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; revised 10 May 2002

15 Aug 1999; revised 10 May 2002

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. Thus, our legs might eventually become almost vestigial appendages, much like the appendix in our gastrointestinal tract.

We have already seen during my lifetime a substantial atrophy of some of the brain function of the modern college student. When I went to college, **every** student was able to carry out simple addition, subtraction, multiplication and division accurately and at a reasonable rate of speed. At that time, there were no electronic calculators. A professional accountant or bookkeeper, actuary, or mathematician might be fortunate enough to have access to a contrivance which would allow him/her merely to enter the numbers and would then take over the performance of the calculation. Such a contrivance typically weighed twenty-five pounds or more, took up a goodly portion of the space on a desktop, was operated either by mechanical power through the repetitive pulling of a lever over a distance of a foot or more and with a spring return, or by electrical power provided through the regular power line, just the same as it is provided to a refrigerator. The contrivance also made a lot of noise and was very slow. Unless we had a lengthy column of numbers, it was usually preferable 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 calculation 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 120-Volt or other standard power from the electric utility is not readily available, and produces reliable, accurate results, provided that the data are input correctly.

A major portion of the operations that take place in a digital computer consists of calculations that are carried out upon numeric data. These operations constitute digital arithmetic. The digital arithmetic operations must be augmented by a variety of support operations that are necessary to enable the calculations to occur. The vast majority of these calculations consist of simple arithmetic, and are conceptually very easy for even the layman to understand; they do not constitute a challenge to the

Computer Scientist. However, despite the fact that the digital arithmetic operations are **conceptually** simple, the detailed understanding of how these operations are carried out internally in the machine **is** something of a 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 as well as 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 important to study.

The overall subject matter can be broken down into two principal topics. First, it is necessary to understand the various principal ways in which numbers are represented in the digital computer. The student should be able to demonstrate this understanding by relating the internal representation of a number to the actual value of the number in each case. In addition, for each form of number representation, there are one or sometimes more ways in which it is possible for arithmetic operations to be carried out. The next challenge for the Computer Scientist is therefore to understand how each kind of arithmetic operation is carried out for each of the schemes of number representation. He/she must also understand what answer will be produced by the computer logic circuits in each case, as well as the potential for error occurring in the results. Finally, the Computer Scientist must also be able through a combination of hardware and software to detect and handle initially erroneous results, and to take appropriate action to assure that the final results attained under defined circumstances will be correct.

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 numbers, and shall then proceed to deal with various alternatives for the representation of numbers both in human societies and in digital computers. Next, we shall focus in on the representation of integer 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. We shall also consider the performance of arithmetic operations in digital computers upon numbers represented as integers (binary arithmetic). Finally, as was already mentioned, for floating-point numbers we shall cover only their representation and not the performance of arithmetic operations upon them..

Learning Objectives:

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

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

2. be thoroughly familiar with the concept of 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 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 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.
```

A relatively modern innovation (from about 1400 years ago in India) is the concept of zero, and a still more radical and much more recent innovation (late 18th to early 19th century) is the concept of negative numbers. A sampling of integer numbers as we know them today might therefore include a substantial range of negative as well as positive integers, as well as zero, such as, for example:

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

The performance of arithmetic operations upon integer decimal numbers is well understood by most laymen as well as Computer Science students and 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. 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 and 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,* for which the denominator is equal to one. Obviously, there are many more *rational numbers* than *integers.*

Real Numbers

The concept of *Real Numbers* includes many numbers that are not *rational,* 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), 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 only an approximation 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 it may have millions of decimal places, can ever express the value of π with absolute precision. 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. Irrational real numbers are generally represented in digital computers by approximation as rational numbers. 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 $\mathbf{a} + \mathbf{b}\boldsymbol{i}$ where **a** and **b** are both real numbers and $\boldsymbol{i} = \sqrt{(-1)}$, 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 representation of the two real coefficients **a** and **b.** Arithmetic operations upon them 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.* The concept is conveyed most easily through illustration. Consider the number `603,550`[a]. This integer number contains two zeroes and two fives. The two zeroes are not equivalent to each other, and the two fives are also not equivalent to each other. The leftmost zero indicates that the number contains no ten-thousands, while the rightmost zero indicates the absence of 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 not only the value but also the *position* of each numeral in the number determines its significance. In modern society, integers are almost universally written in a specific form of positional notation known as *decimal.* The word *decimal* is a derivative of *decem,* which is the Latin word for ten. The significance of each numeral in the number is directly related to how many numerals are between it and the righmost extremity of the number, that is, to its *position* in the number. Hence, the term *positional number representation*[b]. Taking our example number of `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. This has two consequences: first, there is a requirement for exactly ten distinct numerals to be able to represent all possible values for each position in the number, and hence to enable us represent all possible integer numbers in decimal notation. The decimal digits are:

```
0    1    2    3    4    5    6    7    8    9
```

With 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,

[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?

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 from the right has the place value of thousands (= 10^3), etc.

Generalized Positional Integer Notation

Once the concept of positional number notation is clearly grasped, there is very little limitation on the range of possible *radices* or *bases*. In fact, the base can be any integer greater than one. How we can write numbers in any radix can readily be grasped for radices less than ten. 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 of 0's and 1's, a radix 3 number would be composed entirely of 0's, 1's, and 2's. A radix 4 number would consist of 0's, 1's, 2's, and 3's, and likewise, we could have radix 5, radix 6, radix 7, radix 8, and radix 9 numbers.

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

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

The subscript `7` indicates that the number to the right of the equal sign is to be interpreted as *radix 7.* Because of the ubiquity of decimal numbers in our society, decimal numbers are usually written without any subscript, and there is a corresponding assumption that a number written without subscript is a decimal number. Therefore, the number to the left of the equal 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`

[c] 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
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`?

and `10111100`$_2$ are two numbers of radically different value, because the first is decimal and the second is binary. In fact, the binary number has a decimal value of only 188[d].

Positional Integer Notation for Radices Greater than Ten

The problem with numbers having radices higher than ten is that the numerals with which everyone is used to in our society extend only through nine, in consequence of the ubiquity of decimal numbers over almost all of the last few thousand years. With the advent of digital computers, within the field of Computer Science binary (radix 2), octal (radix 8), and hexadecimal (radix 16) numbers have also come into common use. For 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` serving only in radix 16 as the numeral representing a value of 15, `E` serving in both radices 15 and 16 as the numeral representing a value of 14, `D` as the numeral representing 13 in radices 14, 15, and 16, `C` representing 12 in radices 13 through 16, `B` representing 11 in radices 12 through 16, and `A` representing 10 in radices 11 through 16. Obviously, this scheme can readily be extended as far as radix 36, if necessary. Rarely are radices as large as 36 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 Be Represented in a Particular Positional Notation?

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 number of numerals, ***n,*** present in the 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 represented by two numerals, the numeral on the left can have any of ***r*** different values, and for each of these 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 to three, then r^3 different values are possible. By extension, for ***n*** numerals, r^n different numbers can be represented. If simple integers are being represented, then the range of numbers extends from 0 to [r^n - 1].

Radix	**Numerals**	**How many numbers can be represented?**	**Range**
r	*n*	r^n	0 $\leq$ (r^n - 1)
10	7	10^7 = 10,000 000	0 $\leq$ 9,999,999

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

2	12	$2^{12} = 4{,}096$	$0 \leq 4{,}095$
16	8	$16^{8} = 4{,}294{,}967{,}296$	$0 \leq 4{,}294\ 967{,}295$
7	9	$7^{9} = 40{,}353{,}607$	$0 \leq 40{,}353{,}606$

Radices in Use in Human Societies

While the most common radix in use in human societies is decimal, in consequence of the anatomical circumstance that the normal number of fingers is ten[e], nevertheless there are several other radices that have also been use 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, and that is still in use in parts of China. 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 and that 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.

[e] There is an hereditary abnormality in which some people have not 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 norm for people instead of five, then we probably would have settled on a base 12 number system. A base 12 number system would have been much more useful than base 10, because of the divisibility of 12 by 2, 3, 4, and 6.

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 $\texttt{2,122,220}_3$. The place values for radix 3 work out to:

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:

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:

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:

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:

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:

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:

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:

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 $2^{\mathbf{n}} - 1 = 255_{10}$):	`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 $[2^{\mathbf{n}-1} -1]$ 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 $2^{\mathbf{n}} - 1$ 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`[f].

[f] 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^{n-1} up to $[2^n - 1]$ 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^n rather than from $[2^n - 1]$. 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 $2^n = 256_{10}$ (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 $[2^{n-1} - 1]$ 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 $2^n - 1$, 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 zerowhich 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 $[-2^{n-1}]$, 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*[g].

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".

[g] 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.

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:

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
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:

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

For binary numbers, the identity becomes:

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

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:

$$\mathbf{2^{48} = 2^{8} \times 2^{40}} = 256 \text{ Tera.}$$

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

Binary Arithmetic for Integer and other Fixed-Point Numbers

Floating Point Number Representation

Bibliography