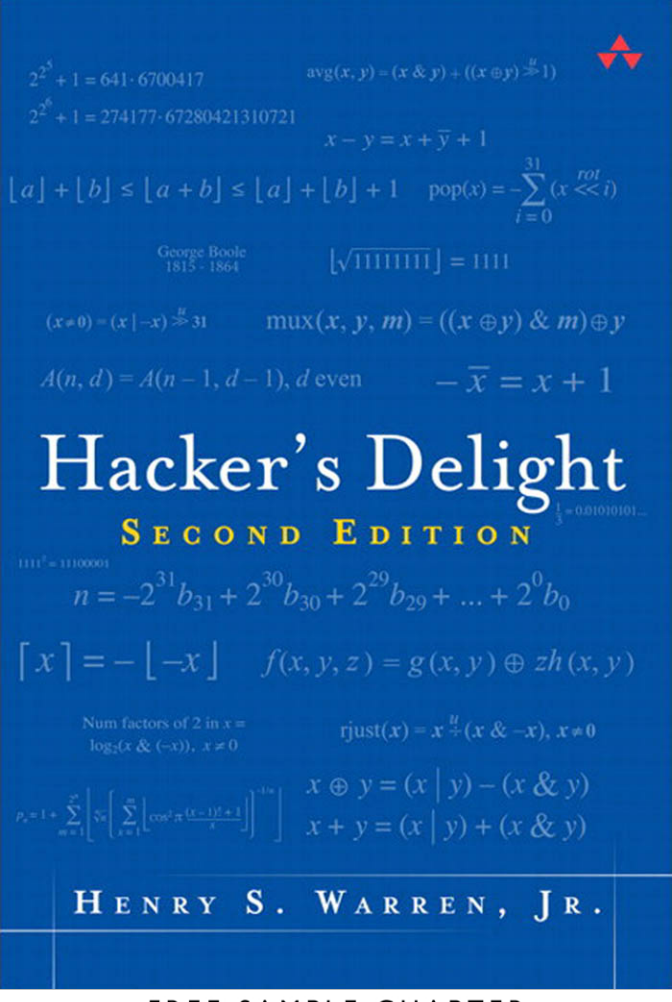


FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Hacker’s Delight

This page intentionally left blank

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

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

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

This page intentionally left blank

CONTENTS

This page intentionally left blank

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

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.

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.

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

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):

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

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

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

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):

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

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):

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

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

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

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.)

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

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

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 contrapositive 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

1. A variation of this algorithm appears in [H&S] sec. 7.6.7.

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

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

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.)

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

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

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

```
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.

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

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 $(x \oplus y)$, 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 $(\boldsymbol{x} \oplus \boldsymbol{y})$, 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 $((\boldsymbol{x} \mid \boldsymbol{y}) \,\&\, \neg(\boldsymbol{x} \,\&\, \boldsymbol{y}))$. 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:

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

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

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

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), $f(x, y) \overset{u}{\leq} g(x, y)$. Clearly, this extends to word-parallel logical operations. One can easily read off such relations (most of which are trivial) as $(x \,\&\, y) \overset{u}{\leq} x \overset{u}{\leq} (x \mid \neg y)$, and so on. Furthermore, if two columns have a row in which one entry is 0 and the other is 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

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 $f(x, y) \overset{u}{\leq} g(x, y)$ 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. $(\boldsymbol{x} \mid \boldsymbol{y}) \overset{u}{\geq} \max(\boldsymbol{x}, \boldsymbol{y})$

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

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

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

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

The proofs of these are quite simple, except possibly for the relation $|\boldsymbol{x} - \boldsymbol{y}| \overset{u}{\leq} (\boldsymbol{x} \oplus \boldsymbol{y})$. By $|\boldsymbol{x} - \boldsymbol{y}|$ we mean the absolute value of $\boldsymbol{x} - \boldsymbol{y}$, which can be computed within the domain of unsigned numbers as $\max(\boldsymbol{x}, \boldsymbol{y}) - \min(\boldsymbol{x}, \boldsymbol{y})$. This relation can be proven by induction on the length of $\boldsymbol{x}$ and $\boldsymbol{y}$ (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 $\boldsymbol{y} \leftarrow \boldsymbol{x} \overset{s}{\gg} 31$, and then one of the following:

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

By "$2\boldsymbol{x}$" we mean, of course, $\boldsymbol{x} + \boldsymbol{x}$ or $\boldsymbol{x} \ll 1$.

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

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

2–5 Average of Two Integers

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

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

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

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

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 $\boldsymbol{x} \oplus \boldsymbol{y}$):

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

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

Some common special cases can be done more efficiently. If $\boldsymbol{x}$ and $\boldsymbol{y}$ are signed integers and known to be nonnegative, then the average can be computed as simply $(\boldsymbol{x} + \boldsymbol{y}) \overset{u}{\gg} 1$. 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 $\boldsymbol{x}$ and $\boldsymbol{y}$ are unsigned integers and $\boldsymbol{x} \overset{u}{\leq} \boldsymbol{y}$, or if $\boldsymbol{x}$ and $\boldsymbol{y}$ are signed integers and $\boldsymbol{x} \leq \boldsymbol{y}$ (signed comparison), then the average is given by $\boldsymbol{x} + ((\boldsymbol{y} - \boldsymbol{x}) \overset{u}{\gg} 1)$. 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 $0 \le n \le 31$ and, if the machine has mod-64 shifts, the last holds for $0 \le n \le 63$. 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.

$$((x + \mathbf{0x80000000}) \overset{u}{\gg} n) - (\mathbf{0x80000000} \overset{u}{\gg} n)$$
$$t \leftarrow \mathbf{0x80000000} \overset{u}{\gg} n; \qquad ((x \overset{u}{\gg} n) \oplus t) - t$$
$$t \leftarrow (x \,\&\, \mathbf{0x80000000}) \overset{u}{\gg} n; \; (x \overset{u}{\gg} n) - (t + t)$$
$$(x \overset{u}{\gg} n) \mid (-(x \overset{u}{\gg} 31) \ll \mathbf{31} - n)$$
$$t \leftarrow -(x \overset{u}{\gg} 31); \qquad ((x \oplus t) \overset{u}{\gg} n) \oplus t$$

In the first two formulas, an alternative for the expression $\mathbf{0x80000000} \overset{u}{\gg} n$ is $\mathbf{1} \ll \mathbf{31} - n$.

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

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

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

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

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):

$$\begin{aligned} &(x > y) - (x < y), \text{ or} \\ &(x \geq y) - (x \leq y). \end{aligned}$$

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 $x < y$, $x \le y$, 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

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

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

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

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

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.

$((x-1)\ \&\ 7)+1$ $((x+7)\mid -8)+9$ $8-(-x\ \&\ 7)$

$((x+7)\ \&\ 7)+1$ $((x+7)\mid 8)-7$ $-(-x\mid -8)$

$((x-1)\mid -8)+9$ $((x-1)\ \&\ 8)+x$

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.

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

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 $-\text{abs}(x)$ for $\text{nabs}(x)$. 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 $x > y$, $x \geq y$, and so on, interchange x and y in the formulas for $x < y$, $x \leq y$, 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:

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

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

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:

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

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.

$x = \mathbf{0}$: $\mathrm{abs}(x) - \mathbf{1}$

$\mathrm{abs}(x + \mathbf{0x80000000})$

$\mathrm{nlz}(x) \ll 26$

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

$\neg(x \mid -x)$

$\neg x \,\&\, (x - \mathbf{1})$

$x \neq \mathbf{0}$: $\mathrm{nabs}(x)$

$\mathrm{nlz}(x) - \mathbf{32}$

$x \mid -x$

$(x \overset{u}{\gg} 1) - x$ [CWG]

$$
\begin{array}{ll}
x < 0: & x \\
x \le 0: & x \mid (x - 1) \\
 & x \mid \neg -x \\
x > 0: & x \oplus \mathrm{nabs}(x) \\
 & (x \overset{s}{\gg} 1) - x \\
 & -x \,\&\, \neg x \\
x \ge 0: & \neg x
\end{array}
$$

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

$$x < y \;=\; x + 2^{31} \overset{u}{<} y + 2^{31},$$

$$x \overset{u}{<} y \;=\; x - 2^{31} < y - 2^{31}.$$

Similar relations hold for $\le$, $\overset{u}{\le}$, 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 $\le$, $\overset{u}{\le}$, and so on.

Using either of these devices enables computing all the usual comparison predicates other than = and $\ne$ in terms of any one of them, with at most three additional instructions on most machines. For example, let us take $x \overset{u}{\le} y$ 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:

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

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

---

2. This is useful to get unsigned comparisons in Java, which lacks unsigned integers.

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

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

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

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

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

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."

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

For $x > y$, use the complement of the expression for $x \leq y$, 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($\boldsymbol{x}$), nabs($\boldsymbol{x}$), and doz($\boldsymbol{x}$, $\boldsymbol{y}$), 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 $\boldsymbol{x} - \boldsymbol{y}$ as $\boldsymbol{x} + \bar{\boldsymbol{y}} + 1$, 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)
$\boldsymbol{x} = \boldsymbol{y}$:	Z	
$\boldsymbol{x} \neq \boldsymbol{y}$:	$\bar{Z}$	
$\boldsymbol{x} < \boldsymbol{y}$:	$N \oplus V$	
$\boldsymbol{x} \leq \boldsymbol{y}$:	$(N \oplus V) + Z$	
$\boldsymbol{x} > \boldsymbol{y}$:	$(N \equiv V)\bar{Z}$	
$\boldsymbol{x} \geq \boldsymbol{y}$:	$N \equiv V$	
$\boldsymbol{x} \overset{u}{<} \boldsymbol{y}$:	$\overline{C_o}$	
$\boldsymbol{x} \overset{u}{\leq} \boldsymbol{y}$:	$\overline{C_o} + Z$	
$\boldsymbol{x} \overset{u}{>} \boldsymbol{y}$:	$C_o\bar{Z}$	
$\boldsymbol{x} \overset{u}{\geq} \boldsymbol{y}$:	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,

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

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.

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

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:

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

The assignment to z in the left column sets $z = \mathbf{0x80000000}$ if x and y have the same sign, and sets $z = \mathbf{0}$ if they differ. Then, the addition in the second expression is done with $x \oplus z$ 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 $(x - \mathbf{2^{31}}) + y + c \ge \mathbf{0}$—that is, iff $x + y + c \ge 2^{31}$, 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 $(x + \mathbf{2^{31}}) + y + c < \mathbf{0}$—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 $x + \mathbf{2^{31}}$ 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 $(x \oplus \mathbf{2^{31}}) + (y \oplus \mathbf{2^{31}})$, 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 = $(x \equiv y)\ \&\ (s \oplus x)$ for addition, and $(x \oplus y)\ \&\ (d \oplus x)$ 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**):

carry	borrow
$(x + y + c) \oplus x \oplus y$	$(x - y - c) \oplus x \oplus y$

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.

$$x + y + c, \text{ unsigned}$$
$$(x \,\&\, y) \mid ((x \mid y) \,\&\, \neg(x + y + c))$$
$$(x \overset{u}{\gg} 1) + (y \overset{u}{\gg} 1) + [((x \,\&\, y) \mid ((x \mid y) \,\&\, c)) \,\&\, 1]$$

$$x - y - c, \text{ unsigned}$$
$$(\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x - y - c))$$
$$(\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y - c))$$
$$(x \overset{u}{\gg} 1) - (y \overset{u}{\gg} 1) - [((\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, c)) \,\&\, 1]$$

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.

$x + y$, unsigned	$x + y + 1$, unsigned	$x - y$, unsigned	$x - y - 1$, unsigned
$\neg x \overset{u}{<} y$	$\neg x \overset{u}{\leq} y$	$x \overset{u}{<} y$	$x \overset{u}{\leq} y$
$x + y \overset{u}{<} x$	$x + y + 1 \overset{u}{\leq} x$	$x - y \overset{u}{>} x$	$x - y - 1 \overset{u}{\geq} x$

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 $\text{hi}(x \times y)$ and $\text{lo}(x \times y)$. Then the overflow predicates can be computed as follows [MIPS]:

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

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

The complication arises when $x = -\mathbf{2^{31}}$ and $y = -\mathbf{1}$. In this case the multiplication overflows, but the machine may very well give a result of $-\mathbf{2^{31}}$. 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 $y < \mathbf{0}\ \&\ x = -\mathbf{2^{31}}$. 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

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

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 $c = \mathbf{2^{31}} - \mathbf{1}$ if x and y have the same sign, and $c = \mathbf{2^{31}}$ otherwise.

TABLE 2–2. OVERFLOW TEST FOR SIGNED MULTIPLICATION

	$y > \mathbf{0}$	$y \leq \mathbf{0}$
$x > \mathbf{0}$	$x > \mathbf{0x7FFFFFFF} \div y$	$y < \mathbf{0x80000000} \div x$
$x \leq \mathbf{0}$	$x < \mathbf{0x80000000} \div y$	$x \neq \mathbf{0}\ \&\ y < \mathbf{0x7FFFFFFF} \div x$

$$\boldsymbol{c} \leftarrow ((\boldsymbol{x} \equiv \boldsymbol{y}) \overset{s}{\gg} 31) + \mathbf{2^{31}}$$
$$\boldsymbol{x} \leftarrow \text{abs}(\boldsymbol{x})$$
$$\boldsymbol{y} \leftarrow \text{abs}(\boldsymbol{y})$$
$$\boldsymbol{y} \neq \mathbf{0} \;\vec{\&}\; \boldsymbol{x} \overset{u}{>} (\boldsymbol{c} \overset{u}{\div} \boldsymbol{y})$$

The *number of leading zeros* instruction can be used to give an estimate of whether or not $\boldsymbol{x} * \boldsymbol{y}$ 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 $\boldsymbol{x}$ and $\boldsymbol{y}$, 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 $\boldsymbol{x} = \mathbf{0}$ or $\boldsymbol{y} = \mathbf{0}$). Overflow occurs if the 64-bit product has fewer than 32 leading 0's. Hence,

$\text{nlz}(\boldsymbol{x}) + \text{nlz}(\boldsymbol{y}) \geq 32$: Multiplication definitely does not overflow.

$\text{nlz}(\boldsymbol{x}) + \text{nlz}(\boldsymbol{y}) \leq 30$: Multiplication definitely does overflow.

For $\text{nlz}(\boldsymbol{x}) + \text{nlz}(\boldsymbol{y}) = 31$, overflow may or may not occur. In this case, the overflow assessment can be made by evaluating $t = x\lfloor y/2 \rfloor$. This will not overflow. Since xy is $2t$ or, if y is odd, $2t + x$, the product xy overflows if $t \geq 2^{31}$. 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

$$\boldsymbol{m} = \text{nlz}(\boldsymbol{x}) + \text{nlz}(\bar{\boldsymbol{x}}), \text{ and}$$
$$\boldsymbol{n} = \text{nlz}(\boldsymbol{y}) + \text{nlz}(\bar{\boldsymbol{y}}).$$

```
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 $m \oplus n \oplus (m * n) < 0$). 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:

$$[\text{abs}(y \oplus \mathbf{0x80000000}) \mid (\text{abs}(x) \,\&\, \text{abs}(y \equiv \mathbf{0x80000000}))] < 0$$

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

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

(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 $\lfloor x/y \rfloor$, 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 $\lceil x/y \rceil$. 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:

$$
\begin{aligned}
x' &= |x| \\
y' &= |y| \\
\delta &= ((x \oplus y) \overset{s}{\gg} 63) \,\&\, y' \\
&\text{if } (x' \overset{u}{<} (y' \ll 31) + \delta) \text{ then } \{\text{will not overflow}\}
\end{aligned}
$$

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, $\mathbf{F} \times \mathbf{F} = \mathbf{01}$ signed, and $\mathbf{F} \times \mathbf{F} = \mathbf{E1}$ 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.

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

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

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

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

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

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

Subtraction

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

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

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 $(x + y) \oplus x \oplus y$ 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
$s \leftarrow x + y$	$d \leftarrow x - y$
$c \leftarrow (s \oplus x \oplus y)$ & **0x00010000**	$b \leftarrow (d \oplus x \oplus y)$ & **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 ← x & 0x80808080      // Isolate signs.
b ← a u>> 7             // Integer 1 where x is negative.
m ← (a – b) | a         // 0xFF where x is negative.
y ← (x ⊕ m) + b         // Complement and add 1 where negative.
```

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

2–19 Doz, Max, Min

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

Signed	Unsigned
$\mathrm{doz}(x, y) = \begin{cases} x - y, & x \geq y, \\ 0, & x < y. \end{cases}$	$\mathrm{dozu}(x, y) = \begin{cases} x - y, & x \geq^u y, \\ 0, & x <^u y. \end{cases}$

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 $x \geq y$ or $x < y$. The comparison result is fed to a multiplexor

3. Mathematicians name the operation *monus* and denote it with $\dot{-}$. The terms *positive difference* and *saturated subtraction* are also used.

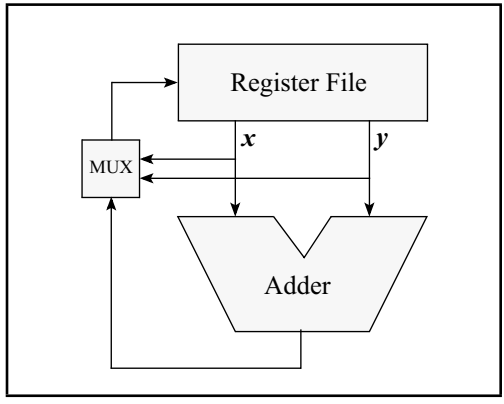


FIGURE 2–3. Implementing max(***x***, ***y***) and min(***x***, ***y***).

(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.

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

Signed	Unsigned
$\max(x, y) = y + \mathrm{doz}(x, y)$	$\mathrm{maxu}(x, y) = y + \mathrm{dozu}(x, y)$
$\min(x, y) = x - \mathrm{doz}(x, y)$	$\mathrm{minu}(x, y) = x - \mathrm{dozu}(x, y)$

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, $z \leftarrow \mathrm{doz}(x, y)$ can be calculated as follows (r0 is a permanent zero register):

4. A destructive operation is one that overwrites one or more of its arguments.

```
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"):

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

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

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

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 $\mathrm{carry}(x - y) - 1$ can be computed in one more instruction after the subtraction of y from x. For example, on the Intel x86 machines, $\mathrm{minu}(x, y)$ 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 $\mathrm{dozu}(x, y)$ 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:

$$\begin{aligned}
&\boldsymbol{d} \leftarrow \boldsymbol{x} - \boldsymbol{y}\\
&\mathrm{doz}(\boldsymbol{x}, \boldsymbol{y}) \;=\; \boldsymbol{d} \;\&\; [(\boldsymbol{d} \equiv ((\boldsymbol{x} \oplus \boldsymbol{y}) \;\&\; (\boldsymbol{d} \oplus \boldsymbol{x}))) \overset{s}{\gg} 31]\\
&\mathrm{dozu}(\boldsymbol{x}, \boldsymbol{y}) \;=\; \boldsymbol{d} \;\&\; \neg[((\neg\boldsymbol{x} \;\&\; \boldsymbol{y}) \mid ((\boldsymbol{x} \equiv \boldsymbol{y}) \;\&\; \boldsymbol{d})) \overset{s}{\gg} 31]
\end{aligned}$$

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, $0 \le x, y \le 2^{31} - 1$.

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

Some uses of the *difference or zero* instruction are given here. In these, the result of $\mathrm{doz}(\boldsymbol{x}, \boldsymbol{y})$ 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]:

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

   Corollary: $|\boldsymbol{x}| \;=\; \mathrm{doz}(\boldsymbol{x}, \boldsymbol{0}) + \mathrm{doz}(\boldsymbol{0}, \boldsymbol{x})$ (other three-instruction solutions are given on page 18).
3. To clamp the upper limit of the true sum of unsigned integers $\boldsymbol{x}$ and $\boldsymbol{y}$ to the maximum positive number $(2^{32} - 1)$ [Knu7]:

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

4. Some comparison predicates (four instructions each):

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

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

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

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]:

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

This works well on a two-address machine. The trick also works if $\oplus$ is replaced by the $\equiv$ 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):

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

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:

$$\begin{aligned} x' &\leftarrow (x \,\&\, \overline{m}) \mid (y \,\&\, m) \\ y &\leftarrow (y \,\&\, \overline{m}) \mid (x \,\&\, m) \\ x &\leftarrow x' \end{aligned}$$

By using "temporaries" for the four *and* expressions, this can be seen to require seven instructions, assuming that either m or $\overline{m}$ 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.

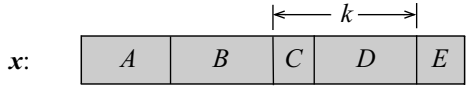
(a)	(b)	(c)
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$t \leftarrow (x \oplus y) \,\&\, m$
$y \leftarrow y \oplus (x \,\&\, m)$	$y \leftarrow y \equiv (x \mid \overline{m})$	$x \leftarrow x \oplus t$
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$y \leftarrow y \oplus t$

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 $\overline{m}$ 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:

$$\begin{aligned} \boldsymbol{t}_1 &= (\boldsymbol{x} \,\&\, \boldsymbol{m}) \ll k \\ \boldsymbol{t}_2 &= (\boldsymbol{x} \overset{u}{\gg} k) \,\&\, \boldsymbol{m} \\ \boldsymbol{x}' &= (\boldsymbol{x} \,\&\, \boldsymbol{m}') \mid \boldsymbol{t}_1 \mid \boldsymbol{t}_2 \end{aligned}$$

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*.

$$\begin{aligned} \boldsymbol{t}_1 &= [\boldsymbol{x} \oplus (\boldsymbol{x} \overset{u}{\gg} k)] \,\&\, \boldsymbol{m} \\ \boldsymbol{t}_2 &= \boldsymbol{t}_1 \ll k \\ \boldsymbol{x}' &= \boldsymbol{x} \oplus \boldsymbol{t}_1 \oplus \boldsymbol{t}_2 \end{aligned}$$

The idea is that $\boldsymbol{t}_1$ contains $B \oplus D$ in position *D* (and 0's elsewhere), and $\boldsymbol{t}_2$ contains $B \oplus D$ 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

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

If $\boldsymbol{a}$ and $\boldsymbol{b}$ are constants, these require only one or two basic RISC instructions. Of course, overflow in calculating $\boldsymbol{a} + \boldsymbol{b}$ 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,

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

(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

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

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 $\boldsymbol{a}-\boldsymbol{c}$ and $\boldsymbol{b}-\boldsymbol{c}$, and then using [GLS1]:

$$f(\boldsymbol{x}) = ((-(\boldsymbol{x}=\boldsymbol{c}))\ \&\ (\boldsymbol{a}-\boldsymbol{c})) + ((-(\boldsymbol{x}=\boldsymbol{a}))\ \&\ (\boldsymbol{b}-\boldsymbol{c})) + \boldsymbol{c}, \quad \text{or}$$
$$f(\boldsymbol{x}) = ((-(\boldsymbol{x}=\boldsymbol{c}))\ \&\ (\boldsymbol{a}\oplus\boldsymbol{c}))\oplus((-(\boldsymbol{x}=\boldsymbol{a}))\ \&\ (\boldsymbol{b}\oplus\boldsymbol{c}))\oplus\boldsymbol{c}.$$

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 $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are distinct, there are two bit positions, n_1 and n_2, where the bits of $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are not all the same, and where the "odd one out" (the one

---

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.

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.

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

Without loss of generality, rename $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ so that $\boldsymbol{a}$ has the odd one out in position n_1 and $\boldsymbol{b}$ 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 $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (0, 1, 1)$ or $(1, 0, 0)$. Similarly, there are two possibilities for the bits at position n_2, namely $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (0, 1, 0)$ or $(1, 0, 1)$. This makes four cases in all, and formulas for each of these cases are shown below.

Case 1. $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (0, 1, 1)$, $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (0, 1, 0)$:

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

Case 2. $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (0, 1, 1)$, $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (1, 0, 1)$:

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

Case 3. $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (1, 0, 0)$, $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (0, 1, 0)$:

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

Case 4. $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (1, 0, 0)$, $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (1, 0, 1)$:

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

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.

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

Because all variables except $\boldsymbol{x}$ 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*, $\oplus$ for *exclusive or*, and either an overbar or a prefix $\neg$ 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 $g(x, y) \oplus zh(x, y)$, 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 $(a \oplus b)c = ac \oplus bc$.

---

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.

This is in the required form with $g(x, y) = f_0(x, y)$ and $h(x, y) = f_0(x, y) \oplus f_1(x, y)$. $f_0(x, y)$, 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 $g(x_1, x_2, \ldots, x_{n-1}) \oplus x_n h(x_1, x_2, \ldots, x_{n-1})$. 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 $2^{2^5} = 4{,}294{,}967{,}296$ 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

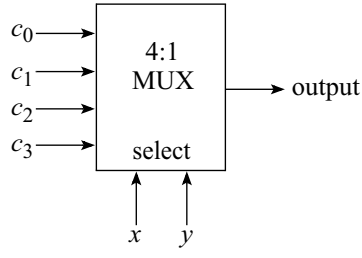
The instruction sets of some computers include all 16 binary Boolean operations. Many of the instructions are useless in that their function can be accomplished with another instruction. For example, the function $f(x, y) = 0$ simply clears a register, and most computers have a variety of ways to do 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

7. The entire 335-page work is available at www.gutenberg.org/etext/15114.

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 $\boldsymbol{x} \neq \mathbf{0}$, where the final assignment to $\boldsymbol{y}$ is the result:

$$\begin{aligned}
&\boldsymbol{y} \leftarrow \boldsymbol{x} + (\boldsymbol{x} \mathbin{\&} -\boldsymbol{x})\\
&\boldsymbol{x} \leftarrow \boldsymbol{x} \mathbin{\&} \neg\boldsymbol{y}\\
&\text{while}((\boldsymbol{x} \mathbin{\&} \mathbf{1}) = \mathbf{0})\ \boldsymbol{x} \leftarrow \boldsymbol{x} \overset{s}{\gg} 1\\
&\boldsymbol{x} \leftarrow \boldsymbol{x} \overset{s}{\gg} 1\\
&\boldsymbol{y} \leftarrow \boldsymbol{y} \mid \boldsymbol{x}
\end{aligned}$$

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 $\boldsymbol{x}$ and $\boldsymbol{y}$, k the number of 1-bits in the strings, and assume the code is executed for all values of $\boldsymbol{x}$ 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 $\boldsymbol{x} \ll (\boldsymbol{x} \mathbin{\&} \mathbf{1})$ [Knu8]. This is a left shift by a variable amount, but it can be computed by

$$\begin{aligned}
&\boldsymbol{x} + (\boldsymbol{x} \mathbin{\&} \mathbf{1})*\boldsymbol{x}, \quad \text{or}\\
&\boldsymbol{x} + (\boldsymbol{x} \mathbin{\&} (-(\boldsymbol{x} \mathbin{\&} \mathbf{1}))),
\end{aligned}$$

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,

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

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

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

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

9. Show how to implement $\boldsymbol{x} \oplus \boldsymbol{y}$ in four instructions with *and-or-not* logic.

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

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

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.

INDEX

C

D

K

L

M

S

T

U