NUMBER REPRESENTATION

"Mathematics, like the Nile, begins in minuteness, but ends in magnificence."
CHARLES CALEB COLTON

"Of all the great things that are found among us the existence of nothing is the greatest."
LEONARDO DA VINCI

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:

1 Numbers and Arithmetic

"Mathematics is the queen of the sciences and arithmetic is the queen of mathematics."
CARL FRIEDRICH GAUSS

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 was interested in twin primes (consecutive odd numbers such as 29 and 31 that are both prime). Nicely's work involved 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 (named for Sweeny, Robertson, and Tocher) algorithm and came up with an example that produces the worst-case error. Using double-precision floating-point computation, the ratio $c = 4\ 195\ 835/3\ 145\ 727 = 1.333\ 820\ 44\cdots$ was 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 than 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 game-playing programs), and many other aspects of computer technology have made the news regularly, 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.

Much of our discussion relates 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 input/output 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

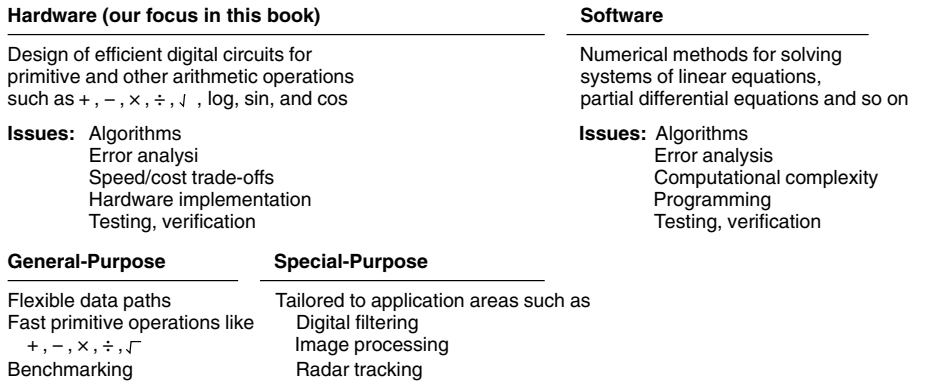

Figure 1.1 The scope of computer arithmetic.

be able to render informed judgment when faced with the problem of choosing a digital signal processor 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 a Pentium processor or a software tool such as Mathematica), and the task is 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, 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 MOTIVATING EXAMPLES

Use a calculator that has the square-root, square, and exponentiation (x^y) functions to perform the following computations. Numerical results, obtained with a $(10 + 2)$-digit scientific calculator, are provided. 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:

$$x = \underbrace{\Big(\Big((u^2)^2\Big)\cdots\Big)^2}_{10 \text{ times}} = 1.999\ 999\ 963$$

$$x' = u^{1024} = 1.999\ 999\ 973$$

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

$$y = \underbrace{\Big(\Big((v^2)^2\Big)\cdots\Big)^2}_{10 \text{ times}} = 1.999\ 999\ 983$$

$$y' = v^{1024} = 1.999\ 999\ 994$$

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 (the author's 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.

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

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 8 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].

The example calculations demonstrate that familiarity with computer arithmetic is helpful for appreciating and correctly interpreting our everyday dealings with numbers. There is much more to computer arithmetic, however. Inattention to fundamentals of this field has led to several documented, and no doubt many more unreported, disasters. In the rest of this section, we describe two such events that were caused by inadequate precision and unduly limited range of numerical results.

The first such event, which may have led to the loss of 28 human lives in February 1991, is the failure of the American Patriot missile battery in Dhahran, Saudi Arabia, to intercept a number of Iraqi Scud missiles. An investigation by the US General Accounting Office [GAO92] blamed the incident on a "software problem" that led to inaccurate calculation of the elapsed time since the last system boot. It was explained that the system's internal clock measured time in tenths of a second. The measured time was then multiplied by a 24-bit truncated fractional representation of 1/10, with an error of about $(3/4) \times 10^{-23} \approx 10^{-7}$. Some error was unavoidable, because 1/10 does not have an exact binary representation. Though rather small, when accumulated over a 10-hour operation period, this error caused the calculated time to be off by roughly 1/3 of a second. Because the Scud missile flew at a speed of about 1700 m/s, its calculated position might have differed from its actual position by more than 1/2 km; an error that is large enough to cause a missed interception.

The second such event is the explosion of an Ariane 5 rocket 30 seconds after liftoff in June 1996. Fortunately, this incident, also attributed to a "software error" [Lion96], did not lead to any loss of life, but its price tag was the embarrassing collapse of an ambitious development project costing US $7 billion. According to the explanations offered, at some point in the control program, a 64-bit floating-point number pertaining to the horizontal velocity of the rocket was to be converted to a 16-bit signed integer. Because the floating-point number had a value greater than what could fit in a 16-bit signed integer, an overflow exception arose that did not have adequate handling provisions by the software. This caused a processor shutdown, which triggered a cascade of events leading to improper attempts at course correction and the eventual disintegration that spread debris over several square kilometers. The doomed conversion routine was a leftover from the software used for the Ariane 4 rocket, carried over intact according to the maxim "if it ain't broke, don't fix it." However, the designers failed to take into account that within the initial 40 seconds of flight when the system in question was active, the Ariane 5 rocket could reach a horizontal velocity that was about five times that of the Ariane 4.

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 the notation 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 $\{-2, -1, 0, 1\}$ and 2-bit integer significand s in $\{0, 1, 2, 3\}$, 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.

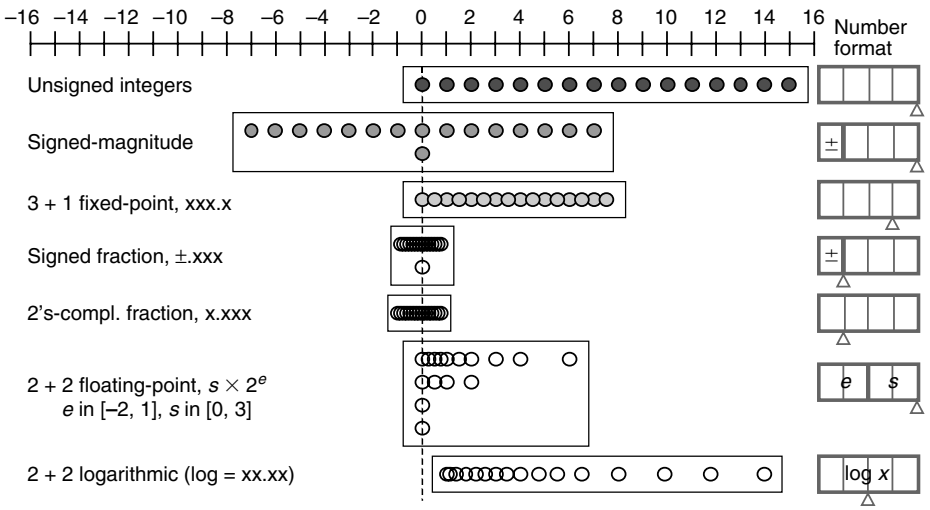

Figure 1.2 Some of the possible ways of assigning 16 distinct codes to represent numbers. Small triangles denote the radix point locations.

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

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

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.

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

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

$$(\cdots x_5x_4x_3x_2x_1x_0\,.\,x_{-1}x_{-2}x_{-3}x_{-4}x_{-5}x_{-6}\cdots)_{-r} = \sum_i x_i(-r)^i$$

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

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

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,

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

$(\bar{}3\ 1\ 5)_{\text{ten}}$ 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:

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

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].

$$(x_{k-1}x_{k-2}\cdots x_1x_0\,.\,x_{-1}x_{-2}\cdots x_{-l})_{\text{one-tenth}} = \sum_i x_i 10^{-i}$$

$$= (x_{-l}\cdots x_{-2}x_{-1}x_0\,.\,x_1x_2\cdots x_{k-2}x_{k-1})_{\text{ten}}$$

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

$$(\cdots x_5x_4x_3x_2x_1x_0\,.\,x_{-1}x_{-2}x_{-3}x_{-4}x_{-5}x_{-6}\cdots)_{\sqrt{2}} = \sum_i x_i(\sqrt{2})^i$$

$$= (\cdots x_4x_2x_0\,.\,x_{-2}x_{-4}x_{-6}\cdots)_{\text{two}} + \sqrt{2}(\cdots x_5x_3x_1\,.\,x_{-1}x_{-3}x_{-5}\cdots)_{\text{two}}$$

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].

$$(\cdots x_5x_4x_3x_2x_1x_0\,.\,x_{-1}x_{-2}x_{-3}x_{-4}x_{-5}x_{-6}\cdots)_{2j} = \sum_i x_i(2j)^i$$

$$= (\cdots x_4x_2x_0\,.\,x_{-2}x_{-4}x_{-6}\cdots)_{-\text{four}} + 2j(\cdots x_5x_3x_1\,.\,x_{-1}x_{-3}x_{-5}\cdots)_{-\text{four}}$$

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 represent 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 the 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:

$$\begin{aligned} u &= w.v \\ &= (x_{k-1} x_{k-2} \cdots x_1 x_0 . x_{-1} x_{-2} \cdots x_{-l})_r \\ &= (X_{K-1} X_{K-2} \cdots X_1 X_0 . X_{-1} X_{-2} \cdots X_{-L})_R \end{aligned}$$

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 $(105.486)_{\text{ten}} \approx (410.220\ 33)_{\text{five}}$.

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 $x_{k-1}r^{k-1} + x_{k-2}r^{k-2} + \cdots + x_1 r + x_0$ [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 $r^l \times (0.v)$ and then divide by r^l in the new radix.

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

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 $(410.220\ 33)_{\text{five}} = (105.485\ 76)_{\text{ten}}$.

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.

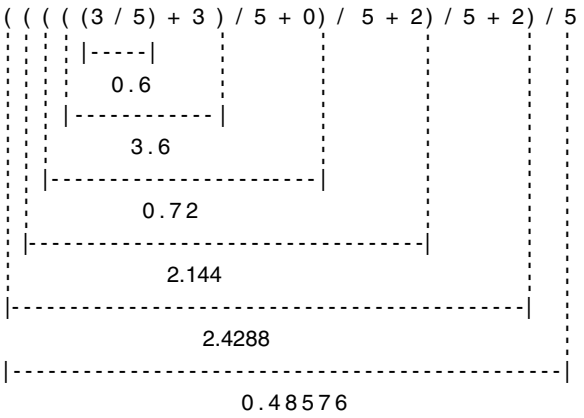

Figure 1.3 Horner's rule used to convert $(.220\ 33)_{\text{five}}$ to decimal.

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.

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.

$$(2\ 301.302)_{\text{four}} = (\underline{10}\ \underline{11}\ \underline{00}\ \underline{01}\ .\ \underline{11}\ \underline{00}\ \underline{10})_{\text{two}}$$
$$= (\underline{10}\ \underline{110}\ \underline{001}\ .\ \underline{110}\ \underline{010})_{\text{two}} = (261.62)_{\text{eight}}$$

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.

This is a good place to introduce a notational tool, that we will find quite useful throughout the book. The established dot notation uses heavy dots to represent standard or positively-weighted bits, which we may call posibits. For example, Fig. 1.4a represents the addition of two 4-bit unsigned binary numbers whose posibits have weights 1, 2, 2^2, and 2^3, from right to left, and whose sum is a 5-bit number. Figure 1.4b depicts the pencil-and-paper algorithm for multiplying two 4-bit unsigned binary numbers, producing four partial products and then adding them, with proper alignments, to derive the 8-bit final result. We will see later that negatively weighted bits, or negabits, are also quite useful, prompting us to introduce the extended dot notation (see Section 2.6).

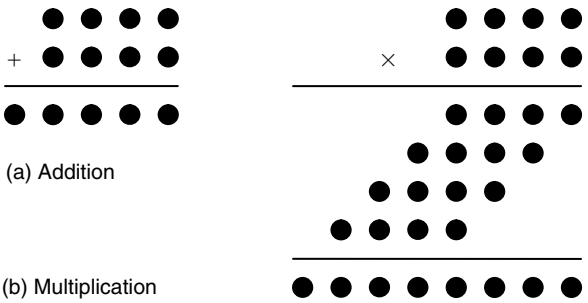

Figure 1.4 Dot notation to depict number representation formats and arithmetic algorithms.

A final point before we conclude this chapter: You can be a proficient arithmetic designer knowing only the following three key number representation systems and their properties:

2's-complement format (Section 2.4)
Binary stored-carry or carry-save format (Section 3.2)
Binary floating-point format (Chapter 17)

All the other formats, discussed in Chapters 2-4, are useful for optimizing application-specific designs or to gain a deeper understanding of the issues involved, but you can ignore them with no serious harm. There are indications, however, that decimal arithmetic may regain the importance it once had, because it avoids errors in the conversion between human-readable numbers and their machine 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 nonzero 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 $(0001\ 1111\ 0010\ 1101)_{-\text{two}}$ 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 $(48\text{A.C2})_{\text{sixteen}}$ and $(192.837)_{\text{ten}}$ 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 1 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 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 width 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 1 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.

1.17 Infinite representations

Consider a radix-r ($r \geq 2$) fixed-point number representation scheme with infinitely many digits to the left and to the right of the radix point.

a. Show that the number represented is rational if and only if the fractional part is ultimately periodic.
b. Characterize the class of rational numbers that have two different representations.
c. Repeat part b for the negative radix $-r$.

1.18 Number radix conversion

a. Show that any number that is finitely representable in binary also has a finite decimal representation.
b. Derive a relationship between the radices r and R such that any number with a finite radix-r representation also has a finite representation in radix R.

1.19 Number representation

Prove or disprove each of the following statements for a rational number $a = b/c$, where b and c are relatively prime integers, with $b \geq 1$ and $c \geq 2$.

a. In an even radix r, the rational number a does not have an exact finite representation if c is odd.
b. In an odd radix r, the rational number a does not have an exact finite representation if c is even.
c. It is possible to represent the rational number a exactly in radix r, using k whole and l fractional digits, if and only if $b < r^{k+l}$ and $c \leq r^l$.

1.20 Number representation

We want to build an abacus for use with the Roman numeral system. There are to be seven positions labeled, from left to right, M, D, C, L, X, V, and I. Each position is to have positive (black) and negative (red) beads to allow representations such as MCDXXIV. What are the minimum required numbers of the two types of beads in each position, given that all unsigned integers up to 1500 are to be representable?

1.21 Compressed decimal numbers

One way to represent decimal numbers in memory is to pack two BCD digits into 1 byte. This representation is somewhat wasteful in that a byte that can encode 256 values is used to represent the digit pairs 00 through 99. One way of improving efficiency is to compress three BCD digits into 10 bits.

a. Devise a suitable encoding for this compression. *Hint:* Let the BCD digits be $x_3x_2x_1x_0$, $y_3y_2y_1y_0$, and $z_3z_2z_1z_0$. Let the 10-bit encoding be $WX_2X_1x_0Y_2Y_1y_0Z_2Z_1z_0$. In other words, the three least-significant bits of the digits are used directly and the remaining 9 bits (3 from each digit) are encoded into 7 bits. Let $W = 0$ encode the case $x_3 = y_3 = z_3 = 0$. In this case, the remaining digits are simply copied in the new representation. Use $X_2X_1 = 00, 01, 10$ to encode the case where only one of the values x_3, y_3, or z_3 is 1. Note that when the most-significant bit of a BCD digit is 1, the digit

is completely specified by its least-significant bits and no other information is needed. Finally, use $X_2X_1 = 11$ for all other cases.

b. Design a circuit to convert three BCD digits into the 10-bit compressed representation.

c. Design a circuit to decompress the 10-bit code to retrieve the three original BCD digits.

d. Suggest a similar encoding to compress two BCD digits into 7 bits.

e. Design the required compression and decompression circuits for the encoding of part d.

1.22 Double-base number systems

Consider the representation of integers by a $k \times m$ matrix of bits, where the bit in row i, column j being 1 indicates that 2^i3^j is included in the sum defining the represented integer x. This corresponds to a double-base number system with the two bases being 2 and 3 [Dimi03]. For example, if a 4×4 matrix is used, and the bits in the matrix are written in row-major order, the number $54 = 2^23^2 + 2^03^2 + 2^13^1 + 2^13^0 + 2^03^0$ can be represented as 1010 1100 0010 0000.

a. In what way are the binary and ternary number systems special cases of the above?

b. Compute *max*, the largest number representable in a double-base (2 and 3) number system as a function of k and m.

c. Show that all unsigned integers up to *max* are representable in the number system of part b. *Hint:* Prove that if $x > 0$ is representable, so is $x - 1$.

d. Show that any representation can be easily transformed so that it does not contain two consecutive 1s in the same row or the same column. Representations that are thus transformed are said to be "addition-ready."

e. Assuming that the transformation of part d is applied after every arithmetic operation, derive an addition algorithm for such numbers.

1.23 Symmetric digit sets

We know that for any odd radix r, the symmetric digit set $[-(r-1)/2, (r-1)/2]$ is adequate for representing all numbers, leads to unique representations, and offers some advantages over the conventional digit set $[0, r-1]$. The balanced ternary number system of Example 1.1 is one such representation. Show that for an even radix r, the symmetric fractional digit set $\{-r/2 + 1/2, \ldots, -1/2, 1/2, \ldots, r/2 - 1/2\}$ is adequate for representing all numbers and discuss some practical limitations of such a number representation system.

1.24 The Cantor set C_0

The Cantor set C_0, a sparse subset of the set of real numbers in [0, 1], is defined as follows. Beginning with the single interval [0, 1], repeat the following process indefinitely. Divide each remaining interval (initially only one) into three equal parts. Of the three subintervals, remove the middle one, except for its endpoints; that is, leave the first and third ones as closed intervals.

a. Show that C_0 consists of real numbers that can be represented as infinite ternary fractions using only the digits 0 and 2.
b. Show that the numbers 1/4, 3/4, and 1/13 are in C_0.
c. Show that any number in $[-1, 1]$ is the difference between two numbers in C_0.

1.25 Fixed-radix positional number systems

Let $N_{k,r}$ be an integer whose k-digit radix-r representation is all 1s, that is, $N_{k,r} = (1\ 1 \cdots 1)_r$, where the number of 1 digits is k.

a. Show the radix-2 representation of the square of $N_{k,2}$.
b. Prove that except for $N_{1,10}$, no $N_{i,10}$ is a perfect square.
c. Show that $N_{i,r}$ divides $N_{j,r}$ if and only if i divides j.

1.26 Fixed-radix positional number systems

Show that the number $(1\ 0\ 1\ 0\ 1)_r$ is not a prime, regardless of the radix r.

1.27 Computer with ternary number representation

The TERNAC computer, implemented at State University of New York, Buffalo in 1973, had a 24-trit integer format and a 48-trit floating-point (42 for mantissa, 6 for exponent) format. It was intended as a feasibility study for ternary arithmetic. Prepare a two-page report on TERNAC, describing its arithmetic unit design and discussing whether it proved to be competitive in speed and cost.

1.28 Arithmetic algorithms

The computation of $f = (333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + a/(2b)$, for $a = 77\ 617$ and $b = 33\ 096$, is known as Rump's example.

a. Without rearranging the terms, compute f, using 32-bit, 64-bit, and, if possible, 128-bit floating-point arithmetic.
b. Compute the exact value of f, using the observation that the values chosen for a and b satisfy $a^2 = 5.5b^2 + 1$ [Loh02].
c. Compare the results of parts a and b and discuss.

REFERENCES AND FURTHER READINGS

[Dimi03] Dimitrov, V. S., and G. A. Jullien, "Loading the Bases: A New Number Representation with Applications," *IEEE Circuits and Systems*, Vol. 3, No. 2, pp. 6–23, 2003.

[GAO92] General Accounting Office, "Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia," US Government Report GAO/IMTEC-92-26, 1992.

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

[Lion96] Lions, J. L., "Ariane 5 Flight 505 Failure," Report by the Inquiry Board, July 19, 1996.

[Loh02] Loh, E., and G. W. Walster, "Rump's Example Revisited," *Reliable Computing*, Vol. 8, pp. 245–248, 2002.

[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.

[Parh02] Parhami, B., "Number Representation and Computer Arithmetic," *Encyclopedia of Information Systems*, Academic Press, Vol. 3, pp. 317–333, 2002.

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

[Silv06] Silverman, J. H., *A Friendly Introduction to Number Theory*, Pearson, 2006.

[Stol04] Stoll, C., "The Curious History of the First Pocket Calculator,"*Scientific American*, Vol. 290, No. 1, pp. 92–99, January 2004.

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