$2^{2^5} + 1 = 641 \cdot 6700417$

$\text{avg}(x, y) = (x \,\&\, y) + ((x \oplus y) \overset{u}{\gg} 1)$

$2^{2^6} + 1 = 274177 \cdot 67280421310721$

$x - y = x + \bar{y} + 1$

$\lfloor a \rfloor + \lfloor b \rfloor \le \lfloor a + b \rfloor \le \lfloor a \rfloor + \lfloor b \rfloor + 1$

$\text{pop}(x) = -\sum_{i=0}^{31} (x \overset{rot}{\ll} i)$

George Boole
1815 - 1864

$\lfloor \sqrt{1111111} \rfloor = 1111$

$(x \ne 0) = (x \mid -x) \overset{u}{\gg} 31$

$\text{mux}(x, y, m) = ((x \oplus y) \,\&\, m) \oplus y$

$A(n, d) = A(n-1, d-1), d \text{ even}$

$-\bar{x} = x + 1$

Hacker's Delight

$\frac{1}{3} = 0.01010101\ldots$

Second Edition

$1111^2 = 1100001$

$n = -2^{31} b_{31} + 2^{30} b_{30} + 2^{29} b_{29} + \ldots + 2^0 b_0$

$\lceil x \rceil = -\lfloor -x \rfloor$

$f(x, y, z) = g(x, y) \oplus zh(x, y)$

Num factors of 2 in $x =$
$\log_2(x \,\&\, (-x)), \; x \ne 0$

$\text{rjust}(x) = x \overset{u}{\div} (x \,\&\, -x), \; x \ne 0$

$p_n = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{n} \left(\sum_{x=1}^{m} \left\lfloor \cos^2 \pi \frac{(x-1)!+1}{x} \right\rfloor \right)^{-1/n} \right\rfloor$

$x \oplus y = (x \mid y) - (x \,\&\, y)$

$x + y = (x \mid y) + (x \,\&\, y)$

Henry S. Warren, Jr.

Hacker's Delight

Second Edition

Henry S. Warren, Jr.

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
CapeTown • Sydney • Tokyo • Singapore • Mexico City

To Joseph W. Gauld, my high school algebra teacher, for sparking in me a delight in the simple things in mathematics

Library of Congress Cataloging-in-Publication Data

Warren, Henry S.
Hacker's delight / Henry S. Warren, Jr. -- 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-84268-5 (hardcover : alk. paper)
1. Computer programming. I. Title.
QA76.6.W375 2013
005.1—dc23
2012026011

ISBN-13: 978-0-321-84268-8
ISBN-10: 0-321-84268-5
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, September 2012

Contents

Foreword

Foreword from the First Edition

When I first got a summer job at MIT's Project MAC almost 30 years ago, I was delighted to be able to work with the DEC PDP-10 computer, which was more fun to program in assembly language than any other computer, bar none, because of its rich yet tractable set of instructions for performing bit tests, bit masking, field manipulation, and operations on integers. Though the PDP-10 has not been manufactured for quite some years, there remains a thriving cult of enthusiasts who keep old PDP-10 hardware running and who run old PDP-10 software—entire operating systems and their applications—by using personal computers to simulate the PDP-10 instruction set. They even write new software; there is now at least one Web site with pages that are served up by a simulated PDP-10. (Come on, stop laughing—it's no sillier than keeping antique cars running.)

I also enjoyed, in that summer of 1972, reading a brand-new MIT research memo called HAKMEM, a bizarre and eclectic potpourri of technical trivia.[1] The subject matter ranged from electrical circuits to number theory, but what intrigued me most was its small catalog of ingenious little programming tricks. Each such gem would typically describe some plausible yet unusual operation on integers or bit strings (such as counting the 1-bits in a word) that could easily be programmed using either a longish fixed sequence of machine instructions or a loop, and then show how the same thing might be done much more cleverly, using just four or three or two carefully chosen instructions whose interactions are not at all obvious until explained or fathomed. For me, devouring these little programming nuggets was like eating peanuts, or rather bonbons—I just couldn't stop—and there was a certain richness to them, a certain intellectual depth, elegance, even poetry.

"Surely," I thought, "there must be more of these," and indeed over the years I collected, and in some cases discovered, a few more. "There ought to be a book of them."

I was genuinely thrilled when I saw Hank Warren's manuscript. He has systematically collected these little programming tricks, organized them thematically, and explained them clearly. While some of them may be described in terms of machine instructions, this is not a book only for assembly language programmers. The subject matter is basic structural relationships among integers and bit strings in a computer and efficient techniques for performing useful operations on them. These techniques are just as useful in the C or Java programming languages as they are in assembly language.

Many books on algorithms and data structures teach complicated techniques for sorting and searching, for maintaining hash tables and binary trees, for dealing with records and pointers. They overlook what can be done with very tiny pieces of data—bits and arrays of bits. It is amazing what can be done with just binary addition and subtraction and maybe some bitwise operations; the fact that the carry chain allows a single bit to affect all the bits to its left makes addition a peculiarly powerful data manipulation operation in ways that are not widely appreciated.

Yes, there ought to be a book about these techniques. Now it is in your hands, and it's terrific. If you write optimizing compilers or high-performance code, you must read this book. You otherwise might not use this bag of tricks every single day—but if you find yourself stuck in some situation where you apparently need to loop over the bits in a word, or to perform some operation on integers and it just seems harder to code than it ought, or you really need the inner loop of some integer or bit-fiddly computation to run twice as fast, then this is the place to look. Or maybe you'll just find yourself reading it straight through out of sheer pleasure.

Guy L. Steele, Jr.
Burlington, Massachusetts
April 2002

Preface

Caveat Emptor: The cost of software maintenance increases with the square of the programmer's creativity.

First Law of Programmer Creativity,
Robert D. Bliss, 1992

This is a collection of small programming tricks that I have come across over many years. Most of them will work only on computers that represent integers in two's-complement form. Although a 32-bit machine is assumed when the register length is relevant, most of the tricks are easily adapted to machines with other register sizes.

This book does not deal with large tricks such as sophisticated sorting and compiler optimization techniques. Rather, it deals with small tricks that usually involve individual computer words or instructions, such as counting the number of 1-bits in a word. Such tricks often use a mixture of arithmetic and logical instructions.

It is assumed throughout that integer overflow interrupts have been masked off, so they cannot occur. C, Fortran, and even Java programs run in this environment, but Pascal and Ada users beware!

The presentation is informal. Proofs are given only when the algorithm is not obvious, and sometimes not even then. The methods use computer arithmetic, "floor" functions, mixtures of arithmetic and logical operations, and so on. Proofs in this domain are often difficult and awkward to express.

To reduce typographical errors and oversights, many of the algorithms have been executed. This is why they are given in a real programming language, even though, like every computer language, it has some ugly features. C is used for the high-level language because it is widely known, it allows the straightforward mixture of integer and bit-string operations, and C compilers that produce high-quality object code are available.

Occasionally, machine language is used, employing a three-address format, mainly for ease of readability. The assembly language used is that of a fictitious machine that is representative of today's RISC computers.

Branch-free code is favored, because on many computers, branches slow down instruction fetching and inhibit executing instructions in parallel. Another problem with branches is that they can inhibit compiler optimizations such as instruction scheduling, commoning, and register allocation. That is, the compiler may be more effective at these optimizations with a program that consists of a few large basic blocks rather than many small ones.

The code sequences also tend to favor small immediate values, comparisons to zero (rather than to some other number), and instruction-level parallelism. Although much of the code would become more concise by using table lookups (from memory), this is not often mentioned. This is because loads are becoming more expensive relative to arithmetic instructions, and the table lookup methods are often not very interesting (although they *are* often practical). But there are exceptional cases.

Finally, I should mention that the term "hacker" in the title is meant in the original sense of an aficionado of computers—someone who enjoys making computers do new things, or do old things in a new and clever way. The hacker is usually quite good at his craft, but may very well not be a professional computer programmer or designer.

The hacker's work may be useful or may be just a game. As an example of the latter, more than one determined hacker has written a program which, when executed, writes out an exact copy of itself.[1] This is the sense in which we use the term "hacker." If you're looking for tips on how to break into someone else's computer, you won't find them here.

Acknowledgments

First, I want to thank Bruce Shriver and Dennis Allison for encouraging me to publish this book. I am indebted to many colleagues at IBM, several of whom are cited in the Bibliography. One deserves special mention: Martin E. Hopkins, whom I think of as "Mr. Compiler" at IBM, has been relentless in his drive to make every cycle count, and I'm sure some of his spirit has rubbed off on me. Addison-Wesley's reviewers have improved the book immensely. Most of their names are unknown to me, but the review by one whose name I did learn was truly outstanding: Guy L. Steele, Jr., completed a 50-page review that included new subject areas to address, such as bit shuffling and unshuffling, the sheep and goats operation, and many others. He suggested algorithms that beat the ones I used. He was extremely thorough. For example, I had erroneously written that the hexadecimal number AAAAAAAA factors as $2 \cdot 3 \cdot 17 \cdot 257 \cdot 65537$; Guy pointed out that the 3 should be a 5. He suggested improvements to style and did not shirk from mentioning minutiae. Wherever you see "parallel prefix" in this book, the material is due to Guy.

H. S. Warren, Jr.
Yorktown, New York
June 2012

See www.HackersDelight.org for additional material related to this book.

Chapter 1. Introduction

1–1 Notation

This book distinguishes between mathematical expressions of ordinary arithmetic and those that describe the operation of a computer. In “computer arithmetic,” operands are bit strings, or bit vectors, of some definite fixed length. Expressions in computer arithmetic are similar to those of ordinary arithmetic, but the variables denote the contents of computer registers. The value of a computer arithmetic expression is simply a string of bits with no particular interpretation. An operator, however, interprets its operands in some particular way. For example, a comparison operator might interpret its operands as signed binary integers or as unsigned binary integers; our computer arithmetic notation uses distinct symbols to make the type of comparison clear.

The main difference between computer arithmetic and ordinary arithmetic is that in computer arithmetic, the results of addition, subtraction, and multiplication are reduced modulo 2^n, where n is the word size of the machine. Another difference is that computer arithmetic includes a large number of operations. In addition to the four basic arithmetic operations, computer arithmetic includes logical *and, exclusive or, compare, shift left*, and so on.

Unless specified otherwise, the word size is 32 bits, and signed integers are represented in two’s-complement form.

Expressions of computer arithmetic are written similarly to those of ordinary arithmetic, except that the variables that denote the contents of computer registers are in bold face type. This convention is commonly used in vector algebra. We regard a computer word as a vector of single bits. Constants also appear in bold-face type when they denote the contents of a computer register. (This has no analogy with vector algebra because in vector algebra the only way to write a constant is to display the vector’s components.) When a constant denotes part of an instruction, such as the immediate field of a *shift* instruction, light-face type is used.

If an operator such as “+” has bold face operands, then that operator denotes the computer’s addition operation (“vector addition”). If the operands are light-faced, then the operator denotes the ordinary scalar arithmetic operation. We use a light-faced variable x to denote the arithmetic value of a bold-faced variable x under an interpretation (signed or unsigned) that should be clear from the context. Thus, if x = **0x80000000** and y = **0x80000000**, then, under signed integer interpretation, $x \mathbf{=} y \mathbf{= -2^{31},}\ x \mathbf{+} y \mathbf{= -2^{32}}$, and $x + y = \mathbf{0}$. Here, **0x80000000** is hexadecimal notation for a bit string consisting of a 1-bit followed by 31 0-bits.

Bits are numbered from the right, with the rightmost (least significant) bit being bit 0. The terms “bits,” “nibbles,” “bytes,” “halfwords,” “words,” and “doublewords” refer to lengths of 1, 4, 8, 16, 32, and 64 bits, respectively.

Short and simple sections of code are written in computer algebra, using its assignment operator (left arrow) and occasionally an *if* statement. In this role, computer algebra is serving as little more than a machine-independent way of writing assembly language code.

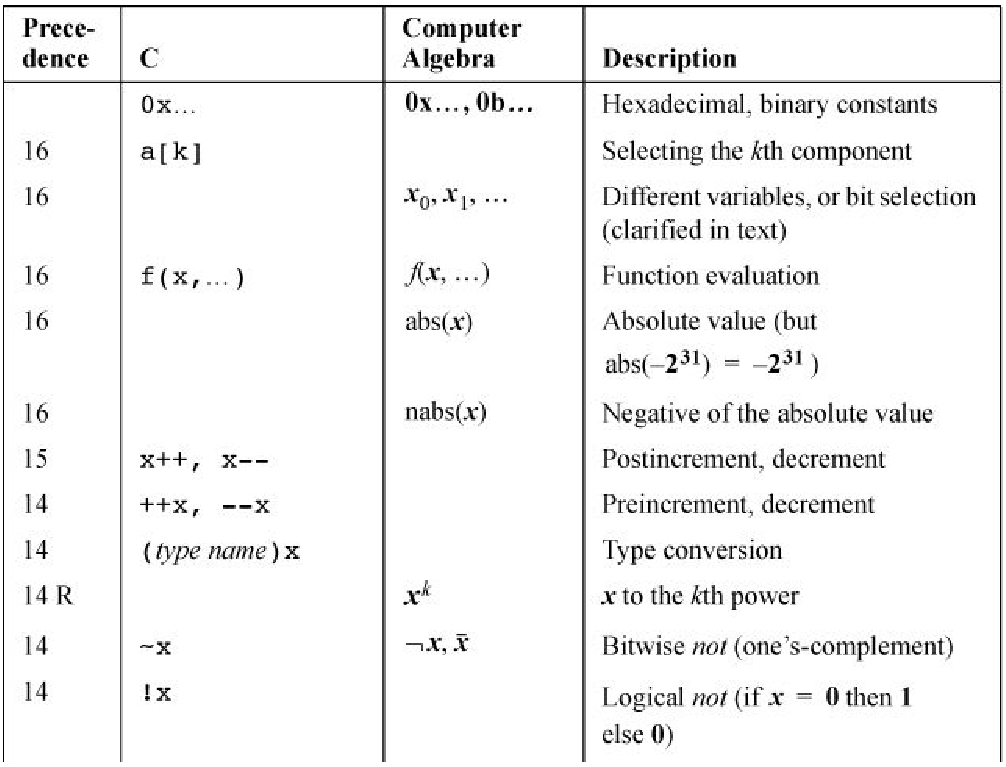
Programs too long or complex for computer algebra are written in the C programming language, as defined by the ISO 1999 standard.

A complete description of C would be out of place in this book, but Table 1–1 contains a brief summary of most of the elements of C [H&S] that are used herein. This is provided for the benefit of the reader who is familiar with some procedural programming language, but not with C. Table 1–1 also shows the operators of our computer-algebraic arithmetic language. Operators are listed from highest precedence (tightest binding) to lowest. In the Precedence column, L means left-associative; that is,

$$a \bullet b \bullet c = (a \bullet b) \bullet c$$

and R means right-associative. Our computer-algebraic notation follows C in precedence and associativity.

TABLE 1–1. EXPRESSIONS OF C AND COMPUTER ALGEBR



Prece-dence	C	Computer Algebra	Description
	`0x...`	**0x..., 0b...**	Hexadecimal, binary constants
16	`a[k]`		Selecting the *k*th component
16		$x_0, x_1, \ldots$	Different variables, or bit selection (clarified in text)
16	`f(x,...)`	$f(x, \ldots)$	Function evaluation
16		abs(x)	Absolute value (but abs($-\mathbf{2}^{31}$) = $-\mathbf{2}^{31}$)
16		nabs(x)	Negative of the absolute value
15	`x++, x--`		Postincrement, decrement
14	`++x, --x`		Preincrement, decrement
14	`(`*type name*`)x`		Type conversion
14 R		x^k	x to the *k*th power
14	`~x`	$\neg x, \bar{x}$	Bitwise *not* (one's-complement)
14	`!x`		Logical *not* (if $x = \mathbf{0}$ then **1** else **0**)

14	`-x`	$-x$	Arithmetic negation
13 L	`x*y`	$x * y$	Multiplication, modulo word size
13 L	`x/y`	$x \div y$	Signed integer division
13 L	`x/y`	$x \overset{u}{\div} y$	Unsigned integer division
13 L	`x%y`	rem(x, y)	Remainder (may be negative), of $(x \div y)$, signed arguments
13 L	`x%y`	remu(x, y)	Remainder of $x \overset{u}{\div} y$, unsigned arguments
		mod(x, y)	x reduced modulo y to the interval $[0, \text{abs}(y) - 1]$; signed arguments
12 L	`x + y, x - y`	$x + y, x - y$	Addition, subtraction
11 L	`x << y, x >> y`	$x \ll y, x \overset{u}{\gg} y$	Shift left, right with 0-fill ("logical" shifts)
11 L	`x >> y`	$x \overset{s}{\gg} y$	Shift right with sign-fill ("arithmetic" or "algebraic" shift)
11 L		$x \overset{rot}{\ll} y, x \overset{rot}{\gg} y$	Rotate shift left, right
10 L	`x < y, x <= y,` `x > y, x >= y`	$x < y, x \le y,$ $x > y, x \ge y$	Signed comparison
10 L	`x < y, x <= y,` `x > y, x >= y`	$x \overset{u}{<} y, x \overset{u}{\le} y,$ $x \overset{u}{>} y, x \overset{u}{\ge} y$	Unsigned comparison
9 L	`x == y, x != y`	$x = y, x \ne y$	Equality, inequality
8 L	`x & y`	$x \& y$	Bitwise *and*
7 L	`x ^ y`	$x \oplus y$	Bitwise *exclusive or*
7 L		$x \equiv y$	Bitwise *equivalence* ($\neg(x \oplus y)$)
6 L	`x | y`	$x \mid y$	Bitwise *or*
5 L	`x && y`	$x \overset{\rightarrow}{\&} y$	Conditional *and* (if $x = 0$ then 0 else if $y = 0$ then 0 else 1)
4 L	`x || y`	$x \overset{\rightarrow}{\mid} y$	Conditional *or* (if $x \ne 0$ then 1 else if $y \ne 0$ then 1 else 0)
3 L		$x \parallel y$	Concatenation
2 R	`x = y`	$x \leftarrow y$	Assignment

In addition to the notations described in Table 1–1, those of Boolean algebra and of standard mathematics are used, with explanations where necessary.

Our computer algebra uses other functions in addition to “abs,” “rem,” and so on. These are defined where introduced.

In C, the expression `x < y < z` means to evaluate `x < y` to a 0/1-valued result, and then compare that result to `z`. In computer algebra, the expression $x < y < z$ means $(x < y)$ & $(y < z)$.

C has three loop control statements: `while`, `do`, and `for`. The `while` statement is written:

`while` (*expression*) *statement*

First, *expression* is evaluated. If **true** (nonzero), *statement* is executed and control returns to evaluate *expression* again. If *expression* is **false** (0), the *while*-loop terminates.

The `do` statement is similar, except the test is at the bottom of the loop. It is written:

`do` *statement* `while` (*expression*)

First, *statement* is executed, and then *expression* is evaluated. If **true**, the process is repeated, and if **false**, the loop terminates.

The `for` statement is written:

`for` (e_1; e_2; e_3) *statement*

First, e_1, usually an assignment statement, is executed. Then e_2, usually a comparison, is evaluated. If **false**, the for-loop terminates. If **true**, *statement* is executed. Finally, e_3, usually an assignment statement, is executed, and control returns to evaluate e_2 again. Thus, the familiar “do i = 1 to n” is written:

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

(This is one of the few contexts in which we use the postincrement operator.)

The ISO C standard does not specify whether right shifts (“>>” operator) of signed quantities are 0-propagating or sign-propagating. In the C code herein, it is assumed that if the left operand is signed, then a sign-propagating shift results (and if it is unsigned, then a 0-propagating shift results, following ISO). Most modern C compilers work this way.

It is assumed here that left shifts are “logical.” (Some machines, mostly older ones, provide an “arithmetic” left shift, in which the sign bit is retained.)

Another potential problem with shifts is that the ISO C standard specifies that if the shift amount is negative or is greater than or equal to the width of the left operand, the result is undefined. But, nearly all 32-bit machines treat shift amounts modulo 32 or 64. The code herein relies on one of these behaviors; an explanation is given when the distinction is important.

1–2 Instruction Set and Execution Time Model

To permit a rough comparison of algorithms, we imagine them being coded for a machine with an instruction set similar to that of today’s general purpose RISC computers, such as the IBM RS/6000, the Oracle SPARC, and the ARM architecture. The machine is three-address and has a fairly large number of general purpose registers—

that is, 16 or more. Unless otherwise specified, the registers are 32 bits long. General register 0 contains a permanent 0, and the others can be used uniformly for any purpose.

In the interest of simplicity there are no “special purpose” registers, such as a condition register or a register to hold status bits, such as “overflow.” The machine has no floating-point instructions. Floating-point is only a minor topic in this book, being mostly confined to Chapter 17.

We recognize two varieties of RISC: a “basic RISC,” having the instructions shown in Table 1–2, and a “full RISC,” having all the instructions of the basic RISC, plus those shown in Table 1–3.

TABLE 1–2. BASIC RISC INSTRUCTION SET

Opcode Mnemonic	**Operands**	**Description**
`add, sub, mul, div, divu, rem, remu`	`RT,RA,RB`	`RT` ← `RA op RB`, where `op` is *add*, *subtract*, *multiply*, *divide signed*, *divide unsigned*, *remainder signed*, or *remainder unsigned*.
`addi, muli`	`RT,RA,I`	`RT` ← `RA op I`, where `op` is *add* or *multiply*, and `I` is a 16-bit signed immediate value.
`addis`	`RT,RA,I`	`RT` ← `RA + (I || 0x0000)`.
`and, or, xor`	`RT,RA,RB`	`RT` ← `RA op RB`, where `op` is bitwise *and*, *or*, or *exclusive or*.
`andi, ori, xori`	`RT,RA,Iu`	As above, except the last operand is a 16-bit unsigned immediate value.
`beq, bne, blt, ble, bgt, bge`	`RT,target`	Branch to target if `RT` = 0, or if `RT` ≠ 0, or if `RT` < 0, or if `RT` ≤ 0, or if `RT` > 0, or if `RT` ≥ 0 (signed integer interpretation of `RT`).
`bt, bf`	`RT,target`	Branch true/false; same as `bne`/`beq` resp.
`cmpeq, cmpne, cmplt, cmple, cmpgt, cmpge, cmpltu, cmpleu, cmpgtu, cmpgeu`	`RT,RA,RB`	`RT` gets the result of comparing `RA` with `RB`; 0 if **false** and 1 if **true**. Mnemonics denote *compare for equality*, *inequality*, *less than*, and so on, as for the branch instructions; and in addition, the suffix “u” denotes an unsigned comparison.
`cmpieq, cmpine, cmpilt, cmpile, cmpigt, cmpige`	`RT,RA,I`	Like the `cmpeq` group, except the second comparand is a 16-bit signed immediate value.

`cmpiequ, cmpineu, cmpiltu, cmpileu, cmpigtu, cmpigeu`	`RT,RA,Iu`	Like the `cmpltu` group, except the second comparand is a 16-bit unsigned immediate value.
`ldbu, ldh, ldhu, ldw`	`RT,d(RA)`	Load an unsigned byte, signed halfword, unsigned halfword, or word into `RT` from memory at location `RA + d`, where `d` is a 16-bit signed immediate value.
`mulhs, mulhu`	`RT,RA,RB`	`RT` gets the high-order 32 bits of the product of `RA` and `RB`; signed and unsigned.
`not`	`RT,RA`	`RT` $\leftarrow$ bitwise one's-complement of `RA`.
`shl, shr, shrs`	`RT,RA,RB`	`RT` $\leftarrow$ `RA` shifted left or right by the amount given in the rightmost six bits of `RB`; 0-fill except for `shrs`, which is sign-fill. (The shift amount is treated modulo 64.)
`shli, shri, shrsi`	`RT,RA,Iu`	`RT` $\leftarrow$ `RA` shifted left or right by the amount given in the 5-bit immediate field.
`stb, sth, stw`	`RS,d(RA)`	Store a byte, halfword, or word, from `RS` into memory at location `RA + d`, where `d` is a 16-bit signed immediate value.

TABLE 1–3. ADDITIONAL INSTRUCTIONS FOR THE "FULL RISC"

Opcode Mnemonic	Operands	Description
abs, nabs	RT,RA	RT gets the absolute value, or the negative of the absolute value, of RA.
andc, eqv, nand, nor, orc	RT,RA,RB	Bitwise *and with complement* (of RB), *equivalence*, *negative and*, *negative or*, and *or with complement*.
extr	RT,RA,I,L	Extract bits I through I+L-1 of RA, and place them right-adjusted in RT, with 0-fill.
extrs	RT,RA,I,L	Like extr, but sign-fill.
ins	RT,RA,I,L	Insert bits 0 through L-1 of RA into bits I through I+L-1 of RT.
nlz	RT,RA	RT gets the number of leading 0's in RA (0 to 32).
pop	RT,RA	RT gets the number of 1-bits in RA (0 to 32).
ldb	RT,d(RA)	Load a signed byte into RT from memory at location RA + d, where d is a 16-bit signed immediate value.
moveq, movne, movlt, movle, movgt, movge	RT,RA,RB	RT $\leftarrow$ RB if RA = 0, or if RA $\neq$ 0, and so on, else RT is unchanged.
shlr, shrr	RT,RA,RB	RT $\leftarrow$ RA rotate-shifted left or right by the amount given in the rightmost five bits of RB.
shlri, shrri	RT,RA,Iu	RT $\leftarrow$ RA rotate-shifted left or right by the amount given in the 5-bit immediate field.
trpeq, trpne, trplt, trple, trpgt, trpge, trpltu, trpleu, trpgtu, trpgeu	RA,RB	*Trap* (interrupt) if RA = RB, or RA $\neq$ RB, and so on.
trpieq, trpine, trpilt, trpile, trpigt, trpige	RA,I	Like the trpeq group, except the second comparand is a 16-bit signed immediate value.
trpiequ, trpineu, trpiltu, trpileu, trpigtu, trpigeu	RA,Iu	Like the trpltu group, except the second comparand is a 16-bit unsigned immediate value.

In Tables 1–2, 1–3, and 1–4, RA and RB appearing as source operands really means the contents of those registers.

A real machine would have branch and link (for subroutine calls), branch to the address contained in a register (for subroutine returns and "switches"), and possibly some instructions for dealing with special purpose registers. It would, of course, have a number of privileged instructions and instructions for calling on supervisor services. It might also have floating-point instructions.

Some other computational instructions that a RISC computer might have are identified in Table 1–3. These are discussed in later chapters.

It is convenient to provide the machine's assembler with a few "extended mnemonics." These are like macros whose expansion is usually a single instruction. Some possibilities are shown in Table 1–4.

TABLE 1–4. EXTENDED MNEMONICS

Extended Mnemonic	Expansion	Description
`b    target`	`beq  R0,target`	*Unconditional branch.*
`li   RT,I`	See text	*Load immediate*, $-2^{31} \le \mathtt{I} < 2^{32}$.
`mov  RT,RA`	`ori  RT,RA,0`	*Move register* RA to RT.
`neg  RT,RA`	`sub  RT,R0,RA`	*Negate* (two's-complement).
`subi RT,RA,I`	`addi RT,RA,-I`	*Subtract immediate* ($\mathtt{I} \ne -2^{15}$).

The *load immediate* instruction expands into one or two instructions, as required by the immediate value *I*. For example, if $0 \le I < 2^{16}$, an *or immediate* (`ori`) from R0 can be used. If $-2^{15} \le I < 0$, an *add immediate* (`addi`) from R0 can be used. If the rightmost 16 bits of *I* are 0, *add immediate shifted* (`addis`) can be used. Otherwise, two instructions are required, such as `addis` followed by `ori`. (Alternatively, in the last case, a load from memory could be used, but for execution time and space estimates we assume that two elementary arithmetic instructions are used.)

Of course, which instructions belong in the basic RISC and which belong in the full RISC is very much a matter of judgment. Quite possibly, *divide unsigned* and the *remainder* instructions should be moved to the full RISC category. Conversely, possibly *load byte signed* should be in the basic RISC category. It is in the full RISC set because it is probably of rather low frequency of use, and because in some technologies it is difficult to propagate a sign bit through so many positions and still make cycle time.

The distinction between basic and full RISC involves many other such questionable judgments, but we won't dwell on them.

The instructions are limited to two source registers and one target, which simplifies the computer (e.g., the register file requires no more than two read ports and one write port). It also simplifies an optimizing compiler, because the compiler does not need to deal with instructions that have multiple targets. The price paid for this is that a program that wants both the quotient and remainder of two numbers (not uncommon) must execute two instructions (*divide* and *remainder*). The usual machine division algorithm produces the remainder as a by-product, so many machines make them both available as a result of one execution of *divide*. Similar remarks apply to obtaining the doubleword product of two words.

The *conditional move* instructions (e.g., `moveq`) ostensibly have only two source operands, but in a sense they have three. Because the result of the instruction depends on the values in RT, RA, and RB, a machine that executes instructions out of order must treat RT in these instructions as both a *use* and a *set*. That is, an instruction that sets RT, followed by a *conditional move* that sets RT, must be executed in that order, and the result of the first instruction cannot be discarded. Thus, the designer of such a machine may elect to omit the *conditional move* instructions to avoid having to consider an instruction with (logically) three source operands. On the other hand, the *conditional move* instructions do save branches.

Instruction formats are not relevant to the purposes of this book, but the full RISC instruction set described above, with floating-point and a few supervisory instructions added, can be implemented with 32-bit instructions on a machine with 32 general purpose registers (5-bit register fields). By reducing the immediate fields of *compare, load, store*, and *trap* instructions to 14 bits, the same holds for a machine with 64 general purpose registers (6-bit register fields).

Execution Time

We assume that all instructions execute in one cycle, except for the *multiply, divide*, and *remainder* instructions, for which we do not assume any particular execution time. Branches take one cycle whether they branch or fall through.

The *load immediate* instruction is counted as one or two cycles, depending on whether one or two elementary arithmetic instructions are required to generate the constant in a register.

Although *load* and *store* instructions are not often used in this book, we assume they take one cycle and ignore any load delay (time lapse between when a load instruction completes in the arithmetic unit and when the requested data is available for a subsequent instruction).

However, knowing the number of cycles used by all the arithmetic and logical instructions is often insufficient for estimating the execution time of a program. Execution can be slowed substantially by load delays and by delays in fetching instructions. These delays, although very important and increasing in importance, are not discussed in this book. Another factor, one that improves execution time, is what is called "instruction-level parallelism," which is found in many contemporary RISC chips, particularly those for "high-end" machines.

These machines have multiple execution units and sufficient instruction-dispatching capability to execute instructions in parallel when they are independent (that is, when neither uses a result of the other, and they don't both set the same register or status bit). Because this capability is now quite common, the presence of independent operations is often pointed out in this book. Thus, we might say that such and such a formula can be coded in such a way that it requires eight instructions and executes in five cycles on a machine with unlimited instruction-level parallelism. This means that if the instructions are arranged in the proper order ("scheduled"), a machine with a sufficient number of adders, shifters, logical units, and registers can, in principle, execute the code in five cycles.

We do not make too much of this, because machines differ greatly in their instruction-level parallelism capabilities. For example, an IBM RS/6000 processor from ca. 1992 has a three-input adder and can execute two consecutive *add*-type instructions in parallel even when one feeds the other (e.g., an *add* feeding a *compare*, or the base register of a *load*). As a contrary example, consider a simple computer, possibly for low-cost embedded applications, that has only one read port on its register file. Normally, this machine would take an extra cycle to do a second read of the register file for an instruction that has two register input operands. However, suppose it has a bypass so that if an instruction feeds an operand of the immediately following instruction, then that operand is available without reading the register file. On such a machine, it is actually advantageous if each instruction feeds the next—that is, if the code has no parallelism.

Exercises

1. Express the loop

`for (`e_1`;` e_2`;` e_3`)` *statement*

in terms of a `while` loop.

Can it be expressed as a `do` loop?

2. Code a loop in C in which the unsigned integer control variable `i` takes on all values from 0 to and including the maximum unsigned number, 0xFFFFFFFF (on a 32-bit machine).

3. For the more experienced reader: The instructions of the basic and full RISCs defined in this book can be executed with at most two register reads and one write. What are some common or plausible RISC instructions that either need more source operands or need to do more than one register write?

Chapter 2. Basics

2–1 Manipulating Rightmost Bits

Some of the formulas in this section find application in later chapters.

Use the following formula to turn off the rightmost 1-bit in a word, producing 0 if none (e.g., 01011000   01010000):

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

This can be used to determine if an unsigned integer is a power of 2 or is 0: apply the formula followed by a 0-test on the result.

Use the following formula to turn on the rightmost 0-bit in a word, producing all 1's if none (e.g., 10100111   10101111):

$$x \mid (x + 1)$$

Use the following formula to turn off the trailing 1's in a word, producing ***x*** if none (e.g., 10100111   10100000):

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

This can be used to determine if an unsigned integer is of the form $2^n - 1$, 0, or all 1's: apply the formula followed by a 0-test on the result.

Use the following formula to turn on the trailing 0's in a word, producing ***x*** if none (e.g., 10101000   10101111):

$$x \mid (x - 1)$$

Use the following formula to create a word with a single 1-bit at the position of the rightmost 0-bit in ***x***, producing 0 if none (e.g., 10100111   00001000):

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

Use the following formula to create a word with a single 0-bit at the position of the rightmost 1-bit in ***x***, producing all 1's if none (e.g., 10101000   11110111):

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

Use one of the following formulas to create a word with 1's at the positions of the trailing 0's in ***x***, and 0's elsewhere, producing 0 if none (e.g., 01011000   00000111):

$$\neg x \,\&\, (x - 1), \quad \text{or}$$
$$\neg(x \mid -x), \quad \text{or}$$
$$(x \,\&\, -x) - 1$$

The first formula has some instruction-level parallelism.

Use the following formula to create a word with 0's at the positions of the trailing

1's in ***x***, and 0's elsewhere, producing all 1's if none (e.g., 10100111   11111000):

$$\neg \boldsymbol{x} \mid (\boldsymbol{x} + \mathbf{1})$$

Use the following formula to isolate the rightmost 1-bit, producing 0 if none (e.g., 01011000   00001000):

$$\boldsymbol{x} \,\&\, (-\boldsymbol{x})$$

Use the following formula to create a word with 1's at the positions of the rightmost 1-bit and the trailing 0's in ***x***, producing all 1's if no 1-bit, and the integer 1 if no trailing 0's (e.g., 01011000   00001111):

$$\boldsymbol{x} \quad (\boldsymbol{x} - \mathbf{1})$$

Use the following formula to create a word with 1's at the positions of the rightmost 0-bit and the trailing 1's in ***x***, producing all 1's if no 0-bit, and the integer 1 if no trailing 1's (e.g., 01010111   00001111):

$$\boldsymbol{x} \quad (\boldsymbol{x} + \mathbf{1})$$

Use either of the following formulas to turn off the rightmost contiguous string of 1's (e.g., 01011100 ==> 01000000) [Wood]:

$$(((\boldsymbol{x} \mid (\boldsymbol{x} - 1)) + 1) \,\&\, \boldsymbol{x}), \text{ or}$$
$$((\boldsymbol{x} \,\&\, -\boldsymbol{x}) + \boldsymbol{x}) \& \boldsymbol{x}$$

These can be used to determine if a nonnegative integer is of the form $2^j - 2^k$ for some $j \geq k \geq 0$: apply the formula followed by a 0-test on the result.

De Morgan's Laws Extended

The logical identities known as De Morgan's laws can be thought of as distributing, or "multiplying in," the ***not*** sign. This idea can be extended to apply to the expressions of this section, and a few more, as shown here. (The first two are De Morgan's laws.)

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

As an example of the application of these formulas, $\neg(\boldsymbol{x} \mid -(\boldsymbol{x} + \mathbf{1})) = \neg\boldsymbol{x} \,\&\, \neg-(\boldsymbol{x} + \mathbf{1}) = \neg\boldsymbol{x} \,\&\, ((\boldsymbol{x} + \mathbf{1}) - \mathbf{1}) = \neg\boldsymbol{x} \,\&\, \boldsymbol{x} = 0$.

Right-to-Left Computability Test

There is a simple test to determine whether or not a given function can be implemented with a sequence of ***add***'s, ***subtract***'s, ***and***'s, ***or***'s, and ***not***'s [War]. We can, of course, expand the list with other instructions that can be composed from the basic list, such as ***shift left*** by a fixed amount (which is equivalent to a sequence of ***add***'s), or ***multiply***. However, we exclude instructions that cannot be composed from the list. The test is contained in the following theorem.

THEOREM. ***A function mapping words to words can be implemented with word-parallel*** add, subtract, and, or, ***and*** not ***instructions if and only if each bit of the result depends only on bits at and to the right of each input operand***.

That is, imagine trying to compute the rightmost bit of the result by looking only at the rightmost bit of each input operand. Then, try to compute the next bit to the left by looking only at the rightmost two bits of each input operand, and continue in this way. If you are successful in this, then the function can be computed with a sequence of ***add***'s, ***and***'s, and so on. If the function cannot be computed in this right-to-left manner, then it cannot be implemented with a sequence of such instructions.

The interesting part of this is the latter statement, and it is simply the contra-positive of the observation that the functions ***add, subtract, and, or***, and ***not*** can all be computed in the right-to-left manner, so any combination of them must have this property.

To see the "if" part of the theorem, we need a construction that is a little awkward to explain. We illustrate it with a specific example. Suppose that a function of two variables ***x*** and ***y*** has the right-to-left computability property, and suppose that bit 2 of the result ***r*** is given by

$$r_2 = x_2 \mid (x_0 \,\&\, y_1). \tag{1}$$

We number bits from right to left, 0 to 31. Because bit 2 of the result is a function of bits at and to the right of bit 2 of the input operands, bit 2 of the result is "right-to-left computable."

Arrange the computer words ***x, x*** shifted left two, and ***y*** shifted left one, as shown below. Also, add a mask that isolates bit 2.

$$\begin{array}{cccccccc}
x_{31} & x_{30} & \cdots & x_3 & x_2 & x_1 & x_0 \\
x_{29} & x_{28} & \cdots & x_1 & x_0 & 0 & 0 \\
y_{30} & y_{29} & \cdots & y_2 & y_1 & y_0 & 0 \\
0 & 0 & \ldots & 0 & 1 & 0 & 0 \\
0 & 0 & \ldots & 0 & r_2 & 0 & 0
\end{array}$$

Now, form the word-parallel ***and*** of lines 2 and 3, ***or*** the result with row 1 (following Equation (1)), and ***and*** the result with the mask (row 4 above). The result is a word of all 0's except for the desired result bit in position 2. Perform similar computations for the other bits of the result, ***or*** the 32 resulting words together, and the result is the desired function.

This construction does not yield an efficient program; rather, it merely shows that it can be done with instructions in the basic list.

Using the theorem, we immediately see that there is no sequence of such instructions that turns off the leftmost 1-bit in a word, because to see if a certain 1-bit should be turned off, we must look to the left to see if it is the leftmost one. Similarly, there can be no such sequence for performing a right shift, or a rotate shift, or a left shift by a variable amount, or for counting the number of trailing 0's in a word (to count trailing 0's, the rightmost bit of the result will be 1 if there are an odd number of trailing 0's, and we must look to the left of the rightmost position to determine that).

A Novel Application

An application of the sort of bit twiddling discussed above is the problem of finding the next higher number after a given number that has the same number of 1-bits. You might very well wonder why anyone would want to compute that. It has application where bit strings are used to represent subsets. The possible members of a set are listed in a linear array, and a subset is represented by a word or sequence of words in which bit ***i*** is on if member ***i*** is in the subset. Set unions are computed by the logical ***or*** of the bit strings, intersections by ***and***'s, and so on.

You might want to iterate through all the subsets of a given size. This is easily done if you have a function that maps a given subset to the next higher number (interpreting the subset string as an integer) with the same number of 1-bits.

A concise algorithm for this operation was devised by R. W. Gosper [HAK, item 175].[1] Given a word ***x*** that represents a subset, the idea is to find the rightmost contiguous group of 1's in ***x*** and the following 0's, and "increment" that quantity to the next value that has the same number of 1's. For example, the string xxx0 1111 0000, where xxx represents arbitrary bits, becomes xxx1 0000 0111. The algorithm first identifies the "smallest" 1-bit in ***x***, with ***s*** = ***x*** &–***x***, giving 0000 0001 0000. This is added to ***x***, giving ***r*** = xxx1 0000 0000. The 1-bit here is one bit of the result. For the other bits, we need to produce a right-adjusted string of ***n*** – 1 1's, where ***n*** is the size of the rightmost group of 1's in ***x***. This can be done by first forming the ***exclusive or*** of ***r*** and ***x***, which gives 0001 1111 0000 in our example.

This has two too many 1's and needs to be right-adjusted. This can be accomplished by dividing it by ***s***, which right-adjusts it (***s*** is a power of 2), and shifting it right two more positions to discard the two unwanted bits. The final result is the ***or*** of this and ***r***.

In computer algebra notation, the result is ***y*** in

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

A complete C procedure is given in Figure 2–1. It executes in seven basic RISC instructions, one of which is division. (Do not use this procedure with ***x*** = 0; that causes division by 0.)

If division is slow but you have a fast way to compute the ***number of trailing zeros*** function ntz(***x***), the ***number of leading zeros*** function nlz(***x***), or ***population count*** (pop(***x***) is the number of 1-bits in ***x***), then the last line of Equation (2) can be replaced with one of the following formulas. (The first two methods can fail on a machine that has modulo 32 shifts.)

$$y \leftarrow r \mid ((x \oplus r) \overset{u}{\gg} (2 + \text{ntz}(x)))$$

$$y \leftarrow r \mid ((x \oplus r) \overset{u}{\gg} (33 - \text{nlz}(s)))$$

$$y \leftarrow r \mid ((1 \ll (\text{pop}(x \oplus r) - 2)) - 1)$$

```
unsigned snoob(unsigned x) {
   unsigned smallest, ripple, ones;
                                // x = xxx0 1111 0000
   smallest = x & -x;           //     0000 0001 0000
   ripple = x + smallest;       //     xxx1 0000 0000
   ones = x ^ ripple;           //     0001 1111 0000
   ones = (ones >> 2)/smallest; //     0000 0000 0111
   return ripple | ones;        //     xxx1 0000 0111
}
```

FIGURE 2–1. Next higher number with same number of 1-bits.

2–2 Addition Combined with Logical Operations

We assume the reader is familiar with the elementary identities of ordinary algebra and Boolean algebra. Below is a selection of similar identities involving addition and subtraction combined with logical operations.

$$
\begin{array}{lrcl}
\text{a.} & -x & = & \neg x + 1 \\
\text{b.} & & = & \neg(x-1) \\
\text{c.} & \neg x & = & -x-1 \\
\text{d.} & -\neg x & = & x+1 \\
\text{e.} & \neg -x & = & x-1 \\
\text{f.} & x+y & = & x - \neg y - 1 \\
\text{g.} & & = & (x \oplus y) + 2(x \,\&\, y) \\
\text{h.} & & = & (x \mid y) + (x \,\&\, y) \\
\text{i.} & & = & 2(x \mid y) - (x \oplus y) \\
\text{j.} & x-y & = & x + \neg y + 1 \\
\text{k.} & & = & (x \oplus y) - 2(\neg x \,\&\, y) \\
\text{l.} & & = & (x \,\&\, \neg y) - (\neg x \,\&\, y) \\
\text{m.} & & = & 2(x \,\&\, \neg y) - (x \oplus y) \\
\text{n.} & x \oplus y & = & (x \mid y) - (x \,\&\, y) \\
\text{o.} & x \,\&\, \neg y & = & (x \mid y) - y \\
\text{p.} & & = & x - (x \,\&\, y) \\
\text{q.} & \neg(x-y) & = & y - x - 1 \\
\text{r.} & & = & \neg x + y \\
\text{s.} & x \equiv y & = & (x \,\&\, y) - (x \mid y) - 1 \\
\text{t.} & & = & (x \,\&\, y) + \neg(x \mid y) \\
\text{u.} & x \mid y & = & (x \,\&\, \neg y) + y \\
\text{v.} & x \,\&\, y & = & (\neg x \mid y) - \neg x
\end{array}
$$

Equation (d) can be applied to itself repeatedly, giving $-\neg-\neg x = x + 2$, and so on. Similarly, from (e) we have $\neg-\neg- x = x - 2$. So we can add or subtract any constant using only the two forms of complementation.

Equation (f) is the dual of (j), where (j) is the well-known relation that shows how to build a subtracter from an adder.

Equations (g) and (h) are from HAKMEM memo [HAK, item 23]. Equation (g) forms a sum by first computing the sum with carries ignored ($x \quad y$), and then adding in the carries. Equation (h) is simply modifying the addition operands so that the combination 0 + 1 never occurs at any bit position; it is replaced with 1 + 0.

It can be shown that in the ordinary addition of binary numbers with each bit independently equally likely to be 0 or 1, a carry occurs at each position with probability about 0.5. However, for an adder built by preconditioning the inputs using (g), the probability is about 0.25. This observation is probably not of value in building an adder, because for that purpose the important characteristic is the maximum number of logic circuits the carry must pass through, and using (g) reduces the number of stages the carry propagates through by only one.

Equations (k) and (l) are duals of (g) and (h), for subtraction. That is, (k) has the interpretation of first forming the difference ignoring the borrows (***x*** ***y***), and then subtracting the borrows. Similarly, Equation (l) is simply modifying the subtraction operands so that the combination 1 – 1 never occurs at any bit position; it is replaced with 0 – 0.

Equation (n) shows how to implement ***exclusive or*** in only three instructions on a basic RISC. Using only ***and-or-not*** logic requires four instructions ((***x*** | ***y***) & ¬(***x*** & ***y***)). Similarly, (u) and (v) show how to implement ***and*** and ***or*** in three other elementary instructions, whereas using DeMorgan's laws requires four.

2–3 Inequalities among Logical and Arithmetic Expressions

Inequalities among binary logical expressions whose values are interpreted as unsigned integers are nearly trivial to derive. Here are two examples:

$$(x \oplus y) \overset{u}{\le} (x \mid y), \quad \text{and}$$

$$(x \,\&\, y) \overset{u}{\le} (x \equiv y).$$

These can be derived from a list of all binary logical operations, shown in Table 2–1.

TABLE 2–1. THE 16 BINARY LOGICAL OPERATIONS

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

Let ***f***(***x, y***) and ***g***(***x, y***) represent two columns in Table 2–1. If for each row in which ***f***(***x,y***) is 1, ***g***(***x,y***) also is 1, then for all (***x,y***), ***f***(***x, y***) $\overset{u}{\le}$ ***g***(***x, y***). Clearly, this extends to word-parallel logical operations. One can easily read off such relations (most of which are trivial) as (***x*** & ***y***) $\overset{u}{\le}$ ***x*** $\overset{u}{\le}$ (***x*** | ¬ ***y***), and so on. Furthermore, if two columns have a row in which one entry is 0 and the other is 1, and another row in which the entries are 1 and 0, respectively, then no inequality relation exists between the corresponding logical expressions. So the question of whether or not ***f***(***x, y***) $\overset{u}{\le}$ ***g***(***x, y***) is completely and easily solved for all binary logical functions ***f*** and ***g***.

Use caution when manipulating these relations. For example, for ordinary arithmetic, if $x + y \leq a$ and $z \leq x$, then $z + y \leq a$, but this inference is not valid if "+" is replaced with ***or***.

Inequalities involving mixed logical and arithmetic expressions are more interesting. Below is a small selection.

a. $(x \mid y) \overset{u}{\geq} \max(x, y)$

b. $(x \,\&\, y) \overset{u}{\leq} \min(x, y)$

c. $(x \mid y) \overset{u}{\leq} x + y$ if the addition does not overflow

d. $(x \mid y) \overset{u}{>} x + y$ if the addition overflows

e. $|x - y| \overset{u}{\leq} (x \oplus y)$

The proofs of these are quite simple, except possibly for the relation $|x - y| \overset{u}{\leq} (x \quad y)$. By $|x - y|$ we mean the absolute value of $x - y$, which can be computed within the domain of unsigned numbers as $\max(x, y) - \min(x, y)$. This relation can be proven by induction on the length of x and y (the proof is a little easier if you extend them on the left rather than on the right).

2–4 *Absolute Value* Function

If your machine does not have an instruction for computing the absolute value, this computation can usually be done in three or four branch-free instructions. First, compute $y \leftarrow x \overset{s}{\gg} 31$, and then one of the following:

abs	nabs
$(x \oplus y) - y$	$y - (x \oplus y)$
$(x + y) \oplus y$	$(y - x) \oplus y$
$x - (2x \,\&\, y)$	$(2x \,\&\, y) - x$

By "***2x***" we mean, of course, $x + x$ or $x << 1$.

If you have fast multiplication by a variable whose value is ±1, the following will do:

$$((x \overset{s}{\gg} 30) \mid 1) * x$$

2–5 Average of Two Integers

The following formula can be used to compute the average of two unsigned integers, $(x + y)/2$ without causing overflow [Dietz]:

$$(x \,\&\, y) + ((x \oplus y) \overset{u}{\gg} 1) \qquad (3)$$

The formula below computes $(x + y)/2$ for unsigned integers:

$$(x \mid y) - ((x \oplus y) \overset{u}{\gg} 1)$$

To compute the same quantities (“floor and ceiling averages”) for signed integers, use the same formulas, but with the unsigned shift replaced with a signed shift.

For signed integers, one might also want the average with the division by 2 rounded toward 0. Computing this “truncated average” (without causing overflow) is a little more difficult. It can be done by computing the floor average and then correcting it. The correction is to add 1 if, arithmetically, ***x*** + ***y*** is negative and odd. But ***x*** + ***y*** is negative if and only if the result of (3), with the unsigned shift replaced with a signed shift, is negative. This leads to the following method (seven instructions on the basic RISC, after commoning the subexpression ***x***   ***y***):

$$t \leftarrow (x \,\&\, y) + ((x \oplus y) \overset{s}{\gg} 1);$$

$$t + ((t \overset{u}{\gg} 31) \,\&\, (x \oplus y))$$

Some common special cases can be done more efficiently. If ***x*** and ***y*** are signed integers and known to be nonnegative, then the average can be computed as simply $(x + y) \overset{u}{\gg} 1$. The sum can overflow, but the overflow bit is retained in the register that holds the sum, so that the unsigned shift moves the overflow bit to the proper position and supplies a zero sign bit.

If ***x*** and ***y*** are unsigned integers and $x \overset{u}{\le} y$, or if ***x*** and ***y*** are signed integers and ***x*** **≤** ***y*** (signed comparison), then the average is given by ***x*** + $(y - x) \overset{u}{\gg} 1$. These are floor averages, for example, the average of −1 and 0 is −1.

2–6 Sign Extension

By “sign extension,” we mean to consider a certain bit position in a word to be the sign bit, and we wish to propagate that to the left, ignoring any other bits present. The standard way to do this is with ***shift left logical*** followed by ***shift right signed***. However, if these instructions are slow or nonexistent on your machine, it can be done with one of the following, where we illustrate by propagating bit position 7 to the left:

$$((x + \mathbf{0x00000080}) \,\&\, \mathbf{0x000000FF}) - \mathbf{0x00000080}$$

$$((x \,\&\, \mathbf{0x000000FF}) \oplus \mathbf{0x00000080}) - \mathbf{0x00000080}$$

$$(x \,\&\, \mathbf{0x0000007F}) - (x \,\&\, \mathbf{0x00000080})$$

The “+” above can also be “–” or “ .” The second formula is particularly useful if you know that the unwanted high-order bits are all 0’s, because then the ***and*** can be omitted.

2–7 Shift Right Signed from Unsigned

If your machine does not have the ***shift right signed*** instruction, it can be computed using the formulas shown below. The first formula is from [GM], and the second is based on the same idea. These formulas hold for 0 ≤ ***n*** ≤ 31 and, if the machine has mod-64 shifts, the last holds for 0 ≤ ***n*** ≤ 63. The last formula holds for any ***n*** if by “holds” we mean “treats the shift amount to the same modulus as does the logical shift.”

When ***n*** is a variable, each formula requires five or six instructions on a basic RISC.

$$((x + \text{0x80000000}) \overset{u}{\gg} n) - (\text{0x80000000} \overset{u}{\gg} n)$$

$$t \leftarrow \text{0x80000000} \overset{u}{\gg} n; \qquad ((x \overset{u}{\gg} n) \oplus t) - t$$

$$t \leftarrow (x \,\&\, \text{0x80000000}) \overset{u}{\gg} n;\ (x \overset{u}{\gg} n) - (t + t)$$

$$(x \overset{u}{\gg} n) \mid (-(x \overset{u}{\gg} 31) \ll 31 - n)$$

$$t \leftarrow -(x \overset{u}{\gg} 31); \qquad ((x \oplus t) \overset{u}{\gg} n) \oplus t$$

In the first two formulas, an alternative for the expression $\text{0x80000000} \overset{u}{\gg} n$ is **1**<<**31 −** ***n***.

If ***n*** is a constant, the first two formulas require only three instructions on many machines. If ***n*** = 31, the function can be done in two instructions with $-(x \overset{u}{\gg} 31)$.

2–8 *Sign* Function

The ***sign***, or ***signum***, function is defined by

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

It can be calculated with four instructions on most machines [Hop]:

$$(x \overset{s}{\gg} 31) \mid (-x \overset{u}{\gg} 31)$$

If you don't have ***shift right signed***, then use the substitute noted at the end of Section 2–7, giving the following nicely symmetric formula (five instructions):

$$-(x \overset{u}{\gg} 31) \mid (-x \overset{u}{\gg} 31)$$

Comparison predicate instructions permit a three-instruction solution, with either

$$\begin{aligned} &(x > 0) - (x < 0), \text{ or} \\ &(x \ge 0) - (x \le 0). \end{aligned} \tag{4}$$

Finally, we note that the formula $(-x \overset{u}{\gg} 31) - (x \overset{u}{\gg} 31)$ almost works; it fails only for $x = -2^{31}$.

2–9 *Three-Valued Compare* Function

The ***three-valued compare*** function, a slight generalization of the ***sign*** function, is defined by

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

There are both signed and unsigned versions, and unless otherwise specified, this section applies to both.

Comparison predicate instructions permit a three-instruction solution, an obvious generalization of Equations in (4):

$$\begin{aligned} &(x > y) - (x < y), \quad \text{or} \\ &(x \ge y) - (x \le y). \end{aligned}$$

A solution for unsigned integers on PowerPC is shown below [CWG]. On this machine, "carry" is "not borrow."

```
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).
```

If limited to the instructions of the basic RISC, there does not seem to be any particularly good way to compute this function. The comparison predicates ***x*** < ***y, x*** ≤ ***y***, and so on, require about five instructions (see Section 2–12), leading to a solution in about 12 instructions (using a small amount of commonality in computing ***x*** < ***y*** and ***x*** > ***y***). On the basic RISC it's probably preferable to use compares and branches (six instructions executed worst case if compares can be commoned).

2–10 *Transfer of Sign* Function

The ***transfer of sign*** function, called ISIGN in Fortran, is defined by

$$\mathrm{ISIGN}(x, y) = \begin{cases} \mathrm{abs}(x), & y \ge 0, \\ -\mathrm{abs}(x), & y < 0. \end{cases}$$

This function can be calculated (modulo 2^{32}) with four instructions on most machines:

$$\begin{aligned} t &\leftarrow y \overset{s}{\gg} 31; \\ \mathrm{ISIGN}(x, y) &= (\mathrm{abs}(x) \oplus t) - t \\ &= (\mathrm{abs}(x) + t) \oplus t \end{aligned} \qquad \begin{aligned} t &\leftarrow (x \oplus y) \overset{s}{\gg} 31; \\ \mathrm{ISIGN}(x, y) &= (x \oplus t) - t \\ &= (x + t) \oplus t \end{aligned}$$

2–11 Decoding a "Zero Means 2** *n*" Field

Sometimes a 0 or negative value does not make much sense for a quantity, so it is encoded in an ***n***-bit field with a 0 value being understood to mean 2***n***, and a nonzero value having its normal binary interpretation. An example is the length field of PowerPC's ***load string word immediate*** (`lswi`) instruction, which occupies five bits.

It is not useful to have an instruction that loads zero bytes when the length is an immediate quantity, but it is definitely useful to be able to load 32 bytes. The length field could be encoded with values from 0 to 31 denoting lengths from 1 to 32, but the “zero means 32” convention results in simpler logic when the processor must also support a corresponding instruction with a variable (in-register) length that employs straight binary encoding (e.g., PowerPC’s `lswx` instruction).

It is trivial to encode an integer in the range 1 to 2^n into the “zero means 2^n” encoding—simply mask the integer with $2^n - 1$. To do the decoding without a test-and-branch is not quite as simple, but here are some possibilities, illustrated for a 3-bit field. They all require three instructions, not counting possible loads of constants.

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

2–12 Comparison Predicates

A “comparison predicate” is a function that compares two quantities, producing a single bit result of 1 if the comparison is **true**, and 0 if the comparison is **false**. Below we show branch-free expressions to evaluate the result into the sign position. To produce the 1/0 value used by some languages (e.g., C), follow the code with a ***shift right*** of 31. To produce the −1/0 result used by some other languages (e.g., Basic), follow the code with a ***shift right signed*** of 31.

These formulas are, of course, not of interest on machines such as MIPS and our model RISC, which have comparison instructions that compute many of these predicates directly, placing a 0/1-valued result in a general purpose register.

$$
\begin{array}{lll}
x = y: & \text{abs}(x - y) - 1 & \\
 & \text{abs}(x - y + \mathbf{0x80000000}) & \\
 & \text{nlz}(x - y) \ll 26 & \\
 & -(\text{nlz}(x - y) \overset{u}{\gg} 5) & \\
 & \neg(x - y \mid y - x) & \\
x \neq y: & \text{nabs}(x - y) & \\
 & \text{nlz}(x - y) - \mathbf{32} & \\
 & x - y \mid y - x & \\
x < y: & (x - y) \oplus [(x \oplus y) \,\&\, ((x - y) \oplus x)] & \\
 & (x \,\&\, \neg y) \mid ((x \equiv y) \,\&\, (x - y)) & \\
 & \text{nabs}(\text{doz}(y, x)) & \text{[GSO]} \\
x \leq y: & (x \mid \neg y) \,\&\, ((x \oplus y) \mid \neg(y - x)) & \\
 & ((x \equiv y) \overset{s}{\gg} 1) + (x \,\&\, \neg y) & \text{[GSO]} \\
x \overset{u}{<} y: & (\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x - y)) & \\
 & (\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y)) & \\
x \overset{u}{\leq} y: & (\neg x \mid y) \,\&\, ((x \oplus y) \mid \neg(y - x)) &
\end{array}
$$

A machine instruction that computes the negative of the absolute value is handy here. We show this function as “nabs.” Unlike absolute value, it is well defined in that it never overflows. Machines that do not have nabs, but have the more usual abs, can use −abs(***x***) for nabs(***x***). If ***x*** is the maximum negative number, this overflows twice, but the result is correct. (We assume that the absolute value and the negation of the maximum negative number is itself.) Because some machines have neither abs nor nabs, we give an alternative that does not use them.

The “nlz” function is the number of leading 0’s in its argument. The “doz” function (***difference or zero***) is described on page 41. For ***x*** > ***y, x*** ≥ ***y***, and so on, interchange ***x*** and ***y*** in the formulas for ***x*** < ***y, x*** ≤ ***y***, and so on. The ***add*** of **0x8000 0000** can be replaced with any instruction that inverts the high-order bit (in ***x, y***, or ***x*** − ***y***).

Another class of formulas can be derived from the observation that the predicate ***x*** < ***y*** is given by the sign of ***x***/2 − ***y***/2, and the subtraction in that expression cannot overflow. The result can be fixed up by subtracting 1 in the cases in which the shifts discard essential information, as follows:

$$x < y: \qquad (x \overset{s}{\gg} 1) - (y \overset{s}{\gg} 1) - (\neg x \,\&\, y \,\&\, 1)$$

$$x \overset{u}{<} y: \qquad (x \overset{u}{\gg} 1) - (y \overset{u}{\gg} 1) - (\neg x \,\&\, y \,\&\, 1)$$

These execute in seven instructions on most machines (six if it has ***and not***), which is no better than what we have above (five to seven instructions, depending upon the fullness of the set of logic instructions).

The formulas above involving nlz are due to [Shep], and his formula for the $x = y$ predicate is particularly useful, because a minor variation of it gets the predicate evaluated to a 1/0-valued result with only three instructions:

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

Signed comparisons to 0 are frequent enough to deserve special mention. There are some formulas for these, mostly derived directly from the above. Again, the result is in the sign position.

$$
\begin{array}{ll}
x = 0: & \text{abs}(x) - 1 \\
 & \text{abs}(x + \mathbf{0x80000000}) \\
 & \text{nlz}(x) \ll 26 \\
 & -(\text{nlz}(x) \overset{u}{\gg} 5) \\
 & \neg(x \mid -x) \\
 & \neg x \;\&\; (x - 1) \\
x \neq 0: & \text{nabs}(x) \\
 & \text{nlz}(x) - 32 \\
 & x \mid -x \\
 & (x \overset{u}{\gg} 1) - x \qquad\qquad [\text{CWG}]
\end{array}
$$

$$
\begin{array}{ll}
x < 0: & x \\
x \leq 0: & x \mid (x - 1) \\
 & x \mid \neg -x \\
x > 0: & x \oplus \text{nabs}(x) \\
 & (x \overset{s}{\gg} 1) - x \\
 & -x \;\&\; \neg x \\
x \geq 0: & \neg x
\end{array}
$$

Signed comparisons can be obtained from their unsigned counterparts by biasing the signed operands upward by 2^{31} and interpreting the results as unsigned integers. The reverse transformation also works.[2] Thus, we have

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

Similar relations hold for $\leq$, $\overset{u}{\leq}$, and so on. In these relations, one can use addition, subtraction, or ***exclusive or*** with 2^{31}. They are all equivalent, as they simply invert the sign bit. An instruction like the basic RISC's ***add immediate shifted*** is useful to avoid loading the constant 2^{31}.

Another way to get signed comparisons from unsigned is based on the fact that if x

and ***y*** have the same sign, then $x < y = x \overset{u}{<} y$, whereas if they have opposite signs, then $x < y = x \overset{u}{>} y$ [Lamp]. Again, the reverse transformation also works, so we have

$$x < y = (x \overset{u}{<} y) \oplus x_{31} \oplus y_{31} \text{ and}$$

$$x \overset{u}{<} y = (x < y) \oplus x_{31} \oplus y_{31},$$

where x_{31} and y_{31} are the sign bits of ***x*** and ***y***, respectively. Similar relations hold for ≤, $\overset{u}{\leq}$, and so on.

Using either of these devices enables computing all the usual comparison predicates other than = and ≠ in terms of any one of them, with at most three additional instructions on most machines. For example, let us take $x \overset{u}{\leq} y$ as primitive, because it is one of the simplest to implement (it is the carry bit from ***y*** − ***x***). Then the other predicates can be obtained as follows:

$$x < y = \neg(y + 2^{31} \overset{u}{\leq} x + 2^{31})$$

$$x \leq y = x + 2^{31} \overset{u}{\leq} y + 2^{31}$$

$$x > y = \neg(x + 2^{31} \overset{u}{\leq} y + 2^{31})$$

$$x \geq y = y + 2^{31} \overset{u}{\leq} x + 2^{31}$$

$$x \overset{u}{<} y = \neg(y \overset{u}{\leq} x)$$

$$x \overset{u}{>} y = \neg(x \overset{u}{\leq} y)$$

$$x \overset{u}{\geq} y = y \overset{u}{\leq} x$$

Comparison Predicates from the Carry Bit

If the machine can easily deliver the carry bit into a general purpose register, this may permit concise code for some of the comparison predicates. Below are several of these relations. The notation carry(***expression***) means the carry bit generated by the outermost operation in ***expression***. We assume the carry bit for the subtraction ***x*** – ***y*** is what comes out of the adder for ***x*** + $\bar{y}$ + 1, which is the complement of "borrow."

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

For ***x*** > ***y***, use the complement of the expression for ***x*** ≤ ***y***, and similarly for other relations involving “greater than.”

The GNU Superoptimizer has been applied to the problem of computing predicate expressions on the IBM RS/6000 computer and its close relative PowerPC [GK]. The RS/6000 has instructions for abs(***x***), nabs(***x***), doz(***x, y***), and a number of forms of ***add*** and ***subtract*** that use the carry bit. It was found that the RS/6000 can compute all the integer predicate expressions with three or fewer elementary (one-cycle) instructions, a result that surprised even the architects of the machine. “All” includes the six two-operand signed comparisons and the four two-operand unsigned comparisons, all of these with the second operand being 0, and all in forms that produce a 1/0 result or a –1/0 result. PowerPC, which lacks abs(***x***), nabs(***x***), and doz(***x, y***), can compute all the predicate expressions in four or fewer elementary instructions.

How the Computer Sets the Comparison Predicates

Most computers have a way of evaluating the integer comparison predicates to a 1-bit result. The result bit may be placed in a “condition register” or, for some machines (such as our RISC model), in a general purpose register. In either case, the facility is often implemented by subtracting the comparison operands and then performing a small amount of logic on the result bits to determine the 1-bit comparison result.

Below is the logic for these operations. It is assumed that the machine computes ***x*** **−** ***y*** as ***x*** + $\bar{y}$ + 1, and the following quantities are available in the result:

C_o, the carry out of the high-order position

C_i, the carry into the high-order position

N, the sign bit of the result

Z, which equals 1 if the result, exclusive of C_o, is all-0, and is otherwise 0

Then we have the following in Boolean algebra notation (juxtaposition denotes ***and***, + denotes ***or***):

V:	$C_i \oplus C_o$	(signed overflow)
$x = y$:	Z	
$x \neq y$:	$\bar{Z}$	
$x < y$:	$N \oplus V$	
$x \leq y$:	$(N \oplus V) + Z$	
$x > y$:	$(N \equiv V)\bar{Z}$	
$x \geq y$:	$N \equiv V$	
$x \overset{u}{<} y$:	$\overline{C_o}$	
$x \overset{u}{\leq} y$:	$\overline{C_o} + Z$	
$x \overset{u}{>} y$:	$C_o\bar{Z}$	
$x \overset{u}{\geq} y$:	C_o	

2–13 Overflow Detection

"Overflow" means that the result of an arithmetic operation is too large or too small to be correctly represented in the target register. This section discusses methods that a programmer might use to detect when overflow has occurred, without using the machine's "status bits" that are often supplied expressly for this purpose. This is important, because some machines do not have such status bits (e.g., MIPS), and even if the machine is so equipped, it is often difficult or impossible to access the bits from a high-level language.

Signed Add/Subtract

When overflow occurs on integer addition and subtraction, contemporary machines invariably discard the high-order bit of the result and store the low-order bits that the adder naturally produces. Signed integer overflow of addition occurs if and only if the operands have the same sign and the sum has a sign opposite to that of the operands. Surprisingly, this same rule applies even if there is a carry into the adder—that is, if the calculation is $x + y + 1$. This is important for the application of adding multiword signed integers, in which the last addition is a signed addition of two fullwords and a carry-in that may be 0 or +1.

To prove the rule for addition, let x and y denote the values of the one-word signed integers being added, let c (carry-in) be 0 or 1, and assume for simplicity a 4-bit machine. Then if the signs of x and y are different,

$$-8 \leq x \leq -1, \text{ and}$$
$$0 \leq y \leq 7,$$

or similar bounds apply if x is nonnegative and y is negative. In either case, by adding these inequalities and optionally adding in 1 for c,

$$-8 \le x + y + c \le 7.$$

This is representable as a 4-bit signed integer, and thus overflow does not occur when the operands have opposite signs.

Now suppose x and y have the same sign. There are two cases:

$$\begin{array}{cc} \text{(a)} & \text{(b)} \\ -8 \le x \le -1 & 0 \le x \le 7 \\ -8 \le y \le -1 & 0 \le y \le 7 \end{array}$$

Thus,

$$\begin{array}{cc} \text{(a)} & \text{(b)} \\ -16 \le x + y + c \le -1 & 0 \le x + y + c \le 15. \end{array}$$

Overflow occurs if the sum is not representable as a 4-bit signed integer—that is, if

$$\begin{array}{cc} \text{(a)} & \text{(b)} \\ -16 \le x + y + c \le -9 & 8 \le x + y + c \le 15. \end{array}$$

In case (a), this is equivalent to the high-order bit of the 4-bit sum being 0, which is opposite to the sign of x and y. In case (b), this is equivalent to the high-order bit of the 4-bit sum being 1, which again is opposite to the sign of x and y.

For subtraction of multiword integers, the computation of interest is $x - y - c$, where again c is **0** or **1**, with a value of **1** representing a borrow-in. From an analysis similar to the above, it can be seen that overflow in the final value of $x - y - c$ occurs if and only if x and y have opposite signs and the sign of $x - y - c$ is opposite to that of x (or, equivalently, the same as that of y).

This leads to the following expressions for the overflow predicate, with the result being in the sign position. Following these with a ***shift right*** or ***shift right signed*** of 31 produces a 1/0- or a −1/0-valued result.

$$\begin{array}{ll} x + y + c & x - y - c \\ (x \equiv y) \mathbin{\&} ((x + y + c) \oplus x) & (x \oplus y) \mathbin{\&} ((x - y - c) \oplus x) \\ ((x + y + c) \oplus x) \mathbin{\&} ((x + y + c) \oplus y) & ((x - y - c) \oplus x) \mathbin{\&} ((x - y - c) \equiv y) \end{array}$$

By choosing the second alternative in the first column, and the first alternative in the second column (avoiding the ***equivalence*** operation), our basic RISC can evaluate these tests with three instructions in addition to those required to compute $x + y + c$ or $x - y - c$. A fourth instruction (***branch if negative***) can be added to branch to code where the overflow condition is handled.

If executing with overflow interrupts enabled, the programmer may wish to test to see if a certain addition or subtraction will cause overflow, in a way that does not cause it. One branch-free way to do this is as follows:

$$\begin{array}{cc} x + y + c & x - y - c \\ z \leftarrow (x \equiv y)\ \&\ \mathbf{0x80000000} & z \leftarrow (x \oplus y)\ \&\ \mathbf{0x80000000} \\ z\ \&\ (((x \oplus z) + y + c) \equiv y) & z\ \&\ (((x \oplus z) - y - c) \oplus y) \end{array}$$

The assignment to z in the left column sets z = **0x80000000** if x and y have the same sign, and sets z = **0** if they differ. Then, the addition in the second expression is done with x z and y having different signs, so it can't overflow. If x and y are nonnegative, the sign bit in the second expression will be 1 if and only if $(x - 2^{31}) + y + c \geq 0$—that is, iff $x + y + c \geq 2^{31}$, which is the condition for overflow in evaluating $x + y + c$. If x and y are negative, the sign bit in the second expression will be 1 iff $(x + 2^{31}) + y + c < 0$—that is, iff $x + y + c < -2^{31}$, which again is the condition for overflow. The ***and*** with z ensures the correct result (0 in the sign position) if x and y have opposite signs. Similar remarks apply to the case of subtraction (right column). The code executes in nine instructions on the basic RISC.

It might seem that if the carry from addition is readily available, this might help in computing the signed overflow predicate. This does not seem to be the case; however, one method along these lines is as follows.

If x is a signed integer, then $x + 2^{31}$ is correctly represented as an unsigned number and is obtained by inverting the high-order bit of x. Signed overflow in the positive direction occurs if $x + y \geq 2^{31}$—that is, if $(x + 2^{31}) + (y + 2^{31}) \geq 3 \cdot 2^{31}$. This latter condition is characterized by carry occurring in the unsigned add (which means that the sum is greater than or equal to 2^{32}) and the high-order bit of the sum being 1. Similarly, overflow in the negative direction occurs if the carry is 0 and the high-order bit of the sum is also 0.

This gives the following algorithm for detecting overflow for signed addition:

> Compute (x 2^{31}) + (y 2^{31}), giving sum s and carry c.
> Overflow occurred iff c equals the high-order bit of s.

The sum is the correct sum for the signed addition, because inverting the high-order bits of both operands does not change their sum.

For subtraction, the algorithm is the same except that in the first step a subtraction replaces the addition. We assume that the carry is that which is generated by computing $x - y$ as $x + \bar{y} + 1$. The subtraction is the correct difference for the signed subtraction.

These formulas are perhaps interesting, but on most machines they would not be quite as efficient as the formulas that do not even use the carry bit (e.g., overflow = ($x \equiv y$)& (s x) for addition, and (x y) &(d x) for subtraction, where s and d are the sum and difference, respectively, of x and y).

How the Computer Sets Overflow for Signed Add/Subtract

Machines often set "overflow" for signed addition by means of the logic "the carry into the sign position is not equal to the carry out of the sign position." Curiously, this logic gives the correct overflow indication for both addition and subtraction, assuming the subtraction $x - y$ is done by $x + \bar{y} + 1$. Furthermore, it is correct whether or not there is a carry- or borrow-in. This does not seem to lead to any particularly good methods for computing the signed overflow predicate in software, however, even though it is easy to compute the carry into the sign position. For addition and subtraction, the carry/borrow into the sign position is given by the sign bit after evaluating the following expressions (where ***c*** is **0** or **1**):

$$\begin{array}{cc} \text{carry} & \text{borrow} \\ (x+y+c) \oplus x \oplus y & (x-y-c) \oplus x \oplus y \end{array}$$

In fact, these expressions give, at each position ***i***, the carry/borrow into position ***i***.

Unsigned Add/Subtract

The following branch-free code can be used to compute the overflow predicate for unsigned add/subtract, with the result being in the sign position. The expressions involving a right shift are probably useful only when it is known that ***c*** = **0**. The expressions in brackets compute the carry or borrow generated from the least significant position.

$$x+y+c, \text{ unsigned}$$

$$(x \mathbin{\&} y) \mid ((x \mid y) \mathbin{\&} \neg(x+y+c))$$

$$(x \overset{u}{\gg} 1) + (y \overset{u}{\gg} 1) + [((x \mathbin{\&} y) \mid ((x \mid y) \mathbin{\&} c)) \mathbin{\&} \mathbf{1}]$$

$$x-y-c, \text{ unsigned}$$

$$(\neg x \mathbin{\&} y) \mid ((x \equiv y) \mathbin{\&} (x-y-c))$$

$$(\neg x \mathbin{\&} y) \mid ((\neg x \mid y) \mathbin{\&} (x-y-c))$$

$$(x \overset{u}{\gg} 1) - (y \overset{u}{\gg} 1) - [((\neg x \mathbin{\&} y) \mid ((\neg x \mid y) \mathbin{\&} c)) \mathbin{\&} \mathbf{1}]$$

For unsigned ***add***'s and ***subtract***'s, there are much simpler formulas in terms of comparisons [MIPS]. For unsigned addition, overflow (carry) occurs if the sum is less (by unsigned comparison) than either of the operands. This and similar formulas are given below. Unfortunately, there is no way in these formulas to allow for a variable ***c*** that represents the carry- or borrow-in. Instead, the program must test ***c***, and use a different type of comparison depending upon whether ***c*** is **0** or **1**.

$$\begin{array}{cccc} x+y, \text{ unsigned} & x+y+\mathbf{1}, \text{ unsigned} & x-y, \text{ unsigned} & x-y-\mathbf{1}, \text{ unsigned} \\ \neg x \overset{u}{<} y & \neg x \overset{u}{\le} y & x \overset{u}{<} y & x \overset{u}{\le} y \\ x+y \overset{u}{<} x & x+y+\mathbf{1} \overset{u}{\le} x & x-y \overset{u}{>} x & x-y-\mathbf{1} \overset{u}{\ge} x \end{array}$$

The first formula for each case above is evaluated before the add/subtract that may overflow, and it provides a way to do the test without causing overflow. The second formula for each case is evaluated after the add/subtract that may overflow.

There does not seem to be a similar simple device (using comparisons) for computing the signed overflow predicate.

Multiplication

For multiplication, overflow means that the result cannot be expressed in 32 bits (it can always be expressed in 64 bits, whether signed or unsigned). Checking for overflow is simple if you have access to the high-order 32 bits of the product. Let us denote the two halves of the 64-bit product by hi($x \times y$) and lo($x \times y$). Then the overflow predicates can be computed as follows [MIPS]:

$x \times y$, unsigned	$x \times y$, signed
$\text{hi}(x \times y) \neq 0$	$\text{hi}(x \times y) \neq (\text{lo}(x \times y) \overset{s}{\gg} 31)$

One way to check for overflow of multiplication is to do the multiplication and then check the result by dividing. Care must be taken not to divide by 0, and there is a further complication for signed multiplication. Overflow occurs if the following expressions are **true**:

Unsigned	Signed
$z \leftarrow x * y$	$z \leftarrow x * y$
$y \neq 0 \overrightarrow{\&} z \overset{u}{\div} y \neq x$	$(y < 0 \;\&\; x = -2^{31}) \mid (y \neq 0 \overrightarrow{\&} z \div y \neq x)$

The complication arises when $x = -2^{31}$ and $y = -1$. In this case the multiplication overflows, but the machine may very well give a result of -2^{31}. This causes the division to overflow, and thus any result is possible (for some machines). Therefore, this case has to be checked separately, which is done by the term $y < 0 \;\&\; x = -2^{31}$. The above expressions use the "conditional ***and***" operator to prevent dividing by 0 (in C, use the `&&` operator).

It is also possible to use division to check for overflow of multiplication without doing the multiplication (that is, without causing overflow). For unsigned integers, the product overflows iff $xy > 2^{32} - 1$, or $x > ((2^{32} - 1)/y)$, or, since x is an integer, $x > \lfloor (2^{32} - 1)/y \rfloor$. Expressed in computer arithmetic, this is

$$y \neq 0 \overrightarrow{\&} x \overset{u}{>} (\text{0xFFFFFFFF} \overset{u}{\div} y).$$

For signed integers, the determination of overflow of $x * y$ is not so simple. If x and y have the same sign, then overflow occurs iff $xy > 2^{31} - 1$. If they have opposite signs, then overflow occurs iff $xy < -2^{31}$. These conditions can be tested as indicated in Table 2–2, which employs signed division. This test is awkward to implement, because of the four cases. It is difficult to unify the expressions very much because of problems with overflow and with not being able to represent the number $+2^{31}$.

The test can be simplified if unsigned division is available. We can use the absolute values of x and y, which are correctly represented under unsigned integer interpretation. The complete test can then be computed as shown below. The variable $c = 2^{31} - 1$ if x and y have the same sign, and $c = 2^{31}$ otherwise.

TABLE 2–2. OVERFLOW TEST FOR SIGNED MULTIPLICATION

	$y > 0$	$y \le 0$
$x > 0$	$x > \mathbf{0x7FFFFFFF} \div y$	$y < \mathbf{0x80000000} \div x$
$x \le 0$	$x < \mathbf{0x80000000} \div y$	$x \ne \mathbf{0} \mathbin{\overrightarrow{\&}} y < \mathbf{0x7FFFFFFF} \div x$

$$c \leftarrow ((x \equiv y) \overset{s}{\gg} 31) + 2^{31}$$
$$x \leftarrow \mathrm{abs}(x)$$
$$y \leftarrow \mathrm{abs}(y)$$
$$y \ne \mathbf{0} \mathbin{\overrightarrow{\&}} x \overset{u}{>} (c \overset{u}{\div} y)$$

The ***number of leading zeros*** instruction can be used to give an estimate of whether or not ***x*** * ***y*** will overflow, and the estimate can be refined to give an accurate determination. First, consider the multiplication of unsigned numbers. It is easy to show that if ***x*** and ***y***, as 32-bit quantities, have ***m*** and ***n*** leading 0's, respectively, then the 64-bit product has either ***m*** + ***n*** or ***m*** + ***n*** + 1 leading 0's (or 64, if either ***x*** = **0** or ***y*** = **0**). Overflow occurs if the 64-bit product has fewer than 32 leading 0's. Hence,

nlz(***x***) + nlz(***y***) ≥ 32: Multiplication definitely does not overflow.
nlz(***x***) + nlz(***y***) ≤ 30: Multiplication definitely does overflow.

For nlz(***x***) + nlz(***y***) = 31, overflow may or may not occur. In this case, the overflow assessment can be made by evaluating ***t*** = ***x*** ***y***/2 . This will not overflow. Since ***xy*** is 2***t*** or, if ***y*** is odd, 2***t*** + ***x***, the product ***xy*** overflows if ***t*** ≥ 2^{31}. These considerations lead to a plan for computing ***xy***, but branching to "overflow" if the product overflows. This plan is shown in Figure 2–2.

For the multiplication of signed integers, we can make a partial determination of whether or not overflow occurs from the number of leading 0's of nonnegative arguments, and the number of leading 1's of negative arguments. Let

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

```
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.
```

FIGURE 2–2. Determination of overflow of unsigned multiplication.

Then, we have

> $m + n \geq 34$: Multiplication definitely does not overflow.
> $m + n \leq 31$: Multiplication definitely does overflow.

There are two ambiguous cases: 32 and 33. The case $m + n = 33$ overflows only when both arguments are negative and the true product is exactly 2^{31} (machine result is -2^{31}), so it can be recognized by a test that the product has the correct sign (that is, overflow occurred if $m \quad n \quad (m * n) < 0$). When $m + n = 32$, the distinction is not so easily made.

We will not dwell on this further, except to note that an overflow estimate for signed multiplication can also be made based on nlz(abs(x)) + nlz(abs(y)), but again there are two ambiguous cases (a sum of 31 or 32).

Division

For the signed division $x \div y$, overflow occurs if the following expression is **true**:

$$y = 0 \mid (x = \mathbf{0x80000000} \; \& \; y = -1)$$

Most machines signal overflow (or trap) for the indeterminate form $0 \div 0$.

Straightforward code for evaluating this expression, including a final branch to the overflow handling code, consists of seven instructions, three of which are branches. There do not seem to be any particularly good tricks to improve on this, but here are a few possibilities:

$$[\text{abs}(y \quad \mathbf{0x80000000}) \mid (\text{abs}(x) \; \& \; \text{abs}(y = \mathbf{0x80000000}))] < 0$$

That is, evaluate the large expression in brackets, and branch if the result is less than 0. This executes in about nine instructions, counting the load of the constant and the final branch, on a machine that has the indicated instructions and that gets the “compare to 0” for free.

Some other possibilities are to first compute z from

$$z \leftarrow (x \quad \mathbf{0x80000000}) \mid (y + 1)$$

(three instructions on many machines), and then do the test and branch on $y = 0 \mid z = 0$ in one of the following ways:

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

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

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

These execute in nine, seven, and eight instructions, respectively, on a machine that has the indicated instructions. The last line represents a good method for PowerPC.

For the unsigned division $x \overset{u}{\div} y$, overflow occurs if and only if $y = 0$.

Some machines have a “long division” instruction (see page 192), and you may want to predict, using elementary instructions, when it would overflow. We will discuss this in terms of an instruction that divides a doubleword by a fullword, producing a fullword quotient and possibly also a fullword remainder.

Such an instruction overflows if either the divisor is 0 or if the quotient cannot be represented in 32 bits. Typically, in these overflow cases both the quotient and remainder are incorrect. The remainder cannot overflow in the sense of being too large to represent in 32 bits (it is less than the divisor in magnitude), so the test that the remainder will be correct is the same as the test that the quotient will be correct.

We assume the machine either has 64-bit general registers or 32-bit registers and there is no problem doing elementary operations (shifts, adds, and so forth) on 64-bit quantities. For example, the compiler might implement a doubleword integer data type.

In the unsigned case the test is trivial: for $x \div y$ with x a doubleword and y a fullword, the division will not overflow if (and only if) either of the following equivalent expressions is true.

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

$$y \neq 0 \;\&\; (x \overset{u}{\gg} 32) < y$$

On a 32-bit machine, the shifts need not be done; simply compare y to the register that contains the high-order half of x. To ensure correct results on a 64-bit machine, it is also necessary to check that the divisor y is a 32-bit quantity (e.g., check that $(y \overset{u}{\gg} 32) = 0$).

The signed case is more interesting. It is first necessary to check that $y \neq 0$ and, on a 64-bit machine, that y is correctly represented in 32 bits (check that $((y \ll 32) \overset{s}{\gg} 32) = y$). Assuming these tests have been done, the table that follows shows how the tests might be done to determine precisely whether or not the quotient is representable in 32 bits by considering separately the four cases of the dividend and divisor each being positive or negative. The expressions in the table are in ordinary arithmetic, not computer arithmetic.

In each column, each relation follows from the one above it in an if-and-only-if way. To remove the floor and ceiling functions, some relations from Theorem D1 on page 183 are used.

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

As an example of interpreting this table, consider the leftmost column. It applies to the case in which $x \geq 0$ and $y > 0$. In this case the quotient is x/y, and this must be strictly less than 2^{31} to be representable as a 32-bit quantity. From this it follows that the real number x/y must be less than 2^{31}, or x must be less than $2^{31}y$. This test can be implemented by shifting y left 31 positions and comparing the result to x.

When the signs of x and y differ, the quotient of conventional division is x/y . Because the quotient is negative, it can be as small as -2^{31}.

In the bottom row of each column the comparisons are all of the same type (less than). Because of the possibility that x is the maximum negative number, in the third and fourth columns an unsigned comparison must be used. In the first two columns the quantities being compared begin with a leading 0-bit, so an unsigned comparison can be used there, too.

These tests can, of course, be implemented by using conditional branches to separate out the four cases, doing the indicated arithmetic, and then doing a final compare and branch to the code for the overflow or non-overflow case. However, branching can be reduced by taking advantage of the fact that when y is negative, $-y$ is used, and similarly for x. Hence the tests can be made more uniform by using the absolute values of x and y. Also, using a standard device for optionally doing the additions in the second and third columns results in the following scheme:

$$x' = |x|$$

$$y' = |y|$$

$$\delta = ((x \oplus y) \overset{s}{\gg} 63) \,\&\, y'$$

$$\text{if } (x' \overset{u}{<} (y' \ll 31) + \delta) \text{ then } \{\text{will not overflow}\}$$

Using the three-instruction method of computing the absolute value (see page 18), on a 64-bit version of the basic RISC this amounts to 12 instructions, plus a conditional branch.

2–14 Condition Code Result of *Add, Subtract*, and *Multiply*

Many machines provide a "condition code" that characterizes the result of integer arithmetic operations. Often there is only one ***add*** instruction, and the characterization reflects the result for both unsigned and signed interpretation of the operands and result (but not for mixed types). The characterization usually consists of the following:

- Whether or not carry occurred (unsigned overflow)
- Whether or not signed overflow occurred
- Whether the 32-bit result, interpreted as a signed two's-complement integer and ignoring carry and overflow, is negative, 0, or positive

Some older machines give an indication of whether the infinite precision result (that is, 33-bit result for ***add***'s and ***subtract***'s) is positive, negative, or 0. However, this indication is not easily used by compilers of high-level languages, and so has fallen out of favor.

For addition, only nine of the 12 combinations of these events are possible. The ones that cannot occur are "no carry, overflow, result > 0," "no carry, overflow, result = 0," and "carry, overflow, result < 0." Thus, four bits are, just barely, needed for the condition code. Two of the combinations are unique in the sense that only one value of inputs produces them: Adding 0 to itself is the only way to get "no carry, no overflow, result = 0," and adding the maximum negative number to itself is the only way to get "carry, overflow, result = 0." These remarks remain true if there is a "carry in"—that is, if we are computing $x + y + 1$.

For subtraction, let us assume that to compute ***x*** – ***y*** the machine actually computes ***x*** + $\bar{y}$ + 1, with the carry produced as for an ***add*** (in this scheme the meaning of "carry" is reversed for subtraction, in that carry = 1 signifies that the result fits in a single word, and carry = 0 signifies that the result does not fit in a single word). Then for subtraction, only seven combinations of events are possible. The ones that cannot occur are the three that cannot occur for addition, plus "no carry, no overflow, result = 0," and "carry, overflow, result = 0."

If a machine's multiplier can produce a doubleword result, then two ***multiply*** instructions are desirable: one for signed and one for unsigned operands. (On a 4-bit machine, in hexadecimal, **F** × **F** = **01** signed, and **F** × **F** = **E1** unsigned.) For these instructions, neither carry nor overflow can occur, in the sense that the result will always fit in a doubleword.

For a multiplication instruction that produces a one-word result (the low-order word of the doubleword result), let us take "carry" to mean that the result does not fit in a word with the operands and result interpreted as unsigned integers, and let us take "overflow" to mean that the result does not fit in a word with the operands and result interpreted as signed two's-complement integers. Then again, there are nine possible combinations of results, with the missing ones being "no carry, overflow, result > 0," "no carry, overflow, result = 0," and "carry, no overflow, result = 0." Thus, considering addition, subtraction, and multiplication together, ten combinations can occur.

2–15 Rotate Shifts

These are rather trivial. Perhaps surprisingly, this code works for ***n*** ranging from 0 to 32 inclusive, even if the shifts are mod-32.

$$\text{Rotate left } n\text{:}\quad y \leftarrow (x \ll n) \mid (x \overset{u}{\gg} (32 - n))$$

$$\text{Rotate right } n\text{:}\quad y \leftarrow (x \overset{u}{\gg} n) \mid (x \ll (32 - n))$$

If your machine has double-length shifts, they can be used to do rotate shifts. These instructions might be written

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

They treat the concatenation of RA and RB as a single double-length quantity, and shift it left or right by the amount given by the immediate field `I`. (If the shift amount is in a register, the instructions are awkward to implement on most RISCs because they require reading three registers.) The result of the left shift is the high-order word of the shifted double-length quantity, and the result of the right shift is the low-order word.

Using `shldi`, a rotate left of Rx can be accomplished by

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

and similarly a rotate right shift can be accomplished with `shrdi`.

A rotate left shift of one position can be accomplished by adding the contents of a register to itself with "end-around carry" (adding the carry that results from the addition to the sum in the low-order position). Most machines do not have that instruction, but on many machines it can be accomplished with two instructions: (1) add the contents of the register to itself, generating a carry (into a status register), and

(2) add the carry to the sum.

2–16 Double-Length Add/Subtract

Using one of the expressions shown on page 31 for overflow of unsigned addition and subtraction, we can easily implement double-length addition and subtraction without accessing the machine's carry bit. To illustrate with double-length addition, let the operands be (x_1, x_0) and (y_1, y_0), and the result be (z_1, z_0). Subscript 1 denotes the most significant half, and subscript 0 the least significant. We assume that all 32 bits of the registers are used. The less significant words are unsigned quantities.

$$z_0 \leftarrow x_0 + y_0$$
$$c \leftarrow [(x_0 \mathbin{\&} y_0) \mid ((x_0 \mid y_0) \mathbin{\&} \neg z_0)] \overset{u}{\gg} 31$$
$$z_1 \leftarrow x_1 + y_1 + c$$

This executes in nine instructions. The second line can be $c \leftarrow (z_0 \overset{u}{<} x_0)$, permitting a four-instruction solution on machines that have this comparison operator in a form that gives the result as a **1** or **0** in a register, such as the "SLTU" (***Set on Less Than Unsigned***) instruction on MIPS [MIPS].

Similar code for double-length subtraction ($x - y$) is

$$z_0 \leftarrow x_0 - y_0$$
$$b \leftarrow [(\neg x_0 \mathbin{\&} y_0) \mid ((x_0 \equiv y_0) \mathbin{\&} z_0)] \overset{u}{\gg} 31$$
$$z_1 \leftarrow x_1 - y_1 - b$$

This executes in eight instructions on a machine that has a full set of logical instructions. The second line can be $b \leftarrow (x_0 \overset{u}{<} y_0)$, permitting a four-instruction solution on machines that have the "SLTU" instruction.

Double-length addition and subtraction can be done in five instructions on most machines by representing the multiple-length data using only 31 bits of the least significant words, with the high-order bit being 0 except momentarily when it contains a carry or borrow bit.

2–17 Double-Length Shifts

Let (x_1, x_0) be a pair of 32-bit words to be shifted left or right as if they were a single 64-bit quantity, with x_1 being the most significant half. Let (y_1, y_0) be the result, interpreted similarly. Assume the shift amount n is a variable ranging from 0 to 63. Assume further that the machine's shift instructions are modulo 64 or greater. That is, a shift amount in the range 32 to 63 or –32 to –1 results in an all-0 word, unless the shift is a signed right shift, in which case the result is 32 sign bits from the word shifted. (This code will not work on the Intel x86 machines, which have mod-32 shifts.)

Under these assumptions, the ***shift left double*** operation can be accomplished as follows (eight instructions):

$$y_1 \leftarrow x_1 \ll n \mid x_0 \overset{u}{\gg} (32 - n) \mid x_0 \ll (n - 32)$$

$$y_0 \leftarrow x_0 \ll n$$

The main connective in the first assignment must be ***or***, not ***plus***, to give the correct result when $n = 32$. If it is known that $0 \le n \le 32$, the last term of the first assignment can be omitted, giving a six-instruction solution.

Similarly, a ***shift right double unsigned*** operation can be done with

$$y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32 - n) \mid x_1 \overset{u}{\gg} (n - 32)$$

$$y_1 \leftarrow x_1 \overset{u}{\gg} n$$

Shift right double signed is more difficult, because of an unwanted sign propagation in one of the terms. Straightforward code follows:

$$\text{if } n < 32 \text{ then } y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32 - n)$$

$$\text{else } y_0 \leftarrow x_1 \overset{s}{\gg} (n - 32)$$

$$y_1 \leftarrow x_1 \overset{s}{\gg} n$$

If your machine has the ***conditional move*** instructions, it is a simple matter to express this in branch-free code, in which form it takes eight instructions. If the conditional move instructions are not available, the operation can be done in ten instructions by using the familiar device of constructing a mask with the ***shift right signed 31*** instruction to mask the unwanted sign propagating term:

$$y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32 - n) \mid [(x_1 \overset{s}{\gg} (n - 32)) \,\&\, ((32 - n) \overset{s}{\gg} 31)]$$

$$y_1 \leftarrow x_1 \overset{s}{\gg} n$$

2–18 Multibyte *Add, Subtract, Absolute Value*

Some applications deal with arrays of short integers (usually bytes or halfwords), and often execution is faster if they are operated on a word at a time. For definiteness, the examples here deal with the case of four 1-byte integers packed into a word, but the techniques are easily adapted to other packings, such as a word containing a 12-bit integer and two 10-bit integers, and so on. These techniques are of greater value on 64-bit machines, because more work is done in parallel.

Addition must be done in a way that blocks the carries from one byte into another. This can be accomplished by the following two-step method:

1. Mask out the high-order bit of each byte of each operand and ***add*** (there will then be no carries across byte boundaries).
2. Fix up the high-order bit of each byte with a 1-bit ***add*** of the two operands and the carry into that bit.

The carry into the high-order bit of each byte is given by the high-order bit of each byte of the sum computed in step 1. The subsequent similar method works for subtraction:

Addition

$$s \leftarrow (x \,\&\, \mathtt{0x7F7F7F7F}) + (y \,\&\, \mathtt{0x7F7F7F7F})$$

$$s \leftarrow ((x \oplus y) \,\&\, \mathtt{0x80808080}) \oplus s$$

Subtraction

$$d \leftarrow (x \mid \mathtt{0x80808080}) - (y \,\&\, \mathtt{0x7F7F7F7F})$$

$$d \leftarrow ((x \oplus y) \mid \mathtt{0x7F7F7F7F}) \equiv d$$

These execute in eight instructions, counting the load of **0x7F7F7F7F**, on a machine that has a full set of logical instructions. (Change the ***and*** and ***or*** of **0x80808080** to ***and not*** and ***or not***, respectively, of **0x7F7F7F7F**.)

There is a different technique for the case in which the word is divided into only two fields. In this case, addition can be done by means of a 32-bit addition followed by subtracting out the unwanted carry. On page 30 we noted that the expression (***x*** + ***y***) ***x*** ***y*** gives the carries into each position. Using this and similar observations about subtraction gives the following code for adding/subtracting two halfwords modulo 2^{16} (seven instructions):

Addition	Subtraction
$s \leftarrow x + y$	$d \leftarrow x - y$
$c \leftarrow (s \oplus x \oplus y) \,\&\, \mathtt{0x00010000}$	$b \leftarrow (d \oplus x \oplus y) \,\&\, \mathtt{0x00010000}$
$s \leftarrow s - c$	$d \leftarrow d + b$

Multibyte ***absolute value*** is easily done by complementing and adding 1 to each byte that contains a negative integer (that is, has its high-order bit on). The following code sets each byte of ***y*** equal to the absolute value of each byte of ***x*** (eight instructions):

```
$a \leftarrow x \,\&\, \mathtt{0x80808080}$          // Isolate signs.
$b \leftarrow a \overset{u}{\gg} 7$                  // Integer 1 where $x$ is negative.
$m \leftarrow (a - b) \mid a$                        // 0xFF where $x$ is negative.
$y \leftarrow (x \oplus m) + b$                      // Complement and add 1 where negative.
```

The third line could as well be ***m*** ← ***a*** + ***a*** − ***b***. The addition of ***b*** in the fourth line cannot carry across byte boundaries, because the quantity ***x*** ***m*** has a high-order 0 in each byte.

2–19 Doz, Max, Min

The “doz” function is “difference or zero,” defined as follows:

Signed

$$\operatorname{doz}(x, y) = \begin{cases} x - y, & x \geq y, \\ 0, & x < y. \end{cases}$$

Unsigned

$$\operatorname{dozu}(x, y) = \begin{cases} x - y, & x \overset{u}{\geq} y, \\ 0, & x \overset{u}{<} y. \end{cases}$$

It has been called "first grade subtraction" because the result is 0 if you try to take away too much.[3] If implemented as a computer instruction, perhaps its most important use is to implement the max(***x, y***) and min(***x, y***) functions (in both signed and unsigned forms) in just two simple instructions, as will be seen. Implementing max(***x, y***) and min(***x, y***) in hardware is difficult because the machine would need paths from the output ports of the register file back to an input port, bypassing the adder. These paths are not normally present. If supplied, they would be in a region that's often crowded with wiring for register bypasses. The situation is illustrated in Figure 2–3. The adder is used (by the instruction) to do the subtraction ***x*** **−** ***y***. The high-order bits of the result of the subtraction (sign bit and carries, as described on page 27) define whether ***x*** **≥** ***y*** or ***x*** < ***y*** The comparison result is fed to a multiplexor (MUX) that selects either ***x*** or ***y*** as the result to write into the target register. These paths, from register file outputs ***x*** and ***y*** to the multiplexor, are not normally present and would have little use. The ***difference or zero*** instructions can be implemented without these paths because it is the output of the adder (or 0) that is fed back to the register file.

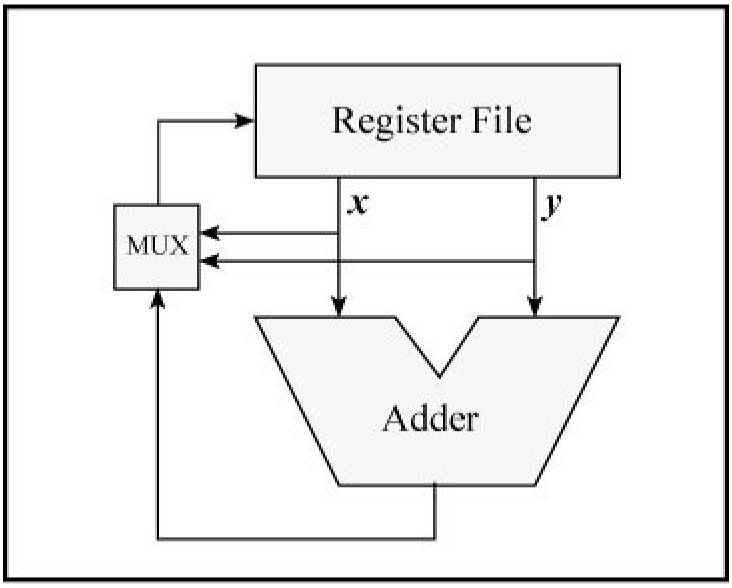


FIGURE 2–3. Implementing max(*x, y*) and min(*x, y*).

Using ***difference or zero***, max(***x, y***) and min(***x, y***) can be implemented in two instructions as follows:

$$\begin{array}{cc} \text{Signed} & \text{Unsigned} \\ \max(x, y) = y + \text{doz}(x, y) & \text{maxu}(x, y) = y + \text{dozu}(x, y) \\ \min(x, y) = x - \text{doz}(x, y) & \text{minu}(x, y) = x - \text{dozu}(x, y) \end{array}$$

In the signed case, the result of the ***difference or zero*** instruction can be negative. This happens if overflow occurs in the subtraction. Overflow should be ignored; the addition of y or subtraction from x will overflow again, and the result will be correct. When doz(x, y) is negative, it is actually the correct difference if it is interpreted as an unsigned integer.

Suppose your computer does not have the ***difference or zero*** instructions, but you want to code doz(x, y), max(x, y), and so forth, in an efficient branch-free way. In the next few paragraphs we show how these functions might be coded if your machine has the ***conditional move*** instructions, comparison predicates, efficient access to the carry bit, or none of these.

If your machine has the ***conditional move*** instructions, it can get doz(x, y) in three instructions, and destructive[4] max(x, y) and min(x, y) in two instructions. For example, on the full RISC, $z \leftarrow$ doz(x, y) can be calculated as follows (r0 is a permanent zero register):

```
sub    z,x,y        Set z = x - y.
cmplt  t,x,y        Set t = 1 if x < y, else 0.
movne  z,t,r0       Set z = 0 if x < y.
```

Also on the full RISC, $x \leftarrow$ max(x, y) can be calculated as follows:

```
cmplt   t,x,y       Set t = 1 if x < y, else 0.
movne   x,t,y       Set x = y if x < y.
```

The min function, and the unsigned counterparts, are obtained by changing the comparison conditions.

These functions can be computed in four or five instructions using comparison predicates (three or four if the comparison predicates give a result of –1 for "true"):

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

On some machines, the carry bit may be a useful aid to computing the unsigned versions of these functions. Let carry($x - y$) denote the bit that comes out of the adder for the operation $x + \bar{y} + 1$, moved to a GPR. Thus, carry($x - y$) = 1 iff $x \geq y$. Then we have

$$\text{dozu}(x, y) = ((x - y) \& \neg(\text{carry}(x - y) - 1))$$

$$\text{maxu}(x, y) = x - ((x - y) \& (\text{carry}(x - y) - 1))$$

$$\text{minu}(x, y) = y + ((x - y) \& (\text{carry}(x - y) - 1))$$

On most machines that have a ***subtract*** that generates a carry or borrow, and another form of ***subtract*** that uses that carry or borrow as an input, the expression carry(***x* − *y***) − **1** can be computed in one more instruction after the subtraction of ***y*** from ***x***. For example, on the Intel x86 machines, minu(***x, y***) can be computed in four instructions as follows:

```
sub eax,ecx    ; Inputs x and y are in eax and ecx resp.
sbb edx,edx    ; edx = 0 if x >= y, else -1.
and eax,edx    ; 0 if x >= y, else x - y.
add eax,ecx    ; Add y, giving y if x >= y, else x.
```

In this way, all three of the functions can be computed in four instructions (three instructions for dozu(***x, y***) if the machine has ***and with complement***).

A method that applies to nearly any RISC is to use one of the above expressions that employ a comparison predicate, and to substitute for the predicate one of the expressions given on page 23. For example:

$$d \leftarrow x - y$$

$$\text{doz}(x, y) = d \,\&\, [(d \equiv ((x \oplus y) \,\&\, (d \oplus x))) \overset{s}{\gg} 31]$$

$$\text{dozu}(x, y) = d \,\&\, \neg[((\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, d)) \overset{s}{\gg} 31]$$

These require from seven to ten instructions, depending on the computer's instruction set, plus one more to get max or min.

These operations can be done in four branch-free basic RISC instructions if it is known that $-2^{31} \le x - y \le 2^{31} - 1$ (that is an expression in ordinary arithmetic, not computer arithmetic). The same code works for both signed and unsigned integers, with the same restriction on ***x*** and ***y***. A sufficient condition for these formulas to be valid is that, for signed integers, $-2^{30} \le x, y \le 2^{30} - 1$, and for unsigned integers, $0 \le x,y \le 2^{31} - 1$.

$$\text{doz}(x, y) = \text{dozu}(x, y) = (x - y) \,\&\, \neg((x - y) \overset{s}{\gg} 31)$$

$$\max(x, y) = \text{maxu}(x, y) = x - ((x - y) \,\&\, ((x - y) \overset{s}{\gg} 31))$$

$$\min(x, y) = \text{minu}(x, y) = y + ((x - y) \,\&\, ((x - y) \overset{s}{\gg} 31))$$

Some uses of the ***difference or zero*** instruction are given here. In these, the result of doz(***x, y***) must be interpreted as an unsigned integer.

1. It directly implements the Fortran IDIM function.
2. To compute the absolute value of a difference [Knu7]:

$$\begin{aligned}|x-y| &= \text{doz}(x,y)+\text{doz}(y,x), && \text{signed arguments,}\\ &= \text{dozu}(x,y)+\text{dozu}(y,x), && \text{unsigned arguments.}\end{aligned}$$

Corollary: $|x|$ = doz(x, **0**) + doz(**0**, x) (other three-instruction solutions are given on page 18).

3. To clamp the upper limit of the true sum of unsigned integers x and y to the maximum positive number ($2^{32} - 1$) [Knu7]:

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

4. Some comparison predicates (four instructions each):

$$\begin{aligned}x > y &= (\text{doz}(x,y) \mid -\text{doz}(x,y)) \overset{u}{\gg} 31,\\ x \overset{u}{>} y &= (\text{dozu}(x,y) \mid -\text{dozu}(x,y)) \overset{u}{\gg} 31.\end{aligned}$$

5. The carry bit from the addition $x + y$ (five instructions):

$$\text{carry}(x+y) = x \overset{u}{>} \neg y = (\text{dozu}(x,\neg y) \mid -\text{dozu}(x,\neg y)) \overset{u}{\gg} 31.$$

The expression doz(x, $-y$), with the result interpreted as an unsigned integer, is in most cases the true sum $x + y$ with the lower limit clamped at 0. However, it fails if y is the maximum negative number.

The IBM RS/6000 computer, and its predecessor the 801, have the signed version of ***difference or zero***. Knuth's MMIX computer [Knu7] has the unsigned version (including some varieties that operate on parts of words in parallel). This raises the question of how to get the signed version from the unsigned version, and vice versa. This can be done as follows (where the additions and subtractions simply complement the sign bit):

$$\begin{aligned}\text{doz}(x,y) &= \text{dozu}(x + 2^{31}, y + 2^{31}),\\ \text{dozu}(x,y) &= \text{doz}(x - 2^{31}, y - 2^{31}).\end{aligned}$$

Some other identities that may be useful are:

$$\begin{aligned}\text{doz}(\neg x, \neg y) &= \text{doz}(y, x),\\ \text{dozu}(\neg x, \neg y) &= \text{dozu}(y, x).\end{aligned}$$

The relation doz($-x$, $-y$) = doz(y, x) fails if either x or y, but not both, is the maximum negative number.

2–20 Exchanging Registers

A very old trick is exchanging the contents of two registers without using a third [IBM]:

$$\begin{aligned}x &\leftarrow x \quad y\\ y &\leftarrow y \quad x\\ x &\leftarrow x \quad y\end{aligned}$$

This works well on a two-address machine. The trick also works if   is replaced by the ≡ logical operation (complement of ***exclusive or***) and can be made to work in various ways with ***add***'s and ***subtract***'s:

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

Unfortunately, each of these has an instruction that is unsuitable for a two-address machine, unless the machine has "reverse subtract."

This little trick can actually be useful in the application of double buffering, in which two pointers are swapped. The first instruction can be factored out of the loop in which the swap is done (although this negates the advantage of saving a register):

$$\begin{array}{rl} \text{Outside the loop:} & t \leftarrow x \oplus y \\ \text{Inside the loop:} & x \leftarrow x \oplus t \\ & y \leftarrow y \oplus t \end{array}$$

Exchanging Corresponding Fields of Registers

The problem here is to exchange the contents of two registers ***x*** and ***y*** wherever a mask bit $m_i = 1$, and to leave ***x*** and ***y*** unaltered wherever $m_i = 0$. By "corresponding" fields, we mean that no shifting is required. The 1-bits of ***m*** need not be contiguous. The straightforward method is as follows:

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

By using "temporaries" for the four ***and*** expressions, this can be seen to require seven instructions, assuming that either ***m*** or $\bar{m}$ can be loaded with a single instruction and the machine has ***and not*** as a single instruction. If the machine is capable of executing the four (independent) ***and*** expressions in parallel, the execution time is only three cycles.

A method that is probably better (five instructions, but four cycles on a machine with unlimited instruction-level parallelism) is shown in column (a) below. It is suggested by the "three ***exclusive or***" code for exchanging registers.

$$\begin{array}{lll} \text{(a)} & \text{(b)} & \text{(c)} \\ x \leftarrow x \oplus y & x \leftarrow x \equiv y & t \leftarrow (x \oplus y) \,\&\, m \\ y \leftarrow y \oplus (x \,\&\, m) & y \leftarrow y \equiv (x \mid \bar{m}) & x \leftarrow x \oplus t \\ x \leftarrow x \oplus y & x \leftarrow x \equiv y & y \leftarrow y \oplus t \end{array}$$

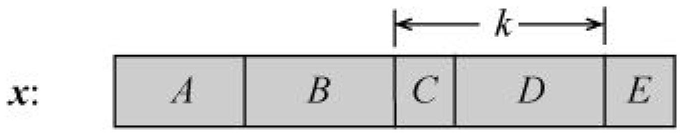
The steps in column (b) do the same exchange as that of column (a), but column (b) is useful if ***m*** does not fit in an immediate field, but $\bar{m}$ does, and the machine has the

equivalence instruction.

Still another method is shown in column (c) above [GLS1]. It also takes five instructions (again assuming one instruction must be used to load ***m*** into a register), but executes in only three cycles on a machine with sufficient instruction-level parallelism.

Exchanging Two Fields of the Same Register

Assume a register ***x*** has two fields (of the same length) that are to be swapped, without altering other bits in the register. That is, the object is to swap fields ***B*** and ***D*** without altering fields ***A, C***, and ***E***, in the computer word illustrated below. The fields are separated by a shift distance ***k***.



Straightforward code would shift ***D*** and ***B*** to their new positions, and combine the words with ***and*** and ***or*** operations, as follows:

$$\begin{aligned} t_1 &= (x \mathbin{\&} m) \ll k \\ t_2 &= (x \overset{u}{\gg} k) \mathbin{\&} m \\ x' &= (x \mathbin{\&} m') \mid t_1 \mid t_2 \end{aligned}$$

Here, ***m*** is a mask with 1's in field ***D*** (and 0's elsewhere), and ***m′*** is a mask with 1′s in fields ***A, C***, and ***E***. This code requires 11 instructions and six cycles on a machine with unlimited instruction-level parallelism, allowing for four instructions to generate the two masks.

A method that requires only eight instructions and executes in five cycles, under the same assumptions, is shown below [GLS1]. It is similar to the code in column (c) on page 46 for interchanging corresponding fields of two registers. Again, ***m*** is a mask that isolates field ***D***.

$$\begin{aligned} t_1 &= [x \oplus (x \overset{u}{\gg} k)] \mathbin{\&} m \\ t_2 &= t_1 \ll k \\ x' &= x \oplus t_1 \oplus t_2 \end{aligned}$$

The idea is that ***t_1*** contains ***B***   ***D*** in position ***D*** (and 0's elsewhere), and ***t_2*** contains ***B***   ***D*** in position ***B***. This code, and the straightforward code given earlier, work correctly if ***B*** and ***D*** are "split fields"—that is, if the 1-bits of mask ***m*** are not contiguous.

Conditional Exchange

The exchange methods of the preceding two sections, which are based on ***exclusive or***, degenerate into no-operations if the mask ***m*** is 0. Hence, they can perform an exchange of entire registers, or of corresponding fields of two registers, or of two fields of the same register, if ***m*** is set to all 1’s if some condition ***c*** is **true**, and to all 0’s if ***c*** is **false**. This gives branch-free code if ***m*** can be set up without branching.

2–21 Alternating among Two or More Values

Suppose a variable ***x*** can have only two possible values ***a*** and ***b***, and you wish to assign to ***x*** the value other than its current one, and you wish your code to be independent of the values of ***a*** and ***b***. For example, in a compiler ***x*** might be an opcode that is known to be either ***branch true*** or ***branch false***, and whichever it is, you want to switch it to the other. The values of the opcodes ***branch true*** and ***branch false*** are arbitrary, probably defined by a C `#define` or `enum` declaration in a header file.

The straightforward code to do the switch is

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

or, as is often seen in C programs,

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

A far better (or at least more efficient) way to code it is either

$$x \leftarrow a + b - x, \quad \text{or}$$
$$x \leftarrow a \oplus b \oplus x.$$

If ***a*** and ***b*** are constants, these require only one or two basic RISC instructions. Of course, overflow in calculating ***a*** + ***b*** can be ignored.

This raises the question: Is there some particularly efficient way to cycle among three or more values? That is, given three arbitrary but distinct constants ***a, b***, and ***c***, we seek an easy-to-evaluate function ***f*** that satisfies

$$\begin{aligned} f(a) &= b, \\ f(b) &= c, \quad \text{and} \\ f(c) &= a. \end{aligned}$$

It is perhaps interesting to note that there is always a polynomial for such a function. For the case of three constants,

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

(The idea is that if ***x*** = ***a***, the first and last terms vanish, and the middle term simplifies to ***b***, and so on.) This requires 14 arithmetic operations to evaluate, and for arbitrary ***a, b***, and ***c***, the intermediate results exceed the computer’s word size. But it is just a quadratic; if written in the usual form for a polynomial and evaluated using Horner’s rule,[5] it would require only five arithmetic operations (four for a quadratic with integer coefficients, plus one for a final division). Rearranging Equation (5) accordingly gives

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

This is getting too complicated to be interesting, or practical.

Another method, similar to Equation (5) in that just one of the three terms survives, is

$$f(x) = ((-(x = c)) \mathbin{\&} a) + ((-(x = a)) \mathbin{\&} b) + ((-(x = b)) \mathbin{\&} c).$$

This takes 11 instructions if the machine has the ***equal*** predicate, not counting loads of constants. Because the two addition operations are combining two 0 values with a nonzero, they can be replaced with ***or*** or ***exclusive or*** operations.

The formula can be simplified by precalculating $a - c$ and $b - c$, and then using [GLS1]:

$$f(x) = ((-(x = c)) \mathbin{\&} (a - c)) + ((-(x = a)) \mathbin{\&} (b - c)) + c, \text{ or}$$

$$f(x) = ((-(x = c)) \mathbin{\&} (a \quad c)) \quad ((-(x = a)) \mathbin{\&} (b \quad c)) \quad c.$$

Each of these operations takes eight instructions, but on most machines these are probably no better than the straightforward C code shown below, which executes in four to six instructions for small `a`, `b`, and `c`.

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

Pursuing this matter, there is an ingenious branch-free method of cycling among three values on machines that do not have comparison predicate instructions [GLS1]. It executes in eight instructions on most machines.

Because a, b, and c are distinct, there are two bit positions, n_1 and n_2, where the bits of a, b, and c are not all the same, and where the "odd one out" (the one whose bit differs in that position from the other two) is different in positions n_1 and n_2. This is illustrated below for the values 21, 31, and 20, shown in binary.

$$\begin{array}{cccccc} 1 & 0 & 1 & 0 & 1 & c \\ 1 & 1 & 1 & 1 & 1 & a \\ 1 & 0 & 1 & 0 & 0 & b \\ & n_1 & & & n_2 & \end{array}$$

Without loss of generality, rename a, b, and c so that a has the odd one out in position n_1 and b has the odd one out in position n_2, as shown above. Then there are two possibilities for the values of the bits at position n_1, namely $(a_{n_1}, b_{n_1}, c_{n_1}) = (0, 1, 1)$ or $(1, 0, 0)$. Similarly, there are two possibilities for the bits at position n_2, namely $(a_n\ , b_n\ , c_n\ ) = (0, 1, 0)$ or $(1, 0, 1)$. This makes four cases in all, and

2 2 2
formulas for each of these cases are shown below.

Case 1. $(a_{n_1}, b_{n_1}, c_{n_1}) = (0, 1, 1), (a_{n_2}, b_{n_2}, c_{n_2}) = (0, 1, 0)$:

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

Case 2. $(a_{n_1}, b_{n_1}, c_{n_1}) = (0, 1, 1), (a_{n_2}, b_{n_2}, c_{n_2}) = (1, 0, 1)$:

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

Case 3. $(a_{n_1}, b_{n_1}, c_{n_1}) = (1, 0, 0), (a_{n_2}, b_{n_2}, c_{n_2}) = (0, 1, 0)$:

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

Case 4. $(a_{n_1}, b_{n_1}, c_{n_1}) = (1, 0, 0), (a_{n_2}, b_{n_2}, c_{n_2}) = (1, 0, 1)$:

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

In these formulas, the left operand of each multiplication is a single bit. A multiplication by 0 or 1 can be converted into an ***and*** with a value of 0 or all 1's. Thus, the formulas can be rewritten as illustrated below for the first formula.

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

Because all variables except x are constants, this can be evaluated in eight instructions on the basic RISC. Here again, the additions and subtractions can be replaced with ***exclusive or***.

This idea can be extended to cycling among four or more constants. The essence of the idea is to find bit positions n_1, n_2, ..., at which the bits uniquely identify the constants. For four constants, three bit positions always suffice. Then (for four constants) solve the following equation for s, t, u, and v (that is, solve the system of four linear equations in which $f(x)$ is a, b, c, or d, and the coefficients x_{n_i} are 0 or 1):

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

If the four constants are uniquely identified by only two bit positions, the equation to solve is

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

2–22 A Boolean Decomposition Formula

In this section, we have a look at the minimum number of binary Boolean operations, or instructions, that suffice to implement any Boolean function of three, four, or five variables. By a “Boolean function” we mean a Boolean-valued function of Boolean arguments.

Our notation for Boolean algebra uses “+” for ***or***, juxtaposition for ***and***,   for ***exclusive or***, and either an overbar or a prefix ¬ for ***not***. These operators can be applied to single-bit operands or “bitwise” to computer words. Our main result is the following theorem:

THEOREM. ***If f(x, y, z) is a Boolean function of three variables, then it can be decomposed into the form g(x, y)   zh(x, y), where g and h are Boolean functions of two variables.***[6]

Proof [Ditlow]. ***f***(***x, y, z***) can be expressed as a sum of minterms, and then $\bar{z}$ and ***z*** can be factored out of their terms, giving

$$f(x, y, z) = \bar{z}f_0(x, y) + zf_1(x, y).$$

Because the operands to “+” cannot both be 1, the ***or*** can be replaced with ***exclusive or***, giving

$$\begin{aligned} f(x, y, z) &= \bar{z}f_0(x, y) \oplus zf_1(x, y) \\ &= (1 \oplus z)f_0(x, y) \oplus zf_1(x, y) \\ &= f_0(x, y) \oplus zf_0(x, y) \oplus zf_1(x, y) \\ &= f_0(x, y) \oplus z(f_0(x, y) \oplus f_1(x, y)), \end{aligned}$$

where we have twice used the identity (***a***   ***b***) ***c*** = ***ac***   ***bc***.

This is in the required form with ***g***(***x, y***) = ***f***$_0$(***x, y***) and ***h***(***x, y***) = ***f***$_0$(***x, y***)   ***f***$_1$(***x, y***). ***f***$_0$(***x, y***), incidentally, is ***f***(***x, y, z***) with ***z*** = 0, and ***f***$_1$(***x, y***) is ***f***(***x, y, z***) with ***z*** = 1.

COROLLARY. ***If a computer’s instruction set includes an instruction for each of the 16 Boolean functions of two variables, then any Boolean function of three variables can be implemented with four (or fewer) instructions***.

One instruction implements ***g***(***x, y***), another implements ***h***(***x, y***), and these are combined with ***and*** and ***exclusive or***.

As an example, consider the Boolean function that is 1 if exactly two of ***x, y***, and ***z*** are 1:

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

Before proceeding, the interested reader might like to try to implement ***f*** with four instructions, without using the theorem.

From the proof of the theorem,

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

which is four instructions.

Clearly, the theorem can be extended to functions of four or more variables. That is, any Boolean function $f(x_1, x_2, \ldots, x_n)$ can be decomposed into the form $g(x_1, x_2, \ldots, x_{n-1}) \quad x_n h(x_1, x_2, \ldots, x_{n-1})$. Thus, a function of four variables can be decomposed as follows:

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

This shows that a computer that has an instruction for each of the 16 binary Boolean functions can implement any function of four variables with ten instructions. Similarly, any function of five variables can be implemented with 22 instructions.

However, it is possible to do much better. For functions of four or more variables there is probably no simple plug-in equation like the theorem gives, but exhaustive computer searches have been done. The results are that any Boolean function of four variables can be implemented with seven binary Boolean instructions, and any such function of five variables can be implemented with 12 such instructions [Knu4, 7.1.2].

In the case of five variables, only 1920 of the 2^{25} = 4,294,967,296 functions require 12 instructions, and these 1920 functions are all essentially the same function. The variations are obtained by permuting the arguments, replacing some arguments with their complements, or complementing the value of the function.

2–23 Implementing Instructions for All 16 Binary Boolean Operations

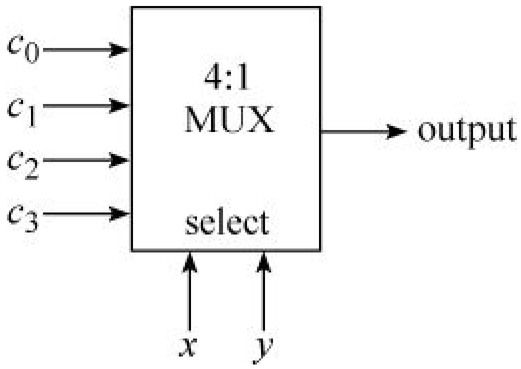
The instruction sets of some computers include all 16 binary Boolean operations. Many of the instructions are useless in that their function can be accomplished with another instruction. For example, the function $f(x, y) = 0$ simply clears a register, and most computers have a variety of ways to do that. Nevertheless, one reason a computer designer might choose to implement all 16 is that there is a simple and quite regular circuit for doing it.

Refer to Table 2–1 on page 17, which shows all 16 binary Boolean functions. To implement these functions as instructions, choose four of the opcode bits to be the same as the function values shown in the table. Denoting these opcode bits by c_0, c_1, c_2, and c_3, reading from the bottom up in the table, and the input registers by x and y, the circuit for implementing all 16 binary Boolean operations is described by the logic expression

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

For example, with $c_0 = c_1 = c_2 = c_3 = 0$, the instruction computes the zero function, $f(x, y) = 0$. With $c_0 = 1$ and the other opcode bits 0 it is the ***and*** instruction. With $c_0 = c_3 = 0$ and $c_1 = c_2 = 1$ it is ***exclusive or***, and so forth.

This can be implemented with n 4:1 MUXs, where n is the word size of the machine. The data bits of x and y are the select lines, and the four opcode bits are the data inputs to each MUX. The MUX is a standard building block in today's technology, and it is usually a very fast circuit. It is illustrated below.



The function of the circuit is to select c_0, c_1, c_2, or c_3 to be the output, depending on whether x and y are 00, 01, 10, or 11, respectively. It is like a four-position rotary switch.

Elegant as this is, it is somewhat expensive in opcode points, using 16 of them. There are a number of ways to implement all 16 Boolean operations using only eight opcode points, at the expense of less regular logic. One such scheme is illustrated in Table 2–3.

TABLE 2–3. EIGHT SUFFICIENT BOOLEAN INSTRUCTIONS

Function Values	Formula	Instruction Mnemonic (Name)
0001	xy	*and*
0010	$x\bar{y}$	*andc* (*and with complement*)
0110	$x \oplus y$	*xor* (*exclusive or*)
0111	$x + y$	*or*
1110	$\overline{xy}$	*nand* (*negative and*)
1101	$\overline{x\bar{y}}$, or $\bar{x} + y$	*cor* (*complement and or*)
1001	$\overline{x \oplus y}$, or $x \equiv y$	*eqv* (*equivalence*)
1000	$\overline{x + y}$	*nor* (*negative or*)

The eight operations not shown in the table can be done with the eight instructions shown, by interchanging the inputs or by having both register fields of the instruction refer to the same register. See exercise 13.

IBM's POWER architecture uses this scheme, with the minor difference that POWER has ***or with complement*** rather than ***complement and or***. The scheme shown in Table 2–3 allows the last four instructions to be implemented by complementing the result of the first four instructions, respectively.

Historical Notes

The algebra of logic expounded in George Boole's ***An Investigation of the Laws of Thought*** (1854)[7] is somewhat different from what we know today as "Boolean algebra." Boole used the ***integers*** 1 and 0 to represent truth and falsity, respectively, and he showed how they could be manipulated with the methods of ordinary numerical algebra to formalize natural language statements involving "and," "or," and "except." He also used ordinary algebra to formalize statements in set theory involving intersection, union of disjoint sets, and complementation. He also formalized statements in probability theory, in which the variables take on real number values from 0 to 1. The work often deals with questions of philosophy, religion, and law.

Boole is regarded as a great thinker about logic because he formalized it, allowing complex statements to be manipulated mechanically and flawlessly with the familiar methods of ordinary algebra.

Skipping ahead in history, there are a few programming languages that include all 16 Boolean operations. IBM's PL/I (ca. 1966) includes a built-in function named BOOL. In BOOL(x, y, z), z is a bit string of length four (or converted to that if necessary), and x and y are bit strings of equal length (or converted to that if necessary). Argument z specifies the Boolean operation to be performed on x and y. Binary 0000 is the zero function, 0001 is xy, 0010 is $x\bar{y}$, and so forth.

Another such language is Basic for the Wang System 2200B computer (ca. 1974), which provides a version of BOOL that operates on character strings rather than on bit strings or integers [Neum].

Still another such language is MIT PDP-6 Lisp, later called MacLisp [GLS1].

Exercises

1. David de Kloet suggests the following code for the snoob function, for $x \neq 0$, where the final assignment to y is the result:

```
$y \leftarrow x + (x \,\&\, -x)$
$x \leftarrow x \,\&\, \neg y$
while$((x \,\&\, 1) = 0)$ $x \leftarrow x \overset{s}{\gg} 1$
$x \leftarrow x \overset{s}{\gg} 1$
$y \leftarrow y \mid x$
```

This is essentially the same as Gosper's code (page 15), except the right shift is done with a ***while***-loop rather than with a ***divide*** instruction. Because division is usually costly in time, this might be competitive with Gosper's code if the while-loop is not executed too many times. Let n be the length of the bit strings x and y, k the number of 1-bits in the strings, and assume the code is executed for all values of x that have exactly k 1-bits. Then for each invocation of the function, how many times, on average, will the body of the ***while***-loop be executed?

2. The text mentions that a left shift by a variable amount is not right-to-left computable. Consider the function $x \ll (x \,\&\, 1)$ [Knu8]. This is a left shift by a variable amount, but it can be computed by

$$x + (x \,\&\, 1) * x, \text{ or}$$
$$x + (x \,\&\, (-(x \,\&\, 1))),$$

which are all right-to-left computable operations. What is going on here? Can you think of another such function?

3. Derive Dietz's formula for the average of two unsigned integers,

$$(x \,\&\, y) + ((x \oplus y) \overset{u}{\gg} 1).$$

4. Give an overflow-free method for computing the average of four unsigned integers, $(a + b + c + d)/4$.

5. Many of the comparison predicates shown on page 23 can be simplified substantially if bit 31 of either ***x*** or ***y*** is known. Show how the seven-instruction expression for $x \overset{u}{\le} y$ can be simplified to three basic RISC, non-comparison, instructions if $y_{31} = 0$.

6. Show that if two numbers, possibly distinct, are added with "end-around carry," the addition of the carry bit cannot generate another carry out of the high-order position.

7. Show how end-around carry can be used to do addition if negative numbers are represented in one's-complement notation. What is the maximum number of bit positions that a carry (from any bit position) might be propagated through?

8. Show that the MUX operation, $(x \,\&\, m) \mid (y \,\&\, {\sim}m)$, can be done in three instructions on the basic RISC (which does not have the ***and with complement*** instruction).

9. Show how to implement ***x***   ***y*** in four instructions with ***and-or-not*** logic.

10. Given a 32-bit word ***x*** and two integer variables ***i*** and ***j*** (in registers), show code to copy the bit of ***x*** at position ***i*** to position ***j***. The values of ***i*** and ***j*** have no relation, but assume that $0 \le i, j \le 31$.

11. How many binary Boolean instructions are sufficient to evaluate any ***n***-variable Boolean function if it is decomposed recursively by the method of the theorem?

12. Show that alternative decompositions of Boolean functions of three variables are

(a) $f(x, y, z) = g(x, y) \quad \bar{z}h(x, y)$ (the "negative Davio decomposition"), and
(b) $f(x, y, z) = g(x, y) \quad (z + h(x, y))$.

13. It is mentioned in the text that all 16 binary Boolean operations can be done with the eight instructions shown in Table 2–3, by interchanging the inputs or by having both register fields of the instruction refer to the same register. Show how to do this.

14. Suppose you are not concerned about the six Boolean functions that are really constants or unary functions, namely $f(x, y) = 0, 1, x, y, \bar{x}$, and $\bar{y}$, but you want your instruction set to compute the other ten functions with one instruction. Can this be done with fewer than eight binary Boolean instruction types (opcodes)?

15. Exercise 13 shows that eight instruction types suffice to compute any of the 16 two-operand Boolean operations with one R-R (register-register) instruction. Show that six instruction types suffice in the case of R-I (register-immediate) instructions. With R-I instructions, the input operands cannot be interchanged or equated, but the second input operand (the immediate field) can be complemented or, in fact, set to any value at no cost in execution time. Assume for simplicity that the immediate fields are the same length as the general purpose registers.

16. Show that not all Boolean functions of three variables can be implemented with three binary logical instructions.

Chapter 3. Power-of-2 Boundaries

3–1 Rounding Up/Down to a Multiple of a Known Power of 2

Rounding an unsigned integer ***x*** down to, for example, the next smaller multiple of 8 is trivial: ***x*** & **−8** does it. An alternative is $(x \overset{u}{\gg} 3) \ll 3$. These work for signed integers as well, provided "round down" means to round in the negative direction (e.g., (**−37**) & (**−8**) = **−40**).

Rounding up is almost as easy. For example, an unsigned integer ***x*** can be rounded up to the next greater multiple of 8 with either of

$$(x + 7)\ \&\ {-8}, \quad \text{or}$$
$$x + (-x\ \&\ 7).$$

These expressions are correct for signed integers as well, provided "round up" means to round in the positive direction. The second term of the second expression is useful if you want to know how much you must add to ***x*** to make it a multiple of 8 [Gold].

To round a signed integer to the nearest multiple of 8 toward 0, you can combine the two expressions above in an obvious way:

$$t \leftarrow (x \overset{s}{\gg} 31)\ \&\ 7;$$
$$(x + t)\ \&\ {-8}$$

An alternative for the first line is $t \leftarrow (x \overset{s}{\gg} 2) \overset{u}{\gg} 29$, which is useful if the machine lacks ***and immediate***, or if the constant is too large for its immediate field.

Sometimes the rounding factor is given as the $\log_2$ of the alignment amount (e.g., a value of 3 means to round to a multiple of 8). In this case, code such as the following can be used, where ***k*** = $\log_2$(alignment amount):

$$\begin{array}{rl} \text{round down:} & x\ \&\ ((-1) \ll k) \\ & (x \overset{u}{\gg} k) \ll k \\ \text{round up:} & t \leftarrow (1 \ll k) - 1; \quad (x + t)\ \&\ \neg t \\ & t \leftarrow (-1) \ll k; \quad (x - t - 1)\ \&\ t \end{array}$$

3–2 Rounding Up/Down to the Next Power of 2

We define two functions that are similar to floor and ceiling, but which are directed roundings to the closest integral power of 2, rather than to the closest integer. Mathematically, they are defined by

$$\text{flp2}(x) = \begin{cases} \text{undefined}, & x < 0, \\ 0, & x = 0, \\ 2^{\lfloor \log_2 x \rfloor}, & \text{otherwise;} \end{cases} \qquad \text{clp2}(x) = \begin{cases} \text{undefined}, & x < 0, \\ 0, & x = 0, \\ 2^{\lceil \log_2 x \rceil}, & \text{otherwise.} \end{cases}$$

The initial letters of the function names are intended to suggest "floor" and "ceiling." Thus, flp2(***x***) is the greatest power of 2 that is **≤*x***, and clp2(***x***) is the least power of 2 that is **≥*x***. These definitions make sense even when ***x*** is not an integer (e.g., flp2(0.1) = 0.0625). The functions satisfy several relations analogous to those involving floor and ceiling, such as those shown below, where ***n*** is an integer.

$$\begin{array}{ll} \lfloor x \rfloor = \lceil x \rceil \text{ iff } x \text{ is an integer} & \text{flp2}(x) = \text{clp2}(x) \text{ iff } x \text{ is a power of 2 or is 0} \\ \lfloor x + n \rfloor = \lfloor x \rfloor + n & \text{flp2}(2^n x) = 2^n \text{flp2}(x) \\ \lceil x \rceil = -\lfloor -x \rfloor & \text{clp2}(x) = 1/\text{flp2}(1/x),\ x \neq 0 \end{array}$$

Computationally, we deal only with the case in which ***x*** is an integer, and we take it to be unsigned, so the functions are well defined for all ***x***. We require the value computed to be the arithmetically correct value modulo 2^{32} (that is, we take clp2(***x***) to be **0** for ***x*** > $\mathbf{2^{31}}$). The functions are tabulated below for a few values of ***x***.

x	flp2(x)	clp2(x)
0	0	0
1	1	1
2	2	2
3	2	4
4	4	4
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

Functions flp2 and clp2 are connected by the relations shown below. These can be used to compute one from the other, subject to the indicated restrictions.

$$\begin{aligned}
\text{clp2}(x) &= 2\,\text{flp2}(x-1), && x \neq 1,\\
&= \text{flp2}(2x-1), && 1 \le x \le 2^{31},\\
\text{flp2}(x) &= \text{clp2}(x \overset{u}{\div} 2 + 1), && x \neq 0,\\
&= \text{clp2}(x+1) \overset{u}{\div} 2, && x < 2^{31}.
\end{aligned}$$

The round-up and round-down functions can be computed quite easily with the ***number of leading zeros*** instruction, as shown below. However, for these relations to hold for $x = 0$ and $x > 2^{31}$, the computer must have its shift instructions defined to produce **0** for shift amounts of –1, 32, and 63. Many machines (e.g., PowerPC) have "mod-64" shifts, which do this. In the case of −1, it is adequate if the machine shifts in the opposite direction (that is, a shift left of –1 becomes a shift right of 1).

$$\begin{aligned}
\text{flp2}(x) &= 1 \ll (31 - \text{nlz}(x))\\
&= 1 \ll (\text{nlz}(x) \oplus 31)\\
&= \text{0x80000000} \overset{u}{\gg} \text{nlz}(x)\\
\text{clp2}(x) &= 1 \ll (32 - \text{nlz}(x-1))\\
&= \text{0x80000000} \overset{u}{\gg} (\text{nlz}(x-1) - 1)
\end{aligned}$$

Rounding Down

Figure 3–1 illustrates a branch-free algorithm that might be useful if ***number of leading zeros*** is not available. This algorithm is based on right-propagating the leftmost 1-bit, and executes in 12 instructions.

```
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);
}
```

FIGURE 3–1. Greatest power of 2 less than or equal to *x*, branch free.

Figure 3–2 shows two simple loops that compute the same function. All variables are unsigned integers. The loop on the right keeps turning off the rightmost 1-bit of ***x*** until ***x* = 0**, and then returns the previous value of ***x***.

```
   y = 0x80000000;              do {
   while (y > x)                   y = x;
      y = y >> 1;                  x = x & (x - 1);
   return y;                    } while(x != 0);
                                return y;
```

FIGURE 3–2. Greatest power of 2 less than or equal to *x*, simple loops.

The loop on the left executes in 4nlz(***x***) + 3 instructions. The loop on the right, for ***x*** ≠ 0, executes in 4 pop(***x***) instructions,[1] if the comparison to 0 is zero-cost.

Rounding Up

The right-propagation trick yields a good algorithm for rounding up to the next power of 2. This algorithm, shown in Figure 3–3, is branch free and runs in 12 instructions.

```
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;
}
```

FIGURE 3–3. Least power of 2 greater than or equal to *x*.

An attempt to compute this with the obvious loop does not work out very well:

```
y = 1;

while (y < x)        // Unsigned comparison.
   y = 2*y;
return y;
```

This code returns 1 for ***x*** = 0, which is probably not what you want, loops forever for ***x*** ≥ 2^{31}, and executes in 4***n*** + 3 instructions, where ***n*** is the power of 2 of the returned integer. Thus, it is slower than the branch-free code, in terms of instructions executed, for ***n***≥ 3 (***x***≥ 8).

3–3 Detecting a Power-of-2 Boundary Crossing

Assume memory is divided into blocks that are a power of 2 in size, starting at address 0. The blocks may be words, doublewords, pages, and so on. Then, given a starting address ***a*** and a length ***l***, we wish to determine whether or not the address range from ***a*** to ***a*** + ***l*** –1, ***l*** ≥ 2, crosses a block boundary. The quantities ***a*** and ***l*** are unsigned and any values that fit in a register are possible.

If ***l*** = 0 or 1, a boundary crossing does not occur, regardless of ***a***. If ***l*** exceeds the block size, a boundary crossing does occur, regardless of ***a***. For very large values of ***l*** (wraparound is possible), a boundary crossing can occur even if the first and last bytes of the address range are in the same block.

There is a surprisingly concise way to detect boundary crossings on the IBM System/370 [CJS]. This method is illustrated below for a block size of 4096 bytes (a common page size).

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

```
                    BO   CROSSES
```

The first instruction forms the logical ***or*** of RA (which contains the starting address ***a***) and the number 0xFFFFF000. The second instruction adds in the length and sets the machine's 2-bit condition code. For the ***add logical*** instruction, the first bit of the condition code is set to 1 if a carry occurred, and the second bit is set to 1 if the 32-bit register result is nonzero. The last instruction branches if both bits are set. At the branch target, RA will contain the length that extends beyond the first page (this is an extra feature that was not asked for).

If, for example, $a = 0$ and $l = 4096$, a carry occurs, but the register result is 0, so the program properly does ***not*** branch to label CROSSES.

Let us see how this method can be adapted to RISC machines, which generally do not have ***branch on carry and register result nonzero***. Using a block size of 8 for notational simplicity, the method of [CJS] branches to CROSSES if a carry occurred ($(a \mid -8) + l \geq 2^{32}$) and the register result is nonzero ($(a \mid -8) + l \neq 2^{32}$). Thus, it is equivalent to the predicate

$$(a \mid -8) + l > 2^{32}.$$

This in turn is equivalent to getting a carry in the final addition in evaluating $((a \mid -8) - 1) + l$. If the machine has ***branch on carry***, this can be used directly, giving a solution in about five instructions, counting a load of the constant −8.

If the machine does not have ***branch on carry***, we can use the fact that carry occurs in $x + y$ iff $\neg x \overset{u}{<} y$ (see "Unsigned Add/Subtract" on page 31) to obtain the expression

$$\neg((a \mid -8) - 1) \overset{u}{<} l.$$

Using various identities such as $\neg(x - 1) = -x$ gives the following equivalent expressions for the "boundary crossed" predicate:

$$-(a \mid -8) \overset{u}{<} l$$

$$\neg(a \mid -8) + 1 \overset{u}{<} l$$

$$(\neg a \,\&\, 7) + 1 \overset{u}{<} l$$

These can be evaluated in five or six instructions on most RISC computers, counting the final conditional branch.

Using another tack, clearly an 8-byte boundary is crossed iff

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

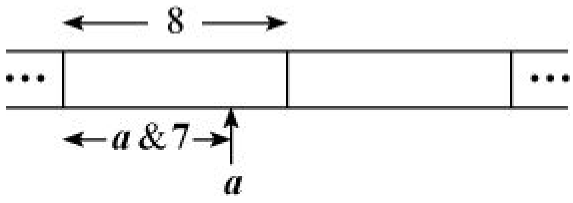
This cannot be directly evaluated because of the possibility of overflow (which occurs if l is very large), but it is easily rearranged to $8 - (a \,\&\, 7) < l$, which can be directly evaluated on the computer (no part of it overflows). This gives the expression

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

which can be evaluated in five instructions on most RISCs (four if it has ***subtract***

from immediate). If a boundary crossing occurs, the length that extends beyond the first block is given by ***l*** − (**8** − (***a*** & **7**)), which can be calculated with one additional instruction (***subtract***).

This formula can be easily understood from the figure below [Kumar], which illustrates that ***a*** & **7** is the offset of ***a*** in its block, and thus **8** − (***a*** & 7) is the space remaining in the block.



Exercises

1. Show how to round an unsigned integer to the nearest multiple of 8, with the halfway case (a) rounding up, (b) rounding down, and (c) rounding up or down, whichever makes the next bit to the left a zero ("unbiased" rounding).

2. Show how to round an unsigned integer to the nearest multiple of 10, with the halfway case (a) rounding up, (b) rounding down, and (c) rounding up or down, whichever results in an even multiple of 10. Feel free to use division, remaindering, and multiplication instructions, and don't be concerned about values very close to the largest unsigned integer.

3. Code a function in C that does an "unaligned load." The function is given an address ***a*** and it loads the four bytes from addresses ***a*** through ***a*** + 3 into a 32-bit GPR, as if those four bytes contained an integer. Parameter ***a*** addresses the low-order byte (that is, the machine is little-endian). The function should be branch free, it should execute at most two load instructions and, if ***a*** is full-word aligned, it must not attempt to load from address ***a*** + 4, because that may be in a read-protected block.

Chapter 4. Arithmetic Bounds

4–1 Checking Bounds of Integers

By "bounds checking" we mean to verify that an integer *x* is within two bounds *a* and *b*—that is, that

$$a \le x \le b.$$

We first assume that all quantities are signed integers.

An important application is the checking of array indexes. For example, suppose a one-dimensional array *A* can be indexed by values from 1 to 10. Then, for a reference *A*(*i*), a compiler might generate code to check that

$$1 \le i \le 10$$

and to *branch* or *trap* if this is not the case. In this section we show that this check can be done with a single comparison, by performing the equivalent check [PL8]:

$$i - 1 \overset{u}{\le} 9.$$

This is probably better code, because it involves only one *compare-branch* (or *compare-trap*), and because the quantity *i*– 1 is probably needed anyway for the array addressing calculations.

Does the implementation

$$a \le x \le b \Rightarrow x - a \overset{u}{\le} b - a$$

always work, even if overflow may occur in the subtractions? It does, provided we somehow know that $a \le b$. In the case of array bounds checking, language rules may require that an array not have a number of elements (or number of elements along any axis) that are 0 or negative, and this rule can be verified at compile time or, for dynamic extents, at array allocation time. In such an environment, the transformation above is correct, as we will now show.

It is convenient to use a lemma, which is good to know in its own right.

LEMMA. *If a and b are signed integers and $a \le b$, then the computed value $b - a$ correctly represents the arithmetic value $b - a$, if the computed value is interpreted as unsigned.*

Proof. (Assume a 32-bit machine.) Because $a \le b$, the true difference $b - a$ is in the range 0 to $(2^{31} - 1) - (-2^{31}) = 2^{32} - 1$. If the true difference is in the range 0 to $2^{31} - 1$, then the machine result is correct (because the result is representable under signed interpretation), and the sign bit is off. Hence the machine result is correct under either signed or unsigned interpretation.

If the true difference is in the range 2^{31} to $2^{32} - 1$, then the machine result will differ by some multiple of 2^{32} (because the result is not representable under signed interpretation). This brings the result (under signed interpretation) to the range -2^{31} to -1. The machine result is too low by 2^{32}, and the sign bit is on. Reinterpreting the result as unsigned increases it by 2 , because the sign bit is given a weight of + 2 rather than -2^{31}. Hence the reinterpreted result is correct.

The “bounds theorem” is

THEOREM. *If a and b are signed integers and $a \le b$, then*

$$a \le x \le b = x - a \overset{u}{\le} b - a. \tag{1}$$

Proof. We distinguish three cases, based on the value of x. In all cases, by the lemma, since $a \le b$, the computed value $b - a$ is equal to the arithmetic value $b - a$ if $b - a$ is interpreted as unsigned, as it is in Equation (1).

Case 1, $x < a$: In this case, $x - a$ interpreted as unsigned is $x - a + 2^{32}$. Whatever the values of x and b are (within the range of 32-bit numbers),

$$x + 2^{32} > b.$$

Therefore

$$x - a + 2^{32} > b - a,$$

and hence

$$x - a \overset{u}{>} b - a.$$

In this case, both sides of Equation (1) are **false**.

Case 2, $a \le x \le b$: Then, arithmetically, $x - a \le b - a$. Because $a \le x$, by the lemma $x - a$ equals the computed value $x - a$ if the latter is interpreted as unsigned. Hence

$$x - a \overset{u}{\le} b - a;$$

that is, both sides of Equation (1) are **true**.

Case 3, $x > b$: Then $x - a > b - a$. Because in this case $x > a$ (because $b \ge a$), by the lemma $x - a$ equals the value of $x - a$ if the latter is interpreted as unsigned. Hence

$$x - a \overset{u}{>} b - a;$$

that is, both sides of Equation (1) are **false**.

The theorem stated above is also true if a and b are *unsigned* integers. This is because for unsigned integers the lemma holds trivially, and the above proof is also valid.

Below is a list of similar bounds-checking transformations, with the theorem above stated again. These all hold for either signed or unsigned interpretations of a, b, and x.

$$\begin{aligned}
&\text{if } a \le b \text{ then } a \le x \le b \;=\; \boldsymbol{x - a \overset{u}{\le} b - a} \;=\; \boldsymbol{b - x \overset{u}{\le} b - a}\\
&\text{if } a \le b \text{ then } a \le x < b \;=\; \boldsymbol{x - a \overset{u}{<} b - a}\\
&\text{if } a \le b \text{ then } a < x \le b \;=\; \boldsymbol{b - x \overset{u}{<} b - a}\\
&\text{if } a < b \text{ then } a < x < b \;=\; \boldsymbol{x - a - 1 \overset{u}{<} b - a - 1} \;=\; \boldsymbol{b - x - 1 \overset{u}{<} b - a - 1}
\end{aligned} \qquad (2)$$

In the last rule, $b - a - \mathbf{1}$ can be replaced with $b + \neg a$.

There are some quite different transformations that may be useful when the test is of the form $-2^{n-1} \le x \le 2^{n-1} - 1$. This is a test to see if a signed quantity x can be correctly represented as an n-bit two's-complement integer. To illustrate with $n = 8$, the following tests are equivalent:

$$\begin{array}{ll}
\text{a.} & \boldsymbol{-128 \le x \le 127}\\
\text{b.} & \boldsymbol{x + 128 \overset{u}{\le} 255}\\
\text{c.} & \boldsymbol{(x \overset{s}{\gg} 7) + 1 \overset{u}{\le} 1}\\
\text{d.} & \boldsymbol{x \overset{s}{\gg} 7 = x \overset{s}{\gg} 31}\\
\text{e.} & \boldsymbol{(x \overset{s}{\gg} 7) + (x \overset{u}{\gg} 31) = 0}\\
\text{f.} & \boldsymbol{(x \ll 24) \overset{s}{\gg} 24 = x}\\
\text{g.} & \boldsymbol{x \oplus (x \overset{s}{\gg} 31) \le 127}
\end{array}$$

Equation (b) is simply an application of the preceding material in this section. Equation (c) is as well, after shifting x right seven positions. Equations (c) – (f) and possibly (g) are probably useful only if the constants in Equations (a) and (b) exceed the size of the immediate fields of the computer's *compare* and *add* instructions.

Another special case involving powers of 2 is

$$0 \le x \le 2^n - 1 \Leftrightarrow (\boldsymbol{x} \overset{u}{\gg} n) = \mathbf{0},$$

or, more generally,

$$a \le x \le a + 2^n - 1 \Leftrightarrow ((\boldsymbol{x - a}) \overset{u}{\gg} n) = \mathbf{0}.$$

4–2 Propagating Bounds through *Add*'s and *Subtract*'s

Some optimizing compilers perform "range analysis" of expressions. This is the process of determining, for each occurrence of an expression in a program, upper and lower bounds on its value. Although this optimization is not a really big winner, it does permit improvements such as omitting the range check on a C "switch" statement and omitting some subscript bounds checks that compilers may provide as a debugging aid.

Suppose we have bounds on two variables x and y as follows, where all quantities are unsigned:

$$\begin{aligned} &a \le x \le b, \quad \text{and} \\ &c \le y \le d. \end{aligned} \qquad (3)$$

Then, how can we compute tight bounds on $x + y$, $x - y$, and $-x$? Arithmetically, of course, $a + c \le x + y \le b + d$; but the point is that the additions may overflow.

The way to calculate the bounds is expressed in the following:

THEOREM. *If a, b, c, d, x, and y are unsigned integers and*

$$\begin{aligned} &a \overset{u}{\le} x \overset{u}{\le} b \quad \text{and} \\ &c \overset{u}{\le} y \overset{u}{\le} d, \end{aligned}$$

then

$$\begin{aligned} &0 \overset{u}{\le} x + y \overset{u}{\le} 2^{32} - 1 \quad \textit{if} \quad a + c \le 2^{32} - 1 \ \textit{ and } \ b + d \ge 2^{32}, \\ &a + c \overset{u}{\le} x + y \overset{u}{\le} b + d \qquad \textit{otherwise;} \end{aligned} \qquad (4)$$

$$\begin{aligned} &0 \overset{u}{\le} x - y \overset{u}{\le} 2^{32} - 1 \quad \textit{if} \quad a - d < 0 \ \textit{ and } \ b - c \ge 0, \\ &a - d \overset{u}{\le} x - y \overset{u}{\le} b - c \qquad \textit{otherwise;} \end{aligned} \qquad (5)$$

$$\begin{aligned} &0 \overset{u}{\le} -x \overset{u}{\le} 2^{32} - 1 \quad \textit{if} \quad a = 0 \ \textit{ and } \ b \ne 0, \\ &-b \overset{u}{\le} -x \overset{u}{\le} -a \qquad \textit{otherwise.} \end{aligned} \qquad (6)$$

Inequalities (4) say that the bounds on $x + y$ are "normally" $a + c$ and $b + d$, but if the calculation of $a + c$ does *not* overflow and the calculation of $b + d$ *does* overflow, then the bounds are 0 and the maximum unsigned integer. Equations (5) are interpreted similarly, but the true result of a subtraction being less than 0 constitutes an overflow (in the negative direction).

Proof. If neither $a + c$ nor $b + d$ overflows, then $x + y$, with x and y in the indicated ranges, cannot overflow, making the computed results equal to the true results, so the second inequality of (4) holds. If both $a + c$ and $b + d$ overflow, then so also does $x + y$. Now arithmetically, it is clear that

$$a + c - 2^{32} \le x + y - 2^{32} \le b + d - 2^{32}.$$

This is what is calculated when the three terms overflow. Hence, in this case also,

$$a + c \overset{u}{\le} x + y \overset{u}{\le} b + d.$$

If $a + c$ does not overflow, but $b + d$ does, then

$$a + c \le 2^{32} - 1 \text{ and } b + d \ge 2^{32}.$$

Because $x + y$ takes on all values in the range $a + c$ to $b + d$, it takes on the values $2^{32} - 1$ and 2^{32}—that is, the computed value $x + y$ takes on the values $2^{32} - 1$ and 0 (although it doesn't take on *all* values in that range).

Lastly, the case that $a + c$ overflows, but $b + d$ does not, cannot occur, because $a \le b$ and $c \le d$.

This completes the proof of inequalities (4). The proof of (5) is similar, but "overflow" means that a true difference is less than 0.

Inequalities (6) can be proved by using (5) with $a = b = 0$, and then renaming the variables. (The expression $-x$ with x an unsigned number means to compute the value of $2^{32} - x$, or of $\neg x + \mathbf{1}$ if you prefer.)

Because unsigned overflow is so easy to recognize (see "Unsigned Add/Subtract" on page 31), these results are easily embodied in code, as shown in Figure 4–1, for addition and subtraction. The computed lower and upper limits are variables `s` and `t`, respectively.

```
s = a + c;
t = b + d;
if (s >= a && t < b) {
   s = 0;
   t = 0xFFFFFFFF;}
```

```
s = a - d;
t = b - c;
if (s > a && t <= b) {
   s = 0;
   t = 0xFFFFFFFF;}
```

FIGURE 4–1. Propagating unsigned bounds through addition and subtraction operations.

Signed Numbers

The case of signed numbers is not so clean. As before, suppose we have bounds on two variables x and y as follows, where all quantities are *signed*:

$$a \le x \le b, \quad \text{and}$$
$$c \le y \le d.$$

We wish to compute tight bounds on $x + y$, $x - y$, and $-x$. The reasoning is very similar to that for the case of unsigned numbers, and the results for addition are shown below.

$$\begin{array}{rl}
a + c < -2^{31},\ b + d < -2^{31}: & \boldsymbol{a + c \le x + y \le b + d} \\
a + c < -2^{31},\ b + d \ge -2^{31}: & \boldsymbol{-2^{31} \le x + y \le 2^{31} - 1} \\
-2^{31} \le a + c < 2^{31},\ b + d < 2^{31}: & \boldsymbol{a + c \le x + y \le b + d} \\
-2^{31} \le a + c < 2^{31},\ b + d \ge 2^{31}: & \boldsymbol{-2^{31} \le x + y \le 2^{31} - 1} \\
a + c \ge 2^{31},\ b + d \ge 2^{31}: & \boldsymbol{a + c \le x + y \le b + d}
\end{array} \qquad (7)$$

The first row means that if both of the additions $a + c$ and $b + d$ overflow in the negative direction, then the computed sum $x + y$ lies between the computed sums $a + c$ and $b + d$. This is because all three computed sums are too high by the same amount (2^{32}). The second row means that if the addition $a + c$ overflows in the negative direction, and the addition $b + d$ either does not overflow or overflows in the positive direction, then the computed sum $x + y$ can take on the extreme negative number and the extreme positive number (although perhaps not all values in between), which is not difficult to show. The other rows are interpreted similarly.

The rules for propagating bounds on signed numbers through the subtraction operation can easily be derived by rewriting the bounds on y as

$$-d \le -y \le -c$$

and using the rules for addition. The results are shown below.

$$\begin{aligned}
a-d<-2^{31}, b-c<-2^{31} &: \boldsymbol{a-d \le x-y \le b-c}\\
a-d<-2^{31}, b-c \ge -2^{31} &: \boldsymbol{-2^{31} \le x-y \le 2^{31}-1}\\
-2^{31} \le a-d<2^{31}, b-c<2^{31} &: \boldsymbol{a-d \le x-y \le b-c}\\
-2^{31} \le a-d<2^{31}, b-c \ge 2^{31} &: \boldsymbol{-2^{31} \le x-y \le 2^{31}-1}\\
a-d \ge 2^{31}, b-c \ge 2^{31} &: \boldsymbol{a-d \le x-y \le b-c}
\end{aligned}$$

The rules for negation can be derived from the rules for subtraction by taking $a = b = 0$, omitting some impossible combinations, simplifying, and renaming. The results are as follows:

$$\begin{aligned}
a=-2^{31}, b=-2^{31} &: \boldsymbol{-x=-2^{31}}\\
a=-2^{31}, b \ne -2^{31} &: \boldsymbol{-2^{31} \le -x \le 2^{31}-1}\\
a \ne -2^{31} &: \boldsymbol{-b \le -x \le -a}
\end{aligned}$$

C code for the case of signed numbers is a bit messy. We will consider only addition. It seems to be simplest to check for the two cases in (7) in which the computed limits are the extreme negative and positive numbers. Overflow in the negative direction occurs if the two operands are negative and the sum is nonnegative (see “Signed Add/Subtract” on page 28). Thus, to check for the condition that $a + c < -2^{31}$, we could let `s = a + c;` and then code something like “`if (a < 0 && c < 0 && s >= 0)` ....” It will be more efficient,[1] however, to perform logical operations directly on the arithmetic variables, with the sign bit containing the true/false result of the logical operations. Then, we write the above condition as “`if ((a & c & ~s) < 0)` ....” These considerations lead to the program fragment shown in Figure 4–2.

```
s = a + c;
t = b + d;
u = a & c & ~s & ~(b & d &~t);
v = ((a ^ c) | ~(a ^ s)) & (~b & ~d & t);
if ((u | v) < 0) {
   s = 0x80000000;
   t = 0x7FFFFFFF;}
```

FIGURE 4–2. Propagating signed bounds through an addition operation.

Here `u` is **true** (sign bit is 1) if the addition `a + c` overflows in the negative direction, and the addition `b + d` does *not* overflow in the negative direction. Variable `v` is **true** if the addition `a + c` does not overflow and the addition `b + d` overflows in the positive direction. The former condition can be expressed as "`a` and `c` have different signs, or `a` and `s` have the same sign." The "`if`" test is equivalent to "`if (u < 0 || v <` 0)—that is, if either `u` or `v` is **true**."

4–3 Propagating Bounds through Logical Operations

As in the preceding section, suppose we have bounds on two variables x and y as follows, where all quantities are unsigned:

$$\begin{aligned} a \le x \le b, &\quad \text{and} \\ c \le y \le d. \end{aligned} \tag{8}$$

Then what are some reasonably tight bounds on $x \mid y$, $x \,\&\, y$, x y, and $\neg x$?

Combining inequalities (8) with some inequalities from Section 2–3 on page 17, and noting that $\neg x = \mathbf{2^{32} - 1} - x$, yields

$$\begin{aligned} \max(\boldsymbol{a}, \boldsymbol{c}) \le (x \mid y) &\le \boldsymbol{b} + \boldsymbol{d}, \\ \mathbf{0} \le (x \,\&\, y) &\le \min(\boldsymbol{b}, \boldsymbol{d}), \\ \mathbf{0} \le (x \oplus y) &\le \boldsymbol{b} + \boldsymbol{d}, \quad \text{and} \\ \neg \boldsymbol{b} \le \neg x &\le \neg \boldsymbol{a}, \end{aligned}$$

where it is assumed that the addition $b + d$ does not overflow. These are easy to compute and might be good enough for the compiler application mentioned in the preceding section; however, the bounds in the first two inequalities are not tight. For example, writing constants in binary, suppose

$$\begin{aligned} \mathbf{00010} \le x \le \mathbf{00100}, &\quad \text{and} \\ \mathbf{01001} \le y \le \mathbf{10100}. \end{aligned} \tag{9}$$

Then, by inspection (e.g., trying all 36 possibilities for x and y), we see that $\mathbf{01010} \le (x \mid y) \le \mathbf{10111}$. Thus, the lower bound is not $\max(a, c)$, nor is it $a \mid c$, and the upper bound is not $b + d$, nor is it $b \mid d$.

Given the values of a, b, c, and d in inequalities (8), how can one obtain tight bounds on the logical expressions? Consider first the minimum value attained by $x \mid y$. A reasonable guess might be the value of this expression with x and y both at their minima—that is, $a \mid c$. Example (9), however, shows that the minimum can be lower than this.

To find the minimum, our procedure is to start with $x = a$ and $y = c$, and then find an amount by which to increase either x or y so as to reduce the value of $x \mid y$. The result will be this reduced value. Rather than assigning a and c to x and y, we work directly with a and c, increasing one of them when doing so is valid and it reduces the value of $a \mid c$.

The procedure is to scan the bits of a and c from left to right. If both bits are 0, the result will have a 0 in that position. If both bits are 1, the result will have a 1 in that position (clearly, no values of *x* and *y* could make the result less). In these cases, continue the scan to the next bit position. If one scanned bit is 1 and the other is 0, then it is possible that changing the 0 to 1 and setting all the following bits in that bound's value to 0 will reduce the value of *a* | *c*. This change will not increase the value of *a* | *c*, because the result has a 1 in that position anyway, from the other bound. Therefore, form the number with the 0 changed to 1 and subsequent bits changed to 0. If that is less than or equal to the corresponding upper limit, the change can be made; do it, and the result is the *or* of the modified value with the other lower bound. If the change cannot be made (because the altered value exceeds the corresponding upper bound), continue the scan to the next bit position.

That's all there is to it. It might seem that after making the change the scan should continue, looking for other opportunities to further reduce the value of *a* | *c*. However, even if a position is found that allows a 0 to be changed to 1, setting the subsequent bits to 0 does not reduce the value of *a* | *c*, because those bits are already 0.

C code for this algorithm is shown in Figure 4–3. We assume that the compiler will move the subexpressions `~a & c` and `a & ~c` out of the loop. More significantly, if the *number of leading zeros* instruction is available, the program can be speeded up by initializing `m` with

```
m = 0x80000000 >> nlz(a ^ c);
```

```
unsigned minOR(unsigned a, unsigned b,
               unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (~a & c & m) {
         temp = (a | m)& -m;
         if (temp <= b) {a = temp; break;}
      }
      else if (a & ~c & m) {
         temp = (c | m) & -m;
         if (temp <= d) {c = temp; break;}
      }
      m = m >> 1;
   }
   return a | c;
}
```

FIGURE 4–3. Minimum value of *x* | *y* with bounds on *x* and *y*.

This skips over initial bit positions in which `a` and `c` are both 0 or both 1. For this speedup to be effective when `a ^ c` is 0 (that is, when `a = c`), the machine's *shift right* instruction should be mod-64. If *number of leading zeros* is not available, it may be worthwhile to use some version of the flp2 function (see page 60) with argument `a ^ c`.

Now let us consider the *maximum* value attained by *x* |*y*, with the variables bounded as shown in inequalities (8). The algorithm is similar to that for the minimum, except it scans the values of bounds *b* and *d* (from left to right), looking for a position in which both bits are 1. If such a position is found, the algorithm tries to increase the value of *c* | *d* by decreasing one of the bounds by changing the 1 to 0, and setting all subsequent bits in that bound to 1. If this is acceptable (if the resulting value is greater than or equal to the corresponding lower bound), the change is made and the result is the value of *c* | *d* using the modified bound. If the change cannot be done, it is attempted on the other bound. If the change cannot be done to either bound, the scan continues. C code for this algorithm is shown in Figure 4–4. Here the subexpression `b & d` can be moved out of the loop, and the algorithm can be speeded up by initializing `m` with

```
unsigned maxOR(unsigned a, unsigned b,
               unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (b & d & m) {
         temp = (b - m) | (m - 1);
         if (temp >= a) {b = temp; break;}
         temp = (d - m) | (m - 1);
         if (temp >= c) {d = temp; break;}
      }
      m = m >> 1;
   }
   return b | d;
}
```

FIGURE 4–4. Maximum value of *x* | *y* with bounds on *x* and *y*.

```
   m = 0x80000000 >> nlz(b & d);
```

There are two ways in which we might propagate the bounds of inequalities (8) through the expression *x* & *y*: algebraic and direct computation. The algebraic method uses DeMorgan’s rule:

$$x \mathbin{\&} y = \neg(\neg x \mid \neg y)$$

Because we know how to propagate bounds precisely through *or*, and it is trivial to propagate them through *not* ($a \overset{u}{\le} x \overset{u}{\le} b \Leftrightarrow \neg b \overset{u}{\le} \neg x \overset{u}{\le} \neg a$), we have

minAND(a, b, c, d) = ¬maxOR($\neg b, \neg a, \neg d, \neg c$), and
maxAND(a, b, c, d) = ¬minOR($\neg b, \neg a, \neg d, \neg c$).

For the direct computation method, the code is very similar to that for propagating bounds through *or*. It is shown in Figures 4–5 and 4–6.

```
   unsigned minAND(unsigned a, unsigned b,
                   unsigned c, unsigned d) {
      unsigned m, temp;

      m = 0x80000000;
      while (m != 0) {
         if (~a & ~c & m) {
            temp = (a | m) & -m;
            if (temp <= b) {a = temp; break;}
            temp = (c | m) & -m;
            if (temp <= d) {c = temp; break;}
         }
         m = m >> 1;
      }
      return a & c;
   }
```

FIGURE 4–5. Minimum value of *x*& *y* with bounds on *x* and *y*.

```
   unsigned maxAND(unsigned a, unsigned b,
                   unsigned c, unsigned d) {
      unsigned m, temp;

      m = 0x80000000;
      while (m != 0) {
         if (b & ~d & m) {
            temp = (b & ~m) | (m - 1);
            if (temp >= a) {b = temp; break;}
         }
         else if (~b & d & m) {
            temp = (d & ~m) | (m - 1);
            if (temp >= c) {d = temp; break;}
         }
         m = m >> 1;
      }
      return b & d;
   }
```

FIGURE 4–6. Maximum value of *x*& *y* with bounds on *x* and *y*.

The algebraic method of finding bounds on expressions in terms of the functions for *and, or*, and *not* works for all the binary logical expressions except *exclusive or* and *equivalence*. The reason these two present a difficulty is that when expressed in terms of *and, or*, and *not*, there are two terms containing *x* and *y*. For example, we are to find

$$\min_{\substack{a \le x \le b \\ c \le y \le d}} (x \oplus y) = \min_{\substack{a \le x \le b \\ c \le y \le d}} ((x \,\&\, \neg y) \mid (\neg x \,\&\, y)).$$

The two operands of the *or* cannot be separately minimized (without proof that it works, which actually it does), because we seek one value of *x* and one value of *y* that minimizes the whole *or* expression.

The following expressions can be used to propagate bounds through *exclusive or*:

$$\begin{aligned}
\text{minXOR}(a, b, c, d) &= \text{minAND}(a, b, \neg d, \neg c) \mid \text{minAND}(\neg b, \neg a, c, d), \\
\text{maxXOR}(a, b, c, d) &= \text{maxOR}(0, \text{maxAND}(a, b, \neg d, \neg c), \\
&\qquad\qquad 0, \text{maxAND}(\neg b, \neg a, c, d)).
\end{aligned}$$

It is straightforward to evaluate the minXOR and maxXOR functions by direct computation. The code for minXOR is the same as that for minOR (Figure 4–3) except with the two `break` statements removed, and the return value changed to `a ^ c`. The code for maxXOR is the same as that for maxOR (Figure 4–4) except with the four lines under the `if` clause replaced with

```
         temp = (b - m) | (m - 1);
```

```
        if (temp >= a) b = temp;
        else {
           temp = (d - m) | (m - 1);
           if (temp >= c) d = temp;
        }
```

and the return value changed to `b ^ d`.

Signed Bounds

If the bounds are *signed* integers, propagating them through logical expressions is substantially more complicated. The calculation is irregular if 0 is within the range *a* to *b*, or *c* to *d*. One way to calculate the lower and upper bounds for the expression *x* | *y* is shown in Table 4–1. A "+" entry means that the bound at the top of the column is greater than or equal to 0, and a "–" entry means that it is less than 0. The column labeled "minOR (signed)" contains expressions for computing the lower bound of *x* | *y*, and the last column contains expressions for computing the upper bound of *x* | *y*. One way to program this is to construct a value ranging from 0 to 15 from the sign bits of *a, b, c*, and *d*, and use a "switch" statement. Notice that not all values from 0 to 15 are used, because it is impossible to have *a* > *b* or *c* > *d*.

TABLE 4–1. SIGNED MINOR AND MAXOR FROM UNSIGNED

a	*b*	*c*	*d*	**minOR (signed)**	**maxOR (signed)**
–	–	–	–	minOR(***a***, ***b***, ***c***, ***d***)	maxOR(***a***, ***b***, ***c***, ***d***)
–	–	–	+	***a***	**–1**
–	–	+	+	minOR(***a***, ***b***, ***c***, ***d***)	maxOR(***a***, ***b***, ***c***, ***d***)
–	+	–	–	***c***	**–1**
–	+	–	+	min(***a***, ***c***)	maxOR(**0**, ***b***, **0**, ***d***)
–	+	+	+	minOR(***a***, **0xFFFFFFFF**, ***c***, ***d***)	maxOR(**0**, ***b***, ***c***, ***d***)
+	+	–	–	minOR(***a***, ***b***, ***c***, ***d***)	maxOR(***a***, ***b***, ***c***, ***d***)
+	+	–	+	minOR(***a***, ***b***, ***c***, **0xFFFFFFFF**)	maxOR(***a***, ***b***, **0**, ***d***)
+	+	+	+	minOR(***a***, ***b***, ***c***, ***d***)	maxOR(***a***, ***b***, ***c***, ***d***)

For signed numbers, the relation

$$a \le x \le b \quad \neg b \le \neg x \le \neg a$$

holds, so the algebraic method can be used to extend the results of Table 4–1 to other logical expressions (except for *exclusive or* and *equivalence*). We leave this and similar extensions to others.

Exercises

1. For unsigned integers, what are the bounds on *x* – *y* if

$$0 \overset{u}{\leq} x \overset{u}{\leq} b \text{ and}$$

$$0 \overset{u}{\leq} y \overset{u}{\leq} d?$$

2. Show how the maxOR function (Figure 4–4) can be simplified if either `a = 0` or `c = 0` on a machine that has the *number of leading zeros* instruction.

Chapter 5. Counting Bits

5–1 Counting 1-Bits

The IBM Stretch computer (ca. 1960) had a means of counting the number of 1-bits in a word, as well as the number of leading 0's. It produced these two quantities as a by-product of all logical operations! The former function is sometimes called *population count* (e.g., on Stretch and the SPARCv9).

For machines that don't have this instruction, a good way to count the number of 1-bits is to first set each 2-bit field equal to the sum of the two single bits that were originally in the field, and then sum adjacent 2-bit fields, putting the results in each 4-bit field, and so on. A more complete discussion of this trick is in [RND]. The method is illustrated in Figure 5–1, in which the first row shows a computer word whose 1-bits are to be summed, and the last row shows the result (23 decimal).

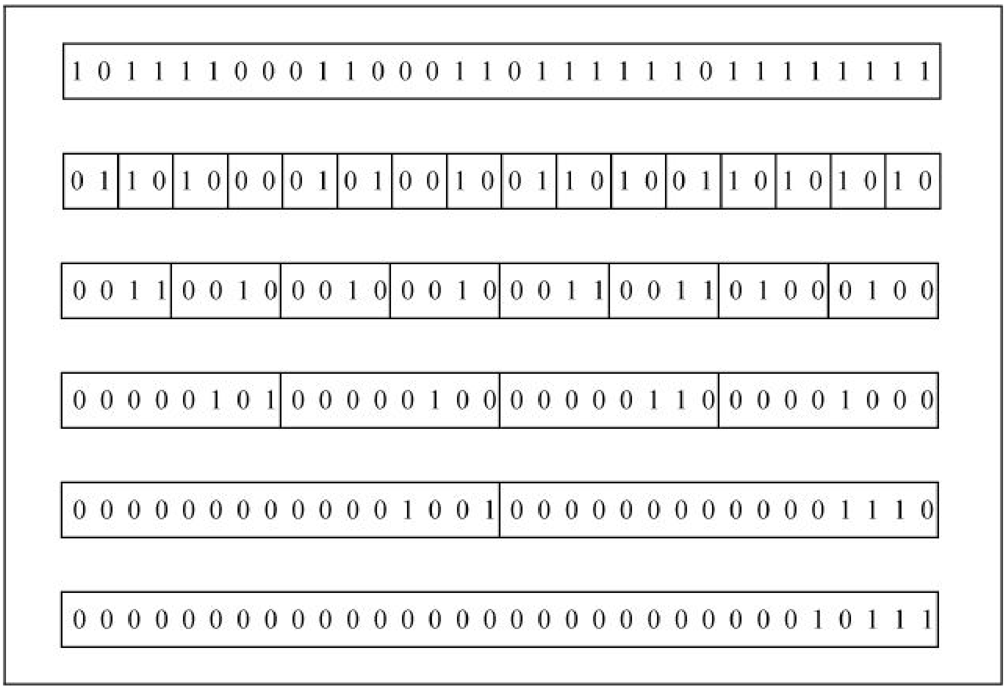


FIGURE 5–1. Counting 1-bits, "divide and conquer" strategy.

This is an example of the "divide and conquer" strategy, in which the original problem (summing 32 bits) is divided into two problems (summing 16 bits), which are solved separately, and the results are combined (added, in this case). The strategy is applied recursively, breaking the 16-bit fields into 8-bit fields, and so on.

In the case at hand, the ultimate small problems (summing adjacent bits) can all be done in parallel, and combining adjacent sums can also be done in parallel in a fixed number of steps at each stage. The result is an algorithm that can be executed in $\log_2(32) = 5$ steps.

Other examples of divide and conquer are the well-known techniques of binary search, a sorting method known as quicksort, and a method for reversing the bits of a word, discussed on page 129.

The method illustrated in Figure 5–1 can be committed to C code as

```
x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF);
```

The first line uses `(x >> 1) & 0x55555555` rather than the perhaps more natural `(x & 0xAAAAAAAA) >> 1`, because the code shown avoids generating two large constants in a register. This would cost an instruction if the machine lacks the *and not* instruction. A similar remark applies to the other lines.

Clearly, the last *and* is unnecessary, and other *and*'s can be omitted when there is no danger that a field's sum will carry over into the adjacent field. Furthermore, there is a way to code the first line that uses one fewer instruction. This leads to the simplification shown in Figure 5–2, which executes in 21 instructions and is branch-free.

```
int pop(unsigned x) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   x = (x + (x >> 4)) & 0x0F0F0F0F;
   x = x + (x >> 8);
   x = x + (x >> 16);
   return x & 0x0000003F;
}
```

FIGURE 5–2. Counting 1-bits in a word.

The first assignment to `x` is based on the first two terms of the rather surprising formula

$$\text{pop}(x) = x - \left\lfloor \frac{x}{2} \right\rfloor - \left\lfloor \frac{x}{4} \right\rfloor - \dots - \left\lfloor \frac{x}{2^{31}} \right\rfloor. \qquad (1)$$

In Equation (1), we must have $x \geq 0$. By treating x as an unsigned integer, Equation (1) can be implemented with a sequence of 31 *shift right immediate*'s of 1, and 31 *subtract*'s. The procedure of Figure 5–2 uses the first two terms of this on each 2-bit field, in parallel.

There is a simple proof of Equation (1), which is shown below for the case of a four-bit word. Let the word be $b_3 b_2 b_1 b_0$, where each $b_i = 0$ or 1. Then,

$$\begin{aligned}
x - \left\lfloor \frac{x}{2} \right\rfloor - \left\lfloor \frac{x}{4} \right\rfloor - \left\lfloor \frac{x}{8} \right\rfloor &= b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \\
&\quad -(b_3 \cdot 2^2 + b_2 \cdot 2^1 + b_1 \cdot 2^0) \\
&\quad -(b_3 \cdot 2^1 + b_2 \cdot 2^0) \\
&\quad -(b_3 \cdot 2^0) \\
&= b_3(2^3 - 2^2 - 2^1 - 2^0) + b_2(2^2 - 2^1 - 2^0) + b_1(2^1 - 2^0) + b_0(2^0) \\
&= b_3 + b_2 + b_1 + b_0.
\end{aligned}$$

Alternatively, Equation (1) can be derived by noting that bit *i* of the binary representation of a nonnegative integer *x* is given by

$$b_i = \left\lfloor \frac{x}{2^i} \right\rfloor - 2\left\lfloor \frac{x}{2^{i+1}} \right\rfloor$$

and summing this for $i = 0$ to 31. Work it out—the last term is 0 because $x < 2^{32}$. Equation (1) generalizes to other bases. For base ten it is

$$\text{sum\_digits}(x) = x - 9\left\lfloor \frac{x}{10} \right\rfloor - 9\left\lfloor \frac{x}{100} \right\rfloor - \dots$$

where the terms are carried out until they are 0. This can be proved by essentially the same technique used above.

A variation of the above algorithm is to use a base 4 analogue of Equation (1) as a substitute for the second executable line of Figure 5–2:

```
x = x - 3*((x >> 2) & 0x33333333)
```

This code, however, uses the same number of instructions as the line it replaces (six), and requires a fast *multiply-by-3* instruction.

An algorithm in HAKMEM memo [HAK, item 169] counts the number of 1-bits in a word by using the first three terms of (1) to produce a word of 3-bit fields, each of which contains the number of 1-bits that were in it. It then adds adjacent 3-bit fields to form 6-bit field sums, and then adds the 6-bit fields by computing the value of the word modulo 63. Expressed in C, the algorithm is (the long constants are in octal)

```
int pop(unsigned x) {
   unsigned n;

   n = (x >> 1) & 033333333333;       // Count bits in
   x = x - n;                         // each 3-bit
   n = (n >> 1) & 033333333333;       // field.
   x = x - n;
   x = (x + (x >> 3)) & 030707070707; // 6-bit sums.
   return x%63;                       // Add 6-bit sums.
}
```

The last line uses the *unsigned modulus* function. (It could be either signed or unsigned if the word length were a multiple of 3.) That the modulus function sums the 6-bit fields becomes clear by regarding the word x as an integer written in base 64. The remainder upon dividing a base b integer by $b - 1$ is, for $b \geq 3$, congruent mod $b - 1$ to the sum of the digits and, of course, is less than $b - 1$. Because the sum of the digits in this case must be less than or equal to 32, mod(x, 63) must be equal to the sum of the digits of x, which is to say equal to the number of 1-bits in the original x.

This algorithm requires only ten instructions on the DEC PDP-10, because that machine has an instruction for computing the remainder with its second operand directly referencing a fullword in memory. On a basic RISC, it requires about 13 instructions, assuming the machine has *unsigned modulus* as one instruction (but not directly referencing a fullword immediate or memory operand). It is probably not very fast, because division is almost always a slow operation. Also, it doesn't apply to 64-bit word lengths by simply extending the constants, although it does work for word lengths up to 62.

The return statement in the code above can be replaced with the following, which runs faster on most machines, but is perhaps less elegant (octal notation again).

```
return ((x * 0404040404) >> 26) +   // Add 6-bit sums.
        (x >> 30);
```

A variation on the HAKMEM algorithm is to use Equation (1) to count the number of 1's in each 4-bit field, working on all eight 4-bit fields in parallel [Hay1]. Then, the 4-bit sums can be converted to 8-bit sums in a straightforward way, and the four bytes can be added with a multiplication by 0x01010101. This gives

```
int pop(unsigned x) {
   unsigned n;

   n = (x >> 1) & 0x77777777;          // Count bits in
   x = x - n;                          // each 4-bit
   n = (n >> 1) & 0x77777777;          // field.
   x = x - n;
   n = (n >> 1) & 0x77777777;
   x = x - n;
   x = (x + (x >> 4)) & 0x0F0F0F0F;    // Get byte sums.
   x = x*0x01010101;                   // Add the bytes.
   return x >> 24;
}
```

This is 19 instructions on the basic RISC. It works well if the machine is two-address, because the first six lines can be done with only one *move register* instruction. Also, the repeated use of the mask 0x77777777 permits loading it into a register and referencing it with register-to-register instructions. Furthermore, most of the shifts are of only one position.

A quite different bit-counting method, illustrated in Figure 5–3, is to turn off the rightmost 1-bit repeatedly [Weg, RND], until the result is 0. It is very fast if the number of 1-bits is small, taking 2 + 5pop(x) instructions.

```
int pop(unsigned x) {
   int n;

   n = 0;
   while (x ! = 0) {
```

```
            n = n+ 1;
            x = x & (x - 1);
        }
        returnn;
    }
```

FIGURE 5–3. Counting 1-bits in a sparsely populated word.

This has a dual algorithm that is applicable if the number of 1-bits is expected to be large. The dual algorithm keeps turning on the rightmost 0-bit with `x = x | (x + 1)`, until the result is all 1's (–1). Then, it returns 32 – *n*. (Alternatively, the original number *x* can be complemented, or *n* can be initialized to 32 and counted down.)

A rather amazing algorithm is to rotate *x* left one position, 31 times, adding the 32 terms [MM]. The sum is the negative of pop(*x*)! That is,

$$\text{pop}(x) = -\sum_{i=0}^{31} (x \overset{rot}{\ll} i), \qquad (2)$$

where the additions are done modulo the word size, and the final sum is interpreted as a two's-complement integer. This is just a novelty; it would not be useful on most machines, because the loop is executed 31 times and thus it requires 63 instructions, plus the loop-control overhead.

To see why Equation (2) works, consider what happens to a single 1-bit of *x*. It gets rotated to all positions, and when these 32 numbers are added, a word of all 1-bits results. This is –1. To illustrate, consider a 6-bit word size and *x* = **001001** (binary):

$$\begin{array}{ll}
\mathbf{001001} & x \\
\mathbf{010010} & x \overset{rot}{\ll} 1 \\
\mathbf{100100} & x \overset{rot}{\ll} 2 \\
\mathbf{001001} & x \overset{rot}{\ll} 3 \\
\mathbf{010010} & x \overset{rot}{\ll} 4 \\
\mathbf{100100} & x \overset{rot}{\ll} 5
\end{array}$$

Of course, *rotate-right* would work just as well.

The method of Equation (1) is very similar to this "rotate and sum" method, which becomes clear by rewriting (1) as

$$\text{pop}(x) = x - \sum_{i=1}^{31} (x \overset{u}{\gg} i).$$

This gives a slightly better algorithm than Equation (2) provides. It is better because it uses *shift right*, which is more commonly available than *rotate*, and because the loop can be terminated when the shifted quantity becomes 0. This reduces the loop-control code and may save a few iterations. The two algorithms are contrasted in Figure 5–4.

```
int pop(unsigned x) {
   int i, sum;

// Rotate and sum method            // Shift right & subtract

   sum = x; // sum = x;
   for (i = 1; i <= 31; i++) {      // while (x != 0) {
       x = rotatel (x, 1);          //    x = x >> 1;
       sum = sum + x;               //    sum = sum - x;
   }                                // }
   return -sum;                     // return sum;
}
```

FIGURE 5–4. Two similar bit-counting algorithms.

A less interesting algorithm that may be competitive with all the algorithms for pop(*x*) in this section is to have a table that contains pop(*x*) for, say, *x* in the range 0 to 255. The table can be accessed four times, adding the four numbers obtained. A branch-free version of the algorithm looks like this:

```
int pop(unsigned x) {                    // Table lookup.
   static char table[256] = {
      0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
      ...
      4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8};
   return table[x         & 0xFF] +
          table[(x >>  8) & 0xFF] +
          table[(x >> 16) & 0xFF] +
          table[(x >> 24)];
}
```

Item 167 in [HAK] contains a short algorithm for counting the number of 1-bits in a 9-bit quantity that is right-adjusted and isolated in a register. It works only on machines with registers of 36 or more bits. Below is a version of that algorithm that works on 32-bit machines, but only for 8-bit quantities.

```
x = x * 0x08040201;  // Make 4 copies.
x = x >> 3;          // So next step hits proper bits.
x = x & 0x11111111;  // Every 4th bit.
x = x * 0x11111111;  // Sum the digits (each 0 or 1).
x = x >> 28;         // Position the result.
```

A version for 7-bit quantities is

```
x = x * 0x02040810;  // Make 4 copies, left-adjusted.
x = x & 0x11111111;  // Every 4th bit.
x = x * 0x11111111;  // Sum the digits (each 0 or 1).
x = x >> 28;         // Position the result.
```

In these, the last two steps can be replaced with steps to compute the remainder of `x` modulo 15.

These are not particularly good; most programmers would probably prefer to use table lookup. The latter algorithm above, however, has a version that uses 64-bit arithmetic, which might be useful for a 64-bit machine that has fast multiplication. Its argument is a 15-bit quantity. (I don't believe there is a similar algorithm that deals with 16-bit quantities, unless it is known that not all 16 bits are 1.) The data type `long long` is a C extension found in many C compilers, old and new, for 64-bit integers. It is made official in the C99 standard. The suffix `ULL` makes `unsigned long long` constants.

```
int pop(unsigned x) {
   unsigned long long y;
   y = x * 0x0002000400080010ULL;
   y = y & 0x1111111111111111ULL;
   y = y * 0x1111111111111111ULL;
   y = y >> 60;
   return y;
}
```

Sum and Difference of Population Counts of Two Words

To compute pop(x) + pop(y) (if your computer does not have the *population count* instruction), some time can be saved by using the first two lines of Figure 5–2 on x and y separately, adding x and y, and then executing the last three stages of the algorithm on the sum. After the first two lines of Figure 5–2 are executed, x and y consist of eight 4-bit fields, each containing a maximum value of 4. Thus, x and y can safely be added, because the maximum value in any 4-bit field of the sum would be 8, so no overflow occurs. (In fact, three words can be combined in this way.)

This idea also applies to subtraction. To compute pop(x) – pop(y), use

$$\begin{aligned}\text{pop}(x)-\text{pop}(y) &= \text{pop}(x)-(32-\text{pop}(\bar{y}))\\ &= \text{pop}(x)+\text{pop}(\bar{y})-32.\end{aligned}$$

Then, use the technique just described to compute pop(x) + pop(y). The code is shown in Figure 5–5. It uses 32 instructions, versus 43 for two applications of the code in Figure 5–2 followed by a subtraction.

```
int popDiff(unsigned x, unsigned y) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   y = ~y;
   y = y - ((y >> 1) & 0x55555555);
   y = (y & 0x33333333) + ((y >> 2) & 0x33333333);
   x = x + y;
   x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
   x = x + (x >> 8);
   x = x + (x >> 16);
   return (x & 0x0000007F) - 32;
}
```

FIGURE 5–5. Computing pop(x) – pop(y).

Comparing the Population Counts of Two Words

Sometimes one wants to know which of two words has the larger population count without regard to the actual counts. Can this be determined without doing a population count of the two words? Computing the difference of two population counts as in Figure 5–5, and comparing the result to 0 is one way, but there is another way that is preferable if either the population counts are expected to be low or if there is a strong correlation between the particular bits that are set in the two words.

The idea is to clear a single bit in each word until one of the words is all zero; the other word then has the larger population count. The process runs faster in its worst and average cases if the bits that are 1 at the same positions in each word are first cleared. The code is shown in Figure 5–6. The procedure returns a negative integer if pop(x) < pop(y), 0 if pop(x) = pop(y), and a positive integer (1) if pop(x) > pop(y).

```
int popCmpr(unsigned xp, unsigned yp) {
   unsigned x, y;
   x = xp & ~yp;                 // Clear bits where
   y = yp & ~xp;                 // both are 1.
   while (1) {
      if (x == 0) return y | -y;
      if (y == 0) return 1;
      x = x & (x - 1);           // Clear one bit
      y = y & (y - 1);           // from each.
   }
}
```

FIGURE 5–6. Comparing pop(*x*) with pop(*y*).

After clearing the common 1-bits in each 32-bit word, the maximum possible number of 1-bits in both words together is 32. Therefore, the word with the smaller number of 1-bits can have at most 16. Thus, the loop in Figure 5–6 is executed a maximum of 16 times, which gives a worst case of 119 instructions executed on the basic RISC ($16 \cdot 7 + 7$). A simulation using uniformly distributed random 32-bit integers showed that the average population count of the word with the smaller population count is approximately 6.186, after clearing the common 1-bits. This gives an average execution time of about 50 instructions executed for random 32-bit inputs, not as good as using Figure 5–5. For this procedure to beat that of Figure 5–5, the number of 1-bits in either x or y, after clearing the common 1-bits, would have to be three or less.

Counting the 1-bits in an Array

The simplest way to count the number of 1-bits in an array (vector) of fullwords, in the absence of the population count instruction, is to use a procedure such as that of Figure 5–2 on page 82 on each word of the array and simply add the results. We call this the “naive” method. Ignoring loop control, the generation of constants, and loads from the array, it takes 16 instructions per word: 15 for the code of Figure 5–2, plus one for the addition. We assume the procedure is expanded in line, the masks are loaded outside the loop, and the machine has a sufficient number of registers to hold all the quantities used in the calculation.

Another way is to use the first two executable lines of Figure 5–2 on groups of three words in the array, adding the three partial results. Because each partial result has a maximum value of 4 in each four-bit field, the sum of the three has a maximum value of 12 in each four-bit field, so no overflow occurs. This idea can be applied to the 8- and 16-bit fields. Coding and compiling this method indicates that it gives about a 20% reduction over the naive method in total number of instructions executed on the basic RISC. Much of the savings are cancelled by the additional housekeeping instructions required. We will not dwell on this method because there is a *much* better way to do it.

The better way seems to have been invented by Robert Harley and David Seal in about 1996 [Seal1]. It is based on a circuit called a *carry-save adder* (CSA), or 3:2 compressor. A CSA is simply a sequence of independent full adders[1] [H&P], and it is often used in binary multiplier circuits.

In Boolean algebra notation, the logic for each full adder is

$$
\begin{aligned}
&h \leftarrow ab + ac + bc = ab + (a + b)c = ab + (a \quad b)c,\\
&l \leftarrow (a \quad b) \quad c.
\end{aligned}
$$

where *a, b*, and *c* are the 1-bit inputs, *l* is the low-bit output (sum) and *h* is the high-bit output (carry). Changing $a + b$ on the first line to $a \quad b$ is justified because when *a* and *b* are both 1, the term *ab* makes the value of the whole expression 1. By first assigning $a \quad b$ to a temporary, the full adder logic can be evaluated in five logical instructions, each operating on 32 bits in parallel (on a 32-bit machine). We will refer to these five instructions as CSA(*h, l, a, b, c*). This is a "macro," with *h* and *l* being outputs.

One way to use the CSA operation is to process elements of the array *A* in groups of three, reducing each group of three words to two, and applying the population count operation to these two words. In the loop, these two population counts are summed. After executing the loop, the total population count of the array is twice the accumulated population count of the CSA's high-bit outputs, plus the accumulated population count of the low-bit outputs.

Let n_c be the number of instructions required for the CSA steps and n_p be the number of instructions required to do the population count of one word. On a typical RISC machine $n_c = 5$ and $n_p = 15$. Ignoring loads from the array and loop control (the code for which may vary quite a bit from one machine to another), the loop discussed above takes $(n_c + 2n_p + 2)/3 \approx 12.33$ instructions per word of the array (the "+2" is for the two additions in the loop). This is in contrast to the 16 instructions per word required by the naive method.

There is another way to use the CSA operation that results in a program that's more efficient and slightly more compact. This is shown in Figure 5–7. It takes $(n_c + n_p + 1)/2 = 10.5$ instructions per word (ignoring loop control and loads). In this code, the CSA operation expands into

```
#define CSA(h,l, a,b,c) \
   {unsigned u = a ^ b; unsigned v = c; \
      h = (a & b) | (u & v); l = u ^ v;}

int popArray(unsigned A[], int n) {

   int tot, i;
   unsigned ones, twos;

   tot = 0;                      // Initialize.
   ones = 0;
   for (i = 0; i <= n - 2; i = i + 2) {
      CSA(twos, ones, ones, A[i], A[i+1])
      tot = tot + pop(twos);
   }
   tot = 2*tot + pop(ones);

   if (n & 1)                    // If there's a last one,
      tot = tot + pop(A[i]);      // add it in.
   return tot;
}
```

FIGURE 5–7. Array population count, processing elements in groups of two.

```
  u = ones ^ A[i];
  v = A[i+1];
  twos = (ones & A[i]) | (u & v);
  ones = u ^ v;
```

The code relies on the compiler to common the loads.

There are ways to use the CSA operation to further reduce the number of instructions required to compute the population count of an array. They are most easily understood by means of a circuit diagram. For example, Figure 5–8 illustrates a way to code a loop that takes array elements eight at a time and compresses them into four quantities, labeled *eights, fours, twos*, and *ones*. The *fours, twos*, and *ones* are fed back into the CSAs on the next loop iteration, and the 1-bits in *eights* are counted by an execution of the word-level population count function, and this count is accumulated. When all of the array has been processed, the total population count is

$$8\text{pop}(\textit{eights}) + 4\text{pop}(\textit{fours}) + 2\text{pop}(\textit{twos}) + \text{pop}(\textit{ones}).$$

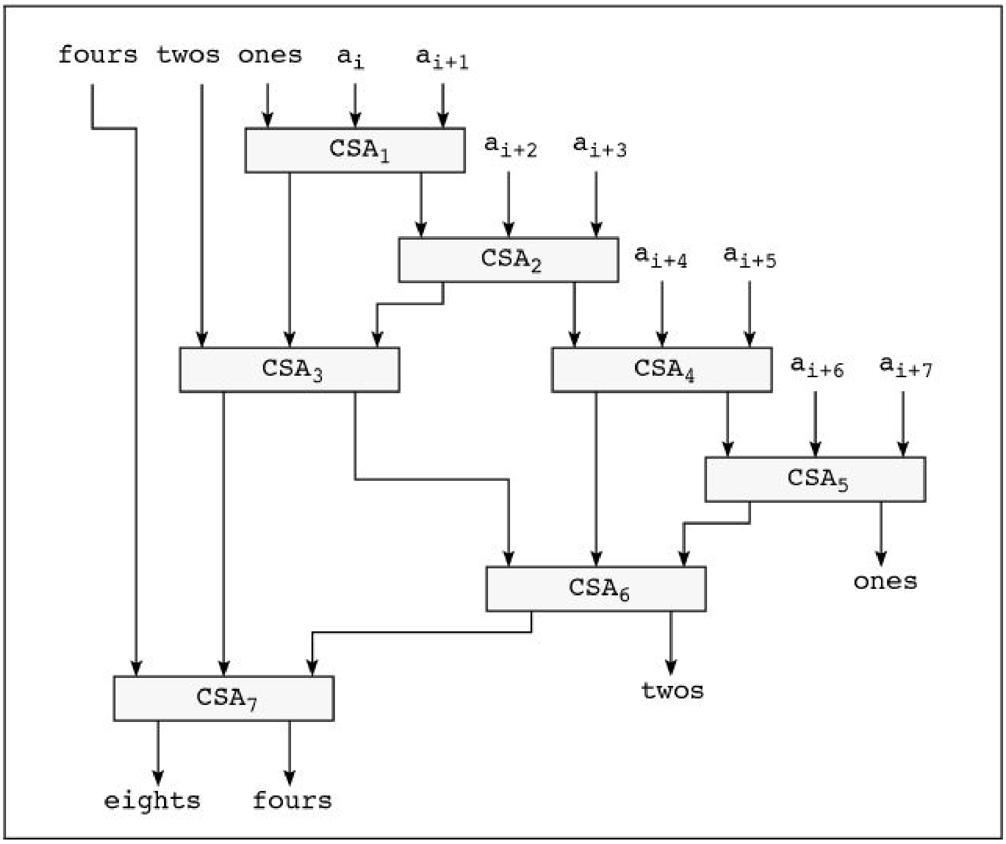


FIGURE 5–8. A circuit for the array population count.

The code is shown in Figure 5–9, which uses the CSA macro defined in Figure 5–7. The numbering of the CSA blocks in Figure 5–8 corresponds to the order of the CSA macro calls in Figure 5–9. The execution time of the loop, exclusive of array loads and loop control, is $(7n_c + n_p + 1)/8 = 6.375$ instructions per word of the array.

```
int popArray(unsigned A[], int n) {

   int tot, i;
   unsigned ones, twos, twosA, twosB,
      fours, foursA, foursB, eights;

   tot = 0;                     // Initialize.
   fours = twos = ones = 0;

   for (i = 0; i <= n - 8; i = i + 8) {
      CSA(twosA, ones, ones, A[i], A[i+1])
      CSA(twosB, ones, ones, A[i+2], A[i+3])
      CSA(foursA, twos, twos, twosA, twosB)
      CSA(twosA, ones, ones, A[i+4], A[i+5])
      CSA(twosB, ones, ones, A[i+6], A[i+7])
      CSA(foursB, twos, twos, twosA, twosB)
      CSA(eights, fours, fours, foursA, foursB)
      tot = tot + pop(eights);
   }
   tot = 8*tot + 4*pop(fours + 2*pop(twos) + pop(ones);

   for (i = i; i < n; i++)        // Simply add in the last
      tot = tot + pop(A[i]);     // 0 to 7 elements.
   return tot;
}
```

FIGURE 5–9. Array population count, processing elements in groups of eight.

The CSAs can be connected in many arrangements other than that shown in Figure 5–8. For example, increased parallelism might result from feeding the first three array elements into one CSA, and the next three into a second CSA, which allows the instructions of these two CSAs to execute in parallel. One might also be able to permute the three input operands of the CSA macros for increased parallelism. With the plan shown in Figure 5–8, one can easily see how to use only the first three CSAs to construct a program that processes array elements in groups of four, and also how to expand it to construct programs that process array elements in groups of 16 or more. The plan shown also spreads out the loads somewhat, which would be advantageous for a machine that has a relatively low limit on the number of loads that can be outstanding at any one time.

The plan of Figure 5–8 can be generalized so that very few word population counts are done. To sketch how this program might be constructed, it needs an array of $m\times 2$ words to hold two of each of the variables we have called *ones, twos, fours*, and so forth. For an array of size n, choosing $m \geq \log_2(n + 1) + 1$ is sufficient ($m = 31$ is sufficient for any size array that can be held in a machine with a 32-bit byte-addressed space). A byte array of size m is also needed to keep track of how many (0, 1, or 2) values are currently in each row of the $m\times 2$ array. The program processes array elements in groups of two. For each group, the CSA is invoked to compress those two array elements with a saved value of *ones*, which is most conveniently kept in the [0,0] position of the $m\times 2$ array. In an inner loop, the resulting *twos* is saved in the array, by scanning down (usually not far at all) to find a row with fewer than two items. If the *twos* row is full, its two values are combined with *twos* (using the CSA). The *twos* output is put in the array, resetting its row count to 1. The scan continues with the *fours* output to find a place to put it, and so forth.

After completing the pass over the input array, the program next makes a pass over the (much shorter) $m \times 2$ array, compressing all full rows, so that all rows contain only one significant value. Lastly, the program invokes the word-level population count operation on the first element of each row until a row with a zero count is encountered, computing the total array population count as

$$\text{pop(row 0)} + 2\text{pop(row 1)} + 4\text{pop(row 2)} + \ldots.$$

The value suggested above for m ensures that the last row will have a zero count, which can be used to terminate the scans.

The resulting program executes exactly $\log_2(n + 3)$ word population counts. Unfortunately it is not practical, because the housekeeping steps for loading from and storing into the intermediate result arrays outweigh the computational instructions that are saved. An experimental program (without trying too hard to optimize it) ran in about 29 instructions per array word (counting all instructions in the loop). This is significantly worse than the naive method.

Table 5–1 summarizes the number of instructions executed by this plan for various group sizes. The values in the middle two columns ignore loads and loop control. The fourth column gives the total loop instruction execution count, per word of the input array, produced by a compiler for the basic RISC machine (which does not have indexed loads).

TABLE 5–1. INSTRUCTIONS PER WORD FOR THE ARRAY POPULATION COUNT

	Instructions Exclusive of Loads and Loop Control		All Instructions in Loop (compiler output)
Program	**Formula**	**For $n_c = 5$, $n_p = 15$**	
Naive method	$n_p + 1$	16	21
Groups of 2	$(n_c + n_p + 1)/2$	10.5	14
Groups of 4	$(3n_c + n_p + 1)/4$	7.75	10
Groups of 8	$(7n_c + n_p + 1)/8$	6.38	8
Groups of 16	$(15n_c + n_p + 1)/16$	5.69	7
Groups of 32	$(31n_c + n_p + 1)/32$	5.34	6.5
Groups of 2^n	$n_c + \dfrac{n_p - n_c + 1}{2^n}$	$5 + \dfrac{11}{2^n}$	–

For small arrays, there are better plans than that of Figure 5–8. For example, for an array of seven words, the plan of Figure 5–10 is quite efficient [Seal1]. It executes in $4n_c + 3n_p + 4 = 69$ instructions, or 9.86 instructions per word. Similar plans exist that apply to arrays of size $2^k - 1$ words for any positive integer k. The plan for 15 words executes in $11n_c + 4n_p + 6 = 121$ instructions, or 8.07 instructions per word.

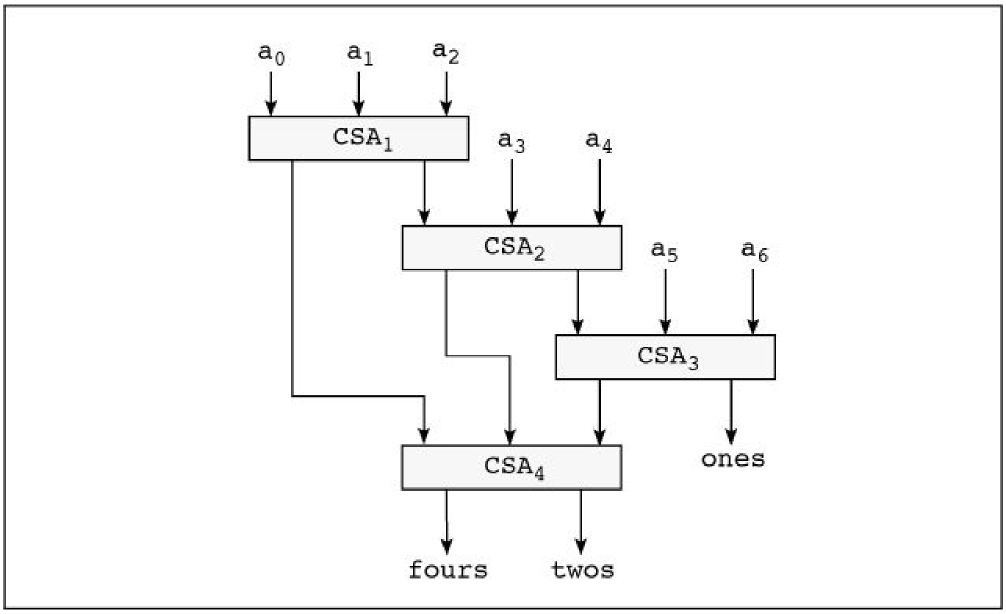


FIGURE 5–10. A circuit for the total population count of seven words.

Applications

An application of the *population count* function is in computing the “Hamming distance” between two bit vectors, a concept from the theory of error-correcting codes. The Hamming distance is simply the number of places where the vectors differ; that is,

$$\text{dist}(x, y) = \text{pop}(x \quad y).$$

See, for example, the chapter on error-correcting codes in [Dewd].

Another application is to allow reasonably fast direct-indexed access to a moderately sparse array *A* that is represented in a certain compact way. In the compact representation, only the defined, or nonzero, elements of the array are stored. There is an auxiliary bit string array *bits* of 32-bit words, which has a 1-bit for each index *i* for which *A*[*i*] is defined. As a speedup device, there is also an array of words *bitsum* such that *bitsum*[*j*] is the total number of 1-bits in all the words of *bits* that precede entry *j*. This is illustrated below for an array in which elements 0, 2, 32, 47, 48, and 95 are defined.

bits	*bitsum*	*data*
0x00000005	0	$A[0]$
0x00018001	2	$A[2]$
0x80000000	5	$A[32]$
		$A[47]$
		$A[48]$
		$A[95]$

Given an index *i*, $0 \le i \le 95$, the corresponding index *sparse_i* into the data array is given by the number of 1-bits in array *bits* that precede the bit corresponding to *i*. This can be calculated as follows:

```
j = i >> 5;                    // j = i/32.
k = i & 31;                    // k = rem(i, 32);
mask = 1 << k;                 // A "1" at position k.
if ((bits[j] & mask) == 0) goto no_such_element;
mask = mask - 1;               // l's to right of k.
sparse_i = bitsum[j] + pop(bits[j] & mask);
```

The cost of this representation is two bits per element of the full array.

The population function can be used to generate binomially distributed random integers. To generate an integer drawn from a population given by BINOMIAL(*t, p*) where *t* is the number of trials and $p = 1/2$, generate *t* random bits and count the number of 1's in the *t* bits. This can be generalized to probabilities *p* other than 1/2; see for example [Knu2, sec. 3.4.1, prob. 27].

Still another application of the population function is in computing the number of trailing 0's in a word (see "Counting Trailing 0's" on page 107).

According to computer folklore, the population count function is important to the National Security Agency. No one (outside of NSA) seems to know just what they use it for, but it may be in cryptography work or in searching huge amounts of material.

5–2 Parity

The "parity" of a string refers to whether it contains an odd or an even number of 1-bits. The string has "odd parity" if it contains an odd number of 1-bits; otherwise, it has "even parity."

Computing the Parity of a Word

Here we mean to produce a 1 if a word *x* has odd parity, and a 0 if it has even parity. This is the sum, modulo 2, of the bits of *x*—that is, the *exclusive or* of all the bits of *x*.

One way to compute this is to compute pop(*x*); the parity is the rightmost bit of the result. This is fine if you have the *population count* instruction, but if not, there are better ways than using the code for pop(*x*).

A rather direct method is to compute

$$y \leftarrow \bigoplus_{i=0}^{n-1} (x \overset{u}{\gg} i),$$

where *n* is the word size, and then the parity of *x* is given by the rightmost bit of *y*. (Here denotes *exclusive or*, but for this formula ordinary addition could be used.)

The parity can be computed much more quickly, for moderately large *n*, as follows (illustrated for *n* = 32; the shifts can be signed or unsigned):

```
y = x ^ (x >> 1);
y = y ^ (y >> 2);
y = y ^ (y >> 4);                                        (3)
y = y ^ (y >> 8);
y = y ^ (y >>16);
```

This executes in ten instructions, as compared to 62 for the first method, even if the implied loop is completely unrolled. Again, the parity bit is the rightmost bit of *y*. In fact, with either of these, if the shifts are unsigned, then bit *i* of *y* gives the parity of the bits of *x* at and to the left of *i*. Furthermore, because *exclusive or* is its own inverse, x_i x_j is the parity of bits *i* – 1 through *j*, for $i \geq j$.

This is an example of the “parallel prefix,” or “scan” operation, which has applications in parallel computing [KRS; HS]. Given a sufficient number of processors, it can convert certain seemingly serial processes from $O(n)$ to $O(\log_2 n)$ time. For example, if you have an array of words and you wish to compute the *exclusive or* scan operation on the entire array of bits, you can first use (3) on the entire array, and then continue with shifts of 32 bits, 64 bits, and so on, doing *exclusive or*’s on the words of the array. This takes more elementary (word length) *exclusive or* operations than a simple left-to-right process, and hence it is not a good idea for a uniprocessor. But on a parallel computer with a sufficient number of processors, it can do the job in $O(\log_2 n)$ rather than $O(n)$ time (where *n* is the number of words in the array).

A direct application of (3) is the conversion of a Gray coded integer to binary (see page 312).

If the code (3) is changed to use left shifts, the parity of the whole word *x* winds up in the leftmost bit position, and bit *i* of *y* gives the parity of the bits of *x* at and to the *right* of position *i*. This is called the “parallel suffix” operation, because each bit is a function of itself and the bits that follow it.

If *rotate shift*’s are used, the result is a word of all 1’s if the parity of *x* is odd, and of all 0’s if even.

The five assignments in (3) can be done in any order (provided variable `x` is used in the first one). If they are done in reverse order, and if you are interested only in getting the parity in the low-order bit of `y`, then the last two lines:

```
y = y ^(y >> 2);
y = y ^(y >> 1);
```

can be replaced with [Huef]

```
y = 0x6996 >> (y & 0xF);
```

This is an "in-register table lookup" operation. On the basic RISC it saves one instruction, or two if the load of the constant is not counted. The low-order bit of `y` has the original word's parity, but the other bits of `y` do not contain anything useful.

The following method executes in nine instructions and computes the parity of x as the integer 0 or 1 (the shifts are unsigned).

```
x = x ^ (x >> 1);
x = (x ^ (x >> 2)) & 0x11111111;
x = x*0x11111111;
p = (x >> 28) & 1;
```

After the second statement above, each hex digit of x is 0 or 1, according to the parity of the bits in that hex digit. The *multiply* adds these digits, putting the sum in the high-order hex digit. There can be no carry out of any hex column during the *add* part of the multiply, because the maximum sum of a column is 8.

The *multiply* and *shift* could be replaced by an instruction to compute the remainder after dividing x by 15, giving a (slow) solution in eight instructions, if the machine has *remainder immediate*.

On a 64-bit machine, the above code employing multiplication gives the correct result after making the obvious changes (expand the hex constants to 16 nibbles, each with value 1, and change the final shift amount from 28 to 60). In this case, the maximum sum in any 4-bit column of the partial products, other than the most significant column, is 15, so again no overflow occurs that affects the result in the most significant column. On the other hand, the variation that computes the remainder upon division by 15 does *not* work on a 64-bit machine, because the remainder is the sum of the nibbles modulo 15, and the sum may be as high as 16.

Adding a Parity Bit to a 7-Bit Quantity

Item 167 in [HAK] contains a novel expression for putting even parity on a 7-bit quantity that is right-adjusted and isolated in a register. By this we mean to set the bit to the left of the seven bits, to make an 8-bit quantity with even parity. Their code is for a 36-bit machine, but it works on a 32-bit machine as well.

$$\text{modu}((x * \mathbf{0x10204081}) \,\&\, \mathbf{0x888888FF}, \mathbf{1920})$$

Here, modu(a, b) denotes the remainder of a upon division by b, with the arguments and result interpreted as unsigned integers, "*" denotes multiplication modulo 2^{32}, and the constant 1920 is $15 \cdot 2^7$. Actually, this computes the sum of the bits of x, and places the sum just to the left of the seven bits comprising x. For example, the expression maps **0x0000007F** to **0x000003FF**, and **0x00000055** to **0x00000255**.

Another ingenious formula from [HAK] is the following, which puts odd parity on a 7-bit integer:

$$\text{modu}((x * \mathbf{0x00204081}) \mid \mathbf{0x3DB6DB00}, \mathbf{1152}),$$

where $1152 = 9 \cdot 2^7$. To understand this, it helps to know that the powers of 8 are ±1 modulo 9. If the **0x3DB6DB00** is changed to **0xBDB6DB00**, this formula applies even parity.

These methods are not practical on today's machines, because memory is cheap but division is still slow. Most programmers would compute these functions with a simple table lookup.

Applications

The parity operation is widely used to calculate a check bit to append to data. It is also useful in multiplying bit matrices in GF(2) (in which the *add* operation is *exclusive or*).

5–3 Counting Leading 0's

There are several simple ways to count leading 0's with a binary search technique. Below is a model that has several variations. It executes in 20 to 29 instructions on the basic RISC. The comparisons are "logical" (unsigned integers).

```
if (x == 0) return(32);
n = 0;
if (x <= 0x0000FFFF) {n = n +16; x = x <<16;}
if (x <= 0x00FFFFFF) {n = n + 8; x = x << 8;}
if (x <= 0x0FFFFFFF) {n = n + 4; x = x << 4;}
if (x <= 0x3FFFFFFF) {n = n + 2; x = x << 2;}
if (x <= 0x7FFFFFFF) {n = n + 1;}
return n;
```

One variation is to replace the comparisons with *and*'s:

```
if ((x & 0xFFFF0000) == 0) {n = n +16; x = x <<16;}
if ((x & 0xFF000000) == 0) {n = n + 8; x = x << 8}
...
```

Another variation, which avoids large immediate values, is to use *shift right* instructions.

The last `if` statement is simply adding 1 to `n` if the high-order bit of `x` is 0, so an alternative, which saves a branch instruction, is:

```
n = n + 1 - (x >> 31);
```

The "+ 1" in this assignment can be omitted if `n` is initialized to 1 rather than to 0. These observations lead to the algorithm (12 to 20 instructions on the basic RISC) shown in Figure 5–11. A further improvement is possible for the case in which `x` begins with a 1-bit: change the first line to

```
if ((int)x <= 0) return (~x >> 26) & 32;
```

```
int nlz(unsigned x) {
   int n;

   if (x == 0) return(32);
   n = 1;
   if ((x >> 16) == 0) {n = n +16; x = x <<16;}
   if ((x >> 24) == 0) {n = n + 8; x = x << 8;}
   if ((x >> 28) == 0) {n = n + 4; x = x << 4;}
   if ((x >> 30) == 0) {n = n + 2; x = x << 2;}
   n = n - (x >> 31);
   return n;
}
```

FIGURE 5–11. *Number of leading zeros*, **binary search.**

Figure 5–12 illustrates a sort of reversal of the above. It requires fewer operations the more leading 0's there are, and avoids large immediate values and large shift amounts. It executes in 12 to 20 instructions on the basic RISC.

```
int nlz(unsigned x)  {
   unsigned y;
   int n;

   n = 32;
   y = x >>16; if (y != 0) {n = n -16; x = y;}
   y = x >> 8; if (y != 0) {n = n - 8; x = y;}
   y = x >> 4; if (y != 0) {n = n - 4; x = y;}
   y = x >> 2; if (y != 0) {n = n - 2; x = y;}
   y = x >> 1; if (y != 0) return n - 2;
   return n - x;
}
```

FIGURE 5–12. *Number of leading zeros*, **binary search, counting down.**

This algorithm is amenable to a "table assist": the last four executable lines can be replaced by

```
   static char table[256] = {0,1,2,2,3,3,3,3,4,4,...,8);
   return n - table[x];
```

Many algorithms can be aided by table lookup, but this will not often be mentioned here.

For compactness, this and the preceding algorithms in this section can be coded as loops. For example, the algorithm of Figure 5–12 becomes the algorithm shown in Figure 5–13. This executes in 23 to 33 basic RISC instructions, ten of which are conditional branches.

```
   int nlz(unsigned x) {
      unsigned y;
      int n, c;

      n = 32;
      c = 16;
      do {
         y = x >> c; if (y != 0) {n = n - c; x = y;}
         c = c >> 1;
      } while (c != 0);
      return n - x;
   }
```

FIGURE 5–13. *Number of leading zeros*, **binary search, coded as a loop.**

One can, of course, simply shift left one place at a time, counting, until the sign bit is on; or shift right one place at a time until the word is all 0. These algorithms are compact and work well if the number of leading 0's is expected to be small or large, respectively. One can combine the methods, as shown in Figure 5–14. We mention this because the technique of merging two algorithms and choosing the result of whichever one stops first is more generally applicable. It leads to code that runs fast on superscalar machines, because of the proximity of independent instructions. (These machines can execute two or more instructions simultaneously, provided they are independent.)

```
int nlz(int x) {
   int y, n;

   n = 0;
   y = x;
L: if (x < 0) return n;
   if (y == 0) return 32 - n;
   n = n + 1;
   x = x << 1;
   y = y >> 1;
   goto L;
}
```

FIGURE 5–14. *Number of leading zeros*, **working both ends at the same time.**

On the basic RISC, this executes in min(3 + 6nlz(x), 5 + 6(32 – nlz(x))) instructions, or 99 worst case. One can imagine a superscalar machine executing the entire loop body in one cycle if the comparison results are obtained as a by-product of the shifts, or in two cycles otherwise, plus the branch overhead.

It is straightforward to convert either of the algorithms of Figure 5–11 or Figure 5–12 to a branch-free counterpart. Figure 5–15 shows a version that does the job in 28 basic RISC instructions.

```
int nlz(unsigned x) {
   int y, m, n;

   y = -(x >> 16);        // If left half of x is 0,
   m = (y >> 16) & 16;    // set n = 16. If left half
   n = 16 - m;            // is nonzero, set n = 0 and
   x = x >> m;            // shift x right 16.
                          // Now x is of the form 0000xxxx.
   y = x - 0x100;         // If positions 8–15 are 0,
   m = (y >> 16) & 8;     // add 8 to n and shift x left 8.
   n = n + m;
   x = x << m;

   y = x - 0x1000;        // If positions 12–15 are 0,
   m = (y >> 16) & 4;     // add 4 to n and shift x left 4.
   n = n + m;
   x = x << m;

   y = x - 0x4000;        // If positions 14–15 are 0,
   m = (y >> 16) & 2;     // add 2 to n and shift x left 2.
   n = n + m;
   x = x << m;

   y = x >> 14;           // Set y = 0, 1, 2, or 3.
   m = y & ~(y >> 1);     // Set m = 0, 1, 2, or 2 resp.
   return n + 2 - m;
}
```

FIGURE 5–15. *Number of leading zeros*, **branch-free binary search.**

If your machine has the *population count* instruction, a good way to compute the *number of leading zeros* function is given in Figure 5–16. The five assignments to `x` can be reversed, or, in fact, done in any order. This is branch-free and takes 11 instructions. Even if *population count* is not available, this algorithm may be useful. Using the 21-instruction code for counting 1-bits given in Figure 5–2 on page 82, it executes in 32 branch-free basic RISC instructions.

```
int nlz(unsigned x) {
   int pop(unsigned x);

   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >>16);
   return pop(~x);
}
```

FIGURE 5–16. *Number of leading zeros*, **right-propagate and count 1-bits.**

Robert Harley [Harley] devised an algorithm for nlz(*x*) that is very similar to Seal's algorithm for ntz(*x*) (see Figure 5–25 on page 111). Harley's method propagates the most significant 1-bit to the right using *shift*'s and *or*'s, and multiplies modulo 2^{32} by a special constant, producing a product whose high-order six bits uniquely identify the number of leading 0's in *x*. It then does a *shift right* and a table lookup (indexed load) to translate the six-bit identifier to the actual number of leading 0's. As shown in Figure 5–17, it consists of 14 instructions, including a *multiply*, plus an indexed load. Table entries shown as `u` are unused.

```
   int nlz(unsigned x) {

      static char table[64] =
        {32,31, u,16, u,30, 3, u,  15, u, u, u,29,10, 2, u,
          u, u,12,14,21, u,19, u,   u,28, u,25, u, 9, 1, u,
         17, u, 4, u, u, u,11, u,  13,22,20, u,26, u, u,18,
          5, u, u,23, u,27, u, 6,   u,24, 7, u, 8, u, 0, u};

      x = x | (x >> 1);     // Propagate leftmost
      x = x | (x >> 2);     // 1-bit to the right.
      x = x | (x >> 4);
      x = x | (x >> 8);
      x = x | (x >>16);
      x = x*0x06EB14F9;     // Multiplier is 7*255**3.
      return table[x >> 26];
   }
```

FIGURE 5–17. *Number of leading zeros*, **Harley's algorithm.**

The multiplier is $7\cdot255^3$, so the multiplication can be done as shown below. In this form, the function consists of 19 elementary instructions, plus an indexed load.

```
   x = (x << 3) - x;     // Multiply by 7.
   x = (x    8) - x;     // Multiply by 255.
```

```
         <<
   x = (x << 8) - x;       // Again.
   x = (x << 8) - x;       // Again.
```

There are many multipliers that have the desired uniqueness property and whose factors are all of the form $2^k \pm 1$. The smallest is 0x045BCED1 = 17 · 65· 129 ·513. There are no such multipliers consisting of three factors if the table size is 64 or 128 entries. If the table size is 256 entries, however, there are a number of such multipliers. The smallest is 0x01033CBF = 65·255·1025 (using this would save two instructions at the expense of a larger table).

Julius Goryavsky [Gor] has found several variations of Harley's algorithm that reduce the table size at the expense of a few instructions, or have improved parallelism, or have other desirable properties. One, shown in Figure 5–18, is a clear winner if the multiplication is done with shifts and adds. The code changes only the table and the lines that contain the *shift right* of 16 and the following *multiply* in Figure 5–17. If the machine has *and not*, this saves two instructions because the multiplier can be factored as 511·2047 · 16383 (mod 2^{32}), which can be done in six elementary instructions rather than eight. If the machine does not have *and not*, it saves one instruction.

```
...
static char table[64] =
  {32,20,19, u, u,18, u, 7,  10,17, u, u,14, u, 6, u,
    u, 9, u,16, u, u, 1,26,   u,13, u, u,24, 5, u, u,
    u,21, u, 8,11, u,15, u,   u, u, u, 2,27, 0,25, u,
   22, u,12, u, u, 3,28, u,  23, u, 4,29, u, u,30,31};
...
x = x & ~(x >> 16);
x = x*0xFD7049FF;
...
```

FIGURE 5–18. *Number of leading zeros*, **Goryavsky's variation of Harley's algorithm.**

Floating-Point Methods

The floating-point post-normalization facilities can be used to count leading zeros. It works out quite well with IEEE-format floating-point numbers. The idea is to convert the given unsigned integer to double-precision floating-point, extract the exponent, and subtract it from a constant. Figure 5–19 illustrates a complete procedure for this.

```
int nlz(unsigned k) {
   union {
      unsigned asInt[2];
      doubleasDouble;
   };
   int n;

   asDouble = (double)k + 0.5;
   n = 1054 - (as!nt[LE] >> 20);
   return n;
}
```

FIGURE 5–19. *Number of leading zeros*, **using IEEE floating-point.**

The code uses the C++ “anonymous union” to overlay an integer with a double-precision floating-point quantity. Variable LE must be 1 for execution on a little-endian machine, and 0 for big-endian. The addition of 0.5, or some other small number, is necessary for the method to work when k = 0.

We will not attempt to assess the execution time of this code, because machines differ so much in their floating-point capabilities. For example, many machines have their floating-point registers separate from the integer registers, and on such machines data transfers through memory may be required to convert an integer to floating-point and then move the result to an integer register.

The code of Figure 5–19 is not valid C or C++ according to the ANSI standard, because it refers to the same memory locations as two different types. Thus, one cannot be sure it will work on a particular machine and compiler. It does work with IBM’s XLC compiler on AIX, and with the GCC compiler on AIX and on Windows 2000 and XP, at all optimization levels (as of this writing, anyway). If the code is altered to do the overlay defining with something like

```
xx = (double)k + 0.5;
n = 1054 - (*((unsigned *)&xx + LE) >> 20);
```

it does *not* work on these systems with optimization turned on. This code, incidentally, violates a second ANSI standard, namely, that pointer arithmetic can be performed only on pointers to array elements [Cohen]. The failure, however, is due to the first violation, involving overlay defining.

In spite of the flakiness of this code,[2] three variations are given below.

```
asDouble = (double)k;
n = 1054 - (asInt[LE] >> 20);
n = (n & 31) + (n >> 9);

k = k & ~(k >> 1);
asFloat = (float)k + 0.5f;
n = 158 - (asInt >> 23);

k = k & ~(k >> 1);
asFloat = (float)k;
n = 158 - (asInt >> 23);
n = (n & 31) + (n >> 6);
```

In the first variation, the problem with k = 0 is fixed not by a floating-point addition of 0.5, but by integer arithmetic on the result n (which would be 1054, or 0x41E, if the correction were not done).

The next two variations use single-precision floating-point, with the “anonymous union” changed in an obvious way. Here there is a new problem: Rounding can throw off the result when the rounding mode is either round to nearest (almost universally used) or round toward $+\infty$. For round to nearest mode, the rounding problem occurs for k in the ranges hexadecimal FFFFFF80 to FFFFFFFF, 7FFFFFC0 to 7FFFFFFF, 3FFFFFE0 to 3FFFFFFF, and so on. In rounding, an add of 1 carries all the way to the left, changing the position of the most significant 1-bit. The correction steps used above clear the bit to the right of the most significant 1-bit, blocking the carry. If k is a 64-bit quantity, this correction is also needed for the code of Figure 5–19 and for the first of the three variations given above.

The GNU C/C++ compiler has a unique feature that allows coding any of these schemes as a macro, giving in-line code for the function references [Stall]. This feature allows statements, including declarations, to be inserted in code where an expression is called for. The sequence of statements would usually end with an expression, which is taken to be the value of the construction. Such a macro definition is shown below, for the first single-precision variation. (In C, it is customary to use uppercase for macro names.)

```
#define NLZ(kp) \
   ({union {unsigned _asInt; float _asFloat;}; \
     unsigned _k = (kp), _kk = _k & ~(_k >> 1); \
     _asFloat = (float)_kk + 0.5f; \
     158 - (_asInt >> 23);})
```

The underscores are used to avoid name conflicts with parameter `kp;` presumably, user-defined names do not begin with underscores.

Comparing the Number of Leading Zeros of Two Words

There is a simple way to determine which of two words *x* and *y* has the larger number of leading zeros [Knu5] without actually computing nlz(*x*) or nlz(*y*). The methods are shown in the equivalences below. The three relations not shown are, of course, obtained by complementing the sense of the comparison on the right.

$$\begin{aligned}
\text{nlz}(x) = \text{nlz}(y) &\quad \text{if and only if} \quad (x \oplus y) \overset{u}{\leq} (x \,\&\, y) \\
\text{nlz}(x) < \text{nlz}(y) &\quad \text{if and only if} \quad (x \,\&\, \neg y) \overset{u}{>} y \\
\text{nlz}(x) \leq \text{nlz}(y) &\quad \text{if and only if} \quad (y \,\&\, \neg x) \overset{u}{\leq} x
\end{aligned}$$

Relation to the Log Function

The “nlz” function is, essentially, the “integer log base 2” function. For unsigned $x \neq \mathbf{0}$,

$$\begin{aligned}
\lfloor \log_2(x) \rfloor &= 31 - \text{nlz}(x), \text{ and} \\
\lceil \log_2(x) \rceil &= 32 - \text{nlz}(x - 1).
\end{aligned}$$

See also Section 11–4, “Integer Logarithm,” on page 291.

Another closely related function is *bitsize*, the number of bits required to represent its argument as a signed quantity in two’s-complement form. We take its definition to be

$$\text{bitsize}(x) = \begin{cases}
1, & x = -1 \text{ or } 0, \\
2, & x = -2 \text{ or } 1, \\
3, & -4 \leq x \leq -3 \text{ or } 2 \leq x \leq 3, \\
4, & -8 \leq x \leq -5 \text{ or } 4 \leq x \leq 7, \\
\dots & \dots \\
32, & -2^{31} \leq x \leq -2^{30} + 1 \text{ or } 2^{30} \leq x \leq 2^{31} - 1.
\end{cases}$$

From this definition, bitsize(x) = bitsize($-x-1$). But $-x-\mathbf{1} = \neg x$, so an algorithm for bitsize is (where the shift is signed)

```
x = x ^ (x >> 31);          // If (x < 0) x = -x - 1;
return 33 - nlz(x);
```

An alternative, which is the same function as bitsize(x) except it gives the result 0 for $x = 0$, is

```
32 - nlz(x ^ (x << 1))
```

Applications

Two important applications of the *number of leading zeros* function are in simulating floating-point arithmetic operations and in various division algorithms (see Figure 9–1 on page 185 and Figure 9–3 on page 196). The instruction seems to have a miscellany of other uses.

It can be used to get the "$x = y$" predicate in only three instructions (see "Comparison Predicates" on page 23), and as an aid in computing certain elementary functions (see pages 281, 284, 290, and 294).

A novel application is to generate exponentially distributed random integers by generating uniformly distributed random integers and taking nlz of the result [GLS1]. The result is 0 with probability 1/2, 1 with probability 1/4, 2 with probability 1/8, and so on. Another application is as an aid in searching a word for a consecutive string of 1-bits (or 0-bits) of a certain length, a process that is used in some disk block allocation algorithms. For these last two applications, the *number of trailing zeros* function could also be used.

5–4 Counting Trailing 0's

If the *number of leading zeros* instruction is available, then the best way to count trailing 0's is, most likely, to convert it to a count *leading* 0's problem:

$$\mathbf{32} - \text{nlz}(\neg x \& (x - \mathbf{1})).$$

If *population count* is available, a slightly better method is to form a mask that identifies the trailing 0's, and count the 1-bits in it [Hay2], such as

$$\text{pop}(\neg x \,\&\, (x - \mathbf{1})), \text{ and}$$
$$\mathbf{32} - \text{pop}(x \mid -x).$$

Variations exist using other expressions for forming a mask that identifies the trailing zeros of x, such as those given in Section 2–1, "Manipulating Rightmost Bits," on page 11. These methods are also reasonable even if the machine has none of the bit-counting instructions. Using the algorithm for pop(x) given in Figure 5–2 on page 82, the first expression above executes in about 3 + 21 = 24 instructions (branch-free).

Figure 5–20 shows an algorithm that does it directly, in 12 to 20 basic RISC instructions (for $x \neq 0$).

```
int ntz(unsigned x) {
   int n;

   if (x == 0) return(32);
   n = 1;
   if ((x & 0x0000FFFF) == 0) {n = n + 16; x = x >>16;}
   if ((x & 0x000000FF) == 0) {n = n +  8; x = x >> 8;}
   if ((x & 0x0000000F) == 0) {n = n +  4; x = x >> 4;}
   if ((x & 0x00000003) == 0) {n = n +  2; x = x >> 2;}
   return n - (x & 1);
}
```

FIGURE 5–20. *Number of trailing zeros*, **binary search.**

The `n + 16` can be simplified to `17` if that helps, and if the compiler is not smart enough to do that for you (this does not affect the number of instructions as we are counting them).

Figure 5–21 shows a variation that uses smaller immediate values and simpler operations. It executes in 12 to 21 basic RISC instructions. Unlike the above procedure, when the number of trailing 0’s is small, the procedure of Figure 5–21 executes a larger number of instructions, but also a larger number of “fall-through” branches.

```
int ntz(unsigned x) {
   unsigned y;
   int n;

   if (x == 0) return 32;
   n = 31;
   y = x <<16;  if (y != 0) {n = n -16; x = y;}
   y = x << 8;  if (y != 0) {n = n - 8; x = y;}
   y = x << 4;  if (y != 0) {n = n - 4; x = y;}
   y = x << 2;  if (y != 0) {n = n - 2; x = y;}
   y = x << 1;  if (y != 0) {n = n - 1;}
   return n;
}
```

FIGURE 5–21. *Number of trailing zeros*, **smaller immediate values.**

The line just above the `return` statement can alternatively be coded

```
n = n - ((x << 1) >> 31);
```

which saves a branch, but not an instruction.

In terms of number of instructions executed, it is hard to beat the “search tree” [Aus2]. Figure 5–22 illustrates this procedure for an 8-bit argument. This procedure executes in seven instructions for all paths except the last two (return 7 or 8), which require nine. A 32-bit version would execute in 11 to 13 instructions. Unfortunately, for large word sizes, the program is quite large. The 8-bit version above is 12 lines of executable source code and would compile into about 41 instructions. A 32-bit version would be 48 lines and about 164 instructions, and a 64-bit version would be twice that.

```
int ntz(char x) {
   if (x & 15) {
      if (x & 3) {
         if (x & 1) return 0;
         else return 1;
      }
      else if (x & 4) return 2;
      else return 3;
   }
   else if (x & 0x30) {
      if (x & 0x10) return 4;
      else return 5;
   }
   else if (x & 0x40) return 6;
   else if (x) return 7;
   else return 8;
}
```

FIGURE 5–22. *Number of trailing zeros*, **binary search tree.**

If the number of trailing 0's is expected to be small or large, then the simple loops shown in Figure 5–23 are quite fast. The algorithm on the left executes in 5 + 3ntz(x), and that on the right in 3 + 3(32 – ntz(x)) basic RISC instructions.

```
int ntz(unsigned x) {
   int n;

   x = ~x & (x - 1);
   n = 0;                           // n = 32;
   while (x != 0) {                 // while (x != 0) {
      n = n + 1;                    //    n = n - 1;
      x = x >> 1;                   //    x = x + x;
   }                                // }
   return n;                        // return n;
}
```

FIGURE 5–23. *Number of trailing zeros*, **simple counting loops.**

Dean Gaudet [Gaud] devised an algorithm that is interesting because with the right instructions it is branch-free, load-free (does not use table lookup), and has lots of parallelism. It is shown in Figure 5–24.

```
int ntz(unsigned x) {
   unsigned y, bz, b4, b3, b2, b1, b0;

   y = x & -x;                 // Isolate rightmost 1-bit.
   bz = y ? 0 : 1;             // 1 if y = 0.
   b4 = (y & 0x0000FFFF) ? 0 : 16;
   b3 = (y & 0x00FF00FF) ? 0 :  8;
   b2 = (y & 0x0F0F0F0F) ? 0 :  4;
   b1 = (y & 0x33333333) ? 0 :  2;
   b0 = (y & 0x55555555) ? 0 :  1;
   return bz + b4 + b3 + b2 + bl +b0;
}
```

FIGURE 5–24. *Number of trailing zeros*, **Gaudet's algorithm.**

As shown, the code uses the C "conditional expression" in six places. This construct has the form `a?b:c`. Its value is `b` if `a` is **true** (nonzero), and `c` if `a` is **false** (zero). Although a conditional expression must, in general, be compiled into compares and branches, for the simple cases in Figure 5–24 branching can be avoided if the machine has a *compare for equality to zero* instruction that sets a target register to 1 if the operand is 0, and to 0 if the operand is nonzero. Branching can also be avoided by using *conditional move* instructions. Using *compare*, the assignment to `b3` can be compiled into five instructions on the basic RISC: two to generate the hex constant, an *and*, the *compare*, and a *shift left* of 3. (The first, second, and last conditional expressions require one, three, and four instructions, respectively.)

The code can be compiled into a total of 30 instructions. All six lines with the conditional expressions can run in parallel. On a machine with a sufficient degree of parallelism, it executes in ten cycles. Present machines don't have that much parallelism, so as a practical matter it might help to change the first two uses of `y` in the program to `x`. This permits the first three executable statements to run in parallel.

David Seal [Seal2] devised an algorithm for computing ntz(x) that is based on the idea of compressing the 2^{32} possible values of x to a small dense set of integers and doing a table lookup. He uses the expression $x \& -x$ to reduce the number of possible values to a small number. The value of this expression is a word that contains a single 1-bit at the position of the least significant 1-bit in x, or is 0 if $x = 0$. Thus, $x \& -x$ has only 33 possible values. But they are not dense; they range from 0 to 2^{31}.

To produce a dense set of 33 integers that uniquely identify the 33 values of $x \& -x$, Seal found a certain constant which, when multiplied by $x \& -x$, produces the identifying value in the high-order six bits of the low-order half of the product of the constant and $x \& -x$. Since $x \& -x$ is an integral power of 2 or is 0, the multiplication amounts to a left shift of the constant, or it is a multiplication by 0. Using only the high-order five bits is not sufficient, because 33 distinct values are needed.

The code is shown in Figure 5–25, where table entries shown as `u` are unused.

```
int ntz(unsigned x) {

   static char table[64] =
     {32, 0, 1,12, 2, 6, u,13,   3, u, 7, u, u, u, u,14,
      10, 4, u, u, 8, u, u,25,   u, u, u, u, u,21,27,15,
      31,11, 5, u, u, u, u, u,   9, u, u,24, u, u,20,26,
      30, u, u, u, u,23, u,19,  29, u,22,18,28,17,16, u};

   x = (x & -x)*0x0450FBAF;
   return table[x >> 26];
}
```

FIGURE 5–25. *Number of trailing zeros*, **Seal's algorithm.**

As an example, if `x` is an odd multiple of 16, then `x & -x` = 16, so the multiplication is simply a left shift of four positions. The high-order six bits of the low-order half of the product are then binary 010001, or 17 decimal. The table translates 17 to 4, which is the correct number of trailing 0's for an odd multiple of 16.

There are thousands of constants that have the necessary uniqueness property. The smallest is 0x0431472F, and the largest is 0xFDE75C6D. Seal chose a constant for which the multiplication can be done with a small number of shifts and adds. Since 0x0450FBAF = 17–65-65535, the multiplication can be done as follows:

```
   x = (x << 4) + x;         // x = x*17.
   x = (x << 6) + x;          // x = x*65.
   x = (x << 16) - x;         // x = x*65535.
```

With this substitution, the code of Figure 5–25 consists of nine elementary instructions, plus an indexed *load*. Seal was interested in the ARM instruction set, which can do a *shift* and *add* in one instruction. On that architecture, the code is six instructions, including the indexed load.

To make the multiplication even easier to do with shifts and adds, one might hope to find a constant of the form $(2^{k_1} \pm 1)(2^{k_2} \pm 1)$ that has the necessary uniqueness property. For a table size of 64, there are no such integers, and there is only one other suitable integer that is a product of three such factors: 0x08A1FBAF = 17 · 65 · 131071. Using a table size of 128 or 256 does not help. However, for a table size of 512 there are four suitable integers of the form $(2^{k_1} \pm 1)(2^{k_2} \pm 1)$; the smallest is 0x0080FF7F = 129 · 65535. We leave it to the reader to determine the table associated with this constant.

There is a variation of Seal's method that is based on de Bruijn cycles [LPR]. These are cyclic sequences over a given alphabet that contain as a subsequence every sequence of the letters of the alphabet of a given length exactly once. For example, a cycle that contains as a subsequence every sequence of {*a, b, c*} of length 2 is *aabacbbcc*. Notice that the sequence *ca* wraps around from the end to the beginning. If the alphabet size is *k* and the length is *n*, there are k^n sequences. For a cycle to contain all of these, it must be of length at least k^n, which would be its length if a different sequence started at each position. It can be shown that there is always a cycle of this minimum possible length that contains all k^n sequences.

For our purposes, the alphabet is {0, 1}, and for dealing with 32-bit words, we are interested in a cycle that contains all 32 sequences 00000, 00001, 00010, ..., 11111. Given such a cycle that begins with at least four 0's, we can compute ntz(*x*) by first reducing *x* to a word that contains a single bit at the position of the least significant bit of *x*, as in Seal's algorithm. Then, by multiplication, we can select a 5-bit field of the de Bruijn cycle, which will be a unique value for each multiplier. This can be mapped to give the number of trailing 0's by a table lookup. The algorithm follows. The de Bruijn cycle used is

```
0000 0100 1101 0111 0110 0101 0001 1111.
```

It is in effect a cycle, because in use it has trailing 0's beyond the 32 bits shown above, which is effectively the same as wrapping to the beginning.

There are 33 possible values of ntz(*x*) and only 32 five-bit subsequences in the de Bruijn cycle. Therefore, two words with different values of ntz(*x*) must map to the same number by the table lookup. The words that conflict are zero and words that end with a 1-bit. To resolve this, the code has a test for 0 and returns 32 in that case. A branch-free way to resolve it, useful if your computer has predicate comparison instructions, is to change the last statement to

```
   return table[x >> 27] + 32*(x == 0);
```

To compare the two algorithms, Seal's does not require the test for zero and it allows the alternative of doing the multiplication with six elementary instructions. The de Bruijn algorithm uses a smaller table. The de Bruijn cycle used in Figure 5–26, discovered by Danny Dubé [Dubé], is a good one because multiplication by it can be done with eight elementary instructions. The constant is 0x04D7651F = (2047 · 5 · 256 + 1) · 31, from which one can see the shifts, adds, and subtracts that do the job.

```
int ntz(unsigned x) {

   static char table[32] =
     { 0, 1, 2,24, 3,19, 6,25,   22, 4,20,10,16, 7,12,26,
      31,23,18, 5,21, 9,15,11,   30,17, 8,14,29,13,28,27};

   if (x == 0) return 32;
   x = (x & -x)*0x04D7651F;
   return table[x >> 27];
}
```

FIGURE 5–26. *Number of trailing zeros* **using a de Bruijn cycle.**

John Reiser [Reiser] observed that there is another way to map the 33 values of the factor `x & -x` in Seal's algorithm to a dense set of unique integers: divide and use the remainder. The smallest divisor that has the necessary uniqueness property is 37. The resulting code is shown in Figure 5–27, where table entries shown as `u` are unused.

```
int ntz(unsigned x) {

   static char table[37] = {32, 0, 1, 26, 2, 23, 27,
                 u, 3, 16, 24, 30, 28, 11, u, 13, 4,
                 7, 17, u, 25, 22, 31, 15, 29, 10, 12,
                 6, u, 21, 14, 9, 5, 20, 8, 19, 18};

   x = (x & -x)%37;
   return table[x];
}
```

FIGURE 5–27. *Number of trailing zeros*, **Reiser's algorithm.**

It is interesting to note that if the numbers x are uniformly distributed, then the average number of trailing 0's is, very nearly, 1.0. To see this, sum the products $p_i n_i$, where p_i is the probability that there are exactly n_i trailing 0's. That is,

$$S \cong \frac{1}{2}\cdot 0+\frac{1}{4}\cdot 1+\frac{1}{8}\cdot 2+\frac{1}{16}\cdot 3+\frac{1}{32}\cdot 4+\frac{1}{64}\cdot 5+\dots$$
$$\cong \sum_{n=0}^{\infty}\frac{n}{2^{n+1}}.$$

To evaluate this sum, consider the following array:

$$\begin{array}{cccccc}
1/4 & 1/8 & 1/16 & 1/32 & 1/64 & \dots \\
 & 1/8 & 1/16 & 1/32 & 1/64 & \dots \\
 & & 1/16 & 1/32 & 1/64 & \dots \\
 & & & 1/32 & 1/64 & \dots \\
 & & & & 1/64 & \dots \\
 & & & & & \dots
\end{array}$$

The sum of each column is a term of the series for S. Hence S is the sum of all the numbers in the array. The sum of the rows are

$$\begin{aligned}
&1/4 + 1/8 + 1/16 + 1/32 + \dots = 1/2\\
&1/8 + 1/16 + 1/32 + 1/64 + \dots = 1/4\\
&1/16 + 1/32 + 1/64 + 1/128 + \dots = 1/8\\
&\dots
\end{aligned}$$

and the sum of these is 1/2 + 1/4 + 1/8 + ... = 1. The absolute convergence of the original series justifies the rearrangement.

Sometimes, a function similar to ntz(x) is wanted, but a 0 argument is a special case, perhaps an error, that should be identified with a value of the function that's easily distinguished from the "normal" values of the function. For example, let us define "the number of factors of 2 in x" to be

$$\text{nfact2}(x) = \begin{cases} \text{ntz}(x), & x \neq 0, \\ -1, & x = 0. \end{cases}$$

This can be calculated from

$$\mathbf{31} - \text{nlz}(x \,\&\, -x).$$

Applications

[GLS1] points out some interesting applications of the *number of trailing zeros* function. It has been called the "ruler function" because it gives the height of a tick mark on a ruler that's divided into halves, quarters, eighths, and so on.

It has an application in R. W. Gosper's loop-detection algorithm, which will now be described in some detail, because it is quite elegant and it does more than might at first seem possible.

Suppose a sequence X_0, X_1, X_2, ... is defined by $X_{n+1} = f(X_n)$. If the range of f is finite, the sequence is necessarily periodic. That is, it consists of a leader X_0, X_1,..., $X_{\mu-1}$ followed by a cycle X_μ, X_{u+1},..., $X_{\mu+\lambda-1}$ that repeats without limit ($X_\mu = X_{\mu+\lambda}$, $X_{\mu+1} = X_{\mu+\lambda+1}$, and so on, where λ is the period of the cycle). Given the function f, the loop-detection problem is to find the index μ of the first element that repeats, and the period λ. Loop detection has applications in testing random number generators and detecting a cycle in a linked list.

One could save all the values of the sequence as they are produced and compare each new element with all the preceding ones. This would immediately show where the second cycle starts. But algorithms exist that are much more efficient in space and time.

Perhaps the simplest is due to R. W. Floyd [Knu2, sec. 3.1, prob. 6]. This algorithm iterates the process

$$x = f(x)$$
$$y = f(f(y))$$

with x and y initialized to X_0. After the nth step, $x = X_n$ and $y = X_{2n}$. These are compared, and if equal, it is known that X_n and X_{2n} are separated by an integral multiple of the period λ—that is, $2n - n = n$ is a multiple of λ. Then μ can be determined by regenerating the sequence from the beginning, comparing X_0 to X_n, then X_1 to X_{n+1}, and so on. Equality occurs when X_μ is compared to $X_{n+\mu}$. Finally, λ can be determined by regenerating more elements, comparing X_μ to $X_{\mu+1}$, $X_{\mu+2}$, ... This algorithm requires only a small and bounded amount of space, but it evaluates f many times.

Gosper's algorithm [HAK, item 132; Knu2, Answers to Exercises for Section 3.1, exercise 7] finds the period λ, but not the starting point μ of the first cycle. Its main feature is that it never backs up to reevaluate f, and it is quite economical in space and time. It is not bounded in space; it requires a table of size $\log_2(\Lambda) + 1$, where Λ is the largest possible period. This is not a lot of space; for example, if it is known a priori that $\Lambda \le 2^{32}$, then 33 words suffice.

Gosper's algorithm, coded in C, is shown in Figure 5–28. This C function is given the function f being analyzed and a starting value X_0. It returns lower and upper bounds on μ, and the period λ. (Although Gosper's algorithm cannot compute μ, it can compute lower and upper bounds μ_l and μ_u such that $\mu_u - \mu_l + 1 \le \max(\lambda - 1, 1)$.) The algorithm works by comparing X_n, for $n = 1, 2, \ldots$, to a subset of size $\log_2 n + 1$ of the elements of the sequence that precede X_n. The elements of the subset are the closest preceding X_i such that $i + 1$ ends in a 1-bit (that is, i is the even number preceding n), the closest preceding X_t such that $i + 1$ ends in exactly one 0-bit, the closest preceding X_t such that **i** + 1 ends in exactly two 0-bits, and so on.

```
void ld_Gosper(int (*f)(int), int X0, int *mu_l,
                              int*mu_u, int *lambda){
   int Xn, k, m, kmax, n, lgl;
   int T[33];

   T[0] = X0;
   Xn = X0;
   for (n = 1; ; n++) {
      Xn = f(Xn);
      kmax = 31 - nlz(n);            // Floor(log2 n).
      for (k = 0; k <= kmax; k++) {
         if (Xn == T[k]) goto L;
      }
      T[ntz(n+1)] = Xn;             // No match.
   }
L:
   // Compute m = max{i | i < n and ntz(i+1) = k}.

   m = ((((n >> k) - 1) | 1) << k) - 1;
   *lambda = n - m;
   lgl = 31 - nlz(*lambda - 1); // Ceil(log2 lambda) - 1.
   *mu_u = m;                       // Upper bound on mu.
   *mu_l = m - max(1, 1 << lgl) + 1;// Lower bound on mu.
}
```

FIGURE 5–28. Gosper's loop-detection algorithm.

Thus, the comparisons proceed as follows:

$X_1 : X_0$

$X_2 : X_0, X_1$

$X_3 : X_2, X_1$

$X_4 : X_2, X_1, X_3$

$X_5 : X_4, X_1, X_3$

$X_6 : X_4, X_5, X_3$

$X_7 : X_6, X_5, X_3$

$X_8 : X_6, X_5, X_3, X_7$

$X_9 : X_8, X_5, X_3, X_7$

$X_{10} : X_8, X_9, X_3, X_7$

$X_{11} : X_{10}, X_9, X_3, X_7$

$X_{12} : X_{10}, X_9, X_{11}, X_7$

$X_{13} : X_{12}, X_9, X_{11}, X_7$

$X_{14} : X_{12}, X_{13}, X_{11}, X_7$

$X_{15} : X_{14}, X_{13}, X_{11}, X_7$

$X_{16} : X_{14}, X_{13}, X_{11}, X_7, X_{15}$

$X_{17} : X_{16}, X_{13}, X_{11}, X_7, X_{15}$

$X_{18} : X_{16}, X_{17}, X_{11}, X_7, X_{15}$

It can be shown that the algorithm always terminates with n somewhere in the second cycle—that is, with $n < \mu + 2\lambda$. See [Knu2] for further details.

The ruler function reveals how to solve the Tower of Hanoi puzzle. Number the n disks from 0 to $n - 1$. At each move k, as k goes from 1 to $2^n - 1$, move disk ntz(k) the minimum permitted distance to the right, in a circular manner.

The ruler function can be used to generate a reflected binary Gray code (see Section 13–1 on page 311). Start with an arbitrary n-bit word, and at each step k, as k goes from 1 to $2^n - 1$, flip bit ntz(k).

Exercises

1. Code Dubé's algorithm for the ntz function, expanding the multiplication.

2. Code the "right justify" function, $x \overset{u}{\gg} \text{ntz}(x)$, $x \neq \mathbf{0}$, in three basic RISC instructions.

3. Are the parallel prefix and suffix (with XOR) operations invertible? If so, how would you compute the inverse functions?

Chapter 6. Searching Words

6–1 Find First 0-Byte

The need for this function stems mainly from the way character strings are represented in the C language. They have no explicit length stored with them; instead, the end of the string is denoted by an all-0 byte. To find the length of a string, a C program uses the “strlen” (string length) function. This function searches the string, from left to right, for the 0-byte, and returns the number of bytes scanned, not counting the 0-byte.

A fast implementation of “strlen” might load and test single bytes until a word boundary is reached, and then load a word at a time into a register, and test the register for the presence of a 0-byte. On big-endian machines, we want a function that returns the index of the first 0-byte from the left. A convenient encoding is values from 0 to 3 denoting bytes 0 to 3, and a value of 4 denoting that there is no 0-byte in the word. This is the value to add to the string length, as successive words are searched, if the string length is initialized to 0. On little-endian machines, one wants the index of the first 0-byte from the right end of the register, because little-endian machines reverse the four bytes when a word is loaded into a register. Specifically, we are interested in the following functions, where “00” denotes a 0-byte, “nn” denotes a nonzero byte, and “xx” denotes a byte that may be 0 or nonzero.

$$\text{zbytel}(x) = \begin{cases} 0, & x = \text{00xxxxxx}, \\ 1, & x = \text{nn00xxxx}, \\ 2, & x = \text{nnnn00xx}, \\ 3, & x = \text{nnnnnn00}, \\ 4, & x = \text{nnnnnnnn}. \end{cases} \qquad \text{zbyter}(x) = \begin{cases} 0, & x = \text{xxxxxx00}, \\ 1, & x = \text{xxxx00nn}, \\ 2, & x = \text{xx00nnnn}, \\ 3, & x = \text{00nnnnnn}, \\ 4, & x = \text{nnnnnnnn}. \end{cases}$$

Our first procedure for the *find leftmost 0-byte* function, shown in Figure 6–1, simply tests each byte, in left-to-right order, and returns the result when the first 0-byte is found.

```
int zbytel(unsigned x) {
   if              ((x >> 24) == 0) return 0;
   else if ((x & 0x00FF0000) == 0) return 1;
   else if ((x & 0x0000FF00) == 0) return 2;
   else if ((x & 0x000000FF) == 0) return 3;
   else return 4;
}
```

FIGURE 6–1. *Find leftmost 0-byte*, **simple sequence of tests.**

This executes in two to 11 basic RISC instructions, 11 in the case that the word has no 0-bytes (which is the important case for the “strlen” function). A very similar program will handle the problem of finding the rightmost 0-byte.

Figure 6–2 shows a branch-free procedure for this function. The idea is to convert each 0-byte to 0x80, and each nonzero byte to 0x00, and then use *number of leading zeros*. This procedure executes in eight instructions, if the machine has the *number of leading zeros* and *nor* instructions. Some similar tricks are described in [Lamp].

```
int zbytel(unsigned x) {
   unsigned y;
   int n;
                          // Original byte: 00 80 other
   y = (x & 0x7F7F7F7F)+ 0x7F7F7F7F;     // 7F 7F 1xxxxxxx
   y = ~(y 1 x 1 0x7F7F7F7F);            // 80 00 00000000
   n = nlz(y) >> 3;             // n = 0 ... 4, 4 if x
   return n;                    // has no 0-byte.
}
```

FIGURE 6–2. *Find leftmost 0-byte*, **branch-free code.**

The position of the *rightmost* 0-byte is given by the number of trailing 0's in the final value of `y` computed above, divided by 8 (with fraction discarded). Using the expression for computing the number of trailing 0's by means of the *number of leading zeros* instruction (see Section 5–4, "Counting Trailing 0's," on page 107), this can be computed by replacing the assignment to `n` in the procedure above with:

```
n = (32 - nlz(~y & (y - 1))) >> 3;
```

This is a 12-instruction solution, if the machine has *nor* and *and not*.

In most situations on PowerPC, incidentally, a procedure to find the rightmost 0-byte would not be needed. Instead, the words can be loaded with the *load word byte-reverse* instruction (`lwbrx`).

The procedure of Figure 6–2 is more valuable on a 64-bit machine than on a 32-bit one, because on a 64-bit machine the procedure (with obvious modifications) requires about the same number of instructions (seven or ten, depending upon how the constant is generated), whereas the technique of Figure 6–1 requires 23 instructions worst case.

If only a test for the presence of a 0-byte is wanted, then a *branch on zero* (or *nonzero*) can be inserted just after the second assignment to `y`.

A method similar to that of Figure 6–2, but for finding the *rightmost* 0-byte in a word `x` (zbyter(*x*)), is [Mycro]:

```
y = (x - 0x01010101) & ~x & 0x80808080;
n = ntz(y) >> 3;
```

This executes in only five instructions exclusive of loading the constants if the machine has the *and not* and *number of trailing zeros* instructions. It cannot be used to compute zbytel(*x*), because of a problem with borrows. It would be most useful for finding the first 0-byte in a character string on a little-endian machine, or to simply test for a 0-byte (using only the assignment to `y`) on a machine of either endianness.

If the nlz instruction is not available, there does not seem to be any really good way to compute the *find first 0-byte* function. Figure 6–3 shows a possibility (only the executable part of the code is shown).

This executes in ten to 13 basic RISC instructions, ten in the all-nonzero case. Thus, it is probably not as good as the code of Figure 6–1, although it does have fewer branch instructions. It does not scale very well to 64-bit machines, unfortunately.

There are other possibilities for avoiding the nlz function. The value of `y` computed by the code of Figure 6–3 consists of four bytes, each of which is either 0x00 or 0x80.

The remainder after dividing such a number by 0x7F is the original value with the up-to-four 1-bits moved and compressed to the four rightmost positions. Thus, the remainder ranges from 0 to 15 and uniquely identifies the original number. For example,

$$\begin{aligned}\text{remu}(\mathbf{0x80808080}, \mathbf{127}) &= \mathbf{15},\\ \text{remu}(\mathbf{0x80000000}, \mathbf{127}) &= \mathbf{8},\\ \text{remu}(\mathbf{0x00008080}, \mathbf{127}) &= \mathbf{3}, \text{etc.}\end{aligned}$$

This value can be used to index a table, 16 bytes in size, to get the desired result. Thus, the code beginning `if (y == 0)` can be replaced with

```
static char table[16] = {4, 3, 2, 2, 1, 1, 1, 1,
                          0, 0, 0, 0, 0, 0, 0, 0};
return table[y%127];
```

where `y` is unsigned. The number 31 can be used in place of 127, but with a different table.

```
                              // Original byte: 00 80 other
y = (x & 0x7F7F7F7F) + 0x7F7F7F7F; // 7F 7F 1xxxxxxx
y = ~(y | x | 0x7F7F7F7F);          // 80 00 00000000
                                    // These steps map:
if (y == 0) return 4;               // 00000000 ==> 4,
else if (y > 0x0000FFFF)            // 80xxxxxx ==> 0,
   return (y >> 31) ^ 1;            // 0080xxxx ==> 1,
else                                // 000080xx ==> 2,
   return (y >> 15) ^ 3;            // 00000080 ==> 3.
```

FIGURE 6–3. *Find leftmost 0-byte*, **not using `nlz`.**

These methods involving dividing by 127 or 31 are really just curiosities, because the *remainder* function is apt to require 20 cycles or more, even if directly implemented in hardware. However, below are two more efficient replacements for the code in Figure 6–3 beginning with `if (y == 0)`:

```
return table[hopu(y, 0x02040810) & 15];
return table[y*0x00204081 >> 28];
```

Here, `hopu(a, b)` denotes the high-order 32 bits of the unsigned product of `a` and `b`. In the second line, we assume the usual HLL convention that the value of the multiplication is the low-order 32 bits of the complete product. This might be a practical method, if either the machine has a fast multiply or the multiplication by 0x204081 is done by *shift*-and-*add*'s. It can be done in four such instructions, as suggested by

$$y(1 + 2^7 + 2^{14} + 2^{21}) = y(1 + 2^7)(1 + 2^{14}).$$

Using this 4-cycle way to do the multiplication, the total time for the procedure comes to 13 cycles (7 to compute `y`, plus 4 for the *shift*-and-*add*'s, plus 2 for the *shift right* of 28 and the table index), and of course it is branch-free.

These scale reasonably well to a 64-bit machine. For the “modulus” method, use

```
return table[y%511];
```

where `table` is of size 256, with values 8, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, ... (i.e., `table[i]` = number of trailing 0’s in *i*).

For the multiplicative methods, use either

```
return table[hopu(y, 0x02040810 20408100) & 255]; or
return table[(y*0x00020408 10204081>>56];
```

where `table` is of size 256, with values 8, 7, 6, 6, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 3, ...

The multiplication by 0x20408 10204081 can be done with

$$t_1 \leftarrow y(1 + 2^7)$$

$$t_2 \leftarrow t_1(1 + 2^{14})$$

$$t_3 \leftarrow t_2(1 + 2^{28})$$

which gives a 13-cycle solution.

All these variations using the table can, of course, implement the *find rightmost 0-byte* function by simply changing the data in the table.

If the machine does not have the *nor* instruction, the *not* in the second assignment to `y` in Figure 6–3 can be omitted, in the case of a 32-bit machine, by using one of the three `return` statements given above, with `table[i]` = 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 4. This scheme does not quite work on a 64-bit machine.

Here is an interesting variation on the procedure of Figure 6–2, again aimed at machines that do not have *number of leading zeros*. Let *a, b, c*, and *d* be 1-bit variables for the predicates “the first byte of *x* is nonzero,” “the second byte of *x* is nonzero,” and so on. Then,

$$\text{zbytel}(x) = a + ab + abc + abcd.$$

The multiplications can be done with *and*’s, leading to the procedure shown in Figure 6–4 (only the executable code is shown). This comes to 15 instructions on the basic RISC, which is not particularly fast, but there is a certain amount of parallelism. On a superscalar machine that can execute up to three arithmetic instructions in parallel, provided they are independent, it comes to only ten cycles.

```
y = (x & 0x7F7F7F7F) + 0x7F7F7F7F;
y = y | x;           // Leading 1 on nonzero bytes.

t1 = y >> 31;                 // tl = a.
t2 = (y >> 23) & tl;          // t2 = ab.
t3 = (y >> 15) & t2;          // t3 = abc.
t4 = (y >>  7) & t3;           // t4 = abcd.
return t1 + t2 + t3 + t4;
```

FIGURE 6–4. *Find leftmost 0-byte* **by evaluating a polynomial.**

A simple variation of this does the *find rightmost 0-byte* function, based on

$$\text{zbyter}(x) = abcd + bcd + cd + d.$$

(This requires one more *and* than the code of Figure 6–4.)

Some Simple Generalizations

Functions zbytel and zbyter can be used to search for a byte equal to any particular value, by first *exclusive or*'ing the argument *x* with a word consisting of the desired value replicated in each byte position. For example, to search *x* for an ASCII blank (0x20), search *x* **0***x* **20202020** for a 0-byte.

Similarly, to search for a byte position in which two words *x* and *y* are equal, search *x* *y* for a 0-byte.

There is nothing special about byte boundaries in the code of Figure 6–2 and its variants. For example, to search a word for a 0-value in any of the first four bits, the next 12, or the last 16, use the code of Figure 6–2 with the mask replaced by 0x77FF7FFF [PHO]. (If a field length is 1, use a 0 in the mask at that position.)

Searching for a Value in a Given Range

The code of Figure 6–2 can easily be modified to search for a byte in the range 0 to any specified value less than 128. To illustrate, the following code finds the index of the leftmost byte having value from 0 to 9:

```
y = (x & 0x7F7F7F7F) + 0x76767676;
y = y | x;
y = y | 0x7F7F7F7F;          // Bytes > 9 are 0xFF.
y = ~y;                       // Bytes > 9 are 0x00,
                              // bytes <= 9 are 0x80.
n = nlz(y) >> 3;
```

More generally, suppose you want to find the leftmost byte in a word that is in the range *a* to *b*, where the difference between *a* and *b* is less than 128. For example, the uppercase letters encoded in ASCII range from 0x41 to 0x5A. To find the first uppercase letter in a word, subtract 0x41414141 in such a way that the borrow does not propagate across byte boundaries, and then use the above code to identify bytes having value from 0 to 0x19 (0x5A – 0x41). Using the formulas for subtraction given in Section 2–18, “Multibyte Add, Subtract, Absolute Value,” on page 40, with obvious simplifications possible with *y* = 0x41414141, gives

```
d = (x | 0x80808080) - 0x41414141;
d = ~((x | 0x7F7F7F7F) ^ d);
y = (d & 0x7F7F7F7F) + 0x66666666;
y = y | d;
y = y | 0x7F7F7F7F;     // Bytes not from 41–5A are FF.
y = ~y;                 // Bytes not from 41–5A are 00,
                        // bytes from 41–5A are 80.
n = nlz(y) >> 3;
```

For some ranges of values, simpler code exists. For example, to find the first byte whose value is 0x30 to 0x39 (a decimal digit encoded in ASCII), simply *exclusive or* the input word with 0x30303030 and then use the code given above to search for a value in the range 0 to 9. (This simplification is applicable when the upper and lower limits have *n* high-order bits in common, and the lower limit ends with 8 – *n* 0’s.)

These techniques can be adapted to handle ranges of 128 or larger with no additional instructions. For example, to find the index of the leftmost byte whose value is in the range 0 to 137 (0x89), simply change the line `y = y | x` to `y = y & x` in the code above for searching for a value from 0 to 9.

Similarly, changing the line `y = y | d` to `y = y & d` in the code for finding the leftmost byte whose value is in the range 0x41 to 0x5A causes it to find the leftmost byte whose value is in the range 0x41 to 0xDA.

6–2 Find First String of 1-Bits of a Given Length

The problem here is to search a word in a register for the first string of 1-bits of a given length *n* or longer, and to return its position, with some special indication if no such string exists. Variants are to return only the yes/no indication and to locate the first string of exactly *n* 1-bits. This problem has application in disk-allocation programs, particularly for disk compaction (rearranging data on a disk so that all blocks used to store a file are contiguous). The problem was suggested to me by Albert Chang, who pointed out that it is one of the uses for the *number of leading zeros* instruction.

We assume here that the *number of leading zeros* instruction, or a suitable subroutine for that function, is available.

An algorithm that immediately comes to mind is to first count the number of leading 0's and skip over them by shifting left by the number obtained. Then count the leading 1's by inverting and counting leading 0's. If this is of sufficient length, we are done. Otherwise, shift left by the number obtained and repeat from the beginning. This algorithm might be coded as shown below. If *n* consecutive 1-bits are found, it returns a number from 0 to 31, giving the position of the leftmost 1-bit in the leftmost such sequence. Otherwise, it returns 32 as a “not found” indication.

```
int ffstr1(unsigned x, int n) {
   int k, p;

   p = 0;                  // Initialize position to return.
   while (x != 0) {
      k = nlz(x);          // Skip over initial 0's
      x = x << k;          // (if any).
      p = p + k;
      k = nlz(~x);         // Count first/next group of 1's.
      if (k >= n)          // If enough,
         return p;         // return.
      x = x << k;          // Not enough 1's, skip over
      p = p + k;           // them.
   }
   return 32;
}
```

This algorithm is reasonable if it is expected that the loop will not be executed very many times—for example, if it is expected that `x` will have long sequences of 1's and of 0's. This might very well be the expectation in the disk-allocation application. Its worst-case execution time, however, is not very good; for example, about 178 full RISC instructions executed for x = 0x55555555 and $n \geq 2$.

An algorithm that is better in worst-case execution time is based on a sequence of *shift left* and *and* instructions. To see how this works, consider searching for a string of eight or more consecutive 1-bits in a 32-bit word *x*. This might be done as follows:

$$x \leftarrow x \,\&\, (x \ll 1)$$

$$x \leftarrow x \,\&\, (x \ll 2)$$

$$x \leftarrow x \,\&\, (x \ll 4)$$

After the first assignment, the 1's in x indicate the starting positions of strings of length 2. After the second assignment, the 1's in x indicate the starting positions of strings of length 4 (a string of length 2 followed by another string of length 2). After the third assignment, the 1's in x indicate the starting positions of strings of length 8. Executing *number of leading zeros* on this word gives the position of the first string of length 8 (or more), or 32 if none exists.

To develop an algorithm that works for any length n from 1 to 32, we will look at this a little differently. First, observe that the above three assignments can be done in any order. Reverse order will be more convenient. To illustrate the general method, consider the case n = 10:

$$x_1 \leftarrow x \;\;\&\, (x \;\;\ll 5)$$

$$x_2 \leftarrow x_1 \,\&\, (x_1 \ll 2)$$

$$x_3 \leftarrow x_2 \,\&\, (x_2 \ll 1)$$

$$x_4 \leftarrow x_3 \,\&\, (x_3 \ll 1)$$

The first statement shifts by n/2. After executing it, the problem is reduced to finding a string of five consecutive 1-bits in x_1. This can be done by shifting left by 5/2 = 2, *and*'ing, and searching the result for a string of length 3 (5 – 2). The last two statements identify where the strings of length 3 are in x_2. The sum of the shift amounts is always n– 1. The algorithm is shown in Figure 6–5. The execution time ranges from 3 to 36 full RISC instructions, as `n` ranges from 1 to 32.

```
int ffstr1(unsigned x, int n) {
   int s;

   while (n > 1) {
      s = n >> 1;
      x = x & (x >> s);
      n = n - s;
   }
   return nlz(x);
}
```

FIGURE 6–5. Find first string of *n* 1's, *shift*-and-*and* sequence.

If `n` is often moderately large, it is not unreasonable to unroll this loop by repeating the loop body five times and omitting the test `n > 1`. (Five is always sufficient for a 32-bit machine.) This gives a branch-free algorithm that runs in a constant time of 20 instructions executed (the last assignment to `n` can be omitted). Although for small values of `n`, the three assignments are executed more than necessary, the result is unchanged by the extra steps, because variable `n` sticks at the value 1, and for this value the three steps have no effect on x or n. The unrolled version is faster than the looping version for n ≥ 5, in terms of number of instructions executed.

A string of exactly *n* 1-bits can be found in six more instructions (four if *and not* is available). The quantity *x* computed by the algorithm of Figure 6–5 has 1-bits wherever a string of length *n* or more 1-bits begins. Hence, using the final value of *x* computed by that algorithm, the expression

$$x \mathbin{\&} \neg(x \overset{u}{\gg} 1) \mathbin{\&} \neg(x \ll 1)$$

contains a 1-bit wherever the final *x* contains an isolated 1-bit, which is to say wherever the original *x* began a string of exactly *n* 1-bits.

The algorithm is also easily adapted to finding strings of length *n* that begin at certain locations. For example, to find strings that begin at byte boundaries, simply *and* the final *x* with 0x80808080.

It can be used to find strings of 0-bits either by complementing *x* at the start, or by changing the *and*'s to *or*'s and complementing *x* just before invoking nlz. For example, below is an algorithm for finding the first (leftmost) 0-byte (see Section 6–1, "Find First 0-Byte," on page 117, for a precise definition of this problem).

$$x \leftarrow x \mid (x \ll 4)$$
$$x \leftarrow x \mid (x \ll 2)$$
$$x \leftarrow x \mid (x \ll 1)$$
$$x \leftarrow \mathrm{0x7F7F7F7F} \mid x$$
$$p \leftarrow \mathrm{nlz}(\neg x) \overset{u}{\gg} 3$$

This executes in 12 instructions on the full RISC (not as good as the algorithm of Figure 6–2 on page 118, which executes in eight instructions).

6–3 Find Longest String of 1-Bits

The nicely concise function shown in Figure 6–6 returns the length of the longest string of 1-bits in x [Hsieh].

```
int maxstr1(unsigned x) {
   int k;
   for (k = 0; x ! = 0; k++) x = x & 2*x;
   return k;
}
```

FIGURE 6–6. Find length of longest string of 1's.

It executes in 4*n* + 3 instructions on the basic RISC, where *n* is the length of the longest string of 1's, or 131 instructions in the worst case.

To reduce the worst-case execution time, a "logarithmic" version is possible. It works by propagating 0's one, two, four, eight, and 16 positions to the left, stopping at the last nonzero word, and then backtracking to find the length of the longest contiguous string of 1’s.

For example, suppose

```
 x = 0011 1111 1111 0011 1111 0011 1111 1000
```

Then

```
 x2 = 0011 1111 1110 0011 1110 0011 1111 0000
 x4 = 0011 1111 1000 0011 1000 0011 1100 0000
 x8 = 0011 1000 0000 0000 0000 0000 0000 0000
x16 = all 0's
```

In this case, the last nonzero word is `x8`. Observe that each 1-bit in `x8` indicates the leftmost position of a string of eight 1’s. Thus, the longest string of 1’s begins at the leftmost position of a 1-bit in `x8`, bit position 29 in the example. To test for a string of length 12, one can test the bit at position 21 (29 – 8) in `x4`. Since that is 0, there is no string of length 12. To test for a string of length 10, one can test the bit at position 21 in `x2`. Since that is 1, position 29 is the start of a string of length 10 (or more). Last, to test for a string of length 11, one can test the bit at position 19 (21 – 2) in `x`. Because that is 0, the longest string is of length 10, and it starts at position 29.

This scheme is coded in Figure 6–7, except the code uses only two variables, `x` and `y`, instead of the five variables `x`, `x2`, `x4`, `x8`, and `x16`. This code finds both the length and position of the longest string of 1’s, with the position being measured from the left end of the string. The scheme does not work if `x` is 0 or all 1’s. These are special-cased, with the latter possibility being handled in a place that is not executed frequently.

```
int fmaxstr1(unsigned x, int *apos) {
   unsigned y;
   int s;

   if (x == 0) {*apos = 32; return 0;}
   y = x & (x << 1);
   if (y == 0) {s = 1; goto L1;}
   x = y & (y << 2);
   if (x == 0) {s = 2; x = y; goto L2;}
   y = x & (x << 4);
   if (y == 0) {s = 4; goto L4;}
   x = y & (y << 8);
   if (x == 0) {s = 8; x = y; goto L8;}
   if (x == 0xFFFF8000) {*apos = 0; return 32;}
   s = 16;

L16: y = x & (x << 8);
     if (y != 0) {s = s + 8; x = y;}
L8:  y = x & (x << 4);
     if (y != 0) {s = s + 4; x = y;}
L4:  y = x & (x << 2);
     if (y != 0) {s = s + 2; x = y;}
L2:  y = x & (x << 1);
     if (y != 0) {s = s + 1; x = y;}
L1:  *apos = nlz(x);
   return s;
}
```

FIGURE 6–7. Find length and position of longest string of 1's.

The worst-case execution time on the basic RISC is 39 instructions, plus those required for the nlz function. If only the length of the longest string of 1's is wanted, there is no significant savings in execution time, except for omitting the use of the nlz function.

6–4 Find Shortest String of 1-Bits

It is more difficult to find the *shortest* string of 1-bits in a word. One way to do it is to mark the beginnings of all strings of 1's in a word `b` and the ends of all such strings in a word `e`. Then, if `b & e` is nonzero, the shortest string is of length 1. Otherwise, shift `e` left one position and test again. For example, if

```
x = 0011 1111 1111 0011 1111 0011 1111 1000
```

then

```
b = 0010 0000 0000 0010 0000 0010 0000 0000
e = 0000 0000 0001 0000 0001 0000 0000 1000
```

After shifting `e` left five places, `b & e` is nonzero. This means that the shortest string of 1-bits is of length 6.

This idea is embodied in the code shown in Figure 6–8. As in the preceding material, the position of the string is measured from the left, and if there are two or more minimal length strings of equal length, this function finds the leftmost one. For example, if `x` = 0x00FF0FF0 it returns length 8, position 8.

```
int fminstr1(unsigned x, int *apos) {
   int k;
   unsigned b, e;       // Beginnings, ends.

   if (x == 0) {*apos = 32; return 0;}
   b = ~(x >> 1) & x;   // 0-1 transitions.
   e = x & ~(x << 1);   // 1-0 transitions.
   for (k = 1; (b & e) == 0; k++)
      e = e << 1;
   *apos = nlz(b & e);
   return k;
}
```

FIGURE 6–8. Find length and position of shortest string of 1's.

The function executes in 8 + 4 n instructions on the basic RISC (without *andc*), plus the time for the nlz function, for $n \geq 2$, where n is the length of the shortest contiguous string of 1's in `x`.

Perhaps the ultimate problem in this class is to find the length and position of the shortest string of 1's in x that is at least as long as a given integer $n > 0$. In terms of the storage allocation problem, this is a "best fit" algorithm. This can be done by first left-propagating the 0's in x by $n - 1$ positions and then finding the shortest string of 1's in the revised x. See the exercises.

Exercises

1. Code an elaboration of Hsieh's algorithm that will find both the length and position of the longest string of 1's in a word x. You may use the nlz function.

2. Code a function for finding the length and position of the shortest string of 1's in a word x that is at least as long as a given integer n.

3. Another way to find the shortest string of 1's in a word x is to successively turn off the rightmost string of 1's in x and observe the change in population count at each step. Code a function for the full RISC that uses this idea and also finds the position of a shortest string of 1's.

4. For "completely random" 32-bit words x (each bit independently 0 or 1 with probability 0.5), what is the average number of strings of 1's in x? The answer determines the average execution time of the function of exercise 3, for such input data.

5. Again, for "completely random" 32-bit words x, what is the average length of the shortest contiguous string of 1's in x? The answer determines the average execution time of function `fminstr1` in Figure 6–8 for such input data. Compute this with a Monte Carlo or exhaustive enumeration program.

6. Of the 2^n binary words of length n, for how many is their shortest contained string of 1's of length 1? That is, how many n-bit words begin with 10, or end with 01, or contain the sequence 010? Find a closed-form solution or a recursion, not an exhaustive enumeration program.

7. Similarly, of the 2^n binary words of length n, for how many is their shortest contained string of 1's of length 2?

Chapter 7. Rearranging Bits and Bytes

7–1 Reversing Bits and Bytes

By “reversing bits” we mean to reflect the contents of a register about the middle so that, for example,

rev(**0x01234567**) = **0xE6A2C480**.

By “reversing bytes” we mean a similar reflection of the four bytes of a register. Byte reversal is a necessary operation to convert data between the “little-endian” format used by DEC and Intel, and the “big-endian” format used by most other manufacturers.

Bit reversal can be done quite efficiently by interchanging adjacent single bits, then interchanging adjacent 2-bit fields, and so on, as shown below [Aus1]. These five assignment statements can be executed in any order. This is the same algorithm as the first *population count* algorithm of Section 5–1, but with addition replaced with swapping.

```
x = (x & 0x55555555)  <<   1 | (x & 0xAAAAAAAA) >>  1;
x = (x & 0x33333333)  <<   2 | (x & 0xCCCCCCCC) >>  2;
x = (x & 0x0F0F0F0F)  <<   4 | (x & 0xF0F0F0F0) >>  4;
x = (x & 0x00FF00FF)  <<   8 | (x & 0xFF00FF00) >>  8;
x = (x & 0x0000FFFF)  <<  16 | (x & 0xFFFF0000) >> 16;
```

A small improvement may result on some machines by using fewer distinct large constants and doing the last two assignments in a more straightforward way, as shown in Figure 7–1 (30 basic RISC instructions, branch-free).

```
unsigned rev(unsigned x) {
   x = (x & 0x55555555) << 1 | (x >> 1) & 0x55555555;
   x = (x & 0x33333333) << 2 | (x >> 2) & 0x33333333;
   x = (x & 0x0F0F0F0F) << 4 | (x >> 4) & 0x0F0F0F0F;
   x = (x << 24) | ((x & 0xFF00) << 8) |
       ((x >> 8) & 0xFF00) | (x >> 24);
   return x;
}
```

FIGURE 7–1. Reversing bits.

The last assignment to *x* in this code does byte reversal in nine basic RISC instructions. If the machine has rotate shifts, however, this can be done in seven instructions with

$$x \leftarrow ((x \mathbin{\&} \mathtt{0x00FF00FF}) \overset{rot}{\ggg} 8) \mid ((x \overset{rot}{\lll} 8) \mathbin{\&} \mathtt{0x00FF00FF}).$$

PowerPC can do the byte-reversal operation in only three instructions [Hay1]: a *rotate left* of 8, which positions two of the bytes, followed by two “rlwimi” (*rotate left word immediate then mask insert*) instructions.

The next algorithm, by Christopher Strachey [Strach 1961], is old by computer standards, but it is instructive. It reverses the rightmost 16 bits of a word, assuming the leftmost 16 bits are clear at the start, and places the reversed halfword in the left half of the register.

Its operation is based on the number of bit positions that each bit must move. The 16 bits, taken from left to right, must move 1, 3, 5, ..., 31 positions. The bits that must move 16 or more positions are moved first, then those that must move eight or more positions, and so forth. The operation is illustrated below, where each letter denotes a single bit, and a period denotes a “don’t care” bit.

```
0000 0000 0000 0000 abcd efgh ijkl mnop Given
0000 0000 ijkl mnop abcd efgh .... .... After shl 16
0000 mnop ijkl efgh abcd .... .... .... After shl 8
00op mnkl ijgh efcd ab.. .... .... .... After shl 4
0pon mlkj ihgf edcb a... .... .... .... After shl 2
ponm lkji hgfe dcba .... .... .... .... After shl 1
```

Straightforward code consists of 16 basic RISC instructions, plus 12 to load the constants:

```
x = x | ((x & 0x000000FF) << 16);
x = (x & 0xF0F0F0F0) | ((x & 0x0F0F0F0F) << 8);
x = (x & 0xCCCCCCCC) | ((x & 0x33333333) << 4);
x = (x & 0xAAAAAAAA) | ((x & 0x55555555) << 2);
x = x << 1;
```

Complementation can be used to reduce the number of distinct masks. By using more irregular masks, the rightmost 16 bits can be preserved.

If rotate shifts are available, Strachey’s idea can be used to reverse a 32-bit word. The idea is to consider how many bit positions each bit must move rotationally to the left to get to its final position. Taking the bits from left to right, the shift amounts are 1, 3, 5, ..., 31, 1, 3, 5, ..., 31 (no bit moves an even number of positions). The algorithm first rotate-moves those bits that must move 16 or more positions, then those that must move eight or more positions, and so forth, and finally those that must move one position (which is all of the bits, because all move amounts are odd). This scheme is shown below, for reversing a 32-bit word `x`. Function `shlr(x, y)` rotates `x` left `y` positions.

```
x = shlr(x & 0x00FF00FF, 16) | x & ~0x00FF00FF;
x = shlr(x & 0x0F0F0F0F,  8) | x & ~0x0F0F0F0F;
x = shlr(x & 0x33333333,  4) | x & ~0x33333333;
x = shlr(x & 0x55555555,  2) | x & ~0x55555555;
x = shlr(x, 1);
```

The code uses *and with complement* to avoid loading some masks. If your machine does not have that instruction, it can be avoided by rewriting the first line of code as

```
x = shlr(x, 16) & 0x00FF00FF | x & ~0x00FF00FF;
```

which is a MUX operation, and using the identity

$$x \mathbin{\&} m \mid y \mathbin{\&} \neg m = ((x \oplus y) \mathbin{\&} m) \oplus y$$

to obtain

```
x = ((shlr(x, 16) ^ x) & 0x00FF00FF) ^ x;
```

and similarly for the other lines that have *and with complement*.

A slightly better way for many machines, in that it has a little instruction-level parallelism, is to use the identity [Karv]

$$x \,\&\, \neg m = (x \,\&\, m) \oplus x,$$

and common the *and* expression. This gives the function shown in Figure 7–2 (17 instructions, plus eight to load constants, or 25 in all).

```
unsigned rev(unsigned x) {
   unsigned t;
   t = x & 0x00FF00FF; x = shlr(t, 16) | t ^ x;
   t = x & 0x0F0F0F0F; x = shlr(t,  8) | t ^ x;
   t = x & 0x33333333; x = shlr(t,  4) | t ^ x;
   t = x & 0x55555555; x = shlr(t,  2) | t ^ x;
   x = shlr(x, 1);
   return x;
}
```

FIGURE 7–2. Reversing bits with rotate shifts.

It is perhaps worth noting that the constants 0x00FF00FF, 0x0F0F0F0F, and so on can be generated one from another as shown below. This is not useful for 32-bit machines (it may even be harmful by reducing parallelism), because 32-bit RISC machines generally can load the constants in two instructions. But it might be useful for a 64-bit machine, for which it is illustrated.

$$C_0 \leftarrow \text{0x00000000 FFFFFFFF}$$

$$C_1 \leftarrow C_0 \oplus (C_0 \ll 16)$$

$$C_2 \leftarrow C_1 \oplus (C_1 \ll 8)$$

...

Another way to reverse bits is to break the word up into three groups of bits, and swap the leftmost and rightmost groups, leaving the center group in place [Baum]. For a 27-bit word, this works as illustrated below.

```
012345678 9abcdefgh ijklmnopq   The given 27-bit word
ijklmnopq 9abcdefgh 012345678   First ternary swap
opqlmnijk fghcde9ab 678345012   Second ternary swap
qponmlkji hgfedcba9 876543210   Third ternary swap
```

Straightforward code for this follows. If run on a 32-bit machine, it reverses bits 0 to 26, placing the result in bit positions 0 to 26, and clearing bits 27 to 31.

```
x = (x & 0x000001FF) << 18 | (x & 0x0003FE00) |
    (x >> 18) & 0x000001FF;
x = (x & 0x001C0E07) <<  6 | (x & 0x00E07038) |
    (x >> 6) & 0x001C0E07;
x = (x & 0x01249249) <<  2 | (x & 0x02492492) |
    (x >> 2) & 0x01249249;
```

This amounts to 21 basic RISC instructions, plus 10 to load the constants, or 31 in all. In comparison, the code of Figure 7–1 is 24 basic RISC instructions, plus six to load constants, plus a shift right of 5 to right-justify the result, or 31 in all. Thus, the ternary method is equal or superior when there are 27 or fewer bits to be reversed.

The next function, by Donald E. Knuth [Knu8], is interesting because it reverses a 32-bit word with only four stages, and the shifting and masking steps are unexpectedly irregular. It uses one rotate shift and three ternary swaps. It works as follows:

```
   01234567 89abcdef ghijklmn opqrstuv   Given
   fghijklm nopqrstu v0123456 789abcde   Rotate left 15
   pqrstuvm nofghijk labcde56 78901234   10-swap
   tuvspqrm nojklifg hebcda96 78541230   4-swap
   vutsrqpo mnlkjihg fedcba98 76543210   2-swap
```

Straightforward code is shown below.

```
   x = shlr(x, 15);             // Rotateleft 15.
   x = (x & 0x003F801F) << 10 | (x & 0x01C003E0) |
       (x >> 10) & 0x003F801F;
   x = (x & 0x0E038421) <<  4 | (x & 0x11C439CE) |
       (x >>  4) & 0x0E038421;
   x = (x & 0x22488842) <<  2 | (x & 0x549556B5) |
       (x >>  2) & 0x22488842;
```

An improvement in operation count, at the expense of parallelism, results from rewriting

```
   x = (x & M1) << s | (x & M2) | (x >> s) & M1;
```

where `M2` is `~(M1 | (M1 << s))`, as:

```
   t = (x ^ (x >> s)) & M1; x = (t | (t << s)) ^ x;
```

This results in the code in Figure 7–3 (19 full RISC instructions, plus six to load constants, or 25 in all).

```
unsigned rev(unsigned x) {
   unsigned t;

   x = shlr(x, 15);              // Rotateleft 15.
   t = (x ^ (x>>10)) & 0x003F801F;  x = (t | (t<<10)) ^ x;
   t = (x ^ (x>> 4)) & 0x0E038421;  x = (t | (t<< 4)) ^ x;
   t = (x ^ (x>> 2)) & 0x22488842;  x = (t | (t<< 2)) ^ x;
   return x;
}
```

FIGURE 7–3. Reversing bits, Knuth's algorithm.

Although Knuth's algorithm does not beat the algorithm shown in Figure 7–2 for reversing a 32-bit quantity with rotate shifts allowed (17 instructions, plus eight to load constants), Knuth's code uses only one rotate shift instruction. If it is coded as

```
x = (x << 15) | (x >> 17);    // Rotate left 15.
```

then Knuth's algorithm is 21 instructions, plus six to load constants, which is the best found by these measures for rotating a 32-bit word using only basic RISC instructions. This makes one wonder if there is a simple way to predict the number of shifts and logical operations required to reverse a word of a given length.

Can Knuth's algorithm be extended to reversing 64 bits on a 64-bit machine? Yes, there is a simple way and a way that is more difficult to work out. The simple way is to first swap the two halves of the 64-bit register, and then apply the 32-bit version of Knuth's algorithm to both halves, in parallel. The resulting code is shown in Figure 7–4. It is 24 operations, if the swap (rotate 32) counts as one.

```
unsigned long long rev(unsigned long long x) {
   unsigned long long t;

   x = (x << 32) | (x >> 32);    // Swap register halves.
   x = (x & 0x0001FFFF0001FFFFLL) << 15 | // Rotate left
       (x & 0xFFFE0000FFFE0000LL) >> 17;  // 15.
   t = (x ^ (x >> 10)) & 0x003F801F003F801FLL;
   x = (t | (t << 10)) ^ x;
   t = (x ^ (x >> 4)) & 0x0E0384210E038421LL;
   x = (t | (t << 4)) ^ x;
   t = (x ^ (x >> 2)) & 0x2248884222488842LL;
   x = (t | (t << 2)) ^ x;
   return x;
}
```

FIGURE 7–4. Knuth's algorithm applied to 64 bits.

The other way is to find shift amounts and masks analogous to those used in Knuth's 32-bit reversal algorithm. This is shown below. It is 25 operations, if the rotate left shift of 31 positions counts as one operation.

```
unsigned long long rev(unsigned long long x) {
   unsigned long long t;

   x = (x << 31) | (x >> 33);   // I.e., shlr(x, 31).
   t = (x ^ (x >> 20)) & 0x00000FFF800007FFLL;
   x = (t | (t << 20)) ^ x;
   t = (x ^ (x >> 8)) & 0x00F8000F80700807LL;
   x = (t | (t << 8)) ^ x;
   t = (x ^ (x >> 4)) & 0x0808708080807008LL;
   x = (t | (t << 4)) ^ x;
   t = (x ^ (x >> 2)) & 0x1111111111111111LL;
   x = (t | (t << 2)) ^ x;
   return x;
}
```

Bit reversal can be aided by table lookup. The code that follows reverses a byte at a time, using a 256-byte table, and accumulates in reverse order the four bytes selected from the table. If the loop is strung out, this amounts to 13 basic RISC instructions, plus four loads, so it could be a winner on some machines.

```
unsigned rev(unsigned x) {
   static unsigned char table[256] = {0x00, 0x80, 0x40,
   0xC0, 0x20, 0xA0, 0x60, 0xE0, ..., 0xBF, 0x7F, 0xFF};
   int i;
   unsigned r;

   r = 0;
   for (i = 3; i >= 0; i--) {
      r = (r << 8) + table[x & 0xFF];
      x = x >> 8;
   }
   return r;
}
```

Generalized Bit Reversal

[GLS1] suggests that the following sort of generalization of bit reversal, which he calls "flip," is a good candidate to consider for a computer's instruction set:

```
if (k &  1) x = (x & 0x55555555) <<  1 | (x & 0xAAAAAAAA) >>  1;
if (k &  2) x = (x & 0x33333333) <<  2 | (x & 0xCCCCCCCC) >>  2;
if (k &  4) x = (x & 0x0F0F0F0F) <<  4 | (x & 0xF0F0F0F0) >>  4;
if (k &  8) x = (x & 0x00FF00FF) <<  8 | (x & 0xFF00FF00) >>  8;
if (k & 16) x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >> 16;
```

(The last two *and* operations can be omitted.) For k = 31, this operation reverses the bits in a word. For k = 24, it reverses the bytes in a word. For k = 7, it reverses the bits in each byte, without changing the positions of the bytes. For k = 16, it swaps the left and right halfwords of a word, and so on. In general, it moves the bit at position *m* to position *m*   *k*. It can be implemented in hardware very similarly to the way a rotate shifter is usually implemented (five stages of MUX's, with each stage controlled by a bit of the shift amount *k*).

Bit-Reversing Novelties

Item 167 in [HAK] contains rather esoteric expressions for reversing 6-, 7-, and 8-bit integers. Although these expressions are designed for a 36-bit machine, the one for reversing a 6-bit integer works on a 32-bit machine, and those for 7- and 8-bit integers work on a 64-bit machine. These expressions are as follows:

$$\text{6-bit:}\qquad \text{remu}((x * \mathbf{0x00082082})\ \&\ \mathbf{0x01122408}, \mathbf{255})$$

$$\text{7-bit:}\qquad \text{remu}((x * \mathbf{0x40100401})\ \&\ \mathbf{0x442211008}, \mathbf{255})$$

$$\text{8-bit:}\qquad \text{remu}((x * \mathbf{0x202020202})\ \&\ \mathbf{0x10884422010}, \mathbf{1023})$$

The result of all these is a "clean" integer—right-adjusted with no unused high-order bits set.

In all these cases the remu function can instead be rem or mod, because its arguments are positive. The *remainder* function is simply summing the digits of a base-256 or base-1024 number, much like casting out nines. Hence, it can be replaced with a *multiply* and a *shift right*. For example, the 6-bit formula has the following alternative on a 32-bit machine (the multiplication must be modulo 2^{32}):

$$t \leftarrow (x * \mathtt{0x00082082})\ \&\ \mathtt{0x01122408}$$

$$(t * \mathtt{0x01010101}) \overset{u}{\gg} 24$$

These formulas are limited in their utility, because they involve a remaindering operation (20 cycles or more) and/or some multiplications, as well as loading of large constants. The formula immediately above requires ten basic RISC instructions, two of which are *multiply*'s, which amounts to about 20 cycles on a present-day RISC. On the other hand, an adaptation of the code of Figure 7–1 to reverse 6-bit integers requires about 15 instructions, and probably about 9 to 15 cycles, depending on the amount of instruction-level parallelism in the machine. These techniques, however, do give compact code. Below are a few more techniques that might possibly be useful, all for a 32-bit machine. They involve a sort of double application of the idea from [HAK], to extend the technique to 8- and 9-bit integers on a 32-bit machine.

The following is a formula for reversing an 8-bit integer:

$$s \leftarrow (x * \mathtt{0x02020202})\ \&\ \mathtt{0x84422010}$$

$$t \leftarrow (x * 8)\ \&\ \mathtt{0x00000420}$$

$$\text{remu}(s + t, 1023)$$

Here the remu cannot be changed to a *multiply* and *shift*. (You have to work these out, and look at the bit patterns, to see why.)

Here is a similar formula for reversing an 8-bit integer, which is interesting because it can be simplified quite a bit:

$$s \leftarrow (x * \mathtt{0x00020202})\ \&\ \mathtt{0x01044010}$$

$$t \leftarrow (x * \mathtt{0x00080808})\ \&\ \mathtt{0x02088020}$$

$$\text{remu}(s + t, 4095)$$

The simplifications are that the second product is just a *shift left* of the first product, the last mask can be generated from the second with just one instruction (*shift*), and the *remainder* can be replaced by a *multiply* and *shift*. It simplifies to 14 basic RISC instructions, two of which are *multiply*'s:

$$u \leftarrow x * \mathbf{0x00020202}$$

$$m \leftarrow \mathbf{0x01044010}$$

$$s \leftarrow u \,\&\, m$$

$$t \leftarrow (u \ll 2) \,\&\, (m \ll 1)$$

$$(\mathbf{0x01001001} * (s + t)) \overset{u}{\gg} 24$$

The following is a formula for reversing a 9-bit integer:

$$s \leftarrow (x * \mathbf{0x01001001}) \,\&\, \mathbf{0x84108010}$$

$$t \leftarrow (x * \mathbf{0x00040040}) \,\&\, \mathbf{0x00841080}$$

$$\text{remu}(s + t, \mathbf{1023})$$

The second multiplication can be avoided, because the product is equal to the first product shifted right six positions. The last mask is equal to the second mask shifted right eight positions. With these simplifications, this requires 12 basic RISC instructions, including the one *multiply* and one *remainder*. The *remainder* operation must be unsigned, and it cannot be changed to a *multiply* and *shift*.

The reader who studies these marvels will be able to devise similar code for other bit-permuting operations. As a simple (and artificial) example, suppose it is desired to extract every other bit from an 8-bit quantity and compress the four bits to the right. That is, the desired transformation is

```
0000 0000 0000 0000 0000 0000 abcd efgh ==>
0000 0000 0000 0000 0000 0000 0000 bdfh
```

This can be computed as follows:

$$t \leftarrow (x * \mathbf{0x01010101}) \,\&\, \mathbf{0x40100401}$$

$$(t * \mathbf{0x08040201}) \overset{u}{\gg} 27$$

On most machines, the most practical way to do all these operations is by indexing into a table of 1-byte (or 9-bit) integers.

Incrementing a Reversed Integer

The Fast Fourier Transform (FFT) algorithm employs an integer *i* and its bit reversal rev(*i*) in a loop in which *i* is incremented by 1 [PuBr]. Straightforward coding would increment *i* and then compute rev(*i*) on each loop iteration. For small integers, computing rev(*i*) by table lookup is fast and practical. For large integers, however, table lookup is not practical and, as we have seen, computing rev(*i*) requires some 29 instructions.

If table lookup cannot be used, it is more efficient to maintain *i* in both normal and bit-reversed forms, incrementing them both on each loop iteration. This raises the question of how best to increment an integer that is in a register in reversed form. To illustrate, on a 4-bit machine we wish to successively step through the values (in hexadecimal)

```
0, 8, 4, C, 2, A, 6, E, 1, 9, 5, D, 3, B, 7, F.
```

In the FFT algorithm, *i* and its reversal are both some specific number of bits in length, almost certainly less than 32, and they are both right-justified in the register. However, we assume here that *i* is a 32-bit integer. After adding 1 to the reversed 32-bit integer, a *shift right* of the appropriate number of bits will make the result usable by the FFT algorithm (both *i* and rev(*i*) are used to index an array in memory).

The straightforward way to increment a reversed integer is to scan from the left for the first 0-bit, set it to 1, and set all bits to the left of it (if any) to 0's. One way to code this is

```
unsigned x, m;

m = 0x80000000;
x = x ^ m;
if ((int)x >= 0) {
   do {
      m = m >> 1;
      x = x ^ m;
   } while (x < m);
}
```

This executes in three basic RISC instructions if `x` begins with a 0-bit, and four additional instructions for each loop iteration. Because `x` begins with a 0-bit half the time, with 10 (binary) one-fourth of the time, and so on, the average number of instructions executed is approximately

$$\begin{aligned}
&3 \cdot \frac{1}{2} + 7 \cdot \frac{1}{4} + 11 \cdot \frac{1}{8} + 15 \cdot \frac{1}{16} + \ldots \\
&= 4 \cdot \frac{1}{2} + 8 \cdot \frac{1}{4} + 12 \cdot \frac{1}{8} + 16 \cdot \frac{1}{16} + \ldots - 1 \\
&= 4\left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots\right) - 1 \\
&= 7.
\end{aligned}$$

In the second line we added and subtracted 1, with the first 1 in the form 1/2 + 1/4 + 1/8 + 1/16 + ... This makes the series similar to the one analyzed on page 113. The number of instructions executed in the worst case, however, is quite large (131).

If *number of leading zeros* is available, adding 1 to a reversed integer can be done as follows:

$$\begin{aligned}
&\text{First execute:} && s \leftarrow \text{nlz}(\neg x) \\
&\text{and then either:} && x \leftarrow x \oplus (\mathbf{0x80000000} \overset{s}{\gg} s) \\
&\text{or:} && x \leftarrow ((x \ll s) + \mathbf{0x80000000}) \overset{u}{\gg} s
\end{aligned}$$

Either method requires five full RISC instructions and, to properly wrap around from **0xFFFFFFFF** to **0**, requires that the shifts be modulo 64. (These formulas fail in this respect on the Intel x86 machines, because the shifts are modulo 32.)

The rather puzzling one-liner below [Möbi] increments a reversed integer in six basic RISC instructions. It is free of branches and loads but includes an integer division operation. It works for integers of length up to that of the word size of the machine, less 1.

$$revi \leftarrow revi \oplus \left(m - \frac{m}{(i \oplus (i+1)) + 1}\right)$$

To use this, both the non-reversed integer *i* and its reversal *revi* must be available. The variable *m* is the modulus; if we are dealing with *n*-bit integers, then $m = 2^n$. Applying the formula gives the next value of the reversed integer. The non-reversed integer *i* would be incremented separately. The reversed integer is incremented "in place"; that is, it is not shifted to the high-order end of the register, as in the two preceding methods.

A variation is

$$revi \leftarrow revi \oplus \left(m - \frac{m/2}{\neg i \,\&\, (i+1)}\right), \qquad (1)$$

which executes in five instructions if the machine has *and not*, and if *m* is a constant so that the calculation of *m* **/ 2** does not count. It works for integers of length up to that of the word size of the machine. (For full word-size integers, use 0 for the first occurrence of *m* in the formula, and 2^{n-1} for *m* / **2**.)

7–2 Shuffling Bits

Another important permutation of the bits of a word is the "perfect shuffle" operation, which has applications in cryptography. There are two varieties, called the "outer" and "inner" perfect shuffles. They both interleave the bits in the two halves of a word in a manner similar to a perfect shuffle of a deck of 32 cards, but they differ in which card is allowed to fall first. In the outer perfect shuffle, the outer (end) bits remain in the outer positions, and in the inner perfect shuffle, bit 15 moves to the left end of the word (position 31). If the 32-bit word is (where each letter denotes a single bit)

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP,
```

then after the outer perfect shuffle it is

```
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP,
```

and after the inner perfect shuffle it is

```
AaBb CcDd EeFf GgHh IiJj KkLl MmNn OoPp.
```

Assume the word size *W* is a power of 2. Then the outer perfect shuffle operation can be accomplished with basic RISC instructions in $\log_2(W / 2)$ steps, where each step swaps the second and third quartiles of successively smaller pieces [GLS1]. That is, a 32-bit word is transformed as follows:

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP
abcd efgh ABCD EFGH ijkl mnop IJKL MNOP
abcd ABCD efgh EFGH ijkl IJKL mnop MNOP
abAB cdCD efEF ghGH ijIJ klKL mnMN opOP
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP
```

Straightforward code for this is

```
x = (x & 0x0000FF00) << 8 | (x >> 8) & 0x0000FF00 | x &
0xFF0000FF;
x = (x & 0x00F000F0) << 4 | (x >> 4) & 0x00F000F0 | x &
0xF00FF00F;
x = (x & 0x0C0C0C0C) << 2 | (x >> 2) & 0x0C0C0C0C | x &
0xC3C3C3C3;
x = (x & 0x22222222) << 1 | (x >> 1) & 0x22222222 | x &
0x99999999;
```

which requires 42 basic RISC instructions. This can be reduced to 30 instructions, although at an increase from 17 to 21 cycles on a machine with unlimited instruction-level parallelism, by using the *exclusive or* method of exchanging two fields of a register (described on page 47). All quantities are unsigned:

```
t = (x ^ (x >> 8)) & 0x0000FF00; x = x ^ t ^ (t << 8);
t = (x ^ (x >> 4)) & 0x00F000F0; x = x ^ t ^ (t << 4);
t = (x ^ (x >> 2)) & 0x0C0C0C0C; x = x ^ t ^ (t << 2);
t = (x ^ (x >> 1)) & 0x22222222; x = x ^ t ^ (t << 1);
```

The inverse operation, the outer unshuffle, is easily accomplished by performing the swaps in reverse order:

```
t = (x ^ (x >> 1)) & 0x22222222; x = x ^ t ^ (t << 1);
t = (x ^ (x >> 2)) & 0x0C0C0C0C; x = x ^ t ^ (t << 2);
t = (x ^ (x >> 4)) & 0x00F000F0; x = x ^ t ^ (t << 4);
t = (x ^ (x >> 8)) & 0x0000FF00; x = x ^ t ^ (t << 8);
```

Using only the last two steps of either of the above two shuffle sequences shuffles the bits of each byte separately. Using only the last three steps shuffles the bits of each halfword separately, and so on. Similar remarks apply to unshuffling, except by using the *first* two or three steps.

To get the inner perfect shuffle, prepend to these sequences a step to swap the left and right halves of the register:

```
x = (x >> 16) | (x << 16);
```

(or use a *rotate* of 16 bit positions). The unshuffle sequence can be similarly modified by *appending* this line of code.

Altering the transformation to swap the *first* and *fourth* quartiles of successively smaller pieces produces the bit reversal of the inner perfect shuffle.

Perhaps worth mentioning is the special case in which the left half of the word x is all 0. In other words, we want to move the bits in the right half of x to every other bit position—that is, to transform the 32-bit word

```
0000 0000 0000 0000 ABCD EFGH IJKL MNOP
```

to

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N 0O0P.
```

The outer perfect shuffle code can be simplified to do this task in 22 basic RISC instructions. The code below, however, does it in only 19, at no cost in execution time on a machine with unlimited instruction-level parallelism (12 cycles with either method). This code does not require that the left half of word x be initially cleared.

```
x = ((x & 0xFF00) << 8) | (x & 0x00FF);
x = ((x << 4) | x) & 0x0F0F0F0F;
x = ((x << 2) | x) & 0x33333333;
x = ((x << 1) | x) & 0x55555555;
```

Similarly, for the inverse of this “half shuffle” operation (a special case of *compress*; see page 150), the outer perfect unshuffle code can be simplified to do the task in 26 or 29 basic RISC instructions, depending on whether or not an initial *and* operation is required to clear the bits in the odd positions. The code below, however, does it in only 18 or 21 basic RISC instructions, and with less execution time on a machine with unlimited instruction-level parallelism (12 or 15 cycles).

```
x = x & 0x55555555;           // (If required.)
x = ((x >> 1) | x) & 0x33333333;
x = ((x >> 2) | x) & 0x0F0F0F0F;
x = ((x >> 4) | x) & 0x00FF00FF;
x = ((x >> 8) | x) & 0x0000FFFF;
```

7–3 Transposing a Bit Matrix

The transpose of a matrix *A* is a matrix whose columns are the rows of *A* and whose rows are the columns of *A*. Here we consider the problem of computing the transpose of a bit matrix whose elements are single bits that are packed eight per byte, with rows and columns beginning on byte boundaries. This seemingly simple transformation is surprisingly costly in instructions executed.

On most machines it would be very slow to load and store individual bits, mainly due to the code that would be required to extract and (worse yet) to store individual bits. A better method is to partition the matrix into 8×8 submatrices, load each 8×8 submatrix into registers, compute the transpose of the submatrix in registers, and then store the 8×8 result in the appropriate place in the target matrix. Figure 7–5 illustrates the transposition of a bit matrix of size 2×3 bytes. *A, B*, ..., *F* are submatrices of size 8×8 bits. A^T, B^T, ... denote the transpose of submatrices *A, B*, ...

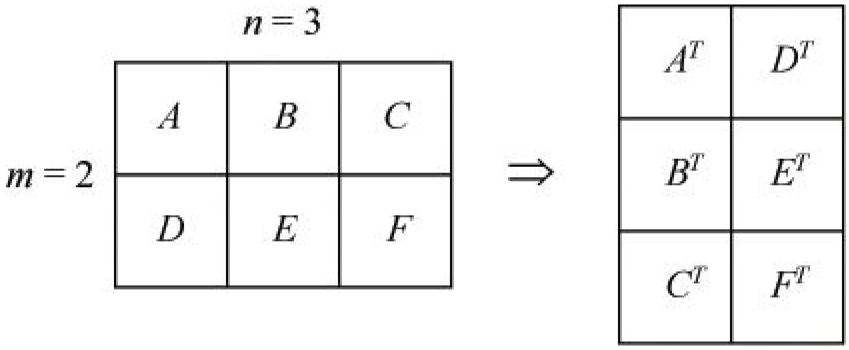


FIGURE 7–5. Transposing a 16×24-bit matrix.

For the purposes of transposing an 8×8 submatrix, it doesn't matter whether the bit matrix is stored in row-major or column-major order; the operations are the same in either event. Assume for discussion that it's in row-major order. Then the first byte of the matrix contains the top row of *A*, the next byte contains the top row of *B*, and so on. If *L* denotes the address of the first byte (top row) of a submatrix, then successive rows of the submatrix are at locations $L + n$, $L + 2n$, ..., $L + 7n$.

For this problem we will depart from the usual assumption of a 32-bit machine and assume the machine has 64-bit general registers. The algorithms are simpler and more easily understood in this way, and it is not difficult to convert them for execution on a 32-bit machine. In fact, a compiler that supports 64-bit integer operations on a 32-bit machine will do the work for you (although probably not as effectively as you can do by hand).

The overall scheme is to load a submatrix with eight *load byte* instructions and pack the bytes left-to-right into a 64-bit register. Then the transpose of the register's contents is computed. Finally, the result is stored in the target area with eight *store byte* instructions.

The transposition of an 8×8 bit matrix is illustrated here, where each character represents a single bit.

```
0123 4567          08go wEMU
89ab cdef          19hp xFNV
ghij klmn          2aiq yGOW
opqr stuv    ⇒     3bjr zHPX
wxyz ABCD          4cks AIQY
EFGH IJKL          5dlt BJRZ
MNOP QRST          6emu CKS$
UVWX YZ$.          7fnv DLT.
```

In terms of doublewords, the transformation to be done is to change the first line to the second line below.

```
01234567 89abcdef ghijklmn opqrstuv wxyzABCD EFGHIJKL MNOPQRST
UVWXYZ$.
08g0wEMU 19hpxFNV 2aiqyGOW 3bjrzHPX 4cksAIQY 5dltBJRZ 6emuCKS$
7fnvDLT.
```

Notice that the bit denoted by 1 moves seven positions to the right, the bit denoted by 2 moves 14 positions to the right, and the bit denoted by 8 moves seven positions to the left. Every bit moves 0, 7, 14, 21, 28, 35, 42, or 49 positions to the left or right. Since there are 56 bits in the doubleword that have to be moved and only 14 different nonzero movement amounts, an average of about four bits can be moved at once, with appropriate masking and shifting. Straightforward code for this follows.

```
y = x & 0x8040201008040201LL         |
    (x & 0x0080402010080402LL) <<  7 |
    (x & 0x0000804020100804LL) << 14 |
    (x & 0x0000008040201008LL) << 21 |
    (x & 0x0000000080402010LL) << 28 |
    (x & 0x0000000000804020LL) << 35 |
    (x & 0x0000000000008040LL) << 42 |
    (x & 0x0000000000000080LL) << 49 |
    (x >>  7) & 0x0080402010080402LL  |
    (x >> 14) & 0x0000804020100804LL  |
    (x >> 21) & 0x0000008040201008LL  |
    (x >> 28) & 0x0000000080402010LL  |
    (x >> 35) & 0x0000000000804020LL  |
    (x >> 42) & 0x0000000000008040LL  |
    (x >> 49) & 0x0000000000000080LL;
```

This executes in 43 instructions on the basic RISC, exclusive of mask generation (which is not important in the application of transposing a large bit matrix, because the masks are loop constants). Rotate shifts do not help. Some of the terms are of the form `(x & mask)<< s`, and some are of the form `(x >> s)& mask`. This reduces the number of masks required; the last seven are repeats of earlier masks. Notice that each mask after the first can be generated from the first with one *shift right* instruction. Because of this, it is a simple matter to write a more compact version of the code that uses a for-loop that is executed seven times.

Another variation is to employ Steele's method of using *exclusive or* to swap bit fields (described on page 47). That technique does not help much in this application. It results in a function that executes in 42 instructions, exclusive of mask generation. The code starts out

```
t = (x ^ (x >> 7)) & 0x0080402010080402LL;
x = x ^ t ^ (t << 7);
```

and there are seven such pairs of lines.

Although there does not seem to be a *really great* algorithm for this problem, the method to be described beats the straightforward method and its variations described above by approximately a factor of 2 on the basic RISC, for the calculation part (not counting loading and storing the submatrices or generating masks). The method gets its power from its high level of bit-parallelism. It would not be a good method if the matrix elements are words. For that, you can't do better than loading each word and storing it where it goes.

First, treat the 8×8-bit matrix as 16 2×2-bit matrices and transpose each of the 16

2×2-bit matrices. Then treat the matrix as four 2×2 submatrices whose elements are 2×2-bit matrices and transpose each of the four 2×2 submatrices. Finally, treat the matrix as a 2×2 matrix whose elements are 4×4-bit matrices and transpose the 2×2 matrix. These transformations are illustrated below [Floyd].

```
0123 4567      082a 4c6e      08go 4cks      08go wEMU
89ab cdef      193b 5d7f      19hp 5dlt      19hp xFNV
ghij klmn      goiq ksmu      2aiq 6emu      2aiq yGOW
opqr stuv      hpjr ltnv      3bjr 7fnv      3bjr zHPX
           ⇒              ⇒              ⇒
wxyz ABCD      wEyG AICK      wEMU AIQY      4cks AIQY
EFGH IJKL      xFzH BJDL      xFNV BJRZ      5dlt BJRZ
MNOP QRST      MUOW QYS$      yGOW CKS$      6emu CKS$
UVWX YZ$.      NVPX RZT.      zHPX DLT.      7fnv DLT.
```

A complete procedure is shown in Figure 7–6. Parameter A is the address of the first byte of an 8×8 submatrix of the source matrix, and parameter B is the address of the first byte of an 8×8 submatrix in the target matrix.

The calculation part of this function executes in 21 instructions. Each of the three major steps is swapping bits, so a version can be written that uses the Steele *exclusive or* bit field swapping device. Using it, the first assignment to x in Figure 7–6 becomes:

```
t = (x ^ (x >> 7)) & 0x00AA00AA00AA00AALL;
x = x ^ t ^ (t << 7);
```

The calculation part of the revised function executes in only 18 instructions, but it has no instruction-level parallelism.

The algorithm of Figure 7–6 runs from fine to coarse granularity, based on the lengths of the groups of bits that are swapped. The method can also be run from coarse to fine granularity. To do this, first treat the 8×8-bit matrix as a 2×2 matrix whose elements are 4×4-bit matrices and transpose the 2×2 matrix. Then, treat each of the four 4×4 submatrices as a 2×2 matrix whose elements are 2×2-bit matrices, and transpose each of the four 2×2 submatrices, and so forth. The code for this is the same as that of Figure 7–6 except for the three assignments that do the bit rearranging being run in reverse order.

```
void transpose8(unsigned char A[8], int m, int n,
                unsigned char B[8]) {
   unsigned long long x;
   int i;

   for (i = 0; i <= 7; i++)     // Load 8 bytes from the
      x = x << 8 | A[m*i];     // input array and pack
                                // them into x.

   x = x & 0xAA55AA55AA55AA55LL        |
      (x & 0x00AA00AA00AA00AALL) <<  7 |
      (x >> 7) & 0x00AA00AA00AA00AALL;
   x = x & 0xCCCC3333CCCC3333LL        |
      (x & 0x0000CCCC0000CCCCLL) << 14 |
      (x >> 14) & 0x0000CCCC0000CCCCLL;
   x = x & 0xF0F0F0F00F0F0F0FLL        |
      (x & 0x00000000F0F0F0F0LL) << 28 |
      (x >> 28) & 0x00000000F0F0F0F0LL;

   for (i = 7; i >= 0; i--) {     // Store result into
      B[n*i] = x; x = x >> 8;}    // output array B.
}
```

FIGURE 7–6. Transposing an 8×8-bit matrix.

As was mentioned, these functions can be modified for execution on a 32-bit machine by using two registers for each 64-bit quantity. If this is done and any calculations that would result in zero are used to make obvious simplifications, the results are that a 32-bit version of the straightforward method described on page 143 runs in 74 instructions (compared to 43 on a 64-bit machine), and a 32-bit version of the function of Figure 7–6 runs in 36 instructions (compared to 21 on a 64-bit machine). Using Steele's bit-swapping technique gives a reduction in instructions executed at the expense of instruction-level parallelism, as in the case of a 64-bit machine.

Transposing a 32×32-Bit Matrix

The same recursive technique that was used for the 8×8-bit matrix can be used for larger matrices. For a 32×32-bit matrix it takes five stages.

The details are quite different from Figure 7–6, because here we assume that the entire 32×32-bit matrix does not fit in the general register space, and we seek a compact procedure that indexes the appropriate words of the bit matrix to do the bit swaps. The algorithm to be described works best if run from coarse to fine granularity.

In the first stage, treat the matrix as four 16×16-bit matrices, and transform it as follows:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} A & C \\ B & D \end{bmatrix}.$$

A denotes the left half of the first 16 words of the matrix, *B* denotes the right half of the first 16 words, and so on. It should be clear that the above transformation can be accomplished by the following swaps:

Right half of word 0 with the left half of word 16,
Right half of word 1 with the left half of word 17,
...
Right half of word 15 with the left half of word 31.

To implement this in code, we will have an index *k* that ranges from 0 to 15. In a loop controlled by *k*, the right half of word *k* will be swapped with the left half of word *k* + 16.

In the second stage, treat the matrix as 16 8×8-bit matrices, and transform it as follows:

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \Rightarrow \begin{bmatrix} A & E & C & G \\ B & F & D & H \\ I & M & K & O \\ J & N & L & P \end{bmatrix}.$$

This transformation can be accomplished by the following swaps:

Bits 0x00FF00FF of word 0 with bits 0xFF00FF00 of word 8,
Bits 0x00FF00FF of word 1 with bits 0xFF00FF00 of word 9, and so on.

This means that bits 0–7 (the least significant eight bits) of word 0 are swapped with bits 8–15 of word 8, and so on. The indexes of the first word in these swaps are *k* = 0, 1, 2, 3, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23. A way to step *k* through these values is

$$k' = (k+9) \,\&\, \neg 8.$$

In the loop controlled by *k*, bits of word *k* are swapped with bits of word *k* + 8.

Similarly, the third stage does the following swaps:

Bits 0x0F0F0F0F of word 0 with bits 0xF0F0F0F0 of word 4,
Bits 0x0F0F0F0F of word 1 with bits 0xF0F0F0F0 of word 5, and so on.

The indexes of the first word in these swaps are *k* = 0, 1, 2, 3, 8, 9, 10, 11, 16, 17, 18, 19, 24, 25, 26, 27. A way to step *k* through these values is

$$k' = (k+5) \,\&\, \neg 4.$$

In the loop controlled by *k*, bits of word *k* are swapped with bits of word *k* + 4.

These considerations are coded rather compactly in the C function shown in Figure 7–7 [GLS1]. The outer loop controls the five stages, with `j` taking on the values 16, 8, 4, 2, and 1. It also steps the mask `m` through the values 0x0000FFFF, 0x00FF00FF, 0x0F0F0F0F, 0x33333333, and 0x55555555. (The code for this, `m = m ^ (m << j)`, is a nice little trick. It does not have an inverse, which is the main reason this code works best for coarse to fine transformations.) The inner loop steps *k* through the values described above. The inner loop body swaps the bits of `a[k]` identified by mask `m` with the bits of `a[k+j]` shifted right `j` and identified by `m`, which is equivalent to the bits of `a[k+j]` identified with the complement of `m`. The code for performing these swaps is an adaptation of the "three *exclusive or*" technique shown on page 46 column (c).

```
void transpose32(unsigned A[32]) {
   int j, k;
   unsigned m, t;

   m = 0x0000FFFF;
   for (j = 16; j != 0; j = j >> 1, m = m ^ (m << j)) {
      for (k = 0; k < 32; k = (k + j + 1) & ~j) {
          t = (A[k] ^ (A[k+j] >> j)) & m;
          A[k] = A[k] ^ t;
          A[k+j] = A[k+j] ^ (t << j);
      }
   }
}
```

FIGURE 7–7. Compact code for transposing a 32×32-bit matrix.

Based on compiling this function with the GNU C compiler to a machine very similar to the basic RISC, this compiles into 31 instructions, with 20 in the inner loop, and 7 in the outer loop but not in the inner loop. Thus, it executes in 4 + 5(7 + 16 · 20) = 1639 instructions. In contrast, if this function were performed using 16 calls on the 8×8 transpose program of Figure 7–6 (modified to run on a 32-bit machine), then it would take 16(101 + 5) = 1696 instructions, assuming the 16 calls are "strung out." This includes five instructions for each function call (observed in compiled code). Therefore, the two methods are, on the surface anyway, very nearly equal in execution time.

On the other hand, for a 64-bit machine the code of Figure 7–7 can easily be modified to transpose a 64×64-bit matrix, and it would take about 4 + 6(7 + 32 · 20) = 3886 instructions. Doing the job with 64 executions of the 8×8 transpose method would take about 64(85 + 5) = 5760 instructions.

The algorithm works in place, and thus if it is used to transpose a larger matrix, additional steps are required to move 32×32-bit submatrices. It can be made to put the result matrix in an area distinct from the source matrix by separating out either the first or last execution of the "for *j*-loop" and having it store the result in the other area.

About half the instructions executed by the function of Figure 7–7 are for loop control, and the function loads and stores the entire matrix five times. Would it be reasonable to reduce this overhead by unrolling the loops? It would, if you are looking for the ultimate in speed, if memory space is not a problem, if your machine's I-fetching can keep up with a large block of straight-line code, and especially if the branches or loads are costly in execution time. The bulk of the program will be the six instructions that do the bit swaps repeated 80 times (5 · 16). In addition, the program will need 32 *load* instructions to load the source matrix and 32 *store* instructions to store the result, for a total of at least 544 instructions.

Figure 7–8 outlines a program in which the unrolling is done by hand. This program is shown as not working in place, but it executes correctly in place, if that is desired, by invoking it with identical arguments. The number of "swap" lines is 80. Our GNU C compiler for the basic RISC machine compiles this into 576 instructions (branch-free, except for the function return), counting prologs and epilogs. This machine does not have the *store multiple* and *load multiple* instructions, but it can save and restore registers two at a time with *store double* and *load double* instructions.

```
#define swap(a0, a1, j, m) t = (a0 ^ (a1 >>j)) & m; \
                           a0 = a0 ^ t; \
                           a1 = a1 ^ (t << j);

void transpose32(unsigned A[32], unsigned B[32]) {
   unsigned m, t;
   unsigned a0, a1, a2, a3, a4, a5, a6, a7,
            a8, a9, a10, a11, a12, a13, a14, a15,
            a16, a17, a18, a19, a20, a21, a22, a23,
            a24, a25, a26, a27, a28, a29, a30, a31;

   a0 = A[ 0]; a1 = A[ 1]; a2 = A[ 2]; a3 = A[ 3];
   a4 = A[ 4]; a5 = A[ 5]; a6 = A[ 6]; a7 = A[ 7];
   . . .
   a28 = A[28]; a29 = A[29]; a30 = A[30]; a31 = A[31];
   m = 0x0000FFFF;
   swap(a0, a16, 16, m)
   swap(a1, a17, 16, m)
   . . .
   swap(a15, a31, 16, m)
   m = 0x00FF00FF;
   swap(a0, a8, 8, m)
   swap(a1, a9, 8, m)
   . . .
   . . .
   swap(a28, a29, 1, m)
   swap(a30, a31, 1, m)

   B[ 0] = a0;   B[ 1] = a1;   B[ 2] = a2;   B[ 3] = a3;
   B[ 4] = a4;   B[ 5] = a5;   B[ 6] = a6;   B[ 7] = a7;
   . . .
   B[28] = a28;  B[29] = a29;  B[30] = a30;  B[31] = a31;
}
```

FIGURE 7–8. Straight-line code for transposing a 32×32-bit matrix.

There is a way to squeeze a little more performance out of this if your machine has a *rotate shift* instruction (either left or right). The idea is to replace all the *swap* operations of Figure 7–8, which take six instructions each, with simpler swaps that do not involve a shift, which take four instructions each (use the swap macro given, with the shifts omitted).

First, rotate right words *A*[16..31] (that is, *A*[*k*] for 16 ≤ *k* ≤ 131) by 16 bit positions. Second, swap the right halves of *A*[0] with *A*[16], *A*[1] with *A*[17], and so on, similarly to the code of Figure 7–8. Third, rotate right words *A*[0..8] and *A*[24..31] by eight bit positions, and then swap the bits indicated by a mask of 0x00FF00FF in words *A*[0] and *A*[8], *A*[1] and *A*[9], and so on, as in the code of Figure 7–8. After five stages of this, you don't quite have the transpose. Finally, you have to rotate left word *A*[1] by one bit position, *A*[2] by two bit positions, and so on (31 instructions). We do not show the code, but the steps are illustrated below for a 4×4-bit matrix.

```
abcd       abcd       abij       abij       aeim       aeim
efgh  ==>  efgh  ==>  efmn  ==>  nefm  ==>  nbfj  ==>  bfjn
ijkl       klij       klcd       klcd       kocg       cgko
mnop       opmn       opgh       hopg       hlpd       dhlp
```

The bit-rearranging part of the program of Figure 7–8 requires 480 instructions (80 swaps at six instructions each). The revised program, using *rotate* instructions, requires 80 swaps at four instructions each, plus 80 *rotate* instructions (16 · 5) for the first five stages, plus a final 31 *rotate* instructions, for a total of 431 instructions. The prolog and epilog code would be unchanged, so using *rotate* instructions in this way saves 49 instructions.

There is another quite different method of transposing a bit matrix: apply three shearing transformations [GLS1]. If the matrix is *n*×*n*, the steps are (1) rotate row *i* to the right *i* bit positions, (2) rotate column *j* upwards (*j* + 1) mod *n* bit positions, (3) rotate row i to the right (*i* + 1) mod *n* bit positions, and (4) reflect the matrix about a horizontal axis through the midpoint. To illustrate, for a 4×4-bit matrix:

```
abcd        abcd        hlpd        dhlp        aeim
efgh        hefg        kocg        cgko        bfjn
      ==>         ==>         ==>         ==>
ijkl        klij        nbfj        bfjn        cgko
mnop        nopm        aeim        aeim        dhlp
```

This method is not quite competitive with the others, because step (2) is costly. (To do it at reasonable cost, rotate upward all columns that rotate by *n/2* or more bit positions by *n* / 2 bit positions [these are columns *n* / 2 – 1 through *n*–2], then rotate certain columns upward *n* / 4 bit positions, and so on.) Steps 1 and 3 require only *n* – 1 instructions each, and step 4 requires no instructions at all if the results are simply stored to the appropriate locations.

If an 8×8-bit matrix is stored in a 64-bit word in the obvious way (top row in the most significant eight bits, and so on), then the matrix transpose operation is equivalent to three outer perfect shuffles or unshuffles [GLS1]. This is a very good way to do it if your machine has shuffle or unshuffle as a single instruction, but it is not a good method on a basic RISC machine.

7–4 *Compress*, or *Generalized Extract*

The APL language includes an operation called *compress*, written B/V, where B is a Boolean vector and V is vector of the same length as B, with arbitrary elements. The result of the operation is a vector consisting of the elements of V for which the corresponding bit in B is 1. The length of the result vector is equal to the number of 1's in B.

Here we consider a similar operation on the bits of a word. Given a mask *m* and a word *x*, the bits of *x* for which the corresponding mask bit is 1 are selected and moved ("compressed") to the right. For example, if the word to be compressed is (where each letter denotes a single bit)

```
abcd efgh ijkl mnop qrst uvwx yzAB CDEF.
```

and the mask is

```
0000 1111 0011 0011 1010 1010 0101 0101,
```

then the result is

```
0000 0000 0000 0000 efgh klop qsuw zBDF.
```

This operation might also be called *generalized extract*, by analogy with the *extract* instruction found on many computers.

We are interested in code for this operation with minimum worst-case execution time, and offer the simple loop of Figure 7–9 as a straw man to be improved upon. This code has no branches in the loop, and it executes in 260 instructions worst case, including the subroutine prolog and epilog.

```
unsigned compress(unsigned x, unsigned m) {
   unsigned r, s, b;    // Result, shift, mask bit.

   r = 0;
   s = 0;
   do {
      b = m & 1;
      r = r | ((x & b) << s);
      s = s + b;
      x = x >> 1;
      m = m >> 1;
   } while (m != 0);
   return r;
}
```

FIGURE 7–9. A simple loop for the *compress* operation.

It is possible to improve on this by repeatedly using the parallel suffix method (see page 97) with the *exclusive or* operation [GLS1]. We will denote the parallel suffix operation by PS-XOR. The basic idea is to first identify the bits of argument *x* that are to be moved right an odd number of bit positions, and move those. (This operation is simplified if *x* is first *and*ed with the mask, to clear out irrelevant bits.) Mask bits are moved in the same way. Next, we identify the bits of *x* that are to be moved an odd multiple of 2 positions (2, 6, 10, and so on), and then we move these bits of *x* and the mask. Next, we identify and move the bits that are to be moved an odd multiple of 4 positions, then those that move an odd multiple of 8, and then those that move 16 bit positions.

Because this algorithm, believed to be original with [GLS1], is a bit difficult to understand, and because it is perhaps surprising that something along these lines can be done at all, we will describe its operation in some detail. Suppose the inputs are

```
x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF,
m = 1000 1000 1110 0000 0000 1111 0101 0101,
    1    1    111
    9    6    333            4444   3 2   1 0
```

where each letter in `x` represents a single bit (with value 0 or 1). The numbers below each 1-bit in the mask `m` denote how far the corresponding bit of `x` must move to the right. This is the number of 0's in `m` to the right of the bit. As mentioned above, it is convenient to first clear out the irrelevant bits of `x`, giving

```
x = a000 e000 ijk0 0000 0000 uvwx 0z0B 0D0F.
```

The plan is to first determine which bits move an odd number of positions (to the right), and move those one bit position. Recall that the PS-XOR operation results in a 1-bit at each position where the number of 1's at and to the right of that position is odd. We wish to identify those bits for which the number of 0's strictly to the right is odd. This can be done by computing `mk = ~m << 1` and performing PS-XOR on the result. This gives

```
mk = 1110 1110 0011 1111 1110 0001 0101 0100,
mp = 1010 0101 1110 1010 1010 0000 1100 1100.
```

Observe that `mk` identifies the bits of `m` that have a 0 immediately to the right, and `mp` sums these, modulo 2, from the right. Thus, `mp` identifies the bits of `m` that have an odd number of 0's to the right.

The bits that will be moved one position are those that are in positions that have an odd number of 0's strictly to the right (identified by `mp`) and that have a 1-bit in the original mask. This is simply `mv = mp & m`:

```
mv = 1000 0000 1110 0000 0000 0000 0100 0100.
```

These bits of `m` can be moved with the assignment

```
m = (m ^ mv) | (mv >> 1);
```

and the same bits of `x` can be moved with the two assignments

```
t = x & mv;
x = (x ^ t) | (t >> 1);
```

(Moving the bits of `m` is simpler because all the selected bits are 1's.) Here the *exclusive or* is turning off bits known to be 1 in `m` and `x`, and the *or* is turning on bits known to be 0 in `m` and `x`. The operations could also, alternatively, both be *exclusive or*, or *subtract* and *add*, respectively. The results, after moving the bits selected by `mv` right one position, are:

```
m = 0100 1000 0111 0000 0000 1111 0011 0011,
x = 0a00 e000 0ijk 0000 0000 uvwx 00zB 00DF.
```

Now we must prepare a mask for the second iteration, in which we identify bits that are to move an odd multiple of 2 positions to the right. Notice that the quantity `mk & ~mp` identifies those bits that have a 0 immediately to the right in the original mask `m`, and those bits that have an even number of 0's to the right in the original mask. These properties apply jointly, although not individually, to the revised mask `m`. (That is to say, `mk` identifies *all* the positions in the revised mask `m` that have a 0 to the immediate right and an even number of 0's to the right.) This is the quantity that, if summed from the right with PS-XOR, identifies those bits that move to the right an odd multiple of 2 positions (2, 6, 10, and so on). Therefore, the procedure is to assign this quantity to `mk` and perform a second iteration of the above steps. The revised value of `mk` is

```
mk = 0100 1010 0001 0101 0100 0001 0001 0000.
```

A complete C function for this operation is shown in Figure 7–10. It does the job in 127 basic RISC instructions (constant)[1], including the subroutine prolog and epilog. Figure 7–11 shows the sequence of values taken on by certain variables at key points in the computation, with the same inputs that were used in the discussion above. Observe that a by-product of the algorithm, in the last value assigned to `m`, is the original `m` with all its 1-bits compressed to the right.

```
unsigned compress(unsigned x, unsigned m) {
   unsigned mk, mp, mv, t;
   int i;

   x = x & m;        // Clear irrelevant bits.
   mk = ~m << 1;     // We will count 0's to right.

   for (i = 0; i < 5; i++) {
      mp = mk ^ (mk << 1);             // Parallel suffix.
      mp = mp ^ (mp << 2);
      mp = mp ^ (mp << 4);
      mp = mp ^ (mp << 8);
      mp = mp ^ (mp << 16);
      mv = mp & m;                    // Bits to move.
      m = m ^ mv | (mv >> (1 << i));  // Compress m.
      t = x & mv;
      x = x ^ t | (t >> (1 << i));    // Compress x.
      mk = mk & ~mp;
   }
   return x;
}
```

FIGURE 7–10. Parallel suffix method for the *compress* operation.

We calculate that the algorithm of Figure 7–10 would execute in 169 instructions on a 64-bit basic RISC, as compared to 516 (worst case) for the algorithm of Figure 7–9.

The number of instructions required by the algorithm of Figure 7–10 can be reduced substantially if the mask `m` is a constant. This can occur in two situations: (1) a call to "`compress(x, m)`" occurs in a loop, in which the value of `m` is not known, but it is a loop constant, and (2) the value of `m` is known, and the code for `compress` is generated in advance, perhaps by a compiler.

Notice that the value assigned to `x` in the loop in Figure 7–10 is not used in the loop for anything other than the assignment to `x`. And `x` is dependent only on itself and variable `mv`. Therefore, the subroutine can be coded with all references to `x` deleted, and the five values computed for `mv` can be saved in variables `mv0`, `mv1`, ..., `mv4`. Then, in situation (1) the function without references to `x` can be placed outside the loop in which "`compress(x, m)`" occurs, and the following statements can be placed in the loop:

```
x = x & m;
t = x & mv0; x = x ^ t | (t >> 1);
t = x & mv1; x = x ^ t | (t >> 2);
t = x & mv2; x = x ^ t | (t >> 4);
t = x & mv3; x = x ^ t | (t >> 8);
t = x & mv4; x = x ^ t | (t >> 16);
```

This is only 21 instructions in the loop (the loading of the constants can be placed outside the loop), a considerable improvement over the 127 required by the full subroutine of Figure 7–10.

```
              x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
              m = 1000 1000 1110 0000 0000 1111 0101 0101
              x = a000 e000 ijk0 0000 0000 uvwx 0z0B 0D0F

i = 0,       mk = 1110 1110 0011 1111 1110 0001 0101 0100
After PS,    mp = 1010 0101 1110 1010 1010 0000 1100 1100
             mv = 1000 0000 1110 0000 0000 0000 0100 0100
              m = 0100 1000 0111 0000 0000 1111 0011 0011
              x = 0a00 e000 0ijk 0000 0000 uvwx 00zB 00DF

i = 1,       mk = 0100 1010 0001 0101 0100 0001 0001 0000
After PS,    mp = 1100 0110 0000 1100 1100 0000 1111 0000
             mv = 0100 0000 0000 0000 0000 0000 0011 0000
              m = 0001 1000 0111 0000 0000 1111 0000 1111
              x = 000a e000 0ijk 0000 0000 uvwx 0000 zBDF
```

```
i = 2,     mk = 0000 1000 0001 0001 0000 0001 0000 0000
After PS, mp = 0000 0111 1111 0000 1111 1111 0000 0000
          mv = 0000 0000 0111 0000 0000 1111 0000 0000
           m = 0001 1000 0000 0111 0000 0000 1111 1111
           x = 000a e000 0000 0ijk 0000 0000 uvwx zBDF

i = 3,     mk = 0000 1000 0000 0001 0000 0000 0000 0000
After PS, mp = 0000 0111 1111 1111 0000 0000 0000 0000
          mv = 0000 0000 0000 0111 0000 0000 0000 0000
           m = 0001 1000 0000 0000 0000 0111 1111 1111
           x = 000a e000 0000 0000 0000 0ijk uvwx zBDF

i = 4,     mk = 0000 1000 0000 0000 0000 0000 0000 0000
After PS, mp = 1111 1000 0000 0000 0000 0000 0000 0000
          mv = 0001 1000 0000 0000 0000 0000 0000 0000
           m = 0000 0000 0000 0000 0001 1111 1111 1111
           x = 0000 0000 0000 0000 000a eijk uvwx zBDF
```

FIGURE 7–11. Operation of the parallel suffix method for the *compress* operation.

In situation (2), in which the value of m is known, the same sort of thing can be done, and further optimization may be possible. It might happen that one of the five masks is 0, in which case one of the five lines shown above can be omitted. For example, mask m1 is 0 if it happens that no bit moves an odd number of positions, and m4 is 0 if no bit moves more than 15 positions, and so on.

As an example, for

```
m = 0101 0101 0101 0101 0101 0101 0101 0101,
```

the calculated masks are

```
mv0 = 0100 0100 0100 0100 0100 0100 0100 0100
mv1 = 0011 0000 0011 0000 0011 0000 0011 0000
mv2 = 0000 1111 0000 0000 0000 1111 0000 0000
mv3 = 0000 0000 1111 1111 0000 0000 0000 0000
mv4 = 0000 0000 0000 0000 0000 0000 0000 0000
```

Because the last mask is 0, in the compiled code situation this compression operation is done in 17 instructions (not counting the loading of the masks). This is not quite as good as the code shown for this operation on page 141 (13 instructions, not counting the loading of masks), which takes advantage of the fact that alternate bits are being selected.

Using *Insert* and *Extract*

If your computer has the *insert* instruction, preferably with immediate values for the operands that identify the bit field in the target register, then in the compiled situation *insert* can often be used to do the *compress* operation with fewer instructions than the methods discussed above. Furthermore, it doesn't tie up registers holding the masks.

The target register is initialized to 0, and then, for each contiguous group of 1's in the mask m, variable x is shifted right to right-justify the next field, and the *insert* instruction is used to insert the bits of x in the appropriate place in the target register. This does the operation in 2n + 1 instructions, where n is the number of fields (groups of consecutive 1's) in the mask. The worst case is 33 instructions, because the maximum number of fields is 16 (which occurs for alternating 1's and 0's).

An example in which the *insert* method uses substantially fewer instructions is `m` = 0x0010084A. Compressing with this mask requires moving bits 1, 2, 4, 8, and 16 positions. Thus, it takes the full 21 instructions for the parallel suffix method, but only 11 instructions for the *insert* method (there are five fields). A more extreme case is `m` = 0x80000000. Here a single bit moves 31 positions, requiring 21 instructions for the parallel suffix method, but only three instructions for the *insert* method and only one instruction (*shift right 31*) if you are not constrained to any particular scheme.

You can also use the *extract* instruction in various simple ways to do the *compress* operation with a known mask in $3n - 2$ instructions, where n is the number of fields in the mask.

Clearly, the problem of compiling optimal code for the *compress* operation with a known mask is a difficult one.

Compress Left

To compress bits to the left, obviously you can reverse the argument `x` and the mask, compress right, and reverse the result. Another way is to compress right and then shift left by pop($\bar{m}$). These might be satisfactory if your computer has an instruction for bit reversal or population count, but if not, the algorithm of Figure 7–10 is easily adapted: Just reverse the direction of all the shifts except the two in the expressions `1 << i` (eight to change).

The BESM-6 computer (ca. 1967) had an instruction for the compress left function ("Pack Bits in A Masked by X") and its inverse ("Unpack ..."), which operated on the machine's 48-bit registers. These instructions are not easy to implement. It is surmised by cryptography experts that their only use was for breaking US codes [Knu8]. The BESM-6 also had the *population count* instruction which, as has been noted, seems to be important to the National Security Agency.

7–5 *Expand*, or *Generalized Insert*

The inverse of the *compress right* function moves bits from the low-order end of a register to positions given by a mask, while keeping the bits in order. For example, expand(0000abcd, 10011010) = a00bc0d0. Thus

$$\text{compress}(\text{expand}(x, m), m) = x.$$

This function has also been called *unpack, scatter*, and *deposit*.

It can be obtained by running the code of Figure 7–10 in reverse [Allen]. To avoid overwriting bits in `x`, it is necessary to move (to the left) the bits that move a large distance first, and to move those that move only one position last. This means that the first five "move" quantities (`mv` in the code) must be computed, saved, and used in the reverse of the order in which they were computed. For many applications this is not a problem, because these applications apply the same mask `m` to large amounts of data, and so they would compute the move quantities in advance and reuse them anyway.

The code is shown in Figure 7–12. It executes approximately 168 basic RISC instructions (constant), including five stores and five loads. A 64-bit version for a 64-bit machine would execute approximately 200 instructions.

For a machine that does not have the *and not* instruction, the MUX operation in the second loop can be coded in one fewer instruction with

```
   x = ((x ^ y) & mv) ^ x;
```

```
unsigned expand(unsigned x, unsigned m) {
  unsigned m0, mk, mp, mv, t;
  unsigned array[5];
  int i;

  m0 = m;         // Save original mask.
  mk = ~m << 1;  // We will count 0's to right.

  for (i = 0; i < 5; i++) {
     mp = mk ^ (mk << 1);             // Parallel suffix.
     mp = mp ^ (mp << 2);
     mp = mp ^ (mp << 4);
     mp = mp ^ (mp << 8);
     mp = mp ^ (mp << 16);
     mv = mp & m;                     // Bits to move.
     array[i] = mv;
     m = (m ^ mv) | (mv >> (1 << i)); // Compress m.
     mk = mk & ~mp;
  }

  for (i = 4; i >= 0; i--) {
     mv = array[i];
     t = x << (1 << i);
     x = (x & ~mv) | (t & mv);
  }
  return x & m0;    // Clear out extraneous bits.
}
```

FIGURE 7–12. Parallel suffix method for the *expand* operation.

7–6 Hardware Algorithms for Compress and Expand

This section gives hardware-oriented algorithms for the *compress right* function and its inverse [Zadeck]. Like the algorithms of the preceding sections, their execution times are proportional to the log of the computer's word size. They are suitable for implementation in hardware, but do not yield fast code if implemented in basic RISC instructions. We simply describe how they work without giving C or machine code.

Compress

To illustrate the operation of the algorithm, we represent each bit of *x* with a letter and consider a specific example mask *m*, shown below.

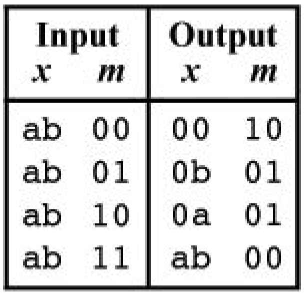
```
   Input x =      abcd efgh ijkl mnop qrst uvwx yzAB CDEF
   Mask m =       0111 1110 0110 1100 1010 1111 0011 0010
```

The algorithm works in $\log_2(W)$ "phases," where *W* is the computer's word size in bits. Each phase operates in parallel on "pockets" of size 2^n bits, for *n* ranging from 1 to $\log_2(W)$. At the end of each phase, each pocket of *x* contains the original pocket of *x* with the bits selected by that pocket of *m* compressed to the right. Each pocket of *m* will contain an integer that is the number of 0-bits in that pocket of the original *m*. This is equal to the number of bits of *x* that are *not* compressed to the right. They are the known leading 0-bits in the pocket of *x*.

In each phase, the algorithm performs the following steps, in parallel, on each pocket of *x* and *m*, where *w* is the pocket size in bits.

1. Set *L* = the left half of the pocket of *x*, extended with *w* / 2 0-bits on the right.
2. Shift *L* (all *w* bits) right by the amount given in the right half of the corresponding pocket of *m*, inserting 0’s on the left. No 1’s will be shifted out on the right, because the maximum shift amount is *w* / 2.
3. Set *R* = *w* / 2 0-bits followed by the right half of the pocket of *x*.
4. Replace the entire *w*-bit pocket of *x* with the *or* of *R* and the shifted *L*.
5. Add the left and right halves of the pocket of *m*, and replace the entire pocket with the sum.

To apply these steps to the first phase (*w* = 2) would require first *and*’ing *x* with *m*, to clear out irrelevant bits of *x*, and complementing *m* so that each bit of *m* is the number of 0-bits in each 1-bit half pocket. It is simpler to make an exception of the first phase, and combine these steps with the first compression operation by applying the logic shown in the table below to each 2-bit pocket of *x* and *m*.



Input		Output	
x	*m*	*x*	*m*
ab	00	00	10
ab	01	0b	01
ab	10	0a	01
ab	11	ab	00

The third line, for example, has *m* = 10 (binary). This means that the left bit of *x* is selected to be part of the result, but the right bit is not. Thus, the left bit (`a`) is compressed to the right. The other bit of *x* is cleared, which ensures that in the final result, all the high-order (not selected) bits will be 0.

Applying this logic to the original *x* and *m* gives:

```
Bit pairs, x = 0bcd ef0g 0j0k mn00 0q0s uvwx 00AB 000E
           m = 0100 0001 0101 0010 0101 0000 1000 1001
```

In the second phase, consider for example the second nibble above (`ef0g`). The quantities *L* = `ef00` and *R* = `000g` are formed. *L* is shifted right by one position (given by the right half of the nibble of *m*), giving `0ef0`. This is *or*’ed with *R*, giving `0efg` as the new value of the nibble. The left and right halves of *m* are added, giving `0001` (no change).

```
Nibbles,   x = 0bcd 0efg 00jk 00mn 00qs uvwx 00AB 000E
           m = 0001 0001 0010 0010 0010 0000 0010 0011
```

Similarly, for the third, fourth, and fifth phases, each byte, halfword, and word of *x* are compressed, and *m* is updated, as follows:

```
Bytes,     x = 00bc defg 0000 jkmn 00qs uvwx 0000 0ABE
           m = 0000 0010 0000 0100 0000 0010 0000 0101
```

```
Halfwords, x = 0000 00bc defg jkmn 0000 000q suvw xABE
           m = 0000 0000 0000 0110 0000 0000 0000 0111


Words,     x = 0000 0000 0000 0bcd efgj kmnq suvw xABE
           m = 0000 0000 0000 0000 0000 0000 0000 1101
```

Upon completion, *m* is an integer that gives the number of known leading 0's in *x*. Subtracting this from the word size gives the number of compressed bits in *x*, which equals the number of 1-bits in the original mask *m*.

The reason this is not a very good algorithm for implementation with basic RISC instructions is that it is hard to shift the half-pockets right by differing amounts. On the other hand, it might possibly be useful on an SIMD machine that has instructions that operate on the pockets of a word in parallel and independently.

Expand

The hardware compression algorithm can be turned into an expansion algorithm by, essentially, running it first forward and then in reverse. As in the algorithms based on the parallel suffix method, the five masks of the hardware compression algorithm are computed, saved, and used in the reverse of the order in which they were computed. Actually, the last mask is not used (nor is it used in the compression algorithm), but an additional one is required (*m0*) that is simply the complement of the original mask. In the forward pass, only the steps for computing the masks need be done; those involving the data *x* can be omitted.

To illustrate, suppose we have

```
Input  x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
Mask   m = 0111 1110 0110 1100 1010 1111 0011 0010
```

Then the result of the expansion should be

```
0nop qrs0 0tu0 vw00 x0y0 zABC 00DE 00F0.
```

The masks are shown below.

```
m0 = 1000 0001 1001 0011 0101 0000 1100 1101
m1 = 0100 0001 0101 0010 0101 0000 1000 1001
m2 = 0001 0001 0010 0010 0010 0000 0010 0011
m3 = 0000 0010 0000 0100 0000 0010 0000 0101
m4 = 0000 0000 0000 0110 0000 0000 0000 0111
```

The integer values of each half of *m4* give the number of 0-bits in the corresponding half of the original mask *m*. In particular, the right half of *m* has seven 0-bits. This means that the seven high-order bits of the right half of *x* do not belong there—they should be in the left half of *x*. Thus, bits 9 through 15 of *x* should be shifted left just enough to put them in the left half of *x*, and higher-order bits of *x* should be shifted left to accommodate them. This can be accomplished by shifting left the entire 32-bit word *x* by seven positions and replacing the left half of *x* with the left half of the shifted quantity. This gives

```
x = hijk lmno pqrs tuvw qrst uvwx yzAB CDEF.
```

In general, the algorithm works with pocket sizes from 32 down to 2, in five phases, using masks *m4* down to *m0*. Each pocket (in parallel) is shifted left, discarding bits that are shifted out on the left, and supplying 0's to vacated positions on the right, so that the shifted quantity is the same length as the pocket from which it came. Then the left half of the pocket is replaced by the left half of the shifted quantity. This will leave "garbage" bits in both halves of the pocket. They will be zeroed-out after the last phase by *and*'ing with the original mask.

Continuing, we treat *m3* as two 16-bit pockets. The left pocket has the integer 4 in its right half, so the left pocket of *x* is shifted left four positions (giving `lmno pqrs tuvw 0000`), and the left half of this replaces the left half of the left pocket in *x*, making the left pocket of *x* = `lmno pqrs`. Performing the same operation on the right 16-bit pocket of *x* gives

```
x = lmno pqrs pqrs tuvw vwxy zABC yzAB CDEF.
```

The next phase uses *m2*, which consists of four 8-bit pockets. Applying it to *x* gives

```
x = mnop pqrs rstu tuvw vwxy zABC BCDE CDEF.
```

The next phase uses *m1*, which consists of eight 4-bit pockets. Applying it to *x* gives

```
x = mnop qrrs sttu vwvw wxxy zABC BCDE DEEF.
```

The last phase uses *m0*, which consists of sixteen 2-bit pockets. Applying it to *x* gives

```
x = mnop qrss stuu vwww xxyy zABC CCDE EEFF.
```

The final step is to *and* this with the original mask to clear irrelevant bits. This gives

```
x = 0nop qrs0 0tu0 vw00 x0y0 zABC 00DE 00F0.
```

The half-pockets of each computed mask contain a count of the number of 0-bits in the corresponding half-pocket of the original mask *m*. Therefore, as an alternative to computing the masks and saving them, the machine could employ circuits for doing a *population count* of the 0's in the half-pockets "on the fly."

7–7 General Permutations, Sheep and Goats Operation

To do general permutations of the bits in a word, or of anything else, a central problem is how to represent the permutation. It cannot be represented very compactly. Because there are 32! permutations of the bits in a 32-bit word, at least $\log_2(32!) = 118$ bits, or three words plus 22 bits, are required to designate one permutation out of the 32!.

One interesting way to represent permutations is closely related to the compression operations discussed in Section 7–4 [GLS1]. Start with the direct method of simply listing the bit position to which each bit moves. For example, for the permutation done by a rotate left of four bit positions, the bit at position 0 (the least significant bit) moves to position 4, 1 moves to 5, ..., 31 moves to 3. This permutation can be represented by the vector of 32 5-bit indexes:

```
00100
00101
 ...
11111
00000
00001
00010
00011
```

Treating that as a bit matrix, the representation we have in mind is its transpose, except reflected about the off diagonal so the top row contains the least significant bits and the result uses little-endian bit numbering. This we store as five 32-bit words in array `p`:

```
p[0] = 1010 1010 1010 1010 1010 1010 1010 1010
p[1] = 1100 1100 1100 1100 1100 1100 1100 1100
p[2] = 0000 1111 0000 1111 0000 1111 0000 1111
p[3] = 0000 1111 1111 0000 0000 1111 1111 0000
p[4] = 0000 1111 1111 1111 1111 0000 0000 0000
```

Each bit of `p[0]` is the least significant bit of the position to which the corresponding bit of `x` moves, each bit of `p[1]` is the next more significant bit, and so on. This is similar to the encoding of the masks denoted by `mv` in the previous section, except that `mv` applies to revised masks in the compress algorithm, not to the original mask.

The compression operation we need compresses to the left all bits marked with 1's in the mask, and compresses to the right all bits marked with 0's.[2] This is sometimes called the "sheep and goats" operation (SAG), or "generalized unshuffle." It can be calculated with

```
SAG(x, m) = compress_left(x, m) | compress(x, ~m).
```

With SAG as a fundamental operation, and a permutation `p` as described above, the bits of a word `x` can be permuted by `p` in the following 15 steps:

```
x    = SAG(x,    p[0]);
p[1] = SAG(p[1], p[0]);
p[2] = SAG(p[2], p[0]);
p[3] = SAG(p[3], p[0]);
p[4] = SAG(p[4], p[0]);

x    = SAG(x,    p[1]);
p[2] = SAG(p[2], p[1]);
p[3] = SAG(p[3], p[1]);
p[4] = SAG(p[4], p[1]);

x    = SAG(x,    p[2]);
p[3] = SAG(p[3], p[2]);
p[4] = SAG(p[4], p[2]);

x    = SAG(x,    p[3]);
p[4] = SAG(p[4], p[3]);

x    = SAG(x,    p[4]);
```

In these steps, SAG is used to perform a stable binary radix sort. Array `p` is used as 32 5-bit keys to sort the bits of `x`. In the first step, all bits of `x` for which `p[0]` = 1 are moved to the left half of the resulting word, and all those for which `p[0]` = 0 are moved to the right half. Other than this, the order of the bits is not changed (that is, the sort is “stable”). Then all the keys that will be used for the next round of sorting are similarly sorted. The sixth line is sorting `x` based on the second least significant bit of the key, and so on.

Similar to the situation of compressing, if a certain permutation `p` is to be used on a number of words `x`, then a considerable savings results by precomputing most of the steps above. The permutation array is revised to

```
p[1] = SAG(p[1], p[0]);
p[2] = SAG(SAG(p[2], p[0]), p[1]);
p[3] = SAG(SAG(SAG(p[3], p[0]), p[1]), p[2]);
p[4] = SAG(SAG(SAG(SAG(p[4], p[0]), p[1]), p[2]), p[3]);
```

and then each permutation is done with

```
x = SAG(x, p[0]);
x = SAG(x, p[1]);
x = SAG(x, p[2]);
x = SAG(x, p[3]);
x = SAG(x, p[4]);
```

A more direct (but perhaps less interesting) way to do general permutations of the bits in a word is to represent a permutation as a sequence of 32 5-bit indexes. The *k*th index is the bit number in the source from which the *k*th bit of the result comes. (This is a “comes from” list, whereas the SAG method uses a “goes to” list.) These could be packed six to a 32-bit word, thus requiring six words to hold all 32 bit indexes. An instruction can be implemented in hardware such as

```
bitgather Rt,Rx,Ri,
```

where register `Rt` is a target register (and also a source), register `Rx` contains the bits to be permuted, and register `Ri` contains six 5-bit indexes (and two unused bits). The operation of the instruction is

$$t \leftarrow (t \ll 6) \mid x_{i_0}x_{i_1}x_{i_2}x_{i_3}x_{i_4}x_{i_5}.$$

In words, the contents of the target register are shifted left six bit positions, and six bits are selected from word *x* and placed in the vacated six positions of *t*. The bits selected are given by the six 5-bit indexes in word *i*, taken in left-to-right order. The bit numbering in the indexes could be either little- or big-endian, and the operation would probably be as described for either type of machine.

To permute a word, use a sequence of six such instructions, all with the same `Rt` and `Rx`, but different index registers. In the first index register of the sequence, only indexes i_4 and i_5 are significant, as the bits selected by the other four indexes are shifted out of the left end of `Rt`.

An implementation of this instruction would most likely allow index values to be repeated, so the instruction can be used to do more than permute bits. It can be used to repeat any selected bit any number of times in the target register. The SAG operation lacks this generality.

It is not unduly difficult to implement this as a fast (e.g., one cycle) instruction. The bit selection circuit consists of six 32:1 MUX’s. If these are built from five stages of 2:1

MUX's in today's technology ($6 \cdot 31 = 186$ MUX's in all), the instruction would be faster than a 32-bit *add* instruction [MD].

Some of the Intel machines have instructions that work much like the bit permutation operation described, but that permute bytes, "words" (16 bits), and "doublewords" (32 bits). These are PSHUFB, PSHUFW, and PSHUFD (Shuffle Packed Bytes/Words/Doublewords).

Permuting bits has applications in cryptography, and the closely related operation of permuting subwords (e.g., permuting the bytes in a word) has applications in computer graphics. Both of these applications are more likely to deal with 64-bit words, or possibly with 128, than with 32. The SAG and *bitgather* methods apply with obvious changes to these larger word sizes.

To encrypt or decrypt a message with the Data Encryption Standard (DES) algorithm requires a large number of permutation-like mappings. First, key generation is done, once per session. This involves 17 permutation-like mappings. The first, called "permuted choice 1," maps from a 64-bit quantity to a 56-bit quantity (it selects the 56 non-parity bits from the key and permutes them). This is followed by 16 permutation-like mappings from 56 bits to 48 bits, all using the same mapping, called "permuted choice 2."

Following key generation, each block of 64 bits in the message is subjected to 34 permutation-like operations. The first and last operations are 64-bit permutations, one being the inverse of the other. There are 16 permutations with repetitions that map 32-bit quantities to 48 bits, all using the same mapping. Finally, there are 16 32-bit permutations, all using the same permutation. The total number of distinct mappings is six. They are all constants and are given in [DES].

DES is obsolete, as it was proved to be insecure in 1998 by the Electronic Frontier Foundation, using special hardware. The National Institute of Standards and Technology (NIST) has endorsed a temporary replacement called Triple DES, which consists of DES run serially three times on each 64-bit block, each time with a different key (that is, the key length is 192 bits, including 24 parity bits). Hence, it takes three times as many permutation operations as does DES to encrypt or decrypt.

The "permanent" replacement for DES and Triple DES, the Advanced Encryption Standard (previously known as the Rijndael algorithm [AES]), involves *no* bit-level permutations. The closest it comes to a permutation is a simple rotation of 32-bit words by a multiple of 8-bit positions. Other encryption methods proposed or in use generally involve far fewer bit-level permutations than DES.

To compare the two permutation methods discussed here, the *bitgather* method has the advantages of (1) simpler preparation of the index words from the raw data describing the permutation, (2) simpler hardware, and (3) more general mappings. The SAG method has the advantages of (1) doing the permutation in five rather than six instructions, (2) having only two source registers in its instruction format (which might fit better in some RISC architectures), (3) scaling better to permute a doubleword quantity, and (4) permuting subwords more efficiently.

Item (3) is discussed in [LSY]. The SAG instruction allows for doing a general permutation of a two-word quantity with two executions of the SAG instruction, a few basic RISC instructions, and two full permutations of single words. The *bitgather* instruction allows for doing it by executing *three* full permutations of single words, plus a few basic RISC instructions. This does not count preprocessing of the permutation to produce new quantities that depend only on the permutation. We leave it to the reader to discover these methods.

Regarding item (4), to permute, for example, the four bytes of a word with *bitgather* requires executing six instructions, the same as for a general bit permutation by *bitgather*. But with SAG it can be done in only two instructions, rather than the five required for a general bit permutation by SAG. The gain in efficiency applies even when the subwords are not a power of 2 in size; the number of steps required is $\log_2 n$, where n is the number of subwords, not counting a possible non-participating group of bits that stays at one end or the other.

[LSY] discusses the SAG and *bitgather* instructions (called "GRP" and "PPERM," respectively), other possible permutation instructions based on networks, and permuting by table lookup.

There is a neat hack to add 1 to the goats—that is, to compute

$$\mathrm{SAG}^{-1}(\mathrm{SAG}(\boldsymbol{x}, \boldsymbol{m}) + 1, \boldsymbol{m})$$

without using the SAG function or its inverse [Knu8]. Here we assume SAG(*x, m*) puts the goats on the right, and the addition does not overflow into the "sheep" field. We leave to the reader the pleasure of discovering this trick.

7–8 Rearrangements and Index Transformations

Many simple rearrangements of the bits in a computer word correspond to even simpler transformations of the coordinates, or indexes, of the bits [GLS1]. These correspondences apply to rearrangements of the elements of any one-dimensional array provided the number of array elements is an integral power of 2. For programming purposes, they are useful primarily when the array elements are a computer word or larger in size.

As an example, the outer perfect shuffle of the elements of an array *A* of size eight, with the result in array *B*, consists of the following moves:

$$\begin{array}{llll} A_0 \rightarrow B_0; & A_1 \rightarrow B_2; & A_2 \rightarrow B_4; & A_3 \rightarrow B_6; \\ A_4 \rightarrow B_1; & A_5 \rightarrow B_3; & A_6 \rightarrow B_5; & A_7 \rightarrow B_7; \end{array}$$

Each *B*-index is the corresponding *A*-index rotated left one position, using a 3-bit rotator. The outer perfect *unshuffle* is, of course, accomplished by rotating *right* each index. Some similar correspondences are shown in Table 7–1. Here n is the number of array elements, "lsb" means least significant bit, and the rotations of indexes are done with a $\log_2 n$-bit rotator.

TABLE 7–1. REARRANGEMENTS AND INDEX TRANSFORMATIONS

Rearrangement	Index Transformation	
	Array Index, or Big-endian Bit Numbering	Little-endian Bit Numbering
Reversal	Complement	Complement
Bit flip, or generalized reversal (page 135)	*Exclusive or* with a constant	*Exclusive or* with a constant
Rotate left k positions	Subtract k (mod n)	Add k (mod n)
Rotate right k positions	Add k (mod n)	Subtract k (mod n)
Outer perfect shuffle	Rotate left one position	Rotate right one position
Outer perfect unshuffle	Rotate right one position	Rotate left one position
Inner perfect shuffle	Rotate left one, then complement lsb	Complement lsb, then rotate right one
Inner perfect unshuffle	Complement lsb, then rotate right	Rotate left one, then complement lsb
Transpose of an 8×8-bit matrix held in a 64-bit word	Rotate (left or right) three positions	Rotate (left or right) three positions
FFT unscramble	Reverse bits	Reverse bits

7–9 An LRU Algorithm

Ever wonder how your computer keeps track of which cache line is the least recently used? Here we describe one such algorithm, known as the *reference matrix* method. It is primarily a hardware algorithm, but it might have application in software.

We won’t go into a long discussion of the intriguing world of caches, but only say that we have in mind the high-speed caches that buffer data between a computer’s main memory and the processor. These caches may get a request for a word every computer cycle, and they should usually respond with the data within a cycle or two, so there is not much time for a complicated algorithm.

A cache contains a copy of a subset of the data in main memory, and the problem we are addressing is: when a cache miss occurs (that is, when a word at a certain address is requested and the data at that address are not in the cache), how does the computer decide which block (or *line*, in cache jargon) to replace with the requested data? Ideally, it should replace the data in the line that will not be referenced for the longest time in the future. But we cannot know the future, so we have to guess. The best guess over a wide variety of application programs seems to be the *least recently used* (LRU) policy. This policy replaces the line that has not been referenced for the longest time.

Caches come in three varieties: *direct-mapped, fully associative*, and *set-associative*. In a direct-mapped cache, certain bits of the address of the load or store instruction directly address a particular cache line. When a miss occurs, there is no question as to what line to replace—it must be the addressed line. There is no need for an LRU or any other guessing policy.

In a fully associative cache, a block from main memory can be placed in *any* cache line. When a load or store is executed, the address is looked up to see if it is in the cache. If not, it is necessary to replace the contents of some line. The machine has complete flexibility in the choice of line to replace. Several strategies have been used (FIFO, random, and LRU are the most common) and, as mentioned above, LRU seems to be the one that most often results in the lowest miss rate. Unfortunately, LRU is the most expensive to implement when there are many lines to consider for replacement.

Often the set-associative organization is chosen. It is a compromise between direct-mapped and fully associative. The designer decides on the degree of associativity, which is usually 2, 4, 8, or 16. The cache is divided into a number of “sets,” each of which contains 2, 4, 8, or 16 lines (typically). The set is directly addressed, using certain bits of the load or store address, but the line within the set must be looked up. The lookup in the set is done much the same as in the case of a fully associative cache. Now, when it is necessary to replace a line, the LRU algorithm need only determine which of the lines within one set is the least recently used, and replace that.

With this brief background, we can describe the reference matrix method. To illustrate, assume the cache is four-way set-associative. This means that there are four lines for which we wish to keep track of the least recently used (referenced). The cache may be fully associative and consist of only four lines, or it may be set-associative with four lines per set.

The reference matrix method employs a square bit matrix of dimension equal to the degree of associativity (in principle; we will modify this statement later). Each associative set has one such matrix. The essence of the method is that when line *i* is referenced, row *i* of the matrix is set to 1’s, and then column *i* is set to 0’s. Figure 7–13 illustrates the changes in the matrix from an initial state to its configuration after a reference to lines 3, 1, 0, 2, 0, 3, and 2, in that order.

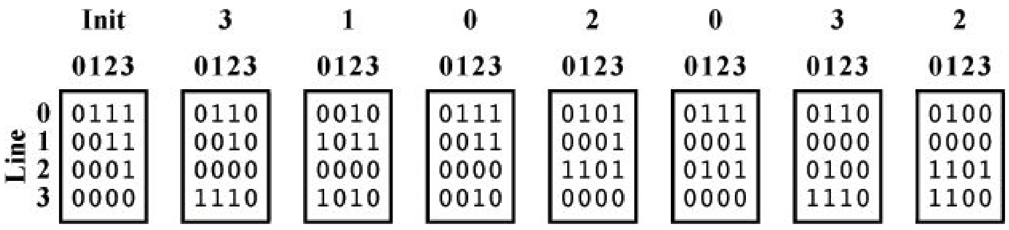


FIGURE 7–13. Illustration of the reference matrix method.

Each matrix has a row containing three 1’s, two 1’s, one 1, and no 1’s. The number of the row with no 1’s is the least recently used line. The number of the row with one 1 is the next least recently used line, and so on. When a cache miss occurs, the machine finds the row with all 0’s and replaces the corresponding line. It then records it as the *most* recently used line by setting its row to all 1’s and its column to all 0’s.

Why does this work? Denoting the matrix by M, the reason it works is that M_{ij} indicates whether or not line i is more recently used than line j. If $M_{ij} = 1$, line i is more recently used than line j, and if $M_{ij} = 0$, line i is not more recently used than line j.

Consider an arbitrary 4×4 matrix for which line 2 is referenced. Then the matrix changes as shown in Figure 7–14. Setting row i to 1’s (except for the element on the main diagonal) is recording that line i is more recently used than line j, for all $j \neq i$. Setting column i to 0’s is recording that line j is not more recently used than line i, for all j. Relations among cache lines other than i are not changed. When all the lines have been referenced, all the “more recently used” relations will be established.

Thus, the reference matrix is antisymmetric and the main diagonal is always all 0’s.

Therefore, only part of the matrix, either the elements above the main diagonal or those below the main diagonal, need be stored in the cache. That is what is done in practice. For an *n*-way associative set, $n(n - 1)/2$ memory bits are required. For $n = 4$, this is six; for $n = 8$, it is 28. Twenty-eight is getting to be a bit large, so the reference matrix method, and in fact the true LRU policy, is not often used for degrees of associativity greater than 8. Instead, there are approximate LRU methods and methods that are not LRU at all.

In software, the LRU policy would probably be implemented with a list of the line numbers (either a simple vector or a linked list). When line *i* is referenced, the list is searched for *i*, and then *i* is moved to the top of the list. The least recently used line number then migrates to the bottom of the list.

That method is relatively slow on references (because of rearranging the list), but fast in deciding which line to replace. Another method, with the opposite speed characteristics, is to have a vector of length equal to the degree of associativity, with position *i* holding both the address that line *i* holds and its “age” (actually “newness”) encoded as an integer. When line *i* is referenced, a single variable that holds the current “age” is incremented, and the resulting value is stored in the vector at position *i*. To find the least recently used line, the vector is searched for the line with the smallest value of “age.” This method fails if the “age” integer overflows.

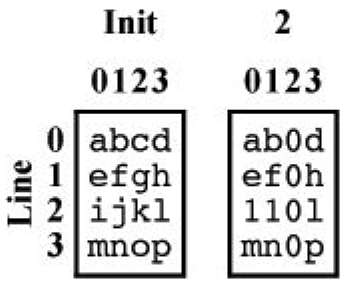


FIGURE 7–14. One step of the reference matrix method.

There might be one “age” integer per associative set, or only one for the whole cache, or in hardware a cycle counter could be used.

The reference matrix method might be useful in software when the degree of associativity is small. For example, suppose an application uses eight-way set-associativity and is to run on a 64-bit machine. Then the reference matrix can be stored in a single 64-bit register. Let the low-order eight bits of the register hold row 0 of the matrix, the next eight bits hold row 1, and so forth. Then when line *i* is referenced, byte *i* of the register should be set to 1’s, and bits $i, i + 8, \ldots, i + 56$ should be cleared. Denoting the register by *m*, this is accomplished as shown here.

$$m \leftarrow m \mid (\text{0xFF} \ll (8 * i))$$

$$m \leftarrow m \,\&\, \neg(\text{0x01010101 01010101} \ll i)$$

This amounts to five or six instructions, plus a few to load constants. To find the least recently used line, search for an all-zero byte (see Section 6–1). The advantage of this method over the other software methods briefly outlined above is that all the work is done in a register.

Exercises

1. Explain the workings of the second Möbius formula (Equation (1), page 139).

2. The perfect outer shuffle operation and its inverse employ the following masks:

$$\begin{aligned} m_0 &= \texttt{0x22222222}, \\ m_1 &= \texttt{0x0C0C0C0C}, \\ m_2 &= \texttt{0x00F000F0}, \text{ and} \\ m_3 &= \texttt{0x0000FF00}. \end{aligned}$$

What is a formula for the general case, m_k? A formula might be useful in situations in which an upper bound on the length of the integers being shuffled is not known in advance, such as in “bignum” applications.

3. Code a function similar to the compress function of Figure 7–9 that does the expand operation.

4. For an *n*-way set-associative cache, what is the theoretical minimum number of bits required to implement the LRU policy? Compare that to the number of bits required for the reference matrix method, for a few small values of *n*.

Chapter 8. Multiplication

8–1 Multiword Multiplication

This can be done with, basically, the traditional grade-school method. But rather than develop an array of partial products, it is more efficient to add each new row, as it is being computed, into a row that will become the product.

If the multiplicand is *m* words, and the multiplier is *n* words, then the product occupies *m* + *n* words (or fewer), whether signed or unsigned.

In applying the grade-school scheme, we would like to treat each 32-bit word as a single digit. This works out well if an instruction that gives the 64-bit product of two 32-bit integers is available. Unfortunately, even if the machine has such an instruction, it is not readily accessible from most high-level languages. In fact, many modern RISC machines do not have this instruction in part *because* it isn't accessible from high-level languages and thus would not be used often. (Another reason is that the instruction would be one of a very few that give a two-register result.)

Our procedure is shown in Figure 8–1. It uses halfwords as the "digits." Parameter `w` gets the result, and `u` and `v` are the multiplier and multiplicand, respectively. Each is an array of halfwords, with the first halfword (`w[0]`, `u[0]`, and `v[0]`) being the least significant digit. This is "little-endian" order. Parameters `m` and `n` are the number of halfwords in `u` and `v`, respectively.

The picture below may help in understanding. There is no relation between `m` and `n`; either may be the larger.

$$\begin{array}{rrrrrrr}
 & u_{m-1}u_{m-2} & \cdots & & \cdots & u_1 & u_0 \\
 & & & \times\ v_{n-1} & \cdots & v_1 & v_0 \\
\hline
w_{m+n-1} & w_{m+n-2} & \cdots & \cdots & \cdots & w_1 & w_0
\end{array}$$

The procedure follows Algorithm M of [Knu2, 4.3.1] but is coded in C and modified to perform signed multiplication. Observe that the assignment to `t` in the upper half of Figure 8–1 cannot overflow, because the maximum value that could be assigned to `t` is $(2^{16} - 1)^2 + 2(2^{16} - 1) = 2^{32} - 1$.

Multiword multiplication is simplest for unsigned operands. In fact, the code of Figure 8–1 performs unsigned multiplication if the "correction" steps (the lines between the three-line comment and the "return" statement) are omitted. An unsigned version can be extended to signed in three ways:

1. Take the absolute value of each input operand, perform unsigned multiplication, and then negate the result if the input operands had different signs.
2. Perform the multiplication using unsigned elementary multiplication, except when multiplying one of the high-order halfwords, in which case use signed × unsigned or signed × signed multiplication.
3. Perform unsigned multiplication and then correct the result somehow.

```
void mulmns(unsigned short w[], unsigned short u[],
   unsigned short v[], int m, int n) {
```

```
   unsigned int k, t, b;
   int i, j;

   for (i = 0; i < m; i++)
      w[i] = 0;

   for (j = 0; j < n; j++) {
      k = 0;
      for (i = 0; i < m; i++) {
         t = u[i]*v[j] + w[i + j] + k;
         w[i + j] = t;       // (I.e., t & 0xFFFF).
         k = t >> 16;
      }
      w[j + m] = k;
   }

   // Now w[] has the unsigned product. Correct by
   // subtracting v*2**16m if u < 0, and
   // subtracting u*2**16n if v < 0.

   if ((short)u[m - 1] < 0) {
      b = 0;                // Initialize borrow.
      for (j = 0; j < n; j++) {
         t = w[j + m] - v[j] - b;
         w[j + m] = t;
         b = t >> 31;
      }
   }
   if ((short)v[n - 1] < 0) {
      b = 0;
      for (i = 0; i < m; i++) {
         t = w[i + n] - u[i] - b;
         w[i + n] = t;
         b = t >> 31;
      }
   }
   return;
 }
```

FIGURE 8–1. Multiword integer multiplication, signed.

The first method requires passing over as many as $m + n$ input halfwords to compute their absolute value. Or, if one operand is positive and one is negative, the method requires passing over as many as $\max(m, n) + m + n$ halfwords to complement the negative input operand and the result. Perhaps more serious, the algorithm would alter its inputs (which we assume are passed by address), which may be unacceptable in some applications. Alternatively, it could allocate temporary space for them, or it could alter them and later change them back. All these alternatives are unappealing.

The second method requires three kinds of elementary multiplication (unsigned × unsigned, unsigned × signed, and signed × signed) and requires sign extension of partial products on the left, with 0's or 1's, making each partial product take longer to compute and add to the running total.

We choose the third method. To see how it works, let u and v denote the values of the two signed integers being multiplied, and let them be of lengths M and N bits, respectively. Then the steps in the upper half of Figure 8–1 erroneously interpret u as an unsigned quantity, having value $u + 2^M u_{M-1}$, where u_{M-1} is the sign bit of u. That is, $u_{M-1} = 1$ if u is negative, and $u_{M-1} = 0$ otherwise. Similarly, the program interprets v as having value $v + 2^N u$.

$N - 1$

The program computes the product of these unsigned numbers—that is, it computes

$$(u + 2^M u_{M-1})(v + 2^N v_{N-1}) = uv + 2^M u_{M-1}v + 2^N v_{N-1}u + 2^{M+N} u_{M-1}v_{N-1}.$$

To get the desired result (uv), we must subtract from the unsigned product the value $2^M u_{M-1}v + 2^N v_{N-1}u$. There is no need to subtract the term $2^{M+N} u_{M-1}v_{N-1}$, because we know that the result can be expressed in $M + N$ bits, so there is no need to compute any product bits more significant than bit position $M + N - 1$. These two subtractions are performed by the steps below the three-line comment in Figure 8–1. They require passing over a maximum of $m + n$ halfwords.

It might be tempting to use the program of Figure 8–1 by passing it an array of fullword integers—that is, by "lying across the interface." Such a program will work on a little-endian machine, but not on a big-endian one. If we had stored the arrays in the reverse order, with `u[0]` being the most significant halfword (and the program altered accordingly), the "lying" program would work on a big-endian machine, but not on a little-endian one.

8–2 High-Order Half of 64-Bit Product

Here we consider the problem of computing the high-order 32 bits of the product of two 32-bit integers. This is the function of our basic RISC instructions *multiply high signed* (`mulhs`) and *multiply high unsigned* (`mulhu`).

For unsigned multiplication, the algorithm in the upper half of Figure 8–1 works well. Rewrite it for the special case $m = n = 2$, with loops unrolled, obvious simplifications made, and the parameters changed to 32-bit unsigned integers.

For signed multiplication, it is not necessary to code the "correction steps" in the lower half of Figure 8–1. These can be omitted if proper attention is paid to whether the intermediate results are signed or unsigned (declaring them to be signed causes the right shifts to be sign-propagating shifts). The resulting algorithm is shown in Figure 8–2. For an unsigned version, simply change all the `int` declarations to `unsigned`.

The algorithm requires 16 basic RISC instructions in either the signed or unsigned version, four of which are multiplications.

```
int mulhs(int u, int v) {
   unsigned u0, v0, w0;
   int u1, v1, w1, w2, t;

   u0 = u & 0xFFFF; u1 = u >> 16;
   v0 = v & 0xFFFF; v1 = v >> 16;
   w0 = u0*v0;
   t  = u1*v0 + (w0 >> 16);
   w1 = t & 0xFFFF;
   w2 = t >> 16;
   w1 = u0*v1 + w1;
   return u1*v1 + w2 + (w1 >> 16);
}
```

FIGURE 8–2. *Multiply high signed*.

8–3 High-Order Product Signed from / to Unsigned

Assume that the machine can readily compute the high-order half of the 64-bit product of two *unsigned* 32-bit integers, but we wish to perform the corresponding operation on *signed* integers. We could use the procedure of Figure 8–2, but that requires four multiplications; the procedure to be given [BGN] is much more efficient than that.

The analysis is a special case of that done to convert Knuth's Algorithm M from an unsigned to a signed multiplication routine (Figure 8–1). Let x and y denote the two 32-bit signed integers that we wish to multiply together. The machine will interpret x as an *unsigned* integer, having the value $x + 2^{32}x_{31}$, where x_{31} is the most significant bit of x (that is, x_{31} is the integer 1 if x is negative, and 0 otherwise). Similarly, y under unsigned interpretation has the value $y + 2^{32}y_{31}$.

Although the result we want is the high-order 32 bits of xy, the machine computes

$$(x + 2^{32}x_{31})(y + 2^{32}y_{31}) = xy + 2^{32}(x_{31}y + y_{31}x) + 2^{64}x_{31}y_{31}.$$

To get the desired result, we must subtract from this the quantity $2^{32}(x_{31}y + y_{31}x) + 2^{64}x_{31}y_{31}$. Because we know that the result can be expressed in 64 bits, we can perform the arithmetic modulo 2^{64}. This means that we can safely ignore the last term, and compute the signed high-order product as shown below (seven basic RISC instructions).

```
$p \leftarrow \text{mulhu}(x, y)$                      // *multiply high unsigned* instruction.
$t_1 \leftarrow (x \overset{s}{\gg} 31) \,\&\, y$          // $t_1 = x_{31}y$.
$t_2 \leftarrow (y \overset{s}{\gg} 31) \,\&\, x$          // $t_2 = y_{31}x$.                    (1)
$p \leftarrow p - t_1 - t_2$                         // $p$ = desired result.
```

Unsigned from Signed

The reverse transformation follows easily. The resulting program is the same as (1), except with the first instruction changed to *multiply high signed* and the last operation changed to $p \leftarrow p + t_1 + t_2$.

8–4 Multiplication by Constants

It is nearly a triviality that one can multiply by a constant with a sequence of *shift left* and *add* instructions. For example, to multiply x by 13 (binary 1101), one can code

$$t_1 \leftarrow x \ll 2$$
$$t_2 \leftarrow x \ll 3$$
$$r \leftarrow t_1 + t_2 + x$$

where r gets the result.

In this section, left shifts are denoted by multiplication by a power of 2, so the above plan is written $r \leftarrow \mathbf{8}x + 4x + x$, which is intended to show four instructions on the basic RISC and most machines.

What we want to convey here is that there is more to this subject than meets the eye. First of all, there are other considerations besides simply the number of *shift*'s and *add*'s required to do a multiplication by a given constant. To illustrate, below are two plans for multiplying by 45 (binary 101101).

$$\begin{array}{ll}
t \leftarrow 4x & t_1 \leftarrow 4x \\
r \leftarrow x + t & t_2 \leftarrow 8x \\
t \leftarrow 2t & t_3 \leftarrow 32x \\
r \leftarrow r + t & r \leftarrow t_1 + x \\
t \leftarrow 4t & t_3 \leftarrow t_3 + t_2 \\
r \leftarrow r + t & r \leftarrow r + t_3
\end{array}$$

The plan on the left uses a variable *t* that holds *x* shifted left by a number of positions that corresponds to a 1-bit in the multiplier. Each shifted value is obtained from the one before it. This plan has these advantages:

- It requires only one working register other than the input *x* and the output *r*.
- Except for the first two, it uses only 2-address instructions.
- The shift amounts are relatively small.

The same properties are retained when the plan is applied to any multiplier.

The scheme on the right does all the *shift*'s first, with *x* as the operand. It has the advantage of increased parallelism. On a machine with sufficient instruction-level parallelism, the scheme on the right executes in three cycles, whereas the scheme on the left, running on a machine with unlimited parallelism, requires four.

In addition to these details, it is nontrivial to find the minimum number of operations to accomplish multiplication by a constant, where by an "operation" we mean an instruction from a typical computer's set of *add* and *shift* instructions. In what follows, we assume this set consists of *add, subtract, shift left* by any constant amount, and *negate*. We assume the instruction format is three-address. However, the problem is no easier if one is restricted to only *add* (adding a number to itself, and then adding the sum to itself, and so on, accomplishes a shift left of any amount), or if one augments the set by instructions that combine a *left shift* and an *add* into one instruction (that is, such an instruction computes $z \leftarrow x + (y \ll n)$). We also assume that only the least-significant 32 bits of the product are wanted.

The first improvement to the basic binary decomposition scheme suggested above is to use *subtract* to shorten the sequence when the multiplier contains a group of three or more consecutive 1-bits. For example, to multiply by 28 (binary 11100), we can compute $32x - 4x$ (three instructions) rather than $16x + 8x + 4x$ (five instructions). On two's-complement machines, the result is correct (modulo 2^{32}) even if the intermediate result of $32x$ overflows.

To multiply by a constant *m* with the basic binary decomposition scheme (using only *shift*'s and *add*'s) requires

$$2\text{pop}(m) - 1 - \delta$$

instructions, where $\delta = 1$ if m ends in a 1-bit (is odd), and $\delta = 0$ otherwise. If *subtract* is also used, it requires

$$4g(m) + 2s(m) - 1 - \delta$$

instructions, where $g(m)$ is the number of groups of two or more consecutive 1-bits in m, $s(m)$ is the number of “singleton” 1-bits in m, and δ has the same meaning as before.

For a group of size 2, it makes no difference which method is used.

The second improvement is to treat specially groups that are separated by a single 0-bit. For example, consider $m = 55$ (binary 110111). The group method calculates this as $(64x - 16x) + (8x - x)$, which requires six instructions. Calculating it as $64x - 8x - x$, however, requires only four. Similarly, we can multiply by binary 110111011 as illustrated by the formula $512x - 64x - 4x - x$ (six instructions).

The formulas above give an upper bound on the number of operations required to multiply a variable x by any given number m. Another bound can be obtained based on the size of m in bits—that is, on $n = \quad \log_2 m \quad + 1$.

Theorem. *Multiplication of a variable x by an n-bit constant m, m* ≥ 1, *can be accomplished with at most n instructions of the type add, subtract, and shift left by any given amount*.

Proof. (Induction on n.) Multiplication by 1 can be done in 0 instructions, so the theorem holds for $n = 1$. For $n > 1$, if m ends in a 0-bit, then multiplication by m can be accomplished by multiplying by the number consisting of the left $n - 1$ bits of m (that is, by $m / 2$), in $n - 1$ instructions, followed by a *shift left* of the result by one position. This uses n instructions altogether.

If m ends in binary 01, then mx can be calculated by multiplying x by the number consisting of the left $n - 2$ bits of m, in $n - 2$ instructions, followed by a *left shift* of the result by 2, and an *add* of x. This requires n instructions altogether.

If m ends in binary 11, then consider the cases in which it ends in 0011, 0111, 1011, and 1111. Let t be the result of multiplying x by the left $n - 4$ bits of m. If m ends in 0011, then $mx = 16t + 2x + x$, which requires $(n - 4) + 4 = n$ instructions. If m ends in 0111, then $mx = 16t + 8x - x$, which requires n instructions. If m ends in 1111, then $mx = 16t + 16x - x$, which requires n instructions. The remaining case is that m ends in 1011.

It is easy to show that mx can be calculated in n instructions if m ends in 001011, 011011, or 111011. The remaining case is 101011.

This reasoning can be continued, with the “remaining case” always being of the form 101010...10101011. Eventually, the size of m will be reached, and the only remaining case is the number 101010...10101011. This n-bit number contains $n / 2 + 1$ 1-bits. By a previous observation, it can multiply x with $2(n / 2 + 1) - 2 = n$ instructions.

Thus, in particular, multiplication by any 32-bit constant can be done in at most 32 instructions, by the method described above. By inspection, it is easily seen that for n even, the n-bit number 101010...101011 requires n instructions, and for n odd, the n-bit number 1010101...010110 also requires n instructions, so the bound is tight.

The methodology described so far is not difficult to work out by hand or to incorporate into an algorithm such as might be used in a compiler; but such an algorithm would not always produce the best code, because further improvement is sometimes possible. This can result from factoring the multiplier m or some intermediate quantity along the way of computing *mx*. For example, consider again *m* = 45 (binary 101101). The methods described above require six instructions. Factoring 45 as 5 · 9, however, gives a four-instruction solution:

$$t \leftarrow 4x + x$$
$$r \leftarrow 8t + t$$

Factoring can be combined with the binary decomposition methods. For example, multiplication by 106 (binary 1101010) requires seven instructions by binary decomposition, but writing it as 7 · 15 + 1 leads to a five-instruction solution. For large constants, the smallest number of instructions that accomplish the multiplication may be substantially fewer than the number obtained by the simple binary decomposition methods described. For example, *m* = 0xAAAAAAAB requires 32 instructions by binary decomposition, but writing this value as 2 · 5 · 17 · 257 · 65537 + 1 gives a ten-instruction solution. (Ten instructions is probably not typical of large numbers. The factorization reflects the simple bit pattern of alternate 1's and 0's.)

There does not seem to be a simple formula or procedure that determines the smallest number of *shift* and *add* instructions that accomplishes multiplication by a given constant *m*. A practical search procedure is given in [Bern], but it does not always find the minimum. Exhaustive search methods to find the minimum can be devised, but they are quite expensive in either space or time. (See, for example, the tree structure of Figure 15 in [Knu2, 4.6.3].)

This should give an idea of the combinatorics involved in this seemingly simple problem. Knuth [Knu2, 4.6.3] discusses the closely related problem of computing a^m using a minimum number of multiplications. This is analogous to the problem of multiplying by *m* using only addition instructions.

Exercises

1. Show that for a 32×32 64 bit multiplication, the low-order 32 bits of the product are the same whether the operands are interpreted as signed or unsigned integers.

2. Show how to modify the `mulhs` function (Figure 8–2) so that it calculates the low-order half of the 64-bit product, as well as the high-order half. (Just show the calculation, not the parameter passing.)

3. Multiplication of complex numbers is defined by

$$(a + bi)(c + di) = ac - bd + (ad + bc)i.$$

This can be done with only three multiplications.[1] Let

$$\begin{aligned} p &= ac, \\ q &= bd, \text{ and} \\ r &= (a+b)(c+d). \end{aligned}$$

Then the product is given by

$$p - q + (r - p - q)i,$$

which the reader can easily verify.

Code a similar method to obtain the 64-bit product of two 32-bit unsigned integers using only three multiplication instructions. Assume the machine's *multiply* instruction produces the 32 low-order bits of the product of two 32-bit integers (which are the same for signed and unsigned multiplication).

Chapter 9. Integer Division

9–1 Preliminaries

This chapter and the following one give a number of tricks and algorithms involving “computer division” of integers. In mathematical formulas we use the expression *x* / *y* to denote ordinary rational division, $x \div y$ to denote signed computer division of integers (truncating toward 0), and $x \overset{u}{\div} y$ to denote unsigned computer division of integers. Within C code, `x/y`, of course, denotes computer division, unsigned if either operand is unsigned, and signed if both operands are signed.

Division is a complex process, and the algorithms involving it are often not very elegant. It is even a matter of judgment as to just how signed integer division should be defined. Most high-level languages and most computer instruction sets define the result to be the rational result truncated toward 0. This and two other possibilities are illustrated below.

		truncating	modulus	floor
7÷3	=	2 rem 1	2 rem 1	2 rem 1
(−7)÷3	=	−2 rem −1	-3 rem 2	−3 rem 2
7÷−(−3)	=	−2 rem 1	−2 rem 1	−3 rem −2
(−7)÷− (−3)	=	2 rem −1	3 rem 2	2 rem −1

The relation *dividend* = *quotient* * *divisor* + *remainder* holds for all three possibilities. We define “modulus” division by requiring that the remainder be nonnegative.[1] We define “floor” division by requiring that the quotient be the floor of the rational result. For positive divisors, modulus and floor division are equivalent. A fourth possibility, seldom used, rounds the quotient to the nearest integer.

One advantage of modulus and floor division is that most of the tricks simplify. For example, division by 2^n can be replaced by a *shift right signed* of *n* positions, and the remainder of dividing *x* by 2^n is given by the logical *and* of *x* and $2^n - 1$. I suspect that modulus and floor division more often give the result you want. For example, suppose you are writing a program to graph an integer-valued function, and the values range from *imin* to *imax*. You want to set up the extremes of the ordinate to be the smallest multiples of 10 that include *imin* and *imax*. Then the extreme values are simply (*imin* ÷ 10) * 10 and ((*imax* + 9) ÷ 10) * 10 if modulus or floor division is used. If conventional division is used, you must evaluate something like:

```
if (imin >= 0)    gmin = (imin/10)*10;
else              gmin = ((imin - 9)/10)*10;
if (imax >= 0)    gmax = ((imax + 9)/10)*10;
else              gmax = (imax/10)*10;
```

Besides the quotient being more useful with modulus or floor division than with truncating division, we speculate that the nonnegative remainder is probably wanted more often than a remainder that can be negative.

It is hard to choose between modulus and floor division, because they differ only when the divisor is negative, which is unusual. Appealing to existing high-level languages does not help, because they almost universally use truncating division for `x/y` when the operands are signed integers. A few give floating-point numbers, or rational numbers, for the result. Looking at remainders, there is confusion. In Fortran 90, the MOD function gives the remainder of truncating division and MODULO gives the remainder of floor division (which can be negative). Similarly, in Common Lisp and ADA, REM is the remainder of truncating division, and MOD is the remainder of floor division. In PL/I, MOD is always nonnegative (it is the remainder of modulus division). In Pascal, A mod B is defined only for B > 0, and then it is the nonnegative value (the remainder of either modulus or floor division).

Anyway, we cannot change the world even if we knew how we wanted to change it,[2] so in what follows we will use the usual definition (truncating) for $x \div y$.

A nice property of truncating division is that it satisfies

$$(-n) \div d = n \div (-d) = -(n \div d), \text{ for } d \neq 0.$$

Care must be exercised when applying this to transform programs, because if n or d is the maximum negative number, $-n$ or $-d$ cannot be represented in 32 bits. The operation $(-2^{31}) \div (-1)$ is an overflow (the result cannot be expressed as a signed quantity in two's-complement notation), and on most machines the result is undefined or the operation is suppressed.

Signed integer (truncating) division is related to ordinary rational division by

$$n \div d = \begin{cases} \lfloor n/d \rfloor, & \text{if } d \neq 0, nd \geq 0, \\ \lceil n/d \rceil, & \text{if } d \neq 0, nd < 0. \end{cases} \qquad (1)$$

Unsigned integer division—that is, division in which both n and d are interpreted as unsigned integers—satisfies the upper portion of (1).

In the discussion that follows, we make use of the following elementary properties of arithmetic, which we don't prove here. See [Knu1] and [GKP] for interesting discussions of the floor and ceiling functions.

THEOREM D1. *For x real, k an integer,*

$$\begin{array}{ll}
\lfloor x \rfloor = -\lceil -x \rceil & \lceil x \rceil = -\lfloor -x \rfloor \\
x - 1 < \lfloor x \rfloor \leq x & x \leq \lceil x \rceil < x + 1 \\
\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1 & \lceil x \rceil - 1 < x \leq \lceil x \rceil \\
x \geq k \Leftrightarrow \lfloor x \rfloor \geq k & x \leq k \Leftrightarrow \lceil x \rceil \leq k \\
x > k \Rightarrow \lfloor x \rfloor \geq k & x < k \Rightarrow \lceil x \rceil \leq k \\
x \leq k \Rightarrow \lfloor x \rfloor \leq k \Rightarrow x < k + 1 & x \geq k \Rightarrow \lceil x \rceil \geq k \Rightarrow x > k - 1 \\
x < k \Leftrightarrow \lfloor x \rfloor < k & x > k \Leftrightarrow \lceil x \rceil > k
\end{array}$$

THEOREM D2. *For n, d integers, d* > 0,

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n - d + 1}{d} \right\rceil \quad and \quad \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n + d - 1}{d} \right\rfloor.$$

If d < 0:

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d-1}{d} \right\rceil \quad and \quad \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d+1}{d} \right\rfloor.$$

THEOREM D3. *For x real, d an integer* > 0:

$$\lfloor \lfloor x \rfloor / d \rfloor = \lfloor x/d \rfloor \quad and \quad \lceil \lceil x \rceil / d \rceil = \lceil x/d \rceil.$$

COROLLARY. *For a, b real, b* ≠ 0, *d an integer* > 0,

$$\left\lfloor \left\lfloor \frac{a}{b} \right\rfloor / d \right\rfloor = \left\lfloor \frac{a}{bd} \right\rfloor \quad and \quad \left\lceil \left\lceil \frac{a}{b} \right\rceil / d \right\rceil = \left\lceil \frac{a}{bd} \right\rceil.$$

THEOREM D4. *For n, d integers, d* ≠ 0, *and x real*,

$$\left\lfloor \frac{n}{d} + x \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \text{ if } 0 \le x < \left|\frac{1}{d}\right|, \quad and \quad \left\lceil \frac{n}{d} + x \right\rceil = \left\lceil \frac{n}{d} \right\rceil \text{ if } -\left|\frac{1}{d}\right| < x \le 0.$$

In the theorems below, rem(*n, d*) denotes the remainder of *n* divided by *d*. For negative *d*, it is defined by rem(*n*, –*d*) = rem(*n, d*), as in truncating and modulus division. We do not use rem(*n, d*) with *n* < 0. Thus, for our use, the remainder is always nonnegative.

THEOREM D5. *For n* ≥ 0, *d* ≠0,

$$\text{rem}(2n, d) = \begin{cases} 2\text{rem}(n, d) & or \\ 2\text{rem}(n, d) - |d|, \end{cases} \quad and \quad \text{rem}(2n+1, d) = \begin{cases} 2\text{rem}(n, d) + 1 & or \\ 2\text{rem}(n, d) - |d| + 1 \end{cases}$$

(*whichever value is greater than or equal to* 0 *and less than |d|*).

THEOREM D6. *For n* ≥ 0, *d* ≠ 0,

$$\text{rem}(2n, 2d) = 2\text{rem}(n, d).$$

Theorems D5 and D6 are easily proved from the basic definition of remainder—that is, that for some integer *q* it satisfies

$$n = qd + \text{rem}(n, d) \text{ with } 0 \le \text{rem}(n, d) < |d|,$$

provided *n* ≥ 0 and *d* ≠ 0 (*n* and *d* can be non-integers, but we will use these theorems only for integers).

9–2 Multiword Division

As in the case of multiword multiplication, multiword division can be done by the traditional grade-school method. The details, however, are surprisingly complicated. Figure 9–1 is Knuth's Algorithm D [Knu2, 4.3.1], coded in C. The underlying form of division it uses is $32 \overset{u}{\div} 16 \Rightarrow 32$.. (Actually, the quotient of these underlying division operations is at most 17 bits long.)

```
int divmnu(unsigned short q[], unsigned short r[],
     const unsigned short u[], const unsigned short v[],
     int m, int n) {

   const unsigned b = 65536;   // Number base (16 bits).
   unsigned short *un, *vn;    // Normalized form of u, v.
   unsigned qhat;              // Estimated quotient digit.
   unsigned rhat;              // A remainder.
   unsigned p;                 // Product of two digits.
   int s, i, j, t, k;

   if (m < n || n <= 0 || v[n-1] == 0)
      return 1;                // Return if invalid param.

   if (n == 1) {                        // Take care of
      k = 0;                            // the case of a
      for (j = m - 1; j >= 0; j--) {  // single-digit
         q[j] = (k*b + u[j])/v[0];    // divisor here.
         k = (k*b + u[j]) - q[j]*v[0];
      }
      if (r != NULL) r[0] = k;
      return 0;
   }

   // Normalize by shifting v left just enough so that
   // its high-order bit is on, and shift u left the
   // same amount. We may have to append a high-order
   // digit on the dividend; we do that unconditionally.

   s = nlz(v[n-1]) - 16;       // 0 <= s <= 16.
   vn = (unsigned short *)alloca(2*n);
   for (i = n - 1; i > 0; i--)
      vn[i] = (v[i] << s) | (v[i-1] >> 16-s);
   vn[0] = v[0] << s;

   un = (unsigned short *)alloca(2*(m + 1));
   un[m] = u[m-1] >> 16-s;
   for (i = m - 1; i > 0; i--)
      un[i] = (u[i] << s) | (u[i-1] >> 16-s);
   un[0] = u[0] << s;
   for (j = m - n; j >= 0; j--) {    // Main loop.
      // Compute estimate qhat of q[j].
      qhat = (un[j+n]*b + un[j+n-1])/vn[n-1];
      rhat = (un[j+n]*b + un[j+n-1]) - qhat*vn[n-1];
again:
      if (qhat >= b || qhat*vn[n-2] > b*rhat + un[j+n-2])
      { qhat = qhat - 1;
        rhat = rhat + vn[n-1];
        if (rhat < b) goto again;
      }

      // Multiply and subtract.
      k = 0;
      for (i = 0; i < n; i++) {
         p = qhat*vn[i];
         t = un[i+j] - k - (p & 0xFFFF);
         un[i+j] = t;
         k = (p >> 16) - (t >> 16);
      }
      t = un[j+n] - k;
      un[j+n] = t;

      q[j] = qhat;          // Store quotient digit.
      if (t < 0) {          // If we subtracted too
```

```
         q[j] = q[j] - 1;   // much, add back.
         k = 0;
         for (i = 0; i < n; i++) {
            t = un[i+j] + vn[i] + k;
            un[i+j] = t;
            k = t >> 16;
         }
         un[j+n] = un[j+n] + k;
       }
    } // End j.
    // If the caller wants the remainder, unnormalize
    // it and pass it back.
    if (r != NULL) {
       for (i = 0; i < n; i++)
          r[i] = (un[i] >> s) | (un[i + 1] << 16-s);
    }
    return 0;
}
```

FIGURE 9–1. Multiword integer division, unsigned.

The algorithm processes its inputs and outputs a halfword at a time. Of course, we would prefer to process a fullword at a time, but it seems that such an algorithm would require an instruction that does $64 \overset{u}{\div} 32 \Rightarrow 32$ division. We assume here that either the machine does not have that instruction or it is hard to access from our high-level language. Although we generally assume the machine has $32 \overset{u}{\div} 32 \Rightarrow 32$ division, for this problem $32 \overset{u}{\div} 16 \Rightarrow 16$ suffices.

Thus, for this implementation of Knuth's algorithm, the base `b` is 65536. See [Knu2] for most of the explanation of this algorithm.

The dividend `u` and the divisor `v` are in "little-endian" order—that is, `u[0]` and `v[0]` are the least significant digits. (The code works correctly on both big- and little-endian machines.) Parameters `m` and `n` are the number of halfwords in `u` and `v`, respectively (Knuth defines `m` to be the length of the quotient). The caller supplies space for the quotient `q` and, optionally, for the remainder `r`. The space for the quotient must be at least `m - n + 1` halfwords, and for the remainder, `n` halfwords. Alternatively, a value of `NULL` can be given for the address of the remainder to signify that the remainder is not wanted.

The algorithm requires that the most significant digit of the divisor, `v[n - 1]`, be nonzero. This simplifies the normalization steps and helps to ensure that the caller has allocated sufficient space for the quotient. The code checks that `v[n - 1]` is nonzero, and also the requirements that `n ≥ 1` and `m ≥ n`. If any of these conditions are violated, it returns with an error code (return value 1).

After these checks, the code performs the division for the simple case in which the divisor is of length 1. This case is not singled out for speed; the rest of the algorithm requires that the divisor be of length 2 or more.

If the divisor is of length 2 or more, the algorithm normalizes the divisor by shifting it left just enough so that its high-order bit is 1. The dividend is shifted left the same amount, so the quotient is not changed by these shifts. As explained by Knuth, these steps are necessary to make it easy to guess each quotient digit with good accuracy. The *number of leading zeros* function, nlz(x), is used to determine the shift amount.

In the normalization steps, new space is allocated for the normalized dividend and divisor. This is done because it is generally undesirable, from the caller's point of view, to alter these input arguments, and because it may be impossible to alter them—they may be constants in read-only memory. Furthermore, the dividend may need an additional high-order digit. C's "alloca" function is ideal for allocating this space. It is usually implemented very efficiently, requiring only two or three in-line instructions to allocate the space and no instructions at all to free it. The space is allocated on the program's stack, in such a way that it is freed automatically upon subroutine return.

In the main loop, the quotient digits are cranked out one per loop iteration, and the dividend is reduced until it becomes the remainder. The estimate `qhat` of each quotient digit, after being refined by the steps in the loop labeled `again`, is always either exact or too high by 1.

The next steps multiply `qhat` by the divisor and subtract the product from the current remainder, as in the grade-school method. If the remainder is negative, it is necessary to decrease the quotient digit by 1 and either re-multiply and subtract or, more simply, adjust the remainder by adding the divisor to it. This need be done at most once, because the quotient digit was either exact or 1 too high.

Lastly, the remainder is given back to the caller if the address of where to put it is non-null. The remainder must be shifted right by the normalization shift amount `s`.

The "add back" steps are executed only rarely. To see this, observe that the first calculation of each estimated quotient digit `qhat` is done by dividing the most significant two digits of the current remainder by the most significant digit of the divisor. The steps in the "again" loop amount to refining `qhat` to be the result of dividing the most significant *three* digits of the current remainder by the most significant *two* digits of the divisor (proof omitted; convince yourself of this by trying some examples using `b` = 10). Note that the divisor is greater than or equal to `b`/2 (because of normalization), and the dividend is less than or equal to `b` times the divisor (because each remainder is less than the divisor).

How accurate is the quotient estimated by using only three dividend digits and two divisor digits? Because normalization was done, it can be shown to be quite accurate. To see this somewhat intuitively (not a formal proof), consider estimating u / v in this way for base ten arithmetic. It can be shown that the estimate is always high (or exact). Thus, the worst case occurs if truncation of the divisor to two digits decreases the divisor by as much as possible in the sense of relative error, and truncation of the dividend to three digits decreases it by as little as possible (which is 0), and if the dividend is as large as possible. This occurs for the case 49900...0/5099...9, which we estimate by 499/50 = 9.98. The true result is approximately 499/51 ≈ 9.7843. The difference of 0.1957 reveals that the estimated quotient digit and the true quotient digit, which are the floor functions of these ratios, will differ by at most 1, and this will occur about 20% of the time (assuming the quotient digits are uniformly distributed). This, in turn, means that the "add back" steps will be executed about 20% of the time.

Carrying out this (non-rigorous) analysis for a general base b yields the result that the estimated and true quotients differ by at most 2 / b. For b = 65536, we again obtain the result that the difference between the estimated and true quotient digits is at most 1, and this occurs with probability 2/65536 ≈ 0.00003. Thus, the "add back" steps are executed for only about 0.003% of the quotient digits.

An example that requires the add back step is, in decimal, 4500/501. A similar example for base 65536 is 0x7FFF8000 00000000/0x8000 00000001.

We will not attempt to estimate the running time of this entire program, but simply note that for large m and n, the execution time is dominated by the multiply/subtract loop. On a good compiler this will compile into about 16 basic RISC instructions, one of which is *multiply*. The "`for j`" loop is executed $m - n + 1$ times, and the multiply/subtract loop n times, giving an execution time for this part of the program of

(15 + *mul*)*n*(*m* – *n* + 1) cycles, where *mul* is the time to multiply two 16-bit variables. The program also executes *m* – *n* + 1 *divide* instructions and one *number of leading zeros* instruction.

Signed Multiword Division

We do not give an algorithm specifically for signed multiword division, but merely point out that the unsigned algorithm can be adapted for this purpose as follows:

1. Negate the dividend if it is negative, and similarly for the divisor.
2. Convert the dividend and divisor to unsigned representation.
3. Use the unsigned multiword division algorithm.
4. Convert the quotient and remainder to signed representation.
5. Negate the quotient if the dividend and divisor had opposite signs.
6. Negate the remainder if the dividend was negative.

These steps sometimes require adding or deleting a most significant digit. For example, assume for simplicity that the numbers are represented in base 256 (one byte per digit), and that in the signed representation, the high-order bit of the sequence of digits is the sign bit. This is much like ordinary two's-complement representation. Then, a divisor of 255, which has signed representation 0x00FF, must be shortened in step 2 to 0xFF. Similarly, if the quotient from step 3 begins with a 1-bit, it must be provided with a leading 0-byte for correct representation as a signed quantity.

9–3 Unsigned Short Division from Signed Division

By "short division" we mean the division of one single word by another (e.g., 32÷32 32). It is the form of division provided by the "/" operator, when the operands are integers, in C and many other high-level languages. C has both signed and unsigned short division, but some computers provide only signed division in their instruction repertoire. How can you implement unsigned division on such a machine? There does not seem to be any really slick way to do it, but we offer some possibilities here.

Using Signed Long Division

Even if the machine has signed long division (64÷32 32), unsigned short division is not as simple as you might think. In the XLC compiler for the IBM RS/6000, it is implemented as illustrated below for $q \leftarrow (n \overset{u}{\div} d)$.

```
if $n \overset{u}{<} d$ then $q \leftarrow 0$
else if $d = 1$ then $q \leftarrow n$
else if $d \leq 1$ then $q \leftarrow 1$
else $q \leftarrow (0 \,\|\, n) \div d$
```

The third line is really testing to see if $d \overset{u}{\geq} 2^{31}$. If *d* is algebraically less than or equal to 1 at this point, then because it is not equal to 1 (from the second line), it must be algebraically less than or equal to 0. We don't care about the case *d* = 0, so for the cases of interest, if the test on the third line evaluates to **true**, the sign bit of *d* is on, that is, $d \overset{u}{\geq} 2^{31}$. Because from the first line it is known that $n \overset{u}{\geq} d$, and because *n* cannot exceed **2** **– 1**, $n \overset{u}{\div} d = 1$.

The notation on the fourth line means to form the double-length integer consisting of 32 0-bits followed by the 32-bit quantity *n*, and divide it by *d*. The test for *d* = **1** (second line) is necessary to ensure that this division does not overflow (it would overflow if $n \overset{u}{\geq} 2^{31}$, and then the quotient would be undefined).

By commoning the comparisons on the second and third lines,[3] the above can be implemented in 11 instructions, three of which are branches. If it is necessary that the *divide* be executed when *d* = **0**, to get the overflow interrupt, then the third line can be changed to “else if *d* < **0** then *q* ← **1**,” giving a 12-instruction solution on the RS/6000.

It is a simple matter to alter the above code so that the probable usual cases $(2 \overset{u}{\leq} d \overset{u}{<} 2^{31})$ do not go through so many tests (begin with if *d* **≤ 1** ...), but the code volume increases slightly.

Using Signed Short Division

This section is written for a 32-bit machine, but it applies to a 64-bit machine (that is, getting unsigned 64÷64 64 division from the same form of signed division) by changing all occurrences of 31 to 63. It can be used to get unsigned division in Java, which lacks unsigned integers.

If signed long division is not available, but signed short division is, then $n \overset{u}{\div} d$ can be implemented by somehow reducing the problem to the case *n, d* < 2^{31} and using the machine’s *divide* instruction. If $d \overset{u}{\geq} 2^{31}$, then $n \overset{u}{\div} d$ can only be **0** or **1**, so this case is easily dispensed with. Then, we can reduce the dividend by using the fact that the expression $((n \overset{u}{\div} 2) \div d) \times 2$ approximates $n \overset{u}{\div} d$ with an error of only 0 or 1. This leads to the following method:

```
1.  if $d < 0$ then if $n \overset{u}{<} d$ then $q \leftarrow 0$
2.      else $q \leftarrow 1$
3.  else do
4.      $q \leftarrow ((n \overset{u}{\div} 2) \div d) \times 2$
5.      $r \leftarrow n - qd$
6.      if $r \overset{u}{\geq} d$ then $q \leftarrow q + 1$
7.  end
```

The test *d* < **0** on line 1 is really testing to determine if $d \overset{u}{\geq} 2^{31}$. If $d \overset{u}{\geq} 2^{31}$, then the largest the quotient could be is $(2^{32} - 1) \div 2^{31} = 1$, so the first two lines compute the correct quotient.

Line 4 represents the code *shift right unsigned 1, divide, shift left 1*. Clearly, $n \overset{u}{\div} 2 \overset{u}{<} 2^{31}$, and at this point $d \overset{u}{<} 2^{31}$ as well, so these quantities can be used in the computer’s signed division instruction. (If *d* = **0**, overflow will be signaled here.)

The estimate computed at line 4 is

$$q = \lfloor \lfloor n/2 \rfloor / d \rfloor \cdot 2 = \lfloor n/(2d) \rfloor \cdot 2 = \frac{n - \text{rem}(n, 2d)}{d},$$

where we have used the corollary of Theorem D3. Line 5 computes the remainder corresponding to the estimated quotient. It is

$$r = n - \frac{n - \text{rem}(n, 2d)}{d} d = \text{rem}(n, 2d).$$

Thus, $0 \le r < 2d$. If $r < d$, then q is the correct quotient. If $r \ge d$, then adding 1 to q gives the correct quotient (the program must use an unsigned comparison here, because of the possibility that $r \ge 2^{31}$).

By moving the *load immediate* of **0** into q ahead of the comparison $n \overset{u}{<} d$, and coding the assignment $q \leftarrow \mathbf{1}$ in line 2 as a branch to the assignment $q \leftarrow q + \mathbf{1}$ in line 6, this can be coded in 14 instructions on most machines, four of which are branches. It is straightforward to augment the code to produce the remainder as well: to line 1 append $r \leftarrow n$, to line 2 append $r \leftarrow n - d$, and to the "then" clause in line 6 append $r \leftarrow r - d$. (Or, at the cost of a *multiply*, simply append $r \leftarrow n - qd$ to the end of the whole sequence.)

An alternative for lines 1 and 2 is

$$\begin{aligned} &\text{if } n \overset{u}{<} d \text{ then } q \leftarrow \mathbf{0} \\ &\text{else if } d < \mathbf{0} \text{ then } q \leftarrow \mathbf{1}, \end{aligned}$$

which can be coded a little more compactly, for a total of 13 instructions, three of which are branches. But it executes more instructions in what is probably the usual case (small numbers with $n > d$).

Using predicate expressions, the program can be written

```
1.  if $d < 0$ then $q \leftarrow (n \overset{u}{\ge} d)$
2.  else do
3.      $q \leftarrow ((n \overset{u}{\div} 2) \div d) \times 2$
4.      $r \leftarrow n - qd$
5.      $q \leftarrow q + (r \overset{u}{\ge} d)$
6.  end
```

which saves two branches if there is a way to evaluate the predicates without branching. On the basic RISC they can be evaluated in one instruction (CMPGEU); on MIPS they take two (SLTU, XORI). On most computers, they can be evaluated in four instructions each (three if equipped with a full set of logic instructions), by using the expression for $x \overset{u}{\le} y$ given in "Comparison Predicates" on page 23, and simplifying because on line 1 of the program above it is known that $d_{31} = 1$, and on line 5 it is known that $d_{31} = 0$. The expression simplifies to

$$n \overset{u}{\geq} d = (n \,\&\, \neg(n - d)) \overset{u}{\gg} 31 \quad \text{on line 1, and}$$

$$r \overset{u}{\geq} d = (r \mid \neg(r - d)) \overset{u}{\gg} 31 \quad \text{on line 5.}$$

We can get branch-free code by forcing the dividend to be **0** when $d \overset{u}{\geq} 2^{31}$. Then, the divisor can be used in the machine's signed *divide* instruction, because when it is misinterpreted as a negative number, the result is set to 0, which is within 1 of being correct. We'll still handle the case of a large dividend by shifting it one position to the right before the division, and then shifting the quotient one position to the left after the division. This gives the following program (ten basic RISC instructions):

```
1.  $t \leftarrow d \overset{s}{\gg} 31$
2.  $n' \leftarrow n \,\&\, \neg t$
3.  $q \leftarrow ((n' \overset{u}{\div} 2) \div d) \times 2$
4.  $r \leftarrow n - qd$
5.  $q \leftarrow q + (r \overset{u}{\geq} d)$
```

9–4 Unsigned Long Division

By "long division" we mean the division of a doubleword by a single word. For a 32-bit machine, this is $64 \overset{u}{\div} 32 \Rightarrow 32$ division, with the result unspecified in the overflow cases, including division by 0.

Some 32-bit machines provide an instruction for unsigned long division. Its full capability, however, gets little use, because only $32 \overset{u}{\div} 32 \Rightarrow 32$ division is accessible with most high-level languages. Therefore, a computer designer might elect to provide only $32 \overset{u}{\div} 32$ division and would probably want an estimate of the execution time of a subroutine that implements the missing function. Here we give two algorithms for providing this missing function.

Hardware Shift-and-Subtract Algorithms

As a first attempt at doing long division, we consider doing what the hardware does. There are two algorithms commonly used, called *restoring* and *nonrestoring* division [H&P, sec. A-2; EL]. They are both basically "shift-and-subtract" algorithms. In the restoring version, shown below, the restoring step consists of adding back the divisor when the subtraction gives a negative result. Here *x, y*, and *z* are held in 32-bit registers. Initially, the double-length dividend is *x* || *y*, and the divisor is *z*. We need a single-bit register *c* to hold the overflow from the subtraction.

```
do $i \leftarrow 1$ to 32
   $c \| x \| y \leftarrow 2(x \| y)$                      // Shift left one.
   $c \| x \leftarrow (c \| x) - (0b0 \| z)$                // Subtract (33 bits).
   $y_0 \leftarrow \neg c$                                  // Set one bit of quotient.
   if $c$ then $c \| x \leftarrow (c \| x) + (0b0 \| z)$    // Restore.
end
```

Upon completion, the quotient is in register *y* and the remainder is in register *x*.

The algorithm does *not* give a useful result in the overflow cases. For division of the doubleword quantity *x* || *y* by 0, the quotient obtained is the one's-complement of *x*, and the remainder obtained is *y*. In particular, $0 \overset{u}{\div} 0 \Rightarrow 2^{32} - 1$ rem 0. The other overflow cases are difficult to characterize.

It might be useful if, for nonzero divisors, the algorithm would give the correct quotient modulo 2^{32}, and the correct remainder. The only way to do this seems to be to make the register represented by *c* || *x* || *y* above 97 bits long, and do the loop 64 times. This is doing $64 \overset{u}{\div} 32 \Rightarrow 64$ division. The subtractions would still be 33-bit operations, but the additional hardware and execution time make this refinement probably not worthwhile.

This algorithm is difficult to implement exactly in software, because most machines do not have the 33-bit register that we have represented by *c* || *x*. Figure 9–2, however, illustrates a shift-and-subtract algorithm that reflects the hardware algorithm to some extent.

The variable `t` is used for a device to make the comparison come out right. We want to do a 33-bit comparison after shifting `x || y`. If the first bit of `x` is 1 (before the shift), then certainly the 33-bit quantity is greater than the divisor (32 bits). In this case, `x | t` is all 1's, so the comparison gives the correct result (**true**). On the other hand, if the first bit of `x` is 0, then a 32-bit comparison is sufficient.

The code of the algorithm in Figure 9–2 executes in 321 to 385 basic RISC instructions, depending upon how often the comparison is **true**. If the machine has *shift left double*, the shifting operation can be done in one instruction, rather than the four used above. This would reduce the execution time to about 225 to 289 instructions (we are allowing two instructions per iteration for loop control).

The algorithm in Figure 9–2 can be used to do $32 \overset{u}{\div} 32 \Rightarrow 32$ division by supplying `x = 0`. The only simplification that results is that the variable `t` can be omitted, as its value would always be 0.

```
unsigned divlu(unsigned x, unsigned y, unsigned z) {
   // Divides (x || y) by z.
   int i;
   unsigned t;

   for (i = 1; i <= 32; i++) {
      t = (int)x >> 31;           // All 1's if x(31) = 1.
      x = (x << 1) | (y >> 31);   // Shift x || y left
      y = y << 1;                 // one bit.
      if ((x | t) >= z) {
            x = x - z;
            y = y + 1;
         }
      }
      return y;                        // Remainder is x.
   }
```

FIGURE 9–2. *Divide long unsigned*, **shift-and-subtract algorithm.**

On the next page is the nonrestoring hardware division algorithm (unsigned). The basic idea is that, after subtracting the divisor *z* from the 33-bit quantity that we denote by *c* || *x*, there is no need to add back *z* if the result was negative. Instead, it suffices to *add* on the next iteration rather than *subtract*. This is because adding *z* (to correct the error of having subtracted *z* on the previous iteration), shifting left, and subtracting *z* is equivalent to adding $z(2(u + z) - z = 2u + z)$. The advantage to hardware is that there is only one add or subtract operation on each loop iteration, and the adder is likely to be the slowest circuit in the loop.[4] An adjustment to the remainder is needed at the end if it is negative. (No corresponding adjustment of the quotient is required.)

The input dividend is the doubleword quantity *x* || *y*, and the divisor is *z*. Upon completion, the quotient is in register *y* and the remainder is in register *x*.

```
$c = 0$
do $i \leftarrow 1$ to 32
   if $c = 0$ then do
      $c \| x \| y \leftarrow 2(x \| y)$              // Shift left one.
      $c \| x \leftarrow (c \| x) - (0b0 \| z)$       // Subtract divisor.
   end
   else do
      $c \| x \| y \leftarrow 2(x \| y)$              // Shift left one.
      $c \| x \leftarrow (c \| x) + (0b0 \| z)$       // Add divisor.
   end
   $y_0 \leftarrow \neg c$                            // Set one bit of quotient.
end
if $c = 1$ then $x \leftarrow x + z$                  // Adjust remainder if negative.
```

This does not seem to adapt very well to a 32-bit algorithm.

The 801 minicomputer (an early experimental RISC machine built by IBM) had a *divide step* instruction that essentially performed the steps in the body of the loop above. It used the machine's carry status bit to hold *c* and the MQ (a 32-bit register) to hold *y*. A 33-bit adder/subtracter is needed for its implementation. The 801's *divide step* instruction was a little more complicated than the loop above, because it performed signed division and it had an overflow check. Using it, a division subroutine can be written that consists essentially of 32 consecutive *divide step* instructions followed by some adjustments to the quotient and remainder to make the remainder have the desired sign.

Using Short Division

An algorithm for $64 \overset{u}{\div} 32 \Rightarrow 32$ division can be obtained from the multiword division algorithm of Figure 9–1 on page 185, by specializing it to the case *m* = 4, *n* = 2. Several other changes are necessary. The parameters should be fullwords passed by value, rather than arrays of halfwords. The overflow condition is different; it occurs if the quotient cannot be contained in a single fullword. It turns out that many simplifications to the routine are possible. It can be shown that the guess `qhat` is always exact; it is exact if the divisor consists of only two halfword digits. This means that the “add back” steps can be omitted. If the “main loop” of Figure 9–1 and the loop within it are unrolled, some minor simplifications become possible.

The result of these transformations is shown in Figure 9–3. The dividend is in `u1` and `u0`, with `u1` containing the most significant word. The divisor is parameter `v`. The quotient is the returned value of the function. If the caller provides a non-null pointer in parameter `r`, the function will return the remainder in the word to which `r` points.

For an overflow indication, the program returns a remainder equal to the maximum unsigned integer. This is an impossible remainder for a valid division operation, because the remainder must be less than the divisor. In the overflow case, the program also returns a quotient equal to the maximum unsigned integer, which may be an adequate indicator in some cases in which the remainder is not wanted.

The strange expression `(-s >> 31)` in the assignment to `un32` is supplied to make the program work for the case `s = 0` on machines that have mod 32 shifts (e.g., Intel x86).

Experimentation with uniformly distributed random numbers suggests that the bodies of the “again” loops are each executed about 0.38 times for each execution of the function. This gives an execution time, if the remainder is not wanted, of about 52 instructions. Of these instructions, one is *number of leading zeros*, two are *divide*, and 6.5 are *multiply* (not counting the multiplications by `b`, which are *shift*’s). If the remainder is wanted, add six instructions (counting the store of `r`), one of which is *multiply*.

What about a signed version of `divlu`? It would probably be difficult to modify the code of Figure 9–3, step by step, to produce a signed variant. That algorithm, however, can be used for signed division by taking the absolute value of the arguments, running `divlu`, and then complementing the result if the signs of the original arguments differ. There is no problem with extreme values such as the maximum negative number, because the absolute value of any signed integer has a correct representation as an unsigned integer. This algorithm is shown in Figure 9–4.

It is hard to devise really good code to detect overflow in the signed case. The algorithm shown in Figure 9–4 makes a preliminary determination identical to that used by the unsigned long division routine, which ensures that $|u / v| < 2^{32}$. After that, it is necessary only to ensure that the quotient has the proper sign or is 0.

```
unsigned divlu(unsigned u1, unsigned u0, unsigned v,
                    unsigned *r) {
   const unsigned b = 65536;        // Number base (16 bits).
   unsigned un1, un0,                 // Norm. dividend LSD's.
            vn1, vn0,                 // Norm. divisor digits.
            q1, q0,                   // Quotient digits.
            un32, un21, un10,         // Dividend digit pairs.
            rhat;                     // A remainder.
   int s;                             // Shift amount for norm.

   if (u1 >= v) {                     // If overflow, set rem.
      if (r != NULL)                  // to an impossible value,
         *r = 0xFFFFFFFF;             // and return the largest
      return 0xFFFFFFFF;}             // possible quotient.

   s = nlz(v);                        // 0 <= s <= 31.
   v = v << s;                        // Normalize divisor.
   vn1 = v >> 16;                     // Break divisor up into
   vn0 = v & 0xFFFF;                  // two 16-bit digits.

   un32 = (u1 << s) | (u0 >> 32 - s) & (-s >> 31);
   un10 = u0 << s;                    // Shift dividend left.

   un1 = un10 >> 16;                  // Break right half of
   un0 = un10 & 0xFFFF;               // dividend into two digits.

   q1 = un32/vn1;                     // Compute the first
   rhat = un32 - q1*vn1;              // quotient digit, q1.
 again1:
   if (q1 >= b || q1*vn0 > b*rhat + un1) {
      q1 = q1 - 1;
      rhat = rhat + vn1;
      if (rhat < b) goto again1;}

   un21 = un32*b + un1 - q1*v;           // Multiply and subtract.

   q0 = un21/vn1;                      // Compute the second
   rhat = un21 - q0*vn1;               // quotient digit, q0.
 again2:
   if (q0 >= b || q0*vn0 > b*rhat + un0) {
     q0 = q0 - 1;
     rhat = rhat + vn1;
     if (rhat < b) goto again2;}

   if (r != NULL)                      // If remainder is wanted,
      *r = (un21*b + un0 - q0*v) >> s;      // return it.
   return q1*b + q0;
 }
```

FIGURE 9–3. *Divide long unsigned*, **using fullword division instruction.**

```
int divls(int u1, unsigned u0, int v, int *r) {
  int q, uneg, vneg, diff, borrow;

  uneg = u1 >> 31;          // -1 if u < 0.
  if (uneg) {               // Compute the absolute
     u0 = -u0;              // value of the dividend u.
     borrow = (u0 != 0);
     u1 = -u1 - borrow;}

  vneg = v >> 31;           // -1 if v < 0.
  v = (v ^ vneg) - vneg;    // Absolute value of v.

  if ((unsigned)u1 >= (unsigned)v) goto overflow;
  q = divlu(u1, u0, v, (unsigned *)r);

  diff = uneg ^ vneg;      // Negate q if signs of
  q = (q ^ diff) - diff;   // u and v differed.
  if (uneg && r != NULL)
     *r = -*r;

  if ((diff ^ q) < 0 && q != 0) {  // If overflow,
overflow:                  // set remainder
     if (r != NULL)        // to an impossible value,
       *r = 0x80000000;    // and return the largest
     q = 0x80000000;}      // possible neg. quotient.
  return q;
}
```

FIGURE 9–4. *Divide long signed*, **using** *divide long unsigned*.

9–5 Doubleword Division from Long Division

This section considers how to do 64 ÷ 64   64 division from 64 ÷ 32   32 division, for both the unsigned and signed cases. The algorithms that follow are most suited to a machine that has an instruction for long division (64 ÷ 32), at least for the unsigned case. It is also helpful if the machine has the *number of leading zeros* instruction. The machine may have either 32-bit or 64-bit registers, but we will assume that if it has 32-bit registers, then the compiler implements basic operations such as adds and shifts on 64-bit operands (the “long long” data type in C).

These functions are known as “_ _udivdi3” and “_ _divdi3” in the GNU C world, and similar names are used here.

Unsigned Doubleword Division

A procedure for this operation is shown in Figure 9–5.

```
unsigned long long udivdi3(unsigned long long u,
                           unsigned long long v) {

   unsigned long long u0, u1, v1, q0, q1, k, n;

   if (v >> 32 == 0) {          // If v < 2**32:
      if (u >> 32 < v)          // If u/v cannot overflow,
         return DIVU(u, v)      // just do one division.
            & 0xFFFFFFFF;
      else {                    // If u/v would overflow:
         u1 = u >> 32;          // Break u up into two
         u0 = u & 0xFFFFFFFF;   // halves.
         q1 = DIVU(u1, v)       // First quotient digit.
            & 0xFFFFFFFF;
         k = u1 - q1*v;         // First remainder, < v.
         q0 = DIVU((k << 32) + u0, v)  // 2nd quot. digit.
            & 0xFFFFFFFF;
         return (q1 << 32) + q0;
      }
   }
                                // Here v >= 2**32.
   n = nlz64(v);                // 0 <= n <= 31.
   v1 = (v << n) >> 32;         // Normalize the divisor
                                // so its MSB is 1.
   u1 = u >> 1;                 // To ensure no overflow.
   q1 = DIVU(u1, v1)            // Get quotient from
```

```
      & 0xFFFFFFFF;            // divide unsigned insn.
   q0 = (q1 << n) >> 31;       // Undo normalization and
                               // division of u by 2.
   if (q0 != 0)                // Make q0 correct or
      q0 = q0 - 1;             // too small by 1.
   if ((u - q0*v) >= v)
      q0 = q0 + 1;             // Now q0 is correct.
   return q0;
}
```

FIGURE 9–5. Unsigned doubleword division from long division.

This code distinguishes three cases: (1) the case in which a single execution of the machine's unsigned long division instruction (DIVU) can be used, (2) the case in which (1) does not apply, but the divisor is a 32-bit quantity, and (3) the cases in which the divisor cannot be represented in 32 bits. It is not too hard to see that the above code is correct for cases (1) and (2). For case (2), think of the grade-school method of doing long division.

Case (3), though, deserves proof, because it is very close to not working in some cases. Notice that in this case only a single execution of DIVU is needed, but the *number of leading zeros* and *multiply* operations are needed.

For the proof, we need these basics (for integer variables):

$$\lfloor \lfloor a/b \rfloor / d \rfloor = \lfloor a/(bd) \rfloor \qquad (2)$$

$$b \lfloor a/b \rfloor = a - \text{rem}(a, b) \qquad (3)$$

From the first line in the section of the procedure of interest (we assume that $v \neq 0$),

$$0 \leq n \leq 31.$$

In computing v_1, the left shift clearly cannot overflow. Therefore,

$$\begin{aligned} v_1 &= \lfloor v/2^{32-n} \rfloor, \quad \text{and} \\ u_1 &= \lfloor u/2 \rfloor. \end{aligned}$$

In computing q_1, u_1 and v_1 are in range for the DIVU instruction and it cannot overflow. Hence,

$$q_1 = \quad u_1 / v_1 \quad .$$

In the first computation of q_0, the left shift cannot overflow because $q_1 < 2^{32}$ (because the maximum value of u_1 is $2^{63} - 1$ and the minimum value of v_1 is 2^{31}). Therefore,

$$q_0 = \quad q_1/2^{31-n} \quad .$$

Now, for the main part of the proof, we want to show that

$$u / v \quad \leq q \leq \quad u / v \quad + 1,$$

0

which is to say, the first computation of q_0 is the desired result or is that plus 1.

Using Equation (2) twice gives

$$\begin{aligned} q_0 &= \left\lfloor \frac{u}{2^{32-n}v_1} \right\rfloor \\ &= \left\lfloor \frac{u}{2^{32-n}\left\lfloor \frac{v}{2^{32-n}} \right\rfloor} \right\rfloor. \end{aligned}$$

Using Equation (3) gives

$$q_0 = \left\lfloor \frac{u}{v - \text{rem}(v, 2^{32-n})} \right\rfloor.$$

Using algebra to get this in the form u / v + something:

$$q_0 = \left\lfloor \frac{u}{v} + \frac{u\,\text{rem}(v, 2^{32-n})}{v(v - \text{rem}(v, 2^{32-n}))} \right\rfloor.$$

This is of the form

$$\left\lfloor \frac{u}{v} + \delta \right\rfloor,$$

and we will now show that $\delta < 1$.

δ is largest when rem(v, 2^{32-n}) is as large as possible and, given that, when v is as small as possible. The maximum value of rem(v, 2^{32-n}) is $2^{32-n} - 1$. Because of the way n is defined in terms of v, $v \geq 2^{63-n}$. Thus, the smallest value of v having that remainder is

$$2^{63-n} + 2^{32-n} - 1.$$

Therefore,

$$\begin{aligned} \delta &\leq \frac{u(2^{32-n} - 1)}{(2^{63-n} + 2^{32-n} - 1)2^{63-n}} \\ &< \frac{u(2^{32-n} - 1)}{(2^{63-n})^2}. \end{aligned}$$

By inspection, for n in its range of 0 to 31,

$$\delta < \frac{u}{2^{64}}.$$

Since u is at most $2^{64} - 1$, $\delta < 1$. Because $q_0 = \lfloor u/v + \delta \rfloor$ and $\delta < 1$ (and obviously $\delta \geq 0$),

$$\left\lfloor \frac{u}{v} \right\rfloor \leq q_0 \leq \left\lfloor \frac{u}{v} \right\rfloor + 1.$$

To correct this result by subtracting 1 when necessary, we would like to code

```
if (u < q0*v) q0 = q0 - 1;
```

(i.e., if the remainder $u - q_0 v$ is negative, subtract 1 from q_0). However, this doesn't quite work, because $q_0\ v$ can overflow (e.g., for $u = 2^{64} - 1$ and $v = 2^{32} + 3$). Instead, we subtract 1 from q_0, so that it is either correct or too *small* by 1. Then $q_0\ v$ will not overflow. We must avoid subtracting 1 if $q_0 = 0$ (if $q_0 = 0$, it is already the correct quotient).

Then the final correction is:

```
if ((u - q0*v) >= v) q0 = q0 - 1;
```

To see that this is a valid computation, we already noted that $q_0 v$ does not overflow. It is easy to show that

$$0 \leq u - q_0 v < 2v.$$

If v is very large ($\geq 2^{63}$), can the subtraction overflow by trying to produce a result greater than v? No, because $u < 2^{64}$ and $q_0 v \geq 0$.

Incidentally, there are alternatives to the lines

```
if (q0 != 0)      // Make q0 correct or
   q0 = q0 - 1    // too small by 1.
```

that may be preferable on some machines. One is to replace them with

```
if (q0 == 0) return 0;
```

Another is to place at the beginning of this section of the procedure, or at the beginning of the whole procedure, the line

```
if (u < v) return 0; // Avoid a problem later.
```

These alternatives are preferable if branches are not costly. The code shown in Figure 9–5 works well if the machine's comparison instructions produce a 0/1 integer result in a general register. Then, the compiler can change it to, in effect,

```
q0 = q0 - (q0 != 0);
```

(or you can code it that way if your compiler doesn't do this optimization). This is just a *compare* and *subtract* on such machines.

Signed Doubleword Division

In the signed case, there seems to be no better way to do doubleword division than to divide the absolute values of the operands, using function `udivdi3`, and then negate the sign of the quotient if the operands have different signs. If the machine has a signed long division instruction, which we designate here as DIVS, then it may be advantageous to single out the cases in which DIVS can be used rather than invoking `udivdi3`. This presumes that these cases are common. Such a function is shown in Figure 9–6.

The "`#define`" in the code in Figure 9–6 uses the GCC facility of enclosing a compound statement in parentheses to construct an expression, a facility that most C compilers do not have. Some other compilers may have `llabs(x)` as a built-in function.

```
#define llabs(x) \
({unsigned long long t = (x) >> 63; ((x) ^ t) - t;})

long long divdi3(long long u, long long v) {

   unsigned long long au, av;
   long long q, t;

   au = llabs(u);
   av = llabs(v);
   if (av >> 31 == 0) {          // If |v| < 2**31 and
      if (au < av << 31) {       // |u|/|v| cannot
         q = DIVS(u, v);         // overflow, use DIVS.
         return (q << 32) >> 32;
      }
   }
   q = au/av;                    // Invoke udivdi3.
   t = (u ^ v) >> 63;            // If u, v have different
   return (q ^ t) - t;           // signs, negate q.
}
```

FIGURE 9–6. Signed doubleword division from unsigned doubleword division.

The test that `v` is in range is not precise; it misses the case in which $v = -2^{31}$. If it is important to use the DIVS instruction in that case, the test

```
   if ((v << 32) >> 32 == v) { // If v is in range and
```

can be used in place of the third executable line in Figure 9–6 (at a cost of one instruction). Similarly, the test that $|u|\ /\ |v|$ cannot overflow is simplified and a few "corner cases" will be missed; the code amounts to using $\delta = 0$ in the signed division overflow test scheme shown in "Division" on page 34.

Exercises

1. Show that for real x, $x = - \; -x$.

2. Find branch-free code for computing the quotient and remainder of modulus division on a basic RISC that has division and remainder instructions for truncating division.

3. Similarly, find branch-free code for computing the quotient and remainder of floor division on a basic RISC that has division and remainder instructions for truncating division.

4. How would you compute n / d for unsigned integers n and d, $0 \le n \le 2^{32} - 1$ and $1 \le d \le 2^{32} - 1$? Assume your machine has an unsigned *divide* instruction that computes n / d .

5. Theorem D3 states that for x real and d an integer, $x / d = x / d$. Show that, more generally, if a function $f(x)$ is (a) continuous, (b) monotonically increasing, and (c) has the property that if $f(x)$ is an integer then x is an integer, then $f(x) = f(x)$ [GKP].

Chapter 10. Integer Division By Constants

On many computers, division is very time consuming and is to be avoided when possible. A value of 20 or more elementary *add* times is not uncommon, and the execution time is usually the same large value even when the operands are small. This chapter gives some methods for avoiding the *divide* instruction when the divisor is a constant.

10–1 Signed Division by a Known Power of 2

Apparently, many people have made the mistake of assuming that a *shift right signed* of k positions divides a number by 2^k, using the usual truncating form of division [GLS2]. It's a little more complicated than that. The code shown below computes $q = n \div \mathbf{2}^k$, for $1 \le k \le 31$ [Hop].

```
shrsi t,n,k-1        Form the integer
shri  t,t,32-k       2**k – 1 if n < 0, else 0.
add   t,n,t          Add it to n,
shrsi q,t,k          and shift right (signed).
```

It is branch free. It simplifies to three instructions in the common case of division by 2 ($k = 1$). It does, however, rely on the machine's being able to shift by a large amount in a short time. The case $k = 31$ does not make too much sense, because the number 2^{31} is not representable in the machine. Nevertheless, the code does produce the correct result in that case (which is $q = -1$ if $n = -2^{31}$and $q = 0$ for all other n).

To divide by -2^k, the above code can be followed by a *negate* instruction. There does not seem to be any better way to do it.

The more straightforward code for dividing by 2^k is

```
      bge   n,label      Branch if n >= 0.
      addi  n,n,2**k-1   Add 2**k - 1 to n,
label shrsi n,n,k        and shift right (signed).
```

This would be preferable on a machine with slow shifts and fast branches.

PowerPC has an unusual device for speeding up division by a power of 2 [GGS]. The *shift right signed* instructions set the machine's carry bit if the number being shifted is negative and one or more 1-bits are shifted out. That machine also has an instruction for adding the carry bit to a register, denoted `addze`. This allows division by any (positive) power of 2 to be done in two instructions:

```
shrsi q,n,k
addze q,q
```

A single `shrsi` of k positions does a kind of signed division by 2^k that coincides with both modulus and floor division. This suggests that one of these might be preferable to truncating division for computers and HLL's to use. That is, modulus and floor division mesh with `shrsi` better than does truncating division, permitting a compiler to translate the expression n / 2 to an `shrsi`. Furthermore, `shrsi` followed by `neg` (negate) does modulus division by -2^k, which is a hint that maybe modulus division is best. (This is mainly an aesthetic issue. It is of little practical significance, because division by a negative constant is no doubt extremely rare.)

10–2 Signed Remainder from Division by a Known Power of 2

If both the quotient and remainder of $n \div \mathbf{2}^k$ are wanted, it is simplest to compute the remainder r from $r = n - q * \mathbf{2}^k$ This requires only two instructions after computing the quotient q:

```
shli  r,q,k
sub   r,n,r
```

To compute only the remainder seems to require about four or five instructions. One way to compute it is to use the four-instruction sequence above for signed division by 2^k, followed by the two instructions shown immediately above to obtain the remainder. This results in two consecutive *shift* instructions that can be replaced by an *and*, giving a solution in five instructions (four if $k = 1$):

```
shrsi t,n,k-1        Form the integer
shri t,t,32-k        2**k - 1 if n < 0, else 0.
add t,n,t            Add it to n,
andi t,t,-2**k       clear rightmost k bits,
sub r,n,t            and subtract it from n.
```

Another method is based on

$$\operatorname{rem}(n, 2^k) = \begin{cases} n \,\&\, (2^k - 1), & n \ge 0, \\ -((-n) \,\&\, (2^k - 1)), & n < 0. \end{cases}$$

To use this, first compute $t \leftarrow n \overset{s}{\gg} 31$, and then

$$r \leftarrow ((\operatorname{abs}(n) \,\&\, (\mathbf{2}^k - \mathbf{1})) \quad t) - t$$

(five instructions) or, for $k = 1$, since $(-n) \,\&\, \mathbf{1} = n \,\&\, \mathbf{1}$,

$$r \leftarrow ((n \,\&\, \mathbf{1}) \quad t) - t$$

(four instructions). This method is not very good for $k > 1$ if the machine does not have *absolute value* (computing the remainder would then require six instructions).

Still another method is based on

$$\operatorname{rem}(n, 2^k) = \begin{cases} n \,\&\, (2^k - 1), & n \ge 0, \\ ((n + 2^k - 1) \,\&\, (2^k - 1)) - (2^k - 1), & n < 0. \end{cases}$$

This leads to

$$t \leftarrow (n \overset{s}{\gg} k - 1) \overset{u}{\gg} 32 - k$$

$$r \leftarrow ((n + t) \,\&\, (2^k - 1)) - t$$

(five instructions for $k > 1$, four for $k = 1$).

The above methods all work for $1 \le k \le 31$.

Incidentally, if *shift right signed* is not available, the value that is $2^k - 1$ for $n < 0$ and 0 for $n \ge 0$ can be constructed from

$$t_1 \leftarrow n \overset{u}{\gg} 31$$

$$r \leftarrow (t_1 \ll k) - t_1,$$

which adds only one instruction.

10–3 Signed Division and Remainder by Non-Powers of 2

The basic trick is to multiply by a sort of reciprocal of the divisor d, approximately $2^{32}/d$, and then to extract the leftmost 32 bits of the product. The details, however, are more complicated, particularly for certain divisors such as 7.

Let us first consider a few specific examples. These illustrate the code that will be generated by the general method. We denote registers as follows:

`n` – the input integer (numerator)
`M` – loaded with a “magic number”
`t` - a temporary register
`q` - will contain the quotient
`r` - will contain the remainder

Division by 3

```
li      M,0x55555556    Load magic number, (2**32+2)/3.
mulhs   q,M,n           q = floor(M*n/2**32).
shri    t,n,31          Add 1 to q if
add     q,q,t           n is negative.

muli    t,q,3           Compute remainder from
sub     r,n,t           r = n - q*3.
```

Proof. The *multiply high signed* operation (`mulhs`) cannot overflow, as the product of two 32-bit integers can always be represented in 64 bits and `mulhs` gives the high-order 32 bits of the 64-bit product. This is equivalent to dividing the 64-bit product by 2^{32} and taking the floor of the result, and this is true whether the product is positive or negative. Thus, for $n \ge 0$ the above code computes

$$q = \left\lfloor \frac{2^{32}+2}{3} \frac{n}{2^{32}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{2n}{3 \cdot 2^{32}} \right\rfloor.$$

Now, $n < 2^{31}$, because $2^{31} - 1$ is the largest representable positive number. Hence, the “error” term $2n / (3 \cdot 2^{32})$ is less than 1/3 (and is nonnegative), so by Theorem D4 (page 183) we have $q = \quad n / 3 \quad$, which is the desired result (Equation (1) on page

182).

For $n < 0$, there is an addition of 1 to the quotient. Hence the code computes

$$q = \left\lfloor \frac{2^{32}+2}{3}\,\frac{n}{2^{32}} \right\rfloor + 1 = \left\lfloor \frac{2^{32}n + 2n + 3\cdot 2^{32}}{3\cdot 2^{32}} \right\rfloor = \left\lceil \frac{2^{32}n + 2n + 1}{3\cdot 2^{32}} \right\rceil,$$

where we have used Theorem D2. Hence

$$q = \left\lceil \frac{n}{3} + \frac{2n+1}{3\cdot 2^{32}} \right\rceil.$$

For $-2^{31} \le n \le -1$,

$$-\frac{1}{3} + \frac{1}{3\cdot 2^{32}} \le \frac{2n+1}{3\cdot 2^{32}} \le -\frac{1}{3\cdot 2^{32}}.$$

The error term is nonpositive and greater than –1 / 3, so by Theorem D4 $q = n / 3$, which is the desired result (Equation (1) on page 182).

This establishes that the quotient is correct. That the remainder is correct follows easily from the fact that the remainder must satisfy

$$n = qd + r,$$

the multiplication by 3 cannot overflow (because $-2^{31} / 3 \le q \le (2^{31} - 1) / 3$), and the *subtract* cannot overflow because the result must be in the range –2 to +2.

The *multiply immediate* can be done with two *add*'s, or a *shift* and an *add*, if either gives an improvement in execution time.

On many present-day RISC computers, the quotient can be computed as shown above in nine or ten cycles, whereas the *divide* instruction might take 20 cycles or so.

Division by 5

For division by 5, we would like to use the same code as for division by 3, except with a multiplier of $(2^{32} + 4) / 5$. Unfortunately, the error term is then too large; the result is off by 1 for about 1/5 of the values of $n \ge 2^{30}$ in magnitude. However, we can use a multiplier of $(2^{33} + 3) / 5$ and add a *shift right signed* instruction. The code is

```
li    M,0x66666667  Load magic number, (2**33+3)/5.
mulhs q,M,n         q = floor(M*n/2**32).
shrsi q,q,1
shri  t,n,31        Add 1 to q if
add   q,q,t         n is negative.

muli  t,q,5         Compute remainder from
sub   r,n,t         r = n - q*5.
```

Proof. The `mulhs` produces the leftmost 32 bits of the 64-bit product, and then the code shifts this right by one position, signed (or “arithmetically”). This is equivalent to dividing the product by 2^{33} and then taking the floor of the result. Thus, for $n \ge 0$ the code computes

$$q = \left\lfloor \frac{2^{33}+3}{5}\frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{5} + \frac{3n}{5 \cdot 2^{33}} \right\rfloor.$$

For $0 \le n < 2^{31}$, the error term $3n / 5 \cdot 2^{33}$ is nonnegative and less than 1/5, so by Theorem D4, $q =$ $n / 5$.

For $n < 0$, the above code computes

$$q = \left\lfloor \frac{2^{33}+3}{5}\frac{n}{2^{33}} \right\rfloor + 1 = \left\lceil \frac{n}{5} + \frac{3n+1}{5 \cdot 2^{33}} \right\rceil.$$

The error term is nonpositive and greater than –1/5, so $q =$ $n / 5$.
That the remainder is correct follows as in the case of division by 3.
The *multiply immediate* can be done with a *shift left* of two and an *add*.

Division by 7

Dividing by 7 creates a new problem. Multipliers of $(2^{32} + 3) / 7$ and $(2^{33} + 6) / 7$ give error terms that are too large. A multiplier of $(2^{34} + 5) / 7$ would work, but it's too large to represent in a 32-bit signed word. We can multiply by this large number by multiplying by $(2^{34} + 5) / 7 - 2^{32}$ (a negative number), and then correcting the product by inserting an `add`. The code is

```
li     M,0x92492493  Magic num, (2**34+5)/7 - 2**32.
mulhs  q,M,n          q = floor(M*n/2**32).
add    q,q,n          q = floor(M*n/2**32) + n.
shrsi  q,q,2          q = floor(q/4).
shri   t,n,31         Add 1 to q if
add    q,q,t          n is negative.

muli   t,q,7          Compute remainder from
sub    r,n,t          r = n - q*7.
```

Proof. It is important to note that the instruction "`add q,q,n`" above cannot overflow. This is because q and n have opposite signs, due to the multiplication by a negative number. Therefore, this "computer arithmetic" addition is the same as real number addition. Hence for $n \ge 0$ the above code computes

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) \Big/ 4 \right\rfloor = \left\lfloor \left\lfloor \frac{2^{34}n + 5n - 7 \cdot 2^{32}n + 7 \cdot 2^{32}n}{7 \cdot 2^{32}} \right\rfloor \Big/ 4 \right\rfloor$$
$$= \left\lfloor \frac{n}{7} + \frac{5n}{7 \cdot 2^{34}} \right\rfloor,$$

where we have used the corollary of Theorem D3.

For $0 \le n \le 2^{31}$, the error term $5n/ 7 \cdot 2^{34}$ is nonnegative and less than 1/7, so q = n / 7 .

For $n < 0$, the above code computes

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) / 4 \right\rfloor + 1 = \left\lceil \frac{n}{7} + \frac{5n+1}{7 \cdot 2^{34}} \right\rceil.$$

The error term is nonpositive and greater than –1/7, so $q =$ $n / 7$

The *multiply immediate* can be done with a *shift left* of three and a *subtract*.

10–4 Signed Division by Divisors ≥ 2

At this point you may wonder if other divisors present other problems. We see in this section that they do not; the three examples given illustrate the only cases that arise (for $d \geq 2$).

Some of the proofs are a bit complicated, so to be cautious, the work is done in terms of a general word size W.

Given a word size $W \geq 3$ and a divisor d, $2 \leq d \leq 2^{W-1}$ we wish to find the least integer m and integer p such that

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \qquad \text{for } 0 \leq n < 2^{W-1}, \text{ and} \qquad (1a)$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \qquad \text{for } -2^{W-1} \leq n \leq -1, \qquad (1b)$$

with $0 \leq m < 2^W$ and $p \geq W$.

The reason we want the *least* integer m is that a smaller multiplier may give a smaller shift amount (possibly zero) or may yield code similar to the “divide by 5” example, rather than the “divide by 7” example. We must have $m \leq 2^W - 1$ so the code has no more instructions than that of the “divide by 7” example (that is, we can handle a multiplier in the range 2^{W-1} to $2^W - 1$ by means of the add that was inserted in the “divide by 7” example, but we would rather not deal with larger multipliers). We must have $p \geq W$, because the generated code extracts the left half of the product mn, which is equivalent to shifting right W positions. Thus, the total right shift is W or more positions.

There is a distinction between the multiplier m and the “magic number,” denoted M. The magic number is the value used in the *multiply* instruction. It is given by

$$M = \begin{cases} m, & \text{if } 0 \leq m < 2^{W-1}, \\ m - 2^W, & \text{if } 2^{W-1} \leq m < 2^W. \end{cases}$$

Because (1b) must hold for $n = -d$, $-md/2^p$ $+ 1 = -1$, which implies

$$\frac{md}{2^p} > 1. \qquad (2)$$

Let n_c be the largest (positive) value of n such that $\text{rem}(n_c, d) = d - 1$. n_c exists

$W - 1$

because one possibility is $n_c = d - 1$. It can be calculated from n_c = 2 / d d – 1 = 2^{W-1} – rem(2^{W-1}, d) – 1. n_c is one of the highest d admissible values of n, so

$$2^{W-1} - d \le n_c \le 2^{W-1} - 1, \tag{3a}$$

and, clearly

$$n_c \ge d - 1. \tag{3b}$$

Because (1a) must hold for $n = n_c$

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d},$$

or

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}.$$

Combining this with (2) gives

$$\frac{2^p}{d} < m < \frac{2^p}{d}\frac{n_c + 1}{n_c}. \tag{4}$$

Because m is to be the least integer satisfying (4), it is the next integer greater than $2^p / d$; that is,

$$\boxed{m = \frac{2^p + d - \text{rem}(2^p, d)}{d}.} \tag{5}$$

Combining this with the right half of (4) and simplifying gives

$$\boxed{2^p > n_c(d - \text{rem}(2^p, d)).} \tag{6}$$

The Algorithm

Thus, the algorithm to find the magic number M and the shift amount s from d is to first compute n_c, and then solve (6) for p by trying successively larger values. If $p < W$, set $p = W$ (the theorem below shows that this value of p also satisfies (6)). When the smallest $p \ge W$ satisfying (6) is found, m is calculated from (5). This is the smallest possible value of m, because we found the smallest acceptable p, and from (4) clearly smaller values of p yield smaller values of m. Finally, $s = p - W$ and M is simply a reinterpretation of m as a signed integer (which is how the `mulhs` instruction interprets it).

Forcing p to be at least W is justified by the following:

THEOREM DC1. *If*(6) *is true for some value of p, then it is true for all larger values of p*.

Proof. Suppose (6) is true for $p = p_0$. Multiplying (6) by 2 gives

$$2^{p_0+1} > n_c(2d - 2\text{rem}(2^{p_0}, d)).$$

From Theorem D5, $\text{rem}(2^{p_0+1}, d) \geq 2\text{rem}(2^{p_0}, d) - d$. Combining gives

$$2^{p_00+1} > n_c((2d - (\text{rem}(2^{p_0+1}, d) + d)), \text{ or}$$
$$2^{p\,0+1} > n_c(d - \text{rem}(2^{p_0+1}, d)).$$

Therefore, (6) is true for $p = p_0 + 1$, and hence for all larger values.

Thus, one could solve (6) by a binary search, although a simple linear search (starting with $p = W$) is probably preferable, because usually d is small, and small values of d give small values of p.

Proof That the Algorithm Is Feasible

We must show that (6) always has a solution and that $0 \leq m < 2^W$. (It is not necessary to show that $p \geq W$, because that is forced.)

We show that (6) always has a solution by getting an upper bound on p. As a matter of general interest, we also derive a lower bound under the assumption that p is not forced to be at least W. To get these bounds on p, observe that for any positive integer x, there is a power of 2 greater than x and less than or equal to $2x$. Hence, from (6),

$$n_c(d - \text{rem}(2^p, d)) < 2^p \leq 2n_c((d - \text{rem}(2^p, d)).$$

Because $0 \leq \text{rem}(2^p, d) \leq d - 1$,

$$n_c + 1 \leq 2^p \leq 2n_c d. \qquad (7)$$

From (3a) and (3b), $n_c \geq \max(2^{W-1} - d, d - 1)$. The lines $f_1(d) = 2^{W-1} - d$ and $f_2(d) = d - 1$ cross at $d = (2^{W-1} + 1) / 2$. Hence $n_c \geq (2^{W-1} - 1) / 2$. Because n_c is an integer, $n_c \geq 2^{W-2}$. Because n_c, $d \leq 2^{W-1} - 1$, (7) becomes

$$2^{W-2} + 1 \leq 2^p \leq 2(2^{W-1} - 1)^2$$

or

$$W - 1 \leq p \leq 2W - 2. \qquad (8)$$

The lower bound $p = W - 1$ can occur (e.g., for $W = 32$, $d = 3$), but in that case we set $p = W$.

If p is not forced to equal W, then from (4) and (7),

$$\frac{n_c + 1}{d} < m < \frac{2n_c d}{d}\frac{n_c + 1}{n_c}.$$

Using (3b) gives

$$\frac{d-1+1}{d} < m < 2(n_c + 1).$$

Because $n_c \le 2^{W-1} - 1$ (3a),

$$2 \le m \le 2^W - 1.$$

If p is forced to equal W, then from (4),

$$\frac{2^W}{d} < m < \frac{2^W}{d}\frac{n_c+1}{n_c}.$$

Because $2 \le d \le 2^{W-1} - 1$ and $n_c \ge 2^{W-2}$,

$$\frac{2^W}{2^{W-1}-1} < m < \frac{2^W}{2}\frac{2^{W-2}+1}{2^{W-2}}, \quad \text{or}$$

$$3 \le m \le 2^{W-1} + 1.$$

Hence in either case m is within limits for the code schema illustrated by the "divide by 7" example.

Proof That the Product Is Correct

We must show that if p and m are calculated from (6) and (5), then Equations (1a) and (1b) are satisfied.

Equation (5) and inequality (6) are easily seen to imply (4). (In the case that p is forced to be equal to W, (6) still holds, as shown by Theorem DC1.) In what follows, we consider separately the following five ranges of values of n:

$$0 \le n \le n_c,$$

$$n_c + 1 \le n \le n_c + d - 1,$$

$$-n_c \le n \le -1,$$

$$-n_c - d + 1 \le n \le -n_c - 1, \quad \text{and}$$

$$n = -n_c - d.$$

From (4), because m is an integer,

$$\frac{2^p}{d} < m \le \frac{2^p(n_c+1)-1}{dn_c}.$$

Multiplying by $n / 2^p$, for $n \ge 0$ this becomes

$$\frac{n}{d} \le \frac{mn}{2^p} \le \frac{2^p n(n_c + 1) - n}{2^p dn_c}, \quad \text{so that}$$

$$\left\lfloor \frac{n}{d} \right\rfloor \le \left\lfloor \frac{mn}{2^p} \right\rfloor \le \left\lfloor \frac{n}{d} + \frac{(2^p - 1)n}{2^p dn_c} \right\rfloor.$$

For $0 \le n \le n_c$, $0 \le (2^p - 1)\, n \,/\, (2^p dn_c) < 1 \,/\, d$, so by Theorem D4,

$$\left\lfloor \frac{n}{d} + \frac{(2^p - 1)n}{2^p dn_c} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor.$$

Hence (1a) is satisfied in this case ($0 \le n \le n_c$).

For $n > n_c$, n is limited to the range

$$n_c + 1 \le n \le n_c + d - 1, \tag{9}$$

because $n \ge n_c + d$ contradicts the choice of n_c as the largest value of n such that rem(n_c, d) = $d - 1$ (alternatively, from (3a), $n \ge n_c + d$ implies $n \ge 2^{W-1}$). From (4), for $n \ge 0$,

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n}{d}\frac{n_c + 1}{n_c}.$$

By elementary algebra, this can be written

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n_c + 1}{d} + \frac{(n - n_c)(n_c + 1)}{dn_c}. \tag{10}$$

From (9), $1 \le n - n_c \le d - 1$, so

$$0 < \frac{(n - n_c)(n_c + 1)}{dn_c} \le \frac{d - 1}{d}\frac{n_c + 1}{n_c}.$$

Because $n_c \ge d - 1$ (by (3b)) and $(n_c + 1) \,/\, n_c$ has its maximum when n_c has its minimum,

$$0 < \frac{(n - n_c)(n_c + 1)}{dn_c} \le \frac{d - 1}{d}\frac{d - 1 + 1}{d - 1} = 1.$$

In (10), the term $(n_c + 1) \,/\, d$ is an integer. The term $(n - n_c)(n_c + 1) \,/\, dn_c$ is less than or equal to 1. Therefore, (10) becomes

$$\left\lfloor \frac{n}{d} \right\rfloor \le \left\lfloor \frac{mn}{2^p} \right\rfloor \le \frac{n_c+1}{d}.$$

For all n in the range (9), n/ d = (n_c + 1) / d. Hence, (1a) is satisfied in this case ($n_c + 1 \le n \le n_c + d - 1$).

For $n < 0$, from (4) we have, because m is an integer,

$$\frac{2^p+1}{d} \le m < \frac{2^p}{d}\frac{n_c+1}{n_c}.$$

Multiplying by $n / 2^p$, for $n < 0$ this becomes

$$\frac{n}{d}\frac{n_c+1}{n_c} < \frac{mn}{2^p} \le \frac{n}{d}\frac{2^p+1}{2^p},$$

or

$$\left\lfloor \frac{n}{d}\frac{n_c+1}{n_c} \right\rfloor + 1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lfloor \frac{n}{d}\frac{2^p+1}{2^p} \right\rfloor + 1.$$

Using Theorem D2 gives

$$\left\lceil \frac{n(n_c+1) - dn_c + 1}{dn_c} \right\rceil + 1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n(2^p+1) - 2^p d + 1}{2^p d} \right\rceil + 1,$$

$$\left\lceil \frac{n(n_c+1) + 1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n(2^p+1) + 1}{2^p d} \right\rceil.$$

Because $n + 1 \le 0$, the right inequality can be weakened, giving

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil. \qquad (11)$$

For $-n_c \le n \le -1$,

$$\frac{-n_c+1}{dn_c} \le \frac{n+1}{dn_c} \le 0, \quad \text{or}$$

$$-\frac{1}{d} < \frac{n+1}{dn_c} \le 0.$$

Hence, by Theorem D4,

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil = \left\lceil \frac{n}{d} \right\rceil,$$

so that (1b) is satisfied in this case ($-n_c \le n \le -1$).

For $n < -n_c$, n is limited to the range

$$-n_c - d \le n \le -n_c - 1. \qquad (12)$$

(From (3a), $n < -n_c - d$ implies that $n < -2^{W-1}$, which is impossible.) Performing elementary algebraic manipulation of the left comparand of (11) gives

$$\left\lceil \frac{-n_c - 1}{d} + \frac{(n + n_c)(n_c + 1) + 1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil. \qquad (13)$$

For $-n_c - d + 1 \le n \le -n_c - 1$,

$$\frac{(-d+1)(n_c+1)}{dn_c} + \frac{1}{dn_c} \le \frac{(n+n_c)(n_c+1)+1}{dn_c} \le \frac{-(n_c+1)+1}{dn_c} = -\frac{1}{d}.$$

The ratio $(n_c + 1) / n_c$ is a maximum when n_c is a minimum; that is, $n_c = d - 1$. Therefore,

$$\frac{(-d+1)(d-1+1)}{d(d-1)} + \frac{1}{dn_c} \le \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0, \quad \text{or}$$

$$-1 < \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0.$$

From (13), because $(-n_c - 1) / d$ is an integer and the quantity added to it is between 0 and –1,

$$\frac{-n_c - 1}{d} \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil.$$

For n in the range $-n_c - d + 1 \le n \le -n_c - 1$,

$$\left\lceil \frac{n}{d} \right\rceil = \frac{-n_c - 1}{d}.$$

Hence, $mn/2^p + 1 = n/d$ —that is, (1b) is satisfied.

The last case, $n = -n_c - d$, can occur only for certain values of d. From (3a), $-n_c - d \le -2^{W-1}$, so if n takes on this value, we must have $n = -n_c - d = -2^{W-1}$, and hence $n_c = 2^{W-1} - d$. Therefore, rem$(2^{W-1}, d)$ = rem$(n_c + d, d) = d - 1$ (that is, d

divides $2^{W-1} + 1$).

For this case ($n = -n_c - d$), (6) has the solution $p = W - 1$ (the smallest possible value of p), because for $p = W - 1$,

$$\begin{aligned} n_c(d - \text{rem}(2^p, d)) &= (2^{W-1} - d)(d - \text{rem}(2^{W-1}, d)) \\ &= (2^{W-1} - d)(d - (d-1)) = 2^{W-1} - d < 2^{W-1} = 2^p. \end{aligned}$$

Then from (5),

$$m = \frac{2^{W-1} + d - \text{rem}(2^{W-1}, d)}{d} = \frac{2^{W-1} + d - (d-1)}{d} = \frac{2^{W-1} + 1}{d}.$$

Therefore,

$$\begin{aligned} \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 &= \left\lfloor \frac{2^{W-1}+1}{d} \frac{-2^{W-1}}{2^{W-1}} \right\rfloor + 1 = \left\lfloor \frac{-2^{W-1}-1}{d} \right\rfloor + 1 \\ &= \left\lceil \frac{-2^{W-1}-d}{d} \right\rceil + 1 = \left\lceil \frac{-2^{W-1}}{d} \right\rceil = \left\lceil \frac{n}{d} \right\rceil, \end{aligned}$$

so that (1b) is satisfied.

This completes the proof that if m and p are calculated from (5) and (6), then Equations (1a) and (1b) hold for all admissible values of n.

10–5 Signed Division by Divisors ≤ –2

Because signed integer division satisfies $n \div (-d) = -(n \div d)$, it is adequate to generate code for $n \div |d|$ and follow it with an instruction to negate the quotient. (This does not give the correct result for $d = -2^{W-1}$, but for this and other negative powers of 2, you can use the code in Section 10–1, "Signed Division by a Known Power of 2," on page 205, followed by a negating instruction.) It will not do to negate the dividend, because of the possibility that it is the maximum negative number.

It is possible to avoid the negating instruction. The scheme is to compute

$$\begin{aligned} q &= \left\lfloor \frac{mn}{2^p} \right\rfloor && \text{if } n \le 0, \text{ and} \\ q &= \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 && \text{if } n > 0. \end{aligned}$$

Adding 1 if $n > 0$ is awkward (because one cannot simply use the sign bit of n), so the code will instead add 1 if $q < 0$. This is equivalent, because the multiplier m is negative (as will be seen).

The code to be generated is illustrated below for the case $W = 32$, $d = -7$.

```
li    M,0x6DB6DB6D  Magic num, -(2**34+5)/7 + 2**32.
mulhs q,M,n          q = floor(M*n/2**32).
```

```
sub   q,q,n         q = floor(M*n/2**32) - n.
shrsi q,q,2         q = floor(q/4).
shri  t,q,31        Add 1 to q if
add   q,q,t         q is negative (n is positive).

muli  t,q,-7        Compute remainder from
sub   r,n,t         r = n - q*(-7).
```

This code is the same as that for division by +7, except that it uses the negative of the multiplier for +7, and a `sub` rather than an `add` after the multiply, and the `shri` of 31 must use *q* rather than *n*, as discussed above. (The case of $d = +7$ could also use *q* here, but there would be less parallelism in the code.) The *subtract* will not overflow, because the operands have the same sign. This scheme, however, does not always work! Although the code above for $W = 32$, $d = -7$ is correct, the analogous alteration of the "divide by 3" code to produce code to divide by –3 does not give the correct result for $W = 32$, $n = -2^{31}$.

Let us look at the situation more closely.

Given a word size $W \geq 3$ and a divisor d, $-2^{W-1} \leq d \leq -2$, we wish to find the least (in absolute value) integer *m* and integer *p* such that

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \qquad \text{for } -2^{W-1} \leq n \leq 0, \text{ and} \tag{14a}$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \qquad \text{for } 1 \leq n < 2^{W-1}, \tag{14b}$$

with $-2^W \leq m \leq 0$ and $p \geq W$.

Proceeding similarly to the case of division by a positive divisor, let n_c be the most negative value of *n* such that $n_c = kd + 1$ for some integer *k*. n_c exists, because one possibility is $n_c = d + 1$. It can be calculated from $n_c =$ $(-2^{W-1} - 1) / d$ $d + 1 = -2^{W-1} + \text{rem}(2^{W-1} + 1, d)$. n_c is one of the least $|d|$ admissible values of *n*, so

$$-2^{W-1} \leq n_c \leq -2^{W-1} - d - 1, \tag{15a}$$

and, clearly

$$n_c \leq d + 1. \tag{15b}$$

Because (14b) must hold for $n = -d$, and (14a) must hold for $n = n_c$, we obtain, analogous to (4),

$$\frac{2^p}{d}\frac{n_c - 1}{n_c} < m < \frac{2^p}{d}. \tag{16}$$

Because *m* is to be the greatest integer satisfying (16), it is the next integer less than $2^p / d$—that is,

$$\boxed{m = \frac{2^p - d - \text{rem}(2^p, d)}{d}.} \qquad (17)$$

Combining this with the left half of (16) and simplifying gives

$$\boxed{2^p > n_c(d + \text{rem}(2^p, d)).} \qquad (18)$$

The proof that the algorithm suggested by (17) and (18) is feasible, and that the product is correct, is similar to that for a positive divisor, and will not be repeated. A difficulty arises in trying to prove that $-2^W \leq m \leq 0$. To prove this, consider separately the cases in which d is the negative of a power of 2, or some other number. For $d = -2^k$, it is easy to show that $n_c = -2^{W-1} + 1$, $p = W + k - 1$, and $m = -2^{W-1} - 1$ (which is within range). For d not of the form -2^k, it is straightforward to alter the earlier proof.

For Which Divisors Is $m(-d) \neq -m(d)$?

By $m(d)$ we mean the multiplier corresponding to a divisor d. If $m(-d) = -m(d)$, code for division by a negative divisor can be generated by calculating the multiplier for $|d|$, negating it, and then generating code similar to that of the "divide by –7" case illustrated above.

By comparing (18) with (6) and (17) with (5), it can be seen that if the value of n_c for $-d$ is the negative of that for d, then $m(-d) = -m(d)$. Hence, $m(-d) \neq m(d)$ can occur only when the value of n_c calculated for the negative divisor is the maximum negative number, -2^{W-1}. Such divisors are the negatives of the factors of $2^{W-1} + 1$. These numbers are fairly rare, as illustrated by the factorings below (obtained from Scratchpad).

$$2^{15} + 1 = 3 \cdot 11 \cdot 331$$

$$2^{31} + 1 = 3 \cdot 715{,}827{,}883$$

$$2^{63} + 1 = 3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77{,}158{,}673{,}929$$

For *all* these factors, $m(-d) \neq m(d)$. Proof sketch: For $d > 0$ we have $n_c = 2^{W-1} - d$. Because $\text{rem}(2^{W-1}, d) = d - 1$, (6) is satisfied by $p = W - 1$ and hence also by $p = W$. For $d < 0$, however, we have $n_c = -2^{W-1}$ and $\text{rem}(2^{W-1}, d) = |d| - 1$. Hence, (18) is not satisfied for $p = W - 1$ or for $p = W$, so $p > W$.

10–6 Incorporation into a Compiler

For a compiler to change division by a constant into a multiplication, it must compute the magic number M and the shift amount s, given a divisor d. The straightforward computation is to evaluate (6) or (18) for $p = W, W+1, \ldots$ until it is satisfied. Then, m is calculated from (5) or (17). M is simply a reinterpretation of m as a signed integer, and $s = p - W$.

The scheme described below handles positive and negative d with only a little extra code, and it avoids doubleword arithmetic.

Recall that n_c is given by

$$n_c = \begin{cases} 2^{W-1} - \text{rem}(2^{W-1}, d) - 1, & \text{if } d > 0, \\ -2^{W-1} + \text{rem}(2^{W-1} + 1, d), & \text{if } d < 0. \end{cases}$$

Hence, $|n_c|$ can be computed from

$$t = 2^{W-1} + \begin{cases} 0, & \text{if d} > 0, \\ 1, & \text{if d} < 0, \end{cases}$$

$$|n_c| = t - 1 - \text{rem}(t, |d|).$$

The remainder must be evaluated using unsigned division, because of the magnitude of the arguments. We have written rem(t, $|d|$) rather than the equivalent rem(t, d), to emphasize that the program must deal with two positive (and unsigned) arguments.

From (6) and (18), p can be calculated from

$$2^p > |n_c|(|d| - \text{rem}(2^p, |d|)), \tag{19}$$

and then $|m|$ can be calculated from (c.f. (5) and (17)):

$$|m| = \frac{2^p + |d| - \text{rem}(2^p, |d|)}{|d|}. \tag{20}$$

Direct evaluation of rem(2^p, $|d|$) in (19) requires "long division" (dividing a 2W-bit dividend by a W-bit divisor, giving a W-bit quotient and remainder), and, in fact, it must be *unsigned* long division. There is a way to solve (19), and to do all the calculations, that avoids long division and can easily be implemented in a conventional HLL using only W-bit arithmetic. We do, however, need unsigned division and unsigned comparisons.

We can calculate rem(2^p, $|d|$) incrementally, by initializing two variables q and r to the quotient and remainder of 2^p divided by $|d|$ with $p = W - 1$, and then updating q and r as p increases.

As the search progresses—that is, when p is incremented by 1—q and r are updated from (see Theorem D5(a))

```
q = 2*q;
r = 2*r;
if (r>= abs(d)) {
   q = q + 1;
   r = r - abs(d);}
```

The left half of inequality (4) and the right half of (16), together with the bounds proved for m, imply that $q = \lfloor 2^p/|d| \rfloor < 2^W$, so q is representable as a W-bit unsigned integer. Also, $0 \le r < |d|$, so r is representable as a W-bit signed or unsigned

$W - 1$

integer. (*Caution:* The intermediate result $2r$ can exceed 2 – 1, so r should be unsigned and the comparison above should also be unsigned.)

Next, calculate $\delta = |d| - r$. Both terms of the subtraction are representable as W-bit unsigned integers, and the result is also ($1 \le \delta \le |d|$), so there is no difficulty here.

To avoid the long multiplication of (19), rewrite it as

$$\frac{2^p}{|n_c|} > \delta.$$

The quantity $2^p / |n_c|$ is representable as a W-bit unsigned integer (similar to (7), from (19) it can be shown that $2^p \le 2|n_c| \cdot |d|$ and, for $d = -2^{W-1}$, $n_c = -2^{W-1} + 1$ and $p = 2W - 2$, so that $2^p / |n_c| = 2^{2W-2} / (2^{W-1} - 1) < 2^W$ for $W \ge 3$). Also, it is easily calculated incrementally (as p increases) in the same manner as for rem(2^p, $|d|$). The comparison should be unsigned, for the case $2^p / |n_c| \ge 2^{W-1}$ (which can occur, for large d).

To compute m, we need not evaluate (20) directly (which would require long division). Observe that

$$\frac{2^p + |d| - \mathrm{rem}(2^p, |d|)}{|d|} = \left\lfloor \frac{2^p}{|d|} \right\rfloor + 1 = q + 1.$$

The loop closure test $2^p / |n_c| > \delta$ is awkward to evaluate. The quantity $2^p / |n_c|$ is available only in the form of a quotient q_1 and a remainder r_1. $2^p / |n_c|$ may or may not be an integer (it is an integer only for $d = 2^{W-2} + 1$ and a few negative values of d). The test $2^p / |n_c| \le \delta$ can be coded as

$$q_1 < \boldsymbol{\delta} \mid (q_1 = \boldsymbol{\delta} \ \& \ r_1 = \mathbf{0}).$$

The complete procedure for computing `M` and `s` from `d` is shown in Figure 10–1, coded in C, for $W = 32$. There are a few places where overflow can occur, but the correct result is obtained if overflow is ignored.

To use the results of this program, the compiler should generate the `li` and `mulhs` instructions, generate the `add` if `d` > 0 and `M` < 0, or the `sub` if `d` < 0 and `M` > 0, and generate the `shrsi` if `s` > 0. Then, the `shri` and final `add` must be generated.

For $W = 32$, handling a negative divisor can be avoided by simply returning a precomputed result for $d = 3$ and $d = 715{,}827{,}883$, and using $m(-d) = -m(d)$ for other negative divisors. However, that program would not be significantly shorter, if at all, than the one given in Figure 10–1.

```
struct ms {int M;           // Magic number
          int s;};           // and shift amount.

struct ms magic(int d) {     // Must have 2 <= d <= 2**31-1
                            // or   -2**31 <= d <= -2.
   int p;
   unsigned ad, anc, delta, q1, r1, q2, r2, t;
   const unsigned two31 = 0x80000000;      // 2**31.
   struct ms mag;
   ad = abs(d);
   t = two31 + ((unsigned)d >> 31);
   anc = t - 1 - t%ad;       // Absolute value of nc.
   p = 31;                  // Init. p.
   q1 = two31/anc;           // Init. q1 = 2**p/|nc|.
   r1 = two31 - q1*anc;      // Init. r1 = rem(2**p, |nc|).
   q2 = two31/ad;            // Init. q2 = 2**p/|d|.
   r2 = two31 - q2*ad;       // Init. r2 = rem(2**p, |d|).
   do {
      p = p + 1;
      q1 = 2*q1;             // Update q1 = 2**p/|nc|.
      r1 = 2*r1;            // Update r1 = rem(2**p, |nc|).
      if (r1 >= anc) {       // (Must be an unsigned
         q1 = q1 + 1;        // comparison here.)
         r1 = r1 - anc;}
      q2 = 2*q2;             // Update q2 = 2**p/|d|.
      r2 = 2*r2;             // Update r2 = rem(2**p, |d|).
      if (r2 >= ad) {        // (Must be an unsigned
         q2 = q2 + 1;        // comparison here.)
         r2 = r2 - ad;}
      delta = ad - r2;
   } while (q1 < delta || (q1 == delta && r1 == 0));

   mag.M = q2 + 1;
   if (d < 0) mag.M = -mag.M; // Magic number and
   mag.s = p - 32;             // shift amount to return.
   return mag;
}
```

FIGURE 10–1. Computing the magic number for signed division.

10–7 Miscellaneous Topics

THEOREM DC2. *The least multiplier m is odd if p is not forced to equal W.*

Proof. Assume that Equations (1a) and (1b) are satisfied with least (not forced) integer p, and m even. Then clearly m could be divided by 2 and p could be decreased by 1, and (1a) and (1b) would still be satisfied. This contradicts the assumption that p is minimal.

Uniqueness

The magic number for a given divisor is sometimes unique (e.g., for $W = 32$, $d = 7$), but often it is not. In fact, experimentation suggests that it is usually not unique. For example, for $W = 32$, $d = 6$, there are four magic numbers:

$$\begin{aligned}
M &= \quad 715{,}827{,}833 \quad ((2^{32}+2)/6), & s &= 0 \\
M &= \quad 1{,}431{,}655{,}766 \quad ((2^{32}+2)/3), & s &= 1 \\
M &= -1{,}431{,}655{,}765 \quad ((2^{33}+1)/3-2^{32}), & s &= 2 \\
M &= -1{,}431{,}655{,}764 \quad ((2^{33}+4)/3-2^{32}), & s &= 2.
\end{aligned}$$

Nevertheless, there is the following uniqueness property:

THEOREM DC3. *For a given divisor d, there is only one multiplier m having the minimal value of p, if p is not forced to equal W.*

Proof. First consider the case $d > 0$. The difference between the upper and lower limits of inequality (4) is $2^p/dn_c$. We have already proved (7) that if p is minimal, then $2^p/dn_c \le 2$. Therefore, there can be at most two values of m satisfying (4). Let m be the smaller of these values, given by (5); then $m + 1$ is the other.

Let p_0 be the least value of p for which $m + 1$ satisfies the right half of (4) (p_0 is not forced to equal W). Then

$$\frac{2^{p_0} + d - \text{rem}(2^{p_0}, d)}{d} + 1 < \frac{2^{p_0}}{d}\frac{n_c + 1}{n_c}.$$

This simplifies to

$$2^{p_0} > n_c\,(2\,d - \text{rem}(2^{p_0}, d)).$$

Dividing by 2 gives

$$2^{p_0 - 1} > n_c\left(d - \frac{1}{2}\text{rem}(2^{p_0}, d)\right).$$

Because $\text{rem}(2^{p0}, d) \le 2\text{rem}(2^{p0-1}, d)$ (by Theorem D5 on page 184),

$$2^{p_0 - 1} > n_c\,(d - \text{rem}(2^{p_0 - 1}, d)),$$

contradicting the assumption that p_0 is minimal.

The proof for $d < 0$ is similar and will not be given.

The Divisors with the Best Programs

The program for $d = 3$, $W = 32$ is particularly short, because there is no `add` or `shrsi` after the `mulhs`. What other divisors have this short program?

We consider only positive divisors. We wish to find integers m and p that satisfy Equations (1a) and (1b), and for which $p = W$ and $0 \le m < 2^{W-1}$. Because any integers m and p that satisfy equations (1a) and (1b) must also satisfy (4), it suffices to find those divisors d for which (4) has a solution with $p = W$ and $0 \le m < 2^{W-1}$. All solutions of (4) with $p = W$ are given by

$$m = \frac{2^W + kd - \text{rem}(2^W, d)}{d}, \quad k = 1, 2, 3, \ldots.$$

Combining this with the right half of (4) and simplifying gives

$$\text{rem}(2^W, d) > kd - \frac{2^W}{n_c}. \qquad (21)$$

The weakest restriction on $\text{rem}(2^W, d)$ is with $k = 1$ and n_c at its minimal value of 2^{W-2}. Hence, we must have

$$\text{rem}(2^W, d) > d - 4;$$

that is, d divides $2^W + 1$, $2^W + 2$, or $2^W + 3$.

Now let us see which of these factors actually have optimal programs.

If d divides $2^W + 1$, then $\text{rem}(2^W, d) = d - 1$. Then a solution of (6) is $p = W$, because the inequality becomes

$$2^W > n_c(d - (d - 1)) = n_c,$$

which is obviously true, because $n_c < 2^{W-1}$. Then in the calculation of m we have

$$m = \frac{2^W + d - (d-1)}{d} = \frac{2^W + 1}{d},$$

which is less than 2^{W-1} for $d \geq 3$ ($d \neq 2$ because d divides $2^W + 1$). Hence, all the factors of $2^W + 1$ have optimal programs.

Similarly, if d divides $2^W + 2$, then $\text{rem}(2^W, d) = d - 2$. Again, a solution of (6) is $p = W$, because the inequality becomes

$$2^W > n_c(d - (d - 2)) = 2n_c,$$

which is obviously true. Then in the calculation of m we have

$$m = \frac{2^W + d - (d-2)}{d} = \frac{2^W + 2}{d},$$

which exceeds $2^{W-1} - 1$ for $d = 2$, but which is less than or equal to $2^{W-1} - 1$ for $W \geq 3$, $d \geq 3$ (the case $W = 3$ and $d = 3$ does not occur, because 3 is not a factor of $2^3 + 2 = 10$). Hence all factors of $2^W + 2$, except for 2 and the cofactor of 2, have optimal programs. (The cofactor of 2 is $(2^W + 2) / 2$, which is not representable as a W-bit signed integer).

If d divides $2^W + 3$, the following argument shows that d does not have an optimal program. Because $\text{rem}(2^W, d) = d - 3$, inequality (21) implies that we must have

$$n_c < \frac{2^W}{kd - d + 3}$$

for some $k = 1, 2, 3, \ldots$. The weakest restriction is with $k = 1$, so we must have $n_c < 2^W / 3$.

From (3a), $n_c \geq 2^{W-1} - d$, or $d \geq 2^{W-1} - n_c$. Hence, it is necessary that

$$d > 2^{W-1} - \frac{2^W}{3} = \frac{2^W}{6}.$$

Also, because 2, 3, and 4 do not divide $2^W + 3$, the smallest possible factor of $2^W + 3$ is 5. Therefore, the largest possible factor is $(2^W + 3) / 5$. Thus, if d divides $2^W + 3$ and d has an optimal program, it is necessary that

$$\frac{2^W}{6} < d \le \frac{2^W + 3}{5}.$$

Taking reciprocals of this with respect to $2^W + 3$ shows that the cofactor of *d*, $(2^W + 3) / d$, has the limits

$$5 \le \frac{2^W + 3}{d} < \frac{(2^W + 3) \cdot 6}{2^W} = 6 + \frac{18}{2^W}.$$

For $W \ge 5$, this implies that the only possible cofactors are 5 and 6. For $W < 5$, it is easily verified that there are no factors of $2^W + 3$. Because 6 cannot be a factor of $2^W + 3$, the only possibility is 5. Therefore, the only possible factor of $2^W + 3$ that might have an optimal program is $(2^W + 3) / 5$.

For $d = (2^W + 3) / 5$,

$$n_c = \left\lfloor \frac{2^{W-1}}{(2^W + 3)/5} \right\rfloor \left(\frac{2^W + 3}{5} \right) - 1.$$

For $W \ge 4$,

$$2 < \frac{2^{W-1}}{(2^W + 3)/5} < 2.5,$$

so

$$n_c = 2\left(\frac{2^W + 3}{5} \right) - 1.$$

This exceeds $(2^W / 3)$, so $d = (2^W + 3) / 5$ does not have an optimal program. Because for $W < 4$ there are no factors of $2^W + 3$, we conclude that no factors of $2^W + 3$ have optimal programs.

In summary, all the factors of $2^W + 1$ and of $2^W + 2$, except for 2 and $(2^W + 2) / 2$, have optimal programs, and no other numbers do. Furthermore, the above proof shows that algorithm *magic* (Figure 10–1 on page 223) always produces the optimal program when it exists.

Let us consider the specific cases $W = 16$, 32, and 64. The relevant factorizations are shown below.

$$2^{16} + 1 = 65537 \text{ (prime)} \qquad 2^{32} + 1 = 641 \cdot 6{,}700{,}417$$

$$2^{16} + 2 = 2 \cdot 3^2 \cdot 11 \cdot 331 \qquad 2^{32} + 2 = 2 \cdot 3 \cdot 715{,}827{,}883$$

$$2^{64} + 1 = 274{,}177 \cdot 67{,}280{,}421{,}310{,}721$$

$$2^{64} + 2 = 2 \cdot 3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77{,}158{,}673{,}929$$

The result for $W = 16$ is that there are 20 divisors that have optimal programs. The ones less than 100 are 3, 6, 9, 11, 18, 22, 33, 66, and 99.

For $W = 32$, there are six such divisors: 3, 6, 641, 6,700,417, 715,827,883, and 1,431,655,766.

For $W = 64$, there are 126 such divisors. The ones less than 100 are 3, 6, 9, 18, 19, 27, 38, 43, 54, 57, and 86.

10–8 Unsigned Division

Unsigned division by a power of 2 is, of course, implemented by a single *shift right logical* instruction, and remainder by *and immediate*.

It might seem that handling other divisors will be simple: Just use the results for signed division with $d > 0$, omitting the two instructions that add 1 if the quotient is negative. We will see, however, that some of the details are actually more complicated in the case of unsigned division.

Unsigned Division by 3

For a non-power of 2, let us first consider unsigned division by 3 on a 32-bit machine. Because the dividend n can now be as large as $2^{32} - 1$, the multiplier $(2^{32} + 2) / 3$ is inadequate, because the error term $2n / 3 \cdot 2^{32}$ (see “divide by 3” example above) can exceed 1/3. However, the multiplier $(2^{33} + 1) / 3$ is adequate. The code is

```
li    M,0xAAAAAAAB  Load magic number, (2**33+1)/3.
mulhu q,M,n         q = floor(M*n/2**32).
shri  q,q,1

muli  t,q,3         Compute remainder from
sub   r,n,t         r = n - q*3.
```

An instruction that gives the high-order 32 bits of a 64-bit unsigned product is required, which we show above as `mulhu`.

To see that the code is correct, observe that it computes

$$q = \left\lfloor \frac{2^{33}+1}{3}\frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{n}{3 \cdot 2^{33}} \right\rfloor.$$

For $0 \le n < 2^{32}$, $0 \le n / (3 \cdot 2^{33}) < 1 / 3$, so by Theorem D4, $q = n/3$.

In computing the remainder, the *multiply immediate* can overflow if we regard the operands as signed integers, but it does not overflow if we regard them and the result as unsigned. Also, the *subtract* cannot overflow, because the result is in the range 0 to 2, so the remainder is correct.

Unsigned Division by 7

For unsigned division by 7 on a 32-bit machine, the multipliers $(2^{32} + 3) / 7$, $(2^{33} + 6) / 7$, and $(2^{34} + 5) / 7$ are all inadequate, because they give too large an error term. The multiplier $(2^{35} + 3) / 7$ is acceptable, but it’s too large to represent in a 32-bit unsigned word. We can multiply by this large number by multiplying by $(2^{35} + 3) / 7 - 2^{32}$ and then correcting the product by inserting an `add`. The code is

```
li    M,0x24924925  Magic num, (2**35+3)/7 - 2**32.
mulhu q,M,n          q = floor(M*n/2**32).
add   q,q,n          Can overflow (sets carry).
shrxi q,q,3          Shift right with carry bit.

muli  t,q,7          Compute remainder from
sub   r,n,t          r = n - q*7.
```

Here we have a problem: The `add` can overflow. To allow for this, we have invented the new instruction *shift right extended immediate* (`shrxi`), which treats the carry from the `add` and the 32 bits of register *q* as a single 33-bit quantity, and shifts it right with 0-fill. On the Motorola 68000 family, this can be done with two instructions: *rotate with extend* right one position, followed by a logical right shift of three (`roxr` actually uses the X bit, but the `add` sets the X bit the same as the carry bit). On most machines, it will take more. For example, on PowerPC it takes three instructions: clear rightmost three bits of *q*, add carry to *q*, and rotate right three positions.

With `shrxi` implemented somehow, the code above computes

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{35}+3}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) / 2^3 \right\rfloor = \left\lfloor \frac{n}{7} + \frac{3n}{7 \cdot 2^{35}} \right\rfloor.$$

For $0 \le n < 2^{32}$, $0 \le 3n/(7 \cdot 2^{35}) < 1/7$, so by Theorem D4, $q = \quad n / 7 \quad$.

Granlund and Montgomery [GM] have a clever scheme for avoiding the `shrxi` instruction. It requires the same number of instructions as the above three-instruction sequence for `shrxi`, but it employs only elementary instructions that almost any machine would have, and it does not cause overflow at all. It uses the identity

$$\left\lfloor \frac{q+n}{2^p} \right\rfloor = \left\lfloor \left(\left\lfloor \frac{n-q}{2} \right\rfloor + q \right) / 2^{p-1} \right\rfloor, \quad p \ge 1.$$

Applying this to our problem, with $q = \quad Mn / 2^{32} \quad$ where $0 \le M < 2^{32}$, the subtraction will not overflow, because

$$0 \le q = \left\lfloor \frac{Mn}{2^{32}} \right\rfloor \le n,$$

so that, clearly, $0 \le n - q < 2^{32}$. Also, the addition will not overflow, because

$$\left\lfloor \frac{n-q}{2} \right\rfloor + q = \left\lfloor \frac{n-q}{2} + q \right\rfloor = \left\lfloor \frac{n+q}{2} \right\rfloor,$$

and $0 \le n,q < 2^{32}$.

Using this idea gives the following code for unsigned division by 7:

```
li    M,0x24924925  Magic num, (2**35+3)/7 - 2**32.
mulhu q,M,n          q = floor(M*n/2**32).
sub   t,n,q          t = n - q.
shri  t,t,1          t = (n - q)/2.
add   t,t,q          t = (n - q)/2 + q = (n + q)/2.
```

```
shri  q,t,2        q = (n+Mn/2**32)/8 = floor(n/7).

muli  t,q,7        Compute remainder from
sub   r,n,t        r = n - q*7.
```

For this to work, the shift amount for the hypothetical `shrxi` instruction must be greater than 0. It can be shown that if $d > 1$ and the multiplier $m \geq 2^{32}$ (so that the `shrxi` instruction is needed), then the shift amount is greater than 0.

10–9 Unsigned Division by Divisors ≥ 1

Given a word size $W \geq 1$ and a divisor d, $1 \leq d < 2^W$, we wish to find the least integer m and integer p such that

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \quad \text{for } 0 \leq n < 2^W, \tag{22}$$

with $0 \leq m < 2^{W+1}$ and $p \geq W$.

In the unsigned case, the magic number M is given by

$$M = \begin{cases} m, & \text{if } 0 \leq m < 2^W, \\ m - 2^W, & \text{if } 2^W \leq m < 2^{W+1}. \end{cases}$$

Because (22) must hold for $n = d$, $md/ 2^p$ = 1, or

$$\frac{md}{2^p} \geq 1. \tag{23}$$

As in the signed case, let n_c be the largest value of n such that rem(n_c, d) = $d - 1$. It can be calculated from n_c = $2^W / d$ $d - 1 = 2^W -$ rem(2^W, d) – 1. Then

$$2^W - d \leq n_c \leq 2^W - 1, \tag{24a}$$

and

$$n_c \geq d - 1. \tag{24b}$$

These imply that $n_c \geq 2^{W-1}$.

Because (22) must hold for $n = n_c$

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d},$$

or

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}.$$

Combining this with (23) gives

$$\frac{2^p}{d} \le m < \frac{2^p}{d}\frac{n_c+1}{n_c}. \tag{25}$$

Because m is to be the least integer satisfying (25), it is the next integer greater than or equal to $2^p / d$—that is,

$$\boxed{m = \frac{2^p + d - 1 - \text{rem}(2^p - 1, d)}{d}.} \tag{26}$$

Combining this with the right half of (25) and simplifying gives

$$\boxed{2^p > n_c(d - 1 - \text{rem}(2^p - 1, d)).} \tag{27}$$

The Algorithm (Unsigned)

Thus, the algorithm is to find by trial and error the least $p \ge W$ satisfying (27). Then, m is calculated from (26). This is the smallest possible value of m satisfying (22) with $p \ge W$. As in the signed case, if (27) is true for some value of p, then it is true for all larger values of p. The proof is essentially the same as that of Theorem DC1, except Theorem D5(b) is used instead of Theorem D5(a).

Proof That the Algorithm Is Feasible (Unsigned)

We must show that (27) always has a solution and that $0 \le m < 2^{W+1}$.

Because for any nonnegative integer x there is a power of 2 greater than x and less than or equal to $2x + 1$, from (27),

$$n_c(d - 1 - \text{rem}(2^p - 1, d)) < 2^p \le 2n_c(d - 1 - \text{rem}(2^p - 1, d)) + 1.$$

Because $0 \le \text{rem}(2^p - 1, d) \le d - 1$,

$$1 \le 2^p \le 2n_c(d-1) + 1. \tag{28}$$

Because $n_c, d \le 2^W - 1$, this becomes

$$1 \le 2^p \le 2(2^W - 1)(2^W - 2) + 1,$$

or

$$0 \le p \le 2W. \tag{29}$$

Thus, (27) always has a solution.

If p is not forced to equal W, then from (25) and (28),

$$\frac{1}{d} \le m < \frac{2n_c(d-1)+1}{d}\frac{n_c+1}{n_c},$$

$$1 \le m < \frac{2d-2+1/n_c}{d}(n_c+1),$$

$$1 \le m < 2(n_c+1) \le 2^{W+1}.$$

If p is forced to equal W, then from (25),

$$\frac{2^W}{d} \le m < \frac{2^W}{d}\frac{n_c+1}{n_c}.$$

Because $1 \le d \le 2^W - 1$ and $n_c \ge 2^{W-1}$,

$$\frac{2^W}{2^W-1} \le m < \frac{2^W}{1}\frac{2^{W-1}+1}{2^{W-1}},$$

$$2 \le m \le 2^W + 1.$$

In either case m is within limits for the code schema illustrated by the “unsigned divide by 7” example.

Proof That the Product Is Correct (Unsigned)

We must show that if p and m are calculated from (27) and (26), then (22) is satisfied.

Equation (26) and inequality (27) are easily seen to imply (25). Inequality (25) is nearly the same as (4), and the remainder of the proof is nearly identical to that for signed division with $n \ge 0$.

10–10 Incorporation into a Compiler (Unsigned)

There is a difficulty in implementing an algorithm based on direct evaluation of the expressions used in this proof. Although $p \le 2W$, which is proved above, the case $p = 2W$ can occur (e.g., for $d = 2^W - 2$ with $W \ge 4$). When $p = 2W$, it is difficult to calculate m, because the dividend in (26) does not fit in a $2W$-bit word.

However, it can be implemented by the “incremental division and remainder” technique of algorithm *magic*. The algorithm is given in Figure 10–2 for $W = 32$. It passes back an indicator `a`, which tells whether or not to generate an `add` instruction. (In the case of signed division, the caller recognizes this by `M` and `d` having opposite signs.)

Some key points in understanding this algorithm are as follows:

- Unsigned overflow can occur at several places and should be ignored.
- $n_c = 2^W - \text{rem}(2^W, d) - 1 = (2^W - 1) - \text{rem}(2^W - d, d)$.
- The quotient and remainder of dividing 2^p by n_c cannot be updated in the same way as is done in algorithm *magic*, because here the quantity `2*r1` can overflow. Hence, the algorithm has the test “`if (r1 > = nc - r l)`,” whereas “`if (2*rl >= nc`

)" would be more natural. A similar remark applies to computing the quotient and remainder of $2^P - 1$ divided by *d*.

- $0 \le \delta \le d - 1$, so δ is representable as a 32-bit unsigned integer.

```
struct mu {unsigned M;      // Magic number,
          int a;            // "add" indicator,
          int s;};          // and shift amount.

struct mu magicu(unsigned d) {
                              // Must have 1 <= d <= 2**32-1.
   int p;
   unsigned nc, delta, q1, r1, q2, r2;
   struct mu magu;

   magu.a = 0;               // Initialize "add" indicator.
   nc = -1 - (-d)%d;         // Unsigned arithmetic here.
   p = 31;                   // Init. p.
   q1 = 0x80000000/nc;       // Init. q1 = 2**p/nc.
   r1 = 0x80000000 - q1*nc;// Init. r1 = rem(2**p, nc).
   q2 = 0x7FFFFFFF/d;        // Init. q2 = (2**p - 1)/d.
   r2 = 0x7FFFFFFF - q2*d; // Init. r2 = rem(2**p - 1, d).
   do {
      p = p + 1;
      if (r1 >= nc - r1) {
         q1 = 2*q1 + 1;            // Update q1.
         r1 = 2*r1 - nc;}          // Update r1.
      else {
         q1 = 2*q1;
         r1 = 2*r1;}
      if (r2 + 1 >= d - r2) {
         if (q2 >= 0x7FFFFFFF) magu.a = 1;
         q2 = 2*q2 + 1;            // Update q2.
         r2 = 2*r2 + 1 - d;}       // Update r2.
      else {
         if (q2 >= 0x80000000) magu.a = 1;
         q2 = 2*q2;
         r2 = 2*r2 + 1;}
      delta = d - 1 - r2;
   } while (p < 64 &&
           (q1 < delta || (q1 == delta && r1 == 0)));

   magu.M = q2 + 1;          // Magic number
   magu.s = p - 32;          // and shift amount to return
   return magu;              // (magu.a was set above).
}
```

FIGURE 10–2. Computing the magic number for unsigned division.

- $m = (2^p + d - 1 - \text{rem}(2^p - 1, d)) / d = \quad (2^p - 1) / d \quad +1 = q_2 + 1$.
- The subtraction of 2^W when the magic number `M` exceeds $2^W - 1$ is not explicit in the program; it occurs if the computation of `q2` overflows.
- The "add" indicator, `magu.a`, cannot be set by a straightforward comparison of `M` to 2^{32}, or of `q2` to $2^{32} - 1$, because of overflow. Instead, the program tests `q2` before overflow can occur. If `q2` ever gets as large as $2^{32} - 1$, so that `M` will be greater than or equal to 2^{32}, then `magu.a` is set equal to 1. If `q2` stays below $2^{32} - 1$, then `magu.a` is left at its initial value of 0.

p

- Inequality (27) is equivalent to $2 \ / n_c > \delta$.
- The loop test needs the condition `p < 64`, because without it, overflow of `q1` would cause the program to loop too many times, giving incorrect results.

To use the results of this program, the compiler should generate the `li` and `mulhu` instructions and, if the "add" indicator `a` = 0, generate the `shri` of `s` (if `s` > 0), as illustrated by the example of "Unsigned Division by 3," on page 227. If `a` = 1 and the machine has the `shrxi` instruction, the compiler should generate the `add` and `shrxi` of `s` as illustrated by the example of "Unsigned Division by 7," on page 228. If `a` = 1 and the machine does not have the `shrxi` instruction, use the example on page 229: generate the `sub`, the `shri` of 1, the `add`, and finally the `shri` of `s` – 1 (if `s` – 1 > 0; `s` will not be 0 at this point except in the trivial case of division by 1, which we assume the compiler deletes).

10–11 Miscellaneous Topics (Unsigned)

THEOREM DC2U. *The least multiplier m is odd if p is not forced to equal W.*

THEOREM DC3U. *For a given divisor d, there is only one multiplier m having the minimal value of p, if p is not forced to equal W.*

The proofs of these theorems follow very closely the corresponding proofs for signed division.

The Divisors with the Best Programs (Unsigned)

For unsigned division, to find the divisors (if any) with optimal programs of two instructions to obtain the quotient (`li`, `mulhu`), we can do an analysis similar to that of the signed case (see "The Divisors with the Best Programs" on page 225). The result is that such divisors are the factors of 2^W or $2^W + 1$, except for $d = 1$. For the common word sizes, this leaves very few nontrivial divisors that have optimal programs for unsigned division. For $W = 16$, there are none. For $W = 32$, there are only two: 641 and 6,700,417. For $W = 64$, again there are only two: 274,177 and 67,280,421,310,721.

The case $d = 2^k$, $k = 1, 2, \ldots$, deserves special mention. In this case, algorithm *magicu* produces $p = W$ (forced), $m = 2^{32 - k}$. This is the minimal value of m, but it is not the minimal value of M. Better code results if $p = W + k$ is used, if sufficient simplifications are done. Then, $m = 2^W$, $M = 0$, $a = 1$, and $s = k$. The generated code involves a multiplication by 0 and can be simplified to a single *shift right k* instruction. As a practical matter, divisors that are a power of 2 would probably be special-cased without using *magicu*. (This phenomenon does not occur for signed division, because for signed division m cannot be a power of 2. Proof: For $d > 0$, inequality (4) combined with (3b) implies that $d - 1 < 2^p / m < d$. Therefore, $2^p / m$ cannot be an integer. For $d < 0$, the result follows similarly from (16) combined with (15b).)

For unsigned division, the code for the case $m \geq 2^W$ is considerably worse than the code for the case $m < 2^W$ if the machine does not have `shrxi`. It is of interest to have some idea of how often the large multipliers arise. For $W = 32$, among the integers less than or equal to 100, there are 31 "bad" divisors: 1, 7, 14, 19, 21, 27, 28, 31, 35, 37, 38, 39, 42, 45, 53, 54, 55, 56, 57, 62, 63, 70, 73, 74, 76, 78, 84, 90, 91, 95, and 97.

Using Signed in Place of Unsigned Multiply, and the Reverse

If your machine does not have `mulhu`, but it does have `mulhs` (or signed long multiplication), the trick given in "High-Order Product Signed from/to Unsigned," on page 174, might make our method of doing unsigned division by a constant still useful.

That section gives a seven-instruction sequence for getting `mulhu` from `mulhs`. However, for this application it simplifies, because the magic number *M* is known. Thus, the compiler can test the most significant bit of the magic number, and generate code such as the following for the operation "`mulhu q,M,n`." Here `t` denotes a temporary register.

```
  M31 = 0           M31 = 1
mulhs q,M,n       mulhs q,M,n
shrsi t,n,31      shrsi t,n,31
and   t,t,M       and   t,t,M
add   q,q,t       add   t,t,n
                  add   q,q,t
```

Accounting for the other instructions used with `mulhu`, this uses a total of six to eight instructions to obtain the quotient of unsigned division by a constant on a machine that does not have unsigned multiply.

This trick can be inverted, to get `mulhs` in terms of `mulhu`. The code is the same as that above, except the `mulhs` is changed to `mulhu` and the final `add` in each column is changed to `sub`.

A Simpler Algorithm (Unsigned)

Dropping the requirement that the magic number be minimal yields a simpler algorithm. In place of (27) we can use

$$2^p \geq 2^W(d - 1 - \text{rem}(2^p - 1, d)), \qquad (30)$$

and then use (26) to compute *m*, as before.

It should be clear that this algorithm is formally correct (that is, that the value of *m* computed does satisfy Equation (22)), because its only difference from the previous algorithm is that it computes a value of *p* that, for some values of *d*, is unnecessarily large. It can be proved that the value of *m* computed from (30) and (26) is less than 2^{W+1}. We omit the proof and simply give the algorithm (Figure 10–3).

```
struct mu {unsigned M;      // Magic number,
          int a;            // "add" indicator,
          int s;};          // and shift amount.

struct mu magicu2(unsigned d) {
                            // Must have 1 <= d <= 2**32-1.
   int p;
   unsigned p32, q, r, delta;
   struct mu magu;
   magu.a = 0;              // Initialize "add" indicator.
   p = 31;                  // Initialize p.
   q = 0x7FFFFFFF/d;        // Initialize q = (2**p - 1)/d.
   r = 0x7FFFFFFF - q*d;    // Init. r = rem(2**p - 1, d).
   do {
      p = p + 1;
      if (p == 32) p32 = 1;     // Set p32 = 2**(p-32).
      else p32 = 2*p32;
      if (r + 1 >= d - r) {
         if (q >= 0x7FFFFFFF) magu.a = 1;
         q = 2*q + 1;           // Update q.
         r = 2*r + 1 - d;       // Update r.
      }
      else {
         if (q >= 0x80000000) magu.a = 1;
         q = 2*q;
         r = 2*r + 1;
      }
      delta = d - 1 - r;
   } while (p < 64 && p32 < delta);
   magu.M = q + 1;          // Magic number and
   magu.s = p - 32;         // shift amount to return
   return magu;             // (magu.a was set above).
}
```

FIGURE 10–3. Simplified algorithm for computing the magic number, unsigned division.

Alverson [Alv] gives a much simpler algorithm, discussed in the next section, but it gives somewhat large values for *m*. The point of algorithm *magicu2* is that it nearly always gives the minimal value for *m* when $d \le 2^{W-1}$. For $W = 32$, the smallest divisor for which *magicu2* does not give the minimal multiplier is d = 102,807, for which *magicu* calculates m = 2,737,896,999 and *magicu2* calculates m = 5,475,793,997.

There is an analog of *magicu2* for signed division by positive divisors, but it does not work out very well for signed division by arbitrary divisors.

10–12 Applicability to Modulus and Floor Division

It might seem that turning modulus or floor division by a constant into multiplication would be simpler, in that the "add 1 if the dividend is negative" step could be omitted. This is not the case. The methods given above do not apply in any obvious way to modulus and floor division. Perhaps something could be worked out; it might involve altering the multiplier *m* slightly, depending upon the sign of the dividend.

10–13 Similar Methods

Rather than coding algorithm *magic*, we can provide a table that gives the magic numbers and shift amounts for a few small divisors. Divisors equal to the tabulated ones multiplied by a power of 2 are easily handled as follows:

1. Count the number of trailing 0's in *d*, and let this be denoted by *k*.
2. Use as the lookup argument $d / 2^k$ (shift right *k*).
3. Use the magic number found in the table.
4. Use the shift amount found in the table, increased by *k*.

Thus, if the table contains the divisors 3, 5, 25, and so on, divisors of 6, 10, 100, and so forth can be handled.

This procedure usually gives the smallest magic number, but not always. The smallest positive divisor for which it fails in this respect for $W = 32$ is d = 334,972, for which it computes m = 3,361,176,179 and s = 18. However, the minimal magic number for d = 334,972 is m = 840,294,045, with $s = 16$. The procedure also fails to give the minimal magic number for $d = –6$. In both these cases, output code quality is affected.

Alverson [Alv] is the first known to the author to state that the method described here works with complete accuracy for all divisors. Using our notation, his method for unsigned integer division by *d* is to set the shift amount $p = W + \quad \log_2 d \quad$, and the
p *p*

multiplier $m = \quad 2 \quad / d$ and then do the division by $n \div d = \quad mn / 2$ (that is, *multiply* and *shift right*). He proves that the multiplier m is less than, 2^{W+1} and that the method gets the exact quotient for all n expressible in W bits.

Alverson's method is a simpler variation of ours in that it doesn't require trial and error to determine p, and is therefore more suitable for building in hardware, which is his primary interest. His multiplier m is always greater than or equal to 2^W, and hence for the software application always gives the code illustrated by the "unsigned divide by 7" example (that is, always has the `add` and `shrxi`, or the alternative four instructions). Because most small divisors can be handled with a multiplier less than 2^W, it seems worthwhile to look for these cases.

For signed division, Alverson suggests finding the multiplier for $|d|$ and a word length of $W - 1$ (then $2^{W-1} \le m < 2^W$), multiplying the dividend by it, and negating the result if the operands have opposite signs. (The multiplier must be such that it gives the correct result when the dividend is 2^{W-1}, the absolute value of the maximum negative number.) It seems possible that this suggestion might give better code than what has been given here in the case that the multiplier $m \ge 2^W$. Applying it to signed division by 7 gives the following code, where we have used the relation $-x = \bar{x} + 1$ to avoid a branch:

```
abs   an,n
li    M,0x92492493  Magic number, (2**34+5)/7.
mulhu q,M,an        q = floor(M*an/2**32).
shri  q,q,2
shrsi t,n,31        These three instructions
xor   q,q,t         negate q if n is
sub   q,q,t         negative.
```

This is not quite as good as the code we gave for signed division by 7 (six versus seven instructions), but it would be useful on a machine that has `abs` and `mulhu`, but not `mulhs`.

The next section gives some representative magic numbers.

10–14 Sample Magic Numbers

TABLE 10–1. SOME MAGIC NUMBERS FOR W = 32

	Signed		Unsigned		
d	M (hex)	s	M (hex)	a	s
–5	99999999	1			
–3	55555555	1			
-2^k	7FFFFFFF	k–1			
1	–	–	0	1	0
2^k	80000001	k–1	2^{32-k}	0	0
3	55555556	0	AAAAAAAB	0	1
5	66666667	1	CCCCCCCD	0	2
6	2AAAAAAB	0	AAAAAAAB	0	2
7	92492493	2	24924925	1	3
9	38E38E39	1	38E38E39	0	1
10	66666667	2	CCCCCCCD	0	3
11	2E8BA2E9	1	BA2E8BA3	0	3
12	2AAAAAAB	1	AAAAAAAB	0	3
25	51EB851F	3	51EB851F	0	3
125	10624DD3	3	10624DD3	0	3
625	68DB8BAD	8	D1B71759	0	9

TABLE 10–2. SOME MAGIC NUMBERS FOR *W* = 64

d	Signed		Unsigned		
	M (hex)	s	M (hex)	a	s
-5	99999999 99999999	1			
-3	55555555 55555555	1			
-2^k	7FFFFFFF FFFFFFFF	$k-1$			
1	–	–	0	1	0
2^k	80000000 00000001	$k-1$	2^{64-k}	0	0
3	55555555 55555556	0	AAAAAAAA AAAAAAAB	0	1
5	66666666 66666667	1	CCCCCCCC CCCCCCCD	0	2
6	2AAAAAAA AAAAAAAB	0	AAAAAAAA AAAAAAAB	0	2
7	49249249 24924925	1	24924924 92492493	1	3
9	1C71C71C 71C71C72	0	E38E38E3 8E38E38F	0	3
10	66666666 66666667	2	CCCCCCCC CCCCCCCD	0	3
11	2E8BA2E8 BA2E8BA3	1	2E8BA2E8 BA2E8BA3	0	1
12	2AAAAAAA AAAAAAAB	1	AAAAAAAA AAAAAAAB	0	3
25	A3D70A3D 70A3D70B	4	47AE147A E147AE15	1	5
125	20C49BA5 E353F7CF	4	0624DD2F 1A9FBE77	1	7
625	346DC5D6 3886594B	7	346DC5D6 3886594B	0	7

10–15 Simple Code in Python

Computing a magic number is greatly simplified if one is not limited to doing the calculations in the same word size as that of the environment in which the magic number will be used. For the unsigned case, for example, in Python it is straightforward to compute n_c and then evaluate Equations (27) and (26), as described in Section 10–9. Figure 10–4 shows such a function.

```
def magicgu(nmax, d):
   nc = (nmax//d)*d - 1
   nbits = int(log(nmax, 2)) + 1
   for p in range(0, 2*nbits + 1):
      if 2**p > nc*(d - 1 - (2**p - 1)%d):
         m = (2**p + d - 1 - (2**p - 1)%d)//d
         return (m, p)
   print "Can't find p, something is wrong."
   sys.exit(1)
```

FIGURE 10–4. Python code for computing the magic number for unsigned division.

The function is given the maximum value of the dividend `nmax` and the divisor `d`. It returns a pair of integers: the magic number `m` and a shift amount `p`. To divide a dividend `x` by `d`, one multiplies `x` by `m` and then shifts the (full length) product right `p` bits.

This program is more general than the others in this chapter in two ways: (1) one specifies the maximum value of the dividend (`nmax`), rather than the number of bits required for the dividend, and (2) the program can be used for arbitrarily large dividends and divisors ("bignums"). The advantage of specifying the maximum value of the dividend is that one sometimes gets a smaller magic number than would be obtained if the next power of two less 1 were used for the maximum value. For example, suppose the maximum value of the dividend is 90, and the divisor is 7. Then function `magicgu` returns (37, 8), meaning that the magic number is 37 (a 6-bit number) and the shift amount is 8. But if we asked for a magic number that can handle divisors up to 127, then the result is (147, 10), and 147 is an 8-bit number.

10–16 Exact Division by Constants

By "exact division," we mean division in which it is known beforehand, somehow, that the remainder is 0. Although this situation is not common, it does arise, for example, when subtracting two pointers in the C language. In C, the result of $p - q$, where p and q are pointers, is well defined and portable only if p and q point to objects in the same array [H&S, sec. 7.6.2]. If the array element size is s, the object code for the difference $p - q$ computes $(p - q) / s$.

The material in this section was motivated by [GM, sec. 9].

The method to be given applies to both signed and unsigned exact division, and is based on the following theorem.

THEOREM MI. *If a and m are relatively prime integers, then there exists an integer* $\bar{a}$, $1 \le \bar{a} < m$, *such that*

$$a\bar{a} \equiv 1 \pmod{m}.$$

That is, $\bar{a}$ is a multiplicative inverse of a, modulo m. There are several ways to prove this theorem; three proofs are given in [NZM, p. 52]. The proof below requires only a very basic familiarity with congruences.

Proof. We will prove something a little more general than the theorem. If a and m are relatively prime (therefore nonzero), then as x ranges over all m distinct values modulo m, ax takes on all m distinct values modulo m. For example, if $a = 3$ and $m = 8$, then as x ranges from 0 to 7, ax = 0, 3, 6, 9, 12, 15, 18, 21 or, reduced modulo 8, ax = 0, 3, 6, 1, 4, 7, 2, 5. Observe that all values from 0 to 7 are present in the last sequence.

To see this in general, assume that it is not true. Then there exist distinct integers that map to the same value when multiplied by a; that is, there exist x and y, with $x \not\equiv y \pmod{m}$, such that

$$ax \equiv ay \pmod{m}.$$

Then there exists an integer k such that

$$ax - ay = km, \text{ or}$$

$$a(x - y) = km.$$

Because a has no factor in common with m, it must be that $x - y$ is a multiple of m; that is,

$$x \equiv y \pmod{m}.$$

This contradicts the hypothesis.

Now, because ax takes on all m distinct values modulo m, as x ranges over the m values, it must take on the value 1 for some x.

The proof shows that there is only one value (modulo m) of x such that $ax \equiv 1 \pmod{m}$—that is, the multiplicative inverse is unique, apart from additive multiples of m. It also shows that there is a unique (modulo m) integer x such that $ax \equiv b \pmod{m}$, where b is any integer.

As an example, consider the case $m = 16$. Then $\bar{3} = 11$, because $3 \cdot 11 = 33 \equiv 1 \pmod{16}$. We could just as well take $\bar{3} = -5$, because $3 \cdot (-5) = -15 \equiv 1 \pmod{16}$. Similarly $\overline{-3} = 5$, because $(-3) \cdot 5 = -15 \equiv 1 \pmod{16}$.

These observations are important because they show that the concepts apply to both signed and unsigned numbers. If we are working in the domain of unsigned integers on a 4-bit machine, we take $\bar{3} = 11$. In the domain of signed integers, we take $\bar{3} = -5$. But 11 and –5 have the same representation in two's-complement (because they differ by 16), so the same computer word contents can serve in both domains as the multiplicative inverse.

The theorem applies directly to the problem of division (signed and unsigned) by an odd integer d on a W-bit computer. Because any odd integer is relatively prime to 2^W, the theorem says that if d is odd, there exists an integer $\bar{d}$ (unique in the range 0 to $2^W - 1$ or in the range -2^{W-1} to $2^{W-1} - 1$) such that

$$d\bar{d} \equiv 1 \pmod{2^W}.$$

Hence, for any integer n that is a multiple of d,

$$\frac{n}{d} \equiv \frac{n}{d}(d\bar{d}) \equiv n\bar{d} \pmod{2^W}.$$

In other words, n/d can be calculated by multiplying n by $\bar{d}$ and retaining only the rightmost W bits of the product.

If the divisor d is even, let $d = d_0 \cdot 2^k$, where d_0 is odd and $k \geq 1$. Then, simply shift n right k positions (shifting out 0's), and then multiply by $\bar{d}_0$ (the shift could be done after the multiplication as well).

Below is the code for division of n by 7, where n is a multiple of 7. This code gives the correct result whether it is considered to be signed or unsigned division.

```
li    M,0xB6DB6DB7  Mult. inverse, (5*2**32 + 1)/7.
mul   q,M,n         q = n/7.
```

Computing the Multiplicative Inverse by the Euclidean Algorithm

How can we compute the multiplicative inverse? The standard method is by means of the “extended Euclidean algorithm.” This is briefly discussed below as it applies to our problem, and the interested reader is referred to [NZM, p. 13] and to [Knu2, 4.5.2] for a more complete discussion.

Given an odd divisor d, we wish to solve for x

$$dx \equiv 1 (\text{mod}/m),$$

where, in our application, $m = 2^W$ and W is the word size of the machine. This will be accomplished if we can solve for integers x and y (positive, negative, or 0) the equation

$$dx + my = 1.$$

Toward this end, first make d positive by adding a sufficient number of multiples of m to it. (d and $d + km$ have the same multiplicative inverse.) Second, write the following equations (in which $d, m > 0$):

$$\begin{aligned} d(-1) + m(1) &= m - d \qquad \text{(i)} \\ d(1) + m(0) &= d. \qquad\quad \text{(ii)} \end{aligned}$$

If $d = 1$, we are done, because (ii) shows that $x = 1$. Otherwise, compute

$$q = \left\lfloor \frac{m-d}{d} \right\rfloor.$$

Third, multiply Equation (ii) by q and subtract it from (i). This gives

$$d(-1-q) + m(1) = m - d - qd = \text{rem}(m - d, d).$$

This equation holds because we have simply multiplied one equation by a constant and subtracted it from another. If $\text{rem}(m - d, d) = 1$, we are done; this last equation is the solution and $x = -1 - q$.

Repeat this process on the last two equations, obtaining a fourth, and continue until the right-hand side of the equation is 1. The multiplier of d, reduced modulo m, is then the desired inverse of d.

Incidentally, if $m - d < d$, so that the first quotient is 0, then the third row will be a copy of the first, so that the second quotient will be nonzero. Furthermore, most texts start with the first row being

$$d(0) + m(1) = m,$$

but in our application $m = 2^W$ is not representable in the machine.

The process is best illustrated by an example: Let $m = 256$ and $d = 7$. Then the calculation proceeds as follows. To get the third row, note that $q = \quad 249 / 7 \quad = 35$.

```
 7(-1) + 256( 1) = 249
 7( 1) + 256( 0) = 7
7(-36) + 256( 1) = 4
7( 37) + 256(-1) = 3
7(-73) + 256( 2) = 1
```

Thus, the multiplicative inverse of 7, modulo 256, is –73 or, expressed in the range 0 to 255, is 183. Check: $7 \cdot 183 = 1281 \equiv 1 \pmod{256}$.

From the third row on, the integers in the right-hand column are all remainders of dividing the number above it into the number two rows above it, so they form a sequence of strictly decreasing nonnegative integers. Therefore, the sequence must end in 0 (as the above would if carried one more step). Furthermore, the value just before the 0 must be 1, for the following reason. Suppose the sequence ends in b followed by 0, with $b \neq 1$. Then, the integer preceding the b must be a multiple of b, let's say $k_1 b$, for the next remainder to be 0. The integer preceding $k_1 b$ must be of the form $k_1 k_2 b + b$, for the next remainder to be b. Continuing up the sequence, every number must be a multiple of b, including the first two (in the positions of the 249 and the 7 in the above example). This is impossible, because the first two integers are $m - d$ and d, which are relatively prime.

This constitutes an informal proof that the above process terminates, with a value of 1 in the right-hand column, and hence it finds the multiplicative inverse of d.

To carry this out on a computer, first note that if $d < 0$, we should add 2^W to it. With two's-complement arithmetic it is not necessary to actually do anything here; simply interpret d as an unsigned number, regardless of how the application interprets it.

The computation of q must use unsigned division.

Observe that the calculations can be done modulo m, because this does not change the right-hand column (these values are in the range 0 to $m - 1$ anyway). This is important, because it enables the calculations to be done in "single precision," using the computer's modulo-2^W unsigned arithmetic.

Most of the quantities in the table need not be represented. The column of multiples of 256 need not be represented, because in solving $dx + my = 1$, we do not need the value of y. There is no need to represent d in the first column. Reduced to its bare essentials, then, the calculation of the above example is carried out as follows:

255	249
1	7
220	4
37	3
183	1

A C program for performing this computation is shown in Figure 10–5.

```
unsigned mulinv(unsigned d) {           // d must be odd.
   unsigned x1, v1, x2, v2, x3, v3, q;

   x1 = 0xFFFFFFFF;      v1 = -d;
   x2 = 1;               v2 = d;
   while (v2 > 1) {
      q = v1/v2;
      x3 = x1 - q*x2;    v3 = v1 - q*v2;
      x1 = x2;           v1 = v2;
      x2 = x3;           v2 = v3;
   }
   return x2;
}
```

FIGURE 10–5. Multiplicative inverse modulo 2^{32} by the Euclidean algorithm.

The reason the loop continuation condition is `(v2 > 1)`, rather than the more natural `(v2 != 1)`, is that if the latter condition were used, the loop would never terminate if the program were invoked with an even argument. It is best that programs not loop forever even if misused. (If the argument `d` is even, `v2` never takes on the value 1, but it does become 0.)

What does the program compute if given an even argument? As written, it computes a number x such that $dx \equiv 0 \pmod{2^{32}}$, which is probably not useful. However, with the minor modification of changing the loop continuation condition to `(v2 != 0)` and returning `x1` rather than `x2`, it computes a number x such that $dx \equiv g \pmod{2^{32}}$, where g is the greatest common divisor of d and 2^{32}—that is, the greatest power of 2 that divides d. The modified program still computes the multiplicative inverse of d for d odd, but it requires one more iteration than the unmodified program.

As for the number of iterations (divisions) required by the above program, for d odd and less than 20, it requires a maximum of 3 and an average of 1.7. For d in the neighborhood of 1000, it requires a maximum of 11 and an average of about 6.

Computing the Multiplicative Inverse by Newton's Method

It is well known that, over the real numbers, $1 / d$, for $d \neq 0$, can be calculated to ever-increasing accuracy by iteratively evaluating

$$x_{n+1} = x_n(2 - dx_n), \tag{31}$$

provided the initial estimate x_0 is sufficiently close to $1/d$. The number of digits of accuracy approximately doubles with each iteration.

It is not so well known that this same formula can be used to find the multiplicative inverse modulo any power of 2!. For example, to find the multiplicative inverse of 3, modulo 256, start with $x_0 = 1$ (any odd number will do). Then,

$$\begin{aligned}
x_1 &= 1(2 - 3 \cdot 1) = -1, \\
x_2 &= -1(2 - 3(-1)) = -5, \\
x_3 &= -5(2 - 3(-5)) = -85, \\
x_4 &= -85(2 - 3(-85)) = -21845 \equiv -85 \pmod{256}.
\end{aligned}$$

The iteration has reached a fixed point modulo 256, so –85, or 171, is the multiplicative inverse of 3 (modulo 256). All calculations can be done modulo 256.

Why does this work? Because if x_n satisfies

$$dx_n \equiv 1 \pmod{m}$$

and if x_{n+1} is defined by (31), then

$$dx_{n+1} \equiv 1 \pmod{m^2}.$$

To see this, let $dx_n = 1 + km$. Then

$$\begin{aligned} dx_{n+1} &= dx_n(2 - dx_n) \\ &= (1 + km)(2 - (1 + km)) \\ &= (1 + km)(1 - km) \\ &= 1 - k^2m^2 \\ &\equiv 1 \pmod{m^2}. \end{aligned}$$

In our application, m is a power of 2, say 2^N. In this case, if

$$dx_n \equiv 1 \pmod{2^N}, \text{ then}$$

$$dx_{n+1} \equiv 1 \pmod{2^{2N}}.$$

In a sense, if x_n is regarded as a sort of approximation to $\bar{d}$, then each iteration of (31) doubles the number of bits of “accuracy” of the approximation.

It happens that modulo 8, the multiplicative inverse of any (odd) number d is d itself. Thus, taking $x_0 = d$ is a reasonable and simple initial guess at $\bar{d}$. Then, (31) will give values of x_1, x_2, ..., such that

$$\begin{aligned} dx_1 &\equiv 1 \pmod{2^6}, \\ dx_2 &\equiv 1 \pmod{2^{12}}, \\ dx_3 &\equiv 1 \pmod{2^{24}}, \\ dx_4 &\equiv 1 \pmod{2^{48}}, \text{ and so on.} \end{aligned}$$

Thus, four iterations suffice to find the multiplicative inverse modulo 2^{32} (if $x \equiv 1 \pmod{2^{48}}$), then $x \equiv 1 \pmod{2^n}$ for $n \leq 48$). This leads to the C program in Figure 10–6, in which all computations are done modulo 2^{32}.

For about half the values of d, this program takes 4.5 iterations, or nine multiplications. For the other half (those for which the initial value of `xn` is “correct to 4 bits”—that is, $d^2 \equiv 1 \pmod{16}$), it takes seven or fewer, usually seven, multiplications. Thus, it takes about eight multiplications on average.

```
unsigned mulinv(unsigned d) {        // d must be odd.
   unsigned xn, t;

   xn = d;
loop: t = d*xn;
      if (t == 1) return xn;
      xn = xn*(2 - t);
      goto loop;
}
```

IGURE **32**

F 10–6. Multiplicative inverse modulo 2 by Newton's method.

A variation is to simply execute the loop four times, regardless of *d*, perhaps "strung out" to eliminate the loop control (eight multiplications). Another variation is to somehow make the initial estimate x_0 "correct to 4 bits" (that is, find x_0 that satisfies $dx_0 \equiv 1 \pmod{16}$). Then, only three loop iterations are required. Some ways to set the initial estimate are

$$x_0 \leftarrow d + 2((d+1)\ \&\ 4), \quad \text{and}$$

$$x_0 \leftarrow d^2 + d - 1.$$

Here, the multiplication by 2 is a left shift, and the computations are done modulo 2^{32} (ignoring overflow). Because the second formula uses a multiplication, it saves only one.

This concern about execution time is, of course, totally unimportant for the compiler application. For that application, the routine would be so seldom used that it should be coded for minimum space. But there may be applications in which it is desirable to compute the multiplicative inverse quickly.

The "Newton method" described here applies only when (1) the modulus is an integral power of some number *a*, and (2) the multiplicative inverse of *d* modulo *a* is known. It works particularly well for *a* = 2, because then the multiplicative inverse of any (odd) number *d* modulo 2 is known immediately—it is 1.

Sample Multiplicative Inverses

We conclude this section with a listing of some multiplicative inverses in Table 10–3.

TABLE 10–3. SAMPLE MULTIPLICATIVE INVERSES

d	$\bar{d}$		
(dec)	mod 16 (dec)	mod 2^{32} (hex)	mod 2^{64} (hex)
–7	–7	49249249	92492492 49249249
–5	3	33333333	33333333 33333333
–3	5	55555555	55555555 55555555
–1	–1	FFFFFFFF	FFFFFFFF FFFFFFFF
1	1	1	1
3	11	AAAAAAAB	AAAAAAAA AAAAAAAB
5	13	CCCCCCCD	CCCCCCCC CCCCCCCD
7	7	B6DB6DB7	6DB6DB6D B6DB6DB7
9	9	38E38E39	8E38E38E 38E38E39
11	3	BA2E8BA3	2E8BA2E8 BA2E8BA3
13	5	C4EC4EC5	4EC4EC4E C4EC4EC5
15	15	EEEEEEEF	EEEEEEEE EEEEEEEF
25		C28F5C29	8F5C28F5 C28F5C29
125		26E978D5	1CAC0831 26E978D5
625		3AFB7E91	D288CE70 3AFB7E91

You may notice that in several cases (d = 3, 5, 9, 11), the multiplicative inverse of d is the same as the magic number for unsigned division by d (see Section 10–14, "Sample Magic Numbers," on page 238). This is more or less a coincidence. It happens that for these numbers, the magic number M is equal to the multiplier m, and these are of the form $(2^p + 1) / d$, with $p \geq 32$. In this case, notice that

$$Md = \left(\frac{2^p + 1}{d}\right)d \equiv 1 \pmod{2^{32}},$$

so that $M \equiv \bar{d} \pmod{2^{32}}$.

10–17 Test for Zero Remainder after Division by a Constant

The multiplicative inverse of a divisor d can be used to test for a zero remainder after division by d[GM].

Unsigned

First, consider unsigned division with the divisor d odd. Denote by $\bar{d}$ the multiplicative

$$d\bar{d} \equiv 1 \pmod{2^W}.$$

inverse of d. Then, because , where W is the machine's word size in bits, $\bar{d}$ is also odd. Thus, $\bar{d}$ is relatively prime to 2^W, and as shown in the proof of theorem MI in the preceding section, as n ranges over all 2^W distinct values modulo 2^W, $n\bar{d}$ takes on all 2^W distinct values modulo 2^W.

It was shown in the preceding section that if n is a multiple of d,

$$\frac{n}{d} \equiv n\bar{d} \pmod{2^W}.$$

That is, for $n = 0, d, 2d, \ldots, (2^W - 1) / d \quad d$, $n\bar{d} \equiv 0, 1, 2, \ldots, (2^W - 1) / d \pmod{2^W}$. Therefore, for n *not* a multiple of d, the value of $n\bar{d}$, reduced modulo 2^W to the range 0 to $2^W - 1$, must exceed $(2^W - 1) / d$.

This can be used to test for a zero remainder. For example, to test if an integer n is a multiple of 25, multiply n by $\overline{25}$ and compare the rightmost W bits to $(2^W - 1) / 25$. On our basic RISC:

```
li       M,0xC28F5C29  Load mult. inverse of 25.
mul     q,M,n          q = right half of M*n.
li       c,0x0A3D70A3  c = floor((2**32-1)/25).
cmpleu t,q,c          Compare q and c, and branch
bt      t,is_mult      if n is a multiple of 25.
```

To extend this to even divisors, let $d = d_o \cdot 2^k$, where d_o is odd and $k \ge 1$. Then, because an integer is divisible by d if and only if it is divisible by d_o and by 2^k, and because n and $n\bar{d}_o$ have the same number of trailing zeros ($\bar{d}_o$ is odd), the test that n is a multiple of d is

$$\text{Set } q = \text{mod}(n\bar{d}_o, 2^W);$$
$$q \le \lfloor (2^W - 1)/d_o \rfloor \text{ and } q \text{ ends in } k \text{ or more 0-bits,}$$

where the mod function is understood to reduce $n\bar{d}$ to the interval $[0, 2^W - 1]$.

Direct implementation of this requires two tests and conditional branches, but it can be reduced to one *compare-branch* quite efficiently if the machine has the *rotate-shift* instruction. This follows from the following theorem, in which $a \overset{rot}{\gg} k$ denotes the computer word a rotated right k positions $(0 \le k \le 32)$.

THEOREM ZRU. *$x \overset{u}{\le} a$ and x ends in k 0-bits if and only if $x \overset{rot}{\gg} k \overset{u}{\le} \lfloor a/2^k \rfloor$*

Proof. (Assume a 32-bit machine.) Suppose $x \overset{u}{\le} a$and x ends in k 0-bits. Then, because $x \overset{u}{\le} a$, $\lfloor x/2^k \rfloor \overset{u}{\le} \lfloor a/2^k \rfloor$ But $\lfloor x/2^k \rfloor = x \overset{rot}{\gg} k$ Therefore, $x \overset{rot}{\gg} k \overset{u}{\le} \lfloor a/2^k \rfloor$ If x does not end in k 0-bits, then $x \overset{rot}{\gg} k$ does not begin with k 0-bits, whereas $a / 2^k$ does, so $x \overset{rot}{\gg} k \overset{u}{\le} \lfloor a/2^k \rfloor$ Lastly, if $x \overset{u}{>} a$ and x ends in k 0-bits, then the integer formed from the first $32 - k$ bits of x must exceed that formed from the first $32 - k$ bits of a, so that $\lfloor x/2^k \rfloor \overset{u}{>} \lfloor a/2^k \rfloor$

Using this theorem, the test that n is a multiple of d, where n and $d > 1$ are unsigned integers and $d = d_o \cdot 2^k$ with d_o odd, is

$$\boxed{\begin{array}{l} q \leftarrow \text{mod}(n\bar{d}_o, 2^W); \\ q \overset{rot}{\gg} k \overset{u}{\leq} \lfloor (2^W - 1)/d \rfloor. \end{array}}$$

Here we used $(2^W - 1) / d_o \quad / 2^k \quad = \quad (2^W - 1) / (d_o \cdot 2^k) \quad = \quad (2^W - 1) / d \quad$.

As an example, the following code tests an unsigned integer n to see if it is a multiple of 100:

```
li     M,0xC28F5C29  Load mult. inverse of 25.
mul    q,M,n          q = right half of M*n.
shrri  q,q,2          Rotate right two positions.
li     c,0x028F5C28  c = floor((2**32-1)/100).
cmpleu t,q,c          Compare q and c, and branch
bt     t,is_mult      if n is a multiple of 100.
```

Signed, Divisor ≥ 2

For signed division, it was shown in the preceding section that if n is a multiple of d and d is odd, then

$$\frac{n}{d} \equiv n\bar{d} \pmod{2^W}.$$

Thus, for $n = \quad -2^{W-1}/d \quad \cdot d, \ldots, -d, 0, d, \ldots, \quad (2^{W-1} - 1) / d \quad \cdot d$, we have $n\bar{d} \equiv \quad -2^{W-1} / d, \ldots, -1, 0, 1, \ldots, \quad (2^{W-1} - 1) / d \quad \pmod{2^W}$. Furthermore, because d *is* relatively prime to 2^W, as n ranges over all 2^W distinct values modulo 2^W, $n\bar{d}$ takes on all 2^W distinct values modulo 2^W. Therefore, n is a multiple of d if and only if

$$\lceil -2^{W-1}/d \rceil \leq \text{mod}(n\bar{d}, 2^W) \leq \lfloor (2^{W-1} - 1)/d \rfloor,$$

where the mod function is understood to reduce $n\bar{d}$ to the interval $[-2^{W-1}\ 2^{W-1} - 1]$

This can be simplified a little by observing that because d is odd and, as we are assuming, positive and not equal to 1, it does not divide $2^W - 1$. Therefore,

$$-2^{W-1} / d \quad = \quad (-2^{W-1} + 1)\ d \quad = - \quad (2^{W-1} - 1)/ d \quad .$$

Thus, for signed numbers, the test that n is a multiple of d, where $d = d_o \cdot 2^k$ and d_o is odd, is

Set $q = \text{mod}(n\bar{d}_o, 2^W);$

$- \quad (2^{W-1} - 1) / d_o \quad \leq q \leq \quad (2^{W-1} - 1)/d_o \quad$ and q ends in k or more 0-bits.

On the surface, this would seem to require three tests and branches. However, as in the unsigned case, it can be reduced to one *compare-branch* by use of the following theorem:

THEOREM ZRS. *If a ≥ 0, the following assertions are equivalent:*

$$(1)\ \ -a \le x \le a \text{ and } x \text{ ends in } k \text{ or more 0-bits,}$$

$$(2)\ \ \text{abs}(x) \overset{rot}{\gg} k \overset{u}{\le} \lfloor a/2^k \rfloor, \text{ and}$$

$$(3)\ \ x + a' \overset{rot}{\gg} k \overset{u}{\le} \lfloor 2a'/2^k \rfloor,$$

where a' is a with its rightmost k bits set to 0 (that is, $a' = a \mathbin{\&} -2^k$).

Proof. (Assume a 32-bit machine). To see that (1) is equivalent to (2), clearly the assertion $-a \le x \le a$ is equivalent to abs(x) ≤ a. Then, Theorem ZRU applies, because both sides of this inequality are nonnegative.

To see that (1) is equivalent to (3), note that assertion (1) is equivalent to itself with a replaced with a'. Then, by the theorem on bounds checking on page 68, this in turn is equivalent to

$$x + a' \overset{u}{\le} 2a'.$$

Because $x + a'$ ends in k 0-bits if and only if x does, Theorem ZRU applies, giving the result.

Using part (3) of this theorem, the test that n is a multiple of d, where n and $d \ge 2$ are signed integers and $d = d_o \cdot 2^k$ with d_o odd, is

$$\boxed{\begin{array}{l} q \leftarrow \text{mod}(n\bar{d}_o, 2^W); \\ a' \leftarrow \lfloor (2^{W-1} - 1)/d_o \rfloor \mathbin{\&} -2^k; \\ q + a' \overset{rot}{\gg} k \overset{u}{\le} \lfloor (2a')/2^k \rfloor. \end{array}}$$

(a' can be computed at compile time, because d is a constant.)

As an example, the following code tests a signed integer n to see if it is a multiple of 100. Notice that the constant $2a'/2^k$ can always be derived from the constant a' by a shift of $k - 1$ bits, saving an instruction or a load from memory to develop the comparand.

```
li      M,0xC28F5C29  Load mult. inverse of 25.
mul     q,M,n         q = right half of M*n.
li      c,0x051EB850  c = floor((2**31 – 1)/25) & –4.
add     q,q,c         Add c.
shrri   q,q,2         Rotate right two positions.
shri    c,c,1         Compute const. for comparison.
cmpleu  t,q,c         Compare q and c, and
bt      t,is_mult     branch if n is a mult. of 100.
```

10–18 Methods Not Using Multiply High

In this section we consider some methods for dividing by constants that do not use the *multiply high* instruction, or a multiplication instruction that gives a double-word result.

We show how to change division by a constant into a sequence of *shift* and *add* instructions, or *shift, add*, and *multiply* for more compact code.

Unsigned Division

For these methods, unsigned division is simpler than signed division, so we deal with unsigned division first. One method is to use the techniques given that use the *multiply high* instruction, but use the code shown in Figure 8–2 on page 174 to do the *multiply high* operation. Figure 10–7 shows how this works out for the case of (unsigned) division by 3. This is a combination of the code on page 228 and Figure 8–2 with "`int`" changed to "`unsigned`." The code is 15 instructions, including four multiplications. The multiplications are by large constants and would take quite a few instructions if converted to *shift*'s and *add*'s. Very similar code can be devised for the signed case. This method is not particularly good and won't be discussed further.

Another method [GLS1] is to compute in advance the reciprocal of the divisor, and multiply the dividend by that with a series of *shift right* and *add* instructions. This gives an approximation to the quotient. It is merely an approximation, because the reciprocal of the divisor (which we assume is not an exact power of two) is not expressed exactly in 32 bits, and also because each *shift right* discards bits of the dividend. Next, the remainder with respect to the approximate quotient is computed, and that is divided by the divisor to form a correction, which is added to the approximate quotient, giving the exact quotient. The remainder is generally small compared to the divisor (a few multiples thereof), so there is often a simple way to compute the correction without using a *divide* instruction.

To illustrate this method, consider dividing by 3, that is, computing $\lfloor n / 3 \rfloor$ where $0 \le n < 2^{32}$. The reciprocal of 3, in binary, is approximately

0.0101 0101 0101 0101 0101 0101 0101 0101.

To compute the approximate product of that and n, we could use

$$q \leftarrow (n \overset{u}{\gg} 2) + (n \overset{u}{\gg} 4) + (n \overset{u}{\gg} 6) + \ldots + (n \overset{u}{\gg} 30) \qquad (32)$$

```
unsigned divu3(unsigned n) {
   unsigned n0, n1, w0, w1, w2, t, q;

   n0 = n & 0xFFFF;
   n1 = n >> 16;
   w0 = n0*0xAAAB;
   t = n1*0xAAAB + (w0 >> 16);
   w1 = t & 0xFFFF;
   w2 = t >> 16;
   w1 = n0*0xAAAA + w1;
   q = n1*0xAAAA + w2 + (w1 >> 16);
   return q >> 1;
}
```

FIGURE 10–7. Unsigned divide by 3 using simulated ***multiply high unsigned*****.**

(29 instructions; the last 1 in the reciprocal is ignored because it would add the term $n \overset{u}{\gg} 32$, which is obviously 0). However, the simple repeating pattern of 1's and 0's in the reciprocal permits a method that is both faster (nine instructions) and more accurate:

$$\begin{aligned} &q \leftarrow (n \overset{u}{\gg} 2) + (n \overset{u}{\gg} 4) \\ &q \leftarrow q + (q \overset{u}{\gg} 4) \\ &q \leftarrow q + (q \overset{u}{\gg} 8) \\ &q \leftarrow q + (q \overset{u}{\gg} 16) \end{aligned} \qquad (1)$$

To compare these methods for their accuracy, consider the bits that are shifted out by each term of (32), if *n* is all 1-bits. The first term shifts out two 1-bits, the next four 1-bits, and so on. Each of these contributes an error of almost 1 in the least significant bit. Since there are 16 terms (counting the term we ignored), the shifts contribute an error of almost 16. There is an additional error due to the fact that the reciprocal is truncated to 32 bits; it turns out that the maximum total error is 16.

For procedure (1), each right shift also contributes an error of almost 1 in the least significant bit. But there are only five shift operations. They contribute an error of almost 5, and there is a further error due to the fact that the reciprocal is truncated to 32 bits; it turns out that the maximum total error is 5.

After computing the estimated quotient *q*, the remainder *r* is computed from

$$r \leftarrow n - q * 3.$$

The remainder cannot be negative, because *q* is never larger than the exact quotient. We need to know how large *r* can be to devise the simplest possible method for computing $r \overset{u}{\div} 3$. In general, for a divisor *d* and an estimated quotient *q* too low by *k*, the remainder will range from $k*d$ to $k*d + d - 1$. (The upper limit is conservative; it may not actually be attained.) Thus, using (1), for which *q* is too low by at most 5, we expect the remainder to be at most 5*3 + 2 = 17. Experimentation reveals that it is actually at most 15. Thus, for the correction we must compute (exactly)

$$r \overset{u}{\div} 3, \quad \text{for } 0 \le r \le 15.$$

Since *r* is small compared to the largest value that a register can hold, this can be approximated by multiplying *r* by some approximation to 1/3 of the form *a*/*b* where *b* is a power of 2. This is easy to compute, because the division is simply a shift. The value of *a*/ *b* must be slightly larger than 1/3, so that after shifting the result will agree with truncated division. A sequence of such approximations is:

1/2, 2/4, 3/8, 6/16, 11/32, 22/64, 43/128, 86/256, 171/512, 342/1024, ...

Usually, the smaller fractions in the sequence are easier to compute, so we choose the smallest one that works; in the case at hand this is 11/32. Therefore, the final, exact, quotient is given by

$$q \leftarrow q + (11 * r \overset{u}{\gg} 5).$$

The solution involves two multiplications by small numbers (3 and 11); these can be changed to *shift*'s and *add*'s.

Figure 10–8 shows the entire solution in C. As shown, it consists of 14 instructions, including two multiplications. If the multiplications are changed to *shift*'s and *add*'s, it amounts to 18 elementary instructions. However, if it is desired to avoid the multiplications, then either alternative `return` statement shown gives a solution in 17 elementary instructions. Alternative 2 has just a little instruction-level parallelism, but in truth this method generally has very little of that.

A more accurate estimate of the quotient can be obtained by changing the first executable line to

```
q = (n >> 1) + (n >> 3);
```

(which makes `q` too large by a factor of 2, but it has one more bit of accuracy), and then inserting just before the assignment to `r`,

```
q = q >> 1;
```

With this variation, the remainder is at most 9. However, there does not seem to be any better code for calculating $r \overset{u}{\div} 3$, with *r* limited to 9 than there is for *r* limited to 15 (four elementary instructions in either case). Thus, using the idea would cost one instruction. This possibility is mentioned because it *does* give a code improvement for most divisors.

```
unsigned divu3(unsigned n) {
   unsigned q, r;

   q = (n >> 2) + (n >> 4);     // q = n*0.0101 (approx).
   q = q + (q >> 4);            // q = n*0.01010101.
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*3;                 // 0 <= r <= 15.
   return q + (11*r >> 5);      // Returning q + r/3.
// return q + (5*(r + 1) >> 4);        // Alternative 1.
// return q + ((r + 5 + (r << 2)) >> 4);// Alternative 2.
}
```

FIGURE 10–8. Unsigned divide by 3.

Figure 10–9 shows two variations of this method for dividing by 5. The reciprocal of 5, in binary, is

0.0011 0011 0011 0011 0011 0011 0011 0011.

As in the case of division by 3, the simple repeating pattern of 1's and 0's allows a fairly efficient and accurate computing of the quotient estimate. The estimate of the quotient computed by the code on the left can be off by at most 5, and it turns out that the remainder is at most 25. The code on the right retains two additional bits of accuracy in computing the quotient estimate, which is off by at most 2. The remainder in this case is at most 10. The smaller maximum remainder permits approximating 1/5 by 7/32 rather than 13/64, which gives a slightly more efficient program if the multiplications are done by *shift*'s and *add*'s. The instruction counts are, for the code on the left: 14 instructions including two multiplications, or 18 elementary instructions; for the code on the right: 15 instructions including two multiplications, or 17 elementary instructions. The alternative code in the `return` statement is useful only if your machine has comparison predicate instructions. It doesn't reduce the instruction count, but merely has a little instruction-level parallelism.

For division by 6, the divide-by-3 code can be used, followed by a *shift right* of 1. However, the extra instruction can be saved by doing the computation directly, using the binary approximation

$$4/6 \approx 0.1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010.$$

```
unsigned divu5a(unsigned n) {
   unsigned q, r;

   q = (n >> 3) + (n >> 4);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*5;
   return q + (13*r >> 6);
}
```

```
unsigned divu5b(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*5;
   return q + (7*r >> 5);
// return q + (r>4) + (r>9);
}
```

FIGURE 10–9. Unsigned divide by 5.

The code is shown in Figure 10–10. The version on the left multiplies by an approximation to 1/6 and then corrects with a multiplication by 11/64. The version on the right takes advantage of the fact that by multiplying by an approximation to 4/6, the quotient estimate is off by only 1 at most. This permits simpler code for the correction; it simply adds 1 to `q` if `r` ≥ 6. The code in the second `return` statement is appropriate if the machine has the comparison predicate instructions. Function `divu6b` is 15 instructions, including one *multiply*, as shown, or 17 elementary instructions if the multiplication by 6 is changed to *shift*'s and *add*'s.

```
unsigned divu6a(unsigned n) {
   unsigned q, r;

   q = (n >> 3) + (n >> 5);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*6;
   return q + (11*r >> 6);
}
```

```
unsigned divu6b(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*6;
   return q + ((r + 2) >> 3);
// return q + (r > 5);
}
```

FIGURE 10–10. Unsigned divide by 6.

For larger divisors, usually it seems to be best to use an approximation to 1/ *d* that is shifted left so that its most significant bit is 1. It seems that the quotient is then off by at most 1 usually (possibly always, this writer does not know), which permits efficient code for the correction step. Figure 10–11 shows code for dividing by 7 and 9, using the binary approximations

$$4/7 \approx 0.1001\ 0010\ 0100\ 1001\ 0010\ 0100\ 1001\ 0010, \quad \text{and}$$

$$8/9 \approx 0.1110\ 0011\ 1000\ 1110\ 0011\ 1000\ 1110\ 0011.$$

If the multiplications by 7 and 9 are expanded into *shift*'s and *add*'s, these functions take 16 and 15 elementary instructions, respectively.

```
unsigned divu7(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 4);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 2;
   r = n - q*7;
   return q + ((r + 1) >> 3);
// return q + (r > 6);
}
```

```
unsigned divu9(unsigned n) {
   unsigned q, r;

   q = n - (n >> 3);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*9;
   return q + ((r + 7) >> 4);
// return q + (r > 8);
}
```

FIGURE 10–11. Unsigned divide by 7 and 9.

Figures 10–12 and 10–13 show code for dividing by 10, 11, 12, and 13. These are based on the binary approximations:

$$8/10 \approx 0.1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100,$$

$$8/11 \approx 0.1011\ 1010\ 0010\ 1110\ 1000\ 1011\ 1010\ 0010,$$

$$8/12 \approx 0.1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010, \quad \text{and}$$

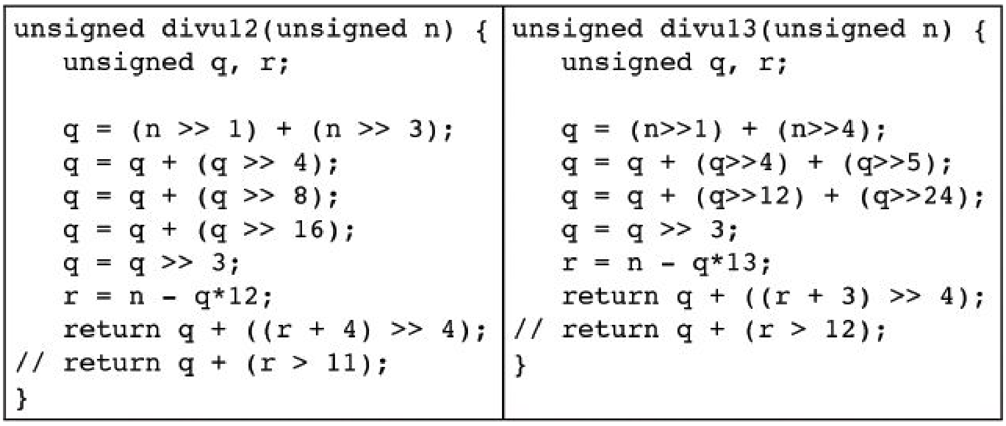
$$8/13 \approx 0.1001\ 1101\ 1000\ 1001\ 1101\ 1000\ 1001\ 1101.$$

If the multiplications are expanded into *shift*'s and *add*'s, these functions take 17, 20, 17, and 20 elementary instructions, respectively.

```
unsigned divu10(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*10;
   return q + ((r + 6) >> 4);
// return q + (r > 9);
}
```

```
unsigned divu11(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 2) -
       (n >> 5) + (n >> 7);
   q = q + (q >> 10);
   q = q + (q >> 20);
   q = q >> 3;
   r = n - q*11;
   return q + ((r + 5) >> 4);
// return q + (r > 10);
```

FIGURE 10–12. Unsigned divide by 10 and 11.



```
unsigned divu12(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*12;
   return q + ((r + 4) >> 4);
// return q + (r > 11);
}
```

```
unsigned divu13(unsigned n) {
   unsigned q, r;

   q = (n>>1) + (n>>4);
   q = q + (q>>4) + (q>>5);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*13;
   return q + ((r + 3) >> 4);
// return q + (r > 12);
}
```

FIGURE 10–13. Unsigned divide by 12 and 13.

The case of dividing by 13 is instructive because it shows how you must look for repeating strings in the binary expansion of the reciprocal of the divisor. The first assignment sets `q` equal to `n`*0.1001. The second assignment to `q` adds `n`*0.00001001 and `n`*0.000001001. At this point, `q` is (approximately) equal to `n`*0.100111011. The third assignment to `q` adds in repetitions of this pattern. It sometimes helps to use subtraction, as in the case of `divu9` above. However, you must use care with subtraction, because it may cause the quotient estimate to be too large, in which case the remainder is negative and the method breaks down. It is quite complicated to get optimal code, and we don’t have a general cookbook method that you can put in a compiler to handle any divisor.

The examples above are able to economize on instructions, because the reciprocals have simple repeating patterns, and because the multiplication in the computation of the remainder `r` is by a small constant, which can be done with only a few *shift*’s and *add*’s. One might wonder how successful this method is for larger divisors. To roughly assess this, Figures 10–14 and 10–15 show code for dividing by 100 and by 1000 (decimal). The relevant reciprocals are

$$64/100 \approx 0.1010\ 0011\ 1101\ 0111\ 0000\ 1010\ 0011\ 1101 \quad \text{and}$$

$$512/1000 \approx 0.1000\ 0011\ 0001\ 0010\ 0110\ 1110\ 1001\ 0111.$$

If the multiplications are expanded into *shift*’s and *add*’s, these functions take 25 and 23 elementary instructions, respectively.

```
unsigned divu100(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 3) + (n >> 6) - (n >> 10) +
       (n >> 12) + (n >> 13) - (n >> 16);
   q = q + (q >> 20);
   q = q >> 6;
   r = n - q*100;
   return q + ((r + 28) >> 7);
// return q + (r > 99);
}
```

FIGURE 10–14. Unsigned divide by 100.

```
unsigned divu1000(unsigned n) {
   unsigned q, r, t;

   t = (n >> 7) + (n >> 8) + (n >> 12);
   q = (n >> 1) + t + (n >> 15) + (t >> 11) + (t >> 14);
   q = q >> 9;
   r = n - q*1000;
   return q + ((r + 24) >> 10);
// return q + (r > 999);
}
```

FIGURE 10–15. Unsigned divide by 1000.

In the case of dividing by 1000, the least significant eight bits of the reciprocal estimate are nearly ignored. The code of Figure 10–15 replaces the binary 1001 0111 with 0100 0000, and still the quotient estimate is within one of the true quotient. Thus, it appears that although large divisors might have very little repetition in the binary representation of the reciprocal estimate, at least some bits can be ignored, which helps hold down the number of *shift*'s and *add*'s required to compute the quotient estimate.

This section has shown, in a somewhat imprecise way, how unsigned division by a constant can be reduced to a sequence of, typically, about 20 elementary instructions. It is nontrivial to get an algorithm that generates these code sequences that is suitable for incorporation into a compiler, because of three difficulties in getting optimal code.

1. It is necessary to search the reciprocal estimate bit string for repeating patterns.
2. Negative terms (as in `divu10` and `divu100`) can be used sometimes, but the error analysis required to determine just when they can be used is difficult.
3. Sometimes some of the least significant bits of the reciprocal estimate can be ignored (how many?).

Another difficulty for some target machines is that there are many variations on the code examples given that have more instructions, but that would execute faster on a machine with multiple shift and add units.

The code of Figures 10–7 through 10–15 has been tested for all 2^{32} values of the dividends.

Signed Division

The methods given above can be made to apply to signed division. The *right shift* instructions in computing the quotient estimate become *signed right shift* instructions, which compute floor division by powers of 2. Thus, the quotient estimate is too low (algebraically), so the remainder is nonnegative, as in the unsigned case.

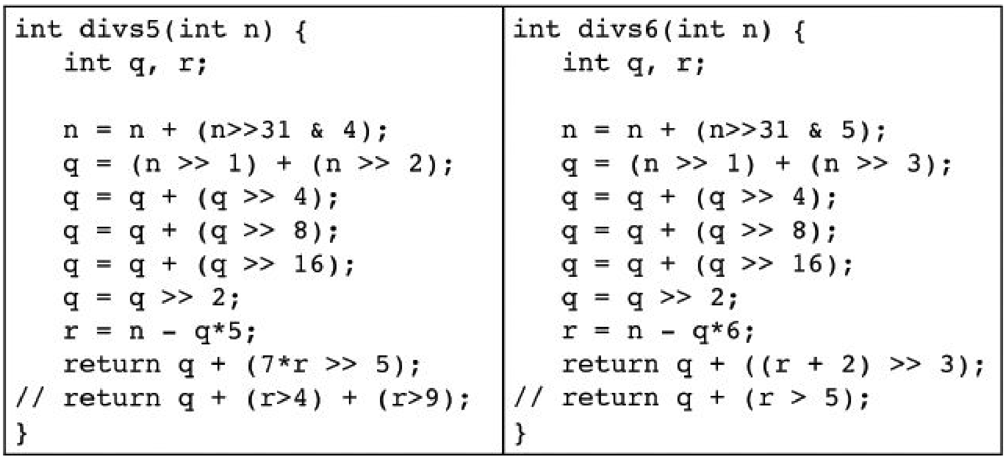
The code most naturally computes the floor division result, so we need a correction to make it compute the conventional truncated-toward-0 result. This can be done with three computational instructions by adding $d - 1$ to the dividend if the dividend is negative. For example, if the divisor is 6, the code begins with (the *shift* here is a *signed shift*)

```
n = n + (n >> 31 & 5);
```

Other than this, the code is very similar to that of the unsigned case. The number of elementary operations required is usually three more than in the corresponding unsigned division function. Several examples are given in Figures 10–16 through 10–22. All have been exhaustively tested.

```
int divs3(int n) {
   int q, r;

   n = n + (n>>31 & 2);          // Add 2 if n < 0.
   q = (n >> 2) + (n >> 4);      // q = n*0.0101 (approx).
   q = q + (q >> 4);             // q = n*0.01010101.
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*3;                  // 0 <= r <= 14.
   return q + (11*r >> 5);       // Returning q + r/3.
// return q + (5*(r + 1) >> 4);          // Alternative 1.
// return q + ((r + 5 + (r << 2)) >> 4);// Alternative 2.
}
```

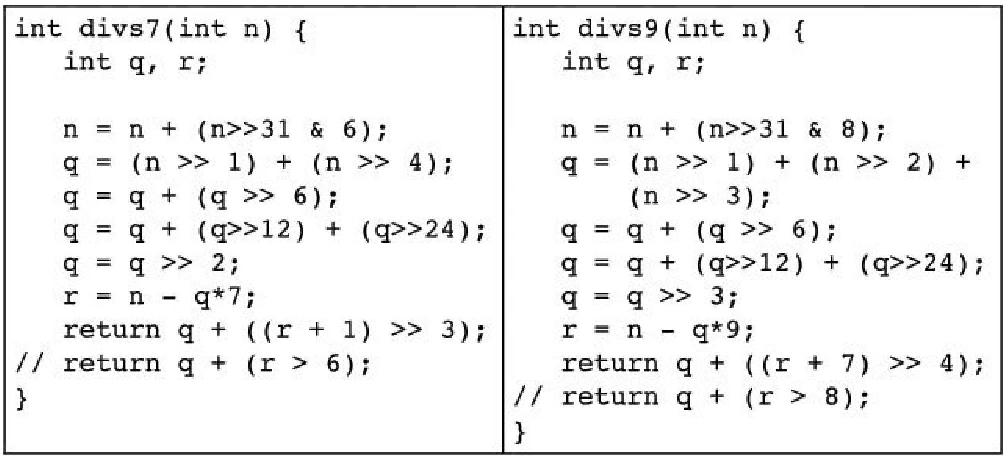
FIGURE 10–16. Signed divide by 3.



```
int divs5(int n) {
   int q, r;

   n = n + (n>>31 & 4);
   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*5;
   return q + (7*r >> 5);
// return q + (r>4) + (r>9);
}
```

```
int divs6(int n) {
   int q, r;

   n = n + (n>>31 & 5);
   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*6;
   return q + ((r + 2) >> 3);
// return q + (r > 5);
}
```

FIGURE 10–17. Signed divide by 5 and 6.



```
int divs7(int n) {
   int q, r;

   n = n + (n>>31 & 6);
   q = (n >> 1) + (n >> 4);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 2;
   r = n - q*7;
   return q + ((r + 1) >> 3);
// return q + (r > 6);
}
```

```
int divs9(int n) {
   int q, r;

   n = n + (n>>31 & 8);
   q = (n >> 1) + (n >> 2) +
       (n >> 3);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*9;
   return q + ((r + 7) >> 4);
// return q + (r > 8);
}
```

FIGURE 10–18. Signed divide by 7 and 9.

```
int divs10(int n) {
   int q, r;

   n = n + (n>>31 & 9);
   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*10;
   return q + ((r + 6) >> 4);
// return q + (r > 9);
}
```

```
int divs11(int n) {
   int q, r;

   n = n + (n>>31 & 10);
   q = (n >> 1) + (n >> 2) -
       (n >> 5) + (n >> 7);
   q = q + (q >> 10);
   q = q + (q >> 20);
   q = q >> 3;
   r = n - q*11;
   return q + ((r + 5) >> 4);
// return q + (r > 10);
}
```

FIGURE 10–19. Signed divide by 10 and 11.

```
int divs12(int n) {
   int q, r;

   n = n + (n>>31 & 11);
   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*12;
   return q + ((r + 4) >> 4);
// return q + (r > 11);
}
```

```
int divs13(int n) {
   int q, r;

   n = n + (n>>31 & 12);
   q = (n>>1) + (n>>4);
   q = q + (q>>4) + (q>>5);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*13;
   return q + ((r + 3) >> 4);
// return q + (r > 12);
}
```

FIGURE 10–20. Signed divide by 12 and 13.

```
int divs100(int n) {
   int q, r;

   n = n + (n>>31 & 99);
   q = (n >> 1) + (n >> 3) + (n >> 6) - (n >> 10) +
       (n >> 12) + (n >> 13) - (n >> 16);
   q = q + (q >> 20);
   q = q >> 6;
   r = n - q*100;
   return q + ((r + 28) >> 7);
// return q + (r > 99);
}
```

FIGURE 10–21. Signed divide by 100.

```
int divs1000(int n) {
   int q, r, t;

   n = n + (n >> 31 & 999);
   t = (n >> 7) + (n >> 8) + (n >> 12);
   q = (n >> 1) + t + (n >> 15) + (t >> 11) + (t >> 14) +
       (n >> 26) + (t >> 21);
   q = q >> 9;
   r = n - q*1000;
   return q + ((r + 24) >> 10);
// return q + (r > 999);
}
```

FIGURE 10-22. Signed divide by 1000.

10-19 Remainder by Summing Digits

This section addresses the problem of computing the remainder of division by a constant without computing the quotient. The methods of this section apply only to divisors of the form $2^k \pm 1$, for k an integer greater than or equal to 2, and in most cases the code resorts to a table lookup (an indexed *load* instruction) after a fairly short calculation.

We will make frequent use of the following elementary property of congruences:

THEOREM C. *If* $a \equiv b$ (mod m) and $c \equiv d$ (mod m), *then*

$$a + c \equiv b + d \pmod{m} \quad \text{and}$$
$$ac \equiv bd \pmod{m}.$$

The unsigned case is simpler and is dealt with first.

Unsigned Remainder

For a divisor of 3, multiplying the trivial congruence $1 \equiv 1 \pmod 3$ repeatedly by the congruence $2 \equiv -1 \pmod 3$, we conclude by Theorem C that

$$2^k \equiv \begin{cases} 1 \pmod 3, & k \text{ even,} \\ -1 \pmod 3, & k \text{ odd.} \end{cases}$$

Therefore, a number n written in binary as $\dots b_3\, b_2\, b_1\, b_0$ satisfies

$$n = \dots + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0 \equiv \dots - b_3 + b_2 - b_1 + b_0 \pmod 3,$$

which is derived by using Theorem C repeatedly. Thus, we can alternately add and subtract the bits in the binary representation of the number to obtain a smaller number that has the same remainder upon division by 3. If the sum is negative, you must add a multiple of 3 to make it nonnegative. The process can then be repeated until the result is in the range 0 to 2.

The same trick works for finding the remainder after dividing a decimal number by 11.

Thus, if the machine has the *population count* instruction, a function that computes the remainder modulo 3 of an unsigned number `n` might begin with

```
n = pop(n & 0x55555555) - pop(n & 0xAAAAAAAA);
```

This can be simplified by using the following surprising identity discovered by Paolo Bonzini [Bonz]:

$$\text{pop}(x \,\&\, \bar{m}) - \text{pop}(x \,\&\, m) = \text{pop}(x \oplus m) - \text{pop}(m). \qquad (2)$$

Proof:

$$\begin{aligned}
&\text{pop}(x \,\&\, \bar{m}) - \text{pop}(x \,\&\, m) \\
&= \text{pop}(x \,\&\, \bar{m}) - (32 - \text{pop}(\overline{x \,\&\, m})) && \text{pop}(a) = 32 - \text{pop}(\bar{a}). \\
&= \text{pop}(x \,\&\, \bar{m}) + \text{pop}(\bar{x} \mid \bar{m}) - 32 && \text{DeMorgan.} \\
&= \text{pop}(x \,\&\, \bar{m}) + \text{pop}(\bar{x} \,\&\, m) + \text{pop}(\bar{m}) - 32 && \text{pop}(a \mid b) = \\
& && \text{pop}(a \,\&\, \bar{b}) + \text{pop}(b). \\
&= \text{pop}((x \,\&\, \bar{m}) \mid (\bar{x} \,\&\, m)) + \text{pop}(\bar{m}) - 32 && \text{Disjoint.} \\
&= \text{pop}(x \oplus m) - \text{pop}(m)
\end{aligned}$$

Since the references to 32 (the word size) cancel out, the result holds for any word size. Another way to prove (2) is to observe that it holds for $x = \mathbf{0}$, and if a 0-bit in x is changed to a 1 where m is 1, then both sides of (2) decrease by 1, and if a 0-bit of x is changed to a 1 where m is 0, then both sides of (2) increase by 1.

Applying (2) to the line of C code above gives

```
n = pop(n ^ 0xAAAAAAAA) - 16;
```

We want to apply this transformation again, until `n` is in the range 0 to 2, if possible. It is best to avoid producing a negative value of `n`, because the sign bit would not be treated properly on the next round. A negative value can be avoided by adding a sufficiently large multiple of 3 to `n`. Bonzini's code, shown in Figure 10–23, increases the constant by 39. This is larger than necessary to make `n` nonnegative, but it causes `n` to range from –3 to 2 (rather than –3 to 3) after the second round of reduction. This simplifies the code on the `return` statement, which is adding 3 if `n` is negative. The function executes in 11 instructions, counting two to load the large constant.

Figure 10–24 shows a variation that executes in four instructions, plus a simple table lookup operation (e.g., an indexed *load byte* instruction).

```
int remu3(unsigned n) {
   n = pop(n ^ 0xAAAAAAAA) + 23;       // Now 23 <= n <= 55.
   n = pop(n ^ 0x2A) - 3;              // Now -3 <= n <= 2.
   return n + (((int)n >> 31) & 3);
}
```

IGURE

F   10–23. Unsigned remainder modulo 3, using *population count*.

```
int remu3(unsigned n) {

   static char table[33] = {2, 0,1,2, 0,1,2, 0,1,2,
          0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
          0,1,2, 0,1};

   n = pop(n ^ 0xAAAAAAAA);
   return table[n];
}
```

FIGURE 10–24. Unsigned remainder modulo 3, using *population count* and a table lookup.

To avoid the *population count* instruction, notice that because 4 ≡ 1 (mod 3), 4^k ≡ 1 (mod 3). A binary number can be viewed as a base 4 number by taking its bits in pairs and interpreting the bits 00 to 11 as a base 4 digit ranging from 0 to 3. The pairs of bits can be summed using the code of Figure 5–2 on page 82, omitting the first executable line (overflow does not occur in the additions). The final sum ranges from 0 to 48, and a table lookup can be used to reduce this to the range 0 to 2. The resulting function is 16 elementary instructions, plus an indexed *load*.

There is a similar, but slightly better, way. As a first step, `n` can be reduced to a smaller number that is in the same congruence class modulo 3 with

```
   n = (n >> 16) + (n & 0xFFFF);
```

This splits the number into two 16-bit portions, which are added together. The contribution modulo 3 of the left 16 bits of `n` is not altered by shifting them right 16 positions, because the shifted number, multiplied by 2^{16}, is the original number, and 2^{16} ≡ 1 (mod 3). More generally, 2^k ≡ 1 (mod 3) if *k* is even. This is used repeatedly (five times) in the code shown in Figure 10–25. This code is 19 instructions. The instruction count can be reduced by cutting off the digit summing earlier and using an in-memory table lookup, as illustrated in Figure 10–26 (nine instructions, plus an indexed *load*). The instruction count can be reduced to six (plus an indexed *load*) by using a table of size 0x2FE = 766 bytes.

To compute the unsigned remainder modulo 5, the code of Figure 10–27 uses the relations 16^k ≡ 1 (mod 5) and 4 ≡ –1 (mod 5). It is 21 elementary instructions, assuming the multiplication by 3 is expanded into a *shift* and *add*.

```
int remu3(unsigned n) {
   n = (n >> 16) + (n & 0xFFFF);        // Max 0x1FFFE.
   n = (n >>  8) + (n & 0x00FF);        // Max 0x2FD.
   n = (n >>  4) + (n & 0x000F);        // Max 0x3D.
   n = (n >>  2) + (n & 0x0003);        // Max 0x11.
   n = (n >>  2) + (n & 0x0003);        // Max 0x6.
   return (0x0924 >> (n << 1)) & 3;
}
```

FIGURE 10–25. Unsigned remainder modulo 3, digit summing and an in-register lookup.

```
int remu3(unsigned n) {
   static char table[62] = {0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2,  0,1};

   n = (n >> 16) + (n & 0xFFFF);    // Max 0x1FFFE.
   n = (n >>  8) + (n & 0x00FF);    // Max 0x2FD.
   n = (n >>  4) + (n & 0x000F);    // Max 0x3D.
   return table[n];
}
```

FIGURE 10–26. Unsigned remainder modulo 3, digit summing and an in-memory lookup.

```
int remu5(unsigned n) {
   n = (n >> 16) + (n & 0xFFFF);            // Max 0x1FFFE.
   n = (n >>  8) + (n & 0x00FF);            // Max 0x2FD.
   n = (n >>  4) + (n & 0x000F);            // Max 0x3D.
   n = (n>>4) - ((n>>2) & 3) + (n & 3);     // -3 to 6.
   return (01043210432 >> 3*(n + 3)) & 7;   // Octal const.
}
```

FIGURE 10–27. Unsigned remainder modulo 5, digit summing method.

The instruction count can be reduced by using a table, similar to what is done in Figure 10–26. In fact, the code is identical, except the table is:

```
static char table[62] = {0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1};
```

For the unsigned remainder modulo 7, the code of Figure 10–28 uses the relation $8^k \equiv 1 \pmod 7$ (nine elementary instructions, plus an indexed *load*).

As a final example, the code of Figure 10–29 computes the remainder of unsigned division by 9. It is based on the relation $8 \equiv -1 \pmod 9$. As shown, it is nine elementary instructions, plus an indexed load. The elementary instruction count can be reduced to six by using a table of size 831 (decimal).

```
int remu7(unsigned n) {

   static char table[75] = {0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
             0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
             0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
             0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4};

   n = (n >> 15) + (n & 0x7FFF);        // Max 0x27FFE.
   n = (n >>  9) + (n & 0x001FF);       // Max 0x33D.
   n = (n >>  6) + (n & 0x0003F);       // Max 0x4A.
   return table[n];
}
```

FIGURE 10–28. Unsigned remainder modulo 7, digit summing method.

```
int remu9(unsigned n) {

   int r;
   static char table[75] = {0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2};

   r = (n & 0x7FFF) - (n >> 15);     // FFFE0001 to 7FFF.
   r = (r & 0x01FF) - (r >>  9);     // FFFFFFC1 to 2FF.
   r = (r & 0x003F) + (r >>  6);     // 0 to 4A.
   return table[r];
}
```

FIGURE 10–29. Unsigned remainder modulo 9, digit summing method.

Signed Remainder

The digit summing method can be adapted to compute the remainder resulting from signed division. There seems to be no better way than to add a few steps to correct the result of the method as applied to unsigned division. Two corrections are necessary: (1) correct for a different interpretation of the sign bit, and (2) add or subtract a multiple of the divisor *d* to get the result in the range 0 to – (*d* – 1).

For division by 3, the unsigned remainder code interprets the sign bit of the dividend *n* as contributing 2 to the remainder (because 2^{31} mod 3 = 2). For the remainder of signed division, the sign bit contributes only 1 (because (-2^{31}) mod 3 = 1). Therefore, we can use the code for an unsigned remainder and correct its result by subtracting 1. Then, the result must be put in the range 0 to –2. That is, the result of the unsigned remainder code must be mapped as follows:

$$(0, 1, 2) \quad (-1, 0, 1) \quad (-1, 0, -2).$$

This adjustment can be done fairly efficiently by subtracting 1 from the unsigned remainder if it is 0 or 1, and subtracting 4 if it is 2 (when the dividend is negative). The code must not alter the dividend `n`, because it is needed in this last step.

This procedure can easily be applied to any of the functions given for the unsigned remainder modulo 3. For example, applying it to Figure 10–26 on page 265 gives the function shown in Figure 10–30. It is 13 elementary instructions, plus an indexed *load*. The instruction count can be reduced by using a larger table.

```
int rems3(int n) {
   unsigned r;
   static char table[62] = {0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1};

   r = n;
   r = (r >> 16) + (r & 0xFFFF);     // Max 0x1FFFE.
   r = (r >>  8) + (r & 0x00FF);     // Max 0x2FD.
   r = (r >>  4) + (r & 0x000F);     // Max 0x3D.
   r = table[r];
   return r - (((unsigned)n >> 31) << (r & 2));
}
```

FIGURE 10–30. Signed remainder modulo 3, digit summing method.

Figures 10–31 to 10–33 show similar code for computing the signed remainder of division by 5, 7, and 9. All the functions consist of 15 elementary operations, plus an indexed *load*. They use signed right shifts, and the final adjustment consists of subtracting the modulus if the dividend is negative and the remainder is nonzero. The number of instructions can be reduced by using larger tables.

```
int rems5(int n) {
   int r;
   static char table[62] = {2,3,4, 0,1,2,3,4, 0,1,2,3,4,
             0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
             0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
             0,1,2,3,4, 0,1,2,3};

   r = (n >> 16) + (n & 0xFFFF);  // FFFF8000 to 17FFE.
   r = (r >>  8) + (r & 0x00FF);  // FFFFFF80 to 27D.
   r = (r >>  4) + (r & 0x000F);  // -8 to 53 (decimal).
   r = table[r + 8];
   return r - (((int)(n & -r) >> 31) & 5);
}
```

FIGURE 10–31. Signed remainder modulo 5, digit summing method.

```
int rems7(int n) {
   int r;
   static char table[75] =   {5,6, 0,1,2,3,4,5,6,
     0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
     0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
     0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2};

   r = (n >> 15) + (n & 0x7FFF);   // FFFF0000 to 17FFE.
   r = (r >>  9) + (r & 0x001FF);  // FFFFFF80 to 2BD.
   r = (r >>  6) + (r & 0x0003F);  // -2 to 72 (decimal).
   r = table[r + 2];
   return r - (((int)(n & -r) >> 31) & 7);
}
```

FIGURE 10–32. Signed remainder modulo 7, digit summing method.

```
int rems9(int n) {
   int r;
   static char table[75] = {7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0};

   r = (n & 0x7FFF) - (n >> 15);   // FFFF7001 to 17FFF.
   r = (r & 0x01FF) - (r >>  9);   // FFFFFF41 to 0x27F.
   r = (r & 0x003F) + (r >>  6);   // -2 to 72 (decimal).
   r = table[r + 2];
   return r - (((int)(n & -r) >> 31) & 9);
}
```

FIGURE 10–33. Signed remainder modulo 9, digit summing method.

10–20 Remainder by Multiplication and Shifting Right

The method described in this section applies, in principle, to all integer divisors greater than 2, but as a practical matter only to fairly small divisors and to divisors of the form $2^k - 1$. As in the preceding section, in most cases the code resorts to a table lookup after a fairly short calculation.

Unsigned Remainder

This section uses the mathematical (not computer algebra) notation *a* mod *b*, where *a* and *b* are integers and $b > 0$, to denote the integer *x*, $0 \le x < b$, that satisfies $x \equiv a \pmod{b}$.

To compute *n* mod 3, observe that

$$n \bmod 3 = \left\lfloor \frac{4}{3}n \right\rfloor \bmod 4. \tag{3}$$

Proof: Let $n = 3k + \delta$, where δ and *k* are integers and $0 \le \delta \le 2$. Then

$$\left\lfloor \frac{4}{3}(3k+\delta) \right\rfloor \bmod 4 = \left\lfloor 4k + \frac{4\delta}{3} \right\rfloor \bmod 4 = \left\lfloor \frac{4\delta}{3} \right\rfloor \bmod 4.$$

Clearly, the value of the last expression is 0, 1, or 2 for δ = 0, 1, or 2 respectively. This allows changing the problem of computing the remainder modulo 3 to one of computing the remainder modulo 4, which is of course much easier on a binary computer.

Relations like (3) do not hold for all moduli, but similar relations do hold if the modulus is of the form $2^k - 1$, for *k* an integer greater than 1. For example, it is easy to show that

$$n \bmod 7 = \left\lfloor \frac{8}{7}n \right\rfloor \bmod 8.$$

For numbers not of the form $2^k - 1$, there is no such simple relation, but there is a certain uniqueness property that can be used to compute the remainder for other divisors. For example, if the divisor is 10 (decimal), consider the expression

$$\left\lfloor \frac{16}{10}n \right\rfloor \bmod 16. \tag{4}$$

Let $n = 10k + \delta$ where $0 \le \delta \le 9$. Then

$$\left\lfloor \frac{16}{10}n \right\rfloor \bmod 16 = \left\lfloor \frac{16}{10}(10k+\delta) \right\rfloor \bmod 16 = \left\lfloor \frac{16\delta}{10} \right\rfloor \bmod 16.$$

For δ = 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, the last expression takes on the values 0, 1, 3, 4, 6, 8, 9, 11, 12, and 14 respectively. The latter numbers are all distinct. Therefore, if we can find a reasonably easy way to compute (4), we can translate 0 to 0, 1 to 1, 3 to 2, 4 to 3, and so on, to obtain the remainder of division by 10. This will generally require a translation table of size equal to the next power of 2 greater than the divisor, so the method is practical only for fairly small divisors (and for divisors of the form 2^k – 1, for which table lookup is not required).

The code to be shown was derived by using a little of the above theory and a lot of trial and error.

Consider the remainder of unsigned division by 3. Following (3), we wish to compute the rightmost two bits of the integer part of $4n/3$. This can be done approximately by multiplying by $2^{32} / 3$ and then dividing by 2^{30} using a *shift right* instruction. When the multiplication by $2^{32} / 3$ is done (using the *multiply* instruction that gives the low-order 32 bits of the product), high-order bits will be lost. But that doesn't matter, and, in fact, it's helpful, because we want the result modulo 4. Therefore, because $2^{32} / 3$ = 0x55555555, a possible plan is to compute

$$r \leftarrow (\mathrm{0x55555555} * n) \overset{u}{\gg} 30.$$

Experiment indicates that this works for n in the range 0 to $2^{30} + 2$. It almost works, I should say; if n is nonzero and a multiple of 3, it gives the result 3. Therefore, it must be followed by a translation step that translates (0, 1, 2, 3) to (0, 1, 2, 0) respectively.

To extend the range of applicability, the multiplication must be done more accurately. Two more bits of accuracy suffice (that is, multiplying by **0x55555555.4**). The following calculation, followed by the translation step, works for all n representable as an unsigned 32-bit integer:

$$r \leftarrow (\mathrm{0x55555555} * n + (n \overset{u}{\gg} 2)) \overset{u}{\gg} 30.$$

It is, of course, possible to give a formal proof of this, but the algebra is quite lengthy and error prone.

The translation step can be done in three or four instructions on most machines, but there is a way to avoid it at a cost of two instructions. The above expression for computing r estimates low. If you estimate slightly high, the result is always 0, 1, or 2. This gives the C function shown in Figure 10–34 (eight instructions, including a *multiply*).

```
int remu3(unsigned n) {
   return (0x55555555*n + (n >> 1) - (n >> 3)) >> 30;
}
```

FIGURE 10–34. Unsigned remainder modulo 3, multiplication method.

The multiplication can be expanded, giving the 13-instruction function shown in Figure 10–35 that uses only *shift*'s and *add*'s.

```
int remu3(unsigned n) {
   unsigned r;

   r = n + (n << 2);
   r = r + (r << 4);
   r = r + (r << 8);
   r = r + (r << 16);
   r = r + (n >> 1);
   r = r - (n >> 3);
   return r >> 30;
}
```

FIGURE 10–35. Unsigned remainder modulo 3, multiplication (expanded) method.

The remainder of unsigned division by 5 can be computed very similarly to the remainder of division by 3. Let $n = 5k + r$ with $0 \le r \le 4$. Then $(8 / 5)n \bmod 8 = (8 / 5)(5k + r) \bmod 8 = (8 / 5)r \bmod 8$. For r = 0, 1, 2, 3, and 4, this takes on the values 0, 1, 3, 4, and 6 respectively. Since $2^{32} / 5$ = 0x33333333, this leads to the function shown in Figure 10–36 (11 instructions, including a *multiply*). The last step (code on the `return` statement) is mapping (0, 1, 3, 4, 6, 7) to (0, 1, 2, 3, 4, 0) respectively, using an in-register method rather than an indexed *load* from memory. By also mapping 2 to 2 and 5 to 4, the precision required in the multiplication by $2^{32} / 5$ is reduced to using just the term `n >> 3` to approximate the missing part of the multiplier (hexadecimal 0.333...). If the “accuracy” term `n >> 3` is omitted, the code still works for `n` ranging from 0 to 0x60000004.

```
int remu5(unsigned n) {
   n = (0x33333333*n + (n >> 3)) >> 29;
   return (0x04432210 >> (n << 2)) & 7;
}
```

FIGURE 10–36. Unsigned remainder modulo 5, multiplication method.

The code for computing the unsigned remainder modulo 7 is similar, but the mapping step is simpler; it is necessary only to convert 7 to 0. One way to code it is shown in Figure 10–37 (11 instructions, including a *multiply*). If the accuracy term `n >> 4` is omitted, the code still works for `n` up to 0x40000006. With both accuracy terms omitted, it works for `n` up to 0x08000006.

```
int remu7(unsigned n) {
   n = (0x24924924*n + (n >> 1) + (n >> 4)) >> 29;
   return n & ((int)(n - 7) >> 31);
}
```

FIGURE 10–37. Unsigned remainder modulo 7, multiplication method.

Code for computing the unsigned remainder modulo 9 is shown in Figure 10–38. It is six instructions, including a *multiply*, plus an indexed *load*. If the accuracy term `n >> 1` is omitted and the multiplier is changed to 0x1C71C71D, the function works for `n` up to 0x1999999E.

```
int remu9(unsigned n) {
   static char table[16] = {0, 1, 1, 2, 2, 3, 3, 4,
                             5, 5, 6, 6, 7, 7, 8, 8};

   n = (0x1C71C71C*n + (n >> 1)) >> 28;
   return table[n];
}
```

FIGURE 10–38. Unsigned remainder modulo 9, multiplication method.

Figure 10–39 shows a way to compute the unsigned remainder modulo 10. It is eight instructions, including a *multiply*, plus an indexed *load* instruction. If the accuracy term `n >> 3` is omitted, the code works for `n` up to 0x40000004. If both accuracy terms are omitted, it works for `n` up to 0x0AAAAAAD.

```
int remu10(unsigned n) {
   static char table[16] = {0, 1, 2, 2, 3, 3, 4, 5,
                             5, 6, 7, 7, 8, 8, 9, 0};

   n = (0x19999999*n + (n >> 1) + (n >> 3)) >> 28;
   return table[n];
}
```

FIGURE 10–39. Unsigned remainder modulo 10, multiplication method.

As a final example, consider the computation of the remainder modulo 63. This function is used by the population count program at the top of page 84. Joe Keane [Keane] has come up with the rather mysterious code shown in Figure 10–40. It is 12 elementary instructions on the basic RISC.

```
int remu63(unsigned n) {
   unsigned t;

   t = (((n >> 12) + n) >> 10) + (n << 2);
   t = ((t >> 6) + t + 3) & 0xFF;
   return (t - (t >> 6)) >> 2;
}
```

FIGURE 10–40. Unsigned remainder modulo 63, Keane's method.

The "multiply and shift right" method leads to the code shown in Figure 10–41. This is 11 instructions on the basic RISC, one being a multiply. This would not be as fast as Keane's method, unless the machine has a very fast multiply and the load of the constant 0x04104104 can move out of a loop.

```
int remu63(unsigned n) {
   n = (0x04104104*n + (n >> 4) + (n >> 10)) >> 26;
   return n & ((n - 63) >> 6);    // Change 63 to 0.
}
```

FIGURE 10–41. Unsigned remainder modulo 63, multiplication method.

On some machines, an improvement can result from expanding the multiplication into shifts and adds as follows (15 elementary instructions for the whole function):

```
r = (n << 2) + (n << 8);        // r = 0x104*n.
r = r + (r << 12);              // r = 0x104104*n.
r = r + (n << 26);              // r = 0x04104104*n.
```

Signed Remainder

As in the case of the digit summing method, the “multiply and shift right” method can be adapted to compute the remainder resulting from signed division. Again, there seems to be no better way than to add a few steps to correct the result of the method as applied to unsigned division. For example, the code shown in Figure 10–42 is derived from Figure 10–34 on page 270 (12 instructions, including a *multiply*).

```
int rems3(int n) {
   unsigned r;

   r = n;
   r = (0x55555555*r + (r >> 1) - (r >> 3)) >> 30;
   return r - (((unsigned)n >> 31) << (r & 2));
}
```

FIGURE 10–42. Signed remainder modulo 3, multiplication method.

Some plausible ways to compute the remainder of signed division by 5, 7, 9, and 10 are shown in Figures 10–43 to 10–46. The code for a divisor of 7 uses quite a few extra instructions (19 in all, including a *multiply*); it might be preferable to use a table similar to that shown for the cases in which the divisor is 5, 9, or 10. In the latter cases, the table used for unsigned division is doubled in size, with the sign bit of the divisor factored in to index the table. Entries shown as u are unused.

```
int rems5(int n) {
   unsigned r;
   static signed char table[16] = {0, 1, 2, 2, 3, u, 4, 0,
                                   u, 0,-4, u,-3,-2,-2,-1};
   r = n;
   r = ((0x33333333*r) + (r >> 3)) >> 29;
   return table[r + (((unsigned)n >> 31) << 3)];
}
```

FIGURE 10–43. Signed remainder modulo 5, multiplication method.

```
int rems7(int n) {
   unsigned r;

   r = n - (((unsigned)n >> 31) << 2);  // Fix for sign.
   r = ((0x24924924*r) + (r >> 1) + (r >> 4)) >> 29;
   r = r & ((int)(r - 7) >> 31);        // Change 7 to 0.
   return r - (((int)(n&-r) >> 31) & 7);// Fix n<0 case.
}
```

FIGURE 10–44. Signed remainder modulo 7, multiplication method.

```
int rems9(int n) {
   unsigned r;
   static signed char table[32] = {0, 1, 1, 2, u, 3, u, 4,
                                    5, 5, 6, 6, 7, u, 8, u,
                                   -4, u,-3, u,-2,-1,-1, 0,
                                    u,-8, u,-7,-6,-6,-5,-5};
   r = n;
   r = (0x1C71C71C*r + (r >> 1)) >> 28;
   return table[r + (((unsigned)n >> 31) << 4)];
}
```

FIGURE 10–45. Signed remainder modulo 9, multiplication method.

```
int rems10(int n) {
   unsigned r;
   static signed char table[32] = {0, 1, u, 2, 3, u, 4, 5,
                                    5, 6, u, 7, 8, u, 9, u,
                                   -6,-5, u,-4,-3,-3,-2, u,
                                   -1, 0, u,-9, u,-8,-7, u};
   r = n;
   r = (0x19999999*r + (r >> 1) + (r >> 3)) >> 28;
   return table[r + (((unsigned)n >> 31) << 4)];
}
```

FIGURE 10–46. Signed remainder modulo 10, multiplication method.

10–21 Converting to Exact Division

Since the remainder can be computed without computing the quotient, the possibility arises of computing the quotient $q = n/d$ by first computing the remainder, subtracting this from the dividend n, and then dividing the difference by the divisor d. This last division is an exact division, and it can be done by multiplying by the multiplicative inverse of d (see Section 10–16, “Exact Division by Constants,” on page 240). This method would be particularly attractive if both the quotient and remainder are wanted.

Let us try this for the case of unsigned division by 3. Computing the remainder by the multiplication method (Figure 10–34 on page 270) leads to the function shown in Figure 10–47.

```
unsigned divu3(unsigned n) {
   unsigned r;

   r = (0x55555555*n + (n >> 1) - (n >> 3)) >> 30;
   return (n - r)*0xAAAAAAAB;
}
```

FIGURE 10–47. Unsigned remainder and quotient with divisor = 3, using exact division.

This is 11 instructions, including two multiplications by large numbers. (The constant `0x55555555` can be generated by shifting the constant `0xAAAAAAAB` right one position.) In contrast, the more straightforward method of computing the quotient `q` using (for example) the code of Figure 10–8 on page 254, requires 14 instructions, including two multiplications by small numbers, or 17 elementary operations if the multiplications are expanded into *shift*'s and *add*'s. If the remainder is also wanted, and it is computed from `r = n - q*3`, the more straightforward method requires 16 instructions, including three multiplications by small numbers, or 20 elementary instructions if the multiplications are expanded into *shift*'s and *add*'s.

The code of Figure 10–47 is not attractive if the multiplications are expanded into *shift*'s and *add*'s; the result is 24 elementary instructions. Thus, the exact division method might be a good one on a machine that does not have *multiply high* but does have a fast modulo 2^{32} *multiply* and slow *divide*, particularly if it can easily deal with the large constants.

For signed division by 3, the exact division method might be coded as shown in Figure 10–48. It is 15 instructions, including two multiplications by large constants.

```
int divs3(int n) {
   unsigned r;

   r = n;
   r = (0x55555555*r + (r >> 1) - (r >> 3)) >> 30;
   r = r - (((unsigned)n >> 31) << (r & 2));
   return (n - r)*0xAAAAAAAB;
}
```

FIGURE 10–48. Signed remainder and quotient with divisor = 3, using exact division.

As a final example, Figure 10–49 shows code for computing the quotient and remainder for unsigned division by 10. It is 12 instructions, including two multiplications by large constants, plus an indexed *load* instruction.

```
unsigned divu10(unsigned n) {
   unsigned r;
   static char table[16] = {0, 1, 2, 2, 3, 3, 4, 5,
                             5, 6, 7, 7, 8, 8, 9, 0};

   r = (0x19999999*n + (n >> 1) + (n >> 3)) >> 28;
   r = table[r];
   return ((n - r) >> 1)*0xCCCCCCCD;
}
```

FIGURE 10–49. Signed remainder and quotient with divisor = 10, using exact division.

10–22 A Timing Test

Many machines have a 32×32   64 *multiply* instruction, so one would expect that to divide by a constant such as 3, the code shown on page 228 would be fastest. If that *multiply* instruction is not present, but the machine has a fast 32×32   32 *multiply* instruction, then the exact division method might be a good one if the machine has a slow *divide* and a fast *multiply*. To test this conjecture, an assembly language program was constructed to compare four methods of dividing by 3. The results are shown in Table 10–4. The machine used was a 667 MHz Pentium III (ca. 2000), and one would expect similar results on many other machines.

TABLE 10–4. UNSIGNED DIVIDE BY 3 ON A PENTIUM III

Division Method	Cycles
Using machine's divide instruction (`divl`)	41.08
Using 32×32 ⇒ 64 *multiply* (code on page 228)	4.28
All elementary instructions (Figure 10–8 on page 254)	14.10
Convert to exact division (Figure 10–47 on page 274)	6.68

The first row gives the time in cycles for just two instructions: an `xorl` to clear the left half of the 64-bit source register, and the `divl` instruction, which evidently takes 40 cycles. The second row also gives the time for just two instructions: *multiply* and *shift right 1* (`mull` and `shrl`). The third row gives the time for a sequence of 21 elementary instructions. It is the code of Figure 10–8 on page 254 using alternative 2, and with the multiplication by 3 done with a single instruction (`leal`). Several *move* instructions are necessary because the machine is (basically) two-address. The last row gives the time for a sequence of 10 instructions: two multiplications (`imull`) and the rest elementary. The two `imull` instructions use 4-byte immediate fields for the large constants. (The signed *multiply* instruction `imull` is used rather than its unsigned counterpart `mull`, because they give the same result in the low-order 32 bits, and `imull` has more addressing modes available.)

The exact division method would be even more favorable compared to the second and third methods if both the quotient and remainder were wanted, because they would require additional code for the computation $r \leftarrow n - q*3$. (The `divl` instruction produces the remainder as well as the quotient.)

10–23 A Circuit for Dividing by 3

There is a simple circuit for dividing by 3 that is about as complex as an adder. It can be constructed very similarly to the elementary way one constructs an n-bit adder from n 1-bit "full adder" circuits. However, in the divider signals flow from most significant to least significant bit.

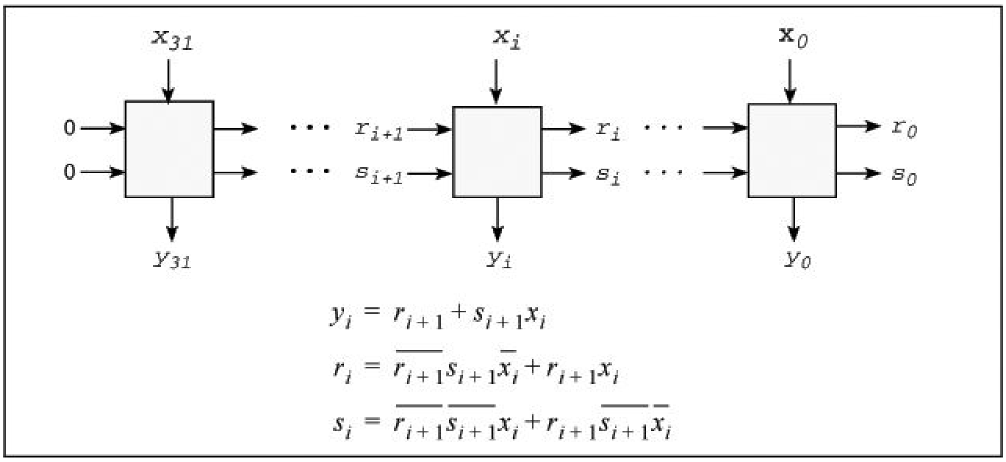
Consider dividing by 3 the way it is taught in grade school, but in binary. To produce each bit of the quotient, you divide 3 into the next bit, but the bit is preceded by a remainder of 0, 1, or 2 from the previous stage. The logic is shown in Table 10–5. Here the remainder is represented by two bits r_i and s_i, with r_i being the most significant bit. The remainder is never 3, so the last two rows of the table represent "don't care" cases.

A circuit for 32-bit division by 3 is shown in Figure 10–50. The quotient is the word consisting of bits y_{31} through y_0, and the remainder is $2r_0 + s_0$.

Another way to implement the divide-by-3 operation in hardware is to use the multiplier to multiply the dividend by the reciprocal of 3 (binary 0.010101...), with appropriate rounding and scaling. This is the technique shown on pages 207 and 228.

TABLE 10–5. LOGIC FOR DIVIDING BY 3

r_{i+1}	s_{i+1}	x_i	y_i	r_i	s_i
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	–	–	–
1	1	1	–	–	–



$$y_i = r_{i+1} + s_{i+1}x_i$$
$$r_i = \overline{r_{i+1}}\,s_{i+1}\,\overline{x_i} + r_{i+1}x_i$$
$$s_i = \overline{r_{i+1}}\,\overline{s_{i+1}}\,x_i + r_{i+1}\,\overline{s_{i+1}}\,\overline{x_i}$$

FIGURE 10–50. Logic circuit for dividing by 3.

Exercises

1. Show that for unsigned division by an even number, the `shrxi` instruction (or equivalent code) can be avoided by first (a) turning off the low-order bit of the dividend (*and* operation) [CavWer] or (b) dividing the dividend by 2 (*shift right 1* instruction) and then dividing by half the divisor.

2. Code a function in Python similar to that of Figure 10–4 on page 240, but for computing the magic number for signed division. Consider only positive divisors.

3. Show how you would use Newton's method to calculate the multiplicative inverse of an integer d modulo 81. Show the calculations for d = 146.

I think that I shall never envision
An op unlovely as division.

An op whose answer must be guessed

And then, through multiply, assessed;

An op for which we dearly pay,
In cycles wasted every day.

Division code is often hairy;
Long division's downright scary.

The proofs can overtax your brain,
The ceiling and floor may drive you insane.

Good code to divide takes a Knuthian hero,
But even God can't divide by zero!

Chapter 11. Some Elementary Functions

11–1 Integer Square Root

By the "integer square root" function, we mean the function $\sqrt{x}$. To extend its range of application and to avoid deciding what to do with a negative argument, we assume x is unsigned. Thus, $0 \le x \le 2^{32} - 1$.

Newton's Method

For floating-point numbers, the square root is almost universally computed by Newton's method. This method begins by somehow obtaining a starting estimate g_0 of $\sqrt{a}$. Then, a series of more accurate estimates is obtained from

$$g_{n+1} = \left(g_n + \frac{a}{g_n}\right)/2.$$

The iteration converges quadratically—that is, if at some point g_n is accurate to n bits, then g_{n+1} is accurate to $2n$ bits. The program must have some means of knowing when it has iterated enough so it can terminate.

It is a pleasant surprise that Newton's method works fine in the domain of integers. To see this, we need the following theorem:

THEOREM. *Let* $g_{n+1} = (g_n + a / g_n)/2$, with g_n, *a integers greater than* 0. *Then*

$$(a)\ \textit{if } g_n > \lfloor\sqrt{a}\rfloor \textit{ then } \lfloor\sqrt{a}\rfloor \le g_{n+1} < g_n, \textit{ and}$$

$$(b)\ \textit{if } g_n = \lfloor\sqrt{a}\rfloor \textit{ then } \lfloor\sqrt{a}\rfloor \le g_{n+1} \le \lfloor\sqrt{a}\rfloor + 1.$$

That is, if we have an integral guess g_n to $\sqrt{a}$ that is too high, then the next guess g_{n+1} will be strictly less than the preceding one, but not less than $\sqrt{a}$. Therefore, if we start with a guess that's too high, the sequence converges monotonically. If the guess $g_n = \sqrt{a}$, then the next guess is either equal to g_n or is 1 larger. This provides an easy way to determine when the sequence has converged: If we start with $g_0 \ge \sqrt{a}$, convergence has occurred when $g_{n+1} \ge g_n$, and then the result is precisely g_n.

The case $a = 0$ must be treated specially, because this procedure would lead to dividing 0 by 0.

Proof. (a) Because g_n is an integer,

$$g_{n+1} = \left\lfloor\left(g_n + \left\lfloor\frac{a}{g_n}\right\rfloor\right)/2\right\rfloor = \left\lfloor\left\lfloor g_n + \frac{a}{g_n}\right\rfloor/2\right\rfloor = \left\lfloor\left(g_n + \frac{a}{g_n}\right)/2\right\rfloor = \left\lfloor\frac{g_n^2 + a}{2g_n}\right\rfloor.$$

Because $g_n > \sqrt{a}$ and g_n is an integer, $g_n > \sqrt{a}$. Define ε by $g_n = (1 + \varepsilon)\sqrt{a}$. Then $\varepsilon > 0$ and

$$\left\lfloor \frac{g_n^2 + a}{2g_n} \right\rfloor = g_{n+1} \le \frac{g_n^2 + a}{2g_n},$$

$$\left\lfloor \frac{(1+\varepsilon)^2 a + a}{2(1+\varepsilon)\sqrt{a}} \right\rfloor = g_{n+1} < \frac{g_n^2 + g_n^2}{2g_n},$$

$$\left\lfloor \frac{2 + 2\varepsilon + \varepsilon^2}{2(1+\varepsilon)}\sqrt{a} \right\rfloor = g_{n+1} < g_n,$$

$$\left\lfloor \frac{2 + 2\varepsilon}{2(1+\varepsilon)}\sqrt{a} \right\rfloor \le g_{n+1} < g_n,$$

$$\lfloor \sqrt{a} \rfloor \le g_{n+1} < g_n.$$

(b) Because $g_n = \sqrt{a}$, $\sqrt{a} - 1 < g_n \le \sqrt{a}$, so that $g_n^2 \le a < (g_n + 1)^2$. Hence, we have

$$\left\lfloor \frac{g_n^2 + g_n^2}{2g_n} \right\rfloor \le g_{n+1} \le \left\lfloor \frac{g_n^2 + (g_n + 1)^2}{2g_n} \right\rfloor,$$

$$\lfloor g_n \rfloor \le g_{n+1} \le \left\lfloor g_n + 1 + \frac{1}{2g_n} \right\rfloor,$$

$$\lfloor \sqrt{a} \rfloor \le g_{n+1} \le \lfloor g_n + 1 \rfloor \quad \text{(because } g_n \text{ is an integer and } \frac{1}{2g_n} < 1\text{)},$$

$$\lfloor \sqrt{a} \rfloor \le g_{n+1} \le \lfloor g_n \rfloor + 1 = \lfloor \sqrt{a} \rfloor + 1.$$

The difficult part of using Newton's method to calculate $\sqrt{x}$ is getting the first guess. The procedure of Figure 11–1 sets the first guess g_0 equal to the least power of 2 that is greater than or equal to $\sqrt{x}$ For example, for $x = 4$, $g_0 = 2$, and for $x = 5$, $g_0 = 4$.

```
int isqrt(unsigned x) {
  unsigned x1;
  int s, g0, g1;

  if (x <= 1) return x;
  s = 1;
  x1 = x - 1;
  if (x1 > 65535) {s = s + 8; x1 = x1 >> 16;}
  if (x1 > 255)   {s = s + 4; x1 = x1 >> 8;}
  if (x1 > 15)    {s = s + 2; x1 = x1 >> 4;}
  if (x1 > 3)     {s = s + 1;}

  g0 = 1 << s;                // g0 = 2**s.
```

```
   g1 = (g0 + (x >> s)) >> 1; // g1 = (g0 + x/g0)/2.

   while (g1 < g0) {          // Do while approximations
      g0 = g1;                // strictly decrease.
      g1 = (g0 + (x/g0)) >> 1;
   }
   return g0;
}
```

FIGURE 11–1. Integer square root, Newton's method.

Because the first guess g_0 is a power of 2, it is not necessary to do a real division to get g_1; instead, a *shift right* suffices.

Because the first guess is accurate to about one bit, and Newton's method converges quadratically (the number of bits of accuracy doubles with each iteration), one would expect the procedure to converge within about five iterations (on a 32-bit machine), which requires four divisions (because the first iteration substitutes a *shift right)*. An exhaustive experiment reveals that the maximum number of divisions is five, or four for arguments up to 16,785,407.

If *number of leading zeros* is available, then getting the first guess is very simple: Replace the first seven executable lines in the procedure above with

```
   if (x <= 1) return x;
   s = 16 - nlz(x - 1)/2;
```

Another alternative, if *number of leading zeros* is not available, is to compute *s* by means of a binary search tree. This method permits getting a slightly better value of g_0: the least power of 2 that is greater than or equal to $\sqrt{x}$. For some values of *x*, this gives a smaller value of g_0, but a value large enough so that the convergence criterion of the theorem still holds. The difference in these schemes is illustrated in the following table.

Range of x for Figure 11–1	Range of x for Figure 11–2	First Guess g_0
0	0	0
1	1 to 3	1
2 to 4	4 to 8	2
5 to 16	9 to 24	4
17 to 64	25 to 80	8
65 to 256	81 to 288	16
...	...	...
$2^{28}+1$ to 2^{30}	$(2^{14}+1)^2$ to $(2^{15}+1)^2-1$	2^{15}
$2^{30}+1$ to $2^{32}-1$	$(2^{15}+1)^2$ to $2^{32}-1$	2^{16}

This procedure is shown in Figure 11–2. It is convenient there to treat small values of $x (0 \le x \le 24)$ specially, so that no divisions are done for them.

```
int isqrt(unsigned x) {
   int s, g0, g1;

   if (x <= 4224)
      if (x <= 24)
         if (x <= 3) return (x + 3) >> 2;
         else if (x <= 8) return 2;
         else return (x >> 4) + 3;
      else if (x <= 288)
         if (x <= 80) s = 3; else s = 4;
      else if (x <= 1088) s = 5; else s = 6;
   else if (x <= 1025*1025 - 1)
      if (x <= 257*257 - 1)
         if (x <= 129*129 - 1) s = 7; else s = 8;
      else if (x <= 513*513 - 1) s = 9; else s = 10;
   else if (x <= 4097*4097 - 1)
      if (x <= 2049*2049 - 1) s = 11; else s = 12;
   else if (x <= 16385*16385 - 1)
      if (x <= 8193*8193 - 1) s = 13; else s = 14;
   else if (x <= 32769*32769 - 1) s = 15; else s = 16;
   g0 = 1 << s;            // g0 = 2**s.

   // Continue as in Figure 11–1.
```

FIGURE 11–2. Integer square root, binary search for first guess.

The worst-case execution time of the algorithm of Figure 11–1, on the basic RISC, is about $26 + (D + 6)\,n$ cycles, where D is the divide time in cycles and n is the number of times the *while*-loop is executed. The worst-case execution time of Figure 11–2 is about $27 + (D + 6)\,n$ cycles, assuming (in both cases) that the *branch* instructions take one cycle. The table that follows gives the average number of times the loop is executed by the two algorithms, for x uniformly distributed in the indicated range.

x	Figure 11–1	Figure 11–2
0 to 9	0.80	0
0 to 99	1.46	0.83
0 to 999	1.58	1.44
0 to 9999	2.13	2.06
0 to $2^{32} - 1$	2.97	2.97

If we assume a divide time of 20 cycles and x ranging uniformly from 0 to 9999, then both algorithms execute in about 81 cycles.

Binary Search

Because the algorithms based on Newton's method start out with a sort of binary search to obtain the first guess, why not do the whole computation with a binary search? This method would start out with two bounds, perhaps initialized to 0 and 2^{16}. It would make a guess at the midpoint of the bounds. If the square of the midpoint is greater than the argument *x*, then the upper bound is changed to be equal to the midpoint. If the square of the midpoint is less than the argument *x*, then the lower bound is changed to be equal to the midpoint. The process ends when the upper and lower bounds differ by 1, and the result is the lower bound.

This avoids division, but requires quite a few multiplications—16 if 0 and 2^{16} are used as the initial bounds. (The method gets one more bit of precision with each iteration.) Figure 11–3 illustrates a variation of this procedure, which uses initial values for the bounds that are slight improvements over 0 and 2^{16}. The procedure shown in Figure 11–3 also saves a cycle in the loop, for most RISC machines, by altering *a* and *b* in such a way that the comparison is $b \geq a$ rather than $b - a \geq 1$.

The predicates that must be maintained at the beginning of each iteration are $a \leq \sqrt{x} + 1$ and $b \geq \sqrt{x}$. The initial value of *b* should be something that's easy to compute and close to $\sqrt{x}$. Reasonable initial values are x, $x \div 4 + 1$, $x \div 8 + 2$, $x \div 16 + 4$, $x \div 32 + 8$, $x \div 64 + 16$, and so on. Expressions near the beginning of this list are better initial bounds for small *x*, and those near the end are better for larger *x*. (The value $x \div 2 + 1$ is acceptable, but probably not useful, because $x \div 4 + 1$ is everywhere a better or equal bound.)

```
int isqrt(unsigned x) {
   unsigned a, b, m;       // Limits and midpoint.

   a = 1;
   b = (x >> 5) + 8;       // See text.
   if (b > 65535) b = 65535;
   do {
      m = (a + b) >> 1;
      if (m*m > x) b = m - 1;
      else         a = m + 1;
   } while (b >= a);
   return a - 1;
}
```

FIGURE 11–3. Integer square root, simple binary search.

Seven variations on the procedure shown in Figure 11–3 can be more or less mechanically generated by substituting $a + 1$ for *a*, or $b - 1$ for *b*, or by changing $m = (a + b) \div 2$ to $m = (a + b + 1) \div 2$, or some combination of these substitutions.

The execution time of the procedure shown in Figure 11–3 is about $6 + (M + 7.5)n$, where *M* is the multiplication time in cycles and *n* is the number of times the loop is executed. The following table gives the average number of times the loop is executed, for *x* uniformly distributed in the indicated range.

x	Average Number of Loop Iterations
0 to 9	3.00
0 to 99	3.15
0 to 999	4.68
0 to 9999	7.04
0 to $2^{32} - 1$	16.00

If we assume a multiplication time of 5 cycles and *x* ranging uniformly from 0 to 9999, the algorithm runs in about 94 cycles. The maximum execution time ($n = 16$) is about 206 cycles.

If *number of leading zeros* is available, the initial bounds can be set from

```
b = (1 << (33 - nlz(x))/2) - 1;
a = (b + 3)/2;
```

That is, $b = 2^{(33 - \mathrm{nlz}(x)) \overset{u}{\div} 2} - 1$. These are very good bounds for small values of *x* (one loop iteration for $0 \le x \le 15$), but only a moderate improvement, for large *x*, over the bounds calculated in Figure 11–3. For *x* in the range 0 to 9999, the average number of iterations is about 5.45, which gives an execution time of about 74 cycles, using the same assumptions as above.

A Hardware Algorithm

There is a shift-and-subtract algorithm for computing the square root that is quite similar to the hardware division algorithm described in Figure 9–2 on page 193. Embodied in hardware on a 32-bit machine, this algorithm employs a 64-bit register that is initialized to 32 0-bits followed by the argument *x*. On each iteration, the 64-bit register is shifted left two positions, and the current result *y* (initially 0) is shifted left one position. Then $2y + 1$ is subtracted from the left half of the 64-bit register. If the result of the subtraction is nonnegative, it replaces the left half of the 64-bit register, and 1 is added to *y* (this does not require an adder, because *y* ends in 0 at this point). If the result of the subtraction is negative, then the 64-bit register and *y* are left unaltered. The iteration is done 16 times.

This algorithm was described in 1945 [JVN].

Perhaps surprisingly, this process runs in about half the time of that of the 64 ÷ 32 32 hardware division algorithm cited, because it does half as many iterations and each iteration is about equally complex in the two algorithms.

To code this algorithm in software, it is probably best to avoid the use of a doubleword shift register, which requires about four instructions to shift. The algorithm in Figure 11–4 [GLS1] accomplishes this by shifting *y* and a mask bit *m* to the right. It executes in about 149 basic RISC instructions (average). The two expressions `y | m` could also be `y + m`.

The operation of this algorithm is similar to the grade-school method. It is

$\lfloor \sqrt{179} \rfloor$

illustrated here, for finding on an 8-bit machine.

```
   1011 0011 x0  Initially, x = 179 (0xB3).
 - 1         b1
   ---------
   0111 0011 x1  0100 0000 y1
 - 101       b2  0010 0000 y2
   ---------
   0010 0011 x2  0011 0000 y2
 - 11 01     b3  0001 1000 y3
   ---------
   0010 0011 x3  0001 1000 y3   (Can't subtract).
 -   1 1001  b4  0000 1100 y4
   ---------
   0000 1010 x4  0000 1101 y4
```

The result is 13 with a remainder of 10 left in register `x`.

```
int isqrt(unsigned x) {
   unsigned m, y, b;

   m = 0x40000000;
   y = 0;
   while(m != 0) {               // Do 16 times.
      b = y | m;
      y = y >> 1;
      if (x >= b) {
         x = x - b;
         y = y | m;
      }
      m = m >> 2;
   }
   return y;
}
```

FIGURE 11–4. Integer square root, hardware algorithm.

It is possible to eliminate the `if x >= b` test by the usual trickery involving *shift right signed 31*. It can be proved that the high-order bit of `b` is always zero (in fact, $b \le 5 \cdot 2^{28}$), which simplifies the `x >= b` predicate (see page 23). The result is that the *if* statement group can be replaced with

```
t = (int)(x | ~(x - b)) >> 31;   // -1 if x >= b, else 0.
x = x - (b & t);
y = y | (m & t);
```

This replaces an average of three cycles with seven, assuming the machine has *or not*, but it might be worthwhile if a conditional branch in this context takes more than five cycles.

Somehow it seems that it should be easier than some hundred cycles to compute an integer square root in software. Toward this end, we offer the expressions that follow to compute it for very small values of the argument. These can be useful to speed up some of the algorithms given above, if the argument is expected to be small.

The expression	is correct in the range	and uses this many instructions (full RISC).
x	0 to 1	0
$x > 0$	0 to 3	1
$(x + 3) \overset{u}{\div} 4$	0 to 3	2
$x \overset{u}{\gg} (x \overset{u}{\div} 2)$	0 to 3	2
$x \overset{u}{\gg} (x > 1)$	0 to 5	2
$(x + 12) \overset{u}{\div} 8$	1 to 8	2
$(x + 15) \overset{u}{\div} 8$	4 to 15	2
$(x > 0) + (x > 3)$	0 to 8	3
$(x > 0) + (x > 3) + (x > 8)$	0 to 15	5

Ah, the elusive square root,
It should be a cinch to compute.
But the best we can do
Is use powers of two
And iterate the method of Newt!

11–2 Integer Cube Root

For cube roots, Newton's method does not work out very well. The iterative formula is a bit complex:

$$x_{n+1} = \frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right),$$

and there is of course the problem of getting a good starting value x_0.

However, there is a hardware algorithm, similar to the hardware algorithm for square root, that is not too bad for software. It is shown in Figure 11–5.

The three *add*'s of 1 can be replaced by *or*'s of 1, because the value being incremented is even. Even with this change, the algorithm is of questionable value for implementation in hardware, mainly because of the multiplication $y * (y + 1)$.

This multiplication is easily avoided by applying the compiler optimization of strength reduction to the `y`-squared term. Introduce another unsigned variable `y2` that will have the value of `y`-squared, by updating `y2` appropriately wherever `y` receives a new value.

Just before `y = 0` insert `y2 = 0`. Just before `y = 2*y` insert `y2 = 4*y2`. Change the assignment to `b` to `b = (3*y2 + 3*y + 1) << s` (and factor out the 3). Just before `y = y + 1`, insert `y2 = y2 + 2*y + 1`. The resulting program has no multiplications except by small constants, which can be changed to *shift*'s and *add*'s. This program has three *add*'s of 1, which can all be changed to *or*'s of 1. It is faster unless your machine's *multiply* instruction takes only two or fewer cycles.

```
int icbrt(unsigned x) {
   int s;
   unsigned y, b;

   y = 0;
   for (s = 30; s >= 0; s = s - 3) {
      y = 2*y;
      b = (3*y*(y + 1) + 1) << s;
      if (x >= b) {
         x = x - b;
         y = y + 1;
      }
   }
   return y;
}
```

FIGURE 11–5. Integer cube root, hardware algorithm.

Caution: [GLS1] points out that the code of Figure 11–5, and its strength-reduced derivative, do not work if adapted in the obvious way to a 64-bit machine. The assignment to `b` can then overflow. This problem can be avoided by dropping the *shift left* of `s` from the assignment to `b`, inserting after the assignment to `b` the assignment `bs = b << s`, and changing the two lines `if (x >= b) {x = x – b ...` to `if (x >= bs && b == (bs >> s)) {x = x – bs ....`

11–3 Integer Exponentiation

Computing x^n by Binary Decomposition of n

A well-known technique for computing x^n, when n is a nonnegative integer, involves the binary representation of n. The technique applies to the evaluation of an expression of the form $x \cdot x \cdot x \cdot \ldots \cdot x$ where $\cdot$ is any associative operation, such as addition, multiplication including matrix multiplication, and string concatenation (as suggested by the notation $(\text{‘ab’})^3$ = ‘ababab’). As an example, suppose we wish to compute $y = x^{13}$. Because 13 expressed in binary is 1101 (that is, 13 = 8 + 4 + 1),

$$x^{13} = x^{8+4+1} = x^8 \cdot x^4 \cdot x^1.$$

Thus, x^{13} can be computed as follows:

$$\begin{aligned} t_1 &\leftarrow x^2 \\ t_2 &\leftarrow t_1^2 \\ t_3 &\leftarrow t_2^2 \\ y &\leftarrow t_3 \cdot t_2 \cdot x \end{aligned}$$

This requires five multiplications, considerably fewer than the 12 that would be required by repeated multiplication by x.

If the exponent is a variable, known to be a nonnegative integer, the technique can be employed in a subroutine, as shown in Figure 11–6.

The number of multiplications done by this method is, for exponent $n \geq 1$,

$$\lfloor \log_2 n \rfloor + \text{nbits}(n) - 1.$$

This is not always the minimal number of multiplications. For example, for $n = 27$, the binary decomposition method computes

$$x^{16} \cdot x^8 \cdot x^2 \cdot x^1,$$

which requires seven multiplications. However, the scheme illustrated by

$$((x^3)^3)^3$$

requires only six. The smallest number for which the binary decomposition method is not optimal is $n = 15$ (*Hint:* $x^{15} = (x^3)^5$).

Perhaps surprisingly, there is no known simple method that, for all n, finds an optimal sequence of multiplications to compute x^n. The only known methods involve an extensive search. The problem is discussed at some length in [Knu2, 4.6.3].

The binary decomposition method has a variant that scans the binary representation of the exponent in left-to-right order [Rib, 32], which is analogous to the left-to-right method of converting binary to decimal. Initialize the result y to 1, and scan the exponent from left to right. When a 0 is encountered, square y. When a 1 is encountered, square y and multiply it by x. This computes $x^{13} = x^{1101_2}$ as

$$(((1^2 \cdot x)^2 \cdot x)^2)^2 \cdot x.$$

```
int iexp(int x, unsigned n) {
   int p, y;

   y = 1;                     // Initialize result
   p = x;                     // and p.
   while(1) {
      if (n & 1) y = p*y;     // If n is odd, mult by p.
      n = n >> 1;             // Position next bit of n.
      if (n == 0) return y;   // If no more bits in n.
      p = p*p;                // Power for next bit of n.
   }
}
```

FIGURE 11–6. Computing x^n by binary decomposition of n.

It always requires the same number of (nontrivial) multiplications as the right-to-left method of Figure 11–6.

2^n in Fortran

The IBM XL Fortran compiler takes the definition of this function to be

$$\text{pow2}(n) = \begin{cases} 2^n, & 0 \le n \le 30, \\ -2^{31}, & n = 31, \\ 0, & n < 0 \text{ or } n \ge 32. \end{cases}$$

It is assumed that n and the result are interpreted as signed integers. The ANSI/ISO Fortran standard requires that the result be 0 if $n < 0$. The definition above for $n \ge 31$ seems reasonable in that it is the correct result modulo 2^{32}, and it agrees with what repeated multiplication would give.

The standard way to compute 2^n is to put the integer 1 in a register and shift it left n places. This does not satisfy the Fortran definition, because shift amounts are usually treated modulo 64 or modulo 32 (on a 32-bit machine), which gives incorrect results for large or negative shift amounts.

If your machine has *number of leading zeros*, pow2(n) can be computed in four instructions as follows [Shep]:

```
$x \leftarrow \text{nlz}(n \overset{u}{\gg} 5);$          $// x \leftarrow 32 \text{ if } 0 \le n \le 31, x < 32 \text{ otherwise.}$
$x \leftarrow x \overset{u}{\gg} 5;$                 $// x \leftarrow 1 \text{ if } 0 \le n \le 31, 0 \text{ otherwise.}$
$pow2 \leftarrow x \ll n;$
```

The *shift right* operations are "logical" (not sign-propagating), even though n is a signed quantity.

If the machine does not have the nlz instruction, its use above can be replaced with one of the $x = 0$ tests given in "Comparison Predicates" on page 23, changing the expression $x \overset{u}{\gg} 5$ to $x \overset{u}{\gg} 31$. A possibly better method is to realize that the predicate $0 \le x \le 31$ is equivalent to $x \overset{u}{<} 32$, and then simplify the expression for $x \overset{u}{<} y$ given in the cited section; it becomes $\neg x \,\&\, (x - 32)$. This gives a solution in five instructions (four if the machine has *and not*):

```
$x \leftarrow \neg n \,\&\, (n - 32);$          $// x < 0 \text{ iff } 0 \le n \le 31.$
$x \leftarrow x \overset{u}{\gg} 31;$              $// x = 1 \text{ if } 0 \le n \le 31, 0 \text{ otherwise.}$
$pow2 \leftarrow x \ll n;$
```

11–4 Integer Logarithm

By the “integer logarithm” function we mean the function $\log_b x$, where x is a positive integer and b is an integer greater than or equal to 2. Usually, $b = 2$ or 10, and we denote these functions by “ilog2” and “ilog10,” respectively. We use “ilog” when the base is unspecified.

It is convenient to extend the definition to $x = 0$ by defining ilog(0) = –1 [CJS]. There are several reasons for this definition:

- The function ilog2(x) is then related very simply to the *number of leading zeros* function, nlz(x), by the formula shown below, including the case $x = 0$. Thus, if one of these functions is implemented in hardware or software, the other is easily obtained.

$$\text{ilog2}(x) = 31 - \text{nlz}(x)$$

- It is easy to compute log(x) using the formula below. For $x = 1$, this formula implies that ilog(0) = –1.

$$\log(x) = \text{ilog}(x - 1) + 1$$

- It makes the following identity hold for $x = 1$ (but it doesn’t hold for $x = 0$).

$$\text{ilog2}(x \div 2) = \text{ilog2}(x) - 1$$

- It makes the result of ilog(x) a small dense set of integers (–1 to 31 for ilog2(x) on a 32-bit machine, with x unsigned), making it directly useful for indexing a table.
- It falls naturally out of several algorithms for computing ilog2(x) and ilog10(x).

Unfortunately, it isn’t the right definition for “number of digits of x,” which is ilog(x) + 1 for all x except $x = 0$. It seems best to consider that anomalous.

For $x < 0$, ilog(x) is left undefined. To extend its range of utility, we define the function as mapping unsigned numbers to signed numbers. Thus, a negative argument cannot occur.

Integer Log Base 2

Computing ilog2(x) is essentially the same as computing the number of leading zeros, which is discussed in “Counting Leading 0’s” on page 99. All the algorithms in that section can be easily modified to compute ilog2(x) directly, rather than by computing nlz(x) and subtracting the result from 31. (For the algorithm of Figure 5–16 on page 102, change the line `return pop(~x)` to `return pop(x) – 1.)`

Integer Log Base 10

This function has application in converting a number to decimal for inclusion into a line with leading zeros suppressed. The conversion process successively divides by 10, producing the least significant digit first. It would be useful to know ahead of time where the least significant digit should be placed, to avoid putting the converted number in a temporary area and then moving it.

To compute ilog10(x), a table search is quite reasonable. This could be a binary search, but because the table is small and in many applications x is usually small, a simple linear search is probably best. This rather straightforward program is shown in

Figure 11–7.

On the basic RISC, this program can be implemented to execute in about $9 + 4\log_{10}x$ instructions. Thus, it executes in five to 45 instructions, with perhaps 13 (for $10 \le x \le 99$) being typical.

The program in Figure 11–7 can easily be changed into an “in register” version (not using a table). The executable part of such a program is shown in Figure 11–8. This might be useful if the machine has a fast way to multiply by 10.

```
int ilog10(unsigned x) {
   int i;
   static unsigned table[11] = {0, 9, 99, 999, 9999,
      99999, 999999, 9999999, 99999999, 999999999,
      0xFFFFFFFF};

   for (i = -1; ; i++) {
      if (x <= table[i+1]) return i;
   }
}
```

FIGURE 11–7. Integer log base 10, simple table search.

```
   p = 1;
   for (i = -1; i <= 8; i++) {
      if (x < p) return i;
      p = 10*p;
   }
   return i;
```

FIGURE 11–8. Integer log base 10, repeated multiplication by 10.

This program can be implemented to execute in about $10 + 6\log_{10}x$ instructions on the basic RISC (counting the *multiply* as one instruction). This amounts to 16 instructions for $10 \le x \le 99$.

A binary search can be used, giving an algorithm that is loop-free and does not use a table. Such an algorithm might compare x to 10^4, then to either 10^2 or to 10^6, and so on, until the exponent n is found such that $10^n \le x < 10^{n+1}$. The paths execute in ten to 18 instructions, four or five of which are branches (counting the final unconditional branch).

The program shown in Figure 11–9 is a modification of the binary search that has a maximum of four branches on any path and is written in a way that favors small x. It executes in six basic RISC instructions for $10 \le x \le 99$, and in 11 to 16 instructions for $x \ge 100$.

The *shift* instructions in this program are *signed* shifts (which is the reason for the `(int)` casts). If your machine does not have this instruction, one of the alternatives below, which use unsigned shifts, may be preferable. These are illustrated for the case of the first `return` statement. Unfortunately, the first two require *subtract from immediate* for efficient implementation, which most machines don’t have. The last involves adding a large constant (two instructions), but this does not matter for the second and third `return` statements, which require adding a large constant anyway. The large constant is $2^{31} - 1000$.

```
return 3 - ((x - 1000) >> 31);
return 2 + ((999 - x) >> 31);
return 2 + ((x + 2147482648) >> 31);
```

An alternative for the fourth `return` statement is

```
return 8 + ((x + 1147483648) | x) >> 31;
```

where the large constant is $2^{31} - 10^9$. This avoids both the *and not* and the signed shift.

Alternatives for the last *if*-*else* construction are

```
return ((int)(x - 1) >> 31) | ((unsigned)(9 - x) >> 31);
return (x > 9) + (x > 0) - 1;
```

either of which saves a branch.

```
int ilog10(unsigned x) {
   if (x > 99)
      if (x < 1000000)
         if (x < 10000)
            return 3 + ((int)(x - 1000) >> 31);
         else
            return 5 + ((int)(x - 100000) >> 31);
      else
         if (x < 100000000)
            return 7 + ((int)(x - 10000000) >> 31);
         else
            return 9 + ((int)((x-1000000000)&~x) >> 31);
   else
      if (x > 9) return 1;
      else       return ((int)(x - 1) >> 31);
}
```

FIGURE 11–9. Integer log base 10, modified binary search.

If nlz(x) or ilog2(x) is available as an instruction, there are better and more interesting ways to compute ilog10(x). For example, the program in Figure 11–10 does it in two table lookups [CJS].

From `table1` an approximation to ilog10(x) is obtained. The approximation is usually the correct value, but it is too high by 1 for x = 0 and for x in the range 8 to 9, 64 to 99, 512 to 999, 8192 to 9999, and so on. The second table gives the value below which the estimate must be corrected by subtracting 1.

This scheme uses a total of 73 bytes for tables and can be coded in only six instructions on the IBM System/370 [CJS] (to achieve this, the values in `table1` must be four times the values shown). It executes in about ten instructions on a RISC that has *number of leading zeros*, but no other uncommon instructions. The other methods to be discussed are variants of this.

The first variation eliminates the conditional branch that results from the *if* statement. Actually, the program in Figure 11–10 can be coded free of branches if the machine has the *set less than unsigned* instruction, but the method to be described can be used on machines that have no unusual instructions (other than *number of leading zeros*).

The method is to replace the *if* statement with a subtraction followed by a *shift right* of 31, so that the sign bit can be subtracted from *y*. A difficulty occurs for large $x(x \geq 2^{31} + 10^9)$, which can be fixed by adding an entry to `table2`, as shown in Figure 11–11.

This executes in about 11 instructions on a RISC that has *number of leading zeros* but is otherwise quite “basic.” It can be modified to return the value 0, rather than –1, for $x = 0$ (which is preferable for the decimal conversion problem) by changing the last entry in `table1` to 1 (that is, by changing “0, 0, 0, 0” to “0, 0, 0, 1”).

```
int ilog10(unsigned x) {
   int y;
   static unsigned char table1[33] = {9, 9, 9, 8, 8, 8,
       7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
       2, 2, 2, 1, 1, 1, 0, 0, 0, 0};
   static unsigned table2[10] = { 1, 10, 100, 1000, 10000,
       100000, 1000000, 10000000, 100000000, 1000000000};

   y = table1[nlz(x)];
   if (x < table2[y]) y = y - 1;
   return y;
}
```

FIGURE 11–10. Integer log base 10 from log base 2, double table lookup.

The next variation replaces the first table lookup with a subtraction, a multiplication, and a shift. This seems likely to be possible because $\log_{10}x$ and $\log_2 x$ are related by a multiplicative constant, namely $\log_{10}2 = 0.30103....$ Thus, it may be possible to compute ilog10(x) by computing $\lfloor c \text{ ilog2}(x) \rfloor$ for some suitable $c \approx 0.30103$, and correcting the result by using a table such as `table2` in Figure 11–11.

```
int ilog10(unsigned x) {
   int y;
   static unsigned char table1[33] = {10, 9, 9, 8, 8, 8,
       7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
       2, 2, 2, 1, 1, 1, 0, 0, 0, 0};
   static unsigned table2[11] = {1, 10, 100, 1000, 10000,
       100000, 1000000, 10000000, 100000000, 1000000000,
       0};

   y = table1[nlz(x)];
   y = y - ((x - table2[y]) >> 31);
   return y;
}
```

FIGURE 11–11. Integer log base 10 from log base 2, double table lookup, branch free.

To pursue this, let $\log_{10}2 = c + \varepsilon$, where $c > 0$ is a rational approximation to $\log_{10}2$ that is a convenient multiplier, and $\varepsilon > 0$. Then, for $x \geq 1$,

$$\begin{aligned}
\text{ilog10}(x) &= \lfloor \log_{10} x \rfloor = \lfloor (c+\varepsilon)\log_2 x \rfloor \\
\lfloor c \log_2 x \rfloor \le \text{ilog10}(x) &= \lfloor c \log_2 x + \varepsilon \log_2 x \rfloor \\
\lfloor c\, \text{ilog2}(x) \rfloor \le \text{ilog10}(x) &\le \lfloor c\,(\text{ilog2}(x)+1) + \varepsilon \log_2 x \rfloor \\
&\le \lfloor c\, \text{ilog2}(x) + c + \varepsilon \log_2 x \rfloor \\
&\le \lfloor c\, \text{ilog2}(x) \rfloor + \lfloor c + \varepsilon \log_2 x \rfloor + 1.
\end{aligned}$$

Thus, if we choose c so that $c + \varepsilon\log_2 x < 1$, then $\lfloor c\ \text{ilog2}(x) \rfloor$ approximates ilog10(x) with an error of 0 or +1. Furthermore, if we take ilog2(0) = ilog10(0) = – 1, then $\lfloor c\ \text{ilog2}(0) \rfloor$ = ilog10(0) (because $0 < c \le 1$), so we need not be concerned about this case. (There are other definitions that would work here, such as ilog2(0) = ilog10(0) = 0.)

Because $\varepsilon = \log_{10}2 - c$, we must choose c so that

$$c + (\log_{10}2 - c)\log_2 x < 1, \text{ or}$$

$$c(\log_2 x - 1) > (\log_{10}2)\log_2 x - 1.$$

This is satisfied for $x = 1$ (because $c < 1$) and 2. For larger x, we must have

$$c > \frac{(\log_{10}2)\log_2 x - 1}{\log_2 x - 1}.$$

The most stringent requirement on c occurs when x is large. For a 32-bit machine, $x < 2^{32}$, so choosing

$$c > \frac{0.30103 \cdot 32 - 1}{32 - 1} \approx 0.27848$$

suffices. Because $c < 0.30103$ (because $\varepsilon > 0$), $c = 9/32 = 0.28125$ is a convenient value. Experimentation reveals that coarser values such as 5/16 and 1/4 are not adequate.

This leads to the scheme illustrated in Figure 11–12, which estimates low and then corrects by adding 1. It executes in about 11 instructions on a RISC that has *number of leading zeros*, counting the *multiply* as one instruction.

This can be made into a branch-free version, but again there is a difficulty with large $x (x > 2^{31} + 10^9)$, which can be fixed in either of two ways. One way is to use a different multiplier (19/64) and a slightly expanded table. The program is shown in Figure 11–13 (about 11 instructions on a RISC that has *number of leading zeros*, counting the *multiply* as one instruction).

The other “fix” is to *or* x into the result of the subtraction to force the sign bit to be on for $x \ge 2^{31}$; that is, change the second executable line of Figure 11–12 to

```
y = y + (((table2[y+1] - x) | x) >> 31);
```

This is the preferable program if multiplication by 19 is substantially more difficult than multiplication by 9 (as it is for a *shift*-and-*add* sequence).

```
   static unsigned table2[10] = {0, 9, 99, 999, 9999, 99999,
      999999, 9999999, 99999999, 999999999};

   y = (9*(31 - nlz(x))) >> 5;
   if (x > table2[y+1]) y = y + 1;
   return y;
```

FIGURE 11–12. Integer log base 10 from log base 2, one table lookup.

```
int ilog10(unsigned x) {
   int y;
   static unsigned table2[11] = {0, 9, 99, 999, 9999,
       99999, 999999, 9999999, 99999999, 999999999,
       0xFFFFFFFF};

   y = (19*(31 - nlz(x))) >> 6;
   y = y + ((table2[y + 1] - x) >> 31);
   return y;
}
```

FIGURE 11–13. Integer log base 10 from log base 2, one table lookup, branch free.

For a 64-bit machine, choosing

$$c > \frac{0.30103 \cdot 64 - 1}{64 - 1} \approx 0.28993$$

suffices. The value 19/64 = 0.296875 is convenient, and experimentation reveals that no coarser value is adequate. The program is (branch-free version)

```
   unsigned table2[20] = {0, 9, 99, 999, 9999, ...,
      9999999999999999999};
   y = ((19*(63 - nlz(x)) >> 6;
   y = y + ((table2[y + 1] - x) >> 63;
   return y;
```

Exercises

1. Is the correct integer fourth root of an integer x obtained by computing the integer square root of the integer square root of x? That is, does

$$\left\lfloor \sqrt{\lfloor \sqrt{x} \rfloor} \right\rfloor = \lfloor \sqrt[4]{x} \rfloor?$$

2. Code the 64-bit version of the cube root routine that is mentioned at the end of Section 11–2. Use the “long long” C data type. Do you see an alternative method for handling the overflow of `b` that probably results in a faster routine?

3. How many multiplications does it take to compute x^{23} (modulo 2^W, where W is the computer’s word size)?

4. Describe in simple terms the functions (a) $2^{\text{ilog2}(x)}$ and (b) $2^{\text{ilog2}(x-1)+1}$ for x an integer greater than 0.

Chapter 12. Unusual Bases for Number Systems

This section discusses a few unusual positional number systems. They are just interesting curiosities and are probably not practical for anything. We limit the discussion to integers, but they can all be extended to include digits after the radix point—which usually, but not always, denotes non-integers.

12–1 Base –2

By using –2 as the base, both positive and negative integers can be expressed without an explicit sign or other irregularity, such as having a negative weight for the most significant bit (Knu3). The digits used are 0 and 1, as in base +2; that is, the value represented by a string of 1's and 0's is understood to be

$$(a_n \ldots a_3 a_2 a_1 a_0) = a_n(-2)^n + \ldots + a_3(-2)^3 + a_2(-2)^2 + a_1(-2) + a_0.$$

From this, it can be seen that a procedure for finding the base −2, or "negabinary," representation of an integer is to successively divide the number by −2, recording the remainders. The division must be such that it always gives a remainder of 0 or 1 (the digits to be used); that is, it must be modulus division. As an example, the plan below shows how to find the base −2 representation of −3.

$$\frac{-3}{-2} = 2 \text{ rem } 1$$

$$\frac{2}{-2} = -1 \text{ rem } 0$$

$$\frac{-1}{-2} = 1 \text{ rem } 1$$

$$\frac{1}{-2} = 0 \text{ rem } 1$$

Because we have reached a 0 quotient, the process terminates (if continued, the remaining quotients and remainders would all be 0). Thus, reading the remainders upward, we see that –3 is written 1101 in base –2.

Table 12–1 shows, on the left, how each bit pattern from 0000 to 1111 is interpreted in base –2, and on the right, how integers in the range –15 to +15 are represented.

TABLE 12–1. CONVERSIONS BETWEEN DECIMAL AND BASE–2

n (base –2)	n (decimal)	n (decimal)	n (base –2)	$-n$ (base –2)
0	0	0	0	0
1	1	1	1	11
10	–2	2	110	10
11	–1	3	111	1101
100	4	4	100	1100
101	5	5	101	1111
110	2	6	11010	1110
111	3	7	11011	1001
1000	–8	8	11000	1000
1001	–7	9	11001	1011
1010	–10	10	11110	1010
1011	–9	11	11111	110101
1100	–4	12	11100	110100
1101	–3	13	11101	110111
1110	–6	14	10010	110110
1111	–5	15	10011	110001

It is not obvious that the 2^n possible bit patterns in an n-bit word uniquely represent all integers in a certain range, but this can be shown by induction. The inductive hypothesis is that an n-bit word represents all integers in the range

$$-(2^{n+1}-2)/3 \text{ to } (2^n-1)/3 \text{ for } n \text{ even, and} \tag{1a}$$

$$(-(2^n-2)/3) \text{ to } ((2^{n+1}-1)/3) \text{ for } n \text{ odd.} \tag{1b}$$

Assume first that n is even. For $n = 2$, the representable integers are 10, 11, 00, and 01 in base –2, or

–2, –1, 0, 1.

This agrees with (1a), and each integer in the range is represented once and only once.

A word of $n + 1$ bits can, with a leading bit of 0, represent all the integers given by

(1a). In addition, with a leading bit of 1, it can represent all these integers biased by $(-2)^n = 2^n$. The new range is

$$2^n - (2^{n+1} - 2) / 3 \text{ to } 2^n + (2^n - 1)/3,$$

or

$$(2^n - 1)/3 + 1 \text{ to } (2^{n+2} - 1)/3.$$

This is contiguous to the range given by (1a), so for a word size of $n + 1$ bits, all integers in the range

$$-(2^{n+1} - 2)/3 \text{ to } (2^{n+2} - 1)/3$$

are represented once and only once. This agrees with (1b), with n replaced by $n + 1$.

The proof that (1a) follows from (1b), for n odd, and that all integers in the range are uniquely represented, is similar.

To add and subtract, the usual rules, such as 0 + 1 = 1 and 1 – 1 = 0, of course apply. Because 2 is written 110, and –1 is written 11, and so on, the following additional rules apply. These, together with the obvious ones, suffice.

$$\begin{aligned} 1 + 1 &= 110 \\ 11 + 1 &= 0 \\ 1 + 1 + 1 &= 111 \\ 0 - 1 &= 11 \\ 11 - 1 &= 10 \end{aligned}$$

When adding or subtracting, there are sometimes two carry bits. The carry bits are to be *added* to their column, even when subtracting. It is convenient to place them both over the next bit to the left and simplify (when possible) using 11 + 1 = 0. If 11 is carried to a column that contains two 0's, bring down a 1 and carry a 1. Below are examples.

```
          Addition                          Subtraction
   11 1 11     11                  1 11  1     1
        1  0  1  1  1     19             1  0  1  0  1     21
   + 1 1  0  1  0  1 +(-11)           -1  0  1  1  1  0  -(-38)
  ----------------- ------            ---------------   -----
   0  1  1  0  0  0      8         1  0  0  1  1  1  1     59
```

The only carries possible are 0, 1, and 11. Overflow occurs if there is a carry (either 1 or 11) out of the high-order position. These remarks apply to both addition and subtraction.

Because there are three possibilities for the carry, a base –2 adder would be more complex than a two's-complement adder.

There are two ways to negate an integer. It can be added to itself shifted left one position (that is, multiply by –1), or it can be subtracted from 0. There is no rule as simple and convenient as the "complement and add 1" rule of two's-complement arithmetic. In two's-complement, this rule is used to build a subtracter from an adder (to compute $A - B$, form $A + \bar{B} + 1$).

For base –2, there is no device quite that simple, but a method that is nearly as simple is to complement the minuend (meaning to invert each bit), add the complemented minuend to the subtrahend, and then complement the sum [Lang]. Here is an example showing the subtraction of 13 from 6 using this scheme on an eight-bit machine.

```
00011010   6
00011101   13
11100101   6 complemented
--------
11110110   (6 complemented) + 13
00001001   Complement of the sum (-7)
```

This method is using

$$A - B = I - ((I - A) + B)$$

in base –2 arithmetic, with I a word of all 1's.

Multiplication of base –2 integers is straightforward. Just use the rule that 1 × 1 = 1 and 0 times either 0 or 1 is 0, and add the columns using base –2 addition.

Division, however, is quite complicated. It is a real challenge to devise a reasonable hardware division algorithm—that is, one based on repeated subtraction and shifting. Figure 12–1 shows an algorithm that is expressed, for definiteness, for an 8-bit machine. It does modulus division (nonnegative remainder).

Although this program is written in C and was tested on a binary two's-complement machine, that is immaterial—it should be viewed somewhat abstractly. The input quantities `n` and `d`, and all internal variables except for `q`, are simply numbers without any particular representation. The output `q` is a string of bits to be interpreted in base –2.

This requires a little explanation. If the input quantities were in base –2, the algorithm would be very awkward to express in an executable form. For example, the test "`if (d > 0)`" would have to test that the most significant bit of `d` is in an even position. The addition in "`c = c + d`" would have to be a base –2 addition. The code would be very hard to read. The way the algorithm is coded, you should think of `n` and `d` as numbers without any particular representation. The code shows the arithmetic operations to be performed, whatever encoding is used. If the numbers are encoded in base –2, as they would be in hardware that implements this algorithm, the multiplication by –128 is a left shift of seven positions, and the divisions by –2 are right shifts of one position.

As examples, the code computes values as follows:

divbm2(6, 2) = 7 (six divided by two is 111_{-2})

divbm2(– 4, 3) = 2 (minus four divided by three is 10_{-2})

divbm2(–4, –3) = 6 (minus four divided by minus 3 is 110_{-2})

```
int divbm2(int n, int d) {          // q = n/d in base -2.
   int r, dw, c, q, i;

   r = n;                          // Init. remainder.
   dw = (-128)*d;                  // Position d.
   c = (-43)*d;                    // Init. comparand.
   if (d > 0) c = c + d;
   q = 0;                           // Init. quotient.
   for (i = 7; i >= 0; i--) {
      if (d > 0 ^ (i&1) == 0 ^ r >= c {
         q = q | (1 << i);          // Set a quotient bit.
         r = r - dw;                // Subtract d shifted.
      }
      dw = dw/(-2);                 // Position d.
      if (d > 0) c = c -2*d;        // Set comparand for
      else c = c + d;               // next iteration.
      c = c/(-2);
   }
   return q;                        // Return quotient in
                                    // base -2.
                                    // Remainder is r,
}                                   // 0 <= r < |d|.
```

FIGURE 12-1. Division in base –2.

The step `q = q | (1 << i);` represents simply setting bit *i* of `q`. The next line—`r = r - dw`—represents reducing the remainder by the divisor `d` shifted left.

The algorithm is difficult to describe in detail, but we will try to give the general idea.

Consider determining the value of the first bit of the quotient, bit 7 of `q`. In base −2, 8-bit numbers that have their most significant bit "on" range in value from −170 to −43. Therefore, ignoring the possibility of overflow, the first (most significant) quotient bit will be 1 if (and only if) the quotient will be algebraically less than or equal to –43.

Because $n = qd + r$ and for a positive divisor $r \le d - 1$, for a positive divisor the first quotient bit will be 1 iff $n \le -43d + (d - 1)$, or $n < -43d + d$. For a negative divisor, the first quotient bit will be 1 iff $n \ge -43d$ ($r \ge 0$ for modulus division).

Thus, the first quotient bit is 1 iff

$$(d > 0 \;\&\; \neg(n \ge -43d + d)) \mid (d < 0 \;\&\; n \ge -43d).$$

Ignoring the possibility that $d = 0$, this can be written as

$$d{>}0 \quad n \ge c,$$

where $c = -43d + d$ if $d \ge 0$, and $c = -43d$ if $d < 0$.

This is the logic for determining a quotient bit for an odd-numbered bit position. For an even-numbered position, the logic is reversed. Hence, the test includes the term `(i&1) == 0`. (The `^` character in the program denotes *exclusive or*.)

At each iteration, `c` is set equal to the smallest (closest to zero) integer that must have a 1-bit at position `i` after dividing by `d`. If the current remainder `r` exceeds that, then bit `i` of `q` is set to 1 and `r` is adjusted by subtracting the value of a 1 at that position, multiplied by the divisor `d`. No real multiplication is required here; `d` is simply positioned properly and subtracted.

The algorithm is not elegant. It is awkward to implement because there are several additions, subtractions, and comparisons, and there is even a multiplication (by a constant) that must be done at the beginning. One might hope for a "uniform" algorithm—one that does not test the signs of the arguments and do different things depending on the outcome. Such a uniform algorithm, however, probably does not exist for base –2 (or for two's-complement arithmetic). The reason for this is that division is inherently a non-uniform process. Consider the simplest algorithm of the *shift-*

and-*subtract* type. This algorithm would not shift at all, but for positive arguments would simply subtract the divisor from the dividend repeatedly, counting the number of subtractions performed until the remainder is less than the divisor. On the other hand, if the dividend is negative (and the divisor is positive), the process is to add the divisor repeatedly until the remainder is 0 or positive, and the quotient is the negative of the count obtained. The process is still different if the divisor is negative.

In spite of this, division *is* a uniform process for the signed-magnitude representation of numbers. With such a representation, the magnitudes are positive, so the algorithm can simply subtract magnitudes and count until the remainder is negative, and then set the sign bit of the quotient to the *exclusive or* of the arguments, and the sign bit of the remainder equal to the sign of the dividend (this gives ordinary truncating division).

The algorithm given above could be made more uniform, in a sense, by first complementing the divisor, if it is negative, and then performing the steps given as simplified by having $d > 0$. Then a correction would be performed at the end. For modulus division, the correction is to negate the quotient and leave the remainder unchanged. This moves some of the tests out of the loop, but the algorithm as a whole is still not pretty.

It is interesting to contrast the commonly used number representations and base –2 regarding the question of whether or not the computer hardware treats numbers uniformly in carrying out the four fundamental arithmetic operations. We don't have a precise definition of "uniformly," but basically it means free of operations that might or might not be done, depending on the signs of the arguments. We consider setting the sign bit of the result equal to the *exclusive or* of the signs of the arguments to be a uniform operation. Table 12–2 shows which operations treat their operands uniformly with various number representations.

One's-complement addition and subtraction are done uniformly by means of the "end around carry" trick. For addition, all bits, including the sign bit, are added in the usual binary way, and the carry out of the leftmost bit (the sign bit) is added to the least significant position. This process always terminates right away (that is, the addition of the carry cannot generate another carry out of the sign bit position).

TABLE 12–2. UNIFORM OPERATIONS IN VARIOUS NUMBER ENCODINGS

	Signed-magnitude	One's-complement	Two's-complement	Base –2
addition	no	yes	yes	yes
subtraction	no	yes	yes	yes
multiplication	yes	no	no	yes
division	yes	no	no	no

In the case of two's-complement multiplication, the entry is "yes" if only the right half of the doubleword product is desired.

We conclude this discussion of the base –2 number system with some observations about how to convert between straight binary and base –2.

To convert to binary from base –2, form a word that has only the bits with positive weight, and subtract a word that has only the bits with negative weight, using the subtraction rules of binary arithmetic. An alternative method that may be a little simpler is to extract the bits appearing in the negative weight positions, shift them one position to the left, and subtract the extracted number from the original number using the subtraction rules of ordinary binary arithmetic.

To convert to base –2 from binary, extract the bits appearing in the odd positions (positions weighted by 2^n with n odd), shift them one position to the left, and add the two numbers using the addition rules of base –2. Here are two examples:

```
     Binary from base -2                  Base -2 from binary

     110111 (-13)                         110111 (55)
   - 1 0 1   (binary subtract)          + 1 0 1   (base -2 add)
   ---------                            ---------
...111110011 (-13)                       1001011 (55)
```

On a computer, with its fixed word size, these conversions work for negative numbers if the carries out of the high-order position are simply discarded. To illustrate, the example on the right above can be regarded as converting −9 to base −2 from binary if the word size is six bits.

The above algorithm for converting to base −2 cannot easily be implemented in software on a binary computer, because it requires doing addition in base −2. Schroeppel [HAK, item 128] overcomes this with a much more clever and useful way to do the conversions in both directions. To convert to binary, his method is

$$B \leftarrow (N \quad \mathbf{0b10...\ 1010) - 0b10\ ...\ 1010}.$$

To see why this works, let the base –2 number consist of the four digits *abcd*. Then, interpreted (erroneously) in straight binary, this is $8a + 4b + 2c + d$. After the *exclusive or*, interpreted in binary it is $8(1 - a) + 4b + 2(1 - c) + d$. After the (binary) subtraction of 8 + 2, it is $-8a + 4b - 2c + d$, which is its value interpreted in base –2.

Schroeppel's formula can be readily solved for N in terms of B, so it gives a three-instruction method for converting in the other direction. Collecting these results, we have the following formulas for converting to binary for a 32-bit machine:

$$B \leftarrow \mathbf{(}N \mathbf{\ \&\ 0x55555555) - (}N \mathbf{\ \&\ \neg 0x55555555)},$$
$$B \leftarrow N \mathbf{- ((}N\mathbf{\&\ 0xAAAAAAAA)} << \mathbf{1)},$$
$$B \leftarrow \mathbf{(}N \quad \mathbf{0xAAAAAAAA) - 0xAAAAAAAA},$$

and the following, for converting to base –2 from binary:

$$N \leftarrow \mathbf{(}B \mathbf{\ +\ 0xAAAAAAAA)} \quad \mathbf{0xAAAAAAAA}.$$

12-2 Base –1 + *i*

By using $-1 + i$ as the base, where i is $\sqrt{-1}$, all *complex* integers (complex numbers with integral real and imaginary parts) can be expressed as a single "number" without an explicit sign or other irregularity. Surprisingly, this can be done using only 0 and 1 for digits, and all integers are represented uniquely. We will not prove this or much else about this number system, but will just describe it very briefly.

It is not entirely trivial to discover how to write the integer 2.[1] But it can be determined algorithmically by successively dividing 2 by the base and recording the remainders. What does a “remainder” mean in this context? We want the remainder after dividing by $-1 + i$ to be 0 or 1, if possible (so that the digits will be 0 or 1). To see that it is always possible, assume that we are to divide an arbitrary complex integer $a + bi$ by $-1 + i$. Then, we wish to find q and r such that q is a complex integer, $r = 0$ or 1, and

$$a + bi = (q_r + q_i i)(-1 + i) + r,$$

where q_r and q_t denote the real and imaginary parts of q, respectively. Equating real and imaginary parts and solving the two simultaneous equations for q gives

$$q_r = \frac{b - a + r}{2}, \text{ and}$$

$$q_i = \frac{-a - b + r}{2}.$$

Clearly, if a and b are both even or are both odd, then by choosing $r = 0$, q is a complex integer. Furthermore, if one of a and b is even and the other is odd, then by choosing $r = 1$, q is a complex integer.

Thus, the integer 2 can be converted to base $-1 + i$ by the plan illustrated below.

Because the real and imaginary parts of the integer 2 are both even, we simply do the division, knowing that the remainder will be 0:

$$\frac{2}{-1+i} = \frac{2(-1-i)}{(-1+i)(-1-i)} = -1 - i \text{ rem } 0.$$

Because the real and imaginary parts of $-1 - i$ are both odd, again we simply divide, knowing that the remainder is 0:

$$\frac{-1-i}{-1+i} = \frac{(-1-i)(-1-i)}{(-1+i)(-1-i)} = i \text{ rem } 0.$$

Because the real and imaginary parts of i are even and odd, respectively, the remainder will be 1. It is simplest to account for this at the beginning by subtracting 1 from the dividend.

$$\frac{i-1}{-1+i} = 1 \text{ (remainder is 1).}$$

Because the real and imaginary parts of 1 are odd and even, the next remainder will be 1. Subtracting this from the dividend gives

$$\frac{1-1}{-1+i} = 0 \text{ (remainder is 1).}$$

Because we have reached a 0 quotient, the process terminates, and the base $-1 + i$ representation for 2 is seen to be 1100 (reading the remainders upward).

Table 12–3 shows how each bit pattern from 0000 to 1111 is interpreted in base $-1 + i$ and how the real integers in the range –15 to +15 are represented.

The addition rules for base $-1 + i$ (in addition to the trivial ones involving a 0-bit) are as follows:

$$\begin{aligned}1+1 &= 1100\\ 1+1+1 &= 1101\\ 1+1+1+1 &= 111010000\\ 1+1+1+1+1 &= 111010001\\ 1+1+1+1+1+1 &= 111011100\\ 1+1+1+1+1+1+1 &= 111011101\\ 1+1+1+1+1+1+1+1 &= 111000000\end{aligned}$$

TABLE 12–3. CONVERSIONS BETWEEN DECIMAL AND BASE −1 + *i*

n (base $-1+i$)	n (decimal)	n (decimal)	n (base $-1+i$)	$-n$ (base $-1+i$)
0	0	0	0	0
1	1	1	1	11101
10	$-1+i$	2	1100	11100
11	i	3	1101	10001
100	$-2i$	4	111010000	10000
101	$1-2i$	5	111010001	11001101
110	$-1-i$	6	111011100	11001100
111	$-i$	7	111011101	11000001
1000	$2+2i$	8	111000000	11000000
1001	$3+2i$	9	111000001	11011101
1010	$1+3i$	10	111001100	11011100
1011	$2+3i$	11	111001101	11010001
1100	2	12	100010000	11010000
1101	3	13	100010001	1110100001101
1110	$1+i$	14	100011100	1110100001100
1111	$2+i$	15	100011101	1110100000001

When adding two numbers, the largest number of carries that occurs in one column is six, so the largest sum of a column is 8 (111000000). This makes for a rather complicated adder. If one were to build a complex arithmetic machine, it would no doubt be best to keep the real and imaginary parts separate,[2] with each represented in some sensible way such as two's-complement.

12–3 Other Bases

The base $-1-i$ has essentially the same properties as the base $-1+i$ discussed above. If a certain bit pattern represents the number $a+bi$ in one of these bases, then the same bit pattern represents the number $a-bi$ in the other base.

The bases $1+i$ and $1-i$ can also represent all the complex integers, using only 0 and 1 for digits. These two bases have the same complex-conjugate relationship to each other, as do the bases $-1 \pm i$. In bases $1 \pm i$, the representation of some integers has an infinite string of 1's on the left, similar to the two's-complement representation of negative integers. This arises naturally by using uniform rules for addition and subtraction, as in the case of two's-complement. One such integer is 2, which (in either base) is written ...11101100. Thus, these bases have the rather complex addition rule 1 + 1 = ...11101100.

By grouping into pairs the bits in the base –2 representation of an integer, one obtains a base 4 representation for the positive and negative numbers, using the digits –2, –1, 0, and 1. For example,

$$-14_{\text{decimal}} = 110110_{-2} = (-1)(1)(-2)_4 = -1 \cdot 4^2 + 1 \cdot -4^1 - 2 \cdot 4^0$$

Similarly, by grouping into pairs the bits in the base $-1+i$ representation of a complex integer, we obtain a base $-2i$ representation for the complex integers using the digits 0, 1, $-1+i$, and i. This is a bit too complicated to be interesting.

The "quater-imaginary" system (Knu2) is similar. It represents the complex integers using $2i$ as a base, and the digits 0, 1, 2, and 3 (with no sign). To represent some integers, namely those with an odd imaginary component, it is necessary to use a digit to the right of the radix point. For example, i is written 10.2 in base $2i$.

12–4 What Is the Most Efficient Base?

Suppose you are building a computer and you are trying to decide what base to use to represent integers. For the registers you have available circuits that are 2-state (binary), 3-state, 4-state, and so on. Which should you use?

Let us assume that the cost of a b-state circuit is proportional to b. Thus, a 3-state circuit costs 50% more than a binary circuit, a 4-state circuit costs twice as much as a binary circuit, and so on.

Suppose you want the registers to be able to hold integers from 0 to some maximum M. Encoding integers from 0 to M in base b requires $\lceil \log_b(M+1) \rceil$ digits (e.g., to represent all integers from 0 to 999,999 in decimal requires $\log_{10}(1{,}000{,}000) = 6$ digits).

One would expect the cost of a register to be equal to the product of the number of digits required times the cost to represent each digit:

$$c = k \log_b(M+1) \cdot b,$$

where c is the cost of a register and k is a constant of proportionality. For a given M, we wish to find b that minimizes the cost.

The minimum of this function occurs for that value of b that makes $dc/db = 0$. Thus, we have

$$\frac{d}{db}(kb\log_b(M+1)) = \frac{d}{db}\left(kb\frac{\ln(M+1)}{\ln b}\right) = k\ln(M+1)\frac{\ln b - 1}{(\ln b)^2}.$$

This is zero when 1nb = 1, or $b = e$.

This is not a very satisfactory result. Because $e \approx 2.718$, 2 and 3 must be the most efficient integral bases. Which is more efficient? The ratio of the cost of a base 2 register to the cost of a base 3 register is

$$\frac{c(2)}{c(3)} = \frac{k \cdot 2\log_2(M+1)}{k \cdot 3\log_3(M+1)} = \frac{2\ln(M+1)/(\ln 2)}{3\ln(M+1)/(\ln 3)} = \frac{2\ln 3}{3\ln 2} \approx 1.056.$$

Thus, base 2 is more costly than base 3, but only by a small amount.

By the same analysis, base 2 is more costly than base e by a factor of about 1.062.

Exercises

1. Schroeppel's formula for converting from base –2 to binary has a dual involving the constant 0x5555555. Can you find it?

2. Show how to add 1 to a base –2 number using the arithmetic and logical operations of a binary computer. For example, 0b111   0b100.

3. Show how to round a base –2 number down (in the negative direction) to a multiple of 16 using the arithmetic and logical operations of a binary computer. For example, 0b10   0b110000.

4. Write a program, in a language of your choice, to convert a base $-1 + i$ integer to the form $a + bi$, where a and b are real integers. For example, if you give the program the integer 33, or 0x21, it should display something like 5 − 4i.

5. How would you convert a number in base $-1 + i$ to its negative? Extract its real part? Extract its imaginary part? Convert it to its complex conjugate? (The complex conjugate of $a + bi$ is $a - bi$.)

Chapter 13. Gray Code

13–1 Gray Code

Is it possible to cycle through all 2^n combinations of *n* bits by changing only one bit at a time? The answer is "yes," and this is the defining property of Gray codes. That is, a Gray code is an encoding of the integers such that a Gray-coded integer and its successor differ in only one bit position. This concept can be generalized to apply to any base, such as decimal, but here we will discuss only binary Gray codes.

Although there are many binary Gray codes, we will discuss only one: the "reflected binary Gray code." This code is what is usually meant in the literature by the unqualified term "Gray code." We will show, usually without proof, how to do some basic operations in this representation of integers, and we will show a few surprising properties.

The reflected binary Gray code is constructed as follows. Start with the strings 0 and 1, representing the integers 0 and 1:

```
0
1
```

Reflect this about a horizontal axis at the bottom of the list, and place a 1 to the left of the new list entries and a 0 to the left of the original list entries:

```
00
01
11
10
```

This is the reflected binary Gray code for $n = 2$. To get the code for $n = 3$, reflect this and attach a 0 or 1 as before:

```
000
001
011
010
110
111
101
100
```

From this construction, it is easy to see by induction on *n* that (1) each of the 2^n bit combinations appears once and only once in the list, (2) only one bit changes in going from one list entry to the next, and (3) only one bit changes when cycling around from the last entry to the first. Gray codes having this last property are called "cyclic," and the reflected binary Gray code is necessarily cyclic.

If $n > 2$, there are non-cyclic codes that take on all 2^n values once and only once. One such code is 000 001 011 010 110 100 101 111.

Figure 13–1 shows, for $n = 4$, the integers encoded in ordinary binary and in Gray code. The formulas show how to convert from one representation to the other at the bit-by-bit level (as it would be done in hardware).

Binary *abcd*	Gray *efgh*		
0000	0000	Gray from Binary	Binary from Gray
0001	0001	$e = a$	$a = e$
0010	0011	$f = a \oplus b$	$b = e \oplus f$
0011	0010	$g = b \oplus c$	$c = e \oplus f \oplus g$
0100	0110	$h = c \oplus d$	$d = e \oplus f \oplus g \oplus h$
0101	0111		
0110	0101		
0111	0100		
1000	1100		
1001	1101		
1010	1111		
1011	1110		
1100	1010		
1101	1011		
1110	1001		
1111	1000		

FIGURE 13–1. 4-bit Gray code and conversion formulas.

As for the number of Gray codes on n bits, notice that one still has a cyclic binary Gray code after rotating the list (starting at any of the 2^n positions and cycling around) or reordering the columns. Any combination of these operations results in a distinct code. Therefore, there are at least $2^n \cdot n!$ cyclic binary Gray codes on n bits. There are more than this for $n \geq 3$.

The Gray code and binary representations have the following dual relationships, evident from the formulas given in Figure 13–1:

- Bit i of a Gray-coded integer is the parity of bit i and the bit to the left of i in the corresponding binary integer (using 0 if there is no bit to the left of i).
- Bit i of a binary integer is the parity of all the bits at and to the left of position i in the corresponding Gray-coded integer.

Converting to Gray from binary can be done in only two instructions:

$$G \leftarrow B \oplus (B \overset{u}{\gg} 1).$$

The conversion to binary from Gray is harder. One method is given by

$$B \leftarrow \bigoplus_{i=0}^{n-1} G \overset{u}{\gg} i.$$

We have already seen this formula in “Computing the Parity of a Word” on page 96. As mentioned there, this formula can be evaluated as illustrated below for $n = 32$.

```
B = G ^ (G >> 1);
B = B ^ (B >> 2);
B = B ^ (B >> 4);
B = B ^ (B >> 8);
B = B ^ (B >> 16);
```

Thus, in general it requires $2 \cdot \lceil \log_2 n \rceil$ instructions.

Because it is so easy to convert from binary to Gray, it is trivial to generate successive Gray-coded integers:

```
for (i = 0; i < n; i++) {
   G = i ^ (i >> 1);
   output G;
}
```

13-2 Incrementing a Gray-Coded Integer

The logic for incrementing a 4-bit *binary* integer *abcd* can be expressed as follows, using Boolean algebra notation:

$$\begin{aligned} d' &= \bar{d} \\ c' &= c \oplus d \\ b' &= b \oplus cd \\ a' &= a \oplus bcd \end{aligned}$$

Thus, one way to build a Gray-coded counter in hardware is to build a binary counter using the above logic and convert the outputs *a′*, *b′*, *c′*, *d′* to Gray by forming the *exclusive or* of adjacent bits, as shown under “Gray from Binary” in Figure 13–1.

A way that might be slightly better is described by the following formulas:

$$\begin{aligned} p &= e \oplus f \oplus g \oplus h \\ h' &= h \oplus \bar{p} \\ g' &= g \oplus hp \\ f' &= f \oplus g\bar{h}p \\ e' &= e \oplus f\bar{g}\bar{h}p \end{aligned}$$

That is, the general case is

$$G_n' = G_n \oplus (G_{n-1}\bar{G}_{n-2}\ldots\bar{G}_0 p), \quad n \ge 2.$$

Because the parity p alternates between 0 and 1, a counter circuit might maintain p in a separate 1-bit register and simply invert it on each count.

In software, the best way to find the successor *G′* of a Gray-coded integer *G* is probably simply to convert *G* to binary, increment the binary word, and convert it back to Gray code. Another way that's interesting and almost as good is to determine which bit to flip in *G*. The pattern goes like this, expressed as a word to be *exclusive or*'d to *G*:

1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16

The alert reader will recognize this as a mask that identifies the position of the leftmost bit that changes when incrementing the integer 0, 1, 2, 3, ..., corresponding to the positions in the above list. Thus, to increment a Gray-coded integer *G*, the bit position to invert is given by the leftmost bit that changes when 1 is added to the binary integer corresponding to *G*.

This leads to the algorithms for incrementing a Gray-coded integer *G* as shown in Figure 13–2. They both first convert *G* to binary, which is shown as `index(G)`.

```
B = index(G);                            B = index(G);
B = B + 1;                               M = ~B & (B + 1);
Gp = B ^ (B >> 1);                       Gp = G ^ M;
```

FIGURE 13–2. Incrementing a Gray-coded integer.

A pencil-and-paper method of incrementing a Gray-coded integer is as follows:

> Starting from the right, find the first place at which the parity of bits at and to the left of the position is even. Invert the bit at this position.

Or, equivalently:

> Let *p* be the parity of the word *G*. If *p* is even, invert the rightmost bit.
>
> If *p* is odd, invert the bit to the left of the rightmost 1-bit.

The latter rule is directly expressed in the Boolean equations given above.

13–3 Negabinary Gray Code

If you write the integers in order in base –2 and convert them using the "shift and exclusive or" that converts to Gray from straight binary, you get a Gray code. The 3-bit Gray code has indexes that range over the 3-bit base –2 numbers, namely –2 to 5. Similarly, the 4-bit Gray code corresponding to 4-bit base –2 numbers has indexes ranging from –10 to 5. It is not a reflected Gray code, but it almost is. The 4-bit negabinary Gray code can be generated by starting with 0 and 1, reflecting this about a horizontal axis at the *top* of the list, and then reflecting it about a horizontal axis at the *bottom* of the list, and so on. It is cyclic.

To convert back to base –2 from this Gray code, the rules are, of course, the same as they are for converting to straight binary from ordinary reflected binary Gray code (because these operations are inverses, no matter what the interpretation of the bit strings is).

13–4 Brief History and Applications

Gray codes are named after Frank Gray, a physicist at Bell Telephone Laboratories, who in the 1930s invented the method we now use for broadcasting color TV in a way that's compatible with the black-and-white transmission and reception methods then in existence; that is, when the color signal is received by a black-and-white set, the picture appears in shades of gray.

Martin Gardner [Gard] discusses applications of Gray codes involving the Chinese ring puzzle, the Tower of Hanoi puzzle, and Hamiltonian paths through graphs that represent hypercubes. He also shows how to convert from the decimal representation of an integer to a decimal Gray code representation.

Gray codes are used in position sensors. A strip of material is made with conducting and nonconducting areas, corresponding to the 1's and 0's of a Gray-coded integer. Each column has a conducting wire brush positioned to read it out. If a brush is positioned on the dividing line between two of the quantized positions so that its reading is ambiguous, then it doesn't matter which way the ambiguity is resolved. There can be only one ambiguous brush, and interpreting it as a 0 or 1 gives a position adjacent to the dividing line.

The strip can instead be a series of concentric circular tracks, giving a rotational position sensor. For this application, the Gray code must be cyclic. Such a sensor is shown in Figure 13–3, where the four dots represent the brushes.

It is possible to construct cyclic Gray codes for rotational sensors that require only one ring of conducting and nonconducting areas, although at some expense in resolution for a given number of brushes. The brushes are spaced around the ring rather than on a radial line. These codes are called *single track Gray codes*, or STGCs.

The idea is to find a code for which, when written out as in Figure 13–1, every column is a rotation of the first column (and that is cyclic, assuming the code is for a rotational device). The reflected Gray code for n = 2 is trivially an STGC. STGCs for n = 2 through 4 are shown here.

$n = 2$	$n = 3$	$n = 4$
00	000	0000
01	001	0001
11	011	0011
10	111	0111
	110	1111
	100	1110
		1100
		1000

STGCs allow the construction of more compact rotational position sensors. A rotational STGC device for n = 3 is shown in Figure 13–4.

These are all very similar, simple, and rather uninteresting patterns. Following these patterns, an STGC for the case n = 5 would have ten code words, giving a resolution of 36 degrees. It is possible to do much better. Figure 13–5 shows an STGC for n = 5 with 30 code words, giving a resolution of 12 degrees. It is close to the optimum of 32 code words.

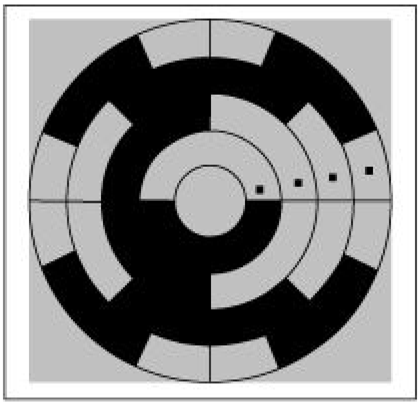


FIGURE 13–3. Rotational position sensor.

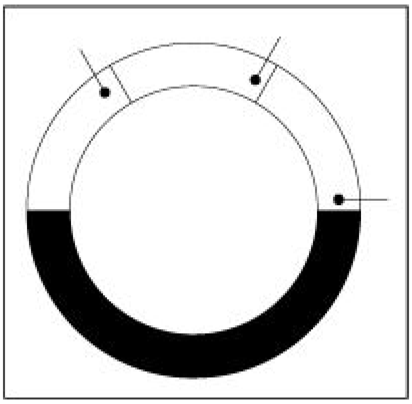


FIGURE 13–4. Single track rotational position sensor.

10000	01000	00100	00010	00001
10100	01010	00101	10010	01001
11100	01110	00111	10011	11001
11110	01111	10111	11011	11101
11010	01101	10110	01011	10101
11000	01100	00110	00011	10001

FIGURE 13–5. An STGC for n = 5.

All the STGCs in this section above are the best possible, in the sense that for $n = 2$ through 5, the largest number of code words possible is 4, 6, 8, and 30.

An STGC has been constructed with exactly 360 code words, with $n = 9$ (the smallest possible value of n, because any code for $n = 8$ has at most 256 code words) [HilPat].

Exercises

1. Show that if an integer x is even, then G(x) (the reflected binary Gray code of x) has an even number of 1-bits, and if x is odd, G(x) has an odd number of 1-bits.

2. A *balanced* Gray code is a cyclic Gray code in which the number of bit changes is the same in all columns, as one cycles around the code.

(a) Show that an STGC is necessarily balanced.

(b) Can you find a balanced Gray code for $n = 3$ that has eight code words?

3. Devise a cyclic Gray code that encodes the integers from 0 to 9.

4. [Knu6] Given a number in prime decomposed form, show how to list all its divisors in such a way that each divisor in the list is derived from the previous divisor by a single multiplication or division by a prime.

Chapter 14. Cyclic Redundancy Check

14–1 Introduction

The cyclic redundancy check, or CRC, is a technique for detecting errors in digital data, but not for making corrections when errors are detected. It is used primarily in data transmission. In the CRC method, a certain number of check bits, often called a checksum, or a hash code, are appended to the message being transmitted. The receiver can determine whether or not the check bits agree with the data to ascertain with a certain degree of probability that an error occurred in transmission. If an error occurred, the receiver sends a “negative acknowledgment” (NAK) back to the sender, requesting that the message be retransmitted.

The technique is also sometimes applied to data storage devices, such as a disk drive. In this situation each block on the disk would have check bits, and the hardware might automatically initiate a reread of the block when an error is detected, or it might report the error to software.

The material that follows speaks in terms of a “sender” and a “receiver” of a “message,” but it should be understood that it applies to storage writing and reading as well.

Section 14–2 describes the theory behind the CRC methodology. Section 14–3 shows how the theory is put into practice in hardware, and gives a software implementation of a popular method known as CRC-32.

Background

There are several techniques for generating check bits that can be added to a message. Perhaps the simplest is to append a single bit, called the “parity bit,” which makes the total number of 1-bits in the *code vector* (message with parity bit appended) even (or odd). If a single bit gets altered in transmission, this will change the parity from even to odd (or the reverse). The sender generates the parity bit by simply summing the message bits modulo 2—that is, by *exclusive or*'ing them together. It then appends the parity bit (or its complement) to the message. The receiver can check the message by summing all the message bits modulo 2 and checking that the sum agrees with the parity bit. Equivalently, the receiver can sum all the bits (message and parity) and check that the result is 0 (if even parity is being used).

This simple parity technique is often said to detect 1-bit errors. Actually, it detects errors in any odd number of bits (including the parity bit), but it is a small comfort to know you are detecting 3-bit errors if you are missing 2-bit errors.

For bit serial sending and receiving, the hardware required to generate and check a single parity bit is very simple. It consists of a single *exclusive or* gate together with some control circuitry. For bit parallel transmission, an *exclusive or* tree may be used, as illustrated in Figure 14–1. Efficient ways to compute the parity bit in software are given in Section 5–2 on page 96.

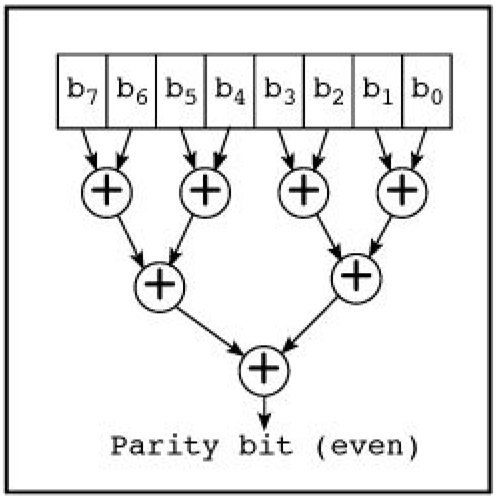


FIGURE 14–1. *Exclusive or* **tree.**

Other techniques for computing a checksum are to form the *exclusive or* of all the bytes in the message, or to compute a sum with end-around carry of all the bytes. In the latter method, the carry from each 8-bit sum is added into the least significant bit of the accumulator. It is believed that this is more likely to detect errors than the simple *exclusive or*, or the sum of the bytes with carry discarded.

A technique that is believed to be quite good in terms of error detection, and which is easy to implement in hardware, is the cyclic redundancy check. This is another way to compute a checksum, usually eight, 16, or 32 bits in length, that is appended to the message. We will briefly review the theory, show how the theory is implemented in hardware, and then give software for a commonly used 32-bit CRC checksum.

We should mention that there are much more sophisticated ways to compute a checksum, or hash code, for data. Examples are the hash functions known as MD5 and SHA-1, whose hash codes are 128 and 160 bits in length, respectively. These methods are used mainly in cryptographic applications and are substantially more difficult to implement, in hardware and software, than the CRC methodology described here. However, SHA-1 is used in certain revision control systems (Git and others) as simply a check on data integrity.

14–2 Theory

The CRC is based on polynomial arithmetic, in particular, on computing the remainder when dividing one polynomial in GF(2) (Galois field with two elements) by another. It is a little like treating the message as a very large binary number, and computing the remainder when dividing it by a fairly large prime such as $2^{32} - 5$. Intuitively, one would expect this to give a reliable checksum.

A polynomial in GF(2) is a polynomial in a single variable x whose coefficients are 0 or 1. Addition and subtraction are done modulo 2—that is, they are both the same as the *exclusive or* operation. For example, the sum of the polynomials

$$x^3 + x + 1 \quad \text{and}$$

$$x^4 + x^3 + x^2 + x$$

4 2

is x + x + 1, as is their difference. These polynomials are not usually written with minus signs, but they could be, because a coefficient of –1 is equivalent to a coefficient of 1.

Multiplication of such polynomials is straightforward. The product of one coefficient by another is the same as their combination by the logical *and* operator, and the partial products are summed using *exclusive or*. Multiplication is not needed to compute the CRC checksum.

Division of polynomials over GF(2) can be done in much the same way as long division of polynomials over the integers. Here is an example.

$$
\begin{array}{r|llllllll}
 & & & & x^4 + & x^3 + & & & 1 \\ \hline
x^3+x+1 & x^7 + & x^6 + & x^5 + & & & x^2 + & x & \\
 & x^7 + & & x^5 + & x^4 & & & & \\ \cline{2-5}
 & & x^6 + & & x^4 & & & & \\
 & & x^6 + & & x^4 + & x^3 & & & \\ \cline{3-6}
 & & & & & x^3 + & x^2 + & x & \\
 & & & & & x^3 + & & x + & 1 \\ \cline{6-9}
 & & & & & & x^2 + & & 1
\end{array}
$$

The reader may verify that the quotient $x^4 + x^3 + 1$ multiplied by the divisor $x^3 + x + 1$, plus the remainder $x^2 + 1$, equals the dividend.

The CRC method treats the message as a polynomial in GF(2). For example, the message 11001001, where the order of transmission is from left to right (110...), is treated as a representation of the polynomial $x^7 + x^6 + x^3 + 1$. The sender and receiver agree on a certain fixed polynomial called the *generator* polynomial. For example, for a 16-bit CRC the CCITT (Le Comité Consultatif International Télégraphique et Téléphonique)[1] has chosen the polynomial $x^{16} + x^{12} + x^5 + 1$, which is now widely used for a 16-bit CRC checksum. To compute an r-bit CRC checksum, the generator polynomial must be of degree r. The sender appends r 0-bits to the m-bit message and divides the resulting polynomial of degree $m + r - 1$ by the generator polynomial. This produces a remainder polynomial of degree $r - 1$ (or less). The remainder polynomial has r coefficients, which are the checksum. The quotient polynomial is discarded. The data transmitted (the code vector) is the original m-bit message followed by the r-bit checksum.

There are two ways for the receiver to assess the correctness of the transmission. It can compute the checksum from the first m bits of the received data and verify that it agrees with the last r received bits. Alternatively, and following usual practice, the receiver can divide all the $m + r$ received bits by the generator polynomial and check that the r-bit remainder is 0. To see that the remainder must be 0, let M be the polynomial representation of the message, and let R be the polynomial representation of the remainder that was computed by the sender. Then the transmitted data corresponds to the polynomial $Mx^r - R$ (or, equivalently, $Mx^r + R$). By the way R was computed, we know that $Mx^r = QG + R$, where G is the generator polynomial and Q is the quotient (that was discarded). Therefore the transmitted data, $Mx^r - R$, is equal to QG, which is clearly a multiple of G. If the receiver is built as nearly as possible just like the sender, the receiver will append r 0-bits to the received data as it computes the remainder R. The received data with 0-bits appended is still a multiple of G, so the computed remainder is still 0.

That’s the basic idea, but in reality the process is altered slightly to correct for certain deficiencies. For example, the method as described is insensitive to the number of leading and trailing 0-bits in the data transmitted. In particular, if a failure occurred that caused the received data, including the checksum, to be all-0, it would be accepted.

Choosing a “good” generator polynomial is something of an art and beyond the scope of this text. Two simple observations: For an r-bit checksum, G should be of degree r, because otherwise the first bit of the checksum would always be 0, which wastes a bit of the checksum. Similarly, the last coefficient should be 1 (that is, G should not be divisible by x), because otherwise the last bit of the checksum would always be 0 (because $Mx^r = QG + R$, if G is divisible by x, then R must be also). The following facts about generator polynomials are proved in [PeBr] and/or [Tanen]:

- If G contains two or more terms, all single-bit errors are detected.
- If G is not divisible by x (that is, if the last term is 1), and e is the least positive integer such that G evenly divides $x^e + 1$, then all double errors that are within a frame of e bits are detected. A particularly good polynomial in this respect is $x^{15} + x^{14} + 1$, for which $e = 32767$.
- If $x + 1$ is a factor of G, all errors consisting of an odd number of bits are detected.
- An r-bit CRC checksum detects all burst errors of length $\le r$. (A burst error of length r is a string of r bits in which the first and last are in error, and the intermediate $r - 2$ bits may or may not be in error.)

The generator polynomial $x + 1$ creates a checksum of length 1, which applies even parity to the message. (*Proof hint:* For arbitrary $k \ge 0$, what is the remainder when dividing x^k by $x + 1$?)

It is interesting to note that if a code of any type can detect all double-bit and single-bit errors, then it can in principle correct single-bit errors. To see this, suppose data containing a single-bit error is received. Imagine complementing all the bits, one at a time. In all cases but one, this results in a double-bit error, which is detected. But when the erroneous bit is complemented, the data is error free, which is recognized. In spite of this, the CRC method does not seem to be used for single-bit error correction. Instead, the sender is requested to repeat the whole transmission if any error is detected.

14–3 Practice

Table 14–1 shows the generator polynomials used by some common CRC standards. The “Hex” column shows the hexadecimal representation of the generator polynomial; the most significant bit is omitted, as it is always 1.

The CRC standards differ in ways other than the choice of generating polynomials. Most initialize by assuming that the message has been preceded by certain nonzero bits, others do no such initialization. Most transmit the bits within a byte least significant bit first, some most significant bit first. Most append the checksum least significant byte first, others most significant byte first. Some complement the checksum.

CRC-12 is used for transmission of 6-bit character streams, and the others are for 8-bit characters, or 8-bit bytes of arbitrary data. CRC-16 is used in IBM’s BISYNCH communication standard. The CRC-CCITT polynomial, also known as ITU-TSS, is used in communication protocols such as XMODEM, X.25, IBM’s SDLC, and ISO’s HDLC

[Tanen]. CRC-32 is also known as AUTODIN-II and ITU-TSS (ITU-TSS has defined both 16- and a 32-bit polynomials). It is used in PKZip, Ethernet, AAL5 (ATM Adaptation Layer 5), FDDI (Fiber Distributed Data Interface), the IEEE-802 LAN/MAN standard, and in some DOD applications. It is the one for which software algorithms are given here.

The first three polynomials in Table 14–1 have $x + 1$ as a factor. The last (CRC-32) does not.

TABLE 14–1. GENERATOR POLYNOMIALS OF SOME CRC CODES

Common Name	r	Generator	
		Polynomial	Hex
CRC-12	12	$x^{12}+x^{11}+x^3+x^2+x+1$	80F
CRC-16	16	$x^{16}+x^{15}+x^2+1$	8005
CRC-CCITT	16	$x^{16}+x^{12}+x^5+1$	1021
CRC-32	32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	04C11DB7

To detect the error of erroneous insertion or deletion of leading 0’s, some protocols prepend one or more nonzero bits to the message. These don’t actually get transmitted; they are simply used to initialize the key register (described below) used in the CRC calculation. A value of r 1-bits seems to be universally used. The receiver initializes its register in the same way.

The problem of trailing 0’s is a little more difficult. There would be no problem if the receiver operated by comparing the remainder based on just the message bits to the checksum received. But, it seems to be simpler for the receiver to calculate the remainder for all bits received (message and checksum), plus r appended 0-bits. The remainder should be 0. With a 0 remainder, if the message has trailing 0-bits inserted or deleted, the remainder will still be 0, so this error goes undetected.

The usual solution to this problem is for the sender to complement the checksum before appending it. Because this makes the remainder calculated by the receiver nonzero (usually), the remainder will change if trailing 0’s are inserted or deleted. How then does the receiver recognize an error-free transmission?

Using the “mod” notation for remainder, we know that

$$(Mx^r + R) \bmod G = 0.$$

Denoting the “complement” of the polynomial R by $\overline{R}$, we have

$$\begin{aligned}(Mx^r + \overline{R}) \bmod G &= (Mx^r + (x^{r-1} + x^{r-2} + \ldots + 1 - R)) \bmod G \\ &= ((Mx^r + R) + x^{r-1} + x^{r-2} + \ldots + 1) \bmod G \\ &= (x^{r-1} + x^{r-2} + \ldots + 1) \bmod G.\end{aligned}$$

Thus, the checksum calculated by the receiver for an error-free transmission should be

$$(x^{r-1} + x^{r-2} + \ldots + 1) \bmod G.$$

This is a constant (for a given G). For CRC-32 this polynomial, called the *residual* or *residue*, is

$$x^{31} + x^{30} + x^{26} + x^{25} + x^{24} + x^{18} + x^{15} + x^{14} + x^{12} +$$
$$x^{11} + x^{10} + x^{8} + x^{6} + x^{5} + x^{4} + x^{3} + x + 1,$$

or hex C704DD7B [Black].

Hardware

To develop a hardware circuit for computing the CRC checksum, we reduce the polynomial division process to its essentials.

The process employs a shift register, which we denote by CRC. This is of length r (the degree of G) bits, not $r + 1$ as you might expect. When the subtractions (*exclusive or*'s) are done, it is not necessary to represent the high-order bit, because the high-order bits of G and the quantity it is being subtracted from are both 1. The division process might be described informally as follows:

```
Initialize the CRC register to all 0-bits.
    Get first/next message bit m.
    If the high-order bit of CRC is 1,
        Shift CRC and m together left 1 position, and XOR the result with the low-
        order r bits of G.
    Otherwise,
        Just shift CRC and m left 1 position.
    If there are more message bits, go back to get the next one.
```

It might seem that the subtraction should be done first, and then the shift. It would be done that way if the CRC register held the entire generator polynomial, which in bit form is $r + 1$ bits. Instead, the CRC register holds only the low-order r bits of G, so the shift is done first, to align things properly.

The contents of the CRC register for the generator $G = x^3 + x + 1$ and the message $M = x^7 + x^6 + x^5 + x^2 + x$ are shown below. Expressed in binary, $G = 1011$ and $M = 11100110$.

000	Initial CRC contents. High-order bit is 0, so just shift in first message bit.
001	High-order bit is 0, so just shift in second message bit, giving:
011	High-order bit is 0 again, so just shift in third message bit, giving:
111	High-order bit is 1, so shift and then XOR with 011, giving:
101	High-order bit is 1, so shift and then XOR with 011, giving:
001	High-order bit is 0, so just shift in fifth message bit, giving:
011	High-order bit is 0, so just shift in sixth message bit, giving:
111	High-order bit is 1, so shift and then XOR with 011, giving:
101	There are no more message bits, so this is the remainder.

These steps can be implemented with the (simplified) circuit shown in Figure 14–2, which is known as a *feedback shift register*. The three boxes in the figure represent the three bits of the CRC register. When a message bit comes in, if the high-order bit (x^2 box) is 0, simultaneously the message bit is shifted into the x^0 box, the bit in x^0 is shifted to x^1, the bit in x^1 is shifted to x^2, and the bit in x^2 is discarded. If the high-order bit of the CRC register is 1, then a 1 is present at the lower input of each of the two *exclusive or* gates. When a message bit comes in, the same shifting takes place, but the three bits that wind up in the CRC register have been *exclusive or*'ed with binary 011. When all the message bits have been processed, the CRC holds M mod G.

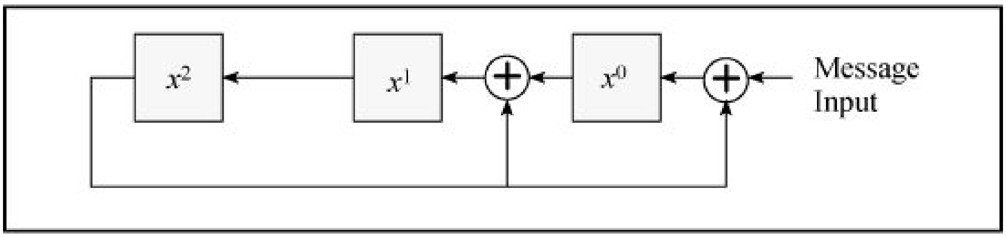


FIGURE 14–2. Polynomial division circuit for $G = x^3 + x + 1$.

If the circuit of Figure 14–2 were used for the CRC calculation, then after processing the message, r (in this case 3) 0-bits would have to be fed in. Then the CRC register would have the desired checksum, Mx^r mod G. There is a way to avoid this step with a simple rearrangement of the circuit.

Instead of feeding the message in at the right end, feed it in at the left end, r steps away, as shown in Figure 14–3. This has the effect of premultiplying the input message M by x^r. But premultiplying and postmultiplying are the same for polynomials. Therefore, as each message bit comes in, the CRC register contents are the remainder for the portion of the message processed, as if that portion had r 0-bits appended.

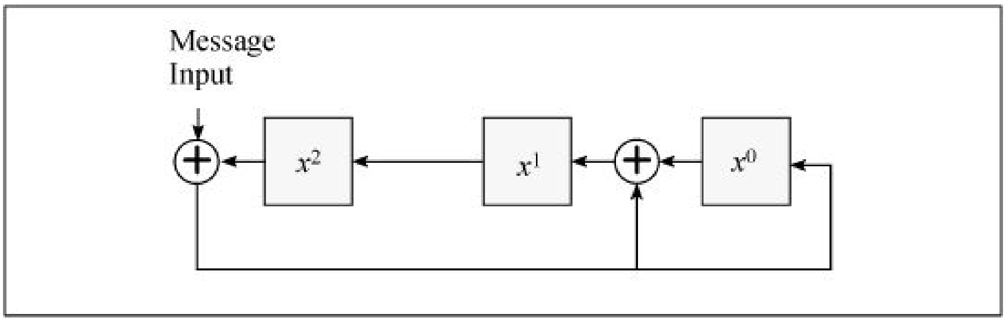


FIGURE 14–3. CRC circuit for $G = x^3 + x + 1$.

Figure 14–4 shows the circuit for the CRC-32 polynomial.

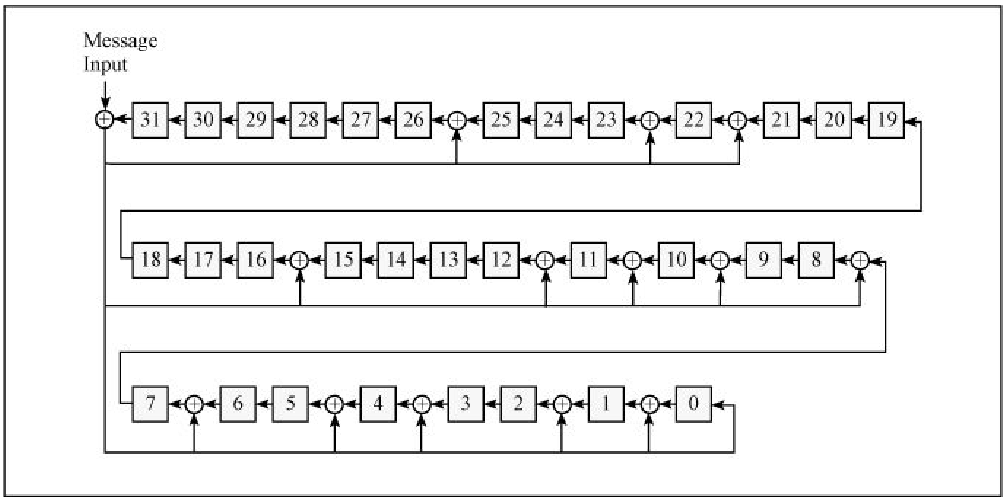


FIGURE 14–4. CRC circuit for CRC-32.

Software

Figure 14–5 shows a basic implementation of CRC-32 in software. The CRC-32 protocol initializes the CRC register to all 1's, transmits each byte least significant bit first, and complements the checksum. We assume the message consists of an integral number of bytes.

To follow Figure 14–4 as closely as possible, the program uses left shifts. This requires reversing each message byte and positioning it at the left end of the 32-bit register, denoted `byte` in the program. The word-level reversing program shown in Figure 7–1 on page 129 can be used (although this is not very efficient, because we need to reverse only eight bits).

The code of Figure 14–5 is shown for illustration only. It can be improved substantially while still retaining its one-bit-at-a-time character. First, notice that the eight bits of the reversed `byte` are used in the inner loop's `if`-statement and then discarded. Also, the high-order eight bits of `crc` are not altered in the inner loop (other than by shifting). Therefore, we can set `crc = crc ^ byte` ahead of the inner loop, simplify the `if`-statement, and omit the left shift of `byte` at the bottom of the loop.

The two reversals can be avoided by shifting right instead of left. This requires reversing the hex constant that represents the CRC-32 polynomial and testing the least significant bit of `crc`. Finally, the `if`-test can be replaced with some simple logic, to save branches. The result is shown in Figure 14–6.

```
unsigned int crc32(unsigned char *message) {
   int i, j;
   unsigned int byte, crc;

   i = 0;
   crc = 0xFFFFFFFF;
   while (message[i] != 0) {
      byte = message[i];              // Get next byte.
      byte = reverse(byte);           // 32-bit reversal.
      for (j = 0; j < = 7; j++) {     // Do eight times.
         if ((int)(crc ^ byte) < 0)
              crc = (crc << 1) ^ 0x04C11DB7;
         else crc = crc << 1;
         byte = byte << 1;            // Ready next msg bit.
      }
      i = i + 1;
   }
   return reverse(~crc);
}
```

FIGURE 14–5. Basic CRC-32 algorithm.

It is not unreasonable to unroll the inner loop by the full factor of eight. If this is done, the program of Figure 14–6 executes in about 46 instructions per byte of input message. This includes a load and a branch. (We rely on the compiler to common the two loads of `message[i]` and to transform the `while`-loop so there is only one branch, at the bottom of the loop.)

```
unsigned int crc32(unsigned char *message) {
   int i, j;
   unsigned int byte, crc, mask;

   i = 0;
   crc = 0xFFFFFFFF;
   while (message[i] != 0) {
      byte = message[i];              // Get next byte.
      crc = crc ^ byte;
      for (j = 7; j >= 0; j--) {      // Do eight times.
         mask = -(crc & 1);
         crc = (crc >> 1) ^ (0xEDB88320 & mask);
      }
      i = i + 1;
   }
   return ~crc;
}
```

FIGURE 14–6. Improved bit-at-a-time CRC-32 algorithm.

Our next version employs table lookup. This is the usual way that CRC-32 is calculated. Although the programs above work one bit at a time, the table lookup method (as usually implemented) works one byte at a time. A table of 256 fullword constants is used.

The inner loop of Figure 14–6 shifts register `crc` right eight times, while doing an *exclusive or* operation with a constant when the low-order bit of `crc` is 1. These steps can be replaced by a single right shift of eight positions, followed by a single *exclusive or* with a mask that depends on the pattern of 1-bits in the rightmost eight bits of the `crc` register.

It turns out that the calculations for setting up the table are the same as those for computing the CRC of a single byte. The code is shown in Figure 14–7. To keep the program self-contained, it includes steps to set up the table on first use. In practice, these steps would probably be put in a separate function to keep the CRC calculation as simple as possible. Alternatively, the table could be defined by a long sequence of array initialization data. When compiled with GCC to the basic RISC, the function executes 13 instructions per byte of input. This includes two loads and one branch instruction.

Faster versions of these programs can be constructed by standard techniques, but there is nothing dramatic known to this writer. One can unroll loops and do careful scheduling of loads that the compiler may not do automatically. One can load the message string a halfword or a word at a time (with proper attention paid to alignment), to reduce the number of loads of the message and the number of *exclusive or*'s of `crc` with the message (see exercise 1). The table lookup method can process message bytes two at a time using a table of size 65536 words. This might make the program run faster or slower, depending on the size of the data cache and the penalty for a miss.

```
unsigned int crc32(unsigned char *message) {
   int i, j;
   unsigned int byte, crc, mask;
   static unsigned int table[256];

   /* Set up the table, if necessary. */

   if (table[1] == 0) {
      for (byte = 0; byte <= 255; byte++) {
         crc = byte;
         for (j = 7; j >= 0; j--) {     // Do eight times.
            mask = -(crc & 1);
            crc = (crc >> 1) ^ (0xEDB88320 & mask);
         }
         table[byte] = crc;
      }
   }

   /* Through with table setup, now calculate the CRC. */

   i = 0;
   crc = 0xFFFFFFFF;
   while ((byte = message[i]) != 0) {
      crc = (crc >> 8) ^ table[(crc ^ byte) & 0xFF];
      i = i + 1;
   }
   return ~crc;
}
```

FIGURE 14–7. Table lookup CRC algorithm.

Exercises

1. Show that if a generator *G* contains two or more terms, all single-bit errors are detected.

2. Referring to Figure 14–7, show how to code the main loop so that the message data is loaded one word at a time. For simplicity, assume the message is full-word aligned and an integral number of words in length, before the zero byte that marks the end of the message.

Chapter 15. Error-Correcting Codes

15–1 Introduction

This section is a brief introduction to the theory and practice of error-correcting codes (ECCs). We limit our attention to binary forward error-correcting (FEC) block codes. This means that the symbol alphabet consists of just two symbols (which we denote 0 and 1), that the receiver can correct a transmission error without asking the sender for more information or for a retransmission, and that the transmissions consist of a sequence of fixed length blocks, called *code words*.

Section 15–2 describes the code independently discovered by R. W. Hamming and M. J. E. Golay before 1950 [Ham]. This code is single error-correcting (SEC), and a simple extension of it, also discovered by Hamming, is single error-correcting and, simultaneously, double error-detecting (SEC-DED).

Section 15–4 steps back and asks what is possible in the area of forward error correction. Still sticking to binary FEC block codes, the basic question addressed is: for a given block length (or *code length*) and level of error detection and correction capability, how many different code words can be encoded?

Section 15–2 is for readers who are primarily interested in learning the basics of how ECC works in computer memories. Section 15–4 is for those who are interested in the mathematics of the subject, and who might be interested in the challenge of an unsolved mathematical problem.

The reader is cautioned that over the past 50 years ECC has become a very big subject. Many books have been published on it and closely related subjects [Hill, LC, MS, and Roman, to mention a few]. Here we just scratch the surface and introduce the reader to two important topics and to some of the terminology used in this field. Although much of the subject of error-correcting codes relies very heavily on the notations and results of linear algebra, and, in fact, is a very nice application of that abstract theory, we avoid it here for the benefit of those who are not familiar with that theory.

The following notation is used throughout this chapter. The terms are defined in subsequent sections.

m	Number of “information” or “message” bits
k	Number of parity-check bits (“check bits,” for short)
n	Code length, $n = m + k$
u	Information bit vector, u_0, u_1, ... u_{m-1}
p	Parity check bit vector, p_0, p_1, ..., p_{k-1}
s	Syndrome vector, s_0, s_1, ..., s_{k-1}

15–2 The Hamming Code

Hamming’s development [Ham] is a very direct construction of a code that permits correcting single-bit errors. He assumes that the data to be transmitted consists of a certain number of *information bits u*, and he adds to these a number of *check bits p*, such that if a block is received that has at most one bit in error, then p identifies the bit that is in error (which might be one of the check bits). Specifically, in Hamming’s code, p is interpreted as an integer that is 0 if no error occurred, and otherwise is the 1-origin index of the bit that is in error. Let m be the number of information bits, and k

the number of check bits used. Because the k check bits must check themselves as well as the information bits, the value of p, interpreted as an integer, must range from 0 to $m + k$, which is $m + k + 1$ distinct values. Because k bits can distinguish 2^k cases, we must have

$$2^k \geq m + k + 1. \qquad (1)$$

This is known as the *Hamming rule*. It applies to any single-error correcting (SEC) binary FEC block code in which all of the transmitted bits must be checked. The check bits will be interspersed among the information bits in a manner described below.

Because p indexes the bit (if any) that is in error, the least significant bit of p must be 1 if the erroneous bit is in an odd position, and 0 if it is in an even position or if there is no error. A simple way to achieve this is to let the least significant bit of p, p_0, be an even parity check on the odd positions of the block and to put p_0 in an odd position. The receiver then checks the parity of the odd positions (including that of p_0). If the result is 1, an error has occurred in an odd position, and if the result is 0, either no error occurred or an error occurred in an even position. This satisfies the condition that p should be the index of the erroneous bit, or be 0 if no error occurred.

Similarly, let the next-from-least significant bit of p, p_1, be an even parity check of positions 2, 3, 6, 7, 10, 11, ... (in binary, 10, 11, 110, 111, 1010, 1011, ...), and put p_1 in one of these positions. Those positions have a 1 in their second-from-least significant binary position number. The receiver checks the parity of these positions (including the position of p_1). If the result is 1, an error occurred in one of those positions, and if the result is 0, either no error occurred or an error occurred in some other position.

Continuing, the third-from-least significant check bit, p_2, is made an even parity check on those positions that have a 1 in their third-from-least significant position number, namely positions 4, 5, 6, 7, 12, 13, 14, 15, 20, ..., and p_2 is put in one of those positions.

Putting the check bits in power-of-two positions (1, 2, 4, 8, ...) has the advantage that they are independent. That is, the sender can compute p_0 independent of p_1, p_2, ... and, more generally, it can compute each check bit independent of the others.

As an example, let us develop a single error-correcting code for $m = 4$. Solving (1) for k gives $k = 3$, with equality holding. This means that all 2^k possible values of the k check bits are used, so it is particularly efficient. A code with this property is called a *perfect* code.[1]

This code is called the (7, 4) Hamming code, which signifies that the code length is 7 and the number of information bits is 4. The positions of the check bits p_i and the information bits u_i are shown here.

p_0	p_1	u_3	p_2	u_2	u_1	u_0
1	2	3	4	5	6	7

Table 15–1 shows the entire code. The 16 rows show all 16 possible information bit configurations and the check bits calculated by Hamming's method.

To illustrate how the receiver corrects a single-bit error, suppose the code word

1001110

is received. This is row 4 in Table 15–1 with bit 6 flipped. The receiver calculates the *exclusive or* of the bits in odd positions and gets 0. It calculates the *exclusive or* of bits 2, 3, 6, and 7 and gets 1. Lastly, it calculates the *exclusive or* of bits 4, 5, 6, and 7 and gets 1. Thus the error indicator, which is called the *syndrome*, is binary 110, or 6. The receiver flips the bit at position 6 to correct the block.

TABLE 15–1. THE (7,4) HAMMING CODE

	1	2	3	4	5	6	7
Information	p_0	p_1	u_3	p_2	u_2	u_1	u_0
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1
10	1	0	1	1	0	1	0
11	0	1	1	0	0	1	1
12	0	1	1	1	1	0	0
13	1	0	1	0	1	0	1
14	0	0	1	0	1	1	0
15	1	1	1	1	1	1	1

A SEC-DED Code

For many applications, a single error-correcting code would be considered unsatisfactory, because it accepts all blocks received. A SEC-DED code seems safer, and it is the level of correction and detection most often used in computer memories.

The Hamming code can be converted to a SEC-DED code by adding one check bit, which is a parity bit (let us assume even parity) on all the bits in the SEC code word. This code is called an *extended Hamming code* [Hill, MS]. It is not obvious that it is SEC-DED. To see that it is, consider Table 15–2. It is assumed *a priori* that either 0, 1, or 2 transmission errors occur. As indicated in Table 15–2, if there are no errors, the overall parity (the parity of the entire *n*-bit received code word) will be even, and the syndrome of the (n – 1) -bit SEC portion of the block will be 0. If there is one error, then the overall parity of the received block will be odd. If the error occurred in the overall parity bit, then the syndrome will be 0. If the error occurred in some other bit, then the syndrome will be nonzero and it will indicate which bit is in error. If there are two errors, then the overall parity of the received block will be even. If one of the two errors is in the overall parity bit, then the other is in the SEC portion of the block. In this case, the syndrome will be nonzero (and will indicate the bit in the SEC portion that is in error). If the errors are both in the SEC portion of the block, then the syndrome will also be nonzero, although the reason is a bit hard to explain.

TABLE 15–2. ADDING A PARITY BIT TO MAKE A SEC-DED CODE

Possibilities			
Errors	**Overall Parity**	**Syndrome**	**Receiver Conclusion**
0	even	0	No error.
1	odd	0	Overall parity bit is in error.
		$\neq 0$	Syndrome indicates the bit in error.
2	even	$\neq 0$	Double error (not correctable).

The reason is that there must be a check bit that checks one of the two bit positions, but not the other one. The parity of this check bit and the bits it checks will thus be odd, resulting in a nonzero syndrome. Why must there be a check bit that checks one of the erroneous bits but not the other one? To see this, first suppose one of the erroneous bits is in an even position and the other is in an odd position. Then, because one of the check bits (p_0) checks all the odd positions and none of the even positions, the parity of the bits at the odd positions will be odd, resulting in a nonzero syndrome. More generally, suppose the erroneous bits are in positions i and j (with $i \neq j$). Then, because the binary representations of i and j must differ in some bit position, one of them has a 1 at that position and the other has a 0 at that position. The check bit corresponding to this position in the binary integers checks the bits at positions in the code word that have a 1 in their position number, but not the positions that have a 0 in their position number. The bits covered by that check bit will have odd parity, and thus the syndrome will be nonzero. As an example, suppose the erroneous bits are in positions 3 and 7. In binary, the position numbers are 0...0011 and 0...0111. These numbers differ in the third position from the right, and at that position the number 7 has a 1 and the number 3 has a 0. Therefore, the bits checked by the third check bit (these are bits 4, 5, 6, 7, 12, 13, 14, 15, ...) will have odd parity.

Thus, referring to Table 15–2, the overall parity and the syndrome together uniquely identify whether 0, 1, or 2 errors occurred. In the case of one error, the receiver can correct it. In the case of two errors, the receiver cannot tell whether just one of the errors is in the SEC portion (in which case it could correct it) or both errors are in the SEC portion (in which case an attempt to correct it would result in incorrect information bits).

The overall parity bit could as well be a parity check on only the even positions, because the overall parity bit is easily calculated from that and the parity of the odd positions (which is the least significant check bit). More generally, the overall parity bit could as well be a parity check on the complement set of bits checked by any one of the SEC parity bits. This observation might save some gates in hardware.

It should be clear that the Hamming SEC code has minimum redundancy. That is, for a given number of information bits, it adds a minimum number of check bits that permit single error correction. This is so because by construction, just enough check bits are added so that when interpreted as an integer, they can index any bit in the code, with one state left over to denote “no errors.” In other words, the code satisfies inequality (1). Hamming shows that the SEC-DED code constructed from a SEC code by adding one overall parity bit is also of minimum redundancy. His argument is to assume that a SEC-DED code exists that has fewer check bits, and he derives from this a contradiction to the fact that the starting SEC code had minimum redundancy.

Minimum Number of Check Bits Required

The middle column of Table 15–3 shows minimal solutions of inequality (1) for a range of values of *m*. The rightmost column simply shows that one more bit is required for a SEC-DED code. From this table one can see, for example, that to provide the SEC-DED level ECC for a memory word containing 64 information bits, eight check bits are required, giving a total memory word size of 72 bits.

TABLE 15–3. EXTRA BITS FOR ERROR CORRECTION / DETECTION

Number of Information Bits m	k for SEC	k for SEC-DED
1	2	3
2 to 4	3	4
5 to 11	4	5
12 to 26	5	6
27 to 57	6	7
58 to 120	7	8
121 to 247	8	9
248 to 502	9	10

Concluding Remarks

In the more mathematically oriented ECC literature, the term “Hamming code” is reserved for the perfect codes described above—that is, those with (n, m) = (3, 1), (7, 4), (15, 11), (31, 26), and so on. Similarly, the extended Hamming codes are the perfect SEC-DED codes described above. Computer architects and engineers often use the term to denote any of the codes that Hamming described, and some variations. The term “extended” is often understood.

The first IBM computer to use Hamming codes was the IBM Stretch computer (model 7030), built in 1961 [LC]. It used a (72, 64) SEC-DED code (not a perfect code). A follow-on machine known as Harvest (model 7950), built in 1962, was equipped with 22-track tape drives that employed a (22, 16) SEC-DED code. The ECCs found on modern machines are usually not Hamming codes, but rather are codes devised for some logical or electrical property, such as minimizing the depth of the parity check trees, and making them all the same length. Such codes give up Hamming’s simple method of determining which bit is in error, and instead use a hardware table lookup.

At the time of this writing (2012), most notebook PCs (personal computers) have no error checking in their memory systems. Desktop PCs may have none, or they may have a simple parity check. Server-class computers generally have ECC at the SEC-DED level.

In the early solid-state computers equipped with ECC memory, the memory was usually in the form of eight check bits and 64 information bits. A memory module (group of chips) might be built from, typically, nine 8-bit-wide chips. A word access (72 bits, including check bits) fetches eight bits from each of these nine chips. Each chip is laid out in such a way that the eight bits accessed for a single word are physically far apart. Thus, a word access references 72 bits that are physically somewhat separated. With bits interleaved in that way, if a few close-together bits in the same chip are altered, as, for example, by an alpha particle or cosmic ray hit, a few words will have single-bit errors, which can be corrected. Some larger memories incorporate a technology known as *Chipkill*. This allows the computer to continue to function even if an entire memory chip fails, for example, due to loss of power to the chip.

The interleaving technique can be used in communication applications to correct burst errors by interleaving the bits in time.

Today the organization of ECC memories is often more complicated than simply having eight check bits and 64 information bits. Modern server memories might have 16 or 32 information bytes (128 or 256 bits) checked as a single ECC word. Each DRAM chip may store two, three, or four bits in physically adjacent positions. Correspondingly, ECC is done on alphabets of four, eight, or 16 characters—a subject not discussed here. Because the DRAM chips usually come in 8- or 16-bit-wide configurations, the memory module often provides more than enough bits for the ECC function. The extra bits might be used for other functions, such as one or two parity bits on the memory address. This allows the memory to check that the address it receives is (probably) the address that the CPU generated.

In modern server-class machines, ECC might be used in different levels of cache memory, as well as in main memory. It might also be used in non-memory areas, such as on busses.

15–3 Software for SEC-DED on 32 Information Bits

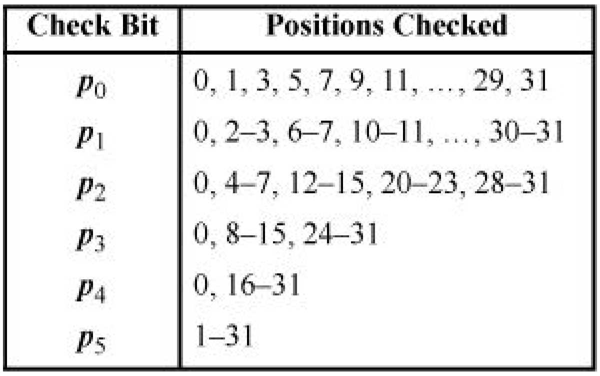
This section describes a code for which encoding and decoding can be efficiently implemented in software for a basic RISC. It does single error correction and double error detection on 32 information bits. The technique is basically Hamming's.

We follow Hamming in using check bits in such a way that the receiver can easily (in software) determine whether zero, one, or two errors occurred, and if one error occurred it can easily correct it. We also follow Hamming in using a single overall parity bit to convert a SEC code to SEC-DED, and we assume the check bit values are chosen to make even parity on the check bit and the bits it checks. A total of seven check bits are required (Table 15–3).

Consider first just the SEC property, without DED. For SEC, six check bits are required. For implementation in software, the main difficulty with Hamming's method is that it merges the six check bits with the 32 information bits, resulting in a 38-bit quantity. We are assuming the implementation is done on a 32-bit machine, and the information bits are in a 32-bit word. It would be very awkward for the sender to spread out the information bits over a 38-bit quantity and calculate the check bits into the positions described by Hamming. The receiver would have similar difficulties. The check bits could be moved into a separate word or register, with the 32 information bits kept in another word or register. But this gives an irregular range of positions that are checked by each check bit. In the scheme to be described, these ranges retain most of the regularity that they have in Hamming's scheme (which ignores word boundaries). The regularity leads to simplified calculations.

The positions checked by each check bit are shown in Table 15–4. In this table, bits are numbered in the usual little-endian way, with position 0 being the least significant bit (unlike Hamming's numbering).

TABLE 15–4. POSITIONS CHECKED BY THE CHECK BITS

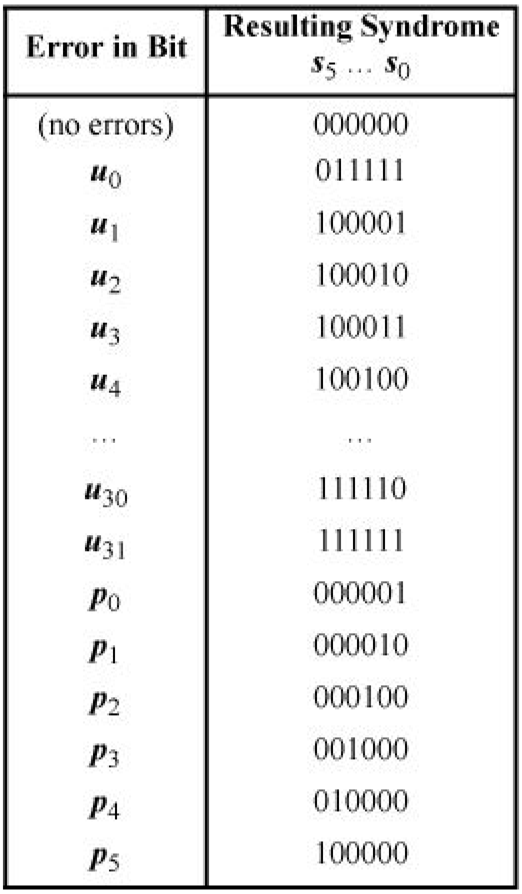


Check Bit	Positions Checked
p_0	0, 1, 3, 5, 7, 9, 11, …, 29, 31
p_1	0, 2–3, 6–7, 10–11, …, 30–31
p_2	0, 4–7, 12–15, 20–23, 28–31
p_3	0, 8–15, 24–31
p_4	0, 16–31
p_5	1–31

Observe that each of the 32 information word bit positions is checked by at least two check bits. For example, position 6 is checked by p_1 and p_2 (and also by p_5). Thus, if two information words differ in one bit position, the code words (information plus check bits) differ in at least three positions (the information bit that was corrupted and two or more check bits), so the code words are at a distance of at least three from one another (see "Hamming Distance" on page 343). Furthermore, if two information words differ in two bit positions, then at least one of p_0 – p_5 checks one of the positions, but not the other, so again the code words will be at least a distance of three apart. Therefore, the above scheme represents a code with minimum distance three (a SEC code).

Suppose a code word is transmitted to a receiver. Let u denote the information bits received, p denote the check bits received, and s (for syndrome) denote the *exclusive or* of p and the check bits calculated from u by the receiver. Then, examination of Table 15–4 reveals that s will be set as shown in Table 15–5, for zero or one errors in the code word.

TABLE 15–5. SYNDROME FOR ZERO OR ONE ERRORS



Error in Bit	Resulting Syndrome $s_5 \ldots s_0$
(no errors)	000000
u_0	011111
u_1	100001
u_2	100010
u_3	100011
u_4	100100
…	…
u_{30}	111110
u_{31}	111111
p_0	000001
p_1	000010
p_2	000100
p_3	001000
p_4	010000
p_5	100000

As an example, suppose information bit u_4 is corrupted in transmission. Table 15–4 shows that u_4 is checked by check bits p_2 and p_5. Therefore, the check bits calculated by the sender and receiver will differ in p_2 and p_5. In this scenario the check bits received are the same as those transmitted, so the syndrome will have bits 2 and 5 set —that is, it will be 100100.

If one of the check bits is corrupted in transmission (and no errors occur in the information bits), then the check bits received and those calculated by the receiver (which equal those calculated by the sender) differ in the check bit that was corrupted, and in no other bits, as shown in the last six rows of Table 15–5.

The syndromes shown in Table 15–5 are distinct for all 39 possibilities of no error or a single-bit error anywhere in the code word. Therefore, the syndrome identifies whether or not an error occurred, and if so, which bit position is in error. Furthermore, if a single-bit error occurred, it is fairly easy to calculate which bit is in error (without resorting to a table lookup) and to correct it. Here is the logic:

If s = **0**, no error occurred.
If **s** = **011111,** u_0 is in error.
If s = **1xxxxx**, with **xxxxx** nonzero, the error is in u at position **xxxxx**.

Otherwise, a single bit in s is set, the error is in a check bit, and the correct check bits are given by the *exclusive or* of the syndrome and the received check bits (or by the calculated check bits).

Under the assumption that an error in the check bits need not be corrected, this can be expressed as shown here, where *b* is the bit number to be corrected.

```
if $(s \& (s-1)) = 0$ then ...        // No correction required.
else do
  if $s = 0b011111$ then $b \leftarrow 0$
  else $b \leftarrow s \& 0b011111$
  $u \leftarrow u \oplus (1 \ll b)$                // Complement bit $b$ of $u$.
end
```

There is a hack that changes the second if-then-else construction shown above into an assignment statement.

To recognize double-bit errors, an overall parity bit is computed (parity of $u_{31:0}$ and $p_{5:0}$), and put in bit position 6 of *p* for transmission. Double-bit errors are distinguished by the overall parity being correct, but with the syndrome ($s_{5:0}$) being nonzero. The reason the syndrome is nonzero is the same as in the case of the extended Hamming code, given on page 334.

Software that implements this code is shown in Figures 15–1 and 15–2. We assume the simple case of a sender and a receiver, and the receiver has no need to correct an error that occurs in the check bits or in the overall parity bit.

```
unsigned int checkbits(unsigned int u) {

   /* Computes the six parity check bits for the
   "information" bits given in the 32-bit word u. The
   check bits are p[5:0]. On sending, an overall parity
   bit will be prepended to p (by another process).

   Bit Checks these bits of u
   p[0] 0, 1, 3, 5, ..., 31 (0 and the odd positions).
   p[1] 0, 2-3, 6-7, ..., 30-31 (0 and positions xxx1x).
   p[2] 0, 4-7, 12-15, 20-23, 28-31 (0 and posns xx1xx).
   p[3] 0, 8-15, 24-31 (0 and positions x1xxx).
   p[4] 0, 16-31 (0 and positions 1xxxx).
   p[5] 1-31 */

   unsigned int p0, p1, p2, p3, p4, p5, p6, p;
   unsigned int t1, t2, t3;

   // First calculate p[5:0] ignoring u[0].
   p0 = u ^ (u >> 2);
   p0 = p0 ^ (p0 >> 4);
   p0 = p0 ^ (p0 >> 8);
   p0 = p0 ^ (p0 >> 16);               // p0 is in posn 1.

   t1 = u ^ (u >> 1);
   p1 = t1 ^ (t1 >> 4);
   p1 = p1 ^ (p1 >> 8);
   p1 = p1 ^ (p1 >> 16);               // p1 is in posn 2.

   t2 = t1 ^ (t1 >> 2);
   p2 = t2 ^ (t2 >> 8);
   p2 = p2 ^ (p2 >> 16);               // p2 is in posn 4.

   t3 = t2 ^ (t2 >> 4);
   p3 = t3 ^ (t3 >> 16);               // p3 is in posn 8.
   p4 = t3 ^ (t3 >> 8)                 // p4 is in posn 16.

   p5 = p4 ^ (p4 >> 16);               // p5 is in posn 0.

   p = ((p0>>1) & 1)  | ((pl>>l) & 2) | ((p2>>2) & 4) |
       ((p3>>5) & 8)  | ((p4>>12) & 16) | ((p5 & 1) << 5);

   p = p ^ (-(u & 1) & 0x3F);         // Now account for u[0].
   return p;
}
```

FIGURE 15–1. Calculation of check bits.

```
int correct(unsigned int pr, unsigned int *ur) {

   /* This function looks at the received seven check
   bits and 32 information bits (pr and ur) and
   determines how many errors occurred (under the
   presumption that it must be 0, 1, or 2). It returns
   with 0, 1, or 2, meaning that no errors, one error, or
   two errors occurred. It corrects the information word
   received (ur) if there was one error in it. */

   unsigned int po, p, syn, b;

   po = parity(pr ^ *ur);         // Compute overall parity
                                  // of the received data.
   p = checkbits(*ur);            // Calculate check bits
                                  // for the received info.
   syn = p ^ (pr & 0x3F);         // Syndrome (exclusive of
                                  // overall parity bit).
   if (po == 0) {
      if (syn == 0) return 0;     // If no errors, return 0.
      else return 2;              // Two errors, return 2.
   }
                                  // One error occurred.
   if (((syn - 1) & syn) == 0)    // If syn has zero or one
      return 1;                   // bits set, then the
                                  // error is in the check
                                  // bits or the overall
                                  // parity bit (no
                                  // correction required).

   // One error, and syn bits 5:0 tell where it is in ur.

   b = syn - 31 - (syn >> 5);   // Map syn to range 0 to 31.
// if (syn == 0x1f) b = 0;     // (These two lines equiv.
// else b = syn & 0x1f;        // to the one line above.)
   *ur = *ur ^ (1 << b);       // Correct the bit.
   return 1;
}
```

FIGURE 15–2. The receiver’s actions.

To compute the check bits, function `checkbits` first ignores information bit u_0 and computes

$$0.\ (x \leftarrow u \oplus (u \overset{u}{\gg} 1))$$

$$1.\ (x \leftarrow x \oplus (x \overset{u}{\gg} 2))$$

$$2.\ (x \leftarrow x \oplus (x \overset{u}{\gg} 4))$$

$$3.\ (x \leftarrow x \oplus (x \overset{u}{\gg} 8))$$

$$4.\ (x \leftarrow x \oplus (x \overset{u}{\gg} 16))$$

except omitting line *i* when computing check bit k_i, for $0 \le i \le 4$. This puts p_i in various positions of word *x*, as shown in Figure 15–1. For p_5, all the above assignments are used. This is where the regularity of the pattern of bits checked by each check bit pays off; a lot of code commoning can be done. This reduces what would be 4×5 + 5 = 25 such assignments to 15, as shown in Figure 15–1.

Incidentally, if the computer has an instruction for computing the parity of a word, or has the *population count* instruction (which puts the word parity in the least significant bit of the target register), then the regular pattern is not needed. On such a machine, the check bits might be computed as

```
p0 = pop(u ^ 0xAAAAAAAB) & 1;
p1 = pop(u & 0xCCCCCCCD) & 1;
```

and so forth.

After packing the six check bits into a single quantity `p`, the `checkbits` function accounts for information bit `u0` by complementing all six check bits if u_0 = 1. (See Table 15-4; p_5 must be complemented because u_0 was erroneously included in the calculation of p_5 up to this point.)

15–4 Error Correction Considered More Generally

This section continues to focus on the binary FEC block codes, but a little more generally than the codes described in Section 15–2. We drop the assumption that the block consists of a set of “information” bits and a distinct set of “check” bits, and any implication that the number of code words must be a power of 2. We also consider levels of error correction and detection capability greater than SEC and SEC-DED. For example, suppose you want a double error-correcting code for a binary representation of decimal digits. If the code has 16 code words (with ten being used to represent the decimal digits and six being unused), the length of the code words must be at least 11 bits. But if a code with only 10 code words is used, the code words can be of length 10 bits. (This is shown in Table 15–8 on page 351, in the column for d = 5, as is explained below.)

A *code* is simply a set of *code words*, and for our purposes the code words are binary strings all of the same length which, as mentioned above, is called the *code length*. The number of code words in the set is called the *code size*. We make no interpretation of the code words; they might represent alphanumeric characters or pixel values in a picture, for example.

As a trivial example, a code might consist of the binary integers from 0 to 7, with each bit repeated three times:

{000000000, 000000111, 000111000, 000111111, 111000000, ... 111111111}.

Another example is the *two-out-of-five* code, in which each code word has exactly two 1-bits:

{00011, 00101, 00110, 01001, 01010, 01100, 10001, 10010, 10100, 11000}.

The code size is 10, and thus it is suitable for representing decimal digits. Notice that if code word 00110 is considered to represent decimal 0, then the remaining values can be decoded into digits 1 through 9 by giving the bits weights of 6, 3, 2, 1, and 0, in left-to-right order.

The *code rate* is a measure of the efficiency of a code. For a code like Hamming's, this can be defined as the number of information bits divided by the code length. For the Hamming code discussed above, it is $4/7 \approx 0.57$. More generally, the code rate is defined as the log base 2 of the code size, divided by the code length. The simple codes above have rates of $\log_2(8)/9 \approx 0.33$ and $\log_2(10)/5 \approx 0.66$, respectively.

Hamming Distance

The central concept in the theory of ECC is that of *Hamming distance*. The Hamming distance between two words (of equal length) is the number of bit positions in which they differ. Put another way, it is the population count of the *exclusive or* of the two words. It is appropriate to call this a distance function because it satisfies the definition of a distance function used in linear algebra:

$$\begin{aligned}
&d(x, y) = d(y, x),\\
&d(x, y) \ge 0,\\
&d(x, y) = 0 \quad \text{iff} \quad x = y, \quad \text{and}\\
&d(x, y) + d(y, z) \ge d(x, z) \quad \text{(triangle inequality)}.
\end{aligned}$$

Here $d(x, y)$ denotes the Hamming distance between code words x and y, which for brevity we will call simply the *distance* between x and y.

Suppose a code has a minimum distance of 1. That is, there are two words x and y in the set that differ in only one bit position. Clearly, if x were transmitted and the bit that makes it distinct from y were flipped due to a transmission error, then the receiver could not distinguish between receiving x with a certain bit in error and receiving y with no errors. Hence in such a code it is impossible to detect even a 1-bit error, in general.

Suppose now that a code has a minimum distance of 2. Then if just one bit is flipped in transmission, an invalid code word is produced, and thus the receiver can (in principle) detect the error. If two bits are flipped, a valid code word might be transformed into another valid code word. Thus, double-bit errors cannot be detected. Furthermore, single-bit errors cannot be *corrected*. This is because if a received word has one bit in error, then there may be two code words that are one bit-change away from the received word, and the receiver has no basis for deciding which is the original code word.

The code obtained by appending a single parity bit is in this category. It is shown below for the case of three information bits ($m = 3$). The rightmost bit is the parity bit, chosen to make even parity on all four bits. The reader may verify that the minimum distance between code words is 2.

```
0000
0011
0101
0110
1001
1010
1100
1111
```

Actually, adding a single parity bit permits detecting any odd number of errors, but when we say that a code permits detecting *k*-bit errors, we mean *all* errors up to *k* bits.

Now consider the case in which the minimum distance between code words is 3. If any one or two bits is flipped in transmission, an invalid code word results. If just one bit is flipped, the receiver can (we imagine) try flipping each of the received bits one at a time, and in only one case will a code word result. Hence in such a code the receiver can detect and correct a single-bit error. A double-bit error might appear to be a single-bit error from another code word, and thus the receiver cannot detect double-bit errors.

Similarly, it is easy to reason that if the minimum distance of a code is 4, the receiver can correct all single-bit errors and detect all double-bit errors (it is a SEC-DED code). As mentioned above, this is the level of capability often used in computer memories.

Table 15–6 summarizes the error-correction and -detection capabilities of a block code based on its minimum distance.

TABLE 15–6. NUMBER OF BITS CORRECTED / DETECTED

Minimum Distance	Correct	Detect
1	0	0
2	0	1
3	1	1
4	1	2
5	2	2
6	2	3
7	3	3
8	3	4
d	$\lfloor (d-1)/2 \rfloor$	$\lfloor d/2 \rfloor$

Error-correction capability can be traded for error detection. For example, if the minimum distance of a code is 3, that redundancy can be used to correct no errors but to detect single- or double-bit errors. If the minimum distance is 5, the code can be used to correct single-bit errors and detect 3-bit errors, or to correct no errors but to detect 4-bit errors, and so forth. Whatever is subtracted from the “Correct” column of Table 15–6 can be added to the “Detect” column.

The Main Coding Theory Problem

Up to this point we have asked, “Given a number of information bits m and a desired minimum distance d, how many check bits are required?” In the interest of generality, we will now turn this question around and ask, “For a given code length n and minimum distance d, how many code words are possible?” Thus, the number of code words need not be an integral power of 2.

Following [Roman] and others, let $A(n, d)$ denote the largest possible code size for a (binary) code with length n and minimum distance d. The remainder of this section is devoted to exploring some of what is known about this function. Determining its values has been called *the main coding theory problem* [Hill, Roman]. Throughout this section we assume that $n \geq d \geq 1$.

It is nearly trivial that

$$A(n, 1) = 2^n, \tag{2}$$

because there are 2^n distinct words of length n.

For minimum distance 2, we know from the single parity bit example that $A(n, 2) \geq 2^{n-1}$. But $A(n, 2)$ cannot exceed 2^{n-1} for the following reason. Suppose there is a code of length n and minimum distance 2 that has more than 2^{n-1} code words. Delete any one column from the code words. (We envision the code words as being arranged in a matrix much like that of Table 15–1 on page 333.) This produces a code of length $n - 1$ and minimum distance at least 1 (deleting a column can reduce the minimum distance by at most 1), and of size exceeding 2^{n-1}. Thus, it has $A(n - 1, 1) > 2^{n-1}$, contradicting Equation (2). Hence,

$$A(n, 2) = 2^{n-1}.$$

That was not difficult. What about $A(n, 3)$? That is an unsolved problem, in the sense that no formula or reasonably easy means of calculating it is known. Of course, many specific values of $A(n, 3)$ are known, and some bounds are known, but the exact value is unknown in most cases.

When equality holds in (1), it represents the solution to this problem for the case $d = 3$. Letting $n = m + k$, (1) can be rewritten

$$2^m \leq \frac{2^n}{n+1}. \tag{3}$$

Here, m is the number of information bits, so 2^m is the maximum number of code words. Hence, we have

$$A(n, 3) \leq \frac{2^n}{n+1},$$

with equality holding when $2^n/(n + 1)$ is an integer (by Hamming’s construction).

For $n = 7$, this gives $A(7, 3) = 16$, which we already know from Section 15–2. For $n = 3$ it gives $A(3, 3) \leq 2$, and the limit of 2 can be realized with code words 000 and 111. For $n = 4$ it gives $A(4, 3) \leq 3.2$, and with a little doodling you will see that it is not possible to get three code words of length 4 with $d = 3$. Thus, when equality does not hold in (3), it merely gives an upper bound, quite possibly not realizable, on the maximum number of code words.

An interesting relation is that for $n \geq 2$,

$$A(n, d) \leq 2A(n-1, d). \qquad (4)$$

Therefore, adding 1 to the code length at most doubles the number of code words possible for the same minimum distance d. To see this, suppose you have a code of length n, distance d, and size $A(n, d)$. Choose an arbitrary column of the code. Either half or more of the code words have a 0 in the selected column, or half or more have a 1 in that position. Of these two subsets, choose one that has at least $A(n, d)/2$ code words, form a new code consisting of this subset, and delete the selected column (which is either all 0's or all 1's). The resulting set of code words has n reduced by 1, has the same distance d, and has at least $A(n, d)/2$ code words. Thus, $A(n - 1, d) \geq A(n, d)/2$, from which inequality (4) follows.

A useful relation is that if d is even, then

$$A(n, d) = A(n-1, d-1). \qquad (5)$$

To see this, suppose you have a code C of length n and minimum distance d, with d odd. Form a new code by appending to each word of C a parity bit, let us say to make the parity of each word even. The new code has length $n + 1$ and has the same number of code words as does C. It has minimum distance $d + 1$. For if two words of C are a distance x apart, with x odd, then one word must have even parity and the other must have odd parity. Thus, we append a 0 in the first case and a 1 in the second case, which increases the distance between the words to $x + 1$. If x is even, we append a 0 to both words, which does not change the distance between them. Because d is odd, all pairs of words that are a distance d apart become distance $d + 1$ apart. The distance between two words more than d apart either does not change or increases. Therefore the new code has minimum distance $d + 1$. This shows that if d is odd, then $A(n+ 1, d + 1) \geq A(n, d)$, or, equivalently, $A(n, d) \geq A(n - 1, d - 1)$ for even $d \geq 2$.

Now suppose you have a code of length n and minimum distance $d \geq 2$ (d can be odd or even). Form a new code by eliminating any one column. The new code has length $n - 1$, minimum distance at least $d - 1$, and is the same size as the original code (all the code words of the new code are distinct because the new code has minimum distance at least 1). Therefore $A(n - 1, d - 1) \geq A(n, d)$. This establishes Equation (5).

Spheres

Upper and lower bounds on $A(n, d)$, for any $d \geq 1$, can be derived by thinking in terms of n-dimensional spheres. Given a code word, think of it as being at the center of a "sphere" of radius r, consisting of all words at a Hamming distance r or less from it.

How many points (words) are in a sphere of radius r? First, consider how many points are in the shell at distance exactly r from the central code word. This is given by the number of ways to choose r different items from n, ignoring the order of choice. We imagine the r chosen bits as being complemented to form a word at distance exactly r from the central point. This "choice" function, often written $\binom{n}{r}$, can be calculated from[2]

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}.$$

Thus, $\binom{n}{0} = 1$, $\binom{n}{1} = n$, $\binom{n}{2} = \frac{n(n-1)}{2}$, $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$, and so forth.

The total number of points in a sphere of radius *r* is the sum of the points in the shells from radius 0 to *r*:

$$\sum_{i=0}^{r}\binom{n}{i}.$$

There seems to be no simple formula for this sum [Knu1].

From this it is easy to obtain bounds on *A*(*n, d*). First, assume you have a code of length *n* and minimum distance *d*, and it consists of *M* code words. Surround each code word with a sphere, all of the same maximal radius such that no two spheres have a point in common. This radius is (*d* – 1)/2 if *d* is odd, and is (*d* – 2)/2 if *d* is even (see Figure 15–3). Because each point is in at most one sphere, the total number of points in the *M* spheres must be less than or equal to the total number of points in the space. That is,

$$M\sum_{i=0}^{\lfloor (d-1)/2\rfloor}\binom{n}{i} \le 2^n.$$

This holds for any *M*, hence for *M* = *A*(*n, d*), so that

$$A(n,d) \le \frac{2^n}{\sum_{i=0}^{\lfloor (d-1)/2\rfloor}\binom{n}{i}}.$$

This is known as the *sphere-packing bound*, or the *Hamming bound*.

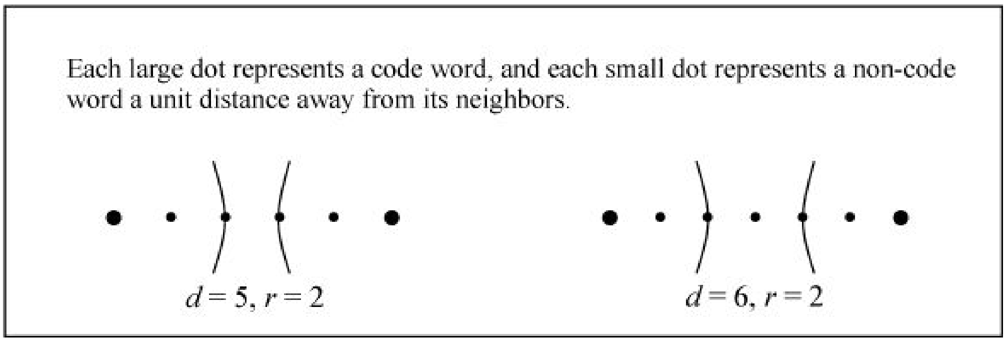

Each large dot represents a code word, and each small dot represents a non-code word a unit distance away from its neighbors.
$d = 5, r = 2$
$d = 6, r = 2$

FIGURE 15–3. Maximum radius that allows correcting points within a sphere.

The sphere idea also easily gives a lower bound on *A*(*n, d*). Assume again that you have a code of length *n* and minimum distance *d*, and it has the maximum possible number of code words—that is, it has *A*(*n, d*) code words. Surround each code word with a sphere of radius *d* – 1. Then these spheres must cover *all* 2^n points in the space (possibly overlapping). For if not, there would be a point that is at a distance *d* or more from all code words, and that is impossible because such a point would be a code word. Thus, we have a weak form of the *Gilbert-Varshamov* bound:

$$A(n, d)\sum_{i=0}^{d-1}\binom{n}{i} \geq 2^n.$$

There is the strong form of the G-V bound, which applies to *linear* codes. Its derivation relies on methods of linear algebra which, important as they are to the subject of linear codes, are not covered in this short introduction to error-correcting codes. Suffice it to say that a linear code is one in which the sum (*exclusive or*) of any two code words is also a code word. The Hamming code of Table 15–1 is a linear code. Because the G-V bound is a lower bound on linear codes, it is also a lower bound on the unrestricted codes considered here. For large *n*, it is the best known lower bound on both linear and unrestricted codes.

The strong G-V bound states that $A(n, d) \geq 2^m$, where *m* is the largest integer such that

$$2^m < \frac{2^n}{\sum_{i=0}^{d-2}\binom{n-1}{i}}.$$

That is, it is the value of the right-hand side of this inequality rounded down to the next strictly smaller integral power of 2. The “strictness” is important for cases such as (n, d) = (8, 3), (16, 3) and (the degenerate case) (6, 7).

Combining these results:

$$\mathrm{GP2LT}\left(\frac{2^n}{\sum_{i=0}^{d-2}\binom{n-1}{i}}\right) \leq A(n, d) \leq \frac{2^n}{\sum_{i=0}^{\lfloor (d-1)/2 \rfloor}\binom{n}{i}}, \tag{6}$$

where GP2LT denotes the greatest integral power of 2 (strictly) less than its argument.

Table 15–7 gives the values of these bounds for some small values of *n* and *d*. A single number in an entry means the lower and upper bounds given by (6) are equal.

TABLE 15–7. THE G - V AND HAMMING BOUNDS ON *A*(*n, d*)

n	$d = 4$	$d = 6$	$d = 8$	$d = 10$	$d = 12$	$d = 14$	$d = 16$	n
6	4 – 5	2	–	–	–	–	–	5
7	8 – 9	2	–	–	–	–	–	6
10	32 – 51	4 – 11	2 – 3	2	–	–	–	9
13	256 – 315	16 – 51	2 – 13	2 – 5	2	–	–	12
16	2048	64 – 270	8 – 56	2 – 16	2 – 6	2 – 3	2	15
19	8192 – 13797	256 – 1524	16 – 265	4 – 64	2 – 20	2 – 8	2 – 4	18
22	65536 – 95325	1024 – 9039	64 – 1342	8 – 277	4 – 75	2 – 25	2 – 10	21
25	2^{19} – 671088	4096 – 55738	256 – 7216	32 – 1295	8 – 302	2 – 88	2 – 31	24
28	2^{22} – 4793490	32768 – 354136	1024 – 40622	128 – 6436	16 – 1321	4 – 337	2 – 104	27
	$d = 3$	$d = 5$	$d = 7$	$d = 9$	$d = 11$	$d = 13$	$d = 15$	

If *d* is even, bounds can be computed directly from (6) or, making use of Equation (5), they can be computed from (6) with *d* replaced with *d* – 1 and *n* replaced with *n* – 1 in the two bounds expressions. It turns out that the latter method always results in tighter or equal bounds. Therefore, the entries in Table 15–7 were calculated only for odd *d*. To access the table for even *d*, use the values of *d* shown in the heading and the values of *n* shown at the left.

The bounds given by (6) can be seen to be rather loose, especially for large *d*. The ratio of the upper bound to the lower bound diverges to infinity with increasing *n*. The lower bound is particularly loose. Over a thousand papers have been written describing methods to improve these bounds, and the results as of this writing are shown in Table 15–8 [Agrell, Brou; where they differ, Table 15–8. shows the tighter bounds].

TABLE 15–8. BEST KNOWN BOUNDS ON $A(n, d)$

n	$d = 4$	$d = 6$	$d = 8$	$d = 10$	$d = 12$	$d = 14$	$d = 16$	n
6	4	2	–	–	–	–	–	5
7	8	2	–	–	–	–	–	6
8	16	2	2	–	–	–	–	7
9	20	4	2	–	–	–	–	8
10	40	6	2	2	–	–	–	9
11	72	12	2	2	–	–	–	10
12	144	24	4	2	2	–	–	11
13	256	32	4	2	2	–	–	12
14	512	64	8	2	2	2	–	13
15	1024	128	16	4	2	2	–	14
16	2048	256	32	4	2	2	2	15
17	2720 – 3276	256 – 340	36	6	2	2	2	16
18	5312 – 6552	512 – 673	64 – 72	10	4	2	2	17
19	10496 – 13104	1024 – 1237	128 – 135	20	4	2	2	18
20	20480 – 26168	2048 – 2279	256	40	6	2	2	19
21	36864 – 43688	2560 – 4096	512	42 – 47	8	4	2	20
22	73728 – 87376	4096 – 6941	1024	64 – 84	12	4	2	21
23	147456 – 173015	8192 – 13674	2048	80 – 150	24	4	2	22
24	294912 – 344308	16384 – 24106	4096	128 – 268	48	6	4	23
25	2^{19} – 599184	16384 – 47538	4096 – 5421	192 – 466	52 – 55	8	4	24
26	2^{20} – 1198368	32768 – 84260	4104 – 9672	384 – 836	64 – 96	14	4	25
27	2^{21} – 2396736	65536 – 157285	8192 – 17768	512 – 1585	128 – 169	28	6	26
28	2^{22} – 4792950	131072 – 291269	16384 – 32151	1024 – 3170	178 – 288	56	8	27
	$d = 3$	$d = 5$	$d = 7$	$d = 9$	$d = 11$	$d = 13$	$d = 15$	

The cases of (n, d) = (7, 3), (15, 3), and (23, 7) are *perfect* codes, meaning that they achieve the upper bound given by (6). This definition is a generalization of that given on page 333. The codes for which n is odd and $n = d$ are also perfect; see exercise 8.

We conclude this chapter by pointing out that the idea of minimum distance over an entire code, which leads to the ideas of p-bit error detection and q-bit error correction for some p and q, is not the only criterion for the "power" of a binary FEC block code. For example, work has been done on codes aimed at correcting burst errors. [Etzion] has demonstrated a (16, 11) code, and others, that can correct any single-bit error and any error in two consecutive bits, and is perfect, in a sense not discussed here. It is not capable of general double-bit error detection. The (16, 11) extended Hamming code is SEC-DED and is perfect. Thus, his code gives up general double-bit error *detection* in return for double-bit error *correction* of consecutive bits. This is, of course, interesting because in many applications errors are likely to occur in short bursts.

Exercises

1. Show a Hamming code for $m = 3$ (make a table similar to Table 15-1).

2. In a certain application of an SEC code, there is no need to correct the check bits. Hence the k check bits need only check the information bits, but not themselves. For m information bits, k must be large enough so that the receiver can distinguish $m + 1$ cases: which of the m bits is in error, or no error occurred. Thus, the number of check bits required is given by $2^k \geq m + 1$. This is a weaker restriction on k than is the Hamming rule, so it should be possible to construct, for some values of m, an SEC code that has fewer check bits than those required by the Hamming rule. Alternatively, one could have just one value to signify that an error occurred somewhere in the check bits, without specifying where. This would lead to the rule $2^k \geq m + 2$.

What is wrong with this reasoning?

3. (Brain teaser) Given m, how would you find the least k that satisfies inequality (1)?

4. Show that the Hamming distance function for any binary block code satisfies the triangle inequality: if x and y are code vectors and $d(x, y)$ denotes the Hamming distance between them, then

$$d(x, z) \leq d(x, y) + d(y, z).$$

5. Prove: $A(2n, 2d) \geq A(n, d)$.

6. Prove the "singleton bound": $A(n, d) \leq 2^{n-d+1}$.

7. Show that the notion of a perfect code as equality in the right-hand portion of inequality (6) is a generalization of the Hamming rule.

8. What is the value of $A(n, d)$ if $n = d$? Show that for odd n, these codes are perfect.

9. Show that if n is a multiple of 3 and $d = 2n/3$, then $A(n, d) = 4$.

10. Show that if $d > 2n/3$, then $A(n, d) = 2$.

11. A two-dimensional parity check scheme for 64 information bits arranges the information bits $u_0 \ldots u_{63}$ into an 8×8 array, and appends a parity bit to each row and column as shown below.

$$\begin{bmatrix} u_0 & u_1 & \cdots & u_6 & u_7 & r_0 \\ u_8 & u_9 & \cdots & u_{14} & u_{15} & r_1 \\ & & \cdots & & & \\ u_{48} & u_{49} & \cdots & u_{54} & u_{55} & r_6 \\ u_{56} & u_{57} & \cdots & u_{62} & u_{63} & r_7 \\ c_0 & c_1 & \cdots & c_6 & c_7 & r_8 \end{bmatrix}$$

The r_i are parity check bits on the rows, and the c_i are parity check bits on the columns. The “corner” check bit could be parity check on the row or the column of check bits (but not both); it is shown as a check on the bottom row (check bits c_0 through c_7).

Comment on this scheme. In particular, is it SEC-DED? Is its error-detection and -correction capability significantly altered if the corner bit r_8 is omitted? Is there any simple relation between the value of the corner bit if it’s a row sum or a column sum?

Chapter 16. Hilbert’s Curve

In 1890, Giuseppe Peano discovered a planar curve[1] with the rather surprising property that it is “space-filling.” The curve winds around the unit square and hits every point (x, y) at least once.

Peano’s curve is based on dividing each side of the unit square into three equal parts, which divides the square into nine smaller squares. His curve traverses these nine squares in a certain order. Then, each of the nine small squares is similarly divided into nine still smaller squares, and the curve is modified to traverse all these squares in a certain order. The curve can be described using fractions expressed in base 3; in fact, that’s the way Peano first described it.

In 1891, David Hilbert [Hil] discovered a variation of Peano’s curve based on dividing each side of the unit square into two equal parts, which divides the square into four smaller squares. Then, each of the four small squares is similarly divided into four still smaller squares, and so on. For each stage of this division, Hilbert gives a curve that traverses all the squares. Hilbert’s curve, sometimes called the “Peano-Hilbert curve,” is the limit curve of this division process. It can be described using fractions expressed in base 2.

Figure 16–1 shows the first three steps in the sequence that leads to Hilbert’s space-filling curve, as they were depicted in his 1891 paper.

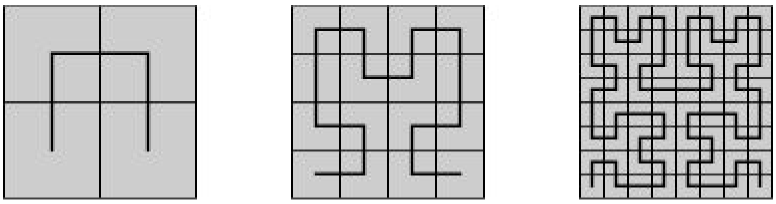


FIGURE 16–1. First three curves in the sequence defining Hilbert’s curve.

Here, we do things a little differently. We use the term “Hilbert curve” for any of the curves on the sequence whose limit is the Hilbert space-filling curve. The “Hilbert curve of order n” means the nth curve in the sequence. In Figure 16–1, the curves are of order 1, 2, and 3. We shift the curves down and to the left so that the corners of the curves coincide with the intersections of the lines in the boxes above. Finally, we scale the size of the order n curve up by a factor of 2^n, so that the coordinates of the corners of the curves are integers. Thus, our order n Hilbert curve has corners at integers ranging from 0 to $2^n - 1$ in both x and y. We take the positive direction along the curve to be from $(x, y) = (0, 0)$ to $(2^n - 1.0)$. Figure 16–2 shows the Hilbert curves of orders 1 through 6.

16–1 A Recursive Algorithm for Generating the Hilbert Curve

To see how to generate a Hilbert curve, examine the curves in Figure 16–2. The order 1 curve goes up, right, and down. The order 2 curve follows this overall pattern. First, it makes a U-shaped curve that goes up, in net effect. Second, it takes a unit step up. Third, it takes a U-shaped curve, a step, and another U, all to the right. Finally, it takes a step down, followed by a U that goes down, in net effect.

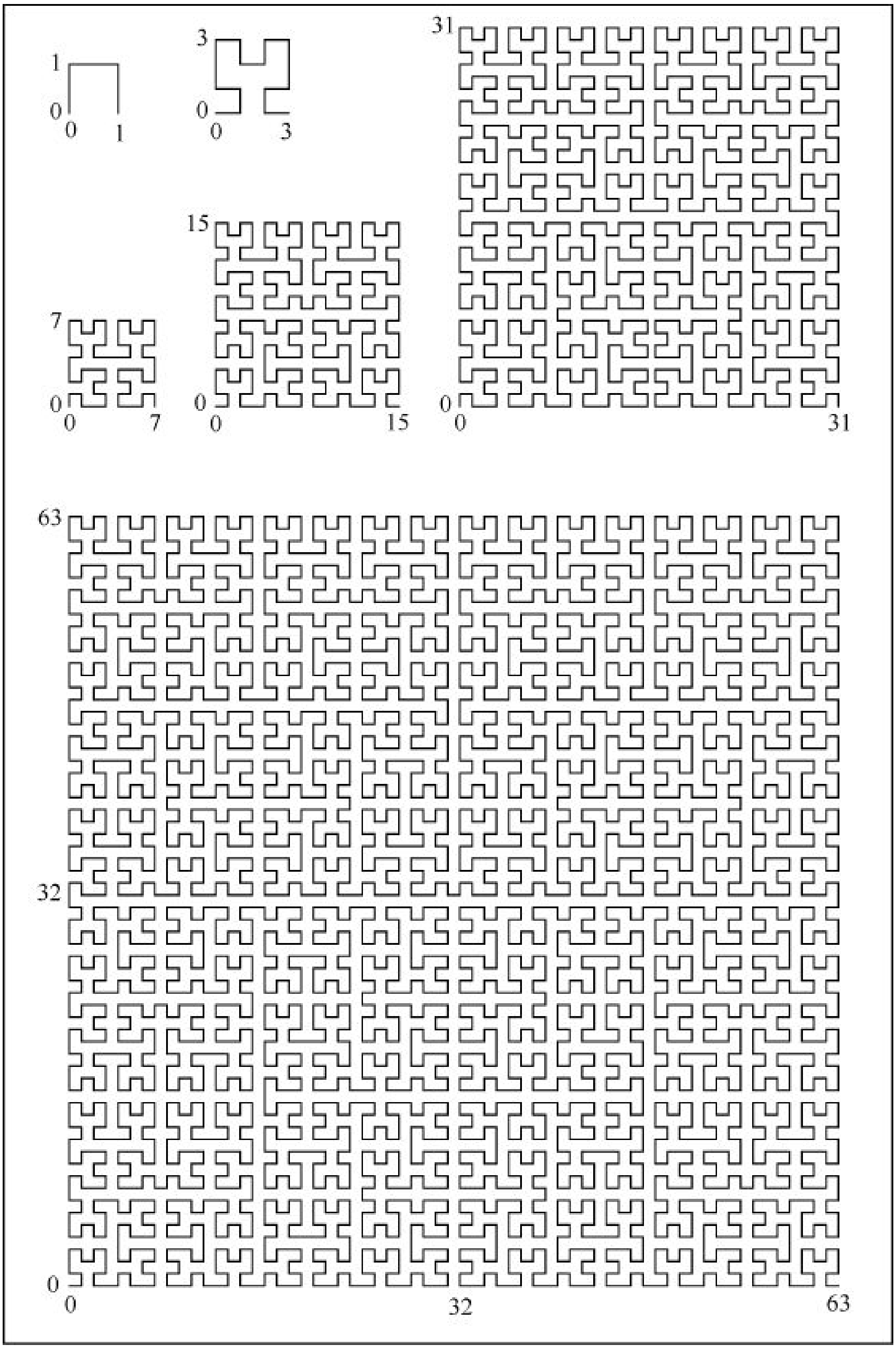


FIGURE 16–2. Hilbert curves of orders 1–6.

The order 1 inverted U is converted into the order 2 Y-shaped curve.

We can regard the Hilbert curve of any order as a series of U-shaped curves of various orientations, each of which, except for the last, is followed by a unit step in a certain direction. In transforming a Hilbert curve of one order to the next, each U-shaped curve is transformed into a Y-shaped curve with the same general orientation, and each unit step is transformed to a unit step in the same direction.

The transformation of the order 1 Hilbert curve (a U curve with a net direction to the right and a clockwise rotational orientation) to the order 2 Hilbert curve goes as follows:

1. Draw a U that goes up and has a counterclockwise rotation.
2. Draw a step up.
3. Draw a U that goes to the right and has a clockwise rotation.
4. Draw a step to the right.
5. Draw a U that goes to the right and has a clockwise rotation.
6. Draw a step down.
7. Draw a U that goes down and has a counterclockwise rotation.

We can see by inspection that all U's that are oriented as the order 1 Hilbert curve are transformed in the same way. A similar set of rules can be made for transforming U's with other orientations. These rules are embodied in the recursive program shown in Figure 16–3 [Voor]. In this program, the orientation of a U curve is characterized by two integers that specify the net linear and the rotational directions, encoded as follows:

dir = 0: right	rot = +1: clockwise
dir = 1: up	rot = –1: counterclockwise
dir = 2: left	
dir = 3: down	

Actually, `dir` can take on other values, but its congruency modulo 4 is what matters.

```
void step(int);

void hilbert(int dir, int rot, int order) {

   if (order == 0) return;

   dir = dir + rot;
   hilbert(dir, -rot, order - 1);
   step(dir);
   dir = dir - rot;
   hilbert(dir, rot, order - 1);
   step(dir);
   hilbert(dir, rot, order - 1);
   dir = dir - rot;
   step(dir);
   hilbert(dir, -rot, order - 1);
}
```

FIGURE 16–3. Hilbert curve generator.

Figure 16–4 shows a driver program and function `step` that is used by program `hilbert`. This program is given the order of a Hilbert curve to construct, and it displays a list of line segments, giving for each the direction of movement, the length along the curve to the end of the segment, and the coordinates of the end of the segment. For example, for order 2 it displays

```
    0  0000  00  00
    0  0001  01  00
    1  0010  01  01
    2  0011  00  01
    1  0100  00  10
    1  0101  00  11
    0  0110  01  11
   -1  0111  01  10
    0  1000  10  10
    1  1001  10  11
    0  1010  11  11
   -1  1011  11  10
   -1  1100  11  01
   -2  1101  10  01
   -1  1110  10  00
    0  1111  11  00
```

```
#include <stdio.h>
#include <stdlib.h>

int x = -1, y = 0;          // Global variables.
int s = 0;                  // Dist. along curve.
int blen;                   // Length to print.

void hilbert(int dir, int rot, int order);

void binary(unsigned k, int len, char *s) {
/*  Converts  the  unsigned  integer  k  to  binary character form.
Result is string s of length len. */
  int i;

  s[len] = 0;
  for (i = len - 1; i >= 0; i--) {
     if (k & 1) s[i] = ‘1’;
     else   s[i] = ‘0’;
     k = k >> 1;
  }
}
void step(int dir) {
   char ii[33], xx[17], yy[17];

   switch(dir & 3) {
      case 0: x = x + 1; break;
      case 1: y = y + 1; break;
      case 2: x = x - 1; break;
      case 3: y = y - 1; break;
   }
   binary(s, 2*blen, ii);
   binary(x, blen, xx);
   binary(y, blen, yy);
   printf("%5d %s %s %s\n", dir, ii, xx, yy);
   s = s + 1;               // Increment distance.
}
int main(int argc, char *argv[]) {
   int order;
   order = atoi(argv[1]);
   blen = order;
   step(0);               // Print init. point.
   hilbert(0, 1, order);
   return 0;
}
```

FIGURE 16–4. Driver program for Hilbert curve generator.

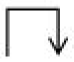
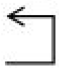
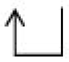
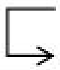
16–2 Coordinates from Distance along the Hilbert Curve

To find the (*x, y*) coordinates of a point located at a distance *s* along the order *n* Hilbert curve, observe that the most significant two bits of the 2*n*-bit integer *s* determine which major quadrant the point is in. This is because the Hilbert curve of any order follows the overall pattern of the order 1 curve. If the most significant two bits of *s* are 00, the point is somewhere in the lower-left quadrant, if 01 it is in the upper-left quadrant, if 10 it is in the upper-right quadrant, and if 11 it is in the lower-right quadrant. Thus, the most significant two bits of *s* determine the most significant bits of the *n*-bit integers *x* and *y*, as follows:

Most significant two bits of *s*	Most significant bits of (*x*, *y*)
00	(0, 0)
01	(0, 1)
10	(1, 1)
11	(1, 0)

In any Hilbert curve, only four of the eight possible U-shapes occur. These are shown in Table 16–1 as graphics and as maps from two bits of *s* to a single bit of each of *x* and *y*.

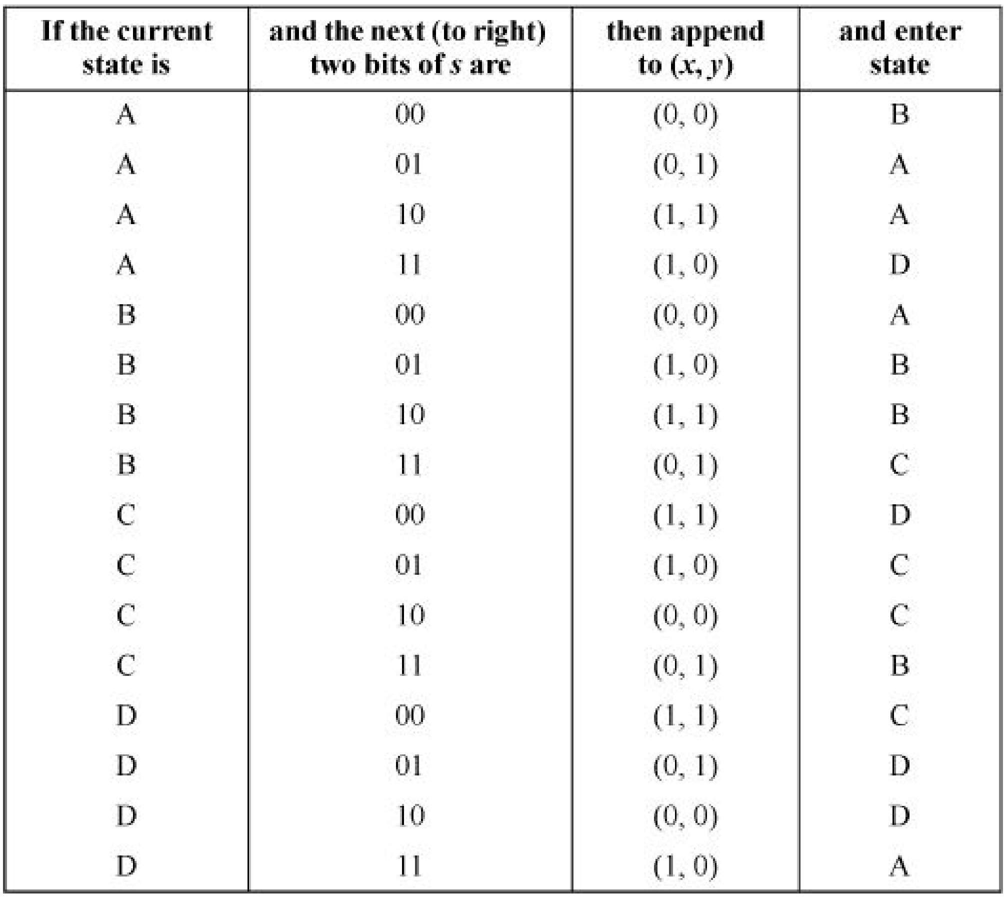
TABLE 16-1. THE FOUR POSSIBLE MAPPINGS

A	B	C	D
			
$00 \rightarrow (0, 0)$	$00 \rightarrow (0, 0)$	$00 \rightarrow (1, 1)$	$00 \rightarrow (1, 1)$
$01 \rightarrow (0, 1)$	$01 \rightarrow (1, 0)$	$01 \rightarrow (1, 0)$	$01 \rightarrow (0, 1)$
$10 \rightarrow (1, 1)$	$10 \rightarrow (1, 1)$	$10 \rightarrow (0, 0)$	$10 \rightarrow (0, 0)$
$11 \rightarrow (1, 0)$	$11 \rightarrow (0, 1)$	$11 \rightarrow (0, 1)$	$11 \rightarrow (1\ 0)$

Observe from Figure 16–2 that in all cases the U-shape represented by map A ( ) becomes, at the next level of detail, a U-shape represented by maps B, A, A, or D, depending on whether the length traversed in the first-mentioned map A is 0, 1, 2,
( )

or 3, respectively. Similarly, a U-shape represented by map B becomes, at the next level of detail, a U-shape represented by maps A, B, B, or C, depending on whether the length traversed in the first-mentioned map B is 0, 1, 2, or 3, respectively.

TABLE 16–2. STATE TRANSITION TABLE FOR COMPUTING (*X*, *Y*) FROMS



If the current state is	and the next (to right) two bits of *s* are	then append to (x, y)	and enter state
A	00	(0, 0)	B
A	01	(0, 1)	A
A	10	(1, 1)	A
A	11	(1, 0)	D
B	00	(0, 0)	A
B	01	(1, 0)	B
B	10	(1, 1)	B
B	11	(0, 1)	C
C	00	(1, 1)	D
C	01	(1, 0)	C
C	10	(0, 0)	C
C	11	(0, 1)	B
D	00	(1, 1)	C
D	01	(0, 1)	D
D	10	(0, 0)	D
D	11	(1, 0)	A

These observations lead to the state transition table shown in Table 16–2, in which the states correspond to the mappings shown in Table 16–1.

To use the table, start in state A. The integer *s* should be padded with leading zeros so that its length is *2n*, where *n* is the order of the Hilbert curve. Scan the bits of *s* in pairs from left to right. The first row of Table 16–2 means that if the current state is A and the currently scanned bits of *s* are 00, then output (0, 0) and enter state B. Then, advance to the next two bits of *s*. Similarly, the second row means that if the current state is A and the scanned bits are 01, then output (0, 1) and stay in state A.

The output bits are accumulated in left-to-right order. When the end of *s* is reached, the *n*-bit output quantities *x* and *y* are defined.

As an example, suppose $n = 3$ and

$$s = 110100.$$

Because the process starts in state A and the initial bits scanned are 11, the process outputs (1, 0) and enters state D (fourth row). Then, in state D and scanning 01, the process outputs (0, 1) and stays in state D. Lastly, the process outputs (1, 1) and enters state C, although the state is now immaterial.

Thus, the output is (101, 011)—that is, $x = 5$ and $y = 3$.

A C program implementing these steps is shown in Figure 16–5. In this program, the current state is represented by an integer from 0 to 3 for states A through D, respectively. In the assignment to variable `row`, the current state is concatenated with the next two bits of `s`, giving an integer from 0 to 15, which is the applicable row number in Table 16–2. Variable `row` is used to access integers (expressed in hexadecimal) that are used as bit strings to represent the rightmost two columns of Table 16–2; that is, these accesses are in-register table lookups. Left-to-right in the hexadecimal values corresponds to bottom-to-top in Table 16–2.

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                       unsigned *yp) {
    int i;
    unsigned state, x, y, row;

    state = 0;                          // Initialize.
    x = y = 0;

    for (i = 2*n - 2; i >= 0; i -= 2) { // Do n times.
       row = 4*state | (s >> i) & 3;    // Row in table.
       x = (x << 1) | (0x936C >> row) & 1;
       y = (y << 1) | (0x39C6 >> row) & 1;
       state = (0x3E6B94C1 >> 2*row) & 3; // New state.
   }
   *xp = x;                             // Pass back
   *yp = y;                             // results.
}
```

FIGURE 16–5. Program for computing (*x, y*) from *s*.

[L&S] give a quite different algorithm. Unlike the algorithm of Figure 16–5, it scans the bits of *s* from right to left. It is based on the observation that one can map the least significant two bits of *s* to (*x, y*) based on the order 1 Hilbert curve, and then test the next two bits of *s* to the left. If they are 00, the values of *x* and *y* just computed should be interchanged, which corresponds to reflecting the order 1 Hilbert curve about the line $x = y$. (Refer to the curves of orders 1 and 2 shown in Figure 16–1 on page 355.) If these two bits are 01 or 10, the values of *x* and *y* are not changed. If they are 11, the values of *x* and *y* are interchanged and complemented. These same rules apply as one progresses leftward along the bits of *s*. They are embodied in Table 16–3 and the code of Figure 16–6. It is somewhat curious that the bits can be prepended to *x* and *y* first, and then the swap and complement operations can be done, including these newly prepended bits; the results are the same.

TABLE 16–3. LAM AND SHAPIRO METHOD FOR COMPUTING (*X, Y*) FROM *S*

If the next (to left) two bits of *s* are	then	and prepend to (x, y)
00	Swap *x* and *y*	(0, 0)
01	No change	(0, 1)
10	No change	(1, 1)
11	Swap and complement *x* and *y*	(1, 0)

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                      unsigned *yp) {
   int i, sa, sb;
   unsigned x, y, temp;

   for (i = 0; i < 2*n; i += 2) {
      sa = (s >> (i+1)) & 1;        // Get bit i+1 of s.
      sb = (s >> i) & 1;            // Get bit i of s.

      if ((sa ^ sb) == 0) {         // If sa,sb = 00 or 11,
         temp = x;                  // swap x and y,
         x = y ^ (-sa);             // and if sa = 1,
         y = temp ^ (-sa);          // complement them.
      }
      x = (x >> 1) | (sa << 31);// Prepend sa to x and
      y = (y >> 1) | ((sa ^ sb) << 31); // (sa ^ sb) to y.
   }
   *xp = x >> (32 - n);            // Right-adjust x and y
   *yp = y >> (32 - n);            // and return them to
}                                   // the caller.
```

FIGURE 16–6. Lam and Shapiro method for computing (*x*, *y*) from *s*.

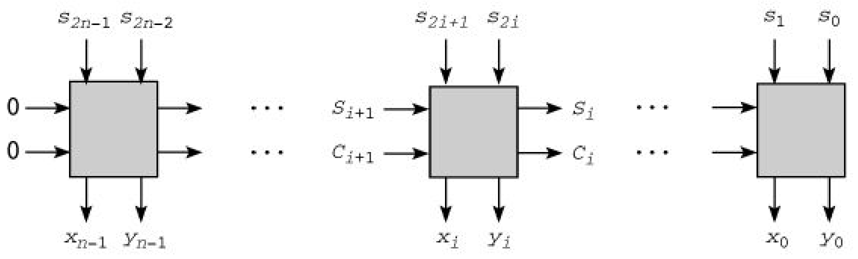
In Figure 16–6, variables `x` and `y` are uninitialized, which might cause an error message from some compilers, but the code functions correctly for whatever values `x` and `y` have initially.

The branch in the loop of Figure 16–6 can be avoided by doing the swap operation with the “three *exclusive or*” trick given in Section 2–20 on page 45. The *if* block can be replaced by the following code, where `swap` and `cmpl` are unsigned integers:

```
swap = (sa ^ sb) - 1;    // -1 if should swap, else 0.
cmpl = -(sa & sb);       // -1 if should compl’t, else 0.
x = x ^ y;
y = y ^ (x & swap) ^ cmpl;
x = x ^ y;
```

This is nine instructions, versus about two or six for the *if* block, so the branch cost would have to be quite high for this to be a good choice.

The “swap and complement” idea of [L&S] suggests a logic circuit for generating the Hilbert curve. The idea behind the circuit, described below, is that as you trace along the path of an order *n* curve, you basically map pairs of bits of *s* to (*x*, *y*) according to map A of Table 16–1. As the trace enters various regions, the mapping output gets swapped, complemented, or both. The circuit of Figure 16–7 keeps track of the swap and complement requirements of each stage, uses the appropriate mapping to map two bits of *s* to (x_i, y_i), and generates the swap and complement signals for the next stage.



$$
\begin{aligned}
x_i &= [s_{2i+1} \oplus (s_{2i}S_{i+1})] \oplus C_{i+1} \\
y_i &= [s_{2i} \oplus s_{2i+1} \oplus (s_{2i}S_{i+1})] \oplus C_{i+1} \\
S_i &= S_{i+1} \oplus (s_{2i} \equiv s_{2i+1}) \\
C_i &= C_{i+1} \oplus (s_{2i}s_{2i+1})
\end{aligned}
$$

FIGURE 16–7. Logic circuit for computing (*x*, *y*) from *s*.

Assume there is a register containing the path length *s* and circuits for incrementing it. Then, to find the next point on the Hilbert curve, first increment *s* and then transform it as described in Table 16–4. This is a left-to-right process, which is a bit of a problem because incrementing *s* is a right-to-left process. Thus, the time to generate a new point on an order *n* Hilbert curve is proportional to 2*n* (for incrementing *s*) plus *n* (for transforming *s* to (*x*, *y*)).

TABLE 16–4. LOGIC FOR COMPUTING (*X*, *Y*) FROM *S*

If the next (to right) two bits of *s* are	then append to (*x*, *y*)	and set
00	(0, 0)*	swap = $\overline{\text{swap}}$
01	(0, 1)*	No change
10	(1, 1)*	No change
11	(1, 0)*	swap = $\overline{\text{swap}}$, cmpl = $\overline{\text{cmpl}}$
* Possibly swapped and/or complemented		

Figure 16–7 shows this computation as a logic circuit. In this figure, *S* denotes the swap signal and *C* denotes the complement signal.

The logic circuit of Figure 16–7 suggests another way to compute (*x*, *y*) from *s*. Notice how the swap and complement signals propagate from left to right through the *n* stages. This suggests that it might be possible to use the parallel prefix operation to quickly (in $\log_2 n$ steps rather than *n* – 1) propagate the swap and complement information to each stage, and then do some word-parallel logical operations to compute *x* and *y*, using the equations in Figure 16–7. The values of *x* and *y* are intermingled in the even and odd bit positions of a word, so they have to be separated by the unshuffle operation (see page 140). This might seem a bit complicated, and likely to pay off only for rather large values of *n*, but let us see how it goes.

A procedure for this operation is shown in Figure 16–8 [GLS1]. The procedure operates on fullword quantities, so it first pads the input *s* on the left with '01' bits. This bit combination does not affect the swap and complement quantities. Next, a quantity `cs` (complement-swap) is computed. This word is of the form `cscs...cs`, where each `c` (a single bit), if 1, means that the corresponding pair of bits is to be complemented, and each `s` means that the corresponding pair of bits is to be swapped, following Table 16–3. In other words, these two statements map each pair of bits of *s* as follows:

s_{2i+1} s_{2i}	cs
0 0	01
0 1	00
1 0	00
1 1	11

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                      unsigned *yp) {
   unsigned comp, swap, cs, t, sr;

   s = s | (0x55555555 << 2*n); // Pad s on left with 01
   sr = (s >> 1) & 0x55555555;  // (no change) groups.
   cs = ((s & 0x55555555) + sr) // Compute complement &
        ^ 0x55555555;           // swap info in two-bit
                                // groups.
   // Parallel prefix xor op to propagate both complement
   // and swap info together from left to right (there is
   // no step "cs ^= cs >> 1", so in effect it computes
   // two independent parallel prefix operations on two
   // interleaved sets of sixteen bits).

   cs = cs ^ (cs >> 2);
   cs = cs ^ (cs >> 4);
   cs = cs ^ (cs >> 8);
   cs = cs ^ (cs >> 16);
   swap = cs & 0x55555555;      // Separate the swap and
   comp = (cs >> 1) & 0x55555555;  // complement bits.

   t = (s & swap) ^ comp;       // Calculate x and y in
   s = s ^ sr ^ t ^ (t << 1);   // the odd & even bit
                                // positions, resp.
   s = s & ((1 << 2*n) - 1);    // Clear out any junk
                                // on the left (unpad).

   // Now "unshuffle" to separate the x and y bits.

   t = (s ^ (s >> 1)) & 0x22222222; s = s ^ t ^ (t << 1);
   t = (s ^ (s >> 2)) & 0x0C0C0C0C; s = s ^ t ^ (t << 2);
   t = (s ^ (s >> 4)) & 0x00F000F0; s = s ^ t ^ (t << 4);
   t = (s ^ (s >> 8)) & 0x0000FF00; s = s ^ t ^ (t << 8);

   *xp = s >> 16;              // Assign the two halves
   *yp = s & 0xFFFF;           // of t to x and y.
}
```

FIGURE 16–8. Parallel prefix method for computing (*x*, *y*) from *s*.

This is the quantity to which we want to apply the parallel prefix operation. PP-XOR is the one to use, going from left to right, because successive 1-bits meaning to complement or to swap have the same logical properties as *exclusive or*: Two successive 1-bits cancel each other.

Both signals (complement and swap) are propagated in the same PP-XOR operation, each working with every other bit of `cs`.

The next four assignment statements have the effect of translating each pair of bits of `s` into (`x`, `y`) values, with `x` being in the odd (leftmost) bit positions, and `y` being in the even bit positions. Although the logic may seem obscure, it is not difficult to verify that each pair of bits of `s` is transformed by the logic of the first two Boolean equations in Figure 16–7. (*Suggestion:* Consider separately how the even and odd bit positions are transformed, using the fact that `t` and `sr` are 0 in the odd positions.)

The rest of the procedure is self-explanatory. It executes in 66 basic RISC instructions (constant, branch-free), versus about $19n + 10$ (average) for the code of Figure 16–6 (based on compiled code; includes prologs and epilogs, which are essentially nil). Thus, the parallel prefix method is faster for $n \geq 3$.

16–3 Distance from Coordinates on the Hilbert Curve

Given the coordinates of a point on the Hilbert curve, the distance from the origin to the point can be calculated by means of a state transition table similar to Table 16–2. Table 16–5 is such a table.

TABLE 16–5. STATE TRANSITION TABLE FOR COMPUTING *S* FROM (*X*, *Y*)

If the current state is	and the next (to right) two bits of (x, y) are	then append to s	and enter state
A	(0, 0)	00	B
A	(0, 1)	01	A
A	(1, 0)	11	D
A	(1, 1)	10	A
B	(0, 0)	00	A
B	(0, 1)	11	C
B	(1, 0)	01	B
B	(1, 1)	10	B
C	(0, 0)	10	C
C	(0, 1)	11	B
C	(1, 0)	01	C
C	(1, 1)	00	D
D	(0, 0)	10	D
D	(0, 1)	01	D
D	(1, 0)	11	A
D	(1, 1)	00	C

Its interpretation is similar to that of the previous section. First, *x* and *y* should be padded with leading zeros so that they are of length *n* bits, where *n* is the order of the Hilbert curve. Second, the bits of *x* and *y* are scanned from left to right, and *s* is built up from left to right.

A C program implementing these steps is shown in Figure 16–9.

```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i;
   unsigned state, s, row;

   state = 0;                          // Initialize.
   s = 0;

   for (i = n - 1; i >= 0; i--) {
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      s = (s << 2) | (0x361E9CB4 >> 2*row) & 3;
      state = (0x8FE65831 >> 2*row) & 3;
   }
   return s;
}
```

FIGURE 16–9. Program for computing *s* from (*x*, *y*).

[L&S] give an algorithm for computing *s* from (*x, y*) that is similar to their algorithm for going in the other direction (Table 16–3). It is a left-to-right algorithm, shown in

Table 16–6 and Figure 16–10.

TABLE 16–6. LAM AND SHAPIRO METHOD FOR COMPUTING *S* FROM (*X, Y*)

If the next (to right) two bits of (*x*, *y*) are	then	and append to *s*
(0, 0)	Swap *x* and *y*	00
(0, 1)	No change	01
(1, 0)	Swap and complement *x* and *y*	11
(1, 1)	No change	10

```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i, xi, yi;
   unsigned s, temp;

   s = 0;                           // Initialize.
   for (i = n - 1; i >= 0; i--) {
      xi = (x >> i) & 1;             // Get bit i of x.
      yi = (y >> i) & 1;            // Get bit i of y.

      if (yi == 0) {
         temp = x;                  // Swap x and y and,
         x = y^(-xi);               // if xi = 1,
         y = temp^(-xi);            // complement them.
      }
      s = 4*s + 2*xi + (xi^yi);     // Append two bits to s.
   }
   return s;
}
```

FIGURE 16–10. Lam and Shapiro method for computing *s* from (*x, y*).

16–4 Incrementing the Coordinates on the Hilbert Curve

Given the (*x, y*) coordinates of a point on the order *n* Hilbert curve, how can one find the coordinates of the next point? One way is to convert (*x, y*) to *s*, add 1 to *s*, and then convert the new value of *s* back to (*x, y*), using algorithms given above.

A slightly (but not dramatically) better way is based on the fact that as one moves along the Hilbert curve, at each step either *x* or *y*, but not both, is either incremented or decremented (by 1). The algorithm to be described scans the coordinate numbers from left to right to determine the type of U-curve that the rightmost two bits are on. Then, based on the U-curve and the value of the rightmost two bits, it increments or decrements either *x* or *y*.

That's basically it, but there is a complication when the path is at the end of a U-curve (which happens once every four steps). At this point, the direction to take is determined by the *previous* bits of *x* and *y* and by the higher order U-curve with which these bits are associated. If that point is also at the end of its U-curve, then the previous bits and the U-curve there determine the direction to take, and so on.

Table 16–7 describes this algorithm. In this table, the A, B, C, and D denote the U-curves as shown in Table 16–1 on page 360. To use the table, first pad *x* and *y* with leading zeros so they are *n* bits long, where *n* is the order of the Hilbert curve. Start in state A and scan the bits of *x* and *y* from left to right. The first row of Table 16–7 means that if the current state is A and the currently scanned bits are (0, 0), then set a variable to indicate to increment *y*, and enter state B. The other rows are interpreted similarly, with a suffix minus sign indicating to decrement the associated coordinate. A dash in the third column means do not alter the variable that keeps track of the coordinate changes.

TABLE 16–7. TAKING ONE STEP ON THE HILBERT CURVE

If the current state is	and the next (to right) two bits of (x, y) are	then prepare to inc/dec	and enter state
A	(0, 0)	$y+$	B
A	(0, 1)	$x+$	A
A	(1, 0)	—	D
A	(1, 1)	$y-$	A
B	(0, 0)	$x+$	A
B	(0, 1)	—	C
B	(1, 0)	$y+$	B
B	(1, 1)	$x-$	B
C	(0, 0)	$y+$	C
C	(0, 1)	—	B
C	(1, 0)	$x-$	C
C	(1, 1)	$y-$	D
D	(0, 0)	$x+$	D
D	(0, 1)	$y-$	D
D	(1, 0)	—	A
D	(1, 1)	$x-$	C

After scanning the last (rightmost) bits of *x* and *y*, increment or decrement the appropriate coordinate as indicated by the final value of the variable.

A C program implementing these steps is shown in Figure 16–11. Variable `dx` is initialized in such a way that if invoked many times, the algorithm cycles around, generating the same Hilbert curve over and over again. (However, the step that connects one cycle to the next is not a unit step.)

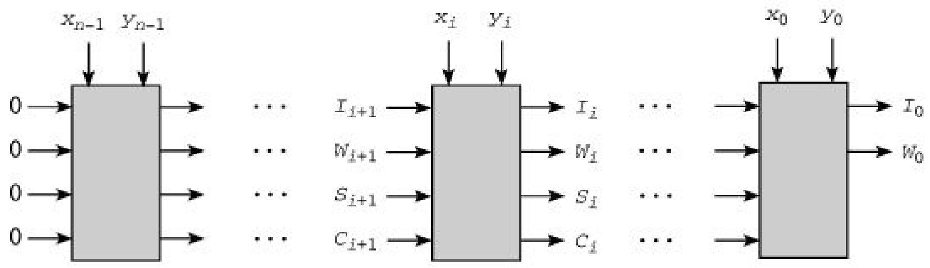
```
void hil_inc_xy(unsigned *xp, unsigned *yp, int n) {

   int i;
   unsigned x, y, state, dx, dy, row, dochange;

   x = *xp;
   y = *yp;
   state = 0;                     // Initialize.
   dx = -((1 << n) - 1);        // Init. -(2**n - 1).
   dy = 0;

   for (i = n-1; i >= 0; i--) {         // Do n times.
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      dochange = (0xBDDB >> row) & 1;
      if (dochange) {
         dx = ((0x16451659 >> 2*row) & 3) - 1;
         dy = ((0x51166516 >> 2*row) & 3) - 1;
      }
      state = (0x8FE65831 >> 2*row) & 3;
   }
   *xp = *xp + dx;
   *yp = *yp + dy;
}
```

FIGURE 16–11. Program for taking one step on the Hilbert curve.

Table 16–7 can readily be implemented in logic, as shown in Figure 16–12. In this figure, the variables have the following meanings:

x_i:	Bit i of input x
y_i:	Bit i of input y
X, Y:	x_i and y_i swapped and complemented, according to S_{i+1} and C_{i+1}
I:	If 1, increment; if 0, decrement (by 1)
W:	If 1, increment or decrement x; if 0, increment or decrement y
S:	If 1, swap x_i and y_i
C:	If 1, complement x_i and y_i

S and C together identify the "state" of Table 16–7, with (C, S) = (0,0), (0,1), (1,0), and (1,1) denoting states A, B, C, and D, respectively. The output signals are I_0 and W_0, which tell, respectively, whether to increment or decrement, and which variable to change. (In addition to the logic shown, an incrementer/decrementer circuit is required, with MUX's to route either x or y to the incrementer/decrementer, and a circuit to route the altered value back to the register that holds x or y. Alternatively, two incrementer/decrementer circuits could be used.)



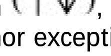
$$
\begin{aligned}
X &= (S_{i+1}y_i + \overline{S_{i+1}}x_i) \oplus C_{i+1} \\
Y &= (S_{i+1}x_i + \overline{S_{i+1}}y_i) \oplus C_{i+1} \\
I_i &= \overline{C_{i+1}}\bar{X} + C_{i+1}XY + I_{i+1}X\bar{Y} \\
W_i &= \overline{S_{i+1}}\bar{X}Y + S_{i+1}(X \equiv Y) + W_{i+1}X\bar{Y} \\
S_i &= S_{i+1} \equiv Y \\
C_i &= C_{i+1} \oplus (X\bar{Y})
\end{aligned}
$$

FIGURE 16–12. Logic circuit for incrementing *(x, y)* by one step along the Hilbert curve.

16–5 Non-Recursive Generating Algorithms

The algorithms of Tables 16–2 and 16–7 provide two non-recursive algorithms for generating the Hilbert curve of any order. Either algorithm can be implemented in hardware without great difficulty. Hardware based on Table 16–2 includes a register holding *s*, which it increments for each step, and then converts to (*x, y*) coordinates. Hardware based on Table 16–7 would not have to include a register for *s*, but the algorithm is more complicated.

16–6 Other Space-Filling Curves

As was mentioned, Peano was first, in 1890, to discover a space-filling curve. The many variations discovered since then are often called "Peano curves." One interesting variation of Hilbert's curve was discovered by Eliakim Hastings Moore in 1900. It is "cyclic" in the sense that the end point is one step away from the starting point. The Peano curve of order 3, and the Moore curve of order 4, are shown in Figure 16–13. Moore's curve has an irregularity in that the order 1 curve is upright-down , but this shape does not appear in the higher-order curves. Except for this minor exception, the algorithms for dealing with Moore's curve are very similar to those for the Hilbert curve.

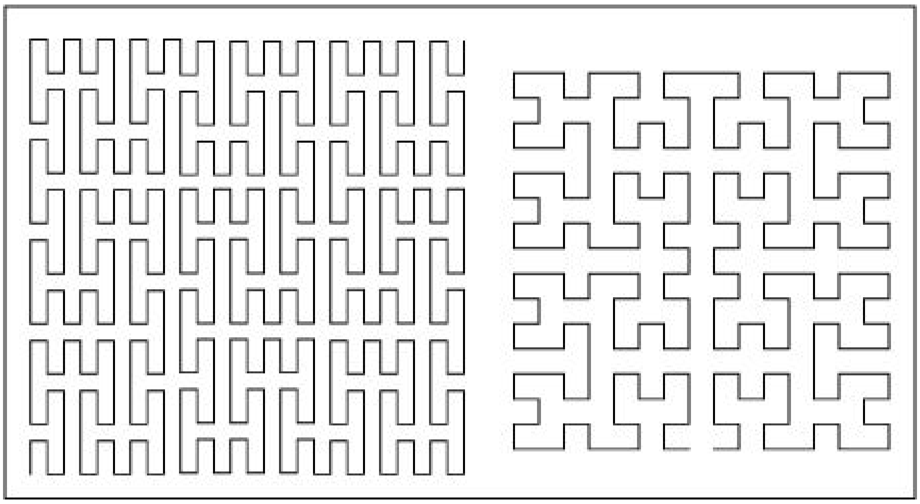
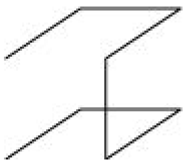


FIGURE 16–13. Peano (left) and Moore (right) curves.

The Hilbert curve has been generalized to arbitrary rectangles and to three and higher dimensions. The basic building block for a three-dimensional Hilbert curve is shown below. It hits all eight points of a 2×2×2 cube. These and many other space-filling curves are discussed in [Sagan].



16–7 Applications

Space-filling curves have applications in image processing: compression, halftoning, and textural analysis [L&S]. Another application is to improve computer performance in ray tracing, a graphics-rendering technique. Conventionally, a scene is scanned by projecting rays across the scene in ordinary raster scan line order (left to right across the screen, and then top to bottom). When a ray hits an object in the simulated scene's database, the color and other properties of the object at that point are determined, and the results are used to illuminate the pixel through which the ray was sent. (This is an oversimplification, but it's adequate for our purposes.) One problem is that the database is often large and the data on each object must be paged in and cast out as various objects are hit by the scanning ray. When the ray scans across a line, it often hits many objects that were hit in the previous scan, requiring them to be paged in again. Paging operations would be reduced if the scanning had some kind of locality property. For example, it might be helpful to scan a quadrant of the screen completely before going on to another quadrant.

The Hilbert curve seems to have the locality property we are seeking. It scans a quadrant completely before scanning another, recursively, and also does not make a long jump when going from one quadrant to another.

Douglas Voorhies [Voor] has simulated what the paging behavior would likely be for the conventional uni-directional scan line traversal, the Peano curve, and the Hilbert curve. His method is to scatter circles of a given size randomly on the screen. A scan path hitting a circle represents touching a new object, and paging it in. When a scan leaves a circle, it is presumed that the object's data remains in memory until the scan exits a circle of radius twice that of the "object" circle. Thus, if the scan leaves the object for just a short distance and then returns to it, it is assumed that no paging operation occurred. He repeats this experiment for many different sizes of circles, on a simulated 1024×1024 screen.

Assume that entering an object circle and leaving its surrounding circle represent one paging operation. Then, clearly the normal scan line causes D paging operations in covering a (not too big) circle of diameter D pixels, because each scan line that enters it leaves its outer circle. The interesting result of Voorhies's simulation is that for the Peano curve, the number of paging operations to scan a circle is about 2.7 and, perhaps surprisingly, is independent of the circle's diameter. For the Hilbert curve, the figure is about 1.4, also independent of the circle's diameter. Thus, the experiment suggests that the Hilbert curve is superior to the Peano curve, and vastly superior to the normal scan line path, in reducing paging operations. (The result that the page count is independent of the circles' diameters is probably an artifact of the outer circle's being proportional in size to the object circle.)

The Hilbert curve has been used to assign jobs to processors when the processors are interconnected in a rectangular 2D or 3D grid [Cplant]. The processor allocation system software uses a linear list of the processors that follows a Hilbert curve over the grid. When a job that requires a number of processors is scheduled to run, the allocator allocates them from the linear list, much as a memory allocator would do. The allocated processors tend to be close together on the grid, which leads to good intercommunication properties.

Exercises

1. A simple way to cover an $n \times n$ grid in a way that doesn't make too many big jumps, and hits every point once and only once, is to have a $2n$-bit variable s that is incremented at each step, and form x from the first and every other bit of s, and y from the second and every other bit of s. This is equivalent to computing the perfect outer unshuffle of s, and then letting x and y be the left and right halves of the result. Investigate this curve's locality property by sketching the curve for $n = 3$.

2. A variation of exercise 1 is to first transform s into Gray(s) (see page 312), and then let x and y be formed from every other bit of the result, as in exercise 1. Sketch the curve for $n = 3$. Has this improved the locality property?

3. How would you construct a three-dimensional analog of the curve of exercise 1?

Chapter 17. Floating-Point

God created the integers,
all else is the work of man.

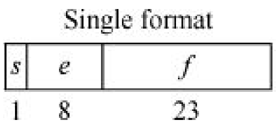
Leopold Kronecker

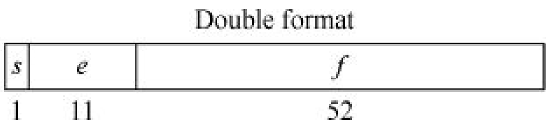
Operating on floating-point numbers with integer arithmetic and logical instructions is often a messy proposition. This is particularly true for the rules and formats of the *IEEE Standard for Floating-Point Arithmetic*, IEEE Std. 754-2008, commonly known as "IEEE arithmetic." It has the NaN (not a number) and infinities, which are special cases for almost all operations. It has plus and minus zero, which must compare equal to one another. It has a fourth comparison result, "unordered." The most significant bit of the fraction is not explicitly present in "normal" numbers, but it is in "subnormal" numbers. The fraction is in signed-true form and the exponent is in biased form, whereas integers are now almost universally in two's-complement form. There are, of course, reasons for all this, but it results in programs that deal with the representation being full of tests and branches, and that present a challenge to implement efficiently.

We assume the reader has some familiarity with the IEEE standard, and summarize it here only very briefly.

17–1 IEEE Format

The 2008 standard includes three binary and two decimal formats. We will restrict our attention to the binary "single" and "double" formats (32- and 64-bit). These are shown below.





The sign bit s is encoded as 0 for plus, 1 for minus. The biased exponent e and fraction f are magnitudes with their most significant bits on the left. The floating-point value represented is encoded as shown on the next page.

Single format

e	f	*value*
0	0	± 0
0	$\neq 0$	$\pm 2^{-126}(0.f)$
1 to 254	–	$\pm 2^{e-127}(1.f)$
255	0	$\pm\infty$
255	$\neq 0$	NaN

Double format

e	f	*value*
0	0	± 0
0	$\neq 0$	$\pm 2^{-1022}(0.f)$
1 to 2046	–	$\pm 2^{e-1023}(1.f)$
2047	0	$\pm\infty$
2047	$\neq 0$	NaN

As an example, consider encoding the number π in single format. In binary [Knu1],

π ≈ 11.0010 0100 0011 1111 0110 1010 1000 1000 1000 0101 1010 0011 0000 10....

This is in the range of the "normal" numbers shown in the third row of the table above. The most significant 1 in π is dropped, as the leading 1 is not stored in the encoding of normal numbers. The exponent $e - 127$ should be 1, to get the binary point in the right place, and hence $e = 128$. Thus, the representation is

```
0 10000000 10010010000111111011011
```

or, in hexadecimal,

```
40490FDB,
```

where we have rounded the fraction to the nearest representable number.

Numbers with $1 \le e \le 254$ are the "normal numbers." These are "normalized," meaning that their most significant bit is 1 and it is not explicitly stored. Nonzero numbers with $e = 0$ are called "subnormal numbers," or simply "subnormals." Their most significant bit *is* explicitly stored. This scheme is sometimes called "gradual underflow." Some extreme values in the various ranges of floating-point numbers are shown in Table 17–1. In this table, "Max integer" means the largest integer such that all integers less than or equal to it, in absolute value, are representable exactly; the next integer is rounded.

For normal numbers, one unit in the last position (ulp) has a relative value ranging from $1 / 2^{24}$ to $1 / 2^{23}$ (about 5.96×10^{-8} to 1.19×10^{-7}) for single format, and from $1 / 2^{53}$ to $1 / 2^{52}$ (about 1.11×10^{-16} to 2.22×10^{-16}) for double format. The maximum "relative error," for round to nearest mode, is half of those figures.

The range of integers that is represented exactly is from -2^{24} to $+2^{24}$(–16,777,216 to +16,777,216) for single format, and from -2^{53} to $+2^{53}$(–9,007,199,254,740,992 to +9,007,199,254,740,992) for double format. Of course, certain integers outside these ranges, such as larger powers of 2, can be represented exactly; the ranges cited are the maximal ranges for which *all* integers are represented exactly.

TABLE 17–1. EXTREME VALUES

Single Precision			
	Hex	**Exact Value**	**Approximate Value**
Smallest subnormal	`0000 0001`	2^{-149}	1.401×10^{-45}
Largest subnormal	`007F FFFF`	$2^{-126}(1 - 2^{-23})$	1.175×10^{-38}
Smallest normal	`0080 0000`	2^{-126}	1.175×10^{-38}
1.0	`3F80 0000`	1	1
Max integer	`4B80 0000`	2^{24}	1.677×10^{7}
Largest normal	`7F7F FFFF`	$2^{128}(1 - 2^{-24})$	3.403×10^{38}
∞	`7F80 0000`	∞	∞
Double Precision			
Smallest subnormal	`0...0001`	2^{-1074}	4.941×10^{-324}
Largest subnormal	`000F...F`	$2^{-1022}(1 - 2^{-52})$	2.225×10^{-308}
Smallest normal	`0010...0`	2^{-1022}	2.225×10^{-308}
1.0	`3FF0...0`	1	1
Max integer	`4340...0`	2^{53}	9.007×10^{15}
Largest normal	`7FEF...F`	$2^{1024}(1 - 2^{-53})$	1.798×10^{308}
∞	`7FF0...0`	∞	∞

One might want to change division by a constant to multiplication by the reciprocal. This can be done with complete (IEEE) accuracy only for numbers whose reciprocals are represented exactly. These are the powers of 2 from 2^{-127} to 2^{127} for single format, and from 2^{-1023} to 2^{1023} for double format. The numbers 2^{-127} and 2^{-1023} are subnormal numbers, which are best avoided on machines that implement operations on subnormal numbers inefficiently.

17–2 Floating-Point To/From Integer Conversions

Table 17–2 gives some formulas for conversion between IEEE floating-point format and integers. These methods are concise and fast, but they do not give the correct result for the full range of input values. The ranges over which they do give the precisely correct result are given in the table. They all give the correct result for ±0.0 and for subnormals within the stated ranges. Most do not give a reasonable result for a NaN or infinity. These formulas may be suitable for direct use in some applications, or in a library routine to get the common cases quickly.

TABLE 17–2. FLOATING-POINT CONVERSIONS

Type	R	Formula	Range	Notes
Double to int64, n	n	$(x \overset{d}{+} c_{521}) - c_{521}$	-2^{51} to $2^{51} + 0.5$	1
Double to int64, d	d	$(x \overset{d}{+} c_{521}) - c_{521}$	$-2^{51} - 0.5$ to $2^{51} + 0.5$	1
Double to int64, u	u	$(x \overset{d}{+} c_{521}) - c_{521}$	-2^{51} to $2^{51} + 1$	1

Double to int64, z	d or z	if ($x \geq 0.0$)
		  $(x \overset{d}{+} c_{52}) - c_{52}$
		else
		  $c_{52} - (c_{52} \overset{d}{-} x)$
			-2^{52} to 2^{52}	1

Double to uint64, n	n	$(x \overset{d}{+} c_{52}) - c_{52}$	-0.25 to 2^{52}	1
Double to uint64, d	d	$(x \overset{d}{+} c_{52}) - c_{52}$	0 to 2^{52}	1
Double to uint64, u	u	$(x \overset{d}{+} c_{52}) - c_{52}$	$-0.5 +$ ulp to $2^{52} + 1$	1
Double to int32 or uint32, n	n	low32$(x \overset{d}{+} c_{521})$	$-2^{31} - 0.5$ to $2^{31} - 0.5 -$ ulp, or -0.5 to $2^{32} - 0.5 -$ ulp	1
Double to int32 or uint32, d	d	low32$(x \overset{d}{+} c_{521})$	-2^{31} to $2^{31} -$ ulp, or 0 to $2^{32} -$ ulp	1

Double to int32 or uint32, u	u	low32($x \overset{d}{+} c_{521}$)	$-2^{31}-1+$ ulp to $2^{31}-1$, or $-1+$ ulp to $2^{32}-1$	1

Double to int32 or uint32, z	d or z	if ($x \ge 0.0$)
		low32($x \overset{d}{+} c_{521}$)
		else
		–low32($c_{521} \overset{d}{-} x$)
			$-2^{31}-1+$ ulp to $2^{31}-$ ulp,
			or $-1+$ ulp to $2^{32}-$ ulp
				1

Float to int32, n	n	$(x \overset{s}{+} c_{231}) - c_{231}$	-2^{22} to $2^{22}+0.5$	
Float to int32, d	d	$(x \overset{s}{+} c_{231}) - c_{231}$	$-2^{22}-0.5$ to $2^{22}+0.5$	
Float to int32, u	u	$(x \overset{s}{+} c_{231}) - c_{231}$	-2^{22} to $2^{22}+1$	

Float to int32, z	d or z	if ($x \ge 0.0$)
		$(x \overset{s}{+} c_{23}) - c_{23}$
		else
		$c_{23} - (c_{23} \overset{s}{-} x)$
			-2^{23} to 2^{23}	

Float to uint32, n	n	$(x \overset{s}{+} c_{23}) - c_{23}$	-0.25 to 2^{23}	
Float to uint32, d	d	$(x \overset{s}{+} c_{23}) - c_{23}$	0 to 2^{23}	
Float to uint32, u	u	$(x \overset{s}{+} c_{23}) - c_{23}$	$-0.5+$ ulp to $2^{23}+1$	
Round double to nearest	n	$(x \overset{d}{+} c_{521}) \overset{d}{-} c_{521}$	-2^{51} to $2^{51}+0.5$	1
Round non-negative double to nearest	n	$(x \overset{d}{+} c_{52}) \overset{d}{-} c_{52}$	-0.25 to 2^{52}	1, 3
Round float to nearest	n	$(x \overset{s}{+} c_{231}) \overset{s}{-} c_{231}$	-2^{22} to $2^{22}+0.5$	2
Round non-negative float to nearest	n	$(x \overset{s}{+} c_{23}) \overset{s}{-} c_{23}$	-0.25 to 2^{23}	2, 3
Int64 to double	–	$(x + c_{521}) \overset{d}{-} c_{521}$	-2^{51} to 2^{51}	4

Uint64 to double	–	$(x + c_{52}) \overset{d}{-} c_{52}$	0 to $2^{52} - 1$	4
Int32 to float	–	$(x + c_{231}) \overset{s}{-} c_{231}$	-2^{22} to 2^{22}	
Uint32 to float	–	$(x + c_{23}) \overset{s}{-} c_{23}$	0 to 2^{23}	

Constants:

$$\begin{aligned} c_{521} &= \text{0x4338000000000000} = 2^{52} + 2^{51} \\ c_{52} &= \text{0x4330000000000000} = 2^{52} \\ c_{231} &= \text{0x4B400000} = 2^{23} + 2^{22} \\ c_{23} &= \text{0x4B000000} = 2^{23} \end{aligned}$$

Notes:

1. The floating-point operations must be done in IEEE double-precision (53 bits of precision) and no more. Most Intel machines do not, by default, operate in this mode. On those machines it is necessary to set the precision (PC field in the FPU Control Word) to double-precision.
2. The floating-point operations must be done in IEEE single-precision (24 bits of precision) and no more. Most Intel machines are not, by default, operated in this mode. On those machines it is necessary to set the precision (PC field in the FPU Control Word) to single-precision.
3. "Nonnegative" means –0.0 or greater than or equal to 0.0.
4. To convert a 32-bit signed or unsigned integer to double, sign- or zero-extend the 32-bit integer to 64 bits and use the appropriate one of these formulas.

The Type column denotes the type of conversion desired, including the rounding mode: n for round to nearest even, d for round down, u for round up, and z for round toward zero. The R column denotes the rounding mode that the machine must be in for the formula to give the correct result. (On some machines, such as the Intel IA-32, the rounding mode can be specified in the instruction itself, rather than in a "mode" register.)

A "double" is an IEEE double, which is 64 bits in length. A "float" is an IEEE single, which is 32 bits in length.

The notation "ulp" means one unit in the last position. For example, 1.0 – ulp denotes the IEEE-format number that is closest to 1.0 but less than 1.0, something like 0.99999.... The notation "int64" denotes a signed 64-bit integer (two's-complement), and "int32" denotes a signed 32-bit integer. "uint64" and "uint32" have similar meanings, but for unsigned interpretations.

The function low32(*x*) extracts the low-order 32 bits of *x*.

The operators $\overset{d}{+}$ and $\overset{s}{+}$ denote double- and single-precision floating-point addition, respectively. Similarly, the operators $\overset{d}{-}$ and $\overset{s}{-}$ denote double- and single-precision subtraction.

It might seem curious that on most Intel machines the double to integer (of any size) conversions require that the machine's precision mode be reduced to 53 bits, whereas for *float* to integer conversions, the reduction in precision is not necessary—the correct result is obtained with the machine running in extended-precision mode (64 bits of precision). This is because for the double-precision add of the constant, the fraction might be shifted right as many as 52 bits, which may cause 1-bits to be shifted beyond the 64-bit limit, and hence lost. Thus, two roundings occur—first to 64 bits and then to 53 bits. On the other hand, for the single-precision add of the constant, the maximum shift is 23 bits. With that small shift amount, no bit can be shifted beyond the 64-bit boundary, so that only one rounding operation occurs. The conversions from float to integer get the correct result on Intel machines in all three precision modes.

$$((x \overset{e}{+} c_1) \overset{e}{-} c_2) - c_3,$$

On Intel machines running in extended-precision mode, the conversions from double to int64 and uint64 can be done without changing the precision mode by using different constants and one more floating-point operation. The calculation is where $\overset{e}{+}$ and $\overset{e}{-}$ denote extended-precision addition and subtraction, respectively. (The result of the *add* must remain in the 80-bit register for use by the extended-precision subtract operation.)

For double to int64,

c_1 = 0x43E00300 00000000 = $2^{63} + 2^{52} + 2^{51}$

c_2 = 0x43E00000 00000000 = 2^{63}

c_3 = 0x43380000 00000000 = $2^{52} + 2^{51}$.

For double to uint64,

c_1 = 0x43E00200 00000000 = $2^{63} + 2^{52}$

c_2 = 0x43E00000 00000000 = 2^{63}

c_3 = 0x43300000 00000000 = 2^{52}.

Using these constants, similar expressions can be derived for the conversion and rounding operations shown in Table 17–2 that are flagged by Note 1. The ranges of applicability are close to those shown in the table.

However, for the round double to nearest operation, if the calculation subtracts first and then adds, that is,

$$((x \overset{e}{-} c_1) \overset{e}{+} c_2) + c_3$$

(using the first set of constants above), then the range for which the correct result is obtained is $-2^{51} - 0.5$ to ∞, but not a NaN.

17–3 Comparing Floating-Point Numbers Using Integer Operations

One of the features of the IEEE encodings is that non-NaN values are properly ordered if treated as signed magnitude integers.

To program a floating-point comparison using integer operations, it is necessary that the “unordered” result not be needed. In IEEE 754, the unordered result occurs when one or both comparands are NaNs. The methods below treat NaNs as if they were numbers greater in magnitude than infinity.

The comparisons are also much simpler if -0.0 can be treated as strictly less than +0.0 (which is not in accordance with IEEE 754). Assuming this is acceptable, the comparisons can be done as shown below, where $\overset{f}{<}$, $\overset{f}{\le}$, and $\overset{f}{=}$ denote floating-point comparisons, and the ≈ symbol is used as a reminder that these formulas do not treat

±0.0 quite right. These comparisons are the same as IEEE 754-2008's "total-ordering" predicate.

$$a \overset{f}{=} b \approx (a = b)$$

$$a \overset{f}{<} b \approx (a \geq 0 \;\&\; a < b) \mid (a < 0 \;\&\; a \overset{u}{>} b)$$

$$a \overset{f}{\leq} b \approx (a \geq 0 \;\&\; a \leq b) \mid (a < 0 \;\&\; a \overset{u}{\geq} b)$$

If -0.0 must be treated as equal to +0.0, there does not seem to be any slick way to do it, but the following formulas, which follow more or less obviously from the above, are possibilities.

$$\begin{aligned} a \overset{f}{=} b &\equiv (a = b) \mid (-a = a \;\&\; -b = b) \\ &\equiv (a = b) \mid ((a \mid b) = \mathrm{0x80000000}) \\ &\equiv (a = b) \mid (((a \mid b) \;\&\; \mathrm{0x7FFFFFFF}) = 0) \end{aligned}$$

$$a \overset{f}{<} b \equiv ((a \geq 0 \;\&\; a < b) \mid (a < 0 \;\&\; a \overset{u}{>} b)) \;\&\; ((a \mid b) \neq \mathrm{0x80000000})$$

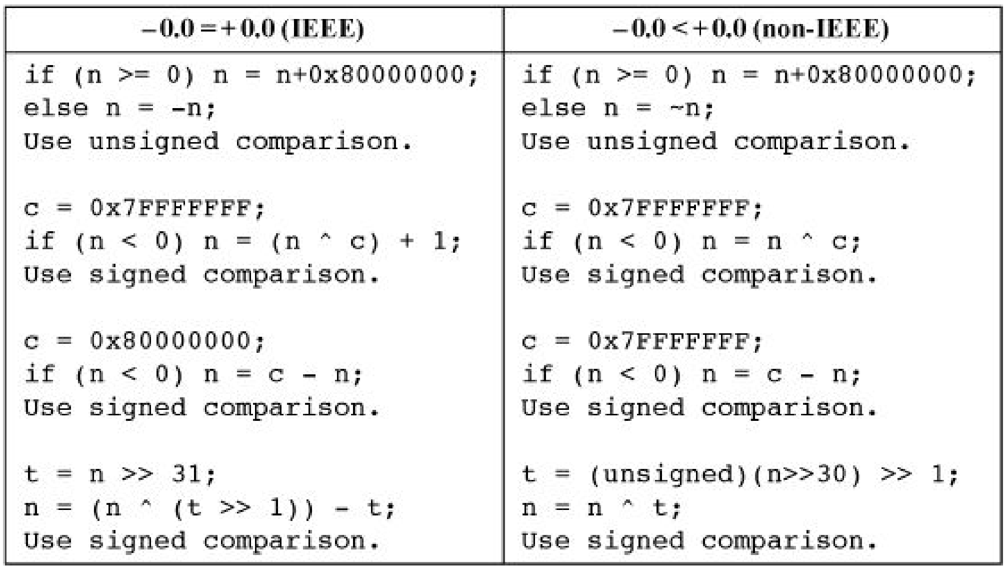
$$a \overset{f}{\leq} b \equiv (a \geq 0 \;\&\; a \leq b) \mid (a < 0 \;\&\; a \overset{u}{\geq} b) \mid ((a \mid b) = \mathrm{0x80000000})$$

In some applications, it might be more efficient to first transform the numbers in some way, and then do a floating-point comparison with a single fixed-point comparison instruction. For example, in sorting *n* numbers, the transformation would be done only once to each number, whereas a comparison must be done at least $\lceil n\log_2 n \rceil$ times (in the minimax sense).

Table 17–3 gives four such transformations. For those in the left column, -0.0 compares equal to +0.0, and for those in the right column, -0.0 compares less than +0.0. In all cases, the sense of the comparison is not altered by the transformation. Variable `n` is signed, `t` is unsigned, and `c` may be either signed or unsigned.

The last row shows branch-free code that can be implemented on our basic RISC in four instructions for the left column, and three for the right column (these four or three instructions must be executed for each comparand).

TABLE 17-3. PRECONDITIONING FLOATING-POINT NUMBERS FOR INTEGER COMPARISONS



−0.0 = +0.0 (IEEE)	−0.0 < +0.0 (non-IEEE)

`if (n >= 0) n = n+0x80000000;`
`else n = -n;`
`Use unsigned comparison.`
	`if (n >= 0) n = n+0x80000000;`
	`else n = ~n;`
	`Use unsigned comparison.`

`c = 0x7FFFFFFF;`
`if (n < 0) n = (n ^ c) + 1;`
`Use signed comparison.`
	`c = 0x7FFFFFFF;`
	`if (n < 0) n = n ^ c;`
	`Use signed comparison.`

`c = 0x80000000;`
`if (n < 0) n = c - n;`
`Use signed comparison.`
	`c = 0x7FFFFFFF;`
	`if (n < 0) n = c - n;`
	`Use signed comparison.`

`t = n >> 31;`
`n = (n ^ (t >> 1)) - t;`
`Use signed comparison.`
	`t = (unsigned)(n>>30) >> 1;`
	`n = n ^ t;`
	`Use signed comparison.`

17–4 An Approximate Reciprocal Square Root Routine

In the early 2000s, there was some buzz in programming circles about an amazing routine for computing an approximation to the reciprocal square root of a number in IEEE single format. The routine is useful in graphics applications, for example, to normalize a vector by multiplying its components *x*, *y*, and *z* by $1/\sqrt{x^2 + y^2 + z^2}$. C code for the function is shown in Figure 17–1 [Taro].

The relative error of the result is in the range 0 to -0.00176 for all normal single-precision numbers (it errs on the low side). It gives the correct IEEE result (NaN) if its argument is a NaN. However, it gives an unreasonable result if its argument is ±∞, a negative number, or -0. If the argument is +0 or a positive subnormal, the result is not what it should be, but it is a large number (greater than 9 x 10^{18}), which might be acceptable in some applications.

The relative error can be reduced in magnitude, to the range ±0.000892, by changing the constant 1.5 in the Newton step to 1.5008908.

Another possible refinement is to replace the multiplication by 0.5 with a subtract of 1 from the exponent of `x`. That is, replace the definition of `xhalf` with

```
union {int ihalf; float xhalf;};
ihalf = ix - 0x00800000;
```

However, the function then gives inaccurate results (although greater than 6 × 10^{18}) for `x` a normal number less than about 2.34 × 10^{-38}, and NaN for `x` a subnormal number. For `x` = 0 the result is ±∞ (which is correct).

The Newton step is a standard Newton-Raphson calculation for the reciprocal square root function (see Appendix B). Simply repeating this step reduces the relative error to the range 0 to -0.0000047. The optimal constant for this is 0x5F37599E.

On the other hand, deleting the Newton step results in a substantially faster function with a relative error within ±0.035, using a constant of 0x5F37642F. It consists of only two integer instructions, plus code to load the constant. (The variable `xhalf` can be deleted.)

```
float rsqrt(float x0) {
   union {int ix; float x;};

   x = x0;                         // x can be viewed as int.
   float xhalf = 0.5f*x;
   ix = 0x5f375a82 - (ix >> 1);    // Initial guess.
   x = x*(1.5f - xhalf*x*x);       // Newton step.
   return x;
}
```

FIGURE 17–1. Approximate reciprocal square root.

To get an inkling of why this works, suppose $x = 2^n(1 + f)$, where n is the unbiased exponent and f is the fraction ($0 \le f < 1$). Then

$$\frac{1}{\sqrt{x}} = 2^{-n/2}(1+f)^{-1/2}.$$

Ignoring the fraction, this shows that we must change the biased exponent from $127 + n$ to $127 - n/2$. If $e = 127 + n$, then $127 - n/2 = 127 - (e - 127)/2 = 190.5 - e/2$. Therefore, it appears that a calculation something like shifting x right one position and subtracting it from 190 in the exponent position, might give a very rough approximation to $(1/\sqrt{x})$. In C, this can be expressed as[1]

```
union {int ix; float x;}; // Make ix and x overlap.
...
0x5F000000 - (ix >> 1);   // Refer to x as integer ix.
```

To find a better value for the constant 0x5F000000 by analysis is difficult. Four cases must be analyzed: the cases in which a 0-bit or a 1-bit is shifted from the exponent field to the fraction field, and the cases in which the subtraction does or does not generate a borrow that propagates to the exponent field. This analysis is done in [Lomo]. Here, we make some simple observations.

Using rep(x) to denote the representation of the floating-point number x in IEEE single format, we want a formula of the form

$$\text{rep}(1/\sqrt{x}) \approx k - (\text{rep}(x) \overset{s}{\gg} 1)$$

for some constant k. (Whether the shift is signed or unsigned makes no difference, because we exclude negative values of x and -0.0.) We can get an idea of roughly what k should be from

$$k \approx \text{rep}(1/\sqrt{x}) + (\text{rep}(x) \overset{s}{\gg} 1),$$

and trying a few values of x. The results are shown in Table 17–4 (in hexadecimal).

It looks like k is approximately a constant. Notice that the same value is obtained for $x = 1.0$ and 4.0. In fact, the same value of k results from any number x and $4x$ (provided they are both normal numbers). This is because, in the formula for k, if x is quadrupled, then the term rep$(1/\sqrt{x})$ decreases by 1 in the exponent field, and the term $\text{rep}(x) \overset{s}{\gg} 1$ increases by 1 in the exponent field.

More significantly, the relative errors for *x* and 4*x* are exactly the same, provided both quantities are normal numbers. To see this, it can be shown that if the argument `x` of the `rsqrt` function is quadrupled, the result of the function is exactly halved, and this is true no matter how many Newton steps are done. Of course, $1/\sqrt{x}$ is also halved. Therefore, the relative error is unchanged.

TABLE 17–4. DETERMINING THE CONSTANT

Trial x	rep(x)	rep($1/\sqrt{x}$)	k
1.0	3F800000	3F800000	5F400000
1.5	3FC00000	3F5105EC	5F3105EC
2.0	40000000	3F3504F3	5F3504F3
2.5	40200000	3F21E89B	5F31E89B
3.0	40400000	3F13CD3A	5F33CD3A
3.5	40600000	3F08D677	5F38D677
4.0	40800000	3F000000	5F400000

This is important, because it means that if we find an optimal value (by some criterion, such as minimizing the maximum absolute value of the error) for values of *x* in the range 1.0 to 4.0, then the same value of *k* is optimal for all normal numbers.

It is then a straightforward task to write a program that, for a given value of *k*, calculates the true value of $1/\sqrt{x}$ (using a known accurate library routine) and the estimated value for some 10,000 or so values of *x* from 1.0 to 4.0, and calculates the maximum error. The optimal value of *k* can be determined by hand, which is tedious but sometimes illuminating. It is quite amazing that there is a constant for which the error is less than ±3.5% in a function that uses only two integer operations and no table lookup.

17–5 The Distribution of Leading Digits

When IBM introduced the System/360 computer in 1964, numerical analysts were horrified at the loss of precision of single-precision arithmetic. The previous IBM computer line, the 704 - 709 - 7090 family, had a 36-bit word. For single-precision floating-point, the format consisted of a 9-bit sign and exponent field, followed by a 27-bit fraction in binary. The most significant fraction bit was explicitly included (in “normal” numbers), so quantities were represented with a precision of 27 bits.

The S/360 has a 32-bit word. For single-precision, IBM chose to have an 8-bit sign and exponent field followed by a 24-bit fraction. This drop from 27 to 24 bits was bad enough, but it gets worse. To keep the exponent range large, a unit in the 7-bit exponent of the S/360 format represents a factor of 16. Thus, the fraction is in base 16, and this format came to be called “hexadecimal” floating-point. The leading digit can be any number from 1 to 15 (binary 0001 to 1111). Numbers with leading digit 1 have only 21 bits of precision (because of the three leading 0’s), but they should constitute only 1/15 (6.7%) of all numbers.

No, it's worse than that! There was a flurry of activity to show, both analytically and empirically, that leading digits are *not* uniformly distributed. In hexadecimal floating-point, one would expect 25% of the numbers to have leading digit 1, and hence only 21 bits of precision.

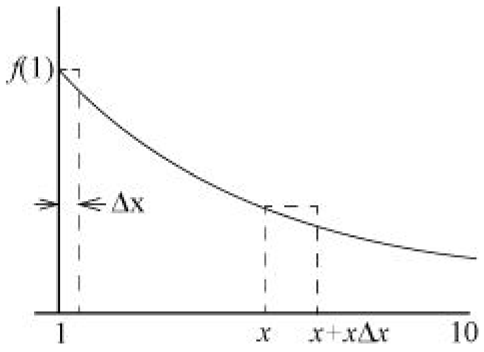
Let us consider the distribution of leading digits in decimal. Suppose you have a large set of numbers with units, such as length, volume, mass, speed, and so on, expressed in "scientific" notation (e.g., 6.022×10^{23}). If the leading digit of a large number of such numbers has a well-defined distribution function, then it must be independent of the units—whether inches or centimeters, pounds or kilograms, and so on. Thus, if you multiply all the numbers in the set by any constant, the distribution of leading digits should be unchanged. For example, considering multiplying by 2, we conclude that the number of numbers with leading digit 1 (those from 1.0 to 1.999... times 10 to some power) must equal the number of numbers with leading digit 2 or 3 (those from 2.0 to 3.999... times 10 to some power), because it shouldn't matter if our unit of length is inches or half inches, or our unit of mass is kilograms or half kilograms, and so on.

Let *f(x)*, for $1 \le x < 10$, be the probability density function for the leading digits of the set of numbers with units. *f(x)* has the property that

$$\int_a^b f(x)dx$$

is the proportion of numbers that have leading digits ranging from *a* to *b*. Referring to the figure below, for a small increment Δx in *x, f* must satisfy

$$f(1) \cdot \Delta x = f(x) \cdot x\Delta x,$$



because $f(1) \cdot \Delta x$ is, approximately, the proportion of numbers ranging from 1 to $1 + \Delta x$ (ignoring a multiplier of a power of 10), and $f(x) \cdot x\Delta x$ is the approximate proportion of numbers ranging from x to $x + x\Delta x$. Because the latter set is the first set multiplied by *x*, their proportions must be equal. Thus, the probability density function is a simple reciprocal relationship,

$$f(x) = f(1) / x.$$

Because the area under the curve from $x = 1$ to $x = 10$ must be 1 (all numbers have leading digits from 1.000... to 9.999...), it is easily shown that

$$f(1) = 1/\ln 10.$$

The proportion of numbers with leading digits in the range *a* to *b*, with $1 \le a \le b < 10$, is

$$\int_a^b \frac{dx}{x\ln 10} = \left.\frac{\ln x}{\ln 10}\right|_a^b = \frac{\ln b/a}{\ln 10} = \log_{10}\frac{b}{a}.$$

Thus, in decimal, the proportion of numbers with leading digit 1 is $\log_{10}(2 / 1) \approx 0.30103$, and the proportion of numbers with leading digit 9 is $\log_{10}(10 / 9) \approx 0.0458$.

For base 16, the proportion of numbers with leading digits in the range *a* to *b*, with $1 \le a \le b < 16$, is similarly derived to be $\log_{16}(b / a)$. Hence, the proportion of numbers with leading digit 1 is $\log_{16}(2 / 1) = 1 / \log_2 16 = 0.25$.

17–6 Table of Miscellaneous Values

Table 17–5 shows the IEEE representation of miscellaneous values that may be of interest. The values that are not exact are rounded to the nearest representable value.

TABLE 17–5. MISCELLANEOUS VALUES

Decimal	Single Format (Hex)	Double Format (Hex)
$-\infty$	FF80 0000	FFF0 0000 0000 0000
−2.0	C000 0000	C000 0000 0000 0000
−1.0	BF80 0000	BFF0 0000 0000 0000
−0.5	BF00 0000	BFE0 0000 0000 0000
−0.0	8000 0000	8000 0000 0000 0000
+0.0	0000 0000	0000 0000 0000 0000
Smallest positive subnormal	0000 0001	0000 0000 0000 0001
Largest subnormal	007F FFFF	000F FFFF FFFF FFFF
Least positive normal	0080 0000	0010 0000 0000 0000
$\pi/180$ (0.01745…)	3C8E FA35	3F91 DF46 A252 9D39
0.1	3DCC CCCD	3FB9 9999 9999 999A
$\log_{10} 2$ (0.3010…)	3E9A 209B	3FD3 4413 509F 79FF
$1/e$ (0.3678…)	3EBC 5AB2	3FD7 8B56 362C EF38
1/ln 10 (0.4342…)	3EDE 5BD9	3FDB CB7B 1526 E50E

0.5	3F00 0000	3FE0 0000 0000 0000
ln 2 (0.6931…)	3F31 7218	3FE6 2E42 FEFA 39EF
$1/\sqrt{2}$ (0.7071…)	3F35 04F3	3FE6 A09E 667F 3BCD
1/ln 3 (0.9102…)	3F69 0570	3FED 20AE 03BC C153
1.0	3F80 0000	3FF0 0000 0000 0000
ln 3 (1.0986…)	3F8C 9F54	3FF1 93EA 7AAD 030B
$\sqrt{2}$ (1.414…)	3FB5 04F3	3FF6 A09E 667F 3BCD
1/ln 2 (1.442…)	3FB8 AA3B	3FF7 1547 652B 82FE
$\sqrt{3}$ (1.732…)	3FDD B3D7	3FFB B67A E858 4CAA
2.0	4000 0000	4000 0000 0000 0000
ln 10 (2.302…)	4013 5D8E	4002 6BB1 BBB5 5516
e (2.718…)	402D F854	4005 BF0A 8B14 5769
3.0	4040 0000	4008 0000 0000 0000
π (3.141…)	4049 0FDB	4009 21FB 5444 2D18
$\sqrt{10}$ (3.162…)	404A 62C2	4009 4C58 3ADA 5B53
$\log_2 10$ (3.321…)	4054 9A78	400A 934F 0979 A371
4.0	4080 0000	4010 0000 0000 0000
5.0	40A0 0000	4014 0000 0000 0000
6.0	40C0 0000	4018 0000 0000 0000
2π (6.283…)	40C9 0FDB	4019 21FB 5444 2D18
7.0	40E0 0000	401C 0000 0000 0000
8.0	4100 0000	4020 0000 0000 0000
9.0	4110 0000	4022 0000 0000 0000
10.0	4120 0000	4024 0000 0000 0000
11.0	4130 0000	4026 0000 0000 0000
12.0	4140 0000	4028 0000 0000 0000
13.0	4150 0000	402A 0000 0000 0000
14.0	4160 0000	402C 0000 0000 0000
15.0	4170 0000	402E 0000 0000 0000
16.0	4180 0000	4030 0000 0000 0000
180/π (57.295…)	4265 2EE1	404C A5DC 1A63 C1F8
$2^{23} - 1$	4AFF FFFE	415F FFFF C000 0000

2^{23}	4B00 0000	4160 0000 0000 0000
$2^{24} - 1$	4B7F FFFF	416F FFFF E000 0000
2^{24}	4B80 0000	4170 0000 0000 0000
$2^{31} - 1$	4F00 0000	41DF FFFF FFC0 0000
2^{31}	4F00 0000	41E0 0000 0000 0000
$2^{32} - 1$	4F80 0000	41EF FFFF FFE0 0000
2^{32}	4F80 0000	41F0 0000 0000 0000
2^{52}	5980 0000	4330 0000 0000 0000
2^{63}	5F00 0000	43E0 0000 0000 0000
2^{64}	5F80 0000	43F0 0000 0000 0000
Largest normal	7F7F FFFF	7FEF FFFF FFFF FFFF
∞	7F80 0000	7FF0 0000 0000 0000
"Smallest" SNaN	7F80 0001	7FF0 0000 0000 0001
"Largest" SNaN	7FBF FFFF	7FF7 FFFF FFFF FFFF
"Smallest" QNaN	7FC0 0000	7FF8 0000 0000 0000
"Largest" QNaN	7FFF FFFF	7FFF FFFF FFFF FFFF

IEEE 754 does not specify how the signaling and quiet NaNs are distinguished. Table 17–5 uses the convention employed by PowerPC, the AMD 29050, the Intel x86 and I860, the SPARC, and the ARM family: The most significant fraction bit is 0 for signaling and 1 for quiet NaN's. A few machines, mostly older ones, use the opposite convention (0 = quiet, 1 = signaling).

Exercises

1. What numbers have the same representation, apart from trailing 0's, in both single- and double-precision?

2. Is there a program similar to the approximate reciprocal square root routine for computing the approximate square root?

3. Is there a similar program for the approximate cube root of a nonnegative normal number?

4. Is there a similar program for the reciprocal square root of a double-precision floating-point number? Assume it is for a 64-bit machine, or at any rate that the "long long" (64-bit integer) data type is available.

Chapter 18. Formulas For Primes

18–1 Introduction

Like many young students, I once became fascinated with prime numbers and tried to find a formula for them. I didn't know exactly what operations would be considered valid in a "formula," or exactly what function I was looking for—a formula for the *n*th prime in terms of *n*, or in terms of the previous prime(s), or a formula that produces primes but not all of them, or something else. Nevertheless, in spite of these ambiguities, I would like to discuss a little of what is known about this problem. We will see that (a) there *are* formulas for primes, and (b) none of them are very satisfying.

Much of this subject relates to the present work in that it deals with formulas similar to those of some of our programming tricks, albeit in the domain of real number arithmetic rather than "computer arithmetic." Let us first review a few highlights from the history of this subject.

In 1640, Fermat conjectured that the formula

$$F_n = 2^{2n} + 1$$

always produces a prime, and numbers of this form have come to be called "Fermat numbers." It is true that F_n is prime for n ranging from 0 to 4, but Euler found in 1732 that

$$F_5 = 2^{25} + 1 = 641 \cdot 6700417.$$

(We have seen these factors before in connection with dividing by a constant on a 32-bit machine). Then, F. Landry showed in 1880 that

$$F_6 = 2^{26} + 1 = 274177 \cdot 67280421310721.$$

F_n is now known to be composite for many larger values of n, such as all n from 7 to 16 inclusive. For no value of $n > 4$ is it known to be prime [H&W]. So much for rash conjectures.[1]

Incidentally, why would Fermat be led to the double exponential? He knew that if m has an odd factor other than 1, then $2^m + 1$ is composite. For if $m = ab$ with b odd and not equal to 1, then

$$2^{ab} + 1 = (2^a + 1)(2^{a(b-1)} - 2^{a(b-2)} + 2^{a(b-3)} - \ldots + 1.$$

Knowing this, he must have wondered about $2^m + 1$ with m not containing any odd factors (other than 1)—that is, $m = 2^n$. He tried a few values of n and found that $2^{2n} + 1$ seemed to be prime.

Certainly everyone would agree that a polynomial qualifies as a "formula." One rather amazing polynomial was discovered by Leonhard Euler in 1772. He found that

$$f(n) = n^2 + n + 41$$

is prime-valued for every n from 0 to 39. His result can be extended. Because

$$f(-n) = n^2\ n + 41 = f(n-1),$$

$f(-n)$ is prime-valued for every n from 1 to 40; that is, $f(n)$ is prime-valued for every n from –1 to –40. Therefore,

$$f(n-40) = (n-40)^2 + (n-40) + 41 = n^2 - 79n + 1601$$

is prime-valued for every n from 0 to 79. (However, it is lacking in aesthetic appeal because it is nonmonotonic and it repeats; that is, for $n = 0, 1, \ldots, 79$, $n^2 - 79n + 1601 = 1601, 1523, 1447, \ldots, 43, 41, 41, 43, \ldots, 1447, 1523, 1601$.)

In spite of this success, it is now known that there is no polynomial $f(n)$ that produces a prime for every n (aside from constant polynomials such as $f(n) = 5$). In fact, any nontrivial “polynomial in exponentials” is composite infinitely often. More precisely, as stated in [H & W],

THEOREM. *If $f(n) = p(n, 2^n, 3^n, \ldots, kn)$ is a polynomial in its arguments, with integral coefficients, and $f(n) \to \infty$ when $n \to \infty$, then $f(n)$ is composite for an infinity of values of n.*

Thus, a formula such as $n^2 \cdot 2^n + 2n^3 + 2n + 5$ must produce an infinite number of composites. On the other hand, the theorem says nothing about formulas containing terms such as 2^{2^n}, n^n, and $n!$.

A formula for the nth prime, in terms of n, can be obtained by using the floor function and a magic number

$$a = 0.203005000700011000013\ldots.$$

The number a is, in decimal, the first prime written in the first place after the decimal point, the second prime written in the next two places, the third prime written in the next three places, and so on. There is always room for the nth prime, because $p_n < 10^n$. We will not prove this, except to point out that it is known that there is always a prime between n and $2n$ (for $n \geq 2$), and hence certainly at least one between n and $10n$, from which it follows that $p_n < 10^n$. The formula for the nth prime is

$$p_n = \left\lfloor 10^{\frac{n^2+n}{2}} a \right\rfloor - 10^n \left\lfloor 10^{\frac{n^2-n}{2}} a \right\rfloor,$$

where we have used the relation $1 + 2 + 3 + \ldots + n = (n^2 + n) / 2$. For example,

$$\begin{aligned} p_3 &= \lfloor 10^6 a \rfloor - 10^3 \lfloor 10^3 a \rfloor \\ &= 203005 - 203000 \\ &= 5. \end{aligned}$$

This is a pretty cheap trick, as it requires knowledge of the result to define a. The formula would be interesting if there were some way to define a independent of the primes, but no one knows of such a definition.

Obviously, this technique can be used to obtain a formula for many sequences, but it begs the question.

18–2 Willans's Formulas

C. P. Willans gives the following formula for the *n*th prime [Will]:

$$p_n = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{n} \left(\sum_{x=1}^{m} \left\lfloor \cos^2 \pi \frac{(x-1)! + 1}{x} \right\rfloor \right)^{-1/n} \right\rfloor.$$

The derivation starts from Wilson's theorem, which states that p is prime or 1 if and only if $(p-1)! \equiv -1 \pmod{p}$. Thus,

$$\frac{(x-1)!+1}{x}$$

is an integer for x prime or $x = 1$ and is fractional for all composite x. Hence,

$$F(x) = \left\lfloor \cos^2 \pi \frac{(x-1)!+1}{x} \right\rfloor = \begin{cases} 1, & x \text{ prime or } 1, \\ 0, & x \text{ composite.} \end{cases} \qquad (1)$$

Thus, if $\pi(m)$ denotes[2] the number of primes $\le m$,

$$\pi(m) = -1 + \sum_{x=1}^{m} F(x). \qquad (2)$$

Observe that $\pi(p_n) = n$, and furthermore,

$$\pi(m) < n, \text{ for } m < p_n, \text{ and}$$

$$\pi(m) \ge n, \text{ for } m \ge p_n.$$

Therefore, the number of values of m from 1 to ∞ for which $\pi(m) < n$ is $p_n - 1$. That is,

$$p_n = 1 + \sum_{m=1}^{\infty} (\pi(m) < n), \qquad (3)$$

where the summand is a "predicate expression" (0/1-valued).

Because we have a formula for $\pi(m)$, Equation (3) constitutes a formula for the wth prime as a function of n. But it has two features that might be considered unacceptable: an infinite summation and the use of a "predicate expression," which is not in standard mathematical usage.

It has been proved that for $n \ge 1$ there is at least one prime between n and $2n$. Therefore, the number of primes $\le 2^n$ is at least n— that is, $\pi(2^n) \ge n$. Thus, the predicate $\pi(m) < n$ is 0 for $m \ge 2^n$, so the upper limit of the summation above can be replaced with 2^n.

Willans has a rather clever substitute for the predicate expression. Let

$$\mathrm{LT}(x, y) = \left\lfloor \sqrt[y]{\frac{y}{1+x}} \right\rfloor, \text{ for } x = 0, 1, 2, \ldots;\ y = 1, 2, \ldots.$$

Then, if $x < y$, $1 \le y/(1+x) \le y$, so $1 \le \sqrt[y]{y/(1+x)} \le \sqrt[y]{y} < 2$. Furthermore, if $x \ge y$, then $0 < y/(1+x) < 1$, so $0 \le \sqrt[y]{y/(1+x)} < 1$. Applying the floor function, we have

$$\mathrm{LT}(x, y) = \begin{cases} 1, & \text{for } x < y, \\ 0, & \text{for } x \ge y, \end{cases}$$

That is, LT(x, y) is the predicate $x < y$ (for x and y in the given ranges).

Substituting, Equation (3) can be written

$$\begin{aligned} p_n &= 1 + \sum_{m=1}^{2^n} \mathrm{LT}(\pi(m), n) \\ &= 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{\frac{n}{1+\pi(m)}} \right\rfloor. \end{aligned}$$

Further substituting Equation (2) for $\pi(m)$ in terms of $F(x)$, and Equation (1) for $F(x)$, gives the formula shown at the beginning of this section.

Second Formula

Willans then gives another formula:

$$p_n = \sum_{m=1}^{2^n} mF(m) \lfloor 2^{-|\pi(m)-n|} \rfloor.$$

Here, F and π are the functions used in his first formula. Thus, $mF(m) = m$ if m is prime or 1, and 0 otherwise. The third factor in the summand is the predicate $\pi(m) = n$. The summand is 0 except for one term, which is the nth prime. For example,

$$\begin{aligned} p_4 &= 1{\cdot}1{\cdot}0 + 2{\cdot}1{\cdot}0 + 3{\cdot}1{\cdot}0 + 4{\cdot}0{\cdot}0 + 5{\cdot}1{\cdot}0 + 6{\cdot}0{\cdot}0 + 7{\cdot}1{\cdot}1 \\ &\quad + 8{\cdot}0{\cdot}1 + 9{\cdot}0{\cdot}1 + 10{\cdot}0{\cdot}1 + 11{\cdot}1{\cdot}0 + \ldots + 16{\cdot}0{\cdot}0 \\ &= 7. \end{aligned}$$

Third Formula

Willans goes on to present another formula for the nth prime that does not use any "nonanalytic"[3] functions such as floor and absolute value. He starts by noting that for $x = 2, 3, \ldots$, the function

$$\frac{((x-1)!)^2}{x} = \begin{cases} \text{an integer} + \dfrac{1}{x} & \text{when } x \text{ is prime,} \\ \text{an integer,} & \text{when } x \text{ is composite or 1.} \end{cases}$$

The first part follows from

$$\frac{((x-1)!)^2}{x} = \frac{((x-1)!+1)\cdot((x-1)!-1)}{x} + \frac{1}{x}$$

and x divides $(x-1)! + 1$, by Wilson's theorem. Thus, the predicate "x is prime," for $x \geq 2$, is given by

$$H(x) = \frac{\sin^2\pi\frac{((x-1)!)^2}{x}}{\sin^2\frac{\pi}{x}}.$$

From this it follows that

$$\pi(m) = \sum_{x=2}^{m} H(x), \text{ for } m = 2, 3, \ldots.$$

This cannot be converted to a formula for p_n by the methods used in the first two formulas, because they use the floor function. Instead, Willans suggests the following formula[4] for the predicate $x < y$, for $x, y \geq 1$:

$$\mathrm{LT}(x, y) = \sin\left(\frac{\pi}{2}\cdot 2^e\right), \text{ where}$$

$$e = \prod_{i=0}^{y-1}(x-i).$$

Thus, if $x < y$, $e = x(x-1)\ldots(0)(-1)\ldots(x-(y-1)) = 0$, so that $LT(x,y) = \sin(\pi/2) = 1$. If $x \geq y$, the product does not include 0, so $e \geq 1$, so that $\mathrm{LT}(x,y) = \sin((\pi/2) \cdot (\text{an even number})) = 0$.

Finally, as in the first of Willans's formulas,

$$p_n = 2 + \sum_{m=2}^{2^n} \mathrm{LT}(\pi(m), n).$$

Written out in full, this is the rather formidable

$$p_n = 2 + \sum_{m=2}^{2^n} \sin\left(\frac{\pi}{2}\cdot 2^{\prod_{i=0}^{n-1}\left(\sum_{x=2}^{m}\frac{\sin^2\pi\frac{((x-1)!)^2}{x}}{\sin^2\frac{\pi}{x}} - i\right)}\right).$$

Fourth Formula

Willans then gives a formula for $p^n + 1$ in terms of p_n:

$$p_{n+1} = 1 + p_n + \sum_{i=1}^{2p_n} \prod_{j=1}^{i} f(p_n + j),$$

where *f(x)* is the predicate "*x* is composite," for $x \geq 2$; that is,

$$f(x) = \left\lfloor \cos^2 \pi \frac{((x-1)!)^2}{x} \right\rfloor.$$

Alternatively, one could use $f(x) = 1 - H(x)$, to keep the formula free of floor functions.

As an example of this formula, let $p_n = 7$. Then,

$$\begin{aligned} p_{n+1} &= 1 + 7 + f(8) + f(8)f(9) + f(8)f(9)f(10) \\ &\quad + f(8)f(9)f(10)f(11) + \ldots + f(8)f(9)\ldots f(14) \\ &= 1 + 7 + 1 + 1\cdot1 + 1\cdot1\cdot1 + 1\cdot1\cdot1\cdot0 + \ldots + 1\cdot1\cdot1\cdot0\cdot1\cdot0\cdot1 \\ &= 11. \end{aligned}$$

18–3 Wormell's Formula

C. P. Wormell [Wor] improves on Willans's formulas by avoiding both trigonometric functions and the floor function. Wormell's formula can, in principle, be evaluated by a simple computer program that uses only integer arithmetic. The derivation does not use Wilson's theorem. Wormell starts with, for $x \geq 2$,

$$B(x) = \prod_{a=2}^{x} \prod_{b=2}^{x} (x - ab)^2 = \begin{cases} \text{a positive integer, if } x \text{ is prime,} \\ 0, \text{ if } x \text{ is composite.} \end{cases}$$

Thus, the number of primes $\leq m$ is given by

$$\pi(m) = \sum_{x=2}^{m} \frac{1 + (-1)^{2^{B(x)}}}{2}$$

because the summand is the predicate "*x* is prime."

Observe that, for $n \geq 1$, $a \geq 0$,

$$\prod_{r=1}^{n} (1 - r + a)^2 = \begin{cases} 0, \text{ when } a < n, \\ \text{a positive integer, when } a \geq n. \end{cases}$$

Repeating a trick above, the predicate $a < n$ is

$$(a < n) = \frac{1 - (-1)^{2^{\prod_{r=1}^{n} (1 - r + a)^2}}}{2}.$$

Because

$$p_n = 2 + \sum_{m=2}^{2^n} (\pi(m) < n),$$

we have, upon factoring constants out of summations,

$$p_n = \frac{3}{2} + 2^{n-1} - \frac{1}{2}\sum_{m=2}^{2^n} (-1)^{2^{\prod_{r=1}^{n}\left(1 - r + \frac{(m-1)}{2} + \frac{1}{2}\sum_{x=2}^{m}(-1)^{2^{\prod_{a=2}^{x}\prod_{b=2}^{x}(x-ab)^2}}\right)^2}}.$$

As promised, Wormell's formula does not use trigonometric functions. However, as he points out, if the powers of -1 were expanded using $(-1)^n = \cos \pi n$, they would reappear.

18–4 Formulas for Other Difficult Functions

Let us have a closer look at what Willans and Wormell have done. We postulate the rules below as defining what we mean by the class of functions that can be represented by "formulas," which we will call "formula functions." Here, $\bar{x}$ is shorthand for $x_1, x_2, \ldots, x_n$ for any $n \geq 1$. The domain of values is the integers ... -2, -1, 0, 1, 2, ...

1. The constants ... -1, 0, 1, ... are formula functions.
2. The projection functions $f(\bar{x}) = x_i$, for $1 \leq i \leq n$, are formula functions.
3. The expressions $x+y$, $x-y$, and xy are formula functions, if x and y are.
4. The class of formula functions is closed under composition (substitution). That is, $f(g_1(\bar{x}), g_2(\bar{x}), \ldots, g_m(\bar{x}))$ is a formula function if f and g_i are, for $i = 1, \ldots, m$.
5. Bounded sums and products, written

$$\sum_{i=a(\bar{x})}^{b(\bar{x})}{}' f(i, \bar{x}) \qquad \prod_{i=a(\bar{x})}^{b(\bar{x})}{}' f(i, \bar{x}),$$

are formula functions, if a, b, and f are, and $a(\bar{x}) \leq b(\bar{x})$.

Sums and products are required to be bounded to preserve the computational character of formulas; that is, formulas can be evaluated by plugging in values for the arguments and carrying out a finite number of calculations. The reason for the prime on the Σ and Π is explained later in this chapter.

When forming new formula functions using composition, we supply parentheses when necessary according to well-established conventions.

Notice that division is not included in the list above; that's too complicated to be uncritically accepted as a "formula function." Even so, the above list is not minimal. It might be fun to find a minimal starting point, but we won't dwell on that here.

This definition of "formula function" is close to the definition of "elementary function" given in [Cut]. However, the domain of values used in [Cut] is the nonnegative integers (as is usual in recursive function theory). Also, [Cut] requires the bounds on the iterated sum and product to be 0 and $x - 1$ (where x is a variable), and allows the range to be vacuous (in which case the sum is defined as 0 and the product is defined as 1).

In what follows, we show that the class of formula functions is quite extensive, including most of the functions ordinarily encountered in mathematics. But it doesn't include every function that is easy to define and has an elementary character.

Our development is slightly encumbered, compared to similar developments in recursive function theory, because here variables can take on negative values. The possibility of a value's being negative can often be accommodated by simply squaring some expression that would otherwise appear in the first power. Our insistence that iterated sums and products not be vacuous is another slight encumbrance.

Here, a "predicate" is simply a 0/1-valued function, whereas in recursive function theory a predicate is a true/false-valued function, and every predicate has an associated "characteristic function" that is 0/1-valued. We associate 1 with true and 0 with false, as is universally done in programming languages and in computers (in what their *and* and *or* instructions do); in logic and recursive function theory, the association is often the opposite.

The following are formula functions:

1. $a^2 = aa$, $a^3 = aaa$, and so on.
2. The predicate $a = b$:

$$(a = b) = \prod_{j=0}^{(a-b)^2}{}'(1-j).$$

3. $(a \neq b) = 1 - (a = b)$.
4. The predicate $a \geq b$:

$$\begin{aligned}(a \geq b) &= \sum_{i=0}^{(a-b)^2}{}'((a-b) = i)\\ &= \sum_{i=0}^{(a-b)^2}{}' \prod_{j=0}^{((a-b)-i)^2}{}'(1-j).\end{aligned}$$

We can now explain why we do not use the convention that a vacuous iterated sum/product has the value 0/1. If we did, we would have such shams as

$$(a = b) = \sum_{i=0}^{-(a-b)^2} 1 \qquad \text{and} \qquad (a \geq b) = \prod_{i=a}^{b-1} 0.$$

The comparison predicates are key to everything that follows, and we don't wish to have them based on anything quite that artificial.

5. $(a > b) = (a \geq b + 1)$.
6. $(a \leq b) = (b \geq a)$.
7. $(a < b) = (b > a)$.
8. $|a| = (2(a \geq 0) - 1)a$.
9. $max(a,b) = (a \geq b)(a - b) + b$.

10. $\min(a, b) = (a \geq b)(b-a) + a$.

Now we can fix the iterated sums and products so that they give the conventional and useful result when the range is vacuous.

11. $$\sum_{i=a(\bar{x})}^{b(\bar{x})} f(i, \bar{x}) = (b(\bar{x}) \geq a(\bar{x})) \sum_{i=a(\bar{x})}^{\max(a(\bar{x}),\, b(\bar{x}))}{}' f(i, \bar{x}).$$.

12. $$\prod_{i=a(\bar{x})}^{b(\bar{x})} f(i, \bar{x}) = 1 + (b(\bar{x}) \geq a(\bar{x}))(-1 + \prod_{i=a(\bar{x})}^{\max(a(\bar{x}),\, b(\bar{x}))}{}' f(i, \bar{x})).$$.

From now on we will use Σ and Π without the prime. All functions thus defined are total (defined for all values of the arguments).

13. $$n! = \prod_{i=1}^{n} i.$$.

This gives $n! = 1$ for $n \leq 0$.

In what follows, P and Q denote predicates.

14. $\neg P(\bar{x}) = 1 - P(\bar{x})$..

15. $P(\bar{x})\ \&\ Q(\bar{x}) = P(\bar{x})Q(\bar{x})$..

16. $P(\bar{x}) \mid Q(\bar{x}) = 1 - (1 - P(\bar{x}))(1 - Q(\bar{x}))$..

17. $P(\bar{x}) \oplus Q(\bar{x}) = (P(\bar{x}) - Q(\bar{x}))^2$..

18. if $P(\bar{x})$ then $f(\bar{y})$ else $g(\bar{z}) = P(\bar{x})f(\bar{y}) + (1 - P(\bar{x}))g(\bar{z})$..

19. a^n = if $n \geq 0$ then $\prod_{i=1}^{n} a$ else 0.

This gives, arbitrarily and perhaps incorrectly for a few cases, the result 0 for $n < 0$, and the result 1 for 0^0.

20. $$(m \leq \forall x \leq n)P(x, \bar{y}) = \prod_{x=m}^{n} P(x, \bar{y}).$$.

21. $$(m \leq \exists x \leq n)P(x, \bar{y}) = 1 - \prod_{x=m}^{n} (1 - P(x, \bar{y})).$$.

is vacuously true; is vacuously false.

22. $$(m \leq \min x \leq n)P(x, \bar{y}) = m + \sum_{i=m}^{n} \prod_{j=m}^{i} (1 - P(j, \bar{y})).$$.

The value of this expression is the least x in the range m *to* n such that the predicate is true, or m if the range is vacuous, or $n + 1$ if the predicate is false throughout the (nonvacuous) range. The operation is called “bounded minimalization” and it is a very powerful tool for developing new formula functions. It is a sort of functional inverse, as illustrated by the next formula. That minimalization can be done by a sum of products is due to Goodstein [Good].

23. $\lfloor \sqrt{n} \rfloor = (0 \leq \min k \leq |n|)((k+1)^2 > n)$..

This is the “integer square root” function, which we define to be 0 for $n < 0$, just to make it a total function.

24. $d|n = (-|n| \le \exists q \le |n|)(n = qd)$.

This is the "d divides n" predicate, according to which 0|0 but ¬(0|n) for $n \neq 0$.

25. $n \div d$ = if $n \ge 0$ then $(-n \le \min q \le)(0 \le \exists r \le |d| - 1)(n = qd + r)$ else $(n \le \min q \le -n)(-|d| + 1 \le \exists r \le 0)(n = qd + r)$.

This is the conventional truncating form of integer division. For $d = 0$, it gives a result of $|n| + 1$, arbitrarily.

26. rem$(n, d) = n-(n \div d)d$.

This is the conventional remainder function. If rem (n, d) is nonzero, it has the sign of the numerator n. If $d = 0$, the remainder is n.

27. isprime$(n) = n \ge 2 \ \& \ \neg(2 \le \exists d \le |n| - 1)(d|n)$.

28. $\pi(n) = \sum_{i=1}^{n}$ isprime(i).

(Number of primes $\le n$.)

29. $p_n = (1 \le \min k \le 2^n)(\pi(k) = n)$.

30. exponent $(p, n) = (0 \le \min x \le |n|)\neg(p^{x+1}|n)$.

This is the exponent of a given prime factor p of n, for $n \ge 1$.

31. For $n \ge 0$:

$$2^n = \prod_{i=1}^{n} 2, \qquad 2^{2^n} = \prod_{i=1}^{2^n} 2, \qquad 2^{2^{2^n}} = \prod_{i=1}^{2^{2^n}} 2, \text{ etc.}$$

32. The nth digit after the decimal point in the decimal expansion of $\sqrt{2}$: rem $(\lfloor \sqrt{2 \cdot 10^{2n}} \rfloor, 10)$.

Thus, the class of formula functions is quite large. It is limited, though, by the following theorem (at least):

THEOREM. *If f is a formula function, then there is a constant k such that*

$$f(\bar{x}) \le 2^{2^{\cdot^{\cdot^{\cdot^{2^{\max(|x_1|, \ldots, |x_n|)}}}}}}$$

where there are k 2's.

This can be proved by showing that each application of one of the rules 1–5 (on page 398) preserves the theorem. For example, if $f(\bar{x}) = c$ (rule 1), then for some h,

$$f(\bar{x}) \le 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} h,$$

where there are h 2's. Therefore,

$$f(\bar{x}) \le 2^{2^{\cdot^{\cdot^{\cdot^{2^{\max(|x_1|, \ldots, |x_n|)}}}}}} \Big\} h + 2,$$

because $\max(|x_1|, \ldots, |x_n|) \ge 0$.

For $f(\bar{x}) = x_i$ (rule 2), $f(\bar{x}) \le \max(|x_1|, \dots, |x_n|)$, so the theorem holds with $k = 0$.

For rule 3, let

$$f(\bar{x}) \le 2^{2^{\cdots 2^{\max(|x_1|, \dots, |x_n|)}}}\}k_1 \qquad \text{and} \qquad g(\bar{x}) \le 2^{2^{\cdots 2^{\max(|x_1|, \dots, |x_n|)}}}\}k_2.$$

Then, clearly

$$f(\bar{x}) \pm g(\bar{x}) \le 2 \cdot 2^{2^{\cdots 2^{\max(|x_1|, \dots, |x_n|)}}}\}\max(k_1, k_2)$$

$$\le 2^{2^{\cdots 2^{\max(|x_1|, \dots, |x_n|)}}}\}\max(k_1, k_2) + 1.$$

Similarly, it can be shown that the theorem holds for $f(x, y) = xy$.

The proofs that rules 4 and 5 preserve the theorem are a bit tedious, but not difficult, and are omitted.

From the theorem, it follows that the function

$$f(x) = 2^{2^{\cdots 2^x}}\}x \qquad (4)$$

is not a formula function, because for sufficiently large x, Equation (4) exceeds the value of the same expression with any fixed number k of 2's.

For those interested in recursive function theory, we point out that Equation (4) is primitive recursive. Furthermore, it is easy to show directly from the definition of primitive recursion that formula functions are primitive recursive. Therefore, the class of formula functions is a proper subset of the primitive recursive functions. The interested reader is referred to [Cut].

In summary, this section shows that not only is there a formula in elementary functions for the nth prime but also for a good many other functions encountered in mathematics. Furthermore, our "formula functions" are not based on trigonometric functions, the floor function, absolute value, powers of -1, or even division. The only sneaky maneuver is to use the fact that the product of a lot of numbers is 0 if any one of them is 0, which is used in the formula for the predicate $a = b$.

It is true, however, that once you see them, they are not interesting. The quest for "interesting" formulas for primes should go on. For example, [Rib] cites the amazing theorem of W. H. Mills (1947) that there exists a θ such that the expression

$$\lfloor \theta^{3^n} \rfloor$$

is prime-valued for all $n \ge 1$. Actually, there are an infinite number of such values (e.g., 1.3063778838+ and 1.4537508625483+). Furthermore, there is nothing special about the "3"; the theorem is true if the 3 is replaced with any real number ≥2.106 (for different values of θ). Better yet, the 3 can be replaced with 2 if it is true that there is always a prime between n^2 and $(n + 1)^2$, which is almost certainly true, but has never been proved. And furthermore, ... well, the interested reader is referred to [Rib] and to [Dud] for more fascinating formulas of this type.

Exercises

1. Prove that for any non-constant polynomial $f(x)$ with integral coefficients, $|f(x)|$ is composite for an infinite number of values of x.

Hint: If $f(x_0) = k$, consider $f(x_0 + rk)$, where r is an integer greater than 1.

2. Prove Wilson's theorem: An integer $p > 1$ is prime if and only if

$$(p-1)! \equiv -1 \pmod{p}.$$

Hint: To show that if p is prime, then $(p-1)! \equiv -1 \pmod{p}$, group the terms of the factorial in pairs (a, b) such that $ab \equiv 1 \pmod{p}$. Use Theorem MI of Section 10–16 on page 240.

3. Show that if n is a composite integer greater than 4, then

$$(n-1)!=0 \pmod{n}.$$

4. Calculate an estimate of the value of θ that satisfies Mills's theorem, and in the process give an informal proof of the theorem. Assume that for $n > 1$ there exists a prime between n^3 and $(n + 1)^3$. (This depends upon the Riemann Hypothesis, although it has been proved independent of RH for sufficiently large *n*.)

5. Consider the set of numbers of the form $a + b\sqrt{-5}$, where a and b are integers. Show that 2 and 3 are primes in this set; that is, they cannot be decomposed into factors in the set unless one of the factors is ±1 (a "unit"). Find a number in the set that has two distinct decompositions into products of primes. (The "fundamental theorem of arithmetic" states that prime decomposition is unique except for units and the order of the factors. Uniqueness does not hold for this set of numbers with multiplication and addition being that of complex numbers. It is an example of a "ring.").

Answers To Exercises

Chapter 1: Introduction

1. The following is pretty accurate:

```
$e_1$;
while ($e_2$) {
  statement
  $e_3$;}
```

If e_2 is not present in the `for` loop, the constant 1 is used for it in the above expansion (which would then be a nonterminating loop, unless something in *statement* terminates it).

Expressing a `for` loop in terms of a `do` loop is somewhat awkward, because the body of a `do` loop is always executed at least once, whereas the body of a `for` loop may not be executed at all, depending on e_1 and e_2. Nevertheless, the `for` loop can be expressed as follows.

```
$e_1$;
if ($e_2$) {
  do {statement; $e_3$;} while ($e_2$);
}
```

Again, if e_2 is not present in the `for` loop, then use 1 for it above.

2. If your code is

```
for (i = 0; i <= 0xFFFFFFFF; i++) {...}
```

then you have an infinite loop. A loop that works is

```
i = 0xFFFFFFFF;
do {i = i + 1;...} while (i < 0xFFFFFFFF);
```

3. The text mentions *multiply*, which for 32 × 32 ==< 64-bit multiplication needs two output registers.

It also mentions *divide*. The usual implementation of this instruction produces a remainder as well as the quotient, and execution time would be saved in many programs if both results were available.

Actually, the most natural machine division operation takes a doubleword dividend, a single word divisor, and produces a quotient and remainder. This uses three source registers and two targets.

Indexed store instructions use three source registers: the register being stored, the base register, and the index register.

To efficiently deal with bit fields in a register, many machines provide *extract* and *insert* instructions. The general form of *extract* needs three sources and one target. The source registers are the register that contains the field being extracted, a starting bit number, and an ending bit number or length. The result is right justified and either zero- or sign-extended and placed in the target register. Some machines provide this instruction only in the form in which the field length is an immediate quantity, which is a reasonable compromise because that is the common case.

The general *insert* instruction reads four source registers and writes one target register. As commonly implemented, the sources are a register that contains the source bits to be inserted in the target (these come from the low-order end of the source register), the starting bit position in the target, and the length. In addition to reading these three registers, the instruction must read the target register, combine it with the bits to be inserted, and write the result to the target register. As in the case of *extract*, the field length may be an immediate quantity, in which case the instruction does three register reads and one write.

Some machines provide a family of *select* instructions:

```
SELcc RT,RA,RB,RC
```

Register RC is tested, and if it satisfies the condition specified in the opcode (shown as `cc`, which may be `EQ`, `GT`, `GE`, etc.), then RA is selected; otherwise, RB is selected. The selected register is copied to the target.

Although not common, a plausible instruction is *bit select*, or *multiplex*:

```
MUX RT,RA,RB,RC
```

Here RC contains a mask. Wherever the mask is 1, the corresponding bit of RA is selected, and wherever it is 0, the corresponding bit of RB is selected. That is, it performs the operation

```
RT <-- RA & RC | RB & ~RC
```

Shift right/left double: A sometimes useful instruction is

```
SHLD RT,RA,RB,RC
```

This concatenates RA and RB, treating them as a double-length register, and shifts them left (or right) by an amount given by RC. RT gets the part of the result that has bits from RA and RB. These instructions are useful in “bignum” arithmetic and in more mundane situations.

In signal processing and other applications, it is helpful to have an instruction that computes `A*B + C`. This applies to both integer and floating-point data.

Of course, there are *load multiple* and *store multiple*, which require many register reads or writes. Although many RISCs have them, they are not usually considered to be RISC instructions.

Chapter 2: Basics

1. (Derivation by David de Kloet) Clearly the body of the *while*-loop is executed a number of times equal to the number of trailing 0’s in x. The k 1-bits partition the n-bit word into $k + 1$ segments, each containing 0 or more 0-bits. The number of 0's in each word *is n-k*. If *N* is the number of words $N = \binom{n}{k}$, but that need not concern us here), then the total number of 0's in all the words is *N(n – k)*. By symmetry, the number of 0's in any segment, summed over all N words, is the same, and is therefore equal to *N(n – k)/(k* + 1). Thus, the average number of 0's in any segment is *(n-k)/(k+1)*, and this applies to the last segment, which is the number of trailing 0's.

As an example, if $n = 32$ and $k = 3$, then the *while*-loop is executed 7.25 times, on average. On many machines the *while*-loop can be implemented in as few as three instructions *(and, shift right*, and *conditional branch)*, which might take as few as four cycles. With these parameters, the *while*-loop takes 4•7.25 = 29 cycles on average. This is less than the divide time on most 32-bit machines, resulting in de Kloet's algorithm being faster than Gosper's. For larger values of *k*, de Kloet's is still more favorable.

2. The *and* with **1** makes the shift amount independent of all bits of *x* except for its rightmost bit. Therefore, by looking at only the rightmost bit of the shift amount, one can ascertain whether the result is *x* or $x << 1$. Since both *x* and $x << 1$ are right-to-left computable, choosing one of these based on a rightmost bit is also. The function $x << (x \;\&\; \mathbf{2})$, incidentally, is not right-to-left computable. But $(x \;\&\; \mathbf{-2}) << (x \;\&\; \mathbf{2})$ is.

Another example is the function x^n, where we take $x^{\mathbf{0}}$ to be 1. This is not right-to-left computable because if *x* is even, then the rightmost bit of the result depends upon whether or not $x = \mathbf{0}$, and thus is a function of bits to the left of the rightmost position. But if it were known *a priori* that the variable *n* is either 0 or 1, then x^n is right-to-left computable. Similarly, $x^{n\&1}$ is right-to-left computable, for example, by

$$x^{n \,\&\, 1} = (x \,\&\, -(n \,\&\, 1)) + 1 - (n \,\&\, 1) = \begin{cases} 1, & n \text{ even,} \\ x, & n \text{ odd.} \end{cases}$$

Notice that x^n is like the left shift function in that x^n is right-to-left computable for any particular value of *n*, or if *n* is a variable restricted to the values 0 and 1, but not if *n* is an unrestricted variable.

3. A somewhat obvious formula for addition is given on page 16, item (g):

$$x + y = (x \quad y) + 2(x \,\&\, y).$$

Dividing each side by 2 gives Dietz's formula. The addition in Dietz's formula cannot overflow because the average of two representable integers is representable.

Notice that if we start with item (i) on page 16, we obtain the formula given in the text for the ceiling average of two unsigned integers.

$$\left\lceil \frac{x+y}{2} \right\rceil = (x \mid y) - ((x \oplus y) \overset{u}{\gg} 1).$$

4. Compute the floor average of *a* and *b*, and also of *c* and *d*, using Dietz's formula. Then compute the floor average of *x* and *y*, and apply a correction:

$$x = (a \,\&\, b) + ((a \oplus b) \overset{u}{\gg} 1),$$

$$y = (c \,\&\, d) + ((c \oplus d) \overset{u}{\gg} 1),$$

$$r = (x \,\&\, y) + ((x \oplus y) \overset{u}{\gg} 1),$$

$$r = r + ((a \oplus b) \,\&\, (c \oplus d) \,\&\, (x \oplus y) \,\&\, 1).$$

The correction step is really four operations, not the seven that it appears to be, because the *exclusive or* terms were calculated earlier. It was arrived at by the following reasoning: The computed value of x can be lower than the true (real number) average by 1/2, and this error occurs if a is odd and b is even, or vice versa. This error amounts to 1/4 after x and y are averaged. If this were the only truncation error, the first value computed for r would be correct, because in this case the true average is an integer plus 1/4, and we want the floor average, so we want to discard the 1/4 anyway. Similarly, the truncation in computing y can make the computed average lower than the true average by 1/4. The first computed value of r can be lower than the true average of x and y by 1/2. These errors accumulate. If they sum to an error less than 1, they can be ignored, because we want to discard the fractional part of the true average anyway. But if all three errors occur, they sum to $1/4 + 1/4 + 1/2 = 1$, which must be corrected by adding 1 to r. The last line does this: if one of a and b is odd, and one of c and d is odd, and one of x and y is odd, then we want to add 1, which the last line does.

5. The expression for $x \overset{u}{\le} y$ to be simplified is

$$(\neg x \mid y) \,\&\, ((x \quad y) \mid \neg(y - x)).$$

Only bit 31 of x and y is relevant in the logical operations of this expression. Because $y_{31} = 0$, the expression immediately simplifies to

$$\neg x \,\&\, (x \mid \neg(y - x)).$$

"Multiplying in" the $\neg x$ (distributive law) gives

$$\neg x \,\&\, \neg(y - x),$$

and applying De Morgan's law further simplifies it to three elementary instructions:

$$\neg(x \mid (y - x)).$$

(Removing the complementation operator gives a two-instruction solution for the predicate $x \overset{u}{>} y$.)

If **y** is a constant, we can use the identity $\neg u = -1 - u$ to rewrite the expression obtained from the distributive law as

$$\neg x \,\&\, (x - (y + 1)),$$

which is three instuctions because the addition of 1 to y can be done before evaluating the expression. This form is preferable when y is a small constant, because the *add immediate* instruction can be used. (Problem suggested by

George Timms.)

6. To get a carry from the second addition, the carry from the first addition must be 1, and the low-order 32 bits of the first sum must be all 1's. That is, the first sum must be at least $2^{33} - 1$. But the operands are each at most $2^{32} - 1$, so their sum is at most $2^{33} - 2$.

7. For notational simplicity, let us consider a 4-bit machine. Let *x* and *y* denote the integer values of 4-bit quantities under unsigned binary interpretation. Let *f* (*x, y*) denote the integer result of applying ordinary binary addition with end-around carry, to *x* and *y*, with a 4-bit adder and a 4-bit result. Then,

$$f(x, y) = \mathrm{mod}\left(x + y + \left\lfloor \frac{x+y}{16} \right\rfloor, 16\right).$$

x	ones(*x*)
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	–7
1001	–6
1010	–5
1011	–4
1100	–3
1101	–2
1110	–1
1111	–0

The table at the right shows the one's-complement interpretation of 4-bit binary words. Observe that the one's-complement interpretation of a word whose straight binary interpretation is *x* is given by

$$\mathrm{ones}(x) = \begin{cases} x, & 0 \le x \le 7, \\ x - 15, & 8 \le x \le 15. \end{cases}$$

We must show that *f* (*x, y*), when interpreted as a one's-complement integer, is the sum of *x* and *y* when they are interpreted as one's-complement integers. That is, we must show that

$$\mathrm{ones}(x) + \mathrm{ones}(y) = \mathrm{ones}(f(x, y)).$$

We are interested only in the non-overflow cases (that is, when the sum can be expressed as a one's-complement integer).

Case 0, $0 \le x, y \le 7$**.** Then, ones(*x*) + ones(*y*) = *x* + *y*, and

$$f(x, y) = \text{mod}(x + y + 0, 16) = x + y.$$

For no overflow, the one's-complement result must be in the range 0 to 7, and from the table it is apparent that we must have $x + y \le 7$. Therefore, ones$(x + y) = x + y$.

Case 1, $0 \le x \le 7$, $8 \le y \le 15$. Overflow cannot occur because ones$(x) \ge 0$ and ones$(y) \le 0$. In this case, ones(x) + ones$(y) = x + y - 15$. If $x + y < 16$,

$$f(x, y) = \text{mod}(x + y + 0, 16) = x + y.$$

In this case $x + y$ must be at least 8, so ones$(x + y) = x + y - 15$. On the other hand, if $x + y \ge 16$,

$$f(x, y) = \text{mod}(x + y + 1, 16) = x + y + 1 - 16 = x + y - 15.$$

Because $x + y$ is at most 22 and is at least 16, $1 \le x + y - 15 \le 7$, so that ones$(x + y - 15) = x + y - 15$.

Case 2, $8 \le x \le 15$, $0 \le y \le 7$. This is similar to case 1 above.

Case 3, $8 \le x \le 15$, $8 \le y \le 15$. Then, ones(x) + ones$(y) = x - 15 + y - 15 = x + y - 30$, and

$$f(x, y) = \text{mod}(x + y + 1, 16) = x + y + 1 - 16 = x + y - 15.$$

Because of the limits on x and y, $16 \le x + y \le 30$. To avoid overflow, the table reveals that we must have $x + y \ge 23$. For, in terms of one's-complement interpretation, we can add –6 and –1, or –6 and –0, but not –6 and –2, without getting overflow. Therefore, $23 \le x + y \le 30$. Hence $8 \le x + y - 15 \le 15$, so that ones$(x + y - 15) = x + y - 30$.

For the carry propagation question, for one's-complement addition, the worst case occurs for something like

```
   111...1111
 + 000...0100
   ----------
   000...0011
 +          1 (end-around carry)
   ----------
   000...0100
```

for which the carry is propagated n places, where n is the word size. In two's-complement addition, the worst case is $n - 1$ places, assuming the carry out of the high-order position is discarded.

The following comparisons are interesting, using 4-bit quantities for illustration: In straight binary (unsigned) or two's-complement arithmetic, the sum of two numbers is always (even if overflow occurs) correct modulo 16. In one's-complement, the sum is always correct modulo 15. If x_n denotes bit n of x, then in two's-complement notation, $x = -8x_3 + 4x_2 + 2x_1 + x_0$. In one's-complement notation, $x = -7x_3 + 4x_2 + 2x_1 + x_0$.

8. $((x \quad y) \& \ m) \quad y$.

9. $x \quad y = (x \mid y) \ \& \ \neg(x \ \& \ y)$.

10. [Arndt] Variable *t* is 1 if the bits differ (six instructions).

$$t \leftarrow ((x \overset{u}{\gg} i) \oplus (x \overset{u}{\gg} j)) \ \& \ 1$$
$$x \leftarrow x \oplus (t \ll j)$$

Adding the line $x \leftarrow x \quad (t << i)$ makes it swap bits *i* and *j*

11. As described in the text, any Boolean function $f(x_1, x_2, ..., x_n)$ can be decomposed into the form $g(x_1, x_2, ..., x_{n-1}) \ \ x_n \ h \ (x_1, x_2, ..., x_{n-1,})$. Let $c\ (n)$ be the number of instructions required for the decomposition of an *n*-variable Boolean function into binary Boolean instructions, for $n \geq 2$. Then

$$c_{n+1} = 2c_n + 2,$$

with $c_2 = 1$. This has the solution

$$c_n = 3 \cdot 2^{n-2} - 2.$$

(The least upper bound is much smaller.)

12. (a)

$$\begin{aligned}
f(x, y, z) &= \bar{z}f_0(x, y) + zf_1(x, y) \\
&= \bar{z}f_0(x, y) \oplus zf_1(x, y) \\
&= \bar{z}f_0(x, y) \oplus (1 \oplus \bar{z})f_1(x, y) \\
&= \bar{z}f_0(x, y) \oplus f_1(x, y) \oplus \bar{z}f_1(x, y) \\
&= f_1(x, y) \oplus \bar{z}(f_0(x, y) \oplus f_1(x, y)),
\end{aligned}$$

which is in the required form.

(b) From part (a),

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

which is in the required form.

13. Using the notation of Table 2–3 on page 54, the missing functions can be obtained from 0000 = *andc* (*x, x*), 0011 = *and* (*x, x*), 0100 = *andc* (*y, x*), 0101 = *and* (*y, y*), 1010 = *nand* (*y, y*), 1011 = *cor* (*y, x*), 1100 = *nand* (*x, x*), and 1111 = *cor* (*x, x*).

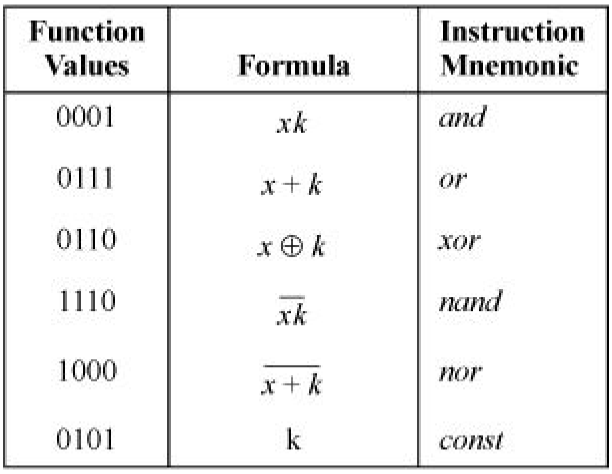
14. No. The ten truly binary functions are, in numeric form,

```
0001  0010  0100  0110  0111
1000  1001  1011  1101  1110
```

By implementing function 0010 you get 0100 by interchanging the operands, and, similarly, 1011 yields 1101. That's all you can accomplish by interchanging the operands, because the other functions are commutative. Equating the operands, of course, reduces a function to a constant or unary function. Therefore, you need eight instruction types.

15. The table below shows one set of six instruction types that accomplish the task. Here, *x* denotes the contents of the register operand, and *k* denotes the contents of the immediate field.

SIX SUFFICIENT R-I BOOLEAN INSTRUCTIONS



Function Values	Formula	Instruction Mnemonic
0001	xk	*and*
0111	$x + k$	*or*
0110	$x \oplus k$	*xor*
1110	$\overline{xk}$	*nand*
1000	$\overline{x + k}$	*nor*
0101	k	*const*

The missing functions can be obtained from 0000 = *and* (*x*, 0), 0010 = *and* (*x*, $\bar{k}$), 0011 = *or* (*x*, 0), 0100 = *nor* (*x*, $\bar{k}$), 1001 = *xor* (*x*, $\bar{k}$), 1010 = *const* (*x*, $\bar{k}$), 1011 = *or* (*x*, $\bar{k}$), 1100 = *nor* (*x*, 0), 1101 = *nand* (*x*, $\bar{k}$), and 1111 = *nand* (*x*, 0).

16. This writer does not know of an "analytic" way to do this. But it is not difficult to write a program that generates all Boolean functions of three variables that can be implemented with three binary instructions. Such a program is given in C below. It is written in as simple a way as possible to give a convincing answer to the question. Some optimizations are possible, which are mentioned below.

The program represents a function by an 8-bit string that is the truth table of the function, with the values for *x, y*, and *z* written in the usual way for a truth table. Each time a function is generated, it is checked off by setting a byte in vector `found` to 1. This vector is 256 bytes long and is initially all zero.

The truth table that the program works with is shown in the table below.

TRUTH TABLE FOR THREE VARIABLES

$f_0 = x$	$f_1 = y$	$f_2 = z$	f_3	f_4	f_5
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

The six columns of the truth table are stored in a vector `fun`. The first three positions of `fun` contain the truth table columns for *x, y*, and *z*. These columns have the values hexadecimal 0F, 33, and 55, which represent the trivial functions *f* (*x, y, z*) = *x*, *f* (*x, y, z*) = *y*, and *f* (*x, y, z*) = *z*. The next three positions will contain the truth table columns for the functions generated by one, two, and three binary instructions, respectively, for the current trial.

The program conceptually consists of three nested loops, one for each instruction currently being tried. The outermost loop iterates over all 16 binary Boolean operations, operating on all pairs of *x, y*, and *z* (16*3*3 = 144 iterations). For each iteration, the result of operating on all eight bits of *x, y*, and/or *z* in parallel is put in `fun[3]`.

The next level of looping similarly iterates over all 16 binary Boolean operations, operating on all pairs of *x, y, z*, and the result of the outermost loop (16*4*4 = 256 iterations). For each iteration, the result is put in `fun[4]`.

The innermost level of looping similarly iterates over all 16 binary Boolean operations, operating on all pairs of *x, y, z*, and the results of the outer two loops (16*5*5 = 400 iterations). For each of these calculated functions, the corresponding byte of `found` is set to 1.

At the end, the program writes out vector `found` in 16 rows of 16 vector elements each. Several positions of vector `found` are 0, showing that three binary Boolean instructions do not suffice to implement all 256 Boolean functions of three variables. The first function that was not calculated is number 0x16, or binary 00010110, which represents the function $\bar{x}yz + x\bar{y}z + xy\bar{z}$.

There are many symmetries that could be used to reduce the number of iterations. For example, for a given operation *op* and operands *x* and *y*, it is not necessary to evaluate both *op*(*x, y*) and *op*(*y, x*), because if *op*(*x, y*) is evaluated, then *op*(*y, x*) will result from *op'*(*x, y*) where *op'* is another of the 16 binary operations. Similarly, it is not necessary to evaluate *op*(*x, x*), because that will be equal to *op'*(*x, y*) for some other function *op'*. Thus, the outermost loops that select combinations of operands to try could be written

```
for (i1 = 0; i1 < 3; i1++) {
for (i2 = i1 + 1; i2 < 3; i2++) {
```

and similarly for the other loops.

Another improvement results from observing that it is not necessary to include all 16 binary Boolean operations in the table. The operations numbered 0, 3, 5, 10, 12, and 15 can be omitted, reducing the loops that iterate over the operations from 16 to ten iterations. The argument in support of this is a little lengthy and is not given here.

The program can be easily changed to experiment with smaller instruction sets, or allow more instructions, or handle more variables. But be forewarned: The execution time increases dramatically with the number of instructions being allowed, because that determines the level of nesting in the main program. As a practical matter, you can’t go beyond five instructions.

```
/* Determines which of the 256 Boolean functions of three
variables can be implemented with three binary Boolean
instructions if the instruction set includes all 16 binary
Boolean operations. */

#include <stdio.h>

char found[256];

unsigned char boole(int op, unsigned char x,
                            unsigned char y) {
    switch (op) {
      case 0: return 0;
      case 1: return x & y;
      case 2: return x & ~y;
      case 3: return x;
      case 4: return ~x & y;
      case 5: return y;
      case 6: return x ^ y;
      case 7: return x | y;
      case 8: return ~(x | y);
      case 9: return ~(x ^ y);
      case 10: return ~y;
      case 11: return x | ~y;
      case 12: return ~x;
      case 13: return ~x | y;
      case 14: return ~(x & y);
      case 15: return 0xFF;
    }
}
#define NB 16            // Number of Boolean operations.
int main() {

   int i, j, o1, i1, i2, o2, j1, j2, o3, k1, k2;
   unsigned char fun[6];// Truth table, 3 columns for
                        // x, y, and z, and 3 columns
                        // for computed functions.

   fun[0] = 0xOF;       // Truth table column for x,
   fun[1] = 0x33;       // y,
   fun[2] = 0x55;       // and z.

   for (o1 = 0; o1 < NB; ol++) {
   for (i1 = 0; i1 < 3; i1++) {
   for (i2 = 0; i2 < 3; i2++) {
      fun[3] = boole(ol, fun[i1], fun[i2]);
      for (o2 = 0; o2 < NB; o2++) {
      for (j1 = 0; j1< 4; j1++) {
      for (j2 = 0; j2 < 4; j2++) {
        fun[4] = boole(o2, fun[j1], fun[j2]);
        for (o3 = 0; o3 < NB; o3++) {
        for (k1 = 0; k1 < 5; k1++) {
        for (k2 = 0; k2 < 5; k2++) {
          fun[5] = boole(o3, fun[kl], fun[k2]);
          found[fun[5]] = 1;
        }}}
      }}}
   }}}
   printf("0 1 2 3 4 5 6 7 8 9 A B C D E F\n");
   for (i = 0; i < 16; i++) {
       printf("%X", i);
       for (j = 0; j < 16; j++)
          printf("%2d", found[16*i + j]);
       printf("\n");
   }
   return 0;
}
```

All ternary Boolean functions computable with three instructions, *continued*.

Chapter 3: Power-of-2 Boundaries

1. (a) (x + **4**) & –**8**.

(b) (x + **3**) & –**8**.

(c) (x + **3** + (($(x \overset{u}{\gg} 3)$) & **1**)) & –**8**.

Part (c) can be done in four instructions if the *extract* instruction is available; it can do ($(x \overset{u}{\gg} 3)$) & **1** in one instruction.

Note: Unbiased rounding preserves the average value of a large set of random integers.

2. The standard way to do part (a) is $((x + 5) \overset{u}{\div} 10) * 10$. If the remainder function is readily available, it can also be done with x + **5** – remu(x + **5, 10**), which saves a multiplication at the expense of an addition.

Part (b) is similar, but replace the 5 with 4 in the answer for part (a).

Part (c): Use the fact that an integer is an odd multiple of 10 if and only if it is an odd multiple of 2.

```
r = x % 10;
y = x - r;
if (r > 5 | (r == 5 & (y & 2) != 0)
    y = y + 10;
```

An alternative (must have $x \le 2^{32} - 6$):

```
r = (x + 5)%10;
y = x + 5 - r;
if (r == 0 & (y & 2) != 0)
    y = y - 10;
```

3. A possible implementation in C is shown below.

```
int loadUnaligned(int *a) {
   int *alo, *ahi;
   int xlo, xhi, shift;

   alo = (int *)((int)a & -4);
   ahi = (int *)(((int)a + 3) & -4);
   xlo = *alo;
```

```
      xhi = *ahi;
      shift = ((int)a & 3) << 3;
      return ((unsigned)xlo >> shift) | (xhi <<; (32-shift));
   }
```

Chapter 4: Arithmetic Bounds

1. For *a* = *c* = **0**, inequalities (5) become

$$\mathbf{0} \overset{u}{\le} x - y \overset{u}{\le} \mathbf{2^{32} - 1} \quad \text{if} \quad -d < 0 \ \text{ and } \ b \ge 0,$$

$$-d \overset{u}{\le} x - y \overset{u}{\le} b \quad \text{otherwise.}$$

Because the quantities are unsigned, -*d* < 0 is equivalent to *d* ≠ 0, and *b* ≥ 0 is true. Therefore, the inequalities simplify to

$$\mathbf{0} \overset{u}{\le} x - y \overset{u}{\le} \mathbf{2^{32} - 1} \quad \text{if } d \ne 0,$$

$$\mathbf{0} \overset{u}{\le} x - y \overset{u}{\le} b \quad \text{if } d = 0.$$

This is simply the observation that if *d*= **0**, then *y*= **0** and so, trivially, **0** ≤ *x*– *y*≤ *b*. On the other hand, if *d* ≠ 0, then the difference can attain the value **0** by choosing *x*= *y*= **0**, and it can attain the maximum unsigned number by choosing *x*= **0** and *y*= **1**.

2. If `a` = 0, the test `if (temp >= a)` is always true. Therefore, when the first position (from the left) is found in which the bits of `b` and `d` are 1, the program sets that bit of `b` equal to `0` and the following bits equal to 1, and returns that value *or*'ed with `d`. This can be accomplished more simply with the following replacement for the body of the procedure. The `if` statement is required only on machines that have mod 32 shifts, such as the Intel x86 family.

```
temp = nlz(c & d);
if (temp == 0) return 0xFFFFFFFF;
m = 1 << (32 - temp);
return b | d | (m - 1);
```

For example, suppose

$$\mathbf{0} \le x \le \mathbf{0b01001000}, \quad \text{and}$$

$$\mathbf{0b00000011} \le y \le \mathbf{0b00101010}.$$

Then to find the maximum value of *x*| *y*, the procedure is to scan from the left for the first position in which *b* and *d* are both **1**. The maximum value is *c*| *d* for bits to the left of that position, and **1**'s for bits at and to the right of that position. For the example, this is **0b01001000 | 0b00101010 | 0b0000 1111** = **0b0110 1111**.

Chapter 5: Counting Bits

1. A version from Norbert Juffa:

```
   int ntz (unsigned int n) {
    static unsigned char tab[32] =
    {  0, 1, 2, 24, 3, 19, 6, 25,
      22, 4, 20, 10, 16, 7, 12, 26,
      31, 23, 18, 5, 21, 9, 15, 11,
      30, 17, 8, 14, 29, 13, 28, 27
    };
    unsigned int k;
    n = n & (-n);      /* isolate lsb */
#if defined(SLOW_MUL)
    k = (n << 11) - n;
    k = (k << 2) + k;
    k = (k << 8) + n;
    k = (k << 5) - k;
#else
    k = n * 0x4d7651f;
#endif
    return n ? tab[k>>27] : 32;
}
```

2. $x \overset{u}{\div} (x \,\&\, -x)$. This is used in the snoob function (page 15).

3. Denote the parallel prefix operation applied to *x* by PP-XOR(*x*). Then, if $y = \text{PP-XOR}(x),\ x = y \oplus (y \overset{u}{\gg} 1)$. To see this, let *x* be the 4-bit quantity *abcd* (where each letter denotes a single bit). Then

$$y = \text{PP-XOR}(x) = (a)(a \oplus b)(a \oplus b \oplus c)(a \oplus b \oplus c \oplus d),$$

$$y \overset{u}{\gg} 1 = (0)(a)(a \oplus b)(a \oplus b \oplus c), \quad \text{so that}$$

$$y \oplus (y \overset{u}{\gg} 1) = (a)(b)(c)(d).$$

For the parallel suffix operation, if *y*= PS-XOR(*x*) then, as you might guess, *x*= *y*   (*y*<< 1).

Chapter 6: Searching Words

1. Length and position of the longest string of 1's (c.f. Norbert Juffa):

```
int fmaxstr1(unsigned x, int *apos) {
   int k;
   unsigned oldx;

   oldx = 0;
   for (k = 0; x != 0; k++) {
      oldx = x;
      x &= 2*x;
   }
   *apos = nlz(oldx);
   return k;
}
```

2. As said in the text, this can be done by first left-propagating the 0's in *x* by *n* – 1 positions, and then finding the shortest string of 1's in the revised *x*. A good way to do the left-propagation is to use the code of Figure 6–5 on page 125, which is logarithmic in its execution time. (But the second part of the algorithm is linear in the length of the shortest string of 1's in the revised *x*.) The code is shown below. It assumes that 1 ≤ *n* ≤ 32. In the "not found" case, the function returns with apos = 32. In this case, the length should be regarded as undefined, but it happens to return a length of *n* – 1.

```
int bestfit(unsigned x, int n, int *apos) {
   int m, s;

   m = n;
   while (m > 1) {
      s = m >> 1;
      x = x & (x << s);
      m = m - s;
   }
   return fminstr1(x, apos) + n - 1;
}
```

3. The code below uses an expression from page 12 for turning off the rightmost contiguous string of 1's.

```
int fminstr1(unsigned x, int *apos) {
   int k, kmin, y0, y;
   unsigned int x0, xmin;

   kmin = 32;
   y0 = pop(x);
   x0 = x;
   do {
      x = ((x & -x) + x) & x;   // Turn off rightmost
      y = pop(x);               // string.
      k = y0 - y;               // k = length of string
      if (k <= kmin) {          // turned off.
         kmin = k;              // Save shortest length
         xmin = x;              // found, and the string.
      }
      y0 = y;
   } while (x != 0);
   *apos = nlz(x0 ^ xmin);
   return kmin;
}
```

The function executes in 5 + 11*n* instructions, where *n* is the number of strings of 1's in *x*, for *n* ≥ 1 (that is, for *x* ≠ 0.) This assumes the *if*-test goes either way half the time, and that pop(*x*) and nlz(*x*) count as one instruction each. By making changes to the sense of the "`if (k <= kmin)`" test, and to the initialization of `kmin`, it can be made to find the longest string of 1's, and either the leftmost or the rightmost in the case of equally long strings. It is also easily modified to perform the "best fit" function.

4. The first bit of *x* will be 1, and hence mark the beginning of a string of 1's, with probability 0.5. Any other bit marks the beginning of a string of 1's with probability 0.25 (it must be 1, and the bit to its left must be 0). Therefore the average number of strings of 1's is 0.5 + 31•0.25 = 8.25.

5. One would expect the vast majority of words, if they are fairly long, to contain a string of 1's of length 1. For, if it begins with 10, or ends with 01, or contains the string 010, then its shortest contained string of 1's is of length 1. Therefore the average length is probably just slightly more than 1.

An exhaustive check of all 2 words shows that the average length of the shortest string of 1's is approximately 1.011795.

6. (Solution by John Gunnels) This problem is surprisingly difficult, but the technique used is a good one to know. The solution is based on a recursion that counts the number of words in each of four sets, as shown in the table below. In this table, "singleton" means a string of 1's of length 1, "nnn" denotes a string of length ≥ 0 that does not contain a singleton, and "sss" means a string of length ≥ 1 that contains a singleton. The ellipsis means 0 or more of the preceding bit. Every binary word is in one and only one of these four sets.

Set	**Words of the Form**	**Description**
A	nnn0… or null	Does not have a singleton, but might at the next step
B	nnn01 or 1	Has a singleton, but might not at the next step
C	nnn011… or 11…	Does not have a singleton, and will not at the next step
D	sss0 or sss01…	Has a singleton, and will at the next step

At each step, a bit is appended to the right-hand end of the word. As this is done, a word moves from one set to another as shown below. It moves to the left alternative if a 0 is appended, and to the right alternative if a 1 is appended.

$$A \Rightarrow A \text{ or } B$$
$$B \Rightarrow D \text{ or } C$$
$$C \Rightarrow A \text{ or } C$$
$$D \Rightarrow D \text{ or } D$$

For example, the word 1101 is in set *B*. If a 0 is appended, it becomes 11010, which is in set *D*. If a 1 is appended, it becomes 11011, which is in set *C*.

Let a_n, b_n, c_n, and d_n denote the sizes of sets *A, B, C*, and *D*, respectively, after *n* steps (when the words are of length *n*). Then

$$\begin{aligned} a_{n+1} &= a_n + c_n, \\ b_{n+1} &= a_n, \\ c_{n+1} &= b_n + c_n, \quad \text{and} \\ d_{n+1} &= b_n + 2d_n. \end{aligned}$$

This is because set *A* at step *n* + 1 contains every member of set *A* at step *n*, with a 0 appended, and also every member of set *C* at step *n*, with a 0 appended. Set *B* at step *n* + 1 contains only every member of set *A* at step *n*, with a 1 appended, and so on.

The initial conditions are $a_0 = 1$ and $b_0 = c_0 = d_0 = 0$.

It is a simple matter to evaluate these difference equations with a computer program or even by hand. The result, for *n* = 32, is

$$\begin{aligned}
a_{32} &= 26{,}931{,}732,\\
b_{32} &= 15{,}346{,}786,\\
c_{32} &= 20{,}330{,}163,\\
d_{32} &= 4{,}232{,}358{,}615, \quad \text{and}\\
b_{32}+d_{32} &= 4{,}247{,}705{,}401.
\end{aligned}$$

The last line gives the number we are interested in—the number of words for which their shortest contained string of 1’s is of length 1. It is about 98.9 percent of the number of 32-bit words (2^{32}).

What about a closed-form solution? This is also difficult to obtain. We will just sketch a solution.

Let $e_n = b_n + d_n$, which is the quantity we desire to find. Then, from the difference equations, and using the fact that $a_n + b_n + c_n + d_n = 2^n$,

$$\begin{aligned}
e_n &= b_n + d_n\\
&= 2^n - a_n - c_n\\
&= 2^n - a_{n+1}.
\end{aligned}$$

Thus, if we can find a closed-form formula for a_n, we will have one for e_n.

We can find a single-variable difference equation for a_n as follows. From the difference equations,

$$\begin{aligned}
a_n &= a_{n-1} + c_{n-1}\\
&= a_{n-1} + b_{n-2} + c_{n-2}\\
&= a_{n-1} + a_{n-3} + c_{n-2}\\
&= a_{n-1} + a_{n-3} + a_{n-1} - a_{n-2}\\
&= 2a_{n-1} - a_{n-2} + a_{n-3}.
\end{aligned}$$

This difference equation can be solved by well-known methods. The process is a bit lengthy and messy and won’t be gone into here. It involves the solution of a cubic polynomial that has two complex roots. When combined with the equation for e_n, we obtain, approximately,

$$\begin{aligned}
e_n \approx 2^n &- 0.41150 \cdot 1.7549^{n+1}\\
&-(0.29425 - 0.13811i)(0.12256 + 0.74486i)^{n+1}\\
&-(0.29425 + 0.13811i)(0.12256 - 0.74486i)^{n+1}.
\end{aligned}$$

If n is an integer, the imaginary parts cancel out, which is not hard to prove. (*Hint:* If x and y are complex conjugates, then so are x^n and y^n.)

We can get a formula involving only real numbers. The real part of the second term of the formula above is certainly less than

$$|0.29425 - 0.13811\,i\,|\cdot|0.12256 + 0.74486i\,|^{n+1}$$

which is, for $n = 0$,

$$0.32505 \cdot 0.75488 \approx 0.24537,$$

and is still smaller for $n > 0$. The same holds for the last term of the equation for e_n. Therefore the real part of the last two terms sum to less than 0.5. Since e_n is known a priori to be an integer, this means that e_n is given by the first term rounded to the nearest integer, or

$$e_n \approx \lfloor 2^n - 0.41150 \cdot 1.7549^{n+1} + 0.5 \rfloor.$$

7. Briefly, this problem can be solved by using 10 sets of words, described below. In this table, "nnn" denotes a string of length ≥ 0 whose shortest contained string of 1's is of length 0 or is ≥ 3, "ddd" denotes a string of length ≥ 2 whose shortest contained string of 1's is of length 2, and "sss" denotes a string of length ≥ 1 whose shortest contained string of 1's is of length 1. (The sets keep track of the words that contain a singleton at a position other than the rightmost, because such words will never have a shortest contained string of 1's of length 2.) The ellipsis means 0 or more of the preceding bit.

Set	Words of the Form	Set	Words of the Form
A	nnn0… or null	F	ddd01
B	nnn01 or 1	G	ddd011
C	nnn011 or 11	H	ddd0111…
D	nnn0111… or 111…	I	sss0
E	ddd0	J	sss01…

At each step, as a bit is appended to the right-hand end of a word from one of these sets, it moves to another set as shown below. It moves to the left alternative if a 0 is appended, and to the right alternative if a 1 is appended.

$$\begin{array}{ll}
A \Rightarrow A \text{ or } B & F \Rightarrow I \text{ or } G \\
B \Rightarrow I \text{ or } C & G \Rightarrow E \text{ or } H \\
C \Rightarrow E \text{ or } D & H \Rightarrow E \text{ or } H \\
D \Rightarrow A \text{ or } D & I \Rightarrow I \text{ or } J \\
E \Rightarrow E \text{ or } F & J \Rightarrow I \text{ or } J
\end{array}$$

Let a_n, b_n, ..., j_n denote the sizes of sets A, B, n steps (when the words are of length n). Then

$$\begin{aligned}
a_{n+1} &= a_n + d_n & f_{n+1} &= e_n \\
b_{n+1} &= a_n & g_{n+1} &= f_n \\
c_{n+1} &= b_n & h_{n+1} &= g_n + h_n \\
d_{n+1} &= c_n + d_n & i_{n+1} &= b_n + f_n + i_n + j_n \\
e_{n+1} &= c_n + e_n + g_n + h_n & j_{n+1} &= i_n + j_n
\end{aligned}$$

The initial conditions are $a_0 = 1$ and all other variables are 0.

The quantity we are interested in, the number of words whose shortest contained string of 1's is of length 2, is given by $c_n + e_n + g_n + h_n$. For $n = 32$, the difference equations give for this the value 44,410,452, which is about 1.034 percent of the number of 32-bit words. As an additional result, the number of words whose shortest contained string of 1's is of length 1 is given by $b_n + f_n + i_n + j_n$, which for $n = 32$ evaluates to 4,247,705,401, confirming the result of the preceding exercise.

This is as far as we are going with this problem.

Chapter 7: Rearranging Bits and Bytes

1. An ordinary integer can be incremented by complementing a certain number of consecutive low-order bits.[1] For example, to add **1** to **0x321F**, it suffices to apply the *exclusive or* operation to it with the mask **0x003F**. Similarly, to increment a reversed integer, it suffices to complement some high-order bits with a mask that consists of an initial string of 1's followed by 0's. Möbius's formula computes this mask and applies it to the reversed integer. (The method in the text that uses the nlz operation also does this.)

For an ordinary integer, the mask consists of 0's followed by 1's from the rightmost 0-bit to the low-order bit. The integer that consists of a 1-bit at the position of the rightmost 0-bit in i is given by the expression $\neg i \,\&\, (i + \mathbf{1})$ (see Section 2–1). To increment an ordinary integer x, we would compute a mask by right-propagating the 1-bit in this integer, and then *exclusive or* the result to x. To increment a reversed integer, we need to compute the reflection, or bit reversal, of that mask. The one-bit (power of 2) quantity $\neg i \,\&\, (i + \mathbf{1})$ can be reflected by dividing it into $m/\mathbf{2}$. (This step is the key to this algorithm.) For example, in the case of 4-bit integers, $m/\mathbf{2} = \mathbf{8}$. **8** / **1** = **8, 8** / **2** = **4, 8** / **4** = **2**, and **8** / **8** = **1**. To compute the mask, it is necessary only to left-propagate the 1-bit of the quotient, which is done by subtracting the quotient from m. Finally, the mask is *exclusive-or*'ed to the reversed integer, which produces the next reversed integer.

As an example, suppose the integers are eight bits in length, so that m = **256**. Let i = **19** (binary **00010011**), so that *revi* = binary **11001000**. Then $\neg i \,\&\, (i + 1)$ = binary **00000100** (decimal **4**). Dividing this into $m/\mathbf{2}$ gives a quotient of **32** (binary **00100000**). Subtracting this from m gives binary **1110 0000**. Finally, *exclusive or*'ing this mask to *revi* gives binary **00101000**, which is the reversed integer for decimal **20**.

2. Notice that

$$\begin{aligned} m_0 &= 2 \cdot \texttt{0x11111111} \\ m_1 &= \texttt{0xC} \cdot \texttt{0x01010101} \\ m_2 &= \texttt{0xF0} \cdot \texttt{0x00010001}, \text{ and} \\ m_3 &= \texttt{0xFF00} \cdot \texttt{0x00000001}. \end{aligned}$$

Also, notice that

$$\begin{aligned} \texttt{0x11111111} &= \lfloor 2^{32}/15 \rfloor, \\ \texttt{0x01010101} &= \lfloor 2^{32}/255 \rfloor, \\ \texttt{0x00010001} &= \lfloor 2^{32}/(2^{16}-1) \rfloor, \text{ and} \\ \texttt{0x00000001} &= \lfloor 2^{32}/(2^{32}-1) \rfloor. \end{aligned}$$

Thus, we have the formulas

$$\begin{aligned} m_0 &= (2-1)2\lfloor 2^{32}/(2^4-1) \rfloor, \\ m_1 &= (2^2-1)2^2\lfloor 2^{32}/(2^8-1) \rfloor, \\ m_2 &= (2^4-1)2^4\lfloor 2^{32}/(2^{16}-1) \rfloor, \text{ and} \\ m_3 &= (2^8-1)2^8\lfloor 2^{32}/(2^{32}-1) \rfloor. \end{aligned}$$

In general,

$$m_k = (2^{2^k}-1)2^{2^k}\lfloor 2^W/(2^{2^{k+2}}-1) \rfloor,$$

where W is the length of the word being shuffled, which must be a power of 2.

3. It is necessary only to change the two lines

```
s = s + b;
x = x >> 1;
```

to

```
s = s + 1;
x = x >> b;
```

4. Any true LRU algorithm must record the complete order of references to the n cache lines in a set. Since there are $n!$ orderings of n things, any implementation of LRU must use at least $\lceil \log_2 n! \rceil$ memory bits. The table below compares this to the number of bits required by the reference matrix method.

Degree of Associativity	Theoretical Minimum	Reference Matrix Method
2	1	1
4	5	6
8	16	28
16	45	120
32	118	496

Chapter 8: Multiplication

1. As shown in Section 8–3, if x and y are the multiplication operands interpreted as signed integers, then their product interpreted as unsigned integers is

$$(x + 2^{32}x_{31})(y + 2^{32}y_{31}) = xy + 2^{32}(x_{31}\,y + y_{31}x) + 2^{64}x_{31}y_{31},$$

where x_{31} and y_{31} are the sign bits of x and y, respectively, as integers 0 or 1. Because the product differs from xy by a multiple of 2^{32}, the low-order 32 bits of the product are the same.

2. Method 1: Chances are the machine has a multiplication instruction that gives the low-order 32 bits of the product of two 32-bit integers. That is,

```
low = u*v;
```

Method 2: Just before the `return` statement, insert

```
low = (w1 << 16) + (w0 & 0xFFFF);
```

Method 3: Save the products `u1*v0` and `u0*v1` in temporaries `t1` and `t2`. Then

```
low = ((t1 + t2) << 16) + w0;
```

Methods 2 and 3 are three basic RISC instructions each, and they work for both `mulhs` and its unsigned counterpart (and may be faster than method 1).

3. Partition the 32-bit operands u and v into 16-bit unsigned components a, b, c, and d, so that

$$\begin{aligned} u &= 2^{16}a + b \quad \text{and} \\ v &= 2^{16}c + d, \end{aligned}$$

where $0 \le a, b, c, d \le 2^{16} - 1$. Let

$$\begin{aligned} p &= ac, \\ q &= bd, \quad \text{and} \\ r &= (-a + b)(c - d). \end{aligned}$$

Then $uv = 2^{32}\,p + 2^{16}(r + p + q) + q$, which is easily verified.

Now $0 \le p, q \le 2^{32} - 2^{17} + 1$, so that p and q can be represented by 32-bit unsigned integers. However, it is easily calculated that

$$-2^{32} + 2^{17}\ 1 \le r \le 2^{32} - 2^{17} + 1,$$

so that r is a signed 33-bit quantity. It will be convenient to represent it by a signed 64-bit integer, with the high-order 32 bits being either all 0's or all 1's. The machine's multiply instruction will compute the low-order 32 bits of r, and the high-order 32 bits can be ascertained from the values of $-a + b$ and $c - d$. These are 17-bit signed integers. If they have opposite signs and are nonzero, then r is negative and hence its high-order 32 bits are all 1's. If they have the same signs or either is 0, then r is nonnegative and hence its high-order 32 bits are all 0's. The test that either $-a + b$ or $c - d$ is 0 can be done by testing only the low-order 32 bits of r. If they are 0, then one of the factors must be 0, because $r < 2^{32}$.

These considerations lead to the following function for computing the high-order 32 bits of the product of u and v.

```
unsigned mulhu(unsigned u, unsigned v) {
    unsigned a, b, c, d, p, q, rlow, rhigh;

    a = u >> 16; b = u & 0xFFFF;
    c = v >> 16; d = v & 0xFFFF;

    p = a*c;
    q = b*d;
    rlow = (-a + b)*(c - d);
    rhigh = (int)((-a + b)^(c - d)) >> 31;
    if (rlow == 0) rhigh = 0; // Correction.
    q = q + (q >> 16);       // Overflow cannot occur here.
    rlow = rlow + p;
    if (rlow < p) rhigh = rhigh + 1;
    rlow = rlow + q;
    if (rlow < q) rhigh = rhigh + 1;

    return p + (rlow >> 16) + (rhigh << 16);
}
```

After computing `p`, `q`, `rlow`, and `rhigh`, the function does the following addition:

```
        |...... p.......|
|....rhigh..... ||..... rlow...... |
                 |....... p....... |
                 |....... q....... |
                          |....... q....... |
```

The statement "`if (rlow < p) rhigh = rhigh + 1`" is adding 1 to `rhigh` if there is a carry from the addition of `p` to `rlow` in the previous statement.

The low-order 32 bits of the product can be obtained from the following expression, inserted just after the "correction" step above:

```
q + ((p + q + rlow) << 16)
```

A branch-free version follows.

```
unsigned mulhu(unsigned u, unsigned v) {
    unsigned a, b, c, d, p, q, x, y, rlow, rhigh, t;

    a = u >> 16; b = u & 0xFFFF;
    c = v >> 16; d = v & 0xFFFF;

    p = a*c;
    q = b*d;
    x = -a + b;
    y = c - d;
    rlow = x*y;
    rhigh = (x ^ y) & (rlow | -rlow);
    rhigh = (int)rhigh >> 31;

    q = q + (q >> 16); // Overflow cannot occur here.
    t = (rlow & 0xFFFF) + (p & 0xFFFF) + (q & 0xFFFF);
    p += (t >> 16) + (rlow >> 16) + (p >> 16) + (q >> 16);
    p += (rhigh << 16);
    return p;
}
```

These functions have more overhead than the four-multiplication function of Figure 8–2 on page 174, and will be superior only if the machine's multiply instruction is slower than that found on most modern computers. In "bignum" arithmetic (arithmetic on multiword integers), the time to multiply is substantially more than the time to add two integers of similar sizes. For that application, a method known as Karatsuba multiplication [Karat] applies the three-multiplication scheme recursively, and it is faster than the straightforward four-multiplication scheme for sufficiently large numbers. Actually, Karatsuba multiplication, as usually described, uses

$$\begin{aligned} p &= ac, \\ q &= bd, \\ r &= (a+b)(c+d), \quad \text{and} \\ uv &= 2^{32}p + 2^{16}(r-p-q) + q. \end{aligned}$$

For our application, that method does not work out very well because *r* can be nearly as large as 2^{34}, and there does not seem to be any easy way to calculate the high-order two bits of the 34-bit quantity *r*.

A signed version of the functions above has problems with overflow. It is just as well to use the unsigned function and correct it as described in Section 8–3 on page 174.

Chapter 9: Integer Division

1. Let $x = x_0 + \delta$, where x_0 is an integer and $0 \le \delta < 1$. Then $\lceil -x \rceil = \lceil -x_0 - \delta \rceil = -x_0$ by the definition of the ceiling function as the next integer greater than or equal to its argument. Hence $-\lceil -x \rceil = x_0$, which is $\lfloor x \rfloor$.

2. Let *n* / *d* denote the quotient of signed, truncating, integer division. Then we must compute

$$\begin{array}{ll} n/d, & \text{if } n \ge 0, d > 0, \\ n/d - 1, & \text{if } n < 0, d > 0, \\ n/d, & \text{if } n \ge 0, d < 0, \\ n/d + 1, & \text{if } n < 0, d < 0. \end{array}$$

(If *d* = 0 the result is immaterial.) This can be computed as *n* / *d* + *c*, where

$$c = ((n \overset{s}{\gg} 31) \oplus (d \overset{s}{\gg} 31)) - (d \overset{s}{\gg} 31),$$

which is four instructions to compute *c* (the term $d \overset{s}{\gg} 31$ commons). Another way to compute *c* in four instructions, but with the shifts unsigned, is

$$c = (d \overset{u}{\gg} 31) - ((n \oplus d) \overset{u}{\gg} 31).$$

If your machine has mod-32 shifts, *c* can be computed in three instructions:

$$c = (n \overset{s}{\gg} 31) \overset{u}{\gg} (d \overset{s}{\gg} 31).$$

For the remainder, let rem(*n, d*) denote the remainder upon dividing the signed integer *n* by the signed integer *d*, using truncating division. Then we must compute

$$\begin{array}{ll} \text{rem}(n, d), & \text{if } n \ge 0, d > 0, \\ \text{rem}(n, d) + d, & \text{if } n < 0, d > 0, \\ \text{rem}(n, d), & \text{if } n \ge 0, d < 0, \\ \text{rem}(n, d) - d, & \text{if } n < 0, d < 0. \end{array}$$

The amount to add to rem(*n, d*) is 0 or the absolute value of *d*. This can be computed from

$$\begin{aligned} |d| &= (d \oplus (d \overset{s}{\gg} 31)) - (d \overset{s}{\gg} 31), \\ c &= |d| \,\&\, (n \overset{s}{\gg} 31), \end{aligned}$$

which is five instructions to compute *c*. It can be computed in four instructions if your machine has mod-32 shifts and you use the multiply instruction (details omitted).

3. To get the quotient of floor division, it is necessary only to subtract 1 from the quotient of truncating division if the dividend and divisor have opposite signs:

$$n/d - ((n \oplus d) \overset{u}{\gg} 31).$$

For the remainder, it is necessary only to add the divisor to the remainder of truncating division if the dividend and divisor have opposite signs:

$$\text{rem}(n, d) + (((n \oplus d) \overset{s}{\gg} 31) \,\&\, d).$$

4. The usual method, most likely, is to compute $\lfloor (n + d - 1)/d \rfloor$. The problem is that $n + d - 1$ can overflow. (Consider computing 12/5 on a 4-bit machine.)

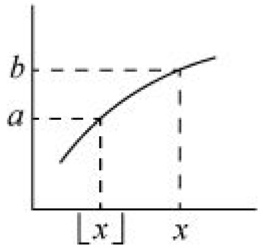
Another standard method is to compute $q = \lfloor n / d \rfloor$ using the machine's *divide* instruction, then compute the remainder as $r = n - qd$, and if r is nonzero, add **1** to q. (Alternatively, add **1** if $n \neq qd$.) This gives the correct result for all n and $d \neq \mathbf{0}$, but it is somewhat expensive because of the multiply, subtract, and conditional add of **1**. On the other hand, if your machine's *divide* instruction gives the remainder as a by-product, and especially if it has an efficient way to do the computation $q = q + (r \not\equiv \mathbf{0})$, then this is a good way to do it.

Still another way is to compute $q = \lfloor (n - \mathbf{1}) / d \rfloor + \mathbf{1}$. Unfortunately, this fails for $n = \mathbf{0}$. It can be fixed if the machine has a simple way to compute the $x \not\equiv \mathbf{0}$ predicate, such as by means of a *compare* instruction that sets the target GPR to the integer **1** or **0** (see also Section 2–12 on page 23). Then one can compute:

$$c \leftarrow (x \neq 0)$$
$$q \leftarrow \lfloor (n - c)/d \rfloor + c$$

Lastly, one can compute $q = \lfloor (n - \mathbf{1}) / d \rfloor + \mathbf{1}$ and then change the result to **0** if $\mathbf{n} = \mathbf{0}$, by means of a conditional move or select instruction, for example.

5. Let $f(\lfloor x \rfloor) = a$ and $f(x) = b$, as illustrated below.



If b is an integer, then by property (c), x is also, so that $\lfloor x \rfloor = x$, and there is nothing to prove. Therefore, assume in what follows that b is not an integer, but a may or may not be.

There cannot be an integer k such that $a < k \leq b$, because if there were, there would be an integer between $\lfloor x \rfloor$ and x (by properties (a), (b), and (c)), which is impossible. Therefore $\lfloor a \rfloor = \lfloor b \rfloor$; that is, $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$.

As examples of the utility of this, we have, for a and b integers,

$$\left\lfloor \frac{\lfloor x \rfloor + a}{b} \right\rfloor = \left\lfloor \frac{x + a}{b} \right\rfloor$$
$$\left\lfloor \sqrt{\lfloor x \rfloor} \right\rfloor = \left\lfloor \sqrt{x} \right\rfloor$$
$$\left\lfloor \log_2(\lfloor x \rfloor) \right\rfloor = \left\lfloor \log_2(x) \right\rfloor$$

It can similarly be shown that if $f(x)$ has properties (a), (b), and (c), then

$$f(\lceil x\rceil)\rceil = \lceil f(x)\rceil.$$

Chapter 10: Integer Division by Constants

1. (a) If the divisor is even, then the low-order bit of the dividend does not affect the quotient (of floor division); if it is 1 it makes the remainder odd. After turning this bit off, the remainder of the division will be an even number. Hence for an even divisor *d*, the remainder is at most $d - 2$. This slight change in the maximum possible remainder results in the maximum multiplier *m* being a *W*-bit number rather than a $(W + 1)$-bit number (and hence the `shrxi` instruction is not needed), as we will now see. In fact, we will investigate what simplifications occur if the divisor ends in *z* 0-bits, that is, if it is a multiple of 2^z, for $z \ge 0$. In this case, the *z* low-order bits of the dividend can be cleared without affecting the quotient, and after clearing those bits, the maximum remainder is $d - 2^z$.

Following the derivation of Section 10–9 on page 230, but changed so that the maximum remainder is $d - 2^z$, we have $n_c = 2^W - \text{rem}(2^W, d) - 2^z$, and inequality (24a) becomes

$$2^W - d \le n_c \le 2^W - 2^z.$$

Inequality (25) becomes

$$\frac{2^p}{d} \le m < \frac{2^p}{d}\frac{n_c + 2^z}{n_c}.$$

Equation (26) is unchanged, and inequality (27) becomes

$$2^p > \frac{n_c}{2^z}(d - 1 - \text{rem}(2^p - 1, d)). \qquad (27')$$

Inequality (28) becomes

$$1 \le 2^p \le \frac{2n_c}{2^z}(d-1) + 1.$$

In the case that *p* is not forced to equal *W*, combining these inequalities gives

$$\frac{1}{d} \le m < \frac{2n_c(d-1) + 2^z n_c + 2^z}{2^z d}\frac{}{n_c},$$

$$1 \le m < \frac{2d - 2 + 2^z/n_c}{2^z d}(n_c + 2^z),$$

$$1 \le m < \frac{2}{2^z}(n_c + 2^z) \le \frac{2}{2^z}2^W.$$

Thus if $z \ge 1$, $m < 2^W$, so that *m* fits in a *W*-bit word. The same result follows in the case that *p* is forced to equal *W*.

To calculate the multiplier for a given divisor, calculate n_c as shown above, then find the smallest $p \geq W$ that satisfies (27'), and calculate m from (26). As an example, for d = 14 and W = 32, we have $n_c = 2^{32} - \text{rem}(2^{32}, 14) - 2$ = 0xFFFFFFFA. Repeated use of (27′) gives p = 35, from which (26) gives $m = (2^{35} + 14 - 1 - 3) / 14$ = 0x92492493. Thus, the code to divide by 14 is

```
ins    n,R0,0,1      Clear low-order bit of n.
li     M,0x92492493  Load magic number.
mulhu  q,M,n         q = floor(M*n/2**32).
shri   q,q,3         q = q/8.
```

(b) Again, if the divisor is a multiple of 2^z, then the low-order z bits of the dividend do not affect the quotient. Therefore, we can clear the low-order z bits of the dividend, and divide the divisor by 2^z, without changing the quotient. (The division of the divisor would be done at compile time.)

Using the revised n and d, both less than 2^{W-z}, (24a) becomes

$$2^{W-z}-d \leq n_c \leq 2^{W-z}-1$$

Equation (26) and inequality (27) are not changed, but they are to be used with the revised values of n_c and d. We omit the proof that the multiplier will be less than 2^W and give an example again for d = 14 and W = 32. In the equations, we use d = 7. Thus, we have $n_c = 2^{31} - \text{rem}(2^{31}, 7) - 1$ = 0x7FFFFFFF. Repeated use of (27) gives p = 34, from which (26) gives $m = (2^{34} + 5) / 7$ = 0x92492493, and the code to divide by 14 is

```
shri   n,n,1         Halve the dividend.
li     M,0x92492493  Load magic number.
mulhu  q,M,n         q = floor(M*n/2**32).
shri   q,q,2         q = q/4.
```

These methods should not *always* be used when the divisor is an even number. For example, to divide by 10, 12, 18, or 22 it is better to use the method described in the text, because there's no need for an instruction to clear the low-order bits of the dividend, or to shift the dividend right. Instead, the algorithm of Figure 10–3 on page 236 should be used, and if it gives an "add" indicator of 1 and the divisor is even, then one of the above techniques can be used to get better code on most machines. Among the divisors less than or equal to 100, these techniques are useful for 14, 28, 38, 42, 54, 56, 62, 70, 74, 76, 78, 84, and 90.

Which is better, (a) or (b)? Experimentation indicates that method (b) is preferable in terms of the number of instructions required, because it seems to always require either the same number of instructions as (a), or one fewer. However, there are cases in which (a) and (b) require the same number of instructions, but (a) yields a smaller multiplier. Some representative cases are shown below. The "Book" method is the code that Figure 10–3 gives. We assume here that the computer's *and immediate* instruction sign-propagates the high-order bit of the immediate field (our basic RISC would use the *insert* instruction).

$d = 6$

Book	(a)	(b)

`li    M,0xaaaaaaab`
`mulhu q,M,n`
`shri  q,q,2`
	`andi  n,n,-2`
	`li    M,0x2aaaaaab`
	`mulhu q,M,n`
		`shri  n,n,1`
		`li    M,0x55555556`
		`mulhu q,M,n`

$d = 28$

Book	(a)	(b)

`li    M,0x24924925`
`mulhu q,M,n`
`add   q,q,n`
`shrxi q,q,5`
	`andi  n,n,-4`
	`li    M,0x24924925`
	`mulhu q,M,n`
	`shri  q,q,2`
		`shri  n,n,2`
		`li    M,0x24924925`
		`mulhu q,M,n`

These techniques are not useful for signed division. In that case, the difference between the best and worst code is only two instructions (as illustrated by the code for dividing by 3 and by 7, shown in Section 10–3 on page 207). The fix-up code for method (a) would require adding 1 to the dividend if it is negative and odd, and subtracting 1 if the dividend is nonnegative and odd, which would require more than two instructions. For method (b), the fix-up code is to divide the dividend by 2, which requires three basic RISC instructions (see Section 10–1 on page 205), so this method is also not a winner.

2. Python code is shown below.

```
def magicg(nmax, d):
   nc = (nmax//d)*d - 1
   nbits = int(log(nmax, 2)) + 1
   for p in range(0, 2*nbits - 1):
      if 2**p > nc*(d - (2**p)%d):
         m = (2**p + d - (2**p)%d)//d
         return (m, p)
   print "Can't find p, something is wrong."
   sys.exit(1)
```

3. Because 81 = 3^4, we need for the starting value, the multiplicative inverse of *d* modulo 3. This is simply the remainder of dividing *d* by 3, because $1 \cdot 1 \equiv 1 \pmod 3$ and $2 \cdot 2 \equiv 1 \pmod 3$ (and if the remainder is 0, there is no multiplicative inverse). For *d* = 146, the calculation proceeds as follows.

$$\begin{aligned}
x_0 &= 146 \bmod 3 = 2, \\
x_1 &= 2(2 - 146 \cdot 2) = -580 \equiv 68 \pmod{81}, \\
x_2 &= 68(2 - 146 \cdot 68) = 674{,}968 \equiv 5 \pmod{81}, \\
x_3 &= 5(2 - 146 \cdot 5) = -3640 \equiv 5 \pmod{81}.
\end{aligned}$$

A fixed point was reached, so the multiplicative inverse of 146 modulo 81 is 5. Check: $146 \cdot 5 = 730 \equiv 1 \pmod{81}$. Actually, it is known *a priori* that two iterations suffice.

Chapter 11: Some Elementary Functions

1. Yes. The result is correct in spite of the double truncation. Suppose $\lfloor \sqrt{x} \rfloor = a$. Then by the definition of this operation, *a* is an integer such that $a^2 \le x$ and $(a + 1)^2 < x$.

Let $\lfloor \sqrt{a} \rfloor = b$. Then $b^2 \le a$ and $(b + 1)^2 < a$. Thus, $b^4 \le a^2$ and, because $a^2 \le x$, $b^4 \le x$.

Because $(b + 1)^2$ a, $(b + 1)2 \ge a + 1$, so that $(b + 1)^4 \ge (a + 1)^2$ Because $(a + 1)^2 x$, $(b + 1)^4 x$. Hence *b* is the integer fourth root of *x*.

This follows more easily from exercise 5 of Chapter 9.

2. Straightforward code is shown below.

```
int icbrt64(unsigned long long x) {
   int s;
   unsigned long long y, b, bs;

   y = 0;
   for (s = 63; s >= 0; s = s - 3) {
      y = 2*y;
      b = 3*y*(y + 1) + 1;
      bs = b << s;
      if (x >= bs && b == (bs >> s)) {
         x = x - bs;
         y = y + 1;
      }
   }
   return y;
}
```

Overflow of `b` (`bs` in the above code) can occur only on the second loop iteration. Therefore, another way to deal with the overflow is to expand the first two iterations of the loop, and then execute the loop only from `s` = 57 on down, with the phrase “`&& b == (bs >> s)`” deleted.

By inspection, the effect of the first two loop iterations is:

If $x \ge 2^{63}$, set $x = x - 2^{63}$ and set $y = 2$.

If $2^{60} \le x < 2^{63}$, set $x = x - 2^{60}$ and set $y = 1$.

If $x < 2^{60}$, set $y = 0$ (and don’t change *x*).

Therefore, the beginning of the routine can be coded as shown below.

```
y = 0;
if (x >= 0x1000000000000000LL) {
   if (x >= 0x8000000000000000LL) {
      x = x - 0x8000000000000000LL;
      y = 2;
   } else {
      x = x - 0x1000000000000000LL;
      y = 1;
   }
}
```

```
    for (s = 57; s >= 0; s = s - 3) {
          ...
```

And, as mentioned, the phrase "`&& b == (bs >> s)`" can be deleted.

3. Six [Knu2]. The binary decomposition method, based on $x^{23} = x^{16} \cdot x^4 \cdot x^2 \cdot x$, takes seven. Factoring x^{23} as $(x^{11})^2 \cdot x$ or as $((x^5)^2 \cdot x)^2 \cdot x$ also takes seven. But computing powers of x in the order x^2, x^3, x^5, x^{10}, x^{13}, x^{23}, in which each term is a product of two previous terms or of x, does it in six multiplications.

4. (a) x rounded down to an integral power of 2. (b) x rounded up to an integral power of 2 (in both cases, x itself if x is an integral power of 2).

Chapter 12: Unusual Bases for Number Systems

1. If B is a binary number and N is its base –2 equivalent, then

$$B \leftarrow \text{0x55555555} - (N \oplus \text{0x55555555}), \text{ and}$$
$$N \leftarrow (\text{0x55555555} - B) \oplus \text{0x55555555}.$$

2. An easy way to do this is to convert the base –2 number x to binary, add 1, and convert back to base –2. Using Schroeppel's formula and simplifying, the result is

$$((x \oplus \text{0xAAAAAAAA}) + 1) \oplus \text{0xAAAAAAAA}, \text{ or}$$
$$((x \oplus \text{0x55555555}) - 1) \oplus \text{0x55555555}.$$

3. As in exercise 1, one could convert the base –2 number x to binary, *and* with 0xFFFFFFF0, and convert back to base –2. This would be five operations. However, it can be done in four operations with either of the formulas below.[2]

$$(((x \oplus \text{0xAAAAAAAA}) - 10) \oplus \text{0xAAAAAAAA}) \,\&\, -16$$
$$(((x \oplus \text{0x55555555}) + 10) \oplus \text{0x55555555}) \,\&\, -16$$

The formulas below round a number *up* to the next greater power of 16.

$$(((x \oplus \text{0xAAAAAAAA}) + 5) \oplus \text{0xAAAAAAAA}) \,\&\, -16$$
$$(((x \oplus \text{0x55555555}) - 5) \oplus \text{0x55555555}) \,\&\, -16$$

There are similar formulas for rounding up or down to other powers of 2.

4. This is very easy to program in Python, because that language supports complex numbers.

```
import sys
import cmath

num = sys.argv[1:]
if len(num) == 0:
   print "Converts a base -1 + 1j number, given in decimal"
   print "or hex, to the form a + bj, with a, b real."
   sys.exit()
num = eval(num[0])
```

```
r = 0
weight = 1
while num > 0:
   if num & 1:
      r = r + weight;
   weight = (-1 + 1j)*weight
   num = num >> 1;
print ‘r =’, r
```

---

5. To convert a base – 1 + *i* number to its negative, either subtract it from 0 or multiply it by –1 (11101), using the rules for base – 1 + *i* arithmetic.

To extract the real part of a number *x*, add in the negative of its imaginary part. Process the bits of *x* in groups of four, starting at the right (low-order) end. Number the bits in each group 0, 1, 2, and 3, from the right. Then:

If bit 1 is on, add – *i* (0111) at the current group’s position.

If bit 2 is on, add 2 *i* (1110100) at the current group’s position.

If bit 3 is on, add –2 *i* (0100) at the current group’s position.

Bit 1 has a weight of – 1 + *i*, so adding in – *i* cancels its imaginary component. A similar remark applies to bits 2 and 3. There is no need to do anything for bit 0, because that has no imaginary component. Each group of four bits has a weight of – 4 times the weight of the group immediately to its right, because 10000 in base – 1 + *i* is – 4 decimal. Thus, the weight of bit *n* of *x* is a real number (– 4) times the weight of bit *n* – 4.

The example below illustrates extracting the real part of the base -1 + *i* number 101101101.

```
     1 0110 1101 x
        111 0100 2i added in for bit 2
            0100 −2i added in for bit 3
       0111      −i(−4) added in for bit 5
   111 0100      2i(−4) added in for bit 6
-------------
1100 1101 1101 sum
```

The reader may verify that *x* is 23 + 4*i*, and the sum is 23. In working out this addition, *many* carries are generated, which are not shown above. Several shortcuts are possible: If bits 2 and 3 are both on, there is no need to add anything in for these bits, because we would be adding in 2*i* and –2*i*. If a group ends in 11, these bits can be simply dropped, because they constitute a pure imaginary (*i*). Similarly, bit 2 can be simply dropped, as its weight is a pure imaginary (–2*i*).

Carried to its extreme, a method employing these kinds of shortcuts would translate each group of four bits independently to its real part. In some cases a carry is generated, and these carries would be added to the translated number. To illustrate, let us represent each group of four bits in hexadecimal. The translation is shown below.

$0 \Rightarrow 0$	$4 \Rightarrow 0$	$8 \Rightarrow \mathrm{C}$	$\mathrm{C} \Rightarrow \mathrm{C}$
$1 \Rightarrow 1$	$5 \Rightarrow 1$	$9 \Rightarrow \mathrm{D}$	$\mathrm{D} \Rightarrow 6$
$2 \Rightarrow 1\mathrm{D}$	$6 \Rightarrow 1\mathrm{D}$	$\mathrm{A} \Rightarrow 1$	$\mathrm{E} \Rightarrow 1$
$3 \Rightarrow 0$	$7 \Rightarrow 0$	$\mathrm{B} \Rightarrow \mathrm{C}$	$\mathrm{F} \Rightarrow \mathrm{C}$

The digits 2 and 6 have real part –1, which is written 1D in base – 1 + *i*. For these digits, replace the source digit with D and carry a 1. The carries can be added in using the basic rules of addition in base – 1 + *i*, but for hand work there is a more expedient way. After translation, there are only four possible digits: 0, 1, C, and D, as the translation table shows. Rules for adding 1 to these digits are shown in the left-hand column below.

$0 + 1 = 1$	$0 + 1\mathrm{D} = 1\mathrm{D}$
$1 + 1 = \mathrm{C}$	$1 + 1\mathrm{D} = 0$
$\mathrm{C} + 1 = \mathrm{D}$	$\mathrm{C} + 1\mathrm{D} = 1$
$\mathrm{D} + 1 = 1\mathrm{D}0$	$\mathrm{D} + 1\mathrm{D} = \mathrm{C}$

Adding 1 to D generates a carry of 1D (because 3 + 1 = 4). We will carry both digits to the same column. The right-hand column above shows how to handle the carry of 1D. In doing the addition, it is possible to get a carry of both 1 and 1D in the same column (the first carry from the translation and the second from the addition). In this case, the carries cancel each other, because 1D is –1 in base – 1 + *i*. It is not possible to get two carries of 1, or two of 1D, in the same column.

The example below illustrates the use of this method to extract the real part of the base – 1 + *i* number EA26 (written in hexadecimal).

```
EA26  x
  11  carries from the translation
11DD  x with its hex digits translated
----
110D  sum
```

The reader may verify that *x* is – 45 + 21 *i* and the sum is – 45.

Incidentally, a base – 1 + *i* number is real iff all of its digits, expressed in hexadecimal, are 0, 1, C, or D.

To extract the imaginary part from *x*, one can, of course, extract the real part and subtract that from *x*. To do it directly by the "shortcut" method, the table below shows the translation of each hexadecimal digit to its pure imaginary part.

$0 \Rightarrow 0$	$4 \Rightarrow 4$	$8 \Rightarrow 74$	$\mathrm{C} \Rightarrow 0$
$1 \Rightarrow 0$	$5 \Rightarrow 4$	$9 \Rightarrow 74$	$\mathrm{D} \Rightarrow 0$
$2 \Rightarrow 3$	$6 \Rightarrow 7$	$\mathrm{A} \Rightarrow 77$	$\mathrm{E} \Rightarrow 3$
$3 \Rightarrow 3$	$7 \Rightarrow 7$	$\mathrm{B} \Rightarrow 77$	$\mathrm{F} \Rightarrow 3$

Thus, a carry of 7 can occur, so we need addition rules to add 7 to the four possible translated digits of 0, 3, 4, and 7. These are shown in the left-hand column below.

$0+7=7$	$0+3=3$
$3+7=0$	$3+3=74$
$4+7=33$	$4+3=7$
$7+7=4$	$7+3=0$

Now a carry of 3 can occur, and the right-hand column above shows how to deal with that.

The example below illustrates the use of this method to extract the imaginary part of the base – 1 + *i* number 568A (written in hexadecimal).

```
568A  x
  77  carries from the translation
4747  x with its hex digits translated
----
4737  sum
```

The reader may verify that *x* is – 87 + 107 *i* and the sum is 107*i*.

A base – 1 + *i* number is imaginary iff all of its digits, expressed in hexadecimal, are 0, 3, 4, or 7.

To convert a number to its complex conjugate, subtract twice a number's imaginary part. A table can be used, as above, but the conversion is more complicated because more carries can be generated, and the translated number can contain any of the 16 hexadecimal digits. The translation table is shown below.

$0 \Rightarrow 0$	$4 \Rightarrow 74$	$8 \Rightarrow 38$	$C \Rightarrow C$
$1 \Rightarrow 1$	$5 \Rightarrow 75$	$9 \Rightarrow 39$	$D \Rightarrow D$
$2 \Rightarrow 6$	$6 \Rightarrow 2$	$A \Rightarrow 3E$	$E \Rightarrow 3A$
$3 \Rightarrow 7$	$7 \Rightarrow 3$	$B \Rightarrow 3F$	$F \Rightarrow 3B$

The carries can be added in using base – 1 + *i* arithmetic or by devising a table that does the addition a hexadecimal digit at a time. The table is larger than those above, because the carries can be added to any of the 16 possible hexadecimal digits.

Chapter 13: Gray Code

1. Proof sketch 1: It is apparent from the construction of the reflected binary Gray code.

Proof sketch 2: From the formula $G(x) = x \oplus (x \overset{u}{\gg} 1)$, it can be seen that G(*x*) is 1 at position *i* wherever there is a transition from 0 to 1 or from 1 to 0 from position *i* to the bit to the left of *i*, and is 0 otherwise. If *x* is even, there are an even number of transitions, and if *x* is odd, there are an odd number of transitions.

Proof sketch 3: By induction on the length of *x*, using the formula given above: The statement is true for the one-bit words 0 and 1. Let *x* be a binary word of length *n*, and assume inductively that the statement is true for *x*. If *x* is prepended with a 0-bit, G(*x*) is also prepended with a 0-bit, and the remaining bits are G(x). If x is prepended with a 1-bit, then G(x) is also prepended with a 1-bit, and its next most significant bit is complemented. The remaining bits are unchanged. Therefore, the number of 1-bits in G(x) is either increased by 2 or is unchanged.

Thus, one can construct a random number generator that generates integers with an even (or odd) number of 1-bits by using a generator of uniformly distributed integers, setting the least significant bit to 0 (or to 1), and converting the result to Gray code [Arndt].

2. (a) Because each column is a cyclic shift of column 1, the result follows immediately.

(b) No such code exists. This is not difficult to verify by enumerating all possible Gray codes for n = 3. Without loss of generality, one can start with

```
000
001
011
```

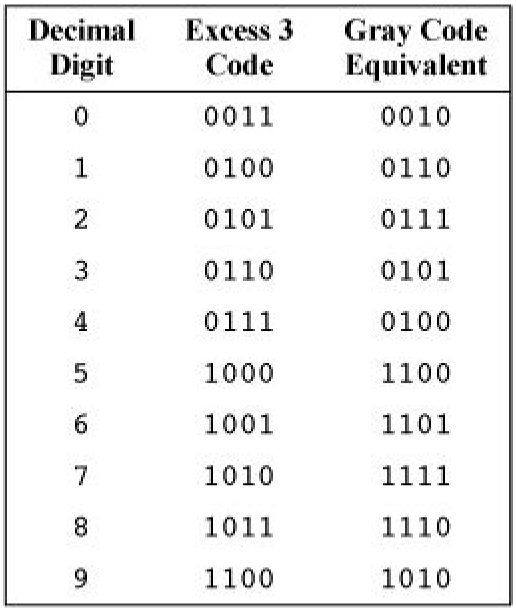
because any Gray code can be made to start that way by complementing columns and rearranging columns. Corollary: There is no STGC for n = 3 that has eight code words.

3. The code below was devised by reflecting the first five code words of the reflected binary Gray code.

```
0000
0001
0011
0010
0110
1110
1010
1011
1001
1000
```

Another code can be derived by taking the "excess 3" binary coded decimal (BCD) code and converting it to Gray. The result turns out to be cyclic. The excess 3 code for encoding decimal digits has the property that addition of coded words generates a carry precisely when addition of the decimal digits would.

EXCESS THREE GRAY CODE



Decimal Digit	Excess 3 Code	Gray Code Equivalent
0	0011	0010
1	0100	0110
2	0101	0111
3	0110	0101
4	0111	0100
5	1000	1100
6	1001	1101
7	1010	1111
8	1011	1110
9	1100	1010

4. It is a simple matter to derive a "mixed base" Gray code, using the principle of reflection. For a number with prime decomposition $2^{e_1}3^{e_2}5^{e_3}$, the columns of the Gray code should be in base $e_1 + 1$, $e_2 + 1$, $e_3 + 1$,.... For example, for the number $72 = 2^3 \cdot 3^2$, the list below shows a "base 4 - base 3" Gray code and the divisor of 72 that each code word represents.

00	1
01	3
02	9
12	18
11	6
10	2
20	4
21	12
22	36
32	72
31	24
30	8

Clearly each divisor follows from the previous one by one multiplication or division by a prime number.

Even simpler: A binary Gray code can be used to iterate over the subsets of a set in such a way that in each step only one member is added or removed.

Chapter 14: Cyclic Redundancy Check

1. From the text, a message polynomial M and generator polynomial G satisfy $Mx^r = QG + R$, where R is the checksum polynomial. Let M' be a message polynomial that differs from M at term x^e. (That is, the binary message differs at bit position e.) Then $M' = M + x^e$, and

$$M'x^r = (M+x^e)x^r = Mx^r + x^{e+r} = QG+R+x^{e+r}$$

The term $x^e + r$ is not divisible by G, because G has two or more terms. (The only divisors of $x^e + r$ are of the form *x>>.)* Therefore, the remainder upon dividing $M'x^r$ by G is distinct from R, so the error is detected.

2. The main loop might be coded as shown below, where `word` is an `unsigned int` [Danne].

```
crc = 0xFFFFFFFF;
while (((word = *(unsigned int *)message) & 0xFF) != 0) {
    crc = crc ^ word;
    crc = (crc >> 8) ^ table[crc & 0xFF];
    crc = (crc >> 8) ^ table[crc & 0xFF];
    crc = (crc >> 8) ^ table[crc & 0xFF];
    crc = (crc >> 8) ^ table[crc & 0xFF];
    message = message + 4;
}
```

Compared to the code of Figure 14–7 on page 329, this saves three *load byte* and three *exclusive or* instructions for each word of `message`. And, there are fewer loop control instructions executed.

Chapter 15: Error-Correcting Codes

1. Your table should look like Table 15–1 on page 333, with the rightmost column and the odd numbered rows deleted.

2. In the first case, if an error occurs in a check bit, the receiver cannot know that, and it will make an erroneous "correction" to the information bits.

In the second case, if an error occurs in a check bit, the syndrome will be one of 100...0, 010...0, 001...0, ..., 000...1 (k distinct values). Therefore k must be large enough to encode these k values, as well as the m values to encode a single error in one of the m information bits, and a value for "no errors." So the Hamming rule stands.

One thing along these lines that could be done is to have a single parity bit for the k check bits, and have the k check bits encode values that designate one error in an information bit (and where it is), or no errors occurred. For this code, k could be chosen as the smallest value for which $2^k \geq m + 1$. The code length would be $m + k + 1$, where the "+1" is for the parity bit on the check bits. But this code length is nowhere better than that given by the Hamming rule, and is sometimes worse.

3. Treating k and m as real numbers, the following iteration converges from below quite rapidly:

$$k_0 = 0,$$
$$k_{i+1} = \lg(k_i + m + 1), \quad i = 0, 1, \ldots,$$

where $\lg(x)$ is the log base 2 of x. The correct result is given by $\text{ceil}(k_2)$ is, only two iterations are required for all $m \geq 0$.

Taking another tack, it is not difficult to prove that for $m \geq 0$,

$$\text{bitsize}(m) \leq k \leq \text{bitsize}(m) + 1.$$

Here bitsize(m) is the size of m in bits, for example, bitsize(3) = 2, bitsize(4) =

3, and so forth. (This is different from the function of the same name described in Section 5–3 on page 99, which is for signed integers.) *Hint:* bitsize(m) = lg(m + 1) = $\lfloor$lg(m) + 1$\rfloor$, where we take lg(0) to be –1. Thus, one can try k = bitsize(m), test it, and if it proves to be too small then simply add 1 to the trial value. Using the *number of leading zeros* function to compute bitsize(m), one way to commit this to code is:

$$k \leftarrow W - \mathrm{nlz}(m),$$
$$k \leftarrow k + (((1 \ll k) - 1 - k) \overset{u}{<} m),$$

where W is the machine's word size and $0 \le m \le 2^W - 1$.

4. Answer: If $d(x,z) > d(x,y) + d(y,z)$, it must be that for at least one bit position i, that bit position contributes 1 to $d(x,z)$ and 0 to $d(x,y) + d(y,z)$. This implies that $x_i \neq z_i$, but $x_i = y_i$ and $y_i = z_i$, clearly a contradiction.

5. Given a code of length n and minimum distance d, simply double-up each 1 and each 0 in each code word. The resulting code is of length $2n$, minimum distance $2d$, and is the same size.

6. Given a code of length n, minimum distance d, and size $A(n, d)$, think of it as being displayed as in Table 15–1 on page 333. Remove an arbitrary d- 1 columns. The resulting code words, of length n-(d-1), have a minimum distance of at least 1. That is, they are all distinct. Hence their number cannot be more than 2”–$^{(d-1)}$. Since deleting columns did not change the code size, the original code's size is at most $2^{n(d-1)}$, so that $A(n,d) \le 2^{n-d+1}$.

7. The Hamming rule applies to the case that $d = 3$ and the code has 2^m code words, where m is the number of information bits. The right-hand part of inequality (6), with $A(n, d) = 2^m$ and $d = 3$, is

$$2^m \le \frac{2^n}{\binom{n}{0} + \binom{n}{1}} = \frac{2^n}{1+n}.$$

Replacing n with $m + k$ gives

$$2^m \le \frac{2^{m+k}}{1+m+k},$$

which on cancelling 2^m on each side becomes inequality (1).

8. The code must consist of an arbitrary bit string and its one's-complement, so its size is 2. That these codes are perfect, for odd n, can be seen by showing that they achieve the upper bound in inequality (6). Proof sketch: An n -bit binary integer may be thought of as representing uniquely a choice from n objects, with a 1-bit meaning to choose and a 0-bit meaning not to choose the corresponding object. Therefore, there are 2^n ways to choose from 0 to n objects from n objects—that is, $\sum_{i=0}^{n} \binom{n}{i} = 2^n$. If n is odd, i ranging from 0 to $(n - 1)/2$ covers half the terms of this sum, and because of the symmetry

$\binom{n}{i} = \binom{n}{n-i}$, it accounts for half the sum. Therefore $\sum_{i=0}^{(n-1)/2} \binom{n}{i} = 2^{n-1}$, so that the upper bound in (6) is 2. Thus, the code achieves the upper bound of (6).

9. For ease of exposition, this proof will make use of the notion of *equivalence* of codes. Clearly a code is not changed in any substantial way by rearranging its columns (as depicted in Table 15–1 on page 333) or by complementing any column. If one code can be derived from another by such transformations, they are said to be *equivalent*. Because a code is an unordered set of code words, the order of a display of its code words is immaterial. By complementing columns, any code can be transformed into an equivalent code that has a code word that is all 0's.

Also for ease of exposition, we illustrate this proof by using the case n = 9 and d = 6.

Wlog (without loss of generality), let code word 0 (the first, which we will call cw_0) be 000 000 000. Then all other code words must have at least six 1's, to differ from cw_0 in at least six places.

Assume (which will be shown) that the code has at least three code words. Then no code word can have seven or more 1's. For if one did, then another code word (which necessarily has six or more 1's) would have at least four of its 1's in the same columns as the word with seven or more 1's. This means the code words would be equal in four or more positions, so they could differ in five or fewer positions (9 – 4), violating the requirement that d = 6. Therefore, all code words other than the first must have exactly six 1's.

Wlog, rearrange the columns so that the first two code words are

cw_0: 000 000 000

cw_1: 111 111 000

The next code word, cw_2, cannot have four or more of its 1's in the left six columns, because then it would be the same as cw_1 in four or more positions, so it would differ from cw_1 in five or fewer positions. Therefore it has three or fewer of its 1's in the left six columns, so that three of its 1's must be in the right three positions. Therefore exactly three of its 1's are in the left six columns. Rearrange the left six columns (of all three code words) so that cw_2 looks like this:

cw_2: 111 000 111

By similar reasoning, the next code word (cw_3) cannot have four of its 1's in the left three and right three positions together, because it would then equal cw_2 in four positions. Therefore it has three fewer 1's in the left three and right three positions, so that three of its 1's must be in the middle three positions. By similarly comparing it to cw_1, we conclude that three of its 1's must be in the right three positions. Therefore cw_3 is:

cw_3: 000 111 111

By comparing the next code word, if one is possible, with cw_1, we conclude that it must have three 1's in the right three positions. By comparing it with cw_2, we conclude it must have three 1's in the middle three positions.

Thus, the code word is 000 111 111, which is the same as cw_3. Therefore a fifth code word is impossible. By inspection, the above four code words satisfy $d = 6$, so $A(9, 6) = 4$.

10. Obviously $A(n, d)$ is at least 2, because the two code words can be all 0's and all 1's. Reasoning as in the previous exercise, let one code word, cw_0, be all 0's. Then all other code words must have more than $2n/3$ 1's. If the code has three or more code words, then any two code words other than cw_0 must have 1's in the same positions for more than $2n/3 - n/3 = n/3$ positions, as suggested by the figure below.

$$\begin{array}{cc} 1111\ldots1111 & 0\ldots0 \\ > 2n/3 & < n/3 \end{array}$$

(The figure represents cw_1 with its 1's pushed to the left. Imagine placing the more than $2n/3$ 1's of cw_2 to minimize the overlap of the 1's.) Since cw_1 and cw_2 overlap in more than $n/3$ positions, they can differ in less than $n - n/3 = 2n/3$ positions, resulting in a minimum distance less than $2n/3$.

11. It is SEC-DED, because the minimum distance between code words is 4. To see this, assume first that two code words differ in a single information bit. Then in addition to the information bit, the row parity, column parity, and corner check bits will be different in the two code words, making their distance equal to 4. If the information words differ in two bits, and they are in the same row, then the row parity bit will be the same in the two code words, but the column parity bit will differ in two columns. Hence their distance is 4. The same result follows if they are in the same column. If the two differing information bits are in different rows and columns, then the distance between the code words is 6. Lastly, if the information words differ in three bits, it is easy to verify that no matter what their distribution among the rows and columns, at least one parity bit will differ. Hence the distance is at least 4.

If the corner bit is not used, the minimum distance is 3. Therefore it is not SEC-DED, but it is a SEC code.

Whether the corner check bit is a row sum or a column sum, it is the modulo 2 sum of all 64 information bits, so it has the same value in either case.

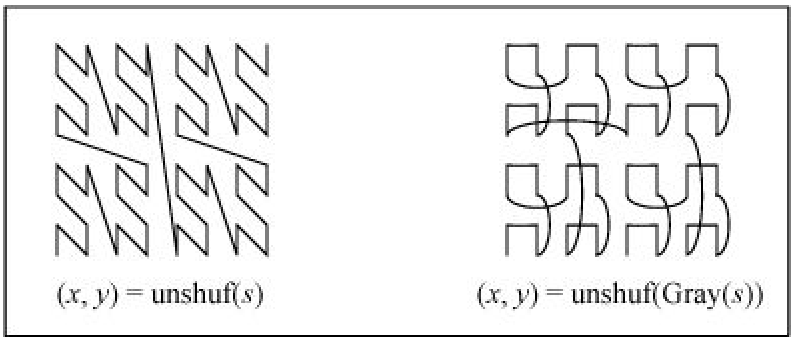
The code requires 17 check bits, whereas the Hamming code requires eight (see Table 15–3 on page 336), so it is not very efficient in that respect.

But it *is* effective in detecting burst errors. Assume the 9×9 array is transmitted over a bit serial channel in the order row 0, row 1,..., row 8. Then any sequence of ten or fewer bits is in one or two rows with at most one bit of overlap. Hence if the only errors in a transmission are a subset of ten consecutive bits, the error will be detected by checking the column parities in most cases, or the row parity bits in the case that the first and tenth bits only are in error.

An error that is *not* detected is four corrupted bits arranged in a rectangle.

Chapter 16: Hilbert's Curve

1. and **2**.



The average jump distance for the traversal shown at the left above is approximately 1.46. That for the traversal shown at the right is approximately 1.33. Therefore, using the Gray code seems to improve locality, at least by this measure. (For the Hilbert curve, the jumps are all of distance 1.)

At Edsger Dijkstra's suggestion, the shuffle algorithm was used in an early Algol compiler to map a matrix onto backing store. The aim was to reduce paging operations when inverting a matrix. He called it the "zip-fastener algorithm." It seems likely that many people have discovered it independently.

3. Use every third bit of *s*.

Chapter 17: Floating-Point

1. ±0, ±2.0, and certain NaNs.

2. Yes! The program is easily derived by noting that if $x = 2^n(1+f)$, then

$$\sqrt{x} = 2^{n/2}(1+f)^{1/2}.$$

Ignoring the fraction, this shows that we must change the biased exponent from 127 + *n* to 127 + *n* /2. The latter is $(127 + n)/2 + 127/2$. Thus, it seems that a rough approximation to $\sqrt{x}$ is obtained by shifting rep(*x*) right one position and adding 63 in the exponent position, which is 0x1F800000. This approximation,

$$\text{rep}(\sqrt{x}) \approx k + (\text{rep}(x) \overset{s}{\gg} 1),$$

also has the property that if we find an optimal value of *k* for values of *x* in the range 1.0 to 4.0, then the same value of *k* is optimal for all normal numbers. After refining the value of *k* with the aid of a program that finds the maximum and minimum error for a given value of *k*, we obtain the program shown below. It includes one step of Newton-Raphson iteration.

```
float asqrt(float x0) {
   union {int ix; float x;};

   x = x0;                       // x can be viewed as int.
   ix = 0x1fbb67a8 + (ix >> 1); // Initial guess.
   x = 0.5f*(x + x0/x);         // Newton step.
   return x;
}
```

For normal numbers, the relative error ranges from 0 to approximately 0.000601. It gets the correct result for x = inf and x = NaN (inf and NaN, respectively). For $x = 0$ the result is approximately 4.0×10^{-20}. For $x = -0$, the result is the rather useless -1.35×10^{19}. For x a positive denorm, the result is either within the stated tolerance or is a positive number less than 10^{-19}.

The Newton step uses division, so on most machines the program is not as fast as that for the reciprocal square root.

If a second Newton step is added, the relative error for normal numbers ranges from 0 to approximately 0.00000023. The optimal constant is 0x1FBB3F80. If no Newton step is included, the relative error is slightly less than ±0.035, using a constant of 0x1FBB4F2E. This is about the same as the relative error of the reciprocal square root routine without a Newton step, and like it, uses only two integer operations.

3. Yes, one can do cube roots of positive normal numbers with basically the same method. The key statement is the first approximation:

```
i = 0x2a51067f + i/3;        // Initial guess.
```

This computes the cube root with a relative error of approximately ±0.0316. The division by 3 can be approximated with

$$\frac{i}{3} \approx \frac{i}{4} + \frac{i}{16} + \frac{i}{64} + \ldots + \frac{i}{65536}$$

(where the divisions by powers of 2 are implemented as right shifts). This can be evaluated with seven instructions and slightly improved accuracy as shown in the program below. (This division trick is discussed in Section 10–18 on page 251.)

```
float acbrt(float x0) {
   union {int ix; float x;};

   x = x0;                      // x can be viewed as int.
   ix = ix/4 + ix/16;           // Approximate divide by 3.
   ix = ix + ix/16;
   ix = ix + ix/256;
   ix = 0x2a5137a0 + ix;        // Initial guess.
   x = 0.33333333f*(2.0f*x + x0/(x*x));  // Newton step.
   return x;
}
```

Although we avoided the division by 3 (at a cost of seven elementary integer instructions), there is a division and four other instructions in the Newton step. The relative error ranges from 0 to approximately +0.00103. Thus, the method is not as successful as in the case of reciprocal square root and square root, but it might be useful in some situations.

If the Newton step is repeated and the same constant is used, the relative error ranges from 0 to approximately +0.00000116.

4. Yes. The program below computes the reciprocal square root of a double-precision floating-point number with an accuracy of about ±3.5%. It is straightforward to improve its accuracy with one or two steps of Newton-

Raphson iteration. Using the constant 0x5fe80...0 gives a relative error in the range 0 to approximately +0.887, and the constant 0x5fe618fdf80...0 gives a relative error in the range 0 to approximately –0.0613.

```
double rsqrtd(double x0) {
    union {long long ix; double x;};

    x = x0;
    ix = 0x5fe6ec85e8000000LL - (ix >> 1);
    return x;
}
```

Chapter 18: Formulas for Primes

1. Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a^0$. Such a polynomial monotonically approaches infinity, in magnitude, as x approaches infinity. (For sufficiently large x, the first term exceeds in magnitude the sum of the others.)

Let x_0 be an integer such that $|f(x)| \geq 2$ for all $x > x_0$. Let $f(x_0) = k$, and let r be any positive integer. Then $|k| \geq 2$, and

$$\begin{aligned} |f(x_0 + rk)| &= |a_n(x_0 + rk)^n + a_{n-1}(x_0 + rk)^{n-1} + ... + a_0| \\ &= |f(x_0) + \text{a multiple of } rk| \\ &= |k + \text{a multiple of } rk|. \end{aligned}$$

Thus, as r increases, $|f(x_0 + rk)|$ ranges over composites that increase in magnitude, and hence are distinct. Therefore $f(x)$ takes on an infinite number of composite values.

Another way to state the theorem is that there is no non-constant polynomial in one variable that takes on only prime numbers, even for sufficiently large values of its argument.

Example: Let $f(x) = x^2 + x + 41$. Then $f(1) = 43$ and

$$\begin{aligned} f(1 + 43r) &= (1 + 43r)^2 + (1 + 43r) + 41 \\ &= (1 + 86r + 43^2 r^2) + (1 + 43r) + 41 \\ &= 1 + 1 + 41 + 86r + 43^2 r^2 + 43r \\ &= 43 + (2 + 43r + 1) \cdot 43r \end{aligned}$$

which clearly produces ever-increasing multiples of 43 as r increases.

2. Suppose p is composite. Write the congruence as

$$(p - 1)! = pk - 1,$$

for some integer k. Let a be a proper factor of p. Then a divides the left side, but not the right side, so equality cannot hold.

The theorem is easily seen to be true for $p = 1, 2,$ and 3. Suppose p is a prime greater than 3. Then in the factorial

$$(p-1)! = (p-1)(p-2)\ldots(3)(2),$$

the first term, $p - 1$, is congruent to –1 modulo p. Each of the other terms is relatively prime to p and therefore has a multiplicative inverse modulo p (see Section 10–16 on page 240), and furthermore, the inverse is unique and not equal to itself.

To see that the multiplicative inverse modulo a prime is not equal to itself (except for 1 and $p - 1$), suppose $a^2 \equiv 1 \pmod{p}$. Then $a^2 - 1 \equiv 0 \pmod{p}$, so that $(a - 1)(a + 1) \equiv 0 \pmod{p}$. Because p is a prime, either $a - 1$ or $a + 1$ is congruent to 0 modulo p. In the former case $a \equiv 1 \pmod{p}$ and in the latter case $a \equiv -1 \equiv p - 1 \pmod{p}$.

Therefore, the integers $p - 2$, $p - 3$, ..., 2 can be paired so that the product of each pair is congruent to 1 modulo p. That is,

$$(p-1)! = (p-1)(ab)(cd)\ldots,$$

where a and b are multiplicative inverses, as are c and d, and so forth. Thus

$$(p-1)! = (-1)(1)(1) \equiv -1 \pmod{p}.$$

Example, $p = 11$: 10! (mod 11)= 10 • 9 • 8 • 7 • 6 • 5 • 4 • 3 • 2 (mod 11) = 10-(9-5)(8-7)(6-2)(4-3) (mod 11) = (-1)(1)(1)(1)(1) (mod 11) = -1 (mod 11).

The theorem is named for John Wilson, a student of the English mathematician Edward Waring. Waring announced it without proof in 1770. The first published proof was by Lagrange in 1773. The theorem was known in medieval Europe around 1000 AD.

3. If $n = ab$, with a and b distinct and neither equal to 1 or n, then clearly a and b are less than n and hence are terms of $(n - 1)!$. Therefore n divides $(n - 1)!$.

If $n = a^2$, then for $a > 2$, $a^2 = n > 2a$, so that both a and $2a$ are terms of $(n - 1)!$. Therefore a^2 divides $(n - 1)!$.

4. This is probably a case in which a calculation gives more insight into a mathematical truth than does a formal proof.

According to Mills's theorem, there exists a real number θ such that $\lfloor \theta^{3^n} \rfloor$ is prime for all integers $n \geq 1$. Let us try the possibility that for $n = 1$, the prime is 2. Then

$$\lfloor \theta^{3^1} \rfloor = 2,$$

so that

$$2 \le \theta^{3^1} < 3, \text{ or} \qquad (1)$$

$$2^{1/3} \le \theta < 3^{1/3}, \text{ or}$$

$$1.2599\ldots \le \theta < 1.4422\ldots.$$

Cubing inequality (1) gives

$$8 \le \theta^{3^2} < 27. \tag{2}$$

There is a prime in this range. (From our assumption, there is a prime between 2^3 and $(2 + 1)^3$.) Let us choose 11 for the second prime. Then, we will have $\lfloor \theta^{3^2} \rfloor = 11$ if we further constrain (2) to

$$11 \le \theta^{3^2} < 12. \tag{3}$$

Continuing, we cube (3), giving

$$1331 \le \theta^{3^3} < 1728. \tag{4}$$

We are assured that there is a prime between 1331 and 1728. Let us choose the smallest one, 1361. Further constraining (4),

$$1361 \le \theta^{3^3} < 1362.$$

So far, we have shown that there exists a real number theta such that $\lfloor \theta^{3^n} \rfloor$ is prime for n = 1, 2, and 3 and, by taking 27th roots of 1361 and 1362, that θ is between 1.30637 and 1.30642.

Obviously the process can be continued. It can be shown that a limiting value of θ exists, but that is not really necessary. If, in the limit, θ is an arbitrary number in some finite range, that still verifies Mills's theorem.

The above calculation shows that Mills's theorem is a little contrived. As far as its being a formula for primes, you have to know the primes to determine θ. It is like the formula for primes involving the constant

$$a = 0.203005000700011000013\ldots,$$

given on page 392. The theorem clearly has little to do with primes. A similar theorem holds for any increasing sequence provided it is sufficiently dense.

The steps above calculate the smallest theta that satisfies Mills's theorem. It is sometimes called Mills' constant, and it has been calculated to over 6850 decimal places [CC].

5. Suppose that there exist integers *a, b, c*, and *d* such that

$$(a + b\sqrt{-5})(c + d\sqrt{-5}) = 2. \tag{5}$$

Equating real and imaginary parts,

$$ac - 5bd = 2, \text{ and} \tag{6}$$

$$ad + bc = 0. \tag{7}$$

Clearly $c \ne 0$, because if $c = 0$ then from (6), $-5bd = 2$, which has no solution in integers.

Also $b \ne 0$, because if $b = 0$, then from (7), either a or d is 0. $a = 0$ does not satisfy (5). Therefore $d = 0$. Then (5) becomes $ac = 2$, so one of the factors in (5) is a unit, which is not an acceptable decomposition.

From (7), $abd + b^2c = 0$. From (6), $a^2c - 5\,abd = 2a$. Combining, $a^2c +$

$5b^2c = 2a$, or

$$a^2 + 5b^2 = 2a/c \qquad (8)$$

(recall that $c \neq 0$). The left side of (8) is at least $a^2 + 5$, which exceeds $2a/c$ whatever the values of a and c are.

To see that 3 is prime, the equation

$$a^2 + 5b^2 = 3a/c$$

can be similarly derived, with $b \neq 0$ and $c \neq 0$. This also cannot be satisfied in integers.

The number 6 has two distinct decompositions into primes:

$$6 = 2 \cdot 3 = (1 + \sqrt{-5})(1 - \sqrt{-5}).$$

We have not shown that $1 \pm \sqrt{-5}$ are primes. This can be shown by arguments similar to those given above (although somewhat longer), but it is not really necessary to do so to demonstrate that prime factorization is not unique in this ring. This is because however each of these numbers might factor into primes, the total decomposition will not be 2.3.

Appendix A. Arithmetic Tables for A 4-Bit Machine

In the tables in Appendix A, underlining denotes signed overflow. For example, in Table A–1, 7 + 1 = 8, which is not representable as a signed integer on a 4-bit machine, so signed overflow occurred.

TABLE A–1. ADDITION

										−8	−7	−6	−5	−4	−3	−2	−1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	1	1	2	3	4	5	6	7	$\underline{8}$	9	A	B	C	D	E	F	10
	2	2	3	4	5	6	7	$\underline{8}$	$\underline{9}$	A	B	C	D	E	F	10	11
	3	3	4	5	6	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	B	C	D	E	F	10	11	12
	4	4	5	6	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	$\underline{B}$	C	D	E	F	10	11	12	13
	5	5	6	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	$\underline{B}$	$\underline{C}$	D	E	F	10	11	12	13	14
	6	6	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	$\underline{B}$	$\underline{C}$	$\underline{D}$	E	F	10	11	12	13	14	15
	7	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	$\underline{B}$	$\underline{C}$	$\underline{D}$	$\underline{E}$	F	10	11	12	13	14	15	16
−8	8	8	9	A	B	C	D	E	F	$\underline{10}$	$\underline{11}$	$\underline{12}$	$\underline{13}$	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$
−7	9	9	A	B	C	D	E	F	10	$\underline{11}$	$\underline{12}$	$\underline{13}$	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$	18
−6	A	A	B	C	D	E	F	10	11	$\underline{12}$	$\underline{13}$	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$	18	19
−5	B	B	C	D	E	F	10	11	12	$\underline{13}$	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$	18	19	1A
−4	C	C	D	E	F	10	11	12	13	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$	18	19	1A	1B
−3	D	D	E	F	10	11	12	13	14	$\underline{15}$	$\underline{16}$	$\underline{17}$	18	19	1A	1B	1C
−2	E	E	F	10	11	12	13	14	15	$\underline{16}$	$\underline{17}$	18	19	1A	1B	1C	1D
−1	F	F	10	11	12	13	14	15	16	$\underline{17}$	18	19	1A	1B	1C	1D	1E

The table for subtraction (Table A–2) assumes that the carry bit for ***a* − *b*** is set as it would be for $a + \bar{b} + 1$, so that carry is equivalent to "not borrow."

TABLE A–2. SUBTRACTION (ROW − COLUMN)

										−8	−7	−6	−5	−4	−3	−2	−1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	10	F	E	D	C	B	A	9	$\underline{8}$	7	6	5	4	3	2	1
	1	11	10	F	E	D	C	B	A	$\underline{9}$	$\underline{8}$	7	6	5	4	3	2
	2	12	11	10	F	E	D	C	B	$\underline{A}$	$\underline{9}$	$\underline{8}$	7	6	5	4	3
	3	13	12	11	10	F	E	D	C	$\underline{B}$	$\underline{A}$	$\underline{9}$	$\underline{8}$	7	6	5	4
	4	14	13	12	11	10	F	E	D	$\underline{C}$	$\underline{B}$	$\underline{A}$	$\underline{9}$	$\underline{8}$	7	6	5
	5	15	14	13	12	11	10	F	E	$\underline{D}$	$\underline{C}$	$\underline{B}$	$\underline{A}$	$\underline{9}$	$\underline{8}$	7	6
	6	16	15	14	13	12	11	10	F	$\underline{E}$	$\underline{D}$	$\underline{C}$	$\underline{B}$	$\underline{A}$	$\underline{9}$	$\underline{8}$	7
	7	17	16	15	14	13	12	11	10	$\underline{F}$	$\underline{E}$	$\underline{D}$	$\underline{C}$	$\underline{B}$	$\underline{A}$	$\underline{9}$	$\underline{8}$
−8	8	18	$\underline{17}$	$\underline{16}$	$\underline{15}$	$\underline{14}$	$\underline{13}$	$\underline{12}$	$\underline{11}$	10	F	E	D	C	B	A	9
−7	9	19	18	$\underline{17}$	$\underline{16}$	$\underline{15}$	$\underline{14}$	$\underline{13}$	$\underline{12}$	11	10	F	E	D	C	B	A
−6	A	1A	19	18	$\underline{17}$	$\underline{16}$	$\underline{15}$	$\underline{14}$	$\underline{13}$	12	11	10	F	E	D	C	B
−5	B	1B	1A	19	18	$\underline{17}$	$\underline{16}$	$\underline{15}$	$\underline{14}$	13	12	11	10	F	E	D	C
−4	C	1C	1B	1A	19	18	$\underline{17}$	$\underline{16}$	$\underline{15}$	14	13	12	11	10	F	E	D
−3	D	1D	1C	1B	1A	19	18	$\underline{17}$	$\underline{16}$	15	14	13	12	11	10	F	E
−2	E	1E	1D	1C	1B	1A	19	18	$\underline{17}$	16	15	14	13	12	11	10	F
−1	F	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10

For multiplication (Tables A–3 and A–4), overflow means that the result cannot be expressed as a 4-bit quantity. For signed multiplication (Table A–3), this is equivalent to the first five bits of the 8-bit result not being all 1’s or all 0’s.

TABLE A–3. SIGNED MULTIPLICATION

										−8	−7	−6	−5	−4	−3	−2	−1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	2	3	4	5	6	7	F8	F9	FA	FB	FC	FD	FE	FF
	2	0	2	4	6	8	A	C	E	F0	F2	F4	F6	F8	FA	FC	FE
	3	0	3	6	9	C	F	12	15	E8	EB	EE	F1	F4	F7	FA	FD
	4	0	4	8	C	10	14	18	1C	E0	E4	E8	EC	F0	F4	F8	FC
	5	0	5	A	F	14	19	1E	23	D8	DD	E2	E7	EC	F1	F6	FB
	6	0	6	C	12	18	1E	24	2A	D0	D6	DC	E2	E8	EE	F4	FA
	7	0	7	E	15	1C	23	2A	31	C8	CF	D6	DD	E4	EB	F2	F9
−8	8	0	F8	F0	E8	E0	D8	D0	C8	40	38	30	28	20	18	10	8
−7	9	0	F9	F2	EB	E4	DD	D6	CF	38	31	2A	23	1C	15	E	7
−6	A	0	FA	F4	EE	E8	E2	DC	D6	30	2A	24	1E	18	12	C	6
−5	B	0	FB	F6	F1	EC	E7	E2	DD	28	23	1E	19	14	F	A	5
−4	C	0	FC	F8	F4	F0	EC	E8	E4	20	1C	18	14	10	C	8	4
−3	D	0	FD	FA	F7	F4	F1	EE	EB	18	15	12	F	C	9	6	3
−2	E	0	FE	FC	FA	F8	F6	F4	F2	10	E	C	A	8	6	4	2
−1	F	0	FF	FE	FD	FC	FB	FA	F9	8	7	6	5	4	3	2	1

TABLE A–4. UNSIGNED MULTIPLICATION

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Tables A–5 and A–6 are for conventional truncating division. Table A–5 shows a result of 8 with overflow for the case of the maximum negative number divided by –1, but on most machines the result in this case is undefined, or the operation is suppressed.

TABLE A–5. SIGNED SHORT DIVISION (ROW ÷ COLUMN)

										−8	−7	−6	−5	−4	−3	−2	−1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	–	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	–	1	0	0	0	0	0	0	0	0	0	0	0	0	0	F
	2	–	2	1	0	0	0	0	0	0	0	0	0	0	0	F	E
	3	–	3	1	1	0	0	0	0	0	0	0	0	0	F	F	D
	4	–	4	2	1	1	0	0	0	0	0	0	0	F	F	E	C
	5	–	5	2	1	1	1	0	0	0	0	0	F	F	F	E	B
	6	–	6	3	2	1	1	1	0	0	0	F	F	F	E	D	A
	7	–	7	3	2	1	1	1	1	0	F	F	F	F	E	D	9
−8	8	–	8	C	E	E	F	F	F	1	1	1	1	2	2	4	8
−7	9	–	9	D	E	F	F	F	F	0	1	1	1	1	2	3	7
−6	A	–	A	D	E	F	F	F	0	0	0	1	1	1	2	3	6
−5	B	–	B	E	F	F	F	0	0	0	0	0	1	1	1	2	5
−4	C	–	C	E	F	F	0	0	0	0	0	0	0	1	1	2	4
−3	D	–	D	F	F	0	0	0	0	0	0	0	0	0	1	1	3
−2	E	–	E	F	0	0	0	0	0	0	0	0	0	0	0	1	2
−1	F	–	F	0	0	0	0	0	0	0	0	0	0	0	0	0	1

TABLE A–6. UNSIGNED SHORT DIVISION (ROW ÷ COLUMN)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	–	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	–	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	–	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	–	3	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4	–	4	2	1	1	0	0	0	0	0	0	0	0	0	0	0
5	–	5	2	1	1	1	0	0	0	0	0	0	0	0	0	0
6	–	6	3	2	1	1	1	0	0	0	0	0	0	0	0	0
7	–	7	3	2	1	1	1	1	0	0	0	0	0	0	0	0
8	–	8	4	2	2	1	1	1	1	0	0	0	0	0	0	0
9	–	9	4	3	2	1	1	1	1	1	0	0	0	0	0	0
A	–	A	5	3	2	2	1	1	1	1	1	0	0	0	0	0
B	–	B	5	3	2	2	1	1	1	1	1	1	0	0	0	0
C	–	C	6	4	3	2	2	1	1	1	1	1	1	0	0	0
D	–	D	6	4	3	2	2	1	1	1	1	1	1	1	0	0
E	–	E	7	4	3	2	2	2	1	1	1	1	1	1	1	0
F	–	F	7	5	3	3	2	2	1	1	1	1	1	1	1	1

Tables A–7 and A–8 give the remainder associated with conventional truncating division. Table A–7 shows a result of 0 for the case of the maximum negative number divided by –1, but on most machines the result for this case is undefined, or the operation is suppressed.

TABLE A–7. REMAINDER FOR SIGNED SHORT DIVISION (ROW ÷ COLUMN)

										-8	-7	-6	-5	-4	-3	-2	-1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	–	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	–	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
	2	–	0	0	2	2	2	2	2	2	2	2	2	2	2	0	0
	3	–	0	1	0	3	3	3	3	3	3	3	3	3	0	1	0
	4	–	0	0	1	0	4	4	4	4	4	4	4	0	1	0	0
	5	–	0	1	2	1	0	5	5	5	5	5	0	1	2	1	0
	6	–	0	0	0	2	1	0	6	6	6	0	1	2	0	0	0
	7	–	0	1	1	3	2	1	0	7	0	1	2	3	1	1	0
-8	8	–	0	0	E	0	D	E	F	0	F	E	D	0	E	0	0
-7	9	–	0	F	F	D	E	F	0	9	0	F	E	D	F	F	0
-6	A	–	0	0	0	E	F	0	A	A	A	0	F	E	0	0	0
-5	B	–	0	F	E	F	0	B	B	B	B	B	0	F	E	F	0
-4	C	–	0	0	F	0	C	C	C	C	C	C	C	0	F	0	0
-3	D	–	0	F	0	D	D	D	D	D	D	D	D	D	0	F	0
-2	E	–	0	0	E	E	E	E	E	E	E	E	E	E	E	0	0
-1	F	–	0	F	F	F	F	F	F	F	F	F	F	F	F	F	0

TABLE A–8. REMAINDER FOR UNSIGNED SHORT DIVISION (ROW ÷ COLUMN)

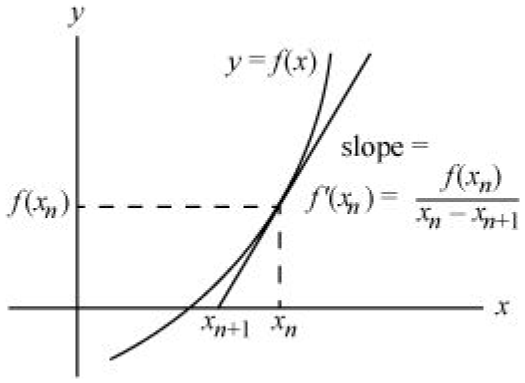
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	–	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	–	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	–	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2
3	–	0	1	0	3	3	3	3	3	3	3	3	3	3	3	3
4	–	0	0	1	0	4	4	4	4	4	4	4	4	4	4	4
5	–	0	1	2	1	0	5	5	5	5	5	5	5	5	5	5
6	–	0	0	0	2	1	0	6	6	6	6	6	6	6	6	6
7	–	0	1	1	3	2	1	0	7	7	7	7	7	7	7	7
8	–	0	0	2	0	3	2	1	0	8	8	8	8	8	8	8
9	–	0	1	0	1	4	3	2	1	0	9	9	9	9	9	9
A	–	0	0	1	2	0	4	3	2	1	0	A	A	A	A	A
B	–	0	1	2	3	1	5	4	3	2	1	0	B	B	B	B
C	–	0	0	0	0	2	0	5	4	3	2	1	0	C	C	C
D	–	0	1	1	1	3	1	6	5	4	3	2	1	0	D	D
E	–	0	0	2	2	4	2	0	6	5	4	3	2	1	0	E
F	–	0	1	0	3	0	3	1	7	6	5	4	3	2	1	0

Appendix B. Newton's Method

To review Newton's method very briefly, we are given a differentiable function f of a real variable x and we wish to solve the equation $f(x) = 0$ for x. Given a current estimate x_n of a root of f, Newton's method gives us a better estimate x_{n+1} under suitable conditions, according to the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Here, $f'(x_n)$ is the derivative of f at $x = x_n$. The derivation of this formula can be read off the figure below (solve for x_{n+1}).



The method works very well for simple, well-behaved functions such as polynomials, provided the first estimate is quite close. Once an estimate is sufficiently close, the method converges quadratically. That is, if r is the exact value of the root, and x_n is a sufficiently close estimate, then

$$|x_{n+1} - r| \le (x_n - r)^2.$$

Thus, the number of digits of accuracy doubles with each iteration (e.g., if $|x_n - r| \le 0.001$ then $|x_{n+1} - r| \le 0.000001$).

If the first estimate is way off, then the iterations may converge very slowly, may diverge to infinity, may converge to a root other than the one closest to the first estimate, or may loop among certain values indefinitely.

This discussion has been quite vague because of phrases like "suitable conditions," "well-behaved," and "sufficiently close." For a more precise discussion, consult almost any first-year calculus textbook.

In spite of the caveats surrounding this method, it is occasionally useful in the domain of integers. To see whether or not the method applies to a particular function, you have to work it out, such as is done in Section 11–1, "Integer Square Root," on page 279.

Table B–1 gives a few iterative formulas derived from Newton's method, for computing certain numbers. The first column shows the number it is desired to compute. The second column shows a function that has that number as a root. The third column shows the right-hand side of Newton's formula corresponding to that function.

TABLE B–1. NEWTON'S METHOD FOR COMPUTING CERTAIN NUMBERS

Quantity to Be Computed	Function	Iterative Formula
$\sqrt{a}$	$x^2 - a$	$\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$
$\sqrt[3]{a}$	$x^3 - a$	$\frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right)$
$\frac{1}{\sqrt{a}}$	$x^{-2} - a$	$\frac{x_n}{2}(3 - ax_n^2)$
$\frac{1}{a}$	$x^{-1} - a$	$x_n(2 - ax_n)$
$\log_2 a$	$2^x - a$	$x_n + \frac{1}{\ln 2}\left(\frac{a}{2^{x_n}} - 1\right)$

It is not always easy, incidentally, to find a good function to use. There are, of course, many functions that have the desired quantity as a root, and only a few of them lead to a useful iterative formula. Usually, the function to use is a sort of inverse of the desired computation. For example, to find $\sqrt{a}$ use $f(x) = x^2 - a$; to find $\log_2 a$ use $f(x) = 2^x - a$, and so on.[1]

The iterative formula for $\log_2 a$ converges (to $\log_2 a$) even if the multiplier $1/\ln 2$ is altered somewhat (for example, to 1, or to 2). However, it then converges more slowly. A value of 3/2 or 23/16 might be useful in some applications ($1/\ln 2 \approx 1.4427$).

Appendix C. A Gallery of Graphs of Discrete Functions

This appendix shows plots of a number of discrete functions. They were produced by Mathematica. For each function, two plots are shown: one for a word size of three bits and the other for a word size of five bits. This material was suggested by Guy Steele.

C–1 Plots of Logical Operations on Integers

This section includes 3D plots of and(*x, y*), or(*x, y*), and xor(*x, y*) as functions of integers *x* and *y*, in Figures C–1, C–2, and C–3, respectively.

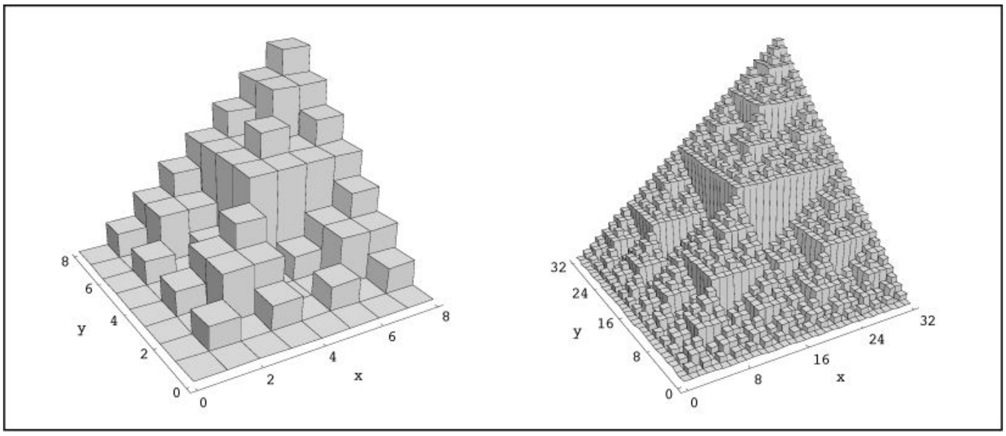


FIGURE C–1. Plots of the logical *and* function.

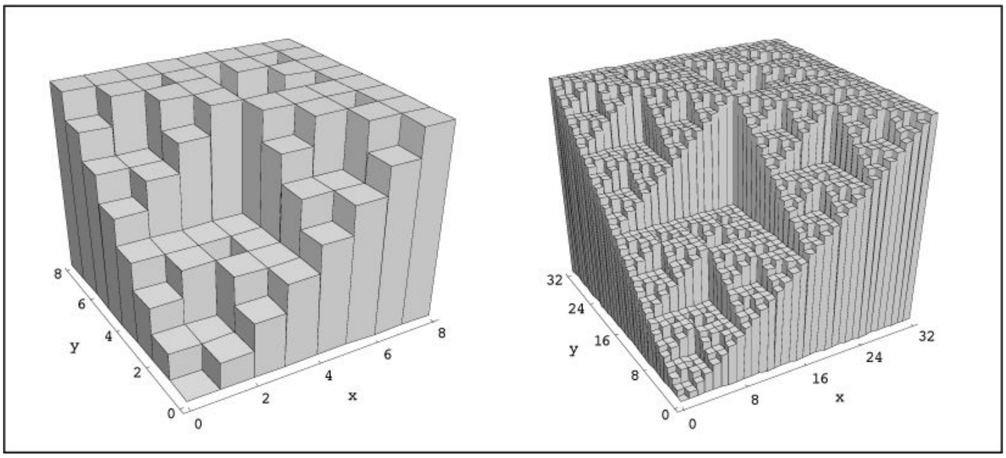


FIGURE C–2. Plots of the logical *or* function.

In Figure C–3, almost half of the points are hidden behind the diagonal plane $x = \bar{y}$.

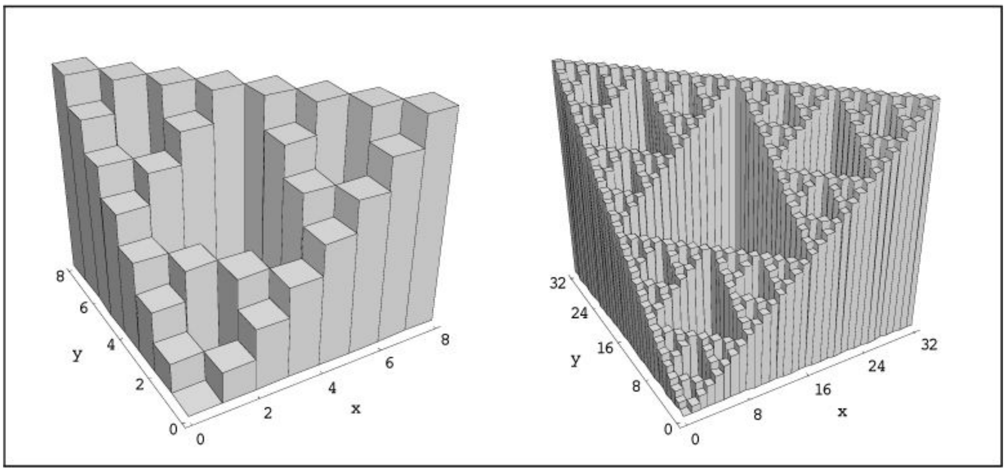


FIGURE C–3. Plots of the logical *exclusive or* function.

For and(*x, y*) (Figure C–1), a certain self-similar, or fractal, pattern of triangles is apparent. If the figure is viewed straight on parallel to the *y*-axis and taken to the limit for large integers, the appearance would be as shown in Figure C–4.

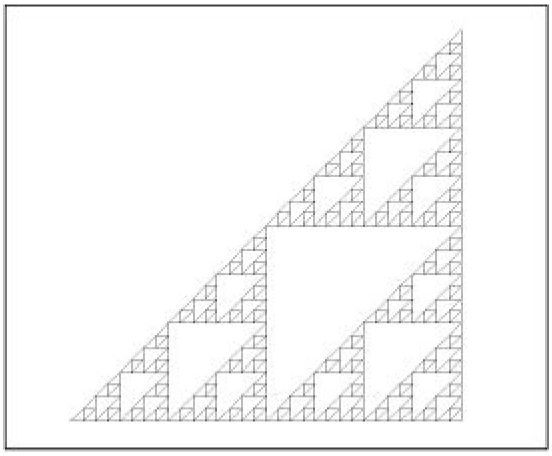


FIGURE C–4. Self-similar pattern made by and(*x, y*).

This is much like the Sierpinski triangle [Sagan], except Figure C–4 uses right triangles whereas Sierpinski used equilateral triangles. In Figure C–3, a pattern along the slanted plane is evident that is precisely the Sierpinski triangle if carried to the limit.

C–2 Plots of Addition, Subtraction, and Multiplication

This section includes 3D plots of addition, subtraction, and three forms of multiplication of unsigned numbers, using "computer arithmetic," in Figures C–5 through C–9. Note that for the plot of the addition operation, the origin is the far-left corner.

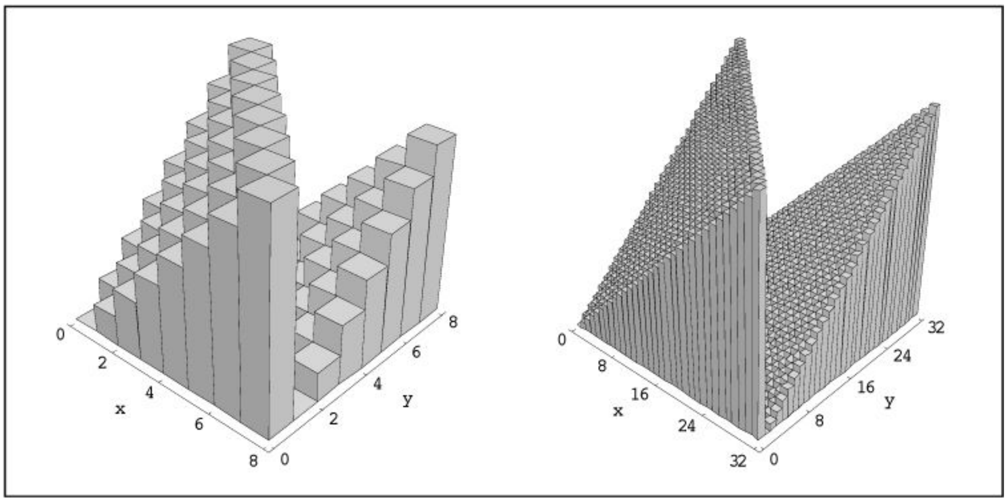


FIGURE C–5. Plots of *x* + *y* (computer arithmetic).

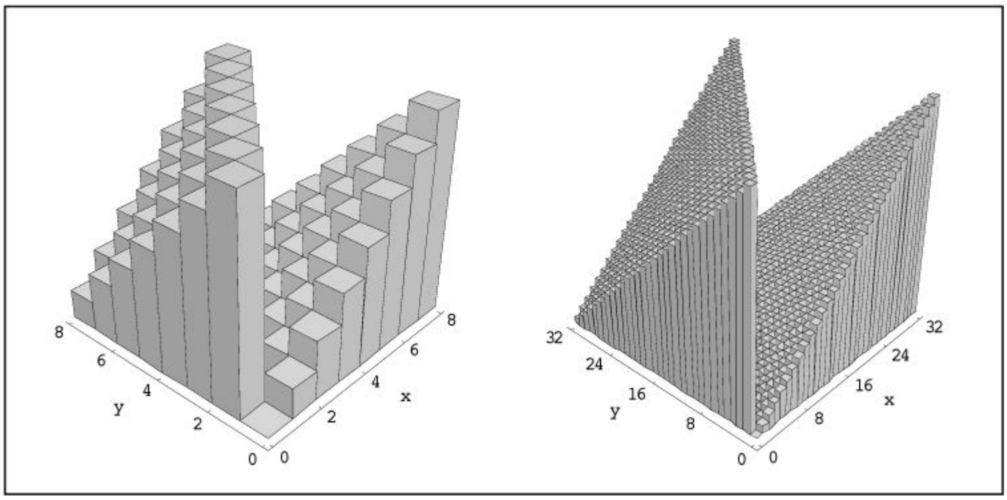


FIGURE C–6. Plots of *x* – *y* (computer arithmetic).

In Figure C–7, the vertical scales are compressed; the highest peaks in the left figure are of height 7·7 = 49.

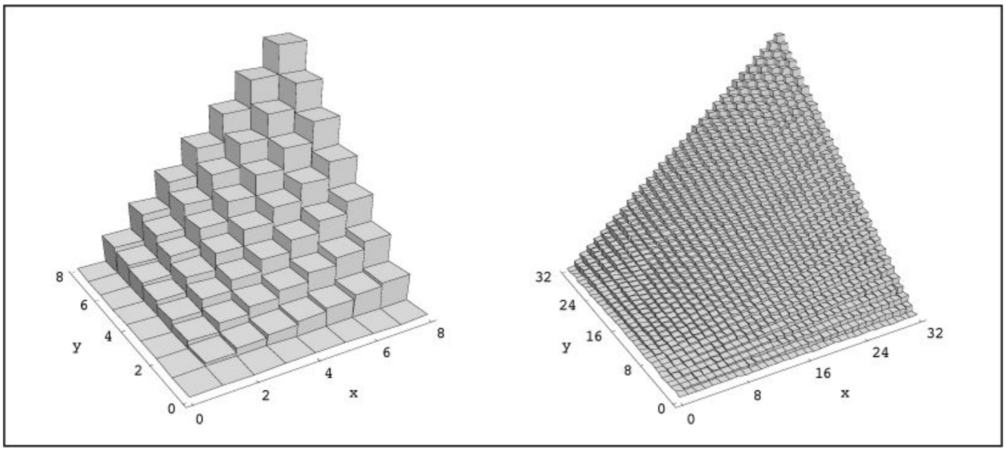


FIGURE C–7. Plots of the unsigned product of *x* and *y*.

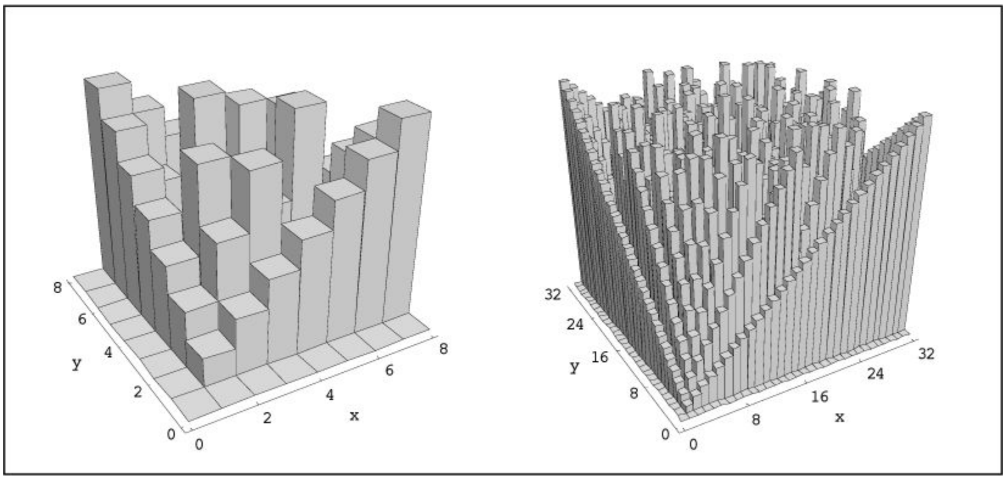


FIGURE C–8. Plots of the low-order half of the unsigned product of *x* and *y*.

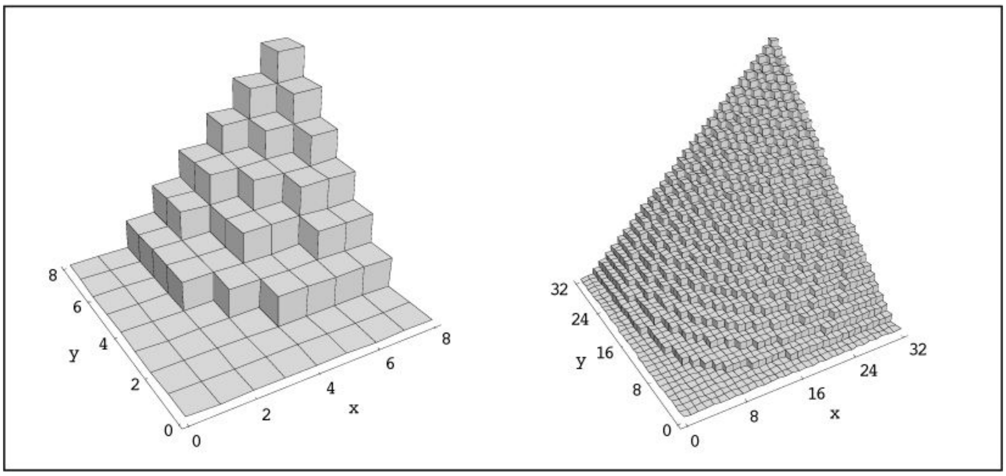


FIGURE C–9. Plots of the high-order half of the unsigned product of *x* and *y*.

C–3 Plots of Functions Involving Division

This section includes 3D plots of the quotient, remainder, greatest common divisor, and least common multiple functions of nonnegative integers *x* and *y*, in Figures C–10, C–11, C–12, and C–13, respectively. Note that in Figure C–10, the origin is the rightmost corner.

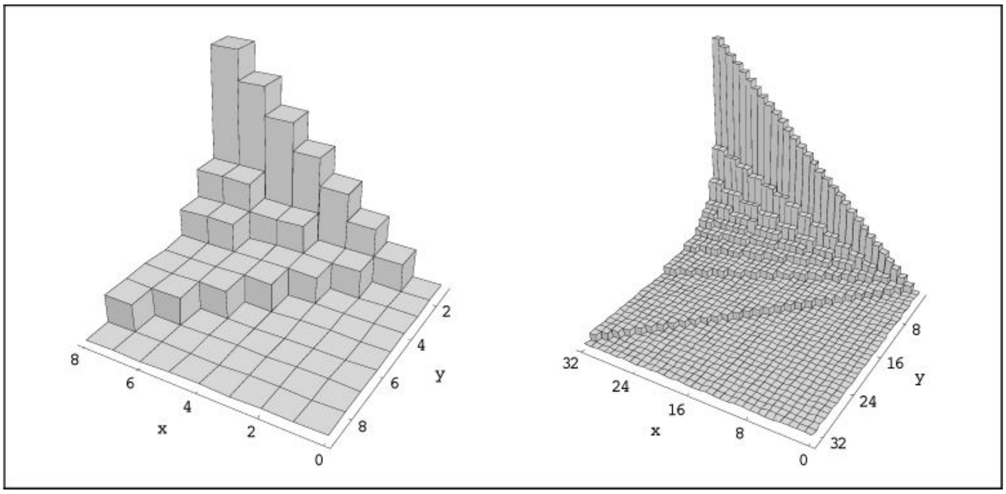


FIGURE C–10. Plots of the integer quotient function *x*÷*y*.

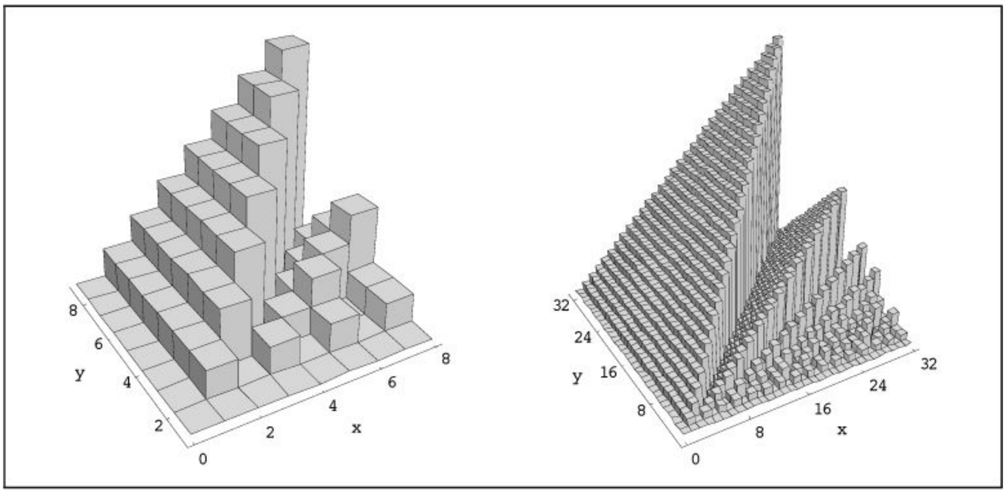


FIGURE C–11. Plots of the remainder function rem(*x, y*).

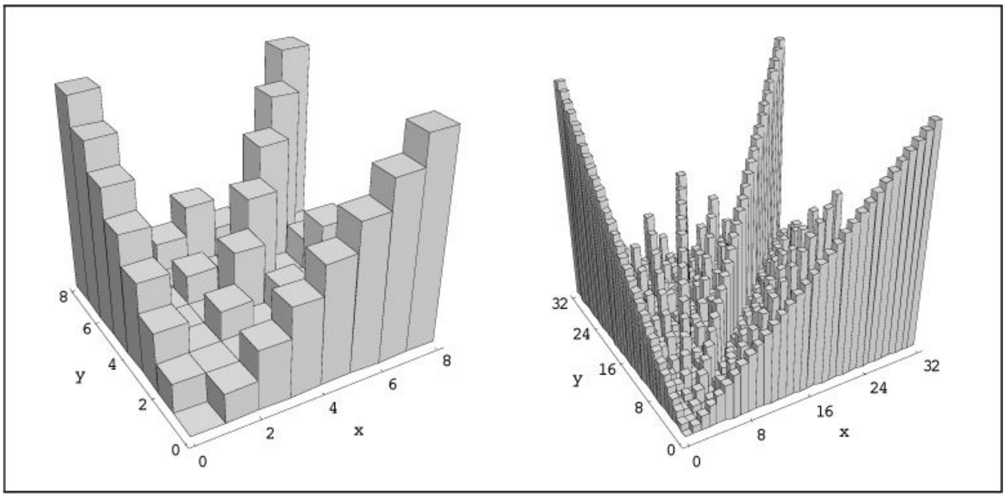


FIGURE C–12. Plots of the greatest common divisor function GCD(*x, y*).

In Figure C–13, the vertical scales are compressed; the highest peaks in the left figure are of height LCM(6, 7) = 42.

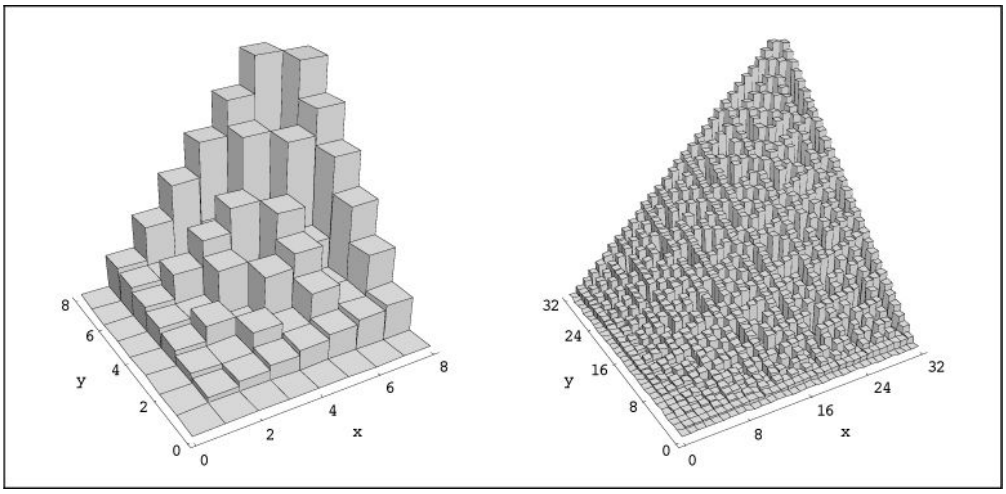


FIGURE C–13. Plots of the least common multiple function LCM(*x, y*).

C–4 Plots of the Compress, SAG, and Rotate Left Functions

This section includes 3D plots of compress(*x, m*), SAG(*x, m*), and rotate left $x \overset{rot}{\ll} r$ as functions of integers *x, m*, and *r*, in Figures C–14, C–15, and C–16, respectively

For compress and SAG, *m* is a mask. For compress, bits of *x* selected by *m* are extracted and compressed to the right, with 0-fill on the left. For SAG, bits of *x* selected by *m* are compressed to the left, and the unselected bits are compressed to the right.

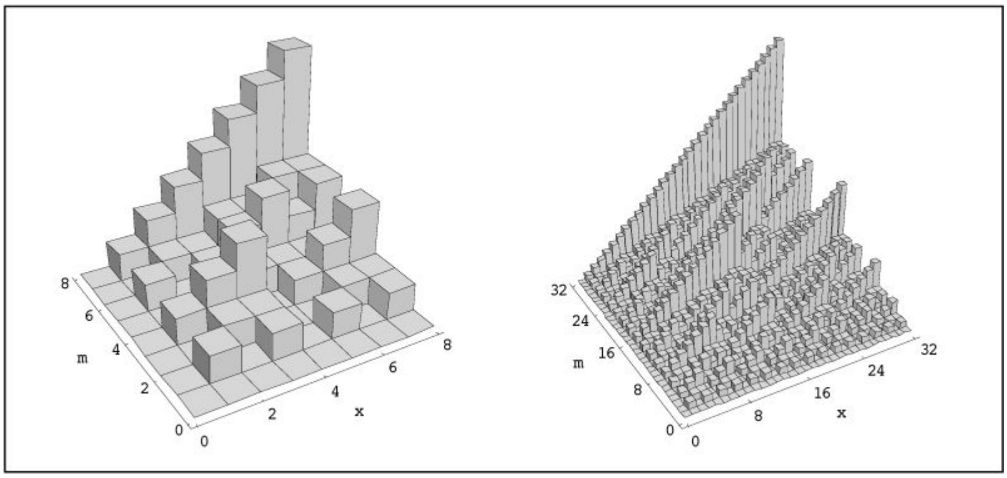


FIGURE C–14. Plots of the generalized extract, or compress(*x*, *m*) function.

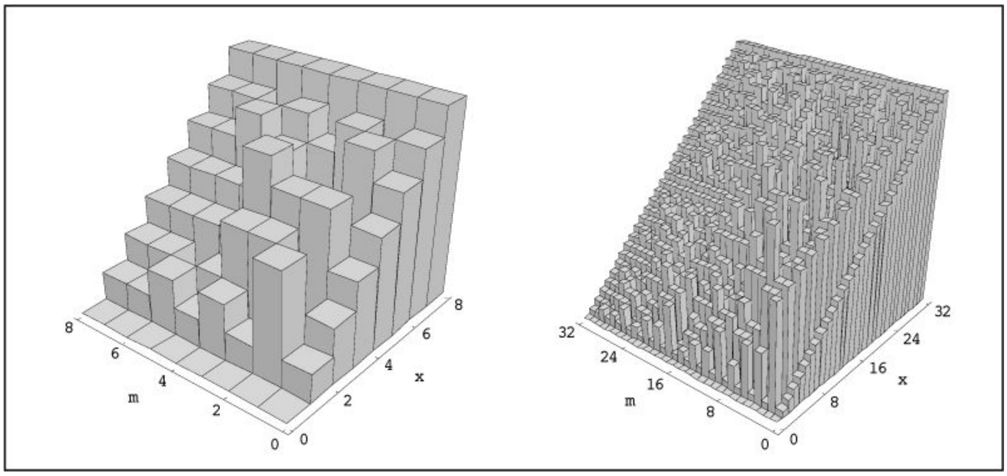


FIGURE C–15. Plots of the sheep and goats function SAG(*x*, *m*).

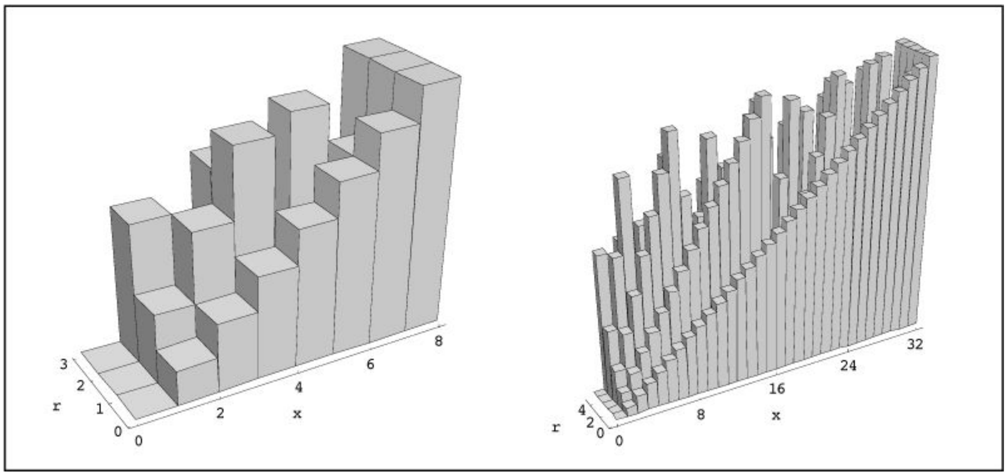


FIGURE C–16. Plots of the rotate left function $x \overset{rot}{\ll} r$

C–5 2D Plots of Some Unary Functions

Figures C–17 through C–21 show 2D plots of some unary functions on bit strings that are reinterpreted as functions on integers. Like the 3D plots, these were also produced by Mathematica. For most functions, two plots are shown: one for a word size of four bits and the other for a word size of seven bits.

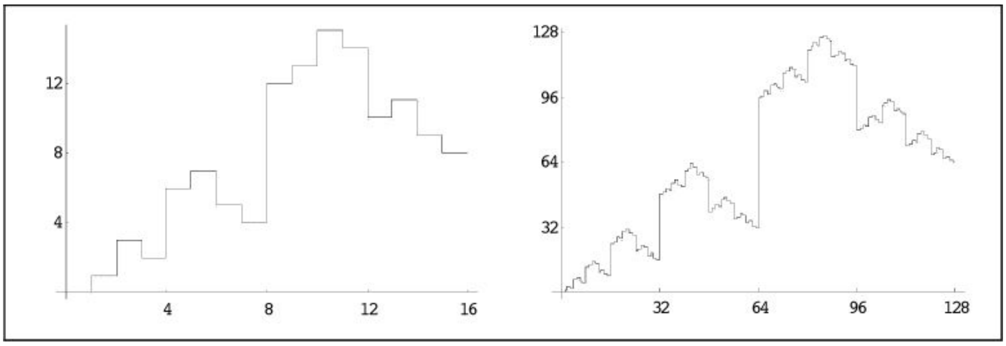


FIGURE C–17. Plots of the Gray code function.

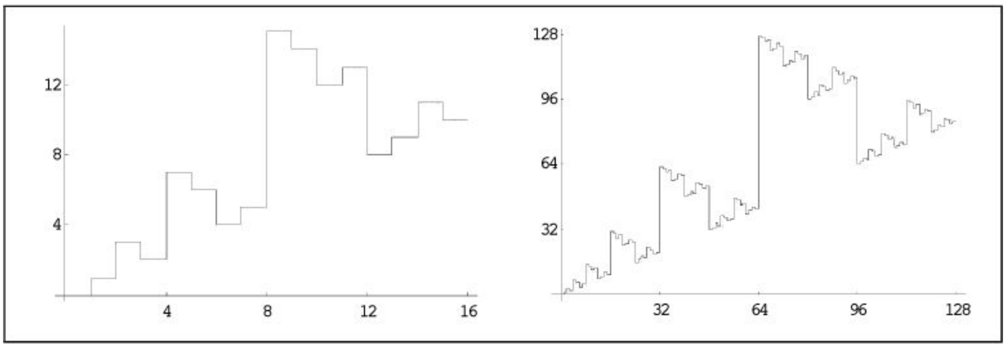


FIGURE C–18. Plots of the inverse Gray code function.

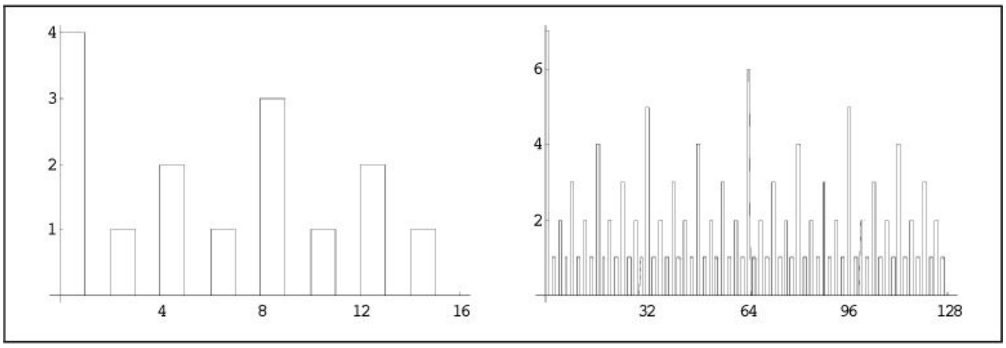


FIGURE C–19. Plots of the ruler function (number of trailing zeros).

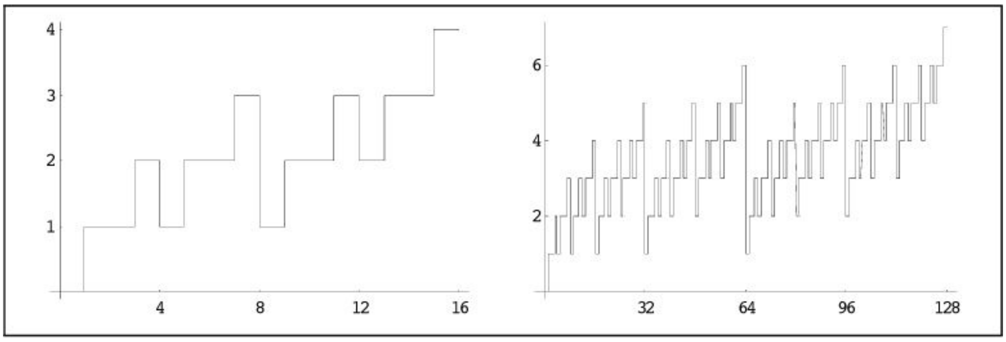


FIGURE C–20. Plots of the population count function (number of 1-bits).

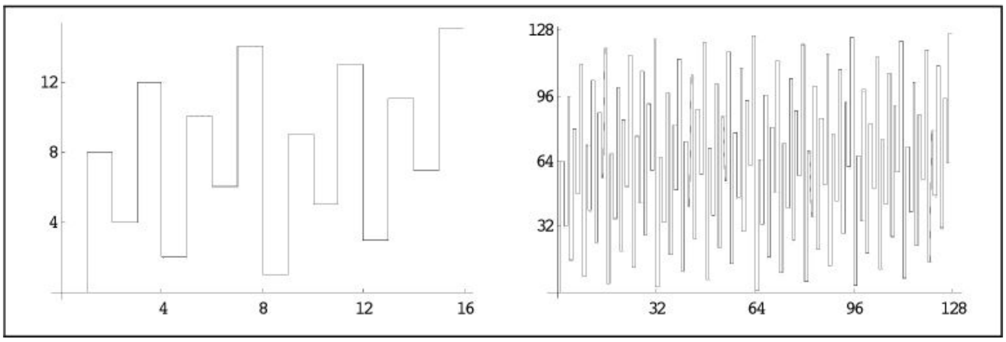


FIGURE C–21. Plots of the bit reversal function.

"Gray code function" refers to a function that maps an integer that represents a displacement or rotation amount to the Gray encoding for that displacement or rotation amount. The inverse Gray code function maps a Gray encoding to a displacement or rotation amount. See Figure 13–1 on page 313.

Figure C–22 shows what happens to a deck of 16 cards, numbered 0 to 15, after one, two, and three outer perfect shuffles (in which the first and last cards do not move). The *x* coordinate is the original position of a card, and the *y* coordinate is the final position of that card after one, two, or three shuffles. Figure C–23 is the same for one, two, and three perfect *inner* shuffles. Figures C–24 and C–25 are for the inverse operations.

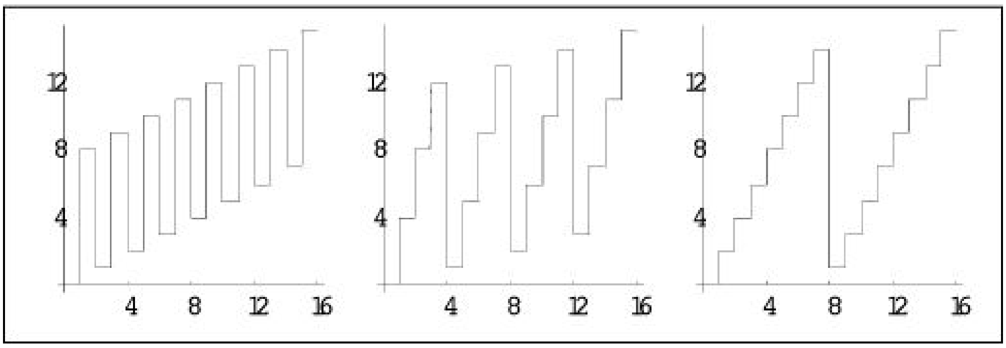


FIGURE C–22. Plots of the outer perfect shuffle function.

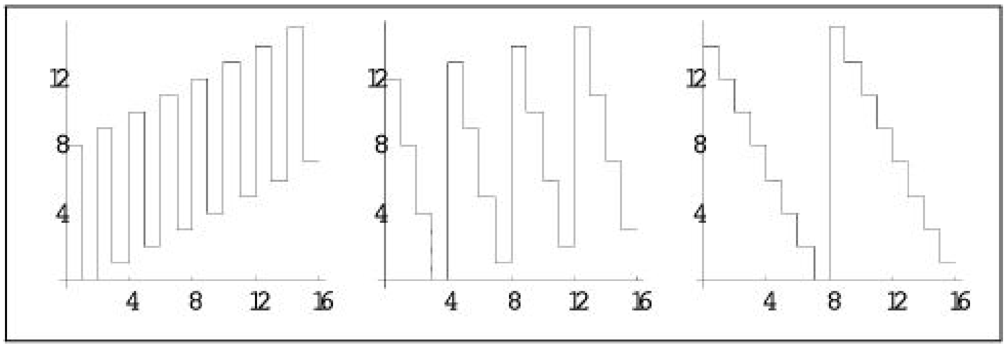


FIGURE C–23. Plots of the inner perfect shuffle function.

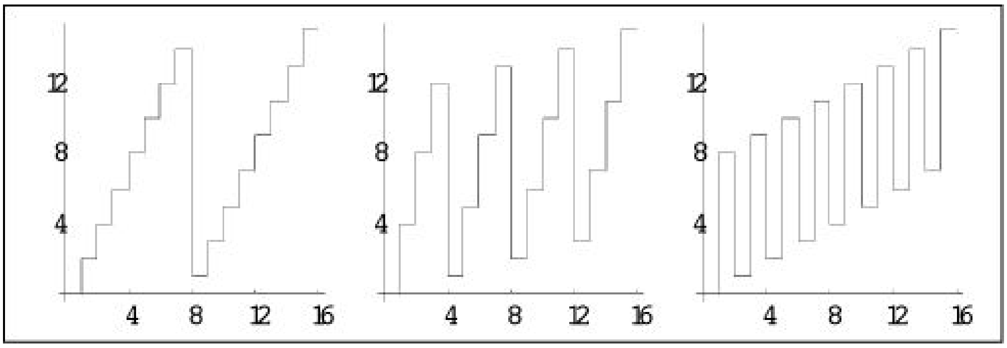


FIGURE C–24. Plots of the outer perfect unshuffle function.

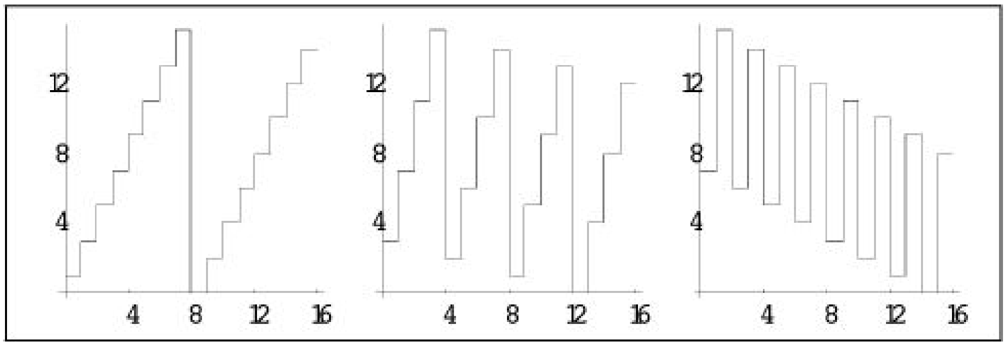


FIGURE C–25. Plots of the inner perfect unshuffle function.

Figures C–26 and C–27 show the mapping that results from shuffling the bits of an integer of four and eight bits in length. Informally,

shuffleBits(x) = asInteger(shuffle(bits(x)))

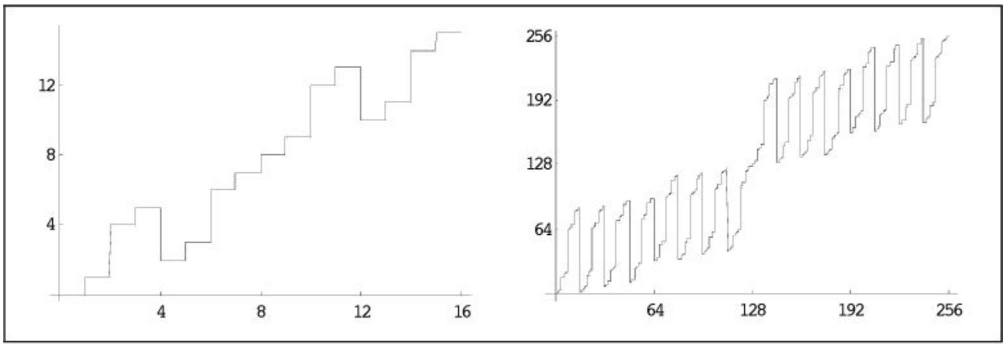


FIGURE C–26. Plots of the outer perfect shuffle bits function.

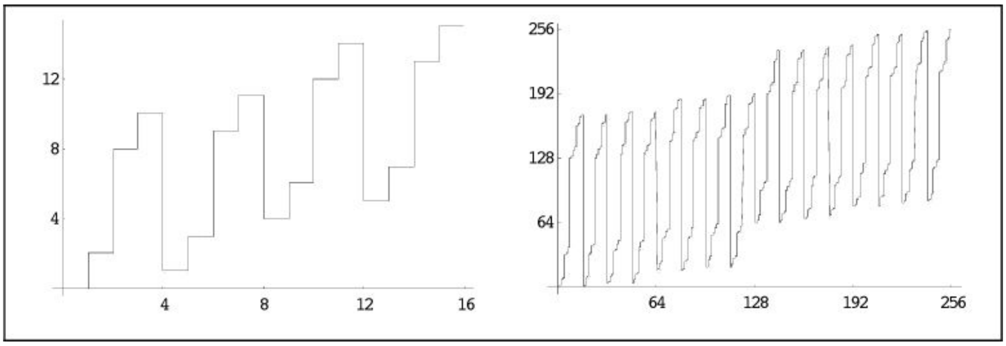


FIGURE C–27. Plots of the inner perfect shuffle bits function.

Bibliography

[AES]

Advanced Encryption Standard (AES), National Institute of Standards and Technology, FIPS PUB 197 (November 2001). Available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[Agrell]

Agrell, Erik. http://webfiles.portal.chalmers.se/s2/research/kit/bounds/, table last updated July 2004.

[Allen]

Allen, Joseph H. Private communication.

[Alv]

Alverson, Robert. “Integer Division Using Reciprocals.” In *Proceedings IEEE 10th Symposium on Computer Arithmetic*, June 26–28, 1991, Grenoble, France, 186–190.

[Arndt]

Arndt, Jörg. *Matters Computational: Ideas, Algorithms, Source Code*. Springer-Verlag, 2010. Also available at http://www.jjj.de/fxt/#fxtbook.

[Aus1]

Found in a REXX interpreter subroutine written by Marc A. Auslander.

[Aus2]

Auslander, Marc A. Private communication.

[Baum]

D. E. Knuth attributes the ternary method to an unpublished memo from the mid-1970s by Bruce Baumgart, which compares about 20 different methods for bit reversal on the PDP10.

[Bern]

Bernstein, Robert. “Multiplication by Integer Constants.” *Software—Practice and Experience 16*, 7 (July 1986), 641–652.

[BGN]

Burks, Arthur W., Goldstine, Herman H., and von Neumann, John. “Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, Second Edition” (1947). In *Papers of John von Neumann on Computing and Computing Theory*, Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987.

[Black]

Black, Richard. Web site www.cl.cam.ac.uk/Research/SRG/bluebook/21/crc/crc.html. University of Cambridge Computer Laboratory Systems Research Group, February 1994.

[Bonz]

Bonzini, Paolo. Private communication.

[Brou]

Brouwer, Andries E. http://www.win.tue.nl/~aeb/codes/binary-1.html, table last updated January 2012.

[CavWer]

Cavagnino, D. and Werbrouck, A. E. “Efficient Algorithms for Integer Division by Constants Using Multiplication.” *The Computer Journal 51*, 4 (2008), 470–480.

[CC]

Caldwell, Chris K. and Cheng, Yuanyou. “Determining Mills’ Constant and a Note on Honaker’s Problem.” *Journal of Integer Sequences 8*, 4 (2005), article 05.4.1, 9 pp. Also available at http://www.cs.uwaterloo.ca/journals/JIS/VOL8/Caldwell/caldwell78.pdf.

[CJS]

Stephenson, Christopher J. Private communication.

[Cohen]

These rules were pointed out by Norman H. Cohen.

[Cplant]

Leung, Vitus J., et. al. “Processor Allocation on Cplant: Achieving General Processor Locality Using One-Dimensional Allocation Strategies.” In *Proceedings 4th IEEE International Conference on Cluster Computing*, September 2002, 296–304.

[Cut]

Cutland, Nigel J. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[CWG]

Hoxey, Karim, Hay, and Warren (Editors). *The PowerPC Compiler Writer’s Guide*. Warthman Associates, 1996.

[Dalton]

Dalton, Michael. Private communication.

[Danne]

Dannemiller, Christopher M. Private communication. He attributes this code to the Linux Source base, www.gelato.unsw.edu.au/lxr/source/lib/crc32.c, lines 105–111.

[DES]

Data Encryption Standard (DES), National Institute of Standards and Technology, FIPS PUB 46-2 (December 1993). Available at http://www.itl.nist.gov/fipspubs/fip46-2.htm.

[Dewd]

Dewdney, A. K. *The Turing Omnibus*. Computer Science Press, 1989.

[Dietz]

Dietz, Henry G. http://aggregate.org/MAGIC/.

[Ditlow]

Ditlow, Gary S. Private communication.

[Dubé]

Dubé, Danny. Newsgroup comp.compression.research, October 3, 1997.

[Dud]

Dudley, Underwood. “History of a Formula for Primes.” *American Mathematics Monthly 76* (1969), 23–28.

[EL]

Ercegovac, Miloš D. and Lang, Tomás. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994.

[Etzion]

Etzion, Tuvi. “Constructions for Perfect 2-Burst-Correcting Codes,” *IEEE Transactions on Information Theory 47*, 6 (September 2001), 2553–2555.

[Floyd]

Floyd, Robert W. “Permuting Information in Idealized Two-Level Storage.” In *Complexity of Computer Computations* (Conference proceedings), Plenum Press, 1972, 105–109. This is the earliest reference I know of for this method of transposing a $2^n \times 2^n$ matrix.

[Gard]

Gardner, Martin. “Mathematical Games” column in *Scientific American 227*, 2 (August 1972), 106–109.

[Gaud]

Gaudet, Dean. Private communication.

[GGS]

Gregoire, Dennis G., Groves, Randall D., and Schmookler, Martin S. *Single Cycle Merge/Logic Unit*, US Patent No. 4,903,228, February 20, 1990.

[GK]

Granlund, Torbjörn and Kenner, Richard. “Eliminating Branches Using a Superoptimizer and the GNU C Compiler.” In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), July 1992, 341–352.

[GKP]

Graham, Ronald L., Knuth, Donald E., and Patashnik, Oren. *Concrete Mathematics: A Foundation for Computer Science, Second Edition*. Addison-Wesley, 1994.

[GLS1]

Steele, Guy L., Jr. Private communication.

[GLS2]

Steele, Guy L., Jr. “Arithmetic Shifting Considered Harmful.” AI Memo 378, MIT Artificial Intelligence Laboratory (September 1976); also in *SIGPLAN Notices 12*, 11 (November 1977), 61–69.

[GM]

Granlund, Torbjörn and Montgomery, Peter L. “Division by Invariant Integers Using Multiplication.” In *Proceedings of the ACM SIGPLAN ’94 Conference on*

Programming Language Design and Implementation (PLDI), August 1994, 61–72.

[Gold]

The second expression is due to Richard Goldberg.

[Good]

Goodstein, Prof. R. L. “Formulae for Primes.” *The Mathematical Gazette 51* (1967), 35–36.

[Gor]

Goryavsky, Julius. Private communication.

[GSO]

Found by the GNU Superoptimizer.

[HAK]

Beeler, M., Gosper, R. W., and Schroeppel, R. *HAKMEM*, MIT Artificial Intelligence Laboratory AIM 239, February 1972.

[Ham]

Hamming, Richard W., “Error Detecting and Error Correcting Codes,” *The Bell System Technical Journal 26*, 2 (April 1950), 147–160.

[Harley]

Harley, Robert. Newsgroup comp.arch, July 12, 1996.

[Hay1]

Hay, R. W. Private communication.

[Hay2]

The first expression was found in a compiler subroutine written by R. W. Hay.

[Hil]

Hilbert, David. “Ueber die stetige Abbildung einer Linie auf ein Flächenstück.” *Mathematischen Annalen 38* (1891), 459–460.

[Hill]

Hill, Raymond. *A First Course in Coding Theory*. Clarendon Press, 1986.

[HilPat]

Hiltgen, Alain P. and Paterson, Kenneth G. “Single-Track Circuit Codes.” *IEEE Transactions on Information Theory 47*, 6 (2001) 2587-2595.

[Hop]

Hopkins, Martin E. Private communication.

[HS]

Hillis, W. Daniel and Steele, Guy L., Jr. “Data Parallel Algorithms.” *Comm. ACM 29*, 12 (December 1986) 1170–1183.

[Hsieh]

Hsieh, Paul. Newsgroup comp.lang.c, April 29, 2005.

[Huef]

Hueffner, Falk. Private communication.

[H&P]

Hennessy, John L. and Patterson, David A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[H&S]

Harbison, Samuel P. and Steele, Guy L., Jr. *C: A Reference Manual*, Fourth Edition. Prentice-Hall, 1995.

[H&W]

Hardy, G. H. and Wright, E. M. *An Introduction to the Theory of Numbers*, Fourth Edition. Oxford University Press, 1960.

[IBM]

From an IBM programming course, 1961.

[Irvine]

Irvine, M. M. "Early Digital Computers at Bell Telephone Laboratories." *IEEE Annals of the History of Computing 23*, 3 (July–September 2001), 22–42.

[JVN]

von Neumann, John. "First Draft of a Report on the EDVAC." In *Papers of John von Neumann on Computing and Computing Theory*, Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987.

[Karat]

Karatsuba, A. and Ofman, Yu. "Multiplication of multidigit numbers on automata." *Soviet Physics-Doklady 7*, 7 (January 1963), 595–596. They show the theoretical result that multiplication of *m*-bit integers is $O(m^{\log_2 3}) \approx O(m^{1.585})$, but the details of their method are more cumbersome than the method based on Gauss's three-multiplication scheme for complex numbers.

[Karv]

Karvonen, Vesa. Found at "The Assembly Gems" web page, www.df.lth.se/~john_e/fr_gems.html.

[Keane]

Keane, Joe. Newsgroup sci.math.num-analysis, July 9, 1995.

[Ken]

Found in a GNU C compiler for the IBM RS/6000 that was ported by Richard Kenner. He attributes this to a 1992 PLDI conference paper by him and Torbjörn Granlund.

[Knu1]

Knuth, Donald E. *The Art of Computer Programming, Volume 1, Third Edition: Fundamental Algorithms*. Addison-Wesley, 1997.

[Knu2]

Knuth, Donald E. *The Art of Computer Programming, Volume 2, Third Edition: Seminumerical Algorithms*. Addison-Wesley, 1998.

[Knu3]

The idea of using a negative integer as the base of a number system for arithmetic has been independently discovered by many people. The earliest reference given by Knuth is to Vittorio Grünwald in 1885. Knuth himself submitted a paper on the subject in 1955 to a “science talent search” for high-school seniors. For other early references, see [Knu2].

[Knu4]

Knuth, Donald E. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*, Section 7.1.1. Addison-Wesley, 2011.

[Knu5]

Ibid, Section 7.1.3. Knuth attributes the equality relation to W. C. Lynch in 2006.

[Knu6]

Ibid, Section 7.2.1.1, Exercise 80.

[Knu7]

Knuth, Donald E. *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX—A RISC Computer for the New Millennium*. Addison-Wesley, 2005.

[Knu8]

Knuth, Donald E. Private communication.

[KRS]

Kruskal, Clyde P., Rudolph, Larry, and Snir, Marc. “The Power of Parallel Prefix.” *IEEE Transactions on Computers C-34*, 10 (October 1985), 965–968.

[Kumar]

This figure was suggested by Gowri Kumar (private communication).

[Lamp]

Lamport, Leslie. “Multiple Byte Processing with Full-Word Instructions.” *Communications of the ACM 18*, 8 (August 1975), 471–475.

[Lang]

Langdon, Glen G. Jr., “Subtraction by Minuend Complementation,” *IEEE Transactions on Computers C-18*, 1 (January 1969), 74–76.

[LC]

Lin, Shu and Costello, Daniel J., Jr. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.

[Lomo]

Lomont, Chris. *Fast Inverse Square Root*. www.lomont.org/Math/Papers/2003/InvSqrt.pdf.

[LPR]

Leiserson, Charles E., Prokop, Harald, and Randall, Keith H. *Using de Bruijn Sequences to Index a 1 in a Computer Word*. MIT Laboratory for Computer Science, July 7, 1998. Also available at http://supertech.csail.mit.edu/papers/debruijn.pdf.

[LSY]

Lee, Ruby B., Shi, Zhijie, and Yang, Xiao. “Efficient Permutation Instructions for Fast Software Cryptography.” *IEEE Micro 21*, 6 (November/December 2001), 56–69.

[L&S]

Lam, Warren M. and Shapiro, Jerome M. “A Class of Fast Algorithms for the Peano-Hilbert Space-Filling Curve.” In *Proceedings ICIP 94*, 1 (1994), 638–641.

[MD]

Denneau, Monty M. Private communication.

[MIPS]

Kane, Gerry and Heinrich, Joe. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[MM]

Morton, Mike. “Quibbles & Bits.” *Computer Language 7*, 12 (December 1990), 45–55.

[Möbi]

Möbius, Stefan K. Private communication.

[MS]

MacWilliams, Florence J. and Sloane, Neil J. A. *The Theory of Error-Correcting Codes, Part II*. North-Holland, 1977.

[Mycro]

Mycroft, Alan. Newsgroup comp.arch, April 8, 1987.

[Neum]

Neumann, Jasper L. Private communication.

[NZM]

Niven, Ivan, Zuckerman, Herbert S., and Montgomery, Hugh L. *An Introduction to the Theory of Numbers, Fifth Edition*. John Wiley & Sons, Inc., 1991.

[PeBr]

Peterson, W. W. and Brown, D. T. “Cyclic Codes for Error Detection.” In *Proceedings of the IRE*, 1 (January 1961), 228–235.

[PHO]

Oden, Peter H. Private communication.

[PL8]

I learned this trick from the PL.8 compiler.

[PuBr]

Purdom, Paul Walton Jr., and Brown, Cynthia A. *The Analysis of Algorithms*. Holt, Rinehart and Winston, 1985.

[Reiser]

Reiser, John. Newsgroup comp.arch.arithmetic, December 11, 1998.

[Rib]

Ribenboim, Paulo. *The Little Book of Big Primes*. Springer-Verlag, 1991.

[RND]

Reingold, Edward M., Nievergelt, Jurg, and Deo, Narsingh. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[Roman]

Roman, Steven. *Coding and Information Theory*. Springer-Verlag, 1992.

[Sagan]

Sagan, Hans. *Space-Filling Curves*. Springer-Verlag, 1994. A wonderful book, thoroughly recommended to anyone even slightly interested in the subject.

[Seal1]

Seal, David. Newsgroup comp.arch.arithmetic, May 13, 1997. Harley was the first known to this writer to apply the CSA to this problem, and Seal showed a particularly good way to use it for counting the bits in a large array (as illustrated in Figures 5–8 and 5–9), and also for an array of size seven (similar to the plan of Figure 5–10).

[Seal2]

Seal, David. Newsgroup comp.sys.acorn.tech, February 16, 1994.

[Shep]

Shepherd, Arvin D. Private communication.

[Stall]

Stallman, Richard M. *Using and Porting GNU CC*. Free Software Foundation, 1998.

[Strach]

Strachey, Christopher. "Bitwise Operations." *Communications of the ACM 4*, 3 (March 1961), 146. This issue contains another paper that gives two methods for bit reversal ("Two Methods for Word Inversion on the IBM 709," by Robert A. Price and Paul Des Jardins; there is a small correction on page A13 of the March 1961 issue). These methods are not discussed in this book because they rely on the somewhat exotic *Convert by Addition from the MQ* (CAQ) instruction of that machine. That instruction does a series of indexed table lookups, adding the word fetched from memory to the accumulator. It is not a RISC instruction.

[Tanen]

Tanenbaum, Andrew S. *Computer Networks*, Second Edition. Prentice Hall, 1988.

[Taro]

The author of this program seems to be lost in history. One of the earliest people to use it and to tweak the constant a bit was Gary Tarolli, probably while he was at SGI. He also helped to make it more widely known and says it goes back to 1995 or earlier. For more on the history see http://www.beyond3d.com/content/articles/8/.

[Voor]

Voorhies, Douglas. "Space-Filling Curves and a Measure of Coherence." *Graphics Gems II*, AP Professional (1991).

[War]

Warren, H. S., Jr. "Functions Realizable with Word-Parallel Logical and Two's-Complement Addition Instructions." *Communications of the ACM 20*, 6 (June 1977), 439–441.

[Weg]

The earliest reference to this that I know of is: Wegner, P. A. “A Technique for Counting Ones in a Binary Computer.” *Communications of the ACM 3*, 5 (May 1960), 322.

[Wells]

Wells, David. *The Penguin Dictionary of Curious and Interesting Numbers*. Penguin Books, 1997.

[Will]

Willans, C. P. “On Formulae for the *n*th Prime Number.” *The Mathematical Gazette 48* (1964), 413–415.

[Wood]

Woodrum, Luther. Private communication. The second formula uses no literals and works well on the IBM System/370.

[Wor]

Wormell, C. P. “Formulae for Primes.” *The Mathematical Gazette 51* (1967), 36–38.

[Zadeck]

Zadeck, F. Kenneth. Private communication.

Footnotes

Foreword

1. Why “HAKMEM”? Short for “hacks memo”; one 36-bit PDP-10 word could hold six 6-bit characters, so a lot of the names PDP-10 hackers worked with were limited to six characters. We were used to glancing at a six-character abbreviated name and instantly decoding the contractions. So naming the memo “HAKMEM” made sense at the time—at least to the hackers.

Preface

1. One such program, written in C, is:
   main(){char*p=”main(){char*p=%c%s%c;(void)printf(p,34,p,34,10);}%c”;(void)printf(p,34,p,34,10);}

Chapter 2

1. A variation of this algorithm appears in [H&S] sec. 7.6.7.
2. This is useful to get unsigned comparisons in Java, which lacks unsigned integers.
3. Mathematicians name the operation *monus* and denote it with $\dot{-}$. The terms *positive difference* and *saturated subtraction* are also used.
4. A destructive operation is one that overwrites one or more of its arguments.
5. Horner’s rule simply factors out x. For example, it evaluates the fourth-degree polynomial $ax^4 + bx^3 + cx^2 + dx + e$ as $x(x(x(ax + b) + c) + d) + e$. For a polynomial of degree n it takes n multiplications and n additions, and it is very suitable for the *multiply-add* instruction.
6. Logic designers will recognize this as Reed-Muller, a.k.a positive Davio, decomposition. According to Knuth [Knu4, 7.1.1], it was known to I. I. Zhegalkin [Matematicheskii Sbornik 35 (1928), 311–369]. It is sometimes referred to as the Russian decomposition.
7. The entire 335-page work is available at www.gutenberg.org/etext/15114.

Chapter 3

1. pop(x) is the number of 1-bits in x.

Chapter 4

1. In the sense of more compact, less branchy, code; faster-running code may result from checking first for the case of no overflow, assuming the limits are not likely to be large.

Chapter 5

1. A full adder is a circuit with three 1-bit inputs (the bits to be added) and two 1-bit outputs (the sum and carry).
2. The flakiness is due to the way C is used. The methods illustrated would be perfectly acceptable if coded in machine language, or generated by a compiler, for a particular machine.

Chapter 7

1. Actually, the first *shift left* can be omitted, reducing the instruction count to 126. The quantity `mv` comes out the same with or without it [Dalton].
2. If big-endian bit numbering is used, compress to the left all bits marked with 0’s, and to the right all bits marked with 1’s.

Chapter 8

1. Reportedly this was known to Gauss.

Chapter 9

1. I may be taken to task for this nomenclature, because there is no universal agreement that

"modulus" implies "nonnegative." Knuth's "mod" operator [Knu1] is the remainder of floor division, which is negative (or 0) if the divisor is negative. Several programming languages use "mod" for the remainder of truncating division. However, in mathematics, "modulus" is sometimes used for the magnitude of a complex number (nonnegative), and in congruence theory the modulus is generally assumed to be positive.

2. Some do try. IBM's PL.8 language uses modulus division, and Knuth's MMIX machine's division instruction uses floor division [Knu7].
3. One execution of the RS/6000's *compare* instruction sets multiple status bits indicating less than, greater than, or equal.
4. Actually, the restoring division algorithm can avoid the restoring step by putting the result of the subtraction in an additional register and writing that register into x only if the result of the subtraction (33 bits) is nonnegative. In some implementations this may require an additional register and possibly more time.

Chapter 12

1. The interested reader might warm up to this challenge.
2. This is the way it was done at Bell Labs back in 1940 on George Stibitz's Complex Number Calculator [Irvine].

Chapter 14

1. Since renamed the ITU-TSS (International Telecommunications Union—Telecommunications Standards Sector).

Chapter 15

1. A perfect code exists for $m = 2^k - k - 1$, k an integer—that is, $m = 1, 4, 11, 26, 57, 120, \ldots$.
2. It is also called the "binomial coefficient" because $\binom{n}{r}$ is the coefficient of the term $x^r y^{n-r}$ in the expansion of the binomial $(x + y)^n$.

Chapter 16

1. Recall that a *curve* is a continuous map from a one-dimensional space to an n-dimensional space.

Chapter 17

1. This is not officially sanctioned C, but with almost all compilers it works.

Chapter 18

1. However, this is the only conjecture of Fermat known to be wrong [Wells].
2. Our apologies for the two uses of π in close proximity, but it's standard notation and shouldn't cause any difficulty.
3. This is my terminology, not Willans's.
4. We have slightly simplified his formula.

Answers To Exercises

1. Base -2 also has this property, but not base $-1 + i$.
2. These formulas were found by the exhaustive expression search program Aha! (A Hacker's Assistant).

Appendix B

1. Newton's method for the special case of the square root function was known to Babylonians about 4,000 years ago.

Index