COMPUTER ARITHMETIC

Algorithms and Hardware Designs

Behrooz Parhami

Department of Electrical and Computer Engineering
University of California, Santa Barbara

New York Oxford
OXFORD UNIVERSITY PRESS
2000

Oxford University Press

Oxford New York
Athens Auckland Bangkok Bogotá Buenos Aires Calcutta
Cape Town Chennai Dar es Salaam Delhi Florence Hong Kong Istanbul
Karachi Kuala Lumpur Madrid Melbourne Mexico City Mumbai
Nairobi Paris São Paulo Singapore Taipei Tokyo Toronto Warsaw

and associated companies in
Berlin Ibadan

Published by Oxford University Press, Inc.
198 Madison Avenue, New York, New York 10016
http://www.oup-usa.org

Oxford is a registered trademark of Oxford University Press

Library of Congress Cataloging-in-Publication Data

Parhami, Behrooz.
Computer arithmetic : algorithms and hardware designs / Behrooz Parhami.
p. cm.
Includes bibliographical references and index.
ISBN 0-19-512583-5 (cloth)
1. Computer arithmetic. 2. Computer algorithms. I. Title.
QA76.9.C62P37 1999
004'.01'513—dc21 98-44899
CIP

Printing (last digit): 9 8 7 6 5 4 3 2 1

Printed in the United States of America
on acid-free paper

To the memory of my father,
Salem Parhami (1922–1992),
and to all others on whom I can count
for added inspiration,
multiplied joy,
and divided anguish.

CONTENTS

PREFACE

THE CONTEXT OF COMPUTER ARITHMETIC

Advances in computer architecture over the past two decades have allowed the performance of digital computer hardware to continue its exponential growth, despite increasing technological difficulty in speed improvement at the circuit level. This phenomenal rate of growth, which is expected to continue in the near future, would not have been possible without theoretical insights, experimental research, and tool-building efforts that have helped transform computer architecture from an art into one of the most quantitative branches of computer science and engineering. Better understanding of the various forms of concurrency and the development of a reasonably efficient and user-friendly programming model have been key enablers of this success story.

The downside of exponentially rising processor performance is an unprecedented increase in hardware and software complexity. The trend toward greater complexity is not only at odds with testability and certifiability but also hampers adaptability, performance tuning, and evaluation of the various trade-offs, all of which contribute to soaring development costs. A key challenge facing current and future computer designers is to reverse this trend by removing layer after layer of complexity, opting instead for clean, robust, and easily certifiable designs, while continuing to try to devise novel methods for gaining performance and ease-of-use benefits from simpler circuits that can be readily adapted to application requirements.

In the computer designers'quest for user-friendliness, compactness, simplicity, high performance, low cost, and low power, computer arithmetic plays a key role. It is one of oldest subfields of computer architecture. The bulk of hardware in early digital computers resided in accumulator and other arithmetic/logic circuits. Thus, first-generation computer designers were motivated to simplify and share hardware to the extent possible and to carry out detailed cost–performance analyses before proposing a design. Many of the ingenious design methods that we use today have their roots in the bulky, power-hungry machines of 30–50 years ago.

In fact computer arithmetic has been so successful that it has, at times, become transparent. Arithmetic circuits are no longer dominant in terms of complexity; registers, memory and memory management, instruction issue logic, and pipeline control have become the dominant consumers of chip area in today's processors. Correctness and high performance of arithmetic circuits is routinely expected, and episodes such as the Intel Pentium division bug are indeed rare.

The preceding context is changing for several reasons. First, at very high clock rates, the interfaces between arithmetic circuits and the rest of the processor become critical. Arithmetic units can no longer be designed and verified in isolation. Rather, an integrated design optimization is required, which makes the development even more complex and costly. Second, optimizing arithmetic circuits to meet design goals by taking advantage of the strengths of new

technologies, and making them tolerant to the weaknesses, requires a reexamination of existing design paradigms. Finally, incorporation of higher-level arithmetic primitives into hardware makes the design, optimization, and verification efforts highly complex and interrelated.

This is why computer arithmetic is alive and well today. Designers and researchers in this area produce novel structures with amazing regularity. Carry-lookahead adders comprise a case in point. We used to think, in the not so distant past, that we knew all there was to know about carry-lookahead fast adders. Yet, new designs, improvements, and optimizations are still appearing. The ANSI/IEEE standard floating-point format has removed many of the concerns with compatibility and error control in floating-point computations, thus resulting in new designs and products with mass-market appeal. Given the arithmetic-intensive nature of many novel application areas (such as encryption, error checking, and multimedia), computer arithmetic will continue to thrive for years to come.

THE GOALS AND STRUCTURE OF THIS BOOK

The field of computer arithmetic has matured to the point that a dozen or so texts and reference books have been published. Some of these books that cover computer arithmetic in general (as opposed to special aspects or advanced/unconventional methods) are listed at the end of the preface. Each of these books has its unique strengths and has contributed to the formation and fruition of the field. The current text, *Computer Arithmetic: Algorithms and Hardware Designs*, is an outgrowth of lecture notes the author developed and refined over many years. Here are the most important features of this text in comparison to the listed books:

Division of material into lecture-size chapters. In my approach to teaching, a lecture is a more or less self-contained module with links to past lectures and pointers to what will transpire in future. Each lecture must have a theme or title and must proceed from motivation, to details, to conclusion. In designing the text, I strived to divide the material into chapters, each of which is suitable for one lecture (1–2 hours). A short lecture can cover the first few subsections, while a longer lecture can deal with variations, peripheral ideas, or more advanced material near the end of the chapter. To make the structure hierarchical, as opposed to flat or linear, lectures are grouped into seven parts, each composed of four lectures and covering one aspect of the field (Fig. P.1).

Emphasis on both the underlying theory and actual hardware designs. The ability to cope with complexity requires both a deep knowledge of the theoretical underpinnings of computer arithmetic and examples of designs that help us understand the theory. Such designs also provide building blocks for synthesis as well as reference points for cost–performance comparisons. This viewpoint is reflected in, for example, the detailed coverage of redundant number representations and associated arithmetic algorithms (Chapter 3) that later lead to a better understanding of various multiplier designs and on-line arithmetic. Another example can be found in Chapter 22, where CORDIC algorithms are introduced from the more intuitive geometric viewpoint.

Linking computer arithmetic to other subfields of computing. Computer arithmetic is nourished by, and in turn nourishes, other subfields of computer architecture and technology. Examples of such links abound. The design of carry-lookahead adders became much more systematic once it was realized that the carry computation is a special case of parallel prefix computation that had been extensively studied by researchers in parallel computing. Arithmetic for and by neural networks is an area that is still being

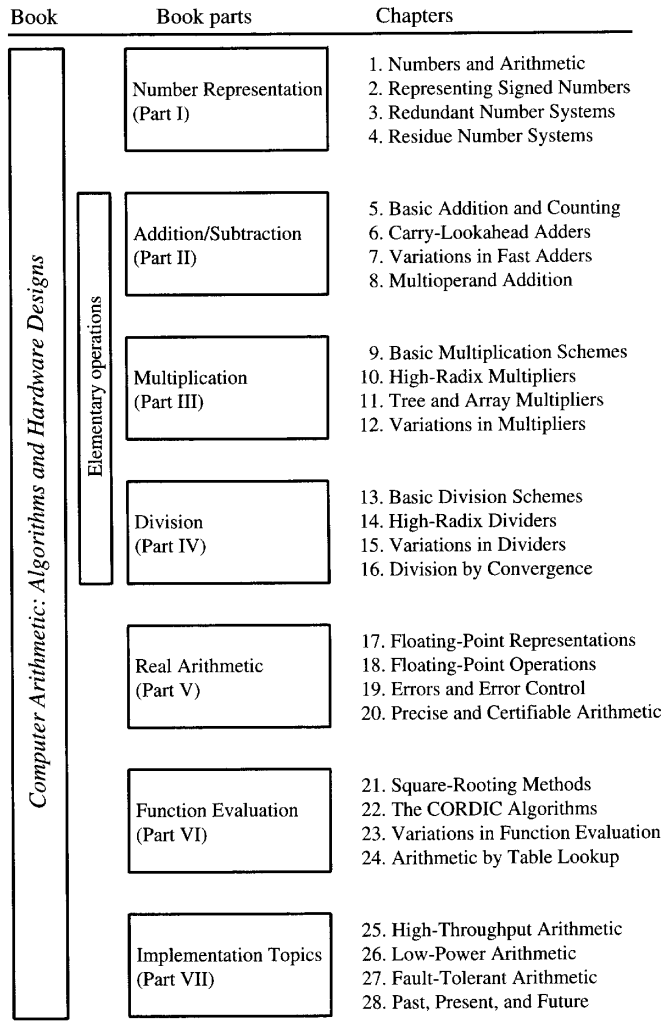


Fig. P.1 The structure of this book in parts and chapters.

explored. The residue number system has provided an invaluable tool for researchers interested in complexity theory and the limits of fast arithmetic, as well as to the designers of fault-tolerant digital systems.

Wide coverage of important topics. The text covers virtually all important algorithmic and hardware design topics in computer arithmetic, thus providing a balanced and complete view of the field. Coverage of unconventional number representation methods (Chapters 3 and 4), arithmetic by table lookup (Chapter 24), which is becoming increasingly important, multiplication and division by constants (Chapters 9 and 13), errors and certifiable arithmetic (Chapters 19 and 20), and the topics in Part VII (Chapters 25–28) do not all appear in other textbooks.

Unified and consistent notation and terminology throughout the text. Every effort is made to use consistent notation and terminology throughout the text. For example, r always stands for the number representation radix and s for the remainder in division or square-rooting. While other authors have done this in the basic parts of their texts, many tend to cover more advanced research topics by simply borrowing the notation and terminology from the reference source. Such an approach has the advantage of making the transition between reading the text and the original reference source easier, but it is utterly confusing to the majority of the students, who rely on the text and do not consult the original references except, perhaps, to write a research paper.

SUMMARY OF TOPICS

The seven parts of this book, each composed of four chapters, were written with the following goals.

Part I sets the stage, gives a taste of what is to come, and provides a detailed perspective on the various ways of representing fixed-point numbers. Included are detailed discussions of signed numbers, redundant representations, and residue number systems.

Part II covers addition and subtraction, which form the most basic arithmetic building blocks and are often used in implementing other arithmetic operations. Included in the discussions are addition of a constant (counting), various methods for designing fast adders, and multioperand addition.

Part III deals exclusively with multiplication, beginning with the basic shift/add algorithms and moving on to high-radix, tree, array, bit-serial, modular, and a variety of other multipliers. The special case of squaring is also discussed.

Part IV covers division algorithms and their hardware implementations, beginning with the basic shift/subtract algorithms and moving on to high-radix, prescaled, modular, array, and convergence dividers.

Part V deals with real number arithmetic, including various methods for representing real numbers, floating-point arithmetic, errors in representation and computation, and methods for high-precision and certifiable arithmetic.

Part VI covers function evaluation, beginning with the important special case of square-rooting and moving on to CORDIC algorithms, followed by general convergence and approximation methods, including the use of lookup tables.

Part VII deals with broad design and implementation topics, including pipelining, low-power arithmetic, and fault tolerance. This part concludes by providing historical perspective and examples of arithmetic units in real computers.

POINTERS ON HOW TO USE THE BOOK

For classroom use, the topics in each chapter of this text can be covered in a lecture lasting 1–2 hours. In my own teaching, I have used the chapters primarily for 1.5-hour lectures, twice a week, in a 10-week quarter, omitting or combining some chapters to fit the material into 18–20

lectures. But the modular structure of the text lends itself to other lecture formats, self-study, or review of the field by practitioners. In the latter two cases, readers can view each chapter as a study unit (for one week, say) rather than as a lecture. Ideally, all topics in each chapter should be covered before the reader moves to the next chapter. However, if fewer lecture hours are available, some of the subsections located at the end of chapters can be omitted or introduced only in terms of motivations and key results.

Problems of varying complexities, from straightforward numerical examples or exercises to more demanding studies or miniprojects, are supplied for each chapter. These problems form an integral part of the book: they were not added as afterthoughts to make the book more attractive for use as a text. A total of 464 problems are included (15–18 per chapter). Assuming that two lectures are given per week, either weekly or biweekly homework can be assigned, with each assignment having the specific coverage of the respective half-part (two chapters) or full-part (four chapters) as its "title."

An instructor's manual, with problem solutions and enlarged versions of the diagrams and tables, suitable for reproduction as transparencies, is planned. The author's detailed syllabus for the course ECE 252B at UCSB is available at:

http://www.ece.ucsb.edu/courses/syllabi/default.html.

A simulator for numerical experimentation with various arithmetic algorithms is available at:

http://www.ecs.umass.edu/ece.koren/arith.simulator

courtesy of Pofessor Israel Koren.

References to classical papers in computer arithmetic, key design ideas, and important state-of-the-art research contributions are listed at the end of each chapter. These references provide good starting points for in-depth studies or for term papers or projects. A large number of classical papers and important contributions in computer arithmetic have been reprinted in two volumes [Swar90].

New ideas in the field of computer arithmetic appear in papers presented at biannual conferences, known as ARITH-*n*, held in odd-numbered years [Arit]. Other conferences of interest include Asilomar Conference on Signals, Systems, and Computers [Asil], International Conference on Circuits and Systems [ICCS], Midwest Symposium on Circuits and Systems [MSCS], and International Conference on Computer Design [ICCD]. Relevant journals include *IEEE Transactions on Computers* [TrCo], particularly its special issues on computer arithmetic, *IEEE Transactions on Circuits and Systems* [TrCS], *Computers & Mathematics with Applications* [CoMa], *IEE Proceedings: Computers and Digital Techniques* [PrCD], *IEEE Transactions on VLSI Systems* [TrVL], and *Journal of VLSI Signal Processing* [JVSP].

ACKNOWLEDGMENTS

Computer Arithmetic: Algorithms and Hardware Designs is an outgrowth of lecture notes the author used for the graduate course "ECE 252B: Computer Arithmetic" at the University of California, Santa Barbara, and, in rudimentary forms, at several other institutions prior to 1988. The text has benefited greatly from keen observations, curiosity, and encouragement of my many students in these courses. A sincere thanks to all of them!

REFERENCES

[Arit] International Symposium on Computer Arithmetic, sponsored by the IEEE Computer Society. This series began with a one-day workshop in 1969 and was subsequently held in 1972, 1975, 1978, and in odd-numbered years since 1981. The 13th symposium in the series, ARITH-13, was held on July 6–9, 1997, in Asilomar, California. ARITH-14 was held April 14–16, 1999, in Adelaide, Australia.

[Asil] Asilomar Conference on Signals Systems, and Computers, sponsored annually by IEEE and held on the Asilomar Conference Grounds in Pacific Grove, California. The 32nd conference in this series was held on November 1–4, 1998.

[Cava84] Cavanagh, J. J. F., *Digital Computer Arithmetic: Design and Implementation*, McGraw-Hill, 1984.

[CoMa] *Computers & Mathematics with Applications*, journal published by Pergamon Press.

[Flor63] Flores, I., *The Logic of Computer Arithmetic*, Prentice-Hall, 1963.

[Gosl80] Gosling, J. B., *Design of Arithmetic Units for Digital Computers*, Macmillan, 1980.

[Hwan79] Hwang, K., *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, 1979.

[ICCD] International Conference on Computer Design, sponsored annually by the IEEE Computer Society. ICCD-98 was held on October 4–7, 1998, in Austin, Texas.

[ICCS] International Conference on Circuits and Systems, sponsored annually by the IEEE Circuits and Systems Society. The latest in this series was held on May 31–June 3, 1998, in Monterey, California.

[JVSP] *J. VLSI Signal Processing*, published by Kluwer Academic Publishers.

[Knut97] Knuth, D. E., *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms*, 3rd ed., Addison-Wesley, 1997. (The widely used second edition, published in 1981, is cited in Parts V and VI.)

[Kore93] Koren, I., *Computer Arithmetic Algorithms*, Prentice-Hall, 1993.

[Kuli81] Kulisch, U. W., and W. L. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, 1981.

[MSCS] Midwest Symposium on Circuits and Systems, sponsored annually by the IEEE Circuits and Systems Society. This series of symposia began in 1955, with the 41st in the series held on August 9–12, 1998, in Notre Dame, Indiana.

[Omon94] Omondi, A. R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*, Prentice-Hall, 1994.

[PrCD] *IEE Proc: Computers and Digital Techniques*, journal published by the Institution of Electrical Engineers, United Kingdom.

[Rich55] Richards, R. K., *Arithmetic Operations in Digital Computers*, Van Nostrand, 1955.

[Scot85] Scott, N. R., *Computer Number Systems and Arithmetic*, Prentice-Hall, 1985.

[Stei71] Stein, M. L., and W. D. Munro, *Introduction to Machine Arithmetic*, Addison-Wesley, 1971.

[Swar90] Swartzlander, E. E., Jr., *Computer Arithmetic,* Vols. I and II, IEEE Computer Society Press, 1990.

[TrCo] *IEEE Trans. Computers*, journal published by the IEEE Computer Society. Occasionally entire special issues or sections are devoted to computer arithmetic (e.g.: Vol. 19, No. 8, August 1970; Vol. 22, No. 6, June 1973; Vol. 26, No. 7, July 1977; Vol. 32, No. 4, April 1983; Vol. 39, No. 8, August 1990; Vol. 41, No. 8, August 1992; Vol. 43, No. 8, August 1994; Vol. 47, No. 7, July 1998).

[TrCS] *IEEE Trans. Circuits and Systems—II: Analog and Digital Signal Processing*, journal published by IEEE.

[TrVL] *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, journal published jointly by the IEEE Circuits and Systems Society, Computer Society, and Solid-State Circuits Council.

[Wase82] Waser, S., and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart, & Winston, 1982.

[Wino80] Winograd, S., *Arithmetic Complexity of Computations*, SIAM, 1980.

COMPUTER ARITHMETIC

PART I NUMBER REPRESENTATION

Number representation is arguably the most important topic in computer arithmetic. In justifying this claim, it suffices to note that several important classes of number representations were discovered, or rescued from obscurity, by computer designers in their quest for simpler and faster circuits. Furthermore, the choice of number representation affects the implementation cost and delay of all arithmetic operations. We thus begin our study of computer arithmetic by reviewing conventional and exotic representation methods for integers. Conventional methods are of course used extensively. Some of the unconventional methods have been applied to special-purpose digital systems or in the intermediate steps of arithmetic hardware implementations where they are often invisible to computer users. This part consists of the following four chapters:

Chapter 1 NUMBERS AND ARITHMETIC

This chapter motivates the reader, sets the context in which the material in the rest of the book is presented, and reviews positional representations of fixed-point numbers. The chapter ends with a review of methods for number radix conversion and a preview of other number representation methods to be covered. Chapter topics include:

1.1 WHAT IS COMPUTER ARITHMETIC?

A sequence of events, begun in late 1994 and extending into 1995, embarrassed the world's largest computer chip manufacturer and put the normally dry subject of computer arithmetic on the front pages of major newspapers. The events were rooted in the work of Thomas Nicely, a mathematician at the Lynchburg College in Virginia, who is interested in twin primes (consecutive odd numbers such as 29 and 31 that are both prime). Nicely's work involves the distribution of twin primes and, particularly, the sum of their reciprocals $S = 1/5 + 1/7 + 1/11 + 1/13 + 1/17 + 1/19 + 1/29 + 1/31 + \cdots + 1/p + 1/(p+2) + \cdots$. While it is known that the infinite sum S has a finite value, no one knows what the value is.

Nicely was using several different computers for his work and in March 1994 added a machine based on the Intel Pentium processor to his collection. Soon he began noticing inconsistencies in his calculations and was able to trace them back to the values computed for $1/p$ and $1/(p+2)$ on the Pentium processor. At first, he suspected his own programs, the compiler, and the operating system, but by October, he became convinced that the Intel Pentium chip was at fault. This suspicion was confirmed by several other researchers following a barrage of e-mail exchanges and postings on the Internet.

The diagnosis finally came from Tim Coe, an engineer at Vitesse Semiconductor. Coe built a model of Pentium's floating-point division hardware based on the radix-4 SRT algorithm and came up with an example that produces the worst-case error. Using double-precision floating-point computation, the ratio $c = 4\,195\,835/3\,145\,727 = 1.333\,820\,44\cdots$ is computed as 1.333 739 06 on the Pentium. This latter result is accurate to only 14 bits; the error is even larger than that of single-precision floating-point and more than 10 orders of magnitude worse that what is expected of double-precision computation [Mole95].

The rest, as they say, is history. Intel at first dismissed the severity of the problem and admitted only a "subtle flaw," with a probability of 1 in 9 billion, or once in 27,000 years for the average spreadsheet user, of leading to computational errors. It nevertheless published a "white paper" that described the bug and its potential consequences and announced a replacement policy for the defective chips based on "customer need"; that is, customers had to show that they were doing a lot of mathematical calculations to get a free replacement. Under heavy criticism from customers, manufacturers using the Pentium chip in their products, and the on-line community, Intel later revised its policy to no-questions-asked replacement.

Whereas supercomputing, microchips, computer networks, advanced applications (particularly chess-playing programs), and many other aspects of computer technology have made the news regularly in recent years, the Intel Pentium bug was the first instance of arithmetic (or anything inside the CPU for that matter) becoming front-page news. While this can be interpreted as a sign of pedantic dryness, it is more likely an indicator of stunning technological success. Glaring software failures have come to be routine events in our information-based society, but hardware bugs are rare and newsworthy.

Having read the foregoing account, you may wonder what the radix-4 SRT division algorithm is and how it can lead to such problems. Well, that's the whole point of this introduction! You need computer arithmetic to understand the rest of the story. Computer arithmetic is a subfield of digital computer organization. It deals with the hardware realization of arithmetic functions to support various computer architectures as well as with arithmetic algorithms for firmware or software implementation. A major thrust of digital computer arithmetic is the design of hardware algorithms and circuits to enhance the speed of numeric operations. Thus much of what is presented here complements the *architectural* and *algorithmic* speedup techniques studied in the context of high-performance computer architecture and parallel processing.

A majority of our discussions relate to the design of top-of-the-line CPUs with high-performance parallel arithmetic circuits. However, we will at times also deal with slow bit-serial designs for embedded applications, where implementation cost and I/O pin limitations are of prime concern. It would be a mistake, though, to conclude that computer arithmetic is useful only to computer designers. We will see shortly that you can use scientific calculators more effectively and write programs that are more accurate and/or more efficient after a study of computer arithmetic. You will be able to render informed judgment when faced with the problem of choosing a digital signal processor (DSP) chip for your project. And, of course, you will know what exactly went wrong in the Pentium.

Figure 1.1 depicts the scope of computer arithmetic. On the hardware side, the focus is on implementing the four basic arithmetic operations (five, if you count square-rooting), as well as commonly used computations such as exponentials, logarithms, and trigonometric functions. For this, we need to develop algorithms, translate them to hardware structures, and choose from among multiple implementations based on cost–performance criteria. Since the exact computations to be carried out by the general-purpose hardware are not known a priori, benchmarking is used to predict the overall system performance for typical operation mixes and to make various design decisions.

On the software side, the primitive functions are given (e.g., in the form of a hardware chip such as the Pentium processor or a software tool such as Mathematica), and the task is

Hardware (our focus in this book)

Design of efficient digital circuits for primitive and other arithmetic operations such as $+, -, \times, \div, \sqrt{\ }$, log, sin, and cos

Issues: Algorithms
Error analysis
Speed/cost trade-offs
Hardware implementation
Testing, verification

General-Purpose

Flexible data paths
Fast primitive operations like $+, -, \times, \div, \sqrt{\ }$
Benchmarking

Special-Purpose

Tailored to application areas such as
Digital filtering
Image processing
Radar tracking

Software

Numerical methods for solving systems of linear equations, partial differential equations and so on

Issues: Algorithms
Error analysis
Computational complexity
Programming
Testing, verification

Fig. 1.1 The scope of computer arithmetic.

to synthesize cost-effective algorithms, with desirable error characteristics, to solve various problems of interest. These topics are covered in numerical analysis and computational science courses and textbooks and are thus mostly outside the scope of this book.

Within the hardware realm, we will be dealing with both general-purpose arithmetic/logic units (ALUs), of the type found in many commercially available processors, and special-purpose structures for solving specific application problems. The differences in the two areas are minor as far as the arithmetic algorithms are concerned. However, in view of the specific technological constraints, production volumes, and performance criteria, hardware implementations tend to be quite different. General-purpose processor chips that are mass-produced have highly optimized custom designs. Implementations of low-volume, special-purpose systems, on the other hand, typically rely on semicustom and off-the-shelf components. However, when critical and strict requirements, such as extreme speed, very low power consumption, and miniature size, preclude the use of semicustom or off-the-shelf components, the much higher cost of a custom design may be justified even for a special-purpose system.

1.2 A MOTIVATING EXAMPLE

Use a calculator that has the square-root, square, and exponentiation (x^y) functions to perform the following computations. I have given the numerical results obtained with my (10+2)-digit scientific calculator. You may obtain slightly different values.

First, compute "the 1024th root of 2" in the following two ways:

$$u = \underbrace{\sqrt{\sqrt{\cdots\sqrt{2}}}}_{10 \text{ times}} = 1.000\,677\,131$$

$$v = 2^{1/1024} = 1.000\,677\,131$$

Save both u and v in memory, if possible. If you can't store u and v, simply recompute them when needed. Now, perform the following two equivalent computations based on u:

$$x = \overbrace{\left(\left((u^2)^2\right)\cdots\right)^2}^{10 \text{ times}} = 1.999\,999\,963$$
$$x' = u^{1024} = 1.999\,999\,973$$

Similarly, perform the following two equivalent computations based on v:

$$y = \overbrace{\left(\left((v^2)^2\right)\cdots\right)^2}^{10 \text{ times}} = 1.999\,999\,983$$
$$y' = v^{1024} = 1.999\,999\,994$$

The four different values obtained for x, x', y, and y', in lieu of 2, hint that perhaps v and u are not really the same value. Let's compute their difference:

$$w = v - u = 1 \times 10^{-11}$$

Why isn't w equal to zero? The reason is that even though u and v are displayed identically, they in fact have different internal representations. Most calculators have hidden or guard digits (mine has two) to provide a higher degree of accuracy and to reduce the effect of accumulated errors when long computation sequences are performed.

Let's see if we can determine the hidden digits for the u and v values above. Here is one way:

$$(u - 1) \times 1000 = 0.677\,130\,680 \qquad [\text{Hidden} \cdots (0)\,68]$$
$$(v - 1) \times 1000 = 0.677\,130\,690 \qquad [\text{Hidden} \cdots (0)\,69]$$

This explains why w is not zero, which in turn tells us why $u^{1024} \neq v^{1024}$. The following simple analysis might be helpful in this regard.

$$v^{1024} = (u + 10^{-11})^{1024}$$
$$\approx u^{1024} + 1024 \times 10^{-11} u^{1023} \approx u^{1024} + 2 \times 10^{-8}$$

The difference between v^{1024} and u^{1024} is in good agreement with the result of the preceding analysis. The difference between $(((u^2)^2)\cdots)^2$ and u^{1024} exists because the former is computed through repeated multiplications while the latter uses the built-in exponentiation routine of the calculator, which is likely to be less precise.

Despite the discrepancies, the results of the foregoing computations are remarkably precise. The values of u and v agree to 11 decimal digits, while those of x, x', y, y' are identical to eight digits. This is better than single-precision, floating-point arithmetic on the most elaborate and expensive computers. Do we have a right to expect more from a calculator that costs $20 or less? Ease of use is, of course, a different matter from speed or precision. For a detailed exposition of some deficiencies in current calculators, and a refreshingly new design approach, see [Thim95].

1.3 NUMBERS AND THEIR ENCODINGS

Number representation methods have advanced in parallel with the evolution of language. The oldest method for representing numbers consisted of the use of stones or sticks. Gradually, as

larger numbers were needed, it became difficult to represent them or develop a feeling for their magnitudes. More importantly, comparing large numbers was quite cumbersome. Grouping the stones or sticks (e.g., representing the number 27 by 5 groups of 5 sticks plus 2 single sticks) was only a temporary cure. It was the use of different stones or sticks for representing groups of 5, 10, etc. that produced the first major breakthrough.

The latter method gradually evolved into a symbolic form whereby special symbols were used to denote larger units. A familiar example is the Roman numeral system. The units of this system are 1, 5, 10, 50, 100, 500, 1000, 10 000, and 100 000, denoted by the symbols I, V, X, L, C, D, M, ((I)), and (((I))), respectively. A number is represented by a string of these symbols, arranged in descending order of values from left to right. To shorten some of the cumbersome representations, allowance is made to count a symbol as representing a negative value if it is to the left of a larger symbol. For example, IX is used instead of VIIII to denote the number 9 and LD is used for CCCCL to represent the number 450.

Clearly, the Roman numeral system is not suitable for representing very large numbers. Furthermore, it is difficult to do arithmetic on numbers represented with this notation. The *positional* system of number representation was first used by the Chinese. In this method, the value represented by each symbol depends not only on its shape but also on its position relative to other symbols. Our conventional method of representing numbers is based on a positional system.

For example in the number 222, each of the "2" digits represents a different value. The leftmost 2 represents 200. The middle 2 represents 20. Finally, the rightmost 2 is worth 2 units. The representation of time intervals in terms of days, hours, minutes, and seconds (i.e., as four-element vectors) is another example of the positional system. For instance, in the vector $T = 5\ 5\ 5\ 5$, the leftmost element denotes 5 days, the second from the left represents 5 hours, the third element stands for 5 minutes, and the rightmost element denotes 5 seconds.

If in a positional number system, the unit corresponding to each position is a constant multiple of the unit for its right neighboring position, the conventional *fixed-radix* positional system is obtained. The decimal number system we use daily is a positional number system with 10 as its constant radix. The representation of time intervals, as just discussed, provides an example of a *mixed-radix* positional system for which the radix is the vector $R = 0\ 24\ 60\ 60$.

The method used to represent numbers affects not just the ease of reading and understanding numbers but also the complexity of arithmetic algorithms used for computing with numbers. The popularity of positional number systems is in part due to the availability of simple and elegant algorithms for performing arithmetic on such numbers. We will see in subsequent chapters that other representations provide advantages over the positional representation in terms of certain arithmetic operations or the needs of particular application areas. However, these systems are of limited use precisely because they do not support universally simple arithmetic.

In digital systems, numbers are encoded by means of binary digits or bits. Suppose you have 4 bits to represent numbers. There are 16 possible codes. You are free to assign the 16 codes to numbers as you please. However, since number representation has significant effects on algorithm and circuit complexity, only some of the wide range of possibilities have found applications.

To simplify arithmetic operations, including the required checking for singularities or special cases, the assignment of codes to numbers must be done in a logical and systematic manner. For example, if you assign codes to 2 and 3 but not to 5, then adding 2 and 3 will cause an "overflow" (yields an unrepresentable value) in your number system.

Figure 1.2 shows some examples of assignments of 4-bit codes to numbers. The first choice is to interpret the 4-bit patterns as 4-bit binary numbers, leading to the representation of natural numbers in the range [0, 15]. The signed-magnitude scheme results in integers in the range $[-7, 7]$ being represented, with 0 having two representations, (viz., ± 0). The 3-plus-1 fixed-point number system (3 whole bits, 1 fractional bit) gives us numbers from 0 to 7.5 in increments of 0.5. Viewing

the 4-bit codes as signed fractions gives us a range of $[-0.875, +0.875]$ or $[-1, +0.875]$, depending on whether we use signed-magnitude or 2's-complement representation.

The 2-plus-2 unsigned floating-point number system in Fig. 1.2, with its 2-bit exponent e in the range $[-2, 1]$ and 2-bit integer significand s in $[0, 3]$, can represent certain values $s \times 2^e$ in $[0, 6]$. In this system, 0.00 has four representations, 0.50, 1.00, and 2.00 have two representations each, and 0.25, 0.75, 1.50, 3.00, 4.00, and 6.00 are uniquely represented. The 2-plus-2 logarithmic number system, which represents a number by approximating its 2-plus-2, fixed-point, base-2 logarithm, completes the choices shown in Fig. 1.2.

1.4 FIXED-RADIX POSITIONAL NUMBER SYSTEMS

A conventional fixed-radix, fixed-point positional number system is usually based on a positive integer *radix* (base) r and an implicit digit set $\{0, 1, \cdots, r-1\}$. Each unsigned integer is represented by a digit vector of length $k + l$, with k digits for the whole part and l digits for the fractional part. By convention, the digit vector $x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l}$ represents the value:

$$(x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_r = \sum_{i=-l}^{k} x_i r^i$$

One can easily generalize to arbitrary radices (not necessarily integer or positive or constant) and digit sets of arbitrary size or composition. In what follows, we restrict our attention to digit sets composed of consecutive integers, since digit sets of other types complicate arithmetic and have no redeeming property. Thus, we denote our digit set by $\{-\alpha, -\alpha+1, \cdots, \beta-1, \beta\} = [-\alpha, \beta]$.

The following examples demonstrate the wide range of possibilities in selecting the radix and digit set.

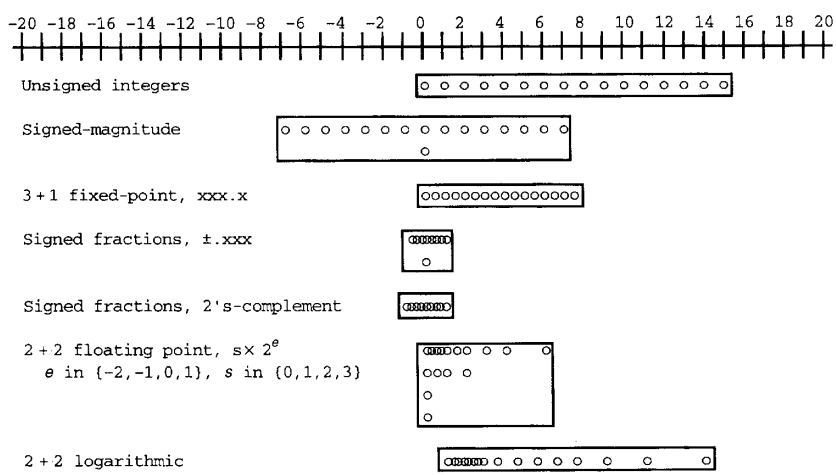


Fig. 1.2 Some of the possible ways of assigning 16 distinct codes to represent numbers.

Example 1.1 Balanced ternary number system: $r = 3$, digit set $= [-1, 1]$.

Example 1.2 Negative-radix number systems: radix $-r$, digit set $= [0, r-1]$.

$$(\cdots x_5 x_4 x_3 x_2 x_1 x_0 . x_{-1} x_{-2} x_{-3} x_{-4} x_{-5} x_{-6} \cdots)_{-r} = \sum_i x_i (-r)^i$$

$$= \sum_{\text{even}i} d_i r^i - \sum_{\text{odd}i} d_i r^i$$

$$= (\cdots 0 x_4 0 x_2 0 x_0 . 0 x_{-2} 0 x_{-4} 0 x_{-6} \cdots)_r - (\cdots x_5 0 x_3 0 x_1 0 . x_{-1} 0 x_{-3} 0 x_{-5} 0 \cdots)_r$$

The special case with $r = -2$ and digit set of $[0, 1]$ is known as the negabinary number system.

Example 1.3 Nonredundant signed-digit number systems: digit set $[-\alpha, r-1-\alpha]$ for radix r. As an example, one can use the digit set $[-4, 5]$ for $r = 10$. We denote a negative digit by preceding it with a minus sign, as usual, or by using a hyphen as a left superscript when the minus sign could be mistaken for subtraction. For example,

$(3\ {}^{-}1\ 5)_{\text{ten}}$ represents the decimal number $295 = 300 - 10 + 5$

$({}^{-}3\ 1\ 5)_{\text{ten}}$ represents the decimal number $-285 = -300 + 10 + 5$

Example 1.4 Redundant signed-digit number systems: digit set $[-\alpha, \beta]$, with $\alpha + \beta \geq r$ for radix r. One can use the digit set $[-7, 7]$, say, for $r = 10$. In such redundant number systems, certain values may have multiple representations. For example, here are some representations for the decimal number 295:

$$(3\ {}^{-}1\ 5)_{\text{ten}} = (3\ 0\ {}^{-}5)_{\text{ten}} = (1\ {}^{-}7\ 0\ {}^{-}5)_{\text{ten}}$$

We will study redundant representations in detail in Chapter 3.

Example 1.5 Fractional radix number systems: $r = 0.1$ with digit set $[0, 9]$.

$$(x_{k-1} x_{k-2} \cdots x_1 x_0 . x_{-1} x_{-2} \cdots x_{-l})_{\text{one-tenth}} = \sum_i x_i 10^{-i}$$

$$= (x_{-l} \cdots x_{-2} x_{-1} x_0 . x_1 x_2 \cdots x_{k-2} x_{k-1})_{\text{ten}}$$

■ **Example 1.6** Irrational radix number systems: $r = \sqrt{2}$ with digit set [0, 1].

$$(\cdots x_5 x_4 x_3 x_2 x_1 x_0 . x_{-1} x_{-2} x_{-3} x_{-4} x_{-5} x_{-6} \cdots)_{\sqrt{2}} = \sum_i x_i (\sqrt{2})^i$$

$$= (\cdots x_4 x_2 x_0 . x_{-2} x_{-4} x_{-6} \cdots)_{\text{two}} + \sqrt{2}(\cdots x_5 x_3 x_1 . x_{-1} x_{-3} x_{-5} \cdots)_{\text{two}}$$

These examples illustrate the generality of our definition by introducing negative, fractional, and irrational radices and by using both nonredundant or minimal (r different digit values) and redundant ($> r$ digit values) digit sets in the common case of positive integer radices. We can go even further and make the radix an imaginary or complex number.

■ **Example 1.7** Complex-radix number systems: the quater-imaginary number system uses $r = 2j$, where $j = \sqrt{-1}$, and the digit set [0, 3].

$$(\cdots x_5 x_4 x_3 x_2 x_1 x_0 . x_{-1} x_{-2} x_{-3} x_{-4} x_{-5} x_{-6} \cdots)_{2j} = \sum_i x_i (2j)^i$$

$$= (\cdots x_4 x_2 x_0 . x_{-2} x_{-4} x_{-6} \cdots)_{-\text{four}} + 2j(\cdots x_5 x_3 x_1 . x_{-1} x_{-3} x_{-5} \cdots)_{-\text{four}}$$

It is easy to see that any complex number can be represented in the quater-imaginary number system of Example 1.7, with the advantage that ordinary addition (with a slightly modified carry rule) and multiplication can be used for complex-number computations. The modified carry rule is that a carry of -1 (a borrow actually) goes two positions to the left when the position sum, or digit total in a given position, exceeds 3.

In radix r, with the standard digit set $[0, r-1]$, the number of digits needed to represents the natural numbers in $[0, max]$ is:

$$k = \lfloor \log_r max \rfloor + 1 = \lceil \log_r (max + 1) \rceil$$

Note that the number of different values represented is $max + 1$.

With fixed-point representation using k whole and l fractional digits, we have:

$$max = r^k - r^{-l} = r^k - ulp$$

We will find the term *ulp*, for unit in least (significant) position, quite useful in describing certain arithmetic concepts without distinguishing between integers and fixed-point representations that include fractional parts. For integers, $ulp = 1$.

Specification of time intervals in terms of weeks, days, hours, minutes, seconds, and milliseconds is an example of mixed-radix representation. Given the two-part radix vector $\cdots r_3 r_2 r_1 r_0 ; r_{-1} r_{-2} \cdots$ defining the mixed radix, the two-part digit vector $\cdots x_3 x_2 x_1 x_0 ; x_{-1} x_{-2} \cdots$ represents the value

$$\cdots x_3 r_2 r_1 r_0 + x_2 r_1 r_0 + x_1 r_0 + x_0 + \frac{x_{-1}}{r_{-1}} + \frac{x_{-2}}{r_{-1} r_{-2}} + \cdots$$

In the time interval example, the mixed radix is $\cdots 7, 24, 60, 60 ; 1000 \cdots$ and the digit vector 3, 2, 9, 22, 57; 492 (3 weeks, 2 days, 9 hours, 22 minutes, 57 seconds, and 492 milliseconds) represents

$$(3 \times 7 \times 24 \times 60 \times 60) + (2 \times 24 \times 60 \times 60) + (9 \times 60 \times 60) + (22 \times 60) + 57 + 492/1000$$
$$= 2\,020\,977.492 \text{ seconds}$$

In Chapter 4, we will see that mixed-radix representation plays an important role in dealing with values represented in residue number systems.

1.5 NUMBER RADIX CONVERSION

Assuming that the unsigned value u has exact representations in radices r and R, we can write:

$$\begin{aligned} u &= w.v \\ &= (x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_r \\ &= (X_{K-1}X_{K-2}\cdots X_1X_0.X_{-1}X_{-2}\cdots X_{-L})_R \end{aligned}$$

If an exact representation does not exist in one or both of the radices, the foregoing equalities will be approximate.

The radix conversion problem is defined as follows:

Given	r	the old radix,
	R	the new radix, and the
	x_is	digits in the radix-r representation of u
find the	X_is	digits in the radix-R representation of u

In the rest of this section, we will describe two methods for radix conversion based on doing the arithmetic in the old radix r or in the new radix R. We will also present a shortcut method, involving very little computation, that is applicable when the old and new radices are powers of the same number (e.g., 8 and 16, which are both powers of 2).

Note that in converting u from radix r to radix R, where r and R are positive integers, we can convert the whole and fractional parts separately. This is because an integer (fraction) is an integer (fraction), independent of the number representation radix.

Doing the arithmetic in the old radix *r*

We use this method when radix-r arithmetic is more familiar or efficient. The method is useful, for example, when we do manual computations and the old radix is $r = 10$. The procedures for converting the whole and fractional parts, along with their justifications or proofs, are given below.

Converting the whole part w

Procedure: Repeatedly divide the integer $w = (x_{k-1}x_{k-2}\cdots x_1x_0)_r$ by the radix-r representation of R. The remainders are the X_is, with X_0 generated first.

Justification: $(X_{K-1}X_{K-2}\cdots X_1X_0)_R - (X_0)_R$ is divisible by R. Therefore, X_0 is the remainder of dividing the integer $w = (x_{k-1}x_{k-2}\cdots x_1x_0)_r$ by the radix-r representation of R.

Example: $(105)_{\text{ten}} = (?)_{\text{five}}$
Repeatedly divide by 5:

Quotient	Remainder
105	0
21	1
4	4
0	

From the above, we conclude that $(105)_{\text{ten}} = (410)_{\text{five}}$.

Converting the fractional part v

Procedure: Repeatedly multiply the fraction $v = (.x_{-1}x_{-2}\cdots x_{-l})_r$ by the radix-r representation of R. In each step, remove the whole part before multiplying again. The whole parts obtained are the X_is, with X_{-1} generated first.

Justification: $R \times (0.X_{-1}X_{-2}\cdots X_{-L})_R = (X_{-1}.X_{-2}\cdots X_{-L})_R$.

Example: $(105.486)_{\text{ten}} = (410.?)_{\text{five}}$
Repeatedly multiply by 5:

Whole part	Fraction
	.486
2	.430
2	.150
0	.750
3	.750
3	.750

From the above, we conclude that $(105.486)_{\text{ten}} \approx (410.22033)_{\text{five}}$.

Doing the arithmetic in the new radix *R*

We use this method when radix-R arithmetic is more familiar or efficient. The method is useful, for example, when we manually convert numbers to radix 10. Again, the whole and fractional parts are converted separately.

Converting the whole part w

Procedure: Use repeated multiplications by r followed by additions according to the formula $((\cdots((x_{k-1}r + x_{k-2})r + x_{k-3})r + \cdots)r + x_1)r + x_0$.

Justification: The given formula is the well-known Horner's method (or rule), first presented in the early nineteenth century, for the evaluation of the $(k-1)$th-degree polynomial $x_{k-1}r^{k-1} + x_{k-2}r^{k-2} + \cdots + x_1r + x_0$ [Knut97].

Example: $(410)_{\text{five}} = (?)_{\text{ten}}$

$$((4 \times 5) + 1) \times 5 + 0 = 105 \Rightarrow (410)_{\text{five}} = (105)_{\text{ten}}$$

Converting the fractional part v

Procedure: Convert the integer $r^l(0.v)$ and then divide by r^l in the new radix.

Justification: $r^l(0.v)/r^l = 0.v$

Example: $(410.220\ 33)_{\text{five}} = (105.?)_{\text{ten}}$

$$(0.220\ 33)_{\text{five}} \times 5^5 = (22\ 033)_{\text{five}} = (1518)_{\text{ten}}$$

$$1518/5^5 = 1518/3125 = 0.485\ 76$$

From the above, we conclude that $(410.22033)_{\text{five}} = (105.48576)_{\text{ten}}$.

Note: Horner's method works here as well but is generally less practical. The digits of the fractional part are processed from right to left and the multiplication operation is replaced with division. Figure 1.3 shows how Horner's method can be applied to the preceding example.

Shortcut method for $r = b^g$ and $R = b^G$

In the special case when the old and new radices are integral powers of a common base b, that is, $r = b^g$ and $R = b^G$, one can convert from radix r to radix b and then from radix b to radix R. Both these conversions are quite simple and require virtually no computation.

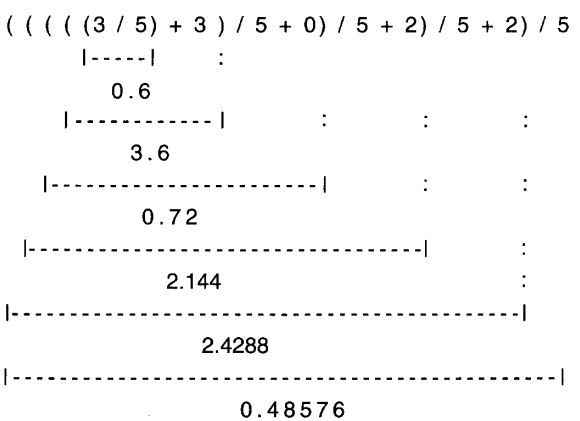


Fig. 1.3 Horner's rule used to convert $(.22033)_{\text{five}}$ to decimal.

To convert from the old radix $r = b^g$ to radix b, simply convert each radix-r digit individually into a g-digit radix-b number and then juxtapose the resulting g-digit numbers.

To convert from radix b to the new radix $R = b^G$, form G-digit groups of the radix-b digits starting from the radix point (to the left and to the right). Then convert the G-digit radix-b number of each group into a single radix-R digit and juxtapose the resulting digits.

Example 1.8 $(2\,301.302)_{\text{four}} = (?)_{\text{eight}}$

We have $4 = 2^2$ and $8 = 2^3$. Thus, conversion through the intermediate radix 2 is used. Each radix-4 digit is independently replaced by a 2-bit radix-2 number. This is followed by 3-bit groupings of the resulting binary digits to find the radix-8 digits.

$$\begin{aligned}(2\,301.302)_{\text{four}} &= (\underset{2}{10}\ \underset{3}{11}\ \underset{0}{00}\ \underset{1}{01}\ .\ \underset{3}{11}\ \underset{0}{00}\ \underset{2}{10})_{\text{two}} \\ &= (\underset{2}{10}\ \underset{6}{110}\ \underset{1}{001}\ .\ \underset{6}{110}\ \underset{2}{010})_{\text{two}} = (261.62)_{\text{eight}}\end{aligned}$$

Clearly, when $g = 1 (G = 1)$, the first (second) step of the shortcut conversion procedure is eliminated. This corresponds to the special case of $R = r^G (r = R^g)$. For example, conversions between radix 2 and radix 8 or 16 belong to these special cases.

1.6 CLASSES OF NUMBER REPRESENTATIONS

In Sections 1.4 and 1.5, we considered the representation of unsigned fixed-point numbers using fixed-radix number systems, with standard and nonstandard digit sets, as well as methods for converting between such representations with standard digit sets. In digital computations, we also deal with signed fixed-point numbers as well as signed and unsigned real values. Additionally, we may use unconventional representations for the purpose of speeding up arithmetic operations or increasing their accuracy. Understanding different ways of representing numbers, including their relative cost–performance benefits and conversions between various representations, is an important prerequisite for designing efficient arithmetic algorithms or circuits.

In the next three chapters, we will review techniques for representing fixed-point numbers, beginning with conventional methods and then moving on to some unconventional representations.

Signed fixed-point numbers, including various ways of representing and handling the sign information, are covered in Chapter 2. Signed-magnitude, biased, and complement representations (including both 1's- and 2's-complement) are covered in some detail.

The signed-digit number systems of Chapter 3 can also be viewed as methods for representing signed numbers, although their primary significance lies in the redundancy that allows addition without carry propagation. The material in Chapter 3 is essential for understanding several speedup methods in multiplication, division, and function evaluation.

Chapter 4 introduces residue number systems (for representing unsigned or signed integers) that allow some arithmetic operations to be performed in a truly parallel fashion at very high speed. Unfortunately, the difficulty of division and certain other arithmetic operations renders

these number systems unsuitable for general applications. In Chapter 4, we also use residue representations to explore the limits of fast arithmetic.

Representation of real numbers can take different forms. Examples include slash number systems (for representing rational numbers), logarithmic number systems (for representing real values), and of course, floating-point numbers that constitute the primary noninteger data format in modern digital systems. These representations are discussed in Chapter 17 (introductory chapter of Part V), immediately before we deal with algorithms, hardware implementations, and error analyses for real-number arithmetic.

By combining features from two or more of the aforementioned "pure" representations, we can obtain many hybrid schemes. Examples include hybrid binary/signed-digit (see Section 3.4), hybrid residue/binary (see Section 4.5), hybrid logarithmic/signed-digit (see Section 17.6), and hybrid floating-point/logarithmic (see Problem 17.16) representations.

PROBLEMS

1.1 Arithmetic algorithms Consider the integral $I_n = \int_0^1 x^n e^{-x}\, dx$ that has the exact solution $n![1 - (1/e)\sum_{r=0}^{n} 1/r!]$. The integral can also be computed based on the recurrence equation $I_n = nI_{n-1} - 1/e$ with $I_0 = 1 - 1/e$.

- **a.** Prove that the recurrence equation is correct.
- **b.** Use a calculator or write a program to compute the values of I_j for $1 \leq j \leq 20$.
- **c.** Repeat part b with a different calculator or with a different precision in your program.
- **d.** Compare your results to the exact value $I_{20} = 0.018\ 350\ 468$ and explain any difference.

1.2 Arithmetic algorithms Consider the sequence $\{u_i\}$ defined by the recurrence $u_{i+1} = iu_i - i$, with $u_1 = e$.

- **a.** Use a calculator or write a program to determine the values of u_i for $1 \leq i \leq 25$.
- **b.** Repeat part a with a different calculator or with a different precision in your program.
- **c.** Explain the results.

1.3 Arithmetic algorithms Consider the sequence $\{a_i\}$ defined by the recurrence $a_{i+2} = 111 - 1130/a_{i+1} + 3000/(a_{i+1}a_i)$, with $a_0 = 11/2$ and $a_1 = 61/11$. The exact limit of this sequence is 6; but on any real machine, a different limit is obtained. Use a calculator or write a program to determine the values of a_i for $2 \leq i \leq 25$. What limit do you seem to be getting? Explain the outcome.

1.4 Positional representation of the integers

- **a.** Prove that an unsigned binary integer x is a power of 2 if and only if the bitwise logical AND of x and $x - 1$ is 0.
- **b.** Prove that an unsigned radix-3 integer $x = (x_{k-1}x_{k-2}\cdots x_1x_0)_{\text{three}}$ is even if and only if $\sum_{i=0}^{k-1} x_i$ is even.
- **c.** Prove that an unsigned binary integer $x = (x_{k-1}x_{k-2}\cdots x_1x_0)_{\text{two}}$ is divisible by 3 if and only if $\sum_{\text{even}i} x_i - \sum_{\text{odd}_i} x_i$ is a multiple of 3.
- **d.** Generalize the statements of parts b and c to obtain rules for divisibility of radix-r integers by $r - 1$ and $r + 1$.

1.5 Unconventional radices

a. Convert the negabinary number $(0001\ 1111\ 0010\ 1101)_{-two}$ to radix 16 (hexadecimal).

b. Repeat part a for radix -16 (negahexadecimal).

c. Derive a procedure for converting numbers from radix r to radix $-r$ and vice versa.

1.6 Unconventional radices Consider the number x whose representation in radix $-r$ (with r a positive integer) is the $(2k + 1)$-element all-1s vector.

a. Find the value of x in terms of k and r.

b. Represent $-x$ in radix $-r$ (negation or sign change).

c. Represent x in the positive radix r.

d. Represent $-x$ in the positive radix r.

1.7 Unconventional radices Let θ be a number in the negative radix $-r$ whose digits are all $r - 1$. Show that $-\theta$ is represented by a vector of all 2s, except for its most- and least-significant digits, which are 1s.

1.8 Unconventional radices Consider a fixed-radix positional number system with the digit set $[-2, 2]$ and the imaginary radix $r = 2j (j = \sqrt{-1})$.

a. Describe a simple procedure to determine whether a number thus represented is real.

b. Show that all integers are representable and that some integers have multiple representations.

c. Can this system represent any complex number with integral real and imaginary parts?

d. Describe simple procedures for finding the representations of $a - bj$ and $4(a + bj)$, given the representation of $a + bj$.

1.9 Unconventional radices Consider the radix $r = -1 + j (j = \sqrt{-1})$ with the digit set $[0, 1]$.

a. Express the complex number $-49 + j$ in this number system.

b. Devise a procedure for determining whether a given bit string represents a real number.

c. Show that any natural number is representable with this number system.

1.10 Number radix conversion

a. Convert the following octal (radix-8) numbers to hexadecimal (radix-16) notation: 12, 5 655, 2 550 276, 76 545 336, 3 726 755

b. Represent $(48A.C2)_{sixteen}$ and $(192.837)_{ten}$ in radices 2, 8, 10, 12, and 16.

c. Outline procedures for converting an unsigned radix-r number, using the standard digit set $[0, r - 1]$, into radices $1/r$, $\sqrt{r}$ and $j\sqrt[4]{r} (j = \sqrt{-1})$, using the same digit set.

1.11 Number radix conversion Consider a fixed-point, radix-4 number system in which a number x is represented with k whole and l fractional digits.

a. Assuming the use of standard radix-4 digit set [0, 3] and radix-8 digit set [0, 7], determine K and L, the numbers of whole and fractional digits in the radix-8 representation of x as functions of k and l.

b. Repeat part a for the more general case in which the radix-4 and radix-8 digit sets are $[-\alpha, \beta]$ and $[-2\alpha, 2\beta]$, respectively, with $\alpha \geq 0$ and $\beta \geq 0$.

1.12 Number radix conversion Dr. N. E. Patent, a frequent contributor to scientific journals, claims to have invented a simple logic circuit for conversion of numbers from radix 2 to radix 10. The novelty of this circuit is that it can convert arbitrarily long numbers. The binary number is input one bit at a time. The decimal output will emerge one digit at a time after a fixed initial delay that is independent of the length of the input number. Evaluate this claim using only the information given.

1.13 Fixed-point number representation Consider a fixed-point, radix-3 number system, using the digit set $[-1, 1]$, in which numbers are represented with k integer digits and l fractional digits as: $d_{k-1}d_{k-2}\cdots d_1 d_0 . d_{-1}d_{-2}\cdots d_{-l}$

a. Determine the range of numbers represented as a function of k and l.

b. Given that each radix-3 digit needs a 2-bit encoding, compute the representation efficiency of this number system relative to the binary representation.

c. Outline a carry-free procedure for converting one of the above radix-3 numbers to an equivalent radix-3 number using the redundant digit set [0, 3]. By a carry-free procedure, we mean a procedure that determines each digit of the new representation locally from a few neighboring digits of the original representation, so that the speed of the circuit is independent of the length of the original number.

1.14 Number radix conversion Discuss the design of a hardware number radix converter that receives its radix-r input digit-serially and produces the radix-R output ($R > r$) in the same manner. Multiple conversions are to be performed continuously; that is, once the last digit of one number has been input, the presentation of the second number can begin with no time gap [Parh92].

1.15 Decimal-to-binary conversion Consider a $2k$-bit register, the upper half of which holds a decimal number, with each digit encoded as a 4-bit binary number (binary-coded decimal or BCD). Show that repeating the following steps k times will yield the binary equivalent of the decimal number in the lower half of the $2k$-bit register: Shift the $2k$-bit register one bit to the right; independently subtract 3 units from each 4-bit segment of the upper half whose binary value equals or exceeds 8 (there are $k/4$ such 4-bit segments).

1.16 Design of comparators An h-bit comparator is a circuit with two h-bit unsigned binary inputs, x and y, and two binary outputs designating the conditions $x < y$ and $x > y$. Sometimes a third output corresponding to $x = y$ is also provided, but we do not need it for this problem.

a. Present the design of a 4-bit comparator.

b. Show how five 4-bit comparators can be cascaded to compare two 16-bit numbers.

c. Show how a three-level tree of 4-bit comparators can be used to compare two 28-bit numbers. Try to use as few 4-bit comparator blocks as possible.

d. Generalize the result of part b to derive a synthesis method for large comparators built from a cascaded chain of smaller comparators.

e. Generalize the result of part c to derive a synthesis method for large comparators built from a tree of smaller comparators.

REFERENCES

[Knut97] Knuth, D. E., *The Art of Computer Programming*, 3rd ed., Vol. 2: *Seminumerical Algorithms*, Addison-Wesley, 1997.

[Mole95] Moler, C., "A Tale of Two Numbers," *SIAM News*, Vol. 28, No. 1, pp. 1, 16, 1995.

[Parh92] Parhami, B., "Systolic Number Radix Converters," *Computer J.*, Vol. 35, No. 4, pp. 405–409, August 1992.

[Scot85] Scott, N. R., *Computer Number Systems and Arithmetic*, Prentice-Hall, 1985.

[Thim95] Thimbleby, H., "A New Calculator and Why It Is Necessary," *Computer J.*, Vol. 38, No. 6, pp. 418–433, 1995.

Chapter 2 REPRESENTING SIGNED NUMBERS

This chapter deals with the representation of signed fixed-point numbers by providing an attached sign bit, adding a fixed bias to all numbers, complementing negative values, attaching signs to digit positions, or using signed digits. In view of its importance in the design of fast arithmetic algorithms and hardware, representing signed fixed-point numbers by means of signed digits is further explored in Chapter 3. Chapter topics include:

2.1 SIGNED-MAGNITUDE REPRESENTATION

The natural numbers $0, 1, 2, \cdots, max$ can be represented as fixed-point numbers without fractional parts (refer to Section 1.4). In radix r, the number k of digits needed for representing the natural numbers up to max is

$$k = \lfloor \log_r max \rfloor + 1 = \lceil \log_r (max + 1) \rceil$$

Conversely, with k digits, one can represent the values 0 through $r^k - 1$, inclusive; that is, the interval $[0, r^k - 1] = [0, r^k)$ of natural numbers.

Natural numbers are often referred to as "unsigned integers," which form a special data type in many programming languages and computer instruction sets. The advantage of using this data type as opposed to "integers" when the quantities of interest are known to be nonnegative is that a larger representation range can be obtained (e.g., maximum value of 255, rather than 127, with 8 bits).

One way to represent both positive and negative integers is to use "signed magnitudes," or the sign-and-magnitude format, in which one bit is devoted to sign. The common convention is

to let 1 denote a negative sign and 0 a positive sign. In the case of radix-2 numbers with a total length of k bits, $k-1$ bits will be available to represent the magnitude or absolute value of the number. The range of k-bit signed-magnitude binary numbers is thus $[-(2^{k-1}-1), 2^{k-1}-1]$. Figure 2.1 depicts the assignment of values to bit patterns for a 4-bit signed-magnitude format.

Advantages of signed-magnitude representation include its intuitive appeal, conceptual simplicity, symmetric range, and simple negation (sign change) by flipping or inverting the sign bit. The primary disadvantage is that addition of numbers with unlike signs (subtraction) must be handled differently from that of same-sign operands.

The hardware implementation of an adder for signed-magnitude numbers either involves a magnitude comparator and a separate subtractor circuit or else is based on the use of complement representation (see Section 2.3) internally within the arithmetic/logic unit (ALU). In the latter approach, a negative operand is complemented at the ALU's input, the computation is done by means of complement representation, and the result is complemented, if necessary, to produce the signed-magnitude output. Because the pre- and postcomplementation steps add to the computation delay, it is better to use the complement representation throughout.

Besides the aforementioned extra delay in addition and subtraction, signed-magnitude representation allows two representations for 0, leading to the need for special care in number comparisons or added overhead for detecting -0 and changing it to $+0$. This drawback, however, is unavoidable in any radix-2 number representation system with symmetric range.

Figure 2.2 shows the hardware implementation of signed-magnitude addition using selective pre- and postcomplementation. The control circuit receives as inputs the operation to be performed (0 = add, 1 = subtract), the signs of the two operands x and y, the carry-out of the adder, and the sign of the addition result. It produces signals for the adder's carry-in, complementation of x, complementation of the addition result, and the sign of the result. Note that complementation hardware is provided only for the x operand. This is because $x - y$ can be obtained by first computing $y - x$ and then changing the sign of the result. You will understand this design much better after we have covered complement representations of negative numbers in Sections 2.3 and 2.4.

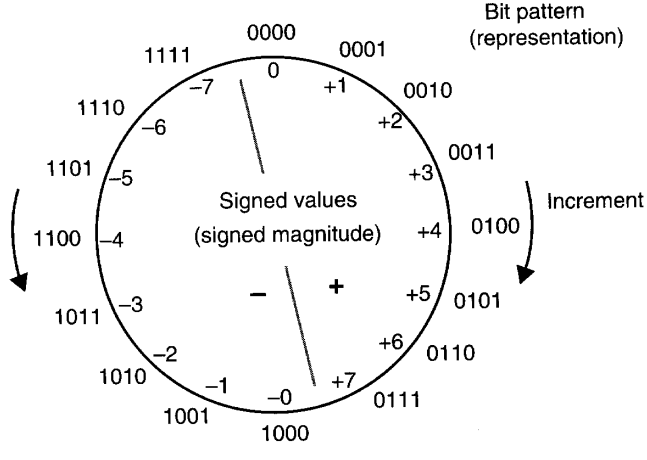


Fig. 2.1 A 4-bit signed-magnitude number representation system for integers.

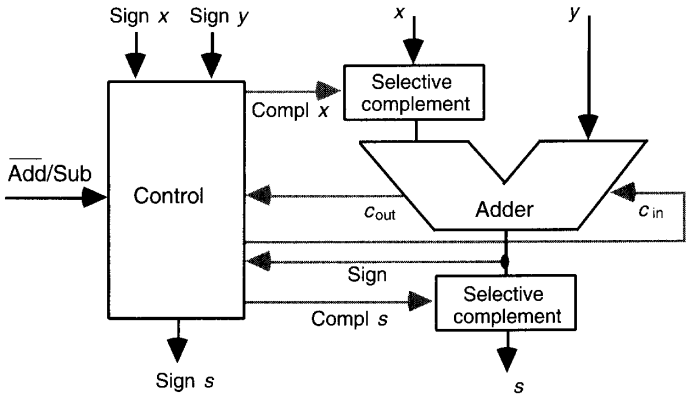


Fig. 2.2 Adding signed-magnitude numbers using pre-complementation and postcomplementation.

2.2 BIASED REPRESENTATIONS

One way to deal with signed numbers is to devise a representation or coding scheme that converts signed numbers into unsigned numbers. For example, the biased representation is based on adding a positive value *bias* to all numbers, allowing us to represents the integers from *–bias* to *max – bias* using unsigned values from 0 to *max*. Such a representation is sometimes referred to as "excess-*bias*" (e.g., excess-3 or excess-128) coding. We will see in Chapter 17 that biased representation is used to encode the exponent part of a floating-point number.

Figure 2.3 shows how signed integers in the range [−8, +7] can be encoded as unsigned values 0 through 15 by using a bias of 8. With k-bit representations and a bias of 2^{k-1}, the leftmost bit indicates the sign of the value represented (0 = negative, 1 = positive). Note that this is the opposite of the commonly used convention for number signs. With a bias of 2^k or $2^k - 1$, the range of represented integers is almost symmetric.

Biased representation does not lend itself to simple arithmetic algorithms. Addition and subtraction become somewhat more complicated because one must subtract or add the bias from/to the result of a normal add/subtract operation, since:

$$x + y + bias = (x + bias) + (y + bias) - bias$$
$$x - y + bias = (x + bias) - (y + bias) + bias$$

With k-bit numbers and a bias of 2^{k-1}, adding or subtracting the bias amounts to complementing the leftmost bit. Thus, the extra complexity in addition or subtraction is negligible.

Multiplication and division become significantly more difficult if these operations are to be performed directly on biased numbers. For this reason, the practical use of biased representation is limited to the exponent parts of floating-point numbers, which are never multiplied or divided.

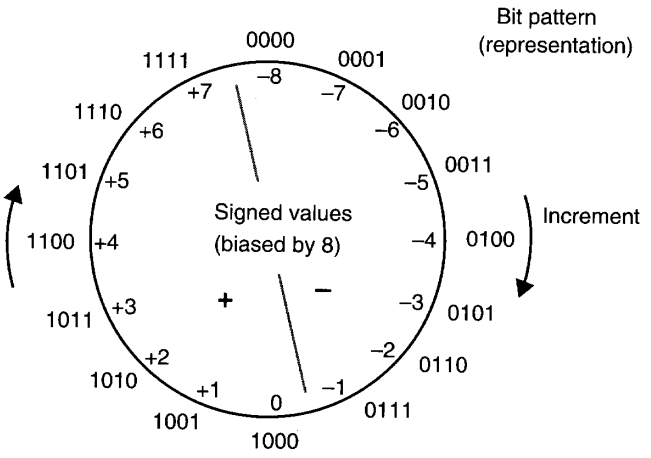


Fig. 2.3 A 4-bit biased integer number representation system with a bias of 8.

2.3 COMPLEMENT REPRESENTATIONS

In a complement number representation system, a suitably large complementation constant M is selected and the negative value $-x$ is represented as the unsigned value $M - x$. Figure 2.4 depicts the encodings used for positive and negative values and the arbitrary boundary between the two regions.

To represent integers in the range $[-N, +P]$ unambiguously, the complementation constant M must satisfy $M \geq N + P + 1$. This is justified by noting that to prevent overlap between the

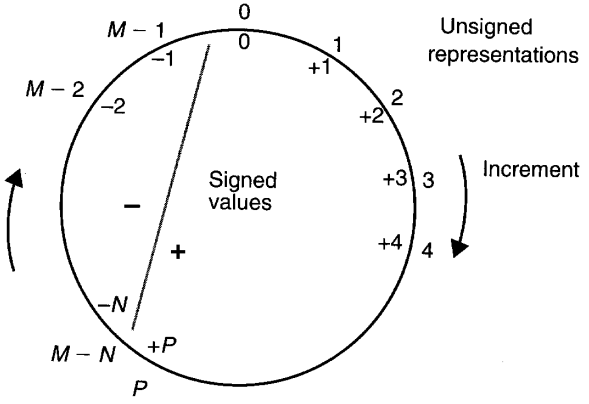


Fig. 2.4 Complement representation of signed integers.

representations of positive and negative values in Figure 2.4, we must have $M - N > P$. The choice of $M = N + P + 1$ yields maximum coding efficiency, since no code will go to waste.

In a complement system with the complementation constant M and the number representation range $[-N, +P]$, addition is done by adding the respective unsigned representations (modulo M). The addition process is thus always the same, independent of the number signs. This is easily understood if we note that in modulo-M arithmetic adding $M - 1$ (e.g.), is the same as subtracting 1. Table 2.1 shows the addition rules for complement representations, along with conditions that lead to overflow.

Subtraction can be performed by complementing the subtrahend and then performing addition. Thus, assuming that a selective complementer is available, addition and subtraction become essentially the same operation, and this is the primary advantage of complement representations.

Complement representation can be used for fixed-point numbers that have a fractional part. The only difference is that consecutive values in the circular representation of Fig. 2.4 will be separated by *ulp* instead of by 1. As a decimal example, given the complementation constant $M = 12.000$ and a fixed-point number range of $[-6.000, +5.999]$, the fixed-point number -3.258 has the complement representation $12.000 - 3.258 = 8.742$.

We note that two auxiliary operations are required for complement representations to be effective: complementation or change of sign (computing $M - x$) and computations of residues mod M. If finding $M - x$ requires subtraction and finding residues mod M implies division, then complement representation becomes quite inefficient. Thus M must be selected such that these two operations are simplified. Two choices allow just this for fixed-point radix-r arithmetic with k whole digits and l fractional digits:

Radix complement	$M = r^k$
Digit or diminished-radix complement	$M = r^k - ulp$

For radix-complement representations, modulo-M reduction is done by ignoring the carry-out from digit position $k-1$ in a $(k+l)$-digit radix-r addition. For digit-complement representations, computing the complement of x (i.e., $M - x$), is done by simply replacing each nonzero digit x_i by $r - 1 - x_i$. This is particularly easy if r is a power of 2. Complementation with $M = r^k$ and mod-M reduction with $M = r^k - ulp$ are similarly simple. You should be able to supply the details for radix r after reading Section 2.4, which deals with the important special case of $r = 2$.

TABLE 2.1
Addition in a complement number system with the complementation constant M and range $[-N, +P]$.

Desired operation	Computation to be performed mod M	Correct result with no overflow	Overflow condition
$(+x) + (+y)$	$x + y$	$x + y$	$x + y > P$
$(+x) + (-y)$	$x + (M - y)$	$x - y$ if $y \leq x$ $M - (y - x)$ if $y > x$	N/A
$(-x) + (+y)$	$(M - x) + y$	$y - x$ if $x \leq y$ $M - (x - y)$ if $x > y$	N/A
$(-x) + (-y)$	$(M - x) + (M - y)$	$M - (x + y)$	$x + y > N$

2.4 TWO'S- AND 1'S-COMPLEMENT NUMBERS

In the special case of $r = 2$, the radix complement representation that corresponds to $M = 2^k$ is known as *two's complement*. Figure 2.5 shows the 4-bit, 2's-complement integer system ($k = 4, l = 0, M = 2^4 = 16$) and the meanings of the 16 representations allowed with 4 bits. The boundary between positive and negative values is drawn approximately in the middle to make the range roughly symmetric and to allow simple sign detection (the leftmost bit is the sign).

The 2's complement of a number x can be found via bitwise complementation of x and the addition of *ulp*:

$$2^k - x = [(2^k - ulp) - x] + ulp = x^{\text{compl}} + ulp$$

Note that the binary representation of $2^k - ulp$ consists of all 1s, making $(2^k - ulp) - x$ equivalent to the bitwise complement of x, denoted as x^{compl}. Whereas finding the bitwise complement of x is easy, adding *ulp* to the result is a slow process, since in the worst case it involves full carry propagation. We will see later how this addition of *ulp* can usually be avoided.

To add numbers modulo 2^k, we simply drop a carry-out of 1 produced by position $k - 1$. Since this carry is worth 2^k units, dropping it is equivalent to reducing the magnitude of the result by 2^k.

The range of representable numbers in a 2's-complement number system with k whole bits is:

$$\text{from} \quad -2^{k-1} \quad \text{to} \quad 2^{k-1} - ulp$$

Because of this slightly asymmetric range, complementation can lead to overflow! Thus, if complementation is done as a separate sign change operation, it must include overflow detection.

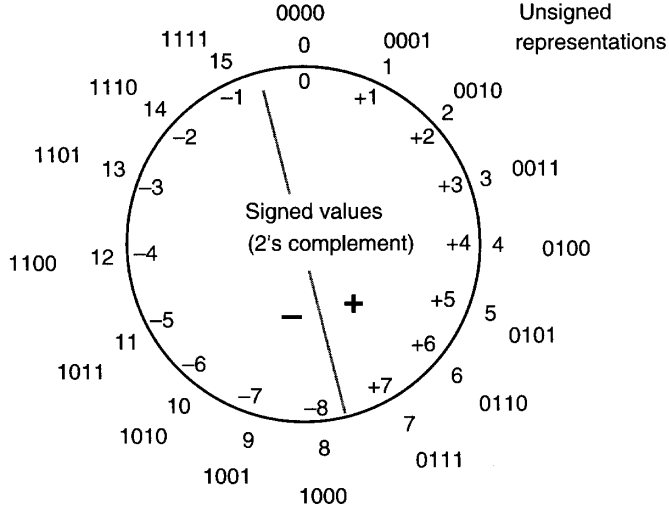


Fig. 2.5 A 4-bit, 2's-complement number representation system for integers.

However, we will see later that complementation needed to convert subtraction into addition requires no special provision.

The name "2's complement" actually comes from the special case of $k = 1$ that leads to the complementation constant $M = 2$. In this case, represented numbers have one whole bit, which acts as the sign, and l fractional bits. Thus, fractional values in the range $[-1, 1 - ulp]$ are represented in such a fractional 2's-complement number system.

The digit or diminished-radix complement representation is known as *one's complement* in the special case of $r = 2$. The complementation constant in this case is $M = 2^k - ulp$. For example, Fig. 2.6 shows the 4-bit, 1's-complement integer system ($k = 4, l = 0, M = 2^4 - 1 = 15$) and the meanings of the 16 representations allowed with 4 bits. The boundary between positive and negative values is again drawn approximately in the middle to make the range symmetric and to allow simple sign detection (the leftmost bit is the sign).

Note that compared to the 2's-complement representation of Fig. 2.5, the representation for -8 has been eliminated and instead an alternate code has been assigned to 0 (technically, -0). This may somewhat complicate 0 detection in that both the all-0s and the all-1s patterns represent 0. The arithmetic circuits can be designed such that the all-1s pattern is detected and automatically converted to the all-0s pattern. Keeping -0 intact does not cause problems in computations, however, since all computations are modulo 15. For example, adding $+1$ (0001) to -0 (1111) will yield the correct result of $+1$ (0001) when the addition is done modulo 15.

The 1's-complement of a number x can be found by bitwise complementation:

$$(2^k - ulp) - x = x^{\text{compl}}$$

To add numbers modulo $2^k - ulp$, we simply drop a carry-out of 1 produced by position $k - 1$ and simultaneously insert a carry-in of 1 into position $-l$. Since the dropped carry is worth 2^k units and the inserted carry is worth ulp, the combined effect is to reduce the magnitude of the result by $2^k - ulp$. In terms of hardware, the carry-out of our $(k + l)$-bit adder should be directly connected to its carry-in; this is known as *end-around carry*.

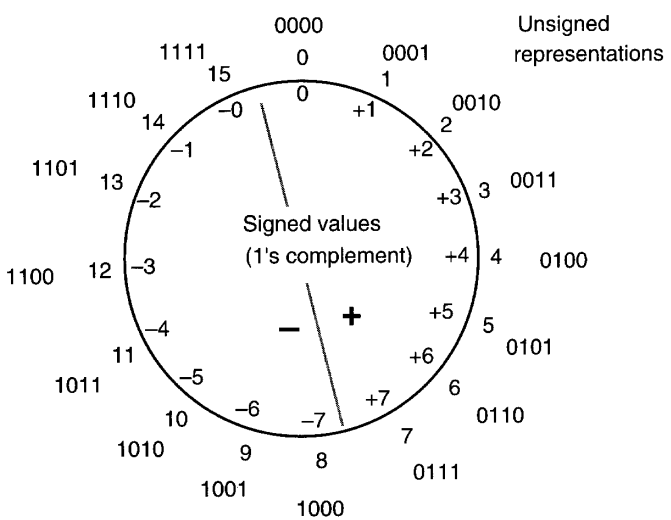


Fig. 2.6 A 4-bit, 1's-complement number representation system for integers.

The foregoing scheme properly handles any sum that equals or exceeds 2^k. When the sum is $2^k - ulp$, however, the carry-out will be zero and modular reduction is not accomplished. As suggested earlier, such an all-1s result can be interpreted as an alternate representation of 0 that is either kept intact (making 0 detection more difficult) or is automatically converted by hardware to +0.

The range of representable numbers in a 1's-complement number system with k whole bits is:

$$\text{from} \quad -(2^{k-1} - ulp) \quad \text{to} \quad 2^{k-1} - ulp$$

This symmetric range is one of the advantages of 1's-complement number representation.

Table 2.2 presents a brief comparison of radix- and digit-complement number representation systems for radix r. We might conclude from Table 2.2 that each of the two complement representation schemes has some advantages and disadvantages with respect to the other, making them equally desirable. However, since complementation is often performed for converting subtraction to addition, the addition of *ulp* required in the case of 2's-complement numbers can be accomplished by providing a carry-in of 1 into the least significant, or $(-l)$th, position of the adder. Figure 2.7 shows the required elements for a 2's-complement adder/subtractor. With the complementation disadvantage mitigated in this way, 2's-complement representation has become the favored choice in virtually all modern digital systems.

Interestingly, the arrangement shown in Fig. 2.7 also removes the disadvantage of asymmetric range. If the operand y is -2^{k-1}, represented in 2's complement as 1 followed by all 0s, its complementation does not lead to overflow. This is because the two's complement of y is essentially represented in two parts: y^{compl}, which represents $2^{k-1} - 1$, and c_{in} which represents 1.

Occasionally we need to extend the number of digits in an operand to make it of the same length as another operand. For example, if a 16-bit number is to be added to a 32-bit number, the former is first converted to 32-bit format, with the two 32-bit numbers then added using a 32-bit adder. Unsigned or signed-magnitude fixed-point binary numbers can be extended from the left (whole part) or the right (fractional part) by simply padding them with 0s. This type of range or precision extension is only slightly more difficult for 2's- and 1's-complement numbers.

Given a 2's-complement number $x_{k-1}x_{k-2}\cdots x_1x_0.x_1x_2\cdots x_{-l}$, extension can be achieved from the left by replicating the sign bit (*sign extension*) and from the right by padding it with 0s.

$$\cdots x_{k-1}x_{k-1}x_{k-1}x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l}000\cdots$$

TABLE 2.2
Comparing radix- and digit-complement number representation systems

Feature/Property	Radix complement	Digit complement
Symmetry ($P = N$?)	Possible for odd r (radices of practical interest are even)	Possible for even r
Unique zero?	Yes	No
Complementation	Complement all digits and add *ulp*	Complement all digits
Mod-M addition	Drop the carry-out	End-around carry

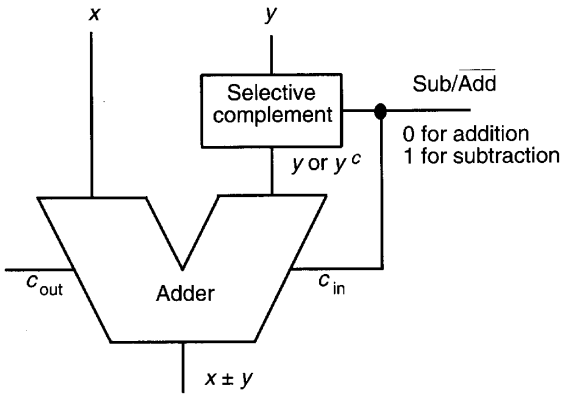


Fig. 2.7 Adder/subtractor architecture for 2's-complement numbers.

To justify the foregoing rule, note that when the number of whole (fractional) digits is increased from k (l) to k' (l'), the complementation constant increases from $M = 2^k$ to $M' = 2^{k'}$. Hence, the difference of the two complementation constants

$$M' - M = 2^{k'} - 2^k = 2^k(2^{k'-k} - 1)$$

must be added to the representation of any negative number. This difference is a binary integer consisting of $k' - k$ 1s followed by k 0s; hence the need for sign extension.

A 1's-complement number must be sign-extended from both ends:

$$\cdots x_{k-1}x_{k-1}x_{k-1}x_{k-1}x_{k-2}\cdots x_1x_0.x_1x_2\cdots x_{-l}x_{k-1}x_{k-1}x_{k-1}\cdots$$

Justifying the rule above for 1's-complement numbers is left as an exercise.

2.5 DIRECT AND INDIRECT SIGNED ARITHMETIC

In the preceding pages, we dealt with the addition and subtraction of signed numbers for a variety of number representation schemes (signed-magnitude, biased, complement). In all these cases, signed numbers were handled directly by the addition/subtraction hardware (*direct signed arithmetic*), consistent with our desire to avoid using separate addition and subtraction units.

For some arithmetic operations, it may be desirable to restrict the hardware to unsigned operands, thus necessitating *indirect signed arithmetic*. Basically, the operands are converted to unsigned values, a tentative result is obtained based on these unsigned values, and finally the necessary adjustments are made to find the result corresponding to the original signed operands. Figure 2.8 depicts the direct and indirect approaches to signed arithmetic.

Indirect signed arithmetic can be performed, for example, for multiplication or division of signed numbers, although we will see in Parts III and IV that direct algorithms are also available for this purpose. The process is trivial for signed-magnitude numbers. If x and y are biased

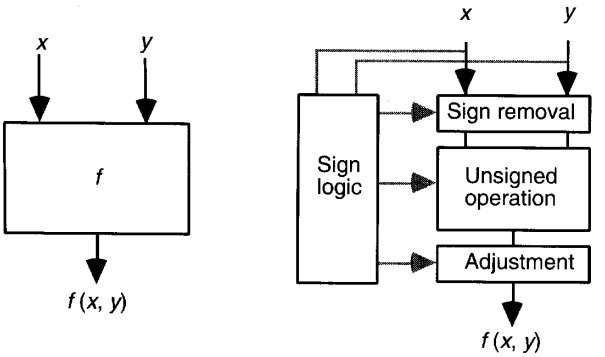


Fig. 2.8 Direct versus indirect operation on signed numbers.

numbers, then both the sign removal and adjustment steps involve addition/subtraction. If x and y are complement numbers, these steps involve selective complementation.

This type of preprocessing for operands, and postprocessing for computation results, is useful not only for dealing with signed values but also in the case of unacceptable or inconvenient operand values. For example, in computing $\sin x$, the operand can be brought to within $[0, \pi/2]$ by taking advantage of identities such as $\sin(-x) = -\sin x$ and $\sin(2\pi + x) = \sin(\pi - x) = \sin x$. Chapter 22 contains examples of such transformations. As a second example, some division algorithms become more efficient when the divisor is in a certain range (e.g., close to 1). In this case, the dividend and divisor can be scaled by the same factor in a preprocessing step to bring the divisor within the desired range (see Section 15.3).

2.6 USING SIGNED POSITIONS OR SIGNED DIGITS

The value of a 2's-complement number can be found by using the standard binary-to-decimal conversion process, except that the weight of the most significant bit (sign position) is taken to be negative. Figure 2.9 shows an example 8-bit, 2's-complement number converted to decimal by considering its sign bit to have the negative weight -2^7.

$$
\begin{array}{lcccccccccc}
x = (& 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0)_{\text{two's-compl}} \\
 & -2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & & \\
 & -128 & & + 32 & & & + 4 & + 2 & & = & -90
\end{array}
$$

Check:

$$
\begin{array}{lcccccccccc}
x = (& 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0)_{\text{two's-compl}} \\
-x = (& 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0)_{\text{two}} \\
 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & & \\
 & & 64 & + & 16 & + 8 & & + 2 & & = & 90
\end{array}
$$

Fig. 2.9 Interpreting a 2's-complement number as having a negatively weighted most significant digit.

This very important property of 2's-complement systems is used to advantage in many algorithms that deal directly with signed numbers. The property is formally expressed as follows:

$$\begin{aligned} x &= (x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{\text{two's-compl}} \\ &= -x_{k-1}2^{k-1} + \sum_{i=-l}^{k-2} x_i\, 2^i \end{aligned}$$

The proof is quite simple if we consider the two cases of $x_{k-1} = 0$ and $x_{k-1} = 1$ separately. For $x_{k-1} = 0$, we have:

$$\begin{aligned} x &= (0x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{\text{two's-compl}} \\ &= (0x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-1})_{\text{two}} \\ &= \sum_{i=-l}^{k-2} x_i\, 2^i \end{aligned}$$

For $x_{k-1} = 1$, we have:

$$\begin{aligned} x &= (1x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{\text{two's-compl}} \\ &= -[2^k - (1x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{\text{two}}] \\ &= -2^{k-1} + \sum_{i=-l}^{k-2} x_i\, 2^i \end{aligned}$$

Developing the corresponding interpretation for 1's-complement numbers is left as an exercise.

A simple generalization of the notion above immediately suggests itself [Kore81]. Let us assign negative weights to an arbitrary subset of the $k + l$ positions in a radix-r number and positive weights to the rest of the positions. A vector

$$\lambda = (\lambda_{k-1}\lambda_{k-2}\cdots\lambda_1\lambda_0.\lambda_{-1}\lambda_{-2}\cdots\lambda_{-l})$$

with elements λ_i in $\{-1, 1\}$, can be used to specify the signs associated with the various positions. With these conventions, the value represented by the digit vector x of length $k + l$ is:

$$(x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{r,\lambda} = \sum_{i=-l}^{k} \lambda_i x_i r^i$$

Note that the scheme above covers unsigned radix-r, 2's-complement, and negative-radix number systems as special cases:

$\lambda =$	1	1	1	$\cdots$	1	1	1	1	Positive radix
$\lambda =$	-1	1	1	$\cdots$	1	1	1	1	Two's complement
$\lambda =$				$\cdots$	-1	1	-1	1	Negative radix

We can take one more step in the direction of generality and postulate that instead of a single sign vector λ being associated with the digit positions in the number system (i.e., with all numbers represented), a separate sign vector is defined for each number. Thus, the digits are viewed as having signed values:

$$x_i = \lambda_i \, |x_i|, \quad \text{with } \lambda_i \in \{-1, 1\}$$

Here, λ_i is the sign and $|x_i|$ is the magnitude of the ith digit. In fact once we begin to view the digits as signed values, there is no reason to limit ourselves to signed-magnitude representation of the digit values. Any type of coding, including biased or complement representation, can be used for the digits. Furthermore, the range of digit values need not be symmetric. We have already covered some examples of such signed-digit number systems in Section 1.4 (see Examples 1.1, 1.3, and 1.4).

Basically, any set $[-\alpha, \beta]$ of r or more consecutive integers that includes 0 can be used as the digit set for radix r. If exactly r digit values are used, then the number system is irredundant and offers a unique representation for each value within its range. On the other hand, if more than r digit values are used, $\rho = \alpha + \beta + 1 - r$ represents the *redundancy index* of the number system and some values will have multiple representations. In Chapter 3, we will see that such redundant representations can eliminate the propagation of carries in addition and thus allow us to implement truly parallel fast adders.

As an example of nonredundant signed-digit representations, consider a radix-4 number system with the digit set $[-1, 2]$. A k-digit number of this type can represent any integer from $-(4^k - 1)/3$ to $2(4^k - 1)/3$. Given a standard radix-4 integer using the digit set $[0, 3]$, it can be converted to the preceding representation by simply rewriting each digit of 3 as $-1 + 4$, where the second term becomes a carry of 1 that propagates leftward. Figure 2.10 shows a numerical example. Note that the result may require $k + 1$ digits.

The conversion process of Fig. 2.10 stops when there remains no digit with value 3 that needs to be rewritten. The reverse conversion is similarly done by rewriting any digit of -1 as 3 with a borrow of 1 (carry of -1).

More generally, to convert between digit sets, each old digit value is rewritten as a valid new digit value and an appropriate transfer (carry or borrow) into the next higher digit position. Because these transfers can propagate, the conversion process is essentially a digit-serial one, beginning with the least significant digit.

As an example of redundant signed-digit representations, consider a radix-4 number system with the digit set $[-2, 2]$. A k-digit number of this type can represent any integer from

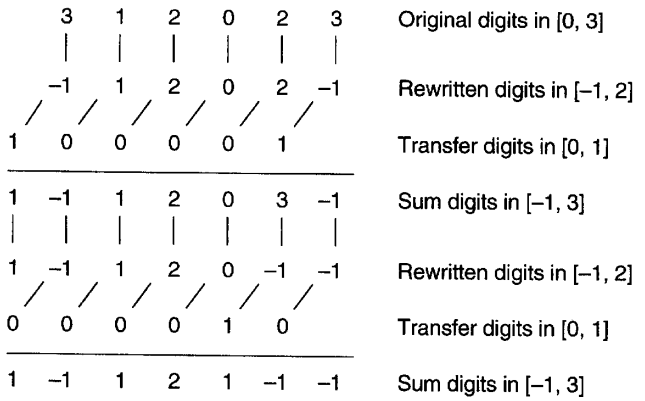


Fig. 2.10 Converting a standard radix-4 integer to a radix-4 integer with the nonstandard digit set $[-1, 2]$.

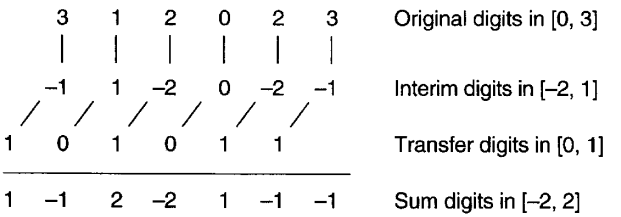


Fig. 2.11 Converting a standard radix-4 integer to a radix-4 integer with the nonstandard digit set $[-2, 2]$.

$-2(4^k - 1)/3$ to $2(4^k - 1)/3$. Given a standard radix-4 number using the digit set $[0, 3]$, it can be converted to the preceding representation by simply rewriting each digit of 3 as $-1 + 4$ and each digit of 2 as $-2 + 4$, where the second term in each case becomes a carry of 1 that propagates leftward. Figure 2.11 shows a numerical example.

In this case, the transfers do not propagate, since each transfer of 1 can be absorbed by the next higher position which has a digit value in $[-2, 1]$, forming a final result digit in $[-2, 2]$. The conversion process from conventional radix-4 to the preceding redundant representation is thus carry-free. The reverse process, however, remains digit-serial.

PROBLEMS

2.1 Signed-magnitude representation Design the control circuit of Fig. 2.2 so that signed-magnitude inputs are added correctly regardless of their signs. Include in your design a provision for overflow detection in the form of a fifth control circuit output.

2.2 Arithmetic on biased numbers Multiplication of biased numbers can be done in a direct or an indirect way.

a. Develop a direct multiplication algorithm for biased numbers. *Hint:* Use the identity $xy + bias = (x + bias)(y + bias) - bias[(x + bias) + (y + bias) - bias] + bias$.

b. Present an indirect multiplication algorithm for biased numbers.

c. Compare the algorithms of parts a and b with respect to delay and hardware implementation cost.

d. Repeat the comparison for part c in the special case of squaring a biased number.

2.3 Representation formats and conversions Consider the following five ways for representing integers in the range $[-127, 127]$ within an 8-bit format: (a) signed-magnitude, (b) 2's complement, (c) 1's complement, (d) excess-127 code (where an integer x is encoded using the binary representation of $x + 127$), (e) excess-128 code. Pick one of three more conventional and one of the two "excess" representations and describe conversion of numbers between the two formats in both directions.

2.4 Representation formats and conversions

a. Show conversion procedures from k-bit 2's-complement format to k-bit biased representation, with $bias = 2^{k-1}$, and vice versa. Pay attention to possible exceptions.

b. Repeat part a for $bias = 2^{k-1} - 1$.

c. Repeat part a for 1's-complement format.

d. Repeat part b for 1's-complement format.

2.5 Complement representation of negative numbers Consider a k-bit integer radix-2 complement number representation system with the complementation constant $M = 2^k$. The range of integers represented is taken to be from $-N$ to $+P$, with $N + P + 1 = M$. Determine all possible pairs of values for N and P (in terms of M) if the sign of the number is to be determined by:

a. Looking at the most significant bit only.

b. Inspecting the three most significant bits.

c. A single 4-input OR or AND gate.

d. A single 4-input NOR or NAND gate.

2.6 Complement representation of negative numbers Diminished radix complement was defined as being based on the complementation constant $r^k - ulp$. Study the implications of using an "augmented radix complement" system based on the complementation constant $r^k + ulp$.

2.7 One's- and 2's-complement number systems We discussed the procedures for extending the number of whole or fractional digits in a 1's- or 2's-complement number in Section 2.4. Discuss procedures for the reverse process of shrinking the number of digits (e.g., converting 32-bit numbers to 16 bits).

2.8 Interpreting 1's-complement numbers Prove that the value of the number $(x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{1\text{'s-compl}}$ can be calculated from the formula $-x_{k-1}(2^{k-1} - ulp) + \sum_{i=-l}^{k-2} x_i 2^i$.

2.9 One's- and 2's-complement number systems

a. Prove that $x - y = (x^c + y)^c$, where the superscript "c" denotes any complementation scheme.

b. Find the difference between the two binary numbers 0010 and 0101 in two ways: First by adding the 2's complement of 0101 to 0010, and then by using the equality of part a, where "c" denotes bitwise complementation. Compare the two methods with regard to their possible advantages and drawbacks.

2.10 Shifting of 1's- or 2's-complement numbers Left/right shifting is used to double/halve the magnitude of unsigned binary integers. How can we use shifting to accomplish the same for 1's- or 2's-complement numbers?

2.11 Arithmetic on 1's-complement numbers Discuss the effect of the end-around carry needed for 1's-complement addition on the worst-case carry propagation delay and the total addition time.

2.12 Range extension for complement numbers Prove that increasing the number of integer and fractional digits in one's-complement representation requires sign extension from both ends (i.e., positive numbers are extended with 0s and negative numbers with 1s at both ends).

2.13 Signed digits or digit positions

a. Present an algorithm for determining the sign of a number represented in a positional system with signed digit positions.

b. Repeat part a for signed-digit representations.

2.14 Signed digit positions Consider a positional radix-r integer number system with the associated position sign vector $\lambda = (\lambda_{k-1}\lambda_{k-2}\cdots\lambda_1\lambda_0)$, $\lambda_i \in \{-1, 1\}$. The additive inverse of a number x is the number $-x$.

a. Find the additive inverse of the k-digit integer Q all of whose digits are $r-1$.

b. Derive a procedure for finding the additive inverse of an arbitrary number x.

c. Specialize the algorithm of part b to the case of 2's-complement numbers.

2.15 Generalizing 2's complement: 2-adic numbers Around the turn of the twentieth century, K. Hensel defined the class of p-adic numbers for a given prime p. Consider the class of 2-adic numbers with infinitely many digits to the left and a finite number of digits to the right of the binary point. An infinitely repeated pattern of digits is represented by writing down a single pattern (the period) within parentheses. Here are some example 2-adic representations using this notation:

$$7 = (0)111. = \cdots 00000000111. \qquad 1/7 = (110)111. = \cdots 110110110111.$$
$$-7 = (1)001. = \cdots 11111111001. \qquad -1/7 = (001). \quad = \cdots 001001001001.$$
$$7/4 = (0)1.11 \qquad 1/10 = (1100)110.1$$

We see that 7 and -7 have their standard 2's-complement forms, with infinitely many digits. The representations of $1/7$ and $-1/7$, when multiplied by 7 and -7, respectively, using standard rules for multiplication, yield the representation of 1. Prove the following for 2-adic numbers:

a. Sign change of a 2-adic number is similar to 2's complementation.

b. The representation of a 2-adic number x is ultimately periodic if and only if x is rational.

c. The 2-adic representation of $-1/(2n+1)$ for $n \geq 0$ is (σ), for some bit string σ, where the standard binary representation of $1/(2n+1)$ is $(0.\sigma\sigma\sigma\cdots)_{\text{two}}$.

REFERENCES

[Aviz61] Avizienis, A., "Signed-Digit Number Representation for Fast Parallel Arithmetic," *IRE Trans. Electronic Computers*, Vol. 10, pp. 389-400, 1961.

[Gosl80] Gosling, J. B., *Design of Arithmetic Units for Digital Computers*, Macmillan, 1980.

[Knut97] Knuth, D. E., *The Art of Computer Programming*, 3rd ed., Vol. 2: *Seminumerical Algorithms*, Addison-Wesley, 1997.

[Kore81] Koren, I., and Y. Maliniak, "On Classes of Positive, Negative, and Imaginary Radix Number Systems," *IEEE Trans. Computers*, No. 5, Vol. 30, pp. 312-317, 1981.

[Korn94] Kornerup, P., "Digit-Set Conversions: Generalizations and Applications," *IEEE Trans. Computers*, Vol. 43, No. 8, pp. 622-629, 1994.

[Parh90] Parhami, B., "Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations," *IEEE Trans. Computers*, Vol. 39, No. 1, pp. 89-98, 1990.

[Parh98] Parhami, B., and S. Johansson, "A Number Representation Scheme with Carry-Free Rounding for Floating-Point Signal Processing Applications," *Proc. Int'l. Conf. Signal and Image Processing*, Las Vegas, Nevada, October 1998, pp. 90–92.

[Scot85] Scott, N. R., *Computer Number Systems and Arithmetic*, Prentice-Hall, 1985.

Chapter 3 REDUNDANT NUMBER SYSTEMS

This chapter deals with the representation of signed fixed-point numbers using a positive integer radix r and a redundant digit set composed of more than r digit values. After showing that such representations eliminate carry propagation, we cover variations in digit sets, addition algorithms, input/output conversions, and arithmetic support functions. Chapter topics include:

3.1 COPING WITH THE CARRY PROBLEM

Addition is a primary building block in implementing arithmetic operations. If addition is slow or expensive, all other operations suffer in speed or cost. Addition can be slow and/or expensive because:

a. With k-digit operands, one has to allow for $O(k)$ worst-case carry-propagation stages in simple ripple-carry adder design.
b. The carry computation network is a major source of complexity and cost in the design of carry-lookahead and other fast adders.

The carry problem can be dealt with in several ways:

1. Limit carry propagation to within a small number of bits.
2. Detect the end of propagation rather than wait for worst-case time.
3. Speed up propagation via lookahead and other methods.
4. Ideal: Eliminate carry propagation altogether!

As examples of option 1, hybrid redundant and residue number system representations are covered in Section 3.4 and Chapter 4, respectively. Asynchronous adder design (option 2) is considered in Section 5.4. Speedup methods for carry propagation are covered in Chapters 6 and 7.

In the remainder of this chapter, we deal with option 4, focusing first on the question: Can numbers be represented in such a way that addition does not involve carry propagation? We will see shortly that this is indeed possible. The resulting number representations can be used as the primary encoding scheme in the design of high-performance systems and are also useful in representing intermediate results in machines that use conventional number representation.

We begin with a decimal example ($r = 10$), assuming the standard digit set [0, 9]. Consider the addition of the following two decimal numbers without carry propagation. For this, we simply compute "position sums" and write them down in the corresponding columns. We can use the symbols $A = 10$, $B = 11$, $C = 12$, etc. for the extended digit values or simply represent them with two standard digits.

	5	7	8	2	4	9	
+	6	2	9	3	8	9	Operand digits in [0, 9]
	11	9	17	5	12	18	Position sums in [0, 18]

So, if we allow the digit set [0, 18], the scheme works, but only for the first addition! Subsequent additions will cause problems.

Consider now adding two numbers in the radix-10 number system using the digit set [0, 18]. The sum of digits for each position is in [0, 36], which can be decomposed into an interim sum in [0, 16] and a transfer digit in [0, 2]. In other words:

$$[0, 36] = 10 \times [0, 2] + [0, 16]$$

Adding the interim sum and the incoming transfer digit yields a digit in [0, 18] and creates no new transfer. In interval notation, we have:

$$[0, 16] + [0, 2] = [0, 18]$$

Figure 3.1 shows an example addition.

So, even though we cannot do true carry-free addition (Fig. 3.2a), the next best thing, where carry propagates by only one position (Fig. 3.2b), is possible if we use the digit set [0, 18] in radix 10. We refer to this best possible scheme as "carry-free" addition. The key to the ability to do carry-free addition is the representational redundancy that provides multiple encodings for some numbers. Figure 3.2c shows that the single-stage propagation of transfers can be eliminated by a simple lookahead scheme; that is, instead of first computing the transfer into position i based on the digits x_{i-1} and y_{i-1} and then combining it with the interim sum, we can determine s_i directly from x_i, y_i, x_{i-1}, and y_{i-1}. This may make the adder logic somewhat more complex, but in general the result is higher speed.

In the decimal example of Fig. 3.1, the digit set [0, 18] was used to effect carry-free addition. The 9 "digit" values 10 through 18 are redundant. However, we really do not need this much redundancy in a decimal number system for carry-free addition; the digit set [0, 11] will do. Our example addition (after converting the numbers to the new digit set) is shown in Fig. 3.3.

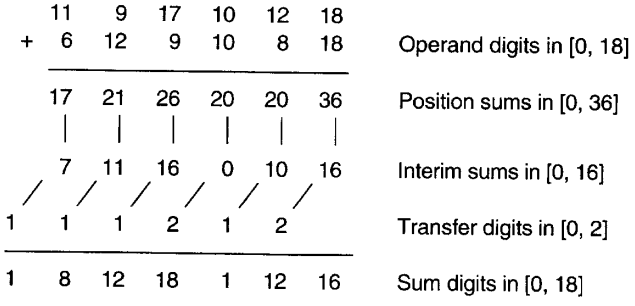


Fig. 3.1 Adding radix-10 numbers with the digit set [0, 18].

A natural question at this point is: How much redundancy in the digit set is needed to enable carry-free addition? For example, will the example addition of Fig. 3.3 work with the digit set [0, 10]? (Try it and see.) We will answer this question in Section 3.5.

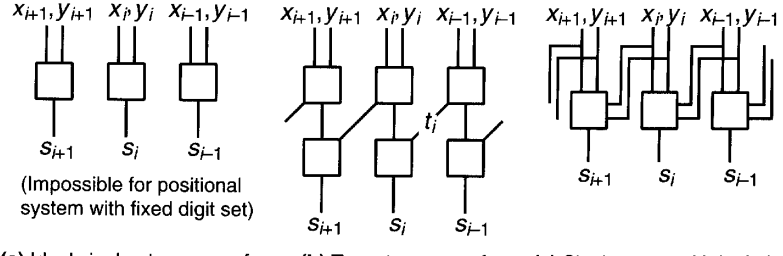
3.2 REDUNDANCY IN COMPUTER ARITHMETIC

Redundancy is used extensively for speeding up arithmetic operations. The oldest example, first suggested in 1959 [Metz59], pertains to carry-save or stored-carry numbers using the radix-2 digit set [0, 2] for fast addition of a sequence of binary operands. Figure 3.4 provides an example, showing how the intermediate sum is kept in stored-carry format, allowing each subsequent addition to be performed in a carry-free manner.

Why is this scheme called carry-save or stored-carry? Figure 3.5 provides an explanation. Let us use the 2-bit encoding

$$0 : (0, 0), \qquad 1 : (0, 1) \text{ or } (1, 0), \qquad 2 : (1, 1)$$

to represent the digit set [0, 2]. With this encoding, each stored-carry number is really composed of two binary numbers, one for each bit of the encoding. These two binary numbers can be added



(a) Ideal single-stage carry-free. **(b)** Two-stage carry-free. **(c)** Single-stage with lookahead.

Fig. 3.2 Ideal and practical carry-free addition schemes.

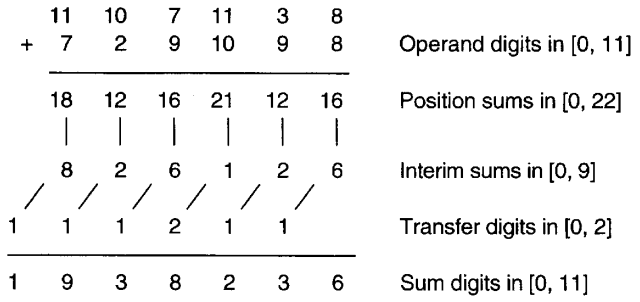


Fig. 3.3 Adding radix-10 numbers with the digit set [0, 11].

to an incoming binary number, producing two binary numbers composed of the sum bits kept in place and the carry bits shifted one position to the left. These sum and carry bits form the partial sum and can be stored in two registers for the next addition. Thus, the carries are "saved" or "stored" instead of being allowed to propagate.

Figure 3.5 shows that one stored-carry number and one standard binary number can be added to form a stored-carry sum in a single full-adder delay (2–4 gate levels, depending on the full adder's logic implementation of the outputs $s = x \oplus y \oplus c_{in}$ and $c_{out} = xy + xc_{in} + yc_{in}$). This

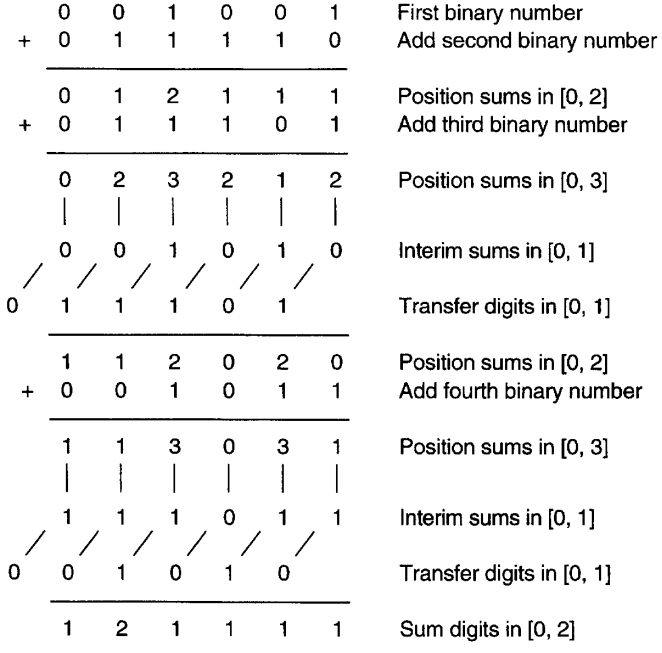


Fig. 3.4 Addition of four binary numbers, with the sum obtained in stored-carry form.

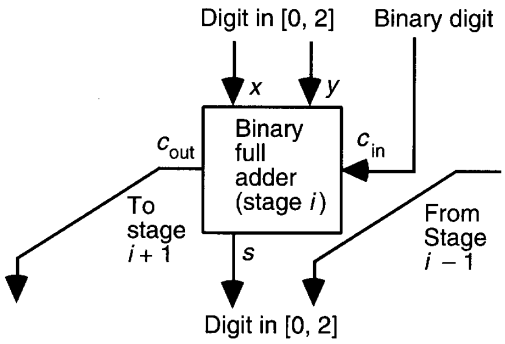


Fig. 3.5 Using an array of independent binary full adders to perform carry-save addition.

is significantly faster than standard carry-propagate addition to accumulate the sum of several binary numbers, even if a fast carry-lookahead adder is used for the latter. Of course once the final sum has been obtained in stored-carry form, it may have to be converted to standard binary by using a carry-propagate adder to add the two components of the stored-carry number. The key point is that the carry-propagation delay occurs only once, at the very end, rather than in each addition step.

Since the carry-save addition scheme of Fig. 3.5 converts three binary numbers to two binary numbers with the same sum, it is sometimes referred to as a 3/2 reduction circuit or (3; 2) counter. The latter name reflects the essential function of a full adder: it counts the number of 1s among its three input bits and outputs the result as a 2-bit binary number. More on this in Chapter 8.

Other examples of the use of redundant representations in computer arithmetic are found in fast multiplication and division schemes, where the multiplier or quotient is represented or produced in redundant form. More on these in Parts III and IV.

3.3 DIGIT SETS AND DIGIT-SET CONVERSIONS

Conventional radix-r numbers use the standard digit set $[0, r - 1]$. However, many other redundant and nonredundant digit sets are possible. A necessary condition is that the digit set contain at least r different digit values. If it contains more than r values, the number system is redundant.

Conversion of numbers between standard and other digit sets is quite simple and essentially entails a digit-serial process in which, beginning at the right end of the given number, each digit is rewritten as a valid digit in the new digit set and a transfer (carry or borrow) into the next higher digit position. This conversion process is essentially like carry propagation in that it must be done from right to left and, in the worst case, the most significant digit is affected by a "carry" coming from the least significant position. The following examples illustrate the process (see also the examples at the end of Section 2.6).

Example 3.1 Convert the following radix-10 number with the digit set [0, 18] to one using the conventional digit set [0, 9].

	11	9	17	10	12	18	Rewrite 18 as 10 (carry 1) + 8
	11	9	17	10	13	8	13 = 10 (carry 1) + 3
	11	9	17	11	3	8	11 = 10 (carry 1) + 1
	11	9	18	1	3	8	18 = 10 (carry 1) + 8
	11	10	8	1	3	8	10 = 10 (carry 1) + 0
	12	0	8	1	3	8	12 = 10 (carry 1) + 2
1	2	0	8	1	3	8	Answer: all digits in [0, 9]

Example 3.2 Convert the following radix-2 carry-save number to binary; that is, from digit set [0, 2] to digit set [0, 1].

	1	1	2	0	2	0	Rewrite 2 as 2 (carry 1) + 0
	1	1	2	1	0	0	2 = 2 (carry 1) + 0
	1	2	0	1	0	0	2 = 2 (carry 1) + 0
	2	0	0	1	0	0	2 = 2 (carry 1) + 0
1	0	0	0	1	0	0	Answer: all digits in [0, 1]

Another way to accomplish the preceding conversion is to decompose the carry-save number into two numbers, both of which have 1s where the original number has a digit of 2. The sum of these two numbers is then the desired binary number.

	1	1	1	0	1	0	First number: "sum" bits
+	0	0	1	0	1	0	Second number: "carry" bits
1	0	0	0	1	0	0	Sum of the two numbers

Example 3.3 Digit values do not have to be positive. We reconsider Example 3.1 using the asymmetric target digit set [−6, 5].

	11	9	17	10	12	18	Rewrite 18 as 20 (carry 2) − 2
	11	9	17	10	14	−2	14 = 10 (carry 1) + 4
	11	9	17	11	4	−2	11 = 10 (carry 1) + 1
	11	9	18	1	4	−2	18 = 20 (carry 1) − 2
	11	11	−2	1	4	−2	11 = 10 (carry 1) + 1
	12	1	−2	1	4	−2	12 = 10 (carry 1) + 2
1	2	1	−2	1	4	−2	Answer: all digits in [−6, 5]

On line 2 of this conversion, we could have rewritten 14 as 20 (carry 2) − 6, which would have led to a different, but equivalent, representation. In general, several representations may be possible with a redundant digit set.

Example 3.4 If we change the target digit set of Example 3.2 from [0, 1] to [−1, 1], we can do the conversion digit-serially as before. However, carry-free conversion is possible for this example if we rewrite each 2 as 2 (carry 1) + 0 and each 1 as 2 (carry 1) −1. The resulting interim digits in [−1, 0] can absorb an incoming carry of 1 with no further propagation.

	1	1	2	0	2	0	Given carry-save number
	−1	−1	0	0	0	0	Interim digits in [−1, 0]
1	1	1	0	1	0		Transfer digits in [0, 1]
1	0	0	0	1	0	0	Answer: all digits in [−1, 1]

3.4 GENERALIZED SIGNED-DIGIT NUMBERS

We have seen thus far that digit set of a radix-r positional number system need not be the standard set $[0, r-1]$. Using the digit set $[-1, 1]$ for radix-2 numbers was proposed by E. Collignon as early as 1897 [Glas81]. Whether this was just a mathematical curiosity, or motivated by an application or advantage, is not known. In the early 1960s, Avizienis [Aviz61] defined the class of signed-digit number systems with symmetric digit sets $[-\alpha, \alpha]$ and radix $r > 2$, where α is any integer in the range $\lfloor r/2 \rfloor + 1 \leq \alpha \leq r-1$. These number systems allow at least $2\lfloor r/2 \rfloor + 3$ digit values, instead of the minimum required r values, and are thus redundant.

More recently, redundant number systems with general, possibly asymmetric, digit sets of the form $[-\alpha, \beta]$ have been studied as tools for unifying all redundant number representations used in practice. This class is called "generalized signed-digit (GSD) representation" and differs from the ordinary signed-digit (OSD) representation of Avizienis in its more general digit set as well as the possibility of higher or lower redundancy.

Binary stored-carry numbers, with $r = 2$ and digit set [0, 2], offer a good example for the usefulness of asymmetric digit sets. Higher redundancy is exemplified by the digit set [−7, 7] in radix 4 or [0, 3] in radix 2. An example for lower redundancy is the binary signed-digit representation with $r = 2$ and digit set [−1, 1]. None of these is covered by OSD.

An important parameter of a GSD number system is its *redundancy index*, defined as $\rho = \alpha + \beta + 1 - r$ (i.e., the amount by which the size of its digit set exceeds the size r of a nonredundant digit set for radix r). Figure 3.6 presents a taxonomy of redundant and nonredundant positional number systems showing the names of some useful subclasses and their various relationships.

Any hardware implementation of GSD arithmetic requires the choice of a binary encoding scheme for the $\alpha + \beta + 1$ digit values in the digit set $[-\alpha, \beta]$. Multivalued logic realizations have been considered, but we limit our discussion here to binary logic and proceed to show the importance and implications of the encoding scheme chosen through some examples.

Consider, for example, the binary signed-digit (BSD) number system with $r = 2$ and the digit set [−1, 1]. One needs at least 2 bits to encode these three digit values. Figure 3.7 shows four of the many possible encodings that can be used.

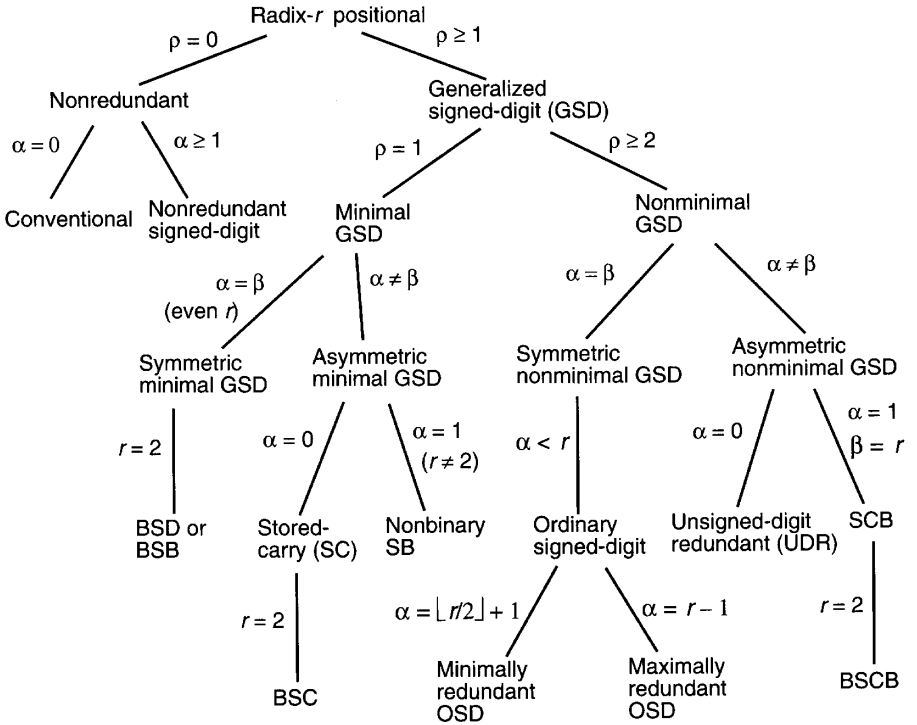


Fig. 3.6 A taxonomy of redundant and nonredundant positional number systems.

With the (n, p) encoding, the code (1, 1) may be considered an alternate representation of 0 or else viewed as an invalid combination. Many implementations have shown that the (n, p) encoding tends to simplify the hardware and also increases the speed by reducing the number of gate levels [Parh88]. The 1-out-of-3 encoding requires more bits per number but allows the detection of some storage and processing errors.

Hybrid signed-digit representations [Phat94] came about from an attempt to strike a balance between algorithmic speed and implementation cost by introducing redundancy in selected positions only. For example, standard binary representation may be used with BSD digits allowed in every third position, as shown in the addition example of Fig. 3.8.

x_i	1	−1	0	−1	0	Representation of +6
(s, v)	01	11	00	11	00	Sign and value encoding
2's-compl	01	11	00	11	00	2-bit 2's-complement
(n, p)	01	10	00	10	00	Negative and positive flags
(n, z, p)	001	100	010	100	010	1-out-of-3 encoding

Fig. 3.7 Four encodings for the BSD digit set [−1, 1].

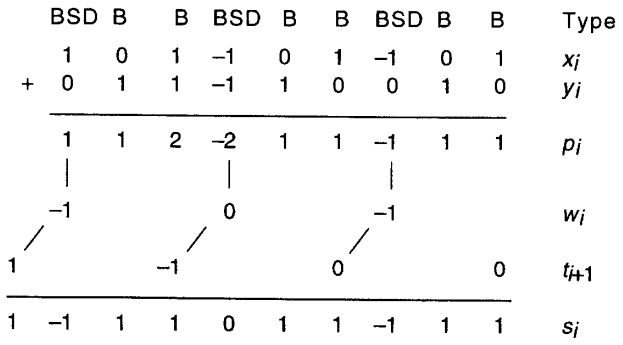


Fig. 3.8 Example of addition for hybrid signed-digit numbers.

The addition algorithm depicted in Fig. 3.8 proceeds as follows. First one completes the position sums p_i that are in [0, 2] for standard binary and [−2, 2] in BSD positions. The BSD position sums are then broken into an interim sum w_i and transfer t_{i+1}, both in [−1, 1]. For the interim sum digit, the value 1 (−1) is chosen only if it is certain that the incoming transfer cannot be 1 (−1); that is, when the two binary operand digits in position $i - 1$ are (not) both 0s. The worst-case carry propagation spans a single group, beginning with a BSD digit that produces a transfer digit in [−1, 1] and ending with the next higher BSD position.

More generally, the group size can be g rather than 3. A larger group size reduces the hardware complexity (since the adder block in a BSD position is more complex than that in other positions) but adds to the carry-propagation delay in the worst case; hence, the hybrid scheme offers a trade-off between speed and cost.

Hybrid signed-digit representation with uniform spacing of BSD positions can be viewed as a special case of GSD systems. For the example of Fig. 3.8, arranging the numbers in 3-digit groups starting from the right end leads to a radix-8 GSD system with digit set [−4, 7]: that is, digit values from $(\bar{1}\ 0\ 0)_{\text{two}}$ to $(1\ 1\ 1)_{\text{two}}$. So the hybrid scheme of Fig. 3.8 can be viewed as an implementation of (digit encoding for) this particular radix-8 GSD representation.

3.5 CARRY-FREE ADDITION ALGORITHMS

The GSD carry-free addition algorithm, corresponding to the scheme of Fig. 3.2b, is as follows:

Carry-free addition algorithm for GSD numbers

Compute the position sums $p_i = x_i + y_i$.

Divide each p_i into a transfer t_{i+1} and an interim sum $w_i = p_i - rt_{i+1}$.

Add the incoming transfers to obtain the sum digits $s_i = w_i + t_i$.

Let us assume that the transfer digits t_i are from the digit set $[-\lambda, \mu]$. To ensure that the last step leads to no new transfer, the following condition must be satisfied:

$$\begin{array}{ccccc} -\alpha+\lambda & \leq & p_i - rt_{i+1} & \leq & \beta-\mu \\ | & & \text{interim sum} & & | \end{array}$$

Smallest interim sum	Largest interim sum
if a transfer of $-\lambda$	if a transfer of μ
is to be absorbable	is to be absorbable

From the preceding inequalities, we can easily derive the conditions $\lambda \geq \alpha/(r-1)$ and $\mu \geq \beta/(r-1)$. Once λ and μ are known, we choose the transfer digit value by comparing the position sum p_i against $\lambda+\mu+2$ constants C_j, $-\lambda \leq j \leq \mu+1$, with the transfer digit taken to be j if and only if $C_j \leq p_i < C_{j+1}$. Figure 3.9 represents the decision process graphically. Formulas giving possible values for these constants can be found in [Parh90]. Here, we describe a simple intuitive method for deriving these constants.

Example 3.5 For $r = 10$ and digit set $[-5, 9]$, we need $\lambda \geq 5/9$ and $\mu \geq 1$. Given minimal values for λ and μ that minimize the hardware complexity, we find by choosing the minimal values for λ and μ, we find:

$$\lambda_{\min} = \mu_{\min} = 1 \qquad \text{(i.e., transfer digits are in } [-1, 1])$$

$$-\infty = C_{-1} \qquad -4 \leq C_0 \leq -1 \qquad 6 \leq C_1 \leq 9 \qquad C_2 = +\infty$$

We next show how the allowable values for the comparison constant C_1, shown above, are derived. The position sum p_i is in $[-10, 18]$. We can set t_{i+1} to 1 for p_i values as low as 6; for $p_i = 6$, the resulting interim sum of -4 can absorb any incoming transfer in $[-1, 1]$ without falling outside $[-5, 9]$. On the other hand, we must transfer 1 for p_i values of 9 or more. Thus, for $p_i \geq C_1$, where $6 \leq C_1 \leq 9$, we choose an outgoing transfer of 1. Similarly, for $p_i < C_0$, we choose an outgoing transfer of -1, where $-4 \leq C_0 \leq -1$. In all other cases, the outgoing transfer is 0.

Assuming that the position sum p_i is represented as a 6-bit, 2's-complement number *abcdef*, good choices for the comparison constants in the above ranges are $C_0 = -4$ and $C_1 = 8$. The logic expressions for the signals g_1 and g_{-1} then become:

$$g_{-1} = a(\bar{c} + \bar{d}) \qquad \text{Generate a transfer of } -1$$

$$g_1 = \bar{a}(b + c) \qquad \text{Generate a transfer of } 1$$

An example addition is shown in Fig. 3.10.

It is proven in [Parh90] that the preceding carry-free addition algorithm is applicable to a redundant representation if and only if one of the following sets of conditions is satisfied:

a. $r > 2, \rho \geq 3$
b. $r > 2, \rho = 2, \alpha \neq 1, \beta \neq 1$

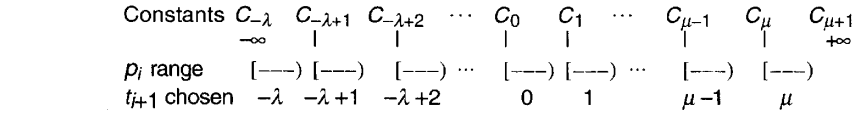


Fig. 3.9 Choosing the transfer digit t_{i+1} based on comparing the interim sum p_i to the comparison constants C_j.

In other words, the carry-free algorithm is not applicable for $r = 2$, $\rho = 1$, or $\rho = 2$ with $\alpha = 1$ or $\beta = 1$. In such cases, a limited-carry addition algorithm is available:

Limited-carry addition algorithm for GSD numbers

Compute the position sums $p_i = x_i + y_i$.

Compare each p_i to a constant to determine whether e_{i+1} = "low" or "high" (e_{i+1} is a binary range estimate for t_{i+1}).

Given e_i, divide each p_i into a transfer t_{i+1} and an interim sum $w_i = p_i - rt_{i+1}$.

Add the incoming transfers to obtain the sum digits $s_i = w_i + t_i$.

This "limited-carry" GSD addition algorithm is depicted in Fig. 3.11a; in an alternative implementation (Fig. 3.11b), the "transfer estimate" stage is replaced by another transfer generation/addition phase.

Even though Figs. 3.11a and 3.11b appear similar, they are quite different in terms of the internal designs of the square boxes in the top and middle rows. In both cases, however, the sum digit s_i depends on x_i, y_i, x_{i-1}, y_{i-1}, x_{i-2}, and y_{i-2}. Rather than wait for the limited transfer propagation from stage $i-2$ to i, one can try to provide the necessary information directly from stage $i-2$ to stage i. This leads to an implementation with parallel carries $t_{i+1}^{(1)}$ and $t_{i+2}^{(2)}$ from stage i, which is sometimes applicable (Fig. 3.11c).

■ **Example 3.6** Figure 3.12 depicts the use of carry estimates in limited-carry addition of radix-2 numbers with the digit set $[-1, 1]$. Here we have $\rho = 1$, $\lambda_{\min} = 1$, and $\mu_{\min} = 1$. The "low" and "high" subranges for transfer digits are $[-1, 0]$ and $[0, 1]$, respectively, with a transfer t_{i+1} in "high" indicated if $p_i \geq 0$.

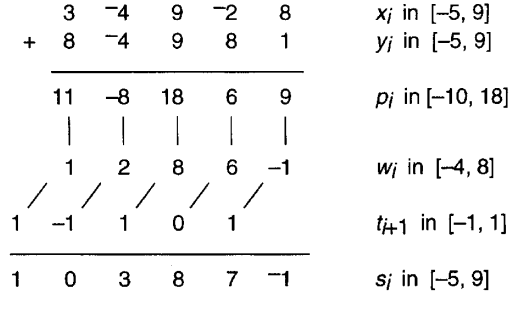
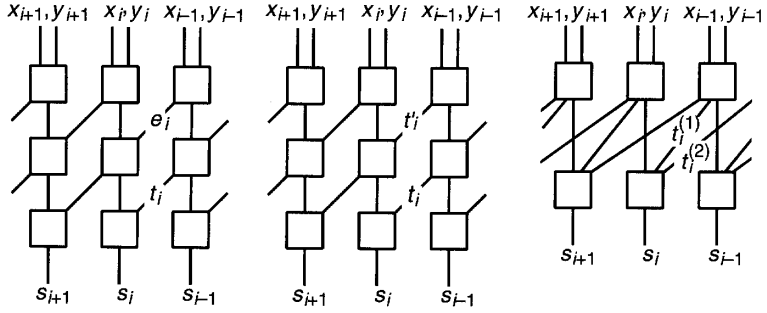


Fig. 3.10 Adding radix-10 numbers with the digit set $[-5, 9]$.



(a) Three-stage carry estimate. **(b)** Three-stage repeated carry. **(c)** Two-stage parallel carries.

Fig. 3.11 Some implementations for limited-carry addition.

■ **Example 3.7** Figure 3.13 shows another example of limited-carry addition with $r = 2$, digit set [0, 3], $\rho = 2$, $\lambda_{\min} = 0$, and $\mu_{\min} = 3$, using carry estimates. The "low" and "high" subranges for transfer digits are [0, 2] and [1, 3], respectively, with a transfer t_{i+1} in "high" indicated if $p_i \geq 4$.

■ **Example 3.8** Figure 3.14 shows the same addition as in Example 3.7 ($r = 2$, digit set [0, 3], $\rho = 2$, $\lambda_{\min} = 0$, $\mu_{\min} = 3$) using the repeated-carry scheme of Fig. 3.11b.

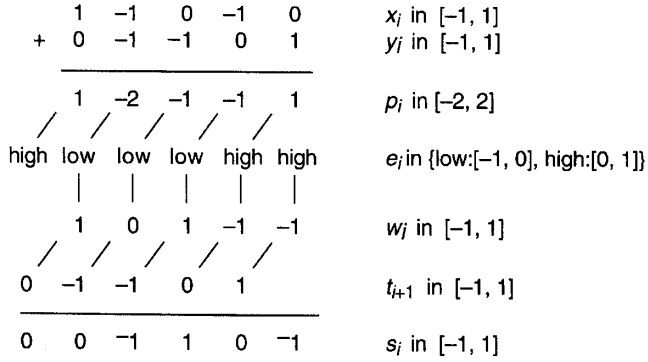


Fig. 3.12 Limited-carry addition of radix-2 numbers with the digit set [−1, 1] by means of carry estimates. A position sum of −1 is kept intact when the incoming transfer is in [0, 1], whereas it is rewritten as 1 with a carry of −1 if the incoming transfer is in [−1, 0]. This scheme guarantees that $t_i \neq w_i$ and thus $-1 \leq s_i \leq 1$.

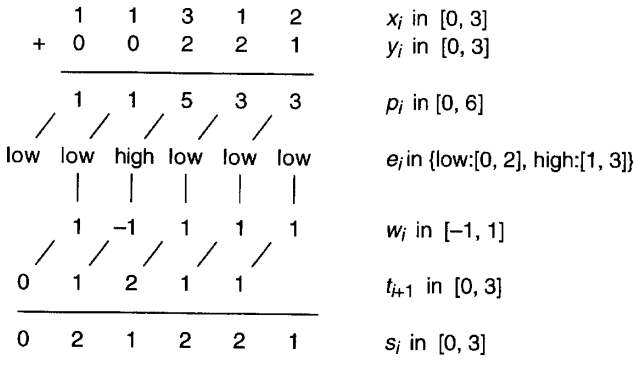


Fig. 3.13 Limited-carry addition of radix-2 numbers with the digit set [0, 3] by means of carry estimates. A position sum of 1 is kept intact when the incoming transfer is in [0, 2], whereas it is rewritten as -1 with a carry of 1 if the incoming transfer is in [1, 3].

■ **Example 3.9** Figure 3.15 shows the same addition as in Example 3.7 ($r = 2$, digit set [0, 3], $\rho = 2$, $\lambda_{\min} = 0$, $\mu_{\min} = 3$) using the parallel-carries scheme of Fig. 3.11c.

Subtraction of GSD numbers is very similar to addition. With a symmetric digit set, one can simply invert the signs of all digits in the subtractor y to obtain a representation of $-y$ and then perform the addition $x + (-y)$ using a carry-free or limited-carry algorithm as already discussed. Negation of a GSD number with an asymmetric digit set is somewhat more complicated, but can still be performed by means of a carry-free algorithm [Parh93]. This algorithm basically

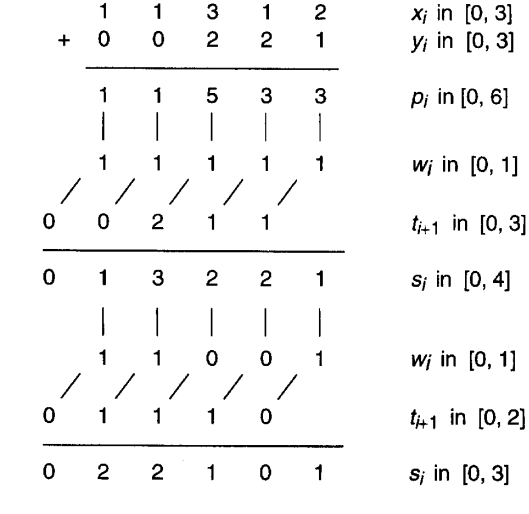


Fig. 3.14 Limited-carry addition of radix-2 numbers with the digit set [0, 3] by means of the repeated-carry scheme.

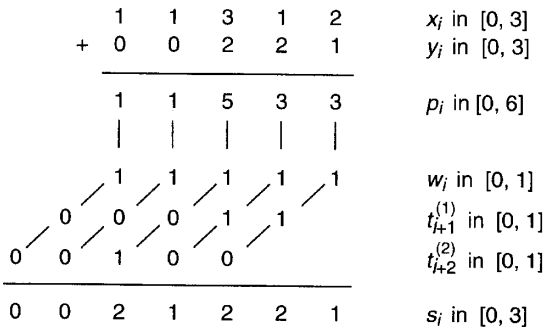


Fig. 3.15 Limited-carry addition of radix-2 numbers with the digit set [0, 3] by means of the parallel-carries scheme.

converts a radix-r number from the digit set $[-\beta, \alpha]$, which results from changing the signs of the individual digits of y, to the original digit set $[-\alpha, \beta]$. Alternatively, a direct subtraction algorithm can be applied by first computing position differences in $[-\alpha-\beta, \alpha+\beta]$, then forming interim differences and transfer digits. Details are omitted here.

3.6 CONVERSIONS AND SUPPORT FUNCTIONS

Since input numbers provided from the outside (machine or human interface) are in standard binary or decimal and outputs must be presented in the same way, conversions between binary or decimal and GSD representations are required.

■ **Example 3.10** Consider number conversions from or to standard binary to or from binary signed-digit representation. To convert from signed binary to BSD, we simply attach the common number sign to each digit, if the (s, v) code of Fig. 3.7 is to be used for the BSD digits. Otherwise, we need a simple digitwise converter from the (s, v) code to the desired code. To convert from BSD to signed binary, we separate the positive and negative digits into a positive and a negative binary number, respectively. A subtraction then yields the desired result. Here is an example:

1	−1	0	−1	0	BSD representation of +6
1	0	0	0	0	Positive part (1 digits)
0	1	0	1	0	Negative part (−1 digits)
0	0	1	1	0	Difference = conversion result

The positive and negative parts required above are particularly easy to obtain if the BSD number is represented using the (n, p) code of Fig. 3.7. The reader should be able to modify the process above for dealing with numbers, or deriving results, in 2's-complement format.

The conversion from redundant to nonredundant representation essentially involves carry propagation and is thus rather slow. Hopefully, however, we will not need conversions very often. Conversion is done at the input and output. Thus, if long sequences of computation are performed between input and output, the conversion overhead can become negligible.

Storage overhead (the larger number of bits that may be needed to represent a GSD digit compared to a standard digit in the same radix) used to be a major disadvantage of redundant representations. However, with advances in VLSI technology, this is no longer a major issue; though the increase in the number of pins for input and output may still be a factor.

In the rest of this section, we review some properties of GSD representations that are important for the implementation of arithmetic support functions: zero detection, sign test, and overflow handling [Parh93].

In a GSD number system, the integer 0 may have multiple representations. For example, the three-digit numbers 0 0 0 and $^-1$ 4 0 both represent 0 in radix 4. However, in the special case of $\alpha < r$ and $\beta < r$, zero is uniquely represented by the all-0s vector. So despite redundancy and multiple representations, comparison of numbers for equality can be simple in this common special case, since it involves subtraction and detecting the all-0s pattern.

Sign test, and thus any relational comparison ($<$, $\leq$, etc.), is more difficult. The sign of a GSD number in general depends on all its digits. Thus sign test is slow if done through signal propagation (ripple design) or expensive if done by a fast lookahead circuit (contrast this with the trivial sign test for signed-magnitude and 2's-complement representations). In the special case of $\alpha < r$ and $\beta < r$, the sign of number is identical to the sign of its most significant nonzero digit. Even in this special case, determination of sign requires scanning of all digits in the worst case, a process that can be as slow as full carry propagation.

Overflow handling is also more difficult in GSD arithmetic. Consider the addition of two k-digit numbers, as shown in Fig. 3.16. Such an addition produces a transfer-out digit t_k. Since t_k is produced using the worst-case assumption about the as yet unknown t_{k-1}, we can get an overflow indication ($t_k \neq 0$) even when the result can be represented with k digits. It is possible to perform a test to see whether the overflow is real and, if it is not, to obtain a k-digit representation for the true result. However, this test and conversion are fairly slow.

The difficulties with sign test and overflow detection can nullify some or all of the speed advantages of GSD number representations. This is why applications of GSD are presently limited to special-purpose systems or to internal number representations, which are subsequently converted to standard representation.

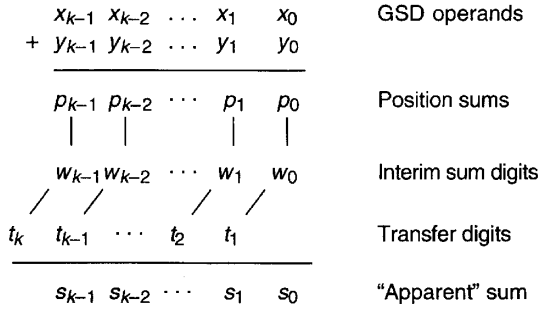


Fig. 3.16 Overflow and its detection in GSD arithmetic.

PROBLEMS

3.1 Stored-carry and stored-borrow representations The radix-2 number systems using the digit sets [0, 2] and [−1, 1] are known as binary stored-carry and stored-borrow representations, respectively. The general radix-r stored-carry and stored-borrow representations are based on the digit sets $[0, r]$ and $[-1, r-1]$, respectively.

- **a.** Show that carry-free addition is impossible for stored-carry/borrow numbers.
- **b.** Supply the details of limited-carry addition for radix-r stored-carry numbers.
- **c.** Supply the details of limited-carry addition for radix-r stored-borrow numbers.
- **d.** Compare the algorithms of parts b and c and discuss.

3.2 Stored-double-carry and stored-triple-carry representations The radix-4 number system using the digit set [0, 4] is a stored-carry representation. Use the digit sets [0, 5] and [0, 6] to form the radix-4 stored-double-carry and stored-triple-carry number systems, respectively.

- **a.** Find the relevant parameters for carry-free addition in the two systems (i.e., the range of transfer digits and the comparison constants). Where there is a choice, select the best value and justify your choice.
- **b.** State the advantages (if any) of one system over the other.

3.3 Stored-carry-or-borrow representations The general radix-r stored-carry-or-borrow representations use the digit set $[-1, r]$.

- **a.** Show that carry-free addition is impossible for stored-carry-or-borrow numbers.
- **b.** Develop a limited-carry addition algorithm for such radix-r numbers.
- **c.** Compare the stored-carry-or-borrow representation to the stored-double-carry representation based on the digit set $[0, r+1]$ and discuss.

3.4 Addition with parallel carries

- **a.** The redundant radix-2 representation with the digit set [0, 3], used in several examples in Section 3.5, is known as the binary stored-double-carry number system [Parh96]. Design a digit slice of a binary stored-double-carry adder based on the addition scheme of Fig. 3.15.
- **b.** Repeat part a with the addition scheme of Fig. 3.13.
- **c.** Repeat part a with the addition scheme of Fig. 3.14.
- **d.** Compare the implementations of parts a–c with respect to speed and cost.

3.5 Addition with parallel or repeated carries

- **a.** Develop addition algorithms similar to those discussed in Section 3.5 for binary stored-triple-carry number system using the digit set [0, 4].
- **b.** Repeat part a for the binary stored-carry-or-borrow number system based on the digit set [−1, 2].
- **c.** Develop a sign detection scheme for binary stored-carry-or-borrow numbers.
- **d.** Can one use digit sets other than [0, 3], [0, 4], and [−1, 2] in radix-2 addition with parallel carries?
- **e.** Repeat parts a–d for addition with repeated carries.

3.6 Nonredundant and redundant digit sets Consider a fixed-point, symmetric radix-3 number system, with k whole and l fractional digits, using the digit set $[-1, 1]$.

a. Determine the range of numbers represented as a function of k and l.

b. What is the representation efficiency relative to binary representation, given that each radix-3 digit needs a 2-bit code?

c. Devise a carry-free procedure for converting a symmetric radix-3 positive number to an unsigned radix-3 number with the redundant digit set $[0, 3]$.

d. What is the representation efficiency of the redundant number system of part c?

3.7 Digit-set and radix conversions Consider a fixed-point, radix-4 number system, with k whole and l fractional digits, using the digit set $[-3, 3]$.

a. Determine the range of numbers represented as a function of k and l.

b. Devise a procedure for converting such a radix-4 number to a radix-8 number that uses the digit set $[-7, 7]$.

c. Specify the numbers K and L of integer and fractional digits in the new radix of part b as functions of k and l.

d. Devise a procedure for converting such a radix-4 number to a radix-4 number that uses the digit set $[-2, 2]$.

3.8 Hybrid signed-digit representation Consider a hybrid radix-2 number representation system with the repeating pattern of two standard binary positions followed by one BSD position. The addition algorithm for this system is similar to that in Fig. 3.8. Show that this algorithm can be formulated as carry-free radix-8 GSD addition and derive its relevant parameters (range of transfer digits and comparison constants for transfer digit selection).

3.9 GSD representation of zero

a. Obtain necessary and sufficient conditions for zero to have a unique representation in a GSD number system.

b. Devise a 0 detection algorithm for cases in which 0 has multiple representations.

c. Design a hardware circuit for detecting 0 in an 8-digit radix-4 GSD representation using the digit set $[-2, 4]$.

3.10 Imaginary-radix GSD representation Show that the imaginary-radix number system with $r = 2j$, where $j = \sqrt{-1}$, and digit set $[-2, 2]$ lends itself to a limited-carry addition process. Define the process and derive its relevant parameters.

3.11 Negative-radix GSD representation Do you see any advantage to extending the definition of GSD representations to include the possibility of a negative radix r? Explain.

3.12 Mixed redundant–conventional arithmetic We have seen that BSD numbers cannot be added in a carry-free manner but that a limited-carry process can be applied to them.

a. Show that one can add a conventional binary number to a BSD number to obtain their BSD sum in a carry-free manner.

b. Supply the complete logic design for the carry-free adder of part a.

c. Compare your design to a carry-save adder and discuss.

3.13 Negation of GSD numbers One disadvantage of GSD representations with asymmetric digit sets is that negation (change of sign) becomes nontrivial. Show that negation of GSD numbers is always a carry-free process and derive a suitable algorithm for this purpose.

3.14 Digit-serial GSD arithmetic GSD representations allow fast carry-free or limited-carry parallel addition. GSD representations may seem less desirable for digit-serial addition because the simpler binary representation already allows very efficient bit-serial addition. Consider a radix-4 GSD representation using the digit set $[-3, 3]$.

a. Show that two such GSD numbers can be added digit-serially beginning at the most significant end (MSD-first arithmetic).

b. Present a complete logic design for your digit-serial adder and determine its latency.

c. Do you see any advantage for MSD-first, as opposed to LSD-first, arithmetic?

3.15 BSD arithmetic Consider binary signed-digit numbers with digit set $[-1, 1]$ and the 2-bit (n, p) encoding of the digits (see Fig. 3.7). The code (1, 1) never appears and can be used as don't-care.

a. Design a fast sign detector for a 4-digit BSD input operand using full lookahead.

b. How can the design of part a be used for 16-digit inputs?

c. Design a single-digit BSD full adder producing the sum digit s_i and transfer t_{i+1}.

3.16 Unsigned-digit redundant representations Consider the hex-digit decimal (HDD) number system with $r = 10$ and digit set $[0, 15]$ for representing unsigned integers.

a. Find the relevant parameters for carry-free addition in this system.

b. Design an HDD adder using 4-bit binary adders and a simple postcorrection circuit.

3.17 Double-LSB 2's-complement numbers Consider k-bit 2's-complement numbers with an extra least significant bit attached to them [Parh98]. Show that such redundant numbers have symmetric range, allow for bitwise 2's-complementation, and can be added using a standard k-bit adder.

REFERENCES

[Aviz61] Avizienis, A., "Signed-Digit Number Representation for Fast Parallel Arithmetic," *IRE Trans. Electronic Computers*, Vol. 10, pp. 389–400, 1961.

[Glas81] Glaser, A., *History of Binary and Other Nondecimal Numeration*, rev. ed., Tomash Publishers, 1981.

[Korn94] Kornerup, P., "Digit-Set Conversions: Generalizations and Applications," *IEEE Trans. Computers*, Vol. 43, No. 8, pp. 622–629, 1994.

[Metz59] Metze, G., and J.E. Robertson, "Elimination of Carry Propagation in Digital Computers," *Information Processing '59* (Proceedings of a UNESCO Conference), 1960, pp. 389–396.

[Parh88] Parhami, B., "Carry-Free Addition of Recoded Binary Signed-Digit Numbers," *IEEE Trans. Computers*, Vol. 37, No. 11, pp. 1470–1476, 1988.

[Parh90] Parhami, B., "Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations," *IEEE Trans. Computers*, Vol. 39, No. 1, pp. 89–98, 1990.

[Parh93] Parhami, B., "On the Implementation of Arithmetic Support Functions for Generalized Signed-Digit Number Systems," *IEEE Trans. Computers*, Vol. 42, No. 3, pp. 379–384, 1993.

[Parh96] Parhami, B., "Comments on 'High-Speed Area-Efficient Multiplier Design Using Multiple-Valued Current Mode Circuits,' " *IEEE Trans. Computers*, Vol. 45, No. 5, pp. 637–638, 1996.

[Parh98] Parhami, B., and S. Johansson, "A Number Representation Scheme with Carry-Free Rounding for Floating-Point Signal Processing Applications," *Proc. Int'l. Conf. Signal and Image Processing*, Las Vegas, Nevada, October 1998, pp. 90–92.

[Phat94] Phatak, D. S., and I. Koren, "Hybrid Signed-Digit Number Systems: A Unified Framework for Redundant Number Representations with Bounded Carry Propagation Chains," *IEEE Trans. Computers*, Vol. 43, No. 8, pp. 880–891, 1994.

Chapter 4 RESIDUE NUMBER SYSTEMS

By converting arithmetic on large numbers to arithmetic on a collection of smaller numbers, residue number system (RNS) representations produce significant speedup for some classes of arithmetic-intensive algorithms in signal processing applications. Additionally, RNS arithmetic is a valuable tool for theoretical studies of the limits of fast arithmetic. In this chapter, we study RNS representations and arithmetic, along with their advantages and drawbacks. Chapter topics include:

4.1 RNS REPRESENTATION AND ARITHMETIC

What number has the remainders of 2, 3, and 2 when divided by the numbers 7, 5, and 3, respectively? This puzzle, written in the form of a verse by the Chinese scholar Sun Tsu more than 1500 years ago [Jenk93], is perhaps the first documented use of number representation using multiple residues. The puzzle essentially asks us to convert the coded representation (2 | 3 | 2) of a residue number system, based on the moduli (7 | 5 | 3), into standard decimal format.

In a residue number system (RNS), a number x is represented by the list of its residues with respect to k pairwise relatively prime moduli $m_{k-1} > \cdots > m_1 > m_0$. The residue x_i of x with respect to the ith modulus m_i is akin to a digit and the entire k-residue representation of x can be viewed as a k-digit number, where the digit set for the ith position is $[0, m_i - 1]$. Notationally, we write

$$x_i = x \bmod m_i = \langle x \rangle_{m_i}$$

and specify the RNS representation of x by enclosing the list of residues, or digits, in parentheses. For example,

$$x = (2|3|2)_{\text{RNS}(7|5|3)}$$

represents the puzzle given at the beginning of this section. The list of moduli can be deleted from the subscript when we have agreed on a default set. In many of the examples of this chapter, the following RNS is assumed:

RNS(8|7|5|3) Default RNS for Chapter 4

The product M of the k pairwise relatively prime moduli is the number of different representable values in the RNS and is known as its *dynamic range*.

$$M = m_{k-1} \times \cdots \times m_1 \times m_0$$

For example, $M = 8 \times 7 \times 5 \times 3 = 840$ is the total number of distinct values that are representable in our chosen 4-modulus RNS. Because of the equality

$$\langle -x \rangle_{m_i} = \langle M - x \rangle_{m_i}$$

the 840 available values can be used to represent numbers 0 through 839, -420 through $+419$, or any other interval of 840 consecutive integers. In effect, negative numbers are represented using a complement system with the complementation constant M.

Here are some example numbers in RNS(8|7|5|3) :

$(0 \mid 0 \mid 0 \mid 0)_{\text{RNS}}$ Represents 0 or 840 or $\cdots$
$(1 \mid 1 \mid 1 \mid 1)_{\text{RNS}}$ Represents 1 or 841 or $\cdots$
$(2 \mid 2 \mid 2 \mid 2)_{\text{RNS}}$ Represents 2 or 842 or $\cdots$
$(0 \mid 1 \mid 3 \mid 2)_{\text{RNS}}$ Represents 8 or 848 or $\cdots$
$(5 \mid 0 \mid 1 \mid 0)_{\text{RNS}}$ Represents 21 or 861 or $\cdots$
$(0 \mid 1 \mid 4 \mid 1)_{\text{RNS}}$ Represents 64 or 904 or $\cdots$
$(2 \mid 0 \mid 0 \mid 2)_{\text{RNS}}$ Represents -70 or 770 or $\cdots$
$(7 \mid 6 \mid 4 \mid 2)_{\text{RNS}}$ Represents -1 or 839 or $\cdots$

Given the RNS representation of x, the representation of $-x$ can be found by complementing each of the digits x_i with respect to its modules m_i (0 digits are left unchanged). Thus, given that $21 = (5 \mid 0 \mid 1 \mid 0)_{\text{RNS}}$, we find:

$$-21 = (8-5 \mid 0 \mid 5 \ -1 \mid 0)_{\text{RNS}} = (3 \mid 0 \mid 4 \mid 0)_{\text{RNS}}$$

Any RNS can be viewed as a weighted representation. We will present a general method for determining the position weights (the Chinese remainder theorem) in Section 4.3. For RNS(8|7|5|3), the weights associated with the four positions are:

105 120 336 280

As an example, $(1 \mid 2 \mid 4 \mid 0)_{\text{RNS}}$ represents the number:

$$\langle (105 \times 1) + (120 \times 2) + (336 \times 4) + (280 \times 0) \rangle_{840} = \langle 1689 \rangle_{840} = 9$$

In practice, each residue must be represented or encoded in binary. For our example RNS, such a representation would require 11 bits (Fig. 4.1). To determine the number representation efficiency of our 4-modulus RNS, we note that 840 different values are being represented using 11 bits, compared to 2048 values possible with binary representation. Thus, the representational efficiency is

$$840/2048 = 41\%$$

Since $\log_2 840 = 9.714$, another way to quantify the representational efficiency is to note that in our example RNS, about 1.3 bits of the 11 bits goes to waste.

As noted earlier, the sign of an RNS number can be changed by independently complementing each of its digits with respect to its modulus. Similarly, addition, subtraction, and multiplication can be performed by independently operating on each digit. The following examples for RNS(8|7|5|3) illustrate the process:

$(5 \mid 5 \mid 0 \mid 2)_{\text{RNS}}$ Represents $x = +5$

$(7 \mid 6 \mid 4 \mid 2)_{\text{RNS}}$ Represents $y = -1$

$(4 \mid 4 \mid 4 \mid 1)_{\text{RNS}}$ $x + y$: $\langle 5 + 7 \rangle_8 = 4$, $\langle 5 + 6 \rangle_7 = 4$, etc.

$(6 \mid 6 \mid 1 \mid 0)_{\text{RNS}}$ $x - y$: $\langle 5 - 7 \rangle_8 = 6$, $\langle 5 - 6 \rangle_7 = 6$, etc.
(alternatively, find $-y$ and add to x)

$(3 \mid 2 \mid 0 \mid 1)_{\text{RNS}}$ $x \times y$: $\langle 5 \times 7 \rangle_8 = 3$, $\langle 5 \times 6 \rangle_7 = 2$, etc.

Figure 4.2 depicts the structure of an adder, subtractor, or multiplier for RNS arithmetic. Since each digit is a relatively small number, these operations can be quite fast and simple in RNS. This speed and simplicity are the primary advantages of RNS arithmetic. In the case of addition, for example, carry propagation is limited to within a single residue (a few bits). Thus, RNS representation pretty much solves the carry propagation problem. As for multiplication, a 4×4 multiplier (e.g.), is considerably more than four times simpler than a 16×16 multiplier, besides being much faster. In fact, since the residues are small (say, 6 bits wide), it is quite feasible to implement addition, subtraction, and multiplication by direct table lookup. With 6-bit residues, say, each operation requires a $4\text{K} \times 6$ table. Thus, excluding division, a complete arithmetic unit module for one 6-bit residue can be implemented with 9 KB of memory.

Unfortunately, however, what we gain in terms of the speed and simplicity of addition, subtraction, and multiplication can be more than nullified by the complexity of division and the difficulty of certain auxiliary operations such as sign test, magnitude comparison, and overflow detection. Given the numbers

$$(7 \mid 2 \mid 2 \mid 1)_{\text{RNS}} \quad \text{and} \quad (2 \mid 5 \mid 0 \mid 1)_{\text{RNS}}$$

we cannot easily tell their signs, determine which of the two is larger, or find out whether $(1 \mid 0 \mid 2 \mid 2)_{\text{RNS}}$ represents their true sum as opposed to the residue of their sum modulo 840.

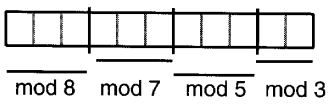


Fig. 4.1 Binary-coded number format for RNS(8|7|5|3).

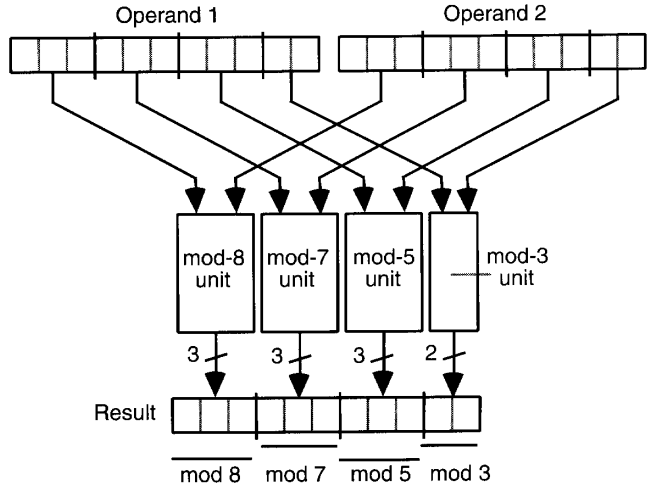


Fig. 4.2 The structure of an adder, subtractor, or multiplier for RNS(8|7|5|3).

These difficulties have thus far limited the application of RNS representations to certain signal processing problems in which additions and multiplications are used either exclusively or predominantly and the results are within known ranges (e.g., digital filters, Fourier transforms). Developments in recent years [Hung94] have greatly lessened the penalty for division and sign detection and may lead to more widespread applications for RNS in future. We discuss division and other "difficult" RNS operations in Section 4.4.

4.2 CHOOSING THE RNS MODULI

The set of the moduli chosen for RNS affects both the representational efficiency and the complexity of arithmetic algorithms. In general, we try to make the moduli as small as possible, since it is the magnitude of the largest modulus m_{k-1} that dictates the speed of arithmetic operations. We also often try to make all the moduli comparable in magnitude to the largest one, since with the computation speed already dictated by m_{k-1}, there is usually no advantage in fragmenting the design of Fig. 4.2 through the use of very small moduli at the right end.

We illustrate the process of selecting the RNS moduli through an example. Let us assume that we want to represent unsigned integers in the range 0 to $(100\,000)_{\text{ten}}$, requiring 17 bits with standard binary representation.

A simple strategy is to pick prime numbers in sequence until the dynamic range M becomes adequate. Thus, we pick $m_0 = 2, m_1 = 3, m_2 = 5$, etc. After we add $m_5 = 13$ to our list, the dynamic range becomes:

$$\text{RNS}(13 \mid 11 \mid 7 \mid 5 \mid 3 \mid 2) \qquad M = 30\,030$$

This range is not yet adequate, so we add $m_6 = 17$ to the list:

$$\text{RNS}(17 \mid 13 \mid 11 \mid 7 \mid 5 \mid 3 \mid 2) \qquad M = 510\,510$$

The dynamic range is now 5.1 times larger than needed, so we can remove the modulus 5 and still have adequate range:

$$\text{RNS}(17 \mid 13 \mid 11 \mid 7 \mid 3 \mid 2) \qquad M = 102\ 102$$

With binary encoding of the six residues, the number of bits needed for encoding each number is:

$$5 + 4 + 4 + 3 + 2 + 1 = 19 \text{ bits}$$

Now, since the speed of arithmetic operations is dictated by the 5-bit residues modulo m_5, we can combine the pairs of moduli 2 and 13, and 3 and 7, with no speed penalty. This leads to:

$$\text{RNS}(26 \mid 21 \mid 17 \mid 11) \qquad M = 102\ 102$$

This alternative RNS still needs $5+5+5+4 = 19$ bits per operand, but has two fewer modules in the arithmetic unit.

Better results can be obtained if we proceed as above, but include powers of smaller primes before moving to larger primes. The chosen moduli will still be pairwise relatively prime, since powers of any two prime numbers are relatively prime. For example, after including $m_0 = 2$ and $m_1 = 3$ in our list of moduli, we note that 2^2 is smaller than the next prime 5. So we modify m_0 and m_1 to get:

$$\text{RNS}(2^2 \mid 3) \qquad M = 12$$

This strategy is consistent with our desire to minimize the magnitude of the largest modulus. Similarly, after we have included $m_2 = 5$ and $m_3 = 7$, we note that both 2^3 and 3^2 are smaller than the next prime 11. So the next three steps lead to:

$$\begin{aligned}
&\text{RNS}(3^2 \mid 2^3 \mid 7 \mid 5) && M = 2520 \\
&\text{RNS}(11 \mid 3^2 \mid 2^3 \mid 7 \mid 5) && M = 27\ 720 \\
&\text{RNS}(13 \mid 11 \mid 3^2 \mid 2^3 \mid 7 \mid 5) && M = 360\ 360
\end{aligned}$$

The dynamic range is now 3.6 times larger than needed, so we can replace the modulus 9 with 3 and then combine the pair 5 and 3 to obtain:

$$\text{RNS}(15 \mid 13 \mid 11 \mid 2^3 \mid 7) \qquad M = 120\ 120$$

The number of bits needed by this last RNS is

$$4 + 4 + 4 + 3 + 3 = 18 \text{ bits}$$

which is better than our earlier result of 19 bits. The speed has also improved because the largest residue is now 4 bits wide instead of 5.

Other variations are possible. For example, given the simplicity of operations with power-of-2 moduli, we might want to backtrack and maximize the size of our even modulus within the 4-bit residue limit:

$$\text{RNS}(2^4 \mid 13 \mid 11 \mid 3^2 \mid 7 \mid 5) \qquad M = 720\ 720$$

We can now remove 5 or 7 from the list of moduli, but the resulting RNS is in fact inferior to RNS(15|13|11|2^3|7). This might not be the case with other examples; thus, once we have converged on a feasible set of moduli, we should experiment with other sets that can be derived from it by increasing the power of the even modulus at hand.

The preceding strategy for selecting the RNS moduli is guaranteed to lead to the smallest possible number of bits for the largest modulus, thus maximizing the speed of RNS arithmetic. However, speed and cost do not just depend on the widths of the residues but also on the moduli chosen. For example, we have already noted that power-of-2 moduli simplify the required arithmetic operations, so that the modulus 16 might be better than the smaller modulus 13 (except, perhaps, with table-lookup implementation). Moduli of the form $2^a - 1$ are also desirable and are referred to as *low-cost* moduli [Merr64], [Parh76]. From our discussion of addition of 1's-complement numbers in Section 2.4, we know that addition modulo $2^a - 1$ can be performed using a standard a-bit binary adder with end-around carry.

Hence, we are motivated to restrict the moduli to a power of 2 and odd numbers of the form $2^a - 1$. One can prove (left as exercise) that the numbers $2^a - 1$ and $2^b - 1$ are relatively prime if and only if a and b are relatively prime. Thus, any list of relatively prime numbers $a_{k-2} > \cdots > a_1 > a_0$ can be the basis of the following k-modulus RNS

$$\text{RNS}(2^{a_{k-2}} \mid 2^{a_{k-2}} - 1 \mid \cdots \mid 2^{a_1} - 1 \mid 2^{a_0} - 1)$$

for which the widest residues are a_{k-2}-bit numbers. Note that to maximize the dynamic range with a given residue width, the even modulus is chosen to be as large as possible.

Applying this strategy to our desired RNS with the target range [0, 100 000], leads to the following steps:

RNS($2^3 \mid 2^3 - 1 \mid 2^2 - 1$)	Basis: 3, 2	$M = 168$
RNS($2^4 \mid 2^4 - 1 \mid 2^3 - 1$)	Basis: 4, 3	$M = 1680$
RNS($2^5 \mid 2^5 - 1 \mid 2^3 - 1 \mid 2^2 - 1$)	Basis: 5, 3, 2	$M = 20\ 832$
RNS($2^5 \mid 2^5 - 1 \mid 2^4 - 1 \mid 2^3 - 1$)	Basis: 5, 4, 3	$M = 104\ 160$

This last system, RNS(32 | 31 | 15 | 7), possesses adequate range. Note that once the number 4 is included in the base list, 2 must be excluded because 4 and 2, and thus $2^4 - 1$ and $2^2 - 1$, are not relatively prime.

The derived RNS requires $5 + 5 + 4 + 3 = 17$ bits for representing each number, with the largest residues being 5 bits wide. In this case, the representational efficiency is close to 100% and no bit is wasted. In general, the representational efficiency of low-cost RNS is provably better than 50% (yet another exercise!), leading to the waste of no more than 1 bit in number representation.

To compare the RNS above to our best result with unrestricted moduli, we list the parameters of the two systems together:

RNS(15 | 13 | 11 | 2^3 | 7)	18 bits	$M = 120\ 120$
RNS($2^5 \mid 2^5 - 1 \mid 2^4 - 1 \mid 2^3 - 1$)	17 bits	$M = 104\ 160$

Both systems provide the desired range. The latter has wider, but fewer, residues. However, the simplicity of arithmetic with low-cost moduli makes the latter a more attractive choice. In general, restricting the moduli tends to increase the width of the largest residues and the optimal choice is dependent on both the application and the target implementation technology.

4.3 ENCODING AND DECODING OF NUMBERS

Since input numbers provided from the outside (machine or human interface) are in standard binary or decimal and outputs must be presented in the same way, conversions between binary/decimal and RNS representations are required.

Conversion from binary/decimal to RNS

The binary-to-RNS conversion problem is stated as follows: Given a number y, find its residues with respect to the moduli $m_i, 0 \leq i \leq k-1$. Let us assume that y is an unsigned binary number. Conversion of signed-magnitude or 2's-complement numbers can be accomplished by converting the magnitude and then complementing the RNS representation if needed.

To avoid time-consuming divisions, we take advantage of the following equality:

$$\langle (y_{k-1} \cdots y_1 y_0)_{\text{two}} \rangle_{m_i} = \langle \langle 2^{k-1} y_{k-1} \rangle_{m_i} + \cdots + \langle 2y_1 \rangle_{m_i} + \langle y_0 \rangle_{m_i} \rangle_{m_i}$$

If we precompute and store $\langle 2^j \rangle_{m_i}$ for each i and j, then the residue x_i of y (mod m_i) can be computed by modulo-m_i addition of some of these constants.

Table 4.1 shows the required lookup table for converting 10-bit binary numbers in the range [0, 839] to RNS(8 | 7 | 5 | 3). Only residues mod 7, mod 5, and mod 3 are given in the table, since the residue mod 8 is directly available as the 3 least significant bits of the binary number y.

Example 4.1 Represent $y = (1010\ 0100)_{\text{two}} = (164)_{\text{ten}}$ in RNS(8 | 7 | 5 | 3).

The residue of y mod 8 is $x_3 = (y_2 y_1 y_0)_{\text{two}} = (100)_{\text{two}} = 4$. Since $y = 2^7 + 2^5 + 2^2$, the required residues mod 7, mod 5, and mod 3 are obtained by simply adding the values stored in the three rows corresponding to $j = 7, 5, 2$ in Table 4.1:

$$x_2 = \langle y \rangle_7 = \langle 2 + 4 + 4 \rangle_7 = 3$$
$$x_1 = \langle y \rangle_5 = \langle 3 + 2 + 4 \rangle_5 = 4$$
$$x_0 = \langle y \rangle_3 = \langle 2 + 2 + 1 \rangle_3 = 2$$

Therefore, the RNS(8 | 7 | 5 | 3) representation of $(164)_{\text{ten}}$ is $(4 \mid 3 \mid 4 \mid 2)_{\text{RNS}}$.

In the worst case, k modular additions are required for computing each residue of a k-bit number. To reduce the number of operations, one can view the given input number as a number in a higher radix. For example, if we use radix 4, then storing the residues of 4^i, 2×4^i and 3×4^i in a table would allow us to compute each of the required residues using only $k/2$ modular additions.

The conversion for each modulus can be done by repeatedly using a single lookup table and modular adder or by several copies of each arranged into a pipeline. For a low-cost modulus $m = 2^a - 1$, the residue can be determined by dividing up y into a-bit segments and adding them modulo $2^a - 1$.

TABLE 4.1
Precomputed residues of the first 10 powers of 2

j	2^j	$\langle 2^j \rangle_7$	$\langle 2^j \rangle_5$	$\langle 2^j \rangle_3$
0	1	1	1	1
1	2	2	2	2
2	4	4	4	1
3	8	1	3	2
4	16	2	1	1
5	32	4	2	2
6	64	1	4	1
7	128	2	3	2
8	256	4	1	1
9	512	1	2	2

Conversion from RNS to mixed-radix form

Associated with any residue number system RNS($m_{k-1} \mid \cdots \mid m_2 \mid m_1 \mid m_0$) is a mixed-radix number system MRS($m_{k-1} \mid \cdots \mid m_2 \mid m_1 \mid m_0$), which is essentially a k-digit positional number system with position weights

$$m_{k-2} \cdots m_2 m_1 m_0 \quad \cdots \quad m_2 m_1 m_0 \quad m_1 m_0 \quad m_0 \quad 1$$

and digit sets $[0, m_{k-1} - 1], \cdots, [0, m_2 - 1], [0, m_1 - 1]$, and $[0, m_0 - 1]$ in its k digit positions. Hence, the MRS digits are in the same ranges as the RNS digits (residues). For example, the mixed-radix system MRS(8 | 7 | 5 | 3) has position weights $7 \times 5 \times 3 = 105, 5 \times 3 = 15, 3$, and 1, leading to:

$$(0 \mid 3 \mid 1 \mid 0)_{\text{MRS}(8 \mid 7 \mid 5 \mid 3)} = (0 \times 105) + (3 \times 15) + (1 \times 3) + (0 \times 1) = 48$$

The RNS-to-MRS conversion problem is that of determining the z_i digits of MRS, given the x_i digits of RNS, so that:

$$y = (x_{k-1} \mid \cdots \mid x_2 \mid x_1 \mid x_0)_{\text{RNS}} = (z_{k-1} \mid \cdots \mid z_2 \mid z_1 \mid z_0)_{\text{MRS}}$$

From the definition of MRS, we have:

$$y = z_{k-1}(m_{k-2} \cdots m_2 m_1 m_0) + \cdots + z_2(m_1 m_0) + z_1(m_0) + z_0$$

It is thus immediately obvious that $z_0 = x_0$. Subtracting $z_0 = x_0$ from both the RNS and MRS representations, we get

$$y - x_0 = (x'_{k-1} \mid \cdots \mid x'_2 \mid x'_1 \mid 0)_{\text{RNS}} = (z_{k-1} \mid \cdots \mid z_2 \mid z_1 \mid 0)_{\text{MRS}}$$

where $x'_j = \langle x_j - x_0 \rangle_{m_j}$. If we now divide both representations by m_0, we get the following in the reduced RNS and MRS from which m_0 has been removed:

$$(x''_{k-1} \mid \cdots \mid x''_2 \mid x''_1)_{\text{RNS}} = (z_{k-1} \mid \cdots \mid z_2 \mid z_1)_{\text{MRS}}$$

Thus, if we demonstrate how to divide the number $y' = (x'_{k-1} \mid \cdots \mid x'_2 \mid x'_1 \mid 0)_{\text{RNS}}$ by m_0 to obtain $(x''_{k-1} \mid \cdots \mid x''_2 \mid x''_1)_{\text{RNS}}$, we have converted the original problem to a similar problem with one fewer modulus. Repeating the same process then leads to the determination of all the z_i digits in turn.

Dividing y', which is a multiple of m_0, by m_0 is known as *scaling* and is much simpler than general division in RNS. Division by m_0 can be accomplished by multiplying each residue by the *multiplicative inverse* of m_0 with respect to the associated modulus. For example, the multiplicative inverses of 3 relative to 8, 7, and 5 are 3, 5, and 2, respectively, because:

$$\langle 3 \times 3 \rangle_8 = \langle 3 \times 5 \rangle_7 = \langle 3 \times 2 \rangle_5 = 1$$

Thus, the number $y' = (0 \mid 6 \mid 3 \mid 0)_{\text{RNS}}$ can be divided by 3 through multiplication by $(3 \mid 5 \mid 2 \mid -)_{\text{RNS}}$:

$$\frac{(0 \mid 6 \mid 3 \mid 0)_{\text{RNS}}}{3} = (0 \mid 6 \mid 3 \mid 0)_{\text{RNS}} \times (3 \mid 5 \mid 2 \mid -)_{\text{RNS}} = (0 \mid 2 \mid 1 \mid -)_{\text{RNS}}$$

Multiplicative inverses of the moduli can be precomputed and stored in tables to facilitate RNS-to-MRS conversion.

Example 4.2 Convert $y = (0 \mid 6 \mid 3 \mid 0)_{\text{RNS}}$ to mixed-radix representation.

We have $z_0 = x_0 = 0$. Based on the preceding discussion, dividing y by 3 yields:

$$\frac{(0 \mid 6 \mid 3 \mid 0)_{\text{RNS}}}{3} = (0 \mid 6 \mid 3 \mid 0)_{\text{RNS}} \times (3 \mid 5 \mid 2 \mid -)_{\text{RNS}} = (0 \mid 2 \mid 1 \mid -)_{\text{RNS}}$$

Thus we have $z_1 = 1$. Subtracting 1 and dividing by 5, we get:

$$\frac{(7 \mid 1 \mid 0 \mid -)_{\text{RNS}}}{5} = (7 \mid 1 \mid 0 \mid -)_{\text{RNS}} \times (5 \mid 3 \mid - \mid -)_{\text{RNS}} = (3 \mid 3 \mid - \mid -)_{\text{RNS}}$$

Next, we get $z_2 = 3$. Subtracting 3 and dividing by 7, we find:

$$\frac{(0 \mid 0 \mid - \mid -)_{\text{RNS}}}{7} = (0 \mid 0 \mid - \mid -)_{\text{RNS}} \times (7 \mid - \mid - \mid -)_{\text{RNS}}$$

$$= (0 \mid - \mid - \mid -)_{\text{RNS}}$$

We conclude by observing that $z_3 = 0$. The conversion is now complete:

$$y = (0 \mid 6 \mid 3 \mid 0)_{\text{RNS}} = (0 \mid 3 \mid 1 \mid 0)_{\text{MRS}} = 48$$

Mixed-radix representation allows us to compare the magnitudes of two RNS numbers or to detect the sign of a number. For example, the RNS representations $(0 \mid 6 \mid 3 \mid 0)_{\text{RNS}}$ and $(5 \mid 3 \mid 0 \mid 0)_{\text{RNS}}$ of 48 and 45 provide no clue to their relative magnitudes, whereas the equivalent mixed-radix representations $(0 \mid 3 \mid 1 \mid 0)_{\text{MRS}}$ and $(0 \mid 3 \mid 0 \mid 0)_{\text{MRS}}$, or $(000 \mid 011 \mid 001 \mid 00)_{\text{MRS}}$ and $(000 \mid 011 \mid 000 \mid 00)_{\text{MRS}}$, when coded in binary, can be compared as ordinary numbers.

Conversion from RNS to binary/decimal

One method for RNS-to-binary conversion is to first derive the mixed-radix representation of the RNS number and then use the weights of the mixed-radix positions to complete the conversion. We can also derive position weights for the RNS directly based on the Chinese remainder theorem (CRT), as discussed below.

Consider the conversion of $y = (3 \mid 2 \mid 4 \mid 2)_{\text{RNS}}$ from RNS(8 | 7 | 5 | 3) to decimal. Based on RNS properties, we can write:

$$\begin{aligned}(3 \mid 2 \mid 4 \mid 2)_{\text{RNS}} &= (3 \mid 0 \mid 0 \mid 0)_{\text{RNS}} + (0 \mid 2 \mid 0 \mid 0)_{\text{RNS}} \\ &\quad + (0 \mid 0 \mid 4 \mid 0)_{\text{RNS}} + (0 \mid 0 \mid 0 \mid 2)_{\text{RNS}} \\ &= 3 \times (1 \mid 0 \mid 0 \mid 0)_{\text{RNS}} + 2 \times (0 \mid 1 \mid 0 \mid 0)_{\text{RNS}} \\ &\quad + 4 \times (0 \mid 0 \mid 1 \mid 0)_{\text{RNS}} + 2 \times (0 \mid 0 \mid 0 \mid 1)_{\text{RNS}}\end{aligned}$$

Thus, knowing the values of the following four constants (the RNS position weights) would allow us to convert any number from RNS(8|7|5|3) to decimal using four multiplications and three additions.

$$\begin{aligned}(1 \mid 0 \mid 0 \mid 0)_{\text{RNS}} &= 105 \\ (0 \mid 1 \mid 0 \mid 0)_{\text{RNS}} &= 120 \\ (0 \mid 0 \mid 1 \mid 0)_{\text{RNS}} &= 336 \\ (0 \mid 0 \mid 0 \mid 1)_{\text{RNS}} &= 280\end{aligned}$$

Thus, we find:

$$(3 \mid 2 \mid 4 \mid 2)_{\text{RNS}} = \langle (3 \times 105) + (2 \times 120) + (4 \times 336) + (2 \times 280) \rangle_{840} = 779$$

It only remains to show how the preceding weights were derived. How, for example, did we determine that $w_3 = (1 \mid 0 \mid 0 \mid 0)_{\text{RNS}} = 105$?

To determine the value of w_3, we note that it is divisible by 3, 5, and 7, since its last three residues are 0s. Hence, w_3 must be a multiple of 105. We must then pick the right multiple of 105 such that its residue with respect to 8 is 1. This is done by multiplying 105 by its multiplicative inverse with respect to 8. Based on the preceding discussion, the conversion process can be formalized in the form of the Chinese remainder theorem.

THEOREM 4.1 (The Chinese remainder theorem) The magnitude of an RNS number can be obtained from the CRT formula:

$$x = (x_{k-1} \mid \cdots \mid x_2 \mid x_1 \mid x_0)_{\text{RNS}} = \left\langle \sum_{i=0}^{k-1} M_i \langle \alpha_i x_i \rangle_{m_i} \right\rangle_M$$

where, by definition, $M_i = M/m_i$, and $\alpha_i = \langle M_i^{-1} \rangle_{m_i}$ is the multiplicative inverse of M_i with respect to m_i.

TABLE 4.2
Values needed in applying the Chinese remainder theorem to RNS(8|7|5|3)

i	m_i	x_i	$\langle M_i \langle \alpha_i x_i \rangle_{m_i} \rangle_M$
3	8	0	0
		1	105
		2	210
		3	315
		4	420
		5	525
		6	630
		7	735
2	7	0	0
		1	120
		2	240
		3	360
		4	480
		5	600
		6	720
1	5	0	0
		1	336
		2	672
		3	168
		4	504
0	3	0	0
		1	280
		2	560

To avoid multiplications in the conversion process, we can store the values of $\langle M_i \langle \alpha_i x_i \rangle_{m_i} \rangle_M$ for all possible i and x_i in tables of total size $\sum_{i=0}^{k-1} m_i$ words. Table 4.2 shows the required values for RNS(8|7|5|3). Conversion is then performed exclusively by table lookups and modulo-M additions.

4.4 DIFFICULT RNS ARITHMETIC OPERATIONS

In this section, we discuss algorithms and hardware designs for sign test, magnitude comparison, overflow detection, and general division in RNS. The first three of these operations are essentially equivalent in that if an RNS with dynamic range M is used for representing signed numbers in the range $[-N, P]$, with $M = N + P + 1$, then sign test is the same as comparison with P and overflow detection can be performed based on the signs of the operands and that of the result. Thus, it suffices to discuss magnitude comparison and general division.

To compare the magnitudes of two RNS numbers, we can convert both to binary or mixed-radix form. However, this would involve a great deal of overhead. A more efficient approach is through approximate CRT decoding. Dividing the equality in the statement of Theorem 4.1 by M, we obtain the following expression for the scaled value of x in $[0, 1)$:

$$\frac{x}{M} = \frac{(x_{k-1} \mid \cdots \mid x_2 \mid x_1 \mid x_0)_{\text{RNS}}}{M} = \langle \sum_{i=0}^{k-1} m_i^{-1} \langle \alpha_i x_i \rangle_{m_i} \rangle_1$$

Here, the addition of terms is performed modulo 1, meaning that in adding the terms $m_i^{-1}\langle \alpha_i x_i \rangle_{m_i}$, each of which is in [0, 1), the whole part of the result is discarded and only the fractional part is kept; this is much simpler than the modulo-M addition needed in standard CRT decoding.

Again, the terms $m_i^{-1}\langle \alpha_i x_i \rangle_{m_i}$ can be precomputed for all possible i and x_i and stored in tables of total size $\sum_{i=0}^{k-1} m_i$ words. Table 4.3 shows the required lookup table for approximate CRT decoding in RNS(8|7|5|3). Conversion is then performed exclusively by table lookups and modulo-1 additions (i.e., fractional addition, with the carry-out simply ignored).

Example 4.3 Use approximate CRT decoding to determine the larger of the two numbers $x = (0|6|3|0)_{\text{RNS}}$ and $y = (5|3|0|0)_{\text{RNS}}$.

Reading values from Table 4.3, we get:

$$\frac{x}{M} \approx \langle .0000 + .8571 + .2000 + .0000 \rangle_1 \approx .0571$$

$$\frac{y}{M} \approx \langle .6250 + .4286 + .0000 + .0000 \rangle_1 \approx .0536$$

Thus, we can conclude that $x > y$, subject to approximation errors to be discussed next.

If the maximum error in each table entry is ε, then approximate CRT decoding yields the scaled value of an RNS number with an error of no more than $k\varepsilon$. In the preceding example, assuming that the table entries have been rounded to four decimal digits, the maximum error in each entry is $\varepsilon = 0.00005$ and the maximum error in the scaled value is $4\varepsilon = 0.0002$. The conclusion $x > y$ is, therefore, safe.

Of course we can use highly precise table entries to avoid the possibility of erroneous conclusions altogether. But this would defeat the advantage of approximate CRT decoding in simplicity and speed. Thus, in practice, a two-stage process might be envisaged: a quick approximate decoding process is performed first, with the resulting scaled value(s) and error bound(s) used to decide whether a more precise or exact decoding is needed for arriving at a conclusion.

In many practical situations, an exact comparison of x and y might not be required and a ternary decision result $x < y$, $x \approx y$ (i.e., too close to call), or $x > y$ might do. In such cases, approximate CRT decoding is just the right tool. For example, in certain division algorithms (to be discussed in Chapter 14), the sign and the magnitude of the partial remainder s are used to choose the next quotient digit q_j from the redundant digit set $[-1, 1]$ according to the following:

$$\begin{aligned} s < 0 &\quad \text{quotient digit} = -1 \\ s \approx 0 &\quad \text{quotient digit} = \ \ 0 \\ s > 0 &\quad \text{quotient digit} = \ \ 1 \end{aligned}$$

In this case, the algorithm's built-in tolerance to imprecision allows us to use it for RNS division. Once the quotient digit in $[-1, 1]$ has been chosen, the value $q_j d$, where d is the divisor, is subtracted from the partial remainder to obtain the new partial remainder for the next iteration.

TABLE 4.3
Values needed in applying approximate Chinese remainder theorem decoding to RNS(8|7|5|3)

i	m_i	x_i	$m_i^{-1}\langle\alpha_i x_i\rangle_{m_i}$
3	8	0	.0000
		1	.1250
		2	.2500
		3	.3750
		4	.5000
		5	.6250
		6	.7500
		7	.8750
2	7	0	.0000
		1	.1429
		2	.2857
		3	.4286
		4	.5714
		5	.7143
		6	.8571
1	5	0	.0000
		1	.4000
		2	.8000
		3	.2000
		4	.6000
0	3	0	.0000
		1	.3333
		2	.6667

Also, the quotient, derived in positional radix-2 format using the digit set $[-1, 1]$, is converted to RNS on the fly.

In other division algorithms, to be discussed in Chapters 14 and 15, approximate comparison of the partial remainder s and divisor d is used to choose a radix-r quotient digit in $[-\alpha, \beta]$. An example includes radix-4 division with the quotient digit set $[-2, 2]$. In these cases, too, approximate CRT decoding can be used to facilitate RNS division [Hung94].

4.5 REDUNDANT RNS REPRESENTATIONS

Just as the digits in a positional radix-r number system do not have to be restricted to the set $[0, r-1]$, we are not obliged to limit the residue digits for the modulus m_i to the set $[0, m_i - 1]$. Instead, we can agree to use the digit set $[0, \beta_i]$ for the mod-m_i residue, provided $\beta_i \geq m_i - 1$. If $\beta_i \geq m_i$, then the resulting RNS is redundant.

One reason to use redundant residues is to simplify the modular reduction step needed after each arithmetic operation. Consider, for example, the representation of mod-13 residues using 4-bit binary numbers. Instead of using residues in [0, 12], we can use pseudoresidues in [0, 15].

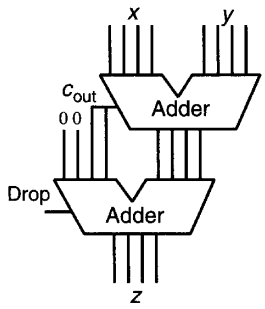


Figure 4.3 Adder design for 4-bit mod-13 pseudoresidues.

Residues 0, 1, and 2 will then have two representations, since $13 = 0 \bmod 13$, $14 = 1 \bmod 13$, and $15 = 2 \bmod 13$. Addition of such pseudoresidues can be performed by a 4-bit binary adder. If the carry-out is 0, the addition result is kept intact; otherwise, the carry-out, which is worth 16 units, is dropped and 3 is added to the result. Thus, the required mod-13 addition unit is as shown in Fig. 4.3.

One can go even further and make the pseudoresidues $2h$ bits wide, where normal mod-m residues would be only h bits wide. This simplifies a multiply-accumulate operation, which is done by adding the $2h$-bit product of two normal residues to a $2h$-bit running total, reducing the $(2h + 1)$-bit result to a $2h$-bit pseudoresidue for the next step by subtracting $2^h m$ from it if needed (Fig. 4.4). Reduction to a standard h-bit residue is then done only once at the end of accumulation.

4.6 LIMITS OF FAST ARITHMETIC IN RNS

How much faster is RNS arithmetic than conventional (say, binary) arithmetic? We will see later in Chapters 6 and 7 that addition of binary numbers in the range $[0, M - 1]$ can be done in

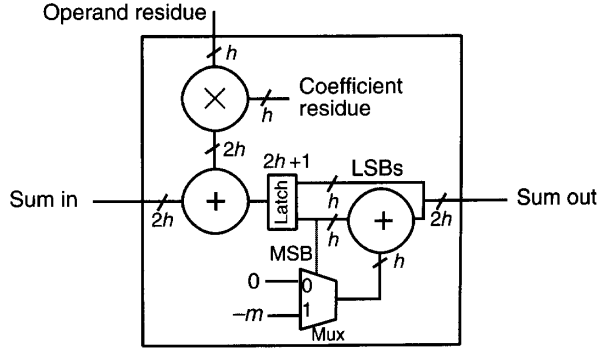


Fig. 4.4 A modulo-m multiply-add cell that accumulates the sum into a double-length redundant pseudoresidue.

O(log log M) time and with O(log M) cost using a variety of methods such as carry-lookahead, conditional-sum, or multilevel carry-select. Both these are optimal to within constant factors, given the fixed-radix positional representation. For example, one can use the constant fan-in argument to establish that the circuit depth of an adder must be at least logarithmic in the number $k = \log_r M$ of digits. Redundant representations allow O(1)-time, O(log M)-cost addition. What is the best one can do with RNS arithmetic?

Consider the residue number system RNS($m_{k-1}|\cdots|m_1|m_0$). Assume that the moduli are chosen as the smallest possible prime numbers to minimize the size of the moduli, and thus maximize computation speed. The following theorems from number theory help us in figuring out the complexity.

THEOREM 4.2 The ith prime p_i is asymptotically equal to $i \ln i$.

THEOREM 4.3 The number of primes in $[1, n]$ is asymptotically equal to $n/(\ln n)$.

THEOREM 4.4 The product of all primes in $[1, n]$ is asymptotically equal to e^n.

Table 4.4 lists some numerical values that can help us understand the asymptotic approximations given in Theorems 4.2 and 4.3.

Armed with these results from number theory, we can derive an interesting limit on the speed of RNS arithmetic.

TABLE 4.4
The ith-prime p_i and the number of primes in [1, n] versus their asymptotic approximations

i	p_i	$i \ln i$	Error (%)	n	Number of primes in [1, n]	$n/(\ln n)$	Error (%)
1	2	0.000	100	5	2	3.107	55
2	3	1.386	54	10	4	4.343	9
3	5	3.296	34	15	6	5.539	8
4	7	5.545	21	20	8	6.676	17
5	11	8.047	27	25	9	7.767	14
10	29	23.03	21	30	10	8.820	12
15	47	40.62	14	40	12	10.84	10
20	71	59.91	16	50	15	12.78	15
30	113	102.0	10	100	25	21.71	13
40	173	147.6	15	200	46	37.75	18
50	229	195.6	15	500	95	80.46	15
100	521	460.5	12	1000	170	144.8	15

THEOREM 4.5 It is possible to represent all k-bit binary numbers in RNS with $O(k/\log k)$ moduli such that the largest modulus has $O(\log k)$ bits.

Proof: If the largest needed prime is n, by Theorem 4.4 we must have $e^n \approx 2^k$. This equality implies $n < k$. The number of moduli required is the number of primes less than n which by Theorem 4.3 is $O(n/\log n) = O(k/\log k)$.

As a result, addition of such residue numbers can be performed in $O(\log \log \log M)$ time and with $O(\log M)$ cost. So, the cost of addition is comparable to that of binary representation whereas the delay is much smaller, though not constant.

If for implementation ease, we limit ourselves to moduli of the form 2^a or $2^a - 1$, the following results from number theory are applicable.

THEOREM 4.6 The numbers $2^a - 1$ and $2^b - 1$ are relatively prime if and only if a and b are relatively prime.

THEOREM 4.7 The sum of the first i primes is asymptotically $O(i^2 \ln i)$.

These theorems allow us to prove the following asymptotic result for low-cost residue number systems.

THEOREM 4.8 It is possible to represent all k-bit binary numbers in RNS with $O((k/\log k)^{1/2})$ low-cost moduli of the form $2^a - 1$ such that the largest modulus has $O((k \log k)^{1/2})$ bits.

Proof: If the largest modulus that we need is $2^l - 1$, by Theorem 4.7 we must have $l^2 \ln l \approx k$. This implies that $l = O((k/\log k)^{1/2})$. By Theorem 4.2, the lth prime is approximately $p_l \approx l \ln l \approx O((k \log k)^{1/2})$. The proof is complete upon noting that to minimize the size of the moduli, we pick the ith modulus to be $2^{p_i} - 1$.

As a result, addition of low-cost residue numbers can be performed in $O(\log \log M)$ time with $O(\log M)$ cost and thus, asymptotically, offers little advantage over standard binary.

PROBLEMS

4.1 RNS representation and arithmetic Consider the RNS system RNS(15 | 13 | 11 | 8 | 7) derived in Section 4.2.

- **a.** Represent the numbers $x = 168$ and $y = 23$ in this RNS.
- **b.** Compute $x + y, x - y, x \times y$, checking the results via decimal arithmetic.
- **c.** Knowing that x is a multiple of 56, divide it by 56 in the RNS. *Hint:* $56 = 7 \times 8$.
- **d.** Compare the numbers $(5 \mid 4 \mid 3 \mid 2 \mid 1)_{\text{RNS}}$ and $(1 \mid 2 \mid 3 \mid 4 \mid 5)_{\text{RNS}}$ using mixed-radix conversion.
- **e.** Convert the numbers $(5 \mid 4 \mid 3 \mid 2 \mid 1)_{\text{RNS}}$ and $(1 \mid 2 \mid 3 \mid 4 \mid 5)_{\text{RNS}}$ to decimal.
- **f.** What is the representational efficiency of this RNS compared to standard binary?

4.2 RNS representation and arithmetic Consider the low-cost RNS system RNS(32 | 31 | 15 | 7) derived in Section 4.2.

- **a.** Represent the numbers $x = 168$ and $y = -23$ in this RNS.
- **b.** Compute $x + y, x - y, x \times y$, checking the results via decimal arithmetic.
- **c.** Knowing that x is a multiple of 7, divide it by 7 in the RNS.
- **d.** Compare the numbers $(4 \mid 3 \mid 2 \mid 1)_{\text{RNS}}$ and $(1 \mid 2 \mid 3 \mid 4)_{\text{RNS}}$ using mixed-radix conversion.
- **e.** Convert the numbers $(4 \mid 3 \mid 2 \mid 1)_{\text{RNS}}$ and $(1 \mid 2 \mid 3 \mid 4)_{\text{RNS}}$ to decimal.
- **f.** What is the representational efficiency of this RNS compared to standard binary?

4.3 RNS representation Find all numbers for which the RNS(8 | 7 | 5 | 3) representation is palindromic (i.e., the string of four "digits" reads the same forward and backward).

4.4 RNS versus GSD representation We are contemplating the use of 16-bit representations for fast integer arithmetic. One option, radix-8 GSD representation with the digit set $[-5, 4]$, can accommodate four-digit numbers. Another is RNS(16 | 15 | 13 | 11) with complement representation of negative values.

- **a.** Compute and compare the range of representable integers in the two systems.
- **b.** Represent the integers $+441$ and -228 and add them in the two systems.
- **c.** Briefly discuss and compare the complexity of multiplication in the two systems.

4.5 RNS representation and arithmetic Consider a residue number system that can be used to represent the equivalent of 24-bit, 2's-complement numbers.

- **a.** Select the set of moduli to maximize the speed of arithmetic operations.
- **b.** Determine the representational efficiency of the resulting RNS.
- **c.** Represent the numbers $x = +295$ and $y = -322$ in this number system.
- **d.** Compute the representations of $x + y, x - y$, and $x \times y$; check the results.

4.6 Binary-to-RNS conversion In a residue number system, 11 is used as one of the moduli.

- **a.** Design a mod-11 adder using two standard 4-bit binary adders and a few logic gates.
- **b.** Using the adder of part a and a 10-word lookup table, show how the mod-11 residue of an arbitrarily long binary number can be computed by a serial-in, parallel-out circuit.

c. Repeat part a, assuming the use of mod-11 pseudoresidues in [0, 15].

d. Outline the changes needed in the design of part b if the adder of part c is used.

4.7 Low-cost RNS Consider residue number systems with moduli of the form 2^{a_i} or $2^{a_i} - 1$.

a. Prove that $m_i = 2^{a_i} - 1$ and $m_j = 2^{a_j} - 1$ are relatively prime if and only if a_i and a_j are relatively prime.

b. Show that such a system wastes at most one bit relative to binary representation.

c. Determine an efficient set of moduli to represent the equivalent of 32-bit unsigned integers. Discuss your efficiency criteria.

4.8 Special RNS representations It has been suggested that moduli of the form $2^{a_i} + 1$ also offer speed advantages. Evaluate this claim by devising a suitable representation for the $(a_i + 1)$-bit residues and dealing with arithmetic operations on such residues. Then, determine an efficient set of moduli of the form 2^{a_i} and $2^{a_i} \pm 1$ to represent the equivalent of 32-bit integers.

4.9 Overflow in RNS arithmetic Show that if $0 \leq x, y < m$, then $(x + y) \bmod m$ causes overflow if and only if the result is less than x (thus the problem of overflow detection in RNS arithmetic is equivalent to the magnitude comparison problem).

4.10 Discrete logarithm Consider a prime modulus p. From number theory, we know that there always exists an integer generator g such that the powers $g^0, g^1, g^2, g^3, \ldots \pmod p$ produce all the integers in $[1, p - 1]$. If $g^i = x \bmod p$, then i is called the mod-p, base-g discrete logarithm of x. Outline a modular multiplication scheme using discrete log and $\log^{-1}$ tables and an adder.

4.11 Halving even numbers in RNS Given the representation of an even number in an RNS with only odd moduli, find an efficient algorithm for halving the given number.

4.12 Symmetric RNS In a symmetric RNS, the residues are signed integers, possessing the smallest possible absolute values, rather than unsigned integers. Thus, for an odd modulus m, symmetric residues range from $-(m - 1)/2$ to $(m - 1)/2$ instead of from 0 to $m - 1$. Discuss the possible advantages of a symmetric RNS over ordinary RNS.

4.13 Approximate Chinese remainder theorem decoding Consider the numbers $x = (0|6|3|0)_{\text{RNS}}$ and $y = (5|3|0|0)_{\text{RNS}}$ of Example 4.3 in Section 4.4.

a. Redo the example and its associated error analysis with table entries rounded to two decimal digits. How does the conclusion change?

b. Redo the example with table entries rounded to three decimal digits and discuss.

4.14 Division of RNS numbers by the moduli

a. Show how an RNS number can be divided by one of the moduli to find the quotient and the remainder, both in RNS form.

b. Repeat part a for division by the product of two or more moduli.

4.15 RNS base extension Consider a k-modulus RNS and the representation of a number x in that RNS. Develop an efficient algorithm for deriving the representation of x in a

$(k + 1)$-modulus RNS that includes all the moduli of the original RNS plus one more modulus that is relatively prime with respect to the preceding k. This process of finding a new residue given k existing residues is known as base extension.

4.16 Automorphic numbers An n-place *automorph* is an n-digit decimal number whose square ends in the same n digits. For example, 625 is a 3-place automorph, since $625^2 = 390\,625$.

a. Prove that $x > 1$ is an n-place automorph if and only if $x \bmod 5^n = 0$ or 1 and $x \bmod 2^n = 1$ or 0, respectively.

b. Relate n-place automorphs to a 2-residue RNS with $m_1 = 5^n$ and $m_0 = 2^n$.

c. Prove that if x is an n-place automorph, then $(3x^2 - 2x^3) \bmod 10^{2n}$ is a $2n$-place automorph.

REFERENCES

[Garn59] Garner, H. L., "The Residue Number System," *IRE Trans. Electronic Computers*, Vol. 8, pp. 140–147, June 1959.

[Jenk93] Jenkins, W. K., "Finite Arithmetic Concepts," in *Handbook for Digital Signal Processing*, S. K. Mitra and J. F. Kaiser, (eds.), Wiley, 1993, pp. 611–675.

[Hung94] Hung, C. Y., and B. Parhami, "An Approximate Sign Detection Method for Residue Numbers and Its Application to RNS Division," *Computers & Mathematics with Applications*, Vol. 27, No. 4, pp. 23–35, 1994.

[Hung95] Hung, C. Y., and B. Parhami, "Error Analysis of Approximate Chinese-Remainder-Theorem Decoding," *IEEE Trans. Computers*, Vol. 44, No. 11, pp. 1344–1348, 1995.

[Merr64] Merrill, R.D., "Improving Digital Computer Performance Using Residue Number Theory," *IEEE Trans. Electronic Computers*, Vol. 13, No. 2, pp. 93–101, April 1964.

[Parh76] Parhami, B., "Low-Cost Residue Number Systems for Computer Arithmetic," *AFIPS Conf. Proc.*, Vol. 45 (1976 National Computer Conference), AFIPS Press, 1976, pp. 951–956.

[Parh93] Parhami, B., and H.-F. Lai, "Alternate Memory Compression Schemes for Modular Multiplication," *IEEE Trans. Signal Processing*, Vol. 41, pp. 1378–1385, March 1993.

[Sode86] Soderstrand, M. A., W. K. Jenkins, G. A. Jullien, and F. J. Taylor (eds.), *Residue Number System Arithmetic*, IEEE Press, 1986.

[Szab67] Szabo, N. S., and R. I. Tanaka, *Residue Arithmetic and Its Applications to Computer Technology*, McGraw-Hill, 1967.

PART II ADDITION/ SUBTRACTION

Addition is the most common arithmetic operation and also serves as a building block for synthesizing all other operations. Within digital computers, addition is performed extensively both in explicitly specified computation steps and as a part of implicit ones dictated by indexing and other forms of address arithmetic. In simple ALUs that lack dedicated hardware for fast multiplication and division, these latter operations are performed as sequences of additions. A review of fast addition schemes is thus an apt starting point in investigating arithmetic algorithms. Subtraction is normally performed by negating the subtrahend and adding the result to the minuend. This is quite natural, given that an adder must handle signed numbers anyway. Even when implemented directly, a subtractor is quite similar to an adder. Thus, in the following four chapters that constitute this part, we focus almost exclusively on addition:

Chapter 5 BASIC ADDITION AND COUNTING

As stated in Section 3.1, propagation of carries is a major impediment to high-speed addition with fixed-radix positional number representations. Before exploring various ways of speeding up the carry propagation process, however, we need to examine simple ripple-carry adders, the building blocks used in their construction, the nature of the carry propagation process, and the special case of counting. Chapter topics include:

5.1 BIT-SERIAL AND RIPPLE-CARRY ADDERS

Single-bit half-adders and full adders are versatile building blocks that are used in synthesizing adders and many other types of arithmetic circuit. A half-adder (HA) receives two input bits x and y, producing a sum bit $s = x \oplus y = x\overline{y} + \overline{x}y$ and a carry bit $c = xy$. Figure 5.1 depicts three of the many possible logic realizations of a half-adder. A half-adder can be viewed as a single-bit binary adder that produces the 2-bit sum of its single-bit inputs, namely, $x + y = (c_{\text{out}}\ s)_{\text{two}}$, where the plus sign in this expression stands for arithmetic sum rather than logical OR.

A single-bit full adder (FA) is defined as follows:

Inputs:	Operand bits x, y and carry-in c_{in}	(or x_i, y_i, c_i for stage i)
Outputs:	Sum bit s and carry-out c_{out}	(or s_i and c_{i+1} for stage i)
	$s = x \oplus y \oplus c_{\text{in}}$	(odd parity function)
	$= xyc_{\text{in}} + \overline{x}\,\overline{y}c_{\text{in}} + \overline{x}y\overline{c}_{\text{in}} + x\overline{y}\,\overline{c}_{\text{in}}$	
	$c_{\text{out}} = xy + xc_{\text{in}} + yc_{in}$	(majority function)

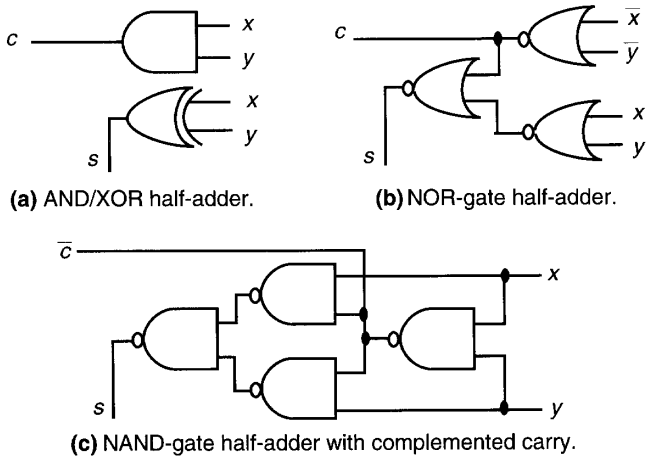


Fig. 5.1 Three implementations of a half-adder.

A full adder can be implemented by using two half-adders and an OR gate as shown in Fig. 5.2a. The OR gate in Fig. 5.2a can be replaced with a NAND gate if the two HAs are NAND-gate half-adders with complemented carry outputs. Alternatively, one can implement a full adder as two-level AND-OR/NAND-NAND circuits according to the preceding logic equations for s and c_{out} (Fig. 5.2b). Because of the importance of the full adder as an arithmetic building block, many optimized FA designs exist for a variety of implementation technologies. Figure 5.2c shows a full adder, built of seven inverters and two 4-to-1 multiplexers (Mux), that is suitable for CMOS transmission-gate logic implementation.

Full and half-adders can be used for realizing a variety of arithmetic functions. We will see many examples in this and the following chapters. For instance, a bit-serial adder can be built from a full adder and a carry flip-flop, as shown in Fig. 5.3a. The operands are supplied to the FA one bit per clock cycle, beginning with the least significant bit, from a pair of shift registers, and the sum is shifted into a result register. Addition of k-bit numbers can thus be completed in k clock cycles. A k-bit ripple-carry binary adder requires k full adders, with the carry-out of the ith FA connected to the carry-in input of the $(i + 1)$th FA. The resulting k-bit adder produces a k-bit sum output and a carry-out; alternatively, c_{out} can be viewed as the most significant bit of a $(k + 1)$-bit sum. Figure 5.3b shows a ripple-carry adder for 4-bit operands, producing a 4-bit or 5-bit sum.

The ripple-carry adder shown in Fig. 5.3b leads directly to a CMOS implementation with transmission gate logic using the full adder design of Fig. 5.2c. A possible layout is depicted in Fig. 5.4, which also shows the approximate area requirements for the 4-bit ripple-carry adder in units of λ (half the minimum feature size). For details of this particular design, refer to [Puck94, pp. 213-223].

The latency of a k-bit ripple-carry adder can be derived by considering the worst-case signal propagation path. As shown in Fig. 5.5, the critical path usually begins at the x_0 or y_0 input, proceeds through the carry-propagation chain to the leftmost FA, and terminates at the s_{k-1} output. Of course, it is possible that for some FA implementations, the critical path

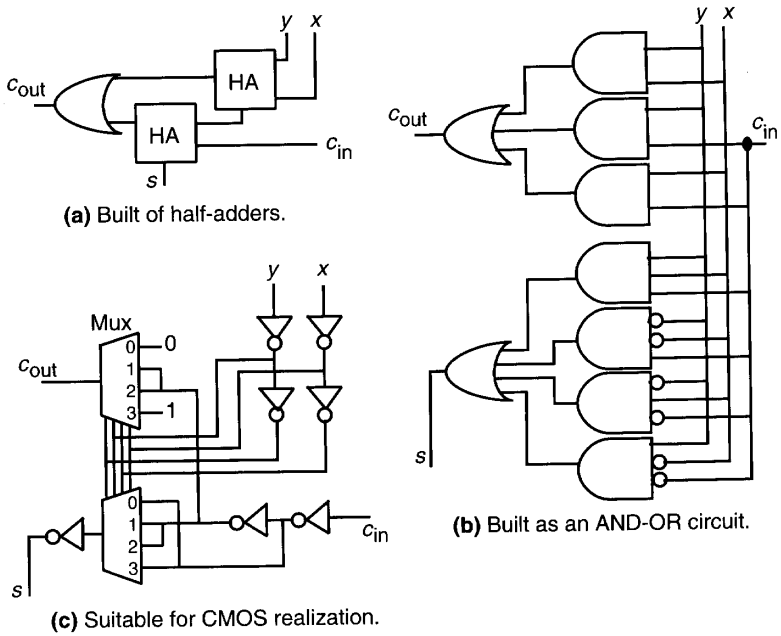


Fig. 5.2 Possible designs for a full adder in terms of half-adders, logic gates, and CMOS transmission gates.

might begin at c_0 and/or terminate at c_k. However, given that the delay from carry-in to carry-out is more important than from x to carry-out or from carry-in to s, full-adder designs often minimize the delay from carry-in to carry-out, making the path shown in Fig. 5.5 the one with the largest delay. We can thus write the following expression for the latency of a k-bit ripple-carry adder:

$$T_{\text{ripple-add}} = T_{\text{FA}}(x, y \rightarrow c_{\text{out}}) + (k-2) \times T_{\text{FA}}(c_{\text{in}} \rightarrow c_{\text{out}}) + T_{\text{FA}}(c_{\text{in}} \rightarrow s)$$

where $T_{\text{FA}}(\text{input} \rightarrow \text{output})$ represents the latency of a full adder on the path between its specified input and output. As an approximation to the foregoing, we can say that the latency of a ripple-carry adder is kT_{FA}.

We see that the latency grows linearly with k, making the ripple-carry design undesirable for large k or for high-performance arithmetic units. Note that the latency of a bit-serial adder is also $O(k)$, although the constant of proportionality is larger here because of the latching and clocking overheads.

Full and half-adders, as well as multibit binary adders, are powerful building blocks that can also be used in realizing nonarithmetic functions if the need arises. For example, a 4-bit binary adder with c_{in}, two 4-bit operand inputs, c_{out}, and a 4-bit sum output can be used to synthesize the four-variable logic function $w + xyz$ and its complement, as depicted and justified in Fig. 5.6. The logic expressions written next to the arrows in Fig. 5.6 represent the carries between various stages. Note, however, that the 4-bit adder need not be implemented as a ripple-carry adder for the results at the outputs to be valid.

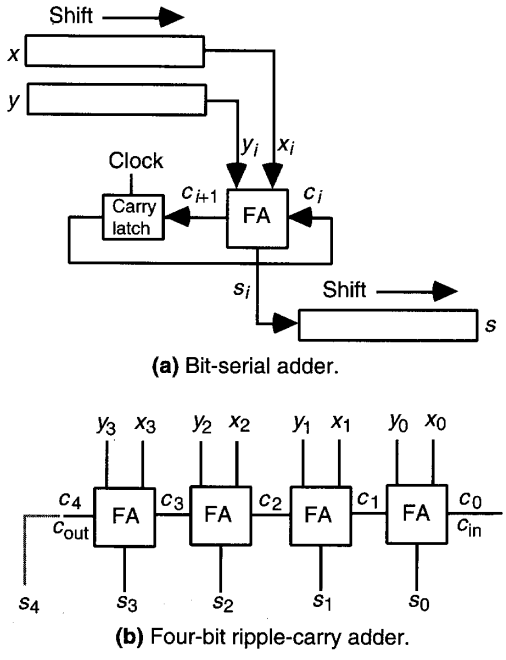


Fig. 5.3 Using full adders in building bit-serial and ripple-carry adders.

5.2 CONDITIONS AND EXCEPTIONS

When a k-bit adder is used in an ALU, it is customary to provide the k-bit sum along with information about the following outcomes, which are associated with flag bits within a condition/exception register:

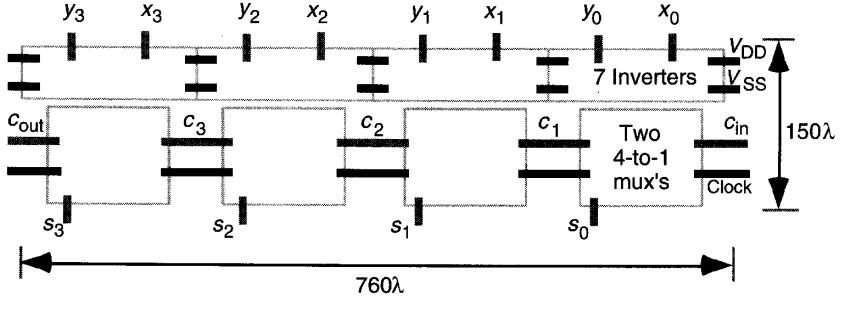


Fig. 5.4 Layout of a 4-bit ripple-carry adder in CMOS implementation [Puck94].

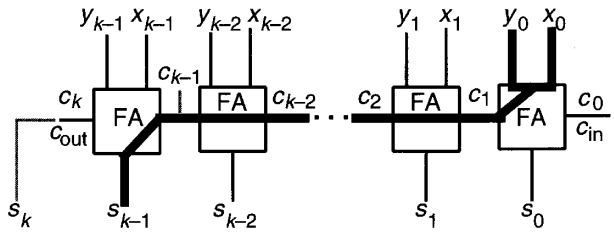


Fig. 5.5 Critical path in a k-bit ripple-carry adder.

c_{out}	Indicating that a carry-out of 1 is produced
Overflow	Indicating that the output is not the correct sum
Negative	Indicating that the addition result is negative
Zero	Indicating that the addition result is zero

When we are adding unsigned numbers, c_{out} and "overflow" are one and the same, and the "sign" condition is obviously irrelevant. For 2's-complement addition, overflow occurs when two numbers of like sign are added and a result of the opposite sign is produced. Thus:

$$\text{Overflow}_{2\text{'s-compl}} = x_{k-1}y_{k-1}\bar{s}_{k-1} + \bar{x}_{k-1}\bar{y}_{k-1}s_{k-1}$$

It is fairly easy to show that overflow in 2's-complement addition can be detected from the leftmost two carries as follows:

$$\text{Overflow}_{2\text{'s-compl}} = c_k \oplus c_{k-1} = c_k\bar{c}_{k-1} + \bar{c}_k c_{k-1}$$

In 2's-complement addition, c_{out} has no significance. However, since a single adder is frequently used to add both unsigned and 2's-complement numbers, c_{out} is useful as well. Figure 5.7 shows a ripple-carry implementation of an unsigned or 2's-complement adder with auxiliary outputs for conditions and exceptions. Because of the large number of inputs into the NOR gate that tests for zero, it must be implemented as an OR tree followed by an inverter.

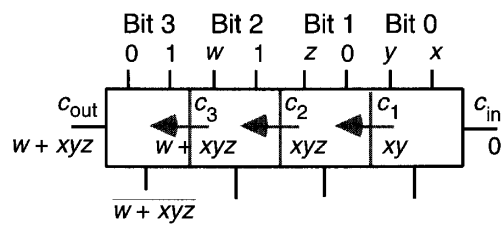


Fig. 5.6 Four-bit binary adder used to realize the logic function $f = w + xyz$ and its complement.

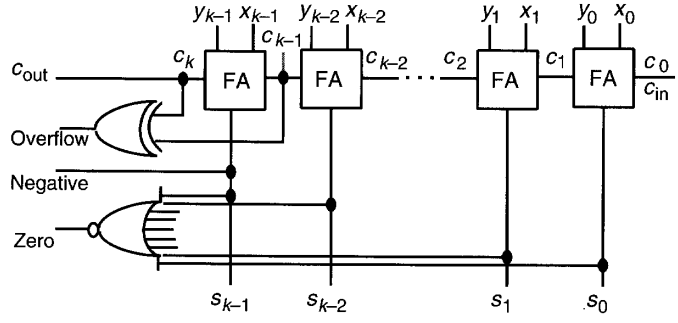


Fig. 5.7 Two's-complement adder with provisions for detecting conditions and exceptions.

5.3 ANALYSIS OF CARRY PROPAGATION

Various ways of dealing with the carry problem were enumerated in Section 3.1. Some of the methods already discussed include limiting the propagation of carries (hybrid signed-digit, RNS) or eliminating carry propagation altogether (GSD). The latter approach, when used for adding a set of numbers in carry-save form, can be viewed as a way of amortizing the propagation delay of the final conversion step over many additions, thus making the per-add contribution of the carry propagation delay quite small. What remains to be discussed, in this and the following chapter, is how one can speed up a single addition operation involving conventional (binary) operands.

We begin by analyzing how and to what extent carries propagate in adding two binary numbers. Consider the example addition of 16-bit binary numbers depicted in Fig. 5.8, where the carry chains of length 2, 3, 6, and 4 are shown. The length of a carry chain is the number of digit positions from where the carry is generated up to and including where it is finally absorbed or annihilated. A carry chain of length 0 thus means "no carry production," and a chain of length 1 means that the carry is absorbed in the next position. We are interested in the length of the longest propagation chain (6 in Fig. 5.8), which dictates the adder's latency.

Given binary numbers with random bit values, for each position i we have:

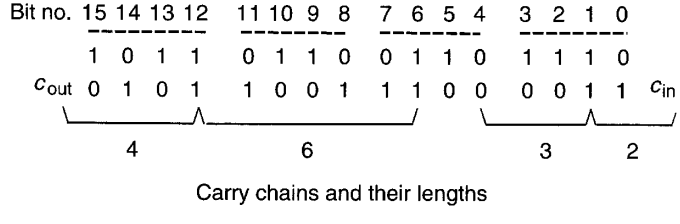


Fig. 5.8 Example addition and its carry-propagation chains.

Probability of carry generation = 1/4
Probability of carry annihilation = 1/4
Probability of carry propagation = 1/2

The probability that a carry generated at position i will propagate up to and including position $j-1$ and stop at position $j (j > i)$ is $2^{-(j-1-i)} \times 1/2 = 2^{-(j-i)}$. The expected length of the carry chain that starts at bit position i is, therefore, given by

$$\sum_{j=i+1}^{k-1} (j-i)2^{-(j-i)} + (k-i)2^{-(k-1-i)} = \sum_{l=1}^{k-1-i} l2^{-l} + (k-i)2^{-(k-1-i)}$$

$$= 2 - (k-i+1)\, 2^{-(k-1-i)} + (k-i)^{-(k-1-i)} = 2 - 2^{-(k-i-1)}$$

where the simplification is based on the identity $\sum_{l=1}^{p} l2^{-l} = 2 - (p+2)2^{-p}$. In the preceding derivation, the term $(k-i)\, 2^{-(k-1-i)}$ is added to the summation because carry definitely stops at position k; so we do not multiply the term $2^{-(k-1-i)}$ by 1/2, as was done for the terms within the summation.

The preceding result indicates that for $i << k$, the expected length of the carry chain that starts at position i is approximately 2. Note that the formula checks out for the extreme case of $i = k-1$, since in this case, the exact carry chain length, and thus its expected value, is 1. We conclude that carry chains are usually quite short.

On the average, the longest carry chain in adding k-bit numbers is of length $\log_2 k$. This was first observed and proved by Burks, Goldstine, and von Neumann in their classic report defining the structure of a stored-program computer [Burk46]. An interesting analysis based on Kolmogorov complexity theory has been offered in [Beig98]. The latter paper also cites past attempts at providing alternate or more complete proofs of the proposition.

Here is one way to prove the logarithmic average length of the worst-case carry chain. Let $\eta_k(h)$ be the probability that the longest carry chain in a k-bit addition is of length h or more. Clearly, the probability of the longest carry chain being of length exactly h is $\eta_k(h) - \eta_k(h+1)$. We can use a recursive formulation to find $\eta_k(h)$. The longest carry chain can be of length h or more in two mutually exclusive ways:

a. The least significant $k-1$ bits have a carry chain of length h or more.
b. The least significant $k-1$ bits do not have such a carry chain, but the most significant h bits, including the last bit, have a chain of the exact length h.

Thus, we have

$$\eta_k(h) \leq \eta_{k-1}(h) + 2^{-(h+1)}$$

where $2^{-(h+1)}$ is the product of 1/4 (representing the probability of carry generation) and $2^{-(h-1)}$ (probability that carry propagates across $h-2$ intermediate positions and stops in the last one). The inequality occurs because the second term is not multiplied by a probability as discussed above. Hence, assuming $\eta_i(h) = 0$ for $i < h$:

$$\eta_k(h) = \sum_{i=h}^{k} [\eta_i(h) - \eta_{i-1}(h)] \leq (k-h+1)\, 2^{-(h+1)} \leq 2^{-(h+1)}k$$

To complete our derivation of the expected length λ of the longest carry chain, we note that:

$$\begin{aligned}\lambda &= \sum_{h=1}^{k} h[\eta_k(h) - \eta_k(h+1)] \\ &= [\eta_k(1) - \eta_k(2)] + 2[\eta_k(2) - \eta_k(3)] + \cdots + k[\eta_k(k) - 0] \\ &= \sum_{h=1}^{k} \eta_k(h)\end{aligned}$$

We next break the final summation above into two parts: the first $\gamma = \lfloor \log_2 k \rfloor - 1$ terms and the remaining $k - \gamma$ terms. Using the upper bound 1 for the first part and $2^{-(h+1)}k$ for the second part, we get:

$$\lambda = \sum_{h=1}^{k} \eta_k(h) \leq \sum_{h=1}^{\gamma} 1 + \sum_{h=\gamma+1}^{k} 2^{-(h+1)}k < \gamma + 2^{-(\gamma+1)}k$$

Now let $\varepsilon = \log_2 k - \lfloor \log_2 k \rfloor$ or $\gamma = \log_2 k - 1 - \varepsilon$, where $0 \leq \varepsilon < 1$. Then, substituting the latter expression for γ in the preceding inequality and noting that $2^{\log_2 k} = k$ and $2^{\varepsilon} < 1 + \varepsilon$, we get:

$$\lambda < \log_2 k - 1 - \varepsilon + 2^{\varepsilon} < \log_2 k$$

This concludes our derivation of the result that the expected length of the worst-case carry chain in a k-bit addition with random operands is upper-bounded by $\log_2 k$. Experimental results verify the $\log_2 k$ approximation to the length of the worst-case carry chain and suggest that $\log_2(1.25k)$ is a better estimate [Hend61].

5.4 CARRY COMPLETION DETECTION

A ripple-carry adder is the simplest and slowest adder design. For k-bit operands, both the worst-case delay and the implementation cost of a ripple-carry adder are linear in k. However, based on the analysis in Section 5.3, the worst-case carry-propagation chain of length k almost never materializes.

A carry completion detection adder takes advantage of the $\log_2 k$ average length of the longest carry chain to add two k-bit binary numbers in O(log k) time on the average. It is essentially a ripple-carry adder in which a carry of 0 is also explicitly represented and allowed to propagate between stages. The carry into stage i is represented by the two-rail code:

$$\begin{aligned}(b_i, c_i) = (0, 0) &\quad \text{Carry not yet known} \\ (0, 1) &\quad \text{Carry known to be 1} \\ (1, 0) &\quad \text{Carry known to be 0}\end{aligned}$$

Thus, just as two 1s in the operands generate a carry of 1 that propagates to the left, two 0s would produce a carry of 0. Initially, all carries are (0, 0) or unknown. After initialization, a bit position with $x_i = y_i$ makes the no-carry/carry determination and injects the appropriate carry $(b_{i+1}, c_{i+1}) = (\overline{x_i + y_i}, x_i y_i)$ into the carry propagation chain of Fig. 5.9 via the OR gates. The

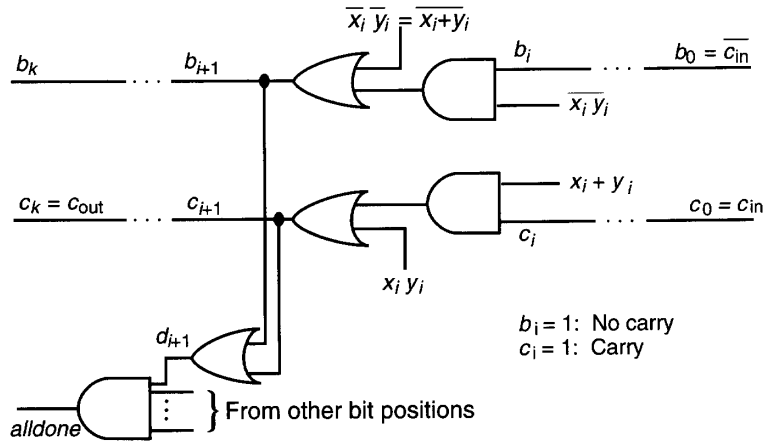


Fig. 5.9 The carry network of an adder with two-rail carries and carry completion detection logic.

carry ($\overline{c}_{in}$, c_{in}) is injected at the right end. When every carry has assumed one of the values (0, 1) or (1, 0), carry propagation is complete. The local "done" signals $d_i = b_i + c_i$ are combined by a global AND function into *alldone*, which indicates the end of carry propagation.

In designing carry completion adders, care must be taken to avoid hazards that might lead to a spurious *alldone* signal. Initialization of all carries to 0 through clearing of input bits and simultaneous application of all input data is one way of ensuring hazard-free operation.

Excluding the initialization and carry completion detection times, which must be considered and are the same in all cases, the latency of a k-bit carry completion adder ranges from 1 gate delay in the best case (no carry propagation at all: i.e., when adding a number to itself) to $2k + 1$ gate delays in the worst case (full carry propagation from c_{in} to c_{out}), with the average latency being about $2\log_2 k + 1$ gate delays. Note that once the final carries have arrived in all bit positions, the derivation of the sum bits is overlapped with completion detection and is thus not accounted for in the preceding latencies.

Because the latency of the carry completion adder is data dependent, the design of Fig. 5.9 is suitable for use in asynchronous systems. Most modern computers, however, use synchronous logic and thus cannot take full advantage of the high average speed of a carry completion adder.

5.5 ADDITION OF A CONSTANT: COUNTERS

When one input of the addition operation is a constant number, the design can be simplified or optimized compared to that of a general two-operand adder. With binary arithmetic, we can assume that the constant y to be added to x is odd, since in the addition $s = x + y_{even} = x + (y_{odd} \times 2^h)$, one can ignore the h rightmost bits in x and add y_{odd} to the remaining bits. The special case of $y = 1$ corresponds to standard counters, while $y = \pm 1$ yields an up/down counter.

Let the constant to be added to $x = (x_{k-1} \cdots x_2 x_1 x_0)_{two}$ be $y = (y_{k-1} \cdots y_2 y_1\ 1)_{two}$. The least significant bit of the sum is $\overline{x}_0$. The remaining bits of s can be determined by a $(k-1)$-bit

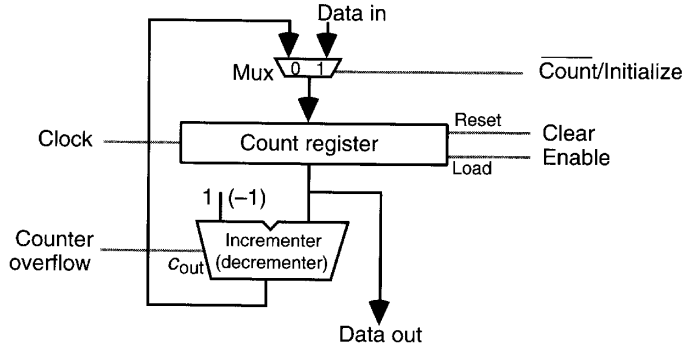


Fig. 5.10 An up (down) counter built of a register, an incrementer (decrementer), and a multiplexer.

ripple-carry adder, with $c_{in} = x_0$, each of its cells being a half-adder ($y_i = 0$) or a modified half-adder ($y_i = 1$). The fast adder designs to be covered later can similarly be optimized to take advantage of the known bits of y.

When $y = 1(-1)$, the resulting circuit is known as an *incrementer* (*decrementer*) and is used in the design of up (down) counters. Figure 5.10 depicts an up counter, with parallel load capability, built of a register, an incrementer, and a multiplexer. The design shown in Fig. 5.10 can be easily converted to an up/down counter by using an incrementer/decrementer and an extra control signal. Supplying the details is left as an exercise.

Many designs for fast counters are available [Ober81]. Conventional synchronous designs are based on full carry propagation in each increment/decrement cycle, thus limiting the counter's operating speed. In some cases, special features of the storage elements used can lead to simplifications. Figure 5.11 depicts an asynchronous counter built of cascaded negative-edge-triggered T (toggle) flip-flops. Each input pulse toggles the flip-flop at the least significant position, each 1-to-0 transition of the LSB flip-flop toggles the next flip-flop, and so on. The next input pulse can be accepted before the carry has propagated all the way to the left.

Certain applications require high-speed counting, with the count potentially becoming quite large. In such cases, a high-speed incrementer must be utilized. Methods used in the design of fast adders (Chapters 6 and 7) can all be adapted for building fast incrementers. When even the highest-speed incrementer cannot keep up with the input rate or when cost considerations preclude the use of an ultrafast incrementer, the frequency of the input can be reduced by applying it to a prescaler. The lower-frequency output of the prescaler can then be counted with less stringent speed requirements. In the latter case, the resulting count will be approximate.

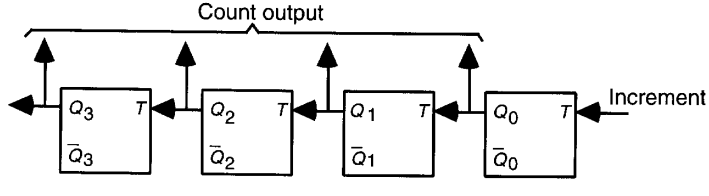


Fig. 5.11 Four-bit asynchronous up counter built only of negative-edge-triggered T flip-flops.

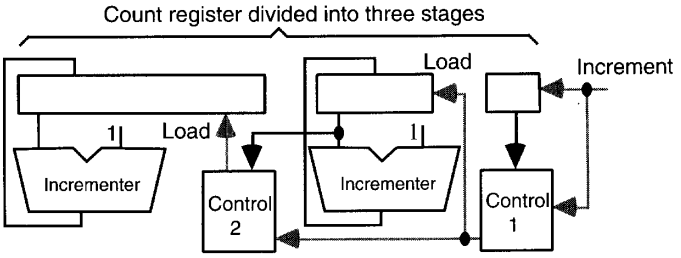


Fig. 5.12 Fast three-stage up counter.

Obviously, the count value can be represented in redundant format, allowing carry-free increment or decrement in constant time [Parh87]. However, with a redundant format, reading out the stored count involves some delay to allow for conversion of the internal representation to standard binary. Alternatively, one can design the counter as a cascade that begins with a very short, and thus fast, counter and continues with increasingly longer counters [Vuil91]. The longer counters on the left are incremented only occasionally and thus need not be very fast (their incremented counts can be precomputed by a slow incrementer and then simply loaded into the register when required). Figure 5.12 shows this principle applied to the design of a three-stage counter.

5.6 MANCHESTER CARRY CHAINS AND ADDERS

In the next three chapters, we will examine methods for speeding up the addition process for two operands (Chapters 6 and 7) and for multiple operands (Chapter 8). For two operands, the key to fast addition is a low-latency carry network, since once the carry into position i is known, the sum digit can be determined from the operand digits x_i and y_i and the incoming carry c_i in constant time through modular addtition:

$$s_i = (x_i + y_i + c_i) \bmod r$$

In the special case of radix 2, the relation above reduces to:

$$s_i = x_i \oplus y_i \oplus c_i$$

So, the primary problem in the design of two-operand adders is the computation of the k carries c_{i+1} based on the $2k$ operand digits x_i and y_i, $0 \leq i < k$.

From the point of view of carry propagation and the design of a carry network, the actual operand digits are not important. What matters is whether in a given position a carry is generated, propagated, or annihilated (absorbed). In the case of binary addition, the *generate*, *propagate*, and *annihilate* (*absorb*) signals are characterized by the following logic equations:

$$g_i = x_i y_i$$
$$p_i = x_i \oplus y_i$$
$$a_i = \overline{x}_i \overline{y}_i = \overline{x_i + y_i}$$

It is also helpful to define a *transfer* signal corresponding to the event that the carry-out will be 1, given that the carry-in is 1:

$$t_i = g_i + p_i = \overline{a}_i = x_i + y_i$$

More generally, for radix r, we have:

$$\begin{aligned} g_i &= 1 \text{ iff } x_i + y_i \geq r \\ p_i &= 1 \text{ iff } x_i + y_i = r - 1 \\ a_i &= 1 \text{ iff } x_i + y_i < r - 1 \end{aligned}$$

Thus, assuming that the signals above are produced and made available, the rest of the carry network design can be based on them and becomes completely independent of the operands or even the number representation radix.

Using the preceding signals, the *carry recurrence* can be written as follows:

$$c_{i+1} = g_i + c_i p_i$$

The carry recurrence essentially states that a carry will enter stage $i + 1$ if it is generated in stage i or it enters stage i and is propagated by that stage. Since

$$\begin{aligned} c_{i+1} &= g_i + c_i p_i = g_i + c_i g_i + c_i p_i \\ &= g_i + c_i(g_i + p_i) = g_i + c_i t_i \end{aligned}$$

the carry recurrence can be written in terms of t_i instead of p_i. This latter version of the carry recurrence leads to slightly faster adders because in binary addition, t_i is easier to produce than p_i (OR instead of XOR).

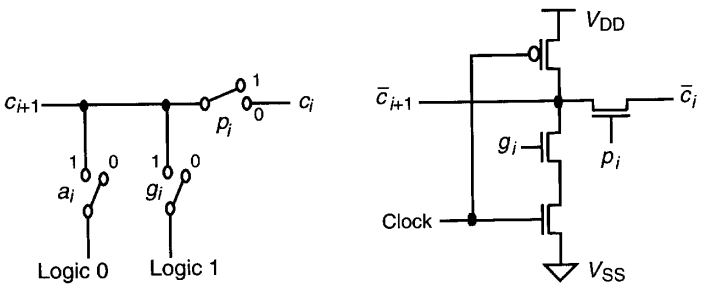
In what follows, we always deal with the carry recurrence in its original form $c_{i+1} = g_i + c_i p_i$, since it is more intuitive, but we keep in mind that in most cases, p_i can be replaced by t_i if desired.

The carry recurrence forms the basis of a simple carry network known as *Manchester carry chain*. A *Manchester adder* is one that uses a Manchester carry chain as its carry network. Each stage of a Manchester carry chain can be viewed as consisting of three switches controlled by the signals p_i, g_i, and a_i, so that the switch closes (conducts electricity) when the corresponding control signal is 1. As shown in Fig. 5.13a, the carry-out signal c_{i+1} is connected to 0 if $a_i = 1$, to 1 if $g_i = 1$, and to c_i if $p_i = 1$, thus assuming the correct logical value $c_{i+1} = g_i + c_i p_i$. Note that one, and only one, of the signals p_i, g_i, and a_i is 1.

Figure 5.13b shows how a Manchester carry chain might be implemented in CMOS. When the clock is low, the c nodes precharge. Then, when the clock goes high, if g_i is high, c_{i+1} is asserted or drawn low. To prevent g_i from affecting c_i, the signal p_i must be computed as the XOR (rather than OR) of x_i and y_i. This is not a problem because we need the XOR of x_i and y_i for computing the sum anyway.

For a k-bit Manchester carry chain, the total delay consists of three components:

1. The time to form the switch control signals.
2. The setup time for the switches.
3. Signal propagation delay through k switches in the worst case.



(a) Conceptual representation. **(b)** Possible CMOS realization.

Fig. 5.13 One stage in a Manchester carry chain.

The first two components of delay are small, constant terms. The delay is thus dominated by the third component, which is at best linear in k. For modern CMOS technology, the delay is roughly proportional to k^2 (as k pass transistors are connected in series), making the method undesirable for direct realization of fast adders. However, when the delay is in fact linear in k, speed is gained over gate-based ripple-carry adders because we have one switch delay rather than two gate delays per stage. The linear or superlinear delay of a Manchester carry chain limits its usefulness for wide words or in high-performance designs. Its main application is in implementing short chains (say, up to 8 bits) as building blocks for use with a variety of fast addition schemes.

PROBLEMS

5.1 Bit-serial 2's-complement adder Present the complete design of a bit-serial 2's-complement adder for 32-bit numbers. Include in your design the control details and provisions for overflow detection.

5.2 Four-function ALU Extend the design of Fig. 5.2c into a bit-slice for a four-function ALU that produces any of the following functions of the inputs x and y based on the values of two control signals: Sum, OR, AND, XOR. *Hint:* What happens if c_{in} is forced to 0 or 1?

5.3 Subtractive adder for 1's-complement numbers Show that the alternate representation of 0 in 1's complement, which is obtained only when x and $-x$ are added, can be avoided by using a "subtractive adder" that always complements y and performs subtraction to compute $x + y$.

5.4 Digit-serial adders

a. A radix-2^g digit-serial adder can be faster than a bit-serial adder. Show the detailed design of a radix-16 digit-serial adder for 32-bit unsigned numbers and compare it with respect to latency and cost to bit-serial and ripple-carry binary adders.

b. Design a digit-serial BCD (binary-coded decimal) adder to add decimal numbers whose digits are encoded as 4-bit binary numbers.

c. Combine the designs of parts a and b into an adder than can act as radix-16 or BCD adder according to the value of a control signal.

5.5 **Binary adders as versatile building blocks** A 4-bit binary adder can be used to implement many logic functions besides its intended function. An example appears in Fig. 5.6. Show how a 4-bit binary adder can be used to realize the following:

a. A 3-bit adder, with carry-in and carry-out.

b. Two independent single-bit full adders.

c. A single-bit full adder and a 2-bit binary adder operating independently.

d. A 4-bit odd parity generator (4-bit XOR).

e. A 4-bit even or odd parity generator under the control of an even/$\overline{\text{odd}}$ signal.

f. Two independent 3-bit odd parity generators.

g. A five-input AND circuit.

h. A five-input OR circuit.

i. A circuit to realize the four-variable logic function $wx + yz$.

j. A circuit to realize the four-variable logic function $wx\bar{y} + wx\bar{z} + \bar{w}yz + \bar{x}yz$.

k. A multiply-by-15 circuit for a 2-bit number x_1x_0, resulting in a 6-bit product.

l. A circuit to compute $x + 4y + 8z$, where x, y, and z are 3-bit unsigned numbers.

m. A five-input "parallel counter" producing the sum $s_2s_1s_0$ of five 1-bit numbers.

5.6 **Binary adders as versatile building blocks** Show how an 8-bit binary adder can be used to realize the following:

a. Three independent 2-bit binary adders, each with carry-in and carry-out.

b. A circuit to realize the six-variable logic function $uv + wx + yz$.

c. A circuit to compute $2w + x$ and $2y + z$, where w, x, y, z are 3-bit numbers.

d. A multiply-by-85 circuit for a number $x_3x_2x_1x_0$, resulting in an 11-bit product.

e. A circuit to compute the 5-bit sum of three 3-bit unsigned numbers.

f. A seven-input "parallel counter" producing the sum $s_2s_1s_0$ of seven 1-bit numbers.

5.7 **Decimal addition** Many microprocessors provide an 8-bit unsigned "add with carry" instruction that is defined as unsigned addition using the "carry flag" as c_{in} and producing two carries: carry-out or c_8, stored in the carry flag, and "middle carry" or c_4, stored in a special flag bit for subsequent use (e.g., as branch condition). Show how the "add with carry" instruction can be used to construct a routine for adding unsigned decimal numbers that are stored in memory with two BCD (binary-coded decimal) digits per byte.

5.8 **Two's-complement adder**

a. Prove that in adding k-bit 2's-complement numbers, overflow occurs if and only if $c_{k-1} \neq c_k$.

b. Show that in a 2's-complement adder that does not provide c_{out}, we can produce it externally using $c_{\text{out}} = x_{k-1}y_{k-1} + \bar{s}_{k-1}(x_{k-1} + y_{k-1})$.

5.9 Carry completion adder The computation of a k-input logic function requires O(log k) time if gates with constant fan-in are used. Thus, the AND gate in Fig. 5.9 that generates the *alldone* signal is really a tree of smaller AND gates that implies O(log k) delay. Wouldn't this imply that the addition time of the carry completion adder is O(log^2 k) rather than O(log k)?

5.10 Carry completion adder

a. Design the sum logic for the carry completion adder of Fig. 5.9.

b. Design a carry completion adder using full and half-adders plus inverters as the only building blocks (besides the completion detection logic).

c. Repeat part a if the sum bits are to be obtained with two-rail (z, p) encoding whereby 0 and 1 are represented by (1, 0) and (0, 1), respectively. In this way, the sum bits are independently produced as soon as possible, allowing them to be processed by other circuits in an asynchronous fashion.

5.11 Balanced ternary adder Consider the balanced ternary number system with $r = 3$ and digit set $[-1, 1]$. Addition of such numbers involves carries in $\{-1, 0, 1\}$. Assuming that both the digit set and carries are represented using the (n, p) encoding of Fig. 3.7:

a. Design a ripple-carry adder cell for balanced ternary numbers.

b. Convert the adder cell of part a to an adder/subtractor with a control input.

c. Design and analyze a carry completion sensing adder for balanced ternary numbers.

5.12 Synchronous binary counter Design a synchronous counterpart for the asynchronous counter shown in Fig. 5.11.

5.13 Negabinary up/down counter Design an up/down counter based on the negabinary (radix -2) number representation in the count register. *Hint:* Consider the negabinary representation as a radix-4 number system with the digit set $[-2, 1]$.

5.14 Design of fast counters Design the two control circuits in Fig. 5.12 and determine optimal lengths for the three counter segments, as well as the overall counting latency (clock period), in each of the following cases. Assume the use of ripple-carry incrementers.

a. An overall counter length of 48 bits.

b. An overall counter length of 80 bits.

5.15 Fast up/down counters Extend the fast counter design of Fig. 5.12 to an up/down counter. *Hint:* Incorporate the sign logic in "Control 1," use a fast 0 detection mechanism, and save the old value when incrementing a counter stage.

5.16 Manchester carry chains Study the effects of inserting a pair of inverters after every g stages in a CMOS Manchester carry chain (Fig. 5.13b). In particular, discuss whether the carry propagation time can be made linear in k by suitable placement of the inverter pairs.

5.17 Analysis of carry propagation In deriving the average length of the worst-case carry propagation chain, we made substitutions and simplifications that led to the upper bound

$\log_2 k$. By deriving an O(log k) lower bound, show that the exact average is fairly close to this upper bound.

REFERENCES

[Beig98] Beigel, R., B. Gasarch, M. Li, and L. Zhang, "Addition in $\log_2 n + O(1)$ Steps on Average: A Simple Analysis," *Theoretical Computer Science*, Vol. 191, Nos. 1–2, pp. 245–248, January 1998.

[Burk46] Burks, A. W., H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Institute for Advanced Study, Princeton, NJ, 1946.

[Gilc55] Gilchrist, B., J. H. Pomerene, and S. Y. Wong, "Fast Carry Logic for Digital Computers," *IRE Trans. Electronic Computers*, Vol. 4, pp. 133–136, 1955.

[Hend61] Hendrickson, H. C., "Fast High-Accuracy Binary Parallel Addition," *IRE Trans. Electronic Computers*, Vol. 10, pp. 465–468, 1961.

[Kilb60] Kilburn, T., D. B. G. Edwards, and D. Aspinall, "A Parallel Arithmetic Unit Using a Saturated Transistor Fast-Carry Circuit," *Proc. IEE*, Vol. 107B, pp. 573–584, 1960.

[Ober81] Oberman, R. M. M., *Counting and Counters*, Macmillan, London, 1981.

[Parh87] Parhami, B., "Systolic Up/Down Counters with Zero and Sign Detection," *Proc. Symp. Computer Arithmetic*, Como, Italy, May 1987, pp. 174–178.

[Puck94] Pucknell, D. A., and K. Eshraghian, *Basic VLSI Design*, 3rd ed., Prentice-Hall, 1994.

[Vuil91] Vuillemin, J. E., "Constant Time Arbitrary Length Synchronous Binary Counters," *Proc. Symp. Computer Arithmetic*, Grenoble, France, June 1991, pp. 180–183.

Chapter 6 CARRY-LOOKAHEAD ADDERS

Adder designs considered in Chapter 5 have worst-case delays that grow at least linearly with the word width k. Since the most significant bit of the sum is a function of all the $2k$ input bits, given that the gate fan-in is limited to d, a lower bound on addition latency is $\log_d(2k)$. An interesting question, therefore, is whether one can add two k-bit binary numbers in O(log k) worst-case time. Carry-lookahead adders, covered in this chapter, represent a commonly used scheme for logarithmic time addition. Other schemes are introduced in Chapter 7.

6.1 UNROLLING THE CARRY RECURRENCE

Recall the g_i (generate), p_i (propagate), a_i (annihilate or absorb), and t_i (transfer) auxiliary signals introduced in Section 5.6:

$g_i = 1$ iff $x_i + y_i \geq r$	Carry is generated
$p_i = 1$ iff $x_i + y_i = r - 1$	Carry is propagated
$t_i = \overline{a}_i = g_i + p_i$	Carry is not annihilated

These signals, along with the carry recurrence

$$c_{i+1} = g_i + p_i c_i = g_i + t_i c_i$$

allow us to decouple the problem of designing a fast carry network from details of the number system (radix, digit set). In fact it does not even matter whether we are adding or subtracting;

any carry network can be used as a borrow network if we simply redefine the preceding signals to correspond to borrow generation, borrow propagation, and so on.

The carry recurrence $c_{i+1} = g_i + p_i c_i$ states that a carry will enter stage $i + 1$ if it is generated in stage i or it enters stage i and is propagated by that stage. One can easily unroll this recurrence, eventually obtaining each carry c_i as a logical function of the operand bits and c_{in}. Here are three steps of the unrolling process for c_i:

$$\begin{aligned} c_i &= g_{i-1} + c_{i-1}p_{i-1} \\ &= g_{i-1} + (g_{i-2} + c_{i-2}p_{i-2})p_{i-1} = g_{i-1} + g_{i-2}p_{i-1} + c_{i-2}p_{i-2}p_{i-1} \\ &= g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + c_{i-3}p_{i-3}p_{i-2}p_{i-1} \\ &= g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + g_{i-4}p_{i-3}p_{i-2}p_{i-1} + c_{i-4}p_{i-4}p_{i-3}p_{i-2}p_{i-1} \end{aligned}$$

The unrolling can be continued until the last product term contains $c_0 = c_{in}$. The unrolled version of the carry recurrence has the following simple interpretation: carry enters into position i if and only if a carry is generated in position $i - 1 (g_{i-1})$, or a carry generated in position $i - 2$ is propagated by position $i - 1 (g_{i-2}p_{i-1})$, or a carry generated in position $i - 3$ is propagated at $i - 2$ and $i - 1 (g_{i-3}p_{i-2}p_{i-1})$, etc.

After full unrolling, we can compute all the carries in a k-bit adder directly from the auxiliary signals (g_i, p_i) and c_{in}, using two-level AND-OR logic circuits with maximum gate fan-in of $k + 1$. For $k = 4$, the logic expressions are as follows:

$$\begin{aligned} c_4 &= g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3 \\ c_3 &= g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2 \\ c_2 &= g_1 + g_0p_1 + c_0p_0p_1 \\ c_1 &= g_0 + c_0p_0 \end{aligned}$$

Here, c_0 and c_4 are the 4-bit adder's c_{in} and c_{out}, respectively. A carry network based on the preceding equations can be used in conjunction with 2-input ANDs, producing the g_i signals, and 2-input XORs, producing the p_i and sum bits, to build a 4-bit binary adder. Such an adder is said to have *full carry lookahead*.

Note that since c_4 does not affect the computation of the sum bits, it can be derived based on the simpler equation

$$c_4 = g_3 + c_3p_3$$

with little or no speed penalty. The resulting carry network is depicted in Fig. 6.1.

Clearly, full carry lookahead is impractical for wide words. The fully unrolled carry equation for c_{31}, for example, consists of 32 product terms, the largest of which contains 32 literals. Thus, the required AND and OR functions must be realized by tree networks, leading to increased latency and cost. Two schemes for managing this complexity immediately suggest themselves:

high-radix addition (i.e., radix 2^g)

multilevel lookahead

High-radix addition increases the latency for generating the auxiliary signals and sum digits but simplifies the carry network. Depending on the implementation method and technology, an optimal radix might exist. Multilevel lookahead is the technique used in practice and is covered in Section 6.2.

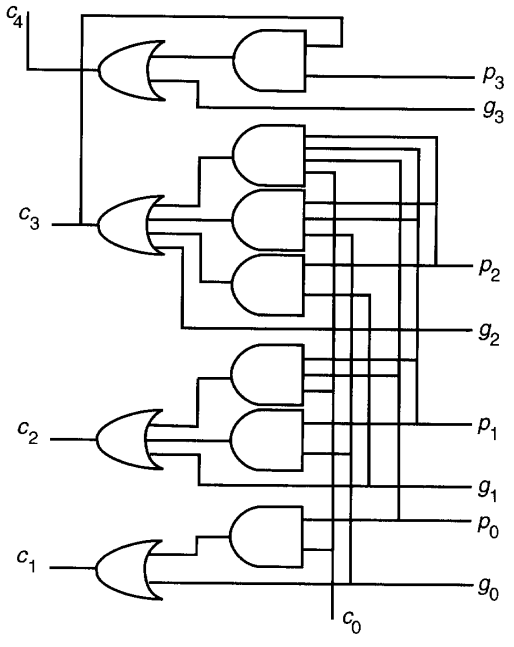


Fig. 6.1 Four-bit carry network with full lookahead.

6.2 CARRY-LOOKAHEAD ADDER DESIGN

Consider radix-16 addition of two binary numbers that are characterized by their g_i and p_i signals. For each radix-16 digit position, extending from bit position i to bit position $i + 3$ of the original binary numbers (where i is a multiple of 4), "block generate" and "block propagate" signals can be derived as follows:

$$g_{[i,i+3]} = g_{i+3} + g_{i+2}p_{i+3} + g_{i+1}p_{i+2}p_{i+3} + g_i p_{i+1} p_{i+2} p_{i+3}$$

$$p_{[i,i+3]} = p_i p_{i+1} p_{i+2} p_{i+3}$$

The preceding equations can be interpreted in the same way as unrolled carry equations: the four bit positions collectively propagate an incoming carry c_i if and only if each of the four positions propagates; they collectively generate a carry if a carry is produced in position $i + 3$, or it is produced in position $i + 2$ and propagated by position $i + 3$, etc.

If we replace the c_4 portion of the carry network of Fig. 6.1 with circuits that produce the block generate and propagate signals $g_{[i,i+3]}$ and $p_{[i,i+3]}$, the 4-bit *lookahead carry generator* of Fig. 6.2 is obtained. Figure 6.3 shows the 4-bit lookahead carry generator in schematic form. We will see shortly that such a block can be used in a multilevel structure to build a carry network of any desired width.

First, however, let us take a somewhat more general view of the block generate and propagate signals. Assuming $i_0 < i_1 < i_2$, we can write:

$$g_{[i_0,i_2-1]} = g_{[i_1,i_2-1]} + g_{[i_0,i_1-1]}p_{[i_1,i_2-1]}$$

This equation essentially says that a carry is generated by the block of positions from i_0 to $i_2 - 1$ if and only if a carry is generated by the $[i_1, i_2 - 1]$ block or a carry generated by the $[i_0, i_1 - 1]$ block is propagated by the $[i_1, i_2 - 1]$ block. Similarly:

$$p_{[i_0,i_2-1]} = p_{[i_0,i_1-1]}p_{[i_1,i_2-1]}$$

In fact the two blocks being merged into a larger block do not have to be contiguous; they can also be overlapping. In other words, for the possibly overlapping blocks $[i_1, j_1]$ and $[i_0, j_0]$, $i_0 \leq i_1 - 1 \leq j_0 < j_1$, we have:

$$g_{[i_0,j_1]} = g_{[i_1,j_1]} + g_{[i_0,j_0]}p_{[i_1,j_1]}$$
$$p_{[i_0,j_1]} = p_{[i_0,j_0]}p_{[i_1,j_1]}$$

Figure 6.4 shows that a 4-bit lookahead carry generator can be used to combine the g and p signals from adjacent or overlapping blocks into the p and g signals for the combined block.

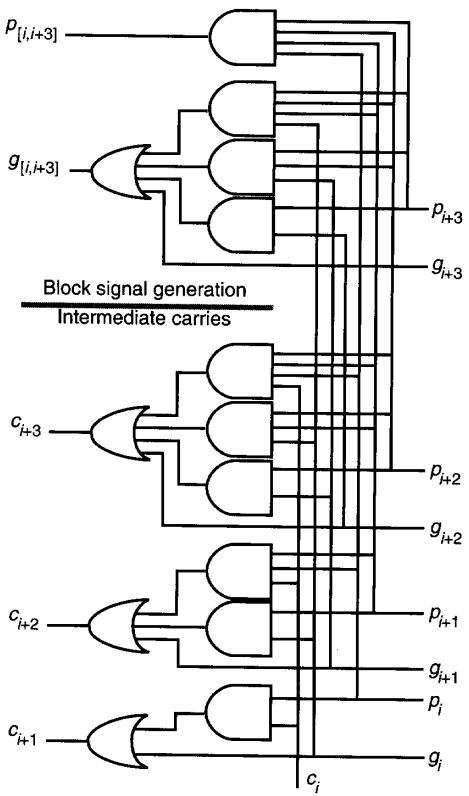


Fig. 6.2 Four-bit lookahead carry generator.

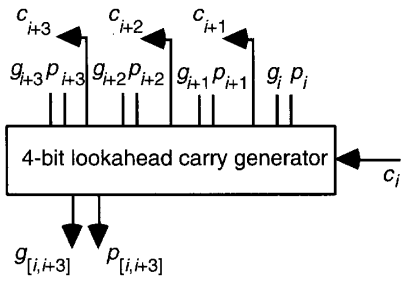


Fig. 6.3 Schematic diagram of a 4-bit lookahead carry generator.

Given the 4-bit lookahead carry generator of Fig. 6.3, it is an easy matter to synthesize wider adders based on a multilevel carry-lookahead scheme. For example, to construct a two-level 16-bit carry-lookahead adder, we need four 4-bit adders and a 4-bit lookahead carry generator, connected together as shown on the upper right quadrant of Fig. 6.5. The 4-bit lookahead carry generator in this case can be viewed as predicting the three intermediate carries in a 4-digit radix-16 addition. The latency through this 16-bit adder consists of the time required for:

Producing the g and p for individual bit positions (1 gate level).

Producing the g and p signals for 4-bit blocks (2 gate levels).

Predicting the carry-in signals c_4, c_8, and c_{12} for the blocks (2 gate levels).

Predicting the internal carries within each 4-bit block (2 gate levels).

Computing the sum bits (2 gate levels).

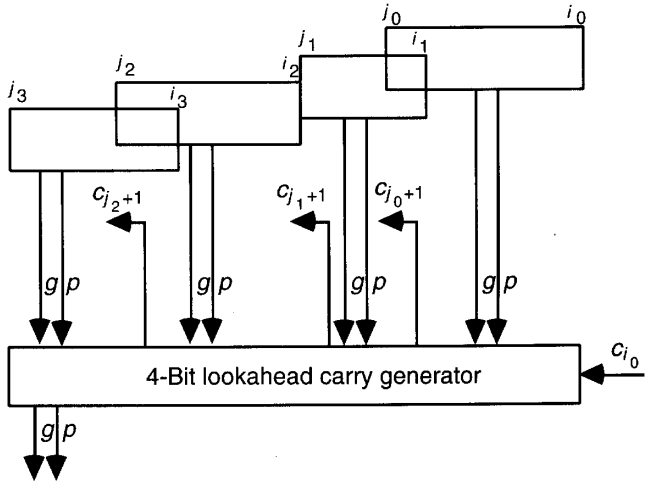


Fig. 6.4 Combining of g and p signals of four (contiguous or overlapping) blocks of arbitrary widths into the g and p signals for the overall block $[i_0, j_3]$.

Thus the total latency for the 16-bit adder is 9 gate levels, which is much better than the 32 gate levels required by a 16-bit ripple-carry adder.

Similarly, to construct a three-level 64-bit carry-lookahead adder, we can use four of the 16-bit adders above plus one 4-bit lookahead carry generator, connected together as shown in Fig. 6.5. The delay will increase by four gate levels with each additional level of lookahead: two levels in the downward movement of the g and p signals, and two levels for the upward propagation of carries through the extra level. Thus, the delay of a k-bit carry-lookahead adder based on 4-bit lookahead blocks is:

$$T_{\text{lookahead-add}} = 4 \log_4 k + 1 \text{ gate levels}$$

Hence, the 64-bit carry-lookahead adder of Fig. 6.5 has a latency of 13 gate levels.

One can of course use 6-bit or 8-bit lookahead blocks to reduce the number of lookahead levels for a given word width. But this may not be worthwhile in view of the longer delays introduced by gates with higher fan-in. When the word width is not a power of 4, some of the inputs and/or outputs of the lookahead carry generators remain unused, and the latency formula becomes $4\lceil \log_4 k \rceil + 1$.

One final point about the design depicted in Fig. 6.5: this 64-bit adder does not produce a carry-out signal (c_{64}), which would be needed in many applications. There are two ways to remedy this problem in carry-lookahead adders. One is to generate c_{out} externally based on auxiliary signals or the operand and sum bits in position $k - 1$:

$$c_{\text{out}} = g_{[0,k-1]} + c_0 p_{[0,k-1]} = x_{k-1} y_{k-1} + \bar{s}_{k-1}(x_{k-1} + y_{k-1})$$

Another is to design the adder to be 1 bit wider than needed (e.g., 61 bits instead of 60), using the additional sum bit as c_{out}.

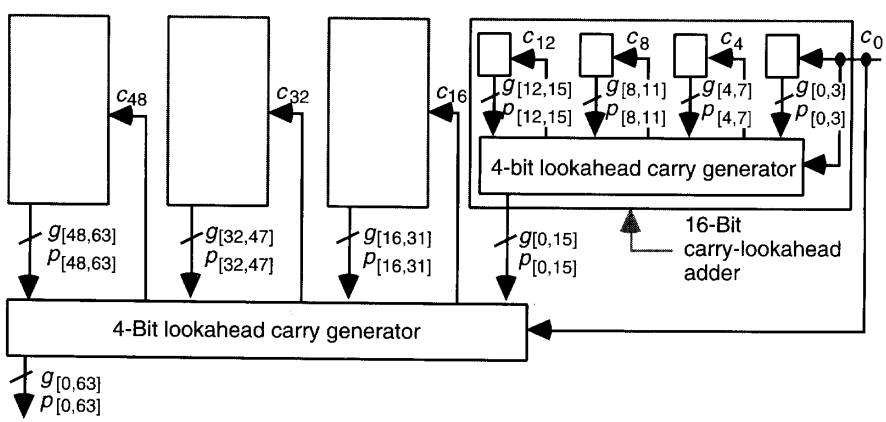


Fig. 6.5 Building a 64-bit carry-lookahead adder from 16 4-bit adders and 5 lookahead carry generators.

6.3 LING ADDER AND RELATED DESIGNS

The Ling adder is a type of carry-lookahead adder that achieves significant hardware savings. Consider the carry recurrence and its unrolling by four steps:

$$\begin{aligned} c_i &= g_{i-1} + c_{i-1}p_{i-1} = g_{i-1} + c_{i-1}t_{i-1} \\ &= g_{i-1} + g_{i-2}t_{i-1} + g_{i-3}t_{i-2}t_{i-1} + g_{i-4}t_{i-3}t_{i-2}t_{i-1} + c_{i-4}t_{i-4}t_{i-3}t_{i-2}t_{i-1} \end{aligned}$$

Ling's modification consists of propagating $h_i = c_i + c_{i-1}$ instead of c_i. To understand the following derivations, we note that g_{i-1} implies $c_i (c_i = 1$ if $g_{i-1} = 1)$, which in turn implies h_i.

$$\begin{aligned} c_{i-1}p_{i-1} &= c_{i-1}p_{i-1} + g_{i-1}p_{i-1} \text{ \{zero\}} + p_{i-1}c_{i-1}p_{i-1} \text{ \{repeated term\}} \\ &= c_{i-1}p_{i-1} + (g_{i-1} + p_{i-1}c_{i-1})p_{i-1} \\ &= (c_{i-1} + c_i)p_{i-1} = h_i p_{i-1} \\ c_i &= g_{i-1} + c_{i-1}p_{i-1} \\ &= h_i g_{i-1} \text{\{since } g_{i-1} \text{ implies } h_i\} + h_i\ p_{i-1} \text{ \{from above\}} \\ &= h_i(g_{i-1} + p_{i-1}) = h_i\ t_{i-1} \\ h_i &= c_i + c_{i-1} = (g_{i-1} + c_{i-1}p_{i-1}) + c_{i-1} \\ &= g_{i-1} + c_{i-1} = g_{i-1} + h_{i-1}t_{i-2} \text{ \{from above\}} \end{aligned}$$

Unrolling the preceding recurrence for h_i, we get:

$$\begin{aligned} h_i &= g_{i-1} + t_{i-2}\ h_{i-1} = g_{i-1} + t_{i-2}(g_{i-2} + h_{i-2}\ t_{i-3}) \\ &= g_{i-1} + g_{i-2} + h_{i-2}\ t_{i-2}\ t_{i-3} \text{ \{since } t_{i-2}\ g_{i-2} = g_{i-2}\} \\ &= g_{i-1} + g_{i-2} + g_{i-3}\ t_{i-3}\ t_{i-2} + h_{i-3}\ t_{i-4}\ t_{i-3}\ t_{i-2} \\ &= g_{i-1} + g_{i-2} + g_{i-3}\ t_{i-2} + g_{i-4}\ t_{i-3}\ t_{i-2} + h_{i-4}\ t_{i-4}\ t_{i-3}\ t_{i-2} \end{aligned}$$

We see that expressing h_i in terms of h_{i-4} needs five product terms, with a maximum four-input AND gate, and a total of 14 gate inputs. By contrast, expressing c_i as

$$c_i = g_{i-1} + g_{i-2}t_{i-1} + g_{i-3}t_{i-2}t_{i-1} + g_{i-4}t_{i-3}t_{i-2}t_{i-1} + c_{i-4}t_{i-4}t_{i-3}t_{i-2}t_{i-1}$$

requires five terms, with a maximum five-input AND gate, and a total of 19 gate inputs. The advantage of h_i over c_i is even greater if we can use wired-OR (3 gates with 9 inputs vs. 4 gates with 14 inputs). Once h_i is known, however, the sum is obtained by a slightly more complex expression compared to $s_i = p_i \oplus c_i$:

$$\begin{aligned} s_i &= p_i \oplus c_i \\ &= p_i \oplus h_i t_{i-1} \quad \text{[and with straightforward manipulation]} \\ &= (t_i \oplus h_{i+1}) + h_i\ g_i\ t_{i-1} \end{aligned}$$

This concludes our presentation of Ling's improved carry-lookahead adder. As indicated, however, related designs have been developed. For example, Doran [Dora88] suggests that one can in general propagate η instead of c where:

$$\eta_{i+1} = f(x_i, y_i, c_i) = \psi(x_i, y_i)c_i + \phi(x_i, y_i)\bar{c}_i$$

The residual functions ψ and ϕ in the preceding Shannon expansion of f around c_i must be symmetric, and there are but eight symmetric functions of the two variables x_i and y_i. Doran [Dora88] shows that not all $8 \times 8 = 64$ possibilities are valid choices for ψ and ϕ, since in some cases the sum cannot be computed based on the η_i values. Dividing the eight symmetric functions of x_i and y_i into the two disjoint subsets $\{0, \bar{t}_i, g_i, \bar{p}_i\}$ and $\{1, t_i, \bar{g}_i, p_i\}$, Doran proves that ψ and ϕ cannot both belong to the same subset. Thus, there are only 32 possible adders. Four of these 32 possible adders have the desirable properties of Ling's adder, which represents the special case of $\psi(x_i, y_i) = 1$ and $\phi(x_i, y_i) = g_i = x_i y_i$.

6.4 CARRY DETERMINATION AS PREFIX COMPUTATION

Consider two contiguous or overlapping blocks B′ and B″ and their associated generate and propagate signal pairs (g', p') and (g'', p''), respectively. As shown in Fig. 6.6, the generate and propagate signals for the merged block B can be obtained from the equations:

$$g = g'' + g'p''$$
$$p = p'p''$$

That is, carry generation in the larger group takes place if the left group generates a carry or the right group generates a carry and the left one propagates it, while propagation occurs if both groups propagate the carry.

We note that in the discussion above, the indices i_0, j_0, i_1, and j_1 defining the two contiguous or overlapping blocks are in fact immaterial, and the same expressions can be written for any two adjacent groups of any length. Let us define the "carry" operator $\diamond$ on (g, p) signal pairs as follows (right side of Fig. 6.6):

$$(g, p) = (g', p') \diamond (g'', p'') \text{ means } g = g'' + g'p'', p = p'p''$$

The carry operator $\diamond$ is *associative*, meaning that the order of evaluation does not affect the value of the expression $(g', p') \diamond (g'', p'') \diamond (g''', p''')$, but it is not *commutative*, since $g'' + g'p''$ is in general not equal to $g' + g''p'$.

Observe that in an adder with no c_{in}, we have $c_{i+1} = g_{[0,i]}$; that is, a carry enters position $i+1$ if and only if one is generated by the block $[0, i]$. In an adder with c_{in}, a carry-in of 1 can be viewed as a carry generated by stage -1; we thus set $p_{-1} = 0$, $g_{-1} = c_{in}$ and compute $g_{[-1,i]}$ for all i. So, the problem remains the same, but with an extra stage ($k+1$ rather than k). The problem of carry determination can, therefore, be formulated as follows:

Given

(g_0, p_0) (g_1, p_1) $\cdots$ (g_{k-2}, p_{k-2}) (g_{k-1}, p_{k-1})

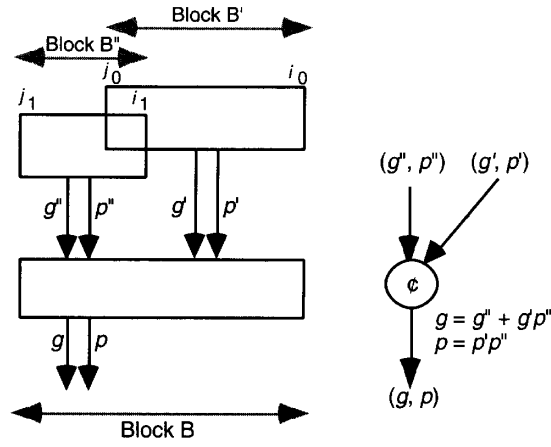


Fig. 6.6 Combining of g and p signals of two (contiguous or overlapping) blocks B' and B'' of arbitrary widths into the g and p signals for the overall block B.

Find

$$(g_{[0,0]}, p_{[0,0]}) \quad (g_{[0,1]}, p_{[0,1]}) \quad \cdots \quad (g_{[0,k-2]}, p_{[0,k-2]}) \quad (g_{[0,k-1]}, p_{[0,k-1]})$$

The desired signal pairs can be obtained by evaluating all the prefixes of

$$(g_0, p_0) \mathbin{¢} (g_1, p_1) \mathbin{¢} \cdots \mathbin{¢} (g_{k-2}, p_{k-1}) \mathbin{¢} (g_{k-1}, p_{k-1})$$

in parallel. In this way, the carry problem is converted to a parallel prefix computation, and any prefix computation scheme can be used to find all the carries.

A parallel prefix computation can be defined with any associative operator. In the following, we use the addition operator with integer operands, in view of its simplicity and familiarity, to illustrate the methods. The *parallel prefix sums* problem is defined as follows:

Given:	x_0	x_1	x_2	x_3	$\cdots$	x_{k-1}
Find:	x_0	$x_0 + x_1$	$x_0 + x_1 + x_2$	$x_0 + x_1 + x_2 + x_3$	$\cdots$	$x_0 + x_1 + \cdots + x_{k-1}$

Any design for this parallel prefix sums problem can be converted to a carry computation network by simply replacing each adder cell with the carry operator of Fig. 6.6. There is one difference worth mentioning, though. Addition is commutative. So if prefix sums are obtained by computing and combining the partial sums in an arbitrary manner, the resulting design may be unsuitable for a carry network. However, as long as blocks whose sums we combine are always contiguous, no problem arises.

Just as one can group numbers in any way to add them, (g, p) signal pairs can be grouped in any way for combining them into block signals. In fact, (g, p) signals give us an additional flexibility in that overlapping groups can be combined without affecting the outcome, whereas in addition, use of overlapping groups would lead to incorrect sums.

6.5 ALTERNATIVE PARALLEL PREFIX NETWORKS

Now, focusing on the problem of computing prefix sums, we can use several strategies to synthesize a parallel prefix sum network. Figure 6.7 is based on a divide-and-conquer approach. The low-order $k/2$ inputs are processed by the subnetwork at the right to compute the prefix sums $s_0, s_1, \cdots, s_{k/2-1}$. Partial prefix sums are computed for the high-order $k/2$ values (the left subnetwork) and $s_{k/2-1}$ (the leftmost output of the first subnetwork) is added to them to complete the computation. Such a network is characterized by the following recurrences for its delay (in terms of adder levels) and cost (number of adder cells):

$$\text{Delay recurrence:} \quad D(k) = D(k/2) + 1 = \log_2 k$$
$$\text{Cost recurrence:} \quad C(k) = 2C(k/2) + k/2 = (k/2)\ \log_2 k$$

A second design for computing prefix sums, again based on a divide-and-conquer approach, is depicted in Fig. 6.8. Here, the inputs are first combined pairwise to obtain the following sequence of length $k/2$:

$$x_0 + x_1 \qquad x_2 + x_3 \qquad x_4 + x_5 \qquad \cdots \qquad x_{k-4} + x_{k-3} \qquad x_{k-2} + x_{k-1}$$

Parallel prefix sum computation on this new sequence yields the odd-indexed prefix sums $s_1, s_3, s_5, \cdots$ for the original sequence. Even-indexed prefix sums are then computed by using $s_{2j} = s_{2j-1} + x_{2j}$. The cost and delay recurrences for the design of Fig. 6.8 are:

$$\text{Delay recurrence:} \quad D(k) = D(k/2) + 2 = 2\ \log_2 k - 1$$
$$\text{actually we will see later that } D(k) = 2\ \log_2 k - 2$$
$$\text{Cost recurrence:} \quad C(k) = C(k/2) + k - 1 = 2k - 2 - \log_2 k$$

So, the first design is faster ($\log_2 k$ as opposed to $2 \log_2 k - 2$ adder levels) but also much more expensive [$(k/2)\ \log_2 k$ as opposed to $2k - 2 - \log_2 k$ adder cells]. The first design also leads to large fan-out requirements if implemented directly in hardware. In other words, the output of one of the adders in the right part must feed the inputs of $k/2$ adders in the left part.

The design shown in Fig. 6.8 is known as the Brent–Kung parallel prefix graph. The 16-input instance of this graph is depicted in Fig. 6.9 [Bren82]. Note that even thoughthe graph of

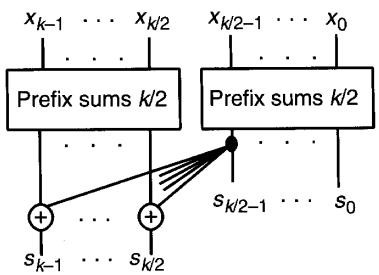


Fig. 6.7 Parallel prefix sums network built of two $k/2$-input networks and $k/2$ adders.

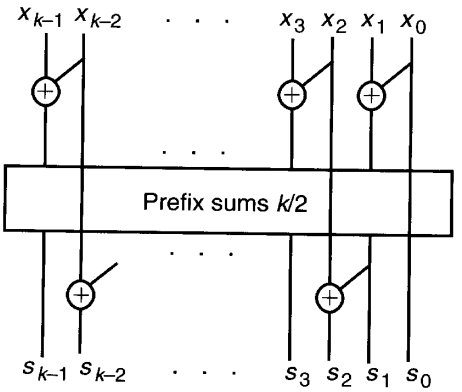


Fig. 6.8 Parallel prefix sums network built of one $k/2$-input network and $k - 1$ adders.

Fig. 6.9 appears to have seven levels, two of the levels near the middle are independent, thus implying a single level of delay. In general, a k-input Brent–Kung parallel prefix graph will have a delay of $2 \log_2 k - 2$ levels and a cost of $2k - 2 - \log_2 k$ cells.

Figure 6.10 depicts a Kogge–Stone parallel prefix graph that has the same delay as the design shown in Fig. 6.7 but avoids its fan-out problem by distributing the computations. A k-input Kogge–Stone parallel prefix graph has a delay of $\log_2 k$ levels and a cost of $k \log_2 k - k + 1$ cells. The Kogge–Stone parallel prefix graph represents the fastest possible implementation of a parallel prefix computation if only two-input blocks are allowed. However, its cost can be prohibitive for large k, in terms of both the number of cells and the dense wiring between them.

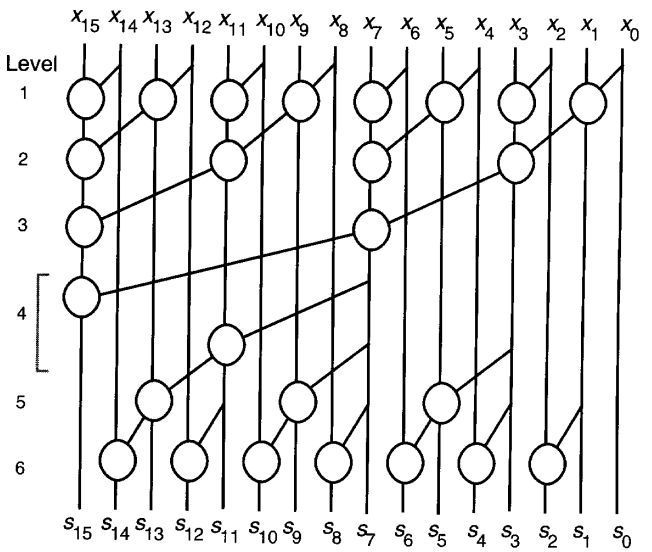


Fig. 6.9 Brent–Kung parallel prefix graph for 16 inputs.

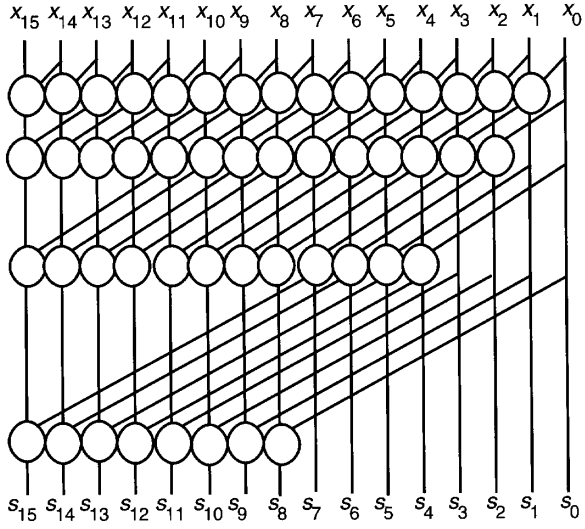


Fig. 6.10 Kogge–Stone parallel prefix graph for 16 inputs.

Many other parallel prefix network designs are possible. For example, it has been suggested that the Brent–Kung and Kogge–Stone approaches be combined to form hybrid designs [Sugl90]. In Fig. 6.11 the middle four of the six levels in the design of Fig. 6.9 (representing an eight-input parallel prefix computation) have been replaced by the eight-input Kogge–Stone network. The resulting design has five levels and 32 cells, placing it between the pure Brent–Kung (six levels, 26 cells) and pure Kogge–Stone (four levels, 49 cells) designs.

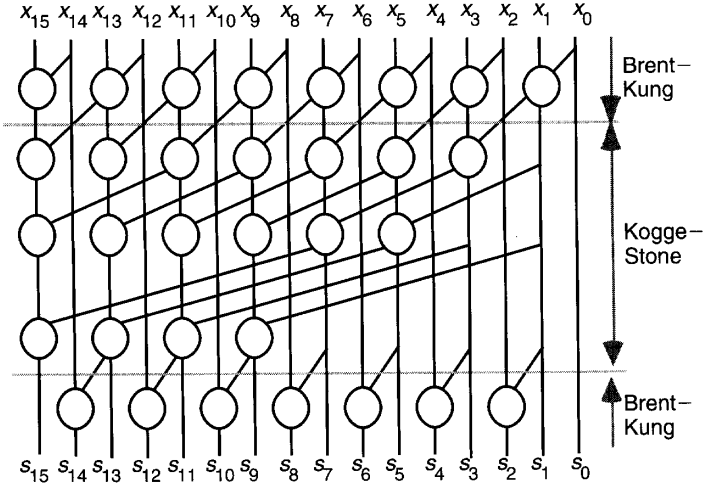


Fig. 6.11 A hybrid Brent–Kung/Kogge–Stone parallel prefix graph for 16 inputs.

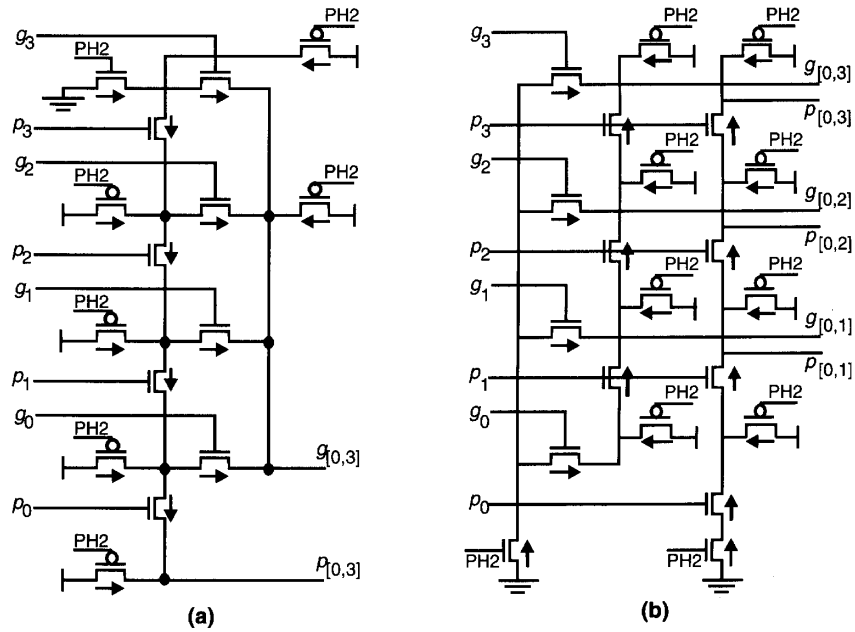


Fig. 6.12 Example 4-bit Manchester carry chain designs in CMOS technology [Lync92].

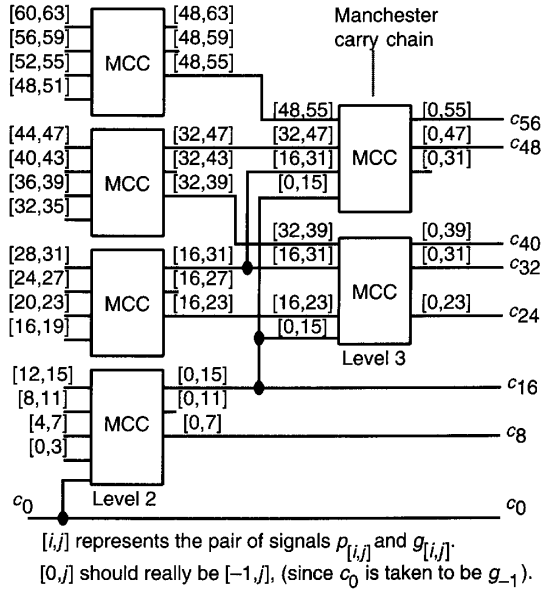


Fig. 6.13 Spanning-tree carry-lookahead network [Lync92]. The 16 MCCs at level 1 that produce generate and propagate signals for 4-bit blocks are not shown.

More generally, if a single Brent–Kung level is used along with a $k/2$-input Kogge–Stone design, delay and cost of the hybrid network become $\log_2 k + 1$ and $(k/2)\log_2 k$, respectively. The resulting design is thus close to minimum in terms of delay (only one level more than Kogge–Stone) but costs roughly half as much.

The theory of parallel prefix graphs is quite rich and well developed. There exist both theoretical bounds and actual designs with different restrictions on fan-in/fan-out and with various optimality criteria in terms of cost and delay (see, e.g., Chapters 5–7, pp. 133–211, of [Laks94]).

In devising their design, Brent and Kung [Bren82] were motivated by the need to reduce the chip area in VLSI layout of the carry network. Other performance or hardware limitations may also be considered. The nice thing about formulating the problem of carry determination as a parallel prefix computation is that theoretical results and a wealth of design strategies carry over with virtually no effort. Not all such relationships between carry networks and parallel prefix networks, or the virtually unlimited hybrid combinations, have been explored in full.

6.6 VLSI IMPLEMENTATION ASPECTS

The carry network of Fig. 6.9 is quite suitable for VLSI implementation, but it might be deemed too slow for high-performance designs and/or wide words. Many designers have proposed alternate networks that offer reduced latency by using features of particular technologies and taking advantage of related optimizations. We review one example here that is based on radix-256 addition of 56-bit numbers as implemented in the Advanced Micro Devices Am29050 CMOS microprocessor. The following description is based on a 64-bit version of the adder.

In radix-256 addition of 64-bit numbers, only the carries c_8, c_{16}, c_{24}, c_{32}, c_{40}, c_{48}, and c_{56} need to be computed. First, 4-bit Manchester carry chains (MCCs) of the type shown in Fig. 6.12a are used to derive g and p signals for 4-bit blocks. These signals, denoted by [0, 3], [4, 7], [8, 11], etc. at the left edge of Fig. 6.13, then form the inputs to one 5-bit and three 4-bit MCCs that in turn feed two more MCCs in the third level. The MCCs in Fig. 6.13 are of the type shown in Fig. 6.12b; that is, they also produce intermediate g and p signals. For example, the MCC with inputs [16, 19], [20, 23], [24, 27], and [28, 31] yields the intermediate outputs [16, 23] and [16, 27], in addition to the signal pair [16, 31] for the entire group.

PROBLEMS

6.1 Borrow-lookahead subtractor We know that any carry network producing the carries c_i based on g_i and p_i signals can be used, with no modification, as a borrow propagation circuit to find the borrows b_i.

a. Define the borrow-generate γ_i and borrow-propagate π_i signals in general and for the special case of binary operands.

b. Present the design of a circuit to compute the difference digit d_i from γ_i, π_i, and the incoming borrow b_i.

6.2 One's-complement carry-lookahead adder Discuss how the requirement for end-around carry in 1's-complement addition affects the design and performance of a carry-lookahead adder.

6.3 High-radix carry-lookahead adder Consider radix-2^h addition of binary numbers and assume that the total time needed for producing the digit g and p signals, and determining the sum digits after all carries are known, equals δh, where δ is a constant. Carries are determined by a multilevel lookahead network using unit-time 2-bit lookahead carry generators. Derive the optimal radix that minimizes the addition latency as a function of δ and discuss.

6.4 Unconventional carry-lookahead adder Consider the following method for synthesizing a k-bit adder from four $k/4$-bit adders and a 4-bit lookahead carry generator. The $k/4$-bit adders have no group g or p output. Both the g_i and p_i inputs of the lookahead carry generator are connected to the carry-out of the ith $k/4$-bit adder, $0 \leq i \leq 3$. Intermediate carries of the lookahead carry generator and c_{in} are connected to the carry-in inputs of the $k/4$-bit adders. Will the suggested circuit add correctly? Find the adder's latency or justify your negative answer.

6.5 Decimal carry-lookahead adder Consider the design of a 15-digit decimal adder for unsigned numbers (width = 60 bits).

a. Design the required circuits for carry-generate and carry-propagate assuming binary-coded decimal digits.

b. Repeat part a with excess-3 encoding for the decimal digits, where digit value a is represented by the binary encoding of $a + 3$.

c. Complete the design of the decimal adder of part b by proposing a carry-lookahead circuit and the sum computation circuit.

6.6 Carry lookahead with overlapped blocks

a. Write down the indices for the g and p signals on Fig. 6.4.

b. Prove that the combining equations for the g and p signals for two contiguous blocks also apply to overlapping blocks (see Fig. 6.6).

6.7 Latency of a carry-lookahead adder Complete Fig. 6.5 by drawing boxes for the g and p logic and the sum computation logic. Then draw a critical path on the resulting diagram and indicate the number of gate levels of delay on each segment of the path.

6.8 Ling adder or subtractor

a. Show the complete design of a counterpart to the lookahead carry generator of Fig. 6.2 using Ling's method.

b. How does the design of a Ling subtractor differ from that of a Ling adder? Present complete designs for all the parts that are different.

6.9 Ling-type adders Based on the discussion at the end of Section 6.3, derive one the other three Ling-type adders proposed by Doran [Dora88]. Compare the derived adder to a Ling adder.

6.10 Fixed-priority arbiters A fixed-priority arbiter has k request inputs $R_{k-1}, \cdots, R_1, R_0$, and k grant outputs G_i. At each arbitration cycle, at most one of the grant signals is 1 and that corresponds to the highest-priority request signal (i.e., $G_i = 1$ iff $R_i = 1$ and $R_j = 0$ for $j < i$).

a. Design a synchronous arbiter using ripple-carry techniques. *Hint:* Consider $c_0 = 0$ along with carry propagate and annihilate rules; there is no carry generation.

b. Design the arbiter using carry-lookahead techniques. Determine the number of lookahead levels required with 64 inputs and estimate the total arbitration delay.

6.11 Carry-lookahead incrementer

a. Design a 16-bit incrementer using the carry-lookahead principle.

b. Repeat part a using Ling's approach.

c. Compare the designs of parts a and b with respect to delay and cost.

6.12 Parallel prefix networks Find delay and cost formulas for the Brent–Kung and Kogge–Stone designs when the word width k is not a power of 2.

6.13 Parallel prefix networks

a. Draw Brent–Kung, Kogge–Stone, and hybrid parallel prefix graphs for 12, 20, and 24 inputs.

b. Using the results of part a, plot the cost, delay, and cost–delay product for the five types of network for $k = 12, 16, 20, 24, 32$ bits and discuss.

6.14 Hybrid carry-lookahead adders

a. Find the depth and cost of a 64-bit hybrid carry network with two levels of the Brent–Kung scheme at each end and the rest built by the Kogge–Stone construction.

b. Compare the design of part a to pure Brent–Kung and Kogge–Stone schemes and discuss.

6.15 Parallel prefix networks

a. Obtain delay and cost formulas for a hybrid parallel prefix network that has l levels of Brent–Kung design at the top and bottom and a $k/2^l$-input Kogge–Stone network in the middle.

b. Use the delay–cost–product figure of merit to find the best combination of the two approaches for word lengths from 8 to 64 (powers of 2 only).

6.16 Speed and cost limits for carry computation Consider the computation of c_i, the carry into the ith stage of an adder, based on the g_j and t_j signals using only two-input AND and OR gates. Note that only the computation of c_i, independent of other carries, is being considered.

a. What is the minimum possible number of AND/OR gates required?

b. What is the minimum possible number of gate levels in the circuit?

c. Can one achieve the minima of parts a and b simultaneously? Explain.

6.17 Variable-block carry-lookahead adders Study the benefits of using nonuniform widths for the MCC blocks in a carry-lookahead adder of the type discussed in Section 6.6 [Kant93].

REFERENCES

[Bayo83] Bayoumi, M. A., G. A. Jullien, and W. C. Miller, "An Area–Time Efficient NMOS Adder," *Integration: The VLSI Journal*, Vol. 1, pp. 317–334, 1983.

[Bren82] Brent, R. P., and H. T. Kung, "A Regular Layout for Parallel Adders," *IEEE Trans. Computers*, Vol. 31, pp. 260–264, 1982.

[Dora88] Doran, R. W., "Variants of an Improved Carry Look-Ahead Adder," *IEEE Trans. Computers*, Vol. 37, No. 9, pp. 1110–1113, 1988.

[Han87] Han, T., and D. A. Carlson, "Fast Area-Efficient Adders," *Proc. 8th Symp. Computer Arithmetic*, pp. 49–56, 1987.

[Kant93] Kantabutra, V., "A Recursive Carry-Lookahead/Carry-Select Hybrid Adder," *IEEE Trans. Computers*, Vol. 42, No. 12, pp. 1495–1499, 1993.

[Laks94] Lakshmivarahan, S., and S. K. Dhall, *Parallel Computing Using the Prefix Problem*, Oxford University Press, 1994.

[Ling81] Ling, H., "High-Speed Binary Adder," *IBM J. Research and Development*, Vol. 25, No. 3, pp. 156–166, 1981.

[Lync92] Lynch, T., and E. Swartzlander, "A Spanning Tree Carry Lookahead Adder," *IEEE Trans. Computers*, Vol. 41, No. 8, pp. 931–939, 1992.

[Ngai84] Ngai, T. F., M. J. Irwin, and S. Rawat, "Regular Area–Time Efficient Carry-Lookahead Adders," *J. Parallel and Distributed Computing*, Vol. 3, No. 3, pp. 92–105, 1984.

[Sugl90] Sugla, B., and D. A. Carlson, "Extreme Area–Time Tradeoffs in VLSI," *IEEE Trans. Computers*, Vol. 39, No. 2, pp. 251–257, 1990.

[Wei90] Wei, B. W. Y., and C. D. Thompson, "Area–Time Optimal Adder Design," *IEEE Trans. Computers*, Vol. 39, No. 5, pp. 666–675, 1990.

[Wein56] Weinberger, A., and J. L. Smith, "A One-Microsecond Adder Using One-Megacycle Circuitry," *IRE Trans. Computers*, Vol. 5, pp. 65–73, 1956.

Chapter 7 VARIATIONS IN FAST ADDERS

The carry-lookahead method of Chapter 6 represents the most widely used design for high-speed adders in modern computers. Certain alternative designs, however, either are quite competitive with carry-lookahead adders or offer advantages with particular hardware realizations or technology constraints. The most important of these alternative designs, and various hybrid combinations, are discussed in this chapter.

7.1 SIMPLE CARRY-SKIP ADDERS

Consider a 4-bit group or block in a ripple-carry adder, from stage i to $i + 3$, where i is a multiple of 4 (Fig. 7.1a). A carry into stage i propagates through this group of 4 bits if and only if it propagates through all four of its stages. Thus, a "group propagate" signal is defined as $p_{[i,i+3]} = p_i p_{i+1} p_{i+2} p_{i+3}$, which is computable from individual propagate signals by a single four-input AND gate. To speed up carry propagation, one can establish bypass or skip paths around 4-bit blocks, as shown in Fig. 7.1b.

Let us assume that the delay of the two-level skip logic is equal to carry propagation delay through a single bit position. Then, the worst-case propagation delay through the carry-skip adder of Fig. 7.1b corresponds to a carry that is generated in stage 0, ripples through stages 1–3, goes through the OR gate, skips the middle two groups, and ripples in the last group from stage 12 to stage 15. This leads to 8.5 stages of propagation (17 gate levels) compared to 16 stages (32 gate levels) for a 16-bit ripple-carry adder.

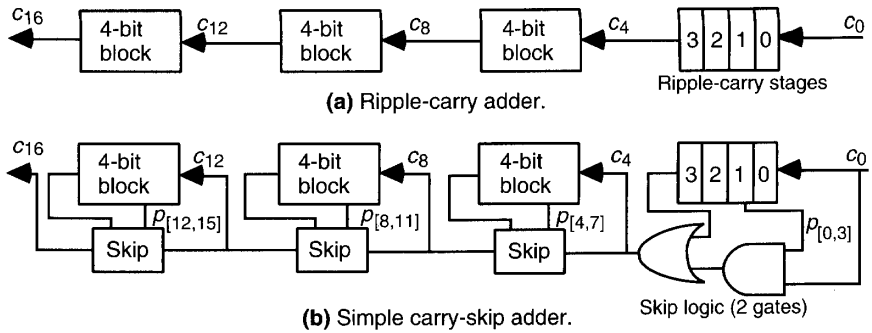


Fig. 7.1 Converting a 16-bit ripple-carry adder into a simple carry-skip adder with 4-bit skip blocks.

Generalizing from the preceding example, the worst-case carry-propagation delay in a k-bit carry-skip adder with fixed block width b, assuming that one stage of ripple has the same delay as one skip, can be derived:

$$\begin{aligned} T_{\text{fixed-skip-add}} &= \underbrace{(b-1)}_{\text{in block 0}} + \underbrace{0.5}_{\text{OR gate}} + \underbrace{(k/b-2)}_{\text{skips}} + \underbrace{(b-1)}_{\text{in last block}} \\ &\approx 2b + k/b - 3.5 \text{ stages} \end{aligned}$$

The optimal fixed block size can be derived by equating $dT_{\text{fixed-skip-add}}/db$ with 0:

$$\frac{dT_{\text{fixed-skip-add}}}{db} = 2 - k/b^2 = 0 \Rightarrow b^{\text{opt}} = \sqrt{k/2}$$

The adder delay with the optimal block size above is:

$$T^{\text{opt}}_{\text{fixed-skip-add}} = 2\sqrt{k/2} + \frac{k}{\sqrt{k/2}} - 3.5 = 2\sqrt{2k} - 3.5$$

For example, to construct a 32-bit carry-skip adder with fixed-size blocks, we set $k = 32$ in the preceding equations to obtain $b^{\text{opt}} = 4$ bits and $T^{\text{opt}}_{\text{fixed-skip-add}} = 12.5$ stages (25 gate levels). By comparison, the propagation delay of a 32-bit ripple-carry adder is more than 2.5 times longer.

Clearly, a carry that is generated in, or absorbed by, one of the inner blocks travels a shorter distance through the skip blocks. We can thus afford to allow more ripple stages for such a carry without increasing the overall adder delay. This leads to the idea of variable skip-block sizes.

Let there be t blocks of widths $b_0, b_1, \cdots, b_{t-1}$ going from right to left (Fig. 7.2). Consider the two carry paths (1) and (2) in Fig. 7.2, both starting in block 0, one ending in block $t-1$ and the other in block $t-2$. Carry path (2) goes through one fewer skip than (1), so we can make block $t-2$ one bit wider than block $t-1$ without increasing the total adder delay. Similarly, by comparing carry paths (1) and (3), we conclude that block 1 can be one bit wider than block 0. So, assuming for ease of analysis that $b_0 = b_{t-1} = b$ and that the number t of blocks is even, the optimal block widths are:

$$b \quad b+1 \quad \cdots \quad \frac{b+t}{2}-1 \quad \frac{b+t}{2}-1 \quad \cdots \quad b+1 \quad b$$

The first assumption ($b_0 = b_{t-1}$) is justified because the total delay is a function of $b_0 + b_{t-1}$ rather than their individual values and the second one (t even) does not affect the results significantly.

Based on the preceding block widths, the total number of bits in the t blocks is:

$$2[b + (b+1) + \cdots + (b + t/2 - 1)] = t(b + t/4 - 1/2)$$

Equating the total above with k yields:

$$b = k/t - t/4 + 1/2$$

The adder delay with the preceding assumptions is:

$$\begin{aligned} T_{\text{var-skip-add}} &= 2(b-1) + 0.5 + t - 2 \\ &= \frac{2k}{t} + \frac{t}{2} - 2.5 \end{aligned}$$

The optimal number of blocks is thus obtained as follows:

$$\frac{dT_{\text{var-skip-add}}}{dt} = \frac{-2k}{t^2} + \frac{1}{2} = 0 \Rightarrow t^{\text{opt}} = 2\sqrt{k}$$

Note that the optimal number of blocks with variable-size blocks is $\sqrt{2}$ times larger than that obtained with fixed-size blocks. Note also that with the optimal number of blocks, b becomes 1/2; thus we take it to be 1. The adder delay with t^{opt} blocks is

$$T^{\text{opt}}_{\text{var-skip-add}} \approx 2\sqrt{k} - 2.5$$

which is roughly a factor of $\sqrt{2}$ smaller than that obtained with optimal fixed-size skip-blocks.

The preceding analyses were based on a number of simplifying assumptions. For example, skip and ripple delays were assumed to be equal and ripple delay was assumed to be linearly proportional to the block width. These may not be true in practice. With CMOS implementation, for example, the ripple delay in a Manchester carry chain grows as the square of the block width. The analyses for obtaining the optimal fixed or variable block size carry-skip adder must be

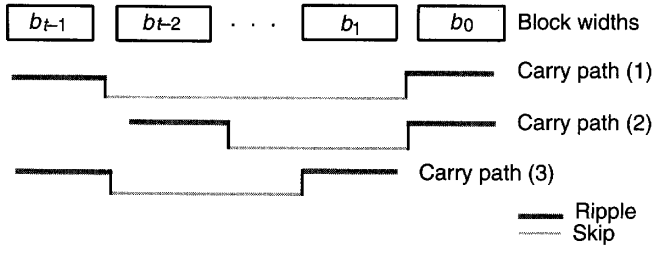


Fig. 7.2 Carry-skip adder with variable-size blocks and three sample carry paths.

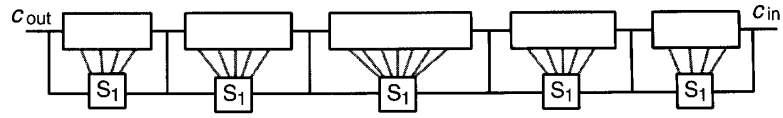


Fig. 7.3 Schematic diagram of a one-level carry-skip adder.

appropriately modified in such cases. A number of researchers have used various assumptions about technology-dependent parameters to deal with this optimization problem. Some of these variations are explored in the end-of-chapter problems.

7.2 MULTILEVEL CARRY-SKIP ADDERS

A (single-level) carry-skip adder of the types discussed in Section 7.1 can be represented schematically as in Fig. 7.3. The dotted lines between the various blocks and the first-level skip logic S_1 indicate that the control signal for the skip logic is derived from the propagate signals of the bit positions within the corresponding block. This relationship will be implicit in everything that follows, so the dotted lines are not shown hereafter.

In our subsequent discussions, we ignore the half-stage delay attributed to the single OR gate immediately preceding the first skip on the carry path (in contrast to the analyses of Section 7.1). This simplifies our discussions but has no significant effect on our procedures or conclusions. In addition, we continue to assume that the ripple and skip delays are equal, although the analyses can be easily modified to account for different ripple and skip delays. We thus equate the carry-skip adder delay with the worst-case sum, over all possible carry paths, of the number of ripple stages and the number of skip stages.

Multilevel carry-skip adders are obtained if we allow a carry to skip over several blocks at once. Figure 7.4 depicts a two-level carry-skip adder in which second-level skip logic has been provided for the leftmost three blocks. The signal controlling this second-level skip logic is derived as the logical AND of the first-level skip signals. A carry that would need 3 time units to skip these three blocks in a single-level carry-skip adder can now do so in a single time unit.

If the rightmost/leftmost block in a carry-skip adder is short, skipping it may not yield any advantage over allowing the carry to ripple through the block. In this case, the carry-skip adder of Fig. 7.4 can be simplified by removing such inefficient skip circuits. Figure 7.5 shows the resulting two-level carry-skip adder. With our simplifying assumption about ripple and skip delays being equal, the first-level skip circuit should be eliminated only for 1-bit, and possibly 2-bit, blocks (remember that generating the skip control signal also takes some time).

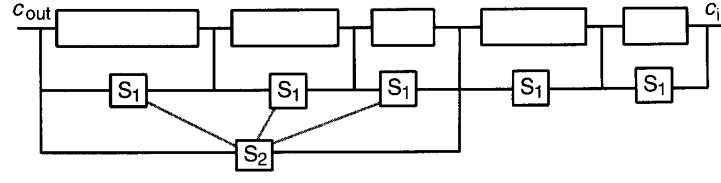


Fig. 7.4 Example of a two-level carry-skip adder.

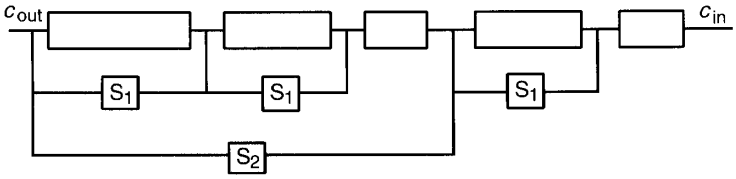


Fig. 7.5 Two-level carry-skip adder optimized by removing the short-block skip circuits.

Example 7.1 Assume that each of the following operations takes 1 unit of time: generation of g_i and p_i signals, generation of a level-i skip signal from level-$(i-1)$ skip signals, ripple, skip, and computation of sum bit once the incoming carry is known. Build the widest possible single-level carry-skip adder with a total delay not exceeding 8 time units.

The numbers given on the adder diagram of Fig. 7.6 denote the time steps when the various signals stabilize, assuming that c_{in} is available at time 0. At the right end, block width is limited by the output timing requirement. For example, b_1 cannot be more than 3 bits if its output is to be available at time 3 (one time unit is taken by g_i, p_i generation at the rightmost bit, plus two time units for propagation across the other two bits). Block 0 is an exception, because to accommodate c_{in}, its width must be reduced by 1 bit. At the left end, block width is limited by input timing. For example, b_4 cannot be more than 3 bits, given that its input becomes available at time 5 and the total adder delay is to be 8 units. Based on this analysis, the maximum possible adder width is $1 + 3 + 4 + 4 + 3 + 2 + 1 = 18$ bits.

Example 7.2 With the same assumptions as in Example 7.1, build the widest possible two-level carry-skip adder with a total delay not exceeding 8 time units.

We begin with an analysis of skip paths at level 2. In Fig. 7.7a, the notation $\{\beta, \alpha\}$ for a block means that the block's carry-out must become available no later than $T_{\text{produce}} = \beta$ and that the block's carry-in can take $T_{\text{assimilate}} = \alpha$ time units to propagate within the block without exceeding the overall time limit of 8 units. The remaining problem is to construct single-level carry-skip adders with the parameters $T_{\text{produce}} = \beta$ and $T_{\text{assimilate}} = \alpha$. Given the delay pair $\{\beta, \alpha\}$, the number of first-level blocks (subblocks) will be $\gamma = min(\beta - 1, \alpha)$, with the width of the ith subblock, $0 \leq i \leq \gamma - 1$, given by $b_i = min(\beta - \gamma + i + 1, \alpha - i)$; the only

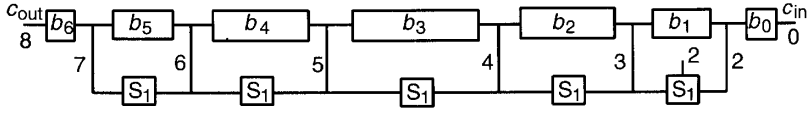


Fig. 7.6 Timing constraints of a single-level carry-skip adder with a delay of 8 units.

exception is subblock 0 in block A, which has one fewer bit (why?). So, the total width of such a block is $\sum_{i=0}^{\gamma-1} min(\beta - \gamma + i + 1, \alpha - i)$. Table 7.1 summarizes our analyses for the second-level blocks A–F. Note that the second skip level has increased the adder width from 18 bits (in Example 7.1) to 30 bits. Figure 7.7b shows the resulting two-level carry-skip adder.

The preceding analyses of one- and two-level carry-skip adders are based on many simplifying assumptions. If these assumptions are relaxed, the problem may no longer lend itself to analytical solution. Chan et al. [Chan92] use dynamic programming to obtain optimal configurations of carry-skip adders for which the various worst-case delays in a block of b full-adder units are characterized by arbitrary given functions (Fig. 7.8). These delays include:

$I(b)$ Internal carry-propagate delay for the block
$G(b)$ Carry-generate delay for the block
$A(b)$ Carry-assimilate delay for the block

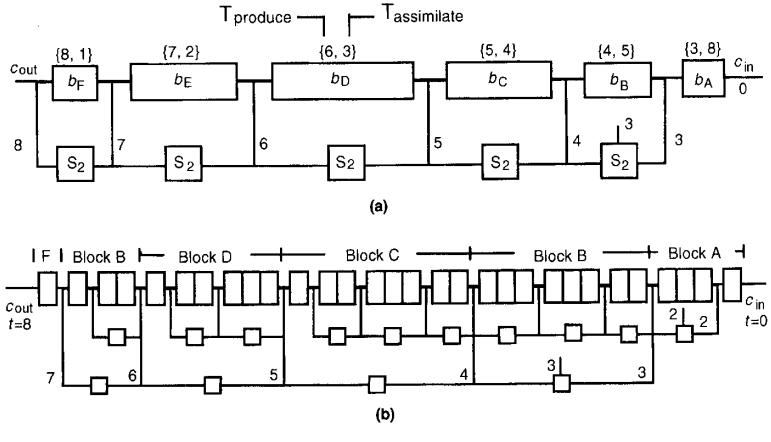


Fig. 7.7 Two-level carry-skip adder with a delay of 8 units: (a) Initial timing constraints, (b) Final design.

TABLE 7.1
Second-level constraints $T_{produce}$ and $T_{assimilate}$, with associated subblock and block widths, in a two-level carry-skip adder with a total delay of 8 time units (Fig. 7.7)

Block	$T_{produce}$	$T_{assimilate}$	Number of subblocks	Subblock widths (bits)	Block Width (bits)
A	3	8	2	1, 3	4
B	4	5	3	2, 3, 3	8
C	5	4	4	2, 3, 2, 1	8
D	6	3	3	3, 2, 1	6
E	7	2	2	2, 1	3
F	8	1	1	1	1

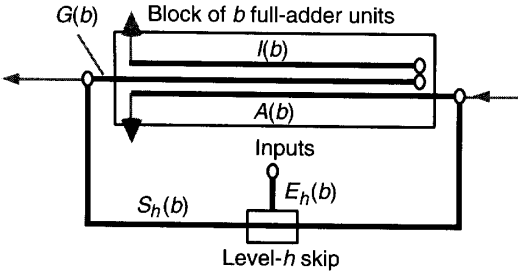


Fig. 7.8 Generalized delay model for carry-skip adders.

In addition, skip and enable delay functions, $S_h(b)$ and $E_h(b)$, are defined for each skip level h. In terms of this general model, our preceding analysis can be characterized as corresponding to $I(b) = b - 1$, $G(b) = b$, $A(b) = b$, $S_h(b) = 1$, and $E_h(b) = h + 1$. This is the model assumed by Turrini [Turr89]. Similar methods can be used to derive optimal block widths in variable-block carry-lookahead adders [Chan92].

7.3 CARRY-SELECT ADDERS

One of the earliest logarithmic time adder designs is based on the conditional-sum addition algorithm. In this scheme, blocks of bits are added in two ways: assuming an incoming carry of 0 or of 1, with the correct outputs selected later as the block's true carry-in becomes known. With each level of selection, the number of known output bits doubles, leading to a logarithmic number of levels and thus logarithmic time addition. Underlying the building of conditional-sum adders is the carry-select principle, described in this section.

A (single-level) carry-select adder is one that combines three $k/2$-bit adders of any design into a k-bit adder (Fig. 7.9). One $k/2$-bit adder is used to compute the lower half of the k-bit sum directly. Two $k/2$-bit adders are used to compute the upper $k/2$ bits of the sum and the carry-out under two different scenarios: $c_{k/2} = 0$ or $c_{k/2} = 1$. The correct values for the adder's carry-out signal and the sum bits in positions $k/2$ through $k-1$ are selected when the value of $c_{k/2}$ becomes known. The delay of the resulting k-bit adder is two gate levels more than that of the $k/2$-bit adders that are used in its construction.

The following simple analysis demonstrates the cost-effectiveness of the carry-select method. Let us take the cost and delay of a single-bit 2-to-1 multiplexer as our units and assume that the cost and delay of a k-bit adder are $C_{\text{add}}(k)$ and $T_{\text{add}}(k)$, respectively. Then, the cost and delay of the carry-select adder of Fig. 7.9 are:

$$C_{\text{select-add}}(k) = 3C_{\text{add}}(k/2) + k/2 + 1$$
$$T_{\text{select-add}}(k) = T_{\text{add}}(k/2) + 1$$

If we take the product of cost and delay as our measure of cost effectiveness, the carry-select scheme of Fig. 7.9 is more cost-effective than the scheme used in synthesizing its component adders if and only if:

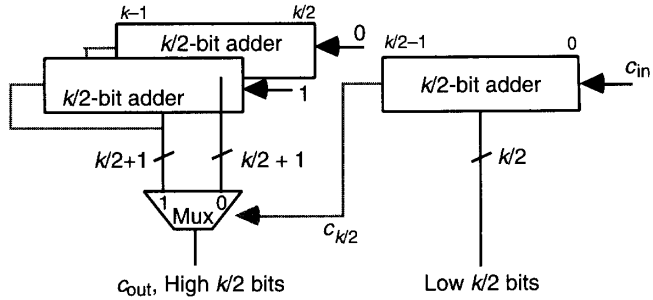


Fig. 7.9 Carry-select adder for k-bit numbers built from three $k/2$-bit adders.

$$[3C_{\text{add}}(k/2) + k/2 + 1][T_{\text{add}}(k/2) + 1] < C_{\text{add}}(k)T_{\text{add}}(k)$$

For ripple-carry adders, we have $C_{\text{add}}(k) = \alpha k$ and $T_{\text{add}}(k) = \tau k$. To simplify the analysis, assume $\tau = \alpha/2 > 1$. Then, it is easy to show that the carry-select method is more cost-effective that the ripple-carry scheme if $k > 16/(\alpha - 1)$. For $\alpha = 4$ and $\tau = 2$, say, the carry-select approach is almost always preferable to ripple-carry. Similar analyses can be carried out to compare the carry-select method against other addition schemes.

Note that in the preceding analysis, the use of three complete $k/2$-bit adders was assumed. With some adder types, the two $k/2$-bit adders at the left of Fig. 7.9 can share some hardware, thus leading to even greater cost effectiveness. For example, if the component adders used are of the carry lookahead variety, much of the carry network can be shared between the two adders computing the sum bits with $c_{k/2} = 0$ and $c_{k/2} = 1$ (how?).

Note that the carry-select method works just as well when the component adders have different widths. For example, Fig. 7.9 could have been drawn with one a-bit and two b-bit adders used to form an $(a + b)$-bit adder. Then c_a would be used to select the upper b bits of the sum through a $(b + 1)$-bit multiplexer. Unequal widths for the component adders is appropriate when the delay in deriving the selection signal c_a is different from that of the sum bits.

Figure 7.10 depicts how the carry-select idea can be carried one step further to obtain a two-level carry-select adder. Sum and carry-out bits are computed for each $k/4$-bit block (except for the rightmost one) under two scenarios. The three first-level multiplexers, each of which is $k/4 + 1$ bits wide, merge the results of $k/4$-bit blocks into those of $k/2$-bit blocks. Note how the carry-out signals of the adders spanning bit positions $k/2$ through $3k/4 - 1$ are used to select the most-significant $k/4$ bits of the sum under the two scenarios of $c_{k/2} = 0$ or $c_{k/2} = 1$. At this stage, $k/2$ bits of the final sum are known. The second-level multiplexer, which is $k/2 + 1$ bits wide, is used to select appropriate values for the upper $k/2$ bits of the sum (positions $k/2$ through $k - 1$) and the adder's carry-out.

Comparing the two-level carry-select adder of Fig. 7.10 to a similar two-level carry-lookahead adder (Fig. 6.5, but with 2-bit, rather than 4-bit, lookahead carry generators), we note that the one-directional top-to-bottom data flow in Fig. 7.10 makes pipelining easier and more efficient. Of course, from Section 6.5 and the example in Fig. 6.13, we know that carry-lookahead adders can also be implemented to possess one-directional data flow. In such cases, comparison is somewhat more difficult, insofar as carry-select adders have a more complex upper structure (the small adders) and simpler lower structure (the multiplexers).

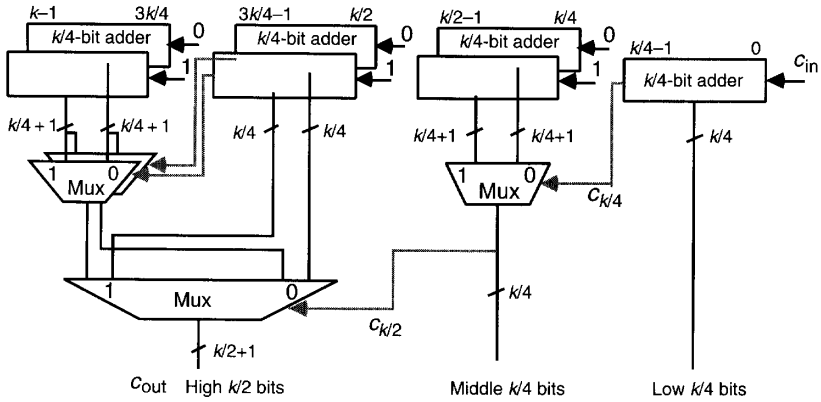


Fig. 7.10 Two-level carry-select adder built of $k/4$-bit adders.

Which design comes out ahead for a given word width depends on the implementation technology, performance requirements, and other design constraints. Very often, the best choice is a hybrid combination of carry-select and carry-lookahead (see Section 7.5).

7.4 CONDITIONAL-SUM ADDER

The process that led to the two-level carry-select adder of Fig. 7.10 can be continued to derive a three-level k-bit adder built of $k/8$-bit adders, a four-level adder composed of $k/16$-bit adders, and so on. A logarithmic time conditional-sum adder results if we proceed to the extreme of having single-bit adders at the very top. Thus, taking the cost and delay of a single-bit 2-to-1 multiplexer as our units, the cost and delay of a conditional-sum adder are characterized by the following recurrences:

$$C(k) \approx 2C(k/2) + k + 2 \approx k(\log_2 k + 2) + kC(1)$$
$$T(k) = T(k/2) + 1 = \log_2 k + T(1)$$

where $C(1)$ and $T(1)$ are the cost and delay of the circuit of Fig. 7.11 used at the top to derive the sum and carry bits with a carry-in of 0 and 1. The term $k + 2$ in the first recurrence represents an upper bound on the number of single-bit 2-to-1 multiplexers needed for combining two $k/2$-bit adders into a k-bit adder.

The recurrence for cost is approximate, since for simplicity, we have ignored the fact that the right half of Fig. 7.10 is less complex than its left half. In other words, we have assumed that two parallel $(b + 1)$-bit multiplexers are needed to combine the outputs from b-bit adders, although in some cases, one is enough.

An exact analysis leads to a comparable count for the number of single-bit multiplexers needed in a conditional-sum adder. Assuming that k is a power of 2, the required number of multiplexers for a k-bit adder is

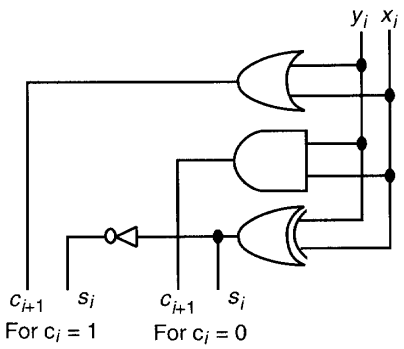


Fig. 7.11 Top-level block for one bit position of a conditional-sum adder.

$$(k/2 + 1) + 3(k/4 + 1) + 7(k/8 + 1) + \cdots + (k - 1)2 = (k - 1)(\log_2 k + 1)$$

leading to an overall cost of $(k - 1)(\log_2 k + 1) + kC(1)$.

The conditional-sum algorithm can be visualized by the 16-bit addition example shown in Table 7.2.

Given that a conditional-sum adder is actually a $(\log_2 k)$-level carry-select adder, the comparisons and trade-offs between carry-select adders and carry-lookahead adders, as discussed at the end of Section 7.3, are relevant here as well.

7.5 HYBRID ADDER DESIGNS

Hybrid adders are obtained by combining elements of two or more "pure" design methods to obtain adders with higher performance, greater cost-effectiveness, lower power consumption, and so on. Since any two or more pure design methods can be combined in a variety of ways, the space of possible designs for hybrid adders is immense. This leads to a great deal of flexibility in matching the design to given requirements and constraints. It also makes the designer's search for an optimal design nontrivial. In this section, we review several possible hybrid adders as representative examples.

The one- and two-level carry-select adders of Figs. 7.9 and 7.10 are essentially hybrid adders, since the top-level $k/2$- or $k/4$-bit adders can be of any type. In fact, a common use for the carry-select scheme is in building fast adders whose width would lead to inefficient implementations with certain pure designs. For example, when 4-bit lookahead carry blocks are used, both 16-bit and 64-bit carry-lookahead adders can be synthesized quite efficiently (Fig. 6.5). A 32-bit adder, on the other hand, would require two levels of lookahead and is thus not any faster than the 64-bit adder. Using 16-bit carry-lookahead adders, plus a single carry-select level to double the width, is likely to lead to a faster 32-bit adder. The resulting adder has a hybrid carry-select/carry-lookahead design.

The reverse combination (viz., hybrid carry-lookahead/carry-select) is also possible and is in fact used quite widely. An example hybrid carry-lookahead/carry-select adder is depicted in Fig. 7.12. The small adder blocks, shown in pairs, may be based on Manchester carry chains that supply the required g and p signals to the lookahead carry generator and compute the final intermediate carries as well as the sum bits once the block carry-in signals have become known.

TABLE 7.2
Conditional-sum addition of two 16-bit numbers: The width of the block for which the sum and carry bits are known doubles with each additional level, leading to an addition time that grows as the logarithm of the word width k

Block width	Block carry-in		Block sum and block carry-out 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	c_{in}
		x	0 0 1 0 0 1 1 1 0 1 1 1 0 1 1 1 0 1	
		y	0 1 0 0 1 0 1 1 1 0 1 0 1 1 1 0 1 1	
1	0	s	0 1 1 0 1 1 0 1 1 0 1 0 1 0 1 1 0	
		c	0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0	0
	1	s	1 0 1 1 0 0 1 0 1 1 0 1 0 1 0 1	
		c	0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1	
2	0	s	0 1 1 0 1 1 0 1 0 0 1 1 0 1 1 1	
		c	0 0 0 1 0 0 1 0 1 0 0 1 0 1 0 0	
	1	s	1 0 1 1 0 0 1 0 0 1 0 0 1 0	
		c	0 0 1 1 1 1 1 1 1 1 1 1 1 1	
4	0	s	0 1 1 0 0 0 0 1 0 0 1 1 0 1 1 1	
		c	0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0	
	1	s	0 1 1 1 0 0 1 0 0 1 0 0	
		c	0 0 1 1 1 1 1 1 1 1 1 1	
8	0	s	0 1 1 1 0 0 0 1 0 1 0 0 0 1 1 1	
		c	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0	
	1	s	0 1 1 1 0 0 1 0	
		c	0 0 1 1 1 1 1 1	
16	0	s	0 1 1 1 0 0 1 0 0 0 1 0 0 0 1 1 1	
		c	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	
	1	s		
		c		

c_{out}

A wider hybrid carry-lookahead/carry-select adder will likely have a multilevel carry-lookahead network rather than a single lookahead carry generator as depicted in Fig. 7.12. If the needed block g and p signals are produced quickly, the propagation of signals in the carry-lookahead network can be completely overlapped with carry propagation in the small carry-select adders. The carry-lookahead network of Fig. 6.13 was in fact developed for use in such a hybrid scheme, with 8-bit carry-select adders based on Manchester carry chains [Lync92]. The 8-bit adders complete their computation at about the same time that the carries c_{24}, c_{32}, c_{40}, c_{48}, and c_{56} become available (Fig. 6.13). Thus, the total adder delay is only two logic levels more than that of the carry-lookahead network.

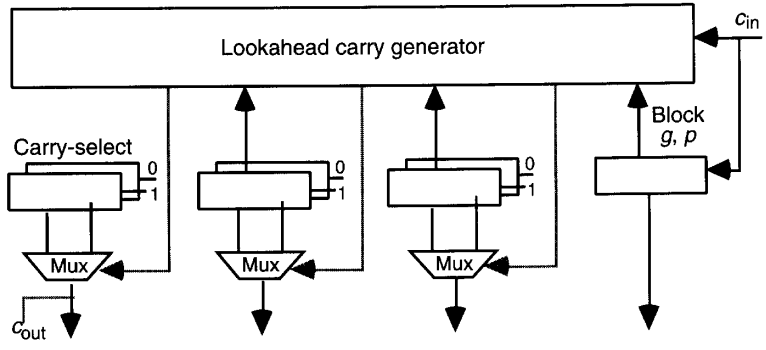


Fig. 7.12 A hybrid carry-lookahead/carry-select adder.

Another interesting hybrid design is the ripple-carry/carry-lookahead adder, an example of which is depicted in Fig. 7.13. This hybrid design is somewhat slower than pure carry-lookahead scheme, but its simplicity and greater modularity may compensate for this drawback. The analysis of cost and delay for this hybrid design relative to pure ripple-carry and carry-lookahead adders is left as an exercise, as is the development and analysis of the reverse carry-lookahead/ripple-carry hybrid combination.

Our final hybrid adder example uses the hybrid carry-lookahead/conditional-sum combination. One drawback of the conditional-sum adder for wide words is the requirement of large fan-out for the signals controlling the multiplexers at the lower levels (Fig. 7.10). This problem can be alleviated by, for example, using conditional-sum addition in smaller blocks, forming the interblock carries through carry-lookahead. For detailed description of one such adder, used in Manchester University's MU5 computer, see [Omon94, pp. 104-111].

Clearly, it is possible to combine ideas from various designs in many different ways, giving rise to a steady stream of new implementations and theoretical proposals for the design of fast adders. Different combinations become attractive with particular technologies in view of their specific cost factors and fundamental constraints [Kant93]. In addition, application requirements, such as low power consumption, may shift the balance in favor of a particular hybrid design.

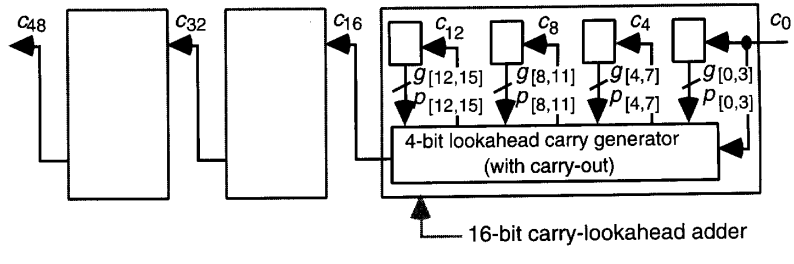


Fig. 7.13 Example 48-bit adder with hybrid ripple-carry/carry-lookahead design.

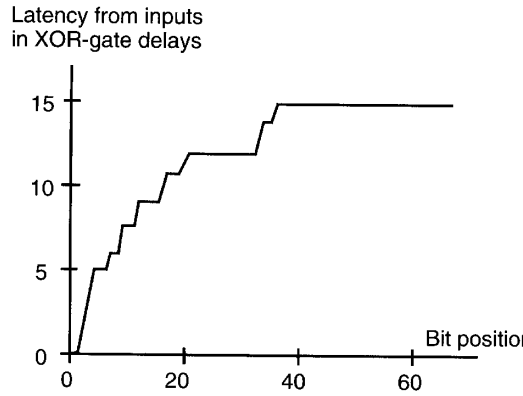


Fig. 7.14 Example arrival times for operand bits in the final fast adder of a tree multiplier [Oklo96].

7.6 OPTIMIZATIONS IN FAST ADDERS

Just as optimal carry-skip adders have variable block widths, it is often possible to reduce the delay of other (pure or hybrid) adders by optimizing the block widths. For example, depending on the implementation technology, a carry-lookahead adder with fixed blocks may not yield the lowest possible delay [Niga95]. Again, the exact optimal configuration is highly technology dependent. In fact, with modern VLSI technology, gate count alone is no longer a meaningful measure of implementation cost. Designs that minimize or regularize the interconnection may actually be more cost-effective despite using more gates. The ultimate test of cost-effectiveness for a particular hybrid design or "optimal" configuration is its actual speed and cost when implemented with the target technology.

So far our discussion of adder delay has been based on the tacit assumption that all input digits are available at the outset, or at time 0, and that all output digits are computed and taken out after worst-case carries have propagated. The other extreme, where input/output digits arrive and leave serially, leads to very simple digit-serial adder designs. In between the two extremes, there are practical situations in which different arrival times are associated with the input digits or certain output digits must be produced earlier than others.

We will later see, for example, that in multiplying two binary numbers, the partial products are reduced to two binary numbers, which are then added in a fast two-operand adder to produce the final product. The individual bits of these two numbers become available at different times in view of the differing logic path depths from primary inputs. Figure 7.14 shows a typical example for the input arrival times at various bit positions of this final fast adder. This information can be used in optimizing the adder design [Oklo96].

PROBLEMS

7.1 Optimal single-level carry-skip adders

a. Derive the optimal block width in a fixed-block carry-skip adder using the assumptions of Section 7.1, except that the carry production or assimilation delay in a block of width b is $b^2/2$ rather than b. Interpret the result.

b. Repeat part a with variable-width blocks. *Hint:* There will be several blocks of width b before the block width increases to $b + 1$.

7.2 Optimal two-level carry-skip adders For the two-level carry-skip adder of Example 7.2, Section 7.2, verify the block sizes given in Table 7.1 and draw a complete diagram for a 24-bit adder derived by pruning the design of Fig. 7.7.

7.3 Optimal variable-block carry-skip adders

a. Build optimal single-level carry-skip adders for word widths $k = 24$ and $k = 80$.

b. Repeat part a for two-level carry-skip adders.

c. Repeat part a for three-level carry-skip adders.

7.4 Carry-skip adders with given building blocks

a. Assume the availability of 4-bit and 8-bit adders with delays of 3 and 5 ns, respectively, and of 0.5-ns logic gates. Each of our building block adders provides a "propagate" signal in addition to the normal sum and carry-out signals. Design an optimal single-level carry-skip adder for 64-bit unsigned integers.

b. Repeat part a for a two-level-skip adder.

c. Would we gain any advantage by going to three levels of skip for the adder of part a?

d. Outline a procedure for designing optimal single-level carry-skip adders from adders of widths $b_1 < b_2 < \cdots < b_h$ and delays $d_1 < d_2 < \cdots < d_h$, plus logic gates of delay δ.

7.5 Fixed-block, two-level carry-skip adders Using the assumptions in our analysis of single-level carry-skip adders in Section 7.1, present an analysis for a two-level carry-skip adder in which the block widths b_1 and b_2 in levels 1 and 2, respectively, are fixed. Hence, assuming that b_1 and b_2 divide k, there are k/b_2 second-level blocks and k/b_1 first-level blocks, with each second-level block encompassing b_2/b_1 first-level blocks. Determine the optimal block widths b_1 and b_2. Note that because of the fixed block widths, skip logic must be included even for the rightmost block at each level.

7.6 Optimized multilevel carry-select adders Consider the hierarchical synthesis of a k-bit multilevel carry-select adder where in each step of the process, an i-bit adder is subdivided into smaller j-bit and $(i - j)$-bit adders.

a. At what value of i does it not make sense to further subdivide the block?

b. When the width i of a block is odd, the two blocks derived from it will have to be of different widths. Is it better to make the right-hand or the left-hand block wider?

c. Evaluate the suggestion that, just as in carry-skip adders, blocks of different widths be used to optimize the design of carry-select adders.

7.7 Design of carry-select adders Design 64-bit adders using ripple-carry blocks and 0, 1, 2, 3, or 4 levels of carry select.

a. Draw schematic diagrams for the three- and four-level carry-select adders, showing all components and selection signals.

b. Obtain the exact delay and cost for each design in terms of the number of gates and gate levels using two-input NAND gates throughout. Construct the ripple-carry blocks using the full-adder design derived from Figs. 5.2a and 5.1c.

c. Compare the five designs with regard to delay, cost, and the composite delay–cost figure of merit and discuss.

7.8 The conditional-sum addition algorithm

a. Modify Table 7.2 to correspond to the same addition, but with $c_{in} = 1$.

b. Using a tabular representation as in Table 7.2, show the steps of deriving the sum of 24-bit numbers 0001 0110 1100 1011 0100 1111 and 0010 0111 0000 0111 1011 0111 by means of the conditional-sum method.

7.9 Design of conditional-sum adders Obtain the exact delay and cost for a 64-bit conditional-sum adder in terms of the number of gates and gate levels using two-input NAND gates throughout. For the topmost level, use the design given in Fig. 7.11.

7.10 Hybrid carry completion adder Suppose we want to design a carry completion adder to take advantage of its good average-case delay but would like to improve on its $O(k)$ worst-case delay. Discuss the suitability for this purpose of each of the following hybrid designs.

a. Completion-sensing blocks used in a single-level carry-skip arrangement.

b. Completion-sensing blocks used in a single-level carry-select arrangement.

c. Ripple-carry blocks with completion-sensing skip logic (separate skip circuits for 0 and 1 carries).

7.11 Hybrid ripple-carry/carry-lookahead adders Consider the hybrid ripple-carry/carry-lookahead adder design depicted in Fig. 7.13.

a. Present a design for the modified lookahead carry generator circuit that also produces the block's carry-out (e.g., c_{16} in Fig. 7.13).

b. Develop an expression for the total delay of such an adder. State your assumptions.

c. Under what conditions, if any, is the resulting adder faster than an adder with pure carry-lookahead design?

7.12 Hybrid carry-lookahead/ripple-carry adders Consider a hybrid adder based on ripple-carry blocks connected together with carry lookahead logic (i.e., the reverse combination compared to the design in Fig. 7.13). Present an analysis for the delay of such an adder and state under what conditions, if any, the resulting design is preferable to a pure carry-lookahead adder or to the hybrid design of Fig. 7.13.

7.13 Hybrid carry-select/carry-lookahead adders Show how carry-lookahead adders can be combined by a carry-select scheme to form a k-bit adder without duplicating the carry-lookahead logic in the upper $k/2$ bits.

7.14 Building fast adders from 4-bit adders Assume the availability of fast 4-bit adders with one (two) gate delay(s) to bit (block) g and p signals and two gate delays to sum and

carry-out once the bit g and p and block carry-in signals are known. Derive the cost and delay of each of the following 16-bit adders:

a. Four 4-bit adders cascaded through their carry-in and carry-out signals.

b. Single-level carry-skip design with 4-bit skip blocks.

c. Single-level carry-skip design with 8-bit skip blocks.

d. Single-level carry-select, with each of the 8-bit adders constructed by cascading two 4-bit adders.

7.15 Carry-lookahead versus hybrid adders We want to design a 32-bit fast adder from standard building blocks such as 4-bit binary full adders, 4-bit lookahead carry circuits, and multiplexers. Compare the following adders with respect to cost and delay:

a. Adder designed with two levels of lookahead.

b. Carry-select adder built of three 16-bit single-level carry-lookahead adders.

7.16 Comparing fast two-operand adders Assume the availability of 1-bit full adders; 1-bit, two-input multiplexers, and 4-bit lookahead carry circuits as unit-delay building blocks. Draw diagrams for, and compare the speeds and costs of, the following 16-bit adder designs.

a. Optimal variable-block carry-skip adder using a multiplexer for each skip circuit.

b. Single-level carry-select adder with 8-bit ripple-carry blocks.

c. Two-level carry-select adder with 4-bit ripple-carry blocks.

d. Hybrid carry-lookahead/carry-select adder with duplicated 4-bit ripple-carry blocks in which the carry-outs with $c_{in} = 0$ and $c_{in} = 1$ are used as the group g and p signals.

7.17 Optimal adders with input timing information For each fast adder type studied in Chapters 6 and 7, discuss how the availability of input bits at different times (Fig. 7.14) could be exploited to derive faster designs.

7.18 Fractional precision addition

a. We would like to design an adder that either adds two 32-bit numbers in their entirety or their lower and upper 16-bit halves independently. For each adder design discussed in Chapters 5–7, indicate how the design can be modified to allow such parallel half-precision arithmetic.

b. Propose a hybrid adder design that is particularly efficient for the design of part a.

c. Repeat part b, this time assuming two fractional precision modes: (4×8)-bit or (2×16)-bit.

REFERENCES

[Bedr62] Bedrij, O. J., "Carry-Select Adder," *IRE Trans. Electronic Computers*, Vol. 11, pp. 340–346, 1962.

[Chan90] Chan, P. K., and M. D. F. Schlag, "Analysis and Design of CMOS Manchester Adders with Variable Carry Skip," *IEEE Trans. Computers*, Vol. 39, pp. 983–992, 1990.

[Chan92] Chan, P. K., M. D. F. Schlag, C. D. Thomborson, and V. G. Oklobdzija, "Delay Optimization of Carry-Skip Adders and Block Carry-Lookahead Adders Using Multidimensional Dynamic Programming," *IEEE Trans. Computers,* Vol. 41, No. 8, pp. 920–930, 1992.

[Guyo87] Guyot, A., and J.-M. Muller, "A Way to Build Efficient Carry-Skip Adders," *IEEE Trans. Computers*, Vol. 36, No. 10, pp. 1144–1152, 1987.

[Kant93] Kantabutra, V., "Designing Optimum One-Level Carry-Skip Adders," *IEEE Trans. Computers*, Vol. 42, No. 6, pp. 759–764, 1993.

[Lehm61] Lehman, M., and N. Burla, "Skip Techniques for High-Speed Carry Propagation in Binary Arithmetic Units," *IRE Trans. Electronic Computers*, Vol. 10, pp. 691–698, December 1961.

[Lync92] Lynch, T., and E. Swartzlander, "A Spanning Tree Carry Lookahead Adder," *IEEE Trans. Computers*, Vol. 41, No. 8, pp. 931–939, 1992.

[Maje67] Majerski, S., "On Determination of Optimal Distributions of Carry Skip in Adders," *IEEE Trans. Electronic Computers*, Vol. 16, pp. 45–58, February 1967.

[Niga95] Nigaglioni, R. H., and E. E. Swartzlander, "Variable Spanning Tree Adder," *Proc. Asilomar Conf. Signals, Systems, and Computers*, 1995, pp. 586–590.

[Oklo96] Oklobdzija, V. G., D. Villeger, and S. S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Trans. Computers*, Vol. 45, No. 3, pp. 294–306, 1996.

[Omon94] Omondi, A. R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementation*, Prentice-Hall, 1994.

[Skla60] Sklansky, J., "Conditional-Sum Addition Logic," *IRE Trans. Electronic Computers*, Vol. 9, No. 2, pp. 226–231, June 1960.

[Turr89] Turrini, S., "Optimal Group Distribution in Carry-Skip Adders," *Proc. 9th Symp. Computer Arithmetic*, pp. 96–103, September 1989.

Chapter 8 MULTIOPERAND ADDITION

In Chapters 6 and 7, we covered several speedup methods for adding two operands. Our primary motivation in dealing with multioperand addition in this chapter is that both multiplication and inner-product computation reduce to adding a set of numbers, namely, the partial products or the component products. The main idea used is that of *deferred carry assimilation* made possible by redundant representation of the intermediate results.

8.1 USING TWO-OPERAND ADDERS

Multioperand addition is implicit in both multiplication and computation of vector inner products (Fig. 8.1). In multiplying a multiplicand a by a k-digit multiplier x, the k partial products $x_i a$ must be formed and then added. For inner-product computation, the component product terms $p^{(j)} = x^{(j)} y^{(j)}$ obtained by multiplying the corresponding elements of the two operand vectors x and y, need to be added. Computing averages (e.g., in the design of a mean filter) is another application that requires multioperand addition.

Figure 8.1 uses what is known as "dot notation," a representation we will find quite useful when only the positioning or alignment of bits, rather than their values, is important. We will assume that the n operands are unsigned integers of the same width k and are aligned at the least significant end, as in the right side of Fig. 8.1. Extension of the methods to signed operands are discussed in Section 8.6. Application to multiplication is the subject of Part III.

Figure 8.2 depicts a serial solution to the multioperand addition problem using a single two-operand adder. The binary operands $x^{(i)}$, $i = 0, 1, \cdots, n-1$, are applied, one per clock cycle,

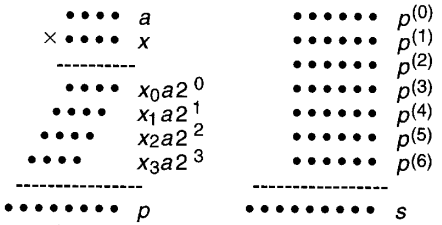


Fig. 8.1 Multioperand addition problems for multiplication or inner-product computation shown in dot notation.

to one input of the adder, with the other input fed back from a partial sum register. Since the final sum can be as large as $n(2^k - 1)$, the partial sum register must be $\log_2(n2^k - n + 1) \approx k + \log_2 n$ bits wide.

Assuming the use of a logarithmic time fast adder, the total latency of the scheme of Fig. 8.2 for adding n operands of width k is:

$$T_{\text{serial-multi-add}} = \text{O}(n \log(k + \log n))$$

Since $k + \log n$ is no less than $\max(k, \log n)$ and no greater than $max(2k, 2 \log n)$, we have $\log(k + \log n) = \text{O}(\log k + \log \log n)$ and:

$$T_{\text{serial-multi-add}} = \text{O}(n \log k + n \log \log n)$$

Therefore, the addition time grows superlinearly with n when k is fixed and logarithmically with k for a given n.

One can pipeline this serial solution to get somewhat better performance. Figure 8.3 shows that if the adder is implemented as a four-stage pipeline, then three adders can be used to achieve the maximum possible throughput of one operand per clock cycle. Even though the clock cycle is now shorter because of pipelining, the latency from the first input to the last output remains asymptotically the same with h-stage pipelining for any fixed h.

Note that the schemes shown in Figs. 8.2 and 8.3 work for any prefix computation involving a binary operator $\otimes$, provided the adder is replaced by a hardware unit corresponding to the binary operator $\otimes$. For example, similar designs can be used to find the product of n numbers or the largest value among them.

For higher speed, a tree of two-operand adders might be used, as in Fig. 8.4. Such a binary tree of two-operand adders needs $n - 1$ adders and is thus quite costly if built of fast adders.

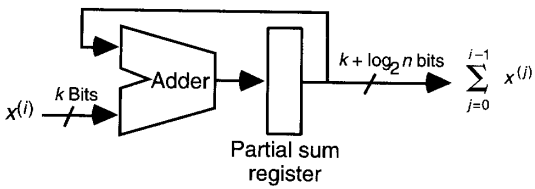


Fig. 8.2 Serial implementation of multioperand addition with a single two-operand adder.

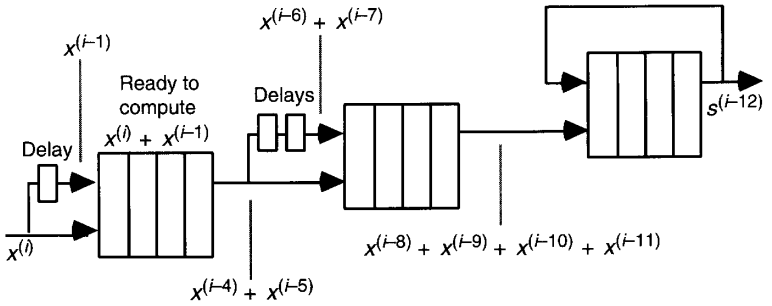


Fig. 8.3 Serial multioperand addition when each adder is a four-stage pipeline.

Strange as it may seem, the use of simple and slow ripple-carry (or even bit-serial) adders may be the best choice in this design. If we use fast logarithmic time adders, the latency will be:

$$\begin{aligned} T_{\text{tree-fast-multi-add}} &= \text{O}(\log\ k + \log(k+1) + \cdots + \log(k + \lceil \log_2 n \rceil - 1)) \\ &= \text{O}(\log\ n\ \log\ k + \log\ n \log\ \log\ n) \end{aligned}$$

The preceding equality can be proven by considering the two cases of $\log_2 n < k$ and $\log_2 n > k$ and bounding the right-hand side in each case. Supplying the needed details of the proof is left as an exercise. If we use ripple-carry adders in the tree of Fig. 8.4, the delay becomes

$$T_{\text{tree-ripple-multi-add}} = \text{O}(k + \log\ n)$$

which can be less than the delay with fast adders for large n. Comparing the costs of this and the preceding schemes for different ranges of values for the parameters k and n is left as an exercise.

Figure 8.5 shows why the delay with ripple-carry adders is $\text{O}(k + \log n)$. There are $\lceil \log_2\ n \rceil$ levels in the tree. An adder in the $(i+1)$th level need not wait for full carry propagation in level i to occur, but rather can start its addition one full-adder delay after level i. In other words, carry propagation in each level lags one time unit behind the preceding level. Thus, we need to allow constant time for all but the last adder level, which needs $\text{O}(k + \log\ n)$ time.

Can we do better than the $\text{O}(k + \log\ n)$ delay offered by the tree of ripple-carry adders of Fig. 8.5? The absolute minimum time is $\text{O}(\log(kn)) = \text{O}(\log\ k + \log\ n)$, where kn is the total number of input bits to be processed by the multioperand adder, which is ultimately composed of constant-fan-in logic gates. This minimum is achievable with carry-save adders.

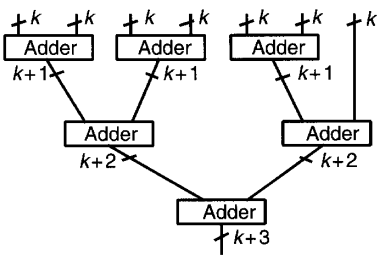


Fig. 8.4 Adding seven numbers in a binary tree of adders.

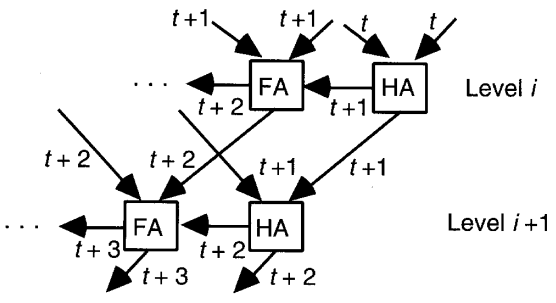


Fig. 8.5 Ripple-carry adders at levels i and $i + 1$ in the tree of full adders and half-adders (HA) used for multioperand addition.

8.2 CARRY-SAVE ADDERS

We can view a row of binary full adders as a mechanism to reduce three numbers to two numbers rather than as one to reduce two numbers to their sum. Figure 8.6 shows the relationship of a ripple-carry adder for the latter reduction and a carry-save adder for the former (see also Fig. 3.5).

Figure 8.7 presents, in dot notation, the relationship shown in Fig. 8.6. To specify more precisely how the various dots are related or obtained, we agree to enclose any three dots that form the inputs to a full adder in a dashed box and to connect the sum and carry outputs of a full-adder by a diagonal line (Fig. 8.8). Occasionally, only two dots are combined to form a sum bit and a carry bit. Then the two dots are enclosed in a dashed box and the use of a half-adder is signified by a cross line on the diagonal line connecting its outputs (Fig. 8.8).

Dot notation suggests another way to view the function of a carry-save adder: as converter of a radix-2 number with the digit set [0, 3] (three bits in one position) to one with the digit set [0, 2] (two bits in one position).

A carry-save adder tree (Fig. 8.9) can reduce n binary numbers to two numbers having the same sum in O(log n) levels. If a fast logarithmic time carry-propagate adder is then used to add

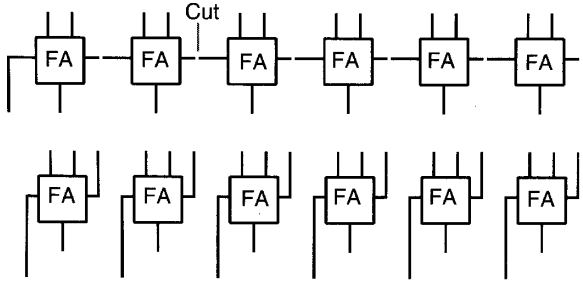


Fig. 8.6 A ripple-carry adder turns into a carry-save adder if the carries are saved (stored) rather than propagated.

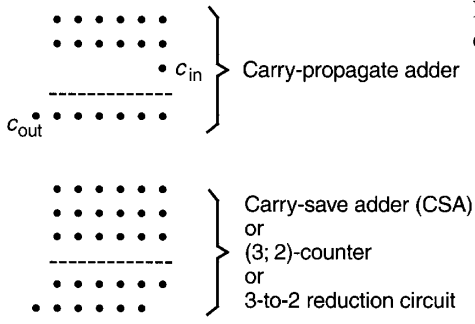


Fig. 8.7 Carry-propagate adder (CPA) and carry-save adder (CSA) functions in dot notation.

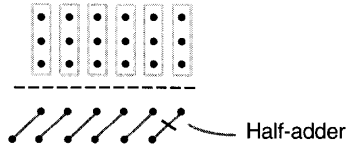


Fig. 8.8 Specifying full- and half-adder blocks, with their inputs and outputs, in dot notation.

the two resulting numbers, we have the following results for the cost and delay of n-operand addition:

$$C_{\text{carry-save-multi-add}} = (n-1)C_{\text{CSA}} + C_{\text{CPA}}$$
$$T_{\text{carry-save-multi-add}} = \text{O(tree height } + T_{\text{CPA}}) = \text{O}(\log\ n + \log\ k)$$

The needed CSAs are of various widths, but generally the widths are close to k bits; the CPA is of width at most $k + \log_2 n$.

An example for adding seven 6-bit numbers is shown in Fig. 8.10. A more compact tabular representation of the same process is depicted in Fig. 8.11, where the entries represent the number of dots remaining in the respective columns or bit positions. We begin on the first row with seven dots in each of bit positions 0–5; these dots represent the seven 6-bit inputs. Two full-adders are used in each 7-dot column, with each FA converting 3 dots in its column to one dot in that column and one dot in the next higher column. This leads to the distribution of dots shown on the second row of Fig. 8.11. Next, one full adder is used in each of the bit positions 0–5 containing

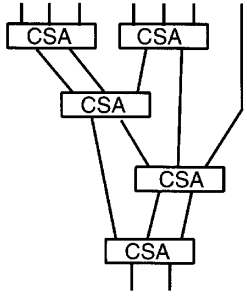


Fig. 8.9 Tree of carry-save adders reducing seven numbers to two.

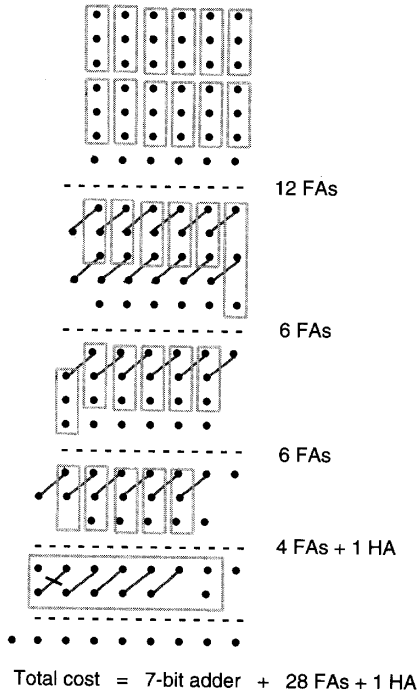


Fig. 8.10 Addition of seven 6-bit numbers in dot notation.

3 dots or more, and so on, until no column contains more than 2 dots (see below for details). At this point, a carry-propagate adder is used to reduce the resulting two numbers to the final 9-bit sum represented by a single dot in each of the bit positions 0–8.

In deriving the entries of a row from those of the preceding one, we begin with column 0 and proceed to the leftmost column. In each column, we cast out multiples of 3 and for each group of three that we cast out, we include 1 bit in the same column and 1 bit in the next column to the left. Columns at the right that have already been reduced to 1 need no further reduction. The rightmost column with a 2 can be either reduced using a half-adder or left intact, postponing its reduction to the final carry-propagate adder. The former strategy tends to make the width of the

8	7	6	5	4	3	2	1	0	Bit position
			7	7	7	7	7	7	6 × 2 = 12 FAs
		2	5	5	5	5	5	3	6 FAs
		3	4	4	4	4	4	1	6 FAs
	1	2	3	3	3	3	2	1	4 FAs + 1 HA
	2	2	2	2	2	1	2	1	7-bit adder
			Carry-propagate adder						
1	1	1	1	1	1	1	1	1	

Fig. 8.11 Representing a seven-operand addition in tabular form.

final CPA smaller, while the latter strategy minimizes the number of full and half-adders at the expense of a wider CPA. In the example of Fig. 8.10, and its tabular form in Fig. 8.11, we could have reduced the width of the final CPA from 7 bits to 6 bits by applying an extra half-adder to the two dots remaining in bit position 1.

Figure 8.12 depicts a block diagram for the carry-save addition of seven k-bit numbers. By tagging each line in the diagram with the bit positions it carries, we see that even though the partial sums do grow in magnitude as more numbers are combined, the widths of the carry-save adders stay pretty much constant throughout the tree. Note that the lowermost CSA in Fig. 8.12 could have been made only $k - 1$ bits wide by letting the two lines in bit position 1 pass through. The carry-propagate adder would then have become $k + 1$ bits wide.

Of course carry-save addition can be implemented serially using a single CSA, as depicted in Fig. 8.13. This is the preferred method when the operands arrive serially or must be read out from memory one by one. Note, however, that in this case both the CSA and final CPA will have to be wider.

8.3 WALLACE AND DADDA TREES

The CSA tree of Fig. 8.12, which reduces seven k-bit operands to two $(k + 2)$-bit operands having the same sum, is known as a seven-input Wallace tree. More generally, an n-input Wallace tree reduces its k-bit inputs to two $(k + \log_2 n - 1)$-bit outputs. Since each CSA reduces the number of operands by a factor of 1.5, the smallest height $h(n)$ of an n-input Wallace tree satisfies the following recurrence:

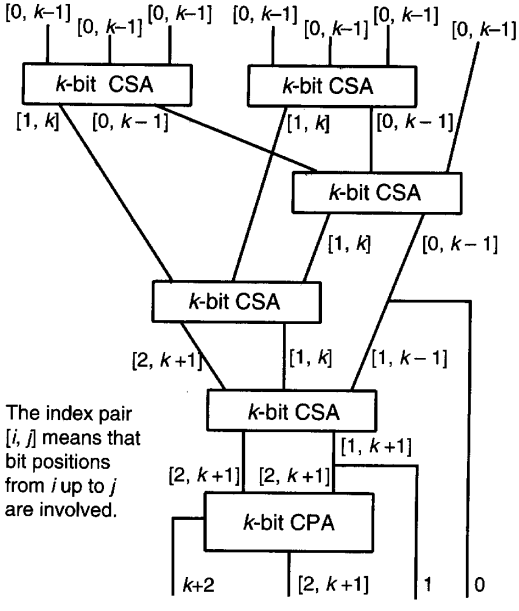


Fig. 8.12 Adding seven k-bit numbers and the CSA/CPA widths required.

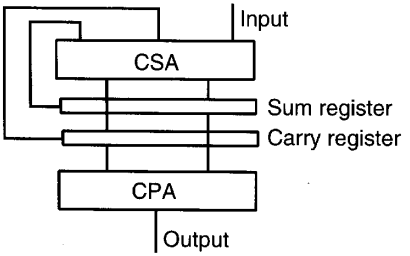


Fig. 8.13 Serial carry-save addition by means of a single CSA.

$$h(n) = 1 + h(\lceil 2n/3 \rceil)$$

Applying this recurrence provides an exact value for the height of an n-input Wallace tree. If we ignore the ceiling operator in the preceding equation and write it as $h(n) = 1 + h(2n/3)$, we obtain a lower bound for the height, $h(n) \geq \log_{1.5}(n/2)$, where equality occurs only for $n = 2, 3$. Another way to look at the above relationship between the number of inputs and the tree height is to find the maximum number of inputs $n(h)$ that can be reduced to two outputs by an h-level tree. The recurrence for $n(h)$ is:

$$n(h) = \lfloor 3n(h-1)/2 \rfloor$$

Again ignoring the floor operator, we obtain the upper bound $n(h) \leq 2(3/2)^h$. The lower bound $n(h) > 2(3/2)^{h-1}$ is also easily established. The exact value of $n(h)$ for $0 \leq h \leq 20$ is given in Table 8.1.

In Wallace trees, we reduce the number of operands at the earliest opportunity (see the example in Fig. 8.10). In other words, if there are m dots in a column, we immediately apply $\lfloor m/3 \rfloor$ full adders to that column. This tends to minimize the overall delay by making the final CPA as short as possible.

However, the delay of a fast adder is usually not a smoothly increasing function of the word width. A carry-lookahead adder, for example, may have essentially the same delay for word widths 17–32 bits. In Dadda trees, we reduce the number of operands to the next lower number in Table 8.1 using the fewest FAs and HAs possible. The justification is that 7, 8, or 9 operands, say, require four CSA levels; thus there is no point in reducing the number of operands below the next lower $n(h)$ value in the table, since this would not lead to a faster tree.

Let us redo the example of Fig. 8.10 by means of Dadda's strategy. Figure 8.14 shows the result. We start with seven rows of dots, so our first task is to reduce the number of rows to

TABLE 8.1
The maximum number $n(h)$ of inputs for an h-level carry-save-adder tree

h	$n(h)$	h	$n(h)$	h	$n(h)$
0	2	7	28	14	474
1	3	8	42	15	711
2	4	9	63	16	1066
3	6	10	94	17	1599
4	9	11	141	18	2398
5	13	12	211	19	3597
6	19	13	316	20	5395

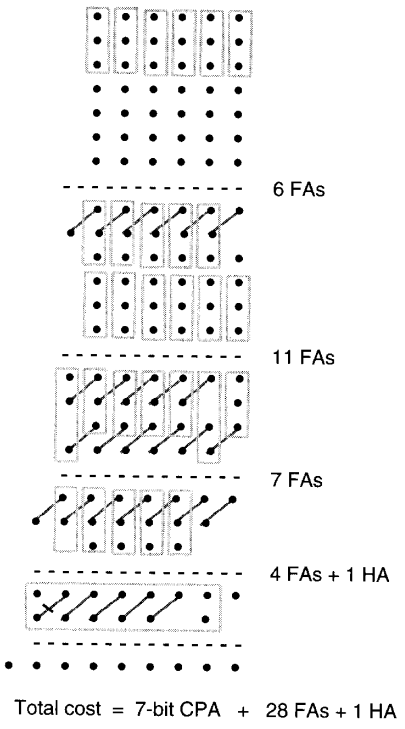


Fig. 8.14 Using Dadda's strategy to add seven 6-bit numbers

the next lower $n(h)$ value (i.e., 6). This can be done by using 6 full adders; next, we aim for four rows, leading to the use of 11 FAs, and so on. In this particular example, the Wallace and Dadda approaches result in the same number of full and half-adders and the same width for the CPA. Again, the CPA width could have been reduced to 6 bits by using an extra half-adder in bit position 1.

Since a CPA has a carry-in signal that can be used to accommodate one of the dots, it is sometimes possible to reduce the complexity of the CSA tree by leaving three dots in the least significant position of the adder. Figure 8.15 shows the same example as in Figs. 8.10 and 8.14, but with two FAs replaced with HAs, leaving an extra dot in each of the bit positions 1 and 2.

8.4 PARALLEL COUNTERS

A single-bit full adder is sometimes referred to as a (3; 2)-counter, meaning that it counts the number of 1s among its three input bits and represents the result as a 2-bit number. This can be easily generalized: an $(n; \lceil \log_2(n+1) \rceil)$-counter has n inputs and produces a $\lceil \log_2(n+1) \rceil$-bit binary output representing the number of 1s among its n inputs. Such a circuit is also known as an n-input parallel counter.

A 10-input parallel counter, or a (10; 4)-counter, is depicted in Fig. 8.16 in terms of both dot notation and circuit diagram with full and half-adders. A row of such (10; 4)-counters, one per bit position, can reduce a set of 10 binary numbers to 4 binary numbers. The dot notation

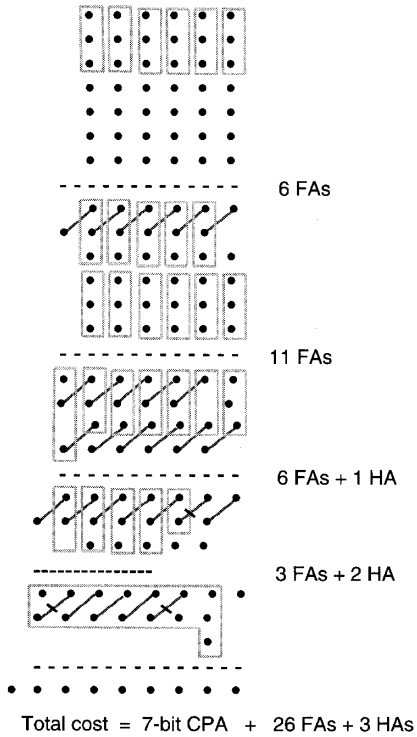


Fig. 8.15 Adding seven 6-bit numbers by taking advantage of the final adder's carry-in.

representation of this reduction is similar to that of (3; 2)-counters, except that each diagonal line connecting the outputs of a (10; 4) counter will go through four dots. A (7; 3)-counter can be similarly designed.

Even though a circuit that counts the number of 1s among n inputs is known as a parallel counter, we note that this does not constitute a true generalization of the notion of a sequential counter. A sequential counter receives a single bit (the count signal) and adds it to a stored count. A parallel counter, then, could have been defined as a circuit that receives n count signals and adds them to a stored count, thus in effect incrementing the count by the sum of the input count signals. Such a circuit has been called an "accumulative parallel counter" [Parh95]. An accumulative parallel counter can be built from a parallel incrementer (a combinational circuit receiving a number and producing the sum of the input number and n count signals at the output) along with a storage register. Both parallel and accumulative parallel counters can be extended by considering signed count signals. These would constitute generalizations of sequential up/down counters [Parh89].

8.5 GENERALIZED PARALLEL COUNTERS

A parallel counter reduces a number of dots in the same bit position into dots in different positions (one in each). This idea can be easily generalized to circuits that receive "dot patterns"

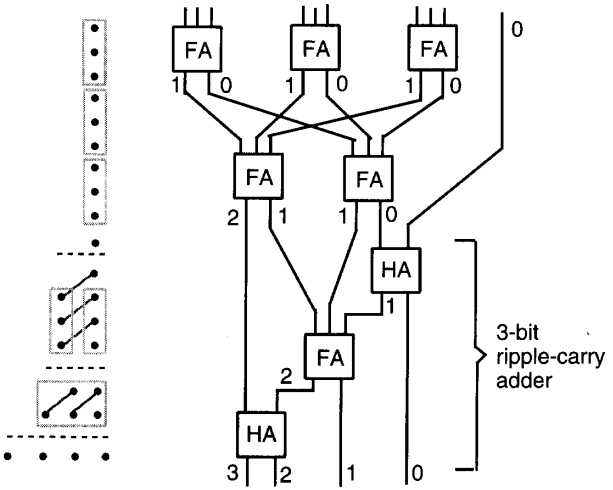


Fig. 8.16 A 10-input parallel counter also known as a (10; 4)-counter.

(not necessarily in a single column) and convert them to other dot patterns (not necessarily one in each column). If the output dot pattern has fewer dots that the input dot pattern, compression takes place; repeated use of such circuits can eventually lead to the reduction of n numbers to a small set of numbers (ideally two).

A generalized parallel counter (parallel compressor) is characterized by the number of dots in each input column and in each output column. We do not consider such circuits in their full generality but limit ourselves to those that output a single dot in each column. Thus, the output side of such parallel compressors is again characterized by a single integer representing the number of columns spanned by the output. The input side is characterized by a sequence of integers corresponding to the number of inputs in various columns.

For example, a (4, 4; 4)-counter receives 4 bits in each of two adjacent columns and produces a 4-bit number representing the sum of the four 2-bit numbers received. Similarly, a (5, 5; 4)-counter, depicted in Fig. 8.17, reduces five 2-bit numbers to a 4-bit number. The numbers of input dots in various columns do not have to be the same. For example, a (4, 6; 4)-counter receives 6 bits of weight 1 and 4 bits of weight 2 and delivers their weighted sum in the form of a 4-bit binary number. For a counter of this type to be feasible, the sum of the output weights must equal or exceed the sum of its input weights.

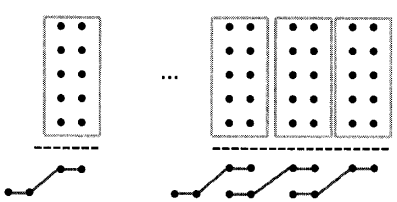


Fig. 8.17 Dot notation for a (5, 5; 4)-counter and the use of such counters for reducing five numbers to two numbers.

Generalized parallel counters are quite powerful. For example, a 4-bit binary full adder is really a (2, 2, 2, 3; 5)-counter.

Since our goal in multioperand carry-save addition is to reduce n numbers to two numbers, we sometimes talk of $(n; 2)$-counters, even though, with our preceding definition, this does not make sense for $n > 3$. By an $(n; 2)$-counter, $n > 3$, we usually mean a slice of a circuit that helps us reduce n numbers to two numbers when suitably replicated. Slice i of the circuit receives n input bits in position i, plus transfer or "carry" bits from one or more positions to the right ($i - 1, i - 2$, etc.), and produces output bits in the two positions i and $i + 1$ plus transfer bits into one or more higher positions ($i + 1, i + 2$, etc.). If ψ_j denotes the number of transfer bits from slice i to slice $i + j$, the fundamental inequality to be satisfied for this scheme to work is

$$n + \psi_1 + \psi_2 + \psi_3 + \cdots \leq 3 + 2\psi_1 + 4\psi_2 + 8\psi_3 + \cdots$$

where 3 represents the maximum value of the 2 output bits. For example, a (7; 2)-counter can be built by allowing $\psi_1 = 1$ transfer bit from position i to position $i + 1$ and $\psi_2 = 1$ transfer bit into position $i + 2$. For maximum speed, the circuit slice must be designed in such a way that transfer signals are introduced as close to the circuit's outputs as possible, to prevent the transfers from rippling through many stages. Design of a (7; 2)-counter using these principles is left as an exercise.

8.6 ADDING MULTIPLE SIGNED NUMBERS

When the operands to be added are 2's-complement numbers, they must be sign-extended to the width of the final result if multiple-operand addition is to yield their correct sum. The example in Fig. 8.18 shows extension of the sign bits x_{k-1}, y_{k-1}, and z_{k-1} across five extra positions.

It appears, therefore, that sign extension may dramatically increase the complexity of the CSA tree used for n-operand addition when n is large. However, since the sign extension bits are identical, a single full adder can do the job of several full adders that would be receiving identical inputs if used. With this hardware-sharing scheme, the CSA widths are only marginally increased. For the three operands in Fig. 8.18a, a single (3; 2)-counter can be used in lieu of six that would be receiving the same input bits x_{k-1}, y_{k-1}, and z_{k-1}.

It is possible to avoid sign extension by taking advantage of the negative-weight interpretation of the sign bit in 2's-complement representation. A negative sign bit $-x_{k-1}$ can be replaced by $1 - x_{k-1} = \bar{x}_{k-1}$ (the complement of x_{k-1}), with the extra 1 canceled by inserting a -1 in that same column. Multiple -1s in a given column can be paired, with each pair replaced by a -1 in the next higher column. Finally, a solitary -1 in a given column is replaced by 1 in that column and -1 in the next higher column. Eventually, all the -1s disappear off the left end and at most a single extra 1 is left in some of the columns.

Figure 8.18b shows how this method is applied when adding three 2's-complement numbers. The three sign bits are complemented and three -1s are inserted in the sign position. These three -1s are then replaced by a 1 in the sign position and two -1s in the next higher position (k). These two -1s are then removed and, instead, a single -1 is inserted in position $k + 1$. The latter -1 is in turn replaced by a 1 in position $k + 1$ and a -1 in position $k + 2$, and so on. The -1 that moves out from the leftmost position is immaterial in view of $(k + 5)$-bit 2's-complement arithmetic being performed modulo-2^{k+5}.

Extended positions	Sign	Magnitude positions
$x_{k-1}\ x_{k-1}\ x_{k-1}\ x_{k-1}\ x_{k-1}$	x_{k-1}	$x_{k-2}\ x_{k-3}\ x_{k-4}\cdots$
$y_{k-1}\ y_{k-1}\ y_{k-1}\ y_{k-1}\ y_{k-1}$	y_{k-1}	$y_{k-2}\ y_{k-3}\ y_{k-4}\cdots$
$z_{k-1}\ z_{k-1}\ z_{k-1}\ z_{k-1}\ z_{k-1}$	z_{k-1}	$z_{k-2}\ z_{k-3}\ z_{k-4}\cdots$

(a)

Extended positions	Sign	Magnitude positions
1 1 1 1 0	$\overline{x}_{k-1}$	$x_{k-2}\ x_{k-3}\ x_{k-4}\cdots$
	$\overline{y}_{k-1}$	$y_{k-2}\ y_{k-3}\ y_{k-4}\cdots$
	$\overline{z}_{k-1}$	$z_{k-2}\ z_{k-3}\ z_{k-4}\cdots$
	1	

(b)

Fig. 8.18 Adding three 2's-complement numbers by means of sign extension (**a**) and by the method based on negatively weighted sign bits (**b**).

PROBLEMS

8.1 Pipelined multioperand addition

a. Present a design similar to Fig. 8.3 for adding a set of n input numbers, with a throughput of one input per clock cycle, if each adder block is a two-stage pipeline.

b. Repeat part a for a pipelined adder with eight stages.

c. Discuss methods for using the pipelined multi-operand addition scheme of Fig. 8.3 when the number of pipeline stages in an adder block is not a power of 2. Apply your method to the case of an adder with five pipeline stages.

8.2 Multioperand addition with two-operand adders Consider all the methods discussed in Section 8.1 for adding n unsigned integers of width k using two-operand adders.

a. Using reasonable assumptions, derive exact, as opposed to asymptotic, expressions for the delay and cost of each method.

b. On a two-dimensional coordinate system, with the axes representing n and k, identify the regions where each method is best in terms of speed.

c. Repeat part b, this time using delay $\times$ cost as the figure of merit for comparison.

8.3 Comparing multioperand addition schemes Consider the problem of adding n unsigned integers of width k.

a. Identify two methods whose delays are $O(\log\ k + n)$ and $O(k + \log\ n)$.

b. On a two-dimensional coordinate system, with logarithmic scales for both n and k, identify the regions in which one design is faster than the other. Describe your assumptions about implementations.

c. Repeat part b, this time comparing cost-effectiveness rather than just speed.

8.4 Building blocks for multioperand addition A carry-save adder reduces three binary numbers to two binary numbers. It costs c units and performs its function with a time

delay d. An "alternative reduction adder" (ARA) reduces five binary numbers to two binary numbers. It costs $3c$ units and has a delay of $2d$.

a. Which of the two elements, CSA or ARA, is more cost-effective for designing a tree that reduces 32 operands to 2 operands if used as the only building block? Ignore the widths of the CSA and ARA blocks and focus only on their numbers.

b. Propose an efficient design for 32-to-2 reduction if both CSA and ARA building blocks are allowed.

8.5 **Carry-save adder trees** Consider the problem of adding eight 8-bit unsigned binary numbers.

a. Using tabular representation, show the design of a Wallace tree for reducing the 8 operands to two operands.

b. Repeat part a for a Dadda tree.

c. Compare the two designs with respect to speed and cost.

8.6 **Carry-save adder trees** We have seen that the maximum number of operands that can be combined using an h-level tree of CSAs is $n(h) = \lfloor 3n(h-1)/2 \rfloor$.

a. Prove the inequality $n(h) \geq 2n(h-2)$.

b. Prove the inequality $n(h) \geq 3n(h-3)$.

c. Show that both bounds of parts a and b are tight by providing one example in which equality holds.

d. Prove the inequality $n(h) \geq n(h-a)\lfloor n(a)/2 \rfloor$ for $a \geq 0$. *Hint:* Think of the h-level tree as the top $h-a$ levels followed by an a-level tree and consider the lines connecting the two parts.

8.7 **A three-operand addition problem** Effective 24-bit addresses in the IBM System 370 family of computers were computed by adding three unsigned values: two 24-bit numbers and a 12-bit number. Since address computation was needed for each instruction, speed was critical and using two addition steps wouldn't do, particularly for the faster computers in the family.

a. Suggest a fast addition scheme for this address computation. Your design should produce an "address invalid" signal when there is an overflow.

b. Extend your design so that it also indicates if the computed address is in the range $[0, u]$, where u is a given upper bound (an input to the circuit).

8.8 **Parallel counters** Design a 255-input parallel counter using (7; 3)-counters and 4-bit binary adders as the only building blocks.

8.9 **Parallel counters** Consider the synthesis of an n-input parallel counter.

a. Prove that $n - \log_2 n$ is a lower bound on the number of full adders needed.

b. Show that n full adders suffice for this task. *Hint*: Think in terms of how many full adders might be used as half-adders in the worst case.

c. Prove that $\log_2 n + \log_3 n - 1$ is a lower bound on the number of full-adder levels required. *Hint:* First consider the problem of determining the least significant output bit, or actually, that of reducing the weight-2^0 column to 3 bits.

8.10 Generalized parallel counters Consider a (1, 4, 3; 4) generalized parallel counter.

a. Design the generalized parallel counter using only full-adder blocks.

b. Show how this generalized parallel counter can be used as a 3-bit binary adder.

c. Use three such parallel counters to reduce five 5-bit unsigned binary numbers into three 6-bit numbers.

d. Show how such counters can be used for 4-to-2 reduction.

8.11 Generalized parallel counters

a. Is a (3, 1; 3)-counter useful? Why (not)?

b. Design a (3, 3; 4)-counter using (3; 2)-counters as the only building blocks.

c. Use the counters of part b, and a 12-bit adder, to build a 6×6 unsigned multiplier.

d. Viewing a 4-bit binary adder as a (2, 2, 2, 3; 5)-counter and using dot notation, design a circuit to add five 6-bit binary numbers using only 4-bit adders as your building blocks.

8.12 Generalized parallel counters We want to design a slice of a (7; 2)-counter as discussed at the end of Section 8.5.

a. Present a design for slice i based on $\psi_1 = 1$ transfer bit from position $i - 1$ along with $\psi_2 = 1$ transfer bit from position $i - 2$.

b. Repeat part a with $\psi_1 = 4$ transfer bits from position $i - 1$ and $\psi_2 = 0$.

c. Compare the designs of parts a and b with respect to speed and cost.

8.13 Generalized parallel counters We have seen that a set of $k/2$ (5, 5; 4)-counters can be used to reduce five k-bit operands to two operands. *Hint:* This is possible because the 4-bit outputs of adjacent counters overlap in 2 bits, making the height of the output dot matrix equal to 2.

a. What kind of generalized parallel counter is needed to reduce seven operands to two operands?

b. Repeat part a for reducing nine operands.

c. Repeat part a for the general case of n operands, obtaining the relevant counter parameters as functions of n.

8.14 Accumulative parallel counters Design a 12-bit, 50-input accumulative parallel counter. The counter has a 12-bit register in which the accumulated count is kept. When the "count" signal goes high, the input count (a number between 0 and 50) is added to the stored count. Try to make your design as fast as possible. Ignore overflow (i.e., assume modulo-2^{12} operation). *Hint:* A 50-input parallel counter followed by a 12-bit adder isn't the best design.

8.15 Unsigned versus signed multioperand addition We want to add four 4-bit binary numbers.

a. Construct the needed circuit, assuming unsigned operands.

b. Repeat part a, assuming sign-extended 2's-complement operands.

c. Repeat part a, using the negative-weight interpretation of the sign bits.

d. Compare the three designs with respect to speed and cost.

8.16 Adding multiple signed numbers

a. Present the design of a multioperand adder for computing the 9-bit sum of sixteen 6-bit, 2's-complement numbers based on the use of negatively weighted sign bits, as described at the end of Section 8.6.

b. Redo the design using straightforward sign extension.

c. Compare the designs of parts a and b with respect to speed and cost and discuss.

8.17 Ternary parallel counters In balanced ternary representation (viz., $r = 3$ and digit set $[-1, 1]$), (4; 2)-counters can be designed [De94]. Choose a suitable encoding for the three digit values and present the complete logic design of such a (4; 2)-counter.

REFERENCES

[Dadd65] Dadda, L., "Some Schemes for Parallel Multipliers," *Alta Frequenza*, Vol. 34, pp. 349–356, 1965.

[Dadd76] Dadda, L., "On Parallel Digital Multipliers," *Alta Frequenza*, Vol. 45, pp. 574–580, 1976.

[De94] De, M., and B. P. Sinha, "Fast Parallel Algorithm for Ternary Multiplication Using Multivalued I^2L Technology," *IEEE Trans. Computers*, Vol. 43, No. 5, pp. 603–607, 1994.

[Fost71] Foster, C. C., and F. D. Stockton, "Counting Responders in an Associative Memory," *IEEE Trans. Computers*, Vol. 20, pp. 1580–1583, 1971.

[Parh89] Parhami, B., "Parallel Counters for Signed Binary Signals," *Proc. 23rd Asilomar Conf. Signals, Systems, and Computers*, pp. 513–516, 1989.

[Parh95] Parhami, B., and C.-H. Yeh, "Accumulative Parallel Counters," *Proc. 29th Asilomar Conf. Signals, Systems, and Computers*, pp. 966–970, 1995.

[Swar73] Swartzlander, E. E., "Parallel Counters," *IEEE Trans. Computers*, Vol. 22, No. 11, pp. 1021–1024, 1973.

[Wall64] Wallace, C. S., "A Suggestion for a Fast Multiplier," *IEEE Trans. Electronic Computers*, Vol. 13, pp. 14–17, 1964.

PART III MULTIPLICATION

Multiplication, often realized by k cycles of shifting and adding, is a heavily used arithmetic operation that figures prominently in signal processing and scientific applications. In this part, after examining shift/add multiplication schemes and their various implementations, we note that there are but two ways to speed up the underlying multioperand addition: reducing the number of operands to be added leads to high-radix multipliers, and devising hardware multioperand adders that minimize the latency and/or maximize the throughput leads to tree and array multipliers. Of course, speed is not the only criterion of interest. Cost, VLSI area, and pin limitations favor bit-serial designs, while the desire to use available building blocks leads to designs based on additive multiply modules. Finally, the special case of squaring is of interest as it leads to considerable simplification. This part consists of the following four chapters:

Chapter 9 BASIC MULTIPLICATION SCHEMES

The multioperand addition process needed for multiplying two k-bit operands can be realized in k cycles of shifting and adding, with hardware, firmware, or software control of the loop. In this chapter, we review such economical, but slow, bit-at-a-time designs and set the stage for speedup methods and variations to be presented in Chapters 10–12. We also consider the special case of multiplication by a constant. Chapter topics include:

9.1 SHIFT/ADD MULTIPLICATION ALGORITHMS

The following notation is used in our discussion of multiplication algorithms:

a	Multiplicand	$a_{k-1}a_{k-2}\cdots a_1a_0$
x	Multiplier	$x_{k-1}x_{k-2}\cdots x_1x_0$
p	Product ($a \times x$)	$p_{2k-1}p_{2k-2}\cdots p_1p_0$

Figure 9.1 shows the multiplication of two 4-bit unsigned binary numbers in dot notation. The two numbers a and x are shown at the top. Each of the following four rows of dots corresponds to the product of the multiplicand a and a single bit of the multiplier x, with each dot representing the product (logical AND) of two bits. Since x_j is in {0, 1}, each term x_ja is either 0 or a. Thus, the problem of binary multiplication reduces to adding a set of numbers, each of which is 0 or a shifted version of the multiplicand a.

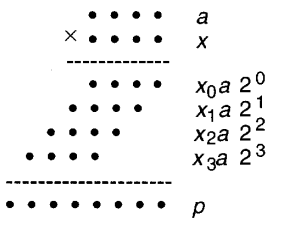


Fig. 9.1 Multiplication of two 4-bit unsigned binary numbers in dot notation.

Figure 9.1 also applies to nonbinary multiplication, except that with $r > 2$, computing the terms $x_j a$ becomes more difficult and the resulting numbers will be one digit wider than a. The rest of the process (multioperand addition), however, remains substantially the same.

Sequential or bit-at-a-time multiplication can be done by keeping a cumulative partial product (initialized to 0) and successively adding to it the properly shifted terms $x_j a$. Since each successive number to be added to the cumulative partial product is shifted by one bit with respect to the preceding one, a simpler approach is to shift the cumulative partial product by one bit in order to align its bits with those of the next partial product. Two versions of this algorithm can be devised, depending on whether the partial product terms $x_j a$ in Fig. 9.1 are processed from top to bottom or from bottom to top.

In multiplication with right shifts, the partial product terms $x_j a$ are accumulated from top to bottom:

$$p^{(j+1)} = (p^{(j)} + x_j a\, 2^k)2^{-1} \quad \text{with } p^{(0)} = 0 \text{ and } p^{(k)} = p$$

$$\text{|—— add ——|}$$

$$\text{|— shift right —|}$$

Because the right shifts will cause the first partial product to be multiplied by 2^{-k} by the time we are done, we premultiply a by 2^k to offset the effect of the right shifts. This premultiplication is done simply by aligning a with the upper half of the $2k$-bit cumulative partial product in the addition steps (i.e., storing a in the left half of a double-length register).

After k iterations, the preceding recurrence leads to:

$$p^{(k)} = ax + p^{(0)}2^{-k}$$

Thus if instead of 0, $p^{(0)}$ is initialized to $y2^k$ the expression $ax + y$ will be evaluated. This multiply-add operation is quite useful for many applications and is performed at essentially no extra cost compared to plain shift/add multiplication.

In multiplication with left shifts, the terms $x_j a$ are added up from bottom to top:

$$p^{(j+1)} = 2p^{(j)} + x_{k-j-1}a \quad \text{with} \quad p^{(0)} = 0 \text{ and } p^{(k)} = p$$

$$\text{|shift|}$$

$$\text{|left|}$$

$$\text{|—— add ——|}$$

After k iterations, the preceding recurrence leads to:

$$p^{(k)} = ax + p^{(0)}2^{k}$$

In this case, the expression $ax + y$ will be evaluated if we initialize $p^{(0)}$ to $y2^{-k}$.

Right-shift algorithm

a		1	0	1	0			
x		1	0	1	1			
$p^{(0)}$		0	0	0	0			
$+x_0a$		1	0	1	0			
$2p^{(1)}$	0	1	0	1	0			
$p^{(1)}$		0	1	0	1	0		
$+x_1a$		1	0	1	0			
$2p^{(2)}$	0	1	1	1	1	0		
$p^{(2)}$		0	1	1	1	1	0	
$+x_2a$		0	0	0	0			
$2p^{(3)}$	0	0	1	1	1	1	0	
$p^{(3)}$		0	0	1	1	1	1	0
$+x_3a$		1	0	1	0			
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

Left-shift algorithm

a				1	0	1	0	
x				1	0	1	1	
$p^{(0)}$				0	0	0	0	
$2p^{(0)}$			0	0	0	0	0	
$+x_3a$				1	0	1	0	
$p^{(1)}$			0	1	0	1	0	
$2p^{(1)}$		0	1	0	1	0	0	
$+x_2a$				0	0	0	0	
$p^{(2)}$			0	1	0	1	0	0
$2p^{(2)}$		0	1	0	1	0	0	0
$+x_1a$				1	0	1	0	
$p^{(3)}$			0	1	1	0	0	1	0
$2p^{(3)}$	0	1	1	0	0	1	0	0
$+x_0a$					1	0	1	0
$p^{(4)}$	0	1	1	0	1	1	1	0

Fig. 9.2 Examples of sequential multiplication with right and left shifts.

Figure 9.2 shows the multiplication of $a = (10)_{\text{ten}} = (1010)_{\text{two}}$ and $x = (11)_{\text{ten}} = (1011)_{\text{two}}$, to obtain their product $p = (110)_{\text{ten}} = (0110\ 1110)_{\text{two}}$, using both the right- and left-shift algorithms.

From the examples in Fig. 9.2, we see that the two algorithms are quite similar. Each algorithm entails k additions and k shifts; however, additions in the left-shift algorithm are $2k$ bits wide (the carry produced from the lower k bits may affect the upper k bits), whereas the right-shift algorithm requires k-bit additions. For this reason, multiplication with right shifts is preferable.

9.2 PROGRAMMED MULTIPLICATION

On a processor that does not have a multiply instruction, one can use shift and add instructions to perform integer multiplication. Figure 9.3 shows the structure of the needed program for the right-shift algorithm. The instructions used in this program fragment are typical of instructions available on many processors.

Ignoring operand load and result store instructions (which would be needed in any case), the function of a multiply instruction is accomplished by executing between $6k+3$ and $7k+3$ machine instructions, depending on the multiplier. For 32-bit operands, this means 200^+ instructions on the average. The situation improves somewhat if a special instruction that does some or all of the required functions within the multiplication loop is available. However, even then, no fewer than 32 instructions are executed in the multiplication loop. We thus see the importance of hardware multipliers for applications that involve a great deal of numerical computations.

Processors with microprogrammed control and no hardware multiplier essentially use a microroutine very similar to the program in Fig. 9.3 to effect multiplication. Since microinstructions typically contain some parallelism and built-in conditional branching, the number of microinstructions in the main loop is likely to be smaller than 6. This reduction, along with the

```
{Multiply, using right shifts, unsigned m_cand and m_ier,
storing the resultant 2k-bit product in p_high and p_low.
Registers: R0 holds 0       Rc for counter
           Ra for m_cand    Rx for m_ier
           Rp for p_high    Rq for p_low}

{Load operands into registers Ra and Rx}

  mult:  load      Ra with m_cand
         load      Rx with m_ier

{Initialize partial product and counter}

         copy      R0 into Rp
         copy      R0 into Rq
         load      k into Rc

{Begin multiplication loop}

 m_loop: shift     Rx right 1          {LSB moves to carry flag}
         branch    no_add if carry = 0
         add       Ra to Rp            {carry flag is set to c_out}
no_add:  rotate    Rp right 1          {carry to MSB, LSB to carry}
         rotate    Rq right 1          {carry to MSB, LSB to carry}
         decr      Rc                  {decrement counter by 1}
         branch    m_loop if Rc ≠ 0

{Store the product}

         store     Rp into p_high
         store     Rq into p_low
 m_done: ...
```

Fig. 9.3 Programmed multiplication using the right-shift algorithm.

savings in machine instruction fetching and decoding times, makes multiplication microroutines significantly faster than their machine-language counterparts, though still slower than hardwired implementations we examine next.

9.3 BASIC HARDWARE MULTIPLIERS

Hardware realization of the multiplication algorithm with right shifts is depicted in Fig. 9.4. The multiplier x and the cumulative partial product p are stored in shift registers. The next bit of the multiplier to be considered is always available at the right end of the x register and is used to select 0 or a for the addition. Addition and shifting can be performed in two separate cycles or in two subcycles within the same clock cycle. In either case, temporary storage for the adder's carry-out signal is needed. Alternatively, shifting can be performed by connecting the ith sum output of the adder to the $(k+i-1)$th bit of the partial product register and the adder's carry-out to bit $2k-1$, thus doing the addition and shifting in the same cycle.

The control portion of the multiplier, which is not shown in Fig. 9.4, consists of a counter to keep track of the number of iterations and a simple circuit to effect initialization and detect

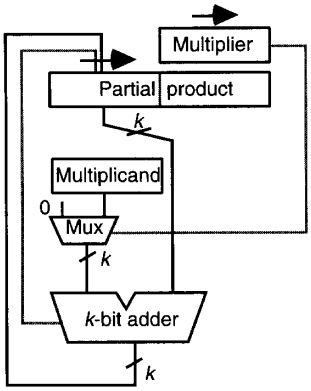


Fig. 9.4 Hardware realization of the sequential multiplication algorithm with additions and right-shifts.

termination. Note that the multiplier and the lower half of the cumulative partial product can share the same register, since as p expands into this register, bits of x are relaxed, keeping the total number of bits at $2k$.

Figure 9.5 shows the double-width register shared by the cumulative partial product and the unused part of the multiplier, along with connections needed to effect simultaneous loading and shifting. Since the register is loaded at the very end of each cycle, the change in its least significant bit, which is controlling the current cycle, will not cause any problem.

Hardware realization of the algorithm with left shifts is depicted in Fig. 9.6. Here too the multiplier x and the cumulative partial product p are stored in shift registers, but the registers shift to the left rather than to the right. The next bit of the multiplier to be considered is always available at the left end of the x register and is used to select 0 or a for the addition. Note that a $2k$-bit adder (actually, a k-bit adder in the lower part, augmented with a k-bit incrementer at the upper end) is needed in the hardware realization of multiplication with left shifts. Because the hardware in Fig. 9.6 is more complex than that in Fig. 9.4, multiplication with right shifts is the preferred method.

The control portion of the multiplier, which is not shown in Fig. 9.6, is similar to that for multiplication with right shifts. Here, register sharing is possible for the multiplier and the upper half of the cumulative partial product, since with each 1-bit expansion in p, one bit of x is relaxed. One difference with the right-shift scheme is that because the double-width register is shifted at the beginning of each cycle, temporary storage is required for keeping the multiplier bit that controls the rest of the cycle.

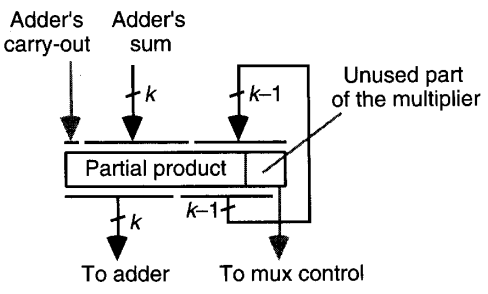


Fig. 9.5 Combining the loading and shifting of the double-width register holding the partial product and the partially used multiplier.

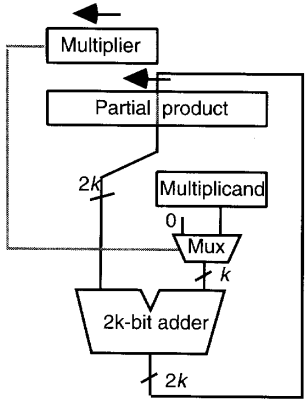


Fig. 9.6 Hardware realization of the sequential multiplication algorithm with left shifts and additions.

9.4 MULTIPLICATION OF SIGNED NUMBERS

The preceding discussions of multiplication algorithms and hardware realizations assume unsigned operands and result. Multiplication of signed-magnitude numbers needs little more, since the product's sign can be computed separately by XORing the operand signs.

One way to multiply signed values with complement representations is to complement the negative operand(s), multiply unsigned values, and then complement the result if only one operand was complemented at the outset. Such an indirect multiplication scheme is quite efficient for 1's-complement numbers but involves too much overhead for 2's-complement representation. It is preferable to use a direct multiplication algorithm for such numbers, as discussed in the remainder of this section.

We first note that the preceding bit-at-a-time algorithms can work directly with a negative 2's-complement multiplicand and a positive multiplier. In this case, each $x_j a$ term will be a 2's-complement number and the sum will be correctly accumulated if we use sign-extend values during the addition process. Figure 9.7 shows the multiplication of a negative multiplicand $a = (-10)_{\text{ten}} = (10110)_{2\text{'s-compl}}$ by a positive multiplier $x = (11)_{\text{ten}} = (01011)_{2\text{'s-compl}}$ using the right-shift algorithm. Note that the leftmost digit of the sum $p^{(i)} + x_i a$ is obtained assuming sign-extended operands.

In view of the negative-weight interpretation of the sign bit in 2's-complement numbers, a negative 2's-complement multiplier can be handled correctly if $x_{k-1}a$ is subtracted, rather than added, in the last cycle. In practice, the required subtraction is performed by adding the 2's-complement of the multiplicand or, actually, adding the 1's-complement of the multiplicand and inserting a carry-in of 1 into the adder (see Fig. 2.7). The required control logic becomes only slightly more complex. Figure 9.8 shows the multiplication of negative values $a = (-10)_{\text{ten}} = (10110)_{2\text{'s-compl}}$ and $x = (-11)_{\text{ten}} = (10101)_{\text{two}}$ by means of the right-shift algorithm.

Multiplication with left shifts becomes even less competitive when we are dealing with 2's-complement numbers directly. Referring to Fig. 9.6, we note that the multiplicand must be sign-extended by k bits. We thus have a more complex adder as well as slower additions. With right shifts, on the other hand, sign extension occurs incrementally; thus the adder needs to be only one bit wider. Alternatively, a k-bit adder can be augmented with special logic to handle the extra bit at the left.

a		1	0	1	1	0			
x		0	1	0	1	1			
$p^{(0)}$		0	0	0	0	0			
$+x_0a$		1	0	1	1	0			
$2p^{(1)}$	1	1	0	1	1	0			
$p^{(1)}$		1	1	0	1	1	0		
$+x_1a$		1	0	1	1	0			
$2p^{(2)}$	1	1	0	0	0	1	0		
$p^{(2)}$		1	1	0	0	0	1	0	
$+x_2a$		0	0	0	0	0			
$2p^{(3)}$	1	1	1	0	0	0	1	0	
$p^{(3)}$		1	1	1	0	0	0	1	0
$+x_3a$		1	0	1	1	0			
$2p^{(4)}$	1	1	0	0	1	0	0	1	0
$p^{(4)}$		1	1	0	0	1	0	0	1	0
$+x_4a$		0	0	0	0	0			
$2p^{(5)}$	1	1	1	0	0	1	0	0	1	0
$p^{(5)}$		1	1	1	0	0	1	0	0	1	0

Fig. 9.7 Sequential multiplication of 2's-complement numbers with right-shifts (positive multiplier).

An alternate way of dealing with 2's-complement numbers is to use Booth's recoding to represent the multiplier x in signed-digit format.

Booth's recoding (also known as Booth's encoding) was first proposed for speeding up radix-2 multiplication in early digital computers. Recall that radix-2 multiplication consists of a sequence of shifts and adds. When 0 is added to the cumulative partial product in a step,

a		1	0	1	1	0			
x		1	0	1	0	1			
$p^{(0)}$		0	0	0	0	0			
$+x_0a$		1	0	1	1	0			
$2p^{(1)}$	1	1	0	1	1	0			
$p^{(1)}$		1	1	0	1	1	0		
$+x_1a$		0	0	0	0	0			
$2p^{(2)}$	1	1	1	0	1	1	0		
$p^{(2)}$		1	1	1	0	1	1	0	
$+x_2a$		1	0	1	1	0			
$2p^{(3)}$	1	1	0	0	1	1	1	0	
$p^{(3)}$		1	1	0	0	1	1	1	0
$+x_3a$		0	0	0	0	0			
$2p^{(4)}$	1	1	1	0	0	1	1	1	0
$p^{(4)}$		1	1	1	0	0	1	1	1	0
$+(-x_4a)$		0	1	0	1	0			
$2p^{(5)}$	0	0	0	1	1	0	1	1	1	0
$p^{(5)}$		0	0	0	1	1	0	1	1	1	0

Fig. 9.8 Sequential multiplication of 2's-complement numbers with right-shifts (negative multiplier).

the addition operation can be skipped altogether. This does not make sense in the designs of Figs. 9.4 and 9.6, since the data paths go through the adder. But in an asynchronous implementation, or in developing a (micro)program for multiplication, shifting alone is faster than addition followed by shifting, and one may take advantage of this fact to reduce the multiplication time on the average. The resulting algorithm or its associated hardware implementation will have variable delay depending on the multiplier value: the more 1s there are in the binary representation of x, the slower the multiplication. Booth observed that whenever there are a large number of consecutive 1s in x, multiplication can be speeded up by replacing the corresponding sequence of additions with a subtraction at the least significant end and an addition in the position immediately to the left of its most significant end. In other words:

$$2^j + 2^{j-1} + \cdots + 2^{i+1} + 2^i = 2^{j+1} - 2^i$$

The longer the sequence of 1s, the larger the savings achieved. The effect of this transformation is to change the binary number x with digit set [0, 1] to the binary signed-digit number y using the digit set $[-1, 1]$. Hence, Booth's recoding can be viewed as a kind of digit-set conversion. Table 9.1 shows how the digit y_i of the recoded number y can be obtained from the two digits x_i and x_{i-1} of x. Thus, as x is scanned from right to left, the digits y_i can be determined on the fly and used to choose add, subtract, or no-operation in each cycle.

For example, consider the following 16-bit binary number and its recoded version:

1 0 0 1	1 1 0 1	1 0 1 0	1 1 1 0	Operand x
(1) $\bar{1}$ 0 1 0	0 $\bar{1}$ 1 0	$\bar{1}$ 1 $\bar{1}$ 1	0 0 $\bar{1}$ 0	Recoded version y

In this particular example, the recoding does not reduce the number of additions. However, the example serves to illustrate two points. First, the recoded number may have to be extended by one bit if the value of x as an unsigned number is to be preserved. Second, if x is a 2's-complement number, then not extending the length (ignoring the leftmost 1 in the recoded version above) leads to the proper handling of negative numbers. Note how in the example, the sign bit of the 2's-complement number has assumed a negative weight in the recoded version, as it should. A complete multiplication example is given in Fig. 9.9.

Radix-2 Booth recoding is not directly applied in modern arithmetic circuits, but it serves as a tool in understanding the radix-4 version of this recoding, to be discussed in Section 10.2.

TABLE 9.1
Radix-2 Booth's recoding

x_i	x_{i-1}	y_i	Explanation
0	0	0	No string of 1s in sight
0	1	1	End of string of 1s in x
1	0	$\bar{1}$	Beginning of string of 1s in x
1	1	0	Continuation of string of 1s in x

```
=============================
a          1 0 1 1 0
x          1 0 1 0 1   Multiplier
y         -1 1-1 1-1   Booth-recoded
=============================
p(0)       0 0 0 0 0
+y0a       0 1 0 1 0
-----------------------------
2p(1)  0 0 1 0 1 0
p(1)       0 0 1 0 1  0
+y1a       1 0 1 1 0
-----------------------------
2p(2)  1 1 1 0 1 1  0
p(2)       1 1 1 0 1  1 0
+y2a       0 1 0 1 0
-----------------------------
2p(3)  0 0 0 1 1 1  1 0
p(3)       0 0 0 1 1  1 1 0
+y3a       1 0 1 1 0
-----------------------------
2p(4)  1 1 1 0 0 1  1 1 0
p(4)       1 1 1 0 0  1 1 1 0
+y4a       0 1 0 1 0
-----------------------------
2p(5)  0 0 0 1 1 0  1 1 1 0
p(5)       0 0 0 1 1  0 1 1 1 0
=============================
```

Fig. 9.9 Sequential multiplication of 2's-complement numbers with right shifts by means of Booth's recoding.

9.5 MULTIPLICATION BY CONSTANTS

When a hardware multiplier, or a corresponding firmware routine, is unavailable, multiplication must be performed by a software routine similar to that in Fig. 9.3. In applications that are not arithmetic-intensive, loss of speed due to the use of such routines is infrequent, hence tolerable. However, many applications involve frequent use of multiplication; in these applications, indiscriminate use of such slow routines may be unacceptable.

Even for applications involving many multiplications, it is true that in a large fraction of cases, one of the operands is a constant that is known at compile time. We all know that multiplication and division by powers of 2 can be done through shifting. It is less obvious that multiplication by many other constants can be performed by short sequences of simple instructions without a need to invoke the complicated general multiplication routine or instruction.

Besides explicit multiplications appearing in arithmetic expressions within programs, there are many implicit multiplications to compute offsets into arrays. For example, if an $m \times n$ array A is stored in row-major order, the offset of the element $A_{i,j}$ (assuming 0-origin indexing) is obtained from the expression $ni + j$. In such implicit multiplications, as well as in a significant fraction of explicit ones, one of the operands is a constant. A multiply instruction takes much longer to execute than a shift or an add instruction even if a hardware multiplier is available. Thus, one might want to avoid the use of a multiply instruction even when it is supported by the hardware.

There are two aspects to multiplication by integer constants. First, one would like to produce optimal or near-optimal code using as few registers as possible. Second, one would like to find the best code by an algorithm that does not require an inordinate amount of time or space. In the examples that follow, R_1 denotes the register holding the multiplicand and R_i will denote

an intermediate result that is i times the multiplicand (e.g., R_{65} denotes the result of multiplying the multiplicand a by 65).

A simple way to multiply the contents of a register by an integer constant multiplier is to write the multiplier in binary format and to use shifts and adds according to the 1s in the binary representation. For example to multiply R_1 by $113 = (1110001)_{two}$, one might use:

$$\begin{aligned} R_2 &\leftarrow R_1 \text{ shift-left } 1 \\ R_3 &\leftarrow R_2 + R_1 \\ R_6 &\leftarrow R_3 \text{ shift-left } 1 \\ R_7 &\leftarrow R_6 + R_1 \\ R_{112} &\leftarrow R_7 \text{ shift-left } 4 \\ R_{113} &\leftarrow R_{112} + R_1 \end{aligned}$$

Only two registers are required; one to store the multiplicand a and one to hold the partially computed result.

If a shift-and-add instruction is available, the sequence above becomes:

$$\begin{aligned} R_3 &\leftarrow R_1 \text{ shift-left } 1 + R_1 \\ R_7 &\leftarrow R_3 \text{ shift-left } 1 + R_1 \\ R_{113} &\leftarrow R_7 \text{ shift-left } 4 + R_1 \end{aligned}$$

If only single-bit shifts are allowed, the last instruction in the preceding sequence must be replaced by three shifts followed by a shift-and-add. Note that the pattern of shift-and-adds and shifts (s&a, s&a, shift, shift, shift, s&a) in this latter version matches the bit pattern of the multiplier if its MSB is ignored (110001).

Many other instruction sequences are possible. For example, one could proceed by computing R_{16}, R_{32}, R_{64}, R_{65}, $R_{97}(R_{65} + R_{32})$, and $R_{113}(R_{97} + R_{16})$. However, this would use up more registers. If subtraction is allowed in the sequence, the number of instructions can be reduced in some cases. For example, by taking advantage of the equality $113 = 128-16+1 = 16(8-1)+1$, one can derive the following sequence of instructions for multiplication by 113:

$$\begin{aligned} R_8 &\leftarrow R_1 \text{ shift-left } 3 \\ R_7 &\leftarrow R_8 - R_1 \\ R_{112} &\leftarrow R_7 \text{ shift-left } 4 \\ R_{113} &\leftarrow R_{112} + R_1 \end{aligned}$$

In general, the use of subtraction helps if the binary representation of the integer has several consecutive 1s, since a sequence of j consecutive 1s can be replaced by $1\,0\,0\,0 \cdots 0\,0\,-1$, where there are $j - 1$ zeros (Booth's recoding).

Factoring a number sometimes helps in obtaining efficient code. For example, to multiply R_1 by 119, one can use the fact that $119 = 7 \times 17 = (8 - 1) \times (16 + 1)$ to obtain the sequence:

$$\begin{aligned} R_8 &\leftarrow R_1 \text{ shift-left } 3 \\ R_7 &\leftarrow R_8 - R_1 \\ R_{112} &\leftarrow R_7 \text{ shift-left } 4 \\ R_{119} &\leftarrow R_{112} + R_7 \end{aligned}$$

With shift-and-add/subtract instructions, the preceding sequence reduces to only two instructions:

$$\begin{aligned} R_7 &\leftarrow R_1 \text{ shift-left } 3 - R_1 \\ R_{119} &\leftarrow R_7 \text{ shift-left } 4 + R_7 \end{aligned}$$

In general, factors of the form $2^b \pm 1$ translate directly into a shift followed by an add or subtract and lead to a simplification of the computation sequence.

In a compiler that removes common subexpressions, moves invariant code out of loops, and performs a reduction of strength on multiplications inside loops (in particular changes multiplications to additions where possible), the effect of multiplication by constants is quite noticeable. It is not uncommon to obtain a 20% improvement in the resulting code, and some programs exhibit 60% improved performance [Bern86].

9.6 PREVIEW OF FAST MULTIPLIERS

If one views multiplication as a multioperand addition problem, there are but two ways to speed it up:

Reducing the number of operands to be added.

Adding the operands faster.

Reducing the number of operands to be added leads to high-radix multipliers in which several bits of the multiplier are multiplied by the multiplicand in one cycle. Speedup is achieved for radix 2^j as long as multiplying j bits of the multiplier by the multiplicand and adding the result to the cumulative partial product takes less than j times as long as multiplying one bit and adding the result. High-radix multipliers are covered in Chapter 10.

To add the partial products faster, one can design hardware multioperand adders that minimize the latency and/or maximize the throughput by using some of the ideas discussed in Chapter 8. These techniques lead to tree and array multipliers, which form the subjects of Chapter 11.

PROBLEMS

9.1 Multiplication in dot notation In Section 9.1, it was stated that for $r > 2$, Fig. 9.1 must be modified (since the partial product terms $x_i a$ will be wider than a). Is there an exception to this general statement?

9.2 Unsigned sequential multiplication Multiply the following 4-bit binary numbers using both the right-shift and left-shift multiplication algorithms. Present your work in the form of Fig. 9.2.

a. $a = 1001$ and $x = 0101$

b. $a = .1101$ and $x = .1001$

9.3 Unsigned sequential multiplication Multiply the following 4-digit decimal numbers using both the right-shift and left-shift multiplication algorithms. Present your work in the form of Fig. 9.2.

a. $a = 8765$ and $x = 4321$

b. $a = .8765$ and $x = .4321$

9.4 Two's-complement sequential multiplication Represent the following signed-magnitude binary numbers in 5-bit, 2's-complement format and multiply them using the right-shift algorithm. Present your work in the form of Fig. 9.7. Then, redo each multiplication using Booth's recoding, presenting your work in the form of Fig. 9.9.

a. $a = +.1001$ and $x = +.0101$

b. $a = +.1001$ and $x = -.0101$

c. $a = -.1001$ and $x = +.0101$

d. $a = -.1001$ and $x = -.0101$

9.5 Programmed multiplication

a. Write the multiplication routine of Fig. 9.3 for a real processor of your choice.

b. Modify the routine of part a to correspond to multiplication with left shifts.

c. Compare the routines of parts a and b with respect to average speed.

d. Modify the routines of parts a and b so that they compute $ax + y$. Compare the resulting routines with respect to average speed.

9.6 Basic hardware multipliers

a. In a hardware multiplier with right shifts (Fig. 9.4), the adder's input multiplexer can be moved to its output side. Show the resulting multiplier design and compare it with respect to cost and speed to that in Fig. 9.4.

b. Repeat part a for the left-shift multiplier depicted in Fig. 9.6.

9.7 Multiplication with left shifts Consider a hardware multiplier with left shifts as in Fig. 9.6, except that multiplier and the upper half of the cumulative partial product share the same register.

a. Draw a diagram similar to Fig. 9.5 for multiplication with left shifts.

b. Explain why carries from adding the multiplicand to the cumulative partial product do not move into, and change, the unused part of the multiplier.

9.8 Basic multiply-add units

a. Show how the multiplier with right shifts, depicted in Fig. 9.4, can be modified to perform a multiply-add step with unsigned operands (compute $ax + y$), where the additive operand y is stored in a special register.

b. Repeat part a for the left-shift multiplier depicted in Fig. 9.6.

c. Extend the design of part a to deal with signed operands.

d. Repeat part b for signed operands and compare the result to part c.

9.9 Direct 2's-complement multiplication

a. Show how the example multiplication depicted in Fig. 9.7 would be done with the left-shift multiplication algorithm.

b. Repeat part a for Fig. 9.8.

c. Repeat part a for Fig. 9.9.

9.10 Booth's recoding Using the fact that we have $y_i = x_{i-1} - x_i$ in Table 9.1, prove the correctness of Booth's recoding algorithm for 2's-complement numbers.

9.11 Direct 1's-complement multiplication Describe and justify a direct multiplication algorithm for 1's-complement numbers. *Hint:* Use initialization of the cumulative partial product and a modified last iteration.

9.12 Multiplication of BSD numbers

a. Multiply the binary signed-digit numbers $(1\ 0\ \bar{1}\ 0\ 1)_{BSD}$ and $(0\ \bar{1}\ 1\ 0\ \bar{1})_{BSD}$ using the right-shift algorithm.

b. Repeat part a using the left-shift algorithm.

c. Design the circuit required for obtaining the partial product $x_j a$ for a sequential BSD hardware multiplier.

9.13 Fully serial multipliers

a. A fully serial multiplier with right shifts is obtained if the adder of Fig. 9.4 is replaced with a bit-serial adder. Show the block diagram of the fully serial multiplier based on the right-shift multiplication algorithm.

b. Design the required control circuit for the fully serial multiplier of part a.

c. Does a fully serial multiplier using the left-shift algorithm make sense?

9.14 Multiplication by constants Using shift and add/subtract instructions only, devise efficient routines for multiplication by the following decimal constants. Assume 32-bit unsigned operands. Make sure that intermediate results do not lead to overflow.

a. 43

b. 129

c. 135

d. 189

d. 211

e. 867

f. 8.75 (the result is to be rounded down to an integer)

9.15 Multiplication by constants

a. Devise a general method for multiplying an integer a by constant multipliers of the form $2^j + 2^i$, where $0 \leq i < j$ (e.g., $36 = 2^5 + 2^2$, $66 = 2^6 + 2^1$).

b. Repeat part a for constants of the form $2^j - 2^i$. Watch for possible overflow.

c. Repeat part a for constants of the form $1 + 2^{-i} + 2^{-j} + 2^{-i-j}$, rounding the result down to an integer.

9.16 Multiplication by constants

a. Devise an efficient algorithm for multiplying an unsigned binary integer by the decimal constant 99. The complexity of your algorithm should be less than those obtained from the binary expansion of 99, with and without Booth's recoding.

b. What is the smallest integer whose binary or Booth-recoded representation does not yield the most efficient multiplication routine with additions and shifts?

REFERENCES

[Bern86] Bernstein, R., "Multiplication by Integer Constants," *Software—Practice and Experience*, Vol. 16, No. 7, pp. 641–652, 1986.

[Boot51] Booth, A. D., "A Signed Binary Multiplication Technique," *Quarterly J. Mechanics and Applied Mathematics*, Vol. 4, Pt. 2, pp. 236–240, June 1951.

[Kore93] Koren, I., *Computer Arithmetic Algorithms*, Prentice-Hall, 1993.

[Omon94] Omondi, A. R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*, Prentice-Hall, 1994.

[Robe55] Robertson, J. E., "Two's Complement Multiplication in Binary Parallel Computers," *IRE Trans. Electronic Computers*, Vol. 4, No. 3, pp. 118–119, September 1955.

[Shaw50] Shaw, R. F., "Arithmetic Operations in a Binary Computer," *Rev. Scientific Instruments*, Vol. 21, pp. 687–693, 1950.

Chapter 10 HIGH-RADIX MULTIPLIERS

In this chapter, we review multiplication schemes that handle more than one bit of the multiplier in each cycle (2 bits per cycle in radix 4, 3 bits in radix 8, etc.). The reduction in the number of cycles, along with the use of recoding and carry-save addition to simplify the required computations in each cycle, leads to significant gains in speed over the basic multipliers of Chapter 9. Chapter topics include:

10.1 RADIX-4 MULTIPLICATION

For a given range of numbers to be represented, a higher representation radix leads to fewer digits. Thus, a digit-at-a-time multiplication algorithm requires fewer cycles as we move to higher radices. This motivates us to study high-radix multiplication algorithms and associated hardware implementations. Since a k-bit binary number can be interpreted as a $\lceil k/2 \rceil$-digit radix-4 number, a $\lceil k/3 \rceil$-digit radix-8 number, and so on, the use of high-radix multiplication essentially entails dealing with more than one bit of the multiplier in each cycle.

We begin by presenting the general radix-r versions of the multiplication recurrences given in Section 9.1:

$$p^{(j+1)} = (p^{(j)} + x_j a\, r^k) r^{-1} \text{ with } p^{(0)} = 0 \text{ and } p^{(k)} = p$$
$$\text{|— add —|}$$
$$\text{|— shift right —|}$$

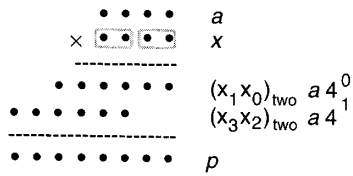


Fig. 10.1 Radix-4, or two-bit-at-a-time, multiplication in dot notation.

$$p^{(j+1)} = rp^{(j)} + x_{k-j-1}a \quad \text{with} \quad p^{(0)} = 0 \text{ and } p^{(k)} = p$$

shift left — add

Since multiplication by r^{-1} or r still entails right or left shifting by one digit, the only difference between high-radix and radix-2 multiplication is in forming the terms $x_i a$, which now require more computation.

For example, if multiplication is done in radix 4, in each step, the partial product term $(x_{i+1}x_i)_{two}\, a$ needs to be formed and added to the cumulative partial product. Figure 10.1 shows the multiplication process in dot notation. Straightforward application of this method leads to the following problem. Whereas in radix-2 multiplication, each row of dots in the partial products matrix represents 0 or a shifted version of a, here we need the multiples $0a$, $1a$, $2a$, and $3a$. The first 3 of these present no problem ($2a$ is simply the shifted version of a). But computing $3a$ needs at least an addition operation ($3a = 2a + a$).

In the remainder of this section, and in Section 10.2, we review several solutions for the preceding problem in radix-4 multiplication.

The first option is to compute $3a$ once at the outset and store it in a register for future use. Then, the rest of the multiplier hardware will be very similar to that depicted in Fig. 9.4, except that the two-way multiplexer is replaced by a four-way multiplexer as shown in Fig. 10.2. An example multiplication is given in Fig. 10.3.

Another possible solution exists when $3a$ needs to be added: we add $-a$ and send a carry of 1 into the next radix-4 digit of the multiplier (Fig. 10.4). Including the incoming carry, the needed multiple in each cycle is in [0, 4]. The multiples 0, 1, and 2 are handled directly, while the multiples 3 and 4 are converted to -1 and 0, respectively, plus an outgoing carry of 1. An extra cycle may be needed at the end because of the carry.

The multiplication schemes depicted in Figs. 10.2 and 10.4 can be extended to radices 8, 16, etc., but the multiple generation hardware becomes more complex for higher radices, nullifying most, if not all, of the gain in speed due to fewer cycles. For example, in radix 8 one needs to precompute the multiples $3a$, $5a$, and $7a$, or else precompute only $3a$ and use a carry

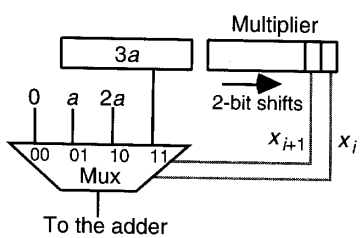


Fig. 10.2 The multiple generation part of a radix-4 multiplier with precomputation of $3a$.

```
================================
a                    0 1 1 0
3a                 0 1 0 0 1 0
x                    1 1 1 0
================================
p(0)                 0 0 0 0
+(x1x0)two a       0 0 1 1 0 0
--------------------------------
4p(1)              0 0 1 1 0 0
p(1)                 0 0 1 1   0 0
+(x3x2)two a       0 1 0 0 1 0
--------------------------------
4p(2)              0 1 0 1 0 1   0 0
p(2)                 0 1 0 1   0 1 0 0
================================
```

Fig. 10.3 Example of radix-4 multiplication using the $3a$ multiple.

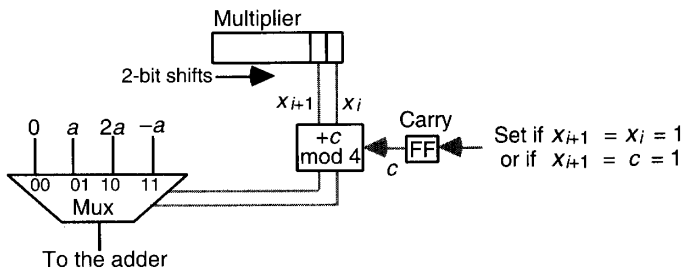


Fig. 10.4 The multiple generation part of a radix-4 multiplier based on replacing $3a$ with $4a$ (carry into the next higher radix-4 multiplier digit) and $-a$.

scheme similar to that in Fig. 10.4 to convert the multiples $5a$, $6a$, and $7a$ to $-3a$, $-2a$, and $-a$, respectively, plus a carry of 1. Supplying the details is left as an exercise.

We will see later in this chapter that with certain other hardware implementations, even higher radices become practical.

10.2 MODIFIED BOOTH'S RECODING

As stated near the end of Section 9.4, radix-2 Booth recoding is not directly applied in modern arithmetic circuits; however, it does serve as a tool in understanding the higher-radix versions of Booth's recoding. It is easy to see that when a binary number is recoded using Table 9.1, the result will not have consecutive 1s or $^-$1s. Thus, if radix-4 multiplication is performed with the recoded multiplier, only the multiples $\pm a$ and $\pm 2a$ of the multiplicand will be required, all of which are easily obtained by shifting and/or complementation.

Now since y_{i+1} depends on x_{i+1} and x_i and y_i depends on x_i and x_{i-1}, the radix-4 digit $z_{i/2} = (y_{i+1}y_i)_{\text{two}}$, i even, can be obtained directly from x_{i+1}, x_i, and x_{i-1} without a need for first forming the radix-2 recoded number y (Table 10.1).

Like the radix-2 version, radix-4 Booth's recoding can be viewed as digit-set conversion: the recoding takes a radix-4 number with digits in [0, 3] and converts it to the digit set [−2, 2].

TABLE 10.1
Radix-4 Booth's recoding yielding $(z_{k/2} \cdots z_1 z_0)_{\text{four}}$

x_{i+1}	x_i	x_{i-1}	y_{i+1}	y_i	$z_{i/2}$	Explanation
0	0	0	0	0	0	No string of 1s in sight
0	0	1	0	1	1	End of a string of 1s in x
0	1	0	1	-1	1	Isolated 1 in x
0	1	1	1	0	2	End of a string of 1s in x
1	0	0	-1	0	-2	Beginning of a string of 1s in x
1	0	1	-1	1	-1	End one string, begin new string
1	1	0	0	-1	-1	Beginning of a string of 1s in x
1	1	1	0	0	0	Continuation of string of 1s in x

As an example, Table 10.1 can be used to perform the following conversion of an unsigned number into a signed-digit number:

$$\begin{aligned}(21\ 31\ 22\ 32)_{\text{four}} &= (10\ 01\ 11\ 01\ 10\ 10\ 11\ 10)_{\text{two}} \\ &= (1\ \bar{2}\ \ 2\ \bar{1}\ \ 2\ \bar{1}\ \bar{1}\ \ 0\ \bar{2})_{\text{four}}\end{aligned}$$

Note that the 16-bit unsigned number turns into a 9-digit radix-4 number. Generally, the radix-4 signed-digit representation of a k-bit unsigned binary number will need $\lfloor k/2 \rfloor + 1 = \lceil (k+1)/2 \rceil$ digits when its most-significant bit is 1. Note also that $x_{-1} = x_k = x_{k+1} = 0$ is assumed.

If the binary number in the preceding example is interpreted as being in 2's-complement format, then simply ignoring the extra radix-4 digit produced leads to correct encoding of the represented value:

$$(1001\ 1101\ 1010\ 1110)_{2\text{'s-compl}} = (\bar{2}\ 2\ \bar{1}\ 2\ \bar{1}\ \bar{1}\ 0\ \bar{2})_{\text{four}}$$

Thus, for k-bit binary numbers in 2's-complement format, the Booth-encoded radix-4 version will have $\lceil k/2 \rceil$ digits. When k is odd, $x_k = x_{k-1}$ is assumed for proper recoding. In any case, $x_{-1} = 0$.

The digit-set conversion process defined by radix-4 Booth's recoding entails no carry propagation. Each radix-4 digit in $[-2, 2]$ is obtained, independently from all others, by examining 3 bits of the multiplier, with consecutive 3-bit segments overlapping in one bit. For this reason, radix-4 Booth's recoding is said to be based on overlapped 3-bit scanning of the multiplier. This can be extended to overlapped multiple-bit scanning schemes for higher radices (see Section 10.4).

An example radix-4 multiplication using Booth's recoding is shown in Fig. 10.5. The 4-bit 2's-complement multiplier $x = (1010)_{\text{two}}$ is recoded as a 2-digit radix-4 number $z = (\bar{1}\ \bar{2})_{\text{four}}$, which then dictates the multiples $z_0 a = -2a$ and $z_1 a = -a$ to be added to the cumulative partial product in the two cycles. Note that in all intermediate steps, the upper half of the cumulative partial product is extended from 4 bits to 6 bits to accommodate the sign extension needed for proper handling of the negative values. Also, note the sign extension during the right shift to obtain $p^{(1)}$ from $4p^{(1)}$.

```
================================
a                 0 1 1 0
x                 1 0 1 0
z                   -1  -2   Radix-4 recoded version of x
================================
p(0)        0 0 0 0 0 0
+z0a        1 1 0 1 0 0
--------------------------------
4p(1)       1 1 0 1 0 0
p(1)        1 1 1 1 0 1   0 0
+z1a        1 1 1 0 1 0
--------------------------------
4p(2)       1 1 0 1 1 1   0 0
p(2)            1 1 0 1   1 1 0 0
================================
```

Fig. 10.5 Example radix-4 multiplication with modified Booth's recoding of the 2's-complement multiplier.

Figure 10.6 depicts a possible circuit implementation for multiple generation based on radix-4 Booth's recoding. Since five possible multiples of a or digits $(0, \pm 1, \pm 2)$ are involved, we need at least 3 bits to encode a desired multiple. A simple and efficient encoding is to devote one bit to distinguish 0 from nonzero digits, one bit to the sign of a nonzero digit, and one bit to the magnitude of a nonzero digit (2 encoded as 1 and 1 as 0). The recoding circuit thus has three inputs (x_{i+1}, x_i, x_{i-1}) and produces three outputs: "neg" indicates if the multiple should be added (0) or subtracted (1), "non0" indicates if the multiple is nonzero, and "two" indicates that a nonzero multiple is 2.

It is instructive to compare the recoding scheme implicit in the design of Fig. 10.4 to Booth's recoding of Fig. 10.6 in terms of cost and delay. This is left as an exercise. Note, in particular, that while the recoding produced in Fig. 10.4 is serial and must thus be done from right to left, Booth's recoding is fully parallel and carry-free. This latter property is of no avail in designing digit-at-a-time multipliers, since the recoded digits are used serially anyway. But we will see later that Booth's recoding can be applied to the design of tree and array multipliers, where all the multiples are needed at once.

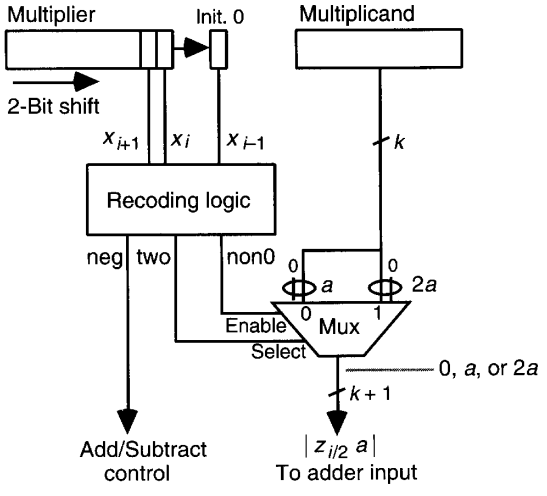


Fig. 10.6 The multiple generation part of a radix-4 multiplier based on Booth's recoding.

10.3 USING CARRY-SAVE ADDERS

Carry-save adders can be used to reduce the number of addition cycles as well as to make each cycle faster. For example, radix-4 multiplication without Booth's recoding can be implemented by using a CSA to handle the $3a$ multiple, as shown in Fig. 10.7. Here, the CSA helps us in doing radix-4 multiplication (generating the required multiples) without reducing the add time. In fact, one can say that the add time is slightly increased, since the CSA overhead is paid in every cycle, regardless of whether we actually need $3a$.

The CSA and multiplexers in the radix-4 multiplier of Fig. 10.7 can be put to better use for reducing the addition time in radix-2 multiplication by keeping the cumulative partial product in stored-carry form. In fact, only the upper half of the cumulative partial product needs to be kept in redundant form, since as we add the three values that form the next cumulative partial product, one bit of the final product is obtained in standard binary form and is shifted into the lower half of the double-width partial product register (Fig. 10.8). This eliminates the need for carry propagation in all but the final addition.

Each of the first $k - 1$ cycles can now be made much shorter, since in these cycles, signals pass through only a few gate levels corresponding to the multiplexers and the CSA. In particular, the delay in these cycles is independent of the word width k. Compared to a simple sequential multiplier (Fig. 9.4), the additional components needed to implement the CSA-based binary multiplier of Fig. 10.8 are a k-bit register, a k-bit CSA, and a k-bit multiplexer; only the extra k-bit register is missing in the design of Fig. 10.7.

The CSA-based design of Fig. 10.8 can be combined with radix-4 Booth's recoding to reduce the number of cycles by 50%, while also making each cycle considerably faster. The changes needed in the design of Fig. 10.8 to accomplish this are depicted in Fig. 10.9, where the small 2-bit adder is needed to combine two bits of the sum, one bit of the carry, and a carry from a preceding cycle into two bits that are shifted into the lower half of the cumulative partial product register and a carry that is kept for the next cycle. The use of the carry-in input of the 2-bit adder is explained shortly.

The Booth recoding and multiple selection logic of Fig. 10.9 is different from the arrangement in Fig. 10.6, since the sign of each multiple must be incorporated in the multiple itself, rather than as a signal that controls addition/subtraction. Figure 10.10 depicts Booth recoding and multiple selection circuits that can be used for high-radix and parallel multipliers.

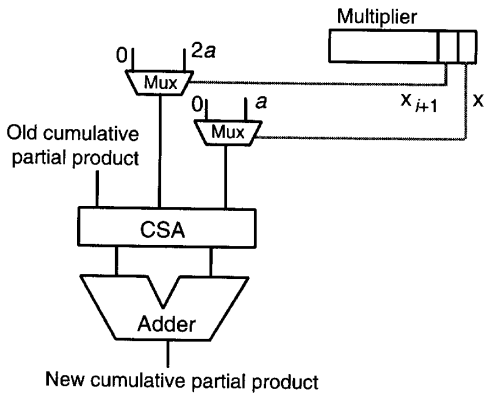


Fig. 10.7 Radix-4 multiplication with a carry-save adder used to combine the cumulative partial product, $x_i a$, and $2x_{i+1}a$ into two numbers.

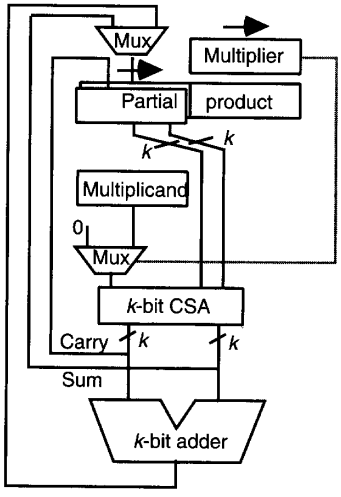


Fig. 10.8 Radix-2 multiplication with the upper half of the cumulative partial product kept in stored-carry form.

Note that in the circuit of Fig. 10.10, the negative multiples $-a$ and $-2a$ are produced in 2's-complement format. As usual, this is done by bitwise complementation of a or $2a$ and the addition of 1 in the LSB position. The multiple a or $2a$ produced from x_i and x_{i+1} is aligned at the right with bit position i and thus must be padded with i zeros at its right end when viewed as a $2k$-bit number. Bitwise complementation of these 0s, followed by the addition of 1 in the LSB position, converts them back to 0s and causes a carry to enter bit position i. For this reason, we can continue to ignore positions 0 through $i - 1$ in the negative multiples and insert the extra "dot" directly in bit position i (Fig. 10.10).

Alternatively, one can do away with Booth's recoding and use the scheme depicted in Fig. 10.7 to accommodate the required $3a$ multiple. Now, four numbers (the sum and carry components of the cumulative partial product, $x_i a$, and $2x_{i+1}a$) need to be combined, thus necessitating a two-level CSA tree (Fig. 10.11).

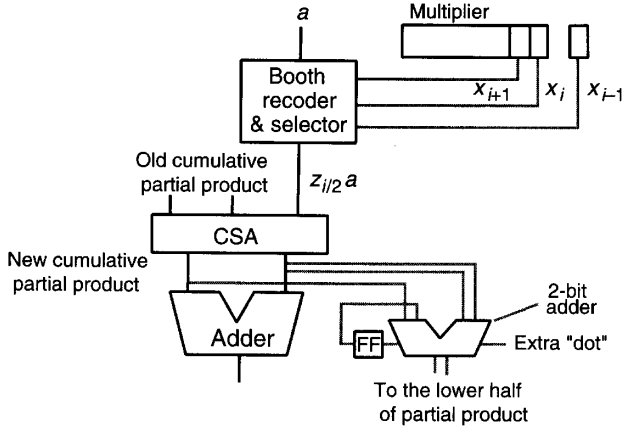


Fig. 10.9 Radix-4 multiplication with a carry-save adder used to combine the stored-carry cumulative partial product and $z_{i/2}a$ into two numbers.

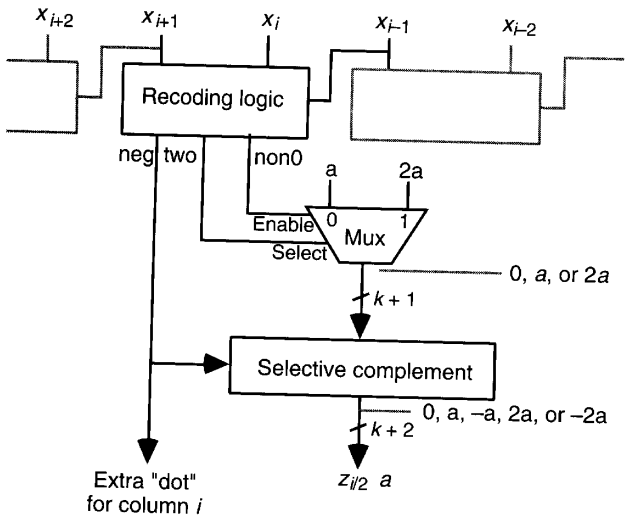


Fig. 10.10 Booth recoding and multiple selection logic for high-radix or parallel multiplication.

10.4 RADIX-8 AND RADIX-16 MULTIPLIERS

From the radix-4 multiplier in Fig. 10.11, it is an easy step to visualize higher-radix multipliers. A radix-8 multiplier, for example, might have a three-level CSA tree to combine the carry-save cumulative partial product with the three multiples $x_i a$, $2x_{i+1}a$, and $4x_{i+2}a$ into a new cumulative partial product in carry-save form. However, once we have gone to three levels

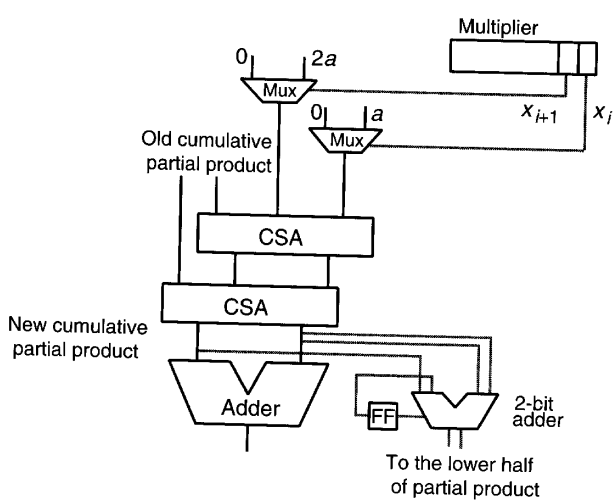


Fig. 10.11 Radix-4 multiplication, with the cumulative partial product, $x_i a$, and $2x_{i+1}a$ combined into two numbers by two carry-save adders

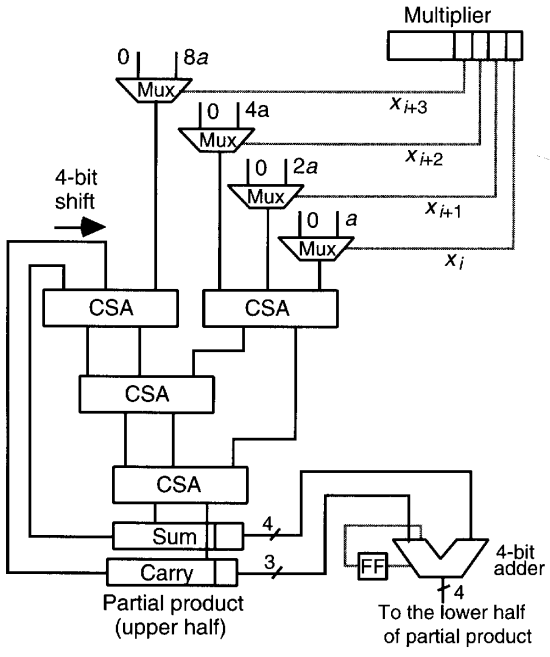


Fig. 10.12 Radix-16 multiplication with the upper half of the cumulative partial product in carry-save form.

of CSA, we might as well invest in one more CSA to implement a radix-16, or 4-bits-at-a-time, multiplier. The resulting design is depicted in Fig. 10.12.

An alternative radix-16 multiplier can be derived from Fig. 10.11 if we replace each of the multiplexers with Booth recoding and multiple selection circuits. Supplying the details of the multiplier design, including proper alignment and sign extension for the inputs to the CSA tree, is left as an exercise.

Which of the preceding radix-16 multipliers (Fig. 10.12, or Fig. 10.11 modified to include Booth's recoding) is faster or more cost-effective depends on the detailed circuit-level designs as well as technological parameters.

Note that in radix-2^b multiplication with Booth's recoding, we have to reduce $b/2$ multiples to 2 using a $(b/2+2)$-input CSA tree whose other two inputs are taken by the carry-save partial product. Without Booth's recoding, a $(b+2)$-input CSA tree would be needed. Whether to use Booth's recoding is a fairly close call, since Booth recoding circuit and multiple selection logic is somewhat slower than a CSA but also has a larger reduction factor in the number of operands (2 vs. 1.5).

Varied as the preceding choices are, they do not exhaust the design space. Other alternatives include radix-8 and radix-16 Booth's recoding, which represent the multiplier using the digit sets $[-4, 4]$ and $[-8, 8]$, respectively. We will explore the recoding process and the associated multiplier design options in the end-of-chapter problems. Note, for example, that with radix-8 recoding, we have the $\pm 3a$ multiples to deal with. As before, we can precompute $3a$ or represent it as the pair of numbers $2a$ and a, leading to the requirement for an extra input into the CSA tree.

There is, of course, no compelling reason to stop at radix 16. A design similar to that in Fig. 10.12 can be used for radix-256 (8-bits-at-a-time) multiplication if Booth's recoding is applied first. This would require that the four multiplexers in Fig. 10.12 be replaced by the Booth recoding and selection logic. Again, whether this new arrangement will lead to a cost-effective

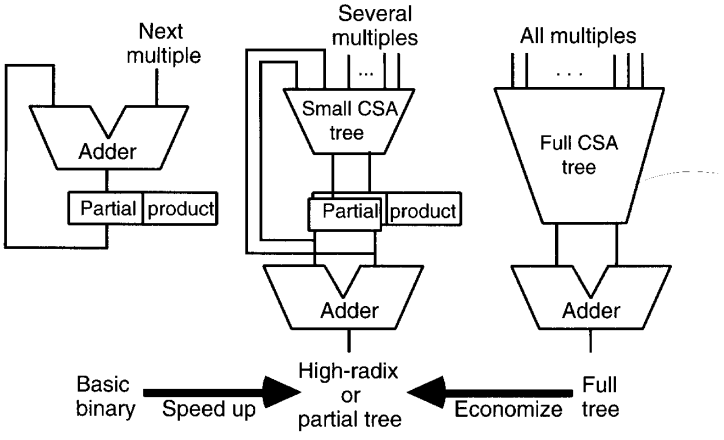


Fig. 10.13 High-radix multipliers as intermediate between sequential radix-2 and full-tree multipliers.

design (compared, e.g., to taking 7 bits of the multiplier and adding nine numbers in a four-level CSA tree) depends on the technology and cannot be discerned in general.

Designs such as the ones depicted in Figs. 10.11 and 10.12 can be viewed as intermediate between basic sequential (one-bit-at-a-time) multiplication and fully parallel tree multipliers to be discussed in Chapter 11. Thus, high-radix or partial-tree multipliers can be viewed as designs that offer speedup over sequential multiplication or economy over fully parallel tree multipliers (Fig. 10.13).

10.5 MULTIBEAT MULTIPLIERS

In the CSA-based binary multiplier shown in Fig. 10.8, CSA outputs are loaded into the same registers that supply its inputs. A common implementation method is to use master–slave flip-flops for the registers. In this method, each register has two sides: the master side accepts new data being written into the register while the slave side, which supplies the register's outputs, keeps the old data for the entire half-cycle when the clock is high. When the clock goes low, the new data in the master side is transferred to the slave side in preparation for the next cycle. In this case, one might be able to insert an extra CSA between the master and slave registers, with little or no effect on the clock's cycle time. This virtually doubles the speed of partial product accumulation.

Figure 10.14 shows a schematic representation of a 3-bit-at-a-time twin-beat multiplier that effectively retires 6 bits of the multiplier in each clock cycle. This multiplier, which uses radix-8 Booth's recoding, is similar to the twin-beat design used in Manchester University's MU5 computer [Gosl71].

Each clock cycle is divided into two phases or beats. In the first beat, the left multiplier register is used to determine the next multiple to be added, while in the second beat, the right multiplier register is used. After each cycle (two beats), the small adder at the lower right of Fig. 10.14 determines 6 bits of the product, which are shifted into the lower half of the cumulative

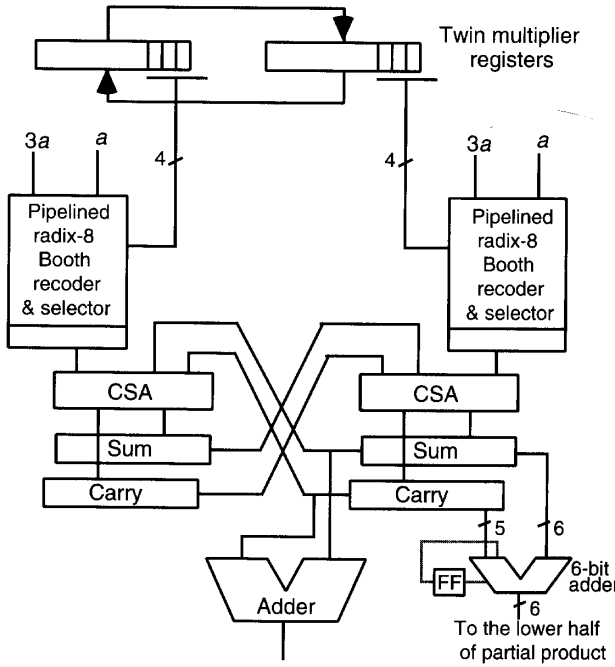


Fig. 10.14 Twin-beat multiplier with radix-8 Booth's recoding.

partial product register. This adder is in all likelihood slower than the CSAs; hence, to make each cycle as short as possible, the adder must be pipelined. Since the product bits, once produced, do not change, the latency in deriving these bits has no effect on the rest of the computation in the carry-save portion of the circuit.

The twin-beat concept can be easily extended to obtain a three-beat multiplier. Such a design can be visualized by putting the three CSAs and associated latches into a ring (Fig. 10.15), whose nodes are driven by a three-phase clock [deAn95]. Each node requires two beats before making its results available to the next node, thus leading to separate accumulation of odd- and even-indexed partial products. At the end, the four operands are reduced to two operands, which are then added to obtain the final product.

10.6 VLSI COMPLEXITY ISSUES

Implementation of sequential radix-2 and high-radix multipliers described thus far in Chapters 9 and 10 is straightforward. The components used are carry-save adders, registers, multiplexers, and a final fast carry-propagate adder, for which standard designs are available. A small amount of random control logic is also required. Note that each 2-to-1 multiplexer with one of the inputs tied to 0 can be simplified to a set of AND gates.

For the CSA tree of a radix-2^b multiplier, typically a bit slice is designed and then replicated. Since without Booth's recoding, the CSA tree receives $b+2$ inputs, the required slice is a $(b+2; 2)$-counter; see Section 8.5. For example, a set of (7; 2)-counter slices can be used to implement the CSA tree of a radix-32 multiplier without Booth's recoding. When radix-2^h Booth's recoding

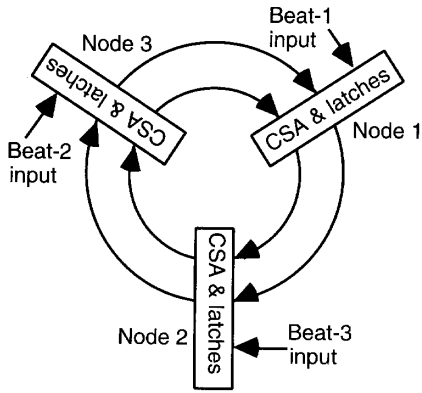


Fig. 10.15 Conceptual view of a three-beat multiplier.

is applied first, then the number of multiples per cycle is reduced by a factor of h and a $(b/h+2; 2)$-counter slice will be needed.

In performing radix-2^b multiplication, bk two-input AND gates are required to form the b multiples for each cycle in parallel. The area complexity of the CSA tree that reduces these b multiples to 2 is $O(bk)$. Since these complexities dominate that of the final fast adder, the overall area requirement is seen to be:

$$A = O(bk)$$

In view of the logarithmic height of the CSA tree, as discussed in Section 8.3, multiplication is performed in k/b cycles of duration $O(\log b)$, plus a final addition requiring $O(\log k)$ time. The overall time complexity thus becomes:

$$T = O((k/b) \log\ b + \log\ k)$$

It is well known that any VLSI circuit computing the product of two k-bit integers must satisfy the following constraints involving its layout area A and computational latency T: AT is at least proportional to $k\sqrt{k}$ and AT^2 grows at least as fast as k^2. For the preceding implementations, we have:

$$AT = O(k^2 \log\ b + bk\ \log\ k)$$
$$AT^2 = O((k^3/b)\ \log^2 b)$$

At the lower end of the complexity scale, where b is a constant, the AT and AT^2 measures for our multipliers become $O(k^2)$ and $O(k^3)$, respectively. At the other extreme corresponding to $b = k$, where all the multiplier bits are considered at once, we have $AT = O(k^2 \log\ k)$ and $AT^2 = O(k^2 \log^2 k)$. Intermediate designs do not yield better values for AT and AT^2; thus, the multipliers remain asymptotically suboptimal for the entire range of the parameter b.

By the AT measure, which is often taken as an indicator of cost-effectiveness, the slower radix-2 multipliers are better than high-radix or tree multipliers. Therefore, in applications calling for a large number of independent multiplications, it may be appropriate to use the available chip area for a large number of slow multipliers as opposed to a small number of faster units.

We will see, in Chapter 11, that the time complexity of high-radix multipliers can actually be reduced from $O((k/b)\ \log\ b + \log\ k)$ to $O(k/b + \log\ k)$ through a more effective pipelining

scheme. Even though the resulting designs lead to somewhat better AT and AT^2 measures, the preceding conclusions do not change.

Despite these negative results pointing to the asymptotic suboptimality of high-radix and tree multipliers, such designs are quite practical for a wide range of the parameter b, given that the word width k is quite modest in practice.

PROBLEMS

10.1 Radix-4 Booth's recoding Prove that radix-4 Booth's recoding defined in Table 10.1 preserves the value of an unsigned or 2's-complement number. *Hint:* First show that the recoded radix-4 digit $z_{i/2}$ can be obtained from the arithmetic expression $-2x_{i+1} + x_i + x_{i-1}$.

10.2 Sequential radix-4 multipliers

a. Consider the radix-4 multiplier depicted in Fig. 10.2. What provisions are needed if 2's-complement multipliers are to be handled appropriately?

b. Repeat part a for the multiplier depicted in Fig. 10.4.

10.3 Alternate radix-4 multiplication algorithms Consider the example unsigned multiplication $(0\ 1\ 1\ 0)_{\text{two}} \times (1\ 1\ 1\ 0)_{\text{two}}$ depicted in Fig. 10.3.

a. Redo the example multiplication using the scheme shown in Fig. 10.4.

b. Redo the example multiplication using radix-4 Booth's recoding.

c. Redo the example multiplication using the scheme shown in Fig. 10.7. Show the intermediate sum and carry values in each step.

10.4 Sequential unsigned radix-4 multipliers

a. Design the recoding logic needed for the multiplier of Fig. 10.4.

b. Give a complete design for the Booth recoding logic circuit shown in Fig. 10.6.

c. Compare the circuits of parts a and b with respect to cost and delay. Which scheme is more cost-effective for sequential unsigned radix-4 multiplication?

d. Compare the radix-4 multiplier shown in Fig. 10.2 against those in part c with respect to cost and delay. Summarize your conclusions.

10.5 Alternate radix-4 recoding scheme

a. The design of the Booth recoder and multiple selection circuits in Fig. 10.6 assumes the use of a multiplexer with an enable control signal. How will the design change if such a multiplexer is not available?

b. Repeat part a for the circuit of Fig. 10.10.

10.6 Recoding for radix-8 multiplication

a. Construct a recoding table (like Table 10.1) to obtain radix-8 digits in $[-4, 4]$ based on overlapped 4-bit groups of binary digits in the multiplier.

b. Show that your recoding scheme preserves the value of a number. *Hint:* Express the recoded radix-8 digit $z_{i/3}$ as a linear function of x_{i+2}, x_{i+1}, x_i, and x_{i-1}.

c. Design the required recoding logic block.

d. Draw a block diagram for the radix-8 multiplier and compare it to radix-4 design.

10.7 Recoding for radix-16 multiplication

a. Construct a recoding table (like Table 10.1) to obtain radix-16 digits in $[-8, 8]$ based on overlapped 5-bit groups of binary digits in the multiplier.

b. Show that your recoding scheme preserves the value of a number. *Hint:* Express the recoded radix-16 digit $z_{i/4}$ as a linear function of $x_{i+3}, x_{i+2}, x_{i+1}, x_i$, and x_{i-1}.

c. Design the required recoding logic block.

d. Draw a block diagram for the radix-16 multiplier and compare it to radix-4 design.

10.8 Alternate radix-4 recoding scheme The radix-4 Booth recoding scheme of Table 10.1 replaces the two bits x_{i+1} and x_i of the multiplier with a radix-4 digit 0, ± 1, or ± 2 by examining x_{i-1} as the recoding context. An alternative recoding scheme is to replace x_{i+1} and x_i with a radix-4 digit 0, ± 2, or ± 4 by using x_{i+2} as the context.

a. Construct the required radix-4 recoding table.

b. Design the needed recoding logic block.

c. Compare the resulting multiplier to that obtained from radix-4 Booth recoding with respect to possible advantages and drawbacks.

10.9 Comparing radix-4 multipliers Compare the multipliers in Figs. 10.9 and 10.11 with regard to speed and hardware implementation cost. State and justify all your assumptions.

10.10 Very-high-radix multipliers The 4-bit adder shown at the lower right of Fig. 10.12 may be slower than the CSA tree, thus lengthening the cycle time. The problem becomes worse for higher radices. Discuss how this problem can be mediated.

10.11 Multibeat multipliers Study the design of the three-beat multiplier in [deAn95]. Based on your understanding of the design, discuss if anything can be gained by going to a four-beat multiplier.

10.12 VLSI complexity of multipliers

a. A proposed VLSI design for $k \times k$ multiplication requires chip area proportional to $k \log k$. What can you say about the asymptotic speed of this multiplier based on AT and AT^2 bounds?

b. What can you say about the VLSI area requirement of a multiplier that operates in optimal $O(\log k)$ time?

10.13 VLSI multiplier realizations Design a slice of the (6; 2)-counter that is needed to implement the multiplier of Fig. 10.12.

10.14 Multiply-add operation

a. Show that the high-radix multipliers of this chapter can be easily adapted to compute $p = ax + y$ instead of $p = ax$.

b. Extend the result of part a to computing $p = ax + y + z$, where all input operands are k-bit unsigned integers. *Hint:* This is particularly easy with carry-save designs.

10.15 Balanced ternary multiplication Discuss the design of a radix-9 multiplier for balanced ternary operands that use the digit set $[-1, 1]$ in radix 3. Consider all the options presented in this chapter, including the possibility of recoding.

10.16 Decimal multiplier Consider the design of a decimal multiplier using a digit-at-a-time scheme. Assume BCD encoding for the digits.

a. Using a design similar to that in Fig. 10.12, supply the hardware details and discuss how each part of the design differs from the radix-16 version. *Hint:* One approach is to design a special decimal divide-by-2 circuit for deriving the multiple $5a$ from $10a$, forming the required multiples by combining $10a$, $5a$, a, and $-a$.

b. Using a suitable recoding scheme, convert the decimal number to digit set $[-5, 5]$. Does this recoding help make multiplication less complex than in part a?

10.17 Signed-digit multiplier Consider the multiplication of radix-3 integers using the redundant digit set $[-2, 2]$.

a. Draw a block diagram for the requisite radix-3 multiplier using the encoding given in connection with radix-4 Booth's recoding (Fig. 10.6) to represent the digits.

b. Show the detailed design of the circuit that provides the multiple $2a$.

c. Present the design of a radix-9 multiplier that relaxes two multiplier digits per cycle.

REFERENCES

[Boot51] Booth, A. D., "A Signed Binary Multiplication Technique," *Quarterly J. Mechanics and Applied Mathematics*, Vol. 4, Pt. 2, pp. 236–240, June 1951.

[deAn95] de Angel, E., A. Chowdhury, and E.E. Swartzlander, "The Star Multiplier," *Proc. 29th Asilomar Conf. Signals, Systems, and Computers*, pp. 604–607, 1995.

[Gosl71] Gosling, J. B., "Design of Large High-Speed Binary Multiplier Units," *Proc. IEE*, Vol. 118, Nos. 3/4, pp. 499–505, 1971.

[MacS61] MacSorley, O. L., "High-Speed Arithmetic in Binary Computers," *Proc. IRE*, Vol. 49, pp. 67–91, 1961.

[Rubi75] Rubinfield, L. P., "A Proof of the Modified Booth's Algorithm for Multiplication," *IEEE Trans. Computers*, Vol. 25, No. 10, pp. 1014–1015, 1975.

[Sam90] Sam, H., and A. Gupta, "A Generalized Multibit Recoding of the Two's Complement Binary Numbers and Its Proof with Application in Multiplier Implementations," *IEEE Trans. Computers*, Vol. 39, No. 8, pp. 1006–1015, 1990.

[Vass89] Vassiliadis, S., E. M. Schwartz, and D. J. Hanrahan, "A General Proof for Overlapped Multiple-Bit Scanning Multiplications," *IEEE Trans. Computers*, Vol. 38, No. 2, pp. 172–183, 1989.

[Wase82] Waser, S., and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart, & Winston, 1982.

[Zura87] Zurawski, J. H. P., and J. B. Gosling, "Design of a High-Speed Square-Root, Multiply, and Divide Unit," *IEEE Trans. Computers*, Vol. 36, No. 1, pp. 13–23, 1987.

Chapter 11 TREE AND ARRAY MULTIPLIERS

Tree, or fully parallel, multipliers constitute limiting cases of high-radix multipliers (radix-2^k). With a high-performance CSA tree followed by a fast adder, logarithmic time multiplication becomes possible. The resulting multipliers are expensive but justifiable for applications in which multiplication speed is critical. One-sided CSA trees lead to much slower, but highly regular, structures known as array multipliers that offer higher pipelined throughput than tree multipliers and significantly lower chip area at the same time. Chapter topics include:

11.1 FULL-TREE MULTIPLIERS

In their simplest forms, parallel or full-tree multipliers can be viewed as extreme cases of the design in Fig. 10.12, where all the k multiples of the multiplicand are produced at once and a k-input CSA tree is used to reduce them to two operands for the final addition. Because all the multiples are combined in one pass, the tree does not require feedback links, making pipelining quite feasible.

Figure 11.1 shows the general structure of a full-tree multiplier. Various multiples of the multiplicand a, corresponding to binary or high-radix digits of the multiplier x or its recoded version, are formed at the top. The multiple-forming circuits may be a collection of AND gates (binary multiplier), radix-4 Booth's multiple generators (recoded multiplier), and so on. These multiples are added in a combinational partial products reduction tree, which produces their sum in redundant form. Finally, the redundant result is converted to standard binary output at the bottom.

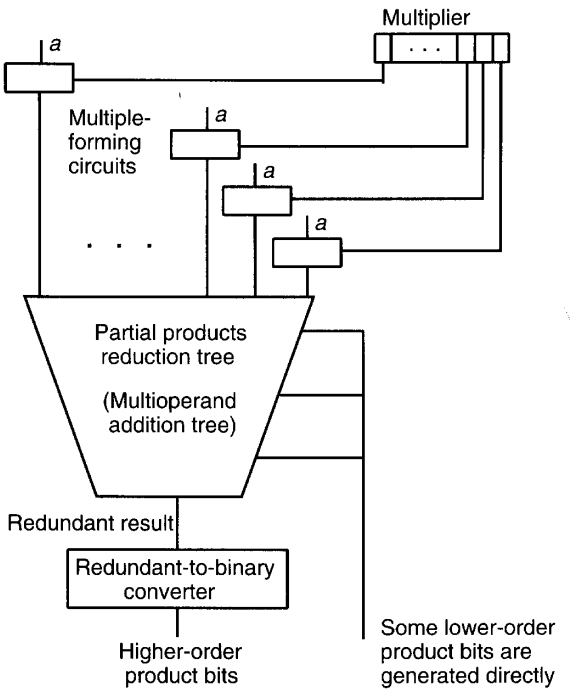


Fig. 11.1 General structure of a full-tree multiplier.

Many types of tree multiplier have been built or proposed. These are distinguished by the designs of the following three elements in Fig. 11.1:

multiple-forming circuits
partial products reduction tree
redundant-to-binary converter

In the remainder of this section, we focus on tree multiplier variations involving unsigned binary multiples and CSA reduction trees. With the redundant result in carry-save form, the final converter is simply a fast adder. Deviations from the foregoing multiple generation and reduction schemes are discussed in Section 11.2. Signed tree multipliers are covered in Section 11.3.

From our discussion of sequential multiplication in Chapters 9 and 10, we know how the partial products can be formed and how, through the use of high-radix methods, the number of partial products can be reduced. The trade-offs mentioned for high-radix multipliers exist here as well: more complex multiple-forming circuits can lead to simplification in the reduction tree. Again, we cannot say in general which combination will lead to greater cost-effectiveness because the exact nature of the trade-off is design- and technology-dependent.

Recall Wallace's and Dadda's strategies for constructing CSA trees discussed in Section 8.3. These give rise to Wallace and Dadda tree multipliers, respectively. Essentially, Wallace's strategy for building CSA trees is to combine the partial product bits at the earliest opportunity, while with Dadda's method, combining takes place as late as possible, consistent with keeping the critical path length of the CSA tree intact. Wallace's method leads to the fastest possible design and Dadda's strategy usually leads to a simpler CSA tree and a wider carry-propagate adder.

As a simple example, we derive Wallace and Dadda tree multipliers for 4×4 multiplication. Figure 11.2 shows the design process and results in tabular form, where the integers indicate the number of dots remaining in the various columns. Each design begins with 16 AND gates forming the $x_i a_j$ terms or dots, $0 \leq i, j \leq 3$. The resulting 16 dots are spread across seven columns in the pattern 1, 2, 3, 4, 3, 2, 1. The Wallace tree design requires 3 FAs and 1 HA in the first level, then 2 FAs and 2 HAs in the second level, and a 4-bit carry-propagate adder at the end. With the Dadda tree design, our first goal is to reduce the height of the partial products dot matrix from 4 to 3, thus necessitating 2 FAs in the first level. These are followed by 2 FAs and 2 HAs in the second level (reducing the height from 3 to 2) and a 6-bit carry-propagate adder at the end.

Intermediate approaches between those of Wallace and Dadda yield various designs that offer speed–cost trade-offs. For example, it may be that neither the Wallace tree nor the Dadda tree leads to a convenient width for the fast adder. In such cases a hybrid approach may yield the best results.

Note that the results introduced for carry-save multioperand addition in Chapter 8 apply to the design of partial products reduction trees with virtually no change. The only modifications required stem from the relative shifting of the operands to be added. For example, in Fig. 8.12, we see that in adding seven right-aligned k-bit operands, the CSAs are all k bits wide. In a seven-operand CSA tree of a 7×7 tree multiplier, the input operands appear with shifts of 0 to 6 bits, leading to the input configuration shown at the top of Fig. 11.3. We see that the shifted inputs necessitate somewhat wider blocks at the bottom of the tree. It is instructive to compare Figs. 11.3 and Fig. 8.12, noting all the differences.

Of course, there is no compelling reason to keep all the bits of the input or intermediate operands together and feed them to multibit CSAs, thus necessitating the use of many half-adders that simply rearrange the dots without contributing to their reduction. Doing the reduction with single-bit FAs and HAs, as in Fig. 11.2, leads to lower complexity and perhaps even greater speed. Deriving the Wallace and Dadda tree multipliers to perform the same function as the circuit of Fig. 11.3 is left as an exercise.

One point is quite clear from Fig. 11.3 or its Wallace tree and Dadda tree equivalents: a logarithmic depth reduction tree based on CSAs has an irregular structure that makes its design and layout quite difficult. Additionally, connections and signal paths of varying lengths lead to logic hazards and signal skew that have implications for both performance and power consumption. In VLSI design, we strive to build circuits from iterated or recursive structures that lend themselves to efficient automatic synthesis and layout. Alternative reduction trees that are more suitable for VLSI implementation are discussed next.

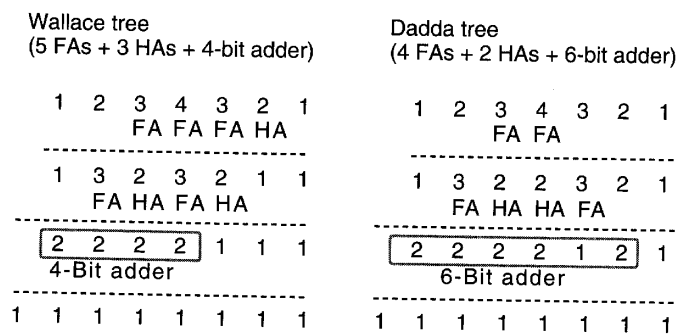


Fig. 11.2 Two different binary 4×4 tree multipliers.

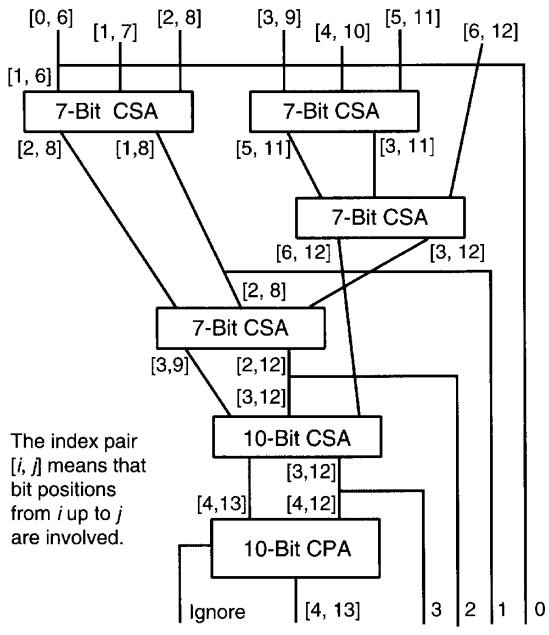


Fig. 11.3 Possible CSA tree for a 7×7 tree multiplier.

11.2 ALTERNATIVE REDUCTION TREES

Recall from our discussion in Section 8.5 that a (7; 2)-counter slice can be designed that takes 7 bits in the same column i as inputs and produces one bit in each of the columns i and $i + 1$ as outputs. Such a slice, when suitably replicated, can perform the function of the reduction tree part of Fig. 11.3. Of course, not all columns in Fig. 11.3 have seven inputs. The preceding iterative circuit can then be left intact and supplied with dummy 0 inputs in the interest of regularity, or it can be pruned by removing the redundant parts in each slice. Such optimizations are well within the power of automated design tools.

Based on Table 8.1, an (11; 2)-counter has at least five full-adder levels. Figure 11.4 shows a particular five-level arrangement of full adders for performing 11-to-2 reduction with the property that all outputs are produced after the same number of full-adder delays. Observe how all carries produced in level i enter FAs in level $i + 1$. The FAs of Fig. 11.4 can be laid out to occupy a narrow vertical slice that can then be replicated to form an 11-input reduction tree of desired width. Such balanced-delay trees are quite suitable for VLSI implementation of parallel multipliers.

The circuit of Fig. 11.4 is composed of three columns containing 1, 3, and 5 FAs, going from left to right. It is now easy to see that the number of inputs can be expanded from 11 to 18 by simply appending to the right of the circuit an additional column of 7 FAs. The top FA in the added column will accommodate three new inputs, while each of the others, except for the lowermost two, can accept one new input; these latter FAs must also accommodate a sum coming from above and a carry coming from the right. Note that the FAs in the various columns are more or less independent in that adjacent columns are linked by just one wire. This property

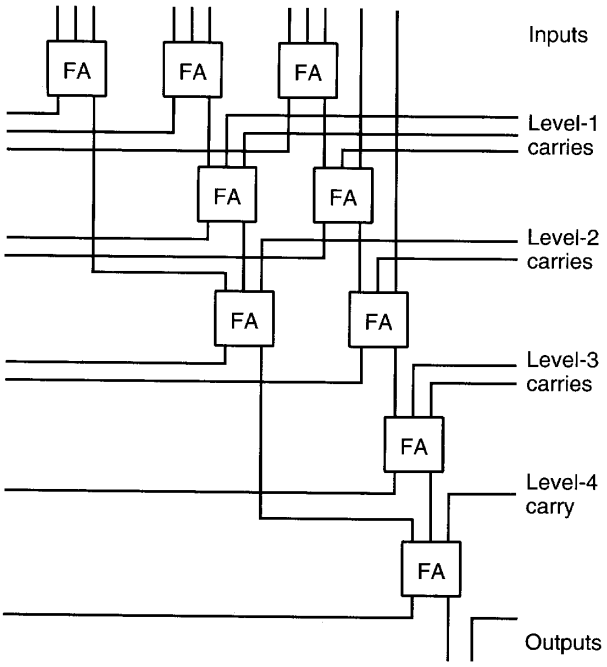


Fig. 11.4 A slice of a balanced-delay tree for 11 inputs.

makes it possible to lay out the circuit in a narrow slice without having to devote a lot of space to the interconnections.

Instead of building partial products reduction trees from CSAs, or (3; 2)-counters, one can use a module that reduces four numbers to two as the basic building block. Then, partial products reduction trees can be structured as binary trees that possess a recursive structure making them more regular and easier to lay out (Fig. 11.5). Figure 11.6 shows a possible way of laying out the seven-module tree of Fig. 11.5. Note that adding a level to the tree of Fig. 11.6 involves duplicating the tree and inserting a 4-to-2 reduction module between them.

In Fig. 11.6, the first, third, fifth, and seventh rectangular boxes correspond to top-level blocks of Fig. 11.5. These blocks receive four multiples of the multiplicand (two from above

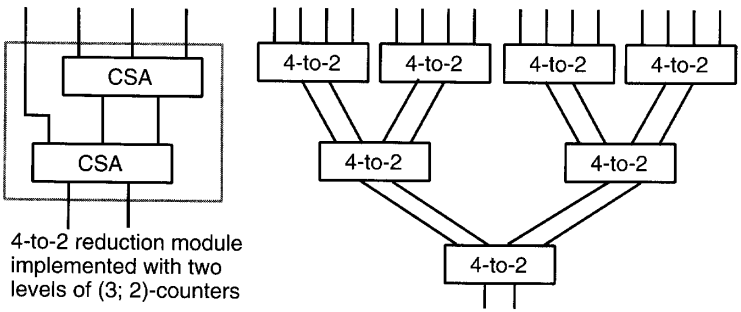


Fig. 11.5 Tree multiplier with a more regular structure based on 4-to-2 reduction modules.

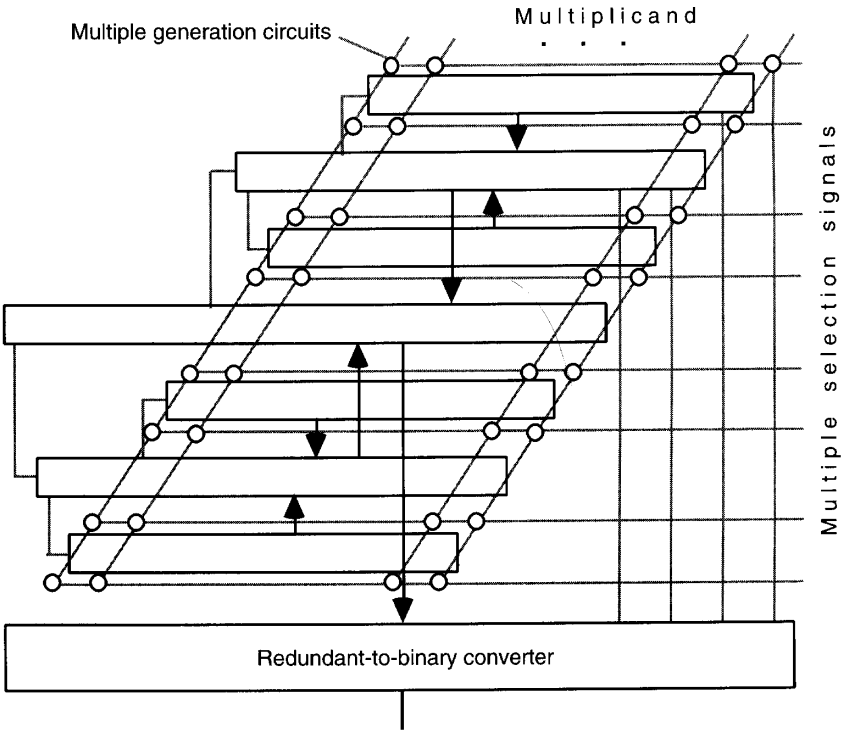


Fig. 11.6 Layout of a partial products reduction tree composed of 4-to-2 reduction modules. Each solid arrow represents two numbers.

and two from below) and reduce them to a pair of numbers for the second and sixth blocks. Each of the latter blocks in turn supplies two numbers to the fourth block, which feeds the redundant-to-binary converter.

If the 4-to-2 reduction modules are internally composed of two CSA levels, as suggested in Fig. 11.5, then there may be more CSA levels in the binary tree structure than in Wallace or Dadda trees. However, regularity of interconnections, and the resulting efficient layout, can more than compensate for the added logic delays due to the greater circuit depth.

Note that a 4-to-2 reduction circuit for binary operands can be viewed as a GSD adder for radix-2 numbers with the digit set [0, 2], where the digits are encoded in the following 2-bit code:

Zero: (0, 0) One: (0, 1) or (1, 0) Two: (1, 1)

A variant of this binary tree reduction scheme is based on binary-signed-digit, rather than carry-save, representation of the partial products [Taka85]. These partial products are combined by a tree of BSD adders to obtain the final product in BSD form. The standard binary result is then obtained via a BSD-to-binary converter, which is essentially a fast subtractor for subtracting the negative component of the BSD number from its positive part. One benefit of BSD partial products is that negative multiples resulting from the sign bit in 2's-complement numbers can be easily accommodated (see Section 11.3). Some inefficiency results from the extra bit used to accommodate the digit signs going to waste for most of the multiples that are positive.

Of course, carry-save and BSD numbers are not the only ones that allow fast reduction via limited-carry addition. Several other digit sets are possible that offer certain advantages depending on technological capabilities and constraints [Parh96]. For example, radix-2 partial products using the digit set [0, 3] lend themselves to an efficient parallel-carries addition process (Fig. 3.11c), while also accommodating three, rather than one or two, multiples of a binary multiplicand. Interestingly, the final conversion from the redundant digit set [0, 3] to [0, 1] is not any harder than conversion from [0, 2] to [0, 1].

Clearly, any method used for building the CSA tree can be combined with radix-2^b Booth's recoding to reduce the tree size. However, for modern VLSI technology, the use of Booth recoding in tree multipliers has been questioned [Vill93]; it seems that the additional CSAs needed for reducing k, rather than k/b, numbers could be less complex than the Booth recoding logic when wiring and the overhead due to irregularity and nonuniformity are taken into account.

11.3 TREE MULTIPLIERS FOR SIGNED NUMBERS

When one is multiplying 2's-complement numbers directly, each of the partial products to be added is a signed number. Thus, for the CSA tree to yield the correct sum of its inputs, each partial product must be sign-extended to the width of the final product. Recall our discussion of signed multioperand addition in Section 8.6, where the 2's-complement operands were assumed to be aligned at their LSBs. In particular, refer to Fig. 8.18 for two possible methods based on sign extension (with hardware sharing) and transforming negative bits into positive bits.

Considerations for adding 2's-complement partial products are similar, the only difference being the shifts. Figure 11.7 depicts an example with three sign-extended partial products. We see that here too a single full adder can produce the results needed in several different columns. If this procedure is applied to all rows in the partial products bit matrix, the resulting structure will be somewhat more complex than the one assuming unsigned operands. Note that because of the shifts, there are fewer repetitions in Fig. 11.7 than in Fig. 8.18, thus making the expansion in width to accommodate the signs slightly larger.

Another approach, due to Baugh and Wooley [Baug73], is even more efficient and is thus often preferred, in its original or modified form, for 2's-complement multiplication. To understand this method, we begin with unsigned multiplication in Fig. 11.8a and note that the negative weight of the sign bit in 2's-complement representation must be taken into account to obtain the correct product (Fig. 11.8b). To avoid having to deal with negatively weighted bits

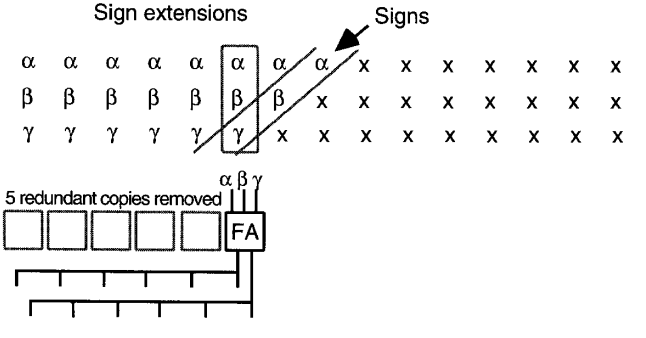


Fig. 11.7 Sharing of full adders to reduce the CSA width in a signed tree multiplier.

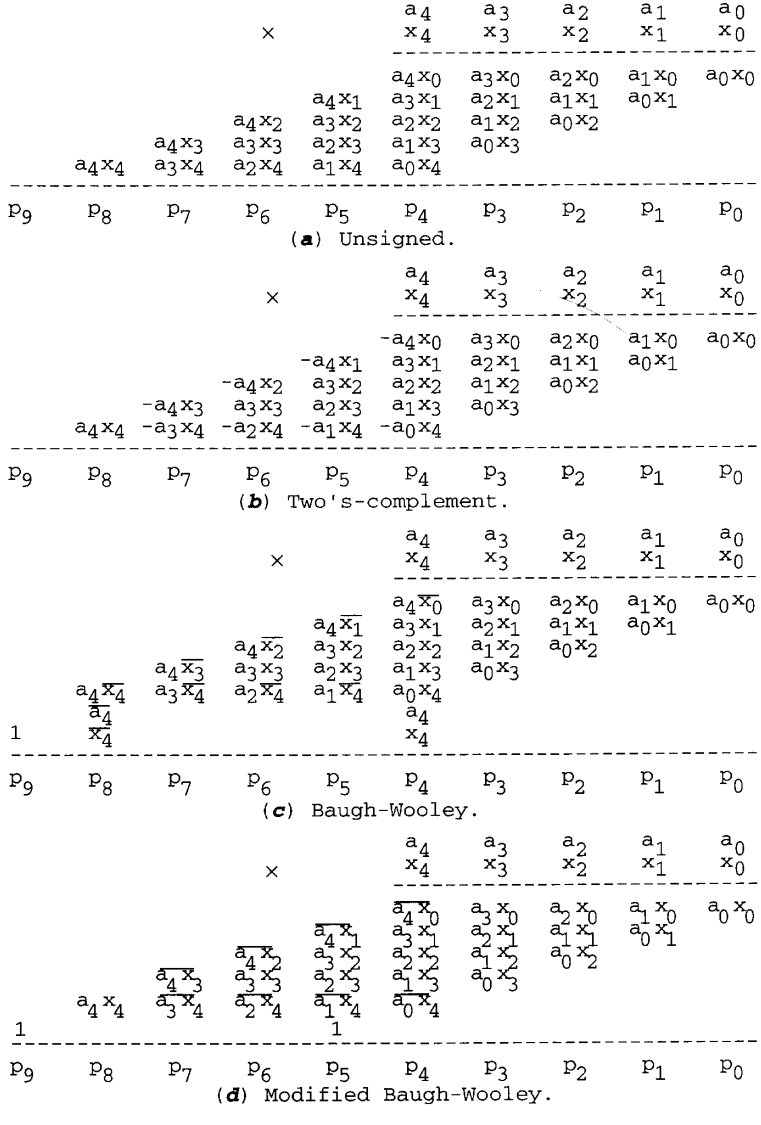


Fig. 11.8 Baugh–Wooley 2's-complement multiplication.

in the partial products matrix, Baugh and Wooley suggest that we modify the bits in the way shown in Fig. 11.8c, adding a few entries to the bit matrix in the process.

Baugh and Wooley's strategy increases the maximum column height by 2, thus potentially leading to greater delay through the CSA tree. For example, in the 5×5 multiplication depicted in Fig. 11.8, column height is increased from 5 to 7, leading to an extra CSA level. In this particular example, however, the extra delay can be avoided by removing the x_4 entry from column 4 and placing two x_4 entries in column 3 which has only four entries. This reduces the height to 6, which can still be handled by a three-level CSA tree.

To prove the correctness of the Baugh–Wooley scheme, let us focus on the entry $a_4\overline{x}_0$ in Fig. 11.8. Given that the sign bit in 2's-complement numbers has a negative weight, this entry should have been $-a_4x_0$. We note that:

$$-a_4x_0 = a_4(1 - x_0) - a_4 = a_4\overline{x}_0 - a_4$$

Hence, we can replace $-a_4x_0$ with the two entries $a_4\overline{x}_0$ and $-a_4$. If instead of $-a_4$ we use an entry a_4, the column sum increases by $2a_4$. To compensate for this, we must insert $-a_4$ in the next higher column. The same argument can be repeated for $a_4\overline{x}_1$, $a_4\overline{x}_2$, and $a_4\overline{x}_3$. Each column, other than the first, gets an a_4 and a $-a_4$, which cancel each other out. The p_8 column gets a $-a_4$ entry, which can be replaced with $\overline{a}_4 - 1$. The same argument can be repeated for the $\overline{a}_ix_4$ entries, leading to the insertion of x_4 in the p_4 column and $\overline{x}_4 - 1$ in the p_8 column. The two -1s thus produced in the eighth column are equivalent to a -1 entry in the p_9 column, which can in turn be replaced with a 1 and a borrow into the nonexistent (and inconsequential) tenth column.

Another way to justify the Baugh–Wooley method is to transfer all negatively weighted a_4x_i terms, $0 \leq i \leq 3$, to the bottom row, thus leading to two negative numbers (the preceding number and the one formed by the a_ix_4 bits, $0 \leq i \leq 3$) in the last two rows. Now, the two numbers x_4a and a_4x must be subtracted from the sum of all the positive elements. Instead of subtracting $x_4 \times a$, we add x_4 times the 2's complement of a, which consists of 1's complement of a plus x_4 (similarly for a_4x). The reader should be able to supply the other details.

A modified form of the Baugh–Wooley method, (Fig. 11.8d) is preferable because it does not lead to an increase in the maximum column height. Justifying this modified form is left as an exercise.

11.4 PARTIAL-TREE MULTIPLIERS

If the cost of a full-tree multiplier is unacceptably high for a particular application, then a variety of mixed serial-parallel designs can be considered. Let h be a number smaller than k. One idea is to perform the k-operand addition needed for $k \times k$ multiplication via $\lceil k/h \rceil$ passes through a smaller CSA tree. Figure 11.9 shows the resulting design that includes an $(h+2)$-input CSA tree for adding the cumulative partial product (in stored-carry form) and h new operands, feeding back the resulting sum and carry to be combined with the next batch of h operands.

Since the next batch of h operands will be shifted by h bits with respect to the current batch, h bits of the derived sum and $h - 1$ bits of the carry can be relaxed after each pass. These are combined using an h-bit adder to yield h bits of the final product, with the carry-out kept in a flip-flop to be combined with the next inputs. Alternatively, these relaxed bits can be kept in carry-save form by simply shifting them to the right in their respective registers and postponing the conversion to standard binary format to the very end. This is why parts of Fig. 11.9 are rendered in dotted form. The latter approach might be followed if a fast double-width adder is already available in the ALU for other reasons.

Note that the design depicted in Fig. 11.9 corresponds to radix-2^h multiplication. Thus, our discussions in Sections 10.3 and 10.4 are relevant here as well. In fact, the difference between high-radix and partial-tree multipliers is quantitative rather than qualitative (see Fig. 10.13). When h is relatively small, say up to 8 bits, we tend to view the multiplier of Fig. 11.9 as a high-radix multiplier. On the other hand, when h is a significant fraction of k, say $k/2$ or $k/4$, then we view the design as a partial-tree multiplier. In Section 11.6, we will see that a pipelined variant of the design in Fig. 11.9 can be considerably faster when h is large.

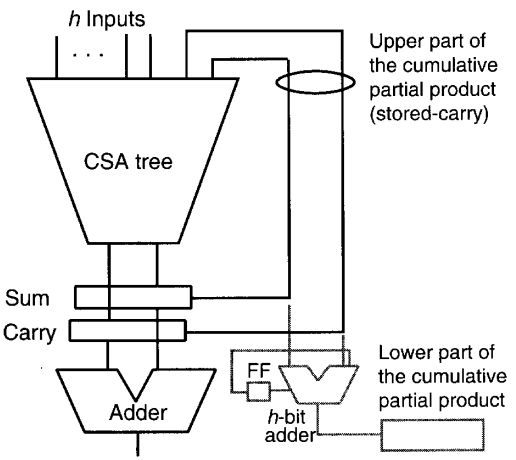


Fig. 11.9 General structure of a partial-tree multiplier.

Figure 11.9 has been drawn with the assumption of radix-2 multiplication. If radix-2^b Booth's recoding is applied first to produce one multiple for every b bits of the multiplier, then b times fewer passes are needed and bh bits can be relaxed after each pass. Thus, the small adder in Fig. 11.9 will be bh bits wide.

11.5 ARRAY MULTIPLIERS

Consider a full-tree multiplier (Fig. 11.1) in which the reduction tree is one-sided and the final adder has a ripple-carry design, as depicted in Fig. 11.10. Such a tree multiplier, which is composed of the slowest possible CSA tree and the slowest possible carry-propagate adder, is known as an array multiplier.

But why would anyone be interested in such a slow multiplier? The answer is that an array multiplier is very regular in its structure and uses only short wires that go from one full adder to horizontally, vertically, or diagonally adjacent full adders. Thus, it has a very simple and efficient layout in VLSI. Furthermore, it can be easily and efficiently pipelined by inserting latches after every CSA or after every few rows (the last row must be handled differently, as discussed in Section 11.6, because its latency is much larger than the others).

The free input of the topmost CSA in the array multiplier of Fig. 11.10 can be used to realize a multiply-add module yielding $p = ax + y$. This is useful in a variety of applications involving convolution or inner-product computation. When only the computation of ax is desired, the topmost CSA in the array multiplier of Fig. 11.10 can be removed, with $x_0 a$ and $x_1 a$ input to the second CSA directly.

Figure 11.11 shows the design of a 5×5 array multiplier in terms of full-adder cells and two-input AND gates. The sum outputs are connected diagonally, while the carry outputs are linked vertically, except in the last row, where they are chained from right to left. The design in Fig. 11.11 assumes unsigned numbers, but it can be easily converted to a 2's-complement array multiplier using the Baugh–Wooley method. This involves adding a full adder at the right end of the ripple-carry adder, to take in the a_4 and x_4 terms, and a couple of full adders at the lower left edge to accommodate the $\overline{a}_4$, $\overline{x}_4$, and 1 terms (see Fig. 11.12). Most of the connections between FA blocks in Fig. 11.12 have been removed to avoid clutter.

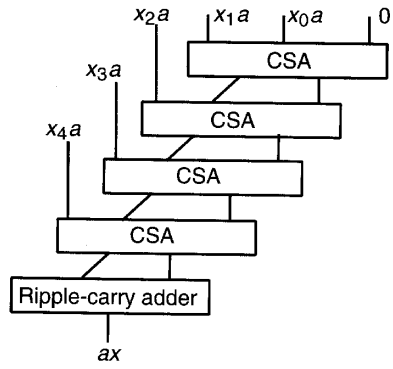


Fig. 11.10 A basic array multiplier uses a one-sided CSA tree and a ripple-carry adder.

In view of the simplicity of an array multiplier for 2's-complement numbers based on the Baugh–Wooley method (Fig. 11.12), we no longer use techniques proposed by Pezaris and others that required in some of the array positions variants of a full-adder cell capable of accommodating some negatively weighted input bits and producing one or both outputs with negative weight(s).

If we build a cell containing a full adder and an AND gate to internally form the term $a_j x_i$, the unsigned array multiplier of Fig. 11.11 turns into Fig. 11.13. Here, the x_i and a_j bits are broadcast to rows and columns of cells, with the row-i, column-j cell, forming the term $a_j x_i$ and using it as an input to its FA. If desired, one can make the design less complex by replacing the cells in the first row, or the first two rows, by AND gates.

The critical path through a $k \times k$ array multiplier, when the sum generation logic of a full-adder block has a longer delay than the carry-generation circuit, goes through the main (top left to bottom right) diagonal in Fig. 11.12 and proceeds horizontally in the last row to the p_9 output. The overall delay of the array multiplier can thus be reduced by rearranging the full-adder

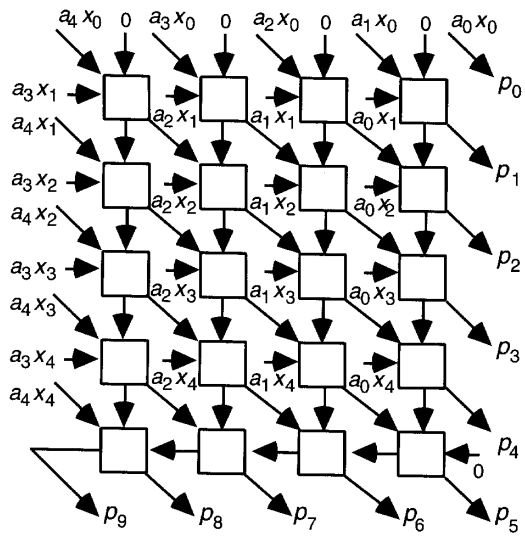


Fig. 11.11 Detailed design of a 5×5 array multiplier using full-adder blocks.

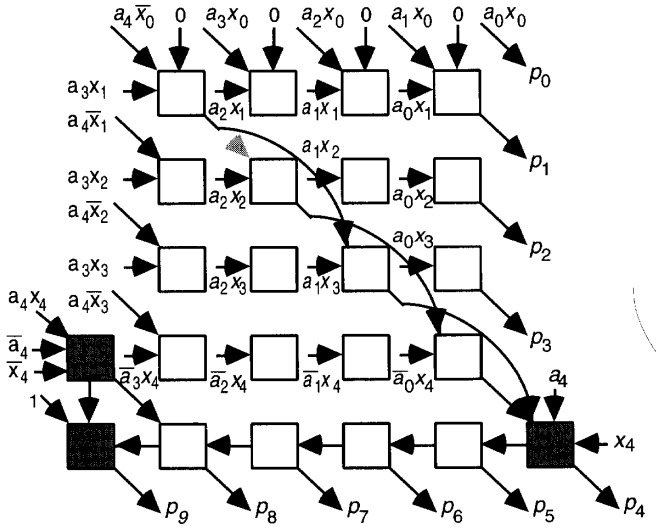


Fig. 11.12 Modifications in a 5×5 array multiplier to deal with 2's-complement inputs using the Baugh–Wooley method (inclusion of the three shaded FA blocks) or to shorten the critical path (the curved links).

inputs such that some of the sum signals skip rows (they go from row i to row $i + h$ for some $h > 1$). Figure 11.12 shows the modified connections on the main diagonal for $h = 2$. The lower right cell now has one too many inputs, but we can redirect one of them to the second cell on the main diagonal, which now has one free input. Note, however, that such skipping of levels makes for a less regular layout, which also requires longer wires, hence may not be a worthwhile modification in practice.

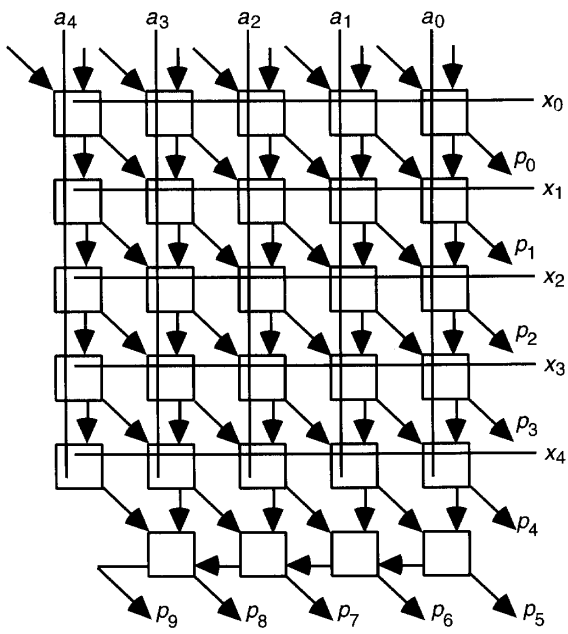


Fig. 11.13 Design of a 5×5 array multiplier with two additive inputs and full-adder blocks that include AND gates.

Since almost half the latency of an array multiplier is due to the cells in the last row, it is interesting to speculate about whether we can do the final addition faster. Obviously, it is possible to replace the last row of cells with a fast adder, but this would adversely affect the regularity of the design. Besides, even a fast adder is still much slower than the other rows, making pipelining more difficult.

To see how the ripple-carry portion of an array multiplier can be eliminated, let us arrange the k^2 terms $a_j x_i$ in a triangle, with bits distributed in $2k - 1$ columns according to the pattern:

$$1\ 2\ 3\ \cdots\ k-1\ \ k\ \ k-1\ \cdots\ 3\ 2\ 1$$

The LSB of the product is output directly, and the other bits are reduced gradually by rows of full- and half-adders (rectangular boxes in Fig. 11.14). Let us focus on the ith level and assume that the first $i - 1$ levels have already yielded two versions of the final product bits past the B_i boundary, one assuming that the next carry-save addition will produce a carry across B_i and another assuming no carry (Fig. 11.15).

At the ith level, the shaded block in Fig. 11.14 produces two versions of its sum and carry, conditional upon a future carry or no carry across B_{i+1}. The conditional sum bits from the shaded block are simply appended to the i bits coming from above. So, two versions of the upper $i + 1$ bits of the product are obtained, conditional upon the future carry across the B_{i+1} boundary. The process is then repeated in the lower levels, with each level extending the length of the conditional portion by one bit and the final multiplexer providing the last k bits of the end product in nonredundant form.

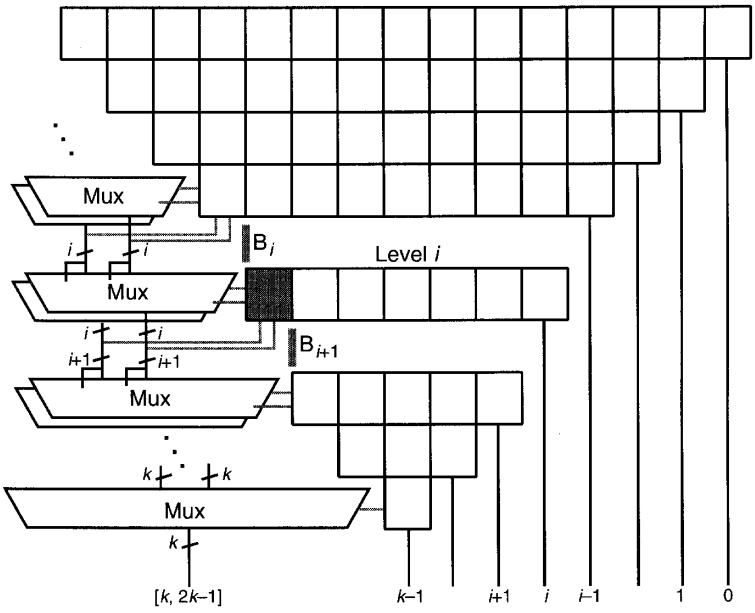


Fig. 11.14 Conceptual view of a modified array multiplier that does not need a final carry-propagate adder.

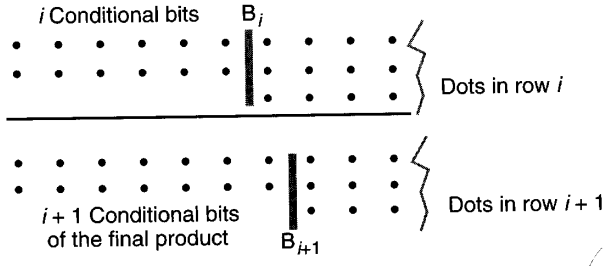


Fig. 11.15 Carry-save addition, performed in level i, extends the conditionally computed bits of the final product.

The conceptual design of Fig. 11.14 can be translated to an actual multiplier circuit after certain optimizations to remove redundant elements [Erce90], [Cimi96].

11.6 PIPELINED TREE AND ARRAY MULTIPLIERS

A full-tree multiplier can be easily pipelined. The partial products reduction tree of a full-tree multiplier is a combinational circuit that can be sliced into pipeline stages. A new set of inputs cannot be applied to the partial-tree multiplier of Fig. 11.9, however, until the sum and carry for the preceding set have been latched. Given that for large h, the depth of the tree can be significant, the rate of the application of inputs to the tree, and thus the speed of the multiplier, is limited.

Now, if instead of feeding back the tree outputs to its inputs, we feed them back into the middle of the $(h + 2)$-input tree, as shown in Fig. 11.16, the pipeline rate will be dictated by the delay through only two CSA levels rather than by the depth of the entire tree. This leads to much faster multiplication.

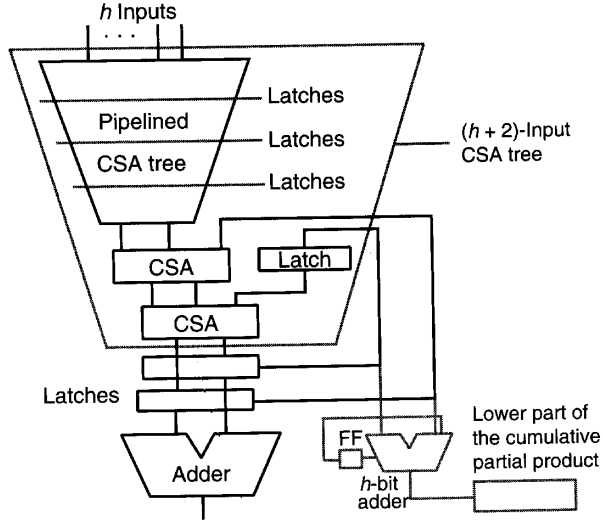


Fig. 11.16 Efficiently pipelined partial-tree multiplier.

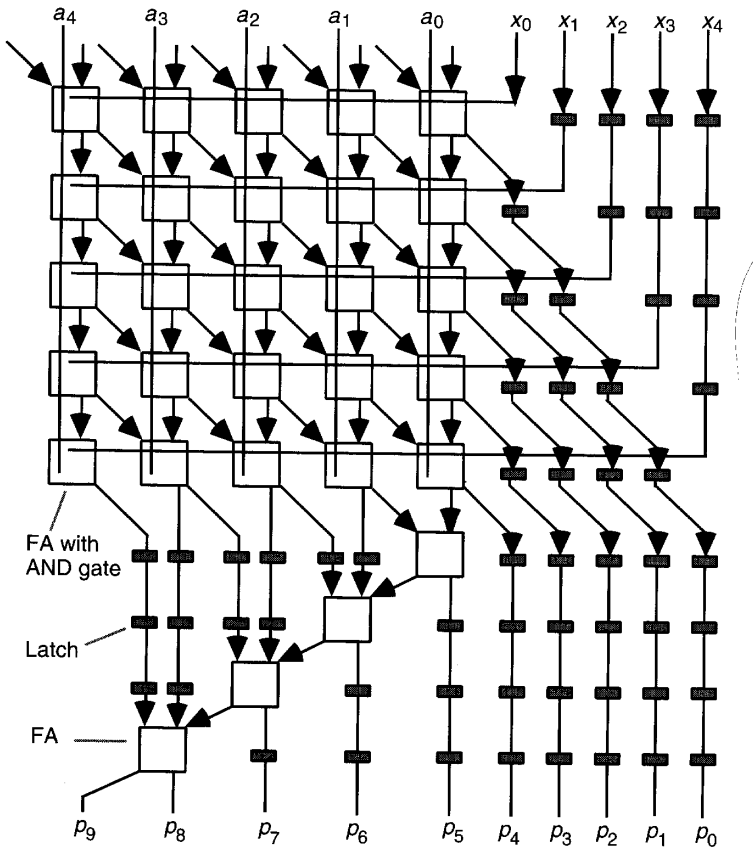


Fig. 11.17 Pipelined 5 × 5 array multiplier using latched FA blocks. The small shaded rectangles are latches.

Figure 11.17 shows one way to pipeline an array multiplier. Inputs are applied from above and the product emerges from below after nine clock cycles ($2k - 1$ in general). These FA blocks used are assumed to have output latches for both sum and carry. Note how the x_i inputs needed for the various rows of the array multiplier are delayed through the insertion of latches in their paths and how the 4-bit ripple-carry adder at the bottom row of Fig. 11.13 has been pipelined in Fig. 11.17.

PROBLEMS

11.1 Unsigned full-tree multipliers Consider the design of a 7 × 7 unsigned full-tree multiplier as depicted in Fig. 11.3.

a. Compare Figs. 11.3 and 8.12, discussing all the differences.

b. Design the required partial products reduction tree using Wallace's method.

c. Design the required partial products reduction tree using Dadda's method.

d. Compare the designs of parts a, b, and c with respect to speed and cost.

11.2 Unsigned full-tree multipliers Consider the design of an 8×8 unsigned full-tree multiplier.

- **a.** Draw a diagram similar to Fig. 11.3 to determine the number and widths of the carry-save adders required.
- **b.** Repeat part a, this time using 4-to-2 reduction circuits built of two CSAs.
- **c.** Design the required partial products reduction tree using Wallace's method.
- **d.** Design the required partial products reduction tree using Dadda's method.
- **e.** Produce one design with its final adder width between those in parts c and d.
- **f.** Compare the designs of parts a–e with respect to speed and cost.

11.3 Balanced-delay trees Find the relationship between the number n of inputs and circuit depth d of a balanced-delay tree (Fig. 11.4) and show that the depth grows as $\sqrt{n}$.

11.4 Variations in full-tree multipliers Tabulate the number of full-adder levels in a tree that reduces k multiples of the multiplicand to 2, for $4 \leq k \leq 1024$, using:

- **a.** Carry-save adders as the basic elements.
- **b.** Elements, internally built from two CSA levels, that reduce four operands to two.
- **c.** Same elements as in part b, except that in the first level of the tree only, the use of CSAs is allowed (this is helpful, e.g., for $k = 24$).
- **d.** Discuss the implications of the results of parts a–c in the design of full-tree multipliers.

11.5 Tree multiplier with Booth's recoding We need a 12×12 signed-magnitude binary multiplier. Design the required 11×11 unsigned multiplication circuit by first generating a recoded version of the multiplier having six radix-4 digits in $[-2, 2]$ and then adding the six partial products represented in 2's-complement form by a minimal network of FAs. *Hint:* 81 FAs should do.

11.6 Modified Baugh–Wooley method Prove that the modified Baugh–Wooley method for multiplying 2's-complement numbers, shown in Fig. 11.8d, is correct.

11.7 Signed full-tree multipliers Consider the design of an 8×8 full-tree multiplier for 2's-complement inputs.

- **a.** Draw a diagram similar to Fig. 11.3 to determine the number and widths of the carry-save adders required if the operands are to be sign-extended (Fig. 11.7).
- **b.** Design the 8×8 multiplier using the Baugh–Wooley method.
- **c.** Design the 8×8 multiplier using the modified Baugh–Wooley method.
- **d.** Compare the designs of parts a–c with respect to speed and cost.

11.8 Partial-tree multipliers In Fig. 11.9, the tree has been drawn with no intermediate output corresponding to the lower-order bits of the sum of its $h + 2$ inputs. If h is large, a few low-order bits of the sum will likely become available before the final sum and carry results. How does this affect the h-bit adder delineated by dotted lines?

11.9 Pezaris array multiplier Consider a 5×5 array multiplier, similar to that in Fig. 11.11 but with 2's-complement inputs, and view the AND terms $a_4 x_i$ and $a_j x_4$ as being negatively weighted. Consider also two modified forms of a full-adder cell: FA′ has one negatively weighted input, producing a negatively weighted sum and a positively weighted carry, while FA″ has two negatively weighted inputs, producing a negative carry and a positive sum. Design a 5×5 Pezaris array multiplier using FA, FA′, and FA″ cells as needed, making sure that any negatively weighted output is properly connected to a negatively weighted input (use small "bubbles" to mark negatively weighted inputs and outputs on the various blocks). Note that FA‴, with all three inputs and two outputs carrying negative weights, is the same as FA. Note also that the output must have only one negatively weighted bit at the sign position.

11.10 Two's-complement array multipliers Consider the design of an 5×5 2's-complement array multiplier. Assume that an FA block has latencies of T_c and T_s ($T_c < T_s < 2T_c$) for its carry and sum outputs.

a. Find the overall latency for the 5×5 array multiplier with the Baugh–Wooley method (Fig. 11.12, regular design without row skipping).

b. Repeat part a with the modified Baugh–Wooley method.

c. Compare the designs in parts a and b and discuss.

d. Generalize the preceding results and comparison to the case of $k \times k$ array multipliers.

11.11 Array multipliers Design array multipliers for the following number representations.

a. Binary signed-digit numbers using the digit set $[-1, 1]$ in radix 2.

b. One's-complement numbers.

11.12 Multiply-add modules Consider the design of a module that performs the computation $p = ax + y + z$, where a and y are k-bit unsigned integers and x and z are l-bit unsigned integers.

a. Show that p is representable with $k + l$ bits.

b. Design a tree multiplier to compute p for $k = 8$ and $l = 4$ based on a Wallace tree and a CPA.

c. Repeat part b using a Dadda tree.

d. Show that an 8×4 array multiplier can be readily modified to compute p.

11.13 Pipelined array multipliers Consider the 5×5 pipelined array multiplier in Fig. 11.17.

a. Show how the four lowermost FAs and the latches immediately above them can be replaced by a number of latched HAs. *Hint:* Some HAs will have to be added in the leftmost column, corresponding to p_9, which currently contains no element.

b. Compare the design in part a with the original design in Fig. 11.17.

c. Redesign the pipelined multiplier in Fig. 11.17 so that the combinational delay in each pipeline stage is equal to two FA delays (ignore the difference in delays between the sum and carry outputs).

d. Repeat part c for the array multiplier derived in part a.

e. Compare the array multiplier designs of parts c and d with respect to throughput and throughput/cost. State your assumptions clearly.

11.14 Effectiveness of Booth's recoding As mentioned at the end of Section 11.2, the effectiveness of Booth recoding in tree multipliers has been questioned [Vill93]. Booth's recoding essentially reduces the number of partial products by a factor of 2. A (4, 2) reduction circuit built (e.g.) from two CSAs offers the same reduction. Show through a simple approximate analysis of the delay and cost of a $k \times k$ unsigned multiplier based on Booth's recoding and (4, 2) initial reduction that Booth's recoding has the edge in terms of gate count but that it may lose on other counts. Assume, for simplicity, that k is even.

11.15 VLSI implementation of tree multipliers Wallace and Dadda trees tend to be quite irregular and thus ill-suited to compact VLSI implementation. Study the bit-slice implementation method for tree multipliers suggested in [Mou92] and apply it to the design of a 12×12 multiplier.

11.16 Faster array multipliers Present the complete design of an 8×8 array multiplier built without a final carry-propagate adder (Fig. 11.14). Compare the resulting design to a simple 8×8 array multiplier with respect to speed, cost, and cost-effectiveness.

11.17 Pipelined partial-tree multipliers

a. Would it be cost-effective to implement an 8×8 unsigned multiplier using the pipelined design of Fig. 11.16 with $h = 4$?

b. With reference to the VLSI complexity discussions in Section 10.6, show that the multiplication time in a pipelined partial-tree multiplier is $O(k/h + \log k)$.

REFERENCES

[Baug73] Baugh, C. R., and B. A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Trans. Computers*, Vol. 22, pp. 1045–1047, December 1973.

[Cimi96] Ciminiera, L., and P. Montuschi, "Carry-Save Multiplication Schemes Without Final Addition," *IEEE Trans. Computers*, Vol. 45, No. 9, pp. 1050–1055, 1996.

[Dadd65] Dadda, L., "Some Schemes for Parallel Multipliers," *Alta Frequenza*, Vol. 34, pp. 349–356, 1965.

[Erce90] Ercegovac, M. D., and T. Lang, "Fast Multiplication Without Carry-Propagate Addition," *IEEE Trans. Computers*, Vol. 39, No. 11, pp. 1385–1390, 1990.

[Mou92] Mou, Z.-J., and F. Jutand, " 'Overturned-Stairs' Adder Trees and Multiplier Design," *IEEE Trans. Computers*, Vol. 41, No. 8, pp. 940–948, 1992.

[Parh96] Parhami, B., "Comments on 'High-Speed Area-Efficient Multiplier Design Using Multiple-Valued Current Mode Circuits'," *IEEE Trans. Computers*, Vol. 45, No. 5, pp. 637–638, 1996.

[Peza71] Pezaris, S. D., "A 40-ns 17-Bit by 17-Bit Array Multiplier," *IEEE Trans. Computers*, Vol. 20, pp. 442–447, April 1971.

[Taka85] Takagi, N., H. Yasuura, and S. Yajima, "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," *IEEE Trans. Computers*, Vol. 34, No. 9, pp. 789–796, 1985.

[Vill93] Villager, D., and V. G. Oklobdzija, "Analysis of Booth Encoding Efficiency in Parallel Multipliers Using Compressors for Reduction of Partial Products," *Proc. Asilomar Conf. Signals, Systems, and Computers*, pp. 781–784, 1993.

[Vuil83] Vuillemin, J., "A Very Fast Multiplication Algorithm for VLSI Implementation," *Integration: The VLSI Journal*, Vol. 1, pp. 39–52, 1983.

[Wall64] Wallace, C. S., "A Suggestion for a Fast Multiplier," *IEEE Trans. Electronic Computers*, Vol. 13, pp. 14–17, 1964.

[Zura86] Zuras, D., and W. H. McAllister, "Balanced Delay Trees and Combinatorial Division in VLSI," *IEEE J. Solid-State Circuits*, Vol. 21, pp. 814–819, October 1986.

Chapter 12 VARIATIONS IN MULTIPLIERS

We do not always synthesize our multipliers from scratch but may desire, or be required, to use building blocks such as adders, small multipliers, or lookup tables. Furthermore, limited chip area and/or pin availability may dictate the use of bit-serial designs. In this chapter, we discuss such variations and also deal with modular multipliers, the special case of squaring, and multiply-accumulators. Chapter topics include:

12.1 DIVIDE-AND-CONQUER DESIGNS

Suppose you have $b \times b$ multipliers and would like to use them to synthesize a $2b \times 2b$ multiplier. Denoting the high and low halves of the multiplicand (multiplier) by a_H and a_L (x_H and x_L), we can use four $b \times b$ multipliers to compute the four partial products $a_L x_L$, $a_L x_H$, $a_H x_L$, and $a_H x_H$ as shown in Fig. 12.1. These four values must then be added to obtain the final product. Actually, as shown on the right side of Fig. 12.1, only three values need to be added, since the nonoverlapping partial products $a_H x_H$ and $a_L x_L$ can be viewed as a single $4b$-bit number.

We see that our original $2b \times 2b$ multiplication problem has been reduced to four $b \times b$ multiplications and a three-operand addition problem. The $b \times b$ multiplications can be performed by smaller hardware multipliers or via table lookup. Then, we can compute the $4b$-bit product by means of a single level of carry-save addition, followed by a $3b$-bit carry-propagate addition. Note that b bits of the product are directly available following the $b \times b$ multiplications.

Larger multipliers, such as $3b \times 3b$ or $4b \times 4b$, can be similarly synthesized from $b \times b$-multiplier building blocks. Figure 12.2 shows that $3b \times 3b$ multiplication leads to five numbers, while $4b \times 4b$ multiplication produces seven numbers. Hence, we can complete the multiplication

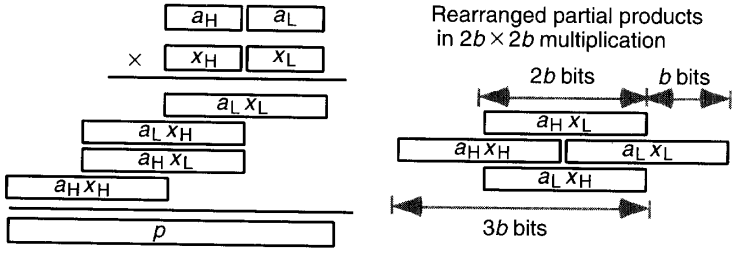


Fig. 12.1 Divide-and-conquer strategy for synthesizing a $2b \times 2b$ multiplier from $b \times b$ multipliers.

process in these two cases by using a row of (5; 2)- or (7; 2)-counters, followed by a $5b$- or $7b$-bit fast adder, respectively. Note that b bits of the product are obtained directly from a small multiplier.

For example, given 4×4 multipliers as building blocks, we can synthesize a 16×16 multiplier using 16 of the small multipliers, along with 24 (7; 2)-counters and a 28-bit fast adder. The structure of a 32×32 multiplier built of 8×8-multiplier building blocks is identical to the device above.

One can view the preceding divide-and-conquer scheme, depicted in Figs. 12.1 and 12.2, as radix-2^b multiplication, except that each radix-2^b digit of the multiplier produces several partial products, one for each radix-2^b digit of the multiplicand, instead just one.

For $2b \times 2b$ multiplication, one can use b-bit adders exclusively to accumulate the partial products, as shown in Fig. 12.3 for $b = 4$. The pair $[i, j]$ of numbers shown next to a solid line in Fig. 12.3 indicate that the 4-bit bundle of wires represented by that line spans bit positions i through j. A dotted line represents one bit, with its positions given by a single integer. We need five b-bit adder blocks, arranged in a circuit of depth 4, to perform the accumulation. This is attractive if b-bit adders are available as economical, off-the-shelf components. The resulting design is not much slower than the design based on CSA reduction if the latter design uses a cascade of three b-bit adders for the final $3b$-bit addition.

Instead of $b \times b$ multipliers, one can use $b_1 \times b_2$ multipliers. For example, with 8×4 multipliers as building blocks, a 16×16 multiplier can be synthesized from eight such units, followed by a 5-to-2 reduction circuit and a 28-bit adder.

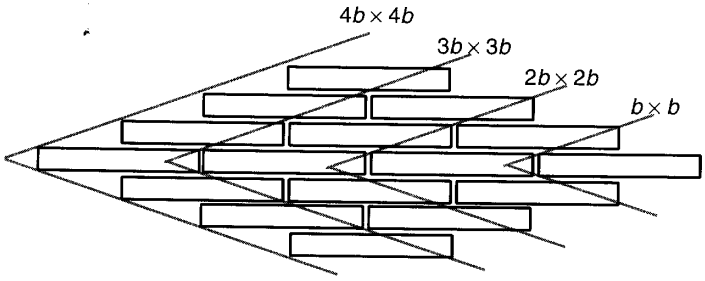


Fig. 12.2 Using $b \times b$ multipliers to synthesize $2b \times 2b$, $3b \times 3b$, and $4b \times 4b$ multipliers.

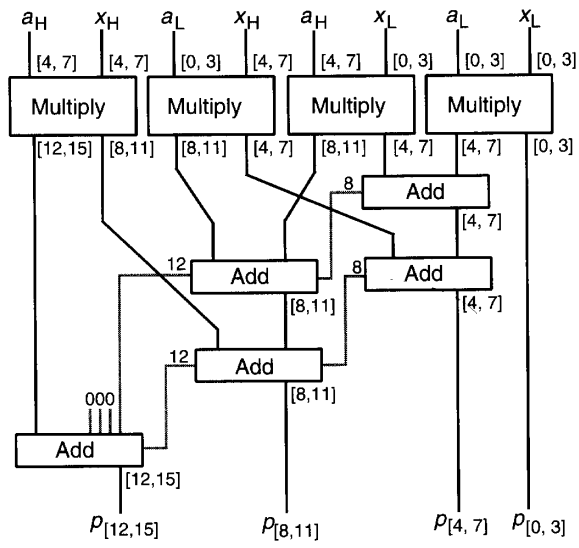


Fig. 12.3 Using 4 × 4 multipliers and 4-bit adders to synthesize an 8 × 8 multiplier.

12.2 ADDITIVE MULTIPLY MODULES

We note from the discussion in Section 12.1, and Fig. 12.3 in particular, that synthesizing large multipliers from smaller ones requires both multiplier and adder units. If we can combine the multiplication and addition functions into one unit, then perhaps a single module type will suffice for implementing such multipliers. This is the idea behind additive multiply modules (AMMs).

The additive multiply module in Fig. 12.4a, performs the computation $p = ax + y + z$, where a and y are 4-bit numbers and x and z are 2-bit numbers. The maximum value of the result p is $(15 \times 3) + 15 + 3 = 63$, which can be represented with 6 bits. Figure 12.4b shows an implementation of this AMM using four full adders (boxes enclosing three dots) and a 4-bit adder in dot notation.

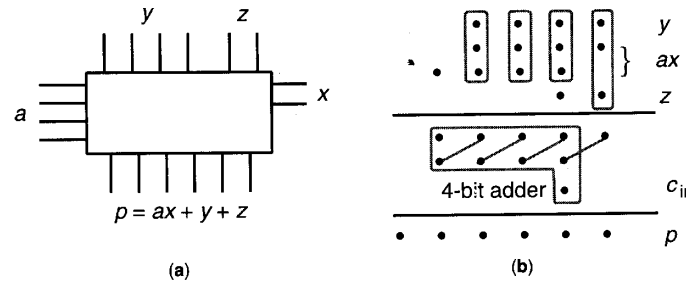


Fig. 12.4 Additive multiply module with 4 × 2 multiplier (ax) plus 4-bit and 2-bit additive inputs (y and z).

Figure 12.5 shows how the 8 × 8 multiplier example of Fig. 12.3 can be built from eight AMMs of the type depicted in Fig. 12.4. Note that eight 4 × 2 multipliers would have been needed for this design; so the number of modules is kept to a minimum. Each AMM is slower than a 4 × 2 multiplier by at most one full-adder level. So, the delay in Fig. 12.5 that is attributable to the addition function is no more than six FA delays (the critical path goes through six AMM modules). Thus, given that the cost of a 4 × 2 AMM is less than the combined costs of a 4 × 2 multiplier and a 4-bit adder, the design shown in Fig. 12.5 is very cost-effective.

Figure 12.6 depicts an alternate design for an 8 × 8 multiplier using the same number and type of 4 × 2 AMMs as in Fig. 12.5 (as well as the same notational conventions). This latter design is slower that the design of Fig. 12.5 because its critical path goes through all eight modules. However, it is more regular and, thus, readily generalizable to any $4h_2 \times 2h_1$ multiplier with compact VLSI layout.

In general, a $b \times c$ AMM will have a pair of b-bit and c-bit multiplicative inputs, two b-bit and c-bit additive inputs, and a $(b+c)$-bit output. The number of bits in the output is just adequate to represent the largest possible output value, as evident from the following identity:

$$(2^b - 1)(2^c - 1) + (2^b - 1) + (2^c - 1) = 2^{b+c} - 1$$

In designing larger multipliers based on $b \times c$ AMMs, the $(b + c)$-bit output of each AMM is divided into a b-bit upper part and a c-bit lower part that are supplied as additive inputs to other AMMs or serve as primary outputs. An AMM that receives $a_{[j,j+b-1]}$ and $x_{[i,i+c-1]}$ as its multiplicative inputs should have values spanning the bit positions $[i + j, i + j + b - 1]$ and $[i + j, i + j + c - 1]$ as its additive inputs (why?). To design a $k \times l$ multiplier, where b and

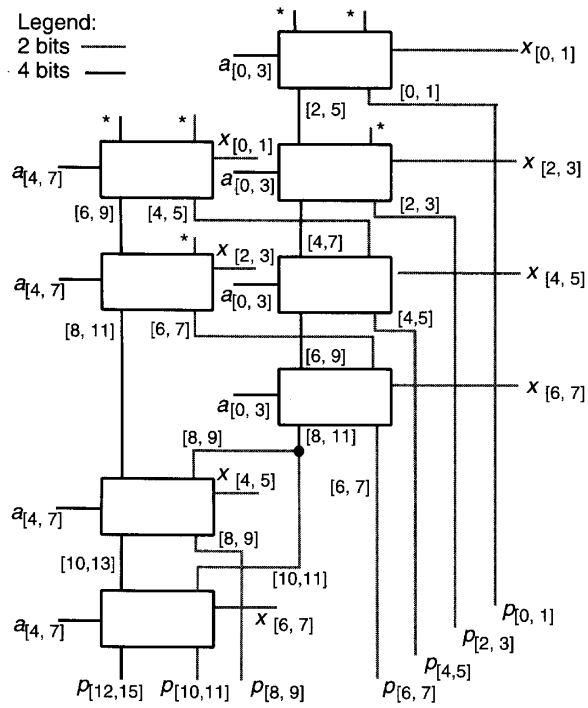


Fig. 12.5 An 8 × 8 multiplier built of 4 × 2 additive multiply modules. Inputs marked with an asterisk carry 0s.

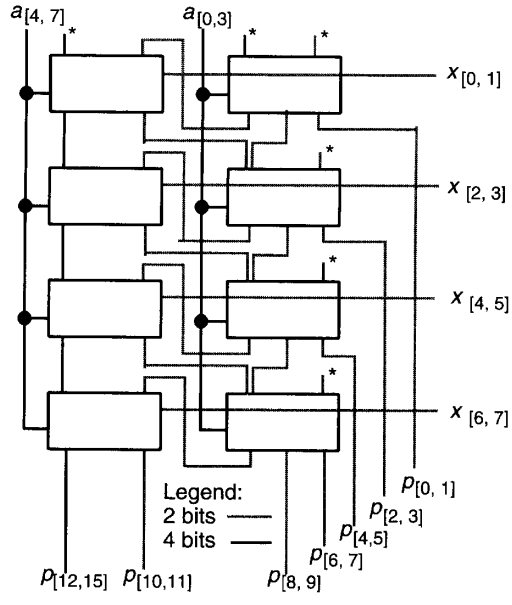


Fig. 12.6 Alternate 8 × 8 multiplier design based on 4 × 2 AMMs. Inputs marked with an asterisk carry 0s.

c divide both k and l, one can organize the $kl/(bc)$ AMMs as a $(k/b) \times (l/c)$ or a $(k/c) \times (l/b)$ array. This provides some flexibility in fitting the design to the available chip area. However, the choice may have nontrivial implications for speed.

12.3 BIT-SERIAL MULTIPLIERS

Bit-serial arithmetic is attractive in view of its smaller pin count, reduced wire length, and lower floor space requirements in VLSI. In fact, the compactness of the design may allow us to run a bit-serial multiplier at a clock rate high enough to make the unit almost competitive with much more complex designs with regard to speed. In addition, in certain application contexts inputs are supplied bit-serially anyway. In such a case, using a parallel multiplier would be quite wasteful, since the parallelism may not lead to any speed benefit. Furthermore, in applications that call for a large number of independent multiplications, multiple bit-serial multipliers may be more cost-effective than a complex highly pipelined unit.

Bit-serial multipliers can be designed as systolic arrays: synchronous arrays of processing elements that are interconnected by only short, local wires thus allowing very high clock rates. Let us begin by introducing a semisystolic multiplier, so named because its design involves broadcasting a single bit of the multiplier x to a number of circuit elements, thus violating the "short, local wires" requirement of pure systolic design [Kung82].

Figure 12.7 shows a semisystolic 4 × 4 multiplier. The multiplicand a is supplied in parallel from above and the multiplier x is supplied bit-serially from the right, with its least significant bit arriving first. Each bit x_i of the multiplier is multiplied by a and the result added to the cumulative partial product, kept in carry-save form in the carry and sum latches. The carry bit stays in its current position, while the sum bit is passed on to the neighboring cell on the

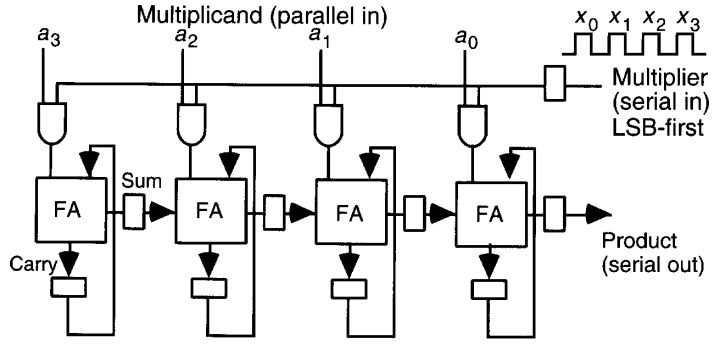


Fig. 12.7 Semisystolic circuit for 4 × 4 multiplication in eight clock cycles.

right. This corresponds to shifting the partial product to the right before the next addition step (normally the sum bit would stay put and the carry bit would be shifted to the left). Bits of the result emerge serially from the right as they become available.

A k-bit unsigned multiplier x must be padded with k zeros to allow the carries to propagate to the output, yielding the correct $2k$-bit product. Thus, the semisystolic multiplier of Fig. 12.7 can perform one $k \times k$ unsigned integer multiplication every $2k$ clock cycles. If k-bit fractions need to be multiplied, the first k output bits are discarded or used to properly round the most significant k bits.

To make the multiplier of Fig. 12.7 fully systolic, we must remove the broadcasting of the multiplier bits. This can be accomplished by a process known as systolic retiming, which is briefly explained below.

Consider a synchronous (clocked) circuit, with each line between two functional parts having an integral number of unit delays (possibly 0). Then, if we cut the circuit into two parts c_L and c_R, we can delay (advance) all the signals going in one direction and advance (delay) the ones going in the opposite direction by the same amount without affecting the correct functioning or external timing relations of the circuit. Of course, the primary inputs and outputs to the two parts c_L and c_R must be correspondingly advanced or delayed, too (see Fig. 12.8).

For the retiming shown in Fig. 12.8 to be possible, all the signals that are advanced by d must have had original delays of d or more (negative delays are not allowed). Note that all the

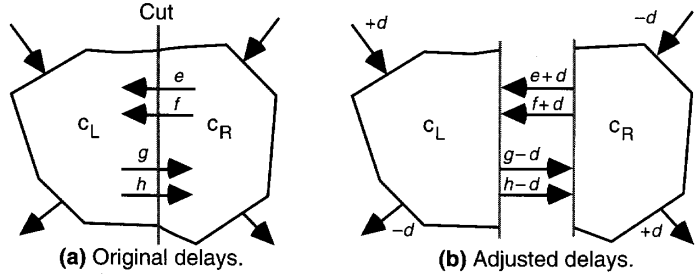


Fig. 12.8 Example of retiming by delaying the inputs to c_L and advancing the outputs from c_L by d units.

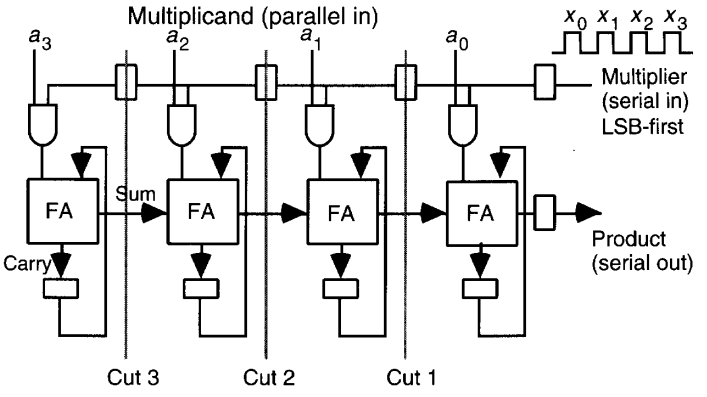


Fig. 12.9 A retimed version of our semisystolic multiplier.

signals going into c_L have been delayed by d time units. Thus, c_L will work as before, except that everything, including output production, occurs d time units later than before retiming. Advancing the outputs by d time units will keep the external view of the circuit unchanged.

We apply the preceding process to the multiplier circuit of Fig. 12.7 in three successive steps corresponding to cuts 1, 2, and 3 in Fig. 12.9, each time delaying the left-moving signal by one unit and advancing the right-moving signal by one unit. Verifying that the multiplier in Fig. 12.9 works correctly is left as an exercise. This new version of our multiplier does not have the fan-out problem of the design in Fig. 12.7 but it suffers from long signal propagation delay through the four FAs in each clock cycle, leading to inferior operating speed. Note that the culprits are zero-delay lines that lead to signal propagation through multiple circuit elements.

One way of avoiding zero-delay lines in our design is to begin by doubling all the delays in Fig. 12.7. This is done by simply replacing each of the sum and carry flip-flops with two cascaded flip-flops before retiming is applied. Since the circuit is now operating at half its original speed, the multiplier x must also be applied on alternate clock cycles. The resulting design in Fig. 12.10 is fully systolic, inasmuch as signals move only between adjacent cells in each clock cycle. However, twice as many cycles are needed.

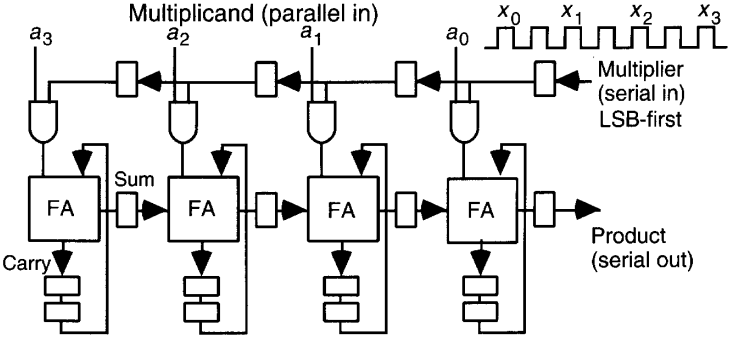


Fig. 12.10 Systolic circuit for 4 × 4 multiplication in 15 cycles.

The easiest way to derive a multiplier with both inputs entering bit-serially is to allow k clock ticks for the multiplicand bits to be put into place in a shift register and then use the design of Fig. 12.7 (or its fully systolic counterpart in Fig. 12.10) to compute the product. This increases the total delay by k cycles.

An alternative bit-serial input/output design is obtained by writing the relationship between the output and inputs in the form of a recurrence and then implementing it in hardware. Let $a^{(i)}$ and $x^{(i)}$ denote the values of a and x up to bit position i ($a^{(0)} = a_0$, $a^{(1)} = (a_1 a_0)_{\text{two}}$, etc.). Assume that the k-bit, 2's-complement inputs are sign-extended to $2k$ bits. Define the partial product $p^{(i)}$ as follows:

$$p^{(i)} = 2^{-(i+1)} a^{(i)} x^{(i)}$$

Then, given that $a^{(i)} = 2^i a_i + a^{(i-1)}$ and $x^{(i)} = 2^i x_i + x^{(i-1)}$, we have:

$$\begin{aligned} 2p^{(i)} &= 2^{-i}(2^i a_i + a^{(i-1)})(2^i x_i + x^{(i-1)}) \\ &= p^{(i-1)} + a_i x^{(i-1)} + x_i a^{(i-1)} + 2^i a_i x_i \end{aligned}$$

Thus, if $p^{(i-1)}$ is stored in double-carry-save form (three rows of dots in dot notation, as opposed to two for ordinary carry-save), it can be combined with the terms $a_i x^{(i-1)}$ and $x_i a^{(i-1)}$ using a (5; 3)-counter to yield a double-carry-save result for the next step. The final term $2^i a_i x_i$ has a single 1 in the ith position where all the other terms have 0s. Thus it can be handled by using a multiplexer (Fig. 12.11). In cycle i, a_i and x_i are input and stored in the ith cell (the correct timing is achieved by a "token" t, which is provided to cell 0 at time 0 and is then shifted leftward with each clock tick). The terms $a^{(i-1)}$ and $x^{(i-1)}$, which are already available in registers, are ANDed with x_i and a_i, respectively, and supplied along with the three bits of $p^{(i-1)}$ as inputs to the (5; 3)-counter. Figures 12.11 and 12.12 show the complete cell design and cell interconnection [Ienn94]. The AND gate computing $a_i x_i$ is replicated in each cell for the sake of uniformity. A single copy of this gate could be placed outside the cells, with its output broadcast to all cells.

Note that the 3-bit sum of the five inputs to the (5; 3)-counter is shifted rightward before being stored in latches by connecting its LSB to the right neighboring cell, keeping its middle

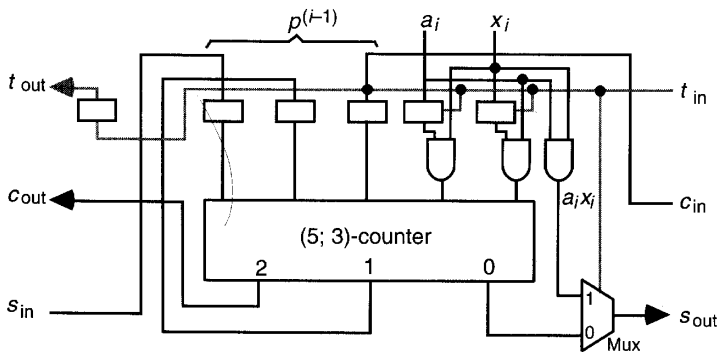


Fig. 12.11 Building block for a latency-free, bit-serial multiplier.

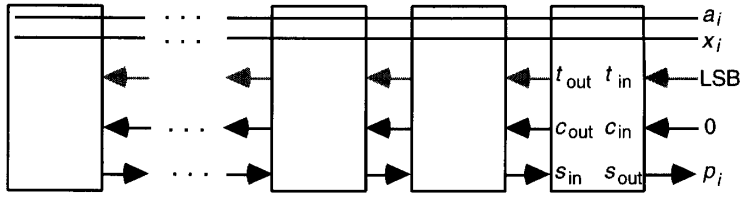


Fig. 12.12 The cellular structure of the bit-serial multiplier based on the cell in Fig. 12.11.

bit in place, and shifting its MSB to the left. The product becomes available bit-serially at the s_{out} output of the rightmost cell. Only $k - 1$ such cells are needed to compute the full $2k$-bit product of two k-bit numbers. The reason is that the largest intermediate partial product is $2k - 1$ bits wide, but by the time we get to this partial product, k bits of the product have already been produced and shifted out.

Figure 12.13 uses dot notation to show the justification for the bit-serial multiplier design above. Figure 12.13a depicts the meanings of the various partial operands and results, while Fig. 12.13b represents the operation of the (5; 3)-counters. Note, in particular, how the dot representing $a_i x_i$ is transferred to the s_{out} output by the cell holding the token (refer to the lower right corner of Fig. 12.11).

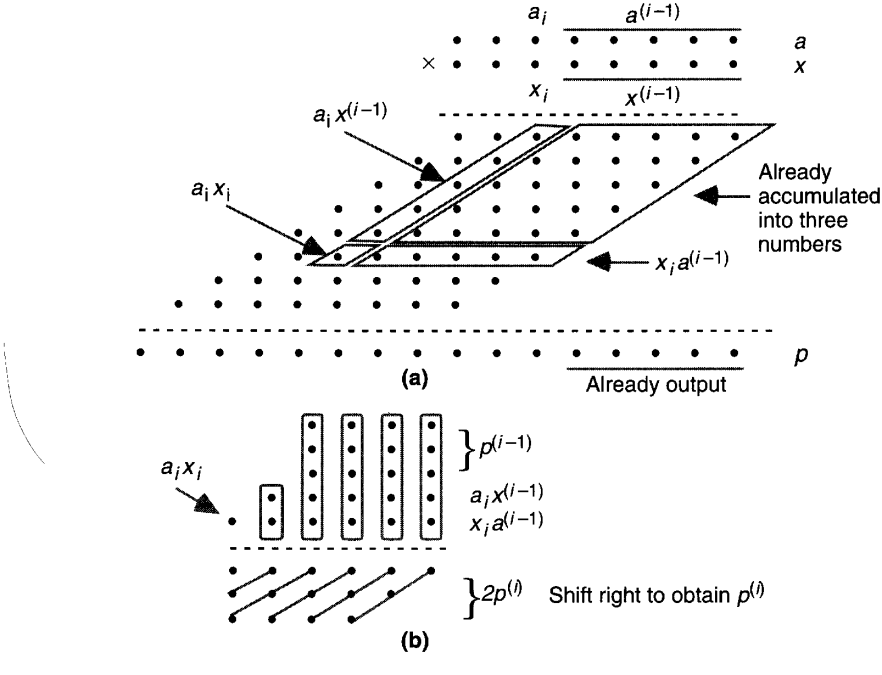


Fig. 12.13 Bit-serial multiplier design in dot notation.

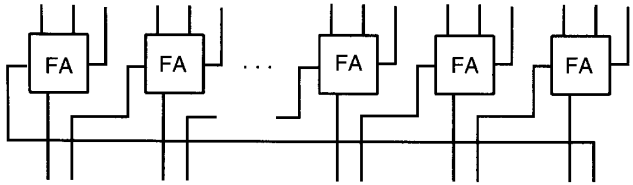


Fig. 12.14 Modulo-$(2^b - 1)$ carry-save adder.

12.4 MODULAR MULTIPLIERS

A modular multiplier is one that produces the product of two (unsigned) integers modulo some fixed constant m. It is useful, for example, for implementing the multiplication operation for residue number systems. A modular multiplier could be implemented by attaching a modular reduction circuit to the output of a standard binary multiplier. However, simpler designs are often possible if the modular reduction is combined with the accumulation of partial products. In particular, this approach obviates the need for keeping longer intermediate values.

The two special cases of $m = 2^b$ and $m = 2^b - 1$ are, as usual, simpler to deal with. For example, if the partial products are accumulated through carry-save addition, then for $m = 2^b$, the modular version simply ignores the carry output of the full adder in position $b - 1$ and for $m = 2^b - 1$, the carry out of position $b - 1$ is combined with bits in column 0 (Fig. 12.14).

As an example, consider the design of a modulo-15 multiplier for 4-bit operands. Since $16 = 1 \bmod 15$, the six heavy dots in the dotted triangle in the upper left corner of Fig. 12.15 can be moved as shown, leading to the square partial products matrix on the lower left. The four 4-bit values can then be reduced by two levels of CSA (with wraparound links, as in Fig. 12.14) followed by a 4-bit adder (again with end-around carry). We see that this particular modular multiplier is in fact simpler than a standard 4×4 binary multiplier.

Similar techniques can be used to handle modular multiplication in the general case. For example, a modulo-13 multiplier can be designed by using the identities $16 = 3 \bmod 13$, $32 = 6 \bmod 13$, and $64 = 12 \bmod 13$. Each dot inside the triangle in Fig. 12.15 must now be replaced with two dots in the four lower-order columns (Fig. 12.16). Thus, some complexity is added

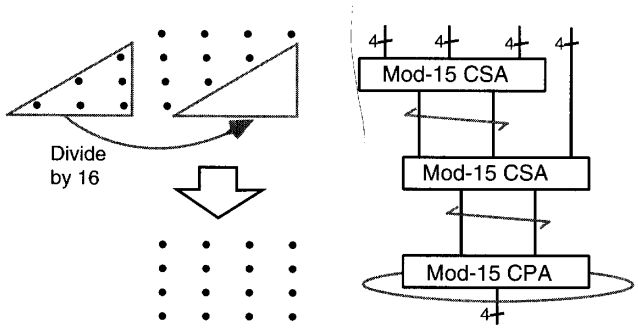


Fig. 12.15 Design of a 4×4 modulo-15 multiplier.

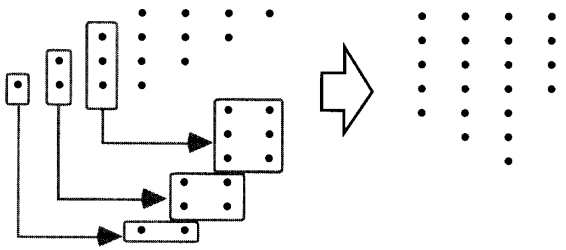


Fig. 12.16 One way to design of a 4 × 4 modulo-13 multiplier.

in view of the larger number of dots to be reduced and the need for the final adjustment of the result to be in [0, 12].

To complete the design of our 4 × 4 modulo-13 multiplier, the values shown on the right-hand side of Fig. 12.16 must be added modulo 13. After a minor simplification, consisting of removing one dot from column 1 and replacing it with two dots in column 0, a variety of methods can be used for the required modular multioperand addition [Pies94].

For example, one can use a CSA tree in which carries into column 4 are reinserted into columns 0 and 1. However, this scheme will not work toward the end of the process and must thus be supplemented with a different modular reduction scheme. Another approach is to keep some of the bits emerging from the left end (e.g., those that cannot be accommodated in the dot matrix without increasing its height) and reduce them modulo 13 by means of a lookup table or specially designed logic circuit. Supplying the details is left as an exercise. Figure 12.17 shows a general method for converting an n-input modulo-m addition problem to a three-input problem.

12.5 THE SPECIAL CASE OF SQUARING

Any standard or modular multiplier can be used for computing $p = x^2$ if both its inputs are connected to x. However, a special-purpose k-bit squarer, if built in hardware, will be significantly lower in cost and delay than a $k \times k$ multiplier.

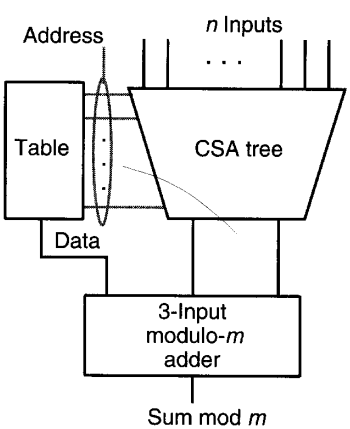


Fig. 12.17 A general method for modular multioperand addition.

To see why, consider the problem of squaring a 5-bit unsigned binary integer $(x_4x_3x_2x_1x_0)_{\text{two}}$. As shown in Fig. 12.18a, the partial products matrix can be considerably simplified before performing multioperand addition. A term x_ix_i reduces to x_i and a pair of terms x_ix_j and x_jx_i in any given column can be replaced by x_ix_j in the next higher column. The resulting simplified partial products matrix for our 5-bit example is shown in Fig. 12.18b. We see that the two least significant bits of the square are obtained with no effort and that computing the remaining bits involves a three-operand addition as opposed to a five-operand addition needed for 5×5 multiplication.

Further simplifications and fine-tuning are often possible. For example, based on the identities

$$\begin{aligned} x_1x_0 + x_1 &= 2x_1x_0 + x_1 - x_1x_0 \\ &= 2x_1x_0 + x_1(1 - x_0) \\ &= 2x_1x_0 + x_1\bar{x}_0 \end{aligned}$$

we can remove the two terms x_1x_0 and x_1 from column 2, replacing them by $x_1\bar{x}_0$ in column 2 and x_1x_0 in column 3. This transformation reduces the width of the final carry-propagate adder from 7 to 6 bits. Similar substitutions can be made for the terms in columns 4 and 6, but they do not lead to any simplification or speedup in this particular example.

For a short word width k, the square of a k-bit number can be easily obtained from a $2^k \times (2k - 2)$ lookup table, whereas a much larger table would be needed for multiplying two k-bit numbers. In fact, two numbers can be multiplied based on two table-lookup evaluations of the square function, and three additions, using the identity $ax = [(a + x)^2 - (a - x)^2]/4$.

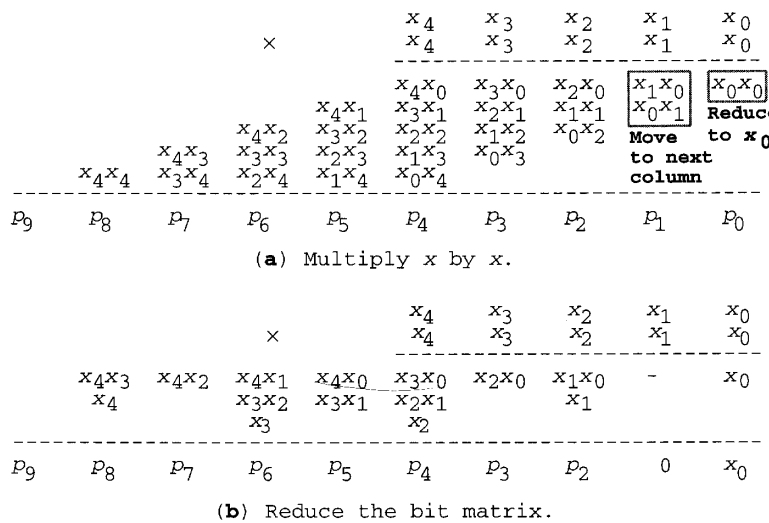


Fig. 12.18 Design of a 5-bit squarer.

Chapter 24 contains a comprehensive discussion of table-lookup methods for performing, or facilitating, arithmetic computations.

Finally, exponentiation can be performed by a sequence of squaring or square-multiply steps. For example, based on the identity

$$x^{13} = x((x(x^2))^2)^2$$

we can compute x^{13} by squaring x, multiplying the result by x, squaring twice, and finally multiplying the result by x. We discuss exponentiation for both real and integer operands in greater detail in Section 23.3.

12.6 COMBINED MULTIPLY-ADD UNITS

In certain computations, such as vector inner-product, convolution, or fast Fourier transform, multiplications are commonly followed by additions. In such cases, implementing a multiply-add unit in hardware to compute $p = ax + y$ might be cost-effective. Since the preceding computations are commonplace in signal processing applications, most modern digital signal processors (DSPs) have built-in hardware capability for multiply-add, or multiply-accumulate, operations. An example of this capability in DSP chips is presented in the last chapter (see Section 28.4).

We have already discussed additive multiply modules (Section 12.2) that add one or two numbers to the product of their multiplicative inputs. Similarly, at several points in this and the preceding three chapters we have hinted at a means of incorporating an additive input into the multiplication process (e.g., by initializing the cumulative partial product to a nonzero value or by entering a nonzero value to the top row of an array multiplier). In all cases, however, the additive inputs are comparable in width to the multiplicative inputs.

The type of multiply-add operation of interest to us here involves an additive input that is significantly wider than the multiplicative inputs (perhaps even wider than their product). For example, we might have 24-bit multiplicative inputs, yielding a 48-bit product, that is then added to a 64-bit running sum. The wider running sum may be required to avoid overflow in the intermediate computation steps or to provide greater precision to counter the accumulation of errors.

Figure 12.19 depicts several methods for incorporating a wide additive input into the multiplication process. First, we might use a CSA tree to find the product of the multiplicative inputs in carry-save form and then add the result to the additive input using a CSA followed by a fast adder (Fig. 12.19a). To avoid a carry-propagate addition in every step, the running sum may itself be kept in carry-save form, leading to the requirement for two CSA levels (Fig. 12.19b). The resulting hardware implementation for this latter scheme is quite similar to the partial-tree multiplier of Fig. 11.9.

Alternatively, the two-step process of computing the product in carry-save form and adding it to the running sum can be replaced by a merged multiply-add operation that directly operates on the dots from the additive input(s) and the partial products dot matrix (Figs. 12.19c and 12.19d). We will revisit this notion of merged arithmetic in Section 23.6.

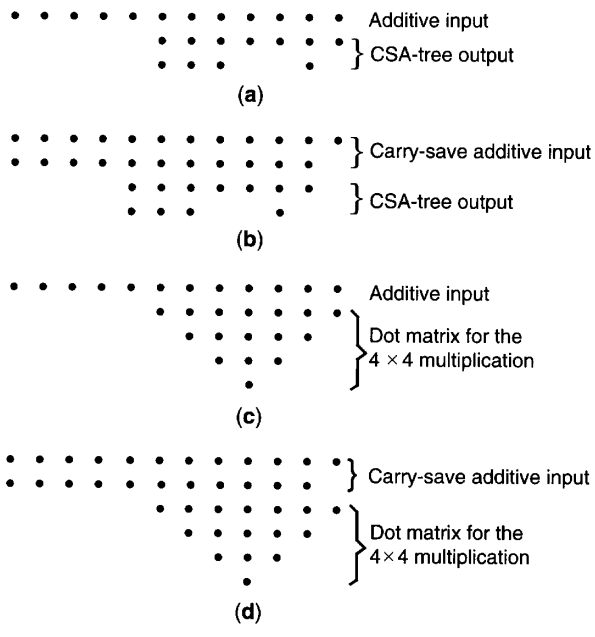


Fig. 12.19 Dot notation representations of various methods for performing a multiply-add operation in hardware.

PROBLEMS

12.1 Multipliers built of smaller modules

a. Draw a schematic diagram of the 16×4 multiplier for unsigned numbers using only 4×4 multipliers and 4-bit adders.

b. Using dot notation, show an implementation for summing the four partial products of part a using only 4-bit CSA modules and 4-bit carry-propagate adders.

c. Repeat part a with the 16-bit number in 2's-complement format.

d. Repeat part b for the multiplier of part c.

12.2 Multipliers built of smaller modules Consider Fig. 12.2 depicting the construction of $gb \times gb$ multipliers from $b \times b$ units.

a. Express the height of the partial products matrix of Fig. 12.2 as a function of g.

b. Generalize the result of part a to $gb \times hb$ multiplier built of $b \times b$ modules.

c. Repeat part a for the case of $b \times c$ multipliers being used to synthesize a $gb \times gc$ multiplier.

d. Generalize the result of part c to $gb \times hc$ multiplier synthesized from $b \times c$ units.

12.3 Multipliers built of AMMs Compare the 8×8 multiplier designs in Figs. 12.5 and 12.6 with respect to speed, assuming the following implementations for the 2×4 AMM of Fig. 12.4.

a. Four-bit CSA followed by 4-bit ripple-carry adder.

b. Four-bit CSA followed by 4-bit carry-lookahead adder.

12.4 Multipliers built of AMMs

a. Design a 2×2 AMM, with two 2-bit additive inputs, using only four single-bit full adders and four AND gates.

b. Show how to connect four AMMs of part a to form a 4×4 unsigned multiplier.

c. Estimate the delay of the 4×4 multiplier of part b, in units of FA delay, by drawing and justifying the critical path on the circuit diagram.

d. Can one use the multiplier of part b as a 4×4 AMM? How or why not?

12.5 Building larger AMMs

a. We have an unlimited supply of 2×4 AMMs of the type depicted in Fig. 12.4. Using a minimal number of these AMMs, and no other component, synthesize a 4×4 AMM (with two 4-bit additive inputs).

b. Repeat part a for a 2×8 AMM (additive inputs are 2 and 8 bits wide).

c. Repeat part a for a 6×6 AMM (additive inputs are both 6 bits wide).

d. Repeat part a for a 4×8 AMM (additive inputs are 4 and 8 bits wide).

e. Build the 4×8 AMM of part d using two of the 4×4 AMMs designed in part a.

f. Compare the designs of parts d and e with respect to speed and cost.

12.6 Multipliers built of AMMs

a. Design a 16×8 multiplier using 4×2 AMMs arranged in a 4×4 array.

b. Repeat part a, this time arranging the modules in an 8×2 array.

c. Compare the designs of parts a and b with respect to speed.

d. Convert the designs of parts a and b into 16×8 AMMs.

12.7 AMMs for 2's-complement multiplication

a. Design a 2×4 AMM, similar to that in Fig. 12.4, but with the following changes. The x input is internally recoded using the digit set $[-2, 2]$, so a third x bit, x_{-1}, is needed as context and a fifth a input, a_{-1}, in case of left shifting. The 2-bit additive input is replaced by a 1-bit input c_i and a 1-bit output c_{i+4} that completes the 5-bit sum of the two 4-bit values. A 6-bit result is needed at the most significant end, so the AMM should also produce the two most significant bits of the result, to be used in lieu of c_{i+4} when needed.

b. Build a 4×4 2's-complement multiplier using the AMMs of part a.

c. Repeat part b for an 8×8 2's-complement multiplier.

12.8 Systolic multipliers

a. Present an argument for the correctness of the systolic multiplier in Fig. 12.10.

b. Trace the steps of the unsigned binary multiplication $(1101)_{\text{two}} \times (0101)_{\text{two}}$ to verify your conclusion in part a.

c. Propose a cell design such that the multiplicand is stored internally and can be modified when needed (this is useful when the multiplicand is a coefficient that seldom changes). There are two operation modes. In "load" mode, the serial input

pin is used to shift the multiplicand into internal latches (LSB-first). In "multiply" mode, the multiplier is supplied as input and the product emerges as output.

12.9 Systolic multipliers A fully bit-serial $k \times k$ systolic multiplier can be designed on the basis of a linear array of $2k$ cells, numbered 0 through $2k - 1$ from right to left, which at the end will hold the $2k$-bit product. The multiplier x is input from the left on even-numbered clock ticks, with x_i arriving at time $2i$. The multiplicand a is input from the right, MSB first, on odd numbered clock ticks, with a_j input at time $2k - 2j + 1$.

a. Show that x_i and a_j meet at cell h if and only if $i + j = h$.

b. Use the result of part a to derive a suitable cell design and intercell connections.

12.10 Modular multipliers Discuss the design of modulo-$(2^b + 1)$ multipliers using a suitable $(b + 1)$-bit encoding of the inputs and intermediate results. *Hint:* Consider using one bit to represent 0 and reducing all nonzero values by 1 for representation with the remaining b bits.

12.11 Modular multipliers

a. Present a complete design for the modulo-13 multiplier discussed at the end of Section 12.4.

b. Compare the design of part a to a standard 4×4 multiplier with respect to speed and cost.

c. Design a 5×5 modulo-29 multiplier. *Hint:* Work with partial results in [0, 31] rather than [0, 28]. When a partial result exceeds 31, subtract 29 from it by discarding the carry-out (worth 32 units) and adding 3. Thus, a wraparound connection similar to that in Fig. 12.14 must be established from the carry-out to the two least significant positions. The final sum in [0, 31] may need adjustment.

12.12 Modular squarers

a. Simplify the reduced partial products matrix of Fig. 12.18 to the extent possible if the square of the 5-bit number x is to be obtained modulo 31.

b. Repeat part a for modulo-29 squaring of a 5-bit number.

c. Discuss how modular multiplication ax mod m can be performed based on modular squaring tables that hold z^2 mod m.

12.13 Design of squarers

a. Show that a 4-bit unsigned squarer can be designed using only two-input AND gates, one full-adder, and a 5-bit binary adder.

b. Using the identity $x_1x_0 + x_1 = 2x_1x_0 + x_1\bar{x}_0$, as discussed near the end of Section 12.5, reduce the complexity of the 4-bit squarer of part a to a 4-bit adder plus a few logic gates.

c. Design a circuit to compute the square of a 4-bit 2's-complement input integer. *Hint:* Use the identity $-x_jx_i = -2x_j + x_j\bar{x}_i + x_j$ and note that the final product is representable in only 7 bits.

12.14 Bit-serial squarers Present a simplified version of the bit-serial multiplier design in Fig. 12.11 for squaring a number x [Ienn94]. *Hint:* The two terms $a_i x^{(i-1)}$ and $x_i a^{(i-1)}$

are the same. So a single value needs to be added to the accumulated result. Because of this, the accumulated result can be kept in carry-save form, rather than as three numbers, allowing the use of a (3; 2)-counter.

12.15 Bit-serial inner-product computation Consider replacing the (5; 3)-counter in Fig. 12.11 by a (7; 3)-counter and using the two extra inputs to accommodate serial inputs b and y, so that the value of $ax + by$ is computed bit-serially [Hayn96].

a. How should the part of the circuit producing s_{out} be modified?

b. Show that the resulting cells can in fact be used to compute $ax + by + z$.

12.16 Multiplication of complex numbers The quater-imaginary number system of Example 1.7 in Section 1.4 can be easily generalized to radix $j\sqrt{r}$ and digit set $[0, r - 1]$. Show that any complex number is representable in such a number system and discuss whether this representation leads to faster multiplication for complex numbers.

12.17 Multipliers with narrower products Our discussions in Chapters 9–12 were based on the assumption that in multiplying two k-bit operands, the full $2k$-bit product must be produced.

a. Present a thorough discussion of how the various multiplier designs are affected if the k-bit product of two k-bit integers, plus an overflow indication, are sufficient.

b. Repeat part a, assuming that the input operands are k-bit fractions yielding a k-bit product by truncating all bits of p beyond p_{-k}.

12.18 Fractional precision multiplication

a. Consider a 6×6 multiplier that uses a Wallace tree to reduce the six partial products to two numbers and then adds them in a fast adder to obtain the product. Suggest modifications in the design such that under the control of a "fractional precision" signal, the multiplier acts as two independent 3×3 multipliers operating on the low and high halves of the 6-bit inputs.

b. Repeat part a, this time assuming that the 6×6 multiplier is built of 3×3 AMMs.

c. Compare the incremental cost of adding the fractional precision arithmetic capability to the multipliers of parts a and b and discuss.

d. Many modern microprocessors have a capability for fractional precision arithmetic that allows them to handle multimedia data efficiently. How would you go about designing a 32×32 multiplier so that it can also view its 32-bit inputs as two pairs of 16-bit values or four pairs of 8-bit values?

REFERENCES

[Alia91] Alia, G., and E. Martinelli, "A VLSI Modulo m Multiplier," *IEEE Trans. Computers*, Vol. 40, No. 7, pp. 873–878, 1991.

[Chen79] Chen, I.-N., and R. Willowner, "An O(n) Parallel Multiplier with Bit-Sequential Input and Output," *IEEE Trans. Computers*, Vol. 28, No. 10, pp. 721–727, 1979.

[Ghes71] Ghest, C., "Multiplying Made Easy for Digital Assemblies," *Electronics*, Vol. 44, pp. 56–61, November 22, 1971.

[Hayn96] Haynal, S., and B. Parhami, "Arithmetic Structures for Inner-Product and Other Computations Based on a Latency-Free Bit-Serial Multiplier Design," *Proc. 30th Asilomar Conf. Signals, Systems, and Computers*, November 1996.

[Hwan79] Hwang, K., *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, 1979.

[Ienn94] Ienne, P., and M. A. Viredaz, "Bit-Serial Multipliers and Squarers," *IEEE Trans. Computers*, Vol. 43, No. 12, pp. 1445–1450, 1994.

[Kung82] Kung, H. T., "Why Systolic Architectures?" *Computer*, Vol. 15, No. 1, pp. 37–46, 1982.

[Parh93] Parhami, B., and H.-F. Lai, "Alternate Memory Compression Schemes for Modular Multiplication," *IEEE Trans. Signal Processing*, Vol. 41, No. 3, pp. 1378–1385, 1993.

[Pies94] Piestrak, S. J., "Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders," *IEEE Trans. Computers*, Vol. 43, No. 1, pp. 68–77, 1994.

PART IV DIVISION

Division is the most complex of the four basic arithmetic operations and the hardest one to speed up. Thus, dividers are more expensive and/or slower than multipliers. Fortunately, division operations are also less common than multiplications. Two classes of dividers are discussed here. In digit-recurrence schemes, the quotient is generated one digit at a time, beginning at the most significant end. Binary versions of digit-recurrence division can be implemented through shifting and addition, in much the same way as shift/add multiplication schemes. Determining the digits of the quotient from the most significant end allows us to "converge" to a k-digit quotient in k cycles. Speeding up of division via reducing the number of shift/add cycles leads to high-radix dividers. Array dividers as well as convergence methods that require far fewer than k iterations, with each iteration being more complex, are also discussed. This part is composed of the following four chapters:

Chapter 13 BASIC DIVISION SCHEMES

Like sequential multiplication of k-bit operands, yielding a $2k$-bit product, the division of a $2k$-bit dividend by a k-bit divisor can be realized in k cycles of shifting and adding (actually subtracting), with hardware, firmware, or software control of the loop. In this chapter, we review such economical, but slow, bit-at-a-time designs and set the stage for speedup methods and variations to be presented in Chapters 14–16. We also consider the special case of division by a constant. Chapter topics include:

13.1 SHIFT/SUBTRACT DIVISION ALGORITHMS

The following notation is used in our discussion of division algorithms:

z	Dividend	$z_{2k-1}z_{2k-2}\cdots z_1z_0$
d	Divisor	$d_{k-1}d_{k-2}\cdots d_1d_0$
q	Quotient	$q_{k-1}q_{k-2}\cdots q_1q_0$
s	Remainder $[z-(d\times q)]$	$s_{k-1}s_{k-2}\cdots s_1s_0$

The expression $z-(d\times q)$ for the remainder s is derived from the basic division equation $z=(d\times q)+s$. This equation, along with the condition $s<d$, completely defines unsigned integer division.

Figure 13.1 shows a $2k$-bit by k-bit unsigned integer division in dot notation. The dividend z and divisor d are shown near the top. Each of the following four rows of dots corresponds to the product of the divisor d and one bit of the quotient q, with each dot representing the product

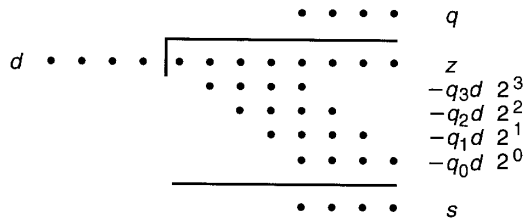


Fig. 13.1 Division of an 8-bit number by a 4-bit number in dot notation.

(logical AND) of two bits. Since q_{k-j} is in {0, 1}, each term $q_{k-j}d$ is either 0 or d. Thus, the problem of binary division reduces to subtracting a set of numbers, each being 0 or a shifted version of the divisor d, from the dividend z.

Figure 13.1 also applies to nonbinary division, except that with $r > 2$, both the selection of the next quotient digit q_{k-j} and the computation of the terms $q_{k-j}d$ become more difficult and the resulting products are one digit wider than d. The rest of the process, however, remains substantially the same.

Just as sequential multiplication was done by repeated additions, sequential division is performed by repeated subtractions. The partial remainder is initialized to $s^{(0)} = z$. In step j, the next quotient digit q_{k-j} is selected. Then, the product $q_{k-j}d$ (which is either 0 or d) is shifted and the result subtracted from the partial remainder. So, compared to multiplication, division has the added complication of requiring quotient digit selection or estimation.

Another aspect of division that is different from multiplication is that whereas the product of two k-bit numbers is always representable in $2k$ bits, the quotient of a $2k$-bit number divided by a k-bit number may have a width of more than k bits. Thus, an overflow check is needed before division algorithm is applied. Since, for unsigned division, we have $q < 2^k$ and $s < d$, to avoid overflow, we must have:

$$z < (2^k - 1)d + d = 2^k d$$

Hence, the high-order k bits of z must be strictly less than d. Note that this overflow check also detects the divide-by-0 condition.

Fractional division can be reformulated as integer division, and vice versa. In an integer division characterized by $z = (d \times q) + s$, we multiply both sides by 2^{-2k}:

$$2^{-2k}z = \left[(2^{-k}d) \times (2^{-k}q)\right] + 2^{-2k}s$$

Now, letting the $2k$-bit and k-bit inputs be fractions, we see that their fractional values are related by:

$$z_{\text{frac}} = (d_{\text{frac}} \times q_{\text{frac}}) + 2^{-k}s_{\text{frac}}$$

Therefore, we can divide fractions just as we divide integers, except that the final remainder must be shifted to the right by k bits. In effect, this means that k zeros are to be inserted after the radix point to make the k-bit (fractional) remainder into a $2k$-bit fractional number with k leading 0s. This makes sense because when we divide z_{frac} by a number d_{frac} that is less than 1, the remainder should be less than *ulp* in the quotient (otherwise, the quotient could be increased without the remainder going negative). The condition for no overflow in this case is $z_{\text{frac}} < d_{\text{frac}}$, which is checked in exactly the same way as for integer division.

Sequential or bit-at-a-time division can be performed by keeping a partial remainder, initialized to $s^{(0)} = z$, and successively subtracting from it the properly shifted terms $q_{k-j}d$ (Fig. 13.1). Since each successive number to be subtracted from the partial remainder is shifted by one bit with respect to the preceding one, a simpler approach is to shift the partial remainder by one bit, to align its bits with those of the next term to be subtracted. This leads to the well-known sequential division algorithm with left shifts:

$$s^{(j)} = 2s^{(j-1)} - q_{k-j}(2^k d) \quad \text{with} \quad s^{(0)} = z \quad \text{and} \quad s^{(k)} = 2^k s$$

|shift left|

|—— subtract ——|

The factor 2^k by which d is premultiplied ensures proper alignment of the values. After k iterations, the preceding recurrence leads to:

$$s^{(k)} = 2^k s^{(0)} - q(2^k d) = 2^k[z - (q \times d)] = 2^k s$$

The fractional version of the division recurrence is:

$$s_{\text{frac}}^{(j)} = 2s_{\text{frac}}^{(j-1)} - q_{-j}d_{\text{frac}} \quad \text{with} \quad s_{\text{frac}}^{(0)} = z_{\text{frac}} \quad \text{and} \quad s_{\text{frac}}^{(k)} = 2^k s_{\text{frac}}$$

Note that unlike multiplication, where the partial products can be produced and processed from top to bottom or bottom to top, in the case of division, the terms to be subtracted from the initial partial remainder must be produced from top to bottom. The reason is that the quotient bits become known sequentially, beginning with the most significant one, whereas in multiplication all the multiplier bits are known at the outset. This is why we do not have a division algorithm with right shifts (corresponding to multiplication with left shifts).

The division of $z = (117)_{\text{ten}} = (0111\ 0101)_{\text{two}}$ by $d = (10)_{\text{ten}} = (1010)_{\text{two}}$ to obtain the quotient $q = (11)_{\text{ten}} = (1011)_{\text{two}}$ and the remainder $s = (7)_{\text{ten}} = (0111)_{\text{two}}$ is depicted on the left-hand side of Fig. 13.2. Figure 13.2 (right) shows the fractional version of the same division, with the operands $z = (117/256)_{\text{ten}} = (.0111\ 0101)_{\text{two}}$, $d = (10/16)_{\text{ten}} = (.1010)_{\text{two}}$ and the results $q = (11/16)_{\text{ten}} = (.1011)_{\text{two}}$, $s = (7/256)_{\text{ten}} = (.0000\ 0111)_{\text{two}}$.

In practice, the required subtraction is performed by adding the 2's complement of $2^k d$ or d to the partial remainder (more on this later). Note that there are but two choices for the value of the next quotient digit q_{k-j} or q_{-j} in radix 2, with the value 1 selected whenever the shifted partial remainder $2s^{(j-1)}$ is greater than $2^k d$ or d. Sections 13.3 and 14.4 contain more detailed discussions on quotient digit selection.

13.2 PROGRAMMED DIVISION

On a processor that does not have a divide instruction, one can use shift and add instructions to perform integer division. Since one quotient digit is produced after each left shift of the partial remainder, we need only two k-bit registers to store the partial remainder and the quotient: Rz for the most significant k bits of the partial remainder, and Rq for the rest of the partial remainder plus the partial quotient produced thus far (Fig. 13.3). In each cycle, the double-width register Rz|Rq is shifted left and the new quotient digit is inserted in the just-vacated LSB of Rq. This insertion is accomplished by incrementing Rq by 1 if the next quotient digit is 1.

Integer division

z		0	1	1	1	0	1	0 1
2^4d		1	0	1	0			
$s^{(0)}$		0	1	1	1	0	1	0 1
$2s^{(0)}$	0	1	1	1	0	1	0	1
$-q_3 2^4 d$		1	0	1	0			$\{q_3 = 1\}$
$s^{(1)}$		0	1	0	0	1	0	1
$2s^{(1)}$	0	1	0	0	1	0	1	
$-q_2 2^4 d$		0	0	0	0			$\{q_2 = 0\}$
$s^{(2)}$		1	0	0	1	0	1	
$2s^{(2)}$	1	0	0	1	0	1		
$-q_1 2^4 d$		1	0	1	0			$\{q_1 = 1\}$
$s^{(3)}$		1	0	0	0	1		
$2s^{(3)}$	1	0	0	0	1			
$-q_0 2^4 d$		1	0	1	0			$\{q_0 = 1\}$
$s^{(4)}$		0	1	1	1			
s						0	1	1 1
q						1	0	1 1

Fractional division

z_{frac}	.	0	1	1	1	0	1	0 1
d_{frac}	.	1	0	1	0			
$s^{(0)}$	.	0	1	1	1	0	1	0 1
$2s^{(0)}$	0	.	1	1	1	0	1	0 1
$-q_{-1}d$		.	1	0	1	0		$\{q_{-1} = 1\}$
$s^{(1)}$		.	0	1	0	0	1	0 1
$2s^{(1)}$	0	.	1	0	0	1	0	1
$-q_{-2}d$		.	0	0	0	0		$\{q_{-2} = 0\}$
$s^{(2)}$		.	1	0	0	1	0	1
$2s^{(2)}$	1	.	0	0	1	0	1	
$-q_{-3}d$		.	1	0	1	0		$\{q_{-3} = 1\}$
$s^{(3)}$		.	1	0	0	0	1	
$2s^{(3)}$	1	.	0	0	0	1		
$-q_{-4}d$		.	1	0	1	0		$\{q_{-4} = 1\}$
$s^{(4)}$		.	0	1	1	1		
s_{frac}		.	0	0	0	0	0	1 1 1
q_{frac}		.	1	0	1	1		

Fig. 13.2 Examples of sequential division with integer and fractional operands.

Figure 13.4 shows the structure of the needed program for sequential division. The instructions used in this program fragment are typical of instructions available on many processors.

The subtract instruction in the program fragment of Fig. 13.4 needs some elaboration. If we reach the subtract instruction by falling through its preceding branch instruction, then Rs $\geq$ Rd, and the desired effect of leaving Rs $-$ Rd in Rs is achieved through subtraction. However, if we reach the subtract instruction from the skip instruction, then the carry flag is 1 and Rs $<$ Rd. In this case, the proper result is to leave $(2^k + \text{Rs}) - \text{Rd}$ in Rs, where 2^k represents the MSB of the shifted partial remainder held in the carry flag. But we have:

$$\begin{aligned}(2^k + \text{Rs}) - \text{Rd} &= \text{Rs} + (2^k - \text{Rd}) \\ &= \text{Rs} + \text{2's-complement of Rd}\end{aligned}$$

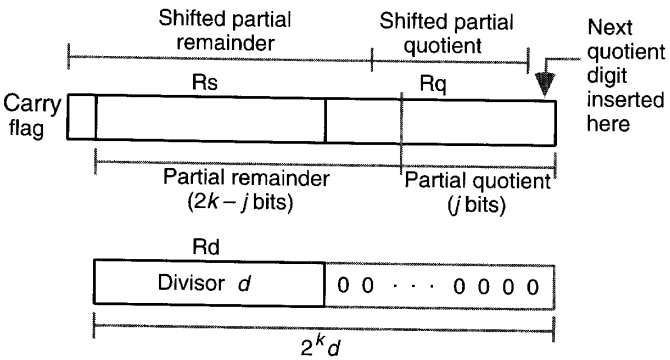


Fig. 13.3 Register usage for programmed division.

```
{Using left shifts, divide unsigned 2k-bit dividend,
z_highlz_low, storing the k-bit quotient and remainder.
Registers:    R0 holds 0              Rc for counter
              Rd for divisor          Rs for z_high & remainder
              Rq for z_low & quotient}

{Load operands into registers Rd, Rs, and Rq}

      div:    load      Rd with divisor
              load      Rs with z_high
              load      Rq with z_low

{Check for exceptions}

              branch    d_by_0 if Rd = R0
              branch    d_ovfl if Rs > Rd

{Initialize counter}

              load      k into Rc

{Begin division loop}

  d_loop:     shift     Rq left 1       {zero to LSB, MSB to carry}
              rotate    Rs left 1       {carry to LSB, MSB to carry}
              skip      if carry = 1
              branch    no_sub if Rs < Rd
              sub       Rd from Rs
              incr      Rq              {set quotient digit to 1}
  no_sub:     decr      Rc              {decrement counter by 1}
              branch    d_loop if Rc ≠ 0

{Store the quotient and remainder}

              store     Rq into quotient
              store     Rs into remainder
  d_by_0:     . . .
  d_ovfl:     . . .
  d_done:     . . .
```

Fig. 13.4 Programmed division using left shifts.

Thus, even though we are performing unsigned division, a 2's-complement subtract instruction produces the proper result in either case.

Ignoring operand load and result store instructions (which would be needed in any implementation), the function of a divide instruction is accomplished by executing between $6k+3$ and $8k+3$ machine instructions, depending on the operands. For 32-bit operands, this means well over 200 instructions on the average. The situation improves somewhat if a special instruction that does some or all of the required functions within the division loop is available. However, even then, no fewer than 32 instructions would be executed in the division loop. We thus see the importance of hardware dividers for applications that involve a great deal of numerical computations.

Microprogrammed processors with no hardware divider use a microroutine very similar to the program in Fig. 13.4 to perform division. For the same reasons given near the end of Section 9.2 in connection with programmed multiplication, division microroutines are significantly faster than their machine-language counterparts, though still slower than the hardwired implementations we examine next.

13.3 RESTORING HARDWARE DIVIDERS

Figure 13.5 shows a hardware realization of the sequential division algorithm for unsigned integers. At the start of each cycle j, the partial remainder $s^{(j-1)}$ is shifted to the left, with its MSB moving into a special flip-flop. Then the trial difference $2s^{(j-1)} - q_{k-j}(2^k d)$ is computed. Because of the 2^k factor in the preceding expression, the divisor is aligned with the upper k bits of the partial remainder for the trial subtraction and the lower part of the partial remainder is not affected.

As stated in connection with programmed division in Section 13.2, the next quotient digit should be 1 if the MSB of $2s^{(j-1)}$, held in the special flip-flop, is 1 or if the trial difference is positive ($c_{out} = 1$). In either case, $q_{k-j} = 1$ becomes the shift input for the quotient register and also causes the trial difference to be loaded into the upper half of the partial remainder register to form the new partial remainder for the next cycle. Otherwise, $q_{k-j} = 0$, and the partial remainder does not change.

We refer to the division scheme of Fig. 13.5 as restoring division. The quotient digit in radix 2 is in {0, 1}. The trial subtraction corresponds to assuming $q_{k-j} = 1$. If the trial difference is positive, then the next quotient digit is indeed 1. Otherwise, $q_{k-j} = 1$ is too large and the quotient digit must be 0. The term "restoring division" means that the remainder is restored to its correct value if the trial subtraction indicates that 1 was not the right choice for q_{k-j}. Note that we could have chosen to load the trial difference in the partial remainder register in all cases, restoring the remainder to its correct value by a compensating addition step when needed. However, this would have led to slower hardware.

Just as the multiplier could be stored in the lower half of the partial product register (Fig. 9.4), the quotient and the lower part of the partial remainder can share the same space, since quotient bits are derived as bits of the partial remainder move left, freeing the required space for them. Excluding the control logic, the hardware requirements of multiplication and division are quite similar, so the two algorithms can share much hardware components (compare Figs. 9.4 and 13.5).

As a numerical example, we use the restoring algorithm to redo the integer division given in Fig. 13.2. The result is shown in Fig. 13.6; note the restoration step corresponding

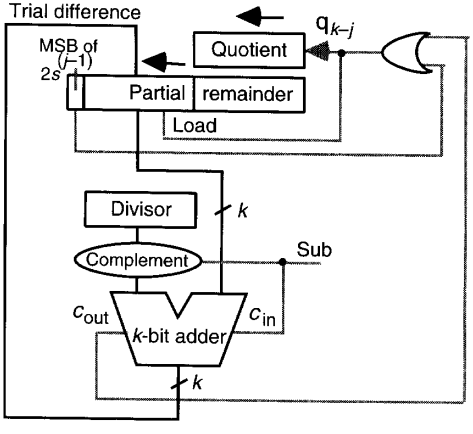


Fig. 13.5 Shift/subtract sequential restoring divider.

to $q_2 = 0$ and the extra bit devoted to sign in intermediate operands. A shifted partial remainder does not need an extra sign bit, since its magnitude is immediately reduced by a trial subtraction.

Thus far, we have assumed unsigned operands and results. For signed operands, the basic division equation $(d \times q) + s$, along with

$$\text{sign}(s) = \text{sign}(z) \quad \text{and} \quad |s| < |d|$$

uniquely define the quotient q and remainder s.

Consider the following examples of integer division with all possible combinations of signs for z and d:

$$\begin{array}{llllll} z = 5 & d = 3 & \Rightarrow & q = 1 & s = 2 \\ z = 5 & d = -3 & \Rightarrow & q = -1 & s = 2 \\ z = -5 & d = 3 & \Rightarrow & q = -1 & s = -2 \\ z = -5 & d = -3 & \Rightarrow & q = 1 & s = -2 \end{array}$$

We see from the preceding examples that the magnitudes of q and s are unaffected by the input signs and that the signs of q and s are easily derivable from the signs of z and d. Hence, one way to do signed division is through an indirect algorithm that converts the operands into unsigned values and, at the end, accounts for the signs by adjusting the sign bits or via complementation. This is the method of choice with the restoring division algorithm.

z		0	1	1	1	0	1	0	1
$24d$	0	1	0	1	0				
$-24d$	1	0	1	1	0				
$s(0)$	0	0	1	1	1	0	1	0	1
$2s(0)$	0	1	1	1	0	1	0	1	
$+(-24d)$	1	0	1	1	0				
$s(1)$	0	0	1	0	0	1	0	1	
$2s(1)$	0	1	0	0	1	0	1		
$+(-24d)$	1	0	1	1	0				
$s(2)$	1	1	1	1	1	0	1		
$s(2) = 2s(1)$	0	1	0	0	1	0	1		
$2s(2)$	1	0	0	1	0	1			
$+(-24d)$	1	0	1	1	0				
$s(3)$	0	1	0	0	0	1			
$2s(3)$	1	0	0	0	1				
$+(-24d)$	1	0	1	1	0				
$s(4)$	0	0	1	1	1				
s						0	1	1	1
q						1	0	1	1

No overflow, since: $(0111)_{\text{two}} < (1010)_{\text{two}}$

Positive, so set $q_3 = 1$

Negative, so set $q_2 = 0$ and restore

Positive, so set $q_1 = 1$

Positive, so set $q_0 = 1$

Fig. 13.6 Example of restoring unsigned division.

13.4 NONRESTORING AND SIGNED DIVISION

Implementation of restoring division requires paying attention to the timing of various events. Each of the k cycles must be long enough to allow the following events in sequence:

Shifting of the registers.
Propagation of signals through the adder.
Storing of the quotient digit.

Thus, the sign of the trial difference must be sampled near the end of the cycle (say at the negative edge of the clock). To avoid such timing issues, which tend to lengthen the clock cycle, one can use the nonrestoring division algorithm. As before, we assume $q_{k-j} = 1$ and perform a subtraction. However, we always store the difference in the partial remainder register. This leads to the partial remainder being temporarily incorrect (hence the name "nonrestoring").

Let us see why it is acceptable to store an incorrect value in the partial remainder register. Suppose that the shifted partial remainder at the start of the cycle was u. If we had restored the partial remainder $u - 2^k d$ to its correct value u, we would proceed with the next shift and trial subtraction, getting the result $2u - 2^k d$. Instead, because we used the incorrect partial remainder, a shift and trial subtraction would yield $2(u - 2^k d) - 2^k d = 2u - (3 \times 2^k d)$, which is not the intended result. However, an addition would do the trick, resulting in $2(u - 2^k d) + 2^k d = 2u - 2^k d$, which is the same value obtained after restoration and trial subtraction. Thus, in nonrestoring division, when the partial remainder becomes negative, we keep the incorrect partial remainder, but note the correct quotient digit and also remember to add, rather than subtract, in the next cycle.

Before discussing the adaptation of nonrestoring algorithm for use with signed operands, let us use the nonrestoring algorithm to redo the example division of Fig. 13.6. The result is shown in Fig. 13.7. We still need just one extra bit for the sign of $s^{(j)}$, which doubles as a magnitude bit for $2s^{(j)}$.

Figure 13.8 depicts the relationship between restoring division and nonrestoring division for the preceding example division, namely, $(117)_{\text{ten}}/(10)_{\text{ten}}$. In each cycle, the value $2^k d = (160)_{\text{ten}}$ is added to or subtracted from the shifted partial remainder.

Recall that in restoring division, the quotient digit values of 0 and 1 corresponded to "no subtraction" (or subtraction of 0) and "subtraction of d," respectively. In nonrestoring division, we always subtract or add. Thus, it is as if the quotient digits are selected from the set $\{1, \bar{1}\}$, with 1 corresponding to subtraction and $\bar{1}$ to addition. Our goal is to end up with a remainder that matches the sign of the dividend (positive in unsigned division). Well, this viewpoint (of trying to match the sign of the partial remainder s with the sign of the dividend z) leads to the idea of dividing signed numbers directly. The rule for quotient digit selection becomes:

$$\text{If sign}(s) = \text{sign}(d) \text{ then } q_{k-j} = 1 \text{ else } q_{k-j} = \bar{1}$$

Two problems must be dealt with at the end:

1. The quotient with digits 1 and $\bar{1}$ must be converted to standard binary.
2. If the final remainder s has a sign opposite that of z, a correction step, involving the addition of $\pm d$ to the remainder and subtraction of ± 1 from the quotient, is needed (since there is no next step to compensate for the nonrestoration of the correct remainder).

Note that the correction step might be required even in unsigned division (when the final remainder is negative). We deal with the preceding two problems in turn.

z		0	1	1	1	0	1	0	1
2^4d	0	1	0	1	0				
-2^4d	1	0	1	1	0				
$s^{(0)}$	0	0	1	1	1	0	1	0	1
$2s^{(0)}$	0	1	1	1	0	1	0	1	
$+(-2^4d)$	1	0	1	1	0				
$s^{(1)}$	0	0	1	0	0	1	0	1	
$2s^{(1)}$	0	1	0	0	1	0	1		
$+(-2^4d)$	1	0	1	1	0				
$s^{(2)}$	1	1	1	1	1	0	1		
$2s^{(2)}$	1	1	1	1	0	1			
$+2^4d$	0	1	0	1	0				
$s^{(3)}$	0	1	0	0	0	1			
$2s^{(3)}$	1	0	0	0	1				
$+(-2^4d)$	1	0	1	1	0				
$s^{(4)}$	0	0	1	1	1				
s						0	1	1	1
q						1	0	1	1

No overflow, since: $(0111)_{\text{two}} < (1010)_{\text{two}}$

Positive, so subtract

Positive, so set $q_3 = 1$ and subtract

Negative, so set $q_2 = 0$ and add

Positive, so set $q_1 = 1$ and subtract

Positive, so set $q_0 = 1$

Fig. 13.7 Example of nonrestoring unsigned division.

To convert a k-digit BSD quotient $q = (q_{k-1}q_{k-2}\cdots q_0)_{\text{BSD}}$ with $q_i \in \{\bar{1}, 1\}$ to a k-bit, 2's-complement number, do as follows:

a. Replace all $\bar{1}$ digits with 0s to get the k-bit number $p = p_{k-1}p_{k-2}\cdots p_0$, with $p_i \in \{0, 1\}$. Note that the p_is and q_is are related by $q_i = 2p_i - 1$.

b. Complement p_{k-1} and then shift p left by 1 bit, inserting 1 into the LSB, to get the 2's-complement quotient $q = (\overline{p}_{k-1}p_{k-2}\cdots p_0 1)_{2\text{'s-compl}}$.

The proof of correctness for the preceding conversion process is straightforward (note that we have made use of the identity $\sum_{i=0}^{k-1} 2^i = 2^k - 1$):

$$
\begin{aligned}
(\overline{p}_{k-1}p_{k-2}\cdots p_0 1)_{2\text{'s-compl}} &= -(1 - p_{k-1})2^k + 1 + \sum_{i=0}^{k-2} p_i 2^{i+1} \\
&= -(2^k - 1) + 2\sum_{i=0}^{k-1} p_i 2^i \\
&= \sum_{i=0}^{k-1} (2p_i - 1)2^i \\
&= \sum_{i=0}^{k-1} q_i 2^i = q
\end{aligned}
$$

From the preceding algorithm, we see that the conversion is quite simple and can be done on the fly as the digits of the quotient are obtained. If the quotient is to be representable as a k-bit,

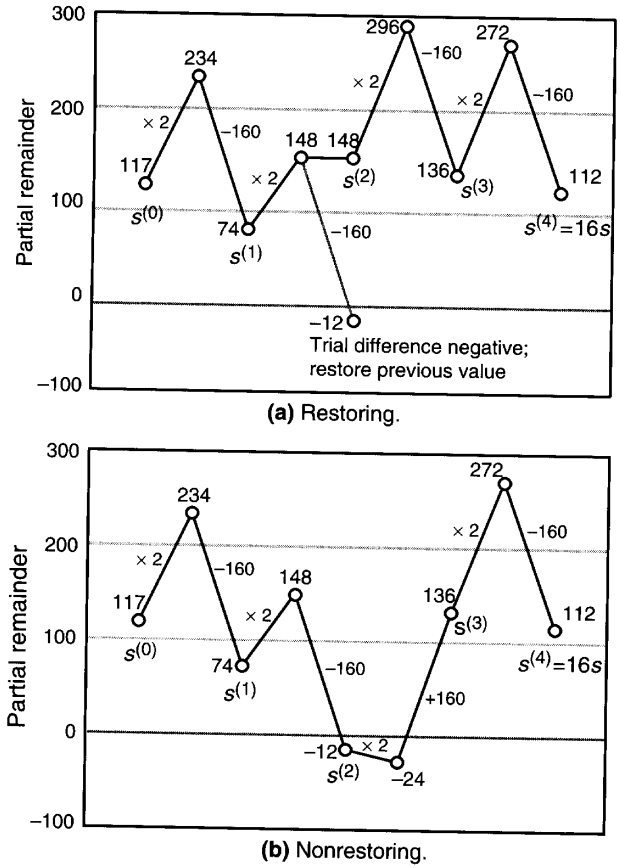


Fig. 13.8 Partial remainder variations for restoring and nonrestoring division.

2's-complement number, then we must have $\overline{p}_{k-1} = p_{k-2}$, leading to the requirement that the BSD digits q_{k-1} and q_{k-2} be different. Thus, overflow is avoided if and only if:

$$\text{sign}(z) \neq \text{sign}(s^{(1)})$$

Hence, on-the-fly conversion consists of setting the quotient sign bit in the initial cycle, producing a 1 (0) for each subtract (add) thereafter, and producing a 1 for the last digit before proceeding to the correction step.

The final correction, if needed, is also quite simple. It involves adding/subtracting 1 to/from q and subtracting/adding $2^k d$ from/to the remainder. Note that the aim of the correction step is to change the sign of the remainder. Thus if $\text{sign}(s^{(k)}) = \text{sign}(d)$, we subtract from s and increment q; otherwise, we add to s and decrement q.

In retrospect, the need for a correction cycle is easy to see: with the digit set $\{-1, 1\}$ we can represent only odd integers. So, if the quotient happens to be even, a correction is inevitable.

Figure 13.9 shows an example nonrestoring division with 2's-complement operands. The example illustrates all aspects of the nonrestoring division algorithm, including remainder correction and quotient conversion/correction. The reader is urged to examine Fig. 13.9 closely and to construct other examples for practice.

z		0	0	1	0	0	0	0	1	Dividend = $(33)_{ten}$
2^4d	1	1	0	0	1					Divisor = $(-7)_{ten}$
-2^4d	0	0	1	1	1					
$s^{(0)}$	0	0	0	1	0	0	0	0	1	
$2s^{(0)}$	0	0	1	0	0	0	0	1		sign($s^{(0)}$) ≠ sign(d),
$+2^4d$	1	1	0	0	1					so set q_3 = ⁻1 and add
$s^{(1)}$	1	1	1	0	1	0	0	1		
$2s^{(1)}$	1	1	0	1	0	0	1			sign($s^{(1)}$) = sign(d),
$+(-2^4d)$	0	0	1	1	1					so set q_2 = 1 and subtract
$s^{(2)}$	0	0	0	0	1	0	1			
$2s^{(2)}$	0	0	0	1	0	1				sign($s^{(2)}$) ≠ sign(d),
$+2^4d$	1	1	0	0	1					so set q_1 = ⁻1 and add
$s^{(3)}$	1	1	0	1	1	1				
$2s^{(3)}$	1	0	1	1	1					sign($s^{(3)}$) = sign(d),
$+(-2^4d)$	0	0	1	1	1					so set q_0 = 1 and subtract
$s^{(4)}$	1	1	1	1	0					sign($s^{(4)}$) ≠ sign(d)
$+(-2^4d)$	0	0	1	1	1					Corrective subtraction
$s^{(4)}$	0	0	1	0	1					
s						0	1	0	1	Remainder = $(5)_{ten}$
q						⁻1	1	⁻1	1	Ucorrected BSD quotient
p						0	1	0	1	⁻1s replaced by 0s
						/	/	/	/	
Shifted p					1	1	0	1	1	Add 1 to correct
$q_{2's\text{-}compl}$						1	1	0	0	Quotient = $(-4)_{ten}$

Fig. 13.9 Example of nonrestoring signed division.

Figure 13.10 shows a hardware realization of the sequential nonrestoring division algorithm. At the start of each cycle j, the partial remainder $s^{(j-1)}$ is shifted to the left, with its MSB moving into a special flip-flop. Except for the first cycle, the quotient digit is derived by XORing the sign of the divisor and the complement of the sign of the partial remainder. The latter is the same as c_{out} (since the two terms added to form the new partial remainder always are opposite in sign).

Once all the digits of q have been derived in k cycles, two to four additional cycles may be needed to correct the quotient q and the final remainder s. Implementation details depend on various hardware issues such as whether q in the quotient register (or lower half of the partial remainder register) can be directly input to the adder for correction or it should be moved to a different register to gain access to the adder. Practical implementation details, including a complete microprogram for nonrestoring division can be found elsewhere [Wase82, pp. 181-192].

13.5 DIVISION BY CONSTANTS

Justification for our discussion of division by constants is similar to that given for multiplication by constants in Section 9.5. The performance benefits of these methods is even more noticeable here, given that division is generally a slower operation than multiplication. In what follows, we

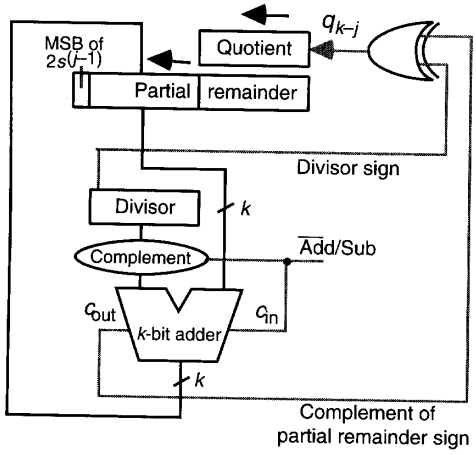


Fig. 13.10 Shift/subtract sequential nonrestoring divider.

consider only division by odd integers, since division by an even integer can be performed by first dividing by an odd integer and then shifting the result. For example, to divide by 20, one can divide by 5 and then shift the result right by two bit positions.

If only a limited number of constant divisors are of interest, their reciprocals can be precomputed with an appropriate precision and stored in a table. Then, the problem of division by any of these constants can be converted to that of multiplication by its constant reciprocal, using the methods discussed in Section 9.5.

Faster constant division routines can be obtained for many small odd divisors by using the mathematical property that for each odd integer d there exists an odd integer m such that $d \times m = 2^n - 1$. Thus:

$$\frac{1}{d} = \frac{m}{2^n - 1} = \frac{m}{2^n(1 - 2^{-n})}$$
$$= \frac{m}{2^n}(1 + 2^{-n})(1 + 2^{-2n})(1 + 2^{-4n}) \cdots$$

Note that the expansion of $1/(1 - 2^{-n})$ involves an infinite number of product terms of the form $1 + 2^{-2^i n}$. Thus to divide z by d, we need to multiply it by $m/2^n$ (which is itself a constant that can be precomputed for integer divisors of interest) and then by several factors of the form $1 + 2^{-j}$. The number of such factors is proportional to the logarithm of the word width and multiplication by each one involves a shift followed by an addition.

Consider as an example division by the constant $d = 5$. We find $m = 3$ and $n = 4$ by inspection. Thus, for 24 bits of precision, we have:

$$\frac{z}{5} = \frac{3z}{2^4 - 1} = \frac{3z}{16(1 - 2^{-4})}$$
$$= \frac{3z}{16}(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{-16})$$

Note that the next term $(1 + 2^{-32})$ would shift out the entire operand and thus does not contribute anything to a result with 24 bits of precision. Based on the preceding expansion, we obtain the following procedure, consisting of shift and add operations, to effect division by 5:

$q \leftarrow z + z$ shift-left 1 $\quad \{3z$ computed$\}$
$q \leftarrow q + q$ shift-right 4 $\quad \{3z(1 + 2^{-4})\}$
$q \leftarrow q + q$ shift-right 8 $\quad \{3z(1 + 2^{-4})(1 + 2^{-8})\}$
$q \leftarrow q + q$ shift-right 16 $\quad \{3z(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{16})\}$
$q \leftarrow q$ shift-right 4 $\quad \{3z(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{-16})/16\}$

The preceding algorithm uses five shifts and four additions to divide z by 5.

In an application reported over a decade ago [Li85], division by odd constants of up to 55 was frequently required. So the corresponding routines were obtained, fine-tuned, and stored in the system. An aspect of the fine-tuning involved compensating for truncation errors in the course of computations. For example, it was found, through experimentation, that replacing the first statement in the preceding algorithm (division by 5) by $q \leftarrow z + 3 + z$ shift-left 1 would minimize the truncation error on the average. Similar modifications were introduced elsewhere.

Simple hardware structures can be devised for division by certain constants [Scho97]. For example, one way to divide a number z by 3 is to multiply it by 4/3, shifting the result to the right by 2 bits to cancel the factor of 4. Multiplication by 4/3 can in turn be implemented by noting that the following recurrence has the solution $q = 4z/3$:

$$q^{(i)} = q^{(i-1)}/4 + z \quad \text{with} \quad q^{(0)} = 0$$

An alternative to computing q sequentially is to use the fact that q is the output of an adder with inputs $y = q/4$ (right-shifted version of the adder's output) and z. The problem with this implementation strategy is that feeding back the output q_i to the input y_{i-2} creates a feedback loop, given carry propagation between the positions $i - 2$ and i. However, the feedback loop can be eliminated by using a carry-save adder instead of a carry-propagate adder. Working out the implementation details is left as an exercise.

13.6 PREVIEW OF FAST DIVIDERS

Like multiplication, sequential division can be viewed as a multioperand addition problem (Fig. 13.11). Thus, there are but two ways to speed it up:

Reducing the number of operands to be added.

Adding the operands faster.

Reducing the number of operands leads to high-radix division. Adding them faster leads to the use of carry-save representation of the partial remainder. One complication makes division more difficult and thus slower than multiplication: the terms to be subtracted from (added to) the dividend z are not known a priori but become known as the quotient digits are computed. The quotient digits are in turn dependent on the relative magnitudes of the intermediate partial remainders and the divisor (or at least the sign of the partial remainder in the radix-2 nonrestoring algorithm). With carry-save representation of the partial remainder, the magnitude or sign information is no longer readily available; rather, it requires full carry propagation in the worst case.

High-radix dividers, introduced in Chapter 14 and further developed in Chapter 15, produce several bits of the quotient, and multiply them by the divisor, at once. Speedup is achieved for

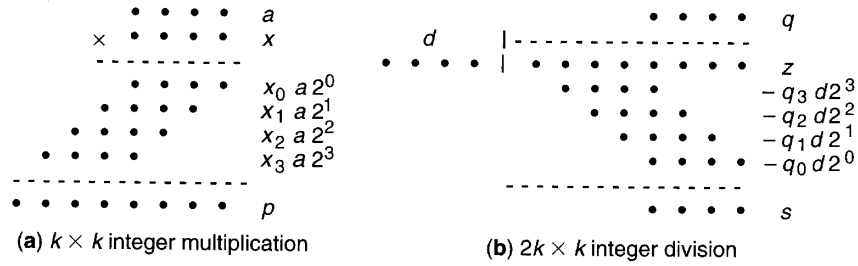


Fig. 13.11 (a) Multiplication and (b) division as multioperand addition problems.

radix 2^j as long as each radix-2^j division cycle is less than j times as long as a radix-2 division cycle. A key issue in the design of high-radix dividers is the selection of the next quotient digit directly from a few bits of the carry-save partial remainder, thus postponing full carry propagation to the very end.

Because of the sequential nature of quotient digit production, there is no counterpart to tree multipliers in the design of dividers. However, array dividers do exist and are discussed in Chapter 15, along with some variations in the design of dividers and combined multiplier/divider units.

Of course there is no reason to limit ourselves to the use of shift and add/subtract operations for implementing dividers. We will see, in Chapter 15, that division by repeated multiplications can be quite cost-effective and competitive in speed, especially when one or more fast parallel multipliers are available.

PROBLEMS

13.1 Unsigned decimal division Perform the division z/d for the following dividend/divisor pairs, obtaining the quotient q and the remainder s. Present your work in tabular form, as in Fig. 13.2.

a. $a = 1234\ 5678$ and $x = 4321$

b. $a = .1234\ 5678$ and $x = .4321$

13.2 Programmed nonrestoring division Write a program similar to the one in Fig. 13.4 for nonrestoring division. Compare the running time of your program to the restoring version and discuss.

13.3 Programmed restoring division

a. Modify the division program of Fig. 13.4 for the case in which both the dividend and the divisor are k bits wide. Analyze the running time of the new program.

b. Modify the division program of Fig. 13.4 to correspond to true restoring division, where subtraction is always performed, but the partial remainder is restored to its original value via addition if it becomes negative. Compare the running time of your modified program to the original one and discuss.

13.4 Fixed-time programmed division We would like to modify the division program of Fig. 13.4 so that it always takes the same number of machine cycles to execute, provided a divide-by-zero or overflow exception does not occur. We do not know the number of machine cycles taken by each instruction, but any particular instruction always takes the

same number of cycles. Suggest the required modifications in the program and compare the running time of the resulting program to the original one.

13.5 Unsigned sequential restoring division Perform the division z/d for the following dividend/divisor pairs, obtaining the quotient q and the remainder s. Use the restoring algorithm and present your work in tabular form, as in Fig. 13.6.

a. $z = 0101$ and $d = 1001$

b. $z = .0101$ and $d = .1001$

c. $z = 1001\ 0100$ and $d = 1101$

d. $z = .1001\ 0100$ and $d = .1101$

13.6 Sequential nonrestoring division

a. After complementing z, redo the division example of Fig. 13.7.

b. After complementing both z and d, redo the division example of Fig. 13.7.

13.7 Sequential nonrestoring division Represent the following signed-magnitude dividends and divisors in 5-bit, 2's-complement format and then perform the division using the nonrestoring algorithm. In each case, convert the quotient to 2's-complement format.

a. $z = +.1001$ and $d = +.0101$

b. $z = +.1001$ and $d = -.0101$

c. $z = -.1001$ and $d = +.0101$

d. $z = -.1001$ and $d = -.0101$

13.8 Sequential multiplication/division Assuming 2's-complement binary operands:

a. Perform the division $z/d = 1.100/0.110$ and obtain the 4-bit, 2's-complement quotient q and remainder s using the nonrestoring method.

b. Check your answer to part a by doing the 2's-complement multiplication $d \times q$, with q as the multiplier, and adding the remainder s to the resulting product.

c. Use the restoring method to perform the division of part a.

13.9 Radix-2 unsigned integer division Given the binary dividend $z = 0110\ 1101\ 1110\ 0111$ and the divisor $d = 1010\ 0111$, perform the unsigned division z/d to determine the 8-bit quotient q and remainder s using both the restoring and nonrestoring algorithms.

13.10 Radix-2 signed division Given the binary 2's-complement operands $z = 1.1010\ 0010\ 11$ and $d = 0.10110$, use both the restoring and nonrestoring algorithms to perform the division z/d to find the 2's-complement quotient $q = q_0.q_{-1}q_{-2}q_{-3}q_{-4}q_{-5}$ and remainder $1.11111s_{-6}s_{-7}s_{-8}s_{-9}s_{-10}$. Present your work in tabular form as in Fig. 13.9.

13.11 Nonrestoring hardware dividers By analyzing all eight possible combinations of signs for the dividend, divisor, and final remainder, along with the corrective actions required in each case, propose an efficient hardware design for a nonrestoring divider. *Hint:* Based on the sign of the final remainder, produce an extra bit q_{-1} of the quotient,

which becomes the LSB of the left-shifted p in converting to 2's-complement. Then, only negative quotients will need correction [Wase82, pp. 183–186].

13.12 Division by constants Using shift and add/subtract instructions only, devise efficient routines for division by the following constants. Assume 32-bit unsigned operands.

a. 19

b. 43

c. 88

d. 129 (*Hint:* $2^{14} - 1 = 127 \times 129$.)

13.13 Division by special constants

a. Discuss the division of unsigned binary numbers by constants of the form $2^b \pm 1$.

b. Extend the procedure of part a to the case of a divisor that can be factored into a product of terms, each of which is of the form $2^b \pm 1$ [e.g., $45 = (2^2 + 1)(2^3 + 1)$].

c. Apply the method of part b to division by 99, with 32 bits of precision.

d. Compare the result of part c to that obtained from the method discussed in Section 13.5.

13.14 Division by special constants

a. Devise general strategies for dividing z by positive constants of the form $2^j - 2^i$, where $0 < i < j$ (e.g., $62 = 2^6 - 2^1$, $28 = 2^5 - 2^2$).

b. Repeat part a for constants of the form $2^j + 2^i$.

13.15 Fully serial dividers

a. A fully serial, nonrestoring divider is obtained if the adder of Fig. 13.10 is replaced with a bit-serial adder. Show the block diagram of the fully serial divider based on the nonrestoring division algorithm.

b. Design the required control circuit for the fully serial divider of part a.

c. Does it make sense to build a fully serial divider based on the restoring algorithm?

13.16 Hardware for division by constants A simple hardware scheme for dividing z by certain constants was discussed at the end of Section 13.5 [Scho97].

a. Supply the details of the required circuit for computing $z/3$.

b. Outline the algorithm and hardware requirements for dividing z by 5.

c. Characterize the class of constants for which this scheme can be used.

REFERENCES

[Kore93] Koren, I., *Computer Arithmetic Algorithms*, Prentice-Hall, 1993.

[Li85] Li, R. S.-Y., "Fast Constant Division Routines," *IEEE Trans. Computers*, Vol. 34, No. 9, pp. 866–869, 1985.

[Omon94] Omondi, A. R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementation*, Prentice-Hall, 1994.

[Scho97] Schoner, B., and S. Molloy, "A New Architecture for Area-Efficient Multiplication by a Class of Rational Coefficients," *Proc. Midwest Symp. Circuits and Systems*, August 1997, Vol. 1, pp. 373–376.

[Wase82] Waser, S., and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart, & Winston, 1982.

Chapter 14 HIGH-RADIX DIVIDERS

In this chapter, we review division schemes that produce more than one bit of the quotient in each cycle (2 bits per cycle in radix 4, 3 bits in radix 8, etc.). The reduction in the number of cycles, along with the use of carry-save addition to simplify the required computations in each cycle, leads to significant speed gain over the basic restoring and nonrestoring dividers discussed in Chapter 13. Chapter topics include:

14.1 BASICS OF HIGH-RADIX DIVISION

Recall, from Chapter 13, that the equation $z = (d \times q) + s$, along with the two conditions $\text{sign}(s) = \text{sign}(z)$ and $|s| < |d|$, completely defines the results q (quotient) and s (remainder) of fixed-point division.

The radix-r counterpart of the binary division recurrence, derived in Section 13.1, can be written as follows:

$$s^{(j)} = rs^{(j-1)} - q_{k-j}(r^k d) \quad \text{with} \quad s^{(0)} = z \quad \text{and} \quad s^{(k)} = r^k s$$

where the radix-r division parameters are:

z	Dividend	$z_{2k-1}z_{2k-2}\cdots z_1 z_0$
d	Divisor	$d_{k-1}d_{k-2}\cdots d_1 d_0$
q	Quotient	$q_{k-1}q_{k-2}\cdots q_1 q_0$
s	Remainder $[z - (d \times q)]$	$s_{k-1}s_{k-2}\cdots s_1 s_0$

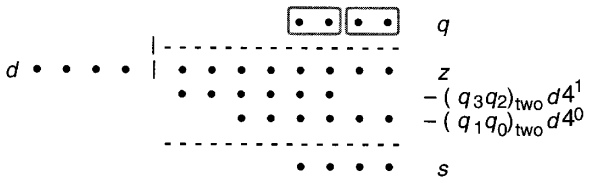


Fig. 14.1 Radix-4 division in dot notation.

High-radix dividers of practical interest have $r = 2^b$ (and, occasionally, $r = 10$). Consider, for example, radix-4 division. Each radix-4 quotient digit, obtained in one division cycle, represents two radix-2 digits. So, radix-4 division can be viewed as radix-2 division with 2 bits of the quotient obtained in each cycle. In an 8-by-4 binary division performed in radix 4, for example, q_3 and q_2 are determined first, with $(q_3q_2)_{two}(4^2d)$ subtracted from $4z$ to obtain the first partial remainder. This partial remainder is then used for determining q_1 and q_0 in the second and final cycle. Figure 14.1 shows the preceding radix-4 division in dot notation.

Figure 14.2 depicts examples of radix-4 and radix-10 division. The radix-4 division example shown has $z = (7003)_{ten} = (0123\ 1123)_{four}$ and $d = (99)_{ten} = (1203)_{four}$, yielding the quotient $q = (70)_{ten} = (1012)_{four}$ and the remainder $s = (73)_{ten} = (1021)_{four}$. The radix-10 example corresponds to the division $(.7003)_{ten}/(.99)_{ten}$, yielding $q = (.70)_{ten}$ and $s = (.0073)_{ten}$.

Dividing binary numbers in radix 2^b reduces the number of cycles required by a factor of b, but each cycle is more difficult to implement because:

Radix-4 integer division

z	0 1 2 3	1 1 2 3
4^4d	1 2 0 3	
$s^{(0)}$	0 1 2 3	1 1 2 3
$4s^{(0)}$	0 1 2 3 1	1 2 3
$-q_3 4^4 d$	0 1 2 0 3	$\{q_3 = 1\}$
$s^{(1)}$	0 0 2 2	1 2 3
$4s^{(1)}$	0 0 2 2 1	2 3
$-q_2 4^4 d$	0 0 0 0 0	$\{q_2 = 0\}$
$s^{(2)}$	0 2 2 1	2 3
$4s^{(2)}$	0 2 2 1 2	3
$-q_1 4^4 d$	0 1 2 0 3	$\{q_1 = 1\}$
$s^{(3)}$	1 0 0 3	3
$4s^{(3)}$	1 0 0 3 3	
$-q_0 4^4 d$	0 3 0 1 2	$\{q_0 = 2\}$
$s^{(4)}$	1 0 2 1	
s		1 0 2 1
q		1 0 1 2

Radix-10 fractional division

z_{frac}	.7 0 0 3	
d_{frac}	.9 9	
$s^{(0)}$	.7 0 0 3	
$10s^{(0)}$	7.0 0 3	
$-q_{-1}d$	6.9 3	$\{q_{-1} = 7\}$
$s^{(1)}$	.0 7 3	
$10s^{(1)}$	0.7 3	
$-q_{-2}d$	0.0 0	$\{q_{-2} = 0\}$
$s^{(2)}$	.7 3	
s_{frac}	.0 0 7 3	
q_{frac}	.7 0	

Fig. 14.2 Examples of high-radix division with integer and fractional operands.

a. The higher radix makes the guessing of the correct quotient digit more difficult; we certainly do not want to try subtracting $2^k d, 2(2^k d), 3(2^k d)$, etc., and noting the sign of the partial remainder in each case, until the correct quotient digit has been determined—this would nullify all the speed gain (in radix 4, two trial subtractions of d and $2d$ would be needed, thus making each cycle almost twice as long with one adder).

b. Unlike multiplication, where all the partial products can be computed initially and then subjected to parallel processing by multiple carry-save adders, the values to be subtracted from (added to) z in division are determined sequentially, one per cycle. Furthermore, the determination of the quotient digits depends on the magnitude and/or sign of the partial remainder; information that is not readily available from the stored-carry representation.

Thus before discussing high-radix division in depth, we try to solve the more pressing problem of using carry-save techniques to speed up the iterations in binary division. Once we have learned how to use a carry-save representation for the partial remainder, we will revisit the problem of high-radix division. The reason we attach greater importance to the use of carry-save partial remainders than to high-radix division is that in going from radix 2 to radix 4, say, the division is at best speeded up by a factor of 2. The use of carry-save partial remainders, on the other hand, can lead to a larger performance improvement via replacing the delay of a carry-propagate adder by the delay of a single full adder.

The key to being able to keep the partial remainder in carry-save form is introducing redundancy in the representation of the quotient. With a nonredundant quotient, there is no room for error. If the binary quotient is $(0110\cdots)_{\text{two}}$, say, subsequent recovery from an incorrect guess setting the MSB of q to 1 will be impossible. However, if we allow the digit set $[-1, 1]$ for the radix-2 quotient, the partial quotient $(1\cdots)_{\text{two}}$ can be modified to $(1\ \bar{1}\cdots)_{\text{two}}$ in the next cycle if we discover that 1 was too large a guess for the MSB. The aforementioned margin for error allows us to guess the next quotient digit based on the approximate magnitude of the partial remainder. The greater the margin for error, the less precision (fewer bits of the carry-save partial remainder) we need in determining the quotient digits.

14.2 RADIX-2 SRT DIVISION

Let us reconsider the radix-2 nonrestoring division algorithm for fractional operands characterized by the recurrence

$$s^{(j)} = 2s^{(j-1)} - q_{-j}d \quad \text{with} \quad s^{(0)} = z \quad \text{and} \quad s^{(k)} = 2^k s$$

with $q_{-j} \in \{-1, 1\}$. Note that the same algorithm can be applied to integer operands if d is viewed as standing for $2^k d$.

The quotient is obtained with the digit set $\{-1, 1\}$ and is then converted (on the fly) to the standard digit set $\{0, 1\}$. Figure 14.3 plots the new partial remainder, $s^{(j)}$, as a function of the shifted old partial remainder, $2s^{(j-1)}$. For $2s^{(j-1)} \geq 0$, we subtract the divisor d from $2s^{(j-1)}$ to obtain $s^{(j)}$, while for $2s^{(j-1)} < 0$, we add d to obtain $s^{(j)}$. These actions are represented by the two oblique lines in Fig. 14.3. The heavy dot in Fig. 14.3 indicates the action taken for $2s^{(j-1)} = 0$.

Nonrestoring division with shifting over 0s is a method that avoids addition or subtraction when the partial remainder is "small." More specifically, when $2s^{(j-1)}$ is in the range $[-d, d)$,