Representation of Numbers and Performance of Arithmetic in Digital Computers

Charles Abzug, Ph.D.
Department of Computer Science
James Madison University, MSC 4103
Harrisonburg, VA 22807

Voice Phone: ***540-568-8746,*** ***FAX: 540-568-8746***
E-mail: ***CharlesAbzug@ACM.org***
Home Page: ***http://www.cs.jmu.edu/users/abzugcx***

15 Aug 1999; last revised 21 Apr 2005

15 Aug 1999; last revised 21 Apr 2005

Table of Contents

Introduction

Some futurists have speculated that thousands of years into the future the human race may evolve to the point where our legs will atrophy, losing much of their size, power and endurance. The reason for this is that in modern society we tend to rely heavily on mechanical devices, such as automobiles, for much of our transportation needs, thus making far less use of our leg muscles than did our ancestors. Therefore, they speculate that our legs might eventually evolve into almost vestigial appendages, much like what has happened to the appendix in our gastrointestinal tract.

Over the span of 40+ years of my personal teaching experience, I have observed a substantial atrophy of some of the brain function of the modern student. When I attended high school and college, **every** student whom I encountered was able to carry out simple arithmetic operations, including addition, subtraction, multiplication and division, using manual methods, with considerable accuracy and at a reasonable rate of speed. Over forty years ago, there were no electronic calculators. A professional accountant or bookkeeper, an actuary, or a mathematician might have been fortunate enough to have had access to either a mechanical or possibly an electromechanical contrivance, called variously either an adding machine or a calculator. This device would enable him/her merely to enter the numbers, and then the machine would then take over and actually perform the calculation. Such a contrivance typically weighed twenty-five pounds or more, and took up a goodly portion of the space on a desktop. It was powered either mechanically, by means of energy supplied manually by the operator by repetitively pulling a lever over a distance of a foot or more, supplemented by a spring return, or else it was powered electrically via electricity supplied via a wall outlet from a central electric utility, just the same as electrical power is provided even today to ordinary household appliances. In either case, the contrivance was very noisy as well as very slow. Unless we had a lengthy column of numbers to add, it was often both faster and easier to perform calculations by hand with the aid of pencil and paper.

Today, it is my practice to ask my students in college to perform what I consider to be simple arithmetic calculations using their old-fashioned, built-in computer, i.e., their brain. When I do this, I usually encounter a chorus of indignant protest. The modern student is accustomed already from elementary school to performing arithmetic using an electronic calculator. Powered either by small batteries or via photovoltaic generation of electricity from ambient light, the device is both small and light enough to be carried conveniently in the hand, can be taken either to the beach or to the top of Mt. Everest or anywhere else where standard power from the electric utility is not readily available, and produces reliable, accurate results, provided that the data are input correctly and that the ambient temperature doesn't extend beyond the operating range declared by the manufacturer in the specifications listed in the instruction manual.

The modern student does not recognize any deficiency on his part because of his/her lack of skill in manually performing arithmetic calculations.

Checking the Correctness of Arithmetic Calculations

There is a problem that we need to contend with whenever we make use of some sort of electronic device to carry out an arithmetic calculation: how can we assure ourselves that the result provided by the calculator is correct? When we perform a calculation manually, we use well-established and accepted methods. Therefore, to confirm the correctness of a manual calculation the main task is to check either by performing the calculation independently a second time or by performing an alternative calculation. It is not enough that the overall methodology used for effecting a calculation is correct; we must also assure ourselves that we have not introduced errors in transcribing the numbers or in mentally carrying out the various small component calculations that contribute to the overall result. To obtain assurance of correctness, we might perform the calculation several times, introducing modest variations in how we formulate the problem, and checking wither identical results are obtained. To confirm the correctness of a multiplication, for example, we might interchange the multiplier and the multiplicand and check to see whether we obtain the same product as before. To confirm the correctness of an addition, we might interchange the augend and the addend and check to see whether we obtain the same sum as before. To confirm the correctness of a subtraction, we might add the subtrahend and the difference, and see whether the sum so obtained is equal to the minuend of the original calculation.

With an electronic device, however, there are several additional considerations that enter into the determination of whether or not the result of the calculation is correct. In order to verify the correctness of an electronically-performed calculation, it is necessary first to understand the limitations inherent in the specific form of electronic representation of the numbers that form the basis for the calculation, as well as the limitations in the electronic means used to implement the calculation. For example, in an electronic computer the numeric inputs to the calculation are usually each represented as a string of bits stored in a special hardware structure known as a **register.** The result of the calculation is also represented in a register. It must be borne in mind that a register is limited in the number, ***n,*** of bits (zeroes and ones) comprising the number whose representation it contains. This number, ***n,*** is called the **width** of the register. The register width, in turn, strictly limits the number of different values of the data that can be stored in it, which cannot exceed 2^n. In addition, there are also several different ***representational schemes*** that are used to define the mapping between each different bit string that can possibly be stored in a register and the numeric value that the bit string represents. Thus, a single bit string can possibly represent one of several different numbers, depending upon which representational scheme is currently in use. Depending upon both the width of the register and the representational system, the correct results of a particular arithmetic operation may ***or may not*** lie within the range of numbers representable in the register. If they lie outside the range, the consequence will be either an **overflow** or an **underflow.** This possibility is characteristic of the world of electronic computers; it is almost entirely unknown in the world of pencil-and-paper arithmetic, because in the latter environment it is almost always possible to accommodate the representation of any desired number by the simple expedient of augmenting the width of the number as measured by how many [decimal] numerals it contains.

Computations and the Digital Computer

The term "Digital Computer" is something of a misnomer. In fact, although the digital computer can be used to store numeric data and to carry out numeric calculations or computations, yet a major part of its usefulness lies in that it can carry out a great variety of other automated tasks, including the organization and storage of data of various types, such as text, sound, photographs, maps and drawings, the searching of large databases of various types for specified content, the re-ordering of data, the control of processes and of machinery, and other functions as well, for which computation is only either a trivial portion of the total work carried out or may not enter into it at all. Thus, actual computation is only one out of a large number of functions carried out by the digital "computer". Nevertheless, particularly for certain specific tasks that constitute a very significant subset of the total utilization of the digital computer of today, a major portion of the operations that take place in the computer consists of calculations that are carried out upon numeric data. Examples of such applications include payroll processing, prediction of weather, simulations of various sorts, engineering design and manufacturing (CAD/CAM), and process control. These activities all depend upon the ability of the digital computer to represent numbers as strings of bits, and the numeric operations involved in carrying out these activities fall within the broad subject area of digital arithmetic. The digital arithmetic operations must be augmented by a variety of support operations that are also necessary in order to enable the calculations to occur accurately and speedily. The vast majority of these calculations consist of simple arithmetic, and how they are carried out is conceptually very easy for even the layman to understand; they do not constitute a challenge to the Computer Scientist. However, despite the fact that most digital arithmetic operations are **conceptually** simple, the detailed understanding of how they are carried out internally within the machine, as well as their practical limitations and the various factors that affect their accuracy, **is** a substantial challenge for the beginning student of Computer Science. Once mastered, however, this subject brings to the Computer Scientist substantial insight into how digital computers work, and especially into some of the programming techniques that must be employed to assure that the calculations performed by the computer yield answers of the requisite degree of accuracy. Therefore, this subject is extremely important to master.

Digital Representation of Numbers

The overall subject matter of the treatment of numbers in digital computers can be broken down into two principal subtopics. First, it is necessary to understand the various ways in which numbers are represented in the digital computer. Several different schemes of number representation are in use, and the computer professional must understand all of the schemes that are in common use. In addition, he/she must have a grasp of the basic underlying principles of digital number representation sufficiently thorough to enable him/her to master any additional representation schemes that might be developed in the future, as he/she might encounter them in the course of his/her work. The student should be able to demonstrate his/her understanding of each scheme of number representation by relating the internal representation of a particular number to the actual numeric value of the number represented at a sufficient level of prowess, being able to map in both directions: from the digital representation of a number to its numerical value, and from any numerical value to its digital representation.

Digital Arithmetic

In addition, for each form of number representation, there is one way, or sometimes more than one way, in which it is possible for arithmetic operations to be carried out. Consequently, there is an even greater challenge for the Computer Scientist: to understand how each kind of arithmetic operation is carried out for each of the schemes of number representation that are commonly used. This must include an understanding of what answer will be produced by the computer logic circuits in every case, as well as the potential for errors occurring in the results. Finally, the Computer Scientist must also be able through a combination of hardware and software to detect initially erroneous computational results when they occur, and to implement appropriate measures in both software and hardware to handle initially erroneous results, and to take appropriate action to assure that the final results attained under defined circumstances either will be correct, or if not then they will at least be clearly marked in the final output as being incorrect.

This tutorial covers in detail only part of the subjects of number representation and digital arithmetic. It is intended to convey a thorough understanding of both subjects at least for integer numbers and for the closely related fixed-point numbers. Floating-point numbers, however, are covered only insofar as their representation in the computer is concerned. This will provide the basis from which the student will be able, through outside reading, to expand his/her understanding of arithmetic floating-point operations starting from the understanding of floating-point number representation that is provided here.

The kinds of arithmetic operations that Computer Scientists are concerned with are addition, subtraction, multiplication, division, and exponentiation. Arithmetic operations are, in general, performed in a different way in digital computers, depending upon the manner in which the underlying numbers (operands) are represented in the computer. In some cases, the differences are relatively minute, but in others they are considerable. In particular, digital arithmetic operations come in two principal varieties: *integer* operations and *floating-point* operations. *Integer* operations are performed on integer numbers, or on numbers stored in a variant of integer number representation known as *fixed-point* representation. *Floating-point* operations are performed upon numbers stored in the computer in floating-point representation. Floating-point operations are considerably different from integer operations, and are more of a challenge to the student. This tutorial covers both integer and floating-point number representations, but only integer arithmetic operations.

In this tutorial, we shall start out by surveying some of the mathematical concepts of the representation of numbers, and shall then proceed to deal with various alternative schemes for the representation of numbers that are in use both in human societies and within digital computers. Next, we shall focus in on the representation of integer numbers and of fractional numbers in digital computers, including a survey of the principal forms of integer digital number representation. In this survey, we shall cover some of the finer points and details and variants of basic integer representation. The representation of integer numbers will also be shown to be a special case of fixed-point number representation. We shall also consider the performance of arithmetic operations in digital computers upon numbers represented as integers or other fixed-point numbers (binary arithmetic). Finally, as was already mentioned, for floating-point numbers we shall cover only their representation, and not the detailed performance upon them of arithmetic operations.

Learning Objectives:

By the end of this tutorial, the student should be able to:

1. understand the mathematical concepts of *Integer Number* of *Rational Number,* and of *Irrational Number;*

2. be thoroughly familiar with the various concepts underlying the positional representation of numbers;

3. convert a rational number of arbitrary specified *base* or *radix* to its decimal equivalent;

4. convert any decimal number to a number of equivalent value in any *base* or *radix* other than ten;

5. freely inter-convert binary, octal, and hexadecimal numbers;

6. accurately interpret and determine the value of a string of bits as a *Non-Explicitly Signed ("Unsigned") Digital Number,* as a *Signed-Magnitude Number,* as a *Ones'-Complement Number,* as a *Two's-Complement Number,* and as an *Excess-**N** Number;*

7. accurately predict the results of simple digital arithmetic operations (addition and subtraction) carried out in the Arithmetic-Logic Unit (ALU) of a digital computer in accordance with the rules of *Non-Explicitly Signed-Number ("Unsigned-Number") Arithmetic,* of *Signed-Magnitude Arithmetic,* of *Ones'-Complement Arithmetic,* of *Two's-Complement Arithmetic,* and of *Saturation Arithmetic;* and

8. convert between a specified value of a Rational Number and its representation in a digital computer as a *Floating-Point* number, given a stylized definition of the particular Floating-Point representation scheme in use in the computer where the number is to be represented.

Mathematical Concepts of Numbers

In mathematics, there are four principal types of numbers. The first two of these are *Integer Numbers* and *Rational Numbers.* Both are very important for the Computer Scientist to understand, and therefore we shall cover these two kinds of numbers in relative depth. Two other kinds of numbers, *Real Numbers* and *Complex Numbers,* although very important from the mathematical standpoint, nevertheless do not represent a special challenge for the Computer Scientist, and therefore we shall briefly define these two kinds of numbers but shall not dwell on them.

Integer Numbers

Integer numbers are sometimes referred to as whole numbers. These are the counting numbers, such as:

```
1          2          3          4          5          etc.
```

There are two concepts that are incorporated into all the representations of numbers that are in common use and that are well known among the populace at large, even to students at the elementary school level. Yet, these concepts are both only relatively recent in origin. The concept of zero originated about 1400 years ago in India, and a still more radical and much more recent innovation is the concept of negative numbers. The use of negative numbers originated as recently as the late 18th to early 19th century. A sampling of integer numbers as we know them today might therefore include negative as well as positive integers, in addition to zero, for example:

```
-3,294,852,317   -79   -2   0   +3   +24   +87,346,129
```

The performance of arithmetic operations upon integer decimal numbers is so well understood by most laymen, as well as Computer Science students, that it need not be reviewed here.

Rational Numbers

A *Rational Number* is one whose value can be expressed with absolute precision as the ratio of two integer numbers. The two integers whose ratio defines the value of the rational number are known as the *numerator* and the *denominator,* and they are usually separated from each other by means of a virgule (/), as, for example, `248/613`. Most fractional numbers that we encounter in the course of daily life are *Rational Numbers.* These include prices for supermarket items in dollars and cents. For example, if a can of salmon is priced at $4.99, then the price is really:

`499` cents/`100` cents per dollar `= $4.99`

The description of the number as being equivalent to the *ratio* of `499/100` emphasizes the *rational* aspect of the number. Other examples of *Rational Numbers* are:

`7,924/3,197      3,429/12      1/789,436`

Rational Numbers are very frequently encountered in modern life, typically as decimally expressed fractions, such as currency, or the individual weights of supermarket items. Because of the ubiquity of rational numbers, being able to represent them is a very important design consideration for digital computers. Note that all *integers* are also *rational numbers,* under the special circumstance where the denominator is equal to one. Obviously, there are many more *rational numbers* than there are *integers.*

Real Numbers

The concept of *Real Numbers* includes many numbers that are <u>not</u> *rational* or *irrational,* that is, they can <u>not</u> be represented precisely as a ratio of two integers. One example of an irrational *real number* is the fundamental mathematical constant π (pi), which is the ratio of circumference to diameter of a circle. The value of this number is typically quoted as being `3.14159,` although in fact `3.14159` is actually a rational number and is therefore only an approximate representation of the true value of π. For engineering or architectural or scientific purposes, the true value of π can be calculated to any desired degree of precision, with the specific requirement for precision being dependent upon the specific need for which π is to be used. Mathematically, however, no rational number, even though its computer representation may go out to millions of decimal places, can ever express the value of π with absolute precision; it is always possible to improve on the precision by adding more decimal places. Another example of a real but irrational number is Euler's constant, *e,* which is the basis for the so-called "natural" logarithms as well as a constant of widespread use in mathematics, physics, and engineering. Other irrational numbers are the square root of 2 and the square root of 3. An irrational real number is generally represented in a digital computer by approximating its value and expressing the approximate value as a rational number. In reality, however, the rational numbers constitute a proper subset of the real numbers; that is, every rational number is also a real number, although not all real numbers are also rational.

Complex Numbers

Complex Numbers are defined as having the form **a + b*i*** where **a** and **b** are both real numbers and ***i*** = √(-1), or the square root of negative one. The representation of complex numbers in a digital computer is not a special problem. They are handled by means of separate representations of the two real coefficients, **a** and **b.** Arithmetic operations performed upon complex numbers are accomplished in accordance with the well-known mathematical rules governing such numbers**.**

Positional Number Notation

Numbers today are almost universally written in a form of notation known as *positional number representation.* In almost all human societies in the modern era, a particular variant of positional number representation is used that is known as *decimal* number representation. The concept of positional number representation is conveyed most easily, therefore, through illustration with a decimal number. Consider the decimal number `603,550`[a]. This integer number contains two zeroes and two fives. Although all zeroes are equal to each other, yet the two zeroes which are only part of this number are not equivalent to each other. Also, the two fives in this number are not equivalent to each other. The leftmost zero indicates that the number contains no ten-thousands, while the rightmost zero indicates that the number contains no units. Likewise, the left-hand five indicates a value of five hundreds, while the right-hand five indicates a value of five tens, or fifty. We can generalize by stating that it is not only the value of a particular numeral within the number, but also the *position* of the numeral that together determine the significance of that numeral and its contribution to representing the value of the overall number.

The word *decimal* is a derivative of *decem,* which is the Latin word for ten. A number represented in positional representation is composed, in general, of some collection of numerals. In a decimal number there are ten such numerals that may be used, ranging in value from zero through nine. The significance of each numeral within the number is directly related to how many numerals are present between it and the rightmost extremity of the number, that is, to its *position* in the number. Hence, the term *positional number representation*[b]. Taking our example, the decimal number `603,550,` its value is understood to be the sum of:

0 units
5 tens
5 hundreds
3 thousands
0 ten-thousands and
6 hundred-thousands

Decimal numbers are positional numbers that have a *base* or *radix* of ten. The selection of ten as the base or radix for decimal numbers has two consequences: first, it is the *radix* that imposes the requirement that there be available exactly ten distinct numerals in order to represent all

[a] This number was selected to illustrate the major features of positional number notation, but it is not a randomly-selected number. It appears prominently in the Bible. Would you happen to remember, or if you are not sure then can you guess, where in the Bible this number appears and what it represents?

[b] Can you think of a form of integer number notation where the significance of each numeral is **not** strictly dependent upon how many positions the numeral is displaced from the rightmost extremity of the number? Is the notation system known as "Roman numerals" a positional number system? Why is it, or why is it not, positional? Can you come up with a single adjective, analogously to *positional,* that adequately describes this form of notation?

possible values for each position in the number, and hence to enable us to represent all possible integer numbers in decimal notation. The decimal digits are:

0 1 2 3 4 5 6 7 8 9

Through the use of these ten numerals, absolutely any integer can be represented in decimal notation. The second consequence of decimal numbers having a *radix* or *base* of ten is that the successive numerals starting from the rightmost extremity of the integer have *place values* that are successive powers of ten. Thus, the rightmost numeral has the place value of units (= 10^0), the second numeral from the right has the place value of tens (= 10^1), the third numeral from the right has the place value of hundreds (= 10^2), the fourth has the place value of thousands (= 10^3), etc.

Generalized Positional Integer Notation

(i) Radices of Ten or Less

Once the concept of positional number notation is clearly grasped, there is very little limitation on the range of possible *radices* or *bases*. In general, the base can be any integer greater than one[c]. How we can write numbers in any radix can be grasped readily for radices of ten or less. Thus, a radix 2 number would certainly be possible, and would consist entirely of 0's and 1's. Note that just as radix ten numbers bear the special name of *decimal,* so too do radix 2 numbers bear the special name of *binary.* Similarly to the radix 2 numbers that are composed entirely of 0's and 1's, a radix 3 number would be composed entirely of 0's, 1's, and 2's. Radix 3 number representation is known as *trinary*. Trinary representation is very rarely used[d]. A radix 4 number would consist of 0's, 1's, 2's, and 3's, and is known as *quaternary*, and likewise, we could have radix 5 (*quinary*), radix 6 (*hexary*), radix 7 (*septary*), radix 8 (*octal*), and radix 9 (*nonal*) numbers, in addition to the decimal radix 10.

Consider our example number of 603,550 (decimal), if we were to express it as a base 7 number of exactly the same value. In base 7 notation, this number would be written as $5,062,423_7$[e]. The equality of value between the decimal and base-7 numbers is typically shown thusly:

[c] Why is one not a permissible value for the radix of a number?

[d] For an example of its use, however, see http://www.trinary.cc/ , and click on "Tutorials".

[e] Is this correct? You should be able to check it out by working out the place values of each numeral in the base-7 representation of the number, and multiplying by the numeral representing the value for that position. Thus, 5,062,423 in radix 7 is equal to the sum of:

3×7^0
2×7^1
4×7^2

$$603,550 \quad = \quad 5,062,423_7.$$

The subscript 7 indicates that the number to the right of the equals sign is to be interpreted as being of *radix 7.* Because of the ubiquity of decimal numbers in our society, decimal numbers are usually written without any subscript, and there is therefore a corresponding assumption that any number written without subscript is a decimal number. Therefore, the number to the left of the equals sign is written without a subscript. Nevertheless, it is also correct, if perhaps a bit pedantic, to write:

$$603,550_{10} \quad = \quad 5,062,423_7$$

and this notation has the advantage of being absolutely unambiguous. Note that because of the convention that numbers written without subscripts are by default decimal, consequently 10111100 and 10111100_2 are two numbers of radically different value, because the first is a decimal number, while the second is binary. In fact, that particular binary number has a decimal value of only 188_{10}[f].

(ii) Radices Greater than Ten

The problem with numbers having radices higher than ten is that the numerals that everyone is used to in our society extend only through nine, in consequence of the near-ubiquity of decimal numbers over almost all of the last few thousand years. With the advent of digital computers, certain other radices have come into use principally within the field of Computer Science: *binary* (radix 2), *octal* (radix 8), and *hexadecimal* (radix 16). For binary and octal numbers, the ten decimal numerals are more than enough. However, of hexadecimal numbers an additional six numerals are needed, to represent the values ten, eleven, twelve, thirteen, fourteen, and fifteen, each as a single numeral. The convention is to use the letters A B C D E and F to serve as the needed numerals. This scheme obviously would also work for each of the radices eleven through fifteen, with F being necessary only for radix 16, as the numeral representing a value of 15. E can serve in both radices 15 and 16 as the numeral representing a value of 14, D can serve as the numeral representing a value of 13, and therefore it is needed only in radices 14, 15, and 16, C can serve as the numeral representing a value of 12, and therefore it is needed in radices 13 through 16, B can serve as the numeral representing 11, and therefore it is needed in radices 12 through 16, and finally A can serve as the value representing

2×7^3
6×7^4
0×7^5
5×7^6

Is this base-7 number equal or not equal to the decimal number 603,550?

[f] The fastidious reader will check to ascertain whether this is correct.

10, and is therefore needed in all of the radices 11 through 16. Obviously, this scheme can readily be extended as far as radix 36, if necessary, by the use of the remaining alphabetic characters `G` through `Z`. Radices as large as 36 are rarely, if ever, used. Higher values of radix are even rarer, and additional symbols would have to be defined to represent the numerals needed for such radices.

How Many Different Numbers Can a Particular Notation Represent?

It is important to be able to calculate how many different numbers can be represented using some particular defined positional notation. There are two features of any positional notation that determine the answer. These are: (1) the value of the radix, ***r,*** and (2) the *width* of the representation, that is, number of numerals, ***n,*** that are available for use in representing the desired number. Remember that ***r*** must be greater than 1. A single numeral can represent ***r*** different numbers, because it can have any of ***r*** different values. These are: 0, 1, . . . [***r*** -1]. For a number of width two, that is, the representation is limited to two numerals, the numeral on the left can have any of ***r*** different values, and for each of these possible values the numeral on the right can also have any or ***r*** different values. Therefore, the ordered pair of numerals can have r^2 different sets of values. If we extend the number of numerals allowed to three, then r^3 different values are possible. By extension, for ***n*** numerals (i.e., width ***n***), it is possible to represent r^n different numbers. If simple integers are being represented, then the range of numbers extends from 0 to [r^n - 1]. The following table gives examples of representation of numbers in a variety of radices, and for a variety of widths as well:

TABLE 1: Quantitative Considerations in Positional Number Representation

Radix	**Width of the Representation (# of Numerals)**	**How many numbers can be represented?**	**Range**
r	***n***	r^n	$0 \leq (r^n - 1)$
2	12	2^{12} = 4,096	$0 \leq$ 4,095
7	9	7^9 = 40,353,607	$0 \leq$ 40,353,606
10	7	10^7 = 10,000 000	$0 \leq$ 9,999,999
16	8	16^8 = 4,294,967,296	$0 \leq$ 4,294 967,295

The notion that the representation of a number is of predetermined width may be a brand-new concept for many readers. In traditional arithmetic, we are not concerned with how many numerals we use to represent a number. When using paper, we can usually expand the width of our numbers to accommodate numbers of any desired size. For example, in the year 1789, President George Washington's first national budget had a value of around $7,000,000.00, requiring only seven digits to represent the integer, or dollar part of the number, plus an additional two digits for the cents. In contrast, President Bush's proposed budget for Fiscal Year

2005 anticipates expenditures of approximately \$2,263,000,000,000.00[g], which requires thirteen decimal digits to represent the integer part of the number, plus the same two digits for the cents that were needed in 1789. Expansion of the total width of the number from eleven digits to fifteen is no problem; there is plenty of room on the paper. Who knows how many decimal digits might be necessary in the next 215 years? Whatever may be the correct answer to that question, when dealing with paper it should be possible to accommodate whatever width might be necessary.

When we are representing numbers inside a digital computer, however, we must accommodate every digit required to represent the number by providing appropriate hardware facilities inside the computer. This translates into electronic circuitry needed to represent the current value of the number, as well as additional circuitry to carry out any arithmetic operations that might have to be performed. We must also provide appropriate hardware facilities for long-term storage of the results of any calculations that we may perform. By working electronically, we gain the advantage of lightning speed with which we can carry out our operations, but we lose the easy flexibility that we have with paper and pencil, of being able to adjust the widths of our numbers on the fly.

In general, we represent an integer $\boldsymbol{N}$ as a number of radix $\boldsymbol{r}$ in the form of an ordered list of numerals [$\boldsymbol{a_{n-1}}$, $\boldsymbol{a_{n-2}}$, . . . $\boldsymbol{a_0}$], where:

$$N = \sum_{i=0}^{n-1} a_i * r^i$$

Radices in Use in Human Societies

The most common radix in use in human societies is decimal. This is a consequence of the anatomical circumstance that the normal number of fingers possessed by most people is ten[h]. Nevertheless, there are several other radices that have also been used to some extent in certain societies over the ages. One such system is the *quinary* system (base 5), in use even today by merchants in the state of Maharashtra in western India. Another is the duodecimal system (base 12), which was used by the Assyrians, Babylonians, and Sumerians. This system is still in use in parts of China, and we have a vestige of it in our division of the day and of the night each into 12

[g] http://www.whitehouse.gov/omb/budget/fy2004/pdf/budget/tables.pdf

[h] There is an hereditary abnormality in which some people are born with not just five but six fingers on each hand or six toes on each foot. These conditions bear the name of *polydactyly* (which means “many fingers” in Greek). If six fingers and six toes had been the human norm instead of five, then society would probably have settled on a base 12 number system. A base 12 number system would have been much more useful than base 10. Our decimal standard is relatively difficult to use, because the radix of 10 is divisible by only 2 and 5. Radix 12 has a distinct advantage in that it is divisible by 2, 3, 4, and 6.

hours. The *vigesimal* system (base 20) was used by the Ainu people in northern Japan, and also by the Aztecs, Celts, Greenland Eskimos, and Mayans. And finally, there is the *sexagesimal* system (base 60), which was used by the Babylonians and Sumerians. This system is the basis for our practice of dividing the hour into sixty minutes, and the minute into sixty seconds.

For practical reasons, computers make use of the binary number system. For ease by people in notating and understanding the content of binary numbers, as well as for interpreting the results of arithmetic operations carried out in binary, it is convenient to make use of either the octal (radix 8) or hexadecimal (radix 16) number systems, both of which are readily interconvertible with binary. The particulars of such interconversion will be covered later.

Conversion of a Number from One Radix to Another

The best way to develop facility in the understanding positional number notation in a variety of radices is to be able to convert numbers from any starting radix $\mathbf{r_a}$ to any destination radix $\mathbf{r_b}$**.** Such conversion usually requires multiple operations of both multiplication and division. The trick to success in performing such conversions comes from arranging that all the multiplications and divisions will be done in decimal, since that is the number scheme with which most people know the arithmetic rules very well. We shall first consider the conversion of integers from other radices to decimal, and then from decimal to other radices. Next, we shall consider the conversion of (non-integral) rational numbers from other radices to decimal and from decimal to other radices. Finally, we shall consider the most productive strategy for conversion from any starting radix $\mathbf{r_a}$ to any destination radix $\mathbf{r_b}$**.**

Conversion of Integers from Other Radices to Decimal

Conversion of an integer number from any other radix to decimal is a straightforward operation. It is accomplished by determining first the place value of every numeral, starting from the rightmost position (always the units digit) and then proceeding stepwise leftwards, progressively multiplying the place value of the previous position by the radix, until the place values of all the numerals are determined. This must be done once for a particular radix of origin; thereafter, the place values so calculated can be re-used for converting many different numbers from that radix to decimal. The second step, after all the place values have been determined for the starting radix, is to multiply each numeral of the particular number being converted by its place value. This gives the decimal value contributed by that particular numeral. Finally, the sum of the decimal equivalents of all the numerals is calculated, thus giving the total decimal equivalent of the original number.

For example, consider the number $2,122,220_3$. The place values for radix 3 work out to:

TABLE 2: Place Values for Radix-3 Numbers

Place	Value
1	$3^0 = 1$
2	$3^1 = 3$
3	$3^2 = 9$
4	$3^3 = 27$
5	$3^4 = 81$
6	$3^5 = 243$
7	$3^6 = 729$

The value of the number 2,122,220$_3$ in decimal thus works out to:

TABLE 3: Decimal Value of a Radix-3 Number

Place	Numeral	Value of the Numeral
1	0	0 x 1 = 0
2	2	2 x 3 = 6
3	2	2 x 9 = 18
4	2	2 x 27 = 54
5	2	2 x 81 = 162
6	1	1 x 243 = 243
7	2	2 x 729 = 1458
Sum in Decimal:		1,941

Next consider the number 2,122,220$_4$ (same numerals as the previous number, but a different radix). The place values for radix 4 work out to:

TABLE 4: Place Values for Radix-4 Integer Numerals

Place	Value
1	$4^0 = 1$
2	$4^1 = 4$
3	$4^2 = 16$
4	$4^3 = 64$
5	$4^4 = 256$
6	$4^5 = 1,024$
7	$4^6 = 4,096$

The value of this number in decimal thus works out to:

TABLE 5: Decimal Value of a Radix-4 Number

Place	Numeral	Value of the Numeral
1	0	0 x 1 = 0
2	2	2 x 4 = 8
3	2	2 x 16 = 32
4	2	2 x 64 = 128
5	2	2 x 256 = 512
6	1	1 x 1,024 = 1,024
7	2	2 x 4,096 = 8,192
Sum in Decimal:		9,896

Overall, please note that the conversion of an integer from an arbitrary starting radix to decimal is a straightforward operation that takes place using multiplications and additions in accordance with the rules of decimal arithmetic.

Conversion of Integers from Decimal to Other Radices:

To convert an integer number from decimal to some other radix, there is an algorithm that is simple to execute. Simply divide the number over and over by the destination radix. Each successive division will produce a result expressed as a quotient and a remainder. The remainder obtained from the first division becomes the units-place numeral for the number in the new radix. Take the quotient from the first division, and divide it once more by the new radix. The remainder from the second division becomes the second-place numeral in the new radix, and the quotient is divided once more by the value of the new radix. This process continues until the quotient of a division becomes zero. Any remainder still left over at this point becomes the leftmost numeral of the number written in the new radix.

Consider the number 1941_{10} converted to radix 7. The sequence of operations is:

TABLE 6: Conversion of a Decimal Number to Radix-7

Step #	Operation	Result
1	1941/7	Quotient = 277; Remainder = 2
2	277/7	Quotient = 39; Remainder = 4
3	39/7	Quotient = 5; Remainder = 4
4	5/7	Quotient = 0; Remainder = 5
5	Stop here: no quotient remaining	
Value of Number in Base 7:		$5,442_7 = 1941_{10}$

Check if this answer is correct by converting $5,442_7$ back to decimal, using the method shown earlier.

Interconversion of Integers between Any Pair of Radices:

It is important to be able to convert a number from any arbitrary radix to any other radix. This is generally difficult to do, since the arithmetic has to be carried out in the starting radix, and the rules for division in the general case of *radix* ***r*** are different from the rules of decimal arithmetic that we are used to from daily living. The easiest way to accomplish this goal, therefore, is to convert first to decimal and then to the target radix. Conversion to decimal is straightforward, as was shown above, and requires exclusively operations that are performed in decimal. Likewise, conversion from decimal to any other radix can take place using decimal arithmetic (successive division by the target radix), and are therefore also easy to carry out. Therefore, in general to convert from *radix* $\boldsymbol{r_1}$ to *radix* $\boldsymbol{r_2}$, just convert first from *radix* $\boldsymbol{r_1}$ to decimal, and then from decimal to *radix* $\boldsymbol{r_2}$**.**

Conversion of Fractional Numbers from Other Radices to Decimal

In general, conversion of a fractional number from some other radix to decimal is just a straightforward extension of the conversion algorithm for integers. That is, first the decimal place values for the different numeral positions for the source radix are calculated, which once done can serve for the conversion of as many numbers as needed from the same source radix to decimal. This calculation is accomplished starting from the **radix point** (which for decimal numbers is called the **decimal point)** and proceeding outwards. Then the separate contribution to the number of each numeral extending rightwards from the radix point must be determined by multiplying the place value expressed in decimal by that numeral. Finally, the sum of the contributions of all numerals of the original number must be taken.

For example, consider the conversion to decimal of the source number 0.2122220_3. The place values counting rightwards from the **radix point** work out for radix 3 to:

TABLE 7: Place Values of Radix-3 Fractional Numerals

Place	Value
-1	3^{-1} = 0.333333333+
-2	3^{-2} = 0.111111111+
-3	3^{-3} = 0.037037037+
-4	3^{-4} = 0.012345679+
-5	3^{-5} = 0.004115226+
-6	3^{-6} = 0.001371742+
-7	3^{-7} = 0.000457237+

The value of the number `0.2,122,220`$_3$ in decimal thus works out to:

TABLE 8: Decimal Value of a Radix-3 Fractional Number		
Place	**Numeral**	**Value of the Numeral**
-1	2	2 x 0.333333333+ = 0.666666666+
-2	1	1 x 0.111111111+ = 0.111111111+
-3	2	2 x 0.037037037+ = 0.074074074+
-4	2	2 x 0.012345679+ = 0.024691358+
-5	2	2 x 0.004115226+ = 0.008230452+
-6	2	2 x 0.001371742+ = 0.002743484+
-7	0	0 x 0.000457237+ = 0.000000000
Sum in Decimal:		0.887517145+
The '+' signs indicate that there are more digits, but that the calculated number is purposely being truncated at an arbitrarily chosen level of precision.		

Please note that for the general case of a rational number, which will be expressed in the form of numerals on both sides of the radix point, it is necessary to calculate the place values of the various numerals going in both directions from the radix point. Several worked examples are given in the "Review Questions on Digital Number Representation".

Conversion of Fractional Numbers from Decimal to Other Radices

To convert a fractional decimal number to another radix, instead of performing division it is necessary instead to multiply the fractional number successively by the radix. Each multiplication, in general, results in a product that has both an integer part and a fractional part. The integer part resulting from the **first** multiplication becomes the first numeral to the right of the radix point for the number in the new radix. Only the fractional part of the first product is multiplied again by the value of the destination radix to give the second product. Again, the integer portion of this product becomes the next numeral to the right of the radix point for the number represented in the new radix. The fractional part of the product is stripped off and multiplied once more by the value of the destination radix to give the next product. This process continues until the desired level of precision is reached. For example, to convert the number `0.887517145`$_{10}$ to base 3, the successive multiplications yield:

TABLE 9: Conversion of a Fractional Decimal Number to Radix-3				
Multiplicand	**Multiplier**	**Product**	**Fractional Part**	**Integer Part**
0.887517145	3	2.662551435	.662551435	2
0.662551435	3	1.987654305	.987654305	1
0.987654305	3	2.962962915	.962962915	2
0.962962915	3	2.888888745	.888888745	2
0.888888745	3	2.666666235	.666666235	2
0.666666235	3	1.999998705	.999998705	1
0.999998705	3	2.999996115	.999996115	2
0.999996115	3	2.999988345	.999988345	2
0.999988345	3	2.999965035	.999965035	2
Radix 3 Number: 0.212221222_3				

Notice that we had started out up above with the number 0.2122220_3. which we first converted up above to the decimal number 0.887517145_{10} and then just now converted back to radix 3. Can you come up with an explanation of why we ended up with a number that is slightly different from that with which we had started out?

Interconversion of Fractional Numbers between Any Pair of Radices:

It is important for fractional numbers, too, to be able to convert a number from any arbitrary source radix to any other destination radix. As for the integers, so, too for the fractions this is generally difficult to do, since the arithmetic has to be carried out in the starting radix, and the rules for both multiplication and addition in the general case of *radix **r*** are different from the rules for multiplication and addition in the decimal arithmetic that we are used to from daily living. The easiest way to accomplish our goal, therefore, is to convert first to decimal and then to the target radix. Conversion to decimal is straightforward, as was shown above, and requires exclusively operations that are performed in decimal. Likewise, conversion from decimal to any other radix can take place using decimal arithmetic (successive multiplication by the target radix), and are therefore also easy to carry out. Therefore, in general to convert from *radix* r_1 to *radix* r_2, just convert first from *radix* r_1 to decimal, and then from decimal to *radix* r_2.

Interconversion of Mixed Numbers between Any Pair of Radices

It is necessary to be able to convert the general case of rational numbers from any starting radix to any destination radix. To accomplish this, simply divide the number at the radix point into its two principal components: the integer part and the fractional part. Follow the procedure already shown for the conversion of integer numbers on the integer portion of the number, and the procedure for the conversion of fractional numbers on the fractional portion of the number, and then reassemble the number in the destination radix from its two components. For several worked examples, please see the "Review Questions on Digital Number Representation".

Binary Numbers

In digital computers, numbers are universally represented in some variant of binary form, that is, as a sequence of 0's and 1's. Each 0 or 1 is referred to as a *binary digit* or *bit.* For a scheme in which **n** bits are used to represent each number, each bit can have a value of either 0 or 1, and therefore a total of $2^{\mathbf{n}}$ different numbers can be represented. There are several different forms of binary number representation. The various forms differ from each other in two ways: in the *range* of numbers represented, and in the scheme by which a given bit sequence is mapped to a specific number within the range. There is a very special aspect of representation of number within a digital computer that needs to be borne in mind. When we are representing numbers with paper-and-pencil notation, if we run out of range within a given number of numeral positions, it is usually a fairly trivial matter to add as many numerals as may be required for the size of the number that we must represent. In digital computers, however, we must normally face the circumstance that we are limited by the computer hardware in terms of the number of bits that we can allocate to the representation of a number. If the number that we must represent is out of range, then we might have to do some fancy footwork in software to provide the functional equivalent of use of a larger number of hardware bits than are available.

We shall mainly consider various binary schemes for the representation of integers. The most important of these are *Non-Explicitly-Signed ("Unsigned") Representation, Signed-Magnitude Representation, Ones'-Complement Representation, Two's-Complement Representation,* and *Excess-**N** Representation.* Of these, the simplest to comprehend is *Unsigned Number representation,* and that is where we shall begin.

Unsigned Number Representation

In the *Unsigned Number* form of binary number representation, all numbers are treated as non-negative integers (that is, the numbers represented are all either positive integers or zero). This is equivalent to the general scheme for representation of integer numbers described above, and is the absolutely simplest scheme of binary number representation.

For an unsigned number composed of **n** bits, a total of $2^{\mathbf{n}}$ different numbers can be represented, and the range of numbers represented in this way extends from 0 up to a maximum value of $[2^{\mathbf{n}} -1]$. Sometimes, it is necessary to look at this issue from the opposite perspective: If we know that we must represent some range of numbers from 0 up to ***N,*** then how many bits, ***n,*** are required to represent them? The answer is $\boldsymbol{n} = \lceil(\log_2 \boldsymbol{N})\rceil$, where the pair of symbols $\lceil \; \rceil$ denotes the **ceiling function.** This is the smallest integer *less than or equal to* the value of the term enclosed by the two symbols. Thus, for example, if we are required to represent 487 different numbers, $\lceil(\log_2 487)\rceil = 9$, and therefore nine bits are required. The reason for this is that eight bits would be too few, being able to represent only 256 different numbers. While nine bits can represent as many as 512 different numbers, which is more than the 487 necessary, nevertheless since we can only have an integral number of bits, the smallest integer greater than eight is necessary, and this comes out, of course, to nine.

In all of the binary integer representation schemes other than unsigned numbers, not only positive but also negative numbers are represented. These schemes all differ from each other in terms of how are the negative numbers represented as well as in the exact range numbers represented.

Signed-Magnitude Representation

In *Signed-Magnitude* representation, the leftmost bit is reserved as a sign bit, and the remaining bits signify the magnitude of the number. For the sign bit, a 0 represents positive sign and a 1 represents negative sign. Consider as an example the following two numbers:

$$\mathbf{a} = 01011011_2$$

$$\mathbf{b} = 11011011_2$$

The seven bits on the right side of both numbers, i.e., the magnitude bits, are identical. Only the leftmost bit (the zero bit) is different. Examining the magnitude bits of either number, the units bit and the 2's bit are both 1, the 4's bit is a 0, the 8's bit and the 16's bit are both 1, the 32's bit is a 0, and the 64's bit is a 1. Hence the magnitude of both numbers is: 1 + 2 + 8 + 16 + 64 = 91. Because of the difference in the sign bits, **a** = +91 and **b** = -91.

Overall, in *Signed-Magnitude* representation, those numbers ranging from zero to [$2^{\mathbf{n}\,-1}$ - 1] are represented identically to the way they are represented in *Unsigned-Number* representation. The remaining bit sequences, which in *Unsigned-Number* representation are used for numbers whose values range from $2^{\mathbf{n}\,-\,1}$ to $2^{\mathbf{n}}$, are co-opted and are used instead to represent negative numbers. Note that each non-negative number represented has a corresponding negative number whose representation is identical in all bit positions except the sign bit. This means that *Signed-Magnitude* has two representations for zero. In 8-bit *Signed-Magnitude* these are `00000000` and `10000000`. These are referred to as "positive zero" and "negative zero" respectively.

Ones'-Complement Representation

In *Ones'-Complement* representation, to represent the negative of a number one subtracts the positive value of the number from a special number consisting of all 1's. Hence, the term

Ones'-Complement (with the apostrophe **after** the **s** in "Ones"). Let us examine how the negative of 91 is represented in *Ones'-Complement:*

All 1's (binary representation of $2^n - 1 = 255_{10}$):	`11111111`
Binary representation of $+91_{10}$:	`01011011`
Ones'-Complement representation of -91_{10}:	`10100100`

Notice that in subtracting the binary representation of $+91_{10}$ from the number consisting of all 1's (which for eight bits represents 255_{10} in *Unsigned-Number* representation), in every bit position where the representation of $+91_{10}$ has a 0, the representation of -91_{10} has a 1. Likewise, in every bit position where the representation of $+91_{10}$ has a 1, the representation of -91_{10} has a 0. Thus, the *Ones'-Complement* representation of a negative number consists of the bit-wise inversion (hence the term "Complement") of the representation of the positive number of equivalent magnitude. This scheme is radically different from the *Signed-Magnitude* representation, in which all bits except for the sign bit are identical between the representation of any pair of positive and negative numbers whose magnitudes are equal.

So far, we have described how to obtain the *Ones'-Complement* representation of a given number. Now, let us consider the opposite problem: Given a binary representation of a number that we know to be in *Ones'-Complement* form, how do we determine the value of the number represented? The procedure is first to examine the sign bit. If this is a 0 (signifying that a positive number is represented), then merely compute the magnitude of the positive number in the usual way by adding up the place values of all bit positions having 1's. If the sign bit is a 1, that indicates that a negative number is represented. In that case, take the *Ones'-Complement* of the negative number to obtain the representation of its magnitude, and then determine the magnitude of this number as before. There are several worked examples of conversion between decimal and *Ones'-Complement* representation in "Review Questions on Digital Number Representation".

Overall, in *Ones'-Complement* representation, those numbers ranging from zero to $[2^{n-1} - 1]$ are represented identically to the way they are represented in *Unsigned-Number* and in *signed-Magnitude* representations. Numbers whose magnitudes lie between 2^{n-1} and $2^n - 1$ are not represented at all. The bit sequences that are used in *Unsigned-Number* representation for these numbers are co-opted in *Ones'-Complement,* and are used instead to represent negative numbers. *Ones'-Complement,* as well as *Signed-Magnitude,* has two representations of zero, a positive zero and a negative zero. The positive zeroes are identically represented in both schemes by a bit string consisting of zeroes in all bit positions. However, the negative zero representation differs between the two systems. In *Signed-Magnitude* it consists of `10000000`, but in *Ones'-Complement* it consists of `11111111`[i].

[i] Check for yourself, based upon the principles for representation of negative numbers in *Ones'-Complement* that have been explained, to see why this is so.

Two's-Complement Representation

In *Two's-Complement* representation, the set of bit sequences that in *Unsigned-Number* representation is utilized to represent numbers in the range of $2^{\mathbf{n}\ -1}$ up to [$2^{\mathbf{n}}$ - 1] is assigned to negative numbers in a closely-related but slightly different way from how this is accomplished in *Ones'-Complement.* In *Two's-Complement* this is accomplished by subtracting the positive number representing the magnitude of the number, whose negative representation is desired, from $2^{\mathbf{n}}$ rather than from [$2^{\mathbf{n}}$ -1]. To accomplish this feat conceptually, it is necessary to add an extra bit to the subtrahend (the number from which is to be subtracted the magnitude of the number whose negative representation is desired). This is best understood from an example such as the following for 8-bit numbers:

Binary representation of $2^{\mathbf{n}}$ = 256_{10} (requires a 9th bit):	`100000000`
Binary representation of $+91_{10}$:	`01011011`
Two's-Complement representation of -91_{10}:	`10100101`

Note that the *Two's-Complement* representation of a positive number is almost but not quite identical to the *Ones'-Complement* representation of the same number. In fact, carrying out the subtraction as shown in the illustration is cumbersome and difficult to implement in digital computers. Therefore, in practice the process of "doing" *Two's-Complementation* in a digital computer is carried out in two steps: first "complementing" (that is, taking the *Ones'-Complement* of) the number and then incrementing the *Ones'-Complement*.

Overall, in *Two's-Complement* representation, those numbers ranging from zero to [$2^{\mathbf{n}\ -1}$ - 1] are represented identically to the way they are represented in *Unsigned-Number* and in *signed-Magnitude* representations. Numbers whose magnitudes lie between $2^{\mathbf{n}\ -1}$ and $2^{\mathbf{n}}$ - 1, as in the other two schemes discussed so far that accommodate the representation of negative numbers, are not represented at all. The bit sequences that are used in *Unsigned-Number* representation for these numbers are also co-opted in *Twos'-Complement,* and are used instead to represent negative numbers. *Two's-Complement,* in contrast both to *Signed-Magnitude* and to *Ones'-Complement,* has only one representation of zero, which is equal to the positive zero of the other two schemes. The bit string consisting of all 1's, which in *Ones'-Complement* represents negative zero, in *Two's-Complement* represents the number [$-2^{\mathbf{n}-1}$], which is not represented either in *Signed-Magnitude* or in *Ones'-Complement..* Thus, in 8-bit *Two's-Complement* the bit string `11111111` represents -128_{10}. Note that +128 has no representation at all in 8-bit *Two's-Complement*[j].

Please try hour hand at interconverting between decimal notation of a number and binary representation in *Two's-Complement* form. There are several worked problems appearing in "Review Questions on Digital Number Representation".

[j] Can you explain why *Two's-Complement* representation has this asymmetry in representation of positive and negative numbers, while both *Signed-Magnitude* and *Ones'-Complement* are completely symmetrical?

Excess-N Representation

The final scheme of binary number representation that will be covered here is called *Excess-**N*** representation. This scheme is important in Computer Science because it is often used for the representation of exponents within Floating-Point numbers. In an *Excess-**N*** representation, a decimal number is represented in binary notation. It is necessary to specify a value for ***N,*** but this is usually equal to $2^{\mathbf{n}-1}$ for representation in an **n**-bit field. The value of **N** must be added to the decimal value of the number to be represented, and then the *Unsigned-Number* representation of the sum is what is stored. This is much easier to understand from illustration than from explanation. Consider an 8-bit field used to store numbers in *Excess-128* notation. To represent the number -91_{10}:

Number to be represented:	-91_{10}
Add the value of ***N*** :	-91 + 128 = +37
Unsigned-Magnitude 8-bit representation of [Number + ***N***]:	`00100101`

To convert in the opposite direction, first calculate the *Unsigned-Magnitude* value of the bit string representing the number, and then subtract the value of ***N.*** The result is the value of the number represented.

Overall, in *Excess-**N*** representation, assuming that the value of ***N*** is $2^{\mathbf{n}-1}$, those numbers are represented that range from $-2^{\mathbf{n}-1}$ to $[+2^{\mathbf{n}-1} - 1]$, which is the same range as for *Two's-Complement.* However, not one of the numbers in the entire range is represented identically to the way it is represented in any of the other binary notations that are covered in this tutorial. Numbers whose magnitudes lie between $2^{\mathbf{n}-1}$ and $2^{\mathbf{n}} - 1$, as in the other three schemes discussed so far that accommodate the representation of negative numbers, are also in *Excess-**N*** not represented at all.

Summary of Binary Number Representation

Several schemes have been discussed for the representation of integers in binary notation. The following table summarizes these schemes. In the leftmost column, "Hexadecimal Value of Number", the actual *value* **(not** the *representation*) of the number is shown in hexadecimal. Hexadecimal numbers have not yet been explained. The reader is advised to ignore this column for now, but to return to this table and re-examine the leftmost column after hexadecimal numbers have been covered.

TABLE 10: Comparison of Various Representational Schemata in use for Four-Bit Numbers

Hexadecimal Value of Number	**Decimal Value of Number**	**Binary Representations**				
		Unsigned Number	**Signed-Magnitude**	**Ones'-Complement**	**Two's-Complement**	**Excess-8**
-8H	-8	*N/R*	*N/R*	*N/R*	1000	0000
-7H	-7	*N/R*	1111	1000	1001	0001
-6H	-6	*N/R*	1110	1001	1010	0010
-5H	-5	*N/R*	1101	1010	1011	0011
-4H	-4	*N/R*	1100	1011	1100	0100
-3H	-3	*N/R*	1011	1100	1101	0101
-2H	-2	*N/R*	1010	1101	1110	0110
-1H	-1	*N/R*	1001	1110	1111	0111
-0H	-0	*N/R*	1000	1111	*N/R*	*N/R*
+0H	0	0000	0000	0000	0000	1000
+1H	1	0001	0001	0001	0001	1001
+2H	2	0010	0010	0010	0010	1010
+3H	3	0011	0011	0011	0011	1011
+4H	4	0100	0100	0100	0100	1100
+5H	5	0101	0101	0101	0101	1101
+6H	6	0110	0110	0110	0110	1110
+7H	7	0111	0111	0111	0111	1111
+8H	8	1000	*N/R*	*N/R*	*N/R*	*N/R*
+9H	9	1001	*N/R*	*N/R*	*N/R*	*N/R*
+AH	10	1010	*N/R*	*N/R*	*N/R*	*N/R*
+BH	11	1011	*N/R*	*N/R*	*N/R*	*N/R*
+CH	12	1100	*N/R*	*N/R*	*N/R*	*N/R*
+DH	13	1101	*N/R*	*N/R*	*N/R*	*N/R*
+EH	14	1110	*N/R*	*N/R*	*N/R*	*N/R*
+FH	15	1111	*N/R*	*N/R*	*N/R*	*N/R*

N/R means that the specified number is *N*ot *R*epresented in the particular representation scheme applicable to the current column.

Points to Ponder:

1. What is the number of substantive entries in each column of the table?
2. Do different columns have different numbers of entries, or are they all equal?
3. What determines the maximum possible number of substantive entries in a column?
4. Of the various number representation schemes shown, which is the best to use for the representation of integers? Explain/justify your answer.
5. Describe the relationship between the contents of the adjacent columns of binary numbers for: (a) the natural numbers; and (b) the non-positive numbers.

Please examine this table very carefully to be certain that you understand the various forms of number representation. Note that the principles that govern the various schemes of number representation apply equally to bit strings of width two or greater without upper limit. In ancient times (for Computer Science, "ancient times" means ten or more years ago), computers were manufactured by different companies with a great variety of "word sizes", that is, of the lengths of bit strings used to represent numbers inside the machine. Today the word size is universally some multiple of eight bits: either 8 or 16 or 32 or 64 or 128. The Computer Scientist needs to be thoroughly familiar with the place values for the bits of binary numbers represented in *Unsigned-Number* notation, as follows:

Table 11: Powers of Two

Bit Position #	**Power of 2**	**Place Value**	**Nominal Value**	**Approximate Value**
0	2^0	1		
1	2^1	2		
2	2^2	4		
3	2^3	8		
4	2^4	16		
5	2^5	32		
6	2^6	64		
7	2^7	128		
8	2^8	256		
9	2^9	512		
10	2^{10}	1,024	1 k	1 thousand
11	2^{11}	2,048	2 k	2 thousand
12	2^{12}	4,096	4 k	4 thousand
13	2^{13}	8,192	8 k	8 thousand
20	2^{20}	1,048,576	1 M (Meg)	1 million
24	2^{24}	16,777,216	16 M (Meg)	16 million
Table 11 (continued): Powers of Two				
30	2^{30}	1,073,741,824	1 G (Gig)	1 billion
32	2^{32}	4,294,967,296	4 G (Gig)	4 billion
36	2^{36}	68,719,476,736	64 G (Gig)	64 billion
40	2^{40}	1,099,511,627,776	1 T (Tera)	1 trillion
50	2^{50}	1,125,899,906,842,624	1 P (Peta)	1 quadrillion

To compute the place value of any bit position, remember the basic exponential identity:

$$X^{\{y + z\}} \equiv X^{y} \times X^{z}$$

For binary numbers, the identity becomes:

$$\mathbf{2^{\{y + z\}} \equiv 2^{y} \times 2^{z}}$$

What this means is that if we want to determine the place value of, for example, bit 47 (the 48th bit position), that works out to:

$$\mathbf{2^{48} = 2^{8} \times 2^{40}} = 256 \text{ Tera} = \mathbf{256 \times (1{,}024)^{4} = 281{,}474{,}976{,}710{,}656.}$$

Display and Description of the Contents of Memory Locations and Registers Using Octal and Hexadecimal Notation

As has already been noted above, either octal notation (radix 8) or hexadecimal notation (radix 16) can be used as an alternative to the traditional decimal notation. For the purpose of general number notation, both octal and hexadecimal can be used for a variety of purposes. These include the representation of positive and negative numbers, of integers, and of fractional numbers, that is, for numbers whose absolute value is greater than or equal to 0.00 and less than or equal to 1.00. They also include the representation of mixed numbers, that is, general rational numbers of unrestricted size, which includes numbers that are purely integers, numbers that are purely fractional, and numbers having both an integer part and a fractional part. For these representations, the scheme for either octal or hexadecimal numbers is just a special case of the general positional number representational scheme that was described above and that is usable for any radix. Such a representation includes either a minus sign or a plus sign to indicate the sign of the number (a positive number being assumed if neither symbol is present). It also includes a radix point (octal point or hexadecimal point) to mark the separation between the integer and fractional parts of the number, zero or more octal or hexadecimal digits situated to the left of the radix point representing the integer part of the number, and zero or more octal or hexadecimal digits situated to the right of the radix point representing the fractional part. If the number does not have a fractional part, then the radix point may be present by implication only, rather than being shown explicitly.

There is, however, a very important additional usage of either octal or hexadecimal notation which may have no direct relation at all to the value of the number ***or other datum*** represented. In this usage, a string of octal or hexadecimal digits is used merely as a means of portraying the contents of a memory location or of a register, irrespective of both the kind of data represented there and the value portrayed by those contents. In modern computers, the width of a memory word or of a register measured in bits is always an integer multiple of four. Therefore, hexadecimal notation is universally used for this purpose. For example, in a computer having 32-bit memory words, the content of one word might be represented as the hexadecimal bit string:

ABCDE000

Note that the hexadecimal representation has no sign. In this case, a sign is truly **not present,** as distinguished from an ***implicit*** sign sometimes forming part of the representation of a numeric value. Such an ***implicit sign*** is implied in the value of the number represented, even though it is not shown. Here, however, there really is no sign, because the hexadecimal digits are being used ***not*** to represent **value** but merely to indicate **content.** In fact, what this means is that the binary content of the memory word is:

Hexadecimal:	A	B	C	D	E	0	0	0
Binary:	1010	1011	1100	1101	1110	0000	0000	0000

This content might possibly represent an integer (Non-Explicitly Signed, Ones'-Complement, Two's-Complement, Signed-Magnitude, Excess-*N*, or any other representational scheme for integers). Alternatively, it might represent a fixed-point binary number of any of the same schemes already enumerated[k]. Furthermore, it might represent all or part of a floating-point

[k] Note that the location of the implied radix point for a fixed-point non-explicitly-signed number may or may not happen to coincide with the boundary between two of the hexadecimal digits used to represent the content of the memory location or register. In the example shown above, possible locations for the implied binary point are:

```
.10101011110011011110000000000000
1.0101011110011011110000000000000
10.101011110011011110000000000000
101.01011110011011110000000000000
1010.1011110011011110000000000000
10101.011110011011110000000000000
101010.11110011011110000000000000
1010101.1110011011110000000000000
10101011.110011011110000000000000
101010111.10011011110000000000000
1010101111.0011011110000000000000
10101011110.011011110000000000000
101010111100.11011110000000000000
1010101111001.1011110000000000000
10101011110011.011110000000000000
101010111100110.11110000000000000
1010101111001101.1110000000000000
10101011110011011.110000000000000
101010111100110111.10000000000000
1010101111001101111.0000000000000
10101011110011011110.000000000000
101010111100110111100.00000000000
1010101111001101111000.0000000000
10101011110011011110000.000000000
101010111100110111100000.00000000
1010101111001101111000000.0000000
10101011110011011110000000.000000
```

(continued on bottom of next page)

number (not yet discussed; discussion of floating-point numbers appears below). There are numerous additional possibilities, all having nothing to do with numbers. Thus, the string of 32 bits might represent four 8-bit ASCII characters, four EBCDIC characters, two Unicode characters, a bit vector of length 32, a machine instruction, or either part or all of some programmer-defined data structure. We are merely borrowing the hexadecimal digits in order to describe the underlying content of the memory word or of the register more economically and with reduced possibility of error than we could with a string of zeroes and ones.

Some years ago, when several computer manufacturers produced machines having 12-bit, 24-bit 36-bit or other word width or register width that was a multiple of three bits as well as of four, several manufactures, most notably Digital Computer Corporation, chose to represent the content of their memory words and registers with strings of octal digits rather than hexadecimal. This was only slightly easier for programmers and system managers to get used to than hexadecimal, since the octal digits are a proper subset of the decimal digits, whereas the hexadecimal digits are a superset of the decimal. Therefore, the use of octal strings to describe the contents of memory words and of registers is of historical importance only, but has no practical value to the modern computer scientist.

```
10101011110011011110000000.00000
101010111100110111100000000.0000
1010101111001101111000000000.000
10101011110011011110000000000.00
101010111100110111100000000000.0
1010101111001101111000000000000.
```

(continued on bottom of next page)

Binary Addition for Integer and other Fixed-Point Numbers

Addition of binary numbers is closely analogous to the addition of decimal numbers. Unlike manual addition, where an entire column of numbers might be added in the course of a single compound operation, addition of numbers in a digital computer is almost always carried out on a pair of numbers at a time. By arbitrary convention, the first number of the pair, which is written above the other number is known as the ***augend.*** The second number, written underneath, is known as the ***addend.*** In primitive computers, just as for manual addition of decimal numbers, the addition of binary integers or fixed-point numbers is also carried out digit by digit, starting from the right extremity of the two numbers and proceeding leftwards one digit at a time. A logic circuit implementing such binary addition via underlying electronics is called a ***Ripple-Carry Adder***. The fundamental operation of the ***Ripple-Carry Adder*** therefore consists of the addition of a single bit of augend and of addend. The output consists of not just one but **two** bits, one representing the sum and the other representing the carry-out from the current bit position, which is also the carry-in to the next bit position to the left. For all bits other than the rightmost one, it is therefore necessary to cope with as many as ***three*** input bits. One bit comes from the augend, one from the addend, and the third originates in the carry out from the addition of the previous bit one position to the right of the current bit. Since there are, in general, three bits of input, there are therefore eight possible situations that the electronic bit-wise adder circuit must be able to cope with. These are:

TABLE 12: Single-Bit Addition

INPUT:	**Carry-In Bit**[l]**:**	1	1	1	1	0	0	0	0
	Bit from *Augend*:	1	1	0	0	1	1	0	0
	Bit from *Addend*:	1	0	1	0	1	0	1	0
OUTPUT:	**SUM:**	1	0	0	1	0	1	1	0
	Carry-Out[m]	1	1	1	0	1	0	0	0

When addition is carried out in a digital computer, there is one possible problem that we must be prepared to recognize whenever it occurs, and to handle whenever necessary. That is the problem of ***overflow.*** This problem almost never crops up when performing addition manually, because when we add numbers manually using pencil and paper, if the sum requires more digits to represent it than are present in the augend and in the addend, we are almost always able to expand the width allotted to the sum by as many digits as necessary to accommodate the size of the number that we must represent. However, in a digital computer we are **always** faced with a limit on ***how many*** numbers can be represented in a register or memory location having a fixed number of bits, as well as ***which particular*** numbers are represented within the width of the storage location, using whatever scheme of representation is applicable at the moment. For example, the Intel 4004, which was the very first microprocessor, had registers that were only

[l] This bit originates from the carry-***out*** from addition of the bit immediately to the right of the current bit position.

[m] This bit will become the carry-***in*** to the addition of the next bit to the left of the current bit position.

four bits wide. The Intel 8088, which was used in the first IBM PC, had registers that were eight bits wide. By the time the Intel Pentium was introduced, the register width had expanded to 32 bits. Because of the fixed width of the registers in any processor, when we are using a digital computer to add two numbers, it may turn out that the correct arithmetic sum of the augend and the addend lies outside of the range of values representable within the constraints of the register where the sum must be stored. The term ***overflow*** is particularly apt; it stems from the analogy of pouring into a container more water than the container is capable of holding. The excess spills out over, i.e., it ***overflows*** the container's top edge.

TABLE 13: Examples of Overflow							
Non-Explicitly-Signed Numbers							
Carry-In Bits:	1	0	0	1	0	0	
Augend Register:	0	1	0	0	1	0	**Value of Augend = (+)18**
Addend Register:	1	1	1	0	1	0	**Value of Addend = (+)48**
Sum Register:	0	0	1	1	0	0	**Value of Sum-Register Content = (+)12** **TRUE SUM = (+)66**
Ones'-Complement Numbers							
Carry-In Bits:	1	1	1	1	1	0	
Augend Register:	0	0	1	1	0	1	**Value of Augend = +13**
Addend Register:	0	1	1	1	1	1	**Value of Addend = +31**
Sum Register:	1	0	1	1	0	0	**Value of Sum-Register Content = —19** **TRUE SUM = +44**
Two's-Complement Numbers:							
Carry-In Bits:	1	1	1	1	1	0	
Augend Register:	0	0	1	1	0	1	**Value of Augend = +13**
Addend Register:	0	1	1	1	1	1	**Value of Addend = +31**
Sum Register:	1	0	1	1	0	0	**Value of Sum-Register Content = —20** **TRUE SUM = +44**
Signed-Magnitude Numbers:							
Carry-In Bits:		0	1	1	1	0	
Augend Register:	1	0	1	1	0	1	**Value of Augend = —13**
Addend Register:	1	1	1	1	1	1	**Value of Addend = —31**
Sum Register:	1	1	1	1	0	0	**Value of Sum-Register Content = —28** **TRUE SUM = —44**
<u>Universal Characteristic of Overflow</u>: The *value* of the content of the sum register is *not* the true arithmetic sum of the Augend and the Addend.							

We can add the numbers manually, and then compare the true arithmetic sum with the value that appears in the sum register after applying the binary rules of addition to the binary representations of the augend and the addend. Thus, manually we can easily recognize when an overflow occurs. Manual detection of the presence of overflow is particularly easy for ***Ones'-Complement*** and ***Two's-Complement*** numbers. In both of these cases, an overflow occurs upon the addition of two numbers of same sign, if the content of the sum register is of the opposite sign. However, for the computer a criterion must be established for each form of representation that the computer's processor can use most easily to determine whether or not an overflow has occurred. For ***Non-Explicitly-Signed Numbers,*** the criterion used by the computer is the presence of a carry-out from the addition of the left-most bit. For **both** ***Ones'-Complement*** **and** ***Two's-Complement*** numbers, the computer processor recognizes the presence of an overflow when the carry-***in*** to the last bit differs from the carry-***out.*** For ***Signed-Magnitude Numbers,*** an overflow may be detected when the augend and addend are of identical sign of there is a carry-out from the **second** bit from the left, i.e., the leftmost magnitude bit.

Floating Point Number Representation

For mixed numbers (that is, numbers having both an integer component and a fractional component), there is an alternative to the fixed-point representation described above that allows for a greatly enhanced **range** of numbers to be represented. This is called ***Floating-Point Number Representation,*** and is based upon a scheme of number notation developed in the nineteenth and twentieth centuries and used by astronomers, astrophysicists, chemists, physicists, biologists, and other scientists to denote numbers either of very large or of very small size. One example of such a number is **Avogadro's Number,** which is the number of molecules of a chemical compound present in one mole or one gram-molecular weight of the compound, and is named after Amadeo Avogadro, a chemist who lived from 1776 to 1856. The value of Avogadro's Number is 6.02214199 * 10^{23}. This form of notation is much easier both to write and to check than is the integer representation of the number, which is 602,214,199,000,000,000,000,000. Avogadro's Number is an example of a very large number. Two examples of very small numbers are the mass of the electron, which is 9.10939 * 10^{-31} kilograms (.000000000000000000000000000000910939 kilograms), and the charge of the electron, which is —1.60217733 * 10^{-19} Coulombs (—.000000000000000000160217733 Coulombs). The basic form of notation employed in the ***Normalized Scientific Notation of Numbers*** is:

$$\pm D.fffffffffffff\ .\ .\ .\ *\ 10^{\pm eeee\ .\ .\ .}\ ;$$

where:

D consists of a single decimal digit other than zero (1..9),

ffffffffff . . . consists of a string of some number of fractional decimal digits (0..9), and

eeee . . . consists of some number of digits of integer exponent.

The number to the left of the multiplication sign, comprised of either a plus sign or a minus sign followed by a single integer digit and any number of fractional digits, is called the **(*normalized*) *mantissa,*** the number 10 immediately to the right of the multiplication sign is the ***radix,*** and the signed integer written as a superscript following the 10 is the ***exponent.***

Schematically, ***Scientific Notation of Numbers*** consists of:

(*Sign of Mantissa*) * (*Magnitude of Mantissa*) * (*Radix* = 10)$^{(\textit{Sign of the Exponent}) * (\textit{Magnitude of Exponent})}$

Note that the radix for the ***Scientific Notation of Numbers*** is **always** 10, and that despite the fact that the radix never varies from 10, nevertheless the well-entrenched custom is to specify the radix of 10 explicitly in the notation of the number. As we shall shortly see, this practice is in marked contrast to what is done in the computerized scheme of ***Floating-Point Number Representation.***

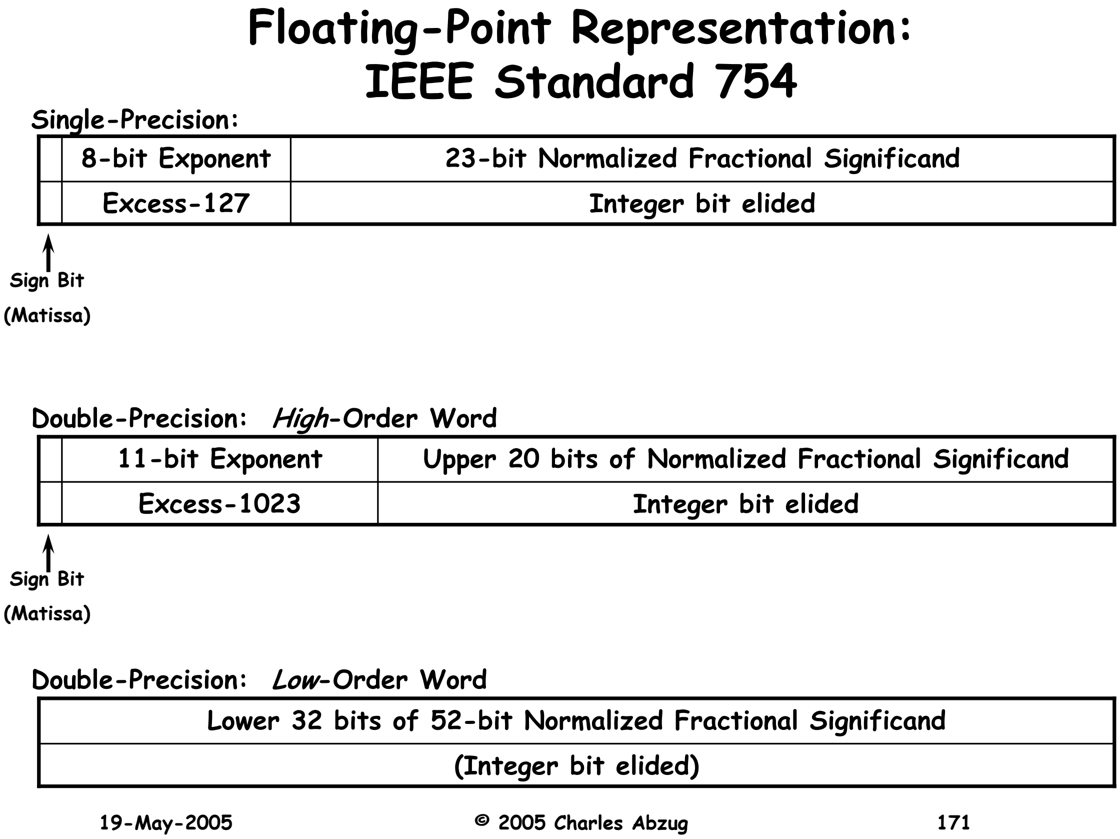
Conceptually, ***Floating-Point Number Representation*** is very similar to the ***Scientific Notation of Numbers.*** The latter consists of the adaptation of scientific number notation to the digital computer environment. Certain details of ***Floating-Point Representation*** are markedly different, however, from those of ***Scientific Notation.*** Most importantly, there is not just **one** form of ***Floating-Point Representation,*** but more than a dozen schemes have been employed in various computers. Overall, all schemes of ***Floating-Point Representation*** are similar, and consist of:

(*Sign Bit for Mantissa*) * (*Significand*) * (***Implicit*** *Radix*) $^{(\textit{Ssigned-Integer Exponent})}$

The most striking differences between ***Floating-Point Representation*** and ***Scientific Notation*** are in the radix. Instead of the explicitly-noted universal solitary scientific radix of 10, the ***Floating-Point*** radix can be **either** 2 or 8 or 16, and it is **never** explicitly noted, but rather is built-in to the electronic circuitry of the computer. The programmer must constantly bear in mind what radix is implied in a particular floating-point representational scheme, and must construct his/her programs accordingly. The sign of the mantissa is indicated via a specific bit. This is 0 if the mantissa is positive, and 1 if negative. The mantissa is recorded as a fixed-point (mixed) number. The floating-point number is almost always **normalized,** which means that the number is adjusted so that the left-most digit of the fixed-point mantissa is non-zero, with compensation for any leftward or rightward movement of the digits of the mantissa being accomplished by adjustment of the exponent. There is always a limitation on how many bits may be used to record the magnitude of the mantissa, and this results in a restriction in the precision of the recorded ***Floating-Point*** number. In some ***Floating-Point*** schemes the entire value of the mantissa must be recorded up to the limit of precision imposed by the number of bits available to record the mantissa. However, there is a small but nevertheless significant advantage gained by using radix 2. In a normalized ***Floating-Point*** number of radix 2, since the leftmost bit is **not** a zero, therefore it is not necessary to represent this bit explicitly, but instead it may be **elided,** that is, its value may be built-in to the processor hardware so that one additional bit is available for augmenting the recorded precision of the mantissa. Since it might not be possible to record all bits of the mantissa up to the level of precision desired, and since in any case the elided bit is not explicitly represented, therefore the field in the number representation that ex explicitly reserved for recording the value of the mantissa is referred to not as the **mantissa** field, but rather as the **significand.**

Regarding the exponent, we usually like to be able to make available a range of values of exponent which is at least approximately symmetrical with respect to positive and negative values. If *n* bits are available for the recording of the exponent, almost all ***Floating-Point Number*** schemes use either an **Excess-2^{n-1}** or an (**Excess-2^{n-1}**-1) representation of exponents.

In the past, each computer manufacturer decided for itself for implementation within its own product line how many schemes for ***Floating-Point Number Representation*** to use and which schemes to implement. Relatively recently, a task group was formed under the overall guidelines inherent in the method. More recently, the Institute for Electrical and Electronics Engineers (IEEE) promulgated Standard 754 for ***Floating-Point Number Representation.*** This actually consists of two standards in one package, one for use in representing 32-bit floating-point numbers, and the other for 64-bit: It will take several more years until all other, proprietary representations outlive their usefulness and are retired in favor of the new standard.



Floating-Point Representation: IEEE Standard 754

Single-Precision:

	8-bit Exponent	23-bit Normalized Fractional Significand
	Excess-127	Integer bit elided

Double-Precision: *High*-Order Word

	11-bit Exponent	Upper 20 bits of Normalized Fractional Significand
	Excess-1023	Integer bit elided

Double-Precision: *Low*-Order Word

Lower 32 bits of 52-bit Normalized Fractional Significand
(Integer bit elided)

19-May-2005 © 2005 Charles Abzug 171

References:

(1) BARON, ROBERT J; & HIGBIE, LEE (1992a). *Computer Architecture.* Reading, MA: Addison-Wesley Publishing Company, Inc. QA76.9.A73H52 1992; 004.2'2—dc20; 91-19130; ISBN 0-201-50923-7.

(2) BARON, ROBERT J; & HIGBIE, LEE (1992b). *Computer Architecture: Case Studies.* Reading, MA: Addison-Wesley Publishing Company, Inc. QA76.9.A73B3733 1992; 004.2'2—dc20; 91-25151; ISBN 0-201-55804-1.

(3) ERCEGOVAC, MILOŠ D.; & LANG, TOMÁS (2004). *Digital Arithmetic.* San Francisco, CA: Morgan Kaufmann Publishers. ISBN 1-55860-798-6.

(4) HENNESSY, JOHN L.; & PATTERSON, DAVID A. (1998). *Computer Organization and Design: The Hardware/Software Interface. Second Edition.* San Francisco, CA: Morgan Kaufmann Publishers, Inc. QA76.9.C643H46 1997; 004.2'2—dc20; 97-16050; ISBN 1-55860-428-6 (cloth), 1-55860-491-X (paper).

(5) MAXFIELD, CLIVE RICHARD (2003). *Bebop to the Boolean Boogie. An Unconventional Guide to Electronics Fundamentals, Components, and Processes. Second Edition.* Boston, MA: Newnes (Elsevier Science). TK7868.D5 M323 2002; 621.381—dc21; 2002038930; ISBN 0-7506-7543-8 (alk. paper).

(6) MAXFIELD, CLIVE RICHARD; & BROWN, ALVIN (1997). *Bebop Bytes Back: An Unconventional Guide to Computers.* Madison, AL: Doone Publications. 97-65591; ISBN 0-9651934-0-3.