2ND EDITION

Computer Arithmetic Algorithms

Israel Koren

COMPUTER ARITHMETIC ALGORITHMS

Second Edition

COMPUTER ARITHMETIC ALGORITHMS

Second Edition

Israel Koren
University of Massachusetts, Amherst

A K Peters
Natick, Massachusetts

Editorial, Sales, and Customer Service Office

A K Peters, Ltd.
63 South Avenue
Natick, MA 01760
www.akpeters.com

Library of Congress Cataloging-in-Publication data

Koren, Israel, 1945-
Computer arithmetic algorithms / Israel Koren.–2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 1-56881-160-8
1. Computer arithmetic. 2. Computer algorithms. I. Title

QA76.9.C62 K67 2001
005.1–dc21

2001045837

Figures 4.3, 4.4, 4.5 and 4.7 are reprinted from D.J. Kuck, *The Structure of Computers and Computations*, Vol. 1 (copyright © 1978 John Wiley) by permission of John Wiley & Sons, Inc. Table 2.2 and Figures 5.32, 6.11, 6.13, 6.20–6.23, 7.5, 7.7, 7.13, and 10.2 are reprinted by permission of the Institute of Electrical and Electronics Engineers (IEEE). Table 4.3 is represented by permission of the IBM Systems Journal.

The first edition of this book was published by Prentice-Hall Inc.

This book is dedicated to my wife, Zahava,
my sons, Yuval and Yaron,
and to the memory of my parents,
Jacob and Dvora.

CONTENTS

FORWORD TO THE SECOND EDITION xi

PREFACE xiii

1 CONVENTIONAL NUMBER SYSTEMS 1

1.1 The Binary Number System 1
1.2 Machine Representations of Numbers 2
1.3 Radix Conversions 4
1.4 Representations of Negative Numbers 6
1.5 Addition and Subtraction 13
1.6 Arithmetic Shift Operations 15
1.7 Exercises 16
1.8 References 17

2 UNCONVENTIONAL FIXED-RADIX NUMBER SYSTEMS 19

2.1 Negative-Radix Number Systems 19
2.2 A General Class of Fixed-Radix Number Systems 21

2.3 Signed-Digit Number Systems 23
2.4 Binary *SD* Numbers 27
2.5 Exercises 32
2.6 References 33

3 SEQUENTIAL ALGORITHMS FOR MULTIPLICATION AND DIVISION 35

3.1 Sequential Multiplication 35
3.2 Sequential Division 39
3.3 Nonrestoring Division 42
3.4 Square Root Extraction 48
3.5 Exercises 50
3.6 References 52

4 BINARY FLOATING-POINT NUMBERS 53

4.1 Preliminaries 53
4.2 Floating-Point Operations 59
4.3 Choice of Floating-Point Representation 65
4.4 The IEEE Floating-Point Standard 67
4.5 Round-off Schemes 71
4.6 Guard Digits 76
4.7 Floating-Point Adders 81
4.8 Exceptions 84
4.9 Round-off Errors and Their Accumulation 87
4.10 Exercises 89
4.11 References 91

5 FAST ADDITION 93

5.1 Ripple-Carry Adders 93
5.2 Carry-Look-Ahead Adders 95
5.3 Conditional Sum Adders 99
5.4 Optimality of Algorithms and Their Implementations 102
5.5 Carry-Look-Ahead Addition Revisited 106
5.6 Prefix Adders 109

5.7 Ling Adders 110
5.8 Carry-Select Adders 113
5.9 Carry-Skip Adders 116
5.10 Hybrid Adders 119
5.11 Carry-Save Adders 124
5.12 Pipelining of Arithmetic Operations 132
5.13 Exercises 135
5.14 References 138

6 HIGH-SPEED MULTIPLICATION 141

6.1 Reducing the Number of Partial Products 141
6.2 Implementing Large Multipliers Using Smaller Ones 149
6.3 Accumulating the Partial Products 145
6.4 Alternative Techniques for Partial Product Accumulation 157
6.5 Fused Multiply-Add Unit 165
6.6 Array Multipliers 167
6.7 Optimality of Multiplier Implementations 174
6.8 Exercises 176
6.9 References 179

7 FAST DIVISION 181

7.1 SRT Division 181
7.2 High-Radix Division 187
7.3 Speeding Up the Division Process 198
7.4 Array Dividers 203
7.5 Fast Square Root Extraction 206
7.6 Exercises 209
7.7 References 210

8 DIVISION THROUGH MULTIPLICATION 213

8.1 Division by Convergence 213
8.2 Division by Reciprocation 218
8.3 Exercises 222
8.4 References 223

9 EVALUATION OF ELEMENTARY FUNCTIONS 225

9.1 The Exponential Function 226
9.2 The Logarithm Function 229
9.3 The Trigonometric Functions 232
9.4 The Inverse Trigonometric Functions 235
9.5 The Hyperbolic Functions 238
9.6 Bounds on the Approximation Error 239
9.7 Speed-up Techniques 241
9.8 Other Techniques for Evaluating Elementary Functions 243
9.9 Exercises 244
9.10 References 245

10 LOGARITHMIC NUMBER SYSTEMS 247

10.1 Sign-Logarithm Number Systems 247
10.2 Arithmetic Operations 249
10.3 Comparison to Binary Floating-Point Numbers 252
10.4 Conversions to/from Conventional Representations 253
10.5 Exercises 255
10.6 References 256

11 THE RESIDUE NUMBER SYSTEM 259

11.1 Preliminaries 259
11.2 Arithmetic Operations 261
11.3 The Associated Mixed-Radix System 264
11.4 Conversion of Numbers from/to the Residue System 266
11.5 Selecting the Moduli 267
11.6 Error Detection and Correction 269
11.7 Exercises 274
11.8 References 275

INDEX 277

FORWORD TO THE SECOND EDITION

This edition includes several new sections as well as many amendments and corrections made since the first edition in 1993. The new sections include floating-point adders, floating-point exceptions, general carry-look-ahead adders, prefix adders, Ling adders, and fused multiply-add units. New algorithms and implementations have been added to almost all chapters. My thanks to the students and readers whose discoveries of errors, and general comments, are reflected in this volume.

Since the first edition, a web page was created for *Computer Arithmetic Algorithms*, which contains updates on the book, solutions to selected problems and relevant links. It can be found at http://www.ecs.umass.edu/ece/koren/arith.

Additionally, there is now an on-line, JavaScript-based simulator for many of the algorithms contained in this book. A useful tool for both students and practitioners in the field, the Java-based simulator can be found at http://www.ecs.umass.edu/ece/koren/arith/simulator.

I very much welcome further comments and suggestions; please e-mail them to me at koren@ecs.umass.edu.

Israel Koren
Amherst, Massachusetts
July 2001

PREFACE

The goal of this textbook is to explain the fundamental principles of algorithms available for performing arithmetic operations in digital computers. These include basic arithmetic operations like addition, subtraction, multiplication, and division in fixed-point and floating-point number systems, and more complex operations, such as square root extraction and evaluation of exponential, logarithmic, and trigonometric functions, and so on. Even the seemingly simple arithmetic operations turn out to be more complex than one expects when attempting to implement them. The descriptions found in many excellent books on computer architecture do not provide the level of detail required, and there is a need for a book that is completely devoted to digital computer arithmetic.

Designs that include arithmetic units have proliferated in recent years. The progress already made in integrated circuit technology, and further advances expected in the near future, has had a significant impact on the design of new arithmetic processors. The higher density of integrated circuits now enables the design and implementation of sophisticated arithmetic processors employing algorithms that were considered prohibitively complex in the past. Consequently, methods that were previously considered unconventional should be examined, since they may now be attractive alternatives. Furthermore, the ease of designing application-specific integrated circuits currently allows engineers to design arithmetic units tailored to their special needs, rather than having to use general-purpose circuits. These specialized arithmetic units can achieve a higher speed of operation for the particular application being considered.

This book describes the principles of computer arithmetic algorithms independently of any particular technology employed for their implementation. The existence of several implementation technologies and the rapid changes in these make a detailed description of any implementation almost immediately obsolete. The book includes numerical examples to illustrate the working of the algorithms presented and explains the concepts behind the algorithms without relying on gate diagrams. Such diagrams are usually straightforward, and any reader with a sufficiently good background in digital design, as is expected from the readers of this book, can draw his/her own. These diagrams are in many cases useless for the practitioner, since the technology that he/she plans to use imposes its constraints on the implementation, and, more importantly, they do not provide any additional insight.

All the algorithms in this book are described within the same framework, so that the similarities between different algorithms become evident, and, consequently, the basic principles behind these algorithms can be easily identified. This should help the reader to better understand the currently available algorithms, know how to select the most appropriate algorithm to match a given technology, and even be able to develop new algorithms if the need arises.

This book is intended to be used as a textbook in a senior-level or first-year graduate-level course in computer arithmetic and as a reference book for practicing engineers. The reader's expected background includes a basic knowledge of digital design and the principles of digital computer organization. The book includes 11 chapters; each chapter has a list of relevant references and a set of exercises. A separate solution manual is available from the publisher upon request.

This book does not include all the algorithms that have ever been suggested, and is meant only to serve as a solid introduction to this rich field, which is continuously evolving. Readers who are interested in further details on a particular topic should consult the list of references at the end of each chapter. Excellent sources for additional information are the proceedings of the biannual IEEE Symposium on Computer Arithmetic and periodicals such as the IEEE Journal on Solid-State Circuits or the IEEE Transactions on Computers. The latter had several special issues devoted to computer arithmetic. Other sources are the several books on computer arithmetic already available, which are listed at the end of Chapter 1.

This textbook evolved from lecture notes prepared for courses in computer arithmetic that I have taught at the University of California at Santa Barbara, the University of Southern California, the Technion in Israel, the University of California at Berkeley, and the University of Massachusetts at Amherst. The order of topics in the book follows their order in my lectures. However, several instructors who used a preliminary version of the book, did not experience any difficulties when covering the chapters out of order.

Many people have contributed in different ways to this book. Prof. James Howard from the University of California at Santa Barbara was the first to suggest the writing of this book. Prof. William Kahan from the University of California at Berkeley, with whom I had long discussions in 1983, influenced my views on many topics. Moshe Gavrielov from LSI Logic read much of the book, and his many suggestions helped to improve the presentation. Prof. Mary Jane Irwin from the Pennsylvania State University and Prof. Behrooz Parhami from the University of California at Santa Barbara agreed to use a preliminary version of this book in their class and provided many helpful comments and suggestions.

Several others reviewed parts of the book and gave me very important feedback. These include Prof. Dan Atkins from the University of Michigan at Ann Arbor, Prof. Milos Ercegovac from the University of California at Los Angeles, Prof. Earl Swartzlander, Tom Callaway, and Michael Schulte from the University of Texas at Austin, and Dr. George Taylor from Sun Microsystems.

The graduate students who took my course in the previously mentioned campuses made many contributions through their questions and suggestions. In particular, I wish to acknowledge the contributions made by Sachin Ghanekar, who prepared the solution manual, and by Ofra Zinaty, Moshe Gavrielov, and Susan Morin.

Last but not least, I wish to thank my wife, Zahava and my sons, Yuval and Yaron, who provided moral support as well as editorial assistance.

Israel Koren
Amherst, Massachusetts

1

CONVENTIONAL NUMBER SYSTEMS

1.1 THE BINARY NUMBER SYSTEM

In conventional digital computers, integers are represented as binary numbers of fixed length. A binary number of length n is an ordered sequence

$$(x_{n-1}, x_{n-2}, \cdots, x_1, x_0)$$

of binary digits where each digit x_i (also known as a *bit*) can assume one of the values 0 or 1. The length n of the sequence is of significance, since binary numbers in digital computers are stored in registers of a fixed length, n. The above sequence of n digits (or n-tuple) represents the integer value

$$X = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2 + x_0 = \sum_{i=0}^{n-1} x_i 2^i. \qquad (1.1)$$

Upper case letters are used in this book to represent numerical values or sequences of digits while lower case letters, usually indexed, represent individual digits. The weight of the digit x_i in (1.1) is the ith power of 2, which is called the *radix* of the number system. The interpretation rule in Equation (1.1) is similar to the rule used for the ordinary decimal numbers. There are, however, two differences between these interpretation rules. First, the radix 10 is used instead of 2 in Equation (1.1) and consequently, the allowed digits in the decimal case are $x_i \in \{0, 1, 2, \cdots, 9\}$ instead of $x_i \in \{0, 1\}$. We call the decimal numbers radix-10

numbers and the binary numbers radix-2 numbers. We indicate the radix to be used when interpreting a given sequence of digits by writing it as a subscript. Thus, the sequence $(101)_{10}$ represents the decimal value 101, while the sequence $(101)_2$ represents the decimal value 5.

Since operands and results in an arithmetic unit are stored in registers of a fixed length, there is a finite number of distinct values that can be represented within an arithmetic unit. Let X_{min} and X_{max} denote the smallest and largest representable values, respectively. We say that $[X_{min}, X_{max}]$ is the *range* of the representable numbers. Any arithmetic operation that attempts to produce a result larger then X_{max} (or smaller than X_{min}), will produce an incorrect result. In such cases the arithmetic unit should indicate that the generated result is in error. This is usually called an *overflow* indication.

Example 1.1

If the conventional binary number system is employed to represent unsigned integers using four binary digits (bits) then, $X_{max} = (15)_{10}$ is represented by $(1111)_2$ and $X_{min} = (0)_{10}$ is represented by $(0000)_2$. Increasing X_{max} by 1 results in $(16)_{10} = (10000)_2$ out of which, in a 4-bit representation, only the last four digits are retained, yielding $(0000)_2 = (0)_{10}$. In general, a number X that is not in the range $[X_{min}, X_{max}] = [0, 15]$ is represented by X modulus 16, or X mod 16, which is the remainder when dividing X by 16. Such a situation can arise, for example, when two operands X and Y are added and their sum exceeds X_{max}. In this case, the final result S satisfies $S = (X + Y)$ mod 16. For example,

X		1	0	1	1	11
$+Y$		0	1	1	1	7
	1	0	0	1	0	

Since the final result has to be stored in a 4-bit register, the most significant bit (whose weight is $2^4 = 16$) is discarded, yielding $(0010)_2 = 2 = 18$ mod 16. □

1.2 MACHINE REPRESENTATIONS OF NUMBERS

The conventional binary system is a specific example of a number system that can be used to represent numerical values in an arithmetic unit. A number system is in general defined by the set of values that each digit can assume and by an interpretation rule that defines the mapping between the sequences of digits and their numerical values. We distinguish between conventional number systems like the binary system described in the previous section (or the commonly used

decimal system) and unconventional systems like the signed-digit number system (to be presented in Chapter 2).

The conventional number systems are *nonredundant, weighted,* and *positional* number systems. In a *nonredundant* number system every number has a unique representation; in other words, no two sequences have the same numerical value. The term *weighted* number system means that there is a sequence of weights

$$w_{n-1}, w_{n-2}, \cdots, w_1, w_0$$

that determines the value of the given n-tuple $(x_{n-1}, x_{n-2}, \cdots, x_0)$ by the equation

$$X = \sum_{i=0}^{n-1} x_i w_i. \tag{1.2}$$

Thus, w_i is the weight assigned to the digit in the ith position, x_i. Finally, in a *positional* number system, the weight w_i depends only on the position i of the digit x_i. In the conventional number systems the weight w_i is the ith power of a fixed integer r, which is the *radix* of the number system. In other words, $w_i = r^i$. Therefore, these number systems are also called *fixed-radix* systems. Since the weight assigned to the digit x_i is r^i, this digit has to satisfy $0 \leq x_i \leq r-1$. Otherwise, if $x_i \geq r$ is allowed, then

$$x_i r^i = (x_i - r)r^i + 1 \cdot r^{i+1},$$

resulting in two machine representations for the same value: $(\cdots, x_{i+1}, x_i, \cdots)$ and $(\cdots, x_{i+1}+1, x_i - r, \cdots)$. In other words, allowing $x_i \geq r$ introduces redundancy into the fixed-radix number system.

A sequence of n digits in a register does not necessarily have to represent an integer. We may use such a sequence to represent a mixed number that has a fractional part as well as an integral part. This is done by partitioning the n digits into two sets: k digits in the integral part and m digits in the fractional part, satisfying $k + m = n$. The value of an n-tuple with a radix point between the k most significant digits and the m least significant digits

$$\underbrace{(x_{k-1}x_{k-2}\cdots x_1 x_0}_{\textit{integral part}} \cdot \underbrace{x_{-1}x_{-2}\cdots x_{-m})_r}_{\textit{fractional part}}$$

is

$$\begin{aligned} X &= x_{k-1}r^{k-1} + x_{k-2}r^{k-2} + \cdots + x_1 r + x_0 + x_{-1}r^{-1} + \cdots + x_{-m}r^{-m} \\ &= \sum_{i=-m}^{k-1} x_i r^i. \end{aligned} \tag{1.3}$$

The radix point is not stored in the register but is understood to be in a fixed position between the k most significant digits and the m least significant digits. For this reason we call such representations *fixed-point* representations. A programmer of a digital computer is not necessarily restricted to the use of numbers having the predetermined position of the radix point but can properly scale the operands. As long as the same scaling factor is used for all operands, the add and subtract operations yield the correct results, since $aX \pm aY = a(X \pm Y)$, where a is the scaling factor. However, corrections are required when performing multiplication and division, since $aX \cdot aY = a^2XY$ and $aX/aY = X/Y$. Commonly used positions for the radix point are at the rightmost side of the number (i.e., pure integers, $m = 0$) and at the leftmost side of the number (i.e., pure fractions, $k = 0$).

Given the length n of the operands, the weight r^{-m} of the least significant digit indicates the position of the radix point. To simplify our discussion from this point on and to avoid the need to distinguish between the different partitions of numbers (into fractional and integral parts), we introduce the notion of a *unit in the last position* (*ulp*), which is the weight of the least significant digit,

$$ulp = r^{-m}. \tag{1.4}$$

1.3 RADIX CONVERSIONS

Radix conversion is the translation of a number X represented in one radix number system (to be called the *source* number system) to its representation in another number system (called the *destination* number system). The major reason for such conversions is the fact that most arithmetic units operate on binary numbers while their users are more accustomed to decimal numbers, which also require a smaller number of digits. We will therefore emphasize conversions between the decimal and the binary number systems but will present the algorithms in a more general form.

Given a number X, we wish to find its representation in the destination number system with radix r_D. For convenience, we distinguish between the conversion of the integral part X_I and that of the fractional part X_F. Starting with the integral part, its representation $(x_{k-1}x_{k-2}\cdots x_1x_0)_{r_D}$ is sought. We can rewrite the appropriate part of Equation (1.3) as follows:

$$X_I = \{[\cdots(x_{k-1}r_D + x_{k-2})r_D + \cdots + x_2]\,r_D + x_1\}\,r_D + x_0, \tag{1.5}$$

where $0 \le x_i < r_D$. Thus, if we divide X_I by r_D we obtain x_0 as the remainder and

$$\{[\cdots(x_{k-1}r_D + x_{k-2})r_D + \cdots + x_2]\,r_D + x_1\}$$

as the quotient. If we now divide the above quotient by r_D, we obtain x_1 as the remainder. We may therefore divide the quotients by r_D repeatedly, retaining the remainders as the required digits until a zero quotient is reached.

To find the representation $(x_{-1}x_{-2}\cdots x_{-m})_{r_D}$ of the fractional part X_F, we rewrite the appropriate part of Equation (1.3) as follows:

$$X_F = r_D^{-1}\left\{x_{-1} + r_D^{-1}\left[x_{-2} + r_D^{-1}(x_{-3} + \cdots)\right]\right\}. \tag{1.6}$$

If we multiply X_F by r_D, we obtain a mixed number with x_{-1} as its integral part and

$$r_D^{-1}\left[x_{-2} + r_D^{-1}(x_{-3} + \cdots)\right]$$

as the fractional part. We may therefore multiply the fractional parts by r_D repeatedly, retaining the generated integers as the required digits. However, unlike the algorithm for the integral part, this algorithm is not guaranteed to terminate, since a finite fraction (one that needs a finite number of nonzero digits) in one number system may correspond to an infinite fraction in another. This does not constitute a problem in practice, since the process can be terminated after m steps (or a few additional ones if rounding is desired).

Example 1.2

The decimal mixed number $X = 46.375_{10}$ is to be converted to binary form. Starting with $X_I = 46$ we obtain the following quotients and remainders when repeatedly dividing by 2:

Quotient	Remainder
23	$0 = x_0$
11	$1 = x_1$
5	$1 = x_2$
2	$1 = x_3$
1	$0 = x_4$
0	$1 = x_5$

We now convert the fractional part $X_F = 0.375$ and obtain the following integers and fractions when repeatedly multiplying by 2:

Integer part	Fractional part
$0 = x_{-1}$	.75
$1 = x_{-2}$	.5
$1 = x_{-3}$	.0

Thus, the final result is $46.375_{10} = 101110.011_2$.

If the fractional part of the given decimal number was $X_F = 0.3$, the above algorithm would never terminate, since the decimal fraction 0.3_{10} is the infinite binary fraction $(0.0100110011\cdots)_2$. □

All the arithmetic operations in the above conversion algorithms were performed in the source number system, which was the decimal system. To convert a binary number to the decimal system, we may either execute the above algorithm in the source binary system or, more conveniently, perform the conversion in the destination decimal system using Equation (1.3).

1.4 REPRESENTATIONS OF NEGATIVE NUMBERS

For fixed-point numbers in a radix r system, we have to determine the way negative numbers are represented. Two different forms are commonly used:

1. Sign and magnitude representation, which is also called the signed-magnitude method
2. Complement representation which comprises two alternatives

1. Signed-magnitude: Here the sign and magnitude are represented separately. The first digit is the sign digit and the remaining $(n-1)$ digits represent the magnitude. In the binary case, the sign bit is normally selected to be 0 for positive numbers and 1 for negative ones. In the nonbinary case, the values 0 and $(r-1)$ are assigned to the sign digit of positive and negative numbers, respectively. Notice that in this case only $2 \cdot r^{n-1}$ out of the r^n possible sequences are utilized. This will be discussed further later on.

Let the $(n-1)$ digits representing the magnitude be partitioned into $(k-1)$ and m digits in the integral and fractional parts, respectively. The largest representable value is then

$$X_{max} = \left(r^{k-1} - ulp\right), \qquad \text{where} \quad ulp = r^{-m}$$

and the corresponding representation is $0(r-1)\cdots(r-1)$. Thus, the range of positive numbers is $\left[0, r^{k-1} - ulp\right]$. The range of negative numbers is similarly $\left[-(r^{k-1} - ulp), -0\right]$, represented by $(r-1)(r-1)\cdots(r-1)$ to $(r-1)0\cdots0$. We therefore have two representations for zero, one positive and one negative. This is inconvenient when implementing an arithmetic unit, since an *equal* indication must be generated in a test for zero operation for the two different representations of zero.

Example 1.3

In the binary case all 2^n sequences are utilized. The 2^{n-1} sequences from $00\cdots0$ to $01\cdots1$ represent positive numbers, while the remaining 2^{n-1} sequences from $10\cdots0$ to $11\cdots1$ represent negative numbers. If $k = n$ (and therefore, $m = 0$ and $ulp = 2^0 = 1$) the range of positive numbers is $\left[0, 2^{n-1} - 1\right]$ and the range of negative numbers is $\left[-(2^{n-1} - 1), -0\right]$. □

A major disadvantage of the signed-magnitude representation is that the operation to be performed may depend on the signs of the operands. For example, when adding a positive number X and a negative number $-Y$ (i.e., Y is the absolute value of the second operand), we need to perform the calculation $X+(-Y)$. If $Y > X$, we should obtain as a final result $-(Y-X)$. We therefore need to first calculate $Y-X$, i.e., switch the order of the operands and perform subtraction rather than addition, and then attach the minus sign. This results in a sequence of decisions that have to be made, costing excess control logic and execution time. This is avoided in the complement representation methods.

2. Complement representations: There are two alternatives:

(i) Radix complement (also called two's complement in the binary system)
(ii) Diminished-radix complement (called one's complement in the binary system)

In both complement methods, a positive number is represented in the same way as in the signed-magnitude method, whereas a negative number, $-Y$, is represented by $(R-Y)$ where R is a constant whose value we will determine next. Such a representation satisfies the basic identity

$$-(-Y) = Y, \tag{1.7}$$

since the complement of $(R-Y)$ is $R-(R-Y) = Y$. One of the major advantages of a complement representation (regardless of the exact value of R) is that no decisions have to be made before executing an addition or subtraction. In the previous example, where a positive and a negative number are to be added, the second operand is represented by $(R-Y)$. Therefore, the addition to be performed is

$$X + (R-Y) = R - (Y-X).$$

If $Y > X$ then the negative result $-(Y-X)$ is already represented in the same complement form; i.e., as $R-(Y-X)$, and there is no need to make any special decisions like interchanging the order of the two operands. However, if $X > Y$, the correct result should be $(X-Y)$ while $X+(R-Y) = R+(X-Y)$. The additional term, R, must be discarded, and the value of R should be selected to simplify or even completely eliminate this correction step.

Another requirement on the value selected for R is that the calculation of the complement $(R-Y)$ of a given number Y be a simple operation that can be done at a high speed. Before deciding on the value of R we define the complement of a single digit x_i, denoted by $\bar{x}_i$, as

$$\bar{x}_i = (r-1) - x_i. \tag{1.8}$$

We denote by $\overline{X}$ the n-tuple $(\bar{x}_{k-1}, \bar{x}_{k-2}, \cdots, \bar{x}_{-m})$ obtained after complementing every digit in the sequence corresponding to X. We now add $\overline{X}$ to X and, based

on Equation (1.8), we obtain $x_i + \bar{x}_i = (r-1)$, independent of the exact value of x_i. We then add ulp to the sum of X and $\overline{X}$, yielding

$$\begin{array}{rccccl} X & x_{k-1} & x_{k-2} & \cdots & x_{-m} & \\ +\overline{X} & \bar{x}_{k-1} & \bar{x}_{k-2} & \cdots & \bar{x}_{-m} & \\ \hline & (r-1) & (r-1) & \cdots & (r-1) & \\ +ulp & & & & 1 & \\ \hline 1 & 0 & 0 & \cdots & 0 & = r^k \end{array}$$

The above calculation can be rewritten as

$$X + \overline{X} + ulp = r^k. \tag{1.9}$$

Note that when the above result is stored into a register of length n ($n = k+m$), the most significant digit is discarded and the final result is zero. In general, storing the result of any arithmetic operation into a fixed-length register is equivalent to taking the remainder after dividing by r^k.

Rearranging the terms in the previous equation results in

$$r^k - X = \overline{X} + ulp.$$

Consequently, if we select the value r^k for R we obtain

$$R - X = r^k - X = \overline{X} + ulp. \tag{1.10}$$

The calculation of the complement $(R - X)$ of a given number X as defined above is quite simple and is independent of the value of k. We call this *radix-complement* representation. No correction is needed for it when the result of the previous operation, $X + (R - Y)$, is positive (i.e., when $X > Y$), since $R = r^k$ is discarded when calculating $R + (X - Y)$.

Example 1.4

For $r = 2$ and $k = n = 4$ (and consequently, $m = 0$ and $ulp = 2^0 = 1$) the radix complement (also called the two's complement in the binary case) of a number X equals $2^4 - X$ but can instead be calculated, according to Equation (1.10), by $\overline{X} + 1$. In this case, the sequences 0000 to 0111 represent the positive numbers 0_{10} to 7_{10}, respectively. The two's complement of the largest positive number is $1000+1 = 1001$ and it represents the value $(-7)_{10}$. The two's complement of zero is $1111 + 1 = 10000 = 0 \bmod 2^4$; i.e., there is a single representation of zero. Thus, each positive number has a corresponding negative number that starts with a 1. There is an

Sequence	Two's complement	One's complement	Signed-magnitude
0111	7	7	7
0110	6	6	6
0101	5	5	5
0100	4	4	4
0011	3	3	3
0010	2	2	2
0001	1	1	1
0000	0	0	0
1111	−1	−0	−7
1110	−2	−1	−6
1101	−3	−2	−5
1100	−4	−3	−4
1011	−5	−4	−3
1010	−6	−5	−2
1001	−7	−6	−1
1000	−8	−7	−0

TABLE 1.1 Three representation methods of binary numbers with $k = n = 4$.

additional sequence that starts with a 1, namely, 1000, which has no corresponding positive number. It represents the negative number $(-8)_{10}$. Therefore, the range of binary numbers in the two's complement method with $k = n = 4$ is $-8 \leq X \leq 7$. The two's complement representations of all values within this range are shown in Table 1.1.

To illustrate the simplicity of executing the operation $X + (-Y)$ with $Y > X$, consider the addition of the numbers 2 and −7, represented by 0010 and 1001, respectively:

$$\begin{array}{rcccc r} & 0 & 0 & 1 & 0 & 2 \\ + & 1 & 0 & 0 & 1 & -7 \\ \hline & 1 & 0 & 1 & 1 & -5 \end{array}$$

This is the correct result represented in the two's complement method, and there is no need for any preliminary decisions or post corrections. Even when $X > Y$, the expected result is calculated without requiring any corrections. For example, when adding 7 and −2, represented by 0111 and 1110, respectively, we obtain

$$\begin{array}{rcccc r} & 0 & 1 & 1 & 1 & 7 \\ + & 1 & 1 & 1 & 0 & -2 \\ \hline 1 & 0 & 1 & 0 & 1 & 5 \end{array}$$

Only the last four least significant digits are retained, yielding 0101. □

A second possible choice for R is $R = r^k - ulp$. This is the *diminished-radix complement,* for which, according to Equation (1.9),

$$R - X = (r^k - ulp) - X = \overline{X}. \tag{1.11}$$

Here, the derivation of the complement is even simpler than that of the radix complement. All the digit-complements $\bar{x}_i$ can be calculated in parallel, leading to a fast computation of $\overline{X}$. On the other hand, a correction step is needed when the result $R + (X - Y)$ is obtained and $(X - Y)$ is positive, as will be explained later.

Example 1.5

For $r = 2$ and $k = n = 4$, the diminished-radix complement (also called the one's complement in the binary case) of a number X equals $(2^4 - 1) - X$, which also equals $\overline{X}$, the sequence of digit complements, according to Equation (1.11). As in the previous example, the sequences 0000 to 0111 represent the positive numbers 0 to 7, respectively. The one's complement of the largest positive number is 1000, representing the value $(-7)_{10}$. The one's complement of zero is 1111; i.e., there are two representations of zero. In summary, the range of binary numbers in the one's complement method with $k = n = 4$ is $-7 \leq X \leq 7$. The different representations of positive and negative numbers with $k = n = 4$ in the three methods are compared in Table 1.1. □

In the binary case, the most significant digit can assume only two values and is thus a "true" sign digit. This holds for all three representation methods as shown in Table 1.1 and the distinction between positive and negative numbers is greatly simplified. In the nonbinary case, restricting the most significant digit to two values only (0 and $(r - 1)$) would considerably reduce the percentage of utilized sequences; only $2 \cdot r^{n-1}$ out of r^n (or 2 out of r) would be used. To make up for this, we can let the most significant digit assume all its possible values and partition the total number, r^n, of sequences equally (or almost equally) between positive and negative values. In general, a given number X is represented in a complement system by X if it is positive or by $R - |X|$ if it is negative. To have unambiguous representations, the regions for positive and negative numbers should not overlap. In other words, the inequality

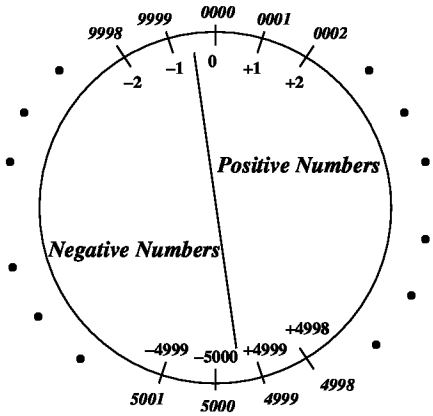
$$|X| \leq R/2$$

must be satisfied. If the value $X = R/2 + 1$ is allowed to be included in the region of representable numbers, then the negative number $-X$ is represented by $R - X = R/2 - 1$, which is identical to the representation of the positive

number $R/2-1$. Nonoverlapping regions for positive and negative numbers can be achieved easily if the radix r is an even number. In this case, in order to satisfy the inequality $|X| \leq R/2 = \frac{r}{2} \cdot r^{n-1}$ (for an integer-only representation), the values $0, 1, \cdots, \frac{r}{2}-1$ for the most significant digit would correspond to positive numbers, while the values $\frac{r}{2}, \cdots, r-1$ would correspond to negative numbers. If, however, the radix is odd, then the representations 0 to $(\frac{r^n}{2}-1)$ must be made positive and the remaining ones negative, making it more difficult to distinguish between positive and negative numbers.

Example 1.6

In the radix-complement decimal system the most significant digit can assume any of its 10 possible values. Thus, all sequences with a leading digit of 0, 1, 2, 3, or 4 represent positive numbers, while those having a leading digit of 5, 6, 7, 8, or 9 represent negative ones. For $n = 4$, the largest positive number is 4999 and the sequences 5000 through 9999 represent negative numbers with values of -5000 through -1, respectively. The range is therefore $-5000 \leq X \leq 4999$ and is shown in the diagram below.



Let $Y = 2345$; to find the representation of $-Y = -2345$ (i.e., the radix complement $R-Y$ with $R = 10^4$) we use the expression $R-Y = \overline{Y}+ulp$. The digit complement in this case is the nine's complement, yielding $\overline{Y} = 7654$ and thus, $\overline{Y}+1 = 7655$. We can verify this result by adding Y to $-Y$, obtaining $2345 + 7655 = 10^4 = 0 \bmod 10^4$. □

The reader should realize by now that algorithms for arithmetic operations can be developed for various fixed-radix number systems and for different partitions of mixed numbers (into their integral and fractional parts). To simplify our

discussion from this point on, we restrict ourselves to binary integers; i.e., $r = 2$ and $k = n$. The extension of what follows to the general case of binary numbers with $m \neq 0$ or radix r numbers with $r \neq 2$ is, in most cases, straightforward.

1.4.1 The Two's Complement Representation

The range of numbers in the two's complement system for $k = n$ is

$$-2^{n-1} \leq X \leq (2^{n-1} - ulp) \quad \text{with} \quad ulp = 2^0 = 1.$$

This range is slightly asymmetric as there is one more negative number than there are positive numbers. The binary number -2^{n-1} (represented by $10\cdots0$) does not have a positive equivalent. Consequently, if a complement operation for this number is attempted, an overflow indication must be generated. On the other hand, there is a unique representation for 0, as shown in Table 1.1.

Given a representation $(x_{n-1}, x_{n-2}, \cdots, x_0)$ in two's complement, we can use the following procedure to find its numerical value X: If $x_{n-1} = 0$, then $X = \sum_{i=0}^{n-1} x_i 2^i$, while if $x_{n-1} = 1$, the given sequence represents a negative number whose absolute value can be obtained by complementing the given sequence and then employing the previous equation.

Example 1.7

Given the 4-tuple 1011, we can find its value by first complementing it to obtain $0100 + 1 = 0101$, then calculating the value of the sequence 0101, which is 5. This indicates that the value of the original sequence is -5. □

Instead of the above procedure we can use the expression

$$X = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i. \tag{1.12}$$

Using Equation (1.12) for 1011 we obtain $-8 + 2 + 1 = -5$.

To prove the validity of Equation (1.12) note first that, if $x_{n-1} = 0$, the exact same positive value is calculated. If $x_{n-1} = 1$, the value of the given representation is

$$\begin{aligned} -[\overline{X} + ulp] &= -\left[\sum_{i=0}^{n-2} \bar{x}_i 2^i + 1\right] = -\left[\sum_{i=0}^{n-2} (1 - x_i) 2^i + 1\right] \\ &= -\left[\sum_{i=0}^{n-2} 2^i - \sum_{i=0}^{n-2} x_i 2^i + 1\right] = -\left[(2^{n-1} - 1) - \sum_{i=0}^{n-2} x_i 2^i + 1\right] \\ &= -2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \end{aligned}$$

which is exactly the value of the right-hand side of Equation (1.12) for $x_{n-1} = 1$.

1.4.2 The One's Complement Representation

The range of representable numbers in the one's complement system for $k = n$ is symmetric, and equals

$$-(2^{n-1} - ulp) \leq X \leq (2^{n-1} - ulp) \quad \text{with} \quad ulp = 2^0 = 1$$

As a result, there are two representations of zero, a positive zero represented by $000\cdots0$, and a negative zero represented by $111\cdots1$, as shown in Table 1.1.

For the one's complement system we have an equation similar to that of the two's complement system:

$$X = -x_{n-1}(2^{n-1} - ulp) + \sum_{i=0}^{n-2} x_i 2^i. \tag{1.13}$$

For example, the 4-tuple 1010 represents the value $-(2^3 - 1) + 2 = -5$. The proof of Equation (1.13) is left to the reader as an exercise.

The derivation of the one's complement is simpler than that of the two's complement. For each digit we have to calculate $\bar{x}_i = 1 - x_i$, which is the Boolean complement and can be done in parallel for all digits.

1.5 ADDITION AND SUBTRACTION

When adding or subtracting numbers represented in the signed-magnitude representation, only the magnitude bits participate in the arithmetic operation, while the sign bits are treated separately. Consequently, a carry-out (or borrow-out) indicates overflow. For example,

0		1	0	1	1	11	
0	+	0	1	1	0	6	
0	1	0	0	0	1	1	Carry-out

The final result is positive (the sum of two positive numbers), but its magnitude in four bits is erroneously obtained as 1 instead of 17, since $1 = 17 \bmod 16$.

In both complement systems, all digits, including the sign digit, participate in the add or subtract operation. A carry-out is therefore not necessarily an indication of an overflow in these systems. For example, when adding the two numbers $X = 13$ and $Y = -8$ represented in the two's complement method, we obtain

	0	1	1	0	1	13	
+	1	1	0	0	0	−8	
1	0	0	1	0	1	5	Carry-out but no overflow

The carry-out is discarded and does not indicate overflow. In general, if X and Y have opposite signs, no overflow can occur regardless whether there is a carry-out or not, as illustrated in the following two examples:

	0	0	1	0	1	5	
+	1	0	1	1	0	−10	
	1	1	0	1	1	−5	No carry-out
	0	1	0	1	0	10	
+	1	1	0	1	1	−5	
1	0	0	1	0	1	5	Carry-out

If X and Y have the same sign and the sign of the result is different from that of the two operands, then an overflow occurs. For example,

	1	1	0	0	1	−7	
+	1	0	1	1	0	−10	
1	0	1	1	1	1	15	Carry-out and overflow
	0	0	1	1	1	7	
+	0	1	0	1	0	10	
	1	0	0	0	1	−15	No carry-out but overflow

In the one's complement system a carry-out is an indication that a correction step is needed. For example, when adding a positive number X and a negative number $-Y$ represented in one's complement, the result is

$$X + (2^n - ulp) - Y = (2^n - ulp) + (X - Y)$$

and if $X > Y$ we should obtain $(X - Y)$. The term 2^n represents the carry-out bit, which is discarded since the final result should be stored into a register of length n. The result is therefore $(X - Y - ulp)$, and the necessary correction is made by adding ulp. For example,

			0	1	0	1	0	10
	+		1	1	0	1	0	−5
1			0	0	1	0	0	
Correction	+						1	*ulp*
			0	0	1	0	1	5

The generated carry-out is called *end-around carry,* meaning that the carry-out is an *indication* that a 1 should be added to the least significant position. If there is no carry-out then no correction is needed. This is the case when $X < Y$ and thus the result, $-(Y - X)$, is negative and should be represented by $(2^n - ulp) - (Y - X)$. For example,

	0	0	1	0	1	5	
+	1	0	1	0	1	−10	
	1	1	0	1	0	−5	No carry-out and hence no correction

As we have seen before, no such correction is necessary in two's complement addition.

In both complement systems, a subtract operation, $X - Y$, is performed by adding the complement of Y to X. In the one's complement system this means adding $\overline{Y}$ to X; in other words, $X - Y = X + \overline{Y}$. In the two's complement system we perform $X - Y = X + (\overline{Y} + ulp)$. This still requires only a single adder operation, since ulp is added through the forced carry input to the binary adder. This will be further explained in Chapter 5.

1.6 ARITHMETIC SHIFT OPERATIONS

Another way of distinguishing among the three methods for representing negative numbers is to consider the infinite extensions to the right and to the left of a given number. In the signed-magnitude method, the magnitude $x_{n-2}, \cdots, x_0$ can be viewed as the infinite sequence

$$\cdots 0, 0, \{x_{n-2}, \cdots, x_0\}, 0, 0 \cdots$$

If any arithmetic operation results in a nonzero prefix, then this constitutes an overflow.

In the radix-complement scheme the infinite extension is

$$\cdots x_{n-1}, x_{n-1}, \{x_{n-1}, \cdots, x_0\}, 0, 0 \cdots$$

where x_{n-1} is the sign digit. Finally, for the diminished-radix complement scheme the sequence is

$$\cdots x_{n-1}, x_{n-1}, \{x_{n-1}, \cdots, x_0\}, x_{n-1}, x_{n-1} \cdots$$

Example 1.8

The sequences 1011., 11011.0, and 111011.00 all represent the value -5 in the two's complement method. Similarly, the sequences 1010., 11010.1, and 1110101.11 all represent the value -5 in the one's complement method. The general proof is left as an exercise for the reader. □

These extensions are useful when adding operands with different numbers of bits. The shorter operand must be extended to the length of the longer one before being added. Based on these extensions we can derive the rules for arithmetic shift operations where a left and right shift are equivalent to multiplication and division by 2, respectively.

Example 1.9

In two's complement,

$$\begin{aligned} Sh.L\ \{00101_2 = +5\} &= 01010_2 = +10 \\ Sh.R\ \{00101_2 = +5\} &= 00010_2 = +2 \\ Sh.L\ \{11011_2 = -5\} &= 10110_2 = -10 \\ Sh.R\ \{11011_2 = -5\} &= 11101_2 = -3 \end{aligned}$$

In one's complement,

$$\begin{aligned} Sh.L\ \{11010_2 = -5\} &= 10101_2 = -10 \\ Sh.R\ \{11010_2 = -5\} &= 11101_2 = -2 \end{aligned}$$

□

Arithmetic shift operations are very useful in many algorithms for multiplication and division, as will be described in Chapter 3.

1.7 EXERCISES

1.1. **(a)** Find the fixed-point representations of the two values $(41.25)_{10}$ and $(-41.25)_{10}$ in the radix complement and diminished-radix complement systems if the radix r is 8, $k = 3$ integer digits, and $m = 1$ fraction digit.
(b) Repeat (a) for $r = 2$, $k = 8$ and $m = 2$.

1.2. Verify the correctness of the following procedure to obtain the two's complement, $(y_{n-1}, y_{n-2}, \cdots, y_0)$ of a given sequence $(x_{n-1}, x_{n-2}, \cdots, x_0)$.
Start from the rightmost bit x_0. For each 0 in X set the corresponding bit in Y to 0 until you reach the first 1. For this 1 set the corresponding bit to 1. From this point on set $y_i = \bar{x}_i$.

1.3. Show that approximately $3.3n$ bits are needed to represent an n-digit decimal number.

1.4. In this problem we attempt to find the most "efficient" fixed-radix number system. Assume that N different values need to be represented and search for the radix r that minimizes the product $E = n \cdot r$ where n is the number of radix r digits that are required to represent N values; i.e., $n = \log_r N$. Thus, the function $E = r \cdot \log_r N = (r/\log_{10} r) \cdot \log_{10} N$ must be minimized.
(a) Justify the selection of the objective function E.
(b) Tabulate the values of $r/\log_{10} r$ for $r = 2, e, 3, \cdots, 10$. What are the best choices for r?

1.5. Prove the extensions of two's complement and one's complement numbers to infinite sequences. In particular, based on Equation (1.12) show that the representation of a signed binary number in $(n+1)$ bits in the two's complement method may be derived from its representation in n bits by repeating the sign bit.

1.6. Given an n-tuple $X = (x_{n-1}, \cdots, x_0)$ find the value of $Sh.R\{X\} + Sh.R\{-X\}$ and of $Sh.L\{X\} + Sh.L\{-X\}$ using either the diminished-radix complement or the radix complement representation.

1.7. What are the rules for overflow detection in add/subtract operations when using two's complement or one's complement representations, assuming that carry-in and carry-out signals for the sign digit are available?

1.8. Prove that the value of any number in the one's complement system can be calculated, for any k and m, using the formula

$$X = -x_{k-1}(2^{k-1} - ulp) + \sum_{i=-m}^{k-2} x_i 2^i$$

1.9. Can the equation $X = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$ be extended to the radix complement representation for any positive radix r?

1.10. The difference $X - Y$ can be formed by adding the complement of X to Y and then complementing the result. Prove that this is true for any complement scheme and any general fixed-radix number system. Is it useful?

1.11. Show that a carry-out in any add or subtract operation of one's complement numbers indicates that an end-around carry must be added to correct the final result. Pay particular attention to the case where two negative numbers need to be added.

1.12. Can the correction step in one's complement addition by adding an end-around carry generate another end-around carry?

1.13. In the signed-magnitude representation there is another way to perform the operation $X + (-Y)$ when $Y > X$. Instead of changing the order of operands and calculating $-(Y - X)$, we can simply subtract Y from X. However, this will result in a need for a correction. For example,

0		1	0	1	1	+11
1	−	1	1	0	1	−13
1		1	1	1	0	−14

while the correct result is $1\ 0010 = (-2)_{10}$. Show that the required correction can be done by taking the two's complement of the result 1110. Explain why the complement operation provides the necessary correction.

1.8 REFERENCES

A number of textbooks focusing on computer arithmetic have been published since 1963. These include:

[1] J. J. F. CAVANAGH, *Digital computer arithmetic: Design and implementation,* McGraw-Hill, New York, 1984.

[2] I. FLORES, *The logic of computer arithmetic,* Prentice Hall, Englewood Cliffs, NJ, 1963.

[3] M. J. FLYNN AND S. F. OBERMAN, *Advanced computer arithmetic design,* Wiley, New York, 2001.

[4] J. B. GOSLING, *Design of arithmetic units for digital computers,* Springer-Verlag, New York, 1980.

[5] K. HWANG, *Computer arithmetic: Principles, architecture, and design,* Wiley, New York, 1978.

[6] J-M. MULLER, *Elementary Functions: Algorithms and Implementations,* Birkhäuser, Boston, 1997.

[7] A. R. OMONDI, *Computer arithmetic systems: Algorithms, architecture and implementations,* Prentice Hall, Englewood Cliffs, NJ, 1994.

[8] B. PARHAMI, *Computer arithmetic: Algorithms and Hardware Designs,* Oxford University Press, New York, 2000.

[9] N. R. SCOTT, *Computer number systems and arithmetic,* Prentice Hall, Englewood Cliffs, NJ, 1985.

[10] O. SPANIOL, *Computer arithmetic: Logic and design,* Wiley, New York, 1981.

[11] S. WASER and M. J. FLYNN, *Introduction to arithmetic for digital system designers,* Holt, Rinehart, Winston, New York, 1982.

Reprints of many classic papers appear in the next two volumes:

[12] E. E. SWARTZLANDER, JR. (Editor), *Computer arithmetic,* vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[13] E. E. SWARTZLANDER, JR. (Editor), *Computer arithmetic,* vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

Several chapters in computer organization textbooks and some survey articles have been devoted to computer arithmetic including:

[14] Y. CHU, *Computer organization and microprogramming,* Prentice Hall, Englewood Cliffs, NJ, 1972, chap. 5.

[15] H. L. GARNER, "Number systems and arithmetic," in *Advances in computers,* vol. 6, F. L. Alt and M. Rubinoff (Eds.), Academic, New York, 1965, pp. 131-194.

[16] H. L. GARNER, "Theory of computer addition and overflows," *IEEE Trans. on Computers, C-27* (April 1978), 297-301.

[17] V. C. HAMACHER, Z. G. VRANESIC, and S. G. ZAKY, *Computer organization,* 2nd ed., McGraw-Hill, New York, 1984.

[18] D. GOLDBERG, "Computer arithmetic," in *Computer architecture: A quantitative approach,* D. A. Patterson and J. L. Hennessy, 2nd edition, Morgan Kaufmann, San Mateo, CA, 1996.

[19] U. KULISCH, "Mathematical foundations of computer arithmetic," *IEEE Trans. on Computers, C-26* (July 1977), 610-620.

[20] O. L. MACSORLEY, "High-speed arithmetic in binary computers," *Proc. of IRE, 49* (Jan. 1961), 67-91.

[21] G. W. REITWIESNER, "Binary arithmetic," in *Advances in computers,* vol. 1, F. L. Alt (Editor), Academic, New York, 1960, pp. 231-308.

[22] C. TUNG, "Arithmetic," in *Computer science,* A. F. Cardenas et al. (Eds.), Wiley-Interscience, New York, 1972.