的返回地址是函数 order 最后一条语句的地址。swap 中的语句交换 s 和 r 的值。因为 r 引用 a，s 引用 b，所以它们交换的是主程序中 a 和 b 的值。

当空函数结束时，要释放它的栈帧，栈帧中的返回地址告诉计算机接下来去执行那条指

令。图 2-21e 展示了从空函数 swap 返回的过程，释放其栈帧。swap 栈帧中的返回地址告诉计算机在释放该栈帧之后顺序执行标号为 ra2 的语句。虽然图 2-20 中的代码 ra2 处并没有语句，但是空函数结尾处隐含有一个 return 语句，这在 HOL6 层是不可见的。

图 2-21f 展示了释放 order 栈帧的过程。order 栈帧中的返回地址告诉计算机在释放该栈帧之后，执行主程序中的 cout 语句。

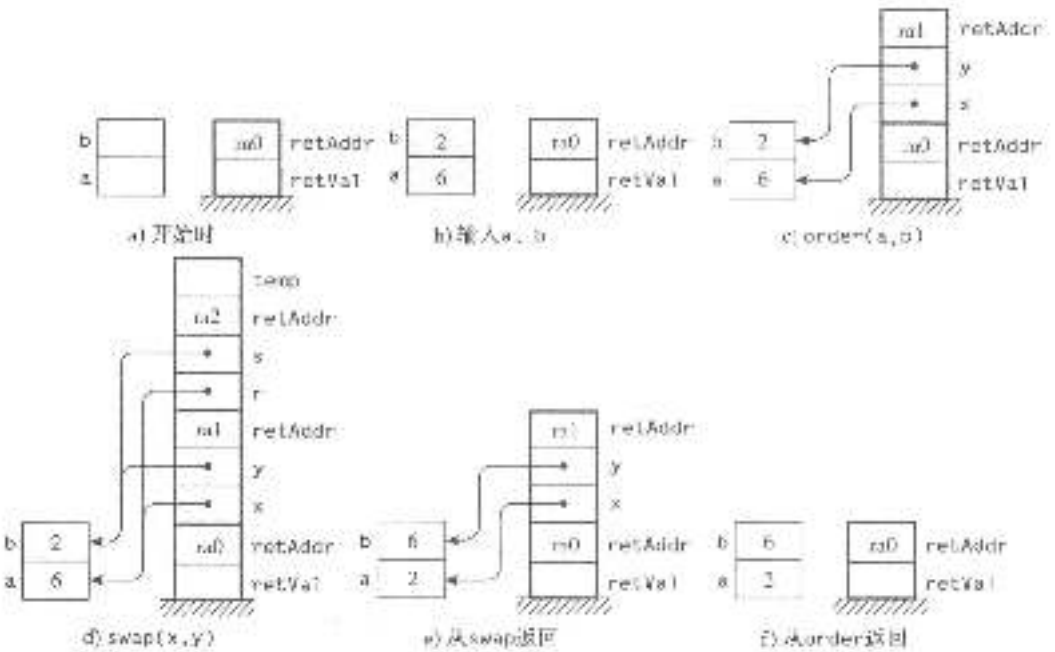


图 2-21　图 2-20 的运行时栈

因为栈是 LIFO 结构，所以在函数结束时，最后一个被压入运行时栈的栈帧将第一个被弹出。因此返回地址将把控制返回到最近的调用函数。运行时栈的 LIFO 属性将是理解 2.4 节中递归的基础。

你可能已经注意到了，main() 是一个总是返回整数的函数。截至目前，所有的程序都向操作系统返回 0。此外，截至目前，所有的主程序函数都没有参数。虽然主程序有参数是很常见的，但是本书中的主程序都没有。为了保持图的简单，后面的图中都将省略主程序的 retVal 和 retAddr。实际的 C++ 编译器必须处理这两者。

2.4　递归

你曾在字典中查找不认识单词的定义时，发现字典恰恰是以另一个不认识的单词来定义它的吗？接着你就查找第二个单词，发现它是用第一个单词来定义的吗？这是一个循环或间接递归的例子。字典的这个问题源于从一开始你就不知道第一个单词的意思。如果第二个单词用你认识的第三个单词来定义，就能得到满意的结果了。

数学上，函数的递归定义（recursive definition）是指函数使用它自己来定义自己。例如，假设函数 $f(n)$ 定义如下：

$$f(n) = nf(n-1)$$

若用这个定义来确定 $f(4)$，就在定义中用 4 代替 n：

$$f(4) = 4f(3)$$

但是现在你不知道 $f(3)$ 是什么，那么在定义中用 3 代替 n，得到：

$$f(3) = 3f(2)$$

把这个代进 $f(4)$ 的公式中，得到：

$$f(4) = 4(3)f(2)$$

但是，现在你不知道 $f(2)$ 是什么，定义告诉你它是 2 乘以 $f(1)$，那么求 $f(4)$ 的公式变为

$$f(4) = 4(3)(2)f(1)$$

可以看到这个定义的问题：没有什么能够结束这个过程，你将无穷尽地计算 $f(4)$。

$$f(4) = 4(3)(2)(0)(-1)(-2)(-3)...$$

这就如同字典给了你一个无穷尽的定义串一样，每个单词都基于另一个你不认识的单词。为了完整，定义必须指定某个特定 n 的 $f(n)$ 值，那么前述过程就能终止，你可以计算出任何 n 的 $f(n)$。

下面是 $f(n)$ 的一个完整递归定义：

$$f(n) = nf(n-1) \quad n>1$$
$$f(1) = 1$$

这个定义说明前述过程可以在 $f(1)$ 停止。因此，$f(4)$ 是

$$\begin{aligned} f(4) &= 4f(3) \\ &= 4(3)f(2) \\ &= 4(3)(2)f(1) \\ &= 4(3)(2)(1) \\ &= 24 \end{aligned}$$

你应该知道这就是阶乘函数的定义。

2.4.1 阶乘函数

C++ 中的递归函数（recursive function）是调用它自己的函数。没有什么有新语法的特殊递归语句需要学习。它在运行时栈上分配存储空间的方法和非递归函数是一样的。唯一的不同是递归函数包括一条调用它自己的语句。

图 2-22 中的函数递归地计算一个数的阶乘，它是 $f(n)$ 递归定义的一个直接应用。

```
#include <iostream>
using namespace std;

int num;

int fact (int n) {
   if (n <= 1) {
      return 1;
   }
   else {
      return n * fact(n - 1); // 调用
   }
}

int main () {
   cout << "Enter a small integer: ";
   cin >> num;
```

图 2-22 递归计算阶乘的函数

```
    cout << "Its factorial is: " << fact (num) << endl; // ra1
    return 0;
}

交互式输入/输出

Enter a small integer: 4
Its factorial is: 24
```

图 2-22　（续）

图 2-23 给出了简化的该程序运行时栈的历史记录。它跟踪了主程序的栈帧。第一个函数调用来自主程序。图 2-23c 展示了第一次调用的栈帧。返回地址是 ra1，它代表主程序中 cout 调用的地址。

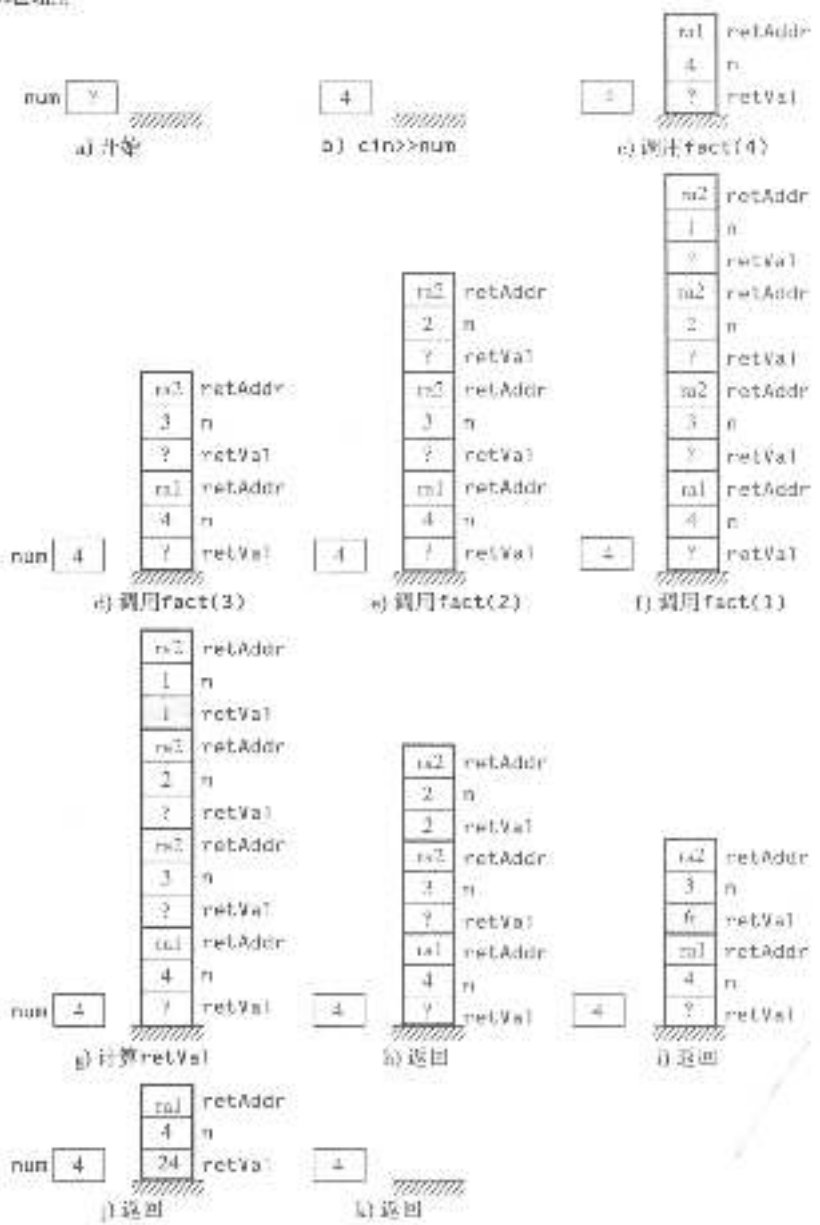


图 2-23　图 2-22 的运行时栈

该函数中第一条语句测试 n 是否为 1。因为 n 的值是 4，所以执行 else 部分。而 else 部分的语句

```
return n * fact (n - 1) // ra2
```

在返回语句的右边包含一个对函数 fact 的调用。

这是一个递归调用，因为它在函数里调用它自己。这个调用和任何其他函数调用所做的事情顺序是一样的。分配新的栈帧，如图 2-23d 所示。第二个栈帧中的返回地址是这个函数中调用语句的地址，由 ra2 来表示。

实参是 n-1。由于图 2-23c 中 n 的值是 4，所以它的值是 3。形参 n 是传值调用的，因此图 2-23d 顶部栈帧的形参 n 赋值为 3。

图 2-23d 展示了一个对递归调用来说很典型的奇特现象。图 2-22 的程序代码中函数 fact 的形参表中只声明了一个 n，但图 2-23d 中 n 出现了两次。n 的旧实例从主程序获得的值 4，而 n 的新实例从递归调用中获得值 3。

计算机暂停该函数旧的执行，并从头开始该函数的一个新的执行。该函数的第一条语句是比较 n 是否等于 1，图 2-23d 中在运行时栈上有 2 个 n。应该比较那个 n 呢？规则是：任何对局部变量或形参的引用指的都是顶部栈帧中的那个。因为 n 的值是 3，所以执行 else 部分。

但是现在函数又进行一次递归调用，分配第三个栈帧，如图 2-23e 所示，接着是第四个，如图 2-23f 所示。每次调用，最新分配的形参 n 的值都比旧值小 1，因为函数调用是

```
fact(n - 1)
```

最后，如图 2-23g 所示，n 的值为 1。该函数给栈上标号为 retVal 的单元赋值为 1，跳过 else 部分，然后终止。这使得控制返回到它的调用语句。

递归返回发生的事情和非递归返回是一样的。retVal 包含返回值，返回地址说明接下来要执行哪条语句。图 2-23g 中，retVal 是 1，返回地址是该函数中的调用的语句。释放顶部栈帧，调用语句

```
return n * fact(n - 1) // ra2
```

完成它的执行。该语句将 n 的值 2 乘以返回值 1，并把这个乘积赋给 retVal。这样 retVal 的值就是 2，如图 2-23h 所示。

每次返回都执行同样的事件序列。图 2-23i、j 展示了从第二次调用返回的值 6，而第一次调用返回的值是 24。图 2-24 展示了图 2-22 程序的调用序列。主程序调用函数 fact，接着函数 fact 调用它自己 3 次。本例中，函数 fact 总共被调用了 4 次。

你可以看到，程序计算 4 的阶乘的方法与从它的递归定义计算 $f(4)$ 的方法一样。计算 $f(4)$ 从 4 乘以 $f(3)$ 开始，接着必须暂停计算 $f(4)$，转而计算 $f(3)$。在得到 $f(3)$ 的值之后，用 4 乘以它就得到 $f(4)$。

类似地，程序必须暂停对该函数的一次执行，再次调用同一个函数。运行时栈跟踪记录变量的当前值，这样当函数实例再继续时，还能使用正确的变量值。

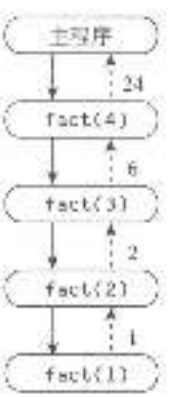


图 2-24　图 2-22 的调用序列。实箭头表示函数调用，虚箭头表示返回，每个返回箭头旁边的是返回值

2.4.2　递归的思考方式

有两种不同的角度来看待递归：微观的和宏观的。图 2-23 是从微观的角度展示的，精

确地给出了执行期间计算机内发生了什么。它考虑的是程序历史记录中运行时栈的细节。宏观的看法不考虑单独的每棵树，它考虑的是整个森林。

56 / 59

为了理解C一怎样实现递归，需要知道微观的看法。在学习Asmb5层怎样实现递归时，必须了解运行时栈的细节。如果只是想写递归函数，应该宏观而不是微观地思考。

写递归函数最难的地方是，必须假设可以调用正在写的程序。为此，你必须忘掉运行时栈，宏观地思考。

数学归纳法证明有助于进行宏观思考。归纳法证明的两个关键元素是

- 建立基础。
- 给定n的公式，证明它对$n+1$是成立的。

同样，设计递归函数的两个关键元素是

- 计算该基础的函数。
- 假设有$n-1$的函数，写出n的函数。

想象你正在写函数fact，写到了这里：

```
int fact (int n) {
  if (n <= 1) {
    return 1;
  }
  else {
```

不知该怎样写下去了。已经计算了基础n=1时的函数，但是现在必须假设能调用函数fact，尽管还没有写完这个函数。必须假设fact(n-1)将返回阶乘的正确值。

这里是必须宏观思考的地方。如果开始想知道fact(n-1)怎样返回正确值，如果栈帧开始在你脑中跳跃，那么这样的思考是不对的。在归纳法证明中，必须假设有n的公式。同样，在写函数fact时，必须假设能调用fact(n-1)，毋庸置疑。

递归程序是基于分治策略的，如果能把一个大问题分解为小问题从而解决它时，这个策略很合适。每次递归调用都使问题变得越来越小，直到程序到达最小的问题，即基础，基础是很容易解决的。

2.4.3　递归加法

这里有递归问题的另一个例子。假设list是一个整数数组，想要递归地求出表中所有

60

整数的和。

第一步是构想出以较小问题来解决大问题的解决方案。如果知道怎样求出list[0]和list[n-1]之间元素的和，可以简单地把这个和加上list[n]，就能得到所有整数的和。

第二步是设计出具有适当参数的函数。这个函数通过调用它自己计算n-1个整数的和来计算n个整数的和。因此参数表里必须有一个参数指明对数组中多少个整数相加。应该得到如下的函数头：

```
int sum (int a[], int n) {
// Returns the sum of the elements of a between a[0] and a[n].
```

怎样建立归纳基础呢？很简单，如果n等于0，那么函数应该把a[0]和a[0]之间的元素求和，一个元素的和就是这个元素a[0]。

现在可以写出

```
if (n == 0) {
  return a[0];
```

```
}
else {
```

现在，宏观地思考。假设 sum(a,n-1) 将返回 a[0] 和 a[n-1] 之间所有整数的和。要有信心！需要做的就是把这个和与 a[n] 相加即可。图 2-25 展示了完成的程序中的这个函数。

```
#include <iostream>
using namespace std;

int list[4];

int sum (int a[], int n) {
// Returns the sum of the elements of a between a[0] and a[n].
   if (n == 0) {
      return a[0];
   }
   else {
      return a[n] + sum(a, n - 1); // ra2
   }
}

int main () {
   cout << "Enter four integers: ";
   cin >> list[0] >> list[1] >> list[2] >> list[3];
   cout << "Their sum is: " << sum(list, 3) << endl; // ra1
   return 0;
}
```

交互式输入 / 输出

```
Enter four integers: 3 2 6 4
Their sum is: 15
```

图 2-25 求两数组中前 n 个数字之和的递归函数

尽管写这个函数时没有从微观角度进行考虑，但是仍然可以跟踪记录运行时栈。图 2-26 展示了对 sum 头两次调用的栈帧。栈帧由返回值、参数 a 和 n 以及返回地址组成。因为这里没有局部变量，所以运行时栈上也没有为它们分配存储空间。

在 C++ 中，数组总是传引用调用的。因此，过程 sum 中的变量 a 引用主程序中的 list。图 2-26b 和 c 中从栈帧中标号为 a 的单元指向标号为 list 的单元的箭头表示 a 引用 list。

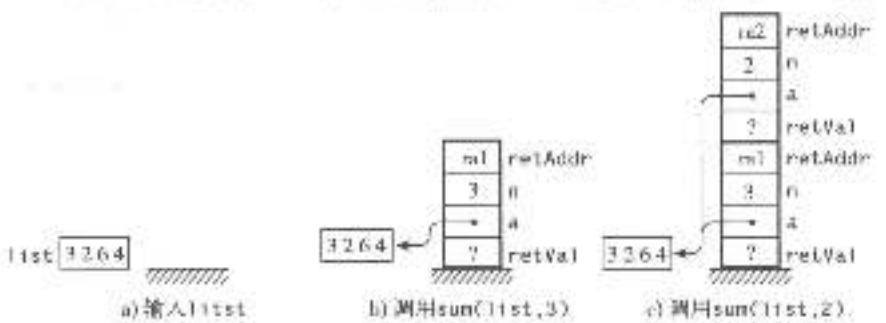


图 2-26 图 2-25 中程序的运行时栈

2.4.4 二项式系数函数

递归函数的下一个例子有一个更加复杂的调用序列。这个函数计算二项式扩展的系数。

考虑如下的扩展：

$$(x+y)^1 = x+y$$
$$(x+y)^2 = x^2+2xy+y^2$$
$$(x+y)^3 = x^3+3x^2y+3xy^2+y^3$$
$$(x+y)^4 = x^4+4x^3y+6x^2y^2+4xy^3+y^4$$

这些项的系数叫作二项式系数（binomial coefficient）。如果不带项只写出这些系数，就形成一个由数值组成的三角，称为帕斯卡三角（Pascal's triangle）。图 2-27 是一个最高到 7 次幂系数的帕斯卡三角。

从图 2-27 可以看到，每个系数是它正上方和左上方系数的和。例如，第 5 行第 2 列的二项式系数 10 等于 4 加 6，6 在 10 的正上方，4 在 10 的左上方。

n 次幂 k 项二项式系数 $b(n, k)$ 的数学表达式是：

$$b(n, k) = b(n-1) + b(n-1, k-1) \quad 0 \leq k \leq n$$

它是一个递归定义，因为函数 $b(n, k)$ 以自己定义了自己。也可以看到，如果 k 等于 0 或者如果 n 等于 k，那么二项式系数的值就是 1。数学表达式：

$$b(n, 0) = 1$$
$$b(k, k) = 1$$

是这个递归函数的归纳基础。

行，n \ 列数，k	0	1	2	3	4	5	6	7
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

图 2-27　二项式系数的帕斯卡三角

图 2-28 是递归计算二项式系数值的程序。该程序主体是立在 $b(n, k)$ 的递归定义之上。图 2-29 是运行时栈的记录。图 2-29b、c 和 d 展示了前 3 个栈帧的分析，分别代表调用 bincoeff(3,1)、bincoeff(2,1) 和 bincoeff(1,1)。第一个栈帧的

```
#include <iostream>
using namespace std;

int binCoeff (int n, int k) {
  int y1, y2;
  if (k == 0) || (n == k)) {
     return 1;
  }
  else {
     y1 = binCoeff (n - 1, k); // ea2
     y2 = binCoeff (n - 1, k - 1); // ea3
     return y1 + y2;
  }
}

int main () {
  cout << "binCoeff (3, 1) = " << binCoeff (3, 1); // ea1
  cout << endl;
  return 0;
}
```

```
输出
binCoeff(3, 1) = 3
```

图 2-28　二项式系数的递归计算

返回地址是主程序中的调用程序，接下来两个栈帧的返回地址是 y1 赋值语句，即标号为 ra2 的那条语句。

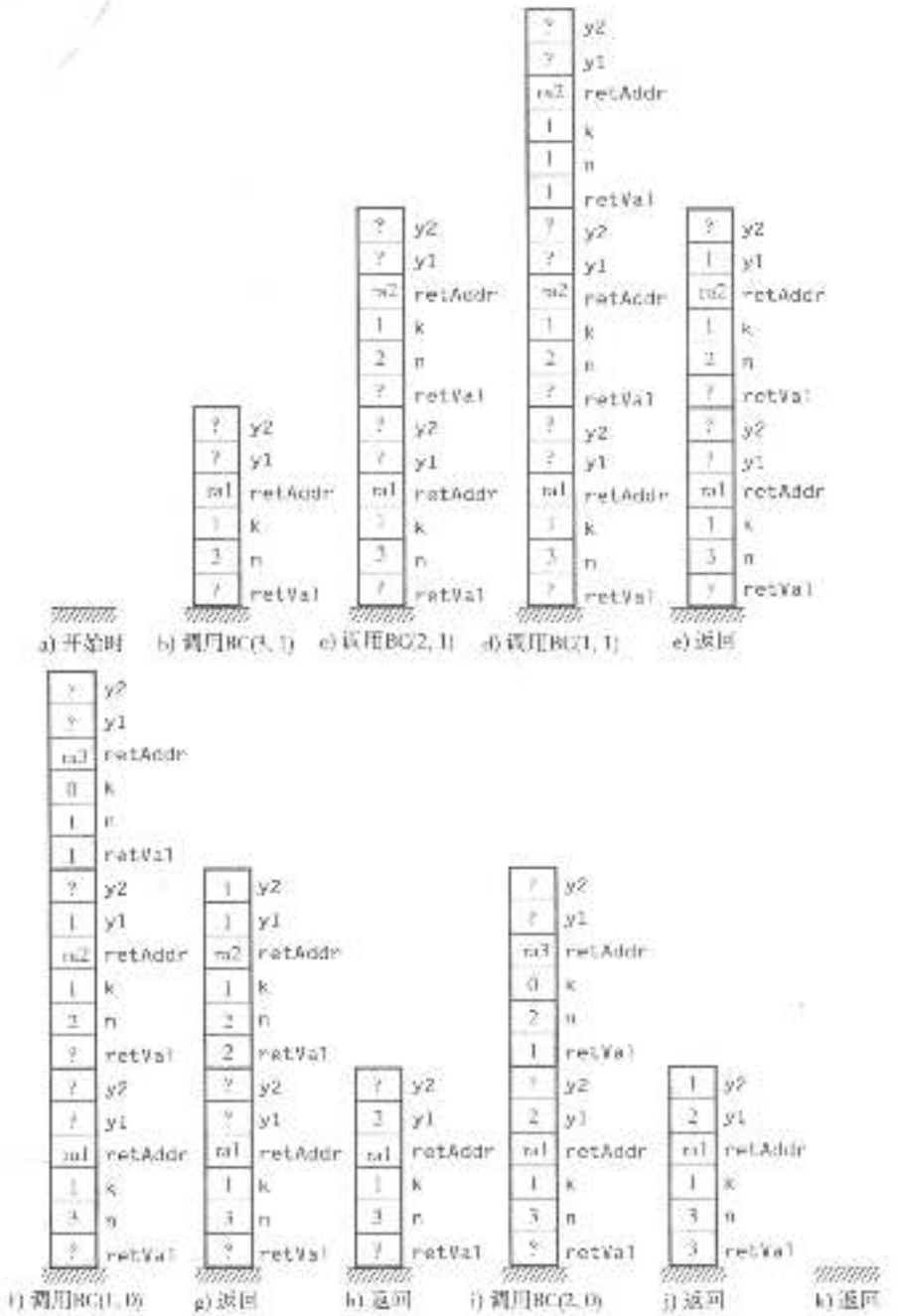


图 2-29 图 2-28 的运行时栈

图 2-29e 展示了从 bincoeff(1,1) 的返回，y1 获得函数的返回值 1，接着 y2 赋值语句调用函数 bincoeff(1,0)。图 2-29f 展示 bincoeff(1,0) 执行期间的运行时栈，每个栈帧都有不同的返回地址。

这个程序的调用序列不同于前面那些递归程序的调用序列。另一个程序是不断分配栈

帧，运行时栈到达最大高度，然后不断释放栈帧直到运行时栈为空。这个程序是分配运行时栈，达到最大高度，但是不会连续把栈里的栈帧都释放。从图 2-29d 到图 2-29e 释放栈帧，但是从图 2-29e 到图 2-29f 分配栈帧；从图 2-29f 到图 2-29g 到图 2-29h 又释放栈帧，而从图 2-29h 到图 2-29i 又分配栈帧。为什么会这样呢？这是因为这个函数有两个递归调用而不是一个。如果基础判断为真，那么函数不执行递归调用；但如果基础判断为假，则函数执行两个递归调用，一个计算 y1，一个计算 y2。图 2-30 展示了该程序的调用序列。可以看到它是树状的。树的每个结点代表一个函数调用。除主程序外，每个结点有两个孩子结点或者没有，分别对应于有两个递归调用或者没有递归调用。

参照图 2-30，调用和返回序列是

```
Main program
    Call BC(3, 1)
        Call BC(2, 1)
            Call BC(1, 1)
        Return to BC(2, 1)
            Call BC(1, 0)
        Return to BC(2, 1)
    Return to BC(3, 1)
        Call BC(2, 0)
    Return to BC(3, 1)
Return to main program
```

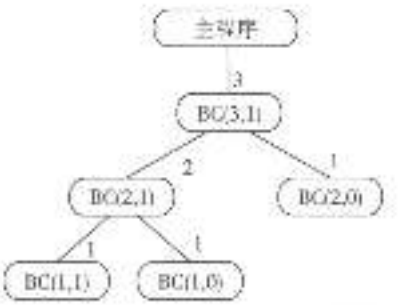


图 2-30　图 2-28 所示程序的调用树

可以把调用树的执行顺序形象化，把调用树想象成海洋的海岸线。一艘船从主程序的左边沿着海岸线开始航行，并一直保持海岸在它的左边。船会按照结点被调用和返回相同的顺序访问结点。图 2-31 给出了访问路径。

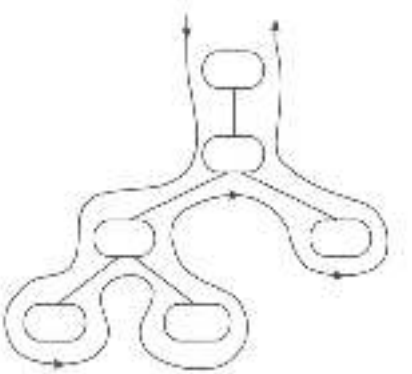


图 2-31　图 2-28 所示程序的执行顺序

当从微观的角度分析递归程序时，在构建运行时栈的记录之前，构建调用树更容易一些。一旦构建了调用树，就很容易看清楚运行时栈的行为。每当船在树中向下访问结点时，程序分配栈帧；每当船在树中向上访问结点时，程序释放栈帧。

可以根据调用树确定运行时栈的最大高度。只需记录到达调用树最低结点时分配的栈帧数量，这个数对应的就是运行时栈的最大高度。

按照执行顺序来画调用树不是最简单的方法。前面那个程序的执行序列从下面开始

```
Main program
    Call BC(3, 1)
        Call BC(2, 1)
            Call BC(1, 1)
```

不应该用这样的顺序来画调用树。从下面这样开始比较容易一些

```
Main program
    Call BC(3, 1)

    Return to BC(3, 1)
```

```
Return to BC(3, 1)
Return to main program
```

从程序代码可以看到，BC(3, 1) 会调用它自己两次：BC(2, 1) 一次，BC(2, 0) 一次。然后回到 BC(2, 1) 确定它的孩子，换句话说，就是要先确定本结点的所有孩子，然后再分析每个孩子的更深层次的调用。

相对于“深度优先”的构造方法，这是用“广度优先”的方法来构造树。在复杂的调用树中多次返回到较高层结点时，深度优先的问题就来了。你可能不记得该结点的执行状态是什么，也就不能确定它的下一个孩子结点是什么。如果一次性确定了一个结点的所有孩子，那么就不需要记得所有结点的执行状态。

61 / 67

2.4.5 逆转数组元素顺序

图 2-32 是一个递归过程而不是函数，它把一个字符数组的元素顺序反转过来。

这个过程把数组 str 中 str[j] 和 str[k] 之间的字符顺序逆转。主程序是想逆转 B 和 d 之间的字符，因此它调用 reverse，参数 j 为 0，k 为 7。

这个过程通过把问题分解成更小的问题来解决。因为 0 小于 7，该过程知道要把 0 和 7 之间的字符逆转，所以它把 str[0] 和 str[7] 互换，接着递归调用自己来交换 str[1] 和 str[6] 之间的字符。如果 j 大于或等于 k，就不需要交换，过程也就什么都不用做。图 2-33 展示了开始时运行时栈的记录。

```
#include <iostream>
using namespace std;

char word[32] = "Backward";

void reverse (char str[], int j, int k) {
   char temp;
   if (j < k) {
      temp = str[j];
      str[j] = str[k];
      str[k] = temp;
      reverse(str, j + 1, k - 1);
   } // ra2
}

int main () {
   reverse (word, 0, 7);
   cout << word << endl; // ra1
   return 0;
}
```

输出

```
drawback
```

图 2-32 逆转数组元素的递归过程

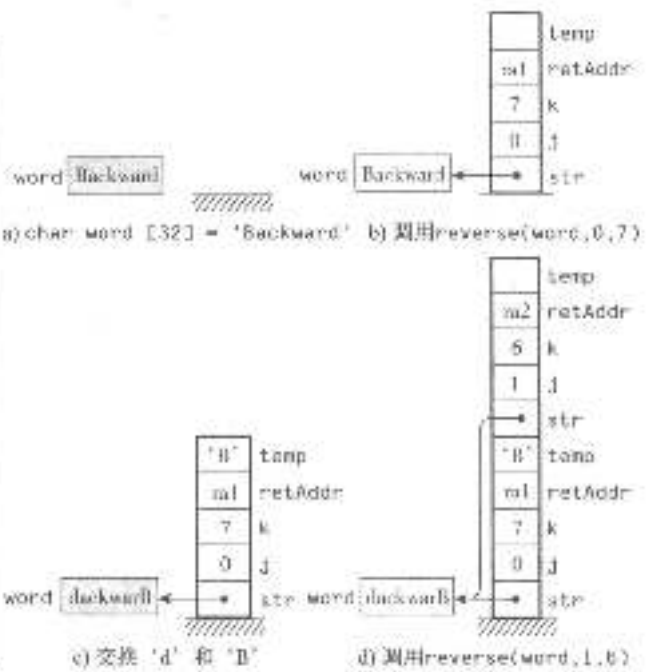


图 2-33 图 2-32 所示程序的运行时栈

2.4.6 汉诺塔

汉诺塔游戏是一个能用递归技巧方便解决的经典计算机科学问题。这个游戏由 3 个柱子和一组直径不同的盘子组成。柱子编号为 1、2 和 3，每个盘子的中央有一个洞，能套在柱

子上。游戏的初始设置是所有的盘子都套在一根柱子上，没有盘子直接放在直径比它小的盘子上。图2-34是4个盘子的初始设置。

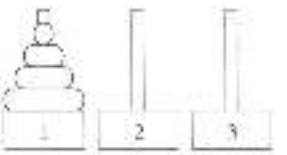


图2-34　汉诺塔问题示意图

要解决的问题是把所有盘子从起始的柱子移到另一根柱子，并遵循下列规则：

- 每次只可以移动一个盘子，只能把一根柱子顶部的盘子移动到另一根柱子顶部。
- 不能把大直径的盘子放在小直径盘子的上面。

解决这个问题的过程有3个参数n、j和k，其中

- n是要移动的盘子数量。
- j是起始柱子。
- k是目标柱子。

68 / 69

j和k是整数，用1标识柱子，给定j和k的值，中间柱子的编号可以用6-j-k计算表示，所谓中间柱子就是既不是起始柱子也不是目标柱子的柱子。例如，如果起始柱子是1而目标柱子是3，那么中间柱子是6-1-3 = 2。

Bjarne Stroustrup



Bjarne Stroustrup 1950年出生于丹麦。虽然并非出身于学术家庭，但他依靠自身的努力从丹麦阿尔胡斯大学（Aarhus University）获得数学专业硕士学位，然后又从剑桥大学（Cambridge University）获得计算机专业博士学位。

Stroustrup小时候并没有接触过计算机，在大学数学系里他才第一次见到计算机。那台计算机占据了一整个房间，他学会了用Algo 60语言在上面编程。他博士期间的工作是用Simula67语言写了一个分布式系统模拟器。依靠写一些其他专职人员交流写出来的小型商业程序的收入，他供自己完成了正规的教育。他认为这段经历帮助他理解了编程在现实世界的重要性。

1979年完成了剑桥的学业后，Stroustrup本来很想到丹麦而，成为了AT&T贝尔实验室在Murry Hill的研究科学家。C语言和UNIX操作系统就是在这个实验室被研发出来并且流行开来的。

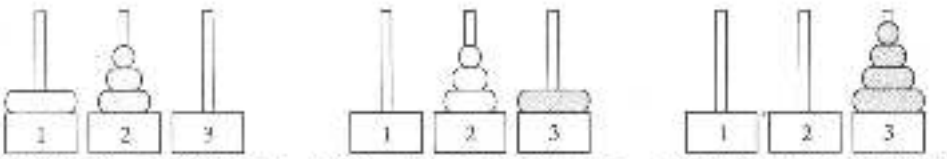
Stroustrup并不满足于当时的编程语言，他发明了一种新的带类的C（C with Classes）语言，在C语言中加入了面向对象的编程特点，就像Simula67中的一样。最终，这个语言演变成了C++。Stroustrup作为C++的发明者为人们所熟知。Stroustrup很早就决定希望他的语言能够支持真实的用户，他了解如果人们不需要去学习一门全新的语言，他的语言就能被广泛接受和使用，所以他使得除了有少数例外，C++与C语言兼容。也就是说，大多数用C语言书写的程序能够用C++编译器进行编译（虽然反过来不行）。这个目标对语言设计来说是限制，但是在它的使用上是大有裨益的。Stroustrup曾经说："即使没有C语言可以与之兼容，我也会选择其他某种语言来兼容的。我曾经，现在依然，相信如果我的时间用来花在创造另一种书写循环的方式上，那就太不值得了。"事实上，C++语言在市场上非常成功。微软的Word、Adobe的Photoshop和Google的搜索引擎都是以C++来编写的。

> Stroustrup 当选了美国国家工程院院士，AT&T 贝尔实验室院士，荣获 ACM 的 Grace Murray Hopper 奖。在本书编写之时，他是德州农工大学（Texas A&M University）计算机科学首席教授（College of Engineering Chair in Computer Science）。
>
> "我总是希望能像使用电话那样简单地使用计算机，我的愿望终于成真了，我不知道该怎样使用电话了。"
>
> ——Bjarne Stroustrup

要把 n 个盘子从柱子 j 移到柱子 k，首先检查是否 n=1。如果是，那么简单地把这个盘子从柱子 j 移到柱子 k 即可。但如果不是，就把问题分解为几个小部分：

- 把 n−1 个盘子从柱子 j 移到中间柱子。
- 把一个盘子从柱子 j 移到柱子 k。
- 把 n−1 个盘子从中间柱子移到柱子 k。

图 2-35 展示了从柱子 1 移动 4 个盘子到柱子 3 的问题的分解。



a) 将3个盘子从柱子1移到柱子2　b) 将1个盘子从柱子1移到柱子3　c) 将3个盘子从柱子2移到柱子3

图 2-35　假设你能把 3 个盘子从一个柱子移动到另一个时，把 4 个盘子从柱子 1 移到柱子 3 的解决方案

假定初始 n 个盘子堆放的顺序是正确的，这样的操作过程保证盘子不会放在比它直径小的盘子上。例如，如图 2-35 所示，要把 4 个盘子从柱子 1 移到柱子 3，这个过程告诉你应该把最上面的 3 个盘子从柱子 1 移到柱子 2，把底部的一个从柱子 1 移到柱子 3，接着再把那 3 个从柱子 2 移到柱子 3。

把最上面的 3 个盘子从柱子 1 移到柱子 2，柱子 1 上就只剩下底部的一个盘子。这个盘子是直径最大的，因此在移动其他盘子的过程中，放在它上面的任何盘子都是更小的。为了把底部这个盘子从柱子 1 移到柱子 3，柱子 3 必须是空的。这样就不会把这个底部的盘子放在一个较小的盘子上。当把那 3 个盘子从柱子 2 移到柱子 3 上时，会把它们放在当前在柱子 3 底部的这个最大的盘子上，这样 3 个盘子就被正确地放在柱子 3 上了。

这个过程是递归的。第一步要把 3 个盘子从柱子 1 移到柱子 2。为此，要把 2 个盘子从柱子 1 移到柱子 3，再把另一个从柱子 1 移到柱子 2，接着把那 2 个从柱子 3 移到柱子 2。图 2-36 展示了这个移动的序列。根据前述推理，能够正确地实施这些步骤。在把 2 个盘子从柱子 1 移到柱子 3 的过程中，可以把这两个盘子中的任意一个放在柱子 1 底部的两个盘子上，不用担心违反规则。



a) 将2个盘子从柱子1移到柱子3　b) 将1个盘子从柱子1移到柱子2　c) 将3个盘子从柱子3移到柱子2

图 2-36　假设你能把两个盘子从一个柱子移动到另一个柱子时，把 3 个盘子从柱子 1 移到柱子 2 的解决方案

最终，可以把这个问题归约到只需移动一个盘子的基础步骤，而一个盘子的解决方案是容易的。本章结尾的一道问题就是对汉诺塔游戏的解决方案进行编程。

2.4.7　相互递归

在有些问题最适合的解决方案中，过程不直接调用它们自己，但是仍然是递归的。假设一个主程序调用过程 a，过程 a 包含一个对过程 b 的调用。如果过程 b 又包含一个对过程 a 的调用，那么 a 和 b 是相互递归的。尽管过程 a 不直接调用它自己，但是通过过程 b，它间接地调用了它自己。

和普通递归相比，相互递归的实现没有什么不同。运行时栈上分配栈帧的方式也是一样的。先分配参数，接着是返回地址，再是局部变量。

不过，在 C++ 程序中，声明相互递归过程时有一个小问题。原因在于 C++ 语言要求程序的声明必须先于它的使用。如果过程 a 调用过程 b，那么在代码中，b 的声明必须在 a 的声明之前；但是如果过程 b 调用过程 a，那么在代码中，a 的声明必须在 b 的声明之前。问题是，如果每个过程都调用另一个，代码中每个程序都必须出现在另一个之前，这显然是不可能的。

针对这种情况，C++ 提供了**函数原型**（function prototype），它允许程序员写出第一个程序的头，而没有过程体。函数原型包括完整的形参列表，但在程序体的位置，放一个分号（;）。在一个过程的函数原型之后，就可以是第二个过程的声明，然后是第一个过程的过程体。

例 2.6　这是刚才讨论的相互递归的过程 a 和 b 的架构：

```
常量、类型、主程序变量
void a (someType x) ;
void b (someOtherType y) {
    b的过程体
}
void a (someType x) {
    a的过程体
}
int main( ) {
    主程序的执行语句
}
```

71 / 72

如果 b 对 a 有一个调用，编译器将会核实实参的数量和类型是否与前面在函数原型里扫描的 a 的形参匹配。如果 a 对 b 有一个调用，那么这个调用会在 a 的程序体中，因为 b 在 a 的代码块之前，因此编译器已经扫描了 b 的声明。□

尽管相互递归不如递归常见，但有一些编译器是基于一种叫递归下降（recursive descent）的技术，这个技术很大程度上使用了相互递归。看看 C++ 语句的结构，你就能明白为什么是这样。把一个 if 嵌套在一个 while 中，而这个 while 又嵌套在另一个 if 中，这是很有可能的。在使用递归下降技术的编译器里有一个过程用于翻译 if 语句，另一个过程用于翻译 while 语句。当正在翻译外层 if 语句的过程遇到 while 语句时，它将调用翻译 while 语句的程序。而当翻译 while 语句的过程遇到嵌套在里面的 if 语句时，它又调用翻译 if 语句的过程，因此是相互递归的。

2.4.8　递归的成本

本节例子的选取只基于一个标准：例子说明递归的能力。可以看到在使用递归的解决方

案中，运行时栈需要大量的存储空间。同时也要花费时间分配和释放栈帧。递归解决方案在空间和时间上都是昂贵的。

如果一个问题有不用递归的简单解法，那么非递归方法通常优于递归方法。图 2-18 中的计算阶乘的非递归函数肯定比图 2-22 的递归阶乘函数好。图 2-25 对数组元素求和与图 2-32 都可以不用递归，只用循环就可以很容易地编程。

二项式系数 $b(n, k)$ 有一个基于阶乘的非递归定义：

$$b(n, k) = \frac{n!}{k!\,(n-k)!}$$

如果非递归地计算阶乘，基于这个定义的程序可能比相应的递归程序效率高很多。两种方法的选择还有一个可以考虑的因素：非递归方法需要乘法和除法，而递归方法仅需要加法。

73

有些问题本质上就是递归的，仅用非递归方法解决非常困难。汉诺塔游戏问题的解决本质上就是递归的。你可以试着不用递归去解决它，看看到底会有多难。快速排序法，最知名的排序算法之一，也属于这种类型，用非递归方法对快速排序进行编程比用递归方法难得多。

2.5 动态内存分配

在 C++ 中，值存储在主存储器的 3 个不同区域：

- 全局变量存储在内存的固定位置。
- 局部变量存储在运行时栈。
- 动态分配的变量存储在堆中。

过程调用和返回时，不能控制堆的分配和释放，而是借助于指针变量来分配堆。堆的分配不是通过过程调用在运行时栈上自动触发，它称为动态内存分配。

2.5.1 指针

当声明一个全局或者局部变量时，指定它的类型。例如，可以把类型指定为整数或字符或数组。类似地，当声明一个指针时，必须声明它所指向的类型。指针本身可以是全局变量，也可以是局部变量，但是它指向的值位于堆中，既不是全局变量也不是局部变量。

C++ 提供了两个运算符来控制动态内存分配：

- `new`，在堆中分配一块空间。
- `delete`，释放堆中的一块空间。

虽然用 `delete` 运算符释放内存非常重要，但本书并不阐述它是怎样进行的。本书中使用指针的程序都是软件设计的坏例子，因为省略了释放的过程。这些程序的目的是展示 HOL6 层和 Asmb5 层之间的关系，到第 6 章这个关系就会变得更明显，因为第 6 章会讲述程序的翻译。

`new` 运算符要求它的右边是类型，该运算执行时做两件事情：

74

- 在堆中分配一个足够大的存储单元用于存放它右边类型的值。
- 返回一个指针，指向新分配的存储空间。

与指针有关的赋值有两种：给指针赋一个值，或者给指针指向的单元赋一个值。第一种赋值叫指针赋值，它按照下列规则执行：

- 如果 p 和 q 是指针，赋值 p = q 使得 p 指向 q 指向的同一单元。

图 2-37 是一个无实际意义的程序，只是为了说明 new 运算符的行为以及指针赋值的规则。它使用全局指针。如果是局部指针，输出也是一样的。如果是局部指针，那么它们会分配在运行时栈上而不是内存的固定位置中。

在全局指针的声明中

```
int *a, *b, *c;
```

变量名前的星号表示这个变量是一个指向整数的指针，而不是整数。图 2-38a 将一个指针的值图形化地表示为了一个小黑点。

图 2-38b 说明了 new 运算符的行为。它在堆上分配了一个足够大的单元来存储整数值，并把这个值返回给指针。这个赋值使得 a 指向新分配的单元。图 2-38c 展示了怎样访问指针指向的单元。因为 a 是指针，所以 *a 是 a 指针指向的单元。图 2-38f 说明了指针赋值规则。赋值 c=a 使 c 指向 a 指向的同一单元；类似地，赋值 a=b 使 a 指向 b 指向的那个单元。在图 2-38h 中，该赋值不是对指针 a 进行赋值，而是对指针 a 指向的单元进行赋值。

```
#include <iostream>
using namespace std;

int *a, *b, *c;

int main () {
   a = new int;
   *a = 5;
   b = new int;
   *b = 3;
   c = a;
   a = b;
   *a = 2 + *c;
   cout << "*a = " << *a << endl;
   cout << "*b = " << *b << endl;
   cout << "*c = " << *c << endl;
   return 0;
}
```

输出

```
*a = 7
*b = 7
*c = 5
```

图 2-37　一个无实际意义的 C++ 程序，只是为了说明指针的类型

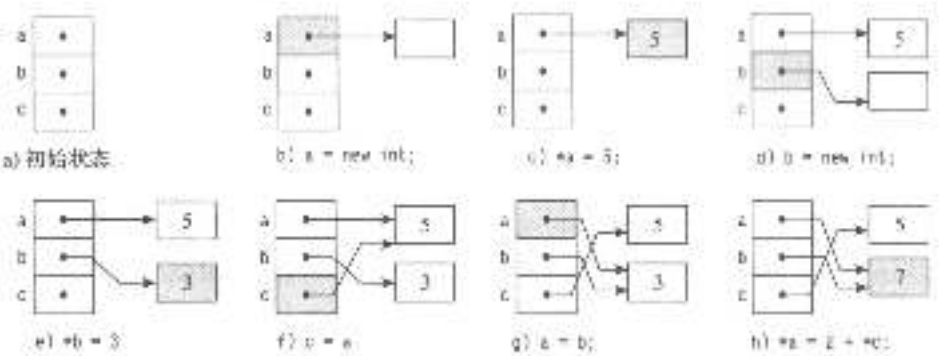


图 2-38　图 2-37 所示程序的历史记录

2.5.2　结构

结构是 C++ 中数据抽象的关键。它们允许程序员把基本类型的变量整合到一个单一的抽象数据类型中。数组和结构都是一组值，不过数组的所有单元要求类型必须是一样的，通过整型数值作为索引访问每个单元；而在结构中，各个单元可以是不同类型的。C++ 提供 struct 结构把多个值集合成一个组。C++ 程序员给每个称为字段的单元一个字段名。

图 2-39 所示的程序声明了一个名为 person 的 struct，它有 4 个字段，分别叫作 first、last、age 和 gender。该程序声明了一个叫 bill 的全局变量，其类型为 person，字段 first、last 和 gender 是 char 类型，字段 age 是 int 类型。

```
#include <iostream>
using namespace std;

struct person {
   char first;
   char last;
   int age;
   char gender;
};
person bill;

int main () {
   cin >> bill.first >> bill.last >> bill.age >> bill.gender;
   cout << "Initials: " << bill.first << bill.last << endl;
   cout << "Age: " << bill.age << endl;
   cout << "Gender: ";
   if (bill.gender == 'm') {
      cout << "male\n";
   }
   else {
      cout << "female\n";
   }
   return 0;
}
```

输入

```
bj 32 m
```

输出

```
Initials: bj
Age: 32
Gender: male
```

图 2-39 C++ 的结构

为了访问结构的字段，在变量和要访问的字段之间放一个句号，例如，if 语句的测试条件

```
if (bill.gender == 'm')
```

将访问变量 bill 的名为 gender 的字段。

2.5.3 链式数据结构

程序员经常把指针和结构结合起来实现链式数据结构。struct 通常称为结点，指针指向结点，结点中又有指针字段。在数据结构中，结点的指针字段作为指向另一个结点的链接。图 2-40 是一个实现链表（linked list）数据结构的程序，第一个循环输入一个整数序列，以特殊的标记符号值 -9999 结束，输入流中的第一个值放在链表的末端；第二个循环输出链表中的每个元素。图 2-41 是图 2-40 所示程序最开始几条语句执行的历史记录。

75
77

指针值为 0 是一个特殊的值，它保证指针不指向任何单元。在 C++ 程序中，它通常用作一个链式结构的标记符号值。语句

```
first = 0;
```

把这个特殊的值赋给局部指针 first。图 2-41b 把这个值图形化地表示为一个虚线三角。

用星号来访问指针指向的单元，用句号来访问结构的字段。如果一个指针指向一个

struct，那么要访问 struct，必须同时使用星号和句号。

例 2.7　下列语句将变量 value 的值赋给变量 first 指向的结构的 data 字段

```
(*first).data = value;
```

因为这种星号和句号的组合太常用了，所以 C++ 提供了箭头运算符 ->。其格式是一个连接符紧接一个大于符号。例 2.7 中的语句可以用这个运算符缩写为

```
first->data = value;
```

如图 2-41f、k 所示。该程序用同样的缩写访问 next 字段，如图 2-41g、l 所示。

```
#include <iostream>
using namespace std;

struct node {
   int data;
   node* next;
};

int main () {
   node *first, *p;
   int value;
   first = 0;
   cin >> value;
   while (value != -9999) {
      p = first;
      first = new node;
      first->data = value;
      first->next = p;
      cin >> value;
   }
   for (p = first; p != 0; p = p->next) {
      cout << p->data << " ";
   }
   return 0;
}
```

输入

```
10 20 30 40 -9999
```

输出

```
40 30 20 10
```

图 2-40　一个输入和输出链表的 C++ 程序

总结

在 C++ 中，值存储在主存储器的 3 个不同区域中：全局变量存放在内存中的固定位置，局部变量存放在运行时栈和堆。改变控制流（顺序的方法有两种：选择和循环。C++ 的 if 和 switch 语句实现选择，while、do 和 for 语句实现循环。所有 5 种语句都用关系运算符测试条件的真伪。

运行时栈的 LIFO 特性用于实现函数和过程调用。函数的分配过程是：压入返回值的存储空间，压入实参，压入返回地址，压入局部变量的存储空间。过程的分配过程除了不压入返回值的存储空间之外都是一样的。栈帧是由一次函数或过程调用中压入运行时栈的所有项组成的。

递归过程是一种调用它自己的过程。为了避免无休止地调用它自己，递归程序必须有一个 if 语句，作为停止递归调用的安全出口。镜观和宏观是两种不同的思考递归的角度。镜观

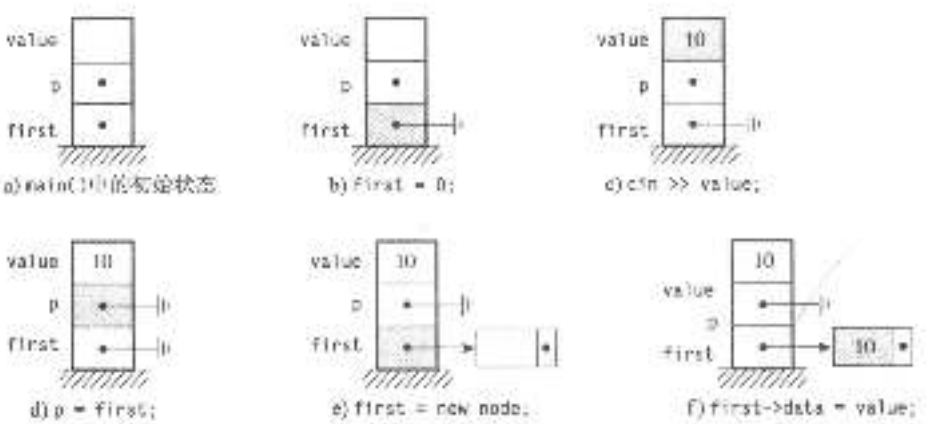


图 2-41　图 2-40 所示程序的前几条语句执行的历史记录

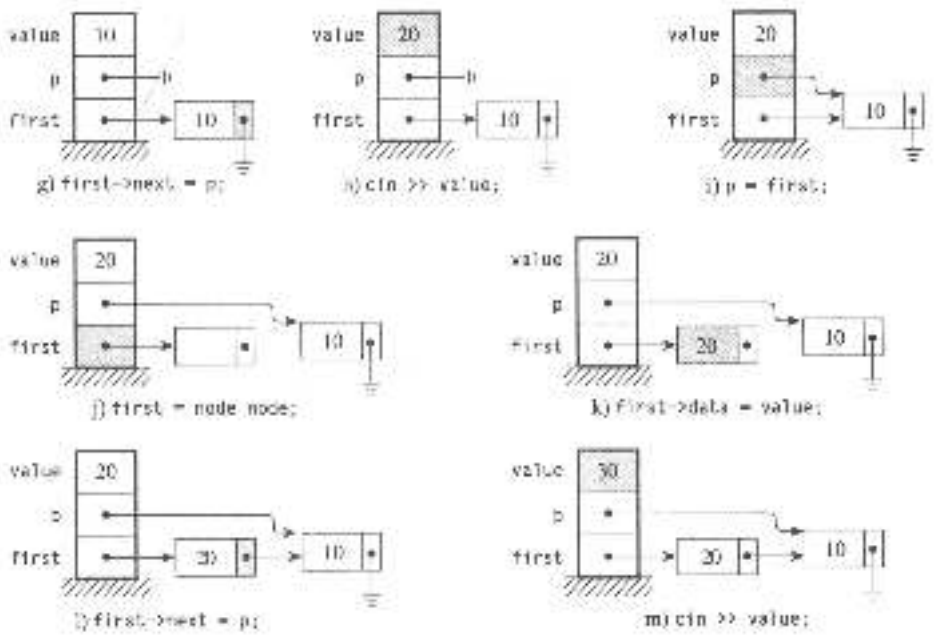


图 2-41 （续）

角度考虑程序执行期间运行时栈的细节，而宏观角度基于更高的抽象层次，与数学归纳法证明相关。微观用于分析，宏观用于设计。

用 new 运算符在堆上分配存储空间称为动态内存分配。new 运算符在堆上分配一个内存单元，并返回一个指向该单元的指针。结构是一组值，它不要求所有的值都是同一类型。每个值存储在一个字段中，每个字段有一个名字。链接数据结构由结点组成。这些结点都是结构，结构中有指针指向其他的结点。链表中的结点有一个值字段，还有一个通常名为 next 的字段，它指向链表中的下一个结点。

78 : 80

练习

2.4 节

1. 主程序第一次调用图 2-25 中的函数 sum，从第二次开始它就自己调用自己。*(a) 该函数总共被调用了多少次？(b) 画出函数第二次被调用后，主程序变量和运行时栈的情况。应该有 3 个栈帧。(c) 画出从 (b) 调用返回前，主程序变量和运行时栈的情况。应该有 3 个栈帧，不过内容和 (b) 的不同。
2. 给出图 2-28 中函数 binCoeff 的调用树，调用树的画法如图 2-30 所示，主程序中的调用语句如下：

```
*(a) binCoeff(4,1)          (b) binCoeff(5,1)
 (c) binCoeff(3,2)          (d) binCoeff(4,4)
 (e) binCoeff(4,2)
```

该函数被调用了多少次？程序执行期间，运行时栈上栈帧的最大数量是多少？程序是按照什么顺序进行调用和返回的？

3. 对练习 2 中的情况，画出从下列函数调用返回前的运行时栈。运行时栈的画法如图 2-29 所示。

```
*(a) binCoeff(2,1)          (b) binCoeff(3,1)
 (c) binCoeff(1,0)          (d) binCoeff(4,4)
 (e) binCoeff(2,1)
```

在 (e) 中，binCoeff(2,1) 被调用两次，画出从第二次对该函数进行调用返回前的运行时栈。

4. 给出图 2-32 中逆转字符串数组字母顺序的程序的调用树，调用树的画法如图 2-30 所示。函数

reverse被调用了多少次？在运行时栈上分配的栈帧最大数量是多少？画出第三次调用函数reverse后的运行时栈。

5. 斐波那契数列是

0 1 1 2 3 5 8 13 21 ...

每个斐波那契数列中的数是数列中它前面两个数之和。数列最开始有两个数，递归地定义为

fib(0) = 0

fib(1) = 1

fib(n) = fib(n − 1) + fib(n − 2)　n > 1

画出下列斐波那契数的调用树：

(a) fib(3)　　(b) fib(4)　　(c) fib(5)

81 对上述每个调用，fib被调用了多少次？在运行时栈上被分配的栈帧最大数量是多少？

6. 对于问题2.15中汉诺塔问题的解决方案，画出4个盘子情况的调用树。程序被调用了多少次？运行时栈上栈帧的最大数量是多少？

7. 神秘数字递归地定义为

myst(0) = 2

myst(1) = 1

myst(n) = 2*myst(n − 1) + myst(n − 2)　n > 1

(a) 画出 myst(4) 的调用序列。(b) myst(4) 的值是什么？

8. 检查下面的C++程序。(a) 画出过程最后一次被调用后的运行时栈。(b) 程序的输出是什么？

```
#include <iostream>
using namespace std;

void what (char word [], int j) {
   if (j > 1) {
      word [j] = word [j - 1];
      what (word, j - 1);
   } // na2
}

int main () {
   char str [5] = "abcd";
   what (str, 3);
   // na1
   cout << str << endl;
   return 0;
}
```

问题

2.1节

9. 写一个C++程序，输入两个整数，输出它们的商和余数。

输入样本

```
13  4
```

输出样本

```
13/4 has value 3
13%4 has value 1
```

82

2.2节

10. 写一个C++程序，输入一个整数，输出这个整数是否是偶数。

输入样本

```
15
```

输出样本

```
15 is not even
```

11. 写一个C++程序，输入两个整数，输出这两个整数之间的整数之和。

输入样本

```
9  12
```

输出样本

```
The sum of the numbers between 9 and 12 inclusive is 42.
```

2.3节

12. 写一个C++函数

```
int rectArea (int len, int wid)
```

返回长len宽wid的矩形的面积。用一个输入矩形的长和宽，输出矩形面积的主程序测试它。在主程序中而不是函数中输出该值。

输入样本

```
6    10
```

输出样本

```
The area of a 6 by 10 rectangle is 60.
```

13. 写一个C++函数

```
void rect (int& ar, int& per, int len, int wid)
```

计算长len宽wid的矩形的面积ar和周长per。用一个输入矩形的长和宽，输出矩形面积和周长的主程序来测试它。在主程序中而不是函数中输出值。

输入样本

```
6    10
```

输出样本

```
Length:  6
Width: 10
Area:  60
Perimeter: 32
```

83

2.4节

14. 写一个C++程序，请用户输入一个小的整数，然后调递归函数返回练习5中定义的斐波那契数值。不要使用循环。在主程序而不是在函数中输出值。

输入/输出样本

```
Which Fibonacci number?  8
The number is 21
```

15. 写一个C++程序打印汉诺塔问题的解决方案。要求用户输入游戏中盘子的数量，所有盘子初始处在某根柱子上，要被移动到哪个柱子上。

输入/输出样本

```
How many disk do you want to move? 3
From which peg? 3
To which peg? 2

Move a disk from peg 3 to peg 2.
Move a disk from peg 3 to peg 1.
Move a disk from peg 2 to peg 1.
Move a disk from peg 3 to peg 2.
Move a disk from peg 1 to peg 3.
Move a disk from peg 1 to peg 2.
Move a disk from peg 3 to peg 2.
```

16. 写一个名为 rotateLeft，返回值为 void 的递归函数，它循环左移一个数组的前 n 个整数。为了循环左移 n 个元素，要递归地循环左移前 n-1 个元素，然后交换最后两个元素。例如，循环左移 5 个元素：

```
50  60  70  80  90
```

先递归地循环左移前 4 个元素：

```
60  70  80  50  90
```

然后交换后两个元素：

```
60  70  80  90  50
```

写一个主程序来测试它，输入一个整数的个数，然后是要循环移动的整数数值。输出是原整数值和循环移动后的值。rotateLeft 中不要使用循环，在主程序而不是函数中输出值。

输入样本

```
5  50 60 70 80 90
```

输出样本

```
Original list: 50  60  70  80  90
Rotated list: 60  70  80  90  50
```

84

17. 写一个函数

```
int maximum (int list[], int n)
```

递归地找出 list[0] 和 list[n] 之间最大的整数。假设数列中至少有一个元素。用主程序测试它，输入是整数数量，接着是整数数值。输出是原整数值，接着是最大整数值。maximum 中不要使用循环。在主程序而不是函数中输出值。

输入样本

```
5  50 30 90 20 60
```

输出样本

```
Original list: 50  30  90  20  60
largest value: 90
```

2.5 节

18. 图 2-40 的程序生成了一个元素顺序与键入时相反的链表。修改程序的第一个循环，使生成的链表和输入顺序一致。不要改动第二个循环。

输入样本

```
10 20 30 40 -9999
```

输出样本

```
10 20 30 40
```

19. 二叉搜索树的结点声明如下：

```
struct node {
   node* leftCh;
   int data;
   node* rightCh;
};
```

leftCh 是指向左子树的指针，rightCh 是指向右子树的指针。写一个 C++ 程序，输入一个整数序列，-9999 为标记符号，把它们插入二叉搜索树。写一个递归过程遍历搜索树，以升序输出这些数值。

输入样本

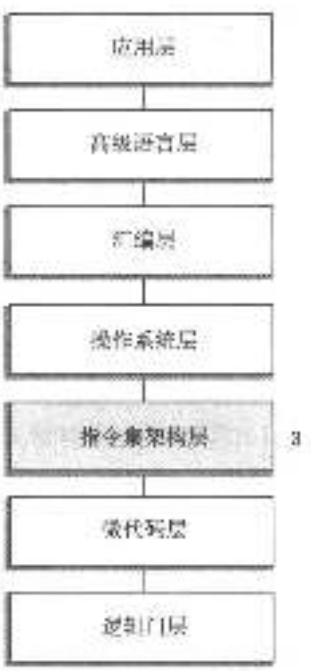
```
40 90 50 10 80 30 70 60 20  9999
```

85
|
86

输出样本

```
10 20 30 40 50 60 70 80 90
```

指令集架构层（第3层）



第 3 章
Computer Systems，Fourth Edition

信息的表示

印刷字是人类最重要的发明之一。我们看到的本书页面上的单词代表存储在纸上的信息，当阅读时，这些信息就传递给我们。就像印刷的页面一样，计算机有存储器用于存储信息。中央处理单元（CPU）能够从存储器获取信息，就像从页面上的单词获取信息一样。

一些计算机术语就是来自这样的类比。CPU 从内存读取（read）信息，把信息写入（write）内存。这些信息被分为字（word）。在一些计算机系统中，一大组字，通常从几百到几千不等，又组成页（page）。

位于 HOL6 层的 C++，信息以存储在内存中的变量或者磁盘中的文件中值的形式存在。本章将展示在 ISA3 层上信息是如何存储的。机器层的信息表示与高级语言级的表示大为不同。在 ISA3 层上，信息表示不太以人为本。我们在后面的章节讨论中间层 Asmb5 层和 OS4 层上的信息表示，以及它们与 HOL6 层和 ISA3 层的关系。

3.1 无符号二进制表示

早期的计算机是机电式的，即所有计算是通过称为继电器（relay）的移动开关来实现的。哈佛大学的 Howard H. Aiken（霍华德·艾肯）于 1944 年建造的 Mark I 计算机就是这种类型的机器。这个项目上，Aiken 获得了国际商用机器公司（IBM）总裁 Thomas J. Watson（托马斯·沃森）的资金支持。当时，Mark I 计算机中的继电器比加法计算器中的机械齿轮计算速度快得多。

87
~
89

甚至在 Mark I 完成之前，艾奥瓦州立大学的 John V. Atanasoff（约翰·阿塔纳索夫）已经构造完成了一台用于解线性方程的电子计算机。1941 年，John W. Mauchly（约翰·莫齐利）访问 Atanasoff 的实验室，1946 年与宾夕法尼亚大学的 J. Presper Eckert（普雷斯伯·埃克特）合作建造了著名的电子数值积分计算机（ENIAC）。相比于 Mark I 继电器的每秒 10 次，ENIAC 的 19 000 个真空管每秒可以执行 5000 次加法。与 ENIAC 一样，现代计算机也是电子的，不过执行计算的是集成电路（IC）而不是真空管。每个 IC 中有数千个与收音机中一样的晶体管。

3.1.1 二进制存储器

电子计算机的存储器不能直接存储数字和字母，它们只能存储电子信号。当 CPU 从内存读取信息时，它检测到一个信号，其电压等于两节手电筒电池产生的电压。

计算机内存有一个最显著的性质：每个存储位置包含一个高电平信号或者低电平信号，绝不会是两者之间的其他信号。存储位置就像是怀孕一样，要么怀了要么没怀，没有折中的可能。

数字（digital）这个词意味着存储在内存中的信号只能有固定数量的数值。二进制意味着仅有两个可能的值。实际上，现在市场上所有计算机都是二进制的，因此每个存储位置包含一个高电平或者一个低电平，每个位置的状态也被描述为开或关，或者描述为包含 1 或 0。

每个存储单元称为二进制数字（binary digit）或位（bit，也译作比特）。1位只能是1或0，绝不可能是其他诸如2、3、A或Z之类。这是基本概念。计算机存储器中存储的每条信息，不管是你的信用卡透支总额还是你的街道地址，都是以二进制1和0的形式存储的。

实际上，计算机存储器中的位被组合在一起形成单元（cell）。例如，一台7位的计算机会以如图3-1所示的每7位组成一组的方式存储它的信息。你可以把单元想作一组方框，每个方框包含一个1或0，除此之外什么都没有。图3-1c的前两行是不可能的，因为有些方框中的值不是0或1，最后一行也是不可能的，因为每个方框必须包含一个值，存储位不能为空。



图3-1 主存中7位的内存单元

不同计算机的每个单元中的位数不同，然而现在大多数计算机的每个单元为8位。本章给出几个单元大小不同的例子来说明普适原理。

诸如数字和字母这样的信息必须以二进制的形式表示才能存储到存储器中。用于存储信息的表示方式叫作编码（code）。本节给出存储无符号整数的编码，本章其他小节描述了存储其他类型数据的编码，下一章将分析存储器中存储程序命令的编码。

3.1.2 整数

数字必须以二进制形式表示才能存储到计算机存储器中。具体的编码视这个数字有小数部分还是整数而定。如果是整数，那么编码也取决于它永远是正的还是也可以为负。

无符号二进制（unsigned binary）用于表示永远为正的整数。在学习二进制之前，我们先复习十进制（decimal或简写dec），然后按这个方法学习二进制。

也许因为我们用10根手指计数和加法而发明了十进制，使用这个优雅规制写的算术书出现在公元8世纪的印度，它被翻译成阿拉伯文并最终被商人带到了欧洲，这里它被从阿拉伯文翻译成拉丁文。因为那时人们认为它起源于阿拉伯，所以数字被称为阿拉伯数字。但是印度-阿拉伯数字应该是更准确的名字，因为它实际上起源于印度。

以10为底的阿拉伯数字计数如下（当然是向下读）：

0	7	14	21	28	35
1	8	15	22	29	36
2	9	16	23	30	37
3	10	17	24	31	38
4	11	18	25	32	.
5	12	19	26	33	.
6	13	20	27	34	.

从0开始，印度人接着发明了下一个数字1的符号，然后是2，直到符号9。这时，他们看着自己的手想到了一个神奇的点子。在最后一根手指后面，他们没有发明新的符号，而是用前面两个符号1和0一起表示下一个数字10。

剩下的故事你都知道了。当到19时，他们看到9是他们创造的符号中最高的符号了，因此他们把它降到0同时把1增加到2，这样形成20。从29到30，最终99到100，不断地

继续，都是同样的处理。

如果我们仅有8个指头而不是10个将会怎样呢？到了7，下一个数字用完了最后一根手指，我们不必发明一个新符号，下一个数字会以10来表示。以8为底（八进制，octal或缩写为oct）计数是这样的：

0	7	16	25	34	43
1	10	17	26	35	44
2	11	20	27	36	45
3	12	21	30	37	46
4	13	22	31	40	.
5	14	23	32	41	.
6	15	24	33	42	.

八进制中77的下一个数字是100。

比较十进制和八进制方法，你会注意到八进制的5（oct）和十进制的5（dec）是同样的数字，但八进制21（oct）和十进制21（dec）是不一样，反而和十进制17（dec）是一样的。数字以八进制表示看上去比它们实际上以十进制表示要显得大一些。

假如我们不是有10个或者8个指头而是有3个指头会怎样呢？规律是一样的，二进制计数是这样的：

0	21	112	210	1001	1022
1	22	120	211	1002	1100
2	100	121	212	1010	1101
10	101	122	220	1011	1102
11	102	200	221	1012	.
12	110	201	222	1020	.
20	111	202	1000	1021	.

最后，我们看看无符号二进制表示。计算机仅有两根手指，以2为底（二进制，binary，或简称bin）计数遵循与八进制和三进制完全相同的方法：

0	111	1110	10101	11100	100011
1	1000	1111	10110	11101	100100
10	1001	10000	10111	11110	100101
11	1010	10001	11000	11111	100110
100	1011	10010	11001	100000	.
101	1100	10011	11010	100001	.
110	1101	10100	11011	100010	.

二进制数看上去比它们实际的值要大很多，二进制数10110（bin）只是十进制的22（dec）。

3.1.3　基本转换

给定一个二进制数，有几种方法来确定它对应的十进制数。一种方法是简单地以二进制和十进制往上数到那个数。这种方法对小的数字非常有效。另一种方法是把二进制数每个为1的位的位置值都加起来。



a) 10110（bin）的位置值　b) 将10110（bin）转换成十进制数

图3-2　将二进制数转换为十进制数

例3.1　图3-2a展示了10110（bin）

的位置值，最右边是1的位置值（称为最低有效位），每个位置值都两倍于它前面的那个位置值。图3-2b展示了得到22（dec）的加法。□

例3.2 无符号二进制类似于我们熟悉的十进制。图3-3展示了58 036（dec）的位置值。数字58 036代表6个1，3个10，没有100，8个1000和5个10 000。从最右边1的位置开始，每个位置值都10倍于它前面的位置值。在二进制中，每个位置值是两倍于它前面的那个位置值。□

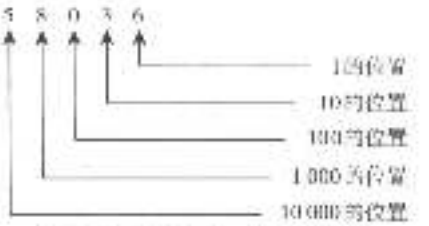


图3-3 58036（dec）的位置值

无符号数值能方便地表示为数制基底的多项式（基底也称为数制的基数（radix））。图3-4给出了10110（bin）和58 036（dec）的多项式表示。最低有效位总是基数的0次方，即1。接下来的有效位是基数的一次方，即基数本身。从多项式的结构可以看到每个位置值是前一个位置值乘以基数。

$1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0$
a）二进制数10110

$5\times10^4+8\times10^3+0\times10^2+3\times10^1+6\times10^0$
b）十进制数58 036

图3-4 无符号数的多项式表示

93

在二进制中，只有1的位置的值是奇数，所有其他位置（2的位置，4的位置，8的位置等）的值都是偶数。如果1的位置的值是0，那么二进制数的值将是几个偶数值相加的和，因此一定是偶数。另一方面，如果二进制数的1的位置的值是1，那么它的值就是把1和几个偶数值相加。因此一定是奇数。与在十进制中一样，你可以通过检测1的位置的数字很容易地确定一个二进制数是奇数还是偶数。

确定十进制数等值的二进制数有点儿棘手。一种方法是连续地把这个数除以2，保留余数的记录，余数的逆序就是这个二进制数。

例3.3 图3-5把22（dec）转换成二进制。22除以2等于11，余数为0，把余数写在右列，接着11除以2等于5余1。继续进行直到商为0，这样形成一个余数列，从下往上读取余数就形成二进制数10110。□

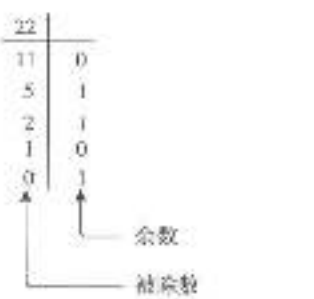


图3-5 将十进制数转化为二进制数

注意：最低有效位是用原始数字除以2得到的余数，这与仅通过检测最低有效位确定一个二进制数是奇数还是偶数是一致的。如果原始数是偶数，除以2将得到余数0，这个0将是最低有效位。相反，如果原始数是奇数，最低有效位将是1。

3.1.4 无符号整数的范围

所有这些基于阿拉伯数字的计数方法都可以表示任意大的数字。然而，一个真实的计算机中每个单元的位数是有限的。图3-6展示了一个7位的单元是如何存储数22（dec）的。注意开头的两个0，它们不影响数值，但是对于指定存储器位置的内容是必需的。对于7位的计算机来说，不带方框，这个数应该写作

001 0110

开头的两个0仍然是必须要有的。本书在显示位串时，从右开始每4位一组，组间有一个空格（为了易读）。



图3-6 7位单元中的数22（dec）

无符号数值的范围取决于一个单元的位数。一个全0的序列代表最小的无符号值，全1的序列代表最大的。

例3.4　一个7位的单元可以存储的最小无符号整数是

94　000 0000（bin）

可以存储的最大无符号整数是

111 1111（bin）

最小的是0（dec），最大的是127（dec）。一个7位单元不能存储大于127的无符号整数。□

3.1.5　无符号加法

无符号二进制数加法和无符号十进制加法一样，只不过更容易，因为只需学习2位加法法则而不是10个数字的加法法则。位加法法则是

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 10

我们熟悉的十进制中的进位技术也一样在二进制中适用。如果一列中的两个数相加大于1，那么必须进1到下一列。

例3.5　假设有一个6位的单元，把01 1010和01 0001两个数相加。简单地从最低有效位列开始把一个数写在另一个上面：

```
  01 1010
ADD 01 0001
-----------
  10 1011
```

注意当到达从右数的第5列时，1 + 1等于10，必须写0并进1到下一列。在最后一列1 + 0 + 0得出和中最左边的1。

为了验证这个进位技术对二进制有效，把这两个数与它们的和都转换为十进制数：

01 1010（bin）= 26（dec）

01 0001（bin）= 17（dec）

01 1011（bin）= 43（dec）

95　毫无疑问，在十进制中，26 + 17 = 43。□

例3.6　这些例子显示了进位可以沿着连续的列传递：

```
  00 0011          00 1111
ADD 01 0001      ADD 00 1001
-----------      -----------
  01 0100          01 1000
```

在第二个例子中，当到达从右数的第四列时，有一个来自前一列的进位，那么1 + 1 + 1等于11，必须写1并进1到下一列。□

3.1.6　进位位

前面例子中的6位单元的表数范围是00 0000到11 1111（bin），或0到63（dec）。两个加数在表数范围内，而和不在是有可能的，这时和就太大了而不能装进6位的存储单元。

为了标记出这种情况，CPU有一个特殊的位称为进位位（carry bit），以字母C表示。当

两个二进制数相加时，如果最左列（最高有效位）的和产生了一个进位，那么C被设置为1，否则C为0。换句话说，C总是包含单元最左列产生的进位。在前面所有的例子中，和都在表数范围内，所以进位位为0。

例3.7 这里有两个显示进位位影响的例子：

```
      01 0110          10 1010
ADD   10 0010    ADD   01 1010
C=0   11 1000    C=1   00 0100
```

在第二个例子中，CPU进行42 + 26的运算，正确的结果是68，但它太大不能装进6位的单元中。注意6位单元的表数范围是0到63，因此只能存储最右的6位，得到不正确的结果4，而进位位也被设置为1，表示最左一列有进位。 □

计算机对两个二进制数的减法是通过加上第二个数的负数来实现的。例如，要计算减法42–26，计算机会计算加法42 + (–26)。因为没有办法存储负数，所以不可能用无符号表示两个整数的减法运算。下一节将描述存储负数的表示法。用这种表示法，C位是加上第二个数的负数的进位。

96

3.2 补码二进制表示

无符号二进制表示法仅适用于非负整数。如果计算机要处理负整数，那么它必须要用一种不同的表示法。

假设有一个6位的单元要存储–5（dec）。因为5（dec）是101（bin），所以你可能想到图3-7所示的样子。但这是不可能的，因为所有的位必须是0或1。牢记计算机是二进制的。上面这种存储值要求每个方框可以存储0或1或破折号，这样的计算机必须是三进制而不是二进制。

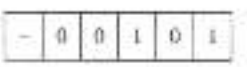


图3-7 尝试在二进制中存储负数

解决这个问题的办法是保留单元的第一个方框来表示符号。这样，6位单元将分为两个部分——1位符号位和5位数值位，如图3-8所示。因为符号位必须是0或1，所以一种可能是让符号位0表示正数，符号位1表示负数。那么+5可以表示为：

00 0101

–5可以用表示为：

10 0101

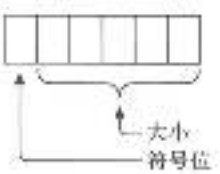


图3-8 有符号整数的结构

在这种表示法中，–5和–5的数值位是相同的，仅仅是符号位不同。

然而，很少有计算机用前面这种代码，因为如果进行十进制的+5加–5，得到0，但如果进行二进制的00 0101加10 0101（符号位和所有），得到

```
      00 0101
ADD   10 0101
C=0   10 1010
```

这显然不等于0。如果CPU硬件可以用无符号二进制加法的普通算法对包含符号位的完整数字–5和–5进行相加并得到0，这会更加方便。

补码（two's complement）二进制表示有这个特性，正数由符号位0和与无符号二进制表示一样的数值位组成。例如，十进制+5（dec）仍然表示为00 0101。

但是 -5（dec）的表示不是 10 0101，而是 11 1011，因为 15 加 -5 是

```
      00 0101
ADD   11 1011
C = 1 00 0000
```

97 注意 6 位的和都是 0，和我们期望的一样。

按照 6 位单元的二进制加法法则，11 1011 称为 00 0101 的*加法逆元*（additive reverse）。求加法逆元的运算称为*求补*（negation），缩写为 NEG。对一个数求补也称为取它的*补码*。

现在我们所需的是求一个数补码的法则。有个简单的法则是基于*反码*（ones'complement），反码是把所有的 1 变为 0，0 变为 1 的二进制序列。反码也称为 NOT 运算。

例 3.8　假设一个 6 位单元，00 0101 的反码是

NOT 00 0101 = 11 1010 □

按出补码规则的线索是把一个数和它的二进制反码相加的结果。因为 1 加 0 等于 1，0 加 1 等于 1，所以任意数和它的反码相加将生成一个全 1 的序列，然而把一个单独的 1 和全 1 的数相加就会得到一个全 0 的数。

例 3.9　00 0101 加上它的反码

```
      00 0101
ADD   11 1011
C = 1 11 1111
```

得到是全 1，再把这个数加 1 得到

```
      11 1111
ADD   00 0001
C = 1 00 0000
```

这个数是全 0 的。 □

换句话说，把一个数加上它的反码，再加 1 就得到全 0 的数，因此一个数的反码加 1 一定是它的补码。

例 3.10　将 00 0101 的反码加 1 得到它的补码。

NOT　00 0101 = 11 1010

```
      11 1010
ADD   00 0001
98    11 1011
```

00 0101 的补码是 11 1011，即，

NEG 00 0101 = 11 1011

11 1011 的确是 00 0101 的负数，因为如前所示两者相加为 0。 □

不管一个数有多少位，对一个数求补的一般法则是：

- 一个数的补码等于它的反码加 1。

或者用 NEG 和 NOT 运算表示为

- NEG x = 1 + NOT x

在我们熟悉的十进制中，如果对一个负数值取负，那么将得到一个正值，代数表达式为

$$-(-x) = x$$

这里 x 为正值。如果求补码的法则是有效的，那么一个负数值的补码应该是相应的正值。

例 3.11 如果对 -5（dec）取补码会怎样？

NOT 11 1011 = 00 0100

```
    00 0100
ADD 00 0001
    00 0101
```

瞧！如你所期望的，又得到了 +5（dec）。 □

3.2.1 补码的表数范围

假定有一个 4 位单元以补码形式存储整数，那么这个单元能表示的整数范围是多少？

数值最大的正整数是 0111（bin），即 17（dec）。这和在无符号二进制数中最大值为 1111 不同，因为第一位被保留作为符号位且一定是 0。在无符号二进制中，用 4 位单元可以存储的最大数是 115（dec），所有 4 位都用于存储数值。在补码表示中，能存储的最大数只是 +7（dec），因为仅有 3 位用于数值。

数值最大的负整数是什么呢？这个问题的答案可能不太显而易见。图 3-9 展示了最大到 7 的每个正整数的补码，在图中看到了什么规律没有？

十 进 制	二 进 制
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

图 3-9 4 位计算机中取补码的结果

我们注意到补码运算自动在负整数的符号位生成了一个 1，它本来就应该这样。偶数仍然以 0 结尾，奇数仍然以 1 结尾。

而且，如你所期望的，二进制 -6 加 1 得到 -5，类似地，二进制 -7 加 1 得到 -6。我们可以从 4 位给出一个更大的负整数 -8。二进制 -8 加 1 得到 -7，因此 -8 应该用 1000 表示。图 3-10 是一个 4 位存储单元的完整有符号整数表。

-8（dec）有一个其他负整数没有的属性，如果取 -7 的补码会得到 -7，如下所示：

NOT 1001 = 0110

```
    0110
ADD 0001
    0111
```

但如果取 -8 的补码，得到的还是 -8：

NOT 1000 = 0111

```
    0111
ADD 0001
    1000
```

因为 4 位无法表示 +8。

我们已经确定了 4 位单元的补码二进制表数范围以二进制表达是 1000 到 0111，或者以十进制表达是 -8 到 +7。

不管单元包含多少位，模式都是一样的。最大的正整数是一个 0 后面全是 1，而数值最大的负整数是一个 1 后面全是 0，它的数值比最大正整数大 1。-1（dec）用全 1 表示。

十 进 制	二 进 制
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

图 3-10 4 位单元的有符号整数

例 3.12　6 位补码表示的表数范围，用二进制表示是

10 0000 到 01 1111

或者十进制表示是

−32 到 31

和其他的负整数不同，10 0000 的补码就是它自己 10 0000。还可以看到 −1（dec）= 11 1111（bin）。□

3.2.2　基数转换

把一个负数从十进制转换为二进制分为两步。首先，把它的数值部分当作无符号表示转换为二进制；其次，按照补码的方式对它取反。

例 3.13　对于 10 位单元的 −7（dec）

+7（dec）= 00 0000 0111（bin）

NOT　00 0000 0111 = 11 1111 1000

```
     11 1111 1000
ADD  00 0000 0001
-------------------
     11 1111 1001
```

因此 −7（dec）的补码是 11 1111 1001（bin）。□

在采用补码表示的计算机中，把一个数从二进制转换成十进制，总是首先检测符号位。如果是 0，这个数是正数，可以按照无符号数表示进行转换；如果是 1，这个数是负数，可选的方法有两种。一种是通过取反得到正数，然后再按照无符号数的法则，把它转换成十进制。

99 / 101

例 3.14　一个 10 位单元的内容为 11 1101 1010，它代表的十进制数是什么？符号位是 1，因此这个数是负数，首先对这个数取反：

NOT　11 1101 1010 = 00 0010 0101

```
     00 0010 0101
ADD  00 0000 0001
-------------------
     00 0010 0110
```

00 0010 0110（bin）= 32 + 4 + 2 = 38（dec）

因此原始的二进制数一定是 38 的负值，即

11 1101 1010（bin）= −38（dec）□

另一种方法是不用取补码，直接转换，即只用在原始二进制数中为 0 的那些位的位置值相加，再加 1。这个方法是正确的，因为取正整数补码的第一步就是按位取反。那些本来对正整数数值有贡献的 1 变为了 0，所以是 0 的不是 1 对负整数的数值有贡献。

例 3.15　图 3-11 显示的是 11 1101 1010（bin）的为 0 的位置，把这些位的位置值之和加 1 得到

1101 1010（bin）= −（1 + 32 + 4 + 1）= −38（dec）

这和前面方法得到的结果是一样的。□

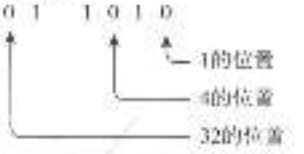


图 3-11　11 1101 1010（bin）中所有 0 的位置值

3.2.3　数轴

看待二进制表示的另一种方法是使用数轴。图 3-12 展示了 3 位单元的无符号二进制表示的数轴，能够表示 8 个数字。

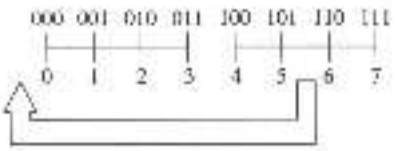
可以通过在数轴上往右移动进行加法运算。例如，4加3，从4开始往右移动3个位置得到7。如果尝试在数轴上6加3，将超出最右端。如果用二进制进行这个加法，将得到不正确的结果，因为结果超出了范围：

```
      110
ADD   011
C=1   001
```

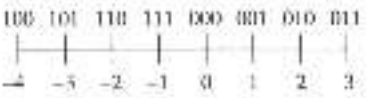


图3-12 3位无符号系统的数轴

把无符号数轴在3到4之间断开，把右半部分移到左边，这样可以得到补码的数轴。图3-13显示了二进制数111现在和000相邻。之前是+7（dec），现在是-1（dec）。



a）把数轴从中间断开



b）将右边一部分移到左边

图3-13 3位补码系统的数轴

即使有经过0，加法仍然是通过在数轴上向右移动来进行的。-2加3，从-2开始向右移动3个位置得到1。如果用二进制计算，答案在范围内，因此是正确的：

```
      110
ADD   011
C=1   001
```

这些位和无符号二进制6加3是一样的。我们注意到结果虽然在表数范围内，但是进位位是1。在补码表示中，进位位不再表示加法的结果是否是在范围内。

102
↓
103

有时候，只考虑移动后的十进制数轴，可以完全避免二进制表示。图3-14展示了补码数轴，用等值的无符号十进制数代替二进制数，这个例子中，每个存储位置有3位，因此最多有2^3或8个数。

现在，从0到3的无符号数与有符号数是一样的。此外，通过减8可以从无符号数得到有符号负数：

```
7-8 = -1
6-8 = -2
5-8 = -3
4-8 = -4
```

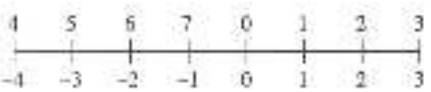


图3-14 无符号十进制数的补码数轴

例3.16 假定有一个8位单元，有2^8或256个可能的整数值，非负数从0到127。假设采用补码表示，97加45等于多少？以无符号二进制数计算，和等于

97 + 45 = 142（dec，unsigned）

但用在补码表示中，这个和为

142 - 256 = -114（dec，signed）

注意，这样的结果完全避免了二进制表示。为了验证结果，首先把97和45转换为二进制并相加：

97（dec）= 0110 0001（bin）

45（dec）= 0010 1101（bin）

```
      0110 0001
ADD   0010 1101
C=0   1000 1110
```

符号位是1，因此这是一个负数。现在，确定它的数值大小

$$\text{NEG } 1000\ 1110 = 0111\ 0010\,(\text{bin})$$
$$= 114\,(\text{dec})$$ □

104 得出预期的结果。

3.2.4 溢出位

在ISA3层上二进制存储的一个主要特性就是每个值都没有与之相关的类型。在前面的例子中，和1000 1110，作为无符号数解释时，它是142（dec），但作为补码表示解释时，它就是−114（dec）。尽管位模式的值取决于它的类型，是无符号还是补码，但是硬件对两种类型不加以区分，只存储位模式。

当CPU对两个存储单元的内容相加时，它不管它们的类型，只采用位序列上的二进制加法法则。对于无符号二进制，如果和超出表数范围，硬件只是简单存储（不正确的）结果，相应地设置C位并继续往下走。由软件来检测相加后的C位，看是否在最高位的那一列有进位发生，并按需采取适当的动作。

我们前面说过，在补码二进制表示中，进位位不再表示一个和是否在表数范围内。当运算结果超出表数范围时，我们称为出现了溢出错误（overflow condition）。为了为有符号数标示这种情况，CPU中有另一个用V表示的特殊位，叫作溢出位（overflow bit）。当CPU对两个二进制整数相加时，如果和超出补码表示的范围，那么V设置为1，否则V设置为0。

不论以何种方式解读位模式，CPU总是执行同样的加法运算。与C位一样，如果发生了补码溢出，CPU不会停下来，它将V位设置为1，并继续它的下一步工作。由软件来检测加法后的V位。

例3.17　这里有几个6位单元的例子，展示了进位位和V位的情况：

两个正数相加：

```
      00 0011          01 0110
ADD   01 0101    ADD   00 1100
-----------      -----------
V = 0 01 1000    V = 1 10 0010
C = 0            C = 0
```

一个正数和一个负数相加：

```
      00 0101          00 1000
ADD   11 0111    ADD   11 1010
-----------      -----------
V = 0 11 1100    V = 0 00 0010
C = 0            C = 1
```

两个负数相加：

```
      11 1010          10 0110
ADD   11 0111    ADD   10 0010
-----------      -----------
V = 0 11 0001    V = 1 00 1000
C = 1            C = 1
```

105 注意V和C的值有可能是所有的组合。 □

怎样才能知道发生了溢出情况呢？一种方法是把两个数转换到十进制，两者相加看它们的和是否超出了以十进制表示的范围。如果是，那就是发生了溢出。

硬件通过将符号位的进位与C位比较来检测是否溢出发生：如果两者不同，溢出发生，V为1；如果两者相同，V为0。

不将符号位的进位与C位比较，也可以通过查看加数与和的符号，直接确定是否发生了溢出。如果两个正数相加得到负数或者两个负数相加得到正数，那么就发生了溢出；一个正数和一个负数相加是不可能发生溢出的。

3.2.5 负数和零位

除了检测无符号整数溢出情况的C位和检测有符号整数溢出情况的V位外，CPU还维护了另外两位，供软件在运算后进行检测。它们是N位和Z位；N位用于检测负数结果，Z位用于检测零结果。总的来说，这4个状态位的函数是

- N = 1，如果结果是负数。
  N = 0，其他情况。
- Z = 1，如果结果全是零。
  Z = 0，其他情况。
- V = 1，如果有符号整数溢出发生。
  V = 0，其他情况。
- C = 1，如果无符号整数溢出发生。
  C = 0，其他情况。

由于N位是符号位的一个副本，所以硬件很容易确定它。而要确定Z位，硬件则要费点儿工夫，因为它必须确定结果的每个位是否都为0。第10章展示了硬件怎样根据结果计算状态位。

例3.18 这里有三个加法例子，展示了结果的4个状态位情况。

	01 0110		00 1000		00 1101
ADD	00 1100	ADD	11 1010	ADD	11 0011
N = 0	10 0010	N = 0	00 0010	N = 0	00 0000
Z = 0		Z = 0		Z = 1	
V = 1		V = 0		V = 0	
C = 0		C = 1		C = 1	

□ 106

3.3 二进制运算

因为计算机中所有信息都是以二进制存储的，所以CPU就用二进制运算来处理这些信息。前面章节提到了NOT、ADD和NEG这些二进制运算：NOT是逻辑运算符，ADD和NEG是算术运算符。本节我们再讲一些其他的在计算机CPU中有用的逻辑和算术运算符。

3.3.1 逻辑运算符

我们熟悉逻辑运算AND和OR。另一个逻辑运算符是异或，写作XOR。若p为真，或者q为真，但不同时为真，那么p和q的异或逻辑值为真。即，p一定真异于q，或者q必须真异于p。

二进制数字一个有趣的属性是可以把它们作为逻辑值来解读。在ISA3层，位为1表示真，位为0表示假。图3-15展示了AND、OR和XOR运算符在ISA3层的真值表。

p	q	p AND q
0	0	0
0	1	0
1	0	0
1	1	1

a)

p	q	p OR q
0	0	0
0	1	1
1	0	1
1	1	1

b)

p	q	p XOR q
0	0	0
0	1	1
1	0	1
1	1	0

c)

图3-15 ISA3层上AND、OR和XOR运算符的真值表

在HOL6层，AND和OR对值为真或假的布尔表达式运算，用在if语句和循环中来测试控制语句执行的条件。下面的C++语句是AND运算符的一个例子

```
if ((ch >= 'a') && (ch <= 'z'))
```

图3-16是AND、OR和XOR在HOL6层的真值表，它和图3-15是一样的，ISA3层的1对应HOL6层的true（真），ISA3层的0对应HOL6层的false（假）。

p	q	p AND q
true	true	true
true	false	false
false	true	false
false	false	false

a)

p	q	p OR q
true	true	true
true	false	true
false	true	true
false	false	false

b)

p	q	p XOR q
true	true	false
true	false	true
false	true	true
false	false	false

c)

图3-16　HOL6层上AND、OR和XOR运算符的真值表

由于不涉及进位，所以逻辑运算比加法更容易执行。对序列中的对应位进行逐位运算。逻辑运算对进位位和溢出位都没有影响。

例3.19　一些6位单元的例子层

```
      01 1010          01 1010          01 1010
ADD   01 0001    OR    01 0001    XOR   01 0001
N = 0 01 0000    N = 0 01 1011    N = 0 00 1011
Z = 0            Z = 0            Z = 0
```

注意，如果把1与1做AND运算，结果是1，没有进位。 □

每个AND、OR和XOR运算用两组位来产生结果，不过NEG运算仅对一组位进行，因此称为*一元运算*（unary operation）。

3.3.2　寄存器传送语言

寄存器传送语言（RTL）的目的是精确指定硬件操作的结果。如果学习过逻辑学，你会熟悉RTL符号。图3-17展示了RTL符号。

在逻辑学中，AND和OR运算称为合取（conjunction）和析取（disjunction）。NOT运算符称为否定（negation）。蕴含（implies）运算符可以翻译为英语"if/then"（中文"如果/那么"）。传递（transfer）运算符是与C++中赋值运算符=等效的硬件。运算符左边的内存单元获得运算符右边的量。位索引运算符把内存单元当做数组，最左边的位是索引0，与C++索引数组元素一样。当形式化描述不够时，可以用非形式化的语言描述，用大括号括起来。

有两种分隔符：一个是顺序分隔符（sequential separator）（分号），用来分隔一个接一个发生的两个动作；另一个是并发分隔符（concurrent separator）（逗号）用来分隔同时发生的两个动作。

例3.20　在例3.19的第3个计算中，假设第一个6位单元用a表示，第二个6位单元用b

操　作	RTL符号
AND	∧
OR	∨
XOR	⊕
NOT	¬
Implies	→
Transfer	←
Bit index	⟨⟩
Informal description	{}
Sequential separator	;
Concurrent separator	,

图3-17　寄存器传送语言的运算及其符号表示

表示，结果为 c，那么 XOR 运算的 RTL 表述是

$$c \leftarrow a \oplus b; N \leftarrow c < 0, Z \leftarrow c = 0$$

首先，c 获得 a 和 b 的异或，这个动作完成后，下面这两个动作同时发生：N 获得一个布尔值，Z 获得一个布尔值。当 $c < 0$ 时，布尔表达式 $c < 0$ 为 1，否则为 0。 □

3.3.3　算术运算符

另外还有两个一元运算符：ASL 表示算术左移（arithmetic shift left）和 ASR 表示算术右移（arithmetic shift right）。如同 ASL 这个名字暗示的，单元中每位往左移动一个位置，最左边的位移动到进位位，而最右边的位得到 0。图 3-18 展示了一个 6 位单元的 ASL 运算的动作。

例 3.21　下面是 3 个算术左移的例子：

ASL 11 1100 = 11 1000,　　N = 1, Z = 0, V = 0, C = 1

ASL 00 0011 = 00 0110,　　N = 0, Z = 0, V = 0, C = 0

ASL 01 0110 = 10 1100,　　N = 1, Z = 0, V = 1, C = 0 □

107 ~ 109

这个运算称为算术移位，因为当这些位用作整数表示时，它的结果类似于算术操作。假设用无符号二进制表示，前面例子中 3 个整数在移动前是

60　3　22（dec，unsigned）

移动后成为

56　6　44（dec，unsigned）

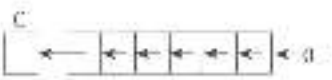


图 3-18　6 位单元的 ASL 运算的行为

ASL 的结果是原数的 2 倍。因为 120 超出 6 位单元能表示的整数范围，所以 ASL 不能把 60 翻倍。当把二进制序列看作无符号整数时，如果移动后进位位是 1，则发生溢出。

在十进制中，左移产生同样的结果，只是整数被乘以 10 而不是 2。例如，对 356 进行十进制的 ASL 会得到 3560，它是原数值的 10 倍。

如果把数解释为补码表示会是什么情况呢？那么移动前 3 个整数是

−4　3　22（dec，signed）

移动后成为

−8　6　−20（dec，signed）

同样，尽管是负数，ASL 的结果仍然是原数的 2 倍。这次，ASL 不能把 22 翻倍，假定用补码表示，44 超出了范围。这个溢出情况使得 V 位被设置为 1。这个情形与加法运算中 C 位检测到无符号值溢出相似，需要用 V 位来检测有符号值的溢出。

对 6 位单元 r 进行算术左移的 RTL 表述为

$$C \leftarrow r\langle 0\rangle, r\langle 0..4\rangle \leftarrow r\langle 1..5\rangle, r\langle 5\rangle \leftarrow 0;$$

$$N \leftarrow r\langle 0\rangle, Z \leftarrow r = 0, V \leftarrow \{溢出\}$$

同时，C 获得 r 最左边的位，r 最左边的 5 位直接获得它们紧邻着的右边位的值，最右边一位获得 0。移位之后，根据 r 的新值设置状态位 N、Z 和 V。区分分号和逗号是很重要的：分号隔开两个事件，每个事件有 3 个部分；在每个部分内，逗号隔开同时发生的事件。大括号非形式化地表示当把值当作有符号整数时，根据结果是否溢出对 V 位进行设置。

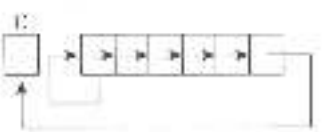


图 3-19　6 位单元上的 ASR 运算的行为

在 ASR 运算中，组中每个位往右移动一个位置，最低有效位移到进位位，最高有效位保持不变。图 3-19

110 展示了一个6位单元的ASR运算的动作。ASR运算不会影响V位。

例3.22　下面是4个算术右移的例子。

ASR 01 0100 = 00 1010,　　N = 0, Z = 0, C = 0
ASR 01 0111 = 00 1011,　　N = 0, Z = 0, C = 1
ASR 11 0010 = 11 1001,　　N = 1, Z = 0, C = 0
ASR 11 0101 = 11 1010,　　N = 1, Z = 0, C = 1　□

ASR运算是特意为补码表示设计的，因为符号位保持不变，负数仍然是负数，正数仍然是正数。

往左移1位是原数乘以2，反之往右移1位是原数除以2。在前面例子中，移动前4个数为

20　23　−14　−11　(dec, signed)

移动后是

10　11　−7　−6　(dec, signed)

偶数正好可以被2整除，因此ASR对它们的结果没什么疑问。当奇数除以2时，结果总是向下取整。例如，23÷2 = 11.5，11.5向下取整为11，同样，−11÷2 = −5.5，−5.5向下取整为−6。注意，因为在数轴上−6在−5.5左边，因此它小于−5.5。

3.3.4　循环移位运算符

和算术运算符相比，循环移位运算符不会把二进制序列看作整数，因此循环移位运算不会影响N、Z和V位，而只会影响C位。有两种循环移位运算符：表示为ROL的循环左移和表示为ROR循环右移。图3-20展示了6位单元的循环移位运算符的动作。循环左移类似于算术左移，在循环左移中C位会循环移到单元的最右边位，而在算数右移中是0。循环右移是在相反的方向做同样的事情。

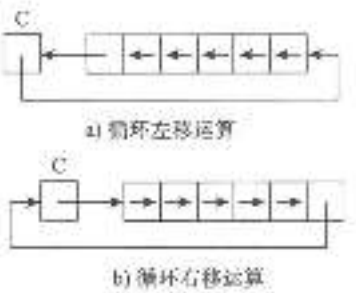


图3-20　循环运算的行为

6位单元循环左移的RTL表述是

C ← r〈0〉, r〈0..4〉 ← r〈1..5〉, r〈5〉 ← c;

例3.23　下面是4个循环移位运算的例子。

C = 1, ROL 01 1101 = 11 1011, C = 0
C = 0, ROL 01 1101 = 11 1010, C = 0
C = 1, ROR 01 1101 = 10 1110, C = 1
C = 0, ROR 01 1101 = 00 1110, C = 1

左边是循环移位前的C值，而右边是循环移位后的C值。　□

3.4　十六进制和符号表示

前面章节中的二进制表示是整数表示，本节介绍另一种基数，将用于下一章中介绍的计算机。本章还将介绍这种计算机是如何存储字母信息的。

3.4.1　十六进制

假定一个人有16根手指而不是10根。那么发明阿拉伯数字时，会发生什么情况呢？记得10根手指模式是从0开始，一直继续发明新的符号1、2，直到倒数第二根手指9，接着

在最后一根手指，把1和0结合在一起表示下一个数10。

如果是16根手指，当达到9时，仍然还剩下不少的指头，必须继续发明新的符号，这些额外的符号通常用英语字母表开头的字母表示。因此以16为底（十六进制，或缩写为hex）的计数是这样的：

0	7	E	15	1C	23
1	8	F	16	1D	24
2	9	10	17	1E	25
3	A	11	18	1F	26
4	B	12	19	20	.
5	C	13	1A	21	.
6	D	14	1B	22	.

111 / 112

当十六进制数字包含许多位时，计数就有点儿麻烦。思考从8BE7、C9D和9FFE开始接下来的5个数字：

8BE7	C9D	9FFE
8BE8	C9E	9FFF
8BE9	C9F	A000
8BEA	CA0	A001
8BEB	CA1	A002
8BEC	CA2	A003

当写八进制数时，数字看上去有比它们实际要大的趋势；写成十六进制数时，效果是相反的，数字有看上去比它们实际要小的趋势。比较十六进制数的列表和十进制数的列表，可以看出18（hex）实际上是24（dec）。

3.4.2 基数转换

在十六进制中，每个位置值都是比它低一位的位置值的16倍。十六进制转换为十进制，可以简单地把位置值乘以该位置的数字，并相加。

例3.24 图3-21给出了把8BE7从十六进制转换到十进制的过程。B的十进制值是11，E的十进制值是14。□



a) 8BE7的位置值　　b) 把8BE7转换为十进制

图3-21 将十六进制数转换为十进制

从十进制转换到十六进制的过程类似于从十进制转换到二进制的过程，不过不是一个接一个地除以2，而是除以16，并保存余数的记录，这些余数就是转换后的十六进制数。

对于小于255（dec）或FF（hex）的数，两种进制互相转换用图3-22所示的表格是很容易做到的。表中的主体是十进制数，左列和顶行是十六进制数。

113

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0_	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1_	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2_	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47

图3-22 十六进制转换表

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3_	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4_	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5_	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6_	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7_	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8_	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9_	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A_	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B_	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C_	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D_	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E_	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F_	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

图 3-22 （续）

例 3.25　把 9C（hex）转换到十进制，查看 9 行和 C 列找到 156（dec）；把 125（dec）转换到十六进制，在表的主体中找到 125，从对应的左列和顶行得出 7D（hex）。□

如果计算机以二进制格式存储信息，那么为什么要学习十六进制呢？答案是，二进制和十六进制之间存在特殊关系，如图 3-23 所示。4 位有 16 种可能的组合，而正好有 16 个十六进制数字，每个十六进制数字代表这 4 位。

十六进制	二进制	十六进制	二进制
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

图 3-23　十六进制和二进制之间的关系

为了节约打印空间，位模式经常被写成十六进制形式。一个 16 位机器的手册可能会说某个内存位置包含 01D3，这要比说它包含 0000 0001 1101 0011 简短多了。

把无符号二进制转换到十六进制，从最右边开始把位划分为每 4 个一组，给每组一个图 3-23 对应的十六进制数字。把十六进制转换到无符号二进制，简单地把过程反过来即可。

例 3.26　写出 10 位无符号二进制数 10 1001 1100 的十六进制形式，从最右边的 4 位 1100 开始：

10 1001 1100（bin）＝ 29C（hex）□

由于 10 位不能刚好分为 4 个一组，所以在图 3-23 中查找最左边的数字时，在前面加 2 个 0。在本例中，最左边的十六进制数字来自

10（bin）＝ 0010（bin）＝ 2（hex）□

例 3.27　对于 14 位单元

0D60（hex）＝ 00 1101 0110 0000（bin）

注意，最末尾的十六进制 0 代表 4 个二进制 0，而最高位的十六进制 0 只代表 2 个二进制 0。□

把十进制转换为无符号二进制，你可能想要用十六进制－十进制表作为中间步骤。通过查找图 3-22 中的十六进制值，不用任何计算，再根据图 3-23 把每个数字转换为二进制即可。

例 3.28　对于 6 位单元，

29 (dec) = 1D (hex) = 01 1101 (bin)

转换中的每一步都是一次简单的查表。 □

在机器语言程序代码或程序记录中，几乎不会把数字写成有负号的十六进制形式，而是把符号位隐含地包含在十六进制表示的位模式中。你必须牢记十六进制只是二进制序列的一个方便的缩写，硬件只存储二进制值。

例 3.29　如果一个 12 位的内存位置包含 F7A (hex)，那么通过思考下面的位模式可以得出十进制数。

F7A (hex) = 1111 0111 1010 (bin)

符号位是 1，因此这个数是负数，转换为十进制是

F7A (hex) = −134 (dec)

注意，尽管可以解释成一个负数，但是十六进制数不会写成有负号的形式。 □

3.4.3　字符

114 ? 115

因为计算机内存是二进制的，所以字母字符必须要编码后才能存储到内存中。美国信息交换标准代码 (American Standard Code for Information Interchange，ASCII) 是一个使用广泛的字母字符二进制编码。

ASCII 包含所有大写和小写的英文字母、10 个数字和特殊符号 (例如，标点)。它的一些符号是不能打印的，主要用于在计算机之间传递信息或用于控制外围设备。

ASCII 是一种 7 位的编码。因为 7 位有 2^7=128 种可能的组合，所以有 128 个 ASCII 字符，图 3-24 给出了所有这些符号。表中第一列是不可打印的符号，它们的意思列在表下，表中其余部分是可打印的符号。

例 3.30　代表响铃 (bell) 的序列 000 0111 使终端发出哔哔声。另一个例子是一组命令，用于控制纸张打印机在新的一行起始开始打印。计算机发送一个回车符 (CR，000 1101)，再发送一个换行符 (LF，000 1010)。CR 使得“打印头”或光标回到纸张的左边，LF 使纸张往下走一行。 □

例 3.30　名字 Tom 会以下列 ASCII 形式存储。

101 0100
110 1111
110 1101

如果将这个位序列发送到输出终端，就会显示“Tom”。 □

例 3.32　街道地址 52 Elm 会以下列 ASCII 形式存储。

011 0101
011 0010
010 0000
100 0101
110 1100
110 1101

2 和 E 之间的空格是一个独立的 ASCII 字符。 □

尽管 ASCII 使用广泛，但它绝对不是表示字符的唯一编码。由于这种 7 位编码没有提

供除英语外其他语言中常见的重音符号，因此它的使用是有限的。由于这个限制，所以对其做了扩展，使用8位来提供很多7位无法表示的重音符号。

字符	二进制	十六进制	字符	二进制	十六进制	字符	二进制	十六进制	字符	二进制	十六进制
NUL	000 0000	00	SP	010 0000	20	@	100 0000	40	`	110 0000	60
SOH	000 0001	01	!	010 0001	21	A	100 0001	41	a	110 0001	61
STX	000 0010	02	"	010 0010	22	B	100 0010	42	b	110 0010	62
ETX	000 0011	03	#	010 0011	23	C	100 0011	43	c	110 0011	63
EOT	000 0100	04	$	010 0100	24	D	100 0100	44	d	110 0100	64
ENQ	000 0101	05	%	010 0101	25	E	100 0101	45	e	110 0101	65
ACK	000 0110	06	&	010 0110	26	F	100 0110	46	f	110 0110	66
DEL	000 0111	07	'	010 0111	27	G	100 0111	47	g	110 0111	67
BS	000 1000	08	(	010 1000	28	H	100 1000	48	h	110 1000	68
HT	000 1001	09	)	010 1001	29	I	100 1001	49	i	110 1001	69
LF	000 1010	0A	*	010 1010	2A	J	100 1010	4A	j	110 1010	6A
VT	000 1011	0B	+	010 1011	2B	K	100 1011	4B	k	110 1011	6B
FF	000 1100	0C	,	010 1100	2C	L	100 1100	4C	l	110 1100	6C
CR	000 1001	0D	-	010 1101	2D	M	100 1101	4D	m	110 1101	6D
SO	000 1110	0E	.	010 1110	2E	N	100 1110	4E	n	110 1110	6E
SI	000 1111	0F	/	010 1111	2F	O	100 1111	4F	o	110 1111	6F
DLE	001 0000	10	0	011 0000	30	P	101 0000	50	p	111 0000	70
DC1	001 0001	11	1	011 0001	31	Q	101 0001	51	q	111 0001	71
DC2	001 0010	12	2	011 0010	32	R	101 0010	52	r	111 0010	72
DC3	001 0011	13	3	011 0011	33	S	101 0011	53	s	111 0011	73
DC4	001 0100	14	4	011 0100	34	T	101 0100	54	t	111 0100	74
NAK	001 0101	15	5	011 0101	35	U	101 0101	55	u	111 0101	75
SYN	001 0110	16	6	011 0110	36	V	101 0110	56	v	111 0110	76
ETB	001 0111	17	7	011 0111	37	W	101 0111	57	w	111 0111	77
CAN	001 1000	18	8	011 1000	38	X	101 1000	58	x	111 1000	78
EM	001 1001	19	9	011 1001	39	Y	101 1001	59	y	111 1001	79
SUB	001 1010	1A	:	011 1011	3A	Z	101 1010	5A	z	111 1010	7A
ESC	001 1011	1B	;	011 1011	3B	[	101 1011	5B	{	111 1011	7B
FS	001 1100	1C	<	011 1100	3C	\	101 1100	5C	|	111 1100	7C
GS	001 1101	1D	=	011 1101	3D	]	101 1101	5D	}	111 1101	7D
RS	001 1110	1E	>	011 1110	3E	^	101 1110	5E	~	111 1110	7E
US	001 1111	1F	?	011 1111	3F	_	101 1111	5F	DEL	111 1111	7F

控制符的缩写

NUL	空字符	FF	走页键	CAN	取消
SOH	标题开始	CR	回车键	EM	介质中断
STX	正文开始	SO	不用切换	SUB	替补
ETX	正文结束	SI	启用切换	ESC	换码（溢出）
EOT	传输结束	DLE	数据链路转义	FS	文件分隔符
ENQ	请求	DC1	设备控制1	GS	分组符
ACK	收到通知	DC2	设备控制2	RS	记录分离符
BEL	响铃	DC3	设备控制3	US	单元分隔符
BS	退格	DC4	设备控制4	SP	空格
HT	水平制表符	NAK	拒绝接收	DEL	删除
LF	换行键	SYN	同步空闲		
VT	垂直制表符	ETB	传输块结束		

图 3-24　美国信息交换标准代码（ASCII）

但即使有了这个扩展仍不足以处理非拉丁字符。由于全球信息交流的至关重要，所以发明了一种称为 Unicode（统一码）的标准。Unicode 的目标是可以对世界上所有语言的符号进行编码，甚至包括已经不再使用的古语言。Unicode 符号集合使用 32 位或 4 字节。由于大多数的应用不会用到这些符号中的大多数，所以 Unicode 标准制定了使用小于 4 字节的技术。基本多文种平面（Basic Multilingual Plane）包含常用的 Unicode 字符，每个字符仅占用 2 字节。用它来存储 1 字节的扩展 ASCII 码，所需存储空间是其两倍。不过基本多文种平面实际上包含了世界上所有的书面语言，包括阿拉伯语、亚美尼亚语、汉语、西里尔语、希腊语、希伯来语、日语、韩语、叙利亚语、许多非洲语言，甚至加拿大土著语言模式。

3.5 浮点数表示

本章前面几节描述的数值表示是对于整数值的。C++ 有 3 种数值类型有小数部分：

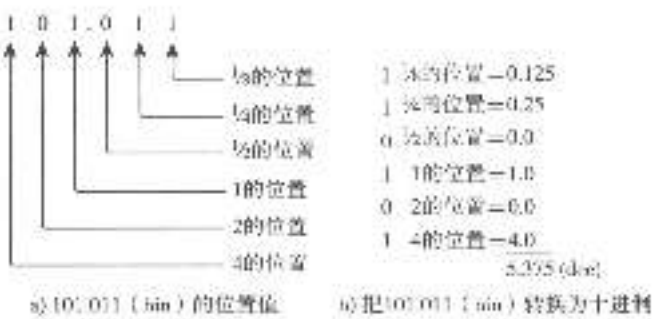
- `float` 单精度浮点数
- `double` 双精度浮点数
- `long double` 长双精度浮点数

这些类型的值在 ISA3 层不能以补码的形式存储，因为存储必须提供存放数字中小数点位置的方式。浮点数值用科学计数法的二进制版本来存储。

3.5.1 二进制小数

二进制小数有一个二进制小数点，它是十进制小数点的二进制版本。

例 3.33 图 3-25a 展示了 101.011（bin）的位置值。二进制小数点左边的位与图 3-2 无符号二进制表示中相应的位有相同的位置值。二进制小数点右边的位置值从 1/2 开始，每个位置值是前一位的一半。图 3-25b 给出的加法表明得到的值是 5.375（dec）。 □



a) 101.011（bin）的位置值　　b) 把101.011（bin）转换为十进制

图 3-25 把二进制小数转换为十进制

图 3-26 是有小数部分数字的多项式表示。小数点左边一位的位置值总是基数的 0 次方，即 1。往左下一个有效位是基数的 1 次方，即基数本身。小数点右边一位的位置值是基数的 -1 次方，往右下一个有效位是基数的 -2 次方，右边每个位置值是它左边位位置值的 1/ 基数倍。

确定二进制小数的十进制值分为两步。首先，用例 3.3 中无符号二进制数转换的方法转换二进制小数点左边的位。然后，用逐位翻倍的算法转换二进制小数点右边的位。

例 3.34 图 3-27 展示了把 6.585 937 5（dec）转化为二进制的过程。转换整数部分就是转化小数点左边的 110（bin）；转换小数部分是把小数点右边的数字写在表格右列的头部，小数部分乘以 2，小数点左边的数字写在左列，小数部分写在右列。下次乘 2 时，不包括整数部分。例如，把 .171 875 乘 2 得到 0.343 75，而不是把 1.171 875 乘 2。左列从上到下的数字就是二进制小数部分从左到右的位，因此 6.585 937 5（dec）= 110.100 101 1（bin）。 □

把小数部分从十进制转换到二进制的算法就像是把整数部分从十进制转换到二进制算法

的镜像。图3-5给出了用逐位除以2的算法转换十进制整数的过程。除法的余数就是得到的数字位，顺序是从二进制小数点开始从右往左。用逐位乘以2的算法转换小数部分。乘法得到的整数部分是生成的数字位，顺序是从二进制小数点开始从左往右。

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

a) 二进制数101.011

$5\times10^2+0\times10^1+6\times10^0+7\times10^{-1}+2\times10^{-2}+1\times10^{-3}$

b) 十进制数506.721

图3-26　浮点数的多项式表示

6.5859375

6 (dec) = 110 (bin)

a) 转换整数部分

	.5859375
1	.171875
0	.34375
0	.6875
1	.375
0	.75
1	.5
1	.0

b) 转换小数部分

图3-27　把十进制小数转换为二进制数

一个可以用有限位十进制表示的数，它的二进制数表示可能是无限位的。

例3.35　图3-28展示的是把0.2（dec）转换为二进制的过程。第一次乘2得到0.4，反复几次后又得到了0.4。显然这个过程不会终止，所以0.2（dec）= 0.001100110011...（bin），位模式0011会不断重复。□

由于所有计算机单元都只能存储有限的位，所以0.2（dec）不能被精确存储，必定是一个近似值。我们应该意识到由于二进制表示值的固有舍入误差，所以如果用像C++这样的HOL6层语言做加法0.2 + 0.2，也许不会精确得到0.4。正是由于这个原因，好的数值软件几乎不会检测两个浮点数是否完全相等，而是用软件维护了一个很小的非零容忍值，用以表示如果两个浮点数的差小于该值就被看作相等。如果容忍值是0.0001，那么1.382 64和1.382 67会被认为是相等的，因为它们的差0.000 03小于容忍值0.0001。

	.2
0	.4
0	.8
1	.6
1	.2
0	.4
0	.8
1	.6
⋮	⋮

图3-28　一个具有无休止的二进制表达的十进制数

3.5.2　余码表示

可以用常见于十进制数的科学计数法的二进制版本来表示浮点数。一个以科学计数法表示的非零数是规格化的，如果它的第一个非零位正好在小数点左边。因为0没有第一个非零位，因此0不能被规格化。

例3.36　十进制数-328.4的科学计数法规格化表示是-3.284×10^2，10的2次方的作用是把小数点往右移动2位。类似地，二进制数-10101.101的科学计数法规格化形式是-1.0101101×2^4，2的4次方的作用是把二进制小数点往右移动4位。□

116
|
120

例3.37　二进制数0.00101101的科学计数法规格化表示是1.01101×2^{-3}，2的-3次方的作用是把二进制小数点往左移动3位。□

一般来说，浮点数可以是正数或负数，它的指数也可以是正整数或负整数。图3-29展示了存储浮点数值的一个内存单元。单元分为3个字段，第一个字段1位，用于存储该数的符号。第二个字段存储代表规格化二进制数的指数位。第三个字段称为有效位数，存储代表数值大小的位。

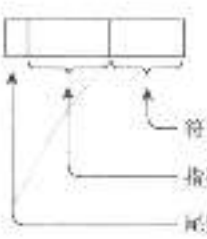


图3-29　浮点数的存储

任何有符号整数的表示方法都可以用于存储指数。你可能会想到

用补码表示，因为大多数计算机存储有符号整数都用它。但是实际上没有用补码，而是用了有偏差的表示方法，后面很快会解释这个原因。

一个5位单元有偏差表示的例子是余15码（Excess 15）。单元存储数字的范围十进制表示为-15到16，二进制表示为00000到11111。把十进制转换到余15码是把十进制数值加15，然后按照无符号数方式转换为二进制。从余15码转换为十进制是把它按照无符号数写作十进制数，然后减去15。在余15码中，第一位表示一个值是正还是负，不过与补码表示不一样，1表示正值，0表示负值。

例3.38 把十进制5转换到余15码，5 + 15 = 20。然后按照无符号数方法把20转换为二进制，20（dec）= 10100（excess 15）。第一位是1表示是一个正值。□

例3.39 把00011从余15码转换到十进制，把00011当作无符号值转换，00011（bin）= 3（dec），然后3 - 15 = -12，因此00011（excess 15）= -12（dec）。□

图3-30展示了一个3位单元以余3码表示和以补码表示存储整数的比较。每种表示法存储8个值，余3码的表数范围是-3到4（dec），而补码的是-4到3（dec）。□

十进制数	余3码	补　码
-4		100
-3	000	101
-2	001	110
-1	010	111
0	011	000
1	100	001
2	101	010
3	110	011
4	111	

图3-30 3位单元存储的有符号整数

3.5.3 隐藏位

假设浮点数以第一个非零位直接存储为小数点左边的规格化形式，那么就不需精确地存储二进制小数点，因为它总是在同样的位置。假设图3-29的符号位为1表示负值，为0表示正值，指数是3位，尾数是4位，那么可以存储尾数为4位的数字。要存储十进制值，首先把它转换为二进制，写成规格化的科学计数法的形式，然后以余3码的形式存储指数，再存储尾数的多个最高有效位。 121

例3.40 存储0.34，把它转换到二进制0.34（dec）= 0.0101011110...，这个位序列是无穷尽的，因此只存储最高位。这个数的规格化科学计数法值是$1.0101110\cdots\times2^{-2}$，指数是-2。从图3-30可以看到它的余3码表示是001。最高的4位是1010。它在第一个数字的后面隐含了小数点。这个数是正数，因此符号位是0。存储这个值的位模式是0 001 1010。

来看一看这个近似值有多接近，把存储值转换回十进制。存储值1.010×2^{-2}（bin）= 0.3125，它和原始十进制值相差0.0275。□

令人遗憾的是不能在尾数中存储更多的高位数字。当然，与实际机器中的浮点数格式相比，指数3位，尾数4位是太小了。例子采用这么小的表示是为了说明起来简单。然而，即使在实际的机器中尾数字段大得多，近似度好很多，但是仍然不可避免近似的发生，因为内存单元是有限的。

可以利用当数字用规格化表示时二进制小数点左边总是1的这个事实。因为1总是在那里，所以可以简单地不存储它，这可以给尾数扩展1位精确度的空间。这个假定在二进制小数点左边而又不显式存储的位叫作隐藏位（hidden bit）。

例3.41 采用假定在尾数中有隐藏位的表示方法，0.34（dec）存储为0 001 0101。二进制小数点右边的前4位是0101，小数点左边的1位是假定的。来看看精确度的改进。现在的存储值是1.0101×2^{-2}（bin）= 0.328 125，它与原始的十进制值相差0.011 875。没有隐藏位的差是0.0275，因此使用隐藏位改进了近似值。□

当然隐藏位是假定的，不要忽略它。当在一个程序中写十进制浮点数时，编译器生成代码把值转换为二进制，会丢弃假定存在的隐藏位，尽可能多地存储二进制小数点右边的位。如果程序把两个存储的浮点数相乘，那么计算机在执行乘法运算前，会抽取尾数位并插入假设的隐藏位。对于乘积，会除去隐藏位后才进行存储。

3.5.4　特殊值

有些实际值需要特殊看待，最明显的是0，因为它的二进制表示中没有为1的位，因此它不能规格化表示，必须为它设置一个特殊的位模式。标准的做法是把指数全置为0，尾数
122 也全置为0。符号位呢？最常见的是0有两种表示：一个正0，一个负0。如果指数3位，尾数4位，两种0的位模式是

1 000 0000（bin）＝ −0.0（dec）

0 000 0000（bin）＝ +0.0（dec）

不过，0的存储还有其他解决方案。如果+0.0的位模式没有特殊指定，那么0 000 0000会被解读成有隐藏位，看作 1.0000×2^{-3}（bin）＝0.125，如果这个值没有被保留为0，那么这就是可以存储的最小正值。如果这个位模式为0保留，那么可存储的最小正值是略大的0 000 0001 ＝ 1.0001×2^{-1}（bin）＝0.132 812 5。除了符号位是1，数值最小负数的尾数应该是一样。具有最小非0尾数的数应该是

1 000 0001（bin）＝ −0.1328125（dec）

0 000 0001（bin）＝ +0.1328125（dec）

可以存储的最大正整数的位模式应该具有最大指数和最大尾数，而具有最大数值大小的负数应该是一样的位模式，除了符号位为1。具有最大数值大小的位模式和它们的十进制值应该是

1 111 1111（bin）＝ −31.0（dec）

0 111 1111（bin）＝ +31.0（dec）

图3-31是0有唯一特殊值的表示方式所对应的数轴。和整数表示一样，可以存储多大的值是有限制的。如果9.5乘以12.0，两者都在范围内，但是乘积的真实值114.0在正上溢区。



图3-31　0有唯一特殊值的实数轴

然而，和整数值不一样的是，实数轴有下溢区。如果0.125乘以0.125，两者都在范围内，但乘积的真值0.156 25却在正
123 下溢区，可以存储的最小正值是0.132 815。

当计算以确切的精度进行时，近似的浮点数的数值计算结果要和预期保持一致。例如，假设9.5乘以12.0，结果应该存储什么呢？假设把最大值31.0作为近似结果存储。再假设它是一个更长计算的中间值，然后要计算它的一半是多少，将得到15.5，这和正确的值相差甚远。

在下溢区有同样的问题。如果把0.0作为0.156 25的近似值存储，然后再把它乘以12.0，就会得到0.0，你会有被看上去合理的值误导的风险。

由上溢和下溢引起的这些问题，通过引入更多的特殊值会有所改善。与0的表示一样，必须用一些特殊的位模式来表示这些特殊值，如果这些位模式不用来表示特殊值，就可以用来表示数轴上的值。除了0以外，有3个常用的特殊值：

- 无穷大
- 非数
- 非规格化数

无穷大用于表示溢出区的值。如果运算结果溢出，那么就存储无穷大的位模式。如果再对这个位模式执行运算，结果就是预期的值——无穷。例如，$3/\infty = 0$，$5 + \infty = \infty$。而无穷大的平方根是无穷大。除以0会得到无穷大，例如，$3/\infty = \infty$，$-4/\infty = -\infty$。如果实数做计算得到无穷大，那么就可以知道某个中间结果发生了溢出。

如果一个值不是一个数（即，非数），它的位模式称为NaN。用NaN来表示非法的浮点运算，例如，取负数的平方根得到NaN，0/0也得到NaN。任何至少有一个NaN操作数的浮点运算都得到NaN，例如，7 + NaN = NaN。

无穷大和NaN的位模式都使用了指数的最大正值，即指数字段是全1，无穷大的尾数为全1，NaN的尾数可以是任何非零模式。把这些位模式保留给无穷大和NaN会减少可以存储的值的范围。对于3位指数和4位尾数来说，最大数值的位模式和它们的十进制值是

1 111 0000（bin）= $-\infty$

1 110 1111（bin）= −15.5（dec）

0 110 1111（bin）= −15.5（dec）

0 111 0000（bin）= $+\infty$ 124

在图3-31中，上溢出区的无穷大值，没有对应的下溢区的无穷小值。不过非规格化数是特殊的值，它们有一个很好的属性，称为逐级下溢。有了逐级下溢，最小正值和0之间的差距减小了很多。主要思想是，选取那些指数字段全0的非零值，把它们平均分布在下溢区中。

因为指数字段全0是为非规格化数保留的，所以最小正规格化数是0 001 000 = 1.000×2^{-2}（bin）= 0.25（dec）。如果指数字段可以是000，那么最小正规格化数是0.132 812 5，我们现在的做法似乎把事情去得更糟了。但是，非规格化数分布在原来表示方法的下溢区间内，实际上是减小了下溢区的大小。

当指数字段全是0，尾数至少包含一个1时，要使用特殊的表示规则。假定指数是3位，尾数是4位。

- 假定二进制小数点左边的隐藏位为0，而不是1。
- 假定指数以余2码而不是余3码的形式存储。

例3.42 对于3位指数、4位尾数的表示方法来说，0 000 0110表示什么十进制值？因为指数全是0，尾数至少包含一个1，所以这个数是非规格化数，它的指数是000（余2码）= 0 − 2 = −2，它的隐藏位是0，所以它的二进制科学计数值是0.0110×2^{-2}。因为这是非规格化数的特殊情况，所以指数以余2码而不是余3码表示。将这个二进制值转换为十进制，得到0.093 75。 □

这样的表示会让下溢区的表示变得更好。计算具有最小数值的数，它是非规格化的。

1 000 0001（bin）= −0.015625（dec）

1 000 0000（bin）= −0.0

0 000 0000（bin）= −0.0

0 000 0001（bin）= −0.015625（dec）

如果没有非规格化数，最小正数是0.132 812 5，因此实际上下溢区变小了很多。

图3-32展示了具有所有特殊值的3位指数和4位尾数表示法的一些重要值。这些值按

照数字从小到大的顺序排列。图3-32表明为什么要用余码来表示浮点数的指数，忽略符号位，只考虑从0.0到$-\infty$的正数。可以看到，如果把最右边的7位看作一个简单的无符号整数，那么从表示0的000 0000到表示∞的111 0000，相邻的值都是加1。如果对两个浮点数进行比较，比如这样一条C++语句

```
if (x < y)
```

那么计算机不需要提取指数字段或者插入隐藏位，只需要把最右边7位当作整数，进行比较，就能判断那个浮点数有更大的数值。整数运算的电路要比浮点数的快很多，因此用余码表示指数实际上提高了性能。

	二　进　制	科学计数法	十　进　制
非数	1 111 非零		
负无穷大	1 111 0000		$-\infty$
负规格化数	1 110 1111	-1.1111×2^{7}	−15.5
	1 110 1110	-1.1110×2^{7}	−15.0
	…	…	…
	1 011 0001	-1.0001×2^{0}	−1.0625
	1 011 0000	-1.0000×2^{0}	−1.0
	1 010 1111	-1.1111×2^{-1}	−0.96875
	…	…	…
	1 001 0001	-1.0001×2^{-6}	−0.265625
	1 001 0000	-1.0000×2^{-6}	−0.25
负非规格化数	1 000 1111	-0.1111×2^{-6}	−0.234375
	1 000 1110	-0.1110×2^{-6}	−0.21875
	…	…	…
	1 000 0010	-0.0010×2^{-6}	−0.03125
	1 000 0001	-0.0001×2^{-6}	−0.015625
负 0	1 000 0000		−0.0
正 0	0 000 0000		+0.0
正非规格化数	0 000 0001	0.0001×2^{-6}	+0.015625
	0 000 0010	0.0010×2^{-6}	+0.03125
	…	…	…
	0 000 1110	0.1110×2^{-6}	0.21875
	0 000 1111	0.1111×2^{-6}	0.234375
正规格化数	0 001 0000	1.0000×2^{-6}	0.25
	0 001 0001	1.0001×2^{-6}	0.265625
	…	…	…
	0 010 1111	1.1111×2^{-1}	0.96875
	0 011 0000	1.0000×2^{0}	1.0
	0 011 0001	1.0001×2^{0}	1.0625
	…	…	…
	0 110 1110	1.1110×2^{7}	15.0
	0 110 1111	1.1111×2^{7}	15.5
正无穷大	0 111 0000		$+\infty$
非数	0 111 非零		

图3-32　3位指数和4位尾数的浮点数值

对于负数，也有同样的模式。可以把最右的7位看作无符号整数，用以比较负数的数值大小。如果指数用补码表示，浮点数就不会有这样的属性。

如果x值计算为-0.0，y值计算为+0.0，那么程序员会预期表达式（x<y）为false。对于实数，正0和负0没有区别。在这种特定情况下，即使位模式表示x是负数y是正数，计算机编程也一定要返回false。

3.5.5 IEEE 754浮点数标准

电气电子工程师学会（IEEE）是一个由会员支持的专业协会，为各种工程领域提供服务，计算机工程就是其中之一。协会内有各种小组起草工业标准。在IEEE提出它的浮点数标准之前，每个计算机厂商都设计它们自己的浮点数值表示法，互不相同。在网络普及前的早期，计算机之间的数据共享很少，因此这种情况尚可容忍。

即便没有大量的数据共享需求，标准的缺失也阻碍了数字计算的研究和发展。在两台不同的计算机上运行两个一样的程序，同样的输入可能产生不同的结果，原因是两台计算机采用了不同的近似值表示法。

1985年IEEE设立了一个委员会来起草浮点数标准。最终产生了两个标准：854更适用于手持计算器，而754广泛应用于计算机。实际上，现在每个计算机厂商的计算机中的浮点数都遵循IEEE 754标准。

125 ~ 127

在本节前面讲述的浮点数表示法中，除了指数字段和有效位的数位不同之外，其余的都和IEEE 754是一样的。图3-33展示了这个标准的两种格式：单精度格式的指数字段是8位单元，采用余127码表示（除了非规格化数，它们用的余126码），尾数是23位；双精度格式的指数字段是11位单元，采用余1023码表示（除了非规格化数，它们用的余1022码），尾数是52位。



图3-33 IEEE 754浮点数标准

有下列单精度格式的位值，正无穷大是

0 1111 1111 000 0000 0000 0000 0000 0000

写成4位一组的全32位模式为

0111 1111 1000 0000 0000 0000 0000 0000

它的十六进制简化表示为7F80 0000（hex）。最大的正值为

0 1111 1110 111 1111 1111 1111 1111 1111

其值近似是2^{128}或10^{38}。它的十六进制表示是7F7F FFFF（hex）。最小的规格化正数是

0 0000 0001 000 0000 0000 0000 0000 0000

它的十六进制表示是0080 0000（hex）。最小的非规格化正数是

0 0000 0000 000 0000 0000 0000 0000 0001

它的十六进制表示是0000 0001（hex），近似值为10^{-45}。

William V. Kahan

1933年，William V. Kahan生于加拿大。他曾就读于多伦多大学，1958年获得数学

博士学位。

1976年，Intel计划为它的微处理器产品线构建一个浮点协处理器。John Palmer负责该项目，他说服Intel需要一个数学标准，这样公司生产的不同芯片对于相同的浮点输入就能得到相同的输出。在Stanford大学时，Palmer听说过Kahan分析了一些当时流行的计算机的浮点值表示。他雇用Kahan作为咨询专家，建立浮点数表示法的细节。



在那之后，IEEE成立了一个委员会开发业界的浮点数标准。Kahan是该委员会的成员，虽然刚开始有一些争议，但是他在Intel的工作还是成为了IEEE 754标准的基础。当时，Digital Equipment Corporation（DEC）在它的VAX系列计算机上采用了一种广受重视的浮点数表示法。在刚开始与Palmer接触时，Kahan甚至建议Intel就采用该方法。但是VAX的表示法没有逐级下溢的非规格化数。在委员会的讨论中，因为认为这种表示法的所有实现执行起来都很慢，所以这个特性成为一个很大的问题。这场关于逐级下溢的论战持续多年，DEC声称具有逐级下溢特性的计算性能都不可能超过VAX。最后，加利福尼亚大学伯克利分校的Dave Patterson的一个研究生，George Taylor，构建了一个Kahan的浮点数标准的工作原型电路板，可以插入VAX机器而不会降低机器的速度。

本章略去了很多有关IEEE 756的细节，包括保护数字、异常和标志等方面的规定。Kahan致力于“让数值计算的世界更安全”。实际上，所有硬件都遵循该标准，只是有些软件系统没有很好地利用异常和标志。当发生这种情况时，Kahan就会很快公布这些问题。Sun Microsystems在推广Java语言时用到这样一句口号“写一次，随处运行”，就曾经被Kahan在名为“How Java's Floating-Point Hurts Everyone Everywhere”的论文中批评过。当Matlab软件最新发布的版本没有像较早的版本那样遵循IEEE 754标准时，Kahan的论文题目就是“Matlab's Loss Is Nobody's Gain”。

1989年，William Kahan因为他在数值分析方面的基础性贡献获得了A. M. Turing奖。在本书书写之时，他是加利福尼亚大学伯克利分校数学系和电器工程和计算机科学系的教授。

例3.43　-47.25的单精度浮点数的十六进制表示是什么？整数47（dec）= 101111（bin），小数0.25（dec）= 0.01（bin），因此47.25（dec）= 1.0111101×2^5。这个数是负数，因此第一位是1，指数5通过5+127 = 132（dec）= 1000 0100（余127）转换为余127码，尾数存储二进制小数点右边的0111101，因此位模式是

128
/
129

1 1000 0100 011 1101 0000 0000 0000 0000

十六进制表示为C23D 0000（hex）。□

例3.44　十六进制表示为3CC8 0000的二进制科学表示法是什么？它的位模式为0 0111 1001 100 1000 0000 0000 0000 0000，符号位是0，因此这个数是正数，指数是0111 1001（excess 127）= 121（unsigned）= 121-127 = -6（dec），尾数的小数点右边是1001，隐藏位为1，因此这个数是1.1001×2^{-6}。□

例3.45　十六进制表示为0050 0000的二进制科学表示法是什么？它的位模式是0 0000 0000 101 0000 0000 0000 0000 0000，符号位是0，因此它是正数，指数字段全是0，

因此它是非规格化数，指数 0000 0000（excess 126）= 0（unsigned）= 0−126 = −126（dec），隐藏位是 0 而不是 1，因此这个数是 0.101×2^{-126}。 □

3.6 跨层的表示方法

C++ 是一种 HOL6 层语言，当程序员在 C++ 中声明变量时，必须指定变量值的类型。而在 ISA3 层，变量是二进制的。

假设在一个 C++ 程序中声明了

```
int i, j;
char ch1, ch2;
```

并在一台 7 位计算机上运行这个程序。在 ISA3 层，int 类型的值以补码二进制表示法存储。如果 i 和 j 的值分别是 8 和 −2，程序包含语句

```
i = j;
```

130

那么在 ISA3 层对这个表达式求值是这样的：

```
        000 1000
ADD     111 1110
-------------------
N = 0   000 0110
Z = 0
V = 0
C = 1
```

在 ISA3 层，char 类型的值以 ASCII 或其他字符编码的形式存储。如果 ch1 的值是 −，ch2 的值是 2，那么这些值在 ISA3 层被存储为

010 1101　　011 0010

这个位模式当然和整数 j 的值不一样，j 的值是 111 1110。

在 C++ 中，在 HOL6 层，每个字符在数轴上都有一个序数值位置。在 ISA3 层，机器层，这个序数值就是作为无符号整数来解释的字符编码的二进制值。因为不同的计算机可以选择使用不同的二进制符号编码，所以它们的序数值可能会不同。

例 3.46 根据 ASCII 表，D 用 100 0100 代表，而 100 0100（bin）= 68（dec）。在一台使用 ASCII 码的计算机上，D 的序数值就是 68。 □

例 3.47 要在你的计算机上振铃，可以执行如下产生振铃的语句

```
int j = 7;
char ch = j;
cout << ch;
```

□

在 HOL6 层，高级语言的一个典型语句是

```
cout << "Tom";
```

字符串常量 Tom 被传送到输出设备。这条语句在 ISA3 层并不这么简单。在机器语言中，你没法“写出 Tom”，而是必须把位序列

101 0100

110 1111

110 1101

发送到输出设备。

131

为什么我们要处理位而不是我们习惯的英文字母和十进制数字呢？因为计算机是电子的。

最便宜最可靠的制造计算机电子部件的方法是把它们做成二进制的。因此我们只能用二进制机器来处理信息。ISA3层的问题是我们要处理的信息是十进制数字和英文字母的形式，而机器语言是以1和0的形式表示信息，因此需要进行编码，例如补码、二进制表示法和ASCII。

要处理的信息形式和表示信息的语言之间的不匹配并不只是在ISA3层有，这也是所有较高层次要考虑的重要问题。

通过旅行推销员问题可以说明HOL6层的这种情况。一个推销员负责8个不同城市的业务，他乘坐航空公司航班穿梭在这些城市之间，这8个城市之间有14条航空线路。要处理的信息形式是航空公司提供的地图，地图展示了连接推销员负责范围内城市之间的所有航线。地图如图3-34所示，它也标出了每条航线航班的价格。



132

图3-34　旅行推销员问题的地图

推销员必须从洛杉矶出发，访问他职责范围内的每个城市，然后返回洛杉矶，他当然想把他的行程计划为花费最少的。

确定最佳行程表听起来像是计算机的完美工作。推销员把地图给程序员，程序员知道怎样用HOL6层语言（例如，C++）来说话。

但是现在程序员要面对这个计算机科学的根本问题。地图和HOL6层语言这两种数据形式不匹配，C++不认识地图，它只能认识诸如实数、整数和数组之类的东西。

程序员必须决定怎样用C++能够处理的形式来表达数据。例如，他可以用如图3-35所示的二维实数数组来表示地图。在这个方案中，顶行和左列的整数代表推销员所负责范围内的城市，表中每个实数表示从行索引代表的城市到列索引代表的城市的美元价格。如果这个

二维数组叫cost，那么

```
cost [0][5] = 65
```

表示从城市0（洛杉矶）飞到城市5（萨克拉门托）的费用是65美元。

```
cost [1][7] = 1e30
```

表示城市1（圣地亚哥）和城市7（拉斯维加斯）之间没有航线。

	0	1	2	3	4	5	6	7
0	1e30	35	45	40	55	65	80	70
1	35	1e30	30	1e30	1e30	1e30	1e30	1e30
2	45	30	1e30	1e30	1e30	1e30	1e30	60
3	40	1e30	1e30	1e30	1e30	1e30	1e30	1e30
4	55	1e30	1e30	70	1e30	40	60	1e30
5	65	1e30	1e30	1e30	40	1e30	40	1e30
6	80	1e30	1e30	1e30	60	40	1e30	50
7	70	1e30	60	1e30	1e30	1e30	50	1e30

City Numbers

0	洛杉矶	4	旧金山
1	圣地亚哥	5	萨克拉门托
2	棕榈泉	6	里诺
3	圣芭芭拉	7	拉斯维加斯

图3-35 图3-34所示航班图的一种数组表示

原始数据转换为C++可以处理的表示后，HOL6层程序员可以继续他的算法设计，并最终解决这个问题。但是第一步必然用这种语言能够处理的形式来表示这些数据。

ISA3层的问题是怎样用机器语言来表示数字和字母，HOL6层的问题是怎样用C++的整数、实数和数组来表示一个有城市、线路和航班费用的地图。在这两个层次上，都有最根本的数据表示问题。

3.6.1 另一种表示

信息表示问题具有挑战性的一面是通常会有多种不同的方法来表示数据。选择哪种表示取决于要怎样处理数据。

这里有一个在ISA3层表示正整数的另一种方法。尽管本章用无符号二进制来表示正数，但这并不是唯一的可能，正整数也能以二进制编码的十进制数（Binary Coded Decimal，BCD）形式来存储。

在BCD中，每个十进制的位刚好需要4位。十进制数142会以二进制0001 0100 0010的形式存储。因为只有10个十进制数字，因此4位的BCD组是0000、0001、0010、0011、0100、0101、0110、0111、1000和1001，从1010到1111的位模式未被使用。

如果数据更多的是在计算机内进行算术运算而不是I/O操作，通常会选择无符号二进制；当数据是金融业务并有很多I/O操作时，通常选用BCD。BCD更容易转换为十进制用于打印报表，然而BCD算术运算电路通常比无符号二进制算术运算电路慢。

推销员问题中也有类似的选择。例如，航空公司没有在所有可能的城市之间开通航线，尤其是小城市。要从棕榈泉到里诺，必须首先从棕榈泉飞到洛杉矶，然后从洛杉矶再到里

请。在图3-35中，尽管只有14条航线，但是cost有64个元素，大多数元素是le30，有28个元素不是le30，而只有14个是真正必需的。

例如，这两项

[133 / 134]

```
cost[0][5] = 65
cost[5][0] = 65
```

代表洛杉矶和萨克拉门托之间只有一条航线，假设两个方向的航班费用是相等的，那么实际上只需要一个项，另一项是多余的。

因此程序员可以决定采用图3-36所示的方式来表示地图。在HOL6层语言中，线路可以实现为一个数组route，数组中的每个记录包含3个字段：from、to和cost，那么

```
route[4].from = 0
route[4].to = 5
route[4].cost = 65
```

表示洛杉矶和萨克拉门托之间的飞行花费是65美元。

线路号	出发地	目的地	开销
0	0	1	35
1	0	2	45
2	0	3	40
3	0	4	55
4	0	5	65
5	0	6	80
6	0	7	70
7	1	2	30
8	2	7	60
9	3	4	70
10	4	5	40
11	4	6	60
12	5	6	40
13	6	7	50

图3-36　图3-34所示航班图的另一种表示

用这种方式表示地图，不会为不存在的航线浪费存储空间。如果城市9和城市11之间没有航线，那么就不用存储它们。

在HOL6层，地图还有其他表示方式，与在ISA3层无符号整数有其他表示方式一样。在任何层，都可能会有多种表示数据的方法，任何一种方法都能得到正确的结果。

确定哪一种表示最好通常是很困难的。实际上，通常要定义什么是“最好的”都很困难。如果你的计算机有大内存，那对你来说，最好的表示可能是一种有点儿浪费存储的方式，因为有足够的空间。另一方面，如果内存不足，最好的表示就是少用点儿存储空间，尽管用这种方式处理数据的算法会很慢。这种空间/时间的折中适用于计算机系统的各个抽象层次。与所有创造性努力会遇到的一样，这种选择并不总是很明确的。

3.6.2　模型

模型是某些物理系统的简化表示。每个科学领域的工作人员，包括计算机科学，构建模型并研究它们的性质，比如天文学家构建和研究的一些太阳系模型。

大约公元前350年，希腊的亚里士多德提出了一个模型，在这个模型中，地球位于宇宙的中心，环绕地球的是55个大球。太阳、月亮、行星和恒星每个都在一个大球上环绕天空。

这个模型和现实相符的程度如何呢？它能成功地解释天空的形状像球的顶部一样，也能解释行星大概的运行。千百年来，亚里士多德的模型被认为是准确的。

1543年，波兰天文学家哥白尼出版了《De Revolutionibus》（天体运行论），在这本书中，[135] 他建立了以太阳为中心的太阳系模型，行星围绕太阳做圆周运转。这个模型比地心模型更接近实体系统。

16世纪后期，丹麦天文学家Tycho Brahe（第谷·布拉赫）进行了一系列精确的天文观察，这些观察与哥白尼的模型有一定的差异。然后，1609年，Johannes Kepler（约翰内斯·开普勒）

设想了一个模型，在这个模型中，地球和所有行星围绕太阳运行，但轨道不是圆形而是扁平的圆形（即椭圆形）。这个模型成功地详细解释了 Tycho Brahe 观察到的行星的复杂运行。

每个模型都是太阳系的一个简化表示。没有一个模型能够完全精确地描述真实的物理世界。现在我们知道，根据爱因斯坦的相对论，甚至 Kepler 的模型也是一个近似模型。没有模型是完美的，每个模型都是现实世界的一个近似。

当信息在计算机存储器中表示时，这个表示也仅仅是一个模型。如同太阳系的每个模型描述真实系统的某个方面比其他方面更加精确一样，一种表示方法描述信息的某个性质比其他性质更精确。

例如，正整数的一个性质是有无穷大的数。不管你写一个多大的整数，别人总能写出更大的数。计算机的无符号二进制表示不能很精确地描述这个性质，因为内存中存储整数的空间大小是有限制的。

我们知道$\sqrt{2}$ = 1.4142136... 是无限不循环的。存储实数的表示方法是一个模型，对于像 2 的平方根这样的数，它只能存储近似数，它不能准确地表示 2 的平方根。

在任何层次解决问题都涉及构建一个不完美的模型并研究它的性质。HOL6 层的推销员问题是确定最少花费的行程，用航线地图把他的花费模型化。这个模型并不包括一些因素，比如一些城市的某些酒店可能在周末比工作日价格更贵。如果采用一个更接近现实花费的模型可能会改变最佳行程。

前面的例子说明计算机在任何时候解决问题，由于模型的限制，总会涉及近似。近似可能由于表示方法的限制而导致产生的，例如在存储 2 的平方根值时实数的精度是有限的，或者由于对问题的简化导致产生的，例如没有把不同的酒店价格计算在内。 136

计算机能模型化各种实体系统——库存清单、国民经济、账务系统和生物神经系统，这里仅举几个例子。在计算机科学中，要模型化的通常是计算机本身。

实际上，计算机只有的实体部分是在 LG1 层。从本质上说，计算机只是一个复杂的、有组织的大量电路和电子信号。在 ISA3 层，高电平信号被模型化为 1，低电平信号被模型化为 0。ISA3 层的程序员在使用模型时，不需要知道电子电路和信号。记住在 ISA3 层，单词 Tom 用 1 和 0 表示为

101 0100
110 1111
110 1101

HOL6 层的程序员在使用模型时，不需要知道位。实际上，在任何层，对计算机编程只需有那个层的计算机模型知识即可。

HOL6 层的程序员可以把计算机模型化为 C++ 机器，这个模型接受 C++ 程序并用它来处理数据。当程序员指示机器

```
cout << "Tom";
```

他不需要考虑计算机在 ISA3 层怎样被模型化为二进制机器。类似地，当 ISA3 层程序员写位序列时，他无须考虑在 LG1 层计算机怎样被模型化为电路的组合。

这种逐级为计算机系统建模的方法并不是计算机科学独有的。考虑一个有 6 个分公司分布全国的大公司，公司总裁的模型是 6 个分公司，每个分公司有一个副总裁向他汇报，他通过看每个分公司的业绩来看全公司的业绩。当要求产品部门增加利润时，他无须考虑产品部门副总裁的模型。

当副总裁给产品部门的每个小部门经理下达命令时，他无须考虑小部门经理的模型。让总裁亲自处理小部门层的事务几乎是不可能的，整个公司有太多小部门层的细节，不可能由一个人去管理。

App7层的计算机用户就像总裁，他给HOL6层的程序员写的程序发出诸如“计算所有大二学生的平均分”的指令，他无须考虑HOL6层模型怎样发布指令。最终，这条App7层指令逐层向下传送到LG1层。最终结果是App7层的用户能够用非常简化的计算机模型控制大量的电子电路和信号。

137

总结

二进制数只可能是两个数值之一。在机器层上，计算机以二进制形式存储信息。位是二进制数字，不是0就是1。非负整数使用无符号二进制表示。最右位是1的位置，它左边的一位是2的位置，再左边一位是4的位置，以此类推。每左边一位的位置值都是前一位位置值的两倍。有符号整数采用补码表示，其中第一位是符号位，剩余的位决定该数的数值大小。对于正数来说，补码表示与无符号表示相同；而对于负数来说，它的补码可以通过对应正数的反码加1得到。

每个二进制整数，无论有符号还是无符号，都有表数范围，这是由内存单元的位数决定的。单元的位数越小，表数范围就越有限。进位位C用来标识无符号整数是否超出表数范围，而溢出位V用来标识补码表示的数是否超出表数范围。二进制整数的运算包括ADD、AND、OR和NOT。ASL表示算术左移，实际上是对一个二进制值乘以2；而ASR表示算术右移，是对一个二进制数除以2。

十六进制数系统，基数为16，提供了一种简洁的表示位模式的方法。十六进制的16个数字是0、1、2、3、4、5、6、7、8、9、A、B、C、D、E和F。一个十六进制数字表示4位。美国信息交换标准代码，简称ASCII，是一种存储字符的常见编码方式。它是一种7位编码，可以表示128个字符，包括英语字母表的大小写字母、十进制数字、标点符号和不可打印的控制字符。

浮点数的存储单元包括3个字段：1位的符号字段、指数字段和尾数字段。除了特殊数值外，数字以二进制科学计数法方式存储，二进制小数点左边的隐藏位假定为1。指数以余码方式存储。4个特殊值是零、无穷大、NaN和非规格化数。IEEE 754标准将指数和尾数字段的位数定义为单精度8位和23位，双精度11和52位。

各个抽象层次的基本问题是待处理信息的形式与表达它的语言之间的不匹配。机器语言书写的程序处理位，高级语言书写的程序处理数字和记录这样的对象。无论程序写在哪个层次上，信息必须装进某种语言能够识别的形式中。将信息和语言进行匹配是所有抽象层次上的基本问题，是解决问题的建模过程中近似产生的根源。

138

练习

3.1节

*1. 数出下列数字后面的10个数（a）八进制从267开始，（b）三进制从2102开始，（c）二进制从10101开始，（d）五进制从2433开始。

2. 数出下列数字后面的10个数（a）八进制从466开始，（b）三进制从1201开始，（c）二进制从11011开始，（d）五进制从3434开始。

*3. 将下列数字从二进制转换到十进制，假定是无符号二进制表示法：
(a) 10010 (b) 110 (c) 1011 (d) 1000 (e) 11111 (f) 101010

4. 将下列数字从二进制转换到十进制，假定是无符号二进制表示法：
(a) 10010 (b) 10 (c) 1011 (d) 10000 (e) 1111 (f) 11110000

*5. 将下列数字从十进制转换到二进制，假定是无符号二进制表示法：
(a) 25 (b) 16 (c) 1 (d) 14 (e) 5 (f) 41

6. 将下列数字从十进制转换到二进制，假定是无符号二进制表示法：
(a) 12 (b) 35 (c) 3 (d) 0 (e) 27 (f) 16

7. 采用无符号二进制表示法，下列单元用二进制和十进制表示的表数范围是什么？
*(a) 2 位单元 *(b) 3 位单元 (c) 4 位单元
(d) 5 位单元 (e) n 位单元

*8. 执行下面的无符号加法运算，假定是 7 位单元。显示进位位的结果。

(a)
```
  010 1011
ADD 100 1001
C =
```

(b)
```
  101 1001
ADD 011 0111
C =
```

(c)
```
  111 1111
ADD 111 1111
C =
```

(d)
```
  111 1111
ADD 000 0001
C =
```

139

9. 执行下面的无符号加法运算，假定是 9 位单元。显示进位位的结果。

(a)
```
  0 1000 1011
ADD 0 1101 0001
C =
```

(b)
```
  1 0001 1101
ADD 0 1110 1000
C =
```

(c)
```
  1 1111 1111
ADD 0 0000 0001
C =
```

(d)
```
  1 1111 1111
ADD 1 1111 1111
C =
```

10. 假定是 12 位单元，求与 0110 0101 0111 相加，和为全 0 的二进制数，即求下面运算中缺失的数字。

```
      0110  0101  0111
ADD   ????  ????  ????
-----------------------
      0000  0000  0000
```

求出的数可能会把进位位置为 1。不看 3.2 节，你能给出求任意数的加十缺失数字的通用规则吗？提示：一个简单的规则涉及 NOT 运算。

11. 根据 3.1 节，你可以通过看 1 的位置上的数字来确定二进制数是奇数还是偶数。这个规则对任何基数都可能吗？请解释。

12. 八进制和十进制之间的转换类似于二进制和十进制之间的转换。*(a) 写出图 3-4 所示的八进制数 70146 的多项式表达。(b) 使用图 3-5 的技巧，把 7291 (dec) 转换为八进制。

13. 二进制小数类似于十进制小数，它包含一个二进制小数点，而不是十进制小数点。*(a) 写出图 3-4 所示的十进制数 29.458 的多项式表达。(b) 写出图 3-4 所示的二进制数 1011.100101 的多项式表达。(c) 在 (b) 中的二进制数的十进制值是什么？

14. 为何 ISA3 层的程序员会混淆万圣节 (Halloween) 和圣诞节 (Christmas)？提示：31 (oct) 等于什么？

3.2 节

*15. 将下列数从十进制转换到二进制，假定用 7 位补码二进制表达式：
(a) 49 (b) −27 (c) 0
(d) −64 (e) −1 (f) −2
(g) 这台计算机二进制和十进制的数值范围是什么？

140

16. 将下列数从十进制转换到二进制，假定用9位补码二进制表达式：

(a) 51　(b) -29　(c) -2

(d) 0　(e) -256　(f) -1

(g) 这个单元二进制和十进制的数值范围是什么？

17. 将下列数从二进制转换到十进制，假定是7位补码二进制表达式：

(a) 001 1101　(b) 101 0101　(c) 111 1100

(d) 000 0001　(e) 100 0000　(f) 100 0001

18. 将下列数从二进制转换到十进制，假定是9位补码二进制表达式：

(a) 0 0001 1010　(b) 1 0110 1010　(c) 1 1111 1100

(d) 0 0000 0001　(e) 1 0000 0000　(f) 1 0000 0001

*19. 执行下面的加法运算，假定是7位补码二进制表示。显示状态位的结果：

(a)
```
      010 1011
ADD   000 1110
-------------
N =
Z =
V =
C =
```

(b)
```
      111 1001
ADD   000 1101
-------------
N =
Z =
V =
C =
```

(c)
```
      100 0110
ADD   101 0101
-------------
N =
Z =
V =
C =
```

(d)
```
      110 0001
ADD   111 0101
-------------
N =
Z =
V =
C =
```

(e)
```
      000 1101
ADD   011 0100
-------------
N =
Z =
V =
C =
```

(f)
```
      100 1001
ADD   010 1011
-------------
N =
Z =
V =
C =
```

20. 执行下面的加法运算，假定是9位补码二进制表示。显示状态位的结果：

(a)
```
      0 1010 1100
ADD   0 0011 1010
-----------------
N =
Z =
V =
C =
```

(b)
```
      1 1110 0101
ADD   0 0011 0101
-----------------
N =
Z =
V =
C =
```

[141]

(c)
```
      0 0101 1011
ADD   1 0101 0100
-----------------
N =
Z =
V =
C =
```

(d)
```
      1 1000 0101
ADD   0 1101 0110
-----------------
N =
Z =
V =
C =
```

(e)
```
      0 0011 0100
ADD   0 1101 0010
-----------------
N =
Z =
V =
C =
```

(f)
```
      1 0010 0111
ADD   0 1010 0111
-----------------
N =
Z =
V =
C =
```

21. 用补码二进制表示，下列单元二进制和十进制表数范围是什么？

*(a) 2 位单元　(b) 3 位单元　(c) 4 位单元

(d) 5 位单元　(e) n 位单元

3.3 节

*22. 假定是 7 位单元，执行下面的逻辑运算：

(a)
```
    010 1100
ADD 110 1010
-----------
N =
Z =
```

(b)
```
    000 1111
ADD 101 0101
-----------
N =
Z =
```

(c)
```
    010 1100
OR  110 1010
-----------
N =
Z =
```

(d)
```
    000 1111
OR  101 0101
-----------
N =
Z =
```

(e)
```
    010 1100
XOR 110 1010
-----------
N =
Z =
```

(f)
```
    000 1111
XOR 101 0101
-----------
N =
Z =
```

(g) NEG 010 1100　(h) NOT 110 1010

23. 假定是 9 位单元，执行下面的逻辑运算：

(a)
```
    0 1001 0011
ADD 1 0111 0101
---------------
N =
Z =
```

(b)
```
    0 0000 1111
ADD 1 0111 0101
---------------
N =
Z =
```

(c)
```
    0 1001 0011
OR  1 0111 0101
---------------
N =
Z =
```

(d)
```
    0 0000 1111
OR  1 0111 0101
---------------
N =
Z =
```

(e)
```
    0 1001 0011
XOR 1 0111 0101
---------------
N =
Z =
```

(f)
```
    0 0000 1111
XOR 1 0111 0101
---------------
N =
Z =
```

(g) NEG 1 1001 0011　(h) NOT 1 0111 0101

*24. 假定是 7 位补码二进制表示，将下列数字从十进制转换到二进制，给出 ASL 运算的结果，再把它转换回十进制。用 ASR 运算再做一次。对于 ASL，给出对 NZVC 位的影响；对于 ASR，给出对 NZC 位的影响。

(a) 24　(b) 37　(c) −26

(d) 1　(e) 0　(f) −1

25. 假定是 9 位补码二进制表示，将下列数字从十进制转换到二进制，给出 ASL 运算的结果，接着把它转换回十进制。用 ASR 运算再做一次。对于 ASL，给出对 NZVC 位的影响；对于 ASR，给出对 NZC 位的影响。

(a) 94　(b) 135　(c) −62

(d) 1　(e) 0　(f) −1

26. (a) 写出 6 位单元算术右移的 RTL 描述。(b) 写出 16 位单元算术左移的 RTL 描述。

*27. 假定是 7 位单元，给出在 C 的初始值下，各个数的循环位移运算结果：

(a) C = 1, ROL 010 1101　(b) C = 0, ROL 010 1101

(c) C = 1, ROR 010 1101　(d) C = 0, ROR 010 1101

28. 假定是 9 位单元，给出在 C 的初始值下，各个数的循环位移运算结果：

(a) C = 1, ROL 0 0110 1101　(b) C = 0, ROL 0 0110 1101

(c) C = 1, ROR 0 0110 1101　(d) C = 0, ROR 0 0110 1101

29. (a) 写出 6 位单元循环右移的 RTL 表述。(b) 写出 16 位单元循环左移的 RTL 表述。

3.4 节

30. 从下面的数开始，往后数 5 个十六进制数：

*(a) 3AB7　(b) 6FD　(c) B9E

31. 将下列十六进制数转换到十进制：

*(a) 2D5E　　(b) 2F　　(c) 7

32. 本章提到了从十进制到十六进制的转换方法，但是没有给出例子。采用该方法把下列数从十进制转换到十六进制：

142 ~ 143

*(a) 26831　　(b) 4096　　(c) 9

33. 把十进制数转换到二进制数的方法稍加改动，就能把十进制数转换到任何基数。(a) 解释从十进制转换到八进制的方法；(b) 解释从十进制转到基数 n 的方法。

*34. 假定是7位补码二进制表示，将下面的数从十六进制转换到十进制，记得要检查符号位：

(a) 5D　　(b) 2F　　(c) 40

35. 假定是9位补码二进制表示，将下面的数从十六进制转换到十进制，记得要检查符号位：

(a) 1B4　　(b) 0E5　　(c) 100

*36. 假定是7位补码二进制表示，写出下面十进制数的十六进制位模式：

(a) −27　　(b) 63　　(c) −1

37. 假定是9位补码二进制表示，写出下面十进制数的十六进制位模式：

(a) −73　　(b) −1　　(c) 94

*38. 将下面加密的ASCII消息解码（慢着读）

100 1000　110 0001　111 0110　110 0101
010 0000　110 0001　010 0000　110 1110
110 1001　110 0011　110 0101　010 0000
110 0100　110 0001　111 1001　010 0001

39. 将下面加密的ASCII消息解码（慢着读）

100 1101　110 0101　110 0101　111 0100
010 0000　110 0001　111 0100　010 0000
110 1101　110 1001　110 0100　110 1110
110 1001　110 0111　110 1000　111 0100
010 1110

*40. 下面的9个字符的字符串是以ASCII怎样存储的？

Pay $0.92

41. 下面的13个字符的字符串是以ASCII怎样存储的？

(321)497-0015

42. 你是和上斯洛波维亚打仗的下斯洛波维亚军队的首席通信官，为了获得斯洛波维亚的领地，你的间谍将潜入敌人的指挥中枢。你知道上斯洛波维亚正在策划一次重要的攻击，你也知道下列情况：(1) 攻击的时间是日落或日出，(2) 攻击将通过陆地、空中或者大海，(3) 攻击将会在3月28、29、30、31或者4月1日进行。你的间谍必须用二进制和你通信，设计一个合适的二进制编码用来传递这些信息，尽可能使用最少的位数。

144

43. 有时候八进制用于代替十六进制来表示位序列。

*(a) 一个八进制数代表多少位？

在下面的单元，如何用八进制来表示十进制数 −13？

(b) 15位单元　　(c) 16位单元　　(d) 8位单元

3.5节

*44. 把下列数从二进制转换到十进制：

(a) 110.101001　　(b) 0.000011　　(c) 1.0

45. 把下列数从二进制转换到十进制：

(a) 101.101001　　(b) 0.000101　　(c) 1.0

*46. 把下列数从十进制转换到二进制：

(a) 13.156 25　　(b) 0.039 062 5　(c) 0.6

47. 把下列数从十进制转换到二进制：

(a) 12.281 25　　(b) 0.023 437 5　(c) 0.7

48. 做一个类似图 3-30 的表，可以比较 4 位单元的所有余 7 码和补码。

49. (a) 用余 7 码表示法，4 位单元以二进制和十进制表示的表数范围是什么？(b) 用余 15 码表示法，5 位单元以二进制和十进制表示的表数范围是什么？(c) 用余 $2^{n-1}-1$ 码表示法，n 位单元以二进制和十进制表示的表数范围是什么？

50. 假定是 3 位指数字段和 4 位有效位数，写出下列十进制值的位模式：

*(a) −12.5　　(b) 13.0　　(c) 0.43　　(d) 0.101 562 5

(d) 0.5　　(e) 0.6　　(f) 256.015 625

51. 假定是 3 位指数字段和 4 位有效位数，下面位模式表示的十进制值是什么：

*(a) 0 010 1101　　(b) 1 101 0110　　(c) 1 111 1001

(d) 0 001 0011　　(e) 1 000 0100　　(f) 0 111 0000

52. 采用 IEEE 754 单精度浮点数表示法，写出下面十进制值的十六进制表示：

*(a) 27.101 562 5　(b) −1.0　　(c) −0.0

(d) 0.5　　(e) 0.6　　(f) 256.015 625

53. 采用 IEEE 754 单精度浮点数表示法，下面用十六进制表示的数的二进制科学计数法表示是什么：

*(a) 4280 0000　　(b) B350 0000　　(c) 0061 0000

(d) FF80 0000　　(e) 7FB4 0000　　(f) 8000 0000

54. 采用 IEEE 754 单精度浮点数表示法，写出下列数的十六进制表示：

(a) 正零　　(b) 最小正的非规格化数　　(c) 最大正的非规格化数

(d) 最小正的规格化数　(e) 1.0　　(f) 最大正的规格化数

(g) 正无穷大

55. 采用 IEEE 754 双精度浮点数表示法，写出下列数的十六进制表示：

(a) 正零　　(b) 最小正的非规格化数　　(c) 最大正的非规格化数

(d) 最小正的规格化数　(e) 1.0　　(f) 最大正的规格化数

(g) 正无穷大

问题

3.1 节

56. 用 C++ 写一个程序，输入一个 4 位八进制数，打印它后面的 10 个八进制数。用 `int octNum[4];` 来定义一个八进制数，用 `octNum[0]` 存储最高的（即最左的）八进制位，`octNum[3]` 为最低的八进制位。采用交互式输入来测试你的程序。

57. 用 C++ 写一个程序，输入一个 8 位二进制数，打印它后面的 10 个二进制数。用 `int binNum[4];` 来定义一个八进制数，用 `binNum[0]` 存储最高的（即最左的）位，`binNum[7]` 为最低位。请用户输入第一个二进制数字，每位用至少一个空格分开。

58. 用 C++ 写一个函数，把 8 位无符号二进制数转换为十进制正整数，使用问题 57 中的二进制数定义。采用交互式输入来测试你的函数。 146

59. 用 C++ 写一个空函数，把十进制正整数转换为 8 位无符号二进制数，使用问题 57 中的二进制数定义。采用交互式输入来测试你的空函数。

60. 像问题 57 那样定义一个二进制数，写一个空函数

```
void BinaryAdd (int* sum, int& cout,
                const int* bin1, const int* bin2)
```

来计算两个二进制数 bin1 和 bin2 的和 sum。cBit 是加法运算后进位位的值。采用交互式输入来测试你的空函数。

3.3 节

61. 用 C++ 写一个函数，把 8 位补码二进制数转换为十进制整数，使用问题 57 的二进制数定义。采用交互式输入来测试你的函数。

62. 用 C++ 写一个空函数，把十进制整数转换为 8 位补码二进制数，使用问题 57 的二进制数定义。采用交互式输入来测试你的空函数。

3.2 节

63. 像问题 57 那样定义一个二进制数，写一个空函数

```
void BinaryAnd (int* bAnd,
                const int* bin1, const int* bin2)
```

来计算两个二进制数 bin1 和 bin2 的 AND 运算值 bAnd。采用交互式输入来测试你的空函数。

64. 写一个问题 63 那样的函数，使用 OR 运算。

65. 像问题 57 那样定义一个二进制数，写一个函数

```
void ShiftLeft (int* binNum, int& cBit)
```

来对 binNum 执行算术左移运算，cBit 是位移后的进位位的值。采用交互式输入来测试你的函数。

66. 写一个问题 65 那样的函数，执行算术右移运算。

3.4 节

67. 用 C++ 写一个程序，输入一个 4 位十六进制数，打印它后面的 10 个十六进制数。用 int hexNum[4] 定义一个十六进制数，十六进制的输入/输出采用大写字母，例如，3C6F 是合法的输入。

147

68. 用 C++ 写一个函数，把 4 位十六进制数转换为十进制正整数，使用问题 67 中十六进制数的定义。采用交互式输入测试你的函数，十六进制输入采用大写字母。

69. 用 C++ 写一个空函数，把十进制正整数转换为 4 位十六进制数，使用 67 题中十六进制数的定义。采用交互式输入测试你的空函数。十六进制输出采用大写字母。

70. 用 C++ 写一个函数，把 4 位十六进制数转换为可能为负的十进制整数，使用问题 67 中十六进制数的定义，假定十六进制值代表一个补码表示的 16 位单元。采用交互式输入测试你的空函数，十六进制输入采用大写字母。

71. 用 C++ 写一个空函数，把可能为负的十进制整数转换为 4 位十六进制数，使用问题 67 中十六进制数的定义，假定十六进制值代表一个补码表示的 16 位单元。采用交互式输入测试你的空函数，十六进制输出采用大写字母。

72. 用 C++ 写一个函数，把一个任意基数的正数转换到十进制。例如，对于 4 位、基数为 6 的数字，声明：

```
const int base = 6;
const int numDigits = 4;
int number[numDigits];
```

采用交互式输入测试你的函数。把要转换的数读入一个字符数组，如果基数值需要，使用大写字母输入。写一个空函数，在转换为十进制前，把它转换为适当的 int 数组类型的值。

必须能够通过仅改变常量 base 就可以修改你的程序，使其用于其他不同基数的运算；必须能够通过仅改变常量 numDigits 就可以修改程序，使其用于不同的位数。

73. 用 C++ 写一个空函数，把一个十进制正整数转换为任意基数的数。number 的声明与问题 72 中一样。采用交互式输入测试你的程序，如果基数值需要，使用大写字母输出。

必须能够通过只改变常量 base 就可以修改你的程序，使其用于其他不同基数的运算；必须能够通过仅改变常量 numDigits 就可以修改程序，使其用于不同的位数。

148

计算机体系结构

建筑师把墙、门和天花板这些部件组合在一起形成建筑物。类似地，计算机架构师把输入设备、内存和CPU寄存器组合在一起形成计算机。

建筑物有各种形状和大小，计算机也是。这就提出一个问题：如果我们选择一台计算机，研究几十种流行的可用模型，那么当这些模型不可避免地被生产商弃用时，我们的知识多多少少都会有些过时。同样，对于那些使用我们没有选研究的计算机的人，这本书的价值也会较低。

但是又有另一种可能性。一本关于建筑学的书可以观察一栋假想的建筑物，同样道理，这本书可以探索一台虚拟的计算机，这台虚拟计算机包含类似于能在所有真实计算机上找到的特性。这种方法有它的优势和劣势。

虚拟计算机的一个好处是它可以被设计成仅用以说明适用于大多数计算机系统的基本概念，那么我们可以专注于要点而不用理会真实计算机各自的奇特属性。专注于基本原理也能避免知识过时，市场上各种计算机来来去去，基本原理总会继续适用。

学习虚拟计算机的主要劣势是，它的一些细节对在汇编语言层或指令集架构层使用特定真实机器工作的人来说关系不大。不过，如果理解了基本概念，就可以很容易地学会任何特定机器的细节。

对于这个两难困境没有100%满意的解决方案。我们选择虚拟计算机的方法是因为它能够阐释基础概念。我们假设的机器叫作Pep/8计算机。

4.1 硬件

Pep/8硬件在指令集架构层（ISA3层）主要由4部分组成：

- 中央处理单元（CPU）
- 主存储器
- 输入设备
- 输出设备

图4-1的框图把每个组成部分用一个方框表示。总线是连接4个主要组成部分的一组线路，它承载方框之间传送的数据信号和控制信号。

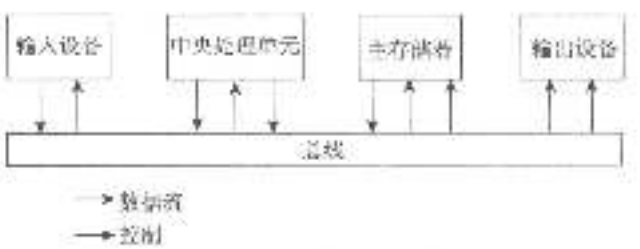


图4-1 Pep/8计算机的组成框图

4.1.1　中央处理单元

中央处理单元（CPU）包含6个专用内存单元，叫作寄存器。如图4-2所示，它们是

- 4位状态寄存器（NZVC）
- 16位累加器（A）
- 16位变址寄存器（X）
- 16位程序计数器（PC）
- 16位栈指针（SP）
- 24位指令寄存器（IR）

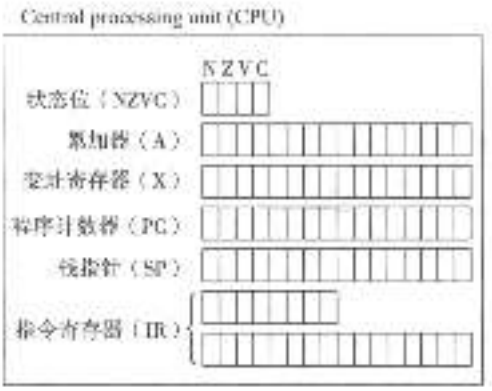


图4-2　Pep/8计算机的中央处理单元

状态寄存器中的N、Z、V和C与3.1节和3.2节讨论的一样，分别是负、零、溢出和进位位。累加器是包含运算结果的寄存器。接下来的3个寄存器——X、PC和SP，帮助CPU存取主存中的信息。变址寄存器用来访问数组的元素，程序计数器用来访问指令，栈指针用来访问运行时栈上的元素。指令寄存器保存从内存中取出的指令。

除了这6个寄存器外，CPU还包含执行Pep/8指令的所有电子器件（在图4-2中未显示）。

4.1.2　主存储器

图4-3展示了Pep/8计算机的主存储器。它包含65 536个8位内存单元。一个8个位的组称为1字节（byte，读作bite）。类似于邮箱上的数字地址，每字节都有一个地址，地址范围以十进制表示是从0～65535，十六进制表示是从0000～FFFF。主存储器有时称为核心存储器。

图4-3在第一行显示了主存的前3个字节，第二行是接下来的字节，再下一行是接下来的3个字节，而在最后一行是最后2个字节。把一行内存看作包含1个、2个还是3个字节取决于问题的上下文。有时候，把一行看作一个字节更方便，而有时是2个或者3个更方便。当然，在物理计算机里，1个字节是存储于电子电路中的8个信号序列。字节在物理上的排列不会像图中所示的那样。

通常如图4-4所示的那样画主存是很方便的，把地址写在块的左边。尽管看上去每行方框的宽度相等，但是一行可以代表1个字节或者几个字节。方框边上的地址是本行最左边字节的地址。

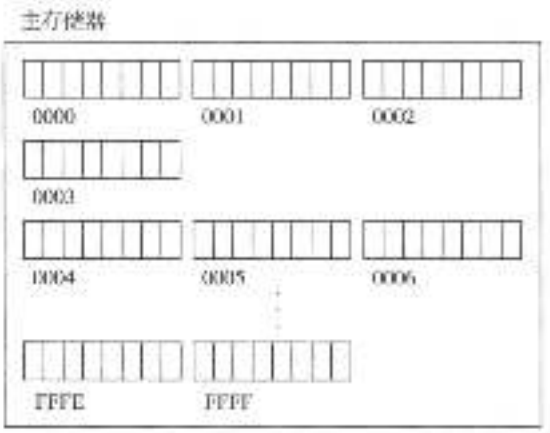


图4-3　Pep/8计算机的主存储器



图4-4　另一种描述主存储器的方式

可以通过地址序列知道一行包含多少字节。在图4-4中，第1行一定是3个字节，因为第2行的地址是0003；第2行一定是1个字节，因为第3行的地址是0004，比0003大1。类似地，第3行和第4行每行有3个字节，第5行有1字节，第6行有2个字节，从图4-4看出，不可能知道第7行有多少字节。图4-4的前3行对应图4-3的前7个字节。

无论在纸上怎么画主存字节，都把小地址的字节称为内存的"顶部"，大地址的字节称为内存的"底部"。

大多数计算机厂商指定一个字是一定数量的字节。在Pep/8计算机中，一个字是两个相邻字节，因此，一个字包含16位。在Pep/8 CPU中的大多数寄存器是字寄存器。在主存中，一个字的地址是这个字第一个字节的地址。例如，图4-5a展示了地址为000B和000C的两个相邻字节，这个16位字的地址是000B。



图4-5 内存位置的内容和它的地址之间的差别

区别内存单元的内容和它的地址是非常重要的。Pep/8计算机的内存地址长度是16位，因此图4-5a中字的二进制地址是0000 0000 0000 1011，但是位于这个地址的字的内容是0000 0010 1101 0001。不能把字的内容和它的地址搞混，它们是不同的。

为了节约纸面上的空间，字节或字的内容通常用十六进制表示。图4-5b给出了地址000B的字以十六进制表示的内容。在机器语言代码中，给出一组字节的第一个字节的地址，然后给出十六进制表示的内容，如图4-5c所示。用这种格式表示，尤其容易混淆字节的地址和它的内容。

在图4-5的例子中，对内存单元的内容有多种解读方法。如果把位序列0000 0010 1101 0001看作补码表示的整数，那么第一个位就是符号位，这个二进制序列表示十进制数721。如果把最右边7位看作ASCII码字符，那么这个二进制序列表示字符Q。主存不会决定以何种方式来解读这个字节，它只记录二进制序列0000 0010 1101 0001。

150
↓
152

4.1.3 输入设备

你可能想知道Pep/8硬件在哪里，你能不能真的接触它。答案是，这个硬件根本不存在！至少作为一台物理机器它是不存在的，不过作为一组可以在计算机系统上执行的程序，它是存在的。这些程序模拟这些章节描述的Pep/8机器的行为。

Pep/8系统模拟两种输入设备——文本文件和键盘。在一个Pep/8程序中不能同时指定这两者。在执行一个程序前，必须指定输入来自于文件还是键盘。如果正在使用一个有图形用户接口的模拟器，那么输入将来自于焦点窗口而不是文件。

4.1.4 输出设备

Pep/8系统也模拟两种输出设备——文本文件和屏幕。与输入一样，不可以在一个Pep/8程序中同时指定这两者。如果指定文本文件作为输出，那么系统会要求给出输出文件的文件名，然后系统会生成该文件。如果正在使用一个有图形用户接口的模拟器，那么将输出到一个新窗口，可以再把它保存到一个新文件中。

4.1.5　数据和控制

图 4-1 中连接方框的实线是数据流线。数据可以从总线上的输入设备流向主存，也可以从总线上的主存流向 CPU，但是不能从输入设备直接流向 CPU。

类似地，数据不能直接从 CPU 流向输出设备。如果想把数据从 CPU 传送到输出设备，必须先传送到主存，再从主存传送到输出设备。

虚线是控制线。控制信号都是从 CPU 发出的，也就是说，CPU 控制计算机的所有其他部分。例如，要使数据从主存沿着实数据流线流到输出设备，CPU 必须沿着虚控制线给内存传送一个发送信号，给输出设备一个接收信号。处理器是真正的中心，它控制计算机所有
153 其他的部分。

4.1.6　指令格式

每种计算机都有它自己的指令集，固化在 CPU 中。厂商之间的指令集是不同的。许多厂商生产一个系列的型号，每个型号和本系列中其他型号具有相同的指令集，但是同一家公司制造的计算机指令集经常不一样。

Pep/8 计算机的指令集有 39 条指令，如图 4-6 所示。一条指令要么是一个字节，称为指令指示符（instruction specifier），要么是指令指示符后紧跟一个称为操作数指示符（operand specifier）的字。没有操作数指示符的指令叫作一元指令。图 4-7 给出了非一元指令和一元指令。

指令指示符	指　令	指令指示符	指　令
0000 0000	停止执行	0010 01nn	未实现操作码，一元陷阱
0000 0001	从陷阱返回	0010 1aaa	未实现操作码，非一元陷阱
0000 0010	将栈指针（SP）传送到累加器（A）	0011 0aaa	未实现操作码，非一元陷阱
0000 0011	将 NZVC 标志传送到累加器（A）	0011 1aaa	未实现操作码，非一元陷阱
0000 010a	无条件分支	0100 0aaa	未实现操作码，非一元陷阱
0000 011a	如果小于等于分支	0100 1aaa	字符输入
0000 100a	如果小于分支	0101 0aaa	字符输出
0000 101a	如果等于分支	0101 1nnn	从调用返回，将 n 个本地字节
0000 110a	如果不等于分支	0110 0aaa	加到栈指针（SP）
0000 111a	如果大于等于分支	0110 1aaa	从栈指针（SP）减去
0001 000a	如果大于分支	0111 raaa	加到寄存器 r
0001 001a	如果 V 分支	1000 raaa	从寄存器 r 减去
0001 010a	如果 C 分支	1001 raaa	与寄存器 r 按位与
0001 011a	调用子程序	1010 raaa	与寄存器 r 按位或
0001 100r	寄存器 r 求位反转	1011 raaa	与寄存器 r 比较
0001 101r	寄存器 r 取反	1100 raaa	从内存加载到寄存器 r
0001 110r	寄存器 r 算数左移	1101 raaa	从内存加载一个字节到寄存器 r
0001 111r	寄存器 r 算数右移	1110 raaa	把寄存器 r 存入内存
0010 000r	寄存器 r 循环左移	1111 raaa	从寄存器 r 存一个字节到内存
0010 001r	寄存器 r 循环右移		

图 4-6　Pep/8 的 ISA3 层指令集

8位指令指示符分为多个部分。第一部分叫作操作码，常称为opcode。操作码是4～8位，例如，图4-6显示了把栈指针移到累加器的指令有8位操作码0000 0010，而字符输入指令有5位操作码0100 1。操作码少于8位的指令，根据指令的不同，它们的指令指示符细分为多个字段。图4-6用字母a、r和n来标识这些字段，每个字母可以为0或1。

例4.1 图4-6给出"branch if equal to"指令的指令指示符为0000 101a。因为字母a可以为0或1，所以指令有两种版本——0000 1010和0000 1011。类似地，十进制输出陷阱指令有8种版本，它的指令指示符是0011 1aaa，aaa可以是从000到111的任意组合。 □

图4-8总结了字母a和r在指令指示符中可能字段的含义。一元陷阱指令中的字母nn和从调用返回指令中的字母nnn的含义在后面的章节中进行描述。

一般来说，字母a代表寻址方式，字母r代表寄存器。当r为0时，指令对累加器进行操作。当r为1时，指令对变址寄存器进行操作。Pep/8的非一元指令有8种可能的寻址方式——立即、直接、间接、栈相对、栈相对间接、变址、栈变址和栈变址间接，后面的章节会描述寻址方式的含义。目前，只需要了解怎样使用图4-7和图4-8中的表格来确定一条给定的指令使用的是什么寄存器和寻址方式。



图4-7 Pep/8的指令格式

aaa	寻址方式
000	立即
001	直接
010	间接
011	栈相对
100	栈相对间接
101	变址
110	栈变址
111	栈变址间接

a) aaa寻址字段

a	寻址方式
0	立即
1	变址

b) a寻址字段

r	寄存器
0	累加器，A
1	变址寄存器，X

c) 寄存器r字段

图4-8 Pep/8的指令指示符字段

例4.2 确定1100 1011指令的操作码、寄存器和寻址方式。从左边开始，根据图4-6，操作码是1100。操作码后面的1位是r位，值为1，表示变址寄存器，r位后面是aaa位，值为011，表示栈相对寻址。因此这条指令使用栈相对寻址方式将内存中一个值载入变址寄存器中。 □

对于非一元指令，操作数指示符表明指令要处理的操作数。根据指令指示符中的某些位，CPU有多种不同的解读操作数指示符的方法。例如，可以把操作数指示符当作ASCII字符、补码表示的整数或者存储操作数的主存地址。

指令存储于主存中，一条指令的地址是该指令第一个字节的地址。

例4.3 图4-9展示了存储于主存地址01A3和01A6的两条相邻的指令。01A6的指令是一元指令，01A3的指令则不是。

在这个例子中，01A3的指令表示

操作码：1000

寄存器-r字段：1

寻址-aaa字段：101

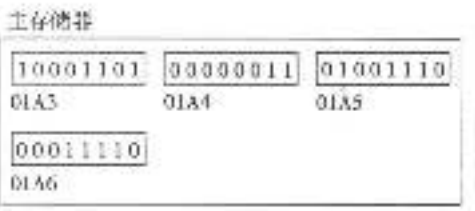


图4-9 主存中的两条指令

操作数指示符：0000 0011 0100 1110

这里，所有的数都是二进制的。根据图 4-6 的操作码表，这是减法指令。寄存器 -r 字段表明要操作的是变址寄存器而不是累加器。这条指令是把变址寄存器的内容减去操作数。寻址 -aaa 字段表示变址寻址，所以对操作数指示符做相应的解读。在本章中，我们的学习仅限于直接寻址，其他模式会在后面的章节中讲解。

01A6 的一元指令表示

操作码：0001 111

寄存器 -r 字段：0

操作码表示这条指令是做算术右移，寄存器 -r 字段表明要进行右移的是累加器。因为这是个一元指令，所以没有操作数指示符。□

在例 4.3 中，下面的指令格式称为机器语言：

1000 1101 0000 0011 0100 1110

0001 1110

机器语言（machine language）是二进制序列（即 0 和 1 的序列），CPU 根据它的指令集操作码进行解读。机器语言代码会在内存地址后面用十六进制表示这两条指令，如下所示：

01A3　8D034E

01A6　1E

如果只有指令的十六进制表示，那么必须把它转换为二进制，检查指令指示符字段的内容来确定这条指令是做什么的。

4.2　直接寻址

154 ~ 157

本节讲述 ISA3 层上某些 Pep/8 指令的操作，这些指令如何以直接寻址方式进行运算。后面的章节将讲述其他寻址方式。

寻址字段决定了 CPU 怎样解读操作数指示符。寻址 -aaa 字段 001 表示直接寻址。如果采用直接寻址，CPU 会把操作数指示符解释为包含操作数的主存单元地址。用数学符号表示是

Oprnd = Mem[OprndSpec]

这里 Oprnd 表示操作数，OprndSpec 表示操作数指示符，Mem 表示主存。

方括号表示你可以把主存想象成一个数组，把操作数指示符想象成数组的索引。在 C++ 中，如果 v 是数组，i 是整数，则 v[i] 是由整数 i 的值所确定的数组中的“单元”。类似地，指令中的操作数指示符标识主存中包含操作数的单元。

接下来要讲的是某些 Pep/8 指令集的指令，每条指令的描述都会列出操作码，并举一个使用直接寻址的指令操作的例子。N、Z、V 和 C 的值都以二进制的形式给出，其他寄存器和内存单元的值是用十六进制表示的。在机器层，所有的值最终都是二进制的。在讲述完每条指令之后，本章结尾会展示怎样用它们一起组成机器语言程序。

4.2.1　停止指令

停止指令的指令指示符是 0000 0000。执行这条指令就是简单地让计算机停下来。Pep/8 是一台模拟的计算机，通过运行计算机上的模拟器来执行它。模拟器有命令选项菜单供你选择，选项之一就是执行 Pep/8 程序。当 Pep/8 程序执行时遇到这条指令时，就会停止并把模拟器返回到命令选项菜单。停止执行指令是一元指令，它没有操作数指示符。

4.2.2　装入指令

装入指令的指令指示符是 1100 raaa，它根据 r 的值，把 1 个字（2 字节）从内存单元装入累加器或者变址寄存器。它影响 N 位和 Z 位。如果操作数是负数，就把 N 位置为 1；否则，把 N 位置为 0。如果操作数是 16 位 0，就把 Z 位置为 1；否则，把 Z 位置为 0。装入指令的寄存器传送语言（RTL）描述是

r ← Oprnd; N ← r < 0, Z ← r = 0　158

例 4.4　假定要执行指令的十六进制表示是 C1004A，图 4-10 给出了二进制表示。本例中寄存器 -r 字段是 0，表示装入累加器而不是变址寄存器。寻址 -aaa 字段是 001，表示是直接寻址。



图 4-10　装入指令

假定 Mem[004A] 的初始内容是 92EF，图 4-11 展示了执行装入指令后的结果。装入指令不会改变内存单元的内容，它把两个内存单元（地址 004A 和 004B）内容的副本发送到寄存器。无论指令执行前寄存器中是什么，在本例中是 036D，它都会被覆盖。由于被装入的位模式的符号位是 1，所以 N 位被置为 1。该位模式不是全 0，因此 Z 位被置为 0。装入指令对 V 和 C 位没有影响。

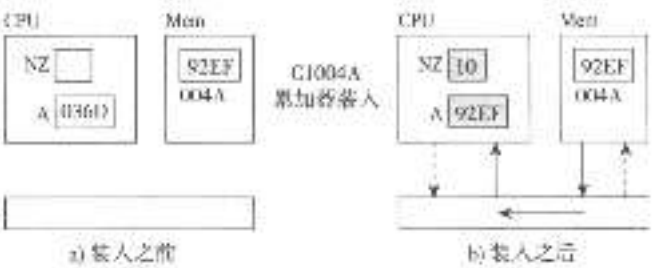


图 4-11　装入指令的执行

图 4-11 展示了装入指令的数据流和控制流。如实线所示，数据流从总线上的主存到 CPU，然后再到寄存器。为了进行此次数据传送，CPU 必须向主存发送控制信号，如虚线所示，让它把数据放到总线上。CPU 也会告知主存到哪个地址取数据。□

4.2.3　存储指令

存储指令的指令指示符是 1110 raaa，它把 1 个字（2 字节）从累加器或者变址寄存器存储到内存单元。对于直接寻址，操作数指定信息存储的内存单元。存储指令的 RTL 描述是

Oprnd ← r　159

例 4.5　假定要执行指令的十六进制表示是 E9004A，图 4-12 是它的二进制表示。这次，寄存器 -r 字段标明指令将作用于变址寄存器。寻址 -aaa 字段 001 表示直接寻址。

假定变址寄存器的初始内容是 16BC，图 4-13 展示了执行存储指令后的结果。存储指令不会改变寄存器的内容，它把寄存器内容的副本发送到两个内存单元（地址 004A 和 004B）。



图 4-12　存储指令

无论指令执行前内存单元中是什么，在本例中是 F082，它都会被覆盖。存储指令不影响任何状态位。 □

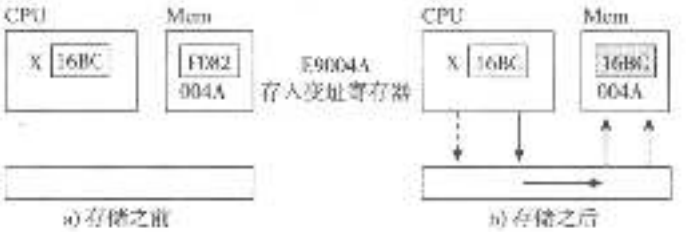


图 4-13　存储指令的执行

4.2.4　加法指令

加法指令的指令指示符是 0111 raaa。类似于装入指令，加法指令中数据也从主存传送到 CPU 的寄存器 r 中。但是，对于加法指令，寄存器的初始内容不会被来自主存的字内容改写，而是把这个字的内容和寄存器的内容相加，再把和放在这个寄存器中，并相应地设置 4 个状态位。与装入指令一样的是内存字的内容被传送到 CPU，而它的原始内容不会改变。加法指令的 RTL 描述是

r ← r + Oprnd；N ← r < 0，Z ← r = 0，V ← {溢出}，C ← {进位}

例 4.6　假定要执行指令的十六进制表示是 79004A，图 4-14 是它的二进制表示。寄存器 r 字段表示指令将作用于变址寄存器，寻址 aaa 字段 001 表示直接寻址。 160



图 4-14　加法指令

假定变址寄存器的初始内容是 0005，Mem[0004] 是 −7（dec）= FFF9（hex）。图 4-15 展示了执行加法指令后的结果。在十进制中，5 +(−7) 是 −2，在图 4-15b 中显示为 FFFE（hex）。图中 NZVC 位以二进制显示。因为和是负数，所以 N 位是 1。由于和不为全 0，所以 Z 位是 0。没有溢出，因此 V 位是 0。最高位没有进位，因此 C 位是 0。 □

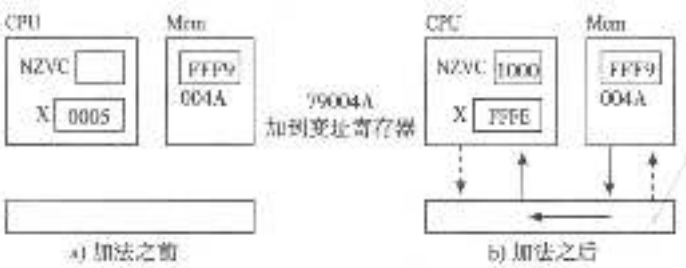


图 4-15　加法指令的执行

4.2.5 减法指令

减法指令的指令指示符是 1000 raaa。除了是把寄存器内容减去操作数外，它类似于加法指令。结果放在寄存器中，操作数不会被改变。对于减法，C 位表示加上操作数的负数之后的进位。减法指令的 RTL 描述是

r ← r–Oprnd; N ← r < 0, Z ← r = 0, V ← {溢出}, C ← {进位}

例 4.7 假定要执行指令的十六进制表示是 81004A，图 4-17 是它的二进制表示。寄存器 -r 字段表示指令将作用于累加器。

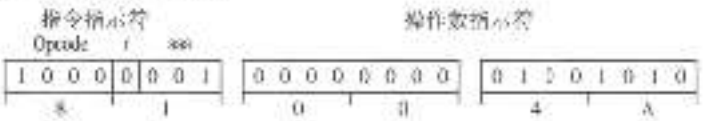


图 4-16 减法指令

假定累加器的初始内容是 0003，Mem[004A] 是 0009，图 4-17 显示了执行减法指令后的结果。在十进制中，3 减去 9 的差是 –6，在图 4-17b 中用十六进制表示为 FFFA（hex）。图中 NZVC 位以二进制显示，因为和是负数，所以 N 位是 1。由于和不是 0，所以 Z 位是 0。没有溢出，因此 V 位是 0。3 加 –9 时没有进位，因此 C 位是 0。 □

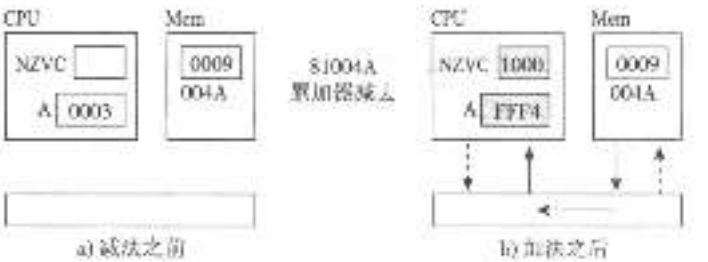


图 4-17 减法指令的执行

4.2.6 与和或指令

与（AND）指令的指令指示符是 1001 raaa，或（OR）指令的指令指示符是 1010 raaa，两条指令都类似于加法指令。这些指令对寄存器执行逻辑运算，而不是把操作数加到寄存器中。AND 运算可以帮助掩去位模式中不希望有的值为 1 的位，OR 运算用于在位模式中往某些位上插入 1。这两条指令对 N 和 Z 位都有影响，但不会改变 V 和 C 位。与和或指令的 RTL 描述是

r ← r ∧ Oprnd; N ← r < 0, Z ← r = 0

r ← r ∨ Oprnd; N ← r < 0, Z ← r = 0

例 4.8 假定要执行指令的十六进制表示是 99004A，图 4-18 是它的二进制表示。操作码表示要执行与指令，寄存器 -r 字段表示指令将作用于变址寄存器。

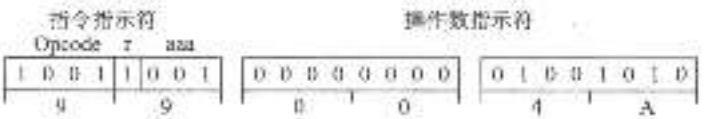


图 4-18 与指令

假定变址寄存器的初始内容是5DC3，Mem[004A]是00FF，图4-19展示了执行与指令后的结果。在二进制中，00FF是0000 0000 1111 1111，Mem[004A]每个值为1的位所对应的变址寄存器的位不改变，每个值为0的位所对应的寄存器位被置为0。图中以二进制展示NZ位，因为变址寄存器的数值不为负，所以N位是0。变址寄存器不全为0，因此Z位是0。□

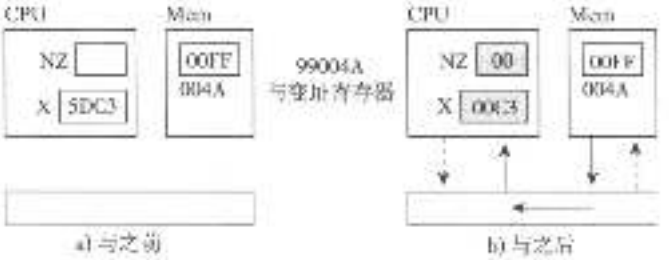


图4-19　与指令的执行

例4.9　图4-20展示了或指令的运算，除了指令指示符的操作码A9是1010外（表示它是或指令），初始状态和例4.8一样。这次，Mem[004A]每个值为0的位所对应的变址寄存器的位不会改变，每个值为1的位所对应的寄存器的位被置为1。如果把变址寄存器当作有符号整数，其值不为负，因此N位是0。□

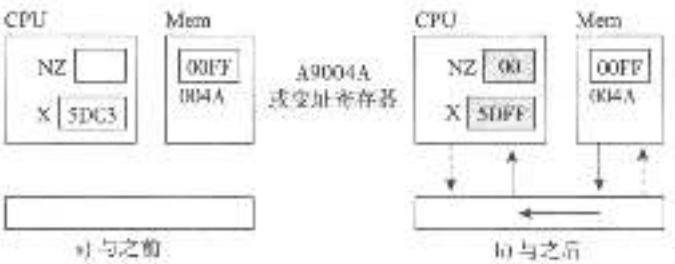


图4-20　或指令的执行

4.2.7　按位取反和取负指令

按位取反指令的指令操作符是0001 100r，取负指令的指令操作符是0001 101r。两条指令都是一元的，它们没有操作数指示符。按位取反指令对寄存器执行NOT运算，即每位1变为0，0变为1。它影响N和Z位。按位取反指令的RTL描述是

r←¬r；N←r<0，Z←r=0

取负指令把寄存器当作有符号整数并对它取负。16位寄存器存储有符号整数的范围是-32 768到32 767。取负指令影响N、Z和V位。因为没有对应的正值32 768，所以当寄存器原始值是-32768时，V位置为1。取负指令的RTL描述是

r←—r；N←r<0，Z←r=0，V←{溢出}

例4.10　假定要执行指令的十六进制表示是18，图4-21是它的二进制表示。操作码表示要执行按位取反指令，寄存器-r字段表示指令将作用于累加器。

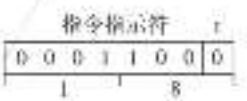


图4-21　按位取反指令

假定累加器的初始内容是0003（hex），即0000 0000 0000 0011（bin），图4-22展示了执行NOT运算的结果。NOT运算将位模式变为1111 1111 1111 1100，如果把它看作有符号整数，因为累加器的数

161
163

值为负，所以N位为1。因为累加器不为全0，所以Z位为0。

例4.11 图4-23展示了取负指令的运算，除了指令指示符的操作码1A为0001 101外（表示它是取负指令），初始状态和例4.10一样。3取负是-3，即1111 1111 1111 1101（bin）= FFFD（hex）。 □

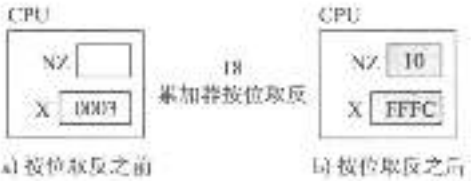


图4-22 按位取反指令的执行

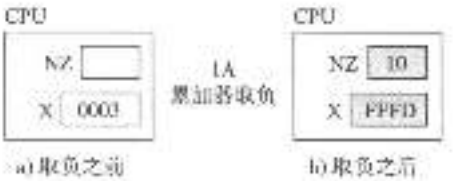


图4-23 取负指令的执行

4.2.8 装入字节和存储字节指令

这两条指令和接下来的两条都是字节指令，字节指令是对单个字节而不是一个字进行操作。装入字节指令的指令操作符是1101 raaa，存储字节指令的指令操作符是1111 raaa。装入字节指令把操作数装入累加器或变址寄存器的右半部分，影响N和Z位，保持寄存器左半部分不变。存储字节指令把累加器或者变址寄存器的右半部分存储到一个单字节内存单元，不影响任何状态位。装入字节指令的RTL描述是

r <8..15> ← byte Oprnd: N ← r < 0, Z ← r = 0

存储字节指令的RTL描述是

byte Oprnd ← r <8..15>

例4.12 假定要执行指令的十六进制表示是D1004A，图4-24是它的二进制表示。本例中寄存器-r字段是0，表示装入累加器而不是变址寄存器。寻址-aaa字段是001，表示是直接寻址。



图4-24 装入字节指令

假定Mem[004A]的初始内容是92。图4-25展示了执行装入字节指令后的结果。因为累加器最终位模式的符号位是0，所以N位为0。因为位模式不为全0，所以Z位为0。 □ 164 | 165

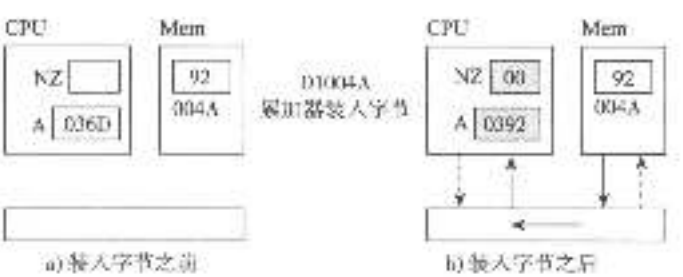


图4-25 取负指令的执行

例4.13 图4-26展示了执行存储字节指令后的结果。除了指令是存储字节而不是装入字节外，初始状态和例4.12一样。累加器的右半部分为6D，传送到地址004A的内

存单元。 □



图 4-26　存储字节指令的执行

4.2.9　字符输入和输出指令

字符输入指令的指令指示符是 0100 1aaa，字符输出指令的指令指示符是 0101 0aaa，两者都是字节指令。字符输入指令接收来自输入设备的下一个 ASCII 字符，并且把该字符相应的二进制编码存储在主存的一个字节中。字符输出指令把内存中一个字节的内容发送到输出设备，输出设备输出相应的 ASCII 字符。两条指令对 CPU 中任何寄存器都没有影响。字符输入指令的 RTL 描述是

byte Oprnd ← {字符输入}

字符输出指令的 RTL 描述是

166 {字符输出} ← byte Oprnd

例 4.14　假定要执行指令的十六进制表示是 49000A。图 4-27 是它的二进制表示，没有寄存器 r 字段，寻址 aaa 字段是 001，表示是直接寻址。

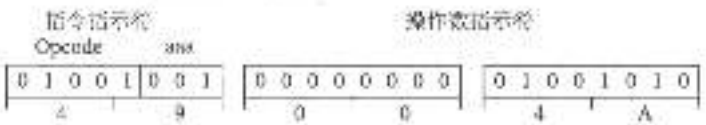


图 4-27　字符输入指令

假定输入流的下一个字符是 W，图 4-28 展示了执行字符输入指令后的结果。输入流的字符可以来自键盘或者文件。字母 W 的 ASCII 值 57（hex）发送到地址 004A 的内存单元。

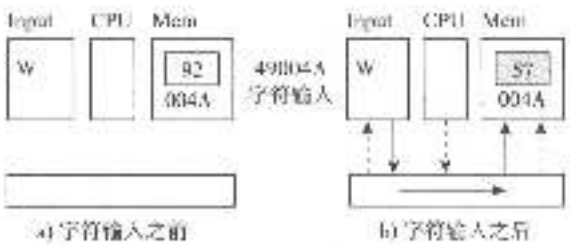


图 4-28　字符输入指令的执行

图中显示 CPU 中没有被指令影响的寄存器。然而，CPU 是计算机系统的一部分，通过它发送到总线的控制信号控制计算机的其他部分。从 CPU 到输入设备的虚线表示控制信号，它指示输入设备把输入流的下一个字符放到总线上。从 CPU 到内存系统的控制信号指示内存子系统把总线上的数据存入内存单元。控制信号包括内存系统存储数据的地址信息。 □

例 4.15　假定地址 004A 内存单元的内容是 48（hex），图 4-29 展示了执行字符输

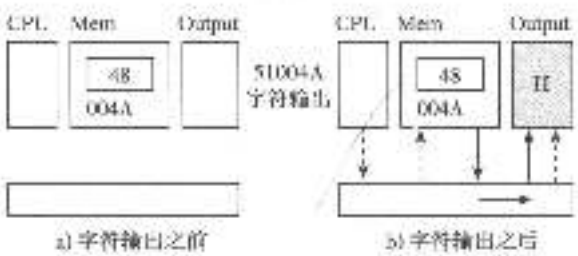


图 4-29　字符输出指令的执行

出指令后的结果。CPU 给内存系统发送控制信号，让它把内存单元 004A 的数据发送到总线上；给输出设备发送控制信号，让它从总线上获取数据，把它当作 ASCII 字符并在输出设备上输出。48 (hex) 对应的 ASCII 字符是字母 H。 □

4.3 冯·诺依曼机器

在最早期的电子计算机中，每个程序都是手工接线的。要改变程序，线路必须要手工重连，这是单调又耗时的过程。3.1 节中描述的 ENIAC 计算机就是这种计算机，它的内存仅仅用来存储数据。

1945 年，约翰·冯·诺依曼在宾夕法尼亚大学的一份报告中提议美国军械部建造一台在主存中可以存储数据也可以存储程序的计算机。那时，存储程序是一个相当激进的理念。1949 年，Maurice V. Wilkes 在剑桥大学建造了电子延迟存储自动计算器（EDSAC）。这是用冯·诺依曼的存储程序理念建造的第一台计算机。实际上，今天所有的商用计算机都是基于存储程序概念。程序和数据共享主存。尽管有人认为 J. Presper Eckert 在冯·诺依曼的论文发表的前几年就提出这个概念，但是这种计算机还是被称作冯·诺依曼计算机。

4.3.1 冯·诺依曼执行周期

Pep/8 计算机是一个典型的冯·诺依曼计算机。图 4-30 是执行一个程序所需步骤的伪代码描述：

这个 do 循环称作冯·诺依曼执行周期，一个周期包括 5 个操作

- 取指
- 译码
- 增加 PC
- 执行
- 重复

```
从机器语言程序
初始化PC和SP
do {
  取下一条指令
  指令指示符解码
  递增PC
  执行取出的指令
}
while (没有执行停止指令)
```

图 4-30 在 Pep/8 计算机上执行程序所需步骤的伪代码描述

冯·诺依曼周期固化在中央处理单元中，下面是执行过程中详细步骤的描述。

为了把机器语言程序装入主存，第一条指令要放在地址 0000 (hex)。第二条指令放在与第一条相邻的地址。如果第一条指令是一元的，那么第二条指令的地址是 0001；否则第一条指令的操作数指示符会放在地址 0001 和 0002 的字节，那么第二条指令的地址将是 0003。类似地，第三条指令放在与第二条相邻的地址，整个机器语言程序就是这样装载到内存的。

为了初始化程序计数器和栈指针，PC 置为 0000 (hex)，SP 置为 Mem[FFF8]。程序计数器的目的是保存下一条要执行指令的地址。因为第一条指令装入内存的地址是 0000，所以 PC 必须要初始化为 0000。栈指针的目的是保存运行时栈顶部的地址。后面将解释 SP 为何设置为 Mem[FFF8]。

冯·诺依曼执行周期的第一个操作是取指令。要获取一条指令，CPU 检测 PC 中的 16 位，把它当作一个地址，到主存的这个地址获取下一条指令的指令指示符（1 字节）。把指令指示符的 8 位带入 CPU，放到指令寄存器（IR）的第一个字节。

冯·诺依曼执行周期的第二个操作是译码。CPU 从指令指示符中提取操作码，确定要执行哪条指令。根据操作码，如果有的话，就提取寄存器指示符和寻址字段。CPU 根据操

作码就能知道指令是不是一元的，如果不是一元指令，那么就从内存获取操作数指示符（一个字），把它存储在 IR 的最后两个字节。

冯·诺依曼执行周期的第三个操作是增加 PC。如果指令是一元指令，CPU 对 PC 加 1，否则加 0003。不管 PC 加那个数，相加后的值都是下一条指令的地址，因为在主存中指令是相邻的。

167～169

冯·诺依曼执行周期的第四个操作是执行。CPU 执行存储在 IR 中的指令，操作码告诉 CPU 执行 39 条指令中的哪一条。

冯·诺依曼执行周期的第五个操作是重复。除非刚刚执行的指令是停止指令，否则 CPU 返回到取指令操作。如果指令试图执行一个非法操作，Pep/8 也将在此时终止。有些指令不允许使用特定的寻址方式。使 Pep/8 终止的最常见的非法操作是试图以禁止的寻址方式执行指令。

图 4-31 是在 Pep/8 计算机上执行一个程序所需步骤的详细伪代码描述。

```
把机器语言程序加载到地址0000开始的内存
PC←0000
SP←Mem[FFFB]
do {
    取PC内容为地址处的指令指示符
    PC←PC+1
    指令指示符解码
    if（指令不是一元的）
        取PC内容为地址处的指令指示符
        PC←PC+2
    }
    执行取出的指令
}
while（没有执行停止指令&&指令是合法的）
```

图 4-31　在 Pep/8 计算机上执行程序所需步骤的更详细伪代码描述

4.3.2　一个字符输出程序

Pep/8 系统可以从键盘输入，输出到屏幕。这些 I/O 设备都是基于 ASCII 字符集的。当你按下一个键，代表 1 个 ASCII 字符的 1 个字节的信息从键盘沿着总线传送到主存。当 CPU 沿着总线把 1 个字节发送到屏幕时，屏幕把这个字节当作一个 ASCII 字符，显示该字符。

在 ISA3 层，机器层，计算机通常只有字节类型的输入和输出指令。对字节的解释发生在输入和输出设备，而不在主存中。Pep/8 唯一的输入指令是把 1 个字节从输入设备传送到主存，唯一的输出指令把 1 字节从主存传送到输出设备。因为通常会把这些字节解读成 ASCII 字符，所以 Pep/8 系统在 ISA3 层的 I/O 称作字符 I/O。

170

图 4-32 展示了一个简单的机器语言程序，它在输出设备上输出字符 H1。它使用了两条指令：字符输出指令 0101 0 和停止指令 0000 0000。第一段代码展示的是二进制机器语言程序，主存存储这些 1 和 0 的序列。第一列是每行上位模式第一个字节的十六进制地址。

第二段代码展示了同一个程序的十六进制简写。尽管这个格式稍微易读一点儿，但是你要谨记内存存储的是位，而不是代码中那样的字面上的十六进制字符。第二段代码的每行都有注释，用分号与机器语言隔开。注释不会随程序装入内存。

```
地址        机器语言（bin）
0000        0101 0001 0000 0000 0000 0111
0003        0101 0001 0000 0000 0000 1000
0006        0000 0000
0007        0100 1000
0008        0110 1001

地址        机器语言（hex）
0000        510007 ;Character output
0003        510008 ;Character output
0006        00     ;Stop
0007        48     ;ASCII H character
0008        69     ;ASCII I character

输出
H1
```

图 4-32　输出字符 H1 的机器语言程序

图 4-33 展示了计算机执行这个程序的每一步。图 4-33a 是 Pep/8 计算机的初始状态，图中没有显示输入设备，也省略了几个程序用不到的

CPU 寄存器，每个问号表示 4 位。初始时，主存单元和 CPU 寄存器的内容是未知的

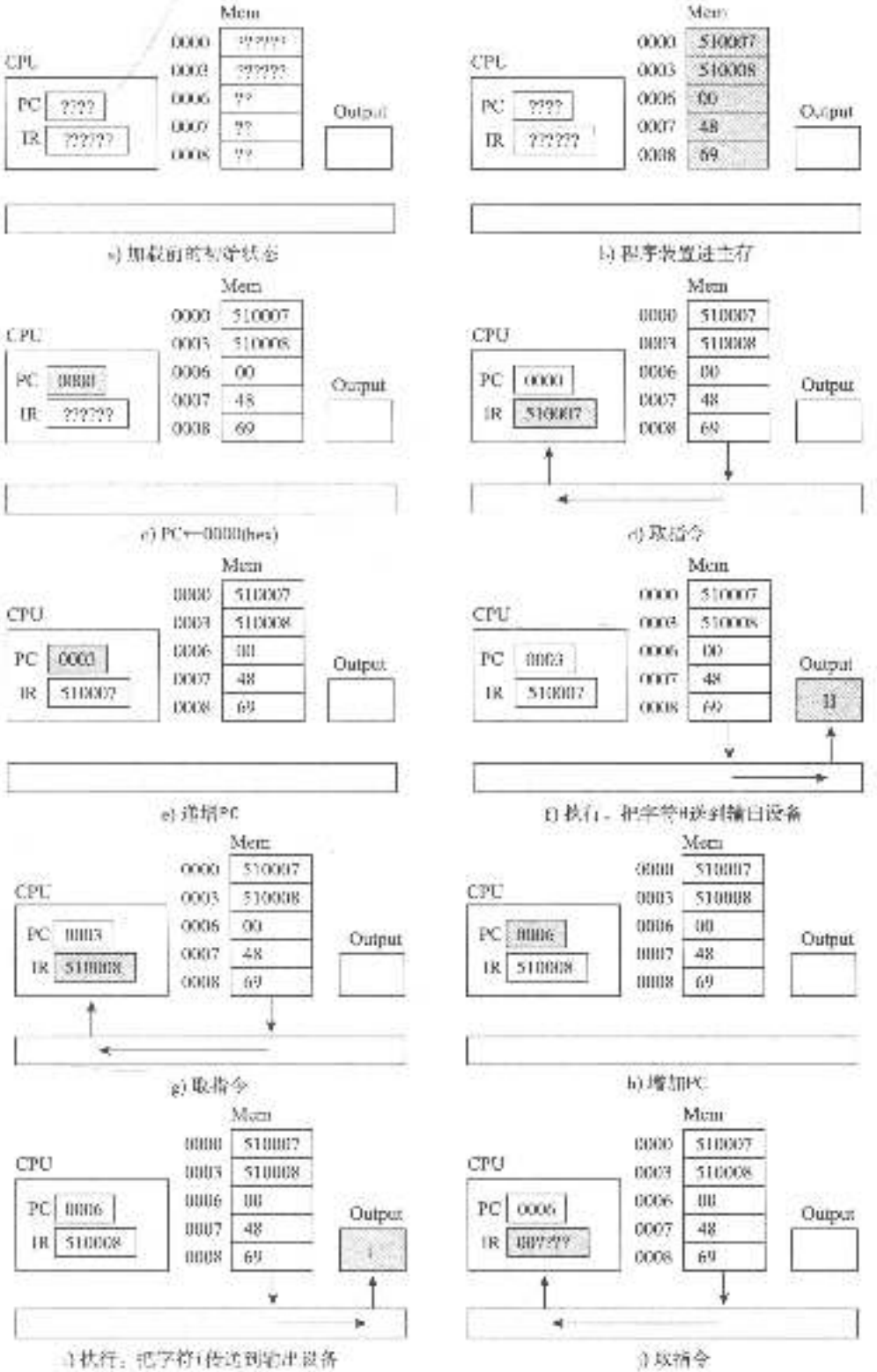


图 4-33 图 4-32 所示程序的冯·诺依曼执行周期

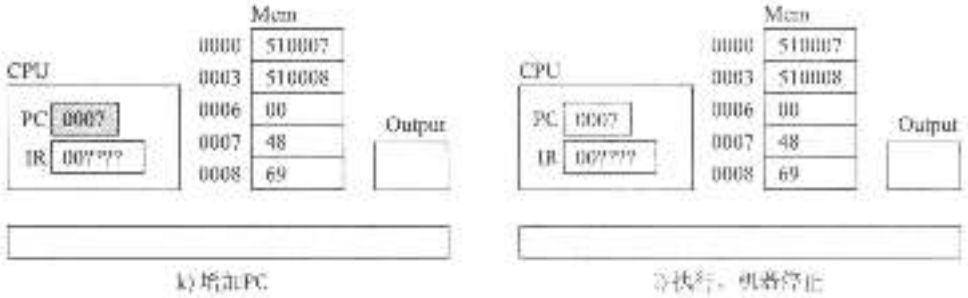


图 4-33（续）

图 4-33b 是过程的第一步。程序装入主存，起始地址是 0000。程序来自哪里和什么把它放入的内存，这些细节将在后面的章节描述。

图 4-33c 是过程的第二步。程序计数器清空置为 0000（hex），图中没有显示对 SP 的初始化，因为这个程序不会用到栈指针。

图 4-33d 是执行周期的取指令部分。CPU 检测到 PC 中的位为 0000（hex），接着它给主存发信号让主存把这个地址的字节发送到 CPU。当 CPU 获得这个字节时，把它填入指令寄存器的第一部分。接着 CPU 对指令指示符解码，根据操作码确定指令不是一元指令，于是把操作数指示符也读入 IR。取指令不会改变地址 0000、0001 和 0002 的内容，主存只是把这 24 位的内容发送到 CPU。

图 4-33e 显示执行周期的增加部分，CPU 给 PC 加上 0003。

图 4-33f 显示执行周期的执行部分，CPU 检测到 IR 的前 5 位为 0101 0，此操作码给电路发信号，执行字符输出指令。

根据 IR 的内容，CPU 检测到寻址方式位是 001，这表示直接寻址。接着 CPU 检测操作数指示符，其内容为 0007（hex），于是给主存发送控制信号，直接走到地址 0007，把这个地址的字节放到总线上。同时 CPU 给输出设备发送控制信号让它从总线获取这个字节。输出设备把这个字节解释为 ASCII 字符并显示它。如果寻址方式不是直接寻址，那么 CPU 不会给主存发信号直接去地址 0007 获取字节。

171
|
173

图 4-33g 显示执行周期的取指令部分。这次 CPU 检测到 PC 里的内容为 0003（hex），于是它取出地址 0003 处的一个字节的内容，发现指令不是一元指令，接着获取在地址 0004 处的一个字，这样做的结果是改变了 IR 的原始内容。

图 4-33h 是执行周期的 PC 增加部分，CPU 给 PC 加 0003 得到 0006（hex）。

图 4-33i 显示执行周期的执行部分。与图 4-33f 一样，CPU 发现操作码是字符输出指令，寻址方式位表明是直接寻址。但是，这次操作数指示符是 0008（hex），地址 0008 处的字节是 69（hex），也就是 0110 1001（bin）。因为最右 7 位是 110 1001，所以输出设备显示 ASCII 字符 i。

图 4-33j 显示执行周期的取指部分，因为 PC 包含 0006（hex），所以这个地址的字节来到 CPU。这次当 CPU 检测操作码时，发现指令是一元指令，因此 CPU 不会获取地址 0007 的字。

图 4-33k 显示执行周期的 PC 增加部分，CPU 给 PC 加 0001 得到 0007。

图 4-33l 显示执行周期的执行部分。这次 CPU 发现 IR 中操作码是停止指令，因此它忽略寻址方式而只是停止执行周期。

仅仅输出两个字符看上去都是有点儿复杂的过程，但对于人类来说它执行得相当快，许

多计算机的执行周期取指部分只需大约不到1纳秒的时间。因为执行周期的执行部分取决于指令的类型，因此执行复杂的指令可能需要很多纳秒，而执行简单的指令只需要几个纳秒的时间。

计算机不会给它电路中的电子信号附加任何含义。具体来说，主存不知道一个特定地址的位是代表数据还是指令，它只知道是1和0。

4.3.3 冯·诺依曼漏洞

在图4-32的程序中，地址0000到0006的位被CPU作为指令，0007到0008的位作为数据。因为程序员知道PC的初始值是0000，在每一个执行周期的循环会被加0001或0003。所以他把指令位放在开头。如果犯错遗漏了停止指令（操作码0000 0000），尽管程序员的本意是想要把下面的字节当作数据，但执行周期将会继续获取下面的字节，把它当作下一条指令的指令指示符来解读。 174

因为程序和数据共享内存，所以机器层程序员在为它们分配内存时必须要小心仔细，否则会出现两种类型的问题：一种是CPU可能会把程序员想作为数据的位序列当作指令，另一种是CPU把程序员想作为指令的位序列当作数据。这两种类型的漏洞都发生在机器层。

虽然如果程序员不仔细，程序和数据共享内存会产生漏洞，但是这也带来了令人激动的潜在价值。程序只是存储在内存中的一组指令，因此程序员可以把一个程序看作另一个程序的数据。这使得写一个用于处理另一个程序的程序成为可能。编译程序、汇编程序和装载器采用的都是把其他程序当作数据这一观点。

4.3.4 一个字符输入程序

图4-34的程序从输入设备输入两个字符，以逆序在输出设备输出。使用直接寻址方式的字符输入指令从输入设备获得字符。

第一条指令49000D的操作码指明是字符输入指令，寻址方式位指明是直接寻址。它把来自输入设备的第一个字符放入位于Mem[000D]的字节。虽然这个字节在代码中没有显示，但是它显然是有效地址，因为内存地址一直到FFFF。第二条指令49000E也是字符输入指令，但是把字符放入到Mem[000E]。

第三条指令51000E的操作码指明是字符输出指令，要输出存储在Mem[000E]处的字节。第四条指令51000D输出存储在Mem[000D]的字节。

```
地址        机器语言（bin）
0000        0100 1001 0000 0000 0000 1101
0003        0100 1001 0000 0000 0000 1110
0006        0101 0001 0000 0000 0000 1110
0009        0101 0001 0000 0000 0000 1101
000C        0000 0000
地址        机器语言（hex）
0000        49000D  ;Character input
0003        49000E  ;Character input
0006        51000E  ;Character output
0009        51000D  ;Character output
000C        00      ;Stop
输入
up
输出
pu
```

图4-34 一个输入2个字符并按照逆序输出的机器语言程序

4.3.5 十进制转换为ASCII

图4-35展示了一个程序，它把两个个位数字相加，输出它们一位数的和。这个程序说明在机器层处理输出的不便性。

两个加数是5和3，程序把它们存储在Mem[0011]和Mem[0013]。第一条指令把5装入累加器，接着第二条指令把累加器加3，此时和在累加器里。

现在问题出现了。我们想输出这个结果，但是ISA3层唯一的输出指令是字符输出指令。问题是结果为0000 1000（bin），如果字符输出指令输出它，会把它当作退格键BS，如图3-24的ASCII表所示。

因此，程序必须把十进制数8，即0000 1000（bin），转换为ASCII字符8，即0011 1000(bin)。ASCII位与无符号二进制位不同，它在第三和第四位有两个1。为了实现这个转换，程序用寄存器OR指令将累加器和掩码0000 0000 0011 0000进行或运算，在结果中插入两个额外的1：

```
          0000 0000 0000 1000
OR        0000 0000 0011 0000
          ---------------------
          0000 0000 0011 1000
```

现在累加器包含ASCII格式的正确的和，存储字节指令把这个字符存储在Mem[0010]，然后字符输出指令输出它。

如果把Mem[0013]的字替换为0009，那么程序会输出什么？尽管累加器中的和在执行加法累加指令后是

```
地址        机器语言（bin）
0000        1100 0001 0000 0000 0001 0001
0003        0111 0001 0000 0000 0001 0011
0006        1010 0001 0000 0000 0001 0101
0009        1111 0001 0000 0000 0001 0000
000C        0101 0001 0000 0000 0001 0000
0001        0000 0000
0010        0000 0000
0011        0000 0000 0000 0101
0013        0000 0000 0000 0011
0015        0000 0000 0011 0000

地址        机器语言（hex）
0000        C10011   ;A := first number
0003        710013   ;Add the two numbers
0006        A10015   ;Convert sum to character
0009        F10010   ;Store the character
000C        510010   ;Output the character
000F        00       ;Stop
0010        00       ;Character to output
0011        0005     ;Decimal 5
0013        0003     ;Decimal 3
0015        0030     ;Mask for ASCII char
输出
8
```

图4-35　计算5加上3并把结果当作一个字符输出的机器语言程序

14（dec）= 0000 0000 0000 1110（bin）

但遗憾的是，它不会输出14。OR指令会把这个位模式变为0000 0000 0011 1110（bin），产生的输出是>。因为ISA3层唯一的输出指令只能输出单个字节，所以程序不能输出包含大于1个字符的结果。在第5章中，我们将看到如何补救这个缺点。

4.3.6　一个修改自身的程序

175 / 177

图4-36说明了基于冯·诺依曼设计原理的一个奇特的可能性。我们注意到这个程序从0006～001B和图4-35的从0000～0015是一样的，然而这个程序在开始有两个图4-35中没有的指令。因为指令往下移了6个字节，所以它们的操作数指示符也比前面程序的大了6。除了有6个字节的调整外，从0006开始的指令会重复图4-35的处理过程。

```
地址        机器语言（bin）
0000        1101 0001 0000 0000 0001 1101
0003        1111 0001 0000 0000 0000 1001
0006        1100 0001 0000 0000 0001 0111
0009        0111 0001 0000 0000 0001 1001
000C        1010 0001 0000 0000 0001 1011
000F        1111 0001 0000 0000 0001 0110
0012        0101 0001 0000 0000 0001 0110
```

图4-36　一个修改自身的机器语言程序，累加器指令被改为了减法指令

```
0015        0000 0000
0016        0000 0000
0017        0000 0000 0000 0101
0019        0000 0000 0000 0011
001B        0000 0000 0011 0000
001D        1C00 0001

地址        对象代码（hex）
0000        010010  ;Load byte accumulator
0003        F10039  ;Store byte accumulator
0006        C10017  ;A := first number
0009        710019  ;Add the two numbers
000C        A10018  ;Convert sum to character
000F        F10016  ;Store the character
0012        510016  ;Output the character
0015        00      ;Stop
0016        00      ;Character to output
0017        0005    ;Decimal 5
0019        0003    ;Decimal 3
001B        0030    ;Mask for ASCII char
001D        01      ;Byte to modify instruction

输出
8
```

图 4-36 （续）

尤其是，装入累加器指令把 5 装入累加器，加法指令对累加器加 3，OR 指令把 8（dec） 178
变为 ASCII 的 8，存储字节累加器指令把 8 放入 Mem[0016]，而字符输出指令输出 8。但是，输出却是 2。

因为在冯·诺依曼计算机中程序和数据共享同一个内存，所以程序有可能把它自己当作数据并进行修改。第一条指令把字节 81（hex）装入累加器的右半部分，接着第二条指令把它放入 Mem[0009]。在这个修改之前 Mem[0009] 的内容是什么？加法累加器指令的指令指示符。现在 Mem[0009] 的位是 1000 0001。当计算机在冯·诺依曼执行周期的取指部分得到这些位时，CPU 检测到操作码为 1000，是减法寄存器指令，寄存器指示符表示是累加器，寻址方式位显示是直接寻址。指令从 5 减去 3 而不是加 3。

当然，这并不是一个非常实际的程序。如果想对两个数进行减法运算，可以简单地用减法指令代替加法指令。写图 4-35 的程序就可以了。但是它确实表明在冯·诺依曼计算机中，主存不会赋予它正在存储的位任何含义，它只知道 1 和 0，并不知道哪些是程序位、哪些是数据位、哪些是 ASCII 字符等。此外，CPU 遵循冯·诺依曼执行周期，相应地解释位，并不了解它们的历史。当获取在 Mem[0009] 的位时，它不知道也不在意这些位是怎么样到这里的，它只是简单地一再重复取指、译码、增加 PC 和执行。

4.4 ISA3 层编程

ISA3 层编程就是写二进制指令。要执行二进制序列，首先要把它装入主存。操作系统负责把二进制序列装入主存。

操作系统就是一个程序。与任何其他程序一样，软件工程师必须设计、编写、测试

和调试操作系统。大多数操作系统非常庞大和复杂，必须由一个工程师团队来编写。操作系统的主要功能是控制应用程序在计算机上的执行。因为操作系统本身也是一个程序，所以要执行操作系统，它也必须在主存中。因此在主存中不仅要存储应用程序，也要存储操作系统。



图 4-37　Pep/8 操作系统在主存中的位置

在 Pep/8 计算机中，主存的底部保留给操作系统，顶部保留给应用程序。图 4-37 展示了操作系统在主存中的位置，它占用从 FBCF 开始的主存，剩下的从 0000 到 FBCE 用于应用程序。

操作系统的装载器（loader）把应用程序装入主存，这样它才能执行。那么什么来装入装载器呢？Pep/8 的装载器和操作系统的许多其他部分是永久存储在主存中的。

约翰·冯·诺依曼

约翰·冯·诺依曼是一个伟大的数学家、物理学家、逻辑学家和计算机科学家。有关于冯·诺依曼非凡解题速度和惊人记忆力的故事一直在流传。他的天才不仅用于深化他的数学理论，还用于记忆整本书，在读过数年之后仍能够背诵出来。但是如果你问高速公路巡警，他准会绝望地举起双手：在开车时，这位数学天才就跟一个叛逆的青少年一样鲁莽。



1903 年，约翰·冯·诺依曼生于匈牙利，他是一个富裕犹太银行家的长子。他 11 岁时进入高中，不久后他的数学老师建议由大学教授来给他授课。他年仅 19 岁时，他就发表了他的第一篇论文，被公认为杰出的数学家。

在第二次世界大战爆发之前，冯·诺依曼离开德国前往美国。战争期间，由于流体力学方面的知识，冯·诺依曼受雇于美国军方和相关的民用机构作为顾问。1943 年，他还被委派参与原子弹的建设。因为这份工作，艾森豪威尔总统在 1955 年任命他加入原子能委员会。

1944 年一次偶然的机会他遇到 Herbert Goldstine，第一代可运行电子数字计算机的先锋，冯·诺依曼开始接触计算机。他在火车站与 Goldstine 偶然的谈话点燃了他新的迷恋。他开始研究存储程序的概念，最终实现了在计算机内部存储程序，消除了在那个时代对计算机重新编程所需要的冗长的劳动。他还开发出了新的计算机体系结构，基于现在有名的冯·诺依曼周期来执行存储的任务。从计算机发明最初到现在，改变的主要是速度和基本电路的组成，而冯·诺依曼设计出的基本体系结构保持至今。

在冯·诺依曼的一生中，他在很多知名机构任教，包括柏林、汉堡和普林斯顿大学。在普林斯顿时，他与天才的但当时还不甚有名的英国学生 Alan Turing 一起工作。他还获得了很多奖项，包括普林斯顿、哈佛和伊斯坦布尔大学的荣誉博士。1957 年，冯·诺依曼由于骨癌在华盛顿特区逝世，时年 54 岁。

"在你都不知道自己在说什么的时候，精确是没有意义的。"

——约翰·冯·诺依曼

4.4.1 只读内存

这里有两种制造内存设备的电子电路元件——读/写电路元件和只读电路元件。

在图 4-36 的程序中，当执行存储字节指令 F10016 时，CPU 把累加器右半部分的内容传送到到 Mem[0016]，Mem[0016] 的原始内容被破坏，现在内存单元内容为 0011 0010(bin)。当接下来执行字符输出指令时，内存单元 0016 的位就被传送到了输出设备。

内存单元 0016 的电路元件是读/写电路。存储指令对它进行了写操作，使它的内容发生改变。字符输出指令对它进行了一个读操作，把它内容的副本发送到输出设备。如果 0016 的电路元件是只读电路，那么存储指令不会改变它的内容。

相对于串行设备，主存电路元件的两种类型——读/写和只读，都是随机存取设备。当字符输出指令对内存单元 0016 进行读操作时，它不需要从 0000 开始依序经过 0001、0002、0003 等直到 0016，它直接找到位置 0016。因为它能直接访问主存的任意位置，所以这种电路元件叫作随机存取设备。

只读内存设备称为 ROM。读/写内存设备应该称为 RWM，遗憾的是，它们称为 RAM，表示随机存取内存。很遗憾叫这个名字，因为只读和读/写设备都是随机存取设备。区别只读内存设备和读/写内存设备的特性是只读内存设备的内容不能被存储指令改变。由于在计算机工业中 RAM 这个词的使用是如此的普遍，所以我们也用它来指代读/写内存设备，但是在心里，我们知道 ROM 也是随机存取的。

主存中通常包含一些 ROM 设备。这些部分是包含永久二进制序列的 ROM，存储指令是不能改变的。此外，每天结束时关机和每天开始时开机，这些二进制序列都保存在 ROM 的电路中。如果关机，RAM 不会保留它的记忆，因此它又称为易失的（volatile）。

计算机厂商给内存系统购买 ROM 有两种方法。它可以向电路厂商指定内存设备中希望的位序列，然后电路厂商相应地生产设备；或者它订购可编程只读内存（PROM）。这是一种全 0 的 ROM，计算机厂商可以把任意希望的位置永久地变为 1。用这样的方法，设备将包含适当的位序列。这个过程叫作“烧入”位模式。

179 / 181

4.4.2 Pep/8 操作系统

Pep/8 操作系统的大部分都已经烧入 ROM。图 4-38 展示了操作系统的 ROM 部分，它从 FC57 开始一直到 FFFF，这部分主存是永久不变的，存储指令不能改变它。如果掉电，又上电，操作系统的这个部分仍然在这里。从 FBCF 到 FC56 的区域是计算机操作系统的 RAM 部分。

操作系统的 RAM 部分用来存储系统变量。当操作系统程序执行时，它们的值会改变。操作系统的 ROM 部分包含装载器，它是永久固定的。它的工作是把应用程序装入从地址 0000 开始的 RAM。在 Pep/8 计算机上，通过从模拟器程序菜单中选择装载器选项来调用装载器。

图 4-39 是 Pep/8 系统一个更详细的内存图。与图 4-38 一样，阴影区域代表操作系统的范围，空白区域代表应用程序的范围。

应用程序的运行时栈叫作用户栈，从内存单元 FBCF 开始，正好在操作系统的上面。CPU 中的栈指针

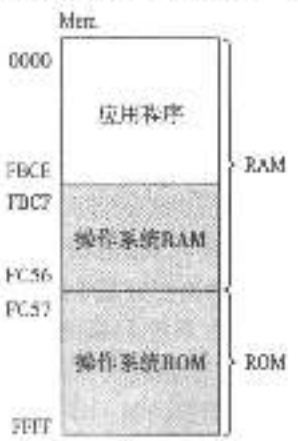


图 4-38 Pep/8 系统中的只读存储器

寄存器存放栈顶的地址。当程序被调用时，会在栈上分配参数、返回地址和局部变量的存储空间，从较低的地址开始连续分配，因此栈在内存中是“向上生长的”。

操作系统的运行时栈从内存单元 FC4F 开始，它位于用户栈起始往下 128 字节的位置。当操作系统执行时，CPU 中的栈指针包含系统栈顶的地址。与用户栈一样，系统栈在内存中也是向上生长的。操作系统的栈绝不会超过 128 字节，因此系统栈不可能试图把它的数据存储在用户栈的范围内。

Pep/8 操作系统由两个程序组成——开始于地址 FC57 的装载器和开始于地址 FC9B 的陷阱处理程序。看图 4-6 你会记得，在 ISA3 层操作码从 0010 01 到 0100 0 的指令是未实现的，陷阱处理程序把这 3 条指令实现给汇编语言程序员。第 5 章会讲述 Asmb5 汇编层的这些指令，第 8 章将展示在 OS4 操作系统层是如何实现它们的。

与操作系统这两个部分相关的是 ROM 最底部的 4 个字，它们被操作系统保留为特殊用途，称为机器向量，地址分别是 FFF8、FFFA、FFFC 和 FFFE，如图 4-39 所示。

182



图 4-39　Pep/8 系统的内存图

当从 Pep/8 模拟器菜单选择装入选项时，发生了下列两个事件：

SP ← Mem[FFFA]

PC ← Mem[FFFC]

换句话说，内存单元 FFFA 的内容被复制到栈指针，内存单元 FFFC 的内容被复制到程序计数器。这两个事件发生后，执行周期开始。图 4-40 展示了这两个事件。

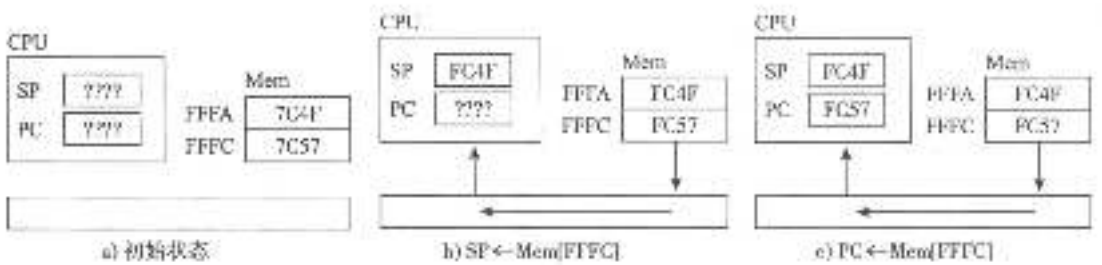


图 4-40　Pep/8 加载选项

实际上，选择装入选项会把栈指针和程序计数器初始化为存储在 FFFA 和 FFFC 中的预定值。地址 FFFA 的值正好是系统栈的底部 FC4F，也就是系统栈为空时栈指针的值。而地址 FFFC 的值正好是 FC57，它是装入器中要执行的第一条指令的地址。

写操作系统的系统程序员决定系统栈和装载器应该在什么位置。因为当选择装入选项时，Pep/8 计算机会从 FFFA 和 FFFC 获取向量，系统程序员会在这些位置放置适当的值。由于执行周期的第一步是取指令，所以选择装入选项后第一条要执行的指令就是装载器的第一条指令。

如果想修改操作系统，装载器不从 FC57 开始，假定从 7BD6 开始，当用户选择装入选项时，计算机仍将去 FFFC 获取向量，因此需要把 7BD6 放在地址 FFFC 处。

这种在特殊保留内存单元中存储地址的方案具有很好的灵活性，它允许系统程序员把装载器放在内存中任何方便的位置。一个更直接但是缺乏灵活性的方案是把计算机设计成当用户选择装入选项时，执行下列操作：

SP ← FC4F

PC ← FC57

如果选择装入选项会产生这两个事件，那么当前操作系统的装载器仍然能正确工作。但是修改操作系统会很困难。装载器不得不总是从 FC57 开始，系统栈会不得不总是从 FC4F 开始，系统程序员不能选择系统各个部分的放置位置。

4.4.3　使用 Pep/8 系统

为了在 Pep/8 计算机上装入一个机器语言程序，所幸的是，不一定非要用二进制来编写这个机器语言程序，可以在文本文件中用 ASCII 十六进制字符进行编写。当装载器装入这个程序时会把它从 ASCII 转换到二进制。 183

图 4-41 中的代码展示了怎样准备一个机器语言程序以便装入。这是图 4-32 中的程序，输出 Hi。可以在文本文件中编写十六进制形式的二进制序列，不需要任何地址或注释。以小写的 zz 结束字节列表，装载器会把 zz 作为一个标记符号。装载器将把这些字节逐个送入内存中从 0000（hex）开始的地址中。

```
地址        机器语言（hex）
0000        510007 ;Character output
0003        510008 ;Character output
0006        00     ;Stop
0007        48     ;ASCII H character
0008        69     ;ASCII i character

从装载器十六进制版本

51 00 07 51 00 08 00 48 69 zz

输出

Hi
```

图 4-41　为加载器准备程序

Pep/8 装载器对机器语言程序的格式是非常挑剔的。为了正确地工作，文本文件中的第一个字符必须是十六进制字符，开头不允许有空行或空格，字节之间必须有且只能有一个空格。如果字节流要从新的一行开始，上一行的结尾一定不能有空格。

在编写完机器语言程序并用装载器选项把它装入之后，必须选择执行选项去运行它。当选择执行选项时，发生下面两个事件：

SP ← Mem[FFF8]

PC ← 0000

然后冯·诺依曼执行周期开始。因为 PC 的值是 0000，所以 CPU 将从 Mem[0000] 处获取第一条指令。装载器刚好把应用程序的第一条指令放在这里。

图 4-39 表明 Mem[FFF8] 的内容是 FBCF，这是用户栈的底部。本例中的应用程序不使用运行时栈。如果应用程序要用到运行时栈，因为 SP 被初始化为用户栈底部的地址，所以程序能够正确地存取栈。

好好享受你现在学到的知识吧！ 184

总结

几乎所有的商用计算机都是基于冯·诺依曼设计原理的，这个原理中主存既存储数据也

存储指令。冯·诺依曼机器的4个组成部分是输入设备、中央处理单元（CPU）、主存和输出设备。CPU包含一组寄存器，其中一个寄存器是程序计数器（PC），它存储下一条要执行指令的地址。

CPU有一组固化在其中的指令集。一条指令由指令指示符和操作数指示符组成。指令指示符依次由操作码，可能有的寄存器字段和寻址方式字段组成。操作码用来确定要执行指令集中的哪条指令，寄存器字段用来确定哪个寄存器参与运算，寻址方式字段用来确定源或目的数据使用哪种寻址方式。

每种寻址方式对应于一种操作数指示符（OprndSpec）和操作数（Oprnd）之间的关系。对于直接寻址方式，操作数指示符是主存中操作数的地址，数学表示为：Oprnd = Mem[OprndSpec]。

要执行一个程序，需要把一组指令和数据装入主存，然后冯·诺依曼执行周期开始。冯·诺依曼执行周期由下列步骤组成：1）获取PC指定的指令；2）对指令指示符译码；3）增加PC；4）执行取出的指令；5）返回第一步重复进行。

由于主存既存储指令也存储数据，所以在机器层有可能出现两种类型的问题。可以把数据位解释为指令，或者可以把指令位解释成数据。此外还有一种可能性，把指令存储在主存中的直接后果是可以像处理数据一样处理程序。装载器和编译程序是使用把指令位当作数据这一观点的重要程序。

操作系统是控制应用程序执行的一个程序，它必须与应用程序和数据一起位于主存中。在某些计算机中，操作系统的一部分烧入只读内存（ROM）中。ROM的一个特性是存储指令改变不了内存单元的内容。操作系统的运行时栈位于随机存储内存（RAM）中。机器向量是操作系统组成部分的地址，例如栈或者程序，它用于存取这个组成部分。装载器和陷阱处理程序是操作系统的两个重要功能单元。

练习

4.1节

*1.（a）Pep/8计算机的主存有多少字节？（b）有多少个字？（c）有多少位？（d）Pep/8的CPU中总共有多少位？（e）以位来度量，主存比CPU大多少倍？

2.（a）假定Pep/8的主存全都是一元指令，那么包含多少一元指令？（b）如果所有指令都不是一元指令，能容纳多少指令？（c）假定主存中全是一元和非一元指令，它们的数量相同，那么总共能容纳多少指令？

185

*3. 回答有关机器语言指令7AF82C和D623D0的下列问题。（a）二进制表示的操作码是什么？（b）指令是做什么的？（c）二进制表示的寄存器-r字段是什么？（d）它指定的是哪一个寄存器？（e）二进制表示的寻址方式字段是什么？（f）它指定的是哪种寻址方式？（g）十六进制表示的操作数指示符是什么？

4. 对于机器语言指令8B00AC和F70BD3，回答练习3中的问题。

4.2节

*5. 假定Pep/8包含下列4个十六进制值：

A: 19AC

X: FE20

Mem[0A3F]: FF00

Mem[0A41]: 103D

如果下列每条语句执行前是这 4 个值，那么每条语句执行后 4 个十六进制值是什么？

(a) C10A3F　(b) D10A3F　(c) D90A41　(d) F10A41　(e) E90A3F
(f) 890A41　(g) 810A3F　(h) A10A3F　(i) 19

6. 对下列语句重复练习 5：

(a) C90A3F　(b) D90A3F　(c) F10A41　(d) E10A41　(e) 790A3F
(f) 810A41　(g) 990A3F　(h) A90A3F　(i) 18

4.3 节

*7. 确定下列 Pep/8 机器语言程序的输出，左边一列是本行第一个字节的内存地址：

```
0000 51000A
0003 51000B
0006 51000C
0009 00
000A 406F
000C 79
```

8. 如果输入是 tab，确定下列 Pep/8 机器语言程序的输出，左边一列是本行第一个字节的内存地址：

```
0000 490010
0003 490011
0006 490012
0009 510011
000C 510010
000F 00
```

186

9. 确定下列 Pep/8 机器语言程序的输出，每部分左边一列是本行第一个字节的内存地址：

*(a)

```
0000 C1000E
0003 910010
0006 F1000D
0009 51000D
000C 00
000D 00
000E A94F
0010 FFFD
```

(b)

```
0000 C1000C
0003 18
0004 F10008
0007 510008
000A 00
000B 00
000C F004
```

4.4 节

10. 假定需要处理 Pep/8 内存中的 31 000 个整数，每个整数占用一个字。一个典型的程序中估计有 20% 的指令是一元指令，那么在这个处理数据的程序中最多能有多少条指令？要记住应用程序是与操作系统和数据共享内存的。

11. (a) 你正在用的计算机是哪家公司制造的？(b) 它的主存是多少字节？(c) 它的 CPU 中有多少寄存器？每个寄存器是多少位的？(d) 一条指令包含多少位？(e) 指令中有多少位留做操作码使用？

问题

4.4 节

12. 写一个在输出设备上输出你名字的机器语言程序，它的格式要适合 Pep/8 模拟器的装载器并能够在上面执行。

13. 写一个在输出设备上输出 4 个字符 Frog 的机器语言程序，它的格式要适合 Pep/8 模拟器的装载器并能够在上面执行。

14. 写一个在输出设备上输出 3 个字符 Cat 的机器语言程序，它的格式要适合 Pep/8 模拟器的装载器并能够在上面执行。

15. 写一个把 3 个数 2、-3 和 6 相加的机器语言程序，在输出设备上输出和，它的格式要适合 Pep/8 模

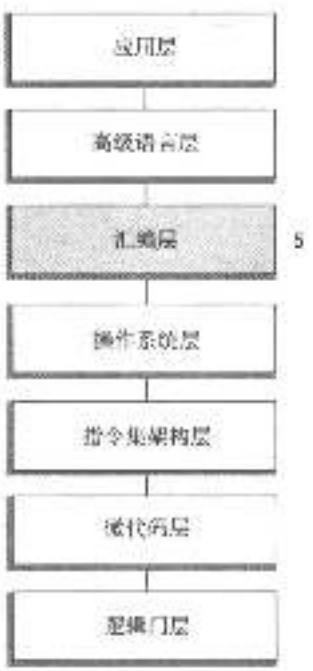
拟器的装载器并能够在上面执行。以十六进制存储 -3。不要使用减法、取反或反转指令。

16. 写一个输入 2 个 1 位数，把它们相加，并输出 1 个 1 位数和，它的格式要适合 Pep/8 模拟器的装载器并能够在上面执行。

17. 以十六进制格式编写图 4-35 的程序用来输入到装载器。在 Pep/8 模拟器上运行它，验证它可以正确地工作。然后修改字节存储指令和字符输出指令，把结果存储在 Mem[FCF5]，再从 Mem[FCF5] 输出字符，输出是什么？请解释。

187
188

汇编层（第5层）



第 5 章
Computer Systems, Fourth Edition

汇编语言

ISA3 层的语言是机器语言，是 1 和 0 的序列，有时简写为十六进制的格式。计算机的先驱不得不使用机器语言编程，但是很快他们就开始厌恶这种很土的方式。要记住机器的操作码，不得不频繁地查 ASCII 表和十六进制表，这些都毫无趣味。发明汇编层就是为了让程序员能免于二进制编程的单调乏味。

第 4 章讲述 ISA3 层，也就是机器层的 Pep/8 计算机。本章将讲述 Asmb5 层，也就是汇编层的 Pep/8。这两层之间是操作系统。记住抽象分层的目的是隐藏系统在更低层次的细节。你可以使用操作系统的陷阱处理程序而不需知道它的操作细节，即你将了解到陷阱处理程序做什么而不需知道它是怎样做的。我们在第 8 章讲述陷阱处理程序的内部工作原理。

5.1 汇编程序

Asmb5 层的语言叫作汇编语言，它提供了一种比二进制更加方便的编写机器语言程序的方法。图 4-32 的程序输出 H1，包含两类位模式，一种是程序，一种是数据。冯·诺依曼设计原理直接导致这两种类型的产生，因为程序和数据共享内存，每种都需要一种二进制表示。

189 ⋮ 191

汇编语言包含两种类型的语句，分别对应这两种类型的位模式。助记符语句对应指令位模式，伪操作对应数据位模式。

5.1.1 指令助记符

假定内存中某个位置有机器语言指令

```
C0009A
```

它是装入寄存器 r 指令，寄存器 -r 位为 0，表示是累加器而不是变址寄存器，寻址 -aaa 字段是 000，表明是立即数寻址。

这条指令用 Pep/8 汇编语言来写是

```
LDA 0x009A, i
```

助记符 LDA 表示装入累加器，用以代替操作码 1100 和寄存器 -r 字段 0。助记符是辅助记忆的工具。记住 LDA 代表装入累加器指令要比记住操作码 1100 和寄存器 -r 为 0 代表装载累加器指令更为容易。操作数指示符以十六进制形式书写，为 009A，前面加 0x，表示是十六进制常址。在 Pep/8 汇编语言中，通过在操作数指示符后放一个或多个字母来指定寻址方式，字母和操作数指示符之间以逗号分隔。图 5-1 展示了与

aaa	寻址方式	字　符
000	立即数	i
001	直接	d
010	间接	n
011	栈相对	s
100	栈相对间接	sf
101	变址	x
110	栈变址	sx
111	栈变址间接	sxf

图 5-1　Pep/8 汇编语言中表示寻址方式的字符

8种寻址方式相对应的字母。

例5.1 这里有一些例子，是用二进制机器语言和汇编语言编写的装入寄存器r指令。LDX对应和LDA一样的机器语言语句，除了对LDX来说，寄存器-r位为1而不是0。

```
1100 0011 0000 0000 1001 1010      LDA 0x009A,i
1100 0111 0000 0000 1001 1010      LDA 0x009A,sx
1100 1011 0000 0000 1001 1010      LDX 0x009A,i
1100 1111 0000 0000 1001 1010      LDX 0x009A,sx
```

图5-2总结了在Asmb5层Pep/8指令集的39条指令。它给出了每个操作码对应的助记符以及指令的含义。寻址方式列说明允许哪些寻址方式或者指令是否是一元指令（U）。状态位列说明指令执行会影响的状态位。 [192]

指令指示符	助记符	指令	寻址方式	状态位
0000 0000	STOP	停止执行	U	
0000 0001	RETTR	从陷阱返回	U	
0000 0010	MOVSPA	把SP移到A	U	
0000 0011	MOVFLGA	把NZVC标志移到A	U	
0000 010a	BR	无条件分支	i,x	
0000 011a	BRLE	小于等于分支	i,x	
0000 100a	BRLT	小于分支	i,x	
0000 101a	BREQ	等于分支	i,x	
0000 110a	BRNE	不等于分支	i,x	
0000 111a	BRGE	大于等于分支	i,x	
0001 000a	BRGT	大于分支	i,x	
0001 001a	BRV	如果V为1，分支	i,x	
0001 010a	BRC	如果C为1，分支	i,x	
0001 011a	CALL	调用子例程	i,x	
0001 100r	NOTr	按位反转r	U	NZ
0001 101r	NEGr	对r取反	U	NZV
0001 110r	ASLr	算术左移r	U	NZVC
0001 111r	ASRr	算术右移r	U	NZC
0010 000r	ROLr	循环左移r	U	C
0010 001r	RORr	循环右移r	U	C
0010 01nn	NOPn	n个字节的非操作	U	
0010 1aaa	NOP	非一元空操作指令	i	
0011 0aaa	DECI	十进制输入陷阱	d, n, s, sf, x, sx, sxf	NZV
0011 1aaa	DECO	十进制输出陷阱	i, d, n, s, sf, x, sx, sxf	
0100 0aaa	STRO	字符串输出陷阱	d, n, sf	
0100 1aaa	CHARI	字符输入	d, n, s, sf, x, sx, sxf	
0101 0aaa	CHARO	字符输出	i, d, n, s, sf, x, sx, sxf	
0101 1nnn	RETn	从调用返回n个本地字节	U	
0110 0aaa	ADDSP	加到栈指针（SP）上	i, d, n, s, sf, x, sx, sxf	NZVC
0110 1aaa	SUBSP	从栈指针（SP）减去	i, d, n, s, sf, x, sx, sxf	NZVC
0111 raaa	ADDr	加到r上	i, d, n, s, sf, x, sx, sxf	NZVC
1000 raaa	SUBr	从r减去	i, d, n, s, sf, x, sx, sxf	NZVC
1001 raaa	ANDr	与r按位AND	i, d, n, s, sf, x, sx, sxf	NZ
1010 raaa	ORr	与r按位OR	i, d, n, s, sf, x, sx, sxf	NZ
1011 raaa	CPr	与r比较	i, d, n, s, sf, x, sx, sxf	NZVC

图5-2 Pep/8的Asmb5层指令集

指令指示符	助记符	指令	寻址方式	状态位
1100 raaa	LDr	从内存加载到 r	i, d, n, s, sf, x, sx, sxf	NZ
1101 raaa	LDBYTEr	从内存加载字节	i, d, n, s, sf, x, sx, sxf	NZ
1110 raaa	STr	把 r 存储到内存	d, n, s, sf, x, sx, sxf	
1111 raaa	STBYTEr	把字节 r 存储到内存	d, n, s, sf, x, sx, sxf	

图 5-2（续）

图 5-2 还给出了 5 条没有对应机器指令的新指令：

NOPn　　一元空操作陷阱
NOP　　非一元空操作陷阱
DECI　　十进制输入陷阱
DECO　　十进制输出陷阱
STRO　　字符串输出陷阱

这些新指令对 Asmb5 层的汇编语言程序员是可用的，但不是 ISA3 层指令集的一部分。OS4 层的操作系统会向它们提供陷阱处理程序。在汇编层，你可以用它们来编程，仿佛它们就是 ISA3 层指令集的一部分，尽管实际上它们并不是。第 8 章会详细介绍操作系统是怎样提供这些指令的。不过用它们进行编程，并不需要知道实现它们的细节。

5.1.2　伪操作

伪操作（pseudo-ops）是汇编语言语句，它没有操作码，不对应 Pep/8 指令集 39 条指令中的任何一条。Pep/8 汇编语言有 8 个伪操作：

.ADDRSS　符号的地址
.ASCII　ASCII 字节字符串
.BLOCK　字节块
.BURN　初始 ROM 烧入
.BYTE　一个字节值
.END　汇编器标记
.EQUATE　将一个符号等同于一个常量值
.WORD　一个字值

193 / 194

除了 .BURN、.END 和 .EQUATE 外，所有的伪操作都把数据位插入机器语言程序中。“伪”的意思是“假”，称它们为伪操作是因为它们产生的位不对应操作码。不像那 39 个指令助记符产生的位那样，它们不是真正的指令操作。伪操作也叫作汇编器指示字（assembler directive），或者叫作点命令（dot command），因为汇编语言中这些指令前都有个点（.）。

接下来的 3 个程序展示怎样使用 .ASCII、.BLOCK、.BYTE、.END 和 .WORD 伪操作，其他伪操作后面讲述。

5.1.3　.ASCII 和 .END 伪操作

图 5-3 是用汇编语言而不是机器语言写的图 4-32 中的程序。与 C++ 不同，Pep/8 汇编语言是面向行的，即每条汇编语言语句必须包含在一行内，不能把一条语句持续到下一行，也不能把两条语句放在同一行中。

注释以分号 ; 开始，一直到本行末端。在一行中只有注释是允许的，但它必须以分号开

始。这个程序的前4行都是注释行，CHARO指令也包含注释，不过是在汇编语句的后面。和C++一样，汇编语言程序中至少应该包含名字、日期和程序的描述。不过本书为了节约篇幅，下面的程序不包含这些程序头。

```
汇编语言输入
;Stan Warford
;January 13, 2009
;A program to output "Hi"
;
CHARO   0x0007,d    ;Output 'H'
CHARO   0x0008,d    ;Output 'i'
STOP
.ASCII  "Hi"
.END

汇编语言输出
51 00 07 51 00 08 00 48 69 zz

程序输出
Hi
```

图5-3 一个汇编语言程序，输出Hi，它是图4-32的汇编语言版

CHARO是字符输出指令的助记符。语句

```
CHARO   0x0007,d    ;Output 'H'
```

表示"用点接寻址方式从Mem[0007]输出一个字符"。

.ASCII伪操作生成连续的ASCII字符字节。在汇编语言中，可以简单地写.ASCII，然后在后面跟双引号括起来的一个字符串。若字符串中包含双引号，那么必须在它前面加一个反斜杠\。若包含反斜杠，就要在它的前面再加一个反斜杠。在n前面加反斜杠可以在字符串中插入一个换行符，在t前面加反斜杠将在字符串中插入一个制表符。

例5.2 这是一个包含两个双引号的字符串：

```
"She said, \"Hello\"."
```

这是一个包含反斜杠符的字符串：

```
"My bash is \\."
```

这是一个有换行符的字符串：

```
"\nThis sentence will output on a new line."
```

□

使用\x特性，字符串常量中可以包含任何一个字节。当字符串常量中包含\x时，汇编程序会预期接下来的两个字符是十六进制数字，指定了你想包含在字符串中的字节。

例5.3 点命令

```
.ASCII "Hello\nworld."
```

和

```
.ASCII "Hello\x0Aworld\x2E"
```

生成同样的字节序列，即

```
48 65 6C 6C 6F 0A 77 6F 72 6C 64 2E
```

□

汇编语言程序必须以.END命令结束。它不会像.ASCII命令那样在程序中插入数据位，它仅表示汇编语言程序的结束。汇编器用.END作为标记符号，以便知道何时应该终止转换。

5.1.4 汇编器

用汇编语言编写的程序与用机器语言编写的相同程序相比，由于在操作码的位置使用了助记符，所以汇编语言更容易理解，用ASCII字符点接写的字符H和i也更易读。

195 ~ 196

遗憾的是，不能简单地用汇编语言写一个程序，就指望计算机可以理解它。计算机只能通过执行冯·诺依曼运行周期（取指、译码、增加PC、执行、重复）来执行程序，这是固化在CPU中的。如第4章所述，为了执行周期能正确地处理程序，它必须以二进制形式存储在从地址0000开始的主存中。因此在装入和执行前，汇编语言语句必须以某种方式被翻译成机器语言。

在早期，程序员用汇编语言写程序，然后手工把每条语句翻译成机器语言。这个翻译其实很简单，它只是查询指令的二进制操作码和在 ASCII 表中查询 ASCII 字符的二进制编码。类似地，用十六进制转换表把十六进制操作数转换为二进制格式。只有在翻译后，程序才可以装入并执行。

翻译长程序是一项无聊又单调的工作。很快程序员就意识到可以写一个计算机程序来做这个翻译工作。这样的程序叫作汇编器。图 5-4 说明了它的主要功能。

汇编器是这样的一个程序，它输入汇编语言程序，输出把这个程序翻译成适合装载器格式的机器语言。汇编器的输入称为源程序，输出称为目标程序。图 5-5 展示了 Pep/8 汇编器处理图 5-3 汇编语言的效果。

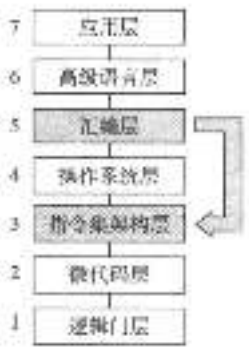


图 5-4　汇编器的功能



图 5-5　Pep/8 汇编器对图 5-3 所示程序所做的操作

认识到汇编器只是把程序翻译成适合装载器的格式是很重要的。它并不执行程序。翻译和执行是两个分离的过程，总是先进行翻译。

因为汇编器本身是一个程序，所以必须用某种编程语言来编写。写出第一批汇编器的计算机先驱不得不使用机器语言来编写。否则，如果他们用汇编语言来写，由于那时还没有汇编器可用，他们就不得不手工再翻译成机器语言。重点是机器只能执行用机器语言写的程序。

5.1.5　.BLOCK 伪操作

图 5-6 是图 4-34 中程序的汇编语言版本。它输入两个字符，按照相反的顺序输出它们。

从汇编程序的输出可以看到第一条输入语句 CHAR1　0x000D，d 翻译成了 49000D，最后一条输出语句 CHAR0　0x000D，d 翻译成了 51000D，再后面的 STOP 语句翻译成 00。

.BLOCK 伪操作生成接下来的两个 0 字节。点命令

```
.BLOCK 1
```

意思是“生成一个 1 字节存储块”。汇编程

汇编器输入

```
CHAR1   0x000D,d   ;Input first character
CHAR1   0x000E,d   ;Input second character
CHAR0   0x000E,d   ;Output second character
CHAR0   0x000D,d   ;Output first character
STOP
.BLOCK  1          ;Storage for first char
.BLOCK  1          ;Storage for second char
.END
```

汇编器输出

```
49 00 0D 49 00 0E 51 00 0E 51 00 0D 00 00 00 00 zz
```

程序输入

```
op
```

程序输出

```
pu
```

图 5-6　一个输入两个字符并以相反顺序输出它们的汇编语言程序。这是图 4-34 所示程序的汇编语言版本

字把任何不以 0x 开头的数字当作十进制常数。因此数字 1 被当作十进制常数。汇编程序使用 .BLOCK 后面是一个常量，然后产生这个数量字节的存储空间，并把它们置为 0。在这个程序中，可以用一条

```
.BLOCK 2
```

代替那两条 .BLOCK 命令，它的意思是“生成一个 2 字节存储块”。尽管汇编器的输出是一样的，但是在汇编语言程序中就不能在 .BLOCK 行上写两条单独的注释。

5.1.6 .WORD 和 .BYTE 伪操作

197 / 198

图 5-7 和图 4-35 一样，计算 5 加上 3，它说明 .WORD 伪操作的用法。

和 .BLOCK 命令一样，.WORD 命令也是为装载器生成代码，但是有两点不同。首先，.WORD 命令总是生成一个字（2 字节）的代码，不能生成任意数量的字节；其次，程序员能指定字的内容。点命令

```
.WORD 5
```

的意思是“生成一个值为 5（dec）的字”。点命令

```
.WORD 0x0030
```

的意思是“生成一个值为 0030（hex）的字”。

```
汇编器输入
LDA     0x0011,d    ;A <- first number
ADDA    0x0013,d    ;Add the two numbers
ORA     0x0015,d    ;Convert sum to character
STBYTEA 0x0017,d    ;Store the character
CHARO   0x0017,d    ;Output the character
STOP
.BLOCK  1           ;Character to output
.WORD   5           ;Decimal 5
.WORD   3           ;Decimal 3
.WORD   0x0030      ;Mask for ASCII char
.END
汇编器输出
C1 00 11 71 00 13 A1 00 15 F1 00 17 51 00 17 00
00 00 05 00 03 00 30 zz
程序输出
8
```

图 5-7 一个把 3 加上 5 并输出单个字符结果的汇编语言程序。它是图 4-35 所示程序的汇编语言版本

.BYTE 命令和 .WORD 命令的工作方式一样，除了它生成 1 字节的值而不是生成一个字的值外。在这个程序中，可以把

```
.WORD 0x0030
```

替换为

```
.BYTE 0x00
.BYTE 0x30
```

它会生成同样的机器语言。

可以把这个汇编语言程序经过汇编器的输出和图 4-35 的十六进制机器语言进行比较，你会发现它们是一样的。把汇编器设计成让它产生的输出完全遵循装载器期望的格式，没有前导空行或空格，字节间只能有 1 个空格，每一行的结尾都没有空格。字节序列以 zz 结束。 199

5.1.7 使用 Pep/8 汇编器

执行图 5-6 中的程序，这个程序输出两个输入字符的应用程序要求在计算机上运行图 5-8 所示的步骤。

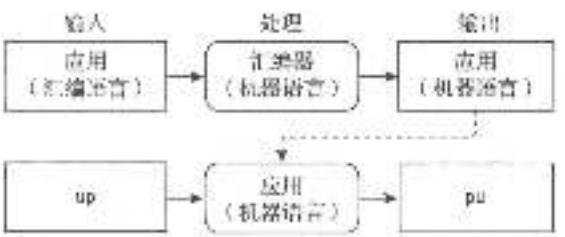


图 5-8 执行图 5-6 所示程序需要计算机运行两条命令

首先把汇编器装入主存，把应用程序作为输入文件。此运行的输出是这个应用程序的机器语言版本，接着第二次运行就把这个输出装入主存。中间两个框里的所

有程序必须是机器语言写的。

Pep/8 系统除了有汇编器之外，还有一个模拟器。当执行汇编器时，必须提供给它一个之前用文本编辑生成的汇编语言程序。如果你的程序没有错误，那么汇编器将生成适合装载器格式的目标代码，否则它将给出一条或多条出错信息并说明未生成代码。从一个没有错误的程序生成代码后，可以用第4章讲述的模拟器来使用它。

```
汇编器输入

chari 0x000D,d    ;Input first character
CHARI   0x000E,d    ;Input second character
charo 0x000E,          d    ;Output second character
CHarO   0x000D,D    ;Output first character
      stop
.block  1 ;Storage for first char
.BLOCK  1          ;Storage for second char
            .END

汇编器列表

Addr  Code   Mnemon  Operand    Comment
0000  49000D CHARI   0x000D,d   ;Input first character
0003  49000E CHARI   0x000E,d   ;Input second character
0006  51000E CHARO   0x000E,d   ;Output second character
0009  51000D CHARO   0x000D,d   ;Output first character
000C  00     STOP
000D  00     .BLOCK  1          ;Storage for first char
000E  00     .BLOCK  1          ;Storage for second char
000F         .END
```

图 5-9　一个合法的源程序和输出的汇编器列表

写汇编语言程序时，助记符或者点命令后面至少要有一个空格。除此之外，对空格没有其他限制。源程序可以大小写字母混用，例如图 5-6 中的源程序可以写为图 5-9 中那样，汇编程序也会认为它是有效的，接受并生成正确的代码。

除了为装载器生成目标代码外，汇编程序允许生成程序列表。汇编器列表把源程序转换成大小写字母和空格一致的格式。图 5-9 中展示的是无格式源程序的汇编器列表。

这个列表也展示了每一行生成的十六进制目标代码，以及装载器会把它装入的第一个字节的地址。注意 .END 命令不会生成任何目标代码。

本书接下来的汇编语言程序都是以汇编器列表的形式给出的，不过不包括这张图里有的、汇编器生成的列头部。第二列是机器语言目标代码，第一列是装载器会把它装入主存中的地址。这是大多数汇编器使用的典型布局。这形象地展示了 ISA3 层机器语言和 Asmb5 层汇编语言之间的对应关系。

5.1.8　交叉汇编器

一个厂商生产的机器通常具有和另一个厂商生产的机器不一样的指令集，因此，一种品牌计算机的机器语言程序不能在另一种品牌的机器上运行。

如果用汇编语言给一种个人计算机写应用程序，通常要在这种计算机上进行汇编。用该汇编器所要转换成的语言编写的汇编器叫作常驻汇编器（resident assembler）。这种汇编器和应用程序运行在同样的机器上，图 5-8 中的汇编器和应用程序就是运行在同一台机器上的。

200 ~ 201

不过也有可能用 X 品牌机器的机器语言写成的汇编程序把应用程序翻译成另一种 Y 品牌机器的机器语言。那么这个应用程序不能在翻译它的机器上执行，必须先把它从 X 品牌机器移到 Y 品牌机器上。

交叉汇编器生成的目标程序所适用的机器，不同于运行汇编器的机器。把应用程序的机器语言版本从 X 品牌机器的输出文件移到 Y 品牌机器的主存，这个过程称作下载（downloading），X 品牌机器叫作宿主机，Y 品牌机器叫作目标机。在图 5-8 中，第一个命令

运行是在宿主机上，第二个命令运行是在目标机上。

这种情况经常发生在当目标机器是某种小型特殊用途的计算机时，比如控制微波炉烹饪周期的计算机。汇编器是需要大量主存、以及输入/输出外围设备的程序。控制微波炉处理器的主存非常小，它的输入只是控制面板上的按键，也许还有来自于温度探测器的信号，它的输出包括数字显示和控制烹饪元件的信号。因为它没有输入/输出文件，所以它不能为自己运行汇编器。必须有一个更大宿主机为它把程序汇编成目标语言，然后它再从宿主机上下载该目标语言程序。

5.2 立即数寻址和陷阱指令

在直接寻址方式中，操作数指示符是操作数在主存中的地址。数学表达是

Oprnd = Mem[OprndSpec]

但是在立即数寻址方式中，操作数指示符就是操作数：

Oprnd = OprndSpec

采用直接寻址方式的指令包含操作数的地址，而采用立即数寻址方式的指令包含操作数本身。

5.2.1 立即数寻址

图 5-10 展示了怎样用立即数寻址方式来写图 5-3 中的程序。这个程序输出 Hi。

汇编程序把字符输出指令

```
CHARO 'H',i
```

翻译成目标代码 500048（hex），即二进制的

```
0101 0000 0000 0000 0100 1000
```

查找图 5-2 会发现 0101 0 是 CHARO 指令的操作码；寻址 -aaa 字段是 000（bin），表示是立即数寻址。如图 5-1 所示，,i 表示立即数寻址。

```
0000  500048 CHARO   'H',i       ;Output 'H'
0003  500069 CHARO   'i',i       ;Output 'i'
0006  00     STOP
0007         .END
输出
Hi
```

图 5-10 一个采用立即数寻址方式的输出 Hi 的程序

字符常量用单引号扩起来，它们总是生成 1 字节的代码。在图 5-10 的程序中，字符常量放在操作数指示符中，操作数指示符占用 2 字节。这种情况下，字符常量位于这 2 字节中右边的那一个。

汇编器就是这样把语句翻译成二进制的。但是当装载器把程序装入并执行第一条语句时会发生什么呢？如果寻址方式是直接寻址，那么 CPU 会把 0048 作为地址，指示主存把 Mem[0048] 放到总线上供输出设备使用。由于寻址方式是立即数寻址，所以 CPU 会把 0048 当作操作数本身（而不是操作数的地址），并把 48 放到总线上供输出设备使用。第二条指令对 0069 进行类似的操作。

与直接寻址相比，立即数寻址有两个优点。因为 ASCII 字符串和指令不需要分开存储，所以程序可以更短。图 5-3 中的程序是 9 字节，而这个程序是 7 字节。因为操作数在指令寄存器中，对 CPU 来说是立即可用的，所以指令执行也更快。若用直接寻址方式，CPU 必须额外访问主存以获取操作数。

5.2.2 DECI、DECO 和 BR 指令

截至目前我们已经学到的汇编语言特色相比于机器语言有了很大的改进，但是仍有几个

不尽如人意的地方。图 5-11 中的程序说明了这些地方，这个程序输入一个十进制的值，把它加 1，然后输出和。

```
0000  040005 BR      0x0005      ;Branch around data
0003  0000   .BLOCK  2           ;Storage for one integer
              ;
0005  310003 DECI    0x0003,d    ;Get the number
0008  390003 DECO    0x0003,d    ;and output it
000B  00002C CHARO   ' ',i       ;Output " + 1 = "
000E  00002B CHARO   '+',i
0011  000020 CHARO   ' ',i
0014  000031 CHARO   '1',i
0017  000020 CHARO   ' ',i
001A  00003D CHARO   '=',i
001D  000020 CHARO   ' ',i
0020  C10003 LDA     0x0003,d    ;A <- the number
0023  700001 ADDA    1,i         ;Add one to it
0025  E10003 STA     0x0003,d    ;Store the sum
0029  390003 DECO    0x0003,d    ;Output the sum
002C  00     STOP
002D         .END
```

输入

```
-479
```

输出

```
-479 + 1 =  -478
```

图 5-11　一个输入十进制值，加 1 并输出和的程序

图 5-7 的第一条指令

```
LDA 0x0011,d ;A <- the number
```

把 Mem[0011] 的内容放入累加器。要写这条语句，程序员必须知道第一个数字要存储在程序指令部分的后面、地址为 0011（hex）的地方。但把数据放到程序末尾的问题是，要到写完程序你才能知道程序指令部分的确切长度，而当写这条需要数据地址的指令时，还不知道数据的地址。

另一个问题是修改程序。假设想在程序中插入一条语句，这样的一个修改将会改变数据的地址，从而也要修改每条引用这些数据的指令以反映新的地址。如果把数据放在程序的上面，对于 Asmb5 层程序来说会更简单。此时，当你写一条引用数据的语句时，就已经知道数据的地址了。

图 5-7 所示程序的另一个不尽如人意的地方是，由于 CHARO 的限制导致对结果只能是单字符的限制。因为 CHARO 只能输出 1 字节的 ASCII 字符，所以很难对 ASCII 表示的超过一个数字的十进制值执行 I/O 操作。

202 ~ 204

图 5-11 中的程序解决了这两个问题。它输入一个整数，加 1，然后输出和。数据存储在程序开始的部分，并允许使用大的十进制值。

在 Pep/8 模拟器选择执行选项时，PC 获得值 0000（hex）。CPU 会把在 Mem[0000] 处的字节作为第一条指令来执行。为了把数据放在程序的上部，我们需要一条指令，当 CPU 获取下一条指令时，这条指令会让 CPU 跳过数据字节。无条件分支指令 BR 就是这样的指令，它只是把指令的操作数放入 PC 中。在这个程序里，

```
BR 0005 ;Branch around data
```

把 0005 放入 PC。BR 指令的 RTL 描述是

```
PC ← Oprnd
```

在下一个执行周期的取指部分，CPU 会从地址 0005 而不是地址 0003 获取指令，如果没有修改 PC，就会去 0003 处取指令。

因为分支指令几乎总是使用立即数寻址方式，所以 Pep/8 汇编器不要求指定寻址方式。如果对转移指令不指定寻址方式，那么汇编器就假设是立即数寻址，并为寻址 -a 字段生成 0。

BR 指令的正确操作取决于冯·诺依曼执行周期的细节。例如，你可能会好奇，为什么这个周期是取指、译码、增加 PC、执行和重复，而不是取指、译码、执行、PC 增加和重复。图 4-33f 展示了当 PC 值为 0003 时，执行指令 510007 输出 H，0003 是指令 51000R 的地址。如果冯·诺依曼执行周期的执行部分在 PC 增加部分的前面，那么当在地址 0000 的指令 510007 执行时，PC 的值为 0000。貌似 PC 对应于正在执行的指令更讲得通，但为何冯·诺依曼执行周期不把执行放在 PC 增加的前面呢？

这是因为如果那样，BR 就无法正确工作了。在图 5-11 中，PC 获得 0000，CPU 就会获取 BR 指令 040005。BR 执行，把 0005 放入 PC 中，接着 PC 将增加到 0008。此时，程序将转移到 0008 而不是 0005。因为指令集包含分支指令，所以冯·诺依曼执行周期的 PC 增加一定要在执行部分之前。

DECI 和 DECO 是两条操作系统提供给汇编层的指令，Pep/8 硬件不在机器层提供这两条指令。DECI 代表十进制输入，它把一个 ASCII 数字字符序列转换为一个字，对应于该数值的补码表示。DECO，即十进制输出，做相反的操作，把一个字长的补码值转换为 ASCII 字符序列。

DECI 允许在输入中有任何数量的前导空格和空行。第一个可打印的字符必须是十进制数字，+ 或者 -，接下来的数字必须是十进制数字。如果输入为 0，DECI 把 Z 置为 1，如果 [205]
输入为负值，它把 N 置为 1。如果输入值超出范围，它把 V 置为 1。因为一个字是 16 位，$2^{16} = 32768$，所以范围是 -32768 到 32767（dec）。DECI 不影响 C 位。

如果值为负，DECO 输出 -，但是如果值为正，不输出 +。它不输出最前面的 0，只输出能正确表示该数值的尽可能少的字符。不能指定字段的宽度。DECO 不影响 NZVC 位。

在图 5-11 中，当面对输入序列 -479 时，语句

```
DECI 0x0003,d ;Get the number
```

把它转换为 1111 1110 0010 0001（bin），存储在 Mem[0003]。DECO 把二进制序列转换为 ASCII 字符串输出。

5.2.3 STRO 指令

你可能已经注意到了，图 5-11 中的程序需要 7 个 CHARO 指令来输出字符串 "+1="，每个输出的 ASCII 字符需要一个 CHARO 指令。图 5-12 中的程序说明了 STRO 指令，指令名字的含义是字符串输出。这又是一条在机器层触发陷阱的汇编层真实指令，它让你只用一条指令就能输出完整的 7 个字符的字符串。

STRO 的操作数是一个连续的字节序列，序列中的每个字节都被当作一个 ASCII 字符。序列的最后一个字节必须是全 0，STRO 把它作为一个标记符号。这条指令从头开始输出字节字符串，直到但是不包括标记符号。在图 5-12 中，伪操作

```
.ASCII "+1=\x00"
```

```
0000  040000 BR      0x000D     ;Branch around data
0003  0000   .BLOCK  2          ;Storage for one integer
0005  202820 .ASCII  " + 1 = \x00"
      312030
      2000
            ;
0008  310003 DECI    0x0003,d   ;Get the number
0010  390003 DECO    0x0003,d   ;and output it
0013  410005 STRO    0x0005,d   ;Output " + 1 = "
0016  C10003 LDA     0x0003,d   ;A <- the number
0019  700001 ADDA    1,i        ;Add one to it
001C  F10003 STA     0x0003,d   ;Store the sum
001F  390003 DECO    0x0003,d   ;Output the sum
0022  00     STOP
0023         .END
输入
-479
输出
-479 + 1 = -478
```

图 5-12　与图 5-11 一样的程序，不过使用的是 STRO 指令

用 \x00 生成这个标记符号字节。这个伪操作生成包括标记符号在内的 8 字节，但是 STRO 指令只输出 7 字节。在计算 BR 指令的操作数时，必须把所有 8 字节都计算在内。

汇编器列表只在目标代码栏分配了 3 字节的空间，如果 .ASCII 伪操作中的字符串生成的字节数大于 3，那么汇编器列表会在后续的行继续目标代码。

5.2.4　解释位模式

第 4 章和第 5 章从低的抽象层次（ISA3）讲到高的抽象层次（Asmb5）。虽然 Asmb5 层的汇编语言隐藏了机器语言的细节，但细节仍然存在，尤其是机器最终还是基于取指、译码、增加 PC、执行、重复这个冯·诺依曼执行周期。用伪操作和助记符来产生数据位和指令位不会改变机器的属性。当指令执行时，它只是执行位，并不知道汇编器怎样生成这些位。图 5-13 展示了一个无实际意义的程序，它的唯一目的是用来说明这种情况。它用一种伪操作来产生数据位，而这些位会被指令以一种意想不到的方式来解释。

在这个程序中，用

```
.WORD 0xFFFE ;First
```

生成了 First 的十六进制值，但是被

```
DECO 0x0003,d ;Interpret First as decimal
```

解释成一个十进制数，并输出 -2。当然，如果程序员想要把位模式 FFFE 解释为十进制数，他也许会写这样的伪操作

```
.WORD -2 ;First
```

这个伪操作生成同样的目标代码，而且目标程序与原始程序相同。当 DECO 执行时，它不知道在翻译时位是怎样生成的，它只知道在执行时它们是什么。

十进制输出指令

```
DECO 0x0005,d ;Interpret Second and Third as decimal
```

```
0000  040009 BR      0x0009      ;Branch around data
0003  FFFE   .WORD   0xFFFE      ;First
0005  00     .BYTE   0x00        ;Second
0006  55     .BYTE   'U'         ;Third
0007  0470   .WORD   1136        ;Fourth
              ;
0009  390003 DECO    0x0003,d    ;Interpret first as decimal
000C  500004 CHARO   '\n',i
000F  390005 DECO    0x0005,d    ;Interpret Second and Third as decimal
0012  500004 CHARO   '\n',i
0015  510006 CHARO   0x0006,d    ;Interpret Third as character
0018  510008 CHARO   0x0008,d    ;Interpret Fourth as character
001B  00     STOP
001C         .END

输出
-2
85
Up
```

图 5-13 滥用位模式解释方式的无实际意义程序

把位于地址 0005 的位解释为一个十进制数，输出 85。DECO 总是输出两个连续字节的十进制值。这种情况下，字节 0055（hex）= 85（dec）。实际情况是这 2 字节是由两个不同的 .TYPE 点命令生成的，一个是从十六进制常量 0x00 生成的，而另一个是从字符常量 'U' 生成的，而这些都没有关系。在执行时，唯一重要的是位是什么，而不是它们来自哪里。

字符输出指令

```
CHARO 0x0006,d ;Interpret Third as character
```

把地址 0006 的位解释为一个字符。这一点也不奇怪，因为这些位是由 .BYTE 点命令用一个字符常量生成的。和预期的一样，输出了字母 U。

最后一个输出指令

```
CHARO 0x0008,d ;Interpret Fourth as character
```

输出字母 p。为什么呢？因为存储单元 0008 的位是 70（hex），它是 ASCII 字符 p 对应的位。这些位是从哪里来的？它们是

```
.WORD 1136 ;Fourth
```

生成位的后半部分。之所以这样是因为 1136（dec）－0470（hex），而这个位模式的第二个字节是 70（hex）。

206 ↓ 208

在所有这些例子中，指令只是经过冯·诺依曼执行周期。我们必须牢记翻译过程不同于执行过程。翻译在执行之前进行。翻译后，当指令执行时，位的来源就不重要了。唯一重要的是位是什么，而不是翻译阶段它们是从哪里来的。

5.2.5 反汇编器

汇编器把每条汇编语言语句都翻译成一条机器语言语句，这样的转换叫作一对一映射（one-to-one mapping）。一条汇编语言语句映射到一条机器语言语句。这和编译程序不同，稍后我们会看到编译程序的一对多映射。

给你一条汇编语言语句，总是可以确定它对应的机器语言语句。反过来呢？给你一个机

器语言程序的位序列，你能确定这条机器语言来自于哪条汇编语言语句吗？

答案是：不，你不能。尽管汇编语言到机器语言的转换是一对一的，但逆向转换却不是唯一的。对于二进制机器语言序列

```
0101 0111
```

不知道汇编语言程序员原来使用的是字符 W 的汇编器指示字，还是使用栈间接相对寻址方式的 CHARO 助记符。不管源程序中是这两条汇编语言语句中的哪一个，汇编器都会生成完全一样的位序列。

此外，在执行时，正存不知道原始的汇编语句是什么，它只知道 CPU 通过执行周期处理的 1 和 0。

图 5-14 给出了生成相同机器语言的两个汇编语言程序，因此生成的输出也一样。当然，严谨的程序员不会写出第二个程序，因为它比第一个程序难理解多了。

```
汇编语言长子

0000  51000A CHARO   0x000A,d
0003  51000B CHARO   0x000B,d
0006  51000C CHARO   0x000C,d
0009  00     STOP
000A  50756E .ASCII  "Pun"
000D         .END

汇编语言程序

0000  51000A CHARO   0x000A,d
0003  51000B CHARO   0x000B,d
0006  51000C CHARO   0x000C,d
0009  00     STOP
000A  50756E CHARO   0x756E,i
000D         .END

程序输出

Pun
```

图 5-14　两个产生相同目标程序从而产生相同输出的源程序

因为有伪操作，所以反汇编映射不是唯一的。如果没有伪操作，从二进制目标代码还原出原始汇编语言语句就会只有一种可能的形式。伪操作用来把数据位，而不是指令位，插入内存中。数据和程序共享内存是反汇编映射不唯一的主要原因。

对于软件开发者来说，从目标程序恢复源程序很困难是一个有利于市场的好处。如果用汇编语言写了一个应用程序，你有两种方式销售它：一种是卖源程序，让客户对它进行汇编，那么客户就拥有源程序和目标程序；一种是你自己汇编，仅出售目标程序。

采用这两种方式，客户都有执行应用程序所必需的目标程序。但是如果客户也有源程序，他就可以很容易地修改源程序以适应他自己的目的。他甚至只需稍微费点儿功夫就可以改进这个源程序，把它作为一个增强版来销售，并和你形成直接竞争。修改机器语言程序就会困难很多。因此为了防止客户篡改程序，大多数商业软件产品只出售目标代码形式的产品。

开源软件运动是计算机工业近来的一个发展。这个理念是由于支持的问题，客户拥有源程序是有益的。如果只有目标程序，那么在发现了需要修补的漏洞，或者发现了一个需要增加的特性时，必须等待卖给你软件的公司修补漏洞或增加特性。如果拥有源程序，就可以自己修改使之适合你自己的需要。有些开源公司确实免费提供源代码，通过为产品提供软件支持获得收入。采用这种策略的一个例子是 Linux 操作系统，它可以免费从因特网获得。尽管这样的软件是免费的，但是使用它需要有较高水平的技能。

反汇编器（disassembler）是一个试图从目标程序恢复源程序的程序。因为反向汇编器映射的非唯一性，所以反汇编器不一定能 100% 成功。本章中的程序把数据放在指令的前面或者指令的后面。在大的程序中，数据部分的放置通常会贯穿程序，这就使得在目标程序中辨别指令位和数据位很困难。反汇编器能够读取每个字节并数次输出它——一次解释为指令指示符，一次解释为 ASCII 字符，一次解释为二进制补码表示的整数等。然后，人们可以尝试去重构源程序，但这个过程非常单调乏味。

209
:
210

5.3 符号

前面一节介绍了转移指令 BR，用来绕开程序开头的数据。尽管这个技术减轻了手工确定数据单元地址的问题，但它不能完全消除这个问题。仍然必须通过十六进制计数来确定这些数据单元的地址，如果数据单元数量很大，就可能出错。而且，如果想修改数据部分，比如删除一条 .WORD 命令，这条删除的指令后面的所有数据单元的地址都将会改变。这样就必须修改所有引用了这些修改过的地址的指令。

汇编语言符号可以消除手工确定地址的问题。类似于 C++ 的标识符，汇编器让一个符号（symbol）和一个内存地址关联起来。在程序中任何需要引用这个地址的地方，都可以引用这个符号。如果通过增加或删除语句修改程序，那么当汇编这个程序时，汇编器将计算和这个符号相关联的新地址，这样就不需要重写通过符号引用了被改变的地址的语句。

5.3.1 带符号的程序

图 5-15 的汇编语言生成的目标代码和图 5-12 中的一样，它使用了 3 个符号：num、msg 和 main。

```
汇编器列表
Addr  Code   Symbol  Mnemon  Operand    Comment
0000  040000         BR      main       ;Branch around data
0003  0000   num:    .BLOCK  2          ;Storage for one integer
0005  202820 msg:    .ASCII  " + 1 = \x00"
      312B3D
      2000
            ;
000D  310003 main:   DECI    num,d      ;Get the number
0010  390003         DECO    num,d      ;and output it
0013  410005         STRO    msg,d      ;Output ' + 1 = '
0016  C10003         LDA     num,d      ;A <- the number
0019  700001         ADDA    1,i        ;Add one to it
001C  E10003         STA     num,d      ;Store the sum
001F  390003         DECO    num,d      ;Output the sum
0022  00             STOP
0023                 .END

Symbol table:
Symbol      Value
main        000D
msg         0005
num         0003

输入
-479

输出
-479 + 1 = -478
```

图 5-15 一个计算十进制值加 1 的程序。除了使用符号之外，其他的部分与图 5-12 相同

符号的语法规则类似于 C++ 标识符的语法规则，第一个字符必须是字母，接下来的字符必须是字母或数字。符号的长度最多是 8 个字符，并且是区分大小写的。例如，Number 和 number 是不同的符号，因为大写字母 N 和小写字母 n 是不同的。

可以通过把它放在任何一个汇编语言行的前面来定义一个符号。当定义了一个符号时，必须用冒号：来结束，最后一个字符和冒号之间不能有空格。在这个程序中，语句

```
num: .BLOCK 2 ;Storage for one integer
```

除了分配了一个2字节的块以外，还定义了一个字符num。这一行中，冒号和伪操作之间有空格，但是汇编器并不要求一定要有空格。

当汇编器检测到符号定义时，就会在符号表中存储这个符号和它的值。这个值是内存中该行生成的目标代码的第一个字节将要被装入的地址。如果在程序中定义了一些符号，那么汇编器列表会输出一个符号表，其中的值以十六进制表示。图5-15展示了这个程序的符号表输出。从表中可以看到符号num的值是0003（hex）。

当引用符号时，不能包括冒号。语句

```
LDA num,d ;A <- the number
```

211
↓
212

引用了符号num。因为num的值是0003（hex），所以这条语句生成的代码和语句

```
LDA 0x0003,d ;A <- the number
```

生成的代码一样。类似地，因为main的值是000D（hex），所以语句

```
BR main ;Branch around data
```

生成的代码和语句

```
BR 0x000D ;Branch around data
```

生成的代码是一样的。

注意，符号的值是地址，而不是那个地址单元的内容。当程序执行时，Mem[0003]将包含-479（hex），它取自输入设备。num的值仍然是0003（hex），而不是-479。这两者是不同的。可以把符号的值想象成来自于汇编器列表中包含该符号定义的那一行的地址栏。

符号不仅把人们从手工计算地址的负担中解脱出来，而且也使程序更易读。在视觉上，num比0x0003更易读。优秀的程序员会非常细心地为他们的程序挑选有意义的符号来增加程序的可读性。

5.3.2　一个冯·诺依曼示例

当在Asmb5层用符号编程时，很容易忘记计算机的冯·诺依曼本质。两个经典的冯·诺依曼漏洞（把指令当作数据操作和把数据当作指令来执行）仍然存在。

例如，思考下面的汇编语言程序：

```
this: DECO this,d
      STOP
      .END
```

你可能认为汇编器会拒绝第一条指令，因为它看上去在把它自己当作数据进行引用，这貌似没有意义。但是汇编器不会往前看执行的结果。因为语法是正确的，所以它相应地进行翻译图5-16所示的汇编器列表所示。

在执行期间，CPU把39看作采用直接寻址的十进制输出指令的操作码，把Mem[0000]的字3900（hex）看作十进制数字，并输出它的值14 952。

```
汇编器列表
0000  3900DD this:  DECO    this,d
0003  00            STOP
0004                .END
输出
14592
```

图5-16　说明机器底层冯·诺依曼本质的没有实际意义的程序

认识到计算机硬件没有天生的智能也没有推理能力

是很重要的。执行周期和指令集固化在CPU中。就像这个程序说明的那样，CPU不知道它处理的位历史。它没有总体图。它只是一遍又一遍地执行冯·诺依曼循环。主存也是这样的。它不知道它存储过的位历史。它只是根据CPU的命令存储1和0。任何智能或推理能力都来自于软件，而软件是人写的。

5.4 从HOL6层翻译

编译器把高级语言（HOL6层）程序转换为较低级语言的程序，最终程序能被机器执行。有些编译器直接翻译成机器语言（ISA3层），如图5-17a所示，这样程序就可以装入内存并执行。另一些编译器翻译成汇编语言（Asmb5层），如图5-17b所示，接下来必须由汇编器把汇编语言程序再翻译成机器语言，然后才能装入和执行。



图5-17 编译器的功能

与汇编器一样，编译器也是一个程序。它必须像其他程序一样地编写和调试。编译器的输入叫作源程序，不管输出是机器语言还是汇编语言都叫作目标程序。这和汇编器的输入/输出术语是一样的。

本节讲述从C++到Pep/8汇编语言的翻译过程。它展示编译器怎样翻译cin、cout和赋值语句，以及它是怎样在C++层实施类型（type）的概念。第6章将继续讨论高级语言层（HOL6层）和汇编语言层（Asmb5层）之间的关系。

5.4.1 cout语句

图5-18中的程序展示了编译器怎样把一个简单的、只有一条输入语句的C++程序翻译成汇编语言。

编译器把C++语句

```
cout << "Love" << endl;
```

翻译为两条可执行的汇编语句

```
STRO msg,d
CHARO '\n',i
```

和一个点命令

```
msg: .ASCII "Love\x00"
```

STRO语句对应于发送“Love”到cout，CHARO指令对应于发送endl到cout。这是一个一对三映射。与汇编器相比，编译器的映射通常不是一对一的，而是一对多的。这个程序和后面的所有程序都把字符串常量放在程序的底部。

```
高级语言
#include <iostream>
using namespace std;
int main () {
   cout << "Love" << endl;
   return 0;
}

汇编语言
0000  410007          STRO    msg,d
0003  500004          CHARO   '\n',i
0006  00              STOP
0007  4c6f76 msg:     .ASCII  "Love\x00"
      6500
000C                  .END

输出
Love
```

图5-18 HOL6层和Asmb5层的cout语句

对应于变量值的数据放在程序的顶部，对应于它们在 HOL6 程序中的放置位置。

编译器把 C++ 语句

```
return 0;
```

翻译为汇编语言语句

```
STOP
```

除了 main() 之外 C++ 函数的 return 语句不翻译为 STOP。这个对 main() 的 return 的翻译是一个简化。实际的 C++ 编译器必须生成在特定操作系统上执行的代码。由操作系统来解释 main() 函数的返回值。通常的惯例是返回值 0 表示程序的执行没有发生错误；如果发生了错误，程序返回某个非零值。但是这种情况下会发生什么取决于具体的操作系统。在 Pep/8 系统中，从 main() 的返回对应于终止程序，因此从 main() 返回将总是被翻译成 STOP。第 6 章将展示编译器怎样翻译 main() 之外的其他函数的 return。

C++ 程序的其他部分甚至直接不转换。例如，

```
#include <iostream>
using namespace std;
```

根本不会在汇编语言程序中出现。实际的 C++ 编译器会用 include 和 using 语句生成到操作系统和它库的正确接口。因为这里我们只做简单的介绍，所以 Pep/8 系统会忽略这种细节。

图 5-19 展示了编译这个程序的编译器的输入和输出。(a) 部分是直接翻译为机器语言的编译器，目标程序可以装入和执行。(b) 部分是翻译为 Asmb5 层汇编语言的编译器，在装入和执行前，还需要汇编为目标程序。

213 ↓ 216

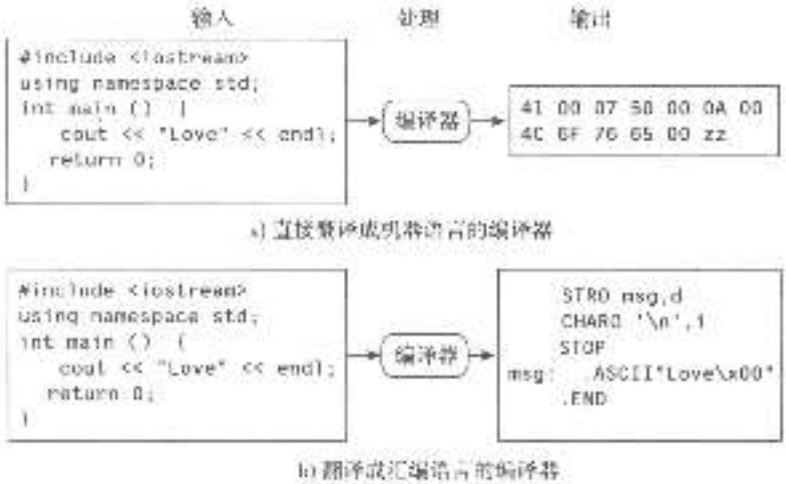


图 5-19　编译器对图 5-18 中的程序所做的行为

5.4.2　变量和类型

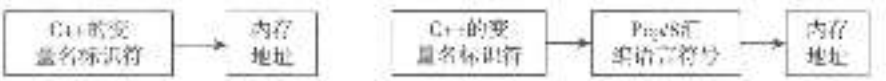
每个 C++ 变量有 3 个属性——变量名、变量类型和变量值。对每个声明的变量，编译器在机器语言程序中会保留一个或多个存储单元。高级语言中的变量在低级语言中就只是一个存储单元。HOL6 层程序通过名字（即 C++ 标识符）引用变量，而 ISA3 层程序通过地址引用它们。变量值是与 C++ 标识符相关联的地址处的存储单元的值。

编译器必须记住哪个地址对应 HOL6 层程序中的哪个变量名。它用一个符号表来建立变

量名和地址之间的关联。

编译器的符号表类似于汇编程序的符号表，但是内在要复杂得多。C++中的变量名没有限制为最多8个字符，而Pep/8中的符号有这个限制。此外，编译器的符号表必须要存储变量类型以及它的关联地址。

直接翻译为机器语言的编译器不需要用汇编器进行二次翻译。图5-20a展示了这样一个编译器的符号表产生的映射。不过，本书中的程序说明了一个翻译为汇编语言的、假设的编译器的翻译过程，因为汇编语言比机器语言更易读。C++变量名对应于Pep/8汇编语言的符号，如图5-20b所示。



a) 直接翻译成机器语言的编译器　　b) 一个用以说明目的的假想汇编器

图5-20 编译器在HOL6层变量和ISA3层存储位置之间所做的映射

图5-20b中的对应关系，对于翻译为汇编语言的编译器来说是不太实际的。设想一个有两个变量discountRate1和discountRate2的C++程序，因为这两个变量的长度都大于8个字符，所以编译器很难把这两个标识符映射到各自唯一的Pep/8符号。为了使C++和汇编语言之间的对应关系清晰，我们的例子会把C++标识符限制为最多8个字符。真实的、翻译到汇编语言的编译器通常不会对变量名使用汇编语言符号。

5.4.3 全局变量和赋值语句

图5-21中的C++程序来自图2-4。它给出了HOL6层的全局变量的赋值语句和相应的汇编语言程序。这里的目标程序包含注释，但是实际的编译器不会产生注释，因为人类程序员通常不需要读目标程序。

```
高级语言
#include <iostream>
using namespace std;

char ch;
int j;

int main () {
   cin >> ch >> j;
   j += 5;
   ch++;
   cout << ch << endl << j << endl;
   return 0;
}

汇编语言
0000  040006          BR      main
0003  00      ch:     .BLOCK  1           ;global variable #1c
0004  0000    j:      .BLOCK  2           ;global variable #2d
              ;
0006  490003 main:    CHARI   ch,d        ;cin >> ch
0009  310004          DECI    j,d         ;   >> j
```

图5-21 HOL6层和Asmb5层的全局变量赋值语句。该C++程序来自图2-4

```
000C  C10004          LDA     j,d         ;j += 5
000F  610005          ADDA    5,i
0012  E10004          STA     j,d
0015  810003          LDBYTEA ch,d        ;ch++
0018  700001          ADDA    1,i
001B  F10003          STBYTEA ch,d
001E  510003          CHARO   ch,d        ;cout << ch
0021  500A0A          CHARO   '\n',i      ;   << endl
0024  390004          DECO    j,d         ;   << j
0027  500A0A          CHARO   '\n',i      ;   << endl
002A  00              STOP
002B                  .END

输入
M 419

输出
M
424
```

图 5-21（续）

记住编译器是一个程序，它必须像其他程序一样进行编写和测试。一个翻译 C++ 程序的编译器可以用任何语言来编写——甚至 C++！下面的程序段说明了这个交错的状态。它是把 C++ 源程序翻译成汇编语言目标程序的简化编译器的一部分。

```
typedef int HexDigit;
enum KindType {sInt, sBool, sChar, sFloat};
struct SymbolTableEntry {
   char symbol[32];
   HexDigit value[4];
   KindType kind;
};

SymbolTableEntry symbolTable[100];
```

符号表中的一个条目包含三部分——符号本身；它的值，即 Pep/8 内存中存储变量值的地址；存储的值的种类，即变量的类型。

图 5-22 展示了这个程序符号表中的条目。第一个变量的符号名是 ch，编译器通过生成 .BLOCK 命令在 Mem[0003] 分配 1 字节，在符号表中把它的类型存储为 sChar，表明这个变量是一个 C++ 字符。第二个变量的符号名是 j，编译器为这个值在 Mem[0004] 处分配 2 字节，把它的类型存储为 sInt，表明是一个 C++ 整型。编译器从 C++ 程序的变量声明得出变量类型。

在代码生成阶段，编译器把

```
cin >> ch >> j;
```

翻译为

```
CHARI 0x0003,d
DECI 0x0004,d
```

	符号	值	类型
[0]	ch	0003	sChar
[1]	j	0004	sInt
[2]	⋮	⋮	⋮

图 5-22　一个假想的编译器翻译图 5-21 所示程序时的符号表

编译的前一阶段会填充符号表，此时，编译器会查阅符号表来确定 CHARI 和 DECI 指令操作数的地址。如同之前解释过的，为了易读性，代码中

会托生成的指令写作

```
CHAR1 ch,0
DEC2 j,0
```

注意，存储在符号表中的值并不是执行期间变量的值，而是存储值的内存地址。如果执行时用户给 j 输入 419，那么存储在 Mem[0004] 的值将是 01A3（hex），即 419（dec）的二进制表示。在翻译时，符号表中符号 j 的值是 0004 而不是 01A3。在翻译时 C++ 变量值是不存在的，它们只存在于执行时。

在 HOL6 层给一个变量赋值对应于在 Asmb5 层把值存储到内存。编译器把赋值语句

```
j += 5;
```

翻译成

```
LDA j,0
ADDA 5,i
STA j,0
```

LDA 和 ADDA 执行赋值语句右边的计算，把计算结果放在累加器中。STA 把计算结果再赋值给 j。

这个赋值语句说明了存取全局变量的一般规则：

- 变量的符号是值的地址。
- 用直接寻址方式访问值。

217
:
220

在本例中，全局变量符号 j 是地址 0004，LDA 和 STA 语句使用直接寻址方式。

类似地，编译器把

```
ch++
```

翻译为

```
LDBYTEA ch,d
ADDA 1,i
STBYTEA ch,d
```

与 j 加 5 相同的指令，ADDA，对 ch 执行增量运算，同样，因为 ch 是全局变量，所以它的值是地址 0003，LDBYTEA 和 STBYTEA 指令使用直接寻址方式。

编译器把

```
cout << ch << endl << j << endl;
```

翻译为

```
CHARO ch,d
CHARO '\n',i
DECO j,d
CHARO '\n',i
```

用直接寻址输出全局变量 ch 和 j 的值。

编译器必须搜索符号表来建立像 ch 这样的符号和它的地址 0003 之间的关联。符号表是一个数组。如果它不按照符号名的字母顺序维护，那么要在表中定位 ch 就需要进行顺序搜索；如果符号名按照字母顺序排序，那么就可以使用二分查找。

5.4.4 类型兼容

为了了解在 HOL6 层是怎样保证类型兼容的，假设一个 C++ 程序中有两个变量，类型

j 和浮点型 y。同时假定有一台不同于 Pep/8 的计算机，它可以存储和处理浮点型值，就叫作 Pep/99 吧。浮点型数的二进制编码方式不同于整数的编码方式，它们以二进制科学表示法存储，指数部分和尾数部分分开存储在不同的单元中。你程序的编译器符号表可能会看上去像图 5-23。

	符号	值	类型
[0]	j	0003	sInt
[1]	y	0005	sFloat
[2]	⋮	⋮	⋮

图 5-23　Pep/99 编译器的符号表

现在来考虑 C++ 中的运算 j%8。% 是模运算符，只限于对整数值进行运算。在二进制中执行 j%8，就是把除了最右边 3 位以外的所有其他位都置为 0。例如，如果 j 的值是 61（dec）= 0011 1101（bin），那么 j%8 的值是 5（dec）= 0000 0101(bin)，就是把除了最右边 3 位外的所有其他位都置为 0。

假定你的 C++ 程序中有下面的语句：

```
i = j % 8;
```

编译器会查阅符号表并确定变量 j 的 kind 是 sInt，同时识别 8 是一个整型常量并确定 % 运算是合法的，接着它将生成目标代码

```
LDA j,d
ANDA 0x0007,i
STA j,d
```

现在假设你的 C++ 程序中有下面的语句：

```
y = y % 8;
```

编译器会查阅符号表并确定变量 y 的 kind 是 sFloat，它确定 % 运算是不合法的，因为该运算仅适用于整型类型。它将生成出错信息

```
error: float operand for %
```

而不会生成目标代码。然而如果没有类型检测，将会生成下面的代码：

```
LDA y,d
ANDA 0x0007,i
STA y,d
```

虽然执行它不会生成有意义的结果，但实际上也确实没有什么能阻止汇编语言程序员写出这样的代码。

让编译器检测类型的兼容性有巨大作用。它可以防止编写无意义的语句，例如对浮点型变量执行 % 运算。当在 Asmb5 层直接用汇编语言编程时，没有类型兼容检测。所有数据都是由位组成的。当由于不正确的数据移位导致漏洞出现时，只能在运行时而不能在翻译时检测它们。也就是说，它们是逻辑错误而不是语法错误。众所周知，逻辑错误比句法错误更难以定位。

221
↓
222

5.4.5　Pep/8 符号跟踪器

对应于 C++ 内存模型的 3 个部分，Pep/8 有 3 种符号跟踪特性——用于全局变量的全局跟踪器；用于参数和局部变量的栈跟踪器；用于动态分配变量的堆跟踪器。为了跟踪一个变量，程序员把跟踪标签嵌入与此变量相关的注释中，并单步调试程序。Pep/8 的集成开发环境会显示变量的运行时值。

有两种跟踪标签：

- 格式跟踪标签
- 符号跟踪标签

跟踪标签放在汇编语言注释中，对生成的目标代码没有影响。每个跟踪标签以#字符开始，向符号跟踪器提供在跟踪窗口中如何格式化和标记存储单元的信息。在汇编代码时，跟踪标签出错会作为警告，允许程序继续执行，只是跟踪功能关闭了。如果不纠正错误，就一直无法跟踪。

全局跟踪器允许用户指定跟踪哪个全局符号，只需要在声明全局变量的 .BLOCK 行的注释中放一个格式跟踪标签。例如，图 5-21 中的这两行，

```
ch: .BLOCK  1   ;global variable #1c
j:  .BLOCK  2   ;global variable #2d
```

包含格式跟踪标签#1c 和#2d。第一个格式跟踪标签可以解读为“1 字节，字符。”这个跟踪标签让符号跟踪器把符号 ch 本身和符号值指定的地址处的 1 字节存储单元的内容一起显示出来。类似地，第二个跟踪标签让符号跟踪器显示 j 指定的地址处的 2 字节单元。把单元的内容当作一个十进制数。

跟踪标签的合法格式是

#1c　1 字节字符
#1d　1 字节十进制
#2d　2 字节十进制
#1h　1 字节十六进制
#2h　2 字节十六进制

全局变量不要求使用符号跟踪标签，因为 Pep/8 符号跟踪器会从放置跟踪标签的 .BLOCK 行获取该符号。但是局部变量要求使用符号跟踪标签，这将在第 6 章中讲述。 223

5.4.6 算术移位和循环移位指令

Pep/8 有两个算术移位指令和两个循环移位指令。这 4 条指令都是一元指令，它们的指令指示符、助记符和所影响的状态位如下所示：

0001 110r	ASLr	算术左移 r	NZVC
0001 111r	ASRr	算术右移 r	NZC
0010 000r	ROLr	循环左移 r	C
0010 001r	RORr	循环右移 r	C

算术移位和循环移位指令没有操作数指示符。每条指令根据 r 的值决定作用于累加器还是变址寄存器。如第 3 章所述，算术左移是有符号整数乘以 2，算术右移是有符号整数除以 2。循环左移是每位往左移动 1 位，最高有效位移到 C 位，C 位移到最低有效位；循环右移是每位往右移动 1 位，最低有效位移到 C 位，C 位移到最高有效位。

ASLr 指令的寄存器传送语言（RTL）描述是

C ← r<0>, r<0..14> ← r<1..15>, r<15> ← 0

N ← r < 0, Z ← r = 0, V ← { 溢出 }

ASRr 指令的 RTL 描述是

C ← r<15>, r<1..15> ← r<0..14>; N ← r < 0, Z ← r = 0

ROLr 指令的 RTL 描述是

C ← r<0>, r<0..14> ← r<1..15>, r<15> ← C

RORr 指令的 RTL 描述是

C ← r<15>, r<1..15> ← r<0..14>, r<0> ← C

例 5.4　假定要执行指令的十六进制形式为 1E，图 5-24 是它的二进制表示。操作码表明将执行 ASRr 指令，寄存器 r 字段表明指令将作用于累加器。

图 5-25 展示了假设累加器的初始内容为 0098（hex）= 152（dec）时，执行 ASRA 指令
224 的结果。ASRA 指令将位模式改变为 004C（hex）= 76（dec），这个值是 152 的一半。因为累加器中的数为正，所以 N 位为 0；因为累加器不为全 0，所以 Z 位为 0；因为位移前最低有效位为 0，所以 C 位为 0。 □



图 5-24　ASRA 指令



图 5-25　ASRA 指令的执行

Atanasoff、Mauchly 和 Eckert



确定现代计算机的缔造者是一个很复杂的问题。我们必须承认这样一些做出贡献的人和事，比如古代中国的算盘、19 世纪 Charles Babbage 的分析引擎、Lovelace 女士对计算的思考，但是我们搜寻的真正目标是电子数字计算机的直接发明者。候选人包括 John Atanasoff、Clifford Berry、John Mauchly、J. Presper Eckert、KonradZuse 和 John von Neumann。这些人中的每一个都在 20 世纪 30 年代末和 40 年代初对电子计算设备的研究和发展做出过积极的追求。

20 世纪 50 年代到 60 年代，Mauchly 和 Eckert 被较为广泛地认可为电子计算机之父，而 1945 年建成的 ENIAC（Electronic Numerical Integrator and Calculator，电子数字积分计算机）被认为是第一台电子计算机。Mauchly 和 Eckert 在 ENIAC 上成功的工作吸引了很多的研究经费，然后又衍生出商业投资，当美国军方决定采用 ENIAC 来帮助决定准确计算弹道表的时候，他们的工作也获得了很高的知名度。

1951 年，Sperry Rand 购买了 ENIAC 的专利和它底层的理念。然后，这家公司要求其他计算机制造商为使用他们拥有的这些重要概念支付版税——特别是，现代计算机中都可以看到的基础体系结构，比如通过逻辑开关做算术运算的电路，防止电子表示的信息失效的内存刷新电路。Honeywell 公司不想要为他们建造和出售的每台计算机都支付昂贵的版税，所以他们让他们的律师研究了现代计算机的历史。

Honeywell 的律师拿出了证据证实荷华州立大学（Iowa State College）的 John Atanasoff 和 Clifford Berry 先于 Mauchly 和 Eckert 几年提出了关键性的技术。ENIAC 到 1945 年才可以实际运行；而 Atanasoff 在 1939 年就有了可工作的原型系统，在 1942 年就有了一个特殊用途的计算机。Atanasoff 的机器包括有 ENIAC 和几乎所有其他商业上成功的机器中使用的内存刷新电路和电子加法器/减法器电路。同时，Honeywell 的律师还发现 Mauchly 在 1941 年 6 月访问过 Atanasoff 几天，在那之后 Atanasoff 的一些思想出现在了 Mauchly 的项目中。所有这些信息都有力地表明 Atanasoff 的工作直接影响了 ENIAC 的

开发。最终，1973年，法庭判定ENIAC的专利无效，认定Atanasoff的贡献非常重要，Mauchly借鉴了这些思想。Atanasoff如果想要保护他自己，是可以申请专利的。但是Atanasoff和Berry并没有很热切地市场化他们的研究，而当第二次世界大战中他们不得不服役时，他们的项目陷入停顿。

几乎与Atanasoff、Berry、Mauchly和Eckert同时，Konrad Zuse也在脑海中构思了一台通用计算机。不幸的是，Zuse当时并没力于德国，所以他思想的种子没有结出果实。

"我们预测全球市场也许可以容纳5台计算机。"

——Tom Watson

IBM主席，1949年

5.4.7 常量和.EQUATE

.EQUATE是少数几个不生成目标代码的伪操作之一。此外，从目标代码地址获取符号值的标准机制在这里不起作用。.EQUATE的操作说明如下：

- 它一定在定义符号的行中。
- 它使得符号的值等于跟在.EQUATE之后的值。
- 它不生成任何目标代码。

C++编译器用.EQUATE命令翻译C++常量。

除了变量是全局的而不是局部的之外，图5-26中的程序和图2-6中的程序是一样的。它展示了怎样把一个C++常量翻译为机器语言，同时也说明了ASRA汇编语言语句的使用。这个程序计算两次测验的平均分加上5分奖励分后得到的分值。

高级语言

```
#include <iostream>
using namespace std;

const int bonus = 5;
int exam1;
int exam2;
int score;

int main () {
   cin >> exam1 >> exam2;
   score = (exam1 + exam2) / 2 + bonus;
   cout << "score = " << score << endl;
   return 0;
}
```

汇编语言

```
0000  040009          BR      main
            bonus:    .EQUATE 5           ;constant
0003  0000  exam1:    .BLOCK  2           ;global variable @3d
0005  0000  exam2:    .BLOCK  2           ;global variable @2d
0007  0000  score:    .BLOCK  2           ;global variable @2d
            ;
0009  310003 main:    DECI    exam1,d     ;cin >> exam1
000C  310005          DECI    exam2,d     ;   >> exam2
```

图5-26 一个程序，编译器会把它的C++常量翻译成机器语言

```
0009  C10003          LDA     exam1,d     ;score = (exam1
0012  710005          ADDA    exam2,d     ;   + exam2)
0015  1E              ASRA                ;   / 2
0016  700005          ADDA    bonus,i     ;   + bonus
0019  E10007          STA     score,d
001C  410026          STRO    msg,d       ;cout << "score = "
001F  390007          DECO    score,d     ;   << score
0022  50000A          CHARO   '\n',i      ;   << endl
0025  00              STOP
0026  73636F msg:     .ASCII  "score = \x00"
      6F7265
      203D00
002F                  .END

Symbol table:
Symbol      Value     Symbol      Value
bonus       0005      exam1       0003
exam2       0005      score       0007
main        0009      msg         0026

输入
68 84

输出
score = 91
```

图 5-26（续）

编译器把

```
const int bonus = 5;
```

翻译为

```
bonus: .EQUATE 5
```

图 5-26 中的汇编语言代码有两点值得注意。首先，含有 .EQUATE 的行在机器语言列中是没有代码的。由于没有代码，地址也就不适用了，所以甚至在地址列中都没有地址。这和 .EQUATE 不生成代码的规则相一致。其次，图 5-26 包括来自汇编器列表的符号表。从表中可以看到符号 bonus 的值为 5，符号 exam2 的值也为 5，但是原因不同。exam2 的值为 5，因为给它生成代码的 .BLOCK 命令在地址 0005（hex）处。但 bonus 没有代码，它的值为 5，因为它被 .EQUATE 点命令设置为 5。

225 ; 227

I/O 和赋值语句类似于前面的程序。cin 按照需求翻译为 DECI 或 CHARI，cout 翻译为 DECO 或 CHARO，对于全局变量都采用直接寻址方式。一般来说，赋值语句翻译为

- 装入寄存器。
- 如果需要，对表达式求值。
- 存储寄存器。

为了计算表达式

```
(exam1 + exam2) / 2 + bonus
```

编译器生成代码把 exam1 的值装入累加器，再加上 exam2 的值，然后用 ASRA 指令把和除以 2。LDA 和 ADDA 指令使用直接寻址，因为 exam1 和 exam2 是全局变量。

但编译器怎样生成加 bonus 的代码呢？它不能用直接寻址，因为没有目标代码对应于

bonus，因此也就没有地址。取而代之的是语句

```
ADDA bonus,i
```

使用立即数寻址。此时，操作数指示符 0005（hex）= 5（dec）就是要加上的值。把 C++ 常量翻译到汇编语言的一般规则是：

- 用 .EQUATE 声明常量。
- 用立即数寻址访问常量。

在更为实际的程序中，score 的类型会是 float，要用实数除法运算来计算平均分。Pep/8 没有硬件支持实数，它的指令集也不包含整数乘法和除法指令。这些运算一定要用算术移位和循环移位指令编程来实现。

5.4.8 指令和数据的放置

本书的目的是展示典型计算机系统中抽象层次之间的关联。所以，Asmb5 层翻译出来的整体程序结构对应于被翻译的 HOL6 程序的结构。具体来说，在 Asmb5 程序和 HOL6 程序中，全局变量都出现在主程序的前面。实际的编译器并没有这个约束，经常会改变程序和数据的位置。图 5-27 是对图 5-26 中 C++ 程序的一个不同的翻译结果，这个翻译的一个好处是没有程序开始处跳转到主程序的那个分支语句。 228

```
0000 310010 main:   DECI    exam1,d     ;cin >> exam1
0003 31001F         DECI    exam2,d     ;   >> exam2
0006 C10010         LDA     exam1,d     ;score = (exam1
0009 710021         ADDA    exam2,d     ;   + exam2)
000C 1F             ASRA                ;   / 2
000D 700005         ADDA    bonus,i     ;   + bonus
0010 E10021         STA     score,d
0013 410023         STRO    msg,d       ;cout << "score = "
0016 390021         DECO    score,d     ;   << score
0019 50000A         CHARO   '\n',i      ;   << endl
001C 00             STOP
                ;
0005            bonus:  .EQUATE 5           ;constant
001D 0000       exam1:  .BLOCK  2           ;global variable #2d
001F 0000       exam2:  .BLOCK  2           ;global variable #2d
0021 0000       score:  .BLOCK  2           ;global variable #2d
0023 73636F msg:    .ASCII  "score = \x00"
     726520
     3D2000
002C                .END

Symbol table:
Symbol      Value       Symbol      Value
main        0000        bonus       0005
exam1       001D        exam2       001F
score       0021        msg         0023
```

图 5-27 图 5-26 中 C++ 程序的另一种翻译，指令和数据的放置不同

总结

汇编器是一个把汇编语言程序翻译为等效机器语言程序的程序。冯·诺依曼设计原理要求指令和数据都存储在主存中。对应于两种位序列，汇编语言语句有两种类型。对于程序语句，汇编语言用助记符代替操作码和寄存器 -r 字段，十六进制代替二进制操作数指示符，助记符字母代替寻址方式。对于数据语句，汇编语言使用伪操作，也叫作点命令。 229

使用直接寻址方式时，操作数指示符是操作数的主存地址，而使用立即数寻址方式时，操作数指示符就是操作数，以数学方法表示为 Oprnd = OprndSpec。立即数寻址比直接寻址更好，因为立即数寻址的操作数不需要和指令分开存储。因为在指令寄存器中的操作数对 CPU 来说是立即可用的，所以这样的指令执行也更快。

汇编语言符号消除了程序中需要手工确定数据和指令地址的问题。符号的值是一个地址。当汇编器检测到一个符号定义时，会把该符号和它的值存储到符号表中。当需要使用某个符号时，汇编器会用符号的值代替符号。

高级语言层（HOL6 层）的变量对应于汇编层（Asmb5 层）的内存地址。在 HOL6 层，把表达式赋值给变量的赋值语句会在 Asmb5 层翻译为一个装入（load），后面跟一个表达式求值，再跟一个存储（store）。HOL6 层的类型兼容通过编译器和它的符号表来实现，这个符号表要比汇编器的符号表复杂很多。在 Asmb5 层，唯一的类型是位，可以对任何位模式做任何操作。

练习

5.1 节

*1. 把下列机器语言指令转换为汇编语言，假设这些指令不是由伪操作生成的：

(a) AAEF2A　　(b) 02　　(c) D7003D

2. 把下列机器语言指令转换为汇编语言，假设这些指令不是由伪操作生成的：

(a) 92B7DE　　(b) 03　　(c) DF63DF

*3. 把下列汇编语言指令转换为十六进制机器语言：

(a) ASLA　　(b) CHART 0x000F,s　　(c) BRNE 0x01E6,i

4. 把下列汇编语言指令转换为十六进制机器语言：

[234] (a) ADDA 0x01FE,i　　(b) STRO 0x000D,df　　(c) LDX 0x01FF,s

*5. 把下列汇编语言伪操作转换为十六进制机器语言：

(a) .ASCII "Beep\x07"　　(b) .BYTE 0xF8　　(c) .WORD 750

6. 把下列汇编语言伪操作转换为十六进制机器语言：

(a) .BYTE 13　　(b) .ASCII "Frog\x00"　　(c) .WORD -6

*7. 预测下面汇编语言程序的输出：

```
CHARO 0xCCOC,d
CHARO 0xCCOB,d
CHARO 0xCCOA,d
STOP
.ASCII "gun"
.END
```

8. 预测下面汇编语言程序的输出：

```
CHARO 0x0008,d
CHARO 0x0007,d
STOP
.ASCII "is"
.END
```

9. 如果输入为 g，预测下面汇编语言程序的输出。如果输入为 A，预测下面汇编语言程序的输出。解释两个结果不同的原因：

```
CHARI 0x0010,d
LDBYTEA 0x0010,d
ANDA 0x0011,d
STBYTEA 0x0010,d
CHARO 0x0010,d
```

```
STOP
.BLOCK 1
.WORD 0xCCCF
.END
```

5.2 节

*10. 如果点命令变成下面这样，预测图 5-13 程序的输出：

```
.WORD 0xFFC7 ;First
.BYTE 0xDD ;Second
.BYTE 'H' ;Third
.WORD 873 ;Fourth
```

231

11. 如果点命令变成下面这样，预测图 5-13 程序的输出：

```
.WORD 0xFE63 ;First
.BYTE 0xCC ;Second
.BYTE 'b' ;Third
.WORD 1401 ;Fourth
```

12. 确定下列汇编语言的目标代码并预测输出：

*(a)

```
DECO 'n',i
CHARO '\n',i
DECO "nn",i
CHARO '\n',i
CHARO 0xCC26,i
STOP
.END
```

(b)

```
DECO 'O',i
CHARO '\n',i
DECO 0xFFC3,i
CHARO '\n',i
CHARO 0xDC7B,i
STOP
.END
```

5.3 节

*13. 在下面的代码中，请确定符号 here 和 there 的值。写出十六进制目标代码（不需预测输出）：

```
        BR      there
here:   .WORD   9
there:  DECO    here,d
        STOP
        .END
```

14. 在下面的代码中，请确定符号 this、that 和 theOther 的值。写出十六进制目标代码（不需预测输出）：

```
        BR      theOther
this:   .WORD   17
that:   .WORD   19
theOther:DECO   this,d
        DECO    that,d
        STOP
        .END
```

*15. 在下面的代码中，请确定符号 this 的值。预测并解释汇编语言程序的输出：

```
this:   CHARO   this,d
        STOP
        .END
```

232

16. 在下面的代码中，请确定符号 this 的值。预测并解释汇编语言程序的输出：

```
this:   DECO    this,d
        STOP
        .END
```

5.4 节

17. 汇编器和编译器的符号表有哪些类似的地方？有哪些不同的地方？

18. C++ 编译器是怎样保证类型兼容的？

19. 假定你有一台 Pep/8 型计算机和下面的磁盘文件：

- A 文件：用机器语言写的 Pep/8 汇编语言汇编器
- B 文件：用汇编语言写的 C++ 到汇编语言的编译器
- C 文件：从一个数据文件读入数字并输出中位数的 C++ 程序
- D 文件：C 文件中求中位数程序的数据文件

要计算中位数，需要图 5-28 所示让计算机运行 4 次，每次运行都包括一个输入文件，某个程序会处理它，并生成一个输出文件。前一次运行生成的输出文件可以作为后面运行的输入文件，或者作为要运行的程序。描述文件 E、F、G 和 H 的内容。用适当的文件字母标注图 5-28 中的空框。

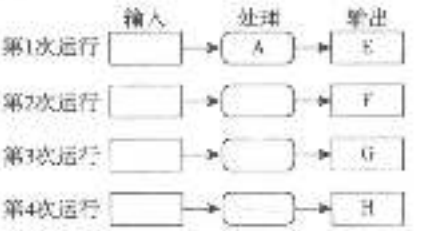


图 5-28　练习 19 中的计算机运行

问题

5.1 节

20. 写一个在屏幕上输出你名字的汇编语言程序。要求使用 .ASCII 伪操作在程序的底部存储字符。使用 CHARO 指令输出字符。

5.2 节

21. 写一个在屏幕上输出你名字的汇编语言程序。要求对于你名字中的每个字母，CHARO 的操作数都使用字符常量立即数寻址。 213

22. 写一个在屏幕上输出你名字的汇编语言程序。要求对于你名字中的每个字母，CHARO 的操作数都使用十进制常量立即数寻址。

23. 写一个在屏幕上输出你名字的汇编语言程序。要求对于你名字中的每个字母，CHARO 的操作数都使用十六进制常量立即数寻址。

5.4 节

24. 写出对应于下面 C++ 程序的汇编语言程序：

```
#include <iostream>
using namespace std;

int num1;
int num2;
main () {
   cin >> num1 >> num2;
   cout << num2 << endl << num1 << endl;
   return 0;
}
```

25. 写出对应于下面 C++ 程序的汇编语言程序：

```
#include <iostream>
using namespace std;

const char chConst = 'a';
char ch1;
char ch2;

int main () {
   cin >> ch1 >> ch2;
   cout << ch1 << chConst << ch2;
   return 0;
}
```

26. 写出对应于下面 C++ 程序的汇编语言程序：

```
#include <iostream>
using namespace std;

const int amount = 20000;
int num;
int sum;

int main () {
   cin >> num;
   sum = num + amount;
   cout << "sum = " << sum << endl;
   return 0;
}
```

234

测试程序两次，第一次，给 num 输入一个值使得 sum 在 Pep/8 计算机允许的范围内。第二次，输入 num 的值，在范围内，但是使 sum 超出范围。注意超出范围的条件不会导致出错信息，只是会给出一个不准确的值。解释这个值。

27. 写出对应于下面 C++ 程序的汇编语言程序：

```
#include <iostream>
using namespace std;

int width;
int length;
int perim;

int main () {
   cin >> width >> length;
   perim = (width + length) * 2;
   cout << "w = " << width << endl;
   cout << "l = " << length << endl;
   cout << endl;
   cout << "p = " << perim << endl;
   return 0;
}
```

28. 写出对应于下面 C++ 程序的汇编语言程序：

```
#include <iostream>
using namespace std;

char ch;

int main () {
   cin >> ch;
   ch--;
   cout << ch << endl;
   return 0;
}
```

29. 写出对应于下面 C++ 程序的汇编语言程序：

```
#include <iostream>
using namespace std;

int num1;
int num2;

int main () {
   cin >> num1;
   num2 = -num1;
   cout << "num1 = " << num1 << endl;
   cout << "num2 = " << num2 << endl;
```

```
    return 0;
}
```

30. 写出对应于下面 C++ 程序的汇编语言程序：

```
#include <iostream>
using namespace std;

int num;

int main () {
   cin >> num;
   num = num / 16;
   cout << "num = " << num << endl;
   return 0;
}
```

31. 写出对应于下面 C++ 程序的汇编语言程序：

```
#include <iostream>
using namespace std;

int num;

int main () {
   cin >> num;
   num = num % 16;
   cout << "num = " << num << endl;
   return 0;
}
```

编译到汇编层

本书的主题是抽象层次的概念在计算机科学中的应用。本章继续这一主题，向你展示高级语言层和汇编层之间的抽象层次关系。本章审视 HOL6 层的 C++ 语言特性，展示编译器怎样把具有这些属性的程序翻译为等效的 Asmb5 层程序。

HOL6 层语言和 Asmb5 层语言的一个主要区别在于 Asmb5 层没有大量的数据类型。在 C++ 中，几乎可以任意组合定义整数、实数、数组、布尔和结构，但汇编语言只有位和字节。如果要用汇编语言定义一个结构数组，就必须相应地区分位和字节。如果在 HOL6 层编程，编译器会自动完成这个工作。

两者之间的另一个不同与控制流有关。C++ 用 `if`、`while`、`do`、`for`、`switch` 和函数语句来改变正常顺序的控制流，但汇编语言受冯·诺依曼基本设计所限，只能使用很原始的控制语句。本章将展示编译器必须怎样把几个原始的 Asmb5 层控制语句联合在一起来执行一条更强大的 HOL6 层控制语句。

6.1 栈寻址和局部变量

当程序调用函数时，程序为返回值、参数和返回地址在运行时栈上分配存储空间，然后函数给它的局部变量分配存储空间。栈相对寻址允许函数访问被压入栈中的信息。

可以把一个 C++ 程序的 `main()` 看作操作系统调用的函数。你可能比较熟悉主程序有 `argc` 和 `argv` 两个参数，像下面这样：

```
int main (int argc, char* argv[])
```


用这样的方式声明 `main()` 函数，`argv` 和 `argc` 与返回地址和局部变量一起被压入运行时栈。

为了使事情更简单，本书总是把 `main()` 声明为不带参数，并忽略给整数返回值和返回地址分配存储空间这一事实。因此，在运行时栈上给 `main()` 分配的唯一存储空间就是给局部变量的。图 2-8 展示了运行时栈上有返回值和返回地址的内存模型。图 2-41 展示了这种简化的内存模型。

6.1.1 栈相对寻址

使用栈相对寻址，操作数和操作数指示符之间的关系是

Oprnd = Mem[SP + OprndSpec]

栈指针作为一个内存地址，操作数指示符会加上这个地址。图 4-39 展示了用户栈在主存中从地址 FBCF 开始向上增加。当一个条目被压入运行时栈时，它的地址小于栈顶条目的地址。

可以把操作数指示符想象成距离栈顶的偏移量。如果操作数指示符是 0，那么指令访问栈顶的值 Mem[SP]；如果操作数指示符是 2，那么就访问栈顶下面 2 字节的值 Mem[SP + 2]。

Pep/8 指令集有两条直接用于操控栈指针的指令，`ADDSP` 和 `SUBSP`（`CALL`、`RETn` 和 `RETTR` 间接操控栈指针）。`ADDSP` 简单地把栈指针值增加一个值，`SUBSP` 将栈指针值减去一

个值。ADDSP 的 RTL 描述是

SP ← SP + Oprnd；N ← SP < 0，Z ← SP = 0，V ← {溢出}，C ← {进位}

SUBSP 的 RTL 描述是

SP ← SP - Oprnd；N ← SP < 0，Z ← SP = 0，V ← {溢出}，C ← {进位}

尽管可以对指针值进行加/减，但是不能用装入指令设置栈指针的值。没有 LDSP 指令。那么究竟怎样设置栈指针呢？当在 Pep/8 模拟器上选择执行选项时，会发生下列两个动作：

SP ← Mem[FFF8]

PC ← 0000

第一个动作把栈指针设置为存储单元 FFF8 的内容。这个位置位于操作系统的 ROM，
238 它包含应用程序运行时栈顶部的地址。因此当选择执行选项时，能正确地初始化栈指针。Pep/8 操作系统会把 SP 默认初始化为 FBCF，应用程序绝不会把它设置为任何其他的值。通常情况下，应用程序只需要在把条目压入运行时栈时增加栈指针，在把条目弹出运行时栈时减小栈指针。

6.1.2　访问运行时栈

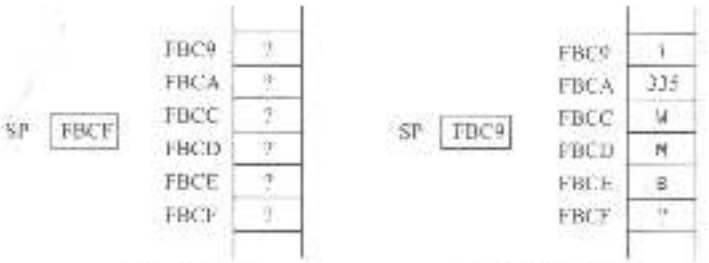
图 6-1 展示了怎样把数据压入栈，用栈相对寻址访问数据，以及从栈中弹出数据。程序把字符串 BMW 压入栈，接着压入十进制整数 335 和字符 '1'，然后输出这些条目，并把它们弹出栈。

```
0000  C00042 LDA     'B',i       ;push B
0003  F3FFFF STBYTEA -1,s
0006  C0004D LDA     'M',i       ;push M
0009  F3FFFE STBYTEA -2,s
000C  C00057 LDA     'W',i       ;push W
000F  F3FFFD STBYTEA -3,s
0012  C0014F LDA     335,i       ;push 335
0015  F3FFFB STA     -5,s
0018  C00069 LDA     '1',i       ;push 1
001B  F3FFFA STBYTEA -6,s
001E  600006 SUBSP   6,i         ;6 bytes on the run-time stack
0021  530005 CHARO   5,s         ;output B
0024  530004 CHARO   4,s         ;output M
0027  530003 CHARO   3,s         ;output W
002A  3B0001 DECO    1,s         ;output 335
002D  530000 CHARO   0,s         ;output 1
0030  600006 ADDSP   6,i         ;deallocate stack storage
0033  00     STOP
0034         .END
输出
BMW3351
```

图 6-1　栈相对寻址

图 6-2a 给出了程序执行前栈指针（SP）和主存的值。机器根据 Mem[FFF8] 的向量把栈指针初始化为 FBCF。

前两条指令



a) 程序执行之前　　b) SUBSP执行之后

图 6-2　把 BMW335i 压入图 6-1 所示的运行时栈中

```
LDA 'B',i
STBYTEA -1,s
```

把 ASCII 字符 'B' 放入栈顶的字节，LDA 把 'B' 字节放入累加器的右半部分，STBYTEA 把它放入栈顶，存储指令采用栈相对寻址，操作数指示符为 -1（dec）= FFFF（hex）。因为栈指针的值是 FBCF，所以 'B' 存储在 Mem[FBCF + FFFF] = Mem[FBCE]。接下来的两条指令分别把 'M' 和 'W' 放在 Mem[FBCD] 和 Mem[FBCC]。

然而，十进制整数 335 占用 2 字节。程序必须把它存储在距离 'W' 地址 2 字节的地方，这就是为何存储 335 的指令是

```
STA -5,s
```

而不是

```
STA -4,s
```

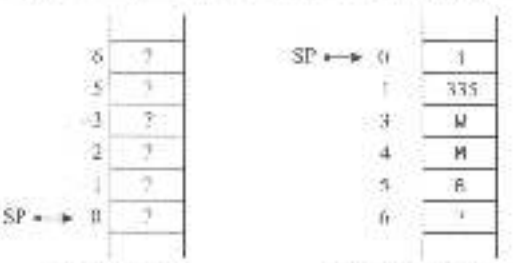
的原因。

通常情况下，当把一个条目压入运行时栈时，必须要考虑每个条目占用的字节数，以及相应地设置好操作数指示符。

SUBSP 指令把栈指针减 6，如图 6-2b 所示，这就完成了入栈操作。

跟踪一个使用栈相对寻址的程序不需要知道栈指针的绝对值。如果栈指针被初始化为其他值，比如 FA18，入栈操作也会做同样的工作。这种情况下，'B'、'M'、'W'、335 和 '1' 会分别在 Mem[FA17]、Mem[FA16]、Mem[FA15]、Mem[FA13] 和 Mem[FA12]，栈指针最后会变成 FA12。尽管绝对内存地址不同，但是这些值相对于栈顶的位置都是相同的。

图 6-3 是一个跟踪运行更便捷的方法，它利用了栈指针中的值是无关的这个事实。不是显示栈指针的值，而是给出指向存储单元的箭头，栈指针中存放着这个存储单元的地址。不显示内存单元的地址，而是给出距离栈顶的偏移量。从现在开始，都会采用这种惯例来画运行时栈的状态图。



a) 程序执行之前　　b) SUBSP执行之后

图 6-3　标明相对地址的图 6-2 的栈

指令

```
CHARO 5,s
```

从栈输出 ASCII 字符 'B'。注意，在 SUBSP 执行前，'B' 的栈相对地址是 -1，但是执行后变成了 5。因为栈指针变了，所以它的栈相对地址不同了。

```
STBYTEA -1,s
```

和

```
CHARO 6,s
```

两者访问相同的内存单元。如图6-3b所示，同样也都用栈偏移量输出其他条目。

指令

```
ADDSP 6,i
```

通过对SP加6在运行时栈上释放了6字节的存储空间。因为栈朝着更小地址的方向向上生长，所以通过减小栈指针值分配存储空间，通过增加栈指针值释放存储空间。

6.1.3　局部变量

前一章我们讲了编译器怎样翻译程序的全局变量。用.BLOCK点命令给全局变量分配存储空间，用直接寻址方式进行存取。然而，局部变量是在运行时栈上分配的，要翻译程序的局部变量，编译器要：

- 用SUBSP分配局部变量。
- 用栈相对寻址访问局部变量。
- 用ADDSP释放存储。

239 | 241

全局变量和局部变量之间的一个重要不同点是分配发生的时间。.BLOCK点命令不是可执行语句，在程序执行前就给全局变量的存储保留了固定的位置。相反，SUBSP是可执行语句，在程序执行期间，在运行时栈上创建局部变量的存储空间。

图6-4中的C++程序来自图2-6，除了变量被声明为main()的局部变量外，和图5-26中的程序是一样的。尽管程序的用户感受不到这个区别，但是编译器执行的翻译却非常不同。图6-5给出了这个程序的运行时栈。和图5-26中一样，bonus是一个常量，用.EQUATE来定义。不过，局部变量也用.EQUATE定义。对于常量，.EQUATE指定常量的值，但对于局部变量，.EQUATE指定在运行时栈上的栈偏移量。例如，图6-5显示局部变量exam1的栈偏移量是4，因此汇编语言程序使得符号exam1等于4。在汇编语言代码中可以看到，.EQUATE不会为局部变量生成任何代码。

高级语言

```
#include <iostream>
using namespace std;

int main () {
   const int bonus = 5;
   int exam1;
   int exam2;
   int score;
   cin >> exam1 >> exam2;
   score = (exam1 + exam2) / 2 + bonus;
   cout << "score = " << score << endl;
   return 0;
}
```

汇编语言

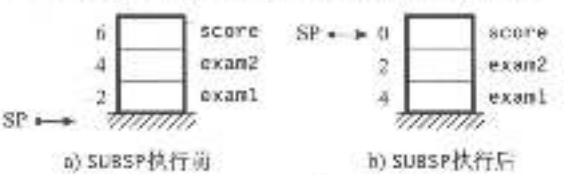
```
0000  040003        BR      main
          bonus:  .EQUATE 5           ;constant
```

图6-4　一个具有局部变量的程序。这个C++程序来自图2-6

```
              exam1:  .EQUATE 4           ;local variable @2d
              exam2:  .EQUATE 2           ;local variable @2d
              score:  .EQUATE 0           ;local variable @2d
                      ;
0003  580006  main:   SUBSP   6,i         ;allocate @exam1 @exam2 @score
0006  330004          DECI    exam1,s     ;cin >> exam1
0009  330002          DECI    exam2,s     ;    >> exam2
000C  C30004          LDA     exam1,s     ;score = (exam1
000F  730002          ADDA    exam2,s     ;       + exam2)
0012  1E              ASRA                ;       / 2
0013  7C0005          ADDA    bonus,i     ;       + bonus
0016  E30000          STA     score,s
0019  410026          STRO    msg,d       ;cout << "score = "
001C  360000          DECO    score,s     ;       << score
001F  5C0004          CHARO   '\n',i      ;       << endl
0022  600006          ADDSP   6,i         ;deallocate @score @exam2 @exam1
0025  00              STOP
0026  73636F72 msg:   .ASCII  "score = \x00"
      653D20
      00
002F                  .END
```

图 6-4 （续）

main() 中可执行语句的翻译有两方面与全局变量不同。首先，SUBSP 和 ADDSP 在运行时栈上给局部变量分配和释放存储。其次，对变量的访问都使用栈相对寻址而不是直接寻址。除此之外，赋值和输出语句的翻译是一样的。



a) SUBSP执行前　　b) SUBSP执行后

图 6-5　图 6-4 中程序的运行时栈

图 6-4 展示怎样写局部变量的调试跟踪标签。汇编语言程序用 .EQUATE 伪操作使用格式跟踪标签 #2d，告诉调试器 exam1、exam2 和 score 应该显示为 2 字节的十进制值。用 SUBSP 指令为这些局部变量在运行时栈上分配存储空间，因此，为了调试程序，就要在 SUBSP 的注释中指定 3 个符号跟踪标签 #exam1、#exam2 和 #score。当单步跟踪程序时，Pep/8 系统会在屏幕上显示一个图标，就像图 6-5b 运行时栈右边的单元符号标签。为了调试器能够准确工作，注释字段中列出的符号跟踪标签的顺序必须与它们压入运行时栈的顺序一致。在这个程序中，exam1 首先压入运行时栈，然后是 exam2 和 score。此外，这个顺序必须和 .EQUATE 伪操作中的偏移量一致。

用 ADDSP 指令来释放变量。同样，列出变量的顺序必须与弹出运行时栈的顺序相对应。因为变量弹出的顺序与压入的顺序相反，所以必须以与 SUBSP 指令相反的顺序列出来。在这个程序中，score 首先弹出，接着是 exam2 和 exam1。

尽管程序的执行不需要跟踪标签，但是它们可以用于记录程序。符号跟踪标签提供的信息对程序的阅读者来说是非常有价值的，因为它描述了 SUBSP 和 ADDSP 指令的目的。本章的汇编语言程序都包括为了记录之用的跟踪标签，你写程序也应该这样写。

242 | 243

6.2　转移指令和控制流

Pep/8 指令集有 8 个条件分支语句：

- BRLE　　小于等于分支

- BRLT　小于分支
- BREQ　等于分支
- BRNE　不等于分支
- BRGE　大于等于分支
- BRGT　大于分支
- BRV　V分支
- BRC　C分支

每个条件分支检测4个状态位N、Z、V和C中的一个或者两个。如果条件为真，那么操作数放入PC，发生转移；如果条件为假，操作数不放入PC，条件分支后面的指令正常执行。可以把这想象成一个16位的结果和0000（hex）做比较。例如，BRLT检测结果是否小于0，如果N为1，则小于0；BRLE检测结果是否小于等于0，如果N为1或者Z为1，则小于等于0。每个条件分支指令的RTL描述如下：

```
BRLE    N = 1 V Z = 1 => PC ← Oprnd
BRLT    N = 1 => PC ← Oprnd
BREQ    Z = 1 => PC ← Oprnd
BRNE    Z = 0 => PC ← Oprnd
BRGE    N = 0 => PC ← Oprnd
BRGT    N = 0 V Z = 0 => PC ← Oprnd
BRV     V = 1 => PC ← Oprnd
BRC     C = 1 => PC ← Oprnd
```

转移是否发生取决于状态位的值，其他指令的执行也会影响状态位。例如，

```
LDA num,x
BRLT place
```

把num的内容装入累加器。如果装入的这个字是负数，即它的符号位为1，那么N位被置为1。BRLT检测N位并转移到位于place的指令。另一方面，如果装入累加器的不是负数，那么N位为0。当BRLT检测N位时，不会发生转移，接下来会执行BRLT后面的指令。

244

6.2.1　翻译if语句

图6-6展示了编译器怎样把if语句从C++翻译成汇编语言。这个程序计算一个整数的绝对值。

高级语言

```
#include <iostream>
using namespace std;

int main () {
   int number;
   cin >> number;
   if (number < 0) {
      number = -number;
   }
   cout << number;
   return 0;
```

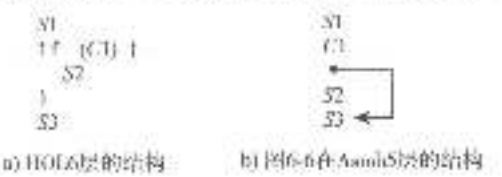
图6-6　HOL6层和Asmb5层的if语句

```
汇编语言
0000  040002          BR      main
              number: .EQUATE 0           ;local variable #2d
              ;
0003  580002 main:    SUBSP   2,i         ;allocate #number
0006  330000          DECI    number,s    ;cin >> number
0009  C30000 if:      LDA     number,s    ;if (number < 0)
000C  0C0016          BRGE    endif
000F  C30000          LDA     number,s    ;   number = -number
0012  18              NEGA
0013  F30000          STA     number,s
0016  360000 endif:   DECO    number,s    ;cout << number
0019  600002          ADDSP   2,i         ;deallocate #number
001C  00              STOP
001D                  .END
```

图 6-6 （续）

汇编语言注释显示对应的高级语言程序。cin 语句翻译为 DECI，cout 语句翻译为 DECO，赋值语句翻译为 LDA、NEGA、STA 序列。

编译器把 if 语句转换为 LDA、BRGE 序列。当执行 LDA 时，如果装入累加器的值是正数或零，那么 N 位为 0。这个条件会跳过 if 语句主体。图 6-7a 展示了 HOL6 层的 if 语句结构，*S1* 代表语句 cin>>number，*C1* 代表条件 number<0，*S2* 代表语句 number=-number，而 *S3* 代表语句 cout<<number。图 6-7b 展示了使用 Asmb5 层更原始的分支指令的结构。*C1* 下面的点代表条件转移 BRGE。



a) HOL6层的结构　　b) 图6-6在Asmb5层的结构

图 6-7 Asmb5 层的 if 语句结构

限定复合语句的花括号 { } 在汇编语言中没有对应的内容。下面的序列

```
语句1
if (number) >= 0{
    语句2
    语句3
}
语句4
```

翻译为

```
        语句1
        LDA number, d
if:     BRLT endif
        语句2
        语句3
endif:  语句4
```

6.2.2 优化编译器

你可能已经注意到图 6-6 有一个不是特别必要的装入语句。可以删除 000F 处的 LDA，因为前面在 0009 处装入的 number 的值仍然在累加器中。

问题是编译器该怎么做呢？答案取决于编译器。编译器是一个需要编写和调试的程序。假如需要设计一个把C++翻译到汇编语言的编译器。当编译器检测到赋值语句时，编程产生下面的序列：(a) 装入累加器；(b) 如果需要，对表达式求值；(c) 把结果存储到变量。这样编译器就会生成图6-6那样的代码，在000F有LDA语句。

可以想象如果要删除不必要的装入语句，对于编译器程序来说有多大的难度。当编译器检测到赋值语句时，它并不总是生成初始的装入语句，而是分析前面生成的语句，记住累加器的内容。如果它发现累加器的值和初始装入语句要放在那里的值一样，它就不生成初始装入语句。在图6-6中，编译器需要记住，根据if语句生成的代码，number的值仍然在累加器中。

245 / 246

一个尽力使目标程序更短更快的编译器叫作优化编译器。可以想象设计优化编译器比设计非优化编译器难很多。不仅优化编译器更难编写，而且也需要更长时间编译，这是因为这种编译器必须更仔细地分析源程序。

优化编译器和非优化编译器哪个更好呢？这取决于使用编译器来干什么。如果是开发软件，为了测试和调试一个过程需要大量的编译，那么就需要一个翻译很快的编译器，也就是非优化编译器。如果是一个大型固定的程序，这个程序会被许多用户重复执行很多遍，那么需要目标程序更快地执行，因此这时需要优化编译器。通常，正在开发和调试的软件使用非优化编译器，最后一次用优化编译器翻译给用户。

实际的编译器介于这两种极端情况之间。本章中的例子偶尔会呈现部分优化的目标代码，大多数赋值语句以非优化的形式呈现，例如图6-6中的赋值语句。

6.2.3　翻译 if/else 语句

图6-8说明了if/else语句的翻译。这个C++程序和图2-10的程序一样。if语句体需要一个额外的无条件分支语句绕过else语句体。如果编译器省略0015的BR，输入就会是127，而输出将会是highlow。

高级语言

```
#include <iostream>
using namespace std;

int main () {
   const int limit = 100;
   int num;
   cin >> num;
   if (num >= limit) {
      cout << "high";
   }
   else {
      cout << "low";
   }
   return 0;
}
```

汇编语言

```
0000  C40003          BR      main
              limit:  .EQUATE 100         ;constant
              num:    .EQUATE 0           ;local variable #2d
              ;
```

图6-8　HOL6和Asmb5层的if/else语句。这个C++程序来自图2-10

```
0003  680002 main:    SUBSP   2,i        ;allocate #num
0006  330000          DECI    num,s      ;cin >> num
0009  C30000 if:      LDA     num,s      ;if (num >= limit)
000C  800064          CPA     limit,i
000F  08001B          BRLT    else
0012  41001F          STRO    msg1,d     ;   cout << "high"
0015  04001B          BR      endIf      ;else
0018  410024 else:    STRO    msg2,d     ;   cout << "low"
001B  600002 endIf:   ADDSP   2,i        ;deallocate #num
001E  00              STOP
001F  686967 msg1:    .ASCII  "high\x00"
      6800
0024  606F77 msg2:    .ASCII  "low\x00"
      00
0028                  .END
```

图 6-8 （续）

和图 6-6 不一样的是图 6-8 的 if 语句不会将变量的值和零比较，它用 CPA 把变量值和另一个非零值进行比较，CPA 表示比较累加器。CPA 对累加器减去操作数并相应地设置 NZVC 位。除了 SUBr 把结果存储在寄存器 r（累加器或变址寄存器），而 CPr 忽略减法的结果外，CPr 和 SUBr 是一样的。CPr 的 RTL 描述是

T ← r−Oprnd; N ← T < 0, Z ← T = 0, V ← { 溢出 }, C ← { 进位 }

这里 T 表示临时值。

这个程序计算 num-limit 并设置 NZVC 位。BRLT 测试 N 位，如果

```
num - limit < 0
```

即如果

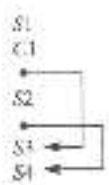
```
num < limit
```

就设置 N。这是执行 else 部分的条件。

图 6-9 展示了这两层上的控制语句结构。图 6-9a 是 HOL6 层的控制语句，图 6-9b 是这个程序到 Asmb5 层的翻译。

```
S1
if (C1) {
  S2
}
else
  S3
}
S4
```

a) HOL6层的结构



b) 图 6-8 在 Asmb5 层的结构

图 6-9　Asmb5 层的 if/else 语句结构

6.2.4　翻译 while 循环

循环需要转移到前面的指令。图 6-10 展示了 while 语句的翻译。这个 C++ 程序和图 2-13 的程序一样，它把输入的 ASCII 字符重复传送到输出，使用 * 作为标记符号，如果输入是 happy*，则输出 happy。

```
高级语言
#include <iostream>
using namespace std;

char letter;

int main () {
   cin >> letter;
   while (letter != '*') {
```

图 6-10　HOL6 和 Asmb5 层的 while 语句。这个 C++ 程序来自图 2-13

```
        cout << letter;
        cin >> letter;
    }
    return 0;
}
```

汇编语言

```
0000  040004          BR      main
0003  00      letter: .BLOCK  1          ;global variable #1c
              ;
0004  490003  main:   CHARI   letter,d   ;cin >> letter
0007  C00000          LDA     0x0000,i
000A  010003  while:  LDBYTEA letter,d   ;while (letter != '*')
000D  80002A          CPA     '*',i
0010  0A001C          BREQ    endWh
0013  510003          CHARO   letter,d   ;   cout << letter
0016  490003          CHARI   letter,d   ;   cin >> letter
0019  04000A          BR      while
001C  00      endWh:  STOP
001D                  .END
```

图 6-10 （续）

对 while 语句的测试是在循环顶部用条件转移实现的。这个程序检测一个字符值，它是 1 字节的量。0007 处的装入指令清除累加器中的两个字节，这样 000A 的装入字节指令执行后，最高有效字节将是 00（hex）。必须保证最高有效字节是 0，因为比较指令比较整个字。

每个 while 循环要以一个无条件分支结束，回到循环顶部的测试。0019 的分支把控制带回开始处的测试。图 6-11 展示了在两层上的 while 语句结构。



a) HOL6层的结构　　b) 图6-10在Asmb5层的结构

图 6-11　Asmb5 层的 while 语句结构

6.2.5　翻译 do 循环

高速巡警停在一个标识后面，司机以 20 米 / 秒超速通过，当这个司机沿着公路驶离 40 米后，巡警以 25 米 / 秒的速度去追赶超速违法者，巡警在离标识多远的地方能追上超速者？

图 6-12 的程序模拟解决这个问题，它和图 2-14 的程序一样。cop 和 driver 的值是两者的位置，初始值分别是 0 和 40。do 循环的每次执行表示时间过去 1 秒，在此期间巡警行进 25 米，超速者行进 20 米，一直持续到巡警追上超速者。

高级语言

```
#include <iostream>
using namespace std;

int cop;
int driver;

int main () {
   cop = 0;
   driver = 40;
```

图 6-12　HOL6 和 Asmb5 层的 do 语句。这个 C++ 程序来自图 2-14

```
    do {
        cop += 25;
        driver += 20;
    }
    while (cop < driver);
    cout << cop;
    return 0;
}
```

汇编语言

```
0000  C40037          BR      main
0003  0000   cop:     .BLOCK  2           ;global variable @2d
0005  0000   driver:  .BLOCK  2           ;global variable @2d
              ;
0007  C00000 main:    LDA     0,i         ;cop = 0
000A  E10003          STA     cop,d
000D  C00028          LDA     40,i        ;driver = 40
0010  E10005          STA     driver,d
0013  C10003 do:      LDA     cop,d       ;   cop += 25
0016  700019          ADDA    25,i
0019  E10003          STA     cop,d
001C  C10005          LDA     driver,d    ;   driver += 20
001F  700014          ADDA    20,i
0022  E10005          STA     driver,d
0025  C10003 while:   LDA     cop,d       ;while (cop < driver)
0028  B10005          CPA     driver,d
002B  0B0013          BRLT    do
002E  390003          DECO    cop,d       ;cout << cop
0031  00              STOP
0032                  .END
```

图 6-12 （续）

do 语句的测试在循环的底部。在这个程序中，编译器把 while 转换成 LDA、CPA、BRLT 序列。如果 N 位被置为 1，BRLT 执行转移。由于 CPA 计算两者之间的差异 cop-driver，所以如果

```
cop - driver < 0
```

即 cop < driver

N 位被置为 1。这是循环重复的条件。图 6-13 展示了在第 6 层和第 5 层的 do 语句结构。



图 6-13 Asmb5 层的 do 语句结构

6.2.6 翻译 for 循环

for 语句类似于 while 语句，两者的测试都是在循环的顶部，编译器必须生成代码来初始化和递增控制变量。图 6-14 展示了编译器怎样生成 for 语句代码，它把 for 语句翻译成下列的 Asmb5 层序列：

- 初始化控制变量。
- 测试控制变量。
- 执行循环体。
- 递增控制变量。

- 转移到测试

高级语言

```
#include <iostream>
using namespace std;

int main () {
   int j;
   for (j = 0; j < 3; j++) {
      cout << "j = " << j << endl;
   }
   cout << "j = " << j << endl;
   return 0;
}
```

汇编语言

```
0000  040003          BR      main
              j:      .EQUATE 0               ;local variable @0
              ;
0003  680002 main:    SUBSP   2,i             ;el acose @1
0006  C00000          LDA     0,i             ;for (j = 0
0009  C30000          STA     j,s
000C  B00003 for:     CPA     3,i             ;   j < 3
000F  0F0027          BRGE    endFor
0012  410034          STRO    msg,d           ;   cout << "j = "
0015  380000          DECO    j,s             ;       << j
0018  500004          CHARO   '\n',i          ;       << endl
001B  C30000          LDA     j,s             ;   j++)
001E  700001          ADDA    1,i
0021  E30000          STA     j,s
0024  040000          BR      for
0027  410034 endFor:  STRO    msg,d           ;cout << "j = "
002A  380000          DECO    j,s             ;   << j
002D  500004          CHARO   '\n',i          ;   << endl
0030  680002          ADDSP   2,i             ;deallocate @1
0033  00              STOP
0034  6A2030 msg:     .ASCII  "j = \x00"
      2000
0039                  .END
```

图 6-14　Asmb5 层的 for 语句结构

在这个程序中，CPA 计算 j - 3，如果 N 位为 0，即如果

```
j - 3 >= 0
```

也即

```
j >= 3
```

BRGE 转移出循环。

j 等于 0、1 和 2，循环体各执行一次，最后一次穿过循环，j 增加到 3，循环后面的输出语句要写这个值。

6.2.7　面条代码

在汇编层，程序员可以写出和 C++ 中并不对应的控制结构。图 6-15 展示了一个可能的

控制流，它在许多 HOL6 层语言中都不可能直接存在。检测条件 C1，如果为真，转移到一个测试条件为 C2 的循环中。C—— 中不能直接写出这样的控制流。

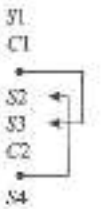


图 6-15　很多 HOL6 语言里都不可能直接有的控制流

编译器生成的汇编语言程序通常比人们直接用汇编语言写的程序长。不仅如此，执行也更慢。如果程序员能比编译器写出更短更快的汇编语言程序，那么为什么人们还要用高级语言编程呢？一个原因是如第 5 章讲到的编译器进行类型检查的能力，另一个原因是如果允许程序员自由使用原始转移指令，这将给他们增加额外的责任负担。如果在 Asmb5 层写程序时不够仔细，转移指令会失去控制，如下一个例子出现的情况。

图 6-16 的程序是一个极端的例子，无限制地使用原始转移指令就会产生这些问题。由于没有注释和缩进以及转移类型不一致，所以程序很难理解。实际上这个程序执行一个非常简单的任务，你能看出它是什么吗？

```
0000  040009          BR      main
0003  0000    n1:     .BLOCK  2
0005  0000    n2:     .BLOCK  2
0007  0000    n3:     .BLOCK  2
              ;
0009  310005 main:    DECI    n2,d
000C  310007          DECI    n3,d
000F  C10005          LDA     n2,d
0012  810007          CPA     n3,d
0015  080024          BRLT    L1
0018  310003          DECI    n1,d
001B  C10003          LDA     n1,d
001E  810007          CPA     n3,d
0021  080074          BRLT    L7
0024  040066          BR      L6
0027  :10007          STA     n3,d
002A  310003 L1:      DECI    n1,d
002D  C10005          LDA     n2,d
0030  510003          CPA     n1,d
0033  080053          BRLT    L5
0036  390002          DECO    n1,d
0039  390005          DECO    n2,d
003C  390007 L2:      DECO    n3,d
003F  00              STOP
0040  390005 L3:      DECO    n2,d
0043  390007          DECO    n3,d
0046  040081          BR      L9
0049  390003 L4:      DECO    n1,d
004C  390005          DECO    n2,d
004F  00              STOP
0050  E10003          STA     n1,d
0053  C10007 L5:      LDA     n3,d
0056  810003          CPA     n1,d
0059  080040          BRLT    L3
005C  390005          DECO    n2,d
005F  390003          DECO    n1,d
0062  040030          BR      L2
0065  390007 L6:      DECO    n3,d
```

图 6-16　一个神秘的程序

```
0060  C10003          LDA     n1,d
0063  810005          CPA     n2,d
0066  060049          BRLT    L4
0069  040077          BR      L8
006C  390003 L7:      DECO    n1,d
006F  390007          DECO    n3,d
0072  390005          DECO    n2,d
0075  00              STOP
0076  390005 L8:      DECO    n2,d
0079  390003 L9:      DECO    n1,d
007C  00              STOP
007D                  .END
```

图 6-16 （续）

C++ 中的 if 语句或者循环体是一个语句块，有时它包含在用花括号 {} 括起来的复合语句中。其他的 if 语句和循环可以完全嵌套在这些块中。图 6-17a 是这种情形的示意图。嵌套在 if/else、switch、while、do 和 for 语句中的控制流称为结构化控制流。

这个神秘程序中的分支不对应于 C++ 的结构化控制结构。尽管对于执行预期任务来说，这个程序的逻辑是正确的，但是由于分支语句到处转移，很难理解。这种程序叫作面条代码。如果从每条分支语句画一个到转移到语句的箭头，这张图看上去就像一碗面条，如图 6-17b 所示。



a) 结构化流　　b) 面条代码

图 6-17　两种不同风格的控制流

用非结构化分支有可能写出高效的程序，这样的程序比用结构化控制流的高级语言写的程序执行得更快，需要的内存更少。有些特殊的应用程序对效率有特别的要求，可以直接用汇编语言来编写。

理解平衡执行时间和内存占用是一件难事。当程序难理解时，它们也就难写，难调试，难修改。这是个经济问题，编写、调试、修改都是需要大量劳力的工作，也是很昂贵的。你一定会问提高的这点效率是不是值得花费更高的成本。

6.2.8　早期语言中的控制流

在结构化控制流出现前，计算机已经存在了很多年。在早期没有高级语言的时候，人人都用汇编语言编程。按今天的标准，那时计算机内存非常昂贵，CPU 速度很慢，效率是至关重要的。由于还没有产生大的软件，所以还没有意识到程序的维护问题。

第一个广泛应用的高级语言是 20 世纪 50 年代开发的 FORTRAN。因为那时的人们习惯使用分支指令，所以把它们也包括在这种语言中。FORTRAN 中的无条件分支是

```
GOTO 260
```

这里 260 是另一条语句的语句号，它称为 goto 语句。条件分支是

```
IF (NUMBER .GE. 100) GOTO 500
```

这里 .GE. 的意思是“大于或等于”，这条语句比较变量 NUMBER 和 100，如果大于或等于 100，那么下一条执行的语句就是编号 500 的语句，否则执行 IF 后面的那条语句。

相对于 Asmb5 层分支指令的 FORTRAN 分支语句 IF 是一个很大的改进，它不需要一

个单独的比较指令来设置状态位。但是可以看到这个控制流和 Asmb5 层的分支还是非常类似的：如果测试为真，执行 GOTO；否则，继续执行下一条指令。

随着开发出更多的软件，人们注意到把语句组成块，用在 if 语句和循环中会很方便。在这方面取得进步最著名的是 1960 年开发的 ALGOL-60，这是第一个广泛应用块结构的语言，尽管它主要在欧洲流行。 247 / 256

6.2.9　结构化编程定律

前几节展示了高级语言结构化控制流怎样被翻译为低层的原始分支语句，同时也展示了怎样在低层写出没有对应结构化结构的分支。这提出了一个有趣而实际的问题：可以用 goto 语句写出一些算法执行结构化结构不能执行的处理吗？也就是说，如果限定使用结构化控制流，会有一些问题你无法解决，而如果允许使用非结构化的 goto，这些问题就可以解决吗？

1966 年，Corrado Bohm 和 Giuseppe Jacopini 在一篇计算机科学杂志文章中回答了这个问题[⊖]。他们从数学上证明了任何含有 goto 语句的算法，不管多么复杂多么无结构，都可以用嵌套的 if 语句和 while 循环来编写。他们的结论称为结构化编程定律。

Bohm 和 Jacopini 的论文高度理论化，起初没有引起太多的关注，因为程序员通常不希望限制他们使用 goto 语句的自由。Bohm 和 Jacopini 展示了用 if 语句和 while 循环可以做什么，但是没有回答为什么程序员要这样限制自己。

不管怎样，人们开始尝试这个理念。他们会拿一个面条代码算法，试着用不带 goto 语句的结构化控制流来重写它。通常新写的程序比原来的更清晰，偶尔甚至更高效。

6.2.10　goto 争论

Bohm 和 Jacopini 的论文发表两年后，荷兰埃因霍温理工大学（Technological University at Eindhoven）的 Edsger W. Dijkstra 给同一家杂志的编辑写信，信中阐述了他个人的观察：优秀的程序员比差的程序员更少使用 goto 语句[⊜]。 257

Edsger Dijkstra

1930 年 Dijkstra 生于 Rotterdam 的一个荷兰化学家家庭，成长过程中一直对世界有形式主义的偏好。在荷兰莱顿大学（University of Leiden）学习时，Dijkstra 计划以物理学作为他的事业，但是他的父亲听说在英国剑桥有一个关于计算的夏季课程，Dijkstra 就在 1950 年计算刚刚开始兴起的时候投身其中。



Dijkstra 对编程最有名的贡献之一就是强烈拥护结构化编程定律，正如他那封有名的批评 goto 语句的书信明的那样。他素有直抒胸臆的名声，通常以大多数人无法忘怀的煽动性或戏剧性的方式来表达。例如，Dijkstra 曾经说过："使用 COBOL 会致人脑残，因此，就该 COBOL 应该被认定为犯罪。"他不止批评过一种语言，

⊖ Corrado Bohm and Giuseppe Jacopini, "Flow-Diagrams,Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM 9* (May 1966): 366-371.

⊜ Edsger W. Dijkstra, "Goto Statement Considered Harmful," *Communications of the ACM 11* (March 1968): 147-648. 此版得到许可。

他还说过："如果学生曾经接触过 BASIC 语言，基本上就不可能再教会他们好的编程方法了。因为作为可能的程序员，他们的智力已经受损，不可能再恢复了。"

除了在语言设计上的工作外，Dijkstra 还以他在程序正确性证明方面的工作闻名。程序正确性领域是把数学应用到计算机编程上。研究人员试图构造一种语言和证明技术，用于无条件证明程序会按照它的说明执行，完全没有错误。无需誓言，无论你的程序是客户记账还是飞行控制系统，能够证明程序具有这样的属性都是极具有价值的。

Dijkstra 实际上对计算机科学的每个领域都有所涉猎。他发明了信号量，本书第 8 章中会讲述到，还发明了一个解决最短路径问题的著名算法。1972 年，国际计算机学会（ACM）为了表彰 Dijkstra 对计算机领域的贡献，授予他图灵奖。在与癌症斗争了多年之后，Dijkstra 于 2002 年在荷兰 Nuenen 的家中去世。

"计算机是否能思考这个问题就像潜水艇是否会游泳一样。"

——Edsger Dijkstra

他的观点是，程序中大量使用 goto 语句表示质量很低。部分原文如下：

> 数年来，我越来越多地观察到程序员的质量与他们写出的程序中 goto 语句的密集程度成反比。最近，我发现了为什么 goto 语句的使用有极糟糕的影响，因此我倾向应该从所有高级编程语言（即任何语言，也许除了底层机器代码外）中废除 goto 语句……goto 语句所处的位置太原始，太容易把程序搞得一团糟。

为了证明这些观点，Dijkstra 开发了用一套坐标来描述程序进展的理念。当人们想理解一个程序时，他必须在心理上，或许是无意识地，维护这套坐标。Dijkstra 展示了用结构化控制流比用非结构化 goto 语句维护这些坐标要简单得多，因此他能精确指出结构化控制流更容易理解的原因。

Dijkstra 知道废除 goto 语句这个理念并不新鲜，他提到了几个在这个事情上影响他的几个人，其中之一是致力于 ALGOL-60 语言的 Niklaus Wirth。

Dijkstra 的信引起了一场反对风暴，这就是现在著名的 goto 争论。理论上可以不用 goto 编程是一回事，但主张从 FORTRAN 这样的高级语言中废除 goto 完全是另外一回事。

旧的观念很难消失，不管怎样争论逐渐平息了。现在一般认为 Dijkstra 实际上是正确的。原因是成本。当软件经理开始和其他结构设计概念一起运用结构化控制流准则时，他们发现这样的软件开发、调试和维护花费很低。这使得额外的内存需求和执行时间变得有价值。

FORTRAN 77 是在 1977 年制定的一个更新的 FORTRAN 版本，goto 争论影响到了它的设计。它的块风格、带 ELSE 部分的 IF 语句，类似于 C——，例如，

```
IF (NUMBER .GE. 100) THEN
   语句1
ELSE
   语句2
ENDIF
```

我们可以用 FORTRAN 77 编写没有 goto 的 IF 语句。

要记住，程序中不用 goto 不能保证程序有好的结构。在只需要 1 个或者 2 个嵌套时，有可能写出有 3 或 4 个 if 语句和 while 循环嵌套的程序。此外，如果任何层上的语言只包含 goto 语句来改变控制流，它也可以总是以结构化的方式来实现 if 语句和 while 循环。这正是 C—— 编译器把程序从 HOL6 层翻译到 Asmb5 层时做的事情。

6.3 函数调用和参数

C++ 函数调用把控制流改变到了函数的第一条可执行语句，在函数结束时，控制回到函数调用下面的语句。编译器用 CALL 指令实现函数调用，CALL 指令有在运行时栈存储返回地址的机制。它用 RETn 实现返回到调用语句，RETn 用保存在运行时栈的返回地址来确定接下来执行哪条指令。

258
259

6.3.1 翻译函数调用

图 6-18 展示了编译器如何翻译不带参数的函数调用，程序输出 3 个星号三角形。

```
高级语言
#include <iostream>
using namespace std;

void printTri () {
   cout << "*" << endl;
   cout << "**" << endl;
   cout << "***" << endl;
}

int main () {
   printTri ();
   printTri ();
   printTri ();
   return 0;
}

汇编语言

0000  04001F          SB      main
              ;
              ;******* void printTri ()
0003  410016 printTri:STRO    msg1,d     ;cout << "*"
0006  50000A          CHARO   '\n',i     ;   << endl
0009  410018          STRO    msg2,d     ;cout << "**"
000C  500004          CHARO   '\n',i     ;   << endl
000F  410018          STRO    msg3,d     ;cout << "***"
0012  50000A          CHARO   '\n',i     ;   << endl
0015  5B              RETO
0016  2A00   msg1:    .ASCII  "*\x00"
0018  2A2A00 msg2:    .ASCII  "**\x00"
0018  2A2A2A msg3:    .ASCII  "***\x00"
      00
              ;
              ;******* int main ()
001F  160003 main:    CALL    printTri   ;printTri ()
0022  160003          CALL    printTri   ;printTri ()
0025  160003          CALL    printTri   ;printTri ()
0028  00              STOP
0029                  .END
```

图 6-18 HOL6 层和 Asmb5 层的过程调用

CALL 指令把程序计数器的内容压入运行时栈，然后把操作数装入程序计数器。CALL 指

令的 RTL 描述是：

SP ← SP – 2; Mem[SP] ← PC; PC ← Oprnd

实际上，这个过程调用的返回地址被压入栈，然后执行一个到该程序的转移。

和分支指令一样，CALL 通常使用立即数寻址方式执行，这种情况下操作数就是操作数指示符。如果不指定寻址方式，Pep/8 汇编程序将假设使用立即数寻址。

图 5-2 展示了 RETn 指令，有一个 3 位 nnn 字段。通常情况下，一个程序可以有任意数量的局部变量。这里有 8 种 RETn 指令，名为 RET0、RET1、……、RET7，这里的 n 是程序局部变量占用的字节数。图 6-18 的 PrintTri 过程没有局部变量，因此编译器在 0015 生成 RET0。RETn 的 RTL 描述是：

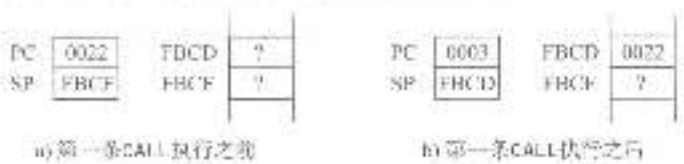
SP ← SP + n; PC ← Mem[SP]; SP ← SP – 2

首先，指令通过栈指针加 n 释放局部变量的存储。释放之后，返回地址就应该在运行时栈顶部了。然后，指令把返回地址从运行时栈顶部移到程序计数器。最后，给栈指针加 2，这是出栈操作。当然，程序可能有多于 7 字节的局部变量，这种情况下，编译器将生成 ADDSP 指令来释放局部变量的存储空间。

在图 6-18 中，

```
BR main
```

把 001F 放入程序计数器，因此下一条要执行的语句是在 001F 的语句，即第一条 CALL 指令。对图 6-1 中程序的讨论解释了栈指针怎样初始化为 FBCF。图 6-19 展示了执行第一条 CALL 语句之前和之后的运行时栈。当然，栈指针的初始值是 FBCF。



a) 第一条CALL执行之前　　b) 第一条CALL执行之后

图 6-19　图 6-18 中第一条 CALL 指令的执行

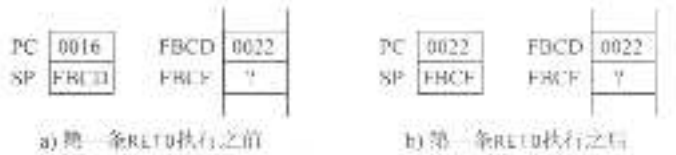
CALL 和 RETn 的运行严重依赖于冯·诺依曼运行周期：取指、译码、增加 PC、执行和重复，特别是增加 PC 发生在执行前，结果就是正在执行的语句并不是地址在程序计数器中的语句，而是程序计数器增加前取出的那条语句，现在它放在指令寄存器中。这个为什么在 CALL 和 RETn 的执行中这么重要呢？

图 6-19a 显示了在第一条 CALL 指令执行前，程序计数器的内容为 0022。这不是第一条 CALL 指令的地址，第一条 CALL 指令的地址 001F。为什么不是呢？因为程序计数器在执行 CALL 前增加到了 0022，因此在执行第一条 CALL 指令期间，程序计数器包含的正好是位于 CALL 指令后面那条指令在主存中的地址。

第一条 CALL 指令执行时发生了什么呢？首先，SP ← SP – 2 对 SP 减 2，得到值 FBCD。其次 Mem[SP] ← PC 把程序计数器的值 0022 放入主存中地址 FBCD 的位置，即运行时栈顶部。最后 PC ← Oprnd 把 0003 放入程序计数器。因为操作数指示符是 0003，寻址方式是立即数寻址，所以结果如图 6-19b 所示。

冯·诺依曼周期继续下一个取指。但是现在程序计数器中是 0003，因此下一个要获取的指令是地址 0003 的指令，这是 PrintTri 程序的第一条指令，执行程序的输出指令，生成一个星号三角图案。

最后，地址0015的RET0指令执行。图6-20a显示在执行RET0之前程序计数器的内容是0016，这可能看上去有点儿奇怪，因为0016甚至不是一条指令的地址，它是字符串"*\x00"的地址。为什么会这样呢？因为RET0是一个一元指令，CPU会把程序计数器加1。执行RET0的第一步是SP ← SP + n，由于n为0，所以SP加0。然后Mem[SP] ← PC把0022放入程序计数器。最后，SP ← SP + 2把栈指针改变回FBCF。



a) 第一条RET0执行之前　　b) 第一条RET0执行之后

图6-20 图6-18中第一条RET0指令的执行

马·诺依曼周期继续下一个取指，但这时程序计数器包含第二条CALL指令的地址，发生和第一个调用一样的事件序列，在输出流生成另一个星号三角图案。第三个调用也是一样的情况，这之后执行STOP指令。注意在STOP指令执行后，程序计数器的值是0029而不是0028，0028是STOP指令的地址。

现在你应该明白冯·诺依曼周期中为什么PC增加在执行之前了吧。要在运行时栈存储返回地址，CALL指令需要存储CALL后面那条指令的地址。只有CALL语句执行前程序计数器已经增加了才能做到这样。

6.3.2 用全局变量翻译传值调用参数

在C++中调用一个空函数时，分配过程是

- 压入实参。
- 压入返回地址。
- 压入局部变量的存储空间。

在HOL6层，在栈上执行这些操作的指令是隐藏的。程序员只写函数调用，执行时栈分配自动进行。

然而，在汇编层，翻译后的程序必须要有明确的指令来完成这些分配。图6-21的程序和图2-16的程序一样，是一个输出柱状图的HOL6层程序，以及它对应的Asmb5层翻译程序。图中显示了压入参数必需的Asmb5层语句，而在HOL6层是不需要明确写出来的。

```
高级语言
#include <iostream>
using namespace std;

int numPls;
int value;
int i;
void printBar (int n) {
   int k;
   for (k = 1; k <= n; k++) {
      cout << '*';
   }
```

图6-21 全局变量的传值调用参数

```
   cout << endl;
}

int main () {
   cin >> numPts;
   for (j = 1; j <= numPts; j++) {
      cin >> value;
      printBar (value);
   }
   return 0;
}
```

汇编代码

```
0000  040026          BR      main
0003  0000    numPts:  .BLOCK  2           ;global variable @2d
0005  0000    value:   .BLOCK  2           ;global variable @2d
0007  0000    j:       .BLOCK  2           ;global variable @2d
              ;
              ;******* void printBar (int n)
              n:       .EQUATE 4           ;formal parameter @2d
              k:       .EQUATE 0           ;local variable @2d
0009  580002 printBar:SUBSP   2,i         ;allocate #k
000C  C00001          LDA     1,i         ;for (k = 1
000F  E30000          STA     k,s
0012  630004 for1:    CPA     n,s         ;k <= n
0015  100027          BRGT    endfor1
0018  50002A          CHARO   '*',i       ;   cout << '*'
001B  C30000          LDA     k,s         ;k++)
001E  700001          ADDA    1,i
0021  E30000          STA     k,s
0024  040012          BR      for1
0027  50000A endFor1: CHARO   '\n',i      ;cout << endl
002A  54              RET2                ;deallocate @k, pop retAddr
              ;
              ;******* main ()
002B  310003 main:    DECI    numPts,d    ;cin >> numPts
002E  C00001          LDA     1,i         ;for (j = 1
0031  E10007          STA     j,d
0034  810003 for2:    CPA     numPts,d    ;j <= numPts
0037  100058          BRGT    endFor2
003A  310006          DECI    value,d     ;   cin >> value
003D  C10005          LDA     value,d     ;   call by value
0040  E3FFFE          STA     -2,s
0043  680002          SUBSP   2,i         ;   push #n
0046  160009          CALL    printBar    ;   push retAddr
0049  600002          ADDSP   2,i         ;   pop #n
004C  C10007          LDA     j,d         ;j++)
004F  700001          ADDA    1,i
0052  E10007          STA     j,d
0055  040034          BR      for2
0058  00      endFor2: STOP
0059                  .END
```

图 6-21　（续）

调用过程负责压入实参以及执行 CALL，CALL 把返回地址压入栈。被调用过程负责在栈上给局部变量分配存储空间。被调用的过程执行后，必须释放局部变量的存储空间，通过执行 RETn 弹出返回地址。在调用过程可以继续之前，一定要释放实参的存储空间。

总的来说，调用和被调用过程会完成如下操作：

- 调用压入实参（执行 SUBSP）。
- 调用压入返回地址（执行 CALL）。
- 被调用分配局部变量（执行 SUBSP）。
- 被调用执行它的函数体。
- 被调用释放局部变量，弹出返回地址（执行 RETn）。
- 调用弹出实参（执行 ADDSP）。

注意这些操作的对称性，后两个操作以相反的顺序执行前 3 个操作的逆操作，这个顺序是栈的后进先出特性导致的。

HOL6 层主程序的全局变量（numPts、value 和 j）对应同样的 Asmb5 层符号，符号值分别是 0003、0005 和 0007，这是保存全局变量运行时值的内存单元地址。图 6-22a 显示了全局变量，左边原来写地址的地方现在写的是符号名。全局变量的值是第一次执行 cin>>value 之后的值。

260 ⋮ 265

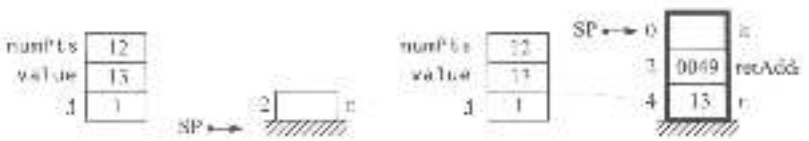


图 6-22 全局变量作为传值调用参数

形参 n、局部变量 k，在 Asmb5 层对应的是什么呢？不是绝对地址而是栈相对地址。过程 printBar 用

```
n: .EQUATE 4
k: .EQUATE 0
```

来定义它们。记住，.EQUATE 不会生成目标代码，翻译时汇编器不给它们保留存储空间，而是在运行时在栈上为 n 和 k 分配存储空间。十进制数 4 和 0 是过程执行期间 n 和 k 的栈偏移量，如图 6-22 所示。过程以栈相对寻址方式来引用它们。

调用过程中与过程调用相对应的语句为

```
LDA   value,d
STA   -2,s
SUBSP 2,i
CALL  printBar
ADDSP 2,i
```

因为参数是传值调用的全局变量，所以 LDA 用直接寻址，它把变量 value 的运行时值放入累加器，接着 STA 把它压入栈。偏移量是 -2，因为 value 是一个 2 字节的整数量，如图 6-22a 所示。

被调用过程中与过程调用相对应的语句为

```
SUBSP 2,i
.
```

```
;
;
RET7
```

266 由于局部变量k是2字节的整数量，所以SUBSP减去2。图6-22a展示了第一次输入全局变量value后、第一次过程调用前的运行时栈，它直接对应于图2-17d。图6-22b展示了过程调用后的栈，直接对应于图2-17g。注意在图2-17中标为ra1的返回地址，这里为0049，是跟在CALL后面那条指令的汇编语言地址。

n的栈地址是4，因为返回地址和k都占用2字节。如果有更多的局部变量，那么n的栈地址会相应地更大。编译器必须根据栈上数据的数量和大小来计算栈地址。

总之，要用全局变量翻译传值调用参数，编译器以如下方式生成代码：

- 压入实参，生成采用直接寻址方式的装入指令。
- 访问形参，生成采用栈相对寻址方式的指令。

6.3.3 用局部变量翻译传值调用参数

除了main()的变量是局部变量而不是全局变量之外，图6-23的程序和图6-21的程序是完全一样的。尽管这个程序的行为很像图6-21的程序，但是内存模型和到Asmb5层的翻译是不同的。

高级语言

```
#include <iostream>
using namespace std;

void printBar (int n) {
   int k;
   for (k = 1; k <= n; k++) {
      cout << '*';
   }
   cout << endl;
}

int main () {
   int numPts;
   int value;
   int j;
   cin >> numPts;
   for (j = 1; j <= numPts; j++) {
      cin >> value;
      printBar (value);
   }
   return 0;
}
```

汇编语言

```
00000  C40025          BR      main
                ;
                ;******* void printBar (int n)
                n:      .EQUATE 4           ;formal parameter @2d
                k:      .EQUATE 0           ;local variable @2d
```

图6-23　局部变量作为传值调用参数

```
0003  680002 printBar:SUBSP   2,i        ;allocate &k
0006  C00001           LDA     1,i       ;for (k = 1
0009  C50000           STA     k,s
000C  B50004 for1:     CPA     n,s       ;k <= n
000F  130021           BRGT    endfor1
0012  60002A           CHARO   '*',i     ;  cout << '*'
0015  C50000           LDA     k,s       ;k++)
0018  700001           ADDA    1,i
001B  E50000           STA     k,s
001E  040000           BR      for1
0021  60000A endfor1:  CHARO   '\n',i    ;cout << endl
0024  5A               RET2              ;deallocate &k, pc> retAddr
                ;
                ;******* main ()
                numPts: .EQUATE 4        ;local variable &3d
                value:  .EQUATE 2        ;local variable &2d
                j:      .EQUATE 0        ;local variable &2d
0025  680006 main:     SUBSP   6,i       ;allocate &numPts &value &j
0028  330004           DECI    numPts,s  ;cin >> numPts
002B  C00001           LDA     1,i       ;for (j = 1
002E  E50000           STA     j,s
0031  B50004 for2:     CPA     numPts,s  ;  <= numPts
0034  13003B           BRGT    endfor2
0037  330002           DECI    value,s   ;  cin >> value
003A  C50002           LDA     value,s   ;  call by value
003D  E3FFFE           STA     -2,s
0040  580002           SUBSP   2,i       ;  push &n
0043  160003           CALL    printBar  ;  push retAddr
0046  580002           ADDSP   2,i       ;  pop &n
0049  C50000           LDA     j,s       ;j++)
004C  700001           ADDA    1,i
004F  E50000           STA     j,s
0052  040031           BR      for2
0055  580006 endfor2:  ADDSP   6,i       ;deallocate &j &value &numPts
0058  00               STOP
0059                   .END
```

图 6-23 （续）

可以看到图 6-21 和图 6-23 中的返回值为 void（空）的函数 printTri 在 HOL6 层是一样。因此编译器为这两个版本生成一样的 Asmb5 层目标代码并不奇怪。两个程序唯一的不同是 main() 的定义。图 6-24a 展示了主程序中 numPts、value 和 j 在运行时栈上的分配。图 6-24b 展示了 printTri 第一次被调用后的运行时栈。因为 value 是一个局部变量，所以编译器生成使用栈相对寻址方式的 LDA value，s，将 value 的实际值压入形参 n 的栈单元。



a) cin>>value执行之后　　b) printBar中SUBSP分配之后

图 6-24 图 6-23 中 RETO 指令的第一次执行

总之，为了用局部变量翻译传值调用参数，编译器生成如下代码：

- 压入实参，生成使用栈相对寻址方式的装入指令。
- 访问形参，生成使用栈相对寻址方式的指令。

6.3.4　翻译非空函数调用

当调用函数时，分配过程是这样的：

- 压入返回值的存储空间。
- 压入实参。
- 压入返回地址。
- 压入局部变量的存储空间。

267
↓
269

非空（non-void）函数调用的分配不同于空函数调用的分配，必须为返回的函数值分配额外的空间。

图 6-25 展示了一个递归计算二项式系数的程序，和图 2-28 的程序一样。它是基于系数的帕斯卡三角，如图 2-27 所示。二项式系数的递归定义是

$$\begin{cases} b(n,0)=1 \\ b(k,k)=1 \\ b(n,k)=b(n-1,k)+b(n-1,k-1) \quad 对于\ 0<k<n \end{cases}$$

高级语言

```
#include <iostream>
using namespace std;

int binCoeff (int n, int k) {
   int y1, y2;
   if ((k == 0) || (n == k)) {
      return 1;
   }
   else {
      y1 = binCoeff (n - 1, k); // ra2
      y2 = binCoeff (n - 1, k - 1); // ra3
      return y1 + y2;
   }
}

int main () {
   cout << "binCoeff (3, 1) = " << binCoeff (3, 1); // ra1
   cout << endl;
   return 0;
}
```

汇编语言

```
00000  040065          BR      main
                ;
                ;******* int binomCoeff (int n, int k)
                retVal: .EQUATE 10          ;returned value @2d
                n:      .EQUATE 8           ;formal parameter @2d
                k:      .EQUATE 6           ;formal parameter @2d
                y1:     .EQUATE 2           ;local variable @2d
```

图 6-25　HOL5 层和 Asmb5 层的递归非空函数。这个 C++ 程序来自图 2-28

```
            y2:     .EQUATE 0           ;local variable #2d
0003  680004 binCoeff:SUBSP   4,i         ;allocate #y1 #y2
0006  C30006 if:      LDA     k,s         ;if (k == 0)
0009  0A0015          BREQ    then
000C  C30008          LDA     n,s         ;|| (n == k)
000F  830006          CPA     k,s
0012  0C001C          BRNE    else
0015  C00001 then:    LDA     1,i         ;return 1
0018  E3000A          STA     retVal,s
001B  5C              RET4                ;deallocate #y2 #y1, pop retAdd
001C  C30008 else:    LDA     n,s         ;push n - 1
001F  800001          SUBA    1,i
0022  E3FFFC          STA     -4,s
0025  C30006          LDA     k,s         ;push k
0028  E3FFFA          STA     -6,s
002B  680006          SUBSP   6,i         ;push #retVal #n #k
002E  160003          CALL    binCoeff    ;binomCoeff (n - 1, k)
0031  600006 ra2:     ADDSP   6,i         ;pop #k #n #retVal
0034  C3FFFE          LDA     -2,s        ;y1 = binomCoeff (n - 1, k)
0037  E30002          STA     y1,s
003A  C30008          LDA     n,s         ;push n - 1
003D  800001          SUBA    1,i
0040  E3FFFC          STA     -4,s
0043  C30006          LDA     k,s         ;push k - 1
0046  800001          SUBA    1,i
0049  E3FFFA          STA     -6,s
004C  680006          SUBSP   6,i         ;push #retVal #n #k
004F  160003          CALL    binCoeff    ;binomCoeff (n - 1, k - 1)
0052  600006 ra3:     ADDSP   6,i         ;pop #k #n #retVal
0055  C3FFFE          LDA     -2,s        ;y2 = binomCoeff (n - 1, k - 1)
0058  E30000          STA     y2,s
005B  C30002          LDA     y1,s        ;return y1 + y2
005E  730000          ADDA    y2,s
0061  E30004          STA     retVal,s
0064  5C        endIf:RET4                ;deallocate #y2 #y1, pop retAdd
                ;
                ;******* main ()
0065  410084 main:    STRO    msg,d       ;cout << "binCoeff (3, 1) = "
0068  C00003          LDA     3,i         ;push 3
006B  E3FFFC          STA     -4,s
006E  C00001          LDA     1,i         ;push 1
0071  E3FFFA          STA     -6,s
0074  680006          SUBSP   6,i         ;push #retVal #n #k
0077  160003          CALL    binCoeff    ;binomCoeff (3, 1)
007A  600006 ra1:     ADDSP   6,i         ;pop #k #n #retVal
007D  3BFFFE          DECO    -2,s        ;<< binCoeff (3, 1)
0080  500004          CHARO   '\n',i      ;cout << endl
0083  00              STOP
0084  62696E msg:     .ASCII  "binCoeff (3, 1) = \x00"
      ...
0097                  .END
```

图 6-25 （续）

函数使用 if 语句来测试基本的情况，测试条件使用了布尔运算符 OR。如果基本情况都不符合，它会递归调用它自己两次——一次计算 $b(n-1, k)$，一次计算 $b(n-1, k-1)$。图 6-26 展

示了主程序中一个以实参（3, 1）调用产生的运行时栈。函数接下来会被再调用两次，参数分别是（2, 1）和（1, 1），然后返回。再执行参数为（1, 0）的调用，接着是第二次返回，以此类推。图 6-26 展示了第二次返回后汇编层的运行时栈，它正好对应于图 2-29g 的 HOL6 层示意图。图 2-29g 中标号为 m2 的返回地址在图 6-26 中是 0031，这是函数中第一次 CALL 后面指令的地址。类似地，图 2-29 中标号为 m1 的地址在图 6-26 中是 007A。

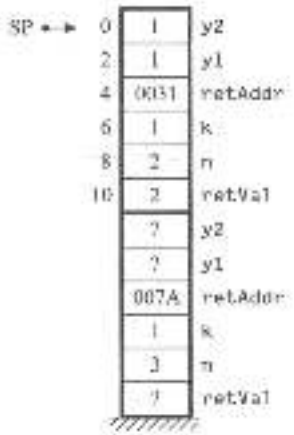


图 6-26　第二次返回后，图 6-25 的运行时栈

主程序开始时，栈指针是初始值，第一个实参的栈偏移量是 -4，第二个实参的是 -6。如果是在一个过程调用（返回值为空的函数）中，两个实参的偏移量分别是 -2 和 -4，它们比前面对应的偏移量大 2，因为这个函数的返回值会在运行时栈上占 2 字节。0074 的 SUBSP 指令分配 6 字节，两个实参每个 2 字节，返回值 2 字节。

当函数将控制返回到 007A 的 ADDSP 时，函数的返回值是栈上两个实参的下面。ADDSP 给栈指针加 6，弹出两个实参和返回值，于是指针指向返回值下面的那个单元。这样 DECO 用栈相对寻址方式输出这个值，使用偏移量 -2。

这个函数按照标准技术通过分配实参来调用它自己。对于第一次递归调用，它计算 n-1 和 k，把这两个值和返回值的存储空间一起压入栈。返回后，序列

```
ADDSP 6,i     ;pop @k @n @retVa
LDA   2,s     ;y1 = binomCoeff (n - 1, k)
STA   y1,s
```

270 … 272 弹出两个实参和返回值，将返回值赋给 y1。对于第二次调用，类似地把 n-1 和 k-1 压入，把返回值赋给 y2。

6.3.5　用全局变量翻译传引用调用参数

C++ 提供传引用调用参数，这样被调用过程可以改变调用过程中实参的值。图 2-20 展示了一个 HOL6 层的程序，使用传引用调用对两个全局变量 a 和 b 排序。图 6-27 给出了这个程序，以及编译器产生的目标程序。

高级语言

```
#include <iostream>
using namespace std;

int a, b;

void swap (int& r, int& s) {
   int temp;
   temp = r;
   r = s;
   s = temp;
}
void order (int& x, int& y) {
   if (x > y) {
      swap (x, y);
```

图 6-27　使用全局变量的传引用调用参数。C++ 程序来自图 2-20

```
   } // ra2
}

int main () {
   cout << "Enter an integer: ";
   cin >> a;
   cout << "Enter an integer: ";
   cin >> b;
   order (a, b);
   cout << "Ordered they are: " << a << ", " << b << endl; // ra1
   return 0;
}
```

汇编语言

```
0000  04C03C          BR      main
0003  0000    a:      .BLOCK  2           ;global variable @2d
0005  0000    b:      .BLOCK  2           ;global variable @2d
              ;
              ;******* void swap (int& r, int& s)
              r:      .EQUATE 6           ;formal parameter #2h
              s:      .EQUATE 4           ;formal parameter #2h
              temp:   .EQUATE 0           ; local variable @2d
0007  68000? swap:    SUBSP   2,i         ;allocate @temp
000A  C40006          LDA     r,sf        ;temp = r
000D  E30000          STA     temp,s
0010  C40004          LDA     s,sf        ;r = s
0013  E40006          STA     r,sf
0016  C30000          LDA     temp,s      ;s = temp
0019  E40004          STA     s,sf
001C  5A              RET2                ;deallocate #temp, pop retAddr
              ;
              ;******* void order (int& x, int& y)
              x:      .EQUATE 4           ;formal parameter @2h
              y:      .EQUATE 2           ;formal parameter @2h
001E  C40004 order:   LDA     x,sf        ;if (x > y)
0020  B40002          CPA     y,sf
0023  060038          BRLE    endIf
0026  C30004          LDA     x,s         ;   push x
0029  E3FFFE          STA     -2,s
002C  C30002          LDA     y,s         ;   push y
002F  E3FFFC          STA     -4,s
0032  680004          SUBSP   4,i         ;   push @r @s
0035  160007          CALL    swap        ;   swap (x, y)
0038  600004          ADDSP   4,i         ;   pop #s #r
003B  58      endIf:  RETO                ;pop retAddr

              ;
              ;******* main ()
003C  41006D main:    STRO    msg1,d      ;cout << "Enter an integer: "
003F  310003          DECI    a,d         ;cin >> a
0042  41006D          STRO    msg1,d      ;cout << "Enter an integer: "
0045  310005          DECI    b,d         ;cin >> b
0048  C00002          LDA     a,i         ;push the address of a
004B  E3FFFC          STA     -2,s
004E  C00005          LDA     b,i         ;push the address of b
```

图 6-27 （续）

```
0051  F3FFFC          STA     -4,s
0054  580004          SUBSP   4,i           ;push &x &y
0057  16001D          CALL    order         ;order (x, y)
005A  580004 tail:    ADDSP   4,i           ;pop &y &x
005D  410060          STRO    msg2,d        ;cout << "Ordered they are: "
0060  390003          DECO    a,d           ;     << a
0063  410093          STRO    msg3,d        ;     << ", "
0066  390005          DECO    b,d           ;     << b
0069  50000A          CHARO   '\n',i        ;     << endl
006C  00              STOP
006D  456E74 msg1:    .ASCII  "Enter an integer: \x00"
      ...
0080  4F7264 msg2:    .ASCII  "Ordered they are: \x00"
      ...
0093  2C2000 msg3:    .ASCII  ", \x00"
0096                  .END
```

图 6-27 （续）

主程序调用一个叫作 order 的过程，它的两个形参 x 和 y 是传引用调用。order 调用 swap，swap 完成实际的交换。swap 有两个传引用调用参数 r 和 s，参数 r 引用参数 s，参数 s 引用参数 a。程序员使用传引用调用，当过程 swap 改变 r 的时候，它实际上改变 a，因为 r 引用 a（通过 s）。

C++ 的传引用调用参数不同于传值调用参数，因为在调用过程中实参提供的是变量的引用而不是变量的值。在汇编层，把实参压入栈的代码会把实参的地址压入。当实参为全局变量时，它的地址就是它符号的值。这样压入全局变量地址的代码就是一个使用立即数寻址的装入指令。在图 6-27 中，压入 a 的地址的代码是

```
LDA a,i  ;push the address of a
```

符号 a 的值是 0003，这是 a 的值存储的地方。这个指令的机器代码是

```
C00003
```

C0 是装入累加器指令的指令指示符，寻址 -aaa 字段为 000 表示立即数寻址。使用立即数寻址方式，操作数指示符就是操作数，因此这条指令把 0003 装入累加器，后面的指令再把它压入运行时栈。

273 / 275

类似地，压入 b 的地址的代码是

```
LDA b,i  ;push the address of b
```

它的机器代码是

```
C00005
```

这里 0005 是 b 的地址。这条指令用立即数寻址把 0005 装入累加器，然后后面的指令把它压入运行时栈。

在图 6-27 中，在 0026，过程 order 调用 swap(x,y)。它必须把 x 压入运行时栈，x 是传引用调用，因此，运行时栈上是 x 的地址。相应的形参 r 也是传引用调用，因此过程 swap 希望 r 的地址在运行时栈上。过程 order 只是给 swap 传送地址供它使用。0026 的语句

```
LDA x,s  ;push x
```

用栈相对寻址把地址放入累加器，下一条指令把它放到运行时栈上。

然而，在过程 order 中，编译器必须翻译

```
temp = r
```

它必须把 r 的值装入累加器，然后把它存储在 temp 中。被调用的过程怎样访问地址在运行时栈上的形参值呢？使用栈相对间接寻址。

记住，栈相对寻址方式的操作数和操作数指示符之间的关系是

Oprnd = Mem[SP − OprndSpec]

操作数是运行时栈上，但是使用传引用调用参数，是操作数的地址在运行时栈上。栈相对间接寻址方式的操作数和操作数指示符之间的关系是

Oprnd = Mem[Mem[SP + OprndSpec]]

换句话说，Mem[SP + OprndSpec] 是操作数的地址，而不是操作数本身。

在 000A 和 000D 行，编译器为赋值语句的翻译生成下面的目标代码：

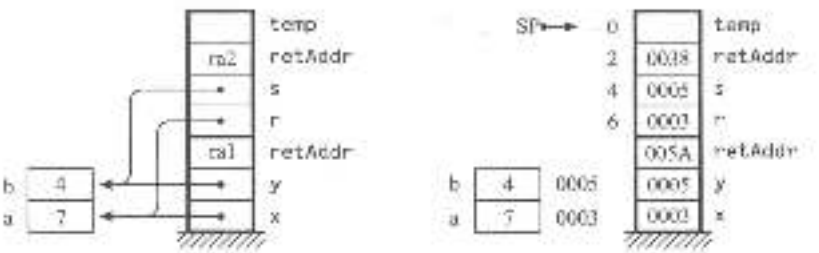
```
LDA r,sf
STA temp,s
```

装入指令中的字母 sf 表示栈相对间接寻址，装入指令的目标代码是

```
C40006
```

276

这里的 0006 是参数 r 的栈相对地址，如图 6-28b 所示，它里面的值是 0003，这是 a 的地址。装入指令把 a 的值 7 装入累加器，存储指令把它放入栈上的 temp 中。



a) HOL6层的运行时栈　　b) Asmb5层的运行时栈

图 6-28　图 6-27 在 HOL6 层和 Asmb5 层的运行时栈

过程 swap 的下一条赋值语句

```
r = s;
```

在赋值运算符两边都有参数。编译器生成 LDA 来装入 s 的值，生成 STA 来存储 r 的值，都是使用的栈相对寻址方式。

```
LDA s,sf
STA r,sf
```

总之，为了用全局变量翻译传引用调用参数，编译器生成如下代码：

- 压入实参，生成使用立即数寻址方式的装入指令。
- 访问形参，生成使用栈相对间接寻址方式的指令。

6.3.6　用局部变量翻译传引用调用参数

图 6-29 是一个计算给定长和宽的矩形周长的程序。主程序通过输入两个局部变量 width 和 height 给出长和宽，perim 是第三个局部变量。主程序调用一个名为 rect 的过

程（返回值为void的函数），通过传值调用传递width和height，通过传引用调用传递
277 perim。图中给出了当用户键入width为8，height为5时的输入和输出。

高级语言

```
#include <iostream>
using namespace std;

void rect (int& p, int w, int h) {
   p = (w + h) * 2;
}

int main () {
   int perim, width, height;
   cout << "Enter width: ";
   cin >> width;
   cout << "Enter height: ";
   cin >> height;
   rect (perim, width, height);
   // rel
   cout << "perim = " << perim << endl;
   return 0;
}
```

汇编语言

```
0000  040000          BR      main
                ;
                ;******* void rect (int& p, int w, int h)
                p:      .EQUATE 6               ;formal parameter #2h
                w:      .EQUATE 4               ;formal parameter #2d
                h:      .EQUATE 2               ;formal parameter #2d
0003  C30004  rect:   LDA     w,s             ;p = (w + h) * 2
0006  730002          ADDA    h,s
0009  1C              ASLA
000A  E40006          STA     p,sf
000D  50      endIf:  RETC                    ;pop retAddr
                ;
                ;******* main ()
                perim:  .EQUATE 4               ;local variable #2d
                width:  .EQUATE 2               ;local variable #2d
                height: .EQUATE 0               ;local variable #2d
000E  600006  main:   SUBSP   6,i             ;allocate #perim @width @height
0011  410046          STRO    msg1,d          ;cout << "Enter width: "
0014  330002          DECI    width,s         ;cin >> width
0017  410054          STRO    msg2,d          ;cout << "Enter height: "
001A  330000          DECI    height,s        ;cin >> height
001D  02              MOVSPA                  ;push the address of perim
001E  700004          ADDA    perim,i
0021  E3FFFE          STA     -2,s
0024  C30002          LDA     width,s         ;push the value of width
0027  E3FFFC          STA     -4,s
002A  C30000          LDA     height,s        ;push the value of height
002D  E3FFFA          STA     -6,s
0030  680006          SUBSP   6,i             ;push #p @w #h
0033  160003          CALL    rect            ;rect (perim, width, height)
0036  600006  rel:    ADDSP   6,i             ;pop #h @w #p
```

图 6-29　使用局部变量的传引用调用参数

```
0039  410063        STRO    msg3,d          ;cout << "perim = "
003C  380004        DECO    perim,s         ;      << perim
003F  60000A        CHARO   '\n',i          ;      << endl
0042  600000        ADDSP   6,i             ;deallocate &height &width &perim
0045  00            STOP
0046  456C74 msg1:  .ASCII  "Enter width: \x00"
      ...
0054  456E74 msg2:  .ASCII  "Enter height: \x00"
      ...
0063  706572 msg3:  .ASCII  "perim = \x00"
      ...
006C                .END

输入/输出
Enter width: 5
Enter height: 8
perim = 26
```

图 6-29 （续）

图 6-30 是程序在 HOL6 层的运行时栈。将此图和传引用调用全局变量的图 6-28a 进行比较，在那个程序里，形参 x、y、r 和 s 引用全局变量 a 和 b，在 Asmb5 层，a 和 b 在翻译时用 .EQUATE 点命令来分配，它们的符号就是它们的地址。然而，图 6-30 显示 perim 是在运行时栈上分配的。000E 处的语句

```
main: SUBSP 6,i
```

给 perim 分配存储空间，用

```
perim: .EQUATE 4
```

定义它的符号，该符号不是它的绝对地址。符号是相对于运行时栈顶部的地址，如图 6-31a 所示。符号的绝对地址是 FBCD，为什么呢？因为这是应用程序运行时栈的底部，如图 4-39 中的内存图所示。

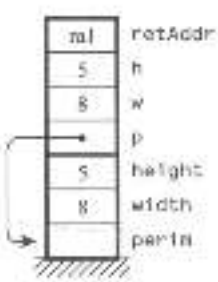


图 6-30 图 6-29 在 HOL6 层的运行时栈

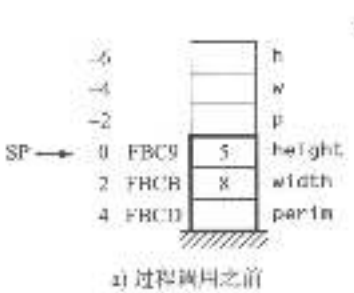


图 6-31 图 6-29 在 Asmb5 层的运行时栈

因此，编译器不能像对全局变量一样生成下面的代码把参数 perim 压入栈中：

```
LDA perim,i
STA -2,s
```

如果它生成那样的指令，过程 rect 就会修改 Mem[0004] 的内容，而 0004 并不是 perim 的位置。

perim 的绝对地址是 FBCD。图 6-31a 可以看到通过将栈指针值加上 perim 的值 4，就会得到这个绝对地址。幸运的是，一元指令 MOVSPA 可以把栈指针的内容移到累加器，

MOVSPA 的 RTL 描述是

A ← SP

为了压入 perim 的地址，编译器在图 6-29 中的 001D 生成如下的指令：

```
MOVSPA
ADDA   perim,1
STA    2,s
```

第一条指令把栈指针的内容移到累加器。那么累加器里就是 FBC9。第二条指令把 perim 的值 4 加到累加器，得到 FBCD。第三条指令把 perim 的地址放到 p 的单元中，过程 rect 把周长存放在这里。图 6-31b 显示执行的结果。

过程 rect 使用 p，就像所有的过程使用任何传引用调用的参数那样。也就是像在 000A 处，过程用栈相对间接寻址来存储值。

278 / 280

```
STA p,sf
```

使用栈相对间接寻址方式，操作数的地址在栈上。操作数是

Oprnd = Mem[Mem[SP + OprndSpec]]

这个指令把操作数指示符 6 加到栈指针 FBC1 上，得到 FBC7。因为 Mem[FBC7] 是 FBCD，所以把累加器的值存储在 Mem[FBCD]。

总之，为了用局部变量翻译传引用调用参数，编译器生成如下代码：

- 为了压入实参，生成一元 MOVSPA 指令，然后是立即数寻址的 ADDA 指令。
- 为了访问形参，生成使用栈相对间接寻址方式的指令。

6.3.7　翻译布尔类型

在汇编层存储布尔类型有多种方法。最适合 C++ 的方法是把值的 true/false（真/假）当作整数常量。这两个值是

```
const int true = 1;
const int false = 0;
```

图 6-32 的程序声明了一个布尔函数 inRange。编译器像上面 true 和 false 声明的那样翻译这个函数。

真实语言

```
#include <iostream>
using namespace std;

const int LOWER = 21;
const int UPPER = 65;

bool inRange (int x) {
   if ((LOWER <= x) && (x <= UPPER)) {
      return true;
   }
   else {
      return false;
   }
}
int main () {
```

图 6-32　布尔类型的翻译

```
   int age;
   cin >> age;
   if (inRange (age)) {
      cout << "Qualified\n";
   }
   else {
      cout << "Unqualified\n";
   }
   return 0;
}
```

汇编语言

```
0000  040023          BR      main
                true:   .EQUATE 1
                false:  .EQUATE 0
                ;
                LOWER:  .EQUATE 21              ;const int
                UPPER:  .EQUATE 55              ;const int
                ;
                ;******* bool inRange (int a)
                retVal: .EQUATE 4               ;returned value @2d
                a:      .EQUATE 2               ;formal parameter @2d
0003  C00015 inRange: LDA     LOWER,i           ;if (LOWER <= a)
0006  B30000 if:      CPA     a,s
0009  100010          BRGT    else
000C  C30000          LDA     a,s               ;   && (a <= UPPER))
000F  B00041          CPA     UPPER,i
0012  100010          BRGT    else
0015  C00001 then:    LDA     true,i            ;   return true
0018  E30002          STA     retVal,s
001B  50              RETO
001C  C00000 else:    LDA     false,i           ;   return false
001F  E30002          STA     retVal,s
0022  50              RETO
                ;
                ;******* main ()
                age:    .EQUATE 0               ;local variable @2d
0023  680002 main:    SUBSP   2,i               ;allocate @age
0026  330000          DECI    age,s             ;cin >> age
0029  C30000 if2:     LDA     age,s             ;if (
002C  E3FFFC          STA     -4,s              ;store the value of age
002F  680004          SUBSP   4,i               ;push @retVal #a
0032  160003          CALL    inRange           ;   inRange (age))
0035  680004          ADDSP   4,i               ;pop #a #retVal
0038  C3FFFE          LDA     -2,s              ;load retVal
003B  0A0044          BREQ    else2             ;branch if retval == false (i.e. 0)
003E  41004B then2:   STRO    msg1,d            ;   cout << "Qualified\n"
0041  040047          BR      endif2
0044  41005B else2:   STRO    msg2,d            ;   cout << "Unqualified\n"
0047  680002 endif2:  ADDSP   2,i               ;deallocate @age
004A  00              STOP
004B  51/61 msg1:     .ASCII  "Qualified\n\x00"
      ...
005B  55/6E msg2:     .ASCII  "Unqualified\n\x00"
      ...
0062                  .END
```

例6-32 (续)

在位层上把 false（假）和 true（真）表示为 0000 和 0001（hex）有优势也有劣势。考虑对布尔量进行逻辑运算和相应的汇编指令 ANDr、ORr 和 NOTr。如果 p 和 q 是全局布尔变量，那么

```
p && q
```

翻译为

```
LDA  p,d
ANDA q,d
```

如果用这个目标代码 AND 0000 和 0001，得到预期的 0000，OR 运算 || 也能得到预期的结果。然而，NOT 运算会有问题，因为如果对 0000 进行 NOT，得到 FFFF 而不是 0001。同样对 0001 进行 NOT，得到的是 FFFE 而不是 0000。因此，在翻译 C++ 赋值语句

281 ↓ 283

```
b = !a
```

时，编译器不会生成 NOT 指令，而是用异或运算 XOR，数学符号为⊕。它有一个非常有用的属性，如果对任意位值 b 和 0 进行 XOR，会得到 b，如果任意位值 b 和 1 进行 XOR，得到 b 的逻辑负值。数学表达式为

$$b \oplus 0 = b$$

$$b \oplus 1 = \neg b$$

不幸的是，Pep/8 计算机指令集中没有 XORr 指令。如果它有这样的指令，编译器将给上面的赋值生成如下的代码：

```
LDA  a,d
XORA 0xC001,i
STA  b,d
```

如果 q 为 false（假），表示 0000（hex），0000 XOR 0001 等于 0001，与预期一样。同样，如果 q 为 true（真），表示 0001（hex），0001 XOR 0001 等于 0000。

直到 1996 年，C++ 语言标准中都没有 bool 类型。老编译器使用的规则是布尔运算符对整数进行运算，把整数值 0 解释为 false（假），其他非零整数解释为 true（真）。为了保证向后兼容，现在的 C++ 编译器维持了这一规则。

6.4　变址寻址和数组

HOL6 层的变量在 ISA3 层是一个存储单元。HOL6 层的变量通过变量名来引用，在 ISA3 层是通过地址。在 Asmb5 层，通过符号名来引用变量，而符号的值是内存单元的地址。

数组的值又是怎样呢？数组包含许多元素，由许多内存单元组成，这些内存单元是连续的，相互邻接的。在 HOL6 层，数组有数组名。在 Asmb5 层，对应符号的地址是数组第一个内存单元的地址。本节说明编译器怎样翻译分配和访问一维数组元素的源程序，翻译使用了多种变址寻址的形式。

图 6-33 总结了 Pep/8 的所有寻址方式。我们在前面的程序中说明了立即、直接、栈相对和栈相对间接寻址。带有数组的程序使用变址、栈变址和栈变址相对寻址。标有 aaa 的那

284

一列展示了在 ISA3 层的寻址 -aaa 字段，标有字母那一列展示了在 Asmb5 层寻址方式的汇编语言名称，标有操作符那一列展示了 CPU 根据操作数指示符（OprndSpec）怎样确定操作数。

寻址模式	aaa	字　母	操　作　数
立即数	000	i	OprndSpec
直接	001	d	Mem [OprndSpec]
间接	010	n	Mem [Mem [OprndSpec]]
栈相对	011	s	Mem [SP + OprndSpec]
栈相对间接	100	sf	Mem [Mem [SP + OprndSpec]]
变址	101	x	Mem [OprndSpec + X]
栈变址	110	sx	Mem [SP + OprndSpec]
栈变址间接	111	sxf	Mem [Mem [SP + OprndSpec] + X]

图 6-33　Pep/8 的寻址模式

6.4.1　翻译全局数组

除了变量不是局部变量而是全局变量之外，图 6-34 的 C++ 程序和图 2-15 的程序是一样的。它给出了一个 HOL6 层程序，声明了有 4 个整数的全局数组 vector 和一个全局整数 j。主程序用一个 for 循环输入 4 个整数到数组，以逆序输出这 4 个整数和它们的索引。

高级语言

```
#include <iostream>
using namespace std;

int vector[4];
int j;

int main () {
   for (j = 0; j < 4; j++) {
      cin >> vector[j];
   }
   for (j = 3; j >= 0; j--) {
      cout << j  << ' ' << vector[j] << endl;
   }
   return 0;
}
```

汇编语言

```
0000  040000          BR      main
0003  000000 vector:  .BLOCK  8           ;global variable @2c4a
      000000
      0000
0008  0000   j:       .BLOCK  2           ;global variable @2c
             ;
             ;******* main ()
000E  C80000 main:    LDX     0,i         ;for (j = 0
0011  E90008          STX     j,d
0014  B80004 for1:    CPX     4,i         ;   j < 4
0017  CE0029          BRGE    endfor1
001A  1D              ASLX                ;   an integer is two bytes
001B  350003          DECI    vector,x    ;   cin >> vector[j]
001E  C90008          LDX     j,d         ;   j++)
0021  780031          ADDX    1,i
```

图 6-34　全局数组

```
0023  E90008          STX     j,d
0026  040013          BR      for1
0029  C90003 endFor1: LDX     3,i           ;for (i = 3
002C  E90008          STX     j,d
002F  580000 for2:    CPX     0,i           ;   j >= 0
0032  08004F          BRLT    endFor2
0035  390008          DECO    j,d           ;   cout << j
0038  500020          CHARO   ' ',i         ;      << ' '
003B  1D              ASLX                  ;   an integer is two bytes
003C  9D0003          DECO    vector,x      ;      << vector[j]
003F  50000A          CHARO   '\n',i        ;      << endl
0042  C90008          LDX     j,d           ;   j--)
0045  880001          SUBX    1,i
0048  E90008          STX     j,d
004B  040027          BR      for2
004E  00      endFor2: STOP
004F                   .END

输入

00 70 80 90

输出

3 90
2 80
1 70
0 60
```

图 6-34 （续）

图 6-35 展示了整数 j 和数组 vector 的内存分配。和所有的全局整数一样，编译器把 HOL6 层的

```
int j;
```

翻译为 Asmb5 层的语句：

```
j: .BLOCK 2
```

2 字节整数分配在地址 000B 处。编译器把 HOL6 层的

```
int vector[4];
```

翻译为 Asmb5 层的语句：

```
vector: .BLOCK 8
```

由于数组有 4 个整数，所以分配 8 字节，每个整数 2 字节。.BLOCK 语句的地址是 0003。从图 6-35 可以看到 0003 是数组第一个元素的地址，第二个元素的地址是 0005，每个元素的地址比前一个元素的地址大 2 字节。

编译器照常翻译第一个 for 语句

```
for (j = 0; j < 4; j++)
```

由于 j 是全局变量，所以用直接寻址访问 j。但是怎样访问 vector[j] 呢？不能只是使用直接寻址，因为符号 vector 的值是数组第一个元素的地址。如果 j 的值是 2，它应该访问数组的第三个元素，而不是第一个。

答案是使用变址寻址。使用变址寻址时，CPU 计算操作数为

Oprnd = Mem[OprndSpec + X]

它将操作数指示符和变址寄存器相加，把和作为主内存地址，然后从这个地址获取操作数。

在图 6-34 中，编译器将 HOL6 层的

```
cin >> vector[i];
```

翻译为 Asmb5 层的

```
ASLX
DECI vector,x
```

这是优化过的编译结果。编译器分析前面生成的代码，确定变址寄存器包含的是现在 j 的值。非优化编译器会生成下面的代码

```
LDX   j,d
ASLX
DECI  vector,x
```

285
}
287

假定 j 的值为 2，LDX 将 j 的值放入变址寄存器（或者，优化编译器确定现在的 j 值已经在变址寄存器里）。ASLX 将 2 乘以 2，把 4 放入变址寄存器。DECI 使用变址寻址。因此，这样计算操作数

Mem[OprndSpec + x]

Mem[0003 + 4]

Mem[0007]

它是图 6-35 中的 vector[2]。如果数组是字符数组，那么 ASLX 运算就不必要了，因为每个字符仅占 1 字节。一般来说，如果数组的每个单元占用 n 字节，那么 j 的值乘以 n 装入变址寄存器，用变址寻址访问数组元素。

类似地，编译器用变址寻址方式将 vector[j] 的输出翻译为

```
ASLX
DECO  vector,x
```

总之，为了翻译全局数组，编译器遵循如下规则生成代码：

- 用 .BLOCK tot 给数组分配存储空间，tot 是数组占用的总字节数。
- 通过把索引乘以每个单元的字节数装入变址寄存器来访问数组元素，使用变址寻址。

格式跟踪标签说明数组有多少个单元以及多少字节。在图 6-34 中的 0003，vector 的声明为

```
vector:  .BLOCK 8  ;global variable #2d4a
```

应该把格式跟踪标签 #2d4a 读作“两字节十进制，4 单元数组”。使用这种说明，Pep/8 调试器将生成类似于图 6-35 那样的每个单元都有标记的数组示意图。

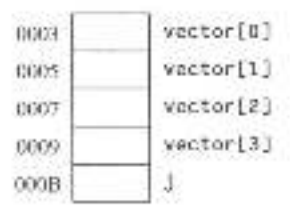


图 6-35　图 6-34 所示全局数据的内存分配

6.4.2　翻译局部数组

就像所有的局部变量一样，局部数组在程序执行期间在运行时栈上进行分配。SUBSP 指令给数组分配空间，ADDSP 指令释放空间。除了索引 j 和数组 vector 是 main() 的局部变量外，图 6-36 的程序和图 6-34 的程序是一样的。

图 6-37 展示了图 6-36 中的程序在运行时栈上的内存分配。编译器把 HOL6 层的

高级语言

```
#include <iostream>
using namespace std;

int main () {
   int vector[4];
   int j;
   for (j = 0; j < 4; j++) {
      cin >> vector[j];
   }
   for (j = 3; j >= 0; j--) {
      cout << j  << ' ' << vector[j] << endl;
   }
   return 0;
}
```

汇编语言

```
0000  040003          BR      main
                ;
                ;******* main ()
                vector: .EQUATE 2               ;local variable &2d4a
                j:      .EQUATE 0               ;local variable &2d
0003  580008 main:    SUBSP   10,i            ;allocate &vector &j
0006  C80000          LDX     0,i             ;for (j = 0
0009  E80000          STX     j,s
000C  880004 for1:    CPX     4,i             ;   j < 4
000F  0FC022          BRGE    endFor1
0012  18              ASLX                    ;   an integer is two bytes
0013  380002          DECI    vector,sx       ;   cin >> vector[j]
0016  C80000          LDX     j,s             ;   j++)
0019  780001          ADDX    1,i
001C  E80000          STX     j,s
001F  040006          BR      for1
0022  C80003 endFor1: LDX     3,i             ;for (j = 3
0025  E80000          STX     j,s
0028  880000 for2:    CPX     0,i             ;   j >= 0
002B  090047          BRLT    endFor2
002E  300000          DECO    j,s             ;   cout << j
0031  500020          CHARO   ' ',i           ;      << ' '
0034  18              ASLX                    ;   an integer is two bytes
0035  3F0002          DECO    vector,sx       ;      << vector[j]
0038  500A0A          CHARO   '\n',i          ;      << endl
003B  C80000          LDX     j,s             ;   j--)
003E  880001          SUBX    1,i
0041  E80000          STX     j,s
0044  04002B          BR      for2
0047  60000A endFor2: ADDSP   10,i            ;deallocate &j &vector
004A  00              STOP
004B                  .END
```

图 6-36　局部数组。这个 C++ 程序来自于图 2-15

```
int vector[4];
int j;
```

翻译成 Asmb5 层的

```
main: SUBSP 10,i
```

给 vector 分配 8 字节，给 j 分配 2 字节，总共 10 字节。用

```
vector: .EQUATE 2
j:      .EQUATE 0
```

设置符号的值，这里 2 是 vector 第一个单元的栈相对地址，0 是 j 的栈相对地址，如图 6-37 所示。

编译器是怎样访问 vector[j] 的呢？不能使用变址寻址，因为符号 vector 的值不是数组第一个元素的地址。需要使用栈变址寻址。使用栈变址寻址，CPU 这样计算操作数

Oprnd = Mem[SP + OprndSpec − X]

将栈指针、操作数指示符和变址寄存器相加，把这个和作为它从主内存获取操作数的地址。

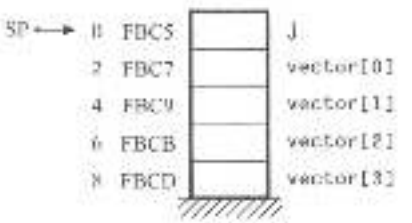


图 6-37 图 6-36 中局部数组的内存分配

在图 6-37 中，编译器把 HOL6 层的

```
cin >> vector[1];
```

翻译成 Asmb5 层的

```
ASLX
DECI  vector,sx
```

和前面的程序一样，这是一个优化的翻译。非优化编译器将会生成下面的代码：

```
LDX   j,d
ASLX
DECI  vector,sx
```

288 : 290

假定 j 的值为 2，LDX 把 j 的值放入变址寄存器。ASLX 把 2 乘以 2，变址寄存器中的内容变为 4。DECI 使用栈变址寻址。这样，像下面这样计算操作数

Mem[SP + OprndSpec − X]

Mem[FBC5 + 2 + 4]

Mem[FBCB]

它是图 6-37 中的 vector[2]。我们可以看到栈变址寻址是如何设计来为运行时栈上的数组服务的。SP 是栈顶地址，OprndSpec 是数组第一个单元的栈相对地址，因此 SP − OprndSpec 就是数组第一个单元的绝对地址。变址寄存器里存放的是 j（乘以数组每个单元的字节数），因此 SP + OprndSpec − X 的和是数组单元 j 的地址。

总之，为了翻译局部数组，编译器遵照如下原则生成代码：

- 用 SUBSP 对数组进行分配，用 ADDSP 释放。
- 通过把索引装入变址寄存器来访问数组元素，把索引乘以每个单元的字节数，使用栈变址寻址。

6.4.3 翻译作为参数传递的数组

在 C++ 中，数组名是数组第一个元素的地址。当要传递一个数组时，即使在形参表中不用 & 指明，也是传递的数组第一个元素的地址。这个效果就与传引用调用数组一样。C 语言的设计者认为程序员几乎从来不会想去传值调用一个数组，因为这样的调用效率太低。由于栈必须包含整个数组，所以要求运行时栈必须有大量的存储空间。还会需要大量的时间，因为每个单元的值要复制到栈中。因此，C++ 中对数组默认的是传引用调用。

图 6-38 展示了编译器怎样翻译一个把局部数组作为参数传送的程序。主程序传递整数数组 vector 和整数 numItms 到程序 getVect 和 putVect，getVect 把值输入到数组并

把 numItms 设置为输入条目的数量值，putVect 输出数组的值。

高级语言

```
#include <iostream>
using namespace std;

void getVect (int v[], int& n) {
   int j;
   cin >> n;
   for (j = 0; j < n; j++) {
      cin >> v[j];
   }
}

void putVect (int v[], int n) {
   int j;
   for (j = 0; j < n; j++) {
      cout << v[j] << ' ';
   }
   cout << endl;
}

int main () {
   int vector[6];
   int numItms;
   getVect (vector, numItms);
   putVect (vector, numItms);
   return 0;
}
```

汇编语言

```
0000  04004C          BR      main
            ;
            ;******* getVect (int v[], int& n)
            v:        .EQUATE 6             ;formal parameter @2n
            n:        .EQUATE 4             ;formal parameter @2b
            j:        .EQUATE 0             ;local variable @2d
0003  680002 getVect: SUBSP   2,i           ;allocate #j
0006  340004          DECI    n,sf          ;cin >> n
0009  C00000          LDX     0,i           ;for (j = 0
000C  E80000          STX     j,s
000F  B00004 for1:    CPX     n,sf          ;   j < n
0012  DE0025          BRGE    endFor1
0015  10              ASLX                  ;   an integer is two bytes
0016  370006          DECI    v,sxf         ;   cin >> v[j]
0019  C80000          LDX     j,s           ;   j++)
001C  780001          ADDX    1,i
001F  E80000          STX     j,s
0022  04000F          BR      for1
0025  5A      endFor1: RET2                 ;pop #j and retAddr
            ;
            ;******* putVect (int v[], int n)
            v2:       .EQUATE 6             ;formal parameter @2n
            n2:       .EQUATE 4             ;formal parameter @2c
            j2:       .EQUATE 0             ;local variable @2d
```

图 6-38　将局部数组作为参数传递

```
0026  680002 putVect: SUBSP   2,i          ;allocate @j2
0029  C80000          LDX     0,i          ;for (j = 0
002C  F80000          STX     j2,s
002F  B80004 for2:    CPX     n2,s         ;   j < n
0032  C1004B          BRGE    endFor2
0035  10              ASLX                 ;   an integer is two bytes
0036  3F0006          DECO    v2,sxf       ;   cout << v[j]
0039  500020          CHARO   ' ',i        ;      << ' '
003C  C80000          LDX     j2,s         ;   j++)
003F  780001          ADDX    1,i
0042  E80000          STX     j2,s
0045  04002F          BR      for2
0048  50000A endFor2: CHARO   '\n',i       ;cout << endl
004B  5A              RET2                 ;pop @j2 and retAddr
                ;
                ;******* main ()
                vector:  .EQUATE 2         ;local variable @2dBs
                numItms: .EQUATE 0         ;local variable @2d
004C  680012 main:    SUBSP   18,i         ;allocate @vector @numItms
004F  02              MOVSPA               ;push address of vector
0050  700002          ADDA    vector,i
0053  E3FFFF          STA     -2,s
0056  02              MOVSPA               ;push address of numItms
0057  700000          ADDA    numItms,i
005A  E3FFFC          STA     -4,s
005D  680004          SUBSP   4,i          ;push @v @n
0060  160003          CALL    getVect      ;getVect (vector, numItms)
0063  600004          ADDSP   4,i          ;pop @n @v
0066  02              MOVSPA               ;push address of vector
0067  700002          ADDA    vector,i
006A  E3FFFE          STA     -2,s
006D  C30000          LDA     numItms,s    ;push value of numItms
0070  E3FFFC          STA     -4,s
0073  680004          SUBSP   4,i          ;push #v2 @n2
0076  160026          CALL    putVect      ;putVect (vector, numItms)
0079  600004          ADDSP   4,i          ;pop @n2 #v2
007C  600012          ADDSP   18,i         ;deallocate #numItms @vector
007F  00              STOP
0080                  .END
```

输入

```
5  40 50 60 70 80
```

输出

```
40 50 60 70 80
```

图 6-38 （续）

如图 6-38 所示，编译器将局部变量

```
int vector[5];
int numItms;
```

翻译成

```
vector:  .EQUATE 2
numItms: .EQUATE 0
main:    SUBSP 10,i
```

SUBSP 指令在运行时栈上分配 18 字节，16 字节用于数组的 8 个整数，2 字节用于整数。点命令 .EQUATE 把符号设置成它们对应的栈偏移量，如图 6-39a 所示。

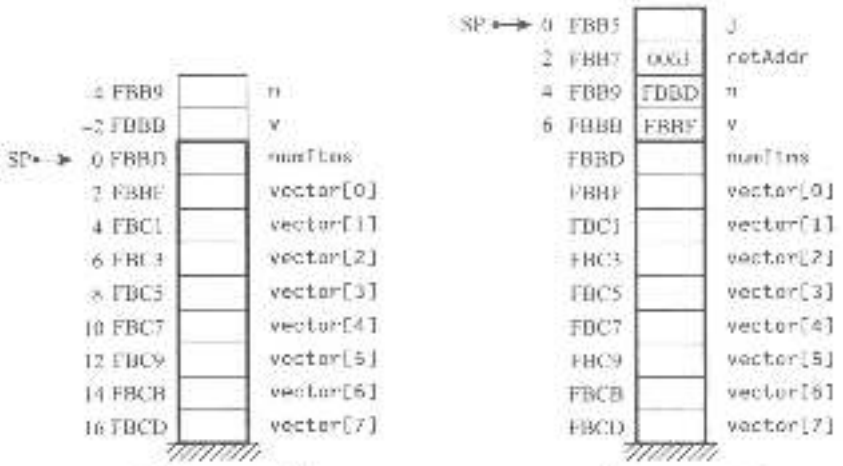


图 6-39　图 6-38 所示程序的运行时栈

291 / 294

编译器要翻译

```
getVect (vector, numItms);
```

首先生成把 vector 的第一个单元的地址压入的代码

```
MOVSPA
ADDA    vector,i
STA     -2,s
```

接着生成压入 numItms 地址的代码

```
MOVSPA
ADDA    numItms,i
STA     -4,s
```

尽管函数

```
void getVect (int v[], int& n)
```

的参数 v[] 没有用 &，但是编译器会生成代码，用 MOVSPA 和 ADDA 指令压入 v 的地址。参数 n 使用了 &，编译器会用同样的方式生成压入 n 的地址的代码。图 6-39b 说明 v 等于 FBBF，即 vector[0] 的地址；n 是 FBBD，即 numItms 的地址。

图 6-39b 也展示了 getVect 中参数和局部变量的栈偏移量，编译器相应地定义符号：

```
v: .EQUATE 6
n: .EQUATE 4
j: .EQUATE 0
```

把输入语句

```
cin >> n;
```

翻译为

```
DECI n,sf
```

这里使用栈相对间接寻址，因为 n 是传引用调用的，n 的地址在栈上。

但是编译器怎样翻译

```
cin >> v[j];
```

呢？不能使用栈变址寻址方式，因为该数组的值不在 getVect 的栈帧中。v 的值为 6，这表示数组第一个单元的地址在栈顶下面 6 字节的位置。该数组的值在 main() 的栈帧中。栈变址间接寻址就是设计用来访问这样的数组元素，数组的地址在顶部的栈帧中，但是实际的值并不在。CPU 用栈变址相对寻址来这样计算操作数：

Oprnd = Mem[Mem[SP + OprndSpec] + X]

栈指针和操作数指示符相加，把这个和作为数组第一个元素的地址，再把这个地址和变址寄存器相加。编译器把输入语句翻译成

```
ASLX
DECI v,sxf
```

这里 sxf 表示是栈变址间接寻址，编译器已经确定变址寄存器里包含 j 的当前值。

例如，假定 j 的值是 2，ASLX 指令把它乘以 2 得到 4，操作数的计算如下

Mem[Mem[SP + OprndSpec] + X]

Mem[Mem[FBB5 + 6] + 4]

Mem[Mem[FBBB] + 4]

Mem[FBBF + 4]

Mem[FBC3]

它是 vector[2]，和图 6-39b 预测的一样。

在图 6-39 中过程 getVect 和 putVect 形参的名字是一样的。在 HOL6 层，参数名的范围，限制在函数体内。程序员知道在 getVect 体中包含 n 的语句引用的是 getVect 参数列表中的 n，而不是 putVect 参数列表中的 n。然而，在 Asmb5 层，符号名的范围却是整个汇编语言程序。编译器不能给 putVect 中的 n 使用与 getVect 中 n 一样的符号，因为重复的符号定义会引起歧义。所有的编译器在将 HOL6 层的名字声明翻译到 Asmb5 层的符号时，必须有某种用来管理这些名字声明范围的机制。图 6-38 中的编译器通过给符号名附加数字 2 形成无二义性的标识符。因此编译器将 HOL6 层的 putVect 中的变量名 n 翻译成 Asmb5 层的符号 n2，对 v 和 j 做同样的处理。

在过程 putVect 中，数组作为参数传递，但 n 是传值调用。在准备过程调用时，和前面一样，将 vector 的地址压入栈，但这次是将 numItms 的值压入栈。在过程 putVect 中，使用栈相对寻址来访问 n2

```
for2: LDA n2,s
```

只是因为 n2 是传值调用，使用栈变址间接寻址来访问 v2

```
ASLX
DECO v2,sxf
```

就像它在 getVect 中一样。

在图 6-38 中，vector 是局部数组。如果它是全局数组，那么对 getVect 和 putVect 的翻译不会改变。使用栈变址间接寻址来访问 v[j]，它预期数组的第一个元素的地址在顶部的栈帧中。唯一的不同的是在准备调用时压入数组第一个元素地址的代码，与图 6-34 中的程序一样，全局数组的符号值是数组第一个单元的地址。因此要压入数组第一个单元的地址，编译器会生成一条使用立即数寻址的 LDA 指令，后面跟一个栈相对寻址的 STA 指令来

295

296

进行压入。

总之，为了把数组作为参数传递，编译器遵循如下原则生成代码：

- 将数组的第一个元素的地址压入运行时栈，或者（a）对于局部数组，用 MOVSPA，后面跟采用立即数寻址的 ADDA；或者（b）对于全局数组，使用立即数寻址的 LDA。
- 通过把索引装入变址寄存器来访问数组元素，将索引乘以每个单元的字节数，使用栈变址间接寻址。

6.4.4　翻译 switch 语句

图 6-40 中的程序就是图 2-12 中的程序，它展示了编译器怎样翻译 C++ 的 switch 语句。这个程序使用了一个有趣的变址寻址和无条件转移 BR 的结合。switch 语句和嵌套的 if 语句是不一样的。如果用户给 guess 输入 2，switch 语句不会将 guess 与 0 和 1 比较，而是直接转移到第三个选择。由于索引机制允许程序员不用遍历所有前面的元素而随机访问任意元素，所以数组是一个随机访问的数据结构。例如，要访问一个整数矢量的第三个元素，可以直接写 vector[2] 而不用先遍历 vector[0] 和 vector[1]。主存实际上是 1 字节数组，字节的地址就对应于这个数组的索引。为了翻译 switch 语句，编译器分配一个叫作转移表（jump table）的地址数组。转移表中的每个条目是一段代码第一条语句的地址，每段代码对应 switch 语句的一种情况。使用变址寻址，程序能够直接转到情况 2（case 2）。

```
高级语言
#include <iostream>
using namespace std;

int main () {
   int guess;
   cout << "Pick a number 0..3: ";
   cin >> guess;
   switch (guess) {
      case 0: cout << "Not close"; break;
      case 1: cout << "Close"; break;
      case 2: cout << "Right on"; break;
      case 3: cout << "Too high";
   }
   cout << endl;
   return 0;
}

汇编语言
0000  040002          BR      main

                ;******* main ()
                guess:  .EQUATE 0           ;local variable #2d
0003  680002  main:   SUBSP   2,i           ;allocate local &guess
0006  410037          STRO    msgIn,d       ;cout << "Pick a number 0..3: "
0009  330000          DECI    guess,s       ;cin >> guess
000C  C80000          LDX     guess,s       ;switch (guess)
000F  10              ASLX                  ;addresses occupy two bytes
0010  050013          BR      guessBT,x
```

图 6-40　switch 语句的翻译。这个 C++ 程序来自图 2-12

```
0013 0013  guessJT: .ADDRSS case0
0015 0021           .ADDRSS case1
0017 0027           .ADDRSS case2
0019 0029           .ADDRSS case3
001B 410040 case0:  STRO    msg0,d      ;cout << "Not close"
001E 040030         BR      endCase     ;break
0021 410056 case1:  STRO    msg1,d      ;cout << "Close"
0024 040030         BR      endCase     ;break
0027 410050 case2:  STRO    msg2,d      ;cout << "Right on"
002A 040030         BR      endCase     ;break
002D 410065 case3:  STRO    msg3,d      ;cout << "Too high"
0030 500004 endCase: CHARO  '\n',i      ;cout << endl
0033 600002         ADDSP   2,i         ;deallocate guess
0035 00             STOP
0037 506963 msgPr:  .ASCII  "Pick a number 0..3: \x00"
     ...
0040 4e6f74 msg0:   .ASCII  "Not close\x00"
     ...
0056 436c6f msg1:   .ASCII  "Close\x00"
     ...
0050 526967 msg2:   .ASCII  "Right on\x00"
     ...
0065 546f6f msg3:   .ASCII  "Too high\x00"
     ...
006E                .END
```

图 6-40 （续）

图 6-40 展示了汇编语言程序中在 0013 处的转移表。在 0013 生成的代码是 001B，这是情况 0（case 0）第一条语句的地址，在 0015 生成的代码是 0021，这是情况 1（case 1）第一条语句的地址，以此类推。编译器用 .ADDRSS 伪操作生成转移表，每个 .ADDRSS 命令后面必须跟一个符号，.ADDRSS 生成的代码是符号的值。例如，case2 是一个符号，它的值是 0027，这是如果 guess 的值为 2 时要执行代码的地址，因此，

```
.ADDRSS case2
```

在 0017 处生成的目标代码是 0027。

假定用户为 guess 的值输入 2，语句

```
LDX guess,s
```

把 2 放入变址寄存器，语句

```
ASLX
```

将 2 乘以 2，寄存器中为 4，语句

```
BR guessJT,x
```

是变址寻址的无条件转移。操作数指示符 guessJT 的值是 0013，这是转移表第一个字的地址。对于变址寻址，CPU 这样计算操作数

Oprnd = Mem[OprndSpec − X]

因此 CPU 计算

Mem[OprndSpec + X]

Mem[0013 + 4]

Mem[0017]

0027

将 0027 作为操作数。BR 指令的 RTL 描述是

PC ← Oprnd

297 / 299 因此 CPU 把 0027 放在程序计数器。因为冯·诺依曼周期，所以下一条要执行的指令是在地址 0027 处的指令，这正好是情况 2（case 2）的第一条指令。

C++ 中的 break 语句被翻译成一条 BR 指令，转移到 switch 语句的末尾。如果在 C++ 程序中省略了 break，那么编译器也会省略这个 BR，控制将转移到下一个情况（case）中。

如果用户键入一个不在 0 ~ 3 范围内的数，那么就会发生运行时错误。例如，如果用户给 guess 键入 4，ASLX 指令将它乘以 2，那么变址寄存器中为 8，CPU 这样计算操作数

Mem[OprndSpec + X]

Mem[0013 + 8]

Mem[001B]

4100

因此将转移到内存单元 4100（hex）处。问题在于汇编程序给 STRO 指令生成 001B，这个 001B 不是作为转移地址来解释的。为了防止用户发生这样的事情，C++ 指定如果 guess 的值不是情况之一，那就任何事情都不做。它也给 switch 语句提供了一种 default 情况，用来处理前面没有出现过的情况。编译器一定要给 guess 生成一个初始的条件转移，用来处理其他情况没有覆盖到的值。本章结尾的问题会讨论 switch 语句的这个特性。

6.5　动态内存分配

编译器的目的是为程序员创造一种高层次的抽象。例如，它让程序员思考单个 while 循环，而不是思考在机器上执行循环所必需的汇编层具体的有条件转移。隐藏较低层次的细节是抽象的本质。

但程序控制的抽象仅仅是问题的一面，另一面是数据的抽象。在汇编层和机器层，唯一的数据类型是位和字节。前面的程序展示了编译器怎样翻译字符、整数和数组这些数据类型，这些类型中的每一种都可以是全局的，使用 .BLOCK 分配；或者是局部的，用 SUBSP 在运行时栈上分配。但是 C++ 程序还能包含结构和指针，这是许多数据结构的基本构件。在 HOL6 层，指针访问用 new 操作符在堆上分配的结构。本节展示一个 Asmb5 层上简单的堆操作，以及编译器怎样翻译包含指针和结构的程序。

6.5.1　翻译全局指针

图 6-41 展示了一个有全局指针的 C++ 程序，以及它到 Pep/8 汇编语言的翻译。这个 300 C++ 程序和图 2-37 中的程序是一样的。图 2-38 展示了当程序执行时堆的分配。堆是不同于栈的内存区域。编译器，与操作系统合作，必须生成执行堆分配和释放的代码。

高级语言

```
#include <iostream>
using namespace std;

int *a, *b, *c;
```

图 6-41　全局指针的翻译。这个 C++ 程序来自图 2-37

```
int main () {
    a = new int;
    *a = 5;
    b = new int;
    *b = 3;
    c = a;
    a = b;
    *a = 2 + *c;
    cout << "*a = " << *a << endl;
    cout << "*b = " << *b << endl;
    cout << "*c = " << *c << endl;
    return 0;
}
```

汇编语言

```
0000  040009          BR      main
0003  0000    a:      .BLOCK  2           ;global variable #2h
0005  0000    b:      .BLOCK  2           ;global variable #2h
0007  0000    c:      .BLOCK  2           ;global variable #2h
              ;
              ;******* main ()
0009  C00002 main:    LDA     2,i         ;a = new int
000C  16006A          CALL    new         ;@a
000F  E90003          STX     a,d
0012  C00005          LDA     5,i         ;*a = 5
0015  E20003          STA     a,n
0018  C00002          LDA     2,i         ;b = new int
001B  16006A          CALL    new         ;@b
001E  E90005          STX     b,d
0021  C00003          LDA     3,i         ;*b = 3
0024  F20005          STA     b,n
0027  C10003          LDA     a,d         ;c = a
002A  E10007          STA     c,d
002D  C10005          LDA     b,d         ;a = b
0030  E10003          STA     a,d
0033  C00002          LDA     2,i         ;*a = 2 + *c
0036  720007          ADDA    c,n
0039  E20003          STA     a,n
003C  41006B          STRO    msg0,d      ;cout << "*a = "
003F  3A0003          DECO    a,n         ;      << *a
0042  50003A          CHARO   '\n',i      ;      << endl
0045  41005E          STRO    msg1,d      ;cout << "*b = "
0048  3A0005          DECO    b,n         ;      << *b
004B  50003A          CHARO   '\n',i      ;      << endl
004E  410064          STRO    msg2,d      ;cout << "*c = "
0051  3A0007          DECO    c,n         ;      << *c
0054  50003A          CHARO   '\n',i      ;      << endl
0057  00              STOP
0058  2A6120 msg0:    .ASCII  "*a = \x00"
      3D2000
005E  2A6220 msg1:    .ASCII  "*b = \x00"
      3D2000
0064  2A6320 msg2:    .ASCII  "*c = \x00"
      3D2000
```

图6-41 （续）

```
                ;
                ;******* operator new
                ;       Precondition: A contains number of bytes
                ;       Postcondition: A contains pointer to bytes
006A  C90074 new:    LDX     hpPtr,d         ;returned pointer
006D  710074         ADDA    hpPtr,d         ;allocate from heap
0070  F10074         STA     hpPtr,d         ;update hpPtr
0073  58             RETD
0074  0076   hpPtr:  .ADDRSS heap            ;address of next free byte
0076  00     heap:   .BLOCK  1               ;first byte in the heap
0077                 .END
```

输出

```
*a = 7
*b = 7
*c = 8
```

图 6-41 （续）

301
:
302

当在 C++ 中用指针编程时，要用 new 操作符在堆上分配存储空间。当程序不再需要这些存储空间时，用 delete 操作符释放它们。有可能在堆上分配多个内存单元，再从中间释放一个单元。内存管理算法必须能够处理这种情形。本书只是入门级介绍，为了描述简单，这里使用的说明堆的程序不显示释放过程。堆位于主存中应用程序的末尾。操作符 new 在堆上分配存储空间，因此堆是向下生长的。一旦内存被分配了，就不会被释放。Pep/8 的这个属性是不切实际的，但是比描述实际情况更易于理解。

图 6-41 中的汇编语言程序展示了堆从 0076 开始，这是符号 heap 的值。分配算法维护一个名称 hpPtr 的全局指针，它代表堆指针。在 0074 的语句

```
hpPtr: .ADDRSS heap
```

将 hpPtr 初始化为堆中第一个字节的地址。应用程序向 new 操作符提供需要的字节数，new 操作符返回 hpPtr 的值，然后把 hpPtr 加上请求的字节数。因此 new 操作符保证 hpPtr 总是指向在堆中下一个可以被分配字节的地址。

操作符 new 的调用协议不同于函数的调用协议。对于函数，信息通过运行时栈上的参数进行传递；而对于 new 操作符，应用程序把要分配的字节数放入累加器，再执行 CALL 语句去调用 new 操作符。new 操作符为应用程序把当前的 hpPtr 值放入变址寄存器，因此成功执行 new 操作的前提条件是累加器中包含要在堆上分配的字节数，后置条件是变址寄存器包含 new 操作要在堆上分配的第一个字节的地址。

new 操作符的调用协议比函数的调用协议更高效。new 的实现只要 4 行汇编语言代码，包括 RETD 语句。006A 的语句

```
new: LDX hpPtr,d
```

把堆指针的当前值放入变址寄存器。006D 的语句

```
ADDA hpPtr,d
```

给堆指针加上要分配的字节数。0070 的语句

```
STA hpPtr,d
```

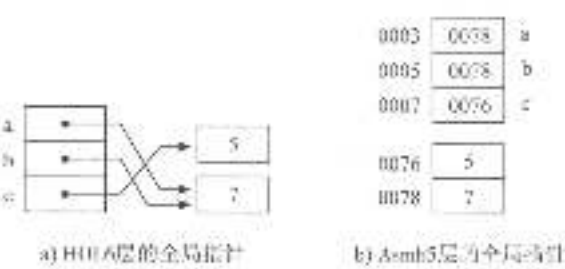
把 hpPtr 更新为堆上第一个未分配字节的地址。

这个协议高效有两个原因。首先，它没有像函数那样很长的参数列表，应用程序只需向new操作符提供一个值。而函数的调用协议必须设计成能够处理任意数量的参数。比如，如 303
果参数列表有4个参数，那么Pep/8的CPU中就没有足够的寄存器来全部保存它们，但是运行时栈可以存储任意数量的参数。其次，new操作符不会调用任何其他的函数，尤其是没有递归调用。函数的调用协议一般来说要设计成允许函数递归地调用其他函数，这样的调用必须要使用运行时栈，但是对于new操作符来说是不必要的。

图6-42a展示了这个HOL6层的C++程序在第一个cout语句之前的内存分配，它对应于图2-38h。图6-42b展示了Asmb5层上同样的内存分配。全局指针a、b和c分别存储在0003、0005和0007。与所有的全局变量一样，用.BLOCK语句对它们进行分配

```
a: .BLOCK 2
b: .BLOCK 2
c: .BLOCK 2
```

HOL6层的指针就是Asmb5层的地址，地址占用2字节，因此给每个全局指针分配2字节。



a) HOL6层的全局指针　　b) Asmb5层为全局变量

图6-42　图6-41在第一条cout语句之前的内存分配

编译器把语句

```
a = new int;
```

翻译成

```
main: LDA  2,i
      CALL new
      STX  a,d
```

LDA指令把2放入累加器。CALL指令调用new操作符，new操作符在堆上分配2字节的存储空间，再把指向分配空间的指针放到变址寄存器中。STX指令把返回的指针存储在全局变量a中。因为a是全局变量，所以STX使用直接寻址。在这一系列指令执行之后，a的值是0076，hpPtr的值是0078，因为它增加了2。

编译器怎样翻译下面这条语句呢？

```
*a = 5;
```
304

当程序执行到此，全局变量a存放的是存储5的地方的地址（这个执行点不对应于图6-42，它是后面的情况）。store指令不能使用直接寻址，因为这样会把地址替换为5，5不是堆上已分配单元的地址。Pep/8提供间接寻址方式，操作数是这样计算的

Oprnd = Mem [Mem[OprndSpec]]

使用间接寻址，操作数指示符是内存中操作数的地址。编译器把赋值语句翻译为

```
LDA 5,i
STA a,n
```

STA语句中的n表示间接寻址。程序此时这样计算操作数

Mem[Mem[OprndSpec]]

Mem[Mem[0003]]

Mem[0076]

这就是堆上的第一个单元。store指令把5存储在主存中地址为0076的位置。

编译器对全局指针赋值的翻译与对任何其他类型全局变量赋值的翻译是一样的。它用间接寻址把

```
c = a;
```

翻译为

```
LDA a,d
STA c,d
```

程序在此时，a 包含 0076，即用上第一个单元的地址。赋值语句赋给 c 同样的值，即单上第一个单元的地址，所以 c 指向 a 指向的同一地址。

对比访问全局指针和访问它指向的单元，编译器将

```
*a = 2 + *c;
```

翻译为

```
LDA 2,i
ADDA c,n
STA a,n
```

305

这里 add 和 store 指令使用间接寻址方式。访问全局指针使用直接寻址，而访问全局指针指向的单元使用间接寻址。可以看到同样的规则适用于 cout 语句的翻译。因为 cout 输出 *a，即 a 指向的单元，所以 003F 的 DECO 指令使用间接寻址。

总之，为了访问全局指针，编译器遵循如下规则生成代码：

- 用 .BLOCK 2 给指针分配存储空间，因为一个地址占用 2 字节。
- 用直接寻址访问指针。
- 用间接寻址访问指针指向的单元。

6.5.2　翻译局部指针

除了指针 a、b 和 c 声明为局部变量而不是全局变量外，图 6-43 中的程序与图 6-41 中的程序是一样的。这两个程序的输出没有什么不同，但是内存模型是完全不同的，因为这个程序的指针是在运行时栈上分配的。

```
高级程序
#include <iostream>
using namespace std;

int main () {
   int *a, *b, *c;
   a = new int;
   *a = 5;
   b = new int;
   *b = 3;
   c = a;
   a = b;
   *a = 2 + *c;
   cout << "*a = " << *a << endl;
   cout << "*b = " << *b << endl;
   cout << "*c = " << *c << endl;
   return 0;
}
汇编程序
0000  C40003          BR      main
```

图 6-43　局部指针的翻译

```
                ;
                ;******* main ()
                a:      .EQUATE 6               ;local variable #2h
                b:      .EQUATE 2               ;local variable #2h
                c:      .EQUATE 0               ;local variable #2h
0003  6B0006  main:   SUBSP   6,i             ;allocate #a #b #c
0006  C00002          LDA     2,i             ;a = new int
0009  16006A          CALL    new             ;&a
000C  EB0004          STX     a,s
000F  C00005          LDA     5,i             ;*a = 5
0012  F40004          STA     a,sf
0015  C00002          LDA     2,i             ;b = new int
0018  16006A          CALL    new             ;&b
001B  EB0002          STX     b,s
001E  C00003          LDA     3,i             ;*b = 3
0021  F40007          STA     b,sf
0024  C30004          LDA     a,s             ;c = a
0027  E30000          STA     c,s
002A  C30002          LDA     b,s             ;a = b
002D  E30004          STA     a,s
0030  C00002          LDA     2,i             ;*a = 2 + *c
0033  740000          ADDA    c,sf
0036  F40004          STA     a,sf
0039  410058          STRO    msg0,d          ;cout << "*a = "
003C  3C0004          DECO    a,sf            ;      << *a
003F  50000A          CHARO   '\n',i          ;      << endl
0042  41005F          STRO    msg1,d          ;cout << "*b = "
0045  3C0002          DECO    b,sf            ;      << *b
0048  50000A          CHARO   '\n',i          ;      << endl
004B  410064          STRO    msg2,d          ;cout << "*c = "
004E  3C0000          DECO    c,sf            ;      << *c
0051  50000A          CHARO   '\n',i          ;      << endl
0054  6B0006          ADDSP   6,i             ;deallocate &c &b &a
0057  00              STOP
0058  2A6120 msg0:    .ASCII  "*a = \x00"
      3D2000
005F  2A6220 msg1:    .ASCII  "*b = \x00"
      3D2000
0064  2A6320 msg2:    .ASCII  "*c = \x00"
      3D2000
                ;
                ;******* operator new
                ;       Precondition: A contains number of bytes
                ;       Postcondition: X contains pointer to bytes
006A  C90074 new:     LDX     hpPtr,d         ;returned pointer
006D  710074          ADDA    hpPtr,d         ;allocate from heap
0070  E10074          STA     hpPtr,d         ;update hpPtr
0073  58              RETO
0074  0076   hpPtr:   .ADDRSS heap            ;address of next free byte
0076  00      heap:   .BLOCK  1               ;first byte in the heap
0077                  .END
```

图 6-43 （续）

图 6-44 展示了图 6-43 中程序在执行第一条 cout 语句前的内存分配，与所有局部变量一样，a、b 和 c 在运行时栈进行分配。图 6-44b 显示了它们距栈顶的偏移量分别为 4、2 和

0。因此，编译器将

```
int *a, *b, *c;
```

翻译为

```
a: .EQUATE 4
b: .EQUATE 2
c: .EQUATE 0
```

因为 a、b 和 c 是局部变量，所以编译器用 SUBSP 给它们生成分配存储空间的代码，用 ADDSP 生成释放存储空间的代码。

编译器将

```
a = new int;
```

翻译为

```
LDA   2,i
CALL  new
STX   a,s
```

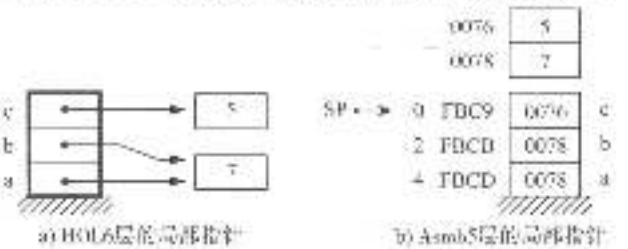


图 6-44　图 6-43 在 cout 语句之前的内存分配

LDA 指令把 2 放入累加器，为调用 new 操作符做准备，放入 2 是因为整数占用 2 字节。CALL 指令调用 new 操作符，new 操作符在堆上分配 2 字节，把它们的地址放过变址寄存器。一般来说，给局部变量赋值用栈相对寻址，因此 STX 语句用栈相对寻址把这个地址赋值给 a。

306
:
308

编译器怎样翻译下面这个赋值呢？

```
*a = 5;
```

a 是指针，把 a 指向的单元赋值为 5。a 也是一个局部变量。这种情形与图 6-27 和图 6-29 中程序的传引用或用参数一样，即操作数的地址在运行时栈上。编译器把赋值语句翻译为

```
LDA 5,i
STA a,sf
```

这里，store 语句使用栈相对间接寻址。

编译器对局部指针赋值的翻译与对任何其他类型的局部变量赋值的翻译是一样的。它使用栈相对寻址把

```
c = b;
```

翻译为

```
LDA c,s
STA c,s
```

这对在程序中，a 包含 0076，即堆上第一个单元的地址。赋值语句给 c 同样的值，即堆上第一个单元的地址，所以 c 指向 a 指向的同一单元。

编译器将

```
*c = 2 + *c;
```

翻译为

```
LDA  2,i
ADDA c,sf
STA  a,sf
```

这里 add 指令使用栈相对间接寻址来访问 c 指向的单元，store 指令使用栈相对间接寻址来访问 a 指向的单元。同样的规则适用于 cout 语句，这里 DECO 指令也使用栈相对间

接寻址。

总之，为了访问局部指针，编译器遵循如下规则生成代码：

- 用 SUBSP 在运行时线给指针分配存储空间，用 ADDSP 释放存储空间。
- 用栈相对寻址访问指针。
- 用栈相对间接寻址访问指针指向的单元。

309

6.5.3 翻译结构

在 HOL6 层，高级语言层，结构是数据抽象的关键。它允许程序员把原始类型的变量组合为一个抽象数据类型。编译器在 HOL6 层提供 struct 结构。在 Asmb5 汇编层，结构是一个连续的字节组，非常像数组字节。但是数组的所有单元都具有相同的类型，因此具有相同的大小，通过索引的整数值来访问每个单元。

使用结构，单元可以有不同的类型和不同的大小。C++ 程序员给每个称为字段的单元一个字段名。在 Asmb5 层，字段名对应于该字段距离结构第一个字节的偏移量。结构的字段名对应于数组的索引。访问结构的字段与访问数组的元素类似，这也就一点都不奇怪了。编译器不是把数组的索引放入变址寄存器，而是生成把字段距离结构第一个字节的偏移量放入变址寄存器的代码。除了这一点不同外，访问结构字段的代码和访问数组元素的代码是一样的。

图 6-45 中的程序，声明了一个叫作 person 的 struct，person 有 4 个字段，first、last、age 和 gender。它和图 2-39 中的程序是一样的。它声明了一个叫作 bill 的全局变量，bill 的类型是 person。图 6-46 展示了这个结构在 HOL6 层和 Asmb5 层的存储空间分配。字段 first、last 和 gender 的类型是 char，每个字段占用 1 字节，字段 age 的类型是 int，占用 2 字节。图 6-46b 给出了结构每个字段的地址，地址左边是距离结构第一个字节的偏移量。除了没有对应于 SP 的到结构顶部的指针外，结构的偏移量类似于栈上元素的偏移量。

高级语言

```
#include <iostream>
using namespace std;

struct person {
   char first;
   char last;
   int age;
   char gender;
};
person bill;

int main () {
   cin >> bill.first >> bill.last >> bill.age >> bill.gender;
   cout << "Initials: " << bill.first << bill.last << endl;
   cout << "Age: " << bill.age << endl;
   cout << "Gender: ";
   if (bill.gender == 'm') {
      cout << "male\n";
   }
   else {
```

图 6-45 结构的翻译。该 C++ 程序来自图 2-39

```
    cout << "female\n";
  }
  return 0;
}
```

汇编之后

```
0000  040008          BR      main
              first:  .EQUATE 0               ;struct field #1c
              last:   .EQUATE 1               ;struct field #1c
              age:    .EQUATE 2               ;struct field #2c
              gender: .EQUATE 4               ;struct field #1c
0003  000000 bill:    .BLOCK  5               ;global variable @first @last @age @gender
      0003
              ;
              ;******* main ()
0008  C80000 main:    LDX     first,i         ;cin >> bill.first
000B  4D0003          CHARI   bill,x
000F  C80001          LDX     last,i          ;   >>bill.last
0012  4D0003          CHARI   bill,x
0015  C80002          LDX     age,i           ;   >>bill.age
0018  350003          DECI    bill,x
001B  C80004          LDX     gender,i        ;   >>bill.gender
001E  4D0003          CHARI   bill,x
0021  41005A          STRO    msg0,d          ;cout << "Initials: "
0024  C80003          LDX     first,i         ;   << bill.first
0027  550003          CHARO   bill,x
002A  C80001          LDX     last,i          ;   << bill.last
002D  550003          CHARO   bill,x
0030  500004          CHARO   '\n',i          ;   << endl
0033  410065          STRO    msg1,d          ;cout << "Age: "
0036  C80002          LDX     age,i           ;   << bill.age
0039  300003          DECO    bill,x
003C  500004          CHARO   '\n',i          ;   << endl;
003F  41006B          STRO    msg2,d          ;cout << "Gender: "
0042  C80004          LDX     gender,i        ;if (bill.gender == 'm')
0045  C00000          LDA     0,i
0048  C50003          LDBYTEA bill,x
004B  B0006D          CPA     'm',i
004E  0C0056          BRNE    else
0051  410074          STRO    msg3,d          ;   cout << "male\n"
0054  040059          BR      endif
0057  41007A else:    STRO    msg4,d          ;   cout << "female\n"
005A  00      endif:  STOP
005B  496E69 msg0:    .ASCII  "Initials: \x00"
      007A
0066  416765 msg1:    .ASCII  "Age: \x00"
      006D
0069  47656E msg2:    .ASCII  "Gender: \x00"
      0074
0075  6D616C msg3:    .ASCII  "male\n\x00"
      007A
007B  66656D msg4:    .ASCII  "female\n\x00"
      0082
0083                  .END
```

图 6-45 （续）

```
输入
bj 32 m
输出
Initials: bj
Age: 32
Gender: male
```

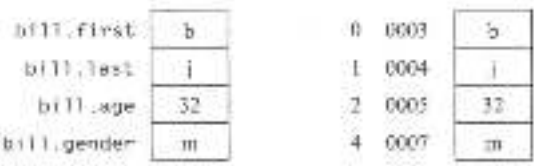
图 6-45　(续)

编译器用 .EQUATE 点命令把

```
struct person {
   char first;
   char last;
   int age;
   char gender;
};
```

翻译为

```
first:   .EQUATE 0
last:    .EQUATE 1
age:     .EQUATE 2
gender:  .EQUATE 4
```



a) HOL6层高全局结构　　b) Asmb6层高全局结构

图 6-46　图 6-45 在 cin 语句之后的内存分配

字段名等于字段距离结构第一个字节的偏移量。因为 first 是结构的第一个字节，所以 first 等于 0。由于 first 占用 1 字节，所以 last 等于 1。因为 first 和 last 共占用 2 字节，所以 age 等于 2。gender 等于 4，因为 first、last 和 age 总共占用 4 字节。编译器把全局变量

```
person bill;
```

翻译为

```
bill:  .BLOCK 5
```

它保留 5 个字节，因为 first、last、age 和 gender 总共占用了 5 个字节。

要访问全局结构的字段，编译器生成把字段距离结构第一个字节的偏移量装入变址寄存器的代码，就像用变址寻址方式访问全局数组的单元一样，用同样的方式访问结构的字段。例如，编译器把

```
cin >> bill.age
```

翻译为

```
LDX   age,i
DECI  bill,x
```

装入指令用立即数寻址把字段 age 的偏移量装入变址寄存器，十进制输入指令用变址寻址来访问这个字段。

编译器类似地把

```
if (bill.gender == 'x')
```

翻译为

```
LDX     gender,i
```

```
LDA     0,i
LDBYTCA bill,x
CPA     'm',i
```

310 / 313

第一个装入指令把 gender 字段的偏移量装入变址寄存器，第二个装入指令清除累加器，确保它最左边的字节为全 0 用于比较。装入字节指令用变址寻址方式访问结构的字段，并把它放入累加器最右的字节。最后，比较指令比较 bill.gender 和字母 m。

总之，为了访问一个全局结构，编译器遵循如下规则生成代码：

- 结构的每个字段等于它距离结构第一个字节的偏移量。
- 用 .BLOCK tot 给结构分配存储空间，tot 是结构占用的总字节数。
- 通过用立即数寻址把字段的偏移量装入变址寄存器，后面跟一条使用变址寻址方式的指令来访问结构的字段。

访问全局结构的字段类似于访问全局数组的元素，与此方式相同，访问局部结构的字段类似于访问局部数组的元素。局部结构在运行时栈上分配，每个字段的名字等于它距离结构第一个字节的偏移量，局部结构的名字等于它距离栈顶的偏移量。编译器生成 SUBSP 给结构和任何其他局部变量分配存储空间，ADDSP 释放存储空间。通过用立即数寻址方式把字段的偏移量装入变址寄存器中，后面跟一个使用栈变址寻址方式的指令来访问结构的字段。本章后面有一道给学生出的问题，要求翻译一个有局部结构的程序。

6.5.4　翻译链式数据结构

程序员经常把指针和结构结合在一起来实现链式数据结构。struct 通常称为结点，指针指向结点，结点有一个指针字段。在数据结构中，结点的指针字段作为到另一个结点的链接。图 6-47 是一个实现了链式数据结构的程序，它和图 2-40 的程序是一样的。

高级语言

```
#include <iostream>
using namespace std;

struct node {
   int data;
   node* next;
};
int main () {
   node *first, *p;
   int value;
   first = 0;
   cin >> value;
   while (value != -9999) {
      p = first;
      first = new node;
      first->data = value;
      first->next = p;
      cin >> value;
   }
   for (p = first; p != 0; p = p->next) {
      cout << p->data << ' ';
   }
```

图 6-47　链表的翻译。该 C++ 程序来自图 2-40

```
   return 0;
}
```

汇编语言

```
0000  040003          BR      main
            data:     .EQUATE 0               ;struct field @2d
            next:     .EQUATE 2               ;struct field @2h
            ;
            ;******* main ()
            first:    .EQUATE 4               ;local variable @2h
            p:        .EQUATE 2               ;local variable @2h
            value:    .EQUATE 0               ;local variable @2d
0003  680006 main:    SUBSP   6,i             ;allocate @first @p @value
0006  C00000          LDA     0,i             ;first = 0
0009  E30004          STA     first,s
000C  330000          DECI    value,s         ;cin >> value
000F  C30000 while:   LDA     value,s         ;while (value != -9999)
0012  00D6F1          CPA     -9999,i
0015  0A003F          BREQ    endWh
0018  C30004          LDA     first,s         ;   p = first
001B  E30002          STA     p,s
001E  C00004          LDA     4,i             ;   first = new node
0021  160067          CALL    new             ;   allocate @data @next
0024  E80004          STX     first,s
0027  C30000          LDA     value,s         ;   first->data = value
002A  E80000          LDX     data,i
002D  E70004          STA     first,sxf
0030  C30002          LDA     p,s             ;   first ->next = p
0033  E80002          LDX     next,i
0036  E70004          STA     first,sxf
0039  330000          DECI    value,s         ;   cin >> value
003C  04000F          BR      while
003F  C30004 endWh:   LDA     first,s         ;for (p = first
0042  E30002          STA     p,s
0045  C30002 for:     LDA     p,s             ;   p != 0
0048  B00000          CPA     0,i
004B  0A0063          BREQ    endFor
004E  C80000          LDX     data,i          ;   cout << p->data
0051  3F0002          DECO    p,sxf
0054  500020          CHARO   ' ',i           ;      << ' '
0057  C80002          LDX     next,i          ;   p = p->next;
005A  C70002          LDA     p,sxf
005D  E30002          STA     p,s
0060  040045          BR      for
0063  600006 endFor:  ADDSP   6,i             ;deallocate @value @p @first
0066  00              STOP
            ;
            ;******* operator new
            ;       Precondition: A contains number of bytes
            ;       Postcondition: X contains pointer to bytes
0067  C90071 new:     LDX     hpPtr,d         ;returned pointer
006A  710071          ADDA    hpPtr,d         ;allocate from heap
006D  E10071          STA     hpPtr,d         ;update hpPtr
0070  58              RETO
```

图 6-47 （续）

```
0071  0073   hpPtr:   .ADDRESS heap          ;address of next free byte
0073  60       heap:   .BLOCK  1             ;first byte in the heap
0074                   .END

输入
10 20 30 40  9999

输出
40 30 20 10
```

图 6-47（续）

编译器把结构的字段

```
struct node {
   int data;
   node* next;
};
```

314
↓
316

设置为它们距离结构第一个字节的偏移量，data 是第一个字段，偏移量是 0，next 是第二个字段，偏移量为 2，因为 data 占用了 2 字节。翻译为

```
data: .EQUATE 0
next: .EQUATE 2
```

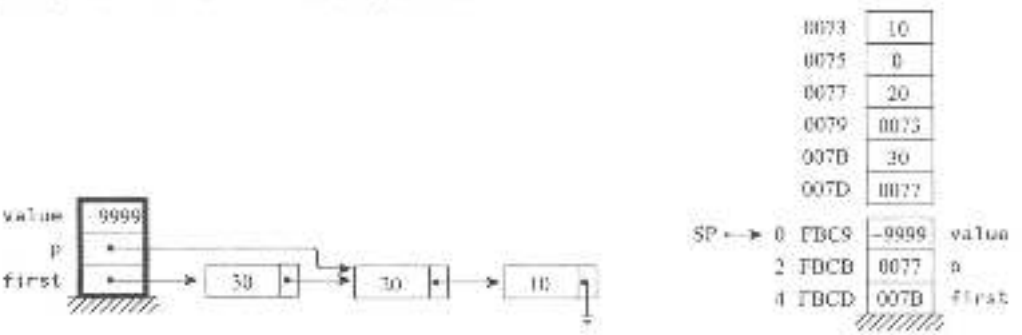
编译器像翻译所有局部变量一样翻译这些局部变量

```
node *first, *p;
int value;
```

它把变量名设置为距离运行时栈顶部的偏移量，翻译为

```
first: .EQUATE 4
p:     .EQUATE 2
value: .EQUATE 0
```

图 6-48b 展示了这些局部变量的偏移量。编译器在 0003 生成 SUBSP，给局部变量分配存储空间，在 0063 的 ADDSP 释放存储空间。



a) HOL6层的栈　　b) Asmb5层的栈

图 6-48　图 6-47 在第三次执行 while 循环后的内存分配

当在 C++ 中使用 new 操作符时，计算机必须在堆上分配足够的内存来存储指针指向的条目。在这个程序中，一个结点占用 4 字节，因此，编译器通过在它生成的代码中调用 new 操作符分配 4 字节来将

```
first = new node;
```

翻译为

```
LDA  4,i
CALL new
STX  first,s
```

317

在准备调用 new 时，装入指令把 4 放入累加器，调用指令调用 new 操作符，new 操作符把被分配结点的第一个字节的地址放入变址寄存器，存储索引指令用栈相对寻址完成给局部变量 first 赋值。

编译器是怎样生成访问局部指针所指向的结点字段的代码呢？记住指针是一个地址，局部指针意味着结点的地址在运行时栈上。此外，struct 的字段对应于数组的索引。如果数组第一个单元的地址在运行时栈上，那么用栈变址间接寻址访问数组元素，访问结点的字段也用同样的方法。把字段偏移量而不是索引值放入变址寄存器中。编译器将

```
first->data = value;
```

翻译为

```
LDA value,s
LDX data,i
STA first,sxf
```

类似地，编译器将

```
first->next = p;
```

翻译为

```
LDA p,s
LDX next,i
STA first,sxf
```

来看一看指向结点的局部指针怎样使用栈变址间接寻址。记住 CPU 这样计算操作数

Oprnd = Mem[Mem[SP + OprndSpec] + X]

它将栈指针和操作数指示符相加，把这个和作为第一个字段的地址，再把它加到变址寄存器上。假定第三个结点已经如图 6-48b 所示的那样分配了。调用 new 已经返回最新分配的结点地址 007B，把它存储在 first 中。这时上面的 LDA 指令已经把 p 的值 0077 放入累加器。LDX 指令也已把 next 值，即偏移量 2 放入变址寄存器。用栈变址寻址执行 STA 指令，操作数指示符是 4，即 first 值。操作数的计算为

Mem[Mem[SP + OprndSpec] + X]

Mem[Mem[FBC9 + 4] + 2]

Mem[Mem[FBCD] + 2]

Mem[Mem[007B + 2]

Mem[007D]

318

这是 first 指向的结点的 next 字段。

总之，为了访问局部指针指向的结点的字段，编译器遵循如下生成代码：

- 结点的字段名等于字段距离结点第一个字节的偏移量，把偏移量装入变址寄存器。
- 访问结点字段的指令使用栈索引间接寻址。

你应该能够确定编译器是怎样翻译全局指针指向的结点的程序了吧。本章末尾有一个给学生的练习就是归纳总结这个翻译规则，还有一道问题是翻译具有指向结点的全局指针的 C++ 程序。

总结

编译器用机器层的条件转移指令翻译高级语言层的 if 语句和循环。if/else 语句需要一个条件转移指令测试 if 条件，else 部分需要一个无条件转移指令。while 或 do 循环的翻译需要一个跳转到前面指令的转移。除此之外，for 循环还需要指令初始化和递增控制变量。

Bohm 和 Jacopini 证明的结构化编程原理认为任何包含 goto 的算法，不管多么复杂和无结构，都可以用嵌套 if 语句和 while 循环来编写。Dijkstra 那封著名的信中指明没有 goto 的程序不仅可行而且更好，并由此引发了针对 goto 的争论。

编译器在主存的固定位置分配全局变量，过程和函数在运行时栈上分配参数和局部变量。通过增大栈指针（SP），值被压入栈中；通过减小 SP，值弹出栈。子程序调用指令把程序计数器（PC）的内容，充当返回地址，压入栈中。子程序返回指令把返回地址从栈弹出到 PC。指令用直接寻址方式访问全局值，运行时栈上的值用栈相对寻址来访问。传引用调用的参数会把它的地址压入运行时栈中，然后用栈相对间接寻址来访问这样的参数。布尔变量的 false（假）存储为值 0，true（真）存储为值 1。

数组值存储在连续的内存单元中。用变址寻址访问全局数组的元素，用栈变址寻址来访问局部数组的元素。两种情况下，变址寄存器都包含数组元素的索引值。作为参数传递的数组总是把数组第一个单元的地址压入运行时栈。用栈变址间接寻址来访问这样数组的元素。编译器用一个地址数组来翻译 switch 语句，该数组的元素是每个 case 第一条语句的地址。

指针和 struct 类型是数据结构的常用构件。指针是堆上一个内存单元的地址。new 操作符在堆上分配内存。用非直接寻址来访问全局指针指向的单元，用栈相对间接寻址来访问局部指针指向的单元。struct 有多个命名的字段，存储为一个连续的字节组。用变址寻址来访问 global 结构的字段，变址寄存器包含字段距离 struct 第一个字节的偏移量。链式数据结构通常有一个指向 struct 的指针，这个 struct 称为结点，这个结点又含有指向另一个结点的指针。如果局部指针指向一个结点，那么使用栈变址间接寻址来访问该结点的字段。

319

练习

6.1 节

1. 请解释全局变量和局部变量内存模型的不同点。它们是怎样分配和存取的？

6.2 节

2. 什么是优化编译器？什么时候使用？什么时候不使用？请解释。

*3. 图 6-14 中目标代码在 000C 处的 CPA 测试 j 的值。由于程序从循环底部跳转到那条指令，为什么编译器此时不在 CPA 之前生成一条 LDA j，d 语句？

4. 研究图 6-16 神秘程序的函数，用一句简短的话说明它是做什么的？

5. 阅读本章所引用的 Bohm、Jacopini 以及 Dijkstra 的论文，据此写一篇综述文章。

6.3 节

*6. 像图 6-19 那样画出图 6-18 中 0022 处的 CALL 语句之前和之后的值。

7. 像图 6-26 那样画出对应于第二次返回前的运行时栈。

6.4 节

*8. 在图 6.40 的 Pep/8 程序中，如果给 Guess 输入 4，在 0010 的转移之后执行什么语句？为什么？

9. 6.4 节没有展示怎样访问二维数组的元素。描述可以怎样存储二维数组，以及从这样的二维数组存取元素所需的汇编语言目标代码。

6.5 节

10. 访问全局指针指向结构的字段的转换规则是什么？ 320

问题

6.2 节

11. 将下列 C++ 程序翻译为 Pep/8 汇编语言：

```
#include <iostream>
using namespace std;

int main () {
   int number;
   cin >> number;
   if (number % 2 == 0) {
      cout << "even\n";
   }
   else {
      cout << "Odd\n";
   }
   return 0;
}
```

12. 将下列 C++ 程序翻译为 Pep/8 汇编语言：

```
#include <iostream>
using namespace std;

const int limit = 5;

int main () {
   int number;
   cin >> number;
   while (number < limit) {
      number++;
      cout << number << ' ';
   }
   return 0;
}
```

13. 将下列 C++ 程序翻译为 Pep/8 汇编语言：

```
#include <iostream>
using namespace std;

int main () {
   char ch;
   cin >> ch;
   if ((ch >= 'A') && (ch <= 'Z')) {
      cout << 'A';
   }
   else if ((ch >= 'a') && (ch <= 'z')) {
      cout << 'a';
   }
   else {
      cout << '1';
   }
   cout << endl;
   return 0;
}
```

14. 将图 6-12 的 C++ 程序翻译为 Pep/8 汇编语言，但是把 do 循环的测试条件改为

```
while (cop <= driver);
```

15. 将下列C++程序转换为Pep/8汇编语言：

```
#include <iostream>
using namespace std;

int main () {
   int numItms, j, data, sum;
   cin >> numItms;
   sum = 0;
   for (j = 1; j <= numItms; j++) {
      cin >> data;
      sum += data;
   }
   cout << "Sum: " << sum << endl;
   return 0;
}
```

示例输入

```
4    8    -3    7    6
```

示例输出

```
Sum: 18
```

6.3 节

16. 将下列C++程序翻译为Pep/8汇编语言：

```
#include <iostream>
using namespace std;

int myAge;

void putNext (int age) {
   int nextYr;
   nextYr = age + 1;
   cout << "Age: " << age << endl;
   cout << "Age next year: " << nextYr << endl;
}

int main () {
   cin >> myAge;
   putNext (myAge);
   putNext (64);
   return 0;
}
```

17. 将上面习题16中的C++程序翻译成Pep/8汇编语言，但是把myAge声明成main()中的局部变量。

18. 将下列C++程序转换为Pep/8汇编语言。它用递归位移相加（recursive shift-and-add）算法进行两个整数的乘法。

```
#include <iostream>
using namespace std;

int times (int mpr, int mcand) {
   if (mpr == 0) {
      return 0;
   }
   else if (mpr % 2 == 1) {
      return times (mpr / 2, mcand * 2) + mcand;
   }
   else {
      return times (mpr / 2, mcand * 2);
```

```
    }
}

int main () {
    int n, m;
    cin >> n >> m;
    cout << "Product: " << times (n, m) << endl;
    return 0;
}
```

19. (a) 写一个把小写字母转换成大写字母的 C++ 程序。转换函数声明为

```
char uppercase (char ch);
```

如果实参不是小写字母，函数返回不改变的字符值。用交互 I/O 在主程序中测试你的函数。(b) 把你写的 C++ 程序翻译成 Pep/8 汇编语言。

20. (a) 写一个 C++ 程序，定义

```
int minimum (int j1, int j2)
```

返回 j1 和 j2 中较小的那个，用交互输入测试这个程序。(b) 把你写的 C++ 程序翻译成 Pep/8 汇编语言。

321 ~ 323

21. 把习题 2.14 中用递归函数计算斐波那契数列的 C++ 方案翻译成 Pep/8 汇编语言。

22. 把习题 2.15 中输出汉诺塔游戏输出指令的 C++ 方案翻译成 Pep/8 汇编语言。

23. 图 6-25 的递归二项式系数函数可以通过像下面那样忽略 y1 和 y2 进行简化：

```
int binCoeff (int n, int k) {
    if (k == 0 || n == k) {
        return 1;
    }
    else {
        return binCoeff (n - 1, k) + binCoeff (n - 1, k - 1);
    }
}
```

写一个 Pep/8 汇编语言程序，调用这个函数。把从 binCoeff(n-1,k) 返回的值保存在栈上，在这个值上面给 binCoeff(n-1,k-1) 调用分配实参。图 6-49 展示了运行时栈的历史信息。这里的栈帧包含 4 个字 (retVal、n、k 和 retAddr)，阴影表示的字是函数调用的返回值。它展示从主程序中调用 binCoeff(3,1) 的历史记录。

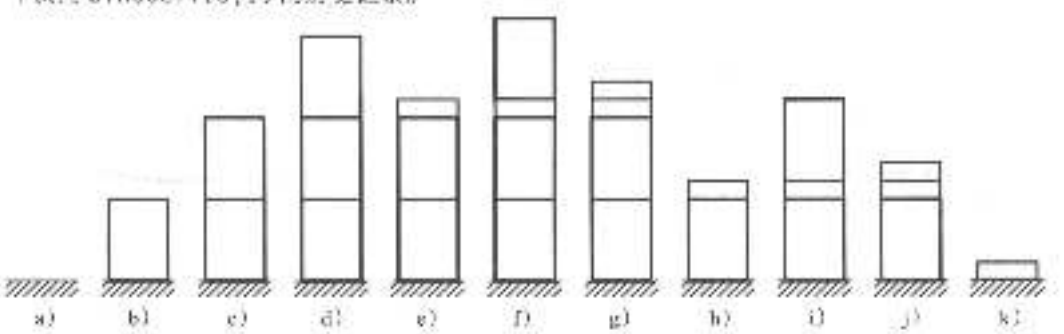


图 6-49　练习 23 的运行时栈历史状态

24. 将下列 C++ 程序翻译为 Pep/8 汇编语言。它用迭代位移相加算法运行两个整数的乘法。

```
#include <iostream>
using namespace std;

int product, n, m;
```

```
void times (int& prod, int mpr, int mcand) {
   prod = 0;
   while (mpr != 0) {
      if (mpr % 2 == 1) {
         prod = prod + mcand;
      }
      mpr /= 2;
      mcand *= 2;
   }
}

int main () {
   cin >> n >> m;
   times (product, n, m);
   cout << "Product: " << product << endl;
   return 0;
}
```

25. 将问题24中的C++程序翻译为Pep/8汇编语言，但是把product、n和m声明为main()的局部变量。

26. (a) 重写图2-22中计算阶乘的C++程序，但是使用问题24中的times过程来进行乘法运算。在fact中用一个额外的局部变量来存储乘积。(b) 将你写的C++程序翻译成Pep/8汇编语言。

6.4节

27. 将下列C++程序翻译为Pep/8汇编语言：

```
#include <iostream>
using namespace std;

int list[16];
int j, numItems;
int temp;

int main () {
   cin >> numItems;
   for (j = 0; j < numItems; j++) {
      cin >> list[j];
   }
   temp = list[0];
   for (j = 0; j < numItems - 1; j++) {
      list[j] = list[j + 1];
   }
   list[numItems - 1] = temp;
   for (j = 0; j < numItems; j++) {
      cout << list[j] << ' ';
   }
   cout << endl;
   return 0;
}
```

示例输入

```
5
11  22  33  44  55
```

示例输出

```
22  33  44  55  11
```

第二个for循环的测试很难翻译，因为<运算符右边的算术表达式。可以通过把这个测试转换成下面数学上等价的测试来简化翻译：

```
j + 1 < numItems;
```

28. 把上面问题27中的C++翻译为Pep/8汇编语言，但是要把list、j、numItems和temp声明成

main() 中的局部变量。

29. 将下列 C++ 程序翻译为 Pep/8 汇编语言：

```
#include <iostream>
using namespace std;

void getList (int ls[], int& n) {
   int j;
   cin >> n;
   for (j = 0; j < n; j++) {
      cin >> ls[j];
   }
}

void putList (int ls[], int n) {
   int j;
   for (j = 0; j < n; j++) {
      cout << ls[j] << ' ';
   }
   cout << endl;
}

void rotate (int ls[], int n) {
   int j;
   int temp;
   temp = ls[0];
   for (j = 0; j < n - 1; j++) {
      ls[j] = ls[j + 1];
   }
   ls[n - 1] = temp;
}

int main () {
   int list[16];
   int numItems;
   getList (list, numItems);
   putList (list, numItems);
   rotate (list, numItems);
   putList (list, numItems);
   return 0;
}
```

示例输入

```
5
11  22  33  44  55
```

示例输出

```
11  22  33  44  55
22  33  44  55  11
```

30. 将上面问题 29 中的 C++ 程序翻译为 Pep/8 汇编语言，但是把 list 和 numItems 声明为全局变量。

31. 将图 2-25 中用递归程序把一个数组中 4 个值相加的 C++ 程序翻译为 Pep/8 汇编语言。

32. 将图 2-32 用递归程序反转数组元素的 C++ 程序翻译为 Pep/8 汇编语言。

33. 将下列 C++ 程序翻译为 Pep/8 汇编语言：

```
#include <iostream>
using namespace std;

int main () {
   int guess;
   cout << "Pick a number 0..3: ";
   cin >> guess;
   switch (guess) {
```

```
        case 0: case 1: cout << "Too low"; break;
        case 2: cout << "Right on"; break;
        case 3: cout << "Too high";
    }
    cout << endl;
    return 0;
}
```

这个程序除了两种情况执行相同的代码外，和图 6-40 的程序是一样的。跳转表必须正好有 4 个条目，但是程序必须只有 3 个情况符号和 3 种情况。

34. 将下列 C++ 程序翻译为 Pep/8 汇编语言：

```
#include <iostream>
using namespace std;

int main () {
int guess;
cout << "Pick a number 0..3: ";
cin >> guess;
switch (guess) {
   case 0: cout << "Not close"; break;
   case 1: cout << "Too low"; break;
   case 2: cout << "Right on"; break;
   case 3: cout << "Too high"; break;
   default: cout << "Illegal input";
}
cout << endl;
return 0;
}
```

6.5 节

35. 将图 6-45 访问结构字段的 C++ 程序翻译为 Pep/8 汇编语言，但是把 bill 声明为 main() 的局部变量。

36. 将图 6-47 的对链表进行操作的 C++ 程序翻译为 Pep/8 汇编语言，但是把 first、p 和 value 声明为全局变量。

37. 在图 6-47 中 main() 的 return 语句前插入下面的 C++ 代码段：

```
sum = 0; p = first;
while (p != 0) {
   sum += p->data;
   p = p->next;
}
cout << "Sum: " << sum << endl;
```

将整个程序翻译为 Pep/8 汇编语言，把 sum 和其他局部变量一样声明如下：

```
node *first, *p;
int value, sum;
```

38. 将下面的代码段插入到图 6-47 中 node 的声明和 main() 之间：

```
void reverse (node* list) {
   if (list != 0) {
      reverse (list->next);
      cout << list->data << ' ';
   }
}
```

把下面的代码段插入到 main() 中 return 语句的前面：

```
cout << endl;
reverse (first);
```

将整个程序翻译为 Pep/8 汇编语言。增加的代码以逆序输出链表。

39. 在图 6-47 中 main() 的 return 语句前插入下面的代码段：

```
first2 = 0; p2 = 0;
for (p = first; p != 0; p = p->next) {
   p2 = first2;
   first2 = new node;
   first2->data = p->data;
   first2->next = p2;
}
for (p2 = first2; p2 != 0; p2 = p2->next) {
   cout << p2->data << " ";
}
```

把 first2 和 p2 与其他局部变量一样声明为：

```
node *first, *p, *first2, *p2;
int value;
```

将整个程序翻译为 Pep/8 汇编语言。增加的代码生成第一个表的副本，以逆序排列并输出它。

40. (a) 写一个 C++ 程序，把一个无序整数列输入到一个二叉搜索树，该数列用 -9999 作为分界标识符号，用中序遍历树输出它们。(b) 把这个程序翻译为 Pep/8 汇编语言。

41. 这道题目是一个项目，用 C++ 写出 Pep/8 计算机的模拟器。(a) 写出一个装载器，以标准格式的 Pep/8 目标文件输入，把它加载到模拟的 Pep/8 计算机的内存中。把主存声明为一个整数数组，如下所示：

```
int Mem[65536];  // Pep/8 main memory
```

把从标准输入来的一个字符串作为输入。写出一个内存转储（memory dump）函数，以代表该程序的十进制整数序列的形式输出主存的内容。例如，如果输入图 4-41 所示

```
51 00 07 51 00 08 00 48 65 zz
```

那么程序应该把十六进制数转换为整数，把它们存储在 Mem 的前 9 个单元中。输出就应该是如下的对应整数值：

```
81 0 7 81 0 8 0 72 105
```

(b) 实现指令 CHARO、DECO 和 STOP，寻址方式为立即数和直接寻址。把 DECO 指令作为原生指令来实现，也就是说，不要实现 8.2 节中讲述的陷阱机制。使用图 4-31 帮助你实现冯·诺依曼执行周期。例如，问题 (a) 中的输入对应的输出应该是 Hi。 [329]

(c) 实现指令 BR、LDr、LDBYTEr、STr、STBYTEr、SUBSP 和 ADDSP，寻址方式为栈相对寻址。用 Pep/8 汇编器汇编图 6-1 的程序，然后把十六进制程序输入你的模拟器，以此来检验你的实现。

(d) 以原生指令方式实现指令 DECI 和 STRO。输入来自 C++ 的标准输入。执行图 6-4 的程序来检验你的实现。

(e) 实现条件跳转指令 BRLE、BRLT、BREQ、BRNE、BRGE、BRGT 和 BRV、一元指令 NOTr 和 NEGr，以及比较指令 CPr。执行图 6-6、图 6-8、图 6-10、图 6-12 和图 6-14 的程序来检验你的实现。

(f) 实现指令 CALL 和 RETn。执行图 6-18、图 6-21、图 6-23 和图 6-25 的程序来检验你的实现。

(g) 实现指令 MOVSPA，寻址方式为栈相对间接。执行图 6-27 和 6-29 的程序来检验你的实现。

(h) 实现指令 ASLr 和 ASRr，寻址方式为变址、栈变址和栈变址间接。执行图 6-34、图 6-36、图 6-38、图 6-40 和图 6-47 的程序来检验你的实现。

(i) 实现间接寻址模式。执行图 6-41 的程序来检验你的实现。 [330]

第 7 章

Computer Systems, Fourth Edition

语言翻译原理

现在，你也是使用多语种的人了，因为你至少懂得 4 种语言了——汉语、C++、Pep/8 汇编语言和机器语言。汉语是自然语言，而其他 3 种是人工语言。

要记住这一点。下面让我们先转到计算机科学的基本问题，即“什么能够被自动化？”我们用计算机来自动化所有的事情，从写工资支票到纠正文稿中的拼写错误。尽管计算机在自然语言翻译上还不算很成功，比如从德语翻译到英语，但在翻译人工语言方面，它已经非常成功了。我们已经学习了如何对 3 种人工语言 C++、Pep/8 汇编语言和机器语言互相转换进行翻译。编译器和汇编器自动化了这些人工语言之间的翻译过程。

因为计算机系统的每一层都有自己的人工语言，所以这些语言之间的自动翻译是计算机科学的核心。计算机科学家已经提出了非常丰富的人工语言及其自动翻译过程的理论，本章的内容就是介绍这一理论，并展示它怎样应运于从 C++ 到 Pep/8 汇编语言的翻译。

语法和语义是人工语言的两个属性。计算机语言的语法（syntax）是一个程序要成为合法的语言程序必须要遵守的一套规则。语义（semantics）是合法程序背后的含义或逻辑。在操作上，翻译器程序可以成功地翻译语法正确的程序。语言的语义决定目标程序执行时，翻译后程序产生的结果。

自动翻译器中比较源程序和语言语法的部分叫作分析器（parser）。给源程序指定含义的部分叫作代码生成器（code generator）。大多数计算机科学理论应用于翻译过程的语法部分而不是语义部分。

这里是描述语言语法的 3 种常用技术：

- 语法
- 有限状态机
- 正则表达式

本章介绍语法和有限状态机，展示怎样构建软件有限状态机来帮助语法分析过程。最后一节讲述了一个完整的程序，包括代码生成，它实现了两种语言之间的自动翻译。由于篇幅限制，本章没有对正则表达式进行讲解。

7.1 语言、语法和语法分析

每种语言都有它的字符表。从形式上来说，每个字符表都是一组有限、非空的字符。例如，C++ 的字符表是非空集合

```
{ a, b, c, d, e, f, g, h, i, j, k, l, m, n,
  o, p, q, r, s, t, u, v, w, x, y, z, A, B,
  C, D, E, F, G, H, I, J, K, L, M, N, O, P,
  Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3,
  4, 5, 6, 7, 8, 9, +, -, ^, /, -, <, >, [,
  ], (, ), [, ], ., ,, :, ;, &, !, $, ", ",
```

_, \, #, ?, ^, |, ~}

Pep/8 汇编语言的的字符表除了标点符号之外其他是一样的，如下所示：

```
{ a, b, c, d, e, f, g, h, i, j, k, l, m, n,
  o, p, q, r, s, t, u, v, w, x, y, z, A, B,
  C, D, E, F, G, H, I, J, K, L, M, N, O, P,
  Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3,
  4, 5, 6, 7, 8, 9, \, ., ,, :, ;, ', ")
```

另一个字符表的例子是实数语言的字符表，不是科学计数法表示的实数。它的字符表集合为

```
{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .}
```

332

7.1.1　连接

抽象数据类型是一个可能值的集合和一组作用于这些值的运算。注意，字符表就是值的集合。值的集合上的一种运算叫连接（concatenation），它简单地连接两个或者更多的字符成为字符串。C++ 字符表中的一个例子是把 ! 和 = 连接成为字符串 !=。在 Pep/8 汇编字符表中，可以把 d 和 # 连接成为 d#，在实数语言中，可以把 -、2、3、. 和 7 连接成为 -23.7。

连接不仅适用于字符表中单个字符构成字符串，也适用于用字符串构成更大的字符串。从 C++ 字符表，可以连接 void、printBar 和 (int n) 来生成过程头

```
void printBar (int n)
```

字符串的长度是字符串中字符的个数。字符串 void 的长度是 4。长度为 0 的字符串叫空字符串，用希腊字母 ε 表示，区别于字母表中的英语字符。它有这样的连接性质

$$\varepsilon x = x\varepsilon = x$$

这里 x 是一个字符串。空字符串对描述语法规则非常有用。

在数学术语中，ε 是连接运算的单位元。一般来说，一个运算的单位元（identity element）i，在作用于一个值 x 时，不会改变 x 的值。

例 7.1　1 是乘法运算的单位元，因为

$$1 \cdot x = x \cdot 1 = x$$

true 是 AND 运算的单位元，因为

$$\text{true AND } q = q \text{ AND true} = q$$

□

7.1.2　语言

如果 T 是一个字符表，T 的闭包表示为 T^*，它是通过连接 T 中的元素可能形成的所有字符串的集合。T^* 是非常大的。例如，如果 T 是英语字符表中字符和标点符号的集合，那么 T^* 会包括所有莎士比亚著作、英文圣经和所有曾经出版的英文百科全书的句子，还包括全世界历史上所有图书馆曾经用这些字符印刷的所有字符串。

333

它不仅包括有意义的字符串，也包括无意义的字符串。这里是英语字符表 T^* 的一些元素：

```
To be or not to be, that is the question.
Go fly a kite.
```

```
Here ever highly toward!
alkeu jfaj ,9nn20mfq23ik !?x!jeo
```

当 T 是实数语言的字符表时，T^* 的一些元素可以是：

```
-2694.31
24
+78.3.80
-234-
5
```

你可以很容易的构建刚才提到的两个字符表的 T^* 的很多其他元素。因为字符串可以无限长，所以任何字符表的闭包中元素个数是无限的。

什么是语言呢？在前面讲述的 T^* 例子中，一些字符串是语言，一些不是。在英语的例子中，前两个字符串是有效的英语句子，即它们是语言。后两个不是语言。语言（language）是它字符表闭包的子集。可以用字符表中的字符组成的字符串通过连接构建无限多的字符串，但是其中只有一些包含在语言中。

例 7.2　思考下面 T^* 中的两个元素，这里 T 是 C++ 语言字符表：

```
#include <iostream>
int main() {
  cout << "Valid";
  return 0;
}

#include <iostream>
int main(); {
  cout << "Valid";
  return 0;
}
```

334　T^* 的第一个元素是 C++ 语言，但第二个不是，因为它有一个语法错误。　□

7.1.3　语法

要定义一种语言，需要一种方式能够说明 T^* 中哪些元素是语言，哪些不是。语法（grammar）就是这样一种系统，它指明可以怎样连结字符表 T 的字符形成一种对该语言来说合法的字符串。形式上，语法包含 4 个部分：

- N，一个非终结字符表。
- T，一个终结字符表。
- P，一套产生式规则。
- S，初始符，为 N 的一个元素。

非终结字符表 N 的一个元素代表终结字符表 T 的一组字符。非终结符号通常用 <> 括起来。当阅读语言时，看到的是终结符。产生式规则使用非终结符来描述语言的结构，这可能不像阅读语言那么明显。

例 7.3　C++ 语法中，非终结符 <compound-statement>（< 复合语句 >）可能代表下面一组终结符：

```
{
  int i;
  cin >> i;
  i++;
```

```
    cout << i;
}
```

C++程序的代码始终是包含终结符而绝不会包含非终结符。我们肯定不会看到这样的C++代码

```
#include <iostream>
main()
<compound statement>
```

非终结符<compound-statement>用于描述C++程序的结构。 □

每种语法都有一个特殊的非终结符，叫作初始符（start symbol），S。注意N是一个集合，但S不是。S是集合N的一个元素。初始符和产生式规则，P一起，使我们能确定一个由终结符组成的字符串是否是该语言的一个合法句子。如果从S开始用产生式规则能产生该终结符组成的字符串，那么该字符串就是一个合法的句子。 335

7.1.4 C++标识符的语法

图7-1中的语法定义了C++标识符。尽管C++标识符可以使用任意的大写或小写字母或者数字，但是为了使例子简短，这个语法仅允许字母a、b、c和数字1、2、3。这样的简化仍然能够使我们了解构成标识符的规则。第一个字符必须是字母，如果有的话，剩下的字符可以是字母或数字的任意组合。

这个语法有3个非终结符，即<identifier>（<标识符>）、<letter>（<字母>）和<digit>（<数字>）。初始符是<identifier>，它是非终结符集合中的一个元素。

产生式规则的形式是

$A \rightarrow w$

这里A是非终结符，w是终结符和非终结组成的字符串。符号→表示"产生"。图7-1中的产生式规则3读作"标识符产生后面带数字的标识符"。

```
N = {<identifier>, <letter>, <digit>}
T = {a, b, c, 1, 2, 3}
P = the productions
    1. <identifier>  → <letter>
    2. <identifier>  → <identifier> <letter>
    3. <identifier>  → <identifier> <digit>
    4. <letter> → a
    5. <letter> → b
    6. <letter> → c
    7. <digit> → 1
    8. <digit> → 2
    9. <digit> → 3
S = <identifier>
```

图7-1 C++标识符的语法

语法通过称为推导（derivation）的过程来指定语言。为了推导一个合法的语言句子，从初始符开始，根据产生式规则不断替代非终结符，直到得到由终结符组成的字符串。下面是根据语法推导出标识符cab3的情况，符号⇒表示一步推导"：

```
<identifier> =><identifier><digit>          规则 3
             =><identifier> 3               规则 9
             =><identifier><letter> 3       规则 2
             =><identifier> b 3             规则 5
             =><identifier><letter> b 3     规则 2
             =><identifier> a b 3           规则 4
             =><letter> a b 3               规则 1
             => c a b 3                     规则 6
```

336

每一步推导的后面是依据这个替换的产生式规则。例如，规则2

<identifier> → <identifier><letter>

用于替换推导步骤

<identifier> 3 -><identifier><letter> 3

中的 <identifier>。应该把这步推导读作“identifier 后面跟着 3 一步推导为 identifier 后面跟着字母，字母后面跟着 3”。

推导运算的闭包类似于字符去的闭包运算，符号 ⇒* 表示“经过零步或多步推导”，可以把前面 8 步推导概括为：

<identifier> ⇒* c a b 3

这个推导证明 cab3 是一个合法的标识符，因为它能够从初始符 <identifier> 推导出来。一个语法定义的语言是由使用产生式规则可以从初始符推导出的所有字符串组成的。语法提供了判定是否属于一个语言的测试规则，如果是不能被推导出来的字符串，那么就不属于该语言。

7.1.5　有符号整数的语法

图 7-2 中的语法定义了有符号整数的语言，d 代表一个十进制数字。初始符 I 代表整数，F 是第一个字符，可以是任意符号，M 表示大小。

有时候为了节省页面空间，产生式规则没有编号，放在一行上。这个语法的产生式规则可以写成这样

I → FM

F → + | - | ε

M → d | dM

这里的竖线 | 是间隔运算符，读作“或”。最后一行读作“M 产生 d，或者 d 后面跟 M”。

N = {I, F, M}
T = {+, -, d}
P = the productions
1. I → FM
2. F → +
3. F → -
4. F → ε
5. M → dM
6. M → d
S = I

图 7-2　有符号整数的语法

下面是这个语法的一些合法有符号整数的推导：

I → FM　　I ⇒ FM　　I ⇒ FM
→ FdM　　⇒ FdM　　⇒ FdM
⇒ FddM　　⇒ Fdd　　⇒ FddM
⇒ Fddd　　⇒ dd　　⇒ FdddM
⇒ -ddd　　⇒ Fdddd
337　　⇒ +dddd

注意第二个推导的最后一步怎样用空字符串从 Fdd 推导出 dd，它使用了产生式 F → ε 和 ε d = d 的事实。这种带空字符串的产生式规则能很方便地表达大小前面的正号或负号是可选的。

ddd+、+-ddd 和 ddd+dd 都是这种语法的非法字符串。试着用语法推导这些字符串来确认它们不属于该语言。你能根据产生式规则证明这几个字符串不属于该语言吗？

这两个语法示例中的产生式都有递归规则，用非终结符来定义它自己。图 7-1 中规则 3 根据 <identifier> 定义 <identifier>，如下所示。

<identifier> → <identifier><digit>

图 7-2 中规则 5 根据 M 定义 M，如下所示。

M → dM

递归规则产生的语言具有无穷多合法的句子。要推导出一个具有任意长度的标识符，只要一直将 <identifier> 替换成 <identifier><digit> 即可。

与所有的递归定义一样，必须有一个安全出口作为定义的基础，否则对非终结符的替换就不会停下来。图 7-2 中 M → d 的规则给 M 的递归提供了基础。

7.1.6　上下文相关的语法

前面讲到的语法的产生式规则都是左边是单个的非终结符。图 7-3 中的语法有一些左边既有非终结符又有终结符的产生式规则。

这是一个使用该语法的终结字符串推导：

A ⇒ aABC	规则 1
⇒ aaABCBC	规则 1
⇒ aaabCBCBC	规则 2
⇒ aaabBCCBC	规则 3
⇒ aaabBCBCC	规则 3
⇒ aaabBBCCC	规则 3
⇒ aaabbBCCC	规则 4
⇒ aaabbbCCC	规则 4
⇒ aaabbbcCC	规则 5
⇒ aaabbbccC	规则 6
⇒ aaabbbccc	规则 6

N = {A, B, C}
T = {a, b, c}
P = the productions
1. A → aABC
2. A → abC
3. CB → BC
4. bB → bb
5. bC → bc
6. cC → cc
S = A

图 7-3　上下文相关语法

338

这个推导中一个替换的例子是在步骤 aaabbbCCC => aaabbbcCC 使用了规则 5，规则 5 允许用 c 替换 C，但是只有当 C 的左边有 b 的时候才行。

在英语中，断章取义是指不考虑一句话周围的语句而引用它。规则 5 就是一个上下文相关规则的例子，它不允许用 c 替换 C，除非 C 有适当的上下文，即它正好在一个 b 的右边。

不严格地说，上下文相关语法（context-sensitive grammar）的产生式规则左边可以包含多于一个的非终结符。与之相比，每个产生式规则左边只有一个非终结符的语法，称为上下文无关（context-free）语法。（上下文相关和上下文无关语法的准确理论定义要比这些定义更为严格。为简单，本章使用前面的这种定义，不过我们应该知道该理论更严格的描述并不像我们定义的那样简单。）

abc、aabbcc 和 aaaabbbbcccc 是这种语法描述语言的一些合法字符串的例子。而 aabc 和 cba 是非法字符串的例子。可以试着推导出这些合法字符串，也可以尝试推导这些非法字符串以证明它们是不合法的。用这些规则做一些练习，你应该能了解这种语言是以一个或多个 a 开始，后面跟同样数量的 b，后面再跟同样数量的 c 的字符串集合。这种语言 L 用数学语言来表达为

$$L = \{a^n b^n c^n \mid n > 0\}$$

读作“语言 L 是字符串 $a^n b^n c^n$ 的集合，n 大于 0”。符号 a^n 表示 n 个 a 的连接。

7.1.7　语法分析问题

从语法推导出合法字符串是非常简单明了的。可以在当前中间字符串的右边任意选择一些非终结符，反复选择规则进行替换，直到得到一个终结符字符串。这样的随机推导可以得到该语言的许多字符串。

然而，自动翻译器是一项更难的任务。给翻译器一个终结符字符串，假设是某个人工语

言的合法句子。在翻译器生成目标代码前，它必须确定终结符字符串是否确实是合法的。确定字符串是否合法的唯一办法是从语法的初始符推导出它。翻译器必须尝试进行这样的推导，如果推导成功了就知道这个字符串是一个合法的句子。确定一个给定的终结符字符串是否是特定语法的合法句子的问题叫作语法分析（parsing），如图7-4所示。

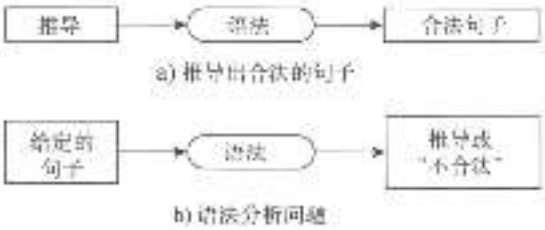


图7-4　推导出任意一个句子与分析一个给定的句子之间的区别

对给定的句子进行语法分析要比推导任意的合法字符串更难。语法分析问题就是某种形式的搜索。分析算法必须搜索出正确的替换序列，推导出给定的字符串。如果给定的字符串是合法的，它不仅要找到这个推导，如果给定的字符串不是合法的，它也必须能够发现。如果你在房间里寻找一枚钻石戒指，但是找不到，并不表示戒指不在你的房间里。可能只是你没找到对的地方。类似地，如果试图找到给定字符串的推导，但是找不到，你怎么知道推导不存在呢？翻译器必须能够证明如果给定的字符串是非法的，一定不存在对应的推导。

7.1.8　表达式的语法

为了了解语法分析器可能遇到的困难，考虑图7-5，它给出了描述算术表达式的语法。假设给定的终结符字符串为

```
( a * a ) + a
```

给定该语法的产生式规则，要求对给定的字符串进行分析。正确的语法分析是：

```
E ⇒ E + T              规则1
  ⇒ T + T              规则2
  ⇒ F + T              规则4
  ⇒ ( E ) + T          规则5
  ⇒ ( T ) + T          规则2
  ⇒ ( T * F ) + T      规则3
  ⇒ ( F * F ) + T      规则4
  ⇒ ( a * F ) + T      规则6
  ⇒ ( a * a ) + T      规则6
  ⇒ ( a * a ) + F      规则4
  ⇒ ( a * a ) + a      规则6
```

N = {E, T, F}
T = {+, *, (,), a}
P = the productions
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → a
S = E

图7-5　表达式的语法。非终结符E表示一个表达式，T表示一个术语，而F表示该表达式的一个因子

339
〈
340

语法分析困难的原因是可能在前面的分析中就做出了错误的决定，虽然在当时看上去是可行的，但是会导致进入死路。例如，看见给定的字符串里有"("，就立即选择规则5。你可能会尝试这样进行分析

```
E ⇒ T              规则2
  ⇒ F              规则4
  ⇒ ( E )          规则5
  ⇒ ( T )          规则2
  ⇒ ( T * F )      规则3
  ⇒ ( F * F )      规则4
  ⇒ ( a * F )      规则6
  ⇒ ( a * a )      规则6
```

到目前为止，看上去是在朝着分析该表达式的目标上前进，因为随着推导的每一步，中间字符串看上去都更像原始的字符串了。不幸的是，却陷入了困境，因为没有办法推导出原

始字符串中的 +。

在到达这个死胡同之后，我们可能会得出结论：该给定的字符串是非法的。但是这是错的。因为找不到一个推导并不意味着这样的推导不存在。

语法分析有趣的一面是它可以表示为一棵树。初始符是树根。树的每个内部结点是一个非终结符，每个叶子结点是一个终结符。内部结点的孩子是在推导中用来替换父亲结点的产生式规则右边部分的符号。这种树称为语法树，原因显而易见。图 7-6 给出了采用图 7-5 所示语法的 (a+a)+a 的语法树。图 7-7 给出了采用图 7-2 中语法的 dd 的语法树。

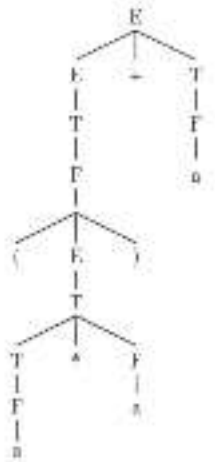


图 7-6　按照图 7-5 对 (a+a)+a 进行分析的语法树

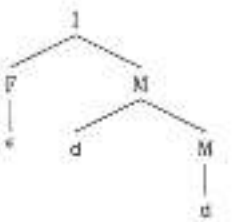


图 7-7　按照图 7-2 对 dd 进行分析的语法树

7.1.9　C++ 语法的一部分

图 7-8 中语法的产生式规则是 C++ 语言的一小部分。这种语言的基本数据类型只有整数和字符，不提供常量或类型定义，不允许引用参数，还省略了 switch 和 for 语句。尽管有诸多限制，但是它仍然给出了如何形式化定义一个真实语言语法的大致思路。

```
<translation-unit> →
    <external-declaration>
    | <translation-unit> <external-declaration>

<external-declaration> →
    <function-definition>
    |<declaration>

<function-definition> →
    <type-specifier> <identifier> ( <parameter-list> ) <compound-statement>
    |<identifier> ( <parameter-list> ) <compound-statement>

<declaration> → <type-specifier> <declarator-list> ;

<type-specifier> → void | char | int

<declarator-list> →
    <identifier>
    | <declarator-list> , <identifier>

<parameter-list> →
    ε
    |<parameter-declaration>
    |<parameter-list> , <parameter-declaration>
```

图 7-8　C++ 语言的一部分语法

```
<parameter-declaration> → <type-specifier> <identifier>

<compound statement> → { <declaration-list> <statement-list> }

<declaration-list> →
    ε
  | <declaration>
  | <declaration-list> <declaration>

<statement-list> →
    ε
  | <statement>
  | <statement-list> <statement>

<statement> →
    <compound-statement>
  | <expression-statement>
  | <selection-statement>
  | <iteration-statement>

<expression-statement> → <expression> ;

<selection-statement> →
    if ( <expression> ) <statement>
  | if ( <expression> ) <statement> else <statement>

<iteration-statement> →
    while ( <expression> ) <statement>
  | do <statement> while ( <expression> ) ;

<expression> →
    <relational-expression>
  | <identifier> = <expression>

<relational-expression> →
    <additive-expression>
  | <relational-expression> < <additive-expression>
  | <relational-expression> > <additive-expression>
  | <relational-expression> <= <additive-expression>
  | <relational-expression> >= <additive-expression>

<additive-expression> →
    <multiplicative-expression>
  | <additive-expression> + <multiplicative-expression>
  | <additive-expression> - <multiplicative-expression>

<multiplicative-expression> →
    <unary-expression>
  | <multiplicative-expression> * <unary-expression>
  | <multiplicative-expression> / <unary-expression>

<unary-expression> →
    <primary-expression>
  | <identifier> ( <argument-expression-list> )

<primary-expression> →
    <identifier>
  | <constant>
  | ( <expression> )

<argument-expression-list> →
    <expression>
  | <argument-expression-list> , <expression>

<constant> →
    <integer-constant>
  | <character-constant>
```

图 7-8 （续）

```
<integer-constant> →
    <digit>
    | <integer-constant> <digit>

<character-constant> → ' <letter> '

<identifier> →
    <letter>
    | <identifier> <letter>
    | <identifier> <digit>

<letter> →
    a | b | c | d | e | f | g | h | i | j | k | l | m
    n | o | p | q | r | s | t | u | v | w | x | y | z
    A | B | C | D | E | F | G | H | I | J | K | L | M
    N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<digit> →
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

图 7-8 （续）

该语法的非终结符用尖括号 <> 括起来。任何没有用尖括号括起来的符号是终结字符，可以出现在 C++ 程序代码中。该语法的初始符是非终结符 <translation-unit>（<翻译单元>）。

用语法的产生式规则来描述编程语言的方法称为巴科斯范式（Backus Naur Form，BNF）。在 BNF 中，产生式符号 > 有时写作 ::=。设计于 1960 年的语言 GLGOL-60 使得 BNF 流行了起来。

下面这个示例是该语法的分析，表明如果 S1 是一个合法的 <statement>（<语句>），那么

```
while (a <= 9)
  S1;
```

也是合法的 <statement>。该语法分析由下述推导组成：

```
<statement>
  → <iteration-statement>
  → while ( <expression> ) <statement>
  ⇒ while ( <relational-expression> ) <statement>
  ⇒ while ( <relational-expression> <= <additive-expression> ) <statement>
  → while ( <additive-expression> <= <additive-expression> ) <statement>
  → while ( <multiplicative-expression> <= <additive-expression> ) <statement>
  ⇒ while ( <unary-expression> <= <additive-expression> ) <statement>
  ⇒ while ( <primary-expression> <= <additive-expression> ) <statement>
  ⇒ while ( <identifier> <= <additive-expression> ) <statement>
  → while ( <letter> <= <additive-expression> ) <statement>
  → while ( a <= <additive-expression> ) <statement>
  → while ( a <= <multiplicative-expression> ) <statement>
  → while ( a <= <unary-expression> ) <statement>
  ⇒ while ( a <= <primary-expression> ) <statement>
  ⇒ while ( a <= <constant> ) <statement>
  ⇒ while ( a <= <integer-constant> ) <statement>
  → while ( a <= <digit> ) <statement>
  → while ( a <= 9 ) <statement>
  → while ( a <= 9 ) <expression-statement>
  ⇒ while ( a <= 9 ) <expression> ;
  →* while ( a <= 9 ) S1;
```

图 7-9 给出了这个分析对应的语法树。非终结符 <statement> 是树根，因为这个分析的目的是要证明该字符串是一个合法的 <statement>。

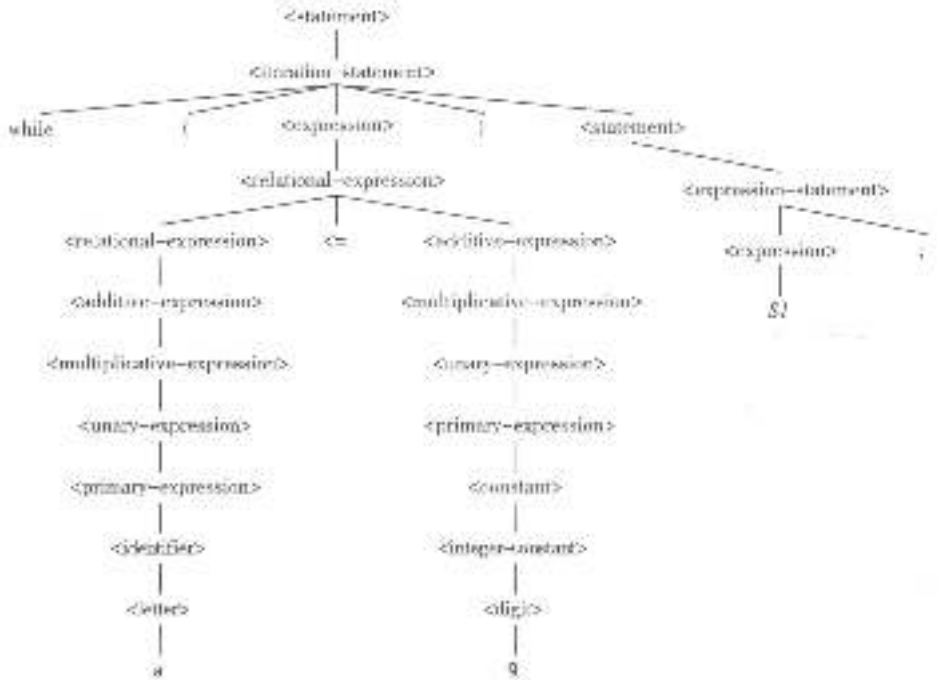


图 7-9　图 7-8 中
```
while a<=9
    s1;
```
的语句分析的语法树

记住这个例子，思考 C++ 编译器的任务。将编译器编写为包含一组产生式规则，类似于图 7-8 那样的规则。程序员向编译器提交一个包含源程序的文本文件，就是一个长长的终结符字符串。首先，编译器必须确定这个终结符字符串表示的是不是一个合法的 C++ <translation-unit>。如果是，那么编译器就必须生成相应的低级语言表示的目标代码。如果不是，编译器必须产生适当的语法错误提示。

标准 C++ 语法中有上百条产生式规则。想象 C++ 编译器的工作量吧！每次提交程序时，它都要在这些产生式规则中排序遍历。幸运的是，计算机理论发展至今已经能够使编译器的语法分析不太困难。使用当前理论设计出的 C++ 编译器保证在程序分析中推导的每一步替换都能够选择正确的产生式。如果分析算法不能找到匹配源程序的 <translation-unit> 的推导，就能够证明这样的推导是不存在的，提交的源程序一定有语法错误。

对编译器来说，代码生成比语法分析更难一些。究其原因是目标代码必须运行在某个制造厂商生产的某种特定的机器上。因为每个厂商的机器架构都不同，指令集也不同，所以一种机器的代码生成技术可能不适用于另一种机器。不存在一种单一标准的基于理论概念的冯·诺依曼架构。这就导致没有出现太多关于代码生成的理论能够指导编译器设计者构建编译器。

341 | 345

7.1.10　C++ 的上下文相关性

从图 7-8 来看 C++ 语言貌似是上下文无关的。每个产生式规则的左边都只有一个非终结符。这与图 7-3 所示的上下文有关语法是不同的，上下文无关语法中允许产生式左边有多于一个的非终结符。外表看上去是很有欺骗性的。虽然 C++ 语法的这个子集和完整的 C++ 语言标准都是上下文无关的，但是 C++ 语言本身的某些方面是上下文相关的。

来看一看图 7-3 中的语法。它的产生式规则怎么保证字符串结尾的 c 的个数必须等于开头 a 的个数？规则 1 和 2 保证每生成一个 a，都会生成一个 C。规则 3 让 C 变换到 B 的右边。最后，规则 5 允许在 C 的左边有 b 的情况下，用 c 替代 C。没有办法用上下文无关语法描述这个语言，因为它需要规则 3 和 5 让 C 到字符串的末尾。

C++ 语言有上下文相关的一面，这在图 7-8 中没有表现出来。例如，<parameter-list>（<形参列表>）的定义允许任意数量的形参，而 <argument-expression-list>（<实参表达式列表>）的定义允许任意数量的实参。可以写一个 C++ 程序包含一个具有 3 个形参的过程，以及一个有 2 个实参的过程调用，图 7-8 中的语法可以从 <translation-unit> 推导出来。但是如果想编译这个程序，编译器会报一个语法错误。

C++ 中形参的数量必须等于实参的数量，类似于图 7-3 中语法定义的字符串开头时的 a 的个数必须等于结尾处 c 的个数。要想对 C++ 的语法做出这个限制，就需要包括一些更为复杂的、上下文相关的规则，让编译器用上下文无关语法分析一个程序，之后再检查是否违反一些规则，这样更简单一些。通常借助于符号表来检查是否有违反这些语法不能描述的规则。

7.2　有限状态机

另一种描述语言中句子语法的方式是有限状态机。以图的形式表示，有限状态机就是一个状态的有限集合，状态用圆圈表示，称为结点。状态间的转换用两圈之间的弧表示。每条弧从一个状态开始，到另一个状态结束，在结束状态有一个箭头。每条弧上都还有一个标号，是该语言终结符表中的一个字符。

有限状态机中的一个状态被指定为起始状态，至少一个，也可以多个状态被指定为终止状态。在图中，起始状态有一个输入箭头，而终止状态用双圈表示。

346

数学上，这样一组以弧连接的结点称作图。当弧有方向时，FSM 就是这样，称为有向图（directed graph 或 digraph）。

7.2.1　用 FSM 来分析标识符

图 7-10 给出了一个 FSM，它分析图 7-1 中语法定义的标识符。状态集合是 {A, B, C}，A 是起始状态，B 是终止状态。对一个字母有 A 到 B 的转换，对一个数字有 A 到 C 的转换，对一个字母或数字有 B 到 B 的转换，对一个字母或数字有 C 到 C 的转换。

要使用 FSM，可以假想输入字符串写在一张纸带上。从起始状态开始，从左至右扫描输入纸带上的字符，每次从纸带上扫描到下一个字符时，就转移到有限状态机的下一个状态上。只能使用弧上标识的字符与扫描进来的字符一致的那些转换。在扫描所有的输入字符之后，如果字符在终止状态中，那么这个字符串就是一个合法的标识符，否则就不是。

例 7.4　要分析字符串 cab3，需要做下面这样的转换：

当前状态：A	输入：cab3	扫描 c，转移到状态 B。
当前状态：B	输入：ab3	扫描 a，转移到状态 B。
当前状态：B	输入：b3	扫描 b，转移到状态 B。
当前状态：B	输入：3	扫描 3，转移到状态 B。
当前状态：B	输入：	检查是否是终止状态。

因为没有输入了，所以最后的状态是 B，是终止状态，cab3 就是一个合法的标识符。

还可以用状态转换表来表示 FSM。图 7-11 是图 7-10 的 FSM 的状态转换表，表中列出

了从一个给定的当前状态在给定输入符号时转换到的下一个状态。

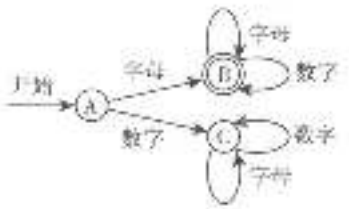


图 7-10　分析标识符的有限状态机（FSM）

当前状态	下一个状态	
	字　母	数　字
→A	B	C
ⓑ	B	B
C	C	C

图 7-11　图 7-10 的 FSM 的状态转换表

7.2.2　简化的有限状态机

通常通过消除 FSM 中的一些状态，对图进行简化会更方便。比如删除那种存在的唯一目的就是为非法输入字符提供转移的状态。这个状态机中的状态 C 就是这么一种状态。一旦转移到状态 C，就永远没办法转移到其他的状态，输入字符串最终会被认定为非法。图 7-12 给出了图 7-10 的 FSM 的简化版本，没有了失败状态。

用这个简化的状态机分析字符串时，在输入字符串中遇到非法字符时，没有办法进行转移。有两种方法能在简化的状态机中检测出非法句子：

- 没有输入，但是却不在一个终止状态中。
- 在某个状态中，但是该状态没有与下一个输入字符相对应的转移。

图 7-13 是图 7-12 对应的状态转换表。对于去掉的转移，简化状态机对应的状态转换表中没有对应的条目。注意，这张表在当前状态 A 的数字一栏中是没有内容的。本章余下部分中的状态机都使用简化的格式。

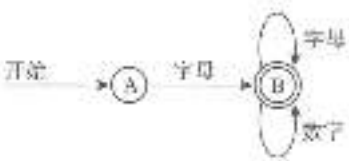


图 7-12　图 7-10 的不带错误状态的 FSM

当前状态	下一个状态	
	字　母	数　字
→A	B	
ⓑ	B	B

图 7-13　图 7-12 的 FSM 的状态转换表

7.2.3　非确定性有限状态机

当使用语法分析句子时，对于一个非导步骤通常必须在多个产生式规则中选择用哪个进行替换。类似地，非确定性有限状态机要求在分析输入字符串时从多个转移中选择。图 7-14 是一个分析有符号整数的非确定性 FSM。称为非确定，因为至少有一个状态对于一个字符有多于一种转移。例如，状态 A 对于一个数字可以转移到 B 也可以转移到 C。对于状态 B 也有一些不确定性，如果下一个输入字符是一个数字，转移到 B 或者 C 都有可能。

例 7.5　用这个非确定性 FSM 来分析 +203，必须要做下面这样一些决策：

当前状态：A	输入：+203	扫描 +，转移到状态 B。
当前状态：B	输入：203	扫描 2，转移到状态 B。
当前状态：B	输入：03	扫描 0，转移到状态 B。
当前状态：B	输入：3	扫描 3，转移到状态 C。
当前状态：C	输入：	检查是否是终止状态。

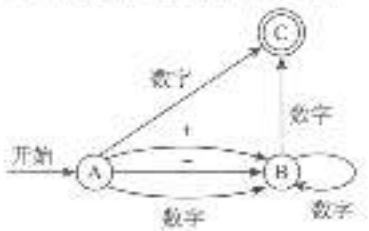
因为没有输入又处于终止状态 C 中，所以这证明了输入字符串 +203 是一个合法的有符

号整数。 □

在用产生式规则进行分析时，有出现前面的分析可能做出不正确选择的风险。可能进入一个死胡同，没有替换能使得终结符和非终结符组成的中间字符串更接近给定的字符串。进入这样的死胡同并不一定就表示该字符串是非法的。所有的非法字符串在任何尝试中都会进入死胡同，但是如果在前面的推导中决策错误，合法的字符串也有可能产生死胡同。

对于非确定性有限状态机，同样有这个问题。对于图 7-14 中的状态机，如果在起始状态 A 中，下一个输入字符是 7，那就必须在转移到 B 还是 C 之间选择。假设选择转移到 C，然后发现还有一个输入字符要扫描。因为从 C 没有转移出来，所以这次分析尝试进入了死胡同。因此你得出结论，输入字符串是非法的，还是合法的？只是在前面某个地方做出了错误的选择。

图 7-15 是图 7-14 中状态机的状态转换表。非确定性很明显，数字栏中有多项（B，C），它们表示在尝试分析时需要进行选择。



当前状态	下一个状态		
	+	-	数　字
→A	B	B	B，C
D			D，C
Ⓒ			

图 7-14　一个用来分析有符号整数的非确定性的 FSM　　图 7-15　图 7-14 的 FSM 的状态转换表

7.2.4　具有空转移的状态机

在产生式规则中引入空字符串会带来方便，同样，构建具有对空字符串转移的有限状态机也会带来方便。这样的转移称为空转移。图 7-16 是图 7-17 中的 FSM 的状态转换表。图 7-17 是一个 FSM，对应于图 7-2 中的语法，用来分析有符号整数。

当前状态	下一个状态			
	+	-	数　字	ε
→I	F	F		F
F			M	
Ⓜ			M	

图 7-16　图 7-17 的 FSM 的状态转换表

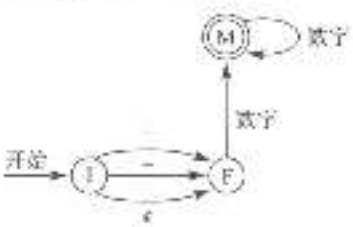


图 7-17　一个用来分析有符号整数的，带空转移的 FSM

在图 7-17 中，F 是输入第一个字符后的状态，M 是数值状态，类似于语法中的非终结符 F 和 M。同样，符号可以是 -、+ 或者两者都不是，从 I 到 F 的转移可以接受 +、- 或 ε。

例 7.6　分析 32，做出如下决策：

当前状态：I	输入：32	扫描 ε，转移到状态 F。
当前状态：F	输入：32	扫描 3，转移到状态 M。
当前状态：M	输入：2	扫描 2，转移到状态 M。
当前状态：M	输入：	检查是否是终止状态。

接受 ε 从 I 到 F 的转移不消耗输入字符。当处于状态 I 时，可以做 3 件事情中的一件：(a) 扫描 +，进入状态 F；(b) 扫描 -，进入状态 F；或者 (c) 什么也不扫描（也就是，空字

符串），进入状态F。 □

具有空转移的状态机总被认为是非确定性的。在例7.6中，不确定性来自当在状态I中
347 ~ 349 面下一个字符是+时必须做出决策，必须决定是从I到F接受+，还是从I到F接受 ε。这是不同的转移，因为虽然它们转移到同一个状态，但是剩下的输入字符串是不同的。

给定一个带有空转移的FSM，总是可以把它变换为一个等价的、不带有空转移的有限状态机。消除空转移的算法分为两步：

- 如果有一个转移接受 ε 从p到q，对于每个接受a从q到r的转移，增加一个接受a从p到r的转移。
- 如果q是一个终止状态，把p也变成终止状态。

这个算法使用 ε 的连接属性：

$$\varepsilon a = a$$

例7.7　图7-18展示了如何从（a）中的状态机中消除空转移，得到等价的（b）中的状态机。因为有一个接受 ε 从状态X到状态Y的转移，以及一个接受a从状态Y到状态Z的转移，所以增加一个接受a从状态X到状态Z的转移，就可以消除这个空转移了。如果在状态X中，可以直接接受a进入状态Z，这和直接接受 ε 从X经过Y到达Z，进入的状态和剩下的输入都是一样的。 □



a）原始的FSM　　b）不带空转移的等价的FSM

图7-18　消除空转移

例7.8　图7-19给出了对图7-17的FSM的变换。从I到F的空转移被替换为接受数字的从I到M的转移，因为有一个接受数字从F到M的转移。 □

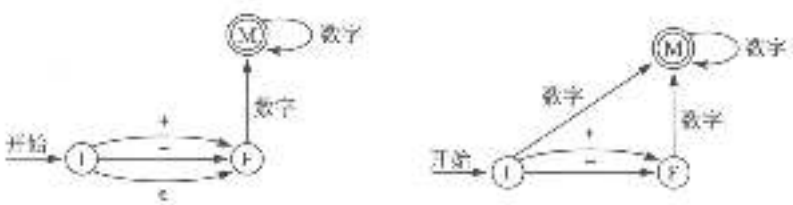


图7-19　消除图7.17的FSM中的空转移

在例7.8中从F到M只有一个转移，所以从I到F的空转移只用替换为一个转移。如果一个FSM有多个转移来自空转移的目标状态，那么在消除空转移时，就必须添加多条转移。

例7.9　要消除图7-20a中W到X的空转移，需要用两个转移来替换它，一个是接受a的从W到Y，另一个是接受b的从W到Z。在这个例子中，因为X是图7-20a的终止状态，所以W就成为图7-20b中等价状

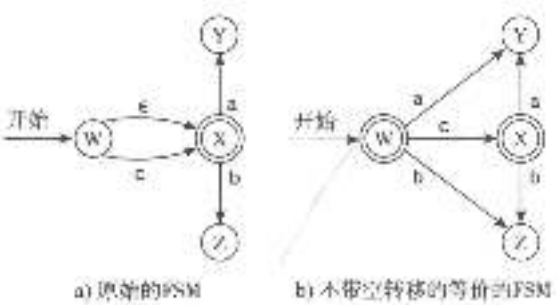


图7-20　消除空转移

态机的一个终止状态，与算法中的第二步保持一致。 □

图 7-17 消除了空转移就得到一个确定性的状态机。不过，一般来说，消除所有的空转移并不能保证得到的 FSM 就是确定性的。虽然所有带有空转移的状态机都是非确定性的，但是不带有空转移的仍然有可能是非确定性的。图 7-14 就是这样一个例子。

如果有选择的话，用确定性的 FSM 分析总比用非确定性的 FSM 好。使用确定性的状态机，对于合法的输入字符串是不可能做出错误的选择而进入死胡同的。一旦出现终止在死胡同里的情况，就可以得出结论该输入字符串是非法的。

计算机科学家已经证明每个非确定性的 FSM 都有一个等价的确定性的 FSM，也就是说，有一个确定性的状态机能够识别完全一样的语言。不过对这个很有用结论的证明超出了本书的范围，证明的大概思路就是说明如何从一个非确定性的状态机构造出等价的确定性的状态机。

7.2.5 语言符号识别器

语言符号（token）是一个终结符的字符串，作为一个整体它有特殊的含义。这组字符通常对应于语言语法中的某个非终结符。例如，看看下面这个 Pep/8 汇编语言：

```
mask: .WORD 0X00FF
```

这个语句中的 token 是 mask:、.WORD、0X 和 00FF。每个字符组都来自于汇编语言符号表，组合到一起具有特殊的含义。这里每个 token 的含义分别是符号定义、点命令、十进制数指示符和十进制数值。

从某种程度上说，可以任意选择哪个符号组文表示哪个 token。例如，可以选择字符组 0X00FF 表示十进制数 token。通常选择使得 FSM 的实现尽可能简单的 token 字符。

350 ↑ 351

在翻译器中 FSM 常见的使用是在源字符串中识别 token。考虑汇编器在收到源字符串时的工作。假设汇编器已经确定 mask: 是符号定义，.WORD 是点命令，并且知道点命令后可以是一个十进制或十六进制常数，所以编程必须接受它们。需要 FSM 能够确认出两者。

图 7-21a 给出了分析十六进制常数前缀和无符号整数的两个状态机。C 是第一个检测 0x 的状态机的终止状态，E 是第二个检测无符号整数的状态机的终止状态。

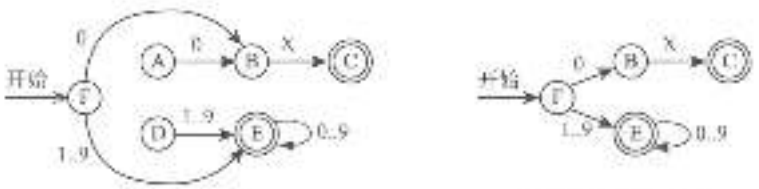


a) 两个独立的识别0x和无符号整数token的状态机　　b) 一个识别0x或无符号整数token的非确定性的FSM

图 7-21　把两个状态机合并得到一个能够识别两种 token 的 FSM

要构建一个能够识别 0x 和无符号整数的 FSM，首先要画一张新的合并后的状态机的起始状态，在这个例子中是 F。然后再画出从新的起始状态到每个单独状态机中起始状态的空转移，在本例中是 F 到 A 和 F 到 D。结果就得到一个可以识别两种 token 的非确定性的 FSM。结束的终止状态表明已经识别出了该 token。分析完成后，如果终止在状态 C，就表明识别出的是 0x，如果终止在状态 E 中，就表明识别出的是无符号整数。

要想让这个状态机的形式能更方便使用，就要消除空转移。图 7-22a 给出了图 7-21b 中 FSM 消除空转移后的状态机。消除空转移之后，状态 A 和 D 是不可达的。也就是说，无论输入字符串是什么，永远无法从起始状态到达这些状态。所以它们不会影响到分析，可以从状态机中去除，得到图 7-22b 所示的图。



a) 消除空转移　　b) 消除不可达状态

图 7-22　对图 7-21b 的变换

下面是另一个例子，说明了翻译器需要识别出多个 token。思考在遇到下面这样两行代码时汇编器的工作：

```
WHILE:  DEA R5,1,R5 ;comment 1
352]    WHILE       ;comment 2
```

Michael O. Rabin 和 Dana S. Scott

Michael O. Rabin 和 Dana S. Scott 分别于 1956 年和 1958 年在普林斯顿大学获得博士学位，在 Alonzo Church（1903—1995）门下学习。Church 是普林斯顿大学最有影响力的一位数学教授，在计算机发明以前，他就在研究基础的数学逻辑，他的工作对计算机领域有很大远的影响。他也是很多在计算机科学领域卓有建树的科学家的研究生导师。Alan Turing 就是 Church 的学生，在他的指导下于 1938 年获得博士学位。其他一些学生包括 John Kemeny，BASIC 编程语言的发明者之一；Stephen Kleene，发现了 Kleene 定理，该定理说明了有限状态机和正则表达式集合之间的等价性。





1931 年，Rabin 生于 Breslau，当时属于德国，现在是波兰的一座城市。虽然 Rabin 的父亲是一位拉比（rabbi，犹太教的法师），但是他想上以科学著名的学校，而不是宗教性质的高中。他说服了父亲让他学习科学，并最终在 20 世纪 50 年代初期进入希伯来大学。那里，他阅读了 Kleene 的著作《Introduction to Metamathematics》（元数学导论）。这本著作中有一章是关于可计算性和图灵机的，它影响了 Rabin 使他致力于计算机科学的基础工作。

1932 年 Scott 生于加利福尼亚州伯克利。青年时代就着迷于数学，进入加利福尼亚大学伯克利分校决心主修数学。获得学士学位后进入普林斯顿大学攻读博士学位，也是数学专业。Scott 在普林斯顿遇到了 Rabin，他认为 Rabin 是一个总是非常活跃且充满想法的人。虽然 Rabin 早 Scott 几年毕业，但是他们俩在普林斯顿之后仍然继续合作。

1957 年的夏天，Dana Scott 和 Michael Rabin 都在 IBM 研究中心工作。因为经典的图灵机被认为内存是无限的，而实际机器的内存都是有限的，所以图灵机作为有限内存的模型，并不是实际机器的准确模型。Scott 和 Rabin 提出了非确定性的有限状态机的概念，并研究了它的属性。因为这种机器的状态数量是有限的，所以它的内存也是有限的。他们证明了本章提到的一个结论，即每个非确定性的 FSM 都存在一个等价的确定性的 FSM。

因为他们 1959 年共同发表的论文 "Finite Automata and their Decision Problem"（有限自动机和它们的决策问题），该论文基于他们在 IBM 时的合作工作，Dana Scott 和 Michael Rabin 在 1976 年获得了 A. M. 图灵奖。

Dana Scott 现在是卡内基梅隆大学计算机、哲学和数学逻辑（名誉退休）Hillman University 教授。Rabin 是哈佛大学计算机科学杰出 Thomas J. Watson Sr. 教授，也是耶路撒冷希伯来大学的客座教授。他的另一项成就是设计出了一个连够在极小出错概率下快速找到极大素数的算法，这也是下面这句引言的出处：

"我们应该放弃获得完全确定性的独立和答案的尝试。"

——Michael O. Rabin

第一行上第一个 token 是符号定义，第二行上第一个 token 是一元指令的助记符。在每一行的开头，翻译器需要 FSM 能够识别出符号定义（它的形式是标识符后面跟一个分号）或助记符（它的形式是标识符）。图 7-23 就是这样一个多 token FSM。

对于第一行，这个状态机做出如下转移：

接受 N，从 A 到 B

接受 O，从 B 到 B

接受 T，从 B 到 B

接受 E，从 B 到 B

接受 :，从 B 到 C

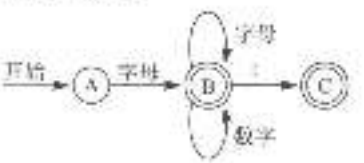


图 7-23　分析 Pep/8 汇编语言标识符或符号定义的 FSM

此后，翻译器知道它识别出了一个符号定义。对于第二行，它做出如下转换：

接受 N，从 A 到 B

接受 O，从 B 到 B

接受 T，从 B 到 B

接受 A，从 B 到 B

因为接下来的输入符号不是分号，所以 FSM 不会转移到状态 C。此时，翻译器知道它识别出了一个标识符，因为终止状态是 B。

7.3　实现有限状态机

编程语言的语法通常用形式语法来描述，这是翻译器分析算法的基础。不像图 7-8 中的语法那样描述所有的句法，形式语法通常只描述较高层次的抽象，把低层次留给正则表达式或有限状态机来描述。

图 7-24 描述了一个典型编译过程中的步骤。低层次的句法分析称为词法分析（lexical analysis），高层次的句法分析称为语法分析（parsing，这个词有时也宽泛地用来包括所有的句法分析）。在大多数人工语言的翻译器中，词法分析器基于确定性的 FSM，输入是一个字符串。语法分析器通常基于语法，输入是来自词法分析器的 token 序列。

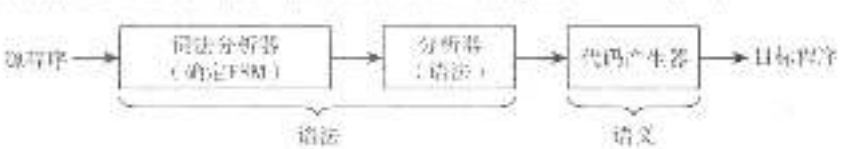


图 7-24　编译过程中的步骤

词法分析器的非终结符是语法分析器的终结符。这样的符号的一个常见例子是标识符。词法分析器的 FSM 的终结符表包括单个的字母和数字，输入是这些字符组成的字符串，然后进行状态转换。如果输入是字符串 abc3，那么 FSM 声明说识别出了一个标识符，并且把该信息传递给语法分析器。语法分析器在分析语言的句子时，会把 <identifier>（<标识符>）作为终结符。

当设计对输入进行语法分析的软件时，描述规则有时不是以 FSM 和语法的方式给出的。不过如果输入的结构不太复杂，可能可以把词法分析和语法分析结合起来，直接从问题的描述画出 FSM 来分析语法。如果 FSM 是非确定性的，则需要把它转化成等价的确定性的 FSM。在画出确定性的 FSM 之后，就可以用程序来实现它。

较为复杂的结构，比如翻译高级语言的编译器，通常都需要用形式语法来描述。通常不能只用一个 FSM 就分析这样一种语言的句法。相反，必须把句法分析分为多个阶段，如图 7-24 所示。使用更高级的技术超出了本书讲授的范围。

实现 FSM 的算法有一个枚举型变量，称为状态变量（state variable），它的可能值对应于 FSM 的可能状态。算法把状态变量初始化为机器的起始状态，在循环的每一次中获取终结符字符串的一个字符。每个字符都会导致状态的一次改变。有两种常见的实现技术：

- 查找表
- 直接编码

在这两种方法中，状态变量获取它下一个值的方式不同。查找表技术存储状态转换表，根据当前状态和输入字符查表获得下一个状态。直接编码技术在代码中检测当前状态和输入字符，直接把下一个状态赋值给状态变量。

7.3.1　查找表分析器

353 / 355

图 7-25 中的程序用查表技术实现了图 7-10 的 FSM。变量 FSM 是图 7-11 所示的状态转换表。程序把输入字符分成字母或数字。因为 B 是终止状态，所以如果循环结束时状态是 B，那么程序会声称输入字符串是一个合法的标识符。

```
#include <iostream>
#include <cctype> // isalpha
#include <string> // string
using namespace std;

enum State {eA, eB, eC};
enum Alphabet {eLETTER, eDIGIT};
State FSM[eC + 1][eDIGIT + 1] = {  // Three rows, two columns
   eB, eC,  // The state transition table of Figure 7.11
   eB, eB,
   eC, eC
};

int main () {
   char ch;
   string line;
   Alphabet FSMChar;
   State state = eA;
```

图 7-25　用查找表技术实现图 7-10 的 FSM

```
    cout << "Enter a string of letters and digits: ";
    cin >> line;
    for (int i = 0; i < line.size (); i++) {
       ch = line[i];
       if (isalpha (ch)) {
          FSMChar = aLETTER;
       }
       else {
          FSMChar = aDIGIT;
       }
       state = FSM[state][FSMChar];
    }
    if (state == aR) {
       cout << line << " is a valid identifier." << endl;
    }
    else {
       cout << line << " is not a valid identifier." << endl;
    }
    return 0;
}
```

输入/输出

```
Enter a string of letters and digits: cab3
cab3 is a valid identifier.
```

输入/输出

```
Enter a string of letters and digits: 3cab
3cab is not a valid identifier.
```

图 7-25 （续）

该程序假设用户只会输入字母和数字。如果用户输入其他的字符，程序会把它当作数字。例如，如果用户输入 cab#，程序会认为它是个合法的标识符，而实际上它不是。在本章末尾，留给读者的一个问题就是改进变量 FSM 并实现它。

7.3.2 直接编码分析器

图 7-26 中的程序使用直接编码技术来分析整数。函数 parseNum 允许用户输入任意字符串。如果该字符串不是合法的整数，那么 parseNum 为 valid 返回 false，程序会发出一条错误消息。否则，valid 为 true，num 是输入整数的正确的值。

```
#include <iostream>
#include <cctype> // isdigit
#include <string> // string, getline
using namespace std;

// Global buffer
string line;
int lineIndex;

void getLine () {
   // Get a line of characters.
   // Install a newline character as sentinel.
```

图 7-26 程序员设计的整数串的语法分析

```
        getline (cin, line);
        line.push_back ('\n');
        lineIndex = 0;
    }

    enum State {eI, eF, eM, eSTOP};

    void parseNum (bool& v, int& n) {
        int sign;
        State state;
        char nextChar;
        v = true;
        state = eI;
    do {
        nextChar = line[lineIndex++];
        switch (state) {
            case eI:
                if (nextChar == '+') {
                    sign = +1;
                    state = eF;
                }
                else if (nextChar == '-') {
                    sign = -1;
                    state = eF;
                }
                else if (isdigit (nextChar)) {
                    sign = +1;
                    n = nextChar - '0';
                    state = eM;
                }
                else {
                    v = false;
                }
                break;
            case eF:
                if (isdigit (nextChar)) {
                    n = nextChar - '0';
                    state = eM;
                }
                else {
                    v = false;
                }
                break;
            case eM:
                if (isdigit (nextChar)) {
                    n = 10 * n + nextChar - '0';
                }
                else if (nextChar == '\n') {
                    n = sign * n;
                    state = eSTOP;
                }
                else {
                    v = false;
                }
```

图 7-26　（续）

```
            break;
        }
    }
    while ((state != eSTOP) && v);
}
int main () {
    bool valid;
    int num;
    cout << "Enter number: ";
    getLine ();
    parseNum (valid, num);
    if (valid) {
        cout << "Number = " << num << endl;
    }
    else {
        cout << "Invalid entry." << endl;
    }
    return 0;
}
```

输入/输出

```
Enter a number: Q
Invalid entry.
```

输入/输出

```
Enter a number: -58
Number = -58
```

图 7-26 （续）

输入函数 getLine 从键盘读入字符到一个字符串，无论用户输入多少个字符，总是会以换行符作为分隔。如果用户没有输入字符，只是输入了一个回车键，getLine 会把换行符放在 line[0]。

函数 parseNum 对应于图 7-19b 的 FSM。这个过程有一个局部枚举型变量 state，它的可能值为 eI，eF 或 eM，对应于这个 FSM 的状态 I、F 和 M。另外，还有一个状态称为 eSTOP，是用来终止循环的。形参 v 对应于主程序中的实参 valid。函数会把 v 初始化为 true，把 state 初始化为启始状态 eI。

do 循环模拟有限状态机中的转移，使用直接编码技术。switch 语句确定当前状态，每种情况中嵌套的 if 语句确定下一个字符，而代码中的赋值语句直接改变状态变量。

在简化的 FSM 中，有两种方式来停止程序——要么输入完处理，要么到达了一个状态，它对于下一个字符没有对应的转移。在后一种情况中，输入字符串是非法的。对应于这两种停止条件，有两种方式来退出 do 循环——当到达输入分隔符且当前状态为终止状态时，或者发现该字符串是非法的时。

do 循环体至少被执行一次。即使回车键是第一个被按下的键，该代码也会正确执行。getLine 把换行符放在 line[0]，parseNum 把 state 初始化为 I，进入 do 循环，立即把 nextChar 设置为换行符。然后 v 置为 false，循环正确终止。

除了确定输入字符串是否合法外，parseNum 还把字符串转换为适当的整数值。如果第一个字符是 + 或者数字，sign 置为 +1。如果第一个符号是 -，sign 置为 -1。在 I 或 F 状

356 : 359

态中检测到的第一个数字会把 n 设置为对应的值。每次检测到后续的数字时，n 的值会适当地变化。如果循环结束时判定是一个合法的数字，那么这个数值会被以 sign。

计算正确的整数值是一个语义行为，而状态分配是一个句法行为。采用直接编码方式能够比较容易地在句法处理中加入语义处理，因为在句法代码中有明确的位置可以包含需要的语法处理。例如，在状态 1 中，如果字符为 -，sign 必须设置为 -1，很容易确定在语句代码中的哪个位置放入这样的赋值语句。

如果用户在合法的数字字符串前加上了一些空格，那么 FSM 会认为该字符串非法。接下来的这个程序会展示如何纠正这个不足。

7.3.3　输入缓冲区类

接下来的两个程序使用同样的技术从输入流获取字符。为了不在两个程序中重复输入处理的代码，本节展示了一个输入缓冲区类的实现，这两个程序都可以使用。这个类的实现存储在一个名为 InBuffer.hpp 的单独文件中，每个程序中都用 #include 语句把它包含进来。图 7-27 给出了这个称为头文件的 .hpp 文件。

正如在接下来两个程序中展示的那样，FSM 函数有时会发现来自输入流的一个字符终结了当前的 token，而对该函数接下来的调用中输入流还会需要它。从概念上说，这个函数必须把这个字符退回到输入流，这样接下来的调用才能再获取到它。backUpInput 提供了对缓冲区类的这样一种操作。虽然 FSM 函数需要访问来自输入缓冲区的字符，但是它并不直接访问缓冲区。只有过程 getLine、advanceInput 和 backUpInput 访问全局缓冲区。这样设计的目的是为了向 FSM 函数提供一种更方便的输入流抽象结构。

```
// File: InBuffer.hpp

#include <string> // string, getline

class InBuffer {
private:
   string line;
   int lineIndex;

public:
   void getLine () {
      getline (cin, line);
      line.push_back ('\n');
      lineIndex = 0;
   }

   void advanceInput (char& ch) {
      ch = line[lineIndex++];
   }

   void backUpInput () {
      lineIndex--;
   }
};
```

图 7-27　包含在图 7-29 和图 7-32 的程序中的输入缓冲区类

7.3.4　多 token 分析器

如果 C++ 编译器的语法分析器在分析字符串

```
total -
```

时它知道接下来的非终结符会是像 amount 这样的标识符或者像 100 这样的整数。因为它不知道接下来到底是哪种 token，所以它请求如图 7-28 所示的，能够识别两种 token 的有限状态机。

标号为 Ident 的状态是识别标识符 token 的终止状态，Int 是识别整数的终止状态。从 Start 到 Start 的转移接受的是空字符，也就是说，允许在两种 token 之前有空格。如果剩下的字符是行尾的空格

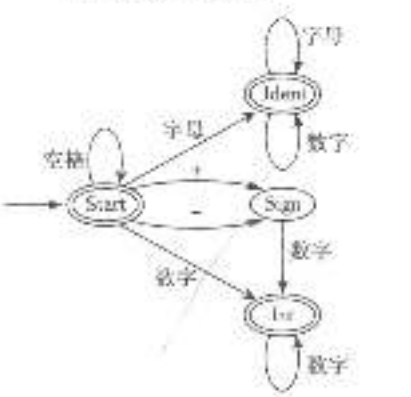


图 7-28　识别标识符和整数的程序的 FSM

符，那么 FSM 过程会返回空 token，这也正是为什么启始状态也是终止状态的原因。

图 7-29 展示了一个实现了图 7-28 中多 token 识别器的程序两次运行的输入 / 输出。第一次运行有两行输入，第一行是 5 个非空 token，第二行是 6 个非空 token。下面是对图 7-29 中第一次运行的解释。

```
Input
Here    is A47 48B
   C 49 ALongIdentifier +50 D16-51

Output
Identifier = Here
Identifier = is
Identifier = A47
Integer    = 48
Identifier = B
Empty token
Identifier = C
Integer    = -49
Identifier = ALongIdentifier
Integer    = 50
Identifier = D16
Integer    = -51
Empty token
```

```
Input
Here    is A47+ 48B
   C+49

ALongIdentifier

Output
Identifier = Here
Identifier = is
Identifier = A47
Syntax error
Identifier = C
Integer    = 49
Empty token
Empty token
Identifier = ALongIdentifier
Empty token
```

图 7-29　识别标识符和整数的程序的输入 / 输出

状态机从启始状态开始，扫描第一个终结符 H，进入 Ident 状态。接下来的终结符 e、r 和 e 都继续转移到同一个状态。接下来的终结符是空格，从状态 Ident 没有接受终结符空格的转移。因为当前状态机是在标识符的终止状态中，所以得出扫描到一个标识符的结论。因为在当前这个状态无法接受终结符空格，所以把它放回输入，作为下一个 token 的第一个终结符。然后，状态机声明扫描到了一个标识符。

状态机重新回到启始状态，用刚刚刻下的空格转移到 Start。更多的空格也都使得继续转移到 Start，之后字符 i 和 s 会使状态机识别出第二个标识符，如下面的输出所示。类似地，会识别出 A47 也是一个标识符。

对于下一个 token，初始的 4 使得状态机进入 Int 状态，8 使得转移到同一个状态。现在，状态机的输入是 B，从状态 Int 没有接受终结符 B 的转移。因为当前状态机是在整数的终止状态中，所以得出扫描到一个整数的结论。因为在当前这个状态无法接受终结符 B，所以把它放回输入，作为下一个 token 的第一个终结符。然后，状态机声明扫描到了一个整数。而在下一轮，B 会被识别为一个标识符。

状态机继续识别 token 直到到达行尾，此时它识别出的是空 token。无论输入最后有没有末尾的空格，它总是会识别到空 token。

第二个输入的示例展示了状态机如何处理包含句法错误的字符串。在确认了 Here，is 和 A47 之后，在下一次调用中，该 FSM 得到 +，然后进入 Sign 状态。因为下一个字符是空格，Sign 状态没有接受空格的转移，所以 FSM 会返回非法的 token。

像所有的多 token 识别器一样，这个状态机按照如下设计原则运行：

一旦进入终止状态就不会失败。如果没有接受刚刚输入的终结符的转移，就是

识别出了一个token，返回最后一个输入。这个字符会作为下一个token的第一个非终结符。

这个状态机能够正确处理一个空行（或者一个只有空格的行），一次调用就会返回空token。

图7-30是token类结构的统一建模语言（Unified Modeling Language，UML）图示。AToken是一个抽象的token，没有属性，有两个公共的抽象操作tokenType和printToken。操作前面的+号是UML中公共访问的标记。空心三角符号是UML中表示继承的符号。图7-30表明实体类TEmpty、TInvalid、TInteger和TIdentifier继承自AToken。UML按惯例把抽象类名和方法用斜体表示。

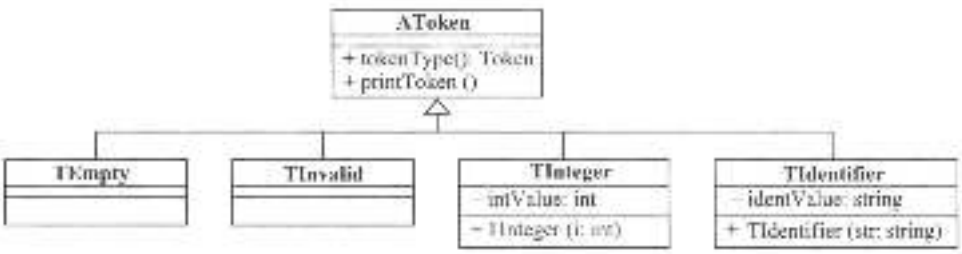


图7-30　AToken类结构的UML图示

每个实体类必须实现它们继承自超类的抽象方法。方法printToken打印如图7-29所示的输出。方法tokenType返回一个枚举类型的值，表明识别出的token的类型。除了继承的方法，类TInteger还有一个私有属性intValue，它存储语法分析器识别的整数值，还有一个公共的构造器。属性前面的-号是UML中私有访问的符号。类似地，类TIdentifier也有一个类型为string的私有属性和自己的构造器。

图7-31展示了图7-30的token类结构对应的C++实现，它是完整程序代码的第一部分，图7-32和图7-33是续。Token是C++的enum类型，它的值对应着4个实体类。

```
#include <iostream>
#include <cctype> // isalpha, isdigit
#include <string> // string, getline
using namespace std;

#include "InBuffer.hpp" // InBuffer
InBuffer inBuffer;

enum Token {eT_IDENTIFIER, eT_INTEGER, eT_EMPTY, eT_INVALID};

class AToken {
public:
   virtual Token tokenType () = 0;
   virtual void printToken () = 0;
};
class TEmpty : public AToken {
public:
   Token tokenType () { return eT_EMPTY; }
   void printToken () { cout << "Empty token" << endl; }
};
```

图7-31　图7-30中AToken类的C++实现

```
class TInvalid : public AToken {
public:
   Token tokenType () { return eT_INVALID; }
   void printToken () { cout << "Syntax error" << endl; }
};

class TInteger : public AToken {
private:
   int intValue;
public:
   TInteger (int i) { intValue = i; }
   Token tokenType () { return eT_INTEGER; }
   void printToken () { cout << "Integer    = " << intValue << endl; }
};

class TIdentifier : public AToken {
private:
   string identValue;
public:
   TIdentifier (string str) { identValue = str; }
   Token tokenType () { return eT_IDENTIFIER; }
   void printToken () { cout << "Identifier = " << identValue << endl;
};
```

图 7-31 （续）

```
enum State {eS_START, eS_IDENT, eS_SIGN, eS_INTEGER, eS_STOP};

void getToken (AToken*& pAT) {
   // Pre: pAT is set to a token object.
   // Post: pAT is set to a token object that corresponds to
   // the next token in the input buffer.
   State state;
   char nextChar;
   int sign;
   int localIntValue;
   string localIdentValue;
   delete pAT;
   pAT = new TEmpty;
   state = eS_START;
   do {
      inBuffer.advanceInput (nextChar);
      switch (state) {
      case eS_START:
         if (isalpha(nextChar)) {
            localIdentValue = nextChar;
            state = eS_IDENT;
         }
         else if (nextChar == '-') {
            sign = -1;
            state = eS_SIGN;
         }
         else if (nextChar == '+') {
            sign = +1;
```

图 7-32 图 7-28 所示 FSM 的 C++ 实现

```
            state = eS_SIGN;
         }
         else if (isdigit(nextChar)) {
            localIntValue = nextChar - '0';
            sign = +1;
            state = eS_INTEGER;
         }
         else if (nextChar == '\n') {
            state = eS_STOP;
         }
         else if (nextChar != ' ') {
            delete pAT;
            pAT = new TInvalid;
         }
         break;
      case eS_IDENT:
         if (isalpha (nextChar) || isdigit (nextChar)) {
            localIdentValue.push_back (nextChar);
         }
         else {
            inBuffer.backUpInput ();
            delete pAT;
            pAT = new TIdentifier (localIdentValue);
            state = eS_STOP;
         }
         break;
      case eS_SIGN:
         if (isdigit (nextChar)) {
            localIntValue = nextChar - '0';
            state = eS_INTEGER;
         }
         else {
            delete pAT;
            pAT = new TInvalid;
         }
         break;
      case eS_INTEGER:
         if (isdigit (nextChar)) {
            localIntValue = 10 * localIntValue + nextChar - '0';
         }
         else {
            inBuffer.backUpInput ();
            delete pAT;
            pAT = new TInteger (sign * localIntValue);
            state = eS_STOP;
         }
         break;
      }
   }
   while (state != eS_STOP) && (pAT->tokenType () != eT_INVALID));
}
```

图 7-32 （续）

图 7-32 是图 7-28 所示 FSM 的直接编码实现。方法 getToken 的输入是一个传引用调用的指针 pAT。助记符 pAT 表示“指向抽象 token 的指针”。getToken 的前提假设是 pAT

已经被设置为指向一个已经分配好空间的token，其初始值没有关系。当这个方法需要改变pAT的值时，会删除旧值，用new操作符分配一个新值。这种使每个new都匹配对应于一个delete的编程风格能够帮助预防内存泄露。

图7-33给出了主函数。它有一个抽象token pAToken，第一行对它进行初始化。在程序结束前执行对应的delete操作，同时还保持使用每个new都匹配对应有一个delete的编程风格。外层的while循环对输入的每一行执行一次。内层的do循环对一行中的每个token执行一次。输出依靠多态来显示识别出的不同类型的token。也就是说，主函数没有显式地测试token的类型来选择如何输出它的值，而只是简单地让抽象的token调用printToken方法。

```
int main () {
   AToken* pAToken = new TEmpty;
   inBuffer.getline ();
   while (!cin.eof ()) {
      do {
         getToken (pAToken);
         pAToken->printToken ();
      }
      while (!(pAToken->tokenType () != eT_EMPTY)
          && (pAToken->tokenType () != eT_INVALID));
      inBuffer.getline ();
   }
   delete pAToken;
   return 0;
}
```

图7-33 图7-32所示token识别器的主函数

7.4 代码生成

翻译是把某种输入字符表的字符串变换为某种输出字符串的字符串。这种翻译的典型阶段是词法分析、语法分析和代码生成。本节包括一个程序，它把一种语言翻译成另一种，用以说明一个简单自动翻译器的所有3个阶段。

7.4.1 语言翻译器

图7-34展示了翻译器的输入/输出。输入是源代码，输出是目标代码。源语言和目标语言都是面向行的，汇编语言也是如此。源语言的句法包括C++函数调用。目标语言赋值语句的句法是使用赋值运算符:=。输入语言的一个语句示例是

```
set (Time, 15)
```

对应的目标语句是

```
Time := 15
```

字set是源语言的保留字，其他保留字包括add、sub、mul、div、neg和end。

Time是用户定义的标识符。标识符的定义规则与C++语言相同。例如前面例子中的15，语法也与C++一样。

set过程有两个参数，通过逗号隔开，用圆括号括起来。第一个参数必须是标识符，而第二个参数可以是标识符，也可以是整数常量。

翻译的另一个例子是

输入

```
set (Time, 15)
set (Accel, 3)
set (TSquared, Time)
mul (TSquared, Time)
set (Position, TSquared)
mul (Position, Accel)
div (Position, 2)
end
```

输出

```
Time := 15
Accel := 3
TSquared := Time
TSquared := TSquared * Time
Position := TSquared
Position := Position * Accel
Position := Position / 2
```

a）第一次运行

输入

```
set (Alpha, 123)
set (Alpha)
set (Alpha, 123)
set, (Alpha)
mul (Alpha, Beta
set (123, Alpha)
neg (Alpha, Beta)
set (Alpha, 123) x
```

输出

```
Error: Second argument not an identifier or integer.
Error: Comma expected after first argument.
Error: Line must begin with function identifier.
Error: Left parenthesis expected after function.
Error: Right parenthesis expected after argument.
Error: First argument not an identifier.
Error: The argument of "neg" is malformed.
Error: Illegal trailing character.
Error: Missing "end" sentinel.
```

b）第二次运行

图 7-34　从一种语言到另一种语言翻译程序的输入/输出

```
mul (TSquared, Time)
```

它的目标代码写作

```
TSquared := TSquared * Time
```

360 : 369

与 set 过程一样，mul 过程调用的第一个参数必须是标识符。要翻译 mul 语句，翻译器必须重复它的第一个参数，它出现在赋值操作符的两边。

其他过程调用类似，除了 neg。neg 只有一个参数。要翻译它，就要在赋值运算符的右边先写一个短画线符号。例如，源程序语句

```
neg (Alpha)
```

被翻译成

```
Alpha := -Alpha
```

保留字 end 是翻译器的分隔符，不会生成任何代码，类似于 Pep/8 汇编语言中的 .END。源程序行中可以出现任意数量的空格，但是标识符和整数中不能有空格。

如果输入流中出现语法错误，翻译器不一定崩溃。在图 7-34 中，还有一次运行中，源文件里有各种错误。程序产生适当的错误信息帮助用户找到源程序中的问题（bug）。

这个程序就于句法的两阶段分析，如图 7-24 所示。不过没有像图中表明的那样用语法来描述分析问题，这个源语言的结构很简单，可以直接基于 FSM 构造分析器。

图 7-35 是产生图 7-34 输出程序的开始部分代码。图 7-37、图 7-39、图 7-41、图 7-42、图 7-44 和图 7-45 都是该程序的代码。图 7-35 没有给出程序开头的 #include 语句。操作符表和助记符表使用 C++ 标准模板库（Standard Template Library，STL）中的 map 数据结构。要使用 map 数据结构，必须包含 <map> 库。操作符表用枚举类型的助记符值作为关键字查找字符串符号，并放到生成的代码中。助记符表用源码保留字的小写字符串表示查找对应的助记符。

```
// ======== Global tables
enum Mnemon {
   eM_ADD, eM_SUB, eM_MUL, eM_DIV,
   eM_NEG, eM_SET, eM_END, eM_EMPTY
};

map <Mnemon, string> operatorTable;
map <string, Mnemon> mnemonTable;

void initGlobalTables () {
   operatorTable [eM_ADD] = "+";
   operatorTable [eM_SUB] = "-";
   operatorTable [eM_MUL] = "*";
   operatorTable [eM_DIV] = "/";
   mnemonTable ["add"] = eM_ADD;
   mnemonTable ["sub"] = eM_SUB;
   mnemonTable ["mul"] = eM_MUL;
   mnemonTable ["div"] = eM_DIV;
   mnemonTable ["neg"] = eM_NEG;
   mnemonTable ["set"] = eM_SET;
   mnemonTable ["end"] = eM_END;
}

void lookUpMnemon (string id, Mnemon& mn, bool& fnd) {
   string lowerId = "";
   for (int i = 0; i < id.size (); i++) {
      lowerId.push_back (tolower (id[i]));
   }
   if (mnemonTable.count (lowerId) == 1) {
      mn = mnemonTable[lowerId];
      fnd = true;
   }
   else {
      fnd = false;
   }
}
```

例 7-35 翻译程序的全局表和查找函数

函数 lookUpMnemon 的输入参数是字符串 id，它检查该字符串是否在助记符表中。如果在，把参数 mn 设置为相应的枚举类型的助记符类型，并把 fnd 设置为 true；否则，就把 fnd 设置为 false。该程序允许源代码不区分大小写。

图 7-36 是抽象参数的 UML 图示，而图 7-37 是它 C++ 实现。因为源代码中的参数可以是标识符，也可以是整数，所以程序中有一个通用的参数 AArg，它在运行时可以是 IdentArg 也可以是 IntArg 类。AArg 定义了一个抽象的方法 putArg，当输出参数的值时，就用它来生成代码。

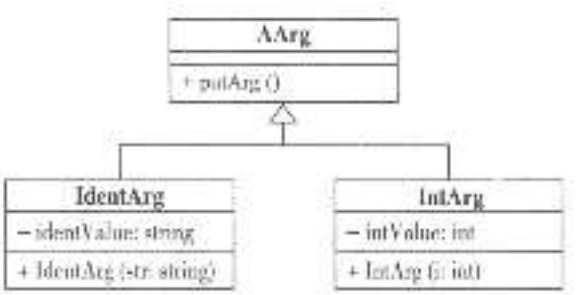


图 7-36 类结构 AArg 的 UML 图示

```
// ======== The argument classes
class AArg {
public:
   virtual void putArg () = 0;
};

class IdentArg : public AArg {
private:
   string identValue;
public:
   IdentArg (string str) { identValue = str; }
   void putArg () { cout << identValue; }
};

class IntArg : public AArg {
private:
   int intValue;
public:
   IntArg (int i) { intValue = i; }
   void putArg () { cout << intValue; }
};
```

图 7-37　对 7-36 中类 AArg 的 C++ 实现

图 7-38 是抽象 token 的 UML 图示，而图 7-39 是它 C— 实现的部分代码。这个 token 的结构类似于前一节中图 7-30 中的 token 结构。方法 tokenType 的作用与前面一样。方法 getArg 返回指向一个抽象参数的指针，它只适用于 TIdentifier 和 TInteger，因为 AToken 的这两个子类是唯一包含代码生成器生成目标代码所需属性的类。对于 FIdentifer，getArg 返回一个根据它的 identValue 新分配的 IdentArg；而对于 FInteger，getArg 返回一个根据它的 intValue 新分配的 IntArg。

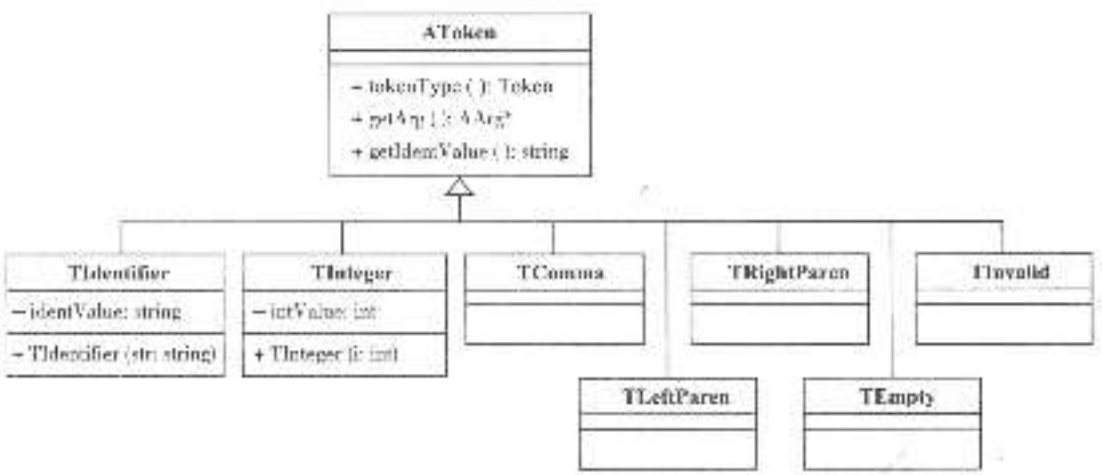


图 7-38　类结构 AToken 的 UML 图示

方法 getIdentValue 只适用于 TIdentifer。当词法分析器遇到保留字和参数时，它返回一个标识符。当它遇到保留字时，语法分析器需要在地记符表中查找这个字，用 getIdentValue 来获取 token 的标识符值。

```
// ========== the token classes
enum Token {
   eT_IDENTIFIER, eT_INTEGER, eT_COMMA, eT_LEFT_PAREN,
   eT_RIGHT_PAREN, eT_EMPTY, eT_INVALID
};

class AToken {
public:
   virtual Token tokenType () = 0;
   virtual AArg* getArg () { return 0; }
   virtual string getIdentValue () { return ""; }
};

class TIdentifier : public AToken {
private:
   string identValue;
public:
   TIdentifier (string str) { identValue = str; }
   AArg* getArg () {
      AArg* pArg = new IdentArg (identValue);
      return pArg;
   }
   Token tokenType () { return eT_IDENTIFIER; }
   string getIdentValue () { return identValue; }
};

class TInteger : public AToken {
private:
   int intValue;
public:
   TInteger (int i) { intValue = i; }
   AArg* getArg () {
      IntArg* pArg = new IntArg (intValue);
      return pArg;
   }
   Token tokenType () { return eT_INTEGER; }
};

class TComma : public AToken {
public:
   Token tokenType () { return eT_COMMA; }
};

...
```

图 7-39 图 7-38 中类 AToken 的 C++ 实现

图 7-40 是抽象代码类 ACode 的 UML 图示，图 7-41 是它 C++ 实现的部分代码。类 ACode 的对象表示目标代码的一行。执行方法 generateCode 把该行代码输出到 cout。所以，一个代码对象必须包含输出该行代码所需的所有数据。例如，图 7-40 展示了类 TwoArgs 的一个对象，它有两个属性，pFirstArg 和 pSecondArg，两者分别都是指向一个抽象参数的指针。除此之外，该对象还有一个属性 mnemonic，继承自它的超类 Valid。考虑图 7-34a 的最后一行输入

```
div (Position, 2)
```

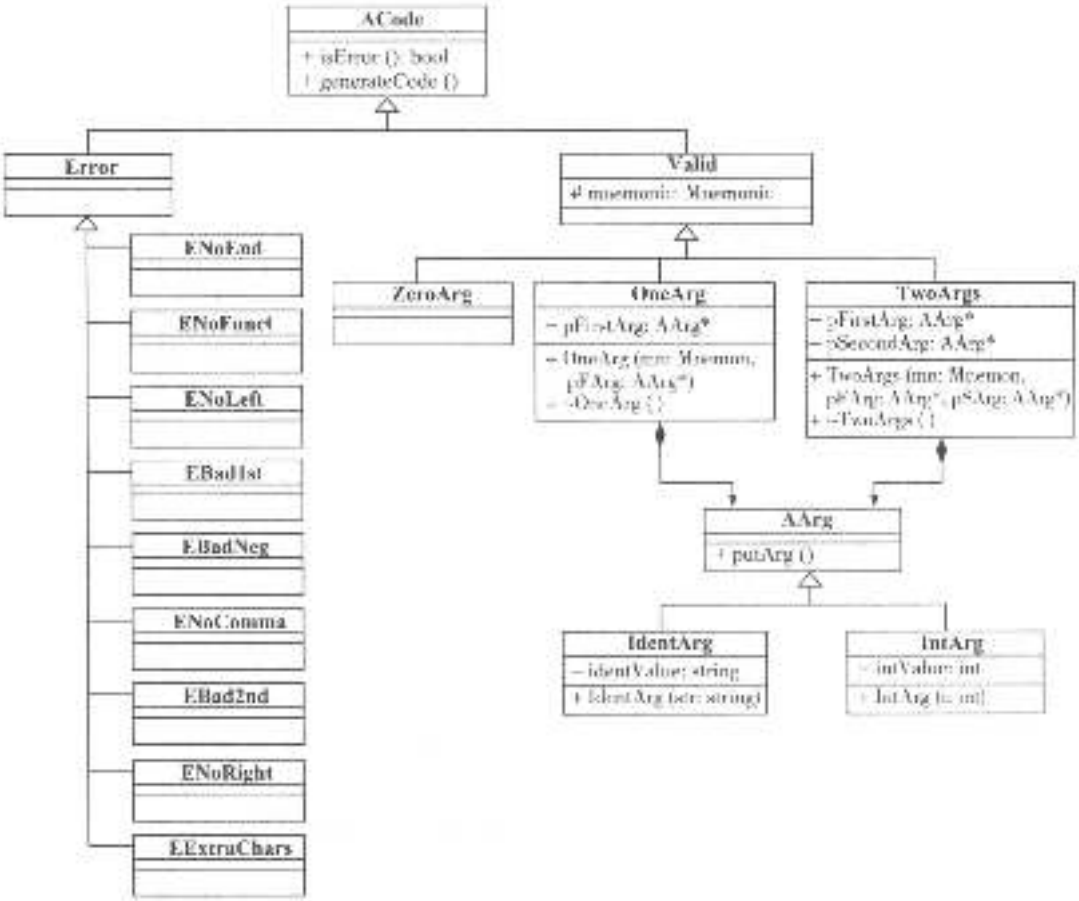


图 7-40　类结构 ACode 的 UML 图示

```
// -------- The base classes
class ACode {
public:
   virtual bool isError () = 0;
   virtual void generateCode () = 0;
};

// -------- The error code classes
class Error : public ACode {
public:
   bool isError () { return true; }
};

class ENoEnd : public Error {
public:
   void generateCode () {
      cout << "Error: Missing \"end\" sentinel.";
   }
};

...
```

图 7-41　图 7-40 中类 ACode 的 C++ 实现

```
// ---------- The valid code classes
class Valid : public ACode {
protected:
   Mnemon mnemonic;
public:
   bool isError () { return false; }
};

class ZeroArg : public Valid {
public:
   ZeroArg (Mnemon mn) { mnemonic = mn; }
   void generateCode () { } // cout nothing
};

class OneArg : public Valid {
private:
   AArg *pFirstArg;
public:
   OneArg (Mnemon mn, AArg* pFArg) {
      mnemonic = mn;
      pFirstArg = pFArg;
   }
   ~OneArg () { delete pFirstArg; }
   void generateCode () {
      pFirstArg->putArg ();
      cout << " := -";
      pFirstArg->putArg ();
   }
};

...
```

图 7-41 （续）

这个代码对象的 mnemonic 等于 eM_DIV，pFirstArg 指向一个 identValue 等于 "Position" 的 IdentArg，pSecondArg 指向一个 intValue 等于 2 的 IntArg。类 OneArg 用于 neg 语句，它只有一个参数。

UML 中表示受保护访问的符号是 #，与图 7-40 中 Valid 类的属性中使用的一样。表示类合成的 UML 符号是实心菱形，连接 OneArg 类的方框和 TwoArgs 类的方框。类合成的含义是 "有一个"，区别于继承的含义 "是一个"。OneArg 对象是一个 Valid 对象，而 OneArg 对象有一个 AArg 对象。

图 7-42 是词法分析器的一部分代码。除了能够识别图 7-38 所示的 7 种 token 之外，函数 getToken 与图 7-32 中的 getToken 函数类似。和前面一样，形参 pAT 是一个指向传引用调用的抽象 token 的指针。

图 7-43 展示了一个描述源语言的确定性的 FSM，图 7-44 是其实现的部分代码。该状态机的转移接受来自词法分析器的 token，即图 7-39 中以 eT 开头的字。终止状态 ePS_FINISH 只能通过输入 token eT_EMPTY 才能到达。如果输入行是空白行或该行只含有空格，那么就会从 ePS_START 转移到 ePS_FINISH。终结符字符串 end 是唯一一个使得状态从 ePS_START 转移到 ePS_END 的标识符。对应于其他保留字（set、and、sub、mul、div 和 neg）的标识符使得状态从 ePS_START 转移到 ePS_FUNCTION。如果在 ePS_START 状态检测到除此

```
// ======== The lexical analyzer
enum LexState {
   eLS_START, eLS_IDENT, eLS_SIGN, eLS_INTEGER, eLS_STOP
};

void getToken (AToken*& pAT) {
   // Pre: pAT is allocated to an irrelevant value.
   char nextChar;
   string localIdentValue;
   int localIntValue;
   int sign;
   delete pAT;
   pAT = new TEmpty;
   LexState state = eLS_START;
   do {
      inBuffer.advanceInput (nextChar);
      switch (state) {
      case eLS_START:
         if (isalpha (nextChar)) {
            localIdentValue = nextChar;
            state = eLS_IDENT;
         }

...

      case eLS_INTEGER:
         if (isdigit (nextChar)) {
            localIntValue = 10 * localIntValue + nextChar - '0';
         }
         else {
            inBuffer.backUpInput();
            delete pAT;
            pAT = new TInteger (sign * localIntValue);
            state = eLS_STOP;
         }
         break;
      }
   }
   while ((state != eLS_STOP) && (pAT->tokenType() != eT_INVALID));
}
```

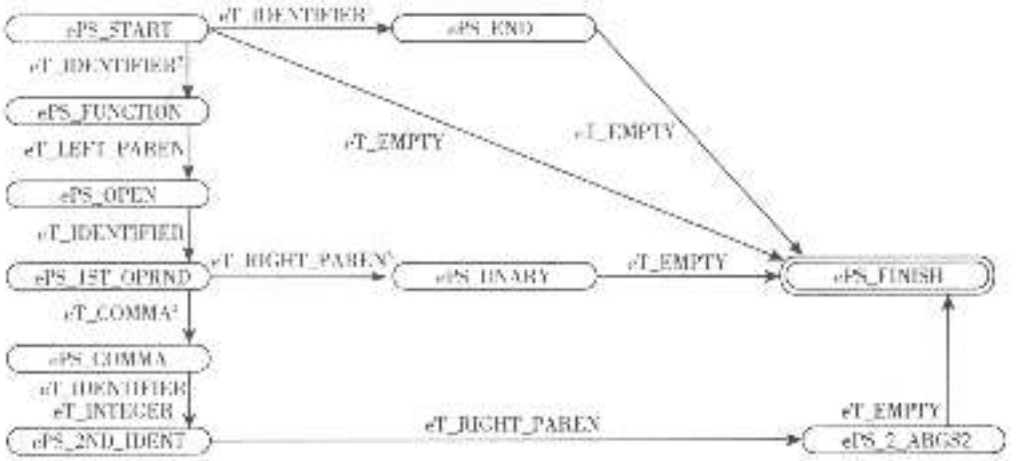
图 7-42　词法分析器

之外所有其他的标识符都是非法的。

图 7-45 是产生图 7-34 所示输出的主程序的完整代码。布尔变量 terminateWithEnd 控制循环的结束，被初始化为 false 并传引用传递给 processSourceLine。只有 end 语句会把 terminateWithEnd 设置为 true，因为 end 对翻译器来说是终止分隔符。如果用户省略 end 语句，主函数检测到已经到达文件的结尾而没有收到 end 语句，那么就会发出一条适当的错误信息。

执行自动翻译的三个阶段的函数分别是：

- 词法分析器：getToken
- 语法分析器：processSourceLine
- 代码生成器：generateCode



Note 1: Only the identifier end.
Note 2: Only the identifiers set, add, sub, mul, div, and neg.
Note 3: Only for mnemonic eM_NEG.
Note 4: Only for mnemonics eM_SET, eM_ADD, eM_SUB, eM_MUL, and eM_DIV.

图 7-43　图 7-44 的语法分析器 processSourceLine 的 FSM

```
// ======== The parser
enum ParseState {
   ePS_START, ePS_END, ePS_FUNCTION, ePS_OPEN, ePS_1ST_OPRND,
   ePS_UNARY, ePS_COMMA, ePS_2ND_OPRND, ePS_NON_UNARY, ePS_FINISH
};

void processSourceLine (ACode*& pAC, bool& term) {
   // Pre: term is false.
   // A source line is in inBuffer ready for processing.
   // pAC is allocated to an irrelevant value.
   AArg* pLocalFirstArg;
   AArg* pLocalSecondArg;
   Mnemon mnemon;
   delete pAC;
   pAC = new ZeroArg (eM_EMPTY);
   AToken* pAToken = new TEmpty;
   ParseState state = ePS_START;
   do {
      getToken (pAToken);
      switch (state) {
      case ePS_START:
         if (pAToken->tokenType () == eT_IDENTIFIER) {
         string tempStr = pAToken->getIdentValue ();
         bool found;
         lookUpMnemon (tempStr, mnemon, found);
         if (found) {
            if (mnemon == eM_END) {
               delete pAC;
               pAC = new ZeroArg (eM_END);
               term = true;
               state = ePS_END;
            }
```

图 7-44　实现图 7-43 的 FSM 的语法分析器

```
            else {
               state = ePS_FUNCTION;
            }
         }
         else {
            delete pAC;
            pAC = new ENoFunct;
         }
      }
      else if (pAToken->tokenType () == eT_EMPTY) {
         // pAC initialized to ZeroArg (eM_EMPTY)
         state = ePS_FINISH;
      }
      else {
         delete pAC;
         pAC = new ENoFunct;
      }
      break;

...

   case ePS_COMMA:
      if (pAToken->tokenType () == eT_IDENTIFIER)
      || (pAToken->tokenType () == eT_INTEGER)) {
         pLocalSecondArg = pAToken->getArg ();
         delete pAC;
         pAC = new TwoArgs (mnemon, pLocalFirstArg, pLocalSecondArg);
         state = ePS_2ND_OPRND;
      }
      else {
         delete pAC;
         pAC = new EBad2nd;
      }
      break;
   case ePS_2ND_OPRND:
      if (pAToken->tokenType () == eT_RIGHT_PAREN) {
         state = ePS_NON_UNARY;
      }
      else {
         delete pAC;
         pAC = new ENoRight;
      }
      break;
   case ePS_NON_UNARY:
      if (pAToken->tokenType () == eT_EMPTY) {
         state = ePS_FINISH;
      }
      else {
         delete pAC;
         pAC = new EExtraChars;
      }
      break;
   }
}
```

图 7-44 （续）

```
    while (!state != eFS_FINISH) && (!pAC->!eError ));
    delete pAToken;
}
```

图 7-44 （续）

```
// ======== The main program
int main () {
   bool terminateWithEnd = false;
   ACode* pACode = new ZeroArg (eM_EMPTY);
   initGlobalTables ();
   inBuffer.getLine ();
   while (! (cin.eof() || terminateWithEnd)) {
      processSourceLine (pACode, terminateWithEnd);
      pACode->generateCode ();
      cout << end;
      inBuffer.getLine ();
   }
   if (!terminateWithEnd) {
      delete pACode;
      pACode = new FNoEnd;
      pACode->generateCode ();
      cout << end;
   }
   delete pACode;
   return 0;
}
```

图 7-45 产生图 7-34 所示输出的翻译器的主程序

语法分析器调用词法分析器，主函数对每一行源代码调用语法分析器。主函数会调用代码生成器，它的代码生成方法分布在不同的对象中。

7.4.2 语法分析器特性

可以不用图 7-43 中的 FSM 来定义元语言的句法，而用语法来定义。这里，这个源语言的形式语法结构很简单。例如，set 语句的产生式规则为

<set-statement> → set (<identifier> , <argument>)

再用一条产生式规则把这里的 <argument>（<实参>）定义为 <identifier>（<标识符>）或 <integer>（<整数>）。与 C++ 中不同的是，这个语法不能包含递归定义。

因为源句法很简单，所以对该语言的语法分析可以基于确定性的 FSM。但是，大多数编程语言的语法分析器没有这么简单。虽然词法分析器通常可以基于有限状态机，但是语法分析器是比较少见能够基于 FSM 的，实际中大多数语言太复杂，没有办法使用这项技术。

因为一个真实语法的产生式规则总是会包含许多递归定义，所以分析程序也包含递归过程以反映语法的递归特性。这样的算法称作递归下降分析器（recursive descent parser）。

370 ~ 381

无论源语言的复杂度或者翻译器的分析技术如何，翻译器中语法分析器与词法分析器

的关系都是一样的：语法分析器的抽象层次高于词法分析器。词法分析器扫描字符，识别token；语法分析器扫描token，识别源语言程序。

总结

计算机科学的基本问题是“什么能够被自动化？”人工语言的自动化翻译是计算机科学的核心。每种人工语言都有一个符号表。一个集合的闭包，T^*，是连接T中元素能够形成的所有可能字符串的集合。语言是它的字符表闭包的一个子集。语法描述语言的句法，有4个部分：一个非终结符表；一个终结符表，一组产生式规则和一个起始符号。推导是语法说明语言的过程。为了推导出语言的一个句子，要从起始符号开始，用产生式规则做替换，直到得到一个终结符字符串。语法分析问题是确定推导序列，使之与一个给定的终结符字符串匹配。标准C++语法有上百条产生式规则。上下文无关语法是限制产生式规则左边只能包含一个非终结符的语法。虽然C++语法是上下文无关的，但是它的某些方面是上下文相关的。

有限状态机（FSM）也能描述语言的句法，由一组状态和状态之间的转移组成。每个转移都标明有一个输入终结符号，有一个状态是起始状态，至少有一个，也可能有多个终止状态。非确定性的FSM中从一个给定状态对于一个输入终结符，可能有多个转移。如果从起始状态出发，有一个转移序列接受句子中的符号，使得最后以终止状态结束，那么这个句子就是合法的。

FSM的两个软件实现技术是：查找表技术和直接编码技术。两种技术都包含由状态变量控制的循环，这个状态变量会被初始化为起始状态。循环的每次执行都对应于FSM中的一次转移。在查找表技术中，转移是由一个二维转移表确定的；而在直接编码技术中，转移是由循环体内的选择语句指定的。

自动翻译器的3个翻译阶段是词法分析器、语法分析器和代码生成器。对于大多数高级语言来说，词法分析器基于FSM，语法分析器基于上下文无关文法，代码产生器严重依赖于目标语言。

练习

7.1节

*1. 计算机科学的基本问题是什么？

2. 整数加法运算的单位元是什么？布尔数的OR运算的单位元是什么？

3. 用图7-1的语法推导出下面的字符串，画出相应的词法树。

*(a) abc123　　(b) a1b2c3　　(c) a321bc

4. 用图7-2的语法推导出下面的字符串，画出相应的词法树。

*(a) -9　　(b) +000　　(c) 0

5. 用图7-3的语法推导出下面的字符串。

*(a) abc　　(b) aabbcc

6. 对于下面每个字符串，说明是否能从图7-5的语法规则推导出来。如果可以，画出对应的语法树：

*(a) a + (a)　　(b) a * (- a)　　(c) a / (a + a)

(d) a * (a + a) * a　　(e) a + (- a)　　(f) (((a)))

7. 对于图7-8的语法，画出<statement>（<语句>）对于下列字符串的语法树，假设S1、S2、S3、S4、C1和C2都是合法的<expression>（<表达式>）：

```
*(a) {  if (C1)
          S1;
        S2;
    }
```

```
(b) {  if (C1)
         if (C2)
           S1;
         else
           S2;
       S3;
    }
```

```
(c) {  if (C1)
         if (C2)
           S1;
         else
           S2;
       else
         S3;
       S4;
    }
```

```
(d) {  S1;
       while (C1)
       {  if (C2)
            S2;
          S3;
       }
    }
```

8. 对于图 7-8 的语法，画出 <statement>（< 语句 >）对于下列字符串的语法树，假设 alpha，beta 和 gamma 都是合法的 <identifier>（< 标识符 >），1 和 24 都是合法的 <constant>（< 常数 >）：

```
*(a)alpha = 1 ;
 (b)alpha = alpha + 1 ;
 (c)alpha = (beta * 1) ;
 (d)alpha = (beta + 1) * (gamma + 24)) ;
 (e)alpha (beta) ;
 (f)alpha (beta, 24) ;
```

383

9. 对于图 7-8 的语法，画出 <translation-unit>（< 翻译单元 >）对于下列字符串的语法树，假设 alpha，beta，gamma 和 main 都是合法的 <identifier>（< 标识符 >），C1，S1 和 S2 都是合法的 <expression>（< 表达式 >）：

```
int main()
{  int gamma;
   alpha (gamma);
   if (C1)
     S1;
   else
     S2;
}
```

10. 本练习提出的问题是“两个不同的语法能产生同一种语言吗？”图 7-46 和图 7-47 中的语法是不同的，它们有不同的非终结符集合和不同的产生式规则。用这两种语法做实验，推导一些终结符字符串，通过你的实验，描述这两种语法产生的语言。可以根据图 7-46 的语法推导出用图 7-47 的语法推导不出来的合法终结符字符串吗？反过来呢？证明你的推测。

N = {A, B}
T = {0,1}
P = the productions
1. A → 0 B
2. B → 1 0 B
3. B → e
S = A

图 7-46　练习 10 的语法

N = {C}
T = {0, 1}
P = the productions
1. C → C 1 0
2. C → 0
S = C

图 7-47　练习 10 的另一个语法

7.2 节

11. 对于图 7-48 给出的每个状态机，(a) 说明这个 FSM 是确定性的还是非确定性的，(b) 识别出所有不可达的状态。

12. 为图 7-49 中的每个有限状态机消除空转移，得到等价的状态机。

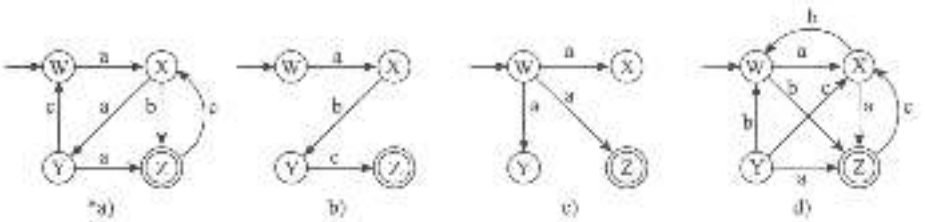


图 7-48　练习 11 的 FSM

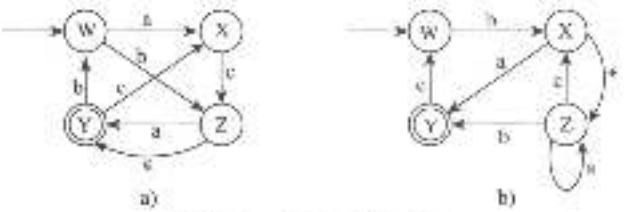


图 7-49　练习 12 的 FSM

384

13. 画一个有限 FSM，能够识别以下标准指定的 1 和 0 的字符串。每个 FSM 拒绝所有非 0 或 1 的字符。*（a）3 个字符的字符串，101。（b）所有长度任意以 101 结尾的字符串。例如，该 FSM 应该能够接受 1101，但是拒绝 1011。（c）所有长度任意以 101 开头的字符串。例如，该 FSM 应该接受 1010，但是拒绝 0101。（d）所有长度任意且至少包含一个 101 的字符串。例如，该 FSM 应该能够接受（a）、（b）和（c）中提到的所有字符串，以及像 11100001011111100111 这样的字符串。

7.3 节

14. 设计一个描述图 7-44 中翻译器的源语言的语法。
15. 画出图 7-32 中过程 getToken 中 FSM 的状态转移图。

问题

7.3 节

16. 按照本书所建议的那样，改进图 7-25 中的程序，在 Alphabet 中定义第三个枚举值 other，它表示既不是字母又不是数字的符号。
17. 用图 7-25 中程序的查找表技术实现练习 13 中的每个 FSM。转移表中要把字符区分为 eONE，eZERO 或 eOTHER。从键盘获取输入。
18. 用图 7-26 中程序的直接编码技术实现练习 13 中的每个 FSM。从键盘获取输入。写一个名为 parsePat 的过程来分析对应于 parseNum 的模式。该过程中不要使用形参 n。
19. 十六进制数字是 '0' … '9'、'a' … 'f' 或 'A' … 'F'。一个十六进制常数由 1 … 4 个的十六进制数字组成。例如 3、a、0d 和 FF4e。用直接编码技术实现一个有限状态机，就像图 7-26 中的一样，分析十六进制常数，并把它转换为非负整数。输入/输出应该与图 7-26 类似。非法输入会产生错误消息，合法十六进制输入字符串会产生非负整数值。下面 <cctype> 中的函数可能会有用。ch 的类型为 char。

    isxdigit(ch)，如果 ch 是一个十六进制数字，返回 true。
    isdigit(ch)，如果 ch 是一个十进制数字，返回 true。
    isupper(ch)，如果 ch 是一个大写字母，返回 true。
    islower(ch)，如果 ch 是一个小写字母，返回 true。
    toupper(ch)，如果 ch 是一个小写字母，就返回 ch 的大写字母，否则返回 ch。

385　tolower(ch)，如果 ch 是一个大写字母，就返回 ch 的小写字母，否则返回 ch。

7.4 节

20. 为 Pep/8 汇编语言写一个有限制的汇编器。源程序包含的汇编语言程序不含符号，具有助记符 BR、DECI、LDA、ADDA、STA、DECO 和 STOP；点命令有 .BLOCK 和 .END；以及十六进制和十进制常数。例如，如果源程序如下所示：

```
BR    0x0007,  d
.BLOCK 4
deci    0x2  ,d
LDA     +2,d
AdDa  -5,  i
STA     0x0004,d
    DECO    0x04,d
STOP
.END
```

那么汇编器应该输出如下的目标程序

```
04 00 07 00 00 00 00 31 00 02 21 00 02 70 ff fb
01 00 04 39 00 04 00 zz
```

装载器不接受目标程序行末有空格。

假设源程序行中没有注释。所有的寻址方式都是显式的，即使是对于 BR 指令。验证一条指令的寻址方式是否合法，如果不合法就给出错误提示。助记符和点命令可以大小写字符混用。除了助记符或点命令之后一定要跟至少一个空格以外，token 之间可以有零个或多个空格。如果源文件包含一个或多个错误，输出相应的消息，建议输出给目标文件。按照列出的顺序实现下面的主程序节点。

(a) 写出过程 getToken，它实现图 7-50 的 FSM。声明下面这样一个枚举类型：

```
enum Token {
   eT_IDENTIFIER, eT_DOT_COMMAND, eT_DEC_CONST,
   eT_HEX_CONST, eT_ADDR_MODE, eT_EMPTY, eT_INVALID
};
```

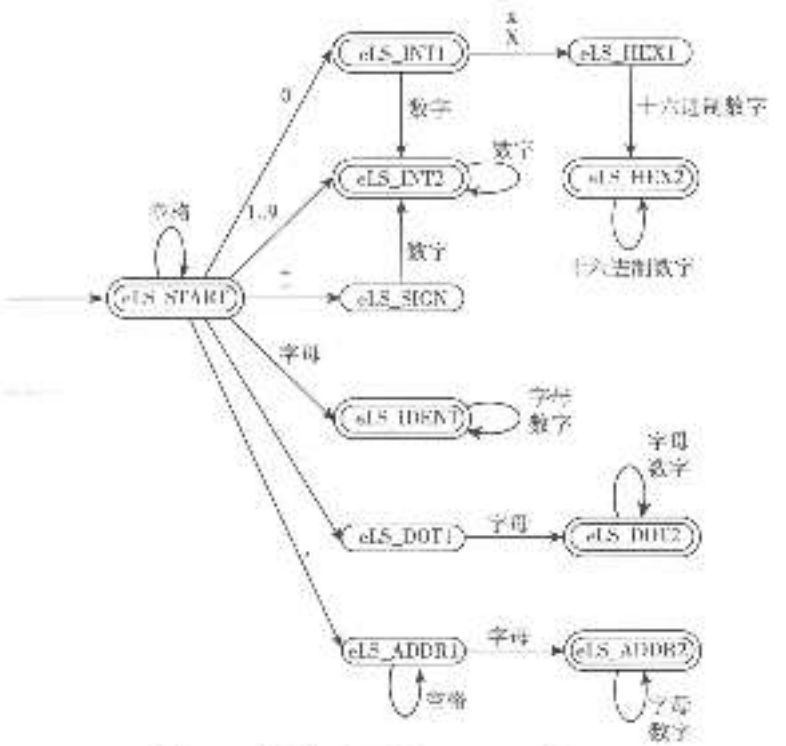


图 7-50 问题 20 (a) 中 getToken 的 FSM

设计一个类似于图7-31中的抽象token，画出它的类似于图7-30中的UML图示。

用类似于图7-33中的main()来测试你的程序，输出token列表和它们的值。点命令的属性和寻址方式应该是字符串，十进制常数和十六进制常数的属性应该是整数。你的测试字符串中应该包含合法token序列的字符串，也应该包含会产生句法错误的字符串。输入用图7-27中的InBuffer.hpp。

(b) 画出对应于图7-43语法分析器的FSM的状态转移图。假设每个转移接受(a)部分中列出的那些token中的一个。

(c) 基于(b)部分中的状态转移图，完成过程processSourceLine。声明下面这样一个枚举类型：

```
enum Mnemon {
   eM_BR, eM_DECI, eM_LDA, eM_ADDA,
   eM_STA, eM_DECO, eM_STOP
};
enum AddrMode {
   eA_IMMED, eA_DIRECT, eA_INDIRECT, eA_STACK_REL,
   eA_STACK_REL_DEF, eA_INDEXED, eA_STACK_INDEXED,
   eA_STACK_INDEXED_DEF
};
```

386
;
387

设计抽象参数AArg，它有两个子类，分别描述十六进制常数和十进制常数，每个都有一个整型属性。往你的token类中添加一个方法getArg。设计你的抽象代码类ACode，像图7-40那样。Valid类不能含有助记符属性，应该有5个子类：Empty是为了空行；Unary是为了一元STOP指令；NonUnary是为了BR、DECI、LDA、ADDA、STA和DECO指令；DotBlock是为了.BLOCK伪操作；而DotEND是为了.END伪操作。Unary和NonUnary类应该有一个枚举类型的助记符。NonUnary和DotBlock类应该有一个抽象参数。NonUnary类应该有一个枚举类型的AddrMode属性。用C++ map实现的寻址方式表把token中的寻址方式的字符串表示转换到枚举类型的值。

用类似于图7-45中的主程序来测试你的程序。现在，你的generateCode方法已经可以输出具有参数和寻址方式的每条指令和点命令分析的描述。你的测试字符串中应该包含合法指令序列的字符串，也应该包含会产生错误的字符串。

(d) 向你的汇编器中增加十六进制代码生成部分。除了变量pACode外，声明一个如下代码结构的数组：

```
ACode* codeTable[256];
int codeIndex;
```

在主程序开头，把codeIndex初始化为0。每分析一个源代码行，就把来自该行的信息存储到CodeTable[codeIndex]，再把codeIndex加1。如果所有的行都分析完了，且没有错误，就在这个代码数组上循环生成目标代码。

(e) 扩展这个汇编器，包括Pep/8指令集中所有39条指令。

(f) 扩展这个汇编器，允许分支指令不指定寻址方式，此时使用默认的方式立即数寻址。

(g) 扩展这个汇编器，生成这样的列表，把行号代码号在生成它的源代码行旁边。使用Pep/8汇编器标准的空格规则和大小写规则输出源代码行。

(h) 扩展这个汇编器，允许用单引号扩起字符常量。

(i) 扩展这个汇编器，允许点命令.WORD和.BYTE。

(j) 扩展这个汇编器，允许点命令.ASCII，字符串用双引号括起来。

(k) 扩展这个汇编器，允许源代码行包含以分号开始的注释。一行可以只包含一条注释，也可以一

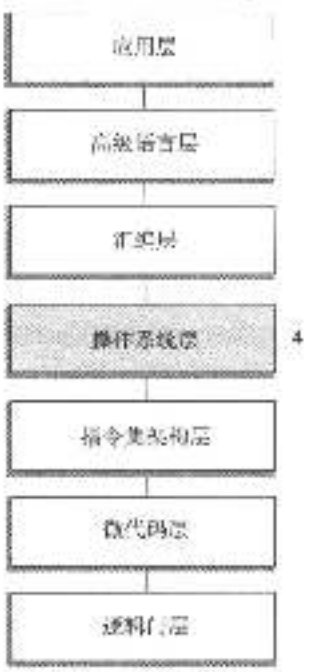
388

条合法指令后面跟一条注释。

第五部分

Computer Systems，Fourth Edition

操作系统层（第4层）



进程管理

操作系统定义了一个比 ISA3 层机器更抽象的机器，更容易对它编程。其目的是向高级语言提供一个更加方便的环境，并更有效地分配系统资源。操作系统层位于汇编层和机器层之间。与一般的抽象一样，操作系统向更高层次的用户隐藏 ISA3 层机器的细节。

典型计算机系统的资源包括 CPU 时间、主存和磁盘存储器。本章讲述操作系统怎样分配 CPU 时间，接下来的第 9 章讲述它怎样分配主存和磁盘存储器。

操作系统一般分为 3 类：

- 单用户
- 多用户
- 实时

许多个人计算机是单用户操作系统，这些计算机对于个人来说不贵买得起用得起，不用和其他人共享。大型计算机几乎都是组织机构所有而不是个人所有，它们使用多用户操作系统。这样的计算机通常只有一个 CPU，但是它却足够快，能同时执行许多用户的作业。计算机中使用的实时系统专门用于控制设备，它们的输入来自传感器，输出是给设备的控制信号。例如，控制汽车发动机的计算机使用的就是实时系统。

389 / 391

Pep/8 操作系统是一个单用户操作系统，它展示了分配 CPU 时间所使用的一些技术，不过它没有说明主存和磁盘存储器的管理。本章的前两节包含了 Pep/8 操作系统的完整代码。



图 8-1 Pep/8 系统的内存图

8.1 装载器

操作系统的一个主要功能是管理用户提交的待执行作业。在多用户系统中，多个用户不断地提交作业，操作系统必须决定运行待执行作业中的哪个。在决定接下来执行哪个作业后，它必须把适当的程序装载到主存并把 CPU 的控制交给这个程序来执行。

8.1.1 Pep/8 操作系统

图 8-1 展示了 Pep/8 操作系统在主存中的位置。它包括 RAM 中的系统栈和 I/O 缓冲区，在地址 FC57 处的装载器，FC9B 处的中断服务例程和 FFF8 到 FFFE 的 4 个机器向量。虽然 Pep/8 操作系统说明了装载器的操作，但是它并没有说明操作系统决定运行哪个待执行作业的过程。

本章讲述的操作系统是 475 行汇编语言代码，包括文档注释。一般的实现都是高级语言和汇编语言混合写的，汇编语言是针对该操作系统控制的特定计算机的。通常情况下，系统的 90% 是高级语言写的，10% 是汇编语言写的。汇编语言部分保留给操

作系统中需要使用高级语言所没有的特性编程的部件，或者那些对效率有额外要求的部件，而这个要求甚至优化编译器也不能实现。

图 8-2 展示了 Pep/8 操作系统的全局常量和变量。符号 TRUE 和 FALSE 用 .EQUATE 命令声明，因此不会生成目标代码。它们的使用将贯穿程序剩下的部分。

符号 osRAM、wordBuff、byteBuff、wordTemp、byteTemp、addrMask 和 opAddr 都用 .BLOCK 命令来定义。通常情况下，.BLOCK 生成代码，且所有生成的代码都从 0000（hex）开始。从代码来看，这些 .BLOCK 没有生成代码，且 osRAM 从 FBCF 开始而不是 0000。

这个奇怪的汇编器行为的原因在于 FC57 的 .BURN 命令。当在程序中含有 .BURN 时，汇编器会假定该程序将烧入 RAM，它会为跟在烧入指令后面的指令生成代码，而不会为它前面的指令生成代码。汇编器同时也假设 ROM 将装在内存底部，而把内存的顶部留给应用程序使用。因此它计算符号表的地址，使得生成的最后一个字节的地址是烧入指令指定的地址。在这段代码中，烧入指令指出最后一个字节应该在地址 FFFF 处。图 8-16（见 8.2.9 节）显示最后的字节 9B（hex）确实在地址 FFFF，在操作系统的末尾。因为 FFFF（hex）是 65536（dec），所以 Pep/8 计算机的主存大小配置为 64KB。 [392]

```
              ;******* Pep/8 Operating System
              ;
              TRUE:    .EQUATE 1
              FALSE:   .EQUATE 0
              ;
              ;******* Operating system RAM
FBCF          osRAM:   .BLOCK  128         ;System stack area
FC4F          wordBuff:.BLOCK  1           ;Input/output buffer
FC50          byteBuff:.BLOCK  1           ;Least significant byte of wordBuff
FC51          wordTemp:.BLOCK  1           ;Temporary word storage
FC52          byteTemp:.BLOCK  1           ;Least significant byte of wordTemp
FC53          addrMask:.BLOCK  2           ;Addressing mode mask
FC55          opAddr:  .BLOCK  2           ;Trap instruction operand address
              ;
              ;******* Operating system ROM
FC57                   .BURN   0xFFFF
```

图 8-2 Pep/8 操作系统的全局常量和变量

8.1.2 Pep/8 装载器

图 8-3 展示了 Pep/8 装载器。要调用装载器，就要在仿真器上选择装载选项，这会触发下面两个事件：

SP ← Mem[FFFA]

PC ← Mem[FFFC]

因为 Mem[FFFA] 包含 FC4F，如图 8-1 和图 8-16 所示，所以栈指针被初始化为 FC4F。类似地，程序计数器被初始化为 FC57，即装载器第一条指令的地址。

装载器先将变址寄存器清零，再将 wordBuff 也清零。图 8-2 显示 wordBuff 声明为 1 字节。因为 wordBuff 后面紧跟的 byteBuff 声明为 1 字节，所以程序把 wordBuff 当作一个 2 字节的缓冲区，byteBuff 是它最右边的字节。CHARI 指令输入到 byteBuff，但后面的 LDA 指令从 wordBuff 读入。LDA 将输入的字符移动到累加器，并保证累加器最左边的字节为零。

```
                ;******* System Loader
                ;Data must be in the following format:
                ;Each hex number representing a byte must contain
                ;exactly two characters. Each character must be
                ;'0..9, A..F, or a..f and must be followed by exactly
                ;one space. There must be no leading spaces at the beginning
                ;of a line and no trailing spaces at the end of a line. The last two
                ;characters in the file must be lowercase zz, which is used as the
                ;terminating sentinel by the loader.
                ;
FC57  C00000 loader:  LDA     0,i         ;X := 0
FC5A  C9FC4F          STA     wordBuff,d  ;Clear input buffer word
                ;
FC5D  49FC50 getChar: CHARI   byteBuff,d  ;Get first hex character
FC60  C1FC4F          LDA     wordBuff,d  ;Put ASCII into low byte of A
FC63  B0007A          CPA     'z',i       ;If end of file sentinel 'z'
FC66  0AFC9A          BREQ    stopLoad    ;then exit loader routine
FC69  B00039          CPA     '9',i       ;If character <= '9', assume decimal
FC6C  06FC72          BRLE    shift       ;and right nybble is correct digit
FC6F  700009          ADDA    9,i         ;else convert nybble to correct digit
FC72  1C      shift:  ASLA                ;Shift left by four bits to send
FC73  1C              ASLA                ;the digit to the most significant
FC74  1C              ASLA                ;position in the byte
FC75  1C              ASLA
FC76  F1FC52          STBYTEA byteTemp,d  ;Save the most significant nybble
FC79  49FC50          CHARI   byteBuff,d  ;Get second hex character
FC7C  C1FC4F          LDA     wordBuff,d  ;Put ASCII into low byte of A
FC7F  B00039          CPA     '9',i       ;If character <= '9', assume decimal
FC82  06FC88          BRLE    combine     ;and right nybble is correct digit
FC85  700009          ADDA    9,i         ;else convert nybble to correct digit
FC88  90000F combine: ANDA    0x000F,i    ;Mask out the left nybble
FC8B  A1FC51          ORA     wordTemp,d  ;Combine both hex digits in binary
FC8E  F50000          STBYTEA 0,x         ;Store in Mem[X]
FC91  780001          ADDX    1,i         ;X := X + 1
FC94  49FC50          CHARI   byteBuff,d  ;Skip blank or <LF>
FC97  04FC5D          BR      getChar     ;
                ;
FC9A  00      stopLoad:STOP               ;
```

图 8-3　Pep/8 操作系统的装载器

393 / 394

装载器是一个单循环的形式，它输入一个字符，将它和分隔符 z 进行比较。如果该字符不是分隔符，那么程序检查它是否属于 '0'..'9'，如果不在这个范围内，那么最右的 4 位，通过加 9 转换为正确的值。4 位又称为四位元组（nybble），是半字节。注意 ASCII 的 A 是 0100 0001（bin），因此当它加上 9，和是 0100 1010，最右的半字节是十六进制数 A 的正确位模式，十六进制的 B 到 F 也是类似的情况。如果字符是在 '0'..'9' 中，那么最右半字节已经是正确的值了。

装载器把半字节 4 位移到左边，临时存储在 byteTemp 中。它输入第二个字符，类似地对半字节进行调整，用 FC88 处的 ANDA 把两个半字节合成一个字节。把程序字节放入内存的指令是地址 FC8E 处的 STBYTEA，它使用变址寻址方式，每次循环都把变址寄存器加 1。STOP 指令会终止装载器，将控制返回给仿真器选项。

通常要装载的程序不是十六进制 ASCII 字符形式，它们已经是二进制形式，准备被装

载了。Pep/8 的目标文件使用 ASCII 字符，因此可以直接用机器语言编程，用文本编辑器查看目标文件。

8.1.3 程序的终止

到目前为止，所有出现的应用程序都是以 STOP 指令终止的。在实际的计算机中很少执行 STOP 指令，C++ 编译器不会在程序末尾生成一个 STOP 指令，而是生成一个把控制返回操作系统的指令。如果程序运行在一台个人计算机上，那么操作系统会设置一个屏幕，等待请求另一个服务。如果程序运行在分时共享系统上，那么操作系统会继续处理其他用户的作业。无论哪种情况，计算机都不会只是简单地停下来。

因为仅有一个 CPU，所以它在执行操作系统作业和应用作业之间来回切换。图 8-4 展示了当操作系统装载和执行一系列作业时 CPU 使用的时间线，阴影部分代表用在执行操作系统上的时间。



图 8-4 当操作系统加载和执行一系列作业时的 CPU 使用时间线

操作系统代表执行业务的必要的开销。当在商场购物时，购买商品的价格不只是反映商品的价值和运输到商场的成本，也反映了售货员的薪水、商场照明的电费、商场经理的附加福利等。类似地，计算机资源并不是 100% 地用于执行用户的程序，一部分的资源必须保留给操作系统。目前我们考虑的资源是 CPU 时间。

395

8.2 陷阱

当在 Asmb5 层用汇编语言编程时，可能会用到 DECI、DECO 和 STRO。图 4-6 显示了在 ISA3 机器层没有这样的指令。取而代之的是，当计算机取出具有这样一些操作码（00110 到 01000）的指令时，硬件会执行陷阱。陷阱类似于子程序转移，但是要更复杂一些。执行的代码称为陷阱例程（trap routine）或者陷阱处理程序（trap handler）而不是子程序。操作系统通过执行从陷阱返回的指令 RETTR 而不是子程序返回的指令 RETn 将控制交回给应用程序。

陷阱处理程序实现 3 条指令，就如同它们是 ISA3 机器层的一部分一样。记住操作系统的目的之一就是向高层编程提供方便的环境。Pep/8 操作系统提供的抽象机器是一个更加方便的机器，因为它包含这 3 条 ISA3 层没有的指令。除了 DECI、DECO 和 STRO 外，操作系统还提供 1 个非一元和 4 个一元陷阱指令，叫作空操作。助记符分别是 NOP、NOP0、NOP1、NOP2 和 NOP3。当执行这些指令时什么都不做，提供这些指令是为了让你能够对它们重编程，执行你自己选择的新指令。

8.2.1 陷阱机制

下面是陷阱指令的寄存器传输语言（RTL）描述：

```
Temp              ← Mem[FFFA];
Mem[Temp - 1]     ← IR〈0..7〉;
Mem[Temp - 3]     ← SP;
Mem[Temp - 5]     ← PC;
```

```
Mem[Temp – 7]          ← X;
Mem[Temp – 9]          ← A;
Mem[Temp – 10]⟨4..7⟩   ← NZVC;
SP                     ← Temp – 10;
PC                     ← Mem[FFFE]
```

为了表述方便，Temp 表示临时值。Mem[FFFA] 包含 FC4F，即系统栈的地址。在第一个操作中，Temp 获得 FC4F，接下来的 6 个操作显示 CPU 把所有寄存器的内容都压入系统栈，从 IR 指令指示符开始，到 NZVC 标记位结束。接着栈指针被修改为指向新的系统栈顶部，程序计数器获得 Mem[FFFE] 的内容。

[396]

图 8-5 展示了像图 5-11 那样陷阱的例子。图 5-11 中的程序包含下面的十进制输出陷阱

```
0029  390003 DECO 0x0003,d ;Output the sum
```

这里的 0029 是该指令的地址，390003 是执行中触发该陷阱的目标代码。

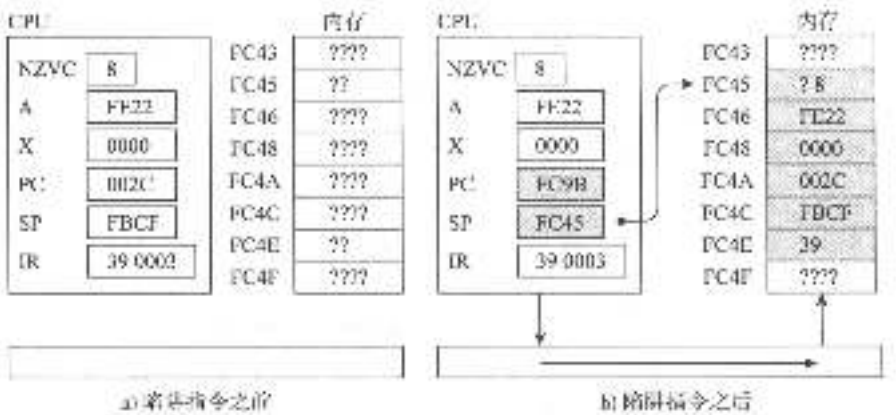


图 8-5　执行 DECO 陷阱指令 390003 触发的陷阱

图 8-5a 展示了陷阱执行前 CPU 的状态，图 8-5b 是陷阱执行后的状态。可以看到只有 IR 的指令指示符部分被压入栈。还可以注意到 4 个 NZVC 位正好位于 Mem[FC45] 处字节的右半部分中，该字节的左半字节未定义。SP 的值是 FC45，即新系统栈的顶部，PC 的内容是 Mem[FFFE]，即陷阱处理程序第一条指令的地址 FC9B。图 8-16（见 8.2.9 节）展示了操作系统怎样用 .ADDRSS 命令在地址 FFFA 和 FFFE 设置机器向量。

8.2.2　RETTR 指令

执行时的程序叫作进程（process）。陷阱机制临时终止一个进程，这样操作系统可以执行服务。主存中包含被陷阱进程寄存器副本的信息块叫作进程控制块（PCB）。这个例子的 PCB 存储在 Mem[FC45] 到 Mem[FC4F] 中，如图 8-5b 所示。

操作系统执行完它的服务后，最后必须把 CPU 的控制交回给被挂起的进程，这样该进程可以继续完成执行。在这个例子中，Pep/8 操作系统执行的服务是执行 DECO 指令。操作系统通过执行陷阱返回指令 RETTR 将控制交回给进程。

[397]

RETTR 的 RTL 描述是

```
NZVC ← Mem[SP]⟨4..7⟩;
A    ← Mem[SP – 1];
```

X ← Mem[SP + 3];
PC ← Mem[SP + 5];
SP ← Mem[SP + 7];

RETTR 把栈上面的 9 个字节弹出栈放入 NZVC、A、X、PC 和 SP 寄存器中。除了不弹出 IR 外，它的操作顺序刚好和陷阱操作的顺序相反。下一条要执行的指令将是新 PC 的值指定的指令。最后修改的寄存器是 SP。

如果陷阱处理程序不修改 PCB 中的任何值，那么当进程重新开始时，RETTR 将恢复 CPU 寄存器的原始值，尤其是 SP，就像在处理陷阱时一样，将重新指向应用程序栈的顶部。另一方面，陷阱处理程序对 PCB 中值的任何改变，在进程重新开始时，都会反映在 CPU 寄存器中。

8.2.3 陷阱处理程序

图 8-6 展示了陷阱处理程序的进入点和退出点。oldIR 是根据陷阱机制存储在系统栈上的 IR 寄存器副本的栈地址。图 8-7a 展示了所有寄存器的栈地址。

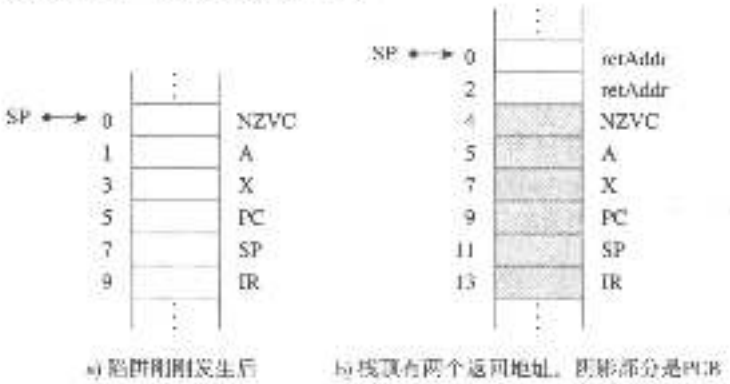
```
                ;******* Trap handler
                oldIR:  .EQUATE 9           ;Stack address of IR on trap
                ;
FC9B  C80000 trap:  LDX     0,i             ;Clear X for a byte compare
FC9E  D00009        LDBYTEX oldIR,s         ;X := trapped IR
FCA1  B80020        CPX     0x0020,i        ;If X >= first nonunary trap opcode
FCA4  CEFCB7        BRGE    nonUnary        ;trap opcode is nonunary
                ;
FCA7  980003 unary: AND     0x0003,i        ;Mask out all but rightmost two bits
FCAA  10            ASLA                    ;An address is two bytes
FCAB  17FCAF        CALL    unaryJT,x       ;Call unary trap routine
FCAE  01            RETTR                   ;Return from trap
                ;
FCAF  FDB6  unaryJT: .ADDRSS opcode24       ;Address of NOP0 subroutine
FCB1  FDB7          .ADDRSS opcode25        ;Address of NOP1 subroutine
FCB3  FDB8          .ADDRSS opcode26        ;Address of NOP2 subroutine
FCB5  FDB9          .ADDRSS opcode27        ;Address of NOP3 subroutine
                ;
FCB7  1F    nonUnary:ASRA                   ;Trap opcode is nonunary
FCB8  1F            ASRA                    ;Discard addressing mode bits
FCB9  1F            ASRA
FCBA  80C005        SUBA    5,i             ;Adjust so that NOP opcode = 0
FCBD  10            ASLX                    ;An address is two bytes
FCBE  17FCC2        CALL    nonUnJT,x       ;Call nonunary trap routine
FCC1  01    return: RETTR                   ;Return from trap
                ;
FCC2  FDBA  nonUnJT: .ADDRSS opcode28       ;Address of NOP subroutine
FCC4  FDC4          .ADDRSS opcode30        ;Address of DECI subroutine
FCC6  FF38          .ADDRSS opcode38        ;Address of DECO subroutine
FCC8  FFC6          .ADDRSS opcode40        ;Address of STRO subroutine
```

图 8-6 Pep/8 操作系统中陷阱处理程序的进入点和退出点

当执行一条陷阱指令时，下一条要执行的指令在 FC9B，即图 8-6 中的第一条指令。下面任何一条指令都能触发陷阱：

0010 01nn，NOPn，一元空操作陷阱
0010 1aaa，NOP，非一元空操作陷阱

0011 0aaa，DECI，非一元十进制输入陷阱
0011 1aaa，DECO，非一元十进制输出陷阱
0100 0aaa，STRO，非一元输出字符串陷阱



a) 陷阱刚刚发生后　　b) 栈顶有两个返回地址。阴影部分是PCB

图 8-7　CPU 寄存器副本的栈地址

图 8-6 中的代码确定哪一条指令触发了陷阱，并调用实现这条指令的特定处理程序。总共有 8 个陷阱处理程序，一元 NOPn 指令有 4 个，非一元指令有 4 个。记住冯·诺依曼循环的取指部分是把指令指示符放在指令寄存器中。陷阱发生后，引发陷阱的指令的指令指示符可以从系统栈上获取，因为根据陷阱机制，它被压到了栈上。图 8-6 中的代码访问被保存的指令指示符来确定哪一条指令触发了该陷阱。

在图 8-6 中第一条指令从被压入系统栈的 IR 副本获取操作码。NOP 指令有第一个非一元操作码 0010 1aaa，且 0010 1000（bin）等于 28（dec）。地址 FCA1 处的 CPX 指令将陷阱代码和 28（dec）进行比较，如果陷阱代码小于 28，那么该陷阱指令是一元的，否则是非一元的。

如果陷阱指令是一元指令，那么它必定是下列 4 条指令之一：

0010 0100，NOP0，最右 2 位是 00
0010 0101，NOP1，最右 2 位是 01
0010 0110，NOP2，最右 2 位是 10
0010 0111，NOP3，最右 2 位是 11

地址 FCA7 的 ANDX 指令将屏蔽除了最右 2 位外的所有位，这 2 位就足以确定 4 条指令中的哪条引发了该陷阱。地址 FCAB 的 CALL 指令使用图 6-40 中程序描述的采用变址寻址的转移表技术。图 6-40 展示了编译器怎样用无条件转移指令 BR 和地址数组来翻译 C++ 的 switch 语句。图 8-6 的代码和图 6-40 的代码稍有不同，因为它使用了 CALL 而不是 BR，但原理是一样的。地址 FCAF 的转移表是一个地址数组，数组中的每个元素是触发该陷阱的特定指令的陷阱处理程序的第一条语句的地址。因为执行 CALL，所以它把返回地址压入栈中。在一个特定的陷阱处理程序中，最后执行的指令是 RETO，它将控制返回到 FCAE。地址 FCAE 处的指令是 RETTR，它从 PCB 恢复 CPU 寄存器，把控制返回该陷阱指令后面的那条指令。

对所有的非一元指令，从 FCB7 到 FCC8 的指令做一样的事情。3 个 ASRX 指令丢弃寻址模式位，SUBX 指令进行调整，把变址寄存器的内容变为

0，如果陷阱 IR 包含 0010 1aaa，NOP
1，如果陷阱 IR 包含 0011 0aaa，DECI
2，如果陷阱 IR 包含 0011 1aaa，DECO

3，如果指令IR包含0100 0aaa，STRO

与一元指令一样，地址FCBE处的CALL会跳转到某条特定指令的陷阱处理程序。陷阱处理程序执行完该指令后，会把控制返回到地址FCC1处的RETTR指令，RETTR接着把控制交回给触发该陷阱的指令后面的那条指令。

398
↓
400

8.2.4 陷阱寻址方式断言

不同的指令有不同的寻址方式。例如，图5-2说明CHARI指令不能用立即数寻址，而STRO指令允许用直接、间接和栈相对间接寻址。因为CHARI指令是固化到CPU中的，所以由硬件检测是否发生了寻址错误。但是陷阱指令，例如STRO，不是CPU原生的。陷阱处理程序用软件来实现它们。那么问题来了，陷阱处理程序怎样检测陷阱指令是否试图使用非法的寻址方式呢？答案是使用图8-8中的寻址方式断言例程。

```
                ;******* Assert valid trap addressing mode
                oldIR4:  .EQUATE 13          ;oldIR + 4 with two return addresses
FCCA  C00001 assertAd:LDA     1,i            ;A := 1
FCCD  D80003          LDBYTEX oldIR4,s       ;X := OldIR
FCD0  980007          ANDX    0x0007,i       ;Keep only the addressing mode bits
FCD3  0AFCD8          BREQ    testAd         ;000 = immediate addressing
FCD6  1C      loop:   ASLA                   ;Shift the 1 bit left
FCD7  880001          SUBX    1,i            ;Subtract from addressing mode count
FCDA  0CFCD6          BRNE    loop           ;Try next addressing mode
FCDD  94FC53 testAd:  ANDA    addrMask,d     ;AND the 1 bit with legal modes
FCE0  0AFCE4          BREQ    addrErr
FCE3  58              RETO                   ;Legal addressing mode, return
FCE4  50000A addrErr: CHARO   '\n',i
FCE7  C0FCF4          LDA     trapMsg,i      ;Push address of error message
FCEA  F3FFFE          STA     -2,s
FCED  680002          SUBSP   2,i            ;Call print subroutine
FCF0  10FFE2          CALL    prntMsg
FCF3  00              STOP                   ;Halt: fatal runtime error
FCF4  455252 trapMsg: .ASCII  "ERROR: Invalid trap addressing mode.\x00"
      ...
```

图8-8 Pep/8操作系统中的陷阱寻址方式断言

寻址方式断言例程必须访问存储在系统栈上的陷阱IR。陷阱发生后，IR的栈地址是9，如图8-7a所示。不过，到调用陷阱寻址方式时，系统栈顶又增加了两个返回地址，一个地址来自图8-6的陷阱处理程序代码中的CALL指令，一个地址来自于特定陷阱处理程序中的CALL。图8-7b展示了寻址方式断言例程被调用后系统栈上的PCB和栈上的两个返回地址。陷阱IR的栈地址现在是13而不是9，因为两个返回地址占用了4字节。

401

图8-8中的例程有如下前提和后置条件：

- 前提条件：addrMask是位掩码，表示允许的寻址方式集合，或指令的PCB在系统栈上。
- 后置条件：如果陷阱指令的寻址方式在允许的寻址方式集合中，那么控制就交回给陷阱处理程序，否则输出无效寻址方式信息，程序中止于致命运行时错误。

寻址方式断言例程是某些HOL6语言assert语句的Asmb5版本。在C++中，断言相关的功能在<asseret>库中，可以在程序中用#include编译器指令包含这个库。

陷阱处理程序使用断言例程，首先如图8-2中地址FC53所示设置全局变量addrMask

的值，使之表示这条指令允许的寻址方式，接着如图8-8中地址FCCA所示调用assertAd。断言例程假定使用一种称为位图表示的、常用的集合表示法。在机器语言中，每位的值为0或者1。允许寻址方式集合的位图表示将一种寻址方式和addrMask中的一位对应。如果该位是0，那么相对应的寻址方式就不在集合中，如果某位的值为1，那么对应的寻址方式在集合中。

图8-9展示了STRO指令的陷阱处理程序预先设置好的addrMask最右的一个字节，该指令允许使用直接、间接和栈相对间接寻址方式。这几种寻址方式对应的位值为1，其他位为0。从数学上说，这个掩码代表了集合{直接，间接，栈相对间接}。

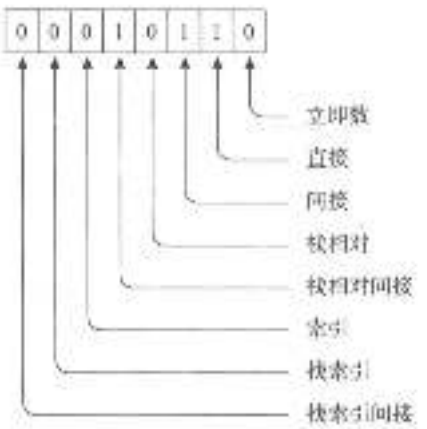


图8-9　addressMaks中的位对应的STRO陷阱指令允许的地址寻址方式

为了说明图8-8中的断言例程怎样测试集合中的成员，假定STRO指令用栈相对间接寻址来执行，这样它的寻址字段就是100，这是一个被允许的寻址方式。首先，地址FCCA的LDA语句把累加器的最右一个字节置为0000 0001，接下来的两条语句根据陷阱指令的寻址-aaa字段把变址寄存器置为4（dec）。接着循环从4开始倒数循环到0，每次循环就把累加器中的那个1位往左移动一位，累加器最后的值是0001 0000，这个1位就在对应于栈相对间接寻址的那个位置。地址FCCD的ANDA语句将图8-9的寻址掩码与累加器进行AND运算，因为从右数第5位的值是1，所以结果为非零，控制返回到陷阱处理程序。如果不允许使用栈相对间接寻址，那么寻址掩码从右数的第5位是0，AND运算的结果是0，于是断言失败。

402

8.2.5　陷阱操作数地址计算

陷阱操作数地址计算是非一元陷阱处理程序调用的另一个例程。原生指令的寻址方式是固化到CPU的，但是陷阱指令以软件而不是硬件的方式实现，因此必须用软件方式来模拟8种寻址方式。图8-10展示了执行这个计算的例程。

```
                ;******* Set address of trap operand
                oldX4:  .EQUATE 7           ;oldX + 4 with two return addresses
                oldPC4: .EQUATE 9           ;oldPC + 4 with two return addresses
                oldSP4: .EQUATE 11          ;oldSP + 4 with two return addresses
FD19  C80000 setAddr: LDBYTER oldIR4,s      ;X <- old instruction register
FD1C  C80007          ANDX    0x0007,i      ;Keep only the addressing mode bits
FD1F  18              ASLX                  ;An address is two bytes
FD20  05FD23          BR      addrJT,x
FD23  FD33  addrJT:   .ADDRSS addrI         ;Immediate addressing
FD25  FD3D            .ADDRSS addrD         ;Direct addressing
FD27  FD4A            .ADDRSS addrN         ;Indirect addressing
FD29  FD5A            .ADDRSS addrS         ;Stack-relative addressing
FD2B  FD6A            .ADDRSS addrSF        ;Stack-relative deferred addressing
FD2D  FD7D            .ADDRSS addrX         ;Indexed addressing
FD2F  FD8D            .ADDRSS addrSX        ;Stack-indexed addressing
FD31  FDA0            .ADDRSS addrSXF       ;Stack-indexed deferred addressing
                ;
```

图8-10　Pep/8操作系统中陷阱操作数地址计算

```
FD33  C80009 addr1:   LDX     oldPC4,s   ;Immediate addressing
FD36  880002          SUBX    2,+        ;Oprnd = OprndSpec
FD39  E9FC55          STX     opAddr,d
FD3C  38              RETO
              ;
FD3D  C80009 addr0:   LDX     oldPC4,s   ;Direct addressing
FD40  880002          SUBX    2,+        ;Oprnd = Mem[OprndSpec]
FD43  CD0000          LDX     0,x
FD46  E9FC55          STX     opAddr,d
FD49  38              RETO
              ;
FD4A  C80009 addrN:   LDX     oldPC4,s   ;Indirect addressing
FD4D  880002          SUBX    2,+        ;Oprnd = Mem[Mem[OprndSpec]]
FD50  ED0000          LDX     0,x
FD53  ED0000          LDX     0,x
FD56  E9FC55          STX     opAddr,d
FD59  38              RETO
              ;
FD5A  C80009 addrS:   LDX     oldPC4,s   ;Stack-relative addressing
FD5D  880002          SUBX    2,+        ;Oprnd = Mem[SP + OprndSpec]
FD60  ED0000          LDX     0,x
FD63  7B0008          ADDX    oldSP4,s
FD66  E9FC55          STX     opAddr,d
FD69  38              RETO
              ;
FD6A  C80009 addrSI:  LDX     oldPC4,s   ;Stack relative deferred addressing
FD6D  880002          SUBX    2,+        ;Oprnd = Mem[Mem[SP + OprndSpec]]
FD70  ED0000          LDX     0,x
FD73  7B0008          ADDX    oldSP4,s
FD76  ED0000          LDX     0,x
FD79  E9FC55          STX     opAddr,d
FD7C  38              RETO
              ;
FD7D  C80009 addrX:   LDX     oldPC4,s   ;Indexed addressing
FD80  880002          SUBX    2,+        ;Oprnd = Mem[OprndSpec + X]
FD83  ED0000          LDX     0,x
FD86  7B0007          ADDX    oldX4,s
FD89  E9FC55          STX     opAddr,d
FD8C  38              RETO
              ;
FD8D  C80009 addrSX:  LDX     oldPC4,s   ;Stack-indexed addressing
FD90  880002          SUBX    2,+        ;Oprnd = Mem[SP + OprndSpec + X]
FD93  ED0000          LDX     0,x
FD96  7B0007          ADDX    oldX4,s
FD99  7B0008          ADDX    oldSP4,s
FD9C  E9FC55          STX     opAddr,d
FD9F  38              RETO
FDA0  C80009 addrSXF: LDX     oldPC4,s   ;Stack-indexed deferred addressing
FDA3  880002          SUBX    2,+        ;Oprnd = Mem[Mem[SP + OprndSpec] + X]
FDA6  ED0000          LDX     0,x
FDA9  7B0008          ADDX    oldSP4,s
FDAC  ED0000          LDX     0,x
FDAF  7B0007          ADDX    oldX4,s
FDB2  E9FC55          STX     opAddr,d
FDB5  38              RETO
```

例 8-10 (续)

图 8-10 的例程有下列的前提和后置条件：

- 前提条件：栈指令的 PCB 在系统栈上。

● 后置条件：opAddr 包含根据陷阱指令的寻址方式计算的操作数地址。

与图 8-8 中寻址方式断言例程一样，PCB 中寄存器副本的栈偏移量比陷阱刚发生时大了 4 字节，如图 8-7b 所示。这个例程用寻址方式断言例程中定义的 oldIR4，类似的还有 oldX4，oldPC4 和 oldSP4，分别来访问保存的寄存器副本、程序计数器和栈指针。

从 FD19 开始的前 4 条指令确定陷阱指令的寻址方式，并转移到该种寻址方式对应的计算，使用的是转移表技术在 8 种可能性中进行选择。8 种选择中每一个的代码都通过检查陷阱发生时 CPU 的状态来计算操作数的地址。

每个计算的前两条指令都是

```
LDX oldPC4,s
SUBX 2,1
```

因为陷阱指令是非一元的，所以陷阱发生时程序计数器指向 2 字节操作数指示符后面的那个字节。第一条指令把保存的程序计数器装载到变址寄存器，第二条指令将它减去 2。这两条指令执行后，变址寄存器包含引发陷阱的指令中的操作数指示符的地址。

对于立即数寻址，操作数指示符就是操作数，因此地址 FD39 的语句

```
STX opAdcr,d
RETG
```

403
~
405

只是如要求的那样把操作数指示符的地址存储到 opAddr 中。

对于直接寻址，操作数指示符是操作数的地址，地址 FD43 开始的语句

```
LDX 0,x
STX opAdcr,d
RETG
```

中的第一条用变址寄存器中地址所指向内存的内容替换变址寄存器。在指令执行前，变址寄存器包含的是操作数指示符的地址，在指令执行后，变址寄存器包含的是操作数指示符本身。因为操作数指示符是操作数的地址，所以就把它存储在 opAddr 中。

对于间接寻址，操作数指示符是操作数地址的地址，和直接寻址一样，地址 FD50 开始的语句

```
LDX 0,x
LDX 0,x
STX opAddr,d
RETG
```

中的第一条用操作数指示符替换变址寄存器的内容，即操作数地址的地址。第二条语句获取操作数的地址，存储在 opAddr 中。

对于栈相对寻址，栈指针加上操作数指示符等于操作数的地址。地址 FD60 开始的语句

```
LDX 0,x
ADDX oldSP4,s
STX opAddr,d
RETG
```

中的第一条把操作数指示符放进变址寄存器中，第二条指令把它加上栈指针的内容，结果就是操作数的地址，将其存储在 opAddr 中。

剩下的 4 种寻址方式用类似的技术来计算操作数的地址。栈相对间接寻址比栈相对寻址多一层间接的过程，要求多执行一条 LDX 0,x。除了把操作数指示符而不是栈指针加到变址寄存器上以外，变址寻址和栈相对寻址是一样的。栈变址和栈变址间接寻址也都是类似的变类。

8.2.6 空操作陷阱处理程序

图 8-11 展示了执行空操作陷阱处理程序的代码。因为 NOP 指令不做任何事情，所以陷阱处理程序除了执行 RETO 把控制返回图 8-6 的出口点并最终返回给陷阱语句后面的语句外，不做任何其他处理。 406

提供 NOP 指令是为了允许我们编写自己的陷阱处理程序。本章末尾的一些问题会让你实现一些 Pep/8 指令集中没有的指令。Pep/8 汇编器让你重新定义陷阱指令的助记符。要写一个陷阱处理程序，把图 8-11 中的一个 NOP 指令的助记符变成新指令的助记符，接着编辑操作系统中的陷阱处理程序，在入口点插入你的代码。例如，要重新定义 NOP0，在 FDB6 插入你的处理程序的代码。处理程序最后一条可执行语句应该是 RETO。

```
                ;******* Opcode 0x24
                ;The NOP0 instruction.
FDB6  58        opcode24:RETO
                ;
                ;******* Opcode 0x25
                ;The NOP1 instruction.
FDB7  58        opcode25:RETO
                ;
                ;******* Opcode 0x26
                ;The NOP2 instruction.
FDB8  58        opcode26:RETO
                ;
                ;******* Opcode 0x27
                ;The NOP3 instruction.
FDB9  58        opcode27:RETO
                ;
                ;******* Opcode 0x28
                ;The NOP instruction.
FDBA  C00001    opcode28:LDA     0x0001,i   ;Assert 1
FDBD  E1FDC3             STA     addrMask,d
FDC0  16FDCA             CALL    assertAd
FDC3  58                 RETO
```

图 8-11 NOP 陷阱处理程序

图 8-11 展示了地址 FDBA 处的非一元 NOP 的实现。图 5-2 说明了它唯一允许的寻址方式是立即数寻址，因此 addrMask 的值被置为 0000 0001，这里最后的 1 所在的位对应于立即数寻址，如图 8-9 所示。 407

8.2.7 DECI 陷阱处理程序

图 8-14 是 DECI 指令的陷阱处理程序。DECI 必须对输入进行语法分析，把 ASCII 字符串转换成适当的补码表示的位。它使用图 8-12 的有限状态机（FSM），而图 8-13 是 DECI 陷阱处理程序的 FSM 的逻辑框架。state 是枚举类型，可能的值是 init、sign 或 digit。

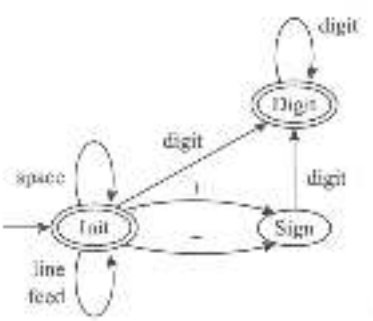


图 8-12 DECI 中断处理程序中的有限状态机

在图 8-14 中，从地址 FDC4 开始的 4 条语句调用寻址方式断言例程和计算陷阱操作数地址例程。在地址 FDD0，处理程序在栈上分配了 6 个局部变量——total、valAscii、

isOvfl、isNeg、state 和 temp，每个变量占 2 字节，为此 SUBSP 将栈指针减去 12。对于应用程序来说，SUBSP 是程序中第一条使用局部变量的可执行语句，SUBSP 必须在前面两个程序调用之后执行，因为这两个调用会访问来自 PCB 的数据，它们假定栈上只有两个返回地址，如图 8-7b 所示。

DECI 陷阱处理程序必须通过 PCB 访问 NZVC 位。地址 FCBE 的 CALL 指令调用此处理程序，CALL 指令把 2 字节的返回地址压入栈。当处理程序访问陷阱发生时存储在栈上的 NZVC 值时，由于局部变量和返回地址的原因，所以它的栈地址将比陷阱刚刚发生后大 14。这就是为什么老的 NZVC 等于 14 而不是 0。

```
isOvfl := false
state := init
do
   CHAR( asciiCh
   switch state
   case init:
      if (asciiCh == '+') {
         isNeg := false
         state := sign
      }
      else if (asciiCh == '-') {
         isNeg := true
         state := sign
      }
      else if (asciiCh is a digit) {
         isNeg := false
         total := Value (asciiCh)
         state := digit
      }
      else if (asciiCh is not <SPACE> or <LF>) {
         Exit with DECI error
      }
   case sign:
      if (asciiCh is a digit) {
         total := Value (asciiCh)
         state := digit
      }
      else {
         Exit with DECI error
      }
   case digit:
      if (asciiCh is a digit) {
         total := 10 * total + Value (asciiCh)
         if (overflow) {
            isOvfl := true
         }
      }
      else {
         Exit normally
      }
   end switch
while (not exit)
```

图 8-13　DECI 陷阱处理程序的程序逻辑

```
                ;******* Opcode 0x30
                ;the DECI instruction.
                ;Input format: Any number of leading spaces or line feeds are
                ;allowed, followed by '+', '-', or a digit as the first character,
                ;after which digits are input until the first nondigit is
                ;encountered. The status flags N, Z, and V are set appropriately
                ;by this DECI routine. The C status flag is not affected.
                ;
                oldNZVC: .EQUATE 14         ;Stack address of NZVC on interrupt
                ;
                total:   .EQUATE 10         ;Cumulative total of DECI number
                valAscii:.EQUATE 8          ;Value(asciiCh)
                isOvfl:  .EQUATE 6          ;Overflow boolean
                isNeg:   .EQUATE 4          ;Negative boolean
                state:   .EQUATE 2          ;State variable
                temp:    .EQUATE 0
                ;
                init:    .EQUATE 0          ;Enumerated values for state
                sign:    .EQUATE 1
                digit:   .EQUATE 2
                ;
FDC4  C000FE opcode30:LDA     0x00FE,i   ;Assert d, n, s, sf, x, sx, sxf
FDC7  81FE93          STA     addrMask,d
FDCA  16FDCA          CALL    assertAd
FDCD  16FD19          CALL    setAddr    ;Set address of trap operand
FDD0  680000          SUBSP   12,i       ;Allocate storage for locals
FDD3  C00000          LDA     FALSE,i    ;isOvfl := FALSE
FDD6  E30006          STA     isOvfl,s
FDD9  C00000          LDA     init,i     ;state := init
FDDC  E30002          STA     state,s
FDDF  C00000          LDA     0,i        ;wordBuff := 0 for input
FDE2  C1FD4F          STA     wordBuff,d
                ;
FDE5  49FDB0 do:      CHARI   byteBuff,d ;Get asciiCh
FDE8  C1FD4F          LDA     wordBuff,d ;Set value(asciiCh)
FDEB  90000F          ANDA    0x000F,i
FDEE  E30008          STA     valAscii,s
FDF1  C1FD4F          LDA     wordBuff,d ;A = asciiCh throughout the loop
FDF4  C80007          LDX     state,s    ;switch (state)
FDF7  1D              ASLX                ;An address is two bytes
FDF8  06FDFB          BR      stateJT,x
                ;
FDFB  FE01    stateJT: .ADDRSS sInit
FDFD  FE58             .ADDRSS sSign
FDFF  FE76             .ADDRSS sDigit
                ;
FE01  B0002B sInit:   CPA     '+','i     ;if (asciiCh == '+')
FE04  0CFE16          BRNE    ifMinus
FE07  CB0000          LDA     FALSE,i    ;isNeg := FALSE
FE0A  EB0004          STA     isNeg,s
FE0D  CB0001          LDA     sign,i     ;state := sign
FE10  EB0002          STA     state,s
FE13  0AFDE5          BR      do
                ;
FE16  B0002D ifMinus: CPA     '-',i      ;else if (asciiCh == '-')
FE19  0CFE20          BRNE    ifDigit
```

图 8-14 DECI 陷阱处理程序

```
FE1C  C80001          LDX     TRUE,i      ;isNeg := TRUE
FE1F  E80004          STX     isNeg,s
FE22  C80001          LDX     sign,i      ;state := sign
FE25  E80002          STX     state,s
FE28  04FDF5          BR      do
                ;
FE2B  B00030 ifDigit: CPA     '0',i       ;else if (asciiCh is a digit)
FE2E  08FE4C          BRLT    ifWhite
FE31  B00039          CPA     '9',i
FE34  10FE4C          BRGT    ifWhite
FE37  C80000          LDX     FALSE,i     ;isNeg := FALSE
FE3A  E80004          STX     isNeg,s
FE3D  C80008          LDX     valAscii,s  ;total := Value(asciiCh)
FE40  E8000A          STX     total,s
FE43  C80007          LDX     digit,i     ;state := digit
FE46  E80002          STX     state,s
FE49  04FDE5          BR      do
                ;
FE4C  B00020 ifWhite: CPA     ' ',i       ;else if (asciiCh is not a space
FE4F  0AFDF5          BREQ    do
FE52  B0000A          CPA     '\n',i      ;or line feed)
FE55  0CFF11          BRNE    deciErr     ;exit with DECI error
FE58  04FDE5          BR      do
                ;
FE5B  B00030 sSign:   CPA     '0',i       ;if asciiCh (is not a digit)
FE5E  08FF11          BRLT    deciErr
FE61  B00039          CPA     '9',i
FE64  10FF11          BRGT    deciErr     ;exit with DECI error
FE67  C80008          LDX     valAscii,s  ;else total := Value(asciiCh)
FE6A  E80004          STX     total,s
FE6D  C80002          LDX     digit,i     ;state := digit
FE70  E80002          STX     state,s
FE73  04FDE5          BR      do
                ;
FE76  B00030 sDigit:  CPA     '0',i       ;if (asciiCh is not a digit)
FE79  08FEC7          BRLT    deciNorm
FE7C  B00039          CPA     '9',i
FE7F  10FEC7          BRGT    deciNorm    ;exit normally
FE82  C80001          LDX     TRUE,i      ;else X := TRUE for later assignments
FE85  C3C00A          LDA     total,s     ;Multiply total by 10 as follows:
FE88  1C              ASLA                ;First, times 2
FE89  12FE8F          BRV     ovfl1       ;if overflow then
FE8C  04FE92          BR      L1
FE8F  E80006 ovfl1:   STX     isOvfl,s    ;isOvfl := TRUE
FE92  E3C000 L1:      STA     temp,s      ;Save 2 * total in temp
FE95  1C              ASLA                ;Now, 4 * total
FE96  12FE9C          BRV     ovfl2       ;if overflow then
FE99  04FE9F          BR      L2
FE9C  E80006 ovfl2:   STX     isOvfl,s    ;isOvfl := TRUE
FE9F  1C      L2:     ASLA                ;Now, 8 * total
FEA0  12FEA6          BRV     ovfl3       ;if overflow then
FEA3  04FEA9          BR      L3
FEA6  E80006 ovfl3:   STX     isOvfl,s    ;isOvfl := TRUE
FEA9  730000 L3:      ADDA    temp,s      ;Finally, 8 * total + 2 * total
```

图 8-14 （续）

```
FEAC  32FF82          BRV     ovfl4       ;If overflow then
FEAF  04FF85          BR      L4
FEB2  680C09 ovfl4:   STA     isOvfl,s    ;isOvfl := TRUE
FEB5  730C08 L4:      ADDA    valAscii,s  ;A := 10 * total + valAscii
FEB8  32FEBE          BRV     ovfl5       ;If overflow then
FEBB  04FEC1          BR      L5
FEBE  680C06 ovfl5:   STA     isOvfl,s    ;isOvfl := TRUE
FEC1  630C0A L5:      STA     total,s     ;Update total
FEC4  04FDE5          BR      do
                ;
FEC7  C30C04 decInRng:LDA     isNeg,s     ;If isNeg then
FECA  04FEE3          BREQ    setNZ
FECD  C30C0A          LDA     total,s     ;If total != 0x8000 then
FED0  B08000          CPA     0x8000,i
FED3  04FED9          BREQ    L6
FED6  1A              NEGA                ;Negate total
FED7  630C0A          STA     total,s
FEDA  04FEE3          BR      setNZ
FEDD  C08000 L6:      LDA     FALSE,i     ;else -32768 is a special case
FEE0  630C05          STA     isOvfl,s    ;isOvfl := FALSE
                ;
FEE3  080C0E setNZ:   LDBYTEX oldNZVC,s   ;Set NZ according to total result:
FEE6  980C01          ANDX    0x0C01,i    ;first initialize NZV to 000
FEE9  C30C0A          LDA     total,s     ;If total is negative then
FEEC  0EFEF2          BRGE    checkZ
FEEF  A80C08          ORX     0x0C08,i    ;set N to 1
FEF2  B00C00 checkZ:  CPA     0,i         ;If total is not zero then
FEF5  0CFEF8          BRNE    setV
FEF8  A80C04          ORX     0x0C04,i    ;set Z to 1
FEFB  C30C06 setV:    LDA     isOvfl,s    ;If not isOvfl then
FEFE  0AFF04          BREQ    storeFl
FF01  A80C02          ORX     0x0C02,i    ;set V to 1
FF04  F80C0E storeFl: STBYTEX oldNZVC,s   ;Store the NZVC flags
                ;
FF07  C30C04 exitDec: LDA     total,s     ;Put total in memory
FF0A  E2FC5A          STA     opAddr,n
FF0D  60000C          ADDSP   12,i        ;Deallocate locals
FF10  50              RET0                ;Return to trap handler
                ;
FF11  53000A decErr:  CHARO   '\n',i
FF14  C0FF21          LDA     decMsg,i    ;Push address of message onto stack
FF17  E3FFFE          STA     -2,s
FF1A  680002          SUBSP   2,i
FF1D  16FFF2          CALL    prntMsg     ;and print
FF20  00              STOP                ;Fatal error: program terminates
                ;
FF21  455252 decMsg:  .ASCII  "ERROR: Invalid DECI input\x00"
      ...
```

图 8-14 （续）

DECI中断处理程序从地址FDD3的LDA语句开始，它的处理遵循图8-13的逻辑。例程检测值超出范围的输入字符串，如果检测到了，就在陷阱期间把PCB中的V位设置为1。当RETTR将控制返回给应用程序后，Asmb5层程序员将能够检测执行DECI后的溢出。isOvfl是一个布尔标识，指示是否发生了溢出。

地址 FDE5 是 FSM 循环的开始，从 do 符号可以看出来。FDEB 的 ANDA 屏蔽了最右 4 位以外其余的输入字符，留下的是十进制 ASCII 数字对应的二进制值。例如，ASCII 的 5，它的二进制表示是 0011 0101，最右 4 位 0101 是十进制数相应的二进制值。地址 FDF1 处，累加器获得该 ASCII 字符，并在整个循环过程中保持它。地址 FDFB 的 stateJT 是 FSM 中 switch 语句的转移表。

从地址 FE01 到 FE58 的代码是 state 为值 sInit 的情况，即 FSM 的起始状态。因为整个循环过程中累加器要保持 ASCII 字符用于比较，所以所有的赋值语句都是通过变址寄存器而不是累加器进行。例如，地址 FE07 的 isNeg 赋值为 FALSE 是通过 LDX 后面跟 STX 而不是 LDA 后面跟 STA 来执行的。

408 ~ 413

从地址 FE5B 到 FE73 的代码是 state 为值 sSign 的情况。从地址 FE76 到 FEC4 是 state 为值 sDigit 的情况。Pep/8 没有把一个值乘以 10（dec）的指令，这段代码用多个左移运算来执行这个乘法。每个 ASLA 把值乘以 2，3 个 ASLA 运算将值乘以 8，把这个值加上原始值乘以 2 就得到这个值乘以 10 的值。在 ASLA 运算和加法运算之后，该例程会检测溢出并相应地设置 isOvfl。

从地址 FEC7 到 FF20 的代码在循环之外。在两种情况下算法会退出循环：正常退出或者检测到输入错误。如果正常退出，它会检测 isNeg 看数字串前面是否有负号，如果有，地址 FED6 的指令通过取补码将这个数取负。

数字 32 768（dec）等于 8000（hex），必须当作一种特殊情况来处理。如果输入是 –32 768，当地址 FEB5 的 32 760 加 8 时，FSM 将把 isOvfl 设置为 true。问题是尽管 –32 768 在范围内，但 32 768 超出范围。该例程在地址 FEE0 为这种特殊情况调整 isOvfl。

从地址 FEE3 到 FF04 的代码需要进入陷阱时存储的 N、Z 和 V 标识位的副本。地址 FEE6 的 ANDX 把 NZV 设置为 000。注意掩码是 01（hex），即 0000 0001（bin）。因为 C 是最右的位，所以 AND 运算后它保持不变。地址 FEE9 的 LDA 把已分析的值放入累加器，并相应地设置 CPU 中 N、Z 和 V 位的当前值。代码把 PCB 中 N 和 Z 的副本设置成等于 CPU 中 N 和 Z 的当前值，根据先前分析中计算的 isOvfl 值设置 V 的副本。

现在已经输入和分析了该十进制值，陷阱处理程序必须要把它存储在内存中，存储地址是由引发陷阱的 DECI 的操作数指定的。FF07 处的指令

```
LDA total,s
STA opAddr,n
```

执行这个存储。LDA 把计算出的值装载到累加器，STA 用间接寻址把它存储到 opAddr，对于间接寻址方式操作数指示符是操作数的地址的地址。回想先前在 FDCD 处，计算了操作数的地址，存储在 opAddr 中，因此 opAddr 就是操作数的地址的地址，它正是我们所需要的。

当输入字符串不能正常分析时，执行从 FF11 到 FF20 的代码，它会通过调用 prntMsg 输出错信息。如图 8-16 所示，prntMsg 是一个输出以零结尾字符串并立即终止应用程序的程序。

8.2.8　DECO 陷阱处理程序

图 8-15 是 DECO 指令的陷阱处理程序。这个程序输出 DECO 的操作数，输出格式等价于 C++ 的对整数值的 cout <<。因为能存储的最大值是 32 767，所以这个例程最多输出 5 个字符。如果需要，会在数值前面加上负号。

414

```
                ;******* Opcode 0x30
                ;The DECO instruction.
                ;Output format: If the operand is negative, the algorithm prints
                ;a single '-' followed by the magnitude. Otherwise it prints the
                ;magnitude without a leading '+'. It suppresses leading zeros.
                ;
                remain:  .EQUATE 0          ;Remainder of value to output
                chOut:   .EQUATE 2          ;Has a character been output yet?
                place:   .EQUATE 4          ;Place value for division
                ;
FF33  C0007F opcode30:LDA     0x00FF,i   ;Assert i, d, r, s, sf, x, sx, sxf
FF36  B1FC53          STA     addrMask,d
FF41  16FC0A          CALL    assertAd
FF44  16FC19          CALL    setAddr    ;Set address of trap operand
FF47  680006          SUBSP   6,'        ;Allocate storage for locals
FF4A  C2FC65          LDA     opAddr,n   ;A := oprnd
FF4D  800000          CPA     0,'        ;If oprnd is negative then
FF8D  0EFF57          BRGE    printMag
FF53  500020          CHARO   '-',i      ;Print leading '-' and
FF55  1A              NEGA               ;make magnitude positive
FF57  E30000 printMag:STA     remain,s   ;remain := abs(oprnd)
FF54  C00000          LDA     FALSE,i    ;Initialize chOut := FALSE
FF60  E30002          STA     chOut,s
FF60  C02710          LDA     10000,i    ;place := 10,000
FF63  E30004          STA     place,s
FF66  16FF91          CALL    divide     ;Write 10,000's place
FF69  C003E8          LDA     1000,i     ;place := 1,000
FF6C  E30004          STA     place,s
FF6F  16FF91          CALL    divide     ;Write 1000's place
FF72  C00064          LDA     100,i      ;place := 100
FF76  E30004          STA     place,s
FF78  16FF91          CALL    divide     ;Write 100's place
FF7B  C0000A          LDA     10,i       ;place := 10
FF7E  E30004          STA     place,s
FF81  16FF91          CALL    divide     ;Write 10's place
FF84  C30000          LDA     remain,s   ;Always write 1's place
FF87  A00030          ORA     0x0030,i   ;Convert decimal to ASCII
FF8A  F1FC50          STBYTEA byteBuff,d
FF8D  51FC50          CHARO   byteBuff,d
FF90  5E              RET6
                ;
                ;Subroutine to print the most significant decimal digit of the
                ;remainder. It assumes that place (place2 here) contains the
                ;decimal place value. It updates the remainder.
                ;
                remain2: .EQUATE 2          ;Stack addresses while executing a
                chOut2:  .EQUATE 4          ;subroutine are greater by two because
                place2:  .EQUATE 6          ;the retAddr is on the stack
                ;
FF91  C30002 divide:  LDA     remain2,s  ;A := remainder
FF94  C80000          LDX     0,i        ;X := 0
FF97  830006 divLoop: SUBA    place2,s   ;Division by repeated subtraction
FF9A  08FFA6          BRLT    writeNum   ;If remainder is negative then done
FF9D  780001          ADDX    1,i        ;X := X + 1
FFA0  E30002          STA     remain2,s  ;Store the new remainder
```

图 8-15 DECO 陷阱处理程序

```
FFA3  04FF97          BR      divLoop
                ;
FFA6  BB0000 writeNum:CPX     0,i         ;if X != 0 then
FFA9  0AFFB5          BRLE    checkOut
FFAC  C00001          LDA     TRUE,i      ;chOut := TRUE
FFAF  E30004          STA     chOut2,s
FFB2  04FFBC          BR      printDgt    ;and branch to print this digit
FFB5  C30004 checkOut:LDA     chOut2,s    ;else if a previous char was output
FFB8  0CFFBC          BRNE    printDgt    ;then branch to print this zero
FFBB  56              RETC                ;else return to calling routine
                ;
FFBC  AB0030 printDgt:ORX     0x0030,i    ;Convert decimal to ASCII
FFBF  E9FFC4          STX     wordBuff,d  ;for output
FFC2  51FFC0          CHARO   byteBuff,d
FFC5  56              RETC                ;return to calling routine
```

图 8-15 （续）

通常，陷阱处理程序从在 FF3B 开头的语句判断寻址方式是否合法，调用例程计算操作数的地址，给局部变量分配存储空间。和 DECI 陷阱处理程序相比，FF4A 的语句

```
LDA opAddr,s
```

415 / 416

用装载指令而不是存储指令来访问操作数，因为 DECO 是一个输出语句而不是输入语句。和 DECI 处理程序一样，采用间接寻址方式通过 opAddr 访问操作数。

从 FF4D 到 FF56 的代码检测是否是负值。如果操作数为负，FF53 的 CHARO 输出负号，接着后面的代码对操作数取负。在 FF57 累加器包含操作数的大小，它存储在 remain 中，代表余数。

从 FF5A 到 FF81 的代码写出操作数大小的万位、千位、百位和十位。为了防止在最开头出现 0，将 chOut 初始化为 false，表示还没有输出任何数字字符。

子程序 divide 输出 place 位置值上的数字字符，减小 remain 用于下次调用。例如，如果在 FF66 处的调用 divide 之前 remain 的值是 24 873，那么 divide 将输出 2，并将 remain 变为 4873，也把 chOut 设置为 true。

在输出字符 0 之前，divide 检测 chOut 是否已经输出了数字字符。如果 chOut 为 false，那么该字符是前导 0，不能输出；否则是中间的 0，可以输出。例如，如果在 FF6F 处的调用之前 remain 是 761，那么 divide 什么都不输出，remain 继续保持为 761，chOut 也继续保持为 false。不管 chOut 的值是什么，从 FF84 开始的代码都写个位，因此如果原始操作数的值是 0，就输出 0。

从 FF91 到 FFC5 的代码是输出 remain 最高有效位的子程序。它通过反复从 remain 减去 place 来确定输出值，记录减法的次数，直到 remain 小于 0。它的作用是计算 remain/place 输出的值。

8.2.9　STRO 陷阱处理程序和 OS 向量

图 8-16 是 STRO 指令的陷阱处理程序。在功能上类似于 DECO 陷阱处理程序。因为它是一条输出指令，所以首先用 FFD2 的指令

```
LDA opAddr,d
```

```
                ;******* Opcode 0x40
                ;The STRO instruction.
                ;Outputs a null terminated string from memory.
                ;
FFC8  C00016 opcode40:LDA     0x0016,i   ;Assert d, n, sf
FFC9  E1FCB2          STA     addrMask,d
FFCC  16FCCA          CALL    assertAd
FFCF  16FD19          CALL    setAddr    ;Set address of trap operand
FFD2  C1FC56          LDA     opAddr,d   ;Push address of string to print
FFD5  E3FFFE          STA     -2,s
FFD8  600002          SUBSP   2,i
FFDB  16FFE2          CALL    prntMsg    ;and print
FFDE  600002          ADDSP   2,i
FFE1  58              RETO
                ;
                ;******* Print subroutine
                ;Prints a string of ASCII bytes until it encounters a null
                ;byte (eight zero bits). Assumes one parameter, which
                ;contains the address of the message.
                ;
                msgAddr: .EQUATE 2          ;Address of message to print
                ;
FFE2  C80000 prntMsg: LDX     0,i        ;X := 0
FFE5  C80000          LDA     0,i        ;A := 0
FFE8  D70002 prntMore:LDBYTEA msgAddr,sxf;Test next char
FFEB  0AFFF7          BREQ    exitPrnt   ;If null then exit
FFEE  570002          CHARO   msgAddr,sxf;else print
FFF1  780001          ADDX    1,i        ;X := X + 1 for next character
FFF4  04FFEE          BR      prntMore
                ;
FFF7  58      exitPrnt:RETO
                ;
                ;******* Vectors for System Memory Format
FFF8  FBCF            .ADDRSS osRAM      ;User stack pointer
FFFA  FC4F            .ADDRSS wordBuff   ;System stack pointer
FFFC  FC57            .ADDRSS loader     ;Loader program counter
FFFE  FC0B            .ADDRSS trap       ;Trap program counter
```

图 8-16 STRO 指令的陷阱处理程序

获取操作数的地址，然后把要输出的字符串的地址压入运行时栈，调用 FFE2 的 prntMsg 子例程。实际上，字符串就是一个字符数组，所以这个过程类似于一个以数组作为参数传递的 C++ 程序的翻译。因此输出子例程在语句

```
LDBYTEA msgAddr,sxf
```

中使用栈变址间接寻址方式，而语句

```
CHARO msgAddr,sxf
```

访问字符数组的一个元素。

机器向量是用 .ADDRSS 汇编指令建立的，把这个代码与图 8-1 和图 8-2 进行比较，你会看到 FFF8 处的向量是 osRAM 的地址，它是操作系统 RAM 最顶部的字节。当用户选择模拟器的执行选项时，硬件将 SP 初始化为这个值，它是用户堆栈最底部的字节。

FFFA 的向量是 wordBuff 的地址。图 8-2 显示 wordBuff 是预留给系统栈的 128 字节存储块下面的字节。当用户选择模拟器的装载选项时，硬件把 SP 初始化为这个值，当执行

陷阱指令时，从这个点开始把 PCB 压入栈。

FFFC 的向量是装载器的地址，如图 8-3 所示。当用户选择装载选项时硬件将 PC 初始化为这个值。FFFE 的向量是中断处理程序进入点的地址，如图 8-6 所示。当执行陷阱指令时，硬件将 PC 初始化为这个值。

8.3　并发进程

记住进程是执行时的程序。8.2 节展示了操作系统在进程的执行过程中如何暂停它来提供服务。对于一个使用 DECI 和 DECO 的进程，它的 CPU 活动时间线如图 8-17 所示。阴影部分代表 CPU 执行陷阱服务例程的时间。除了这个图展示了操作系统在进程结束前暂停它，然后在这个服务完成后继续此进程外，它在形式上类似于图 8-4 的时间线。

417
:
419

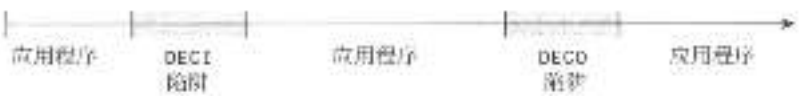


图 8-17　当操作系统执行一个包含 DECI 和 DECO 指令的程序时，CPU 使用的时间线

因为正在执行的进程是通过操作系统代码中未实现的操作码来初始化这些陷阱的，所以 8.2 节中描述的陷阱叫作软中断（software interrupt）。它们也称为同步中断（synchronous interrupt），因为每次执行进程时中断同时发生，中断和代码是同步的，是可以预知的。

另一个初始化同步中断的方法是执行一个操作系统调用。操作系统调用通常的汇编层助记符是 SVC，它代表管理程序调用（supervisor call）。操作数指示符一般作为系统调用的参数，告诉系统程序想请求哪个服务。例如，如果你想用与 C++ 中的 cout.Flush() 对等的函数来刷新缓冲区中流的内容，Flush() 的代码是 27，因此你可以执行

```
SVC 27, 1
```

8.3.1　异步中断

另一种类型的中断是异步中断（asynchronous interrupt），在执行期间，它的发生时间不可预测。异步中断的两个常见原因是

- 超时
- I/O 完成

为了了解超时如何引发异步中断，考虑一个多用户系统，它允许多个用户同时访问计算机。因为大多数计算机仅有一个 CPU，所以操作系统必须轮流把 CPU 分配给每个用户的作业，使用一种称为分时（time sharing）的技术。操作系统给用户作业分配称为时间片（time slice）的时间量，通常大约是 100ms（1/10 秒）。如果用户作业在这个时间内没有完成（这种情况称为超时），那么操作系统暂时停止这个作业，并给下一个作业分配另一个 CPU 时间量。

要实现分时，硬件必须提供一个闹钟，这样操作系统可以设置成在每个时段产生一个中断。

这样的中断是不可预测的原因是它取决于系统服务用户的请求有多忙。如果没有其他的作业等待使用 CPU，那么系统可以让你的作业运行得比标准时间片更长一些。如果另一个用户突然请求服务，那么操作系统会立刻暂停你的进程，并给正在请求的作业分配 CPU。那么此时该进程的中断时间点就不同于一个时间片超时时的时间点。

420

异步中断第二个常见的源是 I/O 完成。I/O 设备的一个基本属性是相比于 CPU 处理速度它们的速度很慢。如果一个正在运行的进程请求从键盘输入，用户的响应以秒计，而此期间

CPU 可以执行另一个进程的几十万条指令。即使进程从磁盘文件请求输入，也比从键盘输入快得多，但是在等待来自磁盘的信息时 CPU 仍然可以执行数千条指令。

为了不浪费 CPU 时间，如果看上去进程需要等待 I/O 完成，那么操作系统可以暂停请求 I/O 的进程。操作系统可以暂时把 CPU 分配给第二个进程，当知道当 I/O 完成时，第一个进程可以立即重新获得 CPU。因为第二个进程不可能预测 I/O 设备何时将完成第一个进程的 I/O 操作，因此它不知道操作系统何时会中断它并把 CPU 交回给第一个进程。

可以在进程间切换保持 CPU 繁忙的操作系统叫作多道程序设计系统（multiprogramming system）。要实现多道程序设计，硬件必须向 I/O 设备提供连线，当它们完成输入/输出操作时可以给 CPU 发送中断信号。

8.3.2 操作系统中的进程

操作系统的一个目的是高效分配系统资源。多道程序分时系统给系统中的作业分配 CPU 时间，目的是保持 CPU 尽可能地执行用户的作业而不是空闲等待 I/O。操作系统尽量公平地调度 CPU 时间，使得所有的作业都可以在合理的时间内完成。

在任何给定的时刻，操作系统都必须维护许多被暂停、正在等待轮到它们使用 CPU 时间的进程。它通过给每个进程分配一个单独的 PCB 来维护所有这些进程，这个 PCB 类似于 Pep/8 系统中断处理程序维护的 PCB。常见的做法是用链接表中的指针把 PCB 连接在一起，称为队列（queue）。图 8-18 展示的就是一个 PCB 队列。

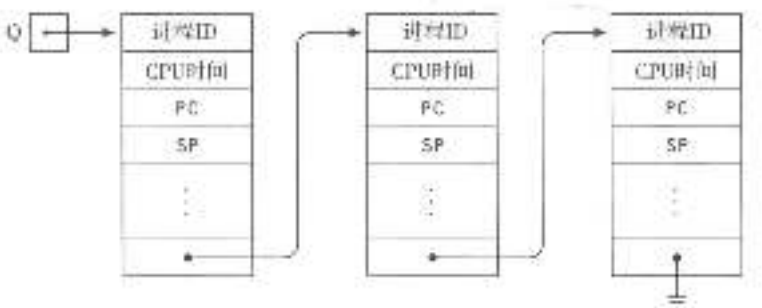


图 8-18 进程控制块的队列

每个 PCB 包含进程在最近一次中断时所有 CPU 寄存器值的副本。寄存器组必须包含程序计数器的副本，这样进程就可以从中断发生的位置继续执行。 421

PCB 包含一些可以帮助操作系统调度 CPU 的信息。一个例子是系统分配的唯一的进程标识号，即图 8-18 中的进程 ID，它起到标识进程的作用。假设一个用户想在一个进程正常执行完之前终止它，他知道 ID 号是 782，他可以发布 KILL(782) 命令，让操作系统搜索 PCB 队列，找出 ID 是 782 的 PCB，将它从队列中删除，并释放它。

另一个存储在 PCB 中信息的例子是暂停的进程截至目前使用的 CPU 时间总量。如果 CPU 变为可用，操作系统必须决定暂停的进程中哪一个将得到 CPU，它能用记录的时间做出一个公平的决定。

作业在系统中执行直至完成，经过多个状态，如图 8-19 所示。这个图是以状态转移图的形式呈现的，是有限状态机的又一个例子。每个转移都标明了导致状态改变的事件。

当用户提交了一个要处理的作业时，操作系统生成一个进程，即为它分配一个新的 PCB，并把它加入等待 CPU 时间的进程队列。它把程序装载到主存，并把 PCB 中 PC 的副

本设置为进程的第一条指令的地址。这样作业就在准备好（ready）状态中了。



图 8-19　操作系统中作业的状态转换图

操作系统最终会让该作业得到一些处理时间。在时间片之后设置闹钟来产生中断，把寄存器副本从 PCB 放入到 CPU。这样作业就在运行状态中了。

在运行状态中时，可能会发生 3 类事件：1）当闹钟中断时，如果运行的进程仍然在执行，就会超时。如果这样，操作系统就把这个进程的 PCB 放入准备好队列中，进程也就再次进入准备好状态。2）进程可能正常完成它的执行，这种情况下，它执行的最后一条指令是 SVC，请求操作系统终止这个进程。3）进程可能需要某种输入，这种情况下它执行 SVC 进行请求，操作系统会把这个请求转给适当的 I/O 设备，并把 PCB 放入另一个进程队列中，即等
422 待 I/O 操作完成的队列。这样进程就在等待 I/O 状态中了。

进程在等待 I/O 状态中时，I/O 设备最终会用所请求的输入中断系统。此时系统把输入放在内存的缓存中，把进程的 PCB 从等待 I/O 队列删除，并放入准备好队列。这样进程就在准备好状态中了，在这个队列中进程最后将得到更多的 CPU 时间。那时进程就可以访问缓存中的输入了。

8.3.3　多处理

对用户而言，作业只是从开始执行到结束。中断对用户而言是不可见的，就像 DECI 中断对 Asmb5 层的汇编语言程序员不可见一样。操作系统层的细节对更高抽象层的用户是不可见的。

用户唯一可以感受到的不同是，如果系统中有很多作业，那么执行该程序将会花更长的时间。提高速度的一种方法是给系统配备不止一个的 CPU，这样的配置叫作多处理系统（multiprocessing system）。图 8-20 展示了一个有两个处理器的多处理系统。

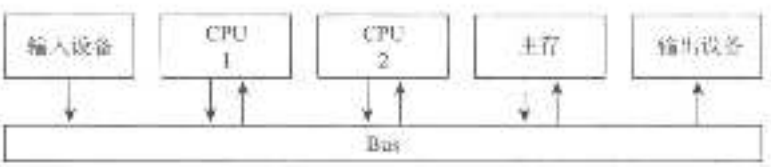


图 8-20　多处理系统框图

在多处理中，因为 CPU 在进程间的切换非常迅速，所以看上去就像它们在并发地执行一样。操作系统在多处理中可以调度不止一个进程来执行，因为有多个处理器。

如果性能的提高能够正比于系统中处理器数量的增加，那就好了。遗憾的是，通常实际情况并不是这样。当给系统增加更多的处理器时，也对系统的通信链路增加了更多的要求。例如，如果把处理器连接到图 8-20 所示的公共总线上，那么总线可能会限制系统的性能。如果两个 CPU 同时请求从输入设备读入，那么其中一个将不得不等待。增加越多的 CPU，就会越频繁地发生这样的冲突。

多处理器系统中固有的通信开销通常会导致图 8-21 所示的性能曲线。虚线说明理论上增加处理器能产生的最大好处，即性能翻倍。实际上性能不会提升那么多。

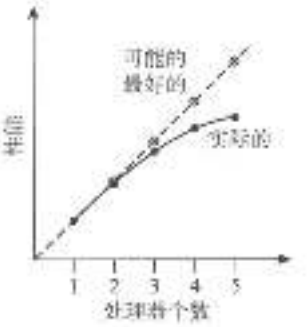


图 8-21 多处理系统中增加处理器导致性能提升

8.3.4 并发处理程序

到目前为止，我们考虑的所有进程都是相互独立的，每个进程属于不同的用户，并且进程之间没有交互。在这种情况下，计算结果不受中断发生时间的影响，中断唯一的影响是增加进程执行的总时间。

实际上，操作系统管理的进程通常需要和其他进程合作来执行它们的任务。图 8-22 中的程序描述了两个必须合作以避免产生错误结果的进程的情况。

假设操作系统要管理一个航空线路的数据库，这个数据库的记录会同时被多个用户访问。每个航班在数据库中有一条记录，除了其他一些信息之外，该记录还包括这个航班已经被预订出去的座位数量。分布在城市中的多家旅行社代理代表可能的用户访问该系统。因为不可能预测某个旅行代理何时需要访问该系统，所以对数据库信息的请求从某种程度上说是随机的。

某天，两个不同代理的客户正好同时想预订相同的航班。操作系统给每个作业创建一个进程，叫作 P1 和 P2，图 8-22 展示了每个进程的代码段。numRes 代表预定出去的数量，当 P1 和 P2 在系统里执行时，它是一个整型变量，值放在内存中。

C++ 层

进程 P1	进程 P2
...	...
numRes++	numRes++
...	...

汇编层

进程 P1	进程 P2
...	...
LDA numRes,d	LDA numRes,d
ADDA 1,i	ADDA 1,i
STA numRes,d	STA numRes,d
...	...

图 8-22 两个抽象层上的并发进程

423 / 424

假设两家代理在给他们的客户预订之前 numRes 的值是 47，交易完成后，numRes 应该是 49。在 C++ 层，每个进程想用赋值语句 numRes++ 把 numRes 增加 1。如果赋值语句是原子的（atomic），即不能分割的，那么不管哪个进程先执行赋值语句，C++ 层的代码段都将会产生正确的结果。如果 P1 先执行，它将使 numRes 变为 48，P2 将使 numRes 变为 49；如果 P2 先执行，它将使 numRes 变为 48，P1 将使 numRes 变为 49。不管哪种情况，都会得到正确的值 49。

问题是，C++ 层的赋值语句不是原子的，它们被编译成 LDA、ADDA 和 STA，在一个任何汇编语言语句之间都有可能发生中断的系统中执行。图 8-23 展示了一个执行序列的历史记录，可以看到什么会出错。A（P1）是 P1 累加器的内容，当 P1 运行时它在 CPU 中，当 P1 暂停时在 PCB 中。A（P2）是 P2 的累加器。

在这个序列中，P1 执行 LDA，将 47 放入累加器，接着是 ADDA，将累加器增加到 48，然后操作系统中断 P1 将处理器时间给 P2。P2 执行它所有 3 条语句，将内存中的 numRes 改为 48。当 P1 终于又继续执行时，它也将 numRes 设为 48。尽管每个进程都执行了它所有的语句，但最终结果是 numRes 为 48 而不是 49。

执行的语句	A(P1)	A(P2)	numRes
	?	?	47
(P1) LDA numRes,d	47	?	47
(P1) ADDA 1,i	48	?	47
(P2) LDA numRes,d	48	47	47
(P2) ADDA 1,i	48	48	47
(P2) STA numRes,d	48	48	48
(P1) STA numRes,d	48	48	48

图 8-23　图 8-22 序列的一种可能的执行历史记录

不管进程是在真正的并发多处理系统中，还是在模拟并发的多道程序系统中执行，都会发生这种问题。在多处理系统中，P1 和 P2 有可能正好同时执行 ADDA 语句，但是如果它们想同时执行 STA 语句，当一个进程在往内存写入值时，硬件会强制另一个进程等待。从逻辑
425 辑的角度来看，不管并发是真的还是假的，这种问题都会发生。

8.3.5　临界区

问题的根本原因是 P1 和 P2 共享部分主存，这部分包含 numRes 的值。无论何时并发进程共享一个变量，总是有可能的结果取决于中断发生的时间。要解决这个问题，我们就需要有一种方法能确保当一个进程访问共享变量时，另一个进程不能进行访问，要一直等到第一个进程结束访问后才可以。

两个进程中互斥的代码叫作临界区（critical section）。为了并发程序能够正确执行，软件必须保证如果一个进程正在执行它临界区中的语句，那么另一个进程不能执行它临界区中的语句。为了解决图 8-22 中的问题，我们需要找到一种方法，把赋值语句放到临界区中，这样在汇编层就不会发生交错执行了。

临界区需要两段额外的代码，叫作入口段（entry section）和出口段（exit section）。P1 的入口段正好在它的临界区前面，它的功能是测试 P2 是否正在执行它的临界区，如果是，就推迟 P1 临界区的执行直到 P2 完成执行它的临界区。P1 的出口段正好在它的临界区后面，它的功能是告知 P2，P1 已经不在临界区，这样 P2 就可以进入它的临界区了。

图 8-22 中每个进程的 C++ 层代码都必须按照如下进行修改：

```
其余部分（remainder section）
入口段（entry section）
numRes++   //临界区（critical section）
出口段（exit section）
其余部分（remainder section）
```

其余部分是代码中可以和其他进程一起并发执行且不会有错误影响的部分，而临界区是代码中必须互斥的部分。

下面的一些程序展示了我们尝试实现入口段和出口段，它们保护进程对临界区的访问。每个程序假设 P1 和 P2 是图 8-24

```
进程 P1                    进程 P2
do                         do
    entry section              entry section
    critical section           critical section
    exit section               exit section
    remainder section          remainder section
while (! done1);           while (! done2);
```

图 8-24　临界区程序的通用格式

那样的通用格式。done1 和 done2 是在其余部分某处被修改的（非共享的）局部布尔变量。

8.3.6 第一次尝试实现互斥

图 8-25 的程序是我们第一次尝试设计入口段和出口段，它使用 turn 这个共享整型变量。入口段由 do 循环组成，它检测 turn 的值；出口段由赋值语句组成，它修改 turn 的值。尽管这里的代码没有给出来，但是假设在进程进入 do 循环前 turn 被初始化为 1 或 2。

```
进程 P1                          进程 P2
do                               do
   while (turn != 1)                while (turn != 2)
      ; //nothing                      ; //nothing
   critical section                 critical section
   turn = 2;                        turn = 1;
   remainder section                remainder section
while (! done1);                 while (! done2);
```

图 8-25 尝试编程实现互斥

入口段中的 do 循环体是一个空 C++ 语句，它在汇编层不会生成代码。P1 的入口段代码被翻译成

```
loop: LDA    turn,d
      CMPA   1,i
      BRNE   loop
```

426 ; 427

假定 turn 被初始化为 1，两个进程同时尝试进入它们的临界区。不管怎样交错执行入口段中的汇编语句，P2 会持续循环，直到 P1 进入它的临界区。当 P1 完成它的临界区，它的出口段将把 turn 设置为 2，这之后 P2 就能够进入它的临界区了。

这个算法保证临界区是互斥的。只有当 turn 是 2 时，P2 才能在它的临界区中，在此期间 P1 不能在它的临界区中，反之亦然。当 P2 离开它的临界区后，将 turn 置为 1，这是 P1 可以进入临界区的信号。

尽管这个算法保证了互斥，但是它也要求进程严格交替执行它们的 do 循环，这可不太好。进程通过共享变量 turn 进行通信，它一直记录着谁该执行它的临界区。如果用户想让 P1 执行数次 do 循环而 P2 根本不执行，而使用这样的入口段和出口段，这样的情况是绝对不会发生的。

8.3.7 第二次尝试实现互斥

为了使一个进程能够执行它的 do 循环而不受另一个进程执行的限制（除去为了满足互斥的要求），图 8-26 的程序使用了两个共享布尔变量 enter1 和 enter2，假设 enter1 和 enter2 都初始化为 false。

如果 P2 在它的其余部分，enter2 一定为假，那么 P1 就可以想执行它的 do 循环多少次就执行多少次。它只用将 enter1 设置为 true，在 while 循环中测试 enter2 一次，执行它的临界区，将 enter1 设置为 false，执行它的其余部分。它可以想怎么重复这个循环就怎么重复。类似地，如果 P1 在它的其余部分，P2 可以重复它的这个循环。

```
进程 P1                          进程 P2
do                               do
   enter1 = TRUE;                   enter2 = TRUE;
   while (enter2)                   while (enter1)
      ; //nothing                      ; //nothing
   critical section                 critical section
   enter1 = FALSE;                  enter2 = FALSE;
   remainder section                remainder section
while (! done1);                 while (! done2);
```

图 8-26 编程实现互斥的另一次尝试

这样的实现保证了互斥。当 P1 把 enter1 设置为 true 时，就是给 P2 发信号告诉它正在尝试进入临界区。如果 P2 正好

在稍早一点的时候在它的 while 测试中用

428

```
LEA enter1,d
```

获取了 enter1，那么 P2 不会立即知道 P1 的目的，P2 可能已经在执行它的临界区了。不管怎样，如果 P2 在它的临界区中，那么 enter2 必定为 true，P1 的 while 循环将阻止 P1 同时也进入它的临界区。当 P2 最终退出时，它把 enter2 设置为假，这样就允许 P1 进入它的临界区。

协同执行进程的设计者面临的问题可能是非常微妙和意想不到的，这个算法就是一个这样的例子。尽管它像前面的程序一样，保证互斥而且不限制 do 循环的执行，但是它仍然有一个严重的漏洞。

图 8-27 展示了一个执行序列，P1 设置 enter1 为 true，然后遇到中断。P2 设置 enter2 为 ture，然后开始执行它的 while 循环。因为 enter1 为 true，因此 while 循环将继续执行一直到 P2 超时，P1 恢复。因为 enter2 为 true，所以 P1 也将会无限地循环。

执行的语句	enter1	enter2
	false	false
(P1) enter1=TRUE;	true	false
(P2) enter2=TRUE;	true	true
(P2) while(enter1)	true	true
(P1) while(enter2)	true	true

图 8-27　图 8-26 所示程序的一个会产生死锁的执行序列

P1 和 P2 两个进程都处于想进入临界区的状态。P1 要等到 P2 进入执行它的临界区将 enter2 设置为 false 才能进入，但是 P2 要等到 P1 进入执行它的临界区将 enter1 设置为 false 才能进入。两个进程都在等待永远不会出现的情况，这种局面叫作死锁（deadlock）。死锁就像死循环，要避免出现这种情况。

8.3.8　Peterson 互斥算法

我们需要一种解决方法，它保证互斥，允许每个进程外面的 do 循环无限制地执行，并且避免死锁的出现。图 8-28 是 Peterson 算法的实现，它结合图 8-25 和图 8-26 的特性来实现所有的目标，其基本思路是 enter1 和 enter2 提供如图 8-26 那样的互斥，即使在同一时间两个进程都想进入临界区，但是 turn 只允许一个进入。enter1 和 enter2 初始为 false，turn 初始为 1 或者 2。

```
进程 P1                                进程 P2
do                                     do
   enter1 = TRUE;                         enter2 = TRUE;
   turn = 2;                              turn = 1;
   while (enter2 && (turn == 2))          while (enter1 && (turn == 1));
      ; //nothing                            ; //nothing
   critical section                       critical section
   enter1 = FALSE;                        enter2 = FALSE;
   remainder section                      remainder section
while (! done1);                       while (! done2);
```

图 8-28　Peterson 互斥算法

来看一看互斥是如何保证的。考虑如果P1和P2同时执行它们的临界区的情况，enter1和enter2都将为ture。在P1中，while测试意味着turn的值为1。因为enter2为true。但在P2中，while测试意味着turn的值为2，因为enter1为true。这个矛盾表明P1和P2不可能同时执行它们的临界区。

但如果P1和P2同时想进入它们的临界区会怎样呢？在入口段有一些交错执行会导致它们同时执行临界区吗？答案是不会，即使有AND运算的while测试在汇编层不是原子的。有两个条件可以使P1通过while测试进入它的临界区：enter2为false或者turn为1。如果任一条件满足，不管另一个条件如何，P1都可以进入临界区。

假设P1通过while测试，因为它用

```
LDA enter2,0
```

获得enter2的值，enter2的值为false。只有P2在它的其余部分中时，这才会发生。即使P1在装载enter2的值后被中断，然后P2设置enter2为true，turn为1，P2也不能进入它的临界区，因为P1已经设置enter1为true且turn现在为1。

假设P1通过while测试，因为它用

```
LDA turn,0
```

获得turn的值，turn的值为1。因为P1前面的指令把turn设置为2，只有P1在它的前一条指令和while测试之间被中断，且P2将turn设为1时，才可能出现这种情况。但是接着，P2又将被阻止通过它的while循环进入临界区，因为P1已经设置enter1为真且turn现在的值为1。

429
~
430

来看一看为什么不会发生死锁。假设两个进程都陷入死锁，(在多处理系统中)并行或(在多道程序系统中)在不同的时间片中执行它们的while循环。P1中的while测试意味着turn的值必须为2，但是P2中while测试意味着turn的值必须为1。这个矛盾的情况表明两个进程不可能同时都在循环。

假设两个进程同时都想用

```
STA turn,0
```

设置turn。在多道程序设计系统中，因为必须在不同的时间片中执行，所以P1给turn的赋值一定会发生在P2的赋值之前或者之后。在多处理系统中，如果两个进程正好同时都想在主存中给turn赋值，那么当其中一个进程执行STA时硬件会强制另一个进程等待。在两种系统中，先给turn赋值的进程将进入它的临界区，这样就不会发生死锁。

8.3.9 信号量

尽管图8-28中的程序解决了临界区问题并避免出现死锁，但是它的缺点是效率低。阻止进程进入临界区的是while循环，循环的唯一目的是拖延进程直到它被中断，给另一个进程时间去完成执行它的临界区。因为进程以循环的方式被锁在自己的临界区之外，所以这样的循环叫作**旋转锁**(spin lock)。

旋转锁是对CPU时间的浪费，尤其是如果进程在多道程序设计系统中执行，并被分配一个新的时间片。如果把这个CPU时间分配给另一个进程执行一些有用的工作，那就会更有效率。**信号量**(semaphore)是大多数操作系统都提供的用以并发编程的共享变量，它们可以让程序员不用旋转锁来实施临界区。

信号量是一个整数型变量。它的值只能由操作系统调用来修改。对信号量s的3个操作是

- init(s)
- wait(s)
- signal(s)

这里 init、wait 和 signal 是操作系统提供的过程。在汇编层，用带有适当操作数指示符的 SVC 来调用这些过程。信号量是带操作的抽象数据类型（ADT）的又一个例子，它的含义
431 程序员是知道的，但它的实现隐藏在更低的抽象层中。（wait(s) 和 signal(s) 通常又分别写作 p(s) 和 v(s)。）

每个信号量 s 有一个关联的进程控制块队列，叫作 sQueue，它代表被挂起的进程。这几个操作的含义是

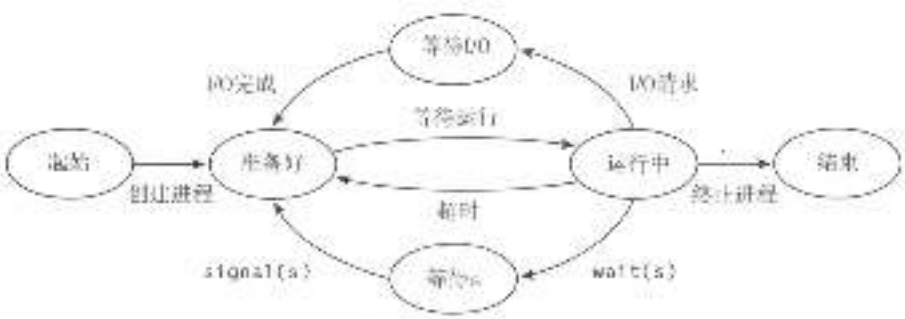
```
init(s)
s = i;
sQueue = 一个进程控制块的空队表

wait(s)
s--
if (s < 0)
    把此次进程，把它添加到 sQueue 中

signal(s)
s++
if (s <= 0)
    从 sQueue 取出一个进程加入准备好队列中
```

每个操作都有一个重要的特性，那就是操作系统保证它们是原子的。例如，不可能两个进程同时执行 signal(s)，而 s 只增加了 1，像图 8-22 中的 numRes 那样。汇编层的赋值语句绝不会交叉执行。

图 8-29 是操作系统中提供信号量的作业的状态转移图。与在准备好状态的进程是暂停的，它的 PCB 是在准备好队列中一样，在 waiting-for-s 状态的进程是暂停的，它的 PCB 在 sQueue 中。这样的进程被阻止运行，因为它必须转移到准备好状态才能执行。



432 图 8-29　操作系统中提供信号量的作业的状态转换图

如果一个正在运行的进程执行 wait(s)，当 s 大于 0 时，wait(s) 只是将 s 减 1，进程继续执行；当 s 小于或等于 0 时，正在运行的进程执行 wait(s) 会转移到等待 s 状态。如果当 s 大于或等于 0 时，一个正在运行的进程执行 signal(s)，它只是将 s 加 1，进程继续执行；当 s 小于 0 时，正在执行 signal(s) 的进程会引发某个正在等待 s 的进程被操作系统挑选出来放到准备好状态中。执行 signal(s) 的进程继续运行。

从 wait 和 signal 的定义来说，s 为负值表示一个或多个进程被阻塞在 sQueue 中，而 s 的大小就是被阻塞进程的数量。例如，如果 s 值为 -3，那么就有 3 个进程阻塞在 sQueue 中。

signal(s) 执行时如果有多于一个进程被阻塞，那么操作系统尽力在选择进程转移到准备好状态这件事情上保持公平。通用的策略是使用先进先出（First In, First Out, FIFO）的调度策略，这样阻塞时间最长的进程会被送到准备好状态。FIFO 是队列区别于栈的特性，栈是后进先出（Last In, First Out, LIFO）。

图 8-29 仅展示了一个信号量等待状态。在提供信号量的系统中，程序员可以想声明多少个就声明多少个不同的信号量，操作系统会为每个信号量维护一个阻塞进程队列。

8.3.10 带信号量的临界区

如果操作系统提供信号量，那对程序来说临界区就很容易了。图 8-30 的程序假设 mutEx 是一个用 init(mutEx) 初始化为 1 的信号量。

第一个执行 wait(mutEx) 的进程将把 mutEx 从 1 变为 0，并进入它的临界区。如果其他进程在此期间执行 wait(mutEx)，那么将把 mutEx 由 0 变为 -1，操作系统将立即阻塞它。当第一个进程最终离开它的临界区时，将执行 signal(mutEx)，这将把另一个进程放入准备好状态中。

```
进程 P1                          进程 P2
do                               do
   wait(mutEx);                     wait(mutEx);
   critical section                 critical section
   signal(mutEx);                   signal(mutEx);
   remainder section                remainder section
while (! done1);                 while (! done2);
```

图 8-30 使用信号量的临界区

433

由于操作系统保证 wait 和 signal 是原子的，所以程序员不需担心在入口段和出口段中的操作交错执行。因为系统会立即把第二个进程放在 mutEx 的等待队列中，因此也不会在旋转锁上浪费时间。当然，隐藏细节并不是消除细节。要提供信号量，操作系统设计者必须利用硬件的特性，还要使用和前面程序中一样的算法推理思路。信号量可以满足操作系统的两个目标——向高级编程提供方便的环境；高效地分配系统资源。

Fernando J. Corbató

1926 年 7 月 1 日 Fernando J. Corbató 生于加利福尼亚州奥克兰。1950 年，他从 MIT 获得硕士学位，1956 年他从 MIT 获得物理学博士学位。他最著名的工作是两个开创性的操作系统——兼容分时系统（Compatible Time-Sharing System，CTSS）和多工信息和计算服务（Multiplexed Information and Computing Service，Multics）。

在操作系统出现以前，一个机构中的计算机用户会请求分配一天中的某段时间，物理上去操作这个计算机，在这期间，其他的用户不能访问这台计算机。最早且基础的操作系统使用批处理，有一个人接收各个用户的作业，也就是一堆打孔的卡，然后代替这些用户去操作计算机。调试程序是很费时的，程序员在每次编译完成到获得程序报出的错误之间，有时需要等待几个小时。

在 20 世纪 50 年代末，许多科学家，包括 MIT 的 John McCarthy 和牛津大学的 Christopher Strachey，开始提出分时的概念，允许多个用户同时通过数字调制解调器和打字控制台远程登录到计算机上。看上去好像每个用户都拥有对计算机的完整访问，这样在统调试极大地降低了软件开发的时间。

Corbató 在 MIT 领导小组，研制出了 CTSS，最早的分时系统之一。它是为 IBM 709

和7090设计的，能够同时支持4个用户——3个使用打字控制台，一个被动用户。内存是32K个36位的字，用户作业占用高地址的27K，分时管理程序占用低地址的5K。这个小组修改硬件，使之能够提供用户作业之间的内存保护。

Multics系统是一个更有雄心的项目，吸取了CTSS项目的经验教训。它的两个突出特点就是使用了PL/I来作为实现语言和内存保护的环机制。在Multics之前，大多数的操作系统是用汇编语言来编写的。就在Multics项目开始的时候，IBM刚刚推出了Program-ming Language One（PL/I）的规范。Corbató的小组为该语言开发出了自己的编译器，因为当时IBM还没能提供一个商用的编译器。



有些讽刺的是，现在的许多个人计算机操作系统很容易受到病毒的侵袭，而Multics是完全免疫的。目前为止最常见的病毒机制就是缓冲区溢出，也就是程序试图访问数组超出界限的地方。今天的大多数操作系统是用C语言编写的，它会允许这样的访问，而PL/I不允许，这也是为什么Multics的安全性这么高的原因。

因为其在CTSS和Multics项目上的开创性的领导工作，Fernando Corbató在1990年获得图灵奖。他现在是MIT电子工程与计算机科学系的名誉退休教授。

8.4　死锁

图8-26中的程序展示了并发处理如何在共享主存变量的两个进程之间产生死锁。当进程共享其他资源时也可能会发生死锁。操作系统管理的资源包括打印机、光盘（CD）驱动器和磁盘文件。并发进程共享这些资源中的任意之一都有可能导致死锁。

举一个共享这些资源导致死锁的例子。假设进程P1请求计算机唯一的一个CD驱动器进行数据输入。操作系统把CD驱动器分配给P1，P1会一直占用它，直到不再需要它。P2可能要从磁盘文件请求输入。操作系统打开这个文件并分配给这个进程。

现在假设P1需要从P2正在访问的磁盘文件读入，但因为P2已经打开了这个文件，所以操作系统不会同意这个请求，要一直等到此文件可用才可以。如果P2请求CD驱动器，操作系统也会阻止它直到P1释放该驱动器。

在这种情况下，进程就处于死锁的状态。P1不能前进直到P2释放磁盘文件为止，P2不能前进直到P1释放CD驱动器为止。两个进程都在等待一个不可能发生的事件，将被操作系统挂起。

8.4.1　资源分配图

为了有效地管理资源，操作系统需要一种方式能够发现可能的死锁。它采用了一种称为资源分配图（resource allocation graph）的结构。资源分配图是系统中进程和资源的图形化描述，展示了哪个资源分配给了哪个进程，哪个进程阻塞在了对哪个资源的请求上。

图8-31中的资源分配图就是P1和P2因为对CD驱动器和磁盘文件的请求形成了死锁。进程和资源是图中的结点，进程用圆圈表示，资源是方框里的实心圆点。有两种类型的边：分配边和请求边。

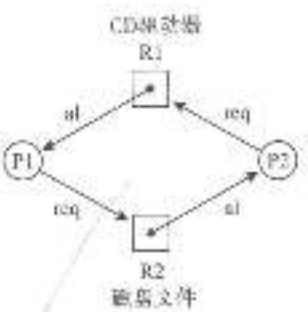


图8-31　一个具有死锁循环的资源分配图

分配边（allocation edge，al）是从资源到进程，表示该资源分配给了该进程。在图中，标记为 al 的从 CD 驱动器到 P1 的边的含义是操作系统把 CD 驱动器分配给了进程 P1。请求边（request edge，req）是从进程到资源，意思是进程被阻塞了，正在等待该资源。标记为 req 的从 P2 到 CD 驱动器的边的含义是 P2 被阻塞在了对 CD 驱动器的等待上。

434
·
435

边组成了一个闭合的路径，从 P1 到 P2 到 R1 再回到 P1，死锁就很明显地能看出来。图中这样的一个闭合回路称为循环（cycle）。资源分配图中的循环表明一个进程阻塞在一个资源上，因为该资源分配给了另一个进程，而它阻塞在另一个资源上，以此类推，最后一个资源分配给了第一个进程。如果循环不能被打破，就出现了死锁。

有时一类中的资源是不做区分的，进程可以请求某类资源中的一个，而不关心具体是哪一个，因为资源都是等价的。比如，有一组 CD 驱动器，或者一组同样的激光打印机。如果进程需要一个激光打印机而不在意是其中的哪一个，那么操作系统就可以把任意一个空闲的打印机分配给它。

资源分配图用长方形方框中的 n 个实心点表示同一类 n 个等价的资源。当操作系统分配一个该类资源时，分配边从代表一个资源的某个点发出。不过请求边指向方框，因为请求进程不在意它获得的是这类资源中的哪一个。

图 8-32a 中 R1 资源类中的两个资源都已经分配出去了，P1 有一个对该类资源未被满足的请求，它不在意获得哪一个资源。

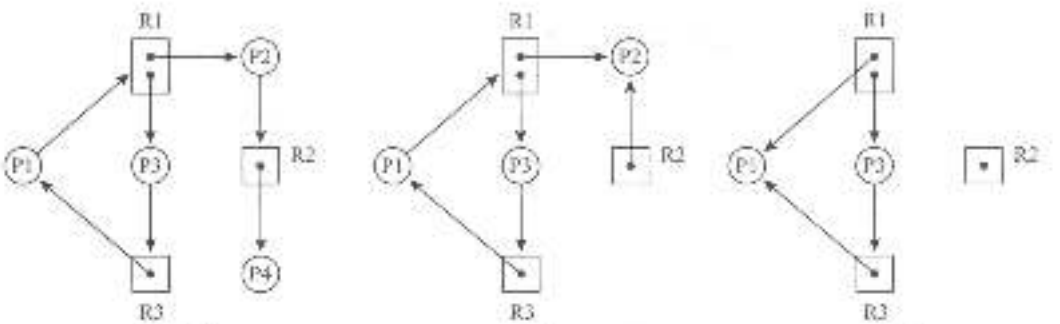


图 8-32 有循环但是没有死锁的资源分配图

显然这张图有循环（P1，R1，P3，R3，P1），但是没有死锁，因为这个循环是可以被打破的。资源分配图中任意一个进程如果没有请求边，那么它就没有被阻塞。这张图中 P4 是未被阻塞的，可以想象它最终会执行完毕，释放 R2。操作系统会满足 P2 对 R2 的请求，把从 P2 到 R2 的请求边改为从 R2 到 P2 的分配边，如图 8-32b 所示。

现在 P2 能够使用 R2 和 R1，可以运行完成，最终释放这些资源。当 P2 释放一个 R1 资源时，操作系统可以把该资源分配给 P1，得到图 8-32c。P1 拥有它需要的所有资源，可以运行完成，之后 P3 也可以完成了。所有的进程都可以完成，所以没有死锁。

在寻找循环的过程中，一定要考虑边的方向。你可能会把图 8-33 中的（R1，P1，R2，P2，R1）当作循环，但是它并不是。没有从 R2 到 P2 的边，也没有从 P2 到 R1 的边。循环是死锁的必

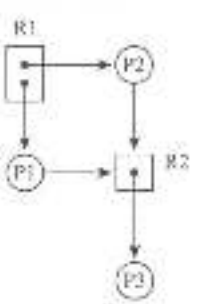


图 8-33 没有循环因此也没有死锁的资源分配图

436 要但是非允分条件。在这张图中，没有循环，所以也没有死锁。

8.4.2 死锁策略

操作系统可以采用下列3种通用策略中的一种来处理死锁：

- 预防
- 发现并恢复
- 无视

某个操作系统中可能有多种不同的策略。系统可以为一组资源使用一种策略，而为另外一组资源使用另一种策略。

预防策略是采用技术保证不会出现死锁。一种技术是要求运行完成所需的所有资源，在执行开始时作业请求和分配一次完成。如果P1同时获得R1和R2，那么图8-31中的死锁是不会产生的。只有一个进程被分配了某个资源，然后又请求另外的资源，形成一个循环，才可能发生死锁循环。

发现并恢复策略允许发生死锁。采用这种策略时，操作系统将周期性地执行一个检测系统中死锁循环的程序。如果操作系统发现死锁，就拿走循环中一个进程所占用的一个资源。因为进程可能已经执行了一部分，所以操作系统通常必须终止该进程，除非能后当该进程再获得该资源时，资源的状态还能重建。

所有的这些策略都是有代价的。预防策略对用户有所限制，特别是，如果进程所需资源是依赖于输入无法事先知道的时候。在发现并恢复策略中，发现和恢复算法需要占用CPU时间，这些时间就不能用在用户作业上。

第三种策略就是无视死锁。如果认为其他策略的开销都太大，而发生死锁的概率又很小，或者出现死锁的后果并不严重，选用这种策略就比较有效率。例如，某个分时系统会定期关机进行例行维护，每次关机时所有的作业都会被清除，包括那些死锁的作业。

总结

操作系统的目标是向高级编程提供方便的环境并高效地分配系统资源。操作系统的一项重要功能是管理用户提交要执行的作业。装载器是操作系统的一部分，它把作业放入内存进行执行。在作业完成执行之后，它把CPU的控制返回给操作系统，操作系统可以继续加载其他的应用程序。

陷阱处理程序为作业执行一些处理操作，向程序员隐藏低层的细节。当陷阱发生时，正在运行的作业的进程控制块（PCB）被存储起来，同时操作系统响应并服务该中断。PCB由
437 进程的状态组成，包括程序计数器、状态位和所有CPU寄存器的内容的副本。要继续执行该作业时，操作系统把PCB放回CPU中。异步中断的运行和过程调用类似，只不过中断是由操作系统发起，而不是由应用程序员的代码发起。

进程是执行时的程序。在多道程序设计系统中，CPU在多个进程间切换。在多处理系统中，有不止一个CPU。多道程序设计和多处理系统中都有并发执行的进程。并发地执行协作的进程，操作系统必须能够保证对临界区的互斥访问，并避免死锁。Peterson算法满足这两个要求。信号量是操作系统提供的整数型变量，对它的操作包括wait和signal，这些操作都是原子的或者说是不可分割的。信号量可以用来满足互斥和不出现死锁。

当进程共享操作系统提供的资源时，也有可能出现死锁。资源分配图由代表资源和进

程的结点组成，结点间的边表示资源分配和请求。如果资源分配图中包含有不能被打破的循环，就表示发生了死锁。

练习

8.1 节

1. 操作系统的两大目标是什么？
2. 图 8-3 中的装载器执行下列输入：

```
04 00 05 00 00 31 00 03 39 00 03 50 00 0A 41 00
12 00 54 68 61 74 27 73 20 61 6C 6C 2E 04 00 zz
```

假设从 FC5D 到 FC97 的循环执行到第 27 次，以 4 个十六进制数字的格式说明下列寄存器中的值：

* (a) 在 F6C0 的 LDA 之后的 A
* (b) 在 FC72 的 ASLA 之前的 A
* (c) 在 FC75 的 ASLA 之后的 A
(d) 在 FC7C 的 LDA 之后的 A
(e) 在 FC88 的 ANDA 之后的 A
(f) 在 FC8B 的 ORA 之后的 A
(g) 在 FC91 的 ADDX 之后的 X

3. 对于循环第 29 次执行时的情况，再做一道练习 2。

8.2 节

4. 下面的程序执行时，产生一个 DECI 中断：

```
0000  040005          BR      main        ;Branch around data
0003  0000    num:    .BLOCK  2           ;Global variable
              ;
0005  310003  main:   DECI    num,d       ;Input decimal value
0008  390003          DECO    num,d       ;Output decimal value
000B  50000A          CHARO   '\n',i
000E  410012          STRO    msg,d       ;Output message
0011  00              STOP
0012  546865  msg:    .ASCII  "That's all.\n\x00"
      742773
      20616C
      6C2E0A
      00
001F                  .END
```

根据图 8-6，进入和退出该陷阱处理程序，以 4 个十六进制数字的格式说明下列寄存器中的值：

* (a) 在 FC9D 的 LDBYTEX 之后的 X
* (b) 在 FCB9 的 ASRX 之后的 X
(c) 在 FCBA 的 SUBX 之后的 X
(d) 在 FCBE 的 CALL 之后的 PC
(e) 在 FCC1 的 RETTR 之后的 PC

*5. 对于 DECO 指令再做一道练习 4。

*6. 对于 STRO 指令再做一道练习 4。

7. 以输入 37 运行练习 4 中的程序。根据图 8-14，DECI 陷阱处理程序，以 4 个十六进制数字的格式说明 (a)–(h) 中寄存器的值，并回答 (i) 中的问题：

* (a) 在第一次执行 FDEB 的 ANDA 之后的 A。
* (b) 在第二次执行 FDEB 的 ANDA 之后的 A。
* (c) 在第一次执行 FDF4 的 LDX 之后的 X。
(d) 在第二次执行 FDF4 的 LDX 之后的 X。
(e) 在第一次执行 FDF8 的 BR 之后的 PC。
(f) 在第二次执行 FDF8 的 BR 之后的 PC。
(g) 在 FEE9 的 LDA 之后的 A。
(h) 在执行 FF04 的 STBYTEX 之前的 X，假设在陷阱发生之前进位位为 0。

(i) 在 FEC7 的 LDA 执行之前执行的是什么语句?

*8. 对于输入 -295 再做一遍练习 7。

9. 以输入 37 运行练习 4 中的程序。根据图 8-15，DECO 陷阱处理程序，以 4 个十六进制数字的格式说明下述寄存器的值：

*(a) 在 FF4A 的 LDA 之后的 A　　*(b) 在 FF57 的 STA 之后的 A

对于下面的问题，假设在 FF78 的 CALL 调用的是子例程 divide：

*(c) 在 FF91 的 LDA 之后的 A　　(d) 在 FFA6 的 CPX 之前的 X

(e) 在 FFBC 的 LDA 之后的 A

对于下面的问题，假设在 FF81 的 CALL 调用的是子例程 divide：

(f) 在 FF91 的 LDA 之后的 A　　(g) 在 FFA6 的 CPX 之前的 X

(h) 在 FFBC 的 ORX 之后的 X

10. 对于输入 -2068 再做一遍练习 9。

11. 运行练习 4 中的程序，执行 STRO 指令。根据图 8-16，STRO 陷阱处理程序，以 4 个十六进制数字的格式说明下述寄存器的值：

438 ↓ 439

(a) 在 FFD2 的 LDA 之后的 A　　(b) 在第一次执行 FFE8 的 LDBYTEA 之后的 A

(c) 在第一次执行 FFF1 的 ADDX 之后的 X　　(d) 在第 5 次执行 FFE8 的 LDBYTEA 之后的 A

(e) 在第 5 次执行 FFF1 的 ADDX 之后的 X

12. 执行图 5-11 中地址 0005 处采用直接寻址方式的 DECI 指令，产生一个陷阱，陷阱处理程序调用图 8-10 的 setAddr 例程。以 4 个十六进制数字的格式说明变址寄存器的值：

(a) 在 FD3D 的 LDX 之后　　(b) 在 FD40 的 SUBX 之后　　(c) 在 FD43 的 LDX 之后

13. 执行图 6-41 中地址 0048 处采用间接寻址方式的 DECO 指令，产生一个陷阱，陷阱处理程序调用图 8-10 的 setAddr 例程。以 4 个十六进制数字的格式说明变址寄存器的值：

(a) 在 FD4A 的 LDX 之后　　(b) 在 FD4D 的 SUBX 之后

(c) 在 FD50 的 LDX 之后　　(d) 在 FD53 的 LDX 之后

14. 执行图 6-4 中地址 0009 处采用栈相对寻址方式的 DECI 指令，产生一个陷阱，陷阱处理程序调用图 8-10 的 setAddr 例程。以 4 个十六进制数字的格式说明变址寄存器的值：

(a) 在 FD5A 的 LDX 之后　　(b) 在 FD5D 的 SUBX 之后

(c) 在 FD60 的 LDX 之后　　(d) 在 FD63 的 LDX 之后

15. 第二次执行图 6-36 中地址 0013 处采用栈变址寻址方式的 DECI 指令，产生一个陷阱，陷阱处理程序调用图 8-10 的 setAddr 例程。以 4 个十六进制数字的格式说明变址寄存器的值：

(a) 在 FD8D 的 LDX 之后　　(b) 在 FD90 的 SUBX 之后　　(c) 在 FD93 的 LDX 之后

(d) 在 FD96 的 ADDX 之后　　(e) 在 FD99 的 ADDX 之后

16. 第二次执行图 6-38 中地址 0016 处采用栈变址间接寻址方式的 DECI 指令，产生一个陷阱，陷阱处理程序调用图 8-10 的 setAddr 例程。以 4 个十六进制数字的格式说明变址寄存器的值：

(a) 在 FDA0 的 LDX 之后　　(b) 在 FDA3 的 SUBX 之后　　(c) 在 FDA6 的 LDX 之后

(d) 在 FDA9 的 ADDX 之后　　(e) 在 FDAC 的 LDX 之后　　(f) 在 FDAF 的 ADDX 之后

8.3 节

17. 图 8-23 中的交错执行序列可以简写为 112221，表示图 8-22 中 P1，P1，P2，P2，P2，P1 执行的语句。

(a) 有多少种可能的、不同的执行序列？用简写法列出每种可能的序列，对于每个序列，说明 numRes 是否有正确的值。

(b) 所有可能的序列中有百分之多少产生不正确争取的结果？

440 (c) 这个比例是否就是程序运行时可能得到不正确结果的概率？请解释。

18. 下面这段代码尝试实现临界区，类似于图 8-26 中的程序，除了入口段中语句的顺序不一样外：

```
进程 P1                          进程 P2
do                               do
   while (enter2)                   while (enter1)
      ; //nothing                      ; //nothing
   enter1 = TRUE                    enter2 = TRUE
   critical section                 critical section
   enter1 = FALSE;                  enter2 = FALSE;
   remainder section                remainder section
while (! done1);                 while (! done2);
```

*(a) 这个算法能保证互斥吗？如果不能，给出一个执行序列使得两个进程能同时进入它们的临界区。

(b) 该算法能预防死锁吗？如果不能，给出能导致 P1 和 P2 死锁的执行序列。

19. 根据 wait 和 signal 的定义，解释 s 的大小是被阻塞进程的数量。

20. I 表示执行 init(s)，W 表示 wait(s)，而 S 表示 signal(s)。例如，IWWS 表示操作系统中某些进程的调用序列为 init(s)、wait(s)、wait(s) 和 signal(s)。对于下面每个调用序列，说明 s 的值和序列中最后一个调用执行后被阻塞的进程数量：

*(a)IW　　(b)IS　　(c)ISSSW　　(d)IWWWS　　(e)ISWWWW

21. 假设 3 个并发的进程执行下面的代码：

```
进程 P1                  进程 P2                  进程 P3
do                       do                       do
   wait(mutEx);             wait(mutEx);             wait(mutEx);
   critical section         critical section         critical section
   signal(mutEx);           signal(mutEx);           signal(mutEx);
   remainder section        remainder section        remainder section
while (! done1);         while (! done2);         while (! done3);
```

解释该代码如何保证对这 3 个临界区的访问是互斥的。

22. 假设 s 和 t 是两个信号量，用 init(s) 和 init(t) 初始化。考虑下面两段并发进程代码：

```
进程 P1                  进程 P2
wait(s);                 wait(t);
wait(t);                 wait(s);
critical section         critical section
signal(s);               signal(t);
signal(t);               signal(s);
remainder section        remainder section
```

441

*(a) 该算法保证互斥吗？如果不能，给出一个执行序列使得两个进程能同时进入它们的临界区。

(b) 该算法能预防死锁吗？如果不能，给出能导致 P1 和 P2 死锁的执行序列。

23. 考虑下面两段并发进程代码：

```
进程 P1          进程 P2
语句 1           语句 4
语句 2           语句 5
语句 3           语句 6
```

修改这段代码保证语句 5 在语句 2 之前执行。用信号量实现。

24. 下面每段代码在入口段或出口段中包含一个漏洞。说明每段代码是否能保证互斥。如果不能，给出会违反互斥的执行序列。说明是否会出现死锁。如果会，给出相应的执行序列。

*(a)

```
进程 P1                          进程 P2
do                               do
   wait(mutEx);                     signal(mutEx);
   critical section                 critical section
   signal(mutEx);                   wait(mutEx);
   remainder section                remainder section
while (! done1);                 while (! done2);
```

(b)

进程 P1

```
do
    signal(mutEx);
    critical section
    wait(mutEx);
    remainder section
while (! done1);
```

进程 P2

```
do
    signal(mutEx);
    critical section
    wait(mutEx);
    remainder section
while (! done2);
```

(c)

进程 P1

```
do
    wait(mutEx);
    critical section
    signal(mutEx);
    remainder section
while (! done1);
```

进程 P2

```
do
    wait(mutEx);
    critical section
    wait(mutEx);
    remainder section
while (! done2);
```

442

(d)

进程 P1

```
do
    wait(mutEx);
    critical section
    signal (mutEx);
    remainder section
while (! done1);
```

进程 P2

```
do
    wait(mutEx);
    critical section
    remainder section
while (! done2);
```

(e)

进程 P1

```
do
    wait(mutEx);
    critical section
    signal(mutEx);
    remainder section
while (! done1);
```

进程 P2

```
do
    critical section
    signal(mutEx);
    remainder section
while (! done2);
```

8.4 节

25. 一个操作系统中有进程 P1、P2、P3 和 P4，资源 R1（一个）、R2（一个）、R3（2 个）和 R4（3 个）。（1，1）、（2，2）、（1，2）表示 P1 请求 R1，然后 P2 请求 R2，然后 P1 请求 R2。注意，前两个请求在资源分配图上生成两条分配边，但是第三个请求生成一条请求边，因为 R2 已经分配给了 P2。画出下面每个请求序列执行后的资源分配图，说明每个资源分配图是否包含循环。如果有，说明是否是死锁循环。

*(a) (1, 1), (2, 2), (1, 2), (2, 1)
*(b) (1, 4), (2, 4), (3, 4), (4, 4)
(c) (1, 1), (2, 1), (3, 1), (4, 1)
(d) (3, 3), (4, 3), (2, 2), (3, 2), (2, 3)
(e) (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4)
(f) (2, 1), (1, 2), (2, 3), (3, 3), (2, 2), (1, 3)
(g) (2, 1), (1, 2), (2, 3), (3, 3), (2, 2), (1, 3), (3, 1)
(h) (1, 4), (2, 3), (3, 3), (2, 1), (3, 4), (1, 3), (4, 4), (3, 1), (2, 4)
(i) (1, 4), (2, 3), (3, 3), (2, 1), (3, 4), (1, 3), (4, 4), (3, 1), (2, 4), (4, 3)

问题

8.2 节

26. 实现一条新的一元指令，替代 NOP0，称为 ASL2。它把累加器左移 2 位。V 位应该保持不变。N 位和 Z 位应该和累加器中的新值保持一致。C 位应该是第二次移位的进位值。写一个程序检测新指令

的所有特性。

27. 实现一条新的一元指令，代替 NOP，称为 ASLMANY。它的操作数是累加器左移的次数。允许立即数、直接和栈相对寻址。V 位应保持不变，N 位和 Z 位应该和累加器中的值保持一致，C 位应该是最后一次移位的进位值。写一个程序检测新指令的所有特性。 443

28. 实现一条新的一元指令，替代 NOP，称为 MULA。它把累加器乘以操作数，结果放入累加器中。允许立即数、直接和栈相对寻址。V 和 C 位应保持不变，N 位和 Z 位应该和累加器中的值保持一致。写一个程序检测新指令的所有特性。

29. 直接寻址或是立即数寻址再间接寻址，间接寻址就是直接寻址再间接寻址。把这个概念再进一步推论，两次间接寻址，也就是间接寻址再在间接寻址。实现一条新指令，替代 NOP0，其助记符为 STADI，代表 Store Accumulator Double Indirect（存储累加器，两次间接寻址）。它使用两次间接寻址来存储累加器。执行下面的程序来测试你的指令。

```
0000  C4000F          BR      main
0003  0000   num1:    .BLOCK  2           ;num1
0005  0003   num1ad:  .ADDRSS num1        ;Address of num1
0007  0005   num1ada:.ADDRSS num1ac       ;Address of address of num1
0009  0000   num2:    .BLOCK  2           ;num2
000B  0009   num2ad:  .ADDRSS num2        ;Address of num2
000D  000B   num2ada:.ADDRSS num2ac       ;Address of address of num2
000F  C00016 main:    LDA     27,i        ;Load accumulator
0012  24              STADI               ;Store num1 double indirect
0013  0007            .ADDRSS num1ada
0015  C00022          LDA     34,i        ;Load accumulator
0018  24              STADI               ;Store num2 double indirect
0019  000D            .ADDRSS num2ada
001B  390003          DECO    num1,d      ;Output num1
001E  500020          CHARO   ' ',i
0021  390009          DECO    num2,d      ;Output num2
0024  50000A          CHARO   '\n',i
0027  00              STOP
0028                  .END
```

对于汇编器和 CPU 来说，NOP0 是一条一元指令。但是你的程序必须把它实现为非一元指令，需要对保存的 PC 加 1 以跳过操作数指示符。

30. 实现一条新的一元指令，替代 NOP，称为 HEX0，它以十六进制形式输出操作数。允许立即数、直接和栈相对寻址。写一个程序检测新指令的所有特性。

31. 实现一条新的一元指令，替代 NOP，称为 BOOL0，意思是输出布尔值。如果操作数为 0，输出 false，否则输出 true。允许立即数、直接和栈相对寻址。写一个程序检测新指令的所有特性。 444

32. 实现一条新的一元指令，替代 NOP0，称为 STACKADD，它把栈顶最高两项替换为它们的和。例如，代码段

```
LDA     5,i         ;Push 5
STA     -2,s
LDA     9,i         ;Push 9
STA     -4,s
SUBSP   4,i
STACKADD            ;Add 5 + 9
DECO    0,s         ;Output top of stack
ADDSP   2,i         ;Pop the sum
```

应该输出 5 和 9 的和。

33. 本问题实现一条新的、处理浮点数的、非一元指令。假设浮点数除了单元占用 2 字节，1 位符号位，6 位指数位，9 位位数位和一个隐藏位之外，其他的都遵循 IEEE 754 标准。指数采用余 31 码表示，而非规格化数采用余 30 码。

（a）实现一条新的一元指令，替代DECO，称为BINFO，表示二进制浮点输出。允许和DECO同样的寻址方式。值3540 (hex)表示规格化数1.101×2^{-5}，应该输出1.101000000b011010，这里字母b表示是2的幂数，b后面的位序列是-5的余30码表示。值0050 (hex)，表示非规格化数$0.001\ 01 \times 2^{-35}$，应该输出为0.001010000b-30，这里对非规格化数来说，幂数总是为-30。如果值为NaN，那么就输出NaN，正无穷大输出Inf，负无穷大输出-Inf。

（b）实现一条新指令，替代DECI，称为BINFI，表示二进制浮点输入。允许和DECI同样的寻址方式。假设输入总是规格化二进制数。输入1.101000000b011010，表示规格化数1.101×2^{-5}，应该存储为3540 (hex)。

（c）实现一条新一元指令，替代NOP，称为ADDFA，表示浮点累加器加法。允许与ADDA同样的寻址方式。对于规格化和非规格化数，都假设两个加数的指数字段是相同的，但是和的指数字段不一定要和初始的指数字段相同。你的实现需要在执行加法之前插入隐藏位，存储结果之前去除隐藏位。考虑操作数中的一个或者两个可能为NaN或者无限的情况。

445 / 446

（d）假设规格化数或者非规格化数的指数字段可能不相同的情况，再做一遍（c）部分的问题。

存储管理

操作系统的目的是为高级编程提供更加方便的环境，并有效分配系统资源。第8章介绍了操作系统如何为系统中的进程分配CPU时间，而本章将介绍它是如何分配存储空间的。存储空间主要分为两类：主存和外围存储。磁盘存储是最常见的外围存储，也是本章将要描述的。

9.1 内存分配

没有在执行的程序通常存在磁盘文件中。要执行程序就需要主存空间和CPU时间。操作系统把程序从磁盘加载到主存，为程序分配空间；把程序计数器设置为加载到内存里的第一条指令的地址，为程序分配时间。

本章前两节讲述分配主存空间的5种技术：

- 单道程序设计（uniprogramming）
- 固定分区多道程序设计（fixed-partition multiprogramming）
- 可变分区多道程序设计（variable-partition multiprogramming）
- 分页（paging）
- 虚拟内存（virtual memory）

这些技术按照列出的顺序依次变得复杂，每一项都解决了一个性能问题，是对上一项技术的改进。本书没有讨论第六种技术——分段。

9.1.1 单道程序设计

最简单的内存分配技术是单道程序设计，Pep/8操作系统就是这样的例子。操作系统驻留在内存的一端，应用程序在另一端。系统一次只执行一个作业。

因为每个作业都加载在同一个位置，所以翻译器也会相应地生成目标代码。例如Pep/8汇编器假设第一个字节加载到地址0000 (hex)，然后从符号表计算符号地址。或者，如果程序包含一个burn（烧入）指示符，则汇编器假设最后一个字节将加载到内存底部的地址。

单道程序设计有优点也有缺点。主要优点体现在大小方面。这样的系统可以很小、设计简单，因此不容易出错，执行起来几乎没有开销。应用程序一旦加载进来，就会100%占用处理器的时间，因为没有其他进程会中断它。单道程序设计系统适用于嵌入式系统。

单道程序设计的主要缺点是CPU时间利用率很低，而且作业调度很不灵活。相比起主存，磁盘存储的访问时间较长。如果应用程序从硬盘读数据，CPU会一直空闲，等待磁盘输入。这段时间如果能用来执行其他用户的作业就会更好。对于一台微型计算机来说，可以容忍浪费一些CPU时间，但是对于花费几十万的计算机来说就不能忍受了，特别是在多用户系统中，进程是并发执行的，那就更不能忍受了。

甚至在单一用户系统中，作业调度不灵活也是很讨厌的。用户可能想要启动两个程序，并在两者之间来回切换而不退出任何一个。例如，你可能想要运行一会儿字处理程序，然后再切换到画图程序画文档的插图，然后再切换回字处理程序继续刚才的文字编辑。

9.1.2　固定分区多道程序设计

多道程序设计允许并发运行多个应用，解决了CPU利用率不足的问题。要在两个进程间进行切换，操作系统就要把两个程序都加载进主存。当运行进程被挂起时，操作系统会把它的进程控制块存储起来，然后再把CPU交给另一个程序。

要实现多道程序设计，操作系统需要把主存划分成不同的分区，分别存储不同的正在执行的进程。在固定分区方案中，操作系统把内存分成几个大小和位置不会随时间改变的分区。图9-1给出了一种可能的划分，这是一个主存大小为64 KB的固定分区多道程序设计系统。操作系统占用内存底部的16KB。它假设作业的大小并不都一样，所以把剩余的48 KB划分为2个4KB的分区、1个8KB的分区和1个32KB的分区。



图9-1　一个64 KB主存中的固定分区。分区大小和地址以KB为单位。操作系统占据底部的16 KB

为不同进程提供不同的内存分区会带来一个问题，那就是目标代码中的内存引用必须进行相应的调整。假设汇编语言程序员写了一个程序，大小为20KB，操作系统把它装载进一个32KB的分区。如果汇编器假设目标代码会被加载进从地址0开始的内存，那么所有内存引用就都错了。

例如，假设代码开始的几行是

```
0000 040005          BR      AbsVal
0003 0000    number: .BLOCK 2
0005 310003 AbsVal:  DECI    number,d
```

汇编器把符号AbsVal的值计算为0005，因为BR是一条3字节指令，number占用2个字节。现在的问题是BR分支跳转到0005，这是第一个分区中进程的代码地址。不仅这个进程会运行错误，还有可能破坏其他进程的数据。操作系统需要保护进程不受其他并发进程的未授权篡改。

能够把程序加载到内存中任意位置的装载器称为可重定位装载器。8.1节中讲述的Pep/8装载器不是可重定位的装载器，因为它把每个程序都加载到内存的同一位置，也就是0000（hex）。

解决这个问题有几种方法。操作系统可以要求汇编语言程序员决定把程序加载到内存的什么位置。汇编器需要提供一个指令以允许程序员指定目标代码的第一个字节的地址。这样的指令常见的名字是.ORG，意思是origin（起点）。在这个例子中，32KB分区的起始地址是16K或者说8000（hex）。代码的前几行改成如下形式：

```
                     .ORG   0x8000
8000 048005          BR      AbsVal
8003 0000    number: .BLOCK 2
8005 318003 AbsVal:  DECI    number,d
```

其效果就是把应用程序代码中的所有内存引用都加上8000。

如果编译器为应用程序生成目标代码，那么程序员是没有内存地址概念的。翻译器需要和操作系统合作生成正确的目标代码中的内存引用。

9.1.3　逻辑地址

要求程序员或编译器事先指定将目标代码装载到哪里有几个缺点。应用程序的程序员不

需要生心分区的大小和位置，这样的信息与程序员应该没有关系。这种方案背离了操作系统要为高级编程提供方便环境的目标。

448 / 449

它还背离了高效分配系统资源的目标。假设程序员指定把一个3KB的程序加载到第二个4KB的分区，它从地址4K开始，并且相应地设置.ORG指令。在系统运行过程中，即使第一个4KB分区是空闲的，该作业还是有可能等待另一个占用第二个4KB分区的作业才能加载。有未使用的内存就表示资源分配不够高效。

要解决这些问题，操作系统需支持程序员或编译器生成"好像"会被加载到地址0的目标代码。在这种假设下生成的地址称为逻辑地址（logical address）。如果程序被加载到地址不是0的分区中，那么操作系统必须把逻辑地址翻译成物理地址（physical address）。

下面公式描述的是物理地址、逻辑地址和程序加载到的分区第一个字节的地址之间的关系：

$$\text{物理地址} = \text{逻辑地址} + \text{分区地址}$$

在前面代码段的例子中，number的逻辑地址是0003，而物理地址是8003。

有两种可行的地址翻译技术。操作系统可以提供一个软件工具，把分区地址加到目标代码中的所有内存引用上。翻译器需要指定对目标代码中哪些部分进行调整，因为只通过检查原始目标代码，工具不能分辨哪些部分是内存引用。

另一种技术依赖于特殊的硬件，称为基址和边界寄存器。基址寄存器（base register）解决了地址翻译的问题，边界寄存器（bound register）解决的是保护的问题。图9-2展示了对前面的例子来说，基址和边界寄存器是怎么工作的。

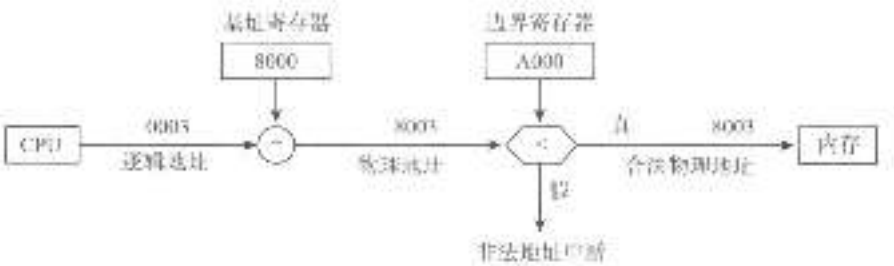


图9-2 用基址和边界寄存器把逻辑地址翻译成物理地址 450

操作系统把使用未修改逻辑地址的目标程序加载到在地址8000的分区中，然后把基址寄存器加载为值8000，边界寄存器加载为值A000（hex）= 48 K，这是该程序加载到的分区的上界。操作系统把程序计数器设置为0000——第一条指令的逻辑地址，也就把CPU交给了该进程。

每当CPU发出一条内存访问请求，硬件就把基址寄存器的内容加到CPU提供的地址上，形成物理地址。CPU会把物理地址与边界寄存器的内容进行比较。如果物理地址小于边界寄存器，那么硬件会完成该内存访问；否则会生成非法地址中断，操作系统必须服务该中断。边界寄存器阻止进程侵入其他进程的内存分区。

这个例子中第一个内存读请求将来自于冯·诺依曼执行周期中的取指令部分。CPU会请求从0000获取一条指令，硬件会翻译成从8000取指令。图9-2给出的是对DECI操作数指示符的逻辑地址0003的翻译。（实际上，在较低的抽象层次上，它是DECI陷阱处理程序中在FF0A处的STA指令的操作数。）

为了切换到另一个进程，操作系统把基址寄存器设置为该进程加载到的分区的地址，把边界寄存器设置为第二个分区的地址。当操作系统从PCB恢复出CPU寄存器（包括程序计数器）时，进程会从它被挂起的地方继续执行。

考虑当操作系统必须调度作业来占用固定分区时所面临的问题。假设图9-1中所有分区都被占用了，除了32KB的分区。如果有一个4KB的作业请求执行，系统应该把它放入这个32KB的分区中，还是应该继续等待更小的可用分区？假设把作业放进32KB的分区中，然后有一个4KB的进程终止。如果此时有一个32KB的作业进入系统，系统就不能加载它了。反之，如果不把小作业调度到大分区中会更好一些，因为这样大作业就能在它一请求执行时就使用大分区了，而小作业也能很快被加载进来。操作系统不能预测进程什么时候结束或者什么时候请求执行，所以没有办法实现最优调度。

另一个问题是操作系统一开始该如何设立这些分区呢？在图9-1中，如果有一个16KB的作业和一个32KB的作业同时请求执行，那么系统只能加载一个，即使实际上用户内存可用空间总量是48KB。另一方面，如果操作系统把用户空间划分为两个大的分区，而有6个或8个4KB的作业请求执行，那么除了两个作业外其他的都会被延迟。还是没有办法实现
451 最优分区，因为操作系统不能预测未来。

9.1.4　可变分区多道程序设计

为了解决固定分区调度中固有的低效率，操作系统可以维护边界可变的分区，方法是只在作业加载进内存时才设立分区。分区的大小可以正好适合作业的大小，这样作业在进入系统时就会有更多内存可用。

当作业停止执行时，它占用的内存区域就可以供其他作业使用了。可供新到来的作业使用的内存区域称为洞（hole）。当操作系统把洞分配给后来的作业时，称为把洞填上了。与固定分区方案一样，操作系统调度作业使得任意时刻内存中的进程数最大。

图9-3给出了一个可用内存区域为48KB的例子，此时尚未调度任何作业。图9-4是一个假想的，对图9-3所示用户内存进行请求的作业序列。当作业停止执行时，会释放它的内存供其他作业使用。

图9-5说明了调度过程。现在问题是必须解决选择标准的问题，也就是当新作业请求内存时先填哪个洞。图9-5使用的方法称为最优适配算法（best-fit algorithm）。在所有大于作业要求内存量的洞中，操作系统选择最小的那个。也就是说，系统选择最适合该作业的洞。

当J1请求12 KB时，对于图9-3中的初始洞，只有一个洞能够用来分配内存。系统把最开始的12KB分给J1，剩下一个36KB的洞。当J2请求8KB时，系统把洞的前面一部分分配给它，对于J3、J4和J5来说过程类似，得到图9-5a所示的内存分配结果。

图9-5b展示了当J1停止执行时的分配状况，此时会清除J1的12KB，在主存顶部生成了一个新的洞。当J6请求一个4KB的区域时，操作系统可以选择将两个洞中的任意一个分配给它。根据最优适配算法，系统选择8KB而不是12-KB的洞，因为较小的洞适配得更好。图9-5c给出了结果。

图9-5d是J5停止之后的分配情况，图9-5e是系统从顶部的洞给J7分配内存之后的情况。现在有三个小的洞分散在整个内存中，这种现象称为碎片（fragmentation）。即使J8想要8KB，可用内存的总量是12 KB，J8也不能运行，因为可用内存不是连续的。

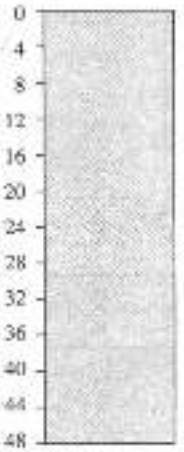


图 9-3 初始可用的用户内存。地址的单位为 KB

作 业	大 小	动 作
J1	12	开始
J2	8	开始
J3	12	开始
J4	4	开始
J5	4	开始
J1	12	停止
J6	4	开始
J5	4	停止
J7	8	开始
J8	8	开始

图 9-4 可变分区多道程序设计系统的作业请求序列。作业大小的单位为 KB

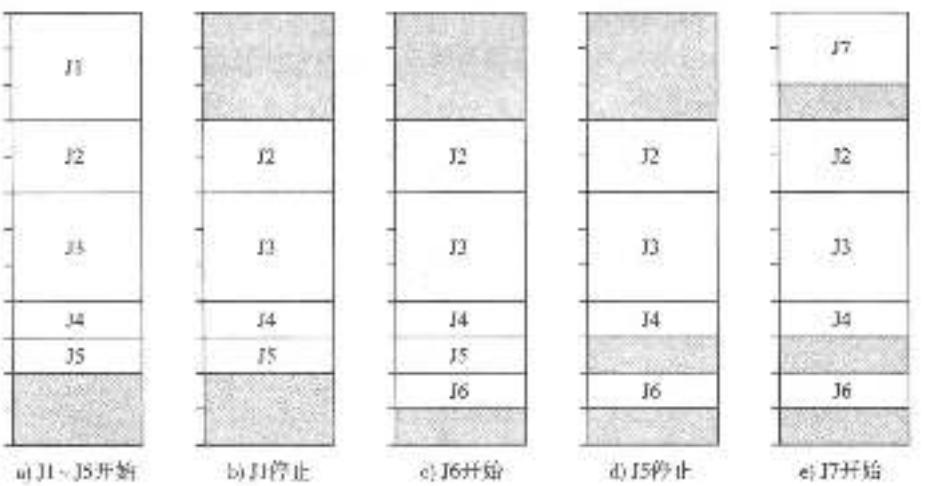


图 9-5 最优适配算法

当出现由于碎片内存而不能满足请求时，操作系统只能等待足够多的进程完成，才能有一块足够大的内存可用。在这个例子中，该请求很小，任意一个作业完成都能释放足够多的内存让 J8 得以加载。

在很拥挤的系统中运行着许多小作业，如果有一个大作业在等待，那么该请求可能要等待很长时间才能得到分配。这种情况下，操作系统可能需要花点儿时间移动一些进程，以得到一个足够大的洞来满足该请求。这个操作称为合并或压缩（compaction）。

图 9-6a 是一种很直观的合并技术。操作系统把进程都移动到内存上部，消除它们之间的所有洞。另一种可行的合并方案是只移动必要的进程，从而得到一个满足请求的、足够大的洞。在图 9-6b 中系统只移动 J6，就可以得到一个足够加载 J8 的洞了。

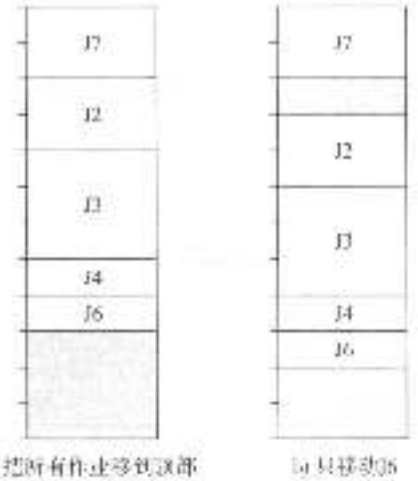
最优适配算法的思路是使用尽可能小的洞，把大的洞留给未来的调度，通过这种方式实现碎片最小化。另一种调度技术看上去可能不那么合理，叫作最先适配算法（first-fit

algorithm）。该算法不是寻找最小的可能的洞，而是从主存顶部开始搜索，从能够容纳该请求的第一个洞开始分配内存。图 9-7 就是最先适配算法对图 9-4 中请求序列的执行轨迹。

图 9-7a 和图 9-7b 与图 9-5 中的最优适配算法一样。在图 9-7c 中，J6 请求一个 4KB 的分区，最先适配算法不会从内存底部最小的洞来分配，而是发现内存顶部的洞，并从它分配空间给 J6。

图 9-7d 中 J5 终止，图 9-7e 中，位于 J6 和 J2 之间的第一个可用的洞就能满足 J7 的 8KB 请求。当 J8 请求 8KB 时，有一个洞可用，系统不需要合并内存。

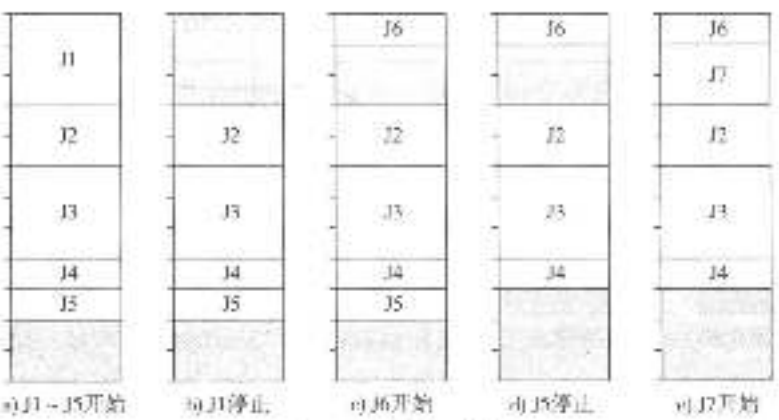
一个例子并不能说明最先适配比最优适配好。实际上，你可以设计一个请求和释放序列，使得使用最先适配比最优适配先需要合并。那么问题就是“平均起来效果是什么样的呢？”实际结果是在内存使用率上，两种算法中没有一种明显优于另一种。



a) 把所有作业移到顶部　b) 释放J6

图 9-6　压缩内存

最先适配算法效果不错的原因在于存储系统总是从内存顶部进行分配，就容易在主存底部形成较大的洞，如图 9-7 所示。



a) J1～J5开始　b) J1终止　c) J6开始　d) J5终止　e) J7开始

图 9-7　最先适配算法

452 ? 454

无论分配策略如何，在可变分区系统中，碎片是不可避免的。非连续的洞就意味着资源分配不够高效。虽然可以通过合并收回那些不可用的内存区域，但这仍然是一个很耗时的过程。

9.1.5　分页

分页是一种解决碎片问题的创造性方法，它不是把几个小的洞合并成一个大的洞供程序使用，而是把程序分解开去适合洞——程序不再是连续的，而是分开分散在整个主存中。

图 9-8 显示的是一个分页系统中正在执行的三个作业。将每个作业划分成页，将主存划分成帧（frame），帧的大小和页相同。该图给出了一个 64KB 内存的前 12KB 空间，帧的大小为 1KB。页的大小总是 2 的幂，实际中通常是 512、1024 或 2048 字节。

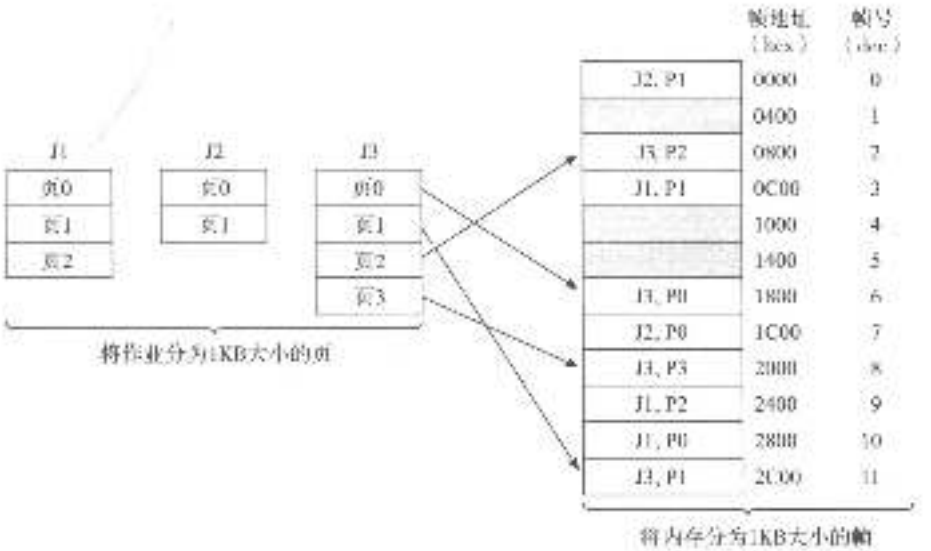


图 9-8 分页系统

作业 J3 的代码分布在主存四个非连续的帧中。位于 1800 的帧中的“J3，P0”表示作业 J3 的页 0，第二页在 2C00，第三页和第四页分别在 0800 和 2000。同样，作业 J1 和 J2 以类似方式分布在内存中。如果作业 J4 到达，需要 3 KB 内存，则操作系统可以把它分布到页 0400、1000 和 1400。系统不需要为新到来的作业而合并内存。

和前面多道程序设计内存管理技术一样，使用分页技术的应用程序会假设使用逻辑地址。操作系统必须在执行时把逻辑地址转换成物理地址。

图 9-9 展示的是图 9-8 所示分页系统中逻辑地址和物理地址的关系。因为页的大小是 1KB，也就是 2^{10}，所以逻辑地址最右边的 10 位是距离页顶部的偏移量，最左边 6 位是页号。

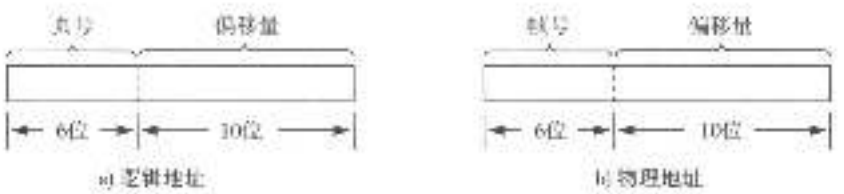


图 9-9 分页系统中的逻辑地址和物理地址

例如地址 058F，二进制表示是 0000 0101 1000 1111。最左边 6 位是 0000 01，表示该地址对应的内存位置在页号 1 内；因为 01 1000 1111 是 399 (dec)，所以该逻辑地址表示距离页号 1 内第一个字节 399 个字节的位置。

参考图 9-8，这个字节的物理地址是距离地址为 2C00 的帧中第一个字节 399 字节的位置。要把逻辑地址翻译为物理地址，操作系统必须把 6 位页号 0000 01 替换成 6 位帧号 0010 11，偏移量保留不变。

在前面的内存管理方案中，一个基址寄存器足以把逻辑地址翻译成物理地址。而分页需要一组帧号，作业的每一页都对应一个帧号。这样的帧号集合称为页表（page table）。图 9-10 给出的是图 9-8 中作业 J3 关联的页表。页表中的每个表项都是用以替换逻辑地址中

页号的帧号。

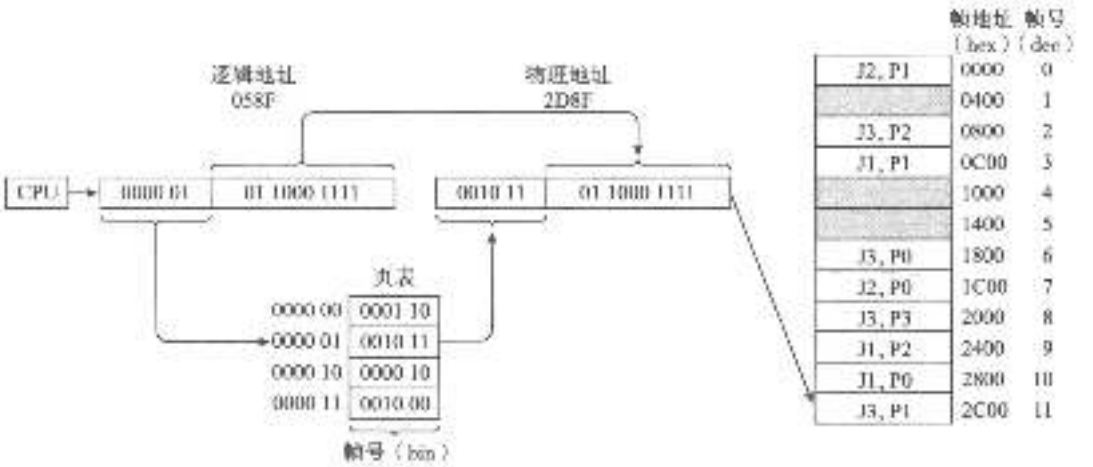


图 9-10　将带页表的逻辑地址翻译成物理地址

假设作业 J3 执行如下语句

```
LDBYTEA 0xC58F,0
```

它会使 CPU 请求从逻辑地址 058F 读内存。操作系统抽出前 6 位 0000 01，然后把它们作为页表地址，一个特殊的硬件内存存储着该作业的帧号。从页表中读出的帧号替换逻辑地址中的页号，得到物理地址。

CPU 发出的指令是读取地址 058F 的内容，实际上是从物理地址 2D8F 读出了一个字节。程序以为它自己被加载到一片从地址 0000 开始的、连续的内存区域，操作系统通过为每个被加载进内存的作业提供一个页表来维护这种假象。这完完全全是个“骗局”，实际上进程在时间上不断中断，在空间上也是分散的，根本不知道自己被“欺骗”了。

分页也没有消除碎片，意识到这一点非常重要。作业的大小很少能刚好是页大小的整数倍，这样一来最后一页中就会有一些内存未被使用，图 9-11 展示的是作业 J3 的情况。作业最后一页中未被使用的内存称为内部碎片（internal fragmentation），区别于可变分区策略中操作系统可见的外部碎片。

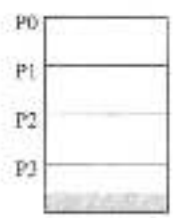


图 9-11　内部碎片

页的大小越小，平均来说内部碎片就越少。不幸的是，这也是需要权衡折中的。页的大小越小，对给定大小的主存，帧数就越多，因此页表就越长。因为每次内存引用都要访问页表，页表通常设计成尽可能快的电路，这是很昂贵的，所以页表必须保持较小以降低成本。

9.2　虚拟内存

看上去好像已经很难再改进分页系统的内存利用率了，但是实际上人们又把分页的概念进一步提升了。考虑一个大型程序的结构，比如说可能会装满 50 页。要执行这个程序，真的有必要把 50 页同时全部装进主存吗？

9.2.1　大程序的行为

大多数大程序都是由几十个过程组成的，其中有些可能根本不会执行。例如负责处理输入错误情况的过程，如果输入没有错误，它就不会被执行。再如初始化数据等其他过程，可

能只执行一次，在剩下的时间里都不再需要执行。

大程序中常见的控制结构是循环。当循环体反复执行时，只有循环内的代码需要驻留在内存中，循环外部（从执行的角度看）距离很远的代码不需要放在内存中。 457

程序可能还包括很大的从未访问的数据区域。例如，如果在C++中声明了一个结构数组，但是不知道程序执行时会遇到多少个这样的结构，可能就会分配比预计多一些的个数。包含这些不会被访问的结构的页就不需要加载。

对典型大程序的分析表明，只把程序中活跃的页加载进内存是可行的。活跃的页包含反复执行的代码和反复访问的数据。

活跃页的集合称为工作集（working set）。随着程序的进展，工作集中会加入新的页，也会有旧的页退出。例如，在执行开始的时候，包含初始化过程的页就在工作集中，随后工作集会包括处理过程的页，而不含有初始化过程的页。

9.2.2 虚拟内存

记住，在较高抽象层次上的程序员以为程序是在逻辑地址从0开始且连续的内存中执行的。假设系统每次只从正在执行的作业加载几个页，但同时仍然保持这个假象，程序员就可能编写根本不能一次性装入主存的超大程序，但是这样的程序仍然能够执行。用户看到的不是已经安装好的有限的内存，而是一个虚拟内存，只受到虚拟地址的限制。

例如，在较老的Pep/7计算机中，地址是16位的，因此理论上可以访问2^{16}字节（64KB）的内存。不过实际上只安装了32KB内存。应用程序从0000开始，不能装超过31 000（dec）字节，否则就落入操作系统的内存区域了。系统安装少于地址位数允许的内存是很常见的，之后用户可以购买更多的内存来升级系统。

假设Pep/7计算机安装了一个支持虚拟内存的操作系统。程序的物理内存被限制为3000字节，但是程序员还是想执行64KB的程序。操作系统把执行程序所需的页面从磁盘加载到内存帧。当一个页含有要执行的语句或要访问的数据时，就需要加载它，操作系统会去除一个不再活跃的页，替换为需要加载的页。程序员看到程序在一个64KB的虚拟地址空间中执行，实际上物理地址空间只有32KB。 458

图9-12展示了如何扩展分页系统以实现虚拟内存系统。系统中有3个作业，J1有10页，

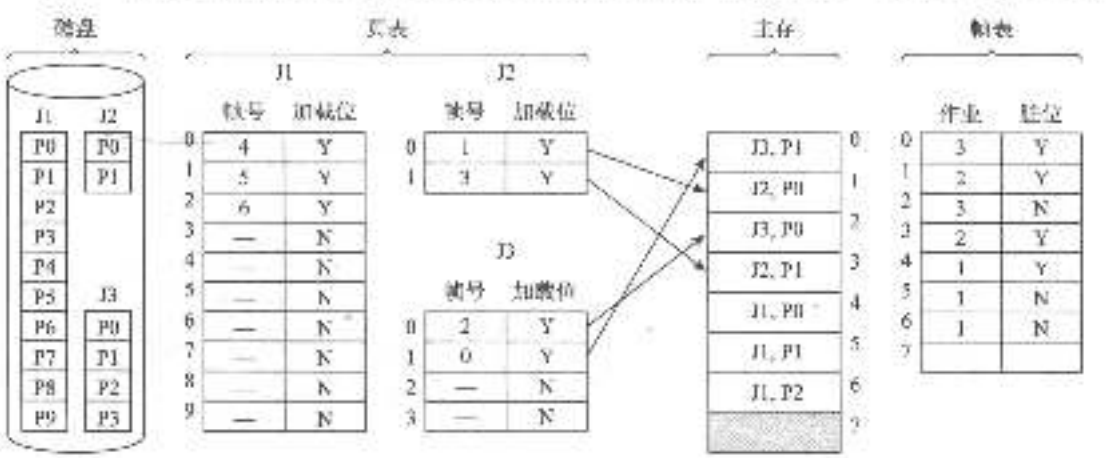


图9-12 虚拟内存的一种实现

J2有2页，J3有4页。注意，物理内存只包含8帧，但是即使J1大于物理内存，页仍然可以执行。操作系统需要一个特殊的硬件，为每个作业保存一张页表，还有一张帧表，每个表项对应一个帧。为了说明简单，帧号用十进制给出。

页表把逻辑地址翻译成物理地址，如图9-10所示。不过在虚拟内存系统中，虽然作业在运行，但是它的有些页并没有加载到内存中。页表中每个页都有额外的一位告诉操作系统该页是否已加载。如果页已经加载到内存中了，则该位为1，否则为0。图9-12用Y表示1，是yes的意思，用N表示0，即no。

帧表是为了帮助操作系统为各个作业从内存中分配帧。第一项表示分配给该帧的作业，第二项这1位称为脏位（dirty bit），它的功能笔后解释。

Ken Thompson和Dennis M. Ritchie

Ken Thompson 1943年生于新奥尔良，他1966年从加州大学伯克利分校电子工程专业硕士毕业，加入贝尔实验室计算研究组。Dennis Ritchie 1941年生于纽约州Bronxville，1968年从哈佛获得数学博士学位，在Thompson之后一年加入贝尔实验室。当时贝尔实验室在与MIT和通用电气就Fernando Corbató领导的Multics项目展开合作，不过在1969年退出了该项目，因为他们认为Multics野心太大，不可能交付一个可用的产品。

贝尔实验室的研究者不愿放弃Multics分时系统的优点，继续努力研发新的可以满足他们需求的操作系统。他们将系统命名为UNIX，和Multics谐音。所以UNIX不是一个首字母缩写词。

贝尔实验室研究组的应用原型是Thompson在Multics系统上写的游戏，名为“星际旅行”，模拟太阳系中行星的运动以及用户引导的一艘火箭飞船。Thompson和Ritchie将该游戏移植到PDP-7计算机上，第一个UNIX文件系统就是在PDP-7上实现的。

UNIX最初的版本是用汇编语言编写的，这是个精挑细选的过程。Thompson和Ritchie不喜欢Multics使用的PL/I语言，因为对于他们在贝尔实验室使用的小型计算机来说，该语言太庞大了。BCPL语言曾用在MIT CTSS系统上，但是有些不好的特性。所以Thompson在BCPL的基础上创造了B语言，使它能够在PDP-7上运行。

1971年，Thompson把UNIX从PDP-7移植到PDP-11，把UNIX移植到新机型上显示出可移植性的重要，以及B语言的所有缺陷。1972年，Ritchie创造了C语言，它的表达目标是用HOL6层语言实现UNIX操作系统，到此可以很容易地把UNIX移植到不同机型上。1973年，Thompson用Ritchie的C语言重写了UNIX。

在20世纪70年代中期，UNIX广泛应用于全世界的学术科研机构。UC Berkeley创造的UNIX的Berkeley Systems Distribution（BSD）对因特网的开发很有帮助。1987年Andy Tanenbaum从零开始完成了UNIX的一个实现，称为Minix，用于他的操作系统教材。1991年芬兰的一个使用他的书的学生——Linus Torvalds使用Minix的思想写出了他自己版本的UNIX，称为Linux。甚至苹果公司也转向UNIX的BSD版本，基于它开发出了自己的最新操作系统，称为OS X。



Alcatel-Lucent版权所有，经授权许可使用

> 由于他们对通用操作系统理论的贡献，特别是实现了UNIX操作系统，Ken Thompson和Dennis M. Ritchie在1983年获得图灵奖。
>
> "UNIX基本上是个简单的操作系统，但是你得是个天才才能理解它的简单。"
>
> ——Dennis Ritchie

9.2.3 按需取页

图9-12说明作业J1和J3并没有把所有页都加载进主存。假设J3在执行页P1中的代码，P1被加载进帧0，接下来就要执行一条LDA指令，它的操作数在P2中。操作系统怎么知道J3需要把P2加载进内存呢？因为操作系统不能预测未来，只有J3实际执行到LDA语句时，它才能知道这一点。

在把逻辑地址翻译为物理地址的过程中，硬件要访问J3的页表，以确定物理地址的帧号。因为加载位为N，所以发生一个称为缺页（page fault）的中断。操作系统会干预并服务该中断。

当缺页发生时，操作系统搜索帧表，确定系统中是否还有空的帧。图9-12显示帧7是可用的，所以操作系统可以把P2加载到该帧中，再更新帧表，记录帧7包含J3的页，也更新J3的页表，将P2记录在帧7中，并把加载位设置为Y。

当操作系统从中断返回时，它会把程序计数器设置为导致缺页的指令地址，即重新执行该指令。这次，当硬件访问J3的页表时，不会再发生中断了，LDA指令的操作数会被放入累加器中。

那么，什么时候操作系统要把一页加载进主存？答案很简单，就是当程序需要它的时候。在前面的例子中，J3通过缺页中断机制请求加载P2。分页机制和按需取页的区别在于按需取页只在有需求的时候才把页载入内存，如果从来不需要某个页，那么就永远也不会装入它。

9.2.4 替换页

当J3需要P3被加载时，操作系统没有问题，因为主存中还有空的帧。不过假设作业请求页的时候，所有帧都被填满了，这时，操作系统必须选择一个已经加载进来的页，把它替换出去，释放它的帧给新请求的页。

被替换的页之后可能还会加载，并可能加载到不同帧中。要保证再次加载的状态和替换前的状态相同，操作系统可能需要保存页的状态，在页被替换时把它写回磁盘。在有些情况下，也可能在被替换时不必把页写回磁盘。

图9-12中，J1有10页存储在磁盘上，其中3页已经加载进了主存。当J1执行LDA和ASLA这样的指令时，不会修改主存中页的状态。LDA引发一个内存读，并把操作数放进累加器中。ASLA会改变累加器的值，该动作不涉及内存读或内存写。这两条指令都不会引起内存写。

但是当J1执行像STA这样的指令时，会改变主存中页的状态。STA把累加器的内容放在操作数的位置上，在进程中进行内存写。如果操作数在帧4的P0中，P0在主存中的状态就会改变。磁盘上的P0页和主存中当前的P0就不完全一致了。如果没有执行过存储指令，那么该页在磁盘上的映像和主存中该页的副本会完全一样。

459 ~ 461

在缺页发生时，操作系统会选择一个页进行替换，如果磁盘上的映像仍然是内存中该页的一个副本，那么就不需要把该页写回磁盘。为了帮助操作系统决定是否需要写回，帧表的硬件中包含了一个特殊位——脏位。

当页被初次加载到空帧中时，操作系统把脏位设置为0，在图9-12中用N表示。如果有存储指令引起写内存，硬件会把该帧的脏位置为1，在图中用Y表示。称这样的页为脏页，是因为它已经不再处于原始的干净状态。如果选择替换某个页，操作系统会检查脏位，决定是否必须在用新页覆盖该帧之前把该页写回磁盘。

9.2.5　页替换算法

在按需分页系统中，操作系统有两项内存管理任务：给作业分配帧，以及当发生缺页且所有帧都满了时，选择替换的页。

一种合理的帧分配策略是假设大作业需要比小作业更多的帧，系统可以按比例分配帧。如果J1是J2的两倍，则执行时需要的帧就应该是J2的两倍。

假设作业执行时的帧数是固定的，操作系统怎么决定缺页发生且该作业的所有帧都满了时，该替换哪个页呢？两种可行的页替换算法是先进先出（First In, First Out, FIFO）和最少被使用（Least-Recently Used, LRU）。

图9-13展示的是FIFO页替换算法的行为，在这个系统中一个作业被分配了三个帧。当作业执行时，CPU向主存发送连续的读写请求流。每个地址的第一组位是页号，如图9-9和9-10所示。页引用执行作业所产生的页号序列。

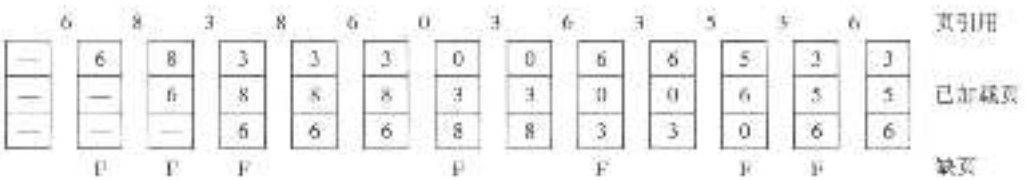


图9-13　使用三个帧的FIFO页替换算法

图9-13表明在第一个请求发生前有三个空页可用。作业请求P6时发生缺页，在图中用F表示，然后P6被加载进一个帧中。

当作业请求P8时又发生缺页，P8被加载到一个空帧中。图中的方框不表示某个特指的帧，可见P6向下移动了一个方框以放入P8，在实际计算机中，P6是不会移动到另一个帧中的。

对P3的引用会再次引发缺页，但是接下来对P8的引用就不会了，因为P8仍然在加载页集合中。类似地，对P6的引用也不会产生缺页。

462

对P0的引用会导致缺页中断，响应该中断必须要选择一个替换页。FIFO算法会替换最先加载的页，因为图中把已有的页往下移，然后放入新的页，所以最先加载的页在底部，即P6。操作系统会用P0替换P6。

当作业有三个帧时，给定的12个页引用引发了7次缺页。如果作业有更多的帧，页引用序列应该会产生更少的缺页。图9-14给出的是FIFO算法对同一个页引用序列但是有四个帧的执行情况。如预期的那样，这个序列产生的缺页次数较少。

通常来说，如果帧数增加，缺页次数会减少，如图9-15a所示，像前面两个例子说明的那样。不过在按需分页系统发展的早期，人们发现了一个很奇怪的现象，对于一个给定的页

引用序列，FIFO 页替换算法实际上可能在使用更多帧的情况下，却产生更多的缺页。

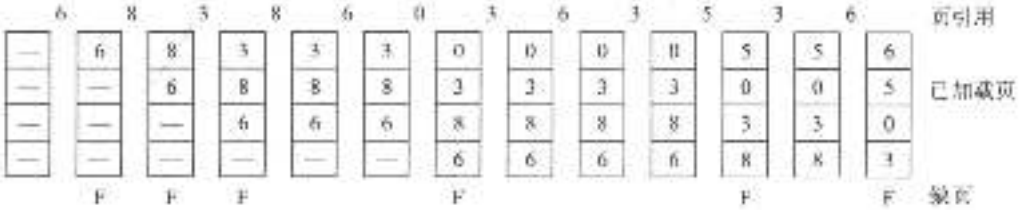
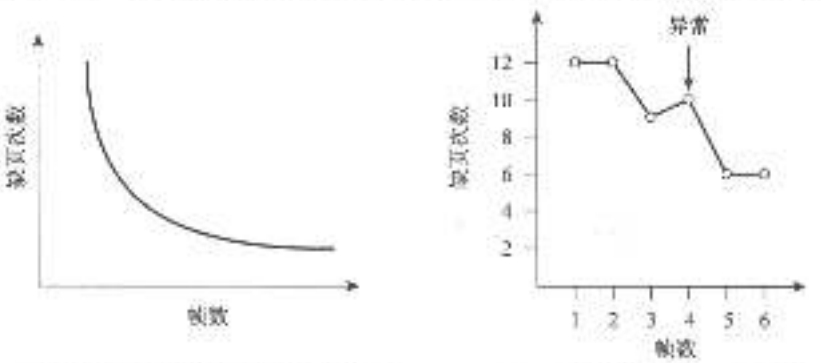


图 9-14 使用 4 个帧的 FIFO 页替换算法

具有这样属性的页引用序列是

0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4

图 9-15b 是该序列帧数与缺页次数的关系图，结果显示 4 个帧的缺页数比 3 个帧更多。这种现象称为 Belady 异常（Belady's anomaly），是根据发现者 L. A. Belady 的名字命名的。



a) 使用更多的帧对缺页次数的预期效果　　b) FIFO替换算法中的Belady异常

图 9-15 帧数对缺页次数的影响

FIFO 算法选择在帧集合里存在时间最长的页，这看上去是个合理的标准。当作业执行时，会进入新的代码和数据区域，来自旧区域的页就不再需要了，所以会替换最老的页。

但是再仔细想想，考虑页最近一次被引用距离当前的时间比考虑页已经在帧集合里的时间可能更好一些。LRU 背后的思想是，最近引用的页比未引用的页更可能在不远的将来被引用。

图 9-16 说明的是对图 9-13 中同样的页引用序列，LRU 页替换算法是怎样工作的。请求 P6，P8 和 P3 会得到同 FIFO 算法一样的状态。接下来请求 P8 会把该页带到顶部的方框里，表明 P8 现在是最近使用过的。后续的请求 P6 会把 P6 带到顶部，把 P8 和 P3 往下移。至中的方框按照前面的使用顺序排序，最近被使用的页在底部。

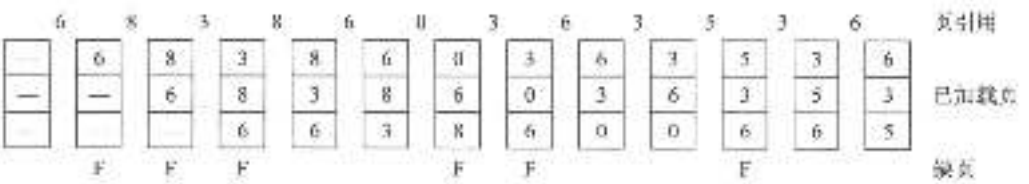


图 9-16 使用三个帧的 LRU 页替换算法

对于这个序列，LRU 算法产生的缺页比 FIFO 算法少一个，不过一个例子并不能说明

LRU 优于 FIFO，因为也有可能构造出一个序列，使 FIFO 产生的缺页次数比 LRU 少。

实际中，操作系统根据具体机器可用的硬件特性会有独特的页替换算法。最常见的页替换算法类似于 LRU，对于真实作业的页请求序列，LRU 的效果通常比 FIFO 更好。从理论上看 LRU 更好一些的一个证明就是 Belady 异常不会发生在 LRU 替换中。

前面例子中的页引用序列只说明了页替换算法，这并不实际。对于一个按需分页的系统，要想高效，缺页率必须保持在每 100 000 次内存引用中会有大约一次缺页。

463 / 464

一个设计合理的、基于按需分页的虚拟内存系统能够满足操作系统的两个目标。它为高级编程提供了方便的环境，因为程序员编写代码时不需要受到物理内存的限制。另一方面，它能高效地分配内存，因为只有在需要的时候，作业的页才会加载进内存。

9.3　文件管理

操作系统还要负责维护磁盘上的文件。文件是一种抽象数据类型（ADT）。对于系统的用户来说，文件包含数据序列，可以由程序或者操作系统命令进行管理。对文件常用的操作包括：

- 创建新文件
- 删除文件
- 重命名文件
- 打开文件进行读
- 从文件读出下一块数据

操作系统要连接起 HOL6 层或 Asmb5 层程序员看到的文件的逻辑组织和文件在磁盘上的物理组织。

9.3.1　磁盘驱动器

图 9-17 展示的是磁盘驱动器的物理特性。图 9-17a 说明硬盘驱动器是由几个覆盖着磁记录物质的盘片组成的，盘片附着在中央转轴上。转轴通常以每分钟 5 400 转的速度旋转。靠近每个磁盘表面的地方有一个读/写头，它连接到一个机械臂。机械臂可以在盘片表面沿着半径方向移动读/写头。

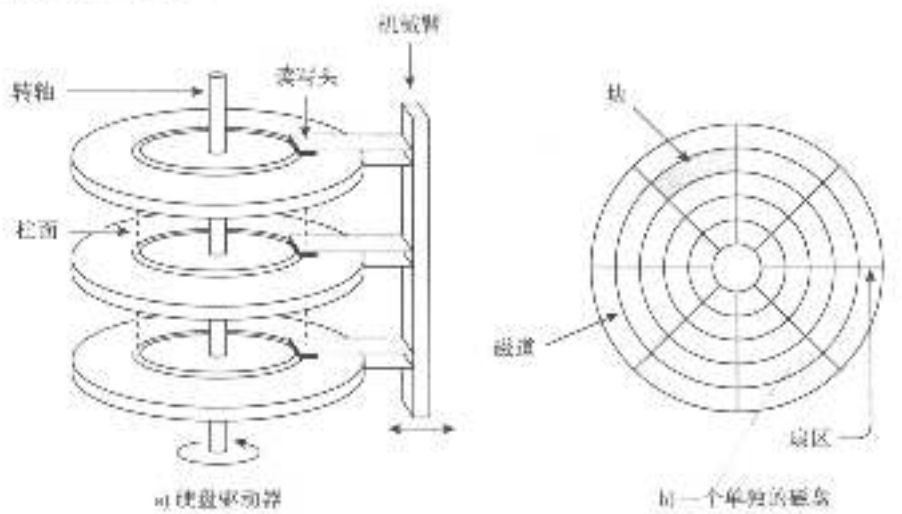


图 9-17　磁盘驱动器的物理特性

图 9-17b 给出的是一个单独的磁盘。如果机械臂固定在一个位置，那么当磁盘旋转时，读/写头下面扫过的区域形成一个环。每个环是一个磁道，它存储着位序列。磁道又划分成馅饼（pie）形状的扇区。块是一个盘片表面的一个磁道的一个扇区。柱面是机械臂位置固定在某个位置时，所有盘片表面上对应磁道的集合。块地址包括三个组成部分——柱面号、盘片表面号和扇区号。

软盘类似于硬盘，除了它的盘片是软的以外。在软盘驱动器中，读/写头物理上是接触磁盘表面的，而在硬盘驱动器中，读/写头是浮在表面之上的，中间有一小层空气垂。硬盘中的磁头碰撞（head crash）是机械故障，磁头擦到了盘片表面，破坏记录物质。

从给定的块读出信息的过程分为四步：1）机械臂把读/写头移动到指定柱面；2）电子电路选择指定盘片表面对应的读/写头；3）等待一段时间，等指定块移动到读/写头下；4）读一个块必须要求整个块经过读/写头。第2）步是一个电子功能，相比起其他3个步骤，发生的时间可以忽略不计。

与3个机械步骤相对应的时间分别是：

- 寻道时间
- 延迟
- 传输时间

寻道时间（seek time）是机械臂移动到指定柱面花费的时间，延迟（latency）是读/写头到位之后块到达读/写头的时间；而传输时间（transmission time）是块经过读/写头的时间。访问一个块所需的时间是这三段时间之和。

9.3.2 文件抽象

在较高抽象层次上的用户不需要担心物理磁道和扇区。操作系统要隐藏物理组织的细节，只向用户展示文件的逻辑组织，把文件作为一种 ADT。

例如，在 C++ 中执行如下语句

```
someFile >> data1
```

这里 someFile 的类型是 <fstream> 中的 ifstream，可以把 someFile 逻辑上看成数据项的线性序列，当前位置处于序列中的某个位置。>> 操作是获取在当前位置的数据项，再把当前位置向前移到序列中下一个数据项处。

物理上，文件中的数据项可以在不同的磁道和盘片上，而且，硬件不维护当前物理位置，读语句的逻辑行为是由操作系统软件控制的。

9.3.3 分配技术

本节最后将讲述三种物理层上的存储分配技术——连续，链接和索引。每种技术都要求操作系统维护一个目录，记录文件的物理位置。这个目录和其他文件一起存储在磁盘上。

如果每个文件都足够小并能装进一个块中，那么文件系统维护起来就很简单，目录只需包含磁盘上所有文件的列表，目录中的每个表项会记录文件的名字和文件存储的块的地址。

如果文件太大不能装进一个块中，那么操作系统必须为它分配多个块。采用连续分配（contiguous allocation）时，操作系统要使文件的物理组织与逻辑组织匹配，把文件连续地放在一个磁道相邻的块中。

如果文件太大放不进一个磁道中，系统会继续往第二个磁道里放。在单面软盘中，第二

465
↓
466

个磁道在同一个盘面上与第一个磁道相邻。在双面软盘或者硬盘中，第二个磁道在第一个磁道的同一盘面上。如果文件还是太大装不进一个柱面，则将文件继续放在相邻的柱面上。

图9-18是连续分配的示意图。每一行8个块代表每个磁道分为8个扇区，和图9-17b一样。每个块上的数字是块地址，是指定块地址所需二个数字的简写形式。块0包含目录。

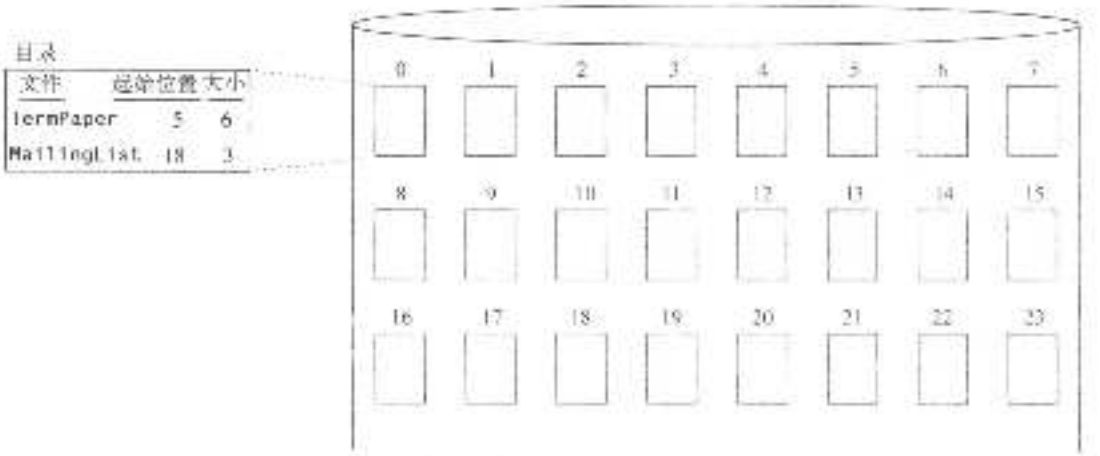


图9-18　磁盘上的连续分配

目录列出了每个文件的名字、起始地址和大小。文件TermPaper从块5开始，包括6个块，最后二个块从第二个磁道继续。系统为什么不把块1～6分配给该文件呢？如果另一个文件先前占用了块1～4，然后又被用户从磁盘删除了，就可能出现图中所示的情况。

图9-18中占用和未占用的磁盘存储模式非常类似于图9-5和9-7中占用和未占用的主存模式。实际上存储管理的问题都是一样的。当文件创建后被删除，存储都会出现碎片。有可能因为有太多分散在磁盘上的小洞而无法创建一个新文件。为了给新文件腾出空间，操作系统提供磁盘合并工具，用来移动磁盘上的文件以制造出一个大的洞，如图9-6所示。

和主存一样，磁盘的合并操作也是非常耗时的。为了避免合并，操作系统可以把文件存储在物理上分散在磁盘上的块中。表9-19中的链接分配技术就是系统用以维护这类文件的一种方式。

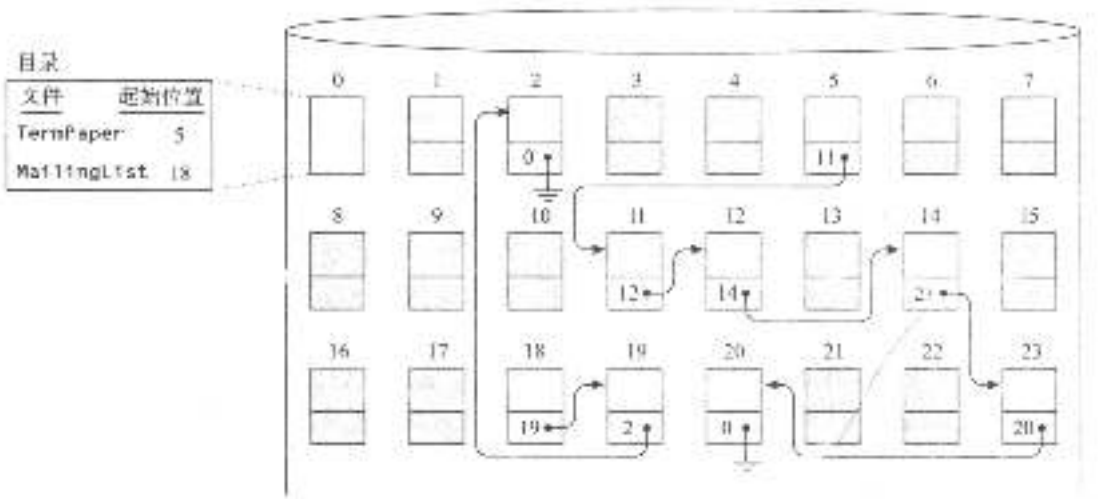


图9-19　磁盘上的链接分配

这个目录包含文件第一块的块址。每个块的最后几个字节保留给下一块的地址。整个块序列形成了一个列表。最后一个块的链接字段值为空（nil），作为分界符。

链接技术的一个缺点是很容易受到故障的影响。在图9-19中，假设块10的链接字段中有一个字节被破坏了。可以因为硬件故障或是软件漏洞，操作系统还能访问文件的前三块，但是没有办法知道后三块在那里。

图9-20中的索引分配技术把所有地址都放进目录里一个称为索引（index）的链表中。这样一来，即使索引中地址10中的一个字节损坏了，操作系统只会丢失那一个块。

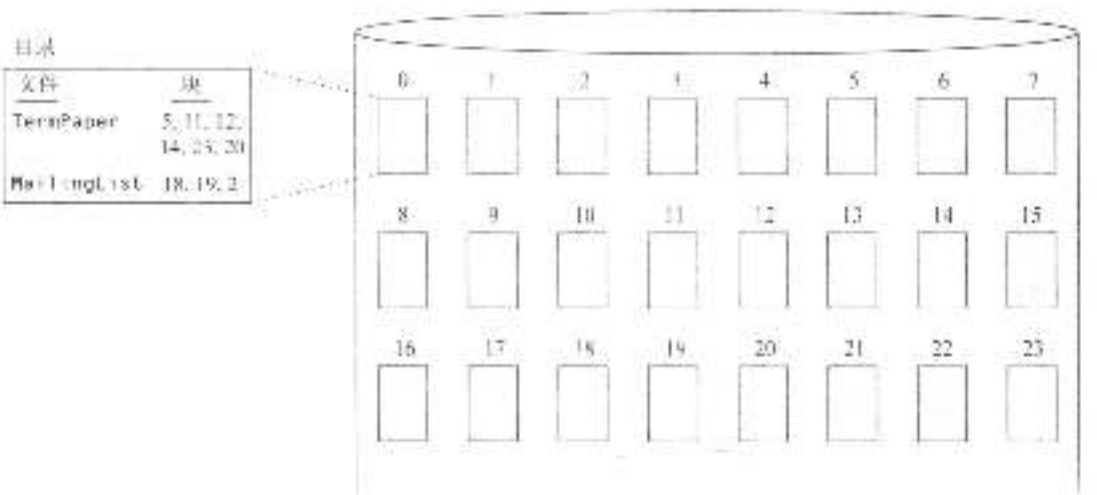


图9-20 磁盘上的索引分配

连续分配相对于非连续分配的主要优点是速度。如果文件存储在一个柱面中，就只需要在开始访问时等待寻道和延迟时间，整个文件的读取速度只受到传输时间的制约。即使文件不止在一个柱面上，在读完第一个柱面后，也只需要等一个很短暂的、到相邻柱面的寻道时间。

如果一个文件的块分散在整个磁盘上，则每访问一个块都需要经历寻道时间和延迟时间。即使是使用非连续分配策略，定期重新组织一下文件的物理布局，使块尽量连续，也是很值得的。这个操作称为*磁盘清理*（defragmenting）。

467
↓
469

9.4 错误检测和纠错码

要想可靠，计算机系统必须能够处理物理错误，这在真实世界里是不可避免的。例如，如果你通过因特网发送了一封电子邮件，传输线路上的一些静电可能会改变一个或几个位，结果接收者收到的位和你发送的不完全一致。再举个例子，系统从主存发送一些数据到磁盘驱动器，由于短暂的机械问题，存储到磁盘上的位模式可能会改变。

对于这类错误有两种解决方法：

- 检测错误并拒绝或丢弃接收到的消息
- 纠正错误

这两种方法使用的都是给消息增加冗余位的技术，用以检测或纠正错误。

9.4.1 错误检测码

假设想要传送一条有关天气情况的消息，有四种可能性——晴朗、多云、下雨和下雪。

发送者和接收者约定好用下面的位模式来对信息编码：

00，晴朗
01，多云
10，下雨
11，下雪

如果下雨，发送者就发送 10。但是在传输线上最后的 0 就变成 1，那么接收者收到 11，会错误地认为是下雪。

检测是否发生了错误最简单的方法是，使用发送者和接收者共同确认的某种计算方法向消息中添加一个冗余位，称为奇偶位（parity bit）。最常见的方法是设置奇偶位为 0 或 1，使得 1 的总数为偶数。采用这种方法，发送者和接受者同意使用下述位模式，其中奇偶位用下划线标明：

000，晴朗
011，多云
101，下雨
110，下雪

470

现在假设发送者发送 101 的下雨消息。如果出现错误把 0 变成了 1，那么接收者会收到 111，并知道发生了错误，因为 111 不是一个事先确定好的位模式。接收者可以请求重传，或者丢弃收到的消息。

注意，错误发生在奇偶位和错误发生在某个数据位一样，接收的消息都是没有用的。例如，如果接收者收到 111，他不知道错误发生在发送 011 时第 1 位错了，还是发送 101 时第 2 位错了，抑或是发送 110 时第 3 位错了。他只知道发生了错误。

发送者和接收者也可商量使用奇校验。奇校验是指校验位的计算是为了让 1 的总数为奇数。对于发送者和接收者来说，他们只需要决定采用什么样的奇偶计算。

如果传输中出现了两个错误会怎么样呢？比如不仅是 0 变成 1，而且最后一个 1 也变成了 0，接收者收到 110。但是现在 110 是一个约定好的模式，那么接收者会错误地认为该消息是下雪的意思。

位模式的集合 {000, 011, 101, 110} 称为编码（code），集合中一个单独的模式（例如 101）称为编码字（code word）。上述编码不能发现两位错，它是一位错检测编码。错误编码会基于一个很现实的假设，即发生一位错的概率要远远小于 1.0，因而发生两位错的概率要远远小于发生一位错的概率。没有哪种编码能 100% 肯定检测出所有错误，因为总有可能同时出现多个错误刚好把一个编码字变为另一个编码字。错误码仍然有用，因为它们能处理大部分错误事件。

9.4.2　编码要求

假设要检测一位或两位错，显然需要更多的奇偶位。问题是“需要多少奇偶位？”并且“该如何设计编码呢？”答案涉及一种距离的概念。两个长度相同的编码字之间的海明距离（hamming distance）定义为它们位不相同的位置的个数。这是根据 Richard Hamming 的名字命名的，1950 年他在贝尔实验室研究并得出了这项理论。

例 9.1　代码字多云 011 和下雨 101 之间的海明距离是 2，因为它们有两个位置上的位不相同，即第一位和第二位。　□

看看天气编码 {000, 011, 101, 110} 可以发现，所有两个代码字之间的距离都是 2。可见能够检测一位错的编码，两个代码字之间的距离不能为 1。假设有这样两个编码字 A 和 B，如果发送者发送 A，传输中发生了一位错，把 A 和 B 不同的那一位反转了，接收者会以为发送的是 B。该编码无法检测出这样的一位错。

编码距离（code distance）是编码中任意一对编码字之间最小的海明距离。

例 9.2 编码 {00110, 11100, 01010, 11101} 的编码距离为 1。虽然几个编码字之间的距离不全为 1，例如 00110 和 11101 的海明距离为 4，但是有一对字之间的距离仅为 1，即 11100 和 11101。如果用这个编码发送天气信息，还是不能保证能够发现所有可能的一位传输错误。 □

要设计一个好的编码，添加奇偶位的方法要使得编码距离尽可能大。要想发现一位错，编码距离必须至少为 2。如果要发现两位错，编码距离应该为多少呢？不能为 2，因为那意味着存在一对编码字 A 和 B 之间的海明距离为 2，如果发送者发送 A，传输中出现的错误刚好把使得 A 区别于 B 的两位反转，接收者就会以为发送的是 B。

图 9-21 是这个概念的示意图。A 是发送的编码字，B 是最接近 A 的编码字，两者之间的空心圆表示并未发送但是由于传输错误有可能接收到的字。图 9-21a 中，发生一位错可能出现 e1，图 9-21b 中一位错会出现 e1，两位错会出现 e2。

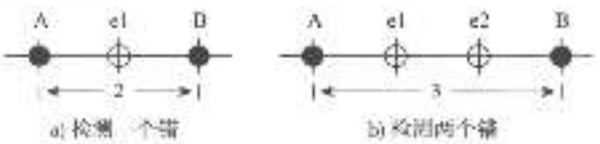


图 9-21 错误检测编码的最小海明距离

总的来说，要发现 d 位错，编码距离必须满足如下公式

$$\text{编码距离} = d + 1$$

例如，要检测三位错，编码距离必须至少为 4，原因是距离 A 最近的字至少为 $d+1$，那么就不可能反转 A 的 d 位把它变成另一个编码字。

距离的概念对纠错码也有用。假设对天气消息采用如下编码：

00000，晴朗

01101，多云

10110，下雨

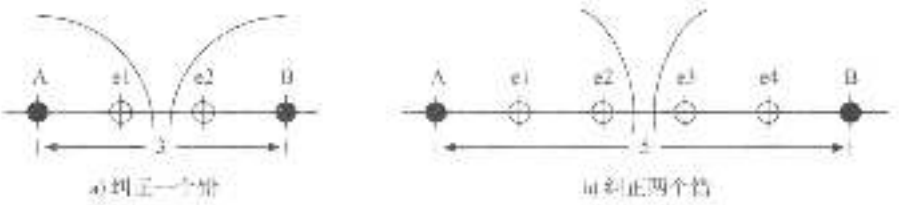
11011，下雪

如果收到 11110，如何判断？可以说发送的是 00000，发生了四位错。但是这是一个合理的结论么？不是，因为 11110 更接近 10110，这是下雨的代码字。如果这样判断，假设的就是发生了一位错，发生一位错的概率要远高于发生四位错的概率。

一般来说，设计纠错编码要添加足够多的奇偶位，使得编码距离足够大，这样接收者才能纠正错误。接收者纠错的方法是计算收到的字和每个编码字的海明距离，选择最接近接收到的字的编码字，这里的“近”是通过海明距离来定义的。

图 9-22 是纠错概念的示意图。同前面一样，A 是发送的编码字，B 是最接近 A 的编码字，空心圆是由于传输错误而接收到的字。图 9-22a 给出的情况是编码能够纠正一位错。编码距离是 3，所以出现了一位错，接收到的 e1 仍然距离 A 比距离 B 更近，所以接收者会认为发送的是 A。如果发送 A，并且出现两位错，收到了 e2，接收者会错误地认为发送的是 B。

图 9-22b 展示的是能够纠正两位错的编码。如果发送 A 且出现两位错，则收到 e2，e2 距离 A 比 B 更近。只有当距离为 5 时才能做到纠正两位错。



a) 纠正一个错　　b) 纠正两个错

图 9-22　纠错码的最小海明距离

总的来说，要纠正 d 位错，编码距离必须满足下面的公式

编码距离 $= 2d + 1$

例如，要纠正三位错，编码距离必须至少为 7。这是由接收方的决定过程导致的：如果接收方收到的字接近 A，就认为发送的是 A，而接收到的字接近 B，就认为发送的是 B。A 和 B 之间的距离必须能够容纳这两组接收到的字，这也就是公式中 d 乘以 2 的原因。同时距离也必须是奇数，所以公式中要 +1。如果距离是偶数，就会出现接收到的字离 A 和 B 一样远，那么接收者就没有办法判断发送哪个字的概率比较高了。

纠正一位错的编码同样可以用来检测两位错，因为它们的编码距离都是 3。这取决于接
473 收者想怎样处理错误。可以假设没有发生两位错并纠正这个错误，也可以更保守一些，假设可能发生了两位错，丢弃该消息或者请求重传。

9.4.3　纠正一位错编码

前面一节中我们描述了检测和纠正错误编码所需要的编码距离。还有一个问题是如何选选编码字以满足要求的编码距离。实际上人们已经设计出了许多不同的能够纠正多错误的编码方法。本节将讨论纠正一位错编码的效率并讲解一种构建此类编码系统的方法。

图 9-23a 给出了一个代码字的结构，有 m 个数据位和 r 个奇偶位。该编码字共 $n = m + r$ 位。编码字如果是 n 位，那么接收到可能的位模式有 2^n 种。图 9-23b 是一种分类方案，可以把这些字分为没有错误和有一位错误的类。图中给出的是 $n = 6$ 时的情况，e1、e2、e3、e4、e5 和 e6 是六种可能收到的字，和 A 有一位不同。如果收到这其中的一个，接收者会认为发送的是 A。类似地，e7、e8、e9、e10、e11 和 e12 是与 B 距离为 1 的可能收到的字。也有可能收到其他不包含在这些组里的字，对应于传输中出现多于一个错的情况，但是这种情况下收到的字与任何编码字的距离不会小于等于 1。



a) 编码字的结构

b) 把接收到的字划分为没有错和有一个错的组

图 9-23　单错误纠错码的结构

一般来说有 n 个字与 A 的距离为 1，所以包括 A 在内，第一组中字的总数是 $(n + 1)$。类似地，B 组、C 组里也都有 $(n + 1)$ 个字，以此类推。每个编码字有一个组，如果有 2^m 个编码字，那么就有 2^m 个组，所以图 9-23b 中字的总数为 $(n + 1)2^m$。可能还有其他一些收到的字不在图 9-23b 中，但是总数不可能超过 2^n。因此，

$$(n + 1)2^m \leq 2^n$$

用 $m=n-r$ 进行替换，两边都乘以 2^n，得到

$$m+r-1 \leqslant 2^r-1$$

这个公式说明了在具有 m 位数据位的消息中，要纠正一位错需要多少个奇偶位 r。

使得上述公式为等号关系的编码称为完美编码（perfect code）。一个例子就是 $m=4$，$r=3$。把奇偶位和数据位一起传输增加了传输时间，对于这种编码，每传4位数据，就要传额外的3位奇偶位。所以，纠错码增加了 3/4 = 75% 的传输时间开销。如果需要传输一个很长的位流，就要把流分割成很多块，再为每一个块添加奇偶位。块越大，开销越小。计算机总是在发送字节流，所以块的大小总是2的幂。图9-24给出了几组 m 值为2的幂时，m 和 r 之间的关系。

数据位 m	奇偶位 r	开销比例
4	3	75
8	4	50
16	5	31
32	6	19
64	7	11
128	8	6

图9-24 单错误纠错码的开销

海明设计了一种非常聪明的方法来决定一位纠错码的奇偶位。它的思想是不要把奇偶位连到编码字的最后，而是把他们分散到编码字中。该技术的优点是接收者可以直接计算出哪一位出错，而不用计算接收到的字与所有编码字之间的距离。图9-25给出的是 $m=8$，$r=4$ 的情况下奇偶位的位置。位的位置从左往右依次编号，奇偶位的位置分别是1、2、4、8，即2的幂数。这里的示例中，传输的数据是1001 1100。但是这些位在编码字中不是连续存储的。

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

图9-25 具有8个数据位的单错误纠错码中4个奇偶位的位置

每个位的位置编号可以写成唯一的一个2的幂数相加之和，如下：

$1=1$　　$5=1+4$　　$9=1+8$

$2=2$　　$6=2+4$　　$10=2+8$

$3=1+2$　　$7=1+2+4$　　$11=1+2+8$

$4=4$　　$8=8$　　$12=4+8$

要确定在位置1的奇偶位的值，注意它出现在位置1、3、5、7、9和11的求和公式的右边。使用偶校验的话，要把该奇偶位设置为使得这些位置中1的总个数为偶数。位置3、7和9是1，所以1的个数是奇数，因此在位置1的奇偶位的值应该设置为1。每个奇偶位要检测的位置分别是：

奇偶位1检测1，3，5，7，9，11

奇偶位2检测2，3，6，7，10，11

奇偶位4检测4，5，6，7，12

奇偶位8检测8，9，10，11，12

你可以自己计算一下编码字中其他几个奇偶位，验证结果 111100101100。

现在假设发送了该编码字，在传输过程中位置10的值发生了错误，接收者收到的是 111100101000，可以看出位置2和8上接收到的奇偶位和计算出来的奇偶位是不同的。因为 2 + 8 = 10，所以可知在位置10的位错了，把位置10的值反转，就纠正了错误。这种纠错技术的好处是接收者不需要把接收到的字与所有编码字进行比较以确定它离哪个编码字最近。

9.5 RAID 存储系统

在计算机发展早期，磁盘体积大且昂贵。随着技术的进步，磁盘体积变小，数据容积增

厂，已没有那么贵了，最后磁盘变得非常便宜，要存储大量数据时，可以把多个驱动器组装成一个驱动器阵列，这样比构造一个超大的驱动器更划算。这样的驱动器组合称为廉价磁盘冗余阵列（Redundant Array of Inexpensive Disks，RAID）系统。

RAID的思想是相对于只有一个转轴的大磁盘驱动器来说，磁盘阵列有更多转轴，每个都有自己的一组读/写头，可以并发操作。同时，冗余也可以用来纠正和检测错误，增加系
476 统的可靠性。RAID控制器向操作系统提供了一层抽象，使得磁盘阵列看上去像一个很大的磁盘。相应地，也可以由软件来提供这层抽象，作为操作系统的一部分。

有几种不同的组织磁盘阵列的方式，几种最常见方案的工业标准术语是：

- RAID 0级：非冗余条带化
- RAID 1级：镜像
- RAID 01和10级：条带化和镜像
- RAID 2级：内存风格的ECC
- RAID 3级：位交叉奇偶校验
- RAID 4级：块交叉奇偶校验
- RAID 5级：块交叉分布奇偶校验

每种组织方式都有各自的优点和缺点，适用于不同的场景。本节接下来将介绍上述各种等级的RAID。

9.5.1　RAID 0级：非冗余条带化

图9-26是RAID 0级的组织结构。本来应该存储在连续块中的数据被分成了条带，分布在阵列中的几个磁盘上。图9-18中块0～7在一条磁道上，8～15在下一条磁道上，以此类推。一个条带包括几个块，例如，如果每个条带2个块，那么图9-18中的块0和块1存在条带0上，块2和块3存在条带1上，以此类推。



图9-26　RAID 0级：非冗余条带化

操作系统看到的是图9-18所示的逻辑磁盘，而实际物理磁盘如图9-26所示。如果操作系统请求磁盘读块0～7，RAID系统会并行地读条带0～3，并发可降低访问时间。如果要服务一个读块0～10的请求，也就是条带0～5，则第一块磁盘需要顺序传输条带0和条带4，第二块磁盘也同样需要顺序传输条带1和条带5。这样的组织架构需要至少两块硬盘。

0级的优点是提高了性能，但是如果大多数读/写请求的都是同一个块或者同一个条带，0
477 级的问题就显现出来了，因为这种情况下没有并行性。同时，它也不像其他级一样有冗余，所以可靠性不是很高。假设所有磁盘质量都相同，那么四块磁盘同时运行出现故障的概率比只运行一块磁盘出现故障的概率要高。

9.5.2　RAID 1级：镜像

对磁盘做镜像是指在另一块独立的磁盘上维护这块磁盘的完全一致的映像，如图9-27所示。不做条带划分，只是严格复制，提供冗余以防磁盘驱动器故障。

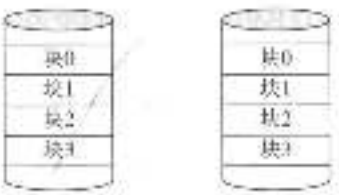
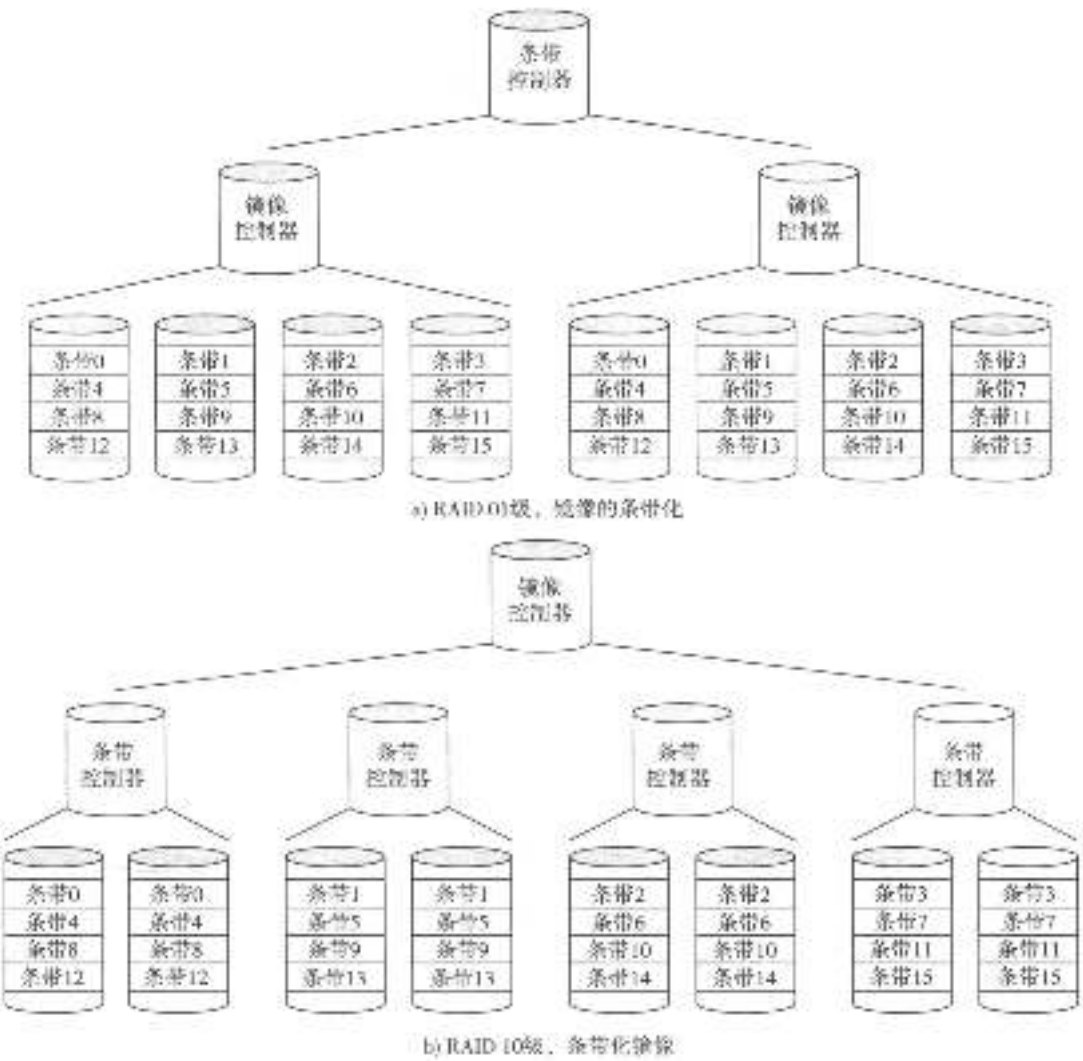


图9-27　RAID 1级：镜像

磁盘写要求写到两个磁盘上，不过这可以并行，所以写性能不会比写一个磁盘差很多。对于磁盘读，控制器可以选择从具有最短寻道时间和延迟的磁盘读，所以磁盘读的性能比只有一个磁盘要好。如果一个磁盘坏掉了，在更换它的同时，另一个磁盘可以继续运行。当替换磁盘装好之后，可以复制好的磁盘，很容易备份好新盘。只有两个磁盘的时候，通常会采用镜像；如果有四个磁盘，通常使用01级的条带化，这样能够带来性能提升，效果更好。

9.5.3 RAID 01级和10级：条带化和镜像

可通过两种方式把RAID 0级和1级结合起来，从而同时具备两者的优势。第一种称为RAID 01级，或0＋1、0/1、“镜像的条带”，如图9-28a所示。采用镜像的条带方式，只需



a) RAID 01级，镜像的条带化

b) RAID 10级，条带化镜像

图9-28 将RAID 0级和1级组合起来

镜像复制0级条带化的磁盘结构。第二种是RAID 10级，或1＋0、1/0、“条带化的镜像”，如图9-28b所示。不是用冗余的磁盘复制0级的磁盘，而是让所有磁盘结对做镜像，然后再在这些镜像之间做条带化。

RAID 10级实现起来比01级代价高一些。图9-28a中的01级，每个条带控制器形成一个系统，使得四个条带化的磁盘对镜像控制器来说看上去像一个磁盘。镜像控制器是使得两个镜像磁盘对计算机来说看上去像一个磁盘的系统。而图9-28b中的10级，每个镜像控制器使得两个互为镜像的磁盘对于条带控制器来说像一个磁盘。条带控制器使得四个条带化的磁盘对于计算机来说看上去像一个磁盘。在这里例子中有八个磁盘，对于01级来说需要三个控制器，而对于10级来说需要五个控制器。

10级相对于01级的优势是可靠性。假设有一块物理磁盘坏了，比如说第三块。图9-28a中，首先第一个条带控制器会向镜像控制器报错，然后镜像控制器会使用右边的镜像磁盘，直到故障盘被替换。实际上在整个故障期间，左边四个物理磁盘都是不工作的。在图9-28b中，第三块盘损坏会导致第二个镜像控制器只使用第四块盘（第二块镜像磁盘），直到故障盘被替换。在故障期间，只有一块物理磁盘是不工作的。

在两种情况中，计算机看到的RAID磁盘的服务都是不间断的，看上去可靠性好像是一样的。但是如果有两块盘同时坏掉，问题就来了。图9-28a中，如果一块坏盘在左边的条带磁盘中，另一块在右边，那么RAID磁盘整个出现故障。如果两块坏盘在同一组条带磁盘中，那么RAID磁盘还能工作。图9-28b中，只有同一对镜像磁盘都坏了，RAID磁盘才不能工作，这个概率要小于01级RAID故障的概率。本章后面有关于这个问题量化分析的练习。

与01级相比，10级还有一个优势是在故障盘被替换后做镜像拷贝的时间更短。图9-28a中，镜像控制器把每个条带磁盘当作一个整体，而不是四块独立的磁盘。每次修复时，镜像控制只能把整个好的条带服务器（即四块盘）的内容都复制到修复好的条带磁盘上。而在10级中，所有镜像都在一对磁盘上完成，修复一块故障盘只需要拷贝一块便盘的内容。

低端RAID系统通常支持01级，而高端系统既支持01又支持10。采用这样的系统，就能既获得条带化的性能优势，又获得镜像的可靠性优势。在有些情况下读性能甚至好于0级。考虑这样一个场景，01级系统，读请求条带0～5。条带0～3可以从第一组驱动器上并发读，条带4和条带5可以从镜像上并发读。01级和10级都要求偶数个硬盘，最少需要四块。

9.5.4　RAID 2级：内存风格的ECC

镜像的存储开销是巨大的，达到100%，因为每块盘都被复制了。图9-24是单错误纠错码（single-Error-Correcting Code，ECC），它的开销更小，通常用在高可靠的存储系统中。用三个奇偶位纠错四个数据位，开销降到75%。它使用2级系统，在位级别上划分条带。图9-29给出了每个半字节在前四个磁盘上的分布。后三个磁盘是单错误纠错码的奇偶校验位。



图9-29　RAID 2级：内存风格的ECC

为了保持性能，驱动器的旋转必须同步。完成一次硬盘写，磁盘控制器计算每个半字节的奇偶校验位，写数据的同时写入校验磁盘。读的时候，控制器从数据计算出校验位，把它们和从校验盘中读出的位进行比较，如果有错误就进行修正。

这种方案用在一些比较老的超级计算机上，通常是32个数据位和6个奇偶校验位，以保证开销较小。今天，便宜的磁盘都有内部位级纠错能力，所以2级系统在商业上不再使用了。

9.5.5 RAID 3级：位交叉奇偶校验

到目前为止，磁盘阵列中最常见的故障是阵列中有一块磁盘损坏，而磁盘控制器可以发现这样的故障，所以系统知道故障在哪里。如果在位级做条带化，那么假如知道哪一位坏掉了，只用一个奇偶位就可以修复该错误。例如，假设要修复半字节1001，使用偶校验，奇偶位是0，那么存储1001 0。图9-30显示1001分别存在位0、位1、位2、位3，而奇偶位0存在P0～3。



图9-30 RAID 3级：位交叉奇偶校验

假设第四块盘坏了，所以知道位3、7、11、15等不可用了。需要读的数据是100x0，x是必须修复的位。因为使用的是偶校验，所以1的个数必须为偶数，所以x必定为1。知道错误发生在哪里有助于将单错误纠错码的开销降低到1个奇偶位。

虽然3级系统比2级系统提高了效率，但还是有一些缺点。3级系统的故障磁盘恢复非常费时。使用镜像，只需把剩下的好盘的内容克隆到替换盘。但是对于位交叉奇偶校验，新替换的磁盘上的位必须从其他磁盘上的位计算而来，因此必须先访问这些位。重建通常是由控制器自动完成的。

只有当磁盘故障需要纠错并恢复替代盘的时候，才会使用奇偶盘。所以，每次写请求都要更新奇偶位。因为每个磁盘驱动器在位级都有自己的ECC，所以每次读请求时不需访问奇偶盘（除非有驱动器坏了）。3级系统的访问时间并不比单个磁盘的访问时间长，但是2级和3级要求每次读/写请求都访问每块数据盘，所以无法获得较大条带带来的并发性。

478 | 481

9.5.6 RAID 4级：块交叉奇偶校验

3级和4级的唯一区别是条带的大小。3级中条带是1位大小的，而4级中条带是一个或多个块。图9-30中P0-3代表1位，但在图9-31中P0-3表示整个条带。



图9-31 RAID 4级：块交叉奇偶校验

例如，如果每个条带是1Kb大小，那么一个文件在条带上的分布如下：

条带0：位0～1023

条带1：位1024～2047

条带 2：位 2048 ~ 3071

条带 3：位 3072 ~ 4095

P0-3 的第 1 位是位 0、1024、2048 和 3072 的奇偶校验位，P0 ~ 3 的第 2 位是 1、1025、2049 和 3073 的奇偶校验位；以此类推。因为条带不是位级别上的，与 2 级和 3 级不同，所以磁盘的旋转不需要同步。

4 级相比起 3 级来说，对于小的随机读请求更有优势。如果每个文件的条带都分布在不同的磁盘上，那么寻道、延迟和传输都可以并行发生。在 3 级系统中，即使只是读一个小文件，也要求所有数据驱动器都保持一致工作，多个小文件必须顺序地读。

虽然比起镜像组织来，4 级的开销已经大幅减少，但是它最大的问题是写请求。如果要写一个跨越条带 0 ~ 3 的文件，可以计算 P0-3 的奇偶位，在写数据的同时写入奇偶校验磁盘。但是假设要写的文件只位含在条带 0 内，那么要改变条带 0 就必须改变 P0-3，而 P0-3 除了是条带 0 的校验盘，同时也是条带 1、2 和 3 的校验盘。看上去需要读条带 1、2、3 和新的条带 0 一起计算新的校验位。不过有更有效的方法，不需要读条带 1、2 和 3，只需读
[482] 出旧的条带 0 和旧的 P0-3。对于新旧数据条带中每个位的位置，如果新的位不同于旧的位，那么就会把 1 的数量从偶数变为奇数（或从奇数变为偶数）。所以必须反转 P0 ~ 3 中对应的位。如果新值和旧值相同，相应的奇偶位就可以保持不变。对于四个数据磁盘，这项技术可以把磁盘读次数从 3 减为 2。

即使采用这样的技术，每次写请求都会要求对奇偶盘进行一次写，不管请求有多小，因此奇偶盘会成为性能瓶颈。

9.5.7　RAID 5 级：块交叉分布奇偶校验

5 级系统能消除奇偶盘成为瓶颈的现象。它不会把所有奇偶位都存储在一个盘上，而是把奇偶信息分散在所有盘上。这样就没有一块盘需要负责整个阵列的奇偶信息了。

图 9-32 给出了一种常见的组织架构，称为左对称奇偶校验分布。把奇偶信息分布到所有磁盘上。它的优点是如果顺序读一组条带，会先访问所有磁盘一次，然后再访问第二次。图中假设按照条带 0、1、2、3、4 的顺序访问，会依次访问第一、第二、第三、第四和第五块磁盘。如果把条带 4 放到图中条带 5 的位置，就需要按顺序访问第一、第二、第三和第一块磁盘，也就是访问了第一块磁盘两次，却一次都没有访问第五块磁盘。可以看到无论从哪个条带开始访问，这个属性都是成立的。

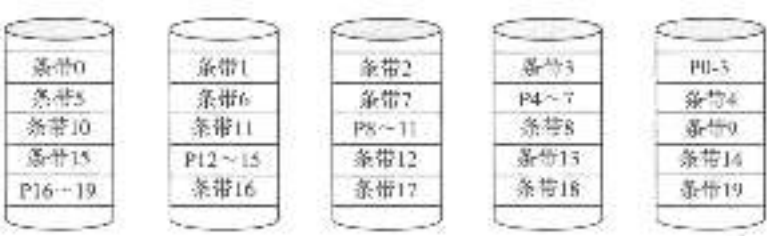


图 9-32　RAID 5 级：块交叉分布奇偶校验

5 级 RAID 是高可靠性、高性能、高容量和低存储开销的理想组合，是目前最流行的高端 RAID 系统。最流行的低端系统可能是 RAID 0 级，它并不是一个真正意义上的 RAID，因为没有冗余，所以也没有增强可靠性。

总结

操作系统以CPU利用率的方式分配时间，以主存和磁盘分配的方式分配存储。分配主存有五种技术：单道程序设计，固定分区多道程序设计，可变分区多道程序设计，分页和虚拟内存。采用单道程序设计，从头到尾都只有一个作业在执行，该作业占用整个主存。固定分区多道程序设计允许几个作业同时执行。要求操作系统在执行任务前决定内存分区的大小。可 483
变分区多道程序设计允许根据作业的要求调整分区的大小，减轻了固定分区多道程序设计中固有的低效率问题。最优适配和最先适配是处理可变分区多道程序设计的碎片问题的两种策略。

分页通过把程序分段装进内存的策略，减轻了碎片问题。程序不连续，而是分段并分散在主存中。作业被划分为大小相等的页。主存也被划分成同样大小的帧。程序员看到的逻辑地址借助于页表被翻译为物理地址。页表包含每个存储在主存中页的帧号。

按需分页也称为虚拟内存，直到作业需要某一页时才把它装载进内存。执行并不要求整个程序都装进主存，相反，只有称为工作集的活跃页才装进内存。先进先出（FIFO）和最近最少被使用（LRU）是两种算法，用来决定当新页需要一个帧来存放时，应将主存中的哪一页替换出去。通常人们会期望随着内存帧数的上升缺页数应该下降，但是Belady异常表明采用FIFO替换算法时，帧数的增加可能会导致缺页数量的增加。

磁盘访问时间由三部分组成：寻道时间，这是机械臂移到指定柱面花费的时间；延迟，这是当机械臂到位后，块旋转到读/写头的时间；传输时间，这是块经过读/写头的时间。磁盘管理的三项技术是连续、链接和索引。和主存一样，磁盘存储也有碎片的问题。

可以往数据位中添加一些冗余位以发现或纠正数据传输或存储中可能发生的错误。两个代码字之间的海明距离是不同位的个数。接收者通过选择距离接收到的字海明距离最近的编码字来纠正错误。通过正确选择冗余位放置的位置，可以不用把接收到的字与所有编码字都做比较就纠正一位错误。

廉价磁盘冗余阵列（RAID）是一组磁盘，在操作系统看来就好像是一个大的磁盘。RAID系统的两个优势是性能和可管性：性能来源于系统中有多转轴实现的并发数据访问；可靠性来源于冗余磁盘驱动器实现的错误纠正和错误发现。

练习

9.1节

1. 采用图9-4中的格式，设计一个作业执行序列，使得最先适配算法比最优适配算法先需要合并压缩。画出每种算法在需要合并之前主存的碎片情况。

2. 图9-10是一个分页系统的页表。它起到和多道程序设计系统中基址寄存器一样的翻译逻辑地址的作用。图中没有给出边界寄存器对应的功能是如何完成的。(a) 要保护进程的内存空间不受非法访问，分页系统是否需要边界寄存器？每页一个还是一个边界寄存器就足够了？(b) 修改图9-10，加上不受其他进程非法访问的内存保护机制。 484

3. 假设一个分页系统的页大小是512字节。(a) 如果大多数文件都很大，比512字节大很多，那么每个文件的平均内部碎片大小会多少（以字节为单位的未使用空间）？解释你的推理。(b) 如果大多数文件都远小于512字节，那么(a)部分中问题的答案是什么？(c) 如果以未使用空间的比率而不是未使用字节数来表示碎片，那么(b)问题的答案又该是什么？

9.2节

4. 一个计算机具有12位地址和划分为16个帧的主存。内存管理使用按需分页。*(a) 虚拟内存有多少字节？(b) 每页中有多少字节？(c) 逻辑地址和物理地址的偏移量是多少位？(d) 一个作业的

页表中虚多有多少表项？

5. 对一个具有 n 位地址和内存划分为 2^k 个帧的计算机，回答练习 4 中的问题。

*6. 图 9-12 中哪些页被虚映像和内存中的页是完全一致的？

7. 图 9-15b 显示的是 Belady 异常，针对书中给出的页引用序列进行验证，以图 9-13 中的格式展示帧的内容。

*8. 设计一个 12 个页引用的序列，使得 FIFO 页替换算法比 LRU 算法好。

9. 采用 FIFO 页替换算法针对图 9-13 的页引用序列，画出类似图 9-15b 所示的图。

*10. 如果操作系统能够预测未来，它会选择能使下一次缺页尽可能迟发生的页面来替换。这样的算法称为 OPT，即最优页替换算法。这是一个很有用的理论算法，因为它代表你能做到的最好情况。当设计者衡量页替换算法时，会试着尽量接近 OPT 的性能。对于图 9-13 和图 9-16 中的序列，OPT 会导致的缺页次数是多少？和 FIFO 和 LRU 相比如何？

9.3 节

11. 假设一块磁盘每分钟旋转 5 400 转，每个盘面分为 16 个扇区。*(a) 可能的最大延迟时间是多少？(b) 可能的最小延迟时间是多少？什么情况下会发生？(c) 根据 (a) 和 (b)，平均延迟时间是多少？(d) 一个块的传输时间是多少？

9.4 节

12. *(a) 存储一个 0～9 的十进制数字需要多少数据位？*(b) 检测出一位错需要多少奇偶校验位？
(c) 写出一个使用偶校验的单错误检测码。奇偶位用下划线标出。(d) 你的编码的编码距离是多少？

485

13. (a) 要检测出五位错，编码距离至少为多少？(b) 要纠正五位错，编码距离至少为多少？

14. (a) 对 9-24 中缺些表项表示完关编码？(b) 补充图 9-24 中的表，包括 m = 4 和 m = 128 之间的所有完关编码，还要写出开销值。(c) 关于把数据位数限制为 2 的幂会带来的开销，可以得出什么结论？

15. 传输 8 个数据位的一组数据，使用图 9-25 中的单错误纠错码。对于收到的下列每个位模式，说明是否发生错误。如果发生请纠正。

*(a) 100110101001　　(b) 110100110010
(c) 000010110100　　(d) 101100100100

9.5 节

17. 图 9-28 中的 RAID 系统有 8 个均匀磁盘。(a) 如果用 6 个物理磁盘，对于 01 级和 10 级系统，各需要多少个镜像控制器和条带控制器？(b) 通常有 $2n$ 个磁盘（图 9-28 中是 $n = 4$），对于 01 级和 10 级系统，各需要多少个镜像控制器和条带控制器？

18. (a) 图 9-28 给出的是具有 8 个物理磁盘的 RAID 01 级和 10 级系统。画出 4 个物理磁盘相应的 01 级和 10 级系统。(b) 假设坏了两个磁盘，序列 BBGG 意思是第一个和第二个磁盘坏了，第三个和第四个是好的。在这种情况下，RAID 01 级系统是好的，因为坏的两个磁盘是在同一组条带化的磁盘上，而 RAID 10 级就不能正常工作了，因为坏掉的两个磁盘在同一组先做镜像的磁盘中。有 2 个 B 和 2 个 G 的四个字母的组合有多少种变化？(c) 画出每种变化并确定对 01 级和 10 级来说，此时 RAID 系统是否能正常工作。(d) 如果两个磁盘故障，根据 (c) 部分，确定 01 级和 10 级 RAID 能够正常工作的概率。哪种 RAID 系统更可靠？(e) 还记有 $2n$ 个磁盘（图 9-28 中是 $n = 4$），有 2 个 B 和 $2n-2$ 个 G 的 $2n$ 个字母有多少种组合？(f) 在 (e) 部分中，有多少种组合会导致 01 级或 10 级系统失效？(g) 如果有两个磁盘故障，根据 (f) 部分，确定 01 级和 10 级 RAID 能够正常工作的概率。(h) 根据 (g) 部分，说明图 9-28 中 RAID 磁盘失效的概率，在两个磁盘故障的情况下，对 01 级来说是 4/7，10 级是 1/7。

19. 有一个 RAID 4 级系统，八个数据盘和一个奇偶盘。(a) 如果不使用旧数据和旧奇偶值，写一个数据条带需要多少磁盘读和磁盘写？(b) 如果使用旧数据和旧奇偶值，写一个数据条带需要多少磁盘读和磁盘写？

486

第六部分

Computer Systems, Fourth Edition

逻辑门层（第1层）



组合电路

我们终于要讲到典型计算机系统的最底层了。每个抽象层都隐藏了上一层用户不需要的细节。LG1 层的细节对 ISA3 层，即指令集架构层的用户来说是隐藏的。要记住 ISA3 层用户看到的是使用机器语言的冯·诺依曼机器，LG1 层设计者的工作是构建 ISA3 层机器。最后三章讲述 LG1 层语言和设计原理，这些内容对于构建冯·诺依曼机器是必需的。

本书中的图都会有指令集架构层和逻辑门层之间的微代码层。一些设计者在机器中选择省略微代码层而直接从 LG1 层构建 ISA3 层机器，还有一些设计者则选择使用微代码层。

这两种设计方法的优点和缺点是什么呢？答案和我们在第 7、第 6 和第 5 层遇到的一样。假定需要给 App7 层的用户设计一个应用程序，你是愿意用 HOL6 层的 C++ 来编写，然后把它编译到较低的层上，还是愿意直接用 Asmb5 层的 Pep/8 汇编语言来编写呢？因为 C++ 在较高的抽象层级，一条 C++ 语句能做多条 Pep/8 语句的工作，所以 C++ 程序要比等效的 Pep/8 程序短很多，因此也更容易设计和调试。但是需要用编译器把 C++ 程序转换到较低的层级，而且优秀的汇编语言程序员通常能生成甚至比优化编译器产生的目标代码还要简短和快速的代码。汇编程序可以执行得更快，但是很难设计和调试，因此开发成本高昂。

487
↓
489

在第 7、第 6 和第 5 层要权衡的是开发成本和执行速度，在第 3、第 2 和第 1 层也是一样。通常，包括 Mc2 层的系统要比省略它的系统简单并且成本更低，但是它们通常比直接从 LG1 层构建的系统执行得更慢。最近的设计趋势是用小指令集构建简单但快速的冯·诺依曼机器，称为精简指令集计算机（Reduced Instruction Set Computer，RISC）。RISC 机器的一个重要特性就是它省略了 Mc2 层。

逻辑门层下有两个层级是很有趣的，如图 10-1 所示，但在本书中不做描述。在电子设备层（第 0 层），设计者连接晶体管、电阻和电容形成 LG1 层的逻辑门。在物理层（第 −1 层），应用物理学家构建电子工程师构成门电路所需的晶体管，而计算机设计师用门来构建冯·诺依曼机器。物理层下没有别的层级，它是所有科学的基础。



图 10-1 逻辑门层以下的层

第 0 和第 −1 层的语言是一组数学公式，对本层的对象行为建模。你也许熟悉其中的一些。第 0 层包括欧姆定律（Ohm's law）、基尔霍夫定律（Kirchoff's rule）以及电子设备的电压电流特性。第 −1 层包括库仑定律（Coulomb's law）、牛顿定律（Newton's law）和一些量子力学定律。在所有层级上，从 App7 层的关系数据库计算到 −1 层的牛顿定律，形式化数学都是对系统行为建模的工具。

10.1 布尔代数和逻辑门

电路（circuit）是物理上由线路连接起来的设备集合。LG1 层电路有两种基本类型：组合电路和时序电路。可以把任一种电路类型表示成一个叫作黑盒（black box）的长方块，有

固定数量的输入线和输出线。图 10-2 展示了一个三输入二输出的电路。

每条线路携带一个信号，其值可以为 1 或 0。在电子学上，信号 1 是一个小电压，通常大约是 5V，信号 0 是 0V。电路被设计成只能检测和产生这样两种二元值。

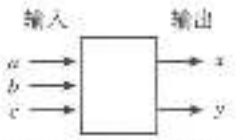


图 10-2　电路的黑盒表示

可以把图 10-2 看成一种说明输入－处理－输出结构的方式，在计算机系统的所有层级都能看到。电路执行处理，把输入转换为输出。

10.1.1　组合电路

对于组合电路（combinational circuit），输入决定输出。例如在图 10-2 中，如果今天输入是 $a=1$，$b=0$，$c=1$（缩写为 $abc=101$），得到输出 $xy=01$；明天输入是 $abc=101$，还是得到输出 $xy=01$。在数学上，x 和 y 是 a、b 和 c 的函数，即 $x=x(a,b,c)$ 和 $y=y(a,b,c)$。 490

时序电路则不然。如果对时序电路输入 $abc=101$，可能有时会得到 $xy=01$，但是几微秒后得到 $xy=11$。第 11 章会解释这种看上去没有什么意义的行为是怎样发生的，以及对于构建计算机而言这种行为为什么是不可或缺的。

描述组合电路行为的三种最通用的方法是：

- 真值表；
- 布尔代数表达式；
- 逻辑图。

本节剩余部分将描述这些表达方法。

10.1.2　真值表

表达组合电路的三种方法中，真值表要比代数表达式和逻辑图的抽象层次更高。真值表说明组合电路做什么，而不是说明怎样做。真值表只是列出输入值每种可能组合的输出（这就是组合电路名称的由来）。

例 10.1　图 10-3 是一个三输入二输出组合电路的真值表。因为有 3 个输入，每个输入有两种可能的值，因此这个表有 $2^3=8$ 个条目。通常情况下，n 输入组合电路的真值表有 2^n 个条目。 □

例 10.2　图 10-4 是另一个由真值表说明组合电路的例子，它是一个四输入的电路，其真值表中有 16 个条目。 □

a	b	c	x	y
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

图 10-3　一个三输入二输出的组合电路

a	b	c	d	x	y
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	1	1	0

a	b	c	d	x	y
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0

图 10-4　一个四输入二输出的组合电路

图 10-2 的黑盒图无其适合组合电路的真值表表达。人们看不到涂黑的盒子里面是什么样的，同样也看不到电路怎样生成真值表定义的函数。

10.1.3　布尔代数

根据布尔代数定律写出的代数表达式不仅说明组合电路做什么，而且说明它是怎样做的。布尔代数在某些方面类似于我们熟悉的实数代数，但是在其他方面是不一样的。实数代数的四个基本运算是加、减、乘、除，布尔代数的三个基本运算是 OR（符号为“+”）、AND（符号为“·”）和“求补”（符号为“′”）。AND 和 OR 是二元运算，求补是一元
491 运算。

布尔代数的 10 个基本属性是

$x+y=y+x$	$x\cdot y=y\cdot x$	交换律
$(x+y)+z=x+(y+z)$	$(x\cdot y)\cdot z=x\cdot(y\cdot z)$	结合律
$x+(y\cdot z)=(x+y)\cdot(x+z)$	$x\cdot(y+z)=(x\cdot y)+(x\cdot z)$	分配律
$x+0=x$	$x\cdot 1=x$	恒等律
$x+(x')=1$	$x\cdot(x')=0$	互补律

其中，x、y 和 z 是布尔变量。和实数代数一样，表达式中插入圆括号表示哪些运算要先执行。为了简化，避免表达式中出现过多括号，布尔运算有如图 10-5 所示的优先结构。使用这种优先规则，分配律变成

$x+y\cdot z=(x+y)\cdot(x+z)$　　$x\cdot(y+z)=x\cdot y+x\cdot y$

互补律变成

$x+x'=1$　　$x\cdot x'=0$

优先级	运算符
最高	求补
	AND
最低	OR

图 10-5　布尔运算符的优先级

实数代数和布尔代数属性之间的一个显著不同是分配律。对于实数，乘法在加法上有分配律，例如

$2\cdot(3+4)=2\cdot 3+2\cdot 4$

但是加法在乘法上没有分配律，例如

$2+3\cdot 4=(2+3)\cdot(2+4)$

是不成立的。然而在布尔代数中，“+”代表 OR，“·”代表 AND，OR 在 AND 上没有分配律。

布尔代数的定律有实数代数没有的对称性，每个布尔属性都有对偶性（dual property）。做如下操作

- 交换“+”和“·”
- 交换 1 和 0

就能得到对偶的表达式。分配律的两种形式就是对偶表达式的一个例子。在分配律

$x+(y\cdot z)=(x+y)\cdot(x+z)$

中，如果交换“+”和“·”运算符，就会得到

492　$x\cdot(y+z)=(x\cdot y)+(x\cdot z)$

这是另一个分配律表达式。布尔代数的每个基本属性都有相应的对偶性。

结合律也使得表达式可以进一步简化。因为执行两个 OR 运算的顺序是不重要的，你可以把

$(x+y)+z$

写成没有括号的

$$x+y+z$$

这也适用于 AND 运算。

10.1.4 布尔代数定理

因为布尔代数和我们熟悉的实数代数有不同的数学结构，所以布尔代数的定理初看之下可能显得很特殊。接下来要讨论从布尔代数的 10 个基本属性证明出来的定理，它们在组合电路的分析和设计中非常有用。

幂等性（idempotent property）表明

$$x+x=x$$

证明这个定理需要一系列的替换步骤，每一步都基于布尔代数的 10 个基本属性之一：

$$\begin{aligned}
& x+x \\
= \quad & <\text{恒等律}> \\
& (x+x)\cdot 1 \\
= \quad & <\text{互补律}> \\
& (x+x)\cdot(x+x') \\
= \quad & <\text{分配律}> \\
& x+(x\cdot x') \\
= \quad & <\text{互补律}> \\
& x+0 \\
= \quad & <\text{恒等律}> \\
& x
\end{aligned}$$

493

对偶性为

$$x\cdot x=x$$

对偶定理的证明和上述步骤完全一样，每个替换都基于原证明中对应步骤的对偶属性：

$$\begin{aligned}
& x\cdot x \\
= \quad & <\text{恒等律}> \\
& (x\cdot x)+0 \\
= \quad & <\text{互补律}> \\
& (x\cdot x)+(x\cdot x') \\
= \quad & <\text{分配律}> \\
& x\cdot(x+x') \\
= \quad & <\text{互补律}> \\
& x\cdot 1 \\
= \quad & <\text{恒等律}> \\
& x
\end{aligned}$$

幂等性质的证明展示了布尔代数中对偶性的一个重要应用。一旦证明了一个定理，那么就可以立即断言它的对偶定律也一定成立。因为 10 个基本属性中的每一个都有对偶属性，所以对应的证明在结构上和原证明是相同的，只不过每一步都基于原始步骤的对偶属性。

这里有三个更有用的定理及它们的对偶定理。证明定理的数学规则是可以在证明中使用任何公理或以前已经证明了的定理。要证明下面第一个定理，可以使用任何基本属性和幂等性。要证明第二个定理，可以使用任何基本属性、幂等性和已经证明了的第一个定理，等等。第一个定理

$$x + 1 = 1 \qquad\qquad x \cdot 0 = 0$$

称作零元定理。0 是 AND 运算符的零元，1 是 OR 运算符的零元。第二个定理

$$x + x \cdot y = x \qquad\qquad x \cdot (x + y) = x$$

叫作吸收律，因为 y 被吸收到 x 中。第三条定理

494　$x \cdot y + x' \cdot z + y \cdot z = x \cdot y + x' \cdot z$　$(x + y) \cdot (x' + z) \cdot (y + z) = (x + y) \cdot (x' + z)$

叫作合意（consensus）定理。这几个定理的证明作为章末练习。

10.1.5　互补证明

x 的补是 x'。要证明表达式 y 是表达式 z 的补，必须证明 y 和 z 遵循同样的互补属性，就像 x 和 x' 一样，即

$$y + z = 1 \qquad\qquad y \cdot z = 0$$

互补证明的一个例子是德·摩根定律（De Morgan's law），它表明

$$(a \cdot b)' = a' + b'$$

要证明 $a \cdot b$ 的补是 $a' + b'$，必须证明

$$(a \cdot b) + (a' + b') = 1 \qquad\qquad (a \cdot b) \cdot (a' + b') = 0$$

这个证明的第一部分如下

$(a \cdot b) + (a' + b')$

$=$　< 交换律 >

$(a' + b') + (a \cdot b)$

$=$　< 分配律 >

$[(a' + b') + a] \cdot [(a' + b') + b]$

$=$　< 交换律和结合律 >

$[b' + (a + a')] \cdot [a' + (b + b')]$

$=$　< 互补律 >

$[b' + 1] \cdot [a' + 1]$

$=$　< 零元定理，$x + 1 = 1$ >

$1 \cdot 1$

$=$　< 恒等律，$(x \cdot 1 = 1)[x := 1]$ >

495　1

证明的第二部分如下

$(a \cdot b) \cdot (a' + b')$

$=$　< 分配律 >

$(a \cdot b) \cdot a' + (a \cdot b) \cdot b'$

$=$　< 交换律和结合律 >

$b \cdot (a \cdot a') + a \cdot (b \cdot b')$

$=$　< 互补律 >

$b \cdot 0 + a \cdot 0$

$=$ <零元定理，$x \cdot 0 = 0$>

$0 + 0$

$=$ <恒等律，$(x + 0 = x)[x := 0]$>

0

德·摩根第二定律

$(a + b)' = a' \cdot b'$

直接从对偶性得出。

德·摩根定律可推广到多个变量的情况。对于 3 个变量，该定律是

$(a \cdot b \cdot c)' = a' + b' + c' \qquad (a + b + c)' = a' \cdot b' \cdot c'$

多于两个变量的一般性定理的证明作为章末练习。

另一个互补定理是 $(x')' = x$，x' 的补是 x，因为 $x' + x = 1$，其证明如下

$x' + x$

$=$ <交换律>

$x + x'$

$=$ <互补律>

1

以及 $x' \cdot x = 0$，其证明如下

$x' \cdot x$

$=$ <交换律>

$x \cdot x'$

$=$ <互补律>

0 496

另一个互补定理是 $1' = 0$，1 是 0 的补，因为 $1 + 0 = 1$，其证明如下

$1 + 0$

$=$ <恒等律，$(x - 0 = x)[x := 1]$>

1

以及 $1 \cdot 0 = 0$，其证明如下

$1 \cdot 0$

$=$ <交换律>

$0 \cdot 1$

$=$ <恒等律，$(x \cdot 1 = x)[x := 0]$>

0

可以直接得出对偶定理 $0' = 1$。

10.1.6 逻辑图

组合电路的第三种表示方法是逻辑门的互联。因为逻辑图中连接门的线路表示连接电路板或集成电路上物理设备的物理线路，所以这种表现形式最接近于硬件。

每个布尔运算由一个门符号来表示，如图 10-6 所示。AND 和 OR 门有两条输入线，标识为 a 和 b；反相器只有一条输入线，输出是 x，这对应于求补是一元运算的事实。图中也

展示出了每个门对应的布尔表达式和真值表。



a	b	x
0	0	0
0	1	0
1	0	0
1	1	1

a) AND门

a	b	x
0	0	0
0	1	1
1	0	1
1	1	1

b) OR门

a	x
0	1
1	0

c) 反相器

497

图 10-6　三类基本的逻辑门

任何布尔函数都能只用 AND、OR 和求补运算写出来，并由此构建任何组合电路，只需图 10-6 中的三个基本门。实际中还有几个其他常用的门，图 10-7 给出了其中的三个。



a	b	x
0	0	1
0	1	1
1	0	1
1	1	0

a) NAND门

a	b	x
0	0	1
0	1	0
1	0	0
1	1	0

b) NOR门

a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

c) XOR门

图 10-7　三类常见的逻辑门

NAND（Not AND，与非）门等价于 AND 门后跟一个反相器，如图 10-8 所示。类似地，NOR（Not OR，与或）门等价于 OR 门后跟一个反相器。在电子学上，构建 NAND 门往往比构建 AND 门容易。实际上经常用 NAND 门后跟反相器来构建 AND 门。NOR 门也要比 OR 门更常用。

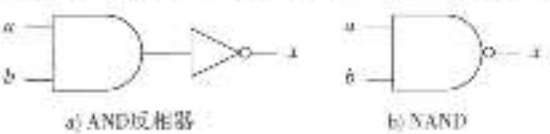


图 10-8　两个等价的组合电路

XOR（exclusive OR）表示异或，由 OR 有时称为包含或（inclusive OR）。如果任一输入或者两个输入都为 1，那么 OR 门的输出为 1。如果只有一个输入为 1，那么 XOR 门的输出为 1；如果两个输入都是 1，那么 XOR 门输出为 0。XOR 运算的代数符号是⊕，$a \oplus b$ 的代数定义是

$$a \oplus b = a \cdot b' + a' \cdot b$$

XOR 运算符的优先级高于 OR 但是低于 AND，如图 10-9 所示。

优先级	运算符
最高	求补
	AND
	XOR
最低	OR

图 10-9　XOR 运算符的优先级

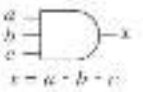
例 10.3　表达式

$a + b \oplus c \cdot d$

完全加上括号得到 $a + (b \oplus (c \cdot d))$。根据 XOR 的定义展开后，表达式变为

$a + b \cdot (c \cdot d)' + b' \cdot (c \cdot d)$ □

也可以使 AND 和 OR 有 2 个以上的输入。图 10-10 展示了一个三输入 AND 门和它的真值表。只有当所有输入均为 1 时，AND 门的输出才为 1。只有当所有输入均为 0 时，OR 门的输出才是 0。



a	b	c	x
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

图 10-10 三输入的 AND 门

10.1.7 其他表达方式

你可能已经看出 AND 门、OR 门和反相器与 C++ 布尔表达中 AND、OR 和 NOT 运算的相似性。它们的真值表是一样的，NOT 对应反相器，C++ 的真值和假值分别对应布尔代数的 1 和 0。

布尔代数的数学结构非常重要，因为它不仅适用于组合电路，而且也适用于语句逻辑。C++ 用语句逻辑来确定包含在 if 和循环语句中的条件的真假。人工智能中最近一组很重要的编程语言甚至更加广泛地使用了语句逻辑，用这些语言编写的程序通过一种称为逻辑编程（logic programming）的技术来模拟人的推理。布尔代数是这个领域的主要组成部分。

布尔代数的另一种表示方式是使用集合运算。如果把一个布尔变量看作一个集合，OR 运算看作集合的并，AND 运算看作集合的交，求补运算看作集合的补，0 看作空集，1 看作全集，那么所有布尔代数的性质和定理也适用于集合。 498 / 499

例 10.4 定理

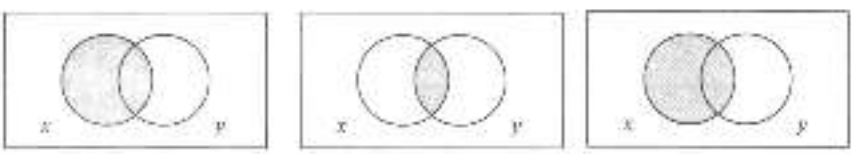
$x + 1 = 1$

说明全集和任何其他集合的并集是全集。 □

例 10.5 图 10-11 展示的是用集合论的文氏图（Venn diagram）来解释吸收律

$x + x \cdot y = x$

图 10-11a 给出的是集合 x，图 10-11b 是 x 和 y 的交集，也就是既在 x 又在 y 中的所有元素的集合。这个集合和 x 的并集如图 10-11c 所示。可以看到图 10-11a 中的区域和图 10-11c 中的区域是相同的，这说明了吸收律。 □



a) x b) $x \cdot y$ c) $x = x \cdot y$

图 10-11 吸收律的集合论诠释

以布尔代数来解释组合电路的描述，以及它作为语句逻辑的基础，表明它是计算机科学中大部分理论和应用的数学基础。它还能描述集合论，也表明了它在其他数学领域里的重要性。

10.2 组合分析

每个布尔表达式有一个对应的逻辑图，每个逻辑图有一个对应的布尔表达式。用数学术语来说，两者之间是一一对应的关系。然而，给定一个真值表会有几种不同的实现方式。图 10-12 展示了一个真值表，有几个对应的布尔表达式和逻辑图。 500

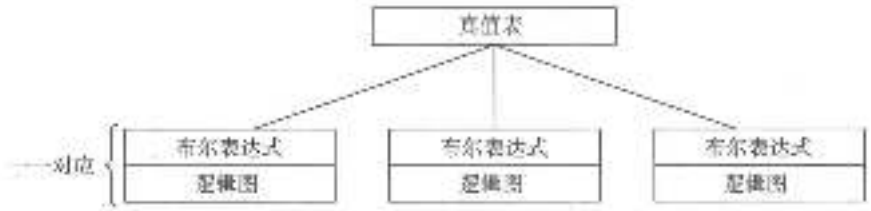


图 10-12　一个给定的真值表可以有多种实现

本节讲述组合电路三种表达方式之间的对应关系。

10.2.1　布尔表达式和逻辑图

布尔表达式由用 AND、OR 和反相运算组合在一起的一个或多个变量组成。电路输入的数量等于变量的数量。本节和下节都专注于单输出电路，本章最后一节会涉及多于一个输出的电路。

画出给定布尔表达式的逻辑图，就是给每个 AND 运算画 AND 门，给每个 OR 运算画 OR 门，给每个求补运算画反相门。根据表达式将一个门的输出连接到另一个门的输入。该组合电路的输出是某一个门的输出，这个门的输出没有连接到另一个门的输入。

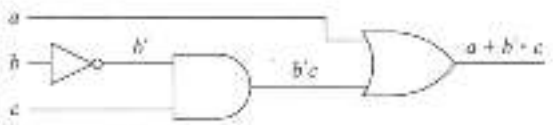


图 10-13　布尔表达式 $a+b' \cdot c$ 的逻辑图

例 10.6　图 10-13 展示了对应布尔表达式 $a + b' \cdot c$ 的逻辑图。从现在开始，我们将忽略 AND 运算的符号，把这个布尔表达式写成

$$a + b'c$$

在每个门的输出上标出它对应的表达式。□

501　当表达式有括号时，必须首先构建括号内的子图。

例 10.7　图 10-14 是三变量表达式

$$((ab + bc')a)'$$

的逻辑图，首先用一个 AND 门形成 ab，用另一个 AND 门形成 bc'。这两个门的输出进行 OR 运算，然后和 a 进行 AND 运算。因为要对整个表达式求补，因此反相器是最后的门。□

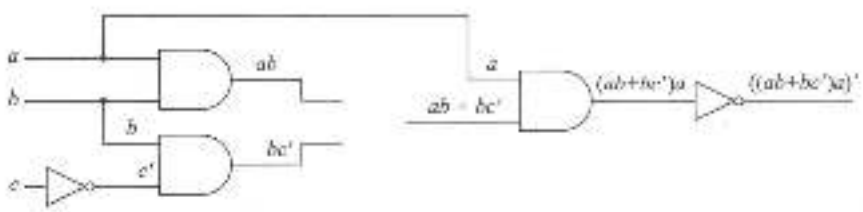


图 10-14　布尔表达式 $((ab + bc')a)'$ 的逻辑图

图 10-14 中的两个小黑点是结合点，在结合点处两条线路物理上是相连接的。回想一下，变量 a 提供的物理信号是电压，当输入 a 来的信号到达结合点时，它不像河里的水在河道里遇到分叉那样，一些水分叉流到一条路径上，另一些水分叉流到另一条路径上。在逻辑图中，不存在一部分信号去一个 AND 门的输入，另一部分信号去另一个 AND 门的输入这种情况，从 a 来的信号完全复制到两个门的输入上。

对于具有一定物理知识的人来说，产生这种现象的原因是电压是电势的度量，导线的电

阻很低，根据欧姆定律，这意味着在导线上的电势改变可以忽略不计，所以沿着任何物理上连接的线路，电压是恒定不变的。因此不管有没有结合点，电压信号在任何点都是一样的。（对不懂物理知识的人来说，这可能是一个学习的动机！）

任何来自变量的信号都能利用结合点进行复制。任何变量的补都能用反相器生成，变量的补同样也能通过结合点复制。在逻辑图中经常忽略不显示变量复制的结合点和变量反相器。也就是可以假设任何变量和它的补都可以作为任何门的输入。

例 10.8　图 10-15 是图 10-14 利用这个假设后的简略版，它也认可 AND 门后面跟反相器等同于 NAND 门。□ 502

这个简略图的缺点是，此网络实际上有三输入这种情形，但没有图 10-14 中表现得明显。

例 10.9　图 10-16 展示的是四输入布尔表达式

$(a'bc \oplus c + a + d)'$

的逻辑图。要注意异或运算符的优先级低于 AND，但是高于 OR。□

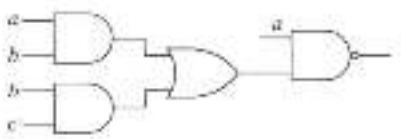


图 10-15　图 10-14 的简化版本



图 10-16　布尔表达式 $(a'bc \oplus c+a+d)'$ 的逻辑图

要根据给定的逻辑图写出布尔表达式，就是简单地把每个门的输出用适当的子表达式标识出来。如果没有给布尔表达式而是给了图 10-16 的逻辑图，那么应该从用 $a'bc$ 标识 AND 门的输出做起。XOR 门的输出应该标识为 $a'bc \oplus c$，它通过 NOR 门生成完整的布尔表达式。

10.2.2　真值表和布尔表达式

根据真值表生成布尔表达式的一种方法是把没有括号的表达式写成几个 AND 项的 OR，每个 AND 项对应真值表中的一个 1。

例 10.10　$a \oplus b$ 的真值表有两个 1，对应的布尔表达式是

$a \oplus b = a'b + ab'$　503

如果 a 为 0、b 为 1，那么第一个 AND 项将会是 1；如果 a 为 1、b 为 0，那么第二个 AND 项将会是 1。不论哪种情况，这两项的 OR 将会是 1。此外，任何其他 a 值和 b 值的组合将使这两个 AND 项都为 0，所以布尔表达式为 0。□

例 10.11　图 10-3 显示当 $abc = 001$ 和 $abc = 011$ 时，x 为 1；abc 为其他组合时，x 为 0。对应的布尔表达式为

$x = a'b'c + a'bc$

当且仅当 $abc = 001$ 时，第一个 AND 项 $a'b'c$ 为 1，当且仅当 $abc = 011$ 时，第二个 AND 项为 1，除了这两个条件以外，这两项的 OR 都将为 0，和真值表一样。□

例 10.12　图 10-4 中 x 的真值表是四个变量的例子，对应的表达式为

$x = a'bc'd + a'bcd + abc'd + abcd$

有 4 种 a、b、c 和 d 的组合会得出真值表里的 1。□

对偶技术能够把表达式写成几个 OR 项的 AND，每个 OR 项对应真值表中的 0。

例 10.13　图 10-17 的表达式是

$x = (a + b' + c')(a' + b' + c)$

如果 $abc = 011$，那么第一个 OR 项为 0；如果 $abc = 110$，那么第二个 OR 项为 0。满足这两个条件中的任意一个，OR 项的 AND 就为 0。abc 的所有其他组合会使两个 OR 项为 1，表达式的值为 1。 □

a	b	c	x
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

图 10-17　三变量真值表

给定一个布尔表达式，构建对应真值表最直接的方式是，计算出变量所有可能组合的表达式的值。

例 10.14　构建

504 $x(a, b) = (a \oplus b)' + a'$

的真值表需要计算

$x(0, 0) = (0 \oplus 0)' + 0' = 1$

$x(0, 1) = (0 \oplus 1)' + 0' = 1$

$x(1, 0) = (1 \oplus 0)' + 1' = 0$

$x(1, 1) = (1 \oplus 1)' + 1' = 1$

这个例子需要计算两个变量 a 和 b 的四种可能组合的值。 □

如果表达式包含多于两个变量，有时候利用布尔代数的属性和定理把布尔表达式转换成几个 AND 项的 OR 要更容易些，由此可以写出真值表。

例 10.15　图 10-16 的表达式简化为

$(a'bc \oplus c + a + d)'$

= 〈⊕的定义〉

$(a'bcc' + (a'bc)'c + a + d)'$

= 〈互补律，$cc' = 0$；零元定理，$x \cdot 0 = 0$〉

$((a'bc)'c + a + d)'$

= 〈德·摩根定律〉

$((a + b' + c')c + a + d)'$

= 〈分配律、互补律和恒等律〉

$(ac + b'c + a + d)'$

= 〈吸收律，$a + ac = a$〉

$(a + b'c + d)'$

= 〈德·摩根定律〉

$a'(b'c)'d'$

= 〈德·摩根定律〉

$a'(b + c')d'$

= 〈分配律〉

$a'bd' + a'c'd'$

真值表有 16 项。检查真值表，在 $abd = 010$（两项）和 $acd = 000$（两项）的地方插入 1，所有其他项是 0，结果如图 10-18 所示。由于 $abd = 010$ 的其中一项也满足 $acd = 000$，因此有 3 个 1 而不是 4 个。

a	b	c	d	x	a	b	c	d	x
0	0	0	0	1	1	0	0	0	0
0	0	0	1	0	1	0	0	1	0
0	0	1	0	0	1	0	1	0	0
0	0	1	1	0	1	0	1	1	0
0	1	0	0	1	1	1	0	0	0
0	1	0	1	0	1	1	0	1	0
0	1	1	0	1	1	1	1	0	0
0	1	1	1	0	1	1	1	1	0

图 10-18　图 10-16 中表达式的真值表

这项技术避免了计算原始表达式 16 次。实际上这个任务可能没有刚开始看上去那么

困难。只要稍微想一下，就能根据 d 为 1 时的原始表达式推论出不管 a、b 和 c 的值是什么，括号内的表达式一定是 1，它的反一定是 0。类似情况，当 a 为 1 时表达式一定为 0。这样就只剩下 $ad = 00$ 的四种情况了。 □ 505

10.2.3　两级电路

每个布尔表达式都可以转换成 AND-OR 表达式，这一事实对组合电路的处理速度有主要的实际影响。当改变门的一个输入信号时，输出不会马上做出响应。信号会通过门的内部电子元件，存在时间上的延迟。门输出对输入改变做出响应要花费的时间称作门延迟（gate delay）。采用不同制造工艺的门具有不同的门延迟。制造延迟短的门要更贵一些，这种门比延迟长的门需要更多的电能来运行。虽然不同设备技术导致门延迟差异很大，但典型的门延迟是 2ns（nanosecond，纳秒）。

十亿分之一秒对于等待输出好像不是一段很长的时间，但是如果一个电路有一长串的门，必须循环进行处理，那么这段时间就有可能很长了。信号以近似 3.0×10^8m/s 的光速通过线路，也就是 1ns 通过 30cm。典型门延迟 2ns 内，信号能通过 60cm 的线路。相比于集成电路或电路板的尺寸，这是相当长的距离。所以实际上门延迟决定了网络处理速度。

例 10.16　思考图 10-16 的电路，如果门延迟是 2ns，b 的一个变化传过 AND 门需要 2ns，传过 XOR 门需要 2ns，传过 NOR 门需要 2ns，即需要 6ns 的传输时间（忽略经过反相器的传输延迟）。

现在考虑用布尔代数把这个电路写成 AND-OR 形式的表达式：

$$\begin{aligned} x &= (a'bc \oplus c + a + d)' \\ &= a'bd' + a'c'd' \end{aligned}$$

图 10-19 展示的是相应的电路。因为输入的改变只需要 2 个门延迟就传送到输出，因此叫作两级电路。 □

将处理时间从 6ns 缩短到 4ns，意味着 33% 的速度提升，效果非常显著。因为任何布尔表达式都能转换成 AND-OR 表达式，而 AND-OR 表达式又对应两级 AND-OR 电路，所以任何函数都可以用最多有两个门延迟处理时间的组合电路来实现。

同样的理论也适用于对偶原则。总是可以把布尔表达式转换成 OR-AND 表达式，OR-AND 表达式对应两级电路，这样的两级电路最多有两个门延迟的处理时间。要得到布尔表达式的 OR-AND 表达式，可以首先得到 AND-OR 表达式的补，然后使用德·摩根定律即可。

例 10.17　图 10-20 是图 10-17 的表达式

$$x = (a + b' + c')(a' + b' + c)$$

的两级 OR-AND 电路。回想一下，每个 OR 项对应真值表中的一个 0。 □

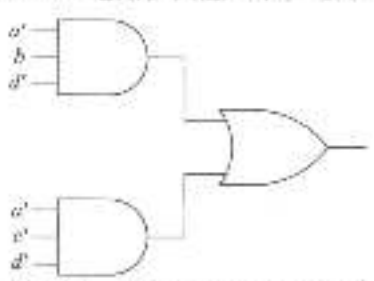


图 10-19　等价于图 10-16 中电路的两级 AND-OR 电路

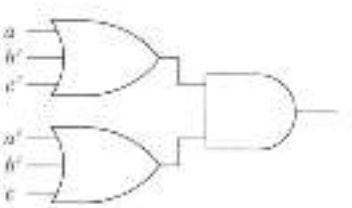


图 10-20　图 10-17 对应的两级 OR-AND 电路

例 10.18　图 10-13 的表达式是

$$x = a + b'c$$

要把这个表达式转换成 OR-AND 表达式，首先写出它的补

$$\begin{aligned} x' &= (a + b'c)' \\ &= a'(b'c)' \\ &= a'(b + c') \\ &= a'b + a'c' \end{aligned}$$

这是一个 AND-OR 表达式。再使用德·摩根定律把 x 写成

$$\begin{aligned} x &= (x')' \\ &= (a'b + a'c')' \\ &= (a'b)'(a'c')' \\ &= (a + b')(a + c) \end{aligned}$$

这就是一个 OR-AND 表达式了。 □

506 / 507

通常三级或更多级的电路比等价的两级电路所需的门更少。因为门占用集成电路的物理空间，所以尽管两级电路能达到更快的处理速度，但代价是为更多的门提供额外的空间。

这也是计算机科学中一个空间/时间折中的例子。值得注意的是从高抽象层级的软件到低抽象层级的硬件，都有这样的空间/时间问题，这确实是一个基本问题。

10.2.4　无所不在的 NAND

表达式 $(abc)'$ 表示一个三输入 NAND 门，德·摩根定律表述为

$$(abc)' = a' + b' + c'$$

可以把第二个表达式想象成一个 OR 门的输出，这个 OR 门在执行 OR 运算前把每个输入反相。逻辑图偶尔把 NAND 门画成反相输入的 NOR，如图 10-21a 所示。

从对偶表达式得出

$$(a + b + c)' = a'b'c'$$

NOR 门等价于反相输入的 AND 门，如图 10-21b 所示。

将这个理念进一步运用到两级电路。考虑

$$abc + def = [(abc)'(def)']'$$

的等价表达式，这也可以从德·摩根定律得出。第一个表达式表示一个两级 AND-OR 电路，而第二个表示的是一个两级的 NAND-NAND 电路。图 10-22 展示的是这些等价的电路。

图 10-22a 展示的是一个 AND-OR 电路，它有两个 AND 门和一个 OR 门。也可以完全用 NAND 门生成等价电路，如图 10-22b 所示。除了最后的 NAND 画成了反相输入的 OR 之外，图 10-22c 和图 10-22b 是一样的。这种画法可以明显看出 AND 运算后面的求补和 OR 运算前面的求补抵消了，门符号变成了和 AND-OR 电路中类似的形状，这有助于表达电路的含义。

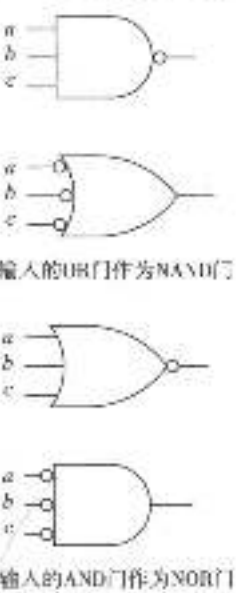
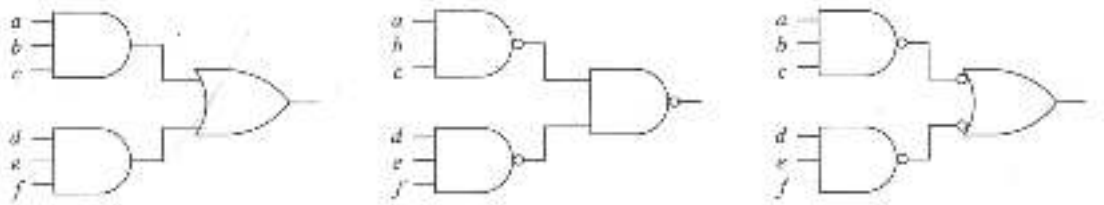


图 10-21　等价的门



a) 一个AND-OR电路　b) 等价的NAND-NAND电路　c) 与b)相同的NAND-NAND电路

图 10-22　一个 AND-OR 电路及其等价的 NAND-NAND 电路

不仅可以用 NAND 门来完全替换任意的 AND-OR 电路，而且可以通过把 NAND 输入连接到一起，由 NAND 门得到反相器，如图 10-23 所示。因为 NAND 用输入 a 和 b 生成 $(ab)'$，如果指定 $b = a$，那么门将生成 $(a \cdot a)' = a'$，即 a 的补。

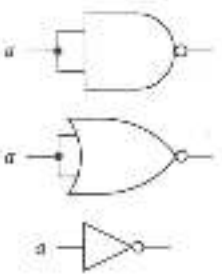


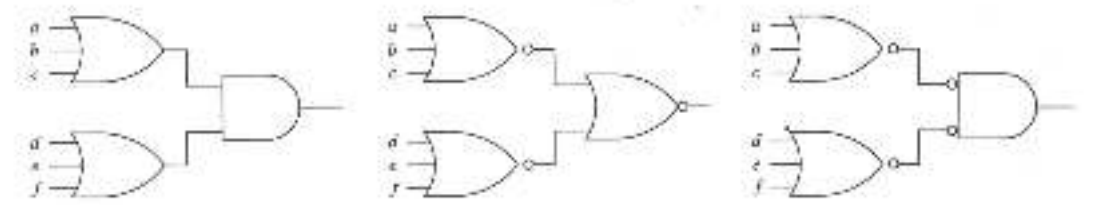
图 10-23　三个等价的电路

从理论上来说，可以仅用 NAND 门构建任何组合电路。此外，NAND 门通常比 AND 门和 OR 门更容易制造，因此 NAND 门是目前集成电路中最常用的门。

当然，同样的原理也适用于对偶电路。两级电路的德·摩根定律是

$$(a + b + c)(d + e + f) = [(a + b + c)' + (d + e + f)']'$$

这说明 OR-AND 电路等价于 NOR-NOR 电路。图 10-24 是图 10-22 的对偶电路。



a) 一个OR-AND电路　b) 等价的NOR-NOR电路　c) 与b)相同的NOR-NOR电路

图 10-24　一个 OR-AND 电路及其等价的 NOR-NOR 电路

适用于 NAND 电路的推理同样适用于 NOR 电路。任何组合电路都可以写成一个两级 OR-AND 电路，这个两级电路可以写成 NOR-NOR 电路。连接 NOR 的输入可以得到反相器。理论上仅用 NOR 门就可以构建任何组合电路。

10.3　组合设计

两级电路的高速是其优于两级以上门电路的优势。有时候，在两级电路中减少门的数量并保持两级门延迟的处理速度是可行的。

例 10.19　布尔表达式

$$x(a, b, c, d) = a'bd' + a'c'd' + a'bc'd'$$

可以用吸收率简化为

$$\begin{aligned} x(a, b, c, d) &= a'bd' + (a'c'd') + (a'c'd')b \\ &= a'bd' + a'c'd' \end{aligned}$$

这个表达式也对应于两级电路，但是它只需要两个三输入 AND 门和一个两输入 OR 门，而

原始表达式对应的电路需要三个 AND 门和一个三输入 OR 门，其中一个 AND 门还需要四输入。□

用布尔代数来简化两级电路中门的数量并不总是简单明了的。本节给出一种系统化的方法来设计具有 3 个或 4 个变量且门数量尽可能少的两级门电路。

10.3.1　范式

上一节讲述了任何布尔表达式都能转换成一个两级 AND-OR 表达式。要简化两级电路，首先要使每个 AND 项只包含所有输入变量一次，这样的 AND 项称为极小项（minterm）。AND-OR 表达式总是可以转换成极小项的 OR。

例 10.20　考虑布尔表达式

$$x(a, b, c) = abc + a'bc + ab$$

因为前两个 AND 项包含所有三个变量，因此它们是极小项，但最后一个不是。转换过程如下

$$\begin{aligned} x(a, b, c) &= abc + a'bc + ab \\ &= abc + a'bc + ab(c + c') \\ &= abc + a'bc + abc + abc' \\ &= abc + a'bc + abc' \end{aligned}$$

508
|
510

最后的表达式称为范式（canonical expression），因为它是没有重复的极小项的 OR。□

范式中的每个极小项代表真值表里的一个 1。所以范式直接和真值表相关。范式及其对应真值表的方便的简化表示法叫作西格玛表示法（sigma notation）。它由大写希腊字母西格玛（Σ）后面跟一组十进制数字组成，这些数字对应真值表中包含 1 的行，大写的西格玛代表 OR 运算。不言而喻，没有列出的行都是包含 0 的行。

例 10.21　在例 10.20 中，因为 x 的范式有 3 个极小项，因此它的真值表有 3 个 1。图 10-25 展示的是这个函数的真值表，每一行都标记有一个十进制数，对应于二进制数 abc。这个函数对应的西格玛表示为

$$x(a, b, c) = \Sigma(3, 6, 7)$$

因为第 3、6 和 7 行含有 1。□

行 (dec)	a	b	c	x
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

图 10-25　一个范式的真值表

对偶范式是一个 OR-AND 表达式，它的每一项只包含所有变量一次，没有重复的 OR 项。这个范式相应的表示法包含的是真值表中 0 组成的列表。这里使用大写希腊字母派（Π）表示 AND 运算，而不是西格玛。

例 10.22　前面例子的对偶范式为

$$x(a, b, c) = (a + b + c)(a + b + c')(a + b' + c)(a' + b + c)(a' + b + c')$$

用派表示法可以写成

$$x(a, b, c) = \Pi(0, 1, 2, 4, 5)$$

因为这 5 行在真值表里为 0。□

例 10.23　使用西格玛表示法，图 10-3 中 x 和 y 为

$$x(a, b, c) = \Sigma(1, 3)$$
$$y(a, b, c) = \Sigma(3, 4)$$

图 10-4 中函数 x 和 y 为

$$x(a, b, c, d) = \Sigma(5, 7, 13, 15)$$
$$y(a, b, c, d) = \Sigma(4, 5, 12, 13)$$

511

西格玛和派表示法要比布尔范式和真值表更简洁。本节剩下的部分假设要简化的函数已经转换为它的唯一范式，或者说已经给定或确定了真值表。

Ada Byron，洛夫莱斯伯爵夫人

Ada Byron，1815 年 12 月 10 日生于伦敦。她是浪漫主义诗人拜伦勋爵（Lord Byron）的女儿，但是她并不认识拜伦，因为 Ada 方出生一个月，拜伦就离开了她的母亲。母亲为了抵消她的父亲危险的诗歌基因，决意让她学习音乐和数学。

17 岁时，Ada 遇到了剑桥的数学教授 Charles Babbage，开始了一段长期的交往，两人之间有大量主题范围广泛的通信，包括数学和逻辑。Ada 在 1935 年与 William King 结婚，生了 3 个孩子。King 继承了贵族头衔，他们两个成为洛夫莱斯伯爵和伯爵夫人。

Babbage 主要因其在计算机器建造原理方面超越时代的远见而闻名。他提出两类机器：用来计算数学表的差分机（difference engine）和功能更强大的分析机（analytical engine）。机器的齿轮的卡控制，能够进行通用的计算。与逻辑门处理数字位 1 和 0 的方式相同，Babbage 的机器同样上有轮和无轮来表示一个位的值。不过，经济和体制上的困难使得他没能建造出那些较大的机器

1841 年秋天，Babbage 在意大利的一个学术讨论会上报告了他的分析机，Luigi Menabrea 以法语发表了一篇题为“Charles Babbage 创造的分析机概述”的论文，Ada 把它翻译成了英语。当她向 Babbage 展示译文时，他建议她可以加上自己的注释。Ada 的注释是原来的文章的三倍，她也因此成名。



Crown 版权所有；UK 政府艺术收藏

Ada 谨慎地控制分析机以生成重要的数学结果，计算贝努利数（Bernoulli number）就是一个例子。贝努利数是数论和数学分析中的重要概念，Bernoulli 本人计算出了前十个贝努利数，数学家 Euler 算到了第 30 个，后来 Martin Ohm 在 1840 年算到了第 62 个。这些人都是用手工计算的，而 Ada 在 Menabrea 的分析机论文中写了一个程序，能够控制分析机来自动计算贝努利数。

Ada Byron 被认为是世界上第一个程序员。她预见机器能够生成图形图像甚至音乐。1979 年，一种编程语言被命名为 Ada，并被广泛使用。Ada 于 1852 年 11 月 27 日逝世，年仅 37 岁，被迫在她并不认识的父亲身边。

10.3.2 三变量卡诺图

两级电路的简化是基于距离的概念。两个极小项之间的距离（distance）是它们不同之处的数量。

例 10.24 来看一下这个三变量函数的范式：

$$x(a, b, c) = a'bc + abc + abc'$$

极小项 $a'bc$ 和 abc 之间的距离是 1，这是因为 a' 和 a 是这两个极小项唯一不同的变量，而两者中变量 b 和 c 是相同的。极小项 $a'bc$ 和 abc' 的距离是 2，这是因为 $a'bc$ 中的 a' 和 c 与 abc' 中的 a 和 c' 不相同。 □

识别相邻极小项（adjacent minterm），即距离为1的极小项，是简化AND-OR表达式的关键。一旦找到两个相邻极小项，就可以用分配律提出公共项，再用取补和幂等性简化它。

例10.25　可以像下面那样合并前两个极小项来简化例10.24中的表达式：

$$\begin{aligned} x(a,b,c) &= a'bc + abc + abc' \\ &= (a' + a)bc + abc' \\ &= bc + abc' \end{aligned}$$

或者可以合并第二和第三个极小项，因为它们也是相邻的。

$$\begin{aligned} x(a,b,c) &= a'bc + abc + abc' \\ &= a'bc + ab(c + c') \\ &= a'bc + ab \end{aligned}$$

无论哪种方式都改进了电路。原表达式是一个有三个三输入AND门和一个三输入OR门的电路。两个简化表达式对应的电路都仅有两个AND门，其中之一仅有两个输入，以及一个仅有两输入的OR门。

识别相邻极小项是很容易的。为了得到更小的最终电路，有时候会临时使表达式更加复杂。当一个极小项和其他两个极小项相邻时就会发生这样的情况，可以用幂等性复制这个极小项，再把它和它的两个相邻极小项合并。□

例10.26　在例10.25中可以首先用幂等性复制abc，然后再和另两个极小项合并。

$$\begin{aligned} x(a,b,c) &= a'bc + abc + abc' \\ &= a'bc + abc + abc - abc' \\ &= (a' + a)bc + ab(c + c') \\ &= bc + ab \end{aligned}$$

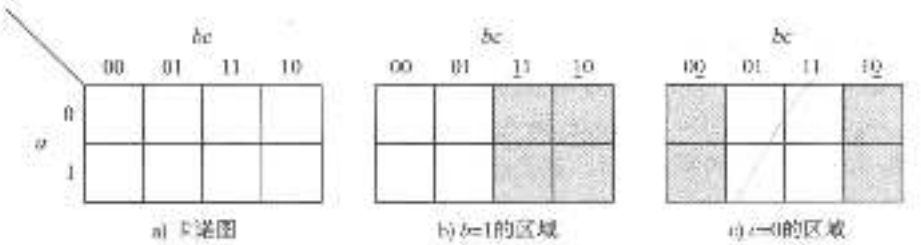
512
⋮
513

这个结果要好于例10.25中的结果，因为两个AND门都只有两个输入。□

用布尔代数进行简化枯燥且容易出错。卡诺图是一个简化两级电路的工具，它能很容易地找出相邻极小项，确定需要用幂等性复制哪些项。卡诺图只是一个组织过的真值表，相邻的条目代表只有一个不同之处的极小项。

图10-26a展示的是一个三变量卡诺图。左上单元代表$abc = 0$，它的右边是代表$abc = 001$的单元，再往右是代表$abc = 011$的单元，接着是$abc = 010$。序列000、001、010、011保证了相邻单元只有1个不同。如果按照数字顺序排列单元000、001、010、011就不会有这种效果了，因为001和010的距离是2。

图中最上面的一行包含了真值表中$a = 0$的条目，最下面一行包含了$a = 1$的条目。每一列是bc的值，例如，第一列是$bc = 00$，第二列是$bc = 01$，最左边两列是$b = 0$，最右边两列是$b = 1$，如图10-26b所示。外边的两列是$c = 0$，中间的两列是$c = 1$，如图10-26c所示。



a) 卡诺图　　b) $b=1$的区域　　c) $c=0$的区域

图10-26　一个三变量函数的卡诺图

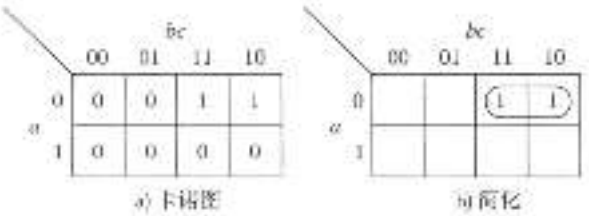
用布尔代数提取出相邻极小项的公共项相当于在卡诺图上对相邻单元分组。在对单元分组之后，通过检查卡诺图的区域可以写出简化项。

例 10.27 图 10-27a 展示的是范式

$$x(a, b, c) = a'bc + a'bc'$$

的卡诺图。

$abc = 011$ 对应单元中的 1 是极小项 $a'bc$ 的真值表单元。图 10-27b 是同样的卡诺图，只是为了清晰省略了 0。因为这两个 1 是相邻的，所以可以用椭圆把它们圈起来。椭圆覆盖的单元是 $a = 0$ 的行和 $b = 1$ 的列，因此是 $ab = 01$ 的区域，这对应于 $a'b$ 项，因此 $x(a, b, c) = a'b$ 可以不用布尔代数，仅通过观察就写出结果。



a) 卡诺图　　b) 简化

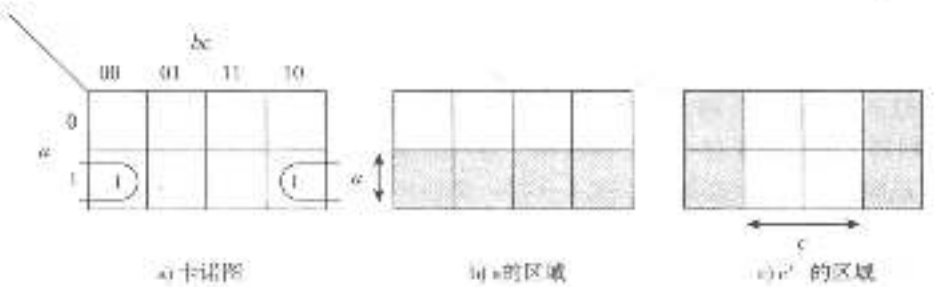
图 10-27 例 10.27 的 AND-OR 表达式的卡诺图

□

例 10.28 图 10-28a 展示的是范式表达式

$$x(a, b, c) = ab'c' + abc'$$

的卡诺图。



a) 卡诺图　　b) a 的区域　　c) c' 的区域

图 10-28 例 10.28 的 AND-OR 表达式的卡诺图

看上去 $ab'c'$ 单元在左下角，abc' 单元在右下角，两者是不相邻的，但是实际上它们是相邻的。应该把卡诺图想象成环绕的，因此它的左边和右边是相邻的，这就是所谓的 Pac-Man 效应。在图中将一个椭圆画成了两个开放的半椭圆来表达卡诺图的这个属性。

这两个单元都位于 $a = 1$ 行和 $c = 0$ 列，如图 10-28b 和图 10-28c 所示。可以把两个单元想象成图中阴影区域的交集，这个区域是 $ac = 10$。因此简化的函数是 $x(a, b, c) = ac'$。 □

514
~
515

用幂等属性复制一个极小项，这样它可以和另两个极小项合并，对应于卡诺图中两个椭圆的重叠。如果 AND-OR 表达式中不止两个极小项，那么就可以随意把真值表中的 1 与多个组合使用。

例 10.29 图 10-29 展示的是

$$x(a, b, c) = a'bc + abc + abc'$$

的卡诺图，这是例 10.26 的规范表达式。图 10-29a 展示的是合并第一和第二极小项的简化，图 10-29b 展示的是合并第二和第三极小项的简化，图 10-29c 中两个合并都用到了第二项，对应于两个椭圆的重叠。 □

当原始的真值表采用西格玛表示法时，可以在卡诺图中 1 的位置插入图 10-30 所示的十进制标号。

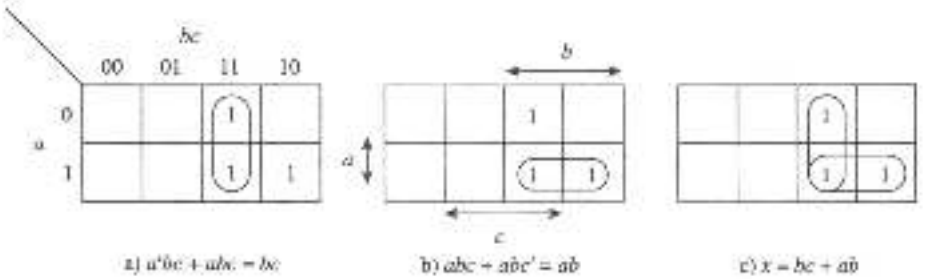


图 10-29　例 10.26 的 AND-OR 表达式的卡诺图

简化的过程就是要确定最好的一组椭圆来覆盖卡诺图中所有的 1。“最好”的意思是这组椭圆对应于一个两级电路，它有最少的门，每个门有最少的输入。椭圆的数量等于 AND 门的数量，一个椭圆覆盖的 1 越多，对应的 AND 门的输入就越少。所以要得到最少的椭圆数，每个椭圆要尽可能多地覆盖所有的 1 而不覆盖 0。允许 1 被多个椭圆覆盖。下面几个例子展示了确定椭圆的一般策略。

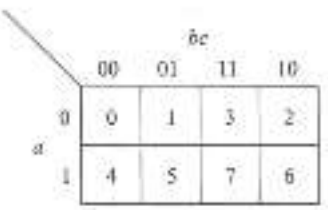


图 10-30　卡诺图中最小项的十进制标号

例 10.30　图 10-31 展示的是一个简化时常见的错误。要简化

$$x(a, b, c) = \Sigma(0, 1, 5, 7)$$

可能首先想合并极小项 1 和 5，如图 10-31a 所示。一开始就这样做并不好，因为极小项 1 与 0 和 5 都相邻，极小项 5 与 1 和 7 都相邻。另一方面，极小项 0 只和 1 相邻。要用尽可能大的椭圆覆盖极小项 0，就必须将它和 1 合并。类似地，极小项 7 只和 5 相邻，要用尽可能大的椭圆覆盖它，就必须将它和 5 合并。

图 10-31b 展示的是这种极小项分组的结果，代表表达式

$$\begin{aligned} x(a, b, c) &= \Sigma(0, 1, 5, 7) \\ &= a'b' + b'c + ac \end{aligned}$$

这需要 3 个两输入 AND 门和 1 个三输入 OR 门。但第一个分组选择并不是必需的。图 10-31c 给出了正确的简化，它表示

$$\begin{aligned} x(a, b, c) &= \Sigma(0, 1, 5, 7) \\ &= a'b' + ac \end{aligned}$$

这个实现只需要两个两输入 AND 门和一个两输入 OR 门。 □

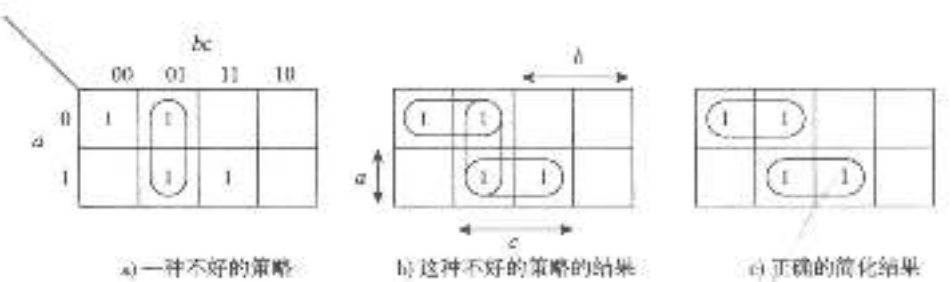


图 10-31　一开始选择不好的结果

前面的例子告诉我们一个经验，即从只有一个邻居的极小项开始分组，因为无论如何邻居必须和它们一起分组，因此可以节省一个不必要的邻居组成的分组。

另一个常见的错误是无法找到一个大的 1 的分组，如例 10.31 所示。

例 10.31　图 10-32a 展示的是一个三变量函数的简化

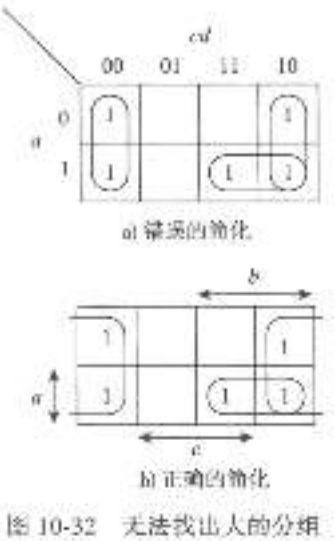
$$x(a, b, c) = \Sigma(0, 2, 4, 6, 7)$$
$$= b'c' + bc' + ab$$

它需要三个两输入 AND 门和一个三输入 OR 门。图 10-32b 是正确的简化，即

$$x(a, b, c) = c' + ab$$

这只需要一个两输入 AND 门和一个两输入 OR 门。　□

在三变量问题中，四个 1 的分组相当于只有一个变量的 AND 项。因为一个组中 1 的个数一定是对应于 a、b、c 以及它们的补的区域的交集，因此组中 1 的数量一定是 2 的幂。例如，一个卡诺图可以包含 1、2 或 4 个 1，但是绝不会是 3 或 5 个 1。



a) 错误的简化　　b) 正确的简化

图 10-32　无法找出大的分组

10.3.3　四变量卡诺图

四变量电路的简化遵循和三变量电路一样的步骤，除了卡诺图中条目的数量是三变量的两倍以外。图 10-33a 是单元的排列。不仅极小项 0 和 2 相邻、4 和 6 相邻，而且极小项 12 和 14、8 和 10 也相邻。同时，最上面一行的单元和最下面一行对应的单元也相邻，即极小项 0 和 8、1 和 9、3 和 11、2 和 10 相邻。

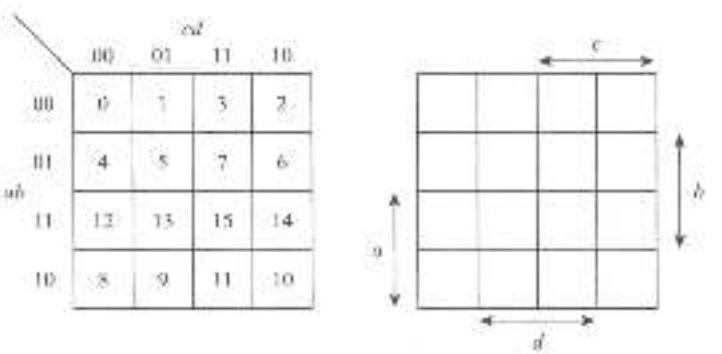
二变量卡诺图中每个单元有三个相邻单元。在四变量卡诺图中，每个单元有四个相邻单元。例如，与 10 相邻的是 2、8、11 和 14，与 4 相邻的是 0、5、6 和 12。

图 10-33b 展示的是变量为 1 的真值表区域。变量 a 在下面两行为 1，变量 b 在中间两行为 1，变量 c 在右边两列为 1，变量 d 在中间两列为 1。

例 10.32　图 10-34 展示的是对

$$x(a, b, c, d) = \Sigma(0, 1, 2, 5, 8, 9, 10, 13)$$
$$= c'd + b'd'$$

的简化。注意两个角上的单元可以被组合成 $b'd'$，卡诺图的第二列代表 $c'd$。　□



a) 卡诺图中最小项的十进制标号　　b) 变量为 1 的区域

图 10-33　四变量项数的卡诺图

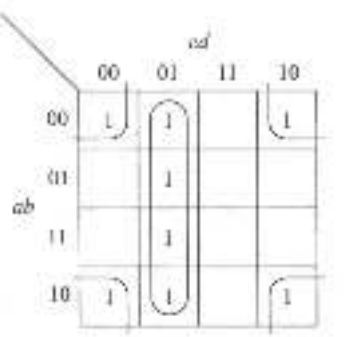


图 10-34　简化一个四变量函数

例 10.33　图 10-35 展示的是

$$x(a, b, c, d) = \Sigma(0, 1, 2, 5, 8, 9, 10,)$$
$$= a'c'd + b'c' + b'd'$$

的简化。和例 10.32 相比，尽管只少了一项，但是简化结果是完全不同的。

极小项 5 只有一个相邻的 1，因此根据经验法则首先将它和 1 组合，这个组合的 AND 项是 $a'c'd$，可以通过观察顶部两行（a'）、左边两列（c'）和中间两列（d）的交集来确定。

用最大的椭圆覆盖极小项 9 需要把它与 0、1 和 8 组合，而不是只与 8 组合，这个组合的 AND 项是 $b'c'$，可以通过观察顶部和底部行（b'）与左边两列（c'）的交集来确定。

剩下还没有被覆盖的 1 是极小项 2 和 10，和前面一样，它们与 0 和 8 组合。 □

例 10.34　图 10-36 展示的简化结果可能不是唯一的。下面这个函数有两个合法的简化结果

$$x(a, b, c, d) = \Sigma(0, 4, 7, 8, 12, 13, 15)$$
$$= c'd' + bcd + abc'$$
$$= c'd' + bcd + abd$$

首先应该组合的极小项是 7，因为它只有一个相邻项 1。极小项 0 必须和 4、8 和 12 组合，因为它没有其他可能的组合方式。这样就只剩下 13 了，它可以和 12 或者 15 组合。 □

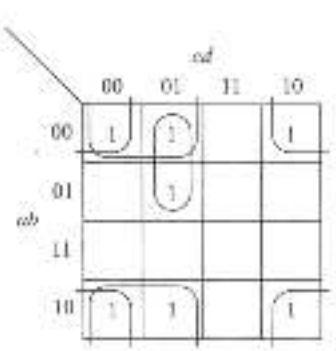
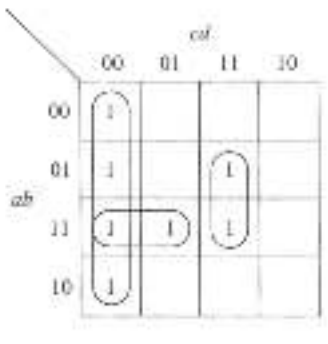
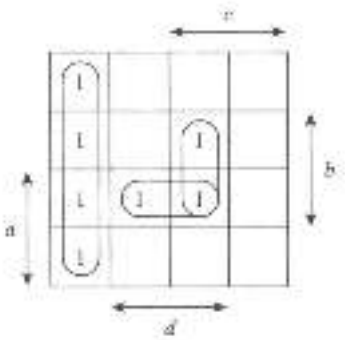


图 10-35　比图 10-34 少一个最小项的表达式



a）一种可能的简化



b）另一种不同的简化

图 10-36　同种正确的不同简化

简化四变量函数并不总是简单明了的，有时候为了确定真正最小的结果，必须尝试几种不同的组合。

例 10.35　图 10-37 展示的就是这样的问题。该函数为

$$\Sigma(0, 1, 2, 3, 5, 6, 7, 8, 9, 12, 13, 14)$$

图 10-37a 是下面推理的结果。来看极小项 12，它所属的最大组是对应 ac' 的四项组，类似的极小项 6 所属的最大组合是四项组 $a'c$。给定这两个组合，可以把极小项 5 组合到 $c'd$ 中，极小项 0 组合到 $a'b'$ 中，极小项 14 组合到 bcd' 中。表达式

$$ac' + a'c + c'd + a'b' + bcd'$$

似乎是合理的，因为没有哪个组合看上去是冗余的，去掉任何一个椭圆都会有 1 没被覆盖到。

给定前两步的选择，余下的三个组合可能都是最好的选择。问题就在于第二步的选择。

图 10-37b 是下面推理的结果。和前面一样，极小项 12 在组合 ac' 中。现在来考虑极小项 14，它必须和 12 或者 6 组合。因为 12 已经被覆盖了，所以 14 和 6 组合。把剩下的 0、1、

2、3、5、7 组合到一起是最有效的，如图 10-37b 所示。得到的表达式

$$ac' + a'd = a'b' + bcd'$$

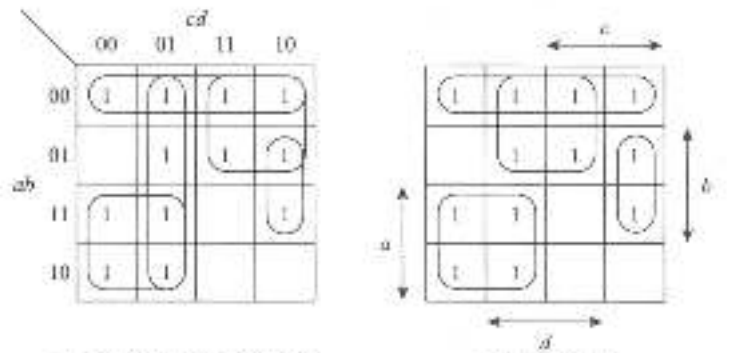
比图 10-37a 少一个 AND 门。

这是一个麻烦的问题，因为通常应该用最大可能的组合来覆盖 1。然而通用的规则不适用于这个问题。一旦确定了组 $a'c$，就不该把极小项 6 放进最大的可能的组里了。

通过图 10-38 可以看到这个问题的解决方案不是唯一的。它首先把极小项 6 组合在 $a'c$ 中，接着把 14 和 12 组合，结果为

$$a'c + b'c' + c'd + abd'$$

□



a) 一种可行但是不好的简化　　b) 正确的简化

图 10-37　一个复杂的简化问题

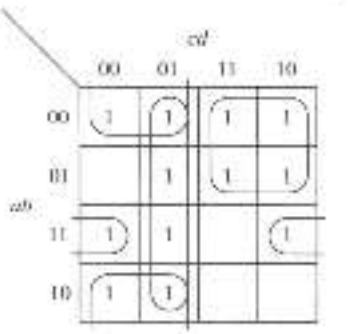


图 10-38　图 10-37 的函数的另一种正确的简化

在面对一个复杂的卡诺图时，如何才能知道该怎样组合极小项呢？其实就是需要多练习、推理和一点点试验。

10.3.4　对偶卡诺图

为了简化 OR-AND 表达式函数，可以简化 AND-OR 表达式形式的函数的补，使用德·摩根定律。

例 10.36　图 10-39 展示的是对图 10-29 中函数的补的简化。原函数是

$$x(a, b, c) = \Sigma(3, 6, 7)$$
$$= \Pi(0, 1, 2, 4, 5)$$

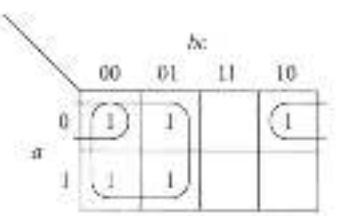


图 10-39　图 10-29 的函数的补

它的补如图所示简化为

$$x'(a, b, c) = \Sigma(0, 1, 2, 4, 5)$$
$$= b' + a'c'$$

简化后的 OR-AND 表达式的原函数是

$$x(a, b, c) = (x'(a, b, c))'$$
$$= (b' + a'c')'$$
$$= b(a + c)$$

简化的 AND-OR 表达式

$$x(a, b, c) = bc + ab$$

需要 3 个门，而这个表达式只需要两个门。

□

在上面的例子中，用一个两级 NOR-NOR 电路而不是 NAND-NAND 电路来实现函数是很划算的。通常情况下，必须简化两种格式来确定哪个需要更少的门。

10.3.5　无关条件

有时候组合电路设计只是为了处理某些输入的组合，其他组合永远不会出现在输入中。即使这些条件出现我们也不关心输出是什么，所以这些组合称为无关条件（don't-care condition）。

在简化过程中，无关条件会带来额外的灵活性。当存在无关条件时，可以随意设计电路产生 0 或 1。通过有选择地让一些无关条件生成 1，另一些生成 0，这样可以改进简化结果。

例 10.37　图 10-40a 是不用无关条件进行简化

$$x(a,b,c) = \Sigma(2,4,6)$$
$$= bc' + ac'$$

现在假设不用极小项 0 和 7 生成 0，在这个问题中，这两个极小项可以生成 0 也可以生成 1。这个描述用符号表示为

$$x(a,b,c) = \Sigma(2,4,6) + d(0,7)$$

这里在极小项标号前面的 d 代表无关条件。图 10-40b 的卡诺图中标记为“×”的单元就表示无关条件。

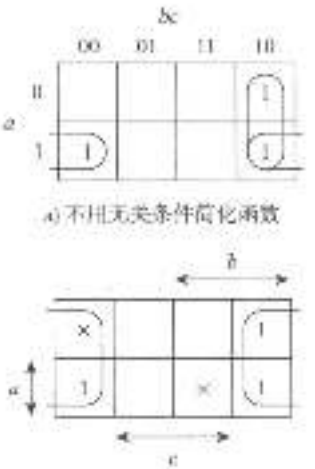


图 10-40　无关条件

当用无关条件进行简化时，可以自由地覆盖或者不覆盖标记为“×”的单元。“×”就像通配符，把它看作 0 或 1 均可。在本例中，如果把极小项 0 看作 1，7 看作 0，简化的结果就是

$$x(a,b,c) = \Sigma(2,4,6) + d(0,7)$$
$$= \Sigma(0,2,4,6)$$
$$= c'$$

不用无关条件，这个函数需要两个 AND 门和一个 OR 门，如果使用无关条件，就不需要 AND 门或者 OR 门了。□

10.4　组合设备

本节讲述一些在计算机设计中普遍用到的组合设备。每个设备都可以被描述成一个黑盒子，有一个对应的真值表定义了输出和输入的关系。本节的所有设备都是组合，可以用两级 AND-OR 电路来实现。这里展示的一些实现用处理时间来换取更小的空间，即更少的门，可能不止两级。

10.4.1　视角

516 ~ 523

下面的几个设备有一条叫作使能（enable）的输入线。使能线就像设备的开关，如果使能线为 0，不管输入线的值是什么，输出线都是 0。此时，设备为关闭或禁用状态。如果使能线为 1，根据描述这个设备的函数，输出线由输入决定。此时，设备为打开或使能状态。

AND 门可以实现使能属性，如图 10-41a 所示。线 a 是这个组合电路（没有在图中显示）的输入之一，是一个 AND 门的输入。该 AND 门的另一个输入是使能线。

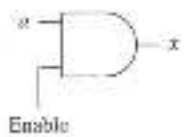
当使能线为 1 时，

$$\begin{aligned} x &= a \cdot (\text{enable}) \\ &= a \cdot 1 \\ &= a \end{aligned}$$

输出等于输入，如图 10-41b 所示。当使能线为 0 时，不管输入是什么，

$$\begin{aligned} x &= a \cdot (\text{enable}) \\ &= a \cdot 0 \\ &= 0 \end{aligned}$$

如图 10-41c 所示。



a) 使能门的逻辑图

Enable = 1	
a	x
0	0
1	1

b) 设备开启时的真值表

Enable = 0	
a	x
0	0
1	0

c) 设备关闭时的真值表

图 10-41 AND 的输入作为使能

实现使能属性不需要一个新的“使能门”，只需用一个不同的视角来看待我们熟悉的 AND 门。可以把输入 a 看作数据线，把使能线看作控制线。使能通过让数据完全不变地通过门或者阻止它通过门来控制数据。

另一个很有用的门是选择反相器（selective inverter）。输入有一条数据线和一条反相线。如果反相线为 1，那么输出是数据线的补；如果反相线为 0，输入不改变地通过门到达输出。

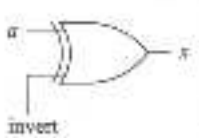
从图 10-42a 可以看到选择反相器是一个 XOR 门，只不过换了一个和以往不一样的视角来看。当反相线为 1 时，

$$\begin{aligned} x &= a \oplus (\text{invert}) \\ &= a' \cdot (\text{invert}) + a \cdot (\text{invert})' \\ &= a' \cdot 1 + a \cdot 1' \\ &= a' \end{aligned}$$

输出等于数据输入的补，如图 10-42b 所示。当反相线为 0 时，

$$\begin{aligned} x &= a \oplus (\text{invert}) \\ &= a' \cdot (\text{invert}) + a \cdot (\text{invert})' \\ &= a' \cdot 0 + a \cdot 0' \\ &= a \end{aligned}$$

数据线不改变地通过这个门。



a) 选择反相器的逻辑图

invert = 1	
a	x
0	1
1	0

b) 反相器开启时的真值表

invert = 0	
a	x
0	0
1	1

c) 反相器关闭时的真值表

图 10-42 XOR 的输入作为反相选择

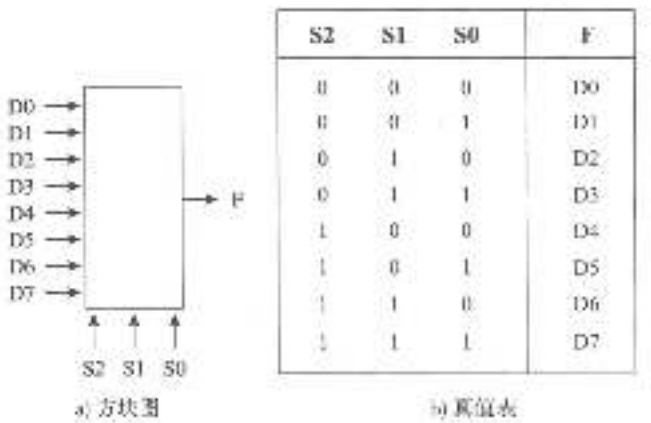
10.4.2　复用器

复用器（multiplexer）是从几个数据输入中选择一个并送到唯一的数据输出的设备。控制线决定要让哪个数据输入通过。

图 10-43a 展示的是一个八输入复用器的方块图。D0 ~ D7 是数据输入线，S2 ~ S0 是选择控制线，F 是唯一的输出线。

由于这个设备有 11 个输入，所以完整的真值表需要 $2^{11}=2048$ 个条目。图 10-43b 是一个简略的真值表。第二个条目显示当选择线是 001 时，输出是 D1。也就是不管 D0 和 D2 ~ D7 为何值，如果 D1 为 1，F 就为 1，如果 D1 为 0，F 就为 0。

因为 n 条选择线可以选择 2^n 条数据线之一，所以复用器的数据输入的数量是 2 的幂。图 10-44 展示了一个四输入复用器的实现，它包含四条数据线 D0 ~ D3，和两条选择线 S1 和 S0。



a) 方块图　　b) 真值表

图 10-43　八输入复用器

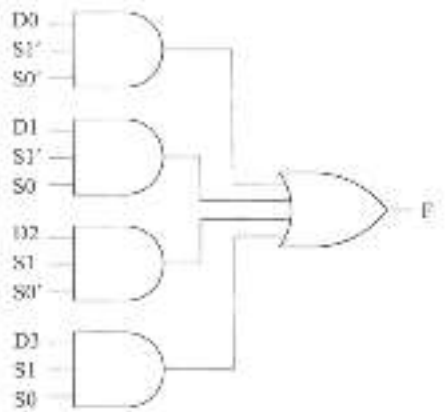


图 10-44　四输入复用器的实现

Pep/8 中 STr 指令的实现是应用复用器的一个例子。这条指令把 CPU 两个寄存器之一的内容通过总线放进内存，CPU 用一个两输入复用器来进行选择，实现这个功能。选择线来自寄存器-r 字段，输入来自 A 和 X 寄存器，输出将会到达总线。

10.4.3　二进制译码器

译码器（decoder）以一个二进制数字作为输入，把几条数据输出线中的一个设置为 1，其余的设置为 0，而哪条数据线设为 1 取决于输入二进制数字的值。

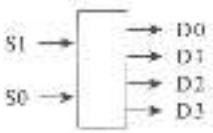
图 10-45a 展示的是一个 2×4 二进制译码器的方块图。S1 和 S0 是两位数字输入，D0 ~ D3 是四个输出，其中之一将会为 1；图 10-45b 是真值表。

因为 n 位数字会有 2^n 个值，所以译码器数据输出的数量是 2 的幂。图 10-46 展示的是一个 2×4 译码器的实现。其他可能的大小是 3×8 和 4×16。

一些译码器带有使能输入。图 10-47 是一个带使能输入的 2×4 译码器。当使能线为 1 时，设备像图 10-45b 那样正常运行。当使能线为 0 时，所有输出为 0。实现带使能功能的译码器要求每个 AND 门有一个附加的输入，细节情况作为章末的一道练习题。

Pep/8 的 CPU 中有应用译码器的一个例子。一些指令有一个三位的寻址-aaa 字段，这个字段指定八种寻址模式之一。硬件有八个地址计算单元，每个模式对应一个。每个单元有

一条使能线。二条 aaa 地址线输入到一个 3×8 译码器，该译码器的每条输出线将会使能八个地址计算单元中的一个。



a) 方块图

S1	S0	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

b) 真值表

图 10-45 2×4 二进制译码器

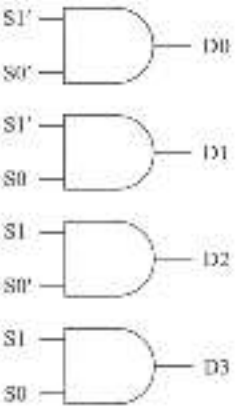


图 10-46 2×4 二进制译码器的实现

10.4.4 多路分配器

复用器是把几个数据输入值中的一个发送到唯一的输出线，多路分配器（demultiplexer）正好相反，它把唯一的输入值发送到几条输出线之一。

图 10-48a 是一个四输出多路分配器的方块图，图 10-48b 是它的真值表。如果 S1 和 S0 为 01，那么除 D1 之外的所有输出线都是 0，D1 的值和数据输入线的值一致。

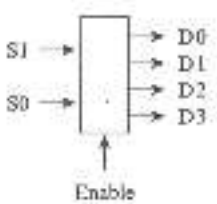
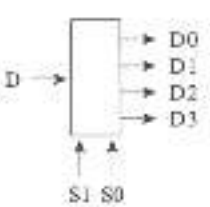


图 10-47 带使能的 2×4 二进制译码器



a) 方块图

S1	S0	D0	D1	D2	D3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

b) 真值表

图 10-48 四输出解码器

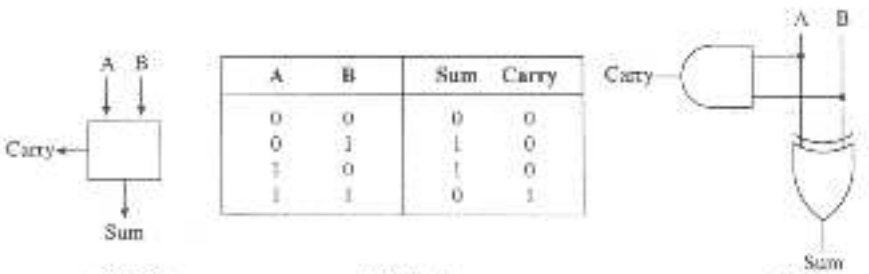
这个真值表类似于图 10-45b 的译码器的真值表。多路分配器实际上就是一个带使能功能的译码器，数据输入线和使能线相连接。如果 D 为 0 则译码器被禁用，S1 和 S0 选择的数据输出线为 0；如果 D 为 1，译码器使能，选择的数据输出线为 1。在这两种情况中，选择的输出线与数据输入线的值都一样。这又是一个用不同的视角来思考组合设备得到一种非常有用的运算的例子。

10.4.5 加法器

思考下面的二进制加法

```
        1011
ADD     0011
-------------
C = 0   1110
V = 0
```

最低位（LSB）的和是1加上1等于0，有一个1进位到下一列。要将两个数的最低位相加需要如图10-49a所示的半加器（half adder）。在图10-49中，A代表第一个数的最低位，B代表第二个数的最低位。本例中，一个输出是Sum（和）0，另一个输出是Carry（进位）1。图10-49b是真值表。Sum和XOR函数一样，Carry和AND函数一样。图10-49c部分是最直观的实现。



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

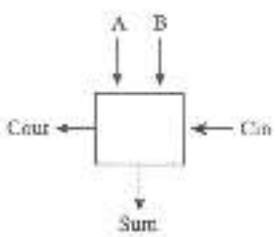
a）方块图　　b）真值表　　c）实现

图10-49　半加器

524 ~ 528

要得出LSB相邻列的和就需要一个有三个输入的组合电路：Cin、A和B，Cin是来自于LSB进位的进位输入，A和B是第一个数和第二个数的位，输出是Sum和Cout，Cout会送到下一列的全加器的Cin。图10-50a是这个线路的方块图，称为全加器（full adder）。图10-50b是它的真值表。如果三个输入的和是奇数，则Sum为1；如果二个输入的和大于1，则Cout为1。

图10-51展示的是一个全加器的实现，使用了两个半加器和一个AND门。第一个半加器将A和B相加，第二个半加器将第一个半加器的和与Cin相加，全加器的和是第二个半加器的和。如果第一个或第二个半加器有进位，则全加器就有进位。



A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

a）方块图　　b）真值表

图10-50　全加器

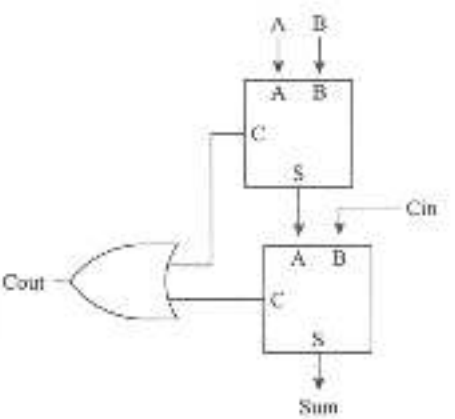


图10-51　用两个半加器实现的全加器

要将两个4位数字相加就需要一个八输入电路，如图10-52a所示。A3 A2 A1 A0是第一个数字的4位，A0是LSB，B3 ~ B0是第二个数字的4位，S3 S2 S1 S0是4位的和，C是进位位。实现4位加法器可以使用一个用于LSB的半加器和三个全加器，剩下的每列一个全加器。因为从LSB位开始的进位要像波浪一样向左边的列传递，所以这个实现称为行波进位加法器（ripple-carry adder）。图10-52b给出的就是这样的实现。

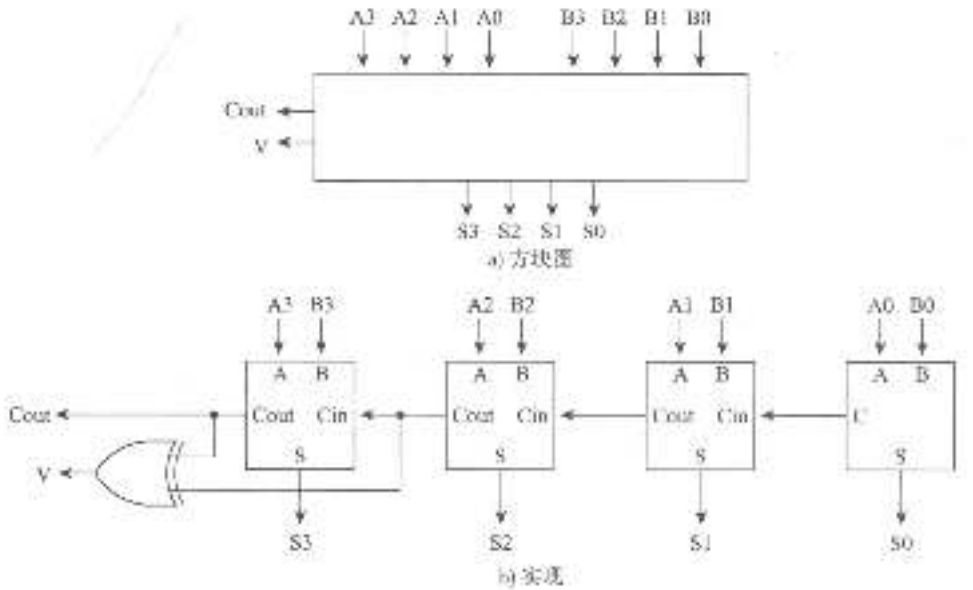


图 10-52 四位行波进位加法器

行波进位加法器的进位是最左边的全加器的 Cout。如果把整数当作无符号数，这个进位位就表明是否发生溢出。如果把整数当作二进制补码表示的有符号数，最左边的位是符号位，和它相邻的是数值最大的最高位。因此对于有符号数，倒数第二个全加器的 Cout 信号（本例中的 S2）是进位位。

V 位表示把数字看成有符号数时是否发生溢出。遇到下面两种情形之一时才可能发生溢出：

- A 和 B 都是正数，结果是负数。
- A 和 B 都是负数，结果是正数。

将两个符号不同的整数相加不会发生溢出。第一种情况中 A3 和 B3 都是 0，在倒数第二个全加器一定会有进位，这个进位将 S3 置为 1，最左边的全加器的 Cout 为 0。第二种情况中 A3 和 B3 都是 1，最左边的全加器一定有一个进位，倒数第二个全加器不会有进位，因为 S3 一定是 0。在这两种情况中，最左边全加器的进位不同于倒数第二个全加器的进位，但这正好是 XOR 函数。只有当它的两个输入不同时，它才为 1。因此可以用 XOR 门来计算 V 位，两个输入来自最左和倒数第二个全加器的 Cout 信号。

行波进位加法器最主要的缺点是进位必须传递经过所有全加器才能产生有效的结果。由于加法是非常基础的数学运算，因此加法器电路得到了广泛研究。先行进位加法器通过在设计中加入一个先行进位单元，解决了行波进位加法器的速度劣势。更复杂的加法器超出了本书的讲述范围。

10.4.6 加法器 / 减法器

要用 A 减 B，可以按照加法器的思路设计一个减法器电路，只不过相应于加法的进位机制，它要有借位机制。不过，其实不需要构造一个独立的减法电路，把 B 取反后再和 A 相加更简单一些。回想一下第 3 章的二进制补码规则：

$$\text{NEG}\,x = 1 + \text{NOT}\,x$$

529 / 530

对一个数取反时，要把这个数的所有位取反，然后加 1。因此，要构建一个既是加法器又是减法器的电路，需要通过一种方法有选择地反转 B 的所有位，还需要通过一种方法有选择地将它加 1。幸运的是 XOR 门能够实现这个功能，因为可以把 XOR 门当作一个选择反相器。

图 10-53 展示的是一个基于这个思路的加法器 / 减法器电路。图 10-53a 中的方块图和行波进位加法器方块图相比，只是多了一个标识为 Sub 的控制线。当 Sub = 0 时，电路作为一个加法器，当 Sub = 1 时，这个电路是个减法器。

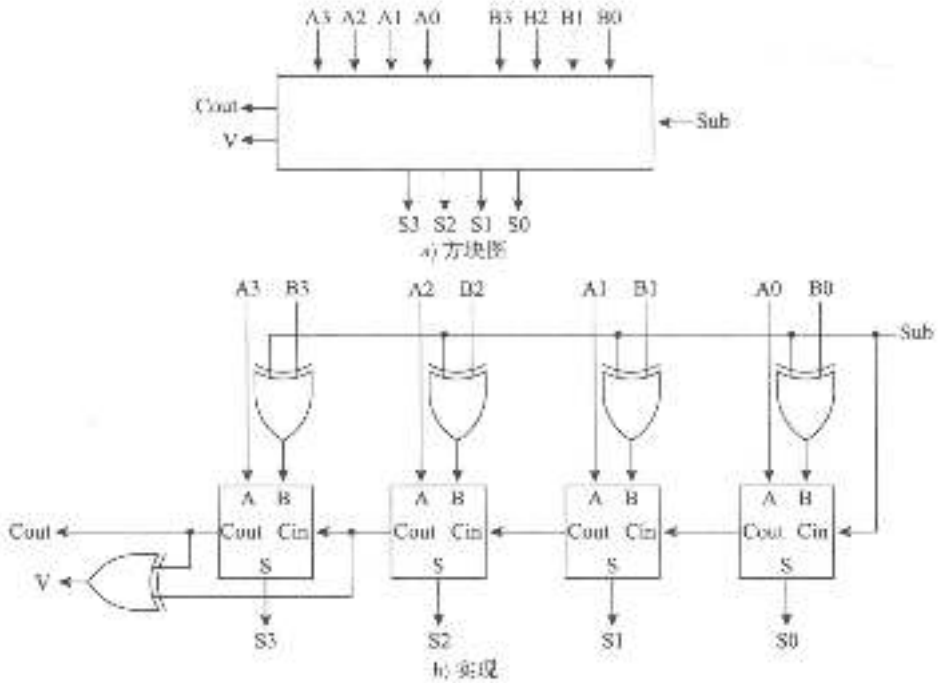


图 10-53　四位行波进位加法器 / 减法器

图 10-53b 是这个电路的实现。对于加法器电路，最低位的运算只需要一个半加器，加法器 / 减法器用一个全加器代替它。考虑 Sub = 0 的情况，这样最低位的全加器的 Cin 为 0，就像是一个半加器。而且，对于上面四个 XOR 门，每个门的左边的输入也是 0，它使得 B 信号毫无改变地通过这些门。该电路计算 A 和 B 的和。

现在来考虑 Sub = 1 的情况，由于上面四个 XOR 门左边的输入都是 1，因此 B 的所有位取反。而且，最低位的全加器的 Cin 是 1，将结果加 1。因此这个和是 A 加上 B 的反的和。

10.4.7　算术逻辑单元

Pep/8 的处理指令包括 ADDr、ANDr 和 ORr。加法是算术运算，然而 AND 和 OR 是逻辑运算。CPU 通常包括一个简单的组合电路，称为算术逻辑单元（Arithmetic Logic Unit，ALU），它会执行这些运算。

图 10-54 展示的是 Pep/8CPU 的 ALU。线上有一条小斜线代表多于 1 个的控制线，斜线旁边的数字指明控

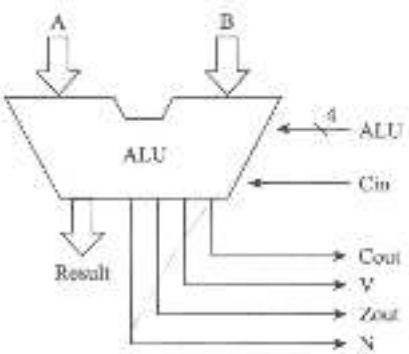


图 10-54　Pep/8 ALU 的框图

制线的数量，标记为 ALU 的线代表四条线。ALU 总共有 21 条输入线：A 输入 8 条线，B 输入 8 条线，4 条线用于指定 ALU 执行的功能，还有一条 Cin 线。它有 12 条输出线：Result 8 条线，此外还有 4 条线对应 Result 的 NZVC 值。进位输出线标记为 Cout，区别于进位输入线 Cin。零输出线标识为 Zout，区别于 CPU 中另一个 Z，这个 Z 将在第 12 章中讲述。

4 条 ALU 控制线指定 ALU 将执行 16 种功能中的哪一个。图 10-55 列出了这 16 种功能，其中大多数直接对应 Pep/8 指令集中的运算。因为“-”符号通常用作逻辑 OR 运算，所以算术运算写成“plus”和“minus”。图中列出了每种运算对应的 NZVC 位的值。

ALU 控制线			状 态 位			
(bin)	(dec)	Result	N	Zout	V	Cout
0000	0	A	N	Z	0	0
0001	1	A plus B	N	Z	V	C
0010	2	A plus B plus Cin	N	Z	V	C
0011	3	A plus $\overline{B}$ plus 1	N	Z	V	C
0100	4	A plus $\overline{B}$ plus Cin	N	Z	V	C
0101	5	A - B	N	Z	0	0
0110	6	$\overline{A - B}$	N	Z	0	0
0111	7	A + B	N	Z	0	0
1000	8	$\overline{A + B}$	N	Z	0	0
1001	9	A ⊕ B	N	Z	0	0
1010	10	$\overline{A}$	N	Z	0	0
1011	11	ASL A	N	Z	V	C
1100	12	ROL A	N	Z	V	C
1101	13	ASR A	N	Z	0	C
1110	14	ROR A	N	Z	0	C
1111	15	0	A<4>	A<5>	A<6>	A<7>

图 10-55 Pep/8 ALU 的 16 种功能

图 10-56 展示的是 ALU 的实现。可以看到从上面和右边进来的 21 条输入线和底部出来的 12 条输出线。从右边来的 4 条 ALU 线驱动一个 4×16 译码器。来回忆一下根据 ALU 输入值的情况，译码器输出线中的一条将会为 1，其余的都为 0。ALU 中的计算单元执行图 10-55 中的前 15 种功能。从译码器来的 15 条进入计算单元输入线，其中的每一条都使能一个执行对应功能的组合电路。

该计算单元有 32 条输入线：A 输入 8 条，B 输入 8 条，Cin 一条，来自译码器的 15 条。它有 10 条输出线：计算结果 8 条，加上 V 和 C 各一条。N 和 Z 位的计算在计算单元之外。在图 10-56 中可以看到 N 位只是计算单元 Result 最高位的一个副本，Z 位是 Result 的 8 个位的 NOR。如果所有 8 个位都是 0，那么 NOR 门的输出为 1；如果其中之一或者更多的输入位为 1，那么 NOR 门的输出为 0，这正好是根据计算结果设置 Z 位的条件。

左下的方框是一组 12 个双输入的复用器，每个复用器的控制线连接到译码器的第 15 号线。这个控制线的功能如下：

- 如果线 15 为 1，则从左边来的 Result 和 NZVC 被送到输出。

- 如果线 15 为 0，则从右边来的 Result 和 NZVC 被送到输出。

来看看图 10-56 如何计算图 10-55 的最后一个功能。如果 ALU 输入是 1111（bin），那么线 15 为 1，从左边来的 Result 和 NZVC 被送到 ALU 的输出。但是在图 10-56 中可以看到 Result 左边的位连接到 0，NZVC 来自 A 的低四位元组（半字节）。这正是该功能要求的。

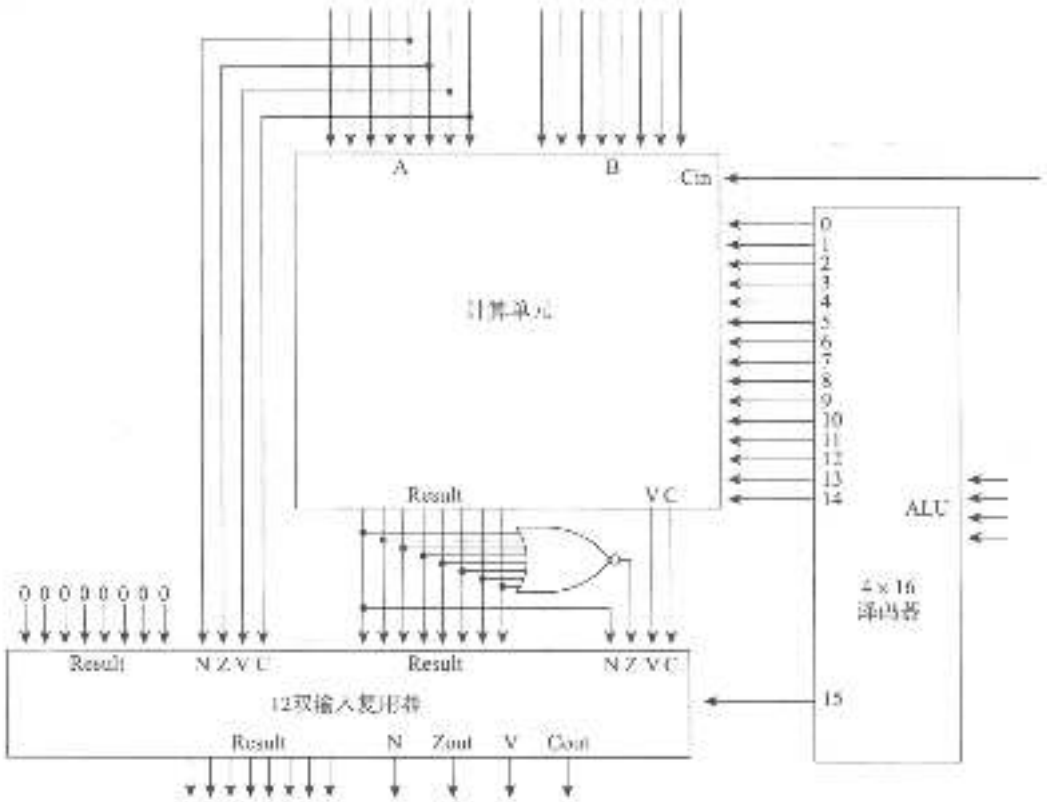


图 10-56　图 10-54 中 ALU 的实现

图 10-57 是图 10-56 计算单元的实现。它由一个 A 单元、一个算术单元和标号为逻辑单元 5 到逻辑单元 14 的 10 个逻辑单元组成。A 单元和每个逻辑单元都由译码器的 15 条线之一来使能。任何单元的使能线 E 如果为 0，那么不管这个单元的其他输入是什么，Result、V 和 C 的所有位都是 0。算术单元负责图 10-55 中功能 1、2、3 和 4 对应的算术运算的 Result、V 和 C 的计算，相应的控制线的标识分别为 d、e、f 和 g。如果 d、e、f 和 g 四个都为 0，那么不管算术单元的其他输入是什么，Result、V 和 C 的所有位都为 0。

计算单元的每个输出都连接到一个 12 输入的 OR 门。此 OR 门的其他 11 个输入是其他 11 个计算单元对应的输出线。例如，所有 12 个计算单元的 V 输出送到一个 OR 门。因为 11 个计算单元都一定是未使能的，确切地说每个 OR 门的 11 个输入都一定为 0，所以不一定为 0 的输入来自于被使能的单元。因为 0 是 OR 运算的零元

$$p \text{ OR } 0 = p$$

所以被使能单元的输出会不改变地通过 OR 门。

图 10-58 是 A 单元的实现。它由 8 个两输入 AND 门组成。这些门是 8 位 A 信号的使能门。图 10-55 表明 V 和 C 应该为 0，因此 V 和 C 输出线在这个实现中都是 0。

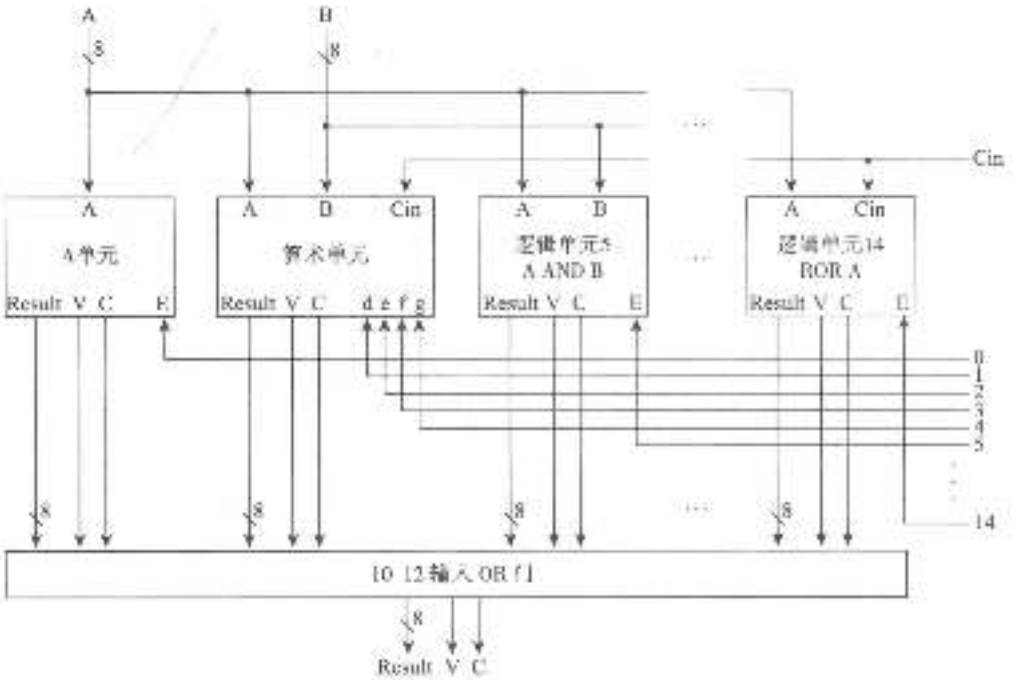


图 10-57　图 10-56 中计算单元的实现

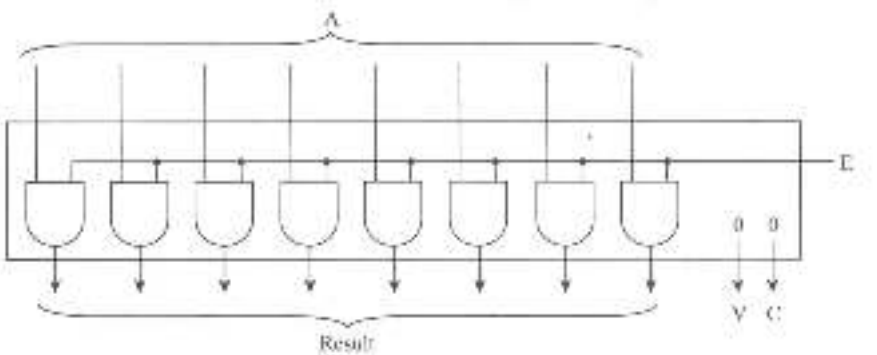


图 10-58　图 10-57 中 A 单元的实现

图 10-59 是算术单元的一种实现，它是图 10-53 的加法器 / 减法器电路的一个扩展，修改后能够处理两种额外情况：用两个 8 位运算来实现 16 位值的加减。图 10-60 展示的是怎样用两个 8 位运算来做 16 位运算。图 10-60a 中，16 位加法是这样的：对 A 和 B 的低字节执行

A plus B

然后对高字节执行

A plus B plus Cin

这里的 Cin 是低字节运算的 Cout。图 10-60b 中，16 位减法是这样的：对 A 和 B 的低字节执行

A plus $\overline{B}$ plus 1

然后对高字节执行

A plus $\overline{B}$ plus Cin

这里的 Cin 同样是低字节运算的 Cout。最后这个运算同样是依靠在硬件上实现 A 减 B 就是把 A 加上 B 的补。低字节运算的进位是加法的进位而不是减法的，这也是为什么电路要加上而不是减去低字节运算的 Cin 的原因。

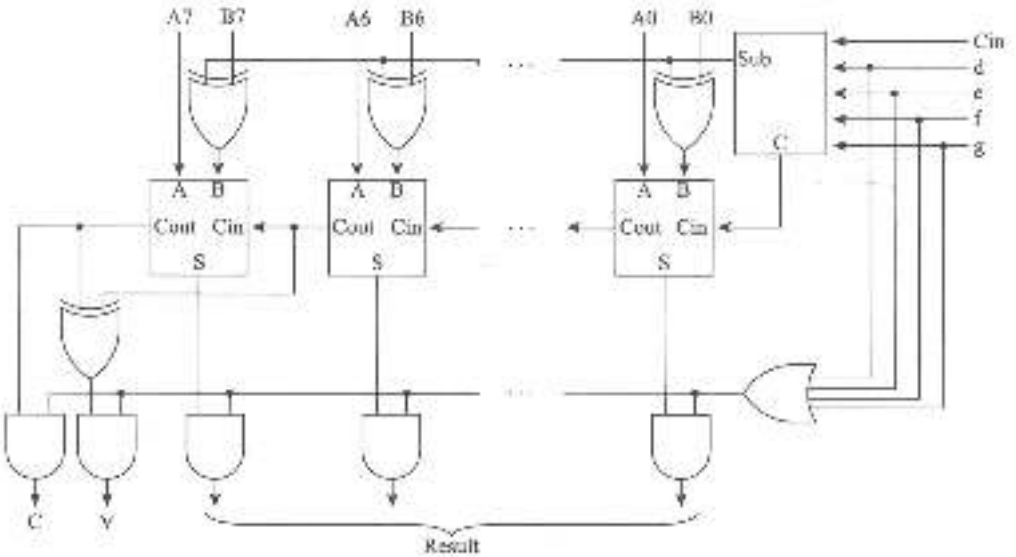


图 10-59　图 10-57 中算数单元的实现

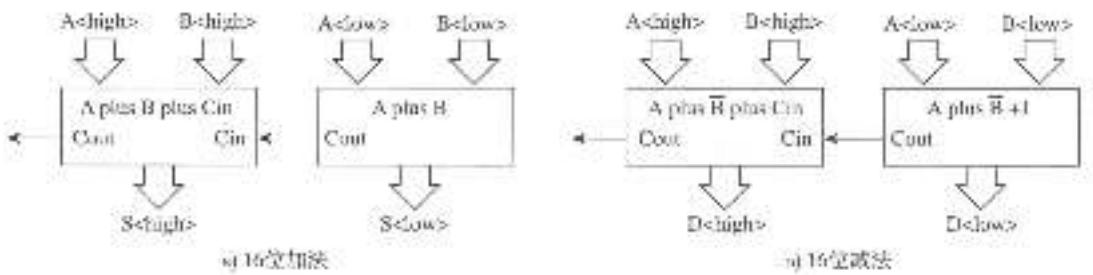


图 10-60　用两个 8 位运算来实现 16 位运算

例 10.38　下面讲解硬件怎样处理 259-261。259（dec）作为一个 16 位数，其表示为 0000 0001 0000 0011，因此 A<high>　0000 0001，A<low>= 0000 0011。261（dec）作为一个 16 位数，其表示为 0000 0001 0000 0101，因此 B<high> = 0000 0001，B<low>= 0000 0101。低字节相加是

```
        0000 0011
        1111 1010
ADD            1
C = 0   1111 1110
```

高字节相加是

```
        0000 0001
        1111 1110
ADD             0
C = 0   1111 1111
V = 0
```

最终的差是 1111 1111 1111 1110（bin）= -2（dec），和预期的一样。最终的 V 位由最后的进位和倒数第二个进位进行异或计算得出，结果如下：

0 ⊕ 0 = 0 □

例 10.39 下面讲解硬件怎样处理 261–259。这次，A<high> = 0000 0001，A<low>= 0000 0101，B<high> = 0000 0001，B<low>= 0000 0011。低字节相加是

```
        0000 0101
        1111 1100
ADD            1
C = 1   0000 0010
```

531 ~ 538

高字节相加是

```
        0000 0001
        1111 1110
ADD            1
C = 1   0000 0000
V = 0
```

最终的差是 0000 0000 0000 0010（bin）= 2（dec），和预期的一样。最终的 V 位由最后的进位和倒数第二个进位进行异或计算得出，结果如下：

1 ⊕ 1 = 0 □

图 10-59 右上部分的控制电路框控制该电路的功能，图 10-61 是它的真值表。将这个框和图 10-53 中控制加法器 / 减法器电路的 Sub 线进行比较，当加法器 / 减法器中的 Sub 为 0 时，B 不会被 XOR 门反转，低字节位计算的进位是 0；当 Sub 为 1 时，B 被反转，低字节位计算的进位是 1。这两个功能对图 10-61 的第一和第三行都是成立的，第二行用于高字节相加，最后一行用于高字节相减。

功　能	d	e	f	g	Sub	C
A plus B	1	0	0	0	0	0
A plus B plus Cin	0	1	0	0	0	Cin
A plus $\overline{B}$ plus 1	0	0	1	0	1	1
A plus $\overline{B}$ plus Cin	0	0	0	1	1	Cin

图 10-61　图 10-59 中控制电路的真值表

从理论上来讲，控制框的输出 Sub 和 C 是 d、e、f 和 g 的函数，然而通过观察真值表可以看到 Sub 可以表达为

Sub = f − g

C 可以表达为

C = e · Cin + g · Cin + f

两个表达都和 d 无关。

算术单元的一个要求是如果 d、e、f 和 g 都为 0，那么不管其他输入是什么，所有输出一定为 0。图 10-59 中四输入 OR 门的输出作为使能信号，当 d、e、f 和 g 之一为 1 时，它允许该单元的所有 10 个输出通过。 539

逻辑单元 5 ~ 14 的实现留作练习。因为普通逻辑门就能实现逻辑运算，所以它们的实现很简单。

10.4.8 LG1 层的抽象

抽象数据类型（ADT）是 HOL6 层的一个重要设计工具。其思想是了解对 ADT 进行操

作的函数和过程是做什么的，从而理解 ADT 的行为，而不必知道它们是怎样做的。一旦实现了一种运算，可以不用会实现细节而专注于解决更高抽象层级上的问题。

相同的原理在硬件层上也适用。本节中的每个组合设备都有方块图和真值表来描述它们的功能。方块图之于硬件就像 ADT 之于软件，它是一种抽象，指定输入和输出而隐藏实现细节。

硬件中更高层级的抽象可以通过构建方块图定义的设备获得，这些方块图的实现是更低抽象层级方块图的互联。图 10-50 是个完美的例子，这个全加器模块用图 10-51 的半加器模块来实现。

硬件的最高抽象层级是我们反复看到的 Pep/8 计算机的方块图。它有四个模块：输入设备、CPU、主存和输出设备，它们之间用总线相连。在稍低的抽象层级，我们可以看到 CPU 中的寄存器，每个寄存器被描绘成一个模块。后面两章将逐步扩展到更高的抽象层级，直到 ISA3 层的 Pep/8 计算机。

总结

在组合电路中，输入决定输出。组合电路的三种表示方式是真值表、布尔代数表达式和逻辑图。这三种表达方式中，真值表位于最高抽象层级，它们指定电路的功能而不指定电路的实现。真值表列出了输入所有可能组合的输出，因此称为“组合电路”。

布尔代数的三种基本运算是 AND、OR 和 NOT。布尔代数的 10 个基本属性包括 5 个定律——交换律、结合律、分配律、恒等律和互补律以及它们的对偶属性，根据它们可以证明出很有用的布尔定理。另一个重要的定理是德·摩根定律，它展示了怎样对几个项的 AND 或 OR 进行 NOT。

一个布尔表达式对应一个逻辑图，逻辑图又对应电子门的连接。三个常见的门是 NAND（AND 后面跟 NOT）、NOR（OR 后面跟 NOT）和 XOR（exclusive OR）。两级电路可以将处理时间减到最少，但是可能比等价的多级电路需要更多的门。这是又一个重要的空间/时间折中的例证。卡诺图可以帮助减少实现两级组合电路的门的数量。

[540] 组合设备包括复用器、译码器、多路分配器、加法器和算术逻辑单元（ALU）。复用器从几个数据输入中选择一个传送到唯一的数据输出。译码器将一个二进制数作为输入，将几个数据输出线中的一个设为 1，其余的设为 0。多路分配器把几个数据输入值中的一个传送到唯一的输出线，逻辑上等同于一个有使能线的译码器。半加器实现的是两个位的相加，全加器是三位相加，其中一个是前一位相加的进位。减法器的工作原理是对第二个操作数取反，然后和第一个操作数相加。ALU 执行算术和逻辑功能。

练习

10.1 节

1. *（a）用布尔代数证明零元定理 $x + 1 = 1$，给出证明中每一步的解释。提示：用互补性来扩展左边的 1，然后使用幂等性。（b）给出（a）中的对偶证明。
2. （a）用布尔代数证明吸收律 $x \cdot x \cdot y = x$，给出证明中每一步的解释。（b）给出（a）中的对偶证明。
3. *（a）用布尔代数证明合意定理 $x \cdot y + x' \cdot z + y \cdot z = x \cdot y + x' \cdot z$，给出证明中每一步的解释。（b）给出（a）中的对偶证明。
4. 用书中证明的对偶来证明德·摩根定律 $(a + b)' = a' \cdot b'$，给出证明中每一步的解释。

5.（a）根据两个变量的第·摩根定律，使用数学归纳法证明第·摩根定律的一般形式

$(a_1 \cdot a_2 \cdot \ldots \cdot a_n)' = a_1' + a_2' + \ldots + a_n'$，$n \geq 2$

（b）给出（a）中的对偶证明。

6. *（a）用布尔代数证明 $(x \cdot y) + (x' + y) = y$，给出证明中每一步的解释。（b）给出（a）中的对偶证明。

7.（a）用布尔代数证明 $(x + y) \cdot (y + x') = x + y$。给出证明中每一步的解释。（b）给出（a）中的对偶证明。

8. *（a）画出一个二输入 OR 门，它的布尔表达式和真值表如图 10-10 所示。（b）用二输入 NAND 门来实现（a）中的要求。（c）用三输入 NOR 门来实现（a）中的要求。

9. 用集合论来解释下面的布尔属性或定理：

*（a）$x + 0 = x$　（b）$x \cdot 1 = x$　（c）$x + x' = 1$　（d）$x \cdot x' = 0$

（e）$x \cdot x = x$　（f）$x + x = x$　（g）$x \cdot 0 = 0$

10. *（a）用 x、y 和 z 的重叠区域的文氏图来说明 OR 运算的结合律。画出下面的区域来表明区域（3）和区域（6）是一样的：

（1）$(x + y)$　（2）z　（3）$(x + y) + z$

（4）x　（5）$(y + z)$　（6）$x + (y + z)$

（b）给出（a）中的对偶。

11.（a）用 x、y 和 z 的重叠区域的文氏图来说明 OR 运算的结合律。画出下面的区域来表明区域（3）和区域（6）是一样的：

（1）x　（2）$y \cdot z$　（3）$x \cdot y \cdot z$

（4）$(x + y)$　（5）$(x + z)$　（6）$(x + y) \cdot (x + z)$

（b）给出（a）中的对偶。

12.（a）用 a 和 b 的重叠区域的文氏图来说明第·摩根定律。画出下面的区域来表明区域（2）和区域（5）是一样的：

（1）$a + b$　（2）$(a + b)'$　（3）a'　（4）b'　（5）$a' + b'$

（b）给出（a）中的对偶。

13. 虽然组合电路的一个布尔变量只能取两个值，但是布尔代数可以描述变量有四种可能取值的系统，这四种可能的取值是：0、1、A 和 B。这样的系统对应于 $\{a, b\}$ 的子集的描述：1 = $\{a, b\}$（全集），A = $\{a\}$，B = $\{b\}$，而 0 = $\{\}$（空集）。二输入 AND 和 OR 运算的真值表有 16 个表项而不是 4 个，求补的真值表有 4 个表项而不是 2 个。构造下述运算的真值表：

*（a）AND　（b）OR　（c）求补

14. 异或 NOR 门，写作 XNOR，等价于 XOR 后面跟一个反相器。*（a）画出二输入 XNOR 门的符号；（b）构造它的真值表。（c）XNOR 也称作比较器，为什么？

10.2 节

15. 画出下列布尔表达式的非简略逻辑图。可以使用 XOR 门。

*（a）$(a')'$　（b）$((a')')'$

*（c）$a'b + ab'$　（d）$ab + a'b'$

（e）$ab + ab' + a'b$　（f）$(ab \oplus b')' + a'b)'$

（g）$(a'bc + a)b$　（h）$(ab'c)'(ac)'$

（i）$((ab)'(b'c)' + a'b'c')'$　（j）$(a \oplus 0 + b' \oplus c')'$

（k）$(abc)' + (a'b'c')'$　（l）$(a + b)(a' + c)(b' + c')$

（m）$(a \oplus b) \oplus c + ab'c$　（n）$((a + b)' + c)' + d)'$

（o）$(ab' + b'c + ca)'$　（p）$((a + b')(b' + c)(c + d))'$

（q）$((ab)'c)'d)'$　（r）$((a \oplus b)' \oplus c)' \oplus d)'$

16. 画出练习题 15 中布尔表达式的简略逻辑图。可以使用 XOR 门。

17. 构造练习题 15 中布尔表达式的真值表。

18. 写出图 10-62 中逻辑图的布尔表达式。

541

542

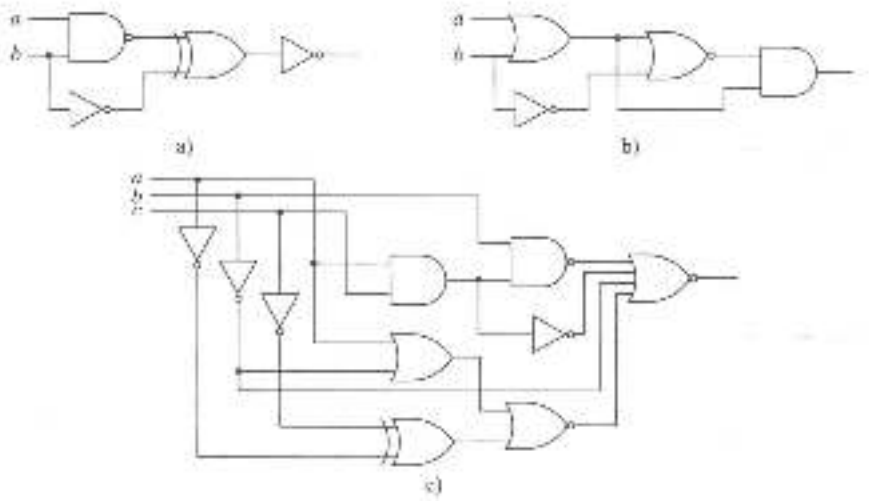


图 10-62　练习 18 的逻辑图

19. 写出下列功能或门的 AND-OR 布尔表达式：

*(a) 图 10-3 中的功能 y　(b) 图 10-4 中的功能 y　(c) 图 10-17 中的功能 x

(d) 图 10-7a 中的 NAND 门　(e) 图 10-7c 中 XOR 门

20. 写出下列功能或门的 OR-AND 布尔表达式：

*(a) 图 10-3 中的功能 y　(b) 图 10-17 中的功能 x

(c) 图 10-7b 中的 NAND 门　(d) 图 10-7c 中的 XOR 门

21. 使用布尔代数的属性和定理将下述表达式规约为不带括号的 AND-OR 表达式。得到的表达式不一定是唯一的。根据最终得到的表达式构造真值表，这是唯一的。

*(a) $(a'b + ab')'$	(b) $(ab + a'b')'$
(c) $(ab + ab' + a'b)'$	*(d) $(a \oplus b)' + ab$
(e) $(a'bc + ab$	(f) $(ab'c)'(ac)'$
(g) $(a \oplus b) \oplus c$	(h) $a \oplus (b \oplus c)$
(i) $(a + b)(a' + c)(b' + c')$	(j) $(a + b)' + c)'$

543

22. 为练习 21 中的表达式构造两级电路，仅使用 NAND 门。

23. 使用布尔代数的属性和定理将下述表达式规约为不带括号的 OR-AND 表达式。得到的表达式不一定是唯一的。根据最终得到的表达式构造真值表，这是唯一的。

(a) $a'b + ab'$	*(b) $ab + a'b'$
(c) $ab + ab' + a'b$	(d) $(ab \oplus b')' + ab)'$
(e) $(a'bc + a)b$	(f) $(ab'c)'(ac)'$
(g) $(a \oplus b) \oplus c$	(h) $a \oplus (b \oplus c)$
(i) $((a + b)(a' + c)(b' + c'))'$	(j) $(a + b)' + c$

24. 为练习 23 中的表达式构造两级电路，仅使用 NOR 门。

25. 画出一个两级电路的逻辑图，实现 XOR 功能但仅使用下述门：

*(a) 仅使用 NAND 门　(b) 仅使用 NOR 门

26. 说明图 10-63 中的每个门是不是下面的门：

(1) AND 门　(2) OR 门　(3) NAND 门　(4) NOR 门

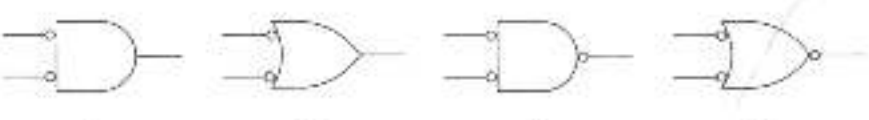


图 10-63　练习 26 的门

10.3 节

*27. 用两栖表示法写出练习 21 中的每个功能。

*28. 用逻表示法写出练习 23 中的每个功能。

29. 图 10-3 中，找出下述最小 AND-OR 表达式：

*(a) $x(a,b,c)$ (b) $y(a,b,c)$

仅使用 NAND 门画出每个表达式的最小两级电路。

30. 找出下述最小 OR-AND 表达式：

*(a) $x(a,b,c)$ (b) $y(a,b,c)$

仅使用 NOR 门画出每个表达式的最小两级电路。

31. 使用卡诺图表出 $x(a, b, c)$ 的最小 AND-OR 表达式：

*(a) $\Sigma(0, 4, 5, 7)$ (b) $\Sigma(2, 3, 4, 6, 7)$ (c) $\Sigma(0, 3, 5, 6)$
(d) $\Sigma(0, 1, 2, 3, 4, 6)$ (e) $\Sigma(1, 2, 3, 4, 5)$ (f) $\Sigma(1, 2, 3, 4, 5, 6, 7)$
(g) $\Sigma(0, 1, 2, 4, 6)$ (h) $\Sigma(1, 4, 6, 7)$ (i) $\Sigma(2, 3, 4, 5, 6)$
(j) $\Sigma(0, 2, 5)$

[544]

*32. 用逻表示法写出练习 31 中的每个表达式，并确定它的最小 OR-AND 表达式。

33. 使用卡诺图表出 $x(a, b, c, d)$ 的最小 AND-OR 表达式：

*(a) $\Sigma(2, 3, 6, 5, 10, 12, 13)$
(b) $\Sigma(1, 5, 6, 7, 9, 12, 13, 15)$
(c) $\Sigma(0, 1, 2, 4, 6, 8, 10)$
(d) $\Sigma(7)$
(e) $\Sigma(2, 4, 5, 11, 13, 15)$
(f) $\Sigma(1, 2, 4, 5, 6, 7, 12, 15)$
(g) $\Sigma(1, 2, 4, 5, 6, 7, 8, 11, 12, 15)$
(h) $\Sigma(1, 7, 10, 12)$
(i) $\Sigma(0, 2, 3, 4, 5, 6, 8, 10, 11, 13)$
(j) $\Sigma(0, 1, 2, 3, 4, 5, 6, 10, 11, 13, 14, 15)$
(k) $\Sigma(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)$

*34. 用逻表示法写出练习 33 中的每个表达式，并确定它的最小 OR-AND 表达式。

35. 使用无关条件用卡诺图表出 $x(a, b, c, d)$ 的最小 AND-OR 表达式。

*(a) $\Sigma(0, 6) + d(1, 3, 7)$ (b) $\Sigma(5) + d(0, 2, 4, 6)$
(c) $\Sigma(1, 3) + d(0, 2, 4, 6)$ (d) $\Sigma(0, 5, 7) + d(3, 4)$
(e) $\Sigma(1, 7) + d(2, 4)$ (f) $\Sigma(4, 5, 6) + d(1, 2, 3, 7)$

36. 使用无关条件用卡诺图表出 $x(a, b, c, d)$ 的最小 AND-OR 表达式：

*(a) $\Sigma(5, 6) + d(2, 7, 9, 13, 14, 15)$
(b) $\Sigma(0, 3, 14) + d(2, 4, 7, 8, 10, 11, 13, 15)$
(c) $\Sigma(3, 4, 5, 10) + d(2, 11, 13, 15)$
(d) $\Sigma(5, 6, 12, 15) + d(0, 4, 10, 14)$
(e) $\Sigma(1, 6, 9, 12) + d(0, 2, 3, 4, 5, 7, 14, 15)$
(f) $\Sigma(0, 2, 3, 4) + d(8, 9, 10, 11, 13, 14, 15)$
(g) $\Sigma(2, 3, 10) + d(0, 4, 6, 7, 8, 9, 12, 14, 15)$

37. (a) 三变量的卡诺图中最小项 0 和 2 相邻，4 和 6 相邻。复制一份图 10-30，把卡诺图剪下来并围成圆柱状，使每相邻的最小项真正相邻。(b) 要想使四变量卡诺图中的相邻最小项实际相邻，需要一个三维的圆环面（形状像一个甜甜圈）。用报卡或其他合适的材料，在上面刻上或写上图 10-33a 所示的单元和十进制标号。例如，标号为 2 的单元应该与标号 0、3、6 和 10 的单元相邻。

[545]

10.4 节

38. 把一条线当作数据线，另一条当作控制线，解释下列二输入门的操作：

*(a) OR (b) NAND (c) NOR (d) XNOR

XNOR 的定义参见练习 14。

39. 画出八输入复用器的非简略逻辑图。

*40. 用五个四输入复用器构造一个16输入复用器。把16输入复用器画成一个大方框，有16条数据线，标号为D0～D15，还有4条选择线，标号为S3～S0。在这个大方框里，每个四输入复用器是一个小方框，数据线为D0～D3，选择线S1和S0。画出实现大复用器需要的从小方框到外部线路的连接，以及方框之间的连接。解释你的电路是如何运行的。

41. 用两个不带使能的八输入复用器以及任何你需要的其他门来完成练习40的要求。解释你的电路是如何运行的。

42. *（a）画出一个32×8二进制译码器的非简略逻辑图。（b）画出一个带使能的2×4二进制译码器的非简略逻辑图。

43. 用五个带使能的2×4二进制译码器构造一个不带使能的4×16二进制译码器。设备的输入可以是常数1。使用练习40的画图规则对外部和内部线路进行标识。解释你的电路是如何运行的。

44. 用两个带使能的3×8二进制译码器和任何需要的其他门构造一个不带使能的4×16二进制译码器。使用练习40的画图规则对外部和内部线路进行标识。解释你的电路是如何运行的。

45. 实现一个如图10-47所示的带使能的2×4二进制译码器。画出你的电路的非简略图。

46. *（a）画出图10-51中全加器的实现，给出半加器的AND和XOR门。*（b）从输入到输出最大的门延迟数是多少？（c）根据图10-50b的真值表，设计Sum和Cout的最小二级网络。（d）将（c）的设计与（a）的设计进行比较，计算门的数量和处理时间的变化百分比。计算结果如何说明空间/时间的折中问题？

47.（a）画出图10-52的电路，包括半加器的XOR、AND和OR门。*（b）从输入到输出最大的门延迟数是多少？假设XOR门需要一个门延迟。这个问题需要仔细思考。虽然该电路的进位是按报式前进的，但还是假设所有八个输入都同时出现。

546

48. 修改图10-52b，增加两个输出：N位和Z位。

49. 实现一个带选择位S的四位ASL移位器，输入为A3 A2 A1 A0，它代表一个数，A0是LSB，A3是符号位。则输出是B3 B2 B1 B0和进位位C。如果S为1，则输出是输入的ASL；如果S为0，则输出与输入一致，C为0。

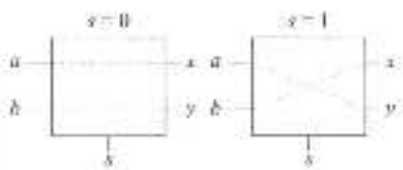


图10-64　练习51的框图

50. 为四位ASR移位器完成练习49的要求。

51. 图10-64的方块图是一个二输入二输出的组合开关电路。如果s为0，则输入a直接送到x，b送到y；如果s是1，则两者交换，a送到y，b送到x。只用AND、OR和反相器构造该电路。

52. 图10-65中的方块图是一个四输入二输出的组合开关电路。如果$s1\ s0 = 00$，输入a广播到x和y；如果$s1\ s0 = 01$，则输入b广播到x和y；如果$s1\ s0 = 10$，则a和b直接通过到x和y；如果$s1\ s0 = 11$，则a和b交换，a送到y，b送到x。只用AND、OR和反相器构造该电路。（a）使用卡诺图构造最小的AND-OR电路。（b）使用卡诺图构造最小的OR-AND电路。

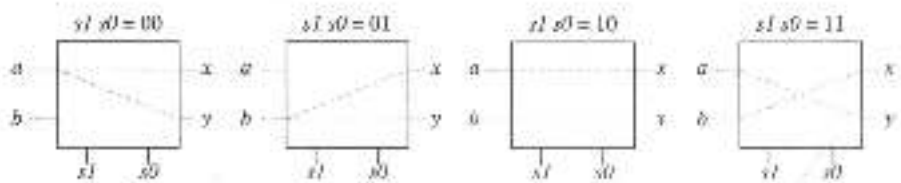


图10-65　练习52的框图

53. 画出图10-56的12个二输入复用器，给出所有输入到输出到连接线。可以用省略号（…）表示八条数据线中间的六条。

54. 实现下面这些 Pep/8 ALU 的逻辑单元：

(a) 逻辑单元 5, A · B　　(b) 逻辑单元 6, $\overline{A \cdot B}$

(c) 逻辑单元 7, A + B　　(d) 逻辑单元 8, $\overline{A + B}$

(e) 逻辑单元 9, A ⊕ B　　(f) 逻辑单元 10, A

(g) 逻辑单元 11, ASL A　　(h) 逻辑单元 12, ROL A

(i) 逻辑单元 13, ASR A　　(j) 逻辑单元 14, ROR A

55. 画出图 10-59 中五输入二输出控制框的实现。

547 : 548

时序电路

第 10 章讨论了组合设备，它们在计算机设计中很常用。尽管如此，仅把组合电路互联起来是不可能构造出哪怕最小的计算机的。在所有提到的组合设备中，输出只取决于输入。当输入改变时，只需要几个门延迟点时间，输出就会发生改变。

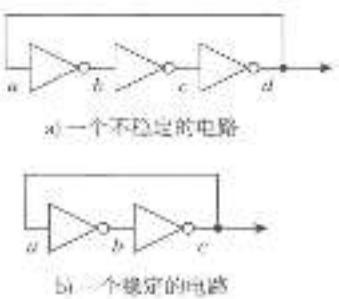
电路的状态（state）是时序电路区别于组合电路的特性，时序电路能记住它处在什么状态，换句话说就是它有记忆。时序电路的输出不仅取决于输入也取决于它的状态。

本章讲述怎样构建基本的时序元件，以及怎样连接它们以在更高抽象层级上形成有用的块，最后讲述一些设备。在下一章中会把这些设备连接起来构成 Pep/8 计算机。

11.1 锁存器和时钟触发器

时序设备由在第 10 章中讲述的同样的门构成，不过它有一种不同的连接方式——反馈（feedback）。在组合电路和描述它们的布尔表达式中，每个门的输出会连接到之前未连接过的门的输入。然而，反馈连接会形成回路或环路，一个或多个门的输出“反馈”到电路中前面的门的输入。

图 11-1 展示的是两个简单的有反馈连接的电路。图 11-1a 是一串三个反相器，反馈到第一个反相器的输入。要分析这个电路的行为，假定 d 点的值为 1。由于反馈回路把 d 连接到了 a 点，因此 a 点的值一定也是 1。一个门延迟后，b 点的值一定为 0（尽管为了简便起见我们在前面忽略了反相器的延迟，但在这个电路里必须要考虑延迟）。再经过一个门延迟后，c 点将为 1。在第三个门延迟后，d 将为 0。



a）一个不稳定的电路

b）一个稳定的电路

图 11-1 简单的反馈电路

现在的问题是分析开始时，我们假设 d 点为 1，现在它变为 0，经过三个门延迟后，它又变成了 1。这个电路会振荡，每隔几个门延迟，电路中每个点的值会在 1 和 0 之间来回变换。仅能保持几个门延迟时间不变的状态称为非稳态（unstable state）。

图 11-1b 中，如果假定 c 点的值为 1，那么 a 点将为 1，b 将为 0，c 将为 1，这和开头的假设是一致的。这样的状态是稳定的，电路中所有的点将永久地保持它们的值。

另一种可能的稳态是点 c 和 a 的值为 0，b 为 1。如果构建这样的一个电路，它会处于哪种状态呢？c 点会是 0 还是 1 呢？和所有电子设备一样，门需要一个电源打开才能运行。如果构建图 11-1b 的电路并打开它，它的状态会随机建立。打开电路时，c 点有一半的机会为 0，一半的机会为 1。电路会一直保持电源打开时建立的那种状态。

11.1.1 SR 锁存器

为了实用，时序设备需要一种设置状态的机制。图 11-2 的 SR 锁存器（SR latch）就是这样一种设备。S 和 R 是它的两个输入，它的两个输出是 Q 和 $\overline{Q}$（读作 Q bar）。两个反馈连

接是从 Q 到下面 NOR 的输入以及从 $\overline{Q}$ 到上面 NOR 的输入。

来看看稳态的可能性，假定 S 和 R 都是 0，Q 也是 0。下面门的两个输入都是 0，这使得 $\overline{Q}$ 为 1。上面 NOR 的两个输入为 0（R）和 1（$\overline{Q}$）。因此上面 NOR 的输出是 0，这和我们开头对 Q 的假设是一致的，因此当 SR = 00 时，$Q\overline{Q}$=01 是稳态。

从这个稳态开始，思考一下如果将输入 S 变为 1 会发生什么。图 11-3 概括了这个事件序列。T_g 表示一个门延迟的时间间隔，通常是 2ns。

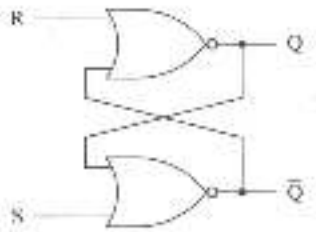


图 11-2 SR 锁存器

时 间	S	R	Q	$\overline{Q}$	稳 定 性
初始值	0	0	0	1	稳定
0	1	0	0	1	不稳定
T_g	1	0	0	0	不稳定
$2T_g$	1	0	1	0	稳定

图 11-3 SR 锁存器中将 S 变为 1

550

在时间 0，S 变为 1，这使得下面门的两个输入变为 1（S）和 0（Q）。一个门延迟后，这个变换的影响传送到 $\overline{Q}$，$\overline{Q}$ 变为 0。现在上面门的两个输入为 0（R）和 0（$\overline{Q}$）。再经过一个门延迟后，上面门的输出变为 1。现在下面门的两个输入为 1（S）和 1（Q），这使得其输出为 0。

不过下面门的输出已经为 0，因此没有变化。因为沿着反馈连接跟踪得到一致的值，所以最后一个状态是稳定的。图 11-3 的中间两个状态是不稳定的，因为它们仅持续了几个门延迟的时间。

如果把 S 改回 0 会怎么样呢？图 11-4 展示了这个事件序列。下面门的两个输入为 0（S）和 1（Q），所以它的输出为 0。它已经为 0 了，电路中也没有其他变化，所以该状态是稳定的。

时 间	S	R	Q	$\overline{Q}$	稳 定 性
初始值	1	0	1	0	稳定
0	0	0	1	0	稳定

图 11-4 SR 锁存器中将 S 变回 0

从图 11-3 和图 11-4 可以看到，当 SR 锁存器处于稳态时，$\overline{Q}$ 总是 Q 的补。上面加一横是补的另一种常用的符号，等同于第 10 章中使用的一撇的符号表示。

将图 11-3 中第一个状态和图 11-4 中最后一个状态进行比较可以看到，两种情况中 SR 都等于 00，但是第一种情况的输出是 Q = 0，第二种情况的输出是 Q = 1。输出不仅取决于输入，也取决于锁存器的状态。

S 变为 1 又变回 0 的效果是将状态设置为 Q = 1。如果锁存器以 SR = 00 和状态 Q = 1 开始，那么通过类似的分析可以看到把 R 变为 1 又变回 0 将会把状态重置为 Q = 0。S 表示设置，R 表示重置。

SR 锁存器类似于墙上的电灯开关。将 S 变为 1 再变回 0 就像把开关推上打开电灯，将 R 变为 1 再变回 0 就像把开关拉下。如果开关已经是打开状态，那么再尝试打开它时不会发生什么改变，开关仍然是打开状态。类似情况，如果 Q 已经是 1 了，把 S 变为 1 再变回 0 时状态不会改变，Q 仍然为 1。

一般情况下 SR 锁存器的输入情况是 SR = 00。要设置或重置锁存器，就要把 S 或者 R 变为 1 再变回 0。通常 S 和 R 不会同时为 1，如果 S 和 R 都是 1，那么 Q 和 $\overline{Q}$ 就会都为 0，$\overline{Q}$ 将不是 Q 的补。此外，如果将 SR = 11 同时改为 SR = 00，那么锁存器的状态是不可

预知的。一半的可能是 $Q\overline{Q}=01$，一半的可能会是 $Q\overline{Q}=10$。实际中不会出现 SR = 11 的情况。

时序图（timing diagram）是时序电路行为的图形化表示。图 11-5 是一个时序图，它展示了输出 Q 和 $\overline{Q}$ 随输入 S 和 R 的改变而改变，横轴是时间，纵轴是电压，高电平为 1，低电平为 0。

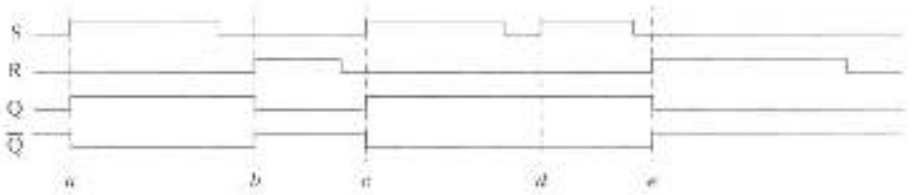


图 11-5　SR 锁存器的时序图

从这个时序图中我们可以看到初始状态 Q = 0，在时间点 *a*，S 变为 1，Q 马上被设置为 1，$\overline{Q}$ 被设置为 0。图中显示变化是同时发生的，就像前面分析的一样，$\overline{Q}$ 将在 S 改变一个门延迟的时间之后才会变化，Q 将在这之后一个门延迟的时间再变化。这张时序图假设的时间比例尺太大，不能展示出像门延迟这样短的时间间隔。

当 S 变回 0 时，状态不改变。在时间点 *b*，R 变为 1 时，Q 重置为 0。当 S 在时间点 *c* 又变为 1 时，Q 被设置为 1。在时间点 *d*，S 从 0 变为 1，这不会改变锁存器的状态，因为 Q 已经是 1 了。在图中所有点处 $\overline{Q}$ 都是 Q 的反。

图 11-5 中显示的转移是瞬间完成的，而现实中没有什么可以不耗时就发生。如果放大时间比例尺，转移将会呈现为在每个点有一定坡度的、更加平缓的变化。可以把时序图中同时瞬间的转移看作一个高层级的抽象，它隐藏了低抽象层级的门延迟和平缓的坡度变化。

11.1.2　钟控 SR 触发器

计算机中的子系统由许多组合设备和时序设备组成。每个时序设备就像一个 SR 锁存器，处于两种状态之一。当计算机执行冯·诺依曼循环时，所有时序设备的状态随着时间而改变。要以有序的方式控制这一大堆设备，计算机就要维护一个时钟，并要求所有设备同时改变它们的状态。时钟始终持续生成脉冲序列，如图 11-6 所示，Ck 表示时钟脉冲（clock pulse）。

551
~
552

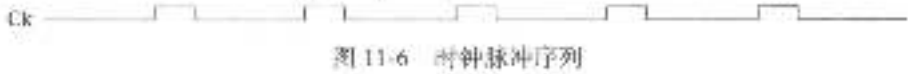


图 11-6　时钟脉冲序列

每个时序设备除了其他输入外，都有一个 Ck 输入。设备仅在时钟脉冲期间响应它的输入。脉冲之间的时间，图中用 *T* 表示，是时钟的周期（period）。周期越短，设备改变状态就越频繁，电路计算速度就越快。

图 11-7 是有时钟输入的 SR 锁存器，叫作触发器（flip-flop）。它由和图 11-2 所示同样的一对有反馈的 NOR 门组成，不过 SR 不是直接输入到 NOR 门，而是先通过两个作为使能端的 AND 门。图 11-7a 是方块图，图 11-7b 是它的一

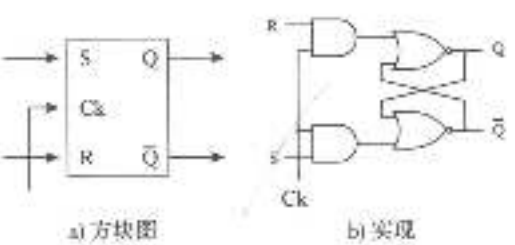


图 11-7　钟控 SR 触发器

种实现。注意这个方块图使用的惯例是把 S 放在方块图上面 Q 的对面，而实现中 S 在 $\overline{Q}$ 的对面。

在 Ck 为低电平期间，不管 S 和 R 的值是什么，NOR 门的输入都为 0。当 NOR 门的输入为 0 时，意味着锁存器不会改变状态。Ck 为高电平期间，S 和 R 不改变地通过使能门，设备按照图 11-2 的 SR 锁存器方式运行。除非 Ck 为高电平，否则 AND 门保护 NOR 门免受 S 和 R 的影响。图 11-8 给出了这个设备行为的时序图，$\overline{Q}$ 一直是 Q 的补，在本图中没有给出。

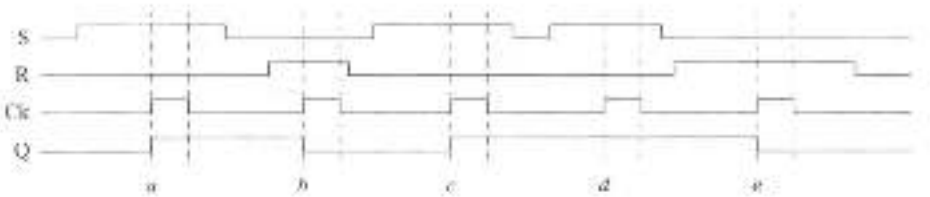


图 11-8 钟控 SR 触发器的时序图

当 S 变为 1 时，这个改变不会影响 Q，因为时钟仍然是低电平。在时间点 a 时钟变为高电平，Ck 允许 SR = 10 通过 AND 门，设置锁存器 Q = 1。稍后当时钟变为低电平时，Ck 禁止 SR 输入锁存器。如果 R 在时间点 b 之前就变为 1，这不会影响锁存器的状态。只有在时间点 b，Ck 再次高电平时，R 才能重置锁存器。

时钟为高电平时，在一个时间间隔内，实际上是有可能让 S 和 R 做几次转移的，但是现实中不会发生这样的情况。电路的主要设计思路是，把 SR 输入设置成想要的转移，然后等待下一个时钟脉冲。当时钟脉冲到来时，状态可以根据 S 和 R 的值而改变。时钟变为低电平之后，电路给下一个脉冲准备 SR 值。

均匀分布的脉冲使任何状态改变只在均匀分布的时间间隔发生。图 11-8 的 S 和 R 输入与图 11-5 的输入一样，但是 Q 中相应的状态变化被时钟改变了。时钟的作用就是使时间（时序图的横轴）数字化，同电子电路使电压信号（纵轴）数字化的方法一样。就像信号一定或者为高电平或者为低电平而绝不会是中间的情形一样，时序电路的状态改变一定发生在某个时钟脉冲或另一个时钟脉冲——绝不会在两个脉冲之间。

11.1.3 主 – 从 SR 触发器

锁存器只有当时钟为高电平时才对 Ck 进行响应，所以图 11-7 的钟控触发器称为电平敏感的（level sensitive）。尽管这个设备会像预期的那样遵循时钟，但是它有一个严重的实际缺陷，图 11-9 说明了这个问题。

此图展示了 SR 设备的一种可能的连接情况。时序设备的输出包含一个穿过某个组合电路的反馈环，最终通向同一时序设备的输入，这是很常见的。图中有一个三输入两输出的组合电路，其中两个输入是 SR 时序设备的输出反馈。这个反馈环同时还是 NOR 门的反馈，图中没有在 SR 框中显示出来。



图 11-9 SR 设备的一种可能的连接

想一想如果 SR 触发器是电平敏感的会

发生什么。假设 SR 是 10，$Q\overline{Q}$ 是 01，时钟是低电平。因为时钟禁止 SR 进入 NOR 锁存器，所以 S 不会将 Q 设置为 1。现在假设时钟变为高电平，经过几个门延迟后，SR 将 Q 设置为 1。

现在想想 $Q\overline{Q}$ 的变化，在用同样的外部输入经过组合电路后，使 SR = 01。如果 Ck 仍然为高电平，那么时钟将允许 SR 的值再经过几个门延迟后将 Q 重置为 0。不幸的是，$Q\overline{Q}$ 的值 01 将再次通过组合电路，把 SR 变为 10。

你应该认识到这个状况是不稳定的。每过几个门延迟，只要时钟是高电平，时序设备的反馈连接就使得 SR 触发器改变状态。时钟为高电平时，状态会改变成百上千次。当时钟在脉冲末端最后变成低电平时，不可能准确预测触发器处于什么状态。

因为穿过组合电路的反馈连接对于构建计算机子系统来说是必需的，所以我们需要时序设备不仅要限定在时钟脉冲期间才会改变状态，而且要避免受到反馈连接的进一步改变。设备需要对它的输入在极其短的时间内敏感——短到不管反馈通过组合电路并改变 SR 有多快，这个改变都不能再影响触发器的状态。

设计这样的设备有两种技术：边沿触发和主 - 从。边沿触发（edge-triggered）触发器不会在时钟为高电平时对输入敏感，而是当时钟从低电平转移到高电平时对输入敏感。边沿触发触发器的实现要比主 - 从触发器更难于理解，尽管它是两种技术中更常用的，但是在这里我们不进行讲解。了解这两类触发器都解决了反馈连接引发的同样的问题，这就足够了。

图 11-10 展示的是主 - 从（master-slave）SR 触发器的实现。主触发器和从触发器都是电平敏感的钟控 SR 触发器。主触发器（$\overline{Q2}$）的 $\overline{Q}$ 输出连接从触发器（R2）的 R 输入，主触发器（Q2）的 Q 输出连接从触发器（S2）的 S 输入，Ck 连接主触发器的使能，Ck 的补连接从触发器的使能。主 - 从触发器的方块图和电平敏感的触发器是一样的。

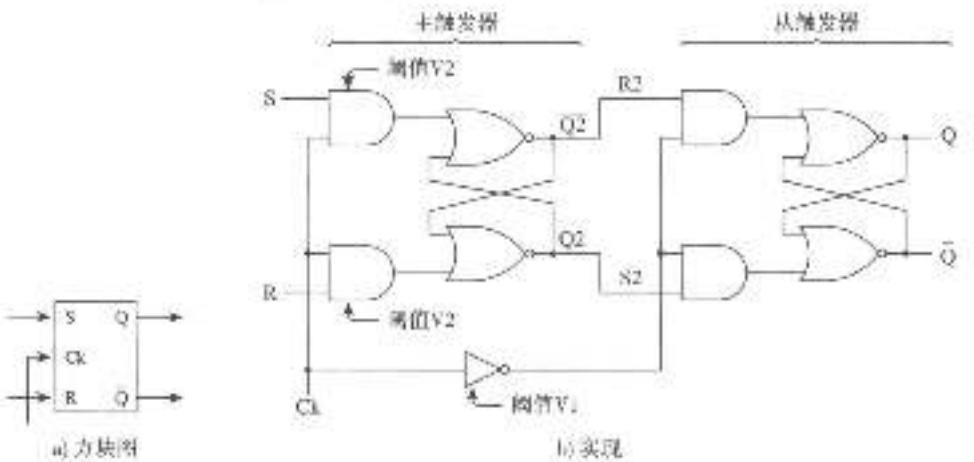


图 11-10　主 - 从 SR 触发器

主触发器是一个 SR 触发器，因此 Q2 和 $\overline{Q2}$ 将会一直互为补。Q2 连接从触发器的 S2，$\overline{Q2}$ 连接从触发器的 R2。那么时钟到达从触发器时，将根据主触发器的状态被设置或者重置。如果主触发器处于状态 Q2 = 1，那么将把从触发器也设置为 Q = 1；如果主触发器处于状态 Q2 = 0，那么将把从触发器重置为 Q = 0。使用主 - 从这个术语就是因为时钟到达从触发器时，它会接受主触发器的状态。

门的阈值（threshold）是刚好能导致输出变化的输入值。要使主 - 从电路正确地工作，

主触发器的反相器和使能门必须具有特殊的阈值。反相器的阈值 V1 必须小于主使能门的阈值 V2。

图 11-11 展示了 V1 和 V2 与 Ck 的一个时钟脉冲放大的时序图的关系。时钟不是从低电平到高电平的瞬时转化，而是逐步增长，首先在时间点 t_1 到达 V1，接着在时间点 t_2 到达 V2，最后到达高电平。在下降的过程中，在时间点 t_3 经过 V2，在时间点 t_4 经过 V1。

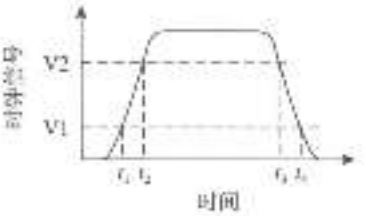


图 11-11 一个时钟脉冲的时间细节

在脉冲开始向上转变前，主触发器是禁止输入的。不管 SR 的值是什么，主触发器将保持已有的状态。因为从触发器输入是使能的，所以反相器确保从输入连到主触发器，从触发器一定处于和主触发器相同的状态。

随着时钟信号的上升和下降，经过时间点 t_1、t_2、t_3 和 t_4，这个电路的效果如下：

- t_1：将从触发器与主触发器隔离开
- t_2：主触发器连接到输入
- t_3：主触发器和输入断开
- t_4：将从触发器连接到主触发器

在时间点 t_1，信号达到反相器的阈值，使反相器的输出由 1 变为 0。从触发器的使能门为 0，保护从触发器不受 S2 和 R2 的影响。不管从触发器在时间点 t_1 是什么状态，它都保持这个状态直到 Ck 超过阈值 V1。

在时间点 t_2，信号达到主触发器使能门的阈值，使主触发器对 SR 输入敏感。如果 SR 是 10，那么输入将会把主触发器设置为 Q2 = 1；如果 SR 为 01，那么输入将重置主触发器为 Q2 = 0；如果 SR 为 00，那么主触发器的状态将保持不变。如果主触发器的状态不变，那么它的新状态就不会影响从触发器，因为在时间点 t_1 从触发器和主触发器是隔离开的。

思考一下这样的安排是怎样让触发器免于受图 11-9 的反馈连接影响的。这个反馈基于从触发器的 $Q\overline{Q}$ 输出，主触发器的输入不会导致它发生改变。V1 小于 V2 确保了在主触发器变为对输入敏感之前，从触发器和主触发器是分离的。就算穿过组合电路的门延迟是 0，反馈也不会影响从触发器的状态。

当 Ck 从高电平转变为低电平时，时钟在时间点 t_3 到达 V2。V2 是主触发器使能门的阈值，因此这时主触发器变得对输入敏感。因为 V2 比 V1 大，所以从触发器仍然免受主触发器的影响。

在时间点 t_4，时钟信号变成低于反相器的阈值，反相器的输出从 0 变为 1，将从触发器和主触发器相连。不管主触发器的状态是什么，从触发器都会被强制设置成这个状态。从触发器可能会改变状态。如果从触发器的输出反馈到主触发器的输入，这不会影响到主触发器，因为在时间点 t_3 主触发器和从触发器的输入是隔离开的。

主－从电路的运行大致类似于宇宙飞船的减压舱。在飞船内，宇航员不需要穿太空服，而去飞船外进行太空行走则要穿上太空服进入有两个门的减压舱，这两个门初始是关闭的。

宇航员打开连接飞船和减压舱的内门进入减压舱，然后关上内门将减压舱和飞船隔离，接着打开外门，减压舱和太空就连上了，从外门出去后，再将外门关上。在任何时候两个门不会同时打开，如果这样，飞船的空气就会全部泄漏到太空里。

类似地，主－从电路有两个门——一个门连接或隔离主触发器的输入，一个门连接或隔离从触发器和主触发器。在任何时候两个门不会同时打开，即主触发器连接输入且从触发器

连接主触发器。如果同时打开，反馈线路会在电路中形成不稳定的状态。

图 11-12 是主－从 SR 触发器行为的时序图。在时间点 t_4 从触发器连接主触发器，从触发器的状态 Q 发生改变，这发生在 Ck 从高到低的转变时期。

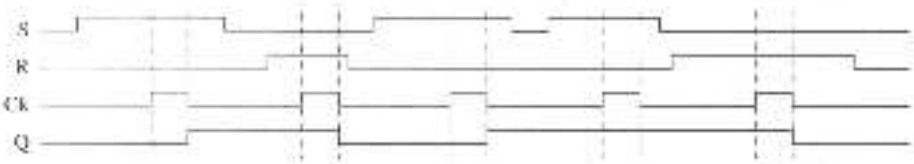


图 11-12　主－从 SR 触发器的时序图

由于触发器不是组合电路，真值表不足以刻画它的行为，取而代之的是特征表（characteristic table），它说明了给定输入和初始状态在一个时钟脉冲后的状态。图 11-13 是 SR 触发器的特征表。

S(t)	R(t)	Q(t)	Q(t + 1)	条　件
0	0	0	0	无变化
0	0	1	1	
0	1	0	0	重置
0	1	1	0	
1	0	0	1	设置
1	0	1	1	
1	1	0	–	未定义
1	1	1	–	

图 11-13　SR 触发器的特征表

S(t) 和 R(t) 是在时钟脉冲之前时间点 t 的输入，Q(t) 是在时钟脉冲之前触发器的状态，Q(t + 1) 是脉冲之后的状态。从这个表可以看出，如果 SR 为 00，那么在时钟到达的时候，设备状态不会改变；如果 SR 为 01，设备设置为 Q = 0；如果 SR 为 10，设备设置为 Q = 1。如果时钟到达时 SR = 11，那么无法预测设备是什么状态。

特征表实质上是一个状态变换表，类似于有限状态机的图 7-11。SR 触发器是一个有限状态机，有两种可能的状态，Q = 0 和 Q = 1。和任何有限状态机一样，它的行为可以用一个状态转移图来表示。图 11-14 是 SR 触发器的状态转移图。

553 ~ 558

圆圈表示状态机的状态，圆圈里是 Q 的值。转移由引起指定转移的 SR 值标记出来，例如从 Q = 0 到 Q = 1 的转移标记为 SR = 10。

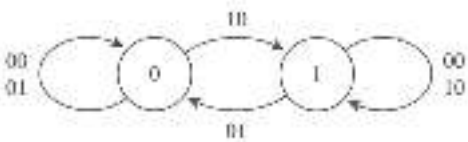


图 11-14　SR 触发器的状态转移图

11.1.4　基本触发器

下面是计算机设计中常用的四类触发器：

- SR　　设置 / 重置
- JK　　设置 / 重置 / 反转
- D　　数据或延迟
- T　　反转

前面一节介绍了怎样构建 SR 触发器，它的特征表定义了它的行为。其他三种触发器也有定义其各自行为的特征表。每一种都能由 SR 触发器加上几个其他的门构成。像 SR 触发器一样，其他触发器都有 Q 和 $\overline{Q}$ 输出。JK 触发器有两个输入，标号为 J 和 K，而不是 S 和 R。D 和 T 触发器都只有一个输入。

从 SR 触发器构建其他触发器有一个系统的过程。一般电路通常包含图 11-9 的结构，SR 触发器的输出 Q 和 $\overline{Q}$ 作为构建中设备的输出 Q 和 $\overline{Q}$。对于 JK 触发器，图 11-9 的输入线

实际上是两条输入线，一条是J一条是K。对于D触发器来说，唯一的输入线标记为D，T触发器的输入标记为T。要设计每一个触发器，必须确定标记为组合电路的框中的逻辑门和它们的互联。

和任何组合电路设计一样，一旦确定了真值表形式中的输入和输出，那么就可以借助于卡诺图来构造最小的AND-OR电路。因此，第一步是写出图11-9中标记为组合电路的方框所需的输入和输出。确定电路描述的一个有用工具是图11-15中SR触发器的*激励表*（excitation table）。

Q(t)	Q(t + 1)	S(t)	R(t)
0	0	0	×
0	1	1	0
1	0	0	1
1	1	×	0

图11-15 SR触发器的激励表

对比激励表和图11-13的特征表，特征表告诉你给定当前输入和状态后接下来的状态是什么，但激励表告诉你如果想要得到某种转移，当前的输入必须是什么。下面讲述根据特征表怎样建立激励表。

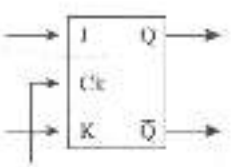
表中第一个条目是Q = 0到Q = 0的转移，两种可能的输入SR = 00和SR = 01允许这种转移。SR值为00是不改变的条件，SR值为01是重置的条件，这两个条件都会使触发器做出从Q = 0到Q = 0的转移。这个条目中R(t)下面的“×”是一个无关条件的值，只要S为0，不必关注R值是什么。不管R的值是什么，转移都是从Q = 0到Q = 0。 559

表中第二个条目是从Q = 0到Q = 1的转移，这个转移发生的唯一方法是SR的输入为10，即设置条件。类似地，第三个条目是从Q = 1到Q = 0的转移，这个转移只有在SR值为01时才会发生。

最后一个条目是从Q = 1到Q = 1的转移，允许这个转移的两种可能的输入条件：一个是SR = 00，即不改变条件；一个是SR = 10，即设置条件。无论S的值是什么，如果R为0，那么这个转移将会发生。S(t)下面的“×”说明了S的无关条件。

11.1.5 JK触发器

JK触发器解决了SR触发器中未定义的转移。J输入像S一样设置设备，K输入像R一样重置设备。但是当JK = 11时，这个条件叫作反转条件。反转意味着从一个状态转移到另一个状态。在反转条件下，如果初始状态为0，最终状态将为1；如果初始状态为1，最终状态将为0。图11-16是JK触发器的方块图和特征表。



a) 方块图

J(t)	K(t)	Q(t)	Q(t + 1)	条件
0	0	0	0	无变化
0	0	1	1	
0	1	0	0	重置
0	1	1	0	
1	0	0	1	设置
1	0	1	1	
1	1	0	1	反转
1	1	1	0	

b) 特征表

图11-16 JK触发器

对于JK触发器，图11-9中标记为组合电路的方框有三个输入和两个输出。输入是J、

K 以及来自反馈连接的 Q，输出是 S 和 R。所以要设计两个二输入的组合电路，一个用于 S，一个用于 R。首先，借助 SR 激励表写出图 11-17 的设计表。

设计表告诉你给定 JK 触发器要做出的转移时，SR 触发器必需的输入。前三列列出了组合电路的二个输入所有可能的组合。JK 的值最好排序，因为它们会显示在卡诺图中。第四列是时钟脉冲之后的 Q 值，每个 $Q(t+1)$ 的值来自于 JK 触发器的特征表。例如，在第三行中 JK = 11，这是触发条件，所以初始状态 $Q(t) = 0$ 反转为 $Q(t+1) = 1$。最后两列来自于给定 $Q(t)$ 和 $Q(t+1)$ 的 SR 触发器的激励表。例如，第三行是 $Q(t)Q(t+1) = 01$ 的转移，激励表显示要实现这个转移，SR 必须为 10。

下一步是写出这个函数的卡诺图。图 11-18a 中的条目来自于设计表中标号为 $S(t)$ 的那一列，图 11-18b 中的条目来自于设计表的 $R(t)$ 列。

Q(t)	J(t)	K(t)	Q(t + 1)	S(t)	R(t)
0	0	0	0	0	×
0	0	1	0	0	×
0	1	1	1	1	0
0	1	0	1	1	0
1	0	0	1	×	0
1	0	1	0	0	1
1	1	1	0	0	1
1	1	0	1	×	0

图 11-17　从 SR 触发器构造 JK 触发器的设计表

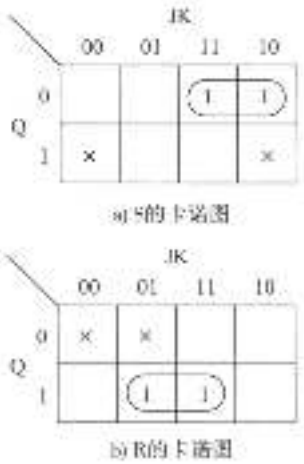


图 11-18　从 SR 触发器构造 JK 触发器的卡诺图

通过检查卡诺图，可以写出 S 的最小 AND-OR 表达 $S = J\overline{Q}$，R 为 $R = KQ$。图 11-19 是完整的设计。可以用一个 SR 触发器和两个二输入 AND 门来实现 JK 触发器。考虑 JK 所有可能的值可以看出这个设计是怎样工作的。如果 JK = 00，那么不管处于什么状态，SR = 00，这个状态不会改变；如果 JK = 11，Q = 0，那么 SR = 10，Q 将变为 1；如果初始 Q = 1，那么 SR = 01，Q 将会变为 0。两种情况下状态都会反转，对 JK = 11 也是同样。你应该也能分析出对于 JK = 01 和 JK = 10，这个电路也会正常工作。

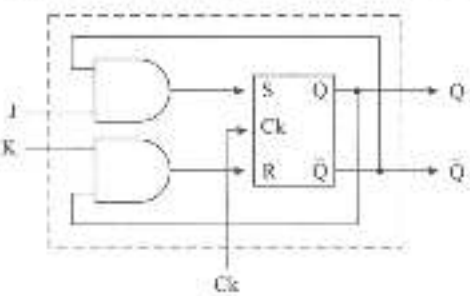
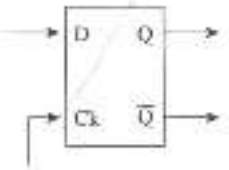


图 11-19　JK 触发器的实现

11.1.6　D 触发器

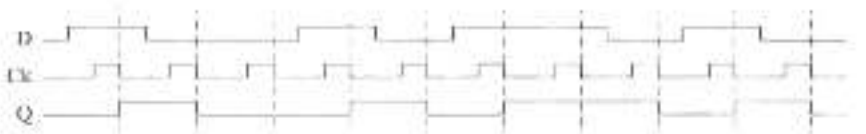
D 触发器是一个除了时钟之外只有一个输入 D 的数据触发器，图 11-20a 是它的方块图，图 11-20b 是它的特征表。表中可以看到 $Q(t+1)$ 与 $Q(t)$ 无关，它只取决于在时间点 t 的 D 值。D 触发器会存储该数据直到下个时钟脉冲。图 11-20c 展示的是时序图。这个触发器也叫作延迟触发器，因为在时序图中，Q 的形态和 D 是一样的，只是有一个时间延迟。



a) 方块图

D(t)	Q(t)	Q(t+1)	条件
0	0	0	延迟
0	1	0	
1	0	1	延迟
1	1	1	

b) 特征表



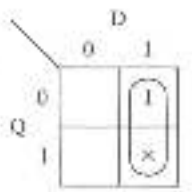
c) 时序图

图 11-20　D 触发器

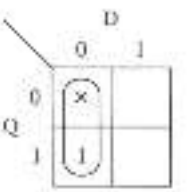
要从 SR 触发器构建一个 D 触发器，首先要写出设计表。因为除了 Q 以外只有一个输入，所以如图 11-21a 所示，表中只有四行。图 11-21b 和图 11-21c 展示的是卡诺图，只包含四个单元而不是八个。AND-OR 电路的简化得到 S = D 和 R = $\overline{D}$。

Q(t)	D(t)	Q(t+1)	S(t)	R(t)
0	0	0	0	×
0	1	1	1	0
1	1	1	×	0
1	0	0	0	1

a) 设计表



b) S的卡诺图



c) R的卡诺图

图 11-21　从 SR 触发器构造 D 触发器的设计表和卡诺图

图 11-22 是 D 触发器的实现。除了 SR 触发器，只需要一个反相器。这个实现方法没有像 JK 触发器的实现那样有来自 Q 或 $\overline{Q}$ 的反馈连接，这是因为下一个状态不依赖于当前的状态。

11.1.7　T 触发器

T 触发器是一个反转触发器。和 D 触发器一样，除了时钟外，它只有一个输入 T。图 11-23a 是方块图，图 11-23b 是特征表。T 输入就像一条控制线，指明是否进行选择性反转。如果 T 为 0，触发器不会改变状态；如果 T 为 1，触发器反转。T 触发器的实现作为章末的一道练习题。

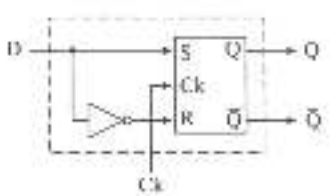
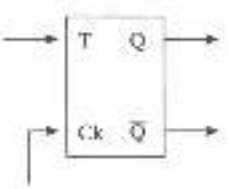


图 11-22　D 触发器的实现



a) 方块图

T(t)	Q(t)	Q(t+1)	条件
0	0	0	无变化
0	1	1	
1	0	1	反转
1	1	0	

b) 特征表

图 11-23　T 触发器

11.1.8　激励表

前面几节讲述了怎样用SR触发器和其他门构建JK、D和T触发器，也可以用同样的系统化过程从其他触发器来构建任何触发器。这看上去好像没什么意义，例如用JK触发器来构建D触发器，因为最开始我们是用SR触发器和两个额外的门来构建JK触发器的，而只需要一个SR触发器和一个反相器就能构造出D触发器。不过可以用任意触发器来构建其他触发器这一事实说明了所有触发器在能力上是等价的，即用某种触发器做的处理都可以用其他触发器以及几个额外的门来实现。

例如，假定有一个JK触发器，想构建一个T触发器，那么就要从T的特征表和JK的激励表写出设计表。通常来说，要从触发器B构建触发器A，需要A的特征表和B的激励表。图11-24展示的是JK、D和T触发器的激励表。

Q(*t*)	Q(*t* + 1)	J(*t*)	K(*t*)
0	0	0	×
0	1	1	×
1	0	×	1
1	1	×	0

a) JK触发器

Q(*t*)	Q(*t* + 1)	D(*t*)
0	0	0
0	1	1
1	0	0
1	1	1

b) D触发器

Q(*t*)	Q(*t* + 1)	T(*t*)
0	0	0
0	1	1
1	0	1
1	1	0

c) T触发器

图11-24　JK、D和T触发器的激励表

我们应该验证激励表的每个条目。例如JK表中第一个条目是Q = 0到Q = 0的转移。用与SR触发器一样的推理，如果JK为00或01，就会发生该转移；因此K(*t*)下面是无关条件。第二个条目也有一个无关条件。在这两种情况下会发生从Q = 0到Q = 1转移。JK可以为10设置条件，或者为11反转条件，两者都允许Q从0变为1。

11.2　时序分析和设计

时序电路是门和触发器的互联。理论上可以把所有门组合进一个组合电路中，把所有触发器组合进一组状态寄存器（state register）中，如图11-25所示。这是图11-9的一个概括，它只包含一个状态寄存器，它的输出不需要来自组合电路的附加的门。

图11-25中的实线箭头表示一条或多条连接线，输入和输出线是外部环境到电路平台的连接，从组合电路到状态寄存器的连线是到SR、JK、D或T触发器的输入线，反馈线是从触发器的Q和$\overline{Q}$输出到组合电路。这个图假设状态寄存器体中的每个触发器都使用一个通用的时钟线（未在图中显示）。

在时钟脉冲之间，电路的组合部分根据外部输入和电路的状态形成输出，即每个触发器的状态。产生组合电路输出和状态寄存器输入所花费时间的多少取决于电路中门级的数量。Ck周期被调整到足够长，以实现在下一个时钟脉冲之前允许输入通过组合电路直到输出。所有状态寄存器都是边沿触发或主－从型的，以防止多次通过反馈回路。

和图11-14一样，可以用状态转移图或与之对应的状态转移表来描述一个通用时序电路的行为。不同之处在于图11-14是一个有两种可能状态的设备，而图11-25是n个触发器，每个触发器有两种可能的状态，因此这个时序电路总共有2^n种状态。

硬件层的分析和设计之间的不同之处和软件层一样。图11-26说明了这种不同。在分析

中，输入和时序电路是给定的，要确定的是输出；而在设计中，输入和预期的输出是给定的，要确定的是时序电路。

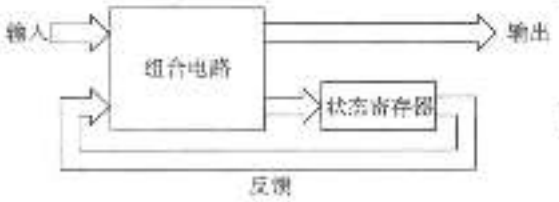


图 11-25　通用的时序电路

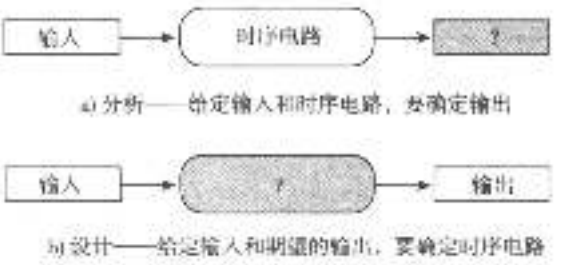


图 11-26　分析和设计的区别

本节展示了怎样由给定的时序电路和输入流来确定输出。总的思路是根据电路构建分析表，根据分析表就可以很容易地确定状态转移表、状态转移图和针对给定输入流的输出流。

11.2.1　时序分析问题

假设给定图 11-27 的电路，状态寄存器是两个 T 触发器，标识分别为 FFA 和 FFB。组合电路是两个 AND 门和一个 OR 门的组合，输入为 X1 和 X2，唯一的输出是 Y。反馈回路包括标识为 A 从 FFA 的 Q 发出的线路与标识为 B 和 $\overline{B}$ 两条从 FFB 的 Q 和 $\overline{Q}$ 发出的线路。到 FFA 的输入标识为 TA，到 FFB 的输入标识为 TB。

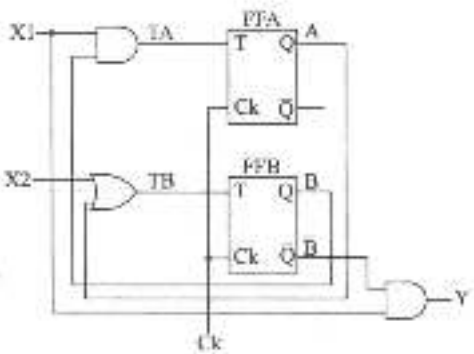


图 11-27　待分析的电路

因为有两个触发器，所以有四种可能的状态

AB = 00

AB = 01

AB = 10

AB = 11

在这里以 AB = 01 为例，它意味着 FFA 的 Q = 0 和 FFB 的 Q = 1。有两个输入，因此有四种可能的输入组合

X1 X2 = 00

X1 X2 = 01

X1 X2 = 10

X1 X2 = 11

问题来了。给定一个初始状态 AB 以及初始输入 X1 和 X2，(a) 初始输出是什么？(b) 一个时钟脉冲后下一个状态将会是什么？因为有四种状态，每个状态有四种可能的输入组合，所以需要回答这些问题 16 道。图 11-28 的分析表提供了一种回答这个问题的系统化工具。

前 4 列是初始状态和初始输入所有可能组合的列表。根据图 11-27，$Y(t)$、$TA(t)$ 和 $TB(t)$ 的布尔表达式是

$$Y(t) = X1(t) \cdot \overline{B}(t)$$

$$TA(t) = X1(t) \cdot B(t)$$

$$TB(t) = X2(t) + A(t)$$

560 / 565

因此，X1(t)列和B(t)列的补进行AND得到Y(t)列，X1(t)列和B(t)列的AND得到TA(t)列，X2(t)列和A(t)列的OR得到TB(t)列，根据T触发器的特征表，触发器的初始状态和它的初始输入可以计算出最后两列。

A(t)	B(t)	X1(t)	X2(t)	Y(t)	TA(t)	TB(t)	A($t+1$)	B($t+1$)
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	1
0	0	1	0	1	0	0	0	0
0	0	1	1	1	0	1	0	1
0	1	0	0	0	0	0	0	1
0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	0	1	1
0	1	1	1	0	1	1	1	0
1	0	0	0	0	0	1	1	1
1	0	0	1	0	0	1	1	1
1	0	1	0	1	0	1	1	1
1	0	1	1	1	0	1	1	1
1	1	0	0	0	0	1	1	0
1	1	0	1	0	0	1	1	0
1	1	1	0	0	1	1	0	0
1	1	1	1	0	1	1	0	0

图11-28　图11-27所示电路的分析表

566
|
567

例11.1　来看一下B($t+1$)列。在第一行中，根据B(t)列看到FFB的初始状态是0，从TB(t)列可以看到触发器的输入为0。根据图11-23b T触发器的特征表，0是无关条件，因此状态保持一致，B($t+1$)为0。□

例11.2　同样看这一列，来看第二行。根据B(t)列，FFB的初始状态又是0，这次根据TB(t)列，触发器的输入是1。根据T触发器的特征表，1是反转条件，因此状态反转，B($t+1$)为1。□

图11-29的状态转移表是从分析表选出几列并简单重新排列的结果。对于指定的初始状态A(t)B(t)和指定输入X1(t)X2(t)，它列出了接下来的状态A($t+1$)B($t+1$)和初始输出Y(t)。状态是一个有序对，表中的每个条目是下一个状态，后面跟着用逗号隔开的初始输出。

相比于状态转移表，通常状态转移图更容易表明电路的行为。图11-30是根据状态转移表构造出的状态转移图。标准惯例是用有序对来标识转移，有序对包括输入，后面跟着用斜线隔开的初始输出。

为了从给定的输入流确定输出，假定从状态AB = 11开始，输入下面的X1 X2值：

11，11，00，10，01

根据状态转移图，将转移到如下的状态

11，00，01，01，11，10

并产生如下的输出

0，1，0，0，0

	X1(t)X2(t)			
A(t)B(t)	00	01	10	11
00	00, 0	01, 0	00, 1	01, 1
01	01, 0	00, 0	11, 0	10, 0
10	11, 0	11, 0	11, 1	11, 1
11	10, 0	10, 0	00, 0	00, 0
	A(t+1) B(t+1), Y(t)			

图 11-29 图 11-27 所示电路的状态转移表

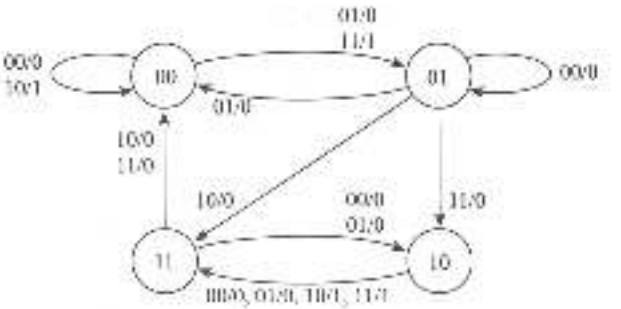


图 11-30 图 11-27 所示电路的状态转移表，转移被标记上 X1(t) X2(t) / Y(t)

对包含其他触发器甚至不同类型触发器混合的时序电路，分析也是类似的。使用组合电路的知识，可以轻松确定输入和状态所有可能的组合下，每个触发器的输入。然后，根据每个触发器的特征表可以确定接下来是什么状态。通常来说，如果有 m 个输入和 n 个触发器，就需要分析 2^m2^n 个转移。

11.2.2 预设置和清除

前面谈到问题的输出序列假设触发器都是在 Q = 1 这个状态。一个很自然的问题是：触发器怎样进入起始状态呢？实际上大多数触发器都有两个额外的输入，称为预设置（preset）和清零（clear）。这两个输入是异步的（asynchronous），即它们不依赖于系统的时钟脉冲便能工作。图 11-31 展示的是一个带异步预设置和清除输入的 SR 触发器的方块图。

在通常的运行中，预设置和清除线都是 0。要把触发器状态初始化到 Q = 1，就要把预设置设为 1 再变回 0；要把触发器初始化为 0，就要把清除设为 1 再变回 0。这两个输入中任一个只要为高电平就能起作用，不需要发送时钟脉冲。异步预设置和清除的实现作为章末的一道练习题。

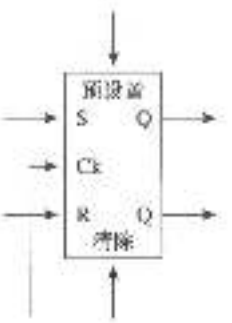


图 11-31 带异步预设置和清除的 SR 触发器的方块图

11.2.3 时序设计

采用时序设计，电路的行为常常以状态转移图的形式给出，需要确定如何用最少数量的门来实现该电路。同时，问题的描述中也会给出时序电路中使用的状态寄存器要采用哪种类型的触发器。

设计过程包括三步。第一步，根据状态转移图将电路的转移制成设计表。对于初始状态和输入的每种组合列出初始输出和下一个状态。然后，根据激励表列出导致转移所需的触发器输入条件。

第二步，将每个触发器输入的条目转移到卡诺图。如果在设计表的制表过程中提前考虑，以和卡诺图一样的顺序列出这些条目，将有助于防止出错。利用卡诺图最小化表达式，设计出时序电路中组合电路的部分。

第三步，画出简化后的组合电路。每个触发器的输入来自于深度最多为两个门级的组合电路。可以把这个过程看作是由 SR 触发器构建 JK、D 和 T 触发器过程的一般性概括。

568
569

11.2.4　一个时序设计问题

这个例子说明的是设计过程。问题是要用 SR 触发器实现图 11-32 的状态转移图。和图 11-30 一样，转移标上了输入值 X1 X2 和初始输出 Y。状态圈中的值是第一个 SR 触发器 FFA 和第二个 SR 触发器 FFB 的 Q 值。

这个机器只有三种输入组合和四种状态，总共 12 种转移。输入的组合是 01、11 和 10。由于组合 00 预期不会发生，因此可以把它看成无关条件帮助进行最小化。

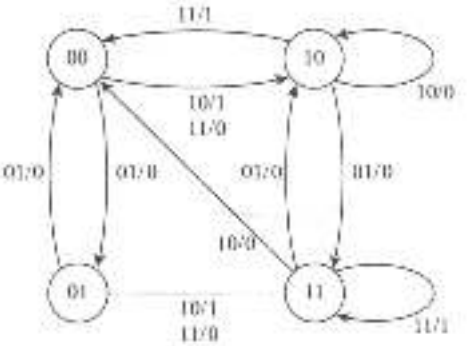


图 11-32　一个设计问题的状态转移图

要实现有八个状态的有限状态机需要三个触发器，实现有 5 ~ 7 个状态的机器也需要三个触发器，但是有些状态是预期不会发生的，可以看作无关状态。

570

图 11-33 是设计过程的第一步。左边四列是初始状态和输入所有可能的组合，中间三列是根据图 11-32 得出的初始输出和下一状态的简表。

初始状态		初始输入		初始输出	下一状态		触发器输入条件		
							FFA	FFB	
A(*t*)	B(*t*)	X1(*t*)	X2(*t*)	Y(*t*)	A(*t* + 1)	B(*t* + 1)	SA(*t*)	RA(*t*)	SB(*t*)	RB(*t*)
0	0	0	1	0	0	1	0	×	1	0
0	1	0	1	0	0	0	0	×	0	1
1	1	0	1	0	1	0	×	0	0	1
1	0	0	1	0	1	1	×	0	1	0
0	0	1	1	0	1	0	1	0	0	×
0	1	1	1	0	1	1	1	0	×	0
1	1	1	1	1	1	1	×	0	×	0
1	0	1	1	1	0	0	0	1	0	×
0	0	1	0	1	1	0	1	0	0	×
0	1	1	0	1	1	1	1	0	×	0
1	1	1	0	0	0	0	0	1	0	1
1	0	1	0	0	1	0	×	0	0	×

图 11-33　图 11-30 的状态转移图的设计表

右边有四列是因为有两个 SR 触发器，每个触发器有两个输入。SA 是 FFA 的 S 输入，RA 是 FFA 的 R 输入，以此类推。产生给定转移的触发器输入条件来自图 11-15 的 SR 触发器激励表。

例如，来看一下第一行从 AB = 00 到 AB = 01 的转移。FFA 的转移是从 Q = 0 到 Q = 0，从激励表可以看到转移由 S 为 0 和 R 为无关条件引发。FFB 的转移是从 Q = 0 到 Q = 1，从激励表可以看到这个转移是由 SR 为 10 导致的。

下一步，考虑每个触发器的输入是一个四变量函数——初始状态、AB 和输入 X1 X2。要设计这个组合电路，每个触发器输入需要一个四变量卡诺图。图 11-34 是根据图 11-33 得出的卡诺图，图 11-34a 是输入 SA 的卡诺图，可以看到行的值是状态 AB，列的值是输入

X1 X2。请注意 X1 X2 = 00 的组合，卡诺图第一列是无关条件。 571

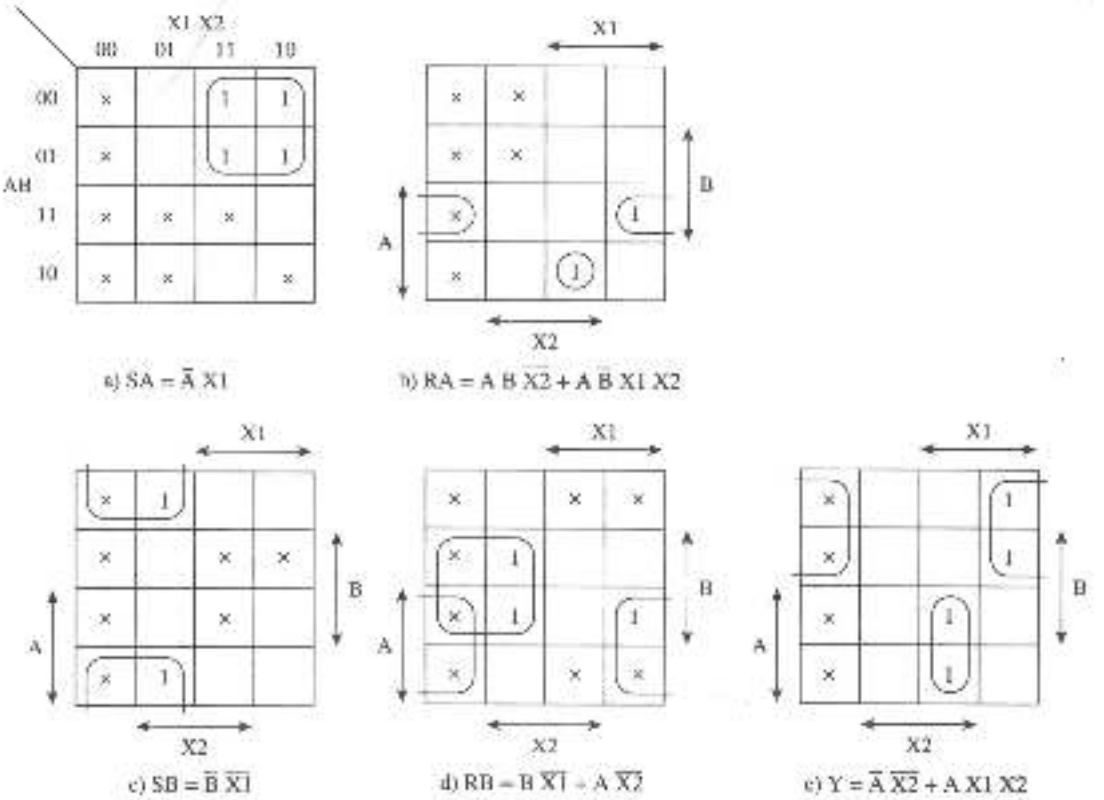


图 11-34　图 11-33 的卡诺图

输出 Y 也是一个初始状态和输入的函数，需要一个卡诺图来最小化。每个卡诺图下面是它对应的最小化表达式。

图 11-35 是设计得出的时序电路。这个图是个简略的形式，并没有画出反馈连接。

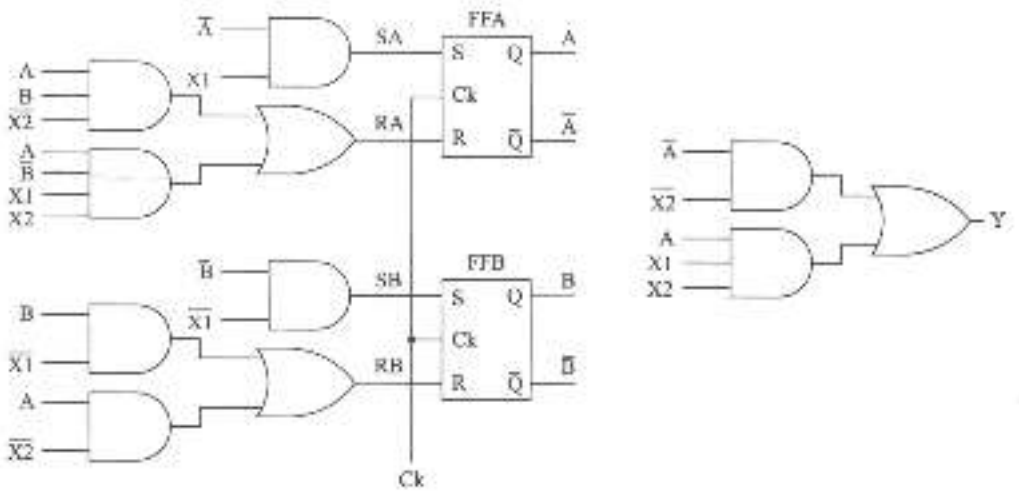


图 11-35　图 11-32 简化的时序网络

完成设计后，你可能注意到图 11-34c 和图 11-34d 中，卡诺图的单元 8 是被覆盖的，所以看上去值为 1。而图 11-34a 和图 11-34b 没有覆盖这个单元，所以它的值看上去为 0，怎么会这样呢？它怎么会同时既为 0 又为 1 呢？卡诺图没有展现出对无关条件实际上会发生什么。图 11-32 的电路描述假设外部输入 X1 X2 来自某个未知的来源，组合 X1 X2 = 00 在实际中绝不会发生。因此单元 8 代表的组合，即 A B X1 X2 = 1000 绝不会发生。可以给这个组合选择任何你想要的电路行为方式，因为它根本不会发生。

如果需要设计一个状态数量不等于 2 的幂的状态机，也可以用同样的方式思考。比如说想设计一个五个状态的状态机，两个触发器就不够了，因为它们只能提供四个状态。使用三个触发器，可能有八种状态，但实际中只会有五个状态发生，其他三种可以是无关条件。在设计了这样一个电路后，从有用状态不能到达无用状态，但反之不然，即可能有从无用状态到有用状态的转移，但是这样的转移是无关的，因为不会发生这样的转移，类似于程序中的死代码。

这个例子中描述的设计过程有两个触发器和两个输入，卡诺图中总共有四个变量。设计三个触发器一个输入或者一个触发器三个输入也需要四变量卡诺图。一个触发器两个输入或者两个触发器一个输入的设计过程也是一样，只不过用三变量卡诺图就足够做最小化了。

572 / 573

一些时序电路需要最小化超过四个变量的组合电路。卡诺图不能很方便地处理这样大小的问题。可用系统化过程处理此类问题，其中最常见的是奎因 - 麦克拉斯基（Quine-McCluskey）方法，不过这些方法都超出了本书讨论的范围。

11.3　计算机子系统

计算机是一组互联的子系统，每个子系统是一个黑盒子，有定义良好的接口。有时候子系统由一个单独的集成电路组成，这种情况中接口是由物理封装的引脚的运行特性描述的。在更低的抽象层级上，这个子系统可以是集成电路内多个子系统的一部分。或者在更高的抽象层级上，该子系统可以是一个印制电路板，由几个集成电路组成。

不过实际上子系统也是在物理上实现的，总线把各个子系统连接在一起。总线可以是一条线路，从子系统的一个门的输出到另一个子系统的一个门的输入，或者它也可以是包含数据和控制信号的一组线路，就像图 4-1 Pep/8 中连接主存到 CPU 的总线。

11.3.1　寄存器

ISA3 层机器的一个基础部件是寄存器。你应该熟悉 Pep/8 CPU 中的 16 位寄存器。指令集包括操控寄存器内容的指令。图 11-36 展示的是一个 4 位寄存器的方块图和实现。

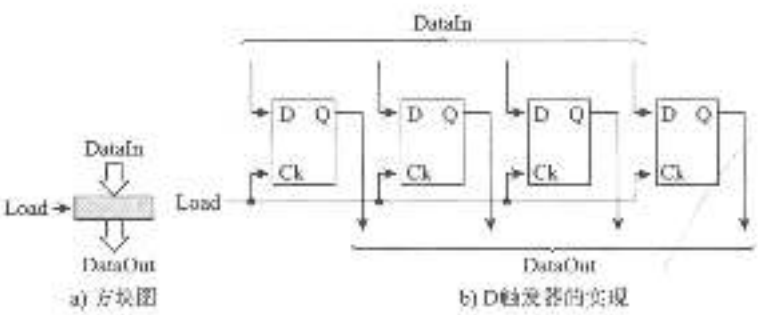


图 11-36　一个 4 位寄存器

这个寄存器有四个数据输入、四个数据输出和一个标记为 Load 的控制输入。方块图中用一个宽箭头表示四个数据输入线，实现中每条数据线是单独画出来的。这个寄存器很简单，就是一组 D 触发器，每条数据输入线连接一个触发器的 D 输入，每条数据输出线连接一个触发器的 Q 输出，Load 输入连接所有触发器的时钟输入。从现在开始，用阴影来表示包含时序电路的图，以区别于组合电路。

寄存器的运行要把希望加载的值放到数据输入线上，然后给装载信号发送 1 再变回 0，借助于时钟把数据装入寄存器：所有四个值同时装入寄存器。如果每个触发器都是主－从设备，那么在图 11-11 的时间点 t_4，当从触发器连接到主触发器时，将会出现输出。

在寄存器中每个 D 触发器是 1 位。一个 8 位寄存器有 8 个触发器，一个 16 位寄存器有 16 个触发器。触发器的数量不会影响装载操作的速度，因为每个触发器的装入是同时发生的。不管寄存器的位数是多少，它的方块图都和图 11-36a 所示的一样。

11.3.2 总线

假设两个子系统 A 和 B 需要来回发送数据。连接它们最简单的方法是用两条单向总线：一条发送从 A 到 B 的数据，一条发送从 B 到 A 的数据。第一条总线是一组线路，每一条线从 A 的一个门输出到 B 的一个门输入。第二条总线的每一条线从 B 的一个门输出到 A 的一个门输入。

这样安排的问题是对于宽总线来说线路的数量太大。如果想同时发送 64 位，就需要两条单向总线，总共 128 条线路。若使用双向总线，这个数量就能减半，但损失的是速度。用两条单向总线，可以同时从 A 向 B 和从 B 向 A 发送数据，而双向总线则不可能做到。如果要改变总线上数据流的方向，则必须在发送数据之前付出一小段建立时间的代价。

图 11-37 展示了实现双向总线必须要解决的问题。AND 门表示作为寄存器一部分的主－从触发器的主时钟使能门，NOR 门代表另一个寄存器的触发器的从 Q 输出门。要从 A 到 B 发送数据，那么门 2 的输出必须连到门 3 的输入。从 B 到 A 发送数据，门 4 的输出必须连到门 1 的输入。问题出在门 2 和门 4。总是可以把一个门的输出连到另一个门的输入，但是不能把两个门的输出连接到一起。假定门 2 想发送一个 1 到门 3，但同时门 4 的输出刚好为 0，它们的输出是冲突的，那么哪一个会占优呢？

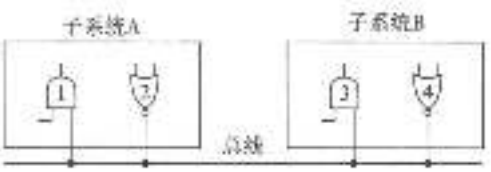


图 11-37 双线总线的问题

答案取决于底层构造门的技术。对于一些逻辑门族，实际上可以把几个门的输出连接到一起，如果一个或多个门输出 1，那么公共的总线将传送 1。由于总线上的信号就像 OR 门的输出，因此称这种门具有线－OR 属性（wired-OR property）。对于其他逻辑门族，把两个门的输出连接到一起会使电路崩溃，引发不可预知的灾难。即使线－OR 门也仍然存在双向总线的问题。例如，如果门 2 想发送 0 到门 3，但是此时门 4 的输出碰巧是 1，那么门 3 就会错误地检测到一个 1。

为了使双向总线正常工作，需要找到一种方法，当门 2 往总线上放数据时让门 4 临时与总线断开连接，反之亦然。三态缓冲器可以精确地完成这个工作，它有一个数据输入，一个使能控制输入和一个输出。图 11-38 是三态缓冲器的真值表，这里的 E 是使能控制线，a 是输入，x 是输出。当设备使能时，输入不变地到达输出；当设备禁止时，输出实际上是和电

路断开的。从电子学角度来讲，输出这时处于高阻抗的状态。因为输出可以有三种状态——0、1和断开，因此这种设备叫作三态缓冲器。

图11-39展示的是三态缓冲器怎样解决双向总线问题。每个门的输出和总线之间有一个三态缓冲器。要从A到B发送数据，使能A中的三态缓冲器，禁止B中的三态缓冲器，反之亦然。要使这个机制能正确发挥作用，子系统必须协调工作，不能让它们的三态缓冲器同时使能。

E	a	x
0	0	断开连接
0	1	断开连接
1	0	0
1	1	1

图11-38　三态缓冲器的真值表



图11-39　使用三态缓冲器的双线总线问题解决方案

574
↓
576

11.3.3　内存子系统

内存子系统由几个集成电路内存芯片构成，图11-40展示的是两个内存芯片，每一个可以存储512位。内存芯片有一组标记从A0开始的地址线，一组表记从D0开始的数据线，一组标记为CS、WE和OE的控制线。数据线有两个箭头，表示它们应该连接到双向总线。图11-40a表明内存的位是64个八位的字。图中有6条地址线，因为2^6=64，输入值的每种可能组合会访问一个独立的8位字。图11-40b展示的是同样数量的位被组织成512个一位的字。图中有9条地址线，因为2^9=512。一般来说，有2^n个字的内存芯片就有n条地址线。为了保持例子简单，图11-40中芯片地址线和数据线的数量少到不切实际，现在制造的内存芯片都有上亿位。

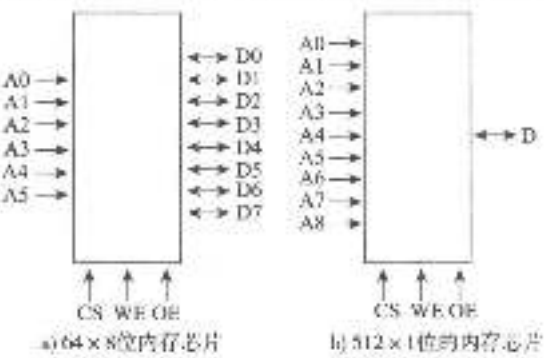


图11-40　两种存储512位的集成电路内存芯片

控制线有如下作用：

- CS（chip select，芯片选择）使能或选择内存芯片。
- WE（write enable，写使能）把一个内存字写入或存储到芯片中。
- OE（output enable，输出使能）使能输出缓冲器，从芯片读取一个字。

要把一个字存储到芯片，就要把地址线设置为这个字要存储的位置，把数据线设置为想存储的值，把CS设置为1来选择这个芯片，再把WE设置为1来执行写入。从芯片中读取一个字，要把地址线设置为要读取的地址，把CS设置为1来选择该芯片，把OE设置为1使输出使能，这样想读取的数据就会出现在数据线上了。在实际中大多数芯片的控制线都是低电平有效的，即它们平常维持在表示为1的高电平，把它们设置成表示为0的低电平即可激活。为了例子简单，本书假定内存芯片的控制线为高电平有效。

577

图11-41是一个4×2位内存芯片的实现，它有两条地址线和两条数据线。存储4个2位的字，每位是一个D触发器。用阴影来表示时序设备以区别于组合设备。地址线驱动一

个 2×4 译码器，它的某个输出为 1，其余三个为 0。译码器为 1 的那条输出线选择一行 D 触发器，这行触发器组成芯片要访问的字。

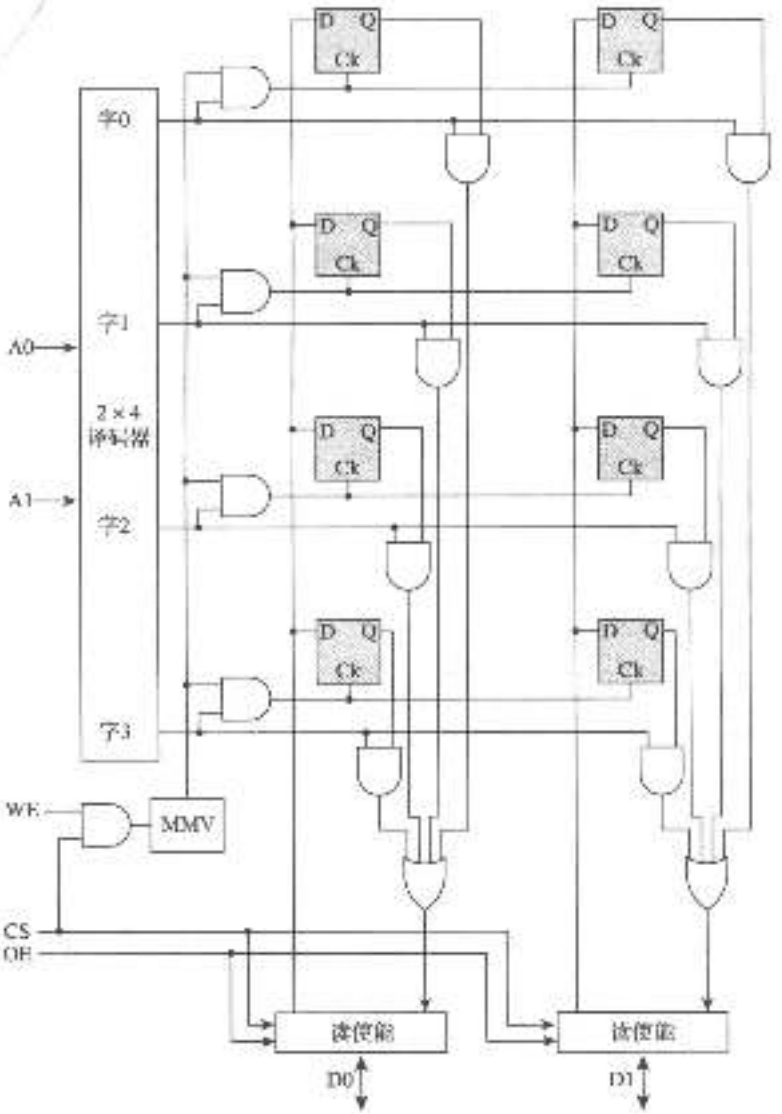


图 11-41　一个 4×2 位的内存芯片

标记为“读使能”的方框提供对双向总线的接口，图 11-42 是它的实现。DR 是数据读取线，来自于图 11-41 中的 OR 门，DW 是数据写入线，去往触发器的 D 输入，D 是到双向总线的数据接口。

图 11-43 是这个电路的真值表。芯片通常是下面三种模式之一：

- CS = 0：芯片未被选中。
- CS = 1，WE = 1，OE = 0：芯片被选中写入。
- CS = 1，WE = 0，OE = 1：芯片被选中读取。

不允许 WE 和 OE 同时都为 1。从真值表和它的实现可以看

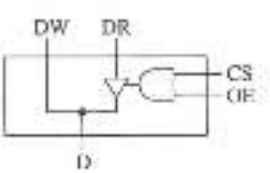


图 11-42　图 11-41 中标记为“读使能”的方框的实现

到当 CS 为 0 时，不管其他控制线如何，DR 总是从双向总线断开的；当 CS 为 1、OE 为 0 时，DR 也是断开的，这就是写入模式，这种情况下数据从双向总线送到 DW；当 CS 为 1、OE 为 1 时，三态缓冲器是使能的，数据从 DR 送到双向总线。

CS	OE	操　作
0	×	断开连接
1	0	断开连接
1	1	将 DR 连接到 D

图 11-43　"读使能"方框的真值表

来看一下内存读取是怎样进行的，考虑一个场景：A1 A0 =10，CS = 1，WE = 0 和 OE = 1。A1 A0 的值使得从译码器发出的标号为"字 2"的线为 1，其他字线为 0。578 / 579 "字 2"线使能连接第 2 行 D 触发器的 Q 输出的 AND 门，禁止连接其他所有行触发器输出的 AND 门。因此来自第二行的数据流过两个 OR 门到达"读使能"方框，进入双向总线。

内存的写入要借助于标记为 MMV 的方框，图 11-41 中 MMV 代表单稳多谐振荡器。假定 D 触发器是主 - 从触发器的变种，存储需要一个 Ck 脉冲从低电平到高电平然后再从高变回低，如图 11-11 所示。单稳多谐振荡器（monostable multivibrator）就是一个提供这样脉冲的设备。图 11-44 展示的是有初始延迟的单稳多谐振荡器的时序图。当输入线升高时就会触发一个延迟电路，经过一个预设的时间间隔后，延迟电路触发单稳多谐振荡器，发出具有预设宽度的时钟脉冲。单稳多谐振荡器激活时会发出一个"单次"（one shot）脉冲，因此它也称为单次设备。

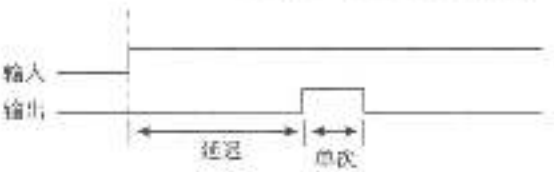


图 11-44　带初始延迟的单稳多谐振荡器的时序图

来看一看内存写入是如何进行的，考虑这样一个场景：A1 A0 =10，CS = 1，WE = 1 和 OE = 0。假设地址线、数据线和控制线的设置是同时发生的，内存电路必须要等待地址信号传送过译码器，然后才能根据时钟脉冲把数据送入触发器。工程上设计 MMV 中的初始延迟，就是为了留出足够的时间使得在根据时钟送入数据之前译码器的输出能够设置好。"读使能"电路把来自双向总线的数据放到所有触发器的输入。不过当 MMV 发出时钟脉冲时，它连接的四个 AND 门中有三个将会禁止脉冲到达它们所在的行，脉冲只能到达"字 2"那一行，所以这是能存储数据的唯一的触发器。

市场上有几种类型的内存芯片，图 11-41 中的电路模型与称为静态内存（static memory）或 SRAM 的内存是最为相似的。实际上，主 - 从 D 触发器并不是位存储器的基础，因为它使用了不必要的晶体管。许多静态 RAM 设备使用的电路是对图 11-1b 的修改，即一个由一对带反馈的反相器组成的稳定电路，它只需两个额外的晶体管来实现设置状态的机制。静态 580 RAM 的优势是速度，但由于每个位单元都需要几个晶体管，因此劣势是芯片尺寸。

为了克服静态内存尺寸的劣势，动态内存（dynamic memory）或 DRAM 的每个位单元只使用一个晶体管和一个电容，通过在电容上存储电荷来存储数据。由于这种位单元的尺寸小，因此 DRAM 芯片的存储容量要比 SRAM 芯片大很多。DRAM 的问题在于电容中的电荷在充满几毫秒之后会慢慢泄漏。在过多的电荷泄漏之前，内存子系统必须要从单元读取数据，如有必要还要给电容充回电荷。和预期的一样，刷新操作要花费时间，所以 DRAM 内存要比 SRAM 内存慢。

相比于读 / 写内存，只读内存（read-only memory）或 ROM 用于存储不会改变的数据。在工厂生产时，芯片的每个位单元的数据就设置好了，这种情况下用户向厂商提供要存储的位模式。除此之外，还有可编程 ROM（programable ROM）或 PROM 芯片，允许用户编写

位模式。这个过程通过有选择地断开内嵌在芯片内的一组保险丝来完成，是不可逆转的。为了解决不可逆转这个缺点，可擦写 PROM（erasable PROM）或 EPROM 可以通过把电路曝光在紫外线下擦除整个芯片。EPROM 芯片封装在一个透明窗口下，这样电路可以暴露在射线中。要擦除 EPROM 芯片就必须把它从计算机中拿出来进行曝光，然后对整个芯片重新编程。电子可擦除 PROM（electrically erasable PROM）或 EEPROM 允许用户用合适的电子信号组合来擦除单个单元，这样就不用把设备取出来，也不用重新编程。对单元进行编程的电路使用的电压不同于在芯片正常操作期间读取数据的电压，因此设计起来会更加复杂。

SRAM 和 DRAM 是易失的，也就是说关掉电路的电源时，数据就会丢失。ROM 设备是非易失的，因为它们维持数据不需要外部电源。对于数码相机、手机和 MP3 播放器这样的手持消费设备来说，闪存（flash memory）是非常受欢迎的。它是一种 EEPROM，具有设备掉电后仍然保持数据这个优势。对于闪存，用户可以读取单个单元，但是只能写入整个单元块。在写单元块之前，必须将其完全擦除。闪存卡由一组闪存芯片组成，闪存驱动器与之类似，只是接口电路使其看上去像一块硬盘。但它实际上并不是硬盘，也没有移动部件。相比于同样大小的硬盘，闪存驱动器速度更快，但存储的数据量要小很多。微硬盘和闪存技术在市场上相互竞争，厂商现在提供封装好的硬盘，它的接口使它表现得如同闪存，你可以把它插入数码相机来替代内存卡。现在有看上去像硬盘的闪存和看上去像闪存卡的硬盘，这是抽象在现实中得以运用的例证。 581

Lynn Conway 和 Carver Mead

早期计算机由晶体管、电阻器和电容器组成。之后主要的进步是小规模集成电路（SSI），一个芯片上最多制造 10 个门。后来出现了中规模集成电路（MSI），一个芯片上最多可以制造 100 个门，然后是大规模集成电路（LSI）和超大规模集成电路（VLSI），一个芯片上分别最多有 10 000 个门和 100 000 个门。现在是特大规模集成电路（ULSI）时代，一个芯片上最多有 10 000 000 个门。

20 世纪 70 年代中期，半导体工业达到 LSI。大的制造商生产出一系列 LSI 芯片，用户购买并连接这些芯片以组成数字系统。集成度的进一步提高则超出了个人或小公司的能力范围，因为这样的系统看上去太过复杂。

不过有两个人准备为半导体工业带来根本性的巨大变革。Lynn Conway 1963 年在哥伦比亚大学获得电子工程专业硕士学位，她在 IBM 研究院工作过一段时间，发明了 ACS-1 项目中的动态指令调度，后来又在其他公司工作过，1973 年进入施乐（Xerox）公司，在令人激动的新 Palo Alto 研究中心工作。Carver Mead 自称 Caltech（加州理工学院）的“无期徒刑者”，他的所有学位都是在那里取得的，直至 1959 年获得电子工程专业博士学位，并且留校执教 40 余年。

Conway 和 Mead 于 1975 年开始合作。Conway 从事硅结构项目，需要及时设计出电子电路，若使用当时的 LSI 芯片，这将是个烦琐耗时的任务。

她和 Mead 一起提出了一个强大的想法——只要抽象层次得当，VLSI 电路设计可以用计算机自动产生。他们归纳出硅设计原则，这样 VLSI 设计者不用再面对单个逻辑门，而只需处理较高抽象层次上的黑盒子。通过隐藏低层的细节，小的设计团队甚至个人都能够设计出自己定制的 VLSI 电路，而不必限制在当时已有的 LSI 芯片上。

工业界的反应是非常怀疑。Mead 预言定制的 VLSI 电路由每个需要芯片的组织自己

设计，然后交付给"硅铸造厂"制造。为了推广这个理念，Mead 和 Conway 在 1979 年写了一本书——《VLSI 系统入门》(Introduction to VLSI Systems)。Conway 1978 ~ 1979 年在 MIT 做客座教授，在一门课程中使用了这本书的样书，她要求学生在一个学期里设计和构造一个 VLSI 芯片。在这门著名的课程获得成功之后，这本书在超过 100 所学校中使用，这些学校的学生最终永远地改变了半导体工业——硅铸造厂现在是每年超过 300 亿美元的产业。





Lynn Conway 获得了许多奖项，包括当选 IEEE Fellow（IEEE 会士）和美国国家工程院院士。Carver Mead 获得的奖项包括 Harry Goode 纪念奖和国家科技勋章。

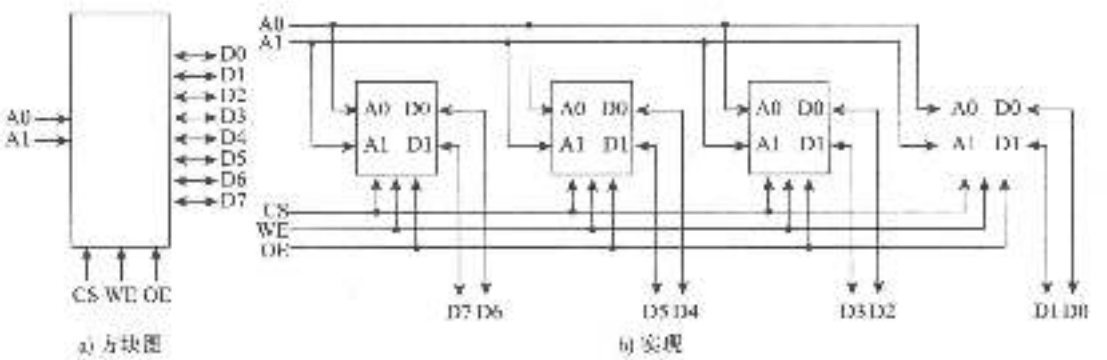
"聆听科技的声音，听清它在诉说什么。"

——Carver Mead

11.3.4　地址译码

单个内存芯片通常不能提供整个计算机所需的主存，必须把几个芯片组合成内存子系统才能提供足够的存储能力。大多数计算机是按字节寻址的，Pep/8 也是一样。像图 11-40a 那样的芯片对于这样的机器来说是非常方便的，因为芯片字的大小和 CPU 寻址单元的大小是一致的。

不过，假设有图 11-41 所示的一组 4 ×2 芯片，想把它用在 Pep/8 中。芯片的字大小为 2，CPU 的可寻址单元大小是 8，所以必须把四个 4×2 的芯片组合构成一个 4×8 的内存模块。图 11-45 给出了连接方式，可以看到这个模块的输入和输出线与一个 4×8 芯片的输入输出线是完全一样的。内存中每个字节的位分布在四个芯片上，地址 A1 A0 = 01 的字节的每个位存储在所有四个芯片的第二行（字 1）。



a) 方块图　　b) 实现

图 11-45　从四个 4×2 的内存芯片构造一个 4×8 的内存模块

类似地，要构建一个 512×8 的内存模块就需要 8 个图 11-40b 所示的芯片。为了获得高可靠性，单元为八位需要使用 11 个芯片，三个额外的芯片用于单错误纠错，如同 9.4 节描述的那样。对于这样的 ECC 系统，每个字节的位将会分布于 11 个芯片。

这些例子展示了怎样把几个 $n \times m$ 芯片组合成一个 $n \times k$ 芯片模块，这里的 k 大于 m，通

常来说，k 必须是 m 的倍数。可以简单地把 k/m 个芯片所有公用的地址线和控制线连接起来，把每个芯片的数据线分配给模块的数据线。

构建内存子系统的另一个问题是，现有几个 $n \times m$ 芯片，m 等于 CPU 可寻址单元的大小，想构建一个 $l \times m$ 的模块，这里的 l 大于 n。换句话说，如果有一套芯片，它的字大小等于 CPU 可寻址单元的大小，怎样连接它们并把内存加到计算机中呢？关键是要使用芯片选择线 CS，使得来自 CPU 的所有地址请求只能选择一个芯片。把内存芯片连接到地址总线的技术叫作地址译码，它有两种类型：全地址译码和部分地址译码。

图 11-46 展示的是一个有 8 条地址线、能存储 2^8 或 256 字节的 CPU 内存图，为了保持例子简单，这个场景小得不切实际。有四个芯片需要连接进 CPU 的地址空间：安装在地址 0 的 64 字节 RAM，安装在地址 64 的 32 字节 RAM，安装在地址 208 的 8 端口 I/O 芯片和安装在地址 224 的 32 字节 ROM。

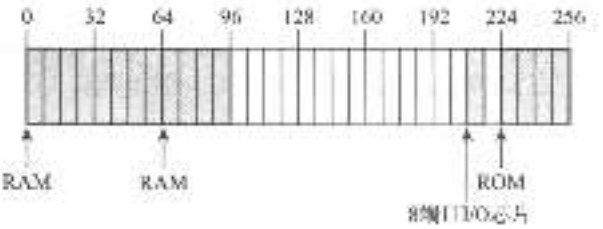


图 11-46 一个具有八位地址线的 256 字节内存的内存映射

8 端口 I/O 芯片实现的是称为内存映射（memory-mapped）的 I/O，这在很多计算机系统中都很常见。指令集里有原生 I/O 指令的计算机（例如 Pep/8），不需使用内存映射 I/O。Pep/8 的原生输入和输出指令是 CHARI 和 CHARO。当执行 CHARI 时，CPU 从 CHARI 指令未明确指定的输入设备获取输入，并把这个字节存储在操作数指示符和寻址模式确定的内存位置上。相比之下，采用内存映射 I/O 系统的指令集不需要任何 I/O 指令，CPU 的 LOAD 指令完成输入，STORE 指令完成输出。在图 11-46 中，8 端口 I/O 芯片将 8 个 I/O 设备映射至 208 ~ 215 的内存地址。例如，键盘可能映射到 208，显示器映射到 209，打印机映射到 210，等等。在没有内存映射 I/O 的计算机上，例如 Pep/8，从键盘输入一个字节的程序是

```
CHARI char,s
```

但是在如图 11-46 中所配置的有内存映射 I/O 的计算机上，从键盘输入一个字节是

```
LDA 208,d
STA char,s
```

582 | 584

类似地，可以用如下存储指令将一个字符发送到打印机

```
LDA char,s
STA 210,d
```

为使内存映射 I/O 正常工作，系统需要电路能检测 I/O 设备映射到的内存地址的装入和存储是否已经完成。检测这类事件会激活必要的电路来控制 I/O 设备。

图 11-46 中的每个 RAM 和 ROM 芯片都有 8 条双向数据线。不过 8 端口 I/O 芯片有 64 条数据线。第一组 8 条数据线把键盘连接到地址 208，第二组 8 条数据线把显示器连接到 209，以此类推。

可以根据图 11-47 的表来确定怎样将芯片连接到地址总线。对于每个芯片，以二进制写出最小的地址，即芯片起始字节的地址，和最大地址，即芯片最后一个字节的地址。比较这两个位模式，就可以确定每个芯片对应地址范围的通用格式。例如，8 端口 I/O 芯片的通用地址是 1101 0xxx，这表示对应的地址范围从 1101 0000 到 1101 0111，每个字母 x 可以是 0 也可以是 1，所以当且仅当前 5 位数字是 11010 时，选中的一定是这个 8 端口 I/O 芯片。

设　备	64×8 RAM	32×8 RAM	8 端口 I/O	32×8 ROM
最小地址	0000 0000	0100 0000	1101 0000	1110 0000
最大地址	0011 1111	0101 1111	1101 0111	1111 1111
通用地址	00xx xxxx	010x xxxx	1101 0xxx	111x xxxx

图 11-47　对图 11-46 中内存映射做地址译码的表格

图 11-48 展示的是用全地址译码方式把芯片连接到地址总线。8 端口 I/O 的三条地址线连接到地址总线的三条最低位线上，5 条最高地址线通过一对反相门和一个 AND 门注入芯片选择端（图中对反相器做了简化，显示成 AND 门的反相输入）。从这个电路可以看出，当且仅当总线上前五位为 11010 时，该 8 端口 I/O 芯片的芯片选择线将会是 1。

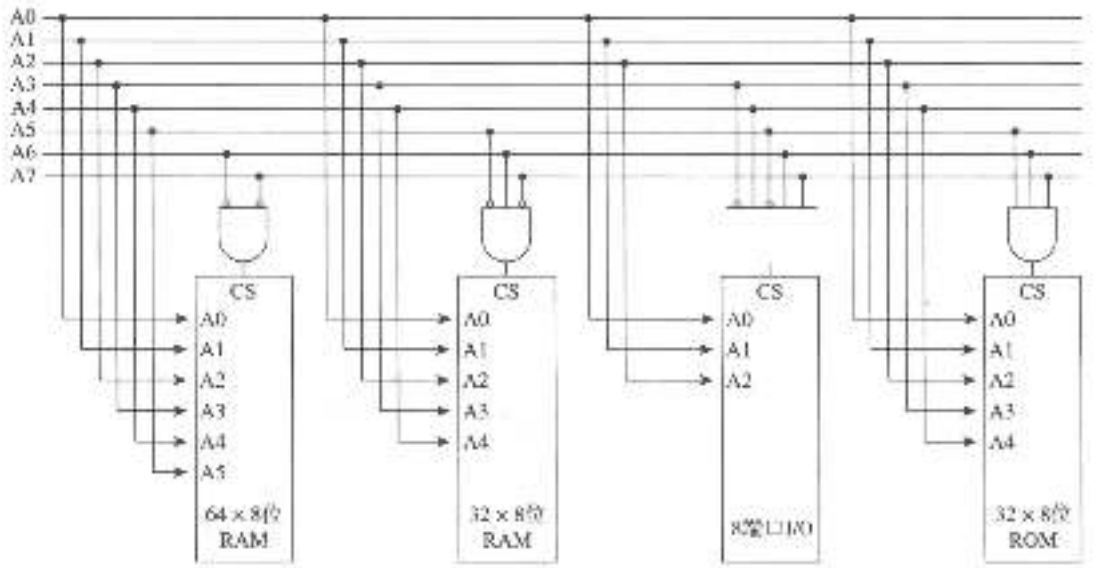


图 11-48　对图 11-46 中内存块射做全地址译码

为了保持图的简洁，在图中没有显示芯片的数据线。RAM 和 ROM 芯片的数据线都连接一个 8 位双向数据总线。图中也没有显示控制输入 WE 和 OE，它们连接内存模块公用的 WE 和 OE。

如果知道某个内存子系统绝不会通过增加更多内存进行升级，那么就可以使用部分地址译码。典型的场景是一个小型的不支持用户升级的计算机控制装置，它的主要思想是在译码电路中把门的数量减少到刚好满足系统访问设备的需要。

这种简化技术就是写出芯片的通用地址，每行一个，然后检查列。对于图 11-47 中的芯片，通用地址为

00xx xxxx，64×8 位 RAM

010x xxxx，32×8 位 RAM

1101 0xxx，8 端口 I/O 芯片

111x xxxx，32×8 位 ROM

来看看第一个芯片，检查通用地址的列，看如何用最少量的信息来唯一确定第一个芯片。可以看到第一个芯片的第二列对应地址线 A6 为 0，其他芯片的该位为 1，因此不管 A7

的值是什么，只要 A6 为 0，就可以选择第一个芯片。

现在来看第二个芯片，若用全地址译码就必须测试三条地址线：A7、A6 和 A5。只测试两条可以吗？例如，能测试 A7 A5 = 00 吗？不能，因为 A7 A6 A5 = 000 将会选择第一个芯片，那么会同时选择两个芯片。能测试 A6 A5 = 10 吗？不能，因为 A7 A6 A5 A4 A3 = 11010 会选择 8 端口 I/O 芯片，又有冲突。不过，通过观察所有列可以发现没有其他芯片的前两位为 01，因此可以通过测试 A7 A6 = 01 来选择第二个芯片。

585
↓
586

通过类似的推理可以看到，可测试 A7 A6 A5 = 110 来选择第三个芯片，测试 A7 A6 A5 = 111 来选择第四个芯片。图 11-49 展示了简化的最终结果。相比于图 11-48，我们去掉了一个两输入 AND 门和三个反相器，将一个 AND 门的输入从 3 个减少到 2 个，另一个 AND 门的输入从 5 个减少到 3 个。

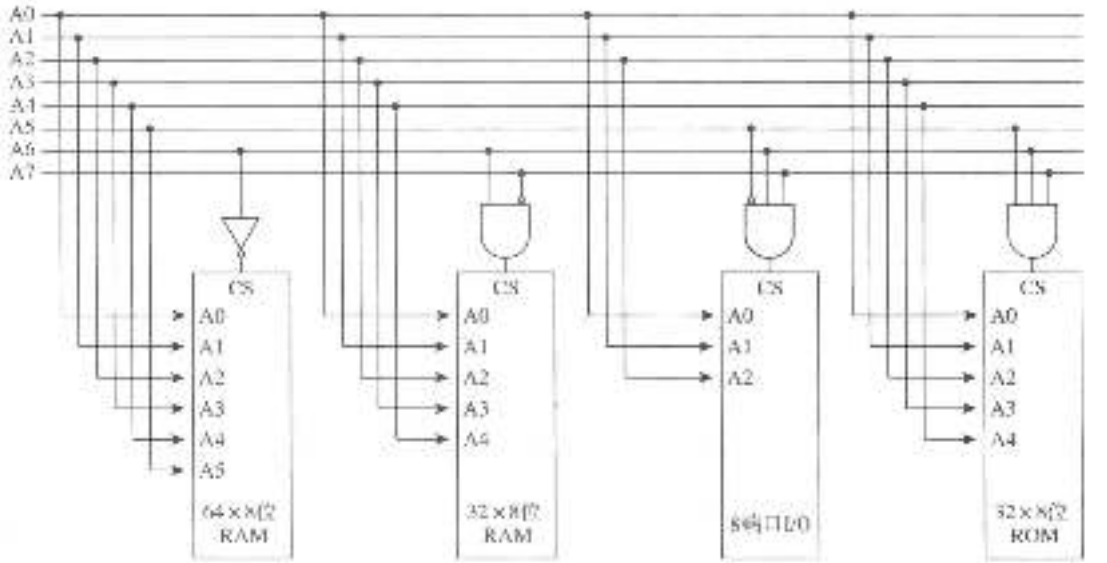


图 11-49 对图 11-46 中内存映射做部分地址译码

这自然会产生一个问题：图 11-48 和图 11-49 的内存模块行为有什么不同？当 CPU 访问图 11-46 内存图的阴影区域之一时，没有任何不同。使用全地址译码时，如果 CPU 访问阴影区域之外的地址，则没有芯片被选中，数据总线上的数据是不可预测的。然而使用部分地址译码，CPU 可能会访问某个芯片。

来看 64×8 位的 RAM，它的通用地址为 00xx xxxx，如果 A6 为 0，它会被选中，分为两种情况：地址请求为 00xx xxxx 和 10xx xxxx。第一个地址范围是设计好的，但是第二个地址范围是部分地址译码的副作用，实际上是把一个物理设备映射到地址空间的两个区域，CPU 在地址 0 和 128 看到的是一个芯片的克隆。类似的推理显示 32×8 位 RAM 在地址 96 又复制了一次，8 端口 I/O 芯片在三个地方进行了复制。ROM 没有复制，因为对于全地址和部分地址译码，它的地址译码电路是一样的。图 11-50 展示的是部分地址译码的内存图。

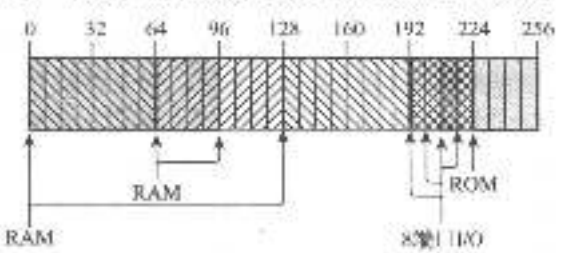


图 11-50 图 11-49 中部分地址译码的内存映射

使用部分地址译码时必须小心。在这个例子中，芯片以无缺口、无重复、完全填充内存图的方式进行复制。如果你的译码在得到的映射图中留下缺口，这不会带来什么危害。如果CPU访问的是空白区域，那么没有芯片会被选中。然而如果你的译码产生了重复，那就意味着如果CPU要访问地址空间的这个区域，那么会有多个芯片被选中，这对系统是有害的。当然，可以假设CPU绝对不会访问这个区域，就像它可以访问真正的芯片就不会访问复制的芯片一样。但是怎么可以假设程序中不会有错误呢？

11.3.5　双端口寄存器体

在前一节介绍的所有内存子系统中，一组地址线对应的都只有一组数据线。这种组织结构对于计算机主存子系统来说是合适的，因为主存和CPU通常不在同一集成电路上。图4-2展示的是Pep/8 CPU中的寄存器，它们的组织结构很像内存子系统，但是位于CPU中的寄存器体里面。CPU中的寄存器体在以下两方面不同于内存芯片的存储结构：

- 数据总线是单向而不是双向的
- 有两个而不是一个输出端口

图11-51展示的是地址从0～31的32个8位寄存器，前5个寄存器对ISA层的程序员是可见的。每个16位寄存器分成两个8位寄存器，这种划分对机器层的程序员是不可见的。

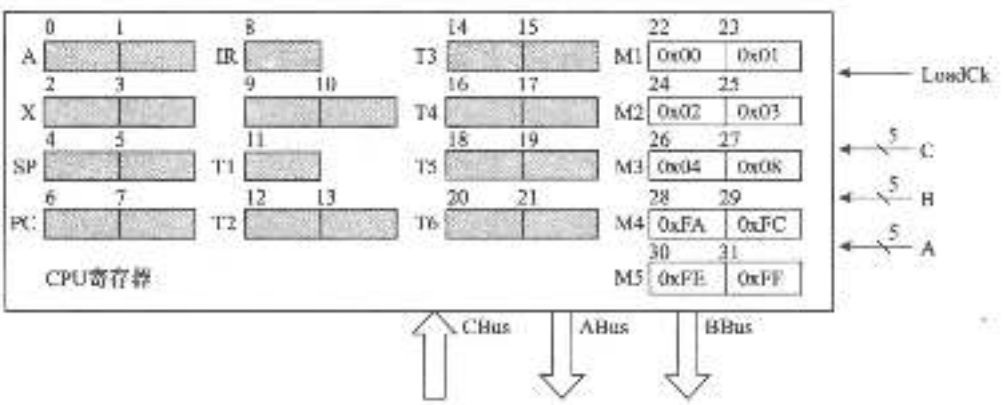


图11-51　Pep/8 CPU中的32个8位寄存器

其余寄存器（地址从11～31）对机器层程序员是不可见的。寄存器11～21用于存储临时变量；22～31是包含定值的只读寄存器，类似于ROM，如果想存储，寄存器中的值不会改变，这些常数值以十六进制给出。只读寄存器实际上不是时序电路，因此没有用阴影表示。它们没有可以改变的状态，因此更像是组合电路。

主存芯片有一组地址线和一组双向数据线，而寄存器体有三组地址A、B和C，以及三条单向数据总线ABus、BBus和CBus，ABus和BBus是两个输出端口，CBus是输入端口。每条数据总线都是8位宽，每组地址线都包含5条线路，能够访问2^5个寄存器中的任何一个。要把一个值存储到寄存器中，就要把寄存器的地址放到C上，把待存储的数据放到CBus上，给标记为LoadCk的控制线时钟信号。通过两个输出端口可以同时读取两个不同的寄存器，可以把一个地址放在A上，一个放在B上，来自这两个寄存器的数据将同时出现在ABus和BBus上。也允许把同一地址放在A和B上，这样来自同一个寄存器的数据将出现在ABus和BBus上。

图 11-52 展示的是双端口寄存器体的实现。输入遵循和主存芯片一样的基础组织结构，C 的 5 条地址线使译码器的一条输出线为 1，其余为 0。CBus 连接全部 32 个寄存器，当 LoadCk 控制线有脉冲时，CBus 上的值就随时钟信号存入某一个寄存器中。

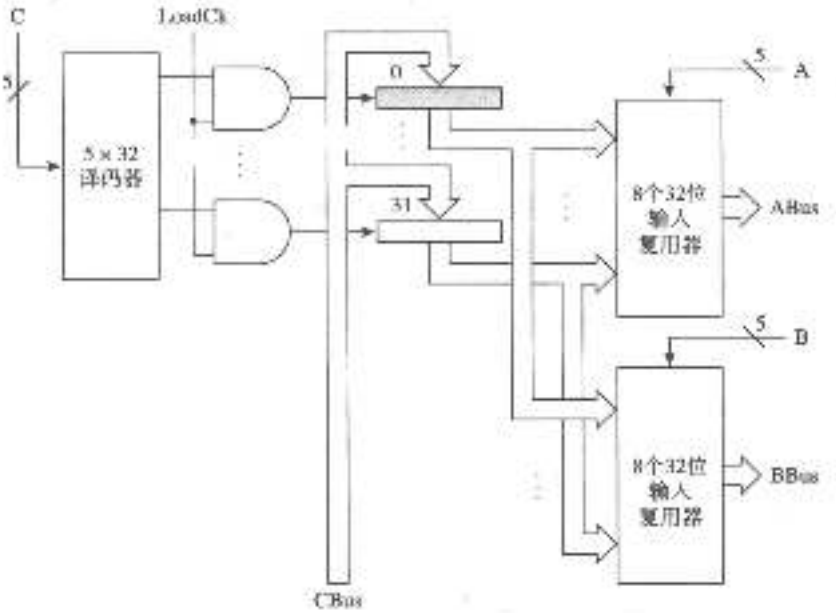


图 11-52　图 11-51 中双端口寄存器体的实现

两输出端口由两个 32 路输入复用器组成，每路可以发送 8 位的量。每个复用器是一组 8 个 32 输入复用器，类似于图 10-43 中的复用器。A 的 5 条线连接到第一个端口中所有 8 个复用器的 5 条选择线。第一个复用器负责选择和传送所有 32 个寄存器的第一位，第二个复用器负责选择和传送所有 32 个寄存器的第二位，以此类推。

总结

由带反馈回路的逻辑门构建的时序电路可以记忆状态。四个基本时序设备是 SR 触发器（设置，重置）、JK 触发器、D 触发器（数据或延迟）和 T 触发器（反转）。SR 触发器的 S 输入把状态设置为 1，R 输入把状态重置为 0，SR = 11 这种输入情况是未定义的。除了定义了 JK = 11 这种情况并反转状态外，JK 的输入与 SR 一样。D 输入直接传送到状态。如果 T 输入为 1，那么状态反转，否则保持不变。每种触发器都可以构造成主 - 从设备，以解决外部反馈引起的不稳定问题。

时序电路通常由一个组合电路组成，该组合电路的输出反馈到一组状态寄存器中，状态寄存器的输出又反馈到组合电路的输入。可以用状态转移图来描述时序电路的特性，状态转移图是有限状态机的一种表现形式。在分析时序电路时，输入和时序电路是给定的，要确定的是输出。

设计时序电路则是给出输入和期望的输出，要确定时序电路。激励表可以帮助进行设计。触发器的激励表由设备四种可能的状态改变（0 到 0、0 到 1、1 到 0、1 到 1）以及产生改变所需的输入条件组成。设计过程包括对产生给定状态转移图所必需的输入条件制表，然后设计组合电路来生成这些输入条件。

587 ~ 590

寄存器是一组 D 触发器。三态缓冲器可以用来实现双向总线。（从概念上说）内存芯片是一个 D 触发器的数组，有一组地址线、数据线和控制线。控制线一般由芯片选择（CS）、写使能（WE）和输入使能（OE）组成。内存映射 I/O 在指令集中不需要有原生输入/输出指令，而是依靠 LOAD 指令输入、依靠 STORE 指令输出。地址译码是一种使用 CS 线、由一组内存芯片构建内存模块的技术，部分地址译码会把选择电路中门的数量减到最少。Pep/8 CPU 中的双端口寄存器体实现了一些对 ISA 程序员可见的寄存器，也实现了对 ISA 程序员不可见的临时寄存器和常量寄存器。

练习

11.1 节

*1. 在什么情况下，一串任意数量的、带反馈回路的反相器（如图 11-1 所示）会形成一个稳定的网络？

2. 构建类似于图 11-3 和图 11-4 的表，说明如果 SR 锁存器的起始状态为 Q = 1，则 R 变为 1 又变回 0 会把它重置为 Q = 0。

3. 在图 11-10 中定义下列点：（1）A 是上面主触发器 AND 门的输出。（2）B 是下面主触发器 AND 门的输出。（3）C 是反相器的输出。（4）D 是上面从锁存器 AND 门的输出。（5）E 是下面从锁存器 AND 门的输出。

   假定时钟脉冲到达前 SR = 10，Q = 0。做表展示出图 11-11 中下述每个间隔期间 A、B、C、D、E、R2、S2、Q 和 $\overline{Q}$ 的值，假设门延迟为 0。

   *（a）t_1 之前　　*（b）t_1 和 t_2 之间　　（c）t_2 和 t_3 之间

   （d）t_3 和 t_4 之间　　（e）t_4 之后

4. 时钟脉冲达到前 SR = 01 和 Q = 1 的情况下再做一遍练习 3。

5. 画出下面触发器的类似图 11-14 的状态转移图：

   （a）JK　　*（b）D　　（c）T

6. 画出 T 触发器的类似图 11-20c 的时序图。

7. 用 SR 触发器构建 T 触发器。

8. 本节展示了怎样用 SR 触发器和一些门构建 JK 触发器和 D 触发器。实际上任何触发器都可以用其他触发器和一些门来构成。用 JK 触发器构建下面的触发器：

591

   *（a）D　　（b）SR　　（c）T

   用 D 触发器构建下面的触发器：

   （d）SR　　（e）JK　　（f）T

   用 T 触发器构建下面的触发器：

   （g）SR　　（h）JK　　（i）D

11.2 节

9. *（a）图 11-10 是 SR 主-从触发器的实现。把它修改为可提供如图 11-31 所示的异步清除输入。Clear 为 0 时，设备应该正常运行；Clear 为 1 时，主触发器的状态 Q2 和从触发器的状态 Q 都被迫制为 0，无论时钟是 0 还是 1。（b）把图 11-10 修改成可提供异步预设置。（c）把图 11-10 修改成既可提供异步清除，又可提供异步预设置。

10. 一个时序电路有两个 JK 触发器 FFA 和 FFB，两个输入 X1 和 X2，触发器输入为

    $JA = X1\,B$　　$JB = X1\overline{A}$

    $KA = X2 - X1\,AB$　　$KB = X2 + X1\,A$

    除了触发器状态外没有其他输出。画出它的时序电路的逻辑图和状态转移图。

11. 一个时序电路有一个 JK 触发器 FFA，一个 T 触发器 FFB，一个输入 X，触发器输入为

$J = X \oplus B$ $T = X \odot A$

$K = \overline{X}\,B$

输出为 $Z = A\,B$。画出它的时序电路的逻辑图和状态转移图。

12. 一个时序电路有两个 SR 触发器 FFA 和 FFB，两个输入 X1 和 X2，触发器输入为

$SA = X1$ $SB = \overline{X1}\,\overline{X2}\,\overline{A}$

$RA = \overline{X1}\,X2$ $RB = X1\,A = X2$

输出为 $Z = X1\,\overline{A}$。画出它的时序电路的逻辑图和状态转移图。

13. 用下列触发器设计图 11-32 的时序电路

*(a) D (b) T

14. 图 11-53 是一个有三个触发器和一个输入的时序电路的状态转移图。输入为 1 时以二进制加 1，输入为 0 时保持状态不变。用下面的触发器设计电路并画出逻辑图：

(a) JK (b) SR (c) D (d) T

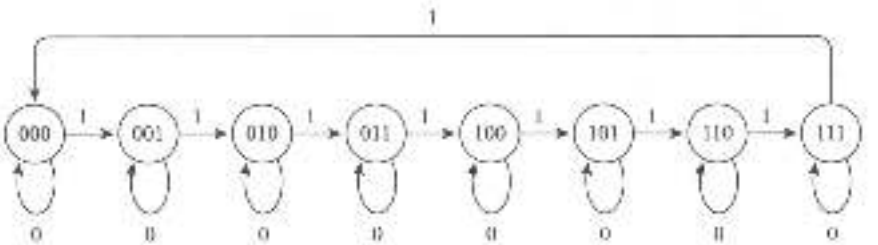


图 11-53 练习 14 的状态转移图

15. 图 11-54 是一个有三个触发器和一个输入的时序电路的状态转移图。输入为 1 时以二进制加 1，输入为 0 以二进制减 1。用下面的触发器设计电路并画出逻辑图：

*(a) JK (b) SR (c) D (d) T

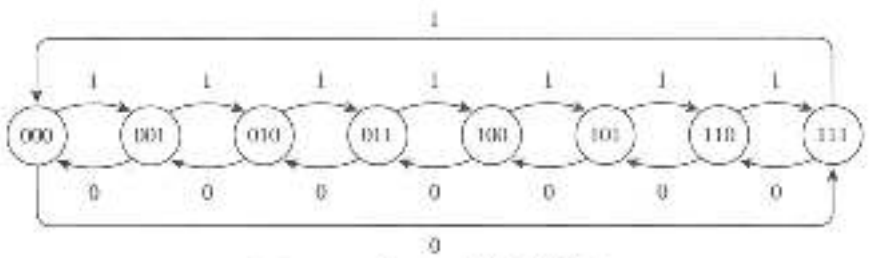


图 11-54 练习 15 的状态转移图

16. 图 11-55 是一个有两个触发器和两个输入的时序电路的状态转移图。输入为 01 时以二进制加 1，输入为 10 以二进制减 1，输入为 00 时状态不变，输入 11 绝对不会发生，可以看作无关条件。用下面的触发器设计电路并画出逻辑图：

(a) JK (b) SR (c) D (d) T

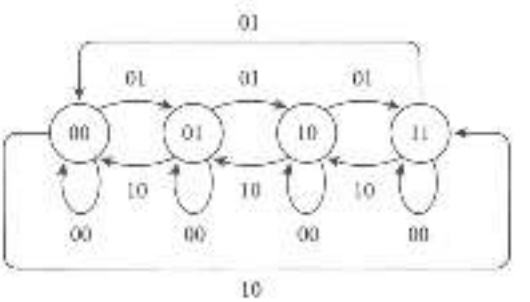


图 11-55 练习 16 的状态转移图

17. 一个时序电路有 6 个状态寄存器和三条输入线，(a) 它有多少种状态？(b) 每个状态有多少种转移？(c) 总共有多少种转移？

11.3 节

18. (a) 图 11-41 的内存芯片里有多少个 AND 门，多

592 ↓ 593

少个 OR 门，多少个反相器？包括译码器中的门，但是不包括 D 触发器中的门。（b）图 11-40a 的内存芯片中每种门有多少个？（c）图 11-40b 的内存芯片中呢？

19. 在实际中，内存芯片的芯片选择线为低电平，这条线标记为 $\overline{CS}$ 而不是 CS。如果所有芯片上的芯片选择线都为低电平有效，图 11-48 会怎样改变？

20. 一个计算机系统的数据总线是 16 位宽。（a）如果有一盒 1K×1 动态 RAM 芯片，这台计算机最少有多少内存字节？（b）如果将 1Kbit 芯片配置成 256×4 的设备，那么（a）中问题的回答又是怎样？

21. 有一个 10 位地址总线的小型 CPU。需要连接一个 64 字节 PROM、一个 32 字节的 RAM 和一个有两条地址线的 4 端口 I/O 芯片。所有芯片的芯片选择为高电平有效。（a）给出使用全地址译码时的连接，PROM 在地址 0，RAM 在地址 384，PIO 在地址 960。（这些地址都是十进制表示的。）（b）芯片还是位于同样的位置，给出使用部分地址译码时的连接。给出使用部分地址译码时的内存图，确保没有区域重叠。

22. 说明图 11-52 上部的每个复用器是怎样连接的。可以使用省略号（...）。

23. 图 11-52 的双端口寄存器体有多少个 AND 门，多少个 OR 门，多少个反相器？包括构建译码器和复用器所需要的门，但不包括组成寄存器的 D 触发器中的门。

594

第七部分

Computer Systems，Fourth Edition

微代码层（第 2 层）



计算机组成

本书最后一章介绍 LG1 层组合电路和时序电路与 ISA3 层机器之间的关联，讲述如何连接硬件设备以形成较高抽象层次上的黑盒子，最终构造出 Pep/8 计算机。

12.1 构造 ISA3 层机器

图 12-1 是 Pep/8 计算机的框图，可以看出 CPU 分成数据区和控制区。数据区从主存子系统接收数据并向它发送数据，控制区向数据区和计算机的其他部分发送控制信号。

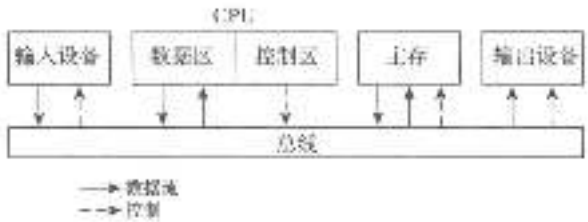


图 12-1 Pep/8 计算机的框图

12.1.1 中央处理单元

图 12-2 是 Pep/8 CPU 的数据区。图中的时序设备标上了阴影，以区别于组合设备。图顶部的 CPU 寄存器与图 11-51 的双端口寄存器体一致。

595 – 597

图 12-2 中没有给出的控制区在数据区的右边，从右边来的控制线就发自控制区。有两类控制信号：组合电路控制和时钟脉冲。所有时钟脉冲的名字都以 Ck 结尾，用来使数据随着时钟进入寄存器或触发器。例如，MDRCk 是 MDR 的时钟输入。当有脉冲到达时，来自 MDRMux 的输入随时钟进入 MDR。

图 12-2 左边的总线就是图 12-1 中的主系统总线，主存和 I/O 设备都连接到总线上。它包括 8 条双向数据线、16 条地址线和 2 条控制线，2 条控制线在图的底部分别标识为 MemWrite 和 MemRead。此外输入和输出设备的控制线在图中没有给出。MAR 是内存地址寄存器，它分为两部分：高位字节 MARA 和低位字节 MARB。标号为“内存”的方框是一个 64KB 的内存系统。总线的 16 位地址线是单向的，所以 MAR 的输出通过总线连接到内存子系统的地址端口输入上。数据总线是双向的，在 MDR 和总线之间有一组八个三态缓冲器（图中未画出），由 MemWrite 控制线使能。通过主系统总线，MemWrite 线和 MemRead 线分别与内存子系统的 WE 线和 OE 线相连。图 12-2 中所有其他用宽箭头表示的总线都是八位数据总线，包括 ABus、BBus、CBus 和把主系统总线的数据线连接到标号为 MDRMux 的方框的总线。

每个复用器（AMux、CMux 和 MDRMux）都是一组八个二输入复用器，它们的控制线连接到一起形成图 12-2 中的一条控制线。例如，图中标号为 AMux 的控制线同时连接到标号为 AMux 的方框中八个复用器体的八条控制线。复用器控制线会按照如下方式让信号通过复用器：

- 复用器控制线为 0，则左边的输入通过到输出。
- 复用器控制线为 1，则右边的输入通过到输出。

例如，如果 MDRMux 控制线为 0，则 MDRMux 会把总线的内容送到 MDR；如果控制

线为 1，就把 CBus 的数据送到 MDR。类似地，如果 AMux 控制线为 0，AMux 会把 MDR 的内容送到 ALU 的左边输入；否则就把来自 ABus 的数据送到 ALU 的左边输入。

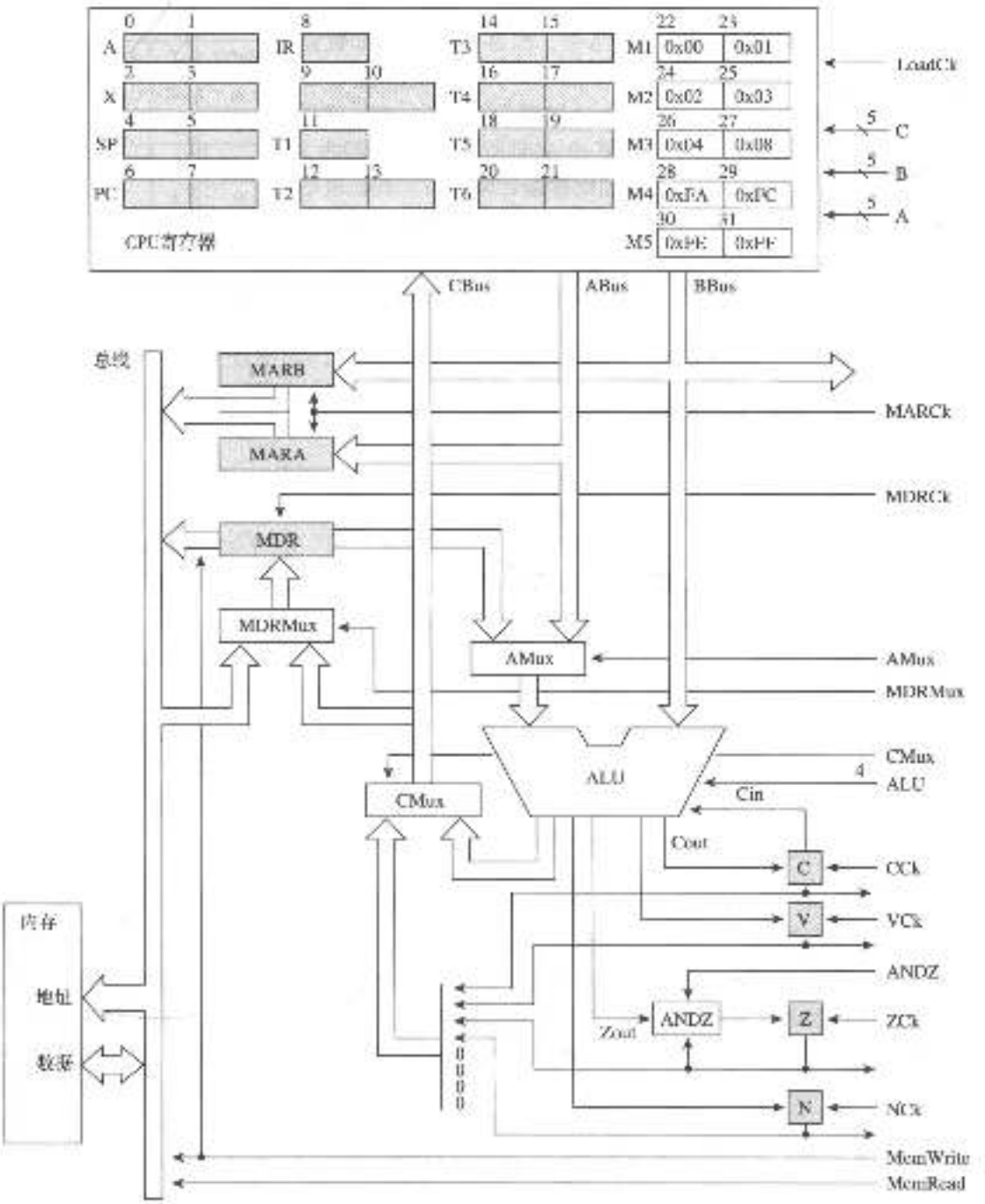


图 12-2　Pep/8 CPU 的数据区

标号为 ALU 的方框是图 10-54 的算术逻辑单元，通过图 12-2 中标号为 ALU 的四条控制线，提供图 10-55 列出的 16 种功能。状态位（N、Z、V 和 C）每个都是一个 D 触发器。例如，标号为 C 的方框是一个 D 触发器，存储进位位的值。该触发器的 D 输入来自 ALU

598 ~ 599

的 Cout 信号；触发器的 Q 输出既在顶部也在底部，顶部的线连接进入 ALU 的 Cin 输入；触发器的时钟输入是标号为 CCk 的控制信号。每个状态位的输出都反馈到进入 CMux 左边总线的低四位字节，高四位硬件设置为 0。每个状态位的输出同时送到控制区。

在 ISA 层，每个寄存器都是 16 位，但是 CPU 的内部数据通路只有 8 位宽。要执行 16 位数字的运算，在 LG1 层就需要两个 8 位数字的运算。如果结果的 16 位都为 0，那么 Z 位就必须设置为 1，也就是两个 8 位运算的 ALU 的 Zout 信号都为 1 的时候。标号为 ANDZ 的组合电路框帮助计算 Z 位，其输出是 Z 位的 D 触发器的输入，它有三个输入：来自控制区的 ANDZ 输入，来自 ALU 的 Zout 输出和来自 Z 位的 D 触发器的 Q 输出。图 12-3 给出的是这个组合电路框的真值表，它有两种运行模式：

- 如果 ANDZ 控制线为 0，则 Zout 直接通过到输出。
- 如果 ANDZ 控制线为 1，则 Zout 和 Z 进行与，结果送到输出。

ANDZ	Z	Zout	输　出
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

图 12-3　图 12-2 中 ANDZ 组合电路的真值表

因此 Z 位要么加载自 ALU 的 Zout 信号，要么加载自 Zout 信号和 Z 位当前值的与，选择哪个取决于来自控制区的 ANDZ 信号。该电路的实现作为章末的一道练习题。

数据流是一个大循环，从顶部的 32 个 8 位寄存器开始，经过 ABus 和 BBus，再经过 AMux 到 ALU，最终经过 CMux 通过 CBus 回到 32 个寄存器。来自主存的数据也可以通过总线加入这个循环，通过 MDRMux 到 MDR。从 MDR 数据可以经过 AMux、ALU 和 CMux 到达任一个 CPU 寄存器。要把 CPU 寄存器的内容送到内存，可以通过 ABus 和 AMux，穿过 ALU、CMux 和 MDRMux 进入 MDR，从 MDR 可以经过总线到达内存子系统。

控制区有 32 条控制输出线和 12 条输入线。32 条输出线控制数据在数据区里的流动，并指明要发生的处理。12 条输入线是来自 BBus 的 8 条线，以及来自状态位的 4 条线，控制区可以测试某些条件。12.2 节会介绍控制区如何产生适当的控制信号。为了介绍数据区是如何工作的，现在假设可以在任何时刻把控制线设置为任何想要的值。

600

12.1.2　冯·诺依曼周期

Pep/8 计算机的核心是冯·诺依曼周期。图 12-2 中的数据区实现了冯·诺依曼周期，它其实不过是一种管道系统。房子里的水通过各种龙头和阀门控制的管道，同样，信号（实际上是电子）流过各种复用器控制的总线线路。在通路上，信号可以通过 ALU，在 ALU 按照需求对数据进行处理。本节介绍实现冯·诺依曼周期所需的控制信号，包括 Pep/8 指令集中某些典型指令的实现，其他指令的实现作为章末练习。

图 4-31 是在 ISA3 层执行程序所需步骤的伪代码描述。do 循环是冯·诺依曼周期。在 LG1 层，虽然指令寄存器的操作数指示符部分是 16 位数字，但实际上 CPU 的数据区是对 8 位数字进行运算。CPU 分两步取操作数指示符，先取高位字节再取低位字节，取完每个字节之后，控制区会把 PC 加 1。图 12-4 是冯·诺依曼周期在 LG1 层的伪代码描述。

控制区把控制信号发到数据区以实现冯·诺依曼周期。图 12-5 是获取指令指示符和 PC 加 1 的控制序列。图中没有给出控制区确定指令是否是一元指令的方法。章末有一道练习要求写出指令不是一元指令时，获取指令指示符的控制信号。

```
do {
    取 PC 中地址处的指令指示符
    PC ← PC + 1
    译码指令指示符
    if (指令不是一元的) {
        取 PC 指定的操作数指示符的低位字节
        PC ← PC + 1
        取 PC 指定的操作数指示符的低位字节
        PC ← PC + 1
    }
    执行取出的指令
}
(未执行停止指令)&&(指令是合法的)
```

图 12-4　冯·诺依曼执行周期的 LG1 层伪代码描述

```
// Save the status bits in T1
1. CMux=0, C=11; LoadCk

// MAR <- PC, fetch instruction specifier.
2. A=6, B=7; MARCk
3. MemRead
4. MemRead, MDRMux=0; MDRCk
5. AMux=0, ALU=0, CMux=1, C=0; LoadCk

// PC <- PC + 1, low-order byte first.
6. A=7, B=23, AMux=1, ALU=1, CMux=1, C=7; CCk, LoadCk
7. A=6, B=22, AMux=1, ALU=2, CMux=1, C=6; LoadCk

// If the instruction is not unary, fetch operand specifier
// and increment PC
...
// Restore the carry bit from T1
n. A=11, AMux=1, ALU=15; CCk

// Execute the instruction fetched
```

图 12-5　获取指令指示符和 PC 加 1 的控制信号

图 12-5 中的每一行都是一个 CPU 时钟周期，由一组控制信号组成，这些控制信号会注入组合设备，通常跟有一个时钟脉冲进入一个或多个寄存器。组合信号用等号表示，其持续时间必须足够长，使得数据能够到达寄存器，并在时钟的控制下存入寄存器。组合信号的设置是并发的，所以用逗号分隔开，逗号就是并发分隔符。组合信号和时钟信号用分号隔开，分号是时序分隔符，因为时钟脉冲是在设置了组合信号之后才施加的。注释用双斜杠（//）表示。

图 12-6 给出的时钟周期对应于图 12-5 中编号 1 ～ 4 的行。以 s 为单位的时钟周期是由以 Hz 为单位的系统时钟频率 f 决定的，$T = 1/f$。在所有其他条件不变的情况下，计算机的频率越高（以 MHz 为单位进行衡量），一个时钟周期的时间就越短，计算机执行得就越快。所以是什么限制了 CPU 的速度呢？周期必须足够长，以允许信号通过组合电路并出现在寄存器的输入（寄存器是时序电路），然后下一个时钟脉冲才能到达。例如，在图 12-6 中周期 1 开始的时候，CMux 线变为 0，然后数据从左边的输入总线通过 CMux，这需要一定的时

间。同时，在这个周期开始时，C 被设置为 11，这个控制信号通过图 11-51 所示的译码器也需要一定的时间。周期 T 必须足够长，能够让这些信号在 LoadCk 脉冲到达前把数据送到寄存器 11 的输入。

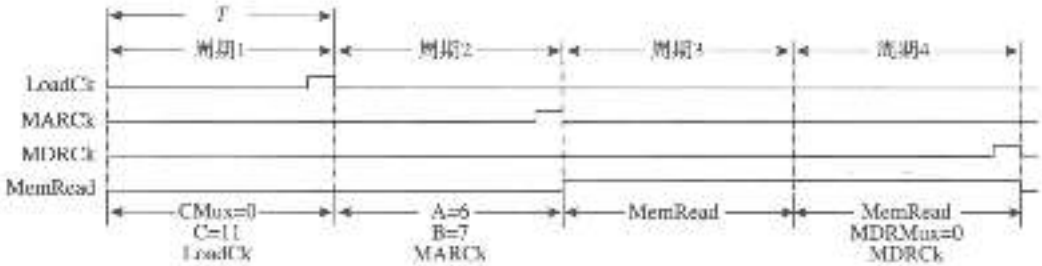


图 12-6　图 12-5 的前四个周期的时序图

程序计数器是一个 16 位的数字。在冯·诺依曼周期中，CPU 会把程序计数器加 1。它把 PC 的低位字节加 1，并存储 C 位，用来加到高位字节上。这样一来，就可以抹掉上一条指令设置的 C 位了。图 12-5 中的周期 1 会把状态位存放在临时寄存器 T1 中。从图 12-2 可以看到状态位反馈到 CMux 的左边输入中。控制信号 C 对 T1 寻址（即图 12-2 中的地址 11），LoadCk 信号使得状态位随着时钟存进 T1。

周期 2 ～ 5 获取指令指示符。在周期 2，A ← 6 把寄存器 6 送到 ABus，B ← 7 把寄存器 7 送到 BBus，从图 12-2 看到寄存器 6 和 7 是程序计数器的两个半边。周期 2 中的 MARCk 脉冲使得 PC 随着时钟进入内存地址寄存器（MAR）。在周期 3，MemRead 信号启动一次内存读。访存地址在周期 2 的末尾被放到主系统总线上。所以现在 MemRead 信号激活了内存子系统中被地址译码电路选中芯片上的 OE 线。内存子系统的延迟很大，通常要经过很多 CPU 周期，数据才能出现在主系统总线上。Pep/8 计算机要求在地址随时钟进入 MAR 后，MemRead 在两个连续的周期内有效，以此来模拟访存延迟较长的情况。在周期 4，MDRMux ← 0 将 MDR 复用器的控制线设置为 0，来自总线的数据会穿过 MDR 复用器到达内存数据寄存器（MDR）的输入。周期 4 中的 MDRCk 脉冲会把指令指示符送入 MDR。

周期 5 把指令指示符从 MDR 送入指令寄存器 IR，过程如下所述。首先，AMux ← 0 将 AMux 控制线设置为 0，这就把 MDR 送到了 AMux 复用器的输出。接下来，ALU ← 0 将 ALU 控制线设置为 0，这就使数据不改变地穿过 ALU，如图 10-55 所示。CMux ← 1 把数据从 ALU 送到 CBus，然后 C ← 8 把 C 设置为 8，这是用来指定指令寄存器的，如图 12-2 所示。最后，LoadCk 把 MDR 的内容送进指令寄存器。

601
↓
603

在周期 5 中，控制区不能同时设置组合信号和 LoadCk 脉冲，在 LoadCk 脉冲到达之前，MDR 的内容必须穿过复用器和 ALU。控制区设计者必须计算穿过这些组合电路的门延迟数量，以确定在时钟到达并送数据进入之前要等待多久。

周期 6 ～ 7 把 PC 加 1。在周期 6，A ← 7 把 PC 的低位字节放到 ABus；B ← 23 把常数 1 放到 BBus；AMux ← 1 选择让 ABus 通过复用器；ALU ← 1 选择算术逻辑单元的 A 加 B 功能。这样 ALU 会把 PC 的低位字节加 1；CMux ← 1 把和送到 CBus，而 C ← 7 把和送回 PC 的低位字节。在同一周期，CCk 把加法的进位送到 C 位。CCk 和 LoadCk 都必须等待足够长的时间，使得数据能够穿过周期 6 中其他控制信号指定的组合设备，然后才能触发并把它们加载进寄存器。

如果 PC 的低位字节原来为 1111 111 (bin)，加 1 会导致向高位字节的进位。在周期 7，A ← 6 把 PC 的高位字节放到 ABus；B ← 22 把常数 0 放到 BBus；ALU ← 2 选择 ALU 的 A 加 B 加 Cin 功能，把从低位字节来的保存的进位加到 PC 的高位字节上；CMux ← 1 把结果送到 CBus；C ← 6 使得 CBus 上的数据加载到 PC 的高位字节上，这个存储是借助 LoadCk 脉冲来完成的。

在指令指示符取出来之后，控制区必须对操作码译码，确定该指令是否是一元的，如果不是，要获取操作数指示符，并把 PC 加 1。在执行取出的指令之前，周期 n 要恢复周期 1 中保存的 C 位。A ← 11 把寄存器 T1 放到 ABus；AMux ← 1 把它送到 ALU 的 A 输入；ALU ← 15 把所有四个保存的状态位送到四个触发器；CCk 使得 C 位随时钟保存，恢复原始的值。

通过把周期 1 和周期 3 组合起来有可能减少周期数。这样会删除周期 1，周期 3（重新编号为周期 2）可写作

```
MemRead, CMux=0, C=11; LoadCk
```

当 CPU 等待从内存读来的结果时，可以同时保存状态位。

不能任意组合控制序列中的时钟周期，必须记住图 12-5 所示的控制序列中的每一个带编号的行都表示一个 CPU 周期。有些周期依赖于前一周期的结果。例如，不能把图 12-5 中的周期 4 和周期 5 组合到一起，因为周期 5 依赖于周期 4 的结果。周期 4 设置 MDR 的内容，而周期 5 要使用 MDR 的内容，所以周期 5 必须在周期 4 之后完成。

在计算机组成中硬件并发是一个很重要的问题。设计者总是保持警觉要利用硬件并发来提高性能。当然，实际中不会使用图 12-5 的七周期序列，因为合并周期 1 和 3 能够不增加电路就提高性能。经过一点思考，还能做得更好。章末有一道练习题要求把图 12-5 中的周期数量从 7 减少到 4。

604

显然这里没有给出控制区的细节，但是也可以想象它会检测刚刚取出的指令，看它是否是一元指令。控制区会把 B 设置为 8，把指令指示符放到 BBus 上进行检测。如果取出的指令不是一元的，那么控制区必须再获取操作数指示符，相应地增加 PC。获取操作数指示符和 PC 加 1 的控制序列是章末的一道练习题。

取指之后，控制区会检测指令指示符，决定要执行 39 条指令中的哪一条。执行指令的控制信号不仅仅取决于操作码，还取决于寄存器 -r 字段和寻址 -aaa 字段。图 12-7 给出了每种寻址方式的操作数和操作数指示符（OprndSpec）。

方括号中的数字是内存地址。要执行一条指令，控制区必须提供让计算区计算内存地址的控制信号。例如，执行使用变址寻址方式的指令，控制区必须执行 16 位加法，用操作数指示符（寄存器 9 和 10）的内容加上 X（寄存器 2 和 3）。加法的结果之后会加载到 MAR，为 LDr 指令的内存读或 STr 指令的内存写做好准备。

寻址模式	操 作 数
立即数	OprndSpec
直接	Mem [OprndSpec]
间接	Mem [Mem [OprndSpec]]
栈相对	Mem [SP + OprndSpec]
栈相对间接	Mem [Mem [SP + OprndSpec]]
变址	Mem [OprndSpec + X]
栈变址	Mem [SP + OprndSpec + X]
栈变址间接	Mem [Mem [SP + OprndSpec] + X]

图 12-7　Pep/8 计算机的寻址方式

图 12-5 中实现冯·诺依曼执行周期第一部分的控制序列看上去很像以某种低级编程语言编写的程序。控制区设计者的任务是设计出这样的电路，对数据区编码实现 ISA3 层（指令集架构层）的指令。LG1 层的语言描述的是

输入组合电路中的二进制信号和输入状态寄存器的时钟脉冲。

接下来的几个例子展示了执行某些有代表性的指令所需的控制序列。每个例子都假设指令已经取出，PC 也相应地增加了。程序中的每条语句都占一行，每行都有编号，每条语句
605 包括一些组合信号，让数据通过复用器或选择 ALU 的功能，之后有一个或几个时钟脉冲来加载一些寄存器。记住，这个抽象层次（LG1 层）上的程序由实现更高抽象层次（ISA3 层）上一条指令所需要的控制信号组成。

12.1.3　实现存储字节指令

图 12-8 给出执行下面这条指令的控制序列

```
STBYTEA there, d
```

这里 there 是一个符号。STBYTEr 指令的 RTL 说明是

```
byte Opend ← r(8..15)
```

这条指令指明了采用直接寻址方式，所以操作数是 Mem[OpendSpec]，即操作数指示符是操作数在内存中的地址。该指令把累加器的低位字节存储到该地址对应的内存单元。状态位不受影响。

```
// MAR <- OprndSpec.
1. A-9, B-10; MARCk

// MDR <- A(low).
2. A-1, AMux-1, ALU-0, CMux-1, MDRMux-1; MDRCk

// Initiate memory write.
3. MemWrite

// Complete memory write.
4. MemWrite
```

图 12-8　实现使用直接寻址方式的存储字节指令的控制信号

这个例子和后面的其他例子一样，假设操作数指示符已经在指令寄存器中，也就是假设冯·诺依曼周期的取指、译码和 PC 加 1 部分已经完成。图 12-8 只给出了冯·诺依曼周期的执行部分。

周期 1 把操作数指示符送到内存地址寄存器。A ← 9 把操作数指示符的高位字节送到 ABus，B ← 10 把操作数指示符的低位字节送到 BBus，而 MARCk 使得 ABus 和 BBus 随时钟送入 MAR 寄存器。

周期 2 把累加器的低位字节送进 MDR。A ← 1 把累加器的低位字节送到 ABus，AMux ← 1 使数据通过 AMux 进入 ALU，ALU ← 0 使数据不经改变地通过 ALU，CMux ← 1 使数据通过 CBus，MDRMux ← 1 使数据经过 MDRMux 到达 MDR，MDRCk 把数据锁存进 MDR 中。

周期 3 和 4 完成内存写，把 MDR 中的数据存储到 MAR 中地址对应的主存中。和内存
606 读一样，内存写要求 MemWrite 线持续两个连续的周期，给内存系统足够的时间从总线获取数据并存储好。

在 ISA 层，存储指令不影响状态位，所以 STBYTEA 的控制序列中没有周期会给 NCk、ZCk、VCk 或 CCk 脉冲。

12.1.4　实现加法指令

图 12-9 给出执行下面这条指令的控制序列

```
ADDA this,I
```

ADDr 的 RTL 表示为

$$r \leftarrow r + \text{Opend};\ N \leftarrow r < 0,\ Z \leftarrow r = 0,\ V \leftarrow [overflow],\ C \leftarrow [carry]$$

这条指令把操作数加到寄存器 r，并把和存放在寄存器 r 中，在这种情况下是累加器。因为该指令采用立即数寻址方式，因此操作数就是操作数指示符。和前面一样，本例假设已经获取了指令指示符，并只存放在指令寄存器中了。

```
// A<low> <- A<low> plus Oprnd<low>. Save carry.
1. A=1, B=10, AMux=1, ALU=1, ANDZ=0, CMux=1, C=1; ZCk, CCk, LoadCk

// A<high> <- A<high> plus Oprnd<high> plus saved carry.
2. A=0, B=9, AMux=1, ALU=2, ANDZ=1, CMux=1, C=0; NCk, ZCk, VCk, CCk, LoadCk
```

图 12-9　实现使用立即数寻址方式的加法指令的控制信号

这条指令会影响所有 4 个状态位。虽然累加器能存放 16 位值，不过 Pep/8 CPU 的数据区只能处理 8 位数字。要完成加法，控制序列必须先把低位字节相加，保存低位相加产生的进位，再把它和高位字节求和。如果把这个两字节的数字当作有符号整数，则当它为负时，N 会被设置为 1；否则把 N 清零。最高位字节的符号位决定 N 的值。如果两字节的数字为全 0，那么 Z 被设置为 1；否则 Z 被清零。所以，和 N 位不同，高位和低位字节的值共同决定 Z 的值。

周期 1 把累加器的低位字节和操作数指示符的低位字节相加。A = 1 把累加器的低位字节放到 ABus，而 B = 10 把操作数指示符的低位字节放到 BBus。AMux = 1 把 ABus 送到复用器，ALU = 1 选择 ALU 的 A 加 B 功能，CMux = 1 把和送到 CBus，C = 1 把 ALU 的输出指向累加器的低位字节准备存储，LoadCk 使之随时钟存入累加器。在同一周期，ANDZ = 0 把 Zout 送到 ANDZ 组合电路的输出，它又是一位寄存器 Z（一个 D 触发器）的输入。ZCk 把这个位锁存进寄存器。

周期 2 把累加器的高位字节加上操作数指示符的高位字节。A = 0 把累加器的高位字节放到 ABus，而 B = 9 把操作数指示符的高位字节送到 BBus。AMux = 1 使 ABus 经过复用器，ALU = 2 选择 ALU 的 A 加 B 加 Cin 功能，CMux = 1 把和送到 CBus，C = 0 把它指向累加器的高位字节准备存储，LoadCk 会在时钟到来的时候把它送入寄存器。ANDZ = 1 把 Zout 与 Z 的结果送到 ANDZ 组合电路的输出，它会作为 Z 寄存器的输入。ZCk 把这个位锁存进寄存器。当且仅当 Zout 和 Z 都为 1 时，Z 寄存器的内容才会锁存为 1。Z 的值会随着周期 1 的 ZCk 被保存，当且仅当低位相加的和为全 0 的时候，它会继续保持为 1。所以，当且仅当和的 16 位都为 0 时，Z 的最终值为 1。其他 3 个状态位（N、V 和 C）反映的是高位相加的状态，它们在周期 2 随着 NCk、VCk 和 CCk 存储。

12.1.5　实现装入指令

图 12-10 给出执行下面这条指令的控制序列

```
LDX this,n
```

LDr 的 RTL 表示为

$$r \leftarrow \text{Oprnd}; N \leftarrow r < 0, Z \leftarrow r = 0$$

这条指令从内存将两个字节加载到变址寄存器。因为使用的是间接寻址方式，所以如图 12-7 所示，操作数是 Mem[Mem[OprndSpec]]，操作数指示符是操作数的地址的地址。控制序列必须从内存中取出一个字，把它作为操作数的地址，还需要再取一次才能得到操作数。

607

```
// T3<high> <- Mem[OprndSpec].
1. A=9, B=10; MARCk
2. MemRead
3. MemRead, MDRMux=0; MDRCk
4. AMux=0, ALU=0, CMux=1, C=14; LoadCk

// T2 <- OprndSpec + 1.
5. A=10, B=23, AMux=1, ALU=1, CMux=1, C=13; CCk, LoadCk
6. A=9, B=22, AMux=1, ALU=2, CMux=1, C=12; LoadCk

// T3<low> <- Mem[T2].
7. A=12, B=13; MARCk
8. MemRead
9. MemRead, MDRMux=0; MDRCk
10. AMux=0, ALU=0, CMux=1, C=15; LoadCk

// Assert: T3 contains the address of the operand.
// X<high> <- Mem[T3].
11. A=14, B=15; MARCk
12. MemRead
13. MemRead, MDRMux=0; MDRCk
14. AMux=0, ALU=0, ABUS=0, CMux=1, C=2; MDRCk, ZCk, LoadCk

// T4 <- T3 + 1.
15. A=15, B=23, AMux=1, ALU=1, CMux=1, C=17; CCk, LoadCk
16. A=14, B=22, AMux=1, ALU=2, CMux=1, C=16; LoadCk

// X<low> <- Mem[T4].
17. A=16, B=17; MARCk
18. MemRead
19. MemRead, MDRMux=0; MDRCk
20. AMux=0, ALU=0, ABUS=1, CMux=1, C=3; ZCk, LoadCk

// Restore C, assumed in T1 from Fetch.
21. A=11, AMux=1, ALU=15; CCk
```

图 12-10　实现使用间接寻址方式的装入指令的控制信号

图 12-11 给出执行图 12-10 的控制序列的效果。假设符号 this 的值为 0x0012，内存的初始值如地址 0012 和 26D1 所示。Mem[0012] 的值为 0x26D1，这是操作数的地址。Mem[26D1] 是操作数，值为 0x53AC，这条指令就是要把它加载进变址寄存器。该图显示了这个控制序列影响的每个寄存器的地址。指令寄存器的第一个字节 CA 是采用间接寻址的 LDX 指令的指令指示符。

周期 1 ~ 4 把 Mem[OprndSpec] 传送到临时寄存器 T3 的高位字节。在周期 1，A = 9 和 B = 10 把操作数指示符分别放到 ABus 和 BBus，而 MARCk 用时钟把它们送进内存地址寄存器。在周期 2，MemRead 发起一个内存读，周期 3 会完成该操作。周期 4 按照通常的方式把数据从 MDR 送到 T3 的高位字节。

周期 5 ~ 6 把操作数指示符加 1，并把结果存储在临时寄存器 T3 中。周期 5 把操作数指示符的低位字节加 1，周期 6 会把低位字节相加可能产生的进位考虑进高位字节加法中。

周期 7 ~ 10 使用 T2 中计算出来的地址获取操作数地址的低位字节。周期 7 把 T2 的内容放到内存地址寄存器。周期 8 发起内存读，周期 9 把这个字节从内存锁存进 MDR 中。周

期 10 把这个字节从 MDR 经过 AMux 送进 T3 的低位字节。

此时，可以断言临时寄存器 T3 包含操作数的地址 0x26D1。最后，可以获取操作数的第一个字节，把它加载进变址寄存器的第一个字节。周期 11 ～ 14 执行加载过程。LDr 的 RTL 说明表明这条指令会影响 N 和 Z 位。N 位是由最高字节的符号位决定的。所以，周期 14 包括时钟脉冲 NCk，把经过 ALU 的字节存进 N 位。Z 位取决于两个字节的值，所以周期 14 包括 ANDZ ← 0 和 ZCk，以把 Zout 信号保存到 Z 寄存器。

周期 15 ～ 16 把第一个字节的地址加 1，并把结果保存到临时寄存器 T4。要获取操作数的第二个字节，周期 17 ～ 20 把增加过的地址从 T4 保存到内存地址寄存器。周期 20 最后会把第二个字节放到变址寄存器。它包括 ANDZ ← 1，这样来自低位字节的 Z 值（在周期 14 存储的）会和来自高位字节的 Zout 与，ZCk 会存储该指令加载进来的 16 位数字的正确的 Z。

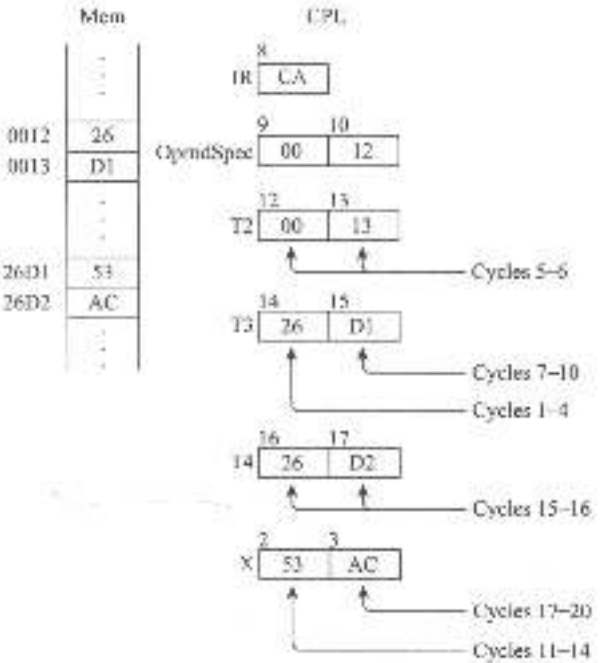


图 12-11　执行列 12-10 的控制序列的结果

装入指令不应该影响 V 或 C 位。不幸的是，在各种地址增加的处理中 C 位会改变好几次。所幸的是冯·诺依曼周期的取指部分会把所有四个状态位保存在临时寄存器 T1，就像图 12-5 的周期 1 显示的那样。可以假设在这期间 T1 没有被篡改过。图 12-11 中的周期 21 会相应地恢复 C 位。

图 12-5 对冯·诺依曼周期的取值和 PC 增加部分的性能进行了优化。用同样方法也可以优化图 12-10 的控制序列。章末的一道练习题要求把图 12-10 的时钟周期从 21 降低到 17。

12.1.6　实现算术右移指令

图 12-12 给出执行这条一元指令的控制序列

ASRA

ASRr 指令的 RTL 表示为

C ← r(15), r(1..15) ← r(0..14); N ← r<0; Z ← r=0

ASRr 指令是一元的，没有内存访问，这使得控制序列很简短。

```
// Arithmetic shift right of high-order byte
1. A=0, AMux=1, ALU=13, ANDZ=0, CMux=1, C=0; NCk, ZCk, CCk, LoadCk

// Rotate right of low-order byte
2. A=1, AMux=1, ALU=14, ANDZ=1, CMux=1, C=1; ZCk, CCk, LoadCk
```

图 12-12　实现一元 ASRA 指令的控制信号

因为 ALU 只能计算 8 位数字，所以必须要把 16 位位移分解成两个 8 位运算。图 12-13

给出四个 ALU 执行的移位和循环操作。要做算术右移，控制序列对高位字节做算术右移，再对低位字节做循环右移。

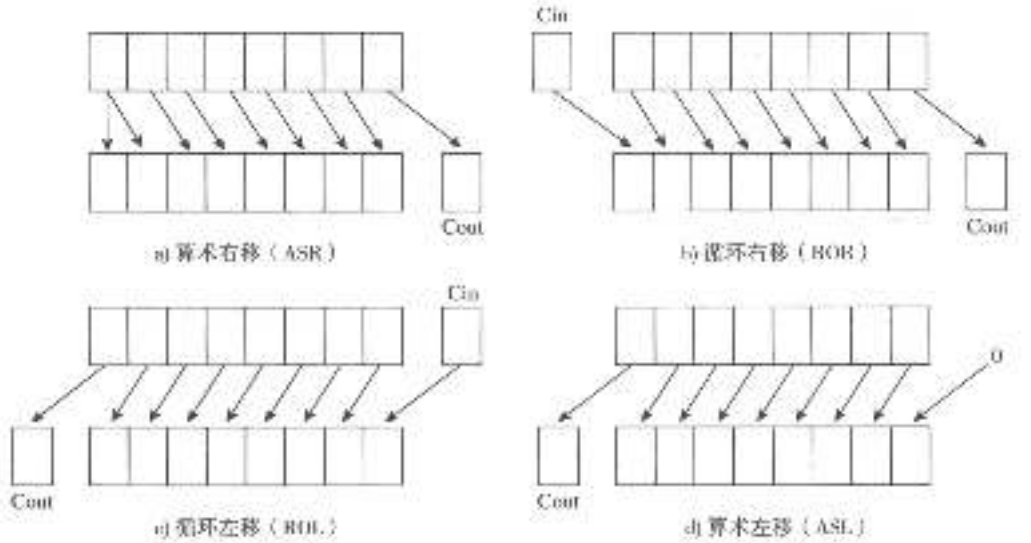


图 12-13　Pep/8 ALU 执行的算术和循环操作

周期 1 中，A ← 0 把累加器的高位字节放到 ABus，AMux ← 1 把它送到 ALU，ALU ← 13 选择算术右移运算，CMux = 1 和 C = 0 把结果送到累加器准备存储，而 LoadCk 会存储它。ANDZ ← 0 把位移运算的 Zout 送到 Z 寄存器，而 ZCk 会保存它。NCk 会存储来自高位运算的 N 位，这是最终的值。CCk 会保存来自高位运算的 C 位，但这不是 C 的最终值。

周期 2 中，A ← 1 把累加器的低位字节放到 ABus，AMux = 1 把它送到 ALU，ALU = 14 选择循环右移运算，CMux ← 1 和 C ← 1 把结果送到累加器准备存储，而 LoadCk 会存储这个结果。ANDZ 会使得 ANDZ 组合电路执行 Zout 和 Z 的与操作。ZCk 会把这个结果存储到 Z 中作为最终值。CCk 会存储 Cout 作为 C 的最终值。

12.2　性能问题

从理论的角度，所有真实的冯·诺依曼计算机器的计算能力都是相同的。给定一种机制将有限容量的磁盘存储与一台机器相连接，这在计算能力上等价于图灵机。在计算能力方面，Pep/8 和世界上最大的超级计算机之间唯一的差别是计算占用的时间。为了计算出一个问题的解答，Pep/8 可能需要一百万年，超级计算机可能只需要 1 毫秒，但是理论上它们可以做同样的事情。

从实际的角度，时间很重要。在所有其他条件相同的情况下，越快越好。虽然 LG1 层图 12-2 的数据区可以实现 ISA3 层的 Pep/8，问题是，速度有多快？本节描述加速计算的技术。提高性能有两个根本原因：

608 / 612

- 空间 / 时间折中
- 并行

时间 / 空间折中是指通过增加电路（这会在集成电路上占用空间）来降低计算时间。在控制序列中合并周期是一种形式的硬件并行，重新组织控制区使得集成电路上可以有更多的

子系统并发地运行，这也可以提高并行度。

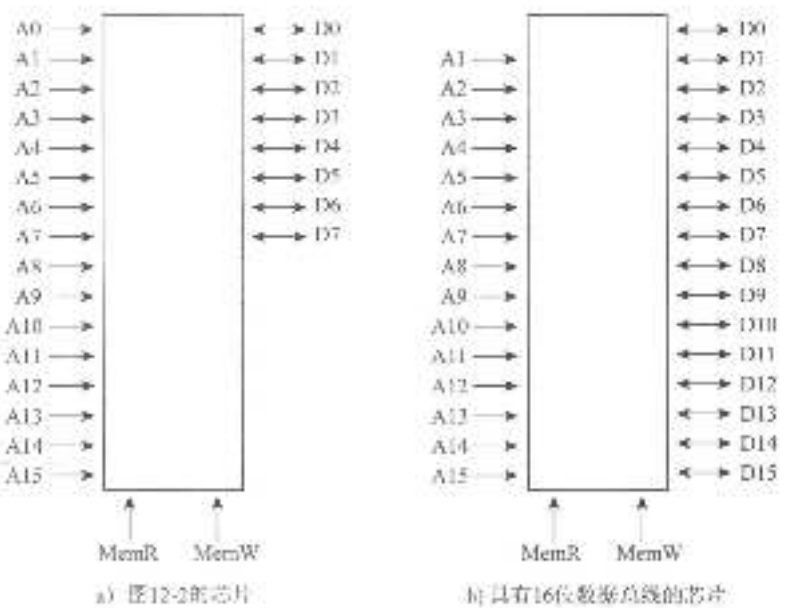
12.2.1　总线宽度

用空间 / 时间折中来提高性能的最直接方法是增加寄存器和总线的宽度，这要求整个集成电路都增加宽度。

例如，在 ISA3 层，Pep/8 是一个 16 位的机器，也就是说保存数据的寄存器（例如累加器）和保存地址的寄存器（例如 PC）都是 16 位宽的。因为 PC 通过 MAR 连接到主系统总线，所以地址总线也是 16 位宽。所有运算都是对 16 位数字的运算。要提高性能，可以把数据区总线（ABus、BBus、CBus 及其他）的数据总线宽度从 8 位提高到 16 位。把数据区总线从 8 位提高到 16 位会带来性能的极大提升。所有需要两个周期才能处理完的指令（一个周期处理高位字节，一个周期处理低位字节）都只需要一个周期了。

不过还是有一些问题。因为 MDR 要和 AMux 输入保持一致，所以要把它也变成 16 位宽。所以该怎么访问内存呢？如果主系统总线有 16 根数据线，那么主存就不再是按字节寻址的了？另一个问题是，该如何重组寄存器体里的寄存器呢？假设所有寄存器都是 16 位宽，那么一个字节的指令指示符该怎么存放在 IR 中呢？

实际上现在所有计算机都是按字节寻址的，但是主系统总线也不止八条数据线。它们的工作方式是省略最低的几位寻址位，位数与数据总线的宽度相对应。图 12-14 展示了这种方法是怎样应用到具有 16 位 MDR 的 Pep/8 上的。图 12-14a 对应于图 12-2，其中有 16 条地址线和 8 条数据线。在图 12-14b 中，MDR 是 16 位宽，数据线的数量翻倍，而少了地址线 A0。



a）图 12-2 的芯片　　b）具有 16 位数据总线的芯片

图 12-14　Pep/8 内存芯片的输出引脚图

假设 CPU 请求在地址 0A57（hex）的字节，这个地址的二进制表示是 0000 1010 0101 0111（bin）。送到内存子系统的地址请求是 0000 1010 0101 0110，最后一位为 0，因为不需要 A0。系统返回地址 0A56 和 0A57 的两个字节。CPU 必须从两个字节的字中抽取出它想要的字节。这种方式的好处是当 CPU 想要一个两字节的字时，例如请求位于 1BD6 的字时，

只需要一次访问就能得到两个字节。

不过现在又有一个问题了。假设CPU想要位于地址1BD5的字，这个地址是一个奇数。这会要求两次访问：一次取出1BD4，抽取出第二个字节；一次取出1BD6，抽取出第一个字节。这就没有加速的优势了，因为想要取的数的地址不是偶数。一个更典型的系统是四字节数据总线，没有A0和A1。对于一个要求1、2、3或4个字节的内存请求，这些字节都落在地址可以被4整除的四字节边界里，那么一次内存访问就能满足。这对性能的提高很大，所以大多数汇编语言都有特殊的点命令，使程序员可以强迫目标代码必须地址对齐，以减小内存访问时间。

可以看到增加总线宽度对芯片大小有很大影响。所有的电路，包括ALU，复用器和寄存器，都必须增加以和总线匹配。计算机的历史显示总线和寄存器在不断地变宽。图12-15展示的是Intel CPU如何把总线宽度从4位增加至32位。4004是第一个片上微处理器。接下来演变中的一大步是从32位机器到64位机器。虽然科学计算工作站和服务器拥有64位的寄存器和总线已经很多年了，但是这个大小的CPU在桌面机器上才刚刚开始出现。

芯　片	日　期	寄存器宽度
4004	1971	4-bit
8008	1972	8-bit
8086	1978	16-bit
80386	1985	32-bit

图12-15　历史上的寄存器/总线宽度

带来这种变化的原因可能仅仅是技术以一种稳定的速度在发展。Gordon Moore，Intel的创始人，在1965年观察到集成电路上晶体管的密度每年都会翻番，而且在可预见的未来还会继续如此。这个速度现在有些减缓了，所谓的摩尔定律（Moore's Law）是指集成电路上晶体管的密度每18个月会翻番。这个速度不会永远保持，因为这种变小化最终会到达原子的尺度，而原子是无法再分割的了。人们一直在热烈讨论摩尔定律具体什么时候会终止，许多人预测它会消亡，结果发现它还在继续。

得到普遍接受的“n位计算机”的意思是n为MAR和在ISA3层可见的CPU寄存器的位数。因为ISA3层可见的寄存器可以存放地址，它们的宽度通常和MAR一样，所以这个定义没有歧义。关于这个定义的疑虑经常出现在市场上。n位计算机的主系统总线不一定有n条地址线，CPU的数据区也不一定有n位数据总线，LG1层的寄存器体也不一定有n位。这些宽度都可以小于n。进一步说，使用图12-14b中的技术，主系统总线中数据线的数量可以大于n。

一个经典的例子是1964年开始的IBM 360系列，这是第一个计算机系列，它们的模型具有同样的ISA3指令集和寄存器。它是32位机器，但是根据模型的不同，CPU中的数据总线可以是8位、16位和32位。因为LG1层细节对ISA3层的程序员是屏蔽的，所以在系列中某个模型上编写和调试的所有软件都可以在另一个模型上运行，无须改变。唯一可以感觉到的模型之间的区别是以执行时间测量的性能。这个概念在当时是革命性的，基于抽象层次的概念提升了计算机的设计。

处理器系列中新的芯片经常没有完整的地址线能匹配MAR的宽度。MAR的宽度是CPU最多能访问的主存字节数上限，如图12-16所示，16位计算机的最大限制是64 KB。当操作系统和应用程序需要更多内存时，唯一的方法是增加MAR的宽度。我们现在也受到同样的制约。推动64位计算机的最大动力是因为多媒体应用和大型数据库需

MAR宽度	可寻址字节数
8	256
16	64K
32	4G
64	17，179，869，184G

图12-16　最大内存限制

要大于 4 GB 的地址空间。64 位计算机可能是最后的诉求，因为 64 位地址线可以访问 160 亿 GB。从一代到下一代的增长不是线性的，而是指数性的。曾经有很多年，32 位计算机有 32 位的 MAR，但是主存总线只有 24 条线，高位的 8 个地址位被忽略。用户可以在这样的机器上安装最多 16MB（2^{24}）内存，这在当时已经足够大了。未来在 64 位计算机上也是一样的，根据市场需要，外部地址线的数量会小于 MAR 的内部宽度。

12.2.2　特殊的硬件单元

利用空间/时间折中技术还有另一种方式，因为改进硬件系统的性能就要确定性能瓶颈，所以可以设计特殊的硬件单元来缓解瓶颈。比起增加整个系统总线带宽的粗暴方法，这种方法更像是手术式的，只花费增加全部总线带宽所需成本的一部分，就可能获得巨大的性能提升。

613 / 615

图 12-10 正是这样一种情况，是实现 LDX　this,n 的控制序列。你可能已经注意到了，需要好几个周期来完成地址加 1，因为需要访问内存中相邻的下一个位置的字节。这需要三个周期：1）地址的低位字节加 1；2）高位字节加上可能的进位；3）把两字节字传送到 MAR。周期 5、6 和 7 就是这样的例子。

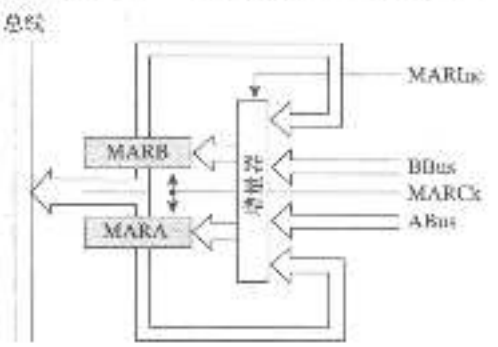


图 12-17　对 MAR 加 1 的特殊的硬件单元

问题是 ALU 是一个通用目的设备，用来对 8 位数字做多种运算。要缓解这个瓶颈，需要一个特殊目的的电路来进行 1 加上一个 16 位数字的操作。图 12-17 展示的是一种可能性。在 A 和 B 总线与 MAR 之间插入一个组合电路，这个组合电路有来自 MARA 和 MARB 的反馈。控制线标号为 MARInc，工作方式如下：

- MARInc = 0，ABus 和 BBus 不经改变地通过。
- MARInc = 1，MAR + 1 的 16 位和通过。

增量器（incrementer）黑盒子的设计作为章末的一道练习题。图 12-18 给出的是实现采用间接寻址方式的装入指令的控制信号，使用的是图 12-17 中的地址增量器。

```
// T3<high> <- Mem[OperandSpec].
// MAR <- OperandSpec + 1.
1. A=9, B=15, MARInc=0; MARCk
2. MemRead
3. MemRead, MDRMux=0, MARInc=1; MDRCk, MARCk
4. MemRead, AMux=0, ALU=0, CMux=1, C=14; LoadCk

// T3<low> <- Mem[OperandSpec + 1].
5. MemRead, MDRMux=0; MDRCk
6. AMux=0, ALU=0, CMux=1, C=15; LoadCk

// Assert: T3 contains the address of the operand.
// X<high> <- Mem[T3].
// MAR <- T3 + 1.
7. A=14, B=15, MARInc=0; MARCk
8. MemRead
```

图 12-18　用图 12-17 的地址增量器实现采用间接寻址方式的装入指令的控制信号

```
9. MemRead, MDRMux=0, MARInc=1; MDRCk, MARCk
10. MemRead, AMux=0, ALU=0, ANDZ=0, CMux=1, C=1; NCk, ZCk, LoadCk

// R<low> <- Mem[T3 + 1].
11. MemRead, MDRMux=0; MDRCk
12. AMux=0, ALU=0, ANDZ=1, CMux=1, C=0; ZCk, LoadCk
```

图 12-18（续）

每次加载 MAR 都要记得指定 MARInc。在周期 3，可以在加载 MDR 的同时增加 MAR，因为这两个操作不需要数据区中相同的资源。主－从设计保证 MAR 中的变化不会影响当前内存读的地址线。周期 9 也利用了同样的推理。在周期 4 和 10，可以开始下一个读，因为在前一个周期数据已经随着时钟进入 MAR 了。地址增量器带来的另一个好处是不使用 C 位，所以不需要恢复它。把图 12-10 中的时钟合并一下，可以得到总共 17 个周期。使用地址增量器把时钟周期数从 17 减少到 12，节省了 5/17=29% 的时间。

地址增量器把地址字作为一个 16 位的数字来处理，从而减少了周期数。要想在数据处理上取得同样的进展，需要特殊目的的硬件，它也以 16 位为单位访问内存。在上面的控制序列中，周期 1 ～ 4 访问一个字的高位字节，周期 5 ～ 7 访问低位字节。如果把图 12-14b 的主系统总线上的数据线数量翻倍，一次内存操作就能访问整个字了。要做到这点，需要一个两字节的 MDR。

进一步来说，如果要求所有地址和所有两字节字操作数都存储在偶数地址，那么可以不使用地址增量器了，因为它唯一的目的是优化对连续字节的内存访问。如果一次访问能获得两个字节，那就不再需要它了。要想这个方案能够行得通，汇编器必须包含对齐点命令，强迫地址和两字节字操作数的目标代码存放在偶数地址。这个要求也适用于全局变量和运行时栈上的局部变量。32 位计算机要求数据存储在能被 4 整除的地址，而 64 位计算机要求地址能被 8 整除。

图 12-18 的控制序列还显示了一个瓶颈，这个问题可以通过修改数据区的通路来减轻。临时寄存器 T3 的唯一目的是在被送到 MAR 之前保存从内存获取的地址。如果有 16 位的数据通路直接从 MDR 到 MAR，就可以省掉把地址放到 T3 再送到 MAR 的那些周期。

在图 12-19 中有两个 MDR 寄存器：MDR0 保存在偶数地址的字节的内容，而 MDR1 保存在奇数地址的字节。每个寄存器都是通过自己的复用器从 CBus 加载进来的。还有一个额外的标号为 01Mux 的复用器选择要把哪个 MDR 送到数据区的主数据通路上。和通常情况下一样，01Mux 控制线为 0 使 MDR0 通过复用器。还有一条数据通路从两字节 MDR 到两字节 MAR，中间经过标号为 MARMux 的复用器。这个复用器控制线选择数据的原则是：

- MARMux 为 0 使得 MDR 送到 MAR。
- MARMux 为 1 使得 ABus 和 BBus 送到 MAR。

图 12-20 展示了采用间接寻址方式的装入指令的控制序列。与图 12-10 不同，它假设地址字和两字节数据字都是对齐的。这个控制序列需要 8 个周期，在原始序列 17 个周期的基础上有 53% 的改进。这个节省比例比使用地址增量器时大，但是对设计也有很大影响。增加主系统总线的数据线影响内存子系统和 CPU 芯片的地址输出引脚。同时，内存对齐要求对汇编语言程序也有影响，并且使得运行时栈的存储更加复杂。不过现在大多数 CPU 的设计都有这个要求，因为带来的性能回报很高。为了保持程序简单，Pep/8 汇编语言没有对齐要求。

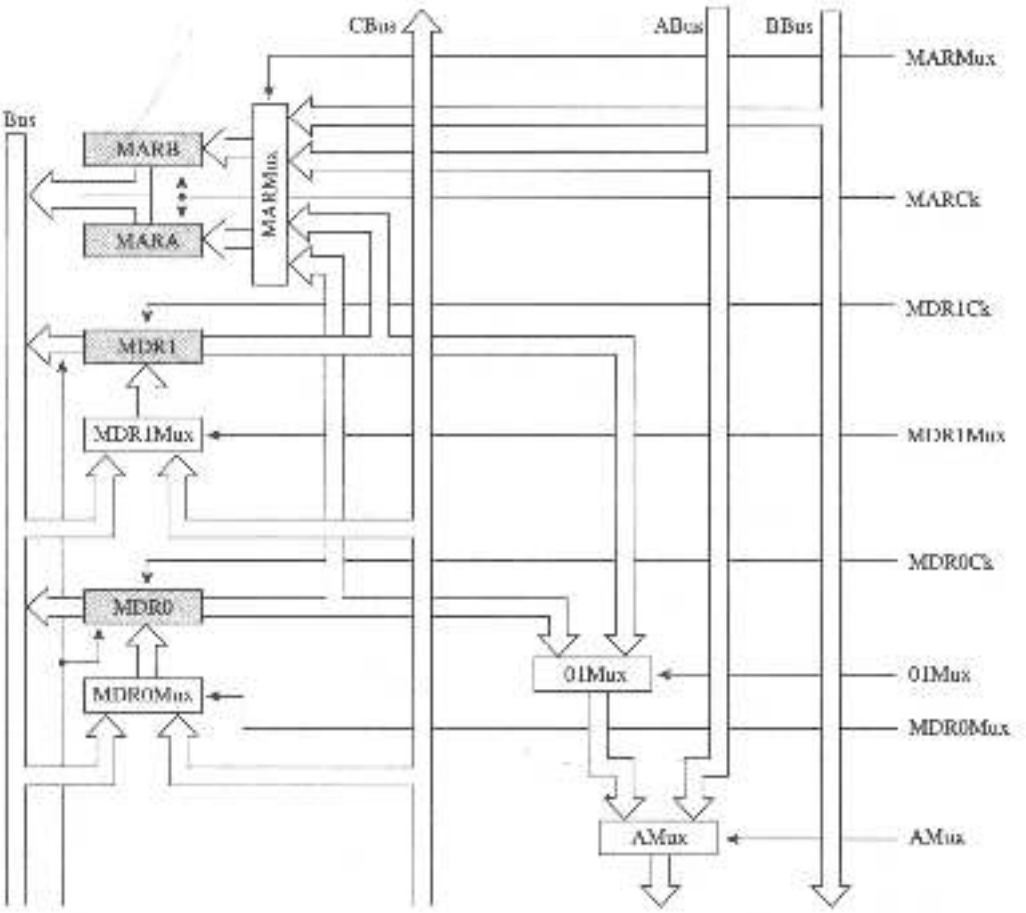


图 12-19　对 12-14b 的 2 字节数据总线的特殊硬件单元

```
// MDR <- Mem[OprndSpec].
1. A=9, B=10, MARMux=1; MARCk
2. MemRead
3. MemRead, MDR0Mux=0, MDR1Mux=0; MDR0Ck, MDR1Ck

// MAR <- MDR.
4. MARMux=0; MARCk

// MDR <- two-byte operand.
5. MemRead
6. MemRead, MDR0Mux=0, MDR1Mux=0; MDR0Ck, MDR1Ck

// X <- MDR, high-order first.
7. 01Mux=0, AMux=0, ALU=0, ANDZ=0, CMux=1, C=2; NCk, ZCk, LoadCk
8. 01Mux=1, AMux=0, ALU=0, ANDZ=1, CMux=1, C=3; ZCk, LoadCk
```

图 12-20　用图 12-19 的 2 字节数据总线实现采用间接寻址方式的装入指令的控制信号

12.2.3　3 个优化领域

衡量一台机器性能的最终指标是它执行起来有多快。图 12-21 说明机器执行程序的时间

是三个因子的乘积。等式中使用的“指令”指的是ISA3层的指令。

前两个因子之间有关联，但是与第三个因子无关。第一个因子下降通常会导致第二个因子上升，反之亦然。也就是说，如果设计ISA3机器的方法导致书写一个给定程序的指令数减少，那么通常会付出执行每条指令的周期数增加的代价。反过来，如果设计ISA3机器导致每条指令使用的周期数尽可能少，通常就必须让指令尽量简单，使得编写一个给定的程序需要更多的指令。

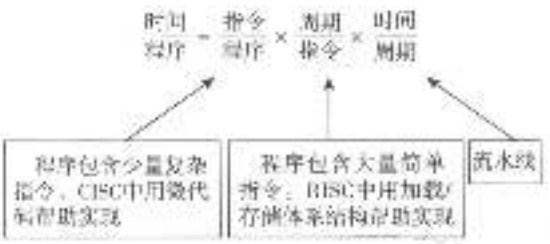


图12-21　冯·诺依曼机器中执行时间的3个组成部分

等式中的第三个因子是基于并行性的，需要重新组织控制区，使得集成电路上更多的子系统能够并发地运行。不过，它和前面两个因子之间没有折中关系。可以在设计中引入流水线，降低每个周期的时间。每个程序所需的时间都会降低，不管选择以第二个因子为代价减小第一个因子，还是以第一个因子为代价减小第二个因子。

在计算历史的早期，设计者的注意力集中于第一个因子。采用这种方法的部分原因是内存的成本太高。程序需要尽可能小，才能装进可用的内存中。指令集和寻址方式的设计需使编译器能很容易地从HOL6翻译到Asmb5。Pep/8就是这种设计的示例。而这里主要是出于教学的原因，而不是因为内存有限。汇编语言用于讲授计算机系统中典型抽象层级之间的翻译原理。翻译过程简单的话，这些原理学习起来就容易。

在20世纪80年代早期，计算领域发生了变化。硬件变得越来越便宜，内存子系统变得越来越大。一些设计者，著名的有IBM的John Cocke，UC Berkeley的David Patterson和Stanford的John Hennessy，开始宣扬基于以增加第一个因子、降低第二个因子为基础的设计。他们的设计特点是ISA3指令的个数很少，寻址方式也较少。这种设计的名称是精简指令集计算机（Reduced Instruction Set Computer，RISC）。与之相对的旧设计开始称作复杂指令集计算机（Complex Instruction Set Computer，CISC）。因为前两个因子之间有折中关系，所以许多人怀疑最终的净结果能否得到更快的计算机。RISC的观点是这样可以在给定的集成电路上比CISC设计获得更高的并行度，尤其是使用流水线的时候。

从20世纪80年代以来，几乎所有新设计的CPU都是RISC机器，其中最著名的有：ARM芯片，在手机市场占据主导；MIPS芯片，基于Stanford设计，用在服务器和任天堂的游戏控制器中；Sparc芯片，基于Berkeley设计，Sun将其用于服务器和工作站；还有PowerPC，基于IBM设计，用在IBM服务器和工作站中。不过，还有一种CISC设计继续主导着桌面市场，那就是Intel的IA-32，其是表12-15列出的芯片的直接继承者。它能够继续主导主要是因为与系列中以前的所有芯片都兼容。把应用程序和操作系统迁移到具有不同ISA3指令和寻址方式的芯片上代价是很高的。而且，CISC设计者采用了RISC的理念，对RISC核进行了一层抽象，核的细节隐藏在更低的层次上，并且在ISA3层实现了CISC机器。

616
↓
620

Maurice V. Wilkes

Maurice Wilkes 1913年生于英格兰Staffordshire郡的Dudley，1936年从剑桥大学获得博士学位。次年，学校成立了计算机实验室，任命Wilkes领导建造一个新的差分分析器。二战爆发时，他离开学校参战，1945年返回实验室。

1946 年 Wilkes 读到冯·诺依曼对 EDVAC 的初步报告，该报告描述了由取指－译码－加 1－执行周期控制的存储程序计算机的概念。他立即“认为这会马上变成一件真实的事”，并投入其中开始构造一个基于冯·诺依曼原理的计算机。他参加了八月在宾夕法尼亚大学进行的一系列讲座，遇到了冯·诺依曼的同事 Herman Goldstine、John Mauchly 和 Presper Eckert。返回剑桥后，他决定设计和建造电子延迟存储自动计算机（Electronic Delay Storage Automatic Computer，EDSAC）。

1949 年 5 月 6 日，世界上第一台冯·诺依曼机器 EDSAC 在剑桥实验室上线了。Wilkes 在手写笔记写道：“机器第一次运行，打印方块表（0～99），程序运行时间 2 分 35 秒。”打印方块是对这个机器漂亮的第一次展示。但是很快 Wilkes 就开始要求更多任务了。程序通过在纸带上打孔输入计算机。Wilkes 用打孔带编程让 EDSAC 解决空气差分方程，$y''-ty=0$，用在物理中对光的反射建模。程序有 126 行，第一次编写的时候有 20 个错误。下面的引言说的就是他奔波在纸带打孔机和 EDSAC 机房之间。Wilkes、D. J. Wheeler 和 S. Gill 在 1951 年出版了第一本编程书籍《The Preparation of Programs for an Electronic Digital Computer》（为电子数字计算机准备程序）。

也是在 1951 年，Wilkes 发表了一篇题为“The Best Way to Design an Automated Calculating Machine”（设计自动计算机器的最好方式）的论文，讲述他称为微程序设计的新技术。Wilkes 不仅发明了 Mc2 抽象层，还构建了第一台带有微程序设计的控制区的计算机。EDSAC 2 在 1958 年早些时候投入使用。Mc2 最是实际上所有商用计算机的标准设计，直到 20 世纪 80 年代加载/存储体系结构变得流行起来。

EDSAC 2 的另一项设计创新是它的位片组织结构。EDSAC 机器是用真空管设计的，因为当时晶体管还没有广泛使用。发现到和替换一个故障的真空管很困难而且很耗时。图 11-40b 的 512×1 位内存芯片是一个 512 字节的 1 位内存片，按照与此类似的方式，EDSAC 2 的数据区是一组真空管的架架，每一个架子是整个内存的一片。如果一个真空管损坏，可以用一个好的架子替换坏的，极大地简化了维护。



Wilkes 是英国计算机学会的杰出会士，皇家学会的会士和皇家工程院会士，并于 1967 年获得 ACM 图灵奖。Maurice V. Wilkes 于 2010 年 11 月 29 日逝世。

“我依然清楚地记得那个时刻，当时我意识到从此以后我生命中很大一部分时间就要花在给自己的程序查找错误上了。”

——Maurice V. Wilkes

12.2.4　微代码

图 12-1 中的 CPU 划分为数据区和控制区。给定一个实现一条 ISA3 指令所需的控制信号序列，例如图 12-8 中实现 STBYTEA 指令的序列，问题是该如何设计控制区来产生这样的信号序列呢？

微代码背后的理念是控制信号的序列实际上是一个程序。图 12-4 描述的是冯·诺依曼周期，看上去都很像一个 C++ 程序。设计控制区的一种方式是创建 Mc2，即微代码抽象层，

位于 ISA3 和 LG1 之间。像所有抽象层一样，这一层有自己的语言，包括一组微编程语句。控制区是一个单独的微机器，有自己的微内存 uMem 和自己的微程序计数器 uPC，以及自己的微指令寄存器 uIR。和 ISA3 层机器不同，Mc2 层的机器只有一个烧入 uMem ROM 当中的程序。一旦芯片制造完成，微程序就不可以改变了。这个程序包含一个循环，唯一的目的是实现 ISA3 的冯·诺依曼周期。

图 12-22 给出 Mc2 层用微代码实现的 Pep/8 的控制区，图中没有画出的数据区在它的左边。这个图中包含的数据通路是对图 12-2 的修改，以支持图 12-19 的两字节数据总线。从数据区到控制区总共有 12 条数据线，8 条来自 BBus，4 条来自状态位。总共有 36 条控制线连到数据区。

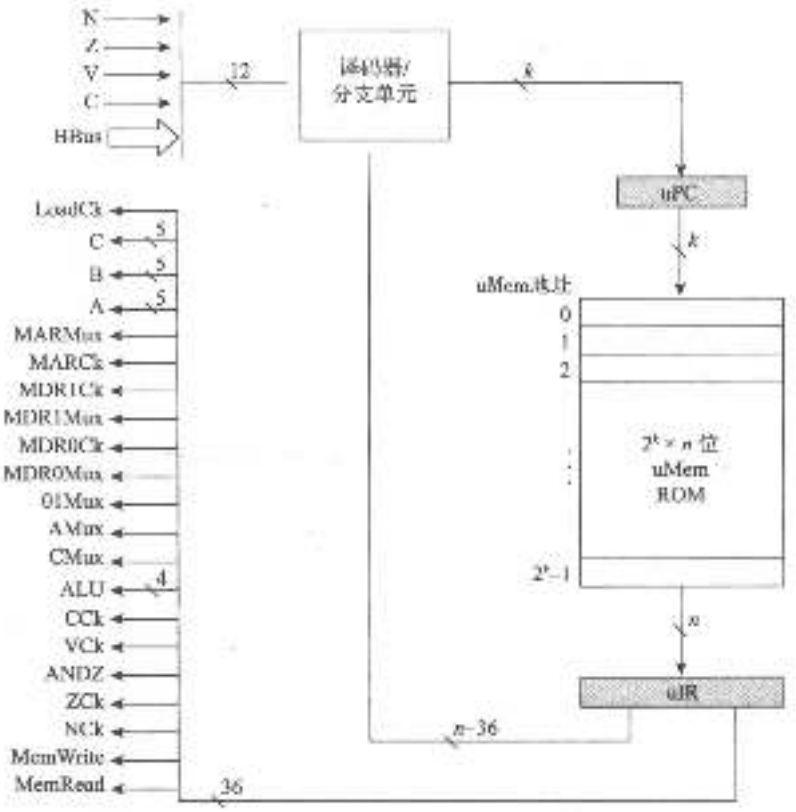


图 12-22　Pep/8 CPU 控制区的微代码实现

微程序计数器包含下一条要执行的微指令的地址。uPC 是 k 位宽，所以可以指向 uMem 中 2^k 条指令中的任意一条。微指令是 n 位宽的，所以 uMem 中每个单元的宽度和 uIR 的宽度也都为 n。在 Mc2 层，没有理由要求微指令的宽度必须是 2 的幂。n 可以是你想要的任意奇怪的值。能这么灵活是因为 uMem 只包含指令不包含数据，指令和数据没有混合到一起，所以没必要要求内存单元的大小必须适合两者。

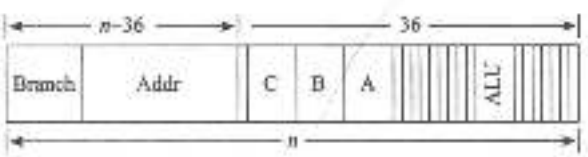


图 12-23　微指令的指令格式

图 12-23 给出了微指令的指令格式。最右边的 36 位是控制信号，要发送到数

据区；剩下的字段分为两部分：Branch（分支）字段和 Addr（地址）字段。ISA3 层的程序计数器每次加 1，因为正常的控制流把指令顺序存放在内存中并且顺序执行，所以唯一的变化是由于分支指令导致的 PC 变化。但是在 Mc2 层，uPC 的变化不是加 1，而是每条微指令包含计算下一条微指令地址的信息。Branch 字段指明如何计算下一条微指令的地址，Addr 字段包含用于计算的数据。

例如，如果下一条微指令不依赖于来自数据区的 12 条信号，那么 Branch 字段会表明这是一个无条件分支，Addr 将是下一条指令的地址。实际上，每条指令都是一条分支指令。要按照顺序执行一组微指令，必须让每条微指令无条件分支转移到下一条微指令。图 12-22 的译码器 / 分支（decoder/branch）部件的作用是，当 Branch 字段表明是一个无条件分支时，就让 Addr 直接通过进入 uPC，无论来自数据区的 12 条线的值是什么。

BRLT 的实现。这个例子能说明条件微分支指令的必要性，BRLT 的 RTL 说明是

$$N = 1 \Rightarrow PC \leftarrow Oprnd$$

如果 N 位是 1，则 PC 获得操作数。要实现 BRLT，微指令必须检查 N 位的值，要么什么都不做，要么分支跳转到另一组微指令序列，用操作数替代 PC。这条微指令包含的 Branch 字段会指明下一个地址是用 N 和 Addr 一起计算得出的。如果 N 为 0，计算会产生一个地址；如果 N 为 1，会产生另一个地址。

通常来说，条件微分支需要根据 Addr 和判断条件依赖的，来自数据区的信号计算出下一条微指令的地址。最大的条件分支是决定要执行哪条 ISA3 指令的分支，换句话说，就是冯 · 诺依曼周期的译码部分。对 ISA 指令译码的微指令的 B 字段为 8，这会把 IR 的第一个字节放到 BBus。（IR 的寄存器地址参见图 12-2。）Branch 字段会说明是指令译码，译码器 / 分支单元会输出实现该指令的微指令序列中第一条的地址。

微指令中 Branch 和 Addr 字段的细节，以及译码器 / 分支单元的实现不在本书讨论范围之内。虽然图 12-22 和图 12-23 省略了微代码层设计中很多实际问题，但还是能说明 Mc2 层的基本设计元素。

12.3　MIPS 机器

Pep/8 是累加器机器的示例，因为计算总是发生在位于内存的操作数和累加器或变址寄存器之间。例如，加法执行 ADDA，它把操作数加到累加器，并把结果存储在累加器中。Peop/8 也有一组丰富的寻址方式，能够用较少的指令把 C++ 程序翻译为 Pep/8 汇编语言。

12.3.1　装入 / 存储体系结构

与累加器机器对比的是装入 / 存储（load/store）机器，它简化图 12-21 中的第二个因子以降低程序的执行时间。MIPS 机器是商用制造的装入 / 存储机器的经典例子，它是一个 32 位机器，CPU 中有 32 个 32 位寄存器。图 12-24 按比例画出了 MIPS CPU 中的寄存器和 Pep/8 中的寄存器。

每个寄存器都有一个特殊的汇编命名，以 $ 符号开头。$0 是一个常数 0 寄存器，类似于图 12-2 中的寄存器 22，不过它在 ISA3 层是可见的。$v0 和 $v1 用于子例程返回值，$a0 ~ $a3 用于子例程的参数，类似于 6.5 节中操作符 new 的调用协议。以 $t 开头的寄存器是临时的，在函数调用之间不会保留，而以 $s 开头的寄存器会保存起来，在函数调用之间得以保留。$k 寄存器预留给操作系统内核，$gp 是全局指针，$sp 是栈指针，$fp 是帧指针，

而 Sra 是返回地址。

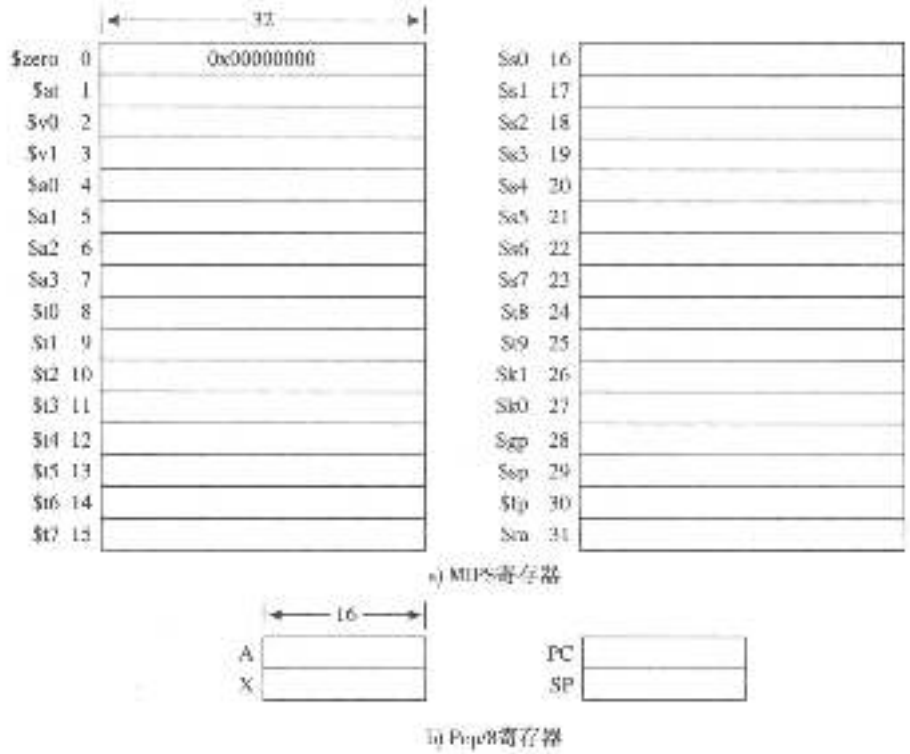


图 12-24　MIPS 和 Pep/8 CPU 寄存器比较

和大多数微处理器相比，Pep/8 是很小的机器。图 12-24 表明 MIPS CPU 寄存器的总位数是 Pep/8 CPU 寄存器的 16 倍，这还不算 MIPS 拥有的另一组浮点寄存器，而 Pep/8 没有浮点寄存器。即使数量上有这么大的差距，从两个方面来说 MIPS 还是比 Pep/8 更简单：它有更少的寻址方式，以及它的指令长度都是一样的。

图 12-4 给出的是 Pep/8 的冯·诺依曼周期。因为存在有两种指令，所以这个周期比较复杂。这两种指令分别是一元指令和三字节非一元指令。RISC 体系架构的一个目标是尽量简化，直到每条指令能在一个周期内执行完。这个目标表明每条指令有相同的长度，在 MIPS 机器里是 4 个字节。为了提高性能，内存对齐问题要求每条指令的第一个字节必须存储在能被 4 整除的地址处。对应于图 12-14b 内存芯片的地址线 A2 ~ A31 和数据线 D0 ~ D31。没有地址线 A0 和 A1，因为每次取数一次会取出 4 字节，所以一次访存就能取出整条指令。图 12-25 展示了 MIPS 机器的冯·诺依曼周期。没有 if 语句来确定指令的大小。

图 12-25 中 MIPS 的冯·诺依曼周期是一个无限循环，这比 Pep/8 的循环更实际。实际机器没有 STOP 指令，因为当一个应用程序结束后操作系统会继续执行。

```
do
    将 PC 中未处理的指令指示符
    PC ← PC + 4
    译码该指令指示符
    执行取出的指令
while (true)
```

图 12-25　MIPS 冯·诺依曼执行周期的伪代码描述

相比 Pep/8 的八种寻址模式，MIPS 只有五种寻址模式，如图 12-26 所示。Pep/8 的操作数指示符总是 16 位，但是 MIPS 的操作数指示符大小取决于寻址方式。MIPS

指令总是包括指令指示符和一个或多个操作数指示符。因为一条指令必须总是正好 4 个字节，所以指令指示符和操作数指示符必须能装进 32 位中，因此任意操作数指示符必须小于 32 位。

寻址方式	OprndSpec 的大小	操作数
立即数	16 位	OprndSpec
寄存器	5 位	Reg [OprndSpec]
基址	5 位和 16 位	Mem [Reg [OprndSpec 1] + [OprndSpec 2]]
PC 相对	16 位	Mem [(PC + 4) + OprndSpec * 4]
伪直接	26 位	Mem [(PC + 4)<0..3> : OprndSpec * 4]

图 12-26　MIPS 寻址方式

立即数寻址、寄存器寻址和基址寻址方式用于访问数据，其中只有基址寻址访问主存。图 12-26 中的记号 Reg 代表寄存器（register），类似于 Mem 表示内存（memory）。因为图 12-24a 的寄存器体中有 32 个寄存器，2^5 等于 32，所以要访问这些寄存器就需要 5 位。Reg[OprndSpec] 表示操作数指示符指定的地址处寄存器的内容。

PC 相对寻址和伪直接（pseudodirect）寻址方式用于访问指令，没有直接寻址方式。取指时，总是从相对于程序计数器的位置取。PC 相对寻址方式中，16 位操作数指示符会先乘以 4（因为每条指令占 4 个字节），然后再加上加 1 之后的程序计数器。加法的结果是操作数在内存中的地址。伪直接寻址方式中，冒号表示连接，连接的左边是加 1 之后的程序计数器的前 4 位（高位）。连接的右边是 26 位操作数指示符乘以 4，也就是左移 2 次，得到一个 28 位数字。连接得到的结果是 32 位地址。条件分支指令使用 PC 相对寻址，而无条件跳转指令使用伪直接寻址。

12.3.2　指令集

图 12-27 是一些 MIPS 指令的总结。虽然有些指令用了额外的字段进一步指明该指令的行为，但是所有指令都有一个 6 位操作码。标记为 sssss 和 ttttt 的操作数指示符是 5 位的源寄存器字段，标记为 ddddd 的为 5 位目标寄存器字段，而标记为 bbbbb 的是 5 位基址寄存器字段。内容为 i 字符的字段是立即数操作数指示符，对加法，立即数做符号扩展，对 AND 和 OR，立即数做零扩展。内容为 a 字符的字段是地址操作数指示符，在地址计算中做符号扩展作为偏移量。标记为 hhhhh 的操作数指示符是移位指令的 5 位位移量。

本节中使用的术语和表示法与官方 MIPS 文档有些不同。一些术语做了修改，以和 Pep/8 的描述相一致。例如，Pep/8 对位的编号是从左到右的，而 MIPS 的惯例是从右到左。本书中，字母 d 总是表示目标寄存器，字母 b 总是表示基址寄存器，而在 MIPS 文档中，字母 t 有时会表示目标寄存器。

图 12-28 比较了 MIPS 和 Pep/8 加法指令的机器语言格式。MIPS 加法指令不访问内存中的操作数，相反，它把两个寄存器的内容相加，把和放到第三个寄存器中。图 12-28a 中，rs 和 rt 是两个源寄存器，而 rd 是存放和的目标寄存器。名为 *shamt* 的字段是位移量，用在移位指令中，不适用于加法指令。名为 *funct* 的字段是功能字段，和操作码字段联合起来指明操作。寻址方式叫作寄存器寻址，原因很明显。

助记符	含义	二进制指令编码
add	加法	0000 00ss ssst tttt dddd d000 0010 0000
addi	立即数加法	0010 00ss ssdd dddd iiii iiii iiii iiii
sub	减法	0000 00ss ssst tttt dddd d000 0010 0010
and	按位 AND	000 00ss ssst tttt dddd d000 0010 0100
andi	按位立即数 AND	0011 00ss ssdd dddd iiii iiii iiii iiii
or	按位 OR	0000 00ss ssst tttt dddd d000 0010 0101
ori	按位立即数 OR	0011 01ss ssdd dddd iiii iiii iiii iiii
sll	逻辑左移	0000 0000 000t tttt dddd dnnn nn00 0000
sra	算数右移	0000 0000 000t tttt dddd dnnn nn00 0011
srl	逻辑右移	0000 0000 000t tttt dddd dnnn nn00 0010
lb	字节加载	1000 00bb bbbd dddd aaaa aaaa aaaa aaaa
lw	字加载	1000 11bb bbbd dddd aaaa aaaa aaaa aaaa
lui	立即数高位加载	0011 1100 000d dddd iiii iiii iiii iiii
sb	存储字节	1010 00bb bbbt tttt aaaa aaaa aaaa aaaa
sw	存储字	1010 11bb bbbt tttt aaaa aaaa aaaa aaaa
beq	如果等于分支	0001 00ss ssst tttt aaaa aaaa aaaa aaaa
bgez	如果大于或等于 0 分支	0000 01ss sss0 0001 aaaa aaaa aaaa aaaa
bgtz	如果大于 0 分支	0001 11ss sss0 0000 aaaa aaaa aaaa aaaa
blez	如果小于或等于 0 分支	0001 10ss sss0 0000 aaaa aaaa aaaa aaaa
bltz	如果小于 0 分支	0000 01ss sss0 0000 aaaa aaaa aaaa aaaa
bne	如果不等于分支	0001 01ss ssst tttt aaaa aaaa aaaa aaaa
j	跳转（如条件分支）	0000 10aa aaaa aaaa aaaa aaaa aaaa aaaa

图 12-27　MIPS 指令集的一些指令

加法指令的 RTL 说明是

```
rd ← rs + rt
```

对应的 MIPS 汇编语言语句是

```
add rd,rs,rt
```

汇编语言中操作数的顺序和机器语言中的顺序不同。

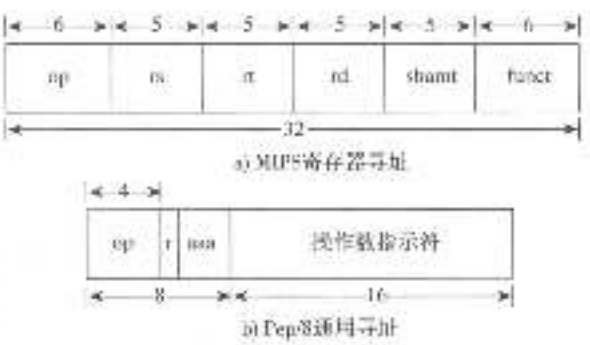


图 12-28　MIPS 和 Pep/8 加法指令格式比较

装入/存储机器的一个关键原理是所有计算都是对寄存器中的值进行的，一条指令不可以把一个来自内存的值加到来自寄存器的值上。必须首先把内存中的值装入另一个寄存器，然后再把两个寄存器相加。能够访存的指令只能是装入和存储指令。MIPS 机器中，内存是按字节寻址的，字长为 32 位，字数据按能被 4 整除的内存地址对齐。两条用于访存的指令是：

- lw 用于装入字
- sw 用于存储字

图 12-29 展示了 lw 指令的 MIPS 指令格式，rb 是基址寄存器，而 rd 是目的寄存器。

装入指令的 RTL 说明是

```
rd ← Mem[rb + address]
```

而对应的 MIPS 汇编语言语句是

```
lw rd,address(rb)
```

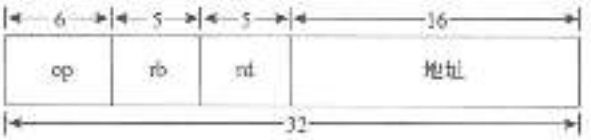


图 12-29　采用基址寻址的加载指令的 MIPS 指令格式

除了源寄存器 rt 代替了目的寄存器 rd 外，存储指令的指令格式与图 12-29 中的一样，参见图 12-27。存储指令的 RTL 说明是

```
Mem[rb + address] ← rt
```

对应的 MIPS 汇编语言语句是

```
sw rt,address(rb)
```

标号为 address 的字段不是绝对地址，而是一个地址偏移量（offset），要把它加到 rb 上才能得到绝对地址。rb 寄存器 32 位宽，但是地址字段只有 16 位宽。CPU 把地址解释为有符号整数，这样一来，装入指令可以从寄存器 rs 中的地址装入范围 $\pm 2^{15}$ 或 32 768 字节内的字。

Pep/8 寻址方式中最类似于 MIPS 装入和存储指令的寻址方式是变址寻址，其中的操作数是

```
Oprnd = Mem[OprndSpec + X]
```

最大的区别在于寄存器的使用。Pep/8 中，寄存器 X 包含索引的值，而 MIPS 机器中，寄存器 rb 包含数组第一个元素的地址，即数组的基址。这也是寄存器 rb 称作基址寄存器的原因，这种寻址方式称作基址寻址。

在像 Pep/8 这样的累加器机器中，变量的当前值存放在内存中。不过装入 / 存储机器有大量寄存器。编译器把寄存器和变量关联起来，在寄存器中完成所有计算。在四种情况中，变量的值必须存储到内存中。第一，程序中变量数目超出了可用的寄存器数量。要对无法存放在 CPU 中的变量进行计算，就要把寄存器中现有的值“溢出”到主存中。第二，在做递归调用时，必须把局部变量复制到内存中的运行时栈。第三，要输出一个值，必须先把它拷贝到内存。第四，数组值需要的存储太多，无法装入寄存器中。不过编译器会把一个 $s 寄存器和数组变量关联起来，使之包含数组第一个元素的地址。

625
630

例 12.1　假设 C++ 编译器把 $s1 和数组 a、$s2 和变量 g、$s3 和数组 b 关联起来，把下面的语句

```
a[2] = g + b[3];
```

翻译成 MIPS 汇编语言

```
lw $t0,12($s3)   # Register $t0 gets b[3]
add $t0,$s2,$t0  # Register $t0 gets g + b[3]
sw $t0,8($s1)    # a[2] gets g + b[3]
```

注释以“#”符号开始。装入指令地址字段为 12，因为它要访问 b[3]，每个字是 4 字节，所以 3 × 4 = 12。类似地，存储指令的地址字段是 8，因为索引值在 a[2]。这些指令的机器语言翻译是

```
100011 10011 01000 0000000000001100
000000 10010 01000 0100000000000010
101011 10001 01000 0000000000001000
```

每条指令的前6位是操作码。图12-24表明$s0是寄存器8(dec) = 01000(bin)，$s3是寄存器19(dec) = 10011(bin)，$s2是寄存器18(dec) = 10010(bin)，而$s1是寄存器17(dec) = 10001(bin)。□

如果想要访问一个下标为变量的数组元素，情况就更复杂一些。与Pep/8不同，MIPS没有变址寄存器。所以，编译器必须生成代码，把索引值加到数组第一个元素的地址，获得想要引用元素的地址。在Pep/8中，这个加法运算是在Mc2层用变址寻址自动完成的。但是装入/存储机器的设计哲学是拥有较少的寻址方式，付出的代价是程序需要更多语句。

在Pep/8中，一个字有2个字节，所以索引必须左移一次，相当于乘以2。在MIPS中，一个字是4个字节，所以索引要左移两次，相当于乘以4。MIPS指令sll用于逻辑左移，用图12-28a中的shamt字段指定位移的量。

例12.2　把$s0的内容左移7位并把结果放到$t2中的MIPS汇编语言语句是

```
sll $t2,$s0,7
```

机器语言的翻译是

631

```
000000 00000 10000 01010 00111 000000
```

第一个字段是操作码；这条指令没有使用第二个字段，因此都被设置为0；第三个字段是rt字段，表明$s0，寄存器16(dec) = 10000；第四个是rd字段，表明$t2，寄存器10(dec) = 01010(bin)；第五个是shamt字段，表明移位的数量；最后一个是funct字段，和操作码一起表明是sll指令。□

例12.3　假设C++编译器把$s0和变量i、$s1和数组a以及$s2和变量g关联起来，把下面的语句

```
g = a[i];
```

翻译成MIPS汇编语言如下

```
sll $t0,$s0,2     # $t0 gets $s0 times 4
add $t0,$s1,$t0   # $t0 gets the address of a[i]
lw $s2,0($t0)     # $s2 gets a[i]
```

注意装入指令的地址字段是0。□

和Pep/8一样，MIPS机器有直接寻址方式。有一个特殊的加法指令，助记符为addi，表示立即数加。图12-30给出立即数寻址的指令格式，其结构跟图12-29的基址寻址方式一样，除了16位的字段是立即数操作数，而不是地址偏移量。

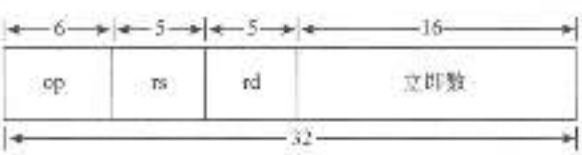


图12-30　采用立即数寻址的MIPS指令格式

例12.4　在运行时栈上分配4字节存储，要执行

```
addi $sp,$sp,-4  # $sp <- $sp - 4
```

这里的-4不是地址，而是立即数。机器语言翻译为

```
001000 11101 11101 1111111111111100
```

632　这里$sp是寄存器29。

你可能已经注意到 addi 指令有一个限制。常数字段只有 16 位宽，而 MIPS 是 32 位机器，使用立即数寻址应该能加一个 32 位的立即数。这里是加载 / 存储体系结构哲学的又一个示例，其主要目标就是寻址方式较少的简单指令。Pep/8 设计允许指令宽度不同，也就是一元指令和非一元指令宽度可以不同。图 12-4 展示了这样的设计决定怎样使得冯·诺依曼周期的取指部分更复杂的：硬件必须取出指令指示符，对它译码以决定是否需要取操作数指示符。这种复杂化是与装入 / 存储哲学相违背的，后者要求指令简单，能够很快地译码出来。这样的简化目标要求所有的指令长度相同。

但是，如果所有的指令都是 32 位宽，那么一条指令又怎么可能包含一个 32 位的立即数常数呢？那样指令格式中就没有操作码的地方了。这里利用了降低图 12-21 中的第二个因素，代价是增加第一个因素。解决 32 位立即数常数的方法是要求执行两条指令。为了做到这点，MIPS 提供了 lui 指令，意思是装入立即数高位，它把一个寄存器的高 16 位设置为它的立即数操作数，并且把低位设置为全 0。第二条指令会设置低 16 位，通常是使用或立即数指令 ori。

例 12.5　假设编译器把寄存器 $s2 与变量 g 关联起来，把 C++ 语句

```
g = 491521;
```

翻译成 MIPS 汇编语言

```
lui $s2,0x0007
ori $s2,$s2,0x8001
```

十进制数 491 521 的二进制需要不止 16 位，491 521(dec) = 0007 8001(hex)。　□

MIPS 最复杂的寻址方式也不会复杂过把指令的一个地址字段加到一个寄存器上。相比之下，Pep/8 最复杂的寻址方式是栈变址间接寻址。

```
Oprnd = Mem[Mem[SP + OprndSpec] + X]
```

来看看要执行一条栈相对间接指令 CPU 必须做些什么。首先，必须把 SP 加上 OprndSpec。其次，必须从那个地址进行一次内存取。然后，把取出的数加到 X。最后，从那个地址取出操作数。这很复杂，而 Pep/8 是 CISC。

但是栈相对间接寻址的目的是什么呢？6.4 节表明访问作为参数传递的数组需要它，6.5 节表明访问通过 new 操作符动态分配的结构中的元素需要它。MIPS 中没有栈相对间接寻址就意味着它无法传递数组参数吗？当然不是。这只是意味着编译器必须生成更多汇编语言语句来计算操作数的地址。如图 12-21 所示，程序包含大量简单的语句。

RISC 的论点是要执行更多指令，但是每条指令需要更少的周期。不仅如此，更简单的设计可以从集成电路中获取更多的并行性，所以最终效果是更快的芯片。装入 / 存储机器的一个特点是没有微代码抽象层。这种机器的控制区硬件中直接有一组有限状态机，产生数据区的控制信号。一般来说增加抽象层会付出性能的代价，而消除抽象层通常会获得更好的性能。

12.3.3　高速缓存

Pep/8 控制序列要求内存读和写需要 2 个周期，因为通过主系统总线访问内存子系统要花费额外的时间。虽然这个要求使得模型更现实，但是实际上主存访问时间和 CPU 周期时间之间速度的不匹配要更严重。假设主存访问需要 10 个周期而不是 2 个周期，想象一下控制序列看上去会像什么样子：很多个 MemReads 周期。大部分时间中 CPU 在等待内存读，浪费了本来可以用来推进作业执行的周期。

但是如果可以预测未来呢？如果提前知道程序需要从内存来的哪个字，那么可以设立一个更贵、速度更高的小存储器，使其紧邻着CPU，称为高速缓存（cache），提前从主存取出指令和数据，从而可以立即为数据区所用。当然，没有人能预测未来，所以这种机制是不可能的。不过，即使不能100%准确地预测未来，如果能达到95%的准确率呢？当预测准确时，访问高速缓存的时间几乎是立即的。如果预测准确的比率足够高，加速比就会很客观了。

633 / 634

问题是如何进行预测。假设能够监控地址线和CPU在执行典型任务时的所有内存请求，那么会发现地址序列是完全随机的吗？也就是给定一个地址请求后，可以预期下一个请求与这一个邻近吗？或者预期它与这个请求的距离是完全随机的？

预期连续的内存请求会彼此接近是基于内存中存储的两样东西，原因也有两个。首先，CPU会在冯·诺依曼周期的取指部分访问指令，只要没有分支指令要执行距离远的指令，CPU就会请求聚集在一起的指令。其次，CPU必须访问来自内存的数据。不过第6章中的汇编语言程序都会把它们的数据在内存中聚集存放。应用程序和堆存放在内存低地址，操作系统和运行时栈存放在高地址。不过也应该注意到，在某些时间段访问会集中在邻近的地方。

内存访问并不是随机的，这个现象称为引用局部性（locality of reference）。如果内存访问是完全随机的，那么高速缓存就完全没有用，因为无法预测要从内存读哪些字节，也就无从预先装入。幸运的是典型的访问会展现出两类局部性：

- 空间局部性：邻近之前访问过的地址很有可能在临近的未来再被请求。
- 时间局部性：之前访问过的地址本身在临近的未来很有可能再被请求。

时间局部性来自于程序中经常使用的循环。

当CPU请求来自于内存的加载时，高速缓存子系统首先会看所请求的数据是否已经加载进高速缓存，这个事件称为高速缓存命中（cache hit）。如果是这样，就直接传送数据。如果不是，则发生高速缓存不命中（cache miss）事件，需从主存取出数据，CPU必须等待更长的时间。当数据最终到达时，会被加载进高速缓存并送给CPU。因为数据刚刚被请求过，所以有很高的概率在不远的将来再被请求。把这样的数据保存在高速缓存中利用了时间局部性。不仅把所请求的数据放进高速缓存，还把所请求的字节附近的一些字节也装入其中，这利用了空间局部性。虽然带入了一些没有被请求的字节，但这是基于对未来的预测进行的提前装载。在不远的将来访问它们的概率很高，因为它们的地址与前面访问过字节的地址距离很近。

为什么不用像高速缓存这样的高速电路构建主存，把它放在高速缓存的位置，也就不需要高速缓存了？这是因为高速内存电路需要在芯片上占用太多面积。最快和最慢的内存技术之间有巨大的大小和速度的差别，这是经典的空间/时间折中。每个存储单元的空间越大，内存操作的速度就越快。

635

内存技术在这两种极端之间提供了多种设计，其中在CPU和主存子系统之间使用三层高速缓存是最典型的情况：

- 在CPU芯片上把L1指令和数据高速缓存分开。
- 在CPU封装中使用统一的L2高速缓存。
- 在处理器主板上使用统一的L3高速缓存。

图12-31给出了这样的三层结构。图中，封装是一个自包含的电子部件，可以单独购买并安装在电路主板上。计算机制造商通常会设计电路主板，购买封装，然后把它们安装到主板上，把主板放进机箱，再把机箱当作一台计算机出售。L1高速缓存比L2高速缓存更小更

快，L2 高速缓存又比 L3 高速缓存更小更快，而 L3 又比主存子系统更小更快。L1 高速缓存收到来自 CPU 的内存请求，如果缓存不命中，再把请求传递到 L2 缓存，如果 L2 缓存不命中，就再传递到 L3，如果 L3 不命中，就传递到主存子系统。

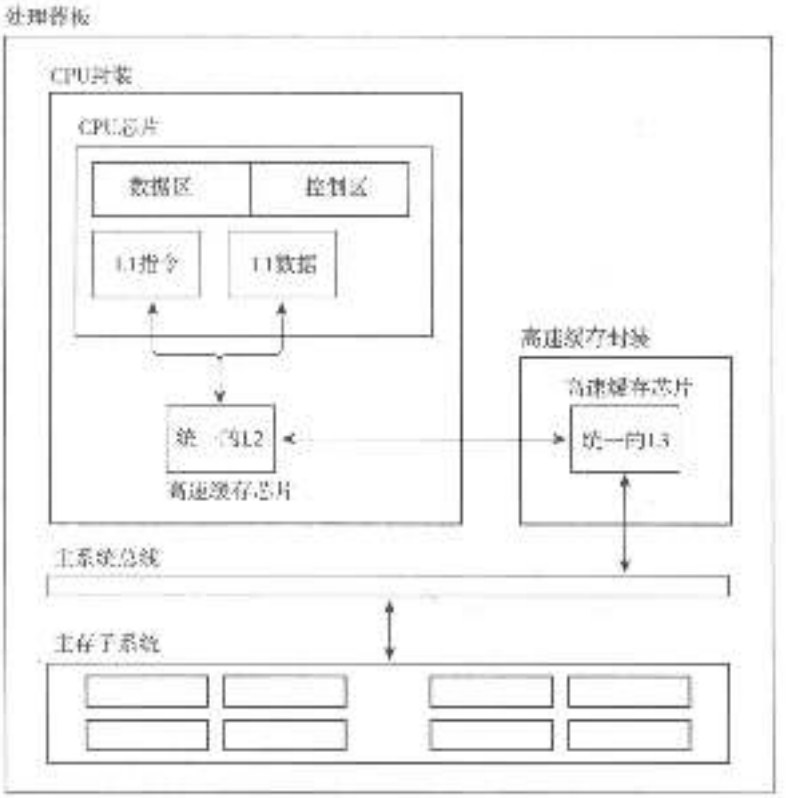


图 12-31　一个典型计算机系统中的三层高速缓存结构

L1 高速缓存位于 CPU 集成电路中，分为指令高速缓存和数据高速缓存。CPU 会区分冯·诺依曼周期的取指令和取操作数的取数据。L2 和 L3 高速缓存也称为统一高速缓存，因为它们对指令和数据不做区分，混合存储。L1 高速缓存的典型大小是 32KB 或 64KB。（说明一下 Pop/8 有多小，它的整个内存正好能装进一个典型的 L1 高速缓存中。）L1 高速缓存必须按照 CPU 的速度工作，L2 高速缓存的速度通常是 L1 的 1/2 或 1/4，时间是后者的 2 ~ 4 倍。L3 通常又比 L2 慢和大了 4 倍。随着主系统总线技术的发展，一些设计会省略 L3 高速缓存，因为 L2 和主存之间没有足够大的速度差，不需要再多增加一层。

有两种高速缓存设计方法：

- 直接映射高速缓存。
- 组相联高速缓存。

两者中比较简单的是直接映射高速缓存，图 12-32 是一个例子。和前面的例子一样，这个例子不切实际地小以帮助进行描述。

这个例子是一个具有 16 条地址线和 2^{16} = 64KB 主存的系统。内存被划分成 16 字节的块，称为高速缓存行。当发生高速缓存不命中时，系统不仅会加载所请求的字节，还会加载包含这个被请求的字节所在行的所有 16 个字节。高速缓存是一个有 8 个单元的小型存储器，

地址从 0 ~ 7。每个单元分为三个字段：有效位（Valid）、标签（Tag）和数据（Data）。数据字段是高速缓存单元的一部分，存放来自内存的高速缓存行的副本。有效位字段是一位，如果高速缓存单元包含有效的来自内存的高速缓存行，那么该位为 1，否则为 0。

地址字段分为三个部分：标签（Tag）、行（Line）和字节（Byte）。字节字段是 4 位，对应于 24 = 16，所以高速缓存的每一行有 16 个字节。行字段是 3 位，对应于 23 = 8，所以高速缓存中有 8 个单元。标签字段包含 16 位地址剩下的位，所以它有 16-3-4 = 9 位。高速缓存单元包含来自地址的标签字段和来自内存的数据。在这个例子中，每个高速缓存单元占用一共 1 + 9 + 128 = 138 位。因为高速缓存中有 8 个单元，所以总共有 138×8 = 1104 位。

当系统启动后，会把高速缓存中的所有有效位设置为 0。第一次内存请求将不命中。来自内存的高速缓存行的地址被加载进地址标签字段，最后 4 位被设置为 0；从内存取出该行，并存在高速缓存单元中，设置有效位为 1；还会从地址中提取出标签字段并存储。

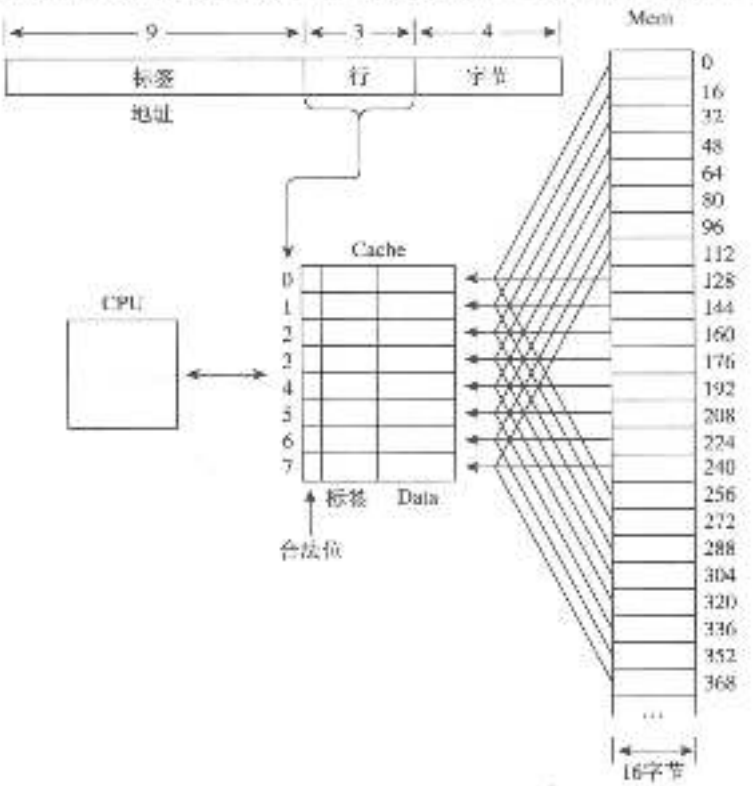


图 12-32　直接映射高速缓存

如果另一个请求要求同一行中的字节则会命中。系统从该地址提取出行字段，到高速缓存中的该行，确定有效位为 1，请求的标签字段与高速缓存单元的标签一致。系统从高速缓存单元的数据部分取出该字节或者字，并送到 CPU，不需要读内存。如果有效位为 1，但是标签字段不匹配，就是不命中，需要进行内存访问，新的标签和数据字段替代同一高速缓存单元中旧的标签和数据字段。

例 12.6　CPU 请求地址 3519（dec）的字节。标签字段的 9 位是什么？字节字段的 4 位是什么？数据会存放在高速缓存的哪个单元？转换为二进制并抽取出相应的字段得到

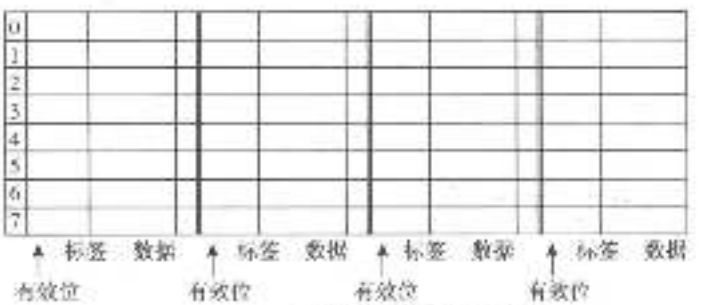
3519（dec）= 000011011 011 1111（bin）

636 / 638

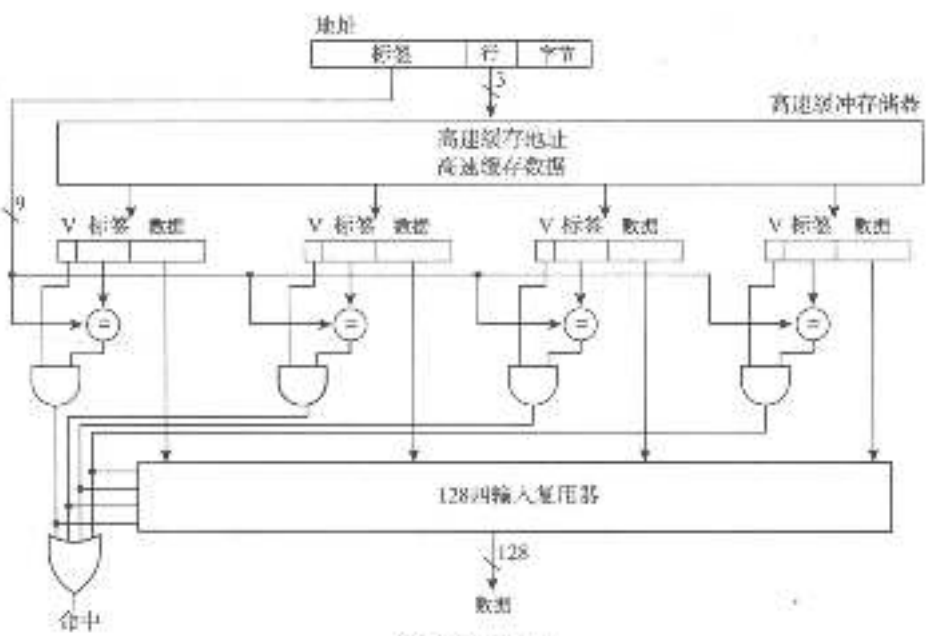
标签字段的 9 位是 000011011，字节字段的 4 位是 1111，而数据存储在高速缓存中的地址是 011（bin）= 3（dec）。□

图 12-32 给出内存在地址 16，144，272，…的块，它们都在竞争高速缓存中的同一个位置，即地址为 1 的条目。由于标签字段有 9 位，因此高速缓存中每个条目在内存中都有 $2^9 = 512$ 块竞争，而每个条目每次只能放下一块。有一种请求模式会导致很低的命中率，这个模式是在内存中两个固定的区域之间来回访问。一个例子是一个程序有几个指针，指向 Pep/8 内存高地址的运行时栈和内存低地址的堆。访问指针和它们指向的单元的程序就会具有上述访问模式。如果指针和它们指向的单元的地址有同样的行字段，命中率就会急剧降低。

组相联高速缓存用于减轻这个问题。并不是每个高速缓存条目都只能存放来自内存的一个高速缓存行，而是可以存放多个行。图 12-33a 描述的是一个四路组相联高速缓存，把图 12-32 中的高速缓存复制了四次，允许一组最多四个具有相同行字段的内存块同时存放在高速缓存中。这个访问电路比直接映射高速缓存的电路更复杂。对于每个读请求，硬件必须并行地检查高速缓存单元的四个部分，如果有的话，就返回行字段匹配的那一个。



a) 高速缓冲存储器的方块图



b) 读电路的实现

图 12-33　四路组相联高速缓存

图 12-33b 给出了读电路的细节。有等号的圆圈是比较器，如果输入相等，就输出 1，否则输出 0。这里的 128 复用器比通常的简单。四输入复用器通常有两个选择线需要译码，但是这个复用器的四条选择线已经译码了。标号为命中的输出在命中时为 1，此时，标号为数据的输出是来自高速缓存行的数据，该行的标签字段和请求的地址的标签字段相同。否则，命中输出为 0。

使用组相联高速缓存的另一个麻烦是当高速缓存不命中发生时，缓存单元的四个部分都是被占用的，需要做出决定。问题是，四个部分中的哪一个该被来自内存的新数据覆盖呢？一项技术是最近最少被使用（LRU）算法。在一个两路组相联高速缓存中，每个缓存单元只需要再增加一位，就可记录哪个单元最近最少被使用。但是在一个四路组相联高速缓存中，要记录最近最少被使用就复杂得多了。必须维护按照使用顺序排列的四个条目列表，而且每次高速缓存请求都要更新这个列表。四路高速缓存的一种近似 LRU 方法是使用三位，一位表明哪个组最近最少被使用，每个组内的位表明该组中哪个条目最近最少被使用。

无论高速缓存是直接映射还是组相联，系统设计者都必须决定如何处理有缓存情况下的内存写。缓存命中时，有两种可能性：

- 写通过（write through）：每个写请求会更新高速缓存和相应的内存块。
- 写回（write back）：写请求只更新高速缓存中的副本。对内存的写只有在该缓存行被替换时才发生。

图 12-34 描述了这两种可能性。写通过是比较简单的设计，当系统要写内存时，CPU 会继续处理。当高速缓存行需要被替换时，内存中的值保证已经是最近的更新了。这样做的问题是，当写请求爆发时，总线上的流量会很大。写回策略降低了总线流量，否则这些总线流量可能会影响其他想使用主系统总线的部件的性能。不过在任何给定的时刻，内存中不一定具有变量当前最新的值。同时，当要替换某个缓存行时，会有一定的延迟，因为要先更新内存，才能把新数据加载进高速缓存。通过设计可使缓存命中率很高，这样的事件不会经常发生。

关于高速缓存另一个要解决的问题是当缓存不命中时，该如何处理写请求。一种称为写分配的策略把块从内存带入高速缓存，可能会替换另一个缓存行，然后按照正常的缓存写策略更新这一行。若不采用写分配，就会绕过高速缓存发起一个内存写。这里的主要思想是 CPU 可以继续处理过程，内存写可以并发地完成。

图 12-35 给出的是使用和不使用写分配的高速缓存写策略。虽然每种缓存不命中时的写策略可以和任意一种缓存命中时的写策略组合，但是写分配通常和缓存命中时的写回一起使用，而写通过高速缓存通常不使用写分配，以保持设计简单。图 12-34 与图 12-35a 和图 12-35b 对应于大多数高速缓存的设计选择。

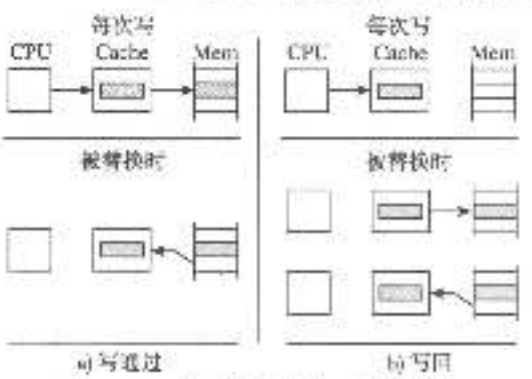


图 12-34　高速缓存命中时的写策略

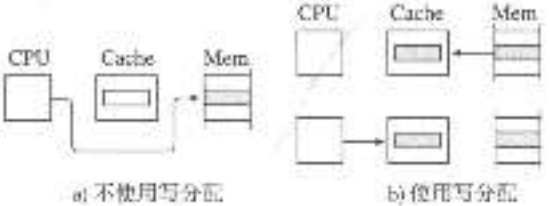


图 12-35　高速缓存不命中时的写策略

如果你读过 9.2 节中有关虚拟内存的讨论，那么有关高速缓冲存储器的讨论应该听起来很耳熟。虚拟内存背后的动机也是一样的，即较小的主存和存储在硬盘上的可执行程序的大小之间的不匹配，而高速缓存背后的动机是较小的高速缓存和主存大小之间的不匹配。虚拟内存中的 LRU 页替换策略对应于高速缓存中的 LRU 替换策略。高速缓存之于主存就像主存之于磁盘。在这两种情况中，都有一个跨越两层的内存层次结构（memory hierarchy），包括一个小但高速的存储子系统和一个大但慢度的存储子系统。两个层次结构中设计方案都依赖于引用局部性。两个领域中的设计有共同的问题和解决方法，这不是巧合。这些原理是普适的。另一个证明是软件中的哈希表数据结构。在图 12-32 中可以看出从主存到高速缓存的映射实际上是一个哈希函数。组相联高速缓存甚至看上去就很像哈希表，它通过链表解决冲突，只不过链表的长度有上限。

12.3.4　MIPS 的计算机组成

图 12-36 展示的是 MIPS CPU 的数据区，它的基本组成结构和图 12-2 的 Pep/8 的数据部分一致，ABus 和 BBus 送到主 ALU，ALU 的输出通过 CBus 最终到达 CPU 寄存器体。组成中最大的不同是通路中的 L1 指令和数据高速缓存。因为大多数高速缓存的命中率都在 90% 之上，所以我们可以假设内存读和写以 CPU 全速度在进行，没有 MemRead 和 MemWrite 延迟，除了偶尔发生缓存不命中的时候。

和 Pep/8 不同，程序计数器不是寄存器体中的通用寄存器之一。图中标号为 Plus4 的方框对 PC 加 4，通过两个复用器 PCMux 和 JMux 送回 PC。因为所有指令都是 4 字节长，所以这个冯·诺依曼周期的地址递加部分比 Pep/8 简单，可以用一个特殊的硬件单元实现，不需要跟主 ALU 绑在一起或占用周期。分支指令使用 PC 相对寻址，符号位扩展 16 位地址字段加上增加过的 PC 得到分支地址。图 12-36 Ex 部分中的 ASL2 方框对地址做移位，因为指令中的地址字段不存储最低两位。这些位永远为 0，因为地址要求在内存中四字节对齐，左移 2 位对应于图 12-26 中的乘以 4。

跳转指令使用伪直接寻址。图 12-36 IF 部分中的 ASL2 方块将地址左移 2 位，结果和增加过的 PC 的前 4 位（高 4 位）连接。这是用特殊目的电路硬件实现图 12-26 中的伪直接寻址方式。

CPU 不会写指令高速缓存，只会请求从 PC 指定的地址读。高速缓存了系统在命中时会从缓存中送出指令，不命中时会延迟 CPU，最终从 L2 高速缓存读到指令，写入 L1 高速缓存，并通知 CPU 可以继续了。因为 CPU 决不会写指令缓存，所以把缓存当作组合电路，这是指令缓存没有画阴影的原因。

标记为寄存器体的方块是一个双端口 32 位寄存器体，如图 12-24a 所示。标记为译码指令的方块是一个组合电路，输入为 32 位指令，它译码出操作码，发送适当的寄存器地址信号到 A、B 和 C，这些都是 5 位地址线，对图 12-24a 中的 32 个寄存器寻址。译码方块有很多连接复用器和 ALU 的控制线，图 12-36 中没有给出。

每个周期 PC 都会收到时钟信号并驱动主指示。寄存器体和数据高速缓存并不总是接受时钟，这取决于在执行的指令。对图 12-36 中三个时序电路有三个时钟：PC 的 PCCk，寄存器体的 LoadCk 和 L1 数据高速缓存的 DCCk。

无条件跳转和有条件分支指令唯一的目的是改变程序计数器。图 12-36 表明 PC 的所有计算都由特殊硬件单元完成，ASL2 把 PC 乘 4，Plus4 把 PC 加 4。下面的例子假设控制线的使用类似于图 12-2 中 Pep/8 控制线的使用，只不过图 12-36 中没有给出。

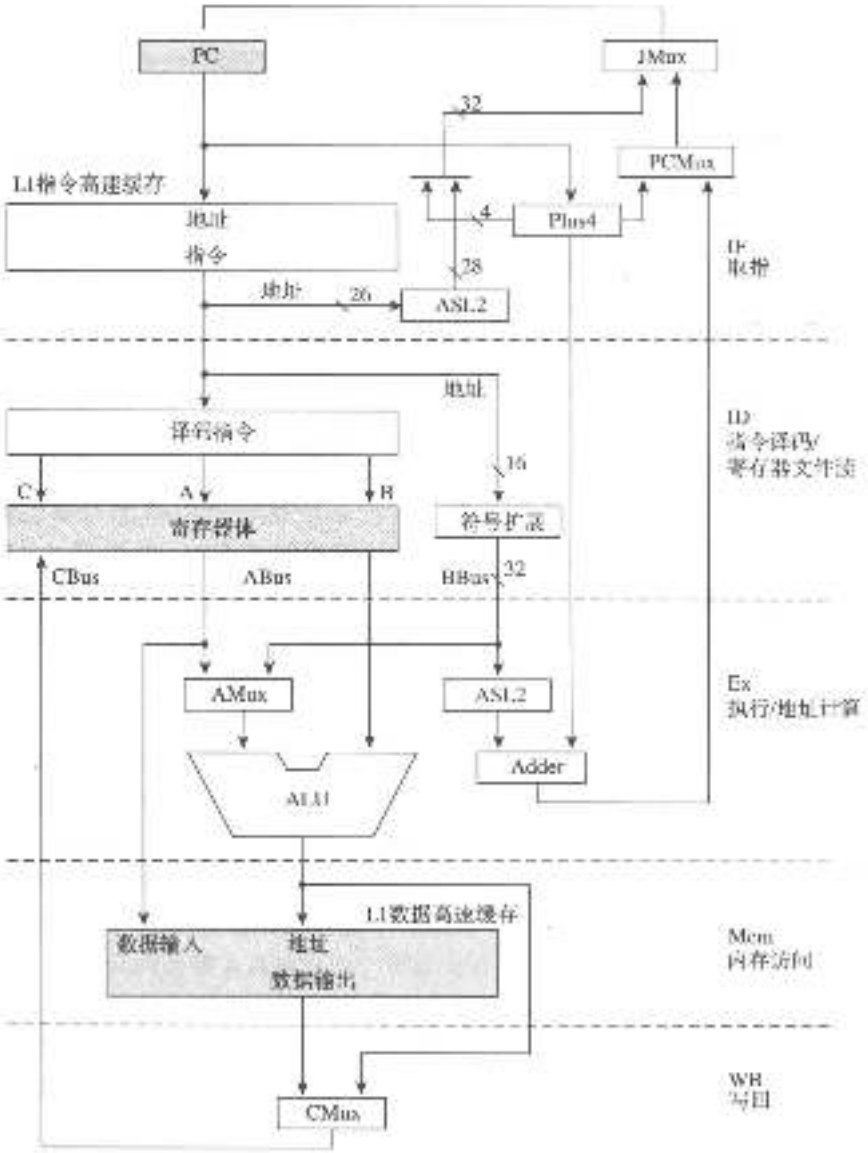


图 12-36　MIPS 数据区。时序电路用阴影表示

例 12.7　跳转指令使用伪直接寻址。假设 Mux 控制线按照惯例，0 选择左边的输入，1 选择右边的输入，那么跳转指令需要下面的控制信号：

```
. . JMux=0; PCCk
```

在周期开始执行前，PC 是跳转指令的地址。26 位地址字段送到 ASL2 的输入，增加过的 PC 的前 4 位和 ASL2 的输出连接，送到 JMux，JMux 的输出送到 PC。时钟脉冲到达时会更新 PC。□

639 / 644

例 12.8　条件分支指令使用 PC 相对寻址，需要下面的控制信号。

```
?. PCMux=1, JMux=1; PCCk
```

在周期开始执行前，PC 是条件分支指令的地址，16 位地址字段送到 ASL2 的输入，ASL2 输出和增加过的 PC 送到特殊目的加法器，加法器的输出送到 PCMux，PCMux 的输出送到

JMux，而 JMux 的输出送到 PC。时钟脉冲到达时会更新 PC。 □

数据区中组成部分的组织方式有助于存储指令。ABus 提供了一条通路，从寄存器体直接到 L1 数据高速缓存的数据输入。此外，主 ALU 的输出连接数据缓存的地址线。因此，存储指令的地址计算的加法是由主 ALU 而不是特殊目的硬件单元完成的。PC 更新和数据写回数据高速缓存同时进行。

例 12.9　字存储指令 sw 的 RTL 说明为

```
Mem [rb + address] ← rt
```

因为这条指令要更新 PC 并写内存，所以这个周期同时需要时钟脉冲 PCCk 和 DCCk。控制信号是

```
1, PCMux=0, JMux=1, A=rt, AMux=1, B=rb, ALU=A plus 3: PCCk, DCCk
```

PCMux=0 和 JMux=1 信号把增加过的 PC 送到 PC。A=rt 信号把 rt 源寄存器的内容送到 ABus，作为要送到高速缓存的数据。AMux=1 信号选择把指令的地址字段作为 ALU 的左输入，B = rb 信号把基址寄存器送到 BBus，作为 ALU 的右输入。选择加法功能，地址计算的结果送到数据高速缓存的地址线。 □

寄存器指令用主 ALU 完成处理，但是不写内存。因此，ALU 的输出有一条通过 CMux 到寄存器体的通路。和存储指令一样，PC 和寄存器体的更新是同时发生的。

例 12.10　加法指令 add 的 RTL 说明为

```
rd ← rs + rt
```

因为它要更新 PC 和写寄存器体，所以这个周期同时需要时钟脉冲 PCCk 和 LoadCk。控制信号是

```
1, PCMux=1, JMux=1, A=rs, AMux=0, B=rt, ALU=A plus 0, CMux=0, C=rd:
PCCk, LoadCk
```

645

PCMux=0 和 JMux=1 信号把增加过的 PC 送到 PC。A=rs 信号把 rs 源寄存器的内容放到 ABus，AMux=0 信号会把它当作数据通过 AMux 送到高速缓存。B=rt 信号把基址寄存器送到 BBus，作为 ALU 的右输入。选择加法功能，用 CMux=1 信号把结果通过 CMux 送到 CBus。信号 C=rd 对寄存器体寻址，地址为目标寄存器 rd。 □

装入指令的控制信号作为章末练习。

12.3.5　流水线

PC 在周期开始时变化，数据必须按照下列顺序通过组合电路：

- IF：指令高速缓存，Plus4 加法器，移位器和复用器。
- ID：译码指令方框，寄存器体，符号扩展方框。
- Ex：AMux，ASL2 移位器，ALU，加法器。
- Mem：数据高速缓存。
- WB：CMux，寄存器体的地址译码器。

CPU 设计者必须把时钟周期设置得足够长，使得数据（PC，寄存器体和数据高速缓存）送到时序电路，然后时钟会把数据写入时序电路。图 12-37 给出了几条指令逐条执行的时间线。方框代表每个阶段的传递时间。

图 12-37 中的情况类似于一个要制造家具的工匠。他的作坊有所有工具来做三件事：切割木头、装配和上漆。因为只有一位工匠，所以他会按照这样的顺序来制造家具，然后一件



图 12-37　不使用流水线的指令执行

一件玩物。如果再有两位工匠，便有几种方法来增加每天能完成的家具数量。其他工匠如果有自己的工具，那么三个工匠可以并行工作。同时为三件家具切割木头、装配和上漆。单位时间的输出确实会是原来的三倍，但是额外的工具也是一笔开销。

一种更经济的替代方法是确认当某人正在用螺丝和胶水进行装配时，切割木头的工具可以用于制造下一件家具。类似地，当第一件家具上漆时，第二件在组装中，第三件则正在切割中。应该可以看出，这种组织结构是工厂装配线的基本架构。

对应于工具的资源是上面列出的五个方面的组合电路：取指，指令译码/寄存器文件读，执行/地址计算，访存和写回。CPU 流水线的理念是把执行一条指令花费的周期数增加为原来的 5 倍，但是把每个周期的时间降低为原来的 1/5。初看上去这样做没有什么好处，但是把指令和指令的执行重叠起来，就能获得并行性，增加每秒执行的指令数。

要实现这个想法需要对图 12-36 的数据通路做一些修改。每个阶段的结果必须保存，以作为下一阶段的输入。在这五个部分之间的边界要放一组寄存器，如图 12-38 所示。在每个新的缩短了的周期结束前，每条要送到下个阶段的数据通路的数据都要存放在边界寄存器中。

只有当每个阶段的传播时间完全相等时，周期时间才能刚好降低为原来的 1/5。实际上不太可能这样，所以新周期时间是所有缩短过的阶段中延迟最长的阶段的传播时间。实现流水线要选择把边界寄存器放在哪里，设计者必须设法均匀划分这些阶段。

图 12-39 展示了流水线是如何工作的。启动时流水线是空的。在周期 1，第一条指令取指。在周期 2，第二条指令取指，同时第一条指令译码和读寄存器体。在周期 3，第三条指令取指，同时第二条指令译码和读寄存器体，同时第一条指令执行，以此类推。这样做能带来加速是因为这使得更多电路部件可同时使用。流水线就是一种形式的并行。理论上，对于一个完美的有五个阶段的流水线，当流水线充满时，每秒执行的指令数能提高 500%。

这是好消息，不过也有坏消息，有些问题会破坏这个看上去很美好的画面。这样的问题分为两类，称为危害（hazard）：

- 控制危害，来自无条件和条件分支
- 数据危害，来自指令间的数据依赖

这两种危害都是由于有指令不能在流水线的某个阶段执行完任务，因为它需要前面尚未执行完毕的指令的结果。危害会导致这条指令不能继续，只能停顿，这在流水线中造成了一个气泡，必须将其清除，然后才可能达到峰值性能。

图 12-40a 展示的是没有危害时流水线从起始时的执行。第一行第一组五个方块表示要执行的第 6 条指令，第二行的第二组表示第 7 条指令，以此类推。从第 5 个周期开始，这个流水线就开始以峰值性能运行了。

考虑分支指令执行时会发生什么。假设指令 7 是一条分支指令，它从周期 7 第二行开始。假设更新过的程序计数器要到这条指令完成时才能为第二条指令所用，那么指令 8 及其

后面的指令都必须停顿。图 12-40b 中气泡是未加阴影的。结果看上去就像流水线必须在周期 12 重新开始。图 12-41 显示分支指令占到 MIPS 机器上典型程序中 15% 的执行语句。所以大概每 7 条指令就要延迟四个周期。

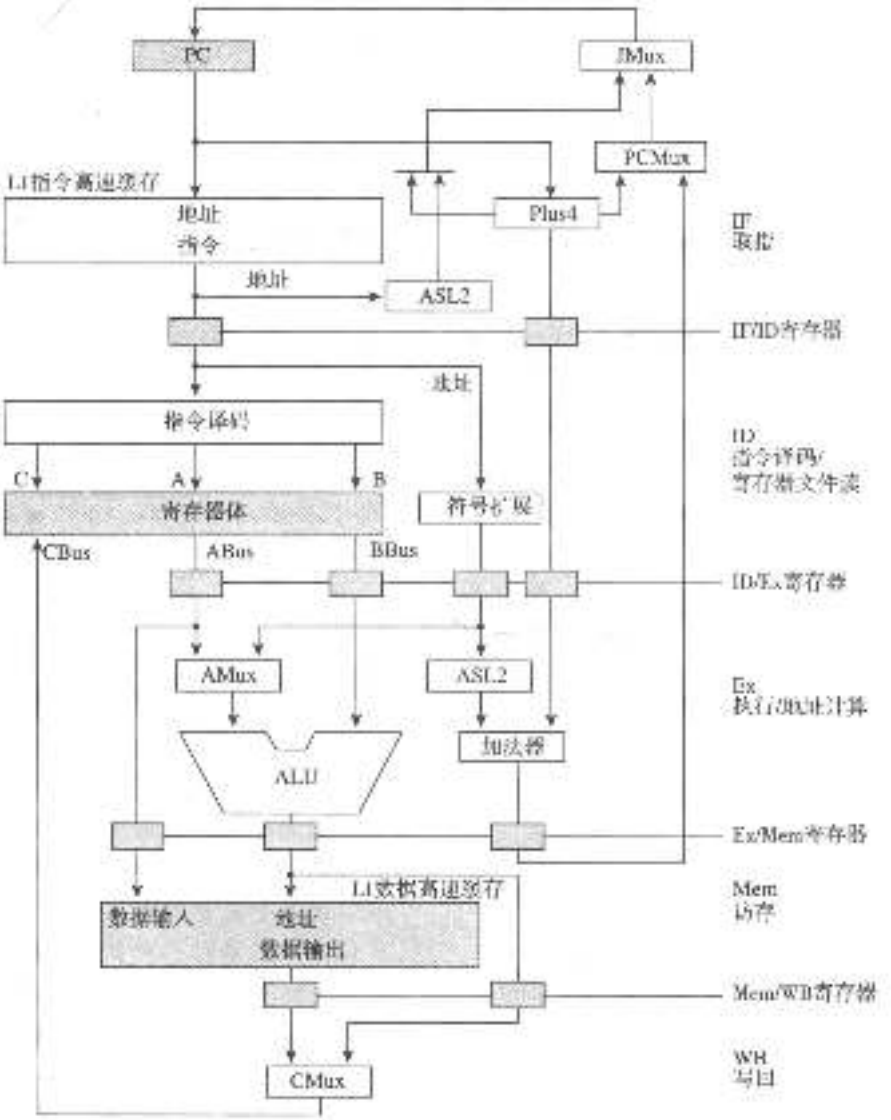


图 12-38　使用流水线的 MIPS 数据区

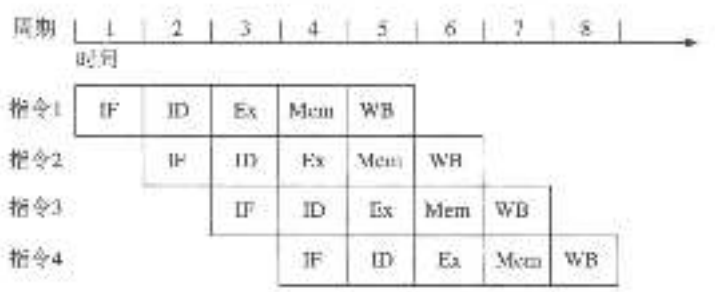
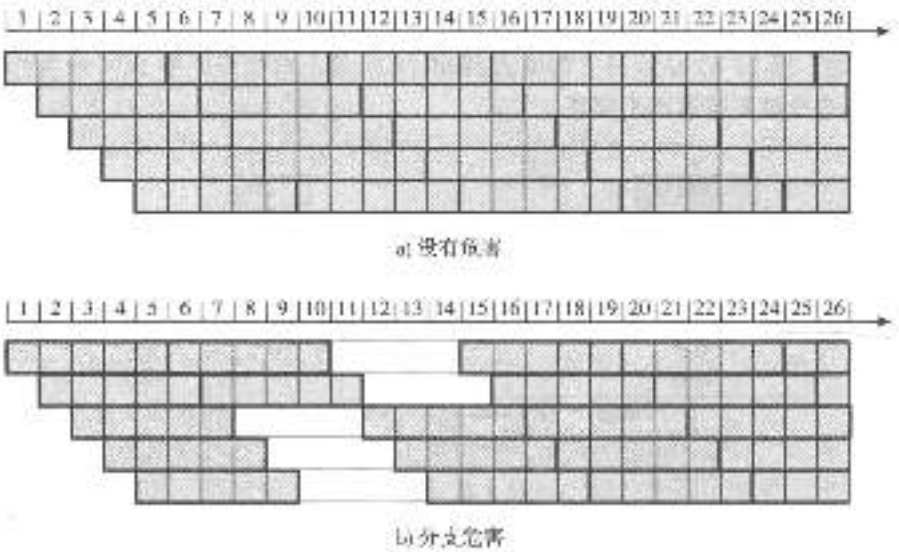


图 12-39　使用流水线时的指令执行



a) 没有危害

b) 分支危害

图 12-40　危害对流水线的影响

有几种方法可降低控制危害带来的性能处罚。图 12-40b 假设分支指令的结果要到写回阶段之后才能得到，但是实际上分支指令不改变寄存器体。所以，假设下一条指令被延迟了，要降低气泡的长度，系统就必须消除分支指令的写回阶段。增加额外的控制硬件会把气泡的长度从四个周期减少到二个。

指　令	频　率
算术	50%
加载 / 存储	35%
分支	15%

表 12-41　MIPS 指令的执行频率

对条件分支另外还有一个机会能降低危害的影响。假设增加额外的控制硬件，分支的处罚是 3 个周期，而计算机执行下面这个 MIPS 程序：

```
beq $s1,$s2,4
add $s3,$s3,$s4
sub $s5,$s6,$s6
andi $s7,$s7,15
slt $s0,$s0,2
ori $s3,$s3,1
```

第一条指令是如果相等分支，地址字段是 4，这意味着这个分支是到下一条指令后面四个的那条指令。所以，如果选择了该分支，就会是 ori 指令。如果没有选择分支，接下来会执行 add。

从图 12-40b 中可以看到浪费了很多并行性。随着气泡被冲洗出流水线，很多阶段都是空闲的。在等待 beq 的结果时，不知道是否该执行 add、sub 和 andi。不过还是可以假设不会选择分支而执行它们。如果实际上没有选择分支，那么就刚好消除了气泡；如果选择了分支，从气泡延迟的角度来说，也不会比不执行 beq 后面的指令更差，在那种情况下，是把气泡冲出流水线。

这样做的问题是需要电路清理如果假设错误选择了分支之后留下的坏影响。必须记录好中间的所有指令，在发现分支是否跳转之前，不允许它们永久地修改数据高速缓存或寄存器体。当发现不选择分支时，就可以提交这些改变了。

假设分支不会跳转是一种原始的预测未来的方法，这就好像去看赛马时总是把赌注压在同一匹马上而不管它之前的战绩如何。可以让历史指引选择，这种技术称为动态分支预测

(dynamic branch prediction)。在分支语句执行时，同时记录这个分支是跳转还是不跳转，如果跳转，则记录它的目的地址。下次这条指令执行时，就预测会发生同样的结果。如果上一次跳转了，那就用分支目的处的指令填充流水线，否则就用分支指令后面的指令填充流水线。

上述方法称为一位分支预测，因为只需要一位就能记录某个分支是跳转还是不跳转。一位存储单元定义了一个有两个状态的有限状态机，两个状态分别对应于预测分支是否会跳转。图 12-42 给出的就是这个有限状态机。



图 12-42　一位动态分支预测的状态转移图

还是用赛马的比喻，可能你不应该这么快改变要把赌注放在哪匹马上。假设有一匹马你已经连续押注二次并且都赢了，第四次，另一匹马赢了。你真的想要只根据这一次结果而不管以前的历史，就把赌注压到另一匹马上？这类似于程序中有嵌套的循环时发生的情况。假设内循环每执行四次，外循环执行一次。编译器用条件分支翻译内循环的代码，这个分支会连续跳转四次，然后会不跳转一次，终止这次循环。下面是分支选择序列和基于图 12-42 的一位动态预测结果。

分支选择：	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N
预测结果：	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y
预测错误：	x				x	x				x	x				x	x				x

采用一位动态分支预测，对于每次外循环，内循环的分支总是会被预测错误两次。

要克服这个不足，常见的做法是采用两位来预测下一个分支。其思想是如果分支有多次跳转，然后遇到一次不跳转，那么不用立即改变预测；改变的条件是有两次连续的不跳转。图 12-43 显示共有四种状态。两个带阴影的状态是连续有两个一样的分支类型时的状态：前面两次分支都跳转或者都不跳转。两个没有阴影的状态表示前面两次分支结果不同。如果是连续的分支跳转，就处于状态 00。如果下一个分支是不跳转，则进入状态 01，但是仍然预测此后的分支会跳转。图 12-43 的有限状态机按照上述分支序列执行的结果表明，从外层循环开始每次预测都是正确的。

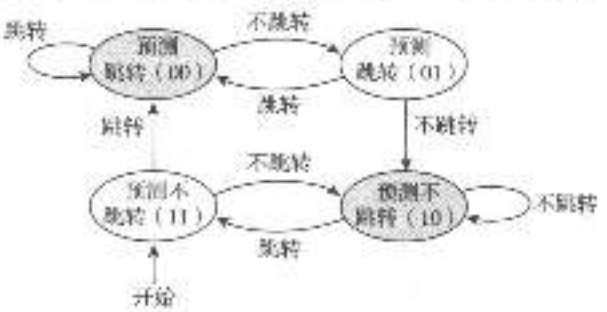


图 12-43　两位动态分支预测的状态转移图

另一项技术用于深度流水线，在深度流水线上如果分支预测错误，处罚会更严重。这项技术是复制流水线，有两份程序计数器，取指电路和所有其余部分。译码一条分支指令时，启动这两条流水线，其中一条装入假设分支不跳转的指令，另一条装入假设分支跳转的指令。如果发现哪条流水线是正确的，就丢掉另一条，继续执行正确的。这个解决方案很昂贵，但不管分支是跳转还是不跳转，都没有气泡。

当一条指令需要前面指令的结果时，就必须停顿直到得到该结果，此时数据危害就发生了。这种是写后读（Read-After-Write，RAW）危害，下面的代码序列就是这样一个例子：

```
add $s2,$s2,$s3  # write $s2
sub $s4,$s4,$s2  # read $s2
```

add 指令改变 $s2 的值，这个值用在 sub 指令。图 12-38 表明 add 指令会在写回阶段 WB 结束前更新 $s2 的值。然后，sub 指令会在它的指令译码 / 寄存器文件读阶段 ID 结束前从寄存器体读出。图 12-44b 给出了有数据危害时这两条指令该如何重叠。sub 指令的 ID 阶段必须在 add 指令的 WB 阶段之后。结果是 sub 指令必须停顿，产生三个周期的气泡。



a) 没有RAW危害的连续指令



b) 有RAW危害的连续指令

图 12-44　RAW 数据危害对流水线的影响

如果在 add 和 sub 之间有另外一条指令没有危害，就只有两个周期的气泡。如果它们之间有两条没有危害的指令，气泡就会减少到一个周期，而如果有三条，就没有气泡了。观察到这些就有一种可能的方法。如果能够在附近找到一些没有危害而且无论分支结果如何都会执行的指令，为什么不乱序执行它们，把它们插入到 add 和 sub 指令之间填充气泡呢？反对这样做的理由是改变指令执行的顺序可能会改变算法的结果。在有些情况中是这样，但是也不全是。如果一个代码块中有很多算术运算，则优化编译器会分析数据依赖，重新调整语句，减少流水线中气泡的数量，而又不改变算法的结果。同样，一个汇编语言程序员也可以做同样的事情。

这是一个抽象付出代价的例子。一层抽象本意是通过隐藏低层的细节简化某一层上的计算。如果汇编语言程序员或编译器设计者不了解 LG1 层流水线的细节，就能生成对 ISA3 层最方便的语句顺序，事情当然更容易。增加一层抽象总是会带来性能损失。问题是简化带来的好处能不能弥补性能损失。使用 LG1 层的细节是为了平衡性能的简化抽象。这样平衡的另一个例子是在设计 ISA3 程序时考虑高速缓存系统的属性。

另一项称为数据传送（data forwarding）的技术能减轻数据危害。图 12-38 表明来自 ALU 的加法指令的结果会在 Ex 阶段结束时随时钟进入一个 Ex/Mem 边界寄存器。对于加法指令，只需在 Mem 阶段结束时把结果送入 Mem/WB 边界寄存器，最终在 WB 阶段结束时进入寄存器体。如果在包含 add 结果的 Ex/Mem 寄存器和 sub 通常会取出结果的寄存器体之间建立一条通路，那么对图 12-44b 唯一要做的调整就是 sub 的 ID 阶段在 add 的 Ex 阶段之后。这样做还是需要一个气泡，但是只有一个周期。

超标量（superscalar）设计利用的是如果两条指令没有数据依赖，那么它们就能并行执行。表 12-45 给出了两种方法。图 12-45a 表明可以建立两条独立的流水线，它有一个速度足够快的取指单元，能够一个周期取出多条指令，还能够每个周期并发地发射两条指令。这样调度比较复杂，因为要管理跨两条流水线的数据依赖。

图 12-45b 是基于这样一个事实，执行单元 Ex 通常是执行链中最薄弱的一环，因为它的传播延迟比流水线中其他阶段要长。和整数处理电路相比，浮点单元尤其耗时。图中标号为 FP 的方框是一个浮点单元，实现的是 IEEE 754。每个执行单元可能比流水线中其他阶段慢三倍。但是，如果它们能并行工作，执行阶段就不会拖慢整个流水线。

超标量机器中，指令调度器必须考虑其他类型的数据危害。当一条指令在前面一条指令读一个寄存器之后写这个寄存器，就会发生读后写（Write-After-Read，WAR）危害。下面是一个 MIPS 代码的例子。

```
add $s3,$s2,$s2  # read $s2
sub $s2,$s4,$s5  # write $s2
```

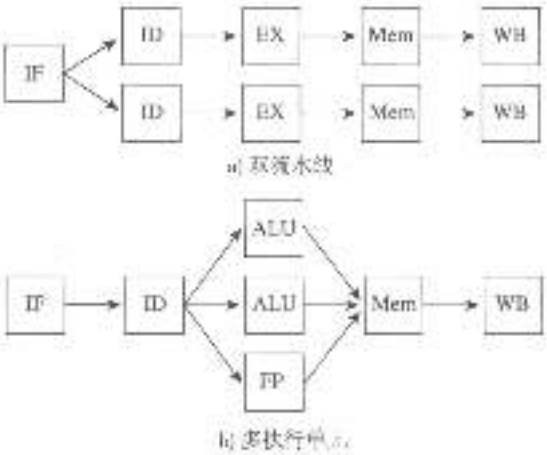


图 12-45　超标量机器

在图 12-38 的流水线化的机器中，这个序列没有危害，因为 add 在它的 ID 阶段结束时把 $s2 随时钟送入 ID/Ex 寄存器中，而 sub 会在它的 WB 阶段结束前写这个寄存器。超标量机器会为了减少气泡调整这些指令的顺序，所以它可能先启动 sub 指令，然后再启动 add。

在完美的流水线中，使用 k 阶段流水线会把时钟频率增加为原来的 k 倍，性能也提高为原来的 k 倍。那为什么不能把这种设计发挥到极致，让一个周期等于一个门延迟呢？因为由控制危害和数据危害引起的复杂性会降低性能，无法得到完美的结果。总会到一个点，增加流水线的长度和增加频率会降低性能。

但是提升时钟频率还有一个好处：广告。个人电脑的时钟频率是消费者决定购买哪一种电脑时的重要因素。时钟频率神话是说两台具有不同时钟频率值的机器，时钟频率更高的机器性能也更高。你现在应该能明白为什么这个神话不是真的了。增加流水线中阶段的数量会提高频率，但是也会增加危害带来的性能处罚。问题是设计者能有效战胜这些处罚。使用时钟频率作为衡量性能的手段还忽略了图 12-21 中因素之间的相互作用。终极问题不仅是 CPU 每秒包含多少周期，还有每个周期能做多少有效工作来执行程序。

646
654

计算历史发展到的现在，流水线技术已经到达一个平台期。虽然商用 CPU 芯片的时钟速度还在提升，但是提升速度已经大不如前。摩尔定律依然有效，数字电路工程师能够在芯片的每平方毫米提供更多的门，但是 CPU 设计的当前趋势是使用额外的电路简单地复制整个 CPU，在一个封装中装进 2 ~ 4 个 CPU，称为核（core）。这种趋势未来还会加剧。这意味着软件设计者要面临更大的挑战，要使用像 8.3 节介绍过的那些并行编程技术来利用多核 CPU 芯片提供的能力。

12.4　结论

Pep/8 说明的是冯·诺依曼计算机的基本本质。数据区由组合电路和时序电路组成，是一个大的有限状态机。数据区的输入包括来自主存的输入和来自控制区的控制信号，数据区的输出也送往主存和控制区。每当控制区向状态寄存器发送时钟脉冲时，状态机就转移到一个不同的状态。

在真实的计算机中，状态的数量非常大，但还是有限的。图 12-2 中的 Pep/8 数据区有

24个可写的8位寄存器和4个状态位，对于总共192位存储，有2^{192}个状态。有8个输入来自主系统总线的数据线，32个控制输入来自控制区，每个状态的转移数是2^{40}。转移的总数是2^{192}乘以2^{40}，即$2^{232}=10^{70}$。地球上原子的个数估计只有10^{50}，而Pep/8只是一台很小的计算机！在最基本的层面上，无论系统多么复杂，计算都不过是执行一个有外围存储的有限状态机。

12.4.1　模型简化

Pep/8计算机说明了真实的冯·诺依曼机器背后的基本组织思想。当然，还是进行了很
655 多简化以使机器简单易懂。

有一个低层细节与实际硬件实现中有所不同，那就是在整个集成电路中使用的是边沿触发的触发器，而不是主－从触发器。这两种触发器都能解决回馈问题。因为主－从原理比边沿触发原理更容易理解，所以描述中全都使用的是主－从触发器。

另一个简化是使用主系统总线的CPU和主存之间的接口。在真实的计算机中，时间约束要更复杂，不是简单地把地址放到总线上，使MemRead保持有效且等待两个周期，并假设数据就能够由时钟送进MDR中。一次内存访问要求不止两个周期，相关协议更详细地说明地址线必须保持有效时长，以及数据必须随时钟送入CPU寄存器中的确切时间。

有关主系统总线的另一个问题是它是如何在CPU、内存和外围设备之间共享。实际中，CPU并不总是控制着总线。相反，如果同时有多个设备想要使用总线，总线使用自己的处理器在竞争的设备之间仲裁。一个例子是使用直接内存访问（Direct Memory Access，DMA），数据从磁盘直接通过总线送到主存，不受CPU控制。DMA的优点是CPU能够把它的周期用在执行程序的有用工作上，而不必分心控制外围设备。

其他一些超出本书讨论范围的议题包括汇编器宏、链接器、流行的外围总线（比如USB和Firewire）、超级计算机以及整个计算机网络领域。学习计算机网络时，你会发现抽象是核心。计算机系统被设计为抽象层级，每一层的细节都向上一层隐藏，因特网通信协议也是这样设计的。每一抽象层的存在都只做一件事情，向更高的一层提供服务，而隐藏如何提供服务的细节。

12.4.2　更大的景象

现在考虑一幅更大的景象，从App7层一直向下到LG1层。假设用户从App7的一个应用程序向数据库系统输入数据。她想录入一个数值，所以敲击这个值，并执行回车命令。这样一个看似无害的举动后面发生了什么呢？

C++程序员编写的数据库系统包括输入数值的过程。这个C++程序编译成汇编语言，然后又被汇编成机器语言。编译器设计者编写编译器，而汇编器设计者编写汇编器。编译器和汇编器是自动翻译器，都包含词法分析阶段、语法分析阶和代码生成阶段，其中词法分析阶段是基于优先状态机的。

在数值输入过程中，C++程序员也使用有限状态机。编译器把过程中每条C++语句翻
656 译成多条汇编语言语句。不过汇编器会把每条汇编语言语句翻译成一条机器语言语句。所以处理用户回车命令的代码会扩展成多条C++命令，而每条C++命令又会扩展成多条ISA3层命令。

然后，每个ISA3层命令会被翻译成控制区信号来获取和执行指令。每个控制信号输入

一个复用器或某个其他组合设备，或者这个信号是一个脉冲，让一个值随着时钟进入状态寄存器。时序电路也是由有限状态机的规则管理的。

每个寄存器都是一个触发器阵列，每个触发器都是一对以主－从原理设计的锁存器。每个锁存器都是一对 NOR 门以及简单的交叉耦合反馈连接。数据区的每个组合部件都是很少几种类型的门的互联。每个门的行为都受布尔代数定律的控制。最终，用户的回车命令被翻译成电子信号，在各个门中流动。

如果在多道程序设计系统中执行，用户的输入命令可能会被操作系统中断。回车命令可能会产生缺页，这种情况下操作系统可能需要执行页替换算法，确定要把哪一页拷贝回磁盘。

当然，这些事件都是在用户对系统低层毫不知情的情况下发生的。任何一层的设计缺陷都会减缓处理，用户会感知到并且抱怨计算机。记住，从 App7 到 LG1，整个系统设计都受到基本的空间 / 时间折中的制约。

LG1 层上通过某个复用器的一个门的信号和 App7 层上执行回车命令的用户之间的关联看起来很遥远，但是它真实存在。实际上有上百万的门在协同工作来执行用户的任务。这么多设备能够被组织成一台有用的机器正是得益于把系统构造成不同的抽象层级。

每个抽象层级都只有少量的简单概念，这真的很惊人。在 LG1 层，只用 NAND 或者 NOR 门就足以构造任何组合电路。只有四种基本的触发器类型，它们都可以用 SR 触发器实现。简单的冯·诺依曼周期是机器运行背后 ISA3 层的控制力量。在 OS4 层，进程是一个运行着的程序，可以通过存储它的进程控制块来中断它。Asmb5 层的汇编语言是到机器语言的一对一简单翻译。HOL6 层的高级语言是到低级语言的一对多翻译。

有限状态机的概念在整个层次结构中随处可见。有限状态机是自动翻译器词法分析的基础，也用来描述时序电路。进程控制块存储着进程的状态。

所有科学都以简单和结构化作为目标。在自然科学中，人们致力于发现自然法则，用最少的数学定律或概念来解释大多数现象。计算机科学家也发现简单是控制复杂的关键。能建造出像计算机这样复杂的机器，完全是因为在每个抽象层次只需要简单的概念来控制它的行为。 657

总结

中央处理器单元分为数据区和控制区。数据区有一个寄存器体，部分或全部对 ISA3 层程序员可见。处理是一个循环，来自寄存器体的数据通过 ABus 和 Bbus，通过 ALU，再通过 CBus 返回寄存器体。内存地址寄存器指定地址处的数据通过主系统总线和内存数据寄存器从内存注入循环中。

控制区的功能是向数据区发送控制序列，实现 ISA3 指令集。机器由冯·诺依曼周期控制：取指、译码、增加、执行和重复。在像 Pep/8 这样的 CISC 机器中，控制信号必须引导数据区取操作数，由于寻址方式复杂，这可能要花费很多周期。像 MIPS 这样的 RISC 机器则寻址方式较少，指令比较简单，使得每条指令的执行都只需要一个周期。

提高性能可能来自于两个方面：基本的空间 / 时间折中和并行性。使用空间 / 时间折中的两种一般方法是分配额外的硬件（空间）提高数据总线的宽度，或者部署特殊的硬件单元。这两种都会导致执行时间降低。

所有性能提升都基于下面这个等式表明的执行时间的 3 个组成部分：

$$\frac{\text{时间}}{\text{程序}}=\frac{\text{指令}}{\text{程序}}\times\frac{\text{周期}}{\text{指令}}\times\frac{\text{时间}}{\text{周期}}$$

累加器机器会降低第一个因素，代价是提高第二个。装入/存储机器会降低第二个因素，代价是提高第一个。这两种组织方法都会用第三个因素来提高性能，主要是通过流水线化。

具有复杂指令和较多寻址方式的计算机在计算历史早期比较流行。它们的特征之一就是 Mc2 层抽象，这层抽象中控制区有自己的微内存、微程序计数器和微指令寄存器。控制区的微程序产生实现 ISA3 指令集的控制序列。装入/存储计算机的特性是没有 Mc2 层抽象，因为它的每条简单指令都能在一个周期内实现。

高速缓冲存储器解决的是 CPU 的快速和主存的慢速之间极端不匹配的问题，它依赖所有真实程序中都会表现出的引用的空间和时间局部性。高速缓存是一个小的高速内存单元，包含一部分很可能被 CPU 访问的主存数据。

流水线类似于工厂里的装配线。要实现流水线就要划分周期，在数据区的数据通路上插
658 入边界寄存器。效果是消加了每条指令的周期数，但是相应地降低了时钟周期。当流水线充满的时候，借助于流水线固有的并行性，能够实现每个周期执行一条指令。不过，控制危害和数据危害会降低性能，不能达到理论上的理想值。

练习

12.1 节

1. 画出 MDR 和主存总线之间的全部 8 位总线，给出三态缓冲器和到 MemWrite 线的连接。
2. 设计图 12-2 的三输入单输出组合电路 ANDZ，用卡诺图简化电路，既考虑用 AND-OR 来实现，又考虑用 OR-AND 来实现，选择较好的一种。
3. 书中提到可以把图 12-5 中的周期 1 和周期 3 合并，加速冯·诺依曼周期。可以把周期 1 和周期 2 合并么？解释之。
4. 书中提到可以把图 12-5 的周期数从 7 降到 4，写出 4 周期时取指令指示符和 PC 加 1 的控制信号。
5. 写出取操作数指示符和 PC 加 1 相应的冯·诺依曼周期控制序列。假设已经取出了指令指示符，并且控制区已经知道这条指令是非一元指令了。
6. 书中提到可以把图 12-10 中的周期数从 21 降到 17，写出 17 周期时实现采用间接寻址的 LDX 指令的控制信号。
7. 对于下面每条 ISA 指令，（1）写出它的 RTL 描述，（2）写出实现该指令的控制序列。假设指令已经取出，如果是非一元指令就包括操作数指示符，还包括保存状态位副本的寄存器 T1。

```
*(a) STBYTEA there,n          (b) STBYTCA there,i
 (c) STBYTEA there,sI         (d) STBYTEA there,x
 (e) STBYTEA there,sx         (f) STBYTEA there,sxf
 (g) BR there                 (h) CALL there
 (i) NCTA                     (j) NEGA
 (k) RCLA                     (l) RORA
 (m) RCTA                     (n) ADDSP this,1
 (o) SUBSP this,1             (p) SUBA this,1
 (q) ANDA this,I              (r) ORA this,i
 (s) CPA this,1               (t) LDBYTCA this,I
 (u) LDBYTEA this,d           (v) MOVSPA
 (w) MOVFLGA                  (x) RETTR
```

8. 写出执行指令 DECO num,i 的控制序列，假设指令已经取出。记住，这条指令的操作码未被实现，
659 你可能想要复习一下 8.2 节中的陷阱部分。

12.2 节

9. * 书中预测不需要从 64 位计算机转变到 128 位计算机，因为我们不会需要大于 160 亿 GB 的主存。

硅晶体是一个由 0.5 nm 的方瓦片组成的平面，每个瓦片由两个原子组成。(a) 假设可以制造一个内存，密度高到硅原子平面上每个原子存储 1 位（忽略线的互联问题）。要存储 64 位机器可以寻址的最大的字节数，正方形芯片的边长应该是多少？给出计算过程。(b) 这个计算能支撑前面的预测吗？解释之。

10. 设计图 12-17 的 33 输入 16 输出地址增量器，画出用半加器实现这个增加的行波进位加法的逻辑图。可以使用省略号 (...)。
11. 假设使用图 12-17 中的地址增量器，写出执行练习 7 中每条语句的控制序列。
12. 假设已经完成了练习 7 和 11，对于每条语句，计算使用这个地址增量器引起的节省的周期数百分比。
13. 假设使用图 12-19 中的 2 字节 MDR，写出执行练习 7 中每条语句的控制序列。假设所有地址和字操作数都在偶地址，所有字节操作数都在奇地址。
14. 假设已经完成了练习 7 和 13，对于每条语句，计算使用这个 2 字节 MDR 引起的节省的周期数百分比。
15. ASLA 指令对累加器做算术左移，并把结果放回累加器中。ASLA 的 RTL 说明表示它会设置 V 位，当数字被解释为有符号数时设置这个位表示有溢出。这与 ASRA 不同。如果原始值的符号和移位后的值的符号不同，就发生了溢出。移位之后，C 位包含原始值的符号，而 N 位包含移位后的值的符号。所以，可以用 V = C ⊕ N 计算 V 位。幸运的是 Pep/8 ALU 中有 XOR 功能。(a) 写出实现 ASLA 的控制语句序列。需要获得状态位的副本并且移位，这样 C 和 N 的值都会在正确的位置上，可以用来计算 V；然后，把两个值 XOR 并存储 Z 位。使用尽可能少的周期数。(b) 用正确的硬件可以在一个周期内算出 ASLA 指令的 V，设计一个特殊的硬件单元来实现。(c) 写出用这个新电路实现 ASLA 的控制序列。移位指令本身需要两个周期，设置 V 又需要一个周期。(d) 这个新设计带来的周期数降低百分比是多少？(e) 讨论这个硬件单元对 CPU 其余设计的影响。(f) 如果这个硬件只能加速指令集中一条指令的 V 位计算，你还认为这样的硬件修改值得么？解释之。

*16. (a) 假设 MIPS 机器的 C++ 编译器把 $s4 和数组 a、$s5 和变量 g、$s6 和数组 b 联系在一起。它会把语句

```
a[4]=g+b[5];
```

翻译成什么 MIPS 汇编语言呢？(b) 写出 (a) 中指令的机器语言翻译。 660

17. (a) 写出 MIPS 汇编语言语句，把寄存器 $s2 的内容左移 9 位，并把结果放到 $t5 中。(b) 写出 (a) 中指令的机器语言翻译。
18. (a) 假设 MIPS 机器的 C++ 编译器把 $s4 和变量 g、$s5 和数组 a、$s6 和变量 1 联系在一起。该如何把语句 g = a[1] 翻译成 MIPS 汇编语言呢？(b) 写出 (a) 中指令的机器语言翻译。
19. (a) 假设 MIPS 机器的 C++ 编译器把 $s4 和变量 g、$s5 和数组 a、$s6 和变量 1 联系在一起。该如何把语句 a[1] = g 翻译成 MIPS 汇编语言呢？(b) 写出 (a) 中指令的机器语言翻译。
20. (a) 假设 MIPS 机器的 C++ 编译器把 $s4 和变量 g、$s5 和数组 a、$s6 和变量 1 联系在一起。该如何把语句 g = a[1+3] 翻译成 MIPS 汇编语言呢？(b) 写出 (a) 中指令的机器语言翻译。
21. (a) 假设 MIPS 机器的 C++ 编译器把 $s5 和数组 a、$s6 和变量 1 联系在一起。该如何把语句 a[1] = a[1+1] 翻译成 MIPS 汇编语言呢？(b) 写出 (a) 中指令的机器语言翻译。
22. (a) 写出 MIPS 汇编语言语句，在运行时栈上分配 12 字节的存储。(b) 写出 (a) 中指令的机器语言翻译。
23. (a) 假设 MIPS 机器的 C++ 编译器把 $s5 和数组 g 联系在一起。该如何把语句 g = 529371 翻译成 MIPS 汇编语言呢？(b) 写出 (a) 中指令的机器语言翻译。
24. 对于图 12-32 的高速缓存，CPU 请求存地址 4675(dec) 的字节。(a) 标签字段的 9 位是什么？(b) 字节字段的 4 位是什么？(c) 存储这个数据的高速缓存单元是哪个？

25. CPU 可以寻址 16 MB 的主存，使用直接映射高速缓存，其中存储着 256 个 8 字节的高速缓存行。（a）一个内存地址需要几位？（b）地址的字节字段需要多少位？（c）地址的行字段需要多少位？
661 （d）地址的标签字段需要多少位？（e）每个高速缓存条目的数据字段需要多少位？（f）每个高速缓存条目的所有字段一共要多少位？（g）整个高速缓存总共需要多少位？

26. 练习 25 的 CPU 使用的是两路组相联高速缓存，带有 256 个 8 字节高速缓存行。每个高速缓存条目需要多少位？

27. 图 12-33 中，（a）画出比较器的实现，比较器就是里面有一个等号的圆圈。（提示：考虑 XOR 后面跟一个反向器的真值表，有时称为 XNOR 门。）（b）画出 128 个四输入复用器的连接。在本习题的两个部分都可以使用省略号（...）。

28. 直接映射高速缓存是高速缓存设计的一个极端，其组相联在中间；另一个极端是全相联高速缓存（fully-associative cache），实际上就是图 12-32 的高速缓存里只有一个条目，地址的行字段为 0，也就是没有行字段，地址中只有标签字段和字节字段。（a）图 12-33 中，不是有 8 个高速缓存单元，每个 4 行，而是可以用同样的位数，只有 1 个高速缓存单元，该单元有 32 行。这个设计比起图 12-33 中的高速缓存，命中率会提高吗？解释之。（b）对于（a）中的高速缓存，读电路中需要多少个比较器？

29. 假设 CPU 可以寻址 1 MB 的主存，使用全相联映射高速缓存（参见练习 28），其中有 16 个 32 字节的高速缓存行。（a）一个内存地址需要几位？（b）地址的字节字段需要多少位？（c）地址的标签字段需要多少位？（d）每个高速缓存条目的数据字段需要多少位？（e）整个高速缓存总共需要多少位？

30. （a）lw 指令的 RTL 说明是什么？（b）对图 12-36，写出执行 lw 指令的控制信号。

31. 图 12-38 中，（a）两个 IF/ID 边界寄存器中每个有多少位？（b）四个 ID/Ex 边界寄存器中每个有多少位？（c）一个 Ex/Mem 边界寄存器中每个有多少位？（d）两个 Mem/WB 边界寄存器中每个有多少位？

32. 对于图 12-40b，在下表中检查每个周期空闲的电路，列出每个周期空闲的电路总数。

周期	7	8	9	10	11	12	13	14	15	16
IF										
ID										
Ex										
Mem										
WB										
空闲数										

662

33. 假设图 12-40a 的五阶段流水线 15% 的时间在执行分支，每个分支导致接下来要执行的指令停顿直到分支完成，如图 12-40b 所示。（a）和没有气泡的理想流水线相比，周期数增加的百分比是多少？（b）假设 n 阶段流水线 x% 的时间执行分支，每个分支导致接下来要执行的指令停顿直到分支完成。和没有气泡的理想流水线相比，周期数增加的百分比是多少？

34. 书中提到在假设下一条指令会被延迟的情况下，可以消除无条件分支的写回阶段。（a）对于图 12-40b 的周期 7 ~ 16 画出使用这样的设计；（b）对于该设计完成练习 32 的表格。

35. （a）图 12-42 中，对于一位动态分支预测，什么样的跳转结果会导致错误预测比率最高？最大比率是多少？（b）图 12-43 中，对于两位动态分支预测，什么样的跳转结果会导致错误预测比率最高？最大比率是多少？

663 / 664

36. 构建图 12-43 的单输入有限状态机，实现两位动态分支预测，用卡诺图简化电路。（a）使用两个 SR 触发器。（b）使用两个 JK 触发器。（c）使用两个 D 触发器。（d）使用两个 T 触发器。

Pep/8 体系结构

本附录总结了 Pep/8 计算机的体系结构。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0_	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1_	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2_	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3_	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4_	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5_	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6_	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7_	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8_	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9_	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A_	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B_	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C_	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D_	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E_	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F_	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

图 A-1 十六进制转换表

十六进制	二进制	十六进制	二进制	十六进制	二进制	十六进制	二进制
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

图 A-2 十六进制和二进制的关系

字符	二进制	十六进制	字符	二进制	十六进制	字符	二进制	十六进制	字符	二进制	十六进制
NUL	000 0000	00	SP	010 0000	20	@	100 0000	40	`	110 0000	60
SOH	000 0001	01	!	010 0001	21	A	100 0001	41	a	110 0001	61
STX	000 0010	02	"	010 0010	22	B	100 0010	42	b	110 0010	62
ETX	000 0011	03	#	010 0011	23	C	100 0011	43	c	110 0011	63
EOT	000 0100	04	$	010 0100	24	D	100 0100	44	d	110 0100	64
ENQ	000 0101	05	%	010 0101	25	E	100 0101	45	e	110 0101	65
ACK	000 0110	06	&	010 0110	26	F	100 0110	46	f	110 0110	66
BEL	000 0111	07	'	010 0111	27	G	100 0111	47	g	110 0111	67
BS	000 1000	08	(	010 1000	28	H	100 1000	48	h	110 1000	68
HT	000 1001	09	)	010 1001	29	I	100 1001	49	i	110 1001	69
LF	000 1010	0A	*	010 1010	2A	J	100 1010	4A	j	110 1010	6A
VT	000 1011	0B	+	010 1011	2B	K	100 1011	4B	k	110 1011	6B
FF	000 1100	0C	,	010 1100	2C	L	100 1100	4C	l	110 1100	6C
CR	000 1101	0D	-	010 1101	2D	M	100 1101	4D	m	110 1101	6D
SO	000 1110	0E	.	010 1110	2E	N	100 1110	4E	n	110 1110	6E
SI	000 1111	0F	/	010 1111	2F	O	100 1111	4F	o	110 1111	6F
DLE	001 0000	10	0	011 0000	30	P	101 0000	50	p	111 0000	70
DC1	001 0001	11	1	011 0001	31	Q	101 0001	51	q	111 0001	71
DC2	001 0010	12	2	011 0010	32	R	101 0010	52	r	111 0010	72
DC3	001 0011	13	3	011 0011	33	S	101 0011	53	s	111 0011	73
DC4	001 0100	14	4	011 0100	34	T	101 0100	54	t	111 0100	74
NAK	001 0101	15	5	011 0101	35	U	101 0101	55	u	111 0101	75
SYN	001 0110	16	6	011 0110	36	V	101 0110	56	v	111 0110	76
ETB	001 0111	17	7	011 0111	37	W	101 0111	57	w	111 0111	77
CAN	001 1000	18	8	011 1000	38	X	101 1000	58	x	111 1000	78
EM	001 1001	19	9	011 1001	39	Y	101 1001	59	y	111 1001	79
SUB	001 1010	1A	:	011 1010	3A	Z	101 1010	5A	z	111 1010	7A
ESC	001 1011	1B	;	011 1011	3B	[	101 1011	5B	{	111 1011	7B
FS	001 1100	1C	<	011 1100	3C	\	101 1100	5C	|	111 1100	7C
GS	001 1101	1D	=	011 1101	3D	]	101 1101	5D	}	111 1101	7D
RS	001 1110	1E	>	011 1110	3E	^	101 1110	5E	~	111 1110	7E
US	001 1111	1F	?	011 1111	3F	_	101 1111	5F	DEL	111 1111	7F

控制符的缩写

NUL	空字符	FF	换页键	CAN	取消
SOH	标题开始	CR	回车键	EM	介质中断
STX	正文开始	SO	不用切换	SUB	替补
ETX	正文结束	SI	启用切换	ESC	换码（溢出）
EOT	传输结束	DLE	数据链路转义	FS	文件分隔符
ENQ	请求	DC1	设备控制 1	GS	分组符
ACK	收到通知	DC2	设备控制 2	RS	记录分隔符
BEL	响铃	DC3	设备控制 3	US	单元分隔符
BS	退格	DC4	设备控制 4	SP	空格
HT	水平制表符	NAK	拒绝接收	DEL	删除
LF	换行键	SYN	同步空闲		
VT	垂直制表符	ETB	传输块结束		

附 A-3　美国信息交换标准代码（ASCII）

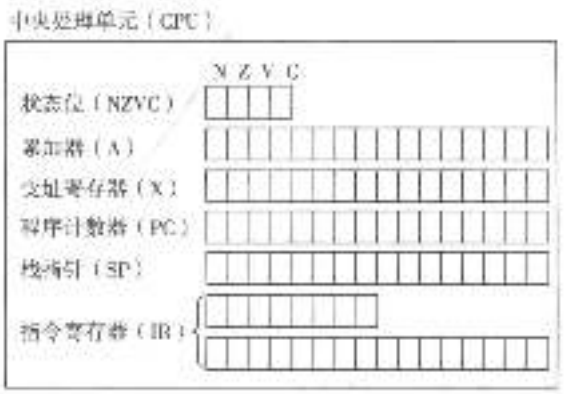


图 A-4 Pep/8 计算机的中央处理单元



图 A-5 Pep/8 的指令格式

aaa	寻址方式
	立即数
	直接
	间接
	栈相对
	栈相对间接
	变址
	栈变址
	栈变址间接

a）寻址 -aaa 字段

a	寻址方式
0	立即数
1	变址

b）寻址 -a 字段

r	寄存器
0	累加器
1	变址寄存器

c）寄存器 -r 字段

图 A-6 Pep/8 指令指示符字段

寻址方式	aaa	字母	操作数
立即数	000	i	OprndSpec
直接	001	d	Mem [OprndSpec]
间接	010	n	Mem [Mem [OprndSpec]]
栈相对	011	s	Mem [SP + OprndSpec]
栈相对间接	100	sf	Mem [Mem [SP + OprndSpec]]
变址	101	x	Mem [OprndSpec + X]
栈变址	110	sx	Mem [SP + OprndSpec + X]
栈变址间接	111	sxf	Mem [Mem [SP + OprndSpec] + X]

图 A-7 Pep/8 的寻址方式

指令指示符	助记符	指令	寻址方式	状态位
0000 0000	STOP	停止执行	U	
0000 0001	RETTR	从陷阱返回	U	
0000 0010	MOVSPA	将栈指针（SP）传送到累加器（A）	U	

图 A-8 Pep/8 指令集

指令指示符	助记符	指 令	寻址方式	状 态 位
0000 0001	MOVFLGA	将 NZVC 标志传送到累加器（A）	U	
0000 010a	BR	无条件分支	i, x	
0000 011a	BRLE	如果小于等于，分支	i, x	
0000 100a	BRLT	如果小于，分支	i, x	
0000 101a	BREQ	如果等于，分支	i, x	
0000 110a	BRNE	如果不等于，分支	i, x	
0000 111a	BRGE	如果大于等于，分支	i, x	
0001 000a	BRGT	如果大于，分支	i, x	
0001 001a	BRV	如果 V，分支	i, x	
0001 010a	BRC	如果 C，分支	i, x	
0001 011a	CALL	调用子程序	i, x	
0001 100r	NOTr	寄存器 r 按位反转	U	NZ
0001 101r	NEGr	寄存器 r 取反	U	NZV
0001 110r	ASLr	寄存器 r 算数左移	U	NZVC
0001 111r	ASRr	寄存器 r 算数右移	U	NZC
0010 000r	ROLr	寄存器 r 循环左移	U	C
0010 001r	RORr	寄存器 r 循环右移	U	C
0010 01nn	NOPn	一元无操作陷阱	U	
0010 1aaa	NOP	非一元无操作陷阱	i	
0011 0aaa	DECI	十进制输入陷阱	d, n, s, sf, x, sx, sxf	NZV
0011 1aaa	DECO	十进制输出陷阱	i, d, n, s, sf, x, sx, sxf	
0100 0aaa	STRO	字符串输出陷阱	d, n, sf	
0100 1aaa	CHARI	字符输入	d, n, s, sf, x, sx, sxf	
0101 0aaa	CHARO	字符输出	i, d, n, s, sf, x, sx, sxf	
0101 1nnn	RETn	从调用返回，带 n 个本地字节	U	
0110 0aaa	ADDSP	加到栈指针（SP）	i, d, n, s, sf, x, sx, sxf	NZVC
0110 1aaa	SUBSP	从栈指针（SP）减去	i, d, n, s, sf, x, sx, sxf	NZVC
0111 raaa	ADDr	加到寄存器 r	i, d, n, s, sf, x, sx, sxf	NZVC
1000 raaa	SUBr	从寄存器 r 减去	i, d, n, s, sf, x, sx, sxf	NZVC
1001 raaa	ANDr	与寄存器 r 按位与	i, d, n, s, sf, x, sx, sxf	NZ
1010 raaa	ORr	与寄存器 r 按位或	i, d, n, s, sf, x, sx, sxf	NZ
1011 raaa	CPr	与寄存器 r 比较	i, d, n, s, sf, x, sx, sxf	NZVC
1100 raaa	LDr	从内存加载到寄存器 r	i, d, n, s, sf, x, sx, sxf	NZ
1101 raaa	LDBYTEr	从内存加载一个字节到寄存器 r	i, d, n, s, sf, x, sx, sxf	NZ
1110 raaa	STr	把寄存器 r 的内容存入内存	d, n, s, sf, x, sx, sxf	
1111 raaa	STBYTEr	从寄存器 r 存一个字节到内存	d, n, s, sf, x, sx, sxf	

图 A-8 （续）

.ADDRSS	符号的地址
.ASCII	ASCII 字节的字符串
.BLOCK	字节块
.BURN	初始化 ROM 陷阱
.BYTE	一个字节的值
.END	汇编器标记
.EQUATE	让一个符号等于一个常数值
.WORD	一个字的值

图 A-9 Pep/8 汇编语言的六个伪操作

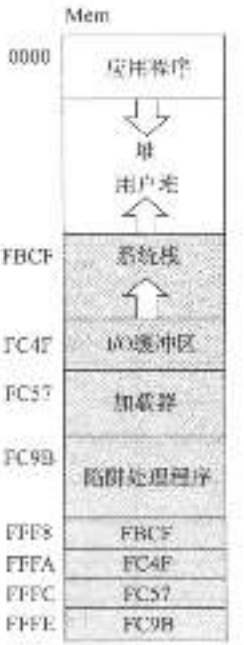


图 A-10 Pep/8 系统的内存列

助 记 符	寄存器传送语言说明
STOP	STOP execution
RETTR	NZVC ← Mem[SP]⟨4..7⟩; A ← Mem[SP+1]; X ← Mem[SP+3]; PC ← Mem[SP+5]; SP ← Mem[SP+7];
MOVSPA	A ← SP
MOVFLAGA	A⟨0..11⟩ ← 0, A⟨12..15⟩ ← NZVC
BR	PC ← Oprnd
BRLE	N = 1 ∨ Z = 1 ⇒ PC ← Oprnd
BRLT	N = 1 ⇒ PC ← Oprnd
BRLO	Z = 1 ⇒ PC ← Oprnd
BRNE	Z = 0 ⇒ PC ← Oprnd
BRGE	N = 0 ⇒ PC ← Oprnd
BRGT	N = 0 ∧ Z = 0 ⇒ PC ← Oprnd
BRV	V = 1 ⇒ PC ← Oprnd
BRC	C = 1 ⇒ PC ← Oprnd
CALL	SP ← SP − 2; Mem[sp] ← PC; PC ← Oprnd
NOTr	r ← ¬r; N ← r<0; Z ← r = 0
NEGr	r ← −r; N ← r<0; Z ← r = 0; V ← *{overflow}*
ASLr	C ← r⟨0⟩, r⟨0..14⟩ ← r⟨1..15⟩, r⟨15⟩ ← 0; N ← r<0; Z ← r = 0; V ← *{overflow}*
ASRr	C ← r⟨15⟩, r⟨1..15⟩ ← r⟨0..14⟩; N ← r<0; Z ← r = 0
ROLr	C ← r⟨0⟩, r⟨0..14⟩ ← r⟨0..15⟩, r⟨15⟩ ← C
RORr	C ← r⟨15⟩, r⟨1..15⟩ ← r⟨0..14⟩, r⟨0⟩ ← C

图 A-11 Pep/8 指令的 RTL 说明

助 记 符	寄存器传送语言说明
NOPn	Trap: Unary no operation
NOP	Trap: Nonunary no operation
DECI	Trap: Oprnd ← *{decimal input}*
DECO	Trap: *{decimal input}* ← Oprnd
STRO	Trap: *{string output}* ← Oprnd
CHARI	byte Oprnd ← *{character input}*
CHARO	*{character input}* ← byte Oprnd
RETn	SP ← SP + n; PC ← Mem[SP]; SP ← SP − 2
ADDSP	SP ← SP + Oprnd; N ← SP < 0, Z ← SP = 0, V ← *{overflow}*, C ← *{carry}*
SUBSP	SP ← SP − Oprnd; N ← SP < 0, Z ← SP = 0, V ← *{overflow}*, C ← *{carry}*
ADDr	r ← r + Oprnd; N ← r < 0, Z ← r = 0, V ← *{overflow}*, C ← *{carry}*
SUBr	r ← r − Oprnd; N ← r < 0, Z ← r = 0, V ← *{overflow}*, C ← *{carry}*
ANDr	r ← r ∧ Oprnd; N ← r < 0, Z ← r = 0
ORr	r ← r ∨ Oprnd; N ← r < 0, Z ← r = 0
CPr	T ← r − Oprnd; N ← T < 0, Z ← T = 0, V ← *{overflow}*, C ← *{carry}*
LDr	r ← Oprnd; N ← r < 0, Z ← r = 0
LDBYTEr	r〈8..15〉 ← byte Oprnd; N ← r < 0, Z ← r = 0
STr	Oprnd ← r
STBYTEr	byte Oprnd ← r〈8..15〉
Trap	T ← Mem[FFFA]; Mem[T−1] ← IR; Mem[T−3] ← SP; Mem[T−5] ← PC; Mem[T−7] ← X; Mem[T−9] ← A; Mem[T−10]〈4..7〉 ← NZVC; SP ← T−10; PC ← Mem[FFFE]

图 A-11 （续）

```
SP := Mem [FFFA]
PC := Mem [FFFC]
```

图 A-12 加载选项

```
SP  := Mem [FFF8]
PC  := 0000
```

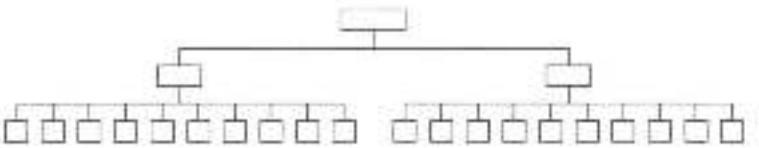
图 A-13 执行选项

部分练习参考答案

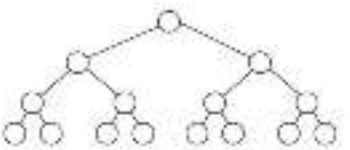
第 1 章

2. (a) 11110，不包括 Khan

3. (a)



4. (a) 参见图；(b) 31



5. 每秒 32 位

9. (a) 153 600 (b) 19 200 字节，大约是 19.2KB

12. (a) 18.5 小时

16.

Temp5

P.Name	P.Major	P.State
Ron	Math	OR

Temp6

S.Name	S.Class	S.Major	S.State
Beth	Soph	Hist	TX
Allison	Soph	Math	AZ

17. (a)

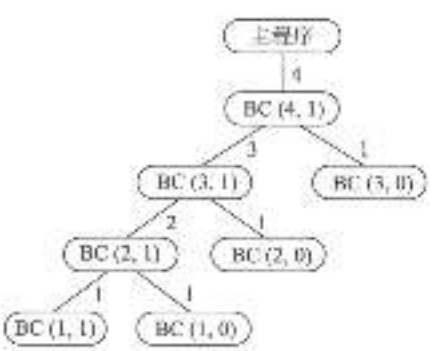
```
select Sor where S.Name = Beth giving Temp
project Temp over S.State giving Result
```

第 2 章

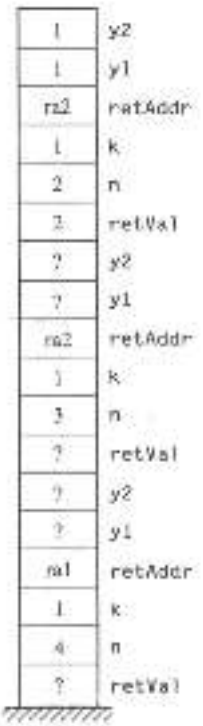
1. (a) 4 次

2. (a) 被调用了 7 次，最多有 4 个活帧。调用顺序是：

```
主程序
  Call BC(4, 1)
      Call BC(3, 1)
          Call BC(2, 1)
              Call BC(1, 1)
          Return to BC(2, 1)
              Call BC(1, 0)
          Return to BC(2, 1)
      Return to BC(3, 1)
          Call BC(2, 0)
      Return to BC(3, 1)
  Return to BC(4, 1)
      Call BC(3, 0)
  Return to BC(4, 1)
Return to main program
```



3. (a) 参见图：



练习 3(a) 的图

第 3 章

1. (a)　(b)　(c)　(d)

267	2102	10101	2433
270	2110	10110	2434
271	2111	10111	2440
272	2112	11000	2441
273	2120	11001	2442
274	2121	11010	2443
275	2122	11011	2444
276	2200	11100	3000
277	2201	11101	3001
300	2202	11110	3002
301	2210	11111	3003

3. (a) 18 (b) 6 (c) 11 (d) 8 (e) 31 (f) 85

5. (a) 11001 (b) 10000 (c) 1 (d) 1110 (e) 101 (f) 101001

7. (a) 00 … 11 (bin)，即 0 … 3 (dec)　　(b) 000 … 111 (bin)，即 0 … 7 (dec)

8. (a) C = 0，111 0100 (b) C = 1 001 0000 (c) C = 1 111 1110 (d) C = 1 000 0000

12. (a) $7\times8^2+0\times8^3+1\times8^2-4\times8^1+6\times8^2$

13. (a) $2\times10^1+9\times10^0-4\times10^{-1}+5\times10^{-2}+8\times10^{-3}$

15. (a) 011 0001 (b) 110 0101 (c) 000 0000 (d) 100 0000 (e) 111 1111 (f) 111 1110

17. (a) 29 (b) -43 (c) -4 (d) 1 (e) -64 (f) -63

19. (a) N=0，Z=0，V=0，C=0，011 1001 (b) N=0，Z=0，V=0，C=1，000 0110
(c) N=0，Z=0，V=1，C=1，001 1011 (d) N=1，Z=0，V=0，C=1，101 0110
(e) N=1，Z=0，V=1，C=0，100 0001 (f) N=1，Z=0，V=0，C=0，111 0100

21. (a) 用二进制表示是 10 … 01，十进制表示是 -2 … 1。
(b) 用二进制表示是 100 … 011，十进制表示是 -4 … 3。

22. (a) N=0，Z=0，010 1000 (b) N=0，Z=0，000 0101 (c) N=1，Z=0，110 1110
(d) N=1，Z=0，101 1111 (e) N=1，Z=0，100 0110 (f) N=1，Z=0，101 1010
(g) 101 0100 (h) 001 0101

24. ASL 操作：
(a) 24 (dec) = 001 1000 (bin)，ASL 001 1000 = 011 0000 (bin) = 48 (dec)，
N=0，Z = 0，V=0，C = 0
(b) 37 (dec) = 010 0101 (bin)，ASL 010 0101 = 100 1010 (bin) = -54 (dec)，
N=1，Z = 0，V=1，C = 0
(c) -26 (dec) = 110 0110 (bin)，ASL 110 0110 = 100 1100 (bin) = -52 (dec)，
N=1，Z = 0，V=0，C = 1
(d) 1 (dec) = 000 0001 (bin)，ASL 000 0001 = 000 0000 (bin) = 2 (dec)，
N=0，Z = 0，V=0，C = 0
(e) 0 (dec) = 0000 0000 (bin)，ASL 000 0000 = 000 0000 (bin) = 0 (dec)，
N=0，Z = 1，V=0，C = 0
(f) -1 (dec) = 111 1111 (bin)，ASL 111 1111 = 111 1110 (bin) = -2 (dec)，
N=1，Z = 0，V=0，C = 1

ASR 操作：

(a) 24 (dec) = 001 1000 (bin), ASR 001 1000 = 000 1100 (bin) = 12 (dec),
N=0, Z = 0, C = 0

(b) 37 (dec) = 010 0101 (bin), ASR 010 0101 = 001 0010 (bin) = 18 (dec),
N=0, Z = 0, C = 1

(c) -26 (dec) = 110 0110 (bin), ASR 110 0110 = 111 0011 (bin) = -13 (dec),
N=1, Z = 0, C = 0

(d) 1 (dec) = 000 0001 (bin), ASR 000 0001 = 000 0000 (bin) = 0 (dec),
N=0, Z = 1, C = 1

(e) 0 (dec) = 0000 0000 (bin), ASR 000 0000 = 000 0000 (bin) = 0 (dec),
N=0, Z = 1, C = 0

(f) -1 (dec) = 111 1111 (bin), ASR 111 1111 = 111 1111 (bin) = -1 (dec),
N=1, Z = 0, C = 1

27. (a) 101 1011, C = 0 (b) 101, 1011, C = 0 (c) 101 0101, C = 1 (d) 001 1011, C = 1

30. (a) 3AB7, 3AB8, 3AB9, 3ABA, 3ABB, 3ABC

31. (a) 11 614 (dec)

32. (a) 68CF

34. (a) -35 (b) 47 (c) -64

36. (a) 65 (hex)(b) 3F (hex)(c) 7F (hex)

36. Have a nice day!

40. 101 000 110 0001 111 10001 010 0000 010 0100 011 0000 010 1110 011 1001 011 0010

43. (a) 八进制数字表示 3 位。

44. (a) 6.640625 (b) 0.046875 (c) 1.0

46. (a) 1101.00101 (b) 0.0000101 (c) 0.100110011100...

50. (a) 1 110 1001

51. (a) 0.90625

52. (a) 41D8 D000

53. (a) $64.0=1.0\times2^5$

第 4 章

1. (a) 65 536 字节 (b) 32 768 字 (c) 524 288 位 (d) 92 位 (e) 大 5699 倍

3. 对于指令 7AF82C

(a) opcode=0111
(b) 这条指令把一个数加到寄存器 r
(c) r=1
(d) 变址寄存器，X
(e) aaa=010
(f) 间接寻址
(g) OprndSpec=F82C

对于指令 D623D0

(a) opcode=1101
(b) 从内存加载一个字节到寄存器 r
(c) r=0
(d) 累加器，A
(e) aaa=110
(f) 线间接寻址
(g) OprndSpec=23D0

5.

	A	X	Mem[0A3F]	Mem[0A41]
原始内容	19AC	FE20	FF00	103D
(a) 加载累加器	FF00	FE20	FF00	103D
(b) 加载字节到累加器	19FF	FE20	FF00	103D

(c) 加数字节到变址寄存器	19AC	FE10	FF00	103D
(d) 存储字节累加器	19AC	FE20	FF00	AC3D
(e) 存储变址寄存器	19AC	FE20	FE20	103D
(f) 从变址寄存器减去	19AC	EDE3	FF00	103D
(g) 从累加器减去	1AAC	FE20	FF00	103D
(h) OR 累加器	FFAC	FE20	FF00	103D
(i) 反转变址寄存器	19AC	01DF	FF00	103D

7. JOY

9. (a) M

第 5 章

1. (a) ORX 0xEF2A, n　(b) MOVSPA　(c) LDBYTEA 0x0030, sxf

3. (a) IC　(b) 4800F　(c) 0C01E6

5. (a) 42 65 61 72 00　(b) F8　(c) 0316

7. mug

10. −57
    72
    H1

12. (a) 目标代码是 38　00　6D　50　00　0A　38　6D　50　00　0A　50　00　26　00
    输出
    109
    28013
    &

13. here 的值是 0003，there 的值是 0005。目标代码是 04　00　05　00　09　39　00　03　00

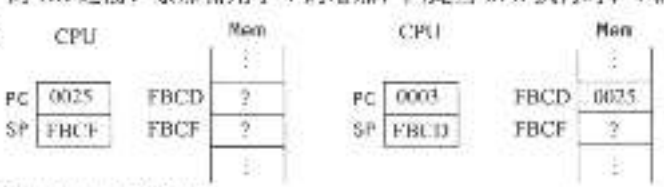
15. this 的值是 0000，输出是 Q，因为 51 (hex) = Q (ASCII)，这里的 51 是目标代码的第一个字节。

18. 编译器用它的符号表来存储每个变量的类型。每当遇到表达式或赋值语句时，编译器就会查询符号表，验证类型是否兼容。

第 6 章

3. 因为无论控制来自 0009 的 STA 还是循环底部的 BR，1 的当前值都会放在累加器中，在循环底部的 BR 之前，累加器用于 1 的增加，因此当 CPA 执行时，1 的当前值仍然会在累加器中。

6.



8. 分支地址计算如下

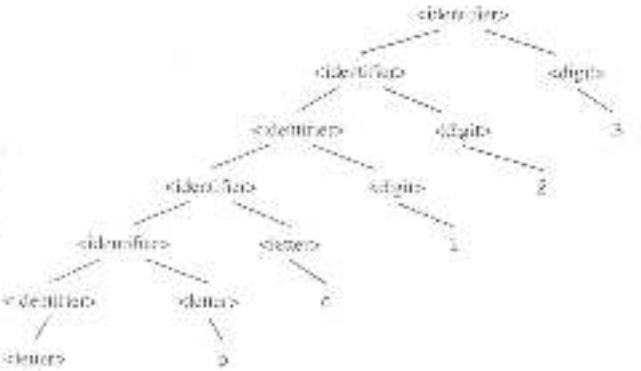
```
Oprnd=Mem[OprndSpec + X]
     =Mem[0013 + 8]
     =Mem[00B]
     =4100
```

从程序代码中无法看出 4100 处的内容是什么，假设它们为全 0，那么冯·诺依曼周期会把地址 4100 处的 00 当作 STOP 指令。

第 7 章

1. 计算机科学的基本问题是：什么能够自动化？

3. (a) <identifier> → <identifier> <digit>
→ <identifier> 3
→ <identifier> <digit> 3
→ <identifier> 2 3
→ <identifier> <digit> 2 3
→ <identifier> 1 2 3
→ <identifier> <digit> 1 2 3
→ <letter> c 1 2 3
→ <identifier> <letter> c 1 2
→ <identifier> b c 1 2 3
→ <letter> b c 1 2 3
→ a b c 1 2 3

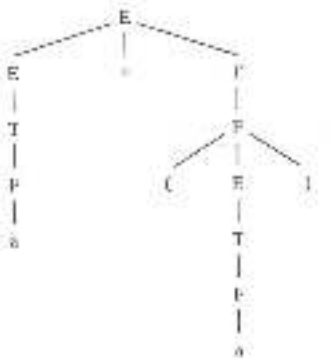


4. (a) L → FM
→ − M Rule 3
→ − d Rule 5

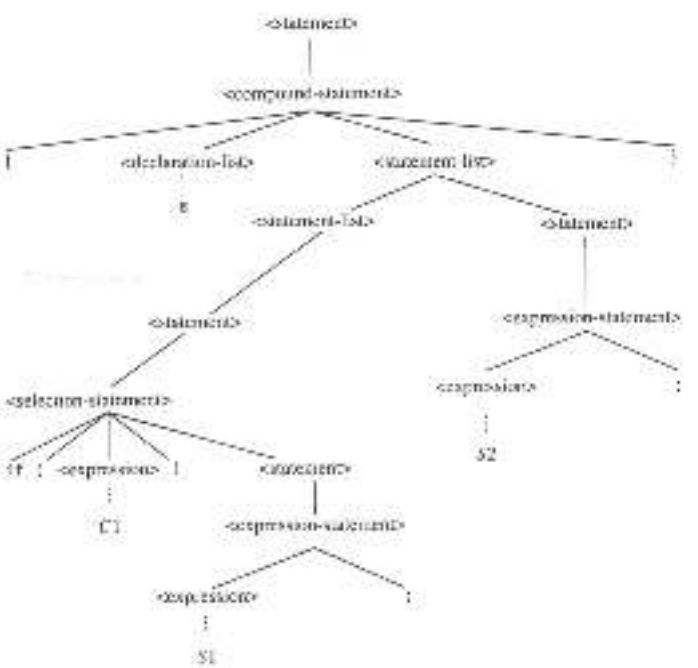


5. (a) A → a b C Rule 2
⇒ a b c Rule 5

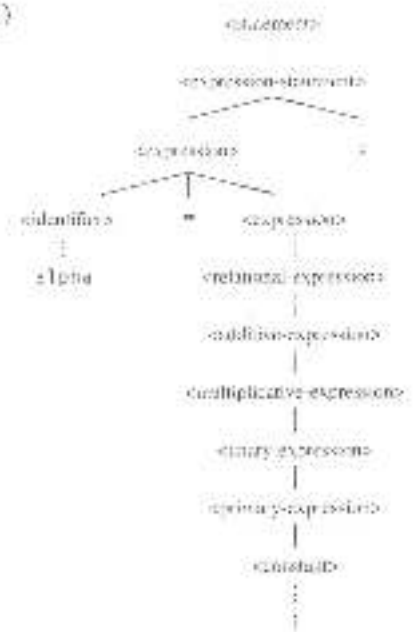
6. (a) 可以按照如下方式推导：



7. (a)



8. (a)



11. (a) 该状态机是确定的，没有不可达状态。

13. (a)



第 8 章

2. (a) 0036 (hex)，是第 27 个字节中 6 对应的 ASCII 码，6C

4. (a) 0051，被中断指令的指令指示符

5. (a) 0039，被中断指令的指令指示符

6. (a) 0041，被中断指令的指令指示符

7. (a) 指令通过掩码左边的位，把累加器的值从 0033（即 3 的 ASCII 值）改变为 0003。(b) 0007，37 中第二个字符的数值；(c) 0000，init 的值，起始状态。

8. 提示：第一个字符输入是 ASCII 的连接符字符。

9. (a) 0025 (hex)，即 37 (dec)。(b) 0025，没有取反，因为它已经是非负的了。(c) FF78 的 CALL 是用于写 100 的位置；累加器的值还是 0025，即 37 (dec)，因为 37 mod 100 就是 37。

10. 提示：-2068 (dec) = F7EC (hex)，它的补码是 0814 (hex)。

11. (a) 0012，要输出的字符串的第一个字节的地址 (b) 0054，字母 T 的 ASCII 值

12. (a) 0008，DECI 后面一条指令的地址 (b) 0006，DECI 操作数指示符的地址 (c) 0003，DECI 操作数的地址

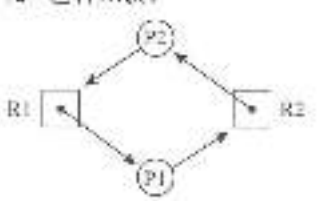
18. (a) 算法不能保证互斥。假设 P1 和 P2 都在它们各自的其余部分中，enter1 和 enter2 均为假。P1 可以执行它的 while 循环测试，然后被中断，此后 P2 会执行它的 while 循环测试，然后它们可能把各自的 Enter 变量赋值为真，同时进入临界区。

20. (a) S = 0，没有被阻塞的进程。

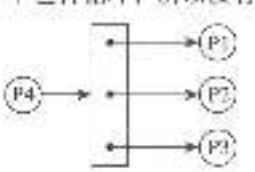
22. (a) 算法能保证互斥。

24. (a) 不可能保证互斥。能找到一个序列使得 P1 和 P2 同时进入它们的临界区吗？不过，不会出现死锁。

25. (a) 包含死锁；



(b) 不包含循环，所以没有死锁；



第 9 章

2. (a) 一个边界寄存器就足够了，因为每次只能有一个进程在执行。如果用户进程试图访问逻辑地址空间外的内存地址，硬件必须中断页表的访问，因为该页不在主存中。能对图 9-9 做这样的修改吗？操作系统必须记录多少边界寄存器？每个进程一个，页表中每个页一个或者每个主存帧一个？

4. (a) 2^{12} 或 4098 字节。

6. 那些能位值为 N 的页，也就是帧 2、5 和 6 中的页。

8. 一个分配了三帧的作业的页访问序列的起始是 1，2，3，1，4，2，…，对这个序列 FIFO 有四次缺页，LRU 有五次缺页。能继续完成这个序列使得 FIFO 在这个特殊的情况下更好么？

10. 产生五次缺页，相比之下 FIFO 是七次，LRU 是六次。可以追踪该算法验证这张图。

11. (a) 提示：最糟的情况是读/写头刚好经过块的起始位置。因此，磁盘必须完整转一圈。可以从 RPM 数算出这个时间。

12. (a) 四个数据位 (b) 一个奇偶位

16. (a) 错误发生在位置 2。纠正后的编码是 1101 1010 1001。

第 10 章

1. (a) $x+1=x\cdot(x+x')$ 互补律

$=(x\cdot x')+x'$ 结合律

$=x+x'$ 幂等律

$=1$ 互补律

4. 要证明 $a+b$ 的补是 $a'b'$，必须证明

$(a+b)\cdot(a'\cdot b')=0$ 和 $(a+b)+(a'\cdot b')=1$

证明的第一部分如下

$(a+b)\cdot(a'\cdot b')=[(a'\cdot b')\cdot a]+[(a'\cdot b')\cdot b]$ 交换律

$=[(a'\cdot a)\cdot b']+[a'\cdot(b\cdot b')]$ 交换律，结合律

$=[0\cdot b']+[a'\cdot 0]$ 互补律

$=0$ 恒等律

6. (a) 提示：

$(x+y)\cdot(x'+y)=(y+x)\cdot(y+x')$ 交换律

$=y+(x\cdot x')$ 分配律

等

8. (a)

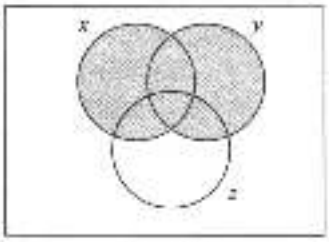


$x=a+b+c$

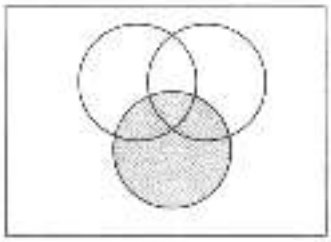
a	b	c	x
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

9. (a) 任何集合和空集的并都是它自己。

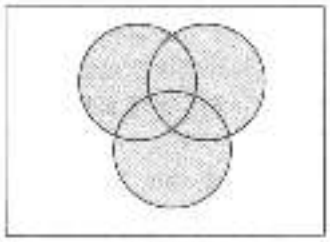
10.(a)



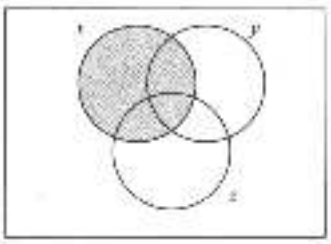
(1) $(x+y)$



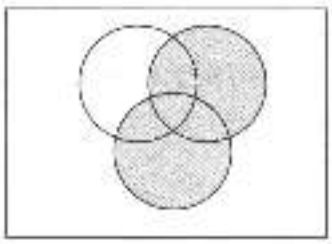
(2) z



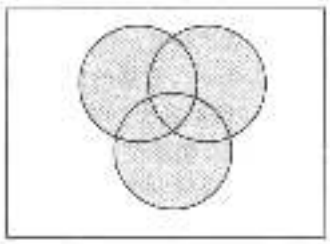
(3) $(x+y)+z$



(4) x



(5) $(y+z)$



(6) $x+(y+z)$

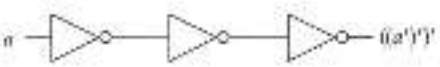
13. (a)

x	y	x AND y	x	y	x AND y	x	y	x AND y
0	0	0	A	B	0	1	0	0
0	A	0	A	1	A	1	A	A
0	B	0	B	0	0	1	B	B
0	1	0	B	A	0	1	1	1
A	0	0	B	B	B			
A	A	A	B	1	B			

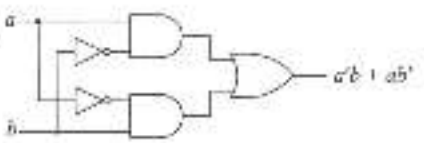
14. (a)



15. (a)



(c)



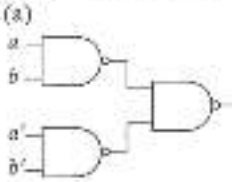
19. (a) $y(a, b, c) = a'bc + ab'c'$

20. (a) $y(a, b, c) = (a + b + c')(a + b + c')(a + b' + c)(a' + b + c')(a' + b' + c)(a' + b' + c')$

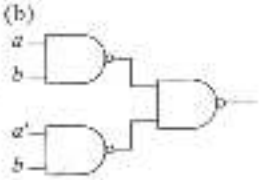
21. (a) $ab + a'b'$ (d) $a'b + ab$

a	b	20 (a)	20(d)
0	0	1	0
0	1	0	1
1	0	0	0
1	1	1	1

22. (a)



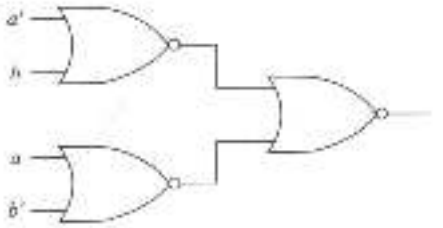
(b)



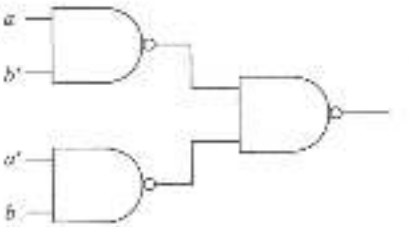
23. (b) $(a' + b)(a + b')$

a	b	
0	0	1
0	1	0
1	0	0
1	1	1

24. (b)

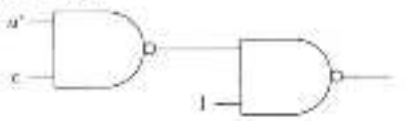


25. (a)

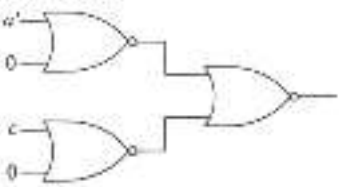


27. (a) $\Sigma(0, 3)$ (d) $\Sigma(1, 3)$ 28. (a) $\prod(1, 2)$

29. (a) $x = a'c$



30. (a) $x = a'c$



31. (a) $x(a, b, c) = ac - b'c'$ 32. (a) $x(a, b, c) = \prod(1, 2, 3, 6) = (a - c')(b' + c')$

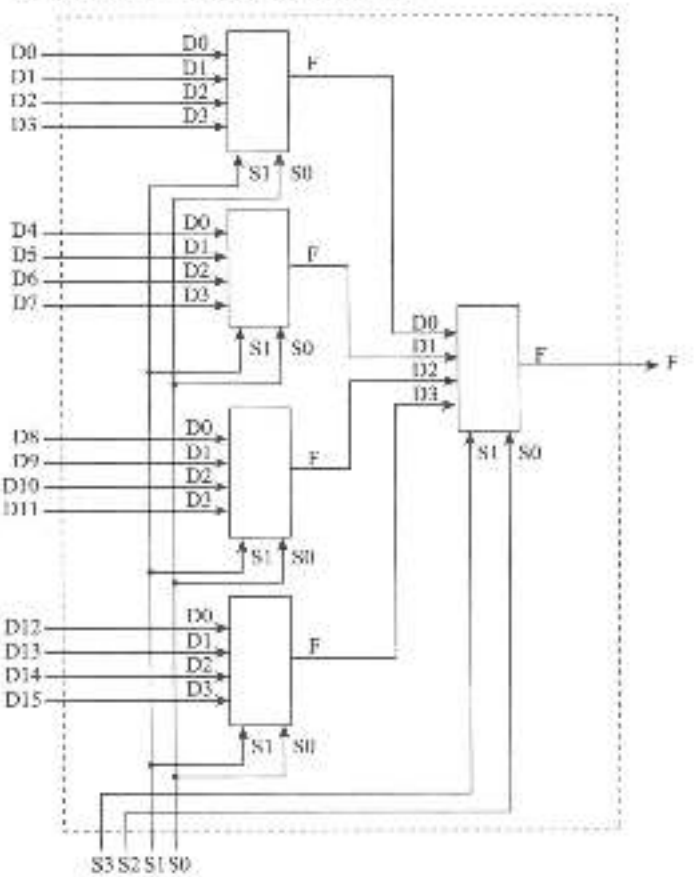
33. (a) $x(a, b, c) = bc' + a'b'c' + b'cd'$

34. (a) $\prod(0, 1, 6, 7, 8, 9, 11, 14, 15)$, $x(a, b, c, d) = (b + c)(b' - c')(a' - c' + d')$ 或者 $x(a, b, c, d) = (b + c)(b' - c')(a' + b + d')$

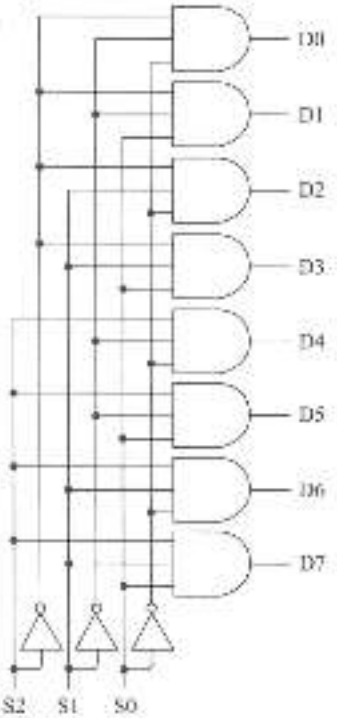
35. (a) $x(a, b, c) = a'b' + ab$ 36. (a) $x(a, b, c, d) = bc + bd$

38. (a) 控制线作为使能，当控制线为 0 时，数据会不改变地通过；当控制线为 1 时，会禁止输出，输出被设置为 1，无论数据输入是什么。

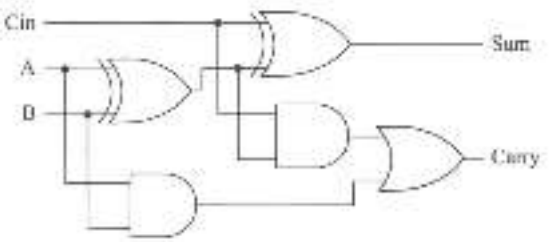
40.



42. (b)



46. (a) Cin



(b) 最大 3 个门的延迟。

47. (b) 提示：如果看一下图 10-52，全加器有 3 个门延迟，半加器有一个门延迟，所以整个的门延迟是 10。不过，其实比这个数值小。

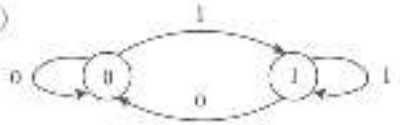
第 11 章

1. 如果有偶数个反相器，网络会稳定。

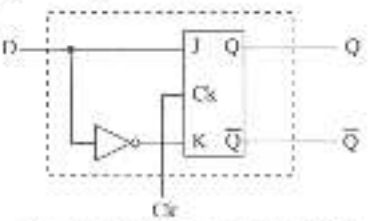
3.

	A	B	C	D	E	R2	S2	Q	$\overline{Q}$
(a)	0	0	1	1	0	1	0	0	1
(b)	0	0	0	0	0	1	0	0	1

5. (b)



8. (a)



9. (a) 提示：使得主和从触发器为 0，Clear 输入应该（1）迫使从触发器连接到主触发器，无论反相器的输出是什么，（2）迫使主触发器变成 Q = 0 状态。要实现（2），考虑增加第三条输入线，连接到下 NOR 门中的一个，或者第三条输入线连接到主 AND 门中的一个。

13. (a) $DA = A\overline{X1} + A\overline{X1} + A\,B\,\overline{X2} + B\,X1\,X2$

$DA = B\,\overline{X1} + B\,X1\,X2 + \overline{A}\,B\,X1$

$Y = A\,X2 + A\,X1\,X2$

15. (a) $JA = \overline{B}\,\overline{C}\,\overline{X} + B\,C\,X$　$KA = \overline{B}\,\overline{C}\,\overline{X} + B\,C\,X$

$JB = \overline{C}\,\overline{X} + C\,X$　$KB = \overline{C}\,\overline{X} + C\,X$

$JC = 1$　$KB = 1$

第 12 章

7. (a)

```
// STBYTEA there,n
// RTL: byte Opmnd <- r<8..15>
```

```
// Oprnd = Mem[Mem[OprndSpec]]

// T3<high> <- Mem[OprndSpec]
1. A=9, B=10; MARCk
2. MemRead
3. MemRead, MDRMux=0; MDRCk
4. AMux=0, ALU=0, CMux=1, C=14; LoadCk

// T2 <- OprndSpec + 1
5. A=10, B=23, AMux=1, ALU=1, CMux=1, C=13; CCk, LoadCk
6. A=9, B=22, AMux=1, ALU=2, CMux=1, C=12; LoadCk

// T3<low> <- Mem[T2]
7. A=12, B=13; MARCk
8. MemRead
9. MemRead, MDRMux=0; MDRCk
10. AMux=0, ALU=0, CMux=1, C=15; LoadCk

// Mem[T3] <- A<low>, restore C in T1 from fetch
11. A=14, B=15; MARCk
12. A=1, AMux=1, ALU=0, CMux=1, MDRMux=1; MDRCk
13. MemWrite
14. MemWrite, A=11, AMux=1, ALU=15; CCk
```

You can combine cycles in the above solution to get an implementation with only 12 cycles as follows:

```
// T3<high> <- Mem[OprndSpec], T2 <- OprndSpec + 1
1. A=9, B=10; MARCk
2. MemRead, A=10, B=23, AMux=1, ALU=1, CMux=1, C=13; CCk, LoadCk
3. MemRead, MDRMux=0, A=9, B=22, AMux=1, ALU=2, CMux=1, C=12;
   MDRCk, LoadCk
4. AMux=0, ALU=0, CMux=1, C=14; LoadCk

// T3<low> <- Mem[T2]
5. A=12, B=13; MARCk
6. MemRead
7. MemRead, MDRMux=0; MDRCk
8. AMux=0, ALU=0, CMux=1, C=15; LoadCk

// Mem[T3] <- A<low>, restore C in T1 from Fetch
9. A=14, B=15; MARCk
10. A=1, AMux=1, ALU=0, CMux=1, MDRMux=1; MDRCk
11. MemWrite
12. MemWrite, A=11, AMux=1, ALU=15; CCk
```

9. 提示：(a) 使用 160 亿 GB 等于 $16\times10^9\times10^9$ 字节，芯片面积是 4×4 平方米，但是必须给出计算过程。(b) 是的，但也必须给出解释。

```
16. (a) lw $t0,20($s6)  # Register $t0 gets b[5]
        sdc $t0,$s5,$t0 # Register $t0 gets g + b[5]
        sw $t0,16($s4)  # a[2] gets g + b[5]

    (b) 100011 10110 01000 0000000000010100
        000000 10101 01000 01000 00000 100000
        101011 10100 01000 0000000000010000
```

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

Absorption property（吸收率），参见 Boolean algebra（吸收率），Boolean algebra

Abstraction（抽象），3-10

　definition of（抽象的定义），3

　in computer system（计算机系统中的抽象），8-10，300，657

Add instruction（加法指令），Pep/8，160-161

add instruction, MIPS（add 指令，MIPS），629，645-646

ADDA instruction, Pep/8（ADDA 指令，Pep/8），198-200，607-608，624

Adder（加法器），528-531

　full（全加器），529

　half（半加器），528

　ripple-carry（行波进位加法器），530-531

Adder/subtracter（加法器 / 减法器），531-532

Addition（加法）

　signed（有符号加法），97-99，537-539

　unsigned（无符号加法），95-96

　16-bit with two 8-bit（用 8 位实现 16 位加法），538-539

Address decoding（地址译码），582-588

Addressing modes，MIPS，（寻址方式，MIPS）626

　base（基址寻址），627，630，645

　immediate（立即数寻址），627，632-633

　PC-relative（PC 相对寻址），627，644-645

　pseudodirect（伪直接寻址），627，644

　register（寄存器寻址），627，629，645-646

Addressing modes，Pep/8（寻址方式，Pep/8），154-156，192，285，605

　direct（直接寻址），157-158，606-607

　immediate（立即数寻址），202-203，607-608

　indexed（变址寻址），287-288，299-300，630

　indirect（间接寻址），305-306，608-611

　stack-indexed（栈变址寻址），290-291

　stack-indexed deferred（栈变址间接寻址），295-297，318-319

　stack-relative（栈相对寻址），238-241

　stack-relative deferred（栈相对间接寻址），276-277，309

　trap assertion（陷阱断言），401-403

　trap operand computation（陷阱操作数计算），403-406

ADDRSS pseudo-op（.ADDRSS 伪操作），194，299

ADDSP instruction（ADDSP 指令），238

Aiken, Howard H.，89

Alphabet（字母表），332，335

Algorithm（算法），17

ALU，532-540

American Standard Codard Code for Information Interchange（美国信息交换标准代码），参见 ASCII

Analysis vs.design（分析与设计）

　hardware（硬件），565

　software（软件），20-22

AND gate（AND 门），497-499

AND operator（AND 运算符），107-108

AND-OR circuit（AND-OR 电路），506-509

And instruction（与指令），162-163

App7，8-10

Arithmetic Logic Unit（算术逻辑单元），参见 ALU

Arithmetic shift left（算术左移），参见 ASL

Arithmetic shift right（算术右移），参见 ASR

Arrays（数组），参见 Parameter（参数）

　in C++（C++ 中的数组），46-47

　called by reference in C++（C++ 中的传引用

调用），61
global（全局数组），285-288
local（局部数组），288-291
MIPS（MIPS 数组），631-632
ASCII，115-117
.ASCII pseudo-op（.ASCII 伪操作），194，195-196，215-217
ASL operation（ASL 运算），109-111
ASLR instruction（ASLr 指令），224-226
ASR operation（ASR 运算），109-111
ASRr instruction（ASRr 指令），224-226，611-612
Assembler（汇编器），196-197
cross（交叉汇编器），201-202
resident（常驻汇编器），201-202
using Pep/8（Pep/8 的汇编器），200-201
Assignment operator（赋值操作），38
Assignment statement（赋值语句），218-221，228
Associative law（结合律），参见 Boolean algebra
Asynchronous interrupt（异步中断），420-421

B

Backus Naur Form（巴科斯范式），参见 BNF
Base conversion（基数转换）
floating point ad decimal（浮点数和十进制数），118-127
hexadecimal and decimal（十六进制数和十进制数），113-115
signed integer and decimal（有符号整数和十进制数），101-104
unsigned integer and decimal（无符号整数和十进制数），93-94
Base register（基址寄存器），参见 Register
BCD，134
Belady's anomaly（Belady 异常），463-464
Best-fit algorithm（最优适配算法），453-454
Binary coded decimal（二进制编码的十进制数），参见 BCD
Binary decoder（二进制译码器），526-527
Bit（位），90
.BLOCK pseudo-op（.BLOCK 伪操作），194，197-198
BNF，344
Boolean algebra（布尔代数），491-197
absorption property（吸收属性），494
associative law（结合律），493
axiom（定理），492
complement（互补律），496-497
consensus theorem（一致性定理），494
DE Morgan's law（德·摩根定律），495-496
distributive law（分配律），492
duality（对偶性），492
idempoteent property（幂等性），493-494
zero theorem（零元定理），494
Boolean expression（布尔表达式）
and logic diagram（和逻辑图），501-503
and truth table（和真值表），503-506
Boolean operator（布尔运算符），42
Boolean type（布尔类型），281-284
Bound register（边界寄存器），参见 Register
BR instruction（BR 指令），203-206
BRC instruction（BRC 指令），244
BREQ instruction（BREQ 指令），244
BRGE instruction（BRGE 指令），244
BRGT instruction（BRGT 指令），244
BRLE instruction（BRLE 指令），244
BRLT instruction（BRLT 指令），244
BRNE instruction（BRNE 指令），244
BRV instruction（BRV 指令），244
.BURN instruction（.BURN 伪操作），194，392-393
Bus（总线），11，575-576
width（总线宽度），613-615
Byte（字节），13
.BYTE pseudo-op（.BYTE 伪操作），194，198-200

C

C bit（C 位），96，105，150-151，532，533，599
C++
compiler（编译器），33-34，214-218
compiler，optimizing（编译器，优化），246-247
machine independence（机器无关性）34
memory model（内存模型），35-36

memory model, heap (内存模型, 堆), 303
Cache memories (高速缓冲存储器), 634-642
direct-mapped (直接映射), 637-638
fully-associative, Exercise12.28 (全相联, 练习 12.28), 662
locality of reference (引用局部性), 635
MIPS, 642-644
set-associative (组相联), 639-642
write policies (写策略), 639-642
CALL instruction (CALL 指令), 260-263
Call-by-reference parameter (传引用参数), 参见 Parameter
Call-by-value parameter (传值参数), 参见 Parameter
Canonical expression (范式), 510-511
Carry bit (进位位), 参见 C bit
Central Processing Unit (CPU, 中央处理单元), 11, 16-17
MIPS, 642-646
Character input/output instruction (字符输入/输出指令), 166-168
Character representation (字符表示), 115-118
Characteristic table (特征表), 558
for D flip-flop (D 触发器), 562
for JK flip-flop (JK 触发器), 560
for SR flip-flop (SR 触发器), 558
for T flip-flop (T 触发器), 563
CHARI instruction (CHARI 指令), 197-198, 218-221
CHARO instruction (CHARO 指令), 195-196
cin statement (cin 语句), 218-221
CISC, 620
Clear flip-flop line (清除触发器线), 569
Clocked SR flip-flop (钟控 SR 触发器), 552-554
Closure of an alphabet (字母表的闭包), 333
Code generator (代码生成器), 331, 368-381
Combinational circuit (组合电路), 490-491
Compiler (编译器), 参见 C++
Complements (互补律), 参见 Boolean algebra
Concatenation (连接), 333
Concurrent processes (并发进程), 424-425
Consensus theorem (合意定理), 参见 Boolean algebra
Constants (常数), 226-228
Context sensitive grammar (上下文相关语法), 338-339, 346
Conversion between base (不同基数之间的转换), 参见 Base conversion
cout statement (cout 语句), 215-221
CPr instruction (CPr 指令), 247-249
Critical section (临界区), 426-427

D

D flip-flop (D 触发器), 562-563
Dare forwarding (数据传送), 参见 Pipelining
Database system (数据库系统), 22-27
Deadlock (死锁), 435-437
DECI instruction (DECI 指令), 194, 203-206, 218-221, 396, 408-414
DECO instruction (DECO 指令), 196, 203-206, 396, 414-417
Decoder (译码器), 参见 Binary decoder
delete operator (delete 操作符), 35, 74
De Morgan's law (德·摩根定律), 参见 Boolean algebra
Demultiplexer (解复用器), 527-528
Denormalized numbers (非正规化数), 125
Derivations in a grammar (语法推导), 336
Deterministic FSM (确定性 FSM), 参见 Finite state machine
Direct addressing (直接寻址), 参见 Addressing modes
Direct memory access (直接内存访问), 656
Disassemblers (反汇编器), 209-211
Disk drives (磁盘驱动器), 465-466
Distributive law (分配律), 参见 Boolean algebra
Division (除法)
integer vs. floating point (整数与浮点数), 40-41
do loop (do 循环), 45-46, 250-252
Don't care condition (无关条件), 参见 Karnaugh
DRAM, 581

Duality（对偶性），参见 Boolean algebra
Dynamic branch prediction（动态分支预测），参见 Pipelining
Dynamic memory allocation（动态内存分配），74-80

E

Eckert, J. Prosper，90，225
Edge-triggered flip-flops（边缘触发的触发器），555
EEPROM，581
Empty string（空字符串），333
Empty transition（空转移），349-351
Enable lines（使能线），523-525
.END pseudo-op（.END 伪操作），194，195-196
ENIAC computer（ENIAC 计算机），90
EPROM，581
.EQUATE pseudo-op（.EQUATE 伪操作），194，226-228，243
Error-correcting codes（纠错码），472-476
Error-detecting codes（检错码），470-472
Excess representation（余码表示），120-121
Excitation tables（激励表），559
  for D flip-flop（D 触发器），564
  for JK flip-flop（JK 触发器），564
  for SR flip-flop（SR 触发器），559
  for T flip-flop（T 触发器），564
Exclusive OR（异或），参见 XOR

F

FIFO page-replacement algorithm（FIFO 页替换算法），462-464
Finite state-machine（有限状态机），332，346-360，655，657
  deterministic（确定型有限状态机），348，351
  direct code（直接编码有限状态机），357-360
  implementation of（有限状态机的实现），355-357
  empty transition in（有限状态机中的空转移），349-351
  nondeterministic（非确定型有限状态机），348-349
  simplified（简化的有限状态机），347-348
  table-lookup implementation of（有限状态机的查找表实现），355-357
First-fit algorithm（最先适配算法），454
Flash memory（闪存），581
Floating point representation（浮点数表示），118-130
for loop（for 循环），46-47，252-253
FORTRAN，256，259
Format trace tag（格式跟踪标签），参见 Trace tag
Full adder（全加器），参见 Adder
Functions（函数），参见 Parameter
  call mechanism（函数调用机制），35，263，265，269
  prototype（函数原型），72
  void（空函数），48-50，260-267

G

Gate delay（门延迟），506
Global varible（全局变量），参见 Variable
Goto controversy（Goto 争议）
Grammar（语法），332，335-346
  context sensitive（上下文有关语法），338-339，346
  for C++ identifiers（C++ 标识符语法），336-337
  for C++ language（C++ 语言语法），341-346
  for expressions（表达式语法），340-341
  for signed integer（有符号整数的语法），337-338
  four parts of（语法的 4 个部分），335

H

Half adder（半加器），参见 Adder
Hamming distance（海明距离），471
Hardware（硬件），10-17
Hazard（危害），参见 Pipelining
Heap（堆），参见 C++ memory model
Hexadecimal representation（十六进制表示），112-115
Hidden bit（隐藏位），121-122
HOL6，8-10

I

Idempotent property（幂等属性）, 参见 Boolean algebra
Identity element（单位元）, 333
IEEE 754 floating point（浮点表示）, 127-130
if statement（if 语句）, 41-43, 245-246, 247-249
Immediate addressing（立即数寻址）, 参见 Addressing mode
Indexed addressing（变址寻址）, 参见 Addressing mode
Indirect addressing（间接寻址）, 参见 Addressing mode
Instruction reordering（指令重排序）, 参见 Pipelining
Instruction set（指令集）
  MIPS（指令集）, 627-628
  Pep/8 at ISA3（ISA3 层的 Pep/8 指令集）, 155
  Pep/8 at Asmb5（Asmb5 层的 Pep/8 指令集）, 193
Instruction specifice（指令指示符）, 154-156
Integer（整数）
  conversions（整数转换）, 参见 Base conversion
  range for signed（有符号整数的范围）, 99-101
  range for unsigned（无符号整数的范围）, 94-95
  signed binary representation（有符号整数的二进制表示）, 97-99
  unsigned binary representation（无符号整数的二进制表示）, 91-92
Invert instruction（按位取反指令）, 164
Inverter（反相器）, 497-499
ISA3, 8-10

J

JK flip-flop（JK 触发器）, 560-562
join operator（join 操作符）, 25-26

K

Karnaugh map（卡诺图）, 512-523
  don't care conditions（无关条件）, 522-523
  dual（对偶）, 521-522
  four-variable（四变量卡诺图）, 518-521
  three-variable（三变量卡诺图）, 512-518

L

Language（语言）, 333-334
Latency（延迟）, 466
LDr instruction（LDr 指令）, 198-200, 608-611
LG1, 8-10
Linked data structures（链接的数据结构）, 77-80, 314-318
Load byte instruction（加载字节指令）, 165-166
Load instruction（装入指令）, 158-159
Loader（加载器）, 392-395
  relocatable（可重定位的）, 449
Local variable（局部变量）, 参见 Variable
Logic diagram（逻辑图）, 497-499, 501
  and boolean expression（逻辑图和布尔表达式）, 501-503
Logical addrss（逻辑地址）, 449-450, 458
LRU pabe-replacement algorithm（页替换算法）, 464
lui instruction, MIPS（lui 指令, MIPS）, 633
lw instuction, MIPS（lw 指令, MIPS）, 629-630

M

Machine vector（机器向量）, 182-183
MAR, 599
Mark I computer（Mark I 计算机）, 89
Master-slave SR flip-flop（主-从 SR 触发器）, 554-559
MDR, 599
Megahertz myth（主频神话）, 655
Memory（存储器）, 16
  alignment（内存对齐）, 613-614
  hierarchies（存储器层次结构）, 642
  main, of Pep/8（Pep/8 的主存）, 151-153, 392
  map of Pep/8（Pep/8 的内存图）, 182
  random access（随机访问存储器）, 181
  read-only（只读存储器）, 181-182
  subsystems（存储器子系统）, 577-581

Memory model（内存模型），参见 C++ memory model
Microcode（微代码），621-624
Minterm（最小项），510，512
MIPS，624-634，624-655
Mnemonics（助记符），192-194
Monostable multivibrator（单稳多谐振荡器），580
Moore's Law（摩尔定律），614
MOVSPA instuction（MOVSPA 指令），280-281
Multiple token recognizers（多语言符号识别器），351-354，361-368
Multiplexer（复用器），525-526
Multiplication algorithm（乘法算法）
  iterative，Probem 6.24（迭代的，问题 6.24），324-325
  recursive，Problem 6.18（递归的，问题 6.18），323
Multiprocessing（多处理），423，425
Multiprogramming（多道程序设计），421，425
  fixed-partion（固定分区的），448-451
  variable-partition（可变分区的），452-454
Mutual exclusion（互斥），427-435

N

N bit（N 位），106，150-151，533，599
NaN，124
NAND gate（NAND 门），498
NAND-NAND circuit（NAND-NAND 电路），508-509
NEG operation（NEG 运算），98-99
Negate instruction（取负指令），164-165
Negative bit（负数位），参见 N bit
new operator（new 操作符），35，74-75，300-306
Nondeterministic FSM（非确定性 FSM），参见 Finite state machine
NOP instruction（NOP 指令），194，396，406-407
NOPn instruction（NOPn 指令），194，396，406-407
NOR-NOR circuit（NOR-NOR 电路），509
NOR gate（NOR 门），498
NOT operation（NOT 运算），98-99
Nybble（4 位字节），394-395

O

Octal（八进制），92
One shot device（单次设备），参见 Monostable multivibrator
Ones' complement（反码），98
Opcode（操作码），154
  unimplemented（未实现的操作码），194
Operand specifier（操作数指示符），154-156
Operating system（操作系统），19-20，391
Optimization（优化）
  compiler（优化编译器），参见 C++
  hardware（优化硬件），619-621
OR-AND circuit（OR-AND 电路），507-509
OR gate（OR 门），497-499
OR operator（OR 运算符），107-108
Or instruction（Or 指令），162-163
ORA instruction（ORA 指令），198-200
ori instruction，MIPS（ori 指令，MIPS），633
Overflow bit（溢出位），参见 V bit

P

Paging（分页），455-457
Parallelism（并行），612-613
Parameter（参数）
  array（数组参数），291-297
  call-by-reference（传引用调用参数），51-55，273-277，277-281
  call-by-value（传值调用参数），48-51，263-269
Parity bit（奇偶位），470
Parser（分析器），331，381-382
Parsing problem（分析问题），339-340
PCB，397，421-423
Pefect code（完美编码），475
Peterson's algorithm（算法），429-431
Pipelining（流水线），646-655
  control hazard（控制危害），650
  dynamic branch prediction（动态分支预测），650-652
  data forwrding（数据传送），653
  hazard（危害），647

instruction reordering（指令重排序），653-654
RAW data hazard（数据危害），652-653
superscalar machines（超标量机器），653-654
WAR data hazard（WAR 数据危害），654
Pointers（指针），74-76
assignment rule（指针分配规则），75
global（全局指针），300-306
local（局部指针），306-309
zero as a special value（零作为特殊值的指针），78
Pop operation（弹出操作），35
Preset flip-flop line（预设置触发器线），569
Process（进程），397
concurrent（并发进程），419-437
Process control block（进程控制块），参见 PCB
Productions in a grammar（语法的产生式），336
Program（程序），18
analysis vs.design（分析 vs. 设计），21
self-modifying（自我修改），177-179
project operator（project 操作符），25-26
PROM，581
Push operation（压入操作），35

Q

Queue of PCB（PCB 队列），420-423

R

RAID storage system（存储系统），476-483
Level 0:Nonredundant striped（RAID 0 级：非冗余条带化），477-478
Level 1: Mirrored（RAID 1 级：镜像），478
Levels 01 and 10: Striped and mirrored（RAID 01 和 10 级：条带化和镜像），478-480
Level 2:Memory-style ECC（RAID 2 级：内存风格的 ECC），480-481
Level 3:Bit-inteleaved parity（RAID 3 级：位交叉奇偶校验），481-482
Level 4:Block-inteleaved parity（RAID 4 级：块交叉奇偶校验），482-483
Level 5: Block-interleaved distributed parity（RAID 5 级：块交叉分布奇偶校验），482-483
RAM，181-182，577-580
RAW data hazard（RAW 数据危害），参见 Pipelining
Recursion（递归），55-74
cost of（递归的开销），73-74
and mathematical induction（递归的数学推导），60
mutual（相互递归），72-73
Reduced Instruction Set Computer（精简指令集计算机），参见 RISC
Register（寄存器）
base and bound（基址和边界寄存器），450
implementation of（寄存器的实现），574-575
MIPS（MIPS 寄存器），625
Pep/8（Pep/8 寄存器），150-151
two-port bank for Pep/8（Pep/8 的双端口寄存器体），588-590
Register transfer language（寄存器传送语言），108-109
Regular expression（正则表达式），332
Relational operator（关系运算符），41
Resource allocation graph（资源分配图），435-437
RETn instruction（RETn 指令），260-263
RETTR instruction（RETTR 指令），397-398
RISC，489-490，620
ROL operation（ROL 运算），111-112
ROLr instruction（ROLr 指令），224-226
ROM，181-182，581
ROR operation（ROR 运算），111-112
RORr instruction（RORr 指令），224-226
Rotate left operation（循环左移运算），参见 ROL
Rotate right operation（循环右移运算），参见 ROR
Run-time stack（运行时栈），35，238-241

S

Seek time（寻道时间），466
select operator（select 操作符），25-26
Semantics（语义），331
Semaphore（信号量），431-435
Set，bit-mapped representation（集合，位图表示），402
Set interpretation of boolean algebra（布尔代数的

集合解释），500
Sign bit（符号位），97
sll instruction, MIPS（sll 指令，MIPS），631-632
Software（软件），17-22
Space/time tradeoff（空间/时间折中），612-613
combinal circuitsv（组合电路），508，523
Exercise 10.46（练习 10.46），546
Spaghetti code（面条代码），253-256
Specialized hardware unit（特殊的硬件单元）
Exercise 12.15（练习 12.15），660
MIPS，642，644，
Pep/8，615-619
Spin lock（旋转锁），431
SR latch（SR 锁存器），550-552
SRAM，580
Stable state（稳态），参见 State
Stack（栈），35
Stack-indexed addressing（栈变址寻址），参见 Addressing mode
Stack-indexed deferred addressing（栈变址间接寻址），参见 Addressing mode
Stack-relative addressing（栈相对寻址），参见 Addressing mode
Stack-relative deferred addressing（栈相对间接寻址），参见 Addressing mode
State（状态），549-550
State transition diagram（状态转移图），346-347
for SR flip-flop（SR 触发器的状态转移图），558
State transition table（状态转移表），347
STBYTEA instruction（STBYTEA 指令），198-200，606-607
Stop instruction（停止指令），158
STOP instruction（STOP 指令），195-196，216
Store byte instruction（存储字节指令），165-166
Store instruction（存储指令），159-160
STRO instruction（STRO 指令），194，206-207，215-217，396，417-419
Structured programming theorem（结构化编程定律），257
Structure（结构），76-77，310-314
SUBSP instruction（SUBSP 指令），238
Subtract instruction（减法指令），161-162
Subtracter（减法器），参见 Adder/subtracter
Subtraction，binary（减法，二进制），537-539
Superscalar machines（超标量机器），参见 Pipelining
Supervisor call（管理程序调用），420
sw instruction（sw 指令），MIPS，630，645
switch statement（switch 语句），43-44，297-300
Symbol trace tags（符号跟踪标签），参见 Trace tag
Symbol tracer（符号跟踪器），223
Symbols（符号），211-213，217-218
Syntax（语法），331
Syntax tree（语法树），341

T

T flip-flop（T 触发器），563
Thompson，Ken，460
Time out（超时），420
Timing diagram（时序图）
of clocked SR flip-flop（分控 SR 触发器时序图）
of master-slave SR flip-flop（主－从 SR 触发器时序图）
of SR latch（SR 锁存器时序图），552
of von Neumann cycle（冯·诺依曼周期的时序图），603
Token（语言符号），351
Trace tags（跟踪标签），223
format（格式跟踪标签），223，243，288
symbol（符号跟踪标签），243
Transmission time（传送时间），466
Trap handlers（陷阱处理程序）
DECI，408-414
DECO，414-417
NOP，406-407
NOPn，406-407
STRO，417-419
Trap（陷阱），396-419
mechanism（陷阱机制），396-397

Tri-state buffer（三态缓冲区），576，579
Truth table（真值表），491，501
  and boolean expression（真值表和布尔表达式），503-506
Two-level circuit（两级电路），506-508
Two's complement binary representation（补码二进制表示），97-106
  range（补码二进制表示范围），99-101
Type comparibility（类型兼容），221-222

U

Unicode，118
Unimplemented opcode（未实现的操作码），参见 Opcode
Uniprogramming（单道程序设计），448
Unstable state（非稳态），参见 State
USB flash drive（USB 闪存），14

V

V bit（V 位），105-106，150-151，532，533，599
Variable（变量），参见 C++ memory model
  attribute of（变量的属性），36
  global（全局变量），36-39，218-221，263-267，273-277
  local（局部变量），39-41，241-243，267-272，277-281
Virtual memory（虚拟内存），458-459
von Neumann bug（冯·诺依曼瓶颈），174-175
von Neumann execution cycle（冯·诺依曼执行周期）
  MIPS，626
  Pep/8 at level ISA3（ISA3 层的 Pep/8），168-170，205，262-263
  Pep/8 at level LG1（LG1 层的 Pep/8），601-604

W

WAR data hazard（数据危害），参见 Pipelining
while loop（while 循环），44-45，149-250
.WORD pseudo-op（.WORD 伪操作），198-200
Working set（工作集），458

X

XOR gate（XOR 门），498
XOR operator（XOR 运算符），107-108，283-284

Z

Z bit（Z 位），106，150-151，533，599
Zero bit（零位），参见 Z bit
Zero theorem（零元定理），参见 Boolean algebra