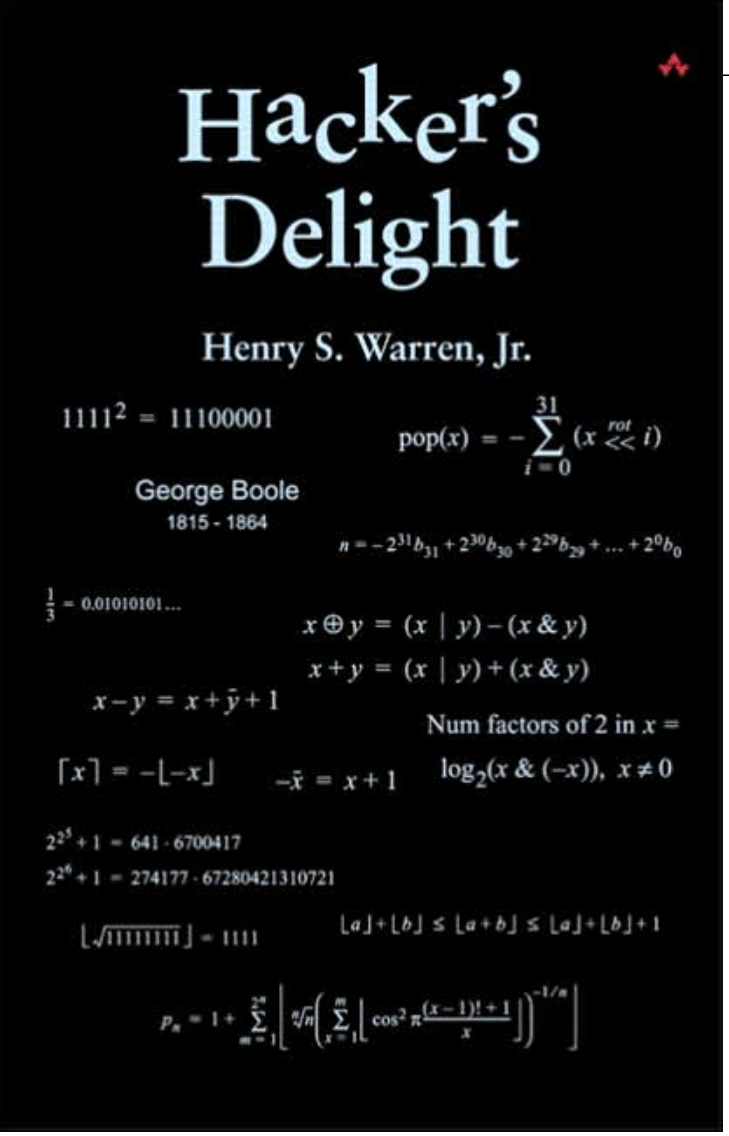
Hacker's Delight
Henry S. Warren, Jr.
$1111^2 = 11100001$
$\text{pop}(x) = -\sum_{i=0}^{31} (x \overset{rot}{\ll} i)$
George Boole
1815 - 1864
$n = -2^{31}b_{31} + 2^{30}b_{30} + 2^{29}b_{29} + \ldots + 2^0 b_0$
$\frac{1}{3} = 0.01010101\ldots$
$x \oplus y = (x \mid y) - (x \,\&\, y)$
$x + y = (x \mid y) + (x \,\&\, y)$
$x - y = x + \bar{y} + 1$
Num factors of 2 in $x =$
$\log_2(x \,\&\, (-x)),\ x \neq 0$
$\lceil x \rceil = -\lfloor -x \rfloor$
$-\bar{x} = x + 1$
$2^{2^5} + 1 = 641 \cdot 6700417$
$2^{2^6} + 1 = 274177 \cdot 67280421310721$
$\lfloor \sqrt{11111111} \rfloor = 1111$
$\lfloor a \rfloor + \lfloor b \rfloor \leq \lfloor a + b \rfloor \leq \lfloor a \rfloor + \lfloor b \rfloor + 1$
$p_n = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{n} \left(\sum_{x=1}^{m} \left\lfloor \cos^2 \pi \frac{(x-1)! + 1}{x} \right\rfloor \right)^{-1/n} \right\rfloor$

Hacker's Delight

By Henry S. Warren,

Publisher: Addison Wesley
Pub Date: July 17, 2002
ISBN: 0-201-91465-4
Pages: 320

Ripped by Caudex 2003

"This is the first book that promises to tell the deep, dark secrets of computer arithmetic, and it delivers in spades. It contains every trick I knew plus many, many more. A godsend for library developers, compiler writers, and lovers of elegant hacks, it deserves a spot on your shelf right next to Knuth."-Josh Bloch

"When I first saw the title, I figured that the book must be either a cookbook for breaking into computers (unlikely) or some sort of compendium of little programming tricks. It's the latter, but it's thorough, almost encyclopedic, in its coverage." -Guy Steele

These are the timesaving techniques relished by computer hackers-those devoted and persistent code developers who seek elegant and efficient ways to build better software. The truth is that much of the computer programmer's job involves a healthy mix of arithmetic and logic. In *Hacker's Delight*, veteran programmer Hank Warren shares the tricks he has collected from his considerable experience in the worlds of application and system programming. Most of these techniques are eminently practical, but a few are included just because they are interesting and unexpected. The resulting work is an irresistible collection that will help even the most seasoned programmers better their craft.

Topics covered include:

- A broad collection of useful programming tricks
- Small algorithms for common tasks
- Power-of-2 boundaries and bounds checking
- Rearranging bits and bytes
- Integer division and division by constants
- Some elementary functions on integers

- Gray code
- Hilbert's space-filling curve
- And even formulas for prime numbers!

This book is for anyone who wants to create efficient code. Hacker's Delight will help you learn to program at a higher level-well beyond what is generally taught in schools and training courses-and will advance you substantially further than is possible through ordinary self-study alone.

Copyright

Library of Congress Cataloging-in-Publication Data

Warren, Henry S.

Hacker's delight / Henry S. Warren, Jr.

p. cm.

Includes bibliographical references and index.

1. Computer programming. I. Title.

QA76.6 .W375 2002

005.1—dc21

2002066501

Text printed on recycled paper

1 2 3 4 5 6 7 8 9 10—MA—0605040302

First printing, July 2002

Dedication

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

Foreword

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 whose pages 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.

[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.

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

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

These techniques are just as useful in the C or Java programming languages as they are in assembly language.

Many books on algorithms and data structures teach complicated techniques for sorting and searching, for maintaining hash tables and binary trees, for dealing with records and pointers. They overlook what can be done with very tiny pieces of data—bits and arrays of bits. It is amazing what can be done with just binary addition and subtraction and maybe some bitwise operations; the fact that the carry chain allows a single bit to affect all the bits to its left makes addition a peculiarly powerful data manipulation operation in ways that are

not widely appreciated.

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

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

Preface

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

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

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

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

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

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

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

Occasionally, machine language is used. It employs 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. This is because on many computers, branches slow down instruction fetching and inhibit executing instructions in parallel. Another problem with branches is that they may 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.

[1] The shortest such program written in C, known to the present author, is by Vlad Taeerov and Rashit Fakhreyev and is 64 characters in length:

```
main(a){printf(a,34,a="main(a){printf(a,34,a=%c%s%c,34);}",34);}
```

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. But 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 that will have to wait for a second edition (). 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
February 2002

Chapter 1. Introduction

1-1 Notation

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

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

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

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

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

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

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

Longer or more complex computer programs are written in the C++ programming language. None of the object-

oriented features of C++ are used; the programs are basically in C with comments in C++ style. When the distinction is unimportant, the language is referred to simply as "C."

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

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

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

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

Table 1-1. Expressions of C and Computer Algebra

Precedence	C	Computer Algebra	Description
	`0x`…	**0x…, 0b…**	Hexadecimal, binary constants
16	`a[k]`		Selecting the *k*th component
16		$\mathbf{x}_0, \mathbf{x}_1, \ldots$	Different variables, or bit selection (clarified in text)
16	`f(x,…)`	$f(\boldsymbol{x}, \ldots)$	Function evaluation
16		abs($\boldsymbol{x}$)	Absolute value (but abs($-\mathbf{2}^{\mathbf{31}}$) = $-\mathbf{2}^{\mathbf{31}}$)
16		nabs($\boldsymbol{x}$)	Negative of the absolute value
15	`x++, x--`		Postincrement, decrement
14	`++x, --x`		Preincrement, decrement

14	`(`*type name*`)x`		Type conversion
14 R		x^k	***x*** to the *k*th power
14	`~x`	$\neg x$, $\overline{x}$	Bitwise *not* (one's-complement)
14	`!x`		Logical *not* (if ***x*** = **0** then **1** else **0**)
14	`-x`	$-x$	Arithmetic negation
13 L	`x*y`	$x * y$	Multiplication, modulo word size
13 L	`x/y`	$x \div y$	Signed integer division
13 L	`x/y`	$x \overset{u}{\div} y$,	Unsigned integer division
13 L	`x%y`	rem(***x, y***)	Remainder (may be negative), of ($x \div y$) signed arguments
13 L	`x%y`	rem(***x, y***)	Remainder of $x \overset{u}{\div} y$, unsigned arguments
		mod(***x, y***)	***x*** reduced modulo ***y*** to the interval [**0,** abs(***y***) - **1**]; signed arguments
12 L	`x + y, x - y`	$x + y$, $x - y$	Addition, subtraction
11 L	`x << y, x >> y`	$x \ll y$, $x \overset{u}{\gg} y$	Shift left, right with 0-fill ("logical" shifts)
11 L	`x >> y`	$x \overset{s}{\gg} y$	Shift right with sign-fill ("arithmetic" or "algebraic" shift)
11 L		$x \overset{rot}{\ll} y$, $x \overset{rot}{\gg} y$	Rotate shift left, right

10 L	`x < y, x <= y,` `x > y, x >= y`	$x < y$, $x \leq y$, $x > y$, $x \geq y$,	Signed comparison
10 L	`x < y, x <= y,` `x > y, x >= y`	$x \overset{u}{<} y$, $x \overset{u}{\leq} y$, $x \overset{u}{>} y$, $x \overset{u}{\geq} y$	Unsigned comparison
9 L	`x == y, x != y`	$x = y$, $x \neq y$	Equality, inequality
8 L	`x & y`	$x \,\&\, y$	Bitwise *and*
7 L	`x ^ y`	$x \oplus y$	Bitwise *exclusive or*
7 L		$x \equiv y$	Bitwise *equivalence* ($\neg(x \oplus y)$)
6 L	`x | y`	x | y	Bitwise *or*
5 L	`x && y`	$x \vec{\&} y$	Conditional *and* (if $x = 0$ then **0** else if $y = 0$ then **0** else **1**)
4 L	`x || y`	$x \vec{\mid} y$	Conditional *or* (if $x \neq 0$ then **1** else if $y \neq 0$ then **1** else **0**)
3 L		$x \,\|\|\, y$	Concatenation
2 R	`x = y`	$x \leftarrow y$	Assignment

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

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

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

`while` (*expression*) *statement*

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

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

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

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

The `for` statement is written:

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

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

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

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

1-2 Instruction Set and Execution Time Model

To permit a rough comparison of algorithms, we imagine them being coded for a machine with an instruction set similar to that of today's general purpose RISC computers, such as the Compaq Alpha, the SGI MIPS, and the IBM RS/6000. The machine is three-address and has a fairly large number of general purpose registers—that is, 16 or more. Unless otherwise specified, the registers are 32 bits long. General register 0 contains a permanent 0, and the others can be used uniformly for any purpose.

In the interest of simplicity there are no "special purpose" registers, such as a condition register or a register to hold status bits, such as "overflow." No floating-point operations are described, because that is beyond the scope of this book.

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

Table 1-2. Basic RISC Instruction Set

Opcode Mnemonic	Operands	Description
`add, sub, mul, div, divu, rem, remu`	`RT,RA,RB`	`RT` ← `RA op RB`, where `op` is *add*, *subtract*, *multiply*, *divide signed*, *divide unsigned*, *remainder signed*, or *remainder unsigned*.
`addi, muli`	`RT,RA,I`	`RT` ← `RA op I`, where `op` is *add* or *multiply*, and `I` is a 16-bit signed immediate value.
`addis`	`RT,RA,I`	`RT` ← `RA + (I || 0x0000)`.
`and, or, xor`	`RT,RA,RB`	`RT` ← `RA op RB`, where `op` is bitwise *and*, *or*, or *exclusive or*.
`andi, ori, xori`	`RT,RA,Iu`	As above, except the last operand is a 16-bit unsigned immediate value.

beq, bne, blt, ble, bgt, bge	RT,target	Branch to target if RT = 0, or if RT ≠ 0, or if RT < 0, or if RT ≤ 0, or if RT > 0, or if RT ≥ 0 (signed integer interpretation of RT).
bt, bf	RT,target	Branch true/false; same as bne/beq resp.
cmpeq, cmpne, cmplt, cmple, cmpgt, cmpge, cmpltu, cmpleu, cmpgtu, cmpgeu	RT,RA,RB	RT gets the result of comparing RA with RB; 0 if **false** and 1 if **true**. Mnemonics denote *compare for equality*, *inequality*, *less than*, and so on, as for the branch instructions, and in addition, the suffix "u" denotes an unsigned comparison.
cmpieq, cmpine, cmpilt, cmpile, cmpigt, cmpige	RT,RA,I	Like cmpeq, and so on, except the second comparand is a 16-bit signed immediate value.
cmpiequ, cmpineu, cmpiltu, cmpileu, cmpigtu, cmpigeu	RT,RA,Iu	Like cmpltu, and so on, except the second comparand is a 16-bit unsigned immediate value.
ldbu, ldh, ldhu, ldw	RT,d(RA)	Load an unsigned byte, signed halfword, unsigned halfword, or word into RT from memory at location RA + d, where d is a 16-bit signed immediate value.
mulhs, mulhu	RT,RA,RB	RT gets the high-order 32 bits of the product of RA and RB; signed and unsigned.
not	RT,RA	RT ⟵ bitwise one's-complement of RA.
shl, shr, shrs	RT,RA,RB	RT ⟵ RA shifted left or right by the amount given in the rightmost six bits of RB; 0-fill except for shrs, which is sign-fill. (The shift amount is treated modulo 64.)
shli, shri, shrsi	RT,RA,Iu	RT ⟵ RA shifted left or right by the amount given in the 5-bit immediate field.

stb, sth, stw	RS,d(RA)	Store a byte, halfword, or word, from RS into memory at location RA + d, where d is a 16-bit signed immediate value.

In these brief instruction descriptions, RA and RB appearing as source operands really means the contents of those registers.

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

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

Table 1-3. Additional Instructions for the "Full RISC"

Opcode Mnemonic	Operands	Description
abs, nabs	RT,RA	RT gets the absolute value, or the negative of the absolute value, of RA.
andc, eqv, nand, nor, orc	RT,RA,RB	Bitwise *and with complement* (of RB), *equivalence*, *negative and*, *negative or*, and *or with complement*.
extr	RT,RA,I,L	Extract bits I through I+L-1 of RA, and place them right-adjusted in RT, with 0-fill.
extrs	RT,RA,I,L	Like extr, but sign-fill.
ins	RT,RA,I,L	Insert bits 0 through L-1 of RA into bits I through I+L-1 of RT.
nlz	RT,RA	RT gets the number of leading 0's in RA (0 to 32).

`pop`	`RT,RA`	`RT` gets the number of 1-bits in `RA` (0 to 32).
`ldb`	`RT,d(RA)`	Load a signed byte into `RT` from memory at location `RA + d`, where `d` is a 16-bit signed immediate value.
`moveq, movne, movlt, movle, movgt, movge`	`RT,RA,RB`	`RT` ⟵ `RB` if `RA` = 0, or if `RA` ≠ 0, and so on, else `RT` is unchanged.
`shlr, shrr`	`RT,RA,RB`	`RT` ⟵ `RA` rotate-shifted left or right by the amount given in the rightmost five bits of `RB`.
`shlri, shrri`	`RT,RA,Iu`	`RT` ⟵ `RA` rotate-shifted left or right by the amount given in the 5-bit immediate field.
`trpeq, trpne, trplt, trple, trpgt, trpge, trpltu, trpleu, trpgtu, trpgeu`	`RA,RB`	*Trap* (interrupt) if `RA` = `RB`, or `RA` ≠ `RB`, and so on.
`trpieq, trpine, trpilt, trpile, trpigt, trpige`	`RA,I`	Like `trpeq`, and so on, except the second comparand is a 16-bit signed immediate value
`trpigtu, trpigeu trpiequ, trpineu, trpiltu, trpileu,`	`RA,Iu`	Like `trpltu`, and so on, except the second comparand is a 16-bit unsigned immediate value.

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

Table 1-4. Extended Mnemonics

Extended Mnemonic	Expansion	Description
`b target`	`beq R0,target`	*Unconditional branch.*
`li RT,I`	See text	*Load immediate*, $-2^{31} \le I < 2^{32}$.

`mov RT,RA`	`ori RT,RA,0`	*Move register* `RA` to `RT`.
`neg RT,RA`	`sub RT,R0,RA`	*Negate* (two's-complement).
`subi RT,RA,I`	`addi RT,RA,-I`	*Subtract immediate* (`I` $\neq -2^{15}$).

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

Of course, which instructions belong in the basic RISC, and which belong in the full RISC is very much a matter of judgment. Quite possibly, *divide unsigned* and the *remainder* instructions should be moved to the full RISC category. *Shift right signed* is another suspicious instruction, given its low frequency of use in the SPEC benchmarks. The trouble is, in C it is easy to accidentally use these instructions, by doing a division with unsigned operands when they could just as well be signed, and by doing a shift right with a signed quantity (`int`) that could just as well be unsigned. Incidentally, *shift right signed* (or *shift right arithmetic*, as it is often called) does *not* do a division of a signed integer by a power of 2; you need to add 1 to the result if the dividend is negative and any nonzero bits are shifted out.

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

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

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

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

Execution Time

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

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

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

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

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

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

Chapter 2. Basics