



PEARSON

名家经典系列

由在IBM工作50余年的资深计算机专家撰写，Amazon全五星评价，算法领域最有影响力的著作之一
Google公司首席架构师、Jolt大奖得主Joshua Bloch和Emacs合作创始人、C语言畅销书作者Guy Steele倾情推荐

算法的艺术和数学的智慧在本书中得到了完美体现，书中总结了大量高效、优雅和奇妙的算法，并从数学角度剖析了其背后的原理

算法心得

高效算法的奥秘

(原书第2版)

(美) Henry S. Warren, Jr. 著
爱飞翔 译



Hacker's Delight
(Second Edition)

Hacker's Delight

(Second Edition)

“这是第一本宣称能讲解计算机算法隐晦细节的书，而且讲得还真不错。我知道的每一条技巧书里都提到了，而且还讲了好多好多我不知道的。不论是在开发程序库或编译器，还是在极力搜求优雅算法，此书都可谓天赐良册，应放在高德纳所著《计算机程序设计艺术》那套书旁边。本书第一版刊印后的10年间，它对我在Sun和Google的工作大有裨益，而第二版所添加新内容亦令我惊羡不已。”

—— Joshua Bloch

“初看本书书名时，我想，这是教人怎么入侵计算机系统的书吗？不太可能吧。嗯，那就肯定是一本编程小技巧的集锦。看了之后发现，没错，这就是一本编程秘籍，然而却是一本包罗万象的秘籍。第二版新增了两个大主题，并用数十个小技巧丰富了本书内容，其中有个小绝招是如何在不溢出的情况下求两数均值，我写二分查找算法时直接就把这条拿来用了。这真是本令算法爱好者开怀畅读的书啊！”

—— Guy Steele

在本书中，作者给我们带来了一大批极为诱人的知识，其中包括各种节省程序运行时间的技巧、算法与窍门。学习了这些技术，程序员就可写出优雅高效的软件，同时还能洞悉其中原理。这些技术极为实用，而且其问题本身又非常有趣，有时甚至像猜谜解谜一般，需要奇思妙想才行。简而言之，软件开发者看到这些改进程序效率的妙计之后，定然大喜。

本书较第1版增补了大量内容

- 新增了循环冗余校验（CRC）一章，其中讲解了常用的CRC-32校验码
- 新增了纠错码（ECC）一章，其中讲解了汉明码
- 详解了除数为常数的整数除法，增补了仅含移位操作和加法操作的算法
- 不计算商而直接求余数
- 扩充了与种群计数和前导0计数有关的知识
- 数组种群计数
- 执行压缩与扩展操作的新算法
- LRU算法
- 浮点数与整数互化
- 估算浮点数的平方根倒数
- 一系列离散函数图像
- 各章均配有习题与参考答案

PEARSON

www.pearson.com

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259

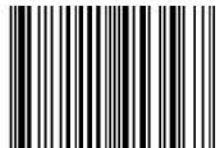
华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn

上架指导：计算机/算法

ISBN 978-7-111-45356-7



9 787111 453567 >

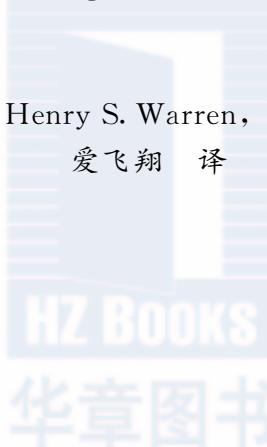
定价：89.00元

名家经典系列

算法心得：高效算法的奥秘 (原书第2版)

Hacker's Delight, Second Edition

(美) Henry S. Warren, Jr. 著
爱飞翔 译



图书在版编目 (CIP) 数据

算法心得：高效算法的奥秘（原书第2版）/（美）沃伦（Warren, Jr., H. S.）著；爱飞翔译。—北京：机械工业出版社，2014.1

（名家经典系列）

书名原文：Hacker's Delight, Second Edition

ISBN 978-7-111-45356-7

I. 算… II. ① 沃… ② 爱… III. 电子计算机—算法理论 IV. TP301.6

中国版本图书馆 CIP 数据核字（2014）第 002804 号

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2013-0212

Authorized translation from the English language edition, entitled *Hacker's Delight, Second Edition*, 9780321842688 by Henry S. Warren, Jr., published by Pearson Education, Inc., Copyright © 2013.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese Simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2014.

本书中文简体字版由 Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括中国台湾地区和中国香港、澳门特别行政区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

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

本书是算法领域最有影响力的著作之一，与大师高德纳所著的《计算机程序设计艺术》共同被誉为所有程序员都应该阅读的计算机著作。它由在 IBM 工作 50 余年的资深计算机专家撰写，Amazon 全五星评价，Google 公司首席架构师、Jolt 大奖得主 Joshua Bloch 和 Emacs 合作创始人、C 语言畅销书作者 Guy Steele 倾情推荐。书中总结了大量高效、优雅和奇妙的算法，并从数学角度剖析了其背后的原理，算法的艺术和数学的智慧在本书中得到了最好的体现。

本书共 18 章。第 1 章是概述；第 2 章介绍了基础知识；第 3~4 章介绍了 2 的幂边界和算术边界；第 5 章讨论了位计数；第 6~7 章讲解了在字组中搜索位串和重排位元与字节；第 8~10 章分别介绍了乘法、整数除法和以除数为常量的整数除法；第 11 章讲解了初等函数；第 12 章介绍了以特殊值为底的数制；第 13~15 章分别讲解了格雷码、循环冗余校验和纠错码；第 16~18 章分别介绍了希尔伯特曲线、浮点数和素数公式。之后是各章习题的参考答案。附录分别介绍了计算机算术运算表、牛顿法和各种离散函数图像。

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：关 敏

印刷

2014 年 3 月第 1 版第 1 次印刷

186mm×240mm·27.25 印张

标准书号：ISBN 978-7-111-45356-7

定 价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

译 者 序

写代码总会遇到难题，时而苦于乘法操作频繁溢出，时而苦于开方算法太过笨拙，于是，程序员之间口耳相传的那些代码秘籍，这些时候就该大显身手了。有些小程序，仅两三行代码即能解决平常数十行代码方能实现的功能；还有些小程序，只用0x24924925 这般神奇的数字，即能成倍提升运算速度。读者若对此感兴趣，则本书定能令你开怀畅读。

作者从事计算机研发工作数十年，他将期间所得之大量技巧融于书中。本书不但讲授算法技巧，而且还会剖析背后的数学原理，令你在学会某个奇妙算法后，可举一反三，推出很多类似技巧，以运用于不同场合。

在研究这些高效而优雅的算法时，作者还会如数家珍地列出许多变体，并旁征博引地讲述可以解决同一问题的其他思路，铺陈完毕后，更会将各自优劣娓娓道来。实际应用中，经常需要权衡各算法之轻重，嵌入式开发、硬件编程、图形渲染、游戏智能等领域尤其如此，若是平素能像作者这样勤于总结、善于对比，那么在需要用到相关技巧时必能信手拈来，左右逢源。

从培养兴趣、锻炼思维、付诸实践三个角度观之，本书皆为精彩而思辨的智慧书。既可静心品读代码之诗意，又能细致体味数学之美感，何其乐哉！

作者乃业界翘楚，学识渊博而思维开阔，文中部分词句与日常用语及数学、计算机等领域一般用法不甚相同，故译文或加注释或添引号，以强调其特殊含义。

翻译过程中，得到机械工业出版社华章公司诸君勉励，于此深表谢意。

本书主要由爱飞翔翻译，舒亚林、张军、王鹏亦参与部分翻译工作。小弟乐意与各位朋友通过个人网站（www.agilemobidev.com）及电子邮件（eastarstormlee@gmail.com）探讨算法问题。由于时间仓促，水平有限，错误与疏漏在所难免，敬请读者不吝赐教。

爱飞翔

2014年2月

序（第1版序）

三十年前的一个暑假，我在麻省理工学院的 Project MAC 实验室^①找到了首份工作。当时很喜欢操作那里的 DEC PDP-10 计算机^②。这台计算机有丰富且易于操控的指令集，可以做位测试和位屏蔽，并操作位段^③与整数，所以和其他计算机相比，拿它写汇编语言代码绝对要更顺手些。尽管已经停产多年，仍有一批狂热的粉丝还在玩 PDP-10，还有一些人拿家用电脑模拟 PDP-10 指令集，这样就可以执行以前的整个 PDP-10 操作系统及其附属软件了。这帮人甚至还在该平台上开发新的程序，比如说现在还有些网站用模拟的 PDP-10 平台来提供网页服务。（拜托，大家别笑，与那些爱开老爷车的人相比，这种怀旧方式也不算什么。）

1972 年夏天，还有一件事让我开心，那就是读到了一本全新的 MIT 研究备忘录：HAKMEM。这部秘籍收录了从电路到数论的各种技术细节^④，而最令我着迷的内容还是其中各项精妙的编程小技巧。通常每一条编程技巧都会讲一个貌似符合编程规范但是看上去却十分怪异的操作。这些整数或位串操作（bit string，例如计算某个字^⑤中值为 1 的

-
- ① “数学与计算研究计划”（Project on Mathematics and Computation）一词的缩写，是 1963 年设立于麻省理工学院（Massachusetts Institute of Technology, MIT）的一个研究项目，为当前“MIT 计算机科学与人工智能实验室”（MIT Computer Science and Artificial Intelligence Laboratory）的前身。详情参见：https://en.wikipedia.org/wiki/Project_MAC#Project_MAC。——译者注
 - ② DEC 公司（Digital Equipment Corporation）于 20 世纪 60 至 80 年代推出的一系列大型计算机。详情参见：<https://en.wikipedia.org/wiki/PDP-10>。——译者注
 - ③ field，在这里指 bit field，也叫“位域”，是一种数据结构，可将各个字段用若干个位元的形式紧密存储，以节省空间。详情参见：<https://zh.wikipedia.org/wiki/位段>。——译者注
 - ④ “HAKMEM”是“hacks memo”的简称，为什么这样写呢？因为 PDP-10 平台的字（word）长度为 36 位，而每个字符占 6 个二进制位，这样，一个字就能容纳 6 个字符，所以很多 PDP-10 程序员用的名字都是 6 个字符，我们这帮人一看到 6 字符的缩写，立马就能还原出它的原意。所以，“HAKMEM”这个怪名字在那个时代是合理的，至少对程序员来说是这样。
 - ⑤ word，在计算机领域里指由一定数量的二进制位（bit，比特）所构成的数据结构。为了与日常汉语中的“字”区别开，译文多以“字组”称之。详情参见：[https://zh.wikipedia.org/wiki/字_\(计算机\)](https://zh.wikipedia.org/wiki/字_(计算机))。此外，为了与十进制的“数位”（digit）和常用的“位置”等词相区分，译文将酌情以“位元”一词对应 bit。——译者注

二进制位共有多少个)，本来用一个较长但是约定俗成的机器指令序列或是循环就能轻松实现，然而秘籍中却用了一种极为巧妙的办法来完成：它只用三四个甚至两个精心选择的指令就够了。这些指令的意思很隐晦，需要有人给你解释，或是自己仔细研究才能搞清楚。品尝这份编程大餐，与吃花生豆或夹心软糖那种小零食一样，也令人欲罢不能。这些技巧内涵丰富，充满了深邃的智慧，体现了程序的美感，甚至让人读出了诗意图。

我心里当时就在想：“这种技巧不会只有这么几条吧？”我把这些年来在各种场合发现的编程心得总结了一下，果然数量非常多。所以我又想，“是不是该有一本专著来谈谈这个问题呢？”

看到 Hank Warren 先生的原稿时，我实在太高兴了：他将这些编程小技巧系统地收集起来，按主题分类，并清晰地讲解其意义。虽说有些技巧是用机器指令描述的，但是这本书可不是只写给汇编语言程序员看的。本书主题是探讨计算机中整数与位串的基本结构关系，以及如何才能更为高效地操作它们。汇编语言中能用的技巧，放在 C 或 Java 语言里照样适用。

很多算法和数据结构的书都在讲排序和搜索这些复杂的技术，告诉你如何维护哈希表和二叉树，怎样操作记录和指针等，然而它们都没有讲到二进制位及位元数组这种微型数据结构的功用。单用二进制加减法和位操作就能实现很多强大的功能。由于进位链 (carry chain) 机制，改变一个位元的值有可能会影响到它左方的所有二进制位。某些利用此机制的二进制加法技巧不为人熟知，然而它们却是操作数据的利器。

的确该有一本书来总结这些技巧了。你手中捧着的这本就是，而且写得非常好。要想优化编译器或编写高效代码，那非读它不可。也许这些技巧不是每天都能用到，但在遇到困难时，还是会派上用场的。有时我们要遍历一个字中的位元，有时要执行某些不太好实现的整数操作，有时为了让运行速度加倍，还要优化内循环[⊖] 中那些棘手的整数或位元运算——如果真碰到上述情况，那就得求助此书啦。当然了，有时也不必想那么多，直接翻开来看，就必能体会到个中乐趣。

Guy L. Steele, Jr.
2002 年 4 月于马萨诸塞州伯灵顿

[⊖] inner loop，指两重循环中内部的那一层，通常优化程序时应首先考虑减少内层循环体的执行时间。详情参见：https://en.wikipedia.org/wiki/Inner_loop。——译者注

前　　言

程序员创造力第一定律：软件维护成本与程序员创造力的平方成正比。

——Robert D. Bliss, 1992 年

本书收集了笔者多年来总结的一些编程小技巧，其中大部分都必须运行在以二进制补码来表示整数的电脑^①上。尽管本书假设寄存器长度是 32 位，但在寄存器长度不是 32 位的电脑中，这些技巧基本上仍能适用。

本书不打算讲高深的排序算法或是编译器的优化技术等大话题，而是着重来谈涉及单个计算机字组或指令的小技巧，比如怎样统计字组中值为 1 的位元数。此类技巧通常需要混用算术与逻辑指令。

我们还需假定整数溢出^②中断已经屏蔽，这样的话，就算整数运算溢出了，也不会出问题。C、Fortran、Java 等程序都是如此，但使用 Pascal 与 Ada 语言的程序员一定要注意这一点！

书中以通俗的语言来讲述各技巧，只有算法含义不明显时才会给出证明，有时甚至略去证明。算法中将会出现计算机算术指令、“地板”函数^③、算术与逻辑操作混搭等内容，它们要证明起来通常比较困难，而且也不易表述。

笔者把很多算法都写成了程序代码，在电脑上执行并验证，以减少笔误及疏忽。用实际存在的编程语言来描述算法，就有这个好处；然而即便如此，每一种计算机语言也还是各有其缺陷。我使用很多人能看懂的 C 代码来描述高级语言部分，因为它可以直接

-
- ① 在本书语境中，电脑、计算机（computer）、机器（machine）、处理器（processor）等词常常都指代中央处理器（CPU），故译文在不致混淆的情况下，将其视为互相通用的概念。——译者注
 - ② integer overflow，指算术运算时由于结果过大或过小而超出存储范围的情形。详情参见：https://en.wikipedia.org/wiki/Integer_overflow。——译者注
 - ③ floor function，又叫向下取整函数或最低值函数，返回不大于自变量的最大整数，也常称为“高斯函数”。详情参见：<https://zh.wikipedia.org/wiki/地板函数>。——译者注

把整数与位串操作写在一起，而且很多 C 语言编译器都能产生高质量的目标代码^①。

书中偶尔也会用到机器语言，它们都以“三地址格式”^②出现，这样读起来更方便。其中的汇编语言是笔者参照当下的 RISC 指令集^③虚构出来的。

大家应该尽量编写没有分支的代码 (branch-free code)，因为分支代码会拖慢很多电脑的指令获取速度，并阻止 CPU 并发执行。此外，它还会妨碍一些编译器优化措施^④，如指令调度^⑤、重复子表达式消除^⑥、寄存器分配^⑦等。也就是说，编译器优化大段的简单代码要比优化很多小代码块的效果更好。

此外，编码时还应该多用数值较小的常量^⑧，多与 0 比较（少与非 0 值比较），尽量写出易于并发执行的指令序列^⑨。如果改写书里很多代码，让它们从内存中查表，其结果会更精确，然而笔者通常不提这种做法。因为与算术指令相比，从内存中加载数据反而更耗时。虽说查表法的确很实用，但是一般来说都比较乏味。当然还有些特例不属于上述情况。

最后要说的是，本书书名^⑩中的“hacker”用的是本意，也就是指痴迷于计算机的人。这种人喜欢拿电脑开发点新玩意儿，或是用新潮而有创意的办法来实现原有的功能。

^① object code，又叫“目的码”，是编译器处理源代码后的产物，一般由机器码或近于机器语言的代码组成。存放此类代码的文件叫目标文件 (object code)，也叫二进制文件。详情参见：<https://zh.wikipedia.org/wiki/目标代码>。——译者注

^② three-address format，即 three address code，缩写为 TAC 或 3AC，又叫“三位址码”、“三地址码”。它把通常的算式“运算结果 = 操作数 1 操作符 操作数 2”写成“操作符 操作数 1，操作数 2，运算结果”的形式，因其中牵涉 3 个变量（即操作数 1、操作数 2、运算结果）而得名。例如 $z = x + y$ 用 TAC 格式来写就是 add x, y, z。详情参见：https://en.wikipedia.org/wiki/Three_address_code。——译者注

^③ RISC，Reduced Instruction Set Computing 的缩写，是一种设计 CPU 的模式，该设计思路精简了指令数目及定址方式，使 CPU 的制作更为容易，也有助于提升其并行能力与执行效率。使用此种指令集的 CPU 目前主要应用于智能手机和平板电脑等移动设备上，与之相对的是 x86 等处理器所用的“复杂指令集”(CISC，C 表示 Complex)。详情参见：<https://zh.wikipedia.org/wiki/精简指令集>。——译者注

^④ 常见的编译器优化技法请参考：https://en.wikipedia.org/wiki/Category:_Compiler_optimizations。——译者注

^⑤ instruction scheduling，是通过指令流水线 (instruction pipeline) 技术提升指令并发执行效果的优化手法。详情参见：https://en.wikipedia.org/wiki/Instruction_scheduling。——译者注

^⑥ commoning，即 common subexpression elimination (CSE)，是通过消减重复的表达式来提升执行速度的优化技术。详情参见：https://en.wikipedia.org/wiki/Common_subexpression_elimination。——译者注

^⑦ register allocation，该技术让众多变量共用一个 CPU 寄存器，以优化执行速度。详情参见：https://en.wikipedia.org/wiki/Register_allocation。——译者注

^⑧ immediate value，也叫“立即值”。——译者注

^⑨ instruction-level parallelism，缩写为 ILP，意为指令级并行度，用于度量指令序列的并发执行程度。例如某段指令在非并发方式下需要 3 个时间单位来执行，而并发执行只需 2 个时间单位，那么其 ILP 就是 3/2。详情参见：https://en.wikipedia.org/wiki/Instruction-level_parallelism。书中还用该词表示 CPU 并发执行指令的能力。——译者注

^⑩ 本书英文书名为 Hacker's Delight。——译者注

他们都挺会用电脑，但却很可能不是一名专职电脑程序员或设计师，这些电脑极客^①开发出来的东西，有些有用，有些只是玩玩而已。说到此类纯属玩乐的戏作，让我想起很多资深编程票友都写过一段小程序，它可以把原样打印出来^②。这才是我使用“hacker”一词的初衷，在书里可别想找到教你怎样入侵他人电脑的把戏哦。

致谢

首先要感谢 Bruce Shriver 与 Dennis Allison 两位先生鼓励我写作本书，然后还要对 IBM 诸位同仁致意，参考文献中也提到了其中一些同事的名字。尤其感谢 Martin E. Hopkins 先生，他可谓 IBM 的“活编译器”(Mr. Compiler)。Hopkins 先生在编码工作中一丝不苟，认真优化每一个循环——这种敬业精神深深感染了我。仰赖 Addison-Wesley 各位评审，本书内容有了极大改观。他们的姓名笔者大多不了解，我唯一记住的是大名鼎鼎的 Guy L. Steele, Jr.。在 50 页书评中，Guy 补充了一些我当时没有谈到的新问题，如位元的打乱与复原，“绵羊与山羊分离操作”(分羊法，sheep and goats operation)，等等，而且他提出的一些算法也比我原来用的更高明。Guy 是个非常仔细的人，比方说，我曾把十六进制数 AAAAAAAA 的质因子分解式错写为 $2 \times 3 \times 17 \times 257 \times 65\ 537$ ，而他发现其中的 3 应该是 5。此外，他还提了一些旨在改善写作风格的建议，并且从不回避书里的细节问题。读者若发现文中有疑似他人捉刀之处^③，那恐怕得归功于 Guy 了。

H. S. Warren, Jr.

2012 年 6 月于纽约州约克镇

本书英文版相关资料请查阅
www.HackersDelight.org

^① 本书语境中的“hacker”可理解成“电脑极客”、“程序玩家”、“编程票友”、“编程达人”、“技术发烧友”等意思，以便与容易引发歧义的“骇客”、“黑客”等通俗叫法区分开。本来 hacker 指计算机技术狂爱好者，而 cracker 指破坏计算机安全的人，但后者的词义逐渐侵蚀了前者，导致 cracker、hacker 及“骇客、黑客、怪客、剑客”等译名全都成了“计算机安全破坏者”，令 hacker 一词丧失了原意。详情参见：<https://zh.wikipedia.org/wiki/骇客>。——译者注

^② 比方说，下面这段用 C 语言写的程序代码：

```
main () {char * p="main () {char * p=%c%s%c; (void) printf (p, 34, p, 34, 10);}%c"; (void) printf (p, 34, p, 34, 10);}
```

^③ 原文为 parallel prefix，意为“并行前置式算法”、“平行前缀算法”，是一套分组归并形式的速算流程。详情参见：https://www.wikipedia.org/wiki/Prefix_sum。作者在此处以之比喻他与 Guy 四手联弹的佳话。——译者注

目 录

译者序	2.10	符号传递函数	22
序 (第 1 版序)	2.11	将值为 0 的位段解码为 2 的 n 次方	22
前言	2.12	比较谓词	23
第 1 章 概述 1	2.12.1	利用进位标志求比较谓词	26
1.1 记法 1	2.12.2	计算机如何设置比较谓词	27
1.2 指令集与执行时间模型 5	2.13	溢出检测	28
1.3 习题 10	2.13.1	带符号的加减法 ...	28
第 2 章 基础知识 11	2.13.2	计算机执行带符号数的加减法时如何设置溢出标志	31
2.1 操作最右边的位元 11	2.13.3	无符号数的加减法	31
2.1.1 德摩根定律的推论 ... 12	2.13.4	乘法	32
2.1.2 从右至左的可计算性测试	2.13.5	除法	34
2.1.3 位操作的新式用法 ... 14	2.14	加法、减法与乘法的特征码	36
2.2 结合逻辑操作的加减运算 16	2.15	循环移位	37
2.3 逻辑与算术表达式中的不等式	2.16	双字长加减法	38
2.3.1 不等式	2.17	双字长移位	38
2.4 绝对值函数 18	2.18	多字节加减法与求绝对值 ...	39
2.5 两数平均值 19	2.19	doz、max、min 函数	41
2.6 符号扩展 20	2.20	互换寄存器中的值	44
2.7 用无符号右移模拟带符号右移操作			
2.8 符号函数 21			
2.9 三值比较函数 21			

2.20.1 交换寄存器中相应的位段	45	的位元数	82
2.20.2 交换同一寄存器内的两个位段	46	5.1.4 应用	86
2.20.3 有条件的交换	47	5.2 奇偶性	87
2.21 在两个或两个以上的值之间切换	47	5.2.1 计算字组的奇偶性	87
2.22 布尔函数分解公式	50	5.2.2 将表示奇偶性的位元添加到 7 位量中	89
2.23 实现 16 种二元布尔操作	51	5.2.3 应用	90
2.24 习题	54	5.3 前导 0 计数	90
第 3 章 2 的幂边界	56	5.3.1 浮点数算法	94
3.1 将数值上调/下调为 2 的已知次幂的倍数	56	5.3.2 比较两个字组前导 0 的个数	96
3.2 调整到上一个/下一个 2 的幂	57	5.3.3 与对数函数的关系	96
3.2.1 向下舍入	58	5.3.4 应用	97
3.2.2 向上舍入	59	5.4 后缀 0 计数	97
3.3 判断取值范围是否跨越了 2 的幂边界	59	5.5 习题	105
3.4 习题	61		
第 4 章 算术边界	63	第 6 章 在字组中搜索位串	106
4.1 检测整数边界	63	6.1 寻找首个值为 0 的字节	106
4.2 通过加减法传播边界	65	6.1.1 0 值字节位置函数的一些简单推广	110
4.3 通过逻辑操作传播边界	69	6.1.2 搜索给定范围内	110
4.4 习题	73	6.2 寻找首个给定长度的全 1 位串	111
第 5 章 位计数	74	6.3 寻找最长全 1 位串	114
5.1 统计值为“1”的位元数	74	6.4 寻找最短全 1 位串	115
5.1.1 两个字组种群计数的和与差	80	6.5 习题	115
5.1.2 比较两个字组的种群计数	80		
5.1.3 统计数组中值为“1”		第 7 章 重排位元与字节	117
		7.1 反转位元与字节	117
		7.1.1 位元反转算法的推广	122
		7.1.2 奇特的位元反转算法	122
		7.1.3 递增反转后的整数	124

7.2	乱序排列位元	126	9.4.1	用硬件实现移位并相 减算法	172
7.3	转置位矩阵	128	9.4.2	用短除法实现无符号 长除法	174
7.4	压缩算法 (广义提取 算法)	136	9.5	用长除法实现双字除法	176
	7.4.1 用“插入”、“提取” 指令实现压缩操作	140	9.5.1 无符号双字除法	176	
	7.4.2 向左压缩	141	9.5.2 带符号双字除法	179	
7.5	展开算法 (广义插入 算法)	141	9.6	习题	180
7.6	压缩与展开操作的硬件 算法	142			
	7.6.1 压缩	142			
	7.6.2 展开	144			
7.7	通用置换算法及分羊操作	145	10.1	除数为 2 的已知次幂的 带符号除法	181
7.8	重排与下标变换	149	10.2	求与 2 的已知次幂相除的 带符号余数	182
7.9	LRU 算法	150	10.3	在除数不是 2 的幂时求 带符号除法及余数	183
7.10	习题	153	10.3.1	除以 3	183
			10.3.2	除以 5	184
			10.3.3	除以 7	185
第 8 章 乘法	154	10.4	除数大于等于 2 的带符号 除法	185	
8.1	多字乘法	154	10.4.1	算法	187
8.2	64 位积的高权重部分	156	10.4.2	算法可行性证明	187
8.3	无符号与带符号的高权重积 互化	157	10.4.3	证明乘积正确	188
8.4	与常数相乘	157	10.5	除数小于等于 -2 的带符号 除法	191
8.5	习题	160	10.6	将除法算法集成至编译 器中	193
第 9 章 整数除法	162	10.7	其他主题	196	
9.1	预备知识	162	10.7.1 唯一性	196	
9.2	多字除法	165	10.7.2 可生成最佳程序 代码的除数	197	
9.3	用带符号除法计算无符号短 除法	169	10.8	无符号除法	199
	9.3.1 用带符号长除法计算 无符号短除法	169	10.8.1 除数为 3 的无符号		
	9.3.2 用带符号短除法计算 无符号短除法	169			
9.4	无符号长除法	171			

除法	199	10.17.2 除数大于等于 2 的带符号除法 ...	219
10.8.2 除数为 7 的无符号 除法	200	10.18 不使用 Multiply High 指令 的除法算法	220
10.9 除数大于等于 1 的无符号 除法	201	10.18.1 无符号除法	221
10.9.1 无符号版算法	202	10.18.2 带符号除法	226
10.9.2 算法可行性证明 ...	202	10.19 合计各数位求余数	229
10.9.3 证明无符号版算法的 乘积正确	203	10.19.1 求无符号除法的 余数	229
10.10 将无符号除法算法集成至 编译器中	203	10.19.2 求带符号除法的 余数	232
10.11 与无符号除法相关的其他 话题	205	10.20 用乘法及右移位求 余数	234
10.11.1 可生成最佳无符号 除法代码的 除数	205	10.20.1 求无符号除法的 余数	234
10.11.2 带符号乘法与无 符号乘法互化 ...	206	10.20.2 求带符号除法的 余数	237
10.11.3 更简单的无符号 除法生成算法 ...	206	10.21 将普通除法化为精确 除法	239
10.12 余数非负式除法与向下取 整式除法的适用性	207	10.22 计时测试	240
10.13 类似算法	208	10.23 用电路计算除数为 3 的 除法	241
10.14 神奇数字示例	209	10.24 习题	242
10.15 用 Python 语言编写的 简单代码	210	第 11 章 初等函数	243
10.16 除数为常量的精确除法	211	11.1 整数平方根	243
10.16.1 用欧几里得算法计 算乘法逆元素 ...	212	11.1.1 用牛顿法开 平方	243
10.16.2 用牛顿法计算乘法 逆元素	215	11.1.2 二分查找	246
10.16.3 乘法逆元素 示例	217	11.1.3 硬件算法	247
10.17 检测除以常数后是否 余 0	217	11.2 整数立方根	249
10.17.1 无符号除法 ...	218	11.3 求整数幂	250
		11.3.1 用 n 的二进制分解式 计算 x^n	250
		11.3.2 用 Fortran 语言	

计算 2^n 251	15.2.2 校验位个数的最 小值 290
11.4 整数对数 252	15.2.3 小结 290
11.4.1 以 2 为底的整数 对数 253	15.3 适用于 32 位信息的软件 SEC-DED 算法 292
11.4.2 以 10 为底的整数 对数 253	15.4 广义错误修正 297
11.5 习题 257	15.4.1 汉明距离 298
第 12 章 以特殊值为底的数制 ... 258	15.4.2 编码论的主要 问题 299
12.1 以 -2 为底的数制 258	15.4.3 n 维球面 301
12.2 以 $-1+i$ 为底的数制 264	15.5 习题 305
12.3 以其他数为底的数制 266	
12.4 最高效的底是什么 267	第 16 章 希尔伯特曲线 307
12.5 习题 267	16.1 生成希尔伯特曲线的递归 算法 308
第 13 章 格雷码 269	16.2 根据希尔伯特曲线上从起点 到某点的途经距离求其坐标 ... 311
13.1 简介 269	16.3 根据希尔伯特曲线上的坐标 求从起点到某点的途经距离 ... 317
13.2 递增格雷码整数 271	16.4 递增希尔伯特曲线上点的 坐标 319
13.3 负二进制格雷码 272	16.5 非递归的曲线生成算法 321
13.4 格雷码简史及应用 273	16.6 其他空间填充曲线 321
13.5 习题 275	16.7 应用 322
第 14 章 循环冗余校验 276	16.8 习题 324
14.1 简介 276	
14.2 理论 277	第 17 章 浮点数 325
14.3 实现 279	17.1 IEEE 格式 325
14.3.1 硬件实现 281	17.2 整数与浮点数互化 327
14.3.2 软件实现 283	17.3 使用整数操作比较浮点数 大小 331
14.4 习题 285	17.4 估算平方根倒数 332
第 15 章 纠错码 286	17.5 前导数位的分布 334
15.1 简介 286	17.6 杂项数值表 336
15.2 汉明码 287	17.7 习题 338
15.2.1 SEC-DED 码 289	

第 18 章 素数公式	339	18.5 习题	350
18.1 简介	339		
18.2 Willans 公式	341	参考答案	351
18.2.1 Willans 第二 公式	342	附录 A 4 位计算机算术运算表	395
18.2.2 Willans 第三 公式	342	附录 B 牛顿法	400
18.2.3 Willans 第四 公式	343	附录 C 各种离散函数图像	402
18.3 Wormell 公式	344		
18.4 用公式来描述其他难解的 函数	345	参考文献	412



第1章

概述

1.1 记法

本书既不是普通的数学算式教程，也不是单纯的电脑程序算法手册，它讲的是“计算机算术”（computer arithmetic），参与运算的数是长度固定的位串与位元数组[⊕]。计算机算术中的表达式与普通的数学表达式近似，不同之处在于其中的变量指代的是CPU寄存器中的内容，而且计算机算术表达式的值是一串不具备特定符号性的位元。在此类表达式中，操作符可能会以不同方式解读其操作数，例如“比较操作符”（comparison operator）有时会将其操作数当成有正负号的二进制整数，有时却把它视为无符号的二进制整数。为免混淆，本书所列计算机算式以不同的符号来区分上述两种情况。

计算机算式与数学算式的主要差别在于：无论是加减法还是乘法，计算机算式的结果总是要跟 2 的 n 次方取模， n 是指当前机器的字长[⊖]。此外，计算机算式的运算种类也远多于数学算式：除了基本的四则运算外，还有逻辑与、异或、比较、左移，等等。

如未特别指明，则默认字长为32位，且带符号的整数均以“2补码”[⊕]形式表示。

-
- ⊕ bit vector，即bit array，也写作bitmap、bitset、bit string等，是由若干位元排列而成的数组，又叫“位向量”、“位矢量”等，详情参见：https://en.wikipedia.org/wiki/Bit_array。译文在不致混淆时，均以“位元数组”称之。——译者注
 - ⊖ word size，就是字组中所含的位元个数，也称word length、word width。在不致混淆的情况下，“字长”、“字宽”均指这一概念。——译者注
 - ⊕ two's complement，也叫“2的补码”、“二补数”，是一种用二进制表示带符号数字的方式。整数和零的补码表示法与其二进制写法相同，只是左方要补足0，而要表示负数，则需先将其绝对值按位取反，也就是求绝对值的一补数（也称反码），然后再加1。例如用8位二进制表示数字时，5可以表示为0000 0101，而要表示-5，则需先求其绝对值5，写成二进制形式0000 0101，然后对其按位取反得到1111 1010，再加1。最终的结果1111 1011就是-5的补码表示。详情参见：<https://zh.wikipedia.org/wiki/补码>。带符号整数与无符号整数的英文分别为signed integer与unsigned integer，其中“带符号”与“无符号”是计算机处理二进制数的两种不同方式。前者会根据最高有效位来判定数字的正负，0为非负，1为负，而后者则一律将其视为非负数。以上面的1111 1011来说，若视为带符号整数，则其值为-5（因最高位为1，故是负数。先将其按位取反得0000 0100，再加1得0000 0101，即十进制的5，再添上负号得-5），若视为无符号整数，则其值为251（最高位的1不再表示负数，而当做128来算）。详情参见：<https://zh.wikipedia.org/wiki/有符号数处理>。——译者注

计算机算式与数学算式的书写方式相同，只是其中指代 CPU 寄存器中内容的变量会以粗体标出。为了和位元数组运算的通则一致，我们把电脑中的字组也视为由一串位元构成的数组。若某常量表示 CPU 寄存器中的值，那么它也会以粗体字出现。（这种情况下在位元数组运算中找不到对应物，如果要在位元数组算式里书写常量，只能逐个列出其中的每个位元。）然而常量如果作为 shift 等指令的操作数，则不加粗。

对于“+”这样的操作符，若其操作数为粗体，则表明执行的是计算机加法，也就是“向量加法”，否则意味着执行的是纯数字加法。如果操作数未加粗，则其对应的操作符就是纯数学运算中的含义。要是我们想拿原来做计算机运算的粗体 x 值做数学运算的话，那就会把它写成不加粗的 x ，其符号性应该能够从上下文中推出。假如 $x = \mathbf{0x8000 0000}$, $y = \mathbf{0x8000 0000}$ ，那么在做带符号的整数运算时， $x = y = -2^{31}$, $x + y = -2^{32}$ ，而 $x + y = \mathbf{0}^{\ominus}$ 。此处的 $\mathbf{0x8000 0000}$ 是用十六进制表示的位串，它最左边的位元是“1”，后面跟着 31 个“0”。

位元的序号从右侧算起，最右方的位元（也就是最低有效位）叫做 0 号位元。术语“位”、“半字节”、“字节”、“半字组”、“字组”、“双字”^① 所对应的位元数量分别是 1、4、8、16、32、64。

简短的代码段用的都是计算机算式，并以左箭头表示赋值操作，偶尔还会用 if 语句。在这种情况下，它只是以一种与电脑平台无关的方式编写汇编语言代码罢了。

过长或过于复杂的计算机算式则用 C 语言来写，其代码遵循 ISO 1999 标准^②。

完整描述 C 语言不是本书该做的事，不过书中用到的大部分 C 语言基本表达式[H & S] 都总结到表 1.1 中了，用程序语言编程但是不熟悉 C 语言的读者应该看看。表中也列出了笔者在计算机算式中用到的对应操作符，它们按照优先级从高到低的顺序排列（高者先算），在优先级这一列中，L 表示左结合，比如乘号就是左结合的运算符： $a \cdot b \cdot c = (a \cdot b) \cdot c$ ，而 R 则表示右结合^③。本书计算机算式里的运算符，其优先级与结合性同 C 语言一致。

表 1.1 C 语言与计算机算术表达式对照表

优先级	C	计算机算式	含义
	0x...	0x..., 0b...	十六进制常数，二进制常数

① 不加粗的 $x + y$ 是普通的数学运算，故结果为 -2^{32} ，而粗体的 $x + y$ 则是按照计算机算术规则做二进制加法，其结果为 1 00000000 00000000 00000000 00000000，也就是 1 后面 32 个零。宽度为 32 的字组无法容纳这个 33 位元的数，故而最高有效位的“1”会被舍去，留下 32 个 0，因此最终的运算结果就是 0。——译者注

② 对应英文写法分别为：bit、nibble、byte、halfword、word、doubleword。——译者注

③ 该标准的正式名称是 ISO/IEC 9899：1999，俗称 C99，详情参见：https://zh.wikipedia.org/wiki/C_语言#C99。C 语言的最新标准是 2011 年发布的 ISO/IEC 9899：2011，俗称 C11。——译者注

④ 左结合与右结合的英文分别是 left-associative 与 right-associative，结合性指的是遇到两个相邻的同优先级运算符时，要从左先算还是从右先算。详情参见：https://en.wikipedia.org/wiki/Operator_associativity。——译者注

(续)

优先级	C	计算机算式	含义
16	a [k]		数组 a 中索引为 k 的元素
16		x_0, x_1, \dots	若干个变量或位元 (具体含义会在文中说明)
16	f (x, ...)	$f(x, \dots)$	求函数值
16		abs (x)	求绝对值 (例外: abs (-2 ³¹) = -2 ³¹)
16		nabs (x)	绝对值对应的负数
15	x ++, x --		后置自增与自减 ^①
14	++ x, -- x		前置自增与自减 ^②
14	(数据类型名) x		类型转换
14R		x^k	x 的 k 次方
14	~x	$\neg x, \bar{x}$	按位取反 (也就是求 x 的反码) ^③
14	! x		逻辑非 (若 x 是 0 则结果为 1, 若 x 非 0 则结果为 0)
14	- x	$-x$	取相反数
13L	x * y	$x * y$	乘法, 根据字组长度裁剪其运算结果 ^④
13L	x/y	$x \div y$	带符号整数的除法
13L	x/y	$x \frac{u}{\div} y$	无符号整数的除法
13L	x % y	rem (x, y)	已知 x 与 y 为带符号的数, 求 x ÷ y 的余数。结果有可能是负数 ^⑤
13L	x % y	remu (x, y)	已知 x 与 y 为无符号的数, 求 x ÷ y 的余数
		mod (x, y)	已知 x 与 y 为带符号的数, 求 x 除以 y 的余数, 并将结果调整到 [0, abs(y)-1] 之间
12L	x + y, x - y	$x + y, x - y$	加法与减法
11L	x<=y, x>=y	$x \ll y, x \frac{u}{\gg} y$	左移位, 右移位 (以 0 填补空缺位元, 又名逻辑移位, logical shift)
11L	x>>y	$x \frac{s}{\gg} y$	右移位 (根据 x 的符号来填补空缺位元, 又名算术移位或数学移位)
11L		$x \frac{rot}{\ll} y, x \frac{mt}{\gg} y$	循环左移, 循环右移
10L	x<y, x<=y, x>y, x>=y	$x < y, x \leq y,$ $x > y, x \geq y$	带符号数的关系比较表达式
10L	x<y, x<=y, x>y, x>=y	$x \frac{u}{<} y, x \frac{u}{\leq} y,$ $x \frac{u}{>} y, x \frac{u}{\geq} y$	无符号数的关系比较表达式
9L	x == y, x != y	$x = y, x \neq y$	比较两数是否相等, 比较两数是否不等
8L	x&y	$x \& y$	按位与
7L	x^y	$x \oplus y$	按位异或
7L		$x \equiv y$	按位等值, 结果与 $\neg(x \oplus y)$ 相同
6L	x y	$x \mid y$	按位或
5L	x && y	$x \dot{\&} y$	条件与 (x 与 y 均不为 0 时, 结果是 1, 否则是 0)
4L	x y	$x \dot{ } y$	条件或 (x 与 y 均为 0 时, 结果是 0, 否则是 1)

(续)

优先级	C	计算机算式	含义
3L		$x \mid\mid y$	连接
2R	$x = y$	$x \leftarrow y$	赋值

- ① postincrement 与 postdecrement，意为先求表达式的值，然后再对其增减。假设 x 是 5，那么 $x++$ 的值还是 5，求完值之后， x 才变成 6。——译者注
- ② preincrement 与 predecrement，意为先对其增减，然后再求表达式的值。假设 x 是 5，那么先将 x 加 1，然后再判定 $++x$ 的值为 6。——译者注
- ③ one's-complement，又叫一补数，是将二进制数中的每个位元反转之后得到的数。例如 10010 的一补数为 01101。详情参见：<https://zh.wikipedia.org/wiki/一补数>。——译者注
- ④ modulo word size，意思是如果运算结果位数太多，超过了字组长度，那么会丢弃超出的部分。此概念直译为“与字长取模”或“模字长”，为了不使人误认为是和 32（字组长度值）取模，故译文用了意译，下文皆同。——译者注
- ⑤ 在 C99 标准中，模除的结果与被除数的正负号相同，详情参见：https://en.wikipedia.org/wiki/Modulo_operation。——译者注

除了表 1.1 列出的写法之外，书中也将借用一些布尔代数运算符与标准数学符号，在用到它们时会给出解释。

计算机算术中除了“abs”、“rem”之外，还会用到其他一些函数，等讲到那些函数时再给出其定义。

C 语言中， $x < y < z$ 这个表达式的意思是：先判断 $x < y$ 是否成立，若成立，则这个子表达式的值就是 1，否则为 0，然后，再将刚才的结果与 z 比较^①。而在本书所讲的计算机算术中，该表达式的意思就是想判断 $x < y$ 与 $y < z$ 是否同时成立而已。

C 语言中有三种循环控制语句：while、do、for。while 语句的语法^②是：

while(expression) statement

首先评估 expression 的值，若为真（也就是非 0），那么就执行 statement 对应的语句，然后再次判断 expression。一旦 expression 为假（也就是 0），while 循环就终止了。

do 语句与之相似，只是判断放在了循环尾部。它的语法是：

do statement while(expression)

先执行 statement 中的语句，然后再判断 expression。若为真，则继续执行循环，若为假，则循环终止。

for 语句的格式为：

for($e_1; e_2; e_3$) statement

首先执行 e_1 ，这部分通常是赋值表达式。然后判断 e_2 ，它一般是个关系表达式，如果

① 例如 C 语言中表达式 $-5 < -4 < -3$ 就不成立，因为必须先求 $-5 < -4$ 这个子表达式的值，它成立，故被视为 1，然后再判断 1 是否小于 -3 ，此判断不成立，故整个表达式不成立。——译者注

② 为了遵循业内惯例，语法格式中的英文单词在译文中保留原样。其中的 expression 意为“表达式”，statement 意为“语句”。——译者注

为假，则终止 for 循环，如果为真，那就执行 statement。最后再执行 e_3 ，这部分通常也是个赋值表达式。执行完毕后，又跳回 e_2 判断。因此，“do $i = 1$ to n ” 用 for 语句写出来就是：

```
for (i = 1; i <= n; i++)
```

(书中用到后置自增运算符的场合不多，这是一例。)

ISO C 标准并未规定右移（“ \gg ”操作符）带符号的数时，左边多出来的空位是用 0 填充还是用符号填充^①。在本书的 C 语言代码中，我们假定如果左操作数是带符号的，那么右移操作就使用符号填充，若无符号，则按 ISO 规范以 0 填充。大多数 C 编译器也都这么做。

左移操作符都采用“逻辑”填充。（某些电脑还有“算术”左移操作，也就是左移之后保持符号不变。这通常出现在老式计算机上。）

移位操作还会出现一个问题，那就是 ISO C 标准规定：假如移位的数量等于或大于左操作数的位宽，则结果未定义。虽说如此，但是几乎所有 32 位机在遇到这种情况时，都会把移位数量和 32 或 64 取模^②。书中代码在对待此问题时，上述各种处理方式都会用到。若不同方式之间的差异关乎运算结果，则会给出解释。

1.2 指令集与执行时间模型

我们假定测试书中算法时所用的这台电脑，其 CPU 使用当前通行的 RISC 指令集，这样就好粗略估量其效率了。IBM RS/6000 系列电脑^③的 CPU、Oracle SPARC^④及 ARM 架构^⑤的处理器都用的是 RISC。这台虚拟电脑的 CPU 使用“三段式地址”表示操作数，其寄存器相当多，至少 16 个。除非另有说明，否则寄存器都是 32 位的。0 号通用寄存器的值永远是 0，其他寄存器作用不限。

有些 CPU 中具备“特殊用途”寄存器^⑥，专门保存条件比较的结果或“溢出”等状

- ① 用 0 填充 (0-propagating) 就是表 1.1 中所说的“逻辑”移位，而用符号填充 (sign-propagating) 的意思是，如果待移位的数是负值，那么就用 1 来填充，这样移位后的结果还是负的；而如果待移位的数非负，则用 0 填充，移位后的结果仍然非负。这种保持正负性的填充方式也就是上表所说的“算术”移位或“数学”移位。——译者注
- ② 举例来说，若要将一个 32 位元的数左移 80 位，那么由于 80 大于等于 32，所以必须先求它除以 32 的余数，也就是 16，然后将该数左移 16 位，得出运算结果。——译者注
- ③ RS6000 是 IBM 公司使用其 RISC 架构的 Power 处理器设计生产的小型计算机。详情参见：<https://zh.wikipedia.org/wiki/RS/6000>。——译者注
- ④ SPARC，全称为“可扩充处理器架构”(Scalable Processor ARChitecture)，是 RISC 微处理器架构之一。详情参见：<https://zh.wikipedia.org/wiki/SPARC>。——译者注
- ⑤ ARM 架构是一种 32 位元精简指令集 (RISC) 处理器架构，因其具备低成本、高效能、低耗电的特性，故广泛用于嵌入式系统。详情参见：https://zh.wikipedia.org/wiki/ARM_架构。——译者注
- ⑥ special purpose register，又叫 special function register 或 special register，详情参见：https://en.wikipedia.org/wiki/Special_function_register。——译者注

态码之类的数据，而我们为了简洁起见，假设虚拟电脑的CPU中没有这种寄存器。此外，它还没有浮点指令。浮点数在书中只占少量篇幅，基本上局限于第17章。

现在规定两套RISC指令集：表1.2是“基本RISC指令集”，这些指令再加上表1.3中的那些，就构成“完整RISC指令集”。

表1.2 基本RISC指令集

操作码助记符	操作数	含义
add, sub, mul, div, divu, rem, remu	RT, RA, RB	$RT \leftarrow RA \text{ op } RB$, 其中op可以是加法、减法、乘法、带符号数的除法、无符号数的除法、带符号数的求余、无符号数的求余
addi, muli	RT, RA, I	$RT \leftarrow RA \text{ op } I$, 其中op可以是加法或乘法, I是16位元的带符号常量 ^①
addis	RT, RA, I	$RT \leftarrow RA + (I \ll 16)$
and, or, xor	RT, RA, RB	$RT \leftarrow RA \text{ op } RB$, 其中op可以是按位与、按位或、按位异或
andi, ori, xori	RT, RA, Iu	同上, 但最后一个操作数是16位元的无符号常量
beq, bne, blt, ble, bgt, bge	RT, target	在条件成立时转入target分支。这6个操作符的判断条件分别是: $RT = 0$, $RT \neq 0$, $RT < 0$, $RT \leq 0$, $RT > 0$, $RT \geq 0$
bt, bf	RT, target	当RT是true/false时转入target, 与bne/beq等效
cmpeq, cmpne, cmplt, cmple, cmpgt, cmpge, cmpltu, cmpleu, cmpgtu, cmpgeu	RT, RA, RB	将RA与RB比较的结果存入RT。若判断不成立, 则RT为0, 若成立, 为1。cmpeq表示compare for equality(判断二者是否相等), cmpne表示compare for inequality(判断两者是否不等), cmplt表示compare for less than(判断前者是否小于后者), 以此类推。cmp后面的字母与分支指令中的含义相同。后缀“u”表示无符号数的比较
cmpeq, cmpine, cmpilt, cmpile, cmpigt, cmpige	RT, RA, I	与cmpeq等6个操作符含义相同, 只是I为16位元的带符号常量
cmpequ, cmpineu, cmpiltu, cmpileu, cmpigtu, cmpigeu	RT, RA, Iu	与cmpltu等6个操作符含义相同, 只是Iu为16位元的无符号常量
ldbu, ldh, ldhu, ldw	RT, d (RA)	分别将地址RA+d中的无符号字节、半字组、无符号半字组、字组载入RT。d为16位元的带符号常量
mulhs, mulhu	RT, RA, RB	将RA与RB之积的高32位存入RT。两指令分别适用于带符号数及无符号数
not	RT, RA	将RA按位取反后的值存入RT
shl, shr, shrs	RT, RA, RB	将RA左移位或右移位后的值存入RT。RB最右方的6个位元代表移位量。shl与shr用0填充, shrs用RA的符号位填充(移位量只取最右方6个位元的原因是, 它需要和64取模, 以便调整到0至63之间)
shli, shri, shrsi	RT, RA, Iu	将RA左移位或右移位后的值存入RT。移位量是Iu所代表的5位元常数
stb, sth, stw	RS, d (RA)	将RS里的字节、半字、字组存入RA+d所表示的内存地址中。d是16位元的带符号常量

^① 本书多次出现immediate value一词, 字面意思为“立即值”、“立即数”, 它是谈到计算机指令时的一种术语, 在不影响文意的情况下, 译文均以“常量”、“常数”称呼之。——译者注

表1.2、表1.3与表1.4中的RA与RB，如果作为源操作数使用，就表示这些寄存器本身的内容。^②

在实际使用的CPU中，尚有分支链接跳转指令（branch and link）以及返回到寄存器中所含地址的分支返回指令，前者可用来调用子程序，后者则用于从子程序中返回或实现“switch”功能。此外，可能存在一些处理专用寄存器的指令，特权指令与调用管理服务的指令当然也不少，或许还会有浮点数指令。

表1.3列出了RISC指令集中可能会用到的其他一些计算类指令，后面的章节中将会讲到它们。

汇编语言中提供了一些“扩展助记符”（Extended Mnemonic），开发人员用起来很方便，它们有点像宏，展开之后通常是一条指令。表1.4列举了一些可能会用到的扩展助记符。

常量加载操作会根据常量I的值而展开成一条或两条指令。如果 $0 \leq I < 2^{16}$ ，那么就展开成R0与I的常量按位或操作（ori指令），而当 $-2^{15} \leq I < 0$ 时，则会展开成R0与I的常量加法操作（addi指令）。若I最右方的16个位元都是0，那么就视为常量移位加法操作（addis指令），而在不属于上述三种情况时，则会展开成一条addis与一条ori指令。^③（在最后一种情况下，也可以直接使用从内存中加载数值的指令来做。但是为了便于估量算法所需的执行时间和空间，我们假定此种常量加载操作总是相当于两个算术指令。）

表1.3 “完整RISC指令集”中的其余附加指令

操作码助记符	操作数	含义
abs, nabs	RT, RA	将RA的绝对值或“负绝对值” ^① 存入RT
andc, eqv, nand, nor, orc	RT, RA, RB	将RB按位取反后与RA按位与，按位比较，按位与后按位取反，按位或后按位取反，将RB按位取反后与RA按位或
extr	RT, RA, I, L	将RA中序号为I至I+L-1之间的位元提取出来，靠右存放在RT中，余位填0
extras	RT, RA, I, L	与extr相同，但是用符号填充
ins	RT, RA, I, L	将RA中序号为0至L-1的位元插入RT中序号为I至I+L-1的位置上
nlz	RT, RA	取得RA中前导0的个数（该操作的结果在0至32之间）
pop	RT, RA	RT计算RA中有多少个值为1的位元（该操作的结果在0至32之间）
ldb	RT, d (RA)	从RA+d所在的内存地址处加载一个带符号的字节到RT中。d为16位元的带符号常量
moveq, movne, movlt, movle, movgt, movge	RT, RA, RB	如果RA与0的关系满足判断条件，就将RB赋值给RT，否则RT的值不变。eq与ne分别表示RA=0、RA≠0，其余依此类推
shlr, shrr	RT, RA, RB	将RA循环左移位或循环右移位。RB最右方的5个位元表示移位的数量

^② 若RA、RB不以带括号形式出现，即表示其本身值，否则视为定位所用的内存地址。——译者注

^③ 常量加载操作要分成三种情况展开的原因，请参考：<http://www.engr.uconn.edu/~jeffm/Classes/CSE240-Spring-2000/Lectures/lecture6/node4.html>。——译者注

(续)

操作码助记符	操作数	含义
shlri, shrri	RT, RA, Iu	将 RA 循环左移位或者循环右移位，移位的数量在 5 位立即数字段中给出
trpeq, trpne, trplt, trple, trpgt, trpge, trpltu, trpleu, trpgtu, trpgeu	RA, RB	当两个操作数满足 $RA = RB$ 、 $RA \neq RB$ 等条件时，令处理器中断（trap）
trpieq, trpine, trpilt, trpile, trpigt, trpige	RA, I	与 trpeq 等指令含义相同，只是第 2 个操作数为 16 位元的带符号常量
trpiequ, trpineu, trpiltu, trpileu, trpigtu, trpigeu	RA, Iu	与 trpltu 等指令含义相同，只是第 2 个操作数为 16 位元的无符号常量

① negative of the absolute value 是一种特有的绝对值解读方式，详情参见：http://pic.dhe.ibm.com/infocenter/aix/v7r1/index.jsp?topic=%2Fcom.ibm.aix.aixassem%2Fdoc%2Falangref%2Fidalangref_nabs_negabs_instrs.htm。——译者注

表 1.4 扩展助记符

扩展助记符	展开式	含义
b target	beq R0, target	无条件执行分支
li RT, I	详见书中解释	将常量加载至 RT 中。 $-2^{31} \leq I < 2^{32}$
mov RT, RA	ori RT, RA, 0	将寄存器 RA 中的值赋给 RT
neg RT, RA	sub RT, R0, RA	将 RA 的相反数（也就是二补码）放入 RT
subi RT, RA, I	addi RT, RA, -I	将 RA 减 I 的差放入 RT ($I \neq -2^{15}$) ^①

① 因为 -2^{15} 的相反数是 2^{15} ，该值无法以 16 个位元容纳，所以在这种特殊情况下，subi 与 addi 展开式不等效。——译者注

一个指令到底应该属于基本 RISC 指令集还是完整 RISC 指令集，其实取决于个人的解读。有人可能会觉得无符号数除法与求余操作应该放在完整 RISC 指令集中，而与此相反，带符号的字节加载（load byte signed, ldb）操作反而应该归入基本 RISC 指令集。ldb 操作之所以会被放在完整指令集中，是因为其使用频率相当低，而且该指令需要将符号位扩展很多次^②，这样做比较耗费 CPU 周期。

划分基本 RISC 指令集与完整 RISC 指令集时，还有许多可以商榷的地方，此处不再赘言。

书中所用指令最多只会用到两个源寄存器与一个目标寄存器，这样做也简化了计算机的 CPU 架构（比方说，寄存器堆^③中最多只需两个读端口与一个写端口即可）。此外也缓解了优化编译^④的工作量，使它不用再处理那些用到多个目标寄存器的指令了。这样做

① 如果待加载字节的最高有效位是 1，则表明其为负数，将它加载到一个 32 位宽的寄存器时，必须把多出来的 24 个空位元全部扩展为 1，这样才能保证寄存器中的值和正负性与原字节相符。——译者注

② Register File，是 CPU 中多个寄存器组成的阵列，通常由快速的静态随机读写存储器（SRAM）实现。这种 RAM 具有专门的读端口与写端口，可以多路并发访问不同的寄存器。详情参见：<https://zh.wikipedia.org/wiki/寄存器堆>。——译者注

③ Optimizing Compiler，是优化程序所用的编译器，可以加快程序执行速度或减低内存占用量等。详情参见：https://en.wikipedia.org/wiki/Optimizing_compiler。——译者注

的代价是，有些本来执行一条指令就能实现的操作现在得分解成两条。比如某些程序想同时知道两个数的商和余数（这种情况很罕见），那么现在就必须用除法和求余两个操作才能完成。在现实环境下的 CPU 中，余数是除法操作的副产品 (by-product)，很多电脑在执行除法指令时顺便计算好了余数。要获取保存两个字组乘积的双字时，也会遇到这个问题。

条件移动指令 (conditional move，例如 `moveq`) 表面上看只有两个源操作数，但换个角度看，也可以说是 3 个。因为该指令的结果取决于 RT、RA 与 RB 的值，所以乱序执行的 CPU 必须把这种指令里的 RT 标注为 use 和 set。也就是说，如果前面一条指令修改了 RT，而后面紧跟一个将要再度修改 RT 的条件移动指令，那么就只能按照这个顺序执行，同时不能丢弃前一条指令的运算结果。有鉴于此，此类处理器的设计者可以选择略过条件移动操作以避免处理这种（从逻辑上讲）需要 3 个源操作数的指令。反之，使用条件移动指令确实能简化分支的实现。

指令格式与本书内容无关，不过上面列出的所有 RISC 指令集，再加上浮点指令和一些管理用的指令，都可以在有 32 个通用寄存器的电脑上用 32 个位元（其中用于指示寄存器序号的字段占 5 个位元^①）加以实现。如果将 `compare`、`load`、`store`、`trap` 指令所支持的常数字段缩减为 14 位，那么这些指令在具备 64 个通用寄存器的电脑上，也照样能用 32 个位元做出来（其中用于指示寄存器序号的字段占 6 个位元^②）。

执行时间

本书假设所有指令都只需 1 个 CPU 周期即可执行完毕，但是乘法、除法及求余指令例外，我们不去预估这 3 个指令的执行时间。而分支指令不论其执行结果是转入分支还是继续沿主线执行，都只花 1 个周期。

常量加载指令会花费一至两个周期，因为要把常数存入寄存器中，所需的算术指令数可能是 1 个，也可能是两个。

书里用到加载与存储指令的场合不多，我们也假设其只需 1 个周期，而且忽略加载延迟（也就是从算术逻辑单元^③执行完加载指令算起，到待加载的数据真正可以用于后续指令为止，这两者的时间间隔）。

然而，只晓得算术与逻辑指令的执行周期，通常无法估算出程序执行时间，因为数

^① 在 CPU 中，除了操作数之外，指令本身也是用二进制来表示的。在由 32 个位元构成的指令中，一般会有指示操作类型的操作码字段、标识寄存器序号的字段，以及容纳常量的字段等。5 个位元能表达值为 0~31 的 32 个数，所以刚好可以对应 32 个通用寄存器的序号。详情参见：<https://zh.wikipedia.org/wiki/指令集架构>。——译者注

^② 一般常量用 16 个位元表示，而指令码本身通常含有两个表示寄存器的字段，现在要将其由 5 位扩展至 6 位，所以原来表示常量用的字段就会缩减为 14 位。——译者注

^③ 原文为 arithmetic unit，应是 arithmetic logic unit (ALU) 的简称，它是中央处理器的执行单元，也是所有 CPU 的核心组成部分，由“And Gate”和“Or Gate”构成，主要用于进行加、减、乘等二进制算术运算。详情参见：<https://zh.wikipedia.org/wiki/算术逻辑单元>。——译者注

据加载延迟及获取指令延迟可能会大大减缓程序执行速度。尽管近些年人们越来越重视这个问题了，但是本书不打算讨论。另外还有一个因素会改善执行效率，那就是所谓的“指令级并发”(instruction-level parallelism)。很多时下流行的RISC芯片都支持该技术，那些“高端”(high-end)电脑尤其如此。

此类机器的CPU都有多个执行单元，并具备足够的指令派发能力，以便并发执行互不依赖的一组指令（也就是说，这些指令的执行结果都不取决于其他指令，而且不会同时修改某个寄存器或状态位）。由于此技术已非常普遍，所以本书会标明那些相互独立的指令。如此一来，我们就可以说：某某公式用8条指令即可实现，并且在一台具备无限指令级并发能力的电脑上运行，只需5个周期。这也意味着，假使CPU能够合理安排(schedule)各条指令的执行顺序，而且其中加法器^②、移位器(shifter)、逻辑单元及寄存器的数量足够多，那么从理论上讲，只要5个周期就能执行完这段代码了。

然而也不能过分强调这一点，因为不同电脑的指令级并行能力差得很远。比如一台1992年加州产的IBM RS/6000系列电脑^③，其CPU的加法器就有3个输入端，甚至能够并发执行两条首位衔接的加法类指令（比如加法指令的目标寄存器当做比较指令或加载指令的源寄存器用^④）。与之相对，那种在低端嵌入式设备上运行应用程序所用的CPU的构造非常简单，其寄存器堆栈可能只有一个读端口。一般说来，这种机器执行需要两个源寄存器的指令时，会多花一个周期来读取第2个操作数。然而这种CPU也许会有一个旁路(bypass)，当某条指令的目标寄存器恰好是下面那条指令的源操作数时，它不用再通过寄存器堆的读端口就可以把寄存器拿来用。在这种设备上以非并行方式执行首位衔接的代码反倒发挥了它的优势。

1.3 习题

1. 将如下循环用while方式改写：

for (e₁; e₂; e₃) statement

- 此循环能用do表示吗？
- 用C语言写一个循环，控制名为i的无符号整数变量，使其值从0开始，递增到32位机所能表示的最大无符号数0xFFFF FFFF（循环范围包含这两个极值）。
 - 此题留给知识丰富的读者：本书所列的基本RISC指令集和完整RISC指令集中，每条指令最多只需读取两次寄存器，写入一次寄存器。请问在常见的RISC指令中，有没有那种貌似简单但实际上却需要操作两个以上源数据，或多次写入寄存器的指令呢？

^① 在电子学中，加法器(adder)是一种用于执行加法运算的数字电路部件，是计算机核心微处理器中算术逻辑单元的基础。在这些电子系统中，它主要负责计算地址、索引等数据。此外，它还是二进制数乘法器等其他硬件的重要组成部分。详情参见：<https://zh.wikipedia.org/wiki/加法器>。——译者注

^② IBM RS/6000系列电脑的发展历程可参阅其官方文档：<http://www-03.ibm.com/ibm/history/documents/pdf/rs6000.pdf>。——译者注

^③ 原书的说法是one feeds the other，即前一条指令将操作结果作为数据源，“喂”给后一条指令。——译者注

第 2 章 基础知识

2.1 操作最右边的位元

本节所讲的公式有一些可能会在后面章节中用到。

下列公式可以将字组中值为 1 且最靠右的位元“关闭”[⊖]，如果不存在值为 1 的位元，那结果就是 0（例如 $0101\ 1110 \Rightarrow 0101\ 0000$ ）：

$$x \& (x - 1)$$

该操作可判断某个无符号整数是不是 2^n 的幂或 0：套用此公式后判断运行结果是否为 0 即可。

下列公式可以将字组中值为 0 且最靠右的位元“打开”[⊖]，如果不存在值为 0 的位元，则结果的每一位都是 1（例如 $1010\ 0111 \Rightarrow 1010\ 1111$ ）：

$$x | (x + 1)$$

下面的公式可以将字组尾部的 1 都变成 0，如果尾部没有 1，则 x 不变（例如 $1010\ 0111 \Rightarrow 1010\ 0000$ ）：

$$x \& (x + 1)$$

套用此公式后判断结果是否为 0，即可确定某个无符号整数是不是 $2^n - 1$ 或 0，也可以判断某个数的所有位元是否均为 1。

以下公式可以将字组尾部的 0 都变成 1，如果尾部没有 0，则 x 不变（例如 $1010\ 1000 \Rightarrow 1010\ 1111$ ）：

$$x | (x - 1)$$

下面的公式可以把 x 中最靠右且值为 0 的位元变为 1，并将其余位元置 0。如果 x 中没有值为 0 的位元，则结果为 0（例如 $1010\ 0111 \Rightarrow 0000\ 1000$ ）：

$$\neg x \& (x + 1)$$

下面的公式可以把 x 中最靠右且值为 1 的位元变成 0，并将其余位元置 1。如果 x 中

[⊖] 原文为 turn off，也就是将位元的值变为 0。——译者注

[⊕] 原文为 turn on，也就是将位元的值变为 1。——译者注

没有值为1的位元，则运算结果中的每个位元均是1（例如 $1010\ 1000 \Rightarrow 1111\ 0111$ ）：

$$\neg x | (x - 1)$$

下列3个公式都可以把字组尾部所有值为0的位元变成1，并将其余位元置0。如果 x 中没有值为0的位元，则结果是0（例如 $0101\ 1000 \Rightarrow 0000\ 0111$ ）：

$$\neg x \& (x - 1), \text{ 或}$$

$$\neg(x | \neg x), \text{ 或}$$

$$(x \& \neg x) - 1$$

第1个公式可以发挥处理器的某些指令级并行能力。

下列公式可以把字组尾部所有值为1的位元都变成0，并将其余位元设为1。如果 x 中没有值为1的位元，则运算结果中的每个位元均是1（例如 $1010\ 0111 \Rightarrow 1111\ 1000$ ）：

$$\neg x | (x + 1)$$

下列公式可以保留字组中最靠右且值为1的位元，并将其余位元置0。若 x 不存在值为1的位元，则运算结果为0（例如 $0101\ 1000 \Rightarrow 0000\ 1000$ ）：

$$x \& (\neg x)$$

下列公式可以将字组最靠右且值为1的位元，及其右方所有值为0的位元都变成1，并将左方位元置0。若 x 中没有值为1的位元，则运算结果的每一位都是1，而当 x 尾部没有值为0的位元时，运算结果是1（例如 $0101\ 1000 \Rightarrow 0000\ 1111$ ）：

$$x \oplus (x - 1)$$

下列公式可以将字组最靠右且值为0的位元，及其右方所有值为1的位元都设为1，并将左方位元置0。若 x 中没有值为0的位元，则运算结果的每一位都是1，而当 x 尾部没有值为1的位元时，运算结果是1（例如 $0101\ 0111 \Rightarrow 0000\ 1111$ ）：

$$x \oplus (x + 1)$$

下列两个公式都可以把字组右侧连续出现且值为1的位元置0（例如 $0101\ 1100 \Rightarrow 0100\ 0000$ ）[Wood]：

$$(((x | (x - 1)) + 1) \& x), \text{ 或}$$

$$((x \& \neg x) + x) \& x$$

如果 x 是非负数，而且套用上述公式运算的结果是0，那就表明它可以写为 $2^j - 2^k$ 的形式。其中 $j \geq k \geq 0$ 。

2.1.1 德摩根定律的推论

描述德摩根定律[⊖]的两个逻辑恒等式可以理解为把取反符号（not sign）“分配”到每一个变量中。结合下列等式（头两个就是德摩根定律）与德摩根定律的思路，可以为本节所讲的公式及其他一些公式变形。

[⊖] 德摩根定律（De Morgan's law）是由英国数学家、逻辑学家奥古斯塔斯··德摩根（Augustus De Morgan, 1806—1871）所陈述的逻辑学定律。详情参见：<https://zh.wikipedia.org/wiki/德摩根定律>。——译者注

$$\begin{aligned}
 \neg(x \&y) &= \neg x \mid \neg y \\
 \neg(x \mid y) &= \neg x \& \neg y \\
 \neg(x + 1) &= \neg x - 1 \\
 \neg(x - 1) &= \neg x + 1 \\
 \neg\neg x &= x - 1 \\
 \neg(x \oplus y) &= \neg x \oplus y = x \equiv y \\
 \neg(x \equiv y) &= \neg x \equiv y = x \oplus y \\
 \neg(x + y) &= \neg x - y \\
 \neg(x - y) &= \neg x + y
 \end{aligned}$$

上述公式可以这样用： $\neg(x \mid -(x + 1)) = \neg x \& \neg\neg(x + 1) = \neg\neg x \& ((x + 1) - 1) = \neg x \& x = 0$ 。

2.1.2 从右至左的可计算性测试

有一种简单的办法可以判断出某个函数是否能通过一系列加法、减法、按位与、按位或及按位取反实现出来^①[War]。当然，还有一些能够通过上述基本操作组合而成的运算方法也在允许范围内，比如移位数量固定的左移操作（等价于一系列加法操作）或乘法操作等。除此之外的其余操作都是不允许的。下面这条定理描述了如何判断一个函数是否可以按照从右至左的顺序实现出来。

定理 一个从字组到字组的映射函数，当且仅当运算结果中的每一个位元只依赖于各操作数中对应位元及其右侧位元时，才可以用一系列字并行^②加减法、按位与、按位或及按位取反操作实现出来。^③

也就是说，要判断某个函数是否能够从右至左实现出来，大家需要设想：运算结果最右侧的那个位元能不能只用各个操作数最右侧的位元来算，而运算结果的倒数第2个位元能不能只能用各操作数最右方的两个位元来算，以此类推。如果能这么算出来，那就说明该函数可以用一系列的加法、按位与等简单操作实现出来。若是不能，那就表示只通过上述基本指令无法实现这个函数。

定理中有意思的地方就是“当且仅当……”这个分句。按照定理中的说法，加、减、按位与、按位或、按位取反等操作都能按照从右至左的顺序实现出来，于是，由这些操作组合而成的各种复合运算方式居然也能从右至左算出来，这实在出乎很多人意料。

^① 也就是判断某个函数是否具备本节标题所谓的“从右至左的可计算性”(right-to-left computability test)。——译者注

^② word-parallel，是将每个字组中的位元横列出来，再把若干个字组纵向对齐写出的一种演算方式。——译者注

^③ 该定理用通俗的话来表述就是：如果一个函数能够以位元为单位从右至左计算出来，那么它就肯定能用那5个简单操作及其复合指令实现；反之，如果能用这些基本指令来描述一个函数，那么它必然也能通过从右至左的按位运算方式做出来。——译者注

为了验证定理中的“当且仅当……”这句话，需要用一点笨办法才行。现在构建如下特例：假设有一个函数具备从右至左的可计算性，它有两个名为 x 、 y 的变量，而且 2 号位元 r 是用如下算式得出的：

$$r_2 = x_2 \mid (x_0 \& y_1) \quad (1)$$

按照从右至左的顺序，将各个位元标为 0 至 31 号。由于计算 2 号位元时只用到了各操作数中序号小于等于 2 的位元，所以 2 号位元就可以“从右至左计算出来”（right-to-left computable）。

把字组 x 、 x 左移两位后的数、 y 左移一位后的数从上至下写出来，再写一个只有 2 号位为 1 的掩码。

x_{31}	x_{30}	…	x_3	x_2	x_1	x_0
x_{29}	x_{28}	…	x_1	x_0	0	0
y_{30}	y_{29}	…	y_2	y_1	y_0	0
0	0	…	0	1	0	0
0	0	…	0	r_2	0	0

将字组横列式中第 2 行与第 3 行按位与，然后和第 1 行按位或（操作的依据是等式（1）），最后跟第 4 行的掩码按位与。这么计算的结果是，除了要计算的 2 号位元外，其余位置都变成了 0。对于待求值的其他 31 个位元，也都按照此方法演算，最后把 32 个字组按位或，于是就实现了整个函数。

按照此方式构建的算法效率并不高，不过这里只是为了演示此类函数的确能用 5 种基本指令拟合出来而已。

根据上述定理，立刻就能断定：无法通过 5 种基本指令序列把字组最左侧且值为 1 的位元置 0，因为要想判断字组中某个值为 1 的位元该不该变成 0，必须继续向左方搜寻，以确保其左侧再也没有其他值为 1 的位元了。同理，右移、循环移位、移位长度为变量的左移、计算字组尾部值为 0 的位元个数等操作都无法用 5 种简单的运算组合出来（在计算尾部值为 0 的位元个数时，如果结果为奇数，那么其最右侧的位元必然是 1，而要想确定这一点，就必须继续向左看，以确保操作数尾部真的有奇数个值为 0 的位元）[⊖]。

2.1.3 位操作的新式用法

位操作还有一种新用法，也用到了上面那些技巧，那就是给定一个数，然后找出下一个比它大而且值为 1 的位元数与之相同的数字。读者可能会问：计算这样的数有什么用

[⊖] 作者的意思是，假如要从右至左算出 0100 1000 这个数尾部有多少个 0（正确答案是 3 个，用二进制写出则是 0000 0011 个），必须先确定答案的最右方位元是 0 还是 1，而这将由操作数尾部 0 的个数是奇数还是偶数来决定，要想确定这个事实，却必须继续向左看，才能知道尾部的 0 到底是奇数个，还是偶数个。这样一来，就违背了“从右至左”的运算规则——计算答案最右方的位元时，只能用操作数最右方的位元来算，不能向左取值。——译者注

呢？它用在以位串来表示子集的领域。集合中的元素可以写成一行，然后可以用一个字组或字组序列来表示它的子集。如果集合中的某个元素在子集中，那么字组里对应的位元就为 1，否则为 0。想求两个子集的并集，只需要把对应的位串按位取或即可，而想求交集，就按位取和，此外还有很多类似的操作。

有时可能需要找出元素个数为某一定值的全部子集。如果有一个函数，能够根据某个给定的整数（将表示子集的位串视为整数），找出下一个比它大而值为 1 的位元个数又与之相同的数，那么很快就能据此找出所有子集了。

R. W. Gosper [HAK, item 175] 设计出了此操作的精确算法[⊖]。给定一个表示子集位串的字组 x ，首先要找到连续出现在 x 右侧且值为 1 的一组位元，然后将该值“加 1”，再把原来后面跟着的那些 0 补上。举例来说，如果待计算的位串是 $xxx0\ 1111\ 0000$ ，那么结果就应该是 $xxx1\ 0000\ 0111$ ，其中 xxx 这三个位元的值不限。该算法首先定义 $s = x \& -x$ ，并算出 s 等于 $0000\ 0001\ 0000$ ，这样就找到了 x 中“最小”[⊖]的那个 1。然后把它与 x 相加，把两数之和 $xxx1\ 0000\ 0000$ 保存在 r 里。此时结果中的一个位元已经算好了，它就是 r 里面的那个“1”。想求出其他位元，还需要把位串中剩下的 $n-1$ 个“1”移到右侧（其中 n 指的是连续出现在 x 右侧且值为 1 的位元个数）。要把它们移到右边，首先得计算 r 和 x 的异或值，在本例中就是 $0001\ 1111\ 0000$ 。

上面那个值中“1”的个数太多了，而且没有靠右对齐。为解决此问题，要将它与 s 相除，这样就可以把那些“1”靠右对齐了（因为 s 是 2 的幂），除之前还要先向右移两位，以便丢弃那两个多余的位元。将此结果与 r 取或，就得到最终答案了。

最终结果 y 用计算机代数来表示，就是：

$$\begin{aligned} s &\leftarrow x \& -x \\ r &\leftarrow s + x \\ y &\leftarrow r \mid (((x \oplus r) \gg^u 2) \div^u s) \end{aligned} \tag{2}$$

图 2.1 中列出了此算法的完整 C 语言实现，它执行了 7 个基本的 RISC 指令，其中 1 个用于除法。（不要以 0 为参数调用此段程序，否则会因与 0 相除而出错。）

```
unsigned snoob(unsigned x) {
    unsigned smallest, ripple, ones;
    // x = xxxx0 1111 0000
    smallest = x & -x;           // 0000 0001 0000
    ripple = x + smallest;      // xxxx1 0000 0000
    ones = x ^ ripple;          // 0001 1111 0000
    ones = (ones >> 2)/smallest; // 0000 0000 0111
    return ripple | ones;        // xxxx1 0000 0111
}
```

图 2.1 计算下一个值比 x 大而 1 的个数与之相同的数

若是嫌除法太慢，可以用另外一种算法代替。设函数 $ntz(x)$ 为 x 尾部 0 的个数， nlz

[⊖] 该算法的另外一种形式请参考 [H & S] 7.6.7 节。

[⊖] 也就是最靠右且值为 1 的位元。——译者注

(x) 为 x 头部 0 的个数，而 $\text{pop}(x)$ 为 x 中 1 的个数（又叫种群统计函数，population count），假如能够快速计算出这三个函数的话，那么就可以用下列三个公式之一来取代等式(2)的最后一个式子。（如果公式中的右移操作用的是“模 32 移位”[⊖]，那么前两种替代方案无法得出正确结果。）

$$\begin{aligned}y &\leftarrow r \mid ((x \oplus r) \gg (2 + \text{ntz}(x))) \\y &\leftarrow r \mid ((x \oplus r) \gg (33 - \text{nlz}(s))) \\y &\leftarrow r \mid ((1 \ll (\text{pop}(x \oplus r) - 2)) - 1)\end{aligned}$$

2.2 结合逻辑操作的加减运算

本书假定读者已经熟悉普通代数运算及布尔运算中的一些基本恒等式。下面列出一组将逻辑操作与加减法结合起来的恒等式。

a.	$-x = \neg x + 1$
b.	$= \neg(x - 1)$
c.	$\neg x = -x - 1$
d.	$- \neg x = x + 1$
e.	$\neg -x = x - 1$
f.	$x + y = x - \neg y - 1$
g.	$= (x \oplus y) + 2(x \& y)$
h.	$= (x y) + (x \& y)$
i.	$= 2(x y) - (x \oplus y)$
j.	$x - y = x + \neg y + 1$
k.	$= (x \oplus y) - 2(\neg x \& y)$
l.	$= (x \& \neg y) - (\neg x \& y)$
m.	$= 2(x \& \neg y) - (x \oplus y)$
n.	$x \oplus y = (x y) - (x \& y)$
o.	$x \& \neg y = (x y) - y$
p.	$= x - (x \& y)$
q.	$\neg(x - y) = y - x - 1$
r.	$= \neg x + y$
s.	$x \equiv y = (x \& y) - (x y) - 1$
t.	$= (x \& y) + \neg(x y)$

[⊖] modulo 32 shift，CPU 在执行此指令前，先要把移位的个数（也就是该操作符的右操作数）调整成一个大于等于 0 且小于 32 的整数，故有此称呼。——译者注

$$\begin{array}{ll} u. & x|y = (x \& \neg y) + y \\ v. & x \& y = (\neg x|y) - \neg x \end{array}$$

等式 (d) 可以重复运用, 例如 $\neg\neg\neg x = x + 2$ 。同理, 多次套用等式 (e), 也可以推出 $\neg\neg\neg x = x - 2$ 。由此可见, 与常量的加减法都可以只用按位取反及求相反数操作组合而成[⊖]。

等式 (f) 与 (j) 对偶。大家都知道: 利用等式 (j) 所描述的关系, 可以通过加法器构造出减法器。

等式 (g) 和 (h) 记载于 HAKMEM 备忘录中 [HAK, item23]。等式 (g) 先对两数做不进位加法 ($x \oplus y$), 然后再补上进位。等式 (h) 改变了加法运算的顺序, 使得在计算任何一个位元的过程中, 都不可能出现 0+1 的组合。原来会出现此情况的地方, 现在都变成了 1+0。

在普通的二进制加法中, 若单独分析运算结果中的每一个位元, 则它等于 0 或等于 1 的概率相等, 而在每个位置上发生进位的概率则是 0.5。但是如果在制作加法器时, 使用等式 (g) 描述的这种方法对输入数字进行预处理的话, 那么进位概率就降到 0.25 了。这一论述对构建加法器的意义不是太大, 因为设计加法器时关心的是, 能不能把进位在逻辑电路中传播的最大位元数降到最低, 而等式 (g) 只能把这个传播距离缩小 1 位。

用于减法操作的等式 (k)、(l) 分别与加法操作中的 (g)、(h) 对偶。也就是说 (k) 先对两数做不进位减法 ($x \oplus y$), 然后再把借来的位从结果中减去。同理, 等式 (l) 只不过调整了操作数, 使得在计算任何一个位元的过程中, 都不可能出现 1-1 这样的组合。原来会出现此情况的地方, 现在都变成了 0-0。

等式 (n) 演示了只需 3 个基本 RISC 指令就可实现异或操作。如果只用与、或、非这 3 种逻辑操作来实现, 则需 4 个指令: $((x|y) \& \neg(x \& y))$ 。同理, 等式 (u) 和 (v) 演示了如何用 3 个基本的指令来实现按位与及按位或操作。这两个操作如果用德摩根定律的形式来做, 需要 4 个指令。

2.3 逻辑与算术表达式中的不等式

如果将二元逻辑表达式[⊖]中的值视为无符号整数, 则很容易推出一系列不等式来。比如下面这两个例子:

[⊖] 原文为 “using only two forms of complementation”, 意为“只用两种形式的补码就可以算出”。因为按位取反与求相反数操作分别相当于求某数的“一补码”(俗称“反码”)和“二补码”(俗称“补码”), 故有此说。——译者注

[⊖] binary logical expression, 也可以理解为“二进制逻辑表达式”, 因为操作数是 0、1 这两个二进制数, 然而 binary 一词更多是在强调这些逻辑运算的操作数多为两个, 而且其值和最后的运算结果也是以“真”、“假”两种形态出现的, 故译文在不致混淆时, 均将其称为“二元逻辑表达式”。——译者注

$$(x \oplus y) \leq^u (x + y) \text{ 和}$$

$$(x \& y) \leq^u (x \equiv y)$$

表2.1列出了所有二元逻辑操作的运算结果。上述二式皆可由此得出。

表2.1 16种二元逻辑操作

x	y	0	$x \& y$	$x \& \neg y$	x	$\neg x \& y$	y	$x \oplus y$	$x + y$	$\neg(x + y)$	$x \equiv y$	$\neg y$	$x \mid \neg y$	$\neg x$	$\neg x \mid y$	$\neg(x \& y)$	1
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	1	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

设 $f(x, y)$ 与 $g(x, y)$ 为表2.1中的两列，如果当每一行的 $f(x, y)$ 值为1时，对应的 $g(x, y)$ 也是1，那么就可以说不等式 $f(x, y) \leq^u g(x, y)$ 对于所有的 (x, y) 都成立。这一规律显然也能推广至字组并列式的各种逻辑操作上。大部分此类不等式都是很容易就能推算出来的，像 $(x \& y) \leq^u x \leq^u (x + \neg y)$ 等。此外，如果发现某一行所对应的两列中，前者是0，后者是1，而在另外一行中，前者是1，后者是0，那么对应的逻辑表达式就不存在不等关系。用上面这些知识，很容易就能判断出不等式 $f(x, y) \leq^u g(x, y)$ 对于每一组二元逻辑函数 f 与 g 是否成立了。

使用这些关系式时要注意：在普通算术中，如果 $x + y \leq a$ 且 $z \leq x$ ，那么 $z + y \leq a$ ，而若将上述不等式中的“+”换成了或运算，这种递推关系就不再成立了。

同时含有逻辑表达式与算术表达式的不等式更有意思，选几个例子列在下面。

- a. $(x + y) \geq^u \max(x, y)$
- b. $(x \& y) \leq^u \min(x, y)$
- c. $(x + y) \leq^u x + y$ 若加法操作未溢出
- d. $(x + y) \geq^u x + y$ 若加法操作溢出
- e. $|x - y| \leq^u (x \oplus y)$

可能除了 $|x - y| \leq^u (x \oplus y)$ 这个式子外，其余几个都不难证明。 $|x - y|$ 的意思就是取 $x - y$ 的绝对值，在无符号数的范围内，该值与算式 $\max(x, y) - \min(x, y)$ 相等。要想证明这一点，可以对 x 和 y 的位数运用数学归纳法（证明其中的递推关系式时，展开等式左侧会比较容易些）。

2.4 绝对值函数

要是你所用的CPU没有计算绝对值的指令，那可以考虑用下面几种方式来计算。这

些算法通常用三、四条指令就能实现，而且不带分支。首先执行 $y \leftarrow x \gg^s 31$ ，然后套用下列式子之一即可：

$$\begin{array}{ll} \text{abs} & \text{nabs} \\ (x \oplus y) - y & y - (x \oplus y) \\ (x + y) \oplus y & (y - x) \oplus y \\ x - (2x \& y) & (2x \& y) - x \end{array}$$

这里的 $2x$ 指的就是 $x + x$ 或 $x \ll 1$ 。

若 CPU 能快速算出一个数与 ± 1 的乘积，那么可以考虑用这个式子求绝对值：

$$((x \gg^s 30) | 1) * x$$

2.5 两数平均值

下列公式 [Dietz] 可以算出两个无符号数的平均值 $\lfloor (x+y)/2 \rfloor$ ，而且还不溢出：

$$(x \& y) + ((x \oplus y) \gg^u 1) \quad (3)$$

而下式则可算出无符号整数式 $\lceil (x+y)/2 \rceil$ 的值：

$$(x | y) - ((x \oplus y) \gg^u 1)$$

若要分别计算带符号整数的两种平均值（一种是出现小数时向下取整，另一种是向上取整），只需把公式中的无符号移位操作换成带符号移位即可。

对于两个带符号的整数，有时可能要计算向 0 取整之后的平均值。想求这种“截尾平均值”[⊖]（在不溢出的情况下）稍微有点难。可以先算出向下取整之后的平均值，然后再修正。修正它的办法是：如果 $x+y$ 的算术值是负奇数，则将结果再加 1。然而当且仅当(3)式运算结果为负时（其中的无符号移位变成带符号移位）， $x+y$ 才会是负数。综上所述，可以得出如下计算方式（在归并了重复的子表达式 $x \oplus y$ 之后，只需 7 个基本 RISC 指令即可）：

$$\begin{aligned} t &\leftarrow (x \& y) + ((x \oplus y) \gg^s 1); \\ t &+ ((t \gg^u 31) \& (x \oplus y)) \end{aligned}$$

有些特例可以用更快的办法算出来。比如假设 x 和 y 都是带符号的非负数，那么只需要算出 $(x+y) \gg^u 1$ 就能知道其平均值了。虽然两数之和可能会溢出，但是溢出的那一

[⊖] truncated average，也就是向 0 取整之后的平均值。其含义是：若平均值是小数，则调整为距离该值最近且更靠近 0 的那个整数。例如 3 和 -8 取平均值，结果是 -2.5，-3 与 -2 这两个整数和它的距离都是 0.5，然而 -2 更靠近 0，所以 3 与 -5 的“截尾平均值”是 -2，而不是 -3。——译者注

位[⊕]仍然保留在存放加法结果的寄存器里，所以只需要执行一次无符号右移，就可以把这个溢出位挪到正确的地方，并且让空出来的那个符号位填上0。

如果 x 和 y 都是无符号整数且 $x \leq y$ ，或是 x 、 y 都是带符号整数且 $x \leq y$ （带符号数的比较），那么平均值就是 $(y - x) \gg 1$ 。这样求出来的平均值是向下取整过的，比如-1与0的平均值是-1。

2.6 符号扩展

这里所说的“符号扩展”（sign extension），意思是先在字组中确定某个位元为符号位，然后再把它的值向左传播，覆盖掉那些位元中原有的值。要完成此操作，通常的做法是：先进行逻辑左移，再进行带符号的右移。然而若是电脑执行这些指令的速度比较慢，或者根本没有这类指令，那么可以考虑用下列算法之一来计算。此处列出将7号位元向左传播所用的三种算式：

$$((x + 0x00000080) \& 0x000000FF) - 0x0000\ 0080$$

$$((x \& 0x000000FF) \oplus 0x00000080) - 0x0000\ 0080$$

$$(x \& 0x0000\ 007F) - (x \& 0x00000080)$$

算式中的“+”也可以用“-”或“⊕”代替。第2个公式特别有用，因为假如你能确定符号位左方那些即将被替换掉的位元[⊖]全都是0，那么其中的按位和操作也可以省了。

2.7 用无符号右移模拟带符号右移操作

如果电脑不支持带符号右移指令，可以考虑使用下列公式来算。第1个公式载于[GM]，第2个公式根据其原理推算而得。这些公式成立的条件都是 $0 \leq n \leq 31$ ，如果电脑有“模64移位”[⊖]指令，那么最后一个公式在 $0 \leq n \leq 63$ 的情况下均成立。如果认为带符号移位操作调整移位量所用的模除算法与逻辑移位操作相同[⊖]，那么不管 n 为何值，最后一个式子都成立。

[⊕] 也就是最左侧的符号位，因为两个带符号的非负数，其符号位肯定都是0，而一旦两数之和溢出，则在表示运算结果的那个字组中，最左侧的符号位肯定是1。——译者注

[⊖] high-order，“高位置的”。一般来说左方的位元权重较高，而且在按照序号称呼字组中的位元时，左侧位元的编号也大，故有此称呼。——译者注

[⊖] mod-64 shift，该指令在移位前，先要把移位的个数（也就是该操作符的右操作数）调整成一个大于等于0且小于64的整数，故有此称呼。——译者注

[⊖] 意思是，如果要实现“模32带符号移位”，那么在公式中也要使用“模32”版本的无符号移位操作，而若要实现“模64带符号移位”，则公式中的无符号移位指令只要是“模64”的才行。只要保证两者相符，那么无论带符号移位操作的移位量是多少，都可以套用这个公式。——译者注

n 若为变量，则实现下述每一条公式需要使用 5 至 6 个基本 RISC 指令。

$$\begin{aligned}
 & ((x + 0x8000\ 0000) \gg^u n) - (0x8000\ 0000 \gg^u n) \\
 & t \leftarrow \gg^u 0x8000\ 0000 \gg^u n; \quad ((x \gg^u n) \oplus t) - t \\
 & t \leftarrow (x \& 0x8000\ 0000) \gg^u n, (x \gg^u n) - (t + t) \\
 & (x \gg^u n) | (- (x \gg^u 31) \ll 31 - n) \\
 & t \leftarrow - (x \gg^u 31); \quad ((x \oplus t) \gg^u n) \oplus t
 \end{aligned}$$

头两个公式中的 $0x8000\ 0000 \gg^u n$ 也可以换成 $1 \ll 31 - n$ 。

如果 n 是常数，那么前两个式子在很多 CPU 中只需 3 个指令就能完成。若 $n = 31$ ，则仅用两个指令即可模拟带符号右移操作： $-(x \gg^u 31)$ 。

2.8 符号函数

下面定义的这个函数称为符号函数 (sign function)，也称正负号函数 (signum function)：

$$\text{sign}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$$

此函数在大部分电脑中仅需 4 个指令即可实现 [Hop]：

$$(x \gg^s 31) | (-x \gg^u 31)$$

假若 CPU 没有带符号右移指令，那么可以根据 2.7 节最后所讲的公式来实现。用无符号右移模拟出来的算式非常对称：

$$-(x \gg^u 31) | (-x \gg^u 31)$$

如果用了比较谓词的话，3 个指令就够了。以下两种办法都能实现：

$$\begin{aligned}
 & (x > 0) - (x < 0), \text{ 或} \\
 & (x \geq 0) - (x \leq 0)
 \end{aligned} \tag{4}$$

最后要提一下 $(-x \gg^u 31) - (x \gg^u 31)$ 这个式子，除了 $x = -2^{31}$ 的情况外，用它也能正确算出绝对值来。

2.9 三值比较函数

三值比较函数略微扩大了符号函数的使用范围，其定义是：

$$\text{cmp}(x, y) = \begin{cases} -1, & x < y, \\ 0, & x = y, \\ 1, & x > y. \end{cases}$$

该函数有带符号与无符号两个版本，除非明确指出，否则本节所讲内容对二者均适用。

通过比较谓词，仅需3个指令即可实现，其计算公式显然是等式(4)的推广：

$$(x > y) - (x \leq y), \text{ 或} \\ (x \geq y) - (x \leq y)$$

在PowerPC^②上，可以用下列指令〔CWG〕计算两个无符号数，这种CPU的进位标记会在减法操作未发生借位时置1。^③

```
subf R5,Ry,Rx      # R5 <-- Rx - Ry. 
subfc R6,Rx,Ry     # R6 <-- Ry - Rx, set carry.
subfe R7,Ry,Rx     # R7 <-- Rx - Ry + carry, set carry.
subfe R8,R7,R5      # R8 <-- R5 - R7 + carry, (set carry).
```

要是只能用基本RISC指令的话，恐怕没有什么好办法能计算这个函数。因为像 $x < y$ ， $x \leq y$ 等比较谓词都要大约5条指令才能实现出来（参见2.12节），从而导致整个算法需要大约12个指令（这还是在考虑到可以将那几个重复的 $x < y$ 与 $x > y$ 运算合并的情况下）。如果非要限定基本的RISC指令，那么采用比较与分支的办法也许更好（因为把重复的比较操作合并之后，最差也只要6条指令就能完成）。

2.10 符号传递函数

符号传递函数（transfer of sign function）在Fortran语言中叫做ISIGN，其定义是：

$$\text{ISIGN}(x, y) = \begin{cases} \text{abs}(x), & y \geq 0 \\ -\text{abs}(x), & y < 0 \end{cases}$$

在大部分电脑上，只需4个指令就能实现该函数（这样算出来的是函数值与 2^{32} 取模之后的结果）：

$t \leftarrow \overset{s}{y} \gg 31;$	$t \leftarrow (\overset{s}{x} \oplus \overset{s}{y}) \gg 31;$
$\text{ISIGN}(x, y) = (\text{abs}(x) \oplus t) - t$	$\text{ISIGN}(x, y) = (x \oplus t) - t$
$= (\text{abs}(x) + t) \oplus t$	$= (x + t) \oplus t$

2.11 将值为0的位段解码为2的n次方

有些时候，某个值如果为0或负数是没有意义的，在这种情况下，就可以拿0来表示

② PowerPC (Performance Optimization With Enhanced RISC-Performance Computing, 有时简称PPC) 是一种精简指令集(RISC)架构的CPU，其基本的设计源自IBM的POWER架构。POWER是1991年由Apple、IBM、Motorola组成的AIM联盟所研发，而PowerPC是AIM联盟平台的一部分。详情参见：<http://zh.wikipedia.org/wiki/PowerPC>。——译者注

③ 原文为“carry”是“not borrow”。在加法操作中，carry flag是表示运算结果是否需要进位的标志，但是在执行减法操作时，计算机却有两种不同的解读方式。有些会在减法需要借位时将此标志置1，而另外一些则相反，在不需借位时置1。PowerPC就属于后者，故作者有此说。详情参见：http://en.wikipedia.org/wiki/Carry_flag。——译者注

2^n ，而让其他非零值保持其原意。比如在 PowerPC 的多字节常数加载指令^②中，加载长度就是用 5 个位元来表示的。载入长度为 0 的常数是没有意义的，而载入一个 32 字节常数还有些用。此时可以用 0 到 31 来表示待加载常数所占的字节数分别是 1 至 32 个。然而如果换一种方式，“用 0 来表示 32”，那么一旦处理器必须实现那种用寄存器中变量来表示常数长度的指令时（例如 PowerPC 的 lswx 指令^③），所需的逻辑就会较上一个方案更简单些，因为可以直接使用二进制编码，而不用再加 1 了。

要把一个 1 至 2^n 之间的数编码为“以 0 表示 2^n ”的位段，只需要把它和 $2^n - 1$ 取掩码就可以了。而解码时如果不想用“测试加分支”的形式来做，就不那么容易了。不过办法还是有的，此处以长度为 3 的位段为例来演示如何解码。若不考虑可能需要的常数加载指令，则实现下列各式均要用 3 条指令。

$$\begin{array}{lll} ((x-1) \& 7) + 1 & ((x+7) |- 8) + 9 & 8 - (-x \& 7) \\ ((x+7) \& 7) + 1 & ((x+7) | 8) - 7 & -(-x) - 8 \\ ((x-1) |- 8) + 9 & ((x-1) \& 8) + x & \end{array}$$

2.12 比较谓词

“比较谓词”（comparison predicate）是一个用于比较两数大小的函数，它的运算结果只占一个位元：若比较关系成立（true），则值为 1；若不成立（false），则值为 0。下面列出一些用无分支表达式实现出来的比较谓词，这些式子都会将运算结果放在符号位中。如果想把结果换算成某些语言（例如 C）所用的 1/0 形式，则将运算结果视为无符号数，并右移 31 位^④即可，而要将其转换成另外一些语言（如 Basic）所需的一 1/0 形式，那就在执行完代码之后把它带符号右移 31 位。

如果计算机用的 CPU 架构是 MIPS^⑤或书中所描述的 RISC 模型等，那么这些公式就没多大用处了。因为此类指令集都有比较指令，能够直接算出很多比较谓词，并且把结果以 0/1 值的形式放在通用寄存器中。

$$x = y : \quad \text{abs}(x - y) - 1$$

- ② load string word immediate，助记符为 lswi。详情参考：<http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.aixassem/doc/alangref/lswi.htm>。——译者注
- ③ load string word indexed，助记符为 lswx，详情参考：<http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.aixassem/doc/alangref/lswx.htm>。——译者注
- ④ 原书写为 shift right of 31，但是若 x 、 y 均为带符号数且比较结果为 1 时，使用 C 语言的右移操作会对其符号位进行扩展，而由于运算结果中的“1”处于符号位，所以扩展之后的结果便成了 -1，与文中要求不符。故译文据此略作修改。——译者注
- ⑤ MIPS 为 Microprocessor without Interlocked Pipeline Stages 的缩写，是一种采用 RISC 指令集的处理器架构，广泛用在电子产品、网络设备、个人娱乐装置与商业装置上。详情参见：http://zh.wikipedia.org/wiki/MIPS_架构。——译者注

	$\text{abs}(x - y + \text{0x8000 000})$
	$\text{nlz}(x - y) \ll 26$
	$-(\text{nlz}(x - y) \overset{u}{\gg} 5)$
	$\neg(x - y \mid y - x)$
$x \neq y:$	$\text{nabs}(x - y)$
	$\text{nlz}(x - y) - 32$
	$x - y \mid y - x$
$x < y:$	$(x - y) \oplus [(x \oplus y) \& ((x - y) \oplus x)]$
	$(x \& \neg y) \mid ((x = y) \& (x - y))$
	$\text{nabs}(\text{doz}(y, x))$ [GSO]
$x \leq y:$	$(x \mid \neg y) \& ((x \oplus y) \mid \neg(y - x))$
	$((x \equiv y) \overset{s}{\gg} 1) + (x \& \neg y)$ [GSO]
$x \overset{u}{<} y:$	$(\neg x \& y) \mid ((x \equiv y) \& (x - y))$
	$(\neg x \& y) \mid ((\neg x \mid y) \& (x - y))$
$x \overset{u}{\leq} y:$	$(\neg x \mid y) \& ((x \oplus y) \mid \neg(y - x))$

如果电脑中有直接能算出绝对值相反数的指令，那上面的公式用起来就方便多了。公式中用“nabs”来表示此函数。和绝对值函数不同，它不会溢出。要是电脑中没有能直接求 nabs 的指令，但有常用的求绝对值（abs）指令，那么就可以用 $-\text{abs}(x)$ 来计算 nabs(x)。在 x 为负极值时，这么做会溢出两次，但结果却是对的。（此处假设负极值的绝对值是其本身[⊕]）由于有些电脑既没有 abs 指令，也没有 nabs 指令，所以书中还给出了不需要此类指令的算法。

“nlz”函数的值是其自变量中前导 0 的个数。“doz”函数（“求差或取零”函数，difference or zero）的定义在第 46 页。想要算出 $x > y$ 和 $x \geq y$ 等比较谓词的值，只需将 $x < y$ 与 $x \leq y$ 等公式中的 x 、 y 互换即可。公式中凡出现 x 、 y 、 $x - y$ 等值与 **0x8000 0000** 的加法操作，均可用任意能够反转最高位的指令代替。

由于比较谓词 $x < y$ 的值与 $x/2 - y/2$ 的符号位相符，而且这个减法算式又不会溢出，所以可由此论述推导出另外一组公式。为了防止移位操作把关键信息丢掉，还需酌情对运算结果减 1 以修正其值。此算法公式如下：

$$\begin{aligned} x < y: \quad & (x \overset{s}{\gg} 1) - (y \overset{s}{\gg} 1) - (\neg x \& y \& 1) \\ x \overset{u}{<} y: \quad & (x \overset{u}{\gg} 1) - (y \overset{u}{\gg} 1) - (\neg x \& y \& 1) \end{aligned}$$

上面这两种算法需要 7 个指令（如果按位与和按位取反能用 1 个指令来完成，那就是

[⊕] 假如字组的位宽是 32，那么负极值就是 -2^{31} ，而书中介绍的 $-\text{abs}$ 算法若要成立，必须假设 -2^{31} 的绝对值仍然是 -2^{31} 。——译者注

6个), 和刚才讲的那组公式比起来并不算好(刚才那组公式所需的指令数, 会依所用指令集中逻辑指令的丰富程度而变, 范围在5到7个之间)。

上述公式中用到了 nlz 函数的那个算法是由〔Shep〕给出的, 他计算 $x=y$ 这个谓词时的算式特别有用, 因为只要稍加修改, 就可以变成下面这个样子。新算式只用3条指令就可以把比较谓词的值化成1/0形式:

$$\text{nlz}(x-y) \overset{u}{\gg} 5$$

带符号数与0的比较也很常用, 所以这里有必要特别讲一下。下面这些公式都可以计算比较谓词, 它们差不多都是从上面那一套公式直接推出来的, 而且运算结果也是放在符号位里面。

$x=0:$	$\text{abs}(x)-1$
	$\text{abs}(x+0x8000 0000)$
	$\text{nlz}(x) \ll 26$
	$-(\text{nlz}(x) \overset{u}{\gg} 5)$
	$\neg(x x)$
	$\neg x \& (x-1)$
$x \neq 0:$	$\text{nabs}(x)$
	$\text{nlz}(x)-32$
	$x x$
	$(x \overset{u}{\gg} 1)-x$ [CWG]
$x < 0:$	x
$x \leq 0:$	$x x(x-1)$
	$x \neg-x$
$x > 0:$	$x \oplus \text{nabs}(x)$
	$(x \overset{s}{\gg} 1)-x$
	$\neg x \& \neg x$
$x \geq 0:$	$\neg x$

把无符号比较谓词中的操作数向上偏移 2^{31} , 就得到了带符号版本的比较谓词, 其逆向变换依然成立[⊖]。所以可得如下二式:

$$\begin{aligned} x < y &= x + 2^{31} \overset{u}{<} y + 2^{31} \\ x \overset{u}{<} y &= x - 2^{31} \overset{u}{<} y - 2^{31} \end{aligned}$$

对于判断 \leq 和 \leq 关系的比较谓词来说, 以上关系式仍然适用。这些算式中 2^{31} 前面的

⊖ 在Java语言中比较无符号数时可用此式, 因为Java语言没有无符号型整数。

运算符，不论是加号、减号，还是按位异或，结果都一样，因为这么做只是为了反转符号位而已。如果CPU支持基本RISC指令集中的“带移位常量加法”指令（add immediate shifted），那么就不用再多花两条指令把 2^{31} 这个常数加载到寄存器中了。

当 x 与 y 正负号相同时， $x < y = x \overset{u}{<} y$ ，而两者正负号不同时， $x < y = x \overset{u}{>} y$ [Lamp]，基于这一原理，可以推出另一种用无符号比较谓词来比较带符号数的方法。与刚才一样，逆向变换也成立。于是可以得出两个公式：

$$\begin{aligned}x < y &= (x \overset{u}{<} y) \oplus x_{31} \oplus y_{31} \\x \overset{u}{<} y &= (x < y) \oplus x_{31} \oplus y_{31}\end{aligned}$$

其中 x_{31} 和 y_{31} 分别表示 x 与 y 的符号位。同样的关系式也适用于 \leq 和 \leq^u 等比较谓词。

任选一个除了 $=$ 和 \neq 之外的比较谓词，都可以根据上述各种关系式用其他形式的比较谓词把它表示出来，而且在大多数电脑上最多只需3个指令。因为 $x \leq y$ 这个比较谓词实现起来很容易（它的值也就是执行 $y - x$ 之后的进位标志），所以我们就以它为例来推算其他比较谓词：

$$\begin{aligned}x < y &= \neg(y + 2^{31} \overset{u}{\leq} x + 2^{31}) \\x \leq y &= x + 2^{31} \overset{u}{\leq} y + 2^{31} \\x > y &= \neg(x + 2^{31} \overset{u}{\leq} y + 2^{31}) \\x \geq y &= y + 2^{31} \overset{u}{\leq} x + 2^{31} \\x \overset{u}{<} y &= \neg(y \overset{u}{\leq} x) \\x \overset{u}{>} y &= \neg(x \overset{u}{\leq} y) \\x \overset{u}{\geq} y &= y \overset{u}{\leq} x\end{aligned}$$

2.12.1 利用进位标志求比较谓词

若是计算机很容易就能把进位标志的值放到一个通用寄存器里，那么可以用该标志准确判断出一些比较谓词的值。下面列出一些此类关系式。其中carry(expression)这个写法表示expression中最外层运算的标志位。此处假定进位标志是加法器用 $x + \bar{y} + 1$ 的方式来计算减法操作 $x - y$ 时产生的，其值与减法的“借位”情况相反[⊖]。

$$\begin{aligned}x = y : \quad &\text{carry}(0 - (x - y)), \text{or } \text{carry}((x + \bar{y}) + 1), \text{or} \\&\text{carry}((x - y - 1) + 1)\end{aligned}$$

[⊖] 原文为the complement of “borrow”，意思是，如果减法操作没有发生借位，则进位标志是1，若发生借位，则该标志位为0。——译者注

$x \neq y:$	carry($(x - y) - 1$), i. e., carry($(x - y) + (-1)$)
$x < y:$	\neg carry($(x + 2^{31}) - (y + 2^{31})$), or \neg carry($x - y \oplus x_{31} \oplus y_{31}$)
$x \leq y:$	carry($(y + 2^{31}) - (x + 2^{31})$), or carry($y - x \oplus x_{31} \oplus y_{31}$)
$x^u < y:$	\neg carry($x - y$)
$x^u \leq y:$	carry($y - x$)
$x = 0:$	carry($0 - x$), or carry($\bar{x} + 1$)
$x \neq 0:$	carry($x - 1$), i. e., carry($x + (-1)$)
$x < 0:$	carry($x + x$)
$x \leq 0:$	carry($2^{31} - (x + 2^{31})$)

若想求 $x > y$ 的值, 只需把 $x \leq y$ 的结果取反就好, 判断其他包含“大于”的关系式是否成立时也是如此: 对与之相反的比较谓词求补[⊖]即可。

在 IBM RS/6000 计算机和其兄弟机型 PowerPC 系列电脑上, 都采用 GNU Super-optimizer[⊕]来求谓词表达式的值 [GK]。RS/6000 电脑不仅具备计算 $\text{abs}(x)$ 、 $\text{nabs}(x)$ 、 $\text{doz}(x, y)$ 函数的指令, 而且还有各种形式的加减法指令, 这些指令都能把进位标志考虑在内。有人发现, RS/6000 最多用 3 条基本指令 (也就是只花 1 个 CPU 周期就能执行完的指令) 就能算出每一种整数谓词表达式的值, 这个结果连此计算机的设计师都觉得吃惊。这里所说的“每一种”, 包括 6 种判断两个带符号数的比较谓词, 以及 4 种判断两个无符号数的比较谓词。上述比较谓词的第 2 个操作数都是 0, 而且结果均是 1/0 或 $-1/0$ 的形式。在缺少 $\text{abs}(x)$ 、 $\text{nabs}(x)$ 及 $\text{doz}(x, y)$ 的 PowerPC 中, 实现上述各种谓词表达式最多需要 4 条指令。

2.12.2 计算机如何设置比较谓词

大多数电脑都会用某种方式将整数比较谓词的值化为 1 个位元。有些 CPU 把表示比较结果的位元存放在“条件寄存器”(condition register)中, 而另外一些(比如本书用到的 RISC 模型)则把它放在通用寄存器里。不管采用那种实现, 都需要有一套机制把比较谓词的两个操作数相减, 并对包含运算结果的那些二进制位做几个逻辑操作, 以提取出一个表示比较结果的位元。

下面列出了求各种比较谓词所用的逻辑操作。假定电脑用 $x + \bar{y} + 1$ 来计算 $x - y$, 而且运算完毕后可从结果中得知这几个量的值:

C_0 : 计算最高位时产生的进位值。

[⊕] 由于比较谓词的值只有一个位元, 非 0 即 1, 所以这里“求补”的意思就是把 1 变为 0, 把 0 变为 1。——译者注

[⊖] 是一个能够实现超级优化技术(Superoptimization)的程序, 详情参见: <http://en.wikipedia.org/wiki/Superoptimization> 及 <http://ftp.gnu.org/gnu/superopt/>。——译者注

C_i : 计算次高位时，带给最高位的进位值。

N : 运算结果的符号位。

Z : 若运算结果与 C_o 求异或后，每一位都是 0，则该值为 1，否则为 0。

有了上述各量，就可以用如下布尔代数操作来求比较谓词了（两值毗邻表示求其逻辑与，而 + 代表逻辑或）：

V :	$C_i \oplus C_o$ (两个带符号数相减是否溢出)
$x = y$:	Z
$x \neq y$:	\bar{Z}
$x < y$:	$N \oplus V$
$x \leq y$:	$(N \oplus V) + Z$
$x > y$:	$(N \equiv V)\bar{Z}$
$x \geq y$:	$N \equiv V$
$x^u < y$:	$\overline{C_o}$
$x^u \leq y$:	$\overline{C_o} + Z$
$x^u > y$:	$C_o \bar{Z}$
$x^u \geq y$:	C_o

2.13 溢出检测

“溢出”是指算术操作的结果太大或太小，以致无法将其正确表示到目标寄存器里。本节讨论一些程序员可能会用到的溢出检测法，它们都无需使用 CPU 中现成的“状态位”(Status Bit)。这么做意义很大，因为有些 CPU (例如 MIPS 架构的) 没有这种状态位，而且即便有，某些高级语言也很难或根本无法访问它们。

2.13.1 带符号的加减法

当今的 CPU 在计算整数加减法时如果发生溢出，则总是会丢弃运算结果的最高位，并且把加法器中较低的那些位元照原样存起来。当且仅当两操作数符号相同但二者之和却与其符号不同时，带符号整数加法才会溢出。让人觉得奇怪的是：哪怕加法器在运算时把进位也带进来，其结果仍符合上述规律。也就是说，在计算 $x + y + 1$ 时，只要 x 、 y 的符号不同，便怎么也溢出不了。这一规律对计算多字带符号数的加法很重要，因为此种加法在最后一步需要把两个字组与从低位带过来的进位一并相加，而这个进位可能是 0，也有可能是 +1。

要证明上述加法规律，现在假定两个被加数 x 、 y 均为一个字组长的带符号整数，而

c （执行运算前的进位标志[⊕]）是 0 或 1，为简化证明过程，规定字组位宽是 4。若 x 与 y 符号不同，则：

$$\begin{aligned} -8 &\leqslant x \leqslant -1, \text{ 且} \\ 0 &\leqslant y \leqslant 7 \end{aligned}$$

若 x 为非负数而 y 为负数，则其取值范围与之类似。不管哪种情况，我们都把两个不等式相加，并且将 c 的值可取 0 或 1 这一点考虑在内[⊕]，于是得出：

$$-8 \leqslant x + y + c \leqslant 7$$

4 位带符号数显然能够容纳上述取值范围，因此两操作数符号不同时是不会溢出的。

若 x 与 y 符号相同，则分两种情况：

(a)	(b)
$-8 \leqslant x \leqslant -1$	$0 \leqslant x \leqslant 7$
$-8 \leqslant y \leqslant -1$	$0 \leqslant y \leqslant 7$

在上述两种情况下，可分别推出：

(a)	(b)
$-16 \leqslant x + y + c \leqslant 1$	$0 \leqslant x + y + c \leqslant 15$

若求和结果在下述取值范围内，则无法表示为 4 位带符号整数，于是就会溢出：

(a)	(b)
$-16 \leqslant x + y + c \leqslant -9$	$8 \leqslant x + y + c \leqslant 15$

式 (a) 所述的情况，相当于在容纳求和结果的 4 个位元中，最高位是 0，而这与 x 、 y 的符号相反。式 (b) 所述的情况，则相当于在容纳求和结果的 4 个位元中，最高位是 1，这也和 x 、 y 的符号相反。

在多字整数的减法操作中，需要计算 $x - y - c$ ，此处 c 的值还是 0 或 1，其中 1 表示由低位字组产生的借位 (borrow-in)。采用与上述分析过程类似的方式，也可证明：当且仅当 x 与 y 符号相反，而 $x - y - c$ 的符号又与 x 相反（也可以说成“与 y 相同”）时， $x - y - c$ 才会溢出。

基于此规律，可以用以下表达式判别溢出谓词，运算结果存放在符号位中。如果套用公式后再右移或带符号右移 31 位，那么就可以将其化为 1/0 或 -1/0 形式的值。

$x + y + c$	$x - y - c$
$(x \equiv y) \& ((x + y + c) \oplus x)$	$(x \oplus y) \& ((x - y - c) \oplus x)$
$((x + y + c) \oplus x) \& ((x + y + c) \oplus y)$	$((x - y - c) \oplus x) \& ((x - y - c) \equiv y)$

如果用第 1 列的第 2 种形式或第 2 列的第 1 种形式来判断溢出（也就是不含按位等值操作 \equiv 的那两个公式），那么在本书所定义的基本 RISC 指令集范围内，除去计算 $x + y + c$

[⊕] 本书中用 carry-in 表示这个概念，如果执行的是带进位的加法 (add with carry)，则在求和时还要将该值一并加入。——译者注

[⊕] 也就是将 $-8 \leqslant x \leqslant -1$ 、 $0 \leqslant y \leqslant 7$ 、 $0 \leqslant c \leqslant 1$ 这三个不等式加起来。——译者注

或 $x - y - c$ 所需的指令外，仅需3条即可。若想用分支代码来处理溢出，那么再加一条负分支（branch if negative）指令即可。

执行代码时若启用了溢出中断，则程序员可能想在不引发溢出的情况下预判某个加减法操作是否会溢出。以下列出无分支的实现方式：

$$\begin{array}{ll} x + y + c & x - y - c \\ z \leftarrow (x \equiv y) \& 0x8000\ 0000 & z \leftarrow (x \oplus y) \& 0x8000\ 0000 \\ z \& (((x \oplus z) + y + c) \equiv y) & z \& (((x \oplus z) - y - c) \oplus y) \end{array}$$

在左边一栏中，如果 x 与 y 符号相同，则 $z = 0x8000\ 0000$ ，若符号不同，则 $z = 0$ 。在第2个表达式的加法操作中， $x \oplus z$ 与 y 的符号相反，所以肯定不会溢出。若 x 与 y 都是非负数，则当且仅当 $(x - 2^{31}) + y + c \geq 0$ 时，也就是 $x + y + c \geq 2^{31}$ 时，第2个表达式的值才会是1。而其中的 $x + y + c \geq 2^{31}$ 也就是 $x + y + c$ 发生溢出的条件。若 x 与 y 都是负数，则当且仅当 $(x + 2^{31}) + y + c \leq 0$ 时，也就是 $x + y + c \leq -2^{31}$ 时，第2个表达式的值才会是1。而其中的 $x + y + c \leq -2^{31}$ 也就是此时能发生溢出的条件。为了保证在 x 与 y 符号不同时能够得到正确结果（也就是让符号位的值是0），还要与 z 按位取和才行。对于减法操作的溢出判断（也就是右侧那一栏）也可以按相似的思路分析。实现上述代码需要9条RISC指令。

要是可以使用加法操作的进位标志，那它也许能用来计算带符号数的溢出谓词。虽说未必真的如此，但是通过这一思路可以构想出下面这种算法。

如果 x 为带符号整数，则 $x + 2^{31}$ 可以用无符号整数表示出来，方法就是把 x 的最高位反转。若 $x + y \geq 2^{31}$ ，也就是 $(x + 2^{31}) + (y + 2^{31}) \geq 3 \cdot 2^{31}$ 时，两数之和会发生正向（positive direction）溢出。而溢出条件中的后一种写法，实际上就是说，在无符号数加法中，进位标志（两数之和大于等于 2^{32} ）与求和结果的最高位都是1。同理，当进位标志与求和结果的最高位均是0时，会发生负向溢出。

于是可以得出一种检测带符号数加法是否溢出的算法：

计算 $(x \oplus 2^{31}) + (y \oplus 2^{31})$ ，设求和结果为 s ，进位标志为 c 。

当且仅当 c 与 s 的最高位相同时，才会发生溢出。

对于两个无符号数，这么算出来的求和结果是正确的，因为同时反转两个操作数的最高位，并不会改变它们的和。

判断减法是否溢出也可以用此算法，只是要把第1步中的加号换成减号。此时假定计算机通过 $x + \bar{y} + 1$ 来计算 $x - y$ 的值，并且进位标志的值也是在计算 $x + \bar{y} + 1$ 的过程中产生的。对于两个带符号数的减法来说，这种算法计算出来的差也是正确的。

这几个公式也许很有意思，不过在大部分电脑上，它们的效率都低于那些根本不用进位标志的式子（比如用 $(x \equiv y) \& (s \oplus x)$ 判断加法是否溢出，用 $(x \oplus y) \& (d \oplus x)$ 来判断减法是否溢出，其中 s 、 d 分别表示 x 、 y 的和与差）。

2.13.2 计算机执行带符号数的加减法时如何设置溢出标志

大部分CPU都会根据“次高位向最高位带入的进位是否与最高位带出的进位不同”这一逻辑来判断是否需要设置“溢出”标志。说来也怪，如果假定 $x - y$ 是通过 $x + \bar{y} + 1$ 算出来的，那么这条标准还真能准确判断出加减法是否会溢出。而且，无论是做带进位的加减法还是不带进位的加减法，这条规律都成立。若要在软件中计算带符号数的溢出谓词，那么用这条规律恐怕推算不出什么好办法，虽说如此，不过它还是带来了一种简便的方式，可用于计算从次高位向最高位带入的进位。对于加法减法操作来说，运算完下列表达式之后（其中 c 为0或1），符号位中的值就是在运算过程中由次高位带入最高位中的进位/借位：

进位	借位
$(x + y + c) \oplus x \oplus y$	$(x - y - c) \oplus x \oplus y$

其实在该表达式的运算结果中，序号为 i 的位元值就是在计算过程中由其右侧位元所带入位置 i 的进位或借位。

2.13.3 无符号数的加减法

下列无分支代码可以计算无符号数加减法的溢出谓词，运算结果放在符号位中。带右移的那个表达式只有在 $c=0$ 时才值得用[⊖]。表达式里中括号之内的部分，用于计算从最低有效位中生成的进位或借位。

$$\begin{aligned}
 & x + y + c, \text{无符号数的加法} \\
 & (x \& y) | ((x|y) \& \neg(x+y+c)) \\
 & (x \overset{u}{\gg} 1) + (y \overset{u}{\gg} 1) + [(x \& y) | ((x|y) \& c)) \& 1] \\
 & x - y - c, \text{无符号数的减法} \\
 & (\neg x \& y) | ((x \equiv y) \& (x - y - c)) \\
 & (\neg x \& y) | ((\neg x|y) \& (x - y - c)) \\
 & (x \overset{u}{\gg} 1) - (y \overset{u}{\gg} 1) - [(\neg x \& y) | ((\neg x|y) \& c)) \& 1]
 \end{aligned}$$

通过比较操作，可以用极为简单的公式判断出无符号加减法是否溢出〔MIPS〕。对于无符号加法来说，只要二数之和小于（无符号比较）其中任何一个操作数，那就表明溢出了（同时也意味着最高位会带出进位）。这个算法与另外几个和它相似的公式都列在下面。可惜的是，没办法把进位标志或借位标志以变量 c 的形式纳入到公式中。这样的话，程序在运算前必须先测试 c ，并根据其值是0还是1来套用不同类型的比较公式。

[⊖] 因为若 c 不为0，则不带右移的算法比它所需的指令数少。——译者注

$x + y,$ 无符号数加法	$x + y + 1,$ 无符号数加法	$x - y,$ 无符号数加法	$x - y - 1,$ 无符号数加法
$\neg x \overset{u}{<} y$ $x + y < x$	$\neg x \overset{u}{\leq} y$ $x + y + 1 \leq x$	$x \overset{u}{<} y$ $x - y > x$	$x \overset{u}{\leq} y$ $x - y - 1 \geq x$

每种情况下的第1个公式，都可以在执行加减法前先判断出是否溢出，这就提供了一个在不引发溢出情况下的检测办法。而第2个公式则必须先执行完可能产生溢出的加减法操作，然后才能得知结果。

似乎没有一套与之相似的简单公式，让我们能通过比较操作求出带符号数的溢出谓词。

2.13.4 乘法

对于乘法来说，溢出就意味着操作结果无法用32位元表达出来（不管是带符号数乘法，还是无符号数乘法，其乘积总可以用64位元来表示）。若是能获取乘积的高32位，那么溢出检测就很容易了。在表示乘积的64位元中，设其左右两端的32位元分别为 $hi(x \times y)$ 与 $lo(x \times y)$ ，则可用下列公式求溢出谓词〔MIPS〕：

$x \times y$, 无符号数乘法 $hi(x \times y) \neq 0$	$x \times y$, 带符号数乘法 $hi(x \times y) \neq (lo(x \times y) \overset{s}{\gg} 31)$
--	---

还有一种检测乘法溢出的方法，是通过对乘积做除法而得出的。使用这种方法时需注意，不要让0做除数，而且用此方法判断带符号乘法是否溢出显得更为复杂。若下列表达式为真，则表明发生溢出了：

带符号数 $z \leftarrow x * y$ $y \neq 0 \And z \div y \neq x$	无符号数 $z \leftarrow x * y$ $(y < 0 \And x = -2^{31}) \mid (y \neq 0 \And z \div y \neq x)$
---	---

复杂之处就在于，如果 $x = -2^{31}$ 而 $y = -1$ ，那么此时乘法操作将溢出。而CPU却很可能会认为此操作的结果是 -2^{31} ，这样一来，除法操作也溢出了，其结果（在某些电脑上）可能是任意值。因此，必须单独检测这种情形，公式中的 $y < 0 \And x = -2^{31}$ 就是用来执行此判断的。上述表达式还用“条件与”（conditional and）操作符来预防0做除数的情况（在C语言中可用`&&`操作符）。

不执行乘法操作（即在不引发溢出的情况下）也可以判定是否会溢出。对无符号整数来说，当且仅当 $xy > 2^{32} - 1$ ，也就是 $x > ((2^{32} - 1)/y)$ 时，乘积才会溢出。由于 x 为整数，故可将条件改写为 $x > \lfloor (2^{32} - 1)/y \rfloor$ 。用计算机算术表示，就是：

$$y \neq 0 \And x > \lfloor (0xFFFFFFFF \div y) \rfloor$$

对于带符号数来说，想判断 $x * y$ 是否溢出就不那么简单了。假如 x 与 y 符号相同，那么当且仅当 $xy > 2^{31} - 1$ 时，乘积才溢出，而如果 x 与 y 符号不同，则当且仅当 $xy < -2^{31}$ 时，乘积才会溢出。可以按照表2.2中列出的不同情况来判断，该表用到了带符号数的除法操作。因为要分4种情况，所以这种测试很难实现。正是由于除法溢出以及无法

用带符号数表示 $+2^{31}$ 这两个问题，才导致表中的4个表达式不好统一。

表 2.2 检测带符号乘法是否溢出所用的表达式

	$y > 0$	$y \leq 0$
$x > 0$	$x > 0x7FFFFFFF \div y$	$y < 0x8000 0000 \div x$
$x \leq 0$	$x < 0x8000 0000 \div y$	$x \neq 0 \wedge y < 0x7FFFFFFF \div x$

要是能用无符号除法的话，那么测试过程就可以简化了。 x 与 y 的绝对值是可以用无符号数正确表示出来的。由此，我们可以把4种情况一起合并为下述算法。如果 x 与 y 符号相同，则变量 $c=2^{31}-1$ ，否则 $c=2^{31}$ 。

```

c←((x≡y)⟩⟩31)+231
x←abs(x)
y←abs(y)
y≠0 → (c÷y)

```

前导0计数指令(the number of leading zero instruction)可用来估算 $x * y$ 是否溢出，并且可调整估算结果，以准确判断溢出情况。首先来看无符号数乘法。如果 x 与 y 都是32位元的无符号量，并且分别具有 m 及 n 个前导0，那么很明显，在表示两者乘积的64位元中，前导0的个数要么是 $m+n$ ，要么是 $m+n+1$ (如果 $x=0$ 或 $y=0$ ，则前导0的个数为64)。如果64位乘积的前导0个数小于32，则表明发生溢出了。因此：

$\text{nlz}(x)+\text{nlz}(y) \geq 32$: 乘法运算肯定不会溢出。

$\text{nlz}(x)+\text{nlz}(y) \leq 30$: 乘法运算必定溢出。

如果 $\text{nlz}(x)+\text{nlz}(y)=31$ ，那么有时会溢出，有时则不会。在这种情况下，可以用 $t=x \lfloor y/2 \rfloor$ 来判断。计算 t 时不会溢出。如果 y 是偶数， xy 就是 $2t$ ，若是奇数，就是 $2t+x$ ，所以当 $t \geq 2^{31}$ 时， xy 的乘积就会溢出。上述思路可以归结为一种计算 xy 的方法，并且让程序在发生溢出时转入名为“overflow”的分支。该方案的代码如图2.2所示。

对于带符号数的乘法，我们可以在操作数为非负时计算前导0的个数，而在操作数为负数时计算前导1的个数，并由此推算出部分结果。设

$$\begin{aligned} m &= \text{nlz}(x) + \text{nlz}(\bar{x}), \text{且} \\ n &= \text{nlz}(y) + \text{nlz}(\bar{y}) \end{aligned}$$

则可得出如下两条论断：

$m+n \geq 34$: 乘法操作绝对不会溢出。

```

unsigned x, y, z, m, n, t;

m = nlz(x);
n = nlz(y);
if (m + n <= 30) goto overflow;
t = x*(y >> 1);
if ((int)t < 0) goto overflow;
z = t*2;
if (y & 1) {
    z = z + x;
    if (z < x) goto overflow;
}
// z is the correct product of x and y.

```

图 2.2 判断无符号乘法是否溢出

$m+n \leq 31$ ：乘法操作必定溢出。

$m+n$ 为 32 或 33 时需要具体分析。如果 $m+n=33$ ，那么只有当参与乘法的两数均为负，且正确的乘积恰好是 2^{31} （这个值无法用 32 位带符号数正确表示出来，会被计算机当成 -2^{31} ）时，才表明发生了溢出。所以，可通过判断乘积的符号是否正确来得知溢出情况（也就是说，如果 $x \oplus y \oplus (x * y) < 0$ ，则乘法溢出）。若 $m+n=32$ ，则不太容易判断是否溢出。

此处不打算深究这个问题了，只是想再补充一句：带符号乘法的溢出还可以用 $\text{nlz}(\text{abs}(x)) + \text{nlz}(\text{abs}(y))$ 判断，然而和上面一样，这个式子也有两种尚需讨论的情况（分别是当求和结果为 31 或 32 时）。

2.13.5 除法

对于带符号数除法 $x \div y$ 来说，若此表达式结果为真，则发生溢出：

$$y = 0 \mid (x = 0x8000\ 0000 \& y = -1)$$

大部分计算机在遇到 $0 \div 0$ 这种结果为定义的情况时，都会产生溢出（或中断）信号。

如果算上最后处理溢出的分支，那么直接用代码来计算这个表达式需要 7 条指令，其中 3 条是分支指令。似乎没有特别好的办法能改进这个算法，不过可以试试下面这种方案：

$$[\text{abs}(y \oplus 0x8000\ 0000) \mid (\text{abs}(x) \& \text{abs}(y = 0x8000\ 0000))] < 0$$

上面这个式子意思是说，先计算中括号内的大表达式，如果结果小于 0，就转入分支。若是计算机支持算式中用到的指令，并且能够迅速“和 0 比较”，那么算上常数加载和最后的分支，大约需要 9 条指令。

还有一种办法，先用下式算出一个 z 值（在很多电脑中需要 3 条指令）：

$$z \leftarrow (x \oplus 0x8000\ 0000) \mid (y + 1)$$

然后若发现 $y = 0 \mid z = 0$ ，就直接转入分支，否则用下面任意一个式子检测溢出：

$$((y \mid y) \& (z \mid -z)) \geq 0$$

$$(\text{nabs}(y) \& \text{nabs}(z)) \geq 0$$

$$((\text{nlz}(y) \mid \text{nlz}(z)) \gg 5) \neq 0$$

如果电脑支持上面用到的那些指令，则上述三种算法所需的总指令数分别为 9 个，7 个，8 个。最后一个式子很适合在 PowerPC 上使用。

对于无符号除法 $x \overset{u}{\div} y$ 来说，当且仅当 $y = 0$ 时，才会溢出。

在某些有“长除法”指令（“long division” instruction，参见 9.4 节）的电脑上，我们可能想要用基本指令来预测此操作会不会溢出。在讨论这个问题时，假定该指令会用双字（doubleword）除以单字[⊖]（fullword），其商也是单字，此外有可能还会产生一个单

[⊖] 原文为 fullword，直译是“全字”，此处就指一个字组，只是为了和双字（doubleword）对称，故加了“full”。——译者注

字的余数。

若除数为 0 或商无法以 32 位表示，则此类除法指令就会溢出。一旦溢出了，那么商和余数通常都是错的。余数不会因为其值太大、无法用 32 个位元表达出来而溢出（因为它从值上看肯定比除数小），所以测试除数是否正确的方式与测试商所用的方法一样。

我们假定电脑有 64 位或 32 位的通用寄存器，并且能够对 64 位的值进行基本操作（诸如移位、加法等）。比方说编译器或许会实现一种双字节整数类型。

无符号的情况比较容易判断：设 x 为双字， y 为单字，当（且仅当）满足下列任意一个条件时， $x \div y$ 的商才会溢出。这两个判别式是等效的。

$$y \neq 0 \quad \& \quad x < (y \ll 32)$$

$$y \neq 0 \quad \& \quad (x \gg^u 32) < y$$

在 32 位机上不需要移位，只比较 y 和存放 x 高 32 位的寄存器即可，而要保证 64 位机结果正确，则必须检查除数 y 是否为一个 32 位的量（例如，判断 $(y \gg^u 32) = 0$ ）。

带符号的情况就更有意思了。首先必须保证 $y \neq 0$ ，而且如果是 64 位机，还要核实 y 的值能够用 32 个位元正确表示出来（检查 $((y \ll 32) \gg^s 32 = y)$ 是否成立）。假设上述条件都满足，那么根据被除数与除数的正负性，可按照下表列出的 4 种情况，准确判断出商是否能够容纳在 32 位元的字组中。表格中的表达式用的是普通算术，而非计算机算术。

每一栏中的关系式都由其上方的关系式推导而来，这些关系式之间互为充要条件。利用 9.1 节定理 D1 中的某些关系式，可以去掉表格中的向下取整函数与向上取整函数。

$x \geq 0, y > 0$	$x \geq 0, y < 0$	$x < 0, y > 0$	$x < 0, y < 0$
$\lfloor x/y \rfloor < 2^{31}$	$\lceil x/y \rceil \geq -2^{31}$	$\lceil x/y \rceil \geq -2^{31}$	$\lfloor x/y \rfloor < 2^{31}$
$x/y = 2^{31}$	$\lceil x/y \rceil > -2^{31} - 1$	$\lceil x/y \rceil > -2^{31} - 1$	$x/y < 2^{31}$
$x < 2^{31}y$	$x/y > -2^{31} - 1$	$x/y > -2^{31} - 1$	$x > 2^{31}y$
	$x < -2^{31}y - y$	$x > -2^{31}y - y$	$-x < 2^{31}(-y)$
	$x < 2^{31}(-y) + (-y)$	$-x < 2^{31}y + y$	

现在演示一下此表的用法。以最左边一栏为例，它适用于 $x \geq 0$ 且 $y > 0$ 的情况，此时商为 $\lfloor x/y \rfloor$ 。要想把它表示为 32 位元的量，必须保证此值小于 2^{31} 。由此可推出：实数 x/y 必须小于 2^{31} ，或者说 x 必须小于 $2^{31}y$ 。要判断这一关系式，可以把 y 左移 31 位，并与 x 相比。

当 x 与 y 的符号不同时，商一般写为 $\lceil x/y \rceil$ ，由于商为负数，所以其最小值是 -2^{31} 。

每一栏中最后一行的比较表达式用的都是同一种比较关系（也就是小于）。鉴于 x 可能会取负极值，所以 3、4 两列中的表达式必须使用无符号比较操作。而前两栏中那些待比较的数值，其最高位均为 0，所以此处也可以执行无符号比较。

这 4 种测试当然可以用条件分支统一起来：只需按情况执行不同的判断算法，然后根据最终比较结果转入处理溢出或不溢出的分支即可。然而我们观察到一个现象：当 y 为负值时，相应的判别式与 y 为正值时的式子相差无几， x 也是这样。因此，可以利用这一

规律减少分支数量，通过使用 x 与 y 的绝对值，进一步合并这4种测试。此外，还要用一种常见的办法来处理第2、3两栏中出现的加法操作。综上所述，可得出下列方案：

$$\begin{aligned}x' &= |x| \\y' &= |y| \\\delta &= ((x \oplus y) \gg^s 63) \& y' \\&\text{if } (x' \ll^u (y' \ll 31) + \delta) \text{ 则 }\{\text{不会溢出}\}\end{aligned}$$

如果在64位机上使用3条指令计算绝对值（参见2.4节），那么此算法只需12条指令，再外加一个分支语句即可。

2.14 加法、减法与乘法的特征码

许多电脑都提供了“特征码”（condition code），用以描述整数算术操作的结果。一般来说，无符号数与带符号数的加法都用一种add指令来做，而不论两个操作数无符号的还是带符号的（两者不能混用），特征码都能反映出操作结果的某些特性。通常可由特征码看出下列属性：

- 计算最高位后是否产生进位（如果是无符号操作，产生进位就表示溢出）。
- 如果将其视为带符号的操作，结果是否溢出。
- 若将运算结果视为一个以2补码表示的32位元带符号整数，那么忽略进位与溢出这两个因素后，它是负值、0，还是正值。

一些老式计算机还会告诉你无限精度结果（infinite precision result，也就是用33个位元来表示两个32位数的加减法操作结果）是整数、负数，还是0。然而此特征不太容易为高级语言编译器所用，于是这种做法就不再流行了。

对于加法运算来说，这12种组合里只有9种情况可能发生。不可能出现的几种是“不进位，溢出，结果大于0”，“不进位，溢出，结果等于0”，“进位，溢出，结果小于0”。这样的话，只需要4个位元就能表达所有特征码的值了。有两种组合非常特殊，只有特定的操作数才能引发：只有将0与其自身相加，才会得出“不进位，不溢出，结果等于0”的特征码，而将负极值与其自身相加，才能出现“进位，溢出，结果等于0”这种特征码。

对于减法操作，需要假定计算机以 $x - y + 1$ 的方式计算 $x - y$ 的值，并且进位标志的产生方式与执行加法指令相同（在这种计算方式下，“进位”在减法操作中的意思与加法刚好相反，进位标志若是1，则表示两数之差可以容纳在32位元的单字中，而进位标志若是0，反倒说明只用1个字组是容不下计算结果的）。于是减法操作只会出现7种特征码。不可能发生的特征码组合除了上面提到的3种外，还有两种是：“不进位，不溢出，结果为0”，“进位，溢出，结果为0”。

如果 CPU 的乘法器[⊕]将两个字组的乘积用双字表示，那么需要两种乘法指令：一个用于带符号数，另一个用于无符号数。（在 4 位机上执行带符号乘法，以十六进制表示，就是： $F \times F = 01$ ，而无符号乘法则为： $F \times F = E1$ 。）因为两个单字的乘积总能放在一个双字中，所以此类指令既不产生进位，也不可能溢出。

对于以单字来存放乘积（也就是取双字结果的低 32 位）的乘法指令来说，如果将操作数与运算结果视为无符号整数，且运算结果无法用一个字组容纳，那么就会产生“进位”；而若将操作数与运算结果视为带符号数，且用带符号的 2 补码无法将运算结果容纳在一个字组中时，则会发生“溢出”。于是又出现了 9 种特征码，不可能出现的那 3 种是：“不进位，溢出，结果大于 0”，“不进位，溢出，结果等于 0”，“进位，不溢出，结果等于 0”。因此，将加法、减法、乘法可能出现的所有特征码一并算上，共有 10 种组合。

2.15 循环移位

下面这两个算式相当简单，让人觉得有些意外的是：它们对大于等于 0 且小于等于 32 的整数都成立，即使进行“模 32 移位”，结果也正确：

循环左移 n 位： $y \leftarrow (x \ll n) | (x \gg (32-n))$

循环右移 n 位： $y \leftarrow (x \gg n) | (x \ll (32-n))$

如果电脑支持双字长移位指令，那么可以用下述办法来实现循环移位：

```
shldi RT,RA,RB,I  
shrdi RT,RA,RB,I
```

这两种指令都会把 RA 与 RB 合起来视为一个双字量，然后根据常量字段 I 的值对其左移或右移。（如果移位量也放在寄存器里，那么这两个指令在大部分 RISC 架构的电脑上就很难实现了，因为那样要读取 3 个寄存器。）左移操作的结果是双字节移位后的高 32 位，而右移操作的结果则是双字节移位后的低 32 位。

使用 shldi 指令，可以将寄存器 Rx 循环左移：

```
shldi RT,Rx,Rx,I
```

同理，循环右移也可以用 shrdi 指令实现。

如果只循环左移 1 位，那么可以将待循环的寄存器与其自身相加，并且算上“首尾循环进位”（end-around carry，也就是把两数求和过程中最高位所产生的进位，加到求和结果的最低位上）。大多数电脑都没有这种指令，不过在很多机子上可以用两条指令实现：（1）把寄存器的内容与其自身相加，如果产生进位，则会存放于状态寄存器中，（2）把进位加到第 1 步算出的结果上。

[⊕] multiplier，一种数字电路元件，可以将两个二进制数相乘。它是由一些更基本的加法器组成的。详情参见：
<http://zh.wikipedia.org/wiki/乘法器>。——译者注

2.16 双字长加减法

2.13.3节介绍了一些判断无符号加减法是否溢出的式子，利用它们很容易就能在不访问CPU进位标志的前提下实现双字长加减法。为了演示双字长加法，此处假设操作数为 (x_1, x_0) 与 (y_1, y_0) ，结果为 (z_1, z_0) 。下标为1者，表示双字中权重较高的32位，为0者，表示权重较低的32位。另外还需假设寄存器的32个位元都参与运算。存放在低位字组中的量，视为无符号数。

$$\begin{aligned} z_0 &\leftarrow x_0 + y_0 \\ c &\leftarrow [(x_0 \& y_0) | ((x_0 \mid y_0) \& \neg z_0)] \gg^u 31 \\ z_1 &\leftarrow x_1 + y_1 + c \end{aligned}$$

上述算法需要9条指令。有些电脑支持无符号数比较操作，并且能把结果以**1**或**0**的形式存放在寄存器中，比如MIPS架构CPU的“SLTU”指令[⊖][MIPS]。在此类电脑上，第2行可以改写为 $c \leftarrow (z_0 <^u x_0)$ ，这样只需要4个指令就够了。

双字长减法 $(x - y)$ 也可以用类似代码实现：

$$\begin{aligned} z_0 &\leftarrow x_0 - y_0 \\ b &\leftarrow [(\neg x_0 \& y_0) | ((x_0 \equiv y_0) \& z_0)] \gg^u 31 \\ z_1 &\leftarrow x_1 - y_1 - b \end{aligned}$$

在配有全套逻辑指令的电脑中，实现此算法共需8条指令。若CPU还支持“SLTU”指令，那么第2行可写为 $b \leftarrow (x_0 <^u y_0)$ ，这样只需4条即可。

在最低有效字中，可以只用31个位元来存放数据，而让权重最高的那个位元在一般情况下置0，仅用来存放进位或借位，要是这么做的话，在大部分电脑上只用5条指令就能实现双字长加减法了。

2.17 双字长移位

设 (x_1, x_0) 为一对尚待左移或右移的32位元字组，且 x_1 的权重较高。现将其整体视为一个64位元的量来移位，设移位结果为 (y_1, y_0) ，且 y_1 的权重也较高。假定移位量 n 是取值范围在0至63之间的变量，再假定CPU的移位指令为了调整移位长度而要把它与一个量取模时，这个量必须大于等于64。意思就是说，如果对某个字组做无符号移位

[⊖] Set on Less Than Unsigned，该指令以无符号数的方式比较第1个源寄存器的值是否小于第2个，若是，则将目标寄存器的值设为1，否则设为0。详情参见：<http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>。——译者注

时，指定的移位长度在 32 至 63 或 -32 至 -1 之间，那么移位结果的每一个位元值都是 0。若是带符号右移，那么移位结果中的 32 个位元，其值都等于移位前那个操作数的符号位。(本节的代码在 Intel x86 架构的电脑上无法正常运行，因为它们执行的是“模 32 移位”。)

在满足上述假设的条件后，双字左移操作可以用下面这个式子算出（需要 8 条指令）：

$$\begin{aligned} y_1 &\leftarrow x_1 \ll n \mid x_0 \gg^{u}(32-n) \mid x_0 \ll^{(n-32)} \\ y_0 &\leftarrow x_0 \ll n \end{aligned}$$

在第 1 个算式中，连接各项的操作符必须是“或”，而不能是“加”，因为只有这样做了，才能在 $n=32$ 时算出正确答案。若已知 $0 \leq n \leq 32$ ，那么第 1 个式子的最后一项就可以省掉，这样的话仅需 4 条指令。

同理，无符号双字右移指令的算法如下：

$$\begin{aligned} y_0 &\leftarrow x_0 \gg^{u} n \mid x_1 \ll^{(32-n)} \mid x_1 \gg^{u}(n-32) \\ y_1 &\leftarrow x_1 \gg n \end{aligned}$$

带符号双字右移实现起来比较难，因为得把算式最后一项中那个多余的符号传播位去掉。一种较为直接的算法代码是：

$$\begin{aligned} \text{if } n < 32 \text{ then } y_0 &\leftarrow x_0 \gg^{u} n \mid x_1 \ll^{(32-n)} \\ \text{else } y_0 &\leftarrow x_1 \gg^{s}(n-32) \\ y_1 &\leftarrow x_1 \gg n \end{aligned}$$

若电脑支持条件移动指令（conditional move instruction），那么很容易就能改成只需 8 条指令的无分支代码了。如果没有条件移动指令，那么可以用带符号右移 31 位的老办法构建一个掩码，然后用它把最后一项中那个多余的符号传播位屏蔽掉，这种做法共需 10 条指令。

$$\begin{aligned} y_0 &\leftarrow x_0 \gg^{u} n \mid x_1 \ll^{(32-n)} \mid [(x_1 \gg^{s}(n-32)) \& ((32-n) \gg 31)] \\ y_1 &\leftarrow x_1 \gg n \end{aligned}$$

2.18 多字节加减法与求绝对值

某些应用程序需要处理由短整数（通常是字节或半字组）构成的数组。如果每次以字组为单位运算，那么执行速度通常就能快一些。为了明确演示这个过程，我们以封装到字组中的 4 个 1 字节整数做例子。当然这种办法也很容易适配到其他封装形式上，比如封装了 1 个 12 位元整数和两个 10 位元整数的字组。在 64 位机上，这些技术的价值更大，因为能够并发执行的指令更多。

要想完成多字节加法，必须设法阻止每个字节向更高字节产生的进位。这可以用下面两个步骤完成：

1. 将每个操作数中各个字节的最高位屏蔽，然后相加（这样就不会产生跨越字节边界的进位了）。

2. 对两个操作数中每个字节的最高位做1位元加法，并加上由次高位带入的进位，以便修正其值。

上述第1步求出来的和，其最高位也就是第2步中说的那个由次高位带来的进位。也可以用与此相似的公式计算减法。

加法

```
s ← (x & 0x7F7F7F7F) + (y & 0x7F7F7F7F)
s ← ((x ⊕ y) & 0x80808080) ⊕ s
```

减法

```
d ← (x | 0x80808080) - (y & 0x7F7F7F7F)
d ← ((x ⊕ y) | 0x7F7F7F7F) ≡ d
```

在支持全套逻辑指令的CPU中，如果算上加载常数0x7F7F 7F7F所需的指令，那么一共需要8条。（分别将式中与0x8080 8080按位与及按位或的操作替换为对0x7F7F 7F7F按位取反之后再按位与及按位或。^①）

如果字组恰好分割成两个位段，那么还有一种办法实现其加减法。在此情况下，可以先做32位元加法，然后再把多余的进位减掉。2.13.2节说过，可以用 $(x+y) \oplus x \oplus y$ 这个式子得出从低位带入每个位元中的进位，与之相似，还有一个适用于减法借位的式子。利用这两个式子，就可以按下述代码执行两个半字的加减法了，其结果已与 2^{16} 取模（此算法需要7条指令）：

加法

```
s ← x + y
c ← (s ⊕ x ⊕ y) & 0x0001 0000
s ← s - c
```

减法

```
d ← x - y
b ← (d ⊕ x ⊕ y) & 0x0001 0000
d ← d + b
```

在求多字节的绝对值时，如果某个字节表示负数（也就是其最高位是1），那么将其按位取反再加1即可。下列代码会计算x中每个字节的绝对值，并将其放入y中（需8条指令）：

```
a ← x & 0x80808080      // 提取每个字节的符号位
b ← a " >> 7          // 若x中的相应字节为负，则b中的对应字节是1
m ← (a - b) | a          // 若x中的相应字节为负，则m中的对应字节是0xFF
y ← (x ⊕ m) + b          // 若x中的相应字节为负，则将其按位取反再加1
```

^① 意思是将公式中 $(x \oplus y) \& 0x8080 8080$ 替换成 $(x \oplus y) \& \neg 0x7F7F 7F7F$ ，将 $x | 0x8080 8080$ 替换成 $x | \neg 0x7F7F 7F7F$ 。——译者注

第3行也可以写成 $m \leftarrow a + a - b$ 。第4行中加 b 这个操作不会产生跨字节边界的进位，因为在 $x \oplus m$ 这个数中，每个字节的最高位都是0。

2.19 doz、max、min 函数

“doz”函数意思是“差或零”(difference or zero)，其定义如下：

无符号数	带符号数
$\text{doz}(x, y) = \begin{cases} x - y, & x \geq y, \\ 0, & x < y. \end{cases}$	$\text{dozu}(x, y) = \begin{cases} x - y, & x \geq y, \\ 0, & x < y. \end{cases}$

此操作也叫“一年级减法”(first grade subtraction)，因为要是减去的量比被减的还多，结果就按0算。^①如果能以一条计算机指令来表示的话，那么最能派上用场的地方就是实现 $\max(x, y)$ 与 $\min(x, y)$ 这两个函数了(带符号版本与无符号版本均可)。大家在后面就会看到：此种情况下仅用两条指令就能实现这两个函数。很难用硬件实现 $\max(x, y)$ 与 $\min(x, y)$ 函数，因为CPU需要一条能够绕开加法器的电路，以便把从寄存器堆(Register File)的读端口中获取到的值重新写回其输入端口。一般来说不存在这种电路。假设真有的话，那么在此区域必须排布数条接线，才能实现这种寄存器旁路。图2.3演示了这一情况。(求最大值或最小值的指令)用加法器(Adder)算出 $x - y$ ，然后根据减法操作的两个最高有效位(2.12.2节中描述了判定方法)来判断 $x \geq y$ 还是 $x < y$ 。比较结果会放入一个数据选择器(MUX)^②中，然后选取器根据这一结果在 x 与 y 中选定一个值，并将其写回寄存器堆中的目标寄存器里。一般情况下并没有这种从寄存器堆里读出 x 、 y 并实现数据选择器的电路，而且也没多大用处。实现“差或零”指令无需依靠这些电路，因为它只会把加法器的结果(或0)直接写回寄存器堆栈，不需要数据选择器。

用“差或零”函数实现 $\max(x, y)$ 与 $\min(x, y)$ ，只需如下两条指令：

带符号数	无符号数
$\max(x, y) = y + \text{doz}(x, y)$	$\maxu(x, y) = y + \text{dozu}(x, y)$
$\min(x, y) = x - \text{doz}(x, y)$	$\minu(x, y) = x - \text{dozu}(x, y)$

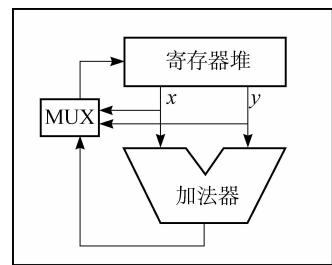


图2.3 实现 $\max(x, y)$ 与 $\min(x, y)$ 所需的电路

^① 数学家把这种运算叫做 monus，用符号 $-^+$ 来表示。此操作也称“正值差”(positive difference)或“饱和减法”(saturated subtraction)。

^② multiplexor，是一种可以从多个输入信号中选择一个信号输出的器件。详情参见：<http://zh.wikipedia.org/wiki/数据选择器>。——译者注

在带符号数的情况下，如果 $x - y$ 溢出了，那么“差或零”指令的结果就是负数。这里的溢出可以忽略，因为把该值与 y 相加或者从 x 里减去，都会导致再次溢出，于是两次溢出之后的结果反而是正确的。其中的原因是：如果 $\text{doz}(x, y)$ 是负的，那么只需把它视为无符号数，即可正确解读出两者的差了。

要是想在没有“差或零”指令的电脑上以无分支的方式快速实现 $\text{doz}(x, y)$ 、 $\max(x, y)$ 等函数的话，可以参考下面几段。我们将会分别演示如何在具有条件移动指令、比较谓词，以及能够高效访问进位标志的 CPU 上实现这些函数，并且还会告诉大家怎样在不具备上述功能的电脑上实现它。

若是电脑支持条件移动指令，那么用 3 条指令就能实现 $\text{doz}(x, y)$ 函数。以破坏原始参数的方式[⊖] 实现 $\max(x, y)$ 与 $\min(x, y)$ 函数，需要两条指令。举例来说，在具备完整 RISC 指令集的计算机上可用如下指令（其中的 r0 表示值恒为 0 的寄存器）实现 $z \leftarrow \text{doz}(x, y)$ ：

sub z, x, y	将 z 设为 $x - y$
cmplt t, x, y	如果 $x < y$ ，就将 t 设为 1，否则设为 0
movne z, t, r0	如果 $x < y$ ，就将 z 设为 0

在具备完整 RISC 指令集的电脑中， $x \leftarrow \max(x, y)$ 可以用下列指令实现：

cmplt t, x, y	如果 $x < y$ ，就将 t 设为 1，否则设为 0
movne x, t, y	如果 $x < y$ ，就将 y 赋给 x

只需改换上述各指令的比较条件，就可以实现出 \min 函数以及这几个函数的无符号版本。

通过比较谓词来实现这些函数，需要 4 至 5 条指令（如果比较谓词用 -1 表示“真”，则需要的指令数就是 3 至 4 条）：

$$\begin{aligned}\text{doz}(x, y) &= (x - y) \& - (x \geq y) \\ \max(x, y) &= y + \text{doz}(x, y) \\ &= ((x \oplus y) \& - (x \geq y)) \oplus y \\ \min(x, y) &= x - \text{doz}(x, y) \\ &= ((x \oplus y) \& - (x \leq y)) \oplus y\end{aligned}$$

在某些电脑上，可以依靠借位标志来计算这些函数的无符号版本。设 $\text{carry}(x - y)$ 为加法器在计算 $x + \bar{y} + 1$ 之后产生并存放于寄存器中的进位。于是，当且仅当 $x \geq y$ 时， $\text{carry}(x - y)$ 才可能等于 1。由此可得：

$$\begin{aligned}\text{dozu}(x, y) &= ((x - y) \& - (\text{carry}(x - y) - 1)) \\ \text{maxu}(x, y) &= x - ((x - y) \& (\text{carry}(x - y) - 1)) \\ \text{minu}(x, y) &= y + ((x - y) \& (\text{carry}(x - y) - 1))\end{aligned}$$

有些机子具备一种能够产生进位或借位的减法，同时还提供另外一种把上述指令产

[⊖] destructive operation，破坏性操作，此类操作会修改其一个或多个参数的值。

生的进位或借位当做输入值的减法。在具备这两种指令的大部分电脑中，只要计算完 $x - y$ 之后再加一条指令，即可求出 $\text{carry}(x - y) - 1$ 的值。举例来说，在 Intel x86 平台上， $\text{minu}(x, y)$ 可以用如下 4 条指令计算：

```
sub eax, ecx      ; 该指令的输入值 x 与 y 分别存放于 eax 及 ecx 寄存器中
sbb edx, edx      ; 如果  $x \geq y$ ，那么 edx = 0，否则为 -1
and eax, edx      ; 如果  $x \geq y$ ，那么结果为 0，否则为  $x - y$ 
add eax, ecx      ; 将上条指令的结果加 y。若  $x \geq y$ ，那么结果为 y，否则为 x
```

依此方式，这 3 个函数都可以用 4 条指令实现出来（要是电脑中有“按位取反后按位与”这个指令，那么 $\text{dozu}(x, y)$ 只需 3 条）。

有一种办法几乎适用于任何 RISC 架构的 CPU，那就是在上述各种表达式中挑选一种使用比较谓词的算法，然后用 2.12 节所讲的任意一种办法替换掉比较谓词。例如：

$$\begin{aligned} d &\leftarrow x - y \\ \text{doz}(x, y) &= d \& [(d \equiv ((x \oplus y) \& (d \oplus x))) \gg^s 31] \\ \text{dozu}(x, y) &= d \& \neg[((\neg x \& y) | ((x \equiv y) \& d)) \gg^s 31] \end{aligned}$$

根据电脑指令集的丰富程度，上述算法需要 7 至 10 条指令。若要实现 max 或 min 函数，则所需指令数还要再加 1。

若已知 $-2^{31} \leq x - y \leq 2^{31} - 1$ （此处为普通算术，而非计算机算术），那么只用 4 条无分支的基本 RISC 指令就可以实现这些函数了。只要 x 与 y 的差满足这个关系式，不论是带符号整数还是无符号整数，都可以用相同的代码实现。对于带符号整数来说，下列公式成立的充分条件（sufficient condition）是 $-2^{30} \leq x, y \leq 2^{30} - 1$ ，而对于无符号数，则是 $0 \leq x, y \leq 2^{31} - 1$ 。

$$\begin{aligned} \text{doz}(x, y) &= \text{dozu}(x, y) = (x - y) \& \neg((x - y) \gg^s 31) \\ \text{max}(x, y) &= \text{maxu}(x, y) = x - ((x - y) \& ((x - y) \gg^s 31)) \\ \text{min}(x, y) &= \text{minu}(x, y) = y + ((x - y) \& ((x - y) \gg^s 31)) \end{aligned}$$

下面列举一些“差或零”指令的用途。在这些情况下， $\text{doz}(x, y)$ 的结果必须视为无符号整数。

1. 直接实现 Fortran 语言的 IDIM 函数。
2. 计算两数之差的绝对值 [Knu7]：

$$\begin{aligned} |x - y| &= \text{doz}(x, y) + \text{doz}(y, x) \quad \text{适用于无符号数} \\ &= \text{dozu}(x, y) + \text{dozu}(y, x) \quad \text{适用于带符号数} \end{aligned}$$

推论： $|x| = \text{doz}(x, 0) + \text{doz}(0, x)$ （2.4 节列出了另外一些用 3 条指令来求绝对值的方法）。

3. 若两个无符号整数 x 与 y 之和超过了 32 位元所能表达的最大正整数（也就是 $2^{32} - 1$ ），那么将求和结果强行下调为 (clamp) 该值 [Knu7]：

$$\neg \text{dozu}(\neg x, y)$$

4. 实现某些比较谓词（各需4条指令）：

$$x > y = (\text{doz}(x, y) | -\text{doz}(x, y)) \gg 31$$

$$x \geq y = (\text{dozu}(x, y) | -\text{dozu}(x, y)) \gg 31$$

5. 计算由加法 $x + y$ 产生的进位（需5条指令）：

$$\text{carry}(x + y) = x \geq -y = (\text{dozu}(x, -y) | -\text{dozu}(x, -y)) \gg 31$$

如果将表达式 $\text{doz}(x, -y)$ 的值视为无符号整数，那么在大多数情况下它就能代表 $x + y$ 的和，只是低于0的结果会被强行上调为0。然而若 y 是负极值 x 不是负极值，则此方法无效。

IBM RS/6000电脑及其前身801^①具有带符号的“差或零”指令，而Knuth的MMIX计算机^②[Knu7]支持无符号版本的指令（包含一些派生指令，它们能够并发操作字组中某些部分）。于是这就产生一个问题：如何用无符号版本的指令来实现带符号的指令，反之亦然。可以用下列公式来转换这两个指令（其中的加减法只是为了反转符号位而已）：

$$\text{doz}(x, y) = \text{dozu}(x + 2^{31}, y + 2^{31})$$

$$\text{dozu}(x, y) = \text{doz}(x - 2^{31}, y - 2^{31})$$

还有两个有用的等式：

$$\text{doz}(\neg x, \neg y) = \text{doz}(y, x)$$

$$\text{dozu}(\neg x, \neg y) = \text{dozu}(y, x)$$

如果 x 和 y 里面有一个是负极值而另一个不是，那么关系式 $\text{doz}(\neg x, \neg y) = \text{doz}(y, x)$ 就不成立了。

2.20 互换寄存器中的值

有一个流传相当久的技巧，可以在不使用第3个寄存器的前提下交换两个寄存器的内容[IBM]：

$$x \leftarrow x \oplus y$$

$$y \leftarrow y \oplus x$$

$$x \leftarrow x \oplus y$$

这个技巧很适用于那种指令里只能使用两个地址的计算机(two-address machine)。也可以用 \equiv 逻辑操作（先按位异或再按位取反）来代替 \oplus 操作，而且还能用多种形式的

^① 是一项IBM于1974年启动的研究计划，其间产生了RISC指令集的概念。详情参见：http://zh.wikipedia.org/wiki/IBM_POWER。——译者注

^② MMIX是高德纳教授为了系统地讲解其算法理论而构想的一台计算机，采用64位RISC架构，便于描述实现算法所用的汇编语言指令。详情参见：<http://en.wikipedia.org/wiki/MMIX>。——译者注

加减法算式来改写：

$$\begin{array}{lll} x \leftarrow x + y & x \leftarrow x - y & x \leftarrow y - x \\ y \leftarrow x - y & y \leftarrow y + x & y \leftarrow y - x \\ x \leftarrow x - y & x \leftarrow y - x & x \leftarrow x + y \end{array}$$

不巧的是，上面这些通过加减法来交换寄存器的算法中，都包含一条不适用于二地址计算机的指令[⊖]，除非这些电脑上有“逆向减法”[⊖]。

这项小技巧在编写使用双缓冲功能的应用程序时很有用，这种程序要交换两个指针。我们把交换算法的第一条指令提出来，放在执行交换操作的那个循环前面（这样做要多用一个寄存器，没办法像原算法那样只用两个寄存器）：

$$\begin{array}{l} \text{循环外: } t \leftarrow x \oplus y \\ \text{循环内: } x \leftarrow x \oplus t \\ \quad \quad \quad y \leftarrow y \oplus t \end{array}$$

2.20.1 交换寄存器中相应的位段

这里研究的问题是根据掩码 m 来交换两个寄存器 x 、 y 中的内容：如果掩码中的某位 $m_i = 1$ ，那就交换 x 与 y 对应位元上的值，如果 $m_i = 0$ ，那么这两个位元就保持不变。这里强调“相应”位段的意思是，在不移位的前提下完成操作。 m 中值为 1 的位元不需要连续出现。一种较为直接的算法如下：

$$\begin{array}{l} x' \leftarrow (x \& \bar{m}) | (y \& m) \\ y \leftarrow (y \& \bar{m}) | (x \& m) \\ x \leftarrow x' \end{array}$$

如果用“临时寄存器”来保存 4 个按位与表达式的运算结果，并假设加载 m 或 \bar{m} 都用一条指令，计算机执行“按位取反后按位与”操作也只需一条指令，那么整个算法用 7 条指令即可实现。要是电脑能够并行计算 4 个按位与表达式，那么此算法仅需 3 个周期就能执行完。

下表中栏 (a) 所列的方法可能更好些（仅需 5 条指令，不过在具备无限指令级并行能力的计算机上要花 4 个周期执行），它是根据前面那个交换寄存器内容所用的“三次求异或”代码推演而来。

(a)	(b)	(c)
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$t \leftarrow (x \oplus y) \& m$
$y \leftarrow y \oplus (x \& m)$	$y \leftarrow y \equiv (x \bar{m})$	$x \leftarrow x \oplus t$
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$y \leftarrow y \oplus t$

[⊖] 指目标寄存器与被加数或被减数不符的那三条： $y \leftarrow x - y$, $x \leftarrow y - x$, $x \leftarrow x + y$ 。——译者注

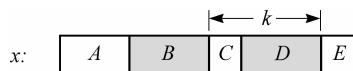
[⊖] 一般减法指令中，先出现的寄存器表示被减数，后出现的表示减数，而“逆向减法”（reverse subtract）与之相反。——译者注

(b) 栏所列算法与 (a) 栏一样，都可以交换寄存器的内容，然而如果 m 的值无法容纳于指令码的常量字段 (immediate field) 中，但 \bar{m} 却可以，并且计算机还支持按位等值指令的话，那么这种算法就很有用了。

(c) 栏中又列出了一种算法 [GLS1]，也需要 5 条指令（还得假设计算机只用 1 条指令就能把 m 载入寄存器），然而在一台指令级并行能力足够好的计算机上，仅用 3 个周期即可执行完。

2.20.2 交换同一寄存器内的两个位段

假设我们想在不改变寄存器中其他位元值的前提下，交换 x 中两个长度相同的位段。也就是说，在下图所描述的计算机字组中，要在不改变 A 、 C 、 E 的前提下，交换 B 与 D 。两个待交换位段之间的距离是 k 。



较为直接的实现方式是将 D 与 B 分别移动到新的位置，然后再用按位与及按位或操作将字组中的各个部件拼合连接起来。其算法如下：

$$t_1 = (x \& m) \ll k$$

$$t_2 = (x \gg^u k) \& m$$

$$x' = (x \& m') | t_1 | t_2$$

算式中的 m 是个掩码，如果位段 D 的某个位元是 1，则 m 的对应位元也是 1（其余位元都是 0）， m' 也是个掩码，其中值为 1 的位元，对应位段 A 、 C 、 D 中的相应位置上也为 1 的那些位元。在指令级并行能力不受限的计算机上，如果算上生成两个掩码所用的 4 条指令，那么实现这些式子共需 11 条指令，执行起来要花 6 个周期。

还有一种办法 [GLS1] 也基于上文所说的那些限定条件，然而只需 8 条指令即可实现，而且执行时间也只有 5 个周期。它与 2.20.1 节 (c) 栏中交换两个寄存器对应位段所用的代码类似。这里的 m 也是个掩码，用于析出位段 D 中值为 1 的位元。

$$t_1 = [x \oplus (x \gg^u k)] \& m$$

$$t_2 = t_1 \ll k$$

$$x' = x \oplus t_1 \oplus t_2$$

这个算法的思路是把 $B \oplus D$ 的值放在位段 D 所处的位置上（其余位置填 0），并用 t_1 表示它，再把同一个值又放在位段 B 所处的位置上，并用 t_2 表示它。此代码与早前那段代码一样，只有当 B 与 D 是两个“相互分离的位段”(split fields) 时才能算出正确结果，换句话说，掩码 m 中不能只出现一串值为 1 的位元。[⊖]

[⊖] 意思就是，必须出现两串相互分离且值为 1 的位元。也可以理解成位段 B 与 D 的间距 k 必须大于 0。——译者注

2.20.3 有条件的交换

在前两小节中，若 m 为 0，则那些基于按位异或的算法就退化成了没有任何效果的空操作（no-operation）。因此，如果我们在某条件 c 为真时，将掩码 m 中相应的位元设置成 1，而在条件 c 为假时，把 m 置为 0，那么就可以有条件地互换两个寄存器、两个寄存器中对应的位段，以及同一寄存器内两个位段的内容了。假如 m 的值能够用无分支代码来设定，那么整个条件式交换算法也就是无分支的。

2.21 在两个或两个以上的值之间切换

假设变量 x 的取值只可能是 a 或 b ，而你需要将 x 设为与当前值不同的另外一个值，并且想让代码逻辑不受制于 a 和 b 的具体值。举例来说，在某个编译器中， x 可能表示一个操作码，它要么是 branch true，要么是 branch false[⊖]，而不管 x 的当前值是哪一个，我们都想把它的值设置成另外一个。这里说的 branch true 与 branch false 操作码是随意举的例子，它们有可能是在 C 语言头文件中用 `#define` 或 `enum` 声明的。

在两个值之间切换，一种较为直接的代码是：

```
if (x == a) x = b;
else x = a;
```

或是像 C 程序中常见的那样：

```
x = x == a ? b : a;
```

下面两种做法远比上面两种好（至少可以说效率比较高一些）：

$$\begin{aligned} x &\leftarrow a + b - x \\ x &\leftarrow a \oplus b \oplus x \end{aligned}$$

若 a 与 b 为常数，则上述算法只需 1 至 2 条基本 RISC 指令。当然了，此处无需理会 $a+b$ 可能产生的溢出。

于是这就冒出来一个问题：有没有一种能在 3 个或 3 个以上的值之间轮替的高效算法？换句话说，给定 3 个互不相同的常数 a 、 b 、 c ，能不能找到一个满足下列条件且易于求值的函数 f ：

$$\begin{aligned} f(a) &= b \\ f(b) &= c \\ f(c) &= a \end{aligned}$$

说来有趣，其实这儿就有个现成的多项式能表达此函数。对于三个常数的情况，这个式子就是：

[⊖] 这两条指令分别表示在判断条件成立时转入分支，以及在判断条件不成立时转入分支。——译者注

$$f(x) = \frac{(x-a)(x-b)}{(c-a)(c-b)}a + \frac{(x-b)(x-c)}{(a-b)(a-c)}b + \frac{(x-c)(x-a)}{(b-c)(b-a)}c \quad (5)$$

(构建这个函数时所用的思路是：若 $x=a$ ，则第一项与最后一项消失，中间一项则化简为 b ，依此类推。) 求该式的值需要 14 次算术操作，而且如果 a 、 b 、 c 为任意值，那么中间运算的结果已经无法容纳于 32 位宽的字组中了。不过没关系，它只不过是个二次函数，所以可先将其展开为普通的二项式，然后再利用霍纳法则[⊖]来求值，这样只要 5 次算术操作就够了（其中 4 个操作用于计算整系数的二次项，再用 1 条计算最后的除法）。于是可将等式 (5) 改写成：

$$\begin{aligned} f(x) = & \frac{1}{(a-b)(a-c)(b-c)} \{ [(a-b)a + (b-c)b + (c-a)c]x^2 \\ & + [(a-b)b^2 + (b-c)c^2 + (c-a)a^2]x \\ & + [(a-b)a^2b + (b-c)b^2c + (c-a)ac^2] \} \end{aligned}$$

这种写法既无趣，又无用。

另外一种方法与等式 (5) 类似，它的 3 项中也只有一项能保留下：

$$f(x) = ((-(x=c)) \& a) + ((-(x=a)) \& b) + ((-(x=b)) \& c)$$

在具备相等谓词 (equal predicate) 的计算机上，不算加载常数所费的指令，实现此算法共需 11 条。由于最后一步的两个加法会把两个零值与一个非零值相加，所以加号可以用按位或、按位异或操作代替。

如果能预先求出 $a-c$ 与 $b-c$ 的值，那么就可以将公式简化为下述算法 [GLS1]：

$$\begin{aligned} f(x) = & ((-(x=c)) \& (a-c)) + ((-(x=a)) \& (b-c)) + c, \text{ 或} \\ f(x) = & ((-(x=c)) \& (a \oplus c)) \oplus ((-(x=a)) \& (b \oplus c)) \oplus c \end{aligned}$$

上述两种方法各需 8 条指令。然而在大部分电脑中，它们都不如下面这种直接以 C 语言代码写出来的算法，因为在 a 、 b 、 c 的值比较小的情况下，后者只需 4 至 6 条指令。

```
if (x == a) x = b;
else if (x == b) x = c;
else x = a;
```

要是还想深究这个问题的话，这里有一种巧妙的无分支算法 [GLS1]，能够在没有比较谓词指令的电脑中实现 3 值轮替。在大部分计算机上，该算法需要 8 条指令。

由于 a 、 b 、 c 互不相同，那么会有两个位置 n_1 与 n_2 ， a 、 b 、 c 三者在这两个位置上的位元值不一致，而且，在 n_1 位置上，其位元值与其他两数相异的数，与在 n_2 位置上，其位元值与其他两数相异的数，肯定不是同一个。下图以二进制形式演示了三数分别为 21、31 与 20 的情况：

[⊖] 霍纳法则的主旨就是提取 x ，比如，对一个四次多项式 $ax^4+bx^3+cx^2+dx+e$ 应用霍纳法则，结果就是 $x(x(ax+b)+c)+d+e$ 。这样的话，一个 n 次多项式，就可以用 n 次乘法与 n 次加法算出来了。于是，那些支持乘积累加指令的计算机，实现该算法就非常方便了。

1	0	1	0	1	<i>c</i>
1	1	1	1	1	<i>a</i>
1	0	1	0	0	<i>b</i>

*n*₁ *n*₂

为了使推导过程更通用，我们按照上图所示重排 *a*、*b*、*c* 三者的顺序，让 *n*₁ 位置上的“特殊数字”是 *a*，而 *n*₂ 位置上的特殊数字是 *b*。于是，在三数轮替过程中，*n*₁ 位置上可能出现的位元排列形式只有两种，(*a*_{*n*₁}, *b*_{*n*₁}, *c*_{*n*₁}) = (0, 1, 1) 或 (1, 0, 0)。同理，*n*₂ 位置上可能出现的位元排列形式也只有两种，(*a*_{*n*₂}, *b*_{*n*₂}, *c*_{*n*₂}) = (0, 1, 0) 或 (1, 0, 1)。于是共有 4 种情况，适用于每种情况的公式如下：

情况 1: (*a*_{*n*₁}, *b*_{*n*₁}, *c*_{*n*₁}) = (0, 1, 1), (*a*_{*n*₂}, *b*_{*n*₂}, *c*_{*n*₂}) = (0, 1, 0):

$$f(x) = x_{n_1} * (a - b) + x_{n_2} * (c - a) + b$$

情况 2: (*a*_{*n*₁}, *b*_{*n*₁}, *c*_{*n*₁}) = (0, 1, 1), (*a*_{*n*₂}, *b*_{*n*₂}, *c*_{*n*₂}) = (1, 0, 1):

$$f(x) = x_{n_1} * (a - b) + x_{n_2} * (a - c) + (b + c - a)$$

情况 3: (*a*_{*n*₁}, *b*_{*n*₁}, *c*_{*n*₁}) = (1, 0, 0), (*a*_{*n*₂}, *b*_{*n*₂}, *c*_{*n*₂}) = (0, 1, 0):

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (c - a) + a$$

情况 4: (*a*_{*n*₁}, *b*_{*n*₁}, *c*_{*n*₁}) = (1, 0, 0), (*a*_{*n*₂}, *b*_{*n*₂}, *c*_{*n*₂}) = (1, 0, 1):

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (a - c) + c$$

在上述公式中，每个乘法操作的左操作数都只有一个二进制位，这种与 0 或 1 相乘的操作，可以替换为同所有位元均是 0 或均是 1 的数求按位与。这样的话，上述第一种情况下的公式就可改写为：

$$f(x) = (((x \ll (31 - n_1)) \gg 31) \& (a - b)) + (((x \ll (31 - n_2)) \gg 31) \& (c - a)) + b$$

由于除了变量 *x* 之外的值均为常数，所以这个式子只需 8 个基本 RISC 指令就能实现。这里的加减法操作也可以替换成按位异或。

此算法的思路可推广到切换 4 个或更多个常数值的操作上，其要旨在于：找到能够区分各常数的一系列位置 *n*₁, *n*₂, …，使这些数在这一系列位置上的位元组合各不相同[⊖]。对于 4 常数轮替的情况，找到 3 个这样的位置就够了。然后（以 4 个常数切换为例），求出下列方程中 *s*、*t*、*u*、*v* 的值（也就是求下面这个一元四次方程在系数 *x*_{*n*_i} 为 0 或 1 时的解，使得 *f(x)* 分别等于 *a*、*b*、*c*、*d*）：

$$f(x) = x_{n_1} s + x_{n_2} t + x_{n_3} u + v$$

⊖ 例如，若 4 个无符号数分别为 0（二进制 0000），1（二进制 0001），4（二进制 0100），9（二进制 1001），则需要选择 *n*₁=3, *n*₂=2, *n*₃=0 这三个位置，因为只有这样，才能以 4 个数字在这 3 个位置上的位元组合（分别是 000, 001, 010, 101）把它们彻底区分开。如果只选取 *n*₁、*n*₂ 两个位置，就没办法把这 4 个数区隔开，因为 0（二进制 0000）和 1（二进制 0001）在这两个位置上的位元组合都是“00”。——译者注

假如只选取两个位置就能把这4个常数区隔开[⊕]，那么待求解的则是下面这个式子：

$$f(x) = x_{n_1} s + x_{n_2} t + x_{n_1} x_{n_2} u + v$$

2.22 布尔函数分解公式

本节我们来看看：想实现一个自变量个数为3、4或5的布尔函数（Boolean function），最少需要几个二元布尔操作（Boolean operation）或布尔指令才行。此处的“布尔函数”是指那种自变量为布尔值且函数值亦为布尔值的函数。

在本书所列的布尔代数算式中，“+”表示逻辑或，两数相邻表示求逻辑与， \oplus 表示逻辑异或，上划线或 \neg 前缀代表逻辑非。这些操作符都可以用“单比特操作数”（single-bit operand）或计算机字组的按位操作表示出来。分解布尔函数主要依靠下述定理：

定理 若 $f(x, y, z)$ 是有三个自变量的布尔函数，则其可分解为 $g(x, y) \oplus z h(x, y)$ 的形式。其中 g 与 h 皆为带两个自变量的布尔函数。^②

证明 [Ditlow]：将 $f(x, y, z)$ 展开为两个小项（minterm）之和，并把 \bar{z} 与 z 分别提取至每项前面，由此可得：

$$f(x, y, z) = \bar{z} f_0(x, y) + z f_1(x, y)$$

由于“+”的两个操作数不可能同时为1，所以可用逻辑异或操作代替逻辑或，于是：

$$\begin{aligned} f(x, y, z) &= \bar{z} f_0(x, y) \oplus z f_1(x, y) \\ &= (1 \oplus z) f_0(x, y) \oplus z f_1(x, y) \\ &= f_0(x, y) \oplus z f_0(x, y) \oplus z f_1(x, y) \\ &= f_0(x, y) \oplus z(f_0(x, y) \oplus f_1(x, y)) \end{aligned}$$

上述推导过程中两次用到恒等式 $(a \oplus b) c = ac \oplus bc$ 。 ■

在上式中， $f_0(x, y)$ 就是定理中要求的 $g(x, y)$ ，而 $f_0(x, y) \oplus f_1(x, y)$ 也就是定理中说的那个 $h(x, y)$ 。顺便说一句，将 $z=0$ 代入 $f(x, y, z)$ ，即可求出函数 $f_0(x, y)$ ；而将 $z=1$ 代入 $f(x, y, z)$ ，即可求出函数 $f_1(x, y)$ 。

推论 若16种双变量布尔函数都能用计算机指令集中的一条指令来实现，则任意三变量布尔函数都可用4条（或更少的）指令实现。

在这4条指令中，一条用于计算 $g(x, y)$ ，另一条用来计算 $h(x, y)$ ，然后再用逻辑

[⊕] 比如4个无符号数0（二进制000）、5（二进制101）、2（二进制010）、7（二进制111），只需选取 $n_1=1$ ， $n_2=0$ 即可用4个数在这两个位置上的位元组合（分别是00、01、10、11）将其分隔开。——译者注

^② 逻辑电路设计者把这叫做Reed-Muller Decomposition，亦称正 Davio 分解（Positive Davio Decomposition）。根据高德纳先生在[Knu4, 7.1.1]中的说法，此式因 I. I. Zhegalkin 而出名 [Matematicheskii Sbornik 35 (1928), 311-369]，有时也称为“俄式分解法”（Russian Decomposition）。

与、逻辑异或指令将其组合起来即可。

我们用下面这个布尔函数来演示一下。在 x 、 y 、 z 三值中，如果有两个值都是 1，而另外一个不是，那么函数值就是 1，否则函数值为 0：

$$f(x, y, z) = xy\bar{z} + x\bar{y}z + \bar{x}yz$$

在继续往下看之前，有兴趣的读者可以试试在不依赖前述定理的情况下，自己用 4 条指令把这个函数实现出来。

从定理的证明过程中可知：

$$\begin{aligned} f(x, y, z) &= f_0(x, y) \oplus z(f_0(x, y) \oplus f_1(x, y)) \\ &= xy \oplus z(xy \oplus (x\bar{y} + \bar{x}y)) \\ &= xy \oplus z(x + y) \end{aligned}$$

于是，只用 4 条指令即可实现该函数。

显然，定理也可以推广到自变量个数为 4 或更多的函数上。也就是说，任意布尔函数 $f(x_1, x_2, \dots, x_n)$ 都可以分解为 $g(x_1, x_2, \dots, x_{n-1}) \oplus x_n h(x_1, x_2, \dots, x_{n-1})$ 。因此，四变量布尔函数可按如下步骤分解：

$$\begin{aligned} f(w, x, y, z) &= g(w, x, y) \oplus zh(w, x, y), \text{ 其中} \\ g(w, x, y) &= g_1(w, x) \oplus yh_1(w, x), \text{ 而} \\ h(w, x, y) &= g_2(w, x) \oplus yh_2(w, x). \end{aligned}$$

由此可以看出，若 16 种双变量布尔函数都可以用 1 条指令实现，那么任意四变量的布尔函数都可用 10 条指令实现，同理，任意五变量的布尔函数都可用 22 条指令实现[⊖]。

然而实际上用不了这么多条指令。对于自变量个数为 4 或大于 4 的布尔函数来说，也许不能像上面这样用定理找到简单的分解式，不过已经有人用计算机彻底研究了这个问题。研究表明：任意四变量布尔函数都只需 7 条二元布尔指令实现。而任意五变量的布尔函数只需 12 条指令即可实现 [Knu4, 7.1.2]。

在 2^5 个（也就是 4 294 967 296 个）五变量布尔函数中，只有 1920 个需要用到 12 条指令，其余那些所需指令数都比这个少。这 1920 个函数基本都大同小异，有的只是重新排列了自变量的位置，有的只是把某些自变量用其补值代替，而有的只是对整个函数结果取逻辑非而已。

2.23 实现 16 种二元布尔操作

某些 CPU 指令集支持全部 16 种二元布尔操作。这些指令中，很多都没有用。比如， $f(x, y) = 0$ 这个函数，实际上就是清除某个寄存器的值而已，况且大多数计算机都有很多种方法可以清除一个寄存器的值，不一定非用布尔指令。然而，设计 CPU 架

[⊖] 推理的方法是，将实现变量数少 1 的布尔函数所需指令数乘以 2，再加 2。 $22 = 10 \times 2 + 2$ 。——译者注

构的人之所以要支持这16种布尔操作，其中一个原因就是有简单且很常见的电路能够实现它们。

2.3节中的表2.1列出了16种二元布尔函数。为了将这些函数实现为指令，我们用每个函数的函数值来当指令码。表中每一栏从下至上的4个函数值，分别记为 c_0 、 c_1 、 c_2 、 c_3 ，将两个输入寄存器的值记为 x 、 y ，则根据下列逻辑表达式即可构造一种能够实现16种二元布尔函数的电路来：

$$c_0xy + c_1x\bar{y} + c_2\bar{x}y + c_3\bar{x}\bar{y}$$

例如当 $c_0=c_1=c_2=c_3=0$ 时，指令计算的就是零值函数 $f(x,y)=0$ 的值。如果 $c_0=1$ 而操作码的其余三个位元都是0，那就相当于实现了“逻辑与”指令，而如果 $c_0=c_3=0$ 且 $c_1=c_2=1$ ，那么上式的值也就是“逻辑异或”指令的操作结果。

可以用 n 个4:1 MUX数据选择器来实现此电路，其中 n 是计算机的字组长度。其中两条选择线（Select Line）分别表示 x 与 y ，而每个MUX的数据输入端则代表操作码中4个二进制位。MUX是当前业界所用的一种标准部件，通常是一块运算速度很快的电路，其图示如下：

这块电路的功能是：根据 x 与 y 的值是00、01、10还是11在 c_0 、 c_1 、 c_2 、 c_3 中选择一个值输出。它有点像那种带4个挡位的旋转开关（four-position rotary switch）。

上面这种电路很优雅，不过如果把16种操作码都用上，那么实现成本有些高。有很多办法，只需8种操作码就可以实现全部16种逻辑操作，不过代价是需要使用一些不太常规的逻辑操作。表2.3演示了其中一种方案。

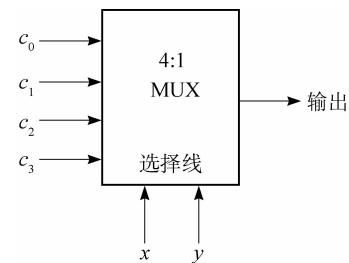


表2.3 8种必备的逻辑指令

函数值	计算公式	指令助记符（指令名 ^① ，英文指令名）
0001	xy	and（逻辑与， and）
0010	$x\bar{y}$	andc（和另一个数的反码求逻辑与， and with complement）
0110	$x \oplus y$	xor（逻辑异或， exclusive or）
0111	$x+y$	or（逻辑或， or）
1110	$\bar{x}\bar{y}$	nand（与非， negative and）
1101	$\bar{x}\bar{y}$ 或 $\bar{x}+y$	cor（求补之后求逻辑或， complement and or）
1001	$\bar{x}\oplus y$ 或 $x=y$	eqv（逻辑等值， equivalence）
1000	$\bar{x}+\bar{y}$	nor（或非， negative or）

^① 表中 andc又称“非蕴含”， cor又称“蕴含”，详情参见：<http://zh.wikipedia.org/wiki/逻辑运算符>。 andc运算通常用“and not”指令实现，该指令中文有时称“非与”，但为了避免同常见的“与非”操作混淆，译文多保留“and not”不译。——译者注

没有出现在表中的 8 种操作都可以用上述 8 种操作实现出来，具体做法是：交换 x 与 y 的输入值，或是把指令中两个表示寄存器的位段全部填成同一个寄存器[⊕]。详情参见本章末第 13 道习题。

IBM 的 POWER 架构 CPU 也用了这个方案，只是用“和另一个数的补值求逻辑或”(or with complement) 取代了“求补之后求逻辑或”(complement and or)。表 2.3 所示的这个方案中，对前 4 条指令的执行结果分别求补，即可实现后 4 条指令。

布尔运算发展史

乔治·布尔[⊖]在 1854 年所著的《思维规律的研究》一书中，详细阐释了逻辑代数(the algebra of logic)，然而它与今天大家熟知的“布尔代数”有某些区别。布尔本人用整数 1 和 0 来表示真(truth) 与假(falsity)，并演示了如何将生活中所说的“并且”、“或者”、“除了”等概念用普通数值代数(ordinary numerical algebra) 规范地表达出来。此外，他还用普通代数规范地阐释了集合论中交集、分离集(disjoint sets) 的并集、补集等概念。概率论中的概率都是 0 到 1 之间的实数，而布尔也用普通代数把这门理论加以规范。著作中还时常谈到一些哲学、宗教、法律等内容。

布尔被称为逻辑领域的大思想家，因为他把逻辑学规范化了，使得我们可以用普通的代数运算把一些复杂的逻辑命题忠实而精准地表达出来。

许多年之后，出现了一大批编程语言，它们都提供了 16 种布尔操作。IBM 的 PL/I 语言[⊕](大约在 1966 年) 引入了一个名叫 BOOL 的内置函数。在 BOOL(x, y, z) 这个函数调用式中， z 是一个长度为 4 的位串(若该参数不是长度为 4 的位串，则此函数在必要时会将其转换为这种位串)，而 x 与 y 则是两个长度相同的位串(如果这两个参数不是长度相同的位串，那么此函数在必要时会进行数据类型转换)。这里的 z 就代表函数要对 x 和 y 执行的布尔操作，例如二进制数 0000 表示恒零函数，0001 表示 xy 的值，而 0010 则表示 $x\bar{y}$ 的值，以此类推。

Wang System 2200B 计算机[⊖]中的 Basic 语言 [Neum] 也是这样，它(大约在 1974 年)也提供一种 BOOL 函数，不过，参与运算的不是位串或整数，而是字符串。

[⊕] 比如 and y , y 或 cor x , x 。——译者注

[⊖] George Boole, 1815—1864, 爱尔兰数学家、哲学家。详情参见：<http://zh.wikipedia.org/wiki/乔治·布尔>。——译者注

[⊕] PL/I, 是 Programming Language One 的简写。当中的 “I” 其实是 “一”的罗马数字。此语言运用于系统软件、图像、仿真、文字处理、网络、商业软件等领域，有些类似 PASCAL 语言。详情参见：<http://zh.wikipedia.org/wiki/PL/I>。——译者注

[⊖] 是由王安电脑公司(Wang Laboratories, 亦称王安实验室)推出的一款小型计算机(Minicomputer)。该公司是美籍华人计算机专家王安(1920—1990)创办的。详情参见：http://en.wikipedia.org/wiki/Wang_2200 与 http://en.wikipedia.org/wiki/Wang_Laboratories。——译者注

支持此类布尔操作的还有一门语言，就是后来改称 MacLisp 的 MIT PDP-6 Lisp[⊖] [GLS1]。

2.24 习题

- David de Kloet 提出了下列用于计算 snoob 函数[⊖]的代码。当 $x \neq 0$ 时，最终赋给 y 的值就是整个函数的结果：

```

y←x+(x & -x)
x←x & -y
while((x & 1)=0)x←x>>s1
x←x>>s1
y←y|x

```

- 这个算法与 2.1.3 节中 Gosper 提到的那段代码基本相同，只是它没有用除法指令，而是通过 while 循环来实现右移的。由于除法指令通常比较耗时，所以假如 while 循环执行次数不多的话，那这个算法也许能和 Gosper 的算法一较高下。设 n 为位串 x 、 y 的长度， k 代表位串中值为 1 的位元个数，并假定我们把所有包含 k 个“1”的 x 值都用这种方法计算一遍，那么请问每次运用此算法时，while 循环的循环体平均执行几轮？
- 本章正文中说过，如果左移操作的移位个数是个变量，那么它就不能按照“从右至左”的方式计算出来。然而有一个函数 [Knu8] $x \ll (x \& 1)$ ，其左移操作的移位长度为变量，但是，使用下面这两个式子，却能把它用“从右至左”的方式算出来：

$$\begin{aligned} &x + (x \& 1) * x, \text{ 或} \\ &x + (x \& (-x \& 1))) \end{aligned}$$

- 请问这是怎么回事？你还能再想一个这种函数吗？
- Dietz 指出，可以用下式计算两个无符号整数的均值。请证明此公式。

$$(x \& y) + ((x \oplus y) \gg^u 1)$$

- 想一个不会产生溢出的办法，来计算四个无符号整数的均值 $\lfloor (a+b+c+d)/4 \rfloor$ 。
- 如果预知了 x 或 y 的第 31 位，那么就可以大大简化 2.12 节中的比较谓词了。假设 $y_{31} = 0$ ，请在不使用比较指令的前提下，将原来计算表达式 $x \leqslant^u y$ 所用的 7 条指令简化为 3

[⊖] MacLisp 也拼写为 MACLISP，是一种 Lisp 编程语言的方言。最早起源于 20 世纪 60 年代麻省理工学院的 MAC 计划 (Project MAC)，并因此得名。主要是理查德·格林布拉特 (Richard Greenblatt) 在 PDP-6 上研发的，之后由约翰·怀特 (John L. White) 负责维持与持续开发。从 20 世纪 70 年代开始，PDP-6 上的 Lisp 又发展出 BBN Lisp 等其他分支，为了与之区分，大家改称其为 Maclisp。详情参见：<http://zh.wikipedia.org/wiki/Maclisp>。而 PDP-6 则是 DEC 在 1963 年推出的一款计算机，详情参见：<http://en.wikipedia.org/wiki/PDP-6>。——译者注

[⊖] 该函数用于求出下一个比 x 大，而值为 1 的位元个数又和 x 相同的数 y 。——译者注

- 条基本 RISC 指令。
6. 假设有两个数，它们的值可能一样，也可能不一样。现在采用“首尾循环进位”法 (end-around carry) 将其相加，请证明：把第一次加法运算在最高位所产生的进位加到求和结果之后，不可能在最高位上产生第二次进位。
 7. 假设负数用“反码”表示，那么请问如何以“首尾循环进位”方式实现加法操作？在这种计算方式下，由某个位置（不管哪个位置都行）上产生的进位，最多能传递几次？
 8. 在一台只具备基本 RISC 指令集的计算机上（这种计算机不支持“和另一个数的补值求与” (and with complement) 指令），用 3 条指令实现 MUX 操作： $(x \& m) | (y \& \sim m)$ 。
 9. 用 4 条指令实现 $x \oplus y$ 。只准使用与、或、非三种逻辑操作。
 10. 给定一个 32 位宽的字组 x 和两个存放在寄存器内的整数 i, j ，编写一段代码，将 x 中序号为 i 的那个位元，复制到位置 j 上。 i, j 两个值之间没有联系，只要满足 $0 \leq i, j \leq 31$ 就行。
 11. 如果利用 2.22 节的定理把一个自变量个数为 n 的布尔函数逐层分解，那么请问需要多少条二元布尔指令才能将其实现出来？
 12. 还有两种分解三变量布尔函数的办法，请证明它们：
 - (a) $f(x, y, z) = g(x, y) \oplus \bar{z}h(x, y)$ （“负 Davio 分解”，negative Davio decomposition）
 - (b) $f(x, y, z) = g(x, y) \oplus (z + h(x, y))$
 13. 本章正文中说过，只要将两个输入值交换，或是让指令中表示两个寄存器的位段都指向同一个寄存器，那么就可以用表 2.3 中的 8 个指令实现全部 16 种二元布尔操作。请说出具体算法。
 14. 有 6 种二元布尔操作： $f(x, y) = 0, 1, x, y, \bar{x}, \bar{y}$ ，它们本质上都是常数函数或一元函数，现在假设在不考虑上面这几个函数的前提下，想设计一套只需一条指令即可实现剩余 10 种布尔操作的指令集，那么请问：有没有可能只用不到 8 种二元布尔指令（也就是表 2.3 中那种操作码）就能把这套指令集做出来？
 15. 习题 13 已经证明：如果指令的两个操作数都是寄存器 (R-R 指令，register-register instruction)，那么仅需 8 种指令，就能实现全部 16 个二元布尔函数。现在请演示：如果指令的两个操作数一个是寄存器而另一个是常数 (R-I 指令，register-immediate instruction)，那么只用 6 种指令，就可以实现全部 16 个二元布尔函数。在使用 R-I 式指令时，不能交换两个操作数的位置，也不能填入两个相等的操作数，不过，可以对第二个输入值（也就是存放在指令码常量位段中的数 I）求补，或将其设定为其他值，这种操作不需要耗费执行时间。为了简洁起见，我们假设常量位段的宽度和通用寄存器所能容纳的位元数相等。
 16. 请证明：并非所有三变量布尔函数都能用 3 条二元逻辑操作指令实现出来。

第 3 章

2 的幂边界

3.1 将数值上调/下调为 2 的已知次幂的倍数

如果想将无符号整数 x 下调为一个值最接近它同时还是 8 的倍数的数，那么只需执行 $x \& -8$ 就可以了。另外一种办法是 $(x \gg^s 3) \ll^u 3$ 。只要我们规定“下调”指的是向负无穷方向调整，那么上面两种算法就同样适用于带符号的整数了（例如 $(-37) \& (-8) = -40$ ）。

上调与下调一样容易。比方说，想将无符号整数 x 上调为值最接近它而又能被 8 整除的数，那么可以使用下列两式计算：

$$(x + 7) \& -8 \text{ 或}$$

$$x + (-x \& 7)$$

只要我们规定“上调”的意思是向正无穷方向取整，那么这两个表达式就同样适用于带符号的整数。第 2 个表达式的第 2 项很有用，如果你想知道最少给 x 加上几才能把它变成 8 的倍数，那么用 $-x \& 7$ 就可以算出来 [Gold]。

如果想把一个带符号整数朝 0 所在的方向调整为值最接近它且能被 8 整除的数，那么可以把上面两个表达式组合一下，这样就能得到下面这个算法了：

$$t \leftarrow (x \gg^s 31) \& 7;$$

$$(x + t) \& -8$$

上面这种算法的第 1 行也可以改成 $t \leftarrow (s \gg^s 2) \gg^u 29$ 。如果计算机没有和常量求与 (and immediate) 的指令，或是常量太大，没办法放在指令码的常量位段中，那么这个表达式就能派上用场了。

有些时候，舍入因子 (rounding factor) 不是对齐量，而是把对齐量[⊖]取以 2 为底的对数，然后用这个对数值来表示舍入因子（比方说，舍入因子值为 3，意思就是朝 8 的倍

⊖ alignment amount，也译作“调整量”、“调整边界”、“对齐边界”等，也就是舍入数值后的标准位置。舍入之后的数值必须是该值的整数倍。——译者注

数对齐[⊕])。在这种情况下，可以按照下列代码来舍入，其中 $k = \log_2$ (对齐量)：

向下调整： $x \& ((-1) \ll k)$

$(x \gg u) \ll k$

向上调整： $t \leftarrow (1 \ll k) - 1; (x + t) \& \neg t$
 $t \leftarrow (-1) \ll k; (x - t - 1) \& t$

3.2 调整到上一个/下一个 2 的幂

现在定义两个与“向下取整”和“向上取整”类似的函数，只是我们这次把舍入之后的结果规定为距离自变量最近的那个 2 的整数幂，而不是像原来那样调整到距其最近的整数。这两个函数的数学定义是：

$$\text{flp2}(x) = \begin{cases} \text{未定义}, & x < 0, \\ 0, & x = 0, \\ 2^{\lfloor \log_2 x \rfloor}, & \text{其他}; \end{cases} \quad \text{clp2}(x) = \begin{cases} \text{未定义}, & x < 0, \\ 0, & x = 0, \\ 2^{\lfloor \log_2 x \rfloor}, & \text{其他}. \end{cases}$$

函数名的头两个字母分别表示“floor”(向下取整)与“ceiling”(向上取整)。因此 $\text{flp2}(x)$ 是小于等于 x 且最接近 x 的 2 的整数幂，而 $\text{clp2}(x)$ 是大于等于 x 且最接近 x 的 2 的整数幂。即便 x 不是整数，也还是可以套用这两个函数的定义(例如， $\text{flp2}(0.1) = 0.0625$)[⊖]。这两个函数也满足一些与向上取整和向下取整函数类似的关系式。下面列出几个这样的式子，其中 n 为整数：

$$\begin{array}{lll} \lfloor x \rfloor = \lceil x \rceil & \text{当且仅当 } x \text{ 为整数} & \text{flp2}(x) = \text{clp2}(x) \quad \text{当且仅当 } x \text{ 是 2 的幂或 } 0 \\ \lfloor x + n \rfloor = \lceil x \rceil + n & & \text{flp2}(2^n x) = 2^n \text{ flp2}(x) \\ \lceil x \rceil = -\lfloor -x \rfloor & & \text{clp2}(x) = 1/\text{flp2}(1/x), \quad x \neq 0 \end{array}$$

实际计算中我们只处理 x 为整数的情形，而且将其定为无符号数，这样的话，上面两个函数对所有的 x 值就都有定义了。此外，我们还规定，这些函数的值是将算数运算的正确结果与 2^{32} 取模后得到的(也就是说，如果 $\text{clp2}(x)$ 中的 x 大于 2^{31} ，那么此函数的值就是 0)。下表列出了一些 x 的函数值。

x	$\text{flp2}(x)$	$\text{clp2}(x)$
0	0	0
1	1	1
2	2	2
3	2	4
4	4	4

[⊕] 因为 $\log_2 8 = \log_2 2^3 = 3$ 。——译者注

[⊖] 因为 $2^{-3} = 0.125$, $2^{-4} = 0.0625$, 0.1 的值位于两者之间，所以对齐向下调整，就是 0.0625。——译者注

5	4	8
...
$2^{31} - 1$	2^{30}	2^{31}
2^{31}	2^{31}	2^{31}
$2^{31} + 1$	2^{31}	0
...
$2^{32} - 1$	2^{31}	0

下面列出了 flp2 与 clp2 函数之间的关系式。在给定的条件下，可以利用这些关系式，根据其中一个函数的值，推算出另一个函数。

$$\begin{aligned} \text{clp2}(x) &= 2\text{flp2}(x-1), & x \neq 1, \\ \text{flp2}(2x-1), & & 1 \leq x \leq 2^{31}, \\ \text{flp2}(x) &= \text{clp2}\left(\frac{x}{2}+1\right), & x \neq 0, \\ &= \text{clp2}\left(\frac{x+1}{2}\right), & x < 2^{31}. \end{aligned}$$

使用前导 0 计数指令 (number of leading zeros instruction) 很容易就能根据下列算式求出上调函数和下调函数的值。然而，如果要在 $x=0$ 及 $x>2^{31}$ 时使用这些关系式，那么就必须保证计算机的左移指令在移位量为 -1 , 32 及 63 时产生的结果是 0 。很多计算机 (例如 Power PC) 都采用“模 64”移位，这种移位方式符合上述要求。在移位量为 -1 的情况下，计算机朝相反方向移位也可以 (也就是说，将左移 -1 位解读为右移 1 位也行)。

$$\begin{aligned} \text{flp2}(x) &= 1 \ll (31 - \text{nlz}(x)) \\ &= 1 \ll (\text{nlz}(x) \oplus 31) \\ &= \text{0x8000 0000} \gg^u \text{nlz}(x) \\ \text{clp2}(x) &= 1 \ll (32 - \text{nlz}(x-1)) \\ &= \text{0x8000 0000} \gg^u (\text{nlz}(x-1) - 1) \end{aligned}$$

3.2.1 向下舍入

图 3.1 演示了如何在没有前导 0 计数指令时实现无分支的向下舍入算法。该算法的核心思路就是把最左方的“1”向右传播，执行起来共需 12 条指令。

图 3.2 分别采用两个简单的循环来计算同一个函数，其中所有变量都是无符号整数。在右侧的那个循环中，如果 x 不等于 0 ，我们就反复“关闭” x 最右侧的“1”；如果 x 等于 0 ，我们就把执行本轮循环前的 x 作为函数值返回。

左侧循环需要 $4\text{nlz}(x)+3$ 条指令实现，而如果忽略与 0 比较的指令，那么在 $x \neq 0$ 时，右侧循环所需的指令数为 $4\text{pop}(x)^\ominus$ 。

\ominus $\text{pop}(x)$ 就是 x 中值为 1 的位元个数。

```
unsigned flp2(unsigned x) {
    x = x | (x >> 1);
    x = x | (x >> 2);
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x | (x >> 16);
    return x - (x >> 1);
}
```

图 3.1 用无分支代码将 x 下调为不大于 x 且值与之最近的 2 的幂

```
y = 0x80000000;
while (y > x)
    y = y >> 1;
return y;
```

```
do {
    y = x;
    x = x & (x - 1);
} while(x != 0);
return y;
```

图 3.2 用简单的循环代码找出值不大于 x 且与之最接近的 2 的幂

3.2.2 向上舍入

利用刚才说的向右传播技巧，可以编写出一个巧妙的算法，把某个值向上调整为下一个 2 的幂。图 3.3 列出了此算法，代码不含分支，共需 12 条指令。

用刚才那种循环的办法实现向上舍入，效果并不好：

```
y = 1;
while (y < x)          //比较两个无符号数
    y = 2*y;
return y;
```

上述代码在 $x=0$ 时的结果为 1，这恐怕与大家的预期不符[⊖]。而且一旦 $x \geq 2^{31}$ ，就会陷入死循环。在正常情况下，它执行时要花费 $4n+3$ 条指令，其中 n 是返回整数 y 所对应的 2 的幂指数[⊖]。因此，从执行所需的指令数来判断，在 $n \geq 3$ （也就是 $x \geq 8$ ）时，该算法比无分支算法慢。

```
unsigned clp2(unsigned x) {
    x = x - 1;
    x = x | (x >> 1);
    x = x | (x >> 2);
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x | (x >> 16);
    return x + 1;
}
```

图 3.3 寻找值不小于 x 且与之最接近的 2 的幂

3.3 判断取值范围是否跨越了 2 的幂边界

假定内存被分隔成若干块 (block)，每一块的大小均为 2 的幂，且内存地址从 0 开始计数。这里说的“块”可以指字组、双字、页 (page) 等。现在给定起始地址 a 与长度 l ($l \geq 2$)，判断从 a 到 $a+l-1$ 之间的这一段内容是否跨越了两个块的边界。 a 与 l 均为无符号数，其值不限，只要能容纳于寄存器中即可。

假如 $l=0$ 或 1，那么不论 a 为何值，都不会跨越块边界。而一旦 l 本身比块的大小还大，那么不论 a 为何值，都肯定会跨越块边界。假如 l 的值特别大（有可能大到 $a+l-1$ 的值已经突破了计算机容许的最大值，从而又折回到 0），那么就算内存范围的第一个字节与最后一个字节位于同一块内，也还是有可能跨越边界。

[⊖] 按照 2.3 节中 clp2 函数的定义，当 x 为 0 时，函数值也应为 0，而不应为 1。——译者注

[⊖] 例如 y 为 8 时， n 为 3，因为 $8=2^3$ ，也可以将 n 理解成 $\log_2 y$ 。——译者注

令人惊讶的是，在IBM System/370计算机^①中，有一种非常精准的办法〔CJS〕能检测边界跨越。我们假设块大小为4096字节（也就是常见的内存页面大小），该方法代码如下：

```
O    RA,=A(-4096)
ALR RA,RL
BO  CROSSES
```

第1条指令的意思是将RA（其中含有起始地址 a ）与数字0xFFFF FF00取逻辑或。第2条指令则是将RA与长度 l 相加，并设置条件码。执行完这条逻辑加（add logical）指令后，如果发生进位，那么特征码的第1个位元置1，如果32位寄存器RA的值非0，则特征码的第2个位元置1。若两个特征码位同时为1，则最后一条指令会令程序转入分支。转入分支时，RA的值就是该取值范围超过首个页面的那部分长度（这是一个未出现在需求中的附加功能）。

例如，若 $a=0$ 且 $l=4096$ ，那么会发生进位，然而结果寄存器中的值是0。于是两个特征码不可能都是1，所以程序也就不会转入CROSSES标签所指的分支了。

现在研究如何将上述算法移植到具备RISC指令集的计算机上，此类计算机通常没有那种“当发生进位且寄存器中运算结果非0时转入分支”的指令。为了描述起来方便，我们假定块的大小为8字节，这样的话，只有当发生进位 $((a| -8) + l \geq 2^{32})$ 且寄存器结果非0 $((a| -8) + l \neq 2^{32})$ 时，〔CJS〕算法才会转入CROSSES分支。于是，上述算法也就等同于这个谓词的值：

$$(a| -8) + l > 2^{32}$$

而这个谓词的值，又与在计算 $((a| -8) - 1) + l$ 的最后一个加法时的进位情况相同，所以在支持“进位时转入分支”（branch on carry）指令的计算机上，可以直接用此式来判断，这样的话，把载入常数-8的那条指令算上，一共需要5条。

如果计算机中没有“进位时转入分支”的指令，那么根据2.13.3节讲的“当且仅当 $\neg x \leq y$ 时， $x + y$ 才会发生进位”这一事实，可以得到下式：

$$\neg((a| -8) - 1) \leq l$$

使用 $\neg(x - 1) = -x$ 等式子，可以推出下述几种等效的“边界跨越”谓词表达式：

$$-(a| -8) \leq l$$

$$\neg(a| -8) + 1 \leq l$$

$$(\neg a \& 7) + 1 \leq l$$

如果算上最后的分支指令，那么在大多数RISC架构的计算机中，上述算式需要5至6条指令。

^① IBM公司1970年推出的大型电脑系列，是IBM System/360的后继产品。详情参见：https://en.wikipedia.org/wiki/IBM_System/370。——译者注

此问题还有另外一种解决思路。因为某个取值范围是否跨越8字节边界的充分必要条件是：

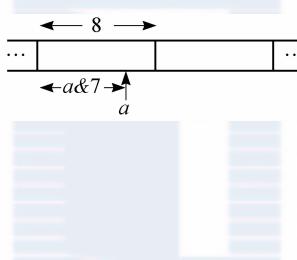
$$(a \& 7) + l - 1 \geq 8$$

因为（当 l 很大时）可能会溢出，所以不能直接求此式的值。然而只要将其重排为 $8 - (a \& 7) < l$ ，就能直接计算了（因为该式各部分均不会溢出）。于是就得到了这个表达式：

$$8 - (a \& 7) \stackrel{u}{<} l$$

在绝大多数 RISC 架构的计算机中，这个式子都只需 5 条指令（如果有“立即数减法”指令，那就是 4 条）。若发生了跨越边界现象，那么可以用 $l - (8 - (a \& 7))$ 求得超出第 1 块的长度。只需多用一条减法指令即可算出此值。

下边这张图 [Kumar] 直观地演示了此公式。其中 $a \& 7$ 就是 a 在块中的偏移量，所以 $8 - (a \& 7)$ 就是当前块中剩余的空间。



3.4 习题

1. 将无符号整数向 8 的倍数舍入。分别实现下列 3 种舍入标准：[⊕]
 - (a) 三舍四入。[⊖]
 - (b) 四舍五入。[⊖]
 - (c) 三舍五入四成双[⊖]。在这种舍入方式下，一个除以 8 余 4 的数，应该舍入为距离其最近的 16 的倍数（这也叫“无偏差”舍入，“unbiased” rounding）。
2. 将无符号整数向 10 的倍数舍入。分别实现下列 3 种舍入标准。
 - (a) 四舍五入。
 - (b) 五舍六入。
 - (c) 四舍六入五成双。在这种舍入方式下，一个除以 10 余 5 的数，应该舍入为距离其

[⊕] 原文为 halfway case，也就是当待舍入的数与左右两个 8 的倍数之间的距离均为 4 时，是应该舍还是应该入。不同标准下的舍入结果有可能不同。比如待舍入的数是 20，它左边那个 8 的倍数是 16，右边那个 8 的倍数是 24，20 与两者距离均为 4，那么 20 究竟舍为 16 还是入为 24，则取决于所选的标准。——译者注

[⊖] 若该数除以 8 的余数大于等于 4，则向上调整，否则向下调整。——译者注

[⊖] 若该数除以 8 的余数小于等于 4，则向下调整，否则向上调整。——译者注

[⊖] 此处借用十进制下的俗语。若该整数除以 8 的余数小于 4，则向下调整，大于 4 则向上调整。若恰好为 4，且商为偶数，则向下调整；若商为奇数，则向上调整，调整后的值必定是 8 的双数倍。比如 28 这个数，与 24 和 32 的距离都是 4，然而 28 与 8 的商为 3，是奇数，所以应该上调为 32，而不能下调为 24，因为 24 不是 8 的偶数倍。——译者注

最近的10的偶数倍。[⊖]

此题可任意使用除法、求余、乘法指令，而且不必考虑那些与最大的无符号整数非常接近的值。

3. 用C语言编写一个实现“不对齐加载”（unaligned load）功能的函数。该函数接收一个起始地址 a ，然后把从 a 到 $a+3$ 的4个字节视为一个整数，载入32位寄存器中。起始地址参数 a 所指的内容，是待加载整数中权重最低的那个字节（也就是说，假定计算机是以“小端序”[⊖]的方式存放多字节整数的）。函数代码不要有分支语句，而且最多只能执行两次加载指令。如果 a 本身恰好位于字组边界处，则函数不应读取 $a+4$ 这个地址中的内容，因为此字节可能处于一个“读保护块”（read-protected block）中。



[⊖] 即按照“四舍六入五成双”标准舍入之后的数必定是20的倍数。其中“成双”一词的意思是，如果发现待舍入的值除以10的余数为5，那么一定要把它调整成10的双数倍（也就是20的倍数）。例如35与30和40的距离均为5，而35与10的商为3，是奇数，所以应该上调为40，而不能下调为30，因为30不是10的双数倍。详情参见：<http://zh.wikipedia.org/wiki/数值简化规则>。——译者注

[⊖] little-endian，又称“小尾序”，它会把权重最低的字节存放在内存地址最小处。详情参见：<http://zh.wikipedia.org/wiki/字节序>。——译者注